Lecture Notes

Deep Reinforcement Learning

When the state-space is relatively small, e.g. a 3x3 grid, or even a few thousand states etc., it is easy to visit each and every state and estimate its Q-value. However, when the state-space is large, visiting every space may not be possible. Also, in many cases like in training a robotic arm to play table tennis, there can be infinite number of state-action pair. So, the classical approach of estimating the Q-value by visiting each state-action pair wouldn't work. Using deep learning, instead of finding the Q-value for every state, you learn the Q-function or policy using function approximation.

Generalizability in Deep RL

In deep reinforcement learning, you try to **approximate the Q-function**. You train the model, that is, the action-value function (or the policy or the value function) based on the 'seen' states and the model should be able **to 'generalize' well on the 'unseen' states**. For example, say you are training an autonomous car. The car will be trained in certain traffic conditions and would be exposed to certain states, but it should be able to perform in new, unseen states as well. This property of "generalization" is valid for any (well-trained) model in machine learning.

You will use a neural network for approximating these functions: state-value, action-value, policy etc. You can use other algorithms as well: decision trees, SVM etc., but in practice, the complexity of most RL tasks require deep neural networks.

Deep Q Learning

When you train a machine learning model, such as linear regression, you assume that the data points are **independent and identically distributed** (IID). For example, say you are using regression to predict the price of a house using its area, so you have (area, price) tuples as your training data. You assume that **each data point is independent** of each other (the price of a house does not affect that of another) and that all the data points are identically distributed. An **identical distribution** means that every time you draw a data point, whether you draw the first or the 100th data point, the probability of getting a particular data point is the same.

The samples are not independent in the case of RL because **the next state is usually highly dependent on the previous state and action**. For example, let's say an agent is trying to learn to play an Atari game. The state is represented by the screenshot of the game at any point. As you move from one screenshot to the next, there will be little change in pixel values. Hence, the next state (screenshot) is **dependent** on the previous one.

The other problem is that of **identical distribution**. We said that an **identical distribution** means that every time you draw a data point, whether you draw the first or the 100th data point, the probability of getting a particular data point is the same.

Now, this is not the case in deep RL - the probability of a (state, action, reward) triplet appearing changes with 'time' (i.e. episodes). That is, say you are creating a training set of 1 million (state, action, reward) triplets. Towards the start, the agent has not learnt much, so the probability of an (s, a, r) triplet will be very different from when the agent has learnt much more during the end of the game.

Replay Buffer

Before training the neural network, to learn the state-value function, you need to have a dataset that can be used for training. However, in reinforcement learning tasks, you generally don't have the dataset. You are required to create a dataset of the form (s,a,r,s'). As you know the equation for Q-learning is:

$$\boldsymbol{Q(s,a) \coloneqq Q(s,a) + \alpha[r + \gamma\big((max_a Q(s',a) - Q(s,a)\big)]}$$

The **predicted** Q-value by the model, using (s,a) will give Q(s,a). Further, next state s' with all possible will generate the target Q(s',a).

When in state s, the agent takes an action a. The environment returns the new state s' and the reward r. This sample is stored as <s,a,r,s'> in memory. Again, in the new state s', the agent performs another action, gets a reward and a new state from the environment. You choose an action according to the epsilon-greedy policy. With ϵ probability, you do exploration while with 1−ϵ you choose the action corresponding to the highest Q-value. You store these experiences in the memory (replay buffer) in the form of <s,a,r,s'>.

These **samples** generated by the interaction of the agent with the environment are collectively known as **experience**. You represent experience as a 4-tuple <state, action, reward, next_state>. The **reward here is the immediate reward** you get from the environment. These encountered experiences are stored in the memory. And this **memory** is called the **replay buffer**.

A neural network is trained in batches in which the data points are random (IID). But the agent's experience is generated **sequentially**, i.e. you go from one state to another state in a sequential, dependent fashion. So, to make a batch from a sequential process, you store them in memory and take **random samples** of batch size to train the network.

When to stop training?

The predicted and the target values gets improved over time as the Q-value generated using neural network gets better after each gradient update. This will give better rewards with each time step. For example, let's say that the predicted Q-value you get from the network is 20 and the expected value is 30. After training for a few time-steps, you will get the predicted value as 27, but the target value could also increase to 35.

This type of situation will come in the initial stage of training, but after training for sufficient time, the network will stabilise. The parameters of the neural network will not change significantly, and that's when you can stop training further. The predicted value Q(s,a) will be close to the target value Q(s',a) and they will not significantly improve over time during training.

Training pseudo code

The deep Q-learning pseudocode is as follows:

- Initialise replay memory D to a capacity of N (if N=2000, then it can store 2000 experiences)
- Initialise the action-value function Q (i.e. the neural net) with random weights
- Total number of episodes is M
- Each episode is of length (time steps) T
- For 1 to M episodes, do:
    - For 1 to T time steps, do:
        - **Generate an experience** of the form <s,a,s',r>
            - With probability epsilon, select a random action **a**
            - Else select $a = arg\ max_a Q(s', a, \Theta)$
            - Go to the next state s'
            - Set next state as the current state
            - Store the experience in the replay memory D
        - **Train the model** on (say) 100 samples (batch size) randomly selected from the memory
            - Randomly sample transitions (s,a,s',r) of batch size from replay buffer
            - Calculate the target $(y) = r + max_a Q(s', a)$
            - Calculate the Q-value for this state-action pair (s,a) as predicted by the network
            - Train the model to minimize the 'squared error': $(Q(s,a) - y)^2$

Note: You are generating only one experience at every timestep while training on (say batch size = 100) 100 experiences at every timestep. This is so because it has been assumed that you must have generated 100

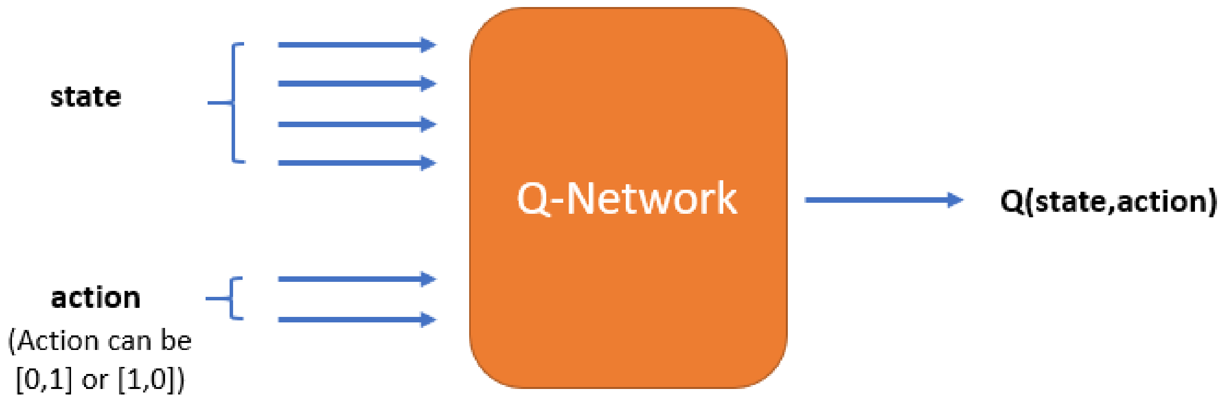
experiences previously and stored in memory. Training will start only after you have your 'first' 100 experiences.

The Q-function is being updated at the end of every episode. Thus, you choose better actions after each update to generate the experiences. This will result in more optimal actions and next states, which will further improve the learning process (i.e. the policy). Both the input and label are changing with time which leads to a dynamic dataset.

Architectures of Deep Q Learning: Architecture 1

The input to the NN is both state and action. For choosing the $max_a Q(s', a)$, for a particular state, you have to input every possible action, calculate its Q-value and choose the action for which Q-value is maximum.

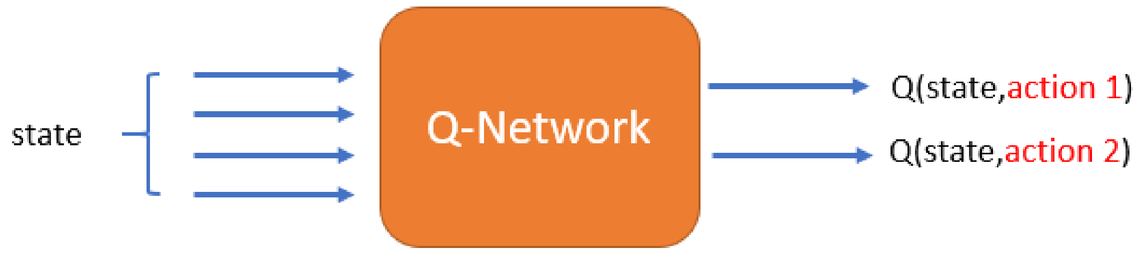
Disadvantage:
Suppose the state is represented by a vector of size **4** and for every possible state, there can be **2** actions. To find $max_a Q(s', a)$, you have to do feedforward **2** times, one for each action and find the Q-value. Then take the action for which Q-value is the maximum.



Q-network

Architectures of Deep Q Learning: Architecture 2

The input to the NN is only the state and its output is the Q-value of every possible action. Here, the size of the input vector will be **4** and the output size will be **2**.



Q network

Advantage:

The advantage here is that, just by giving state as the input to the NN, you will get Q(s, a) for each action, so you have to run the NN just once for every state. Take the action for which Q(s, a) is the maximum.

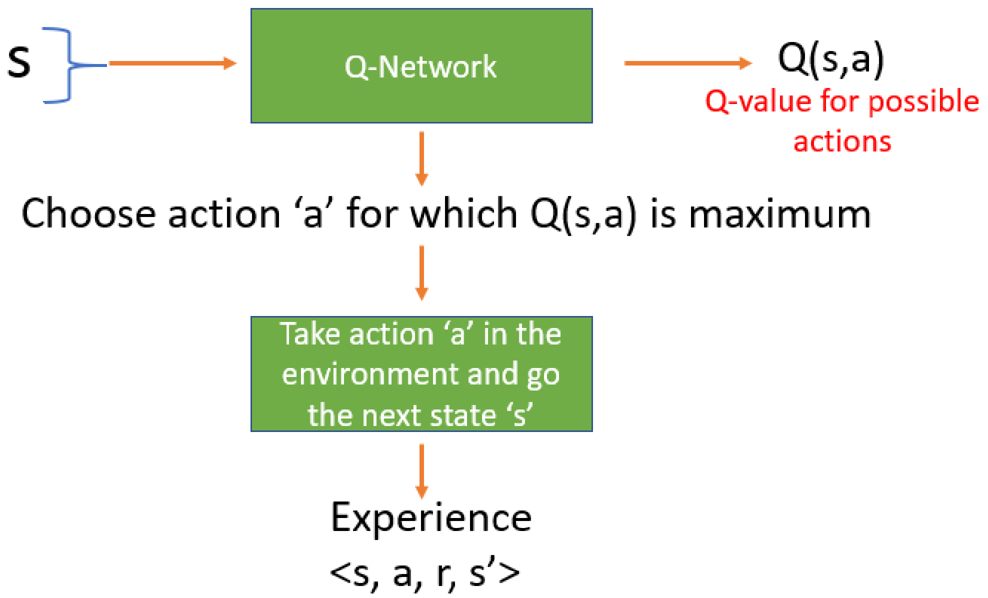
Note: Here, the gradient should flow only from the action you took and not from all the possible Q values.

DQN Architecture II - Visualisation

Let's see to train the second architecture.

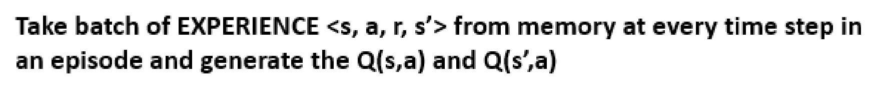
- First, generate the experience using epsilon-greedy policy
- Store the experience in the memory

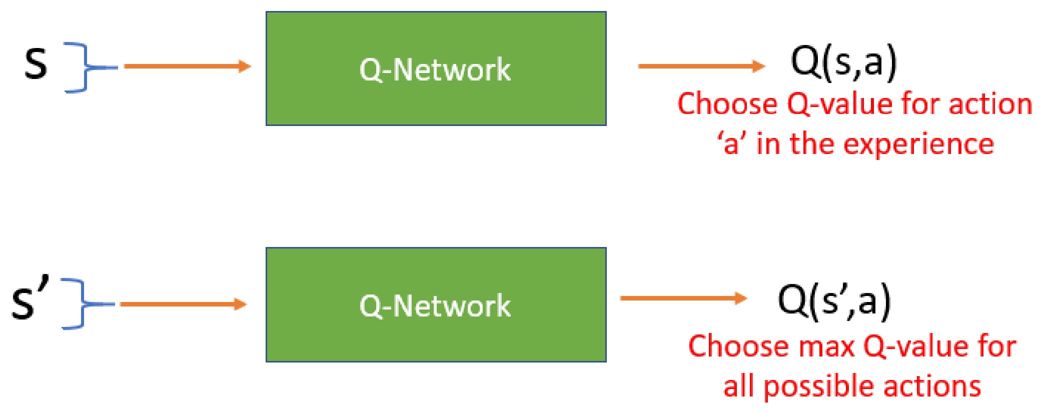
Generate experience at every timestep in an episode. If value of random number generated is less than epsilon, choose random action, otherwise generate action 'a' for which Q(s,a) is maximum.


s
Q-Network
Q(s,a)
Q-value for possible actions
Choose action 'a' for which Q(s,a) is maximum
Take action 'a' in the environment and go the next state 's'
Experience
<s, a, r, s'>

Generating experience

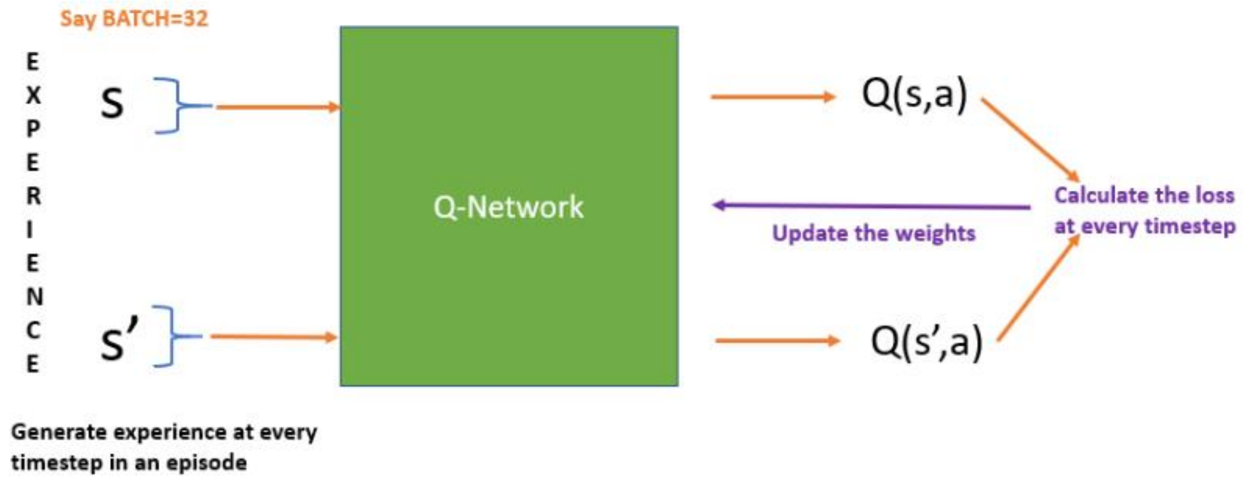
- Find the Q(s, a) and Q(s', a) from the Q-Network.


Take batch of EXPERIENCE <s, a, r, s'> from memory at every time step in an episode and generate the Q(s,a) and Q(s',a)


S
Q-Network
Q(s,a)
Choose Q-value for action 'a' in the experience
s'
Q-Network
Q(s',a)
Choose max Q-value for all possible actions

Generate Q(s,a) and Q(s',a)

- Calculate the predicted value Q(s,a) and the target value: $r + \ arg\ max_a Q(s', a)$
- Now, train the model


Say BATCH=32
EXPERIENCE
S
s'
Q-Network
Q(s,a)
Q(s',a)
Update the weights
Calculate the loss at every timestep
Generate experience at every timestep in an episode

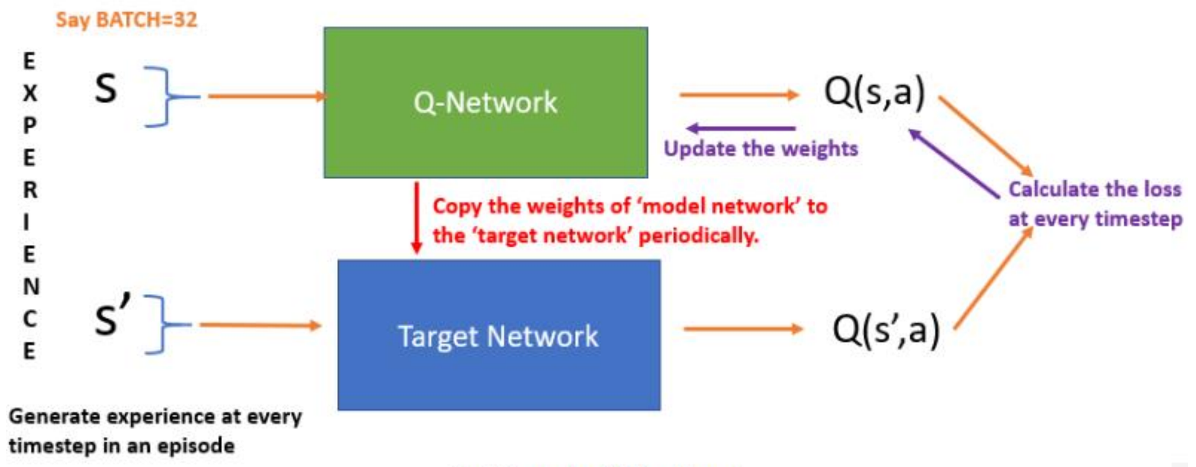
Train the model

- Use the trained Q network to generate another experience and continue the training.

Double DQN - A DQN Variation

During training, until an episode ends, only the parameters of the Q-network are updated, just like you do for a DQN architecture. The parameters of the target network are not updated. If you are constantly shifting the predicted and target Q-values to update the network, it can become destabilized by falling into feedback loops between the predicted and target Q-values.

To avoid this problem, you **keep the target value fixed** for the entire episode. To calculate the target value ($r + arg\ max_a Q(s', a)$) for each sample, the Q(s',a) is calculated using the target network. So, the target value remains fixed during the entire episode and you try to improve the predicted value using the 'Q-Network'. The learning in such a case will be more stable because the 'goal post' is not shifting after every timestep. After the episode ends, the weights of the 'Q-Network' are copied to the 'Target Network' and a new **target value** is calculated for the next episode.


Say BATCH=32
EXPERIENCE
S
S'
Q-Network
Target Network
Q(s,a)
Q(s',a)
Update the weights
Copy the weights of 'model network' to the 'target network' periodically.
Calculate the loss at every timestep
Generate experience at every timestep in an episode

Training using 2 Q-network

There are two deep-Q networks in this architecture. In DQN, you update the Q-network after every timestep in an episode. However, in double DQN you only update the main network (also known as the 'Q-network') after every timestep.

Apart from the 'Q network', there is another neural net called the 'target network'. Recall that you calculate Q(s,a) and Q(s',a) using the Q-network in DQN. However, in double DQN you calculate Q(s,a) using the 'Q-Network' and Q(s',a) using the 'Target Network'.