Solving Overcooked with DQN & PPO

# **1. Introduction**

The problem we are trying to solve is delivering at least 7 soups in a collaborative kitchen operated by two chefs. Conceptually, this seems quite trivial to us humans. But it can be very challenging to an RL agent which, unlike humans, must learn all necessary concepts from scratch: stove, ingredients, delivery, collaboration, theory of mind, and so on. On the other hand, humans can bootstrap from the knowledge and experiences they have built over their lives from playing games and interacting with the world.

The state space is represented by a 96-dimensional vector and is fully observable. The state representation includes information like player orientation, the object being held, the closest object, the closest serving area, whether the pot is cooking, and the number of ingredients in the pot, the equivalent information as seen by the other player, the distance to the other player, and the current position. More details can be found at the project description [1]. The action space consists of 6 discrete actions: up, down, left, right, do nothing, and interact (which is context dependent on the player’s current position). The reward is simply a +20 for every soup delivered. The layout is considered solved when reward of +140 is achieved. I utilize shaped rewards to ensure some learning occurs and avoid sparse reward. The shaped rewards used are: +3 for ingredient placement in pot, +3 for dish pickup, and +3 for soup pickup. Thus, during training, each agent’s reward is the soup reward which is common to both agents (and is what encourages collaboration), plus the individual shaped reward.

There are 5 layouts to solve: *cramped room* is the simplest and each agent essentially sees the same environment. *Asymmetric advantages* is set up so that each agent is better off specializing in one task, that is, one in potting onions and another in delivering soups. However, since each agent can be placed in either of the roles at random, they must specialize in both. *Coordination ring* requires more collaboration that the other two since the agents can run into each other in the ring and there are two stoves. *Circuit* is a harder version of coordination ring as it is a larger layout and thus requires both more exploration and more collaboration to be successful. *Forced coordination* is the hardest layout and requires explicit collaboration to deliver even a single soup. The convergence criterion used was that the average reward over the last 100 episodes is greater than 140 (7 soups delivered).

I attempted two algorithms: DQN and PPO. The DQN algorithm used is the same as that from Project 2 [2] and won’t be repeated here for brevity. *Proximal Policy Optimization (PPO)* [3] is an on-policy, policy gradients algorithm; unlike DQN, it outputs the action probabilities directly. It is based on the actor-critic architecture, where the actor controls how the agent behaves, and the critic measures how good the action taken is. PPO improves training stability over similar algorithms like *Advantage Actor Critic (A2C)* by avoiding policy updates that are too large. To do that, it uses a ratio that indicates the difference between the current and old policy and clips this ratio to a specific range [1-, 1+]. We try to maximize the performance (so gradient ascent) of the policy network (actor). The performance is measured as:

with (1)

Where is the entropy loss added to ensure sufficient exploration; is the ratio of the new probability of selecting an action to the old probability; is the Generalized Advantage Estimate [3], which indicates how much better an action is compared to the average action in that state. GAE is calculated as: , where is the discount rate; is a knob to balance the bias-variance trade-off, a higher incorporates more future rewards, making it more sensitive to long-term effects, which can reduce variance but might introduce bias is the future state-value estimates are inaccurate. is the one step temporal difference error, computed using the values given by the critic.

# **2. DQN Agents**

I started out with the same DQN algorithm as Project 2; the final DQN algorithm for Project 3 differed in only two hyperparameters: the neural network size and discount factor, . The neural network size increased from 3 layers with (64, 64, 4) units each to 3 layers with (256, 128, 6) units each. I first increased the network size to (128, 128, 6), and the rationale behind this is to accommodate the increase in input vector dimension from 8 to 96. I then doubled the size of the first layer from 128 to 256, as doing so led to faster convergence on coordination ring. After doing grid search on cramped room, I found that a lower value of led to faster convergence, so decreased from 0.99 to 0.95. Intuitively, this makes sense: in Lunar Lander, there is one final objective of wanting to land, whereas in Overcooked, the objective is more short-term of making a soup quickly.

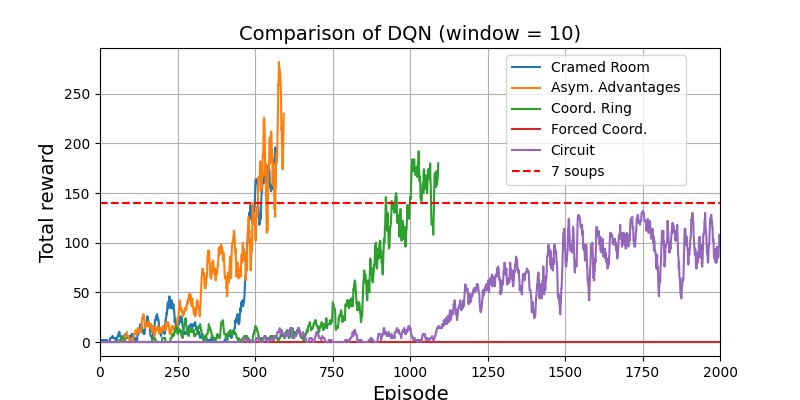
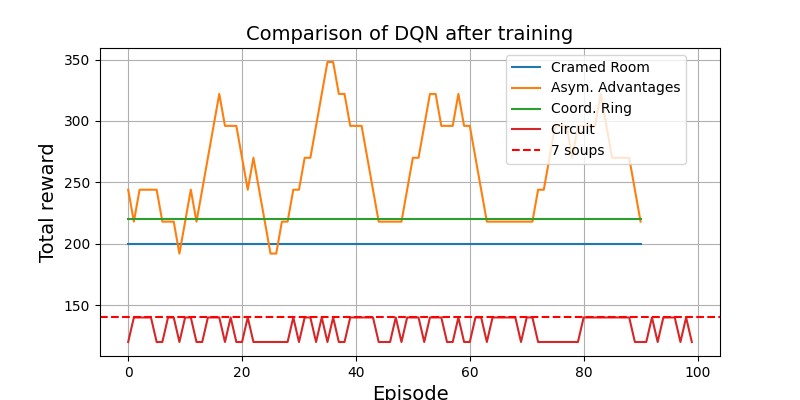
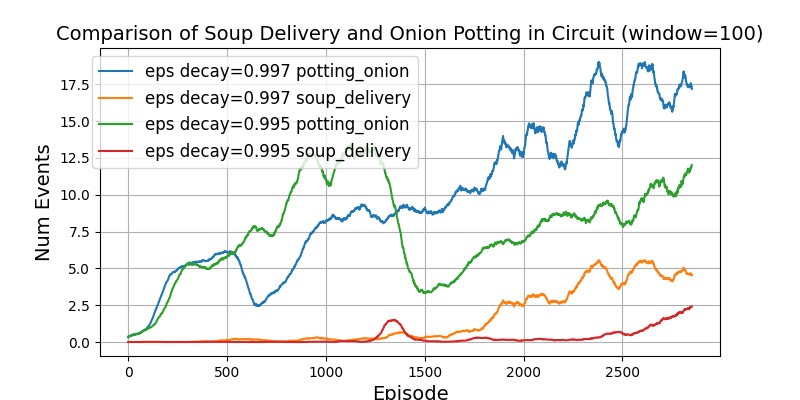
The hyperparameters for the final DQN agents are: =0.95, -decay=0.997 (=1 at start), batch size=64, optimizer=Adam, learning rate=5e-4, network=(256, 128, 6), buffer size=50,000. Each agent was simply an independent DQN, with its own replay buffer. This algorithm was able to solve the first three layouts: cramped room in 577 episodes, asymmetric advantages in 602 episodes, and coordination ring in 1100 episodes. It solved the last layout (circuit) partially: delivering between 6 and 7 soups after training for 1500 episodes. It was unable to solve forced coordination layout, unable to deliver even a single soup throughout training. Figure 1 shows the reward from delivering soups during training on the 5 layouts, and figure 2 shows the same reward for 100 episodes after training.

Figure 2: The reward per episode after training 4 layouts. Forced coord. has 0 reward.

Figure 1: The reward per episode during training for the 5 layouts. A moving average window of 10 is used to reduce the noise

For the first 3 layouts, the project 2 DQN algorithm with the bigger network just worked out of the box. Since this DQN algorithm worked well on coordination ring with -decay rate of 0.995, I assumed it would carry over and work equally well on circuit layout too, since both layouts are quite similar. However, as figure 3 shows, this did not work at well, only increasing the -decay rate to 0.997 made it work. The action of onion potting and soup delivery were discovered much earlier with a higher -decay rate. This makes sense as circuit is a much bigger layout, in terms of surface area, than coordination ring, and therefore requires much more exploration to discover the routes to various points.

Although increasing the -decay rate to encourage more -greedy exploration helped, the agents were unable to consistently deliver 7 soups or more, instead they fluctuated between delivering 6 and 7 soups on circuit. Increasing the -decay rate 0.999 and training for longer didn’t help them find more optimal actions. This might indicate two things: (1) -greedy exploration is not enough, and a different exploration strategy like intrinsic motivation may be required to systematically explore actions; (2) the problem may be with the algorithm itself, and using a MARL algorithm would help. In addition, this DQN algorithm completely failed on the forced coordination layout; and when I visualized the agents in the coordination ring layout, even though they solved it, they could have done much better since only one stove was being used. This suggests that this DQN algorithm is struggling with the temporal credit assignment problem in multi-agent environments: which action by one agent is beneficial to the other agent to maximize the shared reward? Therefore, little collaborative learning occurred. At this point in the project, I was left with three options to try: (1) intrinsic motivation, (2) MARL algorithm like QMIX on DQN, and (3) abandon DQN and try PPO. With 3 weeks left, I gambled on PPO due to its success in RLHF.

Figure 3: Comparison of -decay rate for -greedy exploration with DQN on the Circuit layout. Num of soup delivery and onion potting

# **2. PPO Agents**

My PPO implementation is, like DQN, an independent actor & critic networks for each agent; thus, there were 4 networks in total. The actor loss is specified by Eq. (1) above; the critic loss is the mean squared error between the predicted value and the Monte Carlo return from that state. The pseudocode is:

while not converged:

train\_data = play\_n\_episodes(n)

for epoch in n\_epochs:

for mini\_batch in train\_data:

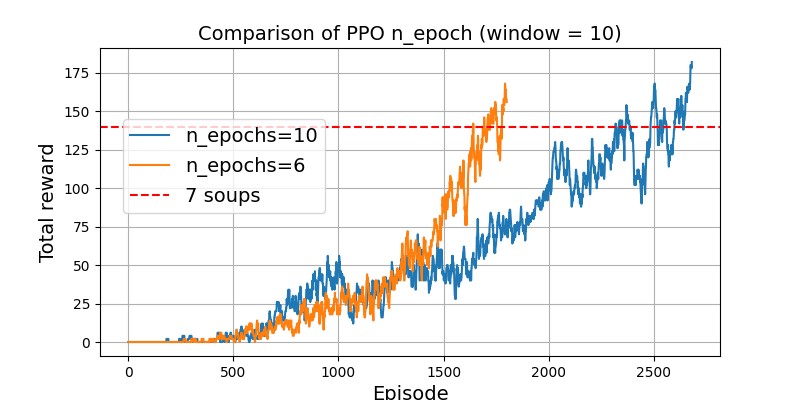
do\_actor\_training(mini\_batch)

do\_critic\_training(mini\_batch)

if (kl\_divergence > target\_kl\_divergence): break

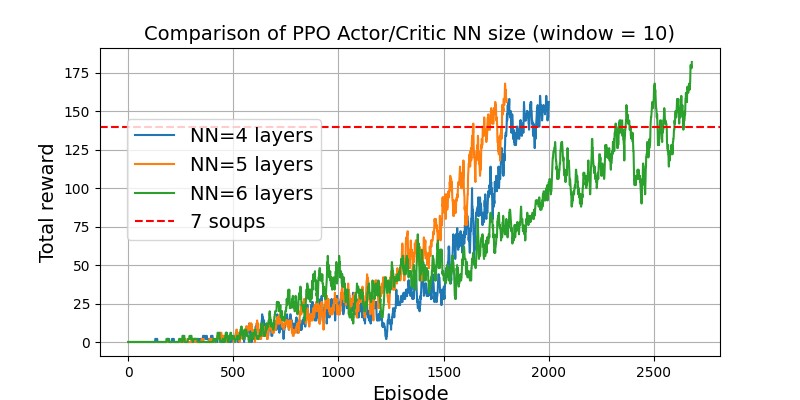
Though my initial PPO implementation worked on the simple CartPole environment from OpenAI Gym, it failed to converge even after 3000 episodes on the simple cramped room layout. I then made several changes to the algorithm itself, based on [3], to get some decent performance out of it.

* Mini-batch update logic: Initially, I was doing n iterations of updates to the actor and critic. On each iteration, a single stochastic mini-batch would be used to perform training. After the change, there would be n epochs of updates, where on each epoch, the whole dataset was used for updates; the dataset was broken into mini batches for each training update. Overall, this helped with reducing the variance of gradient updates as the whole dataset was used.
* Adding entropy loss: Initially, no entropy loss was added to the actor’s loss. This meant that little to no exploration was done. Adding an entropy loss helped with exploration and finding paths. The entropy coefficient started at 0.20 and decayed to 0.01 to ensure smooth transfer from exploration to exploitation as training progressed.
* Gradient clipping before backprop helped with training stability as updates were not large.
* Annealing learning rate: Decaying the learning rate over time helped ensure smoother convergence. This makes sense as one would want smaller updates near the optima.
* Rollout size: Doubling the number of episodes that are played (5 to 10) before training helped improve convergence. This makes sense since as a larger training batch size means more diverse experiences and thus reduces variances in updates, consequently improving training stability.

While none of the above on their own led to an improvement, all of them combined led to convergence. After having a working PPO implementation, I then did extensive grid search to find the best hyperparameters. I used coordination ring as the testbed, as it was the layout with medium difficulty.

For brevity, we will only discuss the hyperparameters n epochs and the network size, though many other hyperparameters played an important role and were compared too. Reducing n epochs helped with faster convergence (figure 4). While this seems counterintuitive at first, it makes sense since doing more updates with the same data causes overfitting. This meant that the agents are not able to adapt to the non-stationary environments quickly enough (due to the other agent learning at the same time).

Figure 4: Comparison of the number of epochs used for actor & critic training in PPO on Coord Ring.

The same network size was used for both the actor and critic to reduce the number of hyperparameters to tune. When testing network size , I started with a 6-layer network: (256, 256, 256, 128, 128, 6), and kept removing the 256-unit layer when testing smaller networks with 5 and 4 layers. The 5-layer network performed the best. The 6-layer network took the longest to converge since it was overparameterized and thus overfitting the changing environment; whereas the 4-layered network had less capacity to capture the complexity of the layout and was slower than 5-layered NN, (figure 5).

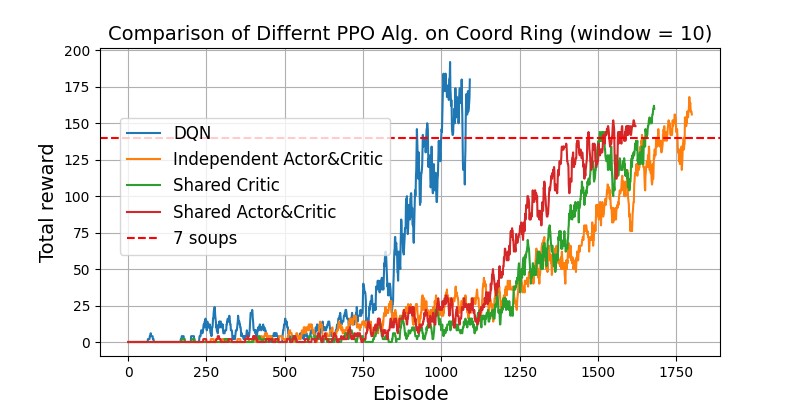
My final hyperparameters were: =0.95, =0.97, batch size=512, n epochs=6, learning rate=4e-4, (clipping factor)=0.02, target KL divergence=0.02, entropy coefficient=0.20, gradient clipping factor=0.50, networks=(256, 256, 128, 128, 6). This algorithm, like DQN, only converged on the first 3 layouts (cramped room: 640 episodes, asymmetric advantages: 670 episodes, coordination ring: 1810 episodes), and unlike DQN, did not even partially solve circuit layout. The sampling efficiency was also much worse than DQN, as it required more episodes to converge on each of the first 3 layouts.

Figure 5: Comparsion of different NN sizes for actor & critic on Coord. Ring.

PPO is known to be more memory efficient (as it is on-policy and doesn’t require a replay buffer), more stable (clipping prevents drastic policy changes that can destabilize training), faster, and suitable for multi-agent environments. Thus, I was determined to make PPO work. After doing extensive hyperparameter search and reimplementing it again to get rid of any bugs, the performance didn’t change. Then, I tried two more variations of PPO: first, I tried a shared critic network between the two agents that took in a concatenated observation vector. I hypothesized that a shared critic would help improve collaboration as the critic can judge the *combined* actions of both agents. This improved the convergence speed on coordination ring from 1810 to 1690 episodes. To push the limits, I then tried both a shared actor (a two-headed NN that outputs actions for both agents) and a shared critic, this improved the speed from 1690 to 1570 episodes. However, as shown in figure 6, DQN was still the best.

Figure 6: Comparison of during training performance of DQN & PPO variants on Coordination Ring.

Overall, DQN was more sample efficient and powerful than PPO as it solved the first three layouts using fewer episodes and nearly solved layout 5. Possible explanations for why DQN may be doing much better than PPO are: (1) the experience replay makes it more sample efficient as it can reuse and learn more from data – this might imitate the role of the hippocampus and dreaming in mammals; (2) DQN was designed for Atari games with discrete actions, making it a perfect match for Overcooked.

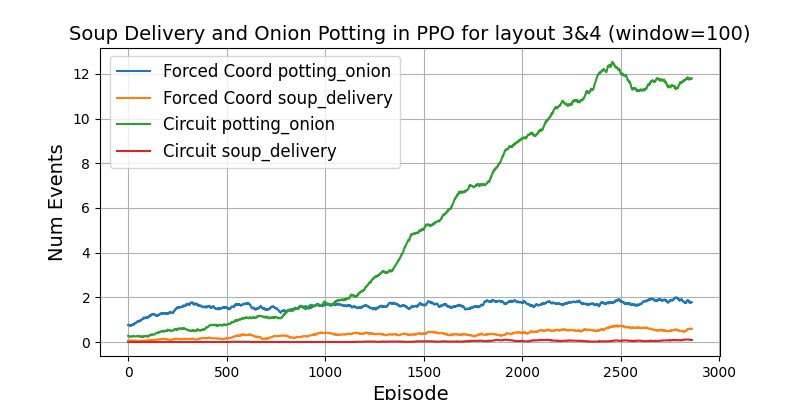
However, DQN still could not solve layout 4 and only nearly solved layout 5. Thus, as a last resort, I attempted *Curriculum Learning*. I fully trained the agents on coordination ring, and then started training on circuit using these same agents in the hope that they would not have to learn concepts like cooking and delivery from scratch. This too, however, did not work, and a possible explanation for this is: at the start of training the shaped and soup reward is zero as it is a new layout and the exploration rate is high, this might have destroyed the representations learned due to large losses and consequently large gradient descent updates. With more time, I would have tried freezing the initial layers of the network during the first phase of training, and then unfreeze them as training progresses. So, why might layouts 4 and 5 be so hard to solve? Well, figure 7 gives a clue: even after 3000 episodes, the PPO agents, just like DQN agents, have done very little onion potting, suggesting that the poor performance is more due to lack of systematic exploration rather than limitation of the algorithm or collaboration: they just couldn’t find actions leading to reward. There is definitely a mix of poor credit assignment too, but my hunch is that with intrinsic motivation the agents will do much better.

Figure 7: Num of soup delivery and onion potting events with PPO agents on layouts 4,5 is very low even after 3000 ep.

# **3. Conclusion**

I believe that I could have solved all the layouts had I just stuck with DQN and chose to implement intrinsic motivation for exploration, QMIX for credit assignment and further tuned the hyperparameters, instead of spending 2 weeks on learning, implementing and tuning PPO. However, keeping the learning objective of the course in mind, I would rather implement two popular algorithms, understand their intricacies, and not solve all layouts instead of implementing just DQN and solving Overcooked. The biggest challenge of the project was the lack of compute. I ran more than 60 experiments over 4 weeks, and the pain points were: (1) slow feedback loop due to ~2hr of training, (2) lack of intuition on the effect each hyperparameter had as I was new to the algorithms, though I developed a strong foundation by the end. With more time, I would have (1) repeated the experiments to remove noise due to random seed, (2) tried more values for each hyperparameter to gauge its impact, (3) tune PPO to match DQN.

**References**: [1] *Overcooked*; [2] Rada, Adit. *Project 2: Solving Lunar Lander Using DQN*; [3] Thomas Simonini. *Proximal Policy Optimization, Hugging Face Deep RL Class.*