Report

December 3, 2019

1 Report

```
In [ ]:
```

1.1 The DQN algorithm

DQN (Deep Q Networks) are a particular variant of Q-learnings from reinforcement learning.

Q-learning is a model-free reinforcement learning algorithm. The goal of Q-learning is to learn a policy(a function mapping states as recieved from the enviroment to actions performed in said given state), which tells an agent what action to take under what circumstances. It does not require a model of the environment, and it can handle problems with stochastic transitions and rewards, without requiring adaptations.

DQN is an appoarch to Q learning where neural networks are used as a function approximater to the Q table.

Here a Q table is essentially a function mapping

$Q : State \times Action \rightarrow Rewards$

Let Q be our Q function / table

Let π be the policy the agent is following

Let $a \in Actions, s \in States$

The goal is to find the optimal policy as is defined by

$\pi(s) = argmax_a Q(s, a)$

1.2 Algorithm and Training

A property of Q learning is the fact that the framework satisfies the Bellman equations

$Q^{\pi}_{t+1}(a, s) = r_t + \gamma Q^{\pi}_t(s_t, \pi(s_t))$

where

γ is a discount factor $0 < \gamma < 1$

r_t is a reward recieved from the enviroment at time t

The algorithm starts of with a randomly intialized neural network as a policy

The agent then acts in the enviroment according to this policy and collects experiences in the form

$\langle s_t, a_t, r_t, s_{t+1} \rangle$

The agent acts ϵ -greedily. This means it will with probabilty $\frac{\epsilon}{|Action|}$ per action the agent will chose an action uniform randomly and $1 - \frac{\epsilon}{|Action|}$ will act according to the policy

After collecting enough experiences the algorithm will update the parameters of the Q-function using its memory store randomly sampling mini-batches of experience using $r_t + \gamma Q_t^{\pi}(s_t, \pi(s_t))$ with fixed weights as a target and train a separate copy of the network using squared loss.

2 Implementation details

2.1 Enviroment

The state for this specific game is defined by a vector of real numbers represent various qunatities related to the enviroment(these numbers are provided by the Unity engine).

https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Learning-Environment-Examples.md#banana-collector

The environment state space has 37 dimensions and four discrete actions are available to the agent. A reward of +1 and -1 is provided for collecting respectively a yellow banana and a blue banana.

2.2 Neural Net architecture

A vanilla two layer feedforward neural network was used.

The 1st hidden layer had 128 units The 2nd hidden layer had 64 units

No output activations were used.

The size of the output is $|Actions|$

2.3 Experience sampling

We store up to a maximum of 10 000 of experience tuples in what is called a "replay buffer". During training time the experiences are a sampled randomly uniformly from the buffer to make mini-batches of size 64

2.4 Exploitation vs Exploration

The agent acts ϵ -greedily. In the beginning we would like to encourage exploration as the agent has no reasonable policy to exploit. To encourage initial exploration we set $\epsilon_{start}, \epsilon_{start} = 0.99, 0.01$ We gradually decrease ϵ throughout each episode in the following way $\epsilon_{episodeT+1} = max(\epsilon_{end}, \epsilon_{decay} \times \epsilon_{episodeT})$ where $0 < \epsilon_{decay} < 1$

3 Results

The agent took 539 epsiodes to surpass a average score of 15

4 Improvements

1) We could potentially use Huber loss instead of the squared loss. The Huber loss is definded by $\mathbb{L} = \frac{1}{|episodes|} \sum_{\langle s,a,r,s\prime \rangle \in episodes} \mathbb{L}(\delta)$

where

$\mathbb{L}(\delta) = \frac{1}{2}\delta^2 \quad if \quad |\delta| \geq 1$

$|\delta|\frac{1}{2} \quad if \quad |\delta| < 1$

2) Prioritzed Experience replay . Here we sample the trajectories from the Replay buffer according to a probability distribution based on the TD-Errors related to the particular experience tuple.

3) The DQN paper is prone to over estimating the values of the Q functions . At the beginning of the training there is not enough information encoded to the approximate Q function(i.e. the neural network) for accurate predictions. Hence, taking the maximum q value over high variance samples(as dictated by the Bellman equations) will lead to general over estimation of the Q values .

In order to solve this over estimation problem we could try implement Double DQN. Double DQN essentially decomposes the max function in the Bellman equation into separate action selection and Q value estimations

Two networks are used (the target and local set of parameters), one set of parameters for estimating the Q value and another set of parameters for for selecting the best action.

```
In [ ]:
```