Università degli Studi di Milano, Data Science for Economics Master programme

Deep Q Learning

Prof. Alfio Ferrara

Reinforcement Learning

Introduction

We are going to explore other (eventually non-linear) methods for Value Function Approximation (VFA) by Deep Learning. Our approach is based on the same definition of state value function and state-action value function that we have seen so far, that is:

Approximate value function

$$\hat{V}(s, \mathbf{w}) \rightarrow \mathbf{x(s)}^T \mathbf{w} \tag{1}$$

Approximate state-action value function

$$\hat{Q}(s, a, \mathbf{w}) \rightarrow \mathbf{x(s, a)}^T \mathbf{w} \tag{2}$$

Still, we are going to focus on differentiable approximation functions and we are going to use stochastic gradient descent to update the values of the two functions by updating the parameters $\mathbf{w}$. The goal of this formulation is to *generalize* the notion of state in order to deal with problems with a large state space, where the agent's behavior can be based on previous experience even if a specific state has never been encountered but only sufficiently similar states.

In this setting, state value and state-action value functions are no more represented by tables and the new information needed to update the parameters is given in form of a tuple (s, a, r, s') (we will use also the notation (s_t, a_t, r_t, s_{t+1})).

The objective function for learning is:

$$L(\mathbf{w}) = \mathbb{E}_\pi \left[\left(V^\pi(s) - \hat{V}(s, \mathbf{w}) \right)^2 \right] \tag{3}$$

and the update value of $\mathbf{w}$ by stochastic gradient descent is:

$$\Delta_{\mathbf{w}} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} L(\mathbf{w}) \tag{4}$$

Let's remember that this is not a supervised setting, because the ground-truth real value $V^\pi(s)$ is not known but must be estimated somehow.

We have two main methods for estimating $V^\pi(s)$:

- MC approach for policy evaluation: we use the sum of rewards G_t for a whole episode, so that $\Delta \mathbf{w} = \alpha \left(G_t - \mathbf{x}(s)^T \mathbf{w} \right) \mathbf{x}(\mathbf{s})$
- TD approach for policy evaluation: we use the (discounted) temporal difference of our estimations wuch that $\Delta_{\mathbf{w}} = \alpha \left(r_t + \gamma \mathbf{x}(s_{t+1})^T \mathbf{w} - \mathbf{x}(s_t)^T \mathbf{w} \right) \mathbf{x}(s_t)$

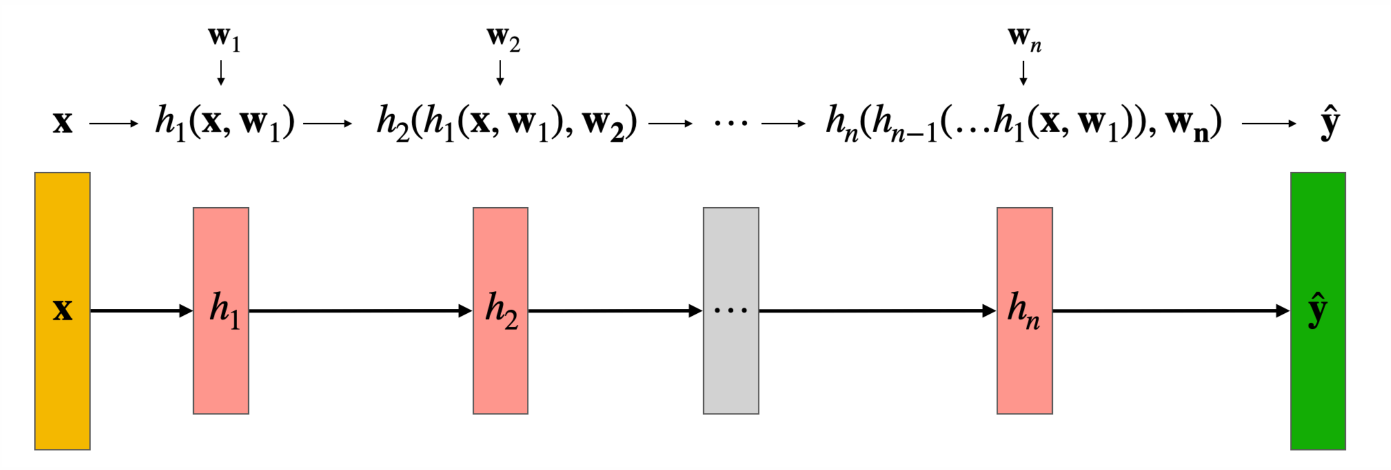
The same applies for $Q^{\pi}(s, a)$.

Linear approximation may be accurate but depends very much on the selection of features. To overcome this limitation, we are going to use other methods for approximating $V(s)$ and $Q(s, a)$, based on *deep neural networks*.

Summary on Deep Neural Networks

The main idea of DNNs is to produce and estimation by a composition of multiple functions.

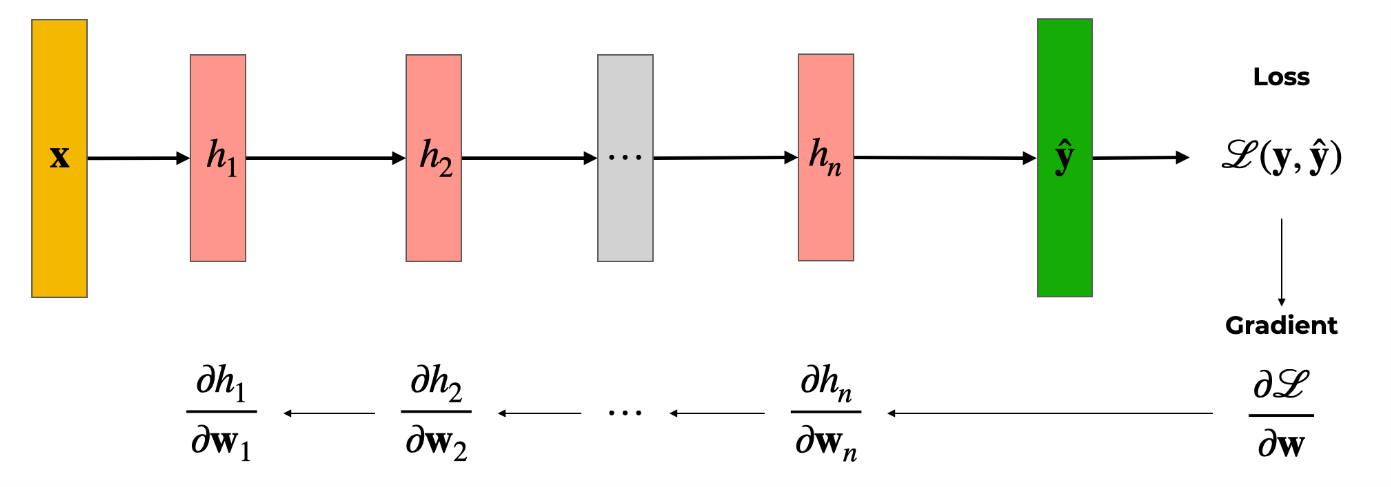
Forward propagation



$\mathbf{w}_1$
$\mathbf{w}_2$
$\mathbf{w}_n$
$\mathbf{x} \longrightarrow h_1(\mathbf{x}, \mathbf{w}_1) \longrightarrow h_2(h_1(\mathbf{x}, \mathbf{w}_1), \mathbf{w}_2) \longrightarrow \cdots \longrightarrow h_n(h_{n-1}(\ldots h_1(\mathbf{x}, \mathbf{w}_1)), \mathbf{w}_n) \longrightarrow \hat{\mathbf{y}}$
$\mathbf{x}$
h_1
h_2
$\cdots$
h_n
$\hat{\mathbf{y}}$

By decomposing a complex and wide spece of functions in a set of subsequent transformation, we ban propagate the gradient of the error back along the composition chain.

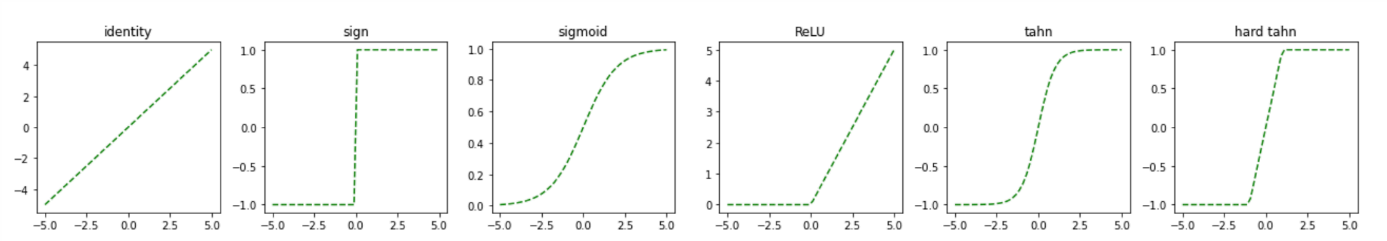
Backward propagation



$\mathbf{x}$
h_1
h_2
$\cdots$
h_n
$\hat{\mathbf{y}}$
Loss
$\mathscr{L}(\mathbf{y}, \hat{\mathbf{y}})$
Gradient
$\frac{\partial h_1}{\partial \mathbf{w}_1} \longleftarrow \frac{\partial h_2}{\partial \mathbf{w}_2} \longleftarrow \cdots \longleftarrow \frac{\partial h_n}{\partial \mathbf{w}_n} \longleftarrow \frac{\partial \mathscr{L}}{\partial \mathbf{w}}$

The only constraint we have on the choice of h_i is that this must be a **differentiable** function in order to compute gradient descent. Actually, h_i can be either linear or not-linear. Tipicall, h_i is designed to be a function $f(h_{i-1})$ of the previous h_{i-1} layer. In this setting f if called **activation function** and h are said to be the network **layers**.

Popular activation functions

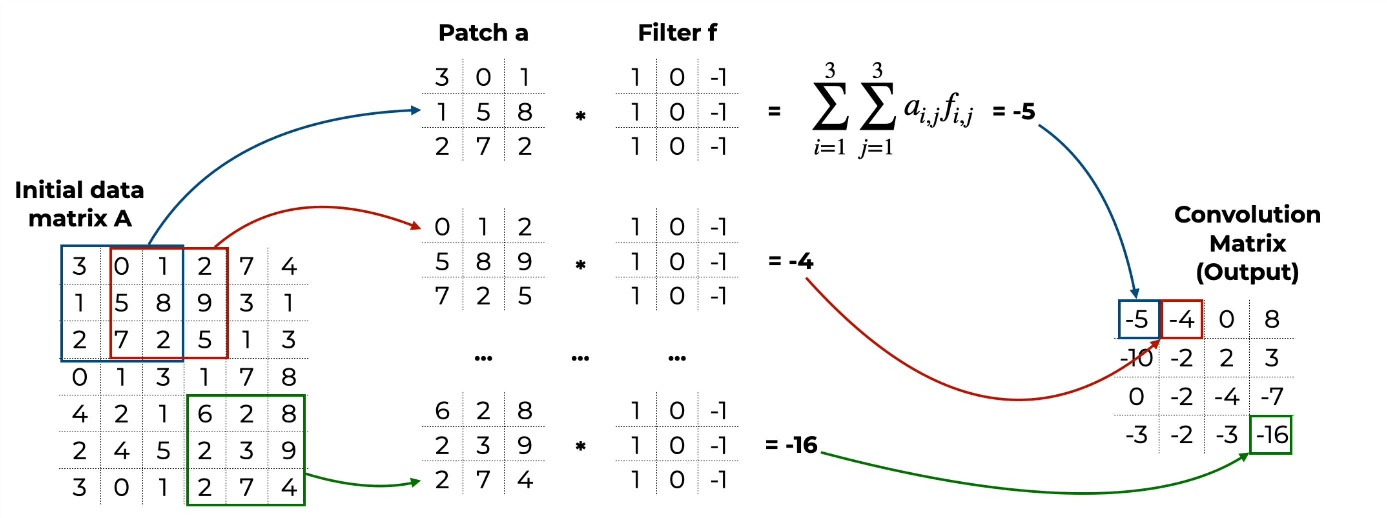


$$x \qquad \begin{cases} 0 & if\ x < 0 \\ 1 & if\ x \geq 0 \end{cases} \qquad \frac{1}{1+e^{-x}} \qquad \begin{cases} 0 & if\ x \leq 0 \\ x & if\ x > 0 \end{cases} \qquad \frac{e^x - e^{-x}}{e^x + e^{-x}} \qquad \begin{cases} -1 & if\ x < -1 \\ 1 & if\ x > 1 \\ x & otherwise \end{cases}$$

Convolutional Neural Networks (CNNs)

In order to use DNN to efficiently handle large space MDPs, we need a NN architecture that efficiently reduces the number of parameters required to compute all the functions that compose the solution.

The main idea of CNN is to use **filters** (or masks) on top of the input in order to reduce the input to smaller portions. But the point is to use the same weights for all the patches selected by the filter in the input, in order to produce a reduced version of the original input data.


Patch a
Filter f
Initial data matrix A
Convolution Matrix (Output)
$= \sum_{i=1}^{3}\sum_{j=1}^{3} a_{i,j}f_{i,j}$ = -5
= -4
= -16
...

Example from https://www.analyticsvidhya.com/blog/2018/12/guide-convolutional-neural-network-cnn/

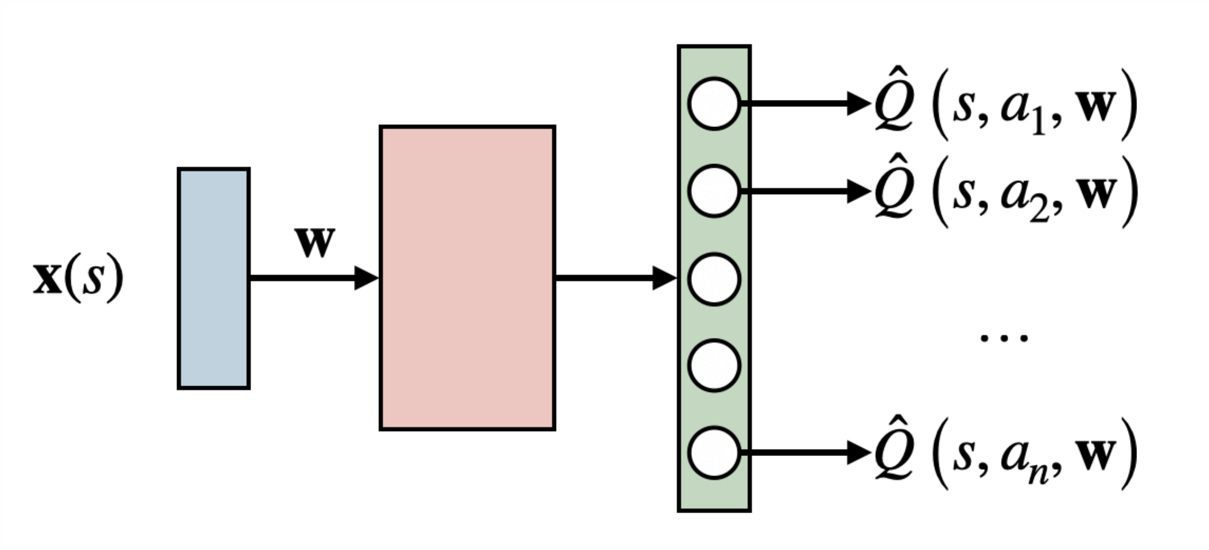
Through this mechanism, CNNs are particularly suited in their different layers for representing different features of the problem space.

In addition to one or more convolutinal layers, CNNs are typically equipped also with pooling layers, to compress information and a final fully connected set of layers.

Neural Networks in RL

Theory tells us that convergence and stability of non linear approximation training process can be a very hard task. One of the first practical succesfull temptative was **Deep Q-Networks (DQNs)** that are based on the following mai ideas:

- A convolutional network gets images from input (DQN was trained on Atari games)
- The CNN is a function approximator for $Q(s, a)$
- The reward is the game score
- We update parameters using signal taken from the rewards by back propagation


$\mathbf{x}(s)$
$\mathbf{w}$
$\hat{Q}(s, a_1, \mathbf{w})$
$\hat{Q}(s, a_2, \mathbf{w})$
...
$\hat{Q}(s, a_n, \mathbf{w})$

Objective function (Loss)

$$L(w_t) = \mathbb{E}_{s,a,r,s'}\left[\left(y_t - \hat{Q}_t(s, a, \mathbf{w})\right)^2\right] \tag{5}$$

We compute $\hat{Q}(s, a; \hat{\mathbf{w}})$ with a memory set of parameters $\hat{\mathbf{w}}$. The target value is:

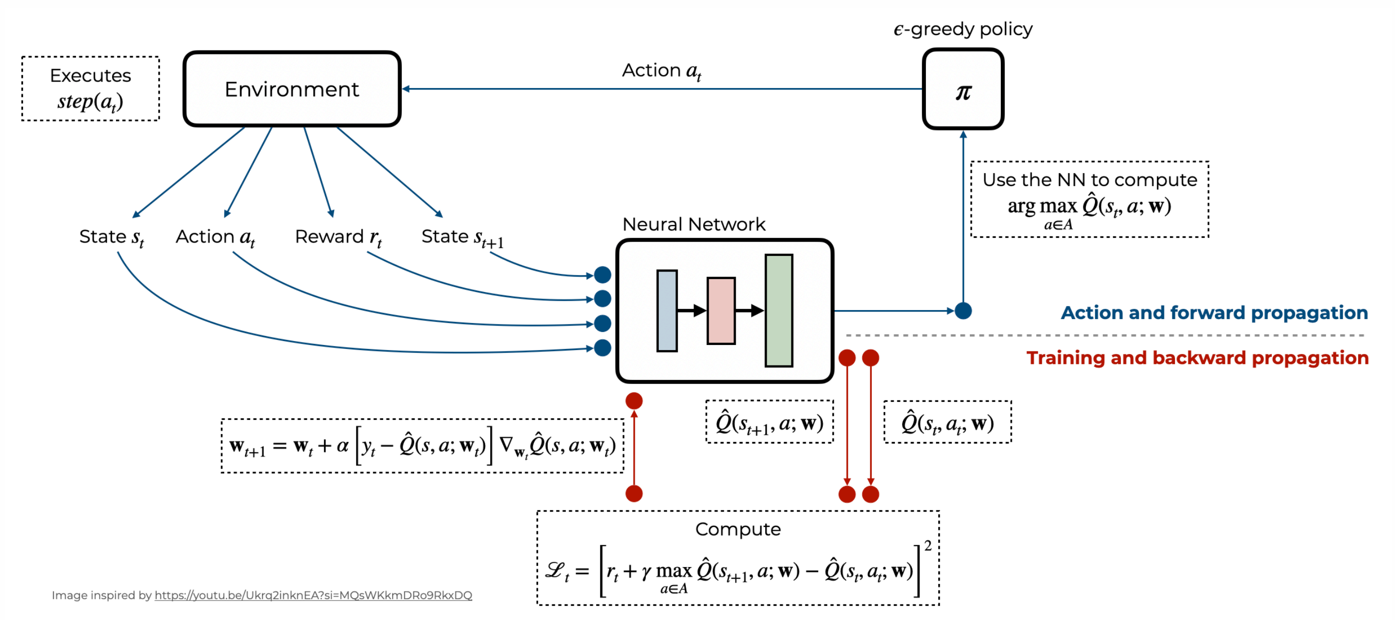
$$y_t = r + \gamma \max_{a' \in A} \hat{Q}(s', a'; \hat{\mathbf{w}}) \tag{6}$$

The update rule is

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[y_t - \hat{Q}(s, a; \mathbf{w}_t)\right] \nabla_{\mathbf{w}_t} \hat{Q}(s, a; \mathbf{w}_t) \tag{7}$$

With $\hat{\mathbf{w}}$ that is updated batch every k steps, such that $\hat{\mathbf{w}} \leftarrow \mathbf{w}_t$.

Workflow


ϵ-greedy policy
Executes $step(a_t)$
Environment
Action a_t
π
Use the NN to compute $\arg\max_{a \in A} \hat{Q}(s_t, a; \mathbf{w})$
State s_t
Action a_t
Reward r_t
State s_{t+1}
Neural Network
Action and forward propagation
Training and backward propagation
$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[y_t - \hat{Q}(s, a; \mathbf{w}_t) \right] \nabla_{\mathbf{w}_t} \hat{Q}(s, a; \mathbf{w}_t)$
$\hat{Q}(s_{t+1}, a; \mathbf{w})$
$\hat{Q}(s_t, a_t; \mathbf{w})$
Compute
$\mathscr{L}_t = \left[r_t + \gamma \max_{a \in A} \hat{Q}(s_{t+1}, a; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w}) \right]^2$
Image inspired by https://youtu.be/Ukrq2inknEA?si=MQsWKkmDRo9RkxDQ

This architecture however produces a lot of instability in the training process and potentially divergence in the estimations. In order to stabilize learning, two main methods are used.

Deep Q-learning

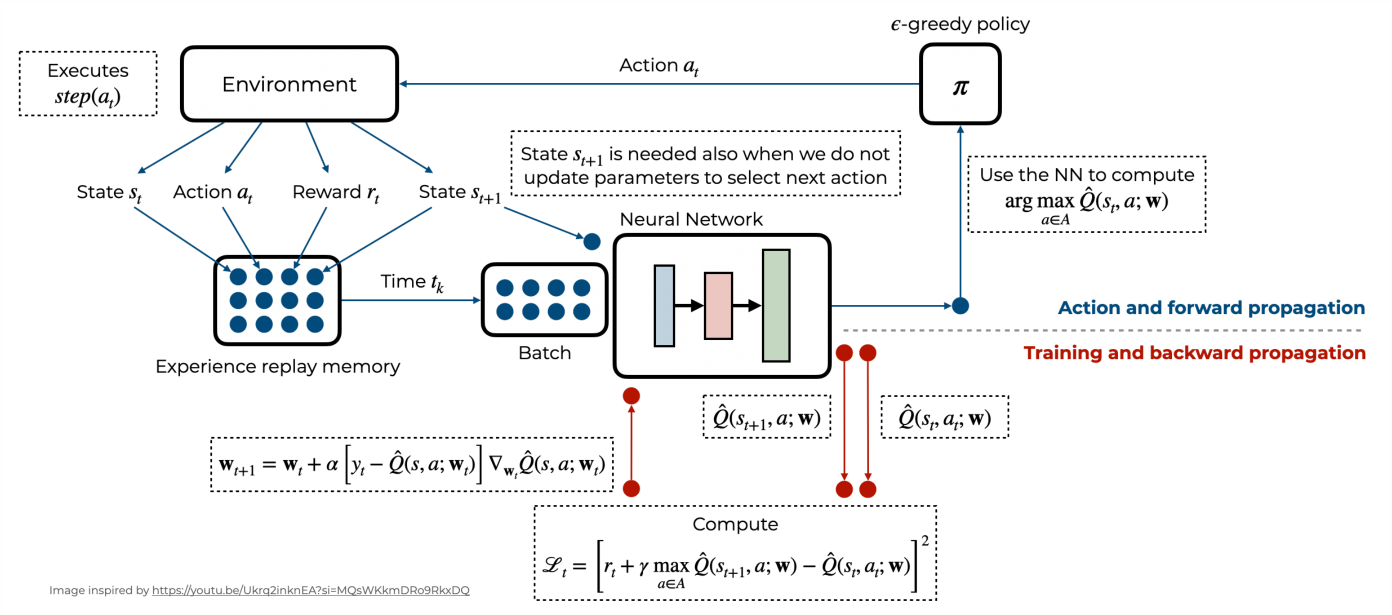
> Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., ... & Hassabis, D. (2015). Human-level control through deep reinforcement learning. *nature*, *518*(7540), 529-533.

Experience replay

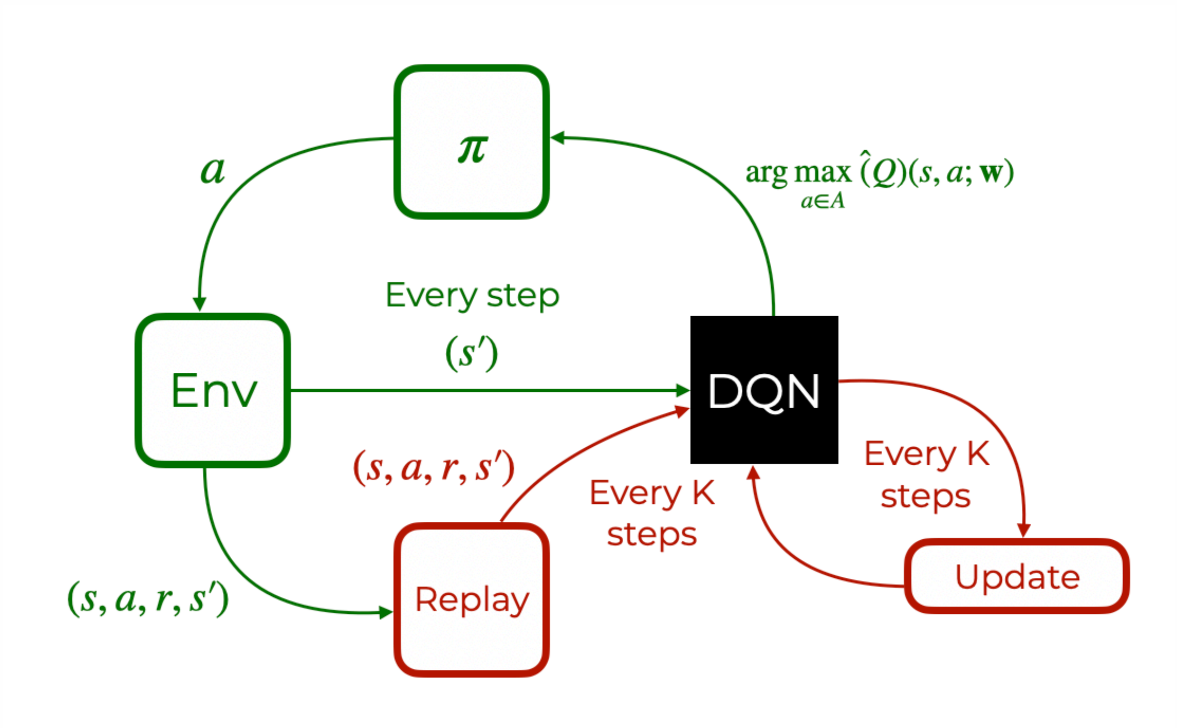
One of the issues with DQN is that the Agent experiences during exploration are correlated. This may produce divergence during the learning process (see Baird's counterexample). One way to avoid this is to sample **some batches from previous explorations**, with the goal of **re-using** them and **reducing variance**.

Procedure:

- we explore the environment and observe $\langle o_t = s_t, a_t, r_t, s_{t+1} \rangle$
- instead of updating $\mathbf{w_t}$ by o_t, we store o_t in a memory
- every k steps, we randomly sample some o_i from the memory and we use them to perform the update


Executes $step(a_t)$
Environment
Action a_t
ϵ-greedy policy
π
State s_t
Action a_t
Reward r_t
State s_{t+1}
State s_{t+1} is needed also when we do not update parameters to select next action
Use the NN to compute $\arg\max_{a\in A} \hat{Q}(s_t, a; \mathbf{w})$
Neural Network
Time t_k
Experience replay memory
Batch
Action and forward propagation
Training and backward propagation
$\hat{Q}(s_{t+1}, a; \mathbf{w})$
$\hat{Q}(s_t, a_t; \mathbf{w})$
$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[y_t - \hat{Q}(s, a; \mathbf{w}_t)\right] \nabla_{\mathbf{w}_t} \hat{Q}(s, a; \mathbf{w}_t)$
Compute
$\mathscr{L}_t = \left[r_t + \gamma \max_{a\in A} \hat{Q}(s_{t+1}, a; \mathbf{w}) - \hat{Q}(s_t, a_t; \mathbf{w})\right]^2$
Image inspired by https://youtu.be/Ukrq2inknEA?si=MQsWKkmDRo9RkxDQ

The workflow is



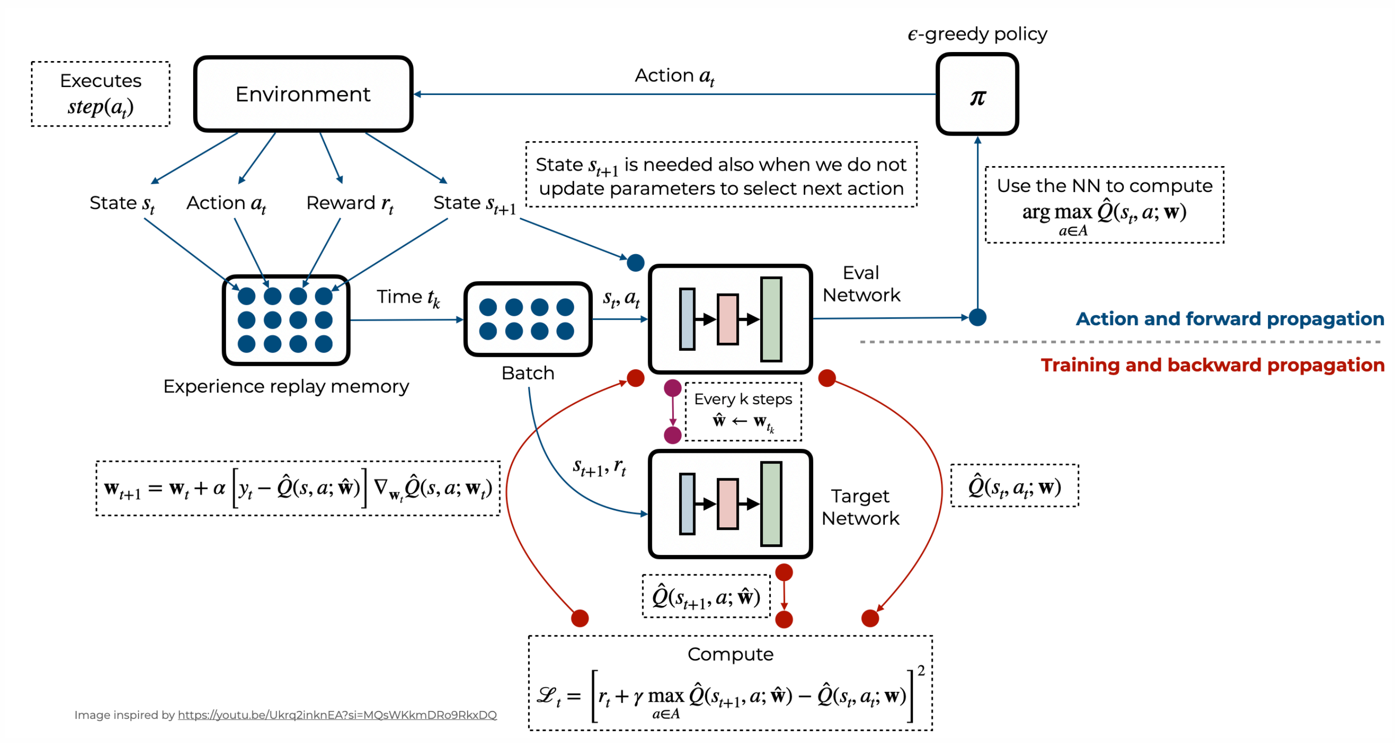
Fixed Q-targets

Another issue for instablity is the fact that the target for learning, that is $\hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w})$, continuously changes due to the update of $\mathbf{w}$.

The solution is to keep a separate set of parameters $\hat{\mathbf{w}}$ that we use to compute the target, such that Q-values are computed differently for s_t and s_{t+1}:

$$s_t \rightarrow \hat{Q}(s_t, a_t; \mathbf{w}) \;\; ; \;\; s_{t+1} \rightarrow \hat{Q}(s_{t+1}, a_{t+1}; \hat{\mathbf{w}}) \tag{8}$$

Then, $\hat{\mathbf{w}}$ are updated only with periodic updates at time k as $\hat{\mathbf{w}} \leftarrow \mathbf{w}_{t_k}$, in order to keep a stable target for at least k steps.



ϵ-greedy policy
Executes $step(a_t)$
Environment
Action a_t
π
State s_{t+1} is needed also when we do not update parameters to select next action
Use the NN to compute $\arg\max_{a \in A} \hat{Q}(s_t, a; \mathbf{w})$
State s_t
Action a_t
Reward r_t
State s_{t+1}
Time t_k
s_t, a_t
Eval Network
Action and forward propagation
Training and backward propagation
Experience replay memory
Batch
Every k steps $\hat{\mathbf{w}} \leftarrow \mathbf{w}_{t_k}$
$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[y_t - \hat{Q}(s, a; \hat{\mathbf{w}}) \right] \nabla_{\mathbf{w}_t} \hat{Q}(s, a; \mathbf{w}_t)$
s_{t+1}, r_t
Target Network
$\hat{Q}(s_t, a_t; \mathbf{w})$
$\hat{Q}(s_{t+1}, a; \hat{\mathbf{w}})$
Compute
$\mathscr{L}_t = \left[r_t + \gamma \max_{a \in A} \hat{Q}(s_{t+1}, a; \hat{\mathbf{w}}) - \hat{Q}(s_t, a_t; \mathbf{w}) \right]^2$
Image inspired by https://youtu.be/Ukrq2inknEA?si=MQsWKkmDRo9RkxDQ

Control rewards

Another issue addressed in the original paper on DQN was the fact that large rewards with outliers affect the dynamic of learning using MSE, in that the gradients could be very large and the update of parameters too fast. A simple solution to this is to just clipping the rewards value in the range $[-1, +1]$.

Issues and extensions

Double DQN (DDQN)

> Hasselt, H. (2010). Double Q-learning. *Advances in neural information processing systems*, *23*.

The idea starts from the observation that with the following estimation of target, we may overestimate Q-values, because we use out current weights to select a'

$$r + \gamma \max_{a' \in A} \hat{Q}(s', a'; \hat{\mathbf{w}}) \tag{9}$$

The main idea of DDQN is to exploit the evaluation network (with $\mathbf{w}$) to select the action and the target network to actually estimate the target.

$$r + \gamma \hat{Q} \left(s', \arg\max_{a' \in A} \hat{Q}(s', a'; \mathbf{w}), \hat{\mathbf{w}} \right) \tag{10}$$

Prioritized Experience Replay

> Schaul, T., Quan, J., Antonoglou, I., & Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Here the idea is that some of the experience in the Replay memory are more useful than others for learning. So, instead of sampling with uniform probability distribution, we can just associate to each observation (s, a, r, s') a measure called **Bellman error** that, given $\hat{y}$ has the target, is:

$$e = \mid \hat{y} - \hat{Q}(s, a; \mathbf{w}) \mid \tag{11}$$

Then, we can prioritize samples with large error values in order to speed up the learning process.

Dueling Architecture

> Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M., & Freitas, N. (2016, June). Dueling network architectures for deep reinforcement learning. In *International conference on machine learning* (pp. 1995-2003).

Here the idea is to split the Q-value function in two components:

- The action independent value function $V(s; \mathbf{v})$
- An advantage function that ideally measures the advantage with respect to the state-value we have by selecting an action $A(s, a; \mathbf{w})$

The final form of the Q-value function will be

$$Q(s, a) = V(s; \mathbf{v}) + A(s, a; \mathbf{w}) \tag{12}$$

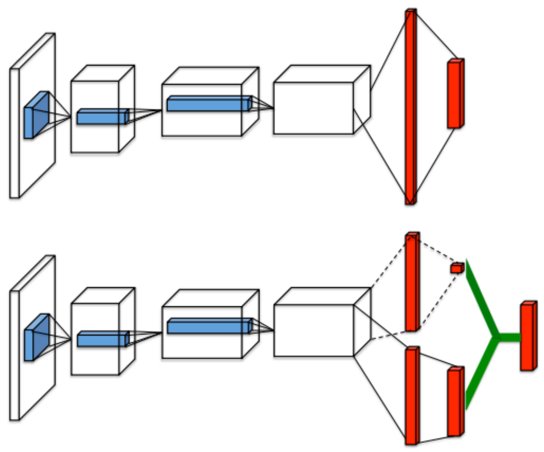


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.