Analysis on Reinforcement Learning Methods on Navigation in a Dynamic Environment

Callum Hendry: 100970932
callumhendry@cmail.carleton.ca

Jason Dunn: 101140827
jason.dunn@carleton.ca

Ujan Sen: 101171605
ujansen@cmail.carleton.ca

Uvernes Somarriba: 101146733
uvernes.somarribacast@cmail.carleton.ca

1 Introduction

2 Methods

2.1 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) was introduced in 2017 [1] and has achieved widespread success in the realm of reinforcement learning. It was a successor to Trust Region Policy Optimization (TRPO) introduced in 2015 [2] which used stochastic gradient ascent by using a trust region constraint to regulate between the old policy and the new updated policy. PPO is considered state-of-the-art and is the default reinforcement learning algorithm used at OpenAI because of its ease of use and significantly better and faster performance than its counterparts.

The biggest difference between PPO and TRPO is that PPO uses a clipping function which effectively ensures that the new learned policy does not deviate more than a certain amount from the old policy. This is done by first calculating two different sets of surrogates for the policy network. The objective for PPO is calculated as follows [1]:

$$L_\theta = \mathbb{E}_t[min(r_t(\theta)A_t, CLIP(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)]$$

For the surrogates, the ratios are first calculated between the predicted actions at the current state given the old policy and the new predicted actions. The first surrogate is calculated by weighting the ratios by the calculated advantages. The second surrogate is calculated in a similar way except the ratios are first clipped between 1 - ϵ, and 1 + ϵ where epsilon is defined as the clip ratio, and then weighted with the advantages. The loss is then defined to be the minimum of these surrogates.

Advantages are calculated using Generalized Advantage Estimation (GAE) introduced in 2015 [3]. GAE is an improvement upon traditional advantage estimation methods because of the introduction of the λ term which allows a trade-off between bias and variance. A λ of 0 reduces GAE to a one-step estimation which is just standard advantage estimation whereas a λ of 1 considers rewards infinitely into the future. It takes into consideration the temporal difference of not only the immediate rewards but also the expected future rewards. The formula for GAE calculation is as follows [3]:

$$\hat{A}_t^{GAE} = \sum_{k=0}^{\infty}(\gamma.\lambda^k).\delta_{t+k}$$

- $\hat{A}_t^{GAE}$ is the GAE at time step t
- γ is the discount factor
- λ is the GAE parameter, the tradeoff between bias and variance
- k is the time step offset

- δ_{t+k} is the advantage at time step $t+k$ calculated using the one step standard advantage calculation

The intuition behind choosing PPO is the same as TRPO: "being safe". Since the loss function is defined as the minimum of the two surrogates, the objective becomes a lower bound of what the agent knows is possible. It approaches the task being a pessimist, which has often times proved to be more beneficial than being optimistic with little chances of recovery. TRPO's objective function achieves a similar thing but is quite different in computation [2].

$$L_\theta = \mathbb{E}_t[D_{KL}(\pi_{\theta_{old}}(.|s_t)||\pi_\theta(.|s_t))] \leq \delta$$

The benefit PPO gives over TRPO is the usage of clipped ratios and the surrogates. TRPO enforces a strict trust region constraint where the KL-divergence between old policies and new policies is small enough, within a parameter δ leading to a second-order optimization problem [2][4]. PPO effectively does the same thing using the clipped ratio and taking the minimum of the surrogate losses resulting in a first-order optimization problem. This leads to PPO utilizing fewer computation resources while providing better results.

The PPO implementation was performed using the library stable_baselines3 available here. It ran with default arguments as described in the documentation.

2.1.1 Environmental Considerations

The environment was modified and the results for each modification of the environment is mentioned in the Results section.

- The targets were reduced from multiple to single.
  - Relative directionality from agent to target introduced using a 4-element 1D array
- Running traffic lights, making u-turns, and going out of bounds were always penalized instead of it being probabilistic
- All penalized actions penalized to the same amount

2.2 Planning Meta-Agent

In this design architecture, an agent consists of two parts. First, it has an agent trained for navigating from a given state to another given state with the maximum reward. However, this agent alone has limited usefulness as it does not have any reasonable way to handle longer runs with multiple targets. To solve this problem another component is added to the agent which is specialized in forming long term plans. Essentially, these components treat the environment as a fully connected graph, and it is able to go from any node to any other node with a single action. It is able to do this as the designer knows that the navigation agent is able to move from any node to any other node in some number of actions.

Several different methods were experimented with to try to create a pathing component. It should be noted that these are not restricted to reinforcement learning, as it is worth considering what other options may offer better performance, as reinforcement learning is not suited to all tasks.

2.2.1 Q-Learning

In this approach standard Q-Learning is used based on a complete graph, with the reward going from any node to any other node being equal to the reward obtained by the navigation agent going between those two nodes. For efficiency two abstractions are made: first, this algorithm only tries to solve the travelling salesman problem for the target nodes, as it is not relevant to plan to visit the others, and second it creates a single reward matrix by running the navigation agent several times, then it simply re-uses those rewards.

Unfortunately, using the maximum at each step may not end up creating a complete cycle. Thus, the designer must choose some way to select which actions are taken based on the Q-values. Obtaining the true optimal choices from the Q-Values would involve solving the travelling salesman problem on the values themselves, which effectively defeats the purpose of using this method.

2.2.2 Gradient Descent Neural Network

The main difficulty that this method faces is the lack of a clear differentiable loss function, unless one has access to the true outputs they are aiming for. Thus, in order to use this approach one would have to implement a different method first, which defeats the purpose of using this algorithm. The only practical use of this could be to reduce computation time by training it to mimic a good heuristic algorithm.

2.2.3 Step-Wise Neural Network

This approach uses a neural network that takes in the current state, and outputs a softmax probability output of which node the navigation agent should target next. It is possible that this may give good performance eventually, however due to the lack of a meaningful gradient, the only feasible way to train this is to use evolutionary methods.

2.2.4 Budinich Neural Network

This method was an early attempts at using the self-organizing properties of neural networks to produce a single cycle path on some number of points on a 2D plane. It theoretically achieves this by mapping each input to a point on a ring of output neurons. In order to create a full path it takes in the euclidean position of each node one at a time and then maps it somewhere on the real number line. This method produces better than random results, however the performance is not as good as other methods. In addition, it is unclear how one would generalize this method to non-euclidean spaces.

2.2.5 Evolutionary Mapping Neural Network

In this approach a neural network is trained to give each node a point on the one-dimensional number line, where it is trivial to solve the travelling salesman problem. In order to do this, it takes in the entire observation given by the environment, and outputs one decimal number per node. These are then sorted and used as the path. This method would eventually give extremely good results as it can converge to a global minima given enough time. However, the main drawbacks of this method are related to its time performance. Evolutionary methods are extremely slow to converge, and it would need to be retrained whenever any aspect of the environment is changed, such as adding or removing a node.

2.2.6 Christofides-Serdyukov Algorithm

Although this algorithm is typically very effective, it is not well suited to this particular problem as the environment does not meet the requirements to be considered a metric space. Namely it violates the following conditions: First, the distance from any node to itself is not zero, as there is a penalty for idling. Next, there is no assumption of symmetry. Finally, there is no guarantee that the triangle inequality will hold depending on the quality of the underlying navigation agent. Thus, the usage of this algorithm would be restricted to the Euclidean distances, which misses significant amounts of nuance in the environment.

2.2.7 Nearest Neighbour

This is essentially the simplest method one could use to obtain a reasonable path. It works by starting with a random node and constructing a complete path by iteratively selecting the nearest node that has never been visited as the next node in the path. It has no performance guarantees, however it should perform reasonably well on most inputs. This method is simple to adapt for our meta agent, as we can construct a distance matrix between every pair of nodes.

2.2.8 Ant Colony Optimization (ACO)

This method uses a large population of simple agents. For each agent, they start at a random node, and need to complete one full cycle. Then they apply a value of $\frac{1}{cyclelength}$ to the action it took along each step of its path. Then, future agents are able to use this pheromone trail to probabilistically select the next node they go to. Since lower cycle lengths give a higher value, the ants will eventually produce a good quality approximation of the shortest path using a time proportional to the number of agent iterations. This produces very good results, and works well in the non-euclidean space as it solely relies on cycle lengths, and not any other property of the environment.

Overall, although it may be possible to train a planning agent using reinforcement learning, in practice hand crafted algorithms are still able to achieve superior results.

3 Results

In this section we will discuss the results for the various implemented algorithms:

Table 1: Results for PPO

Target	Reward Type	Reward Scaling[1]	Agent Score[2]	Random Score[2]	Train Time[3]
Single	Deterministic	Same	875	-396	15
Single	Deterministic	Different[4]	935	-369	15
Single	Probabilistic	Different	890	-334	20
Multiple	Deterministic	Same	205	-405	20
Multiple	Deterministic	Different	379	-390	20
Multiple	Probabilistic	Different	255	-410	30

4 Discussion

From the results, it is clear that for PPO, the more complex the environment becomes, the worse the agent performs. This is expected as it is quite difficult to find an optimal solution to our problem considering how the environment behaves. When there is a single target, there does not seem to be large difference in performance. The higher score for different reward scaling compared to same can be attributed to the fact that when it makes mistakes, the penalty is lower in the different reward scaling scenario, which is coincidentally also the case for the multiple targets scenario. It is worth noting that the total score for multiple targets is difficult to say since the number of targets is not fixed but it is more than 1000 since there is always at least one target.

An interesting idea would be to fix the traffic lights instead of randomly initializing them. This seems quite intuitive and in fact, could even be considered closer to reality since if the environment is a representation of an urban city setting and we consider that it is the same city, traffic lights and their locations will not suddenly change. And even if they do change, they should remain consistent for a significant time once they do. Perhaps this is an avenue worth exploring where traffic lights are initially fixed but can randomly change after a certain amount of time steps.

In both the single target and the multiple target scenario, the agent performs worse than the best agent when penalties are probabilistic which makes sense since this is a difficult environment dynamic to learn. Whenever probabilistic elements are introduced into the environment dynamics, it makes training significantly tougher since even though it might revisit a state and take the same action as it did last time, it might receive a different reward.

Furthermore, it is worth noting that multiple targets yield worse results than singular targets. This is expected because of the fact that when there are multiple targets, there is no way to encode direction, which was something we were able to do when there was a single target. For example, if there was a single target, we had relative directionality indicating where the target was. But when there are multiple scattered targets, this sort of information is difficult to encode.

Given more time and resources, we would like to implement the planning agent that works in tandem with our navigation agent. This could be achieved via the various methods outlined under 2.2 with the most promising ones being Nearest Neighbour 2.2.7 and ACO 2.2.8. One could argue that an

[1] Only negative rewards are altered, positive rewards remain the same

[2] Scores are out of 1000 for single target

[3] In minutes

[4] Stop at green: -2; Stop at node that is not traffic light: -2; Everything else: -5

evolutionary mapping neural network 2.2.5 would also be effective to use on top of our navigation agent because of its eventual convergence to a global minima, but training it is non-trivial.

Thus, the optimal way to solve our environment would be using a good planning agent that picks the next target for our navigating agent that has been trained using PPO, effectively reducing the state space from having multiple targets to a single target once again, allowing us to encode direction again. We can say with confidence that this would be optimal since this is pretty close to how humans would operate with this environment and furthermore, seeing how well PPO performs on the single target scenario, we can say with high certainty that this will indeed provide optimal results.

Given more time and resources, we would also like to test out more scenarios including various combinations of the scenarios mentioned in 2.1.1 along with 2.2. We would also like to test various reinforcement learning algorithms and their performance if the targets had priorities assigned to them. We would also like to add pedestrians and random events in the environment and evaluate performance.

References

[1] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal Policy Optimization Algorithms. arXiv preprint arXiv:1707.06347.

[2] Schulman, J., Levine, S., Moritz, P., Jordan, M. I., & Abbeel, P. (2017). Trust Region Policy Optimization. arXiv preprint arXiv:1502.05477.

[3] Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2018). High-Dimensional Continuous Control Using Generalized Advantage Estimation. arXiv preprint arXiv:1506.02438

[4] Wang, Y., He, H., Wen, C., & Tan, X. (2020). Truly Proximal Policy Optimization. arXiv preprint arXiv:1903.07940.