Self-Organizing Multi-Agent Systems

Coursework 2022-2023

Inigo Selwood (CID: 01718768)

Concept

A number of peasants are stuck at the bottom of a pit. Above them stretch an unknown number of levels, each guarded by a monster. A certain number of peasants must emerge alive to win the game.

Each turn, the peasants can choose to attack the monster, help defend the group, or do neither. Should the monster survive the group's attack, it will deal damage to those peasants who fought (or all of them, in the case that none did). Choosing not to fight will restore one point of health.

Each turn, the group will gain experience points in proportion to how much damage they dealt and deflected, which the peasants can distribute amongst themselves. This experience can be used to increase their stamina, attack, or defence.

Process

This game presents a problem of shared resource management – the peasants themselves, their health and energy. Without cooperation and coordination, it's possible (if not likely) that they will run out of both.

The goal of this experiment is to show how, by structuring how the demands and rewards of the game are meted out, we can foster altruistic behaviour that ultimately improves the peasants' chances of survival. With the peasants to incentivised to work together as a group, they will hopefully cooperate in ways which consistently and reliably result in winning the game.

The expectation is that, before a system of rewards is introduced, the peasants will act in self-interest and die relatively quickly. To test this hypothesis, we're going to examine how the group fares by creating a simple game model and introducing a selfish agent.

This selfish agent will come in the form of a neural network model. Unincentivized to help the group, we should see a decrease in their performance. After introducing a system of incentives however, it's hoped that new patterns of behaviour will develop. The expectation is that, when rewarded for altruism, the peasants will be able to grow stronger and escape from a pit of any depth.

All code for this experiment was written in Python 3.10, and has been made available on GitHub. Instructions for reproducing results and figures can be found there, along with further detail about the artificial intelligence techniques used.

Peasants

Peasants are a 5-tuple of attributes:

$\boldsymbol{P.S}$	Stamina	Energy which can be used in / defending
$\boldsymbol{P.H}$	Health	A measure of their condition, where zero health means death
$\boldsymbol{P.A}$	Attack	Maximum amount of damage that can be dealt per turn
$\boldsymbol{P.D}$	Defence	Maximum amount of damage which can be deflected per turn
$\boldsymbol{P.a(S)}$	Action Function	Determines the peasant's action in each game state

Monsters

Monsters can be thought of as special cases of peasants, with infinite stamina and zero defence. They deal the maximum amount of damage possible on each turn, but cannot defend themselves.

$\boldsymbol{M.H}$	Health	A measure of their condition, where zero health means death
$\boldsymbol{M.A}$	Attack	Maximum amount of damage that can be dealt per turn
M.L	Lifetime	The number of rounds which the monster has survived

Action

Actions are a 2-tuple of attributes which, given the current game state, each peasant will produce once per round. They specify how much stamina the peasant is willing to give to the group's attack/defence.

$\boldsymbol{A.A}$	Attack	Stamina to contribute to the group's attack
$\boldsymbol{A.D}$	Defence	Stamina to contribute to the group's defence

State

The state set contains all the information about the game on a given turn:

$\boldsymbol{S.M}$	Monster	Monster being faced
$\boldsymbol{S}.\mathbb{P}$	Peasants	Set of living peasants
$\boldsymbol{S}.\mathbb{C}$	Combatants	Set of combatants (peasants who've chosen to fight, so far)
$\boldsymbol{S}.\mathbb{A}$	Abstainers	Set of abstainers
$\boldsymbol{S.A_\forall}$	Group Action	Action, the sum of combatants' attack/defence actions so far in the round

Game Model

The game is a state machine which maps between successive states. Using this turn-based deterministic model is key to being able to train our neural network, since it allows us to calculate exactly how beneficial any action is for a given state.

$$\mathcal{S}_{n+1} = \mathcal{G}(S_n)$$

Peasants are selected to choose their next action in a random order. This property allows peasants to make decisions based on the actions of those who've gone before them. Acting sequentially is conducive to the sorts of selfish behaviour we're interested in examining, because those who are chosen towards the end of the group have more information about the round's outcome than those who went at the start. By picking peasants in a random order, no peasant has an unfair advantage (on average).

$$P_{next} \overset{R}{\leftarrow} \boldsymbol{\mathcal{S}}.\mathbb{P} - (\boldsymbol{\mathcal{S}}.\mathbb{C} \cup \boldsymbol{\mathcal{S}}.\mathbb{A})$$

The mechanism for progression, as well as the mechanism we'll be using to reward altruistic behaviour, is experience. When the monster is dealt damage, or has its attacks deflected, it "drops" a certain amount of it. It's then up to the peasants to decide how that experience is handed out.

The amount of experience dropped is based on the *effective* damage the group dealt and deflected each round. The experience is also in proportion to the monster's total health and attack, which makes the reward greater, the stronger the monster.

$$A_\varepsilon(\mathcal{S}) \rightarrow \begin{cases} \mathcal{S}.A_\forall.A, & \mathcal{S}.A_\forall.A < \mathcal{S}.M.H \\ \mathcal{S}.M.H, & else \end{cases}$$

$$D_\varepsilon(\mathcal{S}) \rightarrow \begin{cases} \mathcal{S}.A_\forall.D, & \mathcal{S}.A_\forall.D < \mathcal{S}.M.A \\ \mathcal{S}.M.A, & else \end{cases}$$

$$E(\mathcal{S}) = \left(\frac{A_\varepsilon(\mathcal{S})}{\mathcal{S}.M.H} + \frac{D_\varepsilon(\mathcal{S})}{\mathcal{S}.M.A}\right) * (\mathcal{S}.M.H + \mathcal{S}.M.A) * \varphi$$

In the absence of a reward distribution function, the tuning factor $\boldsymbol{\varphi}$ is what determines the "difficulty" of the game. Values less than 1 will tend to decrease the group's average stamina, and vice-versa.

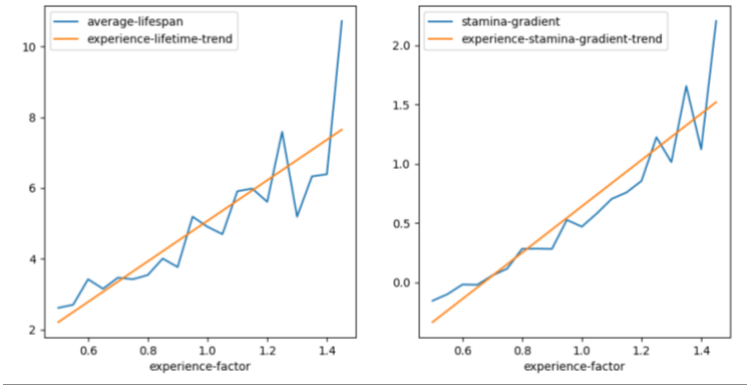
Tuning

To prepare the game for introducing an intelligent actor, it must be balanced first; this means tuning the game's parameters until a "fair" pattern emerges. Since the game model is going to be used for training our neural network, we want it to be unwinnable without showing some sort of strategy. With peasants acting randomly, the following should be true:

- The group dies, eventually
- Some monsters are killed
- The average peasant level (sum of stamina and health) is linear across the game

What we'd expect to see given these criteria, is a bounded average lifespan, and a flat stamina gradient. The average lifespan should be bounded because every member of the group should die at some point. The stamina gradient should be flat since we don't want them to be getting stronger without our incentives for shared resource management.

This might seem like we're fixing the game to show the pattern we want – ie: that cooperation is beneficial. However, remember this in the context of our game's rules: we're fighting monsters, not weak gremlins. In other words, we're adhering to the spirit of the game, in that the peasants must face a difficult enemy.



[**Figure 1:** Effect of Varying Phi on Gameplay]

Figure 1 was obtained by running 5 episodes (complete games) for 50 different values of phi. Looking back at our experience function, it's this value which determines how difficult the game is. Larger values of phi result in stronger peasants – as reflected by an increased average lifespan, and an increasing amount of stamina over the course of the game.

By inspection we want a value of phi in the region of 0.7; this meets our expectations of a fair game, in that the group dies having killed some monsters while maintaining a stable average level. We can deduce this from the fact that the average lifespan is still linear in that region, and the stamina gradient is near-zero.

Training

We're going to use a Double Deterministic Policy Gradient (DDPG) algorithm to learn to play our game, so that we can effectively measure how our reward system is impacting its outcomes. Python comes with a gamut of machine learning algorithms in the form of the Tensorflow + Keras packages, which can be used to create a neural network of this type.

DDPG is a reinforcement-learning (RL) model, which means it can teach itself the most appropriate strategy for play. It does this by trying to maximize the reward it receives from its actions. By combining what it thinks is the best action with a degree of random noise, it can effectively explore the state and action space, hopefully finding the global reward maximum.

There exist a dozen common RL algorithms, notably Deep-Q Learning (DQ) and the Cross-Entropy Method (CEM). However, these algorithms require discrete bounded action and state spaces. Our game's state space is *not* bounded, since (ideally) the group's stamina and the monsters' difficulties could increase with no end. It's also much easier to implement continuous attributes than discrete ones, and continuous values allow us to train with a richer information set. DDPG meets these criteria, so it's the algorithm we'll be using.

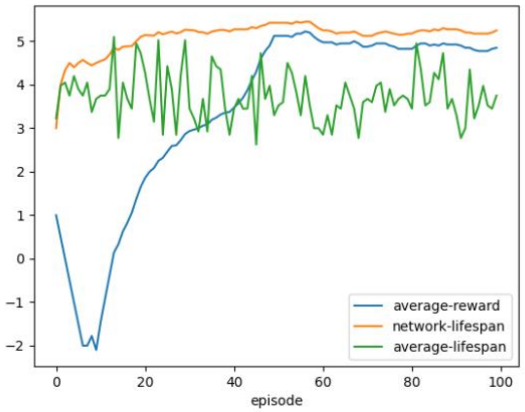
The model itself uses something called the Actor-Critic (AC) method, whereby two networks compete in suggesting an action and estimating its reward. The loss function (that which helps the network evaluate its performance) examines the difference between the expected and actual reward, and "remembers" those actions which performed well.

There exist a number of hyper-parameters which can be varied to potentially maximize the network's performance. Some of them include:

- The learning rate α, a proxy for how readily the network changes its policy based on new information
- The discount rate γ, which determines how eagerly the network seeks out potential future reward
- The adoption rate τ, a measure of how quickly the network alters its weights
- The greedy factor ε, the proportion of actions taken randomly to potentially discover more rewarding policies
- The structure of the network itself, including activation functions and layer cardinalities

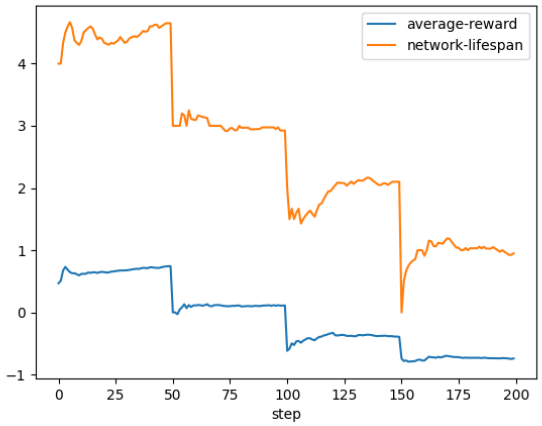
Traditionally these hyper-parameters are tuned in a separate process to the training. However, this isn't worthwhile unless peak performance provides a particular advantage, or the network fails to converge.

Figure 2 shows how the network trained to maximize its reward, and the increase in lifetime (number of monsters defeated) over the course of that process. You can see that, while the network and the randomly acting peasants start off with similar lifespans, our intelligent actor quickly begins to out-perform them. This initial similarity is to be expected, as the network's behaviour at the start is random as well. As the reward function converges, this increase in lifespan plateaus, as was hoped would be the case for a "selfish" actor.



[**Figure 2:** Network Training Convergence]

You may notice that the reward is, for a period of time, negative – this is common strategy in RL, whereby both positive and negative reinforcement are used to help encourage a certain outcome. In our case, we want to penalize the network for loosing stamina and health, as well as for the time it takes to defeat a monster. Only when it makes *gains* in those areas does it receive positive rewards.



[**Figure 3:** Introduction of Selfish Agents Across 200 Training Episodes]

Figure 3 shows what happens when we increase the proportion of selfish to random agents in the group. By taking the trained weights from the random environment, and instancing new peasants with those properties, we can effectively increase the number of peasants who act in self-interest. As you can see, while the network makes slight gains in performance, its lifespan drops off a cliff-edge with each new selfish agent introduced. This is a great example of how selfish action detriments the group as a whole.

Rewarding Cooperation

We now have a functional game, an intelligent actor, and the goal of consistently achieving a win. A sneaky way of accomplishing this might be to make the network's reward function sensitive to its comrades dying – but this isn't reflective of the socially beneficial systems of government which we're trying to simulate. Instead, we're going to play with the experience distribution function as a proxy for responsible behaviour.

Much like how our network training reward function penalises poor performance and encourages the good, we want to give experience to those most deserving. Some of the behaviour we're looking to encourage might include:

- Acting when it's likely to save others
- Acting early
- Acting generously

And on the flip-side, behaviour we want to discourage:

- Not acting when you'd be helpful
- Only acting when you're guaranteed safety
- Acting stingily

Lastly, there are some special circumstances:

- Not acting when you're low health or stamina
- Not acting when the outcome's already decided

The implementation will calculate a reward for each peasant's action, and when it comes time to hand out experience, it'll be as a ratio of those rewards. The more positive behaviours the peasant exhibits, the larger the share of the experience they'll receive – and vice-versa.

We define our reward function to evaluate some human-like analogues for the way the peasant acted. We want the reward to be a sensible combination of these attributes, which doesn't disproportionately reward any of the behaviours. To achieve this, we bias the criteria with weights.

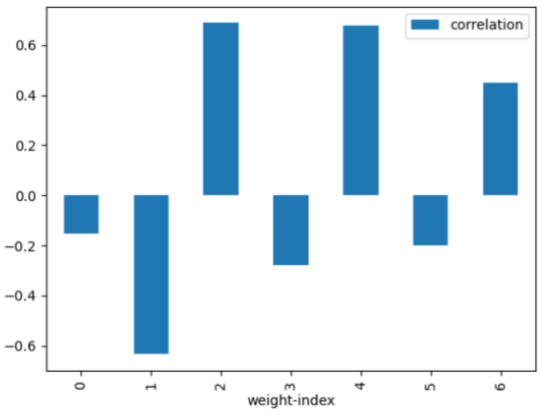
```
def reward(peasant: Peasant,
        action: Action,
        state: State,
        weights: tuple) -> float:

    # Metrics used below evaluated here from state/action
    # ...

    # Your actions directly saved the group or killed the monster
    heroic = killed_monster or saved_group

    # You acted early-on to make things easier for others
    early = acted and completion_percentage < 1 / group_size

    # You contributed when your health was low
    brave = acted and (health_low or stamina_low)

    # You acted generously
    generous = (not monster_killed
            and not group_saved
            and (action.attack + action.defence) >= 0.5 * peasant.stamina)

    # You're not in a position to contribute
    weak = health_low or stamina_low

    # You didn't act when your actions could have saved the group
    selfish = ((not monster_killed and monster_killable)
            or (not group_saved and group_saveable))

    # You acted when you need not have
    foolish = ((action.attack and monster_killed)
            or (action.defence and group_saved))

    # Pair, weight, and sum the criteria
    criteria = [
        heroic,
        early,
        brave,
        generous,
        weak,

        -selfish,
        -foolish
    ]

    assert len(weights) == len(criteria), "weight dimensions invalid"
    pairs = zip(criteria, weights)
    weighted_criteria = [weight * criteria for (weight, criteria) in pairs]

    return sum(weighted_criteria)
```

Optimizing

The last step in completing our game is to optimize the weights for our experience reward function. In an ideal world, we'd use multivariable analysis to optimize the peasant's life expectancy across a range of weights. We have 7 tuneable weights, and might want to try 8 different values in the range $[0:1]$ – but that would mean running nearly 6 million simulations!

Remember that we're now running the game with 10 "intelligent" neural-network agents. At a mere 10 episodes per data point, each simulation now takes an average of 3.2 seconds. We'd have to wait nearly 7 months to get our results.

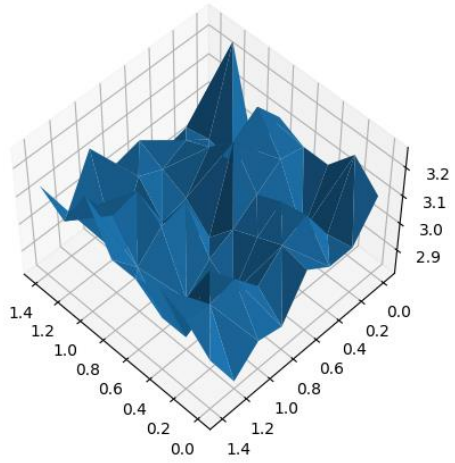
So instead, we'll break the optimization up into two stages. First, vary each weight individually, and extract a correlation score. Once we've found the two weights which most positively influence life expectancy, we can run a multivariable simulation on them, and hopefully find a global maximum. This has the added benefit of allowing us to produce a cool surface-plot.



[**Figure 4**: Weight Index to Lifetime Correlation]

Figure 4 shows the results of the correlation experiment. The weight indices which most positively impact group life expectancy correspond to the `early` and `generous` attributes. Curiously, being `heroic` has a strongly negative impact on group longevity. The implication here is that acting quickly and aggressively are advantageous, while dealing the death blow is actually a bad thing.

This last point is curious. One theory is that this occurs because, when the group is down to just a few members, the average peasant is more likely to be responsible for killing the monster – but also to die soon after. The network training penalty for this will be proportionally larger, making it appear as though that attribute has a negative impact on life expectancy.



[**Figure 5:** Lifetime as a function of `early` and `generous` weights]

The results from our multivariable analysis were disappointing. Figure 5 shows a noisy distribution with no regions that tend to infinity (which we'd expect to see if the peasants were able to keep fighting forever). In fact, the average lifetime (shown on the z-axis) has actually *decreased*, relative to what we saw in Figure 2.

This casts doubt on the validity or usefulness of our correlation experiment, since if our lifetime predictions were actually just noise, the weights were irrelevant to begin with. At least it might explain why we saw negative correlations for attributes which seem logically helpful for the group's survival, like being `heroic`.

One reason for these surprising results could be that the network was never properly weaned off-of its experience with randomly acting peasants. It's possible that, despite continuing the training with intelligent actors in our system of social reward, it failed to learn a new style of play. This is likely because we greatly reduced the number of episodes used for each data point.

This reduction in episodes was done to allow the simulation to complete in a reasonable amount of time. An epoch (episode count) of 50 was chosen by inspection of Figure 2, since that's the region at which we began to see convergence in the reward function. However, it's clear that the network didn't manage to fit its probability function as well under these new conditions.

Another explanation is that our experience reward function is too complicated for the network to fit. The network may simply need to be larger to learn it. That having being said, size of our DDPG network isn't modest, at over 7,500 trainable parameters; on balance of probability, it's more likely that it just didn't receive enough training time.

Conclusion

In this experiment we managed to show that selfish behaviour in a resource-limited environment necessarily decreases the time for which a group can rely on that resource. Looking back at Figure 3, the more selfish actors were introduced to the group, the faster the peasants died. This shows that, without systems in place to reward altruistic behaviour, the game cannot be won. We also examined how, in theory, the peasants can be incentivised to cooperate by tailoring the distribution of a limited resource (experience) to their actions.

Because our simulations of the game under those reward-governed conditions failed to show a pattern of improvement, it's not possible to draw conclusions about how effective they might have been in the context of our game model.

Were the experiment to be repeated, and certainly given more time, it's possible that careful training of the peasant neural network could yield different results. There's certainly cause to believe that our model was responsible for our poor set of results, and this doesn't rule out the possibility of our hypotheses being proved.