Introduzione all'Intelligenza Artificiale

Falbo Andrea - A.A 23/24

Introduzione

Concetto di Agente Intelligente, architetture, agente e ambiente

AI: studio di agenti che ricevono input dall'ambiente ed eseguono azioni.

Agentificazione dell'AI: agente unico che risolve problemi adottando varie tecniche ed approcci.

Agente: Secondo Russell e Norvig un agente può essere visto come qualcosa che *percepisce l'ambiente* attraverso sensori ed *agisce su di esso* attraverso attuatori.

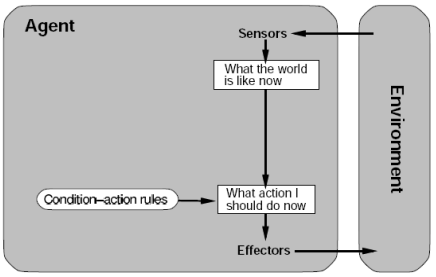
Funzione Agente f: $P^* \rightarrow A$ mappa la storia delle percezioni in azioni.

Classificazione Agenti

Un agente può essere classificato in 4 (o oramai 5) classi:

1. simple reflex agents
2. model-based reflex agents
3. model-based goal-based agents
4. model-based utility-based agents
5. learning agents

1.1 - Simple Reflex Agents



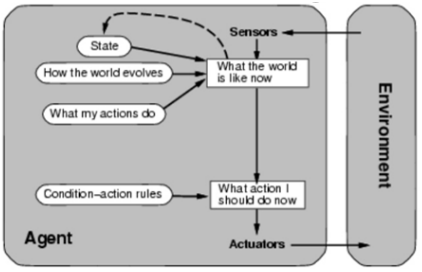
Ha dei sensori ma *non ha rappresentazione su come è il mondo* che lo circonda. Sa solo che quando non può continuare il suo comportamento, prende un azione per ritornare allo stato normale.

Esempio - Roomba da 20 euro: specifica comportamentale che segue il Random Walk:

1. Vai dritto
2. Se la ruota non gira
    a. ruota di un angolo casuale
3. prova ad andare dritto

1.2 - Model-Based Reflex Agents

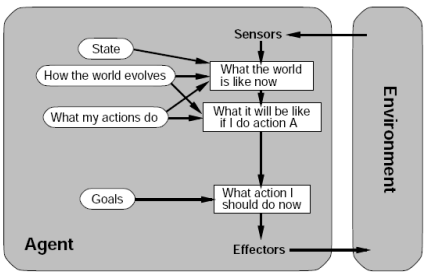
Abbiamo una percezione del mondo, quindi conosciamo il nostro stato e sappiamo come il mondo evolve e come le mie azioni modificano il mondo.



Esempio - Roomba costoso: conosce il suo livello di batteria, riesce a capire da dove parte e dove dovrebbe andare.

1.3 - Model-Based, Goal-Based Agents

Sappiamo come evolve il mondo e come potrebbe evolvere il mondo data una azione. Il tutto è basato su obiettivi



Esempio - Scacchi: Abbiamo un albero di mosse che possiamo intraprendere. Possiamo validare gli alberi, ma sono giganteschi. Problema di Ricerca.

Problema di Ricerca

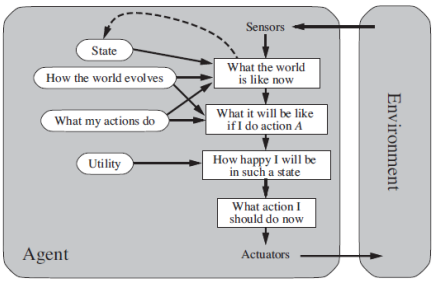
Un problema di ricerca consiste in un insieme di stati di cui 1 iniziale e 1 finale ed una funzione successore. Una *soluzione* è la sequenza di azioni che porta dallo stato iniziale a quella finale. Possiamo classificare gli stati in 2 categorie:

- *world* state: include ogni dettaglio dell'ambiente
- *search* state: tiene conto solo dei dettagli utili per il problema

Gli alberi di ricerca descrivono, dato un nodo iniziale, tutti i possibili scenari. Lo ste space graph fornisce una rappresentazione matematica di un problema di ricerca. In entrambi i casi è difficile rappresentarli completamente. L'idea è "guardare nel futuro" e scegliere i goal delle strutture. Come fare se non vedo un goal? Mi baso sul concetto di utilità.

1.4 - Model-Based Utility-Based Agents

Questo modello riprende il precedente ma utilizza il concetto di utilità. Riusciamo a discriminare dentro lo spazio degli stati.



Esempio - Scacchi: Non per forza perdere un proprio pedone è un male, se comporta una

perdita maggiore per l'avversario.

Osservazione: Stiamo sempre aggiungendo intelligenza al modello, ma per farlo dobbiamo considerare complicazioni fisiche (capacità elaborative maggiori) ma anche realizzative del programmatore (come si discrimina l'utilità?)

Ambiente

Torniamo a parlare di ambienti. Russell e Norvig definiscono l'ambiente come il problema: l'agente è collocato in una situazione che percepisce e sulla quale agisce. Dunque la difficoltà è data da categorie alla quale un ambiente può far parte:

- accessibile vs inaccessibile: definiamo come accessibile se l'agente può ottenere tutte le informazioni dettagliate ed aggiornate
- deterministico vs non deterministico: sarà deterministico se ogni azione produce un singolo effetto garantito, senza incertezza.
- episodico vs non episodico (o sequenziale) : se l'azione futura intrapresa dall'agente dipende dal singolo episodio definiremo come episodico, altrimenti come sequenziale.
- statico vs dinamico: si definisce un ambiente statico se questo rimane invariato, fatta eccezione per le azioni dell'agente.
- discreto vs continuo: definiamo discreto un ambiente dove ci sono un numero finito di azioni che possono essere intraprese.

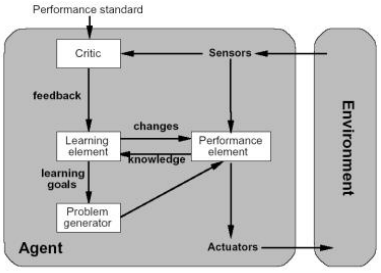
Autonomia

L'agente che ha la tabella di tutte le situazioni possibili e non sbaglia mai è intelligente? Bisogna definire quest'ultima. Più facile parlare di autonomia, anche se questa ha 3 astrazioni diverse:

- bassa: un agente può controllare il suo stato interno
- media: la capacità di decidere le azioni da eseguire in termini di tempo, adempimento richiesta o azione non condizionata da evento esterno.
- alta (quasi utopica al momento): l'abilità di decidere esclusivamente in base alla propria esperienza personale invece che dalla conoscenza cablata.

1.5 - Learning Agents

CI sono due elementi centrali: elemento che agisce (performance), ovvero colui che fa le *inferenze*. L'altro fa apprendimento e *costruisce*. Potremmo vederlo come due agenti separati, dove l'agente di apprendimento genera altri agenti.



Intelligenza Artificiale Simbolica

Knowledge Graph, Semantic Web

Knowledge Graphs in the Web of Data

1.1 Data, Information, and Knowledge

I dati grezzi esistono ma non hanno significato.

Un informazione è un dato a cui è stato dato un significato in una connessione relazionale. Non è ancora sufficiente perché può rispondere solo ad alcune domande, come Quando, Dove, Perché ecc..

Una conoscenza è una collezione di informazioni con l'intenzione di essere utile. Sono arricchite di semantica.

Esempio: 33.6 è un dato, la balenottera è lunga massimo 33.6m è un informazione

Tripartita Analisi della Conoscenza: Si ha conoscenza di p sse:

- p è vero
- S crede in p
- allora S è giustificato nel credere p

Dobbiamo automatizzare il processo di conoscenza, quindi abbiamo bisogno di una rappresentazione della conoscenza formale: Ontologie. Vedremo nelle ultime lezioni.

1.2 How to Represent Knowledge?

La prima domanda che può venire in mente è: Perchè non usare il linguaggio naturale? Perché ha più forme di conoscenza, più simboli che possono esprimere lo stesso concetto e quindi non utilizzabile per rappresentare l'informazione in modo univoco

Esempio: Una sola parola "concetto" Jaguar può significare varie "entità": giaguaro, la macchina e una versione del MacOS.

La comprensione è la capacità di cogliere il significato delle informazioni.

1.3 Art of Understanding

L'informazione viene compresa dal destinatario di un messaggio, se il destinatario interpreta l'informazione correttamente. Dipende da:

- Sintassi, definisce la normativa della strutturazione dei dati.
- Semantica, si focalizza sul senso e significato dei simboli.
- Contesto, un concetto ha un significato in base all'ambiente in cui è espresso.
- Pragmatica, riflette l'intenzione con cui la lingua viene utilizzata per comunicare
- Esperienza, considera tutte le informazioni che abbiamo appreso e inserito
- nel contesto con il mondo in cui vivi.

In questo capitolo vedremo i primi due livelli, nel corso vedremo anche l'ultimo.

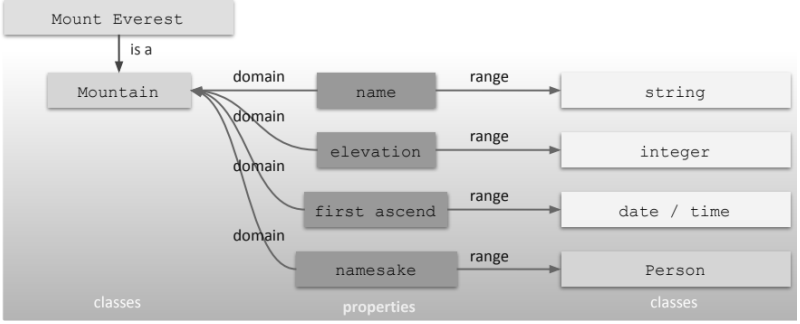
1.4 Towards a Universal Knowledge Representation

Climate Change is the Everest of all problems. Ma Everest può assumere diversi significati. Bisogna disambiguare e trovare la semantica giusta. Inoltre abbiamo un'entità (Monte Everest) e diverse classi (Mountain, Landform, Geomorphic Unit)

```
MountEverest ∈ Mountain
Mountain ⊆ Landform
Landform ⊆ GeomorphicalUnit
Geomorphic Unit ⊆ NaturalGeographicObject
GeorgeEverest ∈ Person
Person ∩ Mountain = ∅
MountEverest ∉ Person ← Inferenza logica
GeorgeEverest ∉ Mountain ← Inferenza logica
```

Rappresentazione della conoscenza vs Strutture Dati:

1. La logica matematica fornisce un quadro per *esprimere formalmente la semantica* delle rappresentazioni della conoscenza.
2. La semantica delle rappresentazioni della conoscenza può essere *definita esplicitamente*.
3. La logica matematica consente inferenze logiche e ragionamenti rappresentazioni della conoscenza.

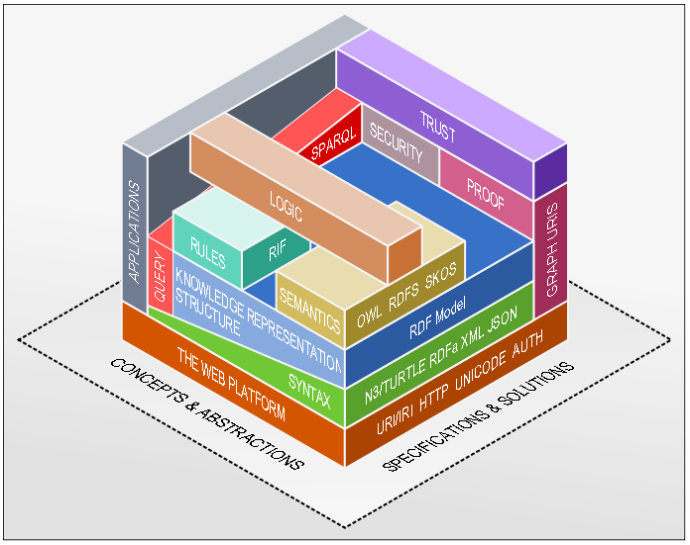


Il Web Semantico è un'estensione del nostro web dove la semantica è formale, ovvero strutturata, e ha una rappresentazione della conoscenza standardizzata.

1.5 Semantic Web

Abbiamo lo Stack Tecnologico dei Web Semantico:

- URI: Uniform Resource Identifier
  - https://dbpedia.org/page/Mount_Everest
- RDF: Resource Description Framework:




```
:Mount_Everest rdf:type dbo:Mountain .
:Mount_Everest foaf:name "Mount Everest"@en .
:Mount_Everest dbo:elevation 8848 .
:Mount_Everest dbo:namedAfter :George_Everest .
:George_Everest rdf:type dbo:Person .
:George_Everest dbo:birthdate "1790-07-04"^^xsd:date .
```

- OWL: Web Ontology Language

```
dbo:Mountain rdf:type owl:class .
dbo:Mountain rdfs:subClassOf dbo:Landform .
dbo:elevation rdf:type rdf:Property .
dbo:elevation rdfs:domain owl:Thing .
dbo:elevation rdfs:range xsd:integer .
dbo:namedAfter rdf:type rdf: Property .
dbo:namedAfter rdfs:domain owl:Thing .
dbo:namedAfter rdfs:range dbo:Person .
```

- Logic: Logic constraint + Logical rules
- SPARQL: Protocol and RDF Query Language

```
#defaultView:Timeline
SELECT DISTINCT ?mountain ?mountain Label ?person
?personLabel ?date ?image
WHERE {
?mountain wdt:P31 wd:Q8502.
?person wdt:P20 ?mountain.
?person wdt:P570 ?date.
OPTIONAL {?person wdt:P18 ?image.}
SERVICE wikibase:label
{bd:serviceParam wikibase:language "en, de, fr, es, it"
}
```

Basic Semantic Technologies

2.1 How to Represent Simple Facts with RDF

RDF si basa su Triple:

- Soggetto: URI
- Predicato: URI
- Oggetto: URI/Letterale

```
<http://dbpedia.org/resource/Greenhouse_effect><http://dbpedia.org/ontology/discoveredIn> "1824" .
```

Creando molte triple riusciamo a costruire dei grafi.

Possiamo avere:

- Individui: ovvero tutte le entità. Sono concetti come "risorse".
  - <http://dbpedia.org/resource/Greenhouse_effect>
- Classi: raccolte di entità
  - <http://dbpedia.org/category/Climate_change> .
- Letterali: Date, coordinate, costanti
  - "47.798599"^^xsd:float .
- Proprietà: aggettivi, posizioni, campi
  - <http://dbpedia.org/ontology/country>
- Vocabolari/Ontologie: base di conoscenza
  - <http://www.w3.org/2003/01/geo/wgs84_pos#

2.2 RDF Turtle Serialization

Turtle permette shortcut e abbreviazioni per migliorare la leggibilità.

@prefix associa il prefisso all'URI
@base fornisce URI da complementare a tutti gli elementi

```
@prefix dbo: <http://dbpedia.org/ontology/> .
@base <http://dbpedia.org/resource/> .
<Greenhouse_effect> dbo:discoveredIn "1824" .
<Greenhouse_effect> dbo:discoverer <Joseph_Fourier> .
```

Ogni tripla termina con un punto.

Se una tripla termina con un ; indica che la prossima tripla ha come soggetto il precedente.

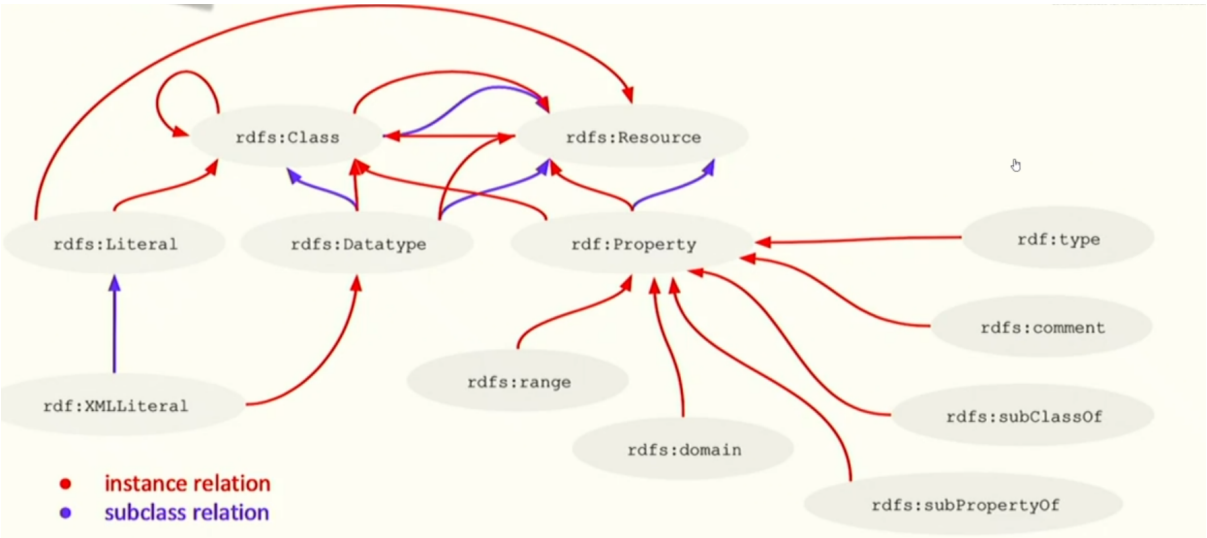
Se una tripla termina con una , indica che la prossima tripla ha come soggetto e predicato il precedente.

Il nostro limite ora è che abbiamo bisogno di più espressività a livello semantico.

2.3 Model Building with RDFS

RDF schema permette di:

- Definire classi attraverso *rdfs:Class* che sono istanziate con rdf:type
- Definire il concetto di proprietà con rdf:Property, applicando restrizioni attraverso rdfs:domain e rdfs:range
- tutto è risorsa rdfs:Resource
    - rdfs:Class rdf:type rdfs:Resource .
    - rdf:Property rdf:type rdfs:Resource .
    - rdfs:Literal rdf:type rdfs:Resource .
    - rdfs:XMLLiteral rdf:type rdfs:Resource .
    - rdfs:Datatype rdf:type rdfs:Resource
- Possiamo anche definire delle rdfs:subClassOf e subPropertyOf



Osservazione: type e property, così come altri elementi, sono presenti anche in RDF standard e infatti hanno prefisso rdf e non rdsf.

Un altra proprietà utile è rdfs:label che permette di definire un nome di risorsa più leggibile rispetto al suo ID.

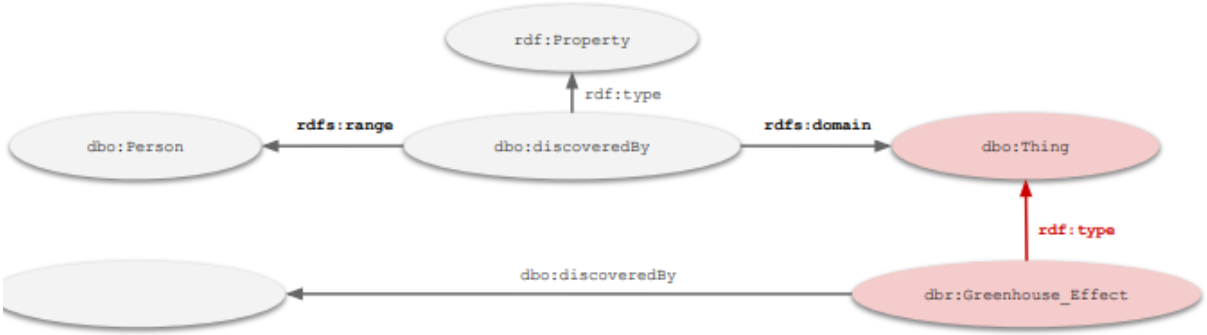
2.4 Logical Inference with RDF(S)

RDFS è basato su semantica formale che ci permette di creare inferenze logiche e valide. Sarà limitato, per migliorare vedremo le Ontologie.

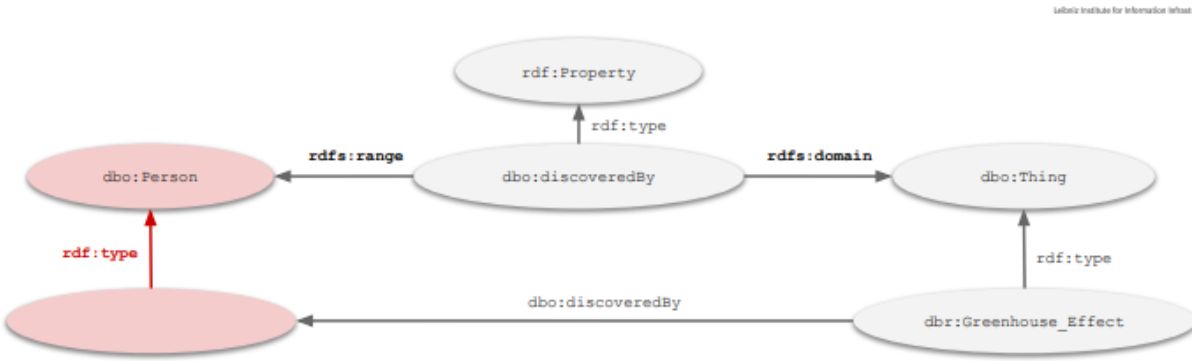
Abbiamo:

- Deduzione di nuovi fatti da una gerarchia di classe

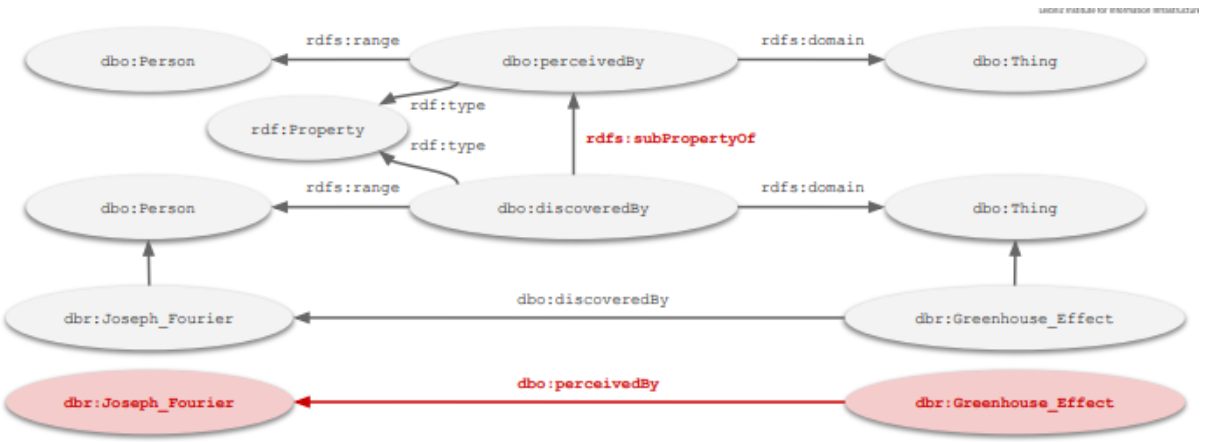
- Deduzione di nuove entità di appartenenza dal dominio della proprietà



- Deduzione di nuove entità di appartenenza dal codominio della proprietà



- Deduzione di nuovi fatti dalle relazioni di sotto proprietà



Querying RDF(S) with SPARQL

3.1 How to Query RDF(S)

SPARQL (Standard Protocol And RDF Query Language) è:

1. un Declarative Query Language per grafi RDF. (Quella di cui ci occupiamo)
2. un Protocol Layer per usare SPARQL via http.
3. una Specifica di Formato per Output XML per SPARQL queries.
4. Standard W3C ispirato da SQL

La prima cosa che bisogna sapere per fare una query è lo schema, ovvero la mappa della tabelle. In RDF le tabelle non esistono, quindi per cercare quello di cui ho bisogno devo:

1. Avere un estratto della base di conoscenza oppure accedere e vedere cosa contiene.
2. Sperare che chi l'ha composta abbia lavorato in maniera "sistematica".

3.2 Complex Queries with SPARQL

Fare una queries significa esprimere una richiesta delle informazioni che si vogliono ottenere, espresse sotto forma di variabili libere, hanno sintassi ?string. La Query SPARQL è uno statement, mentre i risultati SPARQL sono una tabella

SPARQL si basa sul concetto di serializzazione RDF Turtle e pattern matching su grafi.

Un Graph Pattern è una tripla RDF che contiene variabili in qualsiasi posto specificato (Subject, Property, Object). Si possono combinare più query per un singolo match sul grafo.

Le clausole principali sono:

- `PREFIX` per specificare i namespaces
- `SELECT` per specificare le variabili di output
- `FROM` per specificare il grafo su cui effettuare le query
- `WHERE` per specificare il graph pattern che deve matchare
- `FILTER` permette di aggiungere dei filtri alla ricerca, come `LANG` che permette di selezionare la lingua di output
- `MODIFIER` come `ORDER BY, LIMIT, OFFSET, GROUP BY`

Knowledge Representation with Ontologies

4.1 A Brief History of Ontologies

Un'ontologia è una specifica formale esplicita di una concettualizzazione condivisa:

- concettualizzazione: modello astratto
- esplicito: bisogna definire i significati di tutti i concetti
- formale: capibile da una macchina
- condivisa: consenso sull'ontologia.

Una Ontologia può essere rappresentata in due aree:

- Terminologica (Classi): assetto profondo di quello che comprende l'area
- Asserzionale (Istanze): assetto profondo di quello che posso fare all'interno

4.2 Why we do need Logic

Abbiamo bisogno di logica, e quindi RDF(S) non è uno strumento sufficiente in quanto non possiamo:

- Esprimere la località di proprietà globali.
    - Esempio: Cibo può essere composto di sottoclassi Verdure e Carne, ma ci sono animali che mangiano solo una dei due tipi.
- Le relazioni non possono esprimere classi disgiuntive.
    - Esempio: Vivi e Morti sono sottoclassi di persone e sono disgiunte
- Combinazioni di classi definiscono una nuova classe. Nuove classi contengono solo membri dalla combinazioni di vecchie classi
    - Esempio: Road User ha come sottoclassi Motoris, Motorcyclist, Pedestrian, Cyclist ecc.
- Esprimere attributi di cardinalità
    - Esempio: Ogni umano in generale ha due genitori
- Esprimere proprietà speciali come transitività, unicità, inversi
- Possibilità di negare, e non genera in automatico una contraddizione.

4.3 First Steps in OWL

OWL è un frammento di FOL. Ci sono varie forme di OWL, a diversi livelli di rappresentazione semantica e di conseguenza computabilità.

Basic Building Blocks:

- OWL namespace: `@prefix owl: <http://www.w3.org/2002/07/owl#>`
- Sintassi Turtle
- Gli assiomi OWL consistono in uno dei tre seguenti blocchi:
    - Classi, che sono le classi in RDFS

    - Individui che sono le istanze in RDFS
    - Proprietà che sono le proprietà in RDFS

Esistono due classi predefinite:
- owl:Thing classe che contiene tutti gli individui
- owl:Nothing è una classe vuota.
- Esempio - Definizione di una classe: `GreenhouseGas a owl:Class .`

Gli individui possono essere definiti:
- attraverso l'appartenenza a una classe
    - `:JosephFourier a :Person`
- senza appartenere ad una classe e quindi come individuo nominato:
    - `:HaraldSack a owl:NamedIndividual`

Esistono due tipi di proprietà:
- Datatype properties, che riguardano le misurazioni
    - `:discoveredIn a owl:DatatypeProperty`
- Object properties, riguardano le altre caratteristiche di un oggetto
    - `:discoverer a owl:ObjectProperty;`

Per la gerarchia non specifichiamo una nuova proprietà ma usiamo quella di RDF.

In OWL qualsiasi cosa potrebbe essere potenzialmente identica se non specifichiamo esplicitamente lo stato di differenza. Abbiamo owl:disjointWith

Per specificare invece uguaglianze abbiamo:
- per gli individui `owl:sameAs`
- per le classi: `owl:equivalentClass`

Riprendendo le due aree in OWL possiamo definire:
- OWL TBox : terminologica, ovvero intensionale (Schema, Classi)
- OWL ABox: asserzionale, ovvero estensionale (Tuple, Istanze)

4.4 More OWL

In OWL si possono definire degli insiemi in maniera estensionale.

```
:x a owl:Class ;
     owl:oneOf
     ( :y :w :z)
```

In OWL si possono utilizzare predicati logici basati su predicati insiemistici per strutturare le query:
- And →owl:intersectionOf
- Or → owl:unionOf

- Negation → owl:complementOf

Si possono applicare restrizioni per descrivere classi complesse in OWL:

- restrizioni su valori:
    - owl:hasValue applica un determinato valore
    - owl:allValuesFrom specifichiamo un insieme di valori che deve deve essere vero per tutti gli elementi
    - owl:someValuesFrom specifichiamo un insieme di valori che deve essere vero per almeno 1 elemento
- restrizioni su cardinalità:
    - owl:cardinality specifica esattamente la cardinalità
    - owl:minCardinality specifica la minima cardinalità
    - owl:maxCardinality specifica la massima cardinalità

Search and Plan

Search Problems

General Problem Solver: abbiamo due componenti:

- Generate component: genera scelte
- Test component: decide se sono scelte che desideriamo o rifiutiamo

1. Prima fonte di inefficienza: non riesce a testare tutte le condizioni generate
2. Seconda fonte di inefficienza: non riesce a generare tutte le condizioni perchè sono infinite

Applicazione: Risolvere puzzle.

- Condizione Desiderata: arrivare a una condizione finale in modo veloce o con meno mosse.
- Condizione Iniziale: partiamo da una configurazione random iniziale.
- Scelte: sequenze di cambiamenti (complesse).
- Azione: funzione che permette di passare da una condizione ad una successiva.

Un problema di ricerca consiste in uno spazio di stati, una funzione successore che prende uno stato e un'azione e restituisce un nuovo stato, uno stato iniziale e uno o più stati obiettivo.

La soluzione di un problema di ricerca è una sequenza di azioni che, applicata allo stato attuale tramite la funzione successore, trasforma lo stato iniziale in uno stato obiettivo.

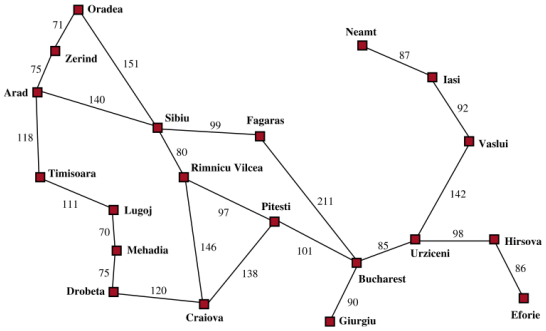
Una soluzione è ottimale se minimizza i costi totali delle azioni compiute.

Un albero di ricerca è un albero che ha come radice lo stato iniziale e ciascun nodo figlio è il risultato dell'applicazione di una diversa azione al nodo padre. Per moltissimi problemi non è possibile costruire l'albero di ricerca completo, poiché richiederebbe spazio e tempo esponenziali. In alcuni casi l'albero è persino infinito.

Un algoritmo di ricerca si definisce completo se riesce a trovare almeno una soluzione se almeno una ne esiste, mentre si definisce ottimale se riesce a trovare almeno una soluzione ottimale.

Esempio:

- Condizione Desiderata: Arrivare a Bucharest
- Condizione Iniziale: Partiamo da Arad
- Scelte: route1, route2, ..., route N+1
- Azione: Andare in una città adiacente



Algoritmo banale: l'efficienza del processo è dettata dai luoghi in cui possiamo agire, sottolineati nel codice.

```
PartialSolutionSet =[Initial_State]
While not(PartialSolutionSet.empty()):
      solution=extractSolution(PartialSolutionSet)
      new_state=getState(solution)
      Successors=[]
      for action in actions_valid_in(new_state):
            Successors.append(successor(new_state,action)
      for state in Successors:
            new_solution=solution.copy().append(state)
            if goalcondition(new_state) then: return "victory"
            PartialSolutionSet.insert(new_solution)
```

Invece di generare tutte le risposte, potremmo provare a seguire il percorso tramite un albero.

Principali varianti:

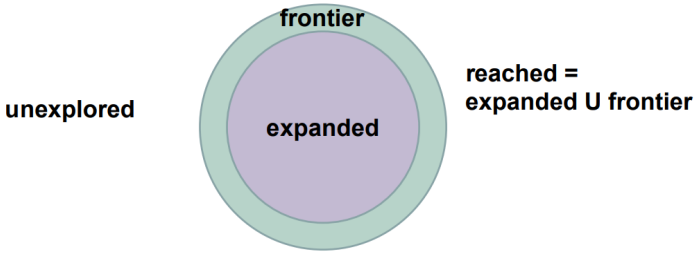
- Quale nodo foglia espandere successivamente?
- Dobbiamo verificare la presenza di stati ripetuti?

Vediamo nella prossima lezione.

Informed Research

Definizioni:

1. La frontiera separa la regione espansa dalla regione inesplorata del grafico dello spazio degli stati
2. Espansione di un nodo di frontiera: Sposta un nodo da frontiera a espanso e aggiunge nodi da inesplorati alla frontiera.



Backtracking search: implementazione ricorsiva che visita l'intero albero di ricerca per un dato problema fino alla profondità D, partendo da s ed esplorando ricorsivamente tutti gli s. Alla fine restituisce il miglior p trovato. Non termina se lo spazio è infinito o maggiore di D. Una frontiera è uno stack LIFO.

- Spazio: $O(bD)$
- Tempo: $O(b^D)$

```
def backtrackingSearch(s, path, D):
If IsEnd(s):
      update minimum cost path
For each action a ∈ Actions(s):
      Extend path with Succ(s, a) and Cost(s, a)
      Call backtrackingSearch(Succ(s, a), path,D)
Return minimum cost path
```

Depth-First-Search: L'algoritmo Depth First Search è come BS, ma si ferma non appena trova il primo p:

- Spazio: $O(bD)$
- Tempo: $O(b^D)$ ma nel caso peggiore, ma potrebbe essere molto migliore se le soluzioni fossero facili da trovare.

```
def DepthFirstSearch(s, path,D):
If length(path)>D:
      return NULL
If IsEnd(s):
```

```
      return path
For each action a ∈ Actions(s):
      Extend path with Succ(s, a)
      Call DepthFirstSearch (Succ(s, a), path,D)
```

L'algoritmo di Breadth First Search parte da s ed esplora la sua frontiera. Se nella frontiera si trova la I soluzione, allora si arresta, altrimenti ripete lo stesso procedimento per ogni nodo della frontiera. Per ogni stato abbiamo s azioni, e la soluzione ha d azioni. Una frontiera è una coda FIFO.

- Spazio: $O(b^d)$ molto peggiorato
- Tempo: $O(b^d)$ ma meglio in quanto dipende da d, profondità soluzione e non D, profondità massima

```
PartialSolutionSet =[Initial_State]
While not(PartialSolutionSet.empty()):
        solution=head(PartialSolutionSet)
        new_state=getState(solution) Successors=[]
        for action in actions_valid_in(new_state):
                Successors.append(successor(new_state,action)
        for state in Successors:
                new_solution=solution.copy().append(state)
                if goalcondition(new_state) then:
                        return "victory",new_solution
                PartialSolutionSet.append(new_solution)
```

Iterative Deepening Search: unisce DFS e BFS per ottenere i vantaggi di spazio di DFS e la completezza di BFS. Itera DFS partendo da d = 1 fino ad arrivare a d = D, fermandosi non appena trova un p.

- Spazio: $O(d)$ migliorato
- Tempo: $O(b^d)$

Uniform Cost Search: esploriamo i cammini a costo minimo. Parte da s_1 ed esplora la sua frontiera. Successivamente esplora il nodo il cui il cammino da s_1 abbia costo minore. Si ripete questa procedura fino a che non si è esplorato tutto T entro D, esplorando sempre per primi i nodi il cui cammino da s abbia costo minore. Quest'ultima nota implica che è necessario mantenere in memoria tutte le frontiere contemporaneamente per scegliere sempre il nodo con cammino minore. Alla fine se esiste soluzione, UCS avrà trovato quella ottimale.

Proviamo a cambiare il goal del nostro problema esempio: Arrivare a Bucharest evitando di passare da 3 città consecutive che contengono nel nome la lettera i.

Idea: Guida informata, dove la ricerca va verso l'obiettivo invece che ovunque

Euristica: stima della distanza dall'obiettivo più vicino per ogni stato. Potrebbe essere la distanza euclidea, la distanza Manhattan (i cateti dell'ipotenusa) ecc.. Per ogni problema possiamo avere un'euristica diversa.

Un'euristica non sono per forza giuste, la riteniamo ammissibile se è minore o uguale all'effettivo costo.

Greedy Algorithm: espandi un nodo che riteniamo sia più vicino allo stato obiettivo. Se almeno una soluzione esiste, viene trovata, ma non è detto che sia quella ottimale. Molto dipende dalla bontà dell'euristica e dal costo effettivo di a da s che non viene considerato, dato che si tiene conto solo di h(s).

A* search: come GS, ma con 2 differenze:

- il nodo esplorato non è quello che ha h(s) minimo, bensì quello che ha g(n) + h(s) minimo, ovvero la somma tra l'euristica e costo effettivo.
- la funzione di euristica deve restituire valori che siano sempre ≤ degli effettivi costi ottimali.

Pro:

- Ricerca albero: A* è ottimale se l'euristica è ammissibile
- Ricerca nel grafico: A* ottimale se l'euristica è coerente
- Coerente implica Ammissibile.
- La maggior parte delle euristiche ammissibili naturali tendono ad essere coerente, soprattutto se proveniente da problemi rilassati.

Cons:

- A* mantiene in memoria l'intera regione esplorata e quindi esaurirà lo spazio prima di trovare la risposta
- Esistono varianti che utilizzano meno memoria

	BS	DFS	BFS	IDS	UCS	GS	A*
Spazio	$O(D)$	$O(D)$	$O(b^d)$	$O(d)$	$O(bD)$	$O(b^d)$	$O(b^m)$
Tempo	$O(b^D)$	$O(b^D)$	$O(b^d)$	$O(b^d)$	$O(b^d)$	$O(b^d)$	$O(b^m)$
Completo	Se |S| != inf	Se |S| != inf	Si	Si	Si	Si	Si
Ottimale	Si	No	No	No	Si	No	Si

`D max depth, d sol depth, b ramificazione, m num max nodi da esplorare`

Classical Planning

Un problema rilassato è caratterizzato da minori condizioni vincolanti rispetto a un problema di riferimento. Spesso le euristiche sono soluzioni a tali problemi.

Un problema di planning è simile a un problema di ricerca, ma è più fortemente connesso al concetto di stati, azioni e dimensione temporale. La struttura di un problema di planning differisce da quella di un problema di ricerca in quanto vengono usati dei linguaggi di pianificazione per rappresentare lo spazio di stati e azioni in modo più ricco e generalizzabile.

Il Classical Planning si occupa di studiare quei problemi in cui lo stato iniziale è completamente conosciuto e le azioni sono deterministiche. Essendo le azioni deterministiche, dato lo stato e data l'azione avremo sempre lo stesso risultato.

Per concettualizzare un problema di planning si utilizzano i concetti di agente e ambiente. L'agente può percepire l'ambiente e compiere delle azioni che ne modificano lo stato.

Se riusciamo a formulare il problema, riusciamo sempre ad avere un risultato. Ci serve però una buona sintassi e semantica per arrivare a formulare il problema.

Poiché per ogni istante di tempo t l'agente si può trovare o meno in uno stato e può compiere o meno un'azione, il numero massimo di ambienti che si possono incontrare è $O(2^{NT})$. Molto grande.

Codificare l'interezza delle possibilità diventa velocemente non fattibile, per questo si usano i linguaggi di pianificazione per rappresentare tutte le possibilità sensate in modo sintetico.

Un problema di planning in STRIPS è rappresentato dalla quadrupla $P = (F, O, I, G)$:

- F è l'insieme degli atomi,
- O l'insieme delle azioni,
- $I \subseteq F$ gli atomi iniziali
- $G \subseteq F$ gli atomi goal.

L'obiettivo è, partendo dall'insieme di atomi I, compiere una sequenza di azioni $A \subseteq O$ tali da trasformare gli atomi di I in quelli di G

Le azioni $o \in O$ sono rappresentate da:

- $p(o)$ le precondizioni affinché si possa compiere o
- $d(o)$ gli atomi rimossi dal compimento di o
- $a(o)$ gli atomi aggiunti dal compimento di o

I costi di ciascuna azione sono fissati a 1.

In STRIPS è anche possibile utilizzare predicati, variabili e tipi per sintetizzare la scrittura di atomi e azioni. L'atto di rimpiazzare le variabili con un valore si chiama istanziazione. Le cosiddette grounded action si ottengono istanziando le variabili delle azioni.

PDDL (Planning Domain Description Language) è un linguaggio di pianificazione successivo a STRIPS: ha una sintassi più rigida, ma con maggiore espressività.

In PDDL, i problemi di planning vengono specificati in 2 parti:

- Dominio: schema di azioni e atomi (predicati) con argomenti tipizzati
- Istanza: stato iniziale, obiettivo e costanti per ogni tipo

Per risolvere un problema di pianificazione classica abbiamo più approcci, vediamo i due che sembrano migliorare meglio a livello computazionale:

- Planning con Ricerca Euristica: il problema P è risolto trovando la path sul grafo associato al modello S(P) con euristica derivata da P
- Planning a soddisfacibilità (SAT): sono efficienti, NP completi ma molto grandi.

FF (Fast Forward) Planner: è un Planning con Ricerca Euristica basato su A*..
Esegue la ricerca nello spazio degli stati in avanti (A*) con un euristica rilassata h_{FF}:

1. Costruisce il problema rilassato
2. Risolve il problema rilassato in tempo polinomiale
3. Usa la lunghezza del problema rilassato come euristica

Il problema è che l'euristica è inammissibile, anche se molto precisa.

Probabilistic Planning

In generale non è vero che un agente possa avere conoscenza completa dello stato dell'ambiente.

L'idea chiave nel Probabilistic Planning è quella di codificare l'incertezza, l'ignoranza, dell'agente. Per farlo si basa sulla teoria della probabilità: l'agente ha dei belief che possono variare a seconda della nuova conoscenza ottenuta tramite i sensori.

Il valore di belief β in una certa proposizione p è 0 ≤ β ≤ 1. 0 significa che l'agente è certo che p sia falsa, mentre 1 significa certezza nel fatto che p sia vera. Se la probabilità β è invece 0 < β < 1 significa che β è la misura di quanto l'agente sia sicuro della verità di p.

$$Formula\ di\ Bayes\colon\ P(x \mid y) = \frac{P(y \mid x)\,P(x)}{P(y)} = \frac{probability * prior}{evidence}$$

Possiamo normalizzare, così ci serve conoscere una probabilità in meno:

$$\eta = \frac{1}{\sum_x P(y \mid x) P(x)}$$

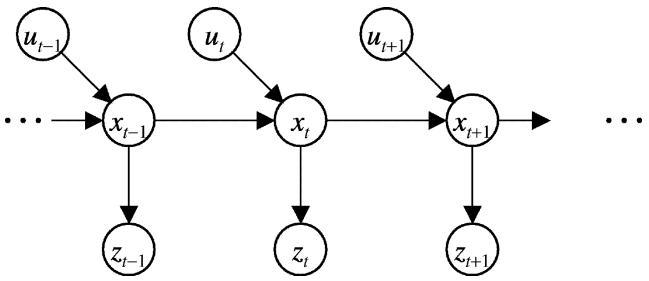
Nella Markov Localization Inizialmente l'agente non conosce lo stato attuale. Attraverso i sensori l'agente percepisce parte dello stato attuale e ricomputa la probabilità dello stato in cui si trova tramite la seguente formula, considerando i possibili rumori:

$$Bel(x_t) = \eta\ P(z_t \mid x_t) \int P(x_t \mid u_t, x_{t-1})\ Bel(x_{t-1})\ dx_{t-1}$$

Assunzione di Markov: lo stato successivo dipende solamente dallo stato precedente e dall'azione che dobbiamo compiere.

Markov Decision Process: Le differenze con un problema di ricerca sono che:

- la transizione da uno stato all'altro è probabilistica
- invece di cercare di minimizzare i costi si cerca di massimizzare le ricompense.



Problema: Per ogni posizione dovremmo compiere un integrale, considerando che lo spazio non è discreto è molto dispendioso. Abbiamo 3 modi per rappresentare lo spazio:

- Grid Based Approaches: Usata da Markov, utilizza una griglia discreta.
- Particle Filters: Usata dai localizzatori Montecarlo, forniscono una stima continua di dove potrebbero trovarsi i punti
- Kalman Tracking: fornisce l'area continua con più punti.

Probabilistic Planning: Come un agente dovrebbe agire in un ambiente in modo da massimizzare la ricompensa a lungo termine **dato il modello del mondo**, ovvero quanto è positivo per l'agente trovarsi in un determinato stato?

Reinforcement Learning: come un agente dovrebbe agire in un ambiente in modo da massimizzare la ricompensa a lungo termine iniziando **senza alcuna conoscenza del mondo**

Probabilistic Planning & Reinforcement Learning: Basic Setup:
- Stati discreti
- Azioni discrete
- Politiche π: definiscono quale azione intraprendere in ciascuno stato.
- Tutti gli obiettivi possono essere descritti dalla **massimizzazione** della ricompensa cumulativa attesa

Quale sia una policy ottimale varia largamente in base alla funzione di ricompensa.

Un agente che cerca di massimizzare la ricompensa cumulativa ma potrebbe trovarsi in una situazione di indecisione nel caso di sequenze di azioni la cui ricompensa totale sia pari.

Per risolvere questa indecisione si introduce il discount factor γ per far diminuire il peso che il valore di azioni future ha rispetto alle azioni più immediate.

Considerando che il mondo è probabilistico avremo che un segnale di reward avrà la seguente formula:

$$\mathcal{R}_t = \mathrm{E}[r_t] + \gamma \mathcal{R}_{t+1}$$

Ogni aspettativa dipende dallo stato attuale, dall'azione e dalla politica:

$$\mathrm{Q}^{\pi}(s,a) = \mathcal{R}_t | (s_t, a_t, \pi) = \mathrm{E}[r_t | s_t, a_t] + \gamma \mathcal{R}_{t+1} | s_t, a_t, \pi$$

Q_{π} (s,a) è la funzione che mappa ogni azione a valore, noi vogliamo trovare l'azione a che massimizza tale funzione.

Per trovare la migliore Q, usiamo la Bellman Equation: il valore di uno stato è la ricompensa attesa della transizione successiva più il valore con il fattore discount dello stato successivo, assumendo che l'agente scelga l'azione ottimale.

Value Iteration: si ripete il Bellman Update finché non converge, e sappiamo che diverge grazie all'effetto contrazione, ovvero nello spazio stiamo avvicinando i due elementi. Infatti converge esponenzialmente.

$$U_{k+1}(s) \leftarrow \max_a \sum_{s'} P(s' \mid a,s) \, [R(s,a,s') + \gamma U_k(s')]$$

Complessità per ogni iterazione: $O(S^2A)$.

Il problema è che stiamo assegnando una funzione e noi vogliamo una politica: effettuiamo la policy extraction:

$$\pi_U(s) = \operatorname{argmax}_a \sum_{s'} P(s' \mid a,s) \, [R(s,a,s') + \gamma \boldsymbol{U}(s')]$$

La qualità di una politica π è misurata dalla policy loss:

$$|| \, \boldsymbol{U}^{\pi} - \boldsymbol{U}^{*} \, ||$$

Problemi della Value Iteration:

- Lento – $O(S^2A)$ per iterazione
- Il "massimo" in ogni stato cambia raramente
- La politica spesso converge molto prima dei valori

Introduciamo allora il concetto di Policy Iteration: I valori convergono prima.

1. Valutazione della politica: calcolare il valore $U_{\pi k}$ per la politica attuale π_k
2. Miglioramento della politica: estrai la nuova politica implicita π_{k+1} da $U_{\pi k}$

Fixed Policies: Se non facessimo più ciò che dice l'azione ottimale, ma ciò che dice π fissando le politiche $\pi(s)$, l'albero sarebbe più semplice in quanto avremmo solo un'azione per stato.

- Idea 1: trasformare le equazioni ricorsive di Bellman in aggiornamenti
- Idea 2: senza i massimi, le equazioni di Bellman sono solo un sistema lineare

Sia iteration value che policy iteration calcolano tutti i valori ottimali e sono DP.

- Calcolare i valori ottimali: iteration policy/iteration value
- Calcolare i valori di una policy: policy evaluation
- Trasformare i valori in una policy: policy extraction

Esistono altri algoritmi oltre al Q-Learning, che però è il più utilizzato, per applicare Reinforcement Learning:

- SARSA $\boldsymbol{\lambda}$-learning: una variante del Q-Learning che aggiorna ad ogni iterazione la tabella Q non solo della singola cella $[s, a]$, bensì anche le ultime k celle che hanno portato alla situazione attuale. Questo è fatto per

velocizzare la convergenza all'ottimo e dare importanza anche alla sequenza di azioni che ha permesso di ottenere la situazione attuale.
- Deep Q-Learning: applicazione di una Rete Neurale Profonda per approssimare i valori della tabella Q per tabelle che sarebbero altrimenti troppo grandi da computare esplicitamente.
- Monte Carlo Tree Search: approccio leggermente diverso dagli altri in quanto si basa su un albero di ricerca per trovare la ricompensa massima conosciuta, per poi, una volta raggiunto uno stato terminale, far back propagation sui nodi che hanno fatto parte del cammino.

Intelligenza Artificiale Subsimbolica

Apprendimento Supervisionato

Basic concepts

Abbiamo avuto un ritorno di attenzione sull'apprendimento automatico in quanto prima era difficile ottenere dei risultati buoni, ora sono migliorati. I 3 motivi principali sono:

- Crescita potenza computazionale (hw dedicato TPU, o anche GPU che lavorano in modo parallelo).
- Disponibilità di dati di immagini annotate
- Framework open source

Machine Learning si divide in due sezioni:

- Supervisionato: i dati nel training set sono etichettati, ovvero già classificati. Lo scopo è quindi quello di apprendere una funzione che mappi le caratteristiche dei dati nelle loro classi. Le tecniche principali sono la Regressione e la Classificazione
- Non Supervisionato: i dati non sono etichettati e lo scopo sarà quindi apprendere quali e quanti classi ci siano e come assegnare l'appartenenza. Le due tecniche principali sono:
    - Clustering: algoritmo che prova a trovare delle istanze di gruppi nel dataset in base a caratteristiche dello spazio dei dati
    - Anomaly Detection: il sistema deve generalizzare il dato abituale per trovare dati anomali.

In realtà è più complicato di così, abbiamo molti altri campi e moltre altre tecniche come:

- Semi Supervised ML: Approccio dove abbiamo dei dati etichettati e molti non etichettati. Esempio: Face Recognition nelle gallerie
- Reinforcement Learning: Approccio completamente diverso dove l'agente viene posto in un ambiente, impara tutte le azioni possibili.
- Instance Based learning: Approccio in cui non ci sono i modelli o un processo di apprendimento, ma esclusivamente i dati. Una nuova istanza si adatta in base a statistiche apprese nella base di conoscenza.

Tecnica di Programmazione:

- Approccio Tradizionale: Scriviamo del codice, definiamo operazioni.
- Approccio Machine Learning: Selezione, raccogliere e produrre dati per addestrare un algoritmo di apprendimento automatico che viene valutato sulla base dei dati e rappresenta lui la soluzione al problema. Possiamo usare tecniche come Regressione.

Spesso ML non implica un continuo processo di apprendimento. Per averlo bisogna avere uno step precedente all'allenamento in cui aggiorniamo i dati, anche se questo non porta per forza ad un miglioramento.

Tipicamente le forme di apprendimento sono:

- Batch learning, allenare usando i dati disponibili, Spesso i tempi sono lunghi quindi vengono fatti offline.
- Online learning, il sistema incrementa alimentando le istanze di dati sequenzialmente. Bisogna specificare quanto velocemente deve adattarsi ai nuovi dati.

Main Challenges ML:

- Insufficienza di quantità di data training
- Data training non rappresentativi
- Data di scarsa qualità
- Feature irrilevanti
- Overfitting: i modelli producono buone previsioni rispetto ai data training ma hanno prestazioni scarse su nuovi campioni.
- Underfitting: si verifica quando il modello di machine learning non è ben adattato al set di training

Decision Trees

Data: Set di record descritti da:

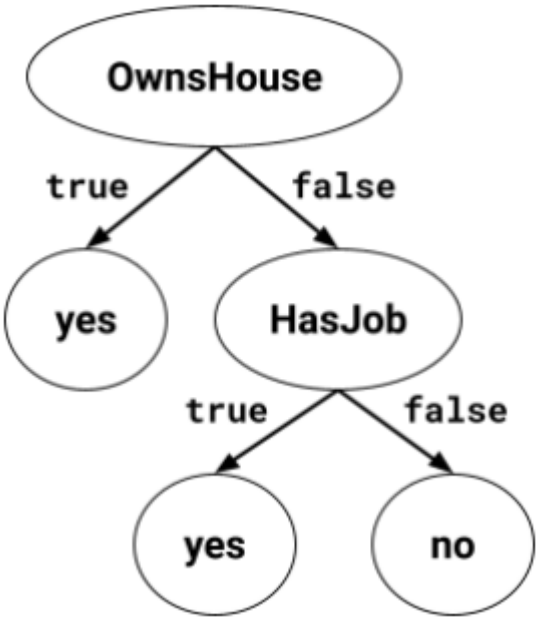
- k attributi: A_1, A_2, ..., A_k
- a classi: Ogni esempio è etichettato con una classe predefinita.

Obiettivo: creare un modello di classificazione dei dati che può essere utilizzato per prevedere le classi di nuove istanze (futuri o di prova).

Assunzione: la distribuzione degli esempi di training è identica alla distribuzione degli esempi di test. Spesso violata. Per raggiungere una buona accuratezza, gli esempi di training devono essere sufficientemente rappresentati nei data dei test.

Gli alberi di decisione sono una delle tecniche più usate nell'ambito della classificazione, poiché offrono un'ottima accuratezza ed efficienza.

Un albero di decisione è un albero in cui gli archi sono rappresentati da valori di features, i nodi foglia sono raggruppamenti per classi, mentre gli altri nodi sono raggruppamenti per features.



In un albero di decisione, quindi, ogni cammino dalla radice a una foglia rappresenta una regola di decisione.
È importante notare che non esiste un unico albero di decisione e che trovare l'albero migliore, ovvero con meno nodi e più accurato, è un problema NP-complesso. Tutti gli attuali algoritmi di costruzione di alberi di decisione sono algoritmi euristici.

Un algoritmo base per ricostruire l'albero top-down usa un algoritmo greedy divide-and-conquer.

L'algoritmo prevede in input:

- D è il set di individui considerati dove inizialmente è il training set ma col tempo creeremo subset
- A è il set di attributi che mancano da analizzare, all'inizio sono tutti ma alla fine è l'insieme vuoto
- T è la foglia che sta venendo generata per sostituire i rimanenti individui del percorso analizzato

L'algoritmo è così strutturato:

- caso base: se D contiene esempi della stessa classe allora siamo in un branch puro dell'albero e quindi possiamo fare una foglia T etichettata con il nome della classe
- caso base: se A è l'insieme vuoto allora non possiamo splittare i dati e quindi creiamo una foglia T etichettata con la classe più frequente
- passo ricorsivo: calcolo l'impurità di D e l'impurità relativa A per ogni attributo. Seleziono A_g, ovvero quella che dà maggiore riduzione di impurità e credo un nodo di decisione T. con m valori e quindi m subset. per ogni m, creiamo un branch T_j e richiamiamo la funzione diminuita del valore (D_j, A-{A_g}, T_j)
- eccezione: Se la riduzione di impurità è troppo piccolo e non vale la pena splittare, andiamo avanti come se non ci fosse possibilità di splittare i dati.

La chiave per costruire un buon albero di decisione è scegliere il giusto attributo nel quale eseguire la branch, riducendo così le impurità il più possibile. L'euristica usata in C4.5 è di scegliere l'attributo con massimo Information Gain o Gain Ratio.

La Teoria dell'Informazione fornisce una base matematica per questo. Fornisce una misura del valore di un'informazione misurato in bit.

L'entropia misura l'impurità di un dataset D, fornendo valori compresi tra 0 e 1, dove più l'entropia è bassa più il dataset è puro. La formula dell'entropia è:

$$entropy(D) = -\sum_{j=1}^{|C|} Pr(cj)ln * Pr(cj)$$

Presa la radice A_i con v valori eseguiamo v subset con ciascuno la sua entropia:

$$entropy\ Ai\ (D) = \sum_{j=1}^{v} |Dj| \div |D| * entropy(Dj)$$

Il guadagno di informazione si ottiene selezionando un attributo A_i e il dataset D:

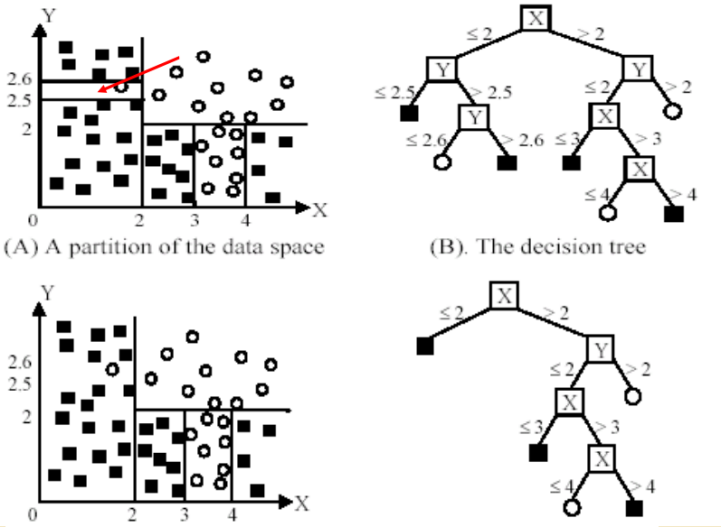
$$gain(D,\ Ai) = entropy(D) - entropy\ Ai\ (D)$$

Per trovare la soglia migliore che divide il subset abbiamo diversi metodi:
- Utilizzare il guadagno di informazione o il rapporto di guadagno
- Ordinare tutti i valori di un attributo continuo in ordine crescente
- Provare tutte le soglie possibili e trovare quella che massimizza il guadagno

Overfitting:
- Buona precisione sui dati di addestramento ma scarsa sui dati di test
- Sintomi: albero troppo profondo o troppe ramificazioni, alcuni valori potrebbero essere presi come anomalie dovute a rumori o valori anomali

Due approcci per evitare l'overfitting:
- Pre-potatura: interrompere anticipatamente la costruzione degli alberi
    - Difficile decidere quando interrompere perché non sappiamo cosa potrebbe succedere se continuiamo a far crescere l'albero.

- Post-potatura: rimuovere rami o sottoalberi dall'albero completamente cresciuto"
    - Questo metodo è quello comunemente usato.
    - C4.5 utilizza un metodo statistico per stimare gli errori su ciascun nodo per la potatura.
    - È possibile utilizzare un set di convalida anche per la potatura.



Altri problemi nell'apprendimento degli alberi decisionali

- Passaggio dall'albero alle regole alla potatura
- Gestione dei valori mancanti
- Gestire distribuzioni asimmetriche
- Gestire attributi e classi con differenti costi.
- Costruzione di attributi

Evaluation of classifiers

Per valutare un metodo di classificazione abbiamo i seguenti valori:

- accuratezza: numero di classificazioni corrette sul numero totale dei casi
- efficienza: tempo per costruire ed usare il modello
- robustezza: gestione di rumori e mancanza di valori
- scalabilità: efficienza a nuovi dati
- interpretabilità: possibilità di capire e fornire informazioni sul modello
- compattezza: grandezza dell'albero o dell'insieme di regole

Holdout Set è il metodo utilizzato quando il dataset D è grande: i dati disponibili D sono divisi in due subset:

- training set D_{train}
- test set D_{test} anche chiamato holdout set

N-fold Cross-Validation:

- I dati disponibili sono partizionati in n sottoinsiemi disgiunti di uguali dimensioni.
- Si utilizza ciascun sottoinsieme come set di test e si combinano i restanti n-1 sottoinsiemi come set di training per l'apprendimento.
- La procedura viene eseguita n volte, fornendo n precisioni.
- L'accuratezza finale stimata dell'apprendimento è la media delle n precisioni.
- Sono comunemente usate le forme da 5 fold e 10 fold
- Metodo utilizzato quando i dati disponibili non sono molti.

Leave-One-Out Cross Validation: Utilizzato quando il dataset è estremamente piccolo: ogni fold ha un singolo esempio di test e i restanti vengono usati nel training.

Validation Set: i dati disponibili sono suddivisi in tre sottoinsiemi:

1. training set
2. validation set
3. test set.

Il validation set viene utilizzato frequentemente per stimare i parametri negli algoritmi di apprendimento. In questi casi, i valori che danno la migliore accuratezza sono utilizzati come valori dei parametri. Il Validation Set può essere utilizzato sia nella strategia Holdout che N-Fold

I parametri vengono quindi testati più volte con valori diversi, con l'obiettivo di ottenere il miglior valore, e prendono il nome di iperparametri.

L'accuratezza (errore = 1 - accuratezza) è un unica misura e risulta scomoda nelle classificazioni che coinvolgono dati asimmetrici o sbilanciati. L'elevata accuratezza non rileva intrusioni.

La classe di interesse è comunemente chiamata classe positiva, e il resto classi negative.

Applicando un concetto di Vero/Falso otteniamo la matrice di confusione:

	Classificato Positivo	Classificato Negativo
Attuale Positivo	TP	FN
Attuale Negativo	FP	TN

TP: vero positivo, classificazione corretta di un positivo
FP: falso positivo, classificazione scorretta di un positivo
FN: falso negativo, classificazione scorretta di un negativo
TN: vero negativo, classificazione corretta di un negativo

La precisione p è il numero di esempi positivi diviso per il numero totale di esempi classificati come positivi.

Il recall r è il numero di esempi positivi correttamente classificati divisi per il numero totale di esempi positivi nel set di test.

Osservazione: precisione e recall misurano solo sulla classe positiva.

F1-score combina la precisione e il recall in unica misura: è la media armonica dei due. Il valore tende al numero più piccolo dei due.

$$p = \frac{TP}{TP+FP} \qquad r = \frac{TP}{TP+FN} \qquad F = \frac{2pr}{p+r}$$

La curva ROC (Receive Operating Characteristic) è un plot del rate dei veri positivi (TPR, è uguale alla precisione p) e il rate dei falsi positivi (FPR, 1 - TPR)

$$TPR = \frac{TP}{TP+FP} \qquad TPR = \frac{FP}{TP+FP}$$

TPR è anche chiamata sensitività, mentre la specificità è 1 - sensitività, che rappresenta il rate dei veri negativi (TNR).

L'area sotto alla curva ROC è chiamata AUC. Se l'AUC per C_i è maggiore di quella di C_j allora il primo è meglio del secondo.

- Se un classificatore è perfetto, il suo valore AUC è 1
- Se un classificatore fa tutte le ipotesi in modo casuale, la sua AUC è 0,5

K-nearest neighbor

Il metodo kNN non crea il modello dai dati di training.

Per classificare un'istanza di test d, kNN definisce l'intorno P come i k valori più vicini a d.

La stima che il nuovo individuo faccia parte di un insieme è $Pr(c_j|d) = n/k$: conta il numero n di istanze di addestramento in P che appartengono alla classe c_j.

Non è necessario tempo di training, mentre il tempo per classificare è lineare nella dimensione del set di addestramento per ogni caso di test.

Pro/Cons:

- Pro
    - kNN può gestire problemi complessi e arbitrari nei confini decisionali.
    - Nonostante la sua semplicità l'accuratezza della classificazione di kNN può essere abbastanza forte e in molti casi come accurati come nei metodi elaborati.
- Cons
    - kNN è lento nella classificazione
    - kNN non produce un modello comprensibile

Ensemble methods

Possiamo combinare più classificatori a produrre un classificatore migliore? Si

Discutiamo due algoritmi principali algoritmi:

- Bagging, se dice una cosa è quasi sempre vera ma perde alcuni casi (alta precisione e basso recall)
- Boosting, non perde casi ma c'è il rischio di creare falsi positivi (bassa precisione e alto recall)

Training per Bagging

- Crea k campioni bootstrap S[1], S[2], ..., S[k]
- Costruisce un classificatore distinto su ogni S[i] per produrre k classificatori, utilizzando lo stesso algoritmo di apprendimento.

Test per Bagging

- Classifichiamo ogni nuova istanza votando k classificatori (pesi uguali)

Funziona quando l'algoritmo di training non è molto stabile, ovvero con piccoli cambiamenti in input causiamo grandi cambiamenti in output (non mi fido del singolo), ma potremmo essere sensibili ai rumori

Training per Boosting

- Produce una sequenza di classificatori sullo stesso learner, ed ogni classificatori dipende dagli errori del precedente
- Esempi predetti in modo sbagliato sono inseriti nel successivo trainer e hanno maggiore rilevanza

Test per Boosting

- Per un caso di test, i risultati della serie dei classificatori vengono combinati per determinare la classe finale del caso di test.

Richiede anche questo che il training non sia stabile e sembra suscettibile ai rumori.

Reti Neurali

Perceptron

Il Percettrone è stato il primo modello di rete neurale artificiale. Rappresenta un classificatore binario che mappa i suoi ingressi x, i quali rappresentano vettori, in un valore di output $f(x)$

Ogni input ha un peso. La loro somma è chiamata **attivazione**:

$$activation_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)$$

Se l'attivazione è:
- Positiva, l'output sarà +1
- Negativa, l'output sarà -1

Schematicamente si ha quindi che in input il percettrone prende una serie di valori e moltiplica ciascuno per il peso ad esso associato. Successivamente si fa una sommatoria di tutti i valori ottenuti e il risultato viene passato in output.

Per calcolare l'attivazione possiamo calcolare:
- il prodotto scalare tra il vettore di valori di w e di $f(x_i)$
    - se è positivo avremo output positivo, negativo altrimenti
- l'angolo delle features e dei pesi
    - se l'angolo è maggiore di 90 gradi avremo output positivo, negativo altrimenti

I Percettroni così come descritti hanno però una grande ed evidente limitazione: possono apprendere solo **separazioni lineari**. Cosa fare quindi se il dataset non permette una tale separazione?

Creiamo più classi, dove in ogni classe avremo un vettore di peso. Calcoliamo il vettore che corrisponde al massimo prodotto scalare e selezioniamo la classe come classe di appartenenza. Il vettore peso cambierà in modo iterativo:
- Se corretto, nessuna modifica!
- Se sbagliato: abbassa il punteggio della risposta sbagliata, aumenta il punteggio della risposta giusta.

Proprietà dei percettroni:
- Separabilità: vera se alcuni parametri rendono il training set perfettamente corretto
- Convergenza: se la formazione è separabile, il percettrone finirà per convergere

- Errore vincolato: esiste un numero massimo di errori che possiamo commettere

Problemi dei percettroni:
- Rumore: se i dati non sono separabili, i pesi potrebbero creare rumori
- Generalizzazione mediocre: trova una soluzione “appena” separante
- Overtraining: la precisione del test solitamente aumenta, poi diminuisce

Sistemare i percettroni: regolare l'aggiornamento del peso per evitare questi problemi:
- MIRA*: usiamo un parametro che corregge la funzione, minimizzando il cambiamento.
- SVM: massimizziamo il margine e trovano il separatore migliore, ovvero quello con maggior distanza dai punti dei due insiemi.

Limiti percettroni:
1. Usa solamente **condizioni lineari**
2. **Non trova la soluzione** che separa meglio gli esempi positivi e negativi
3. Potrebbe impiegare **troppo tempo** per trovare le soluzioni

Una volta applicata una trasformazione appropriata (anche non lineare) alle feature in input è possibile ricondursi ad uno spazio dove una separazione lineare esiste. È però ovvio che la **scelta** di quale **trasformazione** applicare sia da ponderare attentamente, e cambia per ciascun ambito di applicazione.

Abbiamo molte features non lineari e quindi abbiamo buona flessibilità e buona potenza, ma “**incoraggiamo**” il set a seguire il **rumore**.

Non sarebbe possibile far **apprendere** alla macchina qual è la **mappatura** di volta in volta migliore? Introduciamo il Deep Learning.

Deep Learning

Il **Deep Learning** si tratta semplicemente di utilizzare più percettroni per le feature in input e usare le loro attivazioni come input ad altri percettroni. Più si stratifica la rete, più si definisce profonda e più è possibile mappare spazi in altri e trovare la corretta mappatura e separazione lineare, ovviamente al costo di più risorse richieste per l’addestramento e la computazione.

Otteniamo un N-Layer Perceptron Network, suddivisibile in 1 layer di input, 1 layer di output e tutti gli altri layer sono detti **hidden**. Ogni hidden layer rappresenta un livello di astrazione.

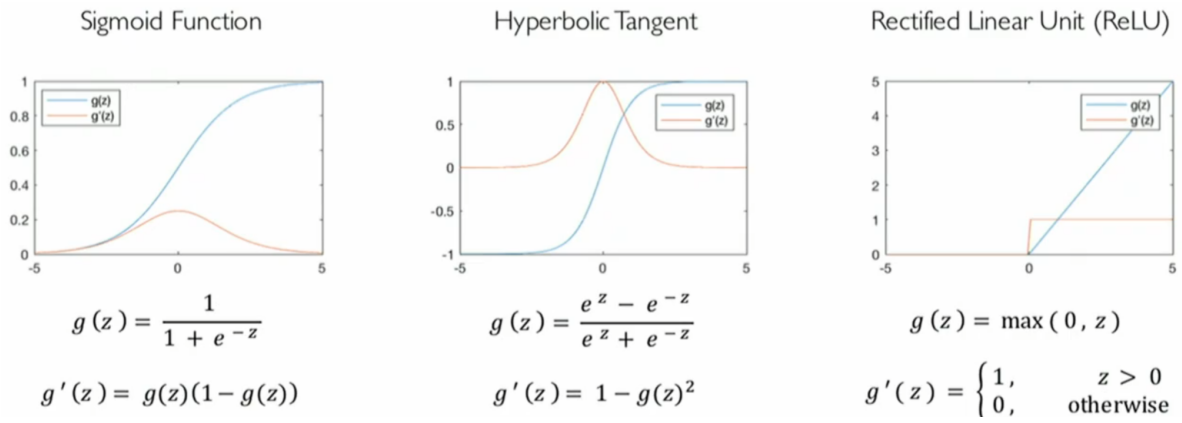
Una **Rete Neurale** di soli percettroni però presenta lo stesso limite dei singoli percettroni: ora impariamo e trovare la feature migliore ma è pur sempre una feature lineare.

Deep Neural Network

Rete Neurale Profonda: La maggior differenza consiste nella funzione di attivazione: se precedentemente con una rete di percettroni la funzione di attivazione era lineare, infatti si trattava di una semplice sommatoria dei valori in input moltiplicati per i loro pesi, questa volta la **funzione di attivazione non è lineare**.

Adesso si effettua la sommatoria come prima, tuttavia questa volta, prima di mandare il valore in output, lo si fa passare attraverso un'altra operazione che deve essere non lineare per permettere alla rete di poter apprendere una qualsiasi funzione di qualsivoglia complessità. Vediamo due esempi.

Una comune funzione di attivazione è la **ReLU**: $max(0, v)$ dove v è il risultato della sommatoria.
Un'altra spesso usata è la **Sigmoide**: $\frac{1}{1+e^{-v}}$.



Ne esistono ovviamente molte altre, ognuna coi suoi aspetti positivi e negativi, e influenzano molto la velocità di apprendimento, la convergenza o meno, e la qualità della separazione appresa.

Loss Function

L'unica parte di una rete che può variare nel corso del tempo è l'insieme dei pesi associati alle connessioni, dato che gli input dipendono dall'esterno o dagli strati precedenti, mentre gli output sono deterministicamente calcolati.
Per permettere ad una rete di apprendere, è necessario definire una misura della qualità delle sue predizioni. Entra in gioco quindi la **Loss Function**. Una volta

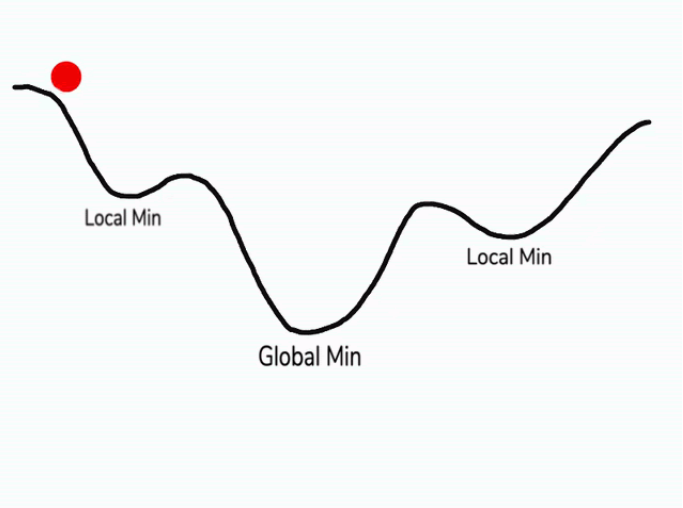
stabilita la loss function più adatta, è quindi necessario scegliere come aggiornare i pesi in modo da riflettere il "punteggio" ottenuto dalla Loss Function con l'obiettivo di migliorarlo.

In genere la **Loss Function** comprende in qualche modo una differenza tra l'output della rete e l'effettivo target e, quindi, bisogna cercare di **minimizzare** il suo valore.

Il gradiente di una funzione è un vettore che ha come componenti le derivate parziali della funzione.

Il metodo del **Gradient Descent (GD)** e è il processo mediante il quale compiamo passi in discesa nella speranza di trovare il **minimo globale** di una funzione. Il minimo di una funzione è ovunque in cui il gradiente è uguale a 0, ricordandoci che nel nostro caso stiamo cercando di trovare il minimo assoluto della funzione di perdita del nostro algoritmo, ovvero vogliamo raggiungere il minimo globale della funzione di perdita per ottenere il minor numero possibile di errori.

Per chiarire le idee, immaginiamo una palla che rotola giù da una serie di colline: questa pallina è il nostro modello nel processo di "addestramento" e nella ricerca dei pesi e dei bias che ci danno il minimo errore. L'errore minimo si ottiene quando la pallina si ferma al minimo globale. Ecco una **rappresentazione** di quello che vorremmo **ottenere**:



Per applicare tale metodo sono quindi necessari:

- α, detto Learning Rate, per definire di quanto spostarsi lungo la direzione opposta al gradiente;
- Calcolare la derivata della Loss Function per ciascuna componente in output della rete;
- Calcolare ogni derivata a ritroso per ciascuno strato della rete.

Sfortunatamente questi calcoli sono molto onerosi, specialmente più è profonda la rete, ma fortunatamente è sempre possibile calcolare ogni derivata, grazie alla **Chain Rule**.

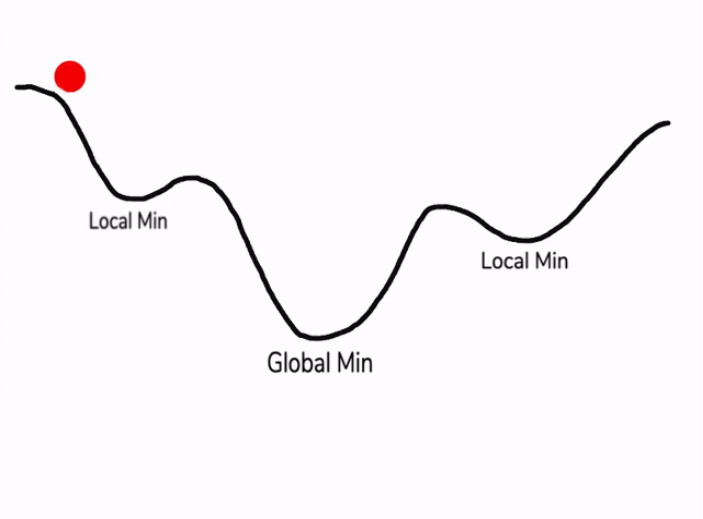
BackPropagation

Per evitare l'esplosione dei calcoli da dover computare, e facendo leva sulla chain rule, è possibile calcolare di volta in volta durante il forward pass le derivate parziali di ciascun nodo, permettendo poi alla fine di calcolare e mandare indietro il gradiente della Loss Function più velocemente, con un costo pari all'elaborazione della rete. Questa tecnica prende il nome di **Backpropagation**.

Loss Function Technique

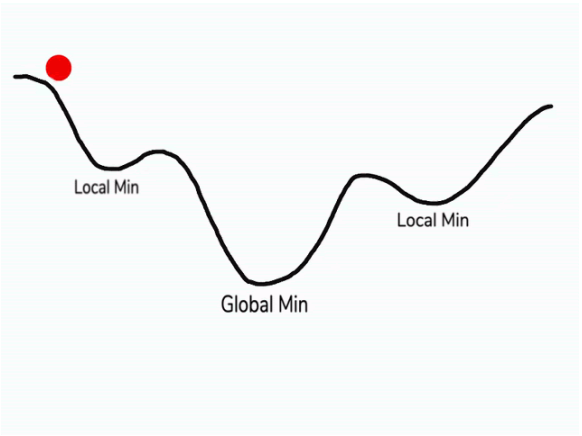
Le problematiche principali legate al GD sono:

- **Progressione oscillante**: quando la funzione ha una discesa ripida in una direzione, mentre diminuisce poco in un'altra, la progressione del GD oscilla facendo fatica a raggiungere il minimo.
- **Stallo su minimi locali**: facilmente si blocca nei punti di minimo locale della funzione, poiché attorno ad essi la direzione di discesa punta sempre verso il minimo locale stesso. Ecco una rappresentazione di tale problema.



Per ridurre il problema della progressione oscillante, si introduce una variazione del GD, detta Stochastic Gradient Descent (**SGD**). A differenza del classico GD, che calcola e aggiorna i pesi del modello utilizzando l'intero set di dati di addestramento ad ogni iterazione, l'SGD aggiorna i pesi utilizzando solo un **sottoinsieme casuale** del dataset, noto come **batch**. Questo porta a una maggiore variabilità nell'approssimazione del gradiente, ma consente all'algoritmo di convergere più rapidamente in quanto richiede meno computazione.

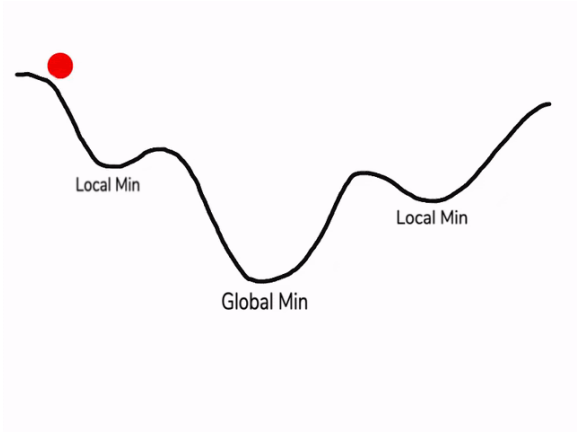
Per ovviare invece alla problematica dell'oscillazione del GD, si introduce il **momentum** in modo da modificare la direzione del gradiente in relazione con la direzione precedente. Per farlo influenziamo la direzione che vogliamo prendere in base alle direzioni nelle iterazioni precedenti, convergendo prima. Ecco ciò che otteniamo.



Effettivamente raggiungiamo il minimo globale, ma anche usando SDG + Momentum abbiamo troppa oscillazione.

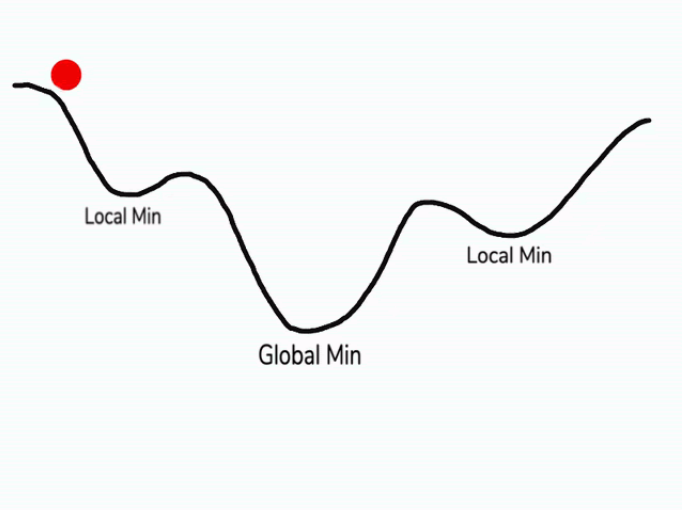
Un altro miglioramento che possiamo effettuare è quello di far variare dinamicamente la distanza lungo la direzione del GD, rendendo il learning rate dinamico vicino al minimo. Questa tecnica è chiamata **AdaGrad**.

RMSProp: variante di AdaGrad dove il gradiente viene calcolato da una media che decade esponenzialmente, anziché dalla somma dei suoi gradienti. In un certo senso, RMSProp sostanzialmente "rallenta" la palla vicino al minimo globale, regolando e adattando di conseguenza il learning rate. Otteniamo ciò:



Siamo arrivati al punto in cui SDG + Momentum è veloce ma non stabile, mentre AdaGrad è lento ma stabile. Ci servono entrambi.

Adam: attualmente il migliore. Invece di adattare i tassi di apprendimento in base al primo momento medio come in RMSProp, Adam utilizza la media dei secondi momenti dei gradienti.



Initializing Neural Networks

Inizializzare una rete è molto importante, dato che il valore iniziale dei pesi e degli iperparametri influenza moltissimo la velocità di apprendimento e la convergenza o meno della rete.

Una prima scelta per inizializzare i **pesi iniziali** potrebbe essere quella di porli tutti a **0**. Questa scelta non è consigliata dato che se tutti i pesi hanno lo stesso valore iniziale, tutte le unità all'interno dello stesso strato produrranno lo stesso output durante la fase di forward propagation. Di conseguenza, durante la fase di backpropagation, tutti i pesi nello stesso strato riceveranno lo stesso segnale di errore e si aggiornano in modo identico. Ciò può portare a **problemi di simmetria** e alla **mancanza di diversità nel modello**, **limitando** la **capacità di apprendimento**.

Un'altra strategia comune è inizializzare i **pesi** con valori **randomici** presi da una distribuzione ampia, come una distribuzione normale con media zero e varianza non piccola. Questa inizializzazione consente ai pesi di avere una maggiore varietà iniziale, che può essere utile quando la rete neurale ha una maggiore complessità e profondità. Tuttavia, **valori troppo grandi** possono portare a **rallentare l'apprendimento** o causare **problemi di stabilità** durante la fase di aggiornamento dei pesi.

HyperParameters

Gli **iperparametri** ricoprono un ruolo ancor più fondamentale e a seconda del modello di rete, della Loss Function e dell'ottimizzazione scelta, possono variare in numero.

Esistono quindi delle linee guida e delle procedure per aiutare a trovare gli iperparametri migliori:

1. Controllare la **loss iniziale** per avere un punto di riferimento.
2. Addestrare il modello su un **sottocampione** del dataset per individuare configurazioni che mostrano overfitting.
3. Trovare un Learning Rate che faccia **diminuire** la loss del punto 1.
4. Fare **Cross Validation** addestrando il modello su diverse combinazioni di iperparametri.
5. Scegliere le **combinazioni migliori** e ripetere il passo 4, questa volta con combinazioni interne ai valori scelti.
6. **Ripetere** i passaggi 4 e 5 iterativamente per affinare la scelta degli iperparametri.

Un modo efficace per scegliere le combinazioni di iperparametri è quello di scegliere dei limiti per i valori di ciascun iperparametro ad occhio. Questi valori saranno raffinati durante le varie iterazioni. Si scelgono poi dei valori campione e si esegue la cross validation, ottenendo dei nuovi valori migliori che saranno i nuovi limiti.

Regularization

La regolarizzazione è una tecnica molto diffusa per **evitare l'overfitting**. Per contrastarlo, ci sono diverse strategie ampiamente utilizzate:

- **Modifica della Loss Function**: viene aggiunto un nuovo termine alla Loss Function. Questo termine extra penalizza i modelli che hanno pesi troppo grandi, limitando così l'eccessiva specializzazione sui dati di addestramento.
- **Early stopping**: fermare l'addestramento del modello quando le prestazioni sul set di validazione smettono di migliorare. Monitorando le metriche di valutazione sul set di validazione durante l'addestramento, è possibile individuare il punto in cui il modello inizia ad adattarsi eccessivamente ai dati di addestramento.
- **Dropout**: una tecnica di regolarizzazione che mira a rendere il modello più robusto e meno dipendente da singoli neuroni specifici. Durante l'addestramento, il dropout disattiva casualmente alcuni neuroni e imposta i loro output a zero. Ciò costringe il modello a imparare caratteristiche utili da diverse combinazioni di neuroni e previene l'eccessiva specializzazione.

Convolution

Nella Computer Vision, ovvero quel campo che si occupa di permettere ai computer di estrapolare informazioni di alto livello da video e immagini, l'introduzione delle reti neurali è stata un passo fondamentale in avanti. Una tecnica fondamentale che viene utilizzata è la **convoluzione**.

La Convoluzione è un'operazione matematica che consiste nel prendere una funzione e applicarla iterativamente a **porzioni** di un'altra funzione.

Il concetto fondamentale è che, data un'immagine, si decide una **finestra** con associata dei **pesi** e la si fa **scorrere** sull'immagine, in modo da ottenere un risultato.

Convolutional Neural Network

Problema nell'applicazione delle Reti Neurali Profonde alla Computer Vision: la perdita delle informazioni spaziali.

Le classiche DNN prendono una sequenza lineare di dati in input, mentre le immagini sono composte da dati che sono relazionati tra loro anche nello **spazio** in cui si trovano. Per ovviare a questo problema sono state create le prime **Reti Neurali Convolutive** (CNN).

Le CNN, o meglio gli strati convolutivi, fanno uso della procedura convolutiva.

1. L'immagine viene descritta in **3 dimensioni**, ovvero larghezza, altezza e profondità. La **profondità** è data dai **canali**, per esempio un'immagine in bianco e nero avrà 1 canale i cui valori per ogni pixel vanno da 0 a 255. Un'immagine a colori avrà 3 canali ciascuno che va da 0 a 255.
2. Scelta una finestra, spesso chiamata anche **kernel** la si scorre per tutta l'immagine. Questa è la parte convolutiva, infatti ad ogni applicazione della finestra sull'input **si moltiplicano i valori dell'input** inquadrati dalla finestra con i **pesi della finestra stessa**, si **sommano** tra di loro e si **salva** il risultato in un unico nodo nello strato successivo.
3. Questo significa che si avranno tot **filtri**, ovvero set di pesi per finestre, e per ciascuno si farà convoluzione sull'input come descritto sopra. Questo permette anche di astrarre l'input dall'immagine, ovvero è possibile applicare strati convoluzionali all'output di altri strati convoluzionali, dato che l'output sarà anch'esso descritto in 3 dimensioni (larghezza, altezza, numero filtri).

Osservazione: gli strati convoluzionali più vicini all'input (all'immagine) imparano a riconoscere feature di basso livello, mentre gli strati convoluzionali più vicini all'output (il risultato della rete) imparano a riconoscere feature di alto livello.

Questo fenomeno è dato dal fatto che i pesi del kernel usato sono in percentuali e fatti in modo che la loro somma dia sempre 100%.

Spesso tra uno strato convoluzionale e l'altro viene aggiunto uno strato di **Max Pooling**, ovvero uno strato che si occupa di ridurre la dimensionalità dei dati facendo una semplice operazione di scelta del massimo in una certa finestra.

Poiché per addestrare una CNN con risultati soddisfacenti è richiesta un'enorme quantità di dati, una pratica molto diffusa è quella del **trasferimento dell'apprendimento**. Questa pratica consiste nell'**estrarre** le attivazioni (embeddings) da uno o più strati di una CNN già addestrata e usarli come input della rete che si vuole addestrare.