Deep Q Learning of Pac-Man

Lusine Kamikyan (Team member: Emma Reid)

December 10, 2018

Link to the video: **https://www.youtube.com/watch?v=Clj1P5y1NmA**

1 Introduction

In this project we applied Deep Reinforcement Learning to a game of Pac-Man. We extended an environment of Pac-Man with an agent , obstacles and ghosts (based on an online python code we found), created images to feed into the network to train it. We used Deep Q Learning with Convolutional Neural Networks as well as Double Deep Q Learning and Dueling Deep Q Learning. First, I will discuss some of the theoretical background, then how we extended the environment, how we tried to implement Deep RL algorithms on the game, the results, what and why things didn't work and conclusion and future work to be done.

2 Theory

2.1 Q Learning

Reinforcement learning is a branch of machine learning where an agent is learning how to navigate and learn the environment optimally by getting positive rewards for the right movements and negative for the wrong. It keeps getting feedback from the environment through rewards and observations. The goal of the agent is to maximize the rewards it gets. In this project one of the algorithms we have used is Deep Q learning which combines well known Q learning algorithm of RL with Neural Networks. The Q learning algorithm is based on the following recursive formula:

$$Q_{k+1}(s_t, a_t) = Q_k(s_t, a_t) + \alpha(r_{t+1} + \gamma max_a Q_k(s_{t+1}, a) - Q_k(s_t, a_t))$$

Here, $Q(s, a)$ is the value of the state given the agent is making action a. It is the expected total discounted return given the agent is in state s and is making action a. α is the learning rate, γ is the discount factor, shows how important are the future rewards. When implementing Q learning, a Q learning table is initialized for all the state and action combination values to be zero. Then as the algorithm runs, the value of each state given action a starts to converge to a value that will maximize the total rewards the agents gets.

2.2 Deep Q Learning

Deep Q learning combines the above algorithm with Neural Networks, that is instead of using a big table to predict the actions that will result in optimal rewards, Deep Q learning uses NN to predict the optimal policy. The Neural Network takes in the state of the agent as input, and outputs values of the state corresponding to the actions. Deep Q Learning incorporates the following methods to make the learning better:

- ϵ-greedy exploration - where the agent takes a random action with probability ϵ and with probability $1 - \epsilon$ it takes the action that maximizes the Q value of the state. ϵ-greedy policy helps the agent explore the environment at the beginning ($\epsilon = 1$), as time passes by, the agent becomes more and more greedy (in our case, ϵ gradually decreases to .02). This allows the agent discover states that will give it higher rewards than if it just followed a greedy policy.
- Experience Replay - this is a collection of experiences [current state s_t, reward r, action a_t, next state s_{t+1}] and has a fixed size, usually some large number N (in our case, N =10000). As the agent moves around in the environment, we keep collecting its experiences into the replay buffer. Then, we sample some number of these experiences from the replay buffer and feed into our Neural Network for training. This allows the agent not to forget previous experiences but also the data won't be correlated.
- Target Network - this network generates the target Q values. Once in every n steps (in our case n =100) we copy the weights from our training network to the target network. This helps with oscillations and makes the convergence better.

The target Q value is calculated using the following formula:

$$y = r + \gamma max_{a \in A} \hat{Q}(s_{t+1}, a; \theta^-)$$

where maximum is taken over actions that maximize the target network ($\hat{Q}$) Q values, θ^{-} are the weights of the target network.
The loss is calculated by the following formula:

$$L = \frac{1}{T}\sum_{t=0}^{T}(Q(s_t, a_t; \theta) - y)^2$$

2.3 Double DQN and Dueling DQN

2.3.1 Doubel DQN

Double DQN is basically the same as DQN except updating the target Q value:

$$y = r + \gamma\hat{Q}(s_{t+1}, argmax_a Q(s_{t+1}, a))$$

where we first find the action that maximizes the training Q value, then use that action to find the target Q value for the given state.
Double DQN solves the probelm of overestimation that DQN has.

2.3.2 Dueling DQN

Dueling DQN has a bit different network architecture than (dueling) DQN (see Figure 2). It separates the Q value of the state into 2 parts: the value of the state and the advantage of the action, i.e. how much better it is to make the action compared to the rest of the actions in the current state. Then those values are put together through the following formula and we obtain the Q value of the state:

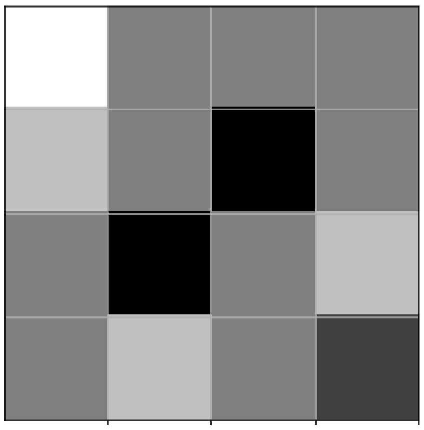
$$Q(s, a) = V(s) + A(s, a) - \frac{1}{|A|}\sum_{a} V(s, a)$$

This makes the training faster.

3 Environment

Our environment is a grid (we tried different size grids). It contains an agent, Pac-Man, obstacles and a ghost. It was build based on an online code that we have found. The original code was a 4 by 4 with 2 obstacles and 1 goal. It uses tkinter to create the environment and move the agent. We extended it to more goals, the agent would be able to actually pick the goals up as it moves on them, we added a ghost which will terminate the game as the agent moves on it. The agent can't move into the walls or outside of the grid, if it tries, it is given a negative reward. Pac-Man also gets positive rewards when it picks the goals up, gets small negative rewards when moves into an empty space (this will help it find the shortest path between the goals), moving onto the ghost makes it get negative reward and terminate the game. If the agent collects all the goals, then the game terminates, the agent gets an extra positive rewards and wins the game.

Originally, we thought that we could use the images from the tkinter environment directly to feed into our network. Thus, we were able to open the tkinter window in a specific part of the screen and capture a screenshot of the image. However, later we realized that taking the images directly was very time consuming and would make the training last much longer than it already did. So we decided, to instead create equivalent images to the environment using numpy arrays. This made the training go much faster than it could have been. In this way, each section of the grid is a pixel, so if we have a 6 by 6 grid, we get an image with 6 pixels by 6 pixels.



4 Network architecture and my contribution

The following are the arcitecture of the DQN (Double DQN) and Dueling DQN:

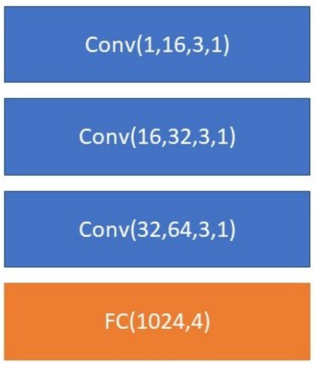


Figure 1: DQN and Double DQN

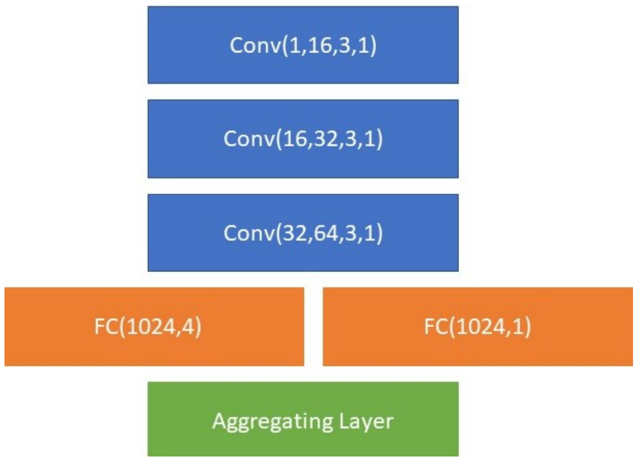


Figure 2: Dueling DQN

The following are my contributions to the project:

- Extension of the environment (together with Emma and Emma also did the code for creating the numpy images to feed into the network)
- Most of DQN code (Emma helped to finish it up (some part of the code in the main file))
- Double DQN and Dueling DQN
- Debug and run the code for different parameter values

5 Results

The training takes a very long time. From some of the sources we learned that depending on the complexity of the environment we might need about 250.000 to more than a million episodes. We didn't have GPU, only our laptops, so we run as long as we could.
Figure 3,4,5,6 show the results for a 4 by 4 grid with 2 obstacles, 1 ghost and 3 goals. Figure 1 is the first run for 2500 episodes, after which we saved the model and run it again for another 1500 (so we can have longer period). As we can see there is decrease in the loss in both of them. Since the agents didn't have enough time to learn, we don't get good rewards, they are all over the place, would be positive sometimes, however, not enough time and good experiences that would allow the rewards to converge to optimal amount. We have about 40% win rate in this case.

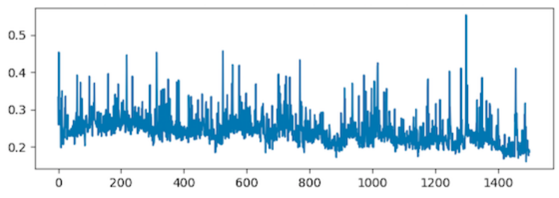


Figure 3: DQN loss for 4 by 4 after 2500 episodes

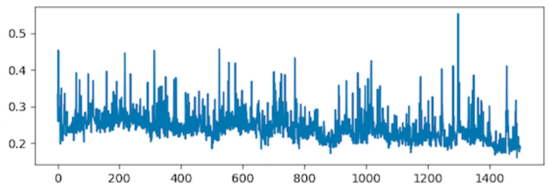


Figure 4: DQN loss for 4 by 4 after another 1500 episodes

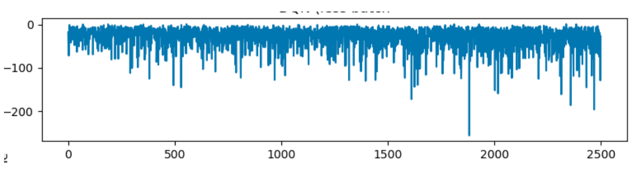


Figure 5: DQN Rewards

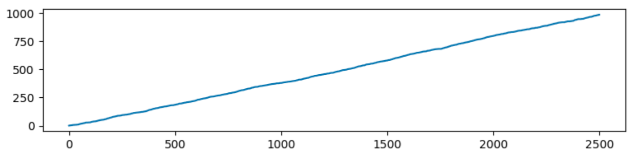


Figure 6: DQN win number

The following are the loss function for the same environment as above, only with Double and Dueling DQN algorithms (Figure 8 and 9). They seem worse than DQN, however, I believe that with more time, they will get better than DQN.
The following are the parameter values we used for the above training:

- $\gamma = .95$

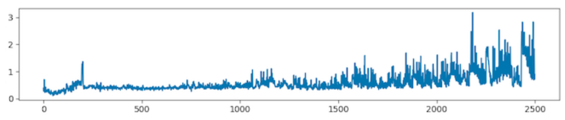


Figure 7: Double DQN Loss

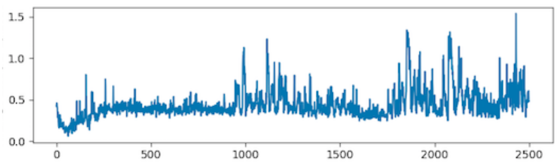


Figure 8: Dueling Double DQN Loss

- batch size = 15
- ϵ decreases from 1.0 to 0.02
- Learning rate is 0.0003 (optimizer Adam)

In Figure 9 we have the loss for the Dueling Double DQN with 4 by 4 grid, 1 goal and 1 ghost. It's better than the 3 goal environment. Figure 10 is the DQN loss per episode for 4by4 grid with 1 goal. As we can see, in both cases the loss is again is gradually decreasing. This were run for 5500 episodes, with batch size 15, length of experience replay 10000, learning rate 0.0003, discount factor 0.95. Dueling Double DQN (DDDQN) has 62% and DQN has 67% accuracy. I expect this to change as well as DDDQN have a better accuracy than DQN as we run the algorithms longer.

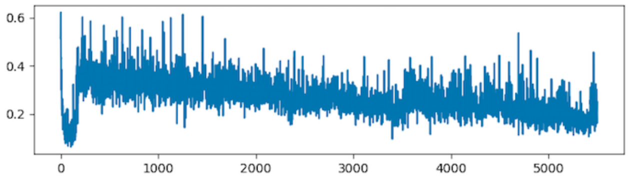


Figure 9: Dueling Double DQN Loss for 4 by 4 with 1 goal, 1 ghost, 2 obstacles

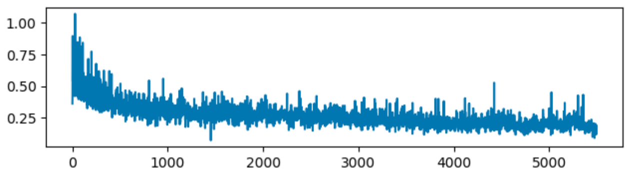


Figure 10: DQN Loss for 4 by 4 with 1 goal, 1 ghost, 2 obstacles

In all of the above cases, the rewards graph looked similar to Figure 5, so I didn't post the rewards vs episode graph for the rest of the runs. However, running the program for much longer time, we would be able to see rewards getting positive and converging to the optimal value

6 Why some things didn't work

- Didn't have powerful enough computer to run the program for longer time to actually see well that the agent is learning
- Since it was taking very long time, it was hard to experiment a lot with different variations of the parameters
- Size of Experience Replay vs the number of experiences sampled. There can be experiences (good ones) that will never be picked so the agents won't be learning no matter how long we run. Because of not having fast enough computers, we couldn't make our batch size big enough to solve this problem. So in the future we could try to prioritize experiences to make the learning better and faster
- How fast ϵ decreases. Making ϵ decrease much faster than the agent is learning will make the agent actually not learn the optimal policy

7 Conclusion and Future Work

In conclusion, we applied Deep RL algorithms to Pac-Man. We extended the envirnoment, wrote the codes from scratch, and run different versions of the code. However, it requires more time than we could have. We need to run the program for much longer to see better results. In the future, we would like to apply prioritized experience replay. We would like to add more ghosts to the environment, move them first randomly as well as teach them to move smart and attack the Pac-Man. Train the Pac-Man and the ghosts by the same and different networks to see what kind of different behaviors will emerge.

8 References

- $cs229.stanford.edu/proj2017/final-reports/5241109.pdf$
- $towardsdatascience.com/advanced-dqns-playing-pac-man-with-deep-reinforcement-learning-3ffbd99e0814$

- *pytorch.org/tutorials/intermediate/reinforcement_q_learning*
- *github.com/udacity/deep − reinforcement − learning*
- *gist.github.com/simoninithomas/*7611*db*5*d*8*a*6*f*3*edde*269*e*18*b*97*fa*4*d*0*c*
- *samyzaf.com/ML/tdf/tdf.html*
- *morvanzhou.github.io/tutorials/*
- *https : //medium.freecodecamp.org/improvements−in−deep−q−learning−dueling−double−dqn−prioritized−experience − replay − and − fixed −* 58*b*130*cc*5682