Deep Reinforcement Learning
P1 Navigation project

Christophe Gosset

May 1, 2021

Abstract

This document describes the implementation of the Deep Q-Network (DQN) algorithm to solve an environment described by a 37-dimensions continuous variable states. The neural network is modified from a CNN to a stack of fully-connected layers. The network learning uses two important features to avoid action sequence correlation and instability: experience replay and a dedicated target network.

1 Objective

The objective of the project is to train an agent to navigate in the "Banana" environment in order to catch only yellow bananas as fast as possible. The goal of the agent is to learn the Q-function related to 37 continuous state variables returned by the environment, and 4 navigation actions (left, right, forward bacward). After each action, the reward is 0 if the agent does not catch any banana, +1 if it catches a yellow banana and -1 for a blue banana. An episode ends after 1000 steps. The Q-funtion to learn is a non-linear continuous funciton defined by a neural network model. As the Q-function returns a value for any state/action pair, the learning problem is a regression problem. The method and the technical elements used to train the network are described in DQN paper (2015).

In section 2, we will present the model used to fit the non-linear Q-function, a neural network achitecture. In section 3, we will introduce the tasks used by the agent to solve the environment. Finally, in section 4, we will discuss the implementation and experimental results, including hyperparameters choice and reward curve versus episodes.

2 Neural network model

In the DQN algorithm, the non-linear function is provided a neural network and the feedback signal from the environment is an image. The suitable architecture

to extract features from an image is an architecture based on convolutional layers, to take advantage of connections between local image representations. One advantage of convolutional layers is the reduction of the number of parameters compared to fully connected layers.

In our problem, the environment provided a feedback signal from a 37-dimensional space, without any order relation between dimensions. In this case, fully connected layers are required to catch potential interactions between all variables. The architecture of the proposed model is a stack of two hidden fully-connected layers. The input layer size is equal to the dimension of the state space (37 input neurons), and the output layer to the size of the action space (4, in our case). The number of neurons of the two hidden layers are hyperparameters. We will see that we choose 64 for the first hidden layer and 32 for the second. Each hidden layer is followed by a RELU function. Actually, the architecture is quiet common and simple. Our goal is to minimize the number of parameters to optimize to obtain the best result.

3 Description of the Learning algorithm

The goal of the agent is to evaluate the value of each action, until the end of the episode, in any state of the environment. The learning aims at optimizing the parameters of the non-linear continuous Q-function defined by the neural network to minimize a suitable loss function. To achieve this "meta goal", the agent must be able to:

- Given the environment state, take an action according to the epsilon-greedy rule.
- Interact with the environment by achieving the action.
- Interpret the feedback state signal generated after an action to learn from the action, i.e. update the Q-function.

The last point is addressed by the DQN paper, which describes the methodology to solve the issues induced during the training of the neural network as: instability, actions sequence correlation. In particular, two methods are implemented:

- Experience replay: It avoids action sequence correlation.
- Target network: It avoids learning instabilities.

3.1 Experience replay

With experience replay, achieveing an action and learning from an action are decorelated. At each step of an episode, the agent:

– chooses an action in the current state with the ε-greedy rule

– executes the action

– receives the reward

– observes the next state

The next state will become the current state in the next action step. This process is achieved until the end of the episode. For each interaction with the environment, the tuple (state, action, reward, next state) is stored in a buffer. A step-tuple contains all the required information to update $\hat{Q}$, but, unlike the Q-learning algorithm, $\hat{Q}$ is updated offline. In our implementation, the buffer is quiet large, equal to 100k state/action trials. The size of the buffer allows to store the interaction between the agent and the environment during the last 100 episodes.

The buffer is used aposteriori to construct random mini-batch of tuples for the learning step, without any consideration of actions order. As the minibatch is constructed randomly, it contains interactions observed during the last 100 episodes. Hence, a state/action value could be estimated several times, but at each time probably in a different context due to the update of others $Q(s, a)$ estimates. The network is updates each 4 steps. Hence, more the size of the minibatch is large more the probability than a state/action pair is evaluated several times increases.
It has been shown that this process allows to avoid action sequence correlation.

3.2 Target network

In the Q-learning algorithm, the update of The Q function is done with the $Sarsamax$ algorithm:

$$\hat{Q}(s_n, a_n) = (1 - \tau)\ \hat{Q}(s_n, a_n) + \tau\ target(r_n, s_{n+1})$$

where the target is computed from $\hat{Q}$:

$$target = r_n + \gamma\ \underset{a}{max}\ \hat{Q}(s_{n+1}, a)$$

The target is an estimation of $\hat{Q}(s, a)$ variation induced by the experience, using the immediate reward and the discounted estimate of the value of the greedy next action.

The goal of the learning algorithm is to learn the weights and bias (parameters) of the neural network model. In contrary to Q-table, the learning from a state/action pair experience will update the entire network parameters through backpropagtion, and so $\hat{Q}$ for all others state/action pairs.

One important feature in the DQN paper is that the Q-function used to compute the target ($\hat{Q}^{target}$) is a different function than the updated Q-function

($\hat{Q}^{local}$) during the training phase: The target function is fixed during the entire processing of a minibatch, such as the training has no impact on the target function. This feature avoids learning instabilities. Nevertheless, the target must take advantage of the learning to better fit the Q ground truth. It is updated but less ofently that the local network, using the local network. In our implementation, it is updated after the processing of each mini-batch.

So, the learning algorithm uses two "independent" networks: a local network and a target network. At each update of the local network during the training phase, only a small ratio of the target network is updated according to:

$$\hat{Q}_{n+1}^{target} = (1 - \tau)Q_n^{target} + \tau Q^{local}$$

As the target and the local networks have the same architecture, the above formula means that each parameter of the target is updated by the same parameter of the local network. This update process is called soft update.

3.3 Learning procedure

After defining above the experience replay principle, the local and the target networks, we can now describe the learning procedure. As said previously, the objective is to learn the Q-function which provides a value (float number) for a 38-D input signal build from the 37-D state signal and the 1-D action signal (4 possibles actions).

The learning aims at minimizing the loss between the the local and the target Q-function. It is thus a regression problem. We choose the $MeanSquareError$ as loss function, which is suitable for regression problems.

We describe hereafter the DQN algorithm using the stochastic gradient descent (SGD) optimizer.

Algorithm 1: Deep Q-Network algorithm

Result: Q-Network weights optimized
$\hat{Q}^{local}$ initialization;
$\hat{Q}^{target}$ initialization;
for $n \leftarrow 0$ **to** n **do**
  Choose randomly a SARS tuple in the replay experience buffer;
  For the state/action pair, compute the expected Q-value from $\hat{Q}^{local}$;
  For the next state, compute the target using $\hat{Q}^{target}$;
  Compute the loss between the expected Q-value and the target;
  For each parameter w_k, compute the loss gradient w.r.t. w_k by backpropagation;
  Update each w_k using the SGD algorithm;
  Soft-update of $\hat{Q}^{target}$ from $\hat{Q}^{local}$;
end

When using minibatch, the loss gradient w.r.t. w_k is the average of loss gradient computes from the set of tuples, the minibatch. Once the end of the minibatch processing, the last operation is the soft update of the target network.

4 Implementation and results

The implementation is based on three files, which handle:

- the agent (agent.py)
- the model (model.py)
- the learning procedure (main.py)

4.1 Model file

The model file contains a class describing the neural network architecture, which inherits from the nn.Module class. The initialization function contains the layers definition. The forward function define how to map input to output using a stack of layers connected each others with an activated function (RELU).

4.2 Agent file

The agent file contains classes allowing interactions with the environment and learn from it, according to the algorithm describes in previous sections.

- Agent class: choose action, execute an action, store SARS tuples in the replay buffer, learn from a minibatch.
- Replay buffer class: create a buffer, add an experience SARS tuple, select a random minibatch among the buffer.

4.3 The Main file

This file allows to define the learning procedure:

- Define and initialize the environment
- Train an agent according to the environment
- Save the agent model to disk

4.4 Hyperparameters and experimental result

The agent training requires to set hyperparameters, which are defined in a configuration file. They set the behavior to algorithm, and are related to different part of the algorithm. We summarize below the values we used to train the agent.

- Model architecture
  - Number of layers: 2
  - Number of neurons per layer: 64/32
- Optimizer
  - Learning rate: 5e-4
  - Batch size: 16
- Experience Replay Buffer size: 100000
- Reinforcement learning
  - Discounted reward coeficient γ: 0.99
  - Start exploration coefficient ϵ: 1
  - Exploration coefficient decay: 0.99
  - Minimum exploration coefficient: 0.01

We show on the figure 1 the evolution of the score during the training. The score is computed as the average cumulative reward over last 100 episodes. In our experiment, the environment is consider solved when the score is equal or above thirteen. With the 2 layers architecture, the agent training requires more are less 400 episodes.

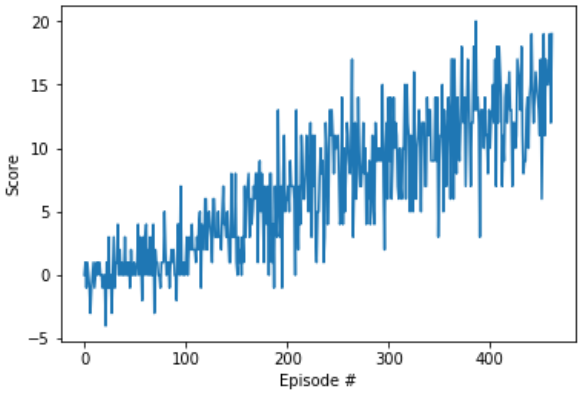

Figure 1: Evolution of the score during the learning (episode #).

5 Further improvements

Since 2015, several improvements have been proposed to increase performance training of the DQN algorithm. In particular, the first improvement I would implement would be Prioritized Experience Replay. In Experience Replay, there is no order of importance between experience tuples. The objective of this feature is to identify transition which leads to better learning, in order to use more ofently experiences which provide better performance improvements of the agent.

Others features have proposed such as:

- Double DQN to limit overestimation of action value
- Dueling DQN to estimate the state value without achieve the learning step

I would also extend the model architecture in order to process image or natural language.