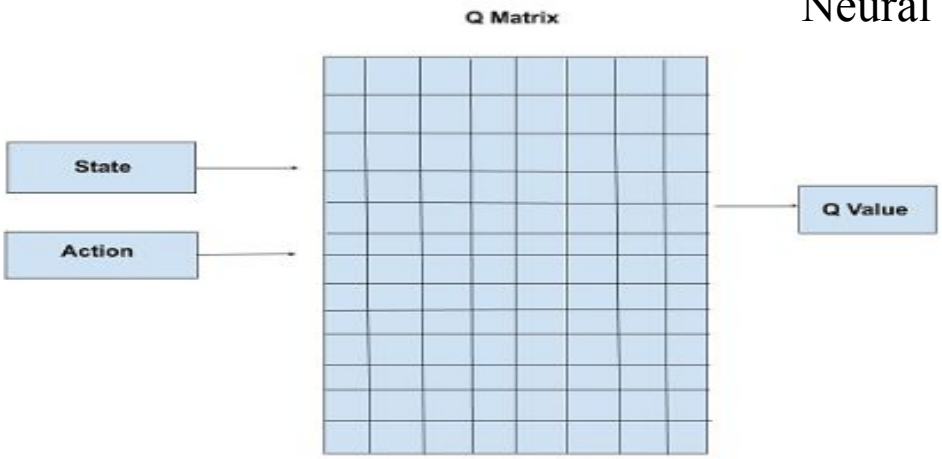
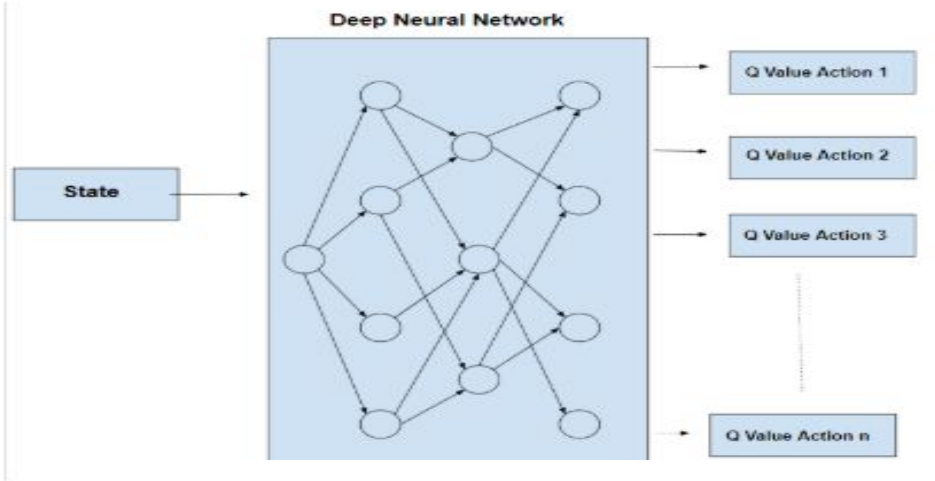
Deep Q Learning

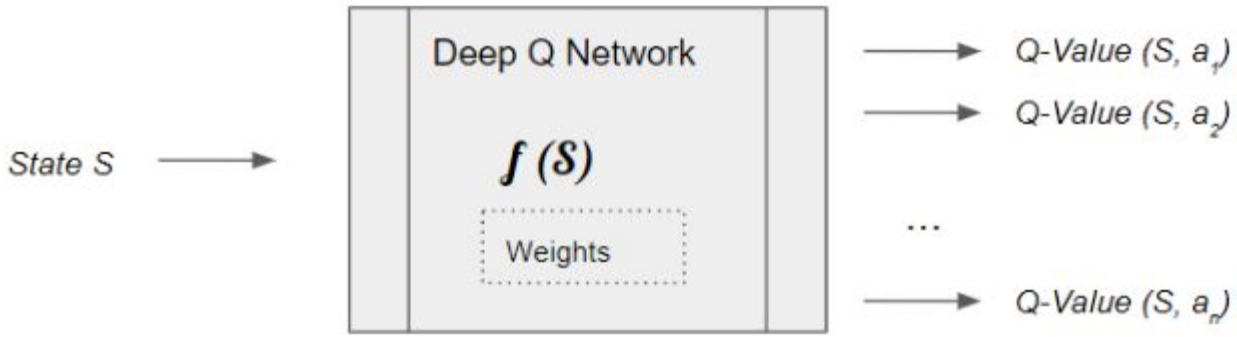
Introduction

- Q-learning is only practical for very small environments and quickly loses it's feasibility when the number of states and actions in the environment increases.
- Deep Q-Learning is a variant of Q-Learning that uses a deep neural network to represent the Q-function, rather than a simple table of values.
- Deep neural network approximates the Q-function, which is used to determine the optimal action for a given state.
- The Q-function represents the expected cumulative reward of taking a certain action in a certain state and following a certain policy.
- The algorithm to handle environments with a large number of states and actions, as well as to learn from high-dimensional inputs such as images or sensor data.

Neural Nets are the best Function Approximators

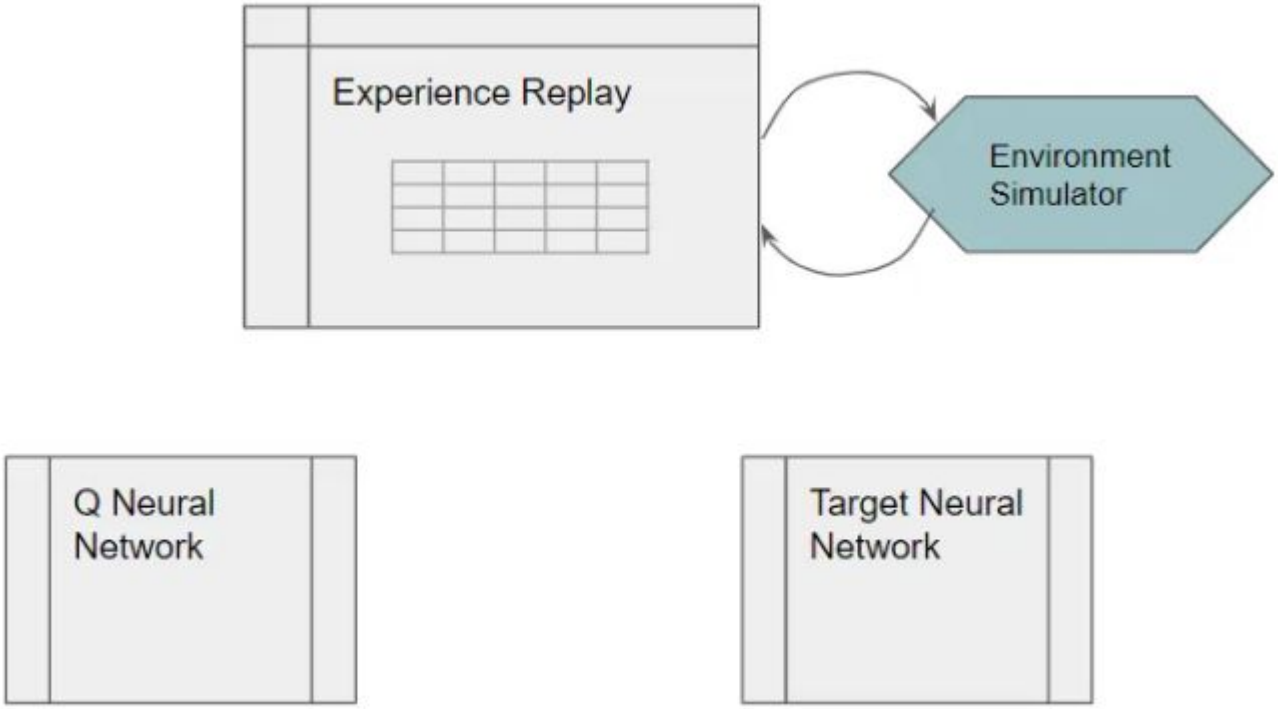






DQN Architecture Components

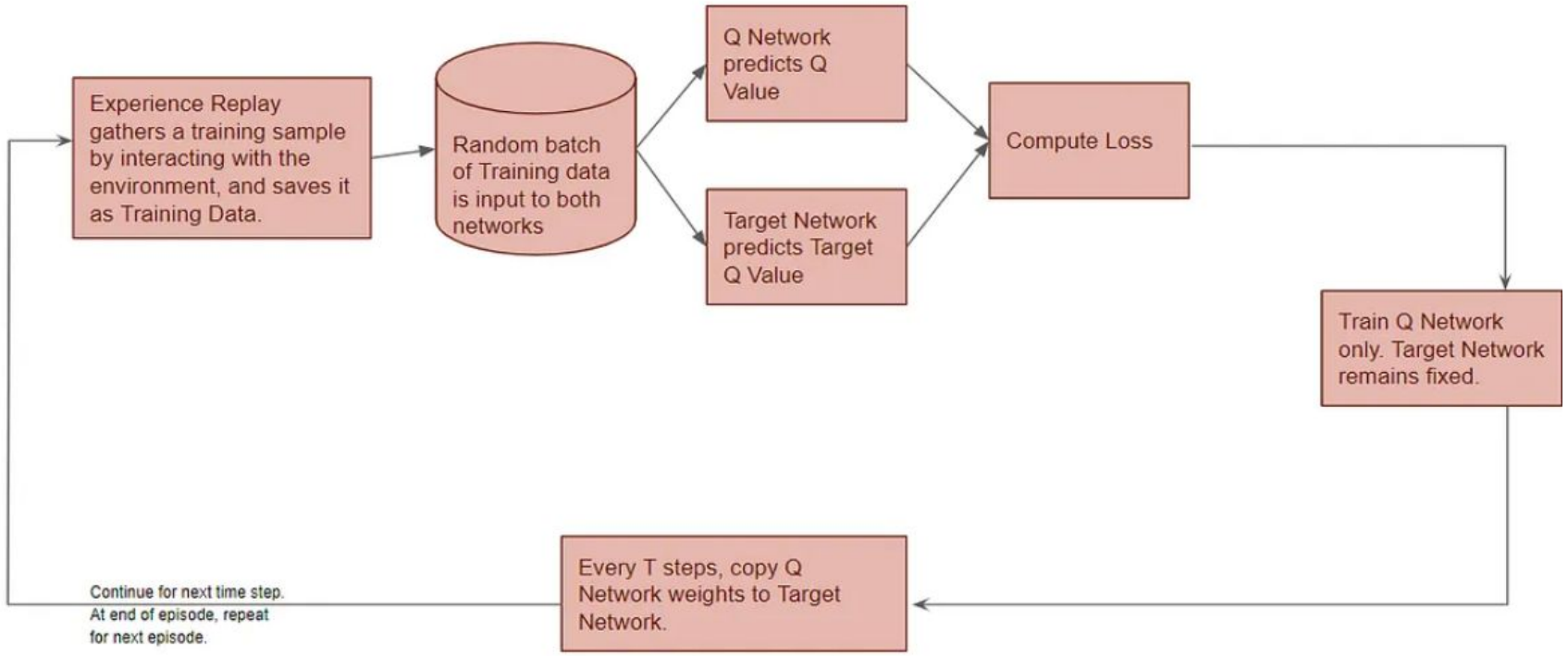
- One of the key challenges in implementing Deep Q-Learning is that the Q-function is typically non-linear and can have many local minima, therefore difficult to converge to the correct Q-function.
- To address this, the DQN architecture has two neural nets, the **Q network** and the **Target networks**, and a component called **Experience Replay**.
- Q network is the agent that is trained to produce the Optimal State-Action value by adjusting the parameters.
- **Target network** is used for computing the target, has the same architecture as the first network but has frozen parameters.
- After an x number of iterations, the parameters are copied to the target network.
- **Experience replay** is a technique where the agent stores a subset of its experiences (state, action, reward, next state) in a memory buffer and samples from this buffer to update the Q-function.



- The Q Network could be as simple as a linear network with a couple of hidden layers if your state can be represented via a set of numeric variables.
- Or if your state data is represented as images or text, you might use a regular CNN or RNN architecture.

- In Deep Q-Learning, we create a loss function that compares Q-value prediction and the Q-target and uses gradient descent to update the weights of our Deep Q-Network to approximate our Q-values better.

High-level DQN Workflow

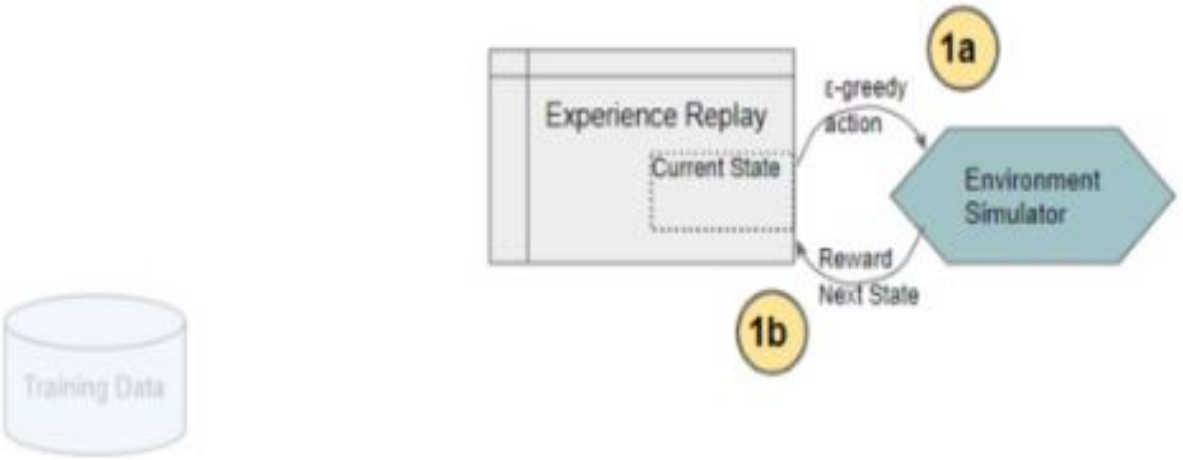


These operations are performed in each time-step (Image by Author)

- The DQN gets trained over multiple time steps over many episodes. It goes through a sequence of operations in each time step.

Training Data

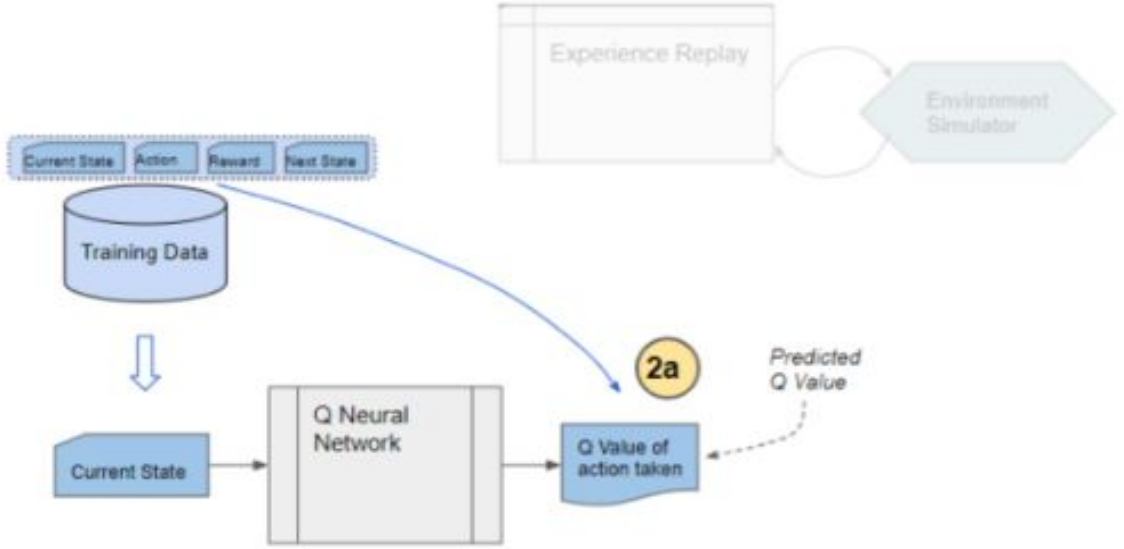
- Experience Replay selects an ε-greedy action from the current state, executes it in the environment, and gets back a reward and the next state.



Experience Replay saves this observation as a sample of training data.

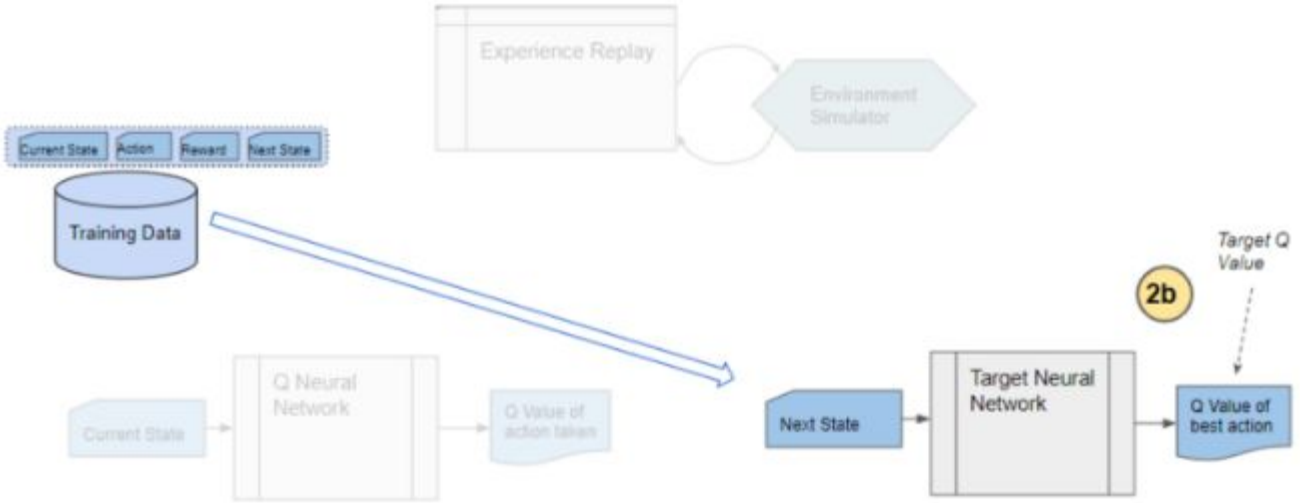
Q Network predicts Q-value

- We now take a random batch of samples from this training data, so that it contains a mix of older and more recent samples.
- This batch of training data is then inputted to both the networks.
- The Q network takes **the current state** and action from each data sample and predicts the Q value for that particular action as ‘Predicted Q Value’.



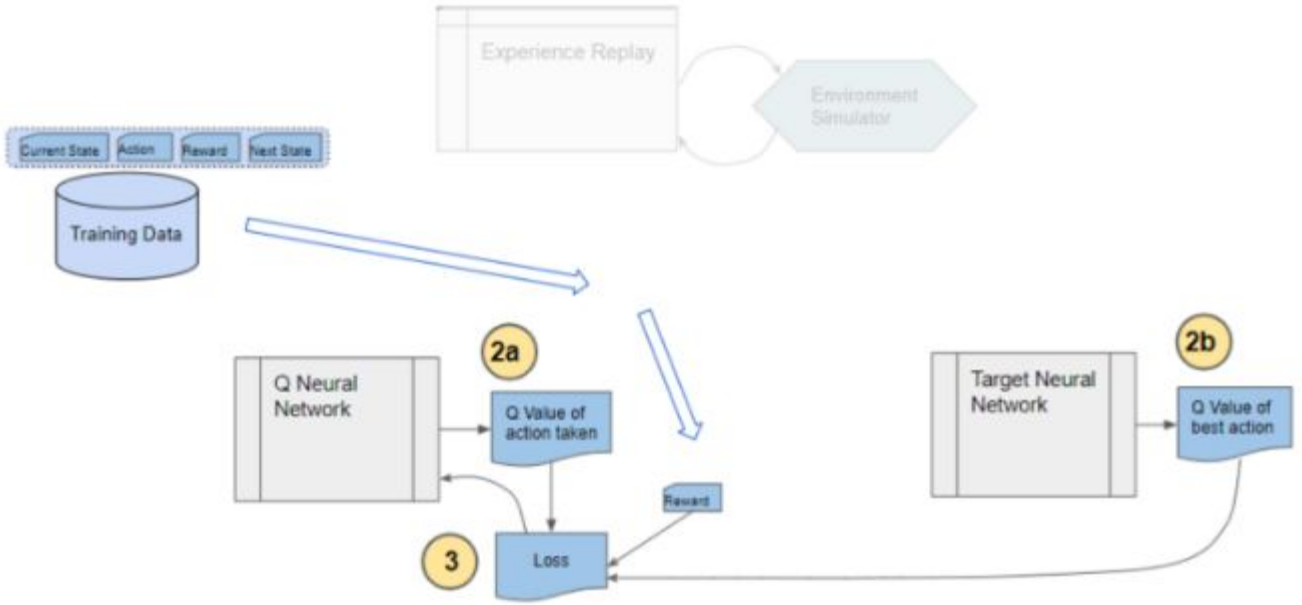
Target Network predicts Target Q-value

- The Target network takes **the next state** from each data sample and predicts the best Q value out of all actions that can be taken from that state. This is the 'Target Q Value'.



Compute Loss and Train Q Network

- The Predicted Q Value, Target Q Value, and the observed reward from the data sample is used to compute the Loss to train the Q Network.
- The Target Network is not trained.



Why do we need Experience Replay?

- If we simply take an action, observe results from the environment, and then feed that data to the Q Network then during training each sample and the corresponding gradients would have too much variance, and the network weights would never converge.
- To train neural networks, a best practice is to select a batch of samples after shuffling the training data randomly.
- This ensures that there is enough diversity in the training data to allow the network to learn meaningful weights that generalize well and can handle a range of data values.
- Would that occur if we passed a batch of data from sequential actions?
- Actions in sequence one after the other and then feed that data as a batch to the Q Network, enough diversity in the training data.
- However, Sequential actions are highly correlated with one another and are not randomly shuffled, as the network would prefer.
- This results in a problem called **Catastrophic Forgetting** where the network unlearns things that it had learned a short while earlier.

- This is why the Experience Replay memory was introduced.
- All of the actions and observations that the agent has taken from the beginning (limited by the capacity of the memory, of course) are stored.
- Then a batch of samples is randomly selected from this memory.
- This ensures that the batch is ‘shuffled’ and contains enough diversity from older and newer samples to allow the network to learn weights that generalize to all the scenarios that it will be required to handle.

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s,a;\theta) \right)$$

Bellman Equation i

Robot learning

- Let's say that at a certain point in time, Robot is trying to find its way around a particular corner of the factory.
- All of the actions that it would take over the next few moves would be confined to that section of the factory.
- If the network tried to learn from that batch of actions, it would update its weights to deal specifically with that location in the factory.
- But it would not learn anything about other parts of the factory.
- If sometime later, the robot moves to another location, all of its actions and hence the network's learning for a while would be narrowly focused on that new location.
- It might then undo what it had learned from the original location.

Why do we need a second neural network (Target Network)?

- It is possible to build a DQN with a single Q Network and no Target Network, as the target network is not getting trained.
- In that case, we do two passes through the Q Network, first to output the Predicted Q value, and then to output the Target Q value.
- But that could create a potential problem.
- The Q Network's weights get updated at each time step, which improves the prediction of the Predicted Q value.
- However, since the network and its weights are the same, it also changes the direction of our predicted Target Q values.
- They do not remain steady but can fluctuate after each update. This is like chasing a moving target .

Target Network

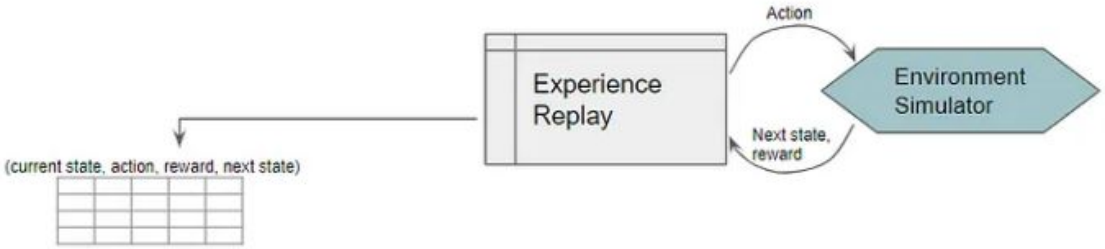
- DQN introduces a second neural network, called the target network, which is used to calculate the target Q-values.
- This target network is updated less frequently than the main network to prevent rapid oscillations in learning.
- By employing a second network that doesn't get trained, we ensure that the Target Q values remain stable, at least for a short period.
- But those Target Q values are also predictions after all and we do want them to improve, so a compromise is made.
- After a pre-configured number of time-steps, the learned weights from the Q Network are copied over to the Target Network.
- It has been found that using a Target Network results in more stable training.

- It's difficult to come up with a perfect heuristic.
- Improving the heuristic generally entails playing the game many times, to determine specific cases where the agent could have made better choices.
- And, it can prove challenging to interpret what exactly is going wrong, and ultimately to fix old mistakes without accidentally introducing new ones.
- It would be much easier if we had a more systematic way of improving the agent with game play experience.

DQN Operation in depth

Initialization

Execute a few actions with the environment to bootstrap the replay data.



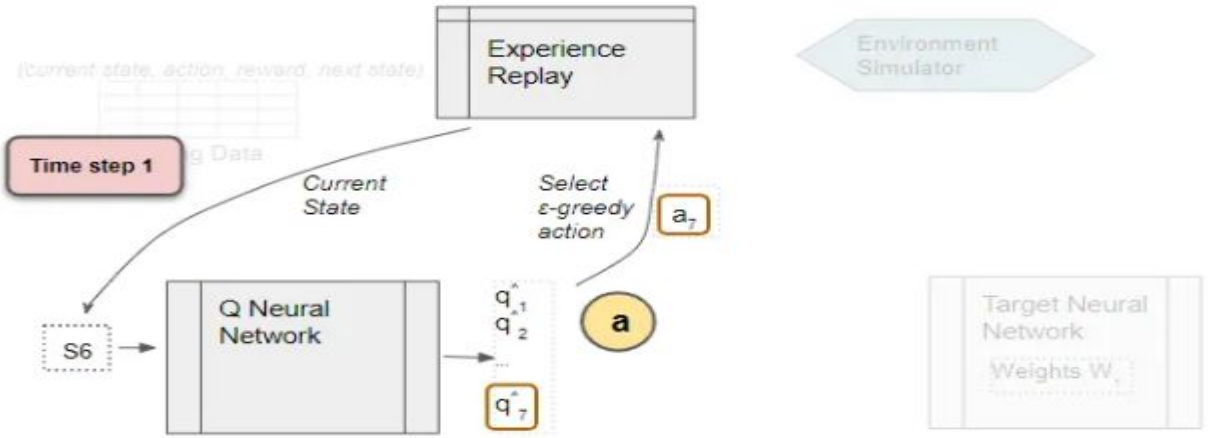
(Image by Author)

Initialize the Q Network with random weights and copy them to the Target Network.

(Image by Author)

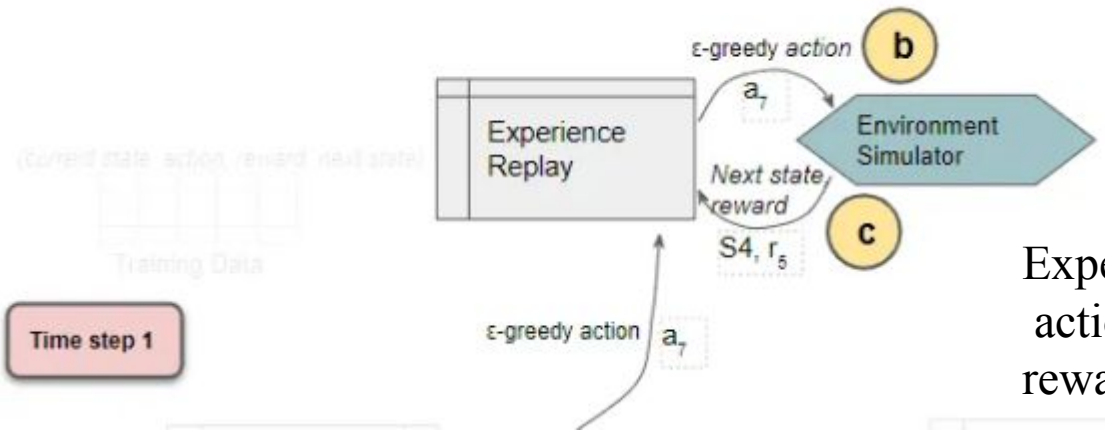
Experience Replay

- The Experience Replay starts the training data generation phase and uses the Q Network to select an ε-greedy action.
- The Q Network acts as the agent while interacting with the environment to generate a training sample.
- No DQN training happens during this phase.
- The Q Network predicts the Q-values of all actions that can be taken from the current state.
- We use those Q-values to select an ε-greedy action.

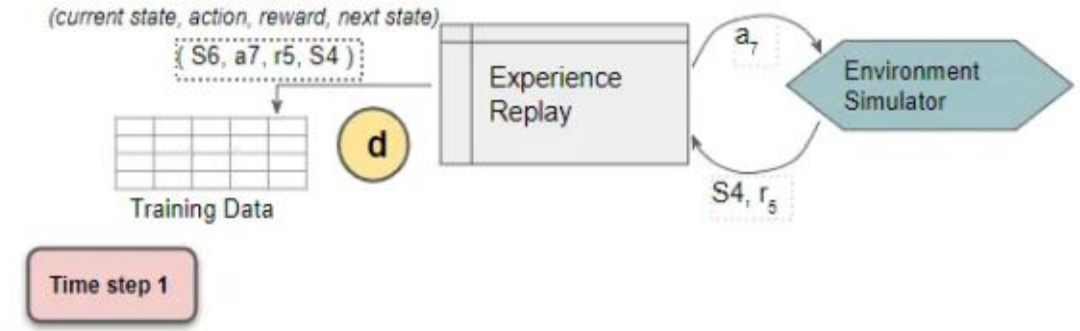


(Image by Author)

The sample data (Current state, action, reward, next state) is saved



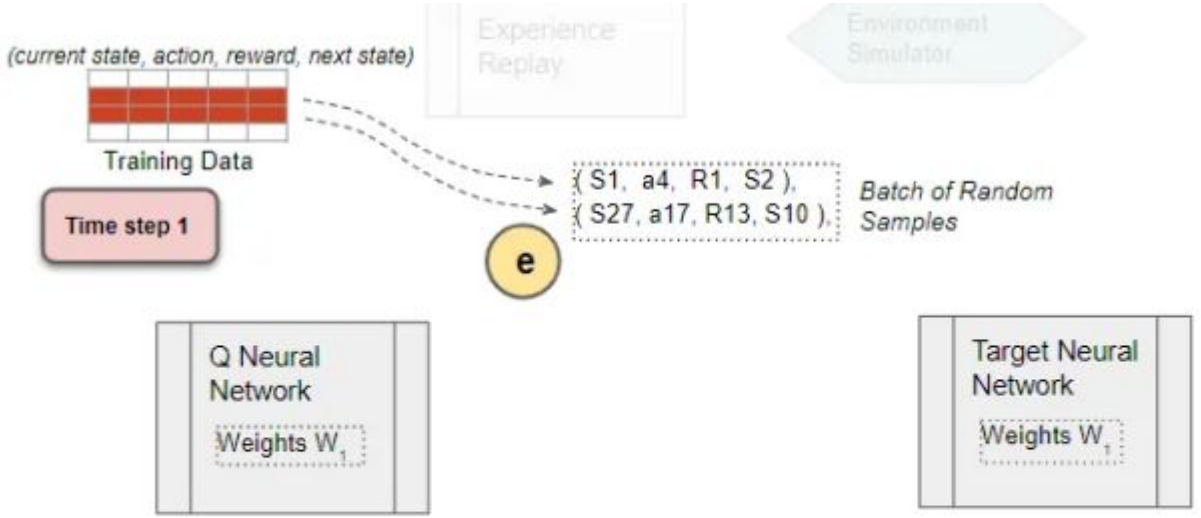
Experience Replay executes the ε-greedy action and receives the next state and reward.



It stores the results in the replay data. Each such result is a sample observation which will later be used as training data.

Select Random Training Batch

We now start the phase to train the DQN. Select a training batch of random samples from the replay data as input for both networks.

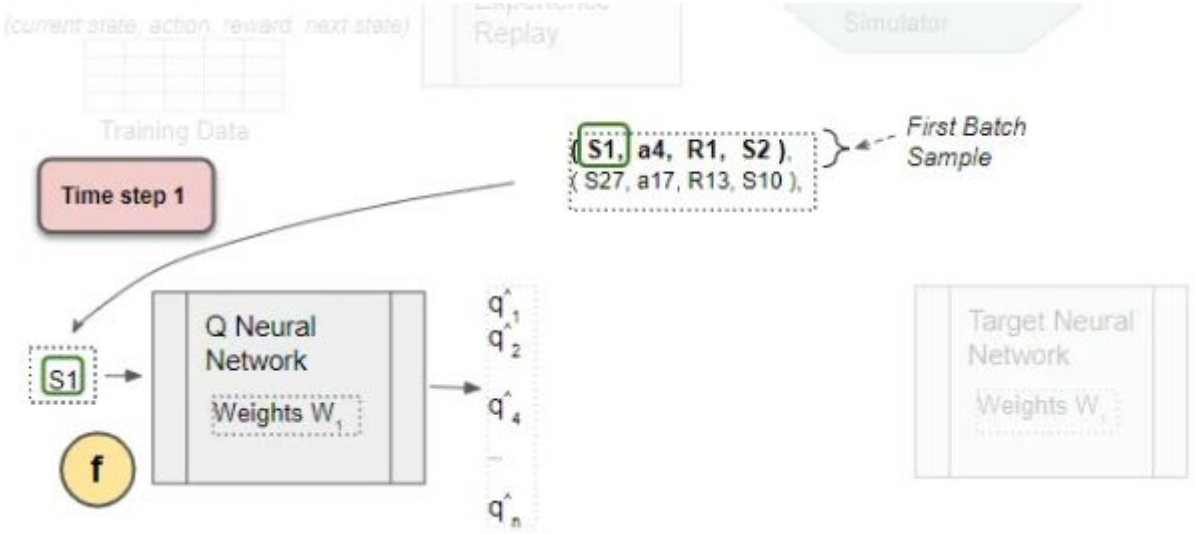


(Image by Author)

Use the current state from the sample as input to predict the Q values for all actions

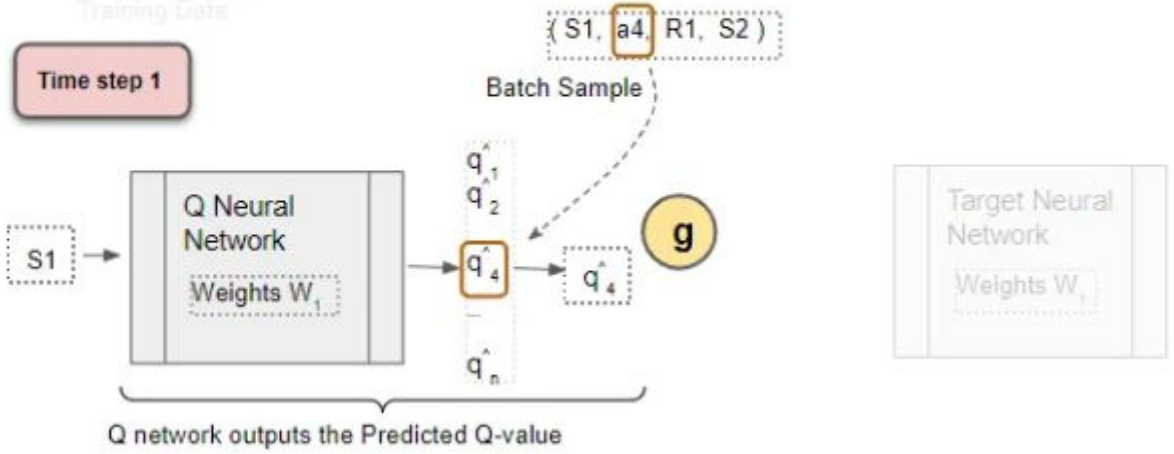
To simplify the explanation, let's follow a single sample from the batch.

The Q network predicts Q values for all actions that can be taken from the state.



Select the Predicted Q-value

From the output Q values, select the one for the sample action. This is the Predicted Q Value.



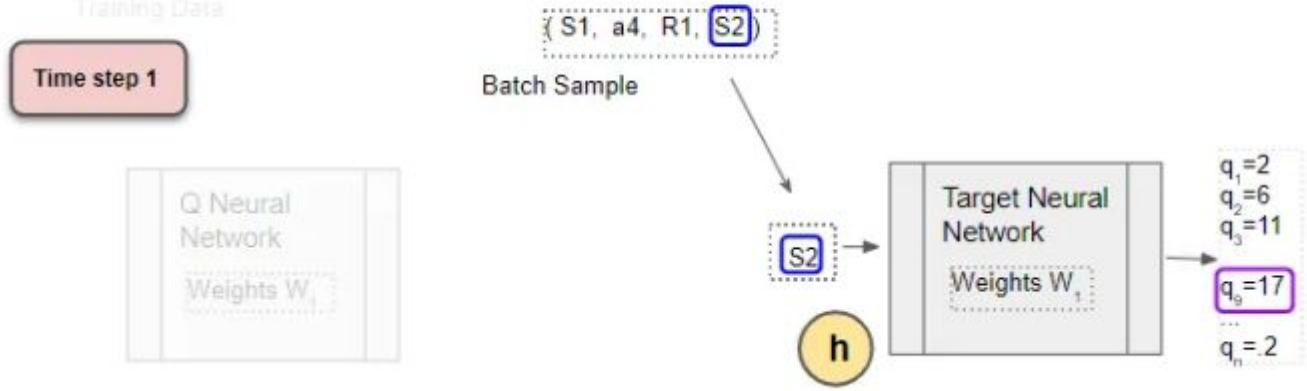
Use the next state from the sample as input to the Target network

The next state from the sample is input to the Target network.

The Target network predicts Q values for all actions that can be taken from the next state, and selects the maximum of those Q values.

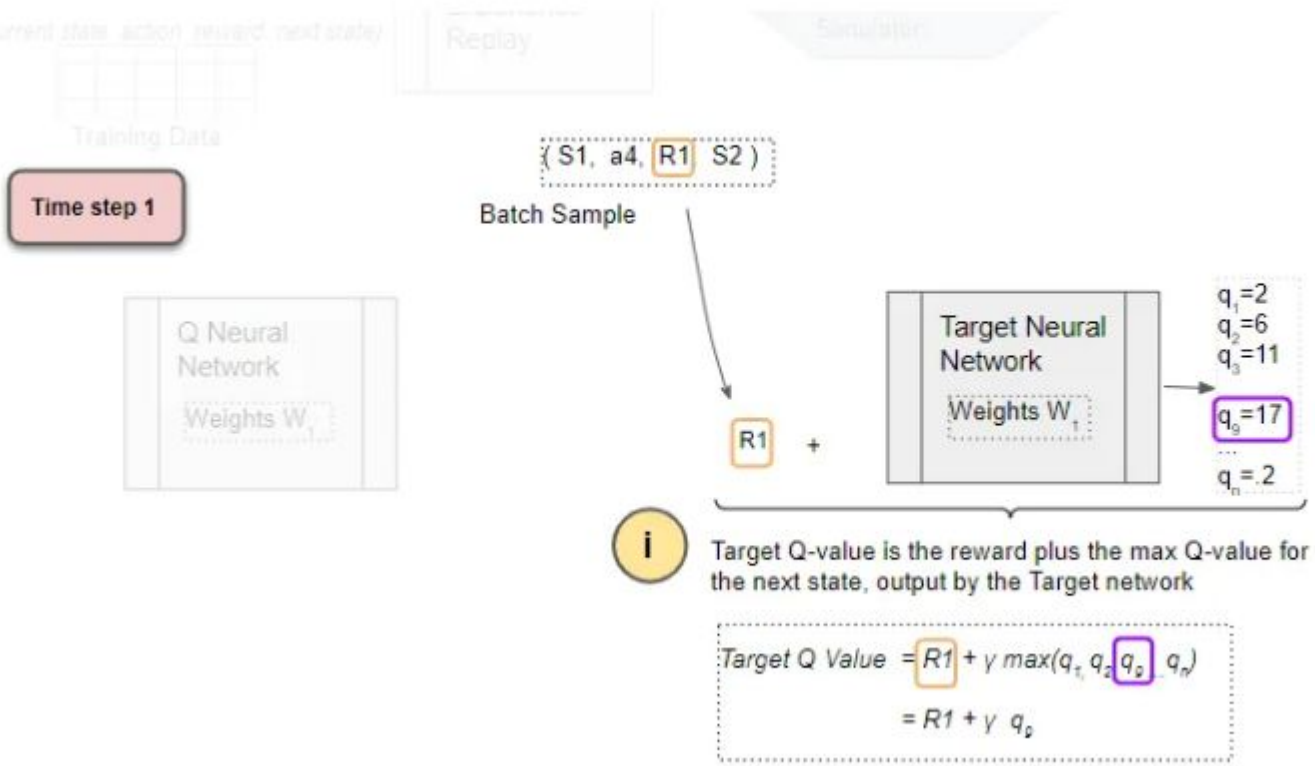
Use the next state as input to predict the Q values for all actions.

The target network selects the max of all those Q-values.



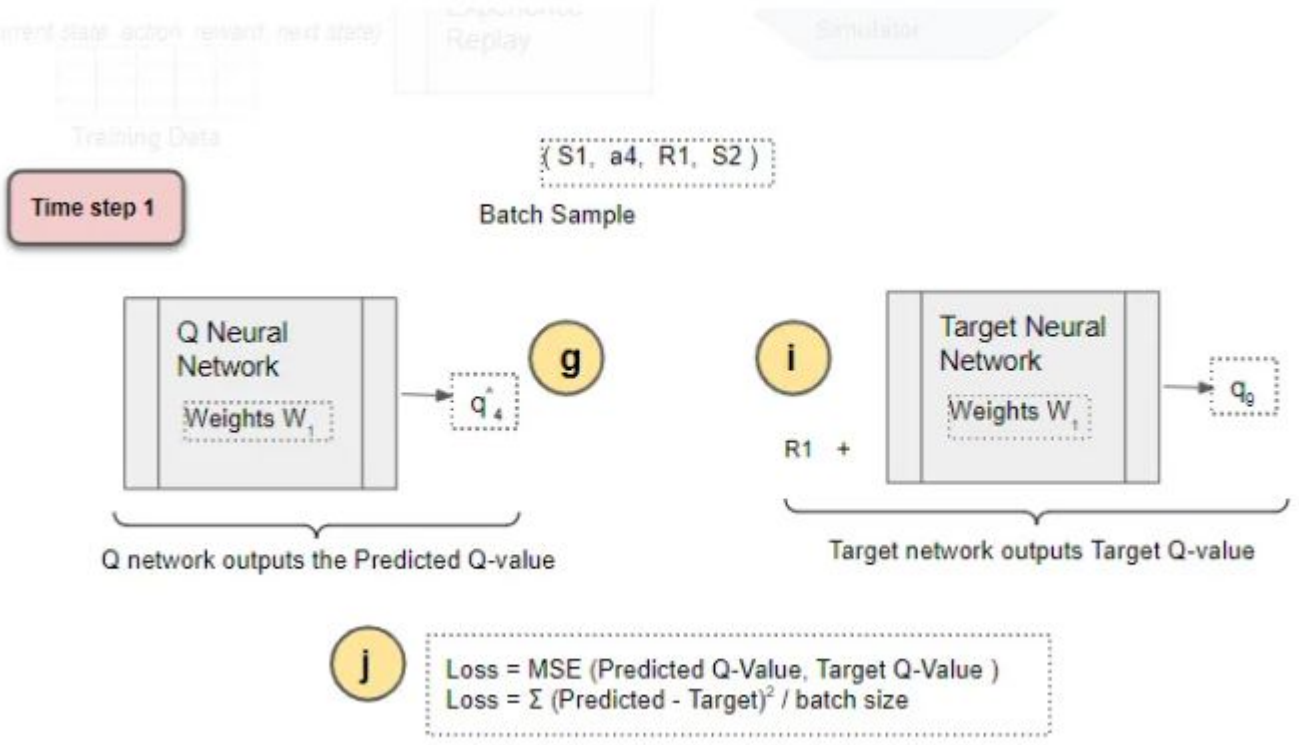
Get the Target Q Value

The Target Q Value is the output of the Target Network plus the reward from the sample.

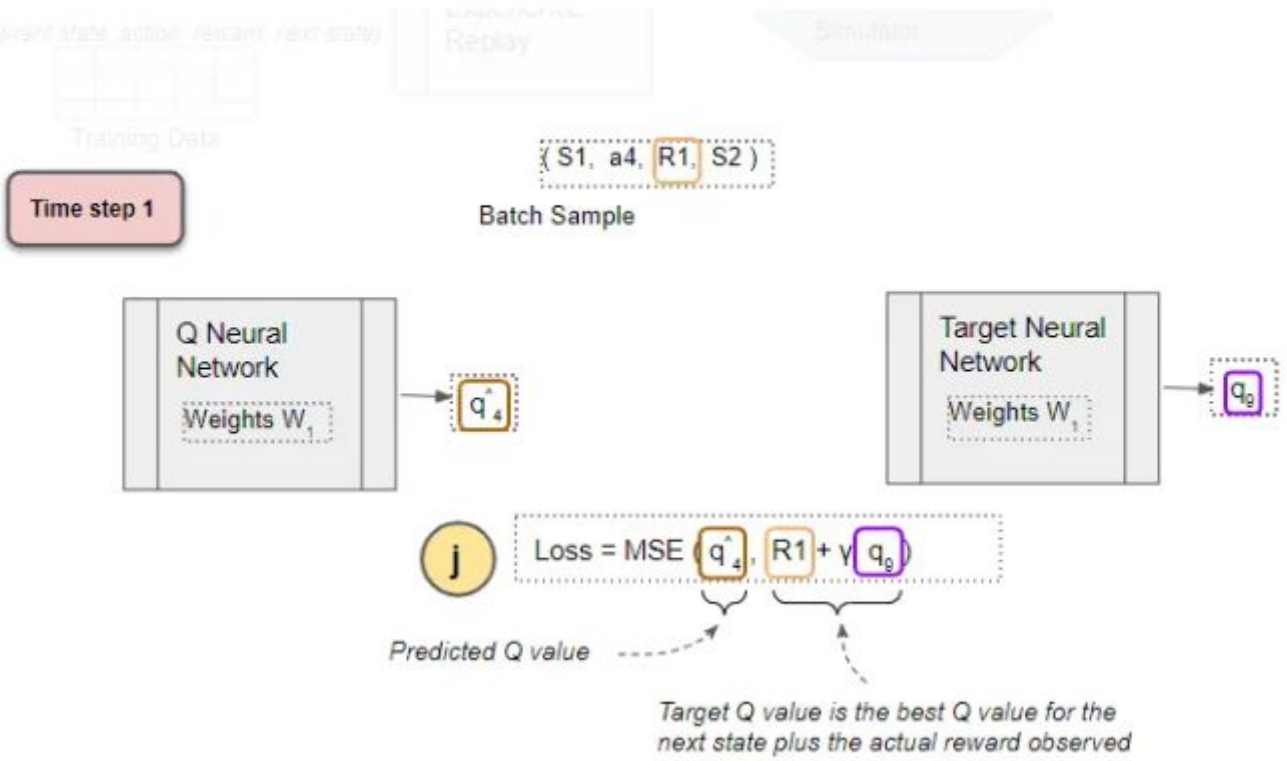


Compute Loss

Compute the Mean Squared Error loss using the difference between the Target Q Value and the Predicted Q Value.

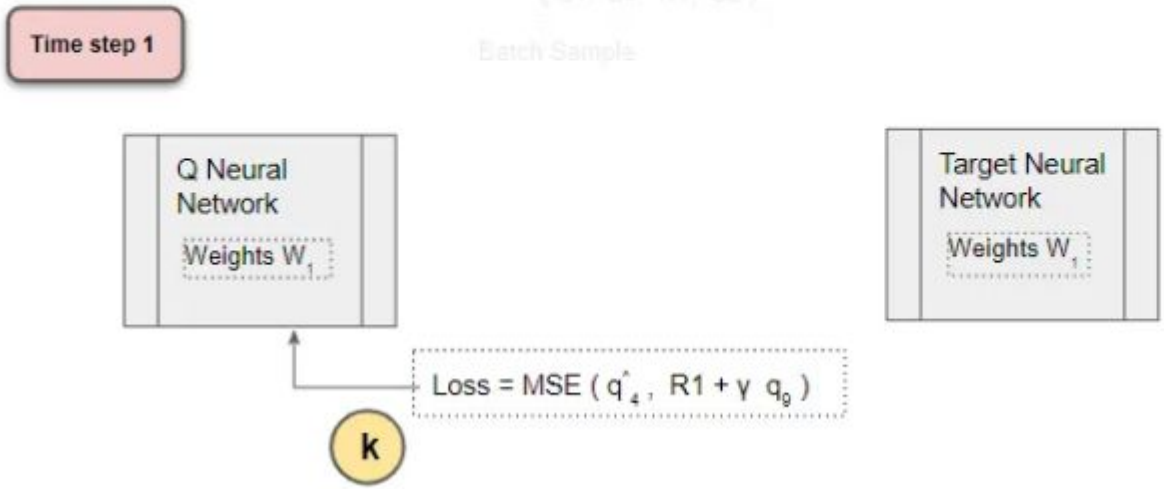


Compute Loss



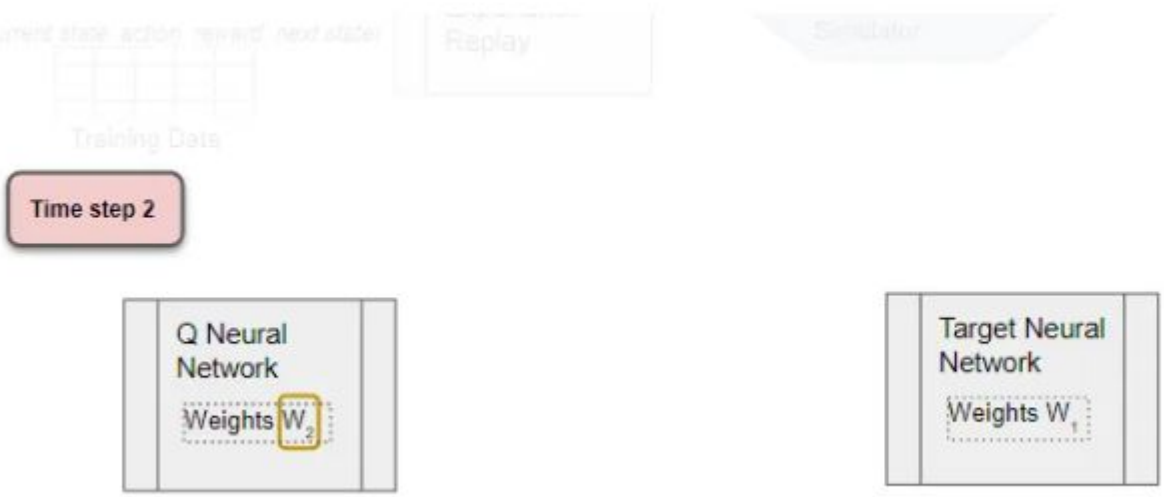
Back-propagate Loss to Q-Network

Back-propagate the loss and update the weights of the Q Network using gradient descent.
The Target network is not trained and remains fixed, so no Loss is computed, and back-propagation is not done.
This completes the processing for this time-step.



Repeat for next time-step

The processing repeats for the next time-step.
The Q network weights have been updated but not the Target network's.
This allows the Q network to learn to predict more accurate Q values, while the target Q values remain fixed for a while, so we are not chasing a moving target.



After T time-steps, copy Q Network weights to Target Network

After T time-steps, copy the Q network weights to the Target network.

This lets the Target network get the improved weights so that it can also predict more accurate Q values. Processing continues as before.

The Q network weights and the Target network are again equal.