CS221 PROJECT REPORT

Playing Space Invaders and Q*bert using Deep Reinforcement Learning

Shreyash Pandey (shreyash@stanford.edu)
Vivekkumar Patel (vivek14@stanford.edu)

Abstract

The objective of this project is to build agents using reinforcement learning to play Atari games such as space-invaders and Q-bert. We use Deep Q learning with Experience Replay to solve this task and then implement techniques such as Double DQN and Dueling DQN to improve the performance. We then present a comparison of these techniques and analyse the game play.

1 Task Definition and Modelling

Our goal is to build an agent for two atari games: Space Invaders and Q*bert. Our agent interacts with the game enviroment through a sequence of observations, actions and rewards. We use the Atari Emulator to emulate the environment for these games. The game is played in discrete time steps. At every time step, the agent chooses an action, based on the current state or randomly, from a possible set of actions. The emulator then simulates this action and brings the game to a new state. It also returns any reward received during this step. Hence, we can model our problem as follows:

- State s: We consider the state at any time instance to be an array of pixel values representing the image frame at that instance.
- Action a: The agent has six possible actions in both the games. For eg, in space invaders, the actions are: SHOOT, LEFT (move left), RIGHT (move right), SHOOTANDLEFT (shoot and move left), SHOOTNRIGHT (shoot and move right), NOOP (do nothing).
- Reward r: The reward is returned by the environment after the agent performs an action. We clip the reward so that it lies in the range [-1,1].

We want to make the agent learn to play the game in such a way such that it maximises the total score.

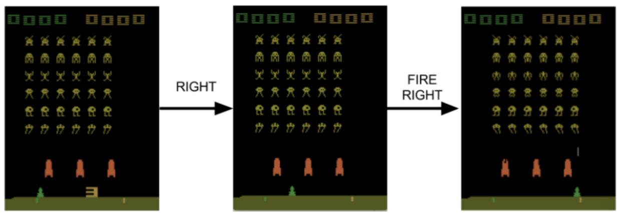


Figure 1: Game state, action and transition

2 Concepts

2.1 Reinforcement Learning

Reinforcement learning has been widely used to solve the problem of programming intelligent agents that learn how to play a game with human-level skills or better. The goal is to learn policies for sequential decision problems by maximizing a discounted cumulative future reward. The reinforcement learning agent must learn an optimal policy for the problem, without being explicitly told if its actions are good or bad.

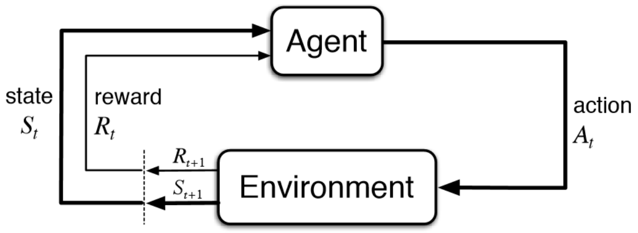


Figure 2: Reinforcement Learning Model

Borrowed from CS 234 Webpage

2.2 Convolutional Neural Network

Recent advances in the use of convolutional neural networks (CNNs), has completely transformed the representation learning domain in Computer Vision - making it possible to automatically learn feature representation from high-dimensional and noisy input data such as images. High-level Computer Vision tasks such as recognition (image classification) and detection have benefited greatly by using CNN based features. Because of their success and versatility, there has been considerable work in enhancing CNN components and improving such deep architectures.

2.3 Q-learning

Q-learning is a model free reinforcement learning technique. Due to a large number of states in the game, it is impossible to make and store a Q-table which would contain the Q-values and best actions for each state. Hence, we use a function to approximation for this task. We use a deep neural network to model this function, which would approximate the Q-table and would calculate the Q-values for each action, for a given state.
$Q^*(s, a)$, where a is an action and s is a state, is the expected value (cumulative discounted reward) of taking action a in state s and then following the optimal policy. Q-learning uses temporal differences to estimate the value of Q*(s,a). An experience (s,a,r,s') provides one data point for the value of Q(s,a). The data point corresponds to the event that the agent received the future value of $r + \gamma V(s')$, where $V(s') = \max_{a'} Q(s', a')$; this is the actual current reward plus the discounted estimated future value. This new data point is called a return. The agent can use the following equation called Bellman's equation to update its estimate for $Q(s, a)$:

$$Q[s, a] \leftarrow Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$$

or, equivalently,

$$Q[s, a] \leftarrow (1 - \alpha)Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a'])$$

. Here, α is the learning rate. Thus lesser the alpha, the more we value our experience and more the α the more we update our experience/table with newly observed returns.

3 Approaches

3.1 Deep Q Networks

We started by first implementing a simple Deep Q Network. The architecture consisted of 3 convolutional layers followed by 2 fully connected layers. The activations in the first four layers are relu. We use a target Q-network, in order to circumvent the moving target problem, for calculating the target value. The target network remains fixed during training and is updated after C (in our case, $C = 10000$) steps, where in we just equate it to the online network.

$$Q_{target} = r_{t+1} + \gamma Q(S_{t+1}, \arg\max_{a} Q(S_{t+1}, a; \boldsymbol{\theta_t^-}); \boldsymbol{\theta_t^-}) \tag{1}$$

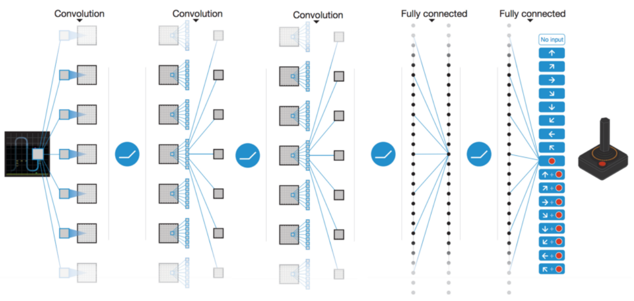


Figure 3: Architecture of the network used in this project

3.2 Exploration-Exploitation Tradeoff

We can train our agent to learn the Q-values of state-action pairs using the equation in the previous subsection. However, this gives rise to a problem. At the beginning, our agent does not have any knowledge of the states or actions. Taking the action that maximizes the expected reward currently may not be the optimal action. It may very well happen that a certain action which has a low Q-value currently, may lead to more rewarding states. This happens because only picking the action that maximizes the expected reward does not lead to the agent exploring all the possible states of the game.

To fix this, we make use of the ϵ greedy policy. In this policy, with probability ϵ, the agent chooses its action randomly from the set of actions, and with probability $1-\epsilon$, it chooses the action with the highest Q-value. We keep ϵ high initially and decrease it gradually. This way, the agent not just explores a lot more states in the state-space, but also learns the optimal policy.

3.3 Experience Replay

Deep Q-Networks need to be trained for long durations in order to get the optimal policy. This may result in several problems. If the environment is continuously changing, then after long training the network may have forgotten information learnt from the past. One solution can then be to repeatedly train the network on the whole sequence of episodes. Also, online training of the network would make use of just 1 or a few frames of input data. This is not very efficient and a much better solution is to use Experience Replay.
To implement this, we store the agent's experiences, the tuple (s_t, a_t, r_t, s_{t+1}), in a circular memory cache. To train the network, we then randomly sample a batch of experiences, and train using the above formulation. Meanwhile, the agent also keeps on playing the game, performing actions for various states, and we keep on storing these experiences in the memory cache. This is an efficient way to learn from past experiences. This way, the agent's learning is logically separated from gaining experience. The model becomes an offline model from an online one. The interleaving of these two processes also ensures that the agent manages to learn and form the optimal policy.
This technique leads to many practical benefits. This way we do not need to train the network on the whole sequence of experience repeatedly because the stored data is reused. Also, during training, we can sample a batch of experiences and hence hardware-efficient mini batches can be used.

3.4 Double DQN

Hasselt et al. [1] discuss that the Q-learning algorithm, which we mentioned above, is known to overestimate action values in certain conditions. In previous works, these overestimates have been attributed insufficient flexibility in function approximation [2], and noise [3]. A high level explanation is as follows: Consider that due to initialization, Q-values of a particular action has a very high bias. As we take the max over the future actions to find the value of the future state, this leads to propagation of this bias to other action values as well and may result in the agent learning suboptimal policies.

Hasselt et al. also propose a solution to the above problem. They decouple the selection from the evaluation in the Q-learning equation. They propose use of two networks $\boldsymbol{\theta}$ and $\boldsymbol{\theta'}$. One set of weights is used to determine the greedy policy whereas the other is used to determine its value. To make minimal changes to the Q-learning algorithm, we can take the greedy policy according to the online network ($\boldsymbol{\theta}$) and use the target network for evaluating it ($\boldsymbol{\theta'}$).

$$Q_{target} = r_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \boldsymbol{\theta_t}); \boldsymbol{\theta_t^-}) \quad (2)$$

3.5 Dueling DQN

Wang, Ziyu, et al. [4] argue that for many states, it is unnecessary to estimate the value of each action choice. For example, in the Space Invaders game setting, knowing whether to move left or right only matters when a hit is imminent. In some states, it is of paramount importance to know which action to take, but in many other states the choice of action has no repercussion on what happens. For bootstrapping based algorithms, however, the estimation of state values is of great importance for every state.

The Q-values correspond to how good it is to take a certain action given a certain state. This can be written as Q(s,a). This action given state can actually be decomposed into two more fundamental notions of value. The first is the value function V(s), which says simple how good it is to be in any given state. The second is the advantage function A(a), which tells how much better taking a certain action would be compared to the others. We can then think of Q as being the combination of V and A. More formally:

$$Q(s,a) = V(s) + A(a)$$

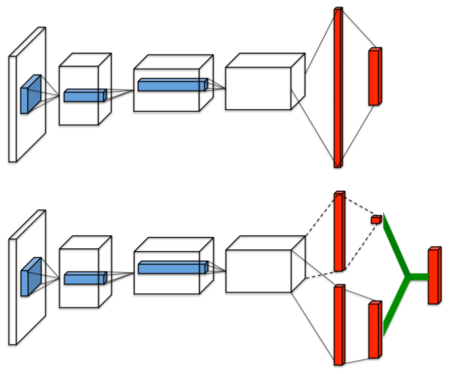


Figure 4: Above: Regular DQN with a single stream for Q-values. Below: Dueling DQN where the value and advantage are calculated separately and then combined only at the final layer into a Q value.

The goal of Dueling DQN is to have a network that separately computes the advantage and value functions, and combines them back into a single Q-function only at the final layer. We can achieve more robust estimates of state value by decoupling it from the necessity of being attached to specific actions.

4 Experiments and Results

4.1 Details of Experiments

We implemented our Deep Networks using Pytorch 0.2.0. We chose pytorch due to its numpy-like syntax which was easy to learn. We use the atari emulator to simulate the games (act as the environment for the agent). We also use the Atari wrappers which take care of certain details such as, but not limited to:

- Converting RGB image frames to grayscale.
- Shrinking and cropping the image to a 84x84 size.
- Continuing an episode till all the lives are over.
- Stacking 4 frames together and taking them as a single input.

Although, the use of these is not clear at start but they are very essential for getting the DQN to work. Our first try at this was without these wrappers and it did not work. The results looked random even after 1.5 days of training.

One of the usefulness of cropping the images is that the top rows only contain the score of the game, which the DQN does not need to know. The DQN already gets the reward information separately. Hence, it is better to remove these rows rather than providing the DQN with useless information which would only waste its time.
Taking a stack of frames together helps the agent in determining the movement of the objects. Using just one frame, determining the movement is not possible. Hence, a stack of 4 frames is taken together as input.

We use the following architecture for our networks:

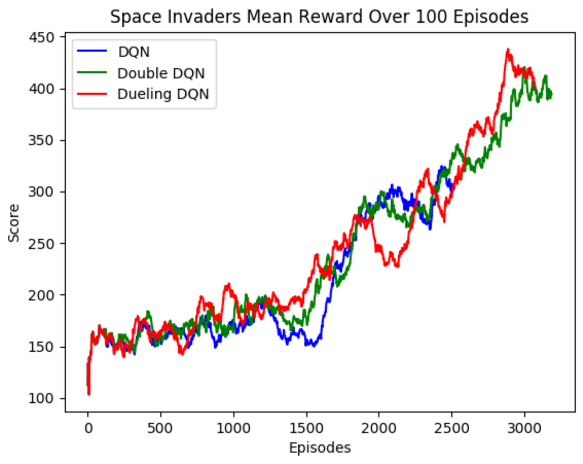
Layer	Type	Out channels/Output size	Kernel Size	Stride
1	Convolutional	32	8	4
2	Convolutional	64	4	2
3	Convolutional	64	3	1
4	Fully Connected	512	NA	NA
5	Fully Connected	Num Actions	NA	NA

Table 1: Architecture Details. Each of the first four layers have relu activation

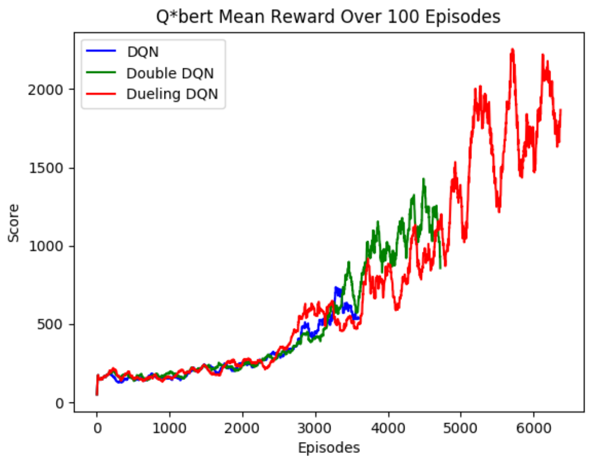
- We have a replay memory of size 1M (i.e) it can store 1M experiences at a time. We implement it in a circular fashion. We start training after 50000 timesteps so that we have enough examples in the replay memory to sample from.
- The target Q network is updated every 10000 actions. We use a learning rate of 0.00025 and RMSProp for gradient descent with alpha as 0.95 and epsilon as 0.01.

The choice of these hyperparamters was based on our literature survey. These values seem to show the best results, which we agree with as well.

4.2 Results

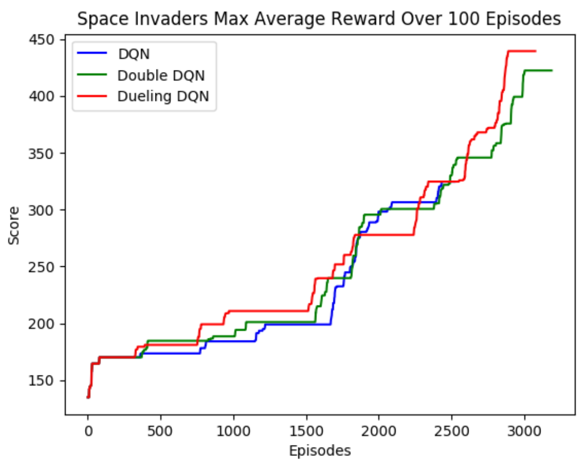


(a) Space Invaders

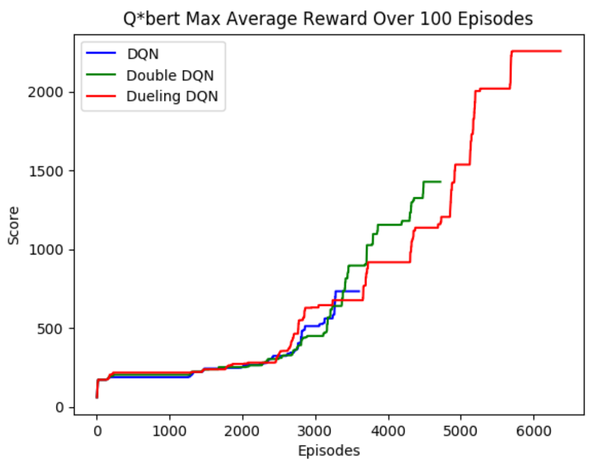


(b) Q*bert

Figure 5: Performance of the three techniques (Average Scores)



(a) Space Invaders



(b) Q*bert

Figure 6: Performance comparison of the three techniques(Max so far)

Algorithm	Episodes trained	Max Scores
Vanilla DQN	2521	324
Double DQN	3187	422
Dueling DQN	3075	440

Table 2: Scores on Space Invaders

Algorithm	Episodes trained	Max Scores
Vanilla DQN	3601	734
Double DQN	4718	1428
Dueling DQN	6369	2256

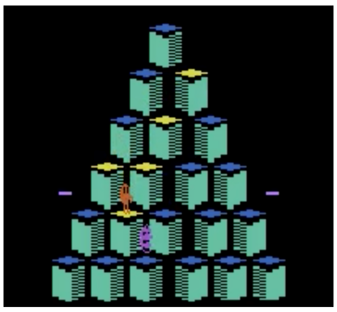
Table 3: Scores on Q*bert

5 Analysis

We did not train our models completely due to limited computation resources. Nonetheless, as can be seen from figure 5, our agent's scores do increase with time (there is a ramp after 1500 episodes), which suggests that the agent is able to learn the optimal policy for the game. Our DQN agents and it's double and dueling variants are able to perform much better than baseline scores of random agents. In fact, our dueling DQN on Q*bert is able to perform better than the initial oracle score of 1500, that we were able to achieve by playing the game for the first time. Also, the plots have not stagnated even after these many episodes which says that training the agent for a longer duration could result in higher scores.

Atari wrappers also saves videos of games plays of the agent. We analyze these as it was the very purpose of this project to make an agent who learnt the best possible strategy. Following are some observations.

As discussed in the Double DQN subsection of the Approaches section, Double DQN does show improvement over normal DQN. The noisy estimates of actions are prevented from passing down to future states due to which the agent learns optimal policies. The below figure also shows that using double dqn, the agent was able to learn optimal policies when an active adversary was present.


(a) Adversary Approaching

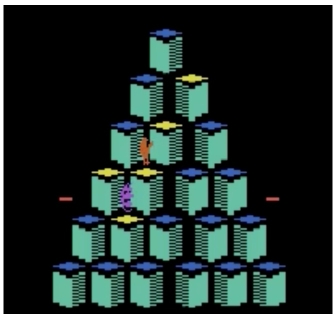
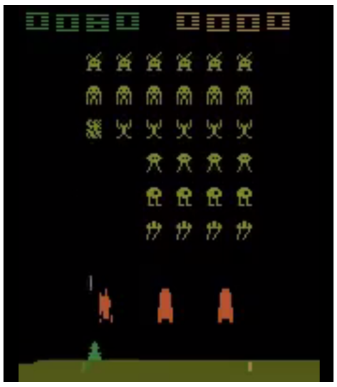

(b) Agent moves farther

Figure 7: Q*bert agent learns to play against active adversary

Dueling DQN shows better performance over both Vanilla DQN and Double DQN. The authors of the original paper comment on how dueling performs particularly well when the action space is large, because then we are bound to get similar actions. This is very much evident in Space Invaders where we have similar looking actions (one example is: LEFT, SHOOTNLEFT). Viewing the video of the gameplays we found that our agent was able to decide between these similar actions. When under attack, it not just moved away but also shot.

A very interesting tactic that the agent was seen to follow in Space Invaders was that it always stayed close to the protecting shields and never wandered too far from them. The adversary group keeps moving from left to right covering the entire width of the screen, but the agent was never found to move far from these shields. It would only oscillate below a shield and never between two shields. It would just move at once between two shields.


(a) Agent under attack

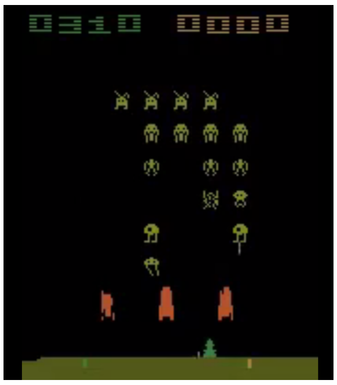

(b) Agent stays close to the shields

Figure 8: Agent never wanders far away from the orange protective shields

5.1 Double-Dueling DQN

Motivated by the improvements of Double and Dueling DQN, we implemented a Double-Dueling DQN, which is just a combination of double and dueling, on Space Invaders. This was much more robust than Double and Dueling separately.

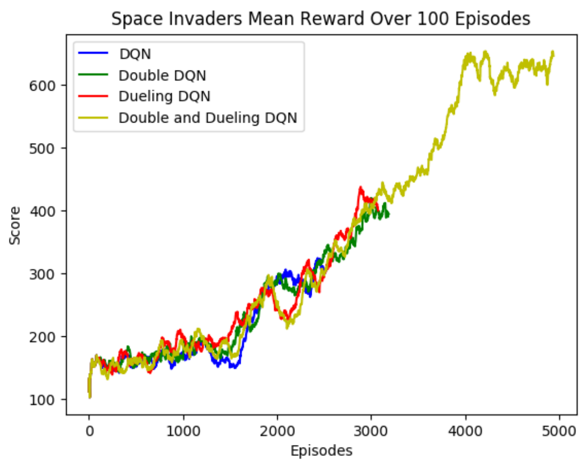


Figure 9: Comparison of previous techniques with Double-Dueling DQN

5.2 Other observations

For the Q*bert one can see oscillations in the graph for mean rewards. We believe this happens because for this game, the rules change with each level. The agent, while learning to play a higher level might perform bad on a lower level, and then with time would improve on both the levels, ultimately resulting in improved performance.

6 Conclusion

We make an agent to play two atari games: Space Invaders and Q*bert. Our choice of games helped us verify the advantages of double and dueling DQN. Double DQN is successful in learning optimal policy even when the rules are changing (Q*bert) whereas dueling DQN is able to choose better actions from multiple similar ones in Space Invaders. We also tried implementing a DRQN but were not able to make it work and hence do not present any results in this paper.

It would be interesting to implement Actor-Critic Methods and attention DRQN over these games and analyse how the performance improves.

7 Acknowledgements

We would like to thank Amani Peddada, our mentor for this project, because of whom we were able to explore many useful techniques. We would also like to thank dxyang, whose github repository helped us figure out nuances of pytorch and other implementation details.

References

[1] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[2] Sebastian Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. In *Proceedings of the 1993 Connectionist Models Summer School*. Erlbaum Associates, January 1993.

[3] Hado V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc., 2010.

[4] Ziyu Wang, Nando de Freitas, and Marc Lanctot. Dueling network architectures for deep reinforcement learning. *CoRR*, abs/1511.06581, 2015.