# Playing Pac-Man using Reinforcement Learning

**K Vijaychandra Reddy[1]**

[1]*UG student, Dept. Of CSE, Mahatma Gandhi Institute of Technology, Telangana, India.*

---

**Abstract –** *DeepMind published the first version of the Deep Q-Network (DQN) in 2013 which was an algorithm capable of human-level performance on a number of classic Atari 2600 games. DQN, and other similar algorithms like AlphaGO and TRPO, fall under the category of Reinforcement Learning (RL), which is a subset of machine learning. In years since, much advancement has been made in the field which enabled algorithms to increase their performance and solve games faster than ever before. I have been working to implement these advancements in Keras. In this paper I'll be implementing and sharing results of how the advancements like DQN, Double Q-Learning and Dueling architecture algorithms work in learning and mastering the Atari 2600 game, Ms. Pac-man.*

***Keywords-***Convolution, Deep Q-Networks, Convolutional Neural Networks, Reinforcement Learning, Tensorflow, Keras

## I. INTRODUCTION

Reinforcement Learning is a subset of machine learning in which an agent exists within an environment and looks to maximize a reward which is defined by the programmer. It proceeds to take actions, which change the environment and feeds it the rewards that are associated with those specific changes. The agent, then, continues to analyze its new state and settles on its next actions, repeating the process endless or until the environment terminates. This decision-making loop is known as Markov decision process (MDP).

Because of the simplicity of the controls, actions and objectives of Atari 2600 games like Ms. Pac-man fit into the MDP framework compared to later titles of consoles like NES and SNES. A player observes the position of Ms. Pac-Man on the game screen and presses buttons on the controller to save it from the ghosts and make it consume as many pellets as it can. The process of making instant decisions based on one of the nearly infinite pixel combinations that show up on the screen at any given time, and one which one had very unlikely to have ever encountered before can be simple for human beings but is daunting for the machine learning agent. To make matters more complex, video games are technically partially observable MDPs, in the sense that we are forced to make choices based on an indirect representation of the game which is on a screen instead of getting the output from the code or memory itself, which might hide some of the information we need to make a fully-informed decision.


**Fig 1:** Ms. Pac-man game on the Atari 2600 console

Games like Ms. Pac-man are very simple to control using the controllers. The finite number of buttons and joystick positions help us map the observation space to a discrete and manageable action space, hence making them an ideal candidate to practice enforcing reinforcement learning techniques at a lower processing cost.

## II. PREPROCESSING FRAMES

The preprocessing network contains 3 convolutional layers. Their sizes are 32*8*8, 64*4*4, 64*3*3 with strides 4, 2 and 1 respectively. They all have ReLU activations. Then there is a dense layer with 512 neurons

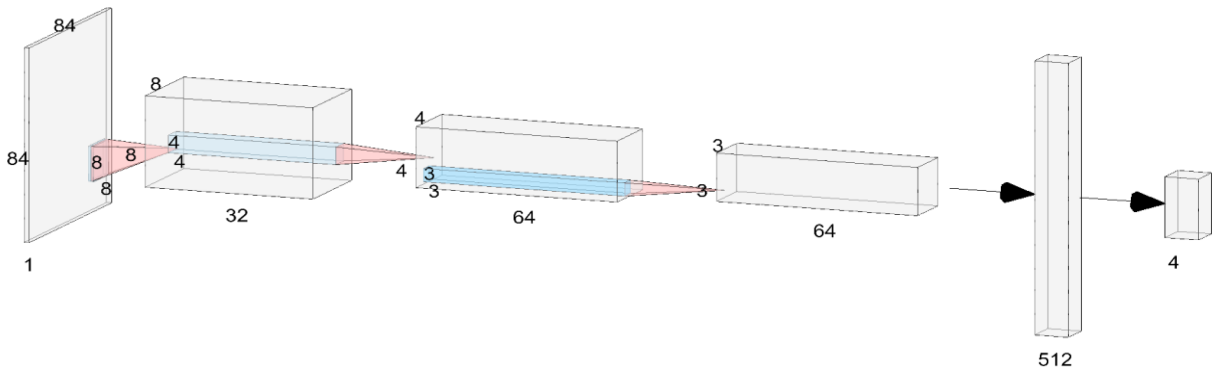


**Fig 2:** Preprocessing Network Model

and then the output layer with number of neurons being number of actions. While the shape of input image of the preprocessing network is 84*84 and the image has all 3 colors, the image from output layer is a heavily down sampled image with only white and black colors.

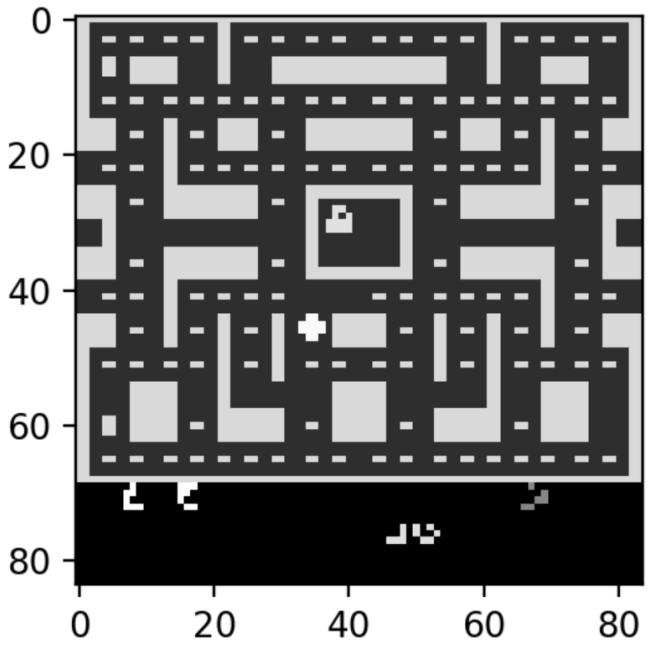


**Fig 3:** Processed Image from Pre-processing Model

**DQN MODEL**

In this paper, the objective is to train the Ms. Pac-man agent to learn playing the game by avoiding the ghosts and eating the food and scared ghosts as much as possible (i.e., to get higher scores)

The convolutional layers of the DQN will learn to detect the increasingly abstract features of the game's input screen. The dense classifier will then map the set of those features present in the current network iteration to the output layer containing a node for every controller combination.

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[\left(r+\gamma \max_{a'} Q(s',a';\theta_i^-) - Q(s,a;\theta_i)\right)^2\right]$$

After passing through the network the pixel images are mapped to Q values, which makes the neural network a Q function approximator making it able to predict the future states with enough training. Gradient descent is used in training which uses the above loss function, which in itself is a temporal difference alteration.

Essentially, this is the difference between the target Q values and the current estimation of them where the target value is the sum of immediate reward and Q value of the next state. Since the network has access to the first reward term, the accuracy of the overall expression is increased.

The DQN takes two measures to stabilize the constantly shifting dataset created during exploration. First, it duplicates the neural network, using the target network to generate the target values while the online network continues to generate the estimations. Only the online network is trained and the weights are updated at the target network. Second, an experience buffer is used. It a dataset where the past experiences of the agent are stored in the form of (s, a, r, t, s') where t is a Boolean that lets the agent know if that was the terminal state of the episode, and s' represents the state that followed s when the agent took action a.

The experience entry contains all of the variables needed to compute the loss function. So, instead of learning the

game during playtime, it's just moving the Pac-man agent around the screen based on what it learned before, but appending all of those experiences into the buffer. Then, the experiences are taken from storage and replayed to the agent so that it can update its parameters for state prediction accuracy. This ensures that the agent is learning from its entire history (or at least as much as the data capacity before it starts overwriting the oldest data), rather than only from its most recent trajectory.

Achieving balance between exploration and exploitation is very delicate task. Instead of taking only one pathway with the highest reward each time the agent also needs to explore new pathways in order to find more efficient ways to solve the problem. It also needs to be careful to not explore every pathway possible as there can be millions and billions of permutations possible depending on the complexity of the problem. A game like Ms. Pac-man can be complicated as there are up to four decisions possible for every pixel position present on the screen. The original DQN solves this by taking the Epsilon-greedy approach. This is done by initializing a variable, epsilon, to 1.0. For every step, a random number between 0 and 1 is generated. If this number is less than epsilon, an action is taken completely at random, regardless of what the agent calculates about that action's Q value. Initially, because the value of epsilon is 1, this is repeated 100% of the time. During training, the epsilon is decreased down to around 0.1, which basically means that the agent takes the action that was determined to be best 90% of the time and explores a new, random direction the other 10%.

In practice, the epsilon value is never let to 0. During testing/evaluation the value is kept as 0.05. This ensures that the agent can never get stuck in a corner or stop moving indefinitely.

I chose to run these comparison experiments for 10 million frames.

## III. DOUBLE DQN AND DUELING ARCHITECTURE

In 2015, van Hasselt et al. applied double q-learning to DQN, which had resulted in performance improvement in some games. In the loss function from above - the target network is used to both compute the q values for every action in the next state and to determine which of those actions the agent would want to take (i.e., the highest one). As it turns out, this can lead to some overestimation issues, sometimes particularly when there is an inconsistency between the target network and online network causing them to recommend different actions given the same state (instead of same action with slightly different q values). To solve this, it is proposed that the target network's responsibility to determine the best action is taken away. It would simply generate the Q values, and online model then can decide which to use. The Double DQN generates target values according to:

$$Y_t^{\text{DoubleDQN}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t), \boldsymbol{\theta}_t^-)$$

In RL theory the Q function can be split up to the sum of two independent terms: the value function V(s), which represents the value of the current state, and an advantage function A (s, a), which represents the relative importance of each action, and ensures that the agent takes the best feasible action, even if that choice may not have any immediately reflect on the game score.

In 2016, Wang et al. published the dueling network architecture, which works by splitting the neural network into two while sharing a convolutional base, one for estimating each function.

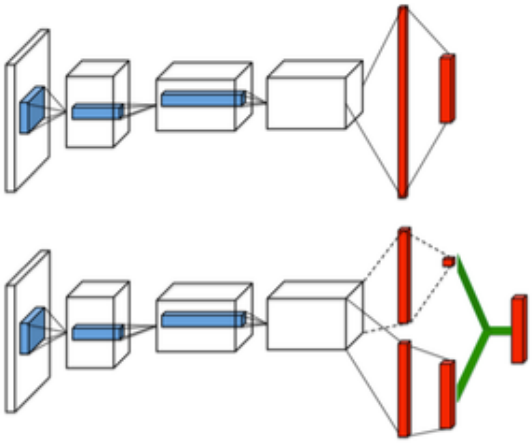

**Fig 4:** DQN architecture (top) vs. Dueling DQN (bottom). This computes the action and value functions separately, and then adds them up to arrive at the Q function.

The green arrows in the architecture schematic above are a generalization for the method used to combine them. To achieve that, we use the following approach:

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + \left( A(s, a; \theta, \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a'} A(s, a'; \theta, \alpha) \right)$$

Where alpha and beta are the parameters of the advantage streams and value streams. The network can use the understanding of its chosen action to change the

advantage function to 0, so that Q (s, a) is almost equal to V(s).

Schaul et al.'s 2016 paper proposed a solution, known as Prioritized Experience Replay (PER). In PER, an additional data structure is used that keeps a record of the priority of each transition. This means that the experiences are now sampled in proportion to their priorities:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

These priority weights are controlled by a hyper parameter beta, which controls how vigorously they need to be compensated. The beta value is typically incremented throughout the duration of the run, rising from an initial value of approximately 0.6 and ending at 1.0 (full compensation). Agents using PER usually have lower learning rates (typically around .25x) to prevent against catastrophic collapse.

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^\beta$$

## IV. BENCHMARKS

**Training Ms. Pac-man agent with DQN**

We analyze the agent's success by charting its cumulative reward at the end of every episode.

The reward function is proportional to and always less than the in-game score, therefore, not meaning much on its own. However, the agent had still learned something about the game since it consistently improved over time. Also, there is a diminishing return on investment.

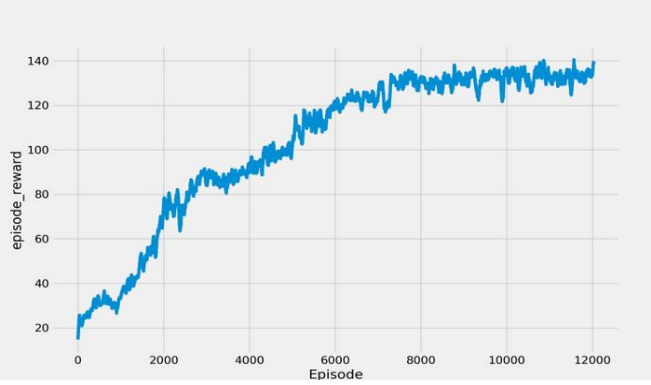


**Fig 5:** Episode cumulative reward against each episode using DQN

The agent had learned to navigate around the maze. It succeeds in clearing large areas when there are still dense pockets to be collected. However, the agent had a hard time going back to single dots that it had missed and it seems to get stuck whenever it finds itself in the middle of two pockets and has to decide which one to go after first.

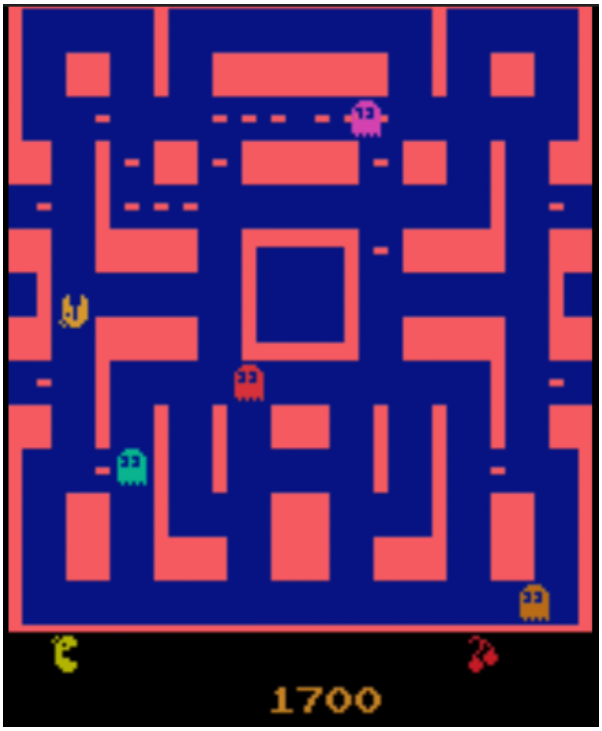


**Fig 6:** Pac-man agent does not find its way back to lonely dots and gets stuck in the middle unable to determine which way to go

**Training Pac-man with Prioritized Double Dueling**

Since the three algorithms are independent of each other, they can exist in the same algorithm. This can be referred to as a Prioritized Double Dueling DQN (PDD). Here is the learning curve, plotted against the previous version:

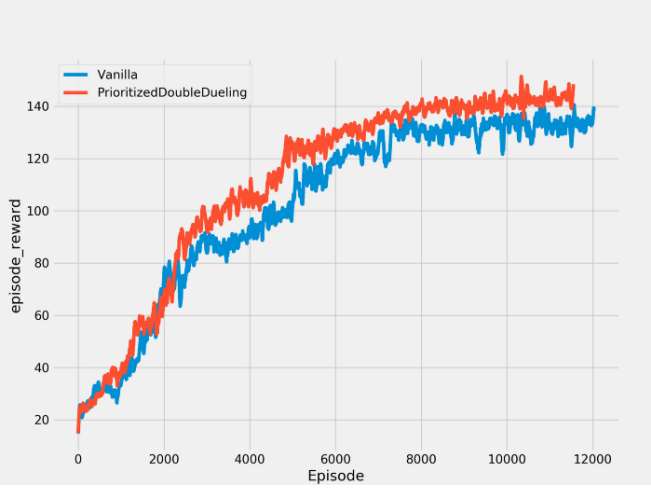


**Fig 7:** Comparison of DQN with Prioritized Double Dueling DQN

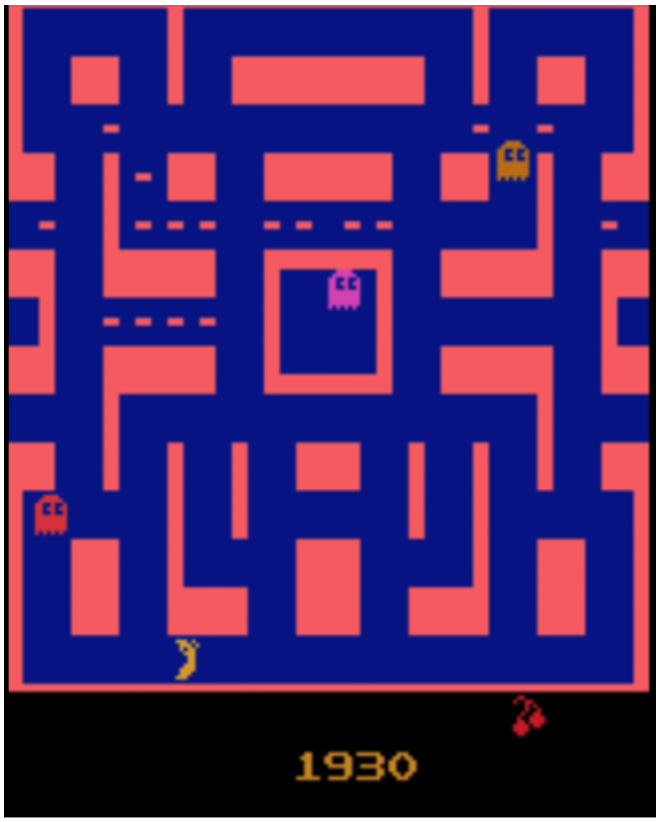


**Fig 8:** Pac-man agent successfully finds its way to the lonely dots and does not get stuck

While their performance is as poor as DQN during the first few thousand episodes, PDD begins to break away at the 2000th episode mark. And while that gap may seem small, analyzing the gameplay gives interesting results.

The new additions had allowed the agent to seek out isolated dashes successfully. In addition. it had also made the association between finding out the energy pills within the corners and earning a higher reward, possibly from accidentally running into ghosts once that behavior makes them vulnerable.

## V. REFERENCES

[1] Volodymyr Mnih et al., "Playing Atari with Deep Reinforcement Learning", Neural Information Processing Systems, 2013
[2] Volodymyr Mnih et al., "Human-level control through deep reinforcement learning", 2015
[3] Hado van Hasselt et al., "Deep Reinforcement Learning with Double Q-learning", Association for the Advancement of Artificial Intelligence, 2015
[4] Ziyu Wang et al., "Dueling Network Architectures for Deep Reinforcement Learning", 2016
[5] Tom Schaul et al., "Prioritized Experience Replay", 2016
[6] https://github.com/keras-rl/keras-rl