Playing with DQN

A project made by:

Andrés Bermeo Marinelli - Davide Basso

TABLE OF CONTENTS

The Problem

- Try to **solve** one of the most popular Atari game, **Space Invaders**, by using **Reinforcement Learning** algorithm **Deep Q-Learning (DQN)**.
- Develop a **neural network** to **approximate** the **Action-Value function Q*(s,a)**.
- **Overcome issues** as:
    - **High correlation** between data samples.
    - **Non-fixed** data **distribution**.
- Make the **Agent efficiently** and **effectively learn**.

The Game - Space invaders

- **Player piloting** a **laser** cannon to **battle** columns of descending **aliens** while using shields to block alien fire.
- The **speed** of the **alien** approach **increases** as the **game progresses**.
- A **bonus alien spaceship** appears from **time to time**, which offers the player an opportunity to **score additional points by blowing it up**.

The Python Setting

- First, download **Space Invader's Rom**.
  - This is perhaps the **most complicated step** as **some ROMS don't work** while others do.
  - We found that the **version from 1983 (specifically) with a .a26 extension solved all issues**.
- Set the environment using **OpenAI's Gym** library.
- Pick one among the possible environment settings:
  - We opted for ***'SpaceInvadersNoFrameskip-v4'***, i.e. we get a fixed frameskip of 1 and the probability of choosing the previous action as next action is set to 0.

The MDP

- **State**: an **RGB image** of the screen of shape **(210, 160, 3)**.
- **6** possible **actions** to take:
    - 0 - Do **nothing**
    - 1 - **Fire**
    - 2 - **Right**
    - 3 - **Left**
    - 4 - **Right Fire**
    - 5 - **Left Fire**

The MDP

- **Rewards:**
    - **Integer value** that **depends on** whether the **agent survived** or **shot down aliens**.
- **End of the episode:**
    - Occurs when the **player** gets **hit by** one of the **Aliens' lasers 3 times**.

Preprocessing Steps

- **Original State space** is too **complex**:
  - **Convert** the image to **grayscale** (colors don't bring any useful informations).
  - **Scoreboard** at the top and **green portion** at the bottom of the frame are **useless**. We can **crop them**.
- **Resize** the image to **(84x84)**.
- **Rewards** can be **clipped** to both **save space** and **generalize** to **different games**:
  - **1, 0, -1** values.

Preprocessing Steps

- We **skip every 3 frames** and **repeat** the **action** on the **skipped frames**.
- A **maximum** is taken **over two consecutive frames**:
  - Avoid possible blurs or ghosting issues.
- **Single frames** don't bring informations on motion of the game:
  - A **state** is then **composed** of **4 stacked frames**.
  - **NB**: These **frames are not consecutive** and are taken every 3 steps, so in a **single stack** we **collect informations** of **12 frames**.

Deep Q-Learning

- The **DQN** learns an **approximation of the Q-table** through **neural nets**:
  - **Mapping** between the **states and actions** that an agent will take.
- We would like to **select an action that** would **maximize** our **future returns**
  - Using **Epsilon Greedy** and **Greedy** strategies.

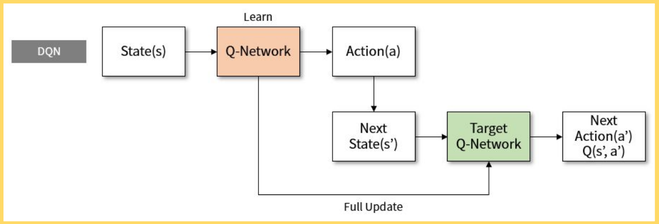
Deep Q-Learning - Problems

- **Data distribution changes** as the algorithm learns new behaviours:
  - **Neural networks** assume **fixed underlying distribution**.
  - **Hard Convergence**
- **Data** is **highly correlated**:
  - Most **Deep Learning algorithms** assume **independent data samples**.
  - **Sub-optimal** solutions.

Deep Q-Learning - Solutions

- To **solve** the **first issue** we can use **two separate Q-value estimators**:
  - **Policy network**: **estimate** the **Q-values for** the **actions**.
  - **Target network**: used to **obtain** the **target Q-value estimation**.

- **Hard update** for the **Target Network**:
  - **Copy Policy Network weights** every **N steps** (N = 10k) in order to have **stable rewards**.

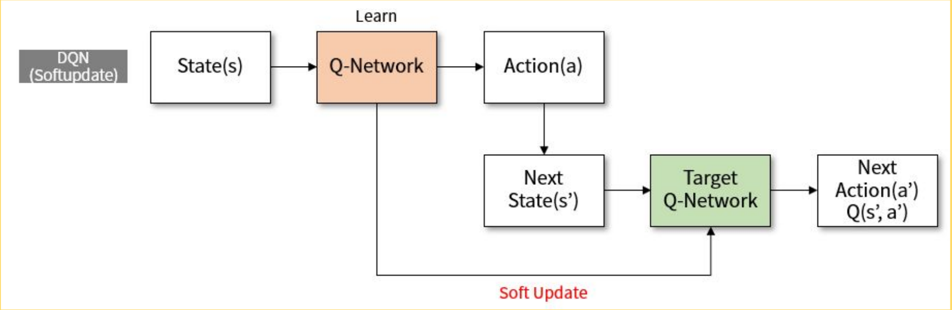


Deep Q-Learning - Solutions

- To **solve** the **second issue** instead we **use** the technique called **Experience Replay**:
  - **Store** a **transition** (state, action, reward, next state) **in a queue of fixed size**.
  - When updating the Q-function, **sample** a **random batch from queue** and **apply GD**.
  - **PRO**: **Extract** as much **informations** as possible out of an environment.
  - **CONS**: Requires **large RAM**.

Deep Q-Learning - Tweaks

- **Soft update** for the **Target Network**:
  - **Instead** of **copying Policy Network weights** every **N steps**, we **do it more frequently** (every time the agent learns) and in a "**smoother**" way (tau=0.001).



$$\theta^- = \theta \times \tau + \theta^- \times (1 - \tau)$$

Double Deep Q-Learning

- **Vanilla DQN** suffers from **overestimating Q-values**.
    - This is **due** to the term **$max_a(Q(s,a))$ in the Q-function**.
- **Solution: remove** the **max** operator from the **target estimate**
    - **Select** the **optimal action from** the **policy DQN** network.
    - **Get** the **target estimate** for this optimal action **from the target DQN**.

$$Q(s,a;\theta) = r + \gamma Q(s', argmax_{a'} Q(s',a';\theta);\theta')$$

Further tweaks

- We **tried different hyperaparameters** configurations.
- We took **inspiration from**:
    - **OpenAI**
    - **DeepMind**
    - **Posts**
    - **Papers**

Possible Improvements

- **Prioritized Experience Replay.**
- **Dueling Deep Q-Learning.**
- Shift to **Policy Gradient:**
  - Try with state of the art **Actor-Critic agents** like **PPO**.