# Reinforcement Learning for Path Following

Juraj Mičko*[1] and Kwot Sin Lee*[1] and Wilson Suen*[1]

***Abstract*— Consider the task of a problem attempting to follow a path in a constrained environment with only a few lines to follow. We attempt this using end-to-end reinforcement learning and explore two algorithms for doing so: Deep Deterministic Policy Gradients (DDPG) and Proximal Policy Optimisation (PPO). We further explore different problem formulations to learn a path-following controller or the velocities of the agent directly, and report our findings.**

## I. Introduction

### A. Problem Description

Our main task is to train end-to-end reinforcement learning algorithms an agent (the robot) to follow a path. Here, we assume the path is arbitrarily created and thus we focus on having the agent follow the given path as closely as possible. To do this, we adopt two different approaches:

1) Learn the velocities of the agent to navigate the environment and follow the path directly.
2) Learn the hyperparameters of a controller that provides input to the agent's motions to follow a path.

As an agent, there could be multiple approaches to model its motion. To keep things simple, we assume the robot is a non-holonomic agent in a 2D environment with 3 degrees of freedom, but only 2 degrees of motion. Doing so, we can model its motions using the forward kinematics equations:

$$\dot{x} = u \cdot \cos\theta \tag{1}$$

$$\dot{y} = u \cdot \sin\theta \tag{2}$$

$$\dot{\theta} = \omega \tag{3}$$

where $\dot{x}$, $\dot{y}$ and $\dot{\theta}$ represents the change in the $x$, $y$ positions and the heading $\theta$ respectively; $u$ and $w$ represents the forward and rotational velocities respectively. To translate these velocities into positions, we adopt a small time difference $\Delta t = 0.1$ for each step in the environment.

### B. Reinforcement Learning

In general, reinforcement learning requires two main ingredients: (a) an agent that continually interacts with the environment based on the observations and rewards it receives, and (b) an environment that changes from the actions taken by the agent. In our case, our agent is simply the robot that performs actions based on different policies that we experiment with, and its main goal is to maximise the total reward. As our actions are coupled with its state observations, we can define the action-value function it can achieve for some state $S_t = s$ and action $A_t = a$ at time step $t$ as:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{4}$$

$$G_t = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \tag{5}$$

where $\pi$ is simply the policy we decide our actions from, and $G_t$ is the cumulative sum of rewards (or penalty) up till some terminal time $T$ as exponentially weighted by some discount factor $\gamma$. $q_\pi(s,a)$ is then simply the Bellman [12] expectation equation. To maximize our reward is akin to solving for the Bellman optimality equation instead:

$$q_*(s,a) = \max_\pi q(s,a) \tag{6}$$

which can be solved by finding some optimal policy $\pi^*$.

In principle, we could use model-based reinforcement learning where we can model the state and transition functions of the environment and solve the Bellman equations optimally using dynamic programming. However, our action-state spaces can be enormous, especially considering we intend to model our state and environments *continuously* (so we have infinitely large state/action spaces). Thus, we consider efficient sample based approaches where we train our models to learn from raw episodes directly.

### C. Policy Gradient Algorithms

Within sample based approaches, there are multiple methods to solve the Bellman equations proposed above, such as using Temporal Difference learning [12] and Monte Carlo (MC) methods that try to learn the action-value function and output an action accordingly. However, we elect to use policy gradient algorithms, where we directly output the actions using a parameterised function, which we can represent using a neural network trained with stochastic gradient descent [7] and backpropagation [8]. Mathematically, we can now parameterise our return function as some $G(\theta)$:

$$G(\theta) = \sum_{s \in \mathscr{S}} d_{\pi_\theta}(s) V_{\pi_\theta}(s) \tag{7}$$

where $\mathscr{S}$ is some set of states and $V_{\pi_\theta}$ is yet another Bellman expectation equation that we can re-express as the expectation of the action-value function over all actions sampled from the parameterised policy $\pi_\theta$:

$$V_{\pi_\theta} = \sum_{a \in \mathscr{A}} \pi(a|s,\theta) Q_\pi(s,a) \tag{8}$$

where $\mathscr{A}$ is the set of actions. Furthermore, our policy could be either deterministic where we have $\pi_\theta(s) = a$ or

*Equal contributions.

1. The authors are with University of Cambridge.

a stochastic one, where $\pi_\theta(a|s) = \mathbb{P}_\pi(S = s|A = a)$. We experiment with algorithms that use both types of policies.

## II. Approach

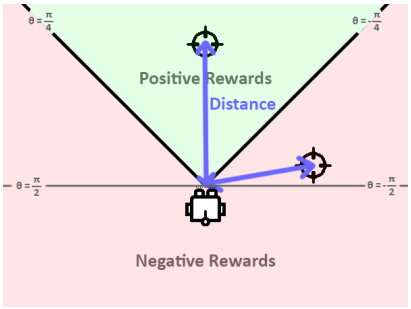


Fig. 1. Illustration of the reward function. When the robot has relative heading in the range of the green region, it receives positive distance and direction reward.

### A. Reward Function

In our environments, we compose the reward value function based on the agent's direction, distance and rotational velocity as such:

$$\begin{aligned} R_{\text{total}} &= R_{\text{direction}} + R_{\text{distance}} + R_{\text{rotation}} \\ R_{\text{direction}} &= \pi - 2\theta' \\ R_{\text{distance}} &= min(\frac{1}{(d_{goal})^2}, \alpha) \cdot cos(\theta') \\ R_{\text{rotation}} &= -|\omega| \\ \theta' &= min(2|\theta|, \pi) \end{aligned} \tag{9}$$

where $\alpha$ is a parameter we tweak to value 10; $d_{\text{goal}}$ is the distance between the robot and the next point to move to; $\theta$ is the heading relative to the goal.

Intuitively, for $R_{\text{distance}}$, we want to reward the robot when it is close to the goal, up till some maximum value $\alpha$, but only if it approximately facing the goal, using the cos term. Figure 1 gives an illustration, where we have the black lines indicating the reward boundaries of values 0. To reinforce the directional importance, we add another term $R_{\text{direction}}$ to incentivise the robot up till some value $\pi$. Finally, we observe during training there is a tendency for high rotational velocities, causing the robot to circle around in the same place and limit exploration. To prevent this, we add a penalty $R_{\text{rotation}}$ to discourage extreme magnitudes of rotational velocities.

### B. Environments

Firstly, we describe the environment where our robot directly outputs velocities to navigate the environment. In order to follow a path, we let the agent know which point in the path it should proceed onto next. To do so, we design this point to be decided by a function $p$ parameterised by some $k \in [0,1]$ such that:

$$p(k) = (x_g, y_g) \tag{10}$$

where $(x_g, y_g)$ refers to the position of the point to head towards, and the path starts with $k = 0$ and ends with $k = 1$. Through modifying the function $p$, we are able to design arbitrary paths for the agent to follow. However, for simplicity, we allow the path function $p$ to be a straight line.

Doing so, we define the observation space of the agent to be simply $(x, y)$, the position of $(x_g, y_g)$ relative to the robot's coordinates, for some $k$ yielding a *next point to follow* within a certain distance on the path ahead of the robot:

$$\begin{bmatrix} x \\ y \end{bmatrix} = M(-\theta) \cdot \begin{bmatrix} x_g - x_r \\ y_g - y_r \end{bmatrix} \tag{11}$$

where $M$ is simply a rotation matrix that rotates the positions by the agent's heading $\theta$. Based on these observations, our agent's action space is then the new forward and rotational velocities $(u, \omega)$ which are output directly. Our agent's internal state is then modified using the forward kinematics equations as illustrated earlier.

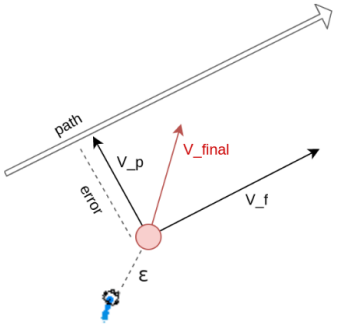


Fig. 2. Illustration of our path controller environment design. Our robot is at some distance $\varepsilon$ away from the holonomic point (pink circle), which has some distance error away from the path origin. The final vector direction taken by the holonomic point is the linear combination of the perpendicular vector $V_p$ and forward vector $V_f$ with constants depending on the PID output.

Now, to integrate a controller for the agent to follow the path decided by a holonomic point, we build an additional path controller environment where we have the observation space as the point's position, which we can easily obtain as it is always some $\varepsilon$ distance in front of the robot. Since we know the exact positions $(x_g, y_g)$ where the point should be on, we can measure the error deviated from the path. This error can then be fed into the controller function $C$, where we choose a PID controller parameterised by a triple $K = (K_p, K_i, K_d)$. To decide the influence of the controller on the point, we define a perpendicular vector $V_p$ facing the path, and $V_f$ is simply the forward vector along the path. Our final vector direction is then a linear combination $V_{\text{final}}$ of the two vectors:

$$V_{\text{final}} = -C_K(e)V_p + V_f \tag{12}$$

Intuitively, when the error $e$ is large, then $C_K(e)$ is large, causing vector $V_p$ to dominate and the holonomic point is inclined to stick closely to the path first before moving forward. Now, we further scale normalised $V_{\text{final}}$ by a scalar speed to give us the horizontal and vertical velocities of the point $(\dot{x_p}, \dot{y_p})$. Our eventual velocities for the robot can then

be decided using feedback linearisation equations:

$$u = \dot{x_p}\cos\theta + \dot{y_p}\sin\theta \tag{13}$$

$$\omega = \varepsilon^{-1}(-\dot{x_p}\sin\theta + \dot{y_p}\cos\theta) \tag{14}$$

We can now let the action space of our environment be the hyperparameters of the PID controller that are updated in each step, for our reinforcement learning algorithms.

## III. Algorithms

In this section, we provide a brief overview of the specific algorithms used in our work.

### A. Deep Deterministic Policy Gradients

Deep Deterministic Policy Gradients (DDPG) [5] is an off-policy actor-critic algorithm that is model-free. In actor-critic algorithms, we aim to also learn the value function through a critic model, instead of just the policy function. This assists us in reducing the variability of gradient updates for the policy function (as represented by the actor model). The critic and actor models could be parameterised as neural networks with some weights, and trained alternatively by freezing one model at a time.

For training these models, we need to obtain gradient updates for both. We cast the critic value estimation problem as a regression problem where its objective function is simply the mean squared error between the true value function and its estimate. Now, as DDPG uses a deterministic policy, where $\pi_\theta(s) = a$, which we can imagine to be a *special case* of the stochastic policy where we have an extremely skewed distribution resulting in a very high probability for just one action. It can be shown in [11] that the deterministic policy can be reparameterised as a stochastic policy with zero variance. Thus, we obtain gradients for our deterministic policy actions and train DDPG using common policy gradient frameworks. Further details can be found in [5], [11].

As DDPG is off-policy, it is sample efficient as we replay past experiences to train the model. However, this sacrifices the optimality of the Bellman equation as we violate a key assumption that all states preserve the Markov property.

### B. Proximal Policy Optimisation

In Proximal Policy Optimisation (PPO) [10], the key idea is that we want to update a *stochastic* policy with large steps for a faster convergence, but want to limit the possibility of taking *bad steps* that can crash the performance of the agent. For example, a large step resulting in a very high forward velocity can allow the agent to crash onto an obstacle immediately. Previous approaches such as Trust Region Policy Optimisation (TRPO) [9] attempt to do this by training the policy model using the following objective:

$$\max_\theta \hat{\mathbb{E}}_t \left[ \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}\hat{A}_t - \beta \text{KL}[\pi_{\theta_{\text{old}}}(\cdot|s_t), \pi_\theta(\cdot|s_t)] \right] \tag{15}$$

where the first term is simply the surrogative objective composed of a ratio of the new and old policy actions, subject to a KL-divergence constraint that prevents the new policy from being too different to the old one. Here, $\beta$ is then simply a scaling coefficient that varies for different problems.

While it is possible to solve the TRPO objective using natural gradients as an analytical solution exists, it is expensive to do so in practice as it requires second-order methods. Furthermore, it is difficult to tune $\beta$ for different problems. PPO instead mitigates these by using first-order methods subject to further softer constraints to mitigate bad solutions. In our work, we specifically refer to the PPO-Clip variant that has worked better in practice. The objective for the policy can be formulated as such:

$$L^{\text{CLIP}}(\theta) = \hat{\mathbb{E}}_t \left[ \min(r_t(\theta))\hat{A}_t, \text{clip}(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t \right] \tag{16}$$

where $r_t(\theta)$ is simply the ratio of new/old policy actions as before:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \tag{17}$$

Intuitively, the algorithm simply prevents the ratio of the actions to be too different in both directions within the range $[1-\varepsilon, 1+\varepsilon]$ where we can set $\varepsilon = 0.2$. We chose this algorithm since it works better in practice than the PPO-Adaptive variant. Further differences can be found in [10] which the reader is encouraged to read.

### C. Curriculum Learning

In curriculum learning [2], one typically starts training the agent with a simple task, which progressively gets more difficult. We perform curriculum learning by having the agent first learn to follow a fixed path for some $n$ epochs, before we change the paths to be composed of randomly generated lines in the environment. We demonstrate empirically that this still allows the algorithms to converge, particularly for DDPG which is more brittle to converge.

## IV. Experiments

### A. Implementation

We built the environment from scratch using OpenAI Gym [3], where we designed the environment to be a 2D box, and the agent is consistently aware of the entire environment. We implemented DDPG and PPO algorithms using OpenAI SpinningUp [1], which internally runs on a PyTorch [6] backend. For the architecture of the policy network, we use an MLP network with 2 hidden layers of 64 units each, with ReLU [4] activations in between and tanh activation in the output.

### B. Rewards Over Time

From Figure 3, we see that PPO performs consistently better than DDPG in both the warm-up and full training phases. Particularly, we observe that during warm-up, PPO converges a lot faster than DDPG, as it reaches close to the peak reward of $10+\pi$. This concurs with our expectation that training DDPG is harder to converge in practice. Due to time constraints, we did not extensively tweak the training hyperparameters.

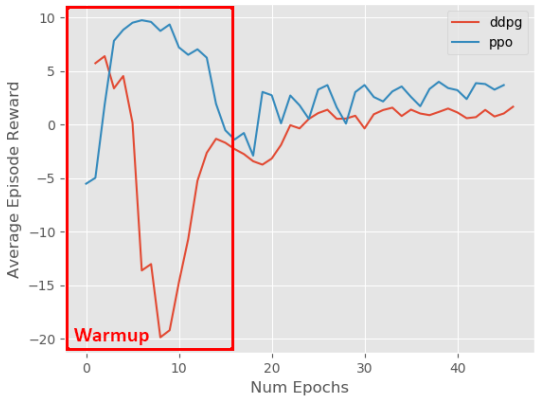


Fig. 3. Average reward per episode over training epochs. We use the first $n = 15$ epochs for the warm-up, as part of curriculum learning.

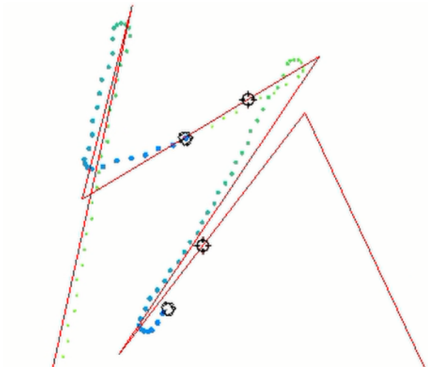

Fig. 4. Robot progressively following a path, which are connected lines generated randomly to create sharp turns.

### C. Following Paths

In Figure 4, we compose the progression of the robot's trajectory over multiple time steps. The robot is given a fixed number of time steps per episode to complete the path, and has to start from the beginning of the path despite being randomly generated at arbitrary positions within the environment. We observe the robot is able to approximately follow the trajectory even with sharp turning points.

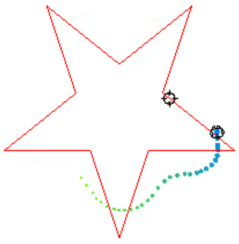

Fig. 5. Star-shaped path in the environment with shorter path lengths at each side of the star.

To accentuate the difficulty, we test the robot on a star-shaped environment where the path lengths are shorter. This means it is required to make sharper turns more regularly, causing the path to be more variable and difficult to follow. From Figure 5, we see that while the robot still attempts to stay on the lines, it is harder for it to be exactly on the line due to more frequent turns.

## V. Conclusion

We showed it is possible to train an agent to follow a path using reinforcement learning, yielding results comparable to planning algorithms. From the two tested algorithms, PPO resulted in higher average reward per episode than DDPG, and the learning was computationally more efficient.

Our approach also demonstrated that an intelligent agent can learn to adapt its behaviour according to the environment, for example slowing down before a sharp turn. As future work, we suggest the following possible extensions:

1) Include multiple path points ahead of the robot in the observation space, to let the agent better adapt its behaviour, i.e. make sharp turn follow path's direction
2) Use advanced simulation engine to more accurately simulate real-world motion errors
3) Learn to avoid obstacles while following a path

## VI. Acknowledgements

We thank Dr Amanda Prorok for this comprehensive course on robotics, with further thanks to Ryan Kortvelesy for providing useful pointers and directions for the project. The division of work in our group is approximately:

- Juraj - DDPG algorithm, agent-environment interaction, path-following controller training.
- Kwot Sin - PPO algorithm, setup and integration of frameworks.
- Wilson - Environment visualisations, reward function design, diagrams.

## References

[1] Joshua Achiam. Spinning Up in Deep Reinforcement Learning. 2018.
[2] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48, 2009.
[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.
[4] Richard HR Hahnloser, Rahul Sarpeshkar, Misha A Mahowald, Rodney J Douglas, and H Sebastian Seung. Digital selection and analogue amplification coexist in a cortex-inspired silicon circuit. *Nature*, 405(6789):947–951, 2000.
[5] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.
[6] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.
[7] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.
[8] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.
[9] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897, 2015.
[10] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.
[11] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.
[12] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.