# MS-Pacman\_v0 OpenAI GYM DQN-Learning

Author: Parsa Pourzargham

e-mail: [lum2ce@inf.elte.hu](mailto:lum2ce@inf.elte.hu)

Neptun ID: LUM2CE

## Introduction

The problem which this article tries to solve is an Atari game called Pacman, in which, the agent must learn how to maximize its reward in the environment provided by GYM python library. Agent must learn how to collect the maximum number of dots, while avoiding the moving ghosts in the closed maze. This game, as well as other Atari games provided by GYM library, are good benchmarks on which different reinforcement algorithms can be trained and compared.

Convolutional neural network is utilized to train the agent which maps the problems input state to actions. The motivation of applying the DQN algorithm on this problem is to see how good this algorithm can perform in this specific task. Also, to find out how much benefit the model gets by utilizing the neural networks alongside with the Q-Learning, compared to more basic solutions like classical Q-Learning [[1](#_Sutton,_Richard_S.,)].

The input to the algorithm is the in-game grayscale screen images, fed to the neural network that output the probability of each action also known as Q-values using the DQN algorithm.

## Related work

[[2](#_van_der_Ouderaa,)] uses a different implementation of Pacman game environment in OpenCV library, but quite interesting experiments and techniques was applied. Some experiments were carried out to see the performance of the model in an unseen environment, namely increasing the number of the ghosts which leads to an unseen testing set for the agent. The performance of the model was expectedly decreased over the number of the ghosts. The model, however, could generalize quite fine in terms of adding new ghosts. The results of another carried experiment demonstrated that the performance of the model was near random when a new map is given to the agent and the model failed to generalize well.

[[3](#_Spalding,_Holt,_Oliver)] did a quite clever approach by utilizing a method called character-centric cropping in Pacman game which leads to significantly faster model in terms of runtime. According to the paper, the model fed by cropped images centered around the agent can achieve quite same performance while decreasing the runtime duration by the factor of 6. As the main image was cropped to the Region of interest, which is a relatively smaller image than the game map image, the runtime of the algorithm could be reduced from approximately 4 hours to 40 minutes which is a quite noticeable improvement.

An interesting idea was implemented by [[4](#_Kelvin,_Mak_Jeffrey,)] which records the direction of agent in the learning process and compared the directions with the actions a human would perform in real-life game situation. According to the paper, the model choses the same action a human would do in the same situation, despite taking the agent far from the seeds it should normally follow.

Three different grid-size were tested in [[5](#_Gnanasekaran,_Abeynaya,_Jordi)] to compare the performance of Q-learning, DQN and approximate Q-learning on each grid size. Although Q-learning performed well in small size grid game, it failed to achieve a good result in bigger space input grid. Also, approximate Q-learning was utilized to use some hand-crafted features to boost the performance of the agent. Some features turned out to help the model like the number of active ghosts or the distance to the closest food (seed).

DQN can also be mixed with other methods like Object sensitive Deep Reinforcement Learning (ODRL) and enhance the model performance by encoding object channels, which results to a method called O-DQN [[6](#_Li,_Yuezhang,_Katia)].

## Problem description

In the game, Pacman is the agent which must collect as much seed as possible while avoiding 4 ghosts moving to catch it in the closed maze field. The agent has 9 possible actions to take in each step which include moving up, moving down, moving right, moving left, moving diagonally, and staying where it is.

The environment for the problem is provided by the GYM library in python programming language. It takes the action from the agent and returns the new state of the environment after performing the action, as well as the immediate reward of the taken action as a feedback to the agent.

Qr code

Description automatically generated

Game board

## Methods

The algorithm used to train the agent is Deep-Q-Learning, DQN in short. DQN is also a reinforcement algorithm which is derived from the classical Q-Learning algorithm. Unlike classical Q-learning algorithm which maps each state-action pair to its corresponding Q-Value using a table called Q-table, DQN maps the state-action pairs to their corresponding Q-value using a deep neural network.

The overall workflow of DQN algorithm is as following:

1. The neural network takes the in-game screen of the game as the input and calculate the tensor of probabilities of actions. To obtain a tensor of probabilities, the activation function of the las layer (fully connected layer with output number of the action space size) is set to None.
2. As the networks take the current state of the environment to give the probability tensor of actions, a grayscale image of the game is taken and fed into the network to calculate the corresponding Q-values.
3. After obtaining the probability tensor, the action with the highest probability is chosen among all possible actions. That is delivered by the function argmax ().
4. Having chosen the action in the previous step, the action is given to GYM and it provides us with the new state of the environment after taking the action, as well as the immediate reward corresponding to the taken action and whether the game is finished or not.
5. After obtaining the above-mentioned parameters from the environment, the parameters are stored in a buffer in the format of (s,a,r,s’,d), where s denotes the state before taking the action, a denote the taken action, s’ denotes the new state after taking the action and d stands for the Boolean value which decides whether the game is done or not.
6. After finishing the step 5, the algorithm needs to calculate the loss to train the network. r’ and a’ are required to calculate the loss value. a’ could be obtained by feeding s’ to the network, while r’ is discounted from r. Having r’ and a’, loss can be calculated. Y values needed for the loss calculation can be obtained by the following bellman equation [[1](#_Sutton,_Richard_S.,)]:

+λ\*

1. Finally, when the training process is finished, the network can be used by the agent to play the game and evaluate the score of the agent. This is done by feeding the current state to the network and choose the action according to the probability tensor provided by the network.

Remarks:

* Updating the model weights frequently in every step may result to low performance of the model as it updates the weights when nothing much has changed. As a result, the model weights are updated every n step which is experimented in the experiment section to obtain the optimal n.
* Two networks are used to train the agent in this algorithm. Although both have the same architecture, they have different set of weights in the model. Two networks are called main network and target network. Main network weights are updated after each n steps in the training process, while the main network weights are copied to the target network every 100 steps. The frequency of copying the main network weights to the target network is also experimented in the experiment part. The correct frequency of copying the main network weight to the target network is obtained by trial and error to 100 steps by [[7](#_Github,Pacman_code,_https://github.)]. Having these two different networks is beneficial in terms of stability in the learning process and helps the model to learn more effectively.
* The environment in the game is fully observable but the gameplay of the ghosts is not deterministic and varies from episode to another and the performance of the model cannot be compared with another model by first look. It is possible that an optimal model achieves low scores in some episodes, but its overall performance is better that others. Hence, apart from the rewards achieved by the agent in all the episodes, the average reward of the agent, as well as the maximum achieved reward and the number of high scores (higher than a certain threshold) is also calculated to compare different models with different hyper parameters meaningfully. The threshold is set to 600.

## Model architecture

The code for the model is taken from [[7](#_Github,Pacman_code,_https://github.)] in the IPython file format and it is executed on Google Colab service. The original model uses 3 convolutional layers with kernel size of (8,8), (4,4) and (3,3) with strides 4,2 and 1 respectively. As the input image to the model is relatively small, no pooling layer is added to the model.

## Experiments and Discussion

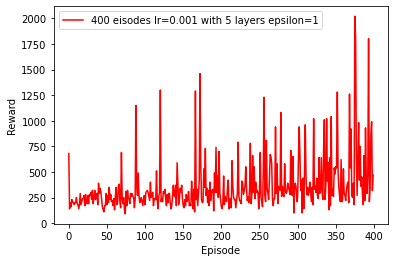
In code [[7](#_Github,Pacman_code,_https://github.)], learning rate is set to 0.001 and the epsilon value is 0.5. The number of episodes initially set to 400 episodes. With the default hyperparameters, the model could achieve the average reward of 350.9 and 48 number of scores were higher than the threshold.

## Experiments

In the following experiments, hyper parameter tuning was performed to improve the model performance.

1. **Increasing learning capacity of the network**

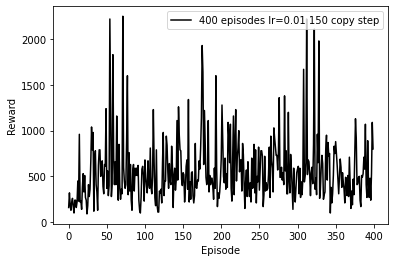
Two additional convolutional layers were added to the model to increase the capacity of learning in the model. Additionally, the filter size was modified in a way that layers have the kernel size (5,5), (4,4), (3,3), (3,3) and (2,2) from the top layer to the last convolutional layer. Due to the fact that adding convolutional layers with strides larger than 1 decreases the input image dimensions, the stride size was also decreased to 2 for the first two layers and 1 for the next three layers. The epsilon number is also set to 1 to force the agent to do more explorations at the beginning which leads to more exploration and hence, learning more in the early stages. After training the model with the mentioned hyperparameters, the model did not show a noticeable improvement despite having higher capacity than the default code. The average reward of the model was 357.125 and the agent could only achieve 48 high scores. In testing section, the trained agent achieved 350 scores in real- game testing stage.



The rewards have a fluctuating behavior over the 400 episodes. As mentioned earlier, two parameters were calculated for comparison of the model performance with other models, namely average reward, and the number of high scores. The model with above-mentioned hyperparameters could achieved approximately 357 reward on average over the 400 episodes and only 48 high scores, which is not promising compared to the base model.

1. **Improving the model by finding the optimal copy frequency and buffer size**

To force the agent to do more explorations in early stages, the epsilon value was set to 1. Also decay rate for epsilon was set to 50000 to slow down the decrease rate of epsilon. The size of the buffer was also increased to 40000 to train over a bigger set of data. Learning rate was set to 0.01 as there was a limitation in computation power and default learning rate needed many more iterations for convergence. In addition, copy-step was changed to 150 and step train decreased to 2 in order to update the target network a bit less often and update the main network more often. By these modifications, the performance of the model was noticeably improved as the agent’s average reward reached 562.275 and the number of high scores reached 141 which shows a good improvement compared to the base model. At the end, the trained agent could achieve the score of 1640 in the real-game testing stage. The IPython file of this model is available with the uploaded files (“best model with 141 high score.ipynb”).



The plot demonstrates that the model could improve its performance as the rewards of the model are centered much higher compared to the base model. Many points are above the threshold which means the agent could achieve more high scores.

## Conclusion, further work

This study evaluates the effectiveness of DQN algorithm on Pacman Atari game. Some experiments were carried out on DQN model to improve its performance. Adding more layers to the network did not play a key role in improving the model performance. In contrast, forcing the agent to learn more by setting the initial value of the epsilon to 1, as well as decreasing the epsilon decay value by the factor of 10 to force the agent to learn more and setting a much bigger buffer could increase the model performance. This idea enhanced the overall performance of the model and the test score of the trained agent.

For future work, in order to get the optimal policy, the hyper parameters could be obtained via neural network. This requires a good deal of computational power. As a result, the optimal value of hyperparameters were found by trial and error and some basic justifications of how the model is trained.

## References

#### Sutton, Richard S., and Andrew G. Barto. Introduction to reinforcement learning. Vol. 135. Cambridge: MIT press, 1998.

#### van der Ouderaa, Tycho. "Deep reinforcement learning in pac-man." Bachelor Thesis, University of Amsterdam (2016).

#### Spalding, Holt, Oliver Newland, and Jivko Sinapov. "Investigation Of The Effects Of Character-centric Cropping On Pacman DQN Performance." (2018).

#### Kelvin, Mak Jeffrey, and Dirk Schneiders. "Learning to Play Computer Games with Deep Learning and Reinforcement Learning." (2018).

#### Gnanasekaran, Abeynaya, Jordi Feliu Faba, and Jing An. "Reinforcement Learning in Pacman." (2017): 1-6.

#### Li, Yuezhang, Katia Sycara, and Rahul Iyer. "Object-sensitive deep reinforcement learning." arXiv preprint arXiv:1809.06064 (2018).

#### Github,Pacman code, https://github.com/EXJUSTICE/Deep\_Qlearning\_OpenAI\_MissPacman