Double deep Q-network with proportional prioritization

Alexandre Popoff

March 16, 2019

A double deep Q-network with proportional prioritization is implemented to train an agent to collect yellow bananas (reward = 1) while avoiding blue bananas (reward = -1). The environment (a continuous 37 dimensional space) is considered solved when the agent gets an average score of 13 over 100 consecutive episodes. The agent is able to solve the problem in less than 500 episodes, almost twice as fast as a DQN agent without prioritized experience replay.

1 A brief review of some ideas behind deep Q-networks

1.1 Learning from the environment

An agent starts in a state S_0 and then takes an action A_0 based on the information available up to the current state S_0. After that, the environment *reacts* and provides useful feedback to learn from, namely a reward R_1; then the agent goes into a new state S_1. The agent can store the transition (S_0, A_0, R_1, S_1) in memory and can use it later to take better actions.

1.2 Choosing the best action

The agent tries to maximize the sum G_0 of all the rewards that it could get from the environment. Since there is typically some randomness in the environment, the agent can only try to maximize the expected value, knowing its initial state (here S_0). If the agent can estimate those future rewards, it can choose the best action. Mathematically, the action A_0 is such that $q_\pi(S_0, A_0)$ is maximum, where $q_\pi(s, a)$ is the value of a state s followed by an action a for a given mapping π (a policy) which says which action to take in a given state. The real number $\pi(a|s) \in [0, 1]$ is the probability to choose action a knowing the current state s.

1.3 Q-learning and deep Q-learning

In this setup, a Q-learning agent can learn from a sequence of transitions

$$(S_0, A_0, R_1, S_1), (S_1, A_1, R_2, S_2), ...$$

The transitions are stored in the agent's memory and will be used to train it. Q-learning can solve many tasks involving discrete states and actions. Typically, we can use a tabular representation of $Q(s, a)$ and can choose an optimal action from a given state s. When the states are continuous (for example, the position and the velocity of a car), we can always discretize the space in order to

build a tabular representation of $Q(s,a)$. However the size of this table will explode as we approach the continuous case. Q-learning algorithm can still be used if we manage to approximate $Q(s,a)$ with another function $\hat{Q}(s,a,\theta)$ that depends on a hopefully small number of parameters θ. For example, we can use a neural network with the weights θ to map a state s to the four Q-values $Q(s,a_i),\ i=1,2,3,4$; i.e. $s \mapsto (Q(s,a_1),Q(s,a_2),Q(s,a_3),Q(s,a_4))$ (in the banana collector problem, there are four possible actions a_1,a_2,a_3,a_4).

1.4 Replay buffer

The replay buffer stores the transitions $(S_t,A_t,R_{t+1},S_{t+1})$ and can generate a sample from it following a given distribution (uniform in the original DQN algorithm).

As stated in [3], there are some conditions under which learning fails (with function approximation such as neural networks). The replay buffer (that samples random past transitions) tries to address this issue. A uniform distribution was used in the original DQN algorithm. The present code uses a distribution where the probability of sampling a transition j is proportional to the absolute value of the TD-error δ_j (i.e. potentially, the agent could learn a lot more from "surprising" examples with high TD-errors). Prioritized replay introduces bias that is corrected by using importance-sampling weights w_i for the current buffer size N

$$w_i = \left(\frac{1}{N}\frac{1}{P(i)}\right)^{\beta}$$

with β increasing linearly from 0.7 to 1 at the end of training (600 episodes). The squared error is clipped between -1 and 1. A transition j is sampled with probability $P(j)=p_j^{\alpha}/\sum_i p_i^{\alpha}$ with $\alpha=0.8$. In the code, the weights w_i are implemented in the loss function.

The sum tree class used to sample a transition $(S_t,A_t,R_{t+1},S_{t+1})$ with a probability $P(j)$ comes from AI-blog/SumTree.py.

The replay buffer is adapted from deep-reinforcement-learning/dqn/solution/dqn_agent.py.

2 Neural network

The neural network is a plain fully connected neural network with two hidden layers with ReLU activation function. Hidden unit numbers are $n_1=n_2=128$.

- input (from a continuous 37 dimensional space)
- fully connected layer (128 neurons), ReLU activation
- fully connected layer (128 neurons), ReLU activation
- a terminal layer with four outputs $Q(s,a_i),\ i=1,2,3,4$.

With these $Q-$values, the agent selects an action using an $\epsilon-$greedy policy (i.e. with probability $1-\epsilon$, it takes an optimal action and a random action otherwise).

3 Algorithm

The double deep Q-network with proportional prioritization is presented in [2]. In the present code, the update (line 14) is implemented with a soft update (i.e. $\hat{\theta} \leftarrow \tau\theta + (1-\tau)\hat{\theta}$ with $\tau = 10^{-2}$) so that θ and $\hat{\theta}$ are updated at the same time. Since δ_j can almost vanish, line 11 becomes $p_j \leftarrow |\delta_j| + \epsilon$ with $\epsilon = 10^{-4}$ so that j can still be sampled.

Algorithm 1 Double DQN with proportional prioritization

input : minibatch k, replay period K and size N, exponents α and β, budget T.

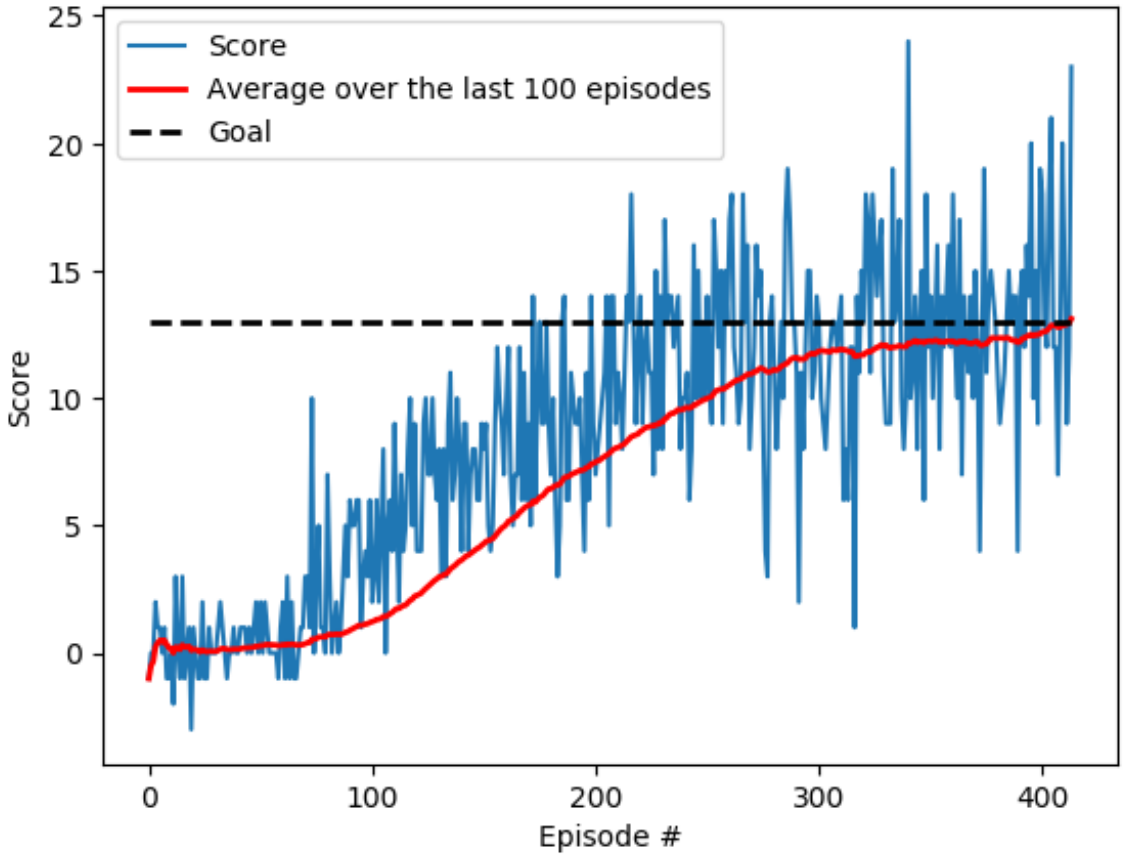
1 Initialize replay memory $\mathcal{H}$, $p_1 = 1$
2 Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
3 **for** *t=1* **to** *T* **do**
4 Observe S_t, R_t, γ_t
5 Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
6 **if** $t \equiv 0 \mod K$ **then**
7 **for** $j = 1$ **to** *k* **do**
8 Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
9 Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i \ w_i$
10 Compute TD-error $\delta_j = R_j + \gamma_j \hat{Q}(S_j, \text{argmax}_a \ Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
11 Update transition priority $p_j \leftarrow |\delta_j|$
12 **end**
13 Update weights θ (gradient descent on unbiased loss)
14 From time to time copy weights into target network $\hat{\theta} \leftarrow \theta$
15 **end**
16 Choose action $A_t \sim \pi_\theta(S_t)$
17 **end**

4 Memory

The buffer size is $\approx 10^4$ (typically 10^6 in research paper) and the batch size is 64. As mentioned in [4], the buffer size is a hyperparameter that should hurt performance if it is too small or too large.

5 Plot of rewards

The environment is solved in 414 episodes (between 400 and 500 on average). The noisy blue line represents the score after each episode, the dashed line is the goal (a score of 13) and the red line is the average score over the last 100 episodes.



6 Further works and improvements

A double DQN agent should be able to solve the same task from a raw pixel input. In order to do so, a convolutional neural network may replace the plain fully connected network used in this code. Improvements of the original DQN algorithm are found in [1]:

- (Double Q-learning) (Implemented here) $Q-$learning tends to overestimate the value of random positive rewards in stochastic environment. This bias slows learning. The bootstrap target can be modified to address this issue.
- (Prioritized replay) (The double DQN with proportional prioritization algorithm is implemented here) The idea is to modify the distribution of the replay buffer so that "surprising" transitions are seen more often. Generally, experience replay reduces variance by reducing the correlation among the samples used during training.
- (Dueling networks) The network is now divided into streams that provide separate estimates of the value and advantage functions: $Q^\pi(s,a) = V^\pi(s) + A^\pi(s,a)$. The output of the neural network is a set of Q values (one for each action) so that this architecture can be used without modifying the underlying reinforcement learning algorithm. For example, it can improve the agent's performance in the DQN algorithms (with prioritized experience replay or uniform experience replay).

- Multi-step learning (Sutton 1988) Define the truncated $n-$ step return from a given state S_t as $R_i^{(n)} := \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1}$. The DQN algorithm tries to minimize the modified loss $(R_t^{(n)} + \gamma_t^{(n)} \max_{a'} q_{\bar{\theta}}(S_{t+n}, a') - q_\theta(S_t, A_t))^2$.

- (Distributional RL) Instead of learning Q ($\approx$ expected returns), we can learn the distribution of returns. A Bellman's equation and a variant of $Q-$learning are derived.

- (Noisy Nets) The idea is to add a noise to the standard linear activation $y = b + \mathbf{W}x$. Over time, the network can learn to ignore the noise. This should improve learning when $\epsilon-$greedy policies show some limitations when many actions must be executed to collect the first reward.

References

[1] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining Improvements in Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1710.02298, Oct 2017.

[2] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized Experience Replay. *arXiv e-prints*, page arXiv:1511.05952, Nov 2015.

[3] Sebastian Thrun and Anton Schwartz. Issues in Using Function Approximation for Reinforcement Learning. 1993.

[4] Shangtong Zhang and Richard S. Sutton. A Deeper Look at Experience Replay. *arXiv e-prints*, page arXiv:1712.01275, Dec 2017.