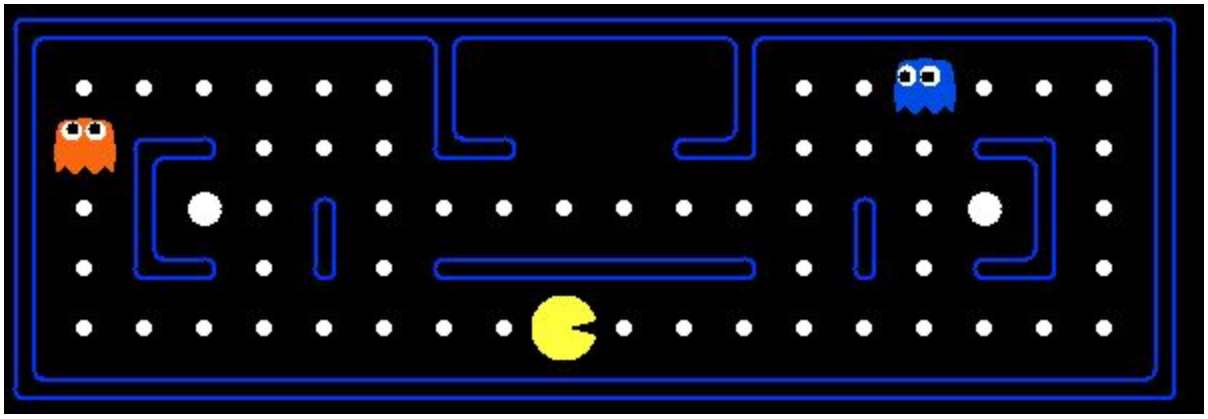
INM707 Coursework Report

IMPLEMENTING A PACMAN AGENT WITH REINFORCEMENT LEARNING TECHNIQUES

Cansın Ayşegül Sapmaz
ID: 190045685



1.Introduction **1**
1.1.Domain and Task 1
1.2.States, State Transitions and Rewards 1
1.3.Graph Representation and R-Matrix 3

2.Methodology **3**
2.1.Simple Q-Learning: Policy 4
2.2.Simple Q-Learning: Parameters and Q-Table Update 5
2.3.Deep Q-Learning with Experience Replay Memory 6

3.Results **6**
3.1.Simple Q-Learning: Performance per Episode 6
3.2.Simple Q-Learning: Analysis 7
3.3.Deep Q-Learning Results 9

References **11**

Appendix **12**
A.Game Layouts 12

1.Introduction

The goal of this project is to implement a Pacman agent that can play the game autonomously with success. Several different reinforcement learning techniques are utilized, namely: simple Q-learning, approximate Q-learning, and deep Q-learning with experience replay memory structure. This implementation is built on the Pac-Man Projects of Berkeley University of California. The Pac-Man Projects include readily made code for game mechanics and graphics without reinforcement learning techniques implemented (DeNero et al., n.d.). This project uses a Python 3 port of the project developed by Spacco (2017).

1.1.Domain and Task

Project domain is the world of Pacman. The task of the implemented agent is to eat all of the food present on the game map while navigating through it and avoiding enemy agents called ghosts. There are several layouts available to test the agent on with varied complexities.

Pacman is an interesting problem as it contains multiple factors to conquer for successfully reaching a goal state with the desired reward, such as exploring the entire map, finding the nearest food available, running from ghosts when the agent does not have a power-up, and defeating ghosts when power-ups are received. This makes Pac-Man a problem that can be generalized into simplified versions of complex real world tasks.

1.2.States, State Transitions and Rewards

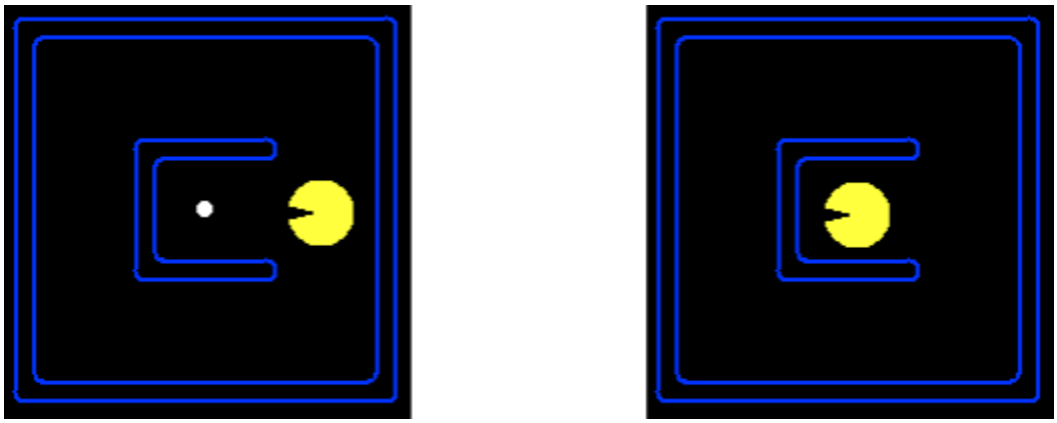
To model this problem as a reinforcement learning task, defining the states of the problem is necessary. A successful state definition should allow the problem to be modeled as a Markov decision process and include all information required to determine the probability distributions of possible rewards and the next possible states (Sutton and Barto, 2018). In this project the states are defined to be the entire game configuration including the location of Pacman, locations of ghosts, whether Pacman has an active power-up or not, and locations of power-up capsules, walls, and food. Terminal states are, then, any state where either all food is eaten or Pacman is caught by a ghost without a power-up. The problem is modeled in a discrete way, so the map is represented as a grid with discrete locations.

Some definitions are required to formally represent states.

- Agent: There are two types of agents; PacmanAgent and GhostAgent. The agent object entails information about the location of the agent and whether or not it is scared of opposite type agents; for instance Pacman is scared of ghosts when he does not have a power-up. Moreover, for algorithmic ease, each agent is indexed with a number that can be used to refer to it.
    - Agent = {Index, Location, IsScared}
- A: Set of all agents present in the game. Pacman is always indexed to be agent 0.
    - A = {Agent 0, Agent 1, Agent 2, …}
- W: Set of all wall locations. Walls include the map borders.
    - W = {Location1, Location2, …}

- F: Set of all food locations.
    - F = {Location1, Location2, …}
- P: Set of all power-up capsule locations.
    - P = {Location1, Location2, …}

For the sake of simplicity, introductory examples will use a simple 3x3 grid layout with no ghost agents, no power-up capsules and one food dot as shown in Figure 1.



(a) *(b)*

Figure 1: Example states of small grid layout. 1.a is the initial state, while 1.b is the next state if Pacman takes action to move left.

Transitions between one state to the other occur with discrete time steps as Pacman and/or ghosts take actions. Pacman moving right when it is allowed to, for instance, transitions the game state to another state where Pacman is located on one square to the right. If there is food or a power-up capsule on the location that Pacman moves to, the food or capsule is "eaten" automatically, i.e. it disappears from the game map. However, even if Pacman chooses not to move, ghosts moving still causes transitions into new states where they are in new locations. An example state transition can be Pacman moving left in the state given in Figure 1.a. The resulting next state is shown in figure 1.b. Pacman has 5 actions in total, namely LEFT, RIGHT, UP, DOWN, and STOP. Stopping is simply the action when Pacman chooses not to move. Formally, using the definitions of A, W, F and P provided above, the state transition function can be defined as

$$T(\{A, W, F, P\}, a) = \{A', W, F', P'\}$$

where a is one of the five actions stated above, and F' and P' are sets of food and power-up locations updated based on action a, respectively. Ghost agents move randomly, so A' is the set of agents with locations updated either by random probability (for ghosts) or based on action a (for Pacman). "IsScared" status is also updated for every agent based on action a of Pacman.

Criteria for success is the game score, which is based on reward. Pacman agent receives a certain reward value in every state transition, and the goal is to maximize the total reward it has accumulated by the end of the episode. To guide Pacman to finish episodes faster, every time step costs -1 reward points. Pacman receives 10 reward points for every food dot it eats, and successfully finishing all of the food on the map receives 500 reward points (510 points with 10

coming from the last food). Beating a ghost agent with the help of a power-up receives 200 reward points, while losing to a ghost agent (colliding with it without a power-up) receives -500.

1.3.Graph Representation and R-Matrix

The problem can be represented as a graph, nodes being states and edges being transitions. Edge weights can be used to represent the immediate reward of the given state transition. For instance, in the layout given as an example in Figure 1, there are 9 available locations that Pacman can be in. Since Pacman is the only agent present with a single food dot, there are 9 different states possible in total, forming a graph of 9 nodes. The example graph can be seen in Figure 2.

The rewards of each state transition can be represented with the R-matrix. The R-matrix shows every possible state, and the immediate rewards that can be received from legal transitions between them. In the R-matrix below, rewards of illegal state transitions are shown with a dash symbol. Possible actions at each state are represented by possible states that can be transitioned into as the columns of the table. State indices can be found on the graph representation shown in Figure 2.

	1	2	3	4	5	6	7	8	9
1	-1	-1	-	-	-	-	-	-1	509
2	-1	-1	-1	-	-	-	-	-	-
3	-	-1	-1	-1	-	-	-	-	-
4	-	-	-1	-1	-1	-	-	-	-
5	-	-	-	-1	-1	-1	-	-	-
6	-	-	-	-	-1	-1	-1	-	-
7	-	-	-	-	-	-1	-1	-1	-
8	-1	-	-	-	-	-	-1	-1	-
9	-	-	-	-	-	-	-	-	-

Table 1: R-matrix; y axis representing states, x axis representing transitions into new ones

2.Methodology

Two reinforcement learning techniques are implemented: basic Q-learning, and deep Q-learning with experience replay memory. These implementations are trained and tested in different game layouts depending on their limitations.

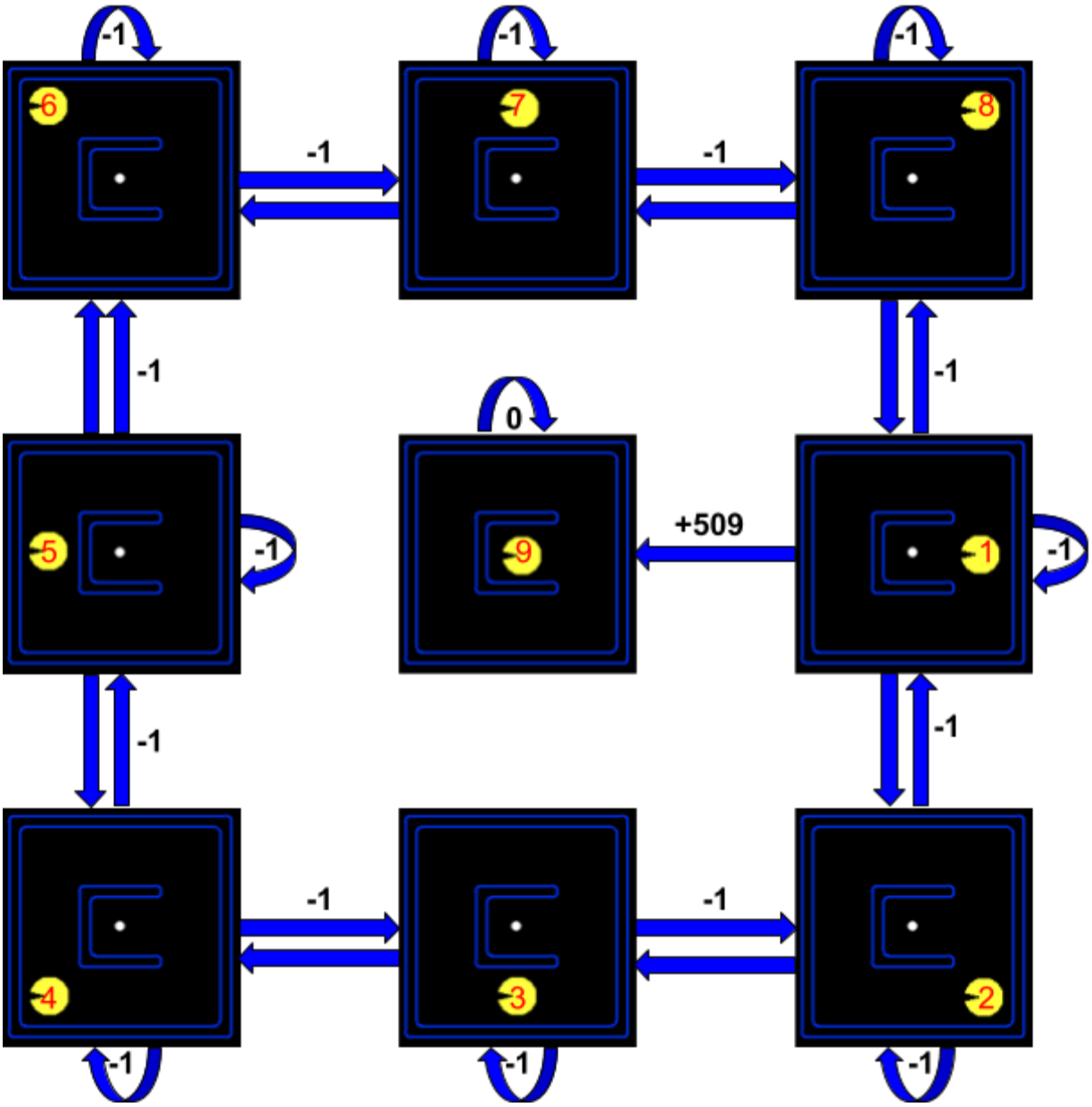


Figure 2: Graph representation of Pacman game in a small grid layout. Graph weights represent the immediate reward received in state transitions. Numbers on Pacman show the state index. Note that state 9 is a terminal state, which is represented by continuously returning to the same state with no more rewards.

2.1.Simple Q-Learning: Policy

A policy is needed for picking an action given a state. The main action exploration policy chosen for this task is epsilon-greedy, which chooses the action with the best Q-value most of the time, but behaves randomly with epsilon probability to ensure new actions are explored and learnt as well (Sutton and Barto, 2018).

2.2.Simple Q-Learning: Parameters and Q-Table Update

With simple Q-learning, optimal play is learnt by updating the Q-table, which holds the estimated returns; *Q(s, a)* for all states (*s*) and all actions (*a*) (Sutton and Barto, 2018). For this particular task, every entry in the Q-table is initialized to be 0. The table, then, is updated based on the Bellman equation. The update rule is as follows,

$$Q'(s, a) = Q(s, a) + \alpha \, . \, (r + \gamma \, . \, V(s') - Q(s, a))$$

Where *Q'(s, a)* is the new Q-value, *Q(s, a)* is the old Q-value, α is the learning rate, *r* is the latest return received, γ is the discount factor and *V(s')* is the optimum return of the next state *s'*. In simple Q-learning, *V(s')* is taken to be the maximum Q-value of *s'* over possible actions (Sutton and Barto, 2018). The main parameters picked are 0.2 for learning rate, 0.8 for discount factor, and 0.05 for epsilon.

As the state transitions from Figure 1.a to Figure 1.b in one time step with the given parameters, rewards, and policy, the Q-table is updated as shown in Table 2.

	UP	DOWN	RIGHT	LEFT	STOP
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0

(a)

	UP	DOWN	RIGHT	LEFT	STOP
1	0	0	0	101.8	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0
5	0	0	0	0	0
6	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0

(b)

Table 2: Q tables before (a) and after (b) the state transition given in Figure 1. Rows show state indices while columns show actions. The new value is calculated as;
Q'(s, a) = 0 + 0.2 x (509 + 0.8 x 0 - 0) = 101.8

Other parameters are tested as well for comparison. Performances of 'far-sighted pacman' (discount factor = 1), 'risk-taking pacman' (epsilon = 0.5) and 'fast-learning pacman' (learning rate = 0.5) can be found in sections 3.1 and 3.2.

2.3.Deep Q-Learning with Experience Replay Memory

In deep Q-learning, the underlying function of Q-values is estimated with a neural architecture. The implementation uses the architecture shown in the official Pytorch reinforcement learning tutorial provided as part of INM707 lab materials (Paszke, 2017). The structure and training were modified to fit the domain better. This project uses a model with a bilinear upsampling layer with scale size 5, three convolutional layers with kernel size 5 and stride 1, three batch normalization layers, and three fully connected layers. The upsampling layer allows the model to handle very small grid layouts as well as larger ones. The chosen optimizer is Adam with mean squared error as the loss function. Learning rate of the optimizer is chosen to be 0.0005. Epsilon is set to 0.1 during training with no decay; this was chosen because the deep Q-learning agent deals with larger layouts with more states to discover. The discount factor is 0.8; the same value used for the basic task.

The deep Q-learning agent has two networks: one policy network that provides q-values based on the current policy, and one target network that provides values based on older estimations. Target network's values are used to calculate estimated Q-values using the Bellman equation similar to simple Q-learning. For training the policy network, the loss between these estimated Q-values and the Q-values provided by the policy network is minimized. At the end of every episode, the target network is updated by copying the policy network's parameters (Paszke, 2017).

This implementation uses an experience replay memory of size 10000. State transitions are saved in the form of tuples of state, action, next state and immediate reward. Once the memory is filled with sufficient number of transitions, training starts in batches of 32 transitions randomly sampled from the memory (Paszke, 2017).

Instead of a screen capture, states are represented as one-hot encodings composed of 6 channels; one for Pacman's location on the board, one for the locations of ghosts scared of Pacman, one for the locations of un-scared ghosts, one for food locations, one for capsule locations, and one for wall locations. To prevent many weights from being cancelled in the network, 0.1 is used instead of 0 in one-hot encoding.

3.Results

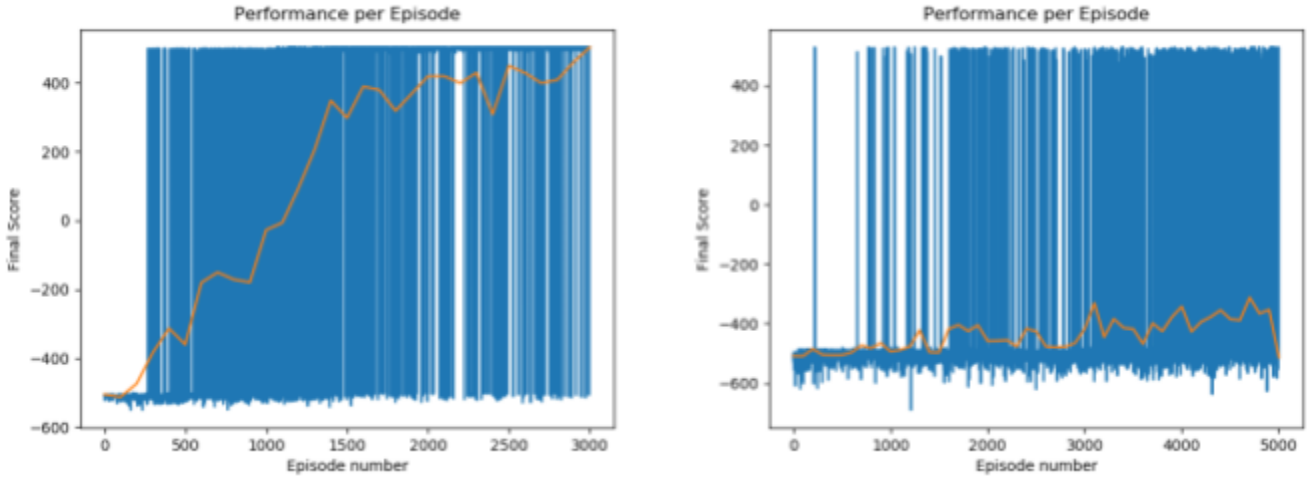
The implementations are trained and tested on three different game board layouts that can be found in the appendix. Small and medium grid layouts provide simple tasks with small maps to learn and a single ghost agent to avoid. Small classic is a more complex task that reflects the real world of Pacman. Test games are conducted by letting the agent play 100 games with epsilon set to 0.

3.1.Simple Q-Learning: Performance per Episode

Simple Q-learning can solve small grid layout in a reasonable amount of time, however larger layouts are troubling. Figure 4.a and 4.b show the performance over episodes in the small grid and medium grid layouts, respectively. These layouts offer very little food to collect rewards so

the final score ends up either around -500 or +500 with few games scoring in between. This results in a highly oscillating score per episode graph that can be seen in Figure 4 as blue lines. Representing the performance by average score per 100 episodes makes it easier to see the agent's behaviour converging to optimal policy, as represented by the orange lines in Figure 4.

Performances of agents on the small grid with other parameters can be seen in Figure 5. Different agents experimented with are: fast learner Pacman, whose learning rate is set to 0.5 with other parameters staying the same; risk taker Pacman, whose epsilon is set to 0.5 with other parameters staying the same; and far sighted Pacman, whose discount factor is set to 1.



(a) *(b)*

Figure 4: Simple Q-learning training scores per episode shown in blue, and average training scores per 100 episodes shown in orange. 4.a shows the results on the small grid while b shows the results on the medium grid layout.

3.2.Simple Q-Learning: Analysis

Simple Q-learning Pacman with default parameters is able to reliably win test games (with epsilon set to 0) on the small grid with an average score of +499.48 and a win rate of 1.00 after 3000 episodes of training. The performance in training, however, has a very high variance of 236583.28. This is due to the fact that with the way rewards are set, it is not very probable to finish the game with a score between -400 and +400. Despite the high variance, the orange plot in Figure 4.a shows convergence. When compared with Figure 4.b, it is clear that the agent is not close to estimating the optimal policy in the medium grid. In this larger layout, after 5000 episodes, the agent's average test score (with epsilon set to 0) is -404.20 with win rate 0.08.

Convergence can be more easily spotted with fast learner Pacman. With a higher learning rate, this agent reaches the optimal estimation earlier and thus stabilizes faster. Comparing Figures 4.a and 5.a, we can see a steeper curve in average scores per 100 episodes with a higher learning rate. Fast learner pacman has an average test score of +499.51 with a win rate of 1.00 after 3000 episodes of training.

Using a higher epsilon value lets the agent explore more, but too much exploration can hinder practicing the correct actions and slow down learning. This behaviour can be seen in Figure 5.b where the average training score remains below 0 and a convergence cannot be seen. However, when it comes to testing with epsilon set to 0, this risk taking Pacman has an average test score of +499.80 and a win rate of 1.00, making it the best performing agent. This is very likely the effect of the agent's exploration of many different state-action combinations and learning faster. This phenomenon also shows the importance of evaluating testing performance along with training scores.

With its discount factor set to 1, Pacman pays equal attention to all previous value estimations. This far sighted Pacman's learning might be impacted as very early, highly incorrect estimations are given the same importance as more recent ones. Adverse effects are not seen in Figure 5.c, however test scores are affected slightly. This Pacman agent has an average score of +496.20 and a win rate of 0.98 in test games, with epsilon set to zero.

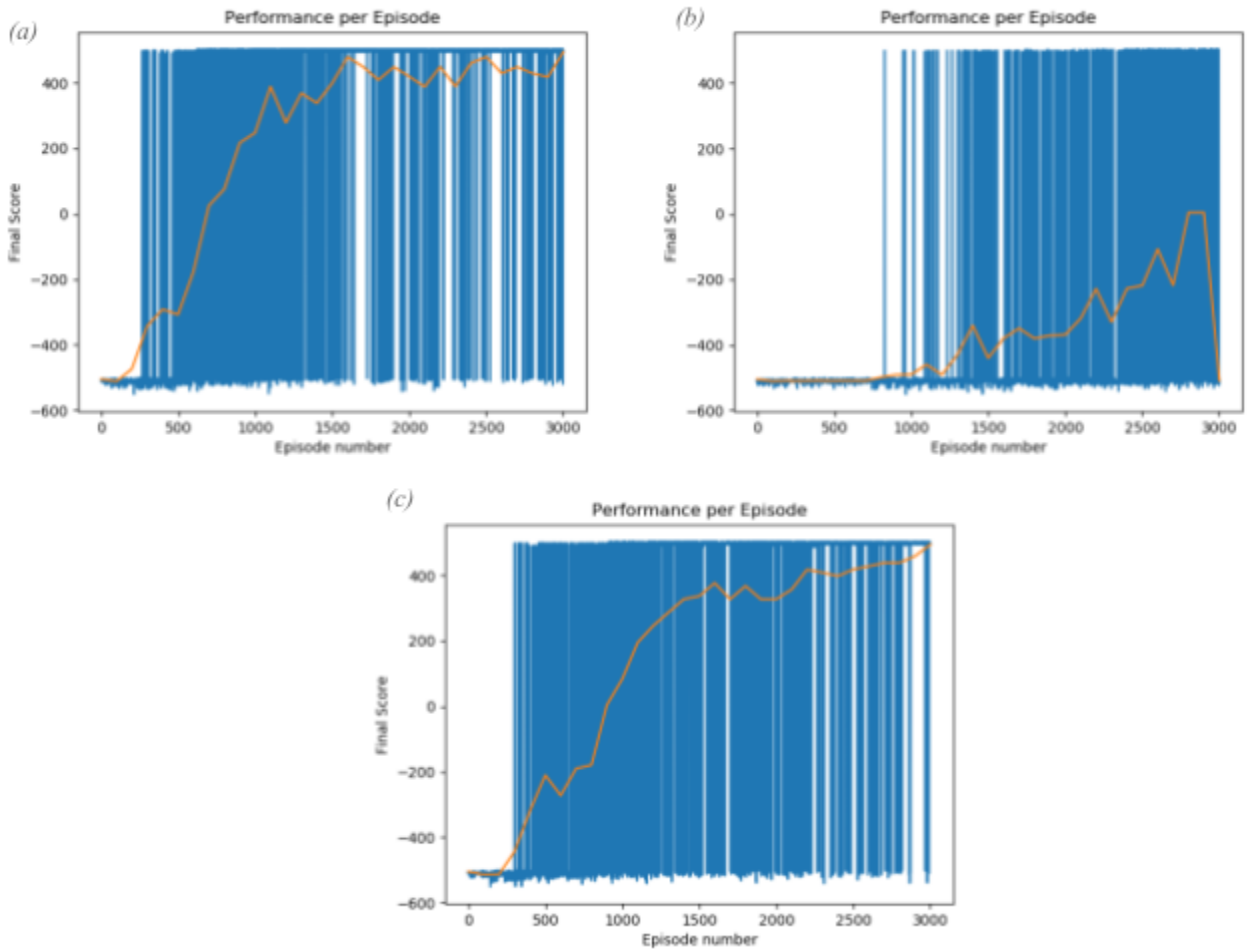


Figure 5: Training scores of fast learner Pacman (a), risk taker Pacman (b), and far sighted Pacman (c) on the small grid. Score per episode is shown in blue while average score per 100 episodes is shown in orange.

3.3.Deep Q-Learning Results

Interestingly, deep Q-learning Pacman's test performance is slightly worse for the small grid compared to simple Q-learning Pacman when trained for the same number of games. After 3000 training episodes, it has an average score of +491.30 and a win rate of 0.99. However, unlike the simple Q-learning agent, deep Q-learning Pacman can deal with larger layouts such as the medium grid, and to a lesser extent the small classic.

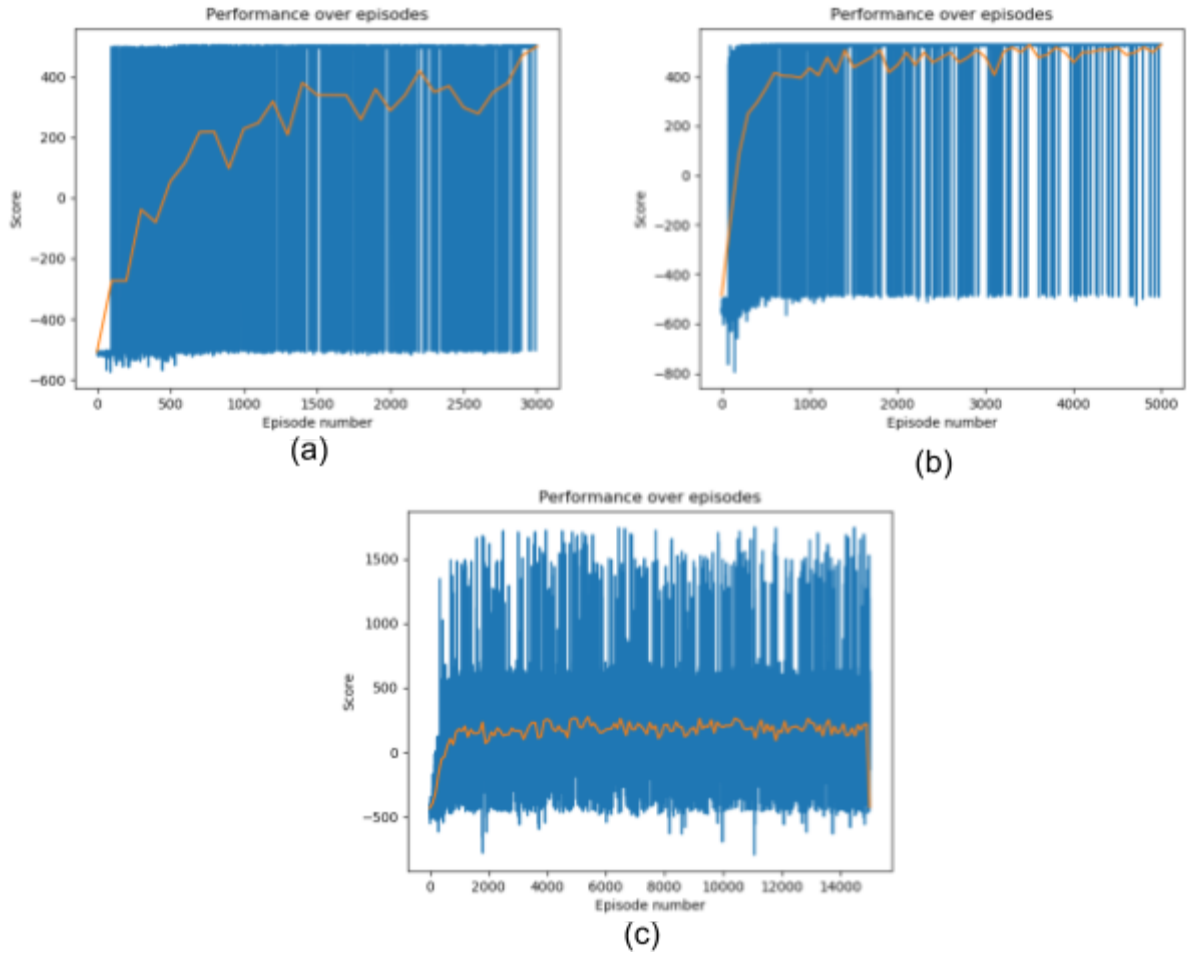


Figure 6: Training scores of deep Q-learning Pacman on the small grid (a), medium grid (b), and small classic layout (c). Score per episode is shown in blue while average score per 100 episodes is shown in orange.

Unlike the simple Pacman, deep Q-learning Pacman can reliably win games on the medium grid after 5000 episodes of training. With its epsilon set to 0, it has a win rate of 0.99 and an average test score of +517.94 in 100 test games. During training in this layout, the model seems to converge faster than it does with the small grid, which can be seen in figure 6.a and 6.b. Considering that this agent can handle the medium grid better than the small grid, and performs worse than the simple Q-learning agent in the small grid, the network used might be too large for the small grid problem.

More significantly, the deep Q-learning Pacman can win a few games on the small classic layout after 15000 training episodes. Its performance graph shows positive scores, however this is only because this layout has many food points which makes it possible to lose with a positive score, within a range of (0, 700). Winning scores can be seen in the region above +1000 score. It has a very low win rate of 0.16 and an average test score of 314.20 in 100 test games. Although low, this win rate is an improvement considering that simple Q-learning cannot even handle the medium grid. Training was stopped after 15000 episodes as performance per episode indicated the model was converging. For future work, a larger network might be able to handle this layout with a better performance.

Scores per episode and average scores per 100 episodes for all layouts can be seen in Figure 6. The small classic layout offers many food points to eat, which makes it possible to finish the game with a wide range of possible scores. Thus the score per episode graph for this layout shows the converging behaviour of the model more clearly than the small and medium grids.

References

Sutton, R.S. and Barto, A. (2018). *Reinforcement learning : an introduction*. 2nd ed. Cambridge, Ma ; London: The Mit Press.

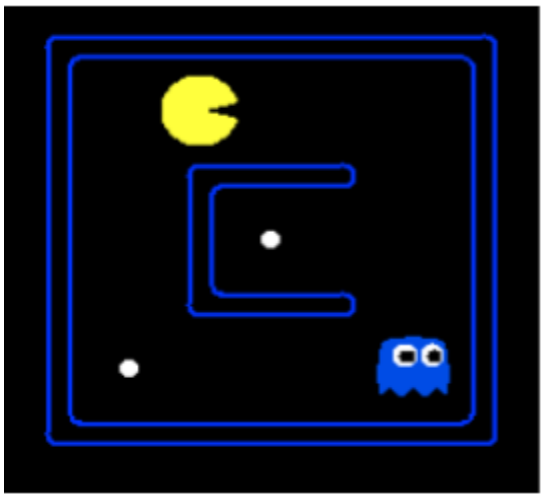
DeNero, J., Klein, D., Hay, N., Miller, B. and Abbeel, P., n.d. *Berkeley AI Materials*. [online] Ai.berkeley.edu. Available at: <http://ai.berkeley.edu/project_overview.html> [Accessed 20 May 2020].

Paszke, A., 2017. *Reinforcement Learning (DQN) Tutorial — Pytorch Tutorials 1.5.0 Documentation*. [online] Pytorch.org. Available at: <https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html> [Accessed 30 May 2020].

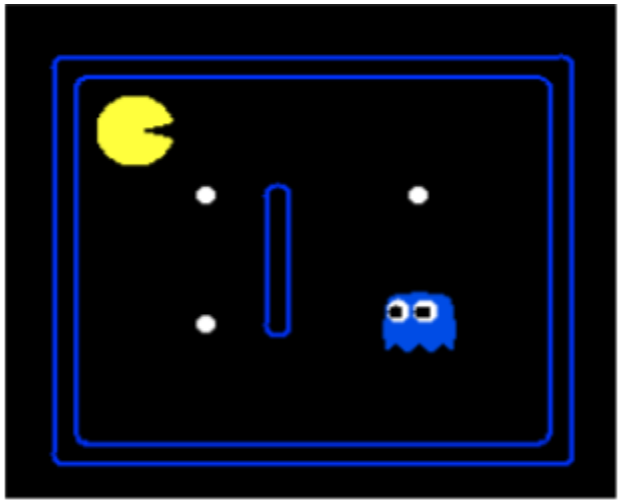
Spacco, J., 2017. *Jspacco/Pac3man*. [online] GitHub. Available at: <https://github.com/jspacco/pac3man> [Accessed 20 May 2020].

Appendix

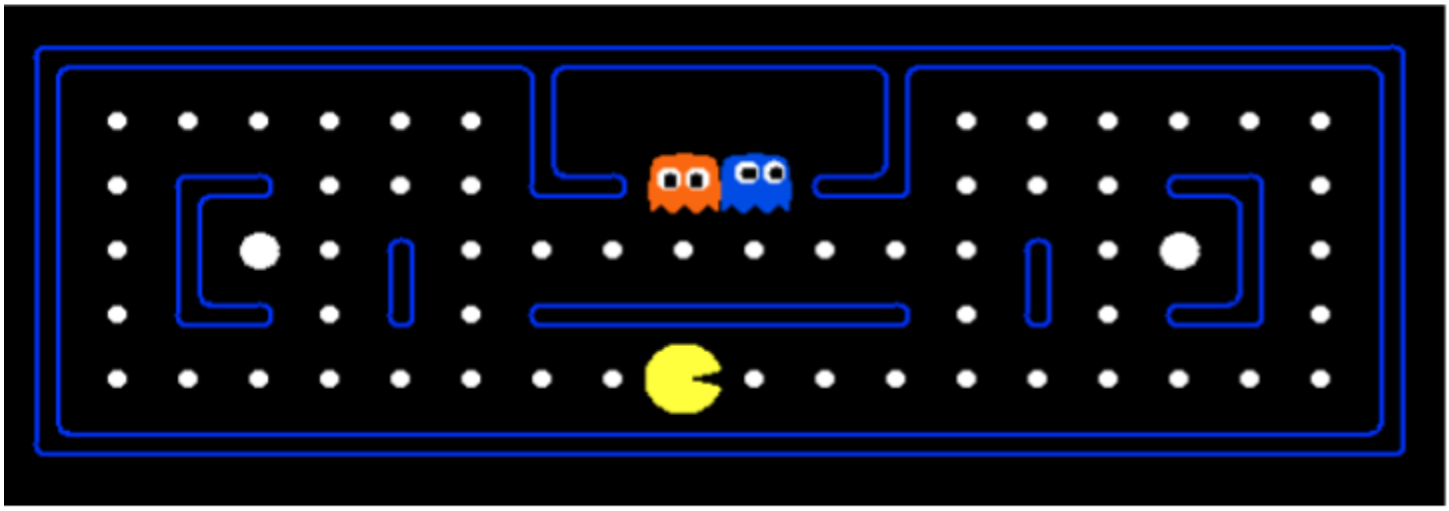
A.Game Layouts



Small Grid



Medium Grid



Small Classic