IEOR 265 - Lecture 8
Infinite Horizon DP: Value Iteration and Deep Q-Learning

1 Infinite-Horizon Dynamic Programming

In order to study the advanced approximate DP/ reinforcement learning (RL) algorithms, we need to analyze the DP problem in the infinite-horizon setting, that is when we let the number of stages N go towards infinity. We will not dive deep into the theory of infinite-horizon DP, but we will provide a general overview and analyze the key results. The motivation to study problems with infinite-horizon lies in the fact that many relevant DP/RL problems contain very long horizon, for example: in self-driving it is common to have a planning horizon of 8-10 seconds ahead, with re-planning every 200-300 milliseconds. If we consider a trajectory that takes, say 30 minutes, the number of stages in this DP becomes extremely large. A similar reasoning can be seen on AI software that play games, or robots that execute complex and long sequences of movements.

Formally, an infinite-horizon DP is given by:

$$J_0^*(x_0) = \min_{\pi\in\Pi} \lim_{N\to\infty} \big\{\mathbb{E}_{w_{k_{k\geq 0}}}\big[\sum_{k=0}^{N-1} \alpha^k g_k(x_k,\mu(x_k),w_k)\big]\big\} \tag{1}$$

where $\alpha \in [0,1]$ is some scalar. When $\alpha < 1$, it is often called the *discount factor*, and it is tied to the intuitive notion that the future costs matter to us less than the same costs if they were incurred in the present (or as given by the classical phrase: "time is money").

In our analyzes, we will focus on *discounted problems*, where $\alpha < 1$. We start our analyzes by linking the infinite-horizon DP problem to the finite N-stage DP problem. We make the following assumptions:

1. both the dynamics function and the stage cost function do not change over time (that is, they are *time-invariant*)
2. the disturbances are all i.i.d..
3. the state and control spaces do not change over time.

Now consider the following sequence of iterations:

$$J^{(t+1)}(x) = \min_{u\in U(x)} \big\{\mathbb{E}_w\big[g(x,u,w) + \alpha J^{(t)}(f(x,u,w))\big]\big\}\ ,\ \forall t = \{0,1,2,...\} \tag{2}$$

given some initial cost function $J^{(0)}(x)$. Let's describe the above in words: starting from $J^{(0)}$ we are iteratively constructing a sequence of functions:

$$\{J^{(0)}, J^{(1)}, J^{(2)}, ...\} \tag{3}$$

Eq(2) is known as the **Value Iteration Algorithm** (or simply VI). In the infinite-horizon DP, we would like to answer the following questions:

1. If we apply the VI algorithm infinitely many times, does it converge to something? Namely, is the following true:
$$J^*(x) = \lim_{t\to\infty} J^{(t)}(x),\ \forall x \in \mathcal{X} \tag{4}$$
2. If it does converge, then the following holds true:
$$J^*(i) = \min_{u\in U(i)} \left\{ \sum_{j}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^*(j)) \right\} \tag{5}$$
The equation above is probably the single most famous equation in the entire Dynamic Programming field and is called the **Bellman's Equation**. Is $J^*(x)$ the optimal cost-to-go of the infinite-horizon DP ,Eq(1), under Assumptions 1-3? Namely, is it true that:
$$J^*(x) = J_0^*(x) \tag{6}$$
3. Lastly, if the above is true, then the optimal cost-to-go does not depend on the particular stage we find ourselves in. Hence if we let the policy $\pi = \{\mu, \mu, ...\}$, where $\mu(x)$ solves Eq(5), then it is the optimal policy. Such a policy is called **stationary**: the optimal action to take given state x **does not** depend on which stage we are when we reach x. This in turn allow us to focus our attention in solely searching for *stationary policies.*

It turns out that all the above is true for discounted infinite-horizon DP problems, as we shall see.

1.1 Infinite-Horizon MDP and the Value Iteration

As we saw on previous lecture, if the state-space of the DP problem is finite, then there is one-to-one equivalence between finite-state DP and Markov Decision Processes (MDP). This is also true when the horizon in infinite (we leave this verification as an exercise). It will suit our purposes to present the results and the following algorithms in the MDP form.

To that end, we use i to denote the states, from a set of integers $i \in \{1, ..., n\}$; we let $U(i)$ be the set of available controls at state i; we define $p_{ij}(u)$ as the probability of transitioning from i to j, given that the chosen control is u; and lastly we define the stage cost to be $g(i,u,j)$ on said transition. Then the VI algorithm becomes:
$$J^{(t+1)}(i) = \min_{u\in U(i)} \left\{ \sum_{j}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^{(t)}(j)) \right\} \tag{7}$$

for all states $i \in \{1, ..., n\}$ and any initial conditions $J^{(0)}(1), ..., J^{(0)}(n)$.

The following theorem establishes the convergence of the VI algorithm and the optimality of stationary policies:

Theorem 1 ***(Convergence of VI):*** *Given any initial conditions $J^{(0)}(1), ..., J^{(0)}(n)$, the sequence $\{J^{(t)}\}_{t\geq 0}$ generated by the VI algorithm:*

$$J^{(t+1)}(i) = \min_{u\in U(i)} \left\{ \sum_{j}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^{(t)}(j)) \right\}$$

converges to $J^*(i)$, for each state $i \in \{1, ..., n\}$, where $J^*(i)$ is the finite optimal value function of the infinite-horizon MDP and is the unique solution of the ***Bellman's Equation****:*

$$J^*(i) = \min_{u\in U(i)} \left\{ \sum_{j}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^*(j)) \right\}$$

In addition, a stationary policy $\pi = \{\mu, \mu, ...\}$ is optimal if and only if $\mu(i)$ attains the minimum (i.e.: it solves) the Bellman Equation for all states $i \in \{1, ..., n\}$.

proof: The proof of this amazing theorem is detailed in [1].

As highlighted before, this theorem is so powerful because it allow us to narrow our search for optimal policies: we need not look for any arbitrary policy, thus focusing our search to stationary policies, as they will be optimal as long as they solve the Bellman's Equation. This fact lies in the core of several approximate DP/RL methods.

1.2 Example: mean-passage time of a Markov Chain

Consider the following DP problem: Let $i \in \{1, ..., n, T\}$ where T is a special termination state. The goal is to design a policy to reach state T as fast as possible, on average, starting from any state $i \neq T$. Then we can model the problem as follows:

$$\alpha = 1 \qquad g(i,u,j) = \begin{cases} 1, & \text{if } i \neq T, j \in \{1, ..., n, T\}, \forall u \in U(i) \\ 0, & \text{if } i = T, j \in \{1, ..., n, T\}, \forall u \in U(i) \end{cases} \tag{8}$$

The transition probabilities are given by $p_{ij}(u)$ and if reach state T, we stay at T forever with probability one. Then the optimal cost-to-go from state i, $J^*(i)$, is the minimum expected time to reach state T from state i and uniquely solves the Bellman's Equation:

$$J^*(i) = \min_{u\in U(i)} \left\{ (1 + \sum_{j=1}^{n} p_{ij}(\mu) J^*(j) \right\} \tag{9}$$

Now assume that there is only only one set of controls for each state i (so $U(i)$ is a singleton set). So the MDP gets reduced to a Markov Chain and the equation above gets reduced to:

$$J^*(i) = 1 + \sum_{j=1}^{n} p_{ij} J^*(j) \tag{10}$$

which are the equations that provide us the mean first passage time from i to T in the associated Markov Chain.

1.3 Example: Optimal Stopping Problem

Consider the problem of selling a house which we covered before in past lectures. But now, suppose that the horizon is infinite: at each stage an offer of value w_k is given for the house, and you get to either sell the house an invest the money in a bank with interest r, or not. The offers are i.i.d. and are infinite, namely a new offer comes at every stage for an infinite number of stages. We saw before that for the finite, the optimal policy reduces to computing a series of thresholds and once the offer received is above the threshold we sell it (we refer to Lecture 4 for the analysis of finite-horizon case).

Let's suppose the i.i.d. offers w follow the distribution below:

$$w = \begin{cases} 50, \text{ w.p. } \frac{1}{2} \\ 75, \text{ w.p. } \frac{1}{4} \\ 100, \text{ w.p. } \frac{1}{4} \end{cases} \tag{11}$$

Let the interest rate be r and let the discount factor be $\alpha = \frac{1}{1+r}$.

Similarly to the finite-horizon case, we can definite our state to be the current outstanding offer, if haven't accepted a offer yet, or be a special termination state T. Hence, since the offers take only three possible values, our states belong to the set $\{50, 75, 100, T\}$. Let's solve the infinite-horizon DP problem using the Value Iteration Algorithm starting with $J^{(0)}(50) = J^{(0)}(75) = J^{(0)}(100) = 75$ and $J^{(0)}(T) = 0$. Then we apply the VI algorithm:

$$J^{(t+1)}(i) = \min_{u \in U(i)} \left\{ \sum_{j}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^{(t)}(j)) \right\} \tag{12}$$

which, for this problem, reduces to (we omit the equation for state T, as it is trivial):

$$J^{(t+1)}(i) = \\ \max\left\{ i, \frac{1}{1+r}\left(\frac{1}{2}J^{(t)}(50) + \frac{1}{4}J^{(t)}(75) + \frac{1}{4}J^{(t)}(100)\right)\right\} \text{ for } i \in \{50, 75, 100\} \tag{13}$$

where we note, that our problem is trying to maximize revenue and algorithm iteration (often called the *VI-step*), consists of comparing the value of accepting the current offer against the future expected value of waiting one more time period discounted to the present. From this equation, we can derive the following optimal policy:

$$\mu^*(i) = \begin{cases} \text{sell, if } i > c \\ \text{do not sell, otherwise} \end{cases} \tag{14}$$

where c is a constant threshold defined as:

$$c = \frac{1}{1+r}\left(\frac{1}{2}J^*(50) + \frac{1}{4}J^*(75) + \frac{1}{4}J^*(100)\right) \tag{15}$$

where $J^*(50), J^*(75)$, and $J^*(100)$ are the solution of the Bellman's Equation applied to this problem. Suppose the interest rate $r = 10\%$. Then, by applying the VI-step many times, we converge to the optimal cost-to-go values:

$$J^*(50) = 72.92,\ J^*(75) = 75,\ J^*(100) = 100, \text{ and } J^*(T) = 0 \tag{16}$$

and the threshold $c = 72.92$. Hence, the optimal policy can be described in words: " the optimal policy is to wait until an offer with value more than 72.92 arrives. When that happens we accept it and sell the house". To this problem, it is equivalent to waiting until an offer with value 75 or 100 arrives, whichever happens first.

An interesting situation to consider is when $r = 0$. In this case, there is no discounting of the future, and the optimal policy reduces to "wait" indefinitely until the highest possible offer appears. This holds true, even if the probability to receive the highest offer becomes very small. In this case, we might wait a very very long time. This fact highlights the benefit of using discounting in decision making across time periods!

2 Fitted Value Iteration

Let's return now to approximations. We state again the VI algorithm for the MDP:

$$J^{(t+1)}(i) = \min_{u \in U(i)} \left\{ \sum_{j}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^{(t)}(j)) \right\}, \forall t \in \{0, 1, 2, ...\} \tag{17}$$

Now suppose we use an approximation architecture, say a Deep Neural Network (DNN), to provide the approximate cost-to-go from any state i: That is a function $\tilde{J}(i, \theta)$, where θ comprises all the parameters of the DNN. Starting from an initial configuration $\theta^{(0)}$, suppose we are able to generate some sample states $\{i^0, ..., i^S\}$ for a total of S samples. Then we can apply the VI algorithm for one iteration (VI-step) and obtain:

$$\beta(i^s) = \min_{u \in U(i)} \left\{ \sum_{j=1}^{n} p_{i^s j}(u)(g(i^s, u, j) + \alpha \tilde{J}^0(j, \theta^{(0)})) \right\}, \forall s = \{1, ..., S\} \tag{18}$$

where $\beta(i^s)$ is the approximate cost-to-go for state i^s after performing one VI-step. These cost values can then be used as "labels" in the following Regression problem:

$$\theta^{(1)} \in \arg\min_{\theta} \left\{ \sum_{s=1}^{S} \left(\tilde{J}(i^s; \theta) - \beta(i^s) \right)^2 \right\} \tag{19}$$

and after solving it, say by some iterative gradient method, we update the architecture parameter from $\theta^{(0)}$ to $\theta^{(1)}$. As state before, Eq(16) is called the *VI-step* and Eq(17) is called the *Regression-step.* Sequential application of the VI-step and the Regressio-step constitutes the **Fitted Value Iteration Algorithm**, which is presented in pseudo-code format in Algorithm 1.

We highlight the fact that the suboptimal policy is obtained by the 1-step lookahead minimization:

$$\tilde{\mu}(i) \in \arg \min_{u \in U(i)} \left\{ \sum_{j=1}^{n} p_{ij}(u)(g(i^s, u, j) + \alpha \tilde{J}(j; \theta^{(T)})) \right\} \tag{20}$$

where T is some arbitrary scalar, denoting the total number of iterations performed, which is in line with the finite-horizon version of the fitted VI algorithm, covered in the last lecture. However, this time around, it is presented in the infinite-horizon setting.

Algorithm 1 Fitted Value Iteration: infinite-horizon

```
Input: Initial DNN parameters $\theta^0$ and architecture $J(i, \theta^{(0)})$.
1: for $p = 1, ..., P$: do (outer-iterations)
2:     Collect a batch of S samples $(i^1, ..., i^S)$.
3:     for $t = 0, ..., T$ do (VI-iterations)
4:         for $i = i^1, ..., i^S$: do (so, for each sample)
5:             Peform the VI step:
               $$\beta(i^s) = \min_{u \in U(i)} \left\{ \sum_{j=1}^{n} p_{i^s j}(u)(g(i^s, u, j) + \alpha \tilde{J}(j; \theta^{(t)})) \right\}, \forall s = \{1, ..., S\}$$
6:         end for
7:         Solve the Regression:
           $$\theta^{(t+1)} \in \arg\min_{\theta} \left\{ \sum_{s=1}^{S} \left( \tilde{J}(i^s; \theta) - \beta(i^s) \right)^2 \right\}$$
8:     end for
9: end for
Output: A suboptimal policy is obtained via 1-step lookahead minimization:
   $$\tilde{\mu}(i) \in \arg\min_{u \in U(i)} \left\{ \sum_{j=1}^{n} p_{ij}(u)(g(i^s, u, j) + \alpha \tilde{J}(j; \theta^{(T)})) \right\}$$
```

Now this algorithm brings forth a new set of questions:

1. Does it converge? Namely, is it true that:

$$J^*(i) = \lim_{t \to \infty} \tilde{J}(i, \theta^{(t)}) \tag{21}$$

   where $J^*(1), ..., J^*(n)$ is the solution of the Bellman's Equation.

2. How can we execute the VI-step if the number of states is extremely large, that is n is very large. What if the transition probabilities $p_{ij}(u)$ are unknown?

3. The algorithm heavily relies on obtaining "good" samples? How can we generate those samples?

On the remaining of this lecture notes, we will proceed to provide answers to those questions.

2.1 Convergence Pathology of Fitted Value Iteration

It turns out the the Fitted Value Iteration (Algorithm 1) does not converge in general. We illustrate the inherent pathology of the algorithm by a simple example.

Consider a two-state MDP, with states $i = \{1, 2\}$, which transitions and costs are represented in the figure below:

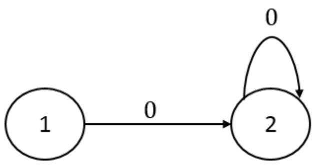


Figure 1: a simple MDP: the costs associated with each transition are zero.

Note that at each state i there is only one single control available. Hence, the Bellman's Equation:

$$J^*(i) = \min_{u \in U(i)} \left\{ \sum_{j}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^*(j)) \right\} \tag{22}$$

gets reduced to:

$$J^*(1) = \alpha J^*(2) \qquad J^*(2) = \alpha J^*(2) \tag{23}$$

which we can readily solve to conclude that $J^*(1) = J^*(2) = 0$. By the same token, the exact VI algorithm is given by:

$$J^{(t+1)}(1) = \alpha J^{(t)}(2) \qquad J^{(t+1)}(2) = \alpha J^{(t)}(2) \tag{24}$$

which converges to the solution of the Bellman Equation, as expected. Now suppose that we use the following approximate function:

$$\tilde{J}(i, \theta) = i\theta \tag{25}$$

where θ is a scalar and it is the architecture parameter. Note that if we pick $\theta = 0$, we will recover exactly the optimal cost-to-go (the solution of the Bellman's Equation). Let's say we pick some initial $\theta^{(0)} = 1$, however. In addition suppose we generate n_1 samples of state 1 and n_2 samples of state 2. Then, the relative frequencies (or "weight") of each sampled state is:

$$\xi_1 = \frac{n_1}{n_1 + n_2} \qquad \xi_2 = \frac{n_2}{n_1 + n_2} \tag{26}$$

Let's see what happens if we use the Fitted Value Iteration Algorithm on this problem. We start by computing the VI step:

$$\beta(1) = \alpha \tilde{J}(2, \theta^{(t)}) = 2\alpha\theta^{(t)} \qquad \beta(2) = \alpha \tilde{J}(2, \theta^{(t)}) = 2\alpha\theta^{(t)} \tag{27}$$

Then we compute the Regression step:

$$\theta^{(t+1)} \in \arg\min_{\theta} \left\{ \sum_{s=1}^{S} \left(\tilde{J}(i^s, \theta) - \beta(i^s) \right)^2 \right\} \tag{28}$$

which, by using the relative frequencies ξ_1 and ξ_2 reduces to:

$$\theta^{(t+1)} \in \arg\min_{\theta} \left\{ \xi_1 \big(\theta - 2\alpha\theta^{(t)}\big)^2 + \xi_2 \big(2\theta - 2\alpha\theta^{(t)}\big)^2 \right\} \tag{29}$$

Now this is a simple least-squares regression with a linear architecture. So the optimal solution is available to us in closed-form and is obtained after taking the derivative w.r.t. θ and setting it zero, which yields:

$$\theta^{(t+1)} = \alpha C \theta^{(t)} \qquad \text{where} \quad C = \frac{2(\xi_1 + 2\xi_2)}{\xi_1 + 4\xi_2} > 1 \tag{30}$$

Now if $\alpha C > 1$, then $\theta^{(t)} \to \infty$ as $t \to \infty$. Hence the Fitted VI algorithm will diverge away from the correct choice of $\theta = 0$.

The reason behind the lack of convergence guarantee, is that the VI-step is a contraction w.r.t. to the $||\cdot||_\infty$ norm while the Regression-step is a contraction w.r.t. to the $||\cdot||_2$ norm. The composition of two contractions of different norms, is not itself a contraction. We refer to the course textbook [1] for an in-depth discussion of convergence and contractions of the Value Iteration Algorithm. For our purposes here, it will suffice to state that the fitted VI algorithm **does not** converge in general.

Lastly, we draw attention to the fact that, in the example, if for instance we sampled the second state much more frequently then the first state so as we let $\xi_2 \to 1$, then the algorithm would, in fact, converge. Even if convergence can not be guaranteed in general, by sampling in a "good" way, we may escape the non-convergent behavior. The take-way again, is the key role of sampling in the performance of approximation algorithms. We shall delve deeper into the sampling issue as we present the next algorithm which attempts to overcome the issues raised by the Fitted VI Algorithm.

3 Deep Q-Learning

In order to solve the three questions posed in the previous question, we need to address the issue of sampling and how to bypass the curse of dimensionality in performing the VI-step. To that end, let's start by restating the Bellman Equation:

$$J^*(i) = \min_{u \in U(i)} \left\{ \sum_{j}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^*(j)) \right\} \tag{31}$$

And the Q-factors, introduced in previous lectures:

$$Q^*(i,u) = \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha J^*(j)) \tag{32}$$

Note that, we can then write the equivalent form of the Bellman Equation:

$$J^*(i) = \min_{u \in U(i)} \left\{ Q^*(i,u) \right\} \tag{33}$$

and then solely in terms of the Q-factors:

$$Q^*(i,u) = \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha \min_{v \in U(j)} Q^*(j,v)), \; \forall i = \{1, ..., n\}, \forall u \in U(i) \tag{34}$$

In a similar fashion, we can write an equivalent form of the exact VI algorithm for the Q-factors, starting with some Q-factors $Q^{(0)}(i,u)$ for all $i \in \{1,...,n\}$ and $u \in U(i)$:

$$Q^{(t+1)}(i,u) = \sum_{j=1}^{n} p_{ij}(u)(g(i,u,j) + \alpha \min_{v \in U(j)} Q^{(t)}(j,v)),\ \forall i = \{1,...,n\}, \forall u \in U(i) \tag{35}$$

This exact VI algorithm is equivalent to the exact VI algorithm using Value Functions, so it shares the nice convergence properties of the exact scheme. Now let's introduce approximations: we will use some approximation architecture (say a DNN) to approximate the Q-factors. Hence we shall have:

$$\tilde{Q}_\theta(i,u) \approx Q^*(i,u),\ \forall i = \{1,...,n\}, \forall u \in U(i) \tag{36}$$

where θ represents the architecture parameters (we leave θ on the subscript, to avoid adding to many arguments inside the Q-factors).

Note that now, the input to such DNN are both state and control, so for every state-control pair (i,u) the DNN outputs an approximation value of the optimal Q-factor $Q^*(i,u)$ at that pair. Now, let's focus our attention to Eq(35): we highlight the fact the right-hand side of Eq(35) is, in fact, an expectation where given the state-control pair (i,u) we take the average w.r.t. all the possible future states that can be reach from i and the control u is used. Then, we will use Sample Average Approximation (SAA) to approximate this expectation via sampling. Suppose we have some *base policy* at our disposal and we use it to generate the following samples of tuples:

$$\left(i^0,u^0,i^1,g^0\right), \left(i^1,u^1,i^2,g^1\right), ..., \left(i^{S-1},u^{S-1},i^S,g^{S-1}\right) \tag{37}$$

where each tuple (i^s,u^s,i^{s+1},g^s) correspond to transition of the MDP: the current state, current control selected, next state, and cost incurred. Then we can compute the following quantities for each of the S samples:

$$\beta^s = g(i^s,u^s,i^{s+1}) + \alpha \min_{u \in U(i^s)} \{\tilde{Q}_{\theta^{(t)}}(i^{s+1},u)\},\ \forall s \in \{1,...,S\} \tag{38}$$

Each β^s is a 1-sample approximation of the Q-factor $Q(i^s,u^s)$, using the DNN aproximation architecture $\theta^{(t)}$ to provide the approximate Q-factor value at the next state.

A key observation is that Eq(38) can be computed **without** the knowledge of the underlying probability model of the MDP (or in other words, the dynamics of the associated DP problem). By using Q-factors approximations we essentially remove the probability model (the $p_{ij}(u)$'s) from the computations, and we push the "burden" onto the simulation. This is one of the central trade-offs in using approximate Q-factors instead of approximate Value Functions.

As we did in the Fitted Value Iteration, given the "labels" $\beta^s, s \in \{1,...,S\}$, we perform the regression step as follows:

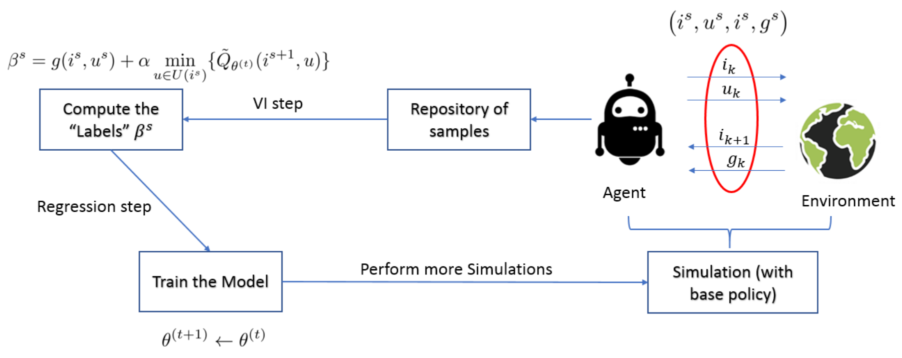
$$\theta^{(t+1)} = \arg\min_\theta \left\{ \sum_{i=1}^{S} \left(\tilde{Q}_\theta(i^s,u^s) - \beta^s\right)^2 \right\} \tag{39}$$

So, the fitted Q-iteration scheme proceed iteratively by applying consecutively the VI-step, Eq(38), and the Regression Step, Eq(39). Lastly, the suboptimal policy is given directly by the approximate Q-factors:

$$\tilde{\mu}(i) = \arg\min_{u \in U(i)} \left\{ \tilde{Q}_{\theta^{(T)}}(i,u) \right\} \tag{40}$$

at the end of T iterations. We highlight the contrast between this sub-optimal policy with the one given by the fitted Value iteration, Eq(20). Here, we **do not need** to know the underlying probabilities, and no summation is required to compute the sub-optimal policy. Again, this shows the benefit of using approximate Q-factors instead of approximate Value Functions.

The fitted Q-iteration scheme can be summarized in the following figure:



(i^s, u^s, i^s, g^s)
$\beta^s = g(i^s, u^s) + \alpha \min_{u \in U(i^s)} \{\tilde{Q}_{\theta^{(t)}}(i^{s+1}, u)\}$
Compute the "Labels" β^s
VI step
Repository of samples
i_k
u_k
i_{k+1}
g_k
Agent
Environment
Regression step
Train the Model
Perform more Simulations
Simulation (with base policy)
$\theta^{(t+1)} \leftarrow \theta^{(t)}$

Figure 2: Fitted Q-Iteration: The agent interacts with the environment via simulation, in a model-free way(without knowing the exact dynamics). The algorithm stores the transition tuples and iteratively applies the VI-step and the Regression-step in order to improve the approximation of the Q-factors.

3.1 Handling the sampling issue: Experience Replay

Note that in order to make the fitted Q-iteration scheme work we require the samples:

$$(i^0, u^0, i^1, g^0), (i^1, u^1, i^2, g^1), ..., (i^{S-1}, u^{S-1}, i^S, g^{S-1}) \tag{41}$$

These samples can be "stringed" together to form a sample *trajectory*. Typically, samples from the same trajectory are highly correlated and this may pose an issue in the Regression-step, Eq(26). To minimize the amount of correlation between samples, we can generate trajectories, in parallel(1); can generate trajectories asynchronously(2); or use a replay buffer(3). We will focus here on the third option, as the other two are closely tied to the usage of GPU or other specific hardware that enable parallelization and multi-processor computation. The idea behind of experience replay (or *Replay Buffer*) is to notice that in the Regression-step the samples are exchangeable: it does not really matter what sample is the first or the second sample, they all have the same "weight" in the summation. Therefore, we can create a repository of transition tuples (i^s, u^s, i^{s+1}, g^s) that are obtained from many different trajectories and at different points. Then, when the Regression-step is to be executed, we sample a batch of size S (uniformly) at random for the repository. This processed is illustrated in figure 3.

Of course, there is no guarantee that the samples are uncorrelated, but the idea is that if the replay buffer is very large compared to the sample batch S, then the degree of correlation will be small. Another important feature is that this repository can be "filled" offline, that is before the sub-optimal

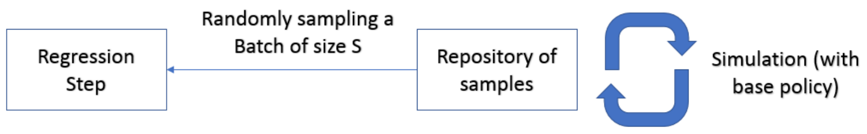

Regression Step
Randomly sampling a Batch of size S
Repository of samples
Simulation (with base policy)

Figure 3: We can store transitions obtained via some base policy, in an unordered fashion in the repository. Then, we generate batches of samples as needed by the Regression-step.

policy needs to be computed in real-time. That allow us to leverage the entire computational power of modern software and hardware to both simulate new sample trajectories and store those transitions efficiently into the replay buffer.

3.2 Generating better samples: improving the base policy

We digress in this subsection to talk about the base policy that is used to generate the sample trajectories. In practice, when the replay buffer is empty (or contain very few transitions), a randomized policy is often enough to collect new transitions (a randomized policy picks any available action uniformly at random). However as the fitted Q-iteration Algorithm progress, samples obtained via this simple randomized policy are often "bad" samples. More formally, bad samples are transitions that are unlikely to occur if we followed the true optimal policy for the underlying MDP. Hence, using those samples in the Regression-step, will not improve the performance of the approximation architecture.

To remedy this, we have to improve the base policy. This is the core goal of **Policy Iteration** algorithm, which we will cover on the next lecture. For now, it is enough to note that we can use the sub-optimal policy, Eq(40), in place of the randomized base policy. The idea is that, hopefully, we create a virtuous cycle of iterations: We use the samples to improve the approximate Q-factors, then we use the policy based on those approximate Q-factors to generate better samples, which are then used to improve the Q-factors, and so on.

The down-side is somewhat evident: This loop emphasizes exploitation, since the Q-factors that are used in generating samples are the ones being updated by those samples. In practice, it is often the case that in order to ensure that we explore never-visited states and obtain new transitions we can combine both the sub-optimal policy with the randomized policy as follows:

$$\tilde{\mu}^{(t)}(i) = (1 - \epsilon^{(t)}) \arg\min_{u \in U(i)} \left\{ \tilde{Q}_{\theta^{(t)}}(i, u) \right\} + \epsilon^{(t)} \mathrm{A}(U(i)) \qquad (42)$$

where we denote $\mathrm{A}(U(i))$ to be a random variable that selects uniformly at random any element of the set $U(i)$. Here the sequence $\{\epsilon^{(t)}\}_{t \geq 1}$, where $0 \leq \epsilon^{(t)} \leq 1$, is often called the "exploration schedule": at iteration t, we use the sub-optimal policy based on the current approximate Q-factors w.p. $(1 - \epsilon^{(t)})$ or we do something random w.p. $\epsilon^{(t)}$.

3.3 Deep Q-Networks(DQN) Algorithm

Let us restate the fitted Q-iteration algorithm, by writing the VI-step:

$$\beta^s = g(i^s, u^s) + \alpha \min_{u \in U(i^s)} \{\tilde{Q}_{\theta^{(t)}}(i^{s+1}, u)\}, \forall s \in \{1, ..., S\} \tag{43}$$

and the Regression-step:

$$\theta^{(t+1)} = \arg\min_{\theta} \left\{ \sum_{i=1}^{S} \big(\tilde{Q}_\theta(i^s, u^s) - \beta^s\big)^2 \right\} \tag{44}$$

Let's substitute Eq(43) into Eq(44):

$$\theta^{(t+1)} = \arg\min_{\theta} \left\{ \sum_{i=1}^{S} \big(\tilde{Q}_\theta(i^s, u^s) - g(i^s, u^s) - \alpha \min_{u \in U(i^s)} \{\tilde{Q}_{\theta^{(t)}}(i^{s+1}, u)\}\big)^2 \right\} \tag{45}$$

Instead of solving the above minimization problem to (local) optimality via some gradient method, we will perform a *single* gradient step, and then we will proceed to obtain more samples. A single gradient-step can be obtained by differentiating Eq(32) w.r.t. θ:

$$\theta^{(t+1)} =$$

$$\theta^{(t)} - \gamma^{(t)} \left\{ \sum_{i=1}^{S} \nabla_\theta \tilde{Q}_{\theta^{(t)}}(i^s, u^s) \big(\tilde{Q}_{\theta^{(t)}}(i^s, u^s) - g(i^s, u^s) - \alpha \min_{u \in U(i^s)} \{\tilde{Q}_{\theta^{(t)}}(i^{s+1}, u)\}\big) \right\} \tag{46}$$

where $\gamma^{(t)}$ is some step-size scalar. We highlight that the goal of the above gradient-step is to minimize what is called the sum of the *Bellman Errors*:

$$\big(\tilde{Q}_{\theta^{(t)}}(i^s, u^s) - g(i^s, u^s) - \alpha \min_{u \in U(i^s)} \{\tilde{Q}_{\theta^{(t)}}(i^{s+1}, u)\}\big)^2, \forall s \in \{1, ..., S\} \tag{47}$$

After performing the gradient-step, we use the suboptimal policy:

$$\tilde{\mu}^{(t+1)}(i) = \arg\min_{u \in U(i)} \left\{ \tilde{Q}_{\theta^{(t+1)}}(i, u) \right\} \tag{48}$$

to generate a new set of samples, that is subsequently added to the replay buffer. Then we repeat towards a new iteration. This process is summarized in Figure 4.

Now we observe that this process introduces one technical challenge: At every gradient-step we are using different samples, which means that the associated objective function is changing across the iterations. This "moving target" effect can me mitigated if we save a previous DNN configuration θ' and use it to compute the target label values β^s. The idea is, then, to keep track of two DNNs: one that changes every iteration that is used to compute the approximate Q-factors; and another, that changes much more slowly that is used to compute the label targets. Thus, the gradient-step gets modified to:

$$\theta^{(t+1)} =$$

$$\theta^{(t)} - \gamma^{(t)} \left\{ \sum_{i=1}^{S} \nabla_\theta \tilde{Q}_{\theta^{(t)}}(i^s, u^s) \big(\tilde{Q}_{\theta^{(t)}}(i^s, u^s) - g(i^s, u^s) - \alpha \min_{u \in U(i^s)} \{\tilde{Q}_{\theta'}(i^{s+1}, u)\}\big) \right\} \tag{49}$$

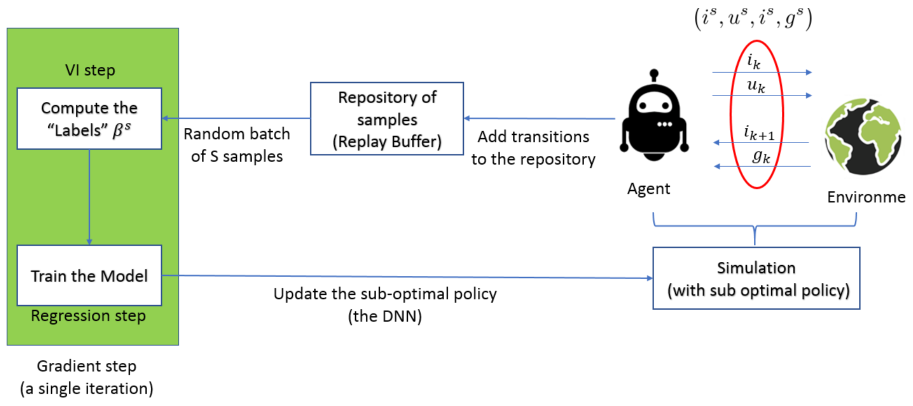

(i^s, u^s, i^s, g^s)
VI step
Compute the "Labels" β^s
Train the Model
Regression step
Gradient step (a single iteration)
Random batch of S samples
Repository of samples (Replay Buffer)
Add transitions to the repository
Agent
i_k
u_k
i_{k+1}
g_k
Environme
Simulation (with sub optimal policy)
Update the sub-optimal policy (the DNN)

Figure 4: Deep Q-Learning: Now we combine the VI-step and the Regression-step into a single gradient-step where we update the DNN parameters; Then, we use the current sub-optimal policy to generare more samples by interacting with the environment, before executing another gradient-step.

The usage of approximate Q-factors, replay buffer, exploration schedule, and two DNN's, comprise the **Deep Q-Networks Algortihm** proposed by [2], which we state in it's basic form as a pseudo-code in Algorithm 2.

The algorithm has many different hyper-parameters: Batch size, target network update frequency, number of iterations, type of architecture, exploration schedule, gradient method used, etc. All these hyper-parameters can be selected based on experience and empirical performance or some via cross-validation. In addition, we highlight that there are many improvements and extensions of this algorithm: Double-Q learning, Adaptive Exploration, multi-step lookahead, etc. But overall, they are all based on the framework described above.

Algorithm 2 DQN Algorithm (Minh et al, 2015)

```
Input: Initial DNN parameters $\theta^0$ and architecture $Q_{\theta^0}(i,u)$. Replay Buffer $\mathcal{B}$.
1: for $p = 1, ..., P$: do (updates on the target network $\theta'$)
2:     Save the network parameters $\theta' \leftarrow \theta$.
3:     for $k = 0, ..., K$ do (obtaining new samples)
4:         Collect $M$ sample transitions $\left\{(i^m, u^m, i^{m+1}, g^m)\right\}_{m=1}^{M}$ using the sub-
       optimal policy:
           $$\tilde{\mu}^{(t+1)}(i) = (1 - \epsilon^{(t+1)}) \arg\min_{u \in U(i)} \left\{\tilde{Q}_{\theta^{(t+1)}}(i,u)\right\} + \epsilon^{(t+1)} \mathrm{A}(U(i))$$
       and add those sample transitions to the Replay Buffer $\mathcal{B}$.
5:         for $t = 1, ..., T$: do (gradient step)
6:             Randomly sample a batch of size S from the Replay Buffer $\mathcal{B}$.
7:             Perform one gradient step:
       $$\theta^{(t+1)} = \theta^{(t)} - \gamma^{(t)} \left\{ \sum_{i=1}^{S} \nabla_\theta \tilde{Q}_{\theta^{(t)}}(i^s, u^s) \big( \tilde{Q}_{\theta^{(t)}}(i^s, u^s) - g(i^s, u^s) - \alpha \min_{u \in U(i^s)} \{\tilde{Q}_{\theta'}(i^{s+1}, u)\} \big) \right\}$$
8:         end for
9:     end for
10: end for
Output: The last DNN configuration $\bar{\theta}$. A suboptimal policy:
       $$\tilde{\mu}^{(t+1)}(i) = \arg\min_{u \in U(i)} \left\{\tilde{Q}_{\bar{\theta}}(i,u)\right\}$$
```

References

[1] D. P. Bertsekas, *Reinforcement learning and optimal control.* Athena Scientific, 2019.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.