Navigation Project

Q Learning

Let's say we know the expected reward of each action at every step. This would essentially be like a cheat sheet for the agent! Our agent will know exactly which action to perform.

It will perform the sequence of actions that will eventually generate the maximum total reward. This total reward is also called the Q-value and we will formalise our strategy as:

$$Q(s,a) = r(s,a) + \gamma \max_a Q(s',a)$$

The above equation states that the Q-value yielded from being at state *s* and performing action *a* is the immediate reward r(s,a) plus the highest Q-value possible from the next state *s'*. Gamma here is the discount factor which controls the contribution of rewards further in the future.

Q(s',a) again depends on Q(s",a) which will then have a coefficient of gamma squared. So, the Q-value depends on Q-values of future states as shown here:

$$Q(s,a) \rightarrow \gamma Q(s',a) + \gamma^2 Q(s'',a) \dots\dots\dots \gamma^n Q(s''^{\dots n},a)$$

Adjusting the value of gamma will diminish or increase the contribution of future rewards.

Since this is a recursive equation, we can start with making arbitrary assumptions for all q-values. With experience, it will converge to the optimal policy. In practical situations, this is implemented as an update:

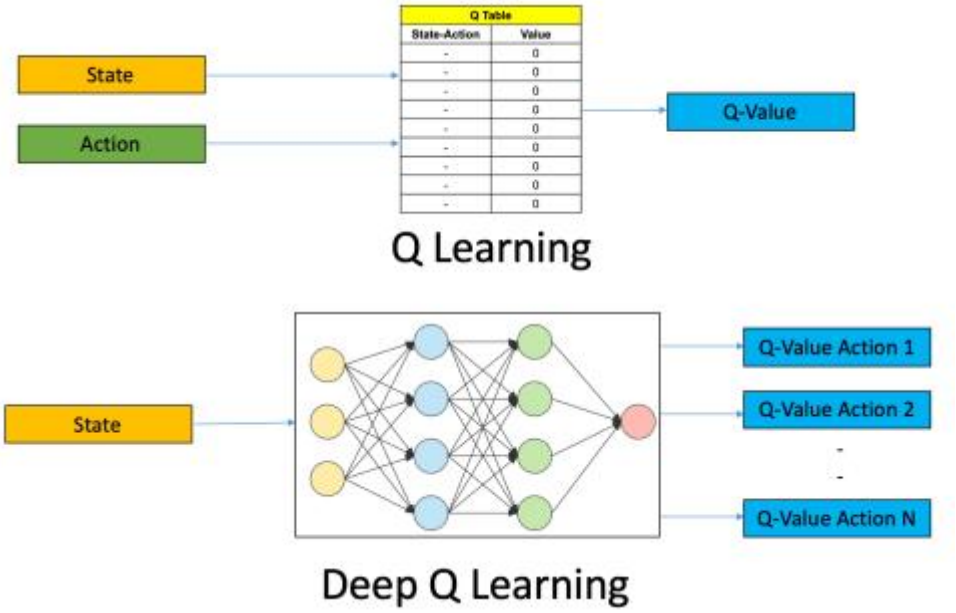
$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\right]$$

where alpha is the learning rate or step size. This simply determines to what extent newly acquired information overrides old information.

Deep Q-Networks

In deep Q-learning, we use a neural network to approximate the Q-value function. The state is given as the input and the Q-value of all possible actions is generated as the

output. The comparison between Q-learning & deep Q-learning is wonderfully illustrated below:



So, what are the steps involved in reinforcement learning using deep Q-learning networks (DQNs)?

1. All the past experience is stored by the user in memory
2. The next action is determined by the maximum output of the Q-network
3. The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem. However, we do not know the target or actual value here as we are dealing with a reinforcement learning problem. Going back to the Q-value update equation derived fromthe Bellman equation. we have:

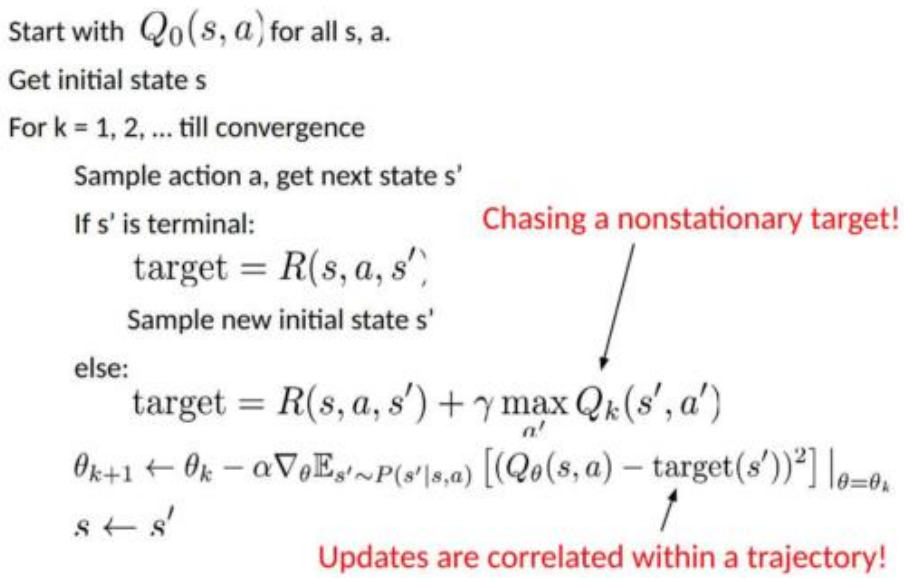
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

The section in green represents the target. We can argue that it is predicting its own value, but since R is the unbiased true reward, the network is going to update its gradient using backpropagation to finally converge.

Challenges in Deep RL as Compared to Deep Learning

So far, this all looks great. We understood how neural networks can help the agent learn the best actions. However, there is a challenge when we compare deep RL to deep learning (DL):

- **Non-stationary or unstable target:** Let us go back to the pseudocode for deep Q-learning:



As you can see in the above code, the target is continuously changing with each iteration. In deep learning, the target variable does not change and hence the training is stable, which is just not true for RL.

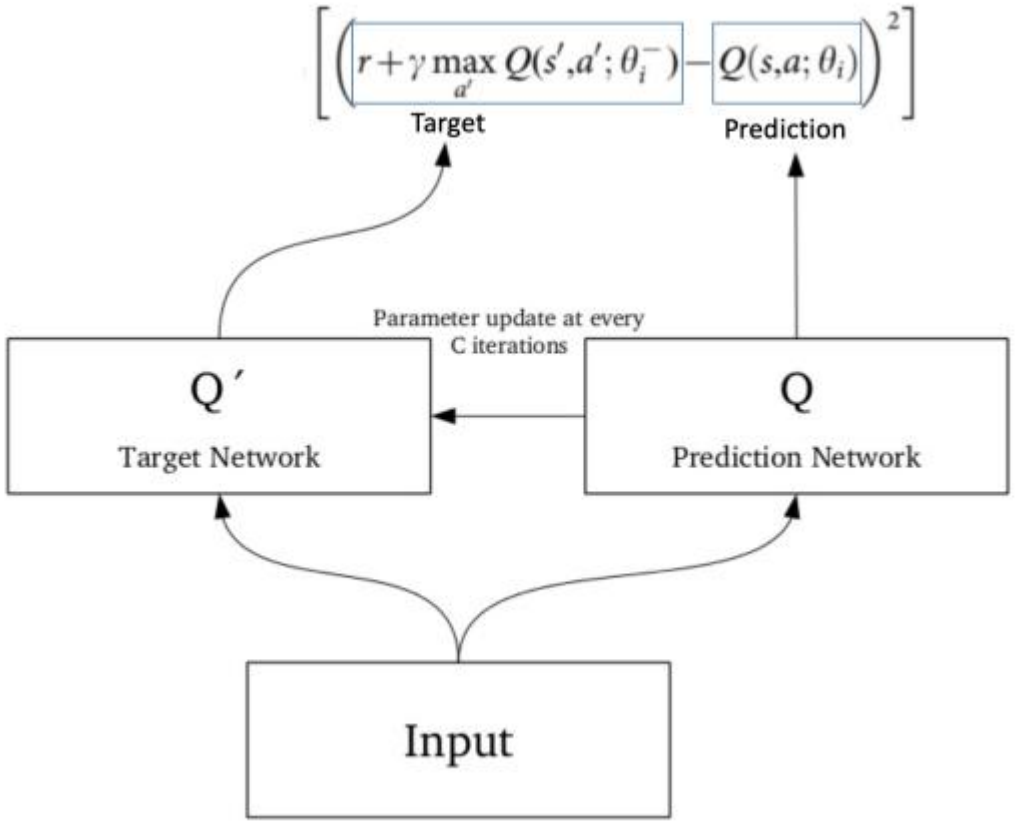
To summarise, we often depend on the policy or value functions in reinforcement learning to sample actions. However, this is frequently changing as we continuously learn what to explore. As we play out the game, we get to know more about the ground truth values of states and actions and hence, the output is also changing.

So, we try to learn to map for a constantly changing input and output. But then what is the solution?

1. Target Network

Since the same network is calculating the predicted value and the target value, there could be a lot of divergence between these two. So, instead of using 1one neural network for learning, we can use two.

We could use a separate network to estimate the target. This target network has the same architecture as the function approximator but with frozen parameters. For every C iterations (a hyperparameter), the parameters from the prediction network are copied to the target network. This leads to more stable training because it keeps the target function fixed (for a while):



2. Experience Replay

To perform experience replay, we store the agent's experiences
— et=(st,at,rt,st+1)

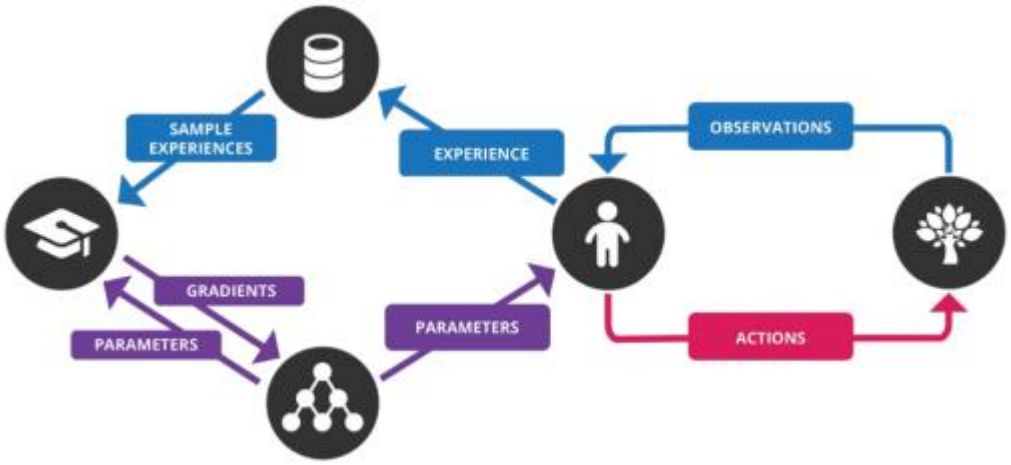
What does the above statement mean? Instead of running Q-learning on state/action pairs as they occur during simulation or the actual experience, the system stores the data discovered for [state, action, reward, next_state] – in a large table.

Let's understand this using an example.

Suppose we are trying to build a video game bot where each frame of the game represents a different state. During training, we could sample a random batch of 64 frames from the last 100,000 frames to train our network. This would get us a subset within which the correlation amongst the samples is low and will also provide better sampling efficiency.

Putting it all Together

The concepts we have learned so far? They all combine to make the deep Q-learning algorithm that was used to achive human-level level performance in Atari games (using just the video frames of the game).



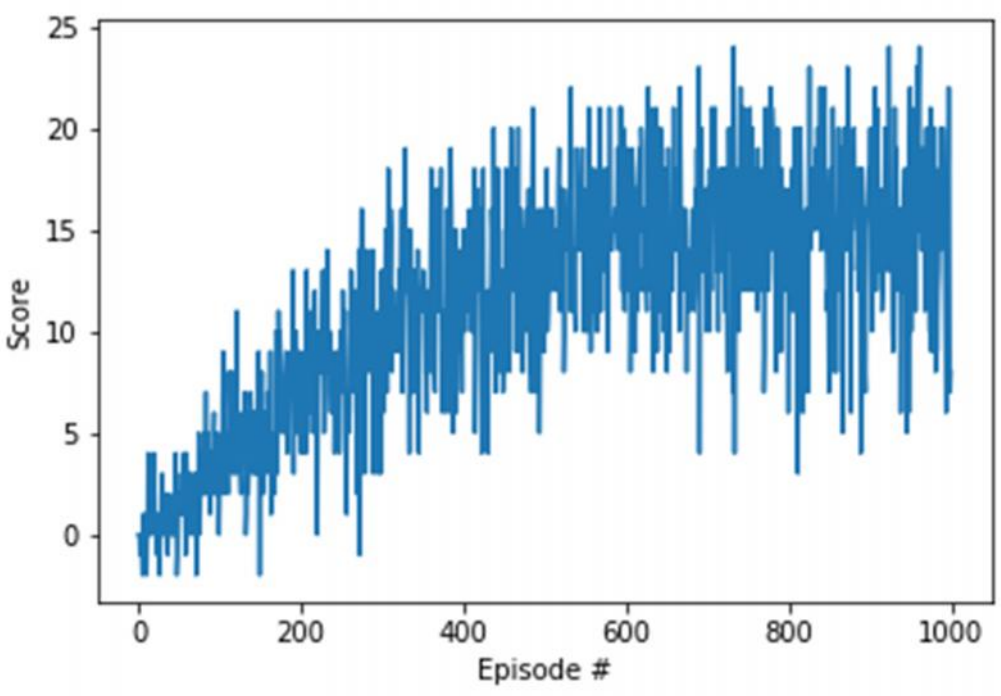
I have listed the steps involved in a deep Q-network (DQN) below:

1. Preprocess and feed the game screen (state s) to our DQN, which will return the Q-values of all possible actions in the state
2. Select an action using the epsilon-greedy policy. With the probability epsilon, we select a random action *a* and with probability 1-epsilon, we select an action that has a maximum Q-value, such as a = argmax(Q(s,a,w))
3. Perform this action in a state *s* and move to a new state *s'* to receive a reward. This state s' is the preprocessed image of the next game screen. We store this transition in our replay buffer as <s,a,r,s'>
4. Next, sample some random batches of transitions from the replay buffer and calculate the loss
5. It is known that: $Loss = (r + \gamma max_{a'} Q(s', a'; \theta') - Q(s, a; \theta))^2$ which is just the squared difference between target Q and predicted Q
6. Perform gradient descent with respect to our actual network parameters in order to minimize this loss
7. After every C iterations, copy our actual network weights to the target network weights
8. Repeat these steps for *M* number of episodes

Hyperparameters

Hyperparameter	Value
Discount Factor	0.99
Number of episodes	1000
Update Interval	4
Replay Buffer Size	1e5
Batch Size	64
Learning Rate	5e-4

RESULTS for DQN



Next Steps:

There are some more advanced Deep RL techniques, such as Double DQN Networks, Dueling DQN and Prioritized Experience replay which can further improve the learning process. These techniques give us better scores using an even lesser number of episodes.

A novel multi-step Q-learning method is proposed to improve data efficiency for DRL. The proposed multi-step Q-learning method is derived by adopting a new return function. The new return function alters the discount of future rewards and loosens the impact of the immediate reward.

Experimental-results shows the proposed methods can improve the data efficiency of DRL agents.

In addition **distributional DQN could be used to improve the results.**