DEEP REINFORCEMENT LEARNING

PROJECT 1. Navigation.

Markus Buchholz

1. GOAL

In this project, the goal was to train the Agent to navigate in square environment how to collect yellow bananas. Project requirement is to collect average score of +13 over 100 consecutive episodes.

2. INTRODUCTION

Origin of Deep Reinforcement Learning is pure Reinforcement Learning, where problems are typically framed as Markov Decision Processes (MDP). The MDP consists of a set of states S and actions A. Transitions between states are performed with transition probability P, reward R and a discount factor gamma. Probability transition P (system dynamics) reflects the number of different transitions and rewards occurrence from one state to the other, where the sequential state and reward depend only on the state and action taken at the previous time step.

Reinforcement Learning defines environment for the Agent to perform certain actions (according to policy) to maximize the reward. The foundation of optimal behaviour of the Agent is defined by Bellman equation, which is a widely used method for solving practical optimization problems. To solve the Bellman optimality equation, we use a dynamic programming.

When the agent exists in environment and transits to the other state (position) we need to estimate the value of the state V(s) (position) – **state value function**. Once we know the value of each state we can figure out what is the best way to act Q(S, A) – **action value function** (simply by following the state that with the highest value).

These two mappings or functions are very much interrelated and help us find
an optimal policy for our problem. We can express that state value function tells us how good is it to be in state S if the Agent follows policy π.
Meaning of symbols is as follows:

E [X] – expectation of random variable X
π – policy
G_t – discounted return at time t
γ – discount rate

$$v_\pi(s) \doteq \mathbb{E}[G_t|S_t = s]$$

However, the action-value function q (s, a) is the expected return starting from state S, taking action A, and following policy π and tells us how good is it to take a particular action from a particular state,

$$q_\pi(s,a) \doteq \mathbb{E}[G_t|S_t = s, A_t = a]$$

It is smart to mention that the difference between the state value function and the *Q* function is that the value function specifies the goodness of a state, while a *Q* function specifies the goodness of an action in a state.

The MDP is solved by **Bellman equation**, named after **Richard Bellman**, American mathematician. The equation contributes in finding the optimal policies and value functions. The Agent chooses the action according to the imposed policy (strategy – formally, policy is defined as the probability distribution over actions for every possible state). Different policies the Agent can follow implies different value functions for the state. However, if the goal is to **maximize the collected rewards,** we have to find the best possible policy, called optimal policy.

$$v_*(s) \doteq \max_\pi v_\pi(s)$$

On the other hand the optimal state value function is the one that has a higher value compared to all other value functions (maximum return), therefore the optimal value function can be also estimated by taking the maximum of the *Q* :

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_*(s,a)$$

Finally, the Bellman equation for the value function can be represented as,

$$v_\pi(s) = \sum_{a \in \mathcal{A}(s)} \pi(a|s) \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s',r|s,a)(r + \gamma v_\pi(s'))$$

Similarly, the Bellman equation for the *Q* function can be represented as follows:

$$q_\pi(s,a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s',r|s,a)(r + \gamma \sum_{a' \in \mathcal{A}(s')} \pi(a'|s') q_\pi(s',a'))$$

Based on optimal state value function and above equations for state value function action-value function we can arrange final equation for optimal value function called Bellman optimality equation:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s',r|s,a)(r + \gamma v_*(s'))$$

$$q_*(s,a) = \sum_{s' \in \mathcal{S}, r \in \mathcal{R}} p(s',r|s,a)(r + \gamma \max_{a' \in \mathcal{A}(s')} q_*(s',a'))$$

3. DEEP REINFORCEMENT LEARNING (DEEP Q – NETWORKS – DQN)

Reinforcement learning can be sufficiently applicable to the environment where the all achievable states can be manged (iterated) and stored in standard computer RAM memory. However, the environment where the number of states overwhelms the capacity of contemporary computers (for Atari games there are 128^{33600} states) the standard Reinforcement Learning approach is not very applicable. Furthermore, in real environment, the Agent has to face with continuous states (not discrete), continuous variables and continuous control (action) problems.
Bearing in mind the complexity of environment the Agent has to operate in (number of states, continuous control) the standard well defined Reinforcement Learning Q – table is replaced by Deep Neural Network (Q – Network) which maps (non – linear approximation) environment states to Agent actions. Network architecture, choice of network hyperparameters and learning is performed during training phase (learning of Q – Network weight).
DQN allows the Agent to explore unstructured environment and acquire knowledge which over time makes them possible for imitating human behaviour.

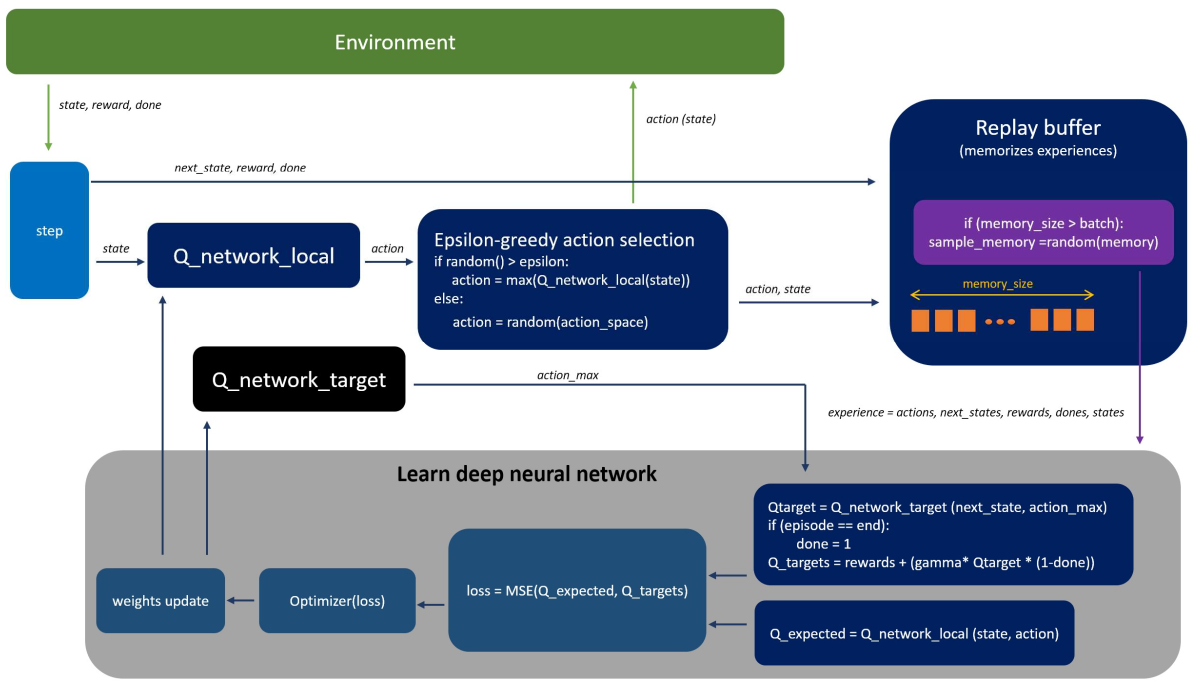
4. LEARNING ALGHORITHM DQN

The main concept of DQN was depicted on below figure (during training process), where Q – network proceeds as a as nonlinear approximator which maps both state into an action value. During the training process, the Agent, interacts with the environment and receives data, which is used during the learning the Q – network. The Agent explores the environment to build a complete picture of transitions and action outcomes. At the beginning the Agent decides about the actions randomly which over time becomes insufficient. While exploring the environment the Agent tries to look on Q – network (approximator) in order to decide how to act. We called this approach (combination of random behaviour and according to Q – network) as an **epsilon – greedy** method (Epsilon -greedy action selection block), which just means changing between random and Q policy using the probability hyperparameter epsilon.

The core of presented Q-learning algorithm is derived from the supervised learning.
Here as it was mention above, the goal is to approximate a complex, nonlinear function *Q(S, A)* with a deep neural network.
Similarly, to supervised learning, in DQN, we can define the loss function as the squared difference between the target and predicted value, and we will also try to minimize the loss by updating the weights (assuming that the Agent performs a transition from one state *s* to the next state *s'* by performing some action *a* and receive a reward *r*.).

$$Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

During the learning process we use two separate Q – networks (Q_network_local and Q_network_target) to calculate the predicted value (weights θ) and target value (weights θ’). The target network is frozen for several time steps and then the target network weights are updated by copying the weights from the actual Q network. Freezing the target Q – network for a while and then updating its weights with the actual Q network weights stabilizes the training.



```
Epsilon-greedy action selection
if random() > epsilon:
    action = max(Q_network_local(state))
else:
    action = random(action_space)
```

```
if (memory_size > batch):
sample_memory =random(memory)
```

```
Qtarget = Q_network_target (next_state, action_max)
if (episode == end):
    done = 1
Q_targets = rewards + (gamma* Qtarget * (1-done))
```

```
Q_expected = Q_network_local (state, action)
```

```
loss = MSE(Q_expected, Q_targets)
```

Figure 1. DQN algorithm concept

On order to make training process more stable (we would like to avoid learning network on data which is relatively corelated, which can happen if we perform learning on consecutive updates – last transition) we apply replay buffer which memorizes experiences of the Agent behaviour. Then, training is performed on random samples from the replay buffer (this reduces the correlation between the agent's experience and helps the agent to learn better from a wide range of experiences).

The DQN algorithm can be describes as follows:

1. Initialize replay buffer
2. Pre-process and the environment and feed state *S*) to DQN, which will return the *Q* values of all possible actions in the state.
3. Select an action using the epsilon-greedy policy: with the probability epsilon, we select a random action A and with probability 1-epsilon. Select an action that has a maximum *Q* value, such as A = argmax(Q(S, A, θ)).
4. After selecting the action *A*, the Agent performs chosen action in a state *S* and move to a new state *S'* and receive a reward R.
5. Store transition in replay buffer as <S,A,R,S'>.
6. Next, sample some random batches of transitions from the replay buffer and calculate the loss using the formula:

$$Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$$

7. Perform gradient descent with respect to actual network parameters in order to minimize this loss.
8. After every *k* steps, copy our actual network weights to the target network weights.
9. Repeat these steps for *M* number of episodes.

5. PROJECT SETUP. RESULTS.

In Navigation project following setup of neural network architecture and hyperparameters were applied:

Depicted plot of rewards per episode illustrates that the Agent is able to receive an average reward (over 100 episodes) of at least +13 while playing 2247 episodes.

Q-Network architecture:

Input layer FC1: 37 nodes in, 64 nodes out
Hidden layer FC2: 64 nodes in, 64 nodes out
Hidden layer FC3: 64 nodes in, 64 nodes out
Output layer: 64 nodes in, 4 out – action size

Applied hyperparameters:

```
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size
GAMMA = 0.99            # discount factor
TAU = 1e-3              # for soft update of target parameters
LR = 5e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network
Epsilon start = 1.0
Epsilon start = 0.01
Epsilon decay = 0.999
```

```
Episode 100     Average Score: 0.08
Episode 200     Average Score: 0.322
Episode 300     Average Score: 1.15
Episode 400     Average Score: 1.62
Episode 500     Average Score: 3.24
Episode 600     Average Score: 4.42
Episode 700     Average Score: 5.08
Episode 800     Average Score: 5.79
Episode 900     Average Score: 6.82
Episode 1000    Average Score: 7.82
Episode 1100    Average Score: 8.82
Episode 1200    Average Score: 8.65
Episode 1300    Average Score: 9.74
Episode 1400    Average Score: 10.01
Episode 1500    Average Score: 10.07
Episode 1600    Average Score: 10.97
Episode 1700    Average Score: 11.63
Episode 1800    Average Score: 12.57
Episode 1900    Average Score: 12.28
Episode 2000    Average Score: 12.37
Episode 2100    Average Score: 12.66
Episode 2200    Average Score: 12.70
Episode 2300    Average Score: 12.81
Episode 2347    Average Score: 13.00
Environment solved in 2247 episodes!    Average Score: 13.00
```

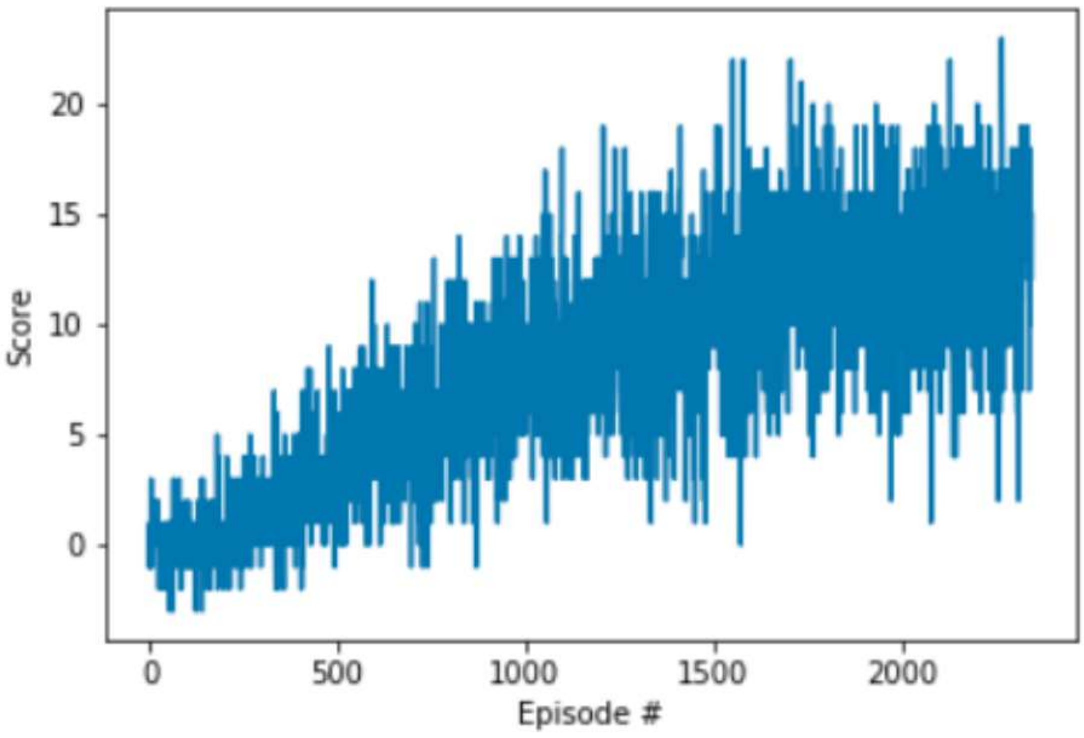


Figure 2. Average score while Agent learns

6. IDEAS OF FUTURE WORK

Bering in mind the experience with Deep Learning the future work will concentrate with applying image management (**Learning from pixels**). The architecture of DQN is shown in the following figure, where we feed a game screen and Q network approximates the Q value for all actions in that game state. Furthermore, the action is estimated as in discussed DQN algorithm.

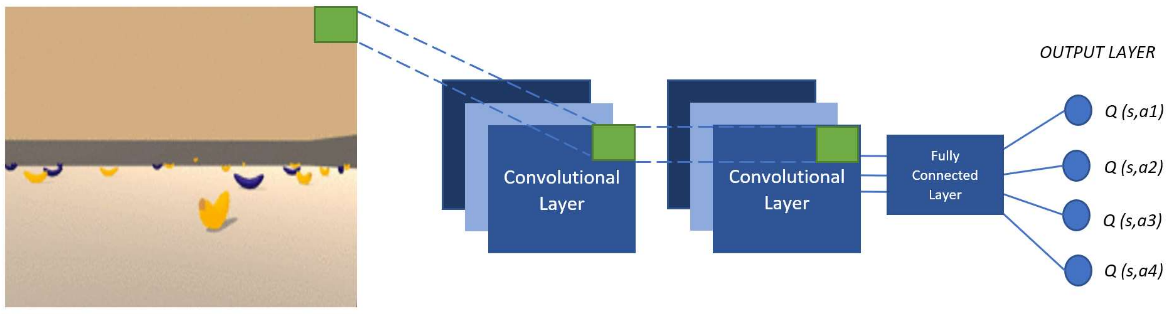


Figure 3. Learning from pixels concept

Secondly the future work will focus on implementing a **Dueling of DQN**. In this new architecture we specify new advantage function, which specifies how good it is for an agent to perform an action *a* compared to other actions (Advantage could be positive or negative).
The architecture of dueling DQN is the same as described above DQN, except that the fully connected layer at the end is divided into two streams (see depicted below figure).
In environments with certain number of action space in a one state the most of the computed actions will not have any effect on the state. Additionally, there will be number of actions with redundant effects. In this case the new dueling DQN will estimate the Q values more precisely than the DQN architecture.
One stream computes the value function, and the other stream computes the advantage function (to decide which action is preferred over the other).

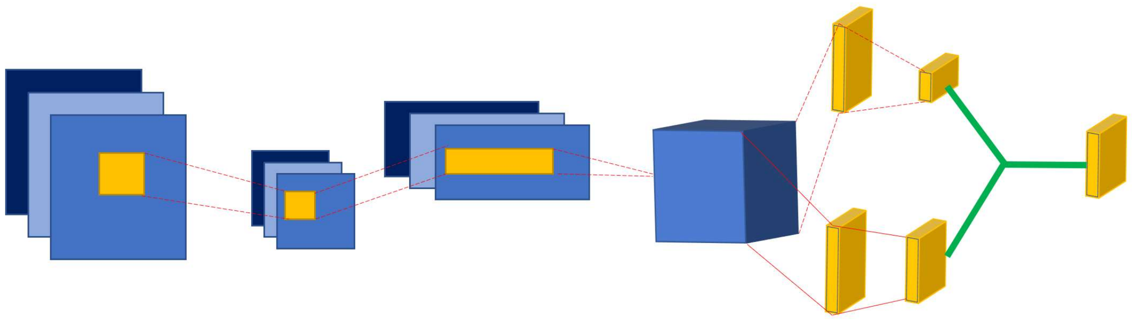


Figure 4. Architecture of dueling DQN

Lastly, we can consider to **Learning from human preference** (OpenAI and DeepMind). The main idea of thin new concept is to learn the Agent according to human feedback. The agent, receiving the human feedback will try to do the actions preferred by the human and set the reward accordingly. Human interaction with the Agent directly contributes to overcome the challenge connected with designing the reward function and complex goal functions.

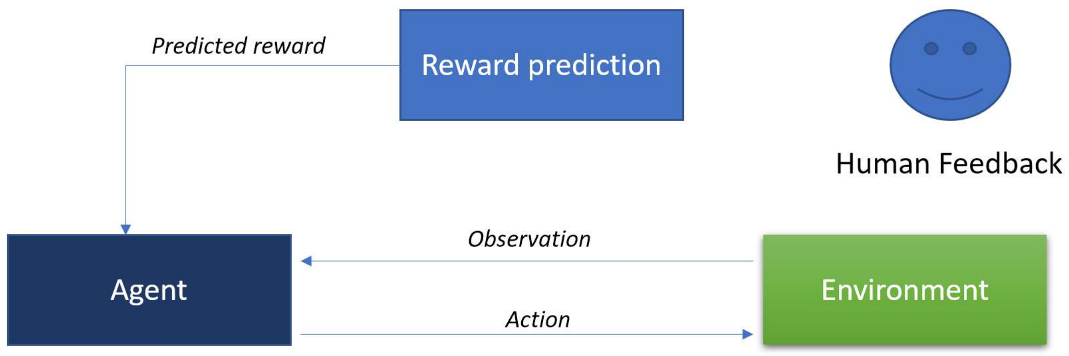


Figure 5. Learning from human preference concept