Playing MsPacman with Reinforcement Learning

Ayadi Aymen, Kaabachi Bayrem, Hojeily Woody, Ben Naceur Walid

***Abstract*—This project is done in the purpose of implementing and applying a Reinforcement learning algorithm, to play the arcade game MsPacman. Particularly, we will be developing the Q-learning model, in order to reach a policy that maximizes the expected value of the earned points. Our agent will learn to eat pellets and avoid ghosts, by making an optimal search based on the current state in the RAM of the game.**

I. Introduction

Reinforcement Learning is a technique that allows an agent to take actions and interact with an environment so as to maximize the total rewards. It covers the capability of learning from experience and is thus a powerful platform for learning to control agents in unknown environments, it also happens that we often require those capabilities while playing computer games. Computer games are often goal-oriented decision problems that rely on learning from experience and finding optimal strategies to overcome hardships. We therefore attempt to apply reinforcement learning to play the game MsPacman.

The player navigates MsPacman through a maze containing dots and trying to avoid ghosts. Score is accumulated by passing through said dots. In this project, we are trying to maximize the score.

Pacman is not a trivial game, the player should master secondary goals to achieve a good performance. The player needs to efficiently scouring a maze for pills, avoiding ghosts or eating them, strategically sacrificing a life to get a difficult-to-reach pellet, and so on – all to achieve an overall primary goal of completing the level. When using images as input other challenge is to understand the difference between pills and pellets, who are the ghosts and who is the pacman, In our case we relied on the RAM version of MSPacman to void this problem.

II. Background and Related Work

The notion of **value function** is of central interest in reinforcement learning tasks. Given a policy π, the value $V^\pi(\mathbf{s})$ is defined as the expected discounted returns obtained when starting from this state until the current episode terminates following policy π :

$$V^\pi(\mathbf{s}) = \mathbb{E}[\sum_{t=0}^{\infty} \gamma^t R(s_t) | s_0 = s, \pi]$$

The value function V must obey the Bellman's equation

$$V^\pi(s) = \mathbb{E}_\pi[R(s_t) + \gamma V^\pi(s_{t+1}) | s_t = s]$$

which expresses a relationship between the values of successive states in the same episode. In the same way, the state-action value function (Q-function), Q(s,a) denotes the expected cumulative reward as received by taking action a in state s and then following the policy π

$$Q^\pi(s,a) = \mathbb{E}_\pi[\sum_{t=0}^{\infty} \gamma^t R(s_t) | s_0 = s, \pi, a = a_0]$$

The objective of reinforcement learning problems is to estimate an optimal policy π^* by choosing actions that yields the optimal action-state value function Q^* :

$$\pi^*(s) = argmax_a(Q^*(s,a))$$

Learning a policy therefore means updating the Q-function to make it more accurate. Recall from the lecture in [2], the goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations. "Q" names the function that returns the reward used to provide the reinforcement and can be said to stand for the "quality" of an action taken in a given state.

Recall the algorithm for Q-Learning :

Algorithm 1 Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
States $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$
procedure QLearning($\mathcal{X}$, A, R, T, α, γ)
  Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
  while Q is not converged **do**
    Start in state $s \in \mathcal{X}$
    while s is not terminal **do**
      Calculate π according to Q and exploration strategy (e.g. $\pi(x) \leftarrow \text{argmax}_a\, Q(x, a)$)
      $a \leftarrow \pi(s)$
      $r \leftarrow R(s, a)$ ▷ Receive the reward
      $s' \leftarrow T(s, a)$ ▷ Receive the new state
      $Q(s', a) \leftarrow (1 - \alpha) \cdot Q(s, a) + \alpha \cdot (r + \gamma \cdot \max_{a'} Q(s', a'))$
      $s \leftarrow s'$
  return Q

The agent we chose follows a variant of Q-learning : the Deep Q-Learning algorithm with experience replay, based on deep conventional neural networks that we will describe further on.

Previous Related Works

It also happens that the problem of solving Atari games is quite popular within the Reinforcement Learning research field. The use of the Atari 2600 emulator as a reinforcement learning platform was introduced by [5], who applied standard reinforcement learning algorithms with linear function approximation and generic visual features. And we also could see when Q-learning was combined with experience replay and neural networks in [6]. Some work has also been made on MsPacman through the intermediary of the MASON platform, where speeding up the learning process without the necessity of Q-function approximation, in order to reach optimum generic policies. [4]

III. The Environment

MsPacman is available as a standard OpenAI Gym environment. The gameplay is as follows : The player navigates MsPacman through a maze containing dots, known as Pac-Dots, and four multi-colored ghosts: Blinky, Pinky, Inky and Clyde. The goal of the game is to accumulate as many points as possible by collecting the dots and eating ghosts. When all of the dots in a stage are eaten, that stage is completed. The four ghosts roam the maze and chase MsPacman. If MsPacman comes into contact with a ghost, she loses a life. The game ends when all lives have been lost. Near the corners of the maze are four flashing Power Pellets that provide MsPacman with the temporary ability to eat the ghosts and earn bonus points.

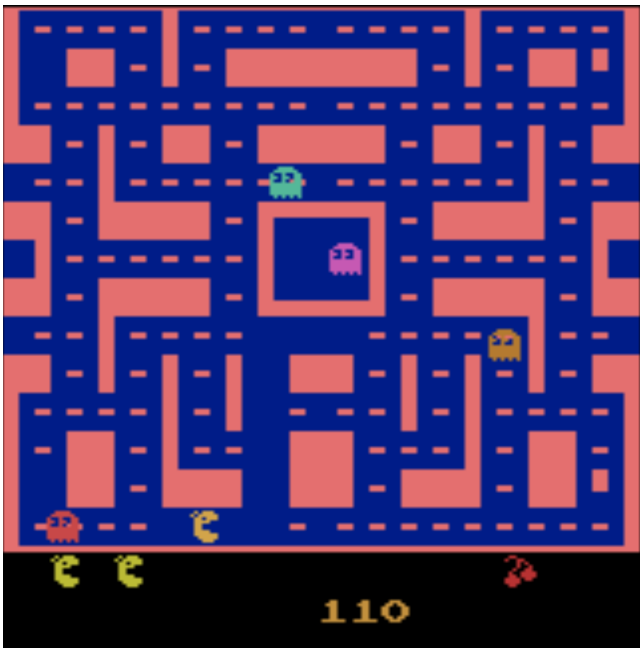

Figure 1: A scene of the game.

The game offers a discrete action space : the four directions in which MsPacman can move UP,DOWN,LEFT,RIGHT. Although the domain is discrete it has a very large state space.

We use the RAM version of MSPacman for several reasons.: -If we train from the pixels, we'll likely use a convolutional net of several layers. interestingly, the final output of the convnet is a a 1D array of features. These we pass to a fully connected layer and maybe output the correct 'action' based on the features the convnet recognized in the image(es). Or we might use a reinforcement layer working on the 1D array of features. -RAM is only 256 bytes array and we know it contains info about the "state" of the game. It's more likely a better representation than pixels since it has info about the velocity of ghosts, their direction etc whereas inferring this information from pixels is hard.

The stat epsca is then dictated by thos e56 bytes of ram.2 Computers can't play this game well, since there are just too many possible game states to consider – 1293 configurations. It's not hard for an AI to find its way through a maze, but couple that with grabbing pills, dodging or eating ghosts, and collecting fruit for a high score, and it becomes hard even for an artificial brain. The electronic player has to appreciate and master secondary goals – efficiently scouring a maze for pills, avoiding ghosts or eating them, strategically sacrificing a life to get a difficult-to-reach pellet, and so on – all to achieve an overall primary goal of completing the level.

The environment is difficult to predict, because the ghost behaviour is stochastic and their paths are unpredictable. Hence, the main challenge this environment poses for an agent lies especially in its evolution within a random dynamic environment and that it involves sequential decision making.

The reward function is defined as follows: Each time MsPacman successfully eats a dot, the agent will immediately receive +10 as reward, and if MsPacman comes into contact with a ghost, the reward returned is -100 and the game ends, otherwise reward is 0.

IV. The Agent

As seen in [2], the goal of Q-learning is to learn a policy, which tells an agent what action to take under what circumstances. It does not require a model (hence the connotation "model-free") of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations. The agent we chose follows a variant of Q-learning : the Deep Q-Learning algorithm with experience replay. There are several advantages of experience replay: It allows a more efficient use of previous experience, by learning with it multiple times. This is key when gaining real-world experience is costly, you can get full use of it. The Q-learning updates are incremental and do not converge quickly, so multiple passes with the same data is beneficial, especially when there is low variance in immediate outcomes (reward, next state) given the same state, action pair. Also, experience replay leads to a better convergence behaviour when training a function approximator. Partly this is because the data is more like i.i.d. data assumed in most supervised learning convergence proofs. We use an epsilon-greedy approach to choose the action. We either choose an action at random (for the first 10000 frames or with a probability epsilon), or we choose the one which will maximize the reward, based on the Q table we build during learning. The epsilon decreases each episode by a set rate, since we consider that after exploring 'enough' our agent doesn't need to continue exploring anymore.

Since training samples in typical RL setup are highly correlated, and less data-efficient, it will leads to harder convergence for the network. A way to solve the sample distribution problem is adopting experience replay. Essentially, the sample transitions are stored, which will then be randomly selected from the "transition pool" to update the knowledge.

The experience replay comes into action when we update the weights.

Deep QLearning description

A Q-learning agent does not have the ability to estimate value for unseen states. To deal with this problem, DQL introduces Neural Networks. DQL leverages a Neural Network to estimate the Q-value function. The input for the network is the current, while the output is the corresponding Q-value for each of the action. An illustration of the network's architecture is shown in figure 2.

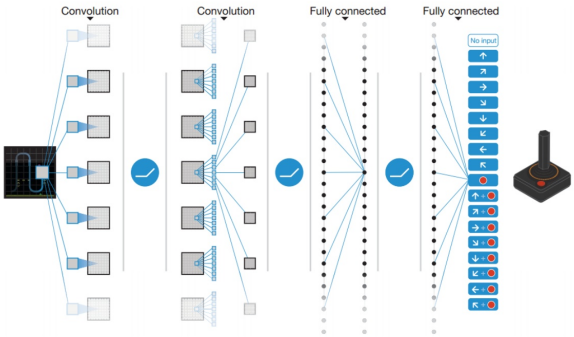

Figure 2: DQL's neural network architecture

We train the network based on the Q-learning update equation. Recall that the target Q-value for Q-learning is:

$$r_j + \gamma \, max_{a'} Q(\phi_{j+1,a';\theta^-})$$

The ϕ is equivalent to the state s, while the θ stands for the parameters in the Neural Network. Thus, the loss function for the network is defined as the Squared Error between target Q-value and the Q-value output from the network. The pseudo code for the deep Q-learning is given below.

Algorithm 2 deep Q-learning with experience replay

Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target-value function Q* with weights $\theta^- = \theta$
for episode = 1,M **do**
  Initialize sequence s_1=x_1 and preprocessed sequence ϕ_1 = $\phi(s_1)$
  for t=1,T **do**
    With probability ϵ select a random action a_t
    Otherwise select a_t=$argmax_a$Q($\phi(s_t), a; \theta$)
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set s_{t+1}=s_t, a_t, x_{t+1} and preprocess ϕ_{t+1} = $\phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    If episode terminates at step j+1
    Set y_j=r_j
    Else
    Set y_j=$r_j + \gamma \, max_{a'} Q(\phi_{j+1,a';\theta^-})$
    Perform a gradient descent step on (y_j - Q($\phi_j, a_j; \theta$))^2 with respect to the network parameters θ
    Every C step reset Q* = Q
    End For
    End for

QLearning parameters

After some tuning we chose to initialize the exploration rate (epsilon) at 0.1, the minimum is set to 0.0001. The discount factor is set to gamma = 0.95. For the Gradient Descent step, the learning rate alpha is fixed at 1e 5, while the sample batch-size is 32. Our program will also start training Neural Network after observing the first 10000 frames.

V. How to run the code, libraries used for the project

A. The code

The code can be found here [3] :

To run the program and train a new network it is sufficient to open the jupyter notebook file and run all the cells.

B. Libraries used for this project

The following packages are required to run our program : • Python3 • Tensorflow • keras • keras-vis • gym • pygame • Pygame-Learning-Environment (ple) • gym-ple

VI. Results and Discussion

In order to put the results into perspective, the four members of our team tried playing the game for about 2 hours to get an average score of 8250 points. Those can be considered to be "regular" human scores. With our developed agent, we achieved the following results:

In about 200 episodes, it reached a maximum score of 1750 points.

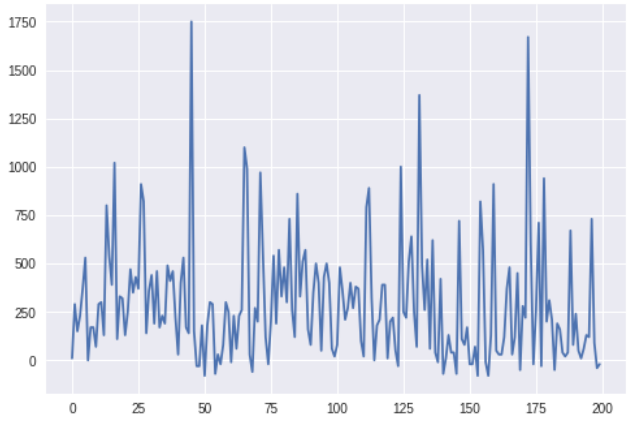

Figure 3: Score graph of the first 200 episodes

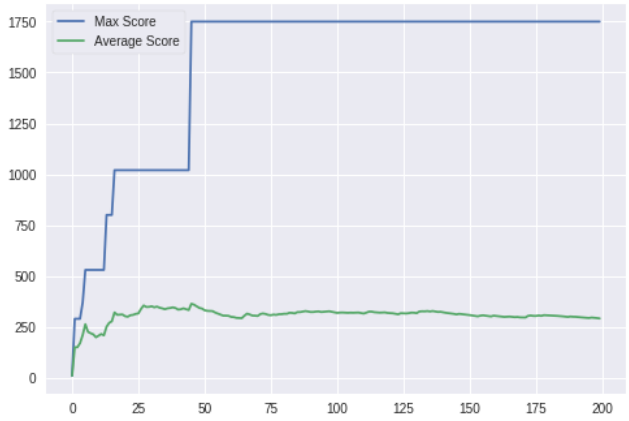

Figure 4: The maximum and average scores a 200 episodes training.

And in about 5000 episodes, MsPacman reached a maximum score of 3270 points.

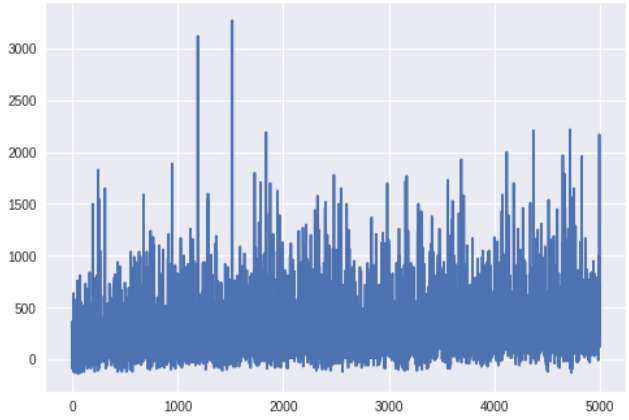

Figure 5: Score graph of the first 5000 episodes

We can see on the figures that the number of eaten dots by MsPacman is constantly increasing. Though it plays better than a lot of people, in general it does not surpass human performance. This is understandable because our employed

Figure 6: The maximum and average scores during a 5000 episodes training.

Neural Network is still small due to the lack of a powerful machine to run it. We judge human performance by the results obtained in DNADRL where they depict that the human average score is 6,951.6. Similar scores are seen in other papers in the literature.

VII. Conclusion and Future Work

This work demonstrates how a Deep Q-Network is capable of playing the arcade game MsPacman and achieving high results even though it is not enough to top the average human score depicted in DNADRL[7] As we enlarge the system, it is expected to outperform human experts in this game. Beside the size aspect, another problem is that the results attained through training seem very inconsistent. As the agent plays more and more, it can get higher scores, but still commit stupid errors, which shows our method is still subdue to a high variance. The key factor may be in reward function definition part as it may require a more intelligent and consistent use of elements of the game state to collect more valuable experiences to replay.

References

[1] PyGame Learning Environment http://pygame-learning-environment.readthedocs.io/en/latest/index.htm
[2] J. Read. Lecture VI - Topics in Reinforcement Learning. *INF581 Advanced Topics in Artificial Intelligence*, 2019.
[3] Repository where our code is stored goo.gl/C6ymLs
[4] Tziortziotis,Tziortziotis, Blekas Play Ms. Pac-Man using an advanced reinforcement learning agent http://www.lix.polytechnique.fr/~ntziortziotis/pubs/SETN14.pdf?fbclid=IwAR1EPVjP8aalZx1NH-M_Shu6ggIUGEMirQWEt9zl3y8DZCFU4swPRx7EonE, 2017.
[5] L.P. Kaelbling, M.L. Littman, and A.W. Moore. Reinforcement learning: A survey. Journal of Artificial Inteligence Research, 4:237–285, 1996
[6] Long-Ji Lin. Reinforcement learning for robots using neural networks. Technical report, DTIC Document, 1993.
[7] Dueling Network Architectures for Deep Reinforcement Learning Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt
[8] Playing Atari with Deep Reinforcement Learning Mnih et a .l