Project 1: Navigation

Adithya Subramanian
Deep Reinforcement Learning Nanodegree, Udacity
12/10/2018

Implementation description :

1. Dqn_agent.py:
    a. Agent class contains the information about the current state representation, allowed actions, action-value functions local & target. The RMSprop optimizer is also defined in this class and replay buffer object is initiaized in this class.
    b. The step function of the agent class adds the current state, action, reward and next state pair into the replay buffer by calling its add function and updates the weights of q-networks every 4 steps if the size of the buffer is greater than the batch size.
    c. The act function returns the optimal actions as per the current ϵ-greedy policy
    d. The learn functions is responsible for computing the mean squared error between the expected Q-value and target Q-value and the soft-update function for the target network update is also called in this function.
    e. The soft_update function updates the target weights using τ parameter and local network weights.
    f. The class ReplayBuffer is the class definition for the buffer memory it is initialized with a deque of pre-defined buffer size which will store a named tuple.
    g. The add function in ReplayBuffer adds the newly visited state transition to the buffer memory.
    h. The sample function samples a complete transition using a uniform sampling technique.
2. Model.py :
    a. The model used for transforming the state information vector into the action-value vector is written in the class DDDQN_network
3. Navigation.ipynb :
    a. The dqn function runs multiple episodes of the agent's interactions with the environment for a certain number of transitions. The network stores the average reward collected by the agent over the 100 episodes. Once the

agent accumulates over the 13+ reward in 100 episodes the environment is considered to solved and model's weight are saved.

Learning algorithm :

The RMSprop optimizer was used to update the weights of the model. The Adam optimizer was found to be unstable during the experiment whereas RMSprop optimizer showed constant improvement in reward accumulation. Weight decay is also added so that the network doesn't overfit to a certain type of transition and generalizes well across multiple transitions. Learning rate of 3e-4 was used in the RMSprop optimizer, learning rate scheduling was also tried during the course of experimentation but it was observed that regardless of this the agent was able to solve the environment in a similar number of episodes but it was interesting to observe that while using learning scheduler the initial rewards for transition were very unstable.

$$L_i(\theta_i) = \mathbb{E}_{s,a,r}\left[(\mathbb{E}_{s'}[y|s,a] - Q(s,a;\theta_i))^2\right]$$
$$= \mathbb{E}_{s,a,r,s'}\left[(y - Q(s,a;\theta_i))^2\right] + \mathbb{E}_{s,a,r}[\mathbb{V}_{s'}[y]].$$

Fig 1. Loss function

The loss function between the target and expected Q - value can be seen in Fig 1.

$$Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma Q(S_{t+1}, \underset{a}{\operatorname{argmax}}\, Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}'_t)\,.$$

Fig 2. Computing the target Q-value

Here E represents the expected value of the squared difference between the target and the expected Q value for action a in state s. The parameters for the local network is defined by θ and the target network is defined by θ'. The target Q-value is computed the equation in Fig 2. The γ was set to 0.99 similar that of the DQN paper.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(Y_t^{\text{Q}} - Q(S_t, A_t; \boldsymbol{\theta}_t))\nabla_{\boldsymbol{\theta}_t} Q(S_t, A_t; \boldsymbol{\theta}_t)$$

Fig 3. Weight update

The weights of the network are updated according to Fig 3.

The network used to compute the action-value function takes the current state representation as the input and computes the action-value function for that state. The network contains two parts one is the auxiliary part and the second is branch part. The auxiliary part of the network is built similar to the resnet architecture, it is 6 layers deep with two bottleneck connections, along with batch normalization and Exponential linear unit activation. The resulting tensor from the auxiliary network is then fed to these branches as an input where one of the branches calculated the state value function (V) and the other branch calculate the advantage function (A) which calculates the normalized advantage of carrying out an action in the input state. The resulting state value function and action value function are combined as shown in Fig 4.

$$Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \frac{1}{|\mathcal{A}|}\sum_{a'} A(s,a';\theta,\alpha) \right).$$

Fig 4. Compting the Action-value function

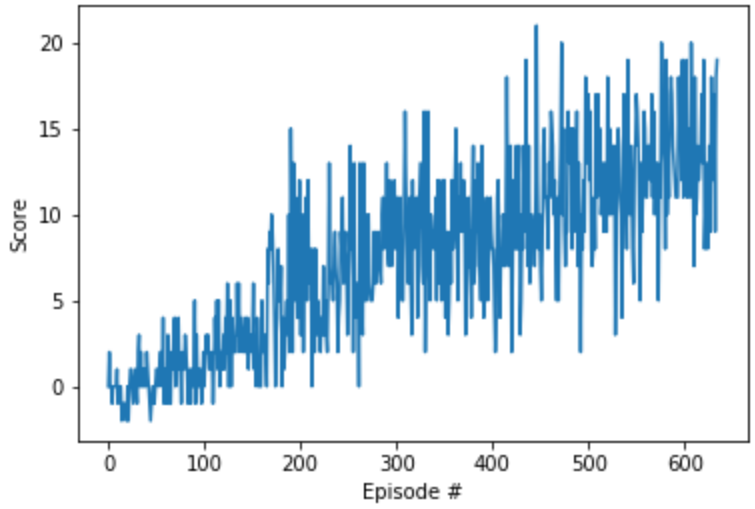
In Fig 4. Alpha and beta represent the parameters of their branches respectively.

The target network parameters are updated in a soft updated manner using equation shown in Fig 5. The τ parameter was set to value of 1e-3.

$$\theta_{target} = \tau * \theta_{local} + (1 - \tau) * \theta_{target}$$

Fig 5. Soft-update of target network parameters

Reward Plot:



Future work:

1. Implementing Rainbow for solving the task.
2. Using images directly instead of vector state representation.

THANK YOU