Evaluating the performance of Deep Reinforcement Learning algorithms in a restaurant delivery robot setting

1. Problem statement

Large restaurants often use delivery robots during peak hours to assist their staff in bringing meals from kitchen to customer. It is important that such a robot can deliver meals **quickly** and **to the right customer**, and **without bumping into anything** or anyone; the better a robot can fulfill these conditions, the more satisfied the customer and staff will be.

To achieve these goals, it is imperative to train the robot using a suitable machine learning approach. **Deep reinforcement learning (DRL)** methods have recently proven successful for similar delivery problems (Chen et al. (2022); Jahanshahi et al. (2022)), and seem suitable to this use case as well. In this report, we investigate the performance of two such DRL algorithms, **DQN** and **PPO**, in the context of the restaurant delivery problem.

2. Environment design

In essence, our environment is a 2D plane, representing the restaurant as seen from above. The robot's **state** is described by 3 continuous variables: its x and y position on the plane, and its rotation ϕ relative to the x-axis. The robot can take 3 **actions**: *move forward*, *rotate left*, or *rotate right*. Moving forward moves the robot in the direction it is currently facing by a fixed distance per time step, and similarly the rotate actions increase or decrease the robot's rotation by some fixed angle. All these transitions happen with 100% probability regardless of the previous state, i.e. the process is Markov and we do not model environmental noise.

Besides the agent itself, the environment also includes a target and obstacles. The *target* is where the robot needs to deliver food to, and is represented as a big green circle; if the robot enters this circle in any time step, it receives a large positive reward and the episode ends. The *obstacles* can be e.g. walls or tables, and are shown as gray squares; if the robot attempts to move into a position covered by an obstacle, the move is canceled and the robot receives a large negative reward. Besides this, the robot also receives a small negative reward each step, with an additional penalty for taking a rotate action as opposed to a move forward action, in order to push it towards finding the *shortest* path to the target. A visualization of what our testing environments look like, together with typical paths that the agents take on them, is shown in the results section in Figure 2.

3. Related work

We implemented DQN since it sparked the field of DRL and is still commonly used as a baseline comparison (François-Lavet and Henderson (2018); Arulkumaran et al. (2017)). For

the second algorithm, we considered several options and potential strengths and weaknesses in our setting (Li (2018); Arulkumaran et al. (2017)):

- **Dueling DQN**: Good for discrete actions, upgrades standard DQN, but needs careful tuning for exploration and doesn't handle partial observability well (Wang et al. (2016)).
- **A3C**: Supports both discrete and continuous actions, is fast with parallel workers, but is relatively unstable and mostly replaced by newer methods (François-Lavet and Henderson (2018); Fujimoto et al. (2018)).
- **DDPG**: Great for continuous actions and high control tasks like robotics, but not suited for discrete actions and is very sensitive to hyperparameters (François-Lavet and Henderson (2018); Arulkumaran et al. (2017)).
- **SAC**: Very stable and excellent for exploration, but more complex, and mainly designed for continuous actions (Haarnoja et al. (2018); Li (2018)).
- **PPO**: Straightforward, stable and scalable for discrete action spaces, but somewhat sensitive to hyperparameter tuning (Schulman et al. (2017)).

We settled on PPO for our task, mainly because recent research (Kiran et al. (2020); Luo et al. (2024)) has shown PPO to work well for robotics and navigation, even when rewards are sparse or the agent doesn't know the target location in advance. PPO also naturally fits our discrete action space, and its stochastic policy and entropy regularization let the agent explore better when the target location is unknown; another major advantage is its *clipped objective*, making learning more stable and less prone to sudden failures. Hyperparameter tuning must be done carefully, but the positives listed above make PPO more suitable to our problem than the alternatives.

4. Technical background

4.1 DQN

DQN extends the tabular Q-learning algorithm from Assignment 1 to our continuous state space. Rather than storing a lookup table of Q-values, we approximate the action-value function by a neural network, implemented as a 3-layer MLP with hidden layer size 128. It takes the 3 state features and outputs one scalar Q-value for each of our 3 actions. Besides this *policy net* with parameters θ, we also have an identical *target net* with weights θ^- used for judging which action is best. The policy net's parameters are copied to the target net every 500 steps; otherwise, the target net is kept "frozen" to stabilize learning.

The agent accumulates experience and stores this in a *replay buffer*. In each training step, we sample a minibatch $\{(s_i, a_i, r_i, s'_i, \mathbf{1}_{s'_i \text{ is terminal}})\}_{i=1}^{B}$ from this buffer, and take a gradient descent step to minimize the *mean-squared Bellman error*:

$$L(\theta) = \frac{1}{B}\sum_{i=1}^{B}\Big[r_i \;+\; (1 - \mathbf{1}_{s'_i \text{ is terminal}})\,\gamma \max_{a'} Q(s'_i, a'; \theta^-) \;-\; Q(s_i, a_i; \theta)\Big]^2,$$

Similar to Q-learning, actions are chosen according to an ϵ-greedy policy with decaying ϵ; We select a random action with probability

$$\epsilon_t = \epsilon_{\text{end}} + (\epsilon_{\text{start}} - \epsilon_{\text{end}}) \exp(-t/\tau)$$

and otherwise we pick the greedy action (i.e. the action that maximizes Q according to the policy net). τ is a hyperparameter determining how quickly ϵ decays.

This basic DQN agent works in principle, but in practice suffers from an effect called *catastrophic forgetting*; it may learn a good path early on, but then forget this path again as it trains further. To mitigate this effect and focus only on learning *successful* strategies, we maintain *two* replay buffers, $\mathcal{D}_{\text{succ}}$ and $\mathcal{D}_{\text{fail}}$, storing transitions from successful and failed episodes respectively (i.e. ones that reached the target vs ones that did not). Each minibatch is sampled so that a fixed fraction comes from $\mathcal{D}_{\text{succ}}$ (with shorter successful trajectories weighted more heavily) and the remainder from $\mathcal{D}_{\text{fail}}$, which helps preserve performance on earlier solved paths. We found that sampling more successes than failures helps mitigate catastrophic forgetting.

4.2 PPO

Similar to DQN, the PPO agent uses a policy and a target net, called the *actor* and *critic* networks in this case. Their architecture is the same as in DQN; two hidden layers of 128 units with ReLU activation followed by a layer outputting values for our 3 actions.

Like DQN, the agent collects data by running its current policy for a number of steps, and then updates both the actor and critic networks using that experience. To train the **actor**, PPO uses a **clipped objective**:

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

where

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

is the ratio of new to old policy probabilities, and

$$\hat{A}_t = \delta_t + (\gamma\lambda)\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots \qquad \text{with} \qquad \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

is the **generalized advantage estimate (GAE)**, which measures how much better action a_t turned out to be compared to what was expected ($V(s_t)$), with hyperparameters γ and λ discounting future rewards. Clipping $r_t(\theta)$ to be within the range $(1-\epsilon, 1+\epsilon)$ limits how much the policy can change in one update, which makes training more stable; we have found the best results with $\epsilon = 0.1$. The loss function encourages taking actions that led to higher advantages, but avoids large policy updates that might hurt performance.

To train the **critic**, the agent uses **mean squared error** between its predicted value and the actual return:

$$\mathcal{L}_{\text{critic}} = \frac{1}{N} \sum_t \left(V(s_t) - R_t \right)^2$$

The **total loss** is a combination of the actor loss, the critic loss, and a small **entropy bonus** to encourage exploration and prevent the policy from becoming too deterministic:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{actor}} + 0.5 \cdot \mathcal{L}_{\text{critic}} - 0.03 \cdot \mathcal{H}[\pi]$$

The relative weights for these factors were decided during hyperparameter tuning. Unlike DQN, which performs one training epoch after every step, PPO first gathers experience for longer (in our case, 2 full episodes) and then trains for several epochs (in our case 10). Training is done using the **Adam optimizer** using minibatches of size 64.

5. Experimental setup

For the experiments, we have devised two obstacle layouts, both representing the dining area of a restaurant. The obstacles we train the robot to avoid are tables, and the `table_hard` environment has more of them than `table_easy`. The layouts are pictured in Figure 2.

We test the performance of both the DQN and PPO agents on both layouts. To evaluate their performance, we measure both the quality of the best path found and how quickly the agent finds (or at least approximates) this best path. The quality of the path is measured by its length, i.e. the *number of steps* it takes to go from start to target, while the speed of converge is measured by the *rolling average of the reward* earned in the last 50 episodes. The faster this average reward converges to the reward for the best path found, the faster the agent learns. We will report the number of episodes it takes to get within 5% of this optimal reward (ignoring the first 50 episodes so the average can stabilize).

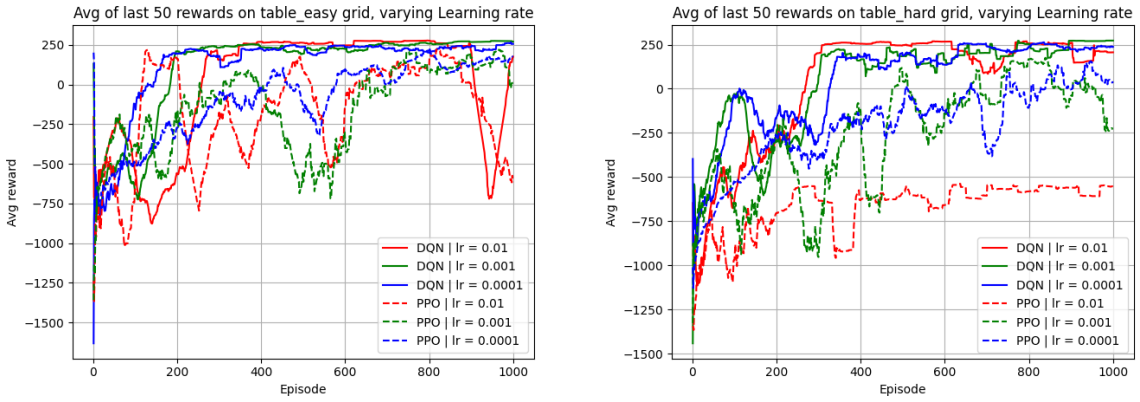
Besides the obstacles in the environment, we also vary several other parameters to see how they affect the agent's performance. We vary these one at a time, keeping the others constant at a pre-tuned default value. To compare the agents with each other, we test the following values for the parameters that are shared between them:

- **Nr. of episodes**: $500, 1000, 2000$, default 1000. More episodes means the agent has more time to learn the optimal path, but may also lead to the agent "unlearning" or fixating on a bad path.
- **Nr. of steps per episode**: 250, 500, 1000, default 500. Shorter episodes mean the agent is less likely to find the target in the initial stage, but also means that if it *does* find the target, the path it took is more likely to be close to optimal.
- **Learning rate**: $10^{-2}, 10^{-3}, 10^{-4}$, default 10^{-3}. A lower learning rate means the agent takes longer to converge, but also allows it to converge to "sharp" minima.
- **Reward for reaching the target**: $100, 300, 700$, default 300. A higher target reward means that the agent is more focused on exploiting the first few paths that reach the target versus exploring more.

In these experiments, both agents are trained for the specified number of episodes, with default parameter values found in the `agent_config.json` file; these were manually tuned beforehand on the default settings mentioned above. Results for the experiments can be found in Table 1.

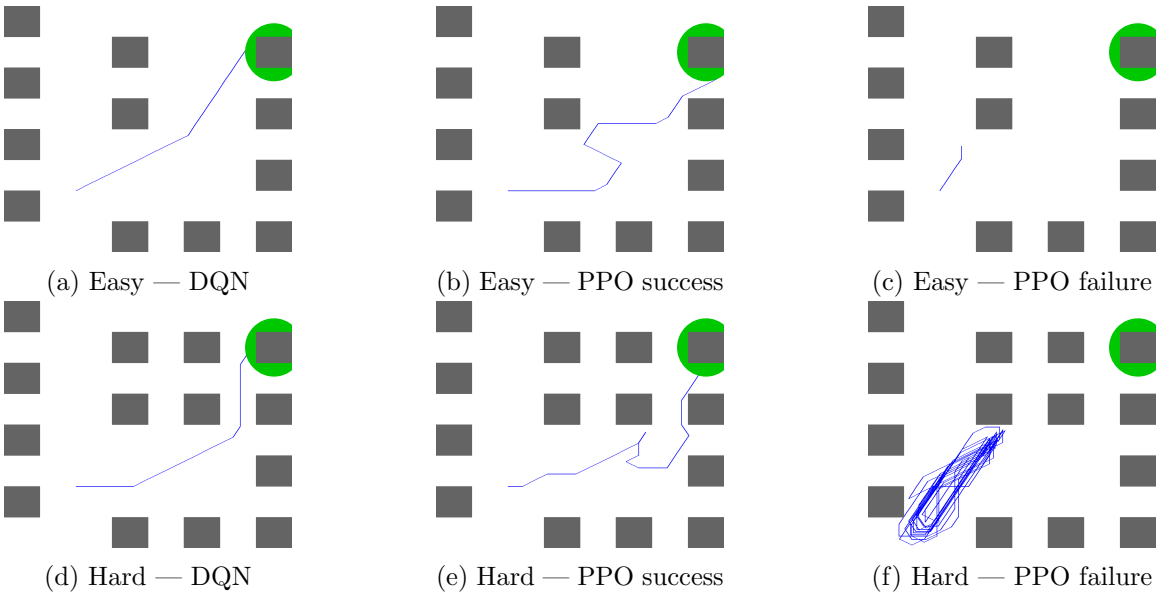
6. Results & analysis

The results of the experiments can be found in Table 1. Plots of the reward curves for the learning rate parameter can be found in Figure 1, and some example paths that the agents took are shown in Figure 2.



(a) `table_easy` (b) `table_hard`

Figure 1: Rolling average of rewards over the course of training in both environments as we vary the learning rate. Here we see that the convergence episode reported in Table 1 doesn't tell the full story; although the reward for high learning rate (red) gets within 5% of the maximum the quickest, the other lines also convergence around this time (or possibly even faster), but they happen to not pass the 5% threshold until later. In general, we can conclude that a smaller learning rate leads to more stable convergence, without being much slower than a high learning rate.



(a) Easy — DQN (b) Easy — PPO success (c) Easy — PPO failure

(d) Hard — DQN (e) Hard — PPO success (f) Hard — PPO failure

Figure 2: Some examples of typical paths in the easy (top row) and hard (bottom row) environments. (a)/(d) The DQN agent is able to consistently find a (near-)optimal path. (b)/(e) The PPO agent may find a path, but often a clearly suboptimal one. (c)/(f) If the robot doesn't find a path, this is often because it spins around in circles or in place.

Grid		Easy		Hard	
Params \ Agent		DQN	PPO	DQN	PPO
Default		22/691	44/746	24/773	110/798
#Episodes	500	20/381	145/374	24/414	$*/*$
	2000	21/675	28/912	21/1168	42/1697
#Steps/episode	250	21/669	85/843	24/612	$*/*$
	1000	22/454	55/192	21/664	39/695
Learning rate	10^{-2}	21/389	58/125	23/333	$*/*$
	10^{-4}	21/979	64/785	23/635	$*/*$
Target reward	100	21/379	42/836	$*/*$	53/854
	700	21/425	32/356	21/367	$*/*$

Table 1: Experimental results. In each cell, the first number is the length of the best path found by the agent, while the second number is the episode where the rolling average of the reward over the last 50 episodes first reaches within 5% of its best value. A $*$ indicates the agent was not able to find a path to the target within the specified number of episodes.

From the results, it is immediately clear that the DQN agent performs best across the board; not only does it consistently find a (near-)optimal path, it also converges to this path faster than the PPO agent. The PPO agent is generally able to find a decent path on the easy grid, but really struggles on the hard grid; and even if it does find a path, it tends to make unnecessary detours, as shown in Figure 2. Now, regarding the effects of the individual parameters:

- Varying the **difficulty of the environment** causes PPO to really struggle; a more complex environment means fewer episodes will reach the target, especially the early ones where movement is mostly random. DQN on the other hand is able to handle the more complex environment quite well, and does not require significantly more episodes to learn a near-optimal path.

- Varying the **number of training episodes** does not have a significant effect on the quality of the found path for DQN, but is more significant for PPO; with just 500 episodes, the agent finds a worse path in the easy environment, and doesn't even find a path at all in the hard environment.

- Varying the **number of steps per episode** has an interesting effect on training speed. On one hand, longer episodes mean the agent experiences more transitions in the same number of episodes, which should cause it to learn faster; this is what we see with DQN on the easy grid, and with PPO on both grids. On the other hand, with a longer episode length, the agent may reach the target in a very roundabout way, which nudges the agent towards learning a suboptimal path. This may be why the higher episode length is no longer advantageous for DQN on the hard grid; it has to spend time "unlearning" a suboptimal path.

- Varying the **learning rate** has a significant effect on convergence speed. A higher learning rate causes the agents (DQN especially) to converge to its optimum faster, though there are some caveats here discussed in Figure 1. It seems PPO in particular is very sensitive to this, and doesn't find a path on the hard grid if its learning rate is too high (making it overshoot the minimum) or too low (making it not find the minimum within 1000 episodes)

- Finally, varying the **reward for reaching the target** does not seem to have a significant effect; the only thing that stands out is that DQN, which managed to find a path in every other situation, is not able to find the target in the hard obstacle layout when the reward signal is too small.

7. Limitations & future work

There are several weaknesses which limit the quality of the conclusions drawn in the previous section. For one, due to long training times of both agents, we were only able to test one run for each (grid, agent, parameter) combination, meaning the results in Table 1 are heavily subject to randomness (especially for the PPO agent). Even though we did use the same random seed each time, it would be best for future work to test e.g. 3 or 5 runs for each combination to limit the effect of good/bad initialization.

Another limitation is that the convergence episode isn't a soundproof measure of the convergence speed. A good example of this is the experiment where we varied the number of episodes; slight optimizations to the path in later episodes cause the convergence threshold (i.e. the reward its path needs to reach for an episode to be considered the convergence episode) to move up, which means a path found at e.g. episode 300 will be close enough to the best path found after 500 episodes, but not to the best path found after 2000 episodes. It would have been better to devise a way to calculate the optimal path beforehand and base the convergence threshold off of the reward for *this* path.

For future work, we also recommend testing more parameter combinations and different environments, such as an environment representing the entire restaurant as opposed to just the dining area, where the main obstacles are walls instead of tables. Another direction of further research is to test with parameters specific to just one of the agents, to see how sensitive that agent is to variations in that parameter.

8. Conclusions

In this report, we have investigated the performance of a restaurant delivery robot with two different deep reinforcement learning algorithms, DQN and PPO, in a simulated dining area with two different levels of complexity. In our experiments, we saw that DQN performs better than PPO across the board, being able to consistently find a near-optimal path in fewer episodes than PPO. We also saw that the episode length and learning rate have a noticeable effect on convergence speed, so these should be selected carefully before training a robot in practice.

Overall, we have found the best results with the DQN agent, when training for 500 episodes of length 500, using a learning rate of 10^{-3} and a target reward 300. We recommend these settings for any restaurant owner seeking to train and apply these robots in practice.

References

K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. A brief survey of deep reinforcement learning, sep 2017. URL https://arxiv.org/pdf/1708.05866.

X. Chen, M. W. Ulmer, and B. W. Thomas. Deep q-learning for same-day delivery with vehicles and drones. *European Journal of Operational Research*, 298(3):939–952, 2022. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2021.06.021. URL https://www.sciencedirect.com/science/article/pii/S0377221721005361.

V. François-Lavet and P. Henderson. An introduction to deep reinforcement learning, Dec 2018. URL https://arxiv.org/pdf/1811.12560.

S. Fujimoto, H. van Hoof, and D. Meger. Addressing function approximation error in actor-critic methods, Oct 2018. URL https://arxiv.org/pdf/1802.09477.

T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, Aug 2018. URL https://arxiv.org/pdf/1801.01290.

H. Jahanshahi, A. Bozanta, M. Cevik, E. M. Kavuk, A. Tosun, S. B. Sonuc, B. Kosucu, and A. Başar. A deep reinforcement learning approach for the meal delivery problem. *Knowledge-Based Systems*, 243:108489, 2022. ISSN 0950-7051. doi: https://doi.org/10.1016/j.knosys.2022.108489. URL https://www.sciencedirect.com/science/article/pii/S0950705122002088.

B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Pérez, Feb 2020. URL https://arxiv.org/abs/2002.00444.

Y. Li. Deep reinforcement learning: An overview, Nov 2018. URL https://arxiv.org/pdf/1701.07274.

F.-M. Luo, T. Xu, H. Lai, X.-H. Chen, W. Zhang, and Y. Yu. Deep reinforcement learning with enhanced ppo for safe mobile robot navigation, Aug 2024. URL https://arxiv.org/pdf/2405.16266.

J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms, Jul 2017. URL https://arxiv.org/abs/1707.06347.

Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas. Dueling network architectures for deep reinforcement learning, APR 2016. URL https://arxiv.org/pdf/1511.06581.

Individual contributions

Name	Contributions
Aniket	Researched second DRL method, assisted development of PPO agent, implemented common train script, wrote related work section
Desiree	Assisted development of DQN agent, researched & implemented improvements for it, implemented final reward function.
Maxim	Implemented GUI, implemented obstacles in environment, did final code cleanup
Natasha	Implemented visualization function, improved common train script, implemented experiments script
Paolo	Implemented and improved DQN agent, wrote technical part for DQN in the report
Sonia	Researched second DRL method, implemented PPO agent, wrote technical part for PPO in the report.
Wouter	Implemented continuous environment, designed & ran experiments, wrote experimental setup and later sections, did final code and report cleanup & review, did general management.

Table 2: Individual contributions of the group members.

GitHub repository

The code for the assignment can be found here: `https://github.com/Aniket-Mishra/2AMC15_Intelligence_Challenge_A2`. The final version can be found on the `main` branch.