Deep Reinforcement Learning Project

Thomas Fishwick

Abstract

Here we experiment with the basic Q Learning algorithm and then look into Deep Q Networks and Evolutionary Strategies

Our code is available at this link `https://github.com/SL477/DRL-coursework`.

1 Main problem: Starship Saboteur

1.1 Define an environment and the problem to be solved

Our agent will start in a random empty location in the map shown in Figure 1. It has two positive reward points; the darker green point is the one with the highest reward and the lighter green point will terminate the training episode. The yellow points are negative reward points, they also deduct health points and if the agent's health drops to zero or less this will also terminate the episode. The agent can transition to any adjacent cell if it is not a wall. Currently the doors are only decorative items.

The story version: Fired from a Solar Federation Space Force Infiltrator, our intrepid Space Marine Commando robot exits its boarding pod in a random place in the Custodian warship. Its mission is to cripple the ship's reactor before its crew are revived from stasis by planting a timed explosive on it. Its survival is optional, after it has completed its mission, the robot can escape back to where a shuttle has cut a hole into the ship to retrieve it. But if the robot is destroyed, well we can always build more robots.

We will be using Q-learning to 'solve' this environment. Q-learning attempts to work out the cost/benefit of going from one state to the various other states around it based on their current and future rewards. Figure 2 shows which nodes are accessible to each other.

1.2 Define a state transition function and the reward function

To make our list of states we flatten the 2D map into an array. To flip from the array to the map we can get the row by integer dividing the index by the height of the map and the column by taking the modulus of the index by the width.

For the reward function we create a matrix with one state per row and one action per column. Using a lookup back to the map we get the code of the cell and lookup its reward (table 1). As the agent is only allowed to claim the primary objective once per episode the reward is set to zero once it has been claimed by the agent.

Table 1: Rewards

Code	**Reward**
Empty space	0
Wall	0
Door	0
The primary goal (the reactor control panel)	50
The secondary goal (the escape route)	30
A trap	-10

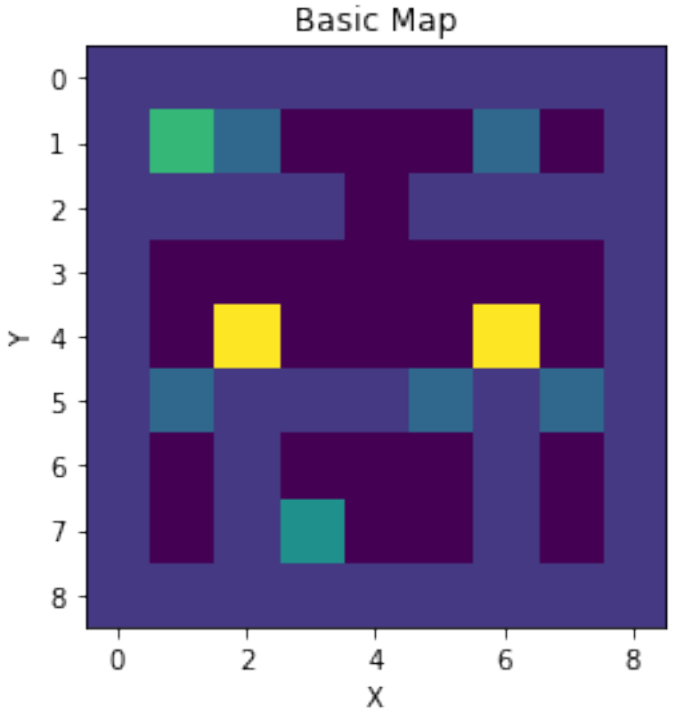


Figure 1: Green: reward points, yellow: obstacles, blue: doors, dark purple: empty space, light purple: walls

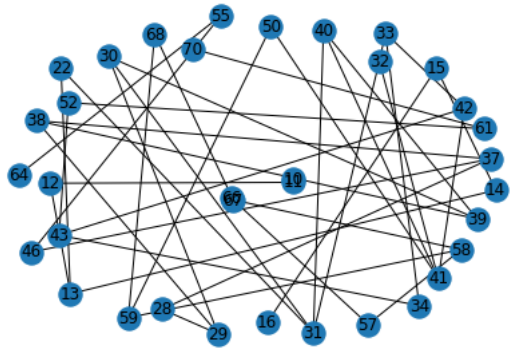


Figure 2: Nodes with access to one another

1.3 Set up the Q-learning parameters (gamma, alpha) and policy

For the Q-learning algorithm we have implemented it as a function (run_q_learning_basic), with the parameters alpha, gamma, epsilon and num episodes.

Alpha, the learning rate, we have set as 1.

Gamma, the discount factor for future rewards, we set as 0.8.

Epsilon, the chance of choosing the policy 'exploit' over 'explore', we set as 0.9.

Num_episodes, the number of learning episodes, we set as 1000.

1.4 Run the Q-learning algorithm and represent its performance

The Q-learning algorithm is:

$Q[s,a] = Q[s,a] + \alpha * R[s,a] + \gamma * (max(Q[s]) - Q[s,a])$

With the exploit/explore part given by:

if RandomNumber $> \epsilon$ then explore, otherwise exploit.

The algorithm takes 22.5 seconds to run on my PC. In Figure 3 we can see the mean rewards versus their corresponding Q variances. The mean total health stayed at roughly 100, with at least one iteration within each iteration doing something to reduce its health. Most of the iteration's mean rewards were around 50, indicating that they were only going for one of the reward points (fortunately the higher reward point).

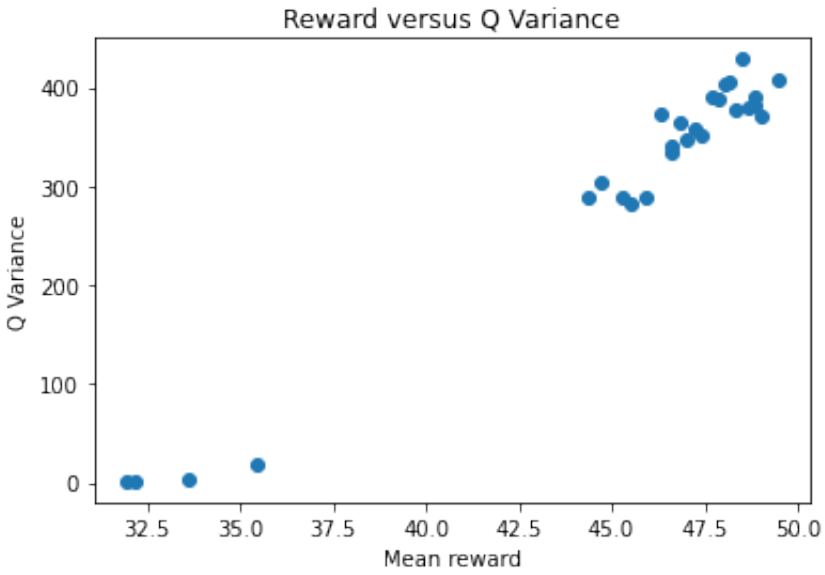


Figure 3: Reward Versus Variance, for Alpha = 1, Gamma = 0.8, Epsilon = 0.9, for 27 rounds of 1000 iterations of training

It also looks like those which converged fastest were also the ones to get the lowest reward (around 30), they likely found that one first and did not explore enough to find the other reward point.

1.5 Repeat the experiment with different parameter values, and policies

Running a full grid search of Alpha, Gamma and Epsilon parameters results in Figure 4, where we can see that as the epsilon (explore/exploit) parameter gets lower (promoting explore) our agent is more likely to wander into one or more of the traps. Considering that these traps give minus ten points and this is the average reward over the number of episodes until convergence (or 1000 episodes) this is quite the effect for some of the more extreme negative rewards (the agent can only step on the traps ten times a run before it is terminated).

Holding the other parameters stationary and running through the different Gamma (discount factor for future reward) sees a steady decline in how much reward it picks up, until at Gamma equals 1 (or in other words future rewards are now worthless) it repeatedly throws itself at the traps (Figure 5). We had it run through a thousand-episode training session three times for each parameter, so it is not that likely to be a statistical anomaly.

For Alpha (the learning rate) running a grid search with Gamma at 0.8 and Epsilon at 0.4, does not seem to have too much effect on the mean reward but it does have a more or less linear relationship with the Q variance (the Gamma variable has a similar effect up until gamma = 1) as shown in Figure 6.

1.6 Analyse the results quantitatively and qualitatively

Here we compare the Q variance and mean rewards for the old parameters (Alpha: 1, Gamma: 0.8, Epsilon: 0.9) and the new parameters (Alpha: 0.8, Gamma: 0.8, Alpha: 0.4), in Figure 7. The new parameters have much lower variance compared with the new parameters in that they are very tightly distributed, and the old parameters are much more spread out. The problem with the new parameters is that they seem to be very good at learning the wrong lessons, as they have learnt to go for the lower reward by going through the traps. The original parameters have learnt to generally avoid the traps, and some have gone for the higher reward point, and some have gone for the lower reward point. The new values have high precision but are somewhat biased away from what we want

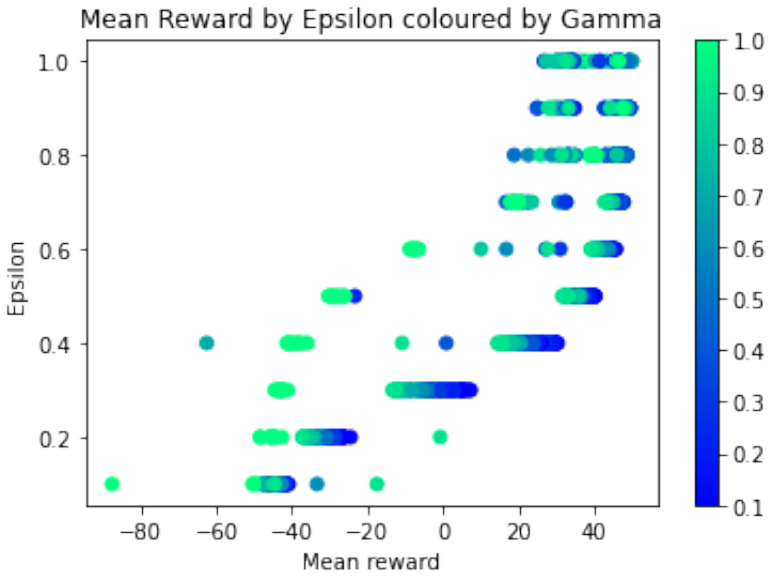


Figure 4: Grid Search of the parameters

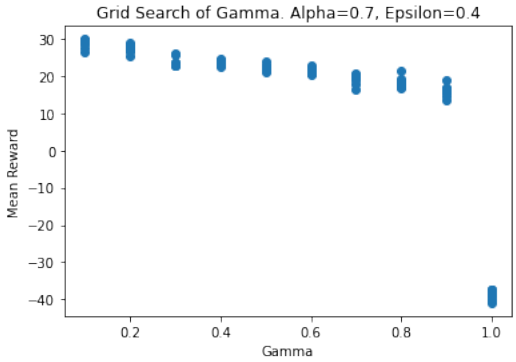


Figure 5: Grid Search of the gamma parameter

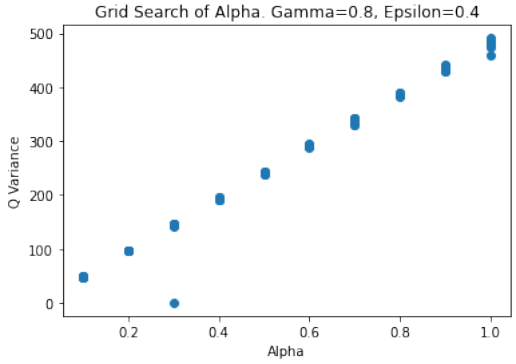


Figure 6: Grid Search of the alpha parameter

them to do (get the higher reward point and ideally the lower one too). The old values have low precision but are generally unbiased towards their primary target (though some get just the secondary target, but not both).

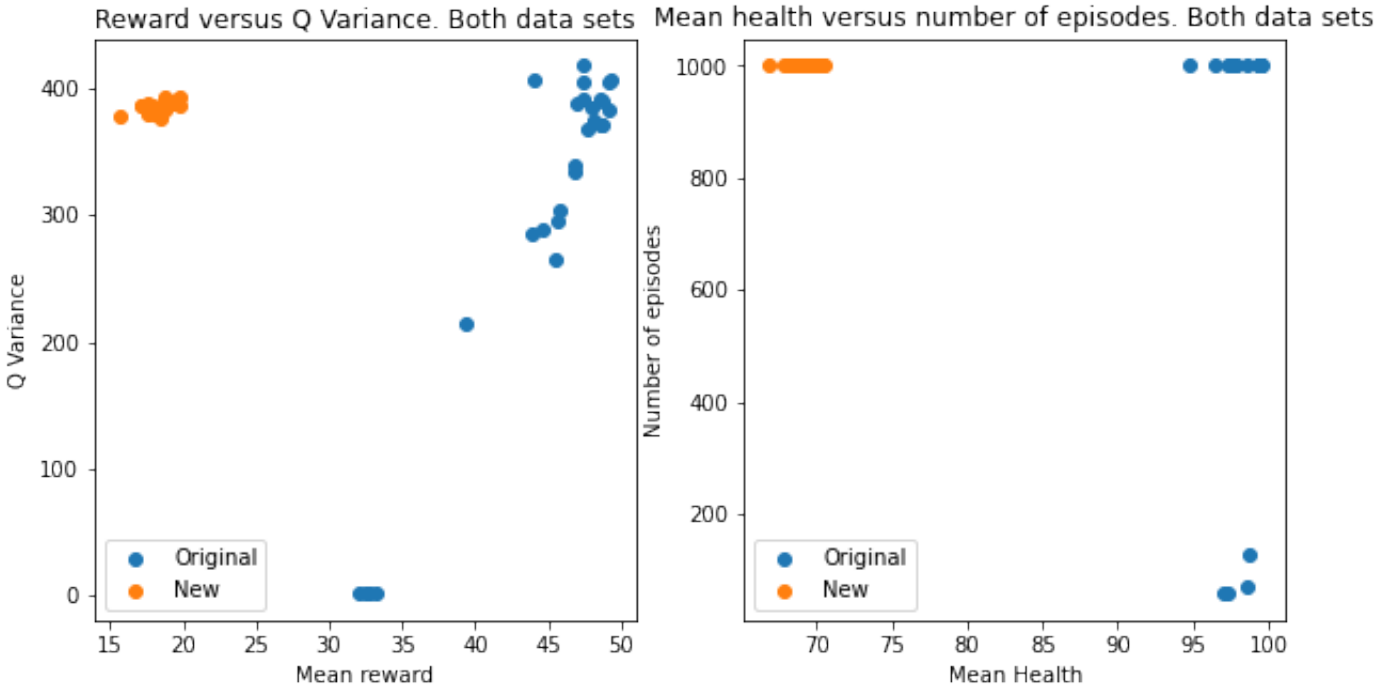


Figure 7: New Versus Old parameters

2 Implement DQN with two improvements

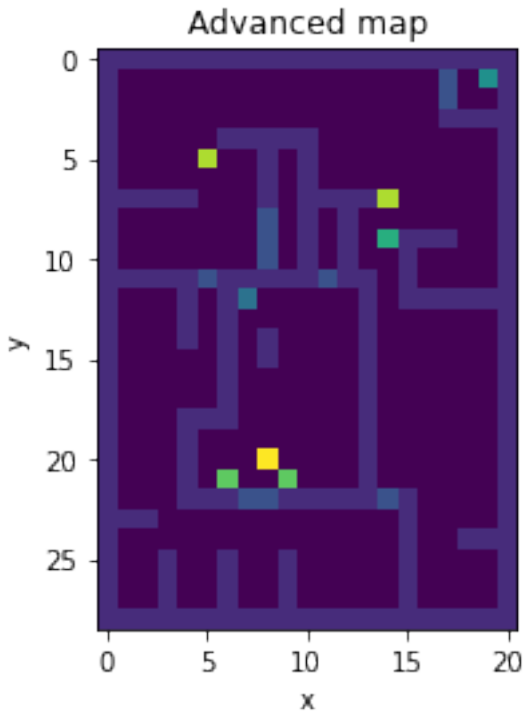


Figure 8: Green: reward points, yellow: obstacles, blue: doors, dark purple: empty space, light purple: walls

Here we have used a similar environment to the basic task, but now have enemies which move around and we now only have a limited field of view (Fig. 8). To counter the limited field of view we have an objective compass

to see where we are going. Our agent also has a gun to remove nearby enemies. Running the unimproved model through 44 variations of parameters leaves us with the best hidden units of 256 and 256, and the best gamma of 0.9 (using RLLib for the grid search).

2.1 DQN with two improvements

We will improve on the basic Q learning algorithm using two methods:

Our first one is Double Q-Learning. "The standard DQN uses the same values both to select and to evaluate an action. This makes it likely to select overestimated values, resulting in overoptimistic value estimates" (van Hasselt et al. 2016, p. 2). As our environment is rather complicated, we would not want the algorithm to over-estimate what it is doing, but to make sensible decisions about where to go and what to do. In the standard algorithm we have seen it improve, get worse and then improve again, so we would prefer to see continuous improvements in the algorithm's behaviour. Using (van Hasselt et al. 2016, p. 2, 4) to get the relevant equations and parts of the code to modify and (Hennis 2019) to check that we were on the right track, we updated the code leaving an on/off switch so that we could run the code with or without this improvement.

Our second improvement will be Prioritised Experience Replay, "an issue with traditional RL techniques is the potential rapid forgetting of possibly rare experiences that would be useful later on" (Schaul et al. 2015, p 1). Our environment is fairly large and only has two reward points, so it would be beneficial for the agent to remember to collect them (and in the correct order, as one point ends the episode so that it cannot collect the other one). The environment also has traps and enemies which it would be best to learn to avoid. A large portion of an episode would be moving from the start point to the reward points with not much else happening, it seems silly that the agent would remember those experiences with the same importance. Using (Schaul et al. 2015, algorithm 1) we were able to make the corresponding changes to the memory store and optimisation part.

2.2 Analyse the results quantitatively and qualitatively

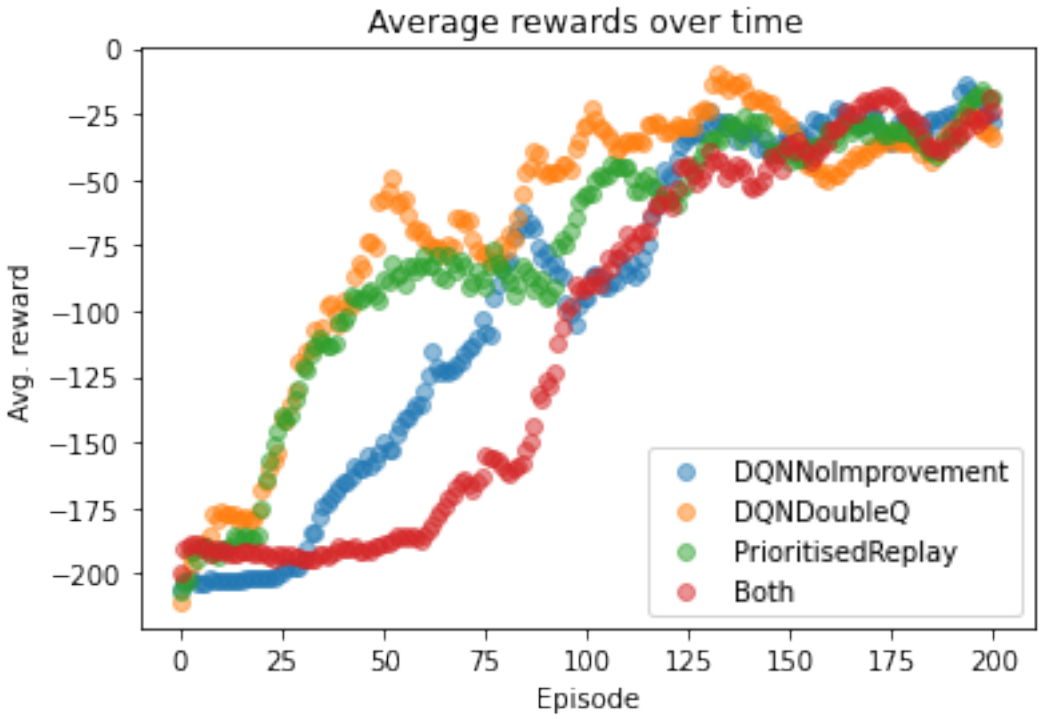


Figure 9: DQN Mean Rewards over time from RLLib

All the rewards over time averages are below 0 (apart from plain DQN which gets near 150 at first), so we may not have been generous enough with the reward amounts (or the turn limit was too high and stopped the agent associating actions with rewards). We took off one point per step, ten points off for hitting a trap or being

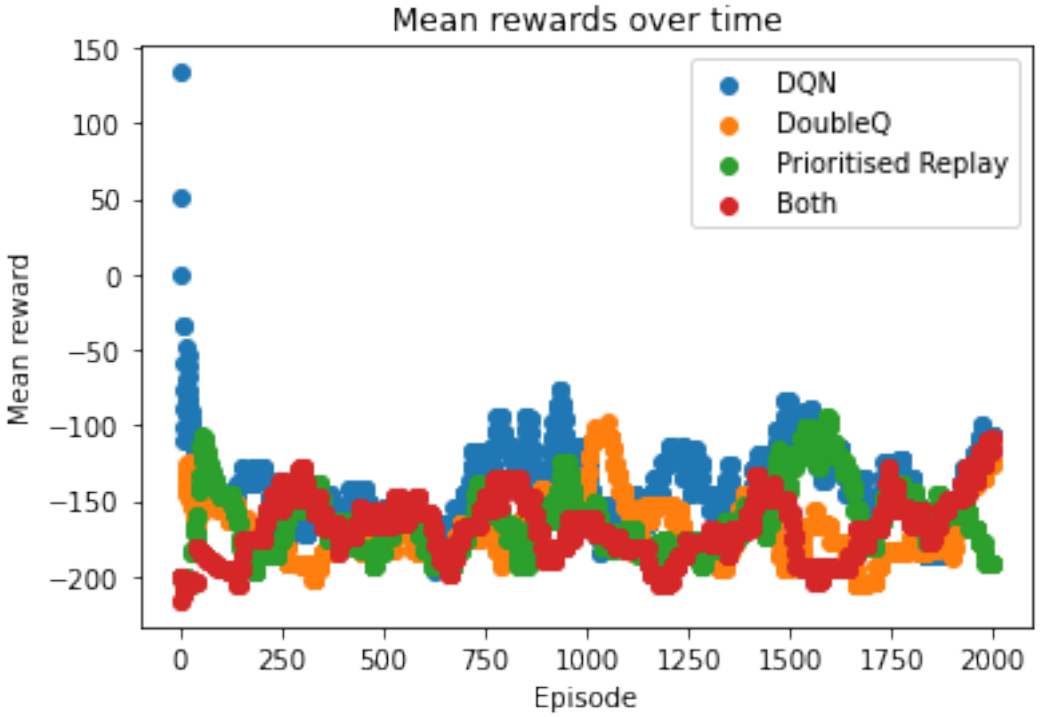


Figure 10: DQN Mean Rewards over time from scratch

hit by an enemy, we gave a thousand points for the primary objective and five hundred points for the secondary objective. We attempted limiting it so that the secondary reward was only given after the primary reward was given, but this just seemed to prevent the algorithm from learning. Our implementation may also need more training, as this is after 2000 episodes, in earlier versions we were going for 10000 episodes.

The basic DQN algorithm seems to have had a better start than the other algorithms in our implementation, but this may have just been pure chance. It is much more tightly clustered than the other algorithms, with it gently oscillating around -100 to -150 as the mean reward. In the video it slides into a wall and stays there. In the RLLib version (using an older version of the environment Fig. 10) the plain DQN algorithm is also a bit unstable, but converges to the same sort of area as the other algorithms.

Double DQN in its first run managed to collect both objectives (highlighting an issue with the compass (now fixed)) in one episode (though this may have been a fluke). It appears rather unstable in our implementation, with it having periods of small and large oscillations, but it does not appear to have learned much. In its video it traps itself in the corner, so it may be that it is interpreting the compass, but not the screen. In the RLLib version it learns the fastest but converges to the same area as the other algorithms.

For prioritised replay we can see that it is much more unstable in its learning, it is possibly tending upwards, but that could just be noise. In its video it traps itself against the wall right next to the door (the doors may not have been the best idea, or at least they should have used a different number and revealed what was on the other side). In RLLib's implementation we can see that it gives an early training advantage like Double Q but not as big an effect, but then converges to the same number (it also is not as unstable as Double Q).

For DQN with both improvements it seems to give it a disadvantage out in learning at first in RLLib's version, but then it converges to the same values as the other algorithms. In our version having both improvements seem to have made it rather unstable in a smaller area than Double Q and Prioritised Replay, so it may have inherited the disadvantages of both algorithms. Its video is the most impressive of the four, as it kills an enemy (and then stops next to the traps, presumably trying to kill them too (which may have been a good idea to implement)). So, it at least has figured out the idea that near things which take points off you should try to shoot them.

As the Deep Q network seems to have had quite a tough experience with this environment, the normal Q learning algorithm would likely have had it a lot worse. An issue with the environment may have been the primary and secondary objectives confusing the algorithm. It may have worked out what to do if there was simply the

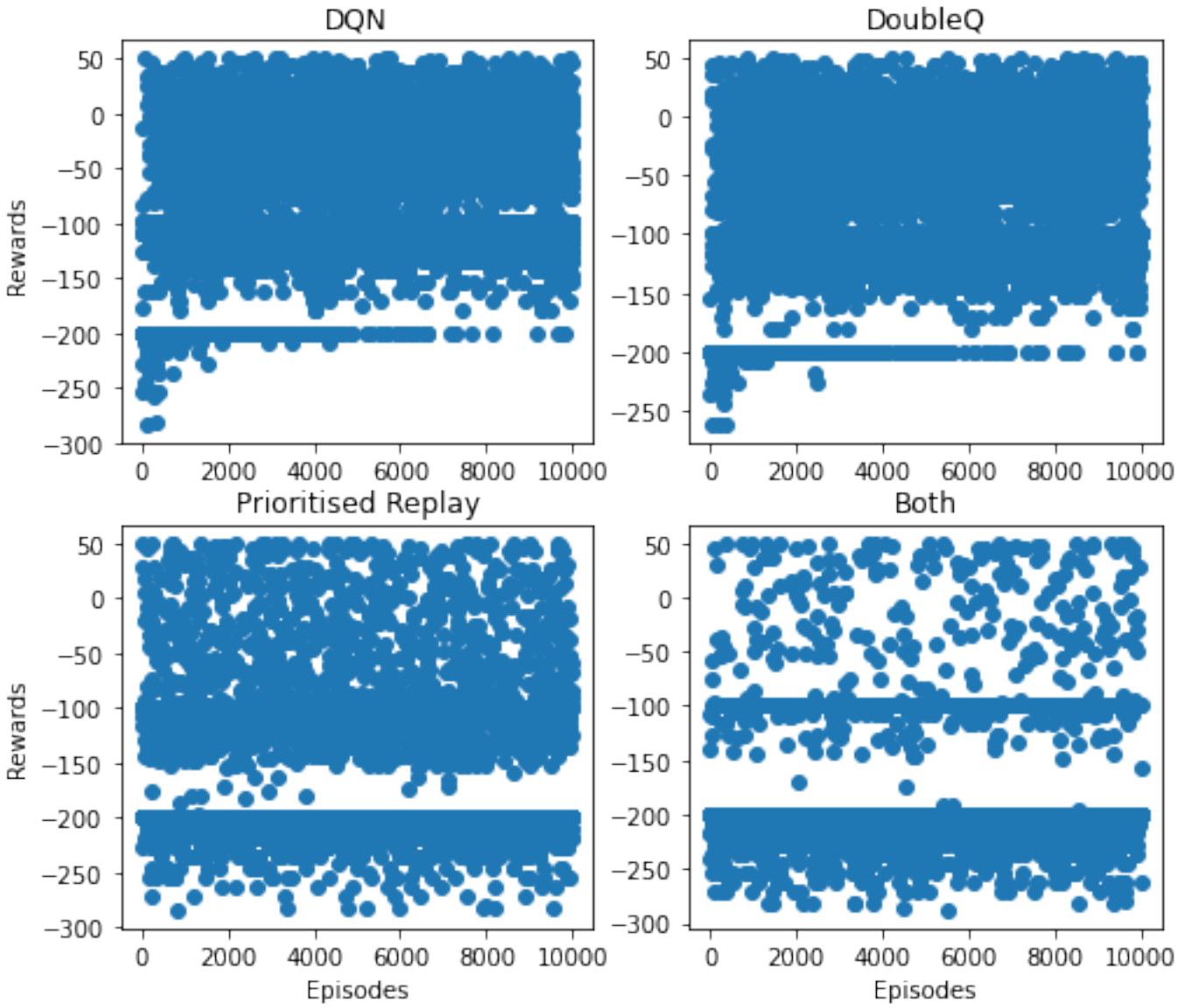


Figure 11: DQN Rewards over time from scratch

primary objective to go and get (possibly the objective being in a random place too).

3 Apply the RL algorithm of your choice (from rllib) to one of the Atari Learning Environment. Briefly present the algorithm and justify your choice

In this section we chose to use the Evolutions Strategies algorithm and applied it onto the Space Invaders environment. Evolution Strategies is rather different to the Deep Q Network, in Deep Q Networks we have a neural network and after each episode of playing the game we will use gradient descent to adjust the weights of the neural network to try to improve the same network in the next episode of playing the game (roughly inspired by the idea of how a brain learns). In Evolution Strategies, "a population of parameter vectors is perturbed and their objective function is evaluated. The highest scoring parameter vectors are then recombined to form the population for the next generation" (Salimans et al. 2017). Or in other words it builds a series of random neural networks based on the prior generation (or a random seed for the first run) and the best one is then used to spawn another generation until the overall optimisation algorithm decides that it cannot get any better (taking its inspiration from evolution

by natural selection). The upshot of the algorithm is that it is embarrassingly parallel, in (Salimans et al. 2017)[p. 7-8] the authors talk about using 80 machines to solve the 3D humanoid environment in 10 minutes. As DQN relies on using the previous episode's gradient to update the algorithm it is not able to parallelise in the same way as ES.

3.1 Analyse the results quantitatively and qualitatively

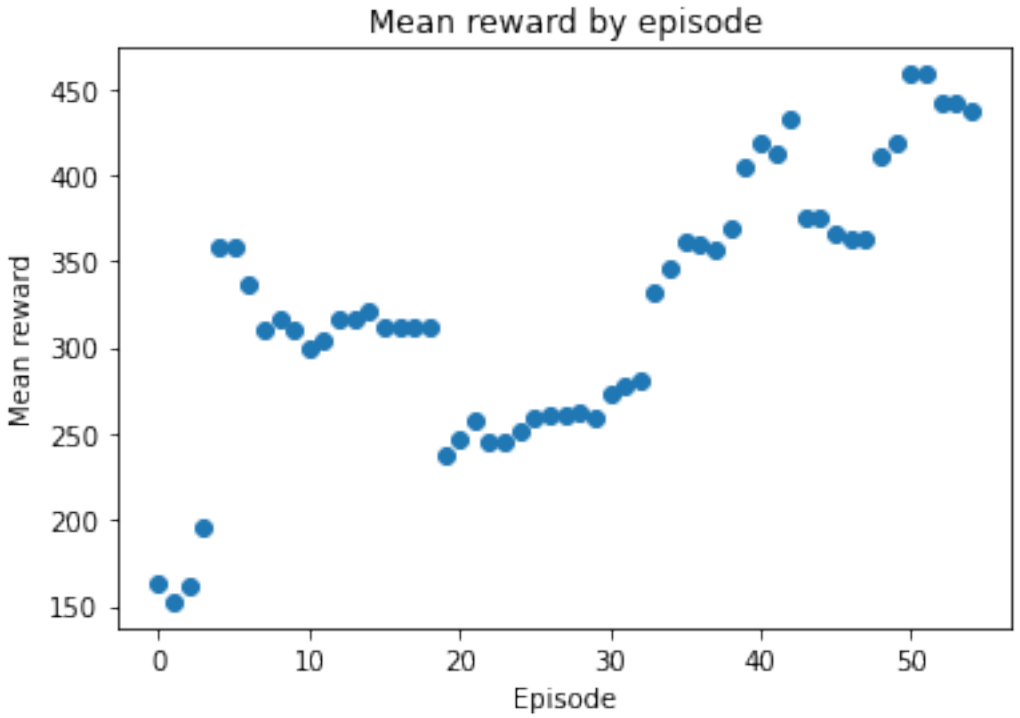


Figure 12: Evolutionary Strategy's Rewards

From Fig. 12 we can see that the mean rewards jump up and down quite a bit, but generally linearly increase over time. This was after approximately 30 hours of training on 6 CPUs (no GPU available). In the video (videos/SpaceInvaders2.mp4), we can see that the model has figured out an exploit where it sits still and continuously fires upwards, where it is able to get quite a big score before being killed. A boring strategy, but one which can get it a reasonable score. It has very obviously learnt that destroying the enemies gets it rewards but has not figured out that staying alive longer would get it a bigger score. This may be why the agent stopped improving over the last few episodes as it had got into a state where no more learning was really possible.

The model managed to score more highly than the one which simply performed random actions (videos/SpaceInvadersRandom.mp4), though by chance that one also stayed relatively still and got a fairly high score. As evolutionary strategies is based upon random neural networks, it is somewhat random what we would get (an earlier model moved around and shot upwards, but got a lower score than the random actions video).

For a computer game such as this there is no regulatory need to provide the reasoning behind an action. If our model was buying or selling stocks on behalf of other people, driving a car/ship/plane, then regulators/auditors/accident investigations would want or legally require an explanation somewhat better than "the model said so". Unfortunately, there currently is not a way to get an understandable explanation behind the agent's actions.

In (Salimans et al. 2017)[p. 12] the author achieved an average reward of 678.5 in one hour, compared to our best of 458.5 in 30 hours. However, as the authors use 720 CPUs compared to our 6, then assuming that our CPUs are comparable we would need to run for 3600 hours to hit the same sort of ballpark of processing time (about 5 months). The best human (Jon Tannahill) scored 218,870 points at Space Invaders (Day 2018), with the best AI scoring 154,380 (GDI-H3) (Code n.d.), so ES has a long way to go before it can get there, but its only score is from an hour's training (with the 720 CPUs). Unfortunately, you would need a lot of money to power a

cluster like that to see if ES could better its score.

4 Implementation of PPO or SAC

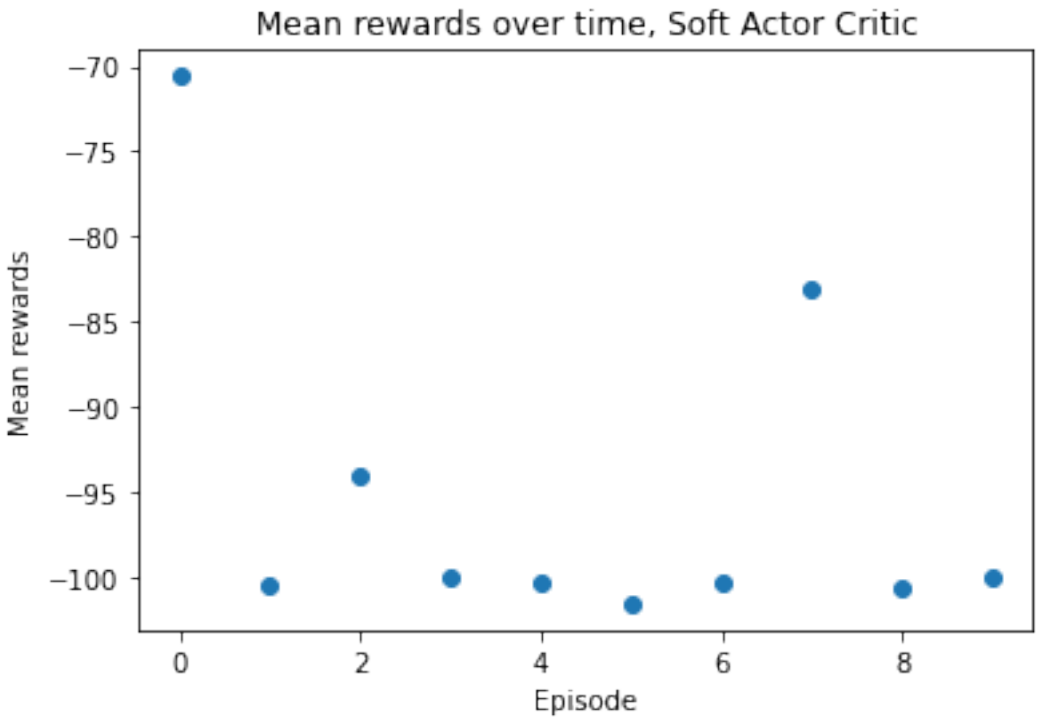


Figure 13: SAC's Rewards

Here we look at implementing Soft Actor Critic on our custom environment. Soft Actor Critic is similar to DQN, but rather than using a random number to determine whether to explore or exploit it uses another neural network to give a number and then that can be used. In Fig. 13 we can see that its mean rewards jumped around a bit as the agent was figuring things out. The theory behind it is good, as in DQN we have a random number generator determining whether or not to follow the model, whereas in SAC another neural network learns whether it is a good idea to follow the network or not. "SAC avoids the complexity and potential instability associated with approximate inference in prior off-policy algorithms based on Q-learning" (Haarnoja et al. 2018, p. 2). Once you account for some strangeness in how the algorithm is written it was relatively straightforward to implement.

5 Summary of contribution

100% me, with some code inspired by external sources (noted on the line, function or cell as relevant).

References

Code, P. W. (n.d.), 'Papers with Code - Atari 2600 Space Invaders Benchmark (Atari Games)'.
URL: *https://paperswithcode.com/sota/atari-games-on-atari-2600-space-invaders*

Day, P. (2018), 'Beating Richie Knucklez: the making of a Space Invaders world champion', *The Guardian* .
URL: *https://www.theguardian.com/games/2018/nov/12/beating-richie-knucklez-the-making-of-a-space-invaders-world-champion*

Haarnoja, T., Zhou, A., Abbeel, P. & Levine, S. (2018), 'Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor'.
URL: *https://arxiv.org/abs/1801.01290*

Hennis, E. (2019), 'Double Q Learning notebook'.
URL: *https://colab.research.google.com/github/ehennis/ReinforcementLearning/blob/master/03-DoubleQLearning.ipynb*

Salimans, T., Ho, J., Chen, X., Sidor, S. & Sutskever, I. (2017), 'Evolution Strategies as a Scalable Alternative to Reinforcement Learning'.
URL: *https://arxiv.org/abs/1703.03864*

Schaul, T., Quan, J., Antonoglou, I. & Silver, D. (2015), 'Prioritized Experience Replay'.
URL: *https://arxiv.org/abs/1511.05952*

van Hasselt, H., Guez, A. & Silver, D. (2016), 'Deep Reinforcement Learning with Double Q-Learning', *Proceedings of the AAAI Conference on Artificial Intelligence* **30**(1).
URL: *https://ojs.aaai.org/index.php/AAAI/article/view/10295*

6 Appendix

Listing 1: helpers/advanced map.py

```
"""here we define the Advanced Map class"""

from .map import adv_map, key
import numpy as np
from .enemy import Enemy
from .get_available_actions import Action, adv_actions

import matplotlib.pyplot as plt

class AdvancedMap():
    """Advanced map class. Holds an instance of the map. Animates the enemies. Handles actions

    Methods:
    reset, reset the map to what it was before & return the observations

    step, send in an Action named tuple to perform the step. Animate the enemies and the agent
        ↪ and send back the reward and other useful info

    display, prints a pretty view of the map

    displayImg, calls a matplotlib image of the map

    agents_view, returns the immediate surroundings of the agent, ray-traced(ish) out from the
        ↪ agent

    direction_to_objective, returns the direction to where the next objective is
    """
    def __init__(self, pos=None) -> None:
        self.map = adv_map()
```

```
        # find the enemies and their positions in the 3 time steps (looking for 7 & 8)
        self.enemies = []

        # set other stats
        self.agent_health = 100

        # reset the environment to set it up
        self.reset(pos=pos)

        # set a limit
        self.limit = 200

    def reset(self, pos=None) -> dict:
        """the function to reset the map"""
        # as we have changed a load of the values set the map back to what it was
        self.map = adv_map()

        # find the enemies and their positions in the 3 time steps (looking for 7 & 8)
        self.enemies = []
        banned_positions = []
        # enemies who bob up and down
        for i in np.argwhere(self.map == 7.):
            # to be consistent flip the positions around so that all tuples are (x, y)
            epos = [(i[1], i[0] - 1), (i[1], i[0]), (i[1], i[0] + 1)]
            banned_positions += epos
            self.enemies.append(Enemy(epos))

        # enemies who bob side to side
        for i in np.argwhere(self.map == 8.):
            epos = [(i[1] - 1, i[0]), (i[1], i[0]), (i[1] + 1, i[0])]
            banned_positions += epos
            self.enemies.append(Enemy(epos))

        # agent's starting point
        # The agent cannot start where the enemies will be stepping
        allowed_positions = np.argwhere(self.map == 0.)
        # Remove the enemy positions
        allowed_positions = filter(lambda x: (x[1], x[0]) not in banned_positions,
            ↪ allowed_positions)

        allowed_positions = list(allowed_positions)
        #print(allowed_positions)
        if pos is None:
            # use randint, as choice only works on 1-d arrays. Use a tuple as it is smaller
            self.agent_pos = tuple(allowed_positions[np.random.randint(0, len(allowed_positions
                ↪ ))])
            # flip round to x, y
            self.agent_pos = (self.agent_pos[1], self.agent_pos[0])
        else:
            self.agent_pos = pos

        # So that we can keep track of whether the agent stood on the floor or a door
        self.old_agent_value = self.map[self.agent_pos[1]][self.agent_pos[0]]
```

```
        self.map[self.agent_pos[1]][self.agent_pos[0]] = 5.

        # set other stats
        self.agent_health = 100

        # set a limit
        self.limit = 200

        # return the observations
        return {
            'is_stop': False,
            'immediate_reward': 0,
            'enemy_count': len(self.enemies), # enemy count
            'agent_view': self.agents_view(), # agent view
            'obj_direction': self.direction_to_objective(), # direction to objective
            'agent_health': self.agent_health, # agent health
            }

    def step(self, action: Action) -> dict:
        """Need to animate the enemies. Perform the action. Return the compass to the objective
            ↪ , update reward, mini-map, etc"""
        # run actions first. If the shoot action has been performed will need to eliminate
            ↪ enemies first
        ret = {'is_stop': False, 'immediate_reward': 0}
        if action.shoot:
            # Shoot
            # Go through the locations of the enemies, if one is within 3 spaces of the agent
                ↪ then remove it from the array
            remove_enemies = []
            for idx, enemy in enumerate(self.enemies):
                # Get the x & y of the enemy
                if enemy.check_if_near_point(self.agent_pos[0], self.agent_pos[1]):
                    remove_enemies.append(idx)

            # pop the enemies from the list and update the map value to 0
            for enemy_idx in reversed(remove_enemies):
                remove_enemy = self.enemies.pop(enemy_idx)
                e_x, e_y = remove_enemy.positions[remove_enemy.current_position]
                self.map[e_y][e_x] = 0.0
        else:
            # check if the cell moving to is empty
            new_x, new_y = self.agent_pos
            new_x, new_y = new_x + action.x, new_y + action.y
            val_new_cell = self.map[new_y][new_x]

            val_key = key(advanced=True)
            if val_key[val_new_cell]['solid']:
                # If the new cell is solid then we cannot move there
                new_x, new_y = self.agent_pos
            else:
                # Move to the new position
                # update the map with the old value
```

```
            self.map[self.agent_pos[1]][self.agent_pos[0]] = self.old_agent_value
            # store the old value of the new cell
            self.old_agent_value = self.map[new_y][new_x]
            # show on the map we have moved
            self.map[new_y][new_x] = 5.
            # update the agent's position
            self.agent_pos = (new_x, new_y)

            # Sort out rewards
            rew = val_key[val_new_cell]['reward']
            if not np.isnan(rew):
                ret['immediate_reward'] += rew
            if val_new_cell == 4: #val_key[val_new_cell]['reward'] == 30:
                #primary_obj = np.argwhere(self.map == 3)
                ## only allow escape if primary objective claimed
                '''if len(primary_obj) <= 0:
                    # remove the reward point from the map so that it cannot be claimed
                        ↪ again
                    self.old_agent_value = 0.
                    # as this is the escape point flag that we should end the session
                    ret['is_stop'] = True
                else:
                    # remove the reward
                    ret['immediate_reward'] -= rew'''
                # remove the reward point from the map so that it cannot be claimed again
                self.old_agent_value = 0.
                # as this is the escape point flag that we should end the session
                ret['is_stop'] = True
            elif val_new_cell == 3: # val_key[val_new_cell]['reward'] == 50:
                # if we land on the primary objective then remove it from the map so we
                    ↪ cannot claim it again
                self.old_agent_value = 0.

    # enemy movement. Sort out damage to the agent here
    for e_idx in range(len(self.enemies)):
        # get current position
        e_x, e_y = self.enemies[e_idx].positions[self.enemies[e_idx].current_position]
        # get current place in list in case we need to rewind backwards if we hit the
            ↪ player
        e_current_position = self.enemies[e_idx].current_position
        # Take the step
        e_val = self.map[e_y][e_x]
        e_x1, e_y1 = self.enemies[e_idx].step()

        if (e_x1, e_y1) == self.agent_pos:
            # We have hit the player so take off the reward
            rew = key(advanced=True)[e_val]['reward']
            if not np.isnan(rew):
                ret['immediate_reward'] += rew

            # rollback the enemy
            self.enemies[e_idx].current_position = e_current_position
        else:
```

```
            # display the enemy moving
            # print('new enemy pos', e_x1, e_y1, e_val, e_x, e_y)
            self.map[e_y1][e_x1] = e_val
            # Update where it was to be zero
            self.map[e_y][e_x] = 0.

    if ret['immediate_reward'] < 0:
        self.agent_health += ret['immediate_reward']
        if self.agent_health <= 0:
            ret['is_stop'] = True

    # sort out limit
    self.limit -= 1
    if self.limit < 1:
        ret['is_stop'] = True

    # penalise time step if nothing is achieved
    if ret['immediate_reward'] == 0:
        ret['immediate_reward'] = -1

    # enemy count
    ret['enemy_count'] = len(self.enemies)

    # agent view
    ret['agent_view'] = self.agents_view()

    # direction to objective
    ret['obj_direction'] = self.direction_to_objective()

    # agent health
    ret['agent_health'] = self.agent_health
    return ret

def display(self) -> None:
    """Show the map in a nice way"""
    # Some trial & error using https://numpy.org/doc/stable/reference/generated/numpy.
        ↪ array2string.html to get rid of the decimal
    print(np.array2string(self.map, formatter={'float_kind': lambda x: "%.0f" % x}))

def displayImg(self) -> None:
    """Show the map as an image"""
    plt.imshow(self.map)
    plt.title("Advanced␣map")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.show()

def agents_view(self) -> np.array:
    """The agent can see in a 7 by 7 grid around itself. But cannot see through doors or
        ↪ walls (unless it is sitting on a door"""
    ret = np.zeros((7,7))
    # the agent is in the middle
    ret[3, 3] = 5.
```

```
# start ray tracing around the agent. There is almost certainly a more efficient way of doing this
# up
""" for i in range(1, 4):
    val = self.map[self.agent_pos[1] - i][self.agent_pos[0]]
    ret[3 - i, 3] = val
    if val in [1, 2]:
        break
# down
for i in range(1, 4):
    val = self.map[self.agent_pos[1] + i][self.agent_pos[0]]
    ret[3 + i, 3] = val
    if val in [1, 2]:
        break
# left
for i in range(1, 4):
    val = self.map[self.agent_pos[1]][self.agent_pos[0] - i]
    ret[3, 3 - i] = val
    if val in [1, 2]:
        break
# right
for i in range(1, 4):
    val = self.map[self.agent_pos[1]][self.agent_pos[0] + i]
    ret[3, 3 + i] = val
    if val in [1, 2]:
        break """

# primary diagonals and up/down/left/right
for xmod, ymod in [(1, 1), (1, -1), (-1, -1), (-1, 1), (1, 0), (0, 1), (-1, 0), (0, -1)]:
    for i in range(1, 4):
        val = self.map[self.agent_pos[1] + (i * ymod)][self.agent_pos[0] + (i * xmod)]
        ret[3 + (i * ymod), 3 + (i * xmod)] = val
        if val in [1, 2]:
            break
# secondary diagonals (those not blocked by a wall on both closest compass directions)
for block in [[(2, 1), (3, 1), (3, 2)], [(2, -1), (3, -1), (3, -2)], [(-2, 1), (-3, 1), (-3, 2)], [(-2, -1), (-3, -1), (-3, -2)]]: # the ones whose view is determined by the x-axis
    for col, row in block:
        # check that we aren't straying out of the boundary
        if self.agent_pos[1] + row < len(self.map) and self.agent_pos[0] + col < len(self.map[0]):
            val = self.map[self.agent_pos[1] + row][self.agent_pos[0] + col]
            # if the row is below need to alter if we're looking at the row above or below
            if row < 0:
                row_plus_minus = 1
            else:
                row_plus_minus = -1
            # same with the columns
            if col < 0:
```

```
                col_plus_minus = 1
            else:
                col_plus_minus = -1
            val_offset_1 = self.map[self.agent_pos[1] + row][self.agent_pos[0] + col +
                ↪ col_plus_minus]
            val_offset_2 = self.map[self.agent_pos[1] + row + row_plus_minus][self.
                ↪ agent_pos[0] + col + col_plus_minus]


            #print('val', val, self.agent_pos[0] + col - 1, self.agent_pos[1] + row)

            if val_offset_1 in [1, 2] and val_offset_2 in [1, 2]:
                break
            ret[3 + row, 3 + col] = val


    #(1, 2), (1, 3), (2, 3)]: # north east missing regions
    for block in [[(1, 2), (1, 3), (2, 3)], [(1, -2), (1, -3), (2, -3)], [(-1, 2), (-1, 3),
        ↪  (-2, 3)], [(-1, -2), (-1, -3), (-2, -3)]]:
        for col, row in block:
            # check that we aren't straying out of the boundary
            if self.agent_pos[1] + row < len(self.map) and self.agent_pos[0] + col < len(
                ↪ self.map[0]):
                val = self.map[self.agent_pos[1] + row][self.agent_pos[0] + col]

                # if the row is below need to alter if we're looking at the row above or
                    ↪ below
                if row < 0:
                    row_plus_minus = 1
                else:
                    row_plus_minus = -1
                # same with the columns
                if col < 0:
                    col_plus_minus = 1
                else:
                    col_plus_minus = -1

                #val_offset_1 = self.map[self.agent_pos[1] + row - 1][self.agent_pos[0] +
                    ↪ col]
                #val_offset_2 = self.map[self.agent_pos[1] + row + 1][self.agent_pos[0] +
                    ↪ col + 1]
                if self.agent_pos[1] + row - row_plus_minus < len(self.map):
                    val_offset_1 = self.map[self.agent_pos[1] + row - row_plus_minus][self.
                        ↪ agent_pos[0] + col]
                else:
                    val_offset_1 = 0.
                if self.agent_pos[1] + row + row_plus_minus < len(self.map):
                    val_offset_2 = self.map[self.agent_pos[1] + row + row_plus_minus][self.
                        ↪ agent_pos[0] + col + col_plus_minus]
                else:
                    val_offset_2 = 0.

                if val_offset_1 in [1, 2] and val_offset_2 in [1, 2]:
```

```
                    break
                ret[3 + row, 3 + col] = val

    return ret

def direction_to_objective(self) -> tuple:
    """Get the relative direction to the objective. check 3 the primary objective first and
        ↪ then 4 the secondary objective"""
    primary_obj = np.argwhere(self.map == 3)
    #return primary_obj[0], len(primary_obj)
    if len(primary_obj) > 0:
        primary_obj = primary_obj[0]
        return primary_obj[1] - self.agent_pos[0], primary_obj[0] - self.agent_pos[1]
    primary_obj = np.argwhere(self.map == 4)
    if len(primary_obj) > 0:
        primary_obj = primary_obj[0]
        return primary_obj[1] - self.agent_pos[0], primary_obj[0] - self.agent_pos[1]
    # if both have been claimed return 0, 0 as there are no more objectives
    return 0, 0

def convert_observations(self, obs: dict): #-> tuple(np.array, int, bool):
    """Convert the observations from step's dictionary into an array of the agent's view,
        ↪ the reward for the round and whether or not to stop

    The dictionary contains:
is_stop: a boolean, used internally to determine whether to stop the episode
immediate_reward: the points we got in the last round
enemy_count: how many active enemies there are
agent_view: the 7x7 view of the surroundings (values between 0-8)
obj_direction: the relative direction to the objective (0-max width, 0-max height)
agent_health: the health of the agent, between 0-100"""

    # sort out the relative coordinates, so that the directions are divided by the size of
        ↪ the environment
    obj_direction = np.divide(np.array(list(obs['obj_direction'])), np.array([21., 29.])) #
        ↪ TODO dynamically get the size

    # normalise the view of the surroundings
    # use ravel to reshape into a 1 row list and normalise it
    agent_view = obs['agent_view'].ravel() / 8.

    # enemy_count
    enemy_count = np.array(obs['enemy_count'] / 3.) # TODO dynamically get the max number
        ↪ of enemies

    # agent_health
    agent_health = np.array(obs['agent_health'] / 100.)

    # concatenate into a single numpy array, 1 row, 2 + 49 + 1 + 1 = 53 columns between -1
        ↪ & 1
    return np.concatenate([obj_direction, agent_view, [enemy_count], [agent_health]]), obs[
        ↪ 'immediate_reward'], obs['is_stop']
```

```
if __name__ == '__main__':
    # run as python3 -m helpers.advanced_map
    m = AdvancedMap(pos=(5, 8)) # 24
    #print(m.map)
    print(m.enemies)
    print(m.agent_pos)
    m.display()
    #m.displayImg()
    print(m.agents_view())
    print(m.direction_to_objective())

    # perform an action
    #print("---SHOOT---")
    #print(m.step(adv_actions()['shoot']))
    #print("---LEFT---")
    #print(m.step(adv_actions()['left']))
    print("---Up---")
    print(m.step(adv_actions()['up']))
    m.display()
    print("---Up---")
    print(m.step(adv_actions()['up']))
    m.display()
    print("---Up---")
    print(m.step(adv_actions()['up']))
    m.display()
    print("---Left---")
    print(m.step(adv_actions()['left']))
    m.display()

    # manual testing
    while True:
        # get the user's input
        inp = input("Press␣a␣direction␣or␣space:␣")

        # test the input, map to actions and quit if no match
        if inp == "a":
            print('--LEFT--')
            print(m.step(adv_actions()['left']))
            m.display()
            print('Agent␣position:',m.agent_pos)
        elif inp == 'w':
            print('--UP--')
            print(m.step(adv_actions()['up']))
            m.display()
            print('Agent␣position:',m.agent_pos)
        elif inp == 's':
            print('--DOWN--')
            print(m.step(adv_actions()['down']))
            m.display()
            print('Agent␣position:',m.agent_pos)
        elif inp == 'd':
            print('--RIGHT--')
            print(m.step(adv_actions()['right']))
```

```
            m.display()
            print('Agent␣position:',m.agent_pos)
        elif inp == '␣':
            print("---SHOOT---")
            print(m.step(adv_actions()['shoot']))
            m.display()
            print('Agent␣position:',m.agent_pos)
        else:
            # Quit text which seemed like a good idea at the time
            print("Goodbye")
            break
```

Listing 2: helpers/AdvancedMap.csv

```
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,4,1
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,1
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1
1,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1
1,0,0,0,0,7,0,0,1,0,1,0,0,0,0,0,0,0,0,0,1
1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,1
1,1,1,1,1,0,0,0,1,0,1,1,1,1,7,0,0,0,0,0,1
1,0,0,0,0,0,0,0,2,0,1,0,1,0,0,0,0,0,0,0,1
1,0,0,0,0,0,0,0,2,0,1,0,1,0,0,1,1,1,0,0,1
1,0,0,0,0,0,0,0,2,0,1,0,1,0,0,1,0,0,0,0,1
1,1,1,1,1,2,1,1,1,1,1,2,1,1,0,1,0,0,0,0,1
1,0,0,0,1,0,1,3,0,0,0,0,0,1,0,1,1,1,1,1,1
1,0,0,0,1,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1
1,0,0,0,1,0,1,0,1,0,0,0,0,1,0,0,0,0,0,0,1
1,0,0,0,0,0,1,0,1,0,0,0,0,1,0,0,0,0,0,0,1
1,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1
1,0,0,0,0,0,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1
1,0,0,0,1,1,1,0,0,0,0,0,0,1,0,0,0,0,0,0,1
1,0,0,0,1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,1
1,0,0,0,1,0,0,0,8,0,0,0,0,1,0,0,0,0,0,0,1
1,0,0,0,1,0,6,0,0,6,0,0,0,1,0,0,0,0,0,0,1
1,0,0,0,1,1,1,2,2,1,1,1,1,1,2,1,0,0,0,0,1
1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,1
1,0,0,1,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1
1,0,0,1,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1
1,0,0,1,0,0,1,0,0,1,0,0,0,0,0,1,0,0,0,0,1
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1
```

Listing 3: helpers/enemy.py

```
"""Defines the enemy class"""

class Enemy():
    """An enemy, goes from state 0 -> 1 -> 2 -> 1 -> repeats"""
    def __init__(self, positions) -> None:
        # By copying state 1 (the mid point), we don't have to do any advanced logic
        self.positions = positions + [positions[1]]
        self.current_position = 1
```

```
    def step(self) -> tuple:
        """Take the next step and return the position"""
        self.current_position += 1
        if self.current_position >= len(self.positions):
            self.current_position = 0
        return self.positions[self.current_position]

    def __repr__(self) -> str:
        # This is so the class prints out nicely
        return f'Enemy({self.positions[:-1]})'

    def check_if_near_point(self, x: int, y: int, close=3) -> bool:
        """Check if the enemy is near a point"""
        e_x, e_y = self.positions[self.current_position]
        if abs(e_x - x) <= close and abs(e_y - y) <= close:
            return True
        return False

if __name__ == '__main__':
    # manual testing
    a = Enemy([1, 2, 3])
    print(a.positions)
    print(a.step())
    print(a)
```

Listing 4: helpers/get available actions.py

```
# originally from the lab available_actions = np.where(~np.isnan(R[s]))[0]

import numpy as np
from collections import namedtuple

Action = namedtuple("Action", ["index", "x", "y", "shoot"])

def get_available_actions(r_matrix: np.array, current_state: int) -> list:
    """Get the actions available for this state"""
    return np.where(~np.isnan(r_matrix[current_state]))[0]

def adv_actions() -> dict:
    """This is for the advanced map. left, right, up, down, fire gun"""

    return {name: Action(idx, x, y, shoot) for name, idx, x, y, shoot in [
        ('left', 0, -1, 0, False),
        ('right', 1, 1, 0, False),
        ('up', 2, 0, -1, False),
        ('down', 3, 0, 1, False),
        ('shoot', 4, 0, 0, True) # handle this one somewhere else, any enemy in the
            ↪ neighbouring cells are destroyed
    ]}

def adv_action_from_index(idx: int) -> Action:
    """Return an action from an index"""
    actions_array = {act.index: act for i, act in adv_actions().items()}
```

```
    return actions_array[idx]

if __name__ == "__main__":
    print(adv_actions())
```

Listing 5: helpers/map.py

```
# This is just to store the maps
# Imports
import numpy as np
import os.path

def key(advanced=False) -> dict:
    """What the numbers mean. More for informational purposes than anything else."""
    return {
        0: {'desc': "Empty space", 'reward': 0, 'solid': False},
        1: {'desc': "A wall", 'reward': 0, 'solid': True},
        2: {'desc': "A door", 'reward': 0, 'solid': False},
        3: {'desc': "The primary goal (the reactor control panel)", 'reward': 1000 if advanced
            ↪ else 50, 'solid': False},
        4: {'desc': "The secondary goal (the escape route)", 'reward': 500 if advanced else 30,
            ↪  'solid': False},
        5: {'desc': "The agent", 'reward': 0, 'solid': False},
        6: {'desc': "A trap", 'reward': -10, 'solid': False},
        7: {'desc': "An enemy which bobs up and down", 'reward': -10, 'solid': True},
        8: {'desc': 'An enemy which bobs side to side', 'reward': -10, 'solid': True}
    }

def basic_map() -> np.array:
    """This is the basic game map"""
    return np.array([
        [1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 4, 2, 0, 0, 0, 2, 0, 1],
        [1, 1, 1, 1, 0, 1, 1, 1, 1],
        [1, 0, 0, 0, 0, 0, 0, 0, 1],
        [1, 0, 6, 0, 0, 0, 6, 0, 1],
        [1, 2, 1, 1, 1, 2, 1, 2, 1],
        [1, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 3, 0, 0, 1, 0, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1]
    ])

def adv_map() -> np.array:
    """Get the advanced map"""
    # reads the Advanced map csv file and converts it into a numpy array (I found the function
        ↪  from stackoverflow, but used numpy's 'help' to grab the delimiter)

    #return np.genfromtxt("helpers/AdvancedMap.csv", delimiter = ",")
    pth = os.path.abspath(os.path.dirname(__file__)) # from https://stackoverflow.com/
        ↪ questions/40416072/reading-file-using-relative-path-in-python-project
    pth = os.path.join(pth, "AdvancedMap.csv")
    return np.genfromtxt(pth, delimiter=',')

if __name__ == "__main__":
```

```
    #print(basic_map())
    print(adv_map())
```

Listing 6: helpers/q matrix.py

```
"""Here we setup the Q-matrix, which is just an array of zeros"""

# imports
import numpy as np
from .states_and_actions import states

def q_matrix(map: np.array) -> np.array:
    """Get the empty Q Matrix"""
    # get the states and actions list
    S = states(map)
    actions = states(map)

    # set up the empty r matrix
    return np.zeros((len(S), len(actions)))
```

Listing 7: helpers/r matrix.py

```
# Here we make the r matrix for the basic map. Where an
# agent can go from one cell to another as long as it isn't 1 and
# attached on the map

# imports
import numpy as np
from .states_and_actions import states
from .map import key
from .q_matrix import q_matrix

# print out the numpy array without it truncating
import sys
np.set_printoptions(threshold=sys.maxsize) # https://stackoverflow.com/questions/1987694/how-
    ↪ to-print-the-full-numpy-array-without-truncation

def r_matrix(map: np.array) -> np.array:
    """Here we make the r matrix for the basic map. Where an
agent can go from one cell to another as long as it isn't 1 and
attached on the map"""
    # get the states list
    S = states(map)

    # set up the empty r matrix
    # rmat = np.zeros((len(S), len(actions))) * np.nan
    rmat = q_matrix(map) * np.nan

    height = len(map)
    width = len(map[0])

    # to simplify can only move up, down, left, right
    # rows are where the agent currently is
    # columns are where the agent is going to
```

```
    # from cell 10 the agent can move to cell 11, but all other routes are blocked
    # from cell 11 the agent can move to cell 10 and get the reward or go to cell 12

    rewardDict = key()

    for cell in S:
        # integer divide by 9 to get the y position, modulus to get the x
        y_pos = cell // height
        x_pos = cell % width
        # get the value of the cells around it
        if map[y_pos, x_pos] != 1:
            if y_pos > 0:
                # Check cell above
                above = map[y_pos - 1, x_pos]
                if above != 1:
                    rmat[cell, (y_pos - 1) * 9 + x_pos] = rewardDict[above]['reward']
            if y_pos < height - 1:
                # check cell below
                below = map[y_pos + 1, x_pos]
                if below != 1:
                    rmat[cell, (y_pos + 1) * 9 + x_pos] = rewardDict[below]['reward']

            if x_pos > 0:
                # check the cell to the left
                left = map[y_pos, x_pos - 1]
                if left != 1:
                    rmat[cell, y_pos * 9 + x_pos - 1] = rewardDict[left]['reward']
            if x_pos < width - 1:
                # check the cell to the right
                right = map[y_pos, x_pos + 1]
                if right != 1:
                    rmat[cell, y_pos * 9 + x_pos + 1] = rewardDict[right]['reward']
        # print(cell, y_pos, x_pos)#, rmat[cell])
    return rmat
```

Listing 8: helpers/random start.py

```
"""Pick a random state to start in"""

# imports
import numpy as np
from .states_and_actions import states

def random_start(given_map: np.array, random_state=None) -> int:
    """Pick a random state to start in which is empty"""
    pos_states = states(given_map)

    height = len(given_map)
    width = len(given_map[0])

    value_of_s = 1
    num_states = len(pos_states)
    s = -1
```

```
    if random_state is None:
        while(value_of_s != 0) :
            s = np.random.randint(0, num_states)
            value_of_s = given_map[s // height, s % width]
    else:
        # this is to make sure the random numbers are random when this is working in parallel
        while value_of_s != 0:
            s = random_state.randint(0, num_states)
            value_of_s = given_map[s // height, s % width]
    return s
```

Listing 9: helpers/states and actions.py

```
# Here we take the map from map and transform it from a 9 by 9 matrix
# into an 81 length array

# imports
import numpy as np

def states(map: np.array) -> list:
    """Return the states from a map"""
    len_rows = len(map)
    if len_rows > 0:
        len_cols = len(map[0])
        return [x for x in range(len_rows * len_cols)]
    else:
        return []
```

Listing 10: Dockerfile

```
# Using some hints from https://code.visualstudio.com/docs/containers/quickstart-python
FROM python:3.8-slim-buster

# Expose the port for jupyter-labs
EXPOSE 8890

ADD requirements.txt .

RUN python -m pip install -r requirements.txt
```

Listing 11: requirements.txt

```
numpy
pytest
pandas
jupyterlab
networkx
matplotlib
torch
ray[rllib]
gym[atari,accept-rom-license]==0.21.0
pyglet
```

Listing 12: tests/helpers test/test adv map.py

```
from helpers.map import adv_map

def test_adv_map():
    assert adv_map().shape == (29,21)
```

Listing 13: tests/helpers test/test basic map.py

```
from helpers.map import basic_map

def test_basic_map():
    assert basic_map().shape == (9,9)
```

Listing 14: tests/helpers test/test enemy.py

```
from helpers.enemy import Enemy

def test_enemy():
    a = Enemy([1, 2, 3])
    assert a.positions == [1, 2, 3, 2]
    assert a.step() == 3
    assert a.step() == 2
    assert a.step() == 1
    assert a.step() == 2
```

Listing 15: tests/helpers test/test get available actions.py

```
from helpers.r_matrix import r_matrix
from helpers.map import basic_map
from helpers.get_available_actions import get_available_actions

def test_get_available_actions():
    m = basic_map()
    r = r_matrix(m)
    assert len(get_available_actions(r, 10)) == 1

    assert len(get_available_actions(r, 11)) == 2
```

Listing 16: tests/helpers test/test q matrix.py

```
from helpers.q_matrix import q_matrix
from helpers.map import basic_map
import numpy as np

def test_q_matrix():
    #assert q_matrix(basic_map()) == np.zeros(81)
    assert np.array_equal(q_matrix(basic_map()), np.zeros((81, 81), float))
```

Listing 17: tests/helpers test/test r matrix.py

```
from helpers.r_matrix import r_matrix
from helpers.map import basic_map
import numpy as np

def test_r_matrix():
    rmat = r_matrix(basic_map())
```

```
    assert rmat.shape == (81,81)

    # Check cell 10
    assert len(np.where(~np.isnan(rmat[10]))[0]) == 1

    # Check cell 11
    assert len(np.where(~np.isnan(rmat[11]))[0]) == 2
```

Listing 18: tests/helpers test/test random start.py

```
from helpers.map import basic_map
from helpers.random_start import random_start
from helpers.states_and_actions import states

def test_random_start():
    m = basic_map()
    s = random_start(m)
    assert s >= 0
    assert s < len(states(m))
    # check the value
    width = len(m[0])
    height = len(m)
    val = m[s // height, s % width]
    assert val != 1
```

Listing 19: tests/helpers test/test states.py

```
from helpers.states_and_actions import states
import numpy as np

def test_states():
    assert states(np.zeros((4,5))) == [x for x in range(4*5)]
```

Space Invaders

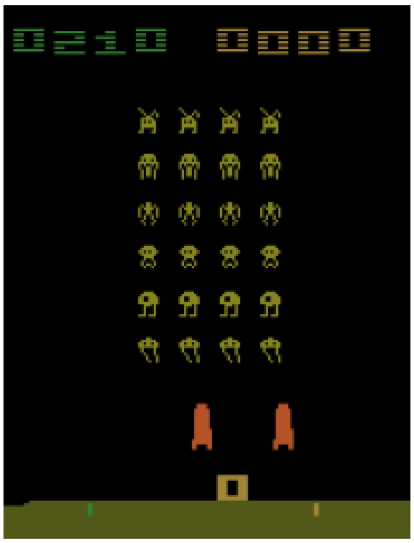
In [8]:
```
# imports
import gym
import matplotlib.pyplot as plt
%matplotlib inline

# for getting the videos
from matplotlib import rc
import matplotlib.animation as animation
rc('animation', html='jshtml')

# getting the evolutionary strategies
import ray.rllib.agents.es as es
# import subprocess to copy the checkpoints
import subprocess
import pickle
from ray import tune
import ray

import pandas as pd
```

Random movements

In [10]:
```
env = gym.make('SpaceInvaders-v4')

obs, info = env.reset(return_info=True)

vid = []
# some inspiration from https://www.gymlibrary.ml/content/api/
for _ in range(1000):
    obs, reward, done, info = env.step(env.action_space.sample())
    vid.append(env.render(mode="rgb_array"))
    if done:
        break
env.close()

# using code from Computer vision lab 5, animation
fig, ax = plt.subplots()
def frame(i: int):
    """Render a frame of video"""
    # dispose of the last frame
    ax.clear()
    # get rid of the axis labels
    ax.axis('off')
    fig.tight_layout()
    # render the agent's view
    return ax.imshow(vid[i])

# build up the video
anim = animation.FuncAnimation(fig, frame, frames=len(vid))
# dispose of the graph object so that the last frame of the video isn't returned
plt.close()
```

```
# set the path to the ffmpeg program (install it using apt-get install ffmpeg, then us
plt.rcParams['animation.ffmpeg_path'] = '/usr/bin/ffmpeg'

writervideo = animation.FFMpegWriter(fps=30) # from https://www.geeksforgeeks.org/how-

anim.save('videos/SpaceInvadersRandom.mp4', writer=writervideo)
anim
```

Out[10]:



Once Loop Reflect

Evolutionary Strategies

In [4]:
```
# get the es config
config = es.DEFAULT_CONFIG.copy()

config['framework'] = 'torch'

# stop it from checking the environment
config['disable_env_checking'] = True
# suppress an error
#config['reuse_actors'] = True
config['env'] = 'SpaceInvaders-v4'
config['num_workers'] = 5
```

In [ ]:
```
ray.init(num_cpus=6)
# use tune (Docker killed off the container at )
stop = {
    "episode_reward_mean": 500,
    "training_iteration": 60,
}
# using code inspired from https://github.com/ray-project/ray/blob/master/rllib/exampl
results = tune.run(
    "ES",
    config=config,
    stop=stop,
    verbose=2,
```

```
    checkpoint_freq=1,
    checkpoint_at_end=True,
    #resources_per_trial={"cpu": 6, "gpu": 0},
    #resources_per_trial=tune.PlacementGroupFactory([{"cpu": 6, "gpu": 0}]),
)
# completed training
checkpoint = results.get_last_checkpoint()
# copy the checkpoints to file
subprocess.run(['cp', checkpoint, 'data/SpaceInvadersES/tune/'], stderr=subprocess.PIP
subprocess.run(['cp', checkpoint + '.tune_metadata', 'data/SpaceInvadersES/tune/'], st
ray.shutdown()
```

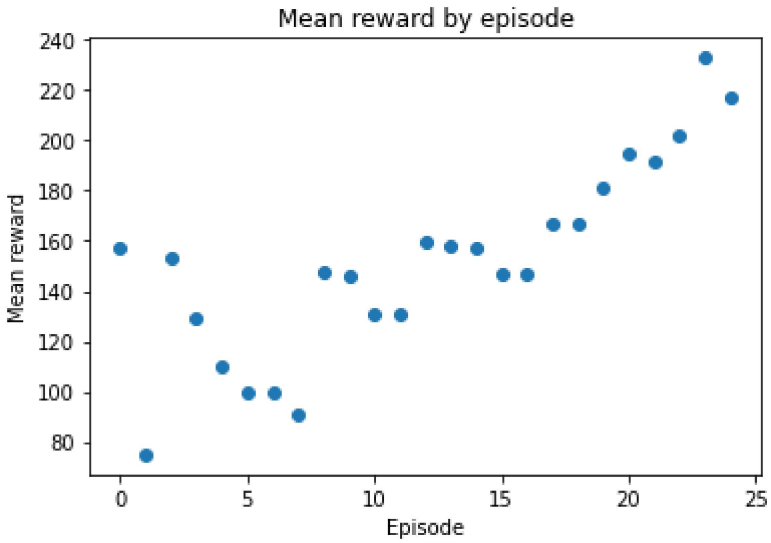
In [ ]:
```
subprocess.run(['cp', '/ray_results/ES/ES_SpaceInvaders-v4_e989d_00000_0_2022-04-22_07
```

In [ ]:
```
trainer = es.ESTrainer(config, env='SpaceInvaders-v4')
mean_reward = []
for ep in range(25):
    result = trainer.train()
    print(f"Episode {ep}, mean reward: {result['episode_reward_mean']}")
    checkpoint = trainer.save()
    # copy the checkpoints to file
    subprocess.run(['cp', checkpoint, 'data/SpaceInvadersES/'], stderr=subprocess.PIPE
    subprocess.run(['cp', checkpoint + '.tune_metadata', 'data/SpaceInvadersES/'], stc
    # append the rewards
    mean_reward.append(result['episode_reward_mean'])
    # save the rewards to file
    with open('data/SpaceInvadersRewards.pkl', 'wb') as f:
        pickle.dump(mean_reward, f)
```

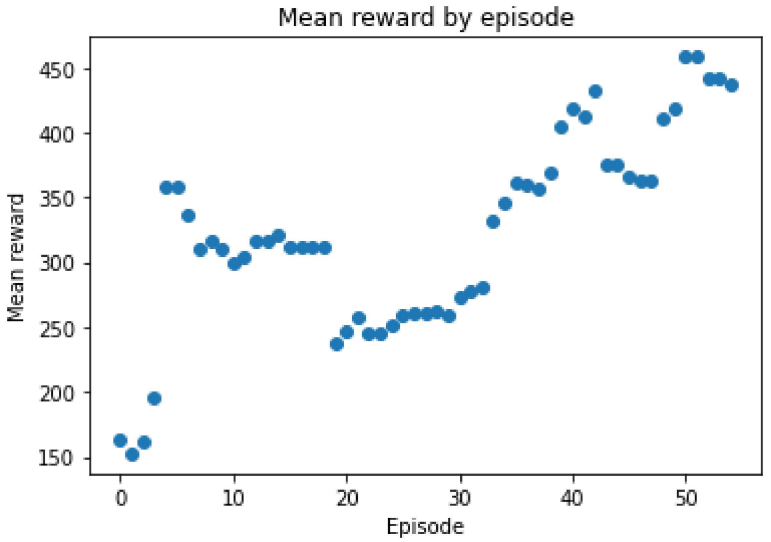
Analyse the data

In [ ]:
```
# instanciate the Evolutionary Strategies model
trainer = es.ESTrainer(config, env='SpaceInvaders-v4')
# load the model weights from the checkpoint
trainer.restore('data/SpaceInvadersES/tune/checkpoint-55')
```

In [29]:
```
# Get the mean reward data back from the pickled array
# open a handle to the pickle file in read bytes mode. Use with so that we don't have
with open('data/SpaceInvadersRewards.pkl', 'rb') as f:
    # use pickle to read back the data into the array
    mean_reward = pickle.load(f)

# graph the data. Use the list, range and len functions chained together to create the
plt.scatter(list(range(len(mean_reward))), mean_reward)
# label the axes
plt.xlabel('Episode')
plt.ylabel('Mean reward')
plt.title('Mean reward by episode')
plt.show()
```



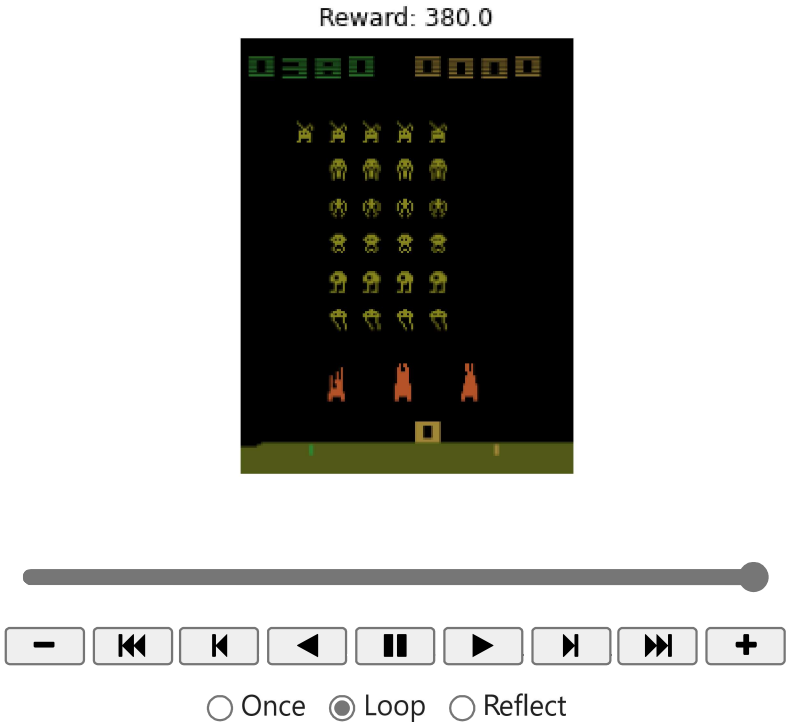
In [9]:
```
# graph the output from Tune
# get the data back
df = pd.read_csv('data/SpaceInvadersES/tune/progress.csv')
# make the graph
plt.scatter(df.index, df['episode_reward_mean'])
# label the axes
plt.xlabel('Episode')
plt.ylabel('Mean reward')
plt.title('Mean reward by episode')
plt.show()
```



In [7]:
```
# Video of the agent playing Space Invaders
total_reward = 0
done = False
env = gym.make('SpaceInvaders-v4')
obs = env.reset()

# store the data for the video
scene = []
total_rewards = []

while not done:
    # work out the next action to take
```

```
    action = trainer.compute_single_action(obs)
    # take the next action
    obs, reward, done, _ = env.step(action)
    # add to the episode reward
    total_reward += reward
    total_rewards.append(total_reward)
    # get the render of the scene and append to the array
    scene.append(env.render(mode="rgb_array"))

# using code from Computer vision lab 5, animation
fig, ax = plt.subplots()

def frame_from_agent(i: int):
    """Render a frame of the video"""
    # wipe the last frame
    ax.clear()
    # get rid of the axis labels
    ax.axis('off')
    # render the image
    plot = ax.imshow(scene[i])
    # show the current total reward
    ax.set_title(f"Reward: {total_rewards[i]}")
    return plot
# build the video
anim = animation.FuncAnimation(fig, frame_from_agent, frames=len(scene))

# set the path to the ffmpeg program (install it using apt-get install ffmpeg, then us
plt.rcParams['animation.ffmpeg_path'] = '/usr/bin/ffmpeg'

writervideo = animation.FFMpegWriter(fps=30) # from https://www.geeksforgeeks.org/how-

anim.save('videos/SpaceInvaders2.mp4', writer=writervideo)

# dispose of the graph object
plt.close()
# show the video
anim
```

Out[7]:

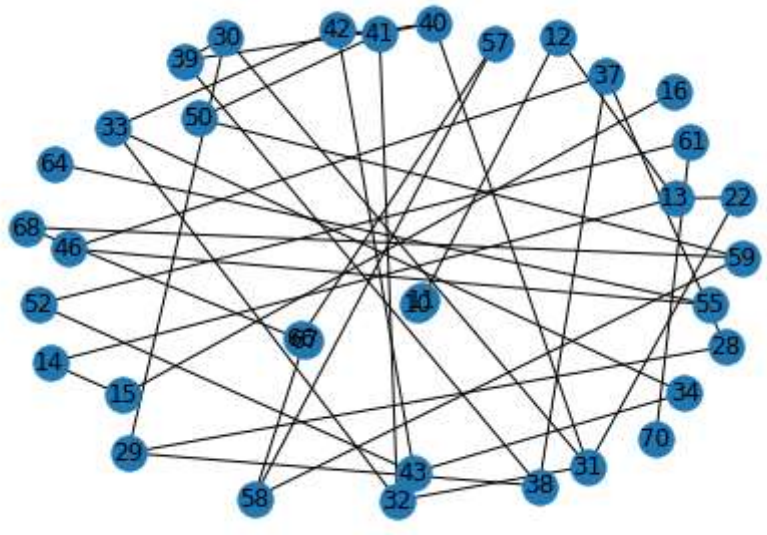


Basic Q-Learning task

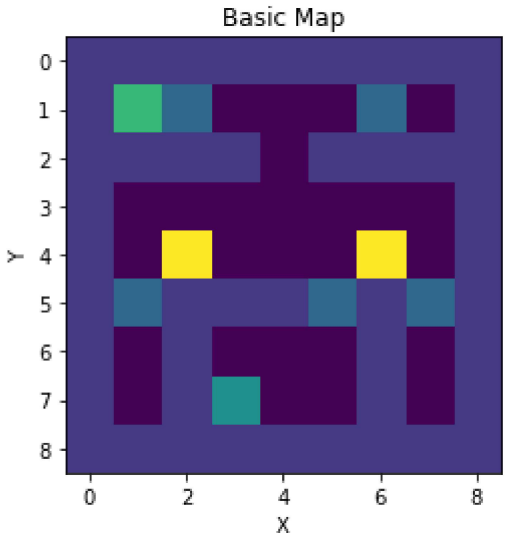
In [2]:
```
# imports
import numpy as np
from helpers.map import basic_map
from helpers.get_available_actions import get_available_actions
from helpers.q_matrix import q_matrix
from helpers.r_matrix import r_matrix
from helpers.random_start import random_start
from helpers.states_and_actions import states
import pandas as pd
from IPython.display import display
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
from functools import partial
from multiprocessing import Pool
```

Graph of Environment

In [3]:
```
# using https://networkx.org/documentation/stable/tutorial.html
# initialise the graph
G = nx.Graph()

# Get the R Matrix to tell which states connect to other states
r = r_matrix(basic_map())

# check whether the state has access to anythere and if it does then add it. Otherwise
ns = [y for y, row in enumerate(r) if len(get_available_actions(r, y)) > 0]
G.add_nodes_from(ns)

# start iterating through the R matrix to get the connections
for y, row in enumerate(r):
    for x, cellweight in enumerate(row):
        # if the cell isn't null add the edge and the weight
        if not np.isnan(cellweight):
            G.add_edge(y, x, weight=cellweight)

# draw the graph
nx.draw(G, with_labels=True)

# Remove the variables
del r
del G
```



In [4]:
```
# Here we use imshow to show the basic map and label the axes
plt.imshow(basic_map())
plt.title('Basic Map')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```



Initialise basic environment

In [5]:
```
def run_q_learning_basic(alpha: float, gamma: float, epsilon: float, num_episodes: int
    show_stats=True, base_map=basic_map(), S=states(basic_map()), goal_state=10,
    rmat=r_matrix(basic_map()), convergence_check_episodes=10, convergence_threshold=0
    """Here we run an episode of Q-learning for the learner
    heavily influenced by Lab 4 code

    As it has a primary and secondary objective, I may need to stop it from repeatedly

    I have moved the map and states definitions out of the function as they don't char

    #base_map = basic_map()
```

```
# States and actions
#S = states(base_map)

#goal_state = 10#66# 10 # TODO get programatically

# R matrix
#rmat = r_matrix(base_map)

# Q matrix
Q = q_matrix(base_map)

# Fixing random state
local_state = np.random.RandomState() # from https://stackoverflow.com/questions/2

# Stats
stats = {
    'num_steps': [],
    'stepPrimaryObj': [],
    'startingStep': [],
    'maxQ': [],
    'total_reward': [],
    'total_health': [],
    'q_var': []
}

# run for episodes
for episode in range(num_episodes):
    s = random_start(base_map)
    #print(f"Starting state is {s}")
    hit_target = False
    stats['startingStep'].append(s)
    total_reward = 0
    total_health = 100
    for step in range(500):
        potential_actions = get_available_actions(rmat, s)

        # get the Q values for these
        q_values = [Q[s, a] for a in potential_actions]

        # get the best actions from the Q values
        max_q = np.max(q_values) # pre calculate ahead of the loop
        best_actions = potential_actions[np.where(q_values == max_q)[0]]
        # get the best actions Q values
        # best_actions_q_values = [Q[s, x] for x in best_actions]

        # determine whether to explore or exploit
        if local_state.uniform() > epsilon:
            a = local_state.choice(potential_actions)
        else:
            a = local_state.choice(best_actions)

        # Get the reward
        reward = rmat[s, a]
        if reward == 50:
            if hit_target:
                reward = 0
            else:
                hit_target = True
                stats['stepPrimaryObj'].append(step)
```

```
            elif reward == -10:
                # Check if it hit a trap
                total_health -= 10

            old_state = s
            s = a

            total_reward += reward

            # Update Q Value
            Q[old_state, a] = Q[old_state, a] + alpha * reward + gamma * (max(Q[s]) -

            # check if goal reached
            if S[s] == goal_state or total_health <= 0:
                # print("Hit goal!")
                break
        # start filling in the statistics
        stats['num_steps'].append(step + 1)
        if not hit_target:
            stats['stepPrimaryObj'].append(-1)
        # cap the Q values statistics decimal places
        stats['maxQ'].append(Q.max().round(1))
        stats['total_reward'].append(total_reward)
        stats['total_health'].append(total_health)
        stats['q_var'].append(Q.var())

        # Early stopping
        # if Q's variance has stayed the same for 10 episodes then assume it has conve
        if episode > convergence_check_episodes and np.var(stats['q_var'][-1 * converg
            break

    if show_stats:
        #print(f"End of episode {episode} Q matrix:\n{Q.round(1)}")
        display(pd.DataFrame(Q.round(1)))
        # put the statistics using pandas
        display(pd.DataFrame.from_dict(stats))

    # return the statistics
    return {
        'alpha': alpha,
        'gamma': gamma,
        'epsilon': epsilon,
        'mean_reward': np.mean(stats['total_reward']), # don't cast the types
        'mean_total_health': np.mean(stats['total_health']),
        'q_var': Q.mean(), # don't take the variance of the variance, take the mean
        'num_episodes': episode + 1 # The episodes until convergence
    }
run_q_learning_basic(1, 0.8, 0.9, 1000)
```

	0	1	2	3	4	5	6	7	8	9	...	71	72	73	74	75	76	77	78	79	80
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
4	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
76	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
77	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
78	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
79	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
80	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

81 rows × 81 columns

	num_steps	stepPrimaryObj	startingStep	maxQ	total_reward	total_health	q_var
0	218	79	22	30.0	10.0	30	4.246876e-01
1	500	2	57	30.9	50.0	100	1.711619e+00
2	500	181	22	46.1	30.0	80	4.121769e+00
3	500	60	57	61.4	50.0	100	7.467843e+00
4	54	-1	70	61.4	20.0	90	7.615209e+00
...	...	...	...	...	...	...	...
995	500	8	61	36383.9	40.0	90	1.222603e+07
996	500	8	33	36425.5	50.0	100	1.223797e+07
997	500	6	43	36467.2	40.0	90	1.224739e+07
998	500	3	68	36508.9	50.0	100	1.225241e+07
999	500	4	41	36550.5	50.0	100	1.226103e+07

1000 rows × 7 columns

Out[5]:
```
{'alpha': 1,
 'gamma': 0.8,
 'epsilon': 0.9,
 'mean_reward': 45.75,
 'mean_total_health': 96.89,
 'q_var': 377.9426438978455,
 'num_episodes': 1000}
```

In [6]:
```
# Grid search
def run_q_learning_basic_tuple(alpha_gamma_epsilon, num_episodes, base_map=None, S=Nor
    """Run run_q_learning from a tuple for the map call"""
```

```
    a, g, e = alpha_gamma_epsilon
    return run_q_learning_basic(a, g, e, num_episodes=num_episodes, show_stats=False,

alphas = [0.0, 0.5, 1.0]
gammas = [0.7, 0.8, 0.9]
epsilons = [0.7, 0.8, 0.9]
def grid_search(alphas: list, gammas: list, epsilons: list, num_episodes=1000, converg
    """Here we loop through all of the alphas, gammas and epsilons for the function ar
    # Build a dictionary to hold the results
    #ret = {
    #    'alpha': [],
    #    'gamma': [],
    #    'epsilon': [],
    #    'mean_reward': [],
    #    'mean_total_health': [],
    #    'q_var': []
    #}
    # start looping
    #for a in alphas:
    #    for g in gammas:
    #        for e in epsilons:
    # go through all of the combinations of values, unpacking the tuples

    #for a, g, e in [(a, g, e) for a in alphas for g in gammas for e in epsilons]:
    #    # call the function and then pull out the results
    #    d = fn(a, g, e, num_episodes, show_stats=False)
    #    for k, v in d.items():
    #        ret[k].append(v)
    # use map to parallelise going through the iterations

    # create the map
    base_map = basic_map()

    # States and actions
    S = states(base_map)

    # R matrix
    rmat = r_matrix(base_map)
    # map the key-word arguments to the Q-learning function
    func = partial(run_q_learning_basic_tuple, num_episodes=num_episodes, base_map=bas
    # map through each iteration of the different combinations
    # using the multiprocessing Pool version cuts the execution time from ~9 mins to ~
    with Pool() as p: # from https://docs.python.org/3/library/multiprocessing.html#mu
        output = list(p.map(
            #lambda x: run_q_learning_basic(x[0], x[1], x[2], num_episodes=num_episode
            func,
            [(a, g, e) for a in alphas for g in gammas for e in epsilons]
        ))
    # convert the results to a dataframe
    #return pd.DataFrame(ret)
    return pd.DataFrame(output)
grid_search(alphas, gammas, epsilons, num_episodes=20)
```

Out[6]:

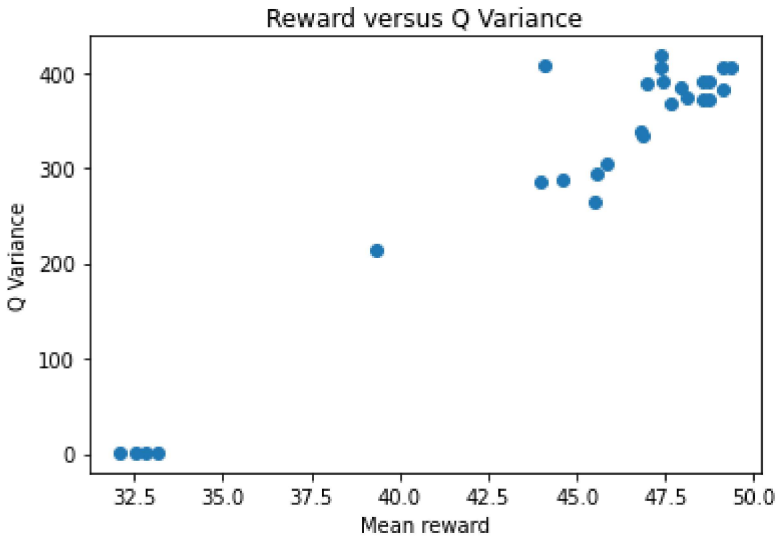
	alpha	gamma	epsilon	mean_reward	mean_total_health	q_var	num_episodes
0	0.0	0.7	0.7	-34.166667	35.833333	0.000000	12
1	0.0	0.7	0.8	-45.000000	21.666667	0.000000	12
2	0.0	0.7	0.9	-34.166667	34.166667	0.000000	12
3	0.0	0.8	0.7	-34.166667	25.833333	0.000000	12
4	0.0	0.8	0.8	-65.000000	9.166667	0.000000	12
5	0.0	0.8	0.9	-44.166667	25.000000	0.000000	12
6	0.0	0.9	0.7	-55.000000	26.666667	0.000000	12
7	0.0	0.9	0.8	-66.666667	7.500000	0.000000	12
8	0.0	0.9	0.9	-50.833333	30.833333	0.000000	12
9	0.5	0.7	0.7	33.000000	88.500000	0.753131	20
10	0.5	0.7	0.8	41.500000	94.500000	0.859353	20
11	0.5	0.7	0.9	42.500000	95.500000	0.698632	20
12	0.5	0.8	0.7	39.500000	91.500000	1.247791	20
13	0.5	0.8	0.8	38.000000	94.000000	0.856638	20
14	0.5	0.8	0.9	41.500000	93.500000	0.772566	20
15	0.5	0.9	0.7	36.000000	89.000000	1.437201	20
16	0.5	0.9	0.8	36.500000	91.500000	0.980372	20
17	0.5	0.9	0.9	40.500000	93.500000	1.061612	20
18	1.0	0.7	0.7	34.000000	88.000000	1.817831	20
19	1.0	0.7	0.8	28.000000	92.000000	0.653457	20
20	1.0	0.7	0.9	34.000000	94.500000	0.602753	20
21	1.0	0.8	0.7	40.000000	92.500000	2.272899	20
22	1.0	0.8	0.8	40.500000	95.500000	1.670608	20
23	1.0	0.8	0.9	33.500000	94.500000	0.778437	20
24	1.0	0.9	0.7	24.000000	87.000000	0.343063	20
25	1.0	0.9	0.8	33.000000	89.500000	2.139921	20
26	1.0	0.9	0.9	40.500000	95.500000	1.892223	20

In [39]:
```
# Use the grid search to repeatedly run the same values
df = grid_search([1.,1.,1.], [0.8,0.8,0.8], [0.9,0.9,0.9])
df
```

Out[39]:

	alpha	gamma	epsilon	mean_reward	mean_total_health	q_var	num_episodes
0	1.0	0.8	0.9	48.720000	99.400000	371.267320	1000
1	1.0	0.8	0.9	45.820000	99.480000	303.713861	1000
2	1.0	0.8	0.9	49.330000	99.370000	406.278219	1000
3	1.0	0.8	0.9	45.480000	99.500000	265.396383	1000
4	1.0	0.8	0.9	48.550000	99.470000	371.539042	1000
5	1.0	0.8	0.9	46.850000	99.470000	333.777993	1000
6	1.0	0.8	0.9	32.539683	98.730159	1.231849	126
7	1.0	0.8	0.9	39.320000	96.510000	214.615584	1000
8	1.0	0.8	0.9	47.350000	97.370000	417.946645	1000
9	1.0	0.8	0.9	47.930000	99.400000	385.072612	1000
10	1.0	0.8	0.9	44.610000	98.580000	288.005358	1000
11	1.0	0.8	0.9	45.580000	99.420000	294.707453	1000
12	1.0	0.8	0.9	48.710000	99.430000	390.620170	1000
13	1.0	0.8	0.9	49.150000	99.370000	405.354358	1000
14	1.0	0.8	0.9	32.816901	98.591549	1.315782	71
15	1.0	0.8	0.9	46.990000	97.870000	389.029357	1000
16	1.0	0.8	0.9	47.390000	97.490000	405.042943	1000
17	1.0	0.8	0.9	48.100000	99.410000	374.542376	1000
18	1.0	0.8	0.9	32.105263	97.368421	1.312321	57
19	1.0	0.8	0.9	46.790000	99.390000	338.839970	1000
20	1.0	0.8	0.9	49.110000	99.510000	383.215288	1000
21	1.0	0.8	0.9	44.110000	94.710000	407.580874	1000
22	1.0	0.8	0.9	33.157895	97.017544	1.260302	57
23	1.0	0.8	0.9	43.960000	97.800000	286.108644	1000
24	1.0	0.8	0.9	47.420000	97.620000	391.147609	1000
25	1.0	0.8	0.9	47.630000	99.490000	367.988081	1000
26	1.0	0.8	0.9	48.560000	99.340000	390.767508	1000

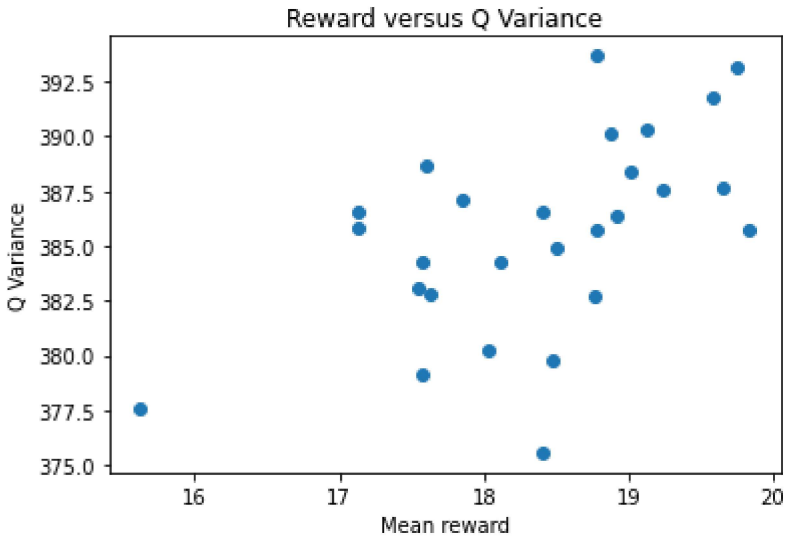
In [40]:
```
# Here we graphically display the output of the Q function
plt.scatter('mean_reward', 'q_var', data=df) # trick from DataCamp to add the datafram
plt.xlabel("Mean reward")
plt.ylabel("Q Variance")
plt.title("Reward versus Q Variance")
plt.show()
```



Repeat using different parameter values, and policies

In [10]:
```
# Grid search on the new parameters to repeatedly show them
# df2 = grid_search([.1, .2, .3, .4, .5, .6, .7, .8, .9, 1.], [.1,.2,.3,.4,.5,.6,.7,.8
# df2 = grid_search([.1, .2, .3, .4, .5, .6, .7, .8, .9, 1.], [.8, .8, .8], [0.4, 0.4,
df2 = grid_search([.8, .8, .8], [.8, .8, .8], [.4, .4, .4])
display(df2)
# Here we graphically display the output of the Q function
plt.scatter('mean_reward', 'q_var', data=df2)
plt.xlabel("Mean reward")
plt.ylabel("Q Variance")
plt.title("Reward versus Q Variance")
plt.show()
```

	alpha	gamma	epsilon	mean_reward	mean_total_health	q_var	num_episodes
0	0.8	0.8	0.4	18.92	69.70	386.334365	1000
1	0.8	0.8	0.4	18.03	69.11	380.258623	1000
2	0.8	0.8	0.4	17.85	68.87	387.093522	1000
3	0.8	0.8	0.4	17.55	68.47	383.073599	1000
4	0.8	0.8	0.4	17.63	68.56	382.800007	1000
5	0.8	0.8	0.4	17.58	68.76	379.157672	1000
6	0.8	0.8	0.4	17.13	68.06	385.787207	1000
7	0.8	0.8	0.4	18.41	69.52	375.568519	1000
8	0.8	0.8	0.4	18.48	69.21	379.809655	1000
9	0.8	0.8	0.4	18.12	68.89	384.229696	1000
10	0.8	0.8	0.4	17.60	68.19	388.671521	1000
11	0.8	0.8	0.4	19.02	69.66	388.411031	1000
12	0.8	0.8	0.4	19.66	70.36	387.627485	1000
13	0.8	0.8	0.4	19.75	70.13	393.126553	1000
14	0.8	0.8	0.4	19.58	70.01	391.799021	1000
15	0.8	0.8	0.4	18.41	69.16	386.586802	1000
16	0.8	0.8	0.4	19.84	70.53	385.699579	1000
17	0.8	0.8	0.4	18.51	69.27	384.919749	1000
18	0.8	0.8	0.4	15.62	66.81	377.571725	1000
19	0.8	0.8	0.4	17.57	68.17	384.297585	1000
20	0.8	0.8	0.4	18.78	69.25	393.684615	1000
21	0.8	0.8	0.4	19.13	69.60	390.277500	1000
22	0.8	0.8	0.4	18.76	69.66	382.668195	1000
23	0.8	0.8	0.4	18.78	69.58	385.756547	1000
24	0.8	0.8	0.4	19.24	69.81	387.586935	1000
25	0.8	0.8	0.4	17.13	67.84	386.589141	1000
26	0.8	0.8	0.4	18.88	69.38	390.137272	1000



In [11]:
```
df2.to_csv("data/NewValues.csv")
```

Basic Task Data Analysis

Look at full grid search

In [5]:
```
# imports
import matplotlib.pyplot as plt
import pandas as pd
%matplotlib inline
```

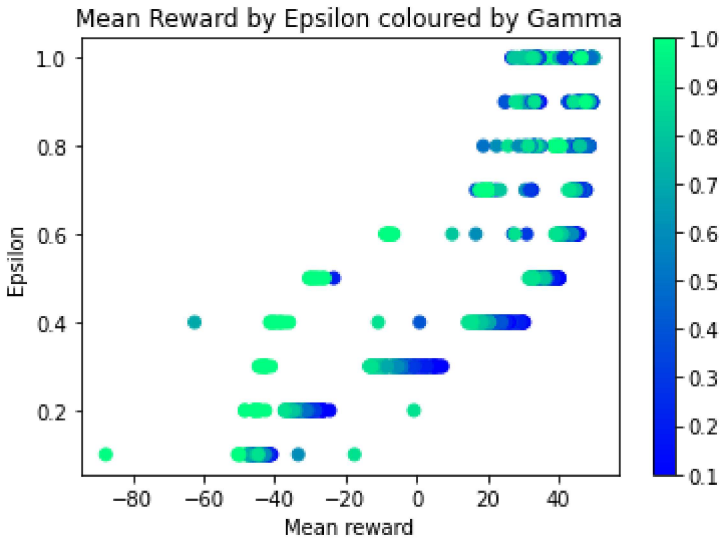
In [4]:
```
# As the full grid search took over an hour to run, I stored the data
full_grid_df = pd.read_csv("data/fullGridSearch.csv")
full_grid_df.drop('Unnamed: 0', axis=1, inplace=True)
full_grid_df
```

Out[4]:

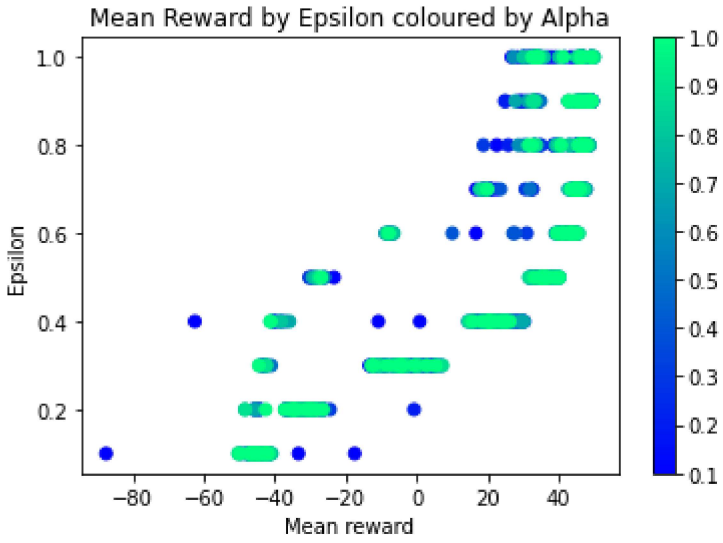
	alpha	gamma	epsilon	mean_reward	mean_total_health	q_var	num_episodes
0	0.1	0.1	0.1	-43.00	17.32	14.880943	1000
1	0.1	0.1	0.2	-27.46	26.72	22.899868	1000
2	0.1	0.1	0.3	6.92	57.87	27.505989	1000
3	0.1	0.1	0.4	29.99	80.21	27.585452	1000
4	0.1	0.1	0.5	39.00	89.52	25.401114	1000
...	...	...	...	...	...	...	...
995	1.0	1.0	0.6	-7.85	44.69	559.202865	1000
996	1.0	1.0	0.7	19.88	70.48	597.690901	1000
997	1.0	1.0	0.8	40.61	90.83	580.576132	1000
998	1.0	1.0	0.9	47.55	98.36	548.750191	1000
999	1.0	1.0	1.0	46.44	99.94	153.802774	1000

1000 rows × 7 columns

In [21]:
```
plt.scatter('mean_reward', 'epsilon', c='gamma', cmap='winter', data=full_grid_df)
plt.xlabel('Mean reward')
plt.ylabel('Epsilon')
plt.colorbar()
plt.title('Mean Reward by Epsilon coloured by Gamma')
plt.show()
```



In [22]:
```
plt.scatter('mean_reward', 'epsilon', c='alpha', cmap='winter', data=full_grid_df)
plt.xlabel('Mean reward')
plt.ylabel('Epsilon')
plt.colorbar()
plt.title('Mean Reward by Epsilon coloured by Alpha')
plt.show()
```



In [23]:
```
full_grid_df.num_episodes.mean()
```

Out[23]:
```
888.502
```

Look at grid search gamma

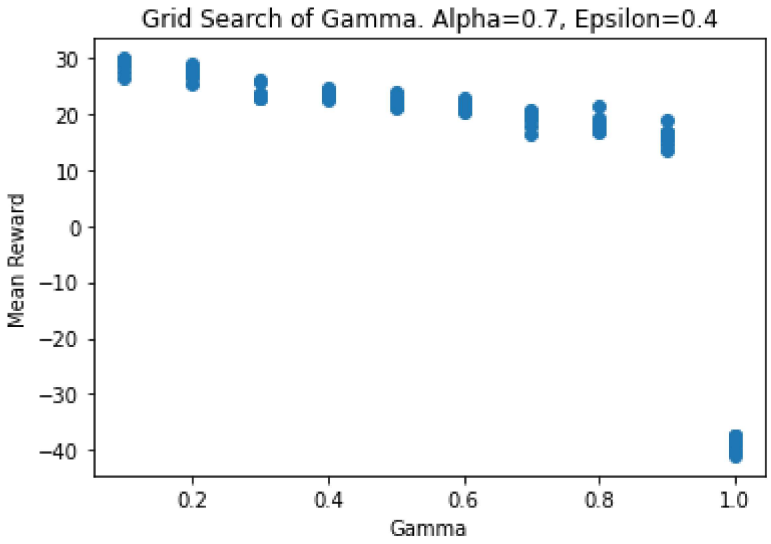
In [25]:
```
grid_gamma_df = pd.read_csv("data/GridSearchGamma.csv")
grid_gamma_df.drop('Unnamed: 0', axis=1, inplace=True)
grid_gamma_df
```

Out[25]:

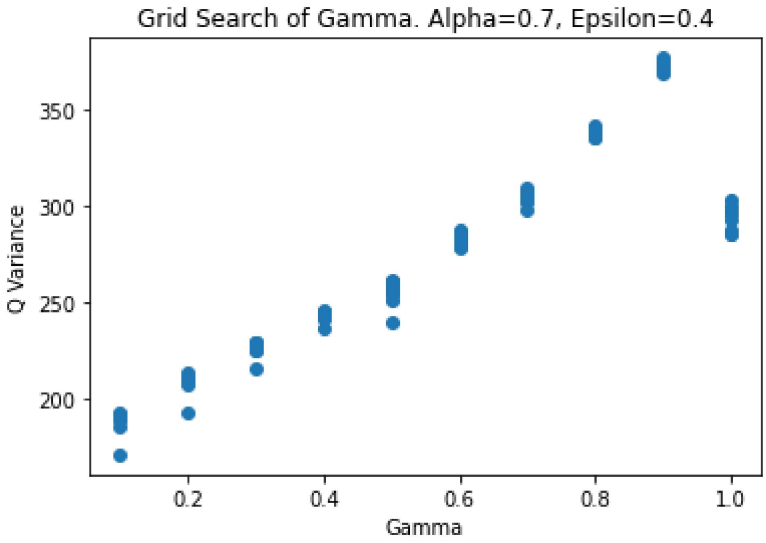
	alpha	gamma	epsilon	mean_reward	mean_total_health	q_var	num_episodes
0	0.7	0.1	0.4	29.64	80.15	189.431498	1000
1	0.7	0.1	0.4	28.98	79.51	189.925789	1000
2	0.7	0.1	0.4	28.35	78.90	190.980150	1000
3	0.7	0.2	0.4	27.20	77.66	211.593114	1000
4	0.7	0.2	0.4	26.89	77.67	207.922045	1000
...	...	...	...	...	...	...	...
85	0.7	0.9	0.4	15.54	66.29	376.894621	1000
86	0.7	0.9	0.4	16.57	67.12	372.003586	1000
87	0.7	1.0	0.4	-37.36	20.63	300.105929	1000
88	0.7	1.0	0.4	-39.10	19.01	296.417619	1000
89	0.7	1.0	0.4	-40.14	18.80	296.764365	1000

90 rows × 7 columns

In [29]:
```
plt.scatter('gamma', 'mean_reward', data=grid_gamma_df)
plt.xlabel('Gamma')
plt.ylabel('Mean Reward')
plt.title('Grid Search of Gamma. Alpha=0.7, Epsilon=0.4')
plt.show()
```



In [35]:
```
plt.scatter('gamma', 'q_var', data=grid_gamma_df)
plt.xlabel('Gamma')
plt.ylabel('Q Variance')
plt.title('Grid Search of Gamma. Alpha=0.7, Epsilon=0.4')
plt.show()
```



Look at Grid Search Alpha

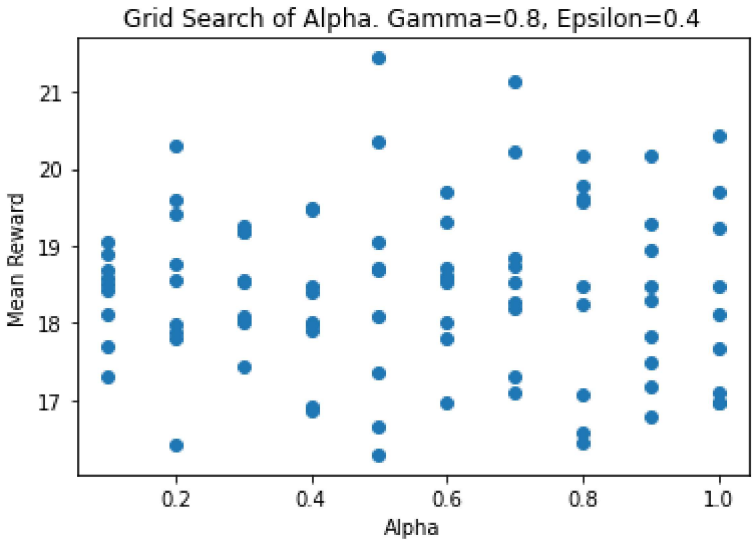
In [30]:
```
grid_alpha_df = pd.read_csv("data/GridSearchAlpha.csv")
grid_alpha_df.drop('Unnamed: 0', axis=1, inplace=True)
grid_alpha_df
```

Out[30]:

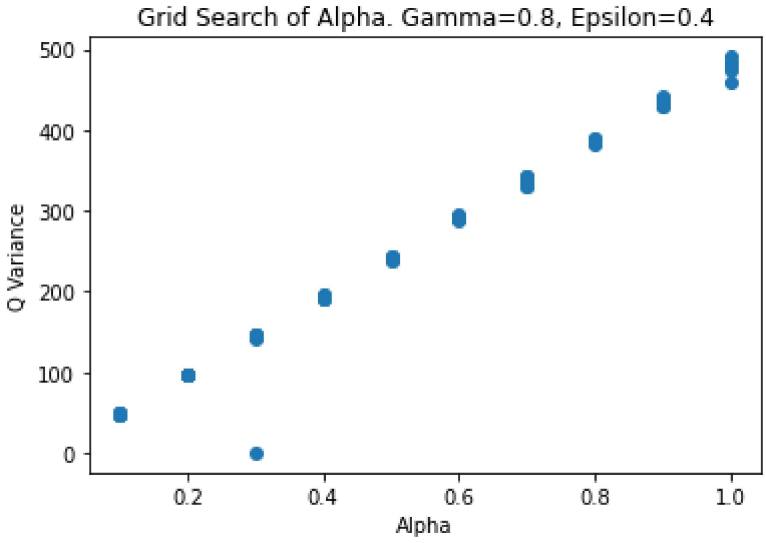
	alpha	gamma	epsilon	mean_reward	mean_total_health	q_var	num_episodes
0	0.1	0.8	0.4	18.89	69.68	47.684038	1000
1	0.1	0.8	0.4	18.51	69.04	48.483852	1000
2	0.1	0.8	0.4	17.69	68.64	47.934443	1000
3	0.1	0.8	0.4	18.59	69.29	48.229333	1000
4	0.1	0.8	0.4	17.31	68.05	48.102333	1000
...	...	...	...	...	...	...	...
85	1.0	0.8	0.4	19.24	70.31	474.072409	1000
86	1.0	0.8	0.4	20.43	70.95	490.864236	1000
87	1.0	0.8	0.4	17.09	68.07	479.883711	1000
88	1.0	0.8	0.4	16.96	67.79	480.559351	1000
89	1.0	0.8	0.4	16.97	68.13	479.569275	1000

90 rows × 7 columns

In [31]:
```
plt.scatter('alpha', 'mean_reward', data=grid_alpha_df)
plt.xlabel('Alpha')
plt.ylabel('Mean Reward')
plt.title('Grid Search of Alpha. Gamma=0.8, Epsilon=0.4')
plt.show()
```



In [36]:
```
plt.scatter('alpha', 'q_var', data=grid_alpha_df)
plt.xlabel('Alpha')
plt.ylabel('Q Variance')
plt.title('Grid Search of Alpha. Gamma=0.8, Epsilon=0.4')
plt.show()
```



Analyse results quantitively and qualitatively

In [37]:
```
org_df = pd.read_csv('data/OrgValues.csv')
org_df.drop('Unnamed: 0', axis=1, inplace=True)
org_df
```

Out[37]:

	alpha	gamma	epsilon	mean_reward	mean_total_health	q_var	num_episodes
0	1.0	0.8	0.9	48.720000	99.400000	371.267320	1000
1	1.0	0.8	0.9	45.820000	99.480000	303.713861	1000
2	1.0	0.8	0.9	49.330000	99.370000	406.278219	1000
3	1.0	0.8	0.9	45.480000	99.500000	265.396383	1000
4	1.0	0.8	0.9	48.550000	99.470000	371.539042	1000
5	1.0	0.8	0.9	46.850000	99.470000	333.777993	1000
6	1.0	0.8	0.9	32.539683	98.730159	1.231849	126
7	1.0	0.8	0.9	39.320000	96.510000	214.615584	1000
8	1.0	0.8	0.9	47.350000	97.370000	417.946645	1000
9	1.0	0.8	0.9	47.930000	99.400000	385.072612	1000
10	1.0	0.8	0.9	44.610000	98.580000	288.005358	1000
11	1.0	0.8	0.9	45.580000	99.420000	294.707453	1000
12	1.0	0.8	0.9	48.710000	99.430000	390.620170	1000
13	1.0	0.8	0.9	49.150000	99.370000	405.354358	1000
14	1.0	0.8	0.9	32.816901	98.591549	1.315782	71
15	1.0	0.8	0.9	46.990000	97.870000	389.029357	1000
16	1.0	0.8	0.9	47.390000	97.490000	405.042943	1000
17	1.0	0.8	0.9	48.100000	99.410000	374.542376	1000
18	1.0	0.8	0.9	32.105263	97.368421	1.312321	57
19	1.0	0.8	0.9	46.790000	99.390000	338.839970	1000
20	1.0	0.8	0.9	49.110000	99.510000	383.215288	1000
21	1.0	0.8	0.9	44.110000	94.710000	407.580874	1000
22	1.0	0.8	0.9	33.157895	97.017544	1.260302	57
23	1.0	0.8	0.9	43.960000	97.800000	286.108644	1000
24	1.0	0.8	0.9	47.420000	97.620000	391.147609	1000
25	1.0	0.8	0.9	47.630000	99.490000	367.988081	1000
26	1.0	0.8	0.9	48.560000	99.340000	390.767508	1000

In [38]:

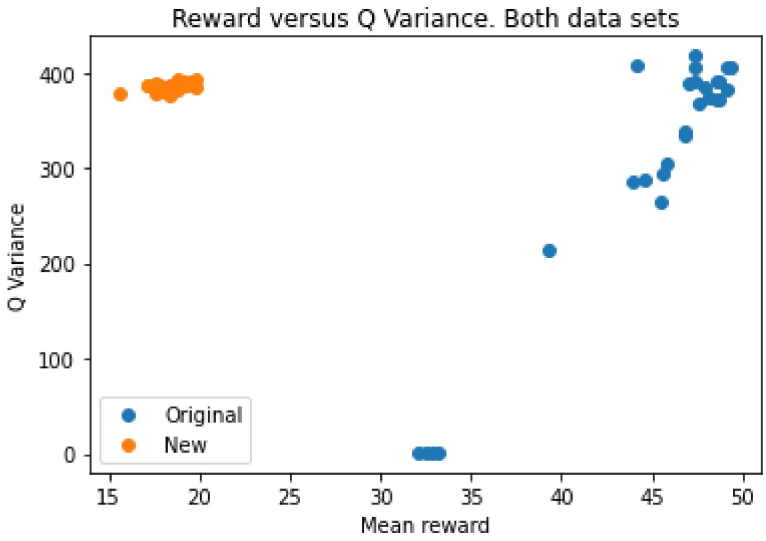
```
new_df = pd.read_csv('data/NewValues.csv')
new_df.drop('Unnamed: 0', axis=1, inplace=True)
new_df
```

Out[38]:

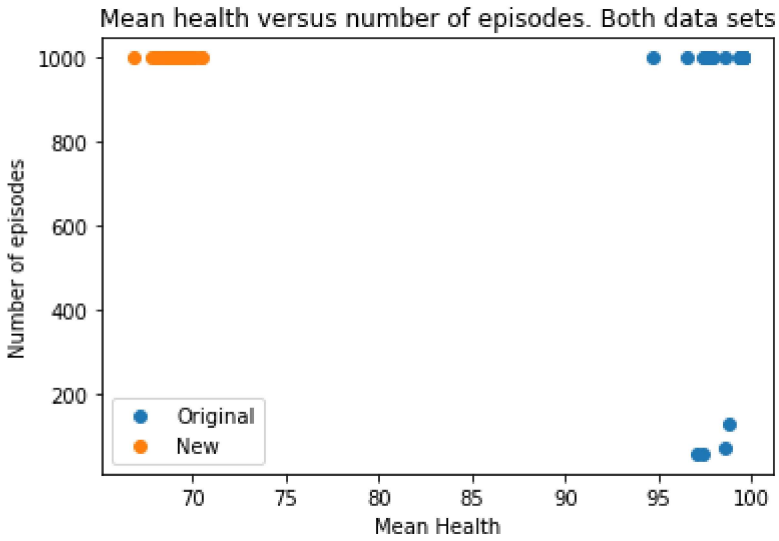
	alpha	gamma	epsilon	mean_reward	mean_total_health	q_var	num_episodes
0	0.8	0.8	0.4	18.92	69.70	386.334365	1000
1	0.8	0.8	0.4	18.03	69.11	380.258623	1000
2	0.8	0.8	0.4	17.85	68.87	387.093522	1000
3	0.8	0.8	0.4	17.55	68.47	383.073599	1000
4	0.8	0.8	0.4	17.63	68.56	382.800007	1000
5	0.8	0.8	0.4	17.58	68.76	379.157672	1000
6	0.8	0.8	0.4	17.13	68.06	385.787207	1000
7	0.8	0.8	0.4	18.41	69.52	375.568519	1000
8	0.8	0.8	0.4	18.48	69.21	379.809655	1000
9	0.8	0.8	0.4	18.12	68.89	384.229696	1000
10	0.8	0.8	0.4	17.60	68.19	388.671521	1000
11	0.8	0.8	0.4	19.02	69.66	388.411031	1000
12	0.8	0.8	0.4	19.66	70.36	387.627485	1000
13	0.8	0.8	0.4	19.75	70.13	393.126553	1000
14	0.8	0.8	0.4	19.58	70.01	391.799021	1000
15	0.8	0.8	0.4	18.41	69.16	386.586802	1000
16	0.8	0.8	0.4	19.84	70.53	385.699579	1000
17	0.8	0.8	0.4	18.51	69.27	384.919749	1000
18	0.8	0.8	0.4	15.62	66.81	377.571725	1000
19	0.8	0.8	0.4	17.57	68.17	384.297585	1000
20	0.8	0.8	0.4	18.78	69.25	393.684615	1000
21	0.8	0.8	0.4	19.13	69.60	390.277500	1000
22	0.8	0.8	0.4	18.76	69.66	382.668195	1000
23	0.8	0.8	0.4	18.78	69.58	385.756547	1000
24	0.8	0.8	0.4	19.24	69.81	387.586935	1000
25	0.8	0.8	0.4	17.13	67.84	386.589141	1000
26	0.8	0.8	0.4	18.88	69.38	390.137272	1000

In [42]:

```
plt.scatter('mean_reward', 'q_var', data=org_df, label='Original')
plt.scatter('mean_reward', 'q_var', data=new_df, label='New')
plt.xlabel("Mean reward")
plt.ylabel("Q Variance")
plt.title("Reward versus Q Variance. Both data sets")
plt.legend()
plt.show()
```

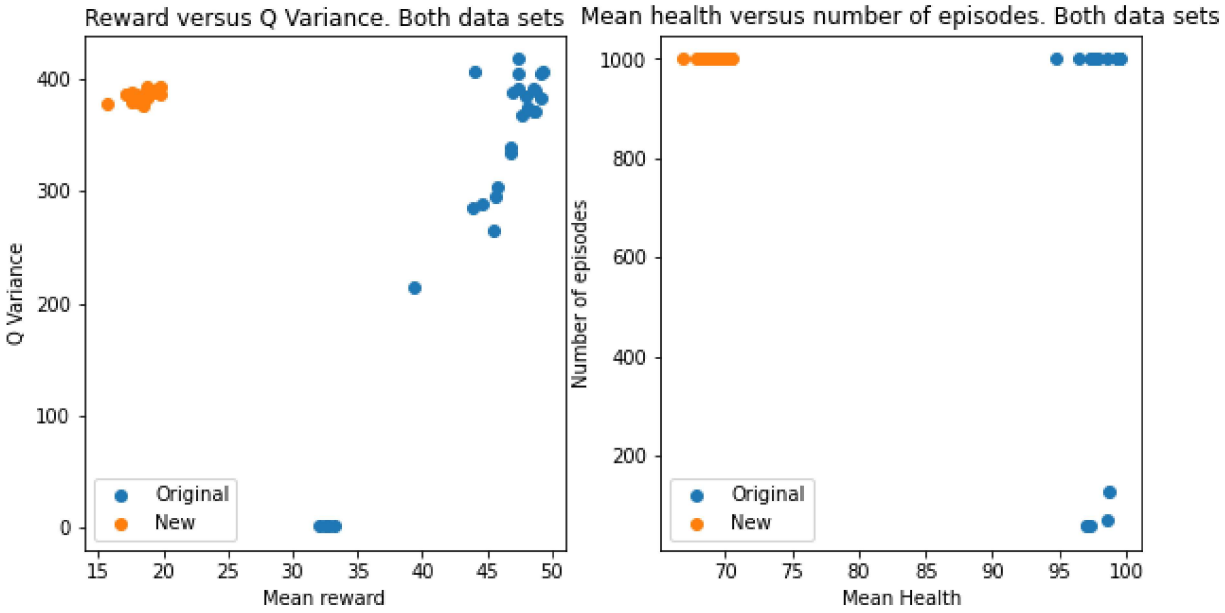


In [44]:
```
plt.scatter('mean_total_health', 'num_episodes', data=org_df, label='Original')
plt.scatter('mean_total_health', 'num_episodes', data=new_df, label='New')
plt.xlabel("Mean Health")
plt.ylabel("Number of episodes")
plt.title("Mean health versus number of episodes. Both data sets")
plt.legend()
plt.show()
```



In [58]:
```
# Put both graphs into one figure
fig, ax = plt.subplots(ncols=2, nrows=1)
ax[0].scatter('mean_reward', 'q_var', data=org_df, label='Original')
ax[0].scatter('mean_reward', 'q_var', data=new_df, label='New')
ax[0].set_xlabel("Mean reward")
ax[0].set_ylabel("Q Variance")
ax[0].set_title("Reward versus Q Variance. Both data sets")
ax[0].legend()
ax[1].scatter('mean_total_health', 'num_episodes', data=org_df, label='Original')
ax[1].scatter('mean_total_health', 'num_episodes', data=new_df, label='New')
ax[1].set_xlabel("Mean Health")
ax[1].set_ylabel("Number of episodes")
ax[1].set_title("Mean health versus number of episodes. Both data sets")
ax[1].legend()
# Adjust the size so the graphs don't crush each other
```

```
fig.set_size_inches((10,5))
plt.show()
```



Setup colab

In [1]:
```
# From Computer vision lab 6, keeps reloading the code for the objects so that you car
%load_ext autoreload
%autoreload 2
```

In [2]:
```
# From lab 7
from google.colab import drive
drive.mount('/content/gdrive')
path = '/content/gdrive/My Drive/Colab Notebooks/DRL/DRL Coursework/'

import sys
sys.path.append(path)
```

```
Mounted at /content/gdrive
```

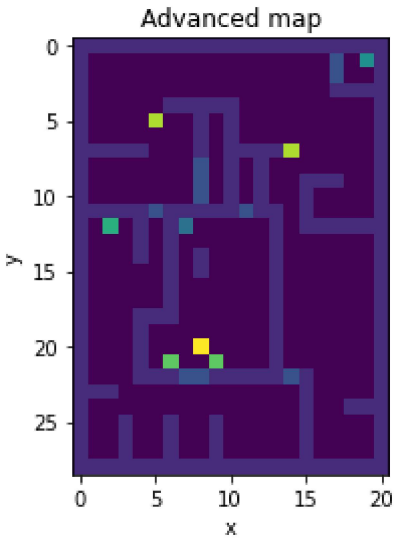
In [ ]:
```
!pip install torch
!pip install -U "ray[rllib]" torch
```

DQN with two improvements

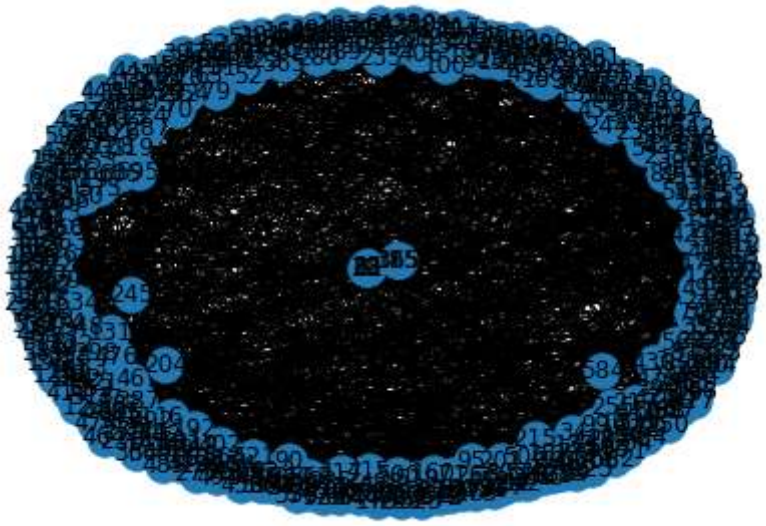
In [2]:
```
# imports
import gym
from ray.rllib.env.env_context import EnvContext
import ray.rllib.agents.dqn as dqn
from helpers.advanced_map import AdvancedMap

from IPython.display import display
import networkx as nx
import matplotlib.pyplot as plt
%matplotlib inline
from helpers.r_matrix import r_matrix
from helpers.get_available_actions import get_available_actions, adv_action_from_index
import numpy as np
import random
import pandas as pd
import subprocess

from multiprocessing import Pool

# for getting the videos
from matplotlib import rc
import matplotlib.animation as animation
rc('animation', html='jshtml')
```

```
/usr/local/lib/python3.8/site-packages/tqdm/auto.py:22: TqdmWarning: IProgress not fo
und. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/s
table/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [3]:
```
m = AdvancedMap()
m.displayImg()
```



In [10]:
```python
# using https://networkx.org/documentation/stable/tutorial.html
# initialise the graph
G = nx.Graph()

# Get the R Matrix to tell which states connect to other states
r = r_matrix(m.map)

# check whether the state has access to anythere and if it does then add it. Otherwise
ns = [y for y, row in enumerate(r) if len(get_available_actions(r, y)) > 0]
G.add_nodes_from(ns)

# start iterating through the R matrix to get the connections
for y, row in enumerate(r):
    for x, cellweight in enumerate(row):
        # if the cell isn't null add the edge and the weight
        if not np.isnan(cellweight):
            G.add_edge(y, x, weight=cellweight)

# draw the graph
nx.draw(G, with_labels=True)

# Remove the variables
del r
del G
```



Advanced Map Wrapper

In [3]:
```
class AdvancedMapEnv(gym.Env):
  """Class that wraps the advanced map to make it compatible with RLLib

  Functions:
  __init__: takes in a config and creates the environment

  convert_observations: takes in a dictionary and returns a numpy array

  seed: set the random seed, with an optional integer

  step: take an action and see what happens

  reset: reset the environment back to the beginning"""

  def __init__(self, config: EnvContext):
    """Starts up the environment

    Inputs:
    config: an EnvContext"""
    # super(AdvancedMapEnv, self).__init__()
    # create an advanced map object
    self.advanced_map = AdvancedMap()

    # Define the action space
    # 5 actions: up, down, left, right, shoot
    self.action_space = gym.spaces.Discrete(5)

    # define the observation space, a 53 column box of floats between -1 and 1
    self.observation_space = gym.spaces.Box(low=-1., high=1., shape=[53,], dtype=np.fl

  def reset(self) -> np.array:
      """Reset the environment and send back the observations"""
      # send through the reset command
      obs = self.advanced_map.reset(pos=None)
      # convert the observations
      return self.convert_observations(obs)
```

```
  def step(self, action: int):
    """take the step, assuming that the action is valid"""
    assert action in [0, 1, 2, 3, 4]

    # get the action
    actionstr = adv_action_from_index(action)

    # pass in the step
    obs = self.advanced_map.step(actionstr)

    rew = obs['immediate_reward']

    # return obs, reward and done
    return self.convert_observations(obs), rew, obs['is_stop'], {}

  def convert_observations(self, obs: dict) -> np.array:
    """Here we take in the dictionary of observations from the environment and returns

    The dictionary contains:
    is_stop: a boolean, used internally to determine whether to stop the episode
    immediate_reward: the points we got in the last round
    enemy_count: how many active enemies there are
    agent_view: the 7x7 view of the surroundings (values between 0-8)
    obj_direction: the relative direction to the objective (0-max width, 0-max height)
    agent_health: the health of the agent, between 0-100"""
    # sort out the relative coordinates, so that the directions are divided by the siz
    obj_direction = np.divide(np.array(list(obs['obj_direction'])), np.array([21., 29.

    # normalise the view of the surroundings
    agent_view = obs['agent_view'].ravel() / 8. # use ravel to reshape into a 1 row li

    # enemy_count
    enemy_count = np.array(obs['enemy_count'] / 3.) # TODO dynamically get the max num

    # agent_health
    agent_health = np.array(obs['agent_health'] / 100.)

    # concatenate into a single numpy array, 1 row, 2 + 49 + 1 + 1 = 53 columns betwee
    return np.concatenate([obj_direction, agent_view, [enemy_count], [agent_health]])

    def seed(self, seed=None) -> None:
      """Set the random seed"""
      random.seed(seed)
```

Start training

In [4]:

```
# helper functions
def training_loop(config: dict, csv_filename: str, checkpoint_folder: str, num_epochs=
  """Train a DQN from the provided config and save the data"""
  # setup the trainer
  trainer = dqn.DQNTrainer(config=config)

  # store the rewards
  ret = {
    'avg_rewards': [],
    'mean_q': []
  }
```

```
  for episode in range(num_epochs):
    # train an episode with DQN
    result = trainer.train()
    # print(result)
    ret['avg_rewards'].append(result['episode_reward_mean'])
    ret['mean_q'].append(result['info']['learner']['default_policy']['learner_stats'][

    if is_save:
      print(f"Episode {episode}, mean reward: {result['episode_reward_mean']}, mean Q:
    # early stopping if mean reward doesn't improve for a while (as it seems to improv
    # from lab 7
    if episode % 10 == 0 and is_save:
      # checkpoint the model
      checkpoint = trainer.save()
      print("Checkpoint:", checkpoint)

  if is_save:
    # build the dataframe
    org_df = pd.DataFrame.from_dict(ret)

    # save the data
    org_df.to_csv(f"{path}/{csv_filename}.csv")
    # save the checkpoint (inspired by the Big Data saving pickles to cloud buckets)
    subprocess.run(['cp', checkpoint, f"{path}/{checkpoint_folder}/"], stderr=subproce
    # save the metadata
    subprocess.run(['cp', checkpoint + '.tune_metadata', f"{path}/{checkpoint_folder}/
  else:
    return max(ret['avg_rewards'])

def graph_results(result_csv: str, path="/content/gdrive/My Drive/Colab Notebooks/DRL/
  """Retrieve the csv file and plot the results"""
  # build the dataframe
  df = pd.read_csv(f"{path}/{result_csv}.csv")
  # get rid of the useless column
  df.drop("Unnamed: 0", axis=1, inplace=True)

  # Build a graph of the mean rewards over time
  plt.scatter(df.index, df['avg_rewards'])
  plt.xlabel("Episode")
  plt.ylabel("Avg. reward")
  plt.title("Average rewards over time")
  plt.show()

  # Build a graph of mean Q
  plt.scatter(df.index, df['mean_q'])
  plt.title('Mean Q value over time')
  plt.xlabel('Episode')
  plt.ylabel('Mean Q value over time')
  plt.show()

def get_video(config: dict, path_to_agent: str, file_name:str) -> animation.FuncAnimat
    """Here we create a video from the relevant agent and save it"""
    # visualise what the agent is doing
    # using code from https://docs.ray.io/en/latest/rllib/rllib-training.html (computi
    # setup environment
    env = AdvancedMapEnv(config)
    # setup the agent and restore from the checkpoint
    agent = dqn.DQNTrainer(config=config)
    agent.restore(path_to_agent)
```

```
    # setup the variables to store the reward, whether it is done and the observations
    episode_reward = 0
    done = False
    obs = env.reset()
    # store the data for the video
    scene = []
    total_reward = []
    directions = []

    while not done:
        # work out the next action to take
        action = agent.compute_action(obs)
        # take the next action
        obs, reward, done, _ = env.step(action)
        # add the reward to the episode reward
        episode_reward += reward
        total_reward.append(episode_reward)
        # get the unnormalised data to show the view
        scene.append(env.advanced_map.agents_view())
        # get the relative direction to the next objective
        directions.append(env.advanced_map.direction_to_objective())

    # using code from Computer vision lab 5, animation
    fig, ax = plt.subplots()

    def frame(i: int):
        """Render a frame of the video"""
        # dispose of the last frame
        ax.clear()
        # get rid of the axis labels
        ax.axis('off')
        #fig.tight_layout()
        # render the agent's view
        plot = ax.imshow(scene[i])
        # get the commentary
        ax.set_title(f"Reward: {total_reward[i]}, direction to objective: {directions[
        return plot
    # build up the video
    anim = animation.FuncAnimation(fig, frame, frames=len(scene))

    # set the path to the ffmpeg program (install it using apt-get install ffmpeg, the
    plt.rcParams['animation.ffmpeg_path'] = '/usr/bin/ffmpeg'

    writervideo = animation.FFMpegWriter(fps=5) # from https://www.geeksforgeeks.org/h
    anim.save(f"videos/{file_name}.mp4", writer=writervideo)

    # dispose of the graph object
    plt.close()
    return anim
```

Grid search

In [ ]:

```
def train_from_search_params(params: tuple) -> float:
  """Here we create a config dictionary and pass it into the training loop"""
  cnf = dqn.DEFAULT_CONFIG.copy()
  cnf['framework'] = 'torch'
  # set the environment
  cnf['env'] = AdvancedMapEnv
```

```
  # disable duelling
  cnf['dueling'] = False
  # disable double Q
  cnf['double_q'] = False
  # use relu (REctified Linear Units) activation
  cnf['model']['post_fcnet_activation'] = 'relu'
  # get the hidden units from the parameters
  cnf['model']['fcnet_hiddens'] = params[0]

  # get gamma from the parameters
  cnf["gamma"] = params[1]

  # Disable environment checking so that it doesn't take forever
  cnf['disable_env_checking'] = True

  return training_loop(cnf, "null", "null", num_epochs=100, is_save=False)

def grid_search() -> tuple:
  """as the ray grid search doesn't want to work with modules, I am creating my own"""
  # define the parameters to loop through
  params = [(hiddens, gamma) for hiddens in [[128, 128], [64, 64], [256, 256], [128, 6

  # store the results
  ret = {
      'param': [],
      'max_reward': []
  }

  # map through each iteration using multiprocessing pool
  with Pool() as p:
    output = list(p.map(train_from_search_params, params))
  ret['param'] = params
  ret['max_reward'] = output
  return pd.DataFrame.from_dict(ret)
# run the grid search to see what the best parameters are
grid_search()
```

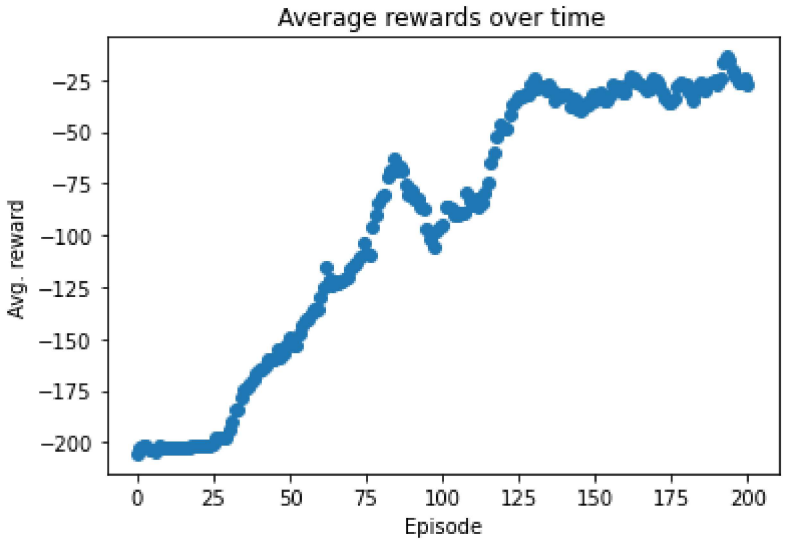
No improvements

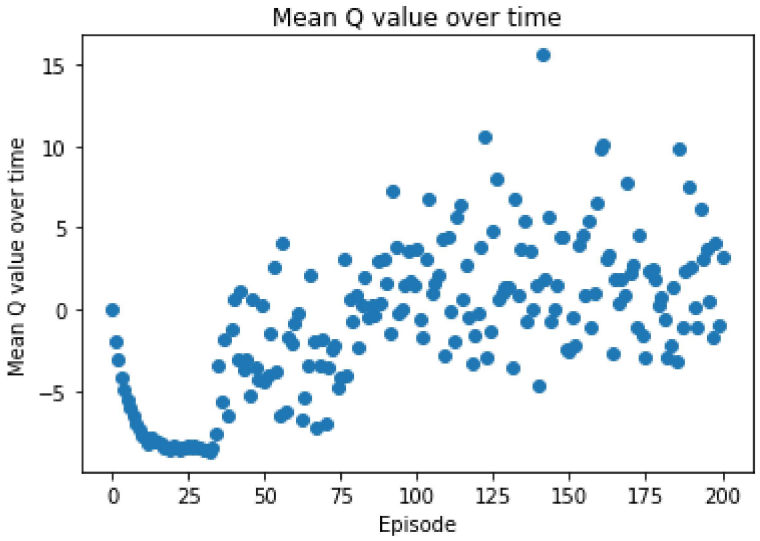
In [5]:
```
# get the default config to use
config = dqn.DEFAULT_CONFIG.copy()
# use torch
config['framework'] = 'torch'
# set the environment
config['env'] = AdvancedMapEnv
# disable duelling
config['dueling'] = False
# disable double Q
config['double_q'] = False
# set the hidden units to 64 then 64
#config['hiddens'] = [64, 64]
# use relu (REctified Linear Units) activation
config['model']['post_fcnet_activation'] = 'relu'
config['model']['fcnet_hiddens'] = [256, 256]
# set the gamma to 0.9, best parameter from the grid search
config['gamma'] = 0.9
```

In [ ]:
```
# run the training loop
```

```
training_loop(config, 'DQNNoImprovement', 'NoImprovement', num_epochs=201, path="data"
```

In [7]:
```
graph_results('DQNNoImprovement', path='data')
```

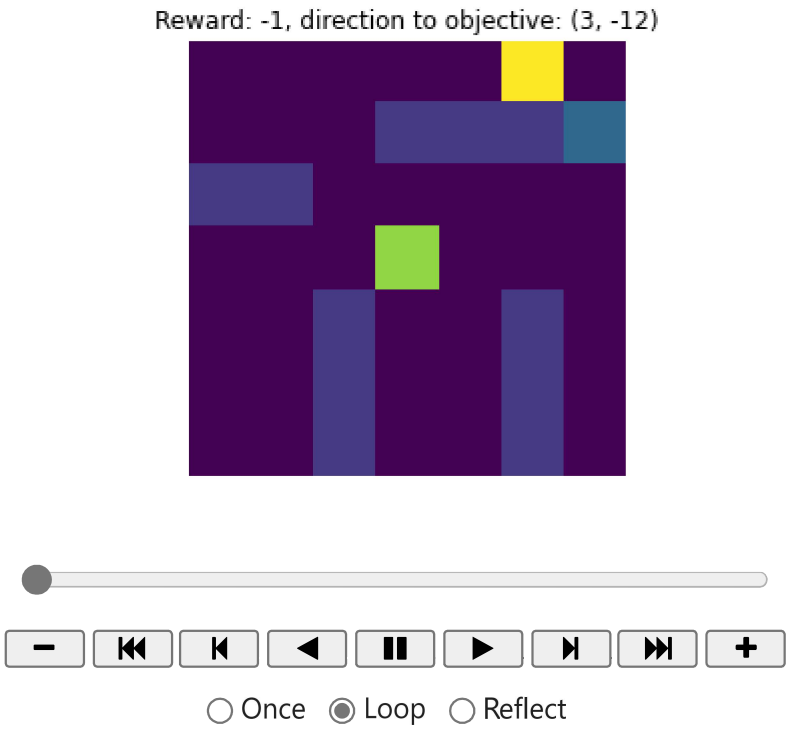




In [8]:
```
# Get a video of what the agent is doing
get_video(config, "data/NoImprovement/checkpoint-201", "NoImprovement")
```

```
2022-04-23 18:02:12,255 WARNING rollout_worker.py:498 -- We've added a module for che
cking environments that are used in experiments. It will cause your environment to fa
il if your environment is not set upcorrectly. You can disable check env by setting `
disable_env_checking` to True in your experiment config dictionary. You can run the e
nvironment checking module standalone by calling ray.rllib.utils.check_env(env).
2022-04-23 18:02:12,256 WARNING env.py:120 -- Your env doesn't have a .spec.max_episo
de_steps attribute. This is fine if you have set 'horizon' in your config dictionary,
or `soft_horizon`. However, if you haven't, 'horizon' will default to infinity, and y
our environment will not be reset.
Install gputil for GPU system monitoring.
2022-04-23 18:02:14,890 WARNING services.py:1983 -- WARNING: The object store is usin
g /tmp instead of /dev/shm because /dev/shm has only 67084288 bytes available. This w
ill harm performance! You may be able to free up space by deleting files in /dev/shm.
If you are inside a Docker container, you can increase /dev/shm size by passing '--sh
m-size=3.31gb' to 'docker run' (or add it to the run_options list in a Ray cluster co
nfig). Make sure to set this to more than 30% of available RAM.
2022-04-23 18:02:18,715 INFO trainable.py:534 -- Restored on 172.17.0.2 from checkpoi
nt: data/NoImprovement/checkpoint-201
2022-04-23 18:02:18,716 INFO trainable.py:543 -- Current state after restoring: {'_it
eration': 201, '_timesteps_total': 6432, '_time_total': 839.1152319908142, '_episodes
_total': 1686}
2022-04-23 18:02:18,766 WARNING deprecation.py:46 -- DeprecationWarning: `compute_act
ion` has been deprecated. Use `Trainer.compute_single_action()` instead. This will ra
ise an error in the future!
```

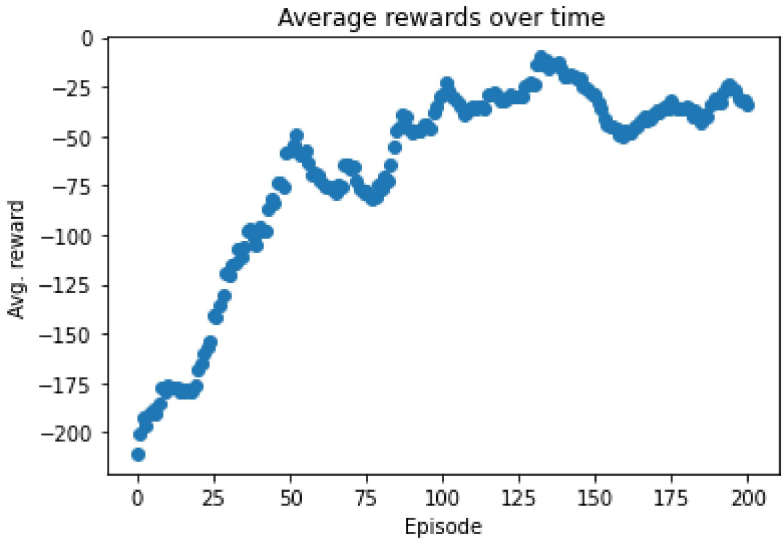
Out[8]:

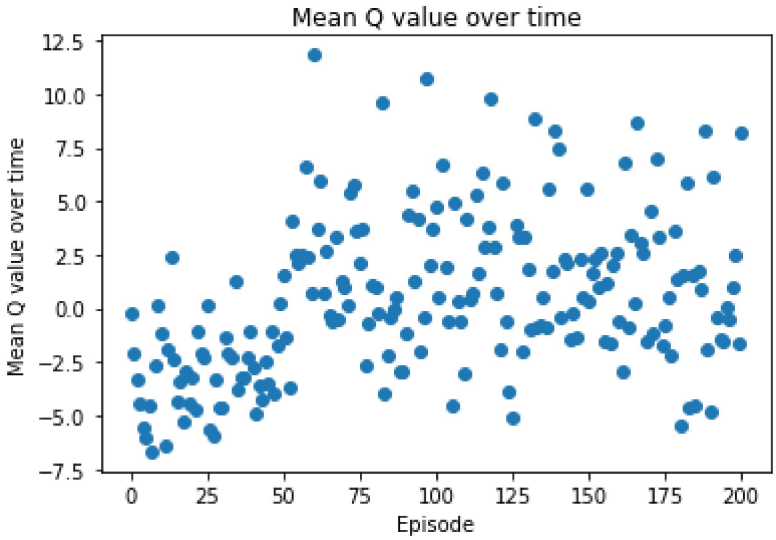


Double Q

In [ ]:
```
# setup the config, get the last config and set it to use double Q
dq_config = config.copy()
dq_config['double_q'] = True

# run the training loop
training_loop(dq_config, 'DQNDoubleQ', 'DoubleQ', num_epochs=201, path='data')
```

In [10]:
```
graph_results('DQNDoubleQ', path='data')
```

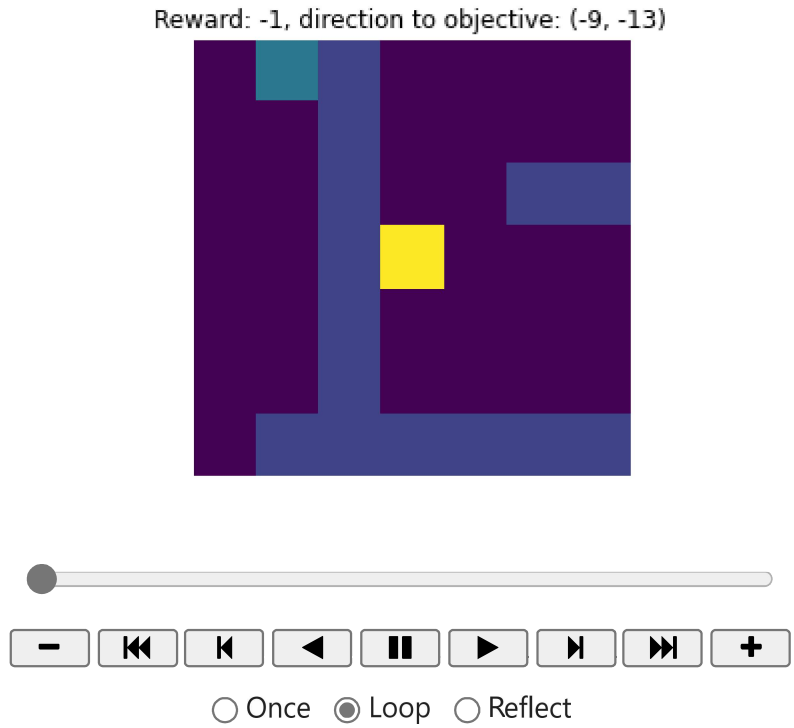




In [11]:
```
get_video(dq_config, "data/DoubleQ/checkpoint-201", "DoubleQ")
```

```
2022-04-23 18:32:07,391 WARNING rollout_worker.py:498 -- We've added a module for che
cking environments that are used in experiments. It will cause your environment to fa
il if your environment is not set upcorrectly. You can disable check env by setting `
disable_env_checking` to True in your experiment config dictionary. You can run the e
nvironment checking module standalone by calling ray.rllib.utils.check_env(env).
2022-04-23 18:32:07,392 WARNING env.py:120 -- Your env doesn't have a .spec.max_episo
de_steps attribute. This is fine if you have set 'horizon' in your config dictionary,
or `soft_horizon`. However, if you haven't, 'horizon' will default to infinity, and y
our environment will not be reset.
2022-04-23 18:32:07,438 WARNING util.py:60 -- Install gputil for GPU system monitorin
g.
2022-04-23 18:32:07,475 INFO trainable.py:534 -- Restored on 172.17.0.2 from checkpoi
nt: data/DoubleQ/checkpoint-201
2022-04-23 18:32:07,477 INFO trainable.py:543 -- Current state after restoring: {'_it
eration': 201, '_timesteps_total': 6432, '_time_total': 897.5175380706787, '_episodes
_total': 1798}
```

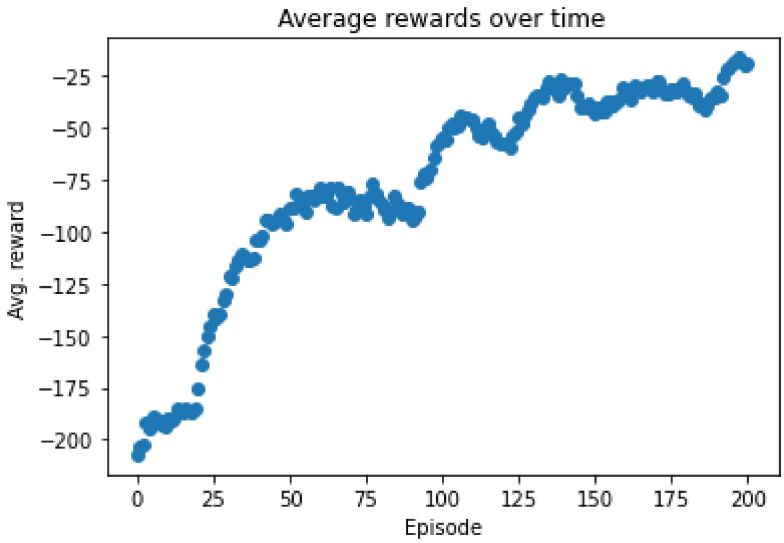
Out[11]:

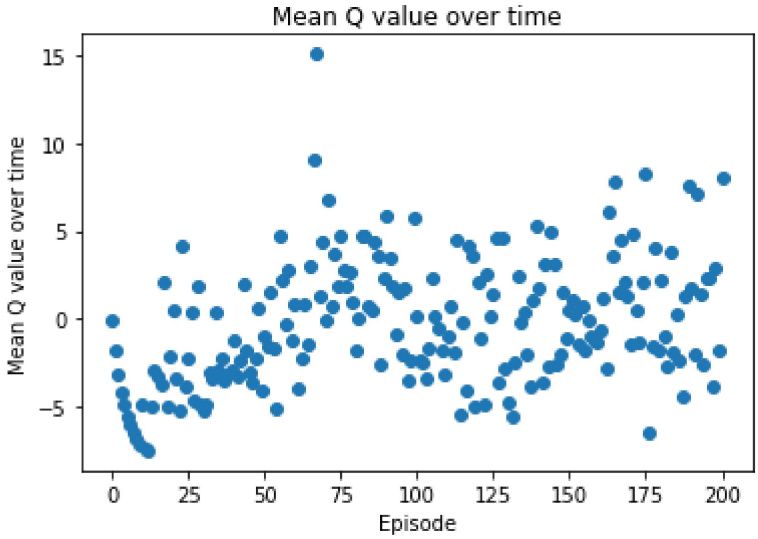


Prioritised Experience Replay

In [ ]:
```
# setup the config, get the last config and set it to use Prioritised Experience Replo
pr_config = config.copy()
# turn on prioritised replay
pr_config['prioritized_replay'] = True
# use the default parameters for prioritised replay
pr_config['prioritized_replay_alpha'] = 0.6
pr_config['prioritized_replay_beta'] = 0.4
pr_config['final_prioritized_replay_beta'] = 0.4
pr_config['prioritized_replay_beta_annealing_timesteps'] = 20000
pr_config['prioritized_replay_eps'] = 1e-6
pr_config['before_learn_on_batch'] = None

# run the training loop
training_loop(pr_config, 'PrioritisedReplay', 'PrioritisedReplay', num_epochs=201, pat
```

In [13]:
```
# show the graphs of the results
graph_results('PrioritisedReplay', path='data')
```

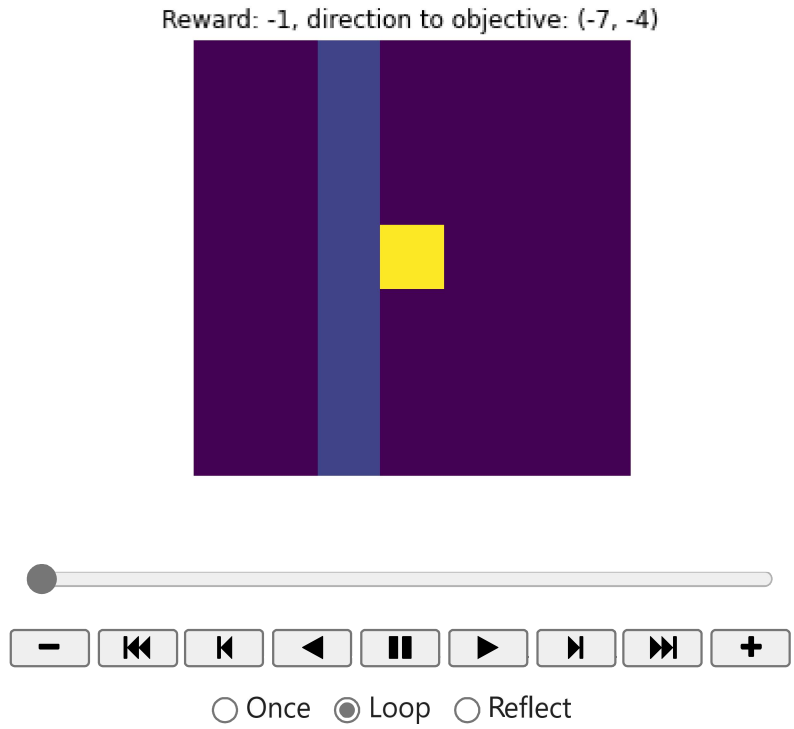




In [14]:
```
# Get the video
get_video(pr_config, "data/PrioritisedReplay/checkpoint-201", "PrioritisedReplay")
```

```
2022-04-23 18:49:04,511 WARNING rollout_worker.py:498 -- We've added a module for che
cking environments that are used in experiments. It will cause your environment to fa
il if your environment is not set upcorrectly. You can disable check env by setting `
disable_env_checking` to True in your experiment config dictionary. You can run the e
nvironment checking module standalone by calling ray.rllib.utils.check_env(env).
2022-04-23 18:49:04,513 WARNING env.py:120 -- Your env doesn't have a .spec.max_episo
de_steps attribute. This is fine if you have set 'horizon' in your config dictionary,
or `soft_horizon`. However, if you haven't, 'horizon' will default to infinity, and y
our environment will not be reset.
2022-04-23 18:49:04,558 WARNING util.py:60 -- Install gputil for GPU system monitorin
g.
2022-04-23 18:49:04,596 INFO trainable.py:534 -- Restored on 172.17.0.2 from checkpoi
nt: data/PrioritisedReplay/checkpoint-201
2022-04-23 18:49:04,597 INFO trainable.py:543 -- Current state after restoring: {'_it
eration': 201, '_timesteps_total': 6432, '_time_total': 895.7082912921906, '_episodes
_total': 1718}
```

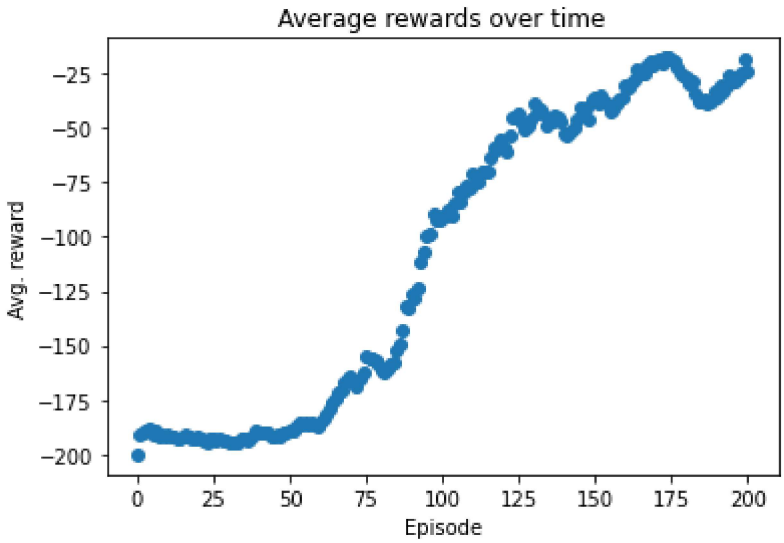
Out[14]:

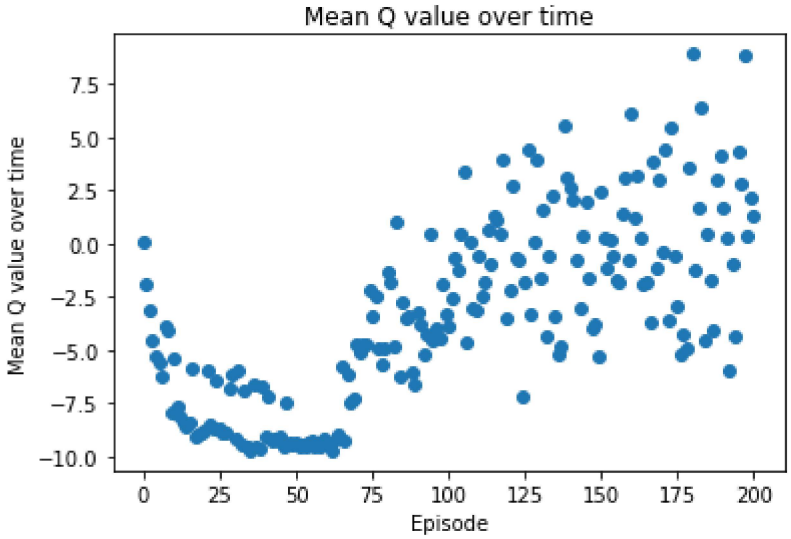


Double Q and Prioritised Replay

In [ ]:
```
# setup the config, get the last config and set it to use dueling
bth_config = config.copy()
bth_config['double_q'] = True
# turn on prioritised replay
bth_config['prioritized_replay'] = True
# use the default parameters for prioritised replay
bth_config['prioritized_replay_alpha'] = 0.6
bth_config['prioritized_replay_beta'] = 0.4
bth_config['final_prioritized_replay_beta'] = 0.4
bth_config['prioritized_replay_beta_annealing_timesteps'] = 20000
bth_config['prioritized_replay_eps'] = 1e-6
bth_config['before_learn_on_batch'] = None

# run the training loop
training_loop(bth_config, 'Both', 'Both', num_epochs=201, path='data')
```

In [16]:
```
# show the graphs of the results
graph_results('Both', path='data')
```

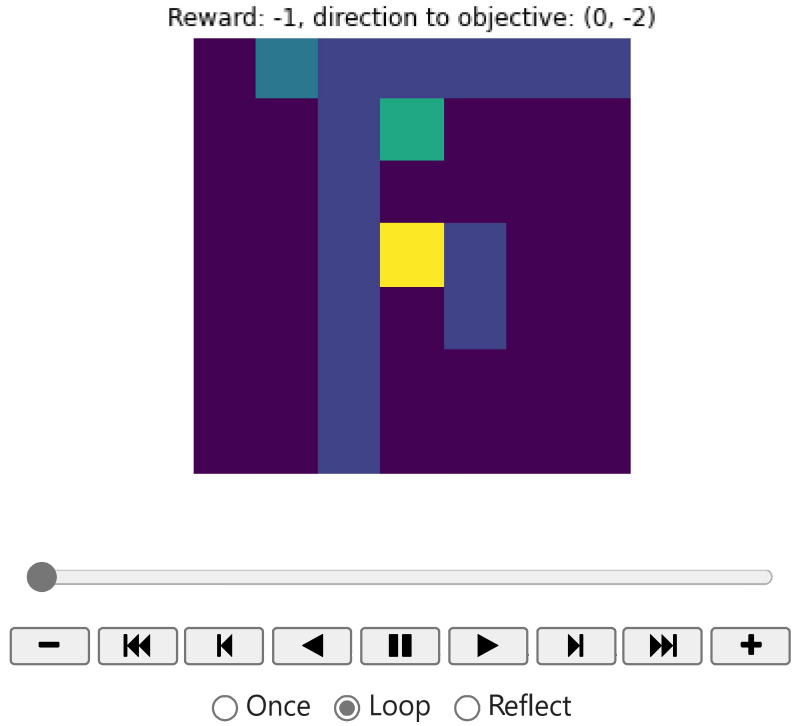




In [17]:
```
# get the video
get_video(bth_config, "data/Both/checkpoint-201","Both")
```

```
2022-04-23 19:04:52,981 WARNING rollout_worker.py:498 -- We've added a module for che
cking environments that are used in experiments. It will cause your environment to fa
il if your environment is not set upcorrectly. You can disable check env by setting `
disable_env_checking` to True in your experiment config dictionary. You can run the e
nvironment checking module standalone by calling ray.rllib.utils.check_env(env).
2022-04-23 19:04:52,981 WARNING env.py:120 -- Your env doesn't have a .spec.max_episo
de_steps attribute. This is fine if you have set 'horizon' in your config dictionary,
or `soft_horizon`. However, if you haven't, 'horizon' will default to infinity, and y
our environment will not be reset.
2022-04-23 19:04:53,027 WARNING util.py:60 -- Install gputil for GPU system monitorin
g.
2022-04-23 19:04:53,068 INFO trainable.py:534 -- Restored on 172.17.0.2 from checkpoi
nt: data/Both/checkpoint-201
2022-04-23 19:04:53,069 INFO trainable.py:543 -- Current state after restoring: {'_it
eration': 201, '_timesteps_total': 6432, '_time_total': 901.633638381958, '_episodes_
total': 1547}
```

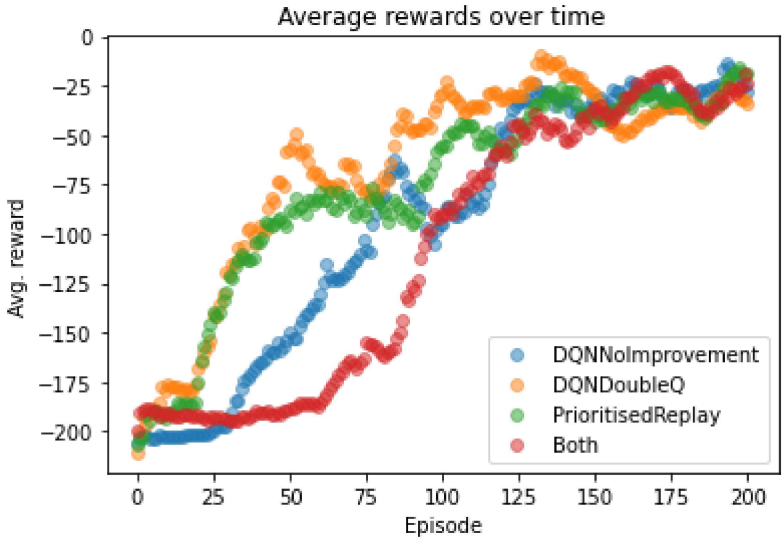
Out[17]:



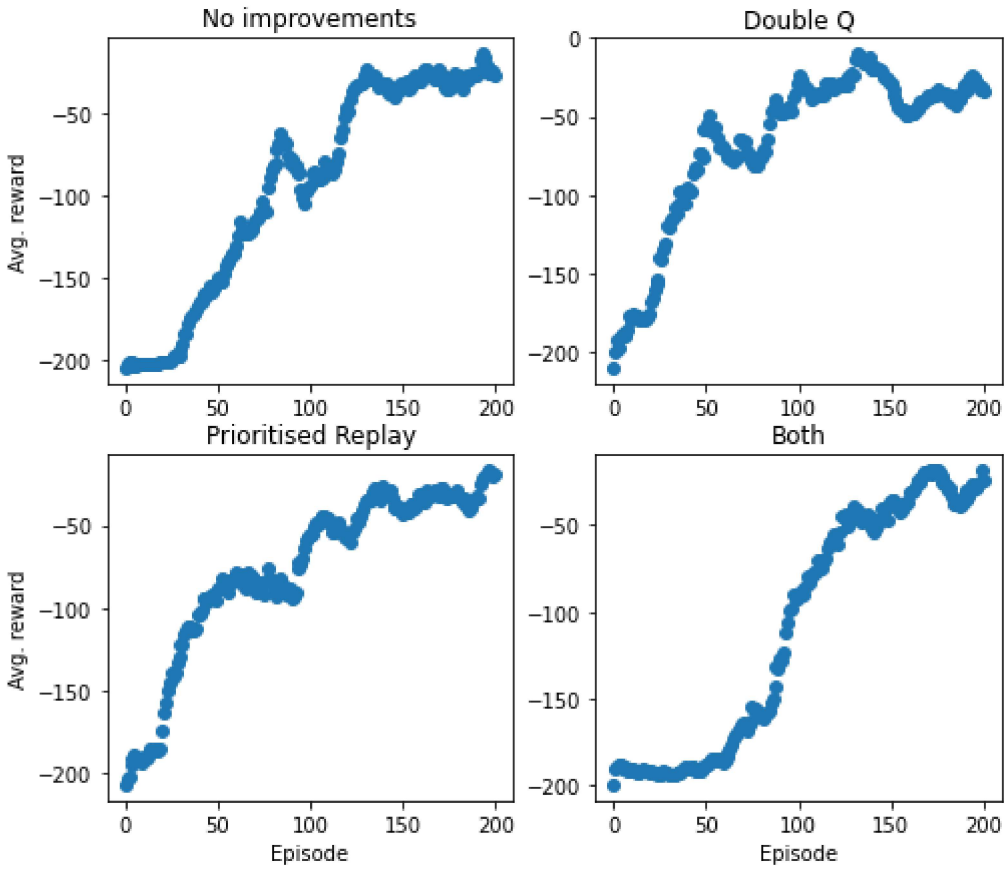
Combined Results

In [18]:

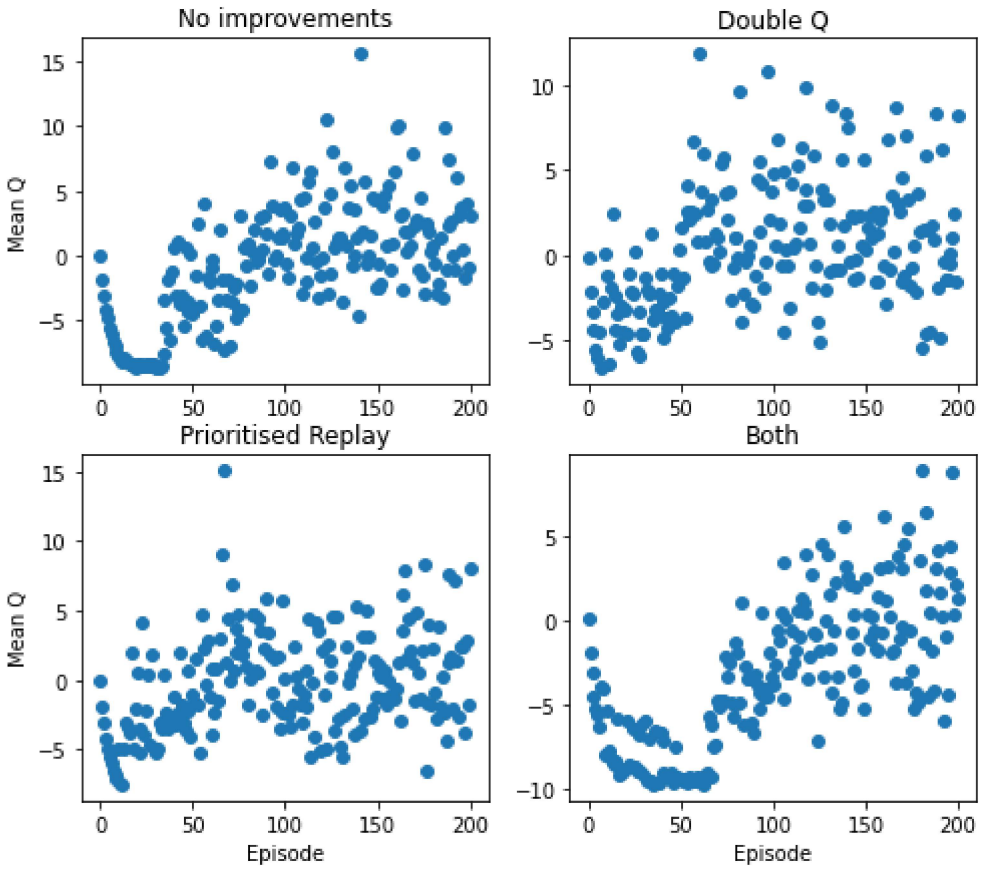
```
# go into the files
for f in ['DQNNoImprovement', 'DQNDoubleQ', 'PrioritisedReplay', 'Both']:
  # build the dataframe
  source = 'data' # /content/gdrive/My Drive/Colab Notebooks/DRL/DRL Coursework/data
  df = pd.read_csv(f"{source}/{f}.csv")
  # get rid of the useless column
  df.drop("Unnamed: 0", axis=1, inplace=True)

  # Build a graph of the mean rewards over time
  plt.scatter(df.index, df['avg_rewards'], label=f, alpha=0.5)
plt.xlabel("Episode")
plt.ylabel("Avg. reward")
plt.title("Average rewards over time")
plt.legend()
plt.show()
```



In [19]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(8,7))

# DQNNoImprovement
ax[0][0].scatter("index", "avg_rewards", data=pd.read_csv('data/DQNNoImprovement.csv')
ax[0][0].set_title("No improvements")
ax[0][0].set_ylabel("Avg. reward")

# Double Q
ax[0][1].scatter("index", "avg_rewards", data=pd.read_csv('data/DQNDoubleQ.csv').drop(
ax[0][1].set_title("Double Q")

# Prioritised Replay
ax[1][0].scatter("index", "avg_rewards", data=pd.read_csv('data/PrioritisedReplay.csv'
ax[1][0].set_title("Prioritised Replay")
ax[1][0].set_ylabel("Avg. reward")
ax[1][0].set_xlabel("Episode")

# Both
ax[1][1].scatter("index", "avg_rewards", data=pd.read_csv('data/Both.csv').drop("Unnam
ax[1][1].set_title("Both")
ax[1][1].set_xlabel("Episode")

plt.show()
```



In [20]:
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(8,7))

# DQNNoImprovement
ax[0][0].scatter("index", "mean_q", data=pd.read_csv('data/DQNNoImprovement.csv').drop
ax[0][0].set_title("No improvements")
ax[0][0].set_ylabel("Mean Q")

# Double Q
ax[0][1].scatter("index", "mean_q", data=pd.read_csv('data/DQNDoubleQ.csv').drop("Unna
ax[0][1].set_title("Double Q")

# Prioritised Replay
ax[1][0].scatter("index", "mean_q", data=pd.read_csv('data/PrioritisedReplay.csv').dro
ax[1][0].set_title("Prioritised Replay")
ax[1][0].set_ylabel("Mean Q")
ax[1][0].set_xlabel("Episode")

# Both
ax[1][1].scatter("index", "mean_q", data=pd.read_csv('data/Both.csv').drop("Unnamed: 0
ax[1][1].set_title("Both")
ax[1][1].set_xlabel("Episode")

plt.show()
```


No improvements
Double Q
Prioritised Replay
Both
Mean Q
Episode

DQN with two improvements

In [1]:
```
# From Computer vision lab 6, keeps reloading the code for the objects so that you car
%load_ext autoreload
%autoreload 2
```

In [ ]:
```
SAVE_PATH=''
```

In [2]:
```
# for colab
from google.colab import drive
drive.mount('/content/drive')
path = '/content/drive/My Drive/Colab Notebooks/DRL/DRL Coursework/'

import sys
sys.path.append(path)

SAVE_PATH=path
```

```
Mounted at /content/drive
```

In [3]:
```
# imports
from helpers.get_available_actions import get_available_actions, adv_action_from_index
import numpy as np
from helpers.advanced_map import AdvancedMap
import matplotlib.pyplot as plt
%matplotlib inline
import random
import torch
import math
import pickle
from collections import namedtuple, deque

# determine if we are using cpu or gpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [4]:
```
# Build the Deep Q Network
class DQN(torch.nn.Module):
    """Deep Q Network, sizes based upon the grid search in RLLib"""

    def __init__(self):
        """Create an instance of the network"""
        # Call the base objects init method
        super().__init__()

        # our advanced network will be sending in a 53 column list of numbers
        self.dense1 = torch.nn.Linear(53, 256)
        # normalise the data in the network
        self.norm1 = torch.nn.BatchNorm1d(256)

        self.dense2 = torch.nn.Linear(256, 256)
        self.norm2 = torch.nn.BatchNorm1d(256)

        self.dense3 = torch.nn.Linear(256, 256)
        self.norm3 = torch.nn.BatchNorm1d(256)
```

```
        # 5 potential actions
        self.dense4 = torch.nn.Linear(256, 5)

    def forward(self, x):
        """Pass the data through the network"""
        # convert the input data to a float, run it through the first dense layer, the
        output = torch.functional.F.relu(self.norm1(self.dense1(x.float())))
        # repeat for
        output = torch.functional.F.relu(self.norm2(self.dense2(output)))

        output = torch.functional.F.relu(self.norm3(self.dense3(output)))
        # reshape the tensor and pass it through the last stage
        return self.dense4(output.view(output.size(0), -1))

#DQN()
```

In [5]:

```
Transition = namedtuple('Transition', ('state', 'action', 'next_state', 'reward')) # j

class ReplayMemory(object):
    """from https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html"
    def __init__(self, capacity):
        # uses a deque structure so that as new items are added to the end of the 'lis
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        """Return a random sample from the memory"""
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)

    def reset(self):
        """Clear the memories"""
        # reset the memory
        self.memory = deque([], maxlen=self.memory.maxlen)
```

In [83]:

```
# Test that it works
# Get an instance of the map
amap = AdvancedMap()
# Get the normalised observations
obs, r, done = amap.convert_observations(amap.reset())
# get an instance of the model
mdl = DQN()
# put it in evaluation mode
mdl.eval()
# convert the observations to a torch tensor and reshape the array to what the model e
out = mdl(torch.from_numpy(obs).unsqueeze(0))
# use argmax to get the index of the action to take and then use item to get it as a p
torch.argmax(out).item()
```

Out[83]:
```
{'agent_health': 100, 'agent_view': array([[0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 1., 0., 0.],
       [1., 1., 1., 1., 1., 1., 1.],
       [0., 0., 0., 5., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0., 1., 1.]]), 'enemy_count': 3, 'immediate_reward': -1, 'is
_stop': False, 'obj_direction': (3,
  11)}
```

In [74]:
```
# setup the networks and send them to the device
Q_net = DQN().to(device)
Q_target = DQN().to(device)
# load the state dictionary of weights from the q network to the target
Q_target.load_state_dict(Q_net.state_dict())
# set the target to evaluation mode
Q_target.eval()

# setup the optimiser
optim = torch.optim.SGD(Q_net.parameters(), lr=0.01)

# setup the memory
memory = ReplayMemory(10000)

# setup hyper-parameters
GAMMA = 0.9 # from RLLib grid search

# from https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200

# from lab 6
BATCH_SIZE = 256
TARGET_UPDATE = 20
```

In [75]:
```
def select_action(state, steps_done) -> torch.tensor:
    """Inspired by https://pytorch.org/tutorials/intermediate/reinforcement_q_learning
    Takes in the current state and the number of steps and returns an action either fr
    # work out the current threshold
    thresh = EPS_END + (EPS_START - EPS_END) * math.exp(-steps_done / EPS_DECAY)

    # if the random number is bigger than the threshold then work out an action from t
    if random.random() > thresh:
        # don't work out the gradient for this
        with torch.no_grad():
            # get the max column value, then gets its index and reshape into the expec
            return Q_net(state).max(1)[1].view(1, 1)
    # as there is a return in the if statement we don't need an else. return a random
    return torch.tensor([[random.randrange(5)]], device=device, dtype=torch.long)
```

In [76]:
```
def optimise_model(doubleQ=False):
    """Optimise the model. Inspired by https://pytorch.org/tutorials/intermediate/reir
    if len(memory) < BATCH_SIZE:
        # stop optimising the model if the memory is less than the batch size
        return
    # get a sample of the memories
    transitions = memory.sample(BATCH_SIZE)
    # convert the batch of transitions to a transition of batches
```

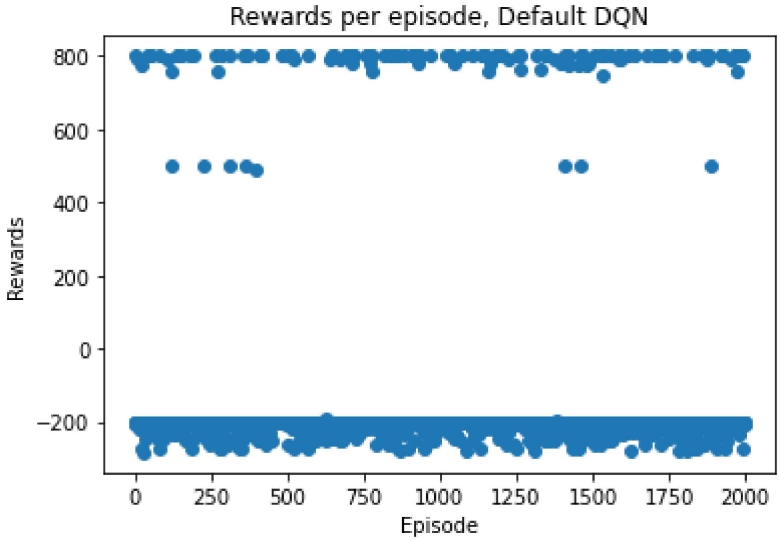
```
    batch = Transition(*zip(*transitions))


    # compute a mask of non-final states
    non_final_mask = torch.tensor(tuple(map(lambda state: state is not None, batch.nex

    # get the next states
    non_final_next_states = torch.cat([torch.tensor([s], device=device) for s in batch

    state_batch = torch.cat([torch.tensor([s], device=device) for s in batch.state])
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)
    action_batch = action_batch.reshape((BATCH_SIZE, 1))

    # work out the Q values
    q_vals = Q_net(state_batch).gather(1, action_batch)

    # compute the next state and compare with the target
    next_q_values = torch.zeros(BATCH_SIZE, device=device)
    if doubleQ:
        # YDoubleDQNt ≡ Rt+1+γQ(St+1, argmax Q(St+1, a; ϑt), ϑq) # equation from https
        #next_q_values[non_final_mask] = Q_net(Q_target(non_final_next_states).argmax(

        # I looked at https://colab.research.google.com/github/ehennis/ReinforcementLe

        # get the Q value from the main model
        mdl = Q_net(non_final_next_states)
        # get the Q value from the target model
        target = Q_target(non_final_next_states)
        # get the argmax columns from the main model
        argmx = np.argmax(mdl.detach().numpy(), axis=1)

        # loop through the targets and select the argmax col. need to index with both
        # a = torch.tensor([[1, 2, 3], [4, 5, 6]])
        # [a[idx, col].item() for idx, col in enumerate([0, 1])]
        targets = [target[idx, col].item() for idx, col in enumerate(argmx.tolist())]
        next_q_values[non_final_mask] = torch.tensor(targets, device=device)

    else:
        # YDQNt ≡ Rt+1 + γ max Q(St+1, a; ϑt). # equation from https://arxiv.org/pdf/1
        next_q_values[non_final_mask] = Q_target(non_final_next_states).max(1)[0].deta

    # get the expected Q values
    expected_q_values = (next_q_values * GAMMA) + reward_batch # equation 2
    expected_q_values = expected_q_values.unsqueeze(1)

    # work out the loss
    loss = torch.functional.F.mse_loss(q_vals, expected_q_values)

    # optimise the model
    optim.zero_grad()
    loss.backward()

    # clip the gradients
    for param in Q_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optim.step()
    return loss
```

In [77]:
```
def fill_memory(Q_net, memory):
  # fill up the memory with random actions
  Q_net.eval()
  memory.reset()
  while not memory.memory.maxlen == len(memory):
      # Get an instance of the map
      amap = AdvancedMap()
      # Get the normalised observations
      obs, reward, done = amap.convert_observations(amap.reset())

      # setup the total reward
      total_reward = 0

      steps_done = 0

      # run until it is done
      while not done:
          # put the state into a tensor
          state = torch.from_numpy(obs).unsqueeze(0)

          # pick an action and increment the number of steps
          action, steps_done = select_action(state, steps_done), steps_done + 1

          # get the action
          action = adv_action_from_index(action.item())

          # take the action
          next_state, reward, done = amap.convert_observations(amap.step(action))

          # add to the reward
          total_reward += reward

          # set next state to nothing if we're done
          if done:
              next_state = None

          # add to the memories
          memory.push(obs, torch.tensor([action.index], device=device), next_state, to

          # pass the next_state through to the next round
          obs = next_state
  print('Finished the warmup')
  return memory
```

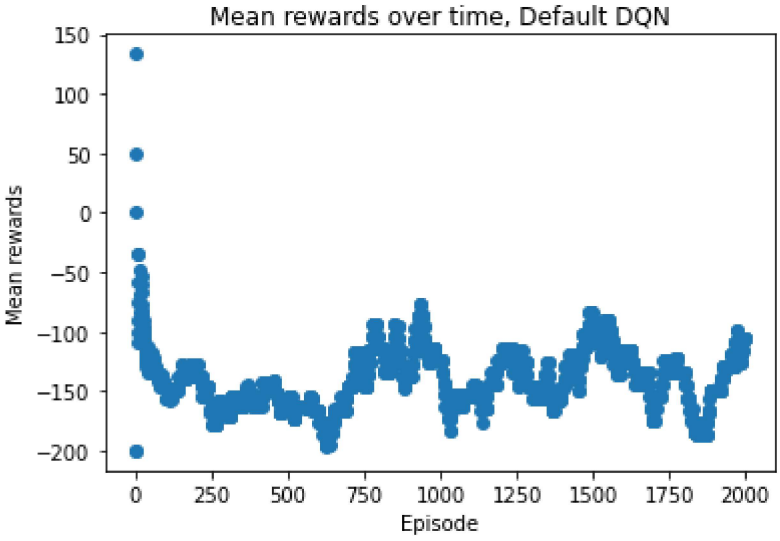
In [ ]:
```
# start training the network
# setup the networks and send them to the device
Q_net = DQN().to(device)
Q_target = DQN().to(device)
# load the state dictionary of weights from the q network to the target
Q_target.load_state_dict(Q_net.state_dict())
# set the target to evaluation mode
Q_target.eval()

memory = fill_memory(Q_net, memory)

rewards = []
mean_rewards = []
for episode in range(2000):
    # Get an instance of the map
```

```
    amap = AdvancedMap()
    # Get the normalised observations
    obs, reward, done = amap.convert_observations(amap.reset())

    # setup the total reward
    total_reward = 0

    steps_done = 0

    while not done:
        Q_net.eval()
        # put the state into a tensor
        state = torch.from_numpy(obs).unsqueeze(0)

        # pick an action and increment the number of steps
        action, steps_done = select_action(state, steps_done), steps_done + 1

        # get the action
        action = adv_action_from_index(action.item())

        # take the action
        next_state, reward, done = amap.convert_observations(amap.step(action))

        # add to the reward
        total_reward += reward

        # set next state to nothing if we're done
        if done:
            next_state = None

        # add to the memories
        memory.push(obs, torch.tensor([action.index], device=device), next_state, torc

        # pass the next_state through to the next round
        obs = next_state

    # train the model
    Q_net.train()
    loss = optimise_model()

    # add the rewards history
    rewards.append(total_reward)

    # update the target
    if episode % TARGET_UPDATE == 0:
        Q_target.load_state_dict(Q_net.state_dict())
    # work out the mean rewards
    if len(rewards) < 100:
        mean_reward = np.mean(rewards)
    else:
        mean_reward = np.mean(rewards[-100:])
    mean_rewards.append(mean_reward)
    # print info
    print(f"Episode: {episode}, reward: {total_reward}, loss: {loss.item()}, Mean rewa

    # early stopping
    if mean_reward > 500:
        print("Stopped early")
        break
```

In [97]:
```
# plot the rewards per episode
plt.scatter(list(range(len(rewards))), rewards)
plt.title("Rewards per episode, Default DQN")
plt.xlabel("Episode")
plt.ylabel("Rewards")
plt.show()
```



In [98]:
```
# plot the rewards per episode
plt.scatter(list(range(len(mean_rewards))), mean_rewards)
plt.title("Mean rewards over time, Default DQN")
plt.xlabel("Episode")
plt.ylabel("Mean rewards")
plt.show()
```



In [99]:
```
# Save the data
# get a reference to the file in write bytes mode
with open(SAVE_PATH + 'data/NoImprovementRaw/rewards.pkl', 'wb') as f:
    # serialise the rewards array
    pickle.dump(rewards, f)
with open(SAVE_PATH + 'data/NoImprovementRaw/mean_rewards.pkl', 'wb') as f:
    # serialise the rewards array
    pickle.dump(mean_rewards, f)
# save the state dict (from https://pytorch.org/tutorials/beginner/saving_loading_mode
torch.save(Q_net.state_dict(), SAVE_PATH + 'data/NoImprovementRaw/qnet.pth')
```

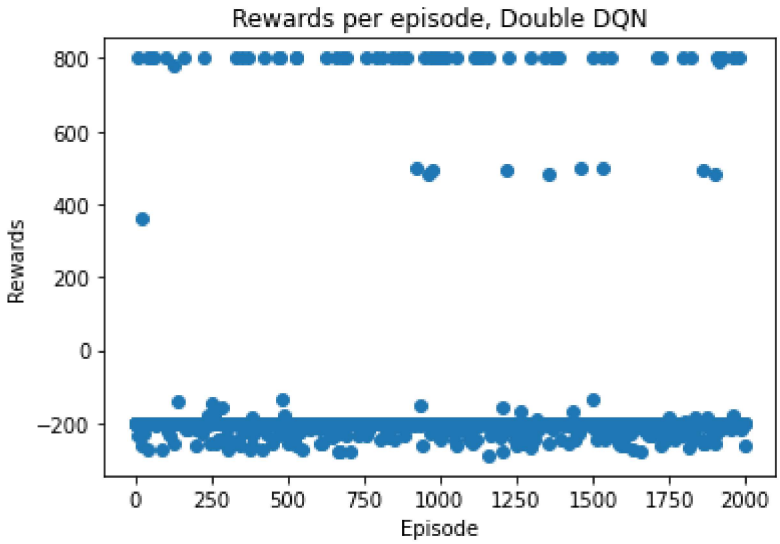
Double Q

In [ ]:
```
# setup the networks and send them to the device
# setup the networks and send them to the device
Q_net = DQN().to(device)
Q_target = DQN().to(device)
# load the state dictionary of weights from the q network to the target
Q_target.load_state_dict(Q_net.state_dict())
# set the target to evaluation mode
Q_target.eval()

memory = fill_memory(Q_net, memory)

# start training the network
rewards = []

mean_rewards = []
for episode in range(2000):
    # Get an instance of the map
    amap = AdvancedMap()
    # Get the normalised observations
    obs, reward, done = amap.convert_observations(amap.reset())

    # setup the total reward
    total_reward = 0

    steps_done = 0

    while not done:
        Q_net.eval()
        # put the state into a tensor
        state = torch.from_numpy(obs).unsqueeze(0)

        # pick an action and increment the number of steps
        action, steps_done = select_action(state, steps_done), steps_done + 1

        # get the action
        action = adv_action_from_index(action.item())

        # take the action
        next_state, reward, done = amap.convert_observations(amap.step(action))

        # add to the reward
        total_reward += reward

        # set next state to nothing if we're done
        if done:
            next_state = None

        # add to the memories
        memory.push(obs, torch.tensor([action.index], device=device), next_state, torc

        # pass the next_state through to the next round
        obs = next_state

    # train the model
    Q_net.train()
```

```
    loss = optimise_model(doubleQ=True)

    # add the rewards history
    rewards.append(total_reward)

    # update the target
    if episode % TARGET_UPDATE == 0:
        Q_target.load_state_dict(Q_net.state_dict())

    # work out the mean rewards
    if len(rewards) < 100:
        mean_reward = np.mean(rewards)
    else:
        mean_reward = np.mean(rewards[-100:])
    mean_rewards.append(mean_reward)
    # print info
    print(f"Episode: {episode}, reward: {total_reward}, loss: {loss.item()}, Mean rewa

    # early stopping
    if mean_reward > 500:
        print("Stopped early")
        break
```

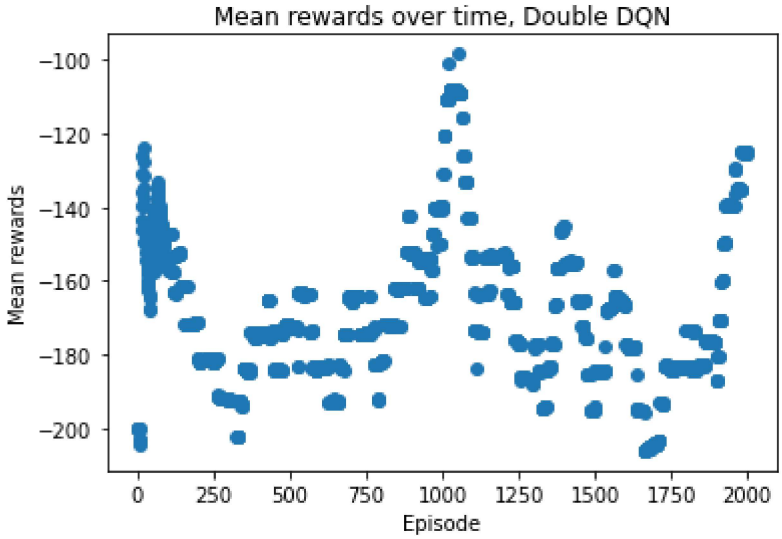
In [101…

```
# plot the rewards per episode
plt.scatter(list(range(len(rewards))), rewards)
plt.title("Rewards per episode, Double DQN")
plt.xlabel("Episode")
plt.ylabel("Rewards")
plt.show()
```



In [102…

```
# plot the rewards per episode
plt.scatter(list(range(len(mean_rewards))), mean_rewards)
plt.title("Mean rewards over time, Double DQN")
plt.xlabel("Episode")
plt.ylabel("Mean rewards")
plt.show()
```



In [104…
```
# Save the data
# get a reference to the file in write bytes mode
with open(SAVE_PATH + 'data/DoubleQRaw/rewards.pkl', 'wb') as f:
    # serialise the rewards array
    pickle.dump(rewards, f)
with open(SAVE_PATH + 'data/DoubleQRaw/mean_rewards.pkl', 'wb') as f:
    # serialise the rewards array
    pickle.dump(mean_rewards, f)
# save the state dict (from https://pytorch.org/tutorials/beginner/saving_loading_mode
torch.save(Q_net.state_dict(), SAVE_PATH + 'data/DoubleQRaw/qnet.pth')
```

Prioritised Replay

In [105…
```
prioritisedTransition = namedtuple('Transition', ('state', 'action', 'next_state', 're

class PriorityReplayMemory(ReplayMemory):
    """enhanced for prioritised replay using algorithm 1 from https://arxiv.org/pdf/15
    def __init__(self, capacity):
        ReplayMemory.__init__(self, capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(prioritisedTransition(*args))

    def sample(self, batch_size):
        """Return a random sample from the memory weighted by their priority"""
        # gets the priority of the transition and uses those in random.choices to weig
        #return random.choices(self.memory, [w.priority for w in self.memory], k=batch

        # return the indexes of the memories to make it easier to update later

        # compute importance sampling weight
        #p_alpha = [p.priority ** ALPHA for p in self.memory]
        # get priority
        #PJ = [p / sum(p_alpha) for p in p_alpha]
        #return random.choices(range(len(self.memory)), [w.priority for w in self.memo
        # get the weight
        #WJ = [(1/p) * (1/len(self.memory)) for p in PJ]
        #WJ = [p ** -1 for p in WJ]
```

```
        #WJ = ((np.array(PJ) * 1/10000) ** BETA).tolist()

        # compute importance sampling weight, use numpy as it is faster
        PJ = np.array([p.priority for p in priority_memory.memory])
        PJ = PJ ** ALPHA
        PJ = PJ / PJ.sum()
        PJ = PJ.tolist()
        return random.choices(range(len(self.memory)), PJ, k=batch_size), PJ
```

In [106…

```
# hyper-params from https://arxiv.org/pdf/1511.05952.pdf using the proportional variar
ALPHA = 0.6
BETA = 0.4
priority_memory = PriorityReplayMemory(10000)
ETA = 0.001
```

In [107…

```
def optimise_model_priority(doubleQ=False):
    """Optimise the model. Inspired by https://pytorch.org/tutorials/intermediate/reir
    if len(priority_memory) < BATCH_SIZE:
        # stop optimising the model if the memory is less than the batch size
        return
    # get a sample of the memories
    #transitions = priority_memory.sample(BATCH_SIZE)
    idxs, PJ_org  = priority_memory.sample(BATCH_SIZE)
    transitions = [priority_memory.memory[x] for x in idxs]
    # convert the batch of transitions to a transition of batches
    batch = prioritisedTransition(*zip(*transitions))

    # compute a mask of non-final states
    non_final_mask = torch.tensor(tuple(map(lambda state: state is not None, batch.nex

    # get the next states
    non_final_next_states = torch.cat([torch.tensor([s], device=device) for s in batch

    state_batch = torch.cat([torch.tensor([s], device=device) for s in batch.state])
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)
    action_batch = action_batch.reshape((BATCH_SIZE, 1))

    # work out the Q values
    q_vals = Q_net(state_batch).gather(1, action_batch)

    # compute the next state and compare with the target
    next_q_values = torch.zeros(BATCH_SIZE, device=device)
    if doubleQ:
        # YDoubleDQNt ≡ Rt+1+γQ(St+1, argmax Q(St+1, a; ϑt), ϑq) # equation from https
        #next_q_values[non_final_mask] = Q_net(Q_target(non_final_next_states).argmax(

        # I looked at https://colab.research.google.com/github/ehennis/ReinforcementLe

        # get the Q value from the main model
        mdl = Q_net(non_final_next_states)
        # get the Q value from the target model
        target = Q_target(non_final_next_states)
        # get the argmax columns from the main model
        argmx = np.argmax(mdl.detach().numpy(), axis=1)

        # loop through the targets and select the argmax col. need to index with both
        # a = torch.tensor([[1, 2, 3], [4, 5, 6]])
```

```
        # [a[idx, col].item() for idx, col in enumerate([0, 1])]
        targets = [target[idx, col].item() for idx, col in enumerate(argmx.tolist())]
        next_q_values[non_final_mask] = torch.tensor(targets, device=device)

    else:
        # YDQNt ≡ Rt+1 + γ max Q(St+1, a; ϑt). # equation from https://arxiv.org/pdf/1
        next_q_values[non_final_mask] = Q_target(non_final_next_states).max(1)[0].deta

    # get the expected Q values
    expected_q_values = (next_q_values * GAMMA) + reward_batch # equation 2
    expected_q_values = expected_q_values.unsqueeze(1)

    # Work out wj
    #WJ = ((1/len(priority_memory)) * (1/np.array(PJ_org))) ** BETA
    # normalise weights
    #WJ = WJ * (1 / WJ.max())
    #WJ = torch.from_numpy(WJ)
    # from algorithm 1
    #WJ = (len(priority_memory) * np.array(PJ_org)) ** -BETA
    WI = ((1 / len(priority_memory)) * (1 / np.array(PJ_org))) ** BETA
    WJ = ((len(priority_memory) * np.array(PJ_org)) ** -BETA) / WI.max()

    # normalise weights
    #WJ = WJ * (1 / WJ.max())
    WJ = torch.from_numpy(WJ)

    # work out the loss
    #loss = torch.functional.F.mse_loss(q_vals, expected_q_values)
    loss = torch.mean((q_vals - expected_q_values) ** 2 * WJ) # from that https://gith

    # optimise the model
    optim.zero_grad()
    loss.backward()

    # δ is the error of the model, how much the network has to learn from the memory (
    new_priority = [abs(x[0][0] - x[1][0]) for x in list(zip(q_vals.tolist(), expected
    #new_priority = np.array(new_priority) ** 2

    # work out WJ
    #WJ = (((1/len(priority_memory)) * np.array(PJ_org)) ** BETA) * (abs(loss.item())
    # normalise the weights
    #WJ = WJ * (1/WJ.max())
    #WJ = WJ.tolist()

    for i in range(len(idxs)):
        # build a new tuple out of the old one and add in the absolute value of the lo
        #priority_memory.memory[idxs[i]] = prioritisedTransition(transitions[i][0], tr
        #priority_memory.memory[idxs[i]] = prioritisedTransition(transitions[i][0], tr
        #priority_memory.memory[idxs[i]] = prioritisedTransition(transitions[i][0], tr
        #priority_memory.memory[idxs[i]] = prioritisedTransition(transitions[i][0], tr
        priority_memory.memory[idxs[i]] = prioritisedTransition(transitions[i][0], tra


    # clip the gradients
    for param in Q_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optim.step()
    return loss
```

In [108…

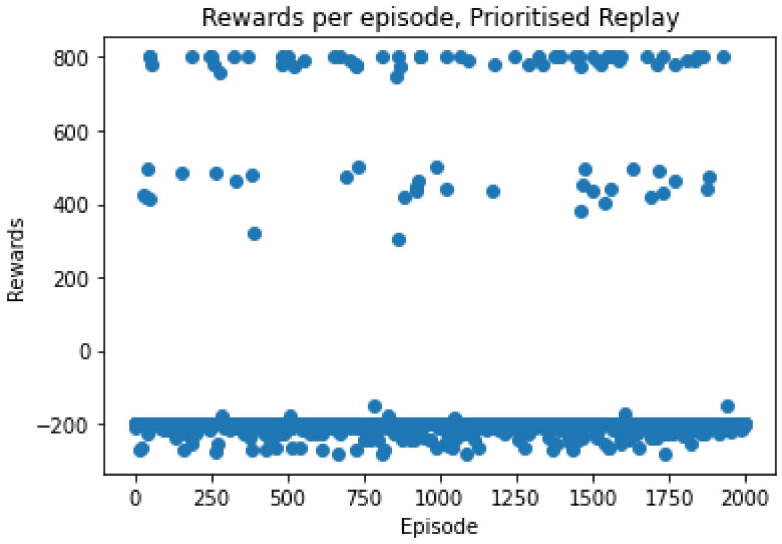
```
def fill_priority_memory(Q_net, priority_memory):
  """fill up the memory with random actions"""
  Q_net.eval()
  priority_memory.reset()
  while not priority_memory.memory.maxlen == len(priority_memory):
      # Get an instance of the map
      amap = AdvancedMap()
      # Get the normalised observations
      obs, reward, done = amap.convert_observations(amap.reset())

      # setup the total reward
      total_reward = 0

      steps_done = 0

      # run until it is done
      while not done:
          # put the state into a tensor
          state = torch.from_numpy(obs).unsqueeze(0)

          # pick an action and increment the number of steps
          action, steps_done = select_action(state, steps_done), steps_done + 1

          # get the action
          action = adv_action_from_index(action.item())

          # take the action
          next_state, reward, done = amap.convert_observations(amap.step(action))

          # add to the reward
          total_reward += reward

          # set next state to nothing if we're done
          if done:
              next_state = None

          # add to the priority memories, use 1 as the default priority
          #priority_memory.push(obs, torch.tensor([action.index], device=device), next
          priority_memory.push(obs, torch.tensor([action.index], device=device), next_

          # pass the next_state through to the next round
          obs = next_state
  print('Finished the warmup')
  return priority_memory
```

In [ ]:

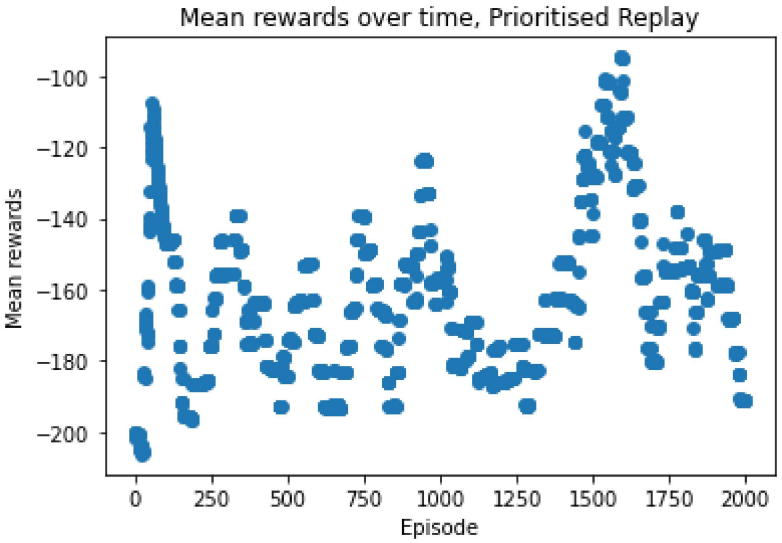
```
# start training the network
# setup the networks and send them to the device
Q_net = DQN().to(device)
Q_target = DQN().to(device)
# load the state dictionary of weights from the q network to the target
Q_target.load_state_dict(Q_net.state_dict())
# set the target to evaluation mode
Q_target.eval()

priority_memory = fill_priority_memory(Q_net, priority_memory)

rewards = []
mean_rewards = []
for episode in range(2000):
```

```
    # Get an instance of the map
    amap = AdvancedMap()
    # Get the normalised observations
    obs, reward, done = amap.convert_observations(amap.reset())

    # setup the total reward
    total_reward = 0

    steps_done = 0

    while not done:
        Q_net.eval()
        # put the state into a tensor
        state = torch.from_numpy(obs).unsqueeze(0)

        # pick an action and increment the number of steps
        action, steps_done = select_action(state, steps_done), steps_done + 1

        # get the action
        action = adv_action_from_index(action.item())

        # take the action
        next_state, reward, done = amap.convert_observations(amap.step(action))

        # add to the reward
        total_reward += reward

        # set next state to nothing if we're done
        if done:
            next_state = None

        # add to the memories
        priority_memory.push(obs, torch.tensor([action.index], device=device), next_st

        # pass the next_state through to the next round
        obs = next_state

    # train the model
    Q_net.train()
    loss = optimise_model_priority()

    # add the rewards history
    rewards.append(total_reward)

    # update the target
    if episode % TARGET_UPDATE == 0:
        Q_target.load_state_dict(Q_net.state_dict())
    # work out the mean rewards
    if len(rewards) < 100:
        mean_reward = np.mean(rewards)
    else:
        mean_reward = np.mean(rewards[-100:])
    mean_rewards.append(mean_reward)
    # print info
    print(f"Episode: {episode}, reward: {total_reward}, loss: {loss.item()}, Mean rewa

    # early stopping
    if mean_reward > 500:
        print("Stopped early")
        break
```

In [110…
```
# plot the rewards per episode
plt.scatter(list(range(len(rewards))), rewards)
plt.title("Rewards per episode, Prioritised Replay")
plt.xlabel("Episode")
plt.ylabel("Rewards")
plt.show()
```



In [111…
```
# plot the rewards per episode
plt.scatter(list(range(len(mean_rewards))), mean_rewards)
plt.title("Mean rewards over time, Prioritised Replay")
plt.xlabel("Episode")
plt.ylabel("Mean rewards")
plt.show()
```



In [ ]:
```
SAVE_PATH = '/content/drive/My Drive/Colab Notebooks/DRL/DRL Coursework/'
```

In [112…
```
# Save the data
# get a reference to the file in write bytes mode
with open(SAVE_PATH + 'data/PrioritisedReplayRaw/rewards.pkl', 'wb') as f:
    # serialise the rewards array
    pickle.dump(rewards, f)
with open(SAVE_PATH + 'data/PrioritisedReplayRaw/mean_rewards.pkl', 'wb') as f:
```

```
    # serialise the rewards array
    pickle.dump(mean_rewards, f)
# save the state dict (from https://pytorch.org/tutorials/beginner/saving_loading_mode
torch.save(Q_net.state_dict(), SAVE_PATH + 'data/PrioritisedReplayRaw/qnet.pth')
```

Both

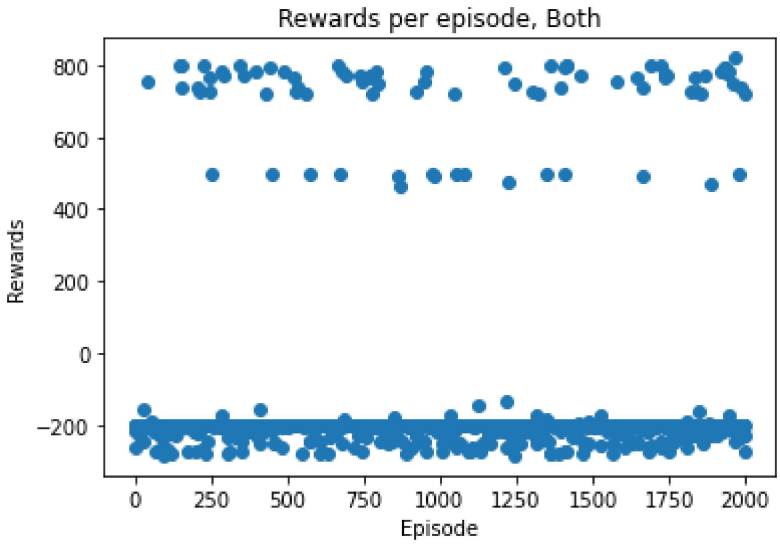
In [ ]:

```
# start training the network
# setup the networks and send them to the device
Q_net = DQN().to(device)
Q_target = DQN().to(device)
# load the state dictionary of weights from the q network to the target
Q_target.load_state_dict(Q_net.state_dict())
# set the target to evaluation mode
Q_target.eval()

priority_memory = fill_priority_memory(Q_net, priority_memory)

rewards = []
mean_rewards = []
for episode in range(2000):
    # Get an instance of the map
    amap = AdvancedMap()
    # Get the normalised observations
    obs, reward, done = amap.convert_observations(amap.reset())

    # setup the total reward
    total_reward = 0

    steps_done = 0

    while not done:
        Q_net.eval()
        # put the state into a tensor
        state = torch.from_numpy(obs).unsqueeze(0)

        # pick an action and increment the number of steps
        action, steps_done = select_action(state, steps_done), steps_done + 1

        # get the action
        action = adv_action_from_index(action.item())

        # take the action
        next_state, reward, done = amap.convert_observations(amap.step(action))

        # add to the reward
        total_reward += reward

        # set next state to nothing if we're done
        if done:
            next_state = None

        # add to the memories
        priority_memory.push(obs, torch.tensor([action.index], device=device), next_st

        # pass the next_state through to the next round
        obs = next_state
```

```
    # train the model
    Q_net.train()
    loss = optimise_model_priority(doubleQ=True)

    # add the rewards history
    rewards.append(total_reward)

    # update the target
    if episode % TARGET_UPDATE == 0:
        Q_target.load_state_dict(Q_net.state_dict())
    # work out the mean rewards
    if len(rewards) < 100:
        mean_reward = np.mean(rewards)
    else:
        mean_reward = np.mean(rewards[-100:])
    mean_rewards.append(mean_reward)
    # print info
    print(f"Episode: {episode}, reward: {total_reward}, loss: {loss.item()}, Mean rewa

    # early stopping
    if mean_reward > 0:
        print("Stopped early")
        break
```

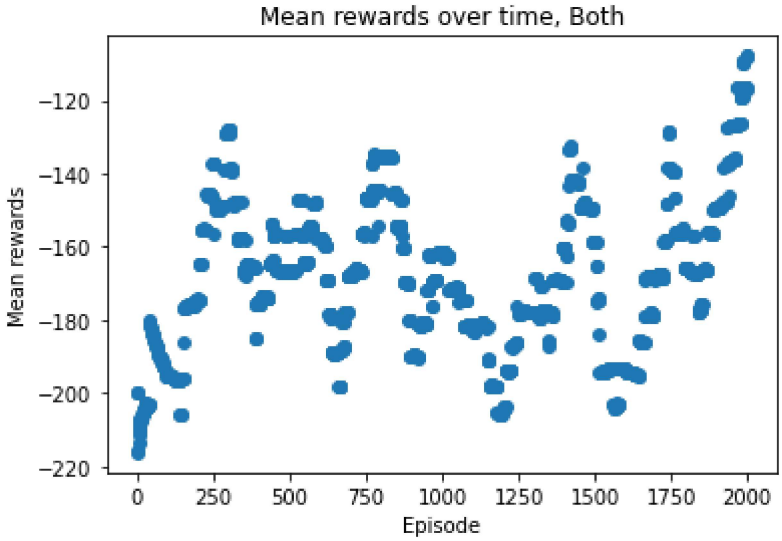
In [114…

```
# plot the rewards per episode
plt.scatter(list(range(len(rewards))), rewards)
plt.title("Rewards per episode, Both")
plt.xlabel("Episode")
plt.ylabel("Rewards")
plt.show()
```



In [115…

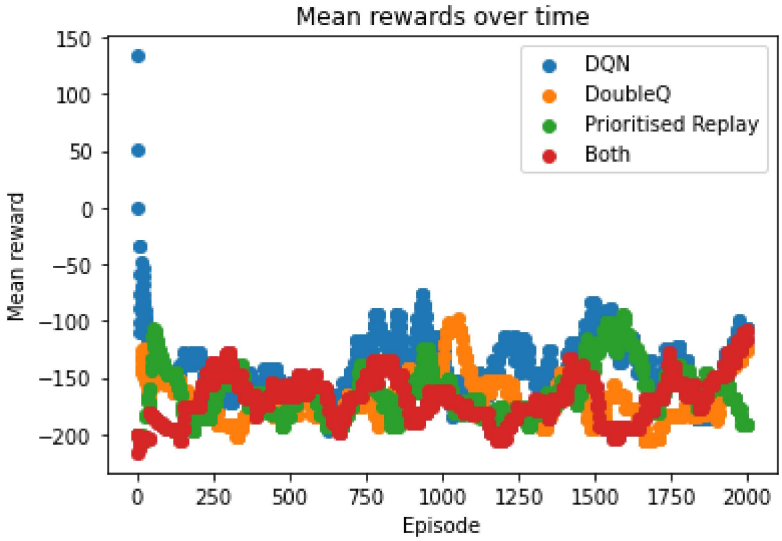
```
# plot the rewards per episode
plt.scatter(list(range(len(mean_rewards))), mean_rewards)
plt.title("Mean rewards over time, Both")
plt.xlabel("Episode")
plt.ylabel("Mean rewards")
plt.show()
```



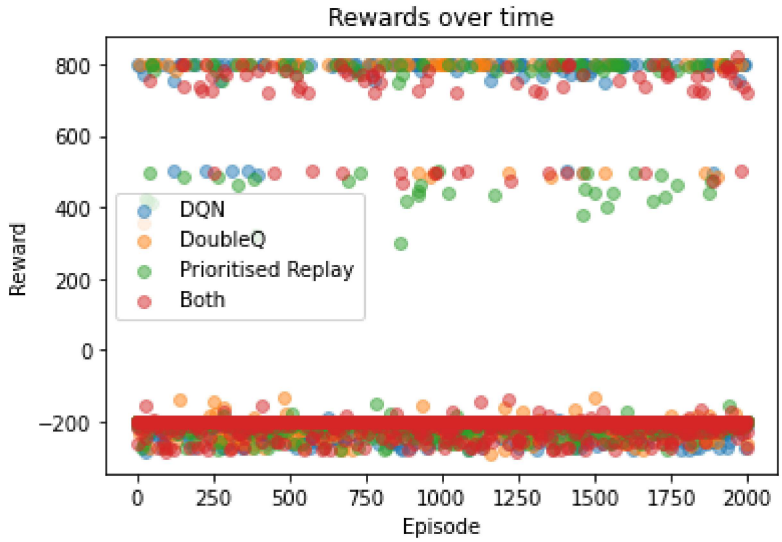
In [116…
```
# Save the data
# get a reference to the file in write bytes mode
with open(SAVE_PATH + 'data/BothRaw/rewards.pkl', 'wb') as f:
    # serialise the rewards array
    pickle.dump(rewards, f)
with open(SAVE_PATH + 'data/BothRaw/mean_rewards.pkl', 'wb') as f:
    # serialise the rewards array
    pickle.dump(mean_rewards, f)
# save the state dict (from https://pytorch.org/tutorials/beginner/saving_loading_mode
torch.save(Q_net.state_dict(), SAVE_PATH + 'data/BothRaw/qnet.pth')
```

Graphs of mean rewards

In [117…
```
for nm, location in [('DQN', 'NoImprovementRaw'), ('DoubleQ', 'DoubleQRaw'), ('Priorit
    # get the mean_rewards and graph them
    with open(SAVE_PATH + 'data/' + location + '/mean_rewards.pkl', 'rb') as f:
        mr = pickle.load(f)
    plt.scatter(list(range(len(mr))), mr, label=nm)
plt.title('Mean rewards over time')
plt.legend()
plt.xlabel('Episode')
plt.ylabel('Mean reward')
plt.show()
```

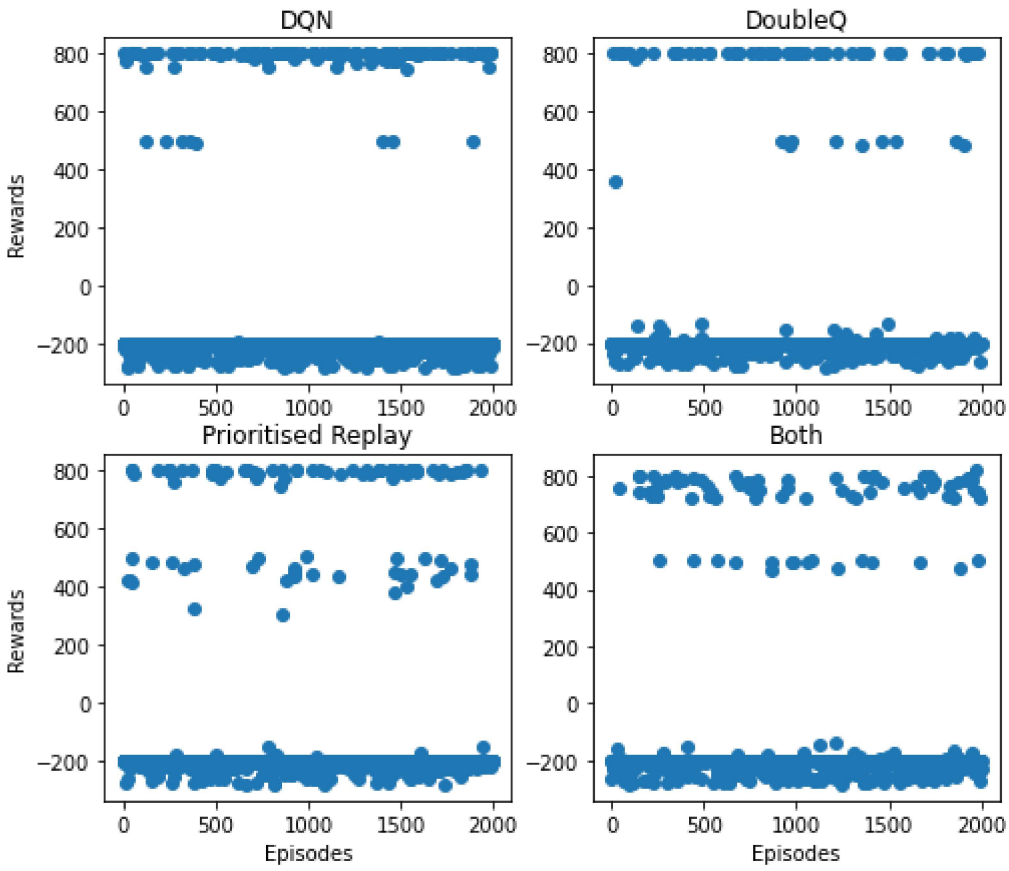


In [118…
```
for nm, location in [('DQN', 'NoImprovementRaw'), ('DoubleQ', 'DoubleQRaw'), ('Priorit
    # get the mean_rewards and graph them
    with open(SAVE_PATH + 'data/' + location + '/rewards.pkl', 'rb') as f:
        mr = pickle.load(f)
    plt.scatter(list(range(len(mr))), mr, label=nm, alpha=0.5)
plt.title('Rewards over time')
plt.legend()
plt.xlabel('Episode')
plt.ylabel('Reward')
plt.show()
```



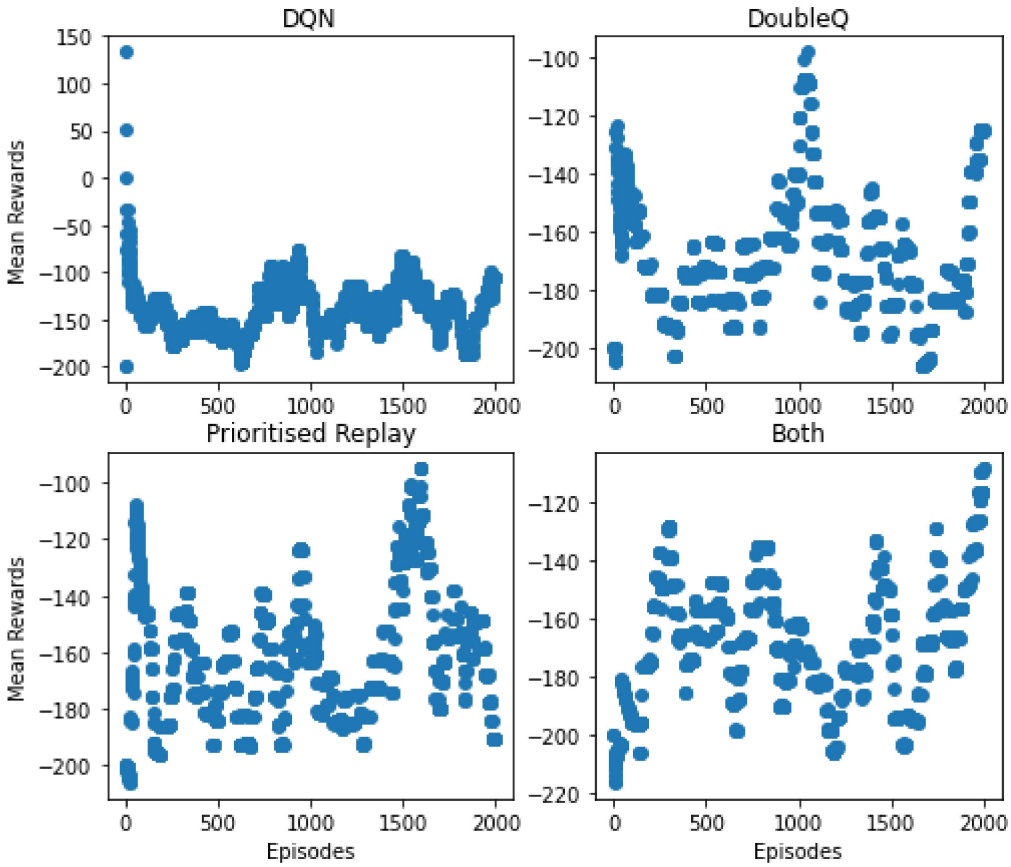
In [119…
```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(8,7))

for nm, location, row, col in [('DQN', 'NoImprovementRaw', 0, 0), ('DoubleQ', 'DoubleQ
    # get the mean_rewards and graph them
    with open(SAVE_PATH + 'data/' + location + '/rewards.pkl', 'rb') as f:
        mr = pickle.load(f)

    # DQNNoImprovement
    ax[row][col].scatter(list(range(len(mr))), mr)
    ax[row][col].set_title(nm)
    if col == 0:
        ax[row][col].set_ylabel("Rewards")
```

```
    if row == 1:
        ax[row][col].set_xlabel('Episodes')
plt.show()
```



In [120…

```
fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(8,7))

for nm, location, row, col in [('DQN', 'NoImprovementRaw', 0, 0), ('DoubleQ', 'DoubleQ
    # get the mean_rewards and graph them
    with open(SAVE_PATH + 'data/' + location + '/mean_rewards.pkl', 'rb') as f:
        mr = pickle.load(f)

    # DQNNoImprovement
    ax[row][col].scatter(list(range(len(mr))), mr)
    ax[row][col].set_title(nm)
    if col == 0:
        ax[row][col].set_ylabel("Mean Rewards")
    if row == 1:
        ax[row][col].set_xlabel('Episodes')

plt.show()
```



Videos of agents solving environment

In [121…
```
# for getting the videos
from matplotlib import rc
import matplotlib.animation as animation
rc('animation', html='jshtml')
```

In [122…
```
def make_video(weights_path: str) -> None:
    """Make a video of the agent in the environment"""
    # load the neural network
    net = DQN().to(device)
    # using code from https://pytorch.org/tutorials/beginner/saving_loading_models.htm
    net.load_state_dict(torch.load(SAVE_PATH + 'data/' + weights_path + '/qnet.pth'))

    # we are not doing anything to the network, so it will be in evaluation mode
    net.eval()

    # Get an instance of the map
    amap = AdvancedMap()
    # Get the normalised observations
    obs, r, done = amap.convert_observations(amap.reset())

    # store the video
    vid = []
    # store the total rewards
    total_rewards = []

    # put in the start
    vid.append(amap.agents_view())
```

```
    total_rewards.append(0)
    total_reward = 0

    # run until the end
    while not done:
        # convert state into tensor
        obs = torch.from_numpy(obs).unsqueeze(0)
        # work out the actions
        out = net(obs)
        # use argmax to get the index of the action to take and then use item to get i
        out = torch.argmax(out).item()

        # take the action
        obs, r, done = amap.convert_observations(amap.step(adv_action_from_index(out))
        # add up the rewards
        total_reward += r

        # append to video
        vid.append(amap.agents_view())
        total_rewards.append(total_reward)

    # using code from Computer vision lab 5, animation
    fig, ax = plt.subplots()
    def frame(i: int):
        """Render a frame of video"""
        # dispose of the last frame
        ax.clear()
        # get rid of the axis labels
        ax.axis('off')
        ax.set_title(f"Total reward {total_rewards[i]}")
        # render the agent's view
        return ax.imshow(vid[i])

    # build up the video
    anim = animation.FuncAnimation(fig, frame, frames=len(vid))

    # dispose of the graph object so that the last frame of the video isn't returned
    plt.close()

    # set the path to the ffmpeg program (install it using apt-get install ffmpeg, the
    plt.rcParams['animation.ffmpeg_path'] = '/usr/bin/ffmpeg'

    writervideo = animation.FFMpegWriter(fps=30) # from https://www.geeksforgeeks.org/

    anim.save(SAVE_PATH + 'videos/' + weights_path + '.mp4', writer=writervideo)

    return anim
```

In [123…

```
# DQN
make_video('NoImprovementRaw')
```

Out[123]:

Once Loop Reflect

In [124…
```
# Double DQN
make_video('DoubleQRaw')
```

Out[124]:

Once Loop Reflect

In [125…
```
# Prioritised Replay
make_video('PrioritisedReplayRaw')
```

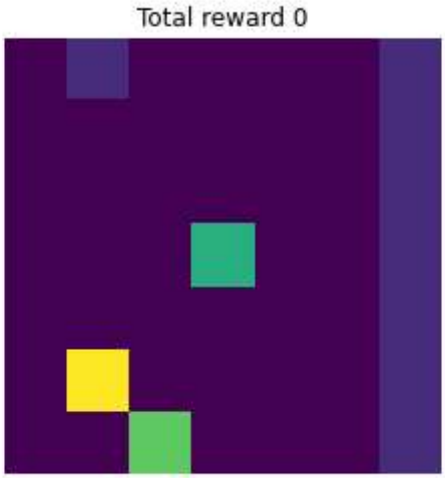
Out[125]:

Once Loop Reflect

In [126…

```
# Both
make_video('BothRaw')
```

Out[126]:



Once Loop Reflect

Soft-Actor Critic

In [1]:
```
# From Computer vision lab 6, keeps reloading the code for the objects so that you car
%load_ext autoreload
%autoreload 2
SAVE_PATH=''
```

In [2]:
```
# for colab
from google.colab import drive
drive.mount('/content/drive')
path = '/content/drive/My Drive/Colab Notebooks/DRL/DRL Coursework/'

import sys
sys.path.append(path)

SAVE_PATH=path
```

```
Mounted at /content/drive
```

In [3]:
```
# imports
from helpers.get_available_actions import get_available_actions, adv_action_from_index
import numpy as np
from helpers.advanced_map import AdvancedMap
import matplotlib.pyplot as plt
%matplotlib inline
import random
import torch
import math
import pickle
from collections import namedtuple, deque

# determine if we are using cpu or gpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In [4]:
```
# tuple from page 3, part 3.1 of https://arxiv.org/pdf/1801.01290.pdf
# state space
# action space
# state_transition_probability is initially unknown, s*s*a
# reward amount
MDP_tuple = namedtuple('MDP_tuple', ['state', 'action', 'next_state', 'reward', 'done'
# from https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html
EPS_START = 0.9
EPS_END = 0.05
EPS_DECAY = 200
```

In [5]:
```
# Build the Deep Q Network
class DQN(torch.nn.Module):
    """Deep Q Network, sizes based upon the grid search in RLLib"""

    def __init__(self):
        """Create an instance of the network"""
        # Call the base objects init method
        super().__init__()

        # our advanced network will be sending in a 53 column list of numbers
```

```
        self.dense1 = torch.nn.Linear(53, 256)
        # normalise the data in the network
        self.norm1 = torch.nn.BatchNorm1d(256)

        self.dense2 = torch.nn.Linear(256, 256)
        self.norm2 = torch.nn.BatchNorm1d(256)

        self.dense3 = torch.nn.Linear(256, 256)
        self.norm3 = torch.nn.BatchNorm1d(256)

        # 5 potential actions
        self.dense4 = torch.nn.Linear(256, 5)

    def forward(self, x):
        """Pass the data through the network"""
        # convert the input data to a float, run it through the first dense layer, the
        output = torch.functional.F.relu(self.norm1(self.dense1(x.float())))
        # repeat for
        output = torch.functional.F.relu(self.norm2(self.dense2(output)))

        output = torch.functional.F.relu(self.norm3(self.dense3(output)))
        # reshape the tensor and pass it through the last stage
        return self.dense4(output.view(output.size(0), -1))
```

In [10]:

```
class Pi(torch.nn.Module):
    """Need to work out whether to explore or not"""

    def __init__(self):
        """Create an instance of the network"""
        # Call the base objects init method
        super().__init__()
        # our advanced network will be sending in a 53 column list of numbers
        self.dense1 = torch.nn.Linear(53, 256)
        # normalise the data in the network
        self.norm1 = torch.nn.BatchNorm1d(256)

        self.dense2 = torch.nn.Linear(256, 256)
        self.norm2 = torch.nn.BatchNorm1d(256)

        self.dense3 = torch.nn.Linear(256, 256)
        self.norm3 = torch.nn.BatchNorm1d(256)

        # 2 potential policies, exploit or explore
        self.dense4 = torch.nn.Linear(256, 1)

    def forward(self, x):
        """Pass the data through the network"""
        # convert the input data to a float, run it through the first dense layer, the
        output = torch.functional.F.relu(self.norm1(self.dense1(x.float())))
        # repeat for
        output = torch.functional.F.relu(self.norm2(self.dense2(output)))

        output = torch.functional.F.relu(self.norm3(self.dense3(output)))
        # reshape the tensor and pass it through the last stage
        return torch.sigmoid(self.dense4(output.view(output.size(0), -1)))
```

In [7]:

```
class ReplayMemory(object):
    """from https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html"
    def __init__(self, capacity):
```

```
        # uses a deque structure so that as new items are added to the end of the 'lis
        self.memory = deque([], maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(MDP_tuple(*args))

    def sample(self, batch_size):
        """Return a random sample from the memory"""
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)

    def reset(self):
        """Clear the memories"""
        # reset the memory
        self.memory = deque([], maxlen=self.memory.maxlen)
```

In [8]:
```
def select_action(state, steps_done, policy) -> torch.tensor:
    """Inspired by https://pytorch.org/tutorials/intermediate/reinforcement_q_learning
    Takes in the current state and the number of steps and returns an action either fr

    # if the random number is bigger than the threshold then work out an action from t
    if policy > 0.5:
        # don't work out the gradient for this
        with torch.no_grad():
            # get the max column value, then gets its index and reshape into the expec
            return Q_net(state).max(1)[1].view(1, 1)
    # as there is a return in the if statement we don't need an else. return a random
    return torch.tensor([[random.randrange(5)]], device=device, dtype=torch.long)
```

In [28]:
```
# using https://spinningup.openai.com/en/latest/algorithms/sac.html

# setup the networks and send them to the device
Q_net = DQN().to(device)
Q_target = DQN().to(device)
# load the state dictionary of weights from the q network to the target
Q_target.load_state_dict(Q_net.state_dict())
# set the target to evaluation mode
Q_target.eval()

# setup the optimiser
Q_optim = torch.optim.SGD(Q_net.parameters(), lr=0.01)

Policy_net = Pi().to(device) # theta
#Policy_target = Pi().to(device)
#Policy_target.load_state_dict(Policy_net.state_dict())

P_optim = torch.optim.SGD(Policy_net.parameters(), lr=0.01)

# from table 1
capacity = 10000
discount = 0.99 # γ

D = ReplayMemory(capacity)
num_updates = 20
BATCH_SIZE = 256
```

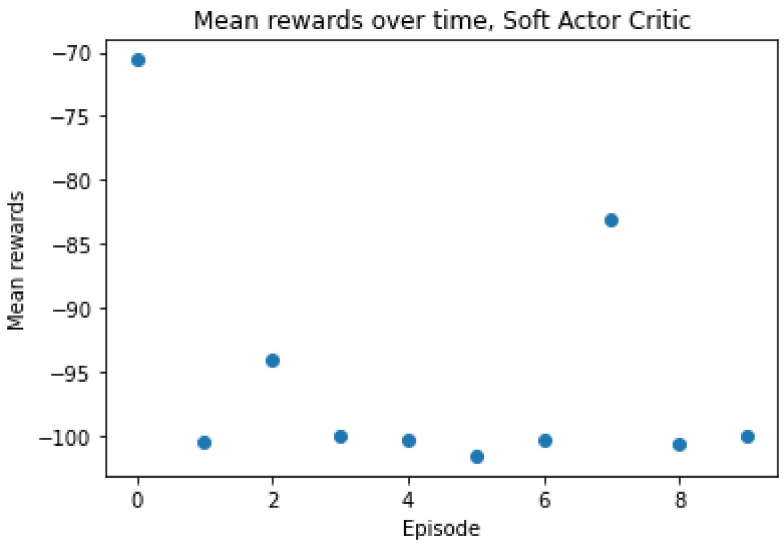
```
ALPHA = 1 # TODO find a less random number
GAMMA = 0.9

mean_rewards = []

TARGET_UPDATE = 1

for episode in range(1, 11):
    # fill up buffer with random actions
    D.reset()
    Q_net.eval()

    rewards = []

    while len(D) < capacity:
        # fill it up
        # Get an instance of the map
        amap = AdvancedMap()
        # Get the normalised observations
        obs, reward, done = amap.convert_observations(amap.reset())

        # setup the total reward
        total_reward = 0

        steps_done = 0
        Policy_net.eval()

        # run until it is done
        while not done:
            # put the state into a tensor
            state = torch.from_numpy(obs).unsqueeze(0)

            # pick an action and increment the number of steps
            pol = Policy_net(state)
            pol = pol.item()
            action, steps_done = select_action(state, steps_done, pol), steps_done + 1

            # get the action
            action = adv_action_from_index(action.item())

            # take the action
            next_state, reward, done = amap.convert_observations(amap.step(action))

            # add to the reward
            total_reward += reward

            # set next state to nothing if we're done
            if done:
                next_state = None

            # add to the memories
            # namedtuple('MDP_tuple', ['state', 'action', 'next_state', 'reward'])
            D.push(obs, torch.tensor([action.index], device=device), next_state, torch

            # pass the next_state through to the next round
            obs = next_state
        rewards.append(total_reward)
    # start updating
    for j in range(num_updates):
        Policy_net.train()
```

```
        # get a batch of transitions from D
        transitions = D.sample(BATCH_SIZE)
        # convert the batch of transitions to a transition of batches
        batch = MDP_tuple(*zip(*transitions))


        # compute a mask of non-final states
        #non_final_mask = torch.tensor(tuple(map(lambda state: state is not None, batc

        # get the next states
        non_final_next_states = torch.cat([torch.tensor([s], device=device) for s in b

        state_batch = torch.cat([torch.tensor([s], device=device) for s in batch.state
        action_batch = torch.cat(batch.action)
        reward_batch = torch.cat(batch.reward)
        action_batch = action_batch.reshape((BATCH_SIZE, 1))

        # compute target for the Q functions
        y = torch.zeros(BATCH_SIZE, device=device)

        targ = Q_target(non_final_next_states).min(1)[0].detach()
        targ = torch.reshape(targ, (len(non_final_next_states), 1))
        y = torch.reshape(targ - ALPHA * torch.log(Policy_net(non_final_next_states)),

        # shape y to fill the gaps
        ty = y.tolist()
        lst = 0
        y = []
        for ns in batch.next_state:
            if ns is None:
                y.append(0)
            else:
                y.append(ty[lst])
                lst = lst + 1
        y = torch.tensor(y, device=device)
        y = reward_batch + (GAMMA * y)

        # Update Q-functions by one step of gradient descent
        # work out the Q values
        q_vals = Q_net(state_batch).gather(1, action_batch)
        q_loss = torch.functional.F.mse_loss(q_vals, y)

        # optimise the Q_model
        Q_optim.zero_grad()
        q_loss.backward()

        # clip the gradients
        for param in Q_net.parameters():
            param.grad.data.clamp_(-1, 1)
        Q_optim.step()

        # Policy's turn (line 14 of https://spinningup.openai.com/en/latest/algorithms
        # update policy by 1 step of gradient ascent
        p_loss = torch.mean(Q_net(state_batch) - ALPHA * torch.log(Policy_net(state_ba

        # optimise the policy model
        P_optim.zero_grad()
        p_loss.backward()
        P_optim.step()
    # append mean of rewards to mean_rewards
```

```
    m_reward = np.mean(rewards)
    mean_rewards.append(m_reward)
    # update the target
    if episode % TARGET_UPDATE == 0:
        Q_target.load_state_dict(Q_net.state_dict())
    print(f"Episode {episode}, mean reward: {m_reward}")
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:139: UserWarning: Using
a target size (torch.Size([256])) that is different to the input size (torch.Size([25
6, 1])). This will likely lead to incorrect results due to broadcasting. Please ensur
e they have the same size.
Episode 1, mean reward: -70.57281553398059
Episode 2, mean reward: -100.50980392156863
Episode 3, mean reward: -94.13725490196079
Episode 4, mean reward: -100.0
Episode 5, mean reward: -100.36633663366337
Episode 6, mean reward: -101.58035714285714
Episode 7, mean reward: -100.31683168316832
Episode 8, mean reward: -83.12621359223301
Episode 9, mean reward: -100.59803921568627
Episode 10, mean reward: -100.0
```

In [29]:
```
# plot the rewards per episode
plt.scatter(list(range(len(mean_rewards))), mean_rewards)
plt.title("Mean rewards over time, Soft Actor Critic")
plt.xlabel("Episode")
plt.ylabel("Mean rewards")
plt.show()
```



In [30]:
```
# Save the data
with open(SAVE_PATH + 'data/SAC/rewards.pkl', 'wb') as f:
    # serialise the rewards array
    pickle.dump(mean_rewards, f)
# save the state dict (from https://pytorch.org/tutorials/beginner/saving_loading_mode
torch.save(Q_net.state_dict(), SAVE_PATH + 'data/SAC/qnet.pth')
torch.save(Policy_net.state_dict(), SAVE_PATH + 'data/SAC/policynet.pth')
```