Summary Report

My name is Vittoria Castelnuovo, my student details are: Vittoria.Castelnuovo@city.ac.uk, Student ID: 190042232.
For this project, I partnered with Geoffrey Payne (Geoffrey.Payne@city.ac.uk, Student ID: 190046385).

The code developed for task 1 takes inspiration from the Labs of the 707 module. I structured the environment in a class, as I believe this aids usability and exportability. Furthermore, it adheres to object-oriented principles. The functions in the Environment class attempt to mimic Open AI Gym's structure, as I believe their structure is quite clear and interpretable. Lastly, I found Phil Tabor's website a useful resource for the environment's implementation. Overall, the percentage of code borrowed for this task is about 5%.

The code developed for task 2 is also influenced by the Labs of this module. The main functionality of this task reflects the Bellman equation, which at the heart of reinforcement learning. Subsequently, used matplotlib to plot some graphs to visualize the algorithm's performance. Overall, the code borrowed for this task is about 5%, which accounts for the Bellman equation.

For task 3, we were influenced by Phil Tabor. We liked his code structure. His implementation is for the SAC algorithm, not SAC-Discrete, and he does not use CNNs for Atari games. See Actor-Critic-Methods-Paper-To-Code/SAC at master · philtabor/Actor-Critic-Methods-Paper-To-Code (github.com) For the Atari frame wrapper classes we lifted the code directly from the git repository for the Packt publication "Deep Reinforcement Learning Hands-On". We made minor changes to the code ans we assumed that extracting every fourth fram and converting images to grayscale to improve performance were good defaults. In this case, there is no need for us to do our own implementation of this code. . See https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On/blob/master/Chapter06/lib/wrappers.py
For the CNN code we also used code from Phil Tabor, see see Deep-Q-Learning-Paper-To-Code/deep_q_network.py at 2815d8a1ba6f7a2338a905c371ed1e429c8aeaac · philtabor/Deep-Q-Learning-Paper-To-Code (github.com).
Code was also taken from Petros Christodoulou, see p-christ/Deep-Reinforcement-Learning-Algorithms-with-PyTorch at 6297608b8524774c847ad5cad87e14b80abf69ce (github.com). The code we needed was spread between different classes so we had to make changes to his implementation. We would estimate given the specific implementation of what we were doing which did not match any of this code that we did about 60% of the code ourselves.

Personal Reflection

Having the opportunity to take this module was very valuable for me. It was extremely fascinating to learn about the state-of-the-art models and techniques used in reinforcement learning. Especially considering I had never studying this topic before. Initially, it was quite challenging to understand the topics introduced in this course as there are many different parts that need to be thoroughly understood in order to create robust RL systems. I found David Silver's video-lectures to be very helpful, as watching these before each week's lecture allowed me to get accustomed to the terminology and topics.
Upon completing this module and its coursework, I find myself excited to learn more about this subject and am eager to develop personal projects using the techniques I have learned thus far.

For this project, I worked with Geoffrey Payne. Working as a pair was really helpful as we were able to discuss and evaluate thoroughly the algorithms we implemented. I believe it is especially crucial to develop teamworking skills, as in many professional settings it is common practice to work alongside other developers.
Obviously, it was quite challenging to work in pairs due to the fact that every aspect of our academic journey is online. However, I believe Geoffrey and I, tried our best to work around this obstacle as we communicated very frequently through Zoom calls and Whatsapp. Initially, we came up with a few ideas each for what type of learning scenarios to tackle, and together decided to implement the Pacman game. For task 3, we researched and discussed which algorithm to implement and subsequently both implemented our own versions of the SAC-Discrete, before narrowing on a singular final version. We used the University's OneDrive so we could both update the report documents for tasks 1 and 3. We both worked on all aspects of the project, although Geoffrey was leading the code part, and I took leadership of the report portion of the project.

Task 1: The Environment

1. **Description of the environment**

The game of Pacman is an arcade game from the 1980's. Its rules are quite simple. Pacman, the yellow circular creature, is the agent. Its goal is to collect all the pellets (also referred to as breadcrumbs) in its environment, without getting intersected or "caught" by the ghost(s) that also reside in the environment. The game terminates when either Pacman collects all the rewards/ breadcrumbs (thus winning the game), or if one of the ghosts intersects Pacman (ie. losing the game).
On the OpenAI gym website there is an emulator for Pacman which can be found in the Atari games environments referred to as MsPacman-v0. This will be used for task 3.
For task 2 we will use our own simplified environment for Q Learning.
In our environment we will have

1. A 7 x 7 grid
2. Agent starts in the middle cell which is empty
3. All of the edge cells plus some other cells that are obstacles/barriers
4. 10 cells contain a breadcrumb. The actual number will be a parameter, typically between 1 and 10. These are randomly placed at the start of each episode
5. The remaining cells are empty
6. 1 ghost that moves randomly around the grid

See table 1 on the next page for the setup showing where the agent starts and where the obstacles, breadcrumbs and ghosts might be located.
For a game the obstacles will always be located in the same place for every episode. The ghost however moves randomly across the grid. In addition, the breadcrumbs are placed in random empty cells, at the start of each game.

2. Description of the agent and its actions

The agent starts each episode in the middle cell of the grid. It is represented as a square covering the cell of the grid.
The action it can carry out is to move one cell for each timestep. It has four actions, it can move; up, down, left, right.

3. Description of the different dynamics of the environment

The rules of the game are;

1. Agent starting position is the in the middle of the grid, cell 24.
2. An agent moves 1 square at a time. If an obstacle is encountered, it will remain in its old cell at the next time step
3. An episode can be completed in 2 ways;
   3a. if it the agent moves onto a ghost which moves randomly around the grid, the agent loses
   3b. If it collects all the breadcrumbs, the agent wins

The table below shows a typical layout, with each cell indexed by a number 0 - 48;

Table 1 Example layout of the environment

OBSTACLE 0	OBSTACLE 1	OBSTACLE 2	OBSTACLE 3	OBSTACLE 4	OBSTACLE 5	OBSTACLE 6
OBSTACLE 7	Breadcrumb 8	Breadcrumb 9	10	OBSTACLE 11	12	OBSTACLE 13
OBSTACLE 14	GHOST 15	16	Breadcrumb 17	Breadcrumb 18	Breadcrumb 19	OBSTACLE 20
OBSTACLE 21	Breadcrumb 22	23	Agent starts here 24	OBSTACLE 25	26	OBSTACLE 27
OBSTACLE 28	OBSTACLE 29	OBSTACLE 30	31	Breadcrumb 32	33	OBSTACLE 34
OBSTACLE 35	Breadcrumb 36	37	Breadcrumb 38	Breadcrumb 39	OBSTACLE 40	OBSTACLE 41
OBSTACLE 42	OBSTACLE 43	OBSTACLE 44	OBSTACLE 45	OBSTACLE 46	OBSTACLE 47	OBSTACLE 48

The state transition function is defined as ***s' = t(s, a)***, where s' is the next state, s is the current state, a is an action out of the available actions, a ∈A and t(s, a) is the transition function from the current state to the next.
The table below shows all the possibilities;

States	Actions (a)	Next state (***s' = t(s, a)***)
Start cell 24	Move to neighbouring obstacle cell 25	Remain in cell 24
Start cell 24	Move to neighbouring empty cell 23	Cell 23
Empty cell 23	Move to neighbouring breadcrumb cell 22	Breadcrumb cell 22 is now empty
Empty cell 16	Move to neighbouring ghost cell 15	Cell 15 and episode completed
The agent has just consumed all the breadcrumbs. The current cell is cell 39	No more actions	Cell 39 and episode completed.

The sooner the agent consumes the breadcrumbs, the less likely it will be intercepted by the ghost and lose.

In order to calculate the shortest route to achieve this aim, we need to assign rewards to provide the best incentives for the agent to win, and this will be the shortest route.

Cell type	Reward
Empty	-1
Breadcrumb	+10
Obstacle	-100
Ghost	-500
Last Breadcrumb	+100

Important to take into account that a cell that contains a breadcrumb can change state and become empty with a reward of -1.
The algorithm will need to take into account that this cell can have 2 reward values depending on whether it contains a breadcrumb of whether it is empty.

4. Implementation

We developed our own individual algorithmic implementations of the environment, as we completed our implementation of Task 2 first before realising that it was a requirement to implement a joint algorithm of the environment first.
My implementation's rewards were scaled by a factor of 2, and does not disrupt the dynamics of the Pacman game.

Task 2: Q-Learning

For Task 2 of this project, the Q-Learning algorithm was implemented for the Pacman game. The environment was developed to mimic the Open AI Gym's MS Pacman to facilitate usability, however the implementation is my own. The environment was programmed in Python, using numpy, matplotlib and the random module.

Q-Learning

Q-Learning is a value-based on-policy reinforcement learning algorithm. An agent operates within an environment and selects its actions based on its current state and its future rewards. This type of algorithm is known as 'model-free', meaning the model does not employ a transition function to generalize its policy. Instead, the agent learns from its experience in the environment directly, attempting to maximize its expected reward. The agent will continue updating its policy as it learns more about its environment by interacting with it.

Q-values are indicators of the quality of a particular action *a* in a given state *s*: *Q(s,a)*. These values indicate the best action for the agent to take in each particular state, and can be described as the agent's estimates of future expected reward at each time step *t*. These values are initialized to zero at the beginning of each episode, signifying that the agent starts each episode in an environment which is unknown to it. It is through the trial and error of interacting with the environment that these Q-values will be updated.
The Q-values are stored in a Q-table, which contains as many rows as the number of possible states and as many columns as the number of actions the agent can take. In this case, the Q-table is a 49x4 table. This is because the environment is a 7x7 grid, where its perimeter consists of barriers the agent cannot cross.

The Q-table contains the best possible actions the agent can take in each state; the Q-table is said to be optimal when its values are optimal, and enable the agent to receive the highest reward possible. Thus, the table represents the agent's best policy for acting in its environment.
The values in the Q-table are updated after each episode, according to the information the agent has received from the environment. At each time step *t*, the agent elects an action a_t, observes a reward r_t, and enters a new state s_{t+1}. This mechanism is elucidated formally by the Bellman equation:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \Big)}^{\text{temporal difference}} - \underbrace{Q(s_t, a_t)}_{\text{old value}}$$

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{new value (temporal difference target)}}$$

Hyperparameters

The learning rate α is a value between 0 and 1 which indicates how new information will replace old one. If α = 0, the agent does not learn any new information about the environment, and solely exploits existing one. Instead, if α = 1, the agent will only consider the newest information.

For this task, the learning rate was set to 0.1, to incentivize the agent to exploit the information learned by interacting with its environment. This proves useful as initially the greedy policy will dictate that the agent explores the environment by choosing random actions, and therefore it is important that the agent learns to quickly about its environment. Subsequently, when epsilon decays and the agent has experienced most parts of its environment, it is important for the agent to exploit the information found.

However, it is important to note that there is no *a priori* way of setting optimal hyperparameter values. This is because each reinforcement learning task may have its own particular set of requirements in order to achieve optimality. Moreover, in this tabular setting, it does not make a substantial difference whether alpha is set to 0.1 or 0.3, as the environment is small enough for the agent to learn it well regardless. This is no the case, however, in deep reinforcement learning, where hyperparameter tuning is very important and can deeply affect the performance and results of the system.

The discount factor γ indicates how valuable future rewards are with regards to immediate rewards. When γ = 0, the agent will not consider any future rewards, thus only attempting to maximize its most immediate reward. Instead, as gamma approaches the value of 1, the agent will strive to maximize of future rewards, even though they are not immediate.

In this task, the discount factor is set to 0.9 so that the agent strives to maximize long-term rewards almost as much as immediate rewards. This is because the only way for Pacman to win each episode, is to actually finish the game. Therefore, it must learn to collect all the pellets but understand the tradeoff between avoiding the ghost (which will lead to a substantial negative reward and the unsuccessful termination of the episode) and collecting pellets for immediate reward. Thus, if Pacman does not maximize its rewards with longevity, it may not be able to successfully terminate the episode at all.

Epsilon-Greedy Policy

In order for the agent to explore and exploit its environment efficiently, the Epsilon-Greedy policy is implemented. Using this policy, the agent is able to alternate between choosing the best action from the Q table, and choosing a random action to learn more information about its environment. This policy allows the agent to exploit useful information it already learned through trial and error, as well as enabling it to explore the environment to further its knowledge and ultimately improve ithe Q-table's Q-values.

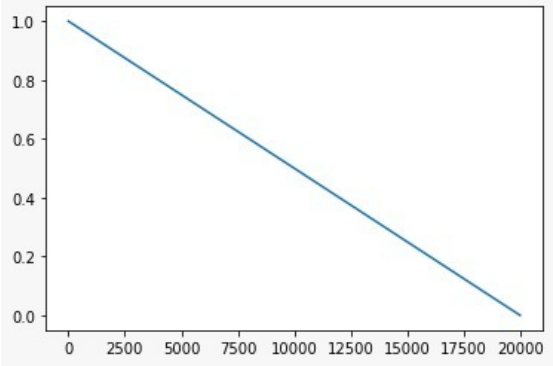

Figure 2.1: Graph of epsilon value steadily decaying at each episode

The epsilon-greedy policy is a simple solution to ensuring a balance between exploration and exploitation. With probability 1-ε, the action with the maximum reward will be chosen from the Q-table; with probability ε an action at random will be chosen. In this task, ε starts with a value of 1, and decays by 0.25 as the agent learns every 1000 episodes. Initially, while epsilon is 1, the agent is more likely to take random actions. This is favorable because the agent still does not know much about its environment, meaning the Q-table is mostly filled with zeros. However, as the agent moves through the environment, the epsilon value

decays, increasing the probability of actions being chosen from the Q-table. Therefore, the agent will start by exploring its environment and at each episode it will become more likely for Pacman to take advantage of, or exploit, learned information.

After 20,000 episodes we can see that ε = 0.025, thus meaning that the agent is exploring its environment much less than it was in the initial episodes, and therefore is now selecting its action based on the maximal reward.

Results and Evaluation

The Q-Learning algorithm performs quite successfully in this environment. The agent is able to win above 90% of games over 20,000 episodes on average. In fact, these results are remarkable considering that there are two elements of stochasticity in the environment: the 10 pellets are placed in different, random locations at the beginning of every episode, and the ghost in the environment moves randomly.

The success of this algorithm is can be attributed to the fact that the environment is a 7x7 grid, which is quite small. In fact, the number of states does not increase drastically as the agent explores more of its environment. This becomes clear when observing the Q-table, as we can see that the number of states at the 10000th episode is approximately 5000, and by the 20000th episode the number of states only increased by 500.

However, this algorithm can be deemed successful only if the environment remains this small. If, for instance, the environment was extended to a 1000x1000 grid, we can see that this algorithm would perform poorly. This is because the number of states would increase dramatically, and the value function could not accurately calculate each state-action value. In addition, having a Q-table of that large would be extremely computationally expensive and not time efficient. Deep reinforcement learning attempts to tackle this problem by implementing value function approximator networks, for environments which cannot be calculated tabularly.

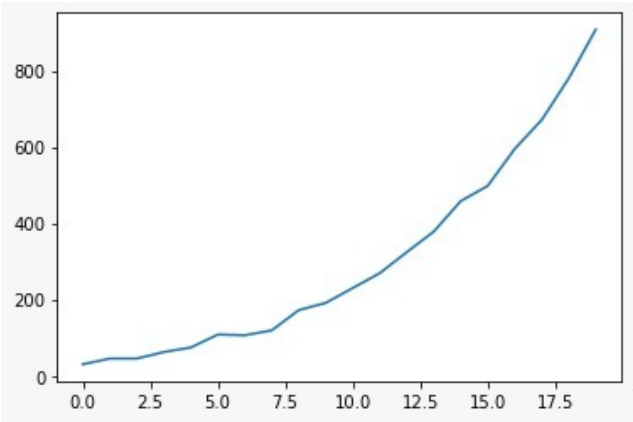

Figure 2.2: Graph of the number of games won by the Pacman agent each 1000 episodes.

As we can see in Figure 2.2, the agent's number of games won per every 1000 games increases each episode. This is promising and indicates that the agent is learning about its environment. Moreover, it demonstrates that the Q-table values are being updated in the correct manner, and thus Pacman's policy for winning the game is ameliorating. In Figure 2.3, we can see the agent learning at each episode, as the number of negative rewards decreases. This can be interpreted as the agent learning to avoid barriers (which give a negative reward) as well as learning to avoid the ghost. This is further demonstrated in Figure 2.4, where the number of moves the agent makes per episode increase with each episode. This indicates that the agent has learned to avoid the ghost in order to win the game, and therefore must take more steps to collect the pellets in the environment.

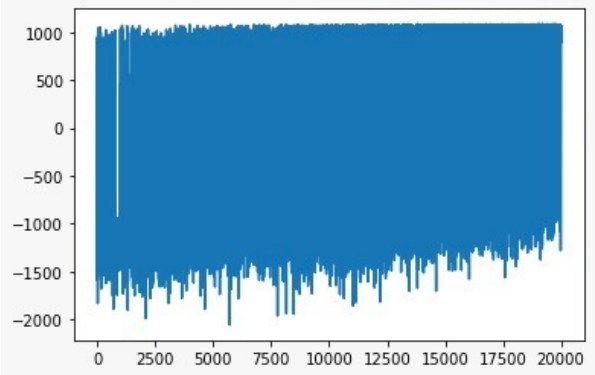

Figure 2.3: Graph of the total number of rewards per episode

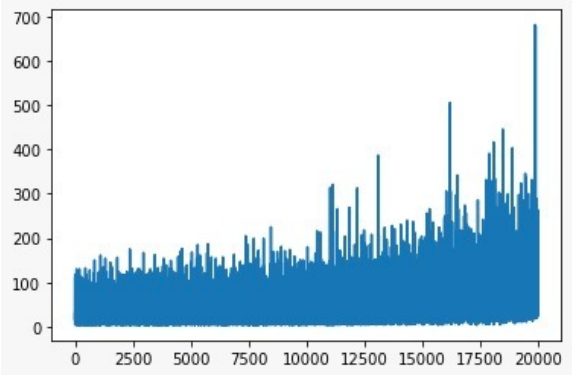

Figure 2.4: Graph of the number of moves per episode

Task 3: Soft Actor-Critic in Discrete-Action Pacman Setting

Soft Actor-Critic Algorithm

The algorithm chosen to complete Task 3 is the Soft Actor-Critic (SAC) one. This is a state-of-the-art deep reinforcement learning algorithm introduced by Haarnoja et al. in 2018. It is an off-policy algorithm employed in model-free scenarios, where the actor attempts to maximize both the entropy and its expected future rewards. The advantage of this algorithm being off-policy is that it is able to learn from past experience through the use of replay buffers which act as the agent's memory, rather than having to collect new samples for each policy update.

The aim for the SAC algorithm is to maximize expected reward at each episode and maximize the policy's entropy. Formally described as:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \sum_{t=0}^{T} E_{(s_t,a_t)\sim\tau_\pi}[\gamma^t(r(s_t,a_t) + \alpha\mathcal{H}(\pi(.|s_t))]$$

Where π is the policy, π* is the optimal policy, T refers to the number of timesteps, r: S x A à R is the reward function, γ is the discount factor, $s_t \in S$ is the state at time step t, $a_t \in A$ is the action at time step t, α is a measure for importance of entropy versus reward (referred to as the temperature parameter), and $H(\pi(.|s_t)$ is the policy's entropy at state s_t.

Entropy can be described as the degree of stochasticity, where if the entropy = 0 there is no degree of randomness or stochasticity. Maximizing entropy will prove useful for maintaining a high level of exploration, for actions with similar Q-values to be given the same probability, as well as guaranteeing that the no one action is selected repeatedly.

As Haarnoja et al. elucidate, the intuition behind SAC is to perform a soft policy evaluation and soft policy improvement in alternation, where the term 'soft' refers to regularized entropy. Through this alternation, the algorithm will converge on the optimal maximum entropy policy. The SAC algorithm is advantageous as it combines value-based and policy-based methods together, thereby allowing the agent to learn quickly in its environment (by updating its policy at each time step) and decrease the variance of the action distribution.

A disadvantage of this algorithm, however, is that the soft policy iteration yields the optimal solution only when performed in a tabular environment. Therefore, the deep learning implementation must employ an approximator of the value function in order to represent the Q-values. This is because the algorithm is created for continuous action domains.

SAC-Discrete

Petros Christodoulou's paper "Soft Actor-Critic for Discrete Action Settings" (2019) articulates the ways in which the SAC algorithm can be modified for discrete actions environments. As our Pacman game uses a set number of actions, this algorithm will be employed for this project's implementation.

The SAC-Discrete algorithm emulates the SAC traditional architecture, which entails two networks: an actor and a critic network. The actor model is responsible for selecting the actions the agent takes, thereby controlling how the agent behaves. As this is a policy-based method, the network is a function of the current state of the agent, the selected action.

Contrarily, the critic network is a value-based method which computes the value function. The aim of value-based models is to assign a value to each state-action pair and return the best action. The critic network is therefore responsible for quantifying how "good" the action taken by the actor network is.

Christodoulou identifies 5 key changes to the SAC algorithm to translate the algorithm from continuous to discrete action domains. Firstly, he explains that the critic network no longer outputs only the action, but rather outputs each possible action's Q-value. This change is possible due to the fact that actions are now a fixed number, whereas they were innumerable in the previous continuous-action settings. Secondly, the actor network is now the action distribution. The last layer of this network uses a softmax function in order to output a valid probability distribution. Moreover, the expectation over the action distribution is directly computed, as an approximation of its expectancy is no longer needed. This in turn means the reparameterization technique is no longer needed. Lastly, this also applies to the temperature loss, which is modified to decrease the variance of this estimate.

Our Implementation

Open AI Gym is a toolkit for implementing reinforcement learning algorithms and includes many different environments including Atari games and the Pacman game. Our implementation of the SAC-Discrete algorithm was programmed in the Python language and we used the Pytorch library.

Once the Pacman environment is installed and step up, the step() method provides the current state, the action, the reward, the new state and the 'done' flag, which indicates if the game reached a terminal state. As the input from the step() method comes from video frames, this must be controlled. This was done by using a wrapper class, which we inherited from PacktPublishing's "Deep Reinforcement Learning Hands On" GitHub repository[1]. We modified these wrapper classes to fit our project's scope, thereby enabling us to select every fourth frame from the video input. We also use these classes to grayscales the images to reduce the memory footprint and improve the performance of the algorithm.

The architecture and intuition of our SAC algorithm implementation was influenced from Phil Tabor's GitHub[2]. This resource was obviously drastically changed in order to meet our discrete-action environment's needs. We referenced Christodoulou's GitHub[3], to see how he implemented the improved SAC algorithm.

Our deep reinforcement learning implementation of the Pacman game using the SAC-Discrete algorithm primarily consists of an actor and a critic network.

The actor class is a neural network with 3 convolutional layers, 2 fully connected ones and a final linear one which pairs the last fully connected layer to the number of actions. The network's learning rate is dictated by the alpha hyper parameter, and we use the Adam optimizer. The forward() method activates the first 5 layers of the network using the ReLU activation function, and uses a softmax function over the action probabilities. The sample_action() method outputs the Q-value probabilities for each of the possible actions, through the use of Pytorch's Categorical() method which is commonly used for discrete action distributions.

Similarly, the critic network also consists of 3 convolutional layers and 2 fully connected ones. The Adam optimizer is also used in this network, however the learning rate is the hyperparameter beta. The forward() method uses the ReLU activation function on the convolutional layers, and returns the computed action value from each state.

The Hyper class encloses a series of hyperparameters for the different networks. As mention above, the actor network will inherit the alpha as its learning rate, whereas the critic network will use beta. The discount factor γ is set to 0.99, incentivizing the agent to maximize its long-term rewards. Lastly, the hyperparameter τ is set to 0.005, as tests show this is its optimal value.

The replay buffer class acts as the agent's memory. One of its main functions is to store the transitions at each time step. In addition, the buffer also randomly chooses a sample of the agent's experience, and returns a set of states, actions, rewards, new states and a Boolean value indicating whether the game is over.

Results & Evaluation

The results yielded by this system are difficult to interpret. In the first 60 episodes, the algorithm seems to increase steeply, giving the impression of learning quite rapidly yet sporadically. In fact, we observed some episode garner up to 3000 rewards. However, shortly thereafter the learning abruptly turns off, and keeps declining until around episode 350. This appears to be a pivotal moment were the learning starts to incline again, this time more slowly and controlled. As there is no clear trajectory, it is challenging to claim for certain whether genuine learning is taking place or not.

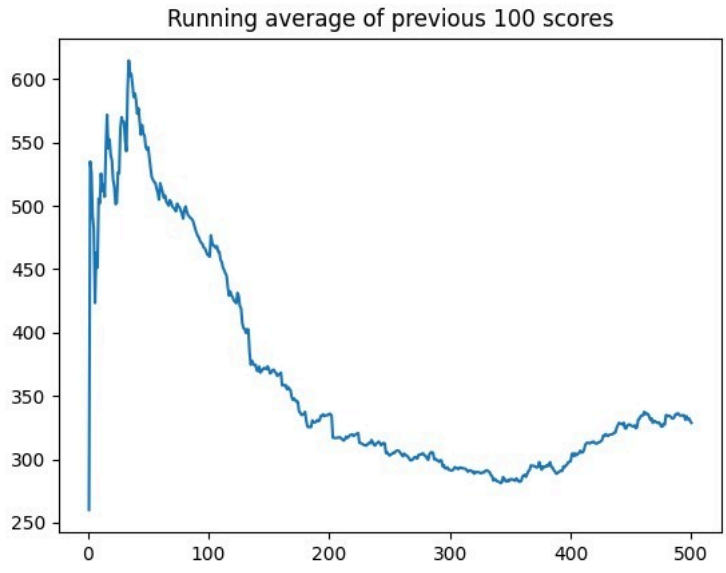

Figure 3.1: Graph of running average of previous 100 scores

Upon reviewing our implementation's performance, we can hypothesize some reasons for the unstable results. Firstly, we may argue that if the system had trained for longer, we would have been able to see a clearer trajectory of learning, after the initial variability previously discussed. In fact, Christodoulou's implementation of the SAC-Discrete algorithm is deemed successful after 100,000 steps; Instead, we only trained our algorithm

for 25,000 steps. Therefore, had we trained it for more steps, we may have been able to garner similar results.

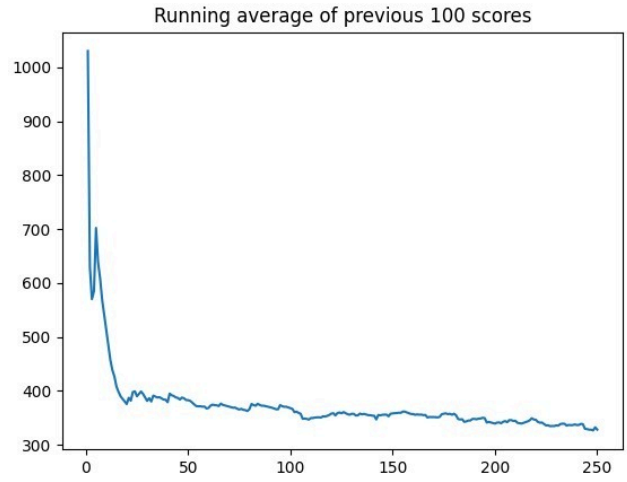

Figure 3.2: Graph of running average of previous 100 scores, where alpha and beta are 0.001

The second factor we identified as a possible culprit for our system's poor performance is a fault in hyperparameter tuning. In our system, we set alpha and beta to 0.0003, gamma to .99 and tau to 0.005. These values were chosen to mimic Christodoulou's implementation, in addition to being cited in deep reinforcement learning literature. However, hyperparameter tuning is very important, as it deeply effects the degree and manner of learning of a network. The quickest way to calculate these values for optimal performance is through trial and error. We would like to have used the Optuna library in our code, but this is not practical as it takes a lot of time to run. To experiment, we set the alpha and beta values to the larger value of 0.001. Although this confirms the previous pattern, the learning rate did not improve. In fact, using these hyperparameter garnered even more underwhelming results. When we used smaller values, the scores became fixed on the same value, so we stopped the test after about 10 games. For our testing so far, an alpha and beta value of 0.0003 has been the best value.

Overall, implementing the SAC algorithm enabled us to learn about many different types of Actor-Critic methods, as we had to research many different variations of it in order to find one well-suited for our Pacman game. The SAC-Discrete algorithm was interesting to implement, as it can be described as the intersection of value-based and policy-based methods, for deep reinforcement learning. However, as this state-of-the-art algorithm is a recent addition to its field, having been published in late 2019, it was difficult to find resources to learn about this algorithm in depth. We experimented a lot with our implementation, in an attempt to find the best possible performance. We implemented a secondary value function network, but ultimately decided not to use it in order to correctly and accurately implement the algorithm Christodoulou described. Moreover, we found that this deep network did not improve the system's performance.

In conclusion, in Task 1 we set up the reinforcement learning environment, describing the rules of the Pacman game, and what rewards the agent was to receive for each action. For Task 2, we developed our own individual implementations of the Q-Learning algorithm, using the environment described in the previous task. Lastly, for Task 3, we researched the SAC algorithm and implemented a variation of it (SAC-Discrete) to fit the specific needs of the Pacman game. For this portion of the task we imported the Open AI Gym, which enabled us to learn how to work with predefined environments.

References

[1] https://github.com/PacktPublishing/Deep-Reinforcement-Learning-Hands-On/blob/master/Chapter06/lib/wrappers.py
[2] Actor-Critic-Methods-Paper-To-Code/SAC at master · philtabor/Actor-Critic-Methods-Paper-To-Code (github.com)
[3] Deep-Reinforcement-Learning-Algorithms-with-PyTorch/SAC_Discrete.py at 6297608b8524774c847ad5cad87e14b80abf69ce · p-christ/Deep-Reinforcement-Learning-Algorithms-with-PyTorch (github.com)