Intelligenza Artificiale e Laboratorio

Lorenzo D'Isidoro
lorenzo.disidoro@gmail.com

Intelligenza Artificiale e Sistemi Informatici A.A. 2022/2023

Contents

1 Introduzione al Corso

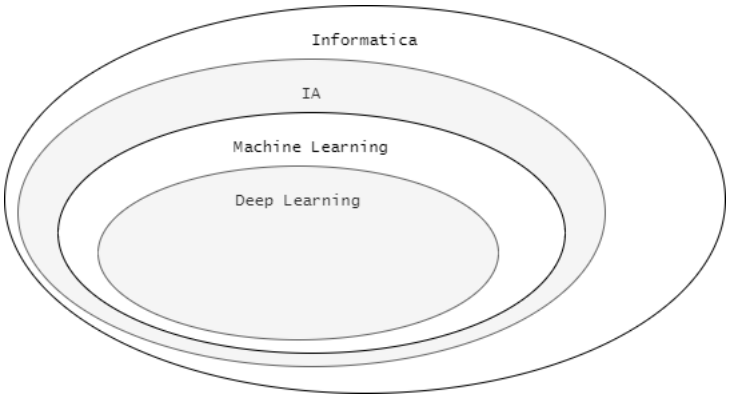
L'insegnamento ha l'obiettivo di approfondire le conoscenze di Intelligenza Artificiale con particolare riguardo alle capacità di un agente intelligente di fare inferenze sulla base di una rappresentazione esplicita della conoscenza sul dominio. Alle competenze metodologiche si affiancano competenze progettuali perché il corso prevede la sperimentazione di metodi di ragionamento basati sul paradigma della programmazione logica, lo sviluppo di un agente intelligente in grado di esibire sia comportamenti reattivi che deliberativi (utilizzando ambienti basati su regole di produzione) e la sperimentazione di strumenti per architetture cognitive. Continua sulla pagina del corso.

2 Intelligenze Artificiale

2.1 Introduzione alla Materia

L'intelligenza artificiale ha origine negli anni 40 con la cibernetica, ovvero costruire modelli meccanici in grado di simulare il comportamento adattivo di sistemi naturali. L'intuizione principale della cibernetica è stato quello di proporre una prospettiva unificata allo studio di organismi biologici e macchine.

"L'intelligenza artificiale è una branca dell'informatica che studia i fondamenti teorici, le metodologie e le tecniche che consentono la progettazione di sistemi hardware e software capaci di fornire a un calcolatore prestazioni che, a un osservatore comune, sembrerebbero essere di pertinenza esclusiva dell'intelligenza umana." (Somalvico, 2003)

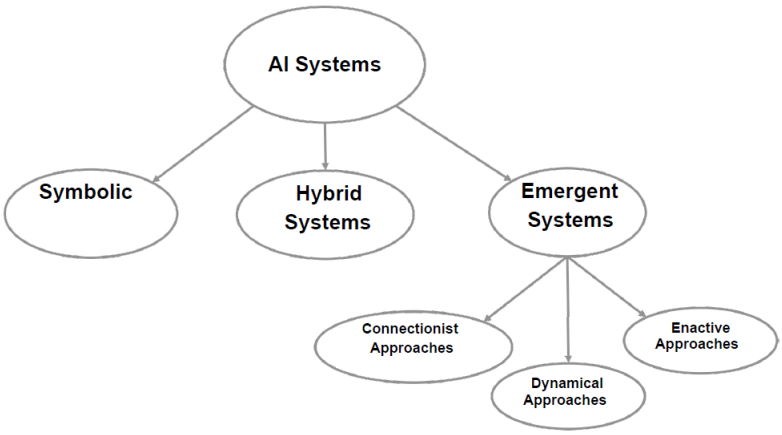


Come illustrato nel diagramma, l'intelligenza artificiale è un sottoinsieme dell'informatica ed è bene sottolineare che al suo interno si incontrano diverse discipline:

- Ingegneristica: costruire macchine capaci di svolgere compiti intelligenti integrando componenti diverse.
- Psicologica: costruire macchine capaci di esprimere le caratteristiche essenziali dell'attività cognitiva umana spiegando i rapporti tra pensiero e fisicità dell'uomo.

2.2 Approcci

Esistono diversi tipi di approccio alla materia:



Gli approcci principali sono:

- Emergentista (Emergent System): tutti i sistemi facente parte di questo approccio sono basati sull'apprendimento, quindi si nutrono di dati.
  - Connectionist Approaches: un approccio che si è sviluppato dai tentativi di capire come funziona il cervello umano a livello neurale e, in particolare, come le persone imparano e ricordano.
  - Dynamical Approaches: un approccio applicato allo studio dei sistemi automatici.
  - Enactive Approaches: è focalizzato sulle azioni mirate e sulle interazioni dinamiche reciproche con l'ambiente riunendo mente/cervello, mente/corpo e corpo/ambiente. Ad esempio, quello che una macchina, un robot, ha con l'ambiente circostante utilizzando i sensori.
- Simbolico (Symbolic System): tutti i sistemi facente parte di questo approccio applicano una logica "top down", vengono implementate delle reti semantiche di concetti e predicati.
- Ibrido (Hybride System): sistemi ibridi

2.3 Intelligenza Artificiale Forte e Debole

Con il termine "IA forte" ci si riferisce a quella teoria che afferma che la macchina che agisce in modo "intelligente" possiede una "mente" e una "coscienza", e quindi "pensa", esattamente nel modo in cui accade agli esseri umani (o ad altri animali).

Con il termine "IA debole" ci si riferisce alla tesi sostenuta dalla comunità scientifica moderna, ovvero che le simulazioni al calcolatore dei processi o capacità mentali non vanno confusi con la loro riproduzione.

2.4 Obiettivi

L'obiettivo dell'IA è quello di rendere le macchine intelligenti, in grado di replicare come l'essere umano agisce, o reagisce, ad uno stimolo sulla base di ciò che ha appreso fino a quel momento.

Di fatto l'essere umano apprende a seconda delle esperienze maturate, positive o negative che siano. Questa conoscenza esplicita viene utilizzata per ragionare e **inferire** nuova conoscenza, ad esempio, sappiamo che "i gatti miagolano", che Tom è un gatto e possiamo inferire che Tom miagola.

Bisogna tenere a mente però che l'essere umano, cosi come i sistemi intelligenti, sono sempre in possesso di conoscenza parziale, incerta o errata. Questo è un problema soprattutto per i sistemi intelligenti perché, a differenza degli esseri umani, il **ragionamento è monotòno**, quindi l'aggiunta di nuove asserzioni o regole alla base di conoscenza non può invalidare teoremi precedentemente dimostrati, ma solo aggiungerne di nuovi.

Il ragionamento, le inferenze, che l'essere umano fa è spesso non monotòno: si fanno inferenze tentative, anche in mancanza di informazioni complete e, a fronte di nuove conoscenza, si rivedono le inferenze o le assunzioni precedentemente fatte.

3 Laboratorio

3.1 Introduzione a Prolog

Il Prolog è un linguaggio di programmazione che adotta il paradigma di programmazione logica. E' un interprete che permette di dimostrare dei fatti a partire da un dominio di conoscenza costituito da **fatti** e **clausole**. Bisogna osservare alcune informazioni importanti prima di proseguire:

- I fatti terminano sempre con il "."
- Le costanti hanno la prima lettera minuscola
- Le variabili hanno la prima lettera maiuscola
- Per uscire dall'editor Ctrl+D o *?- halt.*
- Per utilizzare la modalità debug *?-trace.* e per uscire *?-nodebug.*
- Includere nuovi file è possibile con il comando *?- consult(fileName).* oppure *?- ['nomeFile'].*

Il **fatto** è un insieme di informazioni esplicite relative al mio dominio, ad esempio "Tom è un gatto", la sintassi è la seguente:

```
gatto(tom).
```

Un fatto è costituito da un **predicato**, che è una funzione senza argomento, nel nostro esempio è "gatto" e da una costante, ovvero "tom".

Una **regola** di inferenza è uno schema formale che si applica nell'eseguire un'inferenza, una **clausola** è un insieme finiti di fatti e regole.

Di seguito un semplice esempio di programma che definisce fatti e regole:

```
% Regola: "tutti i gatti sono felini"
felino(X):-gatto(X).

% Regola: "gli uccelli volano"
vola(X):-uccelli(X)
```

Per utilizzare il linguaggio matematico una regola può essere scritta nel seguente modo:

$$\forall X : gatto(X) = true => felino(X) = true$$

Di seguito un semplice esempio di programma che definisce fatti e regole:

```
% Fatti
gatto(tom).
gatto(fred).
tigre(mike).
vivo(tom).

% Regole
felino(X):-gatto(X).
felino(X):-tigre(X).

% Regola: "X miagola se e' un gatto AND e' vivo"
miagola(X):-gatto(X),vivo(X).
```

Da interprete possiamo effettuare un test veloce, come primo passaggio però dovremmo importare il file contenente i fatti e le regole appena mostrate:

```
?- ['Esercizi/lezione1.pl'].
true.
?- gatto(Chi).
Chi = tom ;
Chi = fred.
?- felino(tom). % deduzione
true.
?- gatto(mike).
false.
```

3.2 Congiunzione e Disgiunzione Logica

Come accennato in precedenza la congiunzione, l'and booleano, può essere implementata utilizzando la virgola ",", per quanto riguarda la disgiunzione, quindi l'or booleano, sarà sufficiente definire più clausole.

Di seguito un esempio che racchiude quanto detto:

```
% fatti: genitore e figli
genitore(paolo,mario).
genitore(anna,mario).
genitore(paolo,luisa).
genitore(anna,luisa).
% paolo e' nonno di chiara
genitore(mario,chiara).

 % paolo e' padre di marco, marco non ha una madre
genitore(paolo,marco).

genitore(antonella,chiara).

genitore(mario,alberto).
genitore(francesca,alberto).
genitore(marco,serena).
genitore(serena,fabrizio).
genitore(fabrizio,lorenzo).

% regole
nonno(X,Y):-genitore(X,Z),genitore(Z,Y).

antenato(X,Y):-genitore(X,Y).
antenato(X,Y):-nonno(X,Y).
antenato(X,Y):-genitore(X,Z),antenato(Z,Y).
```

Analizzando quanto scritto notiamo che è stata implementata una disgiunzione logica con le regole:

```
antenato(X,Y):-genitore(X,Y).
antenato(X,Y):-nonno(X,Y).
```

Quanto scritto sta ad indicare che X è un antenato di Y sse X è genitore di Y **oppure** X è nonno di Y, notare che X e Y sono delle variabili.
Inoltre possiamo pensare di generalizzare utilizzando una regola ricorsiva dove il caso base verifica che che X sia antenato di Y con la regola della genitorialità, mentre invece il caso ricorsivo verifica che X sia genitore di Z **oppure** che Z sia antenato di Y:

```
antenato(X,Y):-genitore(X,Y).
antenato(X,Y):-genitore(X,Z),antenato(Z,Y).
```

Dobbiamo sottolineare il fatto che, in Prolog, l'ordine delle clausole impatta sulla corretta esecuzione, possono esserci degli errori, e sull'efficienza della nostra soluzione. Quindi possiamo avere due soluzioni con una base di conoscenza

semanticamente identica ma sintatticamente no, come nel caso che segue dove sono state invertite delle clausole della funzione ricorsiva precedentemente implementata.

```
% invertiamo caso base e caso ricorsivo
% invertiamo nel caso ricorsivo le clausole
antenato(X,Y):-antenato(Z,Y),genitore(X,Z).
antenato(X,Y):-genitore(X,Y).
```

Questo perché Prolog prende in considerazione le regole nell'ordine in cui si trovano, anche se in una formulazione ricorsiva ci si aspetta prima il caso base e poi quello ricorsivo, ma questo è solo un esempio.
In ultimo istanza implementiamo la regola *fratelloGermano* verifica se A e B sono fratelli, dove A è diverso da B:

```
fratelloGermano(A,B):-
    % A != B
    A \== B,

    % A e B sono fratelli se hanno gli stessi genitori
    genitore(PrimoGenitore,A),
    genitore(SecondoGenitore,A),
    genitore(PrimoGenitore,B),
    genitore(SecondoGenitore,B).
```

In questa formulazione però c'è un problema, ovvero che la disuguaglianza è un predicato, non è un vincolo. Quindi viene valutato con il valore corrente di A e B, quando viene chiesto $fratelloGermano(mario, X)$ non viene controllato il valore di X quando è istanziato, di conseguenza tra le risposte avremo anche $X = mario$, quindi ci dice che "Mario" è fratello di se stesso. Va da se che abbiamo un problema anche con la funzione *genitore* nel caso in cui il *PrimoGenitore* è uguale al *SecondoGenitore*, perché due fratelli che hanno un solo genitore in comune possono essere considerati dalla nostra funzione fratelli germani. Ci sono delle librerie, delle estensioni, per risolvere questo tipo di problemi di soddisfacimento di vincoli, al momento ovviamo al problema apportando le seguenti modifiche:

```
fratelloGermano(A,B):-
    % A e B sono fratelli se hanno gli stessi genitori
    % A e B non devono essere la stessa persona
    % PrimoGenitore deve essere diverso a SecondoGenitore

    genitore(PrimoGenitore,A),
    genitore(SecondoGenitore,A),
    PrimoGenitore \== SecondoGenitore,
    genitore(PrimoGenitore,B),
```

```
    A \== B,
    genitore(SecondoGenitore,B).
```

3.3 Le Liste

La definizione di lista può essere data ricorsivamente nel modo seguente:

- L'atomo [] rappresenta una lista vuota
- Il termine $[HEAD|TAIL]$ indica una lista con in testa l'elemento $HEAD$ e in coda gli elementi $TAIL$

Il Prolog fornisce una notazione semplificata per la rappresentazione delle liste: la lista $[HEAD|TAIL]$ può essere rappresentata anche come $.(HEAD, TAIL)$. In generale nell'ambito di una lista possibile individuare due parti strutturalmente significative, abitualmente note come testa e coda, ma possiamo decidere di rappresentarle in funzione della nostra necessità, ad esempio:

- $[X, Y|Z]$ indica una lista composta da un primo elemento X, da un secondo elemento Y e da una coda Z (eventualmente vuota).
- $[X, Y, Z]$ indica una lista composta esattamente da tre elementi X, Y Z.
- $[X, [Y, Z]]$ indica una lista composta da un primo elemento X e da un secondo elemento [Y,Z] che è a sua volta una lista composta da due elementi.

Le liste possono contenere numeri interi, liste di liste (senza alcun limite teorico al livello di annidamento), atomi e variabili.

3.3.1 Aggiungere e Eliminare un Elemento

Per aggiungere l'elemento X in testa alla lista L è sufficiente scrivere $[X|L]$. Oppure utilizzando il predicato **add** che aggiunge l'elemento X in testa alla lista L fornendo come risultato la lista L1:

```
add(X,L,L1).
```

Per elimina dalla lista L una qualsiasi occorrenza di X dando come risultato la lista L1, possiamo usare il predicato **delete**:

```
delete(X,L,L1).
```

3.3.2 Ricerca un Elemento

Per verificare se un elemento è contenuto o no all'interno di una lista possiamo utilizzare il predicato **member**, che andrà a ricercare l'elemento X all'interno della lista L, e può essere implementato nel seguente modo:

```
member(X,L).
```

3.3.3 Concatenare due Liste

Per concatenare due liste possiamo utilizzare il predicato **append**, che anrà ad appendere la lista L2 alla lista L1 e come risultato avremo la lista L, può essere implementato come segue:

```
append(L1, L2, L).
```

3.4 Predicato "is"

Il predicato *is* viene utilizzato in Prolog per forzare la valutazione delle espressioni aritmetiche. Quindi, con i concetti illustrati in questi ultimi paragrafi, possiamo implementare la somma degli elementi di una lista come segue:

```
somma([],0).
somma([Head|Tail],Res):-
    somma(Tail,SommaTail),
    Res is Head+SommaTail.
```

```
?- ['Esercizi/lezione3.pl'].
true.

?- somma([1,2,3],X).
X = 6.

?- somma([1,2,3],6).
true.

?- somma([1,2,3],7).
false.
```