

## Non-Archimedean Numbers

gli non-archimedean numbers, anche detti non standard, non rispettano la Archimedean property.

### Axiom (Archimedes)

Let  $\mathcal{U}$  be any ordered set. Then:

$$\forall x, y \in \mathcal{U}, |x| < |y| \quad (x, y \Rightarrow \exists n \in \mathbb{N} : |nx| > |y|)$$

### Un tipo di NA-numbers

Per definire i numeri euclideani abbiamo usato un'axiomatization chiamata alpha theory.

Essa è basata sull'euclidean theorem invece che lo hume, ma esistono diversi tipi di axiomatization (non è unico).

### Axiom (Euclides - V)

Let  $A$  and  $B$  be two sets such that  $A \subset B$ . Then  $A$  contains less elements than  $B$ , i.e.,  $n(A) < n(B)$ .

### Axiom (Hume)

Let  $A$  and  $B$  be two sets. If there exists a bijection  $\Phi: A \rightarrow B$ , then  $A$  and  $B$  have the same number of elements, i.e.,  $n(A) = n(B)$ .

è importante notare che l'euclidean e hume theorems non possono coesistere con gli insiemi infiniti, infatti:

Se considero l'insieme dei numeri pari e faccio un'associazione tra  $\mathbb{P} \subset \mathbb{N}$  ottengo una bijection:

$$1 \rightarrow 2, \quad 2 \rightarrow 4, \quad 3 \rightarrow 6 \dots$$

Per il teo di hume dovrei avere  $n(\mathbb{P}) = n(\mathbb{N})$ , ma essendo  $\mathbb{P} \subset \mathbb{N}$  dovrei avere  $n(\mathbb{P}) < n(\mathbb{N})$ , quindi ottengo una contraddizione tra i due teoremi.

### 1° assioma

#### Axiom 1 (Existence)

Exists an ordered field  $\mathbb{E} \supset \mathbb{R}$  whose numbers are called Euclidean numbers (sometimes also indicated with  ${}^*\mathbb{R}$ )

- L'idea è di poter lavorare con essi come se fossero numeri reali.
- Esiste almeno un elemento in  $\mathbb{E}$  che non è in  $\mathbb{R}$ .

### 2° assioma

#### Axiom 2 (Numerosity @)

Exists a function num, num :  $\mathcal{U} \rightarrow \mathbb{E}$  which satisfies

- $\text{num}(A) = |A|$ , if  $A$  is finite
- $\text{num}(A) < \text{num}(B)$  if  $A \subset B$
- $\text{num}(A \cup B) = \text{num}(A) + \text{num}(B) - \text{num}(A \cap B)$
- $\text{num}(A \times B) = \text{num}(A) \cdot \text{num}(B)$
- $A \cap B = \emptyset, \exists \Phi: A \rightarrow B \text{ bijective} \Rightarrow \text{num}(A) = \text{num}(B)$
- $\alpha := \text{num}(\mathbb{N})$

- 
- $\alpha \in \mathbb{E}$ ,  $\alpha$  is infinite
  - $\eta = \alpha^{-1} \in \mathbb{E}$  (existence of inverse)
  - $\eta$  is infinitesimal
  - $\mathbb{E}$  contains algebraic manipulations of  $\alpha$ , e.g.,  $\frac{\alpha^7 - \alpha^{\pi} + \eta^{\alpha}}{-3 + 5\eta^3}$

OSS:

- $\alpha$  è infinito, cioè  $\exists k \in \mathbb{N}$  tale che  $k > |\alpha|$
- $\eta$  è infinitesimale, cioè  $\exists k \in \mathbb{N}$  tale che  $|\eta| > \frac{1}{k}$

### 3° assioma

#### Axiom 3 (Transfer principle - 1/3)

Every sequence  $\varphi$  has a unique  $\alpha$ -limit denoted by  $\lim_{n \uparrow \alpha} \varphi(n)$  which satisfies the following properties:

- if  $\varphi$  is constant:

$$\lim_{n \uparrow \alpha} \varphi(n) = \varphi(1).$$

- if  $\varphi : \mathbb{N} \rightarrow \mathbb{R}$

- if  $\xi \in \mathbb{E}$ , then there exists a sequence  $\varphi : \mathbb{N} \rightarrow \mathbb{R}$  such that

$$\xi = \lim_{n \uparrow \alpha} \varphi(n).$$

- if  $\varphi(n) = n$ , then

$$\lim_{n \uparrow \alpha} \varphi(n) = \alpha.$$

- if eventually  $\varphi(n) \geq \psi(n)$  (namely  $\exists n_0 \in \mathbb{N}$  such that  $\forall n \geq n_0, \varphi(n) \geq \psi(n)$ ), then

$$\lim_{n \uparrow \alpha} \varphi(n) \geq \lim_{n \uparrow \alpha} \psi(n).$$

- for every sequence  $\psi$

$$\lim_{n \uparrow \alpha} \varphi(n) + \lim_{n \uparrow \alpha} \psi(n) = \lim_{n \uparrow \alpha} (\varphi(n) + \psi(n)),$$

$$\lim_{n \uparrow \alpha} \varphi(n) \cdot \lim_{n \uparrow \alpha} \psi(n) = \lim_{n \uparrow \alpha} (\varphi(n) \cdot \psi(n)).$$

- if  $f$  is any function such that  $f \circ \varphi$  is defined, then  $f$  can be uniquely extended at  $\xi = \lim_{n \uparrow \alpha} \varphi(n)$  as

$${}^*f(\xi) = \lim_{n \uparrow \alpha} (f \circ \varphi)(n),$$

where the symbol  ${}^*f$  indicates the extension of  $f$  to  $\mathbb{E}$ .

- if  $\varphi$  is a sequence of nonempty sets:

$$\lim_{n \uparrow \alpha} \varphi(n) = \left\{ \lim_{n \uparrow \alpha} \psi(n) \mid \psi(n) \in \varphi(n) \forall n \right\}.$$

Moreover, the  $\alpha$ -limit preserves the order of  $\varphi$ 's elements, if any.

### Osservazioni

4)

However, not everything transfers perfectly. For example, one key property of real numbers (denoted by  $\mathbb{R}$ ) doesn't get transferred over. This property is called 'completeness,' and it's all about how every nonempty set of real numbers that has an upper bound will also have a 'least upper bound' or 'supremum.' In this broader mathematical system, this isn't always true. An example is the set of all numbers that are 'infinitesimally close' to zero; this set doesn't have a least upper bound.

2)  $E$  non è unico, dipende dall' axiomatizzazione

3)  $\alpha \in$  pari,  $\frac{\alpha}{n} \in {}^*\mathbb{N}$   $\forall n \in \mathbb{N}$ ,  $\sqrt{\alpha} \in {}^*\mathbb{N}$ ,  $\sqrt[n]{\alpha} \in {}^*\mathbb{N}$   $\forall n \in \mathbb{N}$

4) Non è vero che:  $\alpha/2 \in {}^*\mathbb{N}$ ,  $\alpha$  è primo,  $d \leq 0$ ,  $\sin(\alpha)=0$

The Transfer Principle is a mathematical axiom that serves as a bridge connecting standard mathematics to a broader mathematical system known as non-standard analysis. This broader system extends real numbers to include new elements, like infinitesimals and infinite quantities.

Here's a simplified breakdown of the Transfer Principle axioms provided:

- Every sequence has a unique  $\alpha$ -limit. If the sequence is constant, this  $\alpha$ -limit is the same as its first term.
- Sequences that map to real numbers can be transformed so that their  $\alpha$ -limits are any given element from an extended set  $\mathbb{E}$ . In the special case where the sequence equals the natural numbers, its  $\alpha$ -limit is  $\alpha$ .
- The  $\alpha$ -limit maintains the order of values (inequalities). It also works well with operations like addition and multiplication; you can distribute the  $\alpha$ -limit across these operations.
- Functions can be extended to operate on  $\alpha$ -limits. The extension is defined based on the  $\alpha$ -limits of function compositions with sequences. This ensures that you can consistently apply functions to  $\alpha$ -limits.
- For sequences of non-empty sets, the  $\alpha$ -limit of the sequence is a set made up of the  $\alpha$ -limits of its elements. This extends the  $\alpha$ -limit concept to sequences of sets.

Essentially, the Transfer Principle allows us to perform arithmetic and analysis in a broader mathematical setting while preserving as much of standard mathematical behavior as possible. It's crucial for studying and working with extended number systems, like those used in non-standard analysis. However, as hinted in the questions, it may not work perfectly in every case, and there might be non-uniqueness in the extended set  $\mathbb{E}$ .

Regenerate response

## Algorithmic Fields

I computer hanno una capacità finita di rappresentare i numeri, il quale può essere un problema per numeri come  $1/3 = 0.333\ldots$  il quale ha infiniti decimali.

La definizione di algorithmic field è l'insieme di tutti i numeri che possono essere rappresentati esattamente in un computer (può contenere simboli che non sono numeri, come NaN oppure  $\pm \infty$ )

I computer usano la fixed length representation perché permette di avere:

- Operazioni standardizzate, cioè eseguite direttamente dalla CPU
- le operazioni possono essere velocificate (tramite la FPU) e sono deterministiche in termini di time consumption (upper bound dovuto alla rapp. finita)

L'unico problema è che sono soggette a un po' di rumore dovuto all'approssimazione effettuata dalla floating point representation

Per usare gli euclidian numbers (non-euclidean) in una macchina dobbiamo creare l'algorithmic field relativo (finite length representation)

## Algorithmic Numbers

### Definition (Algorithmic Number)

$\xi \in \mathbb{E}$  is an Algorithmic Number  $\stackrel{\text{def}}{\iff} \xi = \sum_{k=0}^{\ell} r_k \alpha^{s_k}$  where  $r_k \in \mathbb{R}$ ,  
 $s_k \in \mathbb{Q}$ ,  $s_k > s_{k+1}$ .

Monosemia =  $n \alpha^p$

### Theorem (AN normal form)

Any AN  $\xi$  can be represented in the following form as  $\xi = \alpha^p P(\eta)$ , where  $p \in \mathbb{Q}$ ,  $m \in \mathbb{N}$  and  $P(x)$  is a polynomial with real coefficients such that  $P(0) = r_0 \neq 0$ .  
es:  $\alpha^0 (10m + 12m^2 + 0m^3)$

$E \in E$  si detto algorithmic number se può essere

→ rappresentato da una somma finita di monosemia

$n \cdot \alpha^{s_1}$  = leading monosemia

standard form per gli algorithmic numbers, la quale

→ semplifica la manipolazione e rappresentazione nei calcoli  
(rappresentazione univoca)

Gli algorithmic numbers sono un sottoinsieme di  $E$  t.c. sono più facili da gestire per i computer, il loro problema è che non sono chiusi rispetto all'inversione (il reciproco di un AN può non essere un AN) ed inoltre possono avere lunghezza variabile (es:  $\xi = \alpha + \beta \cdot \alpha^2 + \gamma \cdot \alpha^3 + \dots$  →  $\phi = \xi \cdot \beta = \alpha^2 + 3\alpha + 2 \rightarrow$  maggior lunghezza)

## Bounded Algorithmic Numbers (BAN)

### Definition (BAN)

$\xi = \alpha^p P(\eta)$ ,  $\xi \in \mathbb{E}$  is a BAN  $\stackrel{\text{def}}{\iff} \xi$  is an AN and  $p \in \mathbb{Z}$

Truncated

BAN è uno speciale tipo di AN lightweight e più facile da gestire.

### Definition (Truncation) of an Algorithmic Number

Let  $P(x) = p_0 x^{z_0} + \dots + p_m x^{z_m}$ ,  $z_{i-1} < z_i$ ,  $i = 1, \dots, m$ . Then

$$\text{tr}_n[P(x)] := \begin{cases} P(x) & n \geq m \\ p_0 x^{z_0} + \dots + p_n x^{z_n} & n < m \end{cases}$$

Considero il polinomio con  $n$  monosemia.  
Se  $n \geq m$  allora lo prendo interamente  
altrimenti lo prendo fino all' $n$ -esimo monosemia

## Lexicographical Multi-obj optimization problems (LHOP)

Se LHOP, abbiamo più obiettivi: quali sono ordinati per priorità ( $f_1 > f_2 > \dots > f_n$ ) il che vuol dire "ottimizza  $f_1$ , successivamente ottimizza  $f_2$  se possibile, poi  $f_3$  e così via".

### 1) Preemptive Approach

Nell'approccio preemptive risolviamo n differenti optimization problems.

Dopo ogni run, aggiungiamo il risultato come constraint.

es: ottimizza  $f_1 \rightarrow$  Helli come constraint il risultato  $\rightarrow$  ottimizza  $f_2 \dots$

Problemi:

- i problemi diventano man mano sempre più complessi
- Posso passare da constraint lineari a non-lineari

### 2) Scalarization

L'idea è di pesare ogni obj. function in accordo con la loro priorità, cioè  $w_1 > w_2 > w_3 \dots$

Problemi:

- Non ho garanzia che il problema orig. e scalarizzato siano equiv. perché la scelta dei pesi è arbitraria ed è un approccio trial-and-error
- Non ho garanzia di convergere a un ottimo globale

$$\begin{aligned} \min_x \quad & w_1 f_1(x) + \dots + w_n f_n(x) \\ \text{s.t.} \quad & x \in \mathcal{D} \end{aligned}$$

### 3) Non-archimedean Scalarization

L'idea è di pesare ogni obj. fun con un valore non-archimedeano sempre più piccolo.

Questo mi assicura la convergenza a un ottimo globale perché non c'è un numero appartenente a  $\mathbb{N}$  che può rendere  $\alpha^0 > \alpha^1$  ed inoltre sono sicuri che i problemi sono equivalenti.

Problema: Necessita di un algoritmo che lavora con queste funzioni non standard

$$\begin{aligned} \min_x \quad & f_1(x)\alpha^0 + \dots + f_n(x)\alpha^{1-n} \\ \text{s.t.} \quad & x \in \mathcal{D} \end{aligned}$$

## NA-C-simplex

l'NA-C-simplex è la versione del simplex che lavora con cost function non-archimedeeane.

Esso mantiene tutte le proprietà del simplex originale.

$$\begin{aligned} \min_x \quad & \sum_{i=1}^n c_i^T x \alpha^{1-i} = \tilde{c}^T \cdot x \quad \text{dove } \tilde{c} \in E \\ \text{s.t.} \quad & Ax = b, \\ & x \geq 0 \end{aligned}$$

Se non ho una base  $B$  disponibile da cui iniziare, devo trovarla, aggiungo una slack variable (una per ogni vincolo) ed ottengo un problem embedding

$$Ax = b \implies Ax + Is = b$$

Questo mi garantisce l'esistenza di una feasible starting base  $B'$

$$(x, s) = (0, b) \text{ is feasible} \implies B' = \{n+1, \dots, n+m\}$$

Per rendere il problema equivalente all'originale posso usare due metodi:

#### 1) Two-Phases method

a) Cerco una soluzione feasible per il problema originale usando l'embedding problem

$$\begin{aligned} \min_{x, s} \quad & 1^T s \rightarrow c' = [\underbrace{0 \dots 0}_{\text{per } x}, \underbrace{1 \dots 1}_{\text{per } s}] \\ \text{s.t.} \quad & Ax + Is = b, \\ & x, s \geq 0 \end{aligned}$$

Se  $\bar{s} = 0$  nella soluzione, abbiamo trovato una soluzione  $\bar{x}$  feasible (perché  $A\bar{x} \leq b$  sicuramente)

#### b) Risolvo il problema originale

- $x = \bar{x}$
- $B := B'[1 : n]$
- Use Simplex algorithm using  $B$  as starting basis

#### 2) Big-M method

Il big-M method cerca direttamente la soluzione ottimale  $\bar{x}$  per il problema originale, convergendo a  $\bar{x}'$  (se esiste) e estendendo  $\bar{x}$  da  $\bar{x}'$ .

$$\begin{aligned} \min_{x'} \quad & d^T x \\ \text{s.t.} \quad & A_{le}x + Is = b_{le}, \Rightarrow s, e, r, p \text{ sono slack variables} \\ & A_{eq}x + le = b_{eq}, \\ & A_{ge}x + lr - lp = b_{ge}, \\ & x, s, e, r, p \geq 0 \end{aligned}$$

Il problema originale ed embedded sono equivalenti se  $e=0$  e  $r=0$ , per fare ciò si usa la penalization usando un big penalization weight ( $M \geq 0$ )

$$\begin{aligned} \min_{x'} \quad & 1^T aM + d^T x \quad \text{Artificial variable } a = \{e, r\} \\ \text{s.t.} \quad & A_{le}x + Is = b_{le}, \Rightarrow \begin{aligned} & \text{Un teorema ci assicura che esiste } M \text{ sufficientemente grande} \\ & \text{t.c. } r, e = 0 \text{ e } \bar{x} \text{ è la soluzione del problema originale} \end{aligned} \\ & A_{eq}x + le = b_{eq}, \\ & A_{ge}x + lr - lp = b_{ge}, \\ & x, s, e, r, p \geq 0 \end{aligned}$$

- $a \neq 0 \Rightarrow \bar{x}$  ( $a \neq 0$  non esiste soluzione ottima)
- $\bar{x}$  coincide con i primi  $n$  componenti di  $\bar{x}'$

### Osservazioni:

#### 1) Two - Phases

- Consuma molte risorse eseguendo due volte il simplex
- garantisce la convergenza all' ottimo, se esiste

#### 2) Big-M

- Richiede meno risorse
- Non garantisce la convergenza perché dipende da come scegliere M
- Non possono essere sicuri che  $\alpha \neq 0$  è dovuto dall' infeasibility del problema o semplicemente dalla scelta di un M troppo piccolo

### Infinitely Big-M method (I-Big-M)

L' obiettivo di M è di trovare  $\alpha = 0$  e poi, se rimane spazio di miglioramento, di minimizzare  $d^T e \rightarrow \text{lexigraph}$ .

L' idea dell' algoritmo è di settare  $M = \alpha$  (infinitely big constant M), risolvendo i problemi del big-M method, perché:

- Problema non più parametrico  $\rightarrow$  Non dipende più dalla scelta di M
- Garantisce la convergenza perché  $\alpha$  è sempre grande abbastanza

#### 1 obj function

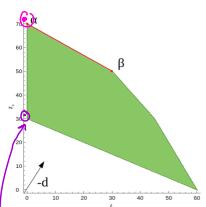


Figure: Standard Kite

Punto che appartiene al poliedro del problema originale

Table: I-Big-M iterations to solve the standard Kite

Iter.	$x_1$	$x_2$	$x_3$	$x'$	$c^T x'$	Basis	$x$
1	[0, 0, 0, 90, 190, 300, 10, 70, 0]				-80 $\alpha$	{4, 5, 6, 7, 8}	[0, 0, 0]
2	[0, 35, 0, 55, 85, 195, 10, 0, 0]				-10 $\alpha$ + 420	{4, 5, 6, 7, 2}	[0, 35, 0]
3	[0, 30, 10, 90, 120, 180, 0, 0, 0]				430	{4, 5, 6, 3, 2}	[0, 30, 10]
4	*	[0, 70, 10, 50, 0, 60, 0, 0, 0, 80]			910	{4, 9, 6, 3, 2}	[0, 70, 10]

entrambi sono fuori che non appartengono al poliedro originale perché non hanno  $e, r, \alpha > 0$  (artificial variable)

$\rightarrow$  2 obj functions

essendo che entrambe le artificial vars. sono uscite il problema non ha più un infine valore perché non c'è più penalizzata da un'infine costante  $M = \alpha$

Oss: I-Big-M trova la soluzione al primo tentativo

$$\begin{aligned} \max_{x'} & - (e + r)\alpha + 8x_1 + 12x_2 + 7x_3 \\ \text{s.t.} & \begin{cases} 2x_1 + x_2 - 3x_3 + s_1 = 90, \\ 2x_1 + 3x_2 - 2x_3 + s_2 = 190, \\ 4x_1 + 3x_2 + 3x_3 + s_3 = 300, \\ x_3 + e = 10, \\ x_1 + 2x_2 + x_3 + r - p = 70, \end{cases} \\ & x, s, e, r, p \geq 0 \end{aligned}$$

#### 2 obj functions

$$\begin{aligned} \max_{x'} & - (e + r)\alpha + (8 + 14\eta)x_1 + (12 + 10\eta)x_2 + (7 + 2\eta)x_3 \\ \text{s.t.} & \begin{cases} 2x_1 + x_2 - 3x_3 + s_1 = 90, \\ 2x_1 + 3x_2 - 2x_3 + s_2 = 190, \\ 4x_1 + 3x_2 + 3x_3 + s_3 = 300, \\ x_3 + e = 10, \\ x_1 + 2x_2 + x_3 + r - p = 70, \end{cases} \\ & x, s, e, r, p \geq 0 \end{aligned}$$

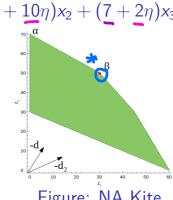


Figure: NA Kite

Table: I-Big-M iterations to solve the NA Kite.

Iter.	$x'$	$c^T x'$	Basis	$x$
1	[0, 0, 0, 90, 190, 300, 10, 70, 0]	-80 $\alpha$	{4, 5, 6, 7, 8}	[0, 0, 0]
2	[0, 35, 0, 55, 85, 195, 10, 0, 0]	-10 $\alpha$ + 420	{4, 5, 6, 7, 2}	[0, 35, 0]
3	[0, 30, 10, 90, 120, 180, 0, 0, 0]	430	{4, 5, 6, 3, 2}	[0, 30, 10]
4	[0, 70, 10, 50, 0, 60, 0, 0, 0, 80]	910	{4, 9, 6, 3, 2}	[0, 70, 10]
5	[30, 50, 10, 0, 0, 0, 0, 0, 0, 70]	910 + 940 $\eta$	{4, 9, 1, 3, 2}	[30, 50, 10]

$\downarrow$  fine misurazione  $f_2 *$

### NA - b - simplex

Simplex algorithm per problemi dove il **b**-vector è composto da numeri non-anclimedeani.

Possibili vantaggi:

- L'algoritmo potrebbe opporre casi in cui la **feasible region** è **unbounded**
- L'algoritmo non deve verificare costantemente la divergenza (-time consuming)
- L'algoritmo potrebbe avere un meccanismo per gestire casi in cui più soluzioni rispettano i constraints
- La soluzione può essere usata come punto iniziale per Branch & cut techniques

### NA - A - simplex

Simplex algorithm per problemi dove l'**A**-matrix è composta da numeri non-anclimedeani.

Theoretically equivalent: 9 constraint che includono NA numbers possono essere, nella teoria, equivalenti ai constraint classici, ma in pratica non è possibile dovuto al horzamento della divisione

Possibili utili:

- Lexicographic zero-sum games
- Degeneracy Mitigation: Meccanismo per gestire casi in cui più soluzioni rispettano i constraints

### NA - simplex

Simplex algorithm per problemi dove il **c**-vector, il **b**-vector, l'**A**-matrix sono composti da numeri NA.

Questo tipo di algoritmo dovrebbe funzionare sicuramente, dovuto:

- Teoricamente, al Transfer Principle
- Praticamente, dovuta alla natura combinatoria del problema

Solo le soluzioni t.c.  $Ax=b$  sarebbero considerate come soluzioni interessanti.

### Numerical Noise

Il rumore può avere conseguenze gravi, ad esempio il cambio del segno e magnitudo

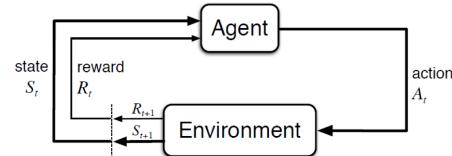
$$3 + 2\eta \rightarrow \underbrace{-1e-6\alpha}_{\text{noise}} + 3 + 2\eta$$

Alcuni modi per mitigare quest'effetto sono:

- |   |  |
|---|--|
| <ul style="list-style-type: none"><li>• Concentrarsi sulla parte più significativa di un calcolo / espressione</li><li>• Denoising: Tecniche per rimuovere il rumore</li><li>• Perturbare infinitesimalmente quando l'algo converge</li></ul> | <ul style="list-style-type: none"><li>• Prevenire che il gradiente diventi troppo grande togliendolo a un valore max ("clipping")</li><li>• Usare algo diversi combinati</li></ul> |
|---|--|

## Reinforcement Learning

Machine learning technique nella quale un agente impara facendo delle azioni su un environment e ricevendo dei reward.



L'obiettivo dell'agente è imparare una strategia (policy) per massimizzare il reward.

### 1) Environment

- **State space (S)**: Un insieme di tutti i possibili stati
- **Action space (A)**: Un insieme di tutte le possibili azioni
- **State transition function (P)**: Dato uno stato e un'azione da la probabilità di finire in ogni possibile stato  
 $\rightarrow P(s' | s, a) = \text{Prob}(s_{t+1} = s' | s_t = s, a_t = a)$
- **Reward function (R)**: Dato uno stato e un'azione ci da il reward ottenuto  
 $\rightarrow R(s, a) = E[R_{t+1} | s_t = s, a_t = a]$  ( $E$  = expected value)
- **Discount factor ( $\gamma$ )**: Valore tra 0 e 1 che descrive l'importanza dei reward successivi:  
 $\rightarrow \approx 0$  = considera solo i reward immediati  
 $\rightarrow \approx 1$  = da molta importanza ai reward futuri

### 2) Agente

- **Policy function ( $\pi$ )**: Funzione che descrive il comportamento di un agente, cioè la probabilità con cui l'agente sceglie una specifica azione in uno specifico stato  
 $\rightarrow$  Data una policy,  $P_{ss'}^{\pi} = \sum_{a \in A} \pi(a|s) P_{ss'}^a = E_a [P_{ss'}^a]$ , cioè la probabilità di finire nello stato  $s'$ , data la policy  $\pi$  e lo stato  $s$ , è data dalla somma sulle azioni della probabilità di fare quell'azione (seguendo la policy) per la probabilità che facendo quell'azione nello stato  $s$ , finiamo in  $s'$ .
- $\rightarrow$  Data una policy,  $R_s^{\pi} = \sum_{a \in A} \pi(a|s) \cdot R_s^a = E_a [R_s^a]$ , cioè il reward nello stato  $s$  seguendo  $\pi$  è dato dalla somma sulle azioni della probabilità di fare quell'azione (seguendo la policy) per il reward ottenuto facendo quell'azione in quel determinato stato.
- **Return ( $G_T$ )** = è la somma dei reward futuri (dal tempo  $t$  in poi) scontati da  $\gamma$ .  
 $\rightarrow G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$

Value function ( $V_\pi$ ): Dato uno stato  $s$ , misura la qualità dello stato cioè l'expected return partendo dallo stato  $s$  e seguendo la policy  $\pi$ .

$$V_\pi(s) = \mathbb{E}_\pi [G_t \mid s_t = s] \rightarrow Ci indica quanto è buono essere in quello stato, non ci dà nessuna info sulle azioni migliori.$$

Action-Value function ( $Q$ ): Dato uno stato e un'azione, misura la qualità dello stato cioè l'expected return partendo da  $s$ , eseguendo l'azione e successivamente seguendo la policy. → Ci dice anche quali sono le azioni migliori da prendere in quello specifico stato.

$$Q_\pi(s, a) = \mathbb{E}_\pi [G_t \mid s_t = s, a_t = a]$$

### Bellman Equation

La Bellman equation permette di calcolare  $V$  e  $Q$  in modo ricorsivo rappresentando il return  $G_t$  ricorsivamente.

$$G_t = r_{t+1} + \gamma G_{t+1} \rightarrow \begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[r_{t+1} + \gamma V_\pi(s_{t+1}) \mid s_t = s] \\ Q_\pi(s, a) &= \mathbb{E}_\pi[r_{t+1} + \gamma Q_\pi(s_{t+1}, a_{t+1}) \mid s_t = s, a_t = a] \end{aligned}$$

### Optimal Agent

Un agente è ottimale quando segue la optimal policy ( $\pi^*$ ), cioè la policy che massimizza l'expected return da ogni stato iniziale.

$$\pi^* = \arg \max_\pi \mathbb{E}_\pi[G_1 \mid s_1 \sim \rho(S)] = \arg \max_\pi \mathbb{E}_\pi[V_\pi(s_1) \mid s_1 \sim \rho(S)]$$

Di conseguenza,  $V$  e  $Q$  ottimali corrispondono alla optimal policy.

$$\begin{aligned} V^*(s) &= \max_\pi V_\pi(s) &= \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V^*(s') \\ Q^*(s, a) &= \max_\pi Q_\pi(s, a) &= R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a \max_{a' \in A} Q^*(s', a') \end{aligned} \quad \text{Bellman optimality equation}$$

### Types of RL problems

a) Policy evaluation: Data  $\pi$  trovare  $V_\pi$  → Indica quanto è buono seguire la policy  $\pi$

↳ Se state space ragionevole, posso usare la closed form derivabile dalla bellman equation, altrimenti iterative.

a) L'agente interagisce con l'env seguendo  $\pi$

b) Mantiene traccia dei reward ottenuti nel tempo

c) Usa i reward per stimare  $V_\pi$  e lo usa come misura della qualità di  $\pi$

$V_\pi = (I - \gamma P^\pi)^{-1} R^\pi$   
matrice composta da  $P_{ss'}^\pi$  vettore composto da  $R_s^\pi$

} se considero le possibilità di visitare ogni stato infinite volte, converge al vero  $V_\pi$

2) Policy optimization: Trovare  $V^*$  e  $\pi^*$

- Scegliere  $\pi$  arbitraria
- Valutare  $\pi$  ottenendo  $V_\pi$  (policy evaluation)
- Migliorare la policy  $\rightarrow$  Modifico la policy per scegliere la migliore in base a  $V_\pi$  (in ogni stato) generando una nuova policy (approccio greedy)
- Ripeto b e c fino a convergenza  $\pi^*$   $\rightarrow$  Continuo finché non ho cambiamenti nella policy

Osservazioni:

- Ogni iterazione (b e c) è chiamata Generalized Policy Iteration (GPI)
- La convergenza è garantita se env stationario e posso visitare ogni stato infinite volte

## Dynamic programming

È possibile utilizzare il dynamic programming nell'implementazione della policy improvement se abbiamo una full model knowledge del modello dell'environment (stati, azioni ...)

### 1) Policy evaluation

```

Input: S, A, R, P, π, ε
V₀(s) = 0 ∀s ∈ S → V inizializzata a 0 stato
k = 0
while k == 0 or ||Vₖ - Vₖ₋₁|| > ε do
    Vₖ₊₁(s) = ∑ₘᵢₙₐ π(a|s) (Rₛᵃ + γ ∑ₛ' ∈ S Pₛₛ' Vₖ(s'))
    k = k + 1
end while
return Vₖ
  
```

### 2) Policy improvement

Policy improvement (ties are broken arbitrarily):

$$\pi(s) \leftarrow \left[ \arg \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_k(s') \right] \text{ Azione che massimizza } V \text{ nello stato } s$$

Osservazioni:

- La policy convergence avviene sempre prima della value convergence
- La policy migliorata greedy è sempre meglio della precedente ]!!
- È possibile velocizzare la convergenza considerando l'expected return su tutte le azioni invece che sulla distib. data dalla policy.  $V_{k+1}(s) = \max_{a \in A} R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a V_k(s)$  \*
- La scelta greedy non è sempre la migliore perché può missare path peggiori all'inizio ma migliora successivamente

## Model-free approaches

Sono algoritmi che non necessitano di conoscere il modello dell'ambiente (probabilità transizione tra gli stati).

**Episodio:** È una lista di esperienze (stato, azione, reward, stato-successivo) fatte nel percorso tra initial state e terminal state (oppure posso dare un numero max di esperienze T)

**Esperienza:** È una singola interazione con l'environment ed è composta da (stato, azione, reward, stato-successivo)

## Monte Carlo RL

Monte Carlo approach aspetta la fine di un episodio per fare il learning (backpropagation)

Esso può essere implementato in due modi:

1) **First-encounter:** Se uno stato viene visitato più volte durante un episodio, solo la prima volta viene considerata

2) **Every encounter:** Viceversa, considera ogni visita

Noi abbiamo visto il caso dell'every encounter.

L'idea è di usare l'incremental mean, in modo da non salvare  $G(s)$  e per aggiornare  $V(s)$

dopo ogni episodio.

$t = T \dots 0 \rightarrow \text{Backpropagation}$

Ad ogni timestamp  $t$  dell'episodio posso aggiornare  $V(s_t)$  dello stato  $s$  a tempo  $t$  con la seguente formula

$$V(s_t) = V(s_t) + \frac{1}{N(s_t)} (G_t - V(s_t))$$

expected return real, non stimato  
n° encounters stato  $s_t$

OSS: Essendo che  $N(s_t) \rightarrow \infty$ , il valore stimato  $V(s)$  convergerà al vero  $V_T(s)$

Nel caso di environment non-stazionario la formula di update diventa:

$$V(s_t) = V(s_t) + \nu(G_t - V(s_t)), \quad \nu \in (0, 1)$$

- Converge alla minima squared error solution (tra observed returns e estimated value fun)
- Non rispetta la Markovian property perché il learning è basato sull'intero episodio
- Offline learning: Aspetta la fine dell'episodio per aggiornare value function e policy
- Unbiased: Stimiamo l'expected return facendo una media dei returns negli episodi
- High variance: Aspettiamo la fine degli episodi, il return può variare molto

## Temporal Difference RL

Yl TD approach dopo ogni esperienza fa il learning (Incremental learning)

1) Ynitializza  $V(s) \approx V_{\text{set}}$

2) Ad ogni esperienza (step) aggiorna  $V(s_t)$

$$V(S_t) \leftarrow V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

↓      ↓  
learning rate      To target → expected return (not real nn approx)

Oss: Yl  $G_t$  usato non è l'expected return effettivo ma è una stima (chiamato TD target)

- TD converge alla maximum likelihood solution (se  $0 < \alpha \ll 1$ )
- Rispetta la Markovian property perché il valore dello stato viene aggiornato sulla base dello stato corrente e successivo soltanto
- Online learning
- Biased perché le early experiences impattano maggiormente il learning, è possibile ridurre il bias usando il batch updating, cioè collezionando un tot. di esperienze (batch-size) e successivamente imparando da esperienze estratte a caso dal batch.
- Low variance perché aggiorna frequentemente

La versione con il singolo step è chiamata TD(0), ma esistono altre versioni:

1) TD(n): Usa n step avanti per l'update invece che il singolo step

$$V(s_t) = V(s_t) + \nu(G_t^{(n)} - V(s_t)), \quad G_t^{(n)} = \sum_{k=0}^n \gamma^k r_{t+k+1} + \gamma^{n+1} V(s_{t+n+1})$$

2) TD( $\lambda$ ): Generalizza TD(n) usando  $\lambda$  per fare una media pesata dei  $G_t^{(n)}$

$$V(s_t) = V(s_t) + \nu(G_t^{(\lambda)} - V(s_t)), \quad G_t^{(\lambda)} = (1 - \lambda) \sum_{n=1}^N \lambda^{n-1} G_t^{(n)}$$

Yl parametro  $\lambda$  viene usato per controllare il tradeoff tra bias e varianza.

- $\lambda$  vicino allo 0 → L'agente si concentra sui ritorni a breve termine (TD(0))
- $\lambda$  vicino all'1 → L'agente si concentra sui ritorni a lungo termine (Monte Carlo)

Entrambe queste due versioni sono approcci offline, imparano da episodi completi.

Voglio trovare un modo per implementare TD( $\lambda$ ) su episodi parziali (e quindi online)

Eligibility Trace: Misura quanto nel futuro uno stato è importante

→ Ynizialmente è a 0 per tutti gli stati, quando uno stato viene incontrato la sua eligibility trace aumenta e, con il passare del tempo, l'eligibility trace di tutti

gli stati diminuisce esponenzialmente per un fattore di  $\gamma \cdot \lambda$

$$E_0(s) = 0, \quad E_t(s) = \underbrace{\gamma \lambda E_{t-1}(s)}_{\text{Time Decay}} + \underbrace{\mathbb{I}(s_t; s)}_{\text{Aumento se lo stato } (e' lo stato} \\ \text{viene incontrato a tempo } t)$$

3) Backward TD (BTD(1)) : è una versione di TD(1) che usa le eligibility traces

→ Idea: Aggiornare tutti gli stati visibili in un episodio, non solo il più recente

- a) Viene mantenuta una eligibility trace  $\forall$  stato
- b) Quando un reward è ricevuto, distribuisce immediatamente il reward a tutti gli stati sulla base delle loro eligibility traces.

Questo permette di imparare online da episodi parziali

$$V(S) = V(S) + \nu \delta_t E_t(S), \quad \delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$$

On-Policy optimization → Dopo aver determinato  $V$  o  $Q$  posso fare la policy optimization (improvement)

On-Policy learning: impara la policy  $\pi$  dalle esperienze della policy  $\pi$  stessa

### 1) Greedy Policy Improvement (GPI)

L'idea del GPI è di valutare la policy  $\pi$  e migliorarla greedy

$$\pi_{k+1}(a|s) = \arg \max_{a \in A} R_s^a + P_{ss'}^a V_{\pi_k}(s')$$

Problemi:

- Usare  $V$  quando non conosciamo l'env model non è possibile → conoscerne  $R_s^a$  e  $P_{ss'}^a$ , che sono sconosciuti nel caso model-free
- Usare soltanto l'appuccio greedy non ci permette di:  
esplorare ma solo exploitation →  $\epsilon$ -greedy policy

Policy improvement con  $V$  richiede la  
conoscenza di  $R_s^a$  e  $P_{ss'}^a$ , che sono  
sconosciuti nel caso model-free  
uso  $Q$ -function

$$\pi_{k+1}(a|s) = \arg \max_{a \in A} Q_{\pi_k}(s, a)$$

→  $\epsilon$  prob. di esplorare (azione random),  $1-\epsilon$  di fare la migliore azione (greedy)

→  $\epsilon$ -greedy update garantisce che la policy generata  $\pi_{k+1}$  è migliore della precedente e se  $\epsilon$ -greedy rispetta anche la GLG property ha la convergenza, ed essa viene rispettata

sse:

1) Ogni coppia stato-azione viene esplorata un infinito numero di volte

2) La policy converge alla greedy policy (deterministic)

3)  $\epsilon$  converge a 0 con  $1/k$  rate ( $\epsilon$  decay in time) → Passo da exploration a exploitation

man mano

Tutti gli algoritmi visti finora hanno la **esience property** e posso applicare  **$\epsilon$ -greedy**

- 2) Monte Carlo  $\rightarrow Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \frac{1}{N(s_t, a_t)}(G_t - Q(s_t, a_t))$ , policy improvement con  **$\epsilon$ -greedy**
- 3) Sarsa (TD family, aggiorna  $Q$ )  $\rightarrow$  esperienza  $\langle s, a, r, s', a' \rangle$ 
  - update rule:  $Q(s, a) \leftarrow Q(s, a) + \nu(r + \gamma Q(s', a') - Q(s, a))$
  - Expected Sarsa  $Q(s, a) \leftarrow Q(s, a) + \nu \left( r + \gamma \sum_{a' \in A} \pi(a'|s') Q(s', a') - Q(s, a) \right)$
  - Sarsa ( $n$ )
    - $Q(s, a) \leftarrow Q(s, a) + \nu(r + \gamma Q_t^{(n)} - Q(s, a))$
    - $Q_t^{(n)} = \sum_{k=0}^n \gamma^k r_{t+k+1} + \gamma^{n+1} Q(s_{t+n+1}, a_{t+n+1})$
  - Sarsa ( $\lambda$ )
    - $Q(s, a) \leftarrow Q(s, a) + \nu(r + \gamma Q_t^{(\lambda)} - Q(s, a))$
    - $Q_t^{(\lambda)} = (1-\lambda) \sum_{n=1}^N \lambda^{n-1} Q_t^{(n)}, \quad \sum_{n=1}^{\infty} (1-\lambda) \lambda^{n-1} = 1$
  - Backward SARSA ( $\lambda$ )
    - $Q(S, A) \leftarrow Q(S, A) + \nu \delta_t E_t(S, A)$
    - $\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$
    - $E_t(s, a) = \gamma \lambda E_{t-1}(s, a) + \mathbb{I}(s_t, a_t; s, a), E_0(s, a) = 0$

### Off-policy optimization

Off-Policy learning: imparo la policy  $\pi$  tramite le esperienze fatte dalla policy  $\mu$

L'off-policy è importante perché permette di:

- Usare una policy più adatta per l'esplorazione e comunque ottimizzare la nostra policy target
- Imparare più policy contemporaneamente usando solo una policy per le azioni
- Poter usare vecchie esperienze generate da vecchie policies

L'ingrediente che permette tutto questo è l'**Importance Sampling**

$$\mathbb{E}_{x \sim P} [f(x)] = \sum_x P(x) \cdot f(x) = \sum_x Q(x) \cdot \frac{P(x)}{Q(x)} \cdot f(x) = \mathbb{E}_{x \sim Q} \left[ \frac{P(x)}{Q(x)} \cdot f(x) \right]$$

$P = \text{target policy}$        $Q = \text{behaviour policy}$

L'importance sampling  $\frac{P(x)}{Q(x)}$  ci permette di pesare l'esperienza fatta da  $Q$  per imparare correttamente  $P$

Let's consider a simple example:

Suppose we have a robot that is learning to navigate a maze. Initially, our robot follows a random behavior policy ' $Q$ ' - it moves randomly through the maze. However, we want to learn about a different, target policy ' $P$ ' - a more deliberate strategy that tries to move towards the exit of the maze.

While the robot is moving randomly (following policy ' $Q$ '), it occasionally stumbles on the correct path towards the exit. These experiences are particularly valuable for learning about the target policy ' $P$ '.

However, these valuable experiences are rare when following the random behavior policy ' $Q$ '. To correctly learn about ' $P$ ' using the experiences generated by ' $Q$ ', we need to adjust or reweight the experiences. That's where importance sampling comes in.

## 1) MC optimization off-policy

Abbiamo due returns,  $G_t^\mu$  e  $G_t^\pi$ , dove:

- $G_t^\mu$  = Return ottenuto comportandosi secondo la policy  $\mu$

- $G_t^\pi$  = Return dedotto tramite  $\pi$  dalle esperienze fatte da  $\mu$

L'update rule è  $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \nu(G_t^\pi - Q(s_t, a_t))$

$$\prod_{i=t}^T \frac{\pi(a_i | s_i)}{\mu(a_i | s_i)} G_i^\mu$$

importance ratio  
più di 1 connessione!

Problema: Varianza alta dovuta agli episodi e aumentata se importance sampling alto (dovuto a un comportamento molto diverso tra  $\pi$  e  $\mu$ )

## 2) TD optimization

$$Q(s, a) \leftarrow Q(s, a) + \nu \left[ \frac{\pi(a|s)}{\mu(a|s)} (r_{t+1} + \gamma Q(s', a')) - Q(s, a) \right]$$

Solo una sampling correction  $\rightarrow$  Riduce varianza

## 3) Q-learning

è un tipo di TD algorithm il quale cerca di ridurre ulteriormente il rumore facendo comportare in modo simile la policy, facendo una scelta particolare:

- $\pi(s) = \arg \max_{a \in A} Q(s, a)$  (greedy policy)

- $\mu(s) = \varepsilon\text{-greedy}(Q(s, a))$

L'update rule è la seguente:

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

New Q-value estimation      Former Q-value estimation      Learning Rate      Immediate Reward      Discounted Estimate optimal Q-value of next state      Former Q-value estimation  
 TD Target      TD Error

scelta nella policy target

Essa non necessita di importance sampling perché nell'update rule non considera l'azione fatta dalla behaviour policy.

Con la max operation facciamo implicitamente la connessione, cercando di ottimizzare direttamente la policy target senza preoccuparci delle azioni scelte da  $\mu$ .

Oss: Il Q-learning è un tabular method, cioè la Q-function è rappresentata da una Q-table nella quale ogni cella corrisponde al valore di una coppia stato-azione

	→	→	↑	↓
↑	0	0	0	0
↓	0	0	0	0
→	0	0	0	0
←	0	0	0	0
↑	0	0	0	0
↓	0	0	0	0

## Value function Approximations

I metodi visti sin ora lavorano bene quando lo spazio delle azioni e degli stati è discreto e finito (tabular methods).

Quando devo lavorare con spazi continui (infinte azioni/stati) devo non più usare delle tabular approximations di  $V$  e  $Q$ , ma delle funzioni parametriche:

- $\hat{V}: S \times \Theta \rightarrow \mathbb{R}$  ;  $\Theta$ : parameters space,  $\theta \in \Theta$ : scelta particolare di parametri
- $\hat{Q}: S \times \Theta \rightarrow \mathbb{R}^{|A|}$

La parametrizzazione mi permette di generalizzare da stati visti a non visti ed inoltre posso aggiustarla in modo da approssimare al meglio la vera value function.

Con i value function approximations abbiamo due problemi principali:

### 1) Non-stationary Data

La data distribution cambia con il cambiamento della policy (le esperienze fatte dall'agente sono basate sulla sua policy, se per esempio decide di muoversi sempre a sx la distribuzione nelle esperienze sarà biased sugli stati/azioni scelti dall'agente muovendosi sempre a sx), inoltre anche l'environment può cambiare nel tempo.

Molti approximations necessitano che i dati siano stationari

Possibile soluzione: State space encoding, cioè rappresentare gli stati in un modo che il learning system possa generalizzare da uno stato ad altri stati simili

### 2) Non-Independent and Identically distributed data (Non-IID data)

Le esperienze raccolte dall'agente sono solitamente non indipendenti e non sono ugualmente distribuite, questo perché stati consecutivi sono solitamente simili e l'agente tende a visitare alcuni stati più spesso rispetto ad altri.

Spesso gli approximations richiedono che i dati siano indipendenti e uniformemente distribuiti.

Possibile soluzione: Collezionare diverse esperienze in un dataset e poi imparare da esperienze estratte a caso da questo dataset (**Experience replay Buffer**)

Oss: Per risolvere questi due problemi perdiamo la convergenza

Ci sono due modi per aggiornare i parametri: a) Incremental methods

b) Batch methods

### Incremental methods

Sono metodi che aggiornano iterativamente i parametri della function approximation, man mano che nuove esperienze / episodi vengono fatte dall'agente.

### Stochastic Gradient Descent (SGD)

Aggiorna i parametri della function approximation con lo scopo di minimizzare la loss function.

- **Loss function**  $L(\theta) = \mathbb{E}[(V_\pi(s) - \hat{V}(s, \theta))^2]$   
true value fun      Approx value fun
- **Update dopo ogni esperienza**  $\Delta\theta = \nu(V_\pi(s) - \hat{V}(s, \theta)) \nabla_\theta \hat{V}(s, \theta)$   
gradient value function

Problema: Non conosciamo il true value function ( $V_\pi$ ) → Soluzione: Usiamo il return  $G_t$

### Monte Carlo Method

Nel caso di Monte Carlo method abbiamo che  $G_t$  è unbiased perché è basato sull'intero episodio, ma noisy.

Garantisce la convergenza sia nel caso di approx. function lineare, sia approx. function non lineare

### TD Methods

a) Nel caso del TD(0) abbiamo che  $G_t$  è stimata, quindi biased, ma meno noisy.

Converge all'ottimo solo nel caso di approx. function lineare

b) TD(+) usa  $G_t^+$  ma ha le stesse problematiche di TD(0)

c) Gradient TD

Il gradient TD cerca di risolvere il problema della convergenza quando l'approx. function non è lineare.

Esso cerca di minimizzare il projected Bellman error, cioè la differenza tra la value function stimata e la miglior stima possibile (nel passo successivo) nello spazio delle funzioni rappresentabili dall'approxim. function.

Table: Algorithms convergence in evaluation problems

	Algorithm	Linear	Nonlinear
On-Policy	MC	✓	✓
	TD	✓	✗
	GTD	✓	✓
Off-Policy	MC	✓	✓
	TD	✗	✗
	GTD	✓	✓

Table: Algorithms convergence in optimization problems

	Algorithm	Linear	Nonlinear
On-Policy	MC	✓	✗
	SARSA	✓	✗
	GTD Q-Learning	✓	✗
Off-Policy	MC	✓	✗
	Q-Learning	✗	✗
	GTD Q-Learning	✓	✗

## Batch Methods

Gradient calculated on more samples

Sono metodi che usano batch di episodi per aggiornare le approx. functions.

Essi permettono di generare gradienti più stabili rispetto agli iterative methods, ad esempio SGD.

## Least squared Error

Una linear approx function può essere ottimizzata dal Mean squared error in closed form

$$\theta^* = (\phi(S)^T \phi(S))^{-1} \phi(S)^T G$$

dove:

- $\phi(s)$  = matrice di visited state encoding

- $G$  = vettore dei reward corrispondenti agli stati

## Deep Q-Network

Il concetto è di usare una Deep Neural Network per approssimare la Q-function.

- Utilizza l'experience replay buffer
- Utilizza ε-greedy policy
- I parametri sono aggiornati tramite SGD
- Utilizza due reti, una primary network trainata e aggiornata usata per stimare il Q-value corrente  $Q(s, a, \theta)$ , e una target network aggiornata periodicamente dalla primary network e usata per stimare il massimo Q-value sulle prossime (state-action) pairs.  
→ Questa tecnica ci permette di stabilizzare il learning process riducendo il bias

Simple DQN loss function:  $L(\theta) = \mathbb{E}_D[(r + \gamma \max_{a' \in A} \hat{Q}(s', a', \theta^-) - \hat{Q}(s, a, \theta))^2]$

→ L'azione con il Q-value massimo viene scelta e valutata dalla target network

Double DQN loss function:  $L(\theta) = \mathbb{E}_D[(r + \gamma \hat{Q}(s', \arg \max_{a' \in A} \hat{Q}(s', a', \theta), \theta^-) - \hat{Q}(s, a, \theta))^2]$

→ L'azione viene selezionata dalla primary network e valutata dalla target network, questo ci permette di evitare il maximization bias (l'operazione di massimo performata su stime numerose tende a sovrastimare i Q-values) dovuto dall'usare lo stesso max Q-value per scegliere l'azione e stimare i Q-values.

Dueling Networks: È un'estensione delle DQN nella quale la Q-value function è divisa in due funzioni separate

- Value function ( $V, E$ ) → Expected reward per essere in un particolare stato

- Advantage function  $A(s, a, \theta) \rightarrow$  stima il vantaggio di scegliere quella particolare azione in quello stato

l'idea è che il modello può imparare quali stati sono migliori e quali azioni sono vantaggiose, indipendentemente, e poi combinarle nel Q-value.

### Policy gradient optimization

Finora abbiamo visto dei metodi che cercavano di stimare la value function ottimale per ricondursi alla policy ottimale.

Con i policy-based methods vogliamo ottimizzare la policy senza imparare prima la value function. I policy gradient methods sono un subset dei policy-based methods.

Data una policy approximation function  $\hat{\pi}(a|s, \theta)$  vogliamo trovare i parametri che massimizzano le performance (miglior cumulative return)

Siamo interessati a policy optimization techniques basate sui gradienti: SGD, conjugate GD e Quasi-Newton, i quali usano il gradiente per guidare la ricerca nel policy space.

- Update rule:  $\theta_{k+1} \leftarrow \theta_k + \nu \nabla_\theta L(\theta)$

- Policy gradient theorem: Ci fornisce una direzione per ottimizzare i parametri ( $\theta$ ) in modo da massimizzare l'expected return.

Nello specifico, ci dice che il gradiente dell'expected return rispetto alla policy ( $\theta$ ) è proporzionale all'expectation (calcolata rispetto alla distribuzione stationaria della policy) del prodotto tra la action-value function  $Q$  e il gradiente della log-probability dell'azione selezionata secondo la policy.

$$\nabla_\theta L(\theta) \propto \mathbb{E}_{\hat{\pi}}[Q_{\hat{\pi}}(S, A) \nabla_\theta \ln \hat{\pi}(A|S, \theta) | d_{\hat{\pi}}(S, A)]$$

- Softmax Policy: Quando lo spazio delle azioni è discreto è possibile usare la softmax policy, la quale seleziona le azioni con prob. proporzionale all'esponentiale del loro valore  $Q$ . Questo fa sì che le azioni con  $Q$  più alta vengono scelte più spesso, ma c'è sia la possibilità di esplorare.

- Gaussian policy**: Quando lo spazio delle azioni è continuo è possibile usare la gaussian policy, la quale modella la policy con una distribuzione gaussiana.

### MC policy gradient

- 1) **On-policy**: Il gradiente dell'expected return è proporzionale alla somma, su tutti gli episodi e passi nel tempo, del prodotto tra il gradiente del log della policy e del ritorno (somma dei premi futuri) da quel passo nel tempo.

$$\nabla_{\theta} L(\theta) \propto \underbrace{\frac{1}{N} \sum_{i=1}^N \left( \sum_{t=1}^T \nabla_{\theta} \ln \hat{\pi}(s_t^i, a_t^i, \theta) \right)}_{\text{Maximum Likelihood Gradient}} \underbrace{\left( \sum_{t=1}^T V_{\hat{\pi}}(s_t^i) \right)}_{\text{Policy Gradient}}$$

- 2) **Off-policy**: In questo caso le esperienze sono state raccolte da una policy diversa a quella da migliorare.

In questo caso, il gradiente è proporzionale alla somma, su tutti gli episodi e passi di tempo, del prodotto tra:

- Il gradiente del log della policy per l'importance sampling
- Ritorno da quel passo nel tempo

$$\nabla_{\theta} L(\theta) \propto \frac{1}{N} \sum_{i=1}^N \left( \sum_{t=1}^T \nabla_{\theta} \ln \hat{\pi}(s_t^i, a_t^i, \theta) \prod_{\tau=1}^T \frac{\hat{\pi}(s_{\tau}^i, a_{\tau}^i, \theta)}{\hat{\pi}(s_{\tau}^i, a_{\tau}^i, \theta^-)} \right) \left( \sum_{t=1}^T V_{\hat{\pi}}(s_t^i) \right)$$

Per ridurre la varianza del MC policy gradient si può usare il metodo Action-Critic il quale combina l'apprendimento della policy  $\hat{\pi}$  (action) con l'apprendimento del valore delle azioni (critic).

- **Action  $\hat{\pi}$** : Genera l'esperienza e le direzioni di aggiornamento per i parametri della policy
  - **Critic  $\hat{Q}$** : Valuta la qualità delle azioni prese dall'action (calcola  $Q(s, a)$ )  $\rightarrow$  Feedback all'action per migliorare la policy
- L'idea è di approssimare il valore  $Q_{\hat{\pi}}(s, a)$  con una approx. function  $\hat{Q}(s, a, w)$  dove  $w$  sono i parametri e, nel caso in cui l'approx function  $\hat{Q}(s, a, w)$  sia una DNN ci sono due modi per gestire  $w$  e  $\theta$ .
- $w + \theta$ : Si usano due DNN separate per  $\theta$  e  $w \rightarrow$  semplice, stabile, ma non c'è condivisione delle caratteristiche apprese
  - $w = \theta$ : Si usa una DNN comune, condividendo le caratteristiche apprese  $\rightarrow$  possibili conflitti nel calcolo del gradiente

**Compatible function approx theorem**: Nell'approccio action-critic se

- $\nabla_w \hat{Q}(s, a, w) = \nabla_{\theta} \ln \hat{\pi}(a|s, \theta)$   $\rightarrow$  Cambiare i parametri  $w$  punta allo stesso

- $\mathbb{E}_{\hat{\pi}}[(Q_{\hat{\pi}} - \hat{Q})^2] \xrightarrow{t \rightarrow \infty} 0 \rightarrow$  Errore quadratico medio tra  $Q_{\hat{\pi}}$  e  $Q$  va a 0 con

\ unbiased property
il crescere delle esperienze  
allora il policy gradient non è ancora valido, cioè  

$$\nabla_{\theta} L(\theta) \propto \mathbb{E}_{\hat{\pi}}[\hat{Q}(S, A, \omega) \nabla_{\theta} \ln \hat{\pi}(A|S, \theta) | d_{\hat{\pi}}(S, A)].$$

Per ridurre ulteriormente la varianza è possibile usare una baseline  $B(\cdot)$  cioè una conoscenza empirica del modello, la quale, se dipende solo dallo stato corrente  $s$ , non altera il gradiente.

Soltanente una buona scelta per la baseline è la value function in  $s$  della policy  $B(s) = V_{\hat{\pi}}(s)$  e viene usata nel seguente modo:

$$\nabla_{\theta} L(\theta) \propto \mathbb{E}_{\hat{\pi}} \left[ \sum_{a \in A} (Q_{\hat{\pi}} - B(\cdot)) \nabla_{\theta} \hat{\pi} | d_{\hat{\pi}}(S) \right]$$

L'uso di questa baseline ci permette di definire la advantage function  $A$ , la quale indica quanto sia migliore (o peggiore) una specifica azione rispetto a un'azione qualunque (media) nello stato, cioè misura il valore aggiunto dell'azione  $a$  nello stato  $s$   $\rightarrow A_{\hat{\pi}}(s, a) = Q_{\hat{\pi}}(s, a) - V_{\hat{\pi}}(s)$

Possiamo quindi definire l'advantage Policy gradient

$$\nabla_{\theta} L(\theta) \propto \mathbb{E}_{\hat{\pi}}[A_{\hat{\pi}}(S, A, \omega) \nabla_{\theta} \ln \hat{\pi}(A|S, \theta) | d_{\hat{\pi}}(S, A)]$$

Nello scenario action-critic, l'uso della advantage function comporterebbe il fatto di salvare i parametri:  $\Theta$  per la policy,  $W$  per la  $\hat{Q}$  e  $P$  per la  $\hat{V}$ .

Però, in realtà, è possibile salvare solo i parametri  $P$  (di  $\hat{V}$ ) e derivare  $\hat{Q}$  oppure esprimere la advantage function in funzione di  $V$ .

- $Q_{\hat{\pi}}(s, a) = \mathbb{E}_{s' \in S}[r + \gamma V_{\hat{\pi}}(s') | s, a]$
- $A_{\hat{\pi}}(s, a) = \mathbb{E}_{s' \in S}[r + \gamma V_{\hat{\pi}}(s') - V_{\hat{\pi}}(s) | s, a]$

### Natural Policy Gradient

I policy gradient possono avere difficoltà in policy space discontinui o con altipiani, portando a una convergenza lenta e elevata non-stationarity.

Per risolvere questi problemi, è possibile cambiare i parametri  $\Theta$  proporzionalmente alla policy variation misurata tramite la KL-divergenza tra  $\hat{\pi}(\theta_t)$  e  $\hat{\pi}(\theta_{t+1})$  la quale misura la perdita di informazione.

Il gradiente può essere riformulato tenendo in considerazione che la KL-divergenza tra la policy a tempo  $t$  e la policy a tempo  $t+1$  sia uguale a  $\varepsilon$  (valore piccolo).

$$\nabla_{\theta} L(\theta_t) \leftarrow \text{optimal } \nabla_{\theta} L(\theta_t) \text{ s.t. } KL(\hat{\pi}(\theta_t), \hat{\pi}(\theta_{t+1})) = \varepsilon$$

Facendo così, ogni direzione che altera di molto la policy  $\hat{\pi}_t$  è penalizzata.

È possibile quindi riscrivere la update rule dei parametri, usando la taylor approx di secondo ordine della KL-divergence, come:

$$\Delta\theta \propto G_\theta^{-1} \nabla_\theta L(\theta)$$

dove  $G$  è la Fisher information Matrix la quale misura quanto i parametri  $\theta$  influenzano la politica  $\pi$ , e può essere calcolata già inavallato

Questo approccio per aggiornare i parametri non richiede degli hyperparametri ed inoltre è possibile usare altre metriche diverse dalla KL-divergence.

OSS:

- NPG altera localmente lo spazio di ricerca dei parametri
- In NPG la direzione ottimale del gradiente cambia in base alla posizione nello spazio dei parametri e i gradienti possibili sono rappresentati dai contorni delle ellissi.
- NPG non dà uguale importanza ai params ma prende in considerazione la loro correlazione.
- KL-divergence può essere difficile da calcolare quando lo spazio delle azioni è stato molto grande.  
In questo caso è possibile usare l'approccio Trust Region Policy optimization (PO) il quale approssima la KL-divergence in modo che dipenda solo dallo stato e non dall'azione.

$$KL(\hat{\pi}(\theta_t) || \hat{\pi}(\theta_{t+1})) \approx \mathbb{E}_{s \sim \mathcal{D}}[KL(\hat{\pi}(s, \theta_t) || \hat{\pi}(s, \theta_{t+1}))]$$

Nell'action-critic con una approx function compatibile, otteniamo una notevole semplificazione del gradiente, infatti otteniamo che il gradiente della advantage function  $A$  (rispetto ai params  $p$ , quelli di  $\hat{V}$ ) diventa uguale alla Fisher information Matrix moltiplicato  $\rho$

$$\begin{aligned} \nabla_p \hat{A}(s, a, p) &= \nabla_\theta \ln \hat{\pi}(a|s, \theta) \Rightarrow \nabla_\theta L(\theta) = \mathbb{E}_{\hat{\pi}} [\nabla_\theta \ln \hat{\pi}(\theta) \hat{A}(\rho)] = \\ &= \mathbb{E}_{\hat{\pi}} [\nabla_\theta \ln \hat{\pi}(\theta) \nabla_\theta \ln \hat{\pi}(\theta)^T \rho] = G_\theta \rho \end{aligned}$$

E di conseguenza l'aggiornamento dei parametri della policy  $\theta$  diventa proporzionale a  $\rho$ .

$$\Delta\theta = G_\theta^{-1} \nabla_\theta L(\theta) = G_\theta^{-1} G_\theta \rho = \rho$$

### Deep Policy Networks

Le Deep Policy Network sono function approx. (DNN) usate nell'ambito della policy optimization, cioè  $\pi^*$  è approssimata da una DNN.

È l'equivalente delle DAN, usate per approssimare la Q-value function, nell'ambito dell'ottimizzazione delle policy.

In questo caso, la baseline rappresenta l'expected total discounted reward  $\rightarrow B(\theta) = \mathbb{E}_{s \sim \mathcal{D}} \left[ \sum_{t=0}^T \gamma^t R_t \mid \theta \right]$

soltanente viene usato l' SGD per aggiornare i parametri.

Quantità che l'agente vuole massimizzare



Nel caso dell' action-critic architecture usata nelle DPN, abbiamo:

- Action: è una policy network che sceglie le azioni basandosi sullo stato attuale  
→ ye gradient della action loss (usato per aggiornare i parametri dell'action) è dato dalla seguente formula  $\nabla_{\theta} L^{\pi}(\theta) = \nabla_{\theta} \ln \hat{\pi}(a_t | s_t, \theta) (\hat{Q}(s_t, a_t, \omega) - B(\theta))$  la quale ci dice che l'action aggiorna la sua policy nella direzione che massimizza lo expected discount reward.
- Critic: è una value network che stima l'expected reward per le azioni fatte dall'action.  
→ ye critic fornisce la loss function usata nell'action, cioè la differenza tra il Q-value stimato dal critic e la baseline.  $L^Q(\omega) = (\hat{Q}(s_t, a_t, \omega) - B(\theta))^2$

Nel caso del Double Q-learning (usato per ridurre il bias, cioè l'overestimate di Q), la loss function viene modificata con l'uso di due reti, una che seleziona l'azione e una che la valuta

$$L^Q(\omega) = (r + \gamma \hat{Q}(s_t, \hat{\pi}(s_t, \theta^-), \omega^-) - \hat{Q}(s_t, a_t, \omega))^2$$

Nel caso deterministico,  $\hat{Q}$  può essere imparata off-policy.

### Model-Based Reinforcement learning

È una famiglia di RL algorithmi, i quali prima imparano il modello dell'ambiente dalle esperienze fatte e poi usa il modello per imparare  $V, Q$  e  $\pi$  con una procedura chiamata "planning".

Ye task di model learning è un problema supervisionato che consiste nell'imparare il mapping che c'è da stati e azioni a stati-successivi ( $p_{ss'}^a$ ) e reward ( $R_s^a$ ).

OSS:

- Ye mapping  $R_s^a$  è un regression problem, mentre  $p_{ss'}^a$  è un density estimation problem.
- Assumiamo la conoscenza a priori degli stati e azioni
- Ye learned model (parametrizzato da  $\eta$ ) consiste nella transition probability function  $P_\eta$  e reward function  $R_\eta$  imparate.  $\rightarrow M_\eta = \langle P_\eta, R_\eta \rangle$
- Abbiamo due errori:
  - 1) Model approximation
  - 2) Value-fun / policy approximation
- Ye modello generato dalle esperienze initialmente raccolte, può essere usato generare ulteriori esperienze

- Nel sample based planning, l'agente usa il modello per generare un insieme di possibili stati futuri e poi sceglie le azioni sulla base dei risultati simulati
- Il modello imparato è un'approssimazione di quello reale, potrebbe non catturare tutti i dettagli e portare a possibili policy non ottimali.

Quando l'incertezza sul modello è troppa, è possibile usare diversi approcci:

- 1) Tornare all'utilizzo di metodi model-free
- 2) Individuare l'incertezza nel come l'agente ragiona
- 3) Usare un appoggio che integra (1) e (2), cioè:
  - a) l'agente impara dalle esperienze reali (interagendo con l'env, usate per controllare ciò che l'agente ha imparato in termini di modello e policy)
  - b) l'agente impara dalle esperienze simulate (fatto sul modello imparato), le quali permettono di esplorare maggiormente e generare esperienze senza l'interazione effettiva con l'env reale.

### Dyna-Q algorithm

Model-based RL algorithm che lavora in due fasi, nella prima interagisce con l'ambiente e raccoglie esperienze, nella seconda ripete iterativamente le esperienze raccolte per imparare il modello e la policy.

- Dyna-Q ha l'abilità di adattarsi automaticamente ai cambiamenti nell'environnement, aggiornando il suo modello e policy di conseguenza
- Dyna-Q+ è una versione migliorata da quale, quando ripete le esperienze, esse vengono selezionate non con la stessa probabilità, ma da una weighted distribution, dove i pesi sono dati dalla seguente formula, la quale incoraggia l'agente a scegliere state-action pairs che non ha esplorato recentemente o ritornano reward maggiori (questo velocizza l'algoritmo).

$$\text{Weights: } w_s(\tau) = \mathbb{E}_a[R_s^a] + k\sqrt{\tau}$$

**Algorithm 1** Dyna-Q algorithm

```

Input:  $S, A, \gamma$ 
Initialize  $Q$  and  $M$ 
while true do
  Choose  $s \in S$ 
   $a \leftarrow \varepsilon\text{-greedy}(Q(s, \cdot))$ 
  Collect  $r$  and  $s'$ 
  /* Model update */
   $M(s, a) \leftarrow \langle s', r \rangle$ 
  TD(0) update of  $Q(s, a)$ 
  for  $n$  times do
    Choose  $s \in S$  from  $\mathcal{H}$ 
    Choose  $a \in A$  from  $\mathcal{H}(s)$ 
     $\langle r, s' \rangle \leftarrow M(s, a)$ 
    TD(0) update of  $Q(s, a)$ 
  end for
end while

```

Loop nel quale l'agente alterna tra interazione  
 → con l'env (collectione esperienze) oppure fare  
 planning (ripete le esperienze e aggiornare i  
 Q-values)

**MC tree Search (planning approach)** → le esperienze iniziali vengono collegate con l'interazione

È un **model-based RL algorithm** il quale **usa** il **forward search process**, che consiste nel **decidere** un'azione **simulando** **scenari futuri**, cioè **costruendo** un **albero di ricerca** con lo **stato corrente** come **radice** ed **esplorando** le **possibili azioni e risultati** per determinare l'azione migliore.

Il **Naive MC search** **simula** episodi multipli per ogni azione e **stima** il **valore** di ogni azione basandosi sui **returns** (cumulative reward) osservati in queste simulazioni. → Non migliora  $\pi$  e non buono nella long-run.

Il **MC Tree Search** è un'estensione del Naive MC search, ed include 4 fasi:

1) **Selection**: Partendo dalla radice cerca un nodo che rappresenta un buon trade-off tra **exploitation** e **exploration** (promising node)

Una strategia tipica usata è la **Upper Confidence Bound applied to Trees (UCT)** : il quale seleziona il nodo basandosi sull'estimated value e il visitation count.

2) **Expansion**: Se il nodo selezionato è una foglia, aggiunge uno o più nodi figli considerando le possibili azioni corrispondenti; altrimenti aggiunge uno o più nodi figlio.

3) **Playout**: Esegue una simulazione dal nodo espanso, fino allo stato terminale per ottenere un risultato che aiuta a stimare il valore del nodo espanso.

4) **Backpropagation**: Aggiornare le informazioni dei nodi dalla radice al nodo espanso sulla base del risultato ottenuto nella **playout phase**

**Policy evaluation**: Media dei returns osservati per ogni azione durante la **playout phase**

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^K \sum_{t=1}^T \mathbb{I}(s, a) G_t$$

**Policy improvement**: Usa un **E-greedy approach**

**Conduitsche MCTS**

- + Focussa il computation time sull'area più promettente del search space, bilanciando **exploitation** e **exploration**
- + È parallelizzabile
- La ricerca esauriva può non essere possibile in spazi complessi

**TD search**

Viene usato il SARSA algorithm e  $\pi$  ottimizzata con E-greedy dopo ogni step

## Dyna-2

È un'estensione del Dyna algorithm, la quale utilizza due Q-functions:

- $Q^P$  (persistent): Viene aggiornata solo sulle esperienze reali
- $Q^T$  (transient): Viene aggiornata dopo ogni planning step (simulazione) e successivamente clearata, non mantiene info tra un planning step e l'altro.

Quando l'agente deve scegliere un'azione, combina le info da entrambe le Q-functions, scegliendo quella che massimizza la somma  $Q^P + Q^T$ ,  $\epsilon$ -greedy ( $Q^P + Q^T$ ), permettendo di imparare sia dalle real experience che da quelle simulate.

## Exploration - Exploitation tradeoff

Possibili exploration techniques:

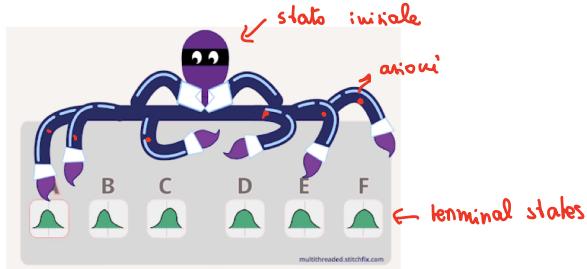
- 1) Random exploration: Metodo semplice, assicura che tutte le azioni vengono provate equivalentemente, inefficiente soprattutto in spazi state-action grandi.
- 2)  $\epsilon$ -greedy: Aggiunge un rumore alla greedy policy permettendo di scegliere con  $\epsilon$ -prob. un'azione random.  
Fornisce un buon tradeoff
- 3) Optimism and Uncertainty: L'agente assume che le azioni sconosciute possono portare a reward migliori e le preferisce. Incentiva l'exploration di state-action pairs sconosciute
- 4) Information state: Usa info (come action-state passati) per guidare l'exploration.
- 5) Look ahead: Usa info di future step simulati per decidere l'azione.

## Multi-armed bandit scenario

È un RL problem classico che mostra il tradeoff tra exploitation ed exploration.

Esso consiste in un casinò con una riga di slot machine, ognuna con un braccio, ed ognuna di esse fornisce un differente reward sulla base di una distib. di probabilità specifica per quella slot machine (non sappiamo questa distribuzione). L'obiettivo è di massimizzare il reward decidendo a quali slot machines giocare, in quale ordine e quante volte. → Imparare  $R^a$ , cioè l'expected reward per ogni azione

- Stati deterministici: Ogni round inizia dallo stesso stato, non porta info dai round precedenti (ogni pull è indipendente)
- Azioni multiple: Un'azione corrisponde a qualche slot machine giocare, quindi più slot machine → più azioni
- Differenti terminal state: Ogni pull corrisponde al termine di un round, quindi a un terminal state ↗
- No state space: Non-standard, è composto da un solo stato non-terminal e gli altri sono tutti terminal states



Un metodo per risolvere questi RL problems è usare strategie che bilanciano il trade-off tra exploitation ed exploration.

### Regret

**Regret**: Misura la differenza fra expected reward della miglior strategia possibile (tenere giù la leva che dà reward maggiore) e la strategia seguita attualmente.

$$L_T = \mathbb{E} \left[ \sum_{t=1}^T V^* - Q(a_t) \right] = \sum_{a \in A} \mathbb{E}[N(a)](V^* - Q(a)) = \sum_{a \in A} \mathbb{E}[N(a)] \Delta_a$$

↳ n° volte a scelta

→ • È la stessa cosa di massimizzare l'expected cumulative reward  
•  $V^*$  non è conosciuto

**Asymptotic Regret**: Limite del regret con n° step ( $T$ ) che va a infinito, più è piccolo questo valore, meglio è perché l'algoritmo sta migliorando nel tempo.

È stato dimostrato che:

- **Greedy algorithms** e  **$\epsilon$ -greedy algorithms** hanno un linear regret, cioè il regret aumenta linearmente con il numero di passi ( $T$ )  
Nella pratica non possibile, non l'abbiamo
- **Decaying  $\epsilon$ -greedy algorithm**: Se abbiamo un oracle disponibile che fornisce info perfette per il problema, la versione  $\epsilon$ -greedy con  $\epsilon$  decay può avere un sublinear regret, cioè il regret aumenta più lentamente che lineare con il numero di passi ( $T$ ).

Altre tecniche empiriche possono essere usate per ridurre il regret:

- High-confidence elimination: Eliminare gradualmente azioni che sembrano subottimali
- Successive Elimination: Eliminare la peggior azione dopo ogni round

**Hoeffding's Inequality**: Fornisce un upper bound sulla prob. che la somma di random variable indipendenti devino dal loro expected value (stima l'incertezza)

**Proposition (Hoeffding's inequality)**  
Let  $X_1, \dots, X_i$  be i.i.d. random variables in  $[0,1]$ , and let  $\bar{x}_i = \frac{1}{i} \sum_{j=1}^i x_j$  be the sample mean. Then,  
 $\mathbb{P}(\mathbb{E}[x] > \bar{x}_i + u) \leq e^{-2u^2}$

L'obiettivo è quello di implementare una sublinear regret exploration, senza oracle.

## Upper Confidence Bound (UCB)

La scelta di un'azione dovrebbe considerare due fattori:

- Expected reward : Exploitation
- Variance (incertezza) : Exploration

La upper confidence di un'azione,  $U_T(a)$ , misura l'incertezza nel reward di quell'azione, ed è inversamente proporzionale con il numero di volte che quell'azione è stata scelta  $U_T(a) \propto \frac{1}{N(a)}$ , e current optimal expected action value,  $Q_T(a^*)$ , è il maggior expected reward tra tutte le azioni. L'idea è di considerare solo le azioni t.c.  $Q_T(a^*) \leq Q_T(a) + U_T(a)$ , cioè azioni che, dato una certa incertezza, possono superare potenzialmente l'attuale expected reward della miglior azione.

Quindi l'azione scelta dovrebbe essere quella che maximizza la somma fra l'expected reward e l'upper confidence bound  $\rightarrow a = \arg \max_a Q_T(a) + U_T(a)$

Quest'idea è conosciuta come Upper Confidence Bound (UCB) algorithm.

Formalmente:

- Riscriviamo la Hoeffding's inequality:  $\mathbb{P}(Q_T(a^*) > Q_T(a) + U_T(a)) \leq e^{-2N_T(a)U_T^2(a)}$
- Richiediamo un certo livello di incertezza ( $p$ ):  $e^{-2N_T(a)U_T^2(a)} = p \implies U_T(a) = \sqrt{\frac{-\ln p}{2N(a)}}$
- Nel tempo, con l'aumentare delle info, l'incertezza deve decrescere (quindi all'aumentare di  $T \rightarrow p$  decresce)  
$$p = T^{-4} \implies U_T(a) = \sqrt{\frac{2 \ln T}{N(a)}}$$
 steps  
decay

Ottieniamo quindi un regreto sublineare ed inoltre UCB garantisce che se eseguiamo l'algoritmo per abbastanza tempo, la selezione di un'azione convergerà all'azione ottimale che maximizza l'expected reward.

## Bayesian Exploration

È un exploration method che usa conoscenza a priori sulla reward distib. per action.

L'obiettivo è di maximizzare la posterior distribution (distribuzione iniziale aggiornata con le esperienze fatte) tramite la Bayes equation.

Bayesian UCB: Algoritmo che estende ucb per incorporare la conoscenza a priori.

$$\rightarrow \text{Sceglie l'azione che maximizza la standard deviation } a_t = \arg \max_{a \in A} \mu_a + \frac{C\sigma_a}{\sqrt{N_T(a)}}$$