Evolving Reinforcement Learning Algorithms (Joint Project INM707)

Eduard-Ernest Pastor S180051071; Fei Phoon, S190032928

This project was a joint effort between Eduard-Ernest Pastor and Fei Phoon for the module INM707 Deep Learning Optimisation, led by prof.Michael Garcia Ortiz. Our report is structured in 4 parts as follows:

1. **Task 1**: the environment Maze and Qmaze : → 10% of code borrowed
2. **Task 2:** Q-learning: → 20- 25 % of code was based on the class implementation of TD and MC methods on FrozenLake environment
3. **Task 3:** A2C on CartPole-v1: → 50%
4. **Task 4:** Test A2C best model: → 50%

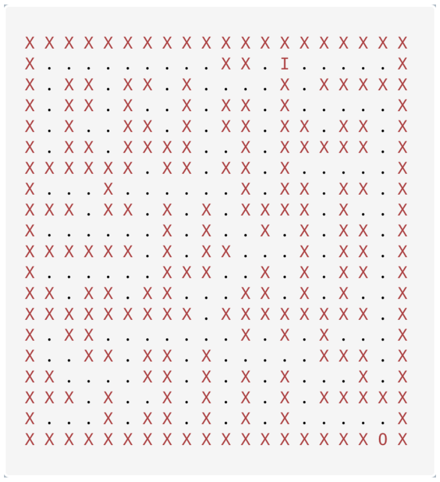
Personal Reflection: This was the first time I worked in a team for an entire module. It was both challenging and a rewarding experience for me. We both knew each other, and we are both PT2 students. Fei has tried to push us from day one to conduct, build and present a quality piece of research, and both of us have dedicated a high amount of hours to achieve this. The main difference between her and me was the background, as she is a data engineer and a CS graduate. This was a challenge as I wasn't that familiar with OOP(which I believe is a must-know requirement for this module) as well as working collaboratively and knowing GITHUB well. I can say we worked well together and I was able to learn not just OOP, but also RL concepts, GITHUB and other software engineering techniques that are being used in the day-to-day work environment. This was the most complex module I have taken, and I am grateful I did, as I learned a lot during a concise period. I believe the coursework requires a lot of technical knowledge from students. I feel this should be emphasised more before joining the course. In terms of building the project, we both contributed to the research phase, coding, running experiments and writing up the report. Our primary channel of collaboration was DISCORD. I would recommend that future students use that, as in our case it made our life easier. Another special software we used was VS CODE which helped us both, as writing in ANACONDA things can break easily, and GITHUB helped us keep track of the commits and versions of our code. The latter was beneficial when our codebase became more extensive and harder to track with both of us doing remote changes. I am grateful I was able to work with Fei and I think pairing was a great idea. One thing I could mention is pairing maybe different backgrounds (technical and less technical) in such a way this will push the less technical to learn a lot more and the more technical to become a better teacher, thus learning more. I hope the reader is pleased with the coursework and I want to thank the module leader for the Reinforcement Learning module.

Task 1: Maze (and QMaze)

Across this coursework we are tasked with implementing a tabular reinforcement learning method (done individually as Task 2) and later exploring a deep learning method to solve a continuous environment challenge.

Tabular reinforcement learning methods offer simple and easy-to-understand approaches to solving problems with finite environments. As tabular methods rely on being able to represent some measure of value of being in different areas of the environment to be solved, the state and action spaces of said environment must be representable in a tabular or array *data structure* [1, pp. 23] - emphasis on data structure as this dictates size or memory limitations.

For this task we decided to create a slightly more challenging variation on the Dungeon environment used in this module's labs. This environment's states are easily mapped to a **lookup table**, allowing a sequential decision making process for step selection or examination - the **Markov Decision Process**.
Our Maze is a gridworld-type environment that is viewable in a "toy text" format, with a single agent and single objective. It has a configurable size, which produces a square environment of the given size, with a randomly-generated, guaranteed-solvable maze based on Prim's algorithm[4]. In brief, an "unterraformed" square space of the specified size is spawned; a random point in the space is selected to start generating the internal maze walls from; odd gaps are filled in as walls and the bounding walls are added. An entrance (agent starting point) and exit (agent goal point) are randomly placed in the second-to-top row (I, on the left) and the rightmost possible cell in the bottom-most row respectively (0 on the left). The maze generation code was rewritten based on guidance from a public Github repository in [5] but with better abstractions, clarity, robustness and no bugs. Some test code was written as a sanity check on producing the environment.



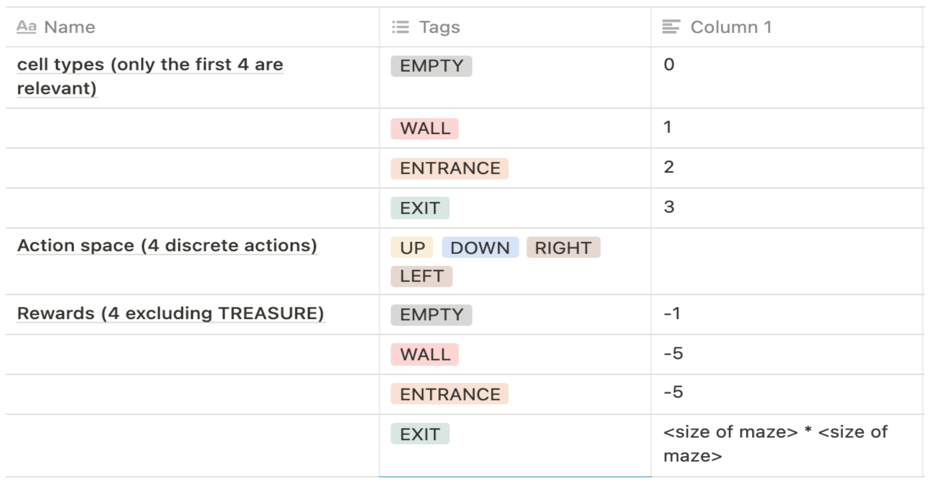
```
X X X X X X X X X X X X X X X X X X X X
X . . . . . . . . . X X . I . . . . . X
X . X X . X X . X . . . . X . X X X X X
X . X X . X . . X . X X . X . . . . . X
X . X . . X X . X . X X . X X . X X . X
X . X X . X X X X . . X . X X X X X . X
X X X X X X . X X . X X . X . . . . . X
X . . . X . . . . . . X . X X . X X . X
X X X . X X . X . X . X X X X . X . . X
X . . . . . . X . X . . X . X . X X . X
X X X X X X . X . X X . . . X . X X . X
X . . . . . . X X X . . X . X . X X . X
X X . X X . X X . . . X X . X . X . . X
X X X X X X X X X . X X X X X X X X . X
X . X X . . . . . . . X . X . X . . . X
X . . X X . X X . X . . . . . X X X . X
X X . . . . X X . X . X . X . . . X . X
X X X . X . . X . X . X . X . X X X X X
X . . . X . X X . X . X . X . . . . . X
X X X X X X X X X X X X X X X X X X 0 X
```

Each cell in this Maze has one of the states in (Fig.1, *Cell types*); the ones currently of consequence to the agent are: Empty, Wall, Entrance and Exit. On reset the maze is kept as it was first created at instantiation, as well as the exit; the entrance and therefore the agent starting point is placed randomly again in the second-to-top row.

The environment's action space (fig.1, *Action space*) comprises four discrete one-step actions of up, down, left and right. The rewards earned by an agent based on a cell state are shown in (fig.1, *Rewards*) - we take the advice of Sutton et. al [1, pp. 53] to track agent progression by "timesteps" (an arbitrary representation of time), each earning the agent a -1 reward, to motivate a quick as possible escape from the maze. Otherwise the agent incurs a -5 reward if it chooses to walk into a wall, with no movement in any direction; earns a similar -5 reward for returning to the Maze entrance, and a large reward amounting to the power of the size of the maze for discovering the exit.

The observation space returned at the end of every movement, comprises agent distance to exit and a capture of the surrounding neighbour cells' states. Lastly, the only condition to terminate an episode is if the agent reaches the exit. Therefore the agent's quickness to reach the exit is reflected in timesteps taken.

The structure of the Maze is generic to reinforcement learning environments, in that it is centred around a `step()`, `reset()` and `display()`. The `display()` method was borrowed, with some small tweaks, from this module's lab Dungeon code, as it suited the display needs of the environment perfectly.
We built the requirements needed for **Task 2** as a separate class QMaze inheriting from Maze, in case we needed `step()` to return the original observation space for a future task. The `step()` and `reset()` functions in QMaze return a state instead of an observation. The QMaze class adds stochasticity after the fashion of the probability of slipping used in this module's Dungeon example - the agent has a 0.6 chance of moving to the cell it intends to, but in the event it does not, it slips to another adjacent cell and incurs the reward attached to that cell's state.
There were some plans to introduce some other random elements, such as short-term rewards or distractions from the most valuable goal, in the form of randomly-scattered treasure with small returns, and a negative reward for the agent doubling back on itself on a path already explored. You will spot these elements in our code, but they may not be fully implemented or used for **Task 2.**



Name	Tags	Column 1
cell types (only the first 4 are relevant)	EMPTY	0
	WALL	1
	ENTRANCE	2
	EXIT	3
Action space (4 discrete actions)	UP DOWN RIGHT LEFT	
Rewards (4 excluding TREASURE)	EMPTY	-1
	WALL	-5
	ENTRANCE	-5
	EXIT	<size of maze> * <size of maze>

Observation space (coordinates tuple, 3x3 array of cell states)	`Observation = namedtuple("Observation", ["dist_to_exit", "neighbours"])`
	
	`Observation(`
	`  dist_to_exit=(8, 5),`
	`  neighbours=array(`
	`    [[1, 1, 1],`
	`     [4, 2, 0],`
	`     [1, 1, 4]]`
	`  )`
	`)`

Terminating conditions	Episode ends when the agent reaches exit; no maximum number of episodes or timestep limit.

Fig.1: Cell types, Action space, Rewards, Observation space & terminating conditions for Maze and QMaze.

This environment is generated fairly quickly up till size 500 (250000 cells). See [6] and [7] for their implementation.

Task 2: Q-Learning

Q-learning is a popular baseline model that is based on the off-policy TD control algorithm. We can denote Q-learning in its simplest form as:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_{\alpha} Q(s_t + 1; a) - Q(s_t, a_t)] .[1]$$

In this case, the learned action-value function-Q, directly approximates the optimal action-value function, independent of the policy being followed. In our case:

$$Q(s, a) = \textit{the maximum total reward we can get by choosing action a in state s}$$

As a result, this will help simplify selecting an action even if the action was not optimal, the information is not included in the q-lookup table.

As a result, even if the action selected in the next state was mediocre, the information was not included in the updating of the Q-function of the current state. The $Q(s, a)$ is updated using the sum of the existing value $Q(s, a)$with the action that optimises the current state. Using Bellman's optimality equation stated, the *q-values* are being updated continuously. In our maze the rewards are presented in a Q-lookup table.

Once we know $Q(s, a)$ finding a policy is easy.

$$\pi(s) = max_{i=0,0...n-1} Q(s, a_i)$$

Q-learning combines dynamic programming and Monte Carlo methods, which have been used to solve the Bellman equation.

For **task 2**, we will implement Q-learning on an extension of the original environment called *Qmaze*.We will use our *Qmaze.py*,which is a class that inherits from *Maze.py* and returns a state instead of an observation. The task is to train our agent to navigate the maze using the most efficient route. We are using ***Epsilon-Greedy Policy*** to allow us to store and compute the *q-values* for each state using each experience to update an exponentially weighted average *forget that experience*, see fig2.1 for detailed implementation.

Experiment Setup.

Decay = 0.999 / Alpha = 0.1 / Gamma = 0.9

Maze Size per 2000 episodes	Average time steps per episode	Average rewards per episode	Training time
15	28.519	132.506	16.68
20	43.891	257.7375	31.06
30	79.324	638.112	86.48
40	113.7745	1227.969	376.33

Fig.2.1 Q-learning Experiment results (policy evaluation)

Decay = 0.999/ Alpha = 0.01 /Gamma = 0.8

Maze Size per 2000 episodes	Average time steps per episode	Average rewards per episode	Training time
15	57.71	42.42	37.06
20	56.15	220.70	178.76
30 (500 episodes)	8816.48	-34227.82	589.52
40 (500 episodes)	41764.41	-163114.16	10912.74

Fig.2.2 Q-learning Experiment results (policy improvement)

For *Q-learning* we used the setup presented in the fig.2.1 and 2.2 which allowed for the experiments to be focused, reducing overestimation,considering CPU times and environment complexities. The scores were averaged per episode. In the first set of experiments we can observe a stable scalability in the environment with increasing training times but stable *mean rewards* and *average timesteps*. Notably we can observe that the agent is quite efficient in optimising route even if maze if scaled. $\varepsilon = 0.999, \alpha = 0.1, \gamma = 0.9$ has a positive effect in reducing exploration during learning, thus minimizing overestimations and penalties. However,

when we changed the parameters of the *e_greedy_policy* $\varepsilon = 0.999, \alpha = 0.01, \gamma = 0.8$ discounting the future rewards more than immediate rewards and encouraging exploration, we can see an immediate effect in increasing training times, lower *average rewards* and increased time steps especially in ex.2 maze 30/40 size where the timesteps increased exponentially even at ⅓ of the total episodes.The training times are higher considerably scaling the maze and the *average_rewards* are negative due to our new *epsilon_greedy* that pays the price of exploration and converges at a sub-optimal point, see.fig 2.2 for details.
As we see in the experiments, Q-learning is computationally expensive and parameterizing the environment has a serious effect on computation, especially considering training was done on a CPU. Our agent was able to solve the maze, but the *exploration-exploitation* was an issue when we changed the *e-greedy policy* parameters and the agent was penalised due to overoptimism based on estimation errors. However,our task was successful in scaling the environment, giving us the chance to test how Q-learning can be used as a baseline for better more efficient models, like DQN and Double DQN[22].

Task 3: A2C on CartPole-v1

A brief explanation of Advantage Actor Critic (A2C)

So far we have compared the value of taking particular potential actions while being in a certain state, by calculating the projected discounted reward of each. But it seems intuitive to learn and optimise a policy directly from our experiences (INM707 Lecture 9). We might instead consider Q-values as the sum of the state-value and the advantage of a combination of state and action: *Q(s, a)* = *V(s)* + *A(s, a)* [2, pp. 315], and use the advantage to manage our policy gradient, atop a baseline state-value. As we don't know how much of *Q(s,a)* is attributed to either however, we employ another player (where in tabular methods, our only player was the agent) to approximate what the state-value is for every observation. This then becomes a regression problem as it will rely on mean squared error to improve fit. Using this state-value approximation, "we can use it to calculate the policy gradient and update our policy network to increase probabilities for actions with good advantage values and decrease the chance of actions with bad advantage values" [2, pp. 315]. This approach of calculating and storing weights is more memory-efficient than tabular learning with a Q-value lookup, and perhaps the increase in computation is even outweighed by the lack of having to fetch values from a lookup structure.
In implementation this means we have a network with two heads: the actor or policy net (*π(a|s)*), which holds a probability distribution of actions, and the critic or value net (*V(s)*), which intermittently assesses the value of the actions taken - which makes this an online learning policy as learning takes place during episodes.

→ **Actor:** updates policy parameters *θ*, in the direction suggested by the critic, *π(a|s; θ)*

→ **Critic:** updates the function parameters *w* using state value function *V(s, w)*.

We assume the following as our advantage function (taken from [10] as a clearer extract of the UC Berkeley CS285 slides [11]), which in short is derived from a substitution of Expectation using Bellman's equation.

$$A(s_t, a_t) = r_{t+1} + \gamma V_v(s_t + 1) - V_v(s_t)$$

In this manner, advantage can quantify the expectation against reality of an action while in a certain state. Our learner will be able to apply a higher weight on actions which paid off in a state [20, pp.248-249].

We chose to explore the synchronous variant of this AC method rather than the Asynchronous Advantage Actor-Critic (A3C) on the basis of an OpenAI Gym article [9] stating that A2C shows equal performance while being less computationally complex.

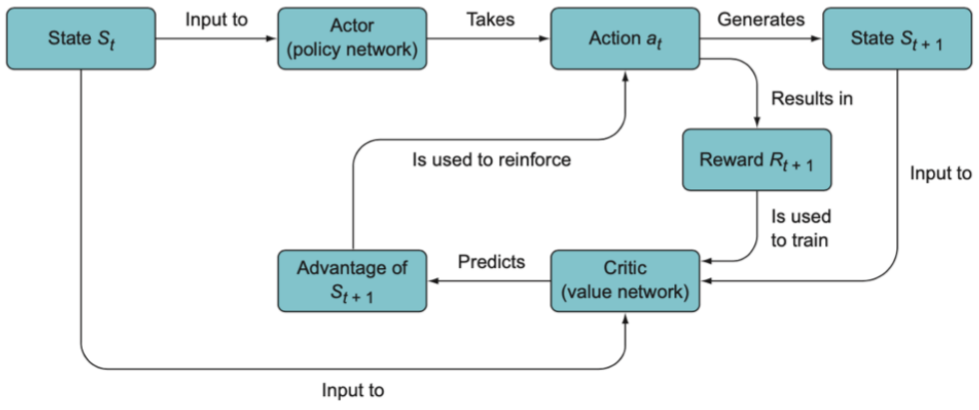


Fig.2: A visualisation of Actor-Critic from [12, pp.117]. First, the actor predicts the best action and selects an action, which generates a new state. The critic network computes the value of the old state and the new state. The relative value of S_{t+1} is called its advantage, and this is the signal used to reinforce the section selected by the actor.

A2C implementation details

The implementation is an online A2C variant (rather than batch) as we found this easier to comprehend. A2C involves learning *synchronously* across multiple instances of our chosen environment, but all running on a single worker, as opposed to A3C where these multiple environments are learned on asynchronously, i.e. not at the same pace, in parallel on separate workers [13, pp. 662-663]. A2C is significantly less complex as all weights are ready from all environments, before they are passed on to the optimiser of choice. This is data parallelism using a mirrored strategy, as all parameters remain in sync for all learning processes [13, pp. 704-707].
The code to construct the actor-critic networks (via our ActorCritic class) in PyTorch is quite straightforward and as shown by the numerous code examples we found, there is little need for this to vary. A2C requires some way to manage a number of concurrent environments through multiprocessing. Many examples adapted (instead of importing) this subprocessing code in parts from OpenAI Gym's baselines with minimal credit, or borrowed the same adaptation amongst themselves. We decided to use the stable_baselines3 library from the German

Aerospace Center (DLR), Institute of Robotics and Mechatronics (RM)[18]. It included an implementation of A2C which we experimented with but did not use.

We made some attempts to code our training logic independently from scratch, but unfortunately ran into a lot of confusion around flavours of A2C examples, and ended up relying almost entirely on a version in [19] (with some very minor code changes to be able to log metrics and inspect values), which we found best explained the process to us, and to be the most in agreement with Pytorch's actor-critic example and the *Deep Reinforcement Learning in Action* book from Manning Press[12]. We added our own logging and metrics around this code to set up our experiments

A description of CartPole-v1 environment

This task is best demonstrated with a continuous environment. We chose Open-AI gym's CartPole-v1 problem [8], with the goal of balancing a weighted pole on a moving cart. CartPole-v1 has an action space of size 2: a force of +1 or -1 that moves the cart left or right. The observation space is a vector of size 4 that indicates the cart position, cart velocity, pole angle, and pole velocity. We receive a reward of +1 for every step the pole has not fallen. CartPole-v1 has 4 terminating states:

- `env.spec.max_episode_steps`: 500
- `env.spec.reward_threshold`: 475.0 (although confusingly we did manage to hit reward 500)
- Pole is more than 15 degrees from vertical
- The cart moves more than 2.4 units from the centre

Case study

There are few complete examples of experiments for A2C on CartPole, so we took the advice of two sources for hyperparameters and hyperparameter ranges to explore. Lapan discusses A2C on OpenAI Gym's Pong [2, pp.323-329], which is significantly more complex as it employs a convolutional neural network to solve frames of the game at different stages. The analysis is not exactly comparable as Lapan observed A2C Pong over 8 million steps, but he offers detailed advice on which metrics to monitor for performance, and hyperparameters of note - we list our logged metrics in Fig.4. Morales offers helpful starting points for hyperparameter values for A2C on CartPole-v1 in [3, pp.371].

We kept to one neural network hidden layer with varying numbers of neurons, and used backpropagation to feed updated weights back. The optimiser of choice is Adam, a stochastic gradient descent optimiser that is highly configurable and is an improvement on AdaGrad and RMSProp. It should be noted that RMSprop was suggested by Mnih et. al [14, Supplementary Material] and the authors of the stable-baselines3 library [16], respectively advising that it would be more robust and would stabilise training. By trial and error we found that RMSProp clearly stabilised the reward variance as promised, but the max reward also remained very low and performance was unsatisfying. Our final hyperparameters resulted in 54 models.

In addition to our final hyperparameters in Fig.3, we also experimented with:

- *Reward step size/bootstrapping*: The number of steps taken by the actor between advantage calculations. We tried this at 5 and 10, but as a larger number meant fewer updates and possibly slower convergence, we found 5 to be sufficient.
- *Regularization coefficient for value*: Made little to no noticeable impact on performance
- Regularization coefficient for entropy: Made little to no noticeable impact on performance
- *Gamma or discount factor:* From previous experience in other learning methods, our gamma would ideally have to be 0.6 and over for the agent to perform well. We decided to keep this at 0.99.
- *Max gradient/gradient clipping threshold*: We took the advice in [3, pp.317] to use this to prevent our gradients from growing too large and overfitting. We experimented with different values but decided to keep this constant at 0.1.
- *Optimiser epsilon*: This is a hyperparameter for our Adam optimiser, used to prevent zero division errors. We experimented with this, but it did not offer much impact for the small numbers we trialled. We decided that the default of 1e-08 was sufficient.

Hyperparameters	**Hyperparameter values**
No. of environments	6,12,24
No. of episodes	50000,10000
Learning rate	0.001, 0.002, 0.003
Hidden layers (number of neurons)	32, 64, 128

Fig.3: Final hyperparameters used for the experiment

Plots over number of episodes i.e. timeseries	**Flat (single number) metrics**
Rewards Reward variance Mean actor loss Mean critic loss Mean entropy loss Mean overall loss	Mean reward Min reward Max reward Reward variance Mean actor loss Mean critic loss Mean entropy loss Mean overall loss Wall time in seconds

Fig.4: Collected experiment metrics

Results

We examined our 54 models through collected plots and metrics in Fig.4 in a notebook [21]. In plotting reward variance, mean overall loss, mean reward and max reward against our hyperparameters, we observed that on metrics, reward variance decreases as mean overall loss increases, and max and mean reward are positively correlated. On increasing the number of episodes, based on the lower reward_variance and mean overall loss, we favour a model trained on **100000 episodes**. In an increasing number of environments, based on a lower mean overall loss, and sizeable increases in max and mean reward, we will favour a model trained on **more environments**, but we will be cautious about reward variance. On increasing learning rate, based on the lower mean overall loss and higher max reward, we will favour a model with **higher learning rate**, but again be cautious of the increase in reward variance. On increasing the size of the hidden layer, based on lowest mean overall loss and highest max reward, we will favour a model with **128 neurons**. Given a surprising drop in reward variance compared with 64 neurons, 128 neurons may be an acceptable tradeoff. So our best model is described as having, in order of priority: 100000 episodes, 128 hidden layer neurons, the highest number of environments, then highest learning rate (because of the bigger improvements on number of environments), lowest mean overall loss, and possibly, max reward. The chosen A2C model is shown in Fig.5, and its training metrics are in Fig.6.

wall_time	num_env	num_episodes	learning_rate	hidden_layers	min_reward	max_reward	mean_reward	reward_variance	mean_overall_loss
423.192914	24	100000	0.003	(128, 128)	37.9	307.1	140.426	51.6447492	0.58074408
433.807385	24	100000	0.002	(128, 128)	28.8	348.3	161.838	59.9241817	0.63369911
439.600475	24	100000	0.001	(128, 128)	20.2	319.1	172.089	56.3654272	0.75259442
325.486098	12	100000	0.003	(128, 128)	28.1	278.8	146.047	51.5161517	0.81071148
323.056237	12	100000	0.002	(128, 128)	29.7	349.5	141.94	53.7222393	0.90156802

Fig.5: Best model (in green) compared to others in the vicinity of sorting.

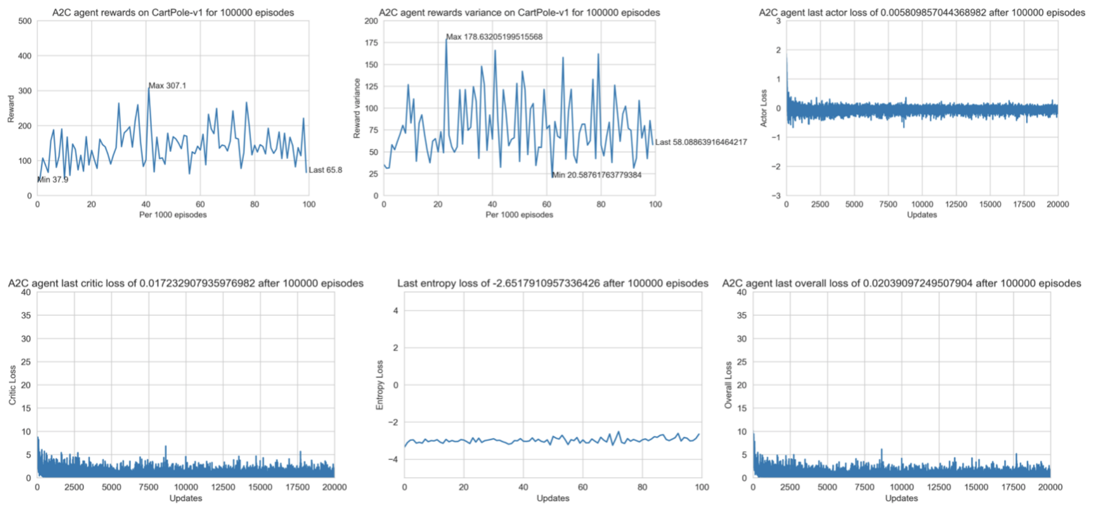


Fig.6: (L to R, top to bottom) Plots for rewards, rewards variance, and actor, critic, entropy & mean overall losses for A2C best model trained on CartPole-v1

As Lapan describes [2, pp.336], actor loss “oscillates” around 0 as the critic continuously corrects the actor, and we see the mean overall loss drop quickly at the beginning and remain about constant (Fig.6). There is an unexplained spike in all loss figures between 7500 and 10000 updates, which seems to correspond with the max spike in rewards at the 40000th episode mark (40000 episodes/5 reward steps = 8000 updates). Interestingly the same cadence of variance seems to have been maintained throughout the training, after the initial increase. The entropy loss seems to increase slightly but overall loss stays down. Lapan explains a similar pattern as showing “the agent becoming more confident in its actions as the policy becomes less uniform”[2, pp.325], although he does not explain where this leads to.

We then tested this model with 100 plays on CartPole-v1. The results are shown in Fig.8, and in row 1 of Fig.9 (both of these are in Task 4), where its performance is later compared against a best A2C model trained on CartPole-v0. The model hits a max reward 448.0 once, but largely underperforms in the range of 100 to 200.

Reflections and future improvements

- We feel that the experiments show the hyperparameters we chose had a significant impact on performance, but we should explore other hyperparameter ranges.
- We should have expected that the metrics were in essence time-series, so would be skewed as learning progresses. Instead of mean metrics, we should have collected full information of rewards and losses over time.
- We should have logged advantage loss; as this is the scale of our policy gradients, we should monitor this [2, pp.326].
- Reproducibility was an issue with our experiments, despite setting seeds for both PyTorch and gym (random and NumPy were not used). Given more time we would have liked to debug the weights from the start, or repeat experiments to collect a mean performance.
- We might look into controlling overfitting by applying some early stopping criteria such as a minimum loss or a minimum discount factor, or regularisation of weights using L1, or introducing noise with dropout.

Task 4: Test A2C best model trained on CartPole-v0, on CartPole-v1 (and further environment swapping)

In Task 3 we observed a pattern of reward variance markedly increasing as episodes were accumulated. We wondered if this was overfitting, and we missed a trick by not considering a simple early stopping criteria of terminating learning after, say, the model achieved over a certain reward threshold for a certain number of times.

CartPole-v0’s dynamics are exactly the same as in CartPole-v1, but with a slight difference in terminating states - the max episode steps and therefore rewards are lower:

- `env.spec.max_episode_steps`: 200

- `env.spec.reward_threshold`: 195.0 (confusingly again we did manage to hit reward 200)

Results and Comparison Based on the same model selection criteria we used in Task 3, our best CartPole-v0 model learned on 100000 episodes, 128 hidden layer neurons, 24 environments, and a learning rate of 0.002.

wall_time	num_env	num_episodes	learning_rate	hidden_layers	min_reward	max_reward	mean_reward	reward_variance	mean_overall_loss
424.64463	24	100000	0.003	(128, 128)	35.3	197.3	134.481	33.1007574	0.77586046

Fig.6: Best A2C model trained on CartPole-v0

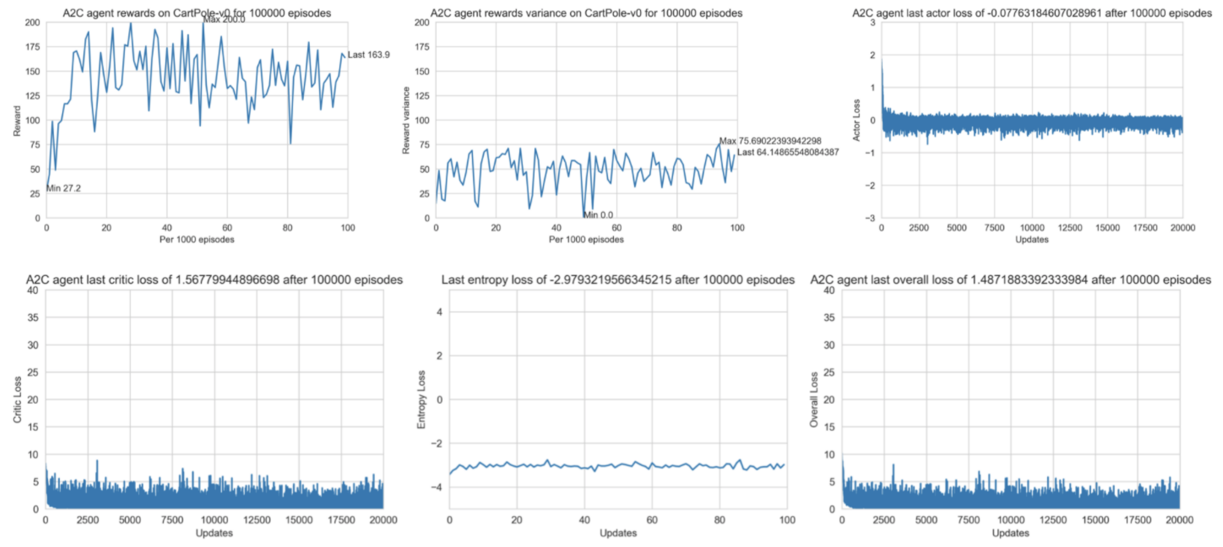


Fig.7: (L to R, top to bottom) Plots for rewards, rewards variance, and actor, critic, entropy & mean overall losses for A2C best model trained on CartPole-v0

We decided to not only test this model on CartPole-v0 (hereon A2C-v0), but also on CartPole-v1, and to run our best A2C model trained on CartPole-v1 (hereon A2C-v1), on CartPole-v0 for completeness. The results were surprising - A2C-v0 not only easily met its max reward goal of 200.0 several times (still with some variance), it also significantly surpassed A2C-v1 on CartPole-v1, hitting the max reward goal of 500.0 several times, with plenty of scores above 300. A2C-v1 on CartPole-v0 did hit the max reward goal of 200.0 several times, but clearly underperformed against A2C-v0 on the same environment. The reward plots can be seen in Fig.8 and 9.

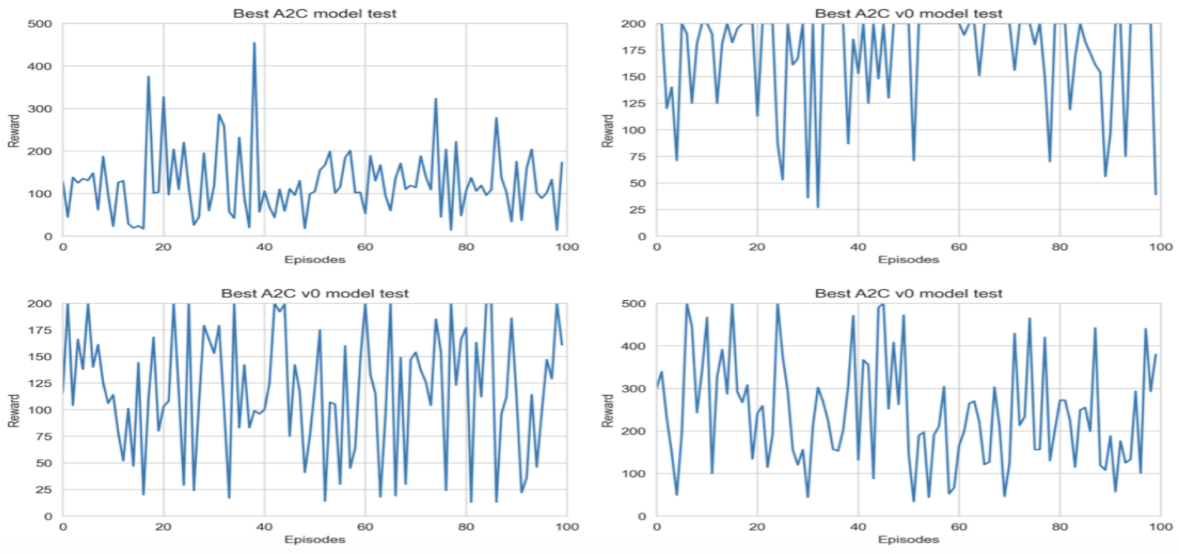


Fig.8: Reward plots in test for: (LtoR, top to bottom) A2C-v1 on CartPole-v1; A2C-v0 on CartPole-v0; A2C-v1 on CartPole-v0; A2C-v0 on CartPole-v1.

Ref	A2C model/trained on	Tested on environment	Min reward	Max reward	Reward variance	Mean reward
1	CartPole-v1	CartPole-v1	11.0	448.0	75.21	119.95
2	CartPole-v0	CartPole-v0	27.0	200.0	46.33	171.62
3	CartPole-v1	CartPole-v0	13.0	200.0	56.20	118.03
4	CartPole-v0	CartPole-v1	28.0	500.0	138.91	271.12

Fig.9: Reward plots in test for: (LtoR, top to bottom) A2C-v1 on CartPole-v1; A2C-v0 on CartPole-v0; A2C-v1 on CartPole-v0; A2C-v0 on CartPole-v1.

In conclusion, it seems possible that parameterising the size of the reward on a continuous environment during training might be worth exploring further.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, MA, USA: Bradford Books, 2018.

[2] M. Lapan, *Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more, 2nd Edition*, 2nd ed. Birmingham: Packt Publishing, 2020.

[3] M. Morales, *Grokking deep reinforcement learning*. New York, NY, USA: Manning Publications, 2020.

[4] Wikipedia contributors, ‘Prim’s algorithm’, *Wikipedia, The Free Encyclopedia*, 01-Apr-2021. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Prim%27s_algorithm&oldid=1015436453. [Accessed: 07-Apr-2021].

[5] O. Zekai, *https://github.com/OrWestSide/python/blob/master/maze.py*. .

[6] F. Phoon and E. Ernest-Pastor, *https://github.com/feiphoon/inm707-joint-coursework/blob/main/src/maze/maze.py*. .

[7] F. Phoon and E. Ernest-Pastor, *https://github.com/feiphoon/inm707-joint-coursework/blob/main/src/maze/q_maze.py*. .

[8] O. Gym, *https://gym.openai.com/envs/CartPole-v1/*. .

[9] Y. Wu, E. Mansimov, S. Liao, A. Radford, and J. Schulman, ‘OpenAI Baselines: ACKTR & A2C’, *Openai.com*, 18-Aug-2017. [Online]. Available: https://openai.com/blog/baselines-acktr-a2c/. [Accessed: 08-Apr-2021].

[10] ‘UC Berkeley CS 285 Deep reinforcement learning: Actor-Critic algorithms’, *Berkeley.edu*. [Online]. Available: http://rail.eecs.berkeley.edu/deeprlcourse/static/slides/lec-6.pdf. [Accessed: 08-Apr-2021].

[11] C. Yoon, ‘Understanding actor critic methods and A2C - towards data science’, *Towards Data Science*, 06-Feb-2019. [Online]. Available: https://towardsdatascience.com/understanding-actor-critic-methods-931b97b6df3f. [Accessed: 08-Apr-2021].

[12] A. Zai and B. Brown, *Deep reinforcement learning in action*. New York, NY, USA: Manning Publications, 2020.

[13] A. Geron, *Hands-on machine learning with scikit-learn, keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems*, 2nd ed. Sebastopol, CA, USA: O’Reilly Media, 2019.

[14] V. Mnih *et al.*, ‘Asynchronous methods for deep reinforcement learning’, *arXiv [cs.LG]*, 2016.

[15] O. Gym, *https://gym.openai.com/envs/CartPole-v0/*. .

[16] ‘A2C — Stable Baselines3 1.1.0a1 documentation’, *Readthedocs.io*. [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html. [Accessed: 09-Apr-2021].

[17] O. Gym, *https://github.com/openai/baselines/tree/master/baselines/common*. .

[18] DLR-RM, *https://github.com/DLR-RM/stable-baselines3*. .

[19] *https://github.com/higgsfield/RL-Adventure-2/blob/master/1.actor-critic.ipynb*. .

[20] M. Pumperla and K. Ferguson, *Deep learning and the game of go*. New York, NY, USA: Manning Publications, 2019.

[21] F. Phoon and E. Ernest-Pastor, *Actor_critic_train_analysis_cartpolev1.ipynb*. [Online]. Available: http://actor_critic_train_analysis_cartpolev1.ipynb. [Accessed: 10-Apr-2021].

[22] Van Hasselt Arthur Guez and David Silver, H. (Ed.). (n.d.). *Deep Reinforcement Learning with Double Q-Learning*. Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence (AAAI-16).

.