PLAYING RETRO GAMES WITH DUELING DQN

ISPR Final project - Michele Morisco (505252)

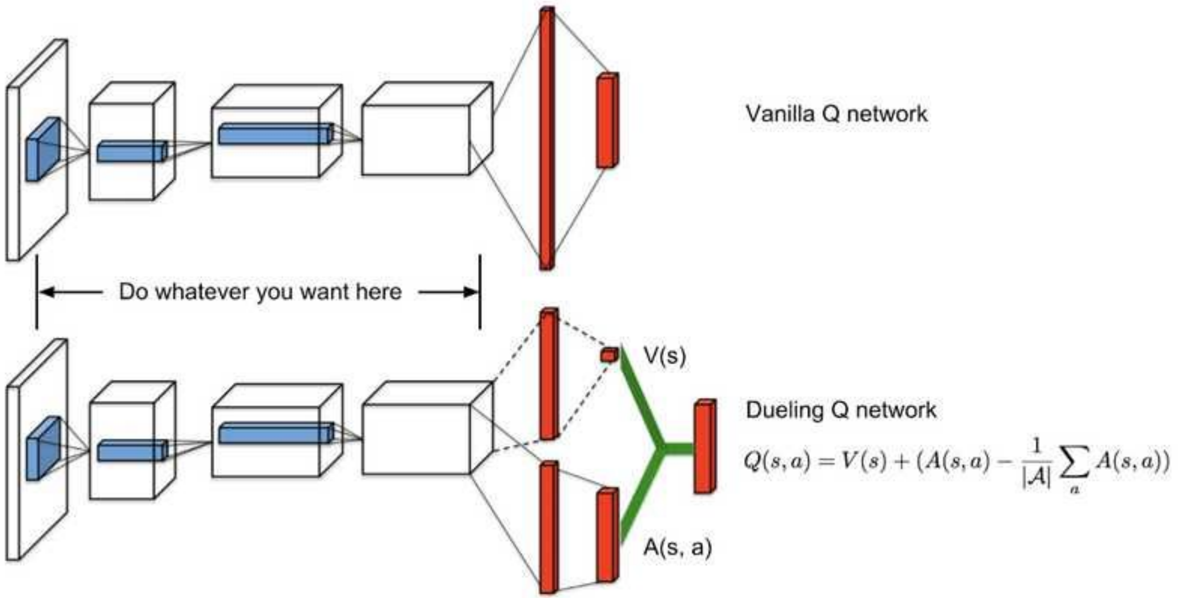
Introduction

My final project includes:

- implementation of a Dueling DQN models
- experiments on two games of Atari environments* and one game of Nintendo Entertainment System environment using OpenAI Gym.
- comparing the results between DQN, Dueling DQN, and Double Dueling DQN models.

*Mnih et al., Playing Atari with Deep Reinforcement Learning. arXiv, 2013.
https://arxiv.org/abs/1312.5602

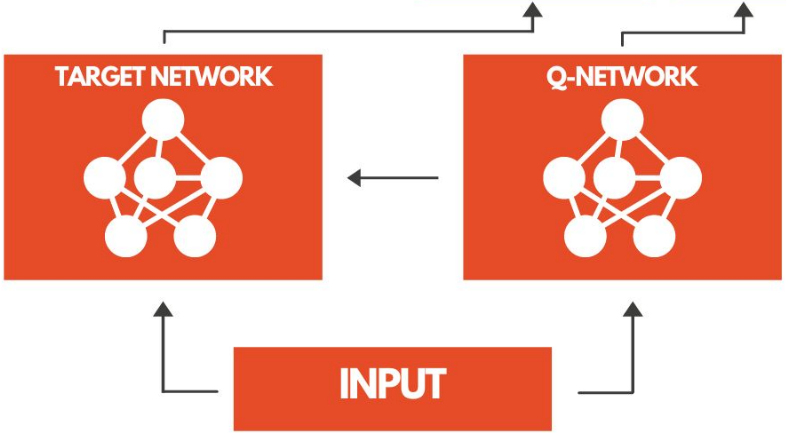
Dueling Deep Q-Network



Q-Network is divided into two parts: ***V***, the Value function and ***A***, the Advantage function.
A captures how better an action is compared to the others at a given state, while as ***V*** captures how good it is to be at this state.
The Q function is represented by a sum of Value and the Advantage function.

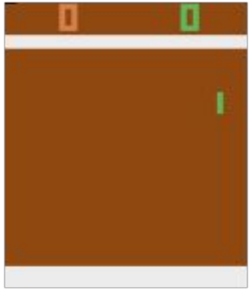
Double Dueling Deep Q-Network

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \boxed{\gamma \max_a Q(S_{t+1}, a)} - \boxed{Q(S_t, A_t)} \right]$$



There are two Deep Q-networks: the DQN is responsible for the **selection** of the next action; on the other hand, the Target network is responsible for the **evaluation** of that action.
The target value is not produced by the maximum Q-value, but by the Target network.
This addresses maximization bias.

Environments



PONG

The agent controls the right paddle.

The agent's goal is to reach 21 points and win the game.

Reward

$$R = agent - cpu$$

where *agent* is player's score and *cpu* is the opponent's score

BOXING

The agent controls the white player.

The agent's goal is to knock out the opponent reaching 100 points.

Reward

$$R = agent - cpu$$

where *agent* is player's score and *cpu* is the opponent's score

SUPER MARIO BROS.

The agent controls Mario.

The agent's goal is to reach the flag, i.e. to complete the level.

Reward

$$R = v + c + d$$

where *v* represents player position, game time *c* and death penalty *d*

Implementation

```python
class DuelingDQN(nn.Module):
    def __init__(self, input_shape, num_actions):
        super(DuelingDQN, self).__init__()

        self.input_shape = input_shape
        self.num_actions = num_actions

        #convolutional network
        self.cnn = nn.Sequential(
            nn.Conv2d(input_shape[0], 32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(32, 64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(64, 64, kernel_size=3, stride=1),
            nn.ReLU()
        )

        #state-values function
        self.value_stream = nn.Sequential(
            nn.Linear(self.feature_size(), 512),
            nn.ReLU(),
            nn.Linear(512, 1)
        )

        #state-dependent action advantage function
        self.advantage_stream = nn.Sequential(
            nn.Linear(self.feature_size(), 512),
            nn.ReLU(),
            nn.Linear(512, self.num_actions)
        )

    def forward(self, x):
        x = x.float()
        x = self.cnn(x)
        x = x.view(x.size(0), -1)
        values = self.value_stream(x)
        advantages = self.advantage_stream(x)

        #combining  both parts into a single output, to estimate the Q-values
        qvals = values + (advantages - advantages.mean())
        return qvals
```

```python
class ReplayMemory:
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def store(self, state, action, new_state, reward, done):
        state = np.expand_dims(state, 0)
        new_state = np.expand_dims(new_state, 0)

        self.memory.append([state, action, new_state, reward, done])

    def replay(self, batch_size):
        state, action, new_state, reward, done = zip(
            *random.sample(self.memory, batch_size)
        )

        return np.concatenate(state), action, np.concatenate(new_state), reward, done

    def __len__(self):
        return len(self.memory)
```

Implementation of the Dueling DQN algorithm with a replay memory

Implementation

The agent has a function that implements the experience replay

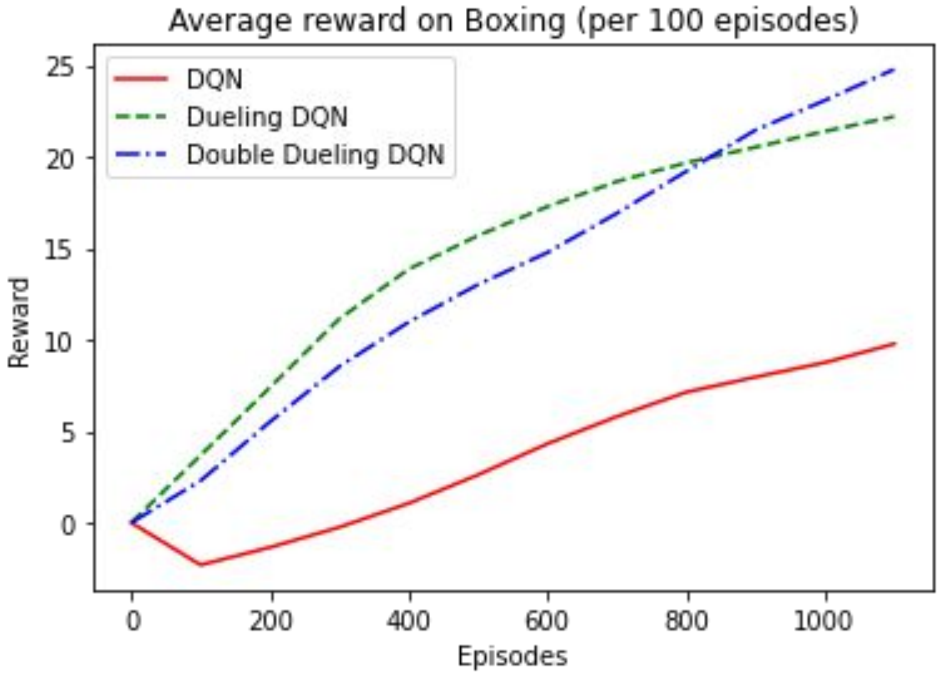
In replay memory D store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ where at is the action achieved according to ε-greedy policy

Sample a random mini-batch of transitions (s, a, r, s') from D

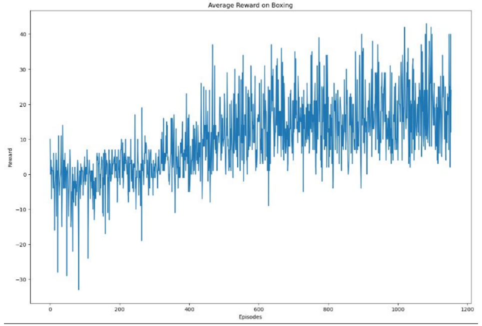
Compute Q-learning targets with respect to old fixed weights

```
def experience_replay(self):
    """Use the Q-update equations to update the network weights"""

    #if the agent is a Double DQN then update the target network every few frames
    if self.double_enabled and self.step % 10000 == 0:
        self.target_net.load_state_dict(self.policy_net.state_dict())

    #check if the memory is quite full to update the network weights
    if self.memory_sample_size > self.replay_buffer.__len__():
        return

    if self.replay_buffer.__len__() <= self.replay_initialization:
        return

    # Sample a batch of experiences
    state, action, next_state, reward, done = self.replay_buffer.replay(self.memory_sample_size)
    state_batch      = torch.tensor(state).to(self.device)
    next_state_batch = torch.tensor(np.array(next_state), requires_grad=False).to(self.device)

    action_batch     = torch.LongTensor(action).to(self.device)
    reward_batch     = torch.FloatTensor(reward).to(self.device)
    done_batch        = torch.FloatTensor(done).to(self.device)

    self.optimizer.zero_grad()
    # Double Q-Learning target is Q*(S, A) <- r + γ max_a Q_target(S', a)
    if self.double_enabled:
        q_values      = self.policy_net(state_batch)
        current       = q_values.gather(1, action_batch.unsqueeze(1)).squeeze(1)

        next_q_values = self.target_net(next_state_batch)
        next_q_value  = next_q_values.max(1)[0]
        target = reward_batch + torch.mul(self.gamma * next_q_value, (1. - done_batch))
    else:
        # Q-Learning target is Q*(S, A) <- r + γ max_a Q(S', a)
        q_values      = self.net(state_batch)
        current       = q_values.gather(1, action_batch.unsqueeze(1)).squeeze(1)

        next_q_values = self.net(next_state_batch)
        next_q_value  = next_q_values.max(1)[0]
        target = reward_batch + torch.mul(self.gamma * next_q_value, (1. - done_batch))

    loss = self.l1(current, target.data)
    loss.backward() # Compute gradients
    self.optimizer.step() # Backpropagate error

    return loss.item()
```

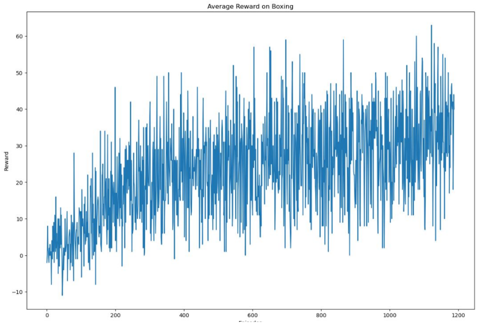
Results (Boxing)



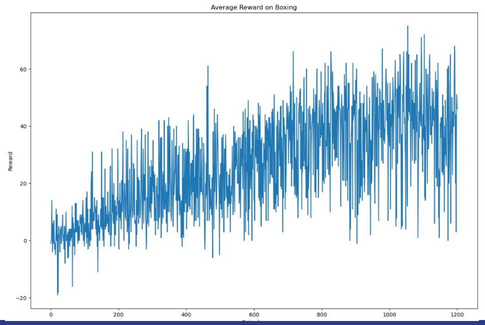
Each plot shows the reward trend during the training phase. Dueling DQN agent learns faster but Double Dueling DQN seems to outperform other models.



DQN

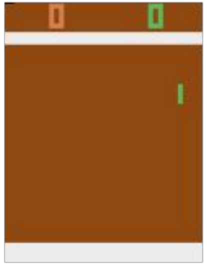
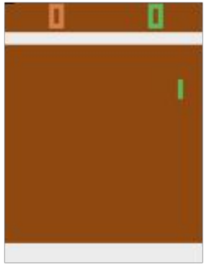
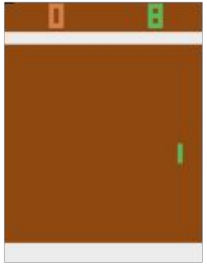


Dueling DQN



Double Dueling DQN

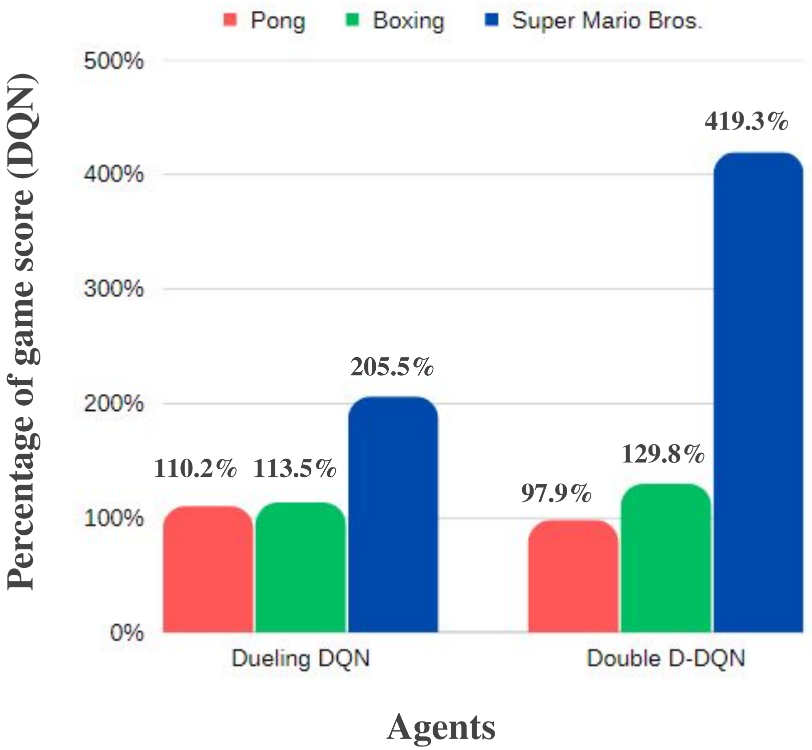
Results (Double Dueling DQN)

Episode	**Pong**
0	
250+	
1000+	

Episode	**Boxing**
0	
250+	
1250+	

Episode	**Super Mario Bros.**
0	
3000+	
5000+	

Results comparison



Comparison of two agents training on the three games with respect to the **mean DQN scores**:

- Double Dueling DQN performs very slightly worse than the DQN in Pong, but in other games, it performs better.
- Dueling DQN agents learn very good in all environments.
- DQN is widely outperformed by other agents, especially on Super Mario Bros.

Conclusions – I

In my experiments, I tested my implementation of Dueling DQN on three environments using OpenAI Gym

Each agent has learned to play much better on Pong.

On the contrary, Boxing and Super Mario Bros. need to observe more frames to improve their performances. Despite that, Double Dueling DQN outperformed other agents and can be able to play pretty well in the three games tested.

Issues:

- Limited computational resources (Kaggle’s limited resources)
- Time (Necessary more observations)
- Different type of games environment (Atari vs. NES)
- High complexity of Super Mario Bros. environment (Applying a different memory replay)
- Managing larger networks (Double Dueling DQN)

Conclusions - II

	Human	DQN		D-DQN		Double D-DQN	
	Score	**Score**	**%Human**	**Score**	**%Human**	**Score**	**%Human**
Pong	9.3	18.7	201.1	20.6	221.5	18.3	196.8
Boxing	4.3	35.6	827.9	40.4	939.5	46.2	1074.4

Comparing the agents with a professional human games tester scores on two Atari games, according to a Nature article.*

According to the table, my agents can outperform the human games tester with very good results.

*Mnih et al., Human-level control through deep reinforcement learning. Nature, 2015.
https://www.nature.com/articles/nature14236

THANK YOU FOR YOUR ATTENTION