Deep Reinforcement Learning Project

Thomas Fishwick

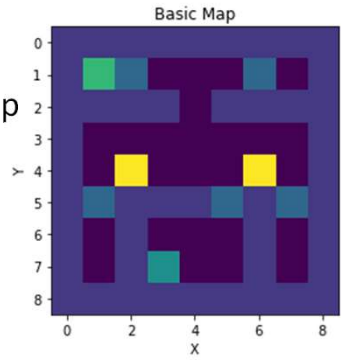
210020607

GitHub Link: https://github.com/SL477/DRL-coursework

Basic - Define an environment and the problem to be solved

- Here we are trying to solve an environment, which is a fairly simple maze defined in a NumPy array
- We have our two reward points, which need to be claimed in order by the agent
- We also have our two yellow traps
- Our agent starts at a random free location in the map

```
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]
    ])
```



The primary goal is the reactor control panel in the bottom middle. The secondary goal is the escape route top left. The purple walls are impassable and the doors (blue) are purely decorative. Our agent teleports into a random free location on the map.

Our environment is a Numpy array in the basic map function in the map module

For the random start it takes the list of potential states, then runs a while loop until it randomly chooses one which has a value of 0 indicating free space. We could have made this more efficient by filtering down to the free locations first

Basic – Define a state transition function and the reward function

- Every position on the map is a potential state and action. We flattened the map array, then using key of the cells we determined which ones were passable from one cell to another. So that each state could have up to four different actions.
- The reward function used the same mapping, but filling in the rewards that each action would give

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

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

Our trap will also take off 10 points of health each time our agent hits it. Once it gets to less than or equal to 0 the episode ends. Same if the agent collects the escape route.

Our Q matrix is first made, as it is just an empty array. The R matrix loops through the array of states working out their X and Y values, then checking the cells around it to work out whether there is a legal action to move there and what the reward would be (if it was a legal action).

The reward dictionary is then used to determine legal moves for the agent.

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

- We setup our Q-learning algorithm as a function and then grid-searched through it in parallel using Python's multi-processing library in order to get the best parameters in a reasonable amount of time
- Alpha (learning rate): 1
- Gamma (discount for future rewards): 0.8
- Epsilon (chance of exploiting over exploring): 0.9
- Number of episodes (not grid searched for): 1000

```
# 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]
    ))
```

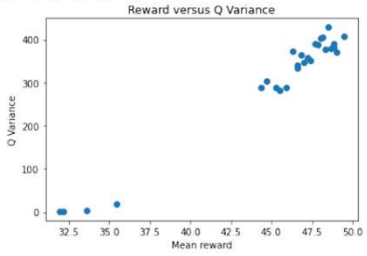
Because running each set of values sequentially would take a very long time we used Python's multi-processing in order to take advantage of the other processor cores.

We had to make sure that the different CPUs used different random seeds for working out whether to explore or exploit, otherwise they returned identical values (we should have done the same thing for the random start as well).

Basic – Run the Q-learning algorithm and represent its performance

- We used our grid-search from earlier, feeding in the same values repeatedly so that we could collect various statistics easily
- We set our Q-Learning algorithm to collect the mean health of the agent, the mean reward, the Q matrix variance and the number of episodes
- Using the dataframe of these we built various graphs to show its performance

```
# 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
}
```

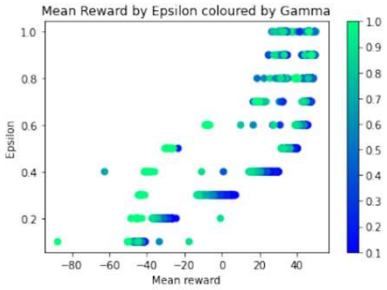


As there was a lot of variability in the scores from any individual runs due to differences in the starting position, we took advantage of our grid searching algorithm in order to process many runs at once. This gave us 27 runs worth of data for our model.

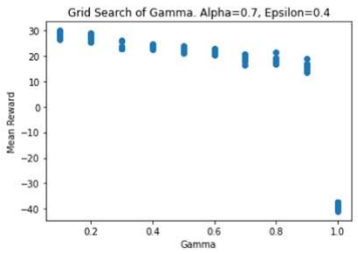
As you can see as the Q variance got higher the average rewards got higher.

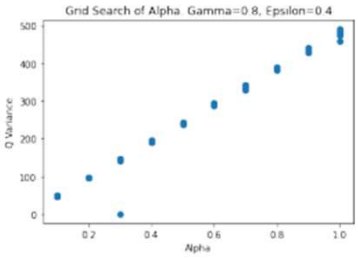
Basic – Repeat the experiment with different parameter values, and policies

- From our earlier grid searches, we looked for other interesting parameters and also looked at holding the Gamma stationary while changing the Alpha parameter (and vice versa). As we already had all of the code from earlier to collect the statistics we plugged the data into various graphs:




```
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()
```





We saved our dataframes to CSV files so that the code wouldn't need to be rerun to generate the graphs again

Basic – Analyse the results quantitatively and qualitatively

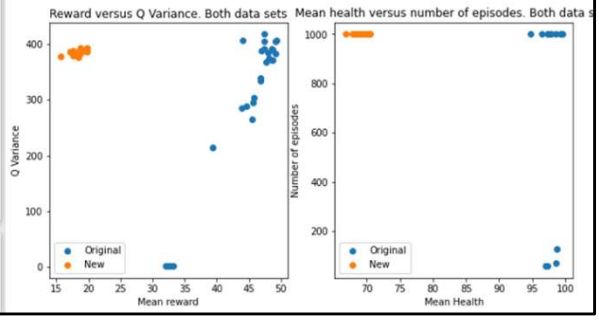
- From our grid search code we could repeatedly run the Q-algorithm to gather data on its performance

```
# 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
```

ownloads/basic_task_data.html 8/9

basic_task_data

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



From the reward data and health we could roughly work out what the algorithm had done. Unfortunately we never collected video data of what the basic algorithm was doing

Advanced-Implement DQN with two improvements, Double Q

- We used a more advanced version of our basic map as our environment
- Using RLLib we grid searched to get the best hyper-parameters
- We setup our neural network based upon this
- Our replay memory was based on the one from PyTorch which uses a dequeue
- So that we did not repeat code (too much) in the optimise model function we had an if statement to check if we were running in Double Q mode. If we were we predicted the column from the main DQN model and then got the target from the target network for that column. Otherwise we just got the biggest column from the target network. Then we compared the Q values and expected Q together

Our advanced map was much larger, featured something akin to ray-tracing for vision, with doors blocking sight. Moving enemies, and a compass to point to the objective

dequeue - like a list but once it hits capacity the first item is dropped to let the next one in

Our main challenge here was working out a way to easily flip between DQN and Double DQN, but it helped that the beginning and end of the algorithm was very similar

Advanced-Implement DQN with two improvements, Prioritised Replay

- The prioritised replay memory inherited from the replay memory class. Adding the sample function to work out the priority of the memory sample and returning that and the weights
- The optimise model function then had an extra section added to it (it was a new function with old one copied to it) which used the equations from the prioritised replay paper to create a weighted RMSE for the loss
- The new weights were then written back to the prioritised memory

```
# 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
```

```
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)
```

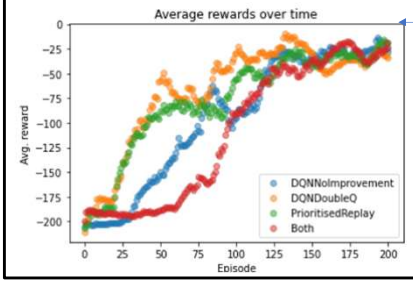
We could then run Prioritised Replay with the Double Q as well

Our main challenge here was in being able to write back the new weights of the memories' priorities and in creating a loss function which took into account the weights. Fortunately we were able to use PyTorch's mean function and used it to generate the weighted RMSE as the loss.

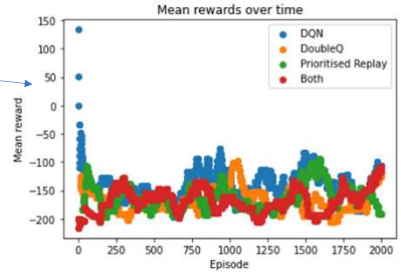
Advanced – Analyse the results quantitively and qualitatively

- We loaded our state dictionary into our neural network class and then used it to predict the next action. Every time it did this we saved the output array from the agent's point of view to a list. We then ran through this list converting them to pictures and saved each picture into a video file so that we could watch exactly what our agent had done.

RLLib



Ours



None of our algorithms did particularly well here.
We had also run an earlier version of our environment (the main change were the reward amounts) through RLLib's version of the algorithms, so that we were able to see how it had done. With either improvement improving on the base DQN, but not both.

Advanced – Apply RL algorithm of your choice (from RLLib) to one of the Atari Learning Environment

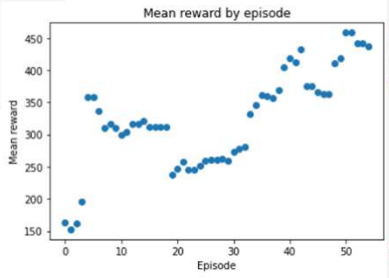
- Here we chose the Space Invaders environment and the Evolutionary strategies algorithm
- We used our video capture code to capture each frame of the environment into a video
- We then used Ray tune to tune our environment (Docker aborted this at around the 30 hours mark). We had some difficulty in setting Ray to use our 6 core CPU, as it wanted something much bigger

After having some issues in getting the environment working, we were able to start training. With Evolutionary Strategies taking a very long time per epoch

Advanced-Atari analyse the results quantitively and qualitatively

- From the CSV file which Ray Tune generated we generated a graph of the mean rewards over time
- We used similar code from the DQN section to capture each frame of what the agent was doing into a video

```
# 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')
```




```
# 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:
    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))
```

Since I did this coursework I implemented a much cleaner and more efficient video capture function which directly saves to MP4 as it goes along (Computer Vision coursework) rather than saving the Numpy arrays to a list and indexing into them, which is rather inefficient and causes your computer to die for bigger videos

Soft-Actor Critic

- Here we have our DQN and Replay Memory classes from earlier. But add another Neural Network Pi, to run a regression on the probability of whether or not to explore/exploit (our policy) (we exploit if it outputs >0.5, otherwise we explore)

```
lass 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 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 f

    # if the random number is bigger than the threshold then work out an action from
    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 expe
            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)
```

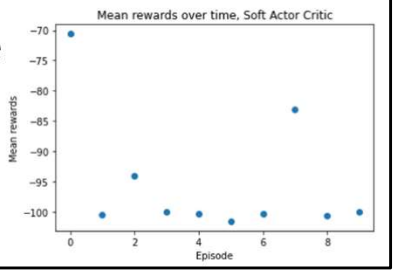
Once we had our models from DQN setup, it was relatively straightforward to setup another model to regress whether to explore or exploit

Soft-Actor Critic

- We fill up our memory at each episode using our Neural Networks to select actions
- We take our sample from the memories
- Compute the target Q values, sorting out which ones have no next state, adding rewards and multiplying by Gamma. Optimising the Q Network (this part is similar to DQN)
- Then we get the loss for the Pi network & optimise
- Then repeat until the networks converge

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

# optimise the policy model
P_optim.zero_grad()
p_loss.backward()
P_optim.step()
```



We fill up our memory at each episode using our Neural Networks to select actions – something that was probably a good idea to do for the DQN models

We ran out of time here so only ran the model through ten rounds of training. Our environment was roughly the one the DQN network used, but with the rewards from the Basic Task