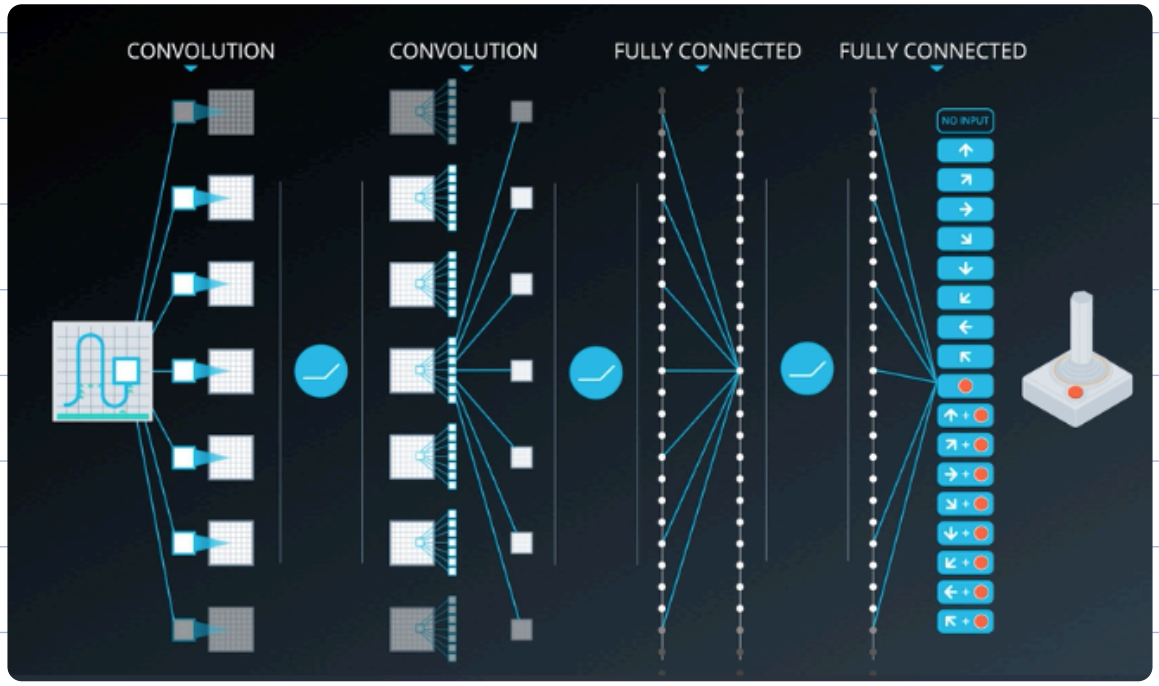
Deep Q-Networks



Deep Q Network Architecture

- Deepmind used DQN to play many Atari games.
- DQN are able to learn the action value given a state as input.
- It used raw image pixels as game state & predicted the action value for each possible game action.
- Their proposed approach used 4 consecutive game frames to capture the temporal information.
- The network comprised of 2 ReLU activated convolution layers followed by 2 Fully connected layers.
- RL is unstable when using neural networks such as DQN. To overcome this issue 2 common methods which were used are :-

Experience Replay :-

Reason R1 :- When the agent interacts with the environment, the sequence of experience tuples (S, A, R, S') can be highly correlated.

$(S_t, A_t, R_{t+1}, S_{t+1})$

$(S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2})$

$(S_{t+2}, A_{t+2}, R_{t+3}, S_{t+3})$

The neural network based approaches (DQN) can get swayed away by these correlations in sequential order experience tuples.

R2:- Some of the (states, actions) can be very rare in an environment & hence throwing them away after a single use (iteration) may not help the network learn.

R3:- Some states which are seen initially only can be forgotten by the network if they are not visited again due to progress in the environment.

To overcome these issues a "Replay Buffer" is used which contains a collection of experience tuples (S, A, R, S') which are added as the agent interacts with the env. Experience Replay is to then randomly sample from the replay buffer to train the network which prevents action values from oscillating or diverging catastrophically. It also allows the rare events to be sampled again due to random selection.

based on some policy (random / learned) ↑

Note:- It is good for the agent to first try different actions on same / different states which are then stored in the buffer & then to train the n/w which allows the agent to explore instead of learning sequentially on the go, which may make it biased towards certain actions.

Fixed Q-Targets:- The Q-learning update step in neural n/w is as follows:-

$$J(w) = E_\pi\left[\left(q_\pi(S,A) - \hat{q}(S,A,w)\right)^2\right]$$

$$\nabla_w J(w) = -2\left(q_\pi(S,A) - \hat{q}(S,A,w)\right)\nabla_w \hat{q}(S,A,w)$$

$$\Delta w = -\alpha \frac{1}{2}\nabla_w J(w)$$

$$= \alpha\left(q_\pi(S,A) - \hat{q}(S,A,w)\right)\nabla_w \hat{q}(S,A,w) \quad -(1)$$

To update w, same w used as target q-value estimator

$$\Delta w = \alpha \left(R + \gamma \max_a \hat{q}(S', a, w) - \hat{q}(S, A, w) \right) \nabla_w \hat{q}(S, A, w) \quad -(2)$$

The problem in moving from eq. (1) to eq. (2) is that $q_\pi(S, A)$ is the target action value that we hope to achieve. But, instead we are again estimating it with model parameters 'w' which is mathematically incorrect.

Here, we are trying to update a guess with a guess which can potentially lead to harmful correlations as we are trying to update 'w' with the same 'w' parameters in the Target Q-value estimation, leading to unstable training.

Instead, we can use the following update rule:-

decoupled

$$\Delta w = \alpha \left(R + \gamma \max_a \hat{q}(S', a, \bar{w}) - \hat{q}(S, A, w) \right) \nabla_w \hat{q}(S, A, w)$$

where $\bar{w}$ are the weights of a separate target network that are not changed during the learning step.

After a certain number of learning steps, updated 'w' weights are copied into '$\bar{w}$' and the process is repeated.

Algorithm: Deep Q-Learning

- Initialize replay memory D with capacity N
- Initialize action-value function $\hat{q}$ with random weights $\mathbf{w}$
- Initialize target action-value weights $\mathbf{w}^- \leftarrow \mathbf{w}$
- **for** the episode $e \leftarrow 1$ to M:
  - Initial input frame x_1
  - Prepare initial state: $S \leftarrow \phi(\langle x_1 \rangle)$
  - **for** time step $t \leftarrow 1$ to T:

$\phi(x_i)$ is a pre-processing function that takes the initial game frame to convert it into required size & stack 4 frames together

SAMPLE

Choose action A from state S using policy $\pi \leftarrow \epsilon\text{-Greedy}(\hat{q}(S, A, \mathbf{w}))$
Take action A, observe reward R, and next input frame x_{t+1}
Prepare next state: $S' \leftarrow \phi(\langle x_{t-2}, x_{t-1}, x_t, x_{t+1} \rangle)$
Store experience tuple (S, A, R, S') in replay memory D
$S \leftarrow S'$

LEARN

Obtain random minibatch of tuples (s_j, a_j, r_j, s_{j+1}) from D
Set target $y_j = r_j + \gamma \max_a \hat{q}(s_{j+1}, a, \mathbf{w}^-)$
Update: $\Delta \mathbf{w} = \alpha \left(y_j - \hat{q}(s_j, a_j, \mathbf{w}) \right) \nabla_{\mathbf{w}} \hat{q}(s_j, a_j, \mathbf{w})$
Every C steps, reset: $\mathbf{w}^- \leftarrow \mathbf{w}$

* You are encouraged to read the original DQN paper by Deepmind which gives many other optimization techniques.

Updating Fixed Target Q Network:-

While copying the weights of w to $\bar{w}$, the weights are not completely overwritten by w, but instead weights of both w & $\bar{w}$ are combined together using a paramater $tau(\tau)$ which governs the division as follows:-

$$\bar{w}_{new} \leftarrow \tau * w + (1-\tau) * \bar{w}_{old} \quad \text{where } 0 \leq \tau \leq 1$$

Improvements over DQN

Double DQN:-

DQNs have a problem of "overestimation of Q-values".

Recall, that in DQN:-

$$\Delta w = \alpha \left(R + \gamma \max_a \hat{q}(S', a, \bar{w}) - \hat{q}(S, A, w) \right) \nabla_w \hat{q}(S, A, w)$$

rewrite as

$$R + \gamma \hat{q}\left(S', \underset{a}{\text{argmax}}\, \hat{q}(S', a, \bar{w}), \bar{w}\right)$$

- $\hat{q}$: Q-value
- S' : state
- $\underset{a}{\text{argmax}}\, \hat{q}(S', a, \bar{w})$: action with max Q-value

When written in this manner, it is clear that the DQN can do mistakes especially in the initial training steps when it has very less knowledge & hasn't tried out all possible actions to know which is a good action.

This leads to an overestimation of Q-values since we are taking the maximum among a set of noisy numbers.

To reduce this overestimation, Double Q-Learning can be used, where you estimate the best action using one set of parameters & get the q-value for that action using other set of parameters to reduce the overestimation.

In Double DQN we already have 2 parameter sets (w & w^-) for local & target Q-networks which are used as follows to reduce overestimation :-

$$R + \gamma\, \hat{q}\left(S', \underbrace{\underset{a}{\operatorname{argmax}}\, \hat{q}(S', a, w)}_{\text{select best action}}, \underbrace{w^-}_{\text{evaluate that action}}\right)$$

This way both the models should agree on that action otherwise the q-value returned won't be high which solves the problem of overestimation.

Prioritized Experience Replay :-

Another issue in DQN is that the Replay Buffer samples experiences with a uniform probability. This can be a problem since some experience might be more useful than the rest & some may be very rare.
This motivates us to have a Prioritized Experience Replay Buffer.

To assign a probability to an experience, we can use the amount of TD Error the experience generates as a measure of importance of that experience :-

$$\delta_t = \underbrace{R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w^-)}_{\text{TD Target}} - \underbrace{\hat{q}(S_t, A, w)}_{\text{TD Expected}}$$

$\leftarrow$ TD Error

then, Priority (p_t) of a sample can be :-

$$p_t = |\delta_t|$$

& can be stored in the replay buffer, with the experience as :- $< S_t, A_t, R_{t+1}, S_{t+1}, p_t >$

then, the Sampling Probability of each sample can be expressed as :-

$$P(i) = \frac{p_i}{\sum_k p_k}$$

When an experience is picked, its priority is updated using the newly computed TD Error.

Some improvements :-

1. If the TD error of a sample is zero, the sample may never be picked even though it may be useful in future. To avoid leaving out on an experience, we can add a small constant (e) to every priority value :-

   Now, Priority $p_t = |\delta_t| + e$

2. Greedily Sampling using the probabilities may lead to overfitting over some samples which have high probability. To avoid this we can reintroduce some sort of uniform sampling as follows :-

   Sampling Probability $P(i) = \frac{p_i^a}{\sum_k p_k^a}$, where $0 \leq a \leq 1$

   $a = 0$, leads to complete uniform sampling

   $a = 1$, leads to complete priority sampling

3. We only want to sample experiences from the buffer using priorities but such a sampling also introduces a bias in the learned Q-values.

To avoid this, we can modify the update rule :-

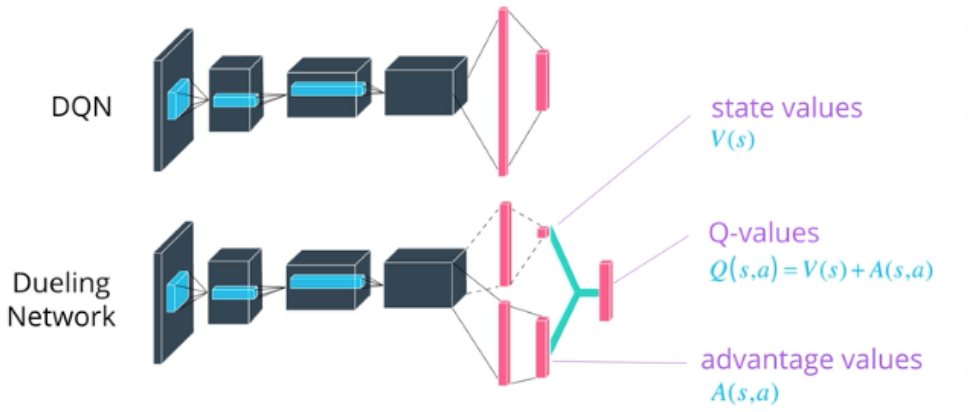
Modified Update Rule :- $\Delta w = \alpha \underbrace{\left(\frac{1}{N} \cdot \frac{1}{P(i)}\right)^b}_{\text{Importance - sampling weight}} \delta_i \nabla_w \hat{q}(S_i, A_i, w)$

here, N = Number of samples in the replay buffer

b = constant such that $0 \leq b \leq 1$

value of 'b' can be gradually increased to 1 as the q-values converge overtime with training.

Dueling Network :-



The motivation behind Dueling N/W architecture is that mostly State values remain the same where as the action values may vary.

To fulfill this, the FC layers are divided into 2 parts, State Values & Advantage Values.

State values gives out a scalar value (V(s)) corresponding to the state

Advantage values gives a vector of values which corresponds to the values of each action.

The outputs are then combined as follows to get the Q-value :-

$$Q(s,a) = V(s) + A(s,a)$$

which is calculated as follows for stable training :-

$$Q(s,a) = V(s) + \left(A(s,a) - \frac{1}{|A|} \sum_{a'} A(s,a') \right)$$

$|A|$ → No. of Actions

Rainbow DQN :-

Deepmind combined six different extensions of DQN together & named the algorithm as Rainbow DQN. The 6 optimizations were :-

- Double DQN
- Prioritized Experience Replay
- Dueling DQN
- Multi-step bootstrap targets (revisited in A3C)
- Distributional DQN
- Noisy DQN

These techniques combined together gave State-of-the-art results.