Stabilising and Navigating Sloped Terrain with a Two-wheeled Robot through Deep Reinforcement Learning Techniques

Raajith Gadam
Department of MAGE
University of Maryland,
College Park,MD,USA
raajithg@umd.edu

Amogh Wyawahare
Department of MAGE
University of Maryland,
College Park,MD,USA
amogh@umd.edu

***Abstract*—There have been attempts to address the performance consistency and model training time issues with self balancing robot using a variety of deep learning approaches. This report describes how DQN and SAC were used to overcome this issue by hyper parameter tuning. It validates the outcomes and compares which model is ideal for a certain specific scenario such as slope or wedge maneuvering. The implementation is being done in Python and tested in Pybullet. It is evident from the implementation results that the DQN algorithm performs the task and trains faster than SAC model. Hence, deep reinforcement learning techniques can significantly enhance the autonomous navigation and stability of TWRs on uneven terrains**

***Index Terms*—Performance, Model training, DQN,SAC, autonomous navigation**

I. Introduction

A. Problem Statement and Motivation

Balancing a Two-Wheel Robot (TWR) is a well-known challenge in robotics with two wheels on either side of its body. Many researchers have worked on creating control systems to keep the robot upright and balanced. While it sounds simple, balancing a TWR requires a complex control system that can handle different factors like the robot's movements, the surroundings, and unexpected disturbances. Using reinforcement learning to train a TWR to balance itself has become a popular research area.The goal of our project is to train a TWR to balance itself on uneven terrain using reinforcement learning. This approach could provide a more adaptable and robust solution compared to traditional methods like Linear Quadratic Regulator (LQR) and Proportional-Integral-Derivative (PID) control. These conventional methods often struggle with changing conditions and disturbances, making them less effective in real-world scenarios. Our motivation is to develop a control strategy that can handle varying environments and disturbances, which is important for practical applications like transportation and automation.We aim to train a TWR to navigate uneven terrain, such as slopes, without human help.

Using PyBullet for simulation, we simulate the TWR in a 3D space. The robot will balance itself by moving forward and backward, adjusting its center of gravity, and controlling the speed of its wheels. The main task is to move from one point to another while keeping balance.Our project uses DQN and Soft Actor-Critic (SAC) reinforcement learning algorithms to train a TWR to balance and navigate slopes. These methods help the TWR learn the best ways to balance and move by trying different actions and learning from the results, providing a promising solution to this challenging problem.

B. Existing Approaches

There are three main ways to control TWRs: linear controllers, non-linear controllers, and machine learning methods.

Linear Controllers: Linear controllers like LQR and PID have been widely used. LQR uses a feedback control method to minimize a cost function based on a linear system model. However, LQR may not handle the TWR's non-linear behaviors well. PID controllers calculate an error signal based on the difference between the desired and actual angle of the TWR, then adjust the motor outputs. PID controllers are fast and work in real-time, but tuning them can be difficult, and they may not adapt well to changing conditions.

Non-Linear Controllers: Non-linear controllers like Fuzzy-PID use fuzzy logic to manage non-linear systems. Fuzzy-PID controllers can handle complex behaviors but may take longer to learn and are harder to design than traditional PIDs. Control Moment Gyroscopes (CMGs) use angular momentum to create large torques that balance the TWR. However, CMGs are heavy, need precise control, and have limited motion range.

Machine Learning Techniques: Machine learning methods, especially reinforcement learning, offer a different approach. Q-Learning adjusts motors based on the angle and speed of the TWR to keep it balanced. This method can be limited because it relies on predefined states and actions. Deep Learning techniques, like Deep Q Networks (DQN), learn

complex relationships between inputs and outputs. DQNs adapt to changing conditions and work well in unstructured environments, but they need a lot of computing power and data.

Our project uses DQN and Soft Actor-Critic (SAC) reinforcement learning algorithms to train a TWR to balance and navigate slopes. These methods help the TWR learn the best ways to balance and move by trying different actions and learning from the results, providing a promising solution to this challenging problem.

II. Literature review

Two-wheel self-balancing robots have gained a lot of interest recently, especially with the introduction of commercial products like the Segway. These robots are designed to maintain balance by adjusting their position to avoid falling. Unlike three or four-wheeled robots, two-wheeled robots are more maneuverable and can access tight spaces.

A. Key Developments in Two-Wheel Self-Balancing Robots:

1. JOE by Felix Grasser: Felix Grasser and his team at the Swiss Federal Institute of Technology developed a two-wheel prototype named JOE. This robot balances its driver on two coaxial wheels, each connected to a DC motor. The system uses a linear state-space controller, incorporating data from a gyroscope and motor encoders to maintain balance and perform stationary U-turns.link

2. SEGWAY PT by Dean L. Kamen: Dean Kamen invented the SEGWAY PT, a self-balancing human transporter. This robot uses five gyroscopes and other sensors to stay upright. It allows users to navigate small steps and various terrains by shifting their body weight. The SEGWAY PT has achieved significant commercial success since its launch.link

3. nBot by David P. Anderson: David Anderson developed nBot, a two-wheeled robot that uses inertial sensors and motor encoder data to balance. The concept is simple: drive the wheels in the direction of the robot's tilt to keep it upright. nBot uses two feedback sensors—a tilt sensor and wheel encoders—to measure the robot's tilt and base position.link

4. Eyebot by Rich Chi Ooi: Rich Chi Ooi, a student at the University of Western Australia, created Eyebot, an autonomous two-wheeled robot. The project used a Kalman Filter for sensor fusion, integrating data from gyroscopes and accelerometers to accurately estimate tilt angles. Eyebot employed Linear Quadratic Regulator (LQR) and PID controllers for balance and trajectory control.link

B. Control Strategies for Two-Wheel Robots

1. Linear Controllers: PID Controllers: PID control is a common method for balancing robots, using proportional, integral, and derivative gains to correct errors. It is straightforward and efficient for real-time control but can be challenging to tune and less adaptable to changing conditions. LQR Controllers: Linear Quadratic Regulator (LQR) control uses a state-space approach to minimize a cost function, offering robust performance. LQR is effective for controlling balance and motion but requires precise modeling and tuning.

2. Non-Linear Controllers: Fuzzy-PID Controllers:These combine fuzzy logic with traditional PID control, allowing for better handling of non-linear systems like two-wheel robots. However, they may converge slower and are more complex to design. Sliding Mode Control:This method provides robust performance under uncertainties and disturbances, making it suitable for balancing tasks.

3. Machine Learning Techniques: Reinforcement Learning (RL): RL methods like Q-Learning adjust the robot's actions based on rewards and penalties. RL can adapt to various conditions but often requires extensive data and computational resources. Deep Learning (DL): Techniques such as Deep Q Networks (DQN) use neural networks to approximate the Q function, allowing for more complex and adaptable control strategies. DL approaches can handle continuous states and are more robust in unstructured environments but are computationally intensive.

C. Experimental Comparisons and Results

Research has shown that LQR controllers generally perform better than PID controllers in terms of robustness and response speed. For instance, experiments conducted on two-wheeled robots demonstrated that LQR controllers had smaller overshoot and faster response times compared to PID controllers. Additionally, advanced algorithms like those using genetic optimization for tuning LQR controllers have shown improved stability and performance.

Two-wheel self-balancing robots have practical applications in transportation, automation, and assistance for the elderly or disabled. Future research will likely focus on enhancing control algorithms, improving robustness, and exploring new applications. Advanced machine learning techniques will continue to play a crucial role in developing more adaptive and efficient control systems. Two-wheel self-balancing robots present a significant challenge in robotics due to their unstable nature. While traditional control methods like PID and LQR have been effective, newer approaches using machine learning offer promising improvements. Continued research and development in this field aim to create more robust, adaptable, and practical self-balancing robots for various applications.

III. Methodology

For our project, we explored using two different algorithms to train the Two-Wheel Robot (TWR) to balance and navigate. The algorithms range from single-agent to multi-agent setups, single and multiple actions per policy, centralized and decentralized policies, and continuous action spaces. We normalized all state observations between 0 to 1 to ensure all features have a similar scale, reducing variance and preventing numerical instability.

A. Deep Q Network (DQN)

DQN uses a replay buffer to train the TWR. The agent controls both wheels by selecting actions that change the wheels' angular velocities simultaneously. The goal is to maximize the cumulative reward, and the Q function approximates the expected rewards for each state-action pair. The equation for the target action-value function is:

$$Q(s, a) = r + \gamma \max_{a'} Q(s', a')$$

where r is the immediate reward, γ is the discount factor, s' is the next state, and a' is the action that maximizes the action-value function in the next state. The temporal difference error is given by:

$$\delta = Q(s, a) - (r + \gamma \max_{a'} Q(s', a'))$$

During training, we sample batches from the replay memory, compute Q-values, and optimize the model using Huber Loss. The target network is updated slowly using a soft update to provide stable targets for the policy network.

B. Soft Actor-Critic (SAC)

SAC maximizes policy entropy and expected reward. It encourages exploration by penalizing deterministic policies. The objective function for SAC is:

$$J_v(\psi) = \mathbb{E}_s[0.5(V_\psi(s_t) - \mathbb{E}_a[Q_\theta(s_t, a_t) - \log \pi_\theta(a_t|s_t)])^2]$$

SAC uses a Gaussian policy to output continuous actions and employs a soft Q-learning approach to estimate expected rewards.

C. RL Framework

We used the following setups for the TWR:

1. State Space: The state space is continuous and includes the TWR's position, orientation, linear and angular velocities, and wheel angular velocities. The total state space is represented by a vector of size 40.

2. Action Space: The action space includes changes in the wheels' angular velocities. For single-action models, the action space is discredited to 9 values per wheel. For multi-action models, there are 81 action permutations. For SAC, the action space is continuous.

3. Reward Structure: The reward structure includes penalties for falling and crossing boundaries, rewards for reaching goals, and shaped rewards for progress. The shaped rewards encourage survival, reducing movement for stability, and moving towards the goal.

4. Termination Conditions: The episode ends when the agent reaches the goal, falls, or exceeds the maximum number of steps.

D. Environment and Training Process

We used a CAD model of the TWR and simulated it in PyBullet using OpenGL. The environment included state observations, wheel control, and terrain settings. The time step was set to 0.05s (20 Hz) to allow enough time for wheel adjustments. The models were trained on CPUs due to hardware limitations, with a maximum of 10,000 episodes per model. Hyperparameters included a learning rate of 0.0001, discount factor of 0.99, and batch size of 5 for DQN. Neural networks had 3 hidden layers of size 128.

We logged performance metrics such as trajectories, time steps, goal achievement percentage, rewards, and displacements. This data helped evaluate and fine-tune the models.

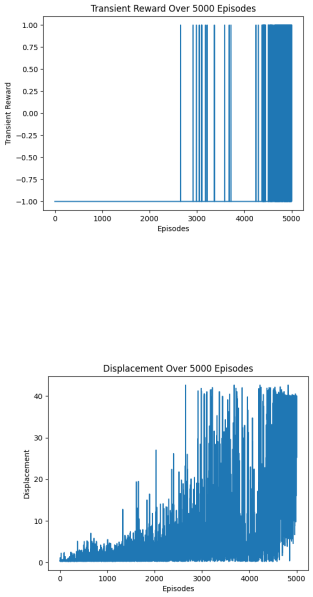

Fig. 1. Performance metric: Transient reward and Displacement

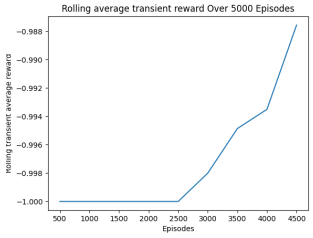

Fig. 2. Performance metric:Rolling average transient reward

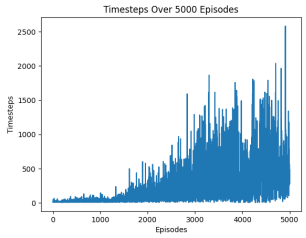

Fig. 3. Performance metric:Timestep

IV. Contributions

Since the objective of out project is to employ DQN and SAC algorithm and to compare the results,leaving building a robot model for simulation out of the scope of this project,we have utilised an existing urdf file from the github [10] for testing purposes.Additionally the pipeline was also adopted from [10] and the technique was formulated using [1] and acts as base for both the algorithms in terms of hyper parameter tuning.We further added layers on top for the existing algorithm to accommodate extra parameters like epsilon greedy and introduced extra working hyper parameter learning rate actor which was not originally not functioning in the [10].After training the model intensively,it was finally able to train successfully and was able give training accuracy of 80% which was not standard with the original algorithm and also upon tuning the hyper parameters and testing multiple different scenarios with different distance,the model was able to perform satisfactorily.

A. Our Specific Contributions

- We enhanced the action space for the Deep Q Network (DQN) algorithm by incorporating new actions specifically designed for navigating uphill and downhill slopes. This allows the robot to handle more complex terrain effectively.
- We developed a new simulation environment by introducing a slope to represent complex terrain. This enabled the robot to be trained and tested in scenarios that more closely resemble real-world conditions.
- We implemented a continuous action space for the Soft Actor-Critic (SAC) algorithm. This allowed for more precise control of the robot's movements. For example, actions were defined as continuous changes in the wheels' angular velocities, providing smoother and more adaptive responses to the terrain.
- We modified the state variables to include information about the Z-axis, which is crucial for accurately traversing slopes. This adjustment ensures the robot can maintain balance and stability while navigating uneven terrains.
- We refined the reward structure to provide more granular feedback to the learning algorithms. This included rewards for maintaining balance, penalties for falling, and additional incentives for successfully navigating slopes. This shaping of rewards helped accelerate the learning process and improve overall performance.
- We experimented with different epsilon decay strategies in the DQN algorithm to balance exploration and exploitation. This included linear, exponential, and inverted exponential decay schedules, allowing us to optimize the learning efficiency and stability of the algorithm.
- We conducted extensive hyperparameter tuning for both DQN and SAC algorithms, including learning rates, discount factors, and batch sizes. This systematic optimization was critical in achieving better performance and faster convergence in training.

V. Results

In our project, we focused on stabilizing and navigating a two-wheeled robot (TWR) using two deep reinforcement learning algorithms. We implemented and tested 2 different algorithms: Deep Q Network (DQN) and Soft Actor Critic (SAC). The robot was trained to balance, move to a specified goal, and traverse slopes.

A. Task 1: Learning to Balance

The goal was to train the TWR to balance for 400 time steps (20 seconds). The TWR was trained for 3000 episodes with an inverted exponential epsilon schedule. The DQN and SAC models were successful, reaching the balance goal.

B. Task 2: Learning to Balance and Move to a Goal Line at 30m

The objective was to balance the TWR and move it to a goal line 30 meters away. DQN was the only model that consistently reached the 30m goal after 3000 episodes. SAC managed to move forward but fell short of the goal.

C. Task 3: Learning to Balance and Traverse a Slope to a Goal Line at 40m

In this task, the TWR had to balance and move over a slope to reach a goal line at 40 meters. DQN and SAC models were successful, navigating the slope and reaching the goal line after 10,000 episodes. The average velocities indicated that DQN achieved faster goal-reaching times compared to SAC.

D. Hyper parameter Tuning

We conducted hyper parameter tuning for epsilon schedules and learning rates to optimize the DQN model. The linear decay epsilon schedule yielded the best results, with 40% goal achievement in the last 500 episodes. Inverted exponential decay reached 30%, and stretched exponential decay showed 70% goal achievement between 3000 to 3500 episodes, later converging to a sub optimal policy.

E. Learning Rate Optimization

For learning rate optimization, a learning rate of 0.0001 enabled the TWR to reach the goal effectively. Higher learning rates caused oscillation and divergence, while lower rates resulted in slow convergence and suboptimal policies.

Advantages and Limitations DQN outperformed SAC due to its simplicity and ease of implementation. SAC,

though more complex, offered continuous action space advantages but required extensive tuning and more training data. The large action space SAC hindered learning efficiency due to sparse rewards.

Observations and Improvements We observed that the TWR's performance was sensitive to time step lengths and the magnitude of angular velocity changes. Proper tuning of these parameters was critical for stable performance. Additionally, the TWR's deviation from straight paths was attributed to contact friction and wheel weight distribution. Increasing the friction coefficient and ensuring consistent wheel contact could mitigate these issues. Setting appropriate termination conditions and using curriculum learning could further improve the training process.

The DQN model demonstrated the best performance across tasks, with efficient training and robust results. Future work includes adding proximity sensors for better environmental awareness, implementing distributed learning, and testing the models in real-world scenarios.

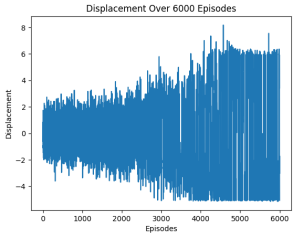

Fig. 4. SAC displacment over 6000 epsiodes

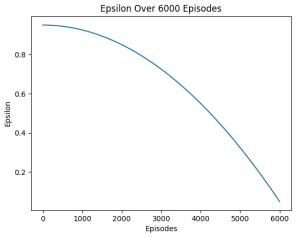

Fig. 5. SAC epsilon over 6000

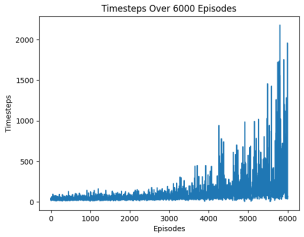

Fig. 6. SAC Timestep

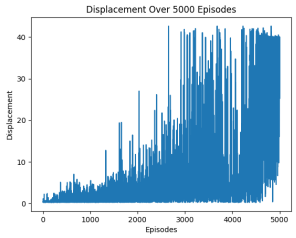

Fig. 7. DQN displacement over 5000 episodes

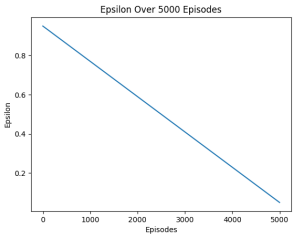

Fig. 8. DQN epsilon over 5000 episodes

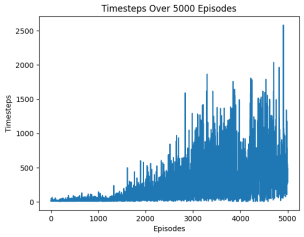

Fig. 9. DQN timesteps over 5000 episodes

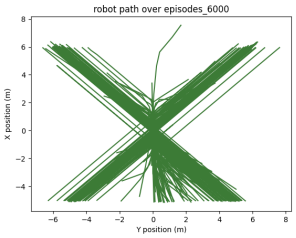

Fig. 10. Robot trajectory under SAC

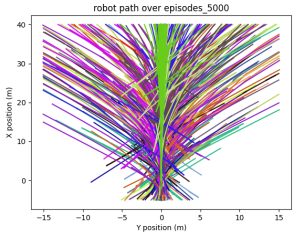

Fig. 11. Robot trajectory under DQN

Now, from the above figures, intuitively, the difference in the paths generated by both the algorithms is very clear. But after running multiple iterations, we confirmed visually for all the test cases and found that DQN is naturally more robust for this application than SAC.From Fig 8 you can see that the epsilon is a linear function and helps the model to train consistently throughout.

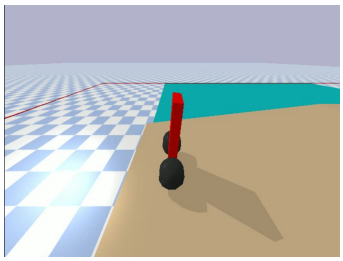

Fig. 12. Results:DQN testrun for 40m

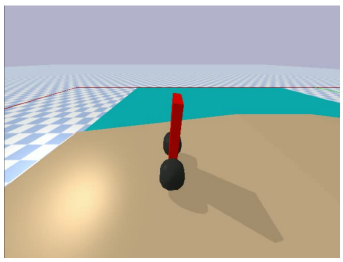

Fig. 13. Results:DQN testrun for 40m

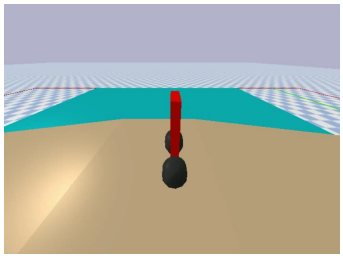

Fig. 14. Results:DQN testrun for 40m

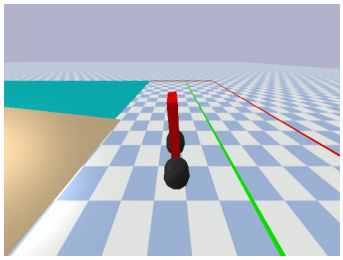

Fig. 15. Results:DQN testrun for 40m

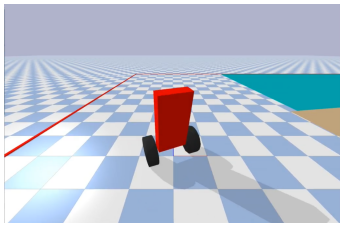

Fig. 16. Results:SAC testrun for 40m

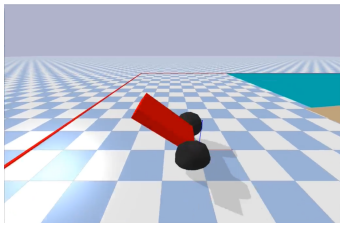

Fig. 17. Results:SAC testrun for 40m

It is evident from the above Fig 15 that the DQN model was able to navigate over the sloped terrain once the training accuracy reached over 80%. However, the SAC test runs failed continuously and failed to perform the task. Also from Fig 11, we can observe that the robot trajectory for the DQN model was reaching equilibrium which is depicted by the green lines at the middle whereas the Fig 10 robot trajectory of the SAC model failed to stabilise for more than one scenario. This explains the higher performance of DQN model over SAC model.

VI. Conclusion

Thus, we were able to achieve our goals for the project which involved implementing self-balancing TWR to maneuver slope obstacles using sophisticated Deep Reinforcement Learning techniques namely DQN and SAC. With the robot now able to navigate sloped terrains autonomously, we have laid a robust foundation for future enhancements and potential real-world applications. Our continued efforts will focus on optimising the algorithms, extending the testing scenarios, and preparing the robot for practical deployment.

A. Challenges faced

Fine-tuning the SAC algorithm to handle the robot's continuous action space presented significant challenges. Balancing exploration and exploitation to optimize the learning process was particularly difficult. The complexity of the SAC model required meticulous adjustments to hyper parameters such as epsilon and learning rate. Enhancing the DQN algorithm to adapt to abrupt changes in terrain required careful adjustments to the reward structure and state representation. The sparse reward structure hindered the agent's learning efficiency, necessitating the implementation of shaped rewards to provide more nuanced feedback signals. Training deep reinforcement learning models, particularly the DQN algorithm, was computationally intensive and time consuming. The limited availability of high-performance computational resources meant

that training times were prolonged. For instance, training the SAC model for 10,000 episodes took approximately 5 hours, whereas the DQN model took 30 minutes. Due to hardware limitations, all models were trained on CPU rather than GPU, which significantly slowed down the training process. Implementing multiprocessing on CPUs helped, but the overall computational speed remained a bottleneck.

REFERENCES

[1] https://github.com/Jason-CKY/DeepRL-pytorch

[2] https://github.com/wongwsvincent/Self-Balancing-Robot

[3] https://www.youtube.com/watch?v=0UuO2Fuiow4.

[4] F. Ünker, "Oscillation Control of Two-Wheeled Robot using a Gyrostabilizer," Gazi University Journal of Science Part C: Design and Technology, vol. 10, no. 3, pp. 547–557, Sep. 2022, doi: https://doi.org/10.29109/gujsc.1062497. 11

[5] W.-F. Kao, C.-F. Hsu, and T.-T. Lee, 'Intelligent control for a dynamically stable two-wheel mobile manipulator', in 2017 Joint 17th World Congress of International Fuzzy Systems Association and 9th International Conference on Soft Computing and Intelligent Systems (IFSA-SCIS), 2017, pp. 1–5.

[6] pranz24, "Pranz24/PyTorch-Soft-actor-critic: PyTorch Implementation of Soft actor critic," GitHub. [Online]. Available: https://github.com/pranz24/PyTorch-soft-actor-critic. [Accessed: 21-Apr-2023].

[7] T. Haarnoja et al., 'Soft Actor-Critic Algorithms and Applications', ArXiv, vol. abs/1812.05905, 2018

[8] Wei An, Yangmin Li. Simulation and Control of a Two-wheeled Self-balancing Robot. International Conference on Robotics and Biomimetics, 2013,pp. 456-461

[9] Chen Haiyun, Du Zhenhua, Zou Ningbo, Shi Mingjiang, "LQR Controller Design for Inverted Pendulum Based on Multi-population Genetic Algorithm". Control Engineering of China, vol. 21, pp.391-394, May 2014

[10] https://github.com/ngzhili/Two-Wheel-Robot-DeepRL?tab=readme-ov-file