

PER ALTRI APPUNTI CONSULTARE IL SITO:
https://luigi-v.github.io/Appunti_Universita/

10. CONOSCENZA INCERTA E RAGIONAMENTO

Dagli agenti logici passiamo a scenari nei quali è presente molta incertezza così da non riuscire a gestirli con un approccio formale come quello logico. Con la logica riusciamo a fare ragionamenti e riusciamo a dedurre nuova conoscenza andando a modellare la KB. Negli scenari più reali, risulta difficile applicare approcci logici, siccome la realtà racchiude incertezze. In questo caso, l'incertezza viene modellata attraverso la probabilità, quindi avremo non più un modello logico ma un modello probabilistico.

Esempio:

Sia $A_t = \text{avviamoci all'aeroporto } t \text{ minuti prima del volo}$. Mi permetterà A_t di giungere in tempo per il volo?

Problemi:

- Osservabilità parziale (stato della strada, piani di altri autisti, etc);
- Sensori rumorosi (sensori che percepiscono informazioni errate o disturbate);
- Incertezza nei risultati dell'azione (foratura gomma);
- Elevata complessità nella modellazione e nella predizione del traffico.

Pertanto, un approccio puramente logico non è fattibile:

- Comporta rischio di falsità: “ A_{25} mi porterà in tempo a destinazione”, oppure
- Porta a conclusioni troppo deboli per il decision making:
 - A_{25} mi porterà a destinazione in tempo a patto che non ci siano incidenti sul ponte, non piove, gomme integre etc.
 - Ragionevolmente si può dire che A_{1440} mi porterebbe a destinazione in tempo, ma dovrei pernottare in aeroporto.

In tal caso abbiamo tante possibilità corrette, ma l'agente deve fornire un'unica risposta.

DECISIONE RAZIONALE:

Un agente logico non è in grado di agire poiché non conosce con quali azioni raggiungere l'obiettivo. L'informazione in possesso dell'agente non può garantire i possibili esiti di A_{90} , ma può fornire un grado di credenza sul loro raggiungimento.

La cosa giusta da fare dipende:

1. Dall'importanza relativa ai vari obiettivi;
2. Dalla probabilità e dalla misura del loro raggiungimento.

INADEGUATEZZA DELL'APPROCCIO LOGICO:

Consideriamo un esempio di diagnosi medica:

Sbagliato! Non tutti i pazienti che accusano mal di denti hanno carie:

Elenco lunghissimo di cause. Regola causale:

$$\forall p \text{ Sintomo}(p, \text{MalDiDenti}) \Rightarrow \text{Malattia}(p, \text{Carie})$$

$$\forall p \text{ Sintomo}(p, \text{MalDiDenti}) \Rightarrow \text{Malattia}(p, \text{Carie}) \vee \text{Malattia}(p, \text{Gengivite}) \vee \text{Malattia}(p, \text{Ascesso})....$$

$$\forall p \text{ Malattia}(p, \text{Carie}) \Rightarrow \text{Sintomo}(p, \text{MalDiDenti})$$

Non tutte le carie causano dolore. Bisogna elencare tutte le cause del mal di denti sul lato sinistro.

PROBABILITÀ:

Il mondo cambia da proposizioni che sono vere o false a proposizioni che hanno un certo **grado di credenza** (rappresentato da un valore di probabilità) del modello di agente. Date le evidenze disponibili so che A_{25} mi porterà in tempo a destinazione con probabilità 0.04. Le asserzioni probabilistiche sintetizzano gli effetti di:

- **Pigrizia:** mancata enumerazione di eccezioni, condizioni, etc., sia perché richiede troppo lavoro, sia perché le regole risulterebbero difficili da usare;
- **Ignoranza teorica:** assenza di fatti rilevanti. Esempio la scienza medica non ha una teoria completa per il suo dominio;
- **Ignoranza pratica:** anche se conosciamo tutte le regole potremmo essere incerti perché non sono state fatte tutte le misurazioni.

Nello scenario che andremo ad analizzare ora, l'agente non ha più certezze in quanto fa uso delle probabilità.

DIFFERENZE CON ONTOLOGIE SPECIFICHE:

Probabilità soggettiva:

- Le probabilità mettono in relazione proposizioni e stato della conoscenza dell'agente (probabilità condizionate), cioè:

$$P(A_{25} | \text{nessun incidente}) = 0.06$$

Queste non sono asserzioni sul mondo reale.

- Le probabilità di proposizioni cambiano con nuove evidenze, cioè:

$$P(A_{25} | \text{nessun incidente, alle 5 a.m.}) = 0.15$$

Il grado di credenza è **diverso** dal grado di verità. Le formule sono sempre vere o false. Una probabilità di 0.8 indica un grado di credenza nella verità dell'80%.

PRENDERE DECISIONI INCERTE:

Supponiamo che io abbia le seguenti convinzioni:

$$P(A_{25} \text{ mi porta in tempo a destinazione} | \dots) = 0.04$$

$$P(A_{90} \text{ mi porta in tempo a destinazione} | \dots) = 0.70$$

$$P(A_{120} \text{ mi porta in tempo a destinazione} | \dots) = 0.95$$

$$P(A_{1440} \text{ mi porta in tempo a destinazione} | \dots) = 0.9999$$

NOTA: i numeri come pedice di A rappresenta t (A_t = avviamoci all'aeroporto t minuti prima del volo).

Il tempo non è l'unico fattore, dipende anche dalle mie preferenze tra perdere il volo rispetto a passare del tempo ad attenderlo, etc.

La Teoria dell'Utilità viene usata per rappresentare ed inferire preferenze. Assieme alla Teoria della Probabilità forma:

Teoria delle Decisioni = Teoria della Probabilità + Teoria dell'Utilità

$$\text{Maximize expected utility : } a^* = \underset{a}{\operatorname{argmax}} \sum_s P(s | a) U(s)$$

DECISION-THEORETIC AGENT CHE SELEZIONA AZIONI RAZIONALI:

Un agente che fa uso della teoria della decisione funziona nel modo seguente:

L'agente prendendo in input una percezione, calcola le probabilità per le azioni, dopo sceglie quella che ci dà la **highest expected utility**.

```
function DT-AGENT(percept) returns an action
  persistent: belief_state, probabilistic beliefs about the current state of the world
              action, the agent's action

  update belief_state based on action and percept
  calculate outcome probabilities for actions,
    given action descriptions and current belief_state
  select action with highest expected utility
    given probabilities of outcomes and utility information
  return action
```

SINTASSI DI PROPOSIZIONI:

Una **variabile casuale** può assumere diversi valori, ed ogni valore ha una certa probabilità. Le variabili casuali si dividono in:

- **Booleans**, possono assumere o TRUE o FALSE;
- **Discrete**, assumono più di due valori possibili. Esempio il *tempo atmosferico* che può assumere <soleggiato, piovoso, nuvoloso, neve>;
- **Continuous**, esprimono una distribuzione come una funzione parametrizzata di valori (che non vengono trattate);

I **valori dei domini** sono **esaurivi**, cioè sono presenti tutti i possibili valori, e sono **mutuamente esclusivi**.

Le **proposizioni elementari**, costruite assegnando un valore ad una variabile casuale:

$$\text{Tempo} = \text{soleggiato}, \text{Carie} = \text{Falso} (\neg \text{carie}).$$

Le **proposizioni complesse**, formate da proposizioni elementari e connettivi logici standard:

$$\text{Tempo} = \text{soleggiato} \vee \text{Carie} = \text{falso}.$$

Un **evento atomico** rappresenta una specifica completa del mondo dell'agente che è incerto.

Se il mondo consiste solo di 2 variabili booleane Carie e MalDiDenti, allora ci sono 4 eventi atomici distinti:

$$\begin{aligned} \text{Carie} = \text{falso} \wedge \text{MalDiDenti} = \text{falso} \\ \text{Carie} = \text{falso} \wedge \text{MalDiDenti} = \text{vero} \\ \text{Carie} = \text{vero} \wedge \text{MalDiDenti} = \text{falso} \\ \text{Carie} = \text{vero} \wedge \text{MalDiDenti} = \text{vero} \end{aligned}$$

Gli eventi atomici sono mutuamente esclusivi (massimo 1 si verifica) ed esaurivi (almeno 1).

PROBABILITÀ A PRIORI:

Sono probabilità secondo le quali una certa variabile casuale assuma un certo valore senza avere nessun'altra conoscenza.

- $P(\text{Carie}=\text{ver})=0$ o $P(\text{Tempo}=\text{soleggiato})=0.72$ corrisponde alla confidenza prima dell'arrivo di qualunque (nuova) evidenza.

Ogni possibile mondo w è associato con un valore di probabilità tale che:

- $0 \leq P(w) \leq 1$

- $\sum_{w \in \Omega} P(w) = 1$

Esempio: se lanciamo due dadi (distinguibili) ci sono 36 possibili mondi da considerare: (1,1), (1,2), ..., (1,6) $P(w)=1/36$

Una **distribuzione di probabilità** fornisce valori per tutti i possibili assegnamenti:

- Es $P(\text{tempo}) = \langle 0.72, 0.1, 0.08, 0.1 \rangle$ (normalizzata, i.e., somma a 1)

- $P(\text{Dadi}) = \langle 1/36, \dots, 1/36 \rangle$

Una **distribuzione di probabilità congiunta** per un insieme di variabili casuali fornisce le probabilità di ogni evento atomico di tali variabili.

Esempio:

$P(\text{Tempo}, \text{Carie})$ = una matrice 4×2 con valori:

Tempo =	soleggiato	piovoso	nuvoloso	neve
<i>Carie</i> = vero	0.144	0.02	0.016	0.02
<i>Carie</i> = falso	0.576	0.08	0.064	0.08

Ogni domanda su un dominio può essere risposta con una distribuzione congiunta.

PROBABILITÀ CONDIZIONATA:

La **probabilità condizionata** è la probabilità che una variabile (es *a*) assume un certo valore sapendo già che un dato evento è accaduto (*b*):

$$P(a | b) = P(a \wedge b) / P(b) \quad \text{if } P(b) > 0 \quad \left| P(\text{doubles} | \text{Die}_1=5) = \frac{P(\text{doubles} \wedge \text{Die}_1=5)}{P(\text{Die}_1=5)} \right.$$

Questa probabilità condizionata è importante per un agente, siccome esso parte da una data conoscenza.

La **regola del prodotto** offre una formulazione alternativa:

$$P(a \wedge b) = P(a | b) P(b) = P(b | a) P(a)$$

Questa regola ci permette di definire la probabilità che due eventi sono accaduti.

Esempio:

$$P(\text{Tempo}, \text{Carie}) = P(\text{Tempo} | \text{Carie}) P(\text{Carie})$$

(Vista come un insieme di 4×2 equazioni, non più come una matrice multipla)

La regola del prodotto anziché avere solo due variabili possiamo generalizzarla ad *n*, questa è chiamata **Chain rule**, che è derivata attraverso successive applicazioni della regola del prodotto:

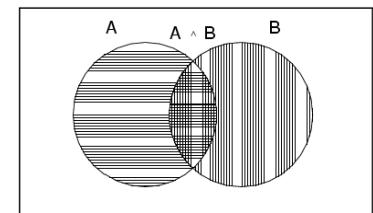
$$\begin{aligned} P(X_1, \dots, X_n) &= P(X_1, \dots, X_{n-1}) P(X_n | X_1, \dots, X_{n-1}) \\ &= P(X_1, \dots, X_{n-2}) P(X_{n-1} | X_1, \dots, X_{n-2}) P(X_n | X_1, \dots, X_{n-1}) \\ &= \dots \\ &= \prod_{i=1}^n P(X_i | X_1, \dots, X_{i-1}) \end{aligned}$$

ASSIOMI DI PROBABILITÀ:

Per ogni coppia di proposizioni A, B:

- $0 \leq P(A) \leq 1$
- $P(\text{true}) = 1$ and $P(\text{false}) = 0$
- $P(A \vee B) = P(A) + P(B) - P(A \wedge B)$

Questi 3 assiomi sono la base su cui si può dimostrare tutto che è stato visto.



INFERENZA TRAMITE ENUMERAZIONE:

Adesso ciò che interessa è implementare un agente che risponde a delle domande definendo un processo inferenziale, facendo un'inferenza per enumerazione, cioè enumerando tutti i possibili valori.

L'approccio si basa sulla **distribuzione congiunta di probabilità**:

	malidenti		\neg malidenti	
	prende	$\neg \text{prende}$	prende	$\neg \text{prende}$
carie	.108	.012	.072	.008
\neg carie	.016	.064	.144	.576

La probabilità di una proposizione ϕ è data dalla somma degli **eventi atomici** ω su cui ϕ diventa vera:

$$P(\phi) = \sum_{\omega: \omega \models \phi} P(\omega)$$

Esempio:

Proposizione: *malidenti* =vera, qual è la probabilità che l'evento *malidenti* sia vero?

Bisogna sommare tutti gli eventi atomici in cui il *malidenti* è vero:

$$P(\text{malidenti}) = 0.108 + 0.012 + 0.016 + 0.064 = 0.2 \quad (\text{probabilità marginale})$$

Oppure, qual è la probabilità che l'evento *caries* sia vero?

$$P(\text{caries}) = 0.108 + 0.012 + 0.072 + 0.008$$

Spesso siamo interessati a calcolare le **probabilità condizionali** di alcune variabili, date le evidenze su altre.

Esempio:

Proposizione: Quale è la probabilità di avere *carie* sapendo di avere *maldimenti*?

$$P(\text{carie} | \text{maldimenti}) = P(\text{carie} \wedge \text{maldimenti}) / P(\text{maldimenti}) = (0.108 + 0.012) / (0.108 + 0.012 + 0.016 + 0.064) = 0.6$$

Il suo negato è il complemento:

$$P(\neg \text{carie} | \text{maldimenti}) = P(\neg \text{carie} \wedge \text{maldimenti}) / P(\text{maldimenti}) = 0.4$$

NORMALIZZAZIONE:

Nei calcoli, il denominatore è lo stesso e può essere visto come una **costante di normalizzazione** α . Serve che il risultato è compreso tra 0 e 1 e che la somma sia 1.

$$\begin{aligned} P(\text{Carie} | \text{maldimenti}) &= \alpha P(\text{Carie}, \text{maldimenti}) \\ &= \alpha [P(\text{Carie}, \text{maldimenti}, \text{prende}) + P(\text{Carie}, \text{maldimenti}, \neg \text{prende})] \\ &= \alpha [0.108, 0.016] + [0.012, 0.064] \\ &= \alpha [0.12, 0.08] = [0.6, 0.4] \quad \text{Non ci serve conoscere } P(\text{maldimenti})! \end{aligned}$$

L'idea generale è calcolare la distribuzione sulla variabile di query fissando variabili di evidenze (maldimenti) e sommando variabili nascoste (prende).

INFERENZA PER ENUMERAZIONE:

In genere, siamo interessati a:

- La distribuzione congiunta a posteriori delle variabili di query X (Carie nell'esempio);
- Dati valori specifici e per le variabili di evidenza E (MalDiDenti nell'esempio).

Siano le **variabili nascoste H** = Y – X – E, il risultato richiesto è ottenuto sommando le variabili nascoste:

$$P(X | E=e) = \alpha P(X, E=e) \alpha \sum_h P(X, E=e, H=h)$$

I termini nella sommatoria rappresentano entry congiunte, perché X, E ed H insieme esauriscono l'insieme di variabili casuali.

INFERENZA PROBABILISTICA:

X rappresenta le query, e rappresenta le evidenze e P la distribuzione congiunta.

Il problema di questo algoritmo è la distribuzione congiunta di variabili perché la dimensione è molto grande.

Quindi **complessità di tempo** nel caso peggiore $O(d^n)$ dove d è la più grande arită.

Complessità di spazio $O(d^n)$ per memorizzare la distribuzione congiunta.

Come trovare i numeri per $O(d^n)$ entry?

```
function ENUMERA-CONGIUNTA-ASK(X, e, P) returns una distribuzione su X
    inputs: X, la variabile della query
            e, i valori osservati per le variabili E
            P, una distribuzione congiunta sulle variabili {X} ∪ E ∪ Y
            /* Y = variabili nascoste */
    Q(X) ← una distribuzione su X, inizialmente vuota
    for each valore  $x_i$  di X do
         $Q(x_i) \leftarrow \text{ENUMERA-CONGIUNTA}(x_i, e, Y, [], P)$ 
    return NORMALIZZA(Q(X))

function ENUMERA-CONGIUNTA(x, e, variabili, valori, P) returns un numero reale
    if VUOTA(variabili) then return P(x, e, valori)
    Y ← PRIMO(variabili)
    return  $\sum_y \text{ENUMERA-CONGIUNTA}(x, e, RESTO(variabili), [y|valori], P)$ 
```

INDIPENDENZA:

Quando gli eventi sono indipendenti possono essere trattati parallelamente. A e B sono **indipendenti** se e solo se:

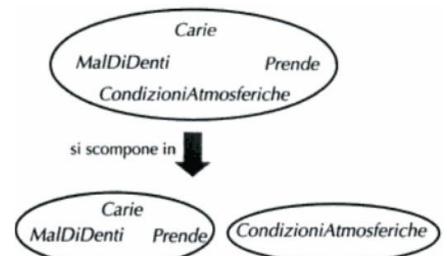
$$P(A|B)=P(A) \text{ or } P(B|A)=P(B) \text{ or } P(A,B)=P(A)P(B).$$

Esempio:

$$P(\text{maldimenti}, \text{prende}, \text{carie}, \text{condizioni}) = P(\text{maldimenti}, \text{prende}, \text{carie})P(\text{condizioni})$$

Da 32 entry si riducono a 12; per n lanci di moneta indipendenti, $O(2^n) \rightarrow O(n)$.

L'indipendenza assoluta è molto potente ma rara nella realtà.



INDIPENDENZA CONDIZIONALE:

Una cosa che succede più frequente è l'indipendenza condizionale che suppone che gli eventi non siano del tutto indipendenti tra loro.

Esempio:

$$P(MalDiDenti, Prende, Carie) \text{ ha } 2^3 - 1 = 7 \text{ entry indipendenti}$$

Se ho una carie, la probabilità che lo strumento appuntito si blocca non dipende dal fatto che io abbia mal di denti:

$$(1) P(\text{prende} | \text{maldidenti, carie}) = P(\text{prende} | \text{carie})$$

La stessa indipendenza vale se io non ho una carie:

$$(2) P(\text{prende} | \text{maldidenti, } \neg \text{carie}) = P(\text{prende} | \neg \text{carie})$$

Prende è **condizionalmente indipendente** da MalDiDenti data Carie:

$$P(\text{Prende} | \text{MalDiDenti, Carie}) = P(\text{Prende} | \text{Carie})$$

Affermazioni equivalenti:

$$P(\text{MalDiDenti} | \text{Prende, Carie}) = P(\text{MalDiDenti} | \text{Carie})$$

$$P(\text{MalDiDenti}, \text{Prende} | \text{Carie}) = P(\text{MalDiDenti} | \text{Carie}) P(\text{Prende} | \text{Carie})$$

Decomposizione della distribuzione congiunta completa tramite **chain rule**:

$$\begin{aligned} & P(\text{MalDiDenti}, \text{Prende}, \text{Carie}) \\ &= P(\text{MalDiDenti} | \text{Prende, Carie}) P(\text{Prende}, \text{Carie}) \\ &= P(\text{MalDiDenti} | \text{Prende, Carie}) P(\text{Prende} | \text{Carie}) P(\text{Carie}) \\ &= P(\text{MalDiDenti} | \text{Carie}) P(\text{Prende} | \text{Carie}) P(\text{Carie}) \end{aligned}$$

cioè $2 + 2 + 1 = 5$ entry indipendenti

Nella maggior parte dei casi, l'uso dell'indipendenza condizionale riduce la dimensione della rappresentazione della distribuzione congiunta da esponenziale in n a lineare in n.

L'indipendenza condizionale è la nostra forma più semplice e solida di conoscenza di ambienti incerti.

REGOLA DI BAYES:

La **regola di Bayes** può essere usata nell'inferenza, viene semplicemente riscritta delle regole condizionali precedenti:

$$\text{Regola del prodotto: } P(a \wedge b) = P(a | b) P(b) = P(b | a) P(a) \quad \rightarrow \quad \text{Bayes' rule: } P(a | b) = P(b | a) P(a) / P(b)$$

Oppure, in forma distribuita:

$$P(Y | X) = P(X | Y) P(Y) / P(X) = \alpha P(X | Y) P(Y)$$

Perché è utile:

- Costruiamo un condizionale dal suo inverso;
- Spesso un condizionale è complicato ma l'altro è semplice;
- Descrive un passaggio di "aggiornamento" dal precedente $P(a)$ al successivo $P(a | b)$.

Utile per valutare la probabilità diagnostica dalla **probabilità causale**:

$$P(\text{Cause} | \text{Effect}) = P(\text{Effect} | \text{Cause}) P(\text{Cause}) / P(\text{Effect})$$

- $P(\text{Effect} | \text{Cause})$ descrive la direzione causale;
- $P(\text{Cause} | \text{Effect})$ descrive la relazione diagnostica.

La regola di Bayes è molto utilizzata perché quando si hanno le probabilità condizionate si hanno delle variabili che sono *cause* e altre che sono *effetto* di quelle cause, il poter invertire l'ordine aiuta siccome si conosce più la probabilità di avere un effetto data una cerca causa.

Esempio diagnosi medica (regola di Bayes):

Dai casi passati sappiamo che: $P(\text{symptoms} | \text{disease})$, $P(\text{disease})$, $P(\text{symptoms})$.

Per un nuovo paziente conosciamo i sintomi e cerchiamo una diagnosi quindi $P(\text{disease} | \text{symptoms})$:

- La meningite provoca un torcicollo il 70% delle volte;
- La probabilità a priori di meningite è 1/50000;
- La probabilità a priori di torcicollo è 1%.

Qual è la probabilità che un paziente con un torcicollo abbia la meningite?

$$P(m | s) = P(s | m) * P(m) / P(s) = 0.7 * 1/50000 / 0.01 = 0.0014$$

Perché la probabilità condizionale per la direzione diagnostica non viene memorizzata direttamente?

- La conoscenza diagnostica è spesso più fragile ai cambiamenti di valori alle variabili rispetto alla conoscenza causale;

Per esempio, se c'è un'improvvisa epidemia di meningite, la probabilità incondizionata di meningite $P(m)$ salirà; quindi, anche $P(m | s)$ dovrebbe salire mentre la relazione casuale $P(s | m)$ non è influenzata dall'epidemia, poiché riflette come funziona la meningite.

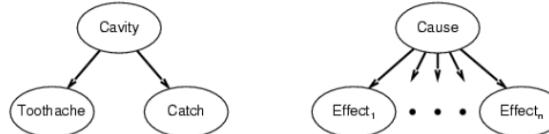
INDIPENDENZA CONDIZIONALE (REGOLA DI BAYES):

Assume che tutti gli effetti siano indipendenti tra di loro, e lo rende molto semplicistico.

$$\begin{aligned} \mathbf{P}(\text{Carie} \mid \text{malidenti} \wedge \text{prende}) \\ = \alpha \mathbf{P}(\text{malidenti} \wedge \text{prende} \mid \text{Carie}) \mathbf{P}(\text{Carie}) \\ = \alpha \mathbf{P}(\text{malidenti} \mid \text{Carie}) \mathbf{P}(\text{prende} \mid \text{Carie}) \mathbf{P}(\text{Carie}) \end{aligned}$$

Questo è un esempio di modello **naïve Bayes**:

$$\mathbf{P}(\text{Cause}, \text{Effect}_1, \dots, \text{Effect}_n) = \mathbf{P}(\text{Cause}) \prod \mathbf{P}(\text{Effect}_i \mid \text{Cause})$$



Il numero totale di parametri è **lineare** in n.

Esempio nel mondo del Wumpus:

Abbiamo un labirinto con pozzi che vengono rilevati nei quadrati vicini attraverso il segnale brezza. Ogni cella contiene un pozzo con probabilità 0.2 (eccetto (1,1)). Dove dovrebbe andare l'agente, se c'è brezza a (1,2) e (2,1)? La pura inferenza logica non può concludere nulla su quale quadrato sia più probabile che sia sicuro!

Assumiamo di **sapere** che:

$$\begin{aligned} b &= \neg b_{1,1} \wedge b_{1,2} \wedge b_{2,1} \\ \text{known} &= \neg p_{1,1} \wedge \neg p_{1,2} \wedge \neg p_{2,1} \end{aligned}$$

Siamo interessati a rispondere a **query** come:

$$\mathbf{P}(P_{1,3} \mid \text{known}, b)$$

La **risposta** può essere calcolata elencando l'intera distribuzione di probabilità congiunta.

Siano Unknown le variabili $P_{i,j}$ eccetto $P_{1,3}$ e Known:

$$\mathbf{P}(P_{1,3} \mid \text{known}, b) = \sum_{\text{Unknown}} \mathbf{P}(P_{1,3}, \text{unknown}, \text{known}, b)$$

Ma significa esplorare tutti i possibili valori delle variabili sconosciute e ci sono $2^{12}=4096$ termini (crescita esponenziale nel numero di stanze).

Possiamo farlo meglio (più velocemente)?

Variabili casuali booleane:

$$\begin{aligned} P_{ij} &= \text{pozzo nella casella } (i,j) \\ B_{ij} &= \text{brezza nella casella } (i,j) \\ (\text{solo per le caselle osservate } B_{1,1}, B_{1,2} \text{ e } B_{2,1}) \end{aligned}$$

1.4	2.4	3.4	4.4
1.3	2.3	3.3	4.3
1.2 B	2.2	3.2	4.2
1.1	2.1 B	3.1	4.1

Distribuzione completa delle probabilità congiunte

$$\mathbf{P}(P_{1,2}, \dots, P_{4,4}, B_{1,1}, B_{1,2}, B_{2,1}) = \mathbf{P}(B_{1,1}, B_{1,2}, B_{2,1} \mid P_{1,2}, \dots, P_{4,4}) * \mathbf{P}(P_{1,2}, \dots, P_{4,4})$$

$$\mathbf{P}(P_{1,2}, \dots, P_{4,4}) = \prod_{i,j} \mathbf{P}(P_{ij})$$

Regola del prodotto

$$\mathbf{P}(P_{1,2}, \dots, P_{4,4}) = 0.2^n * 0.8^{16-n}$$

I pozzi sono distribuiti indipendentemente

la probabilità di pozzi è 0,2 e ci sono n pozzi

Osservazione (Indipendenza condizionale):

Le brezze osservate sono condizionatamente indipendenti dalle altre variabili date le variabili note (bianco), di frontiera (giallo) e di query.

Dividiamo l'insieme delle variabili nascoste in frontiera e altre variabili:

$$\text{Unknown} = \text{Fringe} \cup \text{Other}$$

Dall'indipendenza condizionale abbiamo:

$$\mathbf{P}(b \mid P_{1,3}, \text{known}, \text{unknown}) = \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe})$$

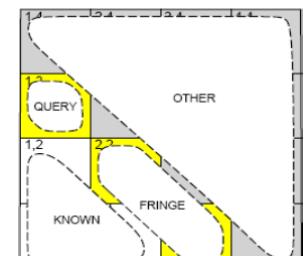
Ora, sfruttiamo questa formula.

Query iniziale ->

$$\mathbf{P}(P_{1,3} \mid \text{known}, b)$$

$$\begin{aligned} &= \alpha \sum_{\text{Unknown}} \mathbf{P}(P_{1,3}, \text{known}, \text{unknown}, b) \\ &= \alpha \sum_{\text{Unknown}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{unknown}) * \mathbf{P}(P_{1,3}, \text{known}, \text{unknown}) \\ &= \alpha \sum_{\text{Fringe}} \sum_{\text{Other}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}, \text{other}) * \mathbf{P}(P_{1,3}, \text{known}, \text{fringe}, \text{other}) \\ &= \alpha \sum_{\text{Fringe}} \sum_{\text{Other}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) * \mathbf{P}(P_{1,3}, \text{known}, \text{fringe}, \text{other}) \\ &= \alpha \sum_{\text{Fringe}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) * \sum_{\text{Other}} \mathbf{P}(P_{1,3}, \text{known}, \text{fringe}, \text{other}) \\ &= \alpha \sum_{\text{Fringe}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) * \sum_{\text{Other}} \mathbf{P}(P_{1,3}) \mathbf{P}(\text{known}) \mathbf{P}(\text{fringe}) \mathbf{P}(\text{other}) \\ &= \alpha \mathbf{P}(\text{known}) \mathbf{P}(P_{1,3}) \sum_{\text{Fringe}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) \mathbf{P}(\text{fringe}) \sum_{\text{Other}} \mathbf{P}(\text{other}) \\ &= \alpha' \mathbf{P}(P_{1,3}) \sum_{\text{Fringe}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) \mathbf{P}(\text{fringe}) \end{aligned}$$

$$\alpha' = \alpha \cdot \mathbf{P}(\text{known}) \sum_{\text{Other}} \mathbf{P}(\text{other}) = 1$$



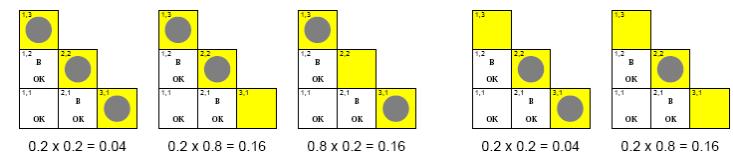
Soluzione:

$$\mathbf{P}(P_{1,3} \mid \text{known}, b) = \alpha' \mathbf{P}(P_{1,3}) \sum_{\text{Fringe}} \mathbf{P}(b \mid P_{1,3}, \text{known}, \text{fringe}) \mathbf{P}(\text{fringe})$$

Esploriamo i possibili modelli (valori) di frontiera compatibili con l'osservazione b.

$$\begin{aligned} \mathbf{P}(P_{1,3} \mid \text{known}, b) \\ = \alpha' \langle 0.2 (0.04 + 0.16 + 0.16), 0.8 (0.04 + 0.16) \rangle \\ = \langle 0.31, 0.69 \rangle \end{aligned}$$

$$\mathbf{P}(P_{2,2} \mid \text{known}, b) = \langle 0.86, 0.14 \rangle$$



Evitare assolutamente il quadrato (2,2)!

11. RAGIONAMENTO PROBABILISTICO

Riassumendo, il problema è gestire l'incertezza, possiamo gestirla utilizzando la probabilità e fare inferenza, ma questo è molto oneroso siccome è esponenziale rispetto al numero di variabili e quindi tutto ciò non è fattibile nella pratica. Possiamo ridurre questo tempo eccessivo con le proprietà di indipendenza delle distribuzioni di probabilità, in particolare, l'indipendenza delle variabili è un evento molto raro però l'indipendenza condizionata è più frequente e questo ci permette di ottenere vantaggi nella computazione.

Le **reti bayesiane** mettono in pratica questi vantaggi sottoforma di un formalismo grafico che permette di andare a rappresentare questi problemi in termini di variabili che dipendono una dall'altra.

I grafici al lato riassumono quello visto nei precedenti capitoli, tra cui distribuzioni di probabilità congiunta dove bisogna fare la sommatoria su tutti i valori delle altre variabili di istanza, ma anche la regola di Bayes sull'indipendenza condizionata che ci permetteva di invertire la formula.

Modelli Naive Bayes: modelli nei quali si assume indipendenza condizionale tra le varie variabili, modello semplicistico con una singola variabile che dipende da tutte le altre.

Basic laws: $0 \leq P(\omega) \leq 1$, $\sum_{\omega \in \Omega} P(\omega) = 1$, $P(A) = \sum_{\omega \in A} P(\omega)$

Random variable $X(\omega)$ has a value in each ω

- ▶ Distribution $P(X)$ gives probability for each possible value x
- ▶ Joint distribution $P(X,Y)$ gives total probability for each combination x,y

Summing out/marginalization: $P(X=x) = \sum_y P(X=x, Y=y)$

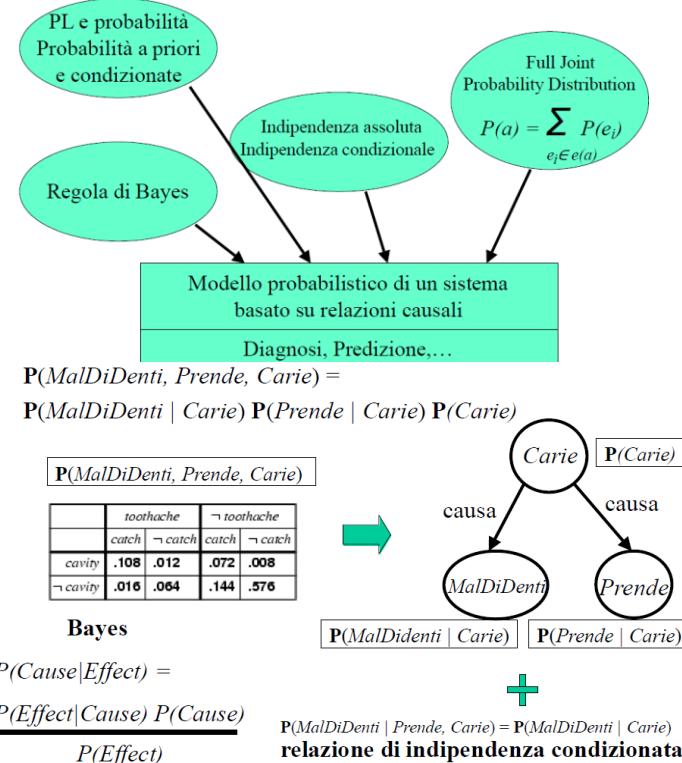
Conditional probability: $P(Y|X) = P(X,Y)/P(X)$

Chain rule: $P(X_1, \dots, X_n) = \prod_i P(X_i | X_1, \dots, X_{i-1})$

Bayes Rule: $P(X|Y) = P(Y|X)P(X)/P(Y)$

Independence: $P(X,Y) = P(X)P(Y)$ or $P(X|Y) = P(X)$ or $P(Y|X) = P(Y)$

Conditional Independence: $P(X|Y,Z) = P(X|Z)$ or $P(X,Y|Z) = P(X|Z)P(Y|Z)$



RETI BAYESIANE:

Ci sposteremo quindi da una **Distribuzione Congiunta di Probabilità** (inefficienti siccome esponenziali alle variabili) ad una **rete bayesiana** che non è altro che una rappresentazione grafica di un insieme di variabili casuali e relazioni di indipendenza tra esse.

Queste reti sono un **grafo orientato aciclico** annotato con distribuzioni di probabilità condizionate:

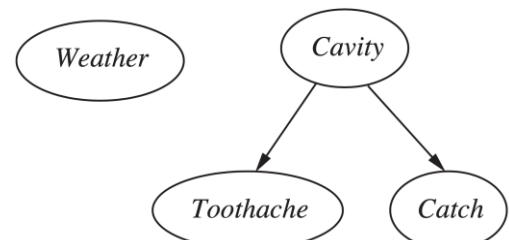
- Insieme di nodi, uno per variabile casuale;
- Insieme di archi orientati che connettono coppie di nodi. Se c'è un arco dal nodo X al nodo Y si dice che X è **parent** di Y (X ha un'influenza diretta su Y)
- Ad ogni nodo X_i è associata una distribuzione di probabilità condizionale che quantifica l'effetto dei parents sul nodo:

$$P(X_i | Parents(X_i))$$

- Per variabili discrete, la distribuzione condizionale è rappresentata come una tabella (**CPT Conditional Probability Table**) che fornisce la distribuzione su X_i per ogni combinazione dei valori dei nodi parents (in direzione causale).

La topologia della rete codifica le asserzioni di indipendenza condizionale:

- Weather è indipendente dalle altre variabili;
- Maldidenti e prende sono condizionalmente indipendenti data Carie (ognuna dipende da carie ma non c'è relazione causale tra le due).



Esempio:

Abbiamo un allarme installato a casa abbastanza affidabile nell'andare a identificare i furti, ma qualche volta si attiva l'allarme in occasione di terremoti (nella zona sono frequenti, a Los Angeles). Ci sono due vicini, Jhon e Mary, che ci chiamano (siamo i proprietari dell'allarme) quando sentono l'allarme. Jhon chiama sempre quando sente l'allarme suonare ma alcune volte lo confonde con il nostro telefono che squilla. Mary, invece, gli piace ascoltare musica ad alto volume, ed alcune volte non ci telefona quando l'allarme suona. Vogliamo stimare la probabilità di un furto.

Le variabili sono 5: Allarme – Telefonate da parte dei due vicini – Furto – Terremoto.

Tramite reti bayesiane, possiamo andare a rappresentare un nodo per ogni variabile casuale e poniamo un arco quando il valore di una variabile è condizionato dal valore di un'altra variabile.

Variabili:

Burglary, Earthquake (Cause), Alarm, JohnCalls, MaryCalls (Effect). Non è un fatto di sintassi, ma definisce quanto il modello è un buon modello della realtà (le relazioni di indipendenza condizionale codificate nel modello sono una sufficiente, per gli scopi dati, approssimazione della realtà).

La topologia della rete riflette la conoscenza causale:

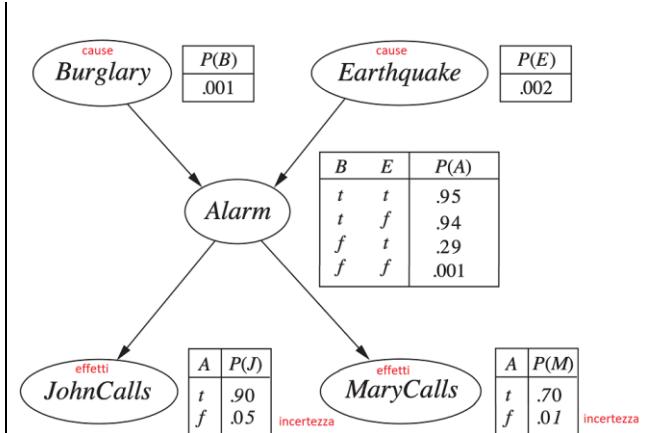
- Il furto con scasso ed il terremoto possono attivare l'allarme;
- Il fatto che Mary o John possano chiamare dipende dall'allarme;
- Mary e John non sono attivati dal furto o dal terremoto e non interagiscono tra di loro.

Permette una rappresentazione della conoscenza incerta e la rete non modella il fatto che Mary ascolta musica ad alto volume e che John confonde il suono del telefono con quello dell'allarme. Tutti questi fattori (e altri potenzialmente infiniti) sono riassunti dall'incertezza associata ai link tra alarm, Mary e John.

Ogni nodo ha una CPT Conditional Probability Table (per variabili discrete).

Ogni riga contiene, per ogni valore del nodo, la probabilità condizionale per un *conditioning case* (una possibile combinazione di valori dei nodi *parents*).

Un nodo senza parents ha una sola riga che rappresenta la probabilità a priori di ogni possibile valore della variabile, nell'esempio Burglary).



Esempio 1 (Gestione del traffico):

Un comune può decidere se bloccare o no le auto per una giornata nel caso in cui si verifichi uno dei seguenti casi:

- Viene raggiunto il livello massimo di inquinamento;
- Si verifica una congestione delle strade.

A seconda della situazione i cittadini dovranno decidere se spostarsi con i mezzi pubblici o prende la macchina.

Avremo **MP** (Mezzo Pubblico) e **M**(macchina) le quali dipendono dal **Blocco** (*B*) e questo dipende dall'*inquinamento* (*I*) e *congestione* (*C*).

Esempio 2 (l'albero di Jack):

Un giorno Jack si accorge che il suo albero di mele perde le foglie. Jack sa che se l'albero è secco allora è normale che perda le foglie ma Jack sa anche che la perdita delle foglie può essere sintomo di malattia per il suo albero.

La rete consiste di 3 nodi:

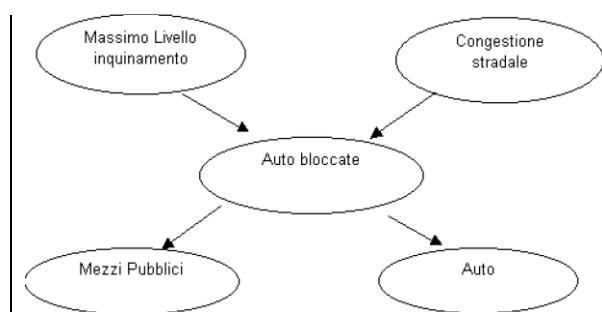
- Malato, Secco e Perde le foglie
- Malato può essere malato o no
- Secco può essere secco o no
- Perde le foglie può essere sì o no

La dipendenza casuale è tra **Malato** e **Perde le foglie** e **Secco** e **Perde le foglie**. Ad ogni nodo è associata una tabella di probabilità, che possono essere a priori o condizionate. Ad esempio:

- $P(\text{Malato} = \text{"malato"}) = 0.1$
- $P(\text{Malato} = \text{"no"}) = 0.9$
- $P(\text{Secco} = \text{"secco"}) = 0.1$
- $P(\text{secco} = \text{"no"}) = 0.9$

<i>B</i>	<i>E</i>	<i>P(A B,E)</i>	<i>P(¬A B,E)</i>
<i>T</i>	<i>T</i>	.95	.05

conditioning case



	Secco="secco"		Secco="No"	
	Malato="Malato"	Malato="No"	Malato="Malato"	Malato="No"
Perde="si"	0.95	0.85	0.90	0.02
Perde="no"	0.05	0.15	0.10	0.98

$P(\text{Perde le foglie} | \text{Malato, Secco})$

SEMANTICA RETI BAYESIANE:

Semantica: una rete è una rappresentazione di una distribuzione di congiunta probabilità. Uno specifico elemento di una distribuzione di probabilità congiunta (evento atomico) è definito come:

$$P(X_1=x_1, \dots, X_n=x_n) \text{ abbreviato in } P(x_1, \dots, x_n)$$

Nelle reti bayesiane, sfruttando l'indipendenza condizionale, abbiamo che ogni variabile dipende solo dai genitori, quindi viene semplificato nel modo seguente. Il valore dell'elemento è:

$$P(x_1, \dots, x_n) = \prod_{i=1, n} P(x_i | \text{Parents}(X_i))$$

- Una distribuzione di probabilità congiunta è definita come il prodotto delle distribuzioni condizionali locali (CPT), date le asserzioni di indipendenza condizionale codificate dalla topologia della rete;
- Le CPT sono la **decomposizione** di una distribuzione di probabilità congiunta.

PROBABILITÀ DELL'EVENTO ATOMICO:

$\text{Parents}(X_i)$ denota i valori specifici delle variabili in X_i definiti in x_1, \dots, x_n (l'elemento corrispondente della CPT di x_i)

$$P(x_1, \dots, x_n) = \prod_{i=1, n} P(x_i | \text{Parents}(X_i))$$

↑
La probabilità di x_i condizionata dai nodi genitori di x_i

Esempio:

$$P(j \wedge m \wedge a \wedge \neg b \wedge \neg e)$$

$$P(x_i | \text{Parents}(X_i))$$

La probabilità di a condizionata dai nodi genitori di a è:

$$P(a | \neg b, \neg e)$$

Applico iterativamente la product rule $P(a \wedge b) = P(a | b) P(b)$:

...

$$= P(j | m \wedge a \wedge \neg b \wedge \neg e) P(m | a \wedge \neg b \wedge \neg e) P(a | \neg b \wedge \neg e) P(\neg b | \neg e) P(\neg e)$$

Utilizzo le relazioni di indipendenza condizionale codificate nel grafo:

$$P(j | m \wedge a \wedge \neg b \wedge \neg e) = P(j | a) \quad j \text{ dipende solo da } a$$

...

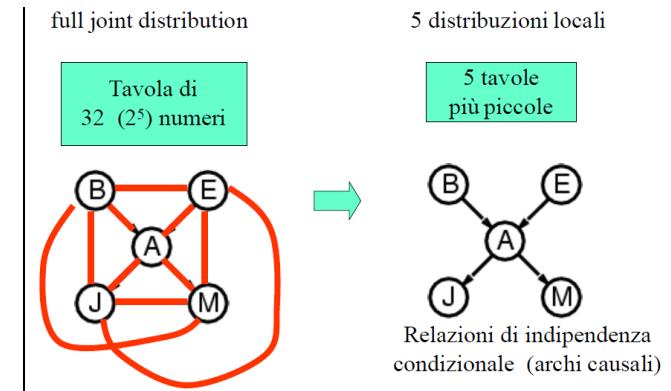
ed ottengo:

$$= P(j | a) P(m | a) P(a | \neg b, \neg e) P(\neg b) P(\neg e)$$

Il risultato finale è dato quindi da $0.90 * 0.70 * 0.001 * 0.999 * 0.998 = 0.00062$.

Alla fine, si riesce a rispondere ad una query su un evento atomico facendo un numero di calcoli molto inferiore rispetto a quello che avevamo visto in precedenza.

In termini numerici, in questo caso abbiamo solo 5 variabili casuali, mentre con la tabella di distribuzione congiunta (full joint distribution) i valori nella tavola di verità erano 32, nella rete bayesiana abbiamo 5 tabelle più piccole



Il vantaggio è la **compattezza**, la rete è una rappresentazione più compatta della distribuzione di probabilità condizionale.

Topologia + CPTs = rappresentazione compatta di una distribuzione di probabilità condizionale.

Se ogni variabile ha non più di k parents, la rete completa di n variabili (booleane) richiede $O(n * 2^k)$ numeri a differenza della distribuzione di probabilità condizionale che cresce esponenzialmente: $O(2^n)$.

Per la rete di allarme precedente: $1+1+4+2+2=10$ mentre l'equivalente distribuzione di probabilità condizionale $25-1=31$.

Se $n=30$ e $k=5$ abbiamo 960 numeri nella RB contro 1 miliardo.

COSTRUZIONE DI UNA RETE:

Data la semantica della rete, come la si costruisce? Come si vede dall'esempio precedente:

- Applicando iterativamente la product rule e si ottiene (**chain rule**):

$$P(x_1, \dots, x_n) = \prod_{i=1, n} P(x_i | X_{i-1}, \dots, X_1)$$

- Utilizzando l'equazione:

$$P(x_1, \dots, x_n) = \prod_{i=1, n} P(X_i | \text{Parents}(X_i))$$

- Si ottiene

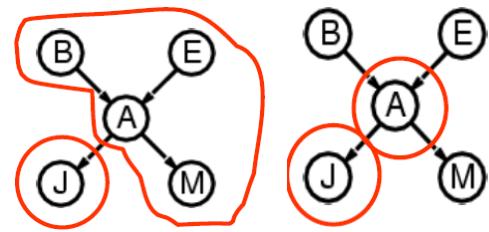
$$P(X_i | X_{i-1}, \dots, X_1) = P(x_i | \text{Parents}(X_i))$$

- La probabilità di X_i condizionata da tutti gli altri nodi è la probabilità di X_i condizionata dai nodi genitori.

In questo caso j dipende solo da a:

$$P(X_i | X_{i-1}, \dots, X_1) = P(x_i | Parents(X_i))$$

Dato l'ordinamento dei nodi X_n, \dots, X_1 , per ogni nodo X_i , la probabilità condizionata dal nodo rispetto a tutti gli altri nodi è la probabilità condizionata rispetto ai nodi genitori. Ogni nodo deve essere condizionalmente indipendente dai suoi predecessori nell'ordinamento dei nodi, dati i suoi nodi genitori. I genitori di ogni nodo devono essere tutti e soli i nodi che influenzano direttamente il nodo.



L'ordine di selezione dei nodi da aggiungere alla rete, sceglieremo prima le cause e poi gli effetti, questo porta a costruire reti ottime. Infatti, l'ordine di scelta dei nodi nella costruzione porta a reti diverse.

Primo modo: costruzione di una rete a partire dal significato numerico

- Si sceglie un ordinamento dei nodi;
- Si scrive un nodo alla volta;
- Si verifica la indipendenza condizionale del nuovo nodo dai precedenti;
- Conseguentemente si scrivono gli archi.

$$P(X_i | X_{i-1}, \dots, X_1) = P(x_i | Parents(X_i))$$

La probabilità di X_i condizionata da tutti gli altri nodi è la probabilità di X_i condizionata dai nodi genitori.

Esempio:

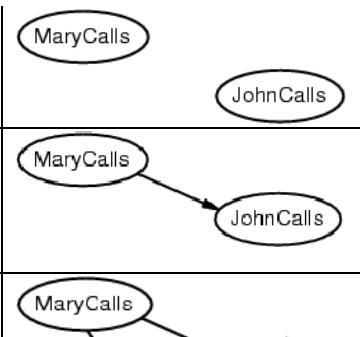
Ordine (diagnostico) dei nodi (si parte prima dagli effetti e poi le cause): M, J, A, B, E

- Inserisco il nodo $M \rightarrow$ non ha parents
- Inserisco il nodo J

Jhon dipende da Mary? - $P(J | M) = P(J)?$

- **No**, J non è condizionalmente indipendente da M.

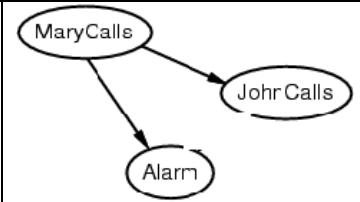
Se Mary chiama significa che probabilmente c'è stato un allarme e quindi è più probabile che anche John chiami.



Inserisco il nodo A:

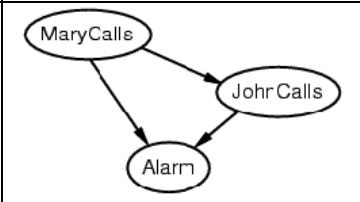
- $P(A | J, M) = P(A | J)?$ **No**

La probabilità che ci sia un allarme dipende dal fatto che Mary abbia chiamato.



- $P(A | J, M) = P(A)?$ **No**

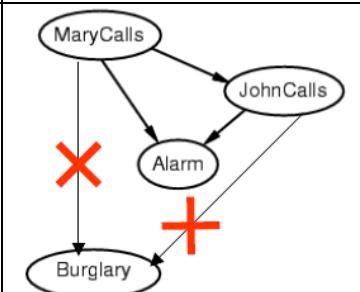
Se sia Mary che John chiamano, la probabilità che ci sia un allarme cambia.



Inserisco il nodo B:

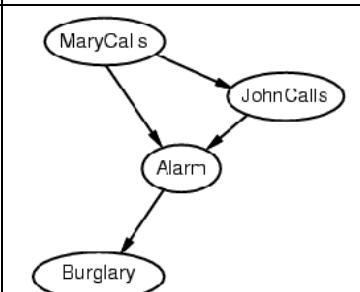
- $P(B | A, J, M) = P(B | A)?$ **Yes**

La probabilità di *Burglary* avendo come evidenza lo stato di *Alarm* non cambia se si aggiungono come evidenze lo stato di *MaryCalls* e *JohnCalls*.



- $P(B | A, J, M) = P(B)?$ **No**

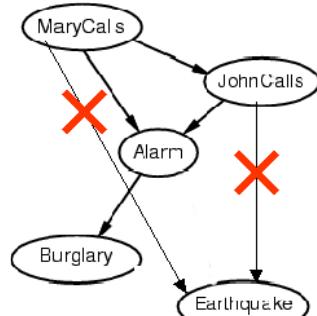
La probabilità di *Burglary* cambia se si ha l'evidenza dello stato di *Alarm*. Siccome il furto dipende dall'allarme.



Inserisco il nodo E:

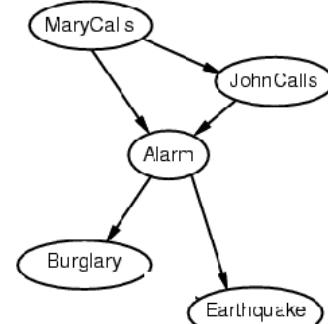
- $P(E | B, A, J, M) = P(E | A, B)$? Yes

Earthquake non cambia conoscendo lo stato delle chiamate di Mary e John.



- $P(E | B, A, J, M) = P(E | B)$? No

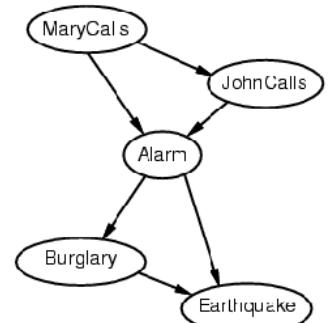
Se ho evidenza ci sia un *Alarm* la probabilità che ci sia un *Earthquake* cambia.



Inserisco il nodo E:

- $P(E | B, A, J, M) = P(E | A)$? No

Se so che c'è stato un *Burglary* in presenza di allarme, *Burglary* può spiegare l'allarme e quindi influenzare la probabilità di *Earthquake*.



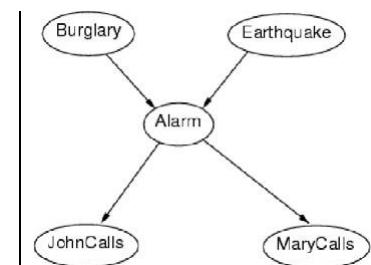
Risultato finale con l'ordinamento:

M, J, A, B, E

sintomo, sintomo, causa intermedia, causa iniziale, causa iniziale

Un numero maggiore di link rispetto a:

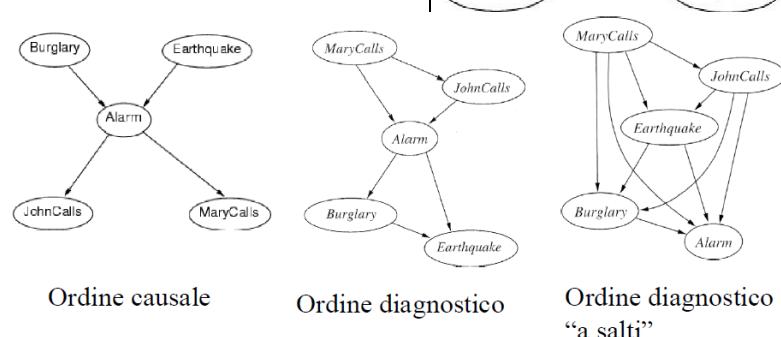
Alcuni link sono innaturali e di difficile stima (Probabilità di *Earthquake*, dato *Alarm* e *Burglary*)



Secondo modo: costruzione di una rete causale:

- Si definiscono le cause prime;
- Si connettono le cause prime agli effetti diretti;
- Si procede allo stesso modo interpretando gli effetti come nuove cause.

L'ordine di scelta causale porta a costruire reti ottime.

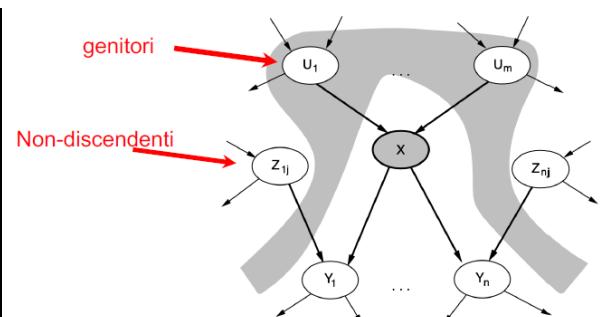


SEMANTICA LOCALE:

L'indipendenza tra nodi può essere codificata tramite una **semantica locale**. In questo modo definendo una semantica locale per ogni nodo possiamo definire la semantica per l'intera rete (global semantics).

Semantica locale (topologica): ogni nodo è condizionatamente indipendente dai suoi non-descendenti dati i suoi genitori. È condizionato solo dai valori che assumono i suoi genitori.

In altre parole, ogni nodo è condizionatamente indipendente da tutti gli altri nodi data la sua **Coperta di Markov**, ovvero genitori + figli + genitori di figli. Tutti questi nodi non influenzano x.



INFERNZA ESATTA SU RETI BAYESIANE:

Vogliamo usare una rete bayesiana per fare inferenza, facendola usare ad un agente e quando questo fa inferenza non è interessato solo a sapere se un evento può accadere oppure no, ma ha anche delle conoscenze, quindi la risposta che vuole ottenere è in funzione di quello che lui già sa.

Con l'inferenza esatta si vuole calcolare la probabilità CONDIZIONATA (a posteriori) di un insieme di **query variables**, dato un evento (un assegnamento di valori ad un insieme di **evidence variables**).

Utilizzando la product rule:

X variabile query, **E** variabili evidenza, **e** valori osservati per **E**, **Y** variabili Hidden i cui valori sono **y**:

Vogliamo rispondere alla query:

$$P(X | e)$$

Distribuzione di probabilità di **X** condizionata da **e**

La formula diventa:

$$P(X | e) = \alpha P(X, e) = \alpha \sum_y P(X, e, y)$$

Somma su tutti i possibili valori **y** delle variabili non osservate

In una rete bayesiana i termini $P(X, e, y)$ (probabilità dell'EVENTO ATOMICO) possono essere scritti come prodotti di probabilità condizionate prese dalla rete (CPT):

$$P(x_1, \dots, x_n) = \prod_{i=1}^n P(x_i | Parents(X_i))$$

Esempio:

Query: $P(Burglary | JohnCalls=true, MaryCalls=true)$

Inferenza in verso diagnostico utilizzando CPT scritte in verso causale (Teorema di Bayes):

$$P(B | j, m) = \alpha P(B, j, m) = \alpha \sum_e \sum_a P(B, j, m, e, a)$$

Somma su tutti i possibili valori y delle variabili hidden e, a

Calcolo per *Burglary = true* utilizzando: $P(x_i | Parents(X_i))$

$$P(b | j, m) = \alpha \sum_e \sum_a P(b) P(j|a) P(m|a) P(e) P(a|b, e)$$

Devo sommare 4 termini (2*2 combinazioni di valori di **e** ed **a**).

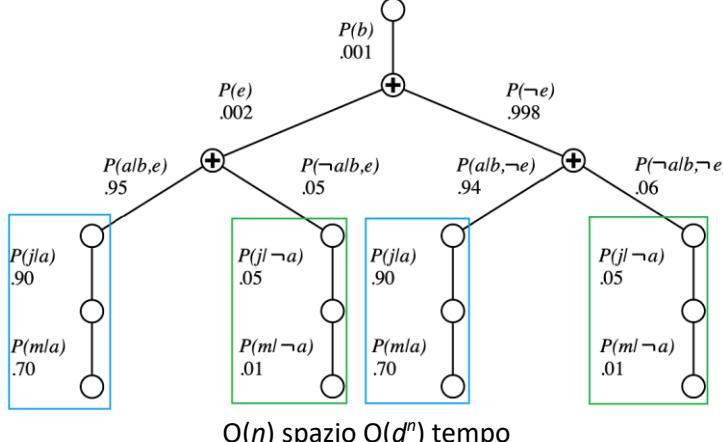
Ogni termine è un prodotto di 5 numeri. Con n variabili booleane $O(n2^n)$.

$$P(b | j, m) = \alpha \sum_e P(e) \sum_a P(j|a) P(m|a) P(a|b, e)$$

$$P(B | j, m) = \alpha <0,00059224, 0,0014919>$$

Normalizzando a 1 con α : $<0,284, 0,716>$

Graficamente viene una visita in profondità:



L'algoritmo effettua dei calcoli ripetitivi, come si può notare con le "foglie". Si possono effettuare delle ottimizzazioni.

INFERNZA ATTRAVERSO L'ELIMINAZIONE DI VARIABILI:

eseguire somme da destra a sinistra, memorizzando risultati intermedi (**fattori**) per evitare la ricomputazione:

$$\begin{aligned} P(B | j, m) &= \alpha \underbrace{P(B)}_{B} \sum_e \underbrace{P(e)}_{E} \sum_a \underbrace{P(a | B, e)}_{A} \underbrace{P(j | a)}_{J} \underbrace{P(m | a)}_{M} \\ &= \alpha P(B) \sum_e P(e) \sum_a P(a | B, e) P(j | a) f_M(a) \\ &= \alpha P(B) \sum_e P(e) \sum_a P(a | B, e) f_J(a) f_M(a) \\ &= \alpha P(B) \sum_e P(e) \sum_a f_A(a, b, e) f_J(a) f_M(a) \\ &= \alpha P(B) \sum_e P(e) f_{\bar{A}JM}(b, e) \quad (\text{sum out } A) \\ &= \alpha P(B) f_{\bar{E}\bar{A}JM}(b) \quad (\text{sum out } E) \\ &= \alpha f_B(b) \times f_{\bar{E}\bar{A}JM}(b) \end{aligned}$$

Vettore di due elementi
 $P(m|a)$ e $P(m| \neg a)$

function ENUMERATION-ASK(X, e, bn) returns a distribution over X
inputs: X , the query variable
 e , observed values for variables E
 bn , a Bayesian network with variables $\{X\} \cup E \cup Y$

$Q(X) \leftarrow$ a distribution over X , initially empty

for each value x_i of X do

extend e with value x_i for E

$Q(x_i) \leftarrow$ ENUMERATE-ALL(VARS[bn], e)

return NORMALIZE($Q(X)$)

function ENUMERATE-ALL($vars, e$) returns a real number
if EMPTY?($vars$) then return 1.0
 $Y \leftarrow$ FIRST($vars$)
if Y has value y in e
then return $P(y | Pa(Y)) \times$ ENUMERATE-ALL(REST($vars$), e)
else return $\sum_y P(y | Pa(Y)) \times$ ENUMERATE-ALL(REST($vars$), e_y)
where e_y is e extended with $Y = y$

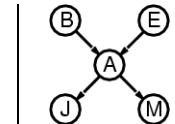
VARIABILI IRRILEVANTI:

Per velocizzare ancora di più possiamo verificare se ci sono variabili irrilevanti, cioè i loro valori di probabilità non influenzano il risultato finale.

Identifichiamo tale variabile analizzando la rete bayesiana e quelle irrilevanti sono quelle che non fanno parte degli antenati né della query né delle evidenze. Ad esempio, se ho una query su J, tutti i dati su M non mi interessano.

Consideriamo la query $P(JohnCalls | Burglary=true)$

$$P(J|b) = \alpha P(b) \sum_e P(e) \sum_a P(a|b, e) P(J|a) \sum_m P(m|a)$$



La somma su m è 1; M è irrilevante per la query

Teorema 1: Y è irrilevante a meno che:

$$Y \in Ancestors(\{X\} \cup \mathbf{E})$$

Here, $X = JohnCalls$, $\mathbf{E} = \{Burglary\}$, and
 $Ancestors(\{X\} \cup \mathbf{E}) = \{Alarm, Earthquake\}$
so $MaryCalls$ is irrelevant

Un algoritmo di eliminazione variabili può quindi eliminare tutte queste variabili prima di valutare la query.

COMPLESSITÀ DELL'INFERENZA ESATTA:

Reti singolarmente connesse (o *polialberi*):

- Due nodi qualsiasi sono collegati al massimo da un percorso non orientato
- Il costo in tempo e spazio dell'eliminazione variabili è $O(d^k n)$

Reti a connessioni multiple:

- Può ridursi a 3SAT \rightarrow NP-Hard
- Equivalenti a contare modelli 3SAT \rightarrow NP-Hard

Fare **clustering** significa trasformare una rete bayesiana classica in un polialbero, accorpando più nodi ed ovviamente la tabella di CPT diventa complicata.

INFERENZA MEDIANTE SIMULAZIONE STOCASTICA:

Dobbiamo usare **metodi approssimati di inferenza** perché l'inferenza esatta su reti Bayesiane è **NP-Hard**.

Per l'inferenza approssimata dobbiamo stimare i valori di probabilità attraverso delle simulazioni (o campionamento):

- Generare N campioni da una distribuzione di campionamento S ;
- Calcolare una probabilità a posteriori approssimativa \hat{P} (**P segnato**);
- Mostrare che converge alla probabilità reale P , così da poter rispondere alla query con \hat{P} .

Tecniche che vedremo (di complessità crescente) per ricavarci \hat{P} :

- Campionamento da una rete vuota**;
- Campionamento di rigetto**: respingere i campioni in disaccordo con le evidenze;
- Pesatura di verosimiglianza**: utilizzare le evidenze per ponderare i campioni;
- Catena di Markov Monte Carlo (MCMC)**: campione da un processo stocastico la cui distribuzione stazionaria è la vera probabilità a posteriori.

CAMPIONAMENTO DA UNA RETE VUOTA:

Partendo da una rete bayesiana, vogliamo ricavarci la probabilità di un determinato evento atomico. Vogliamo poter calcolare la distribuzione di probabilità congiunta. L'algoritmo è il seguente:

Prende in input la rete bayesiana, per un dato evento effettua un campionamento per tutti i valori di questo evento. Per far sì che il valore sia corretto, andiamo ad associare dei valori che rispettano ciò che è stato definito nella rete bayesiana; quindi, tenendo in considerazione la probabilità di X_i dati i suoi parenti (il valore associato dipende dai valori dei genitori). Il campione viene creato partendo dalle variabili causali (quelle che non hanno genitori, assegnando un valore random).



```

function PRIOR-SAMPLE(bn) returns an event sampled from bn
  inputs: bn, a belief network specifying joint distribution  $\mathbf{P}(X_1, \dots, X_n)$ 
  x  $\leftarrow$  an event with  $n$  elements
  for  $i = 1$  to  $n$  do
     $x_i \leftarrow$  a random sample from  $\mathbf{P}(X_i | parents(X_i))$ 
      given the values of  $Parents(X_i)$  in x
  return x

```

Esempio:

Consideriamo la rete bayesiana a lato, abbiamo che l'erba bagnata (Wet Grass) è l'effetto finale che può essere causato da due possibili eventi, ovvero ha piovuto (Rain) o l'irrigatore è stato acceso (Sprinkler), mentre l'ultimo evento è nuvoloso (Cloudy) che influenza i due eventi precedenti siccome se il tempo è nuvoloso, l'irrigatore non viene acceso siccome dovrebbe piovere (ad esempio 80% pioverà).

Una volta definita la rete facciamo partire l'algoritmo, iniziando per ordine topologico, ovvero da Cloudy. A questa variabile viene associato un valore di probabilità che dipenderà da $P(C)$ (siccome non ha genitori). Il valore lo si sceglie in base ad una distribuzione di probabilità, in questo caso i parents sono equiprobabili perché abbiamo il 50% e sceglierà a random se è vero o falso. Supponiamo che viene associato a Cloudy il valore TRUE.

A questo punto si passa ai figli, andando a campionare il valore della variabile Sprinkler. In questo caso, il valore associato ad esso è probabilmente FALSE perché, poiché la variabile C è TRUE, abbiamo il 10% di volte che S è TRUE. Mentre il valore associato alla variabile R è probabilmente TRUE perché nell'80% dei casi lo è quando Cloudy è TRUE.

Infine, per generare il valore da associare alla variabile W bisogna tenere in considerazione i suoi genitori ($P(X_i | Parents(X_i))$), in tal caso sarà TRUE, andando a vedere in tabella abbiamo che lo è il 90% delle volte.

NOTA: L'algoritmo produce solo un evento, pertanto verrà eseguito più volte.

L'algoritmo verrà lanciato un certo numero di volte, una volta avuti i campioni andremo a contare quante volte un evento è capitato.

Facendo crescere il numero di campioni, il valore probabilistico che andremo ad ottenere con il rapporto (numero campioni in cui è vero/numero totale di campioni) coinciderà con la probabilità che quell'evento accade (abbiamo detto essere P).

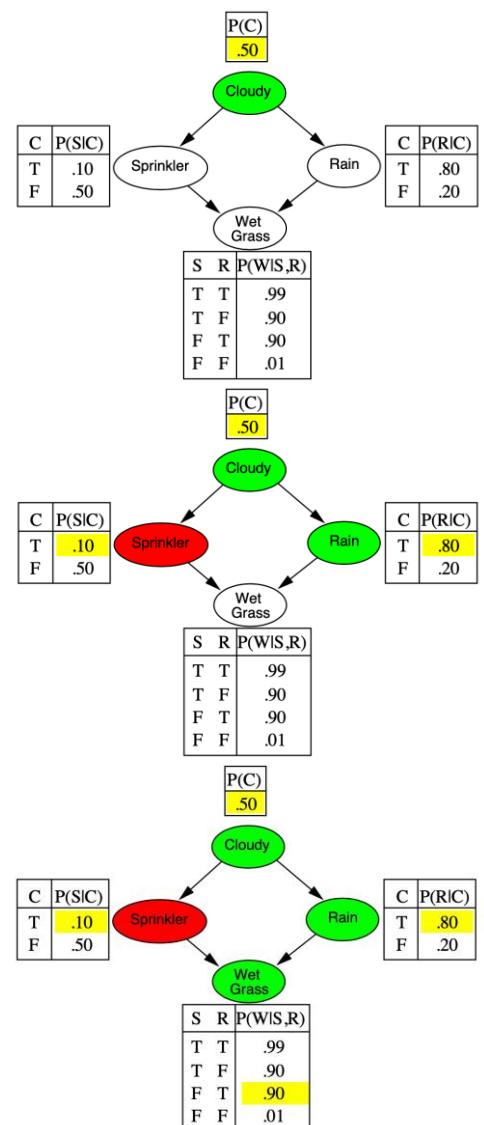
Fondamentalmente, l'algoritmo produce delle probabilità consistenti.

CAMPIONAMENTO DI RIGETTO:

Questo algoritmo è semplice ma inefficiente, rispetto al precedente, teniamo in considerazione delle **evidenze**.

In questo caso vogliamo stimare p di x dato e , quindi va ad applicare l'algoritmo di campionamento a rete vuota N volte. Successivamente, tra tutti i campioni considera solo quelli **utili** ovvero che soddisfano le evidenze mentre quelli che non le soddisfano non li consideriamo perché non sono utili per la probabilità che sto cercando di calcolare.

Infine, va a normalizzare sul numero di campioni utili.



Probabilità che PriorSample generi un evento particolare

$$SPS(x_1 \dots x_n) = \prod_{i=1}^n P(x_i | parents(X_i)) = P(x_1 \dots x_n)$$

cioè la vera probabilità a priori (distr. congiunta rete Bayesiana)

$$\text{Ad esempio } SPS(t, f, t, t) = 0.5 \times 0.9 \times 0.8 \times 0.9 = 0.324 = P(t, f, t, t)$$

Sia $N_{PS}(x_1 \dots x_n)$ numeri di eventi in cui x_1, \dots, x_n è generato

- Esempio: se ho 1000 campioni per la rete WetGrass, e in 511 Rain=true, allora $P(\text{Rain}=true) = 0,511$.

$$\begin{aligned} \text{Allora abbiamo } \lim_{N \rightarrow \infty} \hat{P}(x_1, \dots, x_n) &= \lim_{N \rightarrow \infty} N_{PS}(x_1, \dots, x_n)/N \\ &= SPS(x_1, \dots, x_n) \\ &= P(x_1 \dots x_n) \end{aligned}$$

In altre parole, le stime derivate da PriorSample sono CONSISTENTI

$$\hat{P}(x_1, \dots, x_n) \approx P(x_1 \dots x_n)$$

$\hat{P}(X|e)$ stimato da campioni concordanti con l'evidenza e

```
function REJECTION-SAMPLING( $X, e, bn, N$ ) returns an estimate of  $P(X|e)$ 
  local variables:  $N$ , a vector of counts over  $X$ , initially zero
  for  $j = 1$  to  $N$  do
     $x \leftarrow \text{PRIOR-SAMPLE}(bn)$ 
    if  $x$  is consistent with  $e$  then
       $N[x] \leftarrow N[x]+1$  where  $x$  is the value of  $X$  in  $x$ 
  return NORMALIZE( $N[X]$ )
```

Esempio:

Ad esempio, stima $P(Rain|Sprinkler = \text{true})$ usando 100 campioni

- ▶ 27 campioni hanno $Sprinkler = \text{true}$
- ▶ di questi 8 hanno $Rain = \text{true}$ e 19 hanno $Rain = \text{false}$

$$\hat{P}(Rain|Sprinkler = \text{true}) = \text{NORMALIZE}(\langle 8, 19 \rangle) = \langle 0.296, 0.704 \rangle \quad (\text{la normalizzazione è } 8/27 \text{ e } 19/27)$$

(simile a una procedura di stima empirica nel mondo reale)

Quindi il campionamento di rigetto restituisce stime a posteriori **consistenti**. Il problema è che il tutto è irrimediabilmente costoso se $P(e)$ è piccolo (rifiuterà moltissimi campioni). Infatti, $P(e)$ diminuisce esponenzialmente al crescere del numero di variabili evidenza. Più evidenze ci sono più l'algoritmo è inefficiente.

$$\begin{aligned}\hat{P}(X|e) &= \alpha N_{PS}(X, e) && (\text{algorithm defn.}) \\ &= N_{PS}(X, e)/N_{PS}(e) && (\text{normalized by } N_{PS}(e)) \\ &\approx P(X, e)/P(e) && (\text{property of PRIORSAMPLE}) \\ &= P(X|e) && (\text{defn. of conditional probability})\end{aligned}$$

PESATURA DI VERO SIMIGLIANZA:

L'idea è di fissare le variabili evidenza, campionare solo le variabili non di evidenza e pesare ogni campione in base alla probabilità che si accordi con le evidenze (gli eventi in cui è improbabile che le prove siano verificate devono pesare di meno nel conteggio). L'algoritmo produce il campione insieme ad un peso e va a considerare tutte le variabili della rete bayesiana e se X_i rispetta quella condizione (ha valore in e), vuol dire che è una variabile di evidenza e quindi devo vedere quanto il valore che ho in X , si accorda con le variabili che ho assegnato nella rete bayesiana quindi vado ad aggiornare il peso.

Nel caso in cui non lo è, faccio il campionamento secondo la probabilità condizionata rispetto ai suoi genitori.

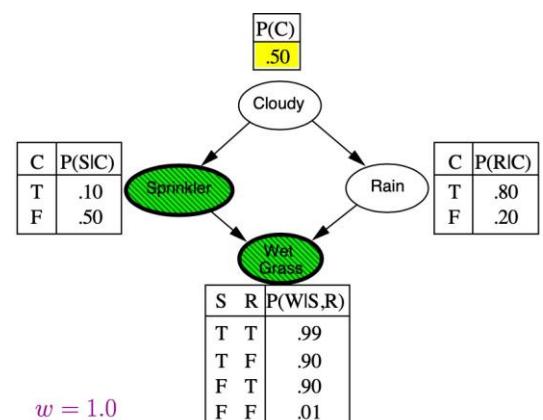
```
function LIKELIHOOD-WEIGHTING( $X, e, bn, N$ ) returns an estimate of  $P(X|e)$ 
local variables:  $\mathbf{W}$ , a vector of weighted counts over  $X$ , initially zero
for  $j = 1$  to  $N$  do
     $\mathbf{x}, w \leftarrow \text{WEIGHTED-SAMPLE}(bn)$ 
     $\mathbf{W}[x] \leftarrow \mathbf{W}[x] + w$  where  $x$  is the value of  $X$  in  $\mathbf{x}$ 
return NORMALIZE( $\mathbf{W}[X]$ )
```

```
function WEIGHTED-SAMPLE( $bn, e$ ) returns an event and a weight
 $\mathbf{x} \leftarrow$  an event with  $n$  elements;  $w \leftarrow 1$ 
for  $i = 1$  to  $n$  do
    if  $X_i$  has a value  $x_i$  in  $e$ 
        then  $w \leftarrow w \times P(X_i = x_i | \text{parents}(X_i))$ 
        else  $x_i \leftarrow$  a random sample from  $P(X_i | \text{parents}(X_i))$ 
return  $\mathbf{x}, w$ 
```

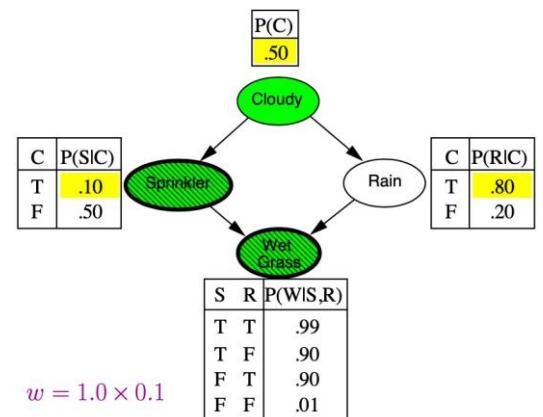
Esempio:

Supponiamo che diamo all'algoritmo la rete a lato, con la rispettiva Query= $P(\text{Rain} | \text{Sprinkler}=\text{true}, \text{WetGrass}=\text{true})$. Alcuni valori di verità sono già fissati e l'algoritmo non li può cambiare, pertanto, l'algoritmo dovrà restituire un vettore che darà un valore a C e R, ed un altro valore che rappresenta il **peso w** che mi dice quanto è importante il campione dei quattro valori di verità.

L'algoritmo parte sempre dalla radice, prende X_1 che sarebbe Cloudy ma essa non è una variabile di evidenza, quindi, va a campionare, trovandoci nello stesso stato del precedente esempio, ovvero ha il 50% di avere TRUE o FALSE.

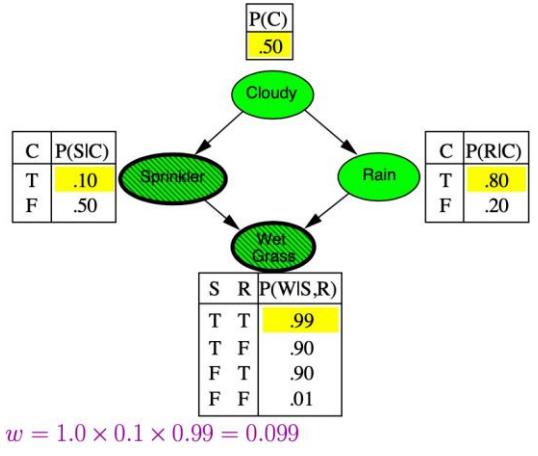


Supponiamo che Cloudy è TRUE e prendiamo la seconda variabile (Sprinkler), andiamo nella funzione dove chiede se X è una variabile di evidenza e in questo caso è sì; pertanto, bisogna aggiornare il peso che sarà w per la probabilità della variabile S data i suoi genitori C. Pertanto, il calcolo sarà $w=1.0 \times 0.1$.



Adesso si prende Rain, esso non è in evidenza quindi usiamo sempre la probabilità condizionata data dai genitori, quindi ci troviamo che 80% è TRUE e pertanto glie lo assegniamo.

L'ultima variabile W è in evidenza, pertanto si prende il valore di probabilità condizionato da due TRUE, questo poi sarà moltiplicato per il peso ed il peso totale sarà $w=1.0 \times 0.1 \times 0.99=0.099$.



ANALISI PESATURA DI VERO SIMIGLIANZA:

L'algoritmo calcola due cose, ovvero il campionamento delle variabili non di evidenza e il calcolo del peso.

La **probabilità di campionamento** per WeightedSample è:

$$S_{ws}(z,e) = \prod_{i=1}^l P(z_i | parents(Z_i))$$

Dove **z** sono tutte le variabili tranne quelle di evidenza, mentre **parents(Z_i)** può includere sia variabili nascoste che variabili di evidenza.

Bisogna prestare attenzione alle evidenze solo negli **antenati** → ignorando le evidenze che non sono presenti tra gli antenati di Z_i.

Il **peso** per un dato campione **z** ed evidenze **e** è:

$$w(z,e) = \prod_{i=1}^m P(e_i | parents(E_i))$$

quindi la probabilità pesata di un campione è:

$$SWS(z,e)w(z,e) = \prod_{i=1}^l P(z_i | parents(Z_i)) \prod_{i=1}^m P(e_i | parents(E_i)) = P(z,e)$$

La quale coincide con la probabilità dell'evento atomico, quindi la probabilità pesata restituisce stime **consistenti** ma le prestazioni peggiorano ancora con molte variabili di evidenza perché alcuni campioni hanno quasi tutto il peso totale.

12. APPRENDIMENTO TRAMITE OSSERVAZIONI

Un aspetto importante degli agenti è dato dalla capacità di apprendere. Fino ad ora abbiamo visto che l'agente percepisce informazioni tramite sensori e la decisione presa dipende da ciò che c'è dietro. Quello che vedremo ora, sarà la capacità dell'agente di sfruttare le informazioni che riceve dall'ambiente, per capire se in futuro potrà cambiare le sue decisioni.

L'apprendimento è **essenziale** in ambienti sconosciuti perché rende l'agente intelligente, in maniera tale che sia in grado di adattarsi all'ambiente che lo circonda.

L'apprendimento è utile come metodo di costruzione del sistema, esporre l'agente alla realtà piuttosto che cercare di descriverla, siccome la conoscenza da dover inserire nel sistema è troppo ampia e complessa da specificare.

L'apprendimento modifica i meccanismi di decisione dell'agente al fine di migliorarne le prestazioni.

Lo schema riassume l'architettura di un agente capace di apprendere:

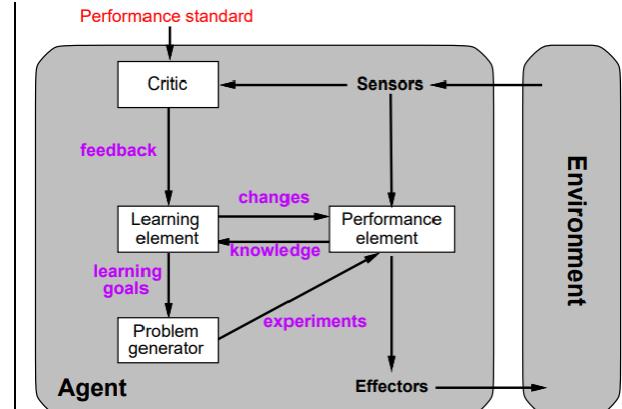
I **sensori** e **attuatori** permettono di interagire con l'ambiente.

Il **performance element** è l'agente che riceve le percezioni dai sensori e decide la migliore azione.

Vogliamo progettare quindi il **Learning element**, che rappresenta la componente chiave e quella che definiremo, perché è il modulo che va a cambiare il comportamento dell'agente; affinché l'agente si adatti in base alle sue performance.

Il **feedback** indica come si sta comportando l'agente, ci dicono se l'azione effettuata ha avuto esito positivo o meno.

Il **problem generator** permette all'agente di esplorare scenari che non erano stati già visitati.



Il **learning element** dipende:

- Dal tipo di **Performance element** usato, cioè da che tipo di agente usiamo (es. agenti basati su algoritmi di ricerca, basati su logica, basati su probabilità);
- Da quali componenti del **Performance element** devono essere appresi (dipende dal tipo di Performance element usato);
- Da come è **rappresentato** quel particolare componente funzionale;
- Dal tipo di **feedback** disponibile per l'apprendimento.

Scenari d'esempio:

Performance element	Componente	Rappresentazione	Feedback
Alpha–beta pruning search	Eval. Function	Weighted linear function	Win/loss
Logical agent	Transition model	Successor–state axioms	Outcome
Utility-based agent	Transition model	Rete Bayesiana	Outcome
Simple reflex agent	Percept–action function	Rete Neurale	Correct action

Supponiamo che l'agente usi l'algoritmo di **Alpha-Beta pruning** (ricerca con avversari). La componente usata è la **funzione di valutazione** e cambiandola si ottiene un comportamento differente dall'algoritmo, ad esempio, cambiando i pesi della funzione così da fare scelte differenti. La funzione di valutazione è **rappresentata** come funzione lineare pesata. Infine, il tipo di **feedback** che si ha è vittoria o sconfitta.

Due **tipologie di apprendimento**:

- **Supervisionato**: risposte corrette per ogni istanza;
- **Rinforzato**: ricompense occasionali.

TIPI DI APPRENDIMENTO (FEEDBACK):

Apprendimento supervisionato:

- Apprendere una funzione da esempi di input/output, siccome abbiamo a disposizione questi;
- In ambienti completamente osservabili può vedere i risultati delle azioni ed imparare a predirli;
- In ambienti parzialmente osservabili gli effetti possono essere invisibili.

Apprendimento non supervisionato:

- Imparare a riconoscere pattern nell'input senza alcuna indicazione dell'output.

Apprendimento per rinforzo:

- L'agente apprende basandosi sul rinforzo, tramite ricompense.

12.1. APPRENDIMENTO INDUTTIVO

L'agente vuole capire se a partire da un insieme di dati (input e output), è possibile capire qual è il nesso tra input e output così da poter decidere l'azione da fare in base al prossimo input.

L'**apprendimento induttivo** rappresenta la forma più semplice di apprendimento, apprendere una funzione dagli esempi; **f** rappresenta la funzione obiettivo, l'input è il vettore **x** mentre l'output **f(x)** è il risultato di una funzione f.

Un esempio è la coppia $x, f(x)$, es. $\begin{pmatrix} \begin{array}{c|c|c} \text{input} & \text{output} \\ \hline O & O & X \\ \hline X & & X \end{array} & , & \begin{array}{c} \text{configurazione} \\ \hline +1 \end{array} \end{pmatrix}$

Ciò che vogliamo fare è ricavare il legame tra input e output; quindi, vogliamo cercare un'ipotesi **h** tale che approssimi f ($h \approx f$) dato un training set di esempi.

Il problema è che noi abbiamo solo una porzione di f, non abbiamo tutti i dati possibili ma solo un sottoinsieme quindi potremmo anche trovare una funzione $h(x) = f(x)$ su tutte le x che abbiamo come input ma in ogni caso è una funzione differente perché non conosciamo il comportamento di f su altri campioni.

Questo è un modello altamente semplificato di apprendimento reale, infatti:

- Ignora la conoscenza precedente
- Assume la presenza di un ambiente deterministico ed osservabile
- Assume che siano forniti degli esempi
- Assume che l'agente voglia apprendere f perché?

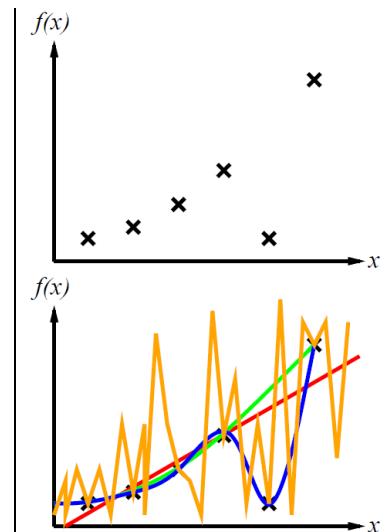
CURVA FITTING:

Possiamo considerare **ogni coppia (x-f(x))** come punti nel piano cartesiano.

L'obiettivo è trovare la funzione **h** che concorda con **f**, sui dati del training set.

Questo problema si chiama **CURVE FITTING**, cioè trovare la migliore **h** che mi vada a coprire tutte le **x** che sono sul piano.

Ogni retta/curva nel piano rappresenta una possibile approssimazione di f, dove alcuni punti del piano sono più vicini o più lontani rispetto ad una determinata curva.



Quale **h** scegliere? Si deve preferire l'ipotesi più semplice consistente con i dati (**Rasoio di Occam**).

Un altro concetto che influenza questo problema è il **dominio delle funzioni** (es. polinomi, funzioni trigonometriche).

Dal punto di vista dell'apprendimento si dice che un problema di apprendimento è **realizzabile** se lo spazio delle ipotesi contiene la funzione reale. Questo perché la funzione h si prende dallo spazio delle ipotesi, ma se all'interno dello spazio non c'è f si è certi che non si può definire una funzione consistente e si dice **non realizzabile**. Non si sa in anticipo se è non realizzabile siccome la f non la conosciamo, quello che si fa è trovare un compromesso tra complessità del problema e la dimensione dello spazio di ipotesi, ad esempio si considera un h non complesso altrimenti il problema non viene risolto.

Esempio (Training set – problema di classificazione):

Considerando il seguente training set con 12 campioni ed ognuno ha un vettore di 10 attributi (discreti) con un valore. Tali dati rappresentano situazioni dove converrebbe aspettare/non aspettare per un tavolo.

Example	Attributes										Target WillWait
	Alt	Bar	Fri	Hun	Pat	Price	Rain	Res	Type	Est	
X ₁	T	F	F	T	Some	\$\$\$	F	T	French	0–10	T
X ₂	T	F	F	T	Full	\$	F	F	Thai	30–60	F
X ₃	F	T	F	F	Some	\$	F	F	Burger	0–10	T
X ₄	T	F	T	T	Full	\$	F	F	Thai	10–30	T
X ₅	T	F	T	F	Full	\$\$\$	F	T	French	>60	F
X ₆	F	T	F	T	Some	\$\$	T	T	Italian	0–10	T
X ₇	F	T	F	F	None	\$	T	F	Burger	0–10	F
X ₈	F	F	F	T	Some	\$\$	T	T	Thai	0–10	T
X ₉	F	T	T	F	Full	\$	T	F	Burger	>60	F
X ₁₀	T	T	T	T	Full	\$\$\$	F	T	Italian	10–30	F
X ₁₁	F	F	F	F	None	\$	F	F	Thai	0–10	F
X ₁₂	T	T	T	T	Full	\$	F	F	Burger	30–60	T

Caratteristiche ristorante:

1. **Alternate**: whether there is a suitable alternative restaurant nearby.
2. **Bar**: whether the restaurant has a comfortable bar area to wait in.
3. **Fri/Sat**: true on Fridays and Saturdays.
4. **Hungry**: whether we are hungry.
5. **Patrons**: how many people are in the restaurant (values are *None*, *Some*, and *Full*).
6. **Price**: the restaurant's price range (\$, \$\$, \$\$\$).
7. **Raining**: whether it is raining outside.
8. **Reservation**: whether we made a reservation.
9. **Type**: the kind of restaurant (French, Italian, Thai, or burger).
10. **WaitEstimate**: the wait estimated by the host (0–10 minutes, 10–30, 30–60, or >60).

Sulla base di queste premesse vogliamo scoprire il criterio (la funzione) che modella il comportamento della persona.

Possiamo andare a riorganizzare i campioni come segue:

cioè ogni campione è costituito da un vettore x_i a 10 dimensioni di valori discreti delle caratteristiche e un'etichetta di destinazione y_i che è booleana.

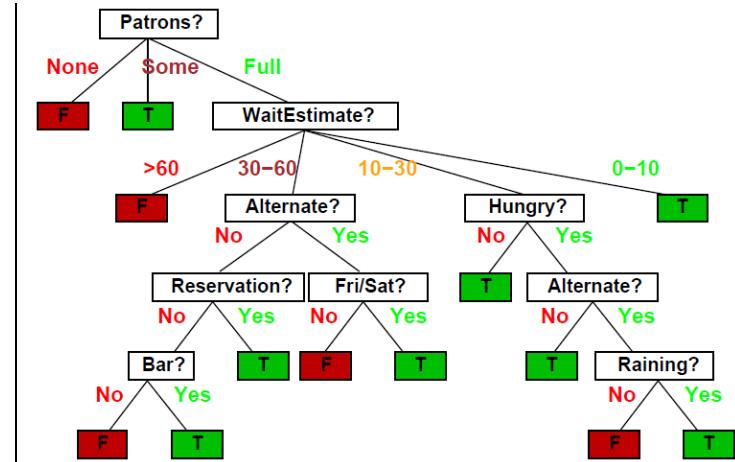
Possiamo rappresentare questo training set tramite gli alberi di decisione.

$x_1 = \left\{ \begin{array}{l} T \\ F \\ F \\ T \\ Some \\ $$$ \\ F \\ T \\ French \\ 0-10 \end{array} \right\}, y_1 = T$	$x_2 = \left\{ \begin{array}{l} T \\ F \\ F \\ T \\ Full \\ \$ \\ F \\ F \\ Thai \\ 30-60 \end{array} \right\}, y_2 = F$
sample 1	sample 2

ALBERO DI DECISIONE:

Un **albero di decisione** rappresenta una funzione che prende in input un vettore di valori e restituisce una **decisione**, un singolo valore. È una possibile rappresentazione per le ipotesi dove i **nodi** sono gli **attributi del dataset** mentre le **foglie** sono le **classificazioni** (vero/falso nell'esempio). Il nodo che rappresenta un attributo ha come **archi uscenti** i possibili **valori** di quell'attributo. Prendendo un campione e seguendo il percorso dalla radice alla foglia, riesco a classificarlo.

L'albero di decisione rappresenta la tabella dell'esempio precedente:



Gli alberi di decisione possono esprimere qualsiasi funzione degli attributi in input.

Un albero definisce un **predicato obiettivo** (*WillWait*):

$$\forall s \ WillWait(s) \Leftrightarrow (P_1(s) \vee P_2(s) \vee \dots \vee P_n(s)) \text{ con } P_i \text{ path per arrivare alla foglia T.}$$

(Il predicato attendere è soddisfatto se e solo se uno dei percorsi da radice a foglia è soddisfatto)

In sostanza, esiste un albero decisionale coerente per qualsiasi training set con un path verso la foglia per ogni esempio (a meno che f sia non deterministico in x), ma probabilmente non si generalizzerà a nuovi esempi. È preferibile ricercare alberi di decisione più **compatti**.

A	B	A xor B
F	F	F
F	T	T
T	F	T
T	T	F

```

graph TD
    A[A] -- F --> B1[B]
    A -- T --> B2[B]
    B1 -- F --> L1[F]
    B1 -- T --> L2[T]
    B2 -- F --> L3[T]
    B2 -- T --> L4[F]
  
```

SPAZIO DELLE IPOTESI:

Quanti alberi di decisione si possono creare con n attributi booleani (n colonne e 2 valori)?

- = numero di funzioni booleane;
- = numero di tabelle di verità distinte con 2^n righe e tutte le possibili tabelle sono 2^{2^n} .

Ad esempio, con 6 attributi booleani, ci sono 18,446,744,073,709,551,616 alberi di decisione.

Esempio:

Quante ipotesi puramente congiuntive si possono formulare (*Hungry \wedge $\neg Rain$*)?

Ogni attributo può essere vero, falso o ignorato quindi abbiamo 3^n ipotesi congiuntive distinte.

Uno spazio delle ipotesi più espressivo aumenta la possibilità che la funzione obiettivo possa essere espressa, aumenta il numero di ipotesi consistenti con il training set e può portare a delle predizioni peggiori.

12.2. APPRENDIMENTO DEGLI ALBERI DI DECISIONE

Soluzione banale per la creazione di alberi decisionali è costruire un percorso fino ad una foglia per ogni campione.

- Memorizza semplicemente le osservazioni (non estrae nessun modello dai dati);
- Produce un'ipotesi consistente ma eccessivamente complessa (ricorda il rasoio di Occam);
- È improbabile che si generalizzi bene a nuove osservazioni.

Dovremmo mirare a costruire l'albero più piccolo (*più è piccolo più il senso dei dati viene evidenziato*) che sia coerente con le osservazioni. Un modo per farlo è testare in modo ricorsivo prima gli attributi più importanti.

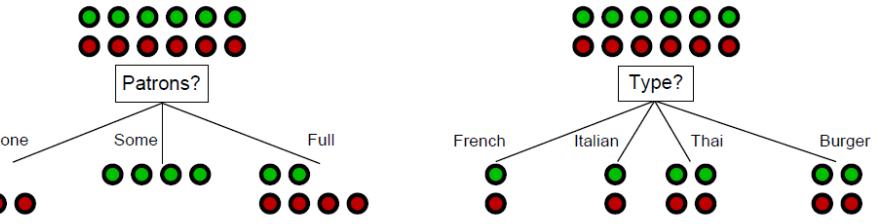
Bisogna capire, quindi, quali sono gli attributi utili per la classificazione.

Esempio:

Meglio testare prima Patrons oppure Type?

Patrons rappresenta una scelta migliore poiché fornisce maggiori informazioni riguardo la classificazione.

L'idea è che un buon attributo divide gli esempi in sottoinsiemi che sono idealmente tutti positivi o tutti negativi.



ALGORITMO DTL (DECISION TREE LEARNING):

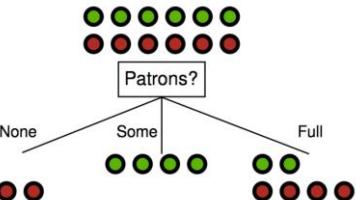
L'**obiettivo** è quello di trovare un albero piccolo e consistente con esempi di training e l'**idea** è di scegliere ricorsivamente l'attributo "più significativo" come radice del (sotto)albero. Se l'algoritmo si trova con tutti campioni della stessa classe, l'algoritmo può terminare e associare a quella foglia la classificazione degli esempi.

La parte fondamentale è scegliere **best** e costruire un albero incentrato su best; quindi, per ogni valore di best va a definire i diversi sottoinsiemi di esempi con valore di best= v_i , e chiama ricorsivamente la funzione, dividendo i campioni in base al valore dell'attributo.

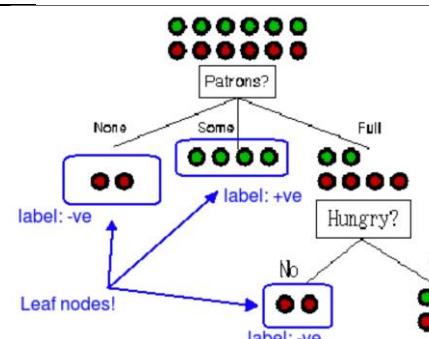
```
function DTL(examples, attributes, default) returns a decision tree
  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attributes is empty then return MODE(examples)
  else
    best ← CHOOSE-ATTRIBUTE(attributes, examples)
    tree ← a new decision tree with root test best
    for each value  $v_i$  of best do
      examples $_i$  ← {elements of examples with best =  $v_i$ }
      subtree ← DTL(examples $_i$ , attributes - best, MODE(examples))
      add a branch to tree with label  $v_i$  and subtree subtree
  return tree
```

Quattro idee alla base dell'algoritmo

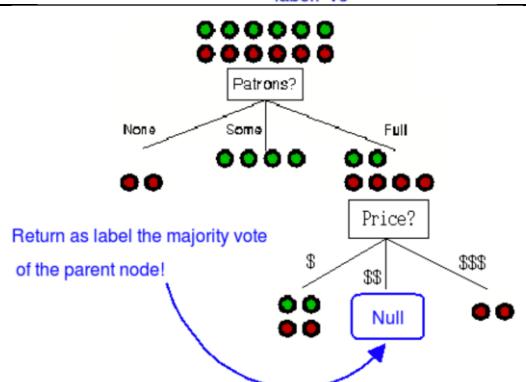
- Se sono presenti campioni positivi e negativi, scegliere l'attributo migliore per dividerli, ad esempio, prova **Patrons** nella radice.



- Se tutti i campioni rimanenti sono tutti positivi o tutti negativi, abbiamo raggiunto un nodo foglia. Assegna l'etichetta come positiva (o negativo), ad esempio →



- Se non sono rimasti campioni, significa che non è stato osservato quel campione. Restituisce un valore predefinito calcolato da classificazione maggioritaria al genitore del nodo, ad esempio, in quel caso verrà associato false, perché è quello maggiore.



- Se non ci sono attributi rimasti, ma ci sono sia campioni positivi che negativi, significa che questi campioni hanno esattamente lo stesso valore delle caratteristiche ma classificazioni diverse. Può succedere perché:
 - Alcuni dati potrebbero essere errati;
 - Gli attributi non forniscono informazioni sufficienti per descrivere completamente la situazione (cioè ci mancano altri attributi utili);
 - Il problema è veramente **non-deterministico**, cioè dati due campioni che descrivono esattamente le stesse condizioni, possiamo prendere diverse decisioni.

Soluzione: chiamarlo nodo foglia e assegnergli il voto di maggioranza come etichetta.

INFORMAZIONE:

L'aspetto chiave dell'algoritmo DTL è il come fa a giudicare un attributo come buono o cattivo. Un modo possibile è **calcolare il contenuto dell'informazione (entropia)** rappresentata da un certo attributo. L'incertezza rappresenta un valore di entropia alto di questa quantità, mentre più il risultato è certo allora più ci vuole una quantità inferiore di informazione per rappresentarlo. Ad esempio, per il nodo R abbiamo:

$$I(R) = \sum_{i=1}^L -P(c_i) \log_2 P(c_i)$$

dove $\{c_1, \dots, c_L\}$ sono le L classi presenti nel nodo e $P(c_i)$ è la probabilità di ottenere la classe c_i al nodo.

Esempio:

Al nodo radice del problema del Ristorante $c_1=True$ e $c_2=False$, ci sono 6 campioni veri e 6 falsi. Pertanto, abbiamo:

$$P(c_1) = \frac{\text{no. of samples} = c_1}{\text{total no. of samples}} = \frac{6}{6+6} = 0.5$$

$$P(c_2) = \frac{\text{no. of samples} = c_2}{\text{total no. of samples}} = \frac{6}{6+6} = 0.5$$

$$I(\text{Root}) = -0.5 \times \log_2 0.5 - 0.5 \times \log_2 0.5 = 1 \text{ bit}$$

La totale incertezza corrisponde ad 1 mentre la totale certezza corrisponderà a 0.

In generale, la quantità di informazioni sarà massima quando tutte le classi sono ugualmente probabili e sono minime quando il nodo è omogeneo (tutti i campioni hanno le stesse etichette).

L'informazione in una risposta quando è a priori $\langle P_1, \dots, P_n \rangle$ è (chiamata anche **entropia** a priori):

$$I(\langle P_1, \dots, P_n \rangle) = \sum_{i=1}^n -P_i \log_2 P_i$$

SCEGLIERE IL MIGLIOR ATTRIBUTO:

Quindi possiamo sfruttare l'entropia per decidere quali attributi conviene selezionare per prima, in altre parole, quali attributi dividono i campioni in classi che richiedono meno informazioni per andare avanti, quindi più certezze verso la classificazione. Più formalmente, un attributo A dividerà i campioni ad un nodo in diversi sottoinsiemi (o nodi figlio) $E_{A=v_1}, \dots, E_{A=v_M}$ dove A ha M valori distinti $\{v_1, \dots, v_m\}$. Generalmente ogni sottoinsieme $E_{A=v_i}$ avrà campioni con etichette diverse; quindi, se andiamo lungo quel ramo avremo bisogno di ulteriori $I(E_{A=v_i})$ bit di informazione.

Esempio:

Nel problema del ristorante, al ramo $\text{Patrons} = \text{Full}$ del nodo radice, abbiamo $c_1 = \text{True}$ con 2 campioni e $c_2 = \text{False}$ con 4 campioni, quindi:

$$I(E_{\text{Patrons}=\text{Full}}) = -\frac{1}{3} \log_2 \frac{1}{3} - \frac{2}{3} \log_2 \frac{2}{3} = 0.9183 \text{ bits}$$

Indichiamo quindi con $P(A=v_i)$ come la probabilità che un campione segua il ramo $A=v_i$. Nel problema del ristorante, al nodo radice abbiamo:

$$\begin{aligned} P(\text{Patrons} = \text{Full}) &= \frac{\text{No. of samples where Patrons} = \text{Full}}{\text{No. of samples at the root node}} \\ &= \frac{6}{12} = 0.5 \end{aligned}$$

Dopo aver testato l'attributo A abbiamo bisogno di un resto di bit per classificare i campioni:

$$\text{Remainder}(A) = \sum_{i=1}^M P(A = v_i) I(E_{A=v_i})$$

Più sono omogenei i campioni, più $I(E_{A=v_i})$ diventa basso, moltiplicandolo per la probabilità di avere quel valore per l'attributo A. Ottengo i bit per continuare il processo di classificazione.

Il **guadagno di informazioni** nel test dell'attributo A è la differenza tra il contenuto dell'informazione originale ed il contenuto delle nuove informazioni, cioè:

$$\begin{aligned} \text{Gain}(A) &= I(R) - \text{Remainder}(A) \\ &= I(R) - \sum_{i=1}^M P(A = v_i) I(E_{A=v_i}) \end{aligned}$$

dove $\{E_{A=v_1}, \dots, E_{A=v_M}\}$ sono i nodi figlio di R dopo il test dell'attributo A.

L'idea chiave alla base della funzione SCEGLI-ATTRIBUTO dell'algoritmo consiste nello scegliere l'attributo che dà il massimo guadagno (GAIN) di informazioni.

Esempio:

Nel problema del ristorante, dobbiamo decidere sulla radice se scegliere *Patrons* o *Type*:

$$\begin{aligned} \text{Gain}(\text{Patrons}) &= 1 - \frac{2}{12}(-\log_2 1) - \frac{4}{12}(-\log_2 1) - \frac{6}{12}(-\frac{2}{6}\log_2 \frac{2}{6} - \frac{4}{6}\log_2 \frac{4}{6}) \\ &\approx 0.5409 \text{ bits.} \end{aligned}$$

Con calcoli simili:

$$\text{Gain}(\text{Type}) = 0$$

Confermando che *Patrons* è un attributo migliore di *Type*. Infatti, alla radice *Patrons* fornisce il maggior guadagno di informazioni.

RIASSUMENDO:

Supponiamo di avere p esempi positivi e n esempi negativi alla radice:

→ $I(p/(p+n), n/(p+n))$ bit richiesti per classificare un nuovo esempio.

Esempio, per 12 ristoranti d'esempio, $p = n = 6$, quindi abbiamo bisogno di 1 bit.

Un attributo divide gli esempi E nei sottoinsiemi E_i , ognuno dei quali (si spera) richiedano meno informazione per completare la classificazione. Sia E_i caratterizzato da esempi positivi p_i e negativi n_i .

→ $I(p_i/(p_i+n_i), n_i/(p_i+n_i))$ bit richiesti per classificare un nuovo esempio.

→ il numero di bit atteso per classificare un esempio su tutti i branch è:

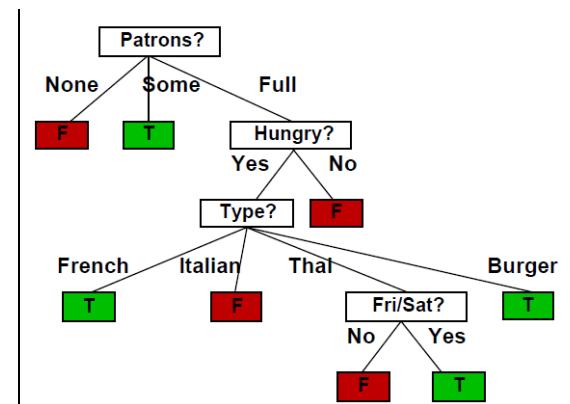
$$\sum_i \frac{p_i+n_i}{p+n} I(p_i/(p_i+n_i), n_i/(p_i+n_i))$$

Per *Patrons?* è 0.459 bit, per *Type* è (ancora) di un 1 bit.

→ scegliere l'attributo che minimizza la quantità d'informazione necessaria rimasta.

Albero di decisione appreso da 12 esempi:

Sostanzialmente più semplice dell'albero visto prima. Uno scarso quantitativo di dati non giustifica la formulazione di un'ipotesi più complessa.



IMPURITÀ:

Un punto di vista diverso è considerare **I(nodo)** come una misura di **impurità**. Più sono "misti" i campioni in un nodo (cioè proporzioni uguali di tutte le label di classe), maggiore è il valore di impurità. D'altra parte, un nodo **omogeneo** (cioè ha campioni di una sola classe) avrà impurità zero.

Il valore Gain(A) può quindi essere visto come la **quantità di riduzione dell'impurità** se dividiamo secondo A.

Ciò offre l'idea intuitiva che possiamo costruire alberi di decisione in modo ricorsivo cercando di ottenere nodi foglia che siano più puri possibile.

MISURAZIONE DELLE PRESTAZIONI:

Come facciamo a sapere se l'albero di decisione è riuscito ad approssimare bene il campione in input?

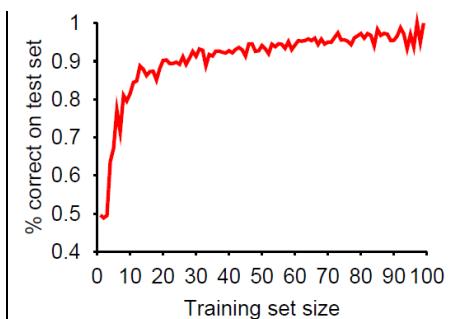
Come facciamo a sapere che $h \approx f$ (h approssima f)?

1. Usare teoremi sulla teoria dell'apprendimento computazionale/statistico;
2. Provare h su un nuovo insieme di esempi di test (usando la stessa distribuzione sullo spazio dell'esempio come insieme di training).

Curva d'apprendimento = % corretti sull'insieme di test in funzione delle dimensioni del training stesso.

Overfitting: rischio di utilizzare predici osservabili irrilevanti per generare un'ipotesi che sia d'accordo con tutti gli esempi nel training set. Il modello sceglie attributi che sono in realtà non rilevanti e potrei ignorarlo, si cerca di risolvere il problema con il pruning.

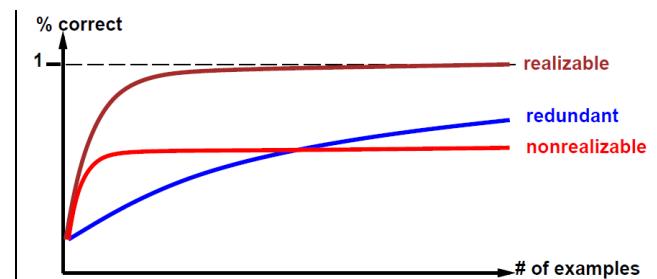
Tree pruning: termina la ricorsione quando # errori/gain è piccolo.



La curva di apprendimento dipende da:

- **Realizzabilità** (può esprimere la funzione target) vs **non realizzabilità**: La non realizzabilità può essere dovuta ad attributi mancanti o ad una classe d'ipotesi limitata;
- Espressività ridondante (ad esempio, molti attributi irrilevanti).

La realizzabilità con l'aumentare dei campioni ci porta quasi all'1 mentre la non realizzabilità si assesta su un valore ed anche con l'aumentare dei campioni restiamo su quel valore perché mancano informazioni cruciali per arrivare a percentuali alte.



GENERALIZZAZIONE E OVERFITTING:

Quello che capita molto spesso è che la funzione h si va a adattare troppo ai dati di training, cioè va a catturare degli aspetti che sono irrilevanti per la funzione f . In tal caso, accade che quando inviamo nuovi dati la funzione non generalizza bene, invece di comportarsi come f va a considerare questi nuovi dati che sono irrilevanti per la classificazione e da delle risposte differenti. L'**overfitting** è quando la funzione h va a modellare troppo i dati di training, al contrario si ha l'**underfitting** quando la funzione h non riesce a modellare f , probabilmente lo spazio delle ipotesi o i dati sono pochi.

Qualitativamente, l'overfitting aumenta con la dimensione dello spazio delle ipotesi e del numero di attributi, ma diminuisce con il numero di esempi.

DECISION TREE PRUNING:

L'Idea è combattere l'overfitting "generalizzando" gli alberi decisionali calcolati da DTL, sfoltendo i nodi irrilevanti.

Per la potatura dell'albero decisionale, ripetere quanto segue su un albero decisionale appreso:

- Trova un nodo di test **terminale** n (ha solo foglie come figli);
- Se il test è irrilevante, cioè ha un basso **information gain**, eliminalo sostituendo n con un nodo foglia.

NOTA: i nodi dove abbiamo poco guadagno di informazioni sono proprio le foglie.

Per vedere se eliminare o meno un nodo, nella pratica, si utilizzare un **test di significatività statistica**.

Un risultato ha **rilevanza statistica**, se la probabilità che possano derivare dall'ipotesi nulla (cioè l'assunzione che non vi sia un pattern sottostante) è molto bassa (di solito 5%).

12.3. VALUTARE E SCEGLIERE LA MIGLIORE IPOTESI

Il problema è che vogliamo apprendere un'ipotesi che si adatta meglio ai dati futuri. L'intuizione è che funziona solo se il training-set è "rappresentativo" per il processo sottostante.

L'idea è suppone che il training set sia indipendente e identicamente distribuito, vuol dire che gli elementi sono rappresentativi così che la funzione h può generalizzare.

Più formale, una sequenza di E_1, \dots, E_n delle variabili casuali è **indipendente e distribuita in modo identico** (IID), se è:

- **Indipendente**, la probabilità di avere un certo esempio non è condizionata dagli altri eventi, cioè $P(E_j | E_{(j-1)}, E_{(j-2)}, \dots) = P(E_j)$;
- **Identicamente distribuito**, tutti gli esempi sono equamente probabili, cioè $P(E_i) = P(E_j)$ per tutti gli i e j .

Esempio, una sequenza di lanci di dadi è IID.

Ipotesi di stazionarietà: assumiamo che l'insieme E di esempi sia IID in futuro.

TASSI DI ERRORE E CROSS-VALIDATION:

Per andare a misurare quanto la funzione che abbiamo appreso si avvicina alla funzione f , andiamo a definire il **tasso di errore**. Il problema è che, avendo il training set, per sapere se il modello si sta comportando bene bisogna valutarlo su dati che non sono stati visti durante il training set e quindi viene valutato su un insieme nuovo di dati, chiamato **test set**.

Dato un problema di apprendimento induttivo $\langle H, f \rangle$, definiamo il **tasso di errore** di un'ipotesi $h \in H$ come la frazione di errori, ovvero quante volte sbaglia rispetto al numero totale di valori del dominio:

$$\frac{\#\{x \in \text{dom}(f) \mid h(x) \neq f(x)\}}{\#\text{dom}(f)}$$

Un basso tasso di errore sul training set non significa che un'ipotesi si generalizzi bene.

La pratica di suddividere i dati disponibili per l'apprendimento in:

- un **training set**, da cui l'algoritmo di apprendimento produce un'ipotesi h ;
- un **test set**, che viene utilizzato per valutare h ,

è chiamato **holdout cross validation** (non è consentito sbirciare nel set di test).

Il problema ora è quanto grande deve essere il training set rispetto al test set:

- piccolo training set → scarse ipotesi, la funzione h non riesce a capire come si comporta f ;
- piccolo test set → scarsa stima dell'accuratezza.

Quello che si fa è una ***k-fold cross validation***, cioè eseguiamo k cicli di apprendimento, ciascuno con $1/k$ dei dati come test set e una media sui k tassi di errore.

Esempio:

$k = 5$ e $k = 10$ sono popolari → buona accuratezza con k volte il tempo di calcolo.

Se $k=5$ con un campione di 1000, questo lo si divide in 5 gruppi (ognuno di 200 campioni). I primi 200 sono usati come test set e i restanti 800 come training set, questo si ripete per 5 volte, dopodiché, si effettua la media.

Esiste una versione estrema chiamata ***leave-one-out cross validation (LOOCV)*** che è il caso in cui si usa un solo campione come test e il resto come training e questo viene ripetuto n volte.

SELEZIONE DEL MODELLO:

Il problema della ***selezione del modello*** consiste nel determinare, a partire dai dati, un buon spazio di ipotesi in cui a cercare h (evitando l'overfitting).

Possiamo risolvere il problema di "apprendere f dalle osservazioni" in un processo in due parti:

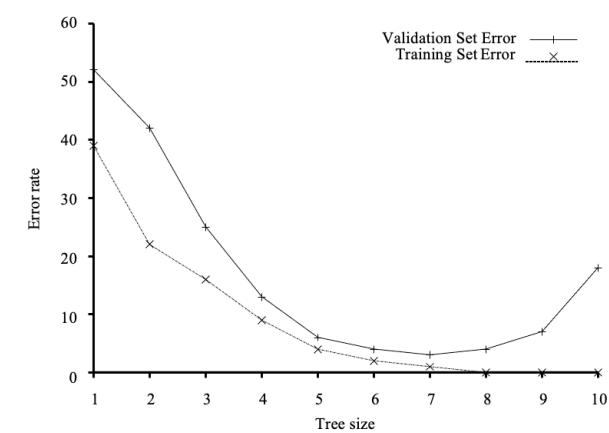
- la ***selezione del modello*** determina uno spazio di ipotesi H ;
- l'***ottimizzazione*** risolve il problema di apprendimento induttivo indotto (H, f).

L'idea è risolvere le due parti insieme, iterando su "dimensione", ma serve una nozione di "dimensione" (ad es. numero di nodi in un albero decisionale).

Un problema concreto è trovare la "dimensione" che meglio bilancia overfitting e underfitting per ottimizzare la precisione del test set.

Si vanno a costruire tanti alberi di decisione con dimensione crescente e si calcola il tasso di errore, sia per il training che per il test set.

Nello schema si sceglie la dimensione 7 siccome successivamente la curva (tasso di errore) ricomincia a salire.



DA ERROR-RATE A LOSS FUNCTION:

Quello appena misurato si chiama ***error-rate***, ovvero quante volte h risponde in maniera diversa da f . In realtà, quando si vanno a costruire questi modelli non interessa misurare l'errore ma misurare la ***loss function***, cioè andare a vedere il tipo di errore ricevuto.

Ad esempio, supponiamo di creare un classificatore di mail, è molto peggio classificare ham (mail legittime) come spam che viceversa (perdita del messaggio). In questo caso, con l'error-rate sono classificati allo stesso modo (conta 1) mentre con la loss-function si può dare un peso differente rispetto al tipo di errore.

Così facendo, anziché parlare di errore si parla di utilità attesa, e quello che si vuole è ***massimizzare l'utilità attesa*** (MEU).

Quindi, l'apprendimento automatico dovrebbe massimizzare l'utilità (non solo ridurre al minimo i tassi di errore).

L'apprendimento automatico si occupa tradizionalmente di utilità sotto forma di "funzioni di loss".

La ***loss function*** L è definita impostando $L(x, y, \hat{y})$ come la quantità di utilità persa dalla predizione $h(x) = y$ invece di $f(x) = y$. Se L è indipendente da x , usiamo spesso $L(y, \hat{y})$.

Esempio:

$$L(\text{spam}, \text{ham}) = 1, \text{ mentre } L(\text{ham}, \text{spam}) = 10.$$

Nota: $L(y, \hat{y}) = 0$ (nessuna perdita se hai esattamente ragione)

Le funzioni di loss popolari sono:

absolute value loss		$L_1(y, \hat{y}) := (y - \hat{y}) $
squared error loss		$L_2(y, \hat{y}) := (y - \hat{y})^2$
0/1 loss		$L_{0/1}(y, \hat{y}) := 0, \text{ if } y = \hat{y}, \text{ else } 1$

L'idea è massimizzare l'***utilità attesa*** scegliendo l'ipotesi h che riduce al minimo la loss attesa su tutte le coppie $(x, y) \in f$.

Sia \mathcal{E} l'insieme di tutti i possibili esempi e $\mathbf{P}(X, Y)$ la distribuzione di probabilità a priori sulle sue componenti, allora la **generalization loss** per un'ipotesi h rispetto a una loss function L è:

$$\text{GenLoss}_L(h) := \sum_{(x,y) \in \mathcal{E}} L(y, h(x)) \cdot \mathbf{P}(x, y)$$

(se si ha il training set e la distribuzione di probabilità di X e Y possiamo definire la perdita generalizzata per una certa ipotesi rispetto ad una funzione di loss = si somma il prodotto per ogni campione di training moltiplichiamo la loss per la probabilità che quel campione appaia, quindi è pesata rispetto alla distribuzione di probabilità)

Quindi bisogna scegliere h che minimizza questa loss generalizzata, e l'ipotesi migliore è:

$$h^* := \underset{h \in \mathcal{H}}{\operatorname{argmin}} \text{GenLoss}_L(h).$$

Sia L una funzione di perdita ed E un insieme di esempi con $\#(E) = N$, allora chiamiamo:

$$\text{EmpLoss}_{L,E}(h) := \frac{1}{N} \left(\sum_{(x,y) \in E} L(y, h(x)) \right)$$

$$\hat{h}^* := \underset{h \in \mathcal{H}}{\operatorname{argmin}} \text{EmpLoss}_{L,E}(h)$$

Infine, la **loss empirica** e \hat{h}^* è la migliore ipotesi stimata.

Ci sono quattro ragioni per cui \hat{h}^* può differire da f :

- **Realizzabilità**: allora dobbiamo accontentarci di un'approssimazione \hat{h}^* di f ;
- **Varianza**: diversi sottoinsiemi di f danno diverse \hat{h}^* → abbiamo bisogno di più esempi;
- **Rumore**: se f non è deterministico, allora non possiamo aspettarci risultati perfetti;
- **Complessità computazionale**: se H è troppo grande per essere esplorato sistematicamente, usiamo un sottoinsieme e otteniamo un'approssimazione.

REGOLARIZZAZIONE:

Per bilanciare complessità e loss è la **regolarizzazione**, ovvero si vuole selezionare un modello e quello che vogliamo scegliere è quello con la loss migliore, ma minimizzando la loss potremmo creare un modello troppo complesso e si va in overfitting.

Quello che si fa è andare a definire il **costo** come la somma pesata tra loss e complessità.

Sia $\lambda \in \mathbb{R}$ (*lambda*), $h \in \mathcal{H}$ ed E un insieme di esempi, chiamiamo:

$$\text{Cost}_{L,E}(h) := \text{EmpLoss}_{L,E}(h) + \lambda \text{Complexity}(h)$$

il costo totale di h su E .

Il processo per trovare un'ipotesi di minimizzazione dei costi totali

$$\hat{h}^* := \underset{h \in \mathcal{H}}{\operatorname{argmin}} \text{Cost}_{L,E}(h)$$

si chiama **regolarizzazione**. La complessità è chiamata **funzione di regolarizzazione** o complessità di ipotesi.

12.4. REGRESSIONE E CLASSIFICAZIONE CON MODELLI LINEARI

Nell'esempio precedente ci siamo concentrati sulla classificazione perché quello che volevamo classificare (il codominio) era booleano, più nel preciso, quando un valore è discreto parliamo di classificazione, mentre quando è continuo parliamo di **regressione**.

Chiamiamo un problema di apprendimento induttivo $\langle H, f \rangle$ un **problema di classificazione** se il *co-dominio*(f) è discreto, e un **problema di regressione** se *co-dominio*(f) è continuo, cioè non discreto (di solito a valori reali).

Il caso più semplice di regressione è quando abbiamo una **funzione univariata**, è una funzione con un solo argomento.

NOTA: un mapping tra spazi vettoriali è detto lineare se conserva l'addizione vettoriale e la moltiplicazione scalare.

Una funzione univariata, lineare $f: \mathbb{R} \rightarrow \mathbb{R}$ è della forma:

$$f(x) = w_1 x + w_0 \text{ per qualche } w_i \in \mathbb{R}.$$

Dato un vettore $w := (w_0, w_1)$, definiamo $h_w(x) := w_1 x + w_0$

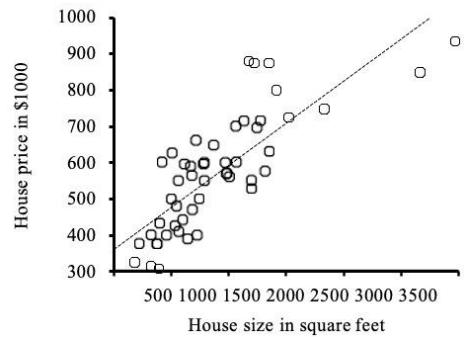
Dato un insieme di esempi $E \subseteq \mathbb{R} \times \mathbb{R}$, il compito di trovare l' h_w che meglio si adatta a E è chiamato **regressione lineare**.

Esempio:

Supponiamo di avere prezzi delle case rispetto ai feet² (metri quadri) nelle case vendute a Berkeley.

Vogliamo trovare i valori w_1 e w_0 che rappresentano la retta che minimizza l'errore quadratico, così che se viene fornito un prezzo, la soluzione dice quanto è grande la casa in metri quadri.

Ipotesi di funzione lineare che minimizza la perdita (quadrato dell'errore $y = 0,232x + 246$).



L'idea è ridurre al minimo la loss come quadrato dell'errore L_2 su $\{(x_i, y_i) | i \leq N\}$ (utilizzato già da Gauss).

$$\text{Loss}(\mathbf{h}_w) = \sum_{j=1}^N L_2(y_j, h_w(x_j)) = \sum_{j=1}^N (y_j - h_w(x_j))^2 = \sum_{j=1}^N (y_j - w_1 x_j + w_0)^2$$

Il compito è trovare:

$$\mathbf{w}^* := \underset{\mathbf{w}}{\operatorname{argmin}} \text{Loss}(\mathbf{h}_w)$$

Ricordando che $\sum_{j=1}^N (y_j - w_1 x_j + w_0)^2$ è minimizzato, quando le derivate parziali rispetto ai w_i sono zero, cioè quando:

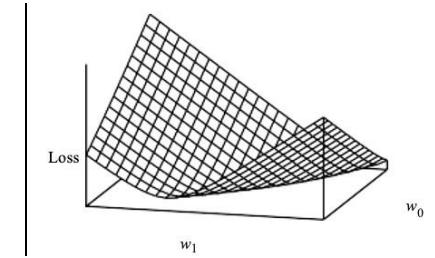
$$\frac{\partial}{\partial w_0} \left(\sum_{j=1}^N (y_j - w_1 x_j + w_0)^2 \right) = 0 \quad \text{and} \quad \frac{\partial}{\partial w_1} \left(\sum_{j=1}^N (y_j - w_1 x_j + w_0)^2 \right) = 0$$

Queste equazioni hanno una soluzione unica, andando a risolvere l'equazione iniziale:

$$w_1 = \frac{N(\sum_j x_j y_j) - (\sum_j x_j)(\sum_j y_j)}{N(\sum_j x_j^2) - (\sum_j x_j)^2} \quad w_0 = \frac{(\sum_j y_j) - w_1(\sum_j x_j)}{N}$$

Andando a disegnare i valori di w_1 e w_0 si ottiene il grafico:

Possiamo variare i valori w_i affinché la loss vada verso un minimo (il punto più basso è il minimo di loss, ovvero ciò che si vuole).



NOTA: molte forme di apprendimento implicano la regolazione dei pesi per ridurre al minimo le perdite.

Lo **spazio dei pesi** è lo spazio di tutte le possibili combinazioni di pesi. La minimizzazione della perdita in uno spazio di peso è chiamata **adattamento del peso**.

Lo spazio dei pesi della regressione lineare univariata è R^2 → possiamo tracciare graficamente la funzione di perdita su R^2 .

DISCESA DEL GRADIENTE:

Il problema è che non usiamo sempre R^2 , ci vuole un approccio più generale che funziona per qualsiasi loss. Possiamo andare a cercare il valore di minimo usando un algoritmo di ricerca locale, in particolare l'algoritmo (hill-climbing) **discesa del gradiente**.

In pratica, se si prende il grafico precedente, il gradiente in un punto ci dà la direzione di crescita, si prende la direzione opposta (ovvero la discesa) e si individua il minimo. L'algoritmo aggiorna i singoli pesi andando nella direzione opposta del gradiente calcolato con la derivata rispetto ad ogni singolo peso.

Il parametro α è chiamato **learning rate**, che è la grandezza dei passi per andare verso il minimo, più è piccolo più tempo è richiesto ma la probabilità aumenta di arrivarci al valore minimo. Può essere una costante fissa o può decadere man mano che l'apprendimento procede.

```
function gradient_descent(f, w, alpha)
    inputs: a differentiable function f and initial weights w = (w0, w1).
    loop until w converges do
        for each wi do
            wi ← wi - alpha ∂/∂wi (f(w))
        end for
    end loop
```

La **discesa del gradiente** aggiorna i pesi con la seguente formula:

$$\frac{\partial \text{Loss}(w)}{\partial w_i} = \frac{\partial (y - h_w(x))^2}{\partial w_i} = 2(y - h_w(x)) \frac{\partial (y - w_1 x + w_0)}{\partial w_i}$$

e così abbiamo:

$$\frac{\partial \text{Loss}(w)}{\partial w_0} = -2(y - h_w(x)) \quad \frac{\partial \text{Loss}(w)}{\partial w_1} = -2(y - h_w(x))x$$

Collegandolo negli aggiornamenti della discesa del gradiente:

$$w_0 \leftarrow w_0 - \alpha - 2(y - h_w(x)) \quad w_1 \leftarrow w_1 - \alpha - 2(y - h_w(x))x$$

Analogamente per n esempi di addestramento (x_j, y_j) :

$$w_0 \leftarrow w_0 - \alpha \left(\sum_j -2(y_j - h_w(x_j)) \right) \quad w_1 \leftarrow w_1 - \alpha \left(\sum_j -2(y_j - h_w(x_n)) x_n \right)$$

Questi aggiornamenti costituiscono la **regola di apprendimento** della discesa del gradiente batch per la regressione lineare univariata. La convergenza all'unico minimo di perdita globale è garantita (a patto che scegliamo α sufficientemente piccolo), ma potrebbe essere molto lenta.

REGRESSIONE LINEARE MULTIVARIATA:

Una **funzione multivariata** o n-aria è una funzione con uno o più argomenti. Possiamo usarla per la regressione lineare multivariata. Le formule usate prima possono essere usate ma bisogna passare da coppie di valori a vettori.

L'idea è che ogni esempio \vec{x}_j è un vettore di n elementi e lo spazio delle ipotesi è l'insieme di funzioni:

$$h_{sw}(\vec{x}_j) = w_0 + w_1 x_{j,1} + \dots + w_n x_{j,n} = w_0 + \sum_i w_i x_{j,i}$$

Il trucco è introduciamo $x_{j,0} := 1$ e usiamo la notazione matriciale:

$$h_{sw}(\vec{x}_j) = \vec{w} \cdot \vec{x}_j = \vec{w}^t \vec{x}_j = \sum_i w_i x_{j,i}$$

Il miglior vettore di pesi, w^* , **minimizza** la perdita di errore quadratico sugli esempi:

$$w^* := \underset{w}{\operatorname{argmin}} \left(\sum_j L_2(y_j)(w \cdot \vec{x}_j) \right)$$

La discesa del gradiente raggiungerà il minimo (unico) della funzione di perdita; l'equazione di aggiornamento per ogni peso

$$w_i \leftarrow w_i - \alpha \left(\sum_j x_{j,i} (y_j - h_w(\vec{x}_j)) \right)$$

Possiamo anche risolvere analiticamente il w^* che **minimizza** la perdita.

Sia \vec{y} il vettore di output per gli esempi di addestramento e X sia la matrice di dati, ovvero la matrice di input con un esempio n dimensionale per riga.

Allora la soluzione è $w^* = (X^t X)^{-1} X^t \vec{y}$ che **minimizza l'errore quadratico**.

REGRESSIONE LINEARE MULTIVARIATA (REGOLARIZZAZIONE):

NOTA: la regressione lineare univariata non soffre di overfit, ma nel caso multivariato potrebbero esserci "dimensioni ridondanti" che si traducono in un overfitting.

L'idea è utilizzare la **regolarizzazione** con una funzione di complessità basata sui pesi.

$$\text{Complexity}(h_w) = L_q(w) = \sum_i |w_i|^q$$

Avvertenza: non confonderlo con le loss function L_1 e L_2 .

Il problema è quale q dovrebbe essere scelto (L_1 e L_2 minimizzano la somma dei valori assoluti / dei quadrati), ma questo dipende dall'applicazione.

NOTA: la regolarizzazione L_1 tende a produrre un **modello sparso**, ovvero imposta molti pesi a 0, dichiarando di fatto gli attributi irrilevanti. Le ipotesi che scartano gli attributi possono essere più facili da comprendere per un essere umano e potrebbero avere meno probabilità di overfit.

CLASSIFICAZIONI LINEARI CON SOGLIA RIGIDA:

L'idea è che il risultato della regressione lineare può essere utilizzato per la classificazione. Fino ad ora avevamo un insieme di dati di training (i punti nello spazio), determinato i pesi della funzione (lineare) quindi una retta, si può usare questa retta per fare classificazione dicendo che tutto ciò che è presente al di sopra della retta appartiene ad una specifica classe e tutto ciò che sta sotto la retta appartiene ad un'altra classe.

Esempio (Verifica del divieto di test nucleari):

Grafici dei parametri dei dati sismici: magnitudo dell'onda di volume x_1 vs. magnitudo dell'onda superficiale x_2 .

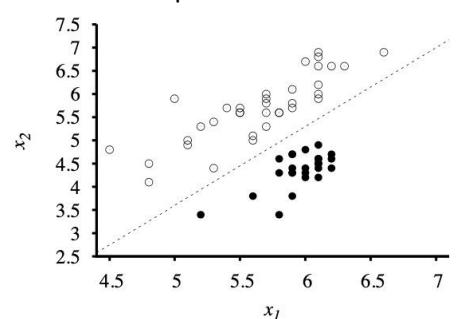
Bianco: terremoti, nero: esplosioni sotterranee

Inoltre: h_{w^*} come confine di decisione $x_2 = 17x_1 - 4.9$.

Un confine di decisione è una linea (o una superficie, in dimensioni superiori) che separa due classi di punti. Un confine di decisione lineare è chiamato separatore lineare e i dati che ne ammettono uno sono chiamati linearmente separabili.

Per l'esempio, il separatore lineare è definito da $-4.9 + 1.7x_1 - x_2 = 0$, le esplosioni sono caratterizzate da $-4.9 + 1.7x_1 - x_2 > 0$, terremoti da $-4.9 + 1.7x_1 - x_2 < 0$.

Trucco: se introduciamo la coordinata fittizia $x_0 = 1$, allora possiamo scrivere l'ipotesi di classificazione come $h_w(x) = 1$ se $w^*x > 0$ e 0 altrimenti.



CLASSIFICAZIONI LINEARI CON SOGLIA RIGIDA (REGOLA DEL PERCETTRONE):

Quindi $h_w(x) = 1$ se $w^*x > 0$ e 0 altrimenti è ben definito, come scegliere w ?

Anziché scrivere $h_w(x) = T(w^*x)$, possiamo scrivere $T(z) = 1$, se $z > 0$ e $T(z) = 0$ altrimenti, chiamiamo T **funzione di soglia**.

Il problema è che T non è derivabile e $\frac{\partial T}{\partial z} = 0$ è definito, non si può usare:

- Nessuna soluzione in forma chiusa impostando;
- Nemmeno i metodi di discesa del gradiente nello spazio dei pesi funzionano.

C'è un'altra tecnica dove, possiamo apprendere i pesi ripetendo la seguente regola:

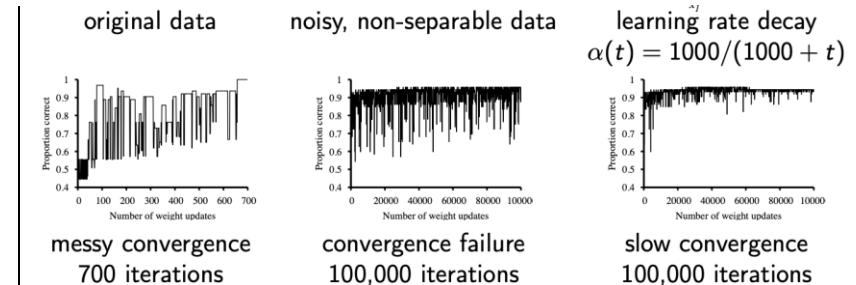
Dato un esempio (x, y) , la regola di **apprendimento del perceptron** è

$$w_i \leftarrow w_i + \alpha \cdot (y - h_w(x)) \cdot x_i$$

Poiché stiamo considerando la classificazione 0/1, ci sono tre possibilità:

1. Se $y = h_w(x)$, allora w_i rimane invariato;
2. Se $y = 1$ e $h_w(x) = 0$, allora w_i è incrementato/diminuito sé x_i è positivo/negativo (vogliamo rendere w^*x più grande in modo che $T(w^*x) = 1$);
3. Se $y = 0$ e $h_w(x) = 1$, allora w_i è decrescente/aumentato se x_i è positivo/negativo (vogliamo per ridurre w^*x in modo che $T(w^*x) = 0$).

Curve di apprendimento (plots dell'accuratezza totale del **training set** rispetto al numero di iterazioni) per la regola del percettrone sui dati di terremoti/esplorazioni:



Trovare l'ipotesi dell'errore minimo è NP hard, ma è possibile con il decadimento del **learning rate**.

CLASSIFICAZIONI LINEARI CON REGRESSIONE LOGISTICA:

Finora abbiamo visto che passando l'output di una funzione lineare attraverso una funzione di soglia T si ottiene un classificatore lineare. Ma il problema è che la natura difficile di T porta problemi:

- T non è differenziabile ne continuo, l'apprendimento tramite la regola del percettrone diventa imprevedibile;
- T è "eccessivamente preciso" vicino al confine, necessita di giudizi più graduati.

L'idea è ammorbidente la soglia, approssimarla con una funzione differenziabile.

Usiamo la funzione **logistica standard**:

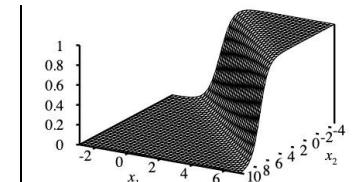
$$l(x) = \frac{1}{1+e^{-x}}$$

Quindi abbiamo:

$$h_w(x) = l(w \cdot x) = \frac{1}{1+e^{-(w \cdot x)}}$$

(Ipotesi di regressione logistica nello spazio peso) Grafico di un'**ipotesi di regressione logistica** per i dati di terremoto/esplorazione. Il valore in (w_0, w_1) è la probabilità di appartenere alla classe etichettata 1.

In tal caso, parliamo intuitivamente della **scogliera** nel classificatore.



REGRESSIONE LOGISTICA:

Il processo di adattamento del peso in $h_w(x) = \frac{1}{1+e^{-(w \cdot x)}}$ viene chiamato **regressione logistica**.

Non esiste una soluzione semplice in forma chiusa, ma la discesa del gradiente è una regressione logistica semplice.

Poiché le nostre ipotesi hanno un output continuo, utilizziamo la funzione di perdita di errore quadratica L_2 .

Per un esempio (x, y) calcoliamo le derivate parziali (tramite **chain rule**):

$$\begin{aligned} \frac{\partial}{\partial w_i} (L_2(w)) &= \frac{\partial}{\partial w_i} (y - h_w(x))^2 \\ &= 2 \cdot h_w(x) \cdot \frac{\partial}{\partial w_i} (y - h_w(x)) \\ &= -2 \cdot h_w(x) \cdot l'(w \cdot x) \cdot \frac{\partial}{\partial w_i} (w \cdot x) \\ &= -2 \cdot h_w(x) \cdot l'(w \cdot x) \cdot x_i \end{aligned}$$

La derivata della funzione logistica soddisfa $l'(z) = l(z)(1-l(z))$, quindi:

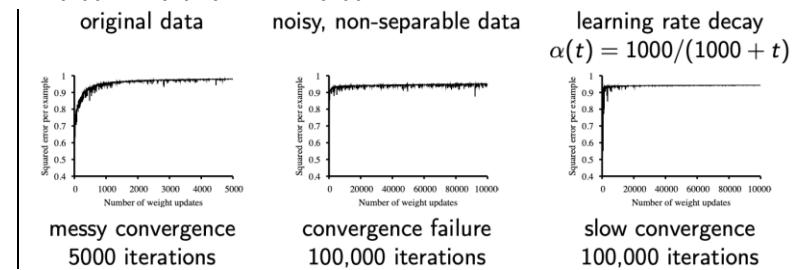
$$l'(\mathbf{w} \cdot \mathbf{x}) = l(\mathbf{w} \cdot \mathbf{x})(1 - l(\mathbf{w} \cdot \mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))$$

La regola per l'aggiornamento logistico (aggiornamento del peso per minimizzare la perdita) è:

$$\mathbf{w}_i \leftarrow \mathbf{w}_i + \alpha \cdot (y - h_{\mathbf{w}}(\mathbf{x})) \cdot h_{\mathbf{w}}(\mathbf{x}) \cdot (1 - h_{\mathbf{w}}(\mathbf{x})) \cdot \mathbf{x}_i$$

Rifacendo le curve di training:

Risulta che l'**aggiornamento logistico** sembra funzionare meglio dell'**aggiornamento del percettrone**.



12.5. MODELLI NON PARAMETRICI

Ricapitolando, si vuole fare apprendimento tramite esempi (campioni). Dato un training set, insieme di valori/vettori \mathbf{x} con associata una label di classificazione, l'obiettivo è cercare di apprendere una funzione h di ipotesi che va a modellare la funzione f effettiva che corrisponde al mapping tra \mathbf{x} e y . Abbiamo visto alcuni casi di classificazione dove la y è discreta o continua (in tal caso diventa un problema di regressione) e poi con gli alberi di decisione. Si è poi stimata una funzione lineare per poi passare a funzioni multivariate.

Quindi, a partire dalla coppia \mathbf{x} e y si è definito i valori dei pesi w dove \mathbf{x}^*w approssima y , che è il risultato dell'apprendimento.

I modelli precedenti sono parametrici perché hanno come parametro quanti pesi dobbiamo stimare (es. nel classificatore lineare bisogna determinare w_0 e w_1). I metodi di apprendimento parametrico sono spesso semplici ed efficaci, ma semplificano eccessivamente ciò che realmente succede.

Esistono altri tipi di modelli, chiamati **modelli non parametrici** dove si vuole che la complessità del modello sia in funzione dei dati. In un classificatore lineare (il più semplice) abbiamo molti dati, accadrà che il modello non si adatta al funzionamento dei dati siccome non ha la capacità espressiva dell'ipotesi di partenza, cioè lo spazio delle ipotesi troppo piccolo; pertanto, non si riesce a modellare la funzione presente dietro ai dati. Con i modelli non parametrici, il parametro esterno non è presente, quindi il modello che si va a costruire dipende solamente dai dati; pertanto, più dati si hanno più il modello riuscirà a catturare tutti gli aspetti del dataset, in altre parole, l'apprendimento non parametrico permette alla complessità delle ipotesi di crescere con i dati; quindi, lo spazio delle ipotesi è in funzione del dataset.

L'apprendimento basato sulle istanze è non parametrico siccome costruisce delle ipotesi direttamente dai dati di training.

MODELLI NEAREST-NEIGHBOR:

Il primo modello non parametrico si chiama **modello K-NN** che è basato sulla vicinanza.

L'idea chiave è che i vicini sono simili:

- Sia X un insieme di esempi etichettati (il training set);
- Dato un punto x da classificare (un nuovo campione non presente nel training set), si calcola l'insieme U dei k punti dell'insieme X più vicini a x , secondo una determinata metrica;
- Si calcola la classe C più frequente all'interno dell'insieme U ;
- Si assegna x a C ;

Un aspetto importante è come definire e quanto deve essere grande il **vicinato N** :

- Se è troppo piccolo, non ci sono data point;
- Se è troppo grande, la densità è la stessa ovunque;
- Una soluzione è quella di definire N per contenere k point, dove k è grande abbastanza da assicurare una stima significativa (di solito tra 5 e 10).

Un altro aspetto è come determinare i punti vicini (Quale data point è più vicino a x ?). Abbiamo quindi bisogno di una distanza metrica $D(x_1, x_2)$ e la distanza euclidea D_E è la più popolare. Quando però ogni dimensione misura qualcosa di diverso è inappropriato usare D_E , è importante standardizzare la scala di ogni dimensione, in tal caso, la distanza di Mahalanobis è una soluzione. Invece, le caratteristiche discrete dovrebbero essere trattate diversamente, magari con la distanza di Hamming.

La difficoltà di determinare i punti vicini è che più aumenta la dimensione, più è complesso determinare i vicini.

SUPPORT-VECTOR MACHINES:

L'approccio utilizzato fino a qualche anno fa è il support-vector machines (SVM oppure support-vector networks) che sono modelli di apprendimento supervisionato (non parametrico) per la classificazione e la regressione.

Le caratteristiche del modello SVM sono:

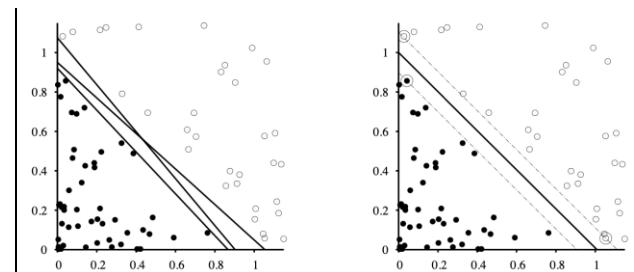
- I dati di training vengono separati tramite un separatore così da separare gli elementi di una classe da un'altra classe costruendo un separatore di **margine massimo**, ovvero un confine di decisione con la maggiore distanza possibile dai punti di esempio. Questo li aiuta a **generalizzare** bene.
- Si può utilizzare anche quando lo spazio non è separabile usando una funzione kernel, può incorporare i dati in uno spazio a dimensione superiore, dove è linearmente separabile dal **trucco del kernel**.
- L'iperpiano di separazione è un'ipersuperficie nei dati originali, danno priorità agli esempi critici (**vettori di supporto**), ovvero tutti i punti vicino al separatore sono quelli più interessanti, tutti gli altri non vengono memorizzati siccome non servono all'algoritmo per fare classificazione (**migliore generalizzazione**).

Uno degli approcci più diffusi per l'apprendimento supervisionato è "off-the-shelf".

Quindi, dato un dataset E linearmente separabile, il **separatore di margine massimo** è il **separatore lineare** s che **massimizza** il **margine**, ovvero la distanza di E da s.

Esempio:

Tutte le righe a sinistra sono validi separatori lineari che separano la classe dei punti neri dalla classe dei punti bianchi:



Ci aspettiamo che il separatore del margine massimo a destra si generalizzi meglio.

L'idea è ridurre al minimo la generalized loss prevista anziché la loss empirica.

TROVARE IL SEPARATORE DI MARGINE MASSIMO:

Rispetto agli algoritmi precedenti, le label non sono 1/0 ma 1/-1 per indicare le due classi, pertanto abbiamo un training set $\{(x_1, y_1), \dots, (x_n, y_n)\}$ dove:

- $y_i \in \{-1, 1\}$ (invece di $\{1, 0\}$);
- $x_i \in \mathbb{R}^p$ (classificazione multilineare).

L'obiettivo è trovare l'iperpiano che va a separare al massimo i punti con $y_i = -1$ da quelli con $y_i = 1$. Questo iperpiano possiamo rappresentarlo con un insieme di punti tale che si ha $\{x | (w^*x) + b = 0\}$, dove:

- w è il vettore normale (non necessariamente normalizzato) dell'iperpiano;
- Il parametro b determina l'offset dell'iperpiano dall'origine lungo il vettore normale w .

L'idea è utilizzare la discesa del gradiente per cercare lo spazio di tutti w e b per massimizzare le combinazioni.

CASO SEPARABILE:

Innanzitutto definiamo come si può definire il margine, ovvero il margine è delimitato dai due iperpiani descritti da $(w \cdot x) + b = -1$ (**limite inferiore**) e $(w \cdot x) + b = 1$ (**limite superiore**), mentre la distanza tra loro è $\frac{2}{\|w\|_2}$ per massimizzare il margine, minimizzare $\|w\|_2$ mantenendo x_i fuori dal margine.

I vincoli sono che: $(w^*x_i) + b \geq 1$ per $y_i = 1$ e $(w^*x_i) + b \leq -1$ per $y_i = -1$ o semplicemente $y_i(w^*x_i - b) \geq 1$ per $1 \leq i \leq n$.

Problema di ottimizzazione: minimizzare $\|w\|_2$ mentre $y_i(w^*x_i - b) \geq 1$ per $1 \leq i \leq n$.

Dopo alcuni passaggi arriviamo ad una **rappresentazione alternativa** (diventando un problema di ottimizzazione):

$$\underset{\alpha}{\operatorname{argmax}} \left(\sum_j \alpha_j - \frac{1}{2} \left(\sum_{j,k} \alpha_j \alpha_k y_j y_k (x_j \cdot x_k) \right) \right) \quad \text{sotto i vincoli } \alpha_j \geq 0 \text{ e } \sum_j \alpha_j y_j = 0.$$

Questa equazione ha tre proprietà importanti:

1. L'espressione è convessa, quindi il massimo può essere trovato in modo efficiente ed è unico;
2. I dati entrano nell'espressione solo sotto forma di prodotti scalari di coppie di punti, quindi una volta che sono stati calcolati gli α_i ottimali, abbiamo:

$$h(x) = \operatorname{sign} \left(\sum_j \alpha_j y_j (x \cdot x_j) - b \right)$$

3. I pesi α_j associati a ciascun punto sono zero tranne che in corrispondenza dei vettori di supporto, i punti più vicini al separatore, non c'è bisogno di tenere in memoria tutti i punti.

Una volta trovato un vettore ottimale α , utilizziamo $w = \sum_j \alpha_j x_j$

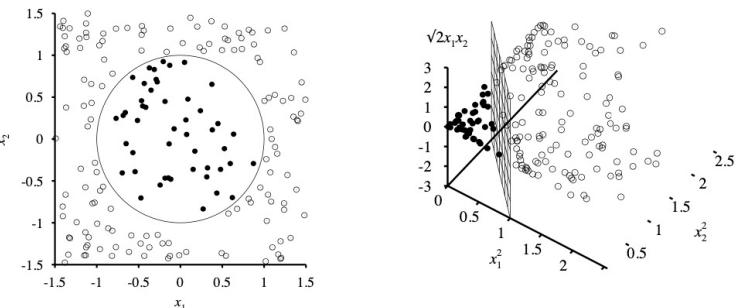
SUPPORT-VECTOR MACHINES (KERNEL TRICK):

Quando i ***dati non sono separabili***, l'idea è di passare da una determinata dimensione ad una superiore. È dimostrabile che se abbiamo un certo numero di campioni si può passare da uno spazio non separabile ad uno separabile aumentando la dimensione.

Esempio:

- A sinistra: il vero limite di decisione è $x_1^2 + x_2^2 \leq 1$;
- A destra: mappare uno spazio di input tridimensionale $\langle x_1^2, x_2^2, \sqrt{2}x_1x_2 \rangle$ è separabile da un iperpiano.

Risultato: Mappiamo ogni vettore di input x su una $F(x)$ con $f_1 = x_1$, $f_2 = x_2$ e $f_3 = \sqrt{2}x_1x_2$.



Esistono diverse funzioni $F(x)$ per l'aumento della dimensione ed hanno diverse proprietà con relativi vantaggi.

Notare che, visto che va fatto il prodotto tra $x_i \cdot x_j$ nella formula precedente, in questo caso sarà $F(x_i) \cdot F(x_j)$ nell'equazione SVM. In realtà, se definiamo $F()$ secondo determinati criteri, come ad esempio fare il quadrato delle x , cioè x_1^2 e x_2^2 e sulla terza dimensione è proprio $\sqrt{2}x_1x_2$, allora per calcolare il prodotto equivale a fare il prodotto delle x^2 , riassumendo:

$$\text{Se } F(x) = \langle x_1^2, x_2^2, \sqrt{2}x_1x_2 \rangle \text{ allora } F(x_i) \cdot F(x_j) = (x_i \cdot x_j)^2$$

Chiamiamo la funzione $(x_i \cdot x_j)^2$ una ***funzione kernel*** (ce ne sono altre).

Sia X un insieme non vuoto, a volte indicato come insieme di indici. Una funzione simmetrica $K: X \times X \rightarrow \mathbb{R}$ è chiamata funzione kernel su X se e solo se:

$$\sum_{i,j=1}^n c_i c_j K(x_i, x_j) \geq 0 \text{ for any } x_i \in X, n \in \mathbb{N}, \text{ and } c_i \in \mathbb{R}$$

Quindi la formula precedente per spazi separabili, nel caso di spazi non separabili, diventa:

$$\underset{\alpha}{\operatorname{argmax}} \left(\sum_j \alpha_j - \frac{1}{2} \sum_{j,k} \alpha_j \alpha_k y_j y_k K(x_j, x_k) \right) \quad \text{dove } K \text{ è una } \textcolor{red}{\text{funzione kernel}}.$$

La funzione $K(x_j, x_k) = (1 + x_j \cdot x_k)^d$ è una funzione kernel corrispondente a uno spazio delle caratteristiche la cui dimensione ed esponenziale in d . Si chiama ***kernel polinomiale***.

12.6. ENSEMBLE LEARNING

Abbiamo visto che a partire dai dati possiamo costruire modelli che però possono avere problemi (over/underfitting) e potrebbero essere non performanti. L'idea dell'***ensemble learning*** è cercare di costruire più modelli (non uguali) così da catturare caratteristiche differenti del problema, cosicché mettendoli assieme è più probabile che la classificazione sia fatta in maniera corretta e performano meglio.

L'idea dell'ensemble learning è quella di selezionare una raccolta, o insieme, di ipotesi, h_1, h_2, \dots, h_n (anziché una sola ipotesi) e combinare le loro previsioni con un criterio, ad esempio facendo la media, votazione o attraverso un altro livello di apprendimento automatico. Chiamiamo le singole ipotesi ***modelli base*** e loro combinazione un ***modello ensemble***.

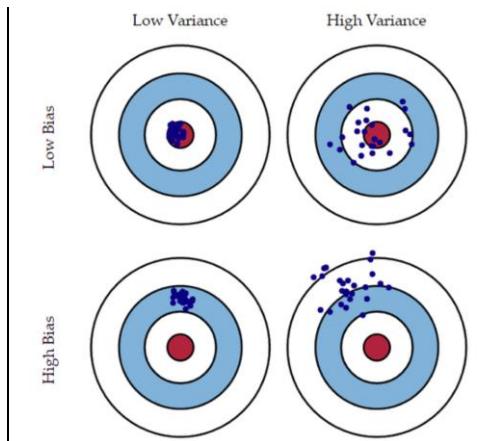
Dal punto di vista teorico, un modello di ensemble learning va ad affrontare due problemi di cui soffrono i classificatori:

1. Ridurre il ***bias*** (distorsione);
2. Ridurre la ***varianza***.

Tramite dei passaggi si può separare l'errore ottenuto in 3 errori:

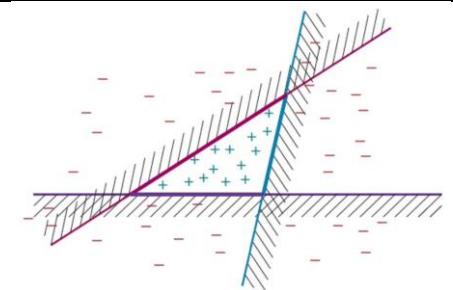
$$\mathbb{E}[(y - t)^2] = \underbrace{(y_* - \mathbb{E}[y])^2}_{\text{bias}} + \underbrace{\text{Var}(y)}_{\text{variance}} + \underbrace{\text{Var}(t)}_{\text{Bayes error}}$$

↗ Perdita attesa



Un ensemble di ***tre classificatori*** lineari può rappresentare una regione triangolare che non potrebbe essere rappresentata da un ***singolo classificatore*** lineare.

Un ensemble di ***n classificatori*** lineari consente di realizzare più funzioni.



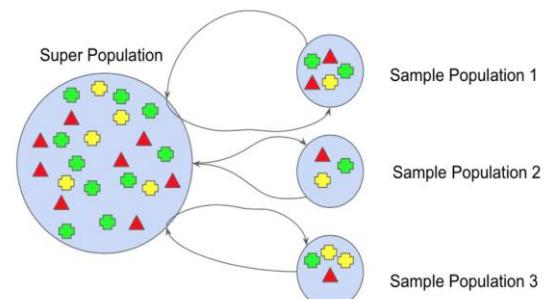
BAGGING (TECNICA DI ENSEMBLE LEARNING):

L'idea è di costruire tanti training set (non solo uno come in precedenza) in maniera casuale facendo campionamento, può capitare che lo stesso campione rientri in più training set. A partire dai K training set, si costruiscono i modelli di base, dopodiché si fa classificazione ed il risultato lo si può unire secondo delle funzioni (tipo per maggioranza).

Più formalmente, nel **bagging**, generiamo K distinti training set campionando con sostituzione dal training set originale.

Il bagging tende a ridurre la **varianza** ed è un approccio standard quando i dati sono limitati o quando si ritiene che il modello di base sia in overfitting.

Lo si usa per lo più su alberi di decisione.



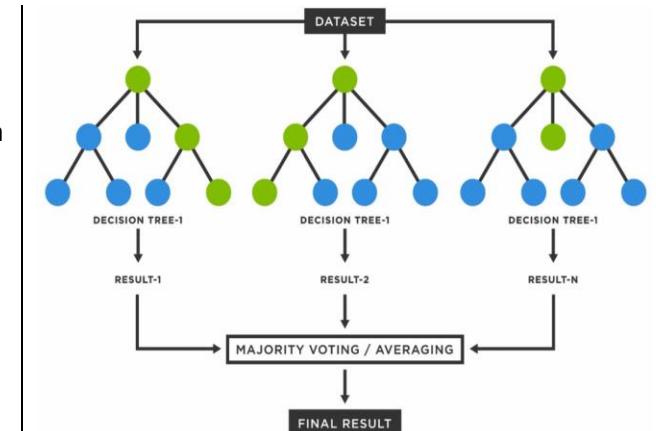
RANDOM FORESTS (TECNICA DI ENSEMBLE LEARNING):

Un'altra tecnica usata con gli alberi di decisione sono i **random forest**. L'idea è costruire tanti alberi di decisione differenti tra loro in modo che gli attributi non si presentano sempre nello stesso ordine ma vengono mischiati casualmente, siccome in base all'ordine degli attributi abbiamo un comportamento differente. I dati di training sono gli stessi ma la differenza sta come vengono costruiti gli alberi.

Il modello della random forest è una forma di **bagging di alberi decisionali** in cui adottiamo ulteriori passaggi per rendere l'insieme di alberi K più diversificato, per ridurre la **varianza**.

Le foreste casuali possono essere utilizzate per la classificazione o la regressione.

L'idea chiave è di variare casualmente le scelte degli attributi (piuttosto che gli esempi di addestramento).

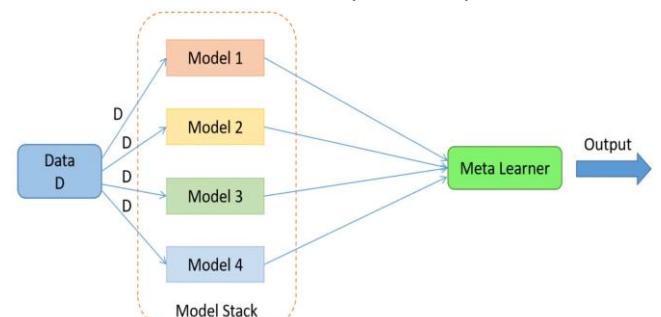


STACKING (TECNICA DI ENSEMBLE LEARNING):

Un'altra tecnica è lo **stacking** (o generalizzazione impilata), si costruisce uno stack dove abbiamo più livelli per classificare.

In questo modello, i dati sono sempre gli stessi (al contrario della tecnica precedente) e si va a costruire usando modelli differenti (es. albero di decisione, modello logistico...), dopodiché viene costruito e allenato un altro modello che prende l'output di ogni modello usato.

Questo modello di solito riduce il **bias**, ovvero riducendo l'errore di classificazione.

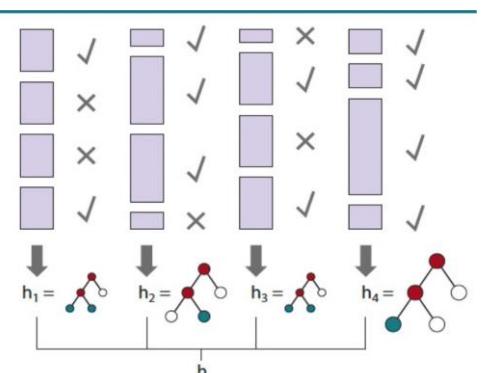


BOOSTING (TECNICA DI ENSEMBLE LEARNING):

Il modello più utilizzato è il **boosting**, l'idea è introdurre dei pesi. Si costruisce un modello h_1 , lo si testa e si evidenziano i campioni che ha sbagliato a classificare, così va a costruire un altro modello h_2 in cui i campioni mal classificati in h_1 hanno un peso maggiore. In sostanza si costruisce man mano un nuovo modello che riesca a classificare correttamente i campioni classificati male al passo precedente. Il numero di classificatori (ipotesi h_i) è un parametro.

Più formalmente, abbiamo un training set pesato, in cui ogni esempio ha un peso associato $W_j \geq 0$ che descrive quanto l'esempio dovrebbe contare durante il training.

Le ipotesi che si sono comportate meglio nel training hanno un peso maggiore nella votazione.



13. APPRENDIMENTO STATISTICO

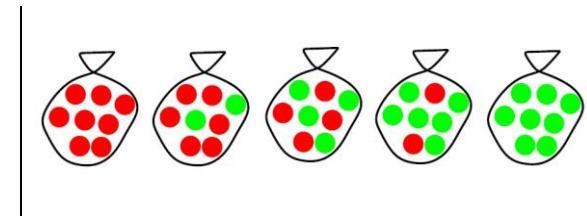
Vogliamo fare apprendimento ma abbiamo bisogno di:

- **Dati:** istanziazioni di alcune o tutte le variabili casuali che descrivono il dominio. Rappresentato dalle prove;
- **Ipotesi:** teorie probabilistiche di come funziona un dominio.

Esempio:

Supponiamo ci siano cinque tipi di sacchetti di caramelle:

- h_1 : 100% caramelle alla ciliegia;
- h_2 : 75% caramelle alla ciliegia + 25% caramelle al lime;
- h_3 : 50% caramelle alla ciliegia + 50% caramelle al lime;
- h_4 : 25% caramelle alla ciliegia + 75% caramelle al lime;
- h_5 : 100% caramelle al lime.



Il problema è che, dati un insieme di caramelle, supponiamo di aver preso un sacchetto a caso ed estratto una serie di caramelle, qual è la probabilità di estrarre una caramella a ciliegia o lime?

Questo è un esempio di apprendimento probabilistico, ovvero si apprende dai campioni presi dal sacchetto.

FORMULAZIONE DEL PROBLEMA:

Dato un nuovo sacchetto:

- Una variabile d'ipotesi H con valori h_1, h_2, \dots, h_5 denota il tipo di sacchetto;
- D_i è una variabile casuale (ciliegia o lime);
- Dopo aver visto D_1, D_2, \dots, D_N vogliamo predire il sapore (ossia il valore) di D_{n+1} .

APPENDIMENTO BAYESIANO COMPLETO:

Considera l'**apprendimento bayesiano** di una distribuzione di probabilità nello **spazio delle ipotesi** ($P(h_1), P(h_2), \dots$). Calcola la probabilità di ogni ipotesi h_i dai dati forniti e su questa base formula delle predizioni.

Cioè, le previsioni sono fatte usando tutte le ipotesi, pesate dalle loro probabilità, piuttosto che usando solo una singola ipotesi "migliore". In questo modo, l'apprendimento è ridotto a un'inferenza probabilistica.

H variabile d'ipotesi, con valori $h_1, h_2, \dots, P(H)$ distribuzione a priori.

La **j-esima osservazione d_j** fornisce il risultato della variabile casuale D_j (ciliegia o lime) partendo dai dati di training $d=d_1, \dots, d_n$ precedentemente in possesso.

Partendo dai dati disponibili fino a questo momento, la probabilità condizionata di ogni ipotesi si calcola:

$$P(h_i | d) = \alpha P(d | h_i) P(h_i)$$

(Applicando la regola di Bayes, la probabilità di avere quella sequenza di estrazione data una certa ipotesi, moltiplicata per la probabilità che si presenti quell'ipotesi)

Dove $P(d | h_i)$ viene chiamato **verosimiglianza** e d sono valori osservati da D , la quale ci dice, data una sequenza quanto si accorda con l'ipotesi. Ad esempio, se su 10 caramelle si prendono 8 lime e 2 ciliegie, allora si accorda sugli ultimi sacchetti. In generale, possiamo generalizzare la predizione a qualsiasi X sconosciuta:

$$\begin{aligned} P(X | d) &= \sum_i P(X | d, h_i) P(h_i | d) \\ &= \sum_i P(X | h_i) P(h_i | d) \\ &= \sum_i P(X | h_i) P(d | h_i) P(h_i) / P(d) \end{aligned}$$

Data una qualsiasi variabile X e un dataset d , possiamo dire qual è la probabilità che X assuma un certo valore, effettuando la sommatoria su tutte le possibili ipotesi (le quali sono variabili nascoste).

Assumendo che h_i determini una distribuzione di probabilità su X . Le predizioni sfruttano una probabilità media ponderata sulla verosimiglianza rispetto alle ipotesi.

Esempio:

Una distribuzione per $P(h_i)$ è $P(h_1, h_2, h_3, h_4, h_5) = <0.1, 0.2, 0.4, 0.2, 0.1>$. Se supponiamo l'estrazione di caramelle:



Di che tipo di sacchetto si tratta? Quale sarà il sapore della prossima caramella?

La verosimiglianza dei dati è calcolata partendo dal presupposto che le osservazioni siano **indipendentemente e identicamente distribuite** così che:

$$P(d | h_i) = \prod_j P(d_j | h_i)$$

Supponiamo l'estrazione di caramelle di alcuni sacchetti, lime come sopra $P(d | h_3) = 0.5^{10}$ perché h_3 la metà delle caramelle sono lime. Mentre se si calcola $P(d | h_4) = 0.75^{10}$.

PROBABILITÀ CONDIZIONATA DELLE IPOTESI:

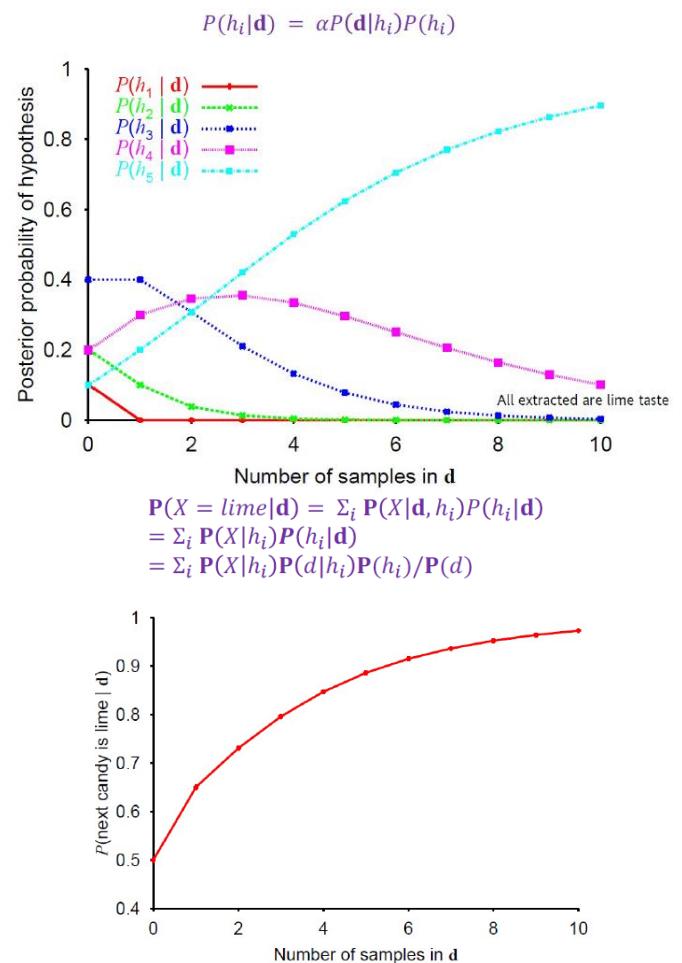
Dal grafico vediamo come cambiano le probabilità per le 5 ipotesi, considerando il vettore di 10 caramelle lime. Le probabilità cambiano all'aumentare delle caramelle estratte.

Se voglio conoscere la probabilità che la prossima caramella sia lime dato il vettore d , la curva di probabilità cresce all'aumentare dei campioni.

Le ipotesi vere alla fine hanno dominato la predizione bayesiana (caratteristica di questo tipo di apprendimento).

La predizione bayesiana è **ottimale**; tuttavia, il suo spazio delle ipotesi spesso è molto grande o addirittura infinito.

Il problema è dato dal numero di ipotesi, che possono essere molte e quindi abbiamo moltissimi valori, richiederanno molto tempo di computazione. Ciò che si fa è un compromesso, ovvero perdere precisione ma ottenere un calcolo più rapido, utilizzando l'approssimazione MAP.



APPROXIMAZIONE MAP:

Invece di analizzare tutte le ipotesi qui andiamo a considerare solo l'unica ipotesi che massimizza il prodotto successivo.

Apprendimento **Maximum a posteriori (MAP)**:

$$h_{\text{map}} \text{ è } h_i \text{ che massimizza } P(h_i | d) \cong P(d | h_i)P(h_i)$$

Le predizioni con h_{map} sono approssimativamente bayesiane $P(X | d) \cong P(X | h_{\text{map}})$, trovare le ipotesi MAP è molto più semplice dell'apprendimento bayesiano.

Esempio:

Nell'esempio $h_{\text{map}}=h_5$ dopo aver mangiato 3 caramelle a lime.

Quindi un agente MAP predirà che la quarta caramella sia lime con probabilità 1 (0,8 è invece la predizione bayesiana) all'aumentare dei dati si avvicina a quella bayesiana.

Entrambe le tecniche fanno uso della distribuzione a priori $P(h_i)$ per ridurre la complessità mentre per le ipotesi deterministiche $P(d | h_i)$ vale 1 se consistente, 0 altrimenti $\rightarrow h_{\text{map}} = \text{l'ipotesi più semplice consistente con i dati}$.

Apprendimento **Maximum a posteriori (MAP)**:

$$h_{\text{map}} \text{ è } h_i \text{ che massimizza } P(h_i | d) \cong P(d | h_i)P(h_i)$$

Equivale a minimizzare $-\log_2 P(d | h_i) - \log_2 P(h_i)$ dove:

- $\log_2 P(h_i)$ equivale al numero di bit necessari a specificare l'ipotesi h_i ;
- $\log_2 P(d | h_i)$ numero di bit aggiuntivi richiesti per la specifica dei dati fissata l'ipotesi h_i .

L'apprendimento MAP sceglie h_i che più comprime i dati, detta anche **minimum description length (MDL)**; se consideriamo l'esempio di prima $\log_2 P(h_5) = \log_2 1 = 0$ non serve alcun bit.

APPROXIMAZIONE ML (MAXIMUM LIKELIHOOD):

Analizzando la formula del MAP, nei problemi reali molto spesso, tutte le ipotesi sono equiprobabili, quindi viene rimosso l'elemento $P(h_i)$. Questo è ragionevole quando non c'è motivo di preferire un'ipotesi rispetto ad un'altra.

Per dataset di grandi dimensioni, la distribuzione a priori $P(h_i)$ diventa irrilevante. L'apprendimento con **massima verosimiglianza** (Maximum Likelihood) rappresenta una buona approssimazione dell'apprendimento bayesiano e di MAP:

- Si sceglie h_{ML} massimizzando $P(d | h_i)$, ottenendo in maniera semplice il migliore adattamento ai dati, **ipotesi di massima verosimiglianza**.

È identico al MAP per la distribuzione a priori, laddove però essa risulti **uniforme** (che è ragionevole se tutte le ipotesi sono della stessa complessità). ML rappresenta il metodo "standard" non bayesiano per l'apprendimento statistico.

APPRENDIMENTO PARAMETRI ML NELLE RETI BAYESIANE:

Nelle reti bayesiane, abbiamo dei nodi con delle tabelle di probabilità condizionate dai valori dei genitori. Possiamo quindi effettuare un apprendimento dei valori di probabilità della rete, analizzando dei dati di training.

Esempio:

Abbiamo un sacchetto da un nuovo produttore; frazione θ di caramelle alla ciliegia? Qualsiasi θ è possibile: continuum d'ipotesi h_θ dove θ è un **parametro** per questa famiglia di modelli semplici. È ragionevole adottare l'approccio ML (dato che i due gusti sono ugualmente probabili).

Supponiamo di scartare N caramelle, c alla ciliegia e $\ell=N-c$ al lime.

Queste sono osservazioni indipendenti e identicamente distribuite, pertanto:

$$P(\mathbf{d}|h_\theta) = \prod_{j=1}^N P(d_j|h_\theta) = \theta^c \cdot (1-\theta)^\ell$$

Massimizzandolo con riferimento a θ , che risulta più facile per la verosimiglianza logaritmica:

$$\begin{aligned} L(\mathbf{d}|h_\theta) &= \log P(\mathbf{d}|h_\theta) = \sum_{j=1}^N \log P(d_j|h_\theta) = c \log \theta + \ell \log(1-\theta) \\ \frac{dL(\mathbf{d}|h_\theta)}{d\theta} &= \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \quad \Rightarrow \quad \theta = \frac{c}{c+\ell} = \frac{c}{N} \end{aligned}$$

Problema: alcuni eventi potrebbero avere valore 0 qualora non fossero stati osservati.

Per massimizzare, andiamo a fare la derivata rispetto a theta, eguagliando a 0, così da ottenere il valore di theta.

PARAMETRI MULTIPLI:

L'incartamento rosso/verde dipende probabilisticamente dal sapore.

Probabilità (ad es.) di avere caramelle alla ciliegia nella carta verde:

$$\begin{aligned} P(F = \text{cherry}, W = \text{green} | h_{\theta, \theta_1, \theta_2}) &= P(F = \text{cherry} | h_{\theta, \theta_1, \theta_2}) P(W = \text{green} | F = \text{cherry}, h_{\theta, \theta_1, \theta_2}) \\ &= \theta \cdot (1 - \theta_1) \end{aligned}$$

N caramelle, r_c caramelle alla ciliegia in carta rossa, ecc...

$$P(\mathbf{d}|h_{\theta, \theta_1, \theta_2}) = \prod_{j=1}^N P(d_j|h_{\theta, \theta_1, \theta_2})$$

$$\begin{aligned} P(\mathbf{d}|h_{\theta, \theta_1, \theta_2}) &= \theta^c (1-\theta)^\ell \cdot \theta_1^{r_c} (1-\theta_1)^{g_c} \cdot \theta_2^{r_\ell} (1-\theta_2)^{g_\ell} \\ L &= [c \log \theta + \ell \log(1-\theta)] \\ &\quad + [r_c \log \theta_1 + g_c \log(1-\theta_1)] \\ &\quad + [r_\ell \log \theta_2 + g_\ell \log(1-\theta_2)] \end{aligned}$$

Le derivate di L contengono solo i parametri rilevanti:

$$\frac{\partial L}{\partial \theta} = \frac{c}{\theta} - \frac{\ell}{1-\theta} = 0 \quad \Rightarrow \quad \theta = \frac{c}{c+\ell}$$

$$\frac{\partial L}{\partial \theta_1} = \frac{r_c}{\theta_1} - \frac{g_c}{1-\theta_1} = 0 \quad \Rightarrow \quad \theta_1 = \frac{r_c}{r_c + g_c}$$

$$\frac{\partial L}{\partial \theta_2} = \frac{r_\ell}{\theta_2} - \frac{g_\ell}{1-\theta_2} = 0 \quad \Rightarrow \quad \theta_2 = \frac{r_\ell}{r_\ell + g_\ell}$$

Con dati completi, i parametri possono essere appresi separatamente.

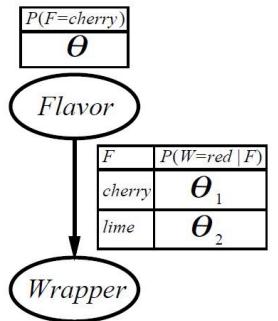
MODELLI BAYESIANE NAIVE:

Rappresentano il modello di rete bayesiana comunemente usato nel machine learning, un solo nodo radice e tanti figli.

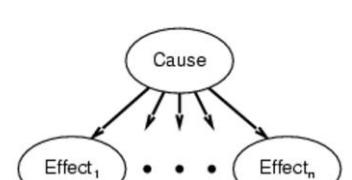
Viene fatta un'assunzione, dove gli effetti sono tutti indipendenti l'uno dall'altro (condizionalmente indipendenti).

La variabile C da predire è la radice, le variabili attributo x_i sono le foglie. Questo modello è ingenuo perché assume che gli attributi sono condizionalmente indipendenti.

Una predizione deterministica può essere ottenuta scegliendo le classi più probabili.

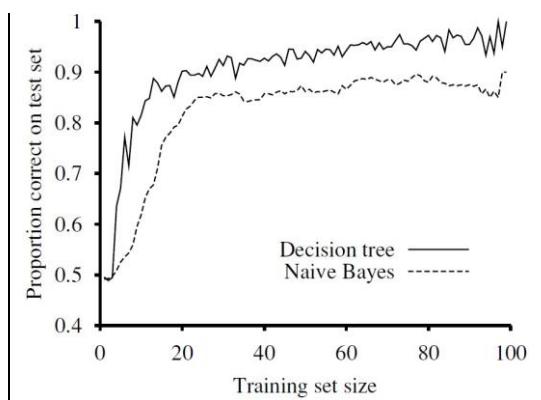


$$P(C|x_1, x_2, \dots, x_n) = \alpha P(C) \prod_i P(x_i|C)$$



Il loro comportamento è leggermente peggiore degli alberi di decisione.

Non ha nessuna difficoltà con i dati rumorosi, per n attributi booleani, ci sono $2n+1$ parametri, non è richiesta nessuna ricerca per trovare h_{ML} .



RIASSUNTO:

L'apprendimento bayesiano formula un apprendimento come una forma d'inferenza probabilistica, usando le osservazioni per aggiornare una distribuzione a priori attraverso le ipotesi.

L'apprendimento MAP seleziona una singola ipotesi più probabile, sfruttando i dati di training $P(\mathbf{d}|h_i), P(h_i)$.

Il metodo della massima verosimiglianza (ML) seleziona le ipotesi che massimizzano la verosimiglianza dei dati (uguale a MAP ma con una distribuzione a priori uniforme) $P(\mathbf{d}|h_i)$.

14. RETI NEURALI

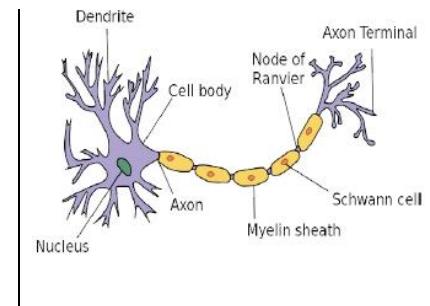
L'idea delle reti neurali è nata come modello che simulava il funzionamento dei neuroni nel cervello.

I circuiti collegati sono stati utilizzati per simulare il suo comportamento intelligente.

Il cervello è composto da neuroni:

- un corpo cellulare
- dendriti (ingressi)
- un assone (uscite)
- sinapsi (elaborano l'input e creano l'output), possono essere stimolate o inibite e possono cambiare nel tempo.

Quando la somma degli ingressi raggiunge una certa soglia, verrà inviato un impulso elettrico sull'assone.



McCULLOCH-PITTS UNIT (RETI NEURALI):

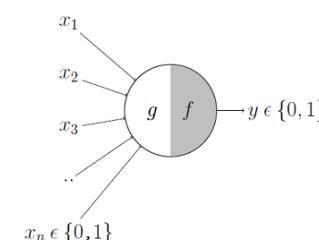
Nel 1943 McCulloch e Pitts proposero il primo modello di neurone ed era molto elementare.

Un'**unità McCulloch Pitts** prende come input valori booleani, li dà ad una funzione che calcola la somma su tutti gli input booleani e poi, il risultato della sommatoria, viene dato ad una funzione di attivazione g che ritorna 1 o 0 in base ad una soglia, quando il valore della sommatoria supera la soglia ritorna 1 altrimenti 0.

Una **rete neurale artificiale** è un **grafo diretto** di unità e collegamenti. Un link dall'unità i all'unità j propaga l'attivazione a_i dall'unità i all'unità j , ed ha un peso $w_{i,j}$ ad essa associato.

$$g(x_1, x_2, x_3, \dots, x_n) = g(\mathbf{x}) = \sum_{i=1}^n x_i$$

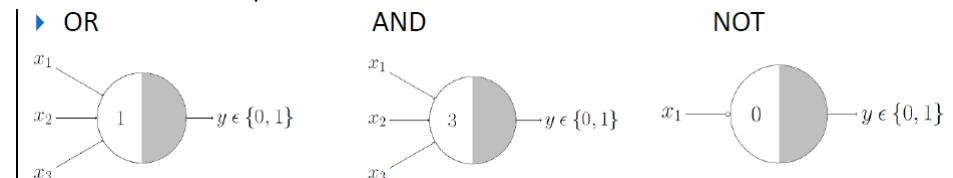
$$\begin{aligned} y = f(g(\mathbf{x})) &= 1 \quad \text{if} \quad g(\mathbf{x}) \geq \theta \\ &= 0 \quad \text{if} \quad g(\mathbf{x}) < \theta \end{aligned}$$



IMPLEMENTAZIONE DI FUNZIONI LOGICHE COME UNITÀ:

Le unità di McCulloch Pitts sono una grossolana semplificazione eccessiva dei neuroni reali, ma il suo scopo è sviluppare la comprensione di ciò che possono fare le reti neurali di unità semplici.

Ogni **funzione booleana** può essere implementata come **reti McCulloch Pitts**:



PERCETTRONE:

Il problema con i neuroni di McCulloch e Pitts sono:

- Modello troppo semplice, non si può modellare qualcosa di più complesso, siccome la rete non accetta dati continui (non booleani);
- Questo tipo di rete prevede dei pesi e delle soglie che sono definite assieme alla rete (non è possibile apprenderli e cambiarli).

L'evoluzione di questo modello è il **percettrone** che risolve diversi problemi precedenti:

- I valori sono continui, bipolare e multivalore;
- I valori dei pesi e delle soglie possono essere determinati analiticamente o mediante un algoritmo di apprendimento. La formula di aggiornamento dei pesi è la seguente:

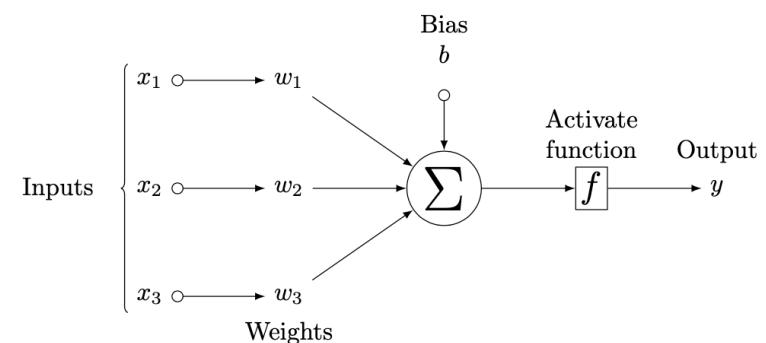
$$w_j^{(t)} \leftarrow w_j^{(t)} + \alpha x_j(t - y)$$

dove t l'output corretto e y la funzione di output della rete mentre α è il fattore di apprendimento

Il Percettrone ha la seguente architettura:

L'input è x_1, \dots, x_n (non booleani) con i pesi associati w_1, \dots, w_n .

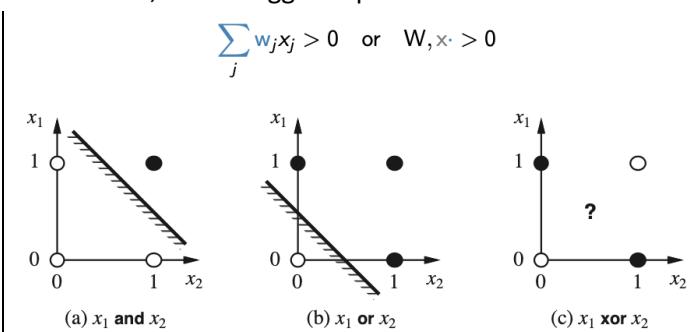
Una volta presi input e pesi, effettua una sommatoria pesata (moltiplicando ogni input per il peso e poi va a sommare), dopodiché il risultato viene dato ad una funzione di attivazione che produrrà l'output.



ESPRESSIVITÀ DEI PERCETTRONI:

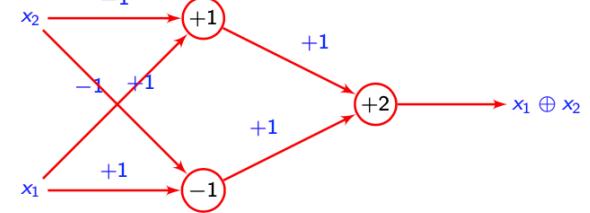
Uno dei problemi del **percettrone** è l'espressività, dove può rappresentare AND, OR, NOT, maggioranza, ecc. (separatori lineari), ma non lo XOR (e quindi nessun sommatore). Per risolvere tutto ciò, si sono aggiunti più strati.

Rappresenta un **separatore lineare** nello spazio di input:



Il seguente Perceptron multistrato può risolvere il problema dello XOR:

Lo strato intermedio è uno strato nascosto.



RETI FEED-FORWARD (STRUTTURE DI RETI):

Quindi per affrontare il problema dell'espressività si vanno a creare le reti feed-forward, fondamentalmente si possono collegare tanti percetroni organizzandoli in layer. Si parte con dei layer di input che ricevono i dati x_n del problema e si aggiungono tanti strati dove all'interno ci sono sempre dei percetroni.

Si chiama **feed-forward** se la rete è aciclica, perché l'output di un percettrone viene dato in input al prossimo percettrone, in questo modo i dati non vengono riusati per altri calcoli, pertanto, si dice che sono reti "senza memoria".

Più formalmente, le reti feed forward sono organizzate in livelli (layers): una rete a n -livelli ha una partizione $\{L_0, \dots, L_n\}$ dei nodi, in modo tale che gli archi colleghino solo i nodi al livello successivo.

L_0 è chiamato **livello di input** ed i suoi elementi **unità di input**, e L_n **livello di output** ed i suoi elementi **unità di output**.

Qualsiasi unità che non si trova nel livello di input o nel livello di output viene chiamata **hidden**.

RETI RICORRENTI (STRUTTURE DI RETI):

Mentre, una rete neurale si chiama **ricorrente** se ha dei cicli, come se la rete abbia una memoria siccome alcuni percetroni possono usare i risultati di altri percetroni prendendo una decisione anche in base al risultato delle operazioni precedenti.

PERCETTRONI SINGLE-LAYER:

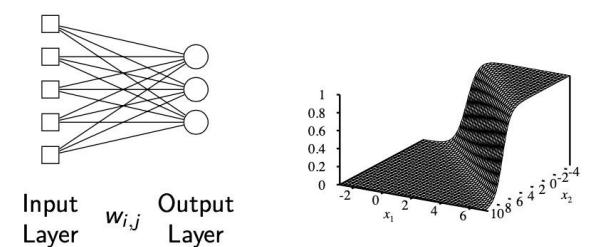
Il modello più semplice di una rete neurale è la **rete perceptron**, che è una rete feed forward con un solo layer che produce direttamente un valore di output. Una rete perceptron a strato singolo è chiamata semplicemente **perceptron**.

NOTA: il layer di output è un problema di classificazione non binaria multiclasse, siccome ogni nodo di output rappresenta una classe. Quindi un solo nodo è una classificazione binaria, con più nodi ci sono più classi.

Tutte le unità di ingresso sono collegate direttamente all'unità di uscita.

Le unità di output funzionano tutte separatamente, nessun peso condiviso, e sono trattate come la combinazione di n unità percetroni.

La **regolazione dei pesi** è la componente che fa cambiare il funzionamento della rete, esempio, sposta la posizione, l'orientamento e la pendenza della scogliera.



Esempio reti neurali Feed-Forward:

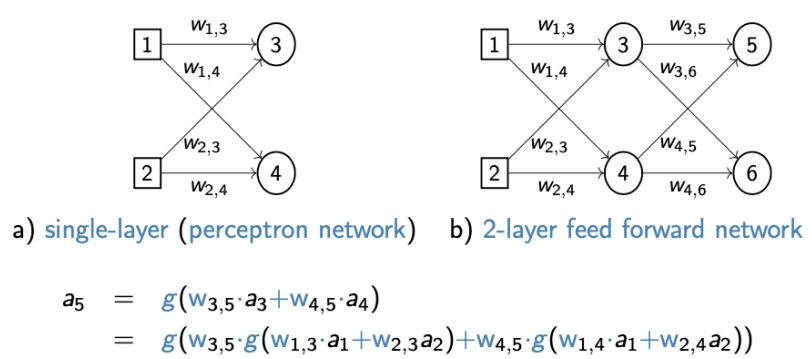
Feed forward network = una famiglia parametrizzata di funzioni non lineari:

Mostriamo due reti feed forward:

- Ad un livello;
- A due livelli.

Come mostrato, per determinare il valore del nodo 3 si calcola la funzione di attivazione con gli output dei nodi 1 e 2 con i rispettivi pesi $w_{1,3}$ e $w_{2,3}$.

NOTA: la regolazione dei pesi cambia la funzione



$$a_5 = g(w_{3,5} \cdot a_3 + w_{4,5} \cdot a_4)$$

$$= g(w_{3,5} \cdot g(w_{1,3} \cdot a_1 + w_{2,3} \cdot a_2) + w_{4,5} \cdot g(w_{1,4} \cdot a_1 + w_{2,4} \cdot a_2))$$

FUNZIONE DI ATTIVAZIONE:

Una **funzione di attivazione** viene aggiunta ad una rete neurale per aiutarla ad apprendere schemi complessi nei dati, essa converte l'output di un neurone in un'altra forma utilizzata come input per il neurone successivo.

Le funzioni di attivazione sono necessarie perché devono capire l'andamento dei dati e dovrebbero essere funzioni non lineari, perché con funzioni lineari tutta la rete neurale si semplifica in una funzione lineare e molti problemi non sono modellabili in questo modo, e devono mantenere l'output di un neurone a un certo intervallo secondo il nostro requisito.

Caratteristiche desiderabili di una funzione di attivazione sono le seguenti proprietà:

- **Gradiente non nullo:** Formiamo reti neurali utilizzando algoritmi basati su gradiente. Quindi, il gradiente deve essere non zero in tutti i punti di dominio. Se il gradiente tende a zero perderemo informazioni e l'apprendimento non funzionerà come deve (problema del *vanishing gradient*);
- **Centrato sullo zero:** L'uscita di una funzione di attivazione deve essere simmetrica a zero. Questo impedisce ai gradienti di spostarsi in una direzione particolare;
- **Costi computazionali:** Le funzioni di attivazione vengono applicate più volte. Dovrebbero essere computazionalmente poco costose per calcolarle insieme alla sua derivata;
- **Differenziabile:** Nell'apprendimento basato sul gradiente, dobbiamo calcolare il gradiente di attivazione funzioni. Pertanto, le funzioni di attivazione devono essere differenziabili.

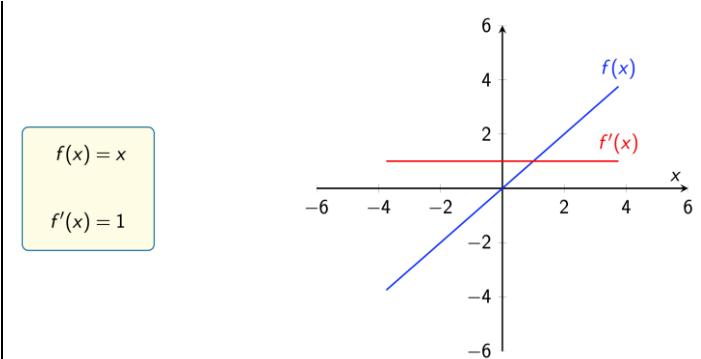
FUNZIONE DI ATTIVAZIONE LINEARE:

Vantaggi:

- Il suo calcolo è semplice.
- La sua derivata è diversa da zero.

Svantaggi:

- L'output non è in un intervallo.
- Nessun risultato nella non linearità.



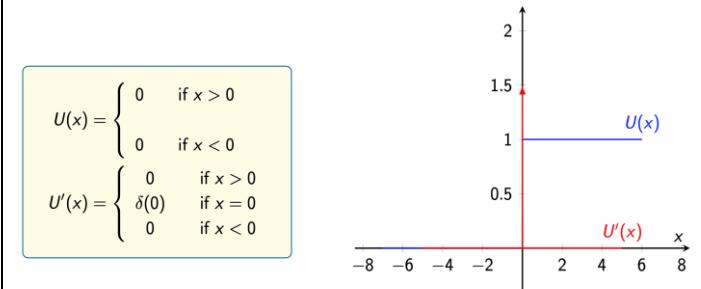
FUNZIONE DI ATTIVAZIONE SOGLIA:

Vantaggi:

- Il suo calcolo è semplice.

Svantaggi:

- L'output non è in un intervallo.
- La sua derivata è zero tranne che all'origine.



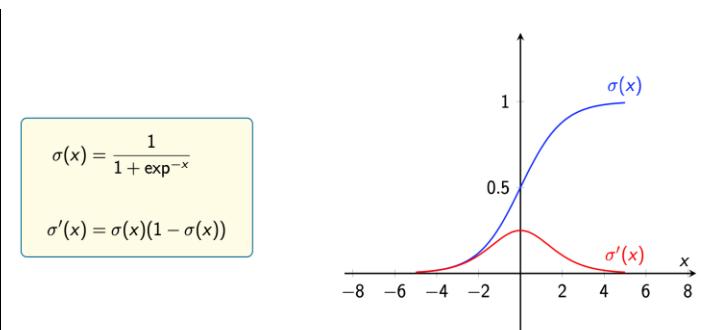
FUNZIONE DI ATTIVAZIONE SIGMOIDE:

Vantaggi:

- L'output è nell'intervallo $[0, 1]$.
- È differenziabile.

Svantaggi:

- Satura e uccide i gradienti (lo porta a 0 siccome applica continuamente la derivata).
- Il suo output non è centrato sullo zero (ma su 0,5).



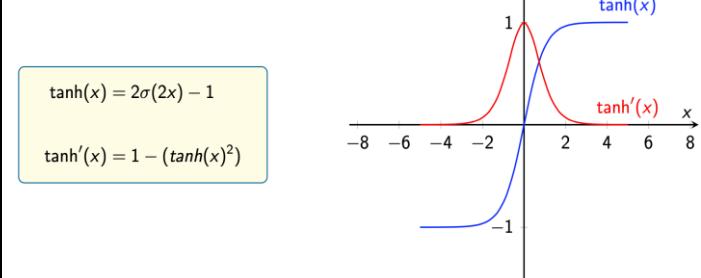
FUNZIONE DI ATTIVAZIONE TANH:

Vantaggi:

- L'output è nell'intervallo $[0, 1]$.
- È differenziabile.
- Il suo output è centrato sullo zero.

Svantaggi:

- Satura e uccide i gradienti.



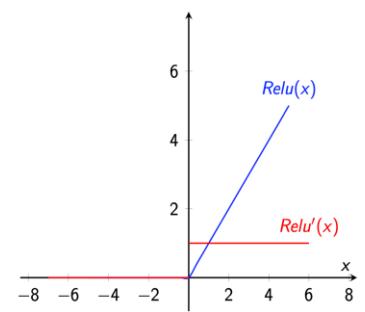
FUNZIONE DI ATTIVAZIONE RELU:

Vantaggi:

- Il suo calcolo è semplice.
 - La sua derivata è diversa da zero quando $x > 0$ (identità).
 - Non è computazionalmente costosa.
- Svantaggi:
- L'output non è in un intervallo.
 - Il gradiente svanisce quando $x < 0$.

$$\text{Relu}(x) = \max(0, x)$$

$$\text{Relu}'(x) = \begin{cases} 1 & \text{if } x > 0 \\ ? & \text{if } x = 0 \\ 0 & \text{if } x < 0 \end{cases}$$



FUNZIONE DI ATTIVAZIONE SOFTMAX:

Softmax è una forma più generalizzata del sigmoide, viene utilizzato nel livello di output delle reti neurali per problemi di classificazione multiclasse.

Esempio:

Sia $x = [1.60, 0.55, 0.98]^\top$.

Applicando softmax si ottiene $a_i = [0.51, 0.18, 0.31]^\top$.

$$a_i(x) = \frac{\exp^{z_i}}{\sum_k \exp^{z_k}}$$

FUNZIONE DI LOSS:

L'obiettivo degli algoritmi di machine learning è costruire un modello (ipotesi) che possa essere utilizzato per stimare t in base a x . Sia il modello in forma di (che va a modellare il funzionamento del neurone):

$$h(x) = w_0 + w_1 x$$

L'obiettivo della creazione di un modello è scegliere parametri in modo che $h(x)$ sia vicino a t per i dati di training x .

Abbiamo sempre bisogno di minimizzare la funzione di loss, ovvero la distanza tra la funzione reale e ciò che ha stimato sia il più piccolo valore possibile. Una funzione che è spesso usato è l'**errore quadratico medio**:

$$J(w) = \frac{1}{2m} \sum_{i=1}^m (h(x_i) - t_i)^2$$

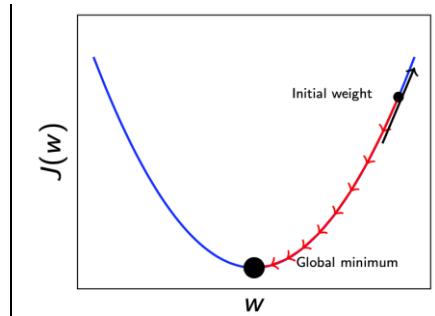
DISCESA DEL GRADIENTE:

Per minimizzare la loss function, come nella regressione, determiniamo il gradiente e modifichiamo i valori dei pesi affinché la loss diminuisca; quindi, andiamo nella direzione opposta del gradiente.

La discesa del gradiente è di gran lunga la strategia di ottimizzazione più popolare, utilizzata al momento nell'apprendimento automatico e nel deep learning.

Il costo (errore) è una funzione dei pesi (parametri).

Vogliamo ridurre al minimo l'errore (punto *initial weight*), il gradiente restituisce la direzione per salire ma andiamo nella direzione opposta, scendendo passo per passo fino a che non raggiungiamo il minimo w .



Abbiamo la seguente ipotesi e dobbiamo adattare i dati di training:

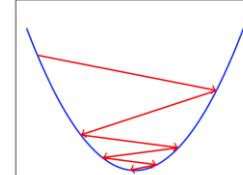
$$h(x) = w_0 + w_1 x$$

Usiamo una funzione di loss come **Errore quadratico medio**:

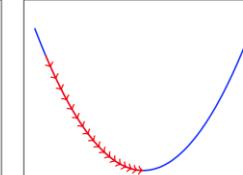
$$J(w) = \frac{1}{2m} \sum_{i=1}^m (h(x_i) - t_i)^2$$

Questa funzione di loss può essere ridotta al minimo utilizzando la discesa del gradiente con il **tasso di apprendimento** α , con un valore di tasso alto passiamo da una curva all'altra mentre con un piccolo tasso si ha un lento aggiornamento dei pesi fino ad un minimo.

Big step size



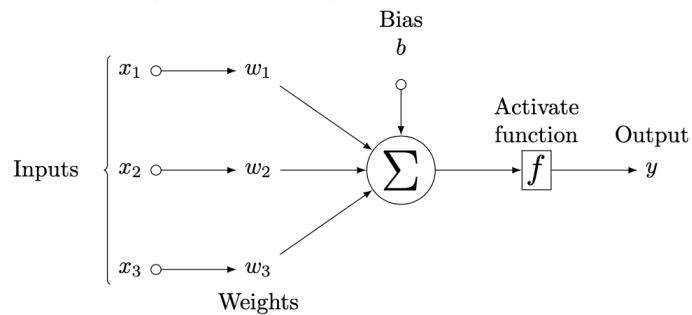
Small step size



$$\begin{aligned} w_0^{(t+1)} &= w_0^{(t)} - \alpha \frac{\partial J(w^{(t)})}{\partial w_0} \\ w_1^{(t+1)} &= w_1^{(t)} - \alpha \frac{\partial J(w^{(t)})}{\partial w_1}, \end{aligned}$$

ALLENARE UN NEURONE CON ATTIVAZIONE SIGMOIDEA (REGRESSIONE):

Consideriamo il seguente singolo neurone:



Vogliamo addestrare questo neurone per ridurre al minimo la seguente funzione di costo:

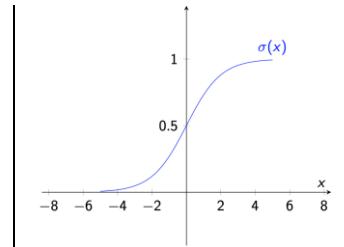
$$J(w) = \frac{1}{2m} \sum_{i=1}^m (h(x^i) - t^i)^2$$

Considerando la *funzione di attivazione sigmoidea*:

$$f(z) = \frac{1}{1+e^{-z}}$$

Vogliamo calcolare:

$$\frac{\partial J(w)}{\partial w_i}$$



Usando la **chain rule**, otteniamo (α è il tasso di apprendimento):

$$\begin{aligned}\frac{\partial J(w)}{\partial w_j} &= \frac{\partial J(w)}{\partial f(z)} \times \frac{\partial f(z)}{\partial z} \times \frac{\partial z}{\partial w_j} \\ \frac{\partial J(w)}{\partial f(z^i)} &= \frac{1}{m} \sum_{i=1}^m (f(z^i) - t^i) \\ \frac{\partial f(z)}{\partial z} &= \frac{e^{-z}}{(1 + e^{-z})^2} = f(z)(1 - f(z)) \\ \frac{\partial z}{\partial w_j} &= x^j \\ w_j^{(t+1)} &= w_j^{(t)} - \alpha \frac{\partial J(w)}{\partial w_j}\end{aligned}$$

Vogliamo addestrare questo neurone per ridurre al minimo la seguente funzione di costo:

$$J(w) = \sum_{i=1}^m [-t^i \ln h(x^i) - (1 - t^i) \ln(1 - h(x^i))]$$

Calcolando i gradienti di $J(w)$ rispetto a w , otteniamo:

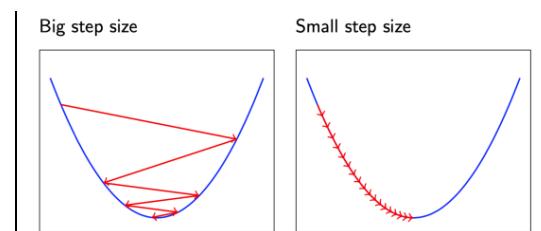
$$\nabla J(w) = \sum_{i=1}^m t^i x^i (h(x^i) - t^i)$$

L'aggiornamento del vettore di peso utilizzando la regola di discesa del gradiente risulterà:

$$w^{(t+1)} = w^{(t)} - \alpha \sum_{i=1}^m t^i x^i (h(x^i) - t^i)$$

TUNING LEARNING RATE (α):

- Se α è troppo alto, l'algoritmo diverge e il punto di minimo potrebbe saltarlo.
- Se α è troppo basso, rallenta la convergenza dell'algoritmo e l'algoritmo richiede tanti passi per arrivare al minimo.



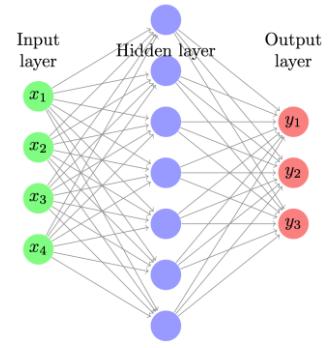
Una pratica comune consiste nel rendere α_k una funzione decrescente del numero di iterazione k :

$$\alpha_k = \frac{c_1}{k + c_2} \quad \text{dove } c_1 \text{ e } c_2 \text{ sono due costanti.}$$

Le prime iterazioni causano grandi cambiamenti nella w , mentre le successive effettuano solo il fine-tuning. Più le iterazioni aumentano più il passo diminuisce siccome si suppone che ci si trovi vicino al minimo.

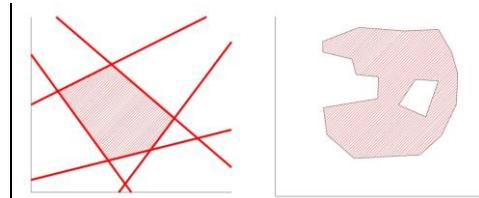
DEEP FEED-FORWARD NETWORKS:

Soltanamente, le reti neurali sono formate da più layer, pertanto, i dati di input attraverseranno tutti questi perceptri.



LA TOPOLOGIA OTTIMALE DELLE RETI:

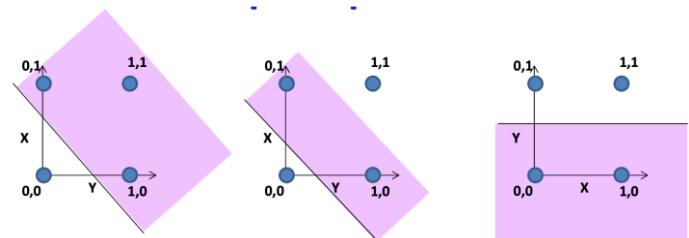
Il funzionamento delle reti neurali deep non è noto siccome ci sono tanti parametri che si aggiornano nella fase di training.
Una rete di tale livello non fa altro che creare delle approssimazioni di funzioni.



SUPERFICIE DECISIONALE DEL PERCEPTRON:

Ad esempio, definendo una rete a tre livelli, si riesce a definire un classificatore che assegna a tutti i punti dentro all'area rettangolare una determinata label e quelli al di fuori un'altra label.

Se bisogna definire una forma più complessa si necessita di più layer, andando a costruire una topologia di rete diversa.



SCELTA DELLA TOPOLOGIA DI RETE:

Specificare la **topologia della rete** in base a:

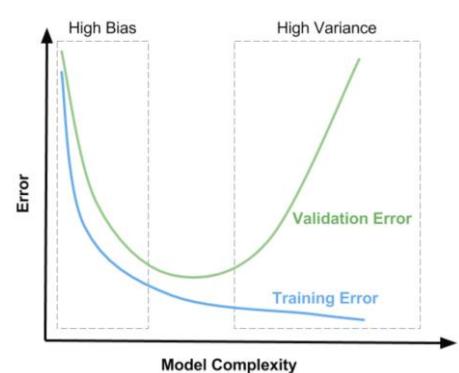
- #-strati
- #-nodi in ogni livello
- funzione di ogni nodo
- attivazione di ogni nodo

Ricordando che le reti neurali sono approssimatori universali e che bisogna specificare la funzione di loss, usando l'algoritmo discesa del gradiente per addestrare la rete.

Bisogna poi scegliere le funzioni di attivazione per i vari layer, ad esempio per l'Output layer (Linear, ReLU, Sigmoid, Softmax) e l'Hidden layers (Linear, ReLU, Sigmoid).

Un [approccio semplice](#) per la scelta della topologia di rete è per tentativi ed errori (**trial and error**). Suddividiamo i dati disponibili in tre parti: dati di training, dati di validation e dati di testing. Sceglio una topologia e addestriamo la rete utilizzando i dati di addestramento. Dopo l'addestramento, valutiamo la rete addestrata utilizzando i dati di convalida.

La parte più importante è la complessità del modello, siccome se il modello è troppo semplice va in underfitting (Bias) mentre troppo complesso va in overfitting (Varianza). Pertanto, bisogna trovare il giusto compromesso.



ALGORITMO BACKPROPAGATION:

Per l'apprendimento su una rete con un solo layer si usa l'algoritmo di **backpropagation**, è sempre l'algoritmo basato sul gradiente solo che si contestualizza ad una rete con più layer.

Il training lo si può fare in diversi modi, esempio se abbiamo un training set di 1000 campioni possiamo usare il **batch size**, ovvero quanti dati vogliamo rendere disponibili per fare training ad ogni iterazione.

L'algoritmo prende un sottoinsieme di campioni, li da in input alla rete e calcola la loss, dopodiché a partire dalla loss va ad aggiornare il valore dei pesi e lo fa in maniera iterativa. Ripete finché la differenza dei pesi si azzera o perché il **numero di iterazioni** specificato viene raggiunto.

Le **epoch** sono quante volte viene considerato l'insieme di training, siccome è possibile usarlo più volte.

Training a neural network

Data: A training set $S = \{(x_1, y_1), \dots, (x_m, y_m)\}$

Result: Weight matrices of the neural network

Initialize randomly weights in the network;

while not at trained **do**

Create a batch $S_B \subseteq S$;

Let $K \leftarrow |S_B|$;

for $i \leftarrow 1$ **to** K **do**

 Give x_i to the network and \hat{y}_i ;

end

Compute $J(w)$;

Compute $\nabla_w J(w)$;

Compute $w^{t+1} \leftarrow w^t - \alpha \nabla_w J(w)$;

end

BATCH SIZE:

La discesa del gradiente può essere utilizzata con batch di diverse dimensioni, esempio:

- $K = 1$ (Discesa stocastica del gradiente);
- $K \ll m$ (Discesa gradiente mini-batch);
- $K = m$ (Discesa gradiente batch).

Esempio:

Sia la dimensione del training set $m = 1000$, nella discesa stocastica del gradiente, utilizziamo 1000 batch di dimensione 1.

Sia $K = 50$, in discesa gradiente mini-batch, utilizziamo 20 lotti di dimensione 50, quindi dopo 20 iterazioni si fa un'epoca.

Nella discesa del gradiente batch, utilizziamo un batch di dimensione 1000.

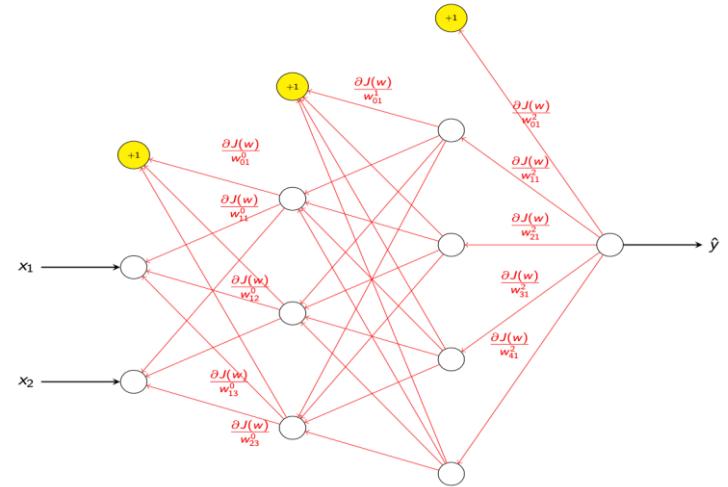
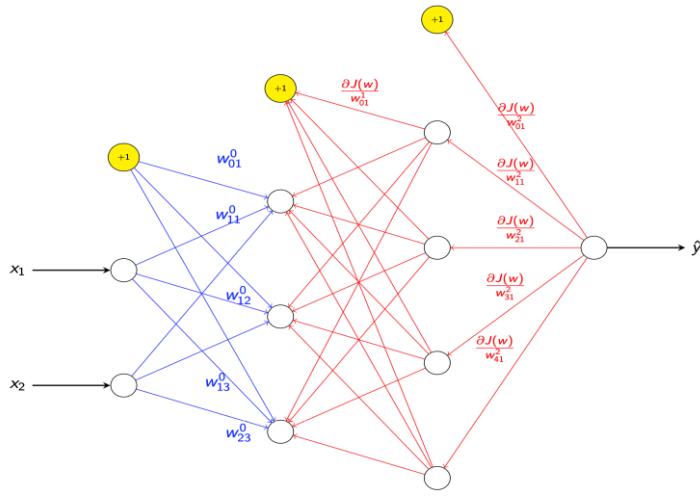
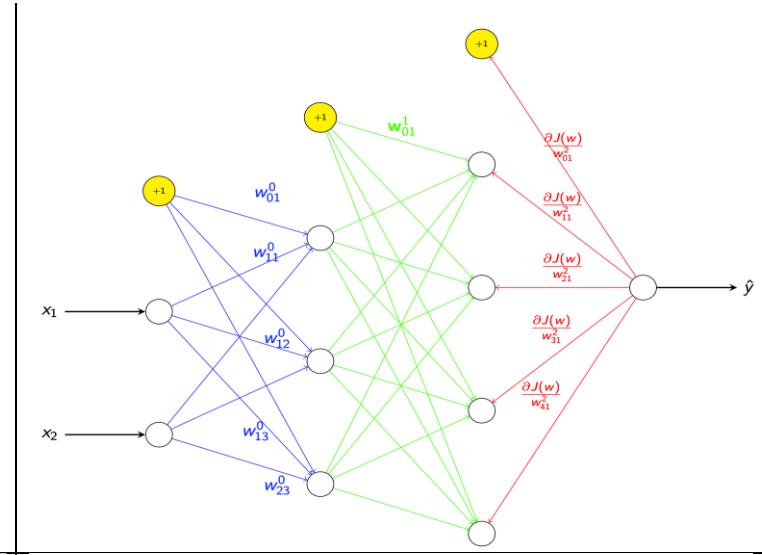
BACKWARD PASS:

La parte importante dell'algoritmo è come aggiornare i pesi nella rete. I pesi in una rete multilayer si trovano a più livelli da 0 a n , ma la loss si trova all'ultimo layer. Quindi bisogna aggiornare i pesi facendo la derivata della loss rispetto ai pesi del layer in cui ci si trova, dopodiché si aggiornano i pesi, si passa al layer successivo e così via andando avanti fino all'origine.

Dopo aver calcolato la funzione di loss, dobbiamo calcolare $\nabla_w J(w)$.

Quindi per ogni peso w , utilizziamo la seguente regola per aggiornare quel peso.

$$w^{t+1} = w^t - \alpha \frac{\partial J(w)}{\partial w}$$

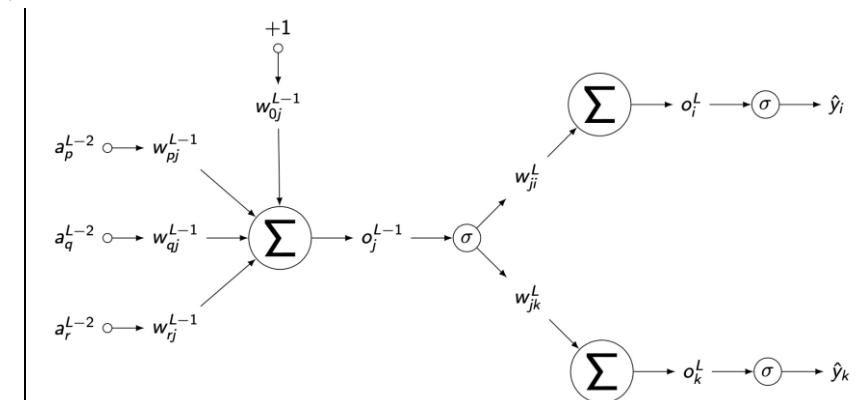


Usiamo la seguente notazione per il calcolo $\nabla_w J(w)$:

Supponiamo di avere una rete con L livelli, in cui l'ultimo livello ha nodi di output C .

Supponiamo che tutti i nodi utilizzano funzioni di attivazione del sigmoide.

Indichiamo l'output del j -esimo nodo nel layer L -esimo strato con a_j^L .



Dobbiamo calcolare $\frac{\partial J(w)}{\partial w}$

$$\frac{\partial J(w)}{\partial w_{pj}^{L-1}} = \frac{\partial \sum_{s=1}^K \sum_{c=1}^C \ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}} = \frac{\partial \sum_{c=1}^C \sum_{s=1}^K \ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}} = \sum_{c=1}^C \frac{\partial \sum_{s=1}^K \ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}}$$

Come calcolare $\sum_{c=1}^C \frac{\partial \sum_{s=1}^K \ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}}$:

$$\sum_{c=1}^C \frac{\partial \sum_{s=1}^K \ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}} = \sum_{c=1}^C \sum_{s=1}^K \frac{\partial \ell(\hat{y}_{cs}, y_{cs})}{\partial \hat{y}_{cs}} \frac{\partial \hat{y}_{cs}}{\partial o_c^L} \frac{\partial o_c^L}{\partial a_j^{L-1}} \frac{\partial a_j^{L-1}}{\partial o_j^{L-1}} \frac{\partial o_j^{L-1}}{\partial w_{pj}^{L-1}}$$

Abbiamo:

$$\sum_{c=1}^C \frac{\partial \sum_{s=1}^K \ell(\hat{y}_{cs}, y_{cs})}{\partial w_{pj}^{L-1}} = \sum_{c=1}^C \sum_{s=1}^K \frac{\partial \ell(\hat{y}_{cs}, y_{cs})}{\partial \hat{y}_{cs}} \frac{\partial \hat{y}_{cs}}{\partial o_c^L} \frac{\partial o_c^L}{\partial a_j^{L-1}} \frac{\partial a_j^{L-1}}{\partial o_j^{L-1}} \frac{\partial o_j^{L-1}}{\partial w_{pj}^{L-1}}$$

Otteniamo:

$$\begin{aligned} \frac{\partial \ell(\hat{y}_{cs}, y_{cs})}{\partial \hat{y}_{cs}} &=? \\ \frac{\partial \hat{y}_{cs}}{\partial o_c^L} &= \sigma(o_c^L) (1 - \sigma(o_c^L)) \\ \frac{\partial o_c^L}{\partial a_j^{L-1}} &= \frac{\partial \sum_l w_{lc}^L a_l^{L-1}}{\partial a_j^{L-1}} = w_{jk}^L \\ \frac{\partial a_j^{L-1}}{\partial o_j^{L-1}} &= \sigma(o_j^{L-1}) (1 - \sigma(o_j^{L-1})) \\ \frac{\partial o_j^{L-1}}{\partial w_{pj}^{L-1}} &= \frac{\partial \sum_l w_{lj}^{L-1} a_l^{L-2}}{\partial w_{pj}^{L-1}} = a_p^{L-2} \end{aligned}$$

RIASSUNTO:

Multi-layer Perceptron (MLP) utilizza l'algoritmo di backpropagation dell'errore per propagare l'errore a tutti i livelli.

MLP utilizza la discesa del gradiente (stocastico/mini-batch) per aggiornare i pesi.

La funzione di loss contiene molti minimi locali e non vi è alcuna garanzia di convergenza.

Quanto bene apprende la MLP e come possiamo migliorarla?

Quanto bene si generalizzera MLP (dati di test esterni)?

15. DEEP LEARNING

I modelli di classificazione derivati per l'apprendimento supervisionato sono semplificazioni della realtà, siccome le semplificazioni si basano su certe ipotesi. Le ipotesi falliscono in alcuni situazioni, ad esempio, a causa dell'incapacità di stimare perfettamente i parametri del modello ML da limitati dati.

TEOREMA "NO FREE LUNCH":

Qualsiasi problema che abbiamo, non esiste la possibilità di definire un modello che funziona per tutti i problemi. Vuol dire che non esiste una soluzione che funzionerà bene per tutti i tipi di problemi.

Formalmente afferma che:

Nessun singolo classificatore funziona al meglio per tutti i possibili problemi.

(Dal momento che dobbiamo fare ipotesi per generalizzare).

DIFFERENZE TRA MACHINE LEARNING (ML) E DEEP LEARNING (DL):

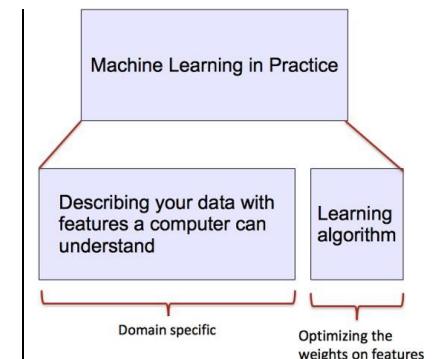
Una differenza importante riguarda le **features**.

I metodi di apprendimento automatico convenzionali si basano su **rappresentazioni di features progettate dall'uomo**. ML diventa solo l'ottimizzazione dei pesi per fare al meglio le predizioni.

Gli algoritmi visti fino ad ora si basano su caratteristiche che l'algoritmo utilizzerà per fare la classificazione. Chi va a progettare la rete neurale deve andare anche a definire quali sono le features più importanti per andare ad aiutare il processo di classificazione. Una volta definite le features, tutto il resto del processo è completamente automatico, il problema consiste poi ad andare ad ottimizzare i pesi in base a dati di esempio.

La parte che impatta sulla qualità del modello sono le features che dipendono dal tipo di problema che si va a risolvere.

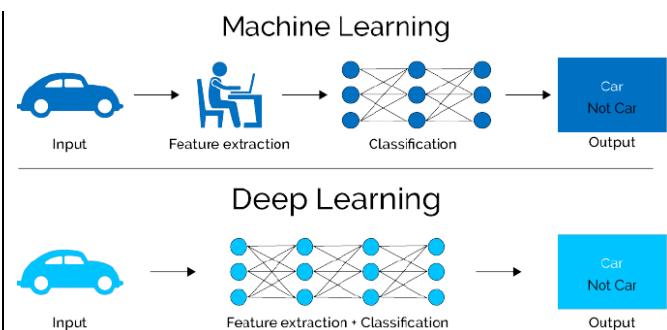
Nella pratica (come è definito dallo schema al lato), il task è dipendente dal dominio, il progettista deve avere conoscenza del dominio per capire quali sono le features utili per definire la funzione di approssimazione del modello, dopodiché si danno i dati all'algoritmo di apprendimento che viene eseguito per apprendere le caratteristiche del modello.



Il **deep learning (DL)** è un sottocampo di machine learning che utilizza più livelli per apprendere le rappresentazioni dei dati. DL è eccezionalmente efficace nell'apprendimento di pattern.

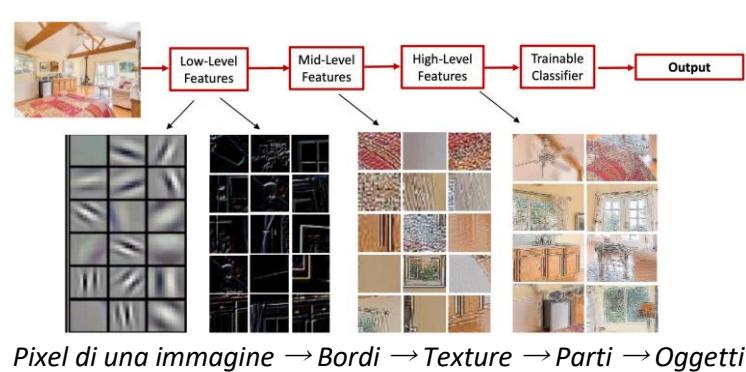
La caratteristica importante è che nel ML il programmatore deve fare l'estrazione delle features, mentre **nel DL la rete è in grado di estrarre in maniera automatica le features dai dati**, bisogna solo configurare la rete.

Le features vengono estratte dal modello grazie ai tanti strati, dove ognuno di esso si occupa di estrarre particolari caratteristiche dell'input.



Esempio DL:

Il DL viene utilizzato molto nell'ambito della visione (delle immagini), perché è un dominio che presenta tantissime features. Anziché definirle a mano, meglio utilizzare una rete profonda che lo faccia in maniera automatica, riuscendosi ad adattare in base all'immagine di input. Si parte dall'immagine di input che può essere in formato RGB, si applicano dei filtri si ottengono diverse features dal più basso (bordi) al più alto livello (oggetti).



Pixel di una immagine → Bordi → Texture → Parti → Oggetti

Esiste però un vincolo per estrarre le feautes, ovvero c'è necessità di dare in input tantissimi dati. Nell'esempio sopra, per estrarre oggetti bisogna dare in input tantissime immagini contenente l'oggetto in questione. Un altro svantaggio sono i tempi richiesti per addestrare la rete, richiede un hardware molto potente.

DL fornisce un framework flessibile e di apprendimento per la rappresentazione di informazioni visive, testo, linguistiche, in più esso può apprendere in modo supervisionato e non.

POTERE RAPPRESENTATIVO:

Le reti DL empiricamente funzionano bene ma non si sa il motivo del perché statisticamente si comporta meglio degli altri modelli, questo non è provabile del perché quell'algoritmo riconosce con una certa accuratezza, il funzionamento interno è sconosciuto all'utilizzatore, infatti, ciò che si fa è cambiare i parametri per vedere quale è la configurazione che si comporta meglio che ottiene le migliori prestazioni.

Infatti, andando ad analizzare il potere rappresentativo (potere espressivo) di una rete neurale semplice, le NN con almeno un livello nascosto sono **approssimatori universali**:

Data una qualsiasi funzione continua $h(x)$ e qualche $\epsilon > 0$, esiste una NN con uno strato nascosto (e con una ragionevole scelta di non linearità) descritto con la funzione $f(x)$, tale che $\forall x, |h(x) - f(x)| < \epsilon$. Cioè, le NN possono approssimare qualsiasi funzione continua complessa.

Le NN utilizzano la mappatura non lineare degli input x agli output $f(x)$ per calcolare confini decisionali complessi.

Praticamente si sta dicendo che con una rete ad un solo layer si riesce a classificare in maniera abbastanza precisa un qualsiasi problema di classificazione; pertanto, esiste h ma il problema è come si fa a trovarla, teoricamente esiste ma praticamente è difficile trovarla.

Il fatto che le NN profonde funzionino meglio è un'osservazione empirica. Matematicamente, le NN profonde hanno lo stesso potere rappresentativo di una NN ad un livello.

INTRO ALLE RETI NEURALI:

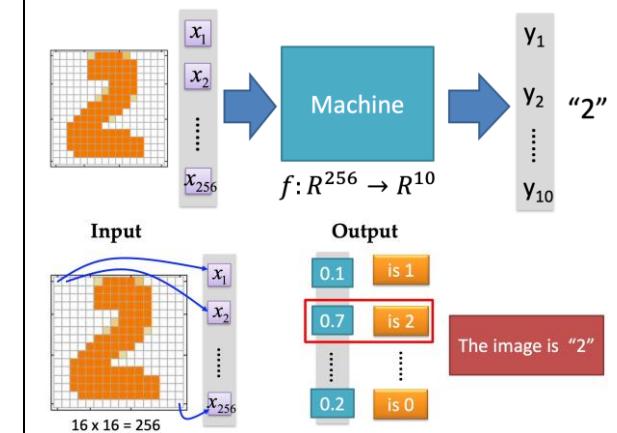
Uno dei casi più facili di classificazione è il riconoscimento di cifre da 0 a 9 da immagini di scrittura.

Supponiamo di avere il numero in una matrice 16x16 pixel, si può rappresentare l'immagine con un vettore di 256 valori, ogni x vale 0 se non è presente inchiostro 1 se è presente.

La rete neurale costruisce un modello che a partire da 256 valori booleani va a produrre un vettore di 10 valori (classificazione multiclasse).

In base al valore di probabilità si prende quello con probabilità maggiore, nell'esempio il numero è 2.

Questo mapping lo fa la funzione f , pertanto f è la nostra rete neurale.

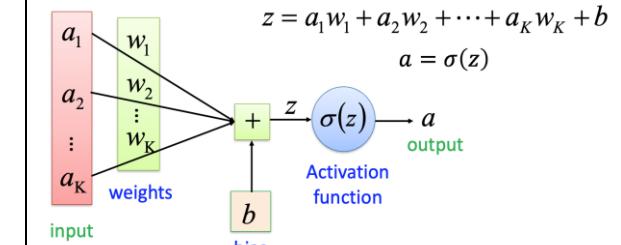


ELEMENTI DI RETE NEURALE:

Le NN sono costituite da strati nascosti con neuroni (cioè unità di calcolo). Un singolo neurone mappa un insieme di input in un numero di output, o $f: R^K \rightarrow R$.

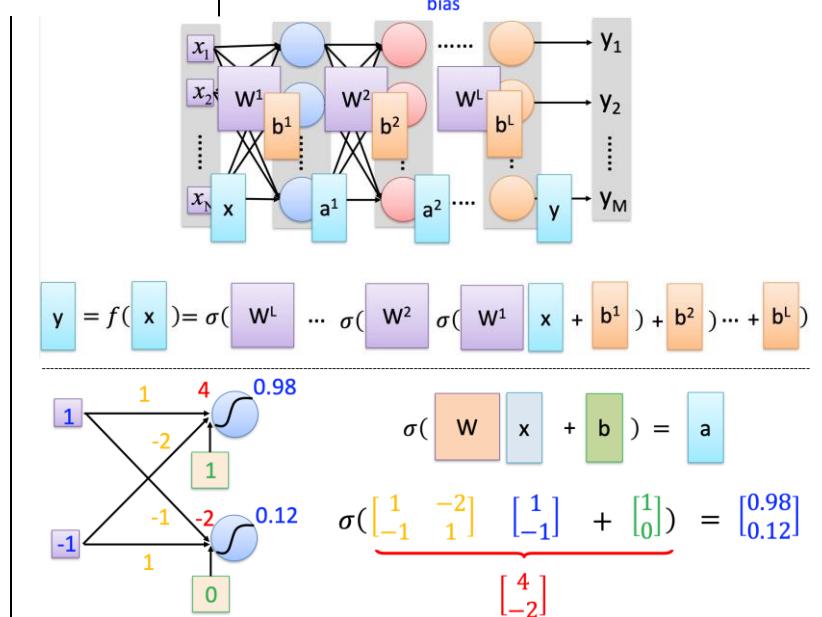
Ogni nodo di una rete semplice effettua i seguenti calcoli:

Fa una somma pesata dell'input con l'aggiunta di un valore (bias), il risultato viene passato ad una funzione di attivazione non lineare che va poi a produrre l'output.



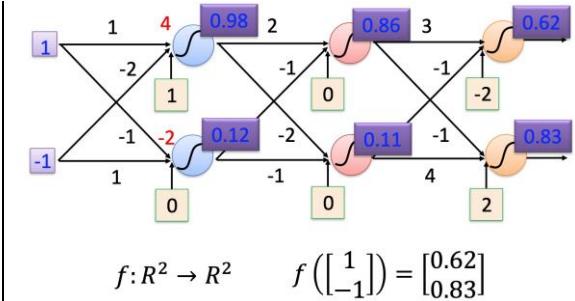
Nelle reti a più livelli si effettuano le stesse operazioni ma questa volta sulle matrici:

possiamo rappresentare il calcolo di h , come prodotto di una matrice per il vettore di input x più il vettore di bias.



Esempio rete più livelli:

Abbiamo 6 neuroni (l'input non si conta), con una rete simile dobbiamo apprendere 26 parametri:



SOFTMAX LAYER:

Nelle attività di **classificazione multiclasse**, il livello di output è in genere uno **strato softmax**, cioè impiega una **funzione di attivazione softmax**.

Avendo tanti nodi di output (classificazione multiclasse) bisogna decidere tra le possibili classi quella più probabile, ovvero quella che ha ottenuto il valore maggiore e questo viene fatto dalla funzione di attivazione softmax.

Questa funzione se abbiamo 3 opzioni (schema al lato) darà più probabilità al primo. Se invece venisse utilizzato uno strato con una funzione di attivazione sigmoidea come strato di output, le previsioni della NN potrebbero non essere facili da interpretare.

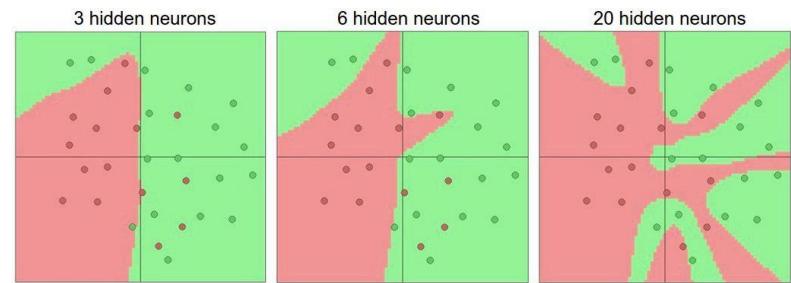
A Layer with Sigmoid Activations

$$\begin{aligned} z_1 &\xrightarrow{\sigma} 0.95 & y_1 &= \sigma(z_1) \\ z_2 &\xrightarrow{\sigma} 0.73 & y_2 &= \sigma(z_2) \\ z_3 &\xrightarrow{\sigma} 0.05 & y_3 &= \sigma(z_3) \end{aligned}$$

FUNZIONI DI ATTIVAZIONE:

Sono necessarie **attivazioni non lineari** per apprendere rappresentazioni di dati complessi (non lineari), altrimenti, NNs sarebbe solo una funzione lineare (come $W_1 W_2 x = Wx$).

NN con un gran numero di strati (e neuroni) possono approssimare funzioni più complesse, più neuroni migliorano la rappresentazione (ma potrebbero andare in overfit).



TRAINING NN:

Supponiamo una rete con 256 input, layer hidden e layer di output, si vogliono apprendere i parametri di questa rete che sono tutti i pesi lungo la rete incluso in bias.

Per fare questo bisogna effettuare una fase di training, ovvero dare in input al modello delle coppie che sono immagine del numero e la classe (label) per andare a determinare theta (θ) che sono tutti i parametri.

I **parametri** di rete θ includono le matrici dei pesi e i vettori di bias per tutti i layer:

$$\theta = \{W^1, b^1, W^2, b^2, \dots, W^L, b^L\}$$

Spesso i parametri del modello θ sono indicati come pesi.

Addestrare un modello per apprendere una serie di parametri θ che sono ottimali (secondo un criterio) è una delle maggiori sfide nel ML.

Quando si trattano immagini si effettua una **fase di preelaborazione** dei dati per migliorare il training, in questo modo il modello dovrebbe apprendere meglio siccome le immagini sono tutte della stessa forma. Le preelaborazioni sono:

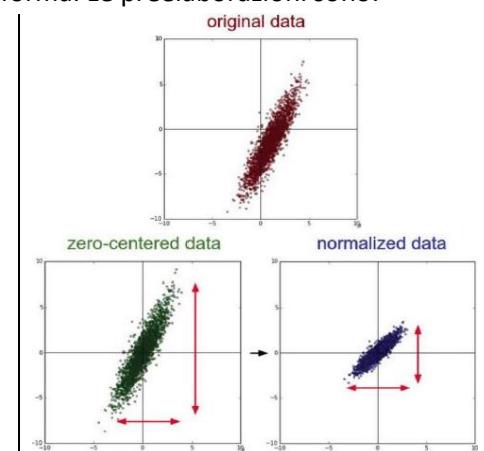
- **Sottrazione della media**, per ottenere dati incentrati sullo zero:

Sottrarre la media per ogni singola dimensione dei dati (feature)

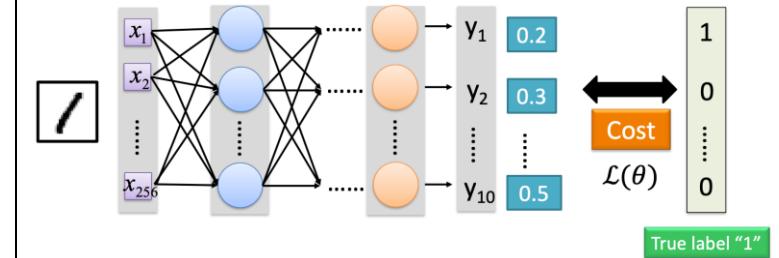
- **Normalizzazione**:

Dividi ogni feature per la sua deviazione standard, per ottenere una deviazione standard di 1 per ciascuna dimensione dei dati (feature).

Oppure, ridimensionare i dati all'interno dell'intervallo $[0,1]$ o $[-1, 1]$, ad esempio, le intensità dei pixel dell'immagine sono divise per 255 per ridimensionare in $[0,1]$.



Quello che si vuole fare durante il training è che, ad esempio, se viene preso in input l'immagine contenente il numero 1 allora y_1 deve avere il valore più alto. Così che la funzione softmax restituisce la giusta classe, questo lo si fa definendo la funzione di loss.



Una **funzione di loss** (funzione di costo/ obiettivo) $\mathcal{L}(\theta)$ calcola la differenza (errore) tra la previsione del modello e l'etichetta vera. Per esempio, può essere errore quadratico medio, cross-entropy, eccetera.

Per un training set di N immagini, calcola la loss totale su tutte le immagini $\mathcal{L}(\theta) = \sum_{n=1}^N \mathcal{L}_n(\theta)$

L'obiettivo è minimizzare la funzione di loss che può essere fatto con un algoritmo iterativo basato sulla discesa del gradiente e ciò che fa è andare a vedere, partendo dalla loss, come modificare i pesi affinché la loss stessa si riduca. Essendo che la loss dipende dai parametri (i pesi), siccome si è fatta prima classificazione e poi applicata la loss.

Nell'esempio di prima erano 26 parametri θ e per tanto bisogna fare 26 calcoli di gradiente e vedere come aggiornare questi 26 pesi. Il gradiente fa vedere come la curva cresce e si va nella direzione opposta ovvero dove diminuisce la funzione.

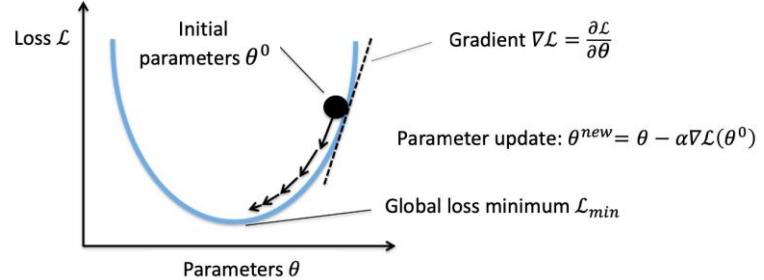
L'algoritmo è iterativo:

1. Inizialmente i pesi sono messi in maniera casuale;
2. Si va a calcolare il gradiente della loss rispetto ai parametri iniziali;
3. I pesi saranno aggiornati secondo la formula allo step 3 in figura;
4. Si ritorna allo step 2 ripetendo l'aggiornamento.

L'algoritmo si ferma quando si raggiunge un **minimo** nella speranza che sia globale e non solo locale.

- Steps in the **gradient descent algorithm**:

1. Randomly initialize the model parameters, θ^0
2. Compute the gradient of the loss function at the initial parameters θ^0 : $\nabla \mathcal{L}(\theta^0)$
3. Update the parameters as: $\theta^{new} = \theta^0 - \alpha \nabla \mathcal{L}(\theta^0)$
 - o Where α is the learning rate
4. Go to step 2 and repeat (until a terminating criterion is reached)



BACKPROPAGATION:

L'**algoritmo di backpropagation** permette di derivare la loss rispetto ai pesi. L'idea è di fare prima una procedura di forward propagation, ovvero dall'input va avanti fino ad arrivare all'output, dopodiché dopo ottenuto l'output si calcola la loss e, per ricavare la derivata della loss rispetto a tutti i pesi, si fa il backpropagation, cioè si fa un passaggio dalla fine verso l'inizio sfruttando la regola della chainrule per le derivate parziali che permettono di fare le derivate all'indietro.

Più formalmente:

- Per addestrare NN, la **forward propagation** (forward pass) si riferisce al passaggio degli input x attraverso i livelli nascosti per ottenere gli output del modello (previsioni) y , è calcolata la perdita $\mathcal{L}(y, \hat{y})$;
- La **back propagation** attraversa la rete in ordine inverso, dagli output y verso gli input x per calcolare i gradienti della perdita $\nabla \mathcal{L}(y, \hat{y})$. La **chain rule** serve per calcolare le derivate parziali della funzione di perdita rispetto ai parametri θ nei diversi strati della rete;
- Ogni aggiornamento dei parametri del modello θ durante l'allenamento effettua un passaggio in avanti e uno indietro (ad es. per un batch di inputs);

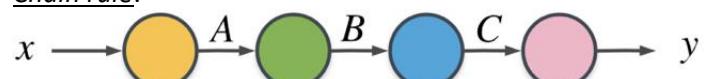
Il calcolo automatico dei gradienti (differenziazione automatica) è disponibile in tutte le librerie di deep learning. Semplifica notevolmente l'implementazione di algoritmi di deep learning, poiché evita di derivare le derivate parziali della funzione di perdita a mano.

Il **problema** è calcolare la derivata della loss che abbiamo alla fine della rete rispetto ai pesi presenti nella rete, cioè si vuole calcolare la derivata di y rispetto a x .

Questa derivata si può calcolare andando a trasformare la frazione (1.) come prodotto della derivata di y rispetto a C , per la derivata di C rispetto a B , per la derivata di B rispetto ad A , per la derivata di A rispetto a x .

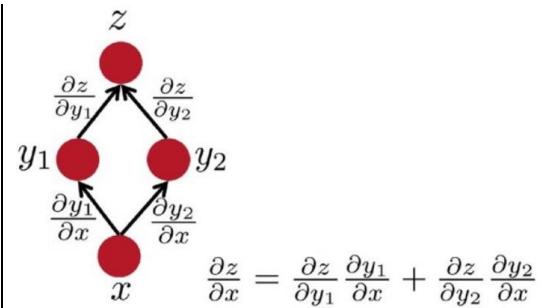
I valori x e y sono distanti tra loro e calcolare la derivata di y rispetto a x è impossibile, ma se lo si scomponete e si calcola in maniera diretta ogni frazione è possibile derivare la frazione iniziale.

Chain rule:



$$(1.) \frac{dy}{dx} = \frac{dy}{dC} \times \frac{dC}{dB} \times \frac{dB}{dA} \times \frac{dA}{dx}$$

Caso in cui abbiamo una diramazione nel grafo, pertanto la formula cambia in una somma di prodotti:



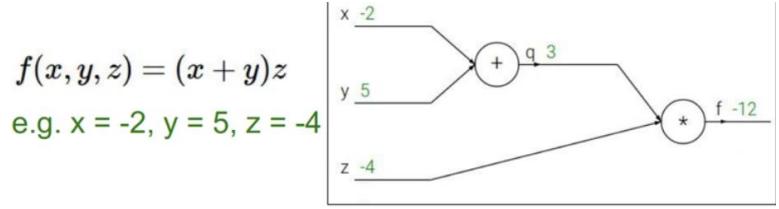
Esempio:

Caso semplice con un grafo computazionale.

x e y sono i due input che vengono prima sommati per produrre $q=x+y$ e poi moltiplicato z per produrre $f=q*z$.

Si vuole capire come aggiornare i pesi per far cambiare i valori di f . Interessa fare la derivata di f rispetto a x , derivata di f rispetto a y e derivata di f rispetto a z .

Dalle formule nei riquadri, si può calcolare solo la derivata rispetto a x e a y (primo riquadro) e la derivata rispetto a q e a z (secondo riquadro), quindi abbiamo il modo di calare solo la derivata di f rispetto a z (terza frazione che si vuole). Le altre due frazioni per calcolarle bisogna ricavare il valore di f rispetto a q che sarebbe z (nell'esempio vale -4).



$$q = x + y \quad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

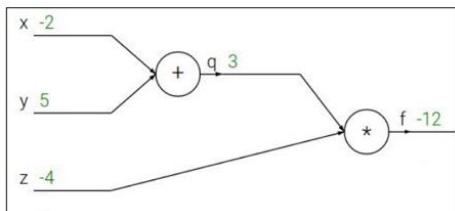
$$f = qz \quad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

$$\text{Want: } \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$$

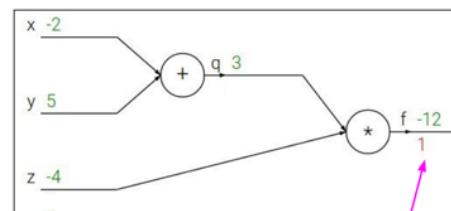
Applicando la chain rule è possibile ricavarsi tutto il necessario:

(il verde è il passo forward, mentre il rosso è la fase di backward)

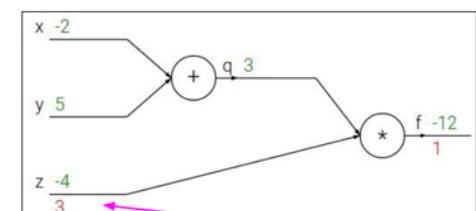
1.



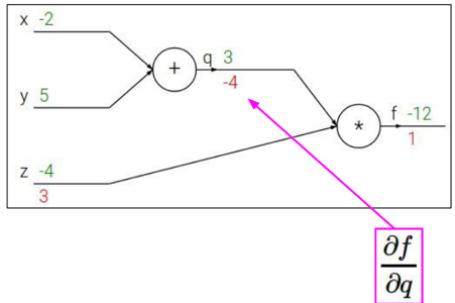
2.



3.

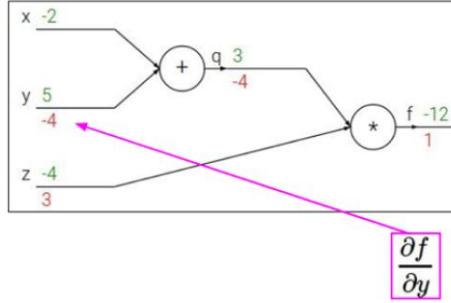


4.



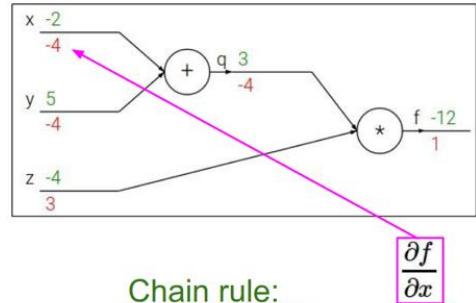
$$\frac{\partial f}{\partial q}$$

5.



$$\frac{\partial f}{\partial y}$$

6.



Chain rule:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

(E) Output (sigmoid)

$$y = \frac{1}{1+exp(-b)}$$

(D) Output (linear)

$$b = \sum_{j=0}^D \beta_j z_j$$

(C) Hidden (sigmoid)

$$z_j = \frac{1}{1+exp(-a_j)}, \forall j$$

(B) Hidden (linear)

$$a_j = \sum_{i=0}^M \alpha_{ji} x_i, \forall j$$

(A) Input

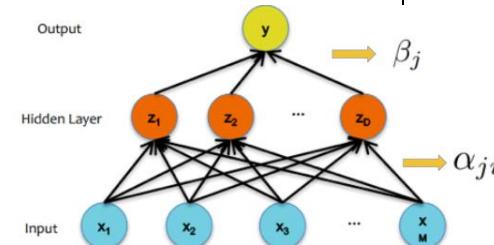
$$\text{Given } x_i, \forall i$$

Riassumendo la rete fa tutto ciò rappresentato dai grafici a destra →

Hidden layer calcola la sommatoria, il risultato va dato alla funzione di attivazione sigmoide, viene calcolato l'output ed infine viene applicato su di esso la sigmoide.

Abbiamo i pesi α sui primi archi e i pesi β sugli altri archi.

La funzione di loss si suppone che sia J (presente a destra), bisogna calcolare la derivata della loss rispetto ai pesi α e β , così da poterli aggiornare.



Assume loss is as follows:

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

Si fa il forward pass che sarebbe calcolare a_j, z_j, b, y per poi andare a calcolare la loss J .

Nel passo backward si parte dal calcolo della derivata della loss rispetto all'output, dopodiché è possibile calcolare la derivata della loss rispetto a b .

Ora è possibile calcolare la derivata della loss rispetto a β che è come il peso β deve essere aggiornato.

Poi si calcola la derivata della loss rispetto a z , e così via fino ad arrivare a calcolare le derivate della loss rispetto ai pesi che è il nostro obiettivo. Facendo poi il prodotto si calcola la derivata finale.

Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^D \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^M \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy} \frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db} \frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$$

$$\frac{dJ}{dz_j} = \frac{dJ}{db} \frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$$

$$\frac{dJ}{da_j} = \frac{dJ}{dz_j} \frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$$

$$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j} \frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$$

$$\frac{dJ}{dx_i} = \frac{dJ}{da_j} \frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^D \alpha_{ji}$$

DISCESA DEL GRADIENTE IN MINI BATCH:

Per quanto riguarda l'esecuzione pratica dell'algoritmo descritto, esistono vari modi. Una prima versione sarebbe andare a calcolare la perdita sull'insieme di dati di training (versione iniziale - vanilla), ma richiede di effettuare tutto il training di tutto il dataset per fare un aggiornamento dei pesi, esso è molto inefficiente.

La versione più utilizzata è la versione con **mini-batch**, dove si prende il dataset e si sceglie un sottoinsieme per sottoporlo al training, si calcola la loss e si aggiornano i pesi, si prende un altro sottoinsieme e si ripete il passaggio. Quando i sottoinsiemi sono finiti si finisce una epoca e si riparte con una epoca successiva.

Più formalmente:

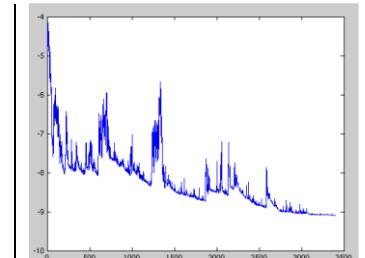
- Calcola la perdita $\mathcal{L}(\theta)$ su un mini-batch di immagini, aggiorna i parametri θ e ripeti finché non sono usate tutte le immagini;
- All'epoca successiva, mescola i dati di training e ripeti il processo precedente.

Il mini-batch porta ad un addestramento molto più veloce. Tipiche taglie di mini-batch: da 32 a 256 immagini. Funziona perché il gradiente di un mini-batch è una buona approssimazione del gradiente dell'intero training set.

DISCESA DEL GRADIENTE STOCASTICO:

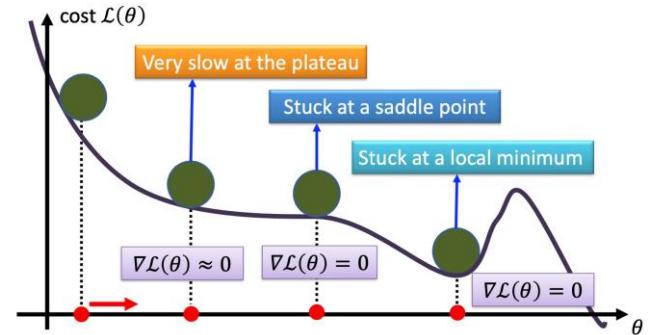
SGD utilizza mini-batch costituiti da un **unico esempio di input**. Ad esempio, una sola immagine di mini-batch. Pertanto, si fa il test, si fa il training di un solo campione, si calcola la loss e si aggiornano i pesi.

Sebbene questo metodo sia molto veloce, può causare fluttuazioni significative nella funzione. Pertanto, è meno comunemente usato, viene preferito GD minibatch.



PROBLEMI CON LA DISCESA DEL GRADIENTE:

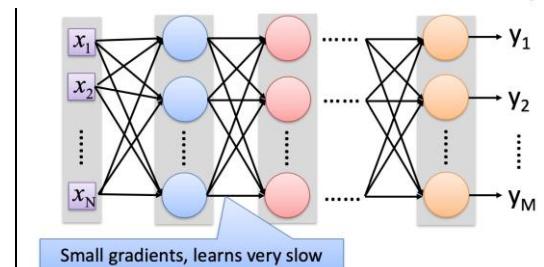
Oltre al problema dei minimi locali, l'algoritmo GD può essere molto lento ai **plateaus**, e può rimanere bloccato in **punti saddle**.



In alcuni casi, durante l'allenamento, i gradienti possono diventare o molto piccoli (**gradienti evanescenti**) o molto grandi (**esplosione del gradiente**).

Portano ad un aggiornamento molto piccolo (i pesi non vengono più aggiornati) o molto grande (variazione dei pesi eccessiva) dei parametri.

La soluzione è il cambio del tasso di apprendimento, attivazioni ReLU, regolarizzazione oppure le unità LSTM in RNN.



REGOLARIZZAZIONE – WEIGHT DECAY:

Chiamata anche **L_2 weight decay** perché fa la sommatoria dei quadrati dei pesi.

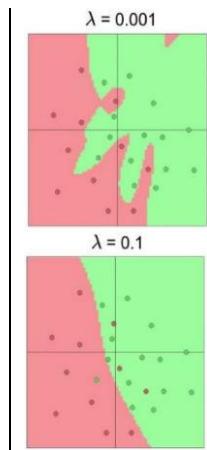
La tecnica del **decadimento del peso** viene utilizzata quando si verifica l'esplosione del gradiente, per **regolarizzare** il valore della loss. Alla loss function si aggiunge un termine di regolarizzazione che penalizza i pesi grandi:

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k \theta_k^2$$

Data loss Regularization loss

Per ogni peso della rete, alla loss aggiungiamo il termine di regolarizzazione. Durante l'aggiornamento dei parametri in GD, ogni peso viene decaduto linearmente verso zero.

Il **coefficiente di decadimento del peso λ** determina quanto sia dominante la regolarizzazione durante il calcolo del gradiente.



Un'altra versione si chiama **L_1 weight decay** perché fa la somma dei valori assoluti dei pesi:

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda \sum_k |\theta_k|$$

Il decadimento del peso **L_1** è meno comune con NN. Spesso ha prestazioni peggiori di **L_2** .

È anche possibile combinare **L_1** e **L_2** chiamata **elastic net regularization**:

$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda_1 \sum_k |\theta_k| + \lambda_2 \sum_k \theta_k^2$$

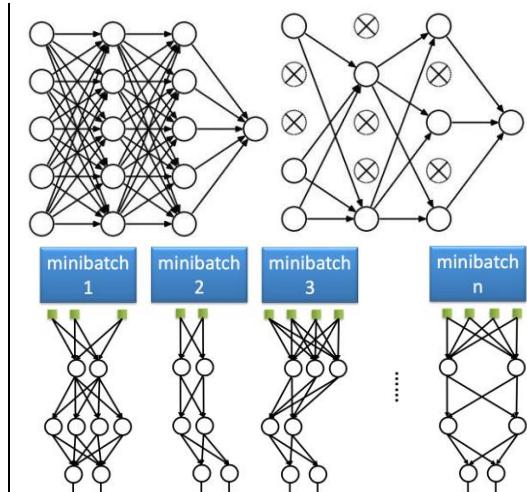
$$\mathcal{L}_{reg}(\theta) = \mathcal{L}(\theta) + \lambda_1 \sigma_k |\theta_k| + \lambda_2 \sigma_k \theta_k^2$$

REGOLARIZZAZIONE – DROPOUT:

Questa tecnica rilascia casualmente le unità (insieme alle loro connessioni) durante il training.

Ciascuna unità viene spenta con un **tasso di dropout p** , indipendente dalle altre unità.

È necessario scegliere l'iperparametro p (tuned), spesso, tra il 20% e il 50% delle unità sono dropped.



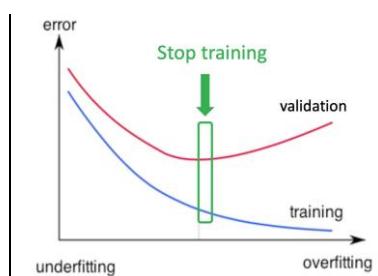
Il dropout può essere visto come una specie di **ensemble learning**, usando un mini-batch diverso per addestrare la rete. Ad esempio, la prima volta si fa training col minibatch 1 e così via... Quello che rimane sono i pesi, siccome i pesi appresi con un minibatch vengono aggiornati nel successivo. In questo modo riescono a catturare delle caratteristiche in maniera diversa in base a quanti nodi sono attivi.

REGOLARIZZAZIONE – FERMATA ANTICIPATA:

Un'altra tecnica è l'**early-stopping**:

- Durante l'addestramento del modello, utilizzare un **set di validazione**.
- Stop quando l'accuracy (o la perdita) della convalida non è migliorata dopo n epoche, il parametro n viene chiamato **patience**.

Tutto ciò evita l'overfitting.



NORMALIZZAZIONE BATCH:

I **livelli di normalizzazione batch** sono dei nodi che si vanno ad aggiungere alla rete (chiamati **layers BatchNorm**) per non far variare i dati rispetto a media e varianza, esso è un processo di normalizzazione.

Calcolano la media μ e la varianza σ di un batch di dati di input e normalizzano i dati x su una media zero e una varianza unitaria →

$$\text{I.e., } \hat{x} = \frac{x - \mu}{\sigma}$$

Portano a convergenza più rapida del training, consentono un tasso di apprendimento più ampio.

HYPER PARAMETRI TUNING:

Il training di NN (reti profonde) può comportare il settaggio di molti **iperparametri**, quelli più comuni sono:

- Il numero di layers ed il numero di neuroni per strato;
- Tasso iniziale del learning rate;
- Tasso di decadimento del learning rate (ad esempio costante);
- Genere di ottimizzatore.

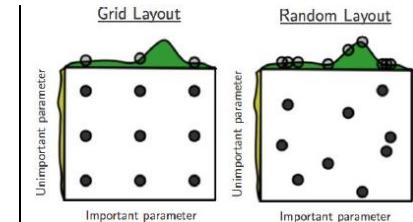
Altri iperparametri possono includere:

- Parametri di regolarizzazione (penalità (#, dropout rate);
- Batch size;
- Funzioni di attivazione;
- Funzione di loss.

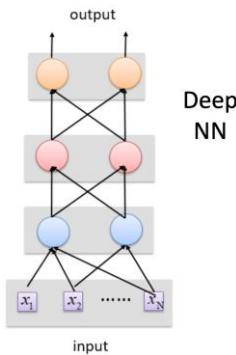
L'ottimizzazione degli iperparametri può richiedere molto tempo per NN grandi. Si parte dall'esperienza appresa della rete, poi bisogna fare degli aggiornamenti e ripetere l'esperimento per vedere come cambia, questo è costoso.

Per fare il Tuning dei iperparametri si possono anche eseguire delle tecniche che sono:

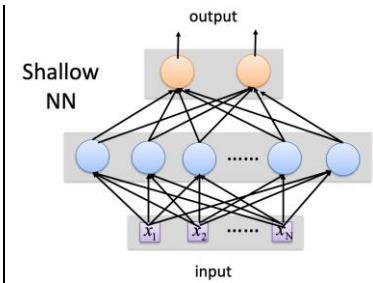
- **Grid search**: Controllare tutti i valori in un intervallo con un valore di step;
- **Random search**: Campiona casualmente i valori per il parametro. Spesso preferito alla ricerca nella griglia;
- **Ottimizzazione bayesiana degli iperparametri**: È un'area attiva di ricerca.



DEEP VS SHALLOW NETWORKS:



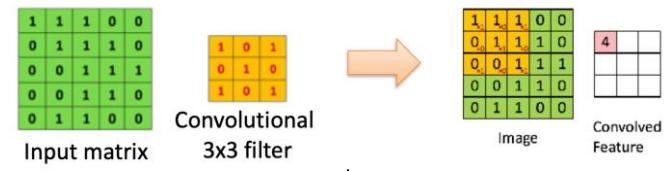
Le **reti più profonde** funzionano meglio di quelle poco profonde reti, ma solo fino a un certo limite, dopo un certo numero di strati, le prestazioni delle reti più profonde si stabilizzano.



RETI NEURALI CONVOLUZIONALI (CNN):

Le **reti neurali convoluzionali** (CNN) sono state progettate principalmente per immagini.

La caratteristica principale delle CNN è che utilizzano, all'interno della rete (nei neuroni), un **operatore convoluzionale** per estrarre le feature.



Un filtro convoluzionale scorre attraverso l'immagine. Quando i filtri convoluzionali vengono scansionati sull'immagine, essi catturano features utili, prima di basso livello come bordi, fino ad estrarre interi oggetti.

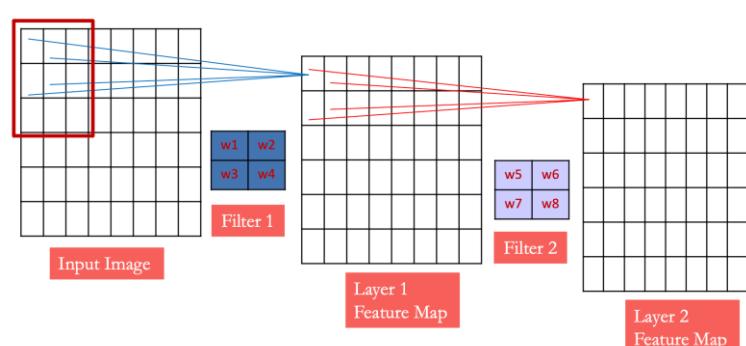


Una cosa importante è che le reti fully connected hanno uno svantaggio riguardante i pesi, essi erano matrici perché partivano da un nodo e andavano in tutti gli altri nodi. Nella realtà, questa dipendenza tra nodi non è sempre necessaria. Con le CNN si riesce ad avere condivisione di pesi tra i nodi e non tutti i nodi sono collegati al livello successivo. Ogni arco ha un peso e nel fully connected abbiamo una quantità di pesi da allenare molto elevata, mentre nelle CNN abbiamo un numero di parametri che si riduce perché gli archi non collegano tutti i nodi.

L'idea principale della rete CNN è l'applicazione del filtro convoluzionale che viene applicato ai dati di input che possono essere visti come delle matrici (rappresentano una immagine) e questo filtro viene fatto passare sopra alla matrice. Questo poi si ripete in base ad un parametro in base dei criteri.

Nelle CNN, le unità nascoste in un livello sono collegate solo a una piccola regione dello strato precedente (chiamato campo **ricettivo locale**).

La profondità di ciascuna **feature map** corrisponde al numero di filtri convoluzionali utilizzati in ciascuna strato.



Dopo aver applicato un operatore convoluzionale si può applicare un **pooling**, che serve per ridurre la quantità di dati, riducendo la dimensione della matrice iniziale:

- **Max pooling**: riporta l'output massimo all'interno di un rettangolo (vicinato);

- **Average pooling**: riporta l'output medio di un rettangolo vicino;

I livelli di pooling riducono la dimensione spaziale delle feature map. Ridurre il numero di parametri, prevenire overfitting.

1	3	5	3
4	2	3	1
3	1	1	3
0	1	0	4

Input Matrix

MaxPool with a 2×2 filter with stride of 2

4	5
3	4

Output Matrix

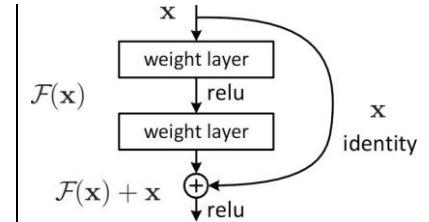
CNN RESIDUALI (ResNet):

Una variante delle CNN sono le **residuali** che fanno avanzare delle informazioni da un layer ad un layer più avanti nella catena. È utile, siccome potrebbe capitare che da un layer al successivo ci sono delle grosse variazioni dovuto a delle perturbazioni.

Tra i vari processi di convoluzione e pooling potremmo perdere informazioni legate all'immagine originale, per prevenire questo portiamo avanti i dati che abbiamo in un certo layer.

Esempio:

Immaginiamo di avere x , potrebbe capitare che i due strati cambiano il significato dei dati. Quindi, propaghiamo x in avanti, mitigando il problema del vanishing gradient durante il training.



RETI NEURALI RICORRENTI (RNN):

Le **Recurrent NN** vengono utilizzate per modellare **dati sequenziali** e dati con input e output di lunghezza variabile, esempio video, testo, parlato, sequenze di DNA, dati di scheletri umani. Le RNN tengono in considerazione non solo x ma anche i dati precedenti, proprio per questo non vengono utilizzati per le immagini, siccome per esse si vuole sapere y solo dato x . La memoria degli input precedenti viene archiviata nello stato del modello ed influenzano le predizioni del modello.

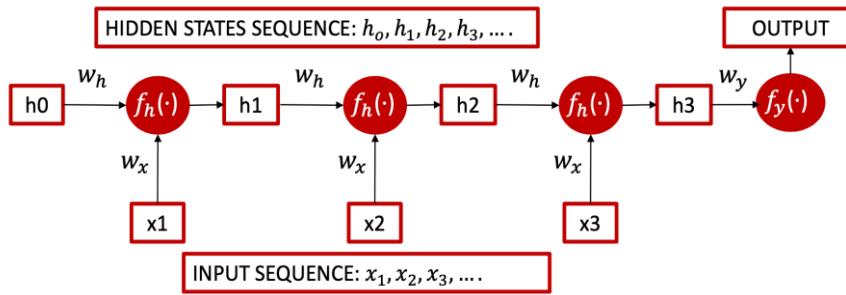
Le RNN sono più sensibili al **problema del gradiente evanescente** rispetto alle CNN, siccome vanno considerati i pesi per i dati precedenti.

Più formalmente, RNN usa lo stesso set di pesi w_h e w_x **in tutti i passi temporali**:

- Viene appresa una sequenza di **hidden state** $\{h_1, h_2, \dots\}$ che rappresenta la memoria della rete;
- Lo stato hidden al passo t , $h(t)$, viene calcolato in base al precedente hidden state $h(t-1)$ e l'input al passo corrente $x(t)$, cioè:

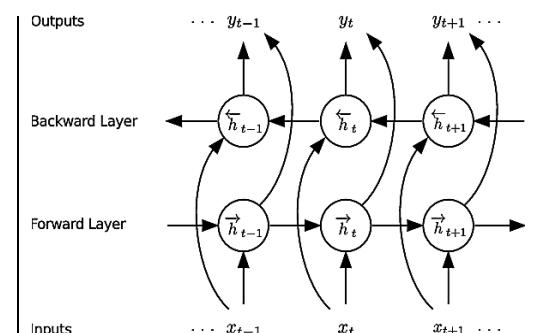
$$h(t) = f_h(w_h * h(t-1) + w_x * x(t))$$

- La funzione $f_h(\cdot)$ è una funzione di attivazione non lineare, e.g., ReLU o tanh



BIDIREZIONALE RNN:

Sono una evoluzione delle precedenti ed è una generalizzazione della precedente, cioè esse guardano non solo indietro ma anche in avanti. Quando analizzano un certo x_i tengono in considerazione sia il layer di forward che di backward.



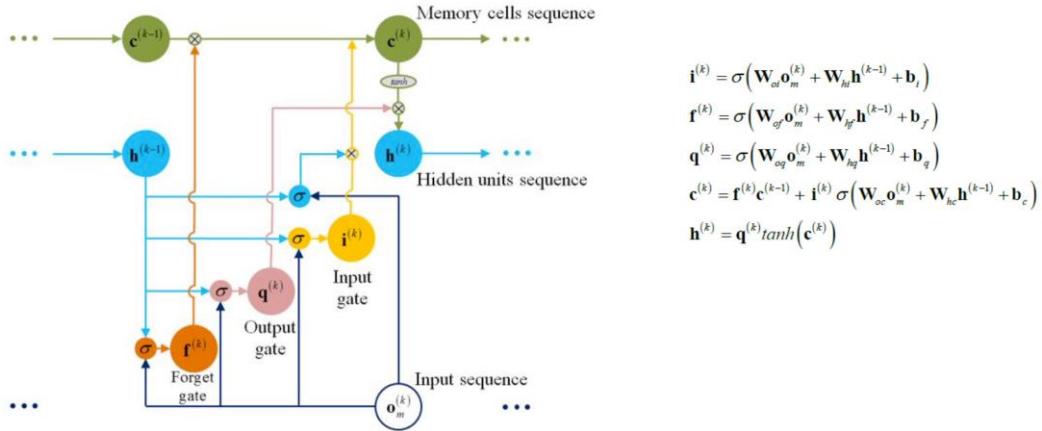
LSTM RETI:

Le **Long Short-Term Memory** (LSTM) sono una variante di RNN, LSTM attenua il problema della fuga del gradiente/esplosione, la soluzione è una *Memory Cell*, aggiornata ad ogni passaggio della sequenza.

Tre porte controllano il flusso di informazioni da e verso la *Memory Cell*:

- *Input Gate*: protegge il passo corrente dagli input irrilevanti;
- *Output Gate*: impedisce al passo corrente di trasmettere informazioni irrilevanti ai passi successivi;
- *Forget Gate*: limita le informazioni passate da una cella alla prossima.

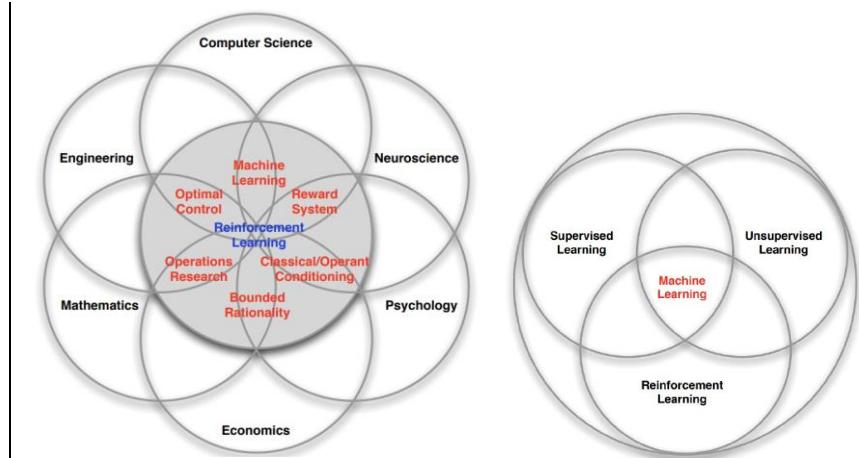
Il problema dei RNN è che si portano avanti delle informazioni ma alcune di esse vengono perse comunque nei vari layer successivi (perdita del contesto), invece con i gate possiamo decidere quanto far passare di contesto al layer successivo (maggior controllo sui dati tramite le porte).



16. REINFORCEMENT LEARNING (RL)

Dal punto di vista teorico l'**apprendimento per rinforzo**, ricopre diverse aree scientifiche e si occupa di diverse problematiche come problemi di ottimizzazione, machine learning, ecc...

Per quanto riguarda la parte di **machine learning**, il **reinforcement learning** è un'altra tecnica usata oltre all'apprendimento supervisionato e non.



PROBLEMI CON APPRENDIMENTO SUPERVISIONATO:

Un agente di **apprendimento supervisionato** apprende osservando passivamente l'esempio coppie ingresso/uscita fornite da un "insegnante".

In altre parole, consiste nel dare degli esempi (training set) con la risposta al problema, così da addestrare un algoritmo a capire come rispondere a determinate domande. Abbiamo una funzione f che vogliamo modellare, tramite il training set cerchiamo di insegnare alla macchina a capire come rispondere a delle domande facendo sì che la funzione scelta h si comporti come f .

Ci sono scenari dove l'apprendimento supervisionato non va bene, siccome non abbiamo a disposizione il dataset per insegnare la macchina cosa fare.

Un esempio sono i problemi della vita reale o semplicemente il gioco degli scacchi. Per costruire un modello per giocare a scacchi si necessita di un training set, se prendiamo tutte le mosse delle partite dei campioni di scacchi, si costruisce il dataset di mosse. Il problema è che la quantità di dati che si riesce ad ottenere è molto inferiore a tutte le possibili configurazioni del gioco, non siamo in grado di costruire questo modello perché non abbiamo abbastanza dati.

Per questi tipi di problemi è opportuno usare tecniche che si utilizzano nella vita quotidiana, cioè cercare di capire il funzionamento andando ad utilizzarlo.

CARATTERISTICHE DEL RL:

L'agente che si andrà a costruire è chiamato **agente per obiettivo** che utilizza l'**apprendimento per rinforzo o interattivo**, cioè la macchina interagisce con l'ambiente e apprende come deve comportarsi tramite un meccanismo che si chiama ricompensa (**reward**), tramite **feedback** la macchina riesce a capire se le azioni che ha fatto sono giuste o sbagliate. Effettuando questa azione molte volte, la macchina riesce a capire le azioni corrette che portano al raggiungimento dell'obiettivo. Il feedback non è istantaneo, cioè la ricompensa di quell'azione potrebbe arrivare dopo molto tempo, in questo contesto il tempo va tenuto in considerazione.

Esempi di RL sono ad esempio un braccio robotico che gli si insegna a svolgere degli esercizi dove ha un oggetto che deve portare ad una certa posizione e lui dà solo inizierà a provare delle azioni e quando arriva all'obiettivo riceve la ricompensa. Man mano che lo ripete più volte riesce ad arrivare all'obiettivo nel minor numero di azioni possibili.

UTILIZZO RL:

Il problema che affronta il RL è:

- Trovare la soluzione al problema (scacchi, robot industriale);
- Necessità di costruire sistemi che nel tempo si adattano all'ambiente, cioè sistemi che col tempo continuano ad apprendere, ad esempio, insegnare ad un robot a scendere le scale se successivamente deve salire deve apprendere anche questo.

Col reinforcement learning l'obiettivo è costruire algoritmi capaci di apprendere tramite interazioni col sistema. Questo richiede di tenere in considerazione:

- Il tempo, quando si fa una certa azione cambia il flusso degli eventi e il feedback possiamo riceverlo dopo un po';
- le conseguenze delle azioni (a lungo termine);
- acquisire esperienza in modo attivo;
- predire il futuro;
- gestire le incertezze.

16.1 FORMALIZZARE IL RENFORCEMENT LEARNING

RICOMPENSE (REWARD):

La ricompensa è quella informazione che l'agente riceve come feedback dell'azione effettuata.

Una **ricompensa R_t** è un feedback rappresentato con uno scalare e indica il grado di efficienza dell'agente allo step t (tempo). Il compito dell'agente è quello di massimizzare la ricompensa cumulativa, cioè tutti i feedback ricevuti li va a sommare e l'obiettivo è raggiungere la ricompensa maggiore.

Il Reinforcement Learning è basato sulla **reward hypothesis**:

Tutti gli obiettivi possono essere descritti come la massimizzazione della ricompensa cumulativa prevista.

Esempio ricompense:

Far camminare un robot umanoide:

- ricompensa **positiva** per aver eseguito la traiettoria **desiderata**
- ricompensa **negativa** in caso di **caduta**

Sconfiggere il campione del mondo di Backgammon

- ricompensa **positiva/negativa** per aver **vinto/perso** una partita

PROCESSO DECISIONALE SEQUENZIALE:

L'obiettivo è selezionare le azioni al fine di massimizzare la ricompensa totale futura, ma va considerato che:

- Le azioni possono avere conseguenze (positive o negative) a lungo termine;
- La ricompensa potrebbe essere ritardata (esempio investimento);
- Può essere preferibile sacrificare una ricompensa immediata per ottenere una maggiore ricompensa a lungo termine.

INTERAZIONE TRA AGENTE E AMBIENTE:

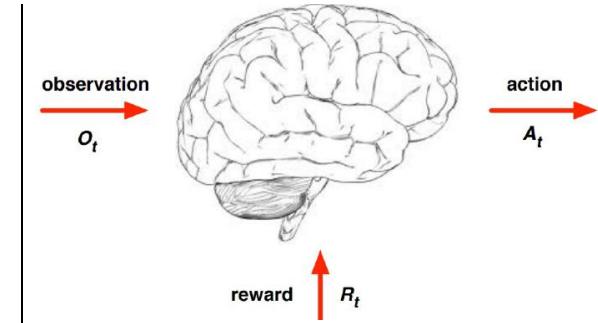
A ciascun step t l'agente:

- Esegue l'azione A_t
- Riceve l'osservazione O_t
- Riceve la ricompensa R_t

L'ambiente:

- Riceve l'azione A_t
- Produce l'osservazione O_{t+1}
- Produce la ricompensa R_{t+1}

t viene incrementato.



STORIA E STATO:

Dal processo iterativo precedente possiamo definire una 'storia' che va a caratterizzare tutto quello che è successo fino ad un certo istante.

La **storia** è la sequenza di osservazioni, azioni, ricompense:

$$H_t = O_1, R_1, A_1, \dots, A_{t-1}, O_t, R_t$$

cioè tutte le variabili osservabili fino al tempo t .

La storia influenza ciò che accade successivamente:

- Le azioni eseguite dall'agente;
- Le osservazioni/ricompense prodotte dall'ambiente.

Lo **stato** è l'informazione utilizzata per determinare ciò che accade successivamente. Formalmente, lo stato S_t è una funzione della storia:

$$S_t = f(H_t)$$

Bisogna però distinguere due possibili stati siccome essi potrebbero non coincidere.

STATO DELL'AGENTE:

Lo **stato dell'agente** S_t^a è la rappresentazione interna dell'agente a . È quello che l'agente conosce ed è quello che utilizza per scegliere l'azione successiva.

Questa è l'informazione utilizzata dagli algoritmi di RL.

Lo stato dell'agente può essere qualsiasi funzione della storia:

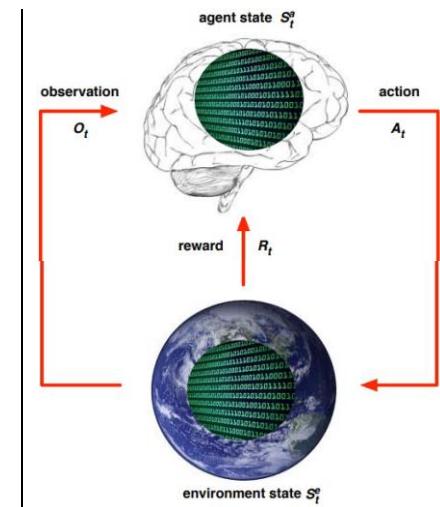
$$S_t^a = f(H_t)$$

STATO DELL'AMBIENTE:

Lo **stato dell'ambiente** S_t^e è la rappresentazione privata dell'ambiente e al tempo t .

Va a descrivere lo stato in cui si trova l'ambiente ed è l'informazione su cui si basa O_t ed R_t , cioè quando produce l'osservazione e la ricompensa successiva lo farà in funzione di questo stato.

Di solito, lo stato dell'ambiente non è visibile all'agente. Anche se S_t^e è visibile, potrebbe contenere informazioni irrilevanti.



STATO DELLE INFORMAZIONI:

Lo stato associato all'agente deve avere la proprietà che deve essere uno **stato di Markov**, cioè all'interno dello stato si va a memorizzare solo informazioni utili della storia. Praticamente, lo stato successivo dipende solo dallo stato precedente e non da tutti gli stati dall'inizio fino a quell'istante (ovvero l'intera storia).

Più formalmente, uno stato S_t è detto di **Markov** se e solo se:

$$P[S_{t+1} | S_1, \dots, S_t] = P[S_{t+1} | S_t]$$

«*Dato il presente, il futuro è indipendente dal passato*»

$$H_{1:t} \rightarrow S_t \rightarrow H_{t+1:\infty}$$

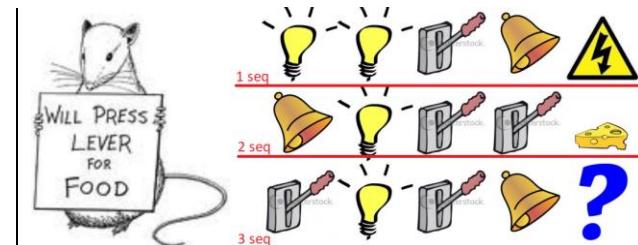
Una volta che lo stato è noto, la storia può non essere considerata.

Lo stato è una statistica sufficiente del futuro. Lo stato dell'ambiente S_t^e e la storia H_t sono di Markov.

Esempio:

Considerando le 3 sequenze a lato.

- Cosa succede se lo stato dell'agente = ultimi 3 elementi della sequenza? Sequenza 1;
- E se lo stato dell'agente = numero di luci, campanelli e leve? Sequenza 2;
- E se lo stato dell'agente = sequenza completa? Sequenza 3;



AMBIENTI COMPLETAMENTE OSSERVABILI:

Osservabilità completa, abbiamo tutta la conoscenza di che cosa accade nell'ambiente, l'agente osserva direttamente lo stato dell'ambiente: $O_t = S_t^a = S_t^e$ (Stato delle informazioni = Stato dell'agente = Stato dell'ambiente).

Formalmente, questo è definito **Processo Decisionale di Markov** (MDP).

AMBIENTE PARZIALMENTE OSSERVABILE:

Osservabilità parziale, abbiamo accesso solo ad una conoscenza ristretta dell'ambiente, l'agente osserva indirettamente l'ambiente. Un agente di trading osserva solo i prezzi correnti oppure un agente che gioca a poker osserva solo le carte sul tavolo. In questo caso Stato dell'agente \neq Stato dell'ambiente.

Formalmente, questo è definito **Processo Decisionale di Markov Parzialmente Osservabile** (POMDP).

L'agente deve costruire la propria rappresentazione dello stato S_t^a , ad esempio usando:

- Storia completa:

$$S_t^a = H_t$$

- Credenze sullo stato dell'ambiente:

$$S_t^a = (\mathbb{P}[S_t^e = s^1], \dots, \mathbb{P}[S_t^e = s^n])$$

- Rete neurale ricorrente:

$$S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$$

16.2 COMPONENTI DI UN AGENTE BASATO SU REINFORCEMENT LEARNING

Un agente basato su RL include una o più delle seguenti componenti:

- **Policy**: funzione di comportamento dell'agente, ovvero dice all'agente come deve comportarsi;
- **Value function**: esprime quanto è valido ogni stato e/o azione, associando un valore che esprime l'importanza;
- **Modello**: rappresentazione dell'ambiente ad opera dell'agente.

POLICY:

Una **policy** π esprime il comportamento dell'agente. Stabilisce l'azione da eseguire in base allo stato in cui si trova l'agente:

- **Policy deterministica**: $a = \pi(s)$;
- **Policy stocastica**: $\pi(a|s) = P[A_t = a | S_t = s]$.

VALUE FUNCTION:

Una **value function** è una previsione della ricompensa futura.

Viene utilizzata per **valutare la validità degli stati** e quindi per la scelta delle azioni da eseguire, ad esempio:

$$v_{\pi}(s) = \mathbb{E}_{\pi} [R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | S_t = s]$$

Ricompensa futura **attesa** (scontata) in base alla politica π dallo stato s .

Per scegliere uno stato rispetto ad un altro bisogna considerare quanto si riceve come ricompensa andando in quello stato considerando anche le ricompense future. Pertanto, uno stato s potrebbe avere sempre una ricompensa bassa ma avere un valore $V_{\pi}(s)$ alto, questo perché gli stati seguenti hanno una ricompensa alta.

MODELLO:

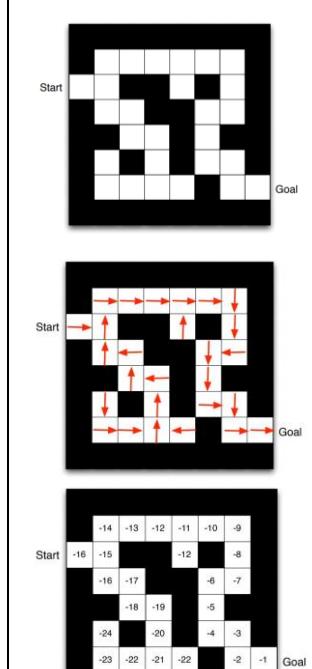
Un **modello** predice ciò che l'ambiente farà in seguito:

- P predice il prossimo stato;
- R predice la ricompensa successiva (immediata), ad esempio:

$$\begin{aligned}\mathcal{P}_{ss'}^a &= \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] \\ \mathcal{R}_s^a &= \mathbb{E}[R_{t+1} | S_t = s, A_t = a]\end{aligned}$$

Esempio labirinto:

- Ricompense: -1 per time-step;
- Azioni: N, E, S, O;
- Stati: posizione dell'agente.



Policy:

Le frecce rappresentano la policy $\pi(s)$ per ogni stato s .

Dato uno stato mi dice l'azione possibile.

Value function:

I numeri rappresentano il valore $v_{\pi}(s)$ di ciascuno stato s .

Tempo previsto per raggiungere l'obiettivo.

Modello:

L'agente può avere un modello interno (imperfetto) dell'ambiente:

- Come le azioni cambiano lo stato;
- Quanta ricompensa si ottiene da ogni stato.

Il layout a griglia rappresenta il modello di transizione $\mathcal{P}_{ss'}^a$.

I numeri rappresentano la ricompensa immediata \mathcal{R}_s^a da ciascuno stato s .

APPRENDERE LE COMPONENTI DEGLI AGENTI:

Tutte le componenti sono funzioni:

- Policies: $\pi : \mathcal{S} \rightarrow \mathcal{A}$ (or to probabilities over \mathcal{A})
- Value functions: $v : \mathcal{S} \rightarrow \mathbb{R}$
- Models: $m : \mathcal{S} \rightarrow \mathcal{S}$ and/or $r : \mathcal{S} \rightarrow \mathbb{R}$
- State update: $u : \mathcal{S} \times \mathcal{O} \rightarrow \mathcal{S}$

Ad esempio, possiamo usare reti neurali o tecniche di deep learning per apprenderle. Spesso non valgono le assunzioni del supervised learning (stazionarietà, iid), cioè il tempo non era importante mentre in questo caso sì.

LEARNING E PLANNING:

Ci sono due problematiche fondamentali in un processo decisionale sequenziale:

Reinforcement Learning, in questo caso non possiamo pianificare e dobbiamo per forza interagire:

- L'ambiente è inizialmente sconosciuto;
- L'agente interagisce con l'ambiente;
- L'agente migliora la sua policy.

Planning, dove l'algoritmo fa delle analisi per decidere da solo che azione fare:

- Un modello dell'ambiente è noto;
- L'agente esegue le computazioni con il suo modello (senza alcuna interazione esterna);
- L'agente migliora la sua policy.

Esempio Atari (RL):

Le regole del gioco sono sconosciute. L'apprendimento è basato sull'esperienza ottenuta dall'interazione con il gioco.

Bisogna scegliere le azioni da eseguire col joystick, vedere i pixel e i punteggi.

Esempio Atari (Planning):

Le regole del gioco sono note. L'agente può interrogare l'emulatore, esiste un modello perfetto per l'agente. Se l'agente esegue un'azione a da uno stato s :

- Quale è il prossimo stato?
- Quale è il punteggio?

Bisogna pianificare in anticipo per trovare una policy ottimale. Ad esempio, si può usare un albero di ricerca.

EXPLORATION & EXPLOITATION (ESPLORAZIONE E SFRUTTAMENTO):

Il **Reinforcement Learning** è simile ad un apprendimento *trial-and-error*.

L'agente dovrebbe individuare una politica buona tramite l'esperienza che acquisisce interagendo con l'ambiente e senza perdere troppa ricompensa lungo il percorso.

Quindi si ripete più volte il task, prova delle azioni e vede che succede, col tempo impara che alcune azioni lo portano a punteggi vari, però c'è comunque uno spazio del problema che lui non conosce:

- L'**exploration** trova più informazioni sull'ambiente;
- L'**exploitation** sfrutta le informazioni note (o acquisite in precedenza) per massimizzare la ricompensa.

Di solito questi due task hanno la stessa importanza e bisogna avere un buon compromesso tra i due.

Esempio:

Selezione di un ristorante:

- Exploitation: Vai al tuo ristorante preferito;
- Exploration: Prova un nuovo ristorante.

PREDICTION E CONTROL:

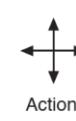
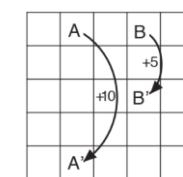
- **Prediction**: valutare il futuro, la policy è nota;
- **Control**: ottimizzare il futuro, trova la miglior policy.

Esempio Grindworld (Prediction):

Fissata una policy la si va a valutare, sfruttando la value function, cioè valuto gli stati della griglia per decidere gli stati più promettenti.

Esempio Grindworld (Control):

Se la andiamo a calcolare troviamo la value function ottimale e la policy ottimale (fase di control).



22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

a) gridworld

3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

b) v_*

→	↔	←	↑	↓
↑	↑	↑	↑	↑
↑	↑	↑	↑	↑
↑	↑	↑	↑	↑
↑	↑	↑	↑	↑

c) π_*

17. PROCESSI DECISIONALI DI MARKOV

I **processi decisionali di Markov** descrivono formalmente un ambiente per il RL, quando l'ambiente è completamente osservabile, cioè lo stato attuale caratterizza completamente il processo.

Quasi tutti i problemi di RL possono essere formalizzati come MDP, anche quando i problemi sono parzialmente osservabili possono essere convertiti in MDP.

PROPRIETÀ DI MARKOV:

“Dato il presente, il futuro è indipendente dal passato”

Uno stato S_t è detto di **Markov** se e solo se:

$$P[S_{t+1} | S_t] = P[S_{t+1} | S_1, \dots, S_t]$$

(La probabilità di andare in uno stato S_{t+1} dipende solo dallo stato attuale, tutti gli stati precedenti non sono influenti)

Lo stato raccoglie tutte le informazioni rilevanti della storia. Una volta che lo stato corrente è noto, la storia può non essere considerata. Lo stato è una **statistica sufficiente** del futuro.

MATRICE DI TRANSIZIONE DI STATO:

Per uno stato di Markov s e uno stato successore s' , la **probabilità di transizione di stato** è definita da:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' \mid S_t = s]$$

(probabilità di andare in s' , partendo da s)

La **matrice di transizione di stato P** definisce le probabilità di transizione da tutti gli stati s a tutti gli stati successori s' :

$$\mathcal{P} = \text{from} \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & & \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \text{to}$$

La matrice di transizione ha sulle righe gli stati di partenza e sulle colonne gli stati di arrivo e si va ad indicare la probabilità di passare da uno stato all'altro. La somma dei valori in ogni riga è pari a 1.

PROCESSO DI MARKOV:

Un processo di Markov è un processo *casuale senza memoria*, cioè una sequenza di stati casuali S_1, S_2, \dots, S_n caratterizzati dalla proprietà di Markov, ovvero la scelta dello stato successivo dipende solo dal precedente.

Un **processo di Markov** (o **catena di Markov**) è una tupla $\langle S, P \rangle$, dove:

- S è un insieme (finito) di stati;
 - P è una matrice di probabilità di transizione di stato: $P_{ss'} = P[S_{t+1} = s' \mid S_t = s]$.

Esempio catena di Markov degli studenti:

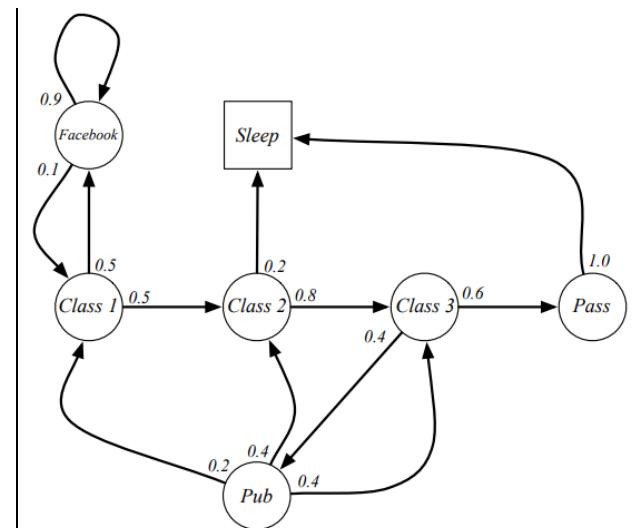
Sugli archi uscenti si hanno le probabilità di transizione di stato e la loro somma è 1. Con questo processo di Markov è possibile generare delle sequenze chiamati episodi, lo sono perché lo stato *Sleep* è uno stato finale (quadrato).

Esempi di episodi ottenuti partendo da $S_1=Class1$ (C_1)

S_1, S_2, \dots, S_t

Dallo stato C_1 è possibile ottenere i seguenti episodi:

- $C_1 C_2 C_3$ Pass Sleep
 - C_1 FB FB $C_1 C_2$ Sleep
 - $C_1 C_2 C_3$ Pub $C_2 C_3$ Pass Sleep
 - C_1 FB FB $C_1 C_2 C_3$ Pub C_1 FB FB FB $C_1 C_2 C_3$ Pub C_2 Sleep



Il diagramma permette di ottenere la matrice di probabilità:

$$P_{ss'} = P(S_{t+1} = s' | S_t = s)$$

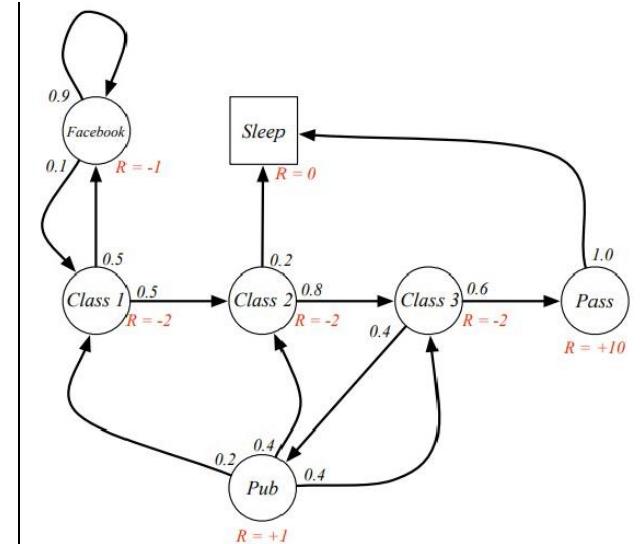
PROCESSO DI RICOMPENSA DI MARKOV:

Un **processo di ricompensa di Markov** (MRP) è una catena di Markov con valori ed è una quadrupla $\langle S, P, R, \gamma \rangle$:

- S è un insieme (finito) di stati;
- P è una matrice di probabilità di transizione di stato: $P_{ss'} = P[S_{t+1} = s' | S_t = s]$;
- R è una funzione di ricompensa (la ricompensa attesa dello stato s): $R_s = E[R_{t+1} | S_t = s]$;
- γ (gamma) è un fattore di sconto, $\gamma \in [0,1]$.

Esempio MRP degli studenti:

Si aggiunge R associato ad ogni stato. Esempio, passare l'esame ha ricompensa 10.



GUADAGNO:

Una volta definita la ricompensa è possibile ricavare il guadagno.

Il **guadagno** G_t è la ricompensa totale calcolata a partire dal time-step t :

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Lo **sconto** $\gamma \in [0,1]$ è il valore attuale delle ricompense future. Il valore della ricompensa ricevuta R dopo $k+1$ time-step è $\gamma^k R$. In questo modo la ricompensa immediata viene privilegiata rispetto a quella a lungo termine:

- γ vicino a 0 porta a una valutazione "miope", non si considerano le ricompense future ma solo quella immediata;
- γ vicino a 1 porta a una valutazione "lungimirante", si considerano le ricompense che si otterranno in seguito.

Il motivo per cui si applica lo sconto è perché:

- Matematicamente conveniente scontare i premi, quando le sequenze sono molto lunghe avere valori bassi permette di tenere sotto controllo i valori delle ricompense;
- Evita guadagni infiniti nei processi di Markov ciclici;
- In alcuni problemi è meglio dare peso a ricompense immediate (quindi si ha $\gamma=0$);
- A volte è possibile utilizzare processi di ricompensa di Markov non scontati (cioè $\gamma = 1$), ad esempio se tutte le sequenze terminano.

VALUE FUNCTION:

La **value function** misura il valore a lungo termine di essere in uno stato s , in pratica dice da quello stato che ricompensa ci si aspetta, ricompensa non attuale ma totale.

La **state-value function $v(s)$** di un MRP è il guadagno previsto partendo dallo stato s :

$$v(s) = E[G_t | S_t = s]$$

(la value function è il guadagno atteso G se mi trovo nello stato s a partire dall'istante t)

Esempio guadagni nel MRP degli studenti:

Esempio dei guadagni ottenuti partendo da $S_1 = C_1$ e con $\gamma = \frac{1}{2}$, possiamo calcolarci la value function dello stato S_1 :

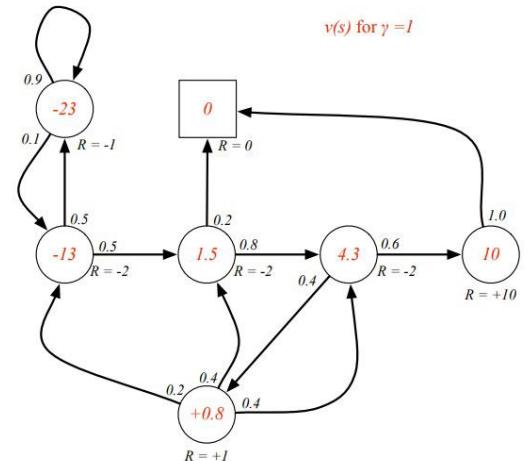
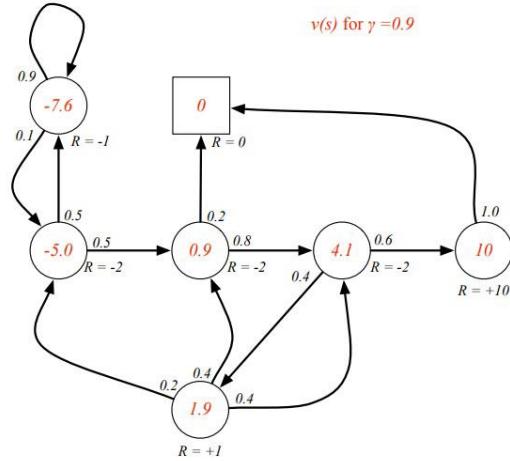
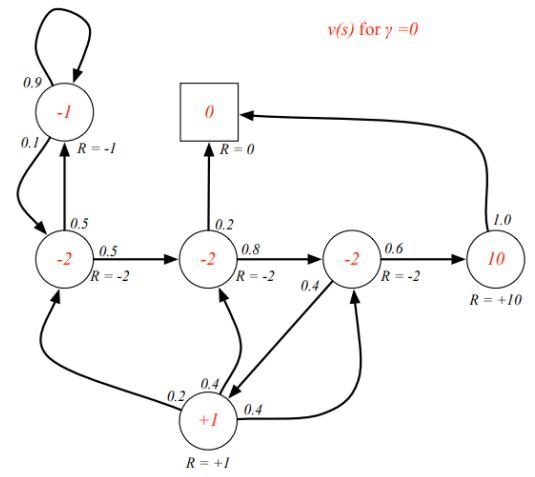
$$G_1 = R_2 + \gamma R_3 + \dots + \gamma^{T-2} R_T$$

C1 C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 10 * \frac{1}{8}$	=	-2.25
C1 FB FB C1 C2 Sleep	$v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16}$	=	-3.125
C1 C2 C3 Pub C2 C3 Pass Sleep	$v_1 = -2 - 2 * \frac{1}{2} - 2 * \frac{1}{4} + 1 * \frac{1}{8} - 2 * \frac{1}{16} \dots$	=	-3.41
C1 FB FB C1 C2 C3 Pub C1 ...	$v_1 = -2 - 1 * \frac{1}{2} - 1 * \frac{1}{4} - 2 * \frac{1}{8} - 2 * \frac{1}{16} \dots$	=	-3.20
FB FB FB C1 C2 C3 Pub C2 Sleep			

Bisogna fare la somma delle ricompense pesate con γ .

Esempio state-value function per il MRP degli studenti:

- Con $\gamma=0$ è la ricompensa dello stato;
- Con $\gamma=0.9$ iniziano a cambiare i valori;
- Con $\gamma=1$ è la sommatoria di tutte le ricompense.



BELLMAN EQUATION PER MRP:

La **value function** può essere suddivisa in due parti:

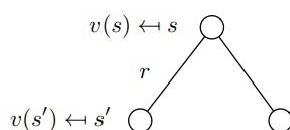
- ricompensa immediata R_{t+1} ;
- valore scontato dello stato successore $\gamma v(S_{t+1})$.

$$\begin{aligned} v(s) &= \mathbb{E}[G_t \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s] \end{aligned}$$

Dalla formula ottenuta precedentemente è possibile separarla in due parti:

1)

$$v(s) = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]$$



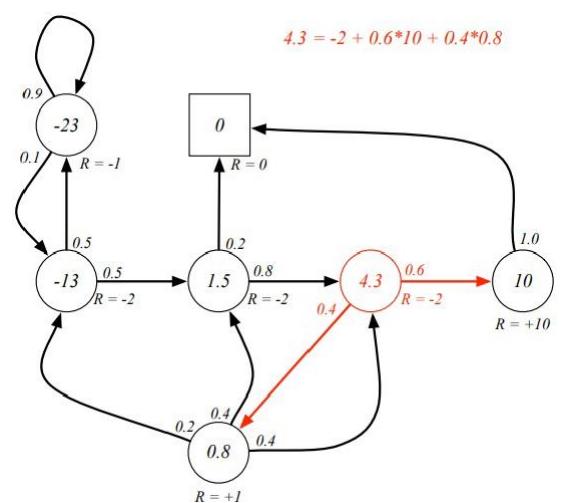
2)

$$\begin{aligned} v(s) &= \underbrace{\mathbb{E}[R_{t+1} \mid S_t = s]}_{\text{Reward function } \mathcal{R}_s} + \gamma \mathbb{E}[v(S_{t+1}) \mid S_t = s] \rightarrow \text{The expected state-value of being in any state reachable from } s \\ &= \mathcal{R}_s + \gamma \sum_{s'} P_{ss'} v(s') \end{aligned}$$

Esempio Bellman Equation per il MRP degli studenti:

Ci vogliamo calcolare la v dello stato rosso (4.3), abbiamo detto che possiamo separarlo in reward dello stato che è $-2 + \gamma$ (la probabilità di tutti gli stati che è possibile raggiungere dallo stato corrente).

$$\text{Quindi è } -2 * (0.4 * 0.8) + (0.6 * 10) = 4.3$$



BELLMAN EQUATION IN FORMA MATRICIALE:

La **Bellman Equation** può essere espressa in modo conciso utilizzando le matrici:
 $v = \mathcal{R} + \gamma \mathcal{P}v$

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \dots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \dots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

dove v è un vettore colonna con una entry per ogni stato.

(somma del vettore delle ricompense + γ * matrice delle transizioni * vettore dei valori che compare due volte)

RISOLUZIONE BELLMAN EQUATION:

È possibile risolvere la formula andando a mettere da parte la v , portando a sinistra γ , \mathcal{P} e v , in modo da avere v in comune, così possiamo andarla a risolvere come $R/(1 - \gamma P)$.

La complessità computazionale è $O(n^3)$ per n stati, è possibile applicare la soluzione diretta solo per MRP di piccole dimensioni.

Esistono molti metodi iterativi per i MRP di grandi dimensioni.

$$\begin{aligned} v &= \mathcal{R} + \gamma \mathcal{P}v \\ (I - \gamma \mathcal{P})v &= \mathcal{R} \\ v &= (I - \gamma \mathcal{P})^{-1} \mathcal{R} \end{aligned}$$

PROCESSI DECISIONALI DI MARKOV:

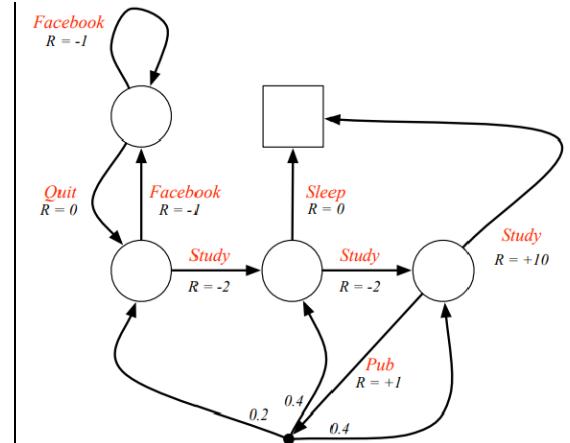
Un processo decisionale di Markov (MDP) non è altro che un processo di ricompensa di Markov con decisioni, dove aggiungiamo un parametro A che rappresenta le azioni. La matrice di probabilità sarà parametrizzata rispetto all'azione, ad esempio, ci spostiamo in S_1 e se siamo in s eseguiamo l'azione a ; quindi, abbiamo un parametro in più quando ci spostiamo da uno stato al successivo, stessa cosa la ricompensa dipenderà dal tipo di azione eseguita.

L'*ambiente* è costituito da stati di Markov. Formalmente, un **processo decisionale di Markov** è una tupla $\langle S, A, P, R, \gamma \rangle$:

- S è un insieme finito di stati;
- **A è un insieme finito di azioni;**
- P è una matrice di probabilità di transizione di stato (come si passa da uno stato all'altro quando si effettua un'azione):
 $P_{ss'}^a = P[S_{t+1} = s' | S_t = s, A_t = a];$
- R è una funzione di ricompensa:
 $R_s^a = E[R_{t+1} | S_t = s, A_t = a];$
- γ è un fattore di sconto, $\gamma \in [0,1]$.

Esempio MDP degli studenti:

In questo caso, gli archi sono le azioni che ci permettono di spostarci in uno stato al successivo:



POLICY:

Una policy è la probabilità di eseguire una certa azione quando l'agente si trova in un certo stato, definisce in modo completo il comportamento di un agente siccome dice cosa deve fare quando si trova in un certo stato. In più, le policy di un MDP dipendono solo dallo stato attuale (e non dalla storia).

Formalmente, una **policy** π è una distribuzione sulle azioni in funzione degli stati:

$$\pi(a|s) = P[A_t = a | S_t = s]$$

Le policy sono **stazionarie**, cioè cambiando il tempo la funzione rimane invariata (funzione che non cambiano nel tempo):

$$A_t \sim \pi(\cdot | S_t), \forall t > 0$$

Definendo tutto in funzione delle policy di ha:

- Un **processo decisionale di Markov** (MDP) $\langle S, A, P, R, \gamma \rangle$ e una **policy** π .
- La **sequenza di stati** S_1, S_2, \dots è un processo di Markov $\langle S, P^\pi \rangle$.
- La **sequenza di stati e ricompense** S_1, R_1, S_2, \dots è un processo di ricompensa di Markov $\langle S, P^\pi, R^\pi, \gamma \rangle$ dove \rightarrow

La matrice di probabilità in funzione della policy ci dice qual è la probabilità di spostarmi da s a s' andando a provare tutte le azioni che posso eseguire allo stato s moltiplicato per la probabilità che l'azione a mi porta da s a s' .

$$\begin{aligned} \mathcal{P}_{s,s'}^\pi &= \sum_{a \in A} \pi(a|s) \mathcal{P}_{ss'}^a \\ \mathcal{R}_s^\pi &= \sum_{a \in A} \pi(a|s) \mathcal{R}_s^a \end{aligned}$$

VALUE FUNCTION:

Da notare che le funzioni sono in funzione della policy, quando bisogna valutare uno stato s lo si fa sempre data una policy.

La **state-value function** $v_\pi(s)$ di un MDP è il guadagno atteso partendo dallo stato s e seguendo la policy π :

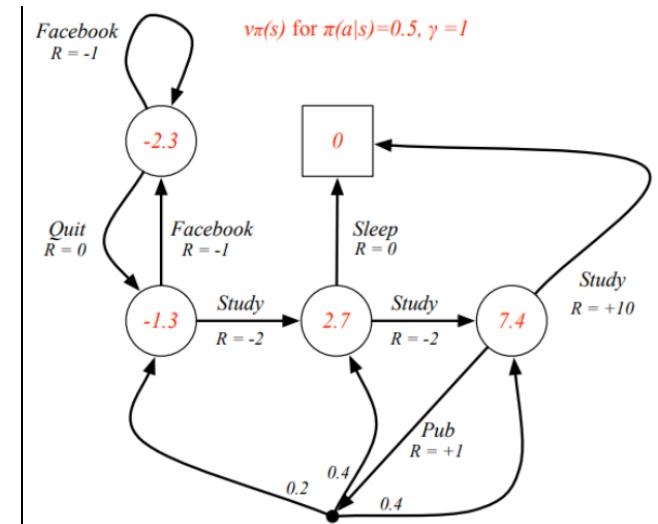
$$v_\pi(s) = E_\pi[G_t \mid S_t = s]$$

La **action-value function** $q_\pi(s, a)$ è il guadagno atteso partendo dallo stato s , eseguendo l'azione a e seguendo la policy π :

$$q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a]$$

Esempio state-value function MDP studenti:

Per calcolare la state-value function andando a considerare la policy, ovvero se mi trovo nello stato s ed eseguo un'azione questa ha probabilità 0.5 (azioni equiprobabili), e gamma $\gamma=1$.



BELLMAN EXPECTATION EQUATION:

La **state-value function** può essere nuovamente scomposta in ricompensa immediata + valore scontato dello stato successivo:

$$v_\pi(s) = \mathbb{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]$$

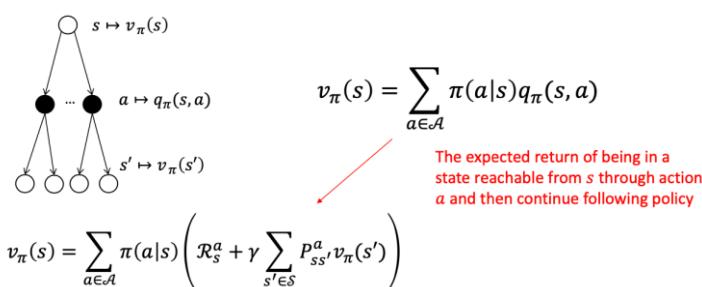
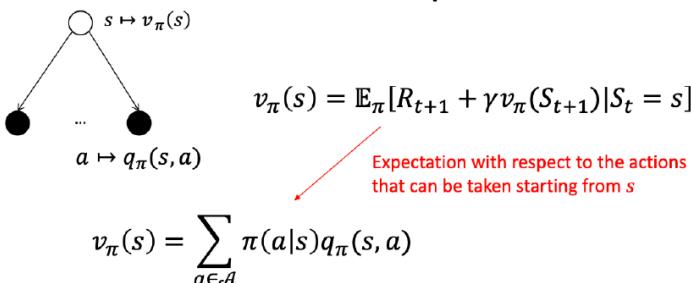
(ricompensa attuale + la ricompensa che otteniamo futura, pesata rispetto a γ)

La **action-value function** può essere scomposta in modo analogo:

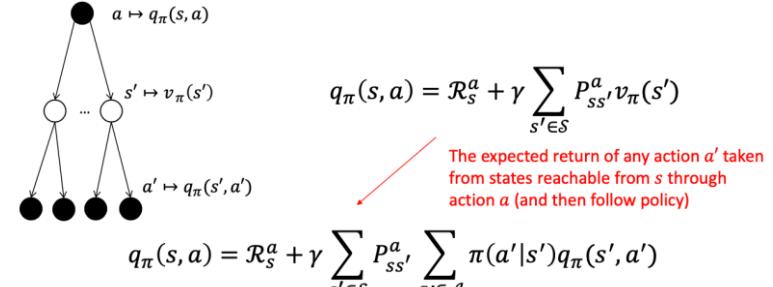
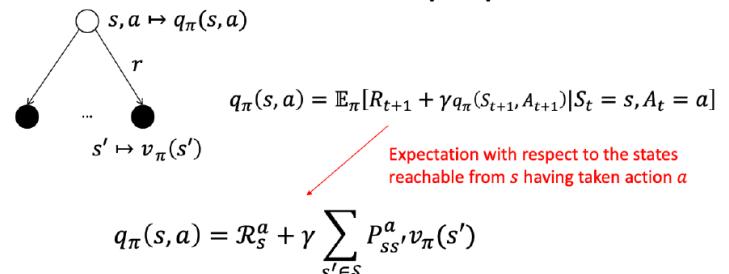
$$q_\pi(s, a) = \mathbb{E}_\pi [R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a]$$

(ricompensa attuale + ricorsivamente la funzione sullo stato S_{t+1} azione A_{t+1} dove s ed a sono all'istante t)

BELLMAN EXPECTATION EQUATION per v_π :



BELLMAN EXPECTATION EQUATION per q_π :



BELLMAN EXPECTATION EQUATION (FORMA MATRICIALE):

La **Bellman Expectation Equation** può essere espressa in modo conciso utilizzando il MRP indotto:

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi$$

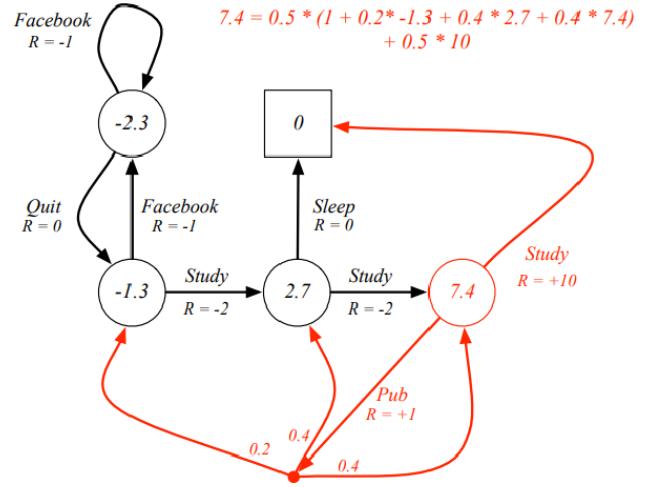
(ricompensa + fattore di sconto * la probabilità di transizione condizionata dalla policy * value function condizionata sempre dalla policy)

con soluzione diretta:

$$v_\pi = (\mathbf{I} - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi$$

Esempio Bellman Expectation Equation per MDP studenti:

Se utilizziamo la formula precedente possiamo calcolare la value function di un nodo. Somma pesata rispetto alla probabilità e 0.5 perché le azioni sono tutte equiprobabili.



OPTIMAL VALUE FUNCTION:

In sintesi: abbiamo un problema di RL, lo andiamo a modellare con un MDP e l'obiettivo finale è andare a identificare la policy che fa ottenere il massimo guadagno. Notare che la ricompensa non bisogna analizzarla nell'istante attuale ma a lungo termine, perché i guadagni possono arrivare anche dopo in certo numero di interazioni.

Per poter definire una policy migliore possibile, bisogna definire la funzione di stato e azione ottimale.

La **optimal state-value function** $v_*(s)$ è la massima value function tra tutte le policy:

$$v_*(s) = \max_{\pi} v_{\pi}(s)$$

La **optimal action-value function** $q_*(s, a)$ è la massima action-value function tra tutte le policy:

$$q_*(s, a) = \max_{\pi} q_{\pi}(s, a)$$

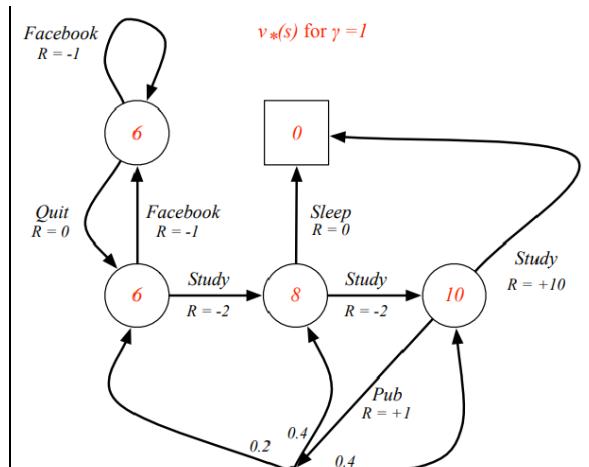
La optimal value function specifica la migliore performance nel MDP.

Un MDP è "risolto" quando si determina l'optimal value function.

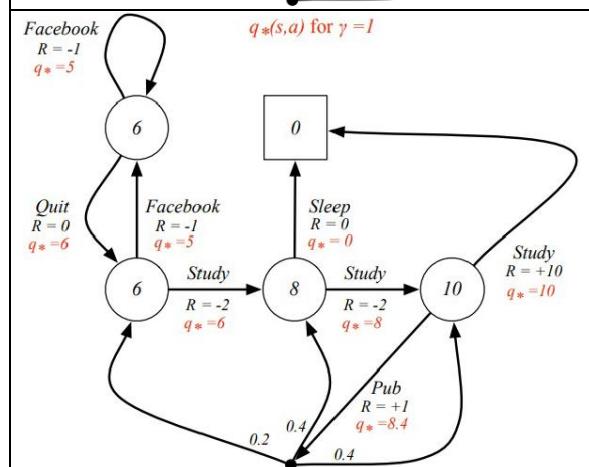
Esempio Optimal State-Value Function per MDP studenti:

$v_*(s)$ per $\gamma=1$ è la funzione ottimale.

I valori in ogni stato sono quelli ottimali. Se si cambiano le policy non si otterrà un valore maggiore di quelli esplicitati.



Esempio Optimal Action-Value Function per MDP studenti:



POLICY OTTIMALE:

Dopo aver definito le funzioni ottimali è possibile definire la **policy ottimale**, che fa ottenere il maggior guadagno per tutti i possibili stati.

Definiamo un ordinamento parziale delle policy:

$$\pi \geq \pi' \text{ if } v_{\pi}(s) \geq v_{\pi'}(s), \forall s$$

(una policy è meglio di un'altra se ottengo dalla value function un valore sempre maggiore da tutti gli stati)

Teorema:

Per ogni processo decisionale di Markov:

- Esiste una policy ottimale π_* che è migliore o uguale a tutte le altre policy, $\pi_* \geq \pi, \forall \pi$;
- Tutte le policy ottimali raggiungono la optimal state-value function, $v_{\pi_*}(s) = v_*(s)$;
- Tutte le policy ottimali raggiungono la optimal action-value function, $q_{\pi_*}(s, a) = q_*(s, a)$.

NOTA: Esistono più policy ottimali e per tanto sono equivalenti, quello che condividono è la funzione state-value e action-value ottimale siccome ne esistono solo una ciascuna ottimale.

INDIVIDUAZIONE POLICY OTTIMALE:

Per individuare una policy ottimale basta identificare un''action-value ottimale, individuata massimizzando $q_*(s, a)$:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} q_*(s, a) \\ 0 & \text{otherwise} \end{cases}$$

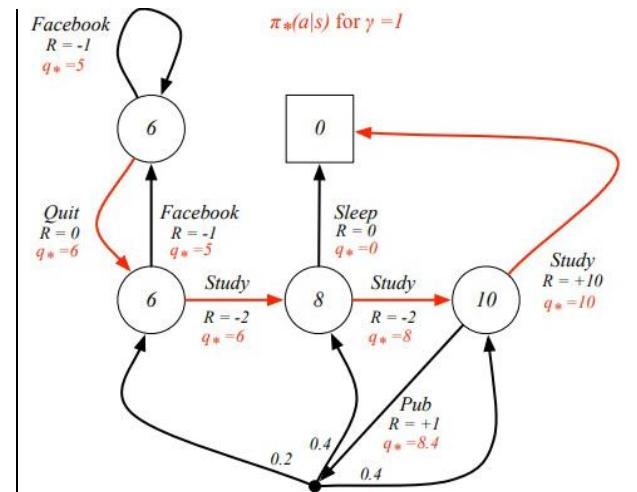
(Quando si esegue l'azione a nello stato s , restituirà 1 se è l'azione che dà il maggior guadagno)

Esiste sempre una policy ottimale deterministica per ogni MDP.

Se $q_*(s, a)$ è noto, si ottiene immediatamente la policy ottimale.

Esempio policy ottimale per MDP studenti:

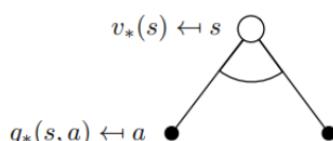
L'agente tenderà ad andare nello stato 10 siccome è il punto di massimo guadagno.



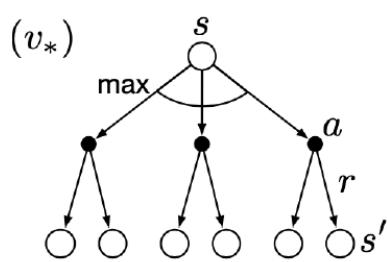
BELLMAN OPTIMALITY EQUATION:

Le optimal value functions sono ricorsivamente correlate dalle Bellman optimality equations:

Per v_* :

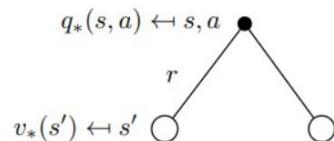


$$v_*(s) \leftarrow s$$

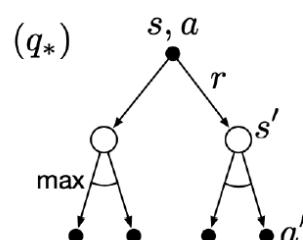


$$v_*(s) = \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_*(s')$$

Per q_* :



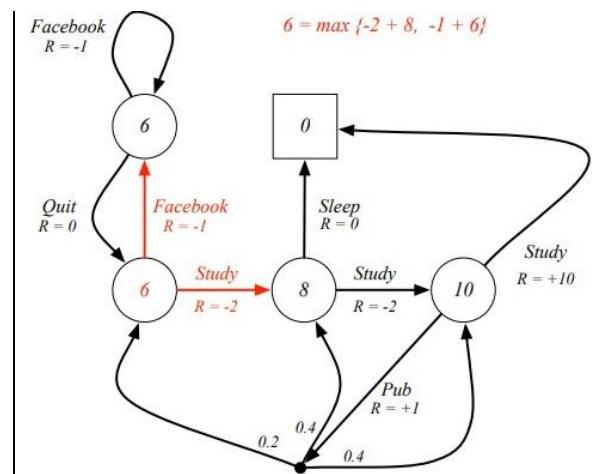
$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_*(s')$$



$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a \max_{a' \in \mathcal{A}} q_*(s', a')$$

Esempio Bellman optimality equation per MDP studenti:

Prendendo lo stato 6, per calcolare questo numero si fa il massimo tra le due azioni, facendo ricompensa + la somma della probabilità * il valore del successivo. Ricordando che la probabilità di eseguire una azione è 1. $\text{Max } \{-2+8, -1+6\} = 6$



RISOLUZIONE BELLMAN OPTIMALITY EQUATION:

Bisogna risolvere le formule precedenti per andare a identificare la policy ottimale e per tanto risolvere un problema di apprendimento per rinforzo. Se consideriamo le equazioni ottimali (in maniera ricorsiva) sono **equazioni non lineari** perché bisogna costruire un sistema dove gli stati variano (abbiamo una equazione per ogni stato). Per poterlo risolvere bisogna per tanto usare dei metodi iterativi (esempio Value Iteration, Policy Iteration, Q-learning e Sarsa). La complessità computazionale è $O(n^3)$ per n stati.

18. PIANIFICAZIONE MEDIANTE PROGRAMMAZIONE DINAMICA

Per **pianificazione** si intende che abbiamo conoscenza del modello e vogliamo identificare la policy ottimale per l'agente. Il problema viene trattato come un problema di programmazione dinamica, è possibile farlo perché abbiamo definito le due function value in maniera ricorsiva.

La programmazione dinamica fu introdotta da Bellman ed è una tecnica di programmazione dove si divide il problema in sotto-problemi, risolvendo i sotto-problemi le soluzioni si combinano per avere la soluzione al problema completo.

Notare che non è il divide-et-impera siccome i sotto-problemi possono avere delle sovrapposizioni.

Esempio di programmazione dinamica è Fibonacci siccome quando viene fatta la ricorsione, il valore successivo lo si calcola in base ai valori precedenti e quindi sotto-problemi ripetuti.

La **programmazione dinamica** (DP) è un metodo generale per la risoluzione di problemi caratterizzati da due proprietà:

- Sottostruttura ottimale, si applica il principio di ottimalità e la soluzione ottima può essere suddivisa in sottoproblemi;
- Overlapping dei sottoproblemi, i sottoproblemi ricorrono più volte e le soluzioni possono essere memorizzate nella cache e riutilizzate.

I **processi decisionali di Markov** soddisfano entrambe le proprietà:

- La Bellman Equation consente una suddivisione ricorsiva
- La value function memorizza e riutilizza le soluzioni

PIANIFICAZIONE TRAMITE PROGRAMMAZIONE DINAMICA:

L'obiettivo è risolvere un problema di pianificazione in un MDP, pertanto:

- un modello dell'ambiente è noto;
- l'agente migliora la sua policy fino a che non si arriva a quella ottimale.

Esistono due fasi:

1. **Predizione**, fase in cui si valuta la policy:

- Input: MDP $\langle S, A, P, R, \gamma \rangle$ ed una policy π
- Output: value function v_π

2. **Controllo**, fase in cui si cerca la policy ottimale:

- Input: MDP $\langle S, A, P, R, \gamma \rangle$
- Output: optimal value function v_* e optimal policy π_*

VALUTARE UNA POLICY (ITERATIVE POLICY EVALUATION):

Per **valutare una policy** può essere fatto **in maniera iterativa**, dove abbiamo una policy π e la possiamo andare a valutare andando a calcolare la value function v per ogni stato del problema in maniera iterativa, cioè vogliamo calcolare la bontà di ogni stato (quanto vado a guadagnare in ogni stato) seguendo una certa policy. In maniera schematica:

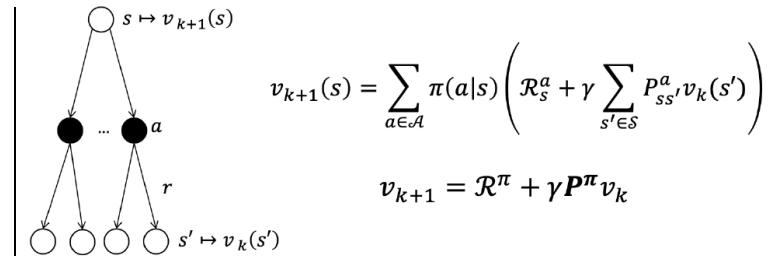
- **Problema:** valutare una policy π
- **Soluzione:** applicazione iterativa del backup della Bellman Expectation $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_\pi$.

Partiamo da un problema non conoscendo nessun valore di v ma conosciamo la policy, per calcolare in valore di v lo si fa in maniera iterativa. Usando la formula ricorsiva si parte da valori iniziali e si applica la funzione v in maniera iterativa finché i valori di $v(s)$ non convergono a qualche valore (ci si ferma quando non ci sono grosse variazioni di guadagno).

L'operazione di aggiornamento viene fatta in maniera sincrona, cioè viene fatta per tutti gli stati.

Utilizzando backup sincroni:

- Ad ogni iterazione $k + 1$;
- Per tutti gli stati $s \in S$;
- Aggiornare $v_{k+1}(s)$ da $v_k(s')$, dove s' è uno stato successore di s .



Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π , the policy to be evaluated

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$ arbitrarily, for $s \in S$, and $V(\text{terminal})$ to 0

Loop:

$$\Delta \leftarrow 0$$

Loop for each $s \in S$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$

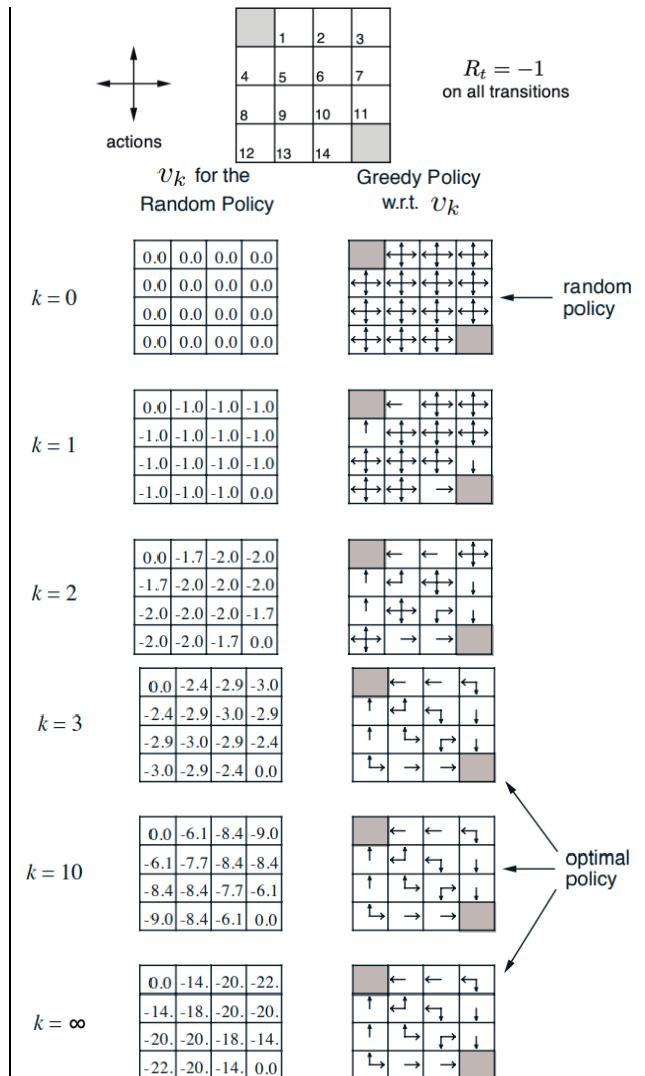
Esempio Gridworld di valutazione di una policy casuale:

- MDP episodica non scontata ($\gamma = 1$), siccome ci sono degli stati terminali (stati in grigio);
- Stati non terminali 1, ..., 14;
- Le azioni che portano fuori dalla griglia lasciano lo stato invariato;
- La ricompensa è -1 fino a quando non viene raggiunto lo stato terminale;
- L'agente segue una policy casuale uniforme:

$$\pi(n|\cdot) = \pi(e|\cdot) = \pi(s|\cdot) = \pi(w|\cdot) = 0.25$$

1. La prima iterazione ($k=0$) tutte le celle partono da 0.0;
2. Nella seconda iterazione ($k=1$) gli stati terminali rimangono a 0.0 mentre tutti gli altri vanno a -1.0, questo perché si fa la sommatoria delle varie probabilità * la ricompensa dell'azione + la probabilità dei successivi. Se si definisce una policy sulla matrice trovata, ci dice le direzioni consigliate (griglie al lato);
3. Nella terza iterazione ($k=2$) i valori cambiano ulteriormente sempre in base alle formule dedicate. La policy ci dice altre direzioni da seguire;

Si rifanno le solite operazioni continuando ad iterare, ma notiamo che otteniamo la stessa policy da $k=3$ in poi, per tanto è possibile fermarsi a $k=3$ anche se la value function non è quella ottimale.



MIGLIORARE UNA POLICY (POLICY ITERATION):

Una volta valutata la policy la si vuole migliorare, questo processo viene fatto un determinato numero di volte con l'obiettivo di trovare appunto la policy ottimale.

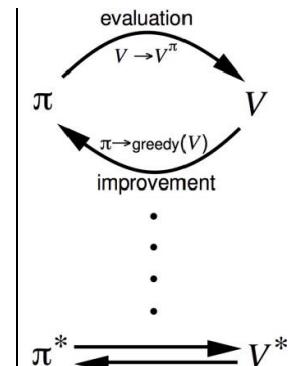
Per migliorare una policy dobbiamo far sì che i valori che otteniamo nella matrice siano migliori della precedente.

- Data una policy π , valutare la policy π : $v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots | S_t = s]$
- Migliorare la policy agendo in maniera greedy rispetto a v_π : $\pi' = \text{greedy}(v_\pi)$

Nell'esempio Gridworld la policy migliorata era ottimale, $\pi' = \pi^*$.

In generale, sono necessarie diverse iterazioni di miglioramento/valutazione. Tuttavia, questo processo di Policy Iteration converge sempre a π_* :

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \dots \xrightarrow{I} \pi_* \xrightarrow{E} v_*$$



POLICY IMPROVEMENT:

Per effettuare l'operazione di miglioramento delle policy, consideriamo una policy deterministica $a = \pi(s)$ (la policy π applicata ad un certo stato s porta a scegliere l'azione a), possiamo *migliorare la policy agendo in modo greedy*:

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}} q_\pi(s, a)$$

(si prende, tra tutte le azioni possibili in s , quella che porta un action-value maggiore)

Questo *migliora il valore di qualsiasi stato s* rispetto ad uno step, ne consegue che la nuova policy sarà \geq alla precedente:

$$q_\pi(s, \pi'(s)) = \underset{a \in \mathcal{A}}{\max} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s)$$

Pertanto, migliora la value function, $v_{\pi'}(s) \geq v_\pi(s)$.

Se i miglioramenti terminano, di conseguenza risulta l'uguaglianza, abbiamo raggiunto la policy ottimale:

$$q_\pi(s, \pi'(s)) = \underset{a \in \mathcal{A}}{\max} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s)$$

Allora la Bellman Optimality Equation è stata soddisfatta:

$$v_\pi(s) = \underset{a \in \mathcal{A}}{\max} q_\pi(s, a)$$

Pertanto, $v_\pi(s) = v_*(s)$ per tutti gli $s \in S$. Quindi **π è una policy ottimale**.

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization

$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(\text{terminal}) \doteq 0$

2. Policy Evaluation

Loop:

$$\Delta \leftarrow 0$$

Loop for each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s)) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement

policy-stable \leftarrow true

For each $s \in \mathcal{S}$:

$$\text{old-action} \leftarrow \pi(s)$$

$$\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

If $\text{old-action} \neq \pi(s)$, then *policy-stable* \leftarrow false

If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Esempio Autolavaggio:

- **Stati:** Due sedi, massimo 20 auto in ciascuna sede;
- **Ricompensa:** 10\$ per ogni auto noleggiata (l'auto deve essere disponibile per quella sede);
- **Azioni:** Spostare fino a 5 auto da una sede all'altra durante la notte (costo 2\$ per ogni auto), siccome può capitare che una sede fa più noleggi dell'altra e per tanto si rendono disponibili le auto in quella che noleggia di più;
- **Transizioni:** Auto restituite e richieste in modo casuale:
 - Si suppone che ci sia una distribuzione di Poisson, n restituzioni/richieste con probabilità $\sim \lambda^n/n!$ e λ ;

1. Prima sede: media delle richieste = 3, media delle restituzioni = 3

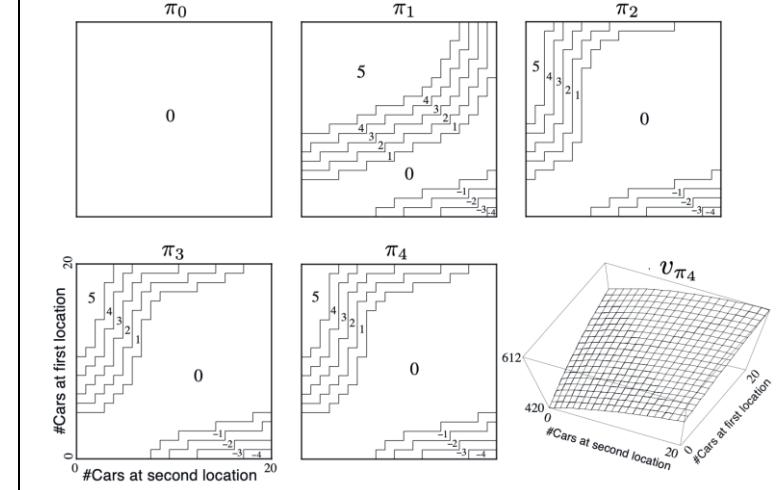
2. Seconda sede: media delle richieste = 4, media delle restituzioni = 2

La seconda sede ha più richieste che restituzioni, pertanto se si utilizza l'algoritmo di policy iteration, partendo da una policy dove tutto va a 0, dobbiamo decidere le auto da spostare tra le sedi.

Sulle ordinate abbiamo in numero di auto della prima sede e sulle ascisse della seconda sede.

La prima policy abbiamo 5 pertanto è il numero di auto da spostare tra le sedi.

Andando ad iterare ulteriormente, la 3° e 4° esecuzione hanno la stessa policy, quindi quella è la policy ottimale.



VALUE ITERATION:

È possibile definire un algoritmo chiamato **value iteration** che combina le fasi viste in una unica formula e si basa su come è fatta una policy ottimale. Ogni **policy ottimale** può essere suddivisa in due componenti:

- Una prima azione ottimale A_* ;
- Seguita da una policy ottimale dallo stato successore s' .

Teorema (Principio di Ottimalità):

Una policy $\pi(a|s)$ raggiunge il valore ottimale dallo stato s (cioè $v_\pi(s) = v_*(s)$) se e solo se per ogni stato s' raggiungibile da s , il valore della policy π raggiunge il valore ottimale dallo stato s' , $v_\pi(s') = v_*(s')$.

Se si conosce la soluzione ottimale dei sotto-problemi (i successivi) $v_*(s')$ è possibile usarli per ottenere la soluzione ottimale su $v_*(s)$. Può essere identificata con un lookahead di un solo passo:

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s')$$

(Se si conosce il valore ottimale su s' che è un successivo, è possibile determinare il valore ottimale sullo stato s , facendo il massimo tra tutte le azioni che sono in a , cioè prendo l'azione che mi fa ottenere il valore maggiore)

L'idea della value iteration consiste nell'applicare questi aggiornamenti iterativamente.

Esempio shortest path:

Vogliamo il percorso più breve da g agli altri stati.

Inizialmente il calcolo viene fatto sulla base del valore che conosciamo (inizialmente è 0).

Successivamente si aggiornano man mano gli altri valori.

La formula non sceglie in base alla policy, essa è il risultato finale di questi calcoli.

g				

Problem	v_1	v_2	v_3

v_4	v_5	v_6	v_7

- **Problema:** trovare la policy ottimale π

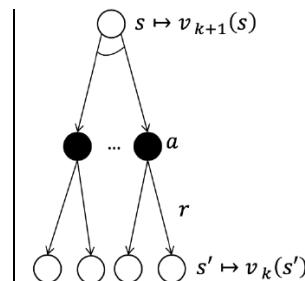
- **Soluzione:** applicazione iterativa del backup della Bellman Optimality $v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_*$.

Utilizzando backup sincroni:

- Ad ogni iterazione $k+1$;
- Per tutti gli stati $s \in S$;
- Aggiornare v_{k+1} da $v_k(s')$.

A differenza della policy iteration, **non c'è una policy esplicita**. La policy la si ricava alla fine quando l'algoritmo converge.

Le intermediate value function potrebbero non corrispondere a nessuna policy.



$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} P_{ss'}^a v_k(s') \right)$$

$$v_{k+1} = \max_{a \in \mathcal{A}} (\mathcal{R}^a + \gamma P^a v_k)$$

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop:

```

|   Δ ← 0
|   Loop for each  $s \in \mathcal{S}$ :
|      $v \leftarrow V(s)$ 
|      $V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$ 
|     Δ ← max(Δ, |v - V(s)|)
until Δ < θ

```

Output a deterministic policy, $\pi \approx \pi_*$, such that

$$\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$$

ALGORITMI DI PROGRAMMAZIONE DINAMICA SINCRONA:

In sintesi, abbiamo visto i seguenti algoritmi:

Problema	Bellman Equation	Algoritmo
Predizione	Bellman Expectation Equation	Iterative Policy Evaluation
Controllo	Bellman Expectation Equation + Greedy Policy Improvement	Policy Iteration
Controllo	Bellman Optimality Equation	Value Iteration

Il problema è sempre la **complessità**. Gli algoritmi sono basati sulla *state-value function* $v_\pi(s)$ o $v_*(s)$:

- Complessità $O(mn^2)$ per iterazione, per m azioni ed n stati.

Potrebbero essere applicati anche alla *action-value function* $q_\pi(s, a) = q_*(s, a)$:

- Complessità $O(m^2n^2)$ per iterazione.

PROGRAMMAZIONE DINAMICA ASINCRONA:

Per migliorare questi algoritmi, siccome potrebbero non essere utili quando si hanno tanti stati, esistono diverse tecniche. Il problema sostanziale di questi algoritmi è che lavorano in maniera sincrona, cioè fanno delle iterazioni su tutti gli stati (chiamato sweep), aggiorna i valori per poi passare allo sweep successivo. Se si hanno troppi stati un algoritmo del genere non si può usare.

Un possibile miglioramento è quello di passare ad una **versione asincrona**, dove la programmazione dinamica asincrona esegue il backup degli stati singolarmente, in qualsiasi ordine. Esistono tre semplici approcci:

- **Programmazione dinamica in-place**, avendo una matrice dove bisogna calcolare dei nuovi valori, o li calcoliamo in una matrice a parte oppure farlo in-place, cioè nella stessa matrice si fanno gli aggiornamenti;
- **Prioritised sweeping**, gli stati non sono tutti uguali ma vengono aggiornati in base a dei criteri (esempio errore Bellman);
- **Programmazione dinamica real-time**, si prendono gli stati più rilevanti da aggiornare e si danno priorità a questi stati che vengono scelti man mano da un altro agente.

L'idea generale è quella di aggiornare un sottoinsieme di stati e non tutti, così da ridurre la computazione.

PROGRAMMAZIONE DINAMICA IN-PLACE:

La value iteration sincrona memorizza **due copie della value function**. Per ogni s in S :

$$\begin{aligned} v_{\text{new}}(s) &\leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_{\text{old}}(s') \\ v_{\text{old}}(s) &\leftarrow v_{\text{new}}(s) \end{aligned}$$

La value iteration in-place memorizza solo **una copia della value function**. Per ogni s in S :

$$v(s) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right)$$

PRIORITISED SWEEPING:

Utilizza il **Bellman error** per guidare la selezione degli stati, ad esempio:

$$\left| \max_{a \in \mathcal{A}} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v(s') \right) - v(s) \right|$$

(Se si va ad applicare la formula di Bellman – il valore associato allo stato, quello con l'istanza più alta lo si va a considerare a priorità maggiore negli aggiornamenti)

Esegue il backup dello stato con il Bellman error residuo più grande. Aggiorna il Bellman error degli stati interessati dopo ogni backup. Richiede la conoscenza delle dinamiche inverse (stati predecessori). Può essere implementato in modo efficiente tramite l'utilizzo di una coda di priorità.

PROGRAMMAZIONE DINAMICA REAL-TIME:

Si usa un agente che effettua la scelta dello stato, entità a parte in grado di capire lo stato migliore da dover aggiornare.

Utilizza l'esperienza dell'agente per guidare la selezione degli stati:

- Dopo ogni time-step S_t, A_t, R_{t+1}
- Esegue il backup dello stato S_t

$$v(S_t) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}_{S_t}^a + \gamma \sum_{s' \in S} P_{S_t s'}^a v(s') \right)$$

BACKUP FULL-WIDTH:

Un problema degli algoritmi di programmazione dinamica è che per alcuni problemi la quantità di successori è elevata.

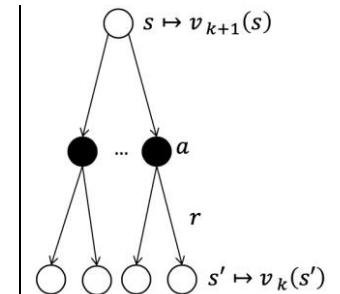
Più precisamente, quando si fa l'analisi andando a costruire il diagramma di backup, lo si costruisce con full-width, cioè si va a considerare tutti i successori, vedendo tra tutte le azioni quali sono le migliori.

Per ogni backup (sincrono o asincrono):

- Viene considerato ogni stato e azione successiva;
- Si utilizza la conoscenza delle transizioni del MDP e della funzione di ricompensa.

La DP è efficace per problemi di medie dimensioni (milioni di stati). Per problemi di grandi dimensioni la DP è soggetta alla **curse of dimensionality**, cioè il numero di stati $n = |S|$ cresce esponenzialmente con il numero di variabili di stato.

Anche un solo backup può essere troppo costoso.



19. MODEL FREE PREDICTION

Il problema è che si vuole determinare una policy ottimale senza conoscere il modello, cioè non si ha conoscenza della funzione di reward, per sapere la ricompensa bisogna per forza eseguire un'azione. Le transizioni non sono note quindi si esegue l'azione e si vede dove ci si ritrova. In questo caso, non è possibile fare pianificazione siccome non abbiamo conoscenza dell'ambiente.

- **Model free:** Nessun modello dell'ambiente e nessuna conoscenza delle transizioni/ricompense del MDP;
- **Model-free prediction:** Stimare la value function di un MDP non noto;
- **Model-free control:** Ottimizzare la value function di un MDP non noto.

MONTE-CARLO (MC) REINFORCEMENT LEARNING:

Il primo approccio applicabile è utilizzare delle simulazioni (**metodo MC**), andando a simulare una sequenza di interazioni tra l'agente e l'ambiente. Le simulazioni di MC prevedono queste interazioni finché non si arriva a degli stati terminali. I modelli di MC vengono utilizzati per MDB episodici, cioè la sequenza di interazioni deve arrivare sempre a una fine.

- MC è **model-free**: nessuna conoscenza delle transizioni/ricompense del MDP;
- **Model-free control:** Ottimizzare la value function di un MDP non noto.

MONTE-CARLO POLICY EVALUATION:

L'obiettivo è che per tutti gli stati si vuole stimare quant'è la ricompensa che sarà una media delle ricompense che si sono ottenute nelle varie simulazioni.

Episodio: azione -> osservazione -> reward, azione -> osservazione -> reward, ecc..., fino a raggiungere uno stato terminale. Si generano tanti episodi, si calcolano le reward, infine si fa la media di tutte le reward e il risultato è la reward media di quello stato. L'unica cosa che è nota è la policy, pertanto le simulazioni devono essere in accordo con la policy.

Formalmente, **apprendere v_π da episodi di esperienza** in base alla policy π :

$$S_1, A_1, R_2, \dots, R_k \sim \pi$$

Ricordiamo che il guadagno è la ricompensa totale scontata:

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$$

Ricordiamo che la value function è il guadagno atteso:

$$v_\pi(s) = \mathbb{E}[G_t | S_t = s]$$

La Monte-Carlo policy evaluation utilizza il **guadagno medio empirico** al posto del guadagno atteso, siccome vengono fatte delle simulazioni ed in base a queste dipenderà la soluzione.

FIRST-VISIT MONTE-CARLO POLICY EVALUATION:

Per andare a determinare questi valori di guadagno esistono due versioni, il primo è il **first-visit MC policy evaluation**. L'idea è che il guadagno G per un certo stato s è dato solo dal valore che si ottiene alla prima visita.

Per valutare lo stato s, **il primo time-step t** in cui lo stato s viene visitato in un episodio:

- Incremento del contatore $N(s) \leftarrow N(s) + 1$;
- Incremento del guadagno totale $S(s) \leftarrow S(s) + G(t)$;
- Stima del valore dal guadagno medio $V(s) = S(s)/N(s)$ (*guadagno ottenuto / il numero di volte*).

Per la legge dei grandi numeri, questo valore al crescere del numero di volte che ci si passa sopra va a convergere con quello atteso:

$$V(s) \rightarrow v_\pi(s) \text{ as } N(s) \rightarrow \infty$$

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated

Initialize:

$$V(s) \in \mathbb{R}, \text{ arbitrarily, for all } s \in \mathcal{S}$$

$Returns(s) \leftarrow \text{an empty list, for all } s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$$G \leftarrow 0$$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$$G \leftarrow \gamma G + R_{t+1}$$

Unless S_t appears in S_0, S_1, \dots, S_{t-1} :

Append G to $Returns(S_t)$

$$V(S_t) \leftarrow \text{average}(Returns(S_t))$$

EVERY-VISIT MONTE-CARLO POLICY EVALUATION:

La seconda versione è la ***every-visit***, cioè se nell'episodio compare più volte s si conteggiano anche gli altri guadagni, pertanto è una media complessiva.

Per valutare lo stato s , ***ogni time-step t*** in cui lo stato s viene visitato in un episodio:

- Incremento del contatore $N(s) <- N(s) + 1$;
- Incremento del guadagno totale $S(s) <- S(s) + G(t)$;
- Stima del valore dal guadagno medio $V(s) = S(s)/N(s)$ (*guadagno ottenuto / il numero di volte*).

Anche qui come prima, per la legge dei grandi numeri:

$$V(s) \rightarrow v_{\pi}(s) \text{ as } N(s) \rightarrow \infty$$

Più campioni si ottengono con quello stato più il valore che si ottiene si avvicina a quello atteso.

Esempio blackjack:

Il giocatore ha due carte mentre il banco ha una carta scoperta, l'obiettivo del gioco è avere un punteggio maggiore del banco (max 21). L'asso vale sia 1 che 11. Il giocatore può chiedere altre carte (twist) o fermarsi (stick).

Stati (in totale 200):

- Somma attuale (12-21)
- Carta scoperta dal banco (asso-10)
- Ho uno *usable ace*? (si-no)

Ricompensa per l'azione ***stick*** (smettere di ricevere carte (e terminare)):

- +1 se la somma delle carte > somma delle carte del banco;
- 0 se la somma delle carte = somma delle carte del banco;
- -1 se la somma delle carte < somma delle carte del banco.

Ricompensa per l'azione ***twist*** (prendere un'altra carta (senza sostituirla)):

- -1 se la somma delle carte > 21 (e termina);
- 0 altrimenti;
- Transizioni: automaticamente twist se la somma delle carte è < 12.

VALUE FUNCTION DEL BLACKJACK DOPO MC LEARNING:

Facendo delle simulazioni di MC possiamo stimare la value function, ma si ha bisogno di una policy che è la seguente:

Policy: stick se la somma delle carte è ≥ 20 , altrimenti twist.

In base a tale policy bisogna vedere il guadagno stimato nei vari stati.

I grafici al lato si distinguono se uno ha un asso tra le prime due carte oppure no. Dopo 10mila episodi e si ha un asso usabile c'è molta incertezza sulla ricompensa siccome porta a risultati molto variabili. Dopo 500mila episodi il valore degli stati converge ad un certo punto siccome ci sono molto meno variazioni rispetto al precedente.

All'aumentare delle simulazioni il valore che viene stimato converge quello atteso. Da notare che questo problema risolverlo con la programmazione dinamica non era utilizzabile perché è difficile costruire un modello per questo gioco, costruire un modello con tutti gli stati è più difficile che fare simulazioni.

MEDIA INCREMENTALE:

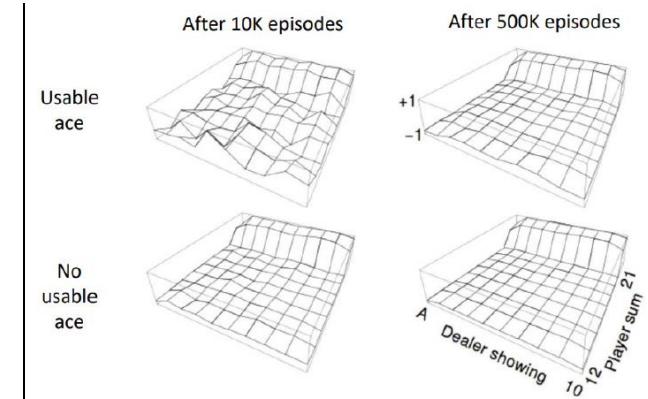
L'algoritmo di MC, il valore che va ad associare ai vari stati (value function) è la media contenuto in ogni stato. Questa media può essere scritta sotto forma della seguente formula.

La media μ_1, μ_2, \dots di una sequenza x_1, x_2, \dots può essere calcolata in modo incrementale:

$$\mu_k = \frac{1}{k} \sum_{j=1}^k x_j = \frac{1}{k} \left(x_k + \sum_{j=1}^{k-1} x_j \right)$$

$$\mu_k = \frac{1}{k} (x_k + (k-1)\mu_{k-1}) = \mu_{k-1} + \frac{1}{k} (x_k - \mu_{k-1})$$

(*Questa media può essere separata in ultimo elemento (k) + tutti gli altri (k-1)*)



AGGIORNAMENTO MEDIA INCREMENTALE DEL MC:

La value function utilizzata dall'algoritmo di MC effettua:

- Aggiorna $V(s)$ in modo incrementale dopo l'episodio $S_1, A_1, R_2, \dots R_t$
- Per ogni stato S_t con guadagno G_t :
 - Incremento del contatore $N(s) <- N(s) + 1$
 - Aggiornamento della value function (con media incrementale)

$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)} (G_t - V(S_t))$$

(media precedente + 1 sul numero degli elementi * (la differenza del guadagno attuale – la vecchia stima))

Questa formula si scrive in maniera più generale, con α che è un parametro dell'algoritmo:

In problemi non stazionari si traccia una media mobile (dimenticando i vecchi episodi)

$$\begin{aligned} V(S_t) &\leftarrow V(S_t) + \alpha(G_t - V(S_t)) \\ \text{NewEstimate} &\leftarrow \text{OldEstimate} + \text{StepSize} [\text{Target} - \text{OldEstimate}] \end{aligned}$$

(la nuova stima della value function è data dalla vecchia stima + step size α abbastanza piccolo * (la differenza del guadagno attuale – la vecchia stima))

TEMPORAL-DIFFERENCE (TD) LEARNING:

L'**algoritmo Temporal-Difference** combina i vantaggi della programmazione dinamica e i vantaggi di MC.

Il limite principale dell'algoritmo di MC è il fatto che si devono costruire delle simulazioni che portano a stati terminali, ma se il problema che si sta risolvendo è continuo MC non può essere utilizzato, siccome stati terminali non esistono (non ci sono episodi). I metodi TD apprendono direttamente da episodi di esperienza.

- TD è **model-free**: nessuna conoscenza delle transizioni/ricompense del MDP;
- TD **apprende da episodi incompleti**, tramite *bootstrapping*, cioè da approssimazioni, facendo una stima da altre stime;
- TD **aggiorna un'ipotesi verso un'altra ipotesi**.

L'obiettivo è lo stesso, apprendere v_π da episodi di esperienza in base alla policy π .

Se si usa MC si aggiorna il valore $V(S_t)$ per ogni stato in funzione del guadagno effettivo G_t :

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

La TD Learning, invece, va ad aggiornare la stima sullo stato in funzione dello stato precedente poi va a considerare come guadagno atteso quello dello stato vicino (S_{t+1}):

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

TD target
TD error δ_t

(Ricompensa da S_t a S_{t+1} + ricompensa che si ottiene nello stato S_{t+1})

Notare che G_t (nella formula MC) è calcolato su una simulazione, mentre TD target (nella formula TD) è calcolato come la ricompensa nell'andare in S_{t+1} più una stima che si ha nello stato S_{t+1} .

Tabular TD(0) for estimating v_π

```

Input: the policy  $\pi$  to be evaluated
Algorithm parameter: step size  $\alpha \in (0, 1]$ 
Initialize  $V(s)$ , for all  $s \in \mathcal{S}^+$ , arbitrarily except that  $V(\text{terminal}) = 0$ 
Loop for each episode:
    Initialize  $S$ 
    Loop for each step of episode:
         $A \leftarrow$  action given by  $\pi$  for  $S$ 
        Take action  $A$ , observe  $R, S'$ 
         $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$ 
         $S \leftarrow S'$ 
    until  $S$  is terminal
  
```

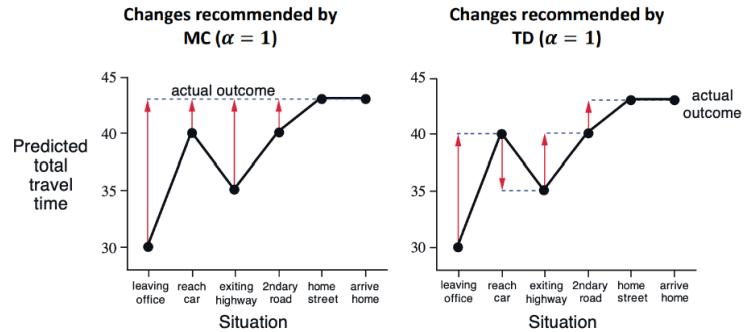
Esempio guida verso casa:

Si vuole predire il tempo di arrivo a casa, facendo aggiornamento in maniera continua in base agli episodi che capitano (es tabella a fianco).

State	Elapsed Time (minutes)	Predicted Time to Go	Predicted Total Time
leaving office	0	30	30
reach car, raining	5	35	40
exit highway	20	15	35
behind truck	30	10	40
home street	40	3	43
arrive home	43	0	43

Notare che questo è un problema online siccome le cose cambiano nel tempo e con MC non può essere risolto, siccome non è possibile fare la simulazione visto che gli eventi capitano durante la giornata.

Gli aggiornamenti con TD sono fatti in funzione del vicinato.



DIFFERENZE TRA MONTE-CARLO E TEMPORAL-DIFFERENCE LEARNING:

Con la TD si è in grado di **apprendere senza conoscere il risultato finale** (non è necessaria la presenza di stati terminali), non si ha necessità di fare simulazioni perché il calcolo viene effettuato facendo un'azione e vedendo il valore dello stato in cui si arriva. Al contrario, MC funziona solo in ambienti episodici (terminanti).

Il TD può essere usato in contesti online, mentre con MC bisogna fare un'analisi offline con simulazioni complete.

BIAS-VARIANCE TRADEOFF:

MC usa il guadagno per fare l'aggiornamento della funzione di valutazione, il calcolo che effettua essendo una simulazione la stima che fa è una stima imparziale, pertanto:

Il guadagno $G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$ è una **stima imparziale (unbiased)** di $v_\pi(S_t)$.

Invece se si considera il valore target che si usa nella TD si può osservare che $R_{t+1} + \gamma v_\pi(S_{t+1})$ è una **stima imparziale (unbiased)** di $v_\pi(S_t)$.

Però nell'algoritmo TD si utilizza una stima dello stato successivo, quindi essendo una stima si può dire che potrebbe essere soggetta a bias, quindi il target di TD $R_{t+1} + \gamma V(S_{t+1})$ è una **stima distorta (biased)** di $v_\pi(S_t)$.

MC ha una **varianza** più alta perché si calcola il guadagno su tante azioni, pertanto, si hanno molti risultati variabili, invece con TD si va a prendere solo una azione pertanto non si ha varianza perché l'azione scelta è in funzione della policy.

Esempio random walk:

Si hanno 5 stati, sugli archi c'è la ricompensa, si ha una ricompensa (maggiore) se si va nello stato terminale a destra.

Si vuole stimare la ricompensa di ogni stato.

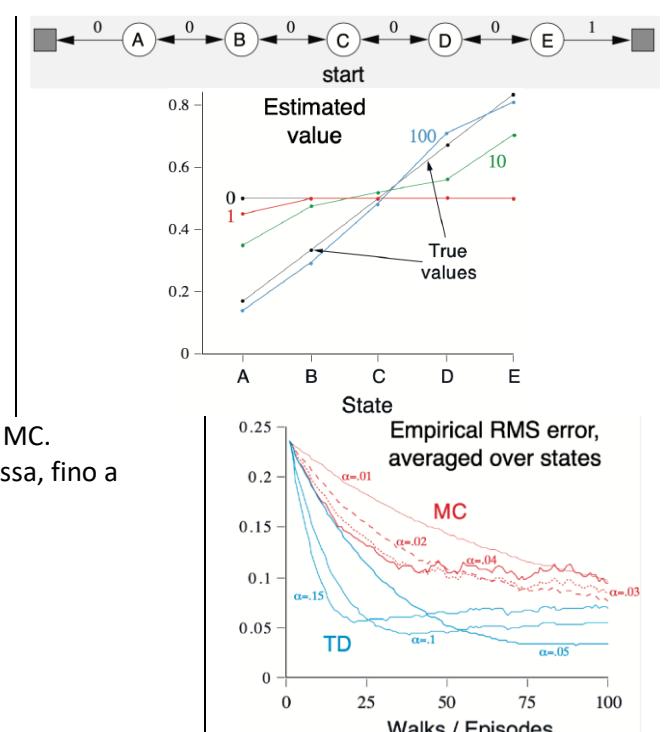
Supponiamo di partire da C e in maniera equiprobabile possiamo andare a sx o dx, per poi andare in uno dei due stati finali.

In E si è molto vicini ad ottenere la ricompensa 1 ed il valore finale è $5/6=0.8$, quella di D è $4/6=0.6$, di C è $3/6=0.5$, di B è $2/6=0.3$ e di A è $1/6=0.1$.

Più ci si allontana dal target (stato con ricompensa 1) minore è la ricompensa.

Il grafico mostra la convergenza dell'algoritmo del TD con quello di MC.

Cambiando i valori di α , all'aumentare degli episodi l'errore si abbassa, fino a tendere a 0.



BATCH MC E TD:

All'aumentare dell'esperienza il valore di V stimato converge col valore reale v_π .

MC e TD convergono: $V(s) \rightarrow v_\pi(s)$ con l'esperienza $\rightarrow \infty$

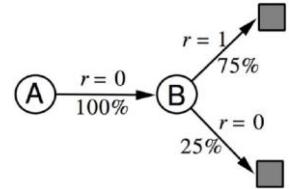
Il problema è che non si hanno molti episodi a disposizione. Per risolvere il problema, poiché sono algoritmi iterativi, si utilizzano gli episodi a disposizione più volte, costruendo dei **batch di episodi** che si utilizzano per aggiornare la value function per ridurre l'errore.

Esempio:

Due stati A e B; nessuna scontistica; 8 episodi di esperienza: (stato, ricompensa)

1. A, 0, B, 0	3. B, 1	5. B, 1	7. B, 1
2. B, 1	4. B, 1	6. B, 1	8. B, 0

Si vuole apprendere $V(A)$ e $V(B)$. Lo si può fare con MC e TD.



Con TD per stimare $V(B)$, per 6 volte si ottiene 1 e per 2 ottieni 0, pertanto il 75% delle volte è 1 e 25% è 0.

Per determinare $V(A)$, secondo MC, bisogna fare la somma di tutte le ricompense e si ottiene 0. Invece, con la TD su $V(A)$ si esegue un'azione e si va in B, usiamo il valore stimato per B per calcolare il valore di A.

CERTAINTY EQUIVARIANCE:

MC converge alla soluzione con il **minimo errore quadratico medio**, pertanto si adatta ai dati ed ha un miglior adattamento ai guadagni osservati:

$$\sum_{k=1}^K \sum_{t=1}^{T_k} (G_t^k - V(s_t^k))^2$$

TD converge alla soluzione del **modello di Markov a massima verosimiglianza**, trova la soluzione del MDP $\langle \mathcal{S}, \mathcal{A}, \mathbf{P}, \mathcal{R}, \gamma \rangle$ che meglio si adatta ai dati:

$$\hat{P}_{ss'}^a = \frac{1}{N(s, a)} \sum_{k=1}^K \sum_{t=1}^{T_k} \mathbf{1}(s_t^k, a_t^k, s_{t+1}^k; s, a, s')$$

$$\hat{\mathcal{R}}_s^a = \frac{1}{N(s, a)} \sum_{k=1}^K \sum_{t=1}^{T_k} \mathbf{1}(s_t^k, a_t^k; s, a) r_t^k$$

TEMPORAL-DIFFERENCE

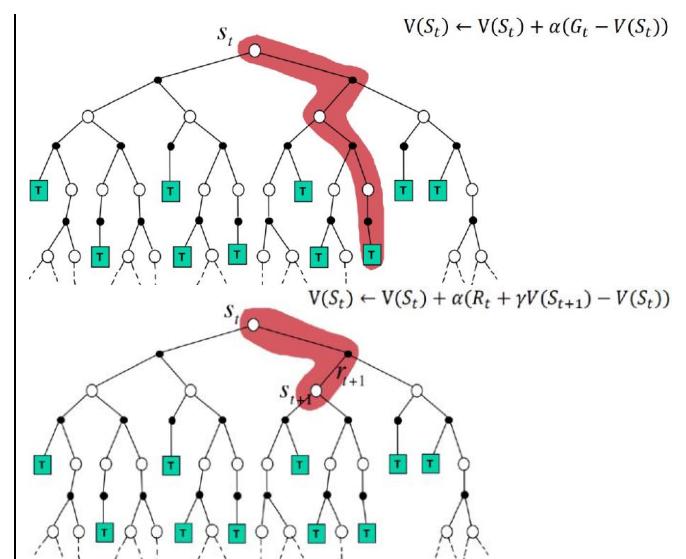
1. può apprendere online dopo ogni step;
2. può apprendere da sequenze incomplete;
3. funziona in ambienti continui (non terminanti);
4. ha una bassa varianza ma alcuni bias;
5. solitamente più efficiente di MC;
6. TD(0) converge a $v_\pi(s)$ (ma non sempre con l'approssimazione della funzione);
7. più sensibile al valore iniziale;
8. sfrutta la proprietà di Markov, siccome si basa solo sullo stato precedente.

MONTE-CARLO

1. deve aspettare la fine dell'episodio prima di conoscere il guadagno;
2. può apprendere solo da sequenze complete;
3. funziona solo in ambienti episodici (terminanti);
4. ha un'alta varianza e zero bias;
5. buone proprietà di convergenza (anche con approssimazione della funzione);
6. non è molto sensibile al valore iniziale;
7. molto semplice da comprendere e utilizzare, ma non sempre applicabile;
8. non sfrutta la proprietà di Markov, ha bisogno di tutta la sequenza di stati.

VISIONE UNIFICATA:

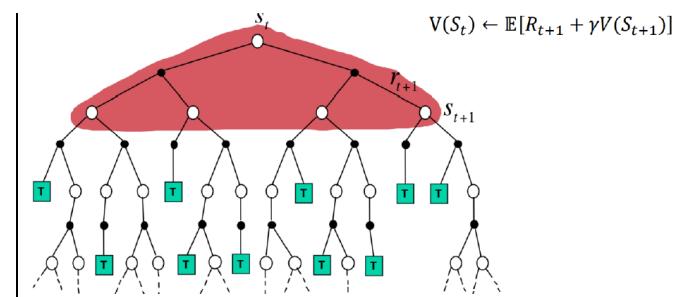
MC per stimare la value function per un certo stato, si fa una simulazione da S_t fino ad uno stato finale e calcola $V(S_t)$ facendo la somma di tutti i guadagni che trova applicando la formula a lato.



Invece, **TD** aggiorna il valore per un certo stato S_t andando a considerare un'unica azione che parte da S_t ed aggiornando il valore dello stato usando il valore che sta nello stato successivo. La stima è fatta usando altre stime.

L'algoritmo di programmazione dinamica (**DP**) va a fare il calcolo della funzione di valutazione calcolando il valore atteso e per fare ciò bisogna analizzare tutti gli stati successivi.

Ovviamente c'è la necessità di conoscerli gli stati successivi e quindi si deve avere un modello.



I componenti importanti di questi algoritmi sono:

- **Bootstrapping** - L'aggiornamento prevede una stima, pertanto:

- MC non esegue il bootstrap, perché i valori sono simulazioni, pertanto sono valori reali;
- DP esegue il bootstrap, parte da valori iniziali e si aggiorna man mano, sono stime che convergono poi ai valori esatti;
- TD esegue il bootstrap, stessa cosa della DP.

- **Sampling** - L'aggiornamento campiona una previsione

- MC esegue il sampling;
- DP non esegue il sampling;
- TD esegue il sampling, va a considerare solo un campione.

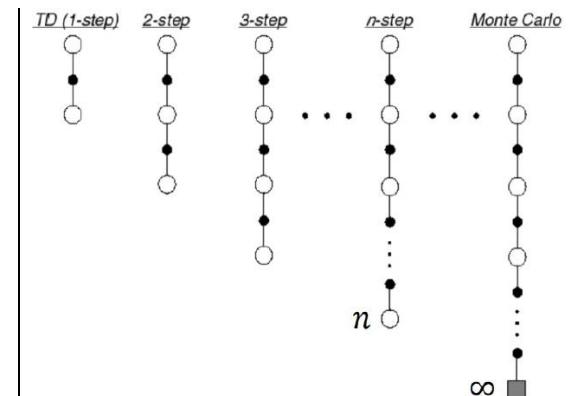
GENERALIZZAZIONE TD:

Esiste un algoritmo ibrido tra **Temporal-Difference** e **Monte-Carlo**.

Dal grafico al lato TD ha profondità 1 mentre MC ha profondità fino ad uno stato terminale (può non esserci ed andare all'infinito).

L'idea è trovare una soluzione intermedia, cioè definire una versione TD dove la profondità della simulazione non è per forza 1, ma un numero di step scelto n .

Esiste la versione con 2-step dove si eseguono 2 azioni e si considerano i guadagni dei due più quello stimato sull'ultimo stato e così via.



Consideriamo le formule con i seguenti n -step return per $n = 1, 2, \dots, \infty$:

$$\begin{aligned} n = 1 & \quad (\text{TD}) \quad G_t^{(1)} = R_{t+1} + \gamma V(S_{t+1}) \\ n = 2 & \quad G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma V(S_{t+2}) \\ & \quad \dots \\ n = \infty & \quad (\text{MC}) \quad G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T \end{aligned}$$

Questo è possibile generalizzarlo a n -step return, arrivati a n si va a prendere la stima calcolata per quello stato:

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

La formula è simile a quella di sempre:

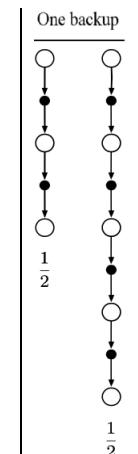
$$V(S_t) \leftarrow V(S_t) + \alpha (G_t^{(n)} - V(S_t))$$

MEDIA DEGLI n -step return:

L'idea è che si vuole stimare il valore del nodo presente al top, si va a prendere i guadagni che si ottengono con n -step per poi andare a fare delle medie.

Questa media è differente in base al numero di step che ci si ferma, possono poi essere combinate e ottenere la stima del nodo radice.

L'algoritmo che generalizza tutto ciò è chiamato **λ -return**.



λ -RETURN:

Questo algoritmo tiene in considerazione tutti i guadagni che si ottengono lungo il percorso.

Con questa tecnica si vanno a calcolare dei guadagni per ognuno dei percorsi, quello che fa il λ -return è prendere il guadagno ottenuto col 1-step e lo si somma con il successivo che ha un peso inferiore e così via (il peso diminuisce in base allo step in cui ci si ferma, dipende da n).

Il valore per T è fatto dalla somma pesata di tutti i guadagni costruiti in maniera incrementale.

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t$$

- $\lambda = 1$, caso Monte Carlo;
- $\lambda = 0$, caso one-step TD,

Esistono due versioni del $\text{TD}(\lambda)$:

1. **Forward View $\text{TD}(\lambda)$** , in realtà questa versione è solo teorica perché non è possibile usarlo nella pratica siccome guarda al futuro per calcolare G_t^λ proprio come MC, può essere calcolato solo da episodi completi;
2. **Backward View $\text{TD}(\lambda)$** , questa versione si utilizza nella pratica, aggiorna gli stati precedenti e non quelli in avanti.

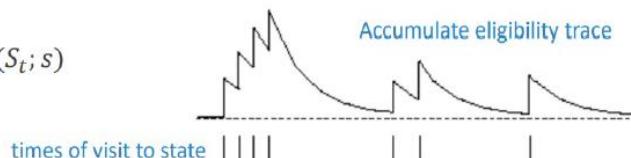
La Backward View tiene traccia anche di un altro valore chiamato **elegibility trace** per ogni stato s , che ci va a dire che il valore trovato nello stato da chi è stato influenzato, così che quando si va ad aggiornare $V(s)$ per ogni stato s , l'errore calcolato viene propagato a tutti gli stati precedenti e va ad aggiornare l'informazione aggiuntiva (l'elegibility trace) su ogni stato.

In altre parole, quando si aggiorna la value function si vanno a considerare gli stati che mi hanno portato allo stato che si sta analizzando. Di solito, i criteri che si utilizzano sono:

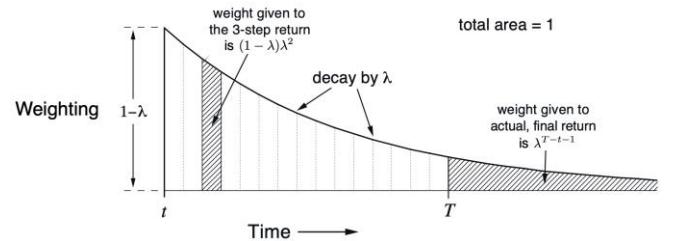
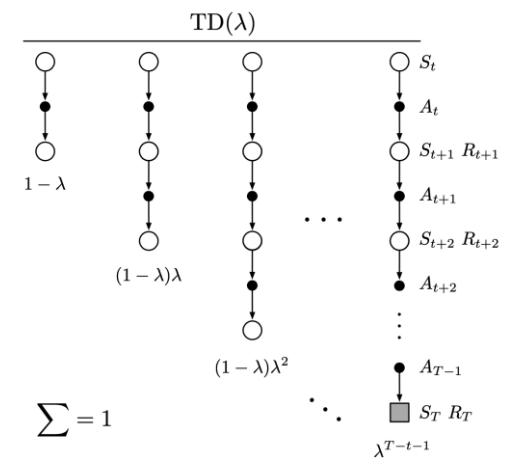
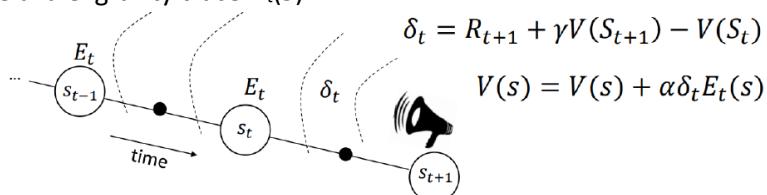
- **Frequency heuristic**: assegna credito agli stati più frequenti;
- **Recency heuristic**: assegna credito agli stati più recenti.

Le eligibility traces combinano entrambe le euristiche:

$$\begin{aligned} E_0(s) &= 0 \\ E_t(s) &= \gamma \lambda E_{t-1}(s) + \mathbf{1}(S_t; s) \end{aligned}$$



In proporzione al TD-error δ_t e alla eligibility trace $E_t(s)$:



20. MODEL FREE CONTROL

- **Model-free prediction**, serve a stimare la value function di un MDP non noto:

Il problema è prendere la policy senza conoscere il modello, non conoscendo le ricompense che possiamo avere e quali sono le transizioni, l'unica cosa da fare è applicare delle tecniche di campionamento. L'agente effettua le simulazioni e calcola quanto ha guadagnato per ogni campione.

- **Model-free control**, serve ad ottimizzare la value function di un MDP non noto:

Si va a carcare la policy ottimale, questo fa cambiare effettivamente il comportamento dell'agente, cioè si vuole che l'agente trovi la policy migliore per trovare la soluzione (es. labirinto si vuole il percorso più breve verso l'uscita).

NOTA: *In realtà non si ottimizza la value function, ma si va a calcolare l'action-value function.*

Per la maggior parte dei problemi di apprendimento per rinforzo, o:

- Il modello del MDP non è noto, ma l'esperienza può essere campionata;
- Il modello del MDP è noto, ma è troppo grande per essere utilizzato, se non per campionamento.

Il Model Free Control può risolvere questi problemi.

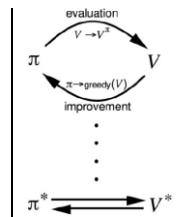
ON-POLICY E OFF-POLICY LEARNING:

Esistono due tipi di algoritmi di apprendimento per la fase di Control, si dividono in:

- **On-policy learning:** la policy viene valutata e aggiornata su sé stessa (miglioriamo la policy analizzata). Questo sarebbe apprendere sul campo, cioè la si mette in pratica e si capisce dove andare a correggerla. Apprendere la policy π dall'esperienza campionata da π .
- **Off-policy learning:** la policy che si va a mettere in pratica non è quella che si vuole ottimizzare. Vediamo qualcuno fare qualcosa e cerchiamo di apprendere osservando. Pertanto, si hanno due policy, quella che viene messa in pratica e una che si vuole ottimizzare. Apprendere la policy π dall'esperienza campionata da μ .

POLICY ITERATION GENERALIZZATA (per On-policy):

L'approccio che utilizza **On-policy** è la **policy iteration generalizzata** che consiste nell'alternare la fase di valutazione con una fase di ottimizzazione (prediction e control) finché non converge all'ottimo. Questa versione generalizzata prevede che si possa usare un qualsiasi algoritmo per fare valutazione della policy e poi andare a fare il miglioramento.



POLICY ITERATION GENERALIZZATA CON ON-POLICY MC:

Una prima tecnica che si può utilizzare è usare il model-free e la valutazione la si fa con Monte-Carlo. Il problema di questa soluzione, però, è che se si vuole migliorare la policy usando un algoritmo di stima della value function bisognerebbe applicare la seguente formula:

Il miglioramento greedy della policy *rispetto a $V(s)$ necessita* del modello di un MDP:

$$\pi'(s) = \arg \max_{a \in \mathcal{A}} \mathcal{R}_s^a + P_{ss'}^a V(s')$$

(per aggiornare π' bisogna calcolare l'azione che porta il valore massimo che è dato dalla ricompensa dallo stato s eseguendo $a + la probabilità per i valori dei successivi$)

Ma il problema è che la formula non si può applicare perché R e P non sono noti.

Gli algoritmi visti in precedenza non vanno bene siccome stimano V , ma il problema può essere affrontato definendo la policy andando ad utilizzare la action-value function, anziché prendere V si prende Q . La policy sceglie nello stato s l'azione che massimizza la ricompensa da s . In pratica, stando in s si va a vedere tutti gli archi uscenti e tramite la funzione Q sappiamo il guadagno per ogni azione. La action-value function dice per ogni stato/azione quant'è la ricompensa.

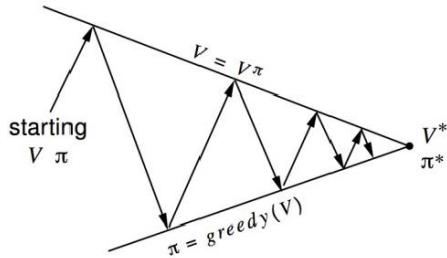
Il miglioramento greedy della policy *rispetto a $Q(s,a)$ è model-free*, essa porta ad un guadagno migliore tra tutte le azioni:

$$\pi'(s) = \arg \max_{a \in \mathcal{A}} Q(s, a)$$

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$
Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
$Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0 , following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \dots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \dots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg \max_a Q(S_t, a)$

POLICY ITERATION GENERALIZZATA

CON ON-POLICY MC:



- Valutazione della Policy – Monte-Carlo policy evaluation, $V=v_\pi$?
- Miglioramento della Policy – Genera un miglioramento greedy della policy?

Quindi l'algoritmo non fa altro che generare campioni, non ci calcoliamo la value function ma l'action-value function e si va a definire una nuova policy.

In realtà, questo algoritmo ha un problema che gli impedisce di convergere. Questo algoritmo è basato sul campionamento, quindi sceglie delle azioni e utilizza sempre la soluzione greedy.

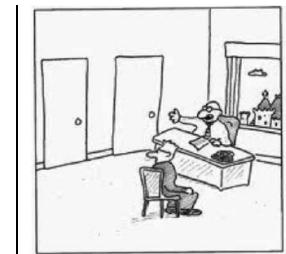
Esempio di selezione di azioni greedy:

Ci sono due porte:

- Apri la porta di sinistra e ottieni ricompensa 0 – $V(\text{sinistra}) = 0$;
- Apri la porta di destra e ottieni ricompensa +1 – $V(\text{destra}) = +1$;
- Apri la porta di destra e ottieni ricompensa +3 – $V(\text{destra}) = +2$;
- Apri la porta di destra e ottieni ricompensa +2 – $V(\text{destra}) = +2$.

Sei sicuro di aver scelto la porta migliore?

L'algoritmo appena visto, quello che fa è che una volta che ha scelto l'azione migliore non la cambierà. Può essere che a sinistra (se si sceglie destra) potremmo guadagnare di più in futuro.



ϵ -GREEDY EXPLORATION:

L'idea è di scegliere quasi sempre (e non sempre, conseguenza della tecnica greedy pura) la migliore azione.

Tutte le m azioni vengono sperimentate con una **probabilità non nulla** (ϵ è un valore alto):

- Con probabilità $1-\epsilon$ viene selezionata l'azione greedy;
- Con probabilità ϵ viene selezionata un'azione random.

$$\pi(a|s) = \begin{cases} \epsilon/m + (1-\epsilon) & \text{if } a^* = \arg \max_{a \in \mathcal{A}} Q(s, a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

On-policy first-visit MC control (for ϵ -soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\epsilon > 0$

Initialize:

$\pi \leftarrow$ an arbitrary ϵ -soft policy
 $Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
 $Returns(s, a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):

Generate an episode following π : $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \dots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$
 $Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
 $A^* \leftarrow \arg \max_a Q(S_t, a)$ (with ties broken arbitrarily)
For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \epsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

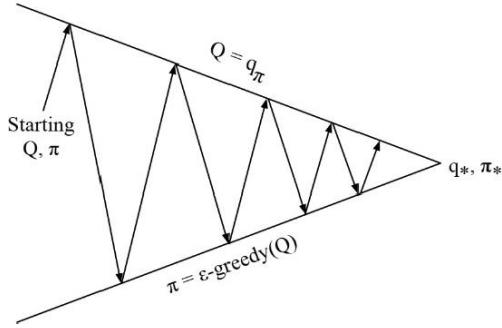
TEOREMA:

Per ogni policy ϵ -greedy π , la policy ϵ -greedy π' rispetto a q_π è un miglioramento, $v_{\pi'}(s) \geq v_\pi(s)$.

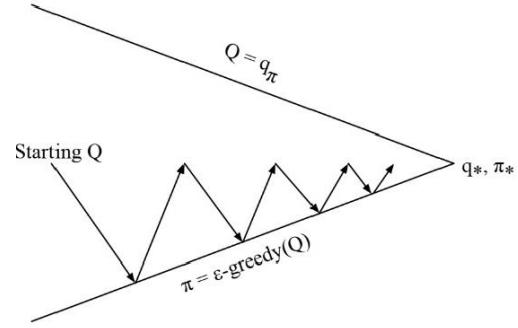
Dal teorema si ha che la policy π' è migliore della policy precedente.

Pertanto, dal **policy improvement theorem** $v_{\pi'}(s) \geq v_\pi(s)$.

MONTE-CARLO POLICY ITERATION:



MONTE-CARLO CONTROL:



L'algoritmo greedy definito va a valutare la action-value function mentre il miglioramento lo si fa con ϵ -greedy.

L'algoritmo usato nella pratica non va a fare la valutazione completa della action-value function. Una volta data la policy per fare il passo verso la Q bisogna generare tanti campioni per coprire tutti gli stati, altrimenti la funzione Q non è definita completamente. Piuttosto che andare a fare la valutazione completa della Q, si genera un episodio (un sottoinsieme di stati) e si va a fare la valutazione della Q solo degli stati di quell'episodio, pertanto non si ha più $Q=q_\pi$ ma $Q \sim q_\pi$ (Q approssimato a q_π).

GREEDY IN THE LIMIT WITH INFINITE EXPLORATION (GLIE):

Questo appena descritto è l'algoritmo di MC che effettua la fase di Control e si dimostra che al tendere del numero di simulazioni (coppie stato-azione) all'infinito l'algoritmo converge ad una policy greedy.

Tutte le coppie stato-azione vengono esplorate **un numero infinito** di volte:

$$\lim_{k \rightarrow \infty} N_k(s, a) = \infty$$

La policy converge ad una policy greedy:

$$\lim_{k \rightarrow \infty} \pi_k(a|s) = \mathbf{1}\left(a; \arg \max_{a' \in \mathcal{A}} Q_k(s, a')\right)$$

ϵ -greedy è GLIE se ϵ si azzerà a $\epsilon_k=1/k$.

La condizione è che si deve definire è il valore ϵ che deve essere un valore che tende a 0 (più iterazioni si fanno più ϵ diminuisce).

GLIE MONTE-CARLO CONTROL:

Definiamo un campione usando una certa policy π , a questo punto andiamo ad aggiornare il valore dell'action-value function, una volta fatto ciò si effettua il miglioramento della policy, aggiornando ϵ in modo che dipende da k (facendo $1/k$) e si applica la formula precedente. Facendo in questo modo l'algoritmo converge dando la policy ottimale.

Campionare il k -esimo episodio utilizzando:

$$\pi: \{S_1, A_1, R_2, \dots, S_T\} \sim \pi$$

Per ogni stato S_t e azione A_t nell'episodio:

$$N(S_t, A_t) \leftarrow N(S_t, A_t) + 1$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{N(S_t, A_t)} (G_t - Q(S_t, A_t))$$

Migliorare la policy in base alla nuova action-value function:

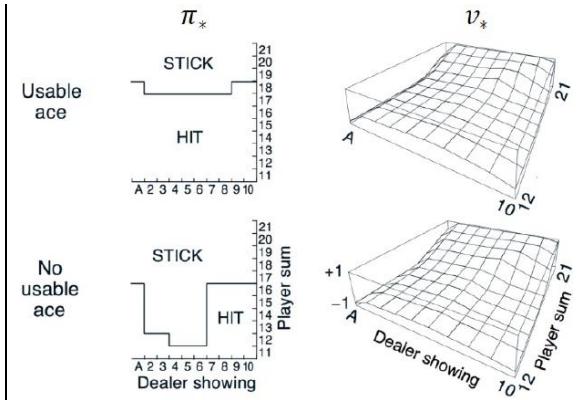
$$\epsilon \leftarrow \frac{1}{k}$$

$$\pi \leftarrow \epsilon - \text{greedy}(Q)$$

Esempio blackjack:

Stick il giocatore non vuole più carte mentre Hit vuole la carta.

Dal grafico, quando il giocatore ha un asso è meglio chiedere una carta.



ON-POLICY TD CONTROL:

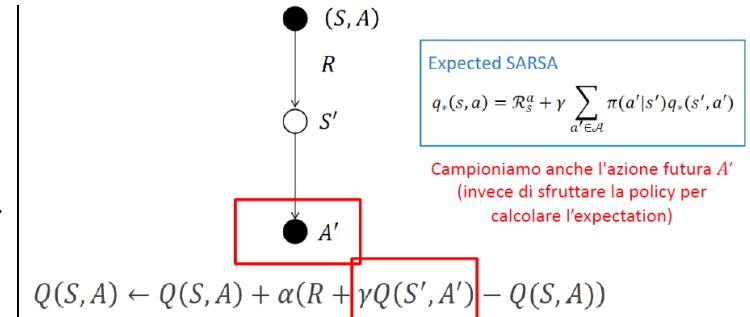
Possiamo definire l'algoritmo di control anche usando il Temporal-Difference, essendo un algoritmo policy vuol dire che aggiorna la policy che si sta valutando. Il TD ha dei vantaggi rispetto a MC, come:

- Varianza inferiore, quando si aggiorna lo si fa rispetto ad un passo e non rispetto a tutta la sequenza di simulazioni;
- È online, gli aggiornamenti si fanno direttamente sugli stati senza aspettare la fine;
- Gestisce sequenze incomplete, non avendo bisogno dello stato terminale.

AGGIORNAMENTO DELLE ACTION-VALUE FUNCTION CON SARSA:

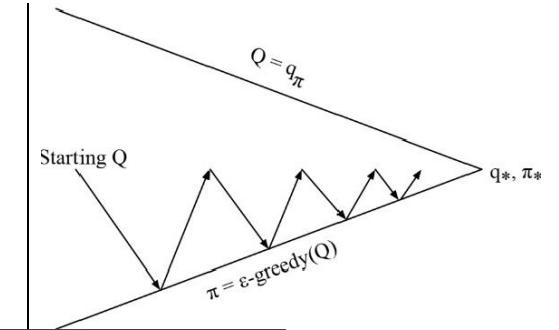
Si aggiorna la formula vista di MC con la formula di TD.

La formula di aggiornamento della action-value, il nuovo valore sarà = al valore precedente + α * (la ricompensa + quanto guadagno dallo stato S' che raggiungo eseguendo l'azione A' , andando a vedere un passo avanti prendendo l'azione che posso eseguire nello stato S') – il valore attuale.



ON-POLICY CONTROL CON SARSA:

Il backup diagram è lo stesso del precedente, la differenza è che si parte da azioni, pertanto si ha la sequenza di azioni-stato-azioni-..., mentre il precedente è stato-azioni-stato ... perché era sulla funzione V, qui invece stiamo sulla funzione Q.



Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
 Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$
 Loop for each episode:
 Initialize S
 Choose A from S using policy derived from Q (e.g., ε -greedy)
 Loop for each step of episode:
 Take action A , observe R, S'
 Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$$

$$S \leftarrow S'; A \leftarrow A';$$

 until S is terminal

TEOREMA:

L'algoritmo SARSA converge verso la action-value function ottimale ($Q(s, a) \rightarrow q_*(s, a)$) a condizione che:

- Sequenza di policy $\pi_t(a|s)$ GLIE
- Sequenza di Robbins-Monro di step-size α_t

$$\begin{aligned} \sum_{t=1}^{\infty} \alpha_t &= \infty \\ \sum_{t=1}^{\infty} \alpha_t^2 &< \infty \end{aligned}$$

SARSA(λ) n-STEP:

L'algoritmo λ è la versione in cui andiamo a considerare 1 solo step. Possiamo fare come il TD, ovvero andare a generalizzare ad n-step. Consideriamo i seguenti n-step return, per $n = 1, 2, \dots, \infty$:

$$\begin{aligned} n = 1 \quad (\text{SARSA}) \quad q_t^{(1)} &= R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) \\ n = 2 \quad q_t^{(2)} &= R_{t+1} + \gamma R_{t+2} + \gamma Q(S_{t+2}) \\ &\dots \\ n = \infty \quad (\text{MC}) \quad q_t^{(\infty)} &= R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T \end{aligned}$$

Definiamo il **n-step Q-return**:

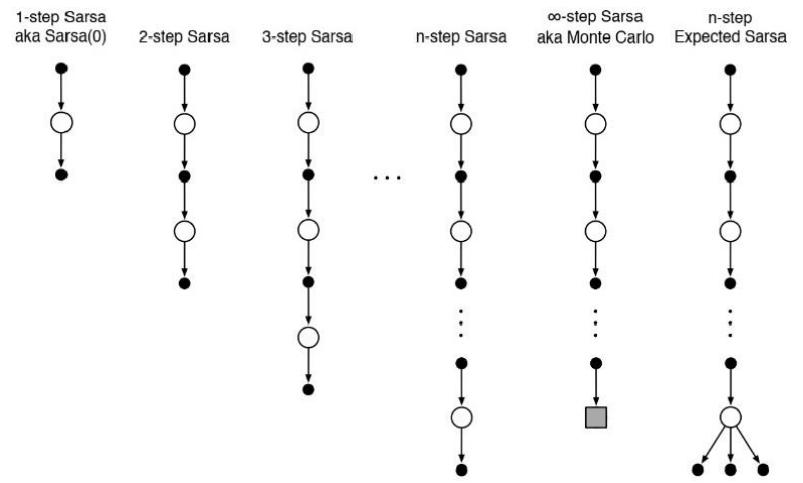
$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, A_{t+n})$$

(definito come somma delle ricompense sempre moltiplicato per un fattore di sconto fino ad arrivare allo stato n e consideriamo la ricompensa ottenuta in S_{t+n} eseguendo l'azione A_{t+n})

n-step SARSA aggiorna $Q(S, A)$ verso il n-step Q-return:

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(q_t^{(n)} - Q(S, A) \right)$$

Andando ad analizzare il diagramma di backup vediamo che è lo stesso di TD(λ) a differenza che qui abbiamo delle action-value, quindi abbiamo azione-stato-azione.



SARSA(λ) – FORWARD VIEW:

Il q^λ return combina tutti gli n -step Q -return q_t^λ .

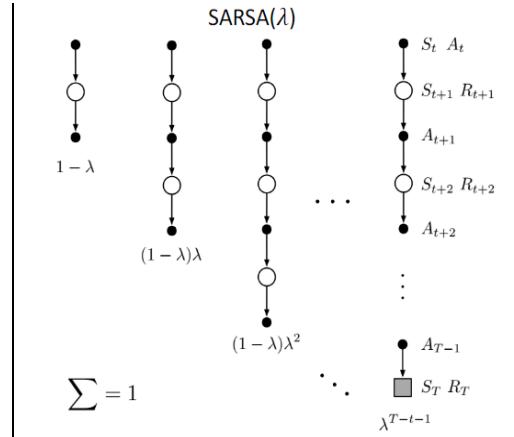
Utilizzando i pesi $(1 - \lambda)$ λ^{n-1} :

$$q_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$$

Forward SARSA update:

$$Q(S, A) \leftarrow Q(S, A) + \alpha(q_t^\lambda - Q(S, A))$$

Ognuno degli n -step li consideriamo per il calcolo della Q dandogli un peso sempre decrescente. L'aggiornamento non dipende solo dalla somma delle ricompense del valore dell'ultimo nodo ma è una media tra tutti i valori che otteniamo da ogni diagramma.



SARSA(λ) – BACKWARD VIEW:

SARSA(λ) necessita di una **eligibility trace per ogni coppia stato-azione**:

$$\begin{aligned} E_0(s, a) &= 0 \\ E_t(s, a) &= \gamma \lambda E_{t-1}(s, a) + \mathbf{1}(S_t, A_t; s, a) \end{aligned}$$

$Q(s, a)$ viene aggiornato per ogni stato s e azione a in proporzione al TD-error δ_t e alla eligibility trace $E_t(s, a)$:

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha \delta_t E_t(s, a) \end{aligned}$$

ALGORITMO SARSA(λ):

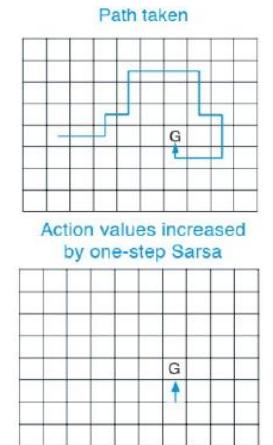
```

Initialize  $Q(s, a)$  arbitrarily, for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
Repeat (for each episode):
   $E(s, a) = 0$ , for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ 
  Initialize  $S, A$ 
  Repeat (for each step of episode):
    Take action  $A$ , observe  $R, S'$ 
    Choose  $A'$  from  $S'$  using policy derived from  $Q$  (e.g.,  $\varepsilon$ -greedy)
     $\delta \leftarrow R + \gamma Q(S', A') - Q(S, A)$ 
     $E(S, A) \leftarrow E(S, A) + 1$ 
    For all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$ :
       $Q(s, a) \leftarrow Q(s, a) + \alpha \delta E(s, a)$ 
       $E(s, a) \leftarrow \gamma \lambda E(s, a)$ 
     $S \leftarrow S'; A \leftarrow A'$ 
  until  $S$  is terminal

```

Esempio Gridworld:

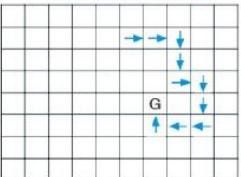
supponiamo il campione a destra (un episodio), parte da uno stato e arriva ad uno stato obiettivo.



Eseguendo l'algoritmo SARSA, calcola la funzione action-value Q , facendo un solo passo, andrà ad aggiornare solo l'ultimo passo (1-step).

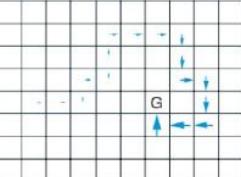
Se invece facciamo 10-step, prende i valori di Q per tutte le celle segnate.

Action values increased by 10-step Sarsa



Se andiamo ad usare la versione SARSA(λ) con $\lambda=0.9$, si ottiene la seguente tabella, dove man mano che è più lontano dalla fine dell'episodio, decresce il valore.

Action values increased by Sarsa(λ) with $\lambda=0.9$



OFF-POLICY TD LEARNING:

Fino ad ora abbiamo visto On-policy, dove prendevamo delle Q e le aggiornavamo con Q stessa. Ora andiamo a prendere la policy target andando ad eseguire un'altra policy. L'obiettivo è sempre andare a definire la policy target π , calcolandoci la Q, come fatto in precedenza. La cosa che cambia è che il comportamento, i campioni che produciamo non li produciamo con la policy che abbiamo ma con un'altra policy μ .

Quindi, valutare la policy target $\pi(a|s)$ per calcolare $v_\pi(s)$ o $q_\pi(s,a)$.

Seguendo la policy comportamentale $\mu(a|s)$:

$$\{S_1, A_1, R_2, \dots, S_T\} \sim \mu$$

IMPORTANCE SAMPLING:

Il problema da affrontare è come fare, data la policy μ , a capire come deve comportarsi π . La prima tecnica è l'importance sampling, dove determina dei pesi che vanno a far il mapping da una distribuzione all'altra.

Vogliamo determinare la distribuzione di valori di X secondo una distribuzione P, per calcolare questa distribuzione possiamo fare la sommatoria per la probabilità di ogni possibile valore di X per F(X). Quindi P è quella che vogliamo determinare, ma conosciamo Q, pertanto moltiplichiamo e dividiamo per Q:

$$\begin{aligned} \mathbb{E}_{X \sim P}[f(X)] &= \sum P(X)f(X) \\ &= \sum Q(X) \frac{P(X)}{Q(X)} f(X) \\ &= \mathbb{E}_{X \sim Q} \left[\frac{P(X)}{Q(X)} f(X) \right] \end{aligned}$$

Estrarre campioni dalla importance distribution $Q(X)$ piuttosto che da $P(X)$

Assegnare pesi tali che l'expectation empirica (su campioni di $Q(X)$) corrisponda all'expectation sotto $P(X)$

IMPORTANCE SAMPLING PER OFF-POLICY MONTE-CARLO:

Quello che si fa nella pratica si cambia il calcolo della ricompensa, dove non è più ottenuto con la policy μ ma bisogna fare i rapporti tra i guadagni in ogni punto del campione.

- Utilizzare i guadagni generati da μ per valutare π ;
- Pesare il guadagno G_t in base alla somiglianza tra le policy;
- Moltiplicare le correzioni dell'importance sampling lungo l'intero episodio:

$$G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \dots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)} G_t$$

- Aggiornare il valore verso il guadagno corretto:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^{\pi/\mu} - V(S_t))$$

L'importance sampling può **aumentare drasticamente la varianza**, è preferibile usare la TD dove viene fatto un solo rapporto riducendo appunto la varianza.

IMPORTANCE SAMPLING PER OFF-POLICY TD:

Viene impiegata la stessa formula precedente, dove il rapporto è fatto solo su una sola azione a partire dallo stato S_t .

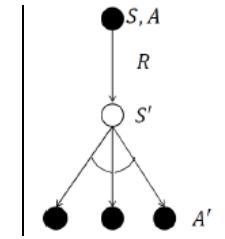
- Utilizzare i **target TD generati da μ per valutare π** ;
- Pesare i target TD $R+\gamma V(S')$ tramite l'importance sampling;
- È necessaria una **singola correzione dell'importance sampling**:

$$V(S_t) \leftarrow V(S_t) + \alpha \left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \right)$$

- Varianza molto più bassa rispetto al MC;
- Le policy devono essere **simili solo per un singolo step**.

Q-LEARNING:

Nella pratica le due versioni precedenti non vengono usate, quello più usato è il Q-learning che è l'algoritmo che va a determinare direttamente l'action-value q_* usando una funzione Q, quindi il valore di importance sampling non serve. Praticamente si va ad aggiornare i valori dell'action A nello stato S, andando a prendere la reward seguendo l'azione, poi dallo stato successivo si prende il massimo delle ricompense tra tutte le possibili azioni.



- Off-policy learning di action-value $Q(s,a)$;
- L'importance sampling non è necessario;
- L'azione successiva viene scelta utilizzando la policy comportamentale $A_{t+1} \sim \mu(\cdot | S_t)$;
- Ma prendiamo in considerazione azioni alternative $A' \sim \pi(\cdot | S_t)$;
- E aggiorniamo $Q(S_t, A_t)$ in base al valore dell'azione alternativa:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t))$$

- Consente di migliorare sia il comportamento che le policy target;
- La policy target π è **greedy** rispetto a $Q(S_t, A_t)$:

$$\pi(S_{t+1}) = \arg \max_{a'} Q(S_{t+1}, a')$$

- La policy comportamentale μ è ϵ -greedy rispetto a $Q(s,a)$;
- L'obiettivo del Q-learning si riduce a:

$$\begin{aligned} R_{t+1} + \gamma Q(S_{t+1}, A') &= R_{t+1} + \gamma Q\left(S_{t+1}, \arg \max_{a'} Q(S_{t+1}, a')\right) \\ &= R_{t+1} + \max_{a'} \gamma Q(S_{t+1}, a') \end{aligned}$$

TEOREMA:

Il Q-learning control converge alla action-value ottimale, $Q(s,a) \rightarrow q_*(s,a)$.

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \max_{a'} \gamma Q(S', a') - Q(S, A))$$

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:

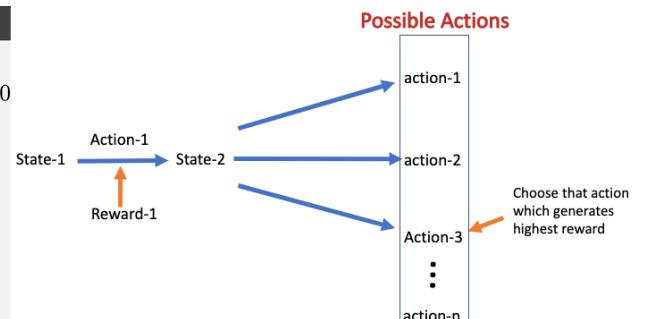
 Choose A from S using policy derived from Q (e.g., ϵ -greedy)

 Take action A , observe R, S'

$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$

 until S is terminal



RIASSUMENDO:

	Full Backup (DP)	Sample Backup (TD)
Bellman Expectation Equation for $v_\pi(s)$	$v_\pi(s) \leftarrow s$ $v_\pi(s') \leftarrow s, a, r$ 	$v_\pi(s) \leftarrow s$
Bellman Expectation Equation for $q_\pi(s, a)$	$q_\pi(s, a) \leftarrow s, a, r$ $q_\pi(s', a') \leftarrow s', a', r$ 	$q_\pi(s, a) \leftarrow s, a$
Bellman Optimality Equation for $q_*(s, a)$	$q_*(s, a) \leftarrow s, a, r$ $q_*(s', a') \leftarrow s', a', r$ 	$q_*(s, a) \leftarrow s, a$

- Model-Free control sfrutta la action-value function: il miglioramento della politica non necessita di un MDP e Generalized policy iteration;
- È necessario mantenere una esplorazione sufficiente (ϵ -greedy);
- Off-policy control: apprendimento della value-function di una policy target a partire dai dati generati da una diversa policy comportamentale, Importance sampling per far coincidere le expectation di due policy;
- TD control: On-policy: SARSA(λ) - Off-policy: Q-learning

21. VALUE FUNCTION APPROXIMATION

Adesso bisogna capire come effettuare l'apprendimento per rinforzo su problemi reali (on the Scale), ovvero con uno spazio degli stati non banale (abbastanza alto), come ad esempio il Backgammon con 1020 stati oppure con un Robot con spazio degli stati continuo. Per questi problemi l'apprendimento per rinforzo non si può utilizzare, non tanto per la memoria ma per la difficoltà computazionale per andare a calcolare i valori di tutti possibili stati (*numero stati elevato*). La soluzione è non considerare delle funzioni esatte ma delle **funzioni approssimate**, siccome fino ad ora abbiamo usato delle funzioni dove per ogni stato abbiamo associato un valore definendo una value function per ogni stato e la action-value function che per ogni coppia stato-azione avevamo un valore (basato su tabella).

- Finora $V(s)/Q(s, a) = \text{lookup table}$ (Value function/Action-Value function)

Una entry per ogni stato s o coppia stato-azione s, a .

- Adesso stimiamo la value function con la **approximation function**:

$$\begin{aligned}\hat{v}(s; \mathbf{w}) &\approx v_\pi(s) \\ \hat{q}(s, a; \mathbf{w}) &\approx q_\pi(s, a)\end{aligned}$$

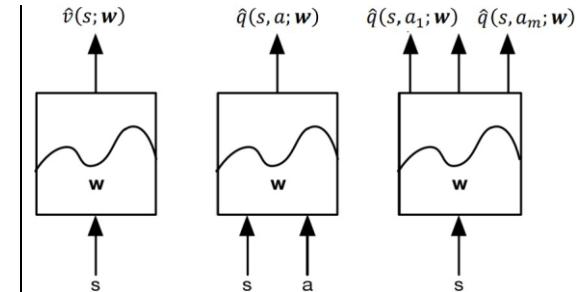
Queste funzioni hanno un parametro \mathbf{w} che sarebbero i pesi che possono essere cambiati per far cambiare il funzionamento della funzione. Pertanto, definiamo una funzione approssimata che dovrebbe essere quanto più simile al comportamento della funzione reale.

Quello che si vuole far vedere è usare algoritmi di Machine Learning per apprendere funzioni approssimate.

Algoritmi che, in base a dei pesi che si vanno ad apprendere, da uno stato dicono qual è la loro funzione andando ad approssimare la value function.

Altri algoritmi lo fanno per l'action-value function, dato uno stato e un'azione danno una approssimazione (sempre in funzione dei pesi esplicitati).

Oppure altri algoritmi che da uno stato danno l'action-value function per tutte le possibili azioni (in base ai pesi dati).



Questa operazione può essere fatta in vari modi, come ad esempio utilizzare una **combinazione lineare di features, reti neurali**, alberi di decisione, Nearest neighbour, Fourier, ecc...

Questi metodi per funzionare bene devono essere in grado di lavorare con **dati non stazionari**, siccome l'apprendimento viene fatto durante l'interazione (a differenza del Machine Learning che viene dato tutto il dataset all'inizio).

21.1 METODI INCREMENTALI

Come visto in precedenza, dobbiamo cercare di approssimare una funzione, ovvero dato un training set e dato $f()$ trovare $h()$ tale che $h()$ sia quanto più simile ad $f()$.

Tutto ciò lo si risolve col metodo del gradiente, andando a calcolare una loss function (errore quadratico) tra la value function e la funzione approssimata. Definiamo questo errore come una media degli errori, facendo la differenza tra il valore che vogliamo predire e quello predetto dalla funzione approssimata.

Da notare che abbiamo una **funzione μ** , andiamo a considerare una funzione che ci dice quanto uno stato è più importante rispetto ad un altro. Con la funzione di approssimazione consideriamo un sottoinsieme di stati, quindi, avremo per ogni stato della nostra funzione, n stati del problema reale.

Formalmente, vogliamo trovare il **vettore dei parametri \mathbf{w} minimizzando l'errore quadrato medio** tra l'approximate value $\hat{v}(s; \mathbf{w})$ e il true value function $v_\pi(s)$:

$$\overline{\text{VE}}(\mathbf{w}) \doteq \sum_{s \in \delta} \mu(s) \left[v_\pi(s) - \hat{v}(s; \mathbf{w}) \right]^2 \quad \mu(s) \geq 0, \sum_s \mu(s) = 1$$

Quali stati sono più rilevanti

Per minimizzare questo errore usiamo il **metodo del gradiente**, calcolando il gradiente e andare nella direzione opposta:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2} \alpha \nabla \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right]^2 \\ &= \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

Possiamo scrivere tutto ciò nel modo seguente:

$$\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \dots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^\top$$

Quello che vogliamo è cambiare i pesi della funzione per far sì che l'errore si minimizzi.

Quindi con l'uso del gradiente abbiamo un algoritmo iterativo incrementale che aggiorna i pesi in maniera tale da minimizzare l'errore.

FEATURES VECTOR STATE:

Per definire la funzione approssimata in funzione di un vettore dei pesi, rappresentiamo ogni stato come **vettore di features**:

$$x(S) = \begin{bmatrix} x_1(S) \\ \vdots \\ x_n(S) \end{bmatrix}$$

LINEAR VALUE FUNCTION APPROXIMATION:

Un modo per definire la funzione di approssimazione è usare una **funzione lineare**, andando a rappresentare la funzione come una combinazione del prodotto tra il vettore dei pesi e il vettore dei features:

$$x(S) = \begin{bmatrix} 1(S; s_1) \\ \vdots \\ 1(S; s_n) \end{bmatrix} \quad \hat{v}(S; w) = \begin{bmatrix} 1(S; s_1) \\ \vdots \\ 1(S; s_n) \end{bmatrix}^T \begin{bmatrix} w_1 \\ \vdots \\ w_n \end{bmatrix}$$

$$\hat{v}(s, w) \doteq w^\top x(s) \doteq \sum_{i=1}^d w_i x_i(s)$$

(la funzione approssimata è il prodotto tra la trasposta dei pesi e il vettore dei features, ma questa è la sommatoria del prodotto feature e peso dello stato)

Se si usa questo tipo di approssimazione, dalla formula precedente si ottiene che la derivata di \hat{v} corrisponde a x , sarebbe la derivata rispetto ai pesi e pertanto rimane solo il vettore x :

$$\nabla \hat{v}(s, w) = x(s) \quad \rightarrow \quad w_{t+1} \doteq w_t + \alpha [U_t - \hat{v}(S_t, w_t)] x(S_t)$$

PREDICTION INCREMENTALE:

$$w_{t+1} = w_t + \alpha [v_\pi(S_t) - \hat{v}(S_t, w_t)] \nabla \hat{v}(S_t, w_t)$$

La formula vista all'inizio (appena riportata) la possiamo utilizzare se usiamo l'**apprendimento supervisionato**, siccome abbiamo il campione con la corrispondente etichetta, ma nell'apprendimento per rinforzo non abbiamo l'etichetta e la funzione v_π la dobbiamo apprendere, pertanto ci serve una stima. In pratica, sostituiamo $v_\pi(s)$ con un target:

- MC – Il target è il guadagno G_t : $\Delta w = \alpha (G_t - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w)$
- TD(0) – Il target è il TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$: $\Delta w = \alpha (R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w)$
- TD(λ) – Il target è λ -return G_t^λ : $\Delta w = \alpha (G_t^\lambda - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w)$

VALUE FUNCTION APPROXIMATION (MONTE CARLO):

Usando MC sostituiamo $v_\pi(s)$ con il guadagno G_t che è un campione non distorto e rumoroso del true value $v_\pi(S_t)$.

È **non distorto** siccome non abbiamo bias, perché il valore di t è calcolato sugli episodi ed essendo questi reali non c'è bias. È **rumoroso** perché c'è molta varianza.

Con MC si può quindi applicare l'apprendimento **supervisionato** sui campioni di addestramento e tutto ciò viene usato per aggiornare i pesi e minimizzare l'errore, definendo \hat{v} .

$$\langle S_1, G_1 \rangle, \langle S_2, G_2 \rangle, \dots, \langle S_T, G_T \rangle$$

Linear Monte-Carlo policy evaluation:

$$\Delta w = \alpha (G_t - \hat{v}(S_t, w)) \nabla_w \hat{v}(S_t, w) = \alpha (G_t - \hat{v}(S_t, w)) x(S_t)$$

La Monte-Carlo evaluation converge verso un ottimo locale.

L'algoritmo genera gli episodi e una volta ottenuti gli episodi si aggiornano i pesi.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$
Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $w \in \mathbb{R}^d$ arbitrarily (e.g., $w = \mathbf{0}$)
Loop forever (for each episode):
Generate an episode $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$ using π
Loop for each step of episode, $t = 0, 1, \dots, T-1$:
$w \leftarrow w + \alpha [G_t - \hat{v}(S_t, w)] \nabla \hat{v}(S_t, w)$

VALUE FUNCTION APPROXIMATION (TEMPORAL-DIFFERENCE 0):

Con TD(0), i valori stimati non sono campioni realistici, siccome si basano sui valori dei successori (Bootstrapping targets).

Usando TD(0) sostituiamo $v_\pi(s)$ con il TD target $R_{t+1} + \gamma \hat{v}(S_{t+1}, w)$ che è un campione **distorto** del true value $v_\pi(S_t)$.

Si può ancora applicare l'apprendimento **supervisionato** sui campioni di addestramento, ma non producono un vero metodo gradient-descent (sono chiamati metodi **semi-gradient**), siccome non di basano su valori reali ma su delle approssimazioni.

$$\langle S_1, R_2 + \gamma \hat{v}(S_2, w) \rangle, \dots, \langle S_{T-1}, R_T \rangle$$

Linear TD(0) policy evaluation:

$$\Delta \mathbf{w} = \alpha(R + \gamma \hat{v}(S'; \mathbf{w}) - \hat{v}(S; \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S; \mathbf{w}) = \alpha \delta \mathbf{x}(S)$$

Linear TD(0) converge (vicino) all'ottimo globale.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_{\pi}$

Input: the policy π to be evaluated

Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$ such that $\hat{v}(\text{terminal}, \cdot) = 0$

Algorithm parameter: step size $\alpha > 0$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:

 Initialize S

 Loop for each step of episode:

 Choose $A \sim \pi(\cdot | S)$

 Take action A , observe R, S'

$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})] \nabla \hat{v}(S, \mathbf{w})$

$S \leftarrow S'$

 until S is terminal

VALUE FUNCTION APPROXIMATION (TEMPORAL-DIFFERENCE λ):

Stessa situazione con TD(λ), anche esso è un metodo **semi-gradiante**.

Usando TD(0) sostituiamo $v_{\pi}(s)$ con λ -return G_t^{λ} che è un campione distorto del true value $v_{\pi}(S_t)$.

Si può ancora applicare l'apprendimento **supervisionato** sui campioni di addestramento.

$$\langle S_1, G_1^{\lambda} \rangle, \dots, \langle S_{T-1}, G_{T-1}^{\lambda} \rangle$$

▪ **Forward** view linear TD(λ):

$$\Delta \mathbf{w} = \alpha (G_t^{\lambda} - \hat{v}(S_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{v}(S_t; \mathbf{w}) = \alpha (G_t^{\lambda} - \hat{v}(S_t; \mathbf{w})) \mathbf{x}(S_t)$$

▪ **Backward** view linear TD(λ):

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}; \mathbf{w}) - \hat{v}(S_t; \mathbf{w}) \\ E_t &= \lambda \gamma E_{t-1} + \mathbf{x}(S_t) \\ \Delta \mathbf{w} &= \alpha \delta_t E_t \end{aligned}$$

CONTROL INCREMENTALE:

Con la stessa idea vista con la value function, si può riproporre con gli algoritmi di controllo, facendo controllo in maniera incrementale, usando gli algoritmi di controllo per apprendere una funzione che, in questo caso, non è la value function ma è la action-value function. Pertanto, vogliamo definire la funzione \hat{q} che dovrebbe essere quanto più simile a q .

Formalmente, vogliamo approssimare la action-value function $\hat{q}(S, A; \mathbf{w}) \approx q_{\pi}(S, A)$.

Vogliamo sempre effettuare la minimizzazione dell'errore quadratico medio tra il approximate action-value $\hat{q}(S, A; \mathbf{w})$ e il true action-value $q_{\pi}(S, A)$:

$$J(\mathbf{w}) = \mathbb{E}_{\pi} [(q_{\pi}(S, A) - \hat{q}(S, A; \mathbf{w}))^2]$$

Utilizzare **stochastic gradient descent** per trovare un minimo locale:

$$\begin{aligned} -\frac{1}{2} \nabla_{\mathbf{w}} J(\mathbf{w}) &= (q_{\pi}(S, A) - \hat{q}(S, A; \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A; \mathbf{w}) \\ \Delta \mathbf{w} &= \alpha (q_{\pi}(S, A) - \hat{q}(S, A; \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S, A; \mathbf{w}) \\ \mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha [U_t - \hat{q}(S_t, A_t, \mathbf{w}_t)] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t) \end{aligned}$$

LINEAR ACTION-VALUE FUNCTION APPROXIMATION:

Anche qui rappresentiamo il vettore x come vettore di features:

$$\mathbf{x}(S, A) = \begin{bmatrix} x_1(S, A) \\ \vdots \\ x_n(S, A) \end{bmatrix}$$

Una possibile funzione da utilizzare è la funzione lineare per andare a fare l'approssimazione, avendo sempre il prodotto di un vettore trasposto per il vettore dei pesi:

$$\hat{q}(S, A; \mathbf{w}) = \mathbf{x}(S, A)^T \mathbf{w}$$

Infine, usiamo sempre la formula del gradiente (stochastic gradient descent update) per aggiornare i pesi:

$$\begin{aligned} \nabla_{\mathbf{w}} \hat{q}(S, A; \mathbf{w}) &= \mathbf{x}(S, A) \\ \Delta \mathbf{w} &= \alpha (q_{\pi}(S, A) - \hat{q}(S, A; \mathbf{w})) \mathbf{x}(S, A) \end{aligned}$$

CONTROL INCREMENTALE:

Per approssimare il valore di q reale, non avendo l'oracolo siccome la funzione q la dobbiamo stimare durante l'apprendimento e possiamo usare uno degli algoritmi già visti:

- MC – Il target è il guadagno G_t :

$$\Delta \mathbf{w} = \alpha (\tilde{G}_t - \hat{q}(S_t, A_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t; \mathbf{w})$$

- TD(0) – Il target è il TD target $R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}; \mathbf{w})$:

$$\Delta \mathbf{w} = \alpha (R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}; \mathbf{w}) - \hat{q}(S_t, A_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t; \mathbf{w})$$

- Forward TD(λ) – Il target è l'action-value λ -return:

$$\Delta \mathbf{w} = \alpha (q_t^\lambda - \hat{q}(S_t, A_t; \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(S_t, A_t; \mathbf{w})$$

- Backward TD(λ) – Il target rimane invariato:

$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}; \mathbf{w}) - \hat{q}(S_t, A_t; \mathbf{w})$$

$$E_t = \lambda \gamma E_{t-1} + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t; \mathbf{w})$$

$$\Delta \mathbf{w} = \alpha \delta_t E_t$$

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$

Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:

$S, A \leftarrow$ initial state and action of episode (e.g., ε -greedy)

Loop for each step of episode:

Take action A , observe R, S'

If S' is terminal:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$$

Go to next episode

Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε -greedy)

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha [R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})] \nabla \hat{q}(S, A, \mathbf{w})$$

$$S \leftarrow S'$$

$$A \leftarrow A'$$

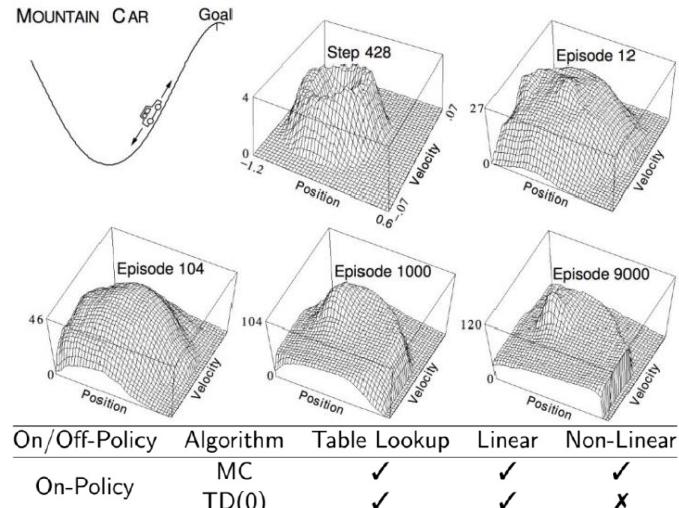
Questo tipo di algoritmo ha delle difficoltà quando dobbiamo utilizzarlo con le off-policy siccome non convergono.

A lato è presente un esempio della non convergenza, anche se siamo in un semplice modello dove non ci sono ricompense, accade che l'algoritmo off-policy TD va a divergere perché ci sono tante soluzioni al problema ed i valori vanno all'infinito.

La **divergenza** è dovuta a **tre caratteristiche**:

- Stiamo cercando di **approssimare la funzione**;
- Usiamo un metodo basato su **bootstrapping** (e non MC);
- Vogliamo allenare una **off-policy**.

Se le usiamo tutte e tre, probabilmente l'algoritmo diverge. Pertanto, dobbiamo evitare che tutte e tre compaiano, ad esempio utilizzando delle funzioni on-policy (convergono quando la funzione è lineare).



On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

21.2 METODI BATCH

Abbiamo visto metodi incrementali, cioè nella fase di training un campione alla volta produce un aggiornamento dei pesi (arriva un campione \rightarrow vengono applicate le formule precedenti \rightarrow si aggiornano i pesi).

Spesso capita che quando arriva un campione potrebbe far cambiare di molto il valore dei pesi (la funzione non tiene in considerazione ciò che è successo in precedenza), conviene invece utilizzare **metodi batch**, cioè non considerare solo un nuovo campione ma considerare l'esperienza recente dell'agente (un batch di campioni).

Quindi abbiamo una funzione che vogliamo approssimare (value function approximation) $\hat{v}(s; \mathbf{w}) \approx v_\pi(s)$ e l'esperienza D costituita da coppie $\langle \text{stato}, \text{valore} \rangle$:

$$\mathcal{D} = \{\langle S_1, v_1^\pi \rangle, \langle S_2, v_2^\pi \rangle, \dots, \langle S_T, v_T^\pi \rangle\}$$

Con i metodi batch, l'obiettivo è considerare il problema nel suo insieme, cioè scoprire i valori migliori per w che approssimano meglio i campioni di μ , in pratica i pesi devono adattarsi a tutta l'esperienza precedente e non solo all'ultimo campione.

Per fare questo, si utilizzano gli algoritmi **Least Squares** che individuano il vettore dei parametri \mathbf{w} che minimizza l'errore quadratico medio tra $\hat{v}(s; \mathbf{w})$ e i valori target $v_\pi(s)$:

$$LS(\mathbf{w}) = \sum_{t=1}^T (v^\pi - \hat{v}(s_t; \mathbf{w}))^2 = \mathbb{E}_{\mathcal{D}} [(v^\pi - \hat{v}(s; \mathbf{w}))^2]$$

EXPERIENCE REPLAY:

L'utilizzo di metodi batch ha portato a dei miglioramenti negli algoritmi sotto il nome di ***experience replay***.

Nella pratica, tutte le coppie $\langle \text{stato}, \text{valore} \rangle$ vengono memorizzate in un database e quando viene fatta la valutazione su un nuovo campione viene preso dal database un mini-batch così che l'aggiornamento dei pesi non viene fatto in relazione all'ultimo campione ma considerando la storia passata dell'agente:

- Ripete:

1. Campiona stato e valore dall'esperienza D :

$$\langle s, v^\pi \rangle \sim D$$

2. Esegui lo stochastic gradient descent update:

$$\Delta w = \alpha(v^\pi - \hat{v}(s; w)) \nabla_w \hat{v}(s; w)$$

- Converge alla soluzione least squares:

$$w = \arg \min_w LS(w)$$

Questo tipo di idea è alla base di un algoritmo chiamato Deep Q-Network (DQN).

DEEP Q-NETWORK (DQN):

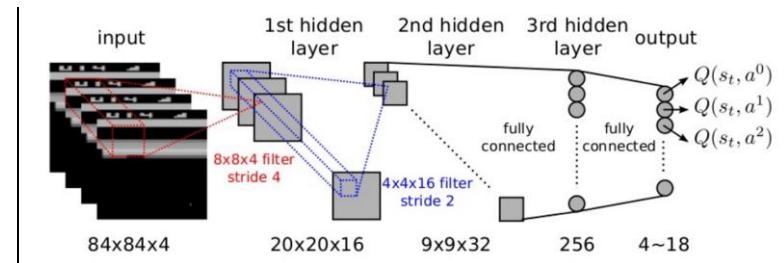
DQN usa ***experience replay*** e ***Q-target fissati***:

- Esegue l'azione a_t in base alla policy ϵ -greedy;
- Memorizza le transizioni (stato, azione, ricompensa, stato successivo) $(s_t, a_t, r_{t+1}, s_{t+1})$ nella ***replay memory*** D ;
- Campiona un ***mini-batch casuale di transizioni*** (s, a, r, s') ;
- Calcola i Q-learning target rispetto ai vecchi parametri fissati w^- ;
- Ottimizza la MSE tra la Q-network e i Q-learning target:

$$\mathcal{L}_i(w_i) = \mathbb{E}_{s,a,r,s' \sim \mathcal{D}_i} \left[\left(r + \gamma \max_{a'} Q(s', a'; w_i^-) - Q(s, a; w_i) \right)^2 \right]$$

- Utilizza una variante del ***stochastic gradient descent***.

Questo tipo di algoritmo è stato usato per dei giochi Atari (Atari-DQN), usa l'apprendimento end-to-end perché prendono come input 4 fotogrammi del gioco, attraversano una rete multi-layer e producono l'azione da fare.



LINEAR LEAST SQUARES PREDICTION:

Per gli algoritmi di predizione e controllo possiamo definire delle tecniche per calcolare l'errore quadratico efficientemente. Abbiamo visto come aggiornare i pesi in maniera incrementale, questa iterazione potrebbe richiedere molte operazioni per convergere, ma se usiamo come funzione di approssimazione una funzione lineare l'operazione del calcolo dei pesi si può semplificare, utilizzando la ***linear value function approximation*** $\hat{v}(s; w) = \mathbf{x}(s)^T w$, possiamo risolvere la soluzione least squares direttamente, questo perché volendo minimizzare l'errore quadratico quello che corrisponde è andare a determinare i valori dei pesi per i quali la sommatoria porta a 0:

$$\begin{aligned} \mathbb{E}_{\mathcal{D}}[\Delta w] &= 0 \\ \alpha \sum_{t=1}^T \mathbf{x}(S_t)(v_t^\pi - \mathbf{x}(S_t)^T w) &= 0 \\ w &= \left(\sum_{t=1}^T \mathbf{x}(S_t) \mathbf{x}(S_t)^T \right)^{-1} \sum_{t=1}^T \mathbf{x}(S_t) v_t^\pi \end{aligned}$$

Quindi, minimizzare la differenza tra valore corretto (true value) e valore predetto, corrisponde ad avere l'equazione sopra uguale a 0, ma per poter rendere 0 l'equazione bisogna determinare il valore di v_t^π (valori per i quali l'equazione va a 0).

Se portiamo v_t^π sulla sinistra calcoliamo per quali valori dei pesi l'errore di azzera. Anche qui, i valori della value function possiamo stimarli con le funzioni:

- Least Squares Monte-Carlo (***LSMC***) usa return $v_t^\pi \approx G_t$:

$$w = \left(\sum_{t=1}^T \mathbf{x}(S_t) \mathbf{x}(S_t)^T \right)^{-1} \sum_{t=1}^T \mathbf{x}(S_t) G_t$$

- Least Squares TD (***LSTD***) usa TD target $v_t^\pi \approx R_{t+1} + \gamma \hat{v}(S_{t+1}; w)$:

$$w = \left(\sum_{t=1}^T \mathbf{x}(S_t) (\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^T \right)^{-1} \sum_{t=1}^T \mathbf{x}(S_t) R_{t+1}$$

- Least Squares TD(λ) (***LSTD***) usa λ -return $v_t^\pi \approx G_t^\lambda$:

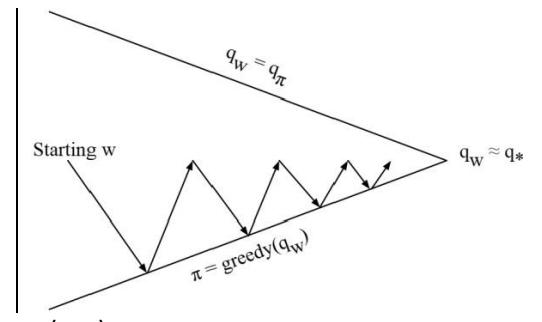
$$w = \left(\sum_{t=1}^T E_t (\mathbf{x}(S_t) - \gamma \mathbf{x}(S_{t+1}))^T \right)^{-1} \sum_{t=1}^T \mathbf{x}(S_t) R_{t+1}$$

In ogni caso bisogna risolvere direttamente il punto fisso di MC/TD/TD(λ).

LEAST SQUARES CONTROL:

Stesse operazioni possiamo farle per la fase di controllo, quindi possiamo usare come valutazione della policy una versione Least Square Q-Learning per poi effettuare il miglioramento della policy con greedy.

Le formule sono le stesse, cambia l'experience che è fatto da (stato, azione, peso) e non solo (stato, peso) come per il predict.



Quindi, vogliamo approssimare $q_\pi(s, a)$ usando combinazioni lineari delle feature $\mathbf{x}(s, a)$:

$$\hat{q}(s, a; \mathbf{w}) = \mathbf{x}(s, a)^T \mathbf{w} \approx q_\pi(s, a)$$

Minimizzare il least squares error tra $\hat{q}(s, a; \mathbf{w})$ e $q_\pi(s, a)$:

- Dall'esperienza generata usando la policy π
- Composta da coppie $\langle(\text{stato}, \text{azione}), \text{valore}\rangle$

$$\mathcal{D} = \{(s_1, a_1), v_1^\pi\}, \dots, (s_T, a_T), v_T^\pi\}$$

22. MODEL-BASED REINFORCEMENT LEARNING

Fino ad ora, abbiamo visto apprendimento di una value function e di una policy dall'**esperienza**, con algoritmi **Model-free RL**. Ora vediamo l'apprendimento di un **modello** dall'**esperienza**, fare planning per costruire una value function o una policy e integrare learning e planning in un'unica architettura.

MODEL-FREE RL:

- Nessun modello;
- Apprendimento della value function (e/o della policy) dall'esperienza.

Non avendo nessun modello bisogna per forza effettuare delle azioni, accumulando esperienza, da questa definiamo value function e policy.

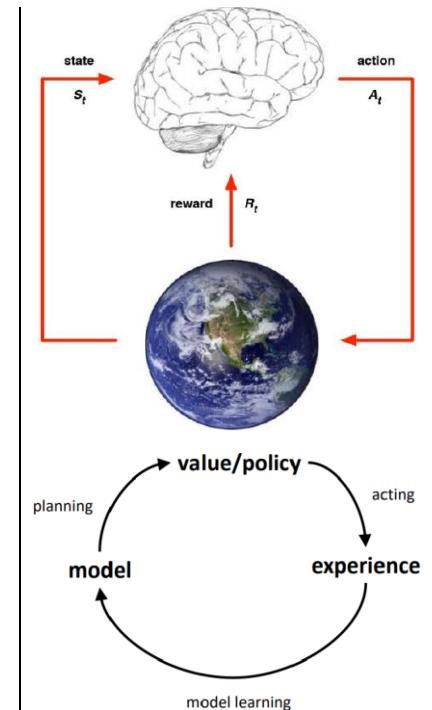
MODEL-BASED RL:

- Apprendimento di un modello dall'esperienza;
- Pianificare la value function (e/o la policy) dal modello.

Adesso apprendiamo il modello usando l'esperienza, con questo modello definiamo value function e policy in maniera indiretta.

L'obiettivo del RL è determinare la migliore policy.

Dalla policy decido quale azione effettuare che definisce l'esperienza (stato, azione, ricompensa) e nel model learning, tutta l'esperienza acquisita non viene usata per determinare la policy ma per costruire un modello. Dopo costruito il modello si può fare planning determinando così la policy.



Il vantaggio principale è che si può apprendere in maniera efficiente con metodi di apprendimento supervisionato.

Lo svantaggio è che si definisce un modello basato su simulazioni e la policy viene determinata con algoritmi approssimati su un modello approssimato, pertanto abbiamo una quantità approssimata alta (fonti di errore di approssimazione).

Le due fasi dell'algoritmo sono come **costruire un modello** e come **fare pianificazione**.

MODEL LEARNING:

Un modello M_η è una rappresentazione di un Markov Decision Process (MDP) $\langle S, A, P, R \rangle$ (stati, azioni, transizioni, ricompense) parametrizzato tramite η . Si supponga che lo spazio degli stati S e lo spazio delle azioni A siano noti, vogliamo determinare il modello di transizioni e quali sono le ricompense.

Quindi un modello $M_\eta = \langle P_\eta, R_\eta \rangle$ rappresenta le **transizioni di stato** $P_\eta \approx P$ e le **ricompense** $R_\eta \approx R$:

$$S_{t+1} = P_\eta(S_{t+1}|S_t, A_t)$$

$$R_{t+1} = R_\eta(R_{t+1}|S_t, A_t)$$

(quale è la probabilità di trovarmi nello stato S_{t+1} se sono in S_t ed eseguo una azione A_t . Stessa cosa con R)

In genere si assume l'**indipendenza condizionale** tra le transizioni di stato e le ricompense:

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

L'obiettivo è stimare il modello M_η dall'esperienza $\{S_1, A_1, R_2, \dots, S_T\}$.

Con algoritmi di apprendimento di modelli è possibile trasformare il problema in un problema di apprendimento supervisionato:

$$S_1, A_1 \rightarrow R_2, S_2$$

$$S_2, A_2 \rightarrow R_3, S_3$$

...

$$S_{T-1}, A_{T-1} \rightarrow R_T, S_T$$

Dall'esperienza acquisita facendo azioni, possiamo costruire il modello e questa esperienza è reale, se in un certo stato faccio una certa azione ci si trova una ricompensa ed uno stato di arrivo. Ripetendo tutto ciò ci si costruisce uno storico così da definire una funzione di ricompensa e una funzione di transizione.

Apprendere la ricompensa ($s, a \rightarrow r$) è un **problema di regressione** siccome sto stimando r che è un numero, mentre l'apprendimento di $s, a \rightarrow s'$ è un **problema di stima della densità** perché si ha che in s eseguendo a si arriva in s' con la probabilità del 20% o in s'' con un'altra probabilità ecc...

Basta andare a definire la loss e andarla a minimizzarla (classico problema di apprendimento supervisionato).

Esistono diversi Learning Model, come **Table Lookup Model**, Linear Expectation Model, Deep Belief Network Model, ...

MODEL LEARNING CON TABLE LOOKUP:

Il modello è un MDP esplicito $\langle \hat{P}, \hat{R} \rangle$ (funzione di transizione e funzione ricompensa), per determinare la funzione che si vuole apprendere teniamo traccia di quante volte visitiamo uno stato e azione, ovvero conta le visite $N(s,a)$ per ogni coppia stato-azione:

$$\hat{P}_{s,s'}^a = \frac{1}{N(s,a)} \sum_{t=1}^T \mathbf{1}(S_t, A_t, S_{t+1}; s, a, s')$$

$$\hat{R}_s^a = \frac{1}{N(s,a)} \sum_{t=1}^T \mathbf{1}(S_t, A_t; s, a) R_t$$

(la sommatoria dice quante volte da s si va in s' con l'azione a , diviso quante volte si è andati in s e si è eseguito a)

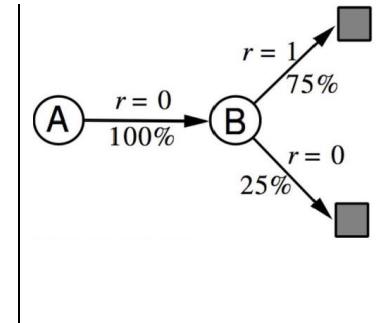
Alternativamente, ad ogni time-step t si memorizza la tupla esperienza $\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$.

Per campionare il modello si scelgono casualmente le tuple corrispondenti $\langle s, a, , \rangle$.

Esempio AB:

Supponiamo di avere due stati A e B, nessun sconto ed abbiamo 8 episodi che sono:

- A, 0, B, 0
- B, 1
- B, 0



Questa è l'esperienza, andiamo ad usare l'algoritmo visto prima per apprendere un modello. Da questa esperienza apprendiamo che quando trova la A poiché compare una sola volta, e da A vado in B, non da nessuna ricompensa. Mentre con lo stato B si può dire che su otto comparse, sei volte da 1 e due volte da 0, quindi abbiamo che il 75% di volte otteniamo 1 e il restante 0.

Abbiamo costruito un modello table lookup dall'esperienza, lavorando sulle frequenze.

Tutto ciò appena fatto era la prima fase ovvero il model learning con table lookup, la seconda fase è il planning.

PLANNING CON UN MODELLO:

Per fare planning, dato un modello, vogliamo fare pianificazione andando a determinare una policy o una value function.

Possiamo farlo con un qualsiasi algoritmo (value iteration, policy iteration, tree-search ...), però gli algoritmi che si utilizzano per fare planning sono quelli basati su campionamento.

- Pertanto, abbiamo un modello $M_\eta = \langle \hat{P}, \hat{R} \rangle$ appreso;
- Utilizzando il modello possiamo generare dei campioni avendo un *campione di esperienza* dal modello:

$$S_{t+1} \sim P_\eta(S_{t+1}|S_t, A_t)$$

$$R_{t+1} = \mathcal{R}_\eta(R_{t+1}|S_t, A_t)$$

- Da questi possiamo applicare gli algoritmi model-free visti (Monte-Carlo, Salsa, Q-Learning ...).

I metodi sample-based planning sono spesso più efficienti.

Esempio AB:

Col modello acquisito prima, possiamo fare planning per stimare la value function.

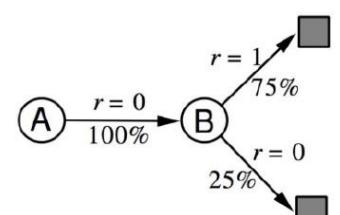
Dall'esperienza reale abbiamo costruito il modello al lato, ora da questo generiamo una esperienza campionata (non reale).

Quindi generiamo uno stato a caso, ad esempio B e la sua ricompensa probabilmente (75%) è 1, prossima generazione da B e questa volta esce 0, ecc...

Notare che A ha il 100% di guadagno 0.

Così facendo generiamo gli 8 campioni usando il modello costruito.

Quindi, una volta formato il modello possiamo usare uno degli algoritmi model-free come Monte Carlo che ci dirà $V(A)=1$ e $V(B)=0.75$.



Monte-Carlo learning: $V(A) = 1$; $V(B) = 0.75$
 Real experience Sampled experience

1. A, 0, B, 0	1. B, 1
2. B, 1	2. B, 0
3. B, 1	3. B, 1
4. B, 1	4. A, 0, B, 1
5. B, 1	5. B, 1
6. B, 1	6. A, 0, B, 1
7. B, 1	7. B, 1
8. B, 0	8. B, 0

Se il modello è **imperfetto**, cioè $\langle P_\eta, R_\eta \rangle \neq \langle P, R \rangle$, la policy non sarà ottimale e pertanto ci conviene usare algoritmi model-free.

INTEGRATED ARCHITECTURES:

Vogliamo definire una architettura integrata che va a combinare gli approcci di definizione di policy e di value function.

Consideriamo due fonti di esperienza:

- **Esperienza reale** – Campionata dall’ambiente (true MDP):

$$\begin{aligned} S' &\sim \mathcal{P}_{S,S}^{a'} \\ R &= \mathcal{R}_S^a \end{aligned}$$

Decidiamo una azione da fare, la eseguiamo ottenendo una ricompensa.

- **Esperienza simulata** – Campionata dal modello costruito (MDP approssimato):

$$\begin{aligned} S' &\sim P\eta(S'|S, A) \\ R &= \mathcal{R}\eta(R|S, A) \end{aligned}$$

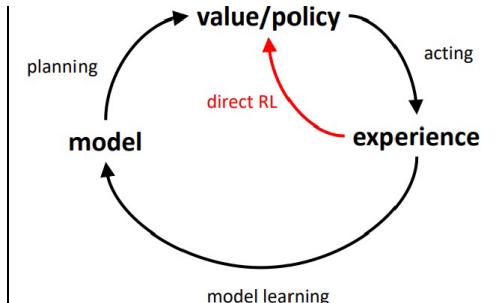
ARCHITETTURA DYNAL:

Fino ad ora, abbiamo visto con i model-free non costruiamo modelli e apprendiamo la value function dall’esperienza reale.

Quello appena visto è che usando algoritmi di campionamento possiamo costruire un **modello dall’esperienza reale** e fare **pianificazione** andando a creare delle simulazioni determinando la value function.

L’idea è di definire un’architettura chiamata **Dyna** che mette assieme i vantaggi di entrambi gli approcci, apprendendo il modello (come il model-based) costruito sull’esperienza reale, dopodiché combina **apprendere** e **pianificare** la value function e/o la policy a partire da esperienze sia reali che simulate.

Ha un **direct RL** che sarebbe il model-free.



ALGORITMO DYNAL-Q:

La prima parte è del Q-Learning che viene fatta nella realtà, scegliendo una azione, si va a vedere cosa si ottiene nella realtà (punto d) e aggiorno il Q-value.

Dopodiché andiamo a memorizzare il model (punto e).

Infine, c’è la parte di simulazione (punto f) dove ripete n volte le operazioni descritte.

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:

- $S \leftarrow$ current (nonterminal) state
- $A \leftarrow \varepsilon\text{-greedy}(S, Q)$
- Execute action A ; observe resultant reward, R , and state, S'
- $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
- $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment)
- Repeat n times:
 $S \leftarrow$ random previously observed state
 $A \leftarrow$ random action previously taken in S
 $R, S' \leftarrow Model(S, A)$
 $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

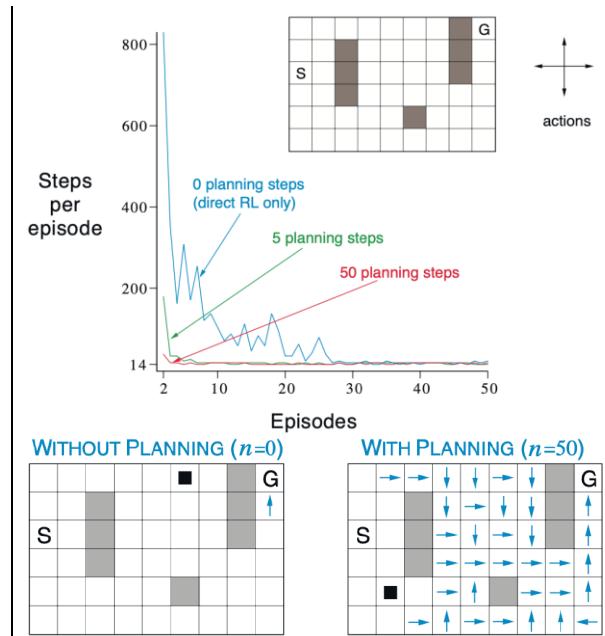
Esempio Maze – Learning curve:

Supponiamo di avere il labirinto a lato con S start e G guadagno, mentre tutte le azioni nel mezzo sono a ricompensa 0.

Abbiamo 3 grafici e cambiano in funzione di n (che sono le ripetizioni del ciclo nell’algoritmo):

- Il blu è $n=0$, dove si usa il classico Q-Learning (nessuna simulazione);
- Il verde è $n=5$, dove si fanno 5 campioni;
- Il rosso è $n=50$.

Da notare che sono indicati il numero di passi degli episodi prima di arrivare alla G, e dal grafico si vede che prima di apprendere il percorso è necessario analizzare circa 30 episodi.



LEARNING CON SIMULAZIONI:

Possiamo effettuare le simulazioni in maniera efficace. Ci troviamo in un certo stato reale e vogliamo generare una simulazione. Gli algoritmi che si utilizzano sono chiamati **forward search**, ovvero si vanno a cercare le azioni migliori che posso andare a eseguire nei vari stati.

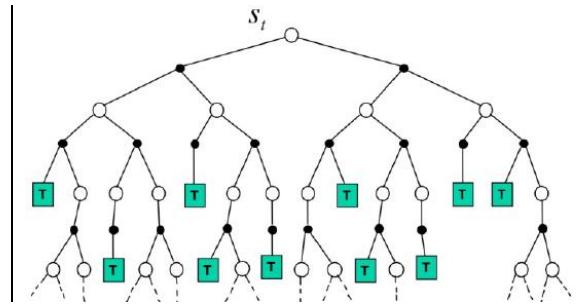
FORWARD SEARCH:

Gli algoritmi forward search selezionano l'azione migliore tramite il lookahead.

Costruiscono un albero di ricerca con lo stato (reale) corrente s_t alla radice, tutto ciò che c'è dopo la radice costituisce una simulazione.

Un algoritmo del genere va a guardare un passo alla volta quale è l'azione migliore attraversando i vari archi, costruendo un albero di ricerca in base all'azione migliore.

Non è necessario risolvere l'intero MDP, ma solo i sub-MDP da ora.



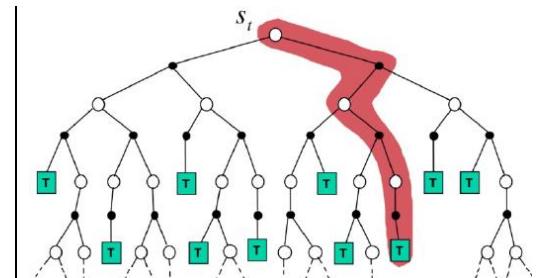
SIMULATION-BASED SEARCH:

Con un algoritmo di forward search costruiamo una simulazione T , a partire da uno stato s_t e scegliendo varie azioni che mi portano ad una foglia.

Simulare episodi di esperienza con il modello:

$$\{s_t^k, A_t^k, R_{t+1}^k; \dots, S_T^k\} \sim \mathcal{M}_v$$

Una volta costruite le simulazioni è possibile utilizzare gli algoritmi di model-free (MC o Sarsa) per definire la policy o la value function.



SIMPLE MONTE-CARLO SEARCH:

La versione semplice è la seguente. Dato un **modello M_v** e una **simulazion policy π** , per ogni azione $a \in A$:

- Si vanno a simulare K episodi a partire dallo stato attuale (reale) s_t :

$$\{s_t, a, R_{t+1}^k; S_{t+1}^k, A_{t+1}^k, \dots, S_T^k\} \sim \mathcal{M}_v, \pi$$

- Si vanno poi a valutare le azioni in base al **guadagno medio** (Monte-Carlo evaluation):

$$Q(s_t, a) = \frac{1}{K} \sum_{k=1}^K G_t \xrightarrow{P} q_\pi(s_t, a)$$

- Una volta fatto questo per tutte le azioni, si va infine a selezionare l'azione corrente (reale) con il valore massimo:

$$a_t = \arg \max_{a \in \mathcal{A}} Q(s_t, a)$$

Questo corrisponde ad 1 passo di miglioramento di policy, perché scelgo l'azione migliore rispetto ad un singolo passo.

MONTE-CARLO TREE SEARCH (MCTS):

L'idea dietro all'algoritmo MCTS è quello di andare a considerare il guadagno da un dato nodo seguendo un'azione fino ad arrivare alla fine (costruendo un albero di ricerca).

Formalmente, dato un **modello M_v** :

- **Simuliamo K episodi** a partire dallo stato attuale s_t usando la simulation policy π :

$$\{s_t, A_t^k, R_{t+1}^k; S_{t+1}^k, A_{t+1}^k, \dots, S_T^k\} \sim \mathcal{M}_v, \pi$$

- **Costruiamo un albero di ricerca** contenente gli stati visitati e le azioni;

- Valutiamo gli stati $Q(s, a)$ attraverso il guadagno medio degli episodi da s, a :

$$Q(s, a) = \frac{1}{N(s, a)} \sum_{k=1}^K \sum_{u=t}^T \mathbf{1}(S_u, A_u; s, a) G_u \xrightarrow{P} q_\pi(s, a)$$

- Al termine della ricerca, selezionare l'azione corrente (reale) con il **valore massimo nell'albero di ricerca**:

$$a_t = \arg \max_{a \in \mathcal{A}} Q(s_t, a)$$

La parte importante è quella di simulazione, per determinare gli episodi (scendere fino ad uno stato terminale) si usa una policy π che si migliora durante la costruzione delle simulazioni. La simulazione fa uso di due policy:

- **Tree policy** (si migliora): scegliere le azioni per massimizzare $Q(S, A)$;
- **Default policy** (viene fissata): scegliere le azioni in modo casuale.

Quindi usa queste policy per generare la simulazione, dopodiché ripetere (per ogni simulazione):

- **Valuta** gli stati $Q(S, A)$ tramite la Monte-Carlo evaluation;
- **Migliora** la tree policy, ad esempio attraverso ϵ -greedy(Q).

È come se fosse un algoritmo di Monte-Carlo control ma applicato all'esperienza simulata. Alla fine, converge all'albero di ricerca ottimale, $Q(S, A) \rightarrow q_*(S, A)$.

Le seguenti fasi si eseguono per ogni simulazione:

Si vogliono creare tanti campioni, tra i quali generiamo una policy con un algoritmo di control già visto.
Migliori sono le simulazioni e migliore è la policy.
Quindi l'idea è che andiamo a costruire da S_t un albero in base alle scelte delle azioni che si fanno.

Nella fase iniziale si usa la tree policy, una volta arrivato alla foglia se questa viene scelta la si espande, pertanto non vengono espansate tutte le foglie ma solo quelle più significative.

Una volta espansa la foglia si fa simulazione, arrivando ad una ulteriore foglia.

Infine, si fa backup riportando il percorso a S_t .

Tutto questo viene fatto in maniera iterativa, generando campioni in base al limite di tempo che si ha.

I vantaggi di MCTS sono che la ricerca best-first è altamente selettiva siccome non espande tutte le foglie, valuta gli stati in modo dinamico (a differenza di DP), utilizza il campionamento per risolvere la curse of dimensionality, funziona per modelli black-box (richiede solo i campioni) ed infine è efficiente dal punto di vista computazionale, è parallelizzabile.

TEMPORAL-DIFFERENCE SEARCH:

Questo stesso tipo di algoritmo può essere usato anziché fare delle simulazioni in serie si usa la **Temporal-Difference**, dove anziché scendere fino alla foglia si effettua solo un passo e si fa aggiornamento tramite Sarsa.

Per il model-free RL e per la simulation-based search, il bootstrapping è utile:

- La TD search riduce la varianza ma aumenta il bias;
- La TD search è solitamente più efficiente della MC search;
- La TD(λ) search può essere molto più efficiente della MC search.

GO CASE STUDY:

Il GO è considerato il gioco da tavolo più difficile, di solito si gioca su un tavolo 19x19, ma anche 13x13 o 9x9.

I due giocatori posizionano le pietre alternativamente, le pietre circondate vengono catturate e rimosse. Il giocatore con più territori vince la partita

Dal punto di vista di apprendimento per rinforzo abbiamo la reward function (non scontata):

- $R_t = 0$ per tutti gli step non terminali $t < T$;
- $R_t = 1$ se il giocatore nero vince;
- $R_t = 0$ se il giocatore bianco vince.

Vogliamo definire una policy $\pi = \langle \pi_B, \pi_W \rangle$ seleziona le **mosse per entrambi i giocatori** (B, W), e vogliamo una policy che fa vincere il giocatore nero, pertanto la value function (quanto è valida la posizione s):

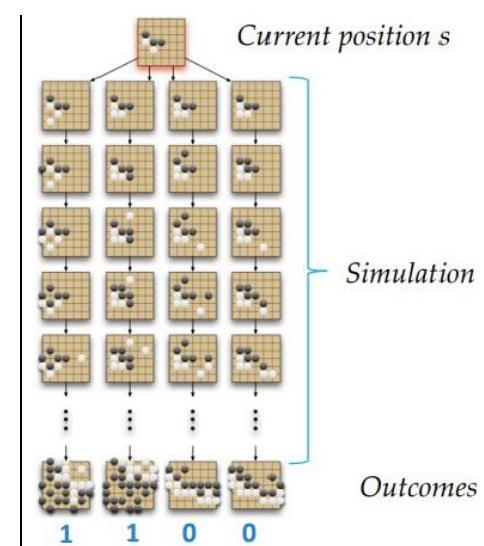
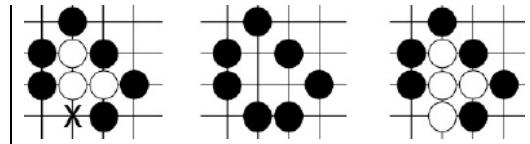
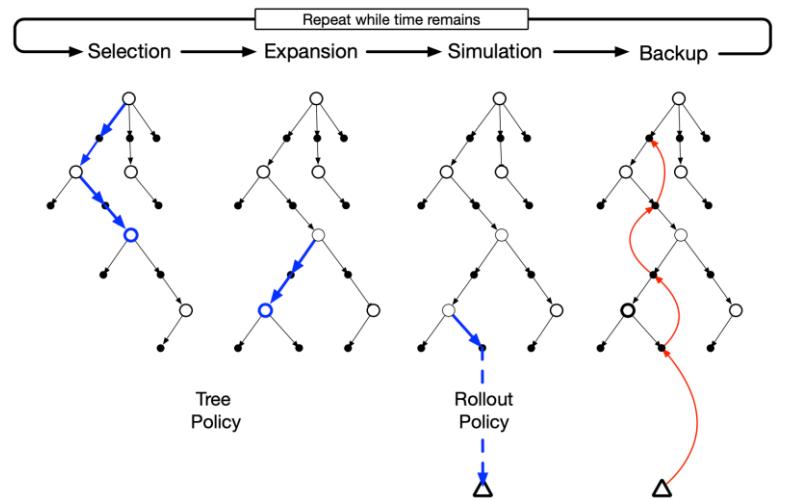
$$v_\pi(s) = \mathbb{E}[R_T | S = s] = P(\text{Black wins} | S = s)$$

$$v_*(s) = \max_{\pi_B} \min_{\pi_W} v_\pi(s)$$

GO – MONTE-CARLO EVALUATION:

Usando Monte-Carlo, partiamo da uno stato reale s , vengono effettuate delle simulazioni, continuando la partita in maniera casuale, fino a fine partita.

In questo caso abbiamo $V(s) = 2/4 = 0.5$ in cui ha vinto il nero.



APPLICAZIONE MONTE-CARLO TREE SEARCH:

All'inizio usiamo la default policy siccome l'albero non esiste

Si parte dalla radice, si fa una azione a caso arrivando a fine partita e vince in nero.

A questo punto facciamo backup e la vittoria è 1/1.

Si va a fare un'altra azione usando la tree policy, creando un figlio e da questo si effettua una simulazione che porta ad una sconfitta, pertanto si fa backup e la vittoria è ½ mentre il figlio ha 0/1.

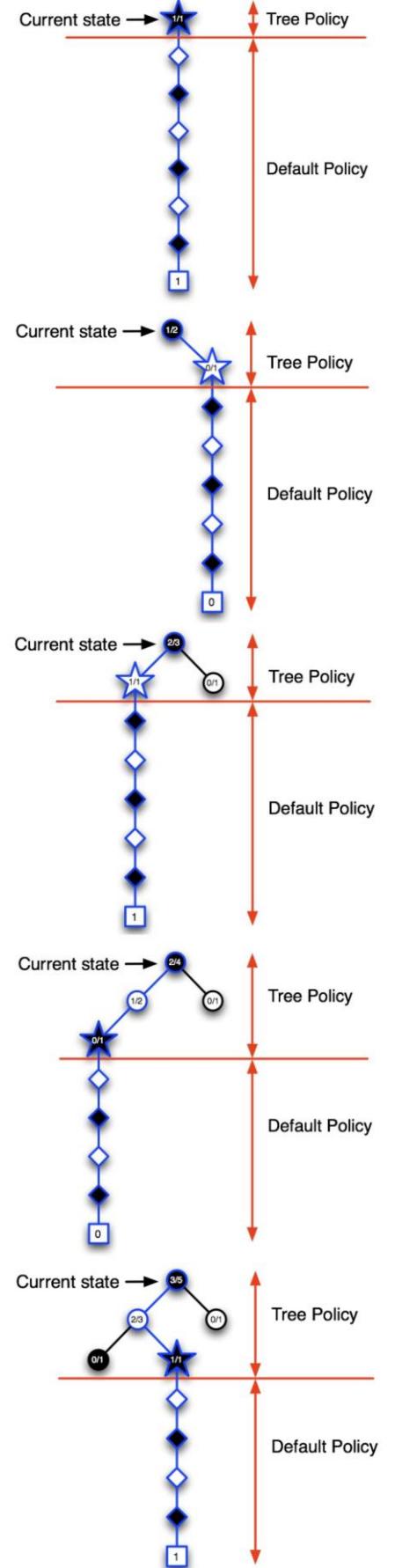
Tornando al current state si effettua un'altra azione che porta ad una vittoria e facendo backup la vittoria è 2/3, mentre quella del figlio appena creato è di 1/1.

Ora, la tree policy sfrutta il figlio che ha la vittoria maggiore, pertanto sceglierà di nuovo quello stato che verrà espanso.

Espandendo il figlio con maggior vittoria, lo si espande e con la simulazione otteniamo una sconfitta, aggiornando il current state a 2/4.

A questo punto, la tree policy continua ad espandere il figlio sinistro, ma andiamo alla sua destra. Con le simulazioni otteniamo una vittoria ed aggiorniamo il tutto.

Tutto ciò è stato fatto su solo 5 simulazioni, ma se espanso a molte di più, l'albero coprirà tutte le esperienze positive.



L'algoritmo **Alpha-Go** usa reti neurali convoluzionali per estrarre una rappresentazione significativa dello stato.

23. NATURAL LANGUAGE PROCESSING (NLP)

Il **Natural Language Processing (NLP)** è un problema che si vuole affrontare con tecniche di intelligenza artificiale. In realtà, questo tipo di problema può essere affrontato con modelli di regressione o Bayesiani, ma architetture più evolute lo risolvono in modo più efficiente.

Il NLP può essere usato in molti ambiti, come ad esempio rispondere ad una domanda di un quiz scritta in linguaggio naturale (non codificato) →

Ma anche per task come analisi di una mail per suggerire determinate azioni, ad esempio creare un appuntamento in modo automatico analizzando il testo della mail →

Ottimamente da una descrizione classificare questa in una categoria, ad esempio se un commento è positivo o negativo →

Altri task possono essere effettuare delle traduzioni, riconoscendo la lingua parlata dalla frase automaticamente oppure suggerire le prossime parole che possono seguire nella frase →

I task di NLP sono tantissimi e possono essere classificati in:

- **Risolti** (problemi banali), ad esempio riconoscere e-mail di spam, hanno una affidabilità molto alta;
- **Con una buona soluzione**, ad esempio traduttori oppure estrazione di informazioni da un testo;
- **Difficili**, ad esempio fare riassunti di un testo in modo automatico o dialogare con un bot.

DIFFICOLTÀ NEL CAPIRE IL LINGUAGGIO:

La difficoltà, in generale, per risolvere questi task sono le **ambiguità**, nel linguaggio naturale alcuni termini sono spesso ambigui siccome uno stesso termine si riferisce a più concetti oppure l'uso di idiomi o neologismi.

Per risolvere nel modo migliore bisogna avere una:

- Conoscenza del **linguaggio**, ovvero sapere come sono costruite le frasi;
- Conoscenza del **mondo**, ovvero sapere come è utilizzato il linguaggio;
- Conoscenza di come sono **combinati** le conoscenze precedenti.

Tutto ciò è un problema di classificazione, si cerca di costruire dei modelli che permettono di effettuare il task che si vuole. Ad esempio, avere un modello che dice che la traduzione "madison" in "house" è probabilmente corretta, rispetto a tradurre "L'avocat general" in "the general avocado" che non è corretta.

Quindi, a partire dalla conoscenza che abbiamo, si costruiscono i modelli.

23.1 ESPRESSIONI REGOLARI

Viene eseguita prima una fase di **pre-processing**, cioè il testo prima di essere elaborato viene normalizzato così da semplificare le fasi successive.

Una attività che si fa nel NLP è quella di estrarre stringhe utili per andare a fare poi l'analisi successiva, è possibile fare questo grazie alle **regular expressions**, da documenti si estraggono particolari stringhe.

Si utilizzano le parentesi quadre [] per denotare disgiunzioni:

Pattern	Matches
[wW]oodchuck	Woodchuck, woodchuck
[1234567890]	Any digit
Pattern	Matches
[A-Z]	An upper case letter
[a-z]	A lower case letter
[0-9]	A single digit
Pattern	Matches
[^A-Z]	Not an upper case letter
[^Ss]	Neither 'S' nor 's'
[^e^]	Neither e nor ^
a^b	The pattern a carat b

Si utilizza il trattino – per denotare i range:

Con ^ si negano i range, esempio tutto ciò che non è una lettera maiuscola:

Con pipe | si sceglie una delle parole definite:

Col ? sono opzionali, ad esempio u? la u è opzionale.
 Con * vuol dire 0 o più.
 Con + vuol dire 1 o più.
 Con . vuol dire qualsiasi simbolo:

Per andare al primo o ultimo carattere si usano le ancora, con apice ^ si indica l'inizio e con dollaro \$ la fine:

Con lo /b anche questa è un ancora, esempio si vuole catturare la "the":

Pattern	Matches
groundhog woodchuck	groundhog, woodchuck
yours mine	yours, mine
a b c	a, b, c
Pattern	Matches
colou?r	Optional previous char
color	color colour
oo*h!	0 or more of previous char
ooh!	oh! ooh! oooh! ooooh!
o+h!	1 or more of previous char
ooh!	oh! ooh! oooh! ooooh!
baa+	baa baaa baaaa baaaaa
beg.n	begin begun begun beg3n

Pattern	Matches
^ [A-Z]	Palo Alto
^ [^A-Za-z]	1 "Hello"
\.\$	The end.
.\$	The end? The end!
/the/	/the/
Misses capitalized examples	Misses capitalized examples
/[tT]he/	[tT]he/
• Returns other or theology	• Returns other or theology
/\b[tT]he\b/	\b[tT]he\b/

Quando si usano queste espressioni possono essere introdotti i **falsi positivi** e i **falsi negativi**, queste misure sono legate al concetto di precisione dell'algoritmo.

Le espressioni regolari possono essere utilizzate anche per costruire sistemi basilari che effettua un pattern matching, ad esempio un chat-bot dove un utente scriveva una frase ed esso rispondeva, estraendo porzioni della risposta tramite pattern matching sulla frase di input.

Esistono comandi Unix che permettono di utilizzare le espressioni regolari, ad esempio trasformando un documento secondo espressioni definite.

Comando: s/ regexp1/ pattern/

Esempio:

Vogliamo racchiudere tra parentesi angolari tutti i numeri in una frase:

the 35 boxes → the <35> boxes
 ↓
 s/ ([0-9]+)/<1>/

ELIZA:

Primo sistema di PNL che imitava uno psicoterapeuta rogeriano, gli veniva chiesto: "I need X" e li traduce in: "What would it mean to you if you got X?" (rigirandogli la domanda).

Esempio conversazione con Eliza:

Men are all alike.

IN WHAT WAY

They're always bugging us about something or other.

CAN YOU THINK OF A SPECIFIC EXAMPLE

Well, my boyfriend made me come here.

YOUR BOYFRIEND MADE YOU COME HERE

He says I'm depressed much of the time.

I AM SORRY TO HEAR YOU ARE DEPRESSED

Come lavora Eliza:

s/. I'M (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/

s/. I AM (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/

s/. all .*/IN WHAT WAY?/

s/. always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE?/

(Fa match con una regola e riscrive la risposta usando quegli elementi)

DIFFICOLTÀ DEL NUMERO DI PAROLE:

Un aspetto importante da considerare nelle frasi quando si analizzano è quante parole ci sono. Se viene analizzata una registrazione, alcuni suoni come "mmh" non sono parti del discorso e pertanto non dovrebbero essere analizzate.

Se si analizza ad esempio un titolo di un libro come "cat and cats" la parola cat e cats viene contata una volta siccome per il sistema può essere considerata dello stesso significato ma in realtà il senso della frase è diverso, pertanto bisogna analizzare anche il contesto.

Bisogna capire quante effettive parole ci sono, pertanto, bisogna fare una distinzione:

"they lay back on the San Francisco grass and looked at the stars and their"

- **Type:** un elemento del vocabolario, sono 13 siccome una parola può comparire più volte e viene contata una volta;
- **Token:** un'istanza nel testo corrente, sono 15 elementi.

Definendo formalmente tutto ciò in:

N = numero di token e V = vocabolario = insieme dei type ($|V|$ è la size del dizionario)

Esiste una regola che mette in relazione il numero di type con in numero di token, descrivendo quanto è grande il vocabolario rispetto al numero di parole, la formula è la seguente:

Heaps law = Herdan's law = $|V| = kN^\beta$ con $.67 < \beta < .75$

Più è grande il dataset più aumentano le nuove parole.

	Tokens = N	Types = V
Switchboard phone conversations	2.4 million	20 thousand
Brown corpus	1 million	38 thousand
Shakespeare	884,000	31 thousand
Google N-grams	1 trillion	13 million

CORPORA:

Una cosa importante dei dataset di NLP, chiamati **Corpora**, sono costruiti su particolari domini, ad esempio conversazioni telefoniche o libri di Shakespeare ecc...

Le parole contenute nei dataset dipendono molto dal tipo di informazione che vanno a memorizzare, è importante che questi Corpora vanno definiti assieme a dei metadati che descrivono da dove provengono le informazioni contenute, il tempo in cui sono state prese le informazioni siccome lo stile di scrittura cambia nei secoli (es. libri di Shakespeare ad oggi) ed altre caratteristiche, ma specificare in linguaggio con cui sono state scritte è importante siccome esistono molte lingue nel mondo essendoci anche vari dialetti.

Infatti, quando si rende disponibile un **Corpus** per la ricerca bisogna associare dei metadati, come il perché è stato creato, da chi, ecc...

PRE-PROCESSING:

Avendo un testo da poter comprendere, una volta acquisito potrebbe essere scritto con tempi differenti o con una struttura diversa, per semplificare l'analisi successiva la prima cosa che viene fatta è una **normalizzazione** che riduce la variabilità del testo rendendo il task più semplice.

Esempi di normalizzazione è la **segmentazione**, dove si hanno parole composte che vengono decomposte oppure definire dei token in una frase, normalizzando in un unico formato.

TOKENIZZAZIONE IN UNIX:

Abbiamo un documento e vogliamo estrarre **token** da esso. È possibile farlo tramite unix col comando **tr** che va a prendete tutte le parole separati da spazi:

```
tr -sc 'A-Za-z' '\n' < shakes.txt
      | sort
      | uniq -c
```

Change all non-alpha to newlines
Sort in alphabetical order
Merge and count each type

Il comando andrà a sostituire tutto ciò che non è lettera con lo \n, ordinato fondendo tutte le parole uguali e conteggiarle.

In realtà, ci sono dei problemi da affrontare, esempio:

- Presenza di apostrofi: *Finland's capital* → *Finland Finlands Finland's?*
- L'uso dei trattini: *Lowercase* → *lower-case lowercase lower case?*
- Due parole in una: *San Francisco* → *one token or two?*
- Punteggiature: *m.p.h., PhD.* → ??

Tutte queste problematiche derivano principalmente dal linguaggio parlato, esempio in italiano: *L'insieme* → *una parola o due?*, oppure in cinese non esistono gli spazi e le parole sono dei simboli, pertanto in tal caso dovremmo segmentare le parole in token.

MAX-MATCH:

Un algoritmo che va bene nel caso del cinese, ma non per le altre lingue, è un algoritmo greedy chiamato **max-match**, cercando la parola più lunga che fa match con la stringa analizzata: *The table down there* → *the table down there*, ma potrebbe essere anche *theta bled own there*. Per il cinese:

莎拉波娃现在居住在美国东南部的佛罗里达 →

莎拉波娃 现在 居住 在 美国 东南部 的 佛罗里达

NORMALIZZAZIONE:

Dopo aver fatto la tokenizzazione si effettua la **normalizzazione**, che fa sì che tutti i termini compaiano nello stesso formato così da semplificare il task successivo, ad esempio se troviamo *U.S.A.* (puntato) si sostituisce con *USA*, oppure se si cerca *windows* bisogna trovare le sue combinazioni con maiuscolo e minuscolo (es. *Windows*, *window* o *Window*).

Case folding: di solito si portano tutti i termini in minuscolo tranne in alcuni casi, ad esempio le sigle *e.g.* = *General Motors*, conviene lasciarle maiuscole.

Lemmatizzazione: si riducono i modi di presentare alcune informazioni come i verbi, ad esempio *am, are, is* → *be* oppure *car, cars, car's, cars'* → *car*. Esempio di lemmatizzazione:

the boy's cars are different colors → *the boy car be different color*

Stemming: le parole hanno una parte principale e degli affissi che è la parte che si allunga alla fine, lo stemming è togliere tutti gli affissi, restituendo la frase solo con la parte importante, ad esempio *automate(s), automatic, automation* possono essere ridotta ad *automat*. Esempio di stemming:

for example compressed and compression are both accepted as equivalent to compress.

→ *for exampl compress and compress ar both accept as equival to compress.*

Algoritmo di Porter (English stemmer):

Step 1a

sses	→ ss	caresses	→ caress
ies	→ i	ponies	→ poni
ss	→ ss	caress	→ caress
s	→ Ø	cats	→ cat

Step 1b

(*v*)ing	→ Ø	walking	→ walk
		sing	→ sing
(*v*)ed	→ Ø	plastered	→ plaster
...			

Step 2 (for long stems)

ational	→ ate	relational	→ relate
izer	→ ize	digitizer	→
		digitize	
ator	→ ate	operator	→ operate
...			

Step 3 (for longer stems)

al	→ Ø	revival	→ reviv
able	→ Ø	adjustable	→ adjust
ate	→ Ø	activate	→ activ
...			

Segmentazione: il ? o ! possono essere ambigui all'interno di una frase oppure il “.” (punto) può avere diversi significati, ad esempio per terminare una frase ma anche per abbreviazioni. Altro esempio sono i numeri decimali, in inglese si usa il punto e non la virgola per dividere la parte intera da quella decimale (3,14 → 3.14).

Per riconoscere un punto fine frase o abbreviatore è possibile addestrare un classificatore che lo va a distinguere:

Costruendo un albero di decisione vediamo se c'è dello spazio dopo il punto allora è di fine frase altrimenti è un abbreviatore, ecc...

Per renderlo più preciso possiamo vedere non solo lo spazio dopo il punto ma anche se la parola seguente ha la lettera maiuscola, mentre se fosse minuscola potrebbe riferirsi ad una sigla, ecc....

DISTANZA MINIMA:

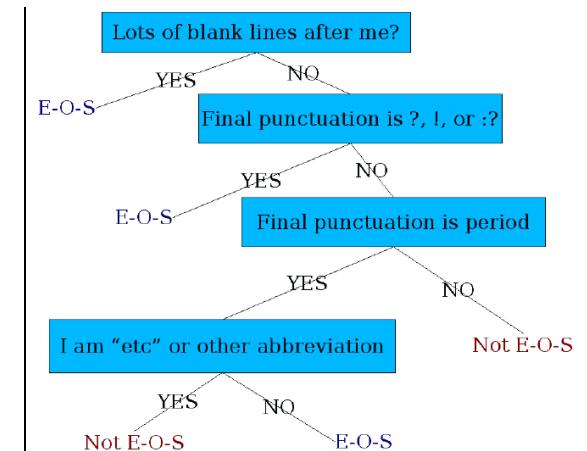
Un altro task utile in problemi di NLP è sapere la **distanza tra due stringhe**, serve principalmente quando si vuole correggere una frase, ad esempio word segnala una parola in rosso e suggerisce quella corretta, andando a controllare nel dizionario e dando la parola con la distanza più piccola rispetto alla parola in esame.

Un altro ambito di utilizzo è quello biologico di sequenze del tipo →

Si vuole determinare l'allineamento tra le due sequenze.

L'algoritmo usato per trovare la distanza tra due stringhe, si chiama **Minimum Edit Distance**, va a conteggiare le operazioni da fare per trasformare una stringa in un'altra e sono inserimento, cancellazione e sostituzione.

Nell'esempio precedente, viene fatta una cancellazione, 3 sostituzioni e un inserimento, pertanto il costo sarà 5 (se il costo di operazione è unitario). Esiste una versione in cui non c'è la sostituzione ma viene fatta prima la cancellazione e poi l'inserimento (il costo aumenta).



I N T E * N T I O N
| | | | | | | | | |
* E X E C U T I O N
d s s i s

ALGORITMO MIN EDIT DISTANCE:

Per calcolare la distanza minima non possiamo usare un algoritmo di ricerca siccome è troppo complesso, ma l'algoritmo che si utilizza è un algoritmo di **programmazione dinamica** che calcola la distanza tra due parole riducendolo al problema di conoscere la distanza delle sottostringhe. L'algoritmo è:

Si prende il minimo tra

- $D(i-1, j) + 1$ che è stato considerato l'inserimento;
- $D(i, j-1) + 1$ è la cancellazione;
- $D(i-1, j-1) + (2 o 0)$, si aggiunge 2 quando sono diversi, 0 se sono uguali.

Fatti i confronti va a costruire una matrice, chiamata Edit Distance Table. Una volta ottenuto il costo, facciamo backtrace e ricavare le operazioni eseguite.

Initialization

$$D(i, 0) = i$$

$$D(0, j) = j$$

Recurrence Relation:

For each $i = 1..M$
For each $j = 1..N$

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 \\ D(i, j-1) + 1 \\ D(i-1, j-1) + 2; & \text{if } X(i) \neq Y(j) \\ 0; & \text{if } X(i) = Y(j) \end{cases}$$

Termination:

$D(N, M)$ is distance

Esempio Edit Distance Table:

Creazione della tabella:

N	9										
O	8										
I	7										
T	6										
N	5										
E	4										
T	3										
N	2										
I	1										
#	0	1	2	3	4	5	6	7	8	9	
#	E	X	E	C	U	T	I	O	N		

$$D(i, j) = \min \begin{cases} D(i-1, j) + 1 \\ D(i, j-1) + 1 \\ D(i-1, j-1) + 2; & \text{if } S_1(i) \neq S_2(j) \\ 0; & \text{if } S_1(i) = S_2(j) \end{cases}$$

Riempimento della tabella:

N	9	8	9	10	11	12	11	10	9	8	9
O	8	7	8	9	10	11	10	9	8	9	9
I	7	6	7	8	9	10	9	8	7	8	9
T	6	5	6	7	8	9	8	9	8	9	10
N	5	4	5	6	7	8	9	10	11	10	10
E	4	3	4	5	6	7	8	9	10	9	9
T	3	4	5	6	7	8	7	8	9	8	8
N	2	3	4	5	6	7	8	7	8	7	7
I	1	2	3	4	5	6	7	6	7	8	8
#	0	1	2	3	4	5	6	7	8	9	
#	E	X	E	C	U	T	I	O	N		

Backtrace:

n	9	↓ 8	↖ 9	↖ 10	↖ 11	↖ 12	↓ 11	↓ 10	↓ 9	↖ 8	
o	8	↓ 7	↖ 8	↖ 9	↖ 10	↖ 11	↓ 10	↓ 9	↖ 8	← 9	
i	7	↓ 6	↖ 7	↖ 8	↖ 9	↖ 10	↓ 9	↖ 8	← 9	← 10	
t	6	↓ 5	↖ 6	↖ 7	↖ 8	↖ 9	↖ 8	← 9	← 10	↖ 11	
n	5	↓ 4	↖ 5	↖ 6	↖ 7	↖ 8	↖ 9	↖ 10	↖ 11	↖ 10	
e	4	↖ 3	← 4	↖ 5	↖ 6	↖ 7	↖ 8	↖ 9	↖ 10	↖ 9	
t	3	↖ 4	↖ 5	↖ 6	↖ 7	↖ 8	↖ 7	↖ 8	↖ 9	↖ 8	
n	2	↖ 3	↖ 4	↖ 5	↖ 6	↖ 7	↖ 8	↓ 7	↖ 8	↖ 7	
i	1	↖ 2	↖ 3	↖ 4	↖ 5	↖ 6	↖ 7	↖ 6	← 7	← 8	
#	0	1	2	3	4	5	6	7	8	9	
#	e	x	e	c	u	t	i	o	n		

Allineamento:

INTE * NTION
| | | | | | | |
* EXECUTION

Esiste anche una **versione pesata**, siccome capita molto spesso che le sostituzioni sono frequenti per alcuni tipi di caratteri, gli errori sono dovuti spesso alla vicinanza dei tasti sulla tastiera. Pertanto, si può definire una distanza pesata così da considerare ad esempio "n" e "m" che sono vicini sulla tastiera. Tutti questi pesi vengono presi da una tabella apposita.

23.2 N-GRAMS

Vogliamo definire dei **Probabilistic Language Models**, che hanno come scopo quello di determinare la probabilità di una certa frase. Avendo una sequenza di parole si vuole determinare con che probabilità quella frase può apparire.

In realtà, questo tipo di probabilità può essere usata in diversi contesti:

Traduttore: quando si va a tradurre una data parola può essere tradotta in diversi modi, per capire quale dei termini utilizzate basta vedere quale è più probabile. Se nel documento in questione appare molte volte una data frase è giusto usarla rispetto ad un'altra, cioè $P(\text{high winds tonite}) > P(\text{large winds tonite})$.

Correzione: se sappiamo la probabilità di avere dopo *fifteen* la parola *minutes* è possibile suggerire che è presente un errore:

The office is about fifteen minuets from my house → $P(\text{about fifteen minutes from}) > P(\text{about fifteen minuets from})$
(in word sono gli errori in giallo)

Speech Recognition: le frasi ricostruite da una registrazione possono essere ambigue:

$P(\text{I saw a van}) >> P(\text{eyes awe of an})$

Pertanto, si va a scegliere delle frasi di senso compiuto, disambiguando il riconoscimento del parlato.

Possono essere usati anche in altri contesti come Summarization, question-answering, ecc...

PROBABILISTIC LANGUAGE MODELS:

Il problema è data una frase W , calcolare la probabilità che appaia la frase W , ma questa la possiamo scrivere come la probabilità che la sequenza di parole appaia. Lo stesso tipo di probabilità è equivalente a calcolare la probabilità condizionata, $P(W)$ o $P(w_n | w_1, w_2, \dots, w_{n-1})$ è chiamato **language model**. Ad esempio:

$$P(W) = P(w_1, w_2, w_3, w_4, w_5, \dots, w_n) \rightarrow P(w_5 | w_1, w_2, w_3, w_4)$$

I termini vengono chiamati **grammar** o **LM**.

Per calcolare la **probabilità $P(W)$** , ad esempio:

$$P(\text{its, water, is, so, transparent, that})$$

Si sfrutta la **chain rule** dove ci ricaviamo la probabilità di due eventi come prodotto dei due eventi, uno singolo e uno condizionato al contrario, formalmente è $P(A, B) = P(A)P(B|A)$:

$$P(w_1, w_2, w_3, w_4, w_5, \dots, w_n) = P(w_1)P(w_2 | w_1)P(w_3 | w_1, w_2) \dots P(w_n | w_1, \dots, w_{n-1})$$

$$P(w_1 w_2 \dots w_n) = \prod_i P(w_i | w_1 w_2 \dots w_{i-1})$$

Sinteticamente è una produttoria:

Se vogliamo la probabilità esatta dovremo calcolarci tutti questi prodotti, ovvero:

$$P(\text{its}) \times P(\text{water} | \text{its}) \times P(\text{is} | \text{its water}) \times P(\text{so} | \text{its water is}) \times P(\text{transparent} | \text{its water is so})$$

Per il calcolo alla fine non sono altro che fare dei conteggi di quante volte appare la parola rispetto a quante volte appare da sola, ma il problema è calcolare l'ultima probabilità condizionata che è condizionata da tutte le precedenti variabili.

Quindi se ad esempio abbiamo:

$$\begin{aligned} P(\text{the} | \text{its water is so transparent that}) &= \\ \frac{\text{Count}(\text{its water is so transparent that the})}{\text{Count}(\text{its water is so transparent that})} \end{aligned}$$

Dovremmo conteggiare quante volte si presenta la frase con "the" alla fine e quante volte si presenta senza. Il problema per fare questi calcoli sono i dati che dobbiamo avere per calcolare queste informazioni, non si hanno abbastanza dati per fare questi calcoli.

ASSUNZIONI DI MARKOV:

Per semplificare il problema si applica l'assunzione di Markov, dove non si considerano più le n parole precedenti ma assumiamo che c'è una dipendenza parziale (riducendo la dipendenza dalle parole). Quindi, la probabilità che "the" preceduto dalle 5 parole precedenti può essere approssimata da la probabilità di avere "the" preceduta da "that":

$$P(\text{the} | \text{its water is so transparent that}) \approx P(\text{the} | \text{that})$$

Oppure

$$P(\text{the} | \text{its water is so transparent that}) \approx P(\text{the} | \text{transparent that})$$

In altre parole, approssimiamo ciascun componente nel prodotto:

$$P(W_1, W_2, \dots, W_{i-1}) \approx P(W_i | W_{i-k}, \dots, W_{i-1})$$

Non avendo più la produttoria precedente, ma fissiamo un k dicendo che la variabile è dipendente solo dalle k parole precedenti e non da tutte le parole precedenti.

MODELLI GRAM:

Possiamo costruire dei modelli che determinano queste probabilità:

- **Unigram**: è il caso più semplice, assume che ogni parola è indipendente dalle parole precedenti:

$$P(W_1 W_2 \dots W_n) \approx \prod_i P(W_i)$$

Se si prende un language model per generare frasi, l'unigram genererà parole completamente a caso in base a quanto sono frequenti quelle parole (non considera il contesto, pertanto, non si usa nella pratica).

- **Bigram**: si memorizzano le probabilità di una parola data la precedente:

$$P(W_i | W_1, W_2, \dots, W_{i-1}) \approx P(W_i | W_{i-1})$$

Esempio di generazione: *outside, new, car, parking, lot, of, the, agreement, reached*

- **N-gram**: nella pratica si generalizza a 3-grams, 4-grams o 5-grams, ovviamente più contesto si considera più le frasi generate hanno senso. Dal dataset va a capire quali sono le parole successive in base alle precedenti, quindi più parole precedenti tiene in considerazione più la parola successiva ha senso. In generale, gli N-grams hanno il limite delle **long-distance dependencies**, ovvero quando si scrive una frase ad un certo punto ci riferiamo ad un soggetto che sta molto prima, e questi modelli non riescono a catturare questa informazione:

"The computer(s) which I had just put into the machine room on the fifth floor is (are) **crashing**."

(**Crashing** è riferito al computer)

Il calcolo di queste probabilità viene fatto utilizzando la **Maximum Likelihood Estimate**:

$$P(w_i | w_{i-1}) = \frac{\text{count}(w_{i-1}, w_i)}{\text{count}(w_{i-1})}$$

Si calcolano le frequenze su un dataset, le probabilità sono il rapporto tra quante volte le parole compaiano nella sequenza in esame rispetto a quando compaiono da sola.

Esempio Maximum Likelihood Estimate:

Voglio la probabilità che w_i segue la w_{i-1} , è dato da quante volte compaiono insieme diviso quando la parola precedente compare da sola.

$$P(w_i | w_{i-1}) = \frac{c(w_{i-1}, w_i)}{c(w_{i-1})}$$

<s> I am Sam </s>
<s> Sam I am </s>
<s> I do not like green eggs and ham </s>

$$P(I | <s>) = \frac{2}{3} = .67 \quad P(Sam | <s>) = \frac{1}{3} = .33 \quad P(am | I) = \frac{2}{3} = .67$$

$$P(</s> | Sam) = \frac{1}{2} = 0.5 \quad P(Sam | am) = \frac{1}{2} = .5 \quad P(do | I) = \frac{1}{3} = .33$$

La probabilità che la frase inizia con "I" è 2/3 dato che il dataset ha solo 3 frasi.

Ottimale la probabilità che "Sam" segue "am" è ½ siccome "Sam" compare solo in due frasi.

Altro esempio Maximum Likelihood Estimate:

Se abbiamo un dataset più grande contenente più frasi, come ad esempio →

- can you tell me about any good cantonese restaurants close by
- mid priced thai food is what i'm looking for
- tell me about chez panisse
- can you give me a listing of the kinds of food that are available
- i'm looking for a good place to eat breakfast
- when is caffè venezia open during the day

Possiamo costruire una matrice di bigram →

Essa ci dice, ad esempio, quante volte "want" è preceduta da "i", che è 827 volte.

La *normalizzazione* con unigram è →

Ad esempio, quante volte compare "want", sarebbero i denominatori

La tabella risultante sarà →

	i	want	to	eat	chinese	food	lunch	spend
i	5	827	0	9	0	0	0	2
want	2	0	608	1	6	6	5	1
to	2	0	4	686	2	0	6	211
eat	0	0	2	0	16	2	42	0
chinese	1	0	0	0	0	82	1	0
food	15	0	15	0	1	4	0	0
lunch	2	0	0	0	0	1	0	0
spend	1	0	1	0	0	0	0	0

i	want	to	eat	chinese	food	lunch	spend
2533	927	2417	746	158	1093	341	278

	i	want	to	eat	chinese	food	lunch	spend
i	0.002	0.33	0	0.0036	0	0	0	0.00079
want	0.0022	0	0.66	0.0011	0.0065	0.0065	0.0054	0.0011
to	0.00083	0	0.0017	0.28	0.00083	0	0.0025	0.087
eat	0	0	0.0027	0	0.021	0.0027	0.056	0
chinese	0.0063	0	0	0	0	0.52	0.0063	0
food	0.014	0	0.014	0	0.00092	0.0037	0	0
lunch	0.0059	0	0	0	0	0.0029	0	0
spend	0.0036	0	0.0036	0	0	0	0	0

Tutto ciò sarebbe il **Language Model** per il Bigram. Questo modello può essere usato per calcolare:

$$P(<s> I want english food </s>) = P(I | <s>) \times P(want | I) \times P(english | want) \times P(food | english) \times P(</s> | food) = .000031$$

Possiamo osservare che "want chinese" è più probabile di "want english" (siccome si usa molto cibo cinese rispetto a cibo inglese), data dalla conoscenza del mondo.

Mentre "want to" a livello grammaticale in "to" segue quasi sempre il "want".

Invece, "spend want" è 0 siccome non si costruiscono frasi in questo modo.

$P(\text{english} | \text{want}) = .0011$ world
 $P(\text{chinese} | \text{want}) = .0065$
 $P(\text{to} | \text{want}) = .66$ grammar
 $P(\text{eat} | \text{to}) = .28$
 $P(\text{food} | \text{to}) = 0$ grammar (contingent zero)
 $P(\text{want} | \text{spend}) = 0$ grammar (structural zero)
 $P(i | <s>) = .25$

Da notare che i valori sono molto piccoli (essendo moltiplicazioni tra probabilità), per evitare di andare in **underflow** è quello di andare a fare una somma di logaritmi:

$$p_1 \times p_2 \times p_3 \times p_4 \Rightarrow \exp(\log p_1 + \log p_2 + \log p_3 + \log p_4)$$

VALUTARE IL LANGUAGE MODEL:

Bisogna verificare se la prossima parola generata è corretta per il contesto e si usano gli stessi approcci per i classificatori, su un **training-set** viene testato il language model per poi valutarlo sul **test-set**, andando a definire delle **metriche**.

La valutazione che viene fatta è chiamata **Extrinsic**, confrontando due modelli A e B. Vengono applicati entrambi in un contesto e successivamente vengono confrontati per dire quale dei due ha metriche migliori.

Fare il testing di modelli, però, è time-consuming, ma un language model può anche essere valutato in maniera approssimata usando una misura intrinseca, non facendo una valutazione esplicita con dei dati. Possiamo usare una misura chiamata **perplexity**, questo può essere usato senza usare dati di test.

Per capire cosa c'è dietro a questa misura vediamo *The Shannon Game* che consiste nel predire la prossima parola:

Vedendo il modello con che probabilità predice la parola corretta. L'unigram ha un risultato scadente, siccome esso non considera il contesto.

La formula della perplexity è →

Ridurre al minimo la perplessità equivale a massimizzare la probabilità.

I always order pizza with cheese and _____
The 33rd President of the US was _____
I saw a _____

mushrooms 0.1
pepperoni 0.1
anchovies 0.01
....
fried rice 0.0001
....
and 1e-100

$$PP(W) = P(w_1 w_2 \dots w_N)^{\frac{1}{N}}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \dots w_N)}}$$

$$\text{Chain rule: } PP(W) = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i | w_1 \dots w_{i-1})}}$$

$$\text{Bigrams: } PP(W) = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i | w_{i-1})}}$$

Esempio:

Se considero la generazione del prossimo numero su 10 numeri e tutte le estrazioni sono equiprobabili, ogni numero ha probabilità 1/10. Andando a calcolare la perplexity sarà 10, vuol dire che il modello tra 0 a 9 numeri non sa quale generare.

Altro esempio:

Se viene calcolata questa metrifica per un dataset del Wall Street Journal notiamo che aumentando il contesto la perplessità diminuisce (ha meno incertezze).

N-gram Order	Unigram	Bigram	Trigram
Perplexity	962	170	109

OVERFITTING:

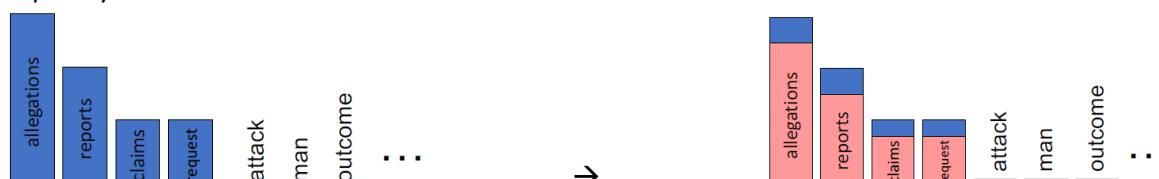
Gli N-grams funzionano bene quando si ha a disposizione una certa quantità di dati, però, nella realtà non si hanno mai abbastanza dati e vorremmo dei modelli che si generalizzano. Il problema è gli 0 nelle tabelle che vengono generate, dovute al fatto che determinate combinazioni non compaiono mai:

Training set:	Test set
... denied the allegations	... denied the offer
... denied the reports	... denied the loan
... denied the claims	
... denied the request	

→ $P(\text{"offer"} | \text{denied the}) = 0$

In tal caso non possiamo valutare la perplexity.

Per risolvere questi tipi di problemi si usa lo **smoothing**, non usando numeri precisi ma si smussano i valori così che togliamo un poco dalle altre probabilità e le diamo a quelle che stanno a 0, così da fare le divisioni per questi valori per calcolare la perplexity.



La tabella risultante rispetto alla precedente sarà:

$$c^*(w_{n-1} w_n) = \frac{[C(w_{n-1} w_n) + 1] \times C(w_{n-1})}{C(w_{n-1}) + V}$$

	i	want	to	eat	chinese	food	lunch	spend
i	3.8	527	0.64	6.4	0.64	0.64	0.64	1.9
want	1.2	0.39	238	0.78	2.7	2.7	2.3	0.78
to	1.9	0.63	3.1	430	1.9	0.63	4.4	133
eat	0.34	0.34	1	0.34	5.8	1	15	0.34
chinese	0.2	0.098	0.098	0.098	0.098	8.2	0.2	0.098
food	6.9	0.43	6.9	0.43	0.86	2.2	0.43	0.43
lunch	0.57	0.19	0.19	0.19	0.19	0.38	0.19	0.19
spend	0.32	0.16	0.32	0.16	0.16	0.16	0.16	0.16

24. TEXT CLASSIFICATION

Abbiamo visto che, un documento prima di essere analizzato bisogna effettuare un pre-processing (normalizzazione, ecc..). Successivamente, i **Language models** (N-grams) che stimano la probabilità che una certa sequenza di parola possa apparire, equivale a dire con che probabilità si può trovare una parola che segue n parole precedenti.

Adesso bisogna affrontare il **problema della classificazione**, cioè dato un documento associare una classe, catalogandolo in base ad una classe (problema multi-classe o classe binaria).

L'ambito del NLP è un caso importante perché molti problemi si riducono a problemi di classificazione, ad esempio i classificatori di e-mail di spam effettuano una **classificazione binaria** (spam o ham) analizzando il testo della mail. Un altro esempio è capire se un dato testo è stato scritto da un uomo o una donna. Oppure identificare una review positiva o negativa di un film. Tutti questi esempi sono classificatori binari. Un esempio di **classificatori multi-classe** è un classificatore che assegna categorie ad articoli scientifici, oppure identificare il linguaggio di un documento.

DEFINIZIONE DI TEXT CLASSIFICATION:

Abbiamo in input un documento e un insieme di classi, l'obiettivo è identificare la classe per il documento. Formalmente:

- **Input:** documento d e insieme di classi $C = \{c_1, c_2, \dots, c_J\}$;
- **Output:** classe prevista $c \in C$.

METODI DI CLASSIFICAZIONE:

Questo problema viene affrontato definendo delle **regole** conoscendo il problema, ad esempio conoscendo una data parola, o un link esterno, la posso usare per verificare se una e-mail è di spam se è presente al suo interno.

Questo algoritmo di classificazione ha un problema, siccome bisogna definire un dizionario di parole, richiede molto tempo, ma soprattutto le regole cambiano nel tempo, pertanto bisogna aggiornare periodicamente il dizionario, manutenzione complessa.

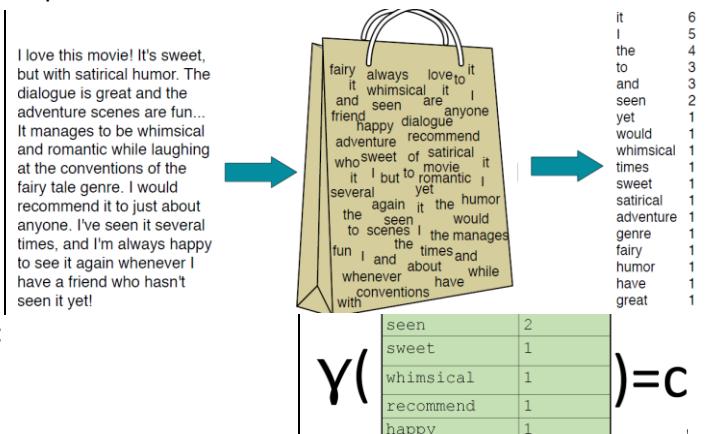
Bisogna costruire un classificatore che può essere addestrato con un training set ed utilizzarlo per classificare il documento, usando approcci supervisionati. Un classificatore del genere richiede meno sforzo siccome non vanno specificate tutte le regole e nel tempo, aggiornando il training set e riaddestrando la rete, si aggiorna in maniera automatica. Formalmente:

- **Input:** documento d , insieme di classi $C = \{c_1, c_2, \dots, c_J\}$ ed un training set di m documenti etichettati $(d_1, c_1), \dots, (d_m, c_m)$;
- **Output:** classificatore appreso $\gamma: d \rightarrow c$.

24.1 TEXT CLASSIFICATION CON NAIVE BAYES

Il metodo **Naive Bayes** è il più semplice. Abbiamo un albero con una unica radice e i figli sono condizionatamente dipendenti solo dal padre. L'assunzione che fa il modello Naive Bayes, per la classificazione del testo, è che il documento viene considerato come "bag of words", cioè come frequenza delle parole.

Se abbiamo un documento, non considera l'ordine delle parole presenti nel testo, siccome il metodo Naive Bayes suppone che le parole siano indipendenti, pertanto, rappresentiamo il testo come un insieme di parole. Successivamente, vengono contate le parole per quante volte appaiano nel testo, considerando la loro frequenza.



Il classificatore viene rappresentato da una funzione γ (gamma):

Andiamo a rappresentare il problema come problema Naive Bayes:

Dato un documento d e una classe c si vuole determinare per ogni classe la probabilità che la classe c è la classe del documento d .

Se sono due classi basta farlo solo per uno dato che l'altro sarà il suo complemento, mentre per multi-classe bisogna calcolare la probabilità per ogni classe, la probabilità più alta sarà la classe che associa al documento.

La probabilità della classe dato il documento, può essere scritta:

$$P(c|d) = \frac{P(d|c)P(c)}{P(d)}$$

(probabilità del documento data la classe * la probabilità della classe / la probabilità del documento)

Quello che fa il classificatore Naive Bayes restituisce la classe che ha la probabilità più alta. Pertanto, bisogna fare la massimizzazione della classe c che mi dà il valore massimo per quella probabilità.
Tutto ciò equivale a togliere il denominatore siccome è un fattore costante, perché la probabilità del documento è la stessa per tutte le classi.
La probabilità del documento la possiamo scrivere come la probabilità delle parole nel documento.

$$\begin{aligned}
 c_{MAP} &= \operatorname{argmax}_{c \in C} P(c | d) \\
 &= \operatorname{argmax}_{c \in C} \frac{P(d | c)P(c)}{P(d)} \quad \text{Bayes Rule} \\
 &= \operatorname{argmax}_{c \in C} P(d | c)P(c) \quad \text{Dropping the denominator} \\
 &= \operatorname{argmax}_{c \in C} P(x_1, x_2, \dots, x_n | c)P(c) \quad \text{Document } d \text{ represented as features } x_1, \dots, x_n \\
 &\text{O}(|X|^n \cdot |C|) \text{ parameters} \quad \text{How often does this class occur?} \\
 &\text{Could only be estimated if a very, very large number of training examples was available.} \quad \text{We can just count the relative frequencies in a corpus}
 \end{aligned}$$

Calcolare la probabilità di tutte le parole del documento richiede molto tempo, ma si può semplificare tutto ciò siccome Naive Bayes fa due assunzioni:

1. **Bag of Words**: la posizione della parola nel documento non è importante;
2. **Indipendenza condizionata**: assumiamo che la probabilità che due parole si presentino una dopo l'altra o si presentano nel documento, come il prodotto delle singole probabilità, $P(x_i | c_j)$.

$$P(x_1, \dots, x_n | c) = P(x_1 | c) \cdot P(x_2 | c) \cdot P(x_3 | c) \cdot \dots \cdot P(x_n | c)$$

Le formule diventano:

$$c_{MAP} = \operatorname{argmax}_{c \in C} P(x_1, x_2, \dots, x_n | c)P(c)$$

$$c_{NB} = \operatorname{argmax}_{c \in C} P(c) \prod_{x \in X} P(x | c)$$

Dove la produttoria prende tutte le parole del documento e calcola la probabilità di quella parola.

Supponiamo di avere un training set che abbia 100 review, 40 positive e 60 negative, la probabilità di una classe sarà le positive 40/100 e negativa 60/100.

Mentre la probabilità che una certa parola si presenti per una certa classe, si vede quanto è frequente quella parola per i documenti di quella classe.

Se si ha un aggettivo positivo nelle review positive, si conteggia quante volte quella proprietà si presenta nei documenti di quella classe diviso la somma di tutte le parole di quella classe.

Se ci troviamo 0 la probabilità precedente, la produttoria sarà 0:

Per risolvere si effettua lo smoothing che consiste nel togliere gli 0 addizionando l'elemento con 1. L'esempio di "fantastic" non uscirà più 0 ma si avvicinerà cosicché la produttoria non vada a 0.

$$\begin{aligned}
 \hat{P}(c_j) &= \frac{\text{doccount}(C = c_j)}{N_{\text{doc}}} \\
 \hat{P}(w_i | c_j) &= \frac{\text{count}(w_i, c_j)}{\sum_{w \in V} \text{count}(w, c_j)} \\
 \hat{P}(\text{"fantastic"} | \text{positive}) &= \frac{\text{count}(\text{"fantastic"}, \text{positive})}{\sum_{w \in V} \text{count}(w, \text{positive})} = 0 \\
 c_{MAP} &= \operatorname{argmax}_c \hat{P}(c) \prod_i \hat{P}(x_i | c) \\
 \hat{P}(w_i | c) &= \frac{\text{count}(w_i, c) + 1}{\sum_{w \in V} (\text{count}(w, c) + 1)} \\
 &= \frac{\text{count}(w_i, c) + 1}{\left(\sum_{w \in V} \text{count}(w, c) \right) + |V|}
 \end{aligned}$$

Per fare **apprendimento** si calcola:

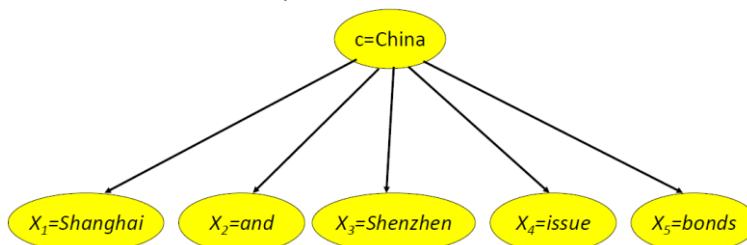
Per ogni classe ci andiamo a mettere tutti i documenti di una classe in un unico documento e ci calcoliamo la probabilità col rapporto tra i documenti di quella classe diviso il numero totale di documenti.

L'altra probabilità si prendono tutti i documenti della classe c_j e si calcola per ogni parola w_k il numero di occorrenze di w_k nel documento della classe c_j , dopodiché il rapporto è il numero di volte che è presente, si mette poi il fattore di smoothing.

$$\begin{aligned}
 P(c_j) &\leftarrow \frac{|\text{docs}_j|}{\text{total # documents}} \\
 P(w_k | c_j) &\leftarrow \frac{n_k + \alpha}{n + \alpha |\text{Vocabulary}|}
 \end{aligned}$$

MODELLO GENERATIVO PER NAIVE BAYES:

Per classificare il documento, i **modelli generativi** vanno a considerare quale è la classe che è più probabile che possa generare quel documento. Abbiamo un nodo in cui dipendono le altre variabili:



Il nodo radice rappresenta la classe ed i nodi figli sono le parole del documento, quindi quello che si va a determinare è la probabilità di ogni parola condizionata dalla classe ed è Naive perché le parole sono indipendenti.

Può essere considerata qualsiasi features estratte da un documento anziché la singola parola, ad esempio come nodo figlio mettiamo un dizionario di parole ritenute positive, pertanto invece di considerare tutte le parole del documento si considerano delle features. Se si considerano le parole il modello Naive Bayes è simile ai Language model.

Supponiamo di avere la classe *pos* ed avere una sequenza di parole che si vuole classificare.

Sotto ogni parola abbiamo a probabilità che questa appartiene alla classe *pos*. La probabilità della frase è data dalla produttoria delle probabilità data la classe. (corrisponde all'unigram model)

Model pos		Model neg	
0.1	I	0.2	I
0.1	love	0.001	love
0.01	this	0.01	this
0.05	fun	0.005	fun
0.1	film	0.1	film
I		love	this
0.1		0.1	0.01
0.2		0.001	0.01
		0.05	0.005
		0.1	0.1

$P(s | pos) = 0.0000005$
 $P(s|pos) > P(s|neg)$

Esempio:

Supponiamo di avere come training set 4 documenti, dove i primi 3 sono classificatori della lingua cinese e uno giapponese.

	Doc	Words	Class
Training	1	Chinese Beijing Chinese	c
	2	Chinese Chinese Shanghai	c
	3	Chinese Macao	c
	4	Tokyo Japan Chinese	j
Test	5	Chinese Chinese Chinese Tokyo Japan	?

Vogliamo calcolare le seguenti probabilità:

La probabilità che la classe sia cinese e giapponese è rispettivamente:

Le probabilità condizionate dobbiamo calcolare tutte le probabilità di ogni parola data la classe:

Vogliamo capire al documento 5 (d5) a quale classe appartiene.

Il 5° documento ha sia parole della classe cinese (c) che giapponese (j), bisogna calcolare la produttoria:

$$\hat{P}(c) = \frac{N_c}{N} \quad \hat{P}(w|c) = \frac{\text{count}(w,c)+1}{\text{count}(c)+|V|}$$

Priors:

$$P(c) = \frac{3}{4} \quad P(j) = \frac{1}{4}$$

Conditional Probabilities:

$$P(\text{Chinese}|c) = (5+1) / (8+6) = 6/14 = 3/7$$

$$P(\text{Tokyo}|c) = (0+1) / (8+6) = 1/14$$

$$P(\text{Japan}|c) = (0+1) / (8+6) = 1/14$$

$$P(\text{Chinese}|j) = (1+1) / (3+6) = 2/9$$

$$P(\text{Tokyo}|j) = (1+1) / (3+6) = 2/9$$

$$P(\text{Japan}|j) = (1+1) / (3+6) = 2/9$$

Choosing a class:

$$P(c|d5) \propto 3/4 * (3/7)^3 * 1/14 * 1/14 \\ \approx 0.0003$$

$$P(j|d5) \propto 1/4 * (2/9)^3 * 2/9 * 2/9 \\ \approx 0.0001$$

Questo modello così costruito è molto semplice, data la sua indipendenza tra i dati, e per scenari non molto complessi risulta essere efficiente. Infatti, può essere definito un algoritmo di spam filtering, andando ad estrarre delle features dal documento (esempio la parola "Viagra", farmacia online, cifre di soldi, codice HTML, riscatta ricompensa, ecc....).

METRICHE DI VALUTAZIONE:

Essendo un task di classificazione, per valutare il classificatore si usano determinate metriche. Si costruisce una **matrice di confusione** in cui si riportano gli errori, le classificazioni corrette rispetto a quelle errate.

Vogliamo riconoscere la classe coffee e 34 di questi sono state riconosciute, mentre altre 10 sono state classificate 3 come classe interest e 7 come trade.

Ci va a dire gli errori che ha fatto il classificatore.

Per fare una sintesi del modello si usano recall e precision, determinando i rapporti con la matrice.

Docs in test set	Assigned UK	Assigned poultry	Assigned wheat	Assigned coffee	Assigned interest	Assigned trade
True UK	95	1	13	0	1	0
True poultry	0	1	0	0	0	0
True wheat	10	90	0	1	0	0
True coffee	0	0	0	34	3	7
True interest	-	1	2	13	26	5
True trade	0	0	2	14	5	10

- Recall:** capacità del sistema nel recuperare tutti i risultati significativi, quanti ne ho classificati correttamente. Se si aumenta la recall si può incorrere in falsi positivi;

- Precision:** la capacità del sistema nel recuperare solo i risultati significativi.

$$\frac{\sum_{i=1}^n c_{ii}}{\sum_{i=1}^n c_{ij}} \quad o \quad \frac{\sum_{i=1}^n c_{ii}}{\sum_{j=1}^n c_{ji}}$$

Per calcolare queste metriche si effettuano facendo le somme per colonna o per riga. Per calcolare queste medie si può fare a due livelli:

- **Macro**: si effettuano recall e precision per tutte le classi e poi si esegue la media della precision e recall. Questo livello funziona bene quando abbiamo pochi elementi per ogni classe, se una certa classe ha molti valori può influenzare il valore delle altre classi;
- **Micro**: Si usa quando le classi non sono bilanciate con gli elementi, si va a quantizzare il numero di istanze (media pesata).

Class 1		Class 2		Micro Ave. Table	
	Truth: yes	Truth: no		Truth: yes	Truth: no
Classifier: yes	10	10	Classifier: yes	90	10
Classifier: no	10	970	Classifier: no	10	890
				Classifier: yes	100
				Classifier: no	20
					1860

- Macroaveraged precision: $(0.5 + 0.9)/2 = 0.7$
- Microaveraged precision: $100/120 = .83$

24.2 TEXT CLASSIFICATION CON REGRESSIONE LOGISTICA

Il problema è lo stesso, ovvero classificare documenti, ma lo affrontiamo in modo diverso, prendiamo il documento e affrontiamo il **problema in termini di features**. Il documento preso non è altro che un insieme di features e vogliamo determinare qual è la classe. Abbiamo sempre un problema supervisionato, quindi in input si ha tanti vettori di input e tante classi di classificazione.

Nell'esempio della mail di spam potremmo calcolare delle features, ad esempio X_1 = "numero di volte che la parola sale è presente nella mail" oppure X_2 = "numero di URL nella mail". Con l'approccio logistico si vuole andare ad associare dei pesi a queste features, tale che le features che caratterizzano la classe abbiano dei pesi più adatti per la classificazione di quel tipo di documento per quella classe.

Se la classificazione è binaria vogliamo che le features che identifica le mail di spam abbiano pesi più alti, le features che non identificano le mail di spam hanno pesi più bassi.

Supponiamo di avere: $x_i = 1$ if review contains 'awesome': w_i very positive +10

$x_j = 1$ if review contains 'abysmal': w_j very negative -10

$x_k = 1$ if review contains 'mediocre': w_k a little negative -2

L'obiettivo è, dato un vettore di input e dati dei pesi, predire la label di output più probabile per il vettore di features.

Formalmente:

- **Input**: documento d , insieme di classi $C = \{c_1, c_2, \dots, c_J\}$, un vettore di features $x = [x_1, x_2, \dots, x_n]$ e dei pesi (uno per features) $W = [w_1, w_2, \dots, w_n]$;

- **Output**: classificatore appreso $\hat{y} \in \{0, 1\}$ (se è multi-classe sarà $y \in \{0, 1, 2, 3, 4\}$).

Per ogni feature x_i , il peso w_i ci dice l'importanza di x_i , sommando tutte le features pesate con bias:

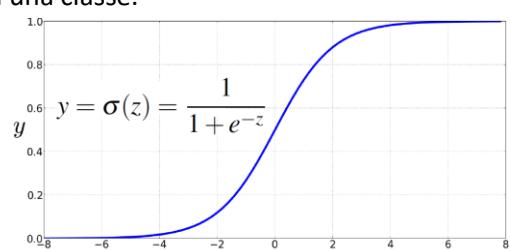
Se questa somma è alta, diciamo $y=1$; se è basso, allora $y=0$.

In realtà, si vuole calcolare la probabilità che il modello dia 1 dato il vettore dei features e il vettore dei pesi, pertanto, il calcolo non va bene, ma volgiamo sapere che probabilità appartenga ad una classe.

Possiamo trasformare i numeri in probabilità attraverso la **funzione sigmoid**:

$$\begin{aligned} z &= \left(\sum_{i=1}^n w_i x_i \right) + b \\ z &= w \cdot x + b \end{aligned}$$

Se il valore è 20, la funzione lo porta nel range 0-1, ottenendo la probabilità.



$$\begin{aligned} P(y=0) &= 1 - \sigma(w \cdot x + b) \\ &= 1 - \frac{1}{1 + e^{-(w \cdot x + b)}} \\ &= \frac{e^{-(w \cdot x + b)}}{1 + e^{-(w \cdot x + b)}} \\ \hat{y} &= \begin{cases} 1 & \text{if } P(y=1|x) > 0.5 \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Questo valore viene scelto a metà della curva. Se il valore ottenuto tramite funzione logistica è maggiore si \hat{y} allora gli assegno probabilità 1.

Esempio regressione logistica:

Consideriamo un task di sentiment analisi, vogliamo definire un classificatore che va a dire se questo testo si può considerare positivo o negativo.

Come primo passo definiamo le features, in questo caso usiamo un dizionario che ci dice quali sono le parole positive (gialle), negative (rosse) e altri tipi di parole (viola).

Supponiamo di aver estratto le feature ed avere il modello già allenato col vettore dei pesi, mi dice la feature quanto è importante rispetto alla classificazione.

Per capire il documento a quale delle due classi lo dobbiamo catalogare bisogna fare il calcolo del prodotto del peso per il vettore di features e otteniamo 0.833, lo diamo alla funzione logistica che restituisce 0.70. L'altra classe è il complemento. In tal caso la review è positiva.

Altro esempio regressione logistica:

Verificare se il punto è di fine frase oppure è un acronimo:

It's **shockey**. There are virtually **no surprises**, and the writing is **second-rate**. So why was it so **enjoyable**? For one thing, the cast is **great**. Another **nice** touch is the music **Divas** overcome with the urge to get off the couch and start dancing. It sucked **me** in, and it'll do the same to **you**.

Var	Definition	Value in Fig. 5.2
x_1	count(positive lexicon) \in doc	3
x_2	count(negative lexicon) \in doc	2
x_3	$\begin{cases} 1 & \text{if "no" } \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	1
x_4	count(1st and 2nd pronouns \in doc)	3
x_5	$\begin{cases} 1 & \text{if "!" } \in \text{doc} \\ 0 & \text{otherwise} \end{cases}$	0
x_6	log(word count of doc)	$\ln(66) = 4.19$

$$p(+|x) = P(Y = 1|x) = \sigma(w \cdot x + b)$$

$$= \sigma([2.5, -5.0, -1.2, 0.5, 2.0, 0.7] \cdot [3, 2, 1, 3, 0, 4.19] + 0.1)$$

$$= \sigma(.833)$$

$$= 0.70 \quad (5.6)$$

$$p(-|x) = P(Y = 0|x) = 1 - \sigma(w \cdot x + b)$$

$$= 0.30$$

CLASSIFICATORE DI REGRESSIONE LOGISTICA (BINARIA):

Given:

- a set of classes: (+ sentiment, - sentiment)
- a vector \mathbf{x} of features $[x_1, x_2, \dots, x_n]$
- $x_1 = \text{count("awesome")}$
- $x_2 = \log(\text{number of words in review})$
- A vector \mathbf{w} of weights $[w_1, w_2, \dots, w_n]$
- w_i for each feature f_i

$$P(y = 1) = \sigma(w \cdot x + b)$$

$$= \frac{1}{1 + e^{-(w \cdot x + b)}}$$

Il problema è da dove vengono i pesi che bisogna utilizzare per fare la classificazione? Dalla fase di training.

Bisogna apprendere il valore dei pesi, in maniera tale che il classificatore vada a minimizzare una funzione di loss, ovvero la differenza tra quello che classifica sul training set rispetto alle label corrette. Utilizziamo la cross-entropy e come algoritmo di training andiamo a minimizzare la discesa del gradiente. Fondamentalmente ci serve definire una loss.

Cross-entropy loss function:

$$\text{Maximize} \quad L(y, \hat{y}) = \log p(y|x) = \log[\hat{y}^y (1 - \hat{y})^{1-y}]$$

Durante la fase di training dobbiamo aggiornare i pesi θ in modo tale che la somma delle loss venga minimizzata in maniera iterativa seguendo la discesa del gradiente:

$$\hat{\theta} = \underset{\theta}{\operatorname{argmin}} \frac{1}{m} \sum_{i=1}^m L_{CE}(y^{(i)}, x^{(i)}; \theta)$$

DIFFERENZA TRA REGRESSIONE LOGISTICA E NAIVE BAYES:

Il **modello Naive Bayes** è un modello generativo perché va a dire la classe andando a comprendere qual è il modello che è più probabile che generi quel dato, al contrario, il **modello discriminativo (logistico)** va a capire le caratteristiche della classe in base i dati di training cosicché conosciute le caratteristiche che va a comprendere per scegliere tra una classe e l'altra.

Esempio:

Data una foto riconoscere se c'è un gatto o un cane.

I modelli Naive bayes cercano di capire come è fatto un gatto/cane da un generatore, mentre il modello logistico va a vedere cosa distingue un gatto da un cane (collare).

Se il problema è multi-classe bisogna utilizzare una funzione differente (softmax regression).



25. VECTOR SEMANTICS

Tutti i moderni modelli di Natural Language processing utilizzano come features le **Word Embeddings**.

Il problema è quello linguistico ed è relativo al **Word Meaning**, cioè quale è il significato delle parole. Quello fatto fino ad ora erano delle classificazioni che si basavano sui dati di training, ma in realtà un modello efficace e capace di analizzare del testo dovrebbe essere in grado anche di associare un significato alle parole.

Negli **N-grams** esprimiamo la probabilità che una data sequenza di parole appaia, le parole sono intese come una *sequenza di caratteri* e quindi c'è match 1v1 solo se le due parole sono identiche, pertanto, non abbiamo alcun significato associato alle parole.

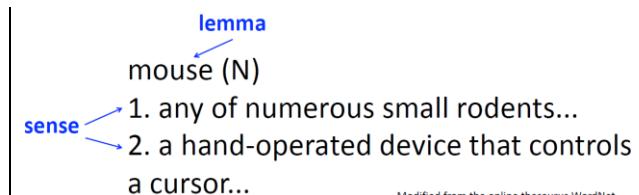
Quello che si vuole fare, a livello linguistico, è cercare di dare un **significato alle parole**. Il problema è, catturando il significato, riuscire a mettere in relazione le parole. Vogliamo essere in grado di capire quando due parole sono semanticamente legate tra loro.

RELAZIONI TRA PAROLE:

Nella realtà, se si vuole sapere il significato di una parola si utilizza il vocabolario cercando la parola, esempio cerchiamo "mouse":

Mouse sarebbe il **lemma**, quello che si trova dopo sono una **sequenza di significati** (stesso lemma ma più significati).

Nell'esempio "mouse" è sia il topo che il dispositivo del computer.



Modified from the online thesaurus WordNet

Altra cosa da considerare della linguistica è che le parole hanno tante relazioni tra loro, come quella di **sinonimia**. Esempio "big" e "large" sono due sinonimi, così come "auto" e "automobile".

Altra relazione è la **similarità**, parole che hanno un significato simile, come "auto" e "bicicletta" indicano oggetti differenti ma appartengono ad una stessa categoria (mezzi di trasporto).

Le parole possono appartenere ad un dato **dominio**, ad esempio "**hospitals**: surgeon, scalpel, nurse, anaesthetic, hospital" oppure "**restaurants**: waiter, menu, plate, food, menu, chef".

Ci sono i **contrari** (Antonymy) che hanno un significato opposto, ad esempio "dark" e "light" oppure "up" e "down".

Ci sono anche le **connotazioni** dei termini, una parola potrebbe essere vera ed una connotazione positiva come "happy".

Le parole potrebbero essere catalogate in base a tre dimensioni affettive:

- **valenza**: la gradevolezza dello stimolo;
- **eccitazione**: l'intensità dell'emozione provocata dallo stimolo;
- **dominanza**: il grado di controllo esercitato dallo stimolo.

	Word	Score		Word	Score
Valence	love	1.000		toxic	0.008
	happy	1.000		nightmare	0.005
Arousal	elated	0.960		mellow	0.069
	frenzy	0.965		napping	0.046
Dominance	powerful	0.991		weak	0.045
	leadership	0.983		empty	0.081

VECTOR SEMANTICS:

L'obiettivo è catturare il significato delle parole, ma le parole tra di loro sono legate da tanti **tipi di relazioni**, vorremmo uno strumento che sia in grado di identificare queste relazioni. Tutto ciò viene effettuato utilizzando un approccio basato su **vettori**. L'idea è quella di prendere un termine e rappresentarlo come un vettore, cosicché se abbiamo dei vettori nella rappresentazione matematica siamo in grado di capire se i due vettori sono simili tra loro, così da catturare le relazioni.

Tutti i modelli di NLP non usano la parola così come è ma la rappresentazione a vettori.

Per capire come catturare le relazioni, l'idea è definire l'uso di una parola considerando il loro ambiente. Non abbiamo un dizionario su cui cercare il significato di una parola bensì si dice che la parola ha quel significato perché viene usata in un dato ambito (l'uso dà il significato).

Zellig Harris (1954): Se A e B hanno ambienti quasi identici (stessi contesti) diciamo che essi sono sinonimi.

In altre parole, il significato di una parola dipende dal contesto in cui viene utilizzata, quindi se due parole compaiono nello stesso contesto hanno probabilmente significato simile.

Esempio:

Supponiamo di avere una parola "Ong choi", vogliamo scoprire il suo significato. Supponiamo di avere delle frasi con questa parola all'interno:

- *Ong choi is delicious sautéed with garlic.*
- *Ong choi is superb over rice*
- *Ong choi leaves with salty sauces*

Dal **contesto** si capisce che è un qualcosa che si cucina, ma ancora non si capisce che cosa è. Supponiamo di conoscere queste altre frasi:

- *...spinach sautéed with garlic over rice*
- *Chard stems and leaves are delicious*
- *Collard greens and other salty leafy greens*

In grassetto ci sono delle parole in comune, possiamo supporre che la nostra parola sia un vegetale (simile ai cavoli) perché gli ambienti dove è presente "Ong choi" sono simili all'ambiente degli spinaci e cavoli.

Infatti "Ong choi" è una pianta simile agli spinaci.

Quello appena fatto può essere ampliato, definendo il significato di una parola andando a vedere come la parola si distribuisce nel linguaggio. Andando a vedere i vicini (i contesti delle parole) cerchiamo di ricavarci le parole simili.

- **Idea 1:** definizione del significato in base alla distribuzione linguistica
- **Idea 2:** la parola diventa un punto nello spazio multidimensionale, un punto nello spazio possiamo vedere il suo intorno cosa c'è (troviamo i sinonimi).

Lo spazio sarà n dimensionale (non rappresentabile graficamente) ma se lo andiamo ad appiattire ci troviamo tutte le parole positive da un lato, quelle negative da un altro, le neutre ecc...



Il termine significato lo si rappresenta tramite gli "**embedding**". Tutti gli algoritmi di NLP fanno uso degli embedding come input per la fase di riconoscimento.

Come detto, meglio usare i vettori (embedding) siccome i modelli visti con le parole esatte sono inefficienti, bisogna ricavarci le relazioni tra le parole nel miglior modo. Se si hanno gli embedding abbiamo la possibilità di usare modelli già esistenti ed è possibile generalizzare a parole non miste (parole non presenti nel training set).

ALGORITMI PER L'EMBEDDING:

Il **TF-IDF** è il primo tipo di embeddings, va a considerare la frequenza dei termini, effettua un conteggio su quante volte una parola si presenta in un insieme di documenti ed è un **vettore sparso**, cioè ha tanti valori nulli, un esempio vettore lungo 1000 e il 90% sono 0.

Questo approccio non viene più usato siccome surclassato dai **vettori densi** che hanno taglia fissa e contengono pochi valori 0, implementato dal **Word2Vec**.

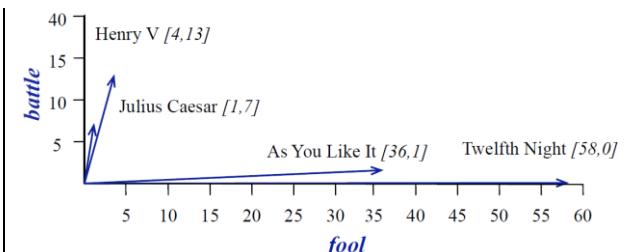
Esempio:

Supponiamo di avere quattro libri e costruire una matrice chiamata **Term-document matrix**, dove sulle colonne mettiamo i documenti e sulle righe abbiamo dei termini presenti in tutti i libri, riportiamo quante volte un termine è presente nel documento.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

La matrice ci dice la tipologia delle parole presenti nel documento, se presi in verticale possono essere visti come dei vettori. Andando ad analizzare il vettore per alcuni valori notiamo che ci sono dei libri che hanno la stessa quantità di termini.

La frequenza dei termini potrebbe essere utilizzata per capire le caratteristiche dei libri, ad esempio se prendiamo gli attributi "battle" e "fool", disegnando i vettori si nota che gli ultimi due libri hanno più termini "battle" pertanto dello stesso genere, mentre gli altri due libri hanno "fool" più alta.



I valori della matrice se visti in verticale, quindi, possiamo capire il genere dei libri o la loro similitudine, ma se presi in orizzontale possiamo vedere che "battle" è il tipo di parola che si presenta negli ultimi due libri mentre nei primi due no (o di meno). Per "fool" invece è il contrario.

	As You Like It	Twelfth Night	Julius Caesar	Henry V
battle	1	0	7	13
good	114	80	62	89
fool	36	58	1	4
wit	20	15	2	3

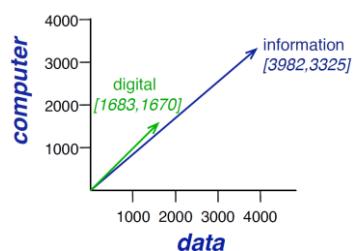
Questo tipo di matrice non ha molto senso, pertanto si costruisce un'altra matrice chiamata **term-context matrix**. Si vanno a mettere le parole sulle righe, mentre sulle colonne le parole di contesto (parole sono simili se hanno lo stesso contesto).

*is traditionally followed by cherry pie, a traditional dessert
often mixed, such as strawberry rhubarb pie. Apple pie
computer peripherals and personal digital assistants. These devices usually
a computer. This includes information available on the internet*

Nella tabella mettiamo ad esempio "digital" quante volte è presente insieme a una data parola del contesto.

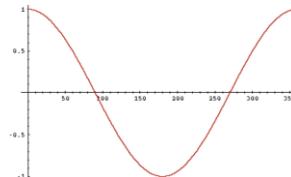
	aardvark	...	computer	data	result	pie	sugar
cherry	0	...	2	8	9	442	25
strawberry	0	...	0	0	1	60	19
digital	0	...	1670	1683	85	5	4
information	0	...	3325	3982	378	5	13

Se prendiamo dei valori e li rappresentiamo su un piano, notiamo che le due linee sono vicine. Vuol dire che i due valori presi (*information* e *digital*) si usano negli stessi contesti (*computer* e *data*).



Una volta ottenuti i vettori, per verificare che *information* e *digital* sono vicini calcoliamo l'angolo delle due rette calcolando la **Cosine as a similarity metric**. Più è basso l'angolo tra le rette più i due vettori sono simili.

- 1: vectors point in opposite directions
- +1: vectors point in same directions
- 0: vectors are orthogonal



TF-IDF:

Il problema di questo algoritmo è che ci sono dei termini che sono presenti in tanti contesti, esempio "the" o "it", ci serve un modo per abbassare il peso di questi valori per non falsare i risultati. Per bilanciare il tutto usiamo la misura **TF-IDF**, semplicemente si moltiplica Term frequency (TF) con un'altra misura chiamata inverse document frequency (IDF):

$$w_{t,d} = \text{tf}_{t,d} \times \text{idf}_t$$

La term frequency dice quante volte t è nel documento d . Per far sì che le parole molto frequenti abbiano un peso inferiore si calcola il **document frequency**, che ci dice quante volte la parola t è presente in quanti documenti, pertanto le parole comuni hanno un IDF molto alto (IDF di "the" sarà molto alta). Conoscendo la frequenza di queste parole possiamo calcolare l'**inverse document frequency (IDF)** che dirà quali sono le parole che compaiono in quel documento ma non negli altri, esempio "Romeo" in 10 documenti compare solo in uno allora l'idf sarà molto alto. Mentre la parola comune a più documenti avrà un idf molto basso.

$$\text{idf}_t = \log_{10} \left(\frac{N}{\text{df}_t} \right)$$

Word	df	idf
Romeo	1	1.57
salad	2	1.27
Falstaff	4	0.967
forest	12	0.489
battle	21	0.246
wit	34	0.037
fool	36	0.012
good	37	0
sweet	37	0

Il problema principale del TF-IDF è che la **dimensione dei vettori** è troppo grande siccome un contesto è formato da tutte le parole del vocabolario (dei documenti), in tal caso il vettore è uguale alla grandezza del vocabolario. L'altra problematica è che abbiamo molti termini a 0, siccome una parola avrà un contesto limitato.

WORD2VEC:

Per risolvere le problematiche della misura TF-IDF, anziché usare vettori sparsi (lunghi e pieni di 0), si utilizzano **vettori densi** dove si fissa una lunghezza e si cercano di costruire vettori più piccoli che generalizzano meglio (con pochi valori 0) riuscendo a catturare le relazioni tra i termini.

Questi vettori si utilizzano nella pratica e si chiamano **word2vec**. Questo tipo di vettore ha il vantaggio di essere molto veloce per l'addestramento, infatti, l'idea è di non fare conteggi (come fa TF-IDF) ma si cerca di determinare il significato di una parola facendo **previsioni**.

Per esempio, con *TF-IDF* calcoliamo quante volte una parola è vicino ad "*albicocca*", mentre, con *word2vec* si costruisce un classificatore che dice quanto è probabile che la parola sia vicina ad "*albicocca*" e lo si fa con un task di predizione binaria. L'idea che c'è dietro a questo algoritmo è il fatto che quello che si costruisce durante il training è un classificatore in cui i pesi del modello sono gli embeddings che andiamo a cercare, pertanto, il classificatore che viene costruito non serve ma teniamo il vettore dei pesi. Il task che interessa fare è sapere con che probabilità una parola appare insieme ad un'altra. Per costruire questo classificatore (per determinare che w si trova vicino ad un'altra parola) si prendono i documenti di input che dà il contesto delle parole, ad esempio, se si hanno dei documenti su cui si vuole eseguire questo task si danno i documenti come training set e si determina il contesto delle parole.

MODELLO SKIP-GRAM (WORD2VEC):

Si vuole allenare un classificatore binario (allenare a comprendere i contesti delle parole), per farlo ci servono contesti positivi (documenti di input) e negativi (vengono generati in maniera casuale, cioè prendiamo delle parole non di contesto). Il modello usa una regressione logistica, il task è binario e una volta allenato il modello, si prendono i pesi e questi corrispondono agli Embeddings, pesi simili significherà contesti simili mentre pesi differenti sono contesti differenti.

- **Training data:** abbiamo il parametro di contesto di 2 (cattura 2 parole prima e 2 dopo);
- **Addestrare un classificatore** a cui viene assegnata una coppia candidata (parola, contesto): coppie positiva (prese dai documenti) e coppie negativa (fuori contesto) prese in maniera casuale (basato sulle frequenze);
- E **assegna** a ciascuna coppia una **probabilità**: che descrive con quale probabilità la parola target t rientri nel contesto c .

...lemon, a [tablespoon of apricot jam, a] pinch...				
c1	c2	[target]	c3	c4
positive examples +		negative examples -		
t	c	t	c	
apricot	tablespoon	apricot	aardvark	apricot
	of	apricot	my	ever
	jam	apricot	where	apricot
	a	apricot	coaxial	apricot

$$P(+|t, c)$$

$$P(-|t, c) = 1 - P(+|t, c)$$

Vogliamo che la probabilità tra t e c sia alta quando sono nello stesso contesto, per calcolare questa probabilità si effettua il prodotto dei vettori di t e di c che sarà un valore alto se si trovano nello stesso contesto.

Il problema è che il prodotto va da $-\infty$ a $+\infty$ e non un valore di probabilità, pertanto, si normalizza usando la funzione logistica. Applicando la funzione sigmoide si riesce a portare il valore del prodotto nel range $[0; 1]$.

$$P(+|w, c) = \sigma(c \cdot w) = \frac{1}{1 + \exp(-c \cdot w)}$$

$$P(-|w, c) = 1 - P(+|w, c)$$

$$= \sigma(-c \cdot w) = \frac{1}{1 + \exp(c \cdot w)}$$

Un'altra cosa che considera il modello skip-gram è che quando si calcola la parola rispetto al contesto lo si considera come un problema indipendente. Se abbiamo la probabilità che una parola target sia di contesto a c_1, \dots, c_k equivale a calcolarlo singolarmente per c_1, \dots, c_k e pertanto è la produttoria delle probabilità singole (lo si può passare in sommatoria usando i logaritmi):

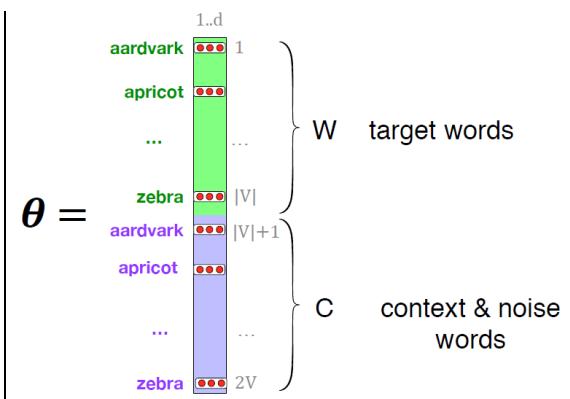
$$P(+|w, c_{1:L}) = \prod_{i=1}^L \sigma(c_i \cdot w)$$

$$\log P(+|w, c_{1:L}) = \sum_{i=1}^L \log \sigma(c_i \cdot w)$$

Riassumendo, abbiamo ridotto il problema di determinare se il target e un contesto sono legati tra loro ad un problema di costruire un classificatore logistico che va a calcolare queste probabilità.

Adesso bisogna implementare il classificatore. Verrà costruito un classificatore che andrà a determinare due insiemi di pesi, cioè il vettore di pesi ha dimensione $2V$, perché saranno pesi per le parole target e i pesi legati al contesto.

Quello che si andrà a fare è costruire tutto il modello, si fa training, il vettore dei pesi, ad esempio per "apricot" ci sarà un vettore di dimensione d , questo vettore è l'embeddings di apricot.



Una volta ottenuti i dati di training, costruiti a partire da un documento, si **costruisce un classificatore** fissando una certa dimensione ai vettori, ad esempio 300 (e pertanto anche gli Embeddings avranno la stessa dimensione). Una volta ottenuto un vettore dei pesi, ad esempio $300 \times V$ (per ogni parola abbiamo un vettore di 300), che vogliamo allenare e all'inizio si parte con valori casuali oppure da 0.

Vogliamo che il vettore dei pesi faccia in modo che quando gli diamo una parola e un contesto positivo allora il valore di probabilità alto, mentre quando gli diamo una parola e un contesto negativo la probabilità deve essere minimizzata.

In maniera iterativa, si danno in input coppie parola e contesto positivo, si ottiene un output, si calcola la loss e si aggiornano i pesi affinché questa probabilità sia incrementata.

In termini matematici, quando si calcola la loss quello che si vuole massimizzare corrisponde alla somma:

Se prendiamo tutte le coppie parole e contesti positivi, vogliamo massimizzare, così come le coppie parole e contesti negativi. Cioè indovinare se il contesto è positivo o negativo. Le parole negative sono k rispetto a quelle positive.

$$\begin{aligned} L_{CE} &= - \left[\log \sigma(c_{pos} \cdot w) + \sum_{i=1}^k \log \sigma(-c_{neg_i} \cdot w) \right] \\ \frac{\partial L_{CE}}{\partial c_{pos}} &= [\sigma(c_{pos} \cdot w) - 1]w \\ \frac{\partial L_{CE}}{\partial c_{neg}} &= [\sigma(c_{neg} \cdot w)]w \\ \frac{\partial L_{CE}}{\partial w} &= [\sigma(c_{pos} \cdot w) - 1]c_{pos} + \sum_{i=1}^k [\sigma(c_{neg_i} \cdot w)]c_{neg_i} \end{aligned}$$

L'idea è che andare a minimizzare l'errore in cui nel caso della probabilità del contesto negativo e della classe negativa equivale a fare $1 - \text{la probabilità del contesto negativo nella classe dei positivi}$:

$$\begin{aligned} c_{pos}^{t+1} &= c_{pos}^t - \eta [\sigma(c_{pos}^t \cdot w^t) - 1]w^t \\ c_{neg}^{t+1} &= c_{neg}^t - \eta [\sigma(c_{neg}^t \cdot w^t)]w^t \\ w^{t+1} &= w^t - \eta \left[[\sigma(c_{pos} \cdot w^t) - 1]c_{pos} + \sum_{i=1}^k [\sigma(c_{neg_i} \cdot w^t)]c_{neg_i} \right] \end{aligned}$$

Abbiamo due vettori, sia i pesi che il vettore c del contesto. L'algoritmo quando andrà a fare il training cercherà di aggiornare questi pesi, sia di c che di w , affinché la loss venga minimizzata.

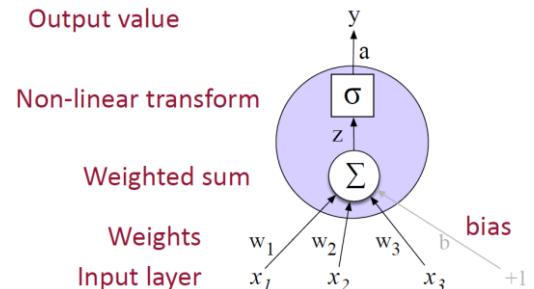
Quello che si apprende non è solo una matrice dei pesi ma sono due, quella di w e di c . Alla fine di tutto possiamo o considerare solo w , o le possiamo fondere.

Un parametro importante è il **contesto**, nel nostro caso ci dice quante parole vedere prima e dopo rispetto alla parola target. Più si aumenta il contesto e più semantica riusciamo a cogliere della parola, siccome si prendono parole molto distanti.

26. NEURAL NETWORKS (NLP)

NEURAL NETWORK UNIT:

L'elemento principale di una rete neurale è formato da un nodo che fa la somma pesata delle features, aggiunge un bias e poi applica una funzione di attivazione.



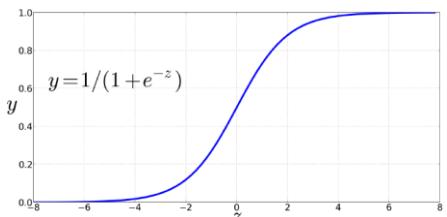
In termini matematici dobbiamo calcolare z , ovvero il prodotto tra il vettore dei pesi e vettore delle features e con la sigmoide riusciamo a determinare il valore tra 0 e 1:

$$z = b + \sum_i w_i x_i$$

Sigmoid

$$z = w \cdot x + b$$

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Funzione finale che l'unità sta calcolando:

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + \exp(-(w \cdot x + b))}$$

Esempio:

Suppose a unit has:

$$w = [0.2, 0.3, 0.9]$$

$$b = 0.5$$

What happens with input x :

$$x = [0.5, 0.6, 0.1]$$

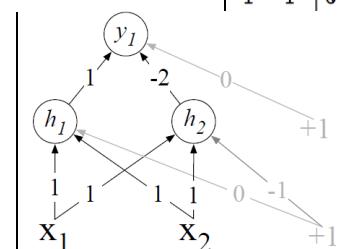
$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} = \frac{1}{1 + e^{-(.5*.2+0.6*.3+0.1*.9+0.5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

PROBLEMA DELLO XOR:

Una rete neurale semplice ha tanti limiti, uno dei quali è lo XOR siccome non si riesce a modellarlo perché la funzione va a dividere con una retta lo spazio e la funzione XOR non è in grado di andare a definire un classificatore appropriato.

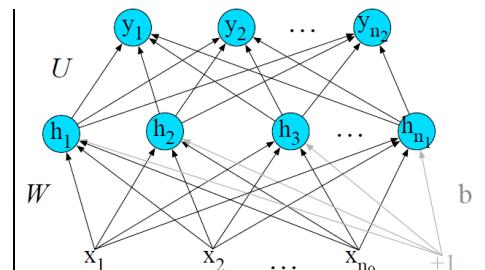
XOR		
x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

Per risolvere il problema si usano più layer, aggiungendo uno strato di nodi hidden con i vari valori associati ai pesi.



FEEDFORWARD NEURAL NETWORKS:

Possiamo usare le reti **Feedforward** che a partire da un vettore di features, vanno attraverso degli strati intermedi (hidden) a fornire degli input agli strati finali che restituiscono l'output della classificazione.



Funzionano bene questi modelli perché i vari layer vanno a catturare caratteristiche particolari delle features. Anziché fare una analisi diretta sulla features selezionata, l'idea è far apprendere le features rilevanti creando tanti strati che sono in grado di catturare gli aspetti importanti delle features di input.

Se consideriamo una rete a due layer, con un problema di classificazione, cioè abbiamo un unico nodo di output.

L'hidden layer fa il calcolo, in questo caso abbiamo una matrice dei pesi W * il vettore delle features x + il bias, e poi abbiamo un vettore dei pesi U che moltiplichiamo per l'hidden layer e applicando una funzione di attivazione otteniamo l'output.

FEEDFORWARD NEURAL NETWORKS PER TASK DI NLP:

Esempi di applicazione della rete **Feedforward** per **task NLP** sono:

- **Text classification**: determinare la classe di un certo documento;
- **Language modeling**: dedurre la probabilità di una certa sequenza di parole, in modo equivalente, probabilità della prossima parola rispetto alle precedenti.

Per questi task le reti Feedforward si comportano bene, ad esempio in un task di text classification che corrisponde alla Sentimental Analysis. Per fare ciò, bisogna costruire il vettore delle features, successivamente si dà il documento e se è una review deve dire se è positiva o negativa.

Esempi di features possono essere:

- Contare le parole positive secondo un certo dizionario;
- Contare le parole negative secondo un certo dizionario;
- Presenza della parola "no";
- Pronomi prima e seconda persona;
- Presenza di punti esclamativi "!";
- Lunghezza del documento.

Si prende il documento, si calcolano le 6 features e si dà tutto in input al classificatore e se è già allenato dirà qual è il sentiment del documento.

Utilizzando una rete a 2 layer, otterremo:

Le foglie saranno le features estratte, uno strato hidden e la radice come valore di output.

Il vantaggio di una rete del genere è che gli strati intermedi cercano di catturare features utili alla classificazione.

NEURAL NET CLASSIFICATION CON EMBEDDINGS:

Visto che abbiamo utilizzato gli **embeddings**, possiamo costruirci una rete neurale per andare a costruire un **Language Model**, un modello che ci dice la probabilità di una certa parola date le precedenti.

Per un task del genere, non si parte con un vettore di features, per predire la prossima parola serve conoscere il significato delle parole, pertanto, si usano gli embeddings.

Se il task è "date le ultime 3 parole dimmi la parola successiva" per ogni parola si va a prendere le embeddings e si utilizza come primo strato della rete neurale.

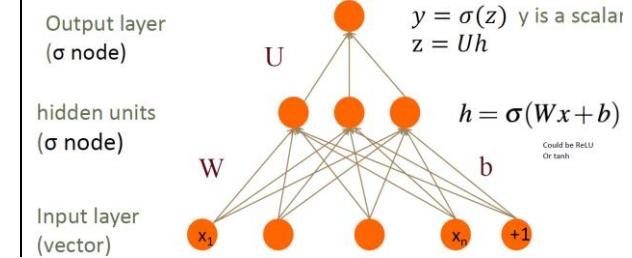
Per quanto riguarda come andare a rappresentare le parole viene usata una codifica chiamata **onehot encoder**, cioè si vanno a costruire dei vettori di lunghezza V dove c'è un unico 1 in corrispondenza della parola, cioè abbiamo V parole ognuna occuperà una posizione nell'array.

Nell'esempio, la parola "the" la si rappresenta con il vettore in cui nella posizione 451 c'è 1 e in tutte le altre posizioni c'è 0.

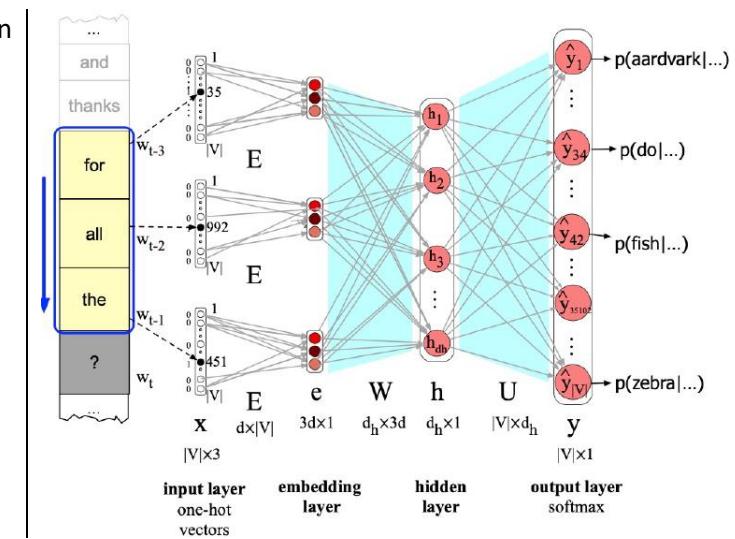
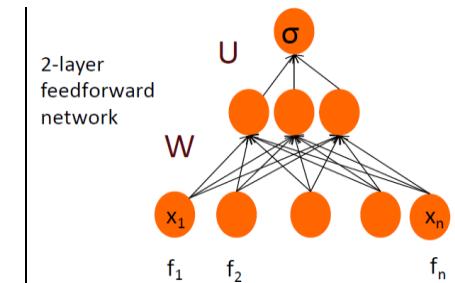
Il vettore one-hot mi permette che tramite il prodotto, con la matrice degli embeddings, si riesce ad ottenere il suo embeddings in maniera diretta siccome otteniamo, moltiplicando per 1, la sua riga.

Dal one-hot vectors si ottengono gli embeddings e questi saranno l'input per gli strati successivi.

In questo caso, poiché vogliamo predire una parola, abbiamo un problema multi-classe, cioè la rete predice la probabilità che ognuna delle V parole sia la prossima parola date le 3 precedenti, e verrà selezionata la parola con probabilità più alta.



Var	Definition
x_1	count(positive lexicon) ∈ doc
x_2	count(negative lexicon) ∈ doc
x_3	{ 1 if "no" ∈ doc 0 otherwise}
x_4	count(1st and 2nd pronouns ∈ doc)
x_5	{ 1 if "!" ∈ doc 0 otherwise}
x_6	log(word count of doc)



Dal punto di vista pratico, un Language Model del genere è più performante degli **N-grams**, uno dei motivi è che si usano gli embeddings anziché il conteggio su parole esatte (questo richiede molti dati di training).

Un problema di questo Language Model è che la **taglia è fissa**, cioè il contesto considerato è di lunghezza fissa (nell'esempio è 3) ma nella realtà non è possibile siccome si considerano tante parole precedenti per capire il significato.

RETI NEURALI RICORRENTI (RNN):

Una soluzione che va a migliorare il concetto del contesto per reti Feedforward sono le **reti neurali ricorrenti**, andando a catturare l'aspetto temporale, ricorrenti perché considerano tutti gli eventi passati (input dati alla rete). Nel caso del testo, con la dipendenza temporale, le reti neurali ricorrenti sono in grado di andare a memorizzare.

L'idea è che quando un hidden layer produce un output, questo viene dato sia allo strato successivo ma anche all'hidden layer successivo.

Questo tipo di architettura ha memoria perché quando fa il calcolo dell'output va a considerare tutto ciò che è accaduto in precedenza, **superando il limite della lunghezza fissa**.

L'architettura è la seguente:

Se considerato all'istante t , quando si andrà a moltiplicare il vettore x_t col vettore dei pesi W , quando si calcola il valore dell'hidden layer si utilizza sia questo calcolo fatto in passato ma anche l'output dell'istante precedente dell'hidden layer h_{t-1} .

Quindi si moltiplica l'output precedente per un'altra matrice chiamata matrice U .

In termini algoritmici:

Applichiamo il prodotto (matrice dei pesi W * vettore features x_i) e gli aggiungiamo la storia passata (valore di output del layer hidden all'istante precedente). Successivamente si applica la funzione g per ottenere il valore h_i , usato sia come output sia come h dell'istante successivo.

Un problema con queste reti è il **training** e il **gradiente**.

Proprio come con le reti feedforward, utilizzeremo un set di addestramento, una funzione di perdita (loss) e una propagazione all'indietro per ottenere i gradienti necessari per regolare i pesi in un RNN. I pesi che dobbiamo aggiornare sono:

- W – i pesi dal livello di input al livello nascosto;
- U – i pesi dal livello nascosto precedente al livello nascosto corrente;
- V – i pesi dal livello nascosto al livello di output.

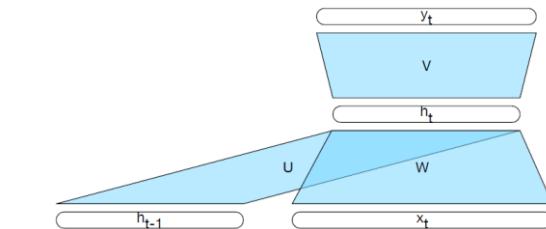
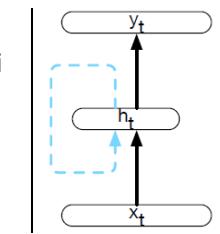
Quando si effettuano i calcoli, con la back propagation bisogna calcolare le derivate del layer corrente considerando le derivate del layer precedente. Bisogna tornare fino all'origine per determinare come aggiornare il peso.

Quindi c'è il problema del gradiente che tramite prodotto delle derivate può o azzerarsi oppure crescere e rimane esponenziale.

Esistono varie tecniche per affrontare il problema, una di queste è passare alle reti **LSTM**.

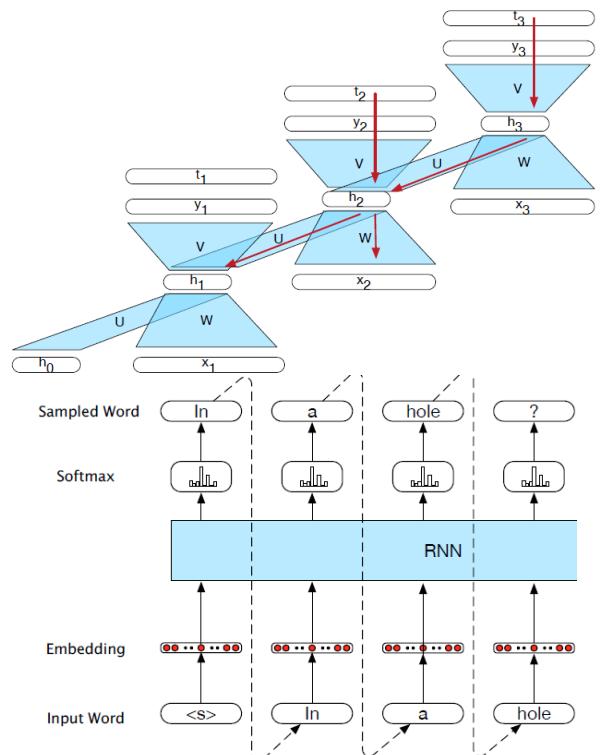
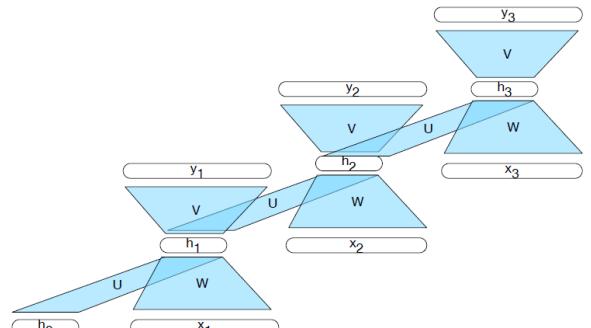
Le RNN si utilizzano per task di NLP in diversi modi, come la **generazione della prossima parola**. Una volta allenata la rete gli possiamo dare in input una frase oppure chiedere di dare la prossima parola.

Ad esempio, richiedendo di iniziale una frase e ci genera "In", chiedendo la prossima e così via. Essa tiene in considerazione tutta la storia passata.



```
function FORWARDRNN( $x, network$ ) returns output sequence  $y$ 
```

```
 $h_0 \leftarrow 0$ 
for  $i \leftarrow 1$  to LENGTH( $x$ ) do
   $h_i \leftarrow g(U h_{i-1} + W x_i)$ 
   $y_i \leftarrow f(V h_i)$ 
return  $y$ 
```



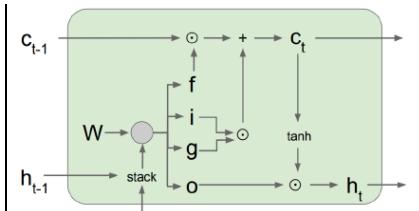
LONG SHORT-TERM MEMORY (LSTM):

L'idea delle **LSTM** è di non usare più le moltiplicazioni siccome azzerano il gradiente, ma utilizzare la **somma**. Un'altra idea è usare i **"gates"** che sono implementate sempre tramite delle funzioni ma che permettono di far avanzare informazioni rilevanti (di contesto) agli strati successivi senza perdita di dati.

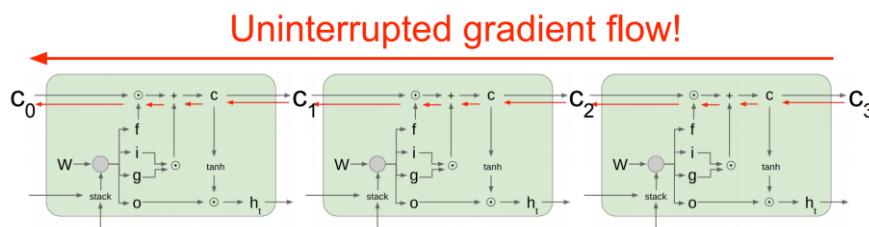
Graficamente, abbiamo che quando analizziamo la features all'istante t , abbiamo l'hidden layer precedente (come RNN) ma anche una c che sarebbe il contesto. Queste informazioni passano attraverso dei gates e danno in output il contesto successivo e l'hidden layer successivo.

I gate sono 3:

Il ***forget gate*** dice cosa devo togliere dal contesto, portando avanti solo le informazioni rilevanti e dimenticando quelle inutili.

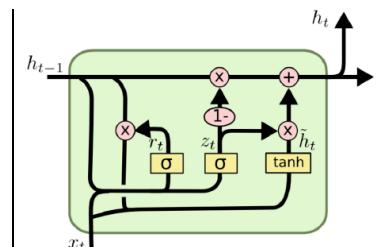


- Input gate (**how much to write**): $i_t = \sigma(W^i h_{t-1} + U^i x_t + b^i) \in \mathbb{R}^h$
- Forget gate (**how much to erase**): $f_t = \sigma(W^f h_{t-1} + U^f x_t + b^f) \in \mathbb{R}^h$
- Output gate (**how much to reveal**): $o_t = \sigma(W^o h_{t-1} + U^o x_t + b^o) \in \mathbb{R}^h$



GATED RECURRENT UNITS (GRU):

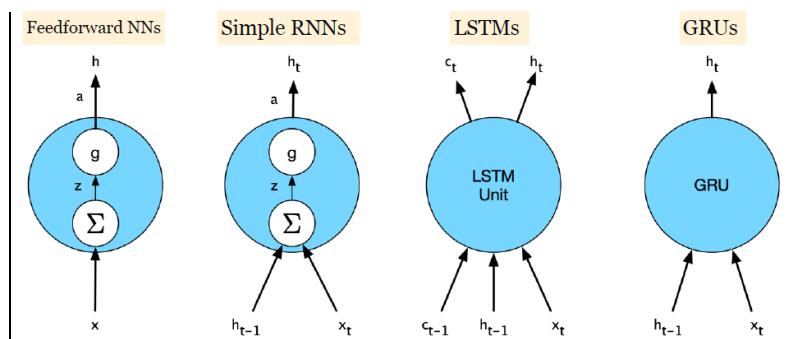
Un'altra versione è la GRU dove i gates passano da tre a due:



- Reset gate: $r_t = \sigma(W^r h_{t-1} + U^r x_t + b^r)$
- Update gate: $z_t = \sigma(W^z h_{t-1} + U^z x_t + b^z)$

COMPARAZIONE:

Le LSTM rispetto alle RNN hanno molti più parametri, si hanno più matrici da apprendere.



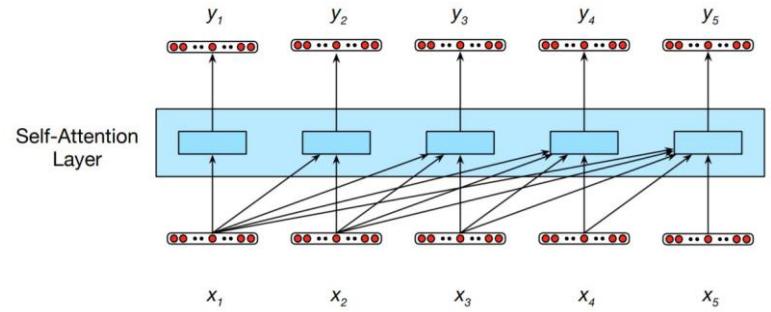
SELF-ATTENTION LAYERS - TRANSFORMERS:

Sia RNN che LSTM hanno difficoltà a far passare informazioni in diversi strati, cioè nel LSTM possiamo decidere cosa far andare avanti però c'è comunque la possibilità che alcune **informazioni vengano perse**. Un altro limite di queste reti è la **sequenzialità**, cioè quando si calcola il risultato per un certo valore di input dobbiamo calcolare prima i valori dalle features precedenti, in pratica i calcoli sono fatti in maniera sequenziale e non è possibile sfruttare il parallelismo.

Questi due limiti sono superati da un altro tipo di architettura che si chiama **Transformers** che è sia in grado di non perdere informazioni di contesto sia permettere la parallelizzazione.

Quello che fanno è mappare un vettore x di n input in un vettore y della stessa dimensione.

La componente principale della rete è la **Self-Attention** che è quella che va a considerare il contesto delle parole, andando a decidere ciò che è rilevante.



L'idea è che ogni nodo del **Self-Attention** prende in input tutte le parole precedenti, il primo nodo ha solo x_1 e produce solo y_1 , mentre il secondo prende x_2 e il precedente x_1 per produrre y_2 e così via...

In questo modo si è risolto il problema di perdita di informazione quando si va ad apprendere. Quando si analizza un elemento si va a considerare x_i con tutto ciò che lo precede.

Per capire il legame che esiste tra una parola x_i con un x_{i-k} si fa il prodotto dei due vettori e questo ci dice se sono simili. Questa misura si chiama **score**:

$$\text{score}(x_i, x_j) = x_i \cdot x_j$$

Anche qui il problema è che il valore di score è tra $-\infty$ e $+\infty$, pertanto si va a normalizzare con la funzione **softmax** ottenendo un valore chiamato α_{ij} (sarà un peso):

$$\begin{aligned} \alpha_{ij} &= \text{softmax}(\text{score}(x_i, x_j)) \quad \forall j \leq i \\ &= \frac{\exp(\text{score}(x_i, x_j))}{\sum_{k=1}^i \exp(\text{score}(x_i, x_k))} \quad \forall j \leq i \end{aligned}$$

Che dice quanto è importante la parola, cioè se x_j è importante per x_i questo valore sarà alto. Facendo il prodotto con x_i e i precedenti, ci calcoliamo tutti i pesi α_{ij} . Successivamente, moltiplichiamo tutti i pesi α_{ij} con il vettore x , ottenendo il vettore di output y_i :

$$y_i = \sum_{j \leq i} \alpha_{ij} x_j$$

(Somma pesata di tutti gli x precedenti rispetto a quanto sono importanti rispetto a x_i)

Dobbiamo rappresentare tutto ciò come **problema di apprendimento**. L'idea è di aggiungere dei parametri, sottoforma di pesi, che ci vanno a rappresentare l'importanza di un vettore di features rispetto all'input.

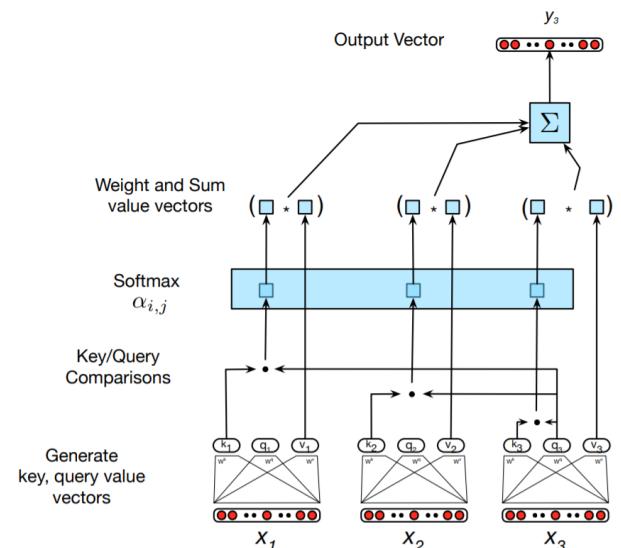
Ogni volta che un vettore x_i viene considerato, ha un ruolo quando viene usato per valutare l'output y_i , ma ha un ruolo differente quando viene usato per valutare y_{i+k} . Pertanto, i pesi che si vanno a determinare dipendono dal differente ruolo che l'embeddings di input svolge nei calcoli e questi ruoli sono 3:

- **Query**: nel caso in cui l'embeddings è la query, cioè è x_i quando si calcola y_i ;
- **Key**: può svolgere un ruolo di chiave, quando viene usato come contesto;
- **Value**: come valore di output, quando viene considerato come focus dell'attention.

Se prendiamo l'embeddings x_3 e viene usato per fare il confronto con gli altri embeddings, in questo caso bisogna utilizzare il vettore di query, quindi confronterò q_3 con k_3, k_2 e k_1 .

L'embeddings avrà un peso differente a seconda del ruolo che ha nei calcoli. Tutti questi pesi sono memorizzati in matrici differenti che chiamiamo V , Q e K .

$$Q = W^Q X; K = W^K X; V = W^V X$$



Per determinare l'output si usa:

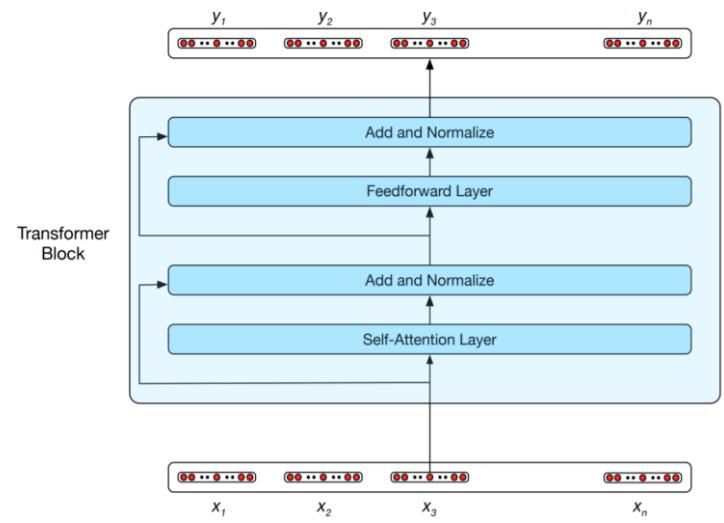
$$\text{SelfAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

(softmax di $K^T Q$ / fattore di normalizzazione * la matrice dei pesi V)

BLOCCO TRANSFORMER:

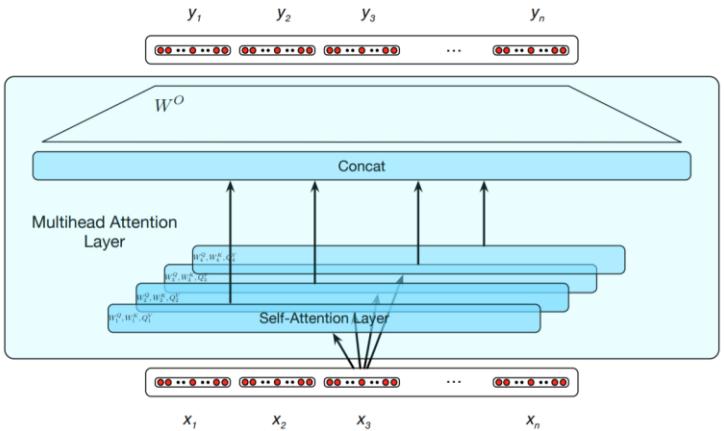
La struttura è la seguente:

Il vettore di embeddings x viene dato in input al layer di Self-Attention, producendo un vettore y della stessa dimensione, dopodiché vengono fatte delle operazioni di normalizzazione e poi c'è un modulo di Feedforward.



Un aspetto importante con questa architettura è il fatto che i calcoli, che effettuano i blocchi interni al Self-Attention Layer, possono essere effettuati in parallelo.

Un altro aspetto è che si utilizzano i Multihead Self-Attention, dove possiamo considerare diverse relazioni tra le parole creando dei legami sintattici/semantici e tanto altro. Per fare tutto ciò è crearsi tanti di questi transformer ed ognuno va a catturare un aspetto della frase per poi concatenare i risultati di ogni head.

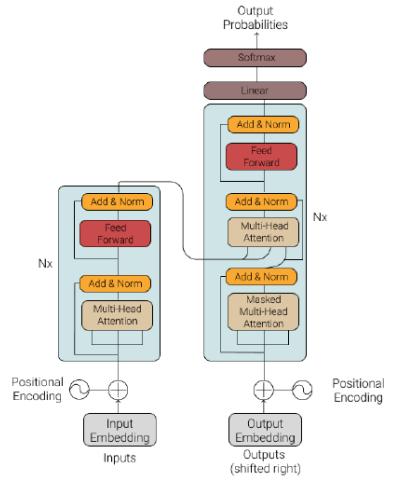


Un limite di questo tipo di architettura è il fatto della sequenza, cioè l'ordine delle parole. Se si cambia l'ordine degli input otteniamo sempre lo stesso risultato, la posizione della parola nella frase si perde. Questo può essere risolto con le **Positional Embeddings**, cioè andare ad associare un numero assieme alla parola, quello che si vuole fare è aggiungere questa informazione sequenziale alle parole.

I transformers è l'architettura più in voga in questi anni, sono utilizzati per la generazione della prossima parola o completamento del testo.

27. ENCODER-DECODER BERT-GPT2-T5

Nel 2017, i ricercatori di Google hanno pubblicato un paper in cui proponevano una nuova architettura di rete neurale per la modellazione di sequenze: **Transformer**. Tale architettura utilizza l'**attention** per aumentare la velocità con cui i modelli possono essere addestrati. Inoltre, combinando l'architettura Transformer con tecniche di apprendimento non supervisionato, si elimina la necessità di addestrare un modello da zero.



TOKENIZZAZIONE:

I modelli basati su Transformer non possono ricevere come input stringhe grezze, ma presuppongono che il testo sia stato **tokenizzato** e codificato in vettori numerici.

La tokenizzazione è il processo di suddivisione di una stringa in unità atomiche, denominate **token**, e viene fatto a diversi livelli:

- **Character-level**: Suddivisione di una stringa in una lista di singoli caratteri:

```
text = "Tokenizing text is a core task of NLP."
tokenized_text = list(text)
print(tokenized_text)
['T', 'o', 'k', 'e', 'n', 't', 'i', 's', ' ', 'a', ' ', 'c', 'o', 'r', 'e', ' ', 't', 'a', 's', 'k', ' ', 'o', 'f', ' ', 'N', 'L', 'P', '.']
```

Successivamente, i modelli accettano vettori numerici e non caratteri, pertanto si passa alla **Numericalization**, associando ogni token (carattere) un ID come il **One-hot encoding**:

```
Token: T
Tensor index: 5
One-hot: tensor([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Il **vantaggio** è che è utile in presenza di errori ortografici e parole rare, mentre lo **svantaggio** è che ignora qualsiasi struttura del testo e tratta l'intera stringa come un flusso di caratteri.

- **Word-level**: Piuttosto che suddividere il testo in caratteri, esso viene suddiviso in parole, ciascuna delle quali viene mappata in un intero:

```
['Tokenizing', 'text', 'is', 'a', 'core', 'task', 'of', 'NLP.']}
```

Anche in questo caso viene fatto il **Numericalization One-hot encoding**, ma associando un ID all'intera parola.

Il **vantaggio** è che l'utilizzo di parole fin dall'inizio permette ad un modello di evitare l'apprendimento delle parole dai caratteri, riducendo così la complessità del processo di addestramento. Mentre lo **svantaggio** è che la dimensione del vocabolario può essere molto grande (a causa di errori ortografici o parole rare).

- **Subword-level**: È un buon compromesso tra la tokenizzazione **character level** e **word-level**, combinando i migliori aspetti dei due approcci. Suddivide parole rare in unità più piccole per permettere al modello di considerare le parole frequenti come entità uniche e quindi di limitare la dimensione degli input. Viene definita in base al corpus ed è solitamente basata su regole statistiche e algoritmi, uno di questi è il **Byte Pair Encoding (BPE)** che è un approccio Greedy perché fa sempre la scelta migliore e va a combinare ad ogni iterazione quei due token che sono più frequenti.

Esempio BPE:

- Considerando un dizionario di parole (usate nel Corpus).
- Andiamo a definire la frequenza con cui queste parole occorrono nel Corpus.
- Le parole si suddividono a livello di carattere.
- Ad ogni iterazione si combinano i token più frequenti.

```
"hug", "pug", "pun", "bun", "hugs"
("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "ug" "s", 5)

Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un"]
Corpus: ("h" "ug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("h" "ug" "s", 5)

Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]
Corpus: ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)
```

In fase di tokenizzazione, quando dovremmo prendere una stringa e ottimizzarla, si valuta il vocabolario costruito, ad esempio la parola "linear" può essere costruita in due modi:

linear = li + near or li + n + ea + r

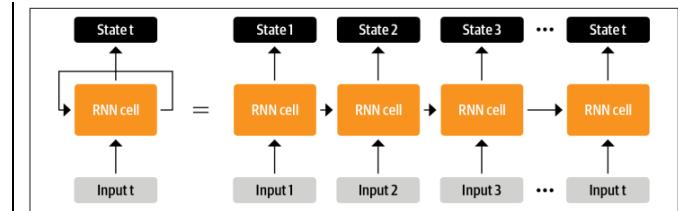
algebra = al + ge + bra or al + g + e + bra

Si vede la frequenza e si prende la combinazione più frequente.

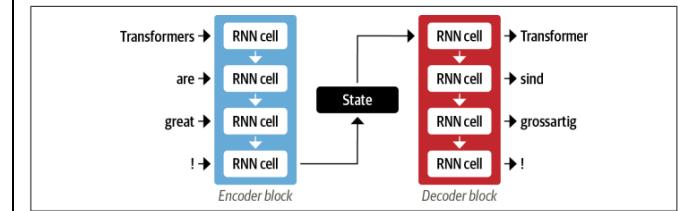
Number	Token	Frequency
1	li	3
2	I	5
3	ea	2
4	eb	4
5	near	6
6	bra	2
7	al	4
8	n	5
9	r	1
10	ge	11
11	g	7
12	e	8

ENCODER-DECODER FRAMEWORK:

Queste architetture contengono un ciclo di feedback nelle connessioni di rete che consente alle informazioni di propagarsi da uno step all'altro, rendendole ideali per la modellazione di dati sequenziali come i testi.



Il compito dell'**encoder** è quello di codificare le informazioni della sequenza di input in una rappresentazione numerica che spesso viene denominata **last hidden state**. Questo stato viene poi passato al decoder, che genera la sequenza di output.

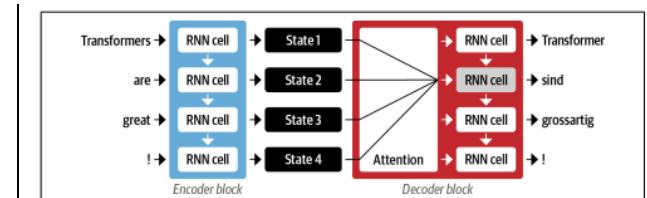


Sebbene sia elegante nella sua semplicità, una debolezza di questa architettura è che il last hidden state dell'encoder crea un **collo di bottiglia dell'informazione**. Esso deve rappresentare il significato dell'intera sequenza di input perché è tutto ciò a cui il decoder ha accesso quando genera l'output. È possibile superare questo problema consentendo al decoder di accedere a tutti gli hidden state dell'encoder.

ATTENTION:

L'**Attention** è un meccanismo che permette alle reti neurali di assegnare un peso differente o 'attenzione' a ciascun elemento in una sequenza.

Per sequenze di testo, gli elementi sono gli embedding dei token. Ovvero, rappresentazioni vettoriali a dimensione fissata.

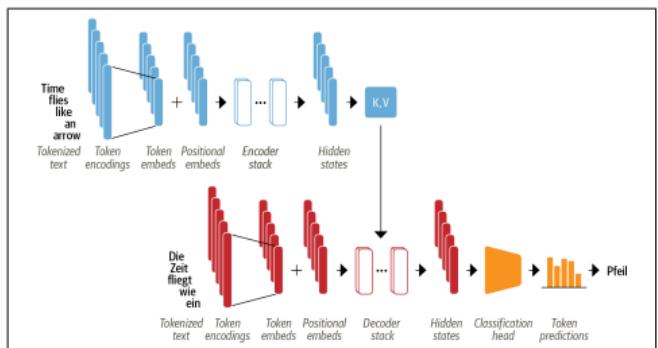


Sebbene l'attenzione abbia permesso di ottenere ottime performance su molti task di NLP, c'è ancora un grosso difetto nell'utilizzo di modelli ricorrenti per la codifica e la decodifica. I calcoli sono intrinsecamente sequenziali e non possono essere parallelizzati sulla sequenza.

ARCHITETTURA TRANSFORMER:

Tale architettura è composta da due componenti:

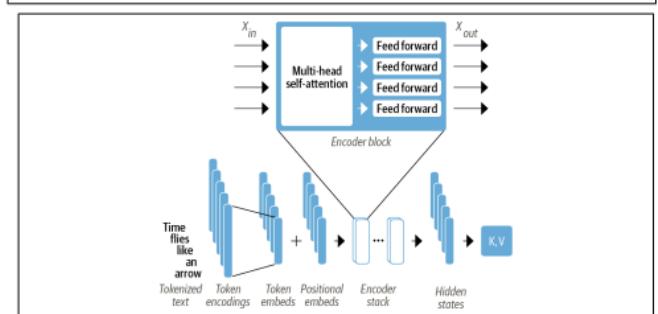
- **Encoder:** Converte la sequenza dei token di input in una sequenza di vettori di embedding, spesso denominata '**hidden states**';
- **Decoder:** Utilizza l'output dell'encoder per generare iterativamente una sequenza di token di output, uno ad ogni step.



ENCODER:

L'encoder è composto da uno stack di encoder layer disposti sequenzialmente. Ogni encoder layer riceve una sequenza di embedding che fornisce in input ai seguenti sublayer:

- Multi-head self-attention layer;
- Fully connected feed-forward layer.

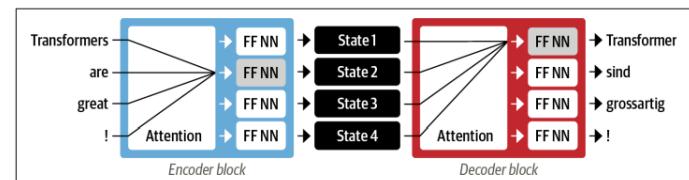


Il ruolo principale dello stack di encoder layer è quello di 'aggiornare' gli embedding di input al fine di ottenere rappresentazioni che codificano alcune informazioni contestuali: **Self-attention Layer**.

SELF-ATTENTION LAYER:

La parte “**Self**” del meccanismo di Self-Attention si riferisce al fatto che i pesi vengono calcolati per tutti gli hidden state dello stesso insieme, ad esempio quelli dell’encoder.

Al contrario, il meccanismo di Attention associato ai modelli ricorrenti prevede il calcolo della rilevanza di ogni hidden state dell’encoder rispetto a quello del decoder in un determinato time step di decodifica.



L’idea principale che sta alla base è che, invece di usare un embedding fisso per ogni token, possiamo usare l’intera sequenza per calcolare una **media ponderata** di ogni embedding.

In altre parole, data una sequenza di embedding di token x_1, \dots, x_n , il meccanismo produce una nuova sequenza x'_1, \dots, x'_n , dove ogni x'_i è una combinazione lineare di tutte le rappresentazioni x_j :

$$x'_i = \sum_{j=1}^n w_{ji} x_j$$

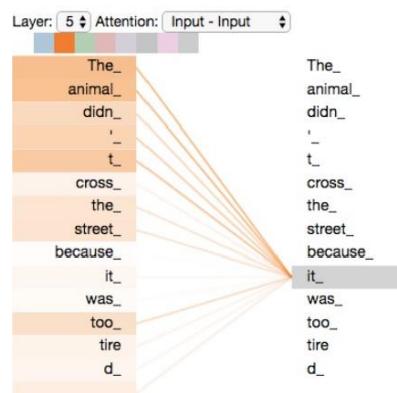
I coefficienti w_{ji} vengono definiti “**attention weights**” e sono normalizzati in modo tale che $\sum_j w_{ji} = 1$.

Invece, gli embedding generati in questo modo vengono definiti “**contextualized embeddings**”.

Per la frase in esempio per un essere umano è facile comprendere che “it” si riferisce ad un animale a strisce, mentre a livello di algoritmo è complicato.

Di conseguenza, utilizzando una libreria, data una sequenza di una stringa, permette di visualizzare quanto ogni parola del suo contesto influenza la sua rappresentazione.

“The animal didn’t cross the street because it was too tired”



▶ A cosa si riferisce ‘it’ in questa frase? Alla strada o all’animale?

SCALED DOT-PRODUCT ATTENTION:

Esistono diversi modi per implementare il self-attention layer, ma il più comune è lo “**scaled dot-product attention**”. Per implementare questo meccanismo sono necessari quattro passaggi principali:

1. Proiettare ogni embedding di input in tre vettori denominati

Query, Chiave e Valore;

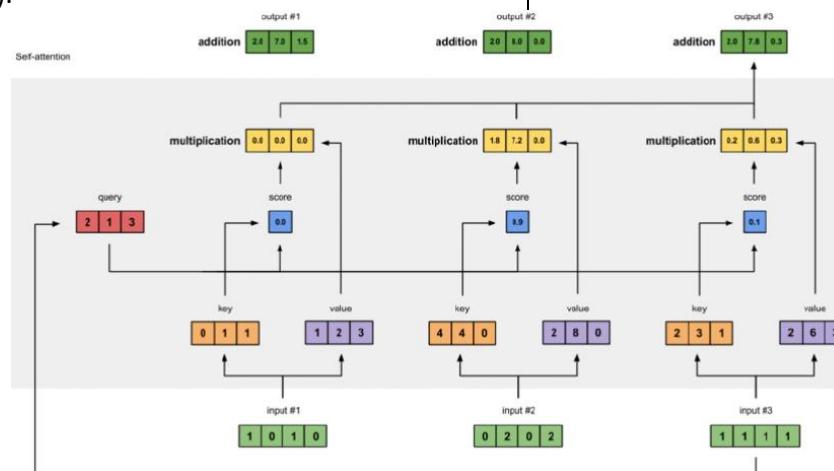
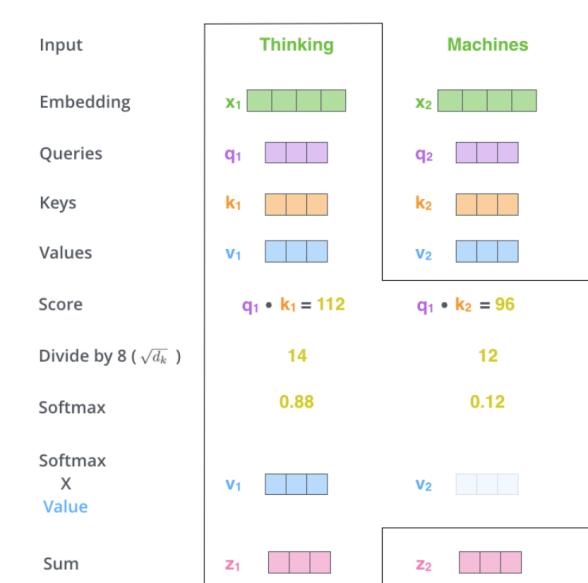
2. Calcolare gli ‘**attention score**’:

Ogni valore viene calcolato eseguendo il prodotto scalare tra il vettore Query della parola che stiamo valutando e il vettore Chiave di tutte le altre parole della sequenza. Quindi, se stiamo elaborando la parola in posizione #1, il primo punteggio è dato dal prodotto scalare tra q_1 e k_1 . Il secondo punteggio sarebbe il prodotto scalare di q_1 e k_2 , ecc.

3. Calcolare gli ‘**attention weights**’;

4. Aggiornare gli embedding dei token:

Tale operazione viene eseguita moltiplicando gli attention weights con i vettori Valore e sommando le rappresentazioni risultante ($x'_i = \sum_j w_{ji} v_j$).



La nozione di vettori query, chiave e valore può sembrare un po' criptica la prima volta che la si incontra. I loro nomi sono stati ispirati dai sistemi di **information retrieval**, ma possiamo motivarne il significato con una semplice analogia:

Immaginate di trovarvi al supermercato per acquistare tutti gli ingredienti necessari per la vostra cena:

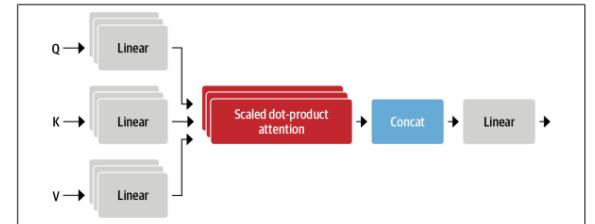
- Ricetta del piatto – Ingredienti (query);
- Passeggiando tra gli scaffali osservando le etichette (chiave);
- Controlliamo se corrispondono ad un ingrediente della lista;
- Se c'è una corrispondenza, si prende il prodotto dallo scaffale (valore).

Le trasformazioni proiettano gli embedding e ogni proiezione porta con sé il proprio insieme di parametri che consentono al self-attention layer di concentrarsi su diversi aspetti semantici della sequenza. Il problema è che il softmax layer tende a concentrarsi su un singolo aspetto, per risolvere tutto ciò si usa il **multi head-attention**.

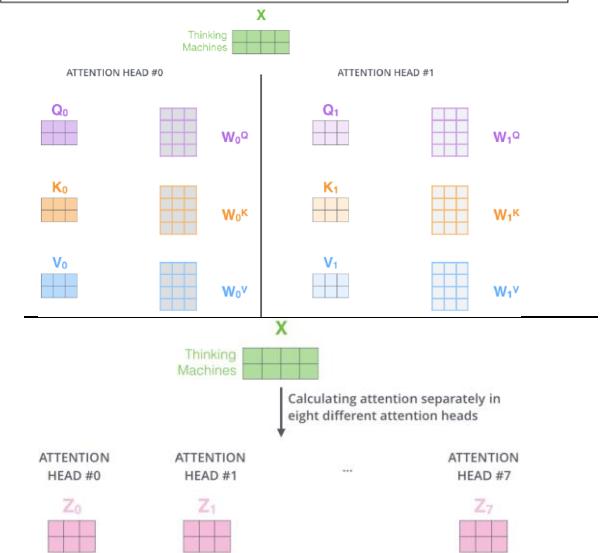
MULTI HEADED-ATTENTION:

Risulta vantaggiosa la presenza di più serie di proiezioni lineari, ognuna delle quali rappresenta una cosiddetta **attention head**.

La presenza di più head consente al modello di concentrarsi su più aspetti contemporaneamente.

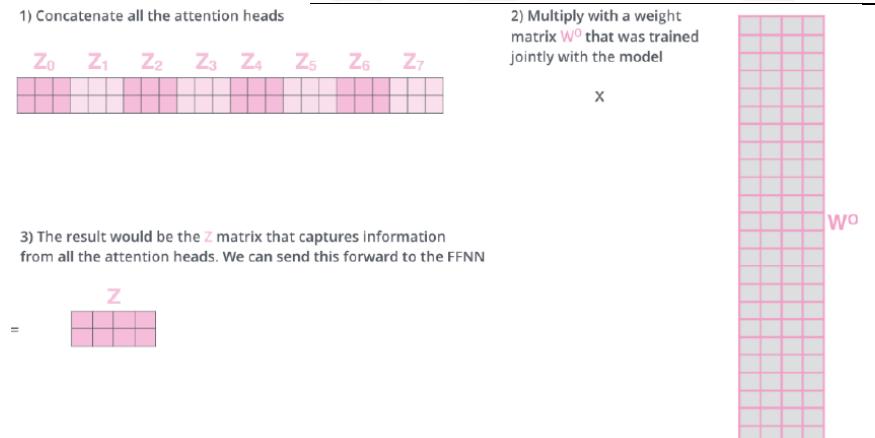


Eseguendo le operazioni precedentemente introdotte e utilizzando diverse matrici di peso in ogni attention head, otteniamo diverse matrici Z.

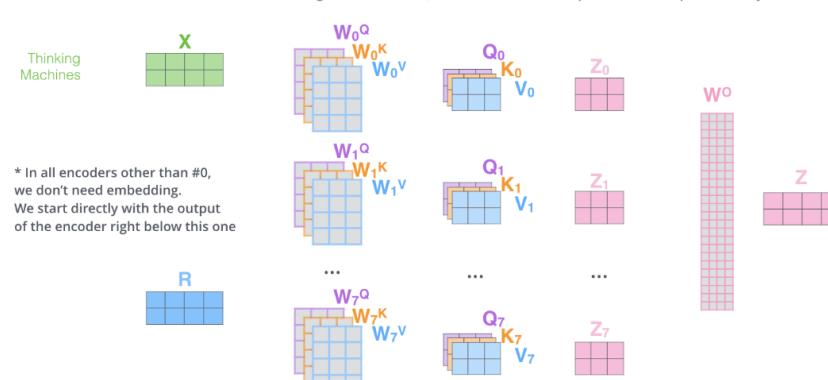


Problema: Il fully connected feed-forward layer si aspetta una singola matrice.

Concateniamo le matrici Z e le moltiplichiamo per la matrice W^o .



- 1) This is our input sentence* 2) We embed each word* 3) Split into 8 heads. We multiply X or R with weight matrices 4) Calculate attention using the resulting $Q/K/V$ matrices 5) Concatenate the resulting Z matrices, then multiply with weight matrix W^o to produce the output of the layer



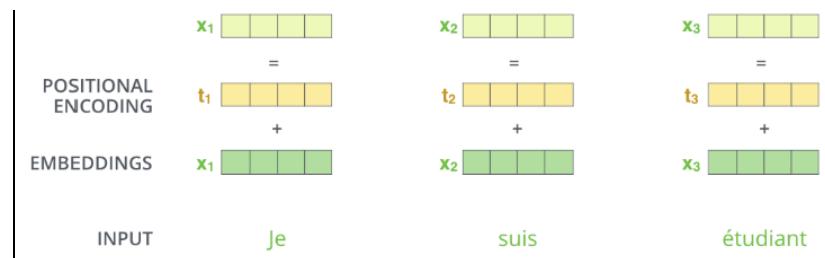
ENCODER LAYER:

Questo è tutto ciò che è necessario per implementare un encoder layer. Tuttavia, c'è un problema, ovvero per come li abbiamo implementati, gli encoder layer non tengono conto dell'ordine (posizione) delle parole nella sequenza di input. Fortunatamente, esiste un trucco molto semplice per codificare tali informazioni: **positional embeddings**.

POSITIONAL EMBEDDINGS:

I positional embeddings si basano su un'idea semplice, ma molto efficace: aggiungere un **position-dependent pattern** negli embedding dei token.

Se il pattern è caratteristico per ogni posizione, le attention head e i feed-forward layer di ogni stack possono imparare a codificare le informazioni sulla posizione nelle loro trasformazioni.



Sebbene i position-dependent pattern siano facili da implementare e ampiamente utilizzati, esistono alcune alternative:

- **Absolute positional representations:**

Pattern statici definiti tramite segnali modulati con le funzioni seno e coseno. Questo metodo funziona particolarmente bene quando non sono disponibili grandi volumi di dati.

- **Relative positional representations:**

Sebbene le posizioni assolute siano importanti, si può sostenere che quando si calcola un embedding, i token circostanti sono più importanti.

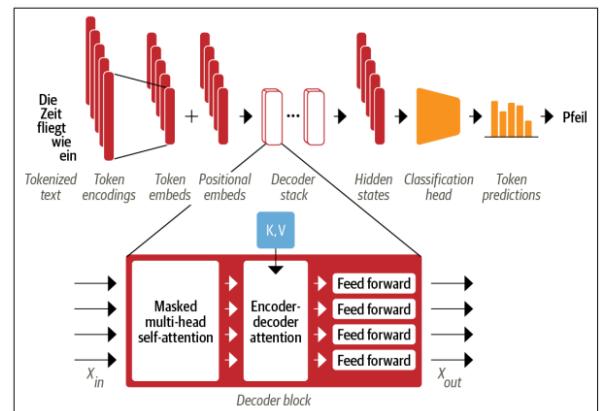
FULLY CONNECTED FEED-FORWARD LAYER:

Il fully connected feed-forward layer presente nell'encoder e nel decoder è una semplice rete neurale a due strati completamente connessa, ma con una modifica. Invece di elaborare l'intera sequenza di embedding come un unico vettore, elabora ogni embedding in modo indipendente. Per questo motivo, questo layer viene spesso chiamato **"position-wise feed-forward layer"**.

DECODER:

La differenza principale tra un decoder e un encoder è che il primo ha due attention sublayer:

- Masked multi-head self-attention layer
- Encoder-decoder attention layer



MASKED MULTI-HEAD SELF-ATTENTION LAYER:

Vediamo le modifiche da apportare per includere la mascheratura nel self-attention layer.

Il trucco consiste nell'introdurre una mask matrix con degli uni sulla diagonale inferiore e degli zeri su quella superiore.

Una volta applicata la mask matrix, possiamo impedire a ogni attention head di 'sbirciare' i token futuri sostituendo tutti gli zeri con $-\infty$.

```
tensor([[1., 0., 0., 0., 0.],
       [1., 1., 0., 0., 0.],
       [1., 1., 1., 0., 0.],
       [1., 1., 1., 1., 0.],
       [1., 1., 1., 1., 1.]])
```

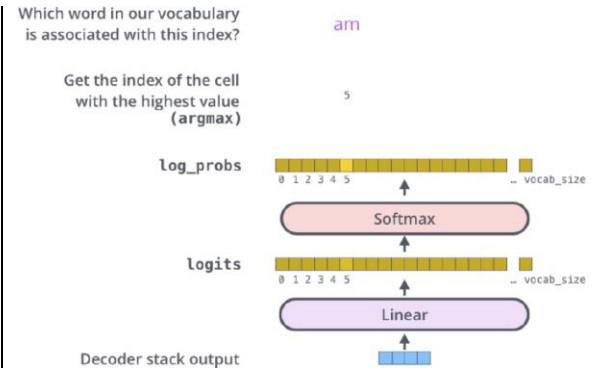
```
tensor([[[26.8082, -inf, -inf, -inf, -inf],
        [-0.6981, 26.9043, -inf, -inf, -inf],
        [-2.3190, 1.2928, 27.8710, -inf, -inf],
        [-0.5897, 0.3497, -0.3807, 27.5488, -inf],
        [ 0.5275, 2.0493, -0.4869, 1.6100, 29.0893]]],
```

```
grad_fn=<MaskedFillBackward0>)
```

LINEAR SOFTMAX LAYER:

Lo stack del decoder emette un vettore di float. Come lo trasformiamo in una parola? Questo è il compito del Linear Layer finale, il quale è seguito da un Softmax Layer. Il Linear Layer è una semplice rete neurale completamente connessa che proietta il vettore prodotto dallo stack di decoder layer in un vettore molto più grande chiamato **vettore logits**.

Il Softmax Layer trasforma i punteggi in probabilità.

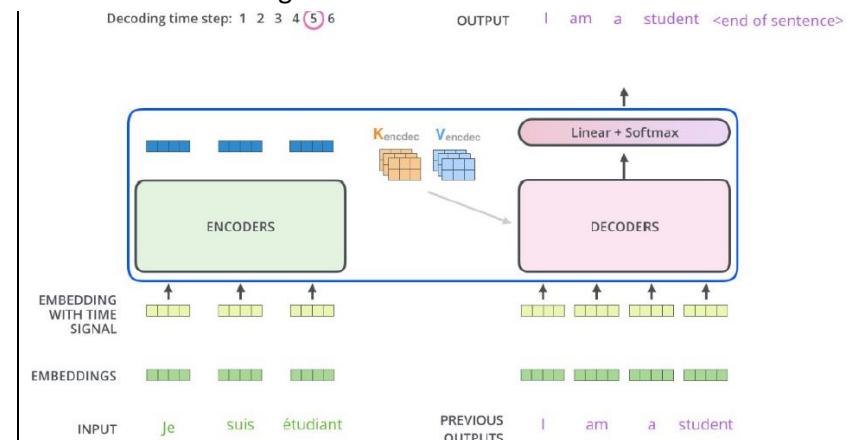


TRANSFORMER:

Immaginate di essere in classe a sostenere un esame (decoder). Il vostro compito è prevedere la parola successiva in base alle parole precedenti (input del decoder). Fortunatamente, un vostro amico (l'encoder) ha il testo completo. Sfortunatamente, si tratta di uno studente erasmus e il testo è nella sua lingua madre.

Da studenti astuti quali siete, trovate un modo per imbrogliare comunque:

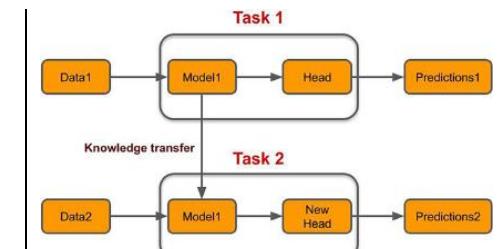
- Disegnate un piccolo fumetto che illustra il testo che avete già (la query) e lo date al vostro amico;
- Lui cerca di capire quale parte del suo testo corrisponde a tale descrizione (la chiave) e disegna una vignetta che descrive la parola che segue quella parte (il valore);
- Utilizzate tale informazione per completare il testo.



TRANSFER LEARNING:

Una delle problematiche più rilevanti quando si ha a che fare con modelli complessi, è la mancanza di dati etichettati sufficienti per riuscire ad addestrarli efficacemente.

Il **Transfer Learning** è una tecnica di Machine Learning in cui un modello, addestrato per uno specifico scopo, viene riproposto per un problema diverso, ma appartenente ad un contesto correlato.



TRANSFER LEARNING – STRATEGIE DI ADDESTRAMENTO:

- **Addestrare l'intera architettura:** addestrare ulteriormente l'intero modello pre-addestrato sul set di dati. In questo caso, il calcolo dei pesi viene propagato all'indietro attraverso l'intera architettura e i pesi pre-addestrati del modello vengono aggiornati in base al nuovo set di dati;
- **Addestrare alcuni layer, congelando altri:** un altro modo per utilizzare un modello pre-addestrato consiste nell'addestrarlo parzialmente. Bisogna mantenere congelati i pesi dei layer iniziali del modello e si riaddestrano i layer più alti;
- **Congelare l'intera architettura:** è possibile congelare tutti i layer del modello pre-addestrato, collegare nuovi layer alla rete neurale e addestrare il nuovo modello. Durante la fase di addestramento verranno aggiornati solo i pesi dei layer collegati al modello pre-addestrato.

TRANSFER LEARNING – VANTAGGI:

- **Sviluppo più rapido:** I pesi dei modelli pre-addestrati codificano già molte informazioni sul linguaggio. Di conseguenza, è sufficiente solo modellarli durante la fase di addestramento, richiedendo complessivamente molto meno tempo per il completamento del processo.
- **Meno dati richiesti:** Grazie ai pesi pre-addestrati, è possibile utilizzare un set di dati più piccolo rispetto a quello che sarebbe richiesto per costruire un modello da zero. Infatti, uno dei principali svantaggi dei modelli di NLP definiti da zero è che spesso per raggiungere una precisione elevata c'è bisogno di un set di dati dalle dimensioni proibitive per la fase di addestramento.
- **Risultati migliori:** È stato dimostrato che questa tipologia di modelli permette di ottenere risultati all'avanguardia per un'ampia varietà di attività: classificazione, inferenza linguistica, generazione di testo, ecc...

TIPI DI APPROCCI:

- Quelli che si servono **solo degli Encoder**: Bidirectional Encoder Representations from Transformers (BERT);
- Quelli che sfruttano **solo i Decoder**: Generative Pre-trained Transformer (GPT);
- Quelli che utilizzano **sia Encoder che Decoder**: Text-To-Text Transfer Transformer (T5).

BERT:

Bidirectional Encoder Representations from Transformers (BERT) è un modello di rappresentazione del linguaggio basato su Transformer e sviluppato da Google. Rilasciato alla fine del 2018.

BERT **base**: 12 encoder, 12 attention head - BERT **large**: 24 encoder, 16 attention head.

Pre-addestrato su Wikipedia e su Book Corpus, un dataset composto da più di 10.000 libri, di diverso genere.

L'encoder del Transformer legge l'intera sequenza di parole in entrambe le direzioni, contemporaneamente.

Per tale motivo è considerato **bidirezionale**.

Le capacità di BERT di elaborare l'input bidirezionalmente ha segnato l'inizio di una nuova era per il Natural Language Processing. A differenza dei modelli linguistici precedenti, BERT tiene conto contemporaneamente dei token precedenti e di quelli successivi. I modelli LSTM combinati da sinistra a destra e da destra a sinistra mancavano di questa parte di «contemporaneità».

Esempi:

Supponiamo di avere una frase da completare:

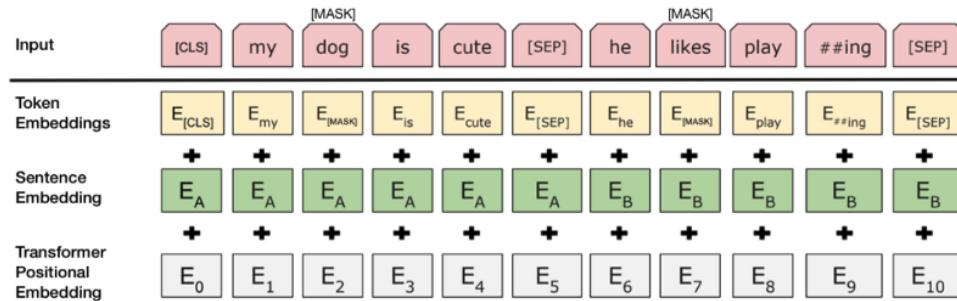
Analizzando la frase sia da destra che da sinistra, capisce completando la frase.

"The woman went to the store
and bought a _____ of shoes."
"I accessed the bank account"

L'obiettivo ultimo di BERT è generare un modello di rappresentazione del linguaggio, codificando un input tramite il meccanismo dell'attenzione, ragion per cui necessita solo della **parte degli encoder**.

Per poter essere processato da BERT l'input necessita di essere coadiuvato da una serie di metadata:

- **Token embeddings:** viene aggiunto un token [CLS] ai token delle parole in input all'inizio della prima frase e un token [SEP] viene inserito alla fine di ogni frase.
- **Segment embeddings:** ad ogni token viene aggiunto un marcitore che indica la frase A o la frase B. Questo permette all'encoder di distinguere tra la frase A e quella B.
- **Positional embeddings:** ad ogni token viene aggiunto un positional embedding per indicare la sua posizione nella frase.

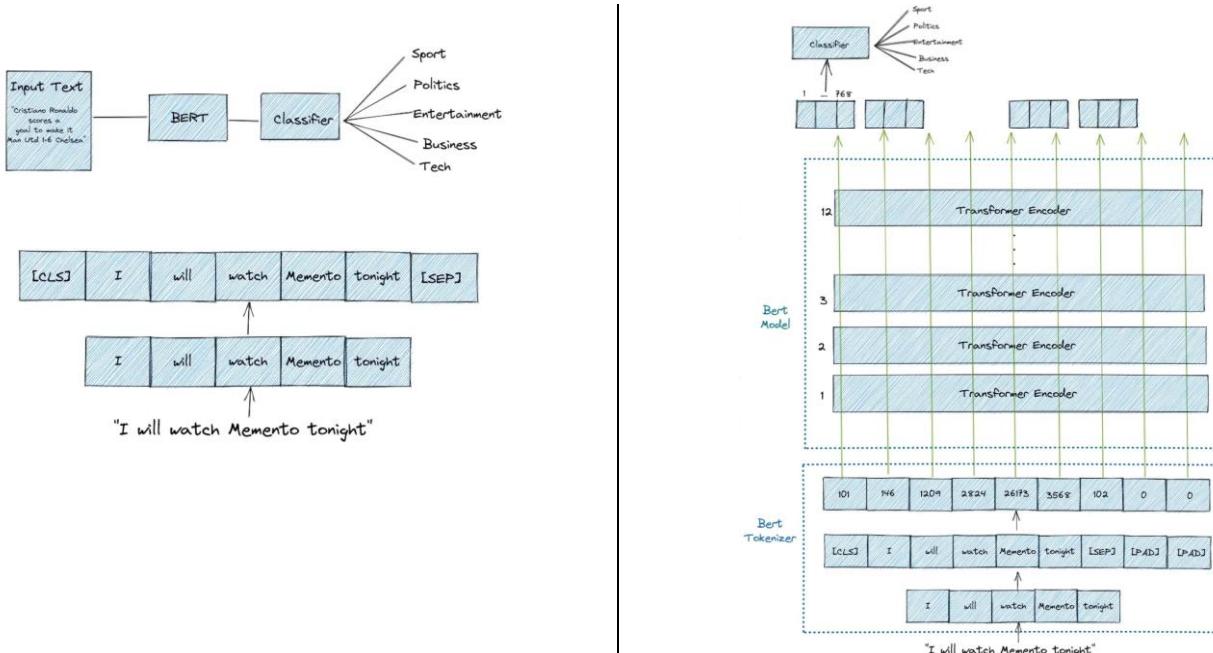


BERT si serve di due strategie di addestramento:

- **Masked Language Modeling (Masked LM):** prima di inserire sequenze di parole in BERT, il 15% delle parole in ciascuna sequenza vengono sostituite con il token [MASK]. Il modello tenta quindi di prevedere il valore originale delle parole mascherate, in base al contesto fornito dalle altre parole non mascherate nella sequenza.
- **Next Sentence Prediction (NSP):** durante il processo di addestramento, BERT riceve in input coppie di frasi e impara a prevedere se la seconda frase nella coppia è successiva alla prima nel documento:
 - 50% dell'input è composto da coppie di frasi in cui la seconda frase è effettivamente la frase successiva nel documento.
 - L'altro 50% è composto da coppie di frasi in cui la seconda frase viene scelta casualmente dal corpus.

FINE-TUNING BERT:

Al fine di personalizzare il comportamento di BERT per un task specifico è sufficiente aggiungere un layer in cima al modello base di BERT. Infatti, il modello BERT produce in output un vettore di embedding di 768 elementi per ciascuno dei token. Questi vettori possono essere passati come input per diversi altri modelli e per diverse applicazioni NLP, che si tratta di classificazione di testo, previsione di frasi successive, Named-Entity-Recognition (NER), o question-answering.



GPT-2:

GPT-2 è un modello di machine learning sviluppato da OpenAI, un gruppo di ricerca sull'intelligenza artificiale con sede a San Francisco. GPT-2 è in grado di generare testi grammaticalmente corretti e notevolmente coerenti.

La versione più «grande» di GPT-2 ha 1,5 miliardi di parametri ma non è stata rilasciata al pubblico temendo che il modello venisse utilizzato per scopi nefasti.

Sono stati rilasciati modelli più piccoli con 124-1558 milioni di parametri e uno strumento per la messa a punto di tali modelli su altri dati.

Addestrato su dataset di 40GB, su 8 Milioni di pagine, prese da **reddit**.

GPT-2 è costruito utilizzando **solo il decoder transformer model**. BERT, invece, utilizza solo gli encoder.

La differenza è che tra i due GPT-2, come i modelli linguistici tradizionali, produce un token alla volta. Chiediamo ad esempio a un GPT-2 ben addestrato di recitare la prima legge della robotica →

Una differenza fondamentale nel livello di self-attention è che il decoder maschera i token futuri, non cambiando la parola in [MASK] come il BERT, ma interferendo nel calcolo della self-attention e bloccando le informazioni dei token che si trovano a destra della posizione calcolata.

È importante che sia chiara la distinzione tra self-attention (quella che usa BERT) e Masked self-attention (quella che usa GPT-2). Un normale self-attention block permette a un token di «dare una sbirciata» sia ai token alla sua sinistra che quelli alla sua destra.

La masked self-attention, invece, impedisce che ciò avvenga.

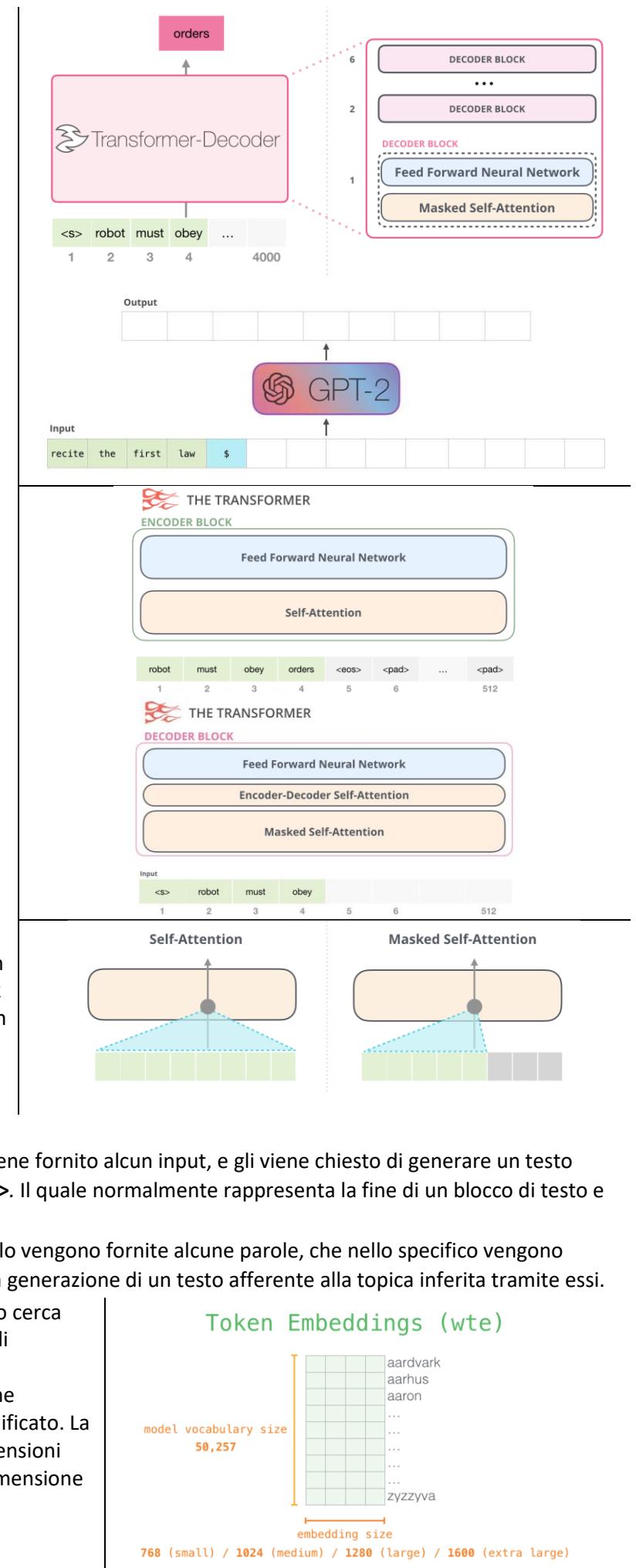
GPT-2 può essere eseguito in due modalità distinte:

1. Generate unconditional samples: al modello non viene fornito alcun input, e gli viene chiesto di generare un testo partendo da un singolo token, ovvero <|endoftext|>. Il quale normalmente rappresenta la fine di un blocco di testo e da questo iniziare a generare un testo.

2. Generate interactive conditional samples: al modello vengono fornite alcune parole, che nello specifico vengono chiamate **prompt**, i quale guidino il modello verso la generazione di un testo afferente alla topica inferita tramite essi.

Partiamo dall'input. Come in altri modelli NLP il modello cerca l'embedding della parola in ingresso nella sua matrice di embedding.

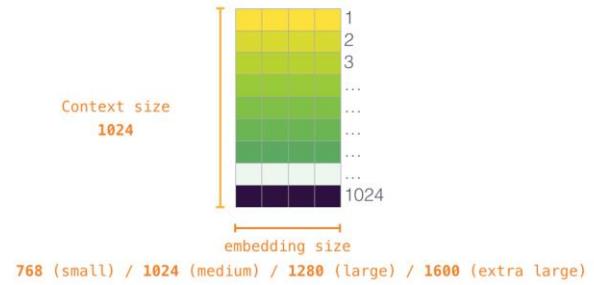
Ogni riga è un word embedding: un elenco di numeri che rappresentano una parola e ne catturano parte del significato. La dimensione di questo elenco varia a seconda delle dimensioni del modello GPT2. Il modello più piccolo utilizza una dimensione di embedding di 768 token per word.



All'inizio, quindi, cerchiamo l'embedding del token iniziale <ss> nella matrice di embedding.

Poi, prima di trasmetterlo al primo blocco del decoder, incorporiamo la sua positional encoding.

Positional Encodings (wpe)

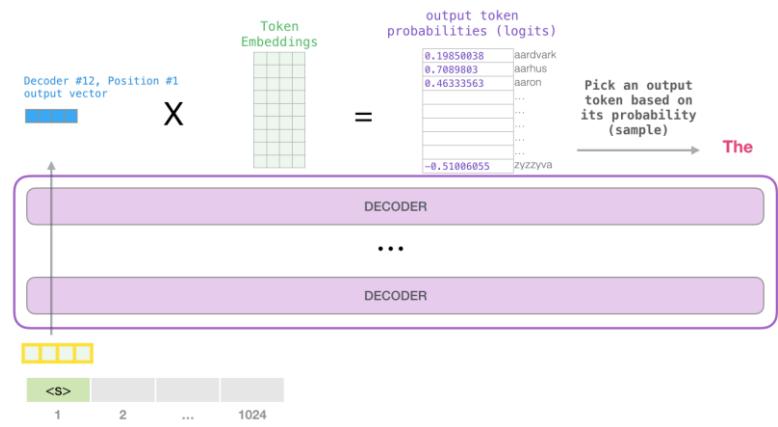


Il primo blocco può ora elaborare il token facendolo passare prima attraverso il processo di masked self-attention e poi attraverso la feed forward neural network.

Una volta che il primo blocco transformer ha elaborato il token, invia il vettore risultante su per lo stack per essere elaborato dal blocco successivo.

Il processo è identico in ogni blocco, ma ogni blocco trattiene i propri pesi in entrambi i sublayer della masked self-attention e FFNN.

Quando l'ultimo blocco, quello più in alto dell'architettura, produce il suo vettore di output, il modello moltiplica tale vettore per la matrice di embedding, ciò produce uno score associabile alle probabilità che l'n-esimo token presente nel vocabolario, sia quello da generare.



T5:

Il modello **Text-to-Text-Transfer-Transformer** proposto da Google, si è posto l'obiettivo di riprogettare tutti i compiti del NLP in un formato unificato text-to-text, in cui l'input e l'output sono sempre stringhe di testo.

Gli autori suggeriscono di utilizzare lo stesso modello, la stessa loss function e gli stessi iperparametri per tutti i compiti NLP. In questo approccio, gli input sono modellati in modo tale che il modello riconosca il compito e l'output è semplicemente la versione "testuale" del risultato atteso.

Il modello è stato addestrato sul dataset Colossal Clean Crawled Corpus (C4), ottenuto tramite processo di scrape di pagine web risalente ad Aprile 2019. Le pagine ottenute, sono state filtrate con i seguenti passaggi:

1. Mantenere le frasi che terminano solo con un segno di punteggiatura terminale valido (punto, punto esclamativo, punto interrogativo o virgolette finali).
2. Rimozione di qualsiasi pagina contenente parole offensive che compaiono nella "List of Dirty, Naughty, Obscene or Otherwise Bad Words"".
3. Gli avvisi del tipo "JavaScript deve essere abilitato" vengono rimossi filtrando tutte le righe che contengono la parola JavaScript.
4. Le pagine con testo segnaposto come "lorem ipsum" vengono rimosse.
5. I codici sorgente vengono rimossi eliminando tutte le pagine che contengono una parentesi graffa "{}" (poiché le parentesi graffe sono presenti in molti linguaggi di programmazione noti).
6. Per rimuovere i duplicati, si considerano gli intervalli di tre frasi. Tutte le occorrenze doppie delle stesse 3 frasi vengono filtrate.
7. Infine, poiché i compiti a valle riguardano soprattutto la lingua inglese, si utilizza langdetect per filtrare tutte le pagine che non sono classificate come inglesi con una probabilità di almeno 0,99.

Uno dei problemi principali di T5 è quello di rendere possibile l'approccio unificato da text-to-text.

Per utilizzare lo stesso modello per tutti i compiti a valle, si aggiunge un prefisso testuale specifico per il compito da svolgere, all'input originale che viene fornito al modello. Questo prefisso testuale viene anche considerato come un iperparametro.

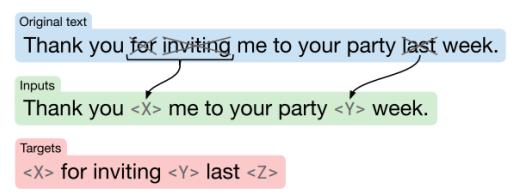
Ad esempio, per chiedere al modello di tradurre la frase "That is good." dall'inglese al tedesco, il modello dovrebbe ricevere la sequenza "translate English to German: That is good." ed essere addestrato a produrre "Das ist gut.".

Allo stesso modo, per i task di classificazione, il modello predice una singola parola corrispondente all'etichetta "target".

Anche T5 è addestrato con lo stesso obiettivo di BERT, ovvero il modello di linguaggio mascherato con una piccola modifica.

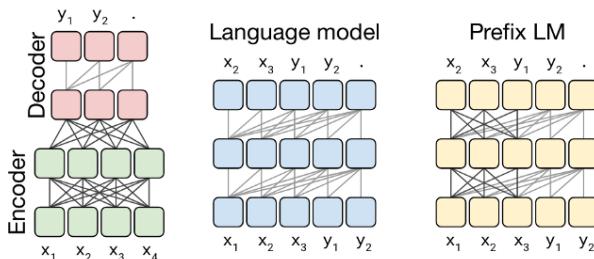
La sottile differenza che il T5 impiega è quella di sostituire più token consecutivi con una singola parola chiave Mask, a differenza del BERT che usa il token Mask per ogni parola.

Poiché l'obiettivo finale è quello di addestrare un modello che prende in input testo e emette testo, i target sono stati progettati per produrre una sequenza, a differenza di BERT, che cerca di emettere una parola attraverso una FFNN e un softmax a livello di output.



T5's mask language modeling (Raffel et al., 2019)

Architettura di training:



T5 viene prima pre-addestrato sul dataset C4 per l'approccio di denoising (BERT-style), andando a mascherare span di token e con un'architettura Encoder-Decoder.

Viene poi fatto fine-tuning sui compiti a valle tramite un approccio supervisionato, con un'appropriata modellazione degli input per l'impostazione text-to-text.

Tuning degli iperparametri:

