PID Controller Tuning with Deep Reinforcement Learning Policy Gradient Methods

Kasra Sinaei¹, Mohammad Reza Ha'iri Yazdi²

¹Center of Advanced Systems and Technology, University of Tehran, Tehran; kasra.sinaei@ut.ac.ir
²School of Mechanical Engineering, University of Tehran, Tehran; myazdi@ut.ac.ir

Abstract
In this paper challenge of tuning a PID controller for a single input single output (SISO) system has been overcome with couple of reinforcement learning agents which can automatically find the optimum values for controller parameters (Kp, Ki, Kd). First, a self-balancing robot with two coaxial wheels was simulated using the PyBullet physics library. Motors, IMU and Inertial Measurement Unit (IMU) were added via PyBullet features. Next, the robot’s Environment has been defined using the OpenAI GYM library. Both state space and action space of RL agents are continuous and ANN was used as function approximator in RL agents. For better computation speed and faster training, agents were implemented with Microsoft COAX, JAX, and Haiku since they have privileges of using GPU acceleration. Neural Network backpropagation is a computationally expensive operation and in case the forward pass of ANN gets more complicated than hardware capabilities it might cause problems for real-time simulation (step-simulation is possible for all cases). During the training agent’s properties recorded and plotted. Finally, we drew comparison between agents themselves and a manually tuned controller with the classic method. Even with the PID controller (not tuned and randomly adjusted), the system itself is still naturally unstable and the stability criteria (controller stability, pitch angle of torso, the center of mass linear or angular speed and etc.) should be considered in reward function for best possible results.

Keywords: Reinforcement Learning, Proximal Policy Optimization, Advantageous Actor-Critic, PID Controller, Self-balancing robot, Automatic tuning

1. Introduction
The classical linear Proportional–Integral–Derivative Controllers (PID) are widely used in industrial process control where over 90% of operating controllers are PIDs¹. A PID controller can perfectly control a plant without much complex computation and only requires perfectly tuned parameters to control the plant with an error from systems feedback. Since the proportional, integral, and derivative (PID) controller finds widespread use in the process industries, a great deal of effort has been directed at finding the best choices for the controller gain, integral, and derivative time constants for various process models (For instance Ziegler and Nichols, 1942; Cohen and Coon, 1953; Lopez et al., 1967; Smith et al., 1975; Rivera et al., 1986; Chien and Fruehauf, 1990; Tyreus and Luyben, 1992; Sung et al., 1995; Lee et al., 1996 have proposed their models)². In this paper we employ a novel method for tuning these parameters. Since 1980s Reinforcement Learning (RL) has attracted researchers interest with wide variety of disciplines. It RL could be applied to diversity of problems and if implemented correctly will yield to an optimum desired policy. As a result of that, it can manage to do myriad of tasks. For instance, in [3] an Advantageous Actor-Critic (A2C) has been used to design a system that naturally acquires control policies that are capable of performing balancing behaviors such as ankle push-offs for humanoid robots, without explicit human design of controllers, and in [4] a fuzzy controller has been designed with RL algorithms for obstacle avoidance.

Reinforcement learning is learning how to map situations to actions so as to maximize a numerical reward signal. The learner is not told which actions to take but instead must discover which actions yield the most reward by trying them.[5]

In this paper two different policy gradient method has been employed to fix the problem. Our observation space is a 2D vector with continuous values. Action space is 3D vector with continuous values. Both our action space and observation space consist of continuous elements. Considering that, we opt to use Artificial Neural Networks (ANN) in our state-value function (a function that maps observation space to a scalar value) as well as our policy function (a function that maps observation space to action space).

Here we have a naturally unstable system, a 2-wheeled robot that has to keep its balance with adjusting its motor torques. There is an MPU-6050 6-axis IMU attached to the main body of the robot (torso). Pitch orientation and linear velocity of torso comes from this sensor’s readings. Physical descriptions of the robot is presented in Table 1.

<table>
<thead>
<tr>
<th>Table 1. Physical Properties of Robot</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensions (m)</td>
</tr>
<tr>
<td>Torso</td>
</tr>
<tr>
<td>Wheel</td>
</tr>
</tbody>
</table>

Since the initial policy of the agent is not accurate and might not work properly training the robot in the real world will lead to damaging the robot physically so we
train the robot in a simulated environment considering all its dynamic features.

2. Simulation

PyBullet is an easy to use Python module for physics simulation for robotics, games, visual effects, and machine learning, with a focus on transferring the simulation to the real robot. In addition to the URDF file, it supports SDF (Spatial Data File), MJCF (Multi-Joint dynamics with Contact), and other types that we can load. PyBullet provides a variety of simulations e.g. forward and inverse dynamics, kinematics, collision detection, and also includes robotic examples such as a simulated quadruped, humanoid running using TensorFlow inference, and KUKA arms grasping an object.[6]

In order to simulate our robot in PyBullet, first, we draw a simplified model of it in SolidWorks, then using URDF exporter we get URDF file(s) and modify its physical properties. As shown in Figure 1 PyBullet loads our simplified robot and we can proceed to the next step. Both joints are velocity controlled (as we have in our real-world robot) and continuous. We can access the torso’s IMU data with PyBullets built-in commands.

![Figure 1. Simulated Robot in PyBullet](image)

Now that everything is set up, we can implement a GYM based environment compatible with all our reinforcement learning agents. Agents reward, action/observation space, and detailed algorithms are explained in the next section.

3. RL Agent Structure

Policy gradient methods work by computing an estimator of the policy gradient and plugging it into a stochastic gradient ascent algorithm. The most commonly used gradient estimator has the form

\[ \hat{g} \approx E_t [\nabla \log \pi(a_t | s_t) \hat{A}] \]  

where \( \pi \) is a stochastic policy and \( \hat{A} \) is an estimator of the advantage function at time step \( t \). Here, the expectation \( E_t \) indicates the empirical average over a finite batch of samples, in an algorithm that alternates between sampling and optimization[7].

Policy gradient methods unlike value-based methods (Q-learning for instance) learn a parameterized policy that can select actions without consulting a value function. A value function may still be used to learn the policy parameter but is not required for action selection[5].

Among Policy Gradient methods, Proximal Policy Optimization (PPO) and Advantageous Actor-Critic (A2C) has been used for tuning PID controller gains. The first agent, A2C, provides a fine-grained evaluation of each item in a prediction sequence. Together with an advantage function from the Actor, our A2C algorithm updates the model with a smooth gradient estimate and substantially reduces the training variance[8].

The pseudocode of Actor-Critic is shown in Table 2.

<table>
<thead>
<tr>
<th>Table 2. A2C algorithm with Eligibility Traces[5]</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input:</strong> ANN for policy function ( \pi(a</td>
</tr>
<tr>
<td>ANN for state-value function ( v(s, w) )</td>
</tr>
<tr>
<td><strong>Set algorithm parameters:</strong> ( \lambda, \alpha )</td>
</tr>
<tr>
<td>( z^w \leftarrow 0 ) (d-component eligibility trace vector)</td>
</tr>
<tr>
<td>( z^\theta \leftarrow 0 ) (d’-component eligibility trace vector)</td>
</tr>
<tr>
<td><strong>Loop forever (on each step time):</strong></td>
</tr>
<tr>
<td>( A \leftarrow \pi(.</td>
</tr>
<tr>
<td>Take action ( A ), observe ( S’, R )</td>
</tr>
<tr>
<td>( \delta \leftarrow R - \hat{R} + \hat{v}(S’, w) - \hat{v}(S, w) )</td>
</tr>
<tr>
<td>( \hat{R} \leftarrow \hat{R} + \alpha \delta )</td>
</tr>
<tr>
<td>Update ( w, \theta ) (based on ( \delta z^\theta, \delta z^w ))</td>
</tr>
<tr>
<td>( S \leftarrow S’ )</td>
</tr>
</tbody>
</table>

The second agent, PPO, is an off-policy algorithm (in off-policy methods, target policy and behavior policy are not identical and the agent uses observations of actions chosen with behavior policy to update its target policy) that mitigates the problem of overestimation, leaving underestimation uncorrected. This alleviation is achieved by effectively clipping the probability ratio in a specific way.

\[ J(\theta; s, a) = \min(\rho_s(s, a)A(s, a), \bar{\rho}_s(s, a)A(s, a)) \]  

(2)

clipped probability ratio could be defined as:

\[ \rho_s(s, a) = clip(\rho_s(s, a), 1 - \epsilon, 1 + \epsilon) \]  

(3)

The clipped estimate removes both overestimation and underestimation. Taking the minimal value between the unclipped and clipped estimates ensures that we don’t correct for underestimation. One reason to do this is that underestimation is harmless, but a more important reason is that it provides a path towards higher values of the expected advantage.

In section 1 we mentioned that both action space and observation space are continuous now we explain each of them specifically so that we can define the reward function.

- Observation space consists of two elements: torso’s pitch angle (robot tilt angle) and torso’s linear velocity magnitude.
* Action Space consists of 3 elements: Proportional gain ($K_p$), Integrator gain ($K_i$), and Derivative gain ($K_d$). The orders of these 3 coefficients are quite different and before the training process, they need to be normalized. Based on previous knowledge we can heuristically guess their order of magnitude and after normalization, both 3 values will be bounded in [0,1].

$$\text{Action} = [a_0, a_1, a_2]$$  \hspace{1cm} (4)

$$K_p = a_0 \times 1000$$  \hspace{1cm} (5)

$$K_i = a_1 \times 100$$  \hspace{1cm} (6)

$$K_d = a_2 \times 0.1$$  \hspace{1cm} (7)

In equations (4,5,6,7) controller parameters have been related to agents’ normalized actions. This will affect agents learning process considerably.

Network architecture of policy function approximator is another important characteristic of both agents. (In A2C both critic and actor require proper architecture for their neural networks, similar to PPO state-value and policy function).

Neural network architecture of policy function has been depicted in Figure 2, with input layer of $\mathbb{R}^3$, two 20 dimensional hidden layers, another 10-dimensional hidden layer, and finally 3-dimensional output layer as illustrated in Figure 2.

![Figure 2. Neural Network architecture of Policy function](image)

For the state value function simpler network has been used (two 20-dimensional hidden layers) as shown in Figure 3.

![Figure 3. Neural Network architecture of state-value function](image)

As shown in the above figures no convolutional layers are necessary. In the policy function after each hidden layer a Relu activation function has been used and at the end, a sigmoid activation function was used to generate the final outputs. These architectures were chosen experimentally with trial and error.

For faster computation, JAX has been used instead of Numpy, since it has the advantage of GPU acceleration. Agent’s Neural Networks had to be implemented as a haiku function so that we can implement agents in Microsoft COAX style.

The most important function that need to be defined in our environment is the reward function; it affects the result of training significantly. If the agent itself was going to balance a 2 wheeled robot, including the torso’s orientation and angular velocity would yield to our desired policy but here in this problem, it is our controller that needs to be optimized. Robustness of the controller could be analyzed with some parameters such as overshoot, rise time, steady-state error. These parameters could be measured by exerting an external force to the system. **Rise time and steady state error** have been included in our reward function at the end of each episode as well as derivatives of the torso’s pitch angular velocity and linear velocity. The reward function is explained in equation (8). First term is calculated at each step of episode and the second term will be calculated once and at the end of episode.

$$\text{reward} = -\left(w_1 (\dot{\theta} - \dot{\theta}_{\text{set}}) + w_2 (\dot{x} - \dot{x}_{\text{set}})\right)$$

$$\text{reward} = -\left(w_3 \dot{f}_{\text{steer}} + w_4 \dot{\theta}_{\text{overhead}}\right)$$  \hspace{1cm} (8)

$\dot{\theta}_{\text{set}}$ and $\dot{x}_{\text{set}}$ should be set to zero in order to keep robot stable and steady. Weights, $w_1$, $w_2$, $w_3$, $w_4$ are proportionate to the random disturbance exerted to robot during each episode. Since the exerted disturbance is random it is necessary to exclude this probabilistic factor while evaluating two agents’ performance.

Adding $\theta$ to our reward function will not affect the agent’s learning process constructively, since it cannot explain anything about the controllers performance solely.

In order to update agents’ policy properly, we need to set learning parameters efficiently so that the agent becomes farsighted. We used n-step return methods with a discount factor of 0.995. We want n to be as high as possible, on the other hand, we cannot increase n too much because of high variance problem. Considering all the limitations, n = 50 steps have been chosen for temporal difference (TD) error calculation.

**4. Classic Method**

As mentioned in introduction there are several ways to tune a PID controller. First we derive transfer function from our 2-wheeled robot dynamic equations of motion. Here we just mention the final results for more details refer to [9].
\[
\Phi(s) = \frac{ml}{s} \frac{q}{q} + \frac{b(l + ml^2)}{s^2} \frac{q}{q} - \frac{(M + m)g}{s} \frac{q}{q} - \frac{bh}{s} \frac{q}{q}
\]

In equation (9) M is mass of wheels, m is mass of torso, b is viscous friction coefficient, I is Inertia moment of torso, and l is half of the torso’s height.

\[
q = [(M + m)(l + ml^2) - (ml)^2]
\]

In Simulink a simple close loop feedback system with a PID controller has been designed. The block diagram of system is shown in Figure 4.

In frequency domain we would like to achieve a system with a closed loop stability such that it has a bandwidth of 5 rad/sec and phase margin of 60°. With help of Matlab we tune PID gains manually. These values are going to be used in section 5 to compare Ziegler Nichols with deep RL gains.

![Figure 4. Block Diagram of simulated model in Matlab](image)

5. Results and Discussion

After 30,000 steps of training, both agents’ policies converged to their optimum value. (50 episodes of 600 steps) For better exploration at the beginning of each episode robot was spawned in the world with a random pitch angle. In addition to that, a random disturbance with magnitude in range (-2N, +2N) was applied to the robot at the 200th step. A2C and PPO learning progress during the first episode has been recorded (reward per step, tilt angle, and magnitude of disturbance during training) and has been illustrated in Figure 8. Fluctuations in reward function curve of Figure 8 are not related to robot’s stability directly; if there is a sudden rise it means the agents recent action (new PID gains) have reached better results and consequently it has achieved higher reward.

Since the physical properties of the robot were unchanged the variance of optimum action value in different states (state-space was a depiction of torso’s pitch angle and linear velocity magnitude) should not be significant; however, for more accurate reporting we averaged the final optimum action values to obtain best possible controller gains. Table 3 contains these averaged values for both agents’ optimum action averaged over all states as well as results of manual PID tuning with Ziegler and Nichols method.

The disturbance exerted to the system was equivalence of 50% of robot’s total mass. Unlike methods that use RL agent as the main controller of the robot, we can see that even in the first episode robot can stay upright (although it is wobbly).

To make sure that the controller gains are converging to their optimum, value has been observed over 50 episodes.

![Figure 5. PPO calculated gain convergence during learning](image)

<table>
<thead>
<tr>
<th>Table 3. Controller Gains</th>
<th>Method</th>
<th>k_p</th>
<th>k_i</th>
<th>k_d</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPO</td>
<td>650.93</td>
<td>0.058</td>
<td>57.29</td>
<td></td>
</tr>
<tr>
<td>A2C</td>
<td>456.02</td>
<td>0.050</td>
<td>53.60</td>
<td></td>
</tr>
<tr>
<td>Z &amp; N</td>
<td>520.47</td>
<td>0.018</td>
<td>21.79</td>
<td></td>
</tr>
</tbody>
</table>

Now that we have 3 controllers we can also observe their response to a 5N impulse at t = 1s. Robot has been tested with PPO gains, and Ziegler & Nichols gains. Their response has been shown in Figure 6 and Figure 7.

![Figure 6. PPO respond to an impulse](image)

![Figure 7. Ziegler and Nichols respond to an impulse](image)
6. Conclusions and future works

This work has investigated the application of deep RL in automatic PID controller tuning. An A2C agent and a PPO agent have been trained in a simulated environment using PyBullet physics library. Random tilt angle before episode initialization and random disturbances applied to the robot during each episode provided better exploration and made evaluation of controller’s performance possible (All criteria should be interpreted as rewards).

After 500 episodes, each containing 600 steps, an optimum policy has been achieved with both agents. In order to compare agents, an impulse with magnitude of 5N (equivalent to half of the robot’s mass) has been applied to it. Based on the response time and overshoot of the controlled system we can say that PPO did better tuning compared to A2C. For evaluating RL algorithms learning curve which illustrates reward per episode during training is another reliable tool. In Figure 9 we can compare learning curves of PPO and A2C. Both agents were trained in the same environment with similar reward functions, and learning parameters (n-step, learning rate and neural network architecture).

This method could be more helpful with other complicated robots. In [10] stability criteria of a wheeled bipedal robot have been defined based on the position of all link’s COM relative to the shared axis of wheels. Figure 10 Shows the structure of a wheeled bipedal robot.

In this case, a PID controller can do the balancing, but since the height of the robot changes during squat, performance of the PID controller will be affected by this change of height; in addition to that, unlike investigated robot in this paper, system becomes non-linear. With use of RL we can add robot’s height (or robot’s COM height) to our state-space and the agent can estimate the best possible PID gains for any height of the robot.

Using a quadratic coding for policy function instead of ANN could yield to better results with simpler computation. In order to do that we need to form a quadratic function in form of our PID controller and for reward function we can only consider robot’s stability. If we do so, there will be no need to measure steady-state error, rise-time and maximum overshoot at the end of each run for assigning reward to each action.

![Figure 7: Learning process during a single episode for a) A2C agent and b) PPO agent](image)

![Figure 9: Sum of reward per episode a) A2C, b) PPO](image)

![Figure 10: SR-600 Bipedal wheeled robot](image)
7. References


