Appunti di Intelligenza Artificiale

Giacomo Simonetto

Primo semestre 2025-26

Sommario

Appunti del corso di Intelligenza Artificiale della facoltà di Ingegneria Informatica dell'Università di Padova.

Indice

1 Introduction to AI

1.1 Introduzione

Definizione (una delle tante possibli)

"L'intelligenza artificiale è lo studio di come far compiere ai computer compiti che attualmente sono svolti in maniera migliore dagli esseri umani"

— Elaine Rich & Kevin Knight, 1965

Branche dell'intelligenza artificiale

L'intelligenza artificiale è multidisciplinare. Da domande e questi di varie discipline sono stati originati vari filoni di ricerca del'IA, tra cui:

- **filosofia**: origine della conoscenza, logica, metodi di ragionamento, struttura fisica della mente
- **matematica**: rappresentazione formale della conoscenza, teoria della probabilità, computabilità
- **neuroscienze**: funzionamento del cervello umano, neural networks, brain machine interfaces
- **psicologia**: percezione, controllo motorio, interazioni umano-macchina
- **linguistica**: rappresentazione della conoscenza, natural language processing
- **control theory**: cybernetica, robotica
- **economia**: teoria dell'utilità, teoria delle decisioni (game theory, multiagent systems)
- **informatica**: algoritmi, strutture dati, hardware dedicato (GPU, TPU) e Internet of Things (IoT)

1.2 Storia

Storia dell'IA

1943	**McCulloch** e **Pitts** propongono il primo modello di circuito booleano basato sulla struttura del cervello umano
1950	**Alan Turing** pubblica il paper "Computing Machinery and Intelligence", in cui propone il Turing Test come criterio per stabilire se una macchina possa essere considerata intelligente.
1956	**John McCarthy** conia il termine "intelligenza artificiale"
1966-73	**primo inverno dell'IA** causato dall'aumento della complessità computazionale degli algoritmi intelligenti da sviluppare e dalla riduzione dei finanziamenti alla ricerca
1970-80	sviluppo di sistemi esperti e sviluppo delle **reti neurali**
1980-87	**secondo inverno dell'IA** causato dalla delusione per le limitazioni delle reti neurali dovute allo scarso sviluppo tecnologico di allora e dalla riduzione dei finanziamenti alla ricerca
1997	**Deep Blue** di IBM sconfigge il campione del mondo di scacchi Garry Kasparov
2005	**Stanley**, un'auto autonoma sviluppata dalla Stanford University, vince la DARPA Grand Challenge
2011	Apple lancia **Siri**, un assistente virtuale basato su intelligenza artificiale e riconoscimento vocale
2012	**AlexNet**, una rete neurale convoluzionale, vince la competizione ImageNet, segnando un grande passo avanti nel riconoscimento delle immagini e riportando popolarità alle reti neurali
2017	**AlphaGo** di DeepMind sconfigge il campione mondiale di Go, Lee Sedol
2019	Boston Dynamics presenta **Spot**, un robot quadrupede capace di muoversi in ambienti complessi
2022	OpenAI rilascia **ChatGPT**, un modello di linguaggio basato su GPT-3.5, che dimostra capacità avanzate di generazione di testo e comprensione del linguaggio naturale

Stato attuale basato sll'AI index di Stanford

1. **performance**: i modelli di intelligenza artificiale continuano a migliorare in vari compiti, avvicinandosi sempre di più alle capacità umane
2. **presenza quotidiana**: la presenza di tecnologie basate su intelligenza artificiale è in crescita in vari settori della vita quotidiana, tra cui sanità, finanza, trasporti e intrattenimento
3. **finanziamenti**: gli investimenti in ricerca e sviluppo nell'ambito dell'intelligenza artificiale sono in crescita, sia da parte di governi che di aziende private
4. **divario USA-Cina**: il divario tra Stati Uniti e Cina in termini di ricerca e sviluppo nell'ambito dell'intelligenza artificiale si sta riducendo, con la Cina che sta facendo progressi significativi
5. **etica**: c'è una crescente attenzione all'etica, alla regolamentazione e alle implicazioni sociali
6. **impiego nazionale**: in generale l'impiego di intelligenza artificiale nei vari paesi è in crescita, anche se è presente ancora un certo divario tra paesi sviluppati e in via di sviluppo
7. **costi**: i costi associati allo sviluppo e all'implementazione di tecnologie basate su intelligenza artificiale stanno diminuendo, rendendo queste tecnologie sempre più accessibili
8. **regolamentazione**: i governi stanno iniziando a sviluppare regolamentazioni specifiche sia per l'uso responsabile dell'intelligenza artificiale che per gli investimenti nel settore
9. **educazione**: c'è un crescente diffusione di programmi educativi e corsi di formazione sull'informatica e sull'intelligenza artificiale, anche se c'è ancora divario tra paesi e regioni
10. **sviluppo**: la ricerca e lo sviluppo nell'ambito dell'intelligenza artificiale stanno continuando a progredire rapidamente, con nuove tecnologie e approcci che emergono costantemente
11. **impatto**: l'impatto dell'intelligenza artificiale sulle ricerche scientifiche di alto livello è molto significativo, dati anche i premi Nobel per la fisica e la chimica del 2024 assegnati a ricerche che hanno fatto uso di intelligenza artificiale
12. **ragionamento complesso**: i modelli di intelligenza artificiale continuano ad avere difficoltà con compiti che richiedono ragionamento complesso e precisione

1.3 Tipi di intelligenza

I 4 tipi di intelligenza

Esistono 4 modi diversi per definire cosa si intende "intelligenza artificiale". In base alla definizione scelta si potrà sviluppare un processo per decidere se un sistema è intelligente o meno e di conseguenza si avranno obiettivi e metodi di sviluppo diversi. Le 4 definizioni sono:

think humanly - pensare come un umano	think rationally - pensare razionalmente
acting humanly - agire come un umano	acting rationally - agire razionalmente

Think humanly - cognitive science

Si basa sullo studio cognitive science, ovvero la branca della psicologia che studia i processi mentali come l'apprendimento, la memoria, il ragionamento e la percezione. L'obiettivo è quello di creare modelli computazionali che simulino i processi cognitivi della mente umana ottenuti dalla predizione dei comportamenti (introspezione ed esperimenti psicologici, top-down approach) o dalla identificazione dei modelli umani (brain-imaging, bottom-up approach).

Acting humanly - Turing test

Dalla definizione di Alan Turing: "Una macchina è intelligente se fa cose intelligenti". Questo principio è alla base del Turing Test o Imitation Game, che serve appunto a valutare se una macchina agisce in modo intelligente come un essere umano. Il test coinvolge vari settori come natural language processing, knowledge representation, automated reasoning e machine learning a cui si aggiungono computer vision, speech recognition e robotics nei test più estesi.

Think rationally - logic

Questa definizione si basa sul ragionamento logico ed è in contrapposizione con la cognitive science in quanto non sempre gli esseri umani pensano in modo razionale. L'obiettivo è quello di creare sistemi che seguano le regole della logica formale per ragionare correttamente e prendere decisioni basate su informazioni disponibili (sfruttando la logica della filosofia greca e i metodi probabilistici per il calcolo dell'incertezza).

Acting rationally - rational agent

Questa definizione si basa sulla teoria dei rational agent (o sistemi intelligenti), ovvero agenti che agiscono in modo da massimizzare la loro performance measure, basandosi su ciò che percepiscono dall'ambiente in cui operano. Questa è l'interpretazione più diffusa e accettata come standard per l'intelligenza artificiale attuale. Un rational agent deve essere in grado di operare autonomamente, percepire l'ambiente, adattarsi ai cambiamenti, vivere per un tempo prolungato e raggiungere i propri obiettivi, attraverso le stesse tecniche definite nel Turing Test e nel ragionamento logico.

1.4 Rischi e benefici dell'IA

Benefici

- diminuzione del lavoro ripetitivo e noioso
- aumento della produttività di servizi e benefici
- accelerazione di ricerche scientifiche

Rischi

- impiego in applicazioni militari
- violazione della privacy
- bias nei dati e nei modelli
- perdita di posti di lavoro
- cybersecurity
- value alignment problem: serve allineare i valori dell'IA per evitare che i goal dell'IA sovrastino i valori umani, creando superintelligenze pericolose e fuori controllo

2 Intelligent agents (o rational agents)

2.1 Introduzione agli agenti intelligenti

Definizione di agente in generale

Un agente (in generale) è un'entità che percepisce il suo ambiente attraverso sensori e agisce su di esso attraverso attuatori. Può essere schematizzato come una funzione matematica che mappa una sequenza di percezioni a una sequenza di azioni.

$$f : P \to A$$

Definizione di agente intelligente

Un agente intelligente (o rational agent) è un agente che, per ogni possibile percezione, seleziona l'azione che si aspetta che massimizzi la sua performance measure, dato ciò che ha percepito fino a quel momento.

$$a = f(p) = \arg\max_{\tilde{a}} \text{ performance}(\, \tilde{a} \mid p \,) \qquad \text{con } \; a \in A, \;\; p \in P$$

Framework PEAS - (Performance, Environment, Actuators, Sensors)

Per definire un agente intelligente si utilizza il framework PEAS, ovvero un insieme di aspetti chiave che caratterizzano uno specifico agente intelligente. Tali aspetti definiscono il task environment dell'agente, ovvero il problema che l'agente deve risolvere. I 4 aspetti chiave (dati dalle iniziali P.E.A.S.) sono:

- **Performance**: una funzione che valuta il comportamento dell'agente in base agli obiettivi prefissati
- **Environment**: l'ambiente in cui l'agente opera, che può essere fisico o virtuale
- **Actuators**: i mezzi attraverso cui l'agente può agire sull'ambiente
- **Sensors**: i mezzi attraverso cui l'agente può percepire l'ambiente

2.2 Proprietà di un task environment

Fully observable vs partially observable

Un ambiente si dice fully observable se l'agente può accedere a tutte le informazioni che compongono lo stato dell'ambiente in ogni momento. Viceversa si dice partially observable se l'agente ha accesso solo a una parte delle informazioni che compongono lo stato. Se l'agente non ha sensori, l'ambiente è completamente non osservabile, ma comunque può perseguire un obiettivo.

Deterministic vs stochastic

Un ambiente si dice deterministic se lo stato successivo dell'ambiente è completamente determinato dallo stato attuale e dall'azione dell'agente. Viceversa si dice stochastic o non-determinisc se lo stato successivo dipende anche da fattori casuali o imprevedibili. Per vivere in un ambiente non-deterministico, l'agente deve essere in grado di gestire l'incertezza e implementare regole di teoria della probabilità.

Episodic vs sequential

Un ambiente si dice episodic se l'esperienza dell'agente è suddivisa in episodi temporalmente distinti e indipendenti, formati da una singola percezione e una singola azione. Viceversa si dice sequential se l'esperienza dell'agente è continua e le azioni dell'agente influenzano le percezioni future. In un ambiente sequenziale, l'agente deve considerare le conseguenze a lungo termine delle sue azioni.

Static vs dynamic

Un ambiente si dice static se rimane invariato mentre l'agente sta prendendo una decisione. Viceversa si dice dynamic se l'ambiente può cambiare mentre l'agente sta prendendo una decisione. In un ambiente dinamico, l'agente deve essere in grado di adattarsi rapidamente ai cambiamenti.

Discrete vs continuous

Un ambiente si dice discrete se stati, percezioni e azioni cambiano ad intervalli di tempo discreti ($t \in \mathbb{Z}$). Spesso un ambiente discreto è anche finito, ovvero ha un numero finito di stati, percezioni e azioni. Viceversa un ambiente si dice continuous se stati, percezioni e azioni cambiano in modo continuo nel tempo ($t \in \mathbb{R}$). Alcune percezioni possono essere discrete (es. immagini digitali) ma se riferite ad un ambiente continuo, l'ambiente viene comunque considerato continuo.

Single agent vs multiagent

Un ambiente si dice single agent se c'è un solo agente che opera nell'ambiente. Viceversa si dice multiagent se ci sono più agenti che operano nell'ambiente. In un ambiente multiagent, si distinguono agenti cooperativi (che lavorano insieme per raggiungere un obiettivo comune) e agenti competitivi (che competono tra loro per risorse limitate o obiettivi contrapposti).

Known vs unknown

Un ambiente si dice known se l'agente ha una conoscenza completa delle dinamiche dell'ambiente, ovvero sa come le sue azioni influenzeranno lo stato futuro dell'ambiente. Viceversa si dice unknown se l'agente ha una conoscenza limitata delle dinamiche dell'ambiente e deve imparare attraverso l'esperienza come le sue azioni influenzeranno lo stato futuro. Un ambiente può essere known anche se è partially observable o stochastic, ovvero l'agente conosce le regole che governano l'ambiente ma non ha accesso a tutte le informazioni per definire lo stato attuale o lo stato futuro.

2.3 Tipi di agenti

Simple reflex agents

Un simple reflex agent agisce basandosi su regole condizionali (if-then-else) per selezionare l'azione da eseguire in base alla percezione attuale. La scelta delle azioni è basata esclusivamente sulla percezione attuale, senza considerare la storia delle percezioni passate o le conseguenze future delle azioni. Questi agenti sono adatti per ambienti fully observable.

Model-based reflex agents

Un model-based reflex agent mantiene un modello interno dello stato dell'ambiente, che viene aggiornato in base alle percezioni attuali. La scelta delle azioni è basata sia sulla percezione attuale che sul modello interno dello stato dell'ambiente. Questi agenti sono adatti per ambienti partially observable, in quanto possono utilizzare il modello interno per inferire informazioni sullo stato attuale dell'ambiente che non sono direttamente osservabili. Inoltre sono più complessi da implementare perché richiedono un transition model e un sensor model per aggiornare il modello internoche basandosi rispettivamente sulle azioni compiute (prediction) e sulle percezioni dell'agente (update). Vengono spesso usati i Kalman-filters per implementare, per l'appunto, le due fasi di prediction e update.

Goal-based agents

Un goal-based agent utilizza obiettivi stabiliti a priori per guidare la scelta delle azioni. La scelta delle azioni è basata sull'analisi delle conseguenze future delle azioni in relazione agli obiettivi dell'agente. Questi agenti sono adatti per ambienti sequenziali, in quanto devono considerare le conseguenze a lungo termine delle loro azioni per raggiungere gli obiettivi prefissati. Inoltre sono più complessi da implementare perché richiedono una funzione di valutazione delle azioni in base alla loro capacità di avvicinare l'agente agli obiettivi prefissati. Esempi di algoritmi utilizzati per implementare goal-based agents sono gli algoritmi di ricerca come DFS, BFS, Dijkstra e A*.

Utility-based agents

Un utility-based agent utilizza una funzione di utilità per valutare lo stato dell'ambiente e guidare la scelta delle azioni. La funzione di utilità assegna un valore numerico a ciascuno stato dell'ambiente, rappresentando il grado di soddisfazione dell'agente in quello stato. La scelta delle azioni è basata sia sul raggiungimento degli obiettivi che sulla massimizzazione della funzione di utilità. Questi agenti sono

adatti per ambienti complessi e dinamici dove ci sono conflitti tra obiettivi multipli, in quanto devono bilanciare il raggiungimento degli obiettivi con la massimizzazione della soddisfazione dell'agente. Per modellare matematicamente un utility-based agent si utilizzano ad esempio i Markov Decision Processes (MDP) con le Bellman equations per calcolare la politica ottimale dell'agente.

Differenza tra goal-based e utility-based agents

La differenza principale tra goal-based agents e utility-based agents è che i primi si concentrano esclusivamente sul raggiungimento degli obiettivi prefissati, mentre i secondi cercano di massimizzare la soddisfazione complessiva dell'agente considerando sia gli obiettivi che la funzione di utilità.

In particolare se l'utilità è costante nel tempo allora un utility-based agent degenera in un goal-based agent (il diagramma degli stati è grafo con archi pesati con pesi costanti) e la policy è costante nel tempo ed equivale ad una specifica sequenza di azioni chiare ed assolute dette cammino minimo. Se l'utilità, invece, varia nel tempo ed ha componenti stocastiche, allora i search algorithms non sono più sufficienti per implementare un utility-based agent e servono gli MDP con le Bellman equations per calcolare la optimal policy.

Ad esempio potrebbe verificarsi che il cammino minimo passa molto vicino ad un burrone e si ha una probabilità non nulla di cadere ed avere un'utilità estremamente svantaggiosa. Un goal-based agent seguirebbe comunque quel cammino minimo, mentre un utility-based agent potrebbe scegliere un cammino più lungo ma più sicuro per massimizzare l'utilità attesa.

Learning agents

Un learning agent è un agente che può migliorare le sue prestazioni attraverso l'apprendimento dall'esperienza. Attraverso l'apprendimento nel tempo permette di risultare particolarmente efficace in ambienti con cambiamenti dinamici e inizialmente sconosciuti. Un learning agent è composto da quattro componenti principali:

- **Performance element**: decide le azioni dell'agente (sfruttando qualsiasi dei precedenti modelli di agente visti in precedenza) e valuta le prestazioni dell'agente in base alla performance measure
- **Learning element**: consente all'agente di apprendere dall'esperienza e migliorare le sue prestazioni correggendo i parametri del performance element
- **Critic**: fornisce feedback al learning element sulla qualità delle azioni dell'agente
- **Problem generator**: suggerisce nuove esperienze o situazioni per l'agente da esplorare

3 Search algorithms

3.1 Formulazione ed esempi di search problems

Agenti utilizzati per risolvere i search problems

Gli agenti che risolvono search problems sono tipicamente goal-based agents in grado di pianificare una sequenza di azioni per raggiungere un obiettivo specifico.

Problem formulation

La problem formulation è il processo di definizione di un problema di ricerca, ovvero rappresentarlo in forma atomica (essenziale e completa) in termini di:

- **elenco degli stati**: l'insieme di tutte le possibili configurazioni dell'ambiente
- **stato iniziale**: la configurazione iniziale dell'ambiente
- **condizione o stati di goal**: l'obiettivo che l'agente deve raggiungere, può essere espresso come condizione da far valere, oppure un insieme di stati da raggiungere
- **azioni**: le possibili azioni che l'agente può compiere da ogni stato
- **transition model**: la funzione che descrive come le azioni influenzano lo stato dell'ambiente
- **action cost function**: la funzione che assegna un costo alle azioni o agli stati

La soluzione di un problema è la sequenza di azioni che porta dallo stato iniziale a uno stato di goal. Una soluzione si dice ottima se minimizza il costo totale delle azioni.

Esempi di search problems

- **Romania problem**: trovare il percorso più breve che collega la città di Arad a Bucarest in Romania in un ambiente rappresentato come un grafo pesato dove i nodi sono le città e gli archi sono le strade con i relativi costi di viaggio
- **travelling salesperson problem**: trovare il percorso più breve che deve compiere il commesso viaggiatore per visitare un insieme di città, ciascuna esattamente una volta e ritornare alla città di partenza, in un ambiente rappresentato come un grafo completo pesato dove i nodi sono le città e gli archi sono le distanze tra le città; a tale problema possono essere associati varianti come il vehicle routing problem (VRP) o il controllo di una fresa CNC
- **8-queens problem**: posizionare 8 regine su una scacchiera 8x8 in modo che nessuna regina possa attaccare un'altra, in un ambiente rappresentato come uno spazio di stati dove ogni stato è una configurazione della scacchiera e le azioni sono i posizionamenti delle regine
- **pancake sorting problem**: ordinare una pila di pancake di diverse dimensioni utilizzando una spatola che può ribaltare una porzione superiore della pila, in un ambiente rappresentato come uno spazio di stati dove ogni stato è una configurazione della pila e le azioni sono i ribaltamenti
- **8-puzzle**: un puzzle formato da una griglia 3x3 con 8 tessere numerate e una casella vuota. L'obiettivo è riordinare le tessere spostandole nella casella vuota, in un ambiente rappresentato come uno spazio di stati dove ogni stato è una configurazione della griglia e le azioni sono gli spostamenti delle tessere

3.2 Introduzione ai tree search algorithms

Definizione dello spazio degli stati del problema

Dato un insieme degli stati del problema, un transition model e le azioni possibili per ogni stato (definiti nel processo di problem formulation) è possibile costruire un grafo orientato in cui i vari stati sono rappresentati come nodi e le azioni come archi diretti che collegano i nodi. Questo grafo è detto spazio degli stati del problema. L'esplorazione dello spazio degli stati attraverso algoritmi di ricerca sui grafi costruisce un albero di ricerca (search tree) con una radice che rappresenta lo stato iniziale, da cui si espandono i vari nodi figli che rappresentano gli stati raggiungibili con ogni azione.

Distinzione tra stati e nodi

Lo stato e il nodo sono due concetti diversi e separati, per cui è importante non confonderli:

- **stato**: rappresenta una configurazione specifica dell'ambiente in un dato momento
- **nodo**: rappresenta una specifica istanza di uno stato all'interno dell'albero di ricerca, includendo informazioni aggiuntive come il nodo genitore, l'azione che ha portato a quello stato, eventuali nodi figli, il costo del percorso (path cost) e la profondità del nodo nell'albero di ricerca

Due nodi distinti possono contenere lo stesso stato se sono stati raggiunti attraverso percorsi diversi nell'albero di ricerca.

General Tree-search algorithm

Un esempio generale di tree-search algorithm consiste nel partire dalla radice (stato iniziale) e iterativamente (o ricorsivamente) espandere progressivamente ciascuno dei nodi della frontiera secondo una certa strategia, generando nuovi nodi che verranno aggiunti alla frontiera. Il processo continua fino a trovare uno stato di goal o fino ad esaurire i nodi da espandere (fallimento).

È fondamentale notare che il goal viene raggiunto soltanto quando si espande un nodo che contiene uno stato di goal, non quando si aggiunge un nodo di goal alla frontiera (vedi commento sul codice).

0: **Algoritmo:** TREE-SEARCH(P, S)
Input: P a problem formulation and S a strategy for choosing expanding node
Output: solution or failure

```
1:  initialize the search tree using the initial state of P as the root node
2:  loop do
3:      if there are no candidates for expansion in the frontier then
4:          return failure
5:      select a node from the frontier according to S
6:      if the node contains a goal state then
7:          return the corresponding solution          ▷ goal reached only when expanding the node
8:      else
9:          expand the node and add all the generated nodes to the frontier
10: end loop
```

Frontiera e strategia di un algoritmo di ricerca

La **frontiera** è l'insieme dei nodi raggiunti dall'algoritmo ricerca, ma che non sono ancora stati espansi. La frontiera separa i nodi già visitati (espansi) dai nodi non ancora visitati (non espansi). La gestione della frontiera avviene tramite un insieme di regole dette **strategia** e determina l'ordine di esplorazione dei nodi. La strategia è specifica per ogni algoritmo di ricerca e ne caratterizza il comportamento. È cruciale per determinare l'efficacia e l'efficienza dell'algoritmo di ricerca.

Valutazione di un algoritmo

Gli algoritmi di ricerca possono essere valutati in base a vari criteri:

- **completeness**: un algoritmo è completo se trova sempre una soluzione se ne esiste almeno una
- **optimality**: un algoritmo è ottimale se garantisce di trovare la soluzione con il costo minimo
- **time complexity**: proporzionale al numero di nodi generati/espansi durante la ricerca
- **space complexity**: proporzionale al numero di nodi in memoria durante la ricerca

Tali criteri si esprimono in funzione di vari parametri del problema:

- b: branching factor, il numero massimo di figli per ogni nodo
- d: depth of the least-cost solution, la profondità della soluzione ottimale
- m: maximum depth of the search tree, la profondità massima dell'albero di ricerca

3.3 Uninformed search strategies

Un algoritmo di ricerca si dice uninformed (o blind search) se non ha alcuna informazione sulla distanza di un certo stato dallo stato di goal.

Breadth-first search - BFS

Espande il nodo più superficiale (livello più vicino alla radice) prima di espandere i nodi più profondi.

- **strategy / frontier**: FIFO queue
- **complete**: sì, se il branching factor è finito
- **optimal**: sì, se il costo delle azioni è costante
- **time complexity**: $O(b^d)$
- **space complexity**: $O(b^d)$
- **vantaggi**: è completo e ottimale (con costi delle azioni costanti)
- **svantaggi**: complessità di spazio elevata dovuta al fatto che la cardinalità della frontiera è pari al numero di nodi di ogni livello dell'albero di ricerca che al caso peggiore è $O(b^d)$

Siccome il primo nodo di goal generato (e inserito nella frontiera) è anche il primo nodo di goal espanso, allora per questioni di ottimizzazione della ricerca, si può interrompere l'espansione dei nodi non appena si genera il primo nodo di goal. Questa regola vale solo per il BFS e non per gli altri algoritmi di ricerca in quanto non sempre la frontiera è una struttura FIFO.

Uniform-cost search - Dijkstra

Espande il nodo con il costo del percorso dalla radice al nodo più basso dalla radice.

- **strategy / frontier**: priority queue ordinata per path cost (lowest-cost first)
- **complete**: sì, se il branching factor è finito
- **optimal**: sì, anche con costi delle azioni non costanti (ma positivi)
- **time complexity**: molto peggiore di $O(b^d)$ per costi molto piccoli
- **space complexity**: molto peggiore di $O(b^d)$ per costi molto piccoli
- **vantaggi**: è completo e ottimale (con costi delle azioni positivi)
- **svantaggi**: complessità di tempo e spazio possono essere molto elevate se i costi delle azioni sono molto piccoli in quanto prima vengono esplorati i nodi con costi più bassi senza allontanarsi troppo dalla radice e soltanto dopo vengono esplorati i nodi con costi più alti che magari portano più velocemente alla soluzione

Depth-first search - DFS

Espande il nodo più profondo (livello più lontano dalla radice) prima di espandere i nodi più superficiali.

- **strategy / frontier**: LIFO stack
- **complete**: no, può rimanere bloccato in cicli infiniti (se non opportunamente gestiti)
- **optimal**: no
- **time complexity**: $O(b^m)$
- **space complexity**: $O(bm)$
- **vantaggi**: complessità di spazio molto bassa
- **svantaggi**: non è completo, non è ottimo e la complessità temporale può essere molto peggiore di $O(b^d)$ se la profondità massima dell'albero di ricerca m è molto maggiore della profondità della soluzione ottimale d

Depth-limited DFS

È una variante del DFS che impone un limite massimo alla profondità dell'albero riducendo m e di conseguenza limitando la complessità temporale dell'algoritmo. Ha le stesse caratteristiche del DFS.

Iterative deepening search

Sfrutta l'approccio del Depht-limited DFS eseguendo ripetutamente il DFS con un limite di profondità crescente ad ogni iterazione fino a trovare la soluzione. Questo permette di combinare i vantaggi del BFS (completezza e ottimalità) con quelli del DFS (bassa complessità spaziale).

- **strategy / frontier**: LIFO stack con limite di profondità crescente
- **complete**: sì, se il branching factor è finito
- **optimal**: sì, se il costo delle azioni è costante
- **time complexity**: $O(b^d)$
- **space complexity**: $O(bd)$
- **vantaggi**: è completo, ottimale e ha una complessità spaziale molto bassa rispetto a BFS e UCS
- **svantaggi**: è leggermente più lento di BFS (anche se asintoticamente sono uguali) a causa della ripetizione della espansione dei nodi nei livelli inferiori

Bidirectional search

Espande contemporaneamente due alberi di ricerca, uno a partire dallo stato iniziale e l'altro a partire dallo stato di goal, fino a quando i due alberi si incontrano in un nodo comune. Questo permette di ridurre significativamente la complessità temporale e spaziale dell'algoritmo, ma richiede di conoscere a priori lo stato di goal e di essere in grado di generare i nodi figli in entrambe le direzioni (forward e backward). In generale si usano BFS o UCS per espandere i due alberi di ricerca.

- **strategy / frontier**: FIFO queue o priority queue come BFS o UCS
- **complete**: sì, se il branching factor è finito
- **optimal**: sì, se il costo delle azioni è costante
- **time complexity**: $O(b^{d/2})$
- **space complexity**: $O(b^{d/2})$
- **vantaggi**: è completo, ottimale e ha una complessità spaziale molto bassa rispetto a BFS e UCS
- **svantaggi**: serve conoscere a priori il goal e generare i nodi figli in entrambe le direzioni

3.4 Informed search strategies

Un algoritmo si dice informed (o heuristic search) se ha a disposizione una funzione euristica che stima la distanza di un certo stato dallo stato di goal.

Heuristic function

Una funzione euristica $h(n)$ è una funzione che stima il costo (o distanza) tra il nodo n e lo stato di goal. Viene utilizzata dagli algoritmi di ricerca informata per guidare la selezione dei nodi da espandere, privilegiando i nodi che sembrano più promettenti in base alla stima fornita dalla funzione euristica. I nodi all'interno della frontiera vengono ordinati in base alla desiderabilità calcolata sfruttando anche la funzione euristica. Le funzioni euristiche possono essere classificate in base a due proprietà principali:

- **ammissibile**: una funzione euristica si dice ammissibile se non sovrastima mai il costo effettivo per raggiungere lo stato di goal da un nodo n. Formalmente, definita $h^*(n)$ come il costo reale del percorso più economico da n allo stato di goal, allora un'euristica è ammissibile se soddisfa le seguenti condizioni:

$$h(n) \geq 0 \qquad \wedge \qquad h(G) = 0 \text{ se } G \text{ è goal} \qquad \wedge \qquad h(n) \leq h^*(n) \ \ \forall n$$

- **consistente (o monotona)**: un'euristica si dice consistente (o monotona) se per ogni nodo n e ogni suo successore n' generato dall'azione a con costo $c(n, a, n')$ si ha che:

$$h(n) \leq c(n, a, n') + h(n')$$

Un'euristica consistente è sempre ammissibile, ma non viceversa.

Misura della qualità di un'euristica

Per misurare la qualità di un'euristica si utilizza l'effective branching factor b^*, che rappresenta il numero medio di figli per nodo nell'albero di ricerca generato dall'algoritmo di ricerca informata che utilizza l'euristica. Un'ottima euristica ha un effective branching factor $b^* = 1$, ovvero l'algoritmo di ricerca espande solo i nodi lungo il cammino ottimale verso il goal e la complessità temporale e spaziale diventano lineari in $O(d)$.

Greedy best-firts search

Espande il nodo che sembra più vicino allo stato di goal in base alla funzione euristica $h(n)$.

- **strategy / frontier**: priority queue ordinata per $h(n)$ (lowest-h(n) first)
- **complete**: sì, se il branching factor è finito e $h(n) \geq 0$
- **optimal**: no
- **time complexity**: $O(b^m)$, ma una buona euristica può ridurlo parecchio
- **space complexity**: $O(b^m)$, come il BFS, tiene tutti i nodi della frontiera in memoria
- **vantaggi**: esplorando solo i nodi più promettenti è in generale più veloce di algoritmi uninformed
- **svantaggi**: non è ottimale e può rimanere bloccato in cicli infiniti o esplorare percorsi subottimali

A* search

Consiste nell'espandere il nodo con il costo stimato totale più basso $f(n) = g(n) + h(n)$, dove $g(n)$ è il costo del percorso dalla radice al nodo n e $h(n)$ è la stima del costo dal nodo n allo stato di goal.

- **strategy / frontier**: priority queue ordinata per $f(n)$ (lowest-f(n) first)
- **complete**: sì, se il branching factor è finito e il costo delle azioni è positivo
- **optimal**: sì, se la funzione euristica è ammissibile
- **time complexity**: $O(b^d)$, dipende dalla qualità dell'euristica
- **space complexity**: $O(b^d)$, come il BFS, tiene tutti i nodi della frontiera in memoria
- **vantaggi**: è completo e ottimale (con euristica ammissibile) e in generale più efficiente di altri algoritmi di ricerca uninformed e informed
- **svantaggi**: complessità di spazio elevata dovuta al fatto che la cardinalità della frontiera è pari al numero di nodi di ogni livello dell'albero di ricerca che al caso peggiore è $O(b^d)$

Si definisce il **search contour** i-esimo di A* come l'insieme dei nodi n contenuti nella frontiera all'istante di tempo i-esimo. Tali nodi soddisfano la proprietà $f_i \leq f_{i+1}$, ovvero tutti i nodi del contour i-esimo hanno costo stimato totale inferiore ai nodi del contour successivo. Si può dimostrare che A* espande tutti i nodi del contour i-esimo prima di espandere qualsiasi nodo del contour $i+1$-esimo. A differenza del BFS e UCS, l'espansione dei contours tende verso il goal.

Per dimostrare l'**ottimalità** di A* si ipotizza di aver espanso un nodo subgoal G_s e si ipotizza che esista un altro nodo n che appartiene alla soluzione ottimale ma non è stato ancora espanso. Si ottengono le seguenti catene di disuguaglianze:

$$\begin{aligned}
f(G_s) = g(G_s) + h(G_s) = g(G_s) && \text{siccome } h(G_s) = 0 \text{ per def. di goal} \\
g(G_s) > g(G_{opt}) && \text{per ipotesi } G_s \text{ non è ottimale} \\
g(G_{opt}) = g(n) + h^*(n) && \text{siccome n è nel cammino ottimo} \\
g(n) + h^*(n) \geq g(n) + h(n) && \text{siccome } h \text{ è ammissibile} \\
g(n) + h(n) = f(n) && \text{per definizione di } f(n)
\end{aligned}$$

$$f(G_s) = g(G_s) > g(G_{opt}) = g(n) + h^*(n) \geq g(n) + h(n) = f(n) \quad \Rightarrow \quad f(G_s) > f(n)$$

Siccome A* espande sempre il nodo con il costo stimato totale più basso, allora n deve essere stato espanso prima di G_s, il che contraddice l'ipotesi iniziale che G_s fosse stato espanso prima di n e non si può quindi espandere un nodo goal subottimale prima di aver espanso tutti i nodi della soluzione ottimale.

Weighted A* search

L'algoritmo di ricerca Weighted A* è una variante dell'algoritmo A* che introduce un fattore di ponderazione W nella funzione di valutazione $f(n)$, permettendo di bilanciare tra l'ottimalità della soluzione e la velocità di ricerca. Non è più necessario che la funzione euristica sia ammissibile. La funzione di valutazione diventa:

$$f(n) = g(n) + W \cdot h(n) \qquad 1 \leq W \leq \infty$$

Si osserva che per $W = 2$ si riduce lo spazio di ricerca di un fattore 7, aumentando il costo della soluzione di circa il 5%. Ovvero l'algoritmo esplora molti meno nodi e peggiora la soluzione trovata di molto poco. Ha le stesse proprietà di A*, tranne l'ottimalità che viene persa per $W > 1$.

Weighted A* come generalizzazione dei search algorithms

Si osserva che il Weighted A* search generalizza diversi algoritmi di ricerca per diversi valori di W:

W	$f(n)$	algoritmo
$W = 0$	$f(n) = g(n)$	Uniform-cost search o Dijkstra
$W = 1$	$f(n) = g(n) + h(n)$	A* search
$1 \leq W \leq \infty$	$f(n) = g(n) + W \cdot h(n)$	Weighted A* search
$W = \infty$	$f(n) = h(n)$	Greedy best-first search

3.5 Local search algorithms e ottimizzazione

I local search algorithms operano su problemi in cui il percorso che porta alla soluzione non è importante, ma conta solo lo stato finale. Ad esempio nei problemi di ottimizzazione non conta come si arriva alla soluzione, ma conta solo la qualità della soluzione stessa. Per risolvere questi problemi si utilizzano algoritmi di ottimizzazione tra cui gli interactive improvement algorithms.

Interactive improvement algorithms

Questa classe di algoritmi mantiene una singola configurazione dello stato alla volta e cerca di migliorare tale stato attraverso piccole modifiche locali, chiamate mosse (moves). Si parte da uno stato iniziale non ottimale e si sceglie di volta in volta la mossa che porta al miglioramento più significativo dello stato corrente in base a una funzione di valutazione detta objective function. Questo processo viene ripetuto fino a quando non sono più possibili miglioramenti o viene raggiunto un criterio di arresto.

Rappresentazione dello spazio degli stati

Risulta utile rappresentare lo spazio degli stati come un grafico bidimensionale in cui nelle ascisse sono riportati i vari stati e nelle ordinate i valori della funzione obiettivo associata a ciascuno stato. In questo modo è possibile visualizzare il processo di miglioramento dello stato come un percorso che si muove verso l'alto lungo il grafico, cercando di raggiungere il punto più alto che rappresenta la soluzione ottimale. Si definiscono i seguenti concetti:

- **global maximum**: lo stato associato al massimo globale della funzione obiettivo (soluzione ottima)
- **local maximum**: uno stato associato ad un massimo locale della funzione obiettivo (subottimale)
- **plateau**: una regione in cui la funzione obiettivo è costante per più stati, è anche detto "flat local maximum" se attorno non ha stati con valori più alti
- **shoulder**: una regione in cui la funzione obiettivo da crescente passa a costante per un certo numero di stati per poi tornare a crescere

Subottimalità degli interactive improvement algorithms

Siccome questi algoritmi esplorano solo una parte limitata dello spazio degli stati, potrebbero non esplorare mai la regione in cui si trova la soluzione ottimale globale, rimanendo bloccati in minimi/massimi locali o in loop infiniti in prossimità di shoulder o plateau. Per evitare questo problema, si possono utilizzare alcune tecniche come il random restart, il simulated annealing o gli algoritmi genetici.

Vantaggi degli interactive improvement algorithms

Il vantaggio principale degli interactive improvement algorithms è la loro efficienza in termini di tempo e spazio. Raggiungono rapidamente e in pochissimi passi soluzioni accettabili e lo stato che viene mantenuto in memoria è soltanto uno.

Hill-climbing - greedy local search

L'algoritmo di hill-climbing (o greedy local search) è un esempio di algoritmo che implementa a livello generale il principio degli interactive improvement algorithms.

0: **Algoritmo:** HILL-CLIMBING(P)
Input: P a problem formulation
Output: solution (not necessarily optimal)

1: create local variable node *current*
2: create local variable node *neighbor*
3: *current* ← initial state of P
4: **loop do**
5: *neighbor* ← a highest-value neighbor of *current*
6: **if** value of *neighbor* ≤ value of *current* **then**
7: **return** the solution corresponding to *current* ▷ local maximum reached
8: *current* ← *neighbor* ▷ iteractive step
9: **end loop**

Random restart hill-climbing

Il random restart hill-climbing è una variante dell'algoritmo di hill-climbing che prevede di eseguire più volte l'algoritmo a partire da stati iniziali casuali diversi. In questo modo si aumentano le probabilità di raggiungere la soluzione ottimale globale evitando di rimanere bloccati in minimi/massimi locali. La soluzione finale è la migliore tra tutte quelle trovate nelle varie esecuzioni.

Random sideways moves hill-climbing

Il random sideways moves hill-climbing è una variante dell'algoritmo di hill-climbing che prevede di consentire mosse laterali casuali (random sideways moves) per un certo numero di iterazioni consecutive quando si raggiunge un plateau. In questo modo si cerca di superare il plateau (che porterebbero a loop infiniti). Non supera i massimi locali e non porta necessariamente alla soluzione ottimale globale.

Simulated annealing

Il simulated annealing è un processo di ottimizzazione ispirato al processo di ricottura dei metalli. L'algoritmo inizia con una temperatura elevata T che permette di accettare mosse peggiorative con una certa probabilità p che dipende anche dalla "badness" dello stato $\Delta E(x)$. In questo modo si favorisce l'esplorazione dello spazio degli stati superando eventuali local maximum. Con il passare del tempo, la temperatura viene gradualmente ridotta e la badness diminuisce di conseguenza, riducendo la probabilità di accettare mosse peggiorative e concentrandosi sulla ricerca della soluzione ottimale. La funzione di probabilità di accettazione delle mosse peggiorative per un certo stato x è data da:

$$p(x) \propto e^{-\Delta E(x)/T}$$

Opportunamente tarato permette di raggiungere la soluzione ottima con probabilità prossima a 1. Il simulated annealing è utilizzato in problemi di ottimizzazione complessi come VLSI layout o airplane scheduling. A pagina successiva è riportato il codice dell'algoritmo.

0: **Algoritmo:** SIMULATED ANNEALING(P, S)
Input: P a problem formulation, S a shedule, a mapping from time t to temperature T
Output: solution (not necessarily optimal)

1: create local variable node *current*
2: create local variable node *next*
3: create local variable temperature T
4: *current* $\leftarrow$ initial state of P
5: **for** $t \leftarrow 1$ to ∞ **do**
6: $T \leftarrow S(t)$
7: **if** $T = 0$ **then**
8: **return** the solution corresponding to *current* ▷ temperature is zero
9: *next* $\leftarrow$ a randomly selected neighbor of *current*
10: $\Delta E \leftarrow$ value of *next* - value of *current*
11: **if** $\Delta E > 0$ **then**
12: *current* $\leftarrow$ *next* ▷ improving move
13: **else**
14: generate a random number $0 \leq r \leq 1$
15: *current* $\leftarrow$ *next* with probability $e^{\Delta E/T}$ ▷ non-improving move

Local beam search

Il local beam search è un algoritmo di ricerca locale che mantiene un insieme di k stati (o configurazioni) contemporaneamente e li aggiorna iterativamente selezionando i k migliori stati tra tutti i figli generati dagli stati correnti. È diverso dal random restart in quanto gli stati dell'iterazione successiva dipendono da tutti gli stati dell'iterazione corrente, permettendo una migliore esplorazione dello spazio degli stati. Tuttavia, può ancora rimanere bloccato in minimi/massimi locali se tutti gli stati convergono verso la stessa regione dello spazio degli stati.

Stochastic beam search

Il stochastic beam search è una variante del local beam search che introduce un elemento di casualità nella selezione degli stati per l'iterazione successiva. La scelta comunque rimane influenzata verso gli stati migliorativi. In questo modo di cerca di evitare di far convergere tutti gli stati verso la stessa regione dello spazio degli stati. Tale idea prende ispirazione dalla selezione naturale e dall'evoluzione biologica.

Genetic algorithms

I genetics algorithms si basano in parte sullo stochastic beam search, ma introducono operazioni di crossover e mutazione ispirate alla ricombinazione genetica biologica che usa la natura per generare diversità. Il processo di divide in cinque fasi principali:

- **inizializzazione**: si crea una popolazione iniziale di stati casuali, oppure si utilizzano quelli provenienti da una precedente iterazione
- **valutazione**: si valuta la fitness (valore della funzione obiettivo) di ciascuno stato
- **selezione**: si selezionano gli stati migliori in base alla fitness per essere i genitori della prossima generazione (in analogo alla selezione naturale) introducendo eventualmente una leggera casualità
- **crossover/ricombinazione**: a coppie, si incrociano gli stati genitori in modo da formare nuovi stati figli che condividono una parte dello stato di ciascun genitore
- **mutazione**: si introducono piccole modifiche casuali negli stati figli per aumentare la diversità

Il processo viene ripetuto per un certo numero di generazioni o fino a quando non viene raggiunto un criterio di arresto.

I genetic algorithms richiedono che gli stati siano rappresentati come stringhe di simboli (tipicamente bit) che possono essere facilmente manipolate con le operazioni di crossover e mutazione.

4 Logic: logical agents, propositional logic and FOL

4.1 Knowledge-based agents

Definizione di Knowledge Base (KB)

Una Knowledge Base (KB) è una raccolta strutturata di conoscenze rappresentate in "sentences" (frasi) espresse con un inguaggio formale. Il linguaggio umano non è formale in quanto ambiguo e impreciso, quindi si utilizzano altri tipi di linguaggi (formali) come la propositional logic o la first order logic (FOL).

Definizione di inferenza

L'inferenza è il processo di derivazione di nuove conoscenze a partire da quelle già presenti nella KB. Si basa su regole logiche che permettono di combinare le frasi esistenti per ottenere nuove frasi. L'inferenza è usata da un agente intelligente per prendere decisioni, risolvere problemi e rispondere a domande basandosi sulla conoscenza rappresentata nella KB. In base al linguaggio formale scelto, esistono diversi metodi e algoritmi di inferenza.

Knowledge-based agent

Un knowledge-based agent è un agente intelligente che utilizza una Knowledge Base per prendere decisioni e agire nell'ambiente. Per costruire un KB agent (con la relativa KB) è possibile usare un approccio dichiarativo in cui si specifica come l'agente deve agire di fronte ad un certo stato dell'ambiente esterno. Un KB agent può essere visto da due punti di vista:

- **knowledge level**: si concentra sulla conoscenza pura che l'agente possiede e su come la utilizza per prendere decisioni (inferenza), è domain-specific overo dipende dal dominio in cui l'agente opera
- **implementation level**: si concentra sui dettagli tecnici e sulle strutture dati utilizzate per implementare il knowledge base e le operazioni di inferenza, è domain-independent ovvero è indipendente dal dominio in cui l'agente opera

L'agente può modificare la sua knowledge base nel tempo attraverso le sue percezioni dell'ambiente esterno e il risultato delle azioni eseguite. Di seguito un esempio di algoritmo di un KB agent.

0: **Algoritmo:** KB-AGENT(*percept*)
Input: *percept* the perception received from the environment
Output: action to be executed

1: use static variable KB, the agent's knowledge base
2: use static variable t, the current time step
3: Tell(KB, Make-Percept-Sentence($percept$, t)) ▷ update the KB with the new percept
4: $action \leftarrow$ Ask(KB, Make-Action-Query(t)) ▷ decide action based on the KB
5: Tell(KB, Make-Action-Sentence($action$, t)) ▷ update the KB with the new action
6: $t \leftarrow t + 1$
7: **return** *action*

4.2 Logic in general

Logica come linguaggio formale

La logica è un sistema formale per rappresentare conoscenze e permettere ragionamenti (inferenza) su di esse e ottenere nuove conoscenze. Le conoscenze derivate sono corrette se le premesse iniziali sono corrette e il processo di inferenza è valido. Un linguaggio logico formale è costituito da:

- **syntax**: regole che definiscono la struttura delle frasi ben formate (well-formed formulas - WFF)
- **semantics**: regole che definiscono il significato delle frasi ben formate, ovvero come interpretarle e valutarle come vere o false in un certo modello

Entailment o conseguenza logica

Una sentence α è una conseguenza logica di una knowledge base KB (α entails from KB) se α è vera in tutti i modelli in cui KB è vera. L'entailment è una relazione tra sentences basate sulla semantica (ovvero sul significato delle sentences). Si indica con:

$$KB \models \alpha$$

4.3 Propositional logic

Struttura della propositional logic

La logica proposizionale è il linguaggio logico formale più semplice. È costituito da proposizioni atomiche (atomi) che possono essere vere o false e si indicano con lettere maiuscole (es. P, Q, R). Esistono, inoltre, operatori logici che agiscono sulle proposizioni per formare proposizioni complesse di seguito elencati in ordine di precedenza decrescente:

- $\neg$: negazione (not), $\neg P$ è vera se P è falsa
- $\wedge$: congiunzione (and), $P \wedge Q$ è vera se sia P che Q sono vere
- $\vee$: disgiunzione (or), $P \vee Q$ è vera se almeno una tra P e Q è vera
- $\Rightarrow$: implicazione (if-then), $P \Rightarrow Q$ è falsa solo se P è vera e Q è falsa
- $\Leftrightarrow$: doppia implicazione (iff), $P \Leftrightarrow Q$ è vera se P e Q sono entrambe vere o entrambe false

Nota: l'implicazione $P \Rightarrow Q$ è sempre vera se le premesse P sono false, in quanto non è possibile dedurre nulla da premesse false. Invece, se P è vera, allora $P \Rightarrow Q$ è vera solo se Q è vera, altrimenti è falsa.

Vantaggi e svantaggi della propositional logic

I vantaggi sono:

- la propositional logic è dichiarativa, ovvero le sentence corrispondono a fatti concreti
- permette di rappresentare informazioni partial, disjunctive e negated
- è componibile, ovvero si possono combinare più sentence per formare nuove sentence
- è indipendente dal contesto, ovvero le sentence sono vere o false indipendentemente dal dominio

Lo svantaggio principale è che la propositional logic non è in grado di rappresentare concetti più complessi come oggetti, relazioni tra oggetti e quantificatori (es. "tutti", "esiste"). Per questi compiti si utilizza la first-order logic (FOL).

4.4 Inference in propositional logic

L'inferenza nella propositional logic può essere effettuata attraverso due modi:

- **inference by model checking**: elencando tutti i modelli possibili della knowledge base KB
- **inference by resolution**: dimostrando per assurdo che $KB \wedge \neg\alpha$ è insoddisfacibile

Di seguito la spiegazione in ciascun paragrafo separato per ogni passaggio.

Inference by model checking

L'inferenza tramite model checking consiste nell'elencare tutti i possibili modelli della knowledge base KB (ovvero costruire la tabella di verità della KB in funzione di tutte le variabili della KB) e vericare se nelle righe in cui KB è vera, anche la proposizione α è vera. Se questo vale, allora $KB \models \alpha$.

Tale metodo risulta poco vantaggioso quando il numero di proposizioni atomiche nella KB è elevato, in quanto il numero di modelli possibili cresce esponenzialmente con il numero di proposizioni atomiche.

Inference by resolution

La dimostrazione per risoluzione è un metodo di inferenza che si basa sulla dimostrazione per assurdo. Per verificare che $KB \models \alpha$, si dimostra che $KB \wedge \neg\alpha$ è insoddisfacibile, ovvero non esiste alcun modello in cui KB è vera e α è falsa. La dimostrazione avviene in due passaggi principali:

- **conversione in conjunctive normal form (CNF)** della KB e di $\neg\alpha$:
  si converte KB e $\neg\alpha$ in CNF, ovvero in una congiunzione di disgiunzioni di letterali, dove un letterale è una proposizione atomica o la sua negazione (equivalente al prodotto di somme), in particolare eliminando le implicazioni e le doppie implicazioni

$$(A \vee B) \wedge (\neg C \vee D \vee E) \wedge (\neg A \vee C \vee \neg D)$$

- **applicazione della regola di risoluzione** alla sentence $KB \wedge \neg\alpha$:
  si applica iterativamente la regola di risoluzione, che permette di unire due clausole che contengono letterali complementari (per esempio A e $\neg A$) per ottenere una nuova clausola che contiene tutti i letterali delle due clausole originali esclusi i letterali complementari

$$\frac{(B \vee \mathbf{A}) \wedge (\neg\mathbf{A} \vee C)}{B \vee C} \qquad \text{l'operazione di risoluzione si indica in frazione: } \frac{\text{before}}{\text{after}}$$

Se attraverso il processo di risoluzione si ottengono clausole (somme) che sono sempre vere (tautologie) perché ad esempio contengono sia A che $\neg A$, allora tali clausole si possono eliminare in quanto elementi neutri delle congiunzioni.

Se alla fine del processo di risoluzione e semplificazione, tra le clausole residue ce ne sono due di complementari (ad esempio A e $\neg A$), allora si ha contraddizione ($A \wedge \neg A = \square$ = false) e si forma una empty clause indicata con $\square$. Si conclude che $KB \wedge \neg\alpha$ è insoddisfacibile, e la tesi $KB \models \alpha$ risulta dimostrata per assurdo.

Nota: le due clausole che formano la empty clause $\square$ devono necessariamente coinvolgere $\neg\alpha$, altrimenti significa che KB contiene delle contraddizioni interne ed è insoddisfacibile di per sé.

4.5 First-order logic

Struttura della first-order logic FOL

La first order logic (FOL) è un linguaggio logico formale che estende la propositional logic introducendo nuovi elementi per rappresentare entità del mondo reale e le proprietà o relazioni tra di esse. Gli elementi base della sintassi della FOL sono:

- **costanti**: rappresentano oggetti specifici del dominio (es. *Alice*, *Bob*)
- **variabili**: rappresentano oggetti generici del dominio (es. x, y, z)
- **funzioni**: rappresentano relazioni tra oggetti o proprietà di oggetti (es. *padre*(x), *somma*(x, y)) e restituiscono un altro oggetto come risultato
- **predicati**: rappresentano proprietà o relazioni tra oggetti (es. *Uomo*(x), *Ama*(x, y)) e restituiscono un valore booleano (vero o falso) a seconda se la proprietà o relazione è soddisfatta o meno
- **connettivi logici**: come nella propositional logic, permettono di combinare più predicati per formare sentence complesse (in ordine di precedenza: $\neg$, $=$, $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$)
- **quantificatori**: permettono di esprimere proposizioni che coinvolgono insiemi di oggetti:
  - **quantificatore universale** ($\forall$): una proposizione è vera per tutti gli oggetti
  - **quantificatore esistenziale** ($\exists$): una proposizione è vera per almeno un oggetto

Le sentence della FOL si classificano in:

- **terms** (o termini): elementi che rappresentano oggetti e possono essere costanti, variabili o funzioni di altri oggetti o variabili (es. *Alice*, x, *padre*($Alice$), *somma*$(x, 5)$)
- **atomic sentences** (o frasi atomiche): proposizioni che coinvolgono predicati applicati su più termini e possono essere vere o false (es. *Uomo*(Alice), *Ama*(x, Bob))
- **complex sentences** (o frasi complesse): proposizioni formate combinando atomic sentences tramite connettivi logici e quantificatori (es. $\forall x \;\; Uomo(x) \Rightarrow Mortale(x)$)

Uso del quantificatore universale $\forall$

Il quantificatore universale $\forall$ si usa per affermare che una certa proprietà o relazione vale per tutti gli oggetti di un certo tipo. Spesso si usa in combinazione con l'implicazione per esprimere regole generali.

$$\forall x \;\; At(x, Oxford) \Rightarrow Smart(x) \qquad \text{everyone at Oxford is smart}$$

È possibile utilizzare anche un altro connettivo logico come la congiunzione, ma il significato cambia totalmente ed è bene fare attenzione a non confondere i due connettivi:

$$\forall x \;\; At(x, Oxford) \wedge Smart(x) \qquad \text{everyone is at Oxford and everyone is smart}$$

È possibile riscrivere una sentence con il quantificatore universale come congiunzione di proposizioni in cui al posto della variabile quantificata x si usano tutti i termini possibili del dominio:

$$\forall x \; At(x, Oxford) \Rightarrow Smart(x) \;\; \rightarrow \;\; \begin{array}{l} (At(KingJohn, Oxford) \Rightarrow Smart(KingJohn)) \;\wedge\; \ldots \\ \ldots \;(At(Richard, Oxford) \Rightarrow Smart(Richard)) \;\wedge\; \ldots \\ \ldots \;(At(Father(John), Oxford) \Rightarrow Smart(Father(John))) \;\wedge\; \ldots \end{array}$$

Uso del quantificatore esistenziale $\exists$

Il quantificatore esistenziale $\exists$ si usa per affermare che esiste almeno un oggetto nel dominio per cui una certa proprietà o relazione è vera. Spesso si usa in combinazione con la congiunzione per esprimere l'esistenza di un oggetto che soddisfa una certa condizione.

$$\exists x \;\; \neg At(x, Oxford) \wedge Smart(x) \qquad \text{someone who is not at Oxford is smart}$$

È possibile utilizzare anche un altro connettivo logico come l'implicazione, ma il significato cambia totalmente ed è bene fare attenzione a non confondere i due connettivi:

$$\exists x \;\; \neg At(x, Oxford) \Rightarrow Smart(x) \qquad \begin{array}{l} \text{there is someone not at Oxford who is smart} \\ \text{there is someone at Oxford who is not smart} \\ \text{there is someone at Oxford who is smart} \end{array}$$

È possibile riscrivere una sentence con il quantificatore esistenziale come disgiunzione di proposizioni in cui al posto della variabile quantificata x si usano tutti i termini possibili del dominio:

$$\exists x \ \neg At(x, Oxford) \wedge Smart(x) \ \rightarrow \begin{array}{l} (\neg At(KingJohn, Oxford) \wedge Smart(KingJohn)) \ \vee \ \ldots \\ \ldots \ (\neg At(Richard, Oxford) \wedge Smart(Richard)) \ \vee \ \ldots \\ \ldots \ (\neg At(Father(John), Oxford) \wedge Smart(Father(John))) \ \vee \ \ldots \end{array}$$

Proprietà dei quantificatori

I quantificatori universale e esistenziale hanno le seguenti proprietà:

- $\forall x \, \forall y \ P(x) \ \equiv \ \forall y \, \forall x \ P(x) \ \equiv \ \forall x, y \ P(x)$
  il quantificatore universale gode della proprietà commutativa e può essere abbreviato

- $\exists x \, \exists y \ P(x) \ \equiv \ \exists y \, \exists x \ P(x) \ \equiv \ \exists x, y \ P(x)$
  il quantificatore esistenziale gode della proprietà commutativa e può essere abbreviato

- 1. $\forall y \, \exists x \ P(x, y) \ \rightarrow$ per ogni valore di y, esiste un x per cui vale P(x,y)
  2. $\exists x \, \forall y \ P(x, y) \ \rightarrow$ esiste un x per cui vale P(x,y) per ogni valore di y

  il quantificatore esistenziale e quello universale insieme non godono della proprietà commutativa, infatti la seconda implica la prima, ma non è detto che la prima implichi la seconda (nella prima non è detto che x sia lo stesso per ogni y come invece è richiesto nella seconda)

- $\forall x \ P(x) \ \equiv \ \neg \exists x \ \neg P(x)$
  la proposizione universale è equivalente alla negazione di una proposizione esistenziale con la negazione all'interno

- $\exists x \ P(x) \ \equiv \ \neg \forall x \ \neg P(x)$
  la proposizione esistenziale è equivalente alla negazione di una proposizione universale con la negazione all'interno

- $\forall x \ P(x) \wedge Q(x) \ \equiv \ (\forall x \ P(x)) \wedge (\forall x \ Q(x))$
  il quantificatore universale distribuisce la congiunzione

- $\exists x \ P(x) \vee Q(x) \ \equiv \ (\exists x \ P(x)) \vee (\exists x \ Q(x))$
  il quantificatore esistenziale distribuisce la disgiunzione

Sostituzione di variabili

La sostituzione di variabili è un'operazione fondamentale nella FOL che consiste nel sostituire una variabile x con un termine t in una formula P. La sostituzione permette di produrre nuova conoscenza a partire da quella già presente nella KB. Attraverso la sostituzione si passa dalla FOL alla propositional logic. La sostituzione si indica con una lettera greca (es. ϕ) come segue:

$$\phi = \{variable_1 \ / \ term_1, \ variable_2 \ / \ term_2\}$$

Ad esempio applicando la sostituzione ϕ alla formula S si ottiene:

$$\begin{array}{l} S = Smarter(x, y) \\ \phi = \{x \ / \ Hillary, \ y \ / \ Bill\} \end{array} \rightarrow \begin{array}{l} S_\phi = Subst(\{x \ / \ Hillary, \ y \ / \ Bill\}, Smarter(x, y)) \\ \quad = Smarter(Hillary, \ Bill) \end{array}$$

Applicando le sostituzioni ai KB agents, quando viene fatta una query alla KB, la risposta è l'insieme di alcune o tutte le possibili sostituzioni che rendono vera la query per la KB. In generale:

$$Ask(KB, \ S) \quad \text{restituisce} \quad \phi \text{ tale che } KB \models S_\phi$$

Ad esempio per il problema del Wumpus World, supponiamo che al tempo $t = 5$ l'agente percepisca *Stench*, *Breeze* e *Glitter* e voglia sapere quale azione eseguire (la risposta sarà raccogliere ciò che luccica). La query alla KB e la risposta sono le seguenti:

assertion at $t = 5$:	$Tell(KB, \ Percept([Stench, \ Breeze, \ Glitter], \ 5))$	
query at $t = 5$:	$Ask(KB, \ \exists a \ Action(a, 5))$	
answer at $t = 5$:	$\phi_5 = \{a \ / \ Grab\}$	con $KB \models Action(a, \ 5)_\phi$

4.6 Inference in FOL

L'inferenza nella FOL è più complessa rispetto alla propositional logic e può essere effettuata attraverso diversi metodi:

- **propositionalization**: convertendo le sentence della FOL in sentence della propositional logic
- **unification**: trovando sostituzioni che rendono due termini o due sentence identici
- **resolution and unification**: applicando la regola di risoluzione con l'unificazione
- **backward chaining**: utilizzando un approccio goal-driven per rispondere a query specifiche

Di seguito la spiegazione in ciascun paragrafo separato per ogni metodo.

Inference by propositionalization

Il primo metodo di inferenza nella FOL studiato è la proposizionalizzazione. Consiste nel convertire le sentence della FOL in sentence della propositional logic e applicare i metodi di inferenza della propositional logic (model checking o resolution). Per convertire una sentence della FOL in una sentence della propositional logic si effettuano i seguenti passaggi:

1. **existential instantiation**: eliminazione dei quantificatori esistenziali
2. **universal instantiation**: eliminazione dei quantificatori universali
3. **propositionalization**: sostituzione di tutti i predicati e termini con proposizioni atomiche

Di seguito la spiegazione in ciascun paragrafo separato per ogni passaggio.

1. Existential instantiation - EI

L'existential instantiation è un procedimento che serve per generare una nuova sentence senza quantificatori esistenziali a partire da una sentence con quantificatori esistenziali. In pratica consiste nel sostituire la variabile quantificata x con un nuovo simbolo detto Skolem constant k ovvero un termine che non compare altrove nella KB. Se la variabile quantificata è all'interno del scope di un quantificatore universale per un'altra variabile y, allora si usa la Skolem function $F(y$. In questo caso la Skolem function prende come argomento la variabile y e restitituisce un termine unico per ogni valore di y. In generale la existential instantiation si indica come segue:

$$\text{con Skolem constant } k \quad \frac{\exists x\ \alpha}{Subst(\{x\ /\ k\},\ \alpha)} \qquad \text{con Skolem function } F(y) \quad \frac{\forall y\ \exists x\ \alpha}{Subst(\{x\ /\ F(y)\},\ \alpha)}$$

È possibile applicare l'existential instantiation soltanto una volta per ogni quantificatore esistenziale presente. Il nuovo KB ottenuto non è logicamente equivalente al KB di partenza, ma rimane soddisfacibile se il KB di partenza è soddisfacibile. Di seguito due esempi con una Skolem constant e una Skolem function:

$$\frac{\exists x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)}{King(k) \wedge Greedy(k) \Rightarrow Evil(k)} \qquad \frac{\forall y\ \exists x\ Loves(x, y)}{Loves(F(y), y)}$$

2. Universal instantiation - UI

L'universal intantiation è un procedimento che serve per generare nuove sentence senza variabili o quantificatori universali a partire da una sentence con un quantificatore universale. In pratica consiste nel sostituire la variabile quantificata x con un ground term g, ovvero un termine che non contiene variabili (ad esempio una costante o una funzione di costanti). Le sentence ottenute sono dette ground instances della sentence originale. In generale la universal instantiation si indica come segue:

$$\frac{\forall x\ \alpha}{Subst(\{x\ /\ g\},\ \alpha)} \qquad \text{con } g \text{ ground term}$$

È possibile applicare la universal instantiation più volte usando sostituzioni diverse, ottenendo di conseguenza ground instances diverse della stessa sentence originale. Il nuovo KB ottenuto dalle UI rimane

logicamente equivalente al KB di partenza. Di seguito un esempio:

$$\frac{\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)}{\left\{ \begin{array}{cc} King(John) \wedge Greedy(John) \Rightarrow Evil(John), & \text{per } \phi_1 = \{x/John\} \\ King(Father(John)) \wedge Greedy(Father(John)) \Rightarrow Evil(Father(John)), & \text{per } \phi_2 = \{x/Father(John)\} \\ \dots & \end{array} \right\}}$$

3. Propositionalization

La proposizionalizzazione è l'ultimo passaggio per convertire una sentence della FOL in una sentence della propositional logic. Consiste nel sostituire ogni predicato con una proposizione atomica unica. Di seguito un esempio:

$$\frac{King(John) \wedge Greedy(John) \Rightarrow Evil(John)}{KingJohn \wedge GreedyJohn \Rightarrow EvilJohn}$$

Problemi dell'inferenza della FOL attraverso la proposizionalizzazione

Durante i processi di universal instantiation e proposizionalizzazione, si possono generare un numero infinito di sentence della propositional logic, tali per cui la maggior parte di esse risulta superflua e inutile ai fini dell'inferenza. Specialmente se alcuni predicati possono essere ricorsivi (come ad esempio $Father(x)$). L'unico caso in cui tale metodo risulta vantaggioso è quando la KB contiene un numero piccolo e finito di costanti, funzioni e predicati, per mantenere il numero di sentence della propositional logic generato a un livello gestibile.

Inference by unification

Il processo di unificazione è un metodo di inferenza che consiste nel trovare una sostituzione ϕ che rende due termini o due sentence identici. La sostituzione ϕ è detta unificatore. L'unificazione è utile per confrontare e combinare sentence che contengono variabili, funzioni e predicati. Di seguito un esempio di unificazione tra due sentence α e β con la relativa unificazione θ:

$$\alpha = \forall x\ Knows(John,\ x), \qquad \beta = \forall y\ Knows(y,\ Mary), \qquad \theta = Unify(\alpha, \beta) = \{x/Mary, y/John\}$$

$$\alpha_\theta = \beta_\theta = Subst(\theta, \alpha) = Subst(\theta, \beta) = Knows(John, Mary)$$

È possibile che le due sentence contengano la stessa variabile quantificata e di conseguenza non sia possibile unificarle. Per risolvere questo problema si ricorre alla standardizzazione delle variabili, ovvero si rinominano le variabili quantificate in modo che ogni quantificatore abbia una variabile unica.

Inference by resolution and unification

L'inferenza per risoluzione nella FOL funziona in modo simile a quella della propositional logic, ma con alcune differenze dovute alla presenza di variabili, funzioni e quantificatori. In analogia alla propositional logic, per dimostrare che $KB \models \alpha$, si utilizza la dimostrazione per assurdo, verificando che $KB \wedge \neg\alpha$ è insoddisfacibile. Per applicare la risoluzione nella FOL, si seguono i seguenti passaggi:

1. **conversione in conjunctive normal form (CNF)** della KB e di $\neg\alpha$
2. **applicazione della regola di risoluzione** tramite l'unificazione

Di seguito la spiegazione in ciascun paragrafo separato per ogni passaggio.

1. Conversione in CNF nella FOL

La conversione in CNF (congiunzione di disgiunzioni) nella FOL avviene in più passaggi:

1. eliminazione delle implicazioni e delle doppie implicazioni (avendo solo congiunzioni o disgiunzioni)
2. riduzione della negazione all'interno (con le leggi di De Morgan e le proprietà dei quantificatori)
3. standardizzazione delle variabili (rinominazione delle variabili con stesso nome)
4. applicazione della existential instantiation per eliminare i quantificatori esistenziali
5. eliminazione dei quantificatori universali semplicemente rimuovendoli (si sottintende che tutte le variabili siano quantificate universalmente per cui non ha senso specificarne il quantificatore)
6. distribuzione della disgiunzione sulla congiunzione per ottenere la CNF

2. Risoluzione con unificazione

A differenza della risoluzione nella propositional logic, in cui si uniscono due clausole che contengono letterali complementari, nella FOL è necessario prima unificare i letterali complementari per renderli identici. In pratica si calcola la sostituzione θ che unifica i letterali complementari tra due clausole, si applica la sostituzione alle clausole come nel processo di unificazione ed infine si possono unire le clausole come nella risoluzione della propositional logic. In generale si indica come segue:

$$\frac{(B \vee \mathbf{A}) \wedge (\neg \mathbf{A} \vee C)}{Subst(\theta, B \vee C)} \qquad \text{con } \theta = Unify(A, \neg A)$$

Di seguito un esempio pratico di risoluzione con unificazione:

$$\frac{(\neg Rich(x) \vee Unhappy(x)) \wedge Rich(Ken)}{Unhappy(Ken)} \qquad \text{con } Unify(\neg Rich(x), \neg Rich(Ken)) = \{x/Ken\}$$

Come per la propositional logic, se alla fine del processo di risoluzione e semplificazione, tra le clausole residue si ottiene la empty clause $\square$, allora si ha contraddizione. Si conclude, quindi, che $KB \wedge \neg\alpha$ è insoddisfacibile, e la tesi $KB \models \alpha$ risulta dimostrata per assurdo.

Inference by backward chaining

L'inferenza per backward chaining è un metodo di inferenza goal-driven che parte da una query specifica e cerca di risalire alle premesse che devono essere vere per soddisfare la query. In pratica si parte dalla query e si cercano le sostituzioni o le regole nella KB che soddisfano tutte le sentence della KB. Se si trova che una delle premesse è falsa, oppure che non può essere dimostrata, allora si conclude che la query non può essere soddisfatta.

Tale metodo è molto più efficiente e più veloce rispetto alla risoluzione con unificazione, in quanto si concentra solo sulle parti rilevanti della KB per rispondere alla query, evitando di esplorare l'intera KB. Lavora in modo simile al DFS (depth-first search) ed è usato dal linguaggio Prolog. L'unico svantaggio è che non è ottimale.

4.7 Knowledge engineering in FOL

La knowledge engineering è il processo di progettazione, costruzione e manutenzione delle knowledge base per garantire che esse siano accurate, complete ed efficienti per poter essere usate su KB agents. Si divide in diverse fasi:

1. **identificazione della domanda**: analogo alla problem formulation attraverso gli agent's PEAS
2. **raccolta della conoscenza**: raccolta delle informazioni rilevanti dal dominio di interesse
3. **ontologia**: definizione delle entità, delle relazioni e delle regole del dominio
4. **codifica della conoscenza**: rappresentazione della conoscenza in un linguaggio logico formale
5. **codifica di un'istanza del problema**: rappresentazione di un'istanza specifica del problema
6. **fare query alla KB**: utilizzo di metodi di inferenza per rispondere a query specifiche
7. **degub e valutazione**: verifica della correttezza e dell'efficienza della KB

5 Bayesian networks and probabilistic reasoning

5.1 Gestione dell'incertezza

Introduzione alla gestione dell'incertezza

Analizzando un agente intelligente si nota che il mondo osservato spesso contiene incertezze dovute a vari fattori come partial observability, processi stocastici, rumore nei sensori e comportamenti imprevedibili di altri agenti. Un agente intelligente (che agisce razionalmente) deve essere in grado di tenere conto di queste incertezze per prendere decisioni informate e agire in modo efficace. In generale esistono tre motivi principali per cui un agente intelligente che non è un grado di gestire l'incertezza potrebbe fallire di fronte a situazioni complesse:

- **lazyness**: un certo fenomeno può avere molte cause possibili, elencarle tutte richiede troppo tempo e risorse computazionali ed è necessario un sistema per selezionare solo le cause più probabili
- **theoretical ignorance**: un agente può non avere conoscenza completa del mondo, ad esempio non conoscere le leggi fisiche che regolano un fenomeno, è richiesto saper fare inferenze basate su osservazioni passate
- **practical ignorance**: anche se un agente ha conoscenza completa delle dinamiche del mondo, potrebbe non disporre di tutte le informazioni necessarie per definire totalmente lo stato dell'ambiente, ad esempio a causa di sensori imperfetti o limitati

Incertezza e agenti intelligenti

Quasi tutti i modelli di agenti intelligenti (model-based, goal-based, utility-based, ma non reflex-based) possono essere in grado di gestire l'incertezza:

- **reflex-based agents**: agiscono solo di fronte ad impulsi immediati (come l'arco riflesso) e non sono in grado di modulare la propria azione in base a informazioni incomplete o incerte, si usano infatti in ambienti fully observable, deterministici, di cui si conoscono tutte le dinamiche a priori
- **model-based agents**: per mantenere una rappresentazione interna dello stato del mondo di fronte ad ambienti non fully observable, l'agente deve essere in grado di gestire l'incertezza delle percezioni e aggiornare costantemente lo stato interno (ad esempio Kalman filters)
- **goal-based agents**: analogo al model-based agent, l'unico cambiamento è che l'agente è in grado di pianificare le proprie azioni per raggiungere un obiettivo, ma la gestione dell'incertezza rimane solo per l'aggiornamento dello stato interno
- **utility-based agents**: il vantaggio principale di questi agenti è la capacità di valutare e pesare le proprie azioni in base a una funzione di utilità che attraverso la decision theory considera sia i goal da raggiungere che la probabilità di successo delle azioni stesse, per questo motivo sono i più utilizzati per operare in ambienti con incertezza

Decision theory and utility based agents

La decision theory unisce l'utility theory degli agenti utility-based, per calcolare l'utilità attesa di ogni azione possibile, insieme alla probability theory che pesa l'utilità in base alla probabilità di ogni possibile risultato. In questo modo, un agente può scegliere l'azione che massimizza l'utilità attesa, tenendo conto sia dei benefici potenziali che dei rischi associati a ciascuna azione in un contesto di incertezza. Di seguito lo pseudocodice per un utility-based agent che utilizza la decision theory:

0: **Algoritmo:** DT-AGENT(*percept*)
Input: *percept*: percezione dell'ambiente che può contenere incertezza
Output: *action*: miglior azione da eseguire

1: variabile statica *belief-state* con lo stato dell'ambiente basato sul modello interno dell'agente
2: *belief-state* ← UPDATE-BELIEF-STATE(*belief-state*, *percept*, *sensor-model*)
3: *action-outcomes* ← ACTIONS-OUTCOMES-PROBABILITY(*belief-state*, *transition-model*)
4: *best-action* ← SELECT-HIGHEST-UTILITY-ACTION(*action-outcomes*, *utility-function*)
5: **return** *best-action*

5.2 Teoria della probabilità

Introduzione alla probabilità - definizioni e nomenclatura

- **probabilità**: funzione matematica $P(\omega)$ che associa un valore numerico ad ogni possibile world $\omega \in W$ proveniente da uno spazio di mondi W
- **evento**: sottoinsieme di possibili outcomes
- **random variable**: variabile che rappresenta una proprietà o funzionalità di un processo attraverso un insieme di possibili numeri
- **assimomi della probabilità**: $0 \leq P(e) \leq 1, \quad \sum_{e \in W} P(e) = 1$
- **joint probability** o probabilità congiunta: $P(e_1, e_2, \dots, e_n) = P(e_1 \wedge e_2 \wedge \dots \wedge e_n)$
- **unconditional probability** o prior probability: $P(e)$, indica il grado di credenza in un evento in assenza di informazioni aggiuntive
- **conditional probability** o posterior probability: $P(e_1 \mid e_2) = P(e_1, e_2)/P(e_2)$, indica la probabilità che un evento si verifichi dato che un altro evento è già noto
- **independence**: due eventi a e b sono indipendenti se $P(a \mid b) = P(a)$ oppure se $P(b \mid a) = P(b)$ oppure se $P(a, b) = P(a)P(b)$, per indicare l'indipendenza tra due variabili si scrive $A \perp\!\!\!\perp B$
- **conditional independence**: due eventi a e b sono condizionatamente indipendenti dato un terzo evento c se $P(a \mid b, c) = P(a \mid c)$ oppure se $P(b \mid a, c) = P(b \mid c)$ oppure se $P(a, b \mid c) = P(a \mid c)P(b \mid c)$, per indicare l'indipendenza condizionata tra due variabili si scrive $A \perp\!\!\!\perp B \mid C$
- **law of total probability**: data una partizione dello spazio degli eventi $\{c_1, c_2, \dots, c_n\}$, allora per ogni evento a vale che $P(a) = \sum_i P(a \mid c_i)P(c_i)$
- **probability distribution**: insieme di tutte le probabilità associate a tutti i possibili valori di una variabile casuale X: $\mathbf{P}(X) = \{P(x_1), P(x_2), \dots\}$
- **probability density function** o pdf: la probabilità che la variabile aleatoria continua X assuma uno specifico valore x è detta densità di probabilità: $p(X = x) \equiv p(x)$, con $\int p(x)dx = 1$

Bayes rules e product rule

$$\text{Product rule}: \quad P(a, b) = P(a \mid b)P(b) \qquad \text{Bayes rule}: \quad \begin{aligned} P(a \mid b) &= P(b \mid a)P(a)/P(b) \\ P(b \mid a) &= P(a \mid b)P(b)/P(a) \end{aligned}$$

La bayes rule viene usata per calcolare la probabilità che una certa causa a abbia generato un certo effetto b ($P(a \mid b)$) conoscendo la probabilità con cui la causa a provoca l'effetto b ($P(b \mid a)$) e la probabilità a priori della causa a ($P(a)$) e dell'effetto b ($P(b)$).

In base all'ordine delle due variabili a e b si ottengono due tipi di relazioni:

- **causal relationship**: $P(\textit{effect} \mid \textit{cause})$ indica la probabilità che un certo effetto si verifichi dato che una certa causa è presente, ad esempio $P(\text{mal di testa} \mid \text{influenza})$ indica la probabilità di avere mal di testa dato che si ha l'influenza
- **diagnostic relationship**: $P(\textit{cause} \mid \textit{effect})$ indica la probabilità che una certa causa sia presente dato che un certo effetto si è verificato, ad esempio $P(\text{influenza} \mid \text{mal di testa})$ indica la probabilità di avere l'influenza dato che si ha mal di testa

Fattorizzazione e indipendenza

La fattorizzazione, insieme alle nozioni di indipendenza e indipendenza condizionata, permettono di riscrivere una distribuzione di probabilità congiunta in termini di probabilità condizionate più semplici eliminando ridondanze e semplificando i calcoli. In pratica si sfrutta la product rule per esprimere la probabilità congiunta come il prodotto di probabilità condizionate, e si utilizzano le proprietà di indipendenza per semplificare ulteriormente queste probabilità condizionate.

$$\mathbf{P}(A, B, C, D) = \mathbf{P}(A \mid B, C, D) \cdot \mathbf{P}(B, C, D) = \mathbf{P}(A) \cdot \mathbf{P}(B, C, D) \qquad \text{con } A \perp\!\!\!\perp B, C, D$$

Chain Rule e product decomposition

La chain rule si basa sulla applicazione ricorsiva della product rule per esprimere una distribuzione di probabilità congiunta di più variabili aleatorie come il prodotto di tante probabilità condizionate più semplici in cui applicare le opportune semplificazioni per l'indipendenza e indipendenza condizionata.

$$\mathbf{P}(A_1, A_2, \ldots, A_n) = \prod_{i=1}^{n} \mathbf{P}(A_i \mid A_1, A_2, \ldots, A_{i-1})$$

Ad esempio, per tre variabili aleatorie x_1, x_2, x_3, in base alle indipendenze che sussitono tra le tre variabili, si può avere che:

$$\mathbf{P}(x_1, x_2, x_3) = \mathbf{P}(x_1 \mid x_2, x_3)\mathbf{P}(x_2 \mid x_3)\mathbf{P}(x_3) = \left\langle \begin{array}{ll} \mathbf{P}(x_3 \mid x_2)\mathbf{P}(x_2 \mid x_1)\mathbf{P}(x_1) & x_3 \perp\!\!\!\perp x_1 \mid x_2 \\ \mathbf{P}(x_3 \mid x_2, x_1)\mathbf{P}(x_2)\mathbf{P}(x_1) & x_2 \perp\!\!\!\perp x_1 \\ \ldots & \end{array} \right.$$

5.3 Inference using full joint distribution

Struttura di una full joint distribution

Una **full joint distribution** (FJD) è una rappresentazione completa in struttura tabellare di tutte le possibili combinazioni di valori delle variabili aleatorie in un dominio, a cui, ad ogni combinazione, è associata la relativa probabilità congiunta. La FJD consente di rispondere a qualsiasi domanda riguardante qualsiasi tipo di probabilità (a priori, a posteriori, congiunta) di qualsiasi evento o combinazione di eventi nel dominio utilizzando le opportune regole della probabilità e ricorrendo eventualemnte a tecniche semplificative come normalizzazione e marginalizzazione.

L'univo svantaggio della FJD è che la sua dimensione cresce esponenzialmente con il numero di variabili aleatorie coinvolte, rendendo impraticabile la sua costruzione e utilizzo in domini complessi con molte variabili. Richiede infatti di calcolare e memorizzare qualsiasi combinazione possibile di valori delle variabili aleatorie, il che potrebbe essere un grande spreco di tempo.

Calcolo della probabilità (a priori o congiunta) di un evento specifico

Per calcolare la probabilità di un evento specifico (o la probabilità congiunta di più eventi) utilizzando una FJD, si procede semplicemente sommando le probabilità corrispondenti alle celle della tabella che soddisfano le condizioni dell'evento di interesse.

Calcolo della probabilità a posteriori usando la normalizzazione

Nel caso si volesse calcolare la distribuzione di probabilità condizionate $\mathbf{P}(A \mid B)$ utilizzando una FJD, si osserva che compare un fattore di normalizzazione α (proveniente dalla Bayes rule) che serve per garantire che la somma delle probabilità condizionate sia pari a 1. Questo fattore si calcola come l'inverso della somma delle probabilità congiunte:

$$\mathbf{P}(A \mid b) = \mathbf{P}(A, b)/P(b) = \alpha\mathbf{P}(A, b) \qquad \text{con } \alpha = 1/P(b) \text{ che in generale soddisfa } \alpha \sum_{a \in A} \mathbf{P}(A, b) = 1$$

Calcolo della probabilità a posteriori usando la marginalizzazione

Nel caso in cui si voglia calcolare la distribuzione di probabilità condizionata $\mathbf{P}(A \mid b)$ utilizzando una FJD che include anche altre variabili aleatorie C di cui non si ha interesse o non si conosce il valore, è necessario ricorrere alla tecnica della marginalizzazione per eliminare le variabili non rilevanti. La marginalizzazione consiste nel sommare le probabilità congiunte su tutte le possibili combinazioni di valori delle variabili da eliminare, in modo da ottenere una distribuzione di probabilità condizionata che dipende solo dalle variabili di interesse.

$$\mathbf{P}(A \mid b) = \alpha\mathbf{P}(A \mid b) = \alpha \sum_{c \in C} \mathbf{P}(A, b, c)$$

Si applica spesso quando si vuole calcolare la probabilità a posteriori di certe variabili A data una certa evidenza b per l'insieme di variabili B, quando l'ambiente ha anche altre variabili C nascoste.

5.4 Bayesian networks

Parent set di una variabile aleatoria

L'insieme di variabili che condizionano una variabile aleatoria è chiamata **parent set** ed è un sottoinsieme delle variabili che precedono la variabile aleatoria condizionata. È possibile quindi riscrivere la chain rule condizionando ciascuna variabile aleatoria solo sul suo parent set:

$$\mathbf{P}(A_1, A_2, \ldots, A_n) = \prod_{i=1}^{n} \mathbf{P}(A_i \mid A_1, A_2, \ldots, A_{i-1}) = \prod_{i=1}^{n} \mathbf{P}(A_i \mid \text{Parents}(A_i))$$

$$\text{con Parents}(A_i) \subseteq \{A_1, A_2, \ldots, A_{i-1}\}$$

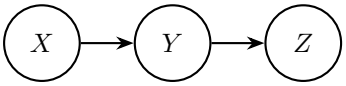
Definizione di bayesian network

Una bayesian network (BN) (anche detta "belief network", "probabilistic graphic model" (PGM) o "direct acyclic graph" (DAG)) è una rappresentazione grafica sottoforma di grafo diretto aciclico che modella le relazioni di dipendenza probabilistica tra un insieme di variabili aleatorie. In una BN, ogni nodo rappresenta una variabile aleatoria, mentre gli archi diretti tra i nodi indicano le relazioni di dipendenza condizionata tra le variabili. Gli archi entranti di un nodo rappresentano le variabili che condizionano quella variabile, ovvero provengono dal parent set di quella variabile.

Esistono tre tipi di strutture di dipendenza che possono sussistere tra le variabili aleatorie in una BN di seguito illustrate nei tre seguenti paragrafi

1. Chain

La chain prevede una connessione "head-to-tail" tra le variabili aleatorie. Si hanno le seguenti relazioni di dipendenza:

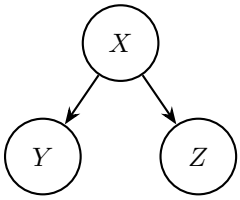


$$X \to Y, Z \qquad Y \to Z \qquad X \perp\!\!\!\perp Z \mid Y$$

$$\mathbf{P}(X, Y, Z) = \mathbf{P}(Z \mid Y)\, \mathbf{P}(Y \mid X)\, \mathbf{P}(X)$$

2. Fork

La fork prevede una connessione "tail-to-tail" tra le variabili aleatorie. Si hanno le seguenti relazioni di dipendenza:

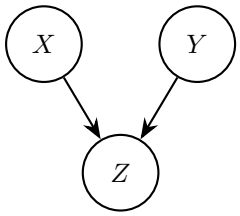


$$X \to Y, Z \qquad Y \perp\!\!\!\perp Z \mid X$$

$$\mathbf{P}(X, Y, Z) = \mathbf{P}(Y \mid X)\, \mathbf{P}(Z \mid X)\, \mathbf{P}(X)$$

3. Collider

La collider prevede una connessione "head-to-head" tra le variabili aleatorie. Si hanno le seguenti relazioni di dipendenza:



$$X \to Z \qquad Y \to Z \qquad X \perp\!\!\!\perp Y \qquad X \not\perp\!\!\!\perp Y \mid Z$$

$$\mathbf{P}(X, Y, Z) = \mathbf{P}(Z \mid X, Y)\mathbf{P}(X)\mathbf{P}(Y)$$

$$\mathbf{P}(X, Y) = \mathbf{P}(X)\mathbf{P}(Y) \qquad \text{dato } X \perp\!\!\!\perp Y$$

$$\mathbf{P}(X, Y \mid Z) = \mathbf{P}(X \mid Y, Z)\mathbf{P}(Y \mid Z) \qquad \text{dato } X \not\perp\!\!\!\perp Y \mid Z$$

La particolarità di questa struttura (su cui bisogna fare attenzione) è che le variabili X e Y sono indipendenti a priori, ma diventano dipendenti condizionatamente alla variabile Z.

Conditional probability tables - CPTs

Le conditional probability tables (CPTs) sono tabelle associate a ciascuna variabile aleatoria (cioè a ciascun nodo del grafo) che specificano la distribuzione di probabilità condizionata di quella variabile dato il suo parent set. Per questioni di ottimizzazione dello spazio di memoria, le CPTs considerano solo il caso in cui la variabile condizionata assuma il valore vero, in quanto il suo complementare (falso) può essere calcolato facilmente come $P(\neg a \mid \text{Parents}(A)) = 1 - P(a \mid \text{Parents}(A))$.

Nota: se una variabile aleatoria non ha genitori (parent set vuoto), ovvero non dipende da nessun'altra variabile, allora la sua CPT coincide con la sua distribuzione di probabilità a priori.

Di seguito un esempio di CPT per le varie strutture di dipendenza illustrate nei paragrafi precedenti:

Chain ($Y \to Z$)

Y	$P(Z = true \mid Y)$
true	0.9
false	0.2

Fork ($Y \leftarrow X \to Z$)

X	$P(Y = true \mid X)$
true	0.8
false	0.1

Collider ($X \to Z \leftarrow Y$)

X	Y	$P(Z = true \mid X, Y)$
true	true	0.9
true	false	0.6
false	true	0.7
false	false	0.1

Vantaggi delle bayesian networks e delle CPTs

Attraverso le bayesian networks e le CPTs è possibile rappresentare tutte le probabilità congiunte di un dominio in modo più compatto ed efficiente rispetto a una full joint distribution (FJD) riducendo notevolmente lo spazio in memoria occupato.

Inoltre, avendo meno parametri da memorizzare, è possibile stimarli in maniera più accurata se si dispone di un numero limitato di samples di addestramento rispetto ad una struttura più complessa come una FJD. Infatti diminuendo il numero di parametri aumenta il numero di samples per parametro e di conseguenza anche l'accuratezza della stima.

Costruzione delle bayesian networks

Qualsiasi ordinamento topologico delle variabili aleatorie di un dominio consente di costruire una bayesian network corretta per il problema. Infatti la dipendenza tra le variabili aleatorie è simmetrica per cui è possibile invertire l'ordine di qualsiasi coppia di variabili aleatorie senza alterare la correttezza della BN (sempre evitando cicli).

Tuttavia, per costruire una BN più compatta ed efficiente è necessario scegliere un ordinamento topologico in cui le dipendenze tra le variabili aleatorie siano orientate secondo il rapporto di causalità (causa $\to$ effetto), ovvero dove il parent set di ogni variabile è l'insieme delle cause dirette di tale variabile.

Inference by enumeration in bayesian networks

Per calcolare la probabilità a posteriori di una variabile aleatoria X data una certa evidenza e si utilizza la tecnica dell'inference by enumeration, che consiste nel calcolare la probabilità congiunta $P(X, e)$ sommando su tutte le possibili combinazioni di valori delle variabili aleatorie nascoste Y (cioè quelle che non sono né X né e) e normalizzando il risultato per ottenere una distribuzione di probabilità condizionata:

$$\mathbf{P}(X \mid e) = \alpha \sum_{y} \mathbf{P}(X, e, y) \qquad \text{con } \alpha = \frac{1}{\sum_x \sum_y \mathbf{P}(x, e, y)} \text{ dalla product rule}$$

5.5 Approximate inference

Vantaggi dell'approximate inference

I metodi di inferenza visti in precedenza (inference by enumeration) restituiscono risultati esatti, ma il costo computazionale per calcolarli risulta parecchio elevato, specialmente per domini complessi con molte variabili aleatorie e tante relazioni di dipendenza. Per questo motivo, in molti casi, è preferibile utilizzare metodi di inferenza approssimata che restituiscono risultati non esatti ma comunque accettabili in tempi ragionevoli, riducendo notevolmente il costo computazionale.

Probabilistic sampling come metodo di approximate inference

Per calcolare la probabilità approssimata di un evento specifico è possibile ricorrere alla tecnica del probabilistic sampling che consiste nel generare un certo numero di campioni casuali (samples) da una distribuzione di probabilità e utilizzare questi campioni per stimare la probabilità dell'evento di interesse in base alla frequenza relativa con cui si verifica tale evento nei campioni generati. Man mano che la quantità di campioni aumenta, l'approssimazione diventa via via sempre più accurata, avvicinandosi alla probabilità reale dell'evento.

Monte Carlo algorithms

Esistono vari algoritmi di probabilistic sampling che prendono il nome di Monte Carlo algorithms. Si classificano in tre categorie principali in base alla tecnica di campionamento utilizzata:

- **direct sampling**: si generano campioni casuali senza considerare l'evidenza osservata, sono molto facili da implementare, ma risultano molto inefficienti quando l'evidenza è rara perché tanti campioni vengono scartati; fanno parte di questa categoria algoritmi come prior sampling e rejection sampling
- **importance sampling**: si generano campioni casuali come nel direct sampling, ma si tiene conto anche dell'evidenza osservata, assegnando un peso a ciascun campione, evitando di scartare campioni; fanno parte di questa categoria algoritmi come likelihood weighting
- **Markov Chain Monte Carlo (MCMC) sampling**: si generano campioni casuali in modo iterativo, in cui ogni campione dipende dal campione precedente, formando una catena di Markov; fanno parte di questa categoria algoritmi come Gibbs sampling

Metodi di sampling di una variabile aleatoria

Per associare un valore ad una variabile aleatoria rispettando la sua distribuzione di probabilità discreta si possono utilizzare due metodi principali:

- **inverse transform sampling**: si calcola la funzione di distribuzione cumulativa (CDF) della distribuzione di probabilità, poi si genera un numero casuale u nell'intervallo $[0, 1]$ ed infine si trova il valore x tale che $F(x) = u$, ovvero l'inverso della CDF applicato a u: $x = F^{-1}(u)$ (vale anche con variabili aleatorie continue)
- **stochastic method**: si divide un segmento di lunghezza unitaria ($[0, 1]$) in n sottointervalli corrispondenti agli n valori della variabile aleatoria, in cui ogni sottointervallo ha una lunghezza proporzionale alla probabilità del valore corrispondente; si genera, quindi, un numero casuale u nell'intervallo $[0, 1]$ e si determina in quale sottointervallo cade u, per poi assegnare il valore corrispondente alla variabile aleatoria

Prior sampling - for prior probability only

Il prior sampling è il più semplice metodo di probabilistic sampling e fa parte della categoria dei direct sampling. In pratica generano sampling casuali per ogni variabile aleatoria. Siccome il sampling di ogni variabile utilizza le CPTs, è richiesto conoscere i valori assegnati alle variabili genitori, cioè serve processare le variabili secondo un ordinamento topologico della bayesian network.

Una volta generati un certo numero di campioni, si calcola la probabilità approssimata dell'evento di interesse in base alla frequenza relativa con cui si verifica tale evento nei campioni generati. La probabilità approssimata si indica con un "ˆ".

$$P(x_1, x_2, \dots) \approx \hat{P}(x_1, x_2, \dots) = \frac{\#\text{ samples in cui si verifica } (x_1, x_2, \dots)}{\#\text{ samples in totale}}$$

Nota: non è possibile calcolare probabilità condizionate dato che non si considera l'evidenza osservata (si chiama per l'appunto prior sampling). Per fare ciò si utilizza il rejection sampling.

Di seguito lo pseudocodice per il prior sampling:

Algoritmo: PRIOR-SAMPLING(*bn*, *num-samples*, X)
Input: *bn*: bayesian network con CPTs, *num-samples*, X variabile di interesse
Output: $\mathbf{P}(X)$: stima della probabilità a priori di X

C vettore che conta la frequenza di ogni simbolo di X nei campioni generati
for $i = 1$ to *num-samples* **do**
  sample ← GENERATE-PRIOR-SAMPLE(*bn*)
  $C[j] \leftarrow C[j] + 1$ con j tale per cui vale $sample[X] = x_j$
return NORMALIZE(C) ▷ normalizza per avere un vettore di probabilità

Algoritmo: GENERATE-PRIOR-SAMPLE(*bn*)
Input: *bn*: bayesian network con le CPTs
Output: *sample*: campione generato

sample ← vuoto
for ogni variabile aleatoria X_i in *bn* secondo un ordinamento topologico **do**
  $sample[X_i]$ ← campiona un valore per X_i dalla sua CPT dati *parent-values* già precalcolati
return *sample*

Rejection sampling - specific for conditional probability

Il rejection sampling è un metodo di probabilistic sampling che fa parte della categoria dei direct sampling e, a differenza del prior sampling, permette di calcolare probabilità condizionate dato che considera l'evidenza osservata e. In pratica, si comporta allo stesso modo del prior sampling, soltanto che dopo aver generato un campione, si verifica se il campione soddisfa le condizioni dell'evidenza e e in caso contrario si scarta il campione appena calcolato.

Il rejection sampling risulta molto inefficiente, specialmente quando l'evidenza e è rara, in quanto la maggior parte dei campioni generati viene scartata, rendendo l'approssimazione costosa e imprecisa.

Di seguito lo pseudocodice per il rejection sampling, si nota che al suo interno viene richiamato il prior sampling per generare i campioni casuali.

Algoritmo: REJECTION-SAMPLING(*bn*, *num-samples*, X, e)
Input: *bn*, *num-samples* e X come nel prior-sampling, e: evidenza osservata
Output: $\mathbf{P}(X \mid e)$: stima della probabilità a posteriori di X data l'evidenza e

C vettore che conta la frequenza di ogni simbolo di X nei campioni generati
for $i = 1$ to *num-samples* **do**
  sample ← GENERATE-PRIOR-SAMPLE(*bn*)
  if *sample* soddisfa le condizioni dell'evidenza e **then**
    $C[j] \leftarrow C[j] + 1$ con j tale per cui vale $sample[X] = x_j$
return NORMALIZE(C) ▷ normalizza per avere un vettore di probabilità

Likelihood weighting

Il likelihood weighting è un metodo di probabilistic sampling che fa parte della categoria degli importance sampling. In pratica, si forza ogni campione a soddisfare le condizioni dell'evidenza e in modo da evitare di scartare campioni e sprecare tempo, inoltre si assegna a ciascun campione un peso che riflette la probabilità di osservare l'evidenza e dato il campione generato.

Tale algoritmo risulta molto più efficiente del rejection sampling, in quanto non si hanno campioni scartati, ma è comunque poco efficiente all'aumentare del numero di variabili di evidenza, in quanto il peso dei campioni tende a diminuire.

Di seguito lo pseudocodice per il likelihood weighting:

0: **Algoritmo:** LIKELIHOOD-WEIGHTING(*bn*, *num-samples*, X, e)
Input: *bn*, *num-samples*, X ed e come in precedenza
Output: $\mathbf{P}(X \mid e)$: stima della probabilità a posteriori di X data l'evidenza e

1: W vettore che somma i pesi di ogni simbolo di X nei campioni generati
2: **for** $i = 1$ to *num-samples* **do**
3: *sample*, $w \leftarrow$ GENERATE-WEIGHTED-SAMPLE(*bn*, e)
4: $W[j] \leftarrow W[j] + w$ con j tale per cui vale $sample[X] = x_j$
5: **return** NORMALIZE(W) ▷ normalizza per avere un vettore di probabilità

0: **Algoritmo:** GENERATE-WEIGHTED-SAMPLE(*bn*, e)
Input: *bn*: bayesian network con CPTs, e: evidenza osservata
Output: *sample*: campione generato, w: peso del campione

1: *sample* $\leftarrow$ sample vuoto ad eccezione dei valori specificati in e
2: $w \leftarrow 1$ ▷ peso iniziale del campione
3: **for** ogni variabile aleatoria X_i in *bn* secondo un ordinamento topologico **do**
4: **if** X_i è una variabile di evidenza (cioè è presente in e) **then**
5: $sample[X_i] \leftarrow$ valore di X_i specificato in e ▷ assegna a X_i il valore specificato in e
6: $w \leftarrow w \cdot P(X_i = sample[X_i] \mid \text{Parents}(X_i))$ ▷ aggiorna il peso del campione
7: **else**
8: $sample[X_i] \leftarrow$ campiona un valore per X_i dalla sua CPT dati *parent-values* già precalcolati
9: **return** *sample*, w

Markov Chain Monte Carlo (MCMC) methods

I Markov Chain Monte Carlo (MCMC) methods sono una classe di algoritmi di probabilistic sampling che generano campioni casuali cambiando ad ogni passo i valori di una o più variabili aleatorie in modo da formare per l'appunto una catena di Markov. In questo modo si evita di ricalcolare da zero ogni campione e si sfrutta il campione precedente per generare il successivo, ottimizzando il processo di campionamento.

Markov Blanket

Il Markov Blanket di una variabile aleatoria X in una bayesian network è l'insieme di variabili aleatorie che rendono X condizionatamente indipendente da tutte le altre variabili aleatorie della rete. Il Markov Blanket di X è composto dalle variabili genitori di X, dalle variabili figlie di X e dalle variabili genitori delle variabili figlie di X.

$$MB(X) = \text{Parents}(X) \cup \text{Children}(X) \cup \text{Parents}(\text{Children}(X))$$

Gibbs sampling

Il Gibbs sampling è un algoritmo di Markov Chain Monte Carlo (MCMC) che utilizza il concetto di Markov Blanket per generare campioni casuali. In pratica, ad ogni passo, si sceglie una variabile aleatoria X tra tutte le variabili aleatorie della rete, escluse quelle coinvolte nell'evidenza e, e si campiona un nuovo valore per X sfruttando la probabilità condizionata dai valori correnti delle variabili nel suo Markov Blanket. In questo modo non è necessario ricalcolare da zero ogni campione ed inoltre, sfruttando il Markov Blanket, ci si limita a considerare solo piccolo un sottoinsieme di variabili aleatorie per generare il nuovo campione, ottimizzando ulteriormente il processo di campionamento.

Di seguito lo pseudocodice per il Gibbs sampling:

0: **Algoritmo:** GIBBS-SAMPLING(*bn*, *num-samples*, X, e)
Input: *bn*, *num-samples*, X ed e come in precedenza
Output: $\mathbf{P}(X \mid e)$: stima della probabilità a posteriori di X data l'evidenza e

1: $C \leftarrow$ vettore vuoto, conta la frequenza di ogni simbolo di X nei campioni generati
2: *sample* $\leftarrow$ sample con valori casuali ad eccezione di quelli specificati in e
3: **for** $i = 1$ to *num-samples* **do**
4: **choose** Z_i tra le variabili di *bn* escuse quelle in e, secondo una distribuzione $\rho(i)$
5: $sample[Z_i] \leftarrow$ campione di Z_i generato dalla probabilità $\mathbf{P}(Z_i \mid \text{MarkovBlanket}(Z_i))$
6: $C[j] \leftarrow C[j] + 1$ con j tale per cui vale $sample[X] = x_j$
7: **return** NORMALIZE(C) ▷ normalizza per avere un vettore di probabilità

5.6 Probabilistic reasoning over time - Hidden Markov models

Time and uncertanty

In alcuni problemi di intelligenza artificiale, è necessario rappresentare su ambienti stocastici e sequenziali, ovvero in cui è necessario rappresentare sia l'incertezza che il passaggio del tempo. Si considerano le seguenti assunzioni:

- **discrete time**: il tempo è rappresentato come una sequenza di istanti discreti $t = 0, 1, 2, \ldots$ equidistanti temporalmente
- $\mathbf{X}_t$ e $\mathbf{E}_t$: variabili aleatorie che rappresentano rispettivamente lo stato del mondo e le evidenze osservate all'istante di tempo t
- $\mathbf{E}_t = \mathbf{e}_t$: evidenza osservata all'istante di tempo t (valore specifico di $\mathbf{E}_t$)

Transition model

Per rappresentare la dinamica di un ambiente stocastico e sequenziale si utilizza un transition model, ovvero un modello che prevede che lo stato del mondo all'istante di tempo t dipenda in generale da tutti gli stati del mondo precedenti.

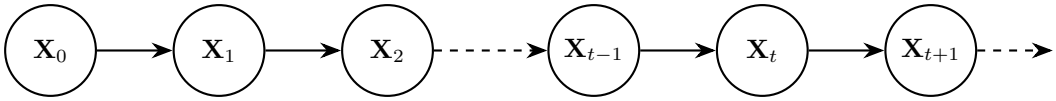
$$\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1}, \mathbf{X}_{t-2}, \ldots, \mathbf{X}_0) \equiv \mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{0:t-1})$$

Markov assumption e Markov process

La Markov assumption è un'assunzione che semplifica il transition model. Prevede che lo stato del mondo all'istante di tempo t dipenda solo da un certo numero costante di stati precedenti chiamato ordine. In questo modo si riduce notevolmente la complessità del transition model.

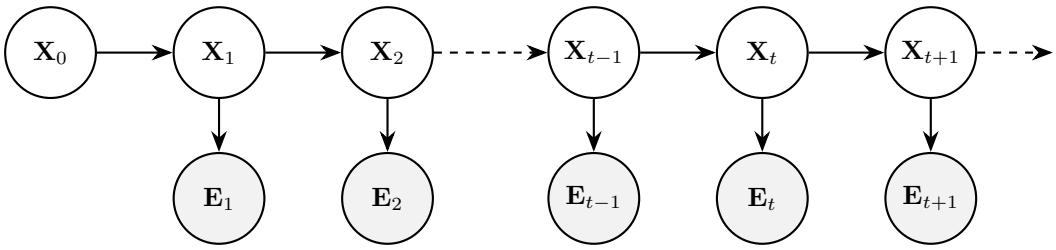
first order Markov assumption :	$\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$
second order Markov assumption :	$\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1}, \mathbf{X}_{t-2})$
...	

Una dinamica di evoluzione di un sistema che rispetta la Markov assumption viene chiamata Markov process. Se di ordine superiore al primo è utilie specificarne anche l'ordine (es. second order Markov process). La rappresentazione grafica di una Markov process di primo ordine è una rete bayesiana strutturata come catena in cui ogni nodo corrisponde ad uno stato del mondo in un istante di tempo specifico, influenzato dallo stato del mondo nell'istante di tempo precedente.



Hidden Markov models - HMMs

Gli hidden Markov models (HMMs) sono una particolare classe di Markov process in cui lo stato del mondo non è direttamente osservabile (hidden), ma può essere inferito attraverso delle evidenze osservabili che dipendono soltanto dallo stato del mondo corrente. Gli HMMs sono utilizzati per modellare ambienti non fully observable.



Sensor model degli HMMs

Oltre al transition model che descrive la dinamica di evoluzione del sistema, gli HMMs prevedono anche un sensor model (anche detto observation model) che descrive la relazione tra lo stato del mondo corrente e l'evidenza relativa osservata. Data la Markov assumption, si ha che l'evidenza osservata all'istante di tempo t è indipendente da tutte le altre variabili della rete, se condizionata allo stato del mondo allo stesso istante di tempo t.

$$\text{chain } \mathbf{X}_0 \to \mathbf{X}_1 \to \cdots \to \mathbf{X}_t \to \mathbf{E}_t \qquad \Rightarrow \qquad \mathbf{E}_t \perp\!\!\!\perp \{\mathbf{X}_{0:t-1}, \mathbf{E}_{0:t-1}\} \mid \mathbf{X}_t$$

$$\text{fork } \mathbf{E}_t \leftarrow \mathbf{X}_t \to \mathbf{X}_{t+1} \to \cdots \to \mathbf{X}_T \qquad \Rightarrow \qquad \mathbf{E}_t \perp\!\!\!\perp \{\mathbf{X}_{t+1:T}, \mathbf{E}_{t+1:T}\} \mid \mathbf{X}_t$$

$$\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_{0:t}, \mathbf{E}_{0:t-1}) = \mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$$

5.7 Inference in temporal models

Le azioni di inferenza che si possono compiere sugli HMMs sono le seguenti:

- **filtering** o **state estimation**: calcolare la probabilità a posteriori $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ dello stato $\mathbf{X}_t$ del sistema, conoscendo tutte le evidenze precedenti $\mathbf{e}_{1:t}$
- **prediction**: calcolare la probabilità a priori $\mathbf{P}(\mathbf{X}_{t+k} \mid \mathbf{e}_{1:t})$ dello stato futuro $\mathbf{X}_{t+k}$ del sistema, conoscendo tutte le evidenze precedenti $\mathbf{e}_{1:t}$
- **smoothing**: calcolare la probabilità a posteriori $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$ dello stato passato $\mathbf{X}_k$ del sistema, conoscendo tutte le evidenze fino all'istante attuale $\mathbf{e}_{1:t}$
- **most likely explanation**: calcolare la sequenza di stati più probabile $\mathbf{X}^*_{1:t}$ del sistema, conoscendo tutte le evidenze fino all'istante attuale $\mathbf{e}_{1:t}$

Di queste si analizzerà in dettaglio soltanto la prima, ovvero il filtering, che è l'azione di inferenza più comune e importante. Infatti è utilizzata dagli agenti intelligenti per mantenere una stima aggiornata dello stato del mondo in cui si trovano, quando il mondo non è fully observable.

Filtering in generale

L'operazione di filtering serve a calcolare la probabilità a posteriori $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$:

$$\begin{aligned}
\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t-1}, \mathbf{e}_t) && \text{divisione delle evidenze} \\
&= \alpha \cdot \underbrace{\mathbf{P}(\mathbf{e}_t \mid \mathbf{X}_t)}_{\text{update}} \cdot \underbrace{\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t-1})}_{\text{prediction}} && \text{dal teorema di Bayes} \\
&= \alpha \cdot \underbrace{\mathbf{P}(\mathbf{e}_t \mid \mathbf{X}_t)}_{\text{sensor model}} \cdot \sum_{\mathbf{X}_{t-1}} \underbrace{\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})}_{\text{transition model}} \cdot \underbrace{\mathbf{P}(\mathbf{X}_{t-1} \mid \mathbf{e}_{1:t-1})}_{\text{recursion}} && \text{dalla total probability rule}
\end{aligned}$$

Filtering come prediction e update

Si nota che si può scomporre il calcolo in due fasi principali:

- **prediction**: calcolo della probabilità $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t-1})$ dello stato $\mathbf{X}_t$ conoscendo tutte le evidenze precedenti $\mathbf{e}_{1:t-1}$; si utilizza il transition model $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$ e la probabilità calcolata ricorsivamente al passo precedente $\mathbf{P}(\mathbf{X}_{t-1} \mid \mathbf{e}_{1:t-1})$
- **update**: si calcola la probabilità a posteriori $\mathbf{P}(\mathbf{e}_t \mid \mathbf{X}_t)$ dell'evidenza osservata all'istante di tempo t data lo stato a cui è associata, utilizzando il sensor model $\mathbf{P}(\mathbf{e}_t \mid \mathbf{X}_t)$

Filtering dal punto di vista ricorsivo

Si osserva per che il calcolo è ricorsivo, in quanto per calcolare $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ è richiesta $\mathbf{P}(\mathbf{X}_{t-1} \mid \mathbf{e}_{1:t-1})$. È possibile, quindi, riscriverlo come funzione ricorsiva *FORWARD* che riceve in input due parametri $\mathbf{e}_t$ e $\mathbf{f}_{t-1} = \mathbf{P}(\mathbf{X}_{t-1} \mid \mathbf{e}_{1:t-1})$ e restituisce in output $\mathbf{f}_t = \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$:

$$\mathbf{f}_t = \mathit{FORWARD}(\mathbf{f}_{t-1}, \mathbf{e}_t) = \mathit{FORWARD}(\mathit{FORWARD}(\mathbf{f}_{t-2}, \mathbf{e}_{t-1}), \mathbf{e}_t) = \dots$$

Matrix representation of transition model

È possibile rappresentare il transition model $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$ come una matrice di transizione $\mathbf{T}$ in cui ogni elemento T_{ij} rappresenta la probabilità di transizione dallo stato $\mathbf{x}_i$ allo stato $\mathbf{x}_j$:

$$T_{ij} = P(\mathbf{X}_t = \mathbf{x}_j \mid \mathbf{X}_{t-1} = \mathbf{x}_i) \qquad \mathbf{P}(\mathbf{X}_t) = \mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1}) \cdot \mathbf{P}(\mathbf{X}_{t-1}) = \mathbf{P}(\mathbf{X}_{t-1}) \cdot \mathbf{T} = \mathbf{T}^T \cdot \mathbf{P}(\mathbf{X}_{t-1})$$

$$\mathbf{T} = \begin{pmatrix} P(\mathbf{X}_t = \mathbf{x}_1 \mid \mathbf{X}_{t-1} = \mathbf{x}_1) & P(\mathbf{X}_t = \mathbf{x}_2 \mid \mathbf{X}_{t-1} = \mathbf{x}_1) & \dots & P(\mathbf{X}_t = \mathbf{x}_n \mid \mathbf{X}_{t-1} = \mathbf{x}_1) \\ P(\mathbf{X}_t = \mathbf{x}_1 \mid \mathbf{X}_{t-1} = \mathbf{x}_2) & P(\mathbf{X}_t = \mathbf{x}_2 \mid \mathbf{X}_{t-1} = \mathbf{x}_2) & \dots & P(\mathbf{X}_t = \mathbf{x}_n \mid \mathbf{X}_{t-1} = \mathbf{x}_2) \\ \vdots & \vdots & \ddots & \vdots \\ P(\mathbf{X}_t = \mathbf{x}_1 \mid \mathbf{X}_{t-1} = \mathbf{x}_n) & P(\mathbf{X}_t = \mathbf{x}_2 \mid \mathbf{X}_{t-1} = \mathbf{x}_n) & \dots & P(\mathbf{X}_t = \mathbf{x}_n \mid \mathbf{X}_{t-1} = \mathbf{x}_n) \end{pmatrix}$$

Matrix representation of sensor model

In maniera analoga, è possibile rappresentare il sensor model $\mathbf{P}(\mathbf{e}_t \mid \mathbf{X}_t)$ come matrice di osservazione $\mathbf{O}_t$ associata ad ogni possibile evidenza $\mathbf{e}_t$. Esistono, quindi, tante matrici di osservazione $\mathbf{O}_t$ quante sono le possibli evidenze $\mathbf{e}_t$ osservabili. Le matrici di osservazione sono matrici diagonali in cui ogni elemento $O_{t,jj}$ rappresenta la probabilità di osservare l'evidenza $\mathbf{e}_t$ dato lo stato $\mathbf{x}_j$:

$$O_{t,jj} = P(\mathbf{e}_t \mid \mathbf{X}_t = \mathbf{x}_j) \qquad \mathbf{P}(\mathbf{e}_t) = \mathbf{P}(\mathbf{e}_t \mid \mathbf{X}_t) \cdot \mathbf{P}(\mathbf{X}_t) = \mathbf{P}(\mathbf{X}_t) \cdot \mathbf{O}_t = \mathbf{O}_t \cdot \mathbf{P}(\mathbf{X}_t)$$

$$\mathbf{O}_t = \begin{pmatrix} P(\mathbf{e}_t \mid \mathbf{X}_t = \mathbf{x}_1) & 0 & \dots & 0 \\ 0 & P(\mathbf{e}_t \mid \mathbf{X}_t = \mathbf{x}_2) & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & P(\mathbf{e}_t \mid \mathbf{X}_t = \mathbf{x}_n) \end{pmatrix}$$

Filtering with matrix representations

Utilizzando le rappresentazioni matriciali del transition model e del sensor model, è possibile riscrivere l'operazione di filtering in modo più compatto e efficiente:

$$\begin{aligned} \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t}) &= \alpha \cdot \mathbf{P}(\mathbf{e}_t \mid \mathbf{X}_t) \cdot \sum_{\mathbf{X}_{t-1}} \mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1}) \cdot \mathbf{P}(\mathbf{X}_{t-1} \mid \mathbf{e}_{1:t-1}) \\ &= \alpha \cdot \mathbf{O}_t \cdot \mathbf{T}^T \cdot \mathbf{P}(\mathbf{X}_{t-1} \mid \mathbf{e}_{1:t-1}) \end{aligned} \quad \Big| \quad \mathbf{f}_t = \alpha_t \cdot \mathbf{O}_t \cdot \mathbf{T}^T \cdot \mathbf{f}_{t-1}$$

5.8 Learning bayesian networks

Introduzione

Il processo di apprendimento di una bayesian network (BN) da dati osservati consiste nel determinare i vari parametri e la struttura delle bayesian networks in modo da poter utilizzarle per effettuare inferenze e prendere decisioni. Il processo di apprendimento può essere suddiviso in due fasi principali:

- **structure learning**: consiste nell'apprendere la struttura della rete, ovvero le relazioni di dipendenza tra le variabili aleatorie rappresentate dai nodi della rete
- **parameter learning**: consiste nell'apprendere le probabilità delle CPTs per ogni variabile della rete, una volta che la struttura della rete è stata determinata

Parameter learning and maximum-likelihood estimation (MLE)

Data una struttura di rete già definita e un dataset di dati osservati da cui apprendere i parametri, per calcolare i valori delle CPTs è possibile utilizzare il maximum likelihood estimation (MLE) che associa ad ogni probabilità di un certo evento, la frequenza con cui quell'evento si verifica nei dati osservati. Il processo è molto simile a quello che avviene nel prior o rejection sampling.

Laplacian smoothing

Alcune volte può capitare che, a causa di un dataset limitato, alcune combinazioni di valori delle variabili aleatorie non vengano osservate, portando a stime di probabilità pari a zero per quelle combinazioni. Per evitare questo problema, si può utilizzare tecniche di smoothing come il Laplacian smoothing

Il Laplacian smoothing consiste nell'inizializzare tutte le distribuzioni di probabiiltà a probabilità uniformi. Ad esempio per una variabile aleatoria binaria X con k valori possibili si inizializza la CPT di X con probabilità uniformi $P(X = x_i) = 1/k$ per ogni valore $i \in [1, k]$. Successivamente, considerando un dataset con N samples in cui la combinazione di valori $X = x_i$ si verifica m_i volte, si aggiorna la CPT di X con la seguente formula:

$$P(X = x_i) = \frac{m_i + 1}{N + k}$$

In questo modo, anche se una combinazione di valori non è stata osservata nel dataset, avrà una probabilità associata molto piccola, ma comunque non nulla.

Incomplete data

Il dataset utilizzato per l'apprendimento dei parametri potrebbe essere incompleto, ovvero potrebbero mancare dati per due motivi principali:

- **missing data**: alcuni campioni potrebbero non avere osservazioni per alcune variabili aleatorie in quanto potrebbero non essere state raccolte oppure non si avevano le risorse per raccoglierle
- **hidden variables**: alcune variabili aleatorie potrebbero non avere nessun dato osservato in quanto non osservabili direttamente, ma si sa che esistono e influenzano le altre variabili della rete

Di seguito si analizzeranno le tecniche per gestire i missing data e le hidden variables.

Missing values and estimation techniques

Per gestire i missing values, è possibile utilizzare diverse strategie, tra cui:

- **deletion**: si eliminano i campioni che contengono valori mancanti, potrebbe portare a una perdita di informazione significativa se i campioni eliminati sono molti
- **unknown value**: si sostituiscono i valori mancanti con un valore speciale che rappresenta l'assenza di dati, risulta però difficile da interpretare e potrebbe portare a risultati distorti se i valori mancanti sono molti
- **substitution**: si sostituiscono i valori mancanti con il valore più frequente per quella variabile, potrebbe portare a una distorsione dei dati se i valori mancanti sono molti o se la variabile ha una distribuzione molto sbilanciata
- **estimation**: si utilizzano tecniche di stima per stimare i valori mancanti, è la tecnica più sofisticata, accurata ed utilizzata, ma richiede più tempo e risorse computazionali

Un esempio di tecnica di stima dei valori mancanti prevede ad esempio di:

- calcolare la distribuzione di probabilità a posteriori della variabile con valori mancanti in un determinato campione, basandosi sugli altri campioni completi del dataset
- utilizzare questa distribuzione per dedurre il valore più probabile, ovvero quello con probabilità a posteriori maggiore sapendo gli altri valori osservati per quel campione

Hidden variables and expectation-maximization algorithm (EM)

Per stimare i valori delle variabili nascoste, si utilizza l'algoritmo di expectation-maximization (EM). L'algoritmo EM che consiste in due fasi principali di expectation e di maximization che vengono progressivamente ripetute fino a quando non si raggiunge una convergenza, ovvero quando i parametri stimati non cambiano più in modo significativo. Di seguito tutte le fasi dell'algoritmo EM:

- **initialization of observable variables**: si inizializzano i parametri della rete con le stime ottenute dai dati osservati utilizzando ad esempio il MLE o il Laplacian smoothing
- **initialization of hidden variables**: si inizializzano i valori delle variabili nascoste nel dataset con valori casuali, e si inizializzano i parametri delle CPTs delle hidden variables usando di solito distribuzioni uniformi
- **expectation step**: si stimano i valori delle variabili nascoste per ogni sample del dataset, utilizzando i parametri attuali della rete calcolati al passo precedente
- **value update**: si aggiornano i valori delle variabili nascoste nel dataset con quelli appena stimati
- **maximization step**: si ricalcolano le probabilità delle CPTs utilizzando il dataset aggiornato al passo precedente
- **convergence check**: finché i parametri stimati non convergono, ovvero non cambiano più in modo significativo, si ripetono le fasi di expectation, value update e maximization

Structure learning

Il problema di structure learning è estremamente complesso in quanto il numero di possibili strutture di rete cresce superesponenzialmente con il numero di variabili aleatorie ed è richiesto un dataset molto grande per poter stimare correttamente la struttura della rete. In base alla struttura (tree, polytree, general DAG) esistono numerosi approcci diversi, di cui verrà approfondito solo quello per general DAG.

In generale i metodi per il structure learning sono stati classificati in due categorie principali:

- **global methods**: algoritmi search and score che esplorano lo spazio delle possibili strutture in cerca della struttura che massimizza una certa funzione di score
- **local methods**: algoritmi che analizzano l'indipendenza tra insiemi di variabili aleatorie per determinare se esiste una relazione di dipendenza tra di esse, l'algoritmo più utilizzato è il PC algorithm, affrontato nella sezione sulla causal discovery

Global methods

I global methods sono algoritmi search and score e richiedono una funzione di score per valutare la bontà di una struttura di rete e un algoritmo di ricerca euristica per esplorare lo spazio delle possibili strutture.

Le **funzioni di score** servono per valutare quanto una struttura di rete rappresenti bene i dati osservati. Le più utilizzate sono il Bayesian Information Criterion (BIC), il Maximum Likelihood (ML), il Bayesian score (BD) e il Minimum Description Length (MDL). Le funzioni di score devono bilanciare accuratezza e complessità del modello. Ad esempio, la funzione di score BIC è definita come segue, dove $P(D \mid m)$ è la likelihood del modello dato i dati osservati, $|m|$ è il numero di parametri del modello e $|D|$ è il numero di osservazioni nel dataset.

$$\text{BIC} = \log P(D \mid m) - |m| \cdot \log |D|/2$$

Le **heuristic search** servono per esplorare lo spazio delle possibili strutture in modo efficiente. La più utilizzata è la hill climbing search, che parte da una struttura iniziale ed effettuando delle modifiche alla struttura (aggiunta, rimozione o inversione di archi) cerca di migliorare la funzione di score scelta.

6 Causal inference

6.1 Causality and Simpson's paradox

Correlazione vs causalità

In generale correlazione non implica causalità. Quando si osserva una correlazione tra due variabili $X \to Y$, non è possibile concludere automaticamente che la prima variabile X è causa dell'altra Y. La correlazione infatti potrebbe essere dovuta a una terza variabile Z che causa sia X che Y, oppure potrebbe essere una casuale coincidenza statistica.

Alcuni esempi di correlazione senza causalità sono:

- "when the rooster crows, the sun rises": c'è correlazione tra il canto del gallo e l'alba, ma il canto del gallo non causa l'alba, è invece l'alba che causa il canto del gallo
- "drownings increase with ice cream consmption": c'è correlazione tra il numero di annegamenti e il consumo di gelato, ma non è il consumo di gelato a causare gli annegamenti, è invece l'estate che causa sia l'aumento del consumo di gelato che l'aumento degli annegamenti
- "countries with more chocolate consumption have more Nobel laureates": c'è correlazione statistica tra il consumo di cioccolato e il numero di premi Nobel, ma nessuna delle due variabili causa l'altra
- "cholesterol concentration is directly proportional to the physical activity of a person": c'è correlazione tra la concentrazione di colesterolo e l'attività fisica di una persona, ma nessuna delle due è causa dell'altra, è invece l'età che causa sia l'aumento della concentrazione di colesterolo che l'aumento dell'attività fisica svolta
- recommendation systems: un utente che acquista un computer potrebbe essere interessato anche ad un mouse, ma non è detto che un utente che acquista un mouse sia interessato anche a un computer

Simpson's paradox

Il Simpson's paradox è un fenomeno statistico in cui una tendenza che appare in diversi gruppi di dati scompare o si inverte quando i gruppi vengono combinati insieme. Questo paradosso può verificarsi quando c'è una variabile confondente anche detta **confounding variable** che influisce sui dati in modo diverso nei diversi gruppi. In altre parole la confounding variable è la causa comune di due o più variabili osservate, e se non viene tenuta in considerazione può portare a conclusioni errate sulla relazione causale tra le variabili osservate. Questo aspetto verrà affrontato più avanti.

Un esempio può essere il caso descritto sopra in cui la concentrazione di colesterolo e l'attività fisica di una persona sembrano essere in correlazione diretta. Analizzando i dati per fasce di età si scopre, invece, che in realtà c'è correlazione inversa tra le due variabili osservate, ovvero più una persona pratica attività fisica più bassa è la sua concentrazione di colesterolo. L'età è la variabile confondente, per calcolare effettivamente la relazione tra colesterolo e attività fisica è necessario eliminare l'effetto dell'età sull'attività fisica, ad esempio analizzando i dati per fasce di età.

Un altro esempio classico del Simpson's paradox è il caso di valutazione dell'efficacia di un farmaco ad esempio con i dati seguenti:

	si - farmaco	**no - farmaco**
Uomini	81/87 (93%)	234/270 (87%)
Donne	192/263 (73%)	55/80 (69%)
Totale	273/350 (78%)	289/350 (83%)

Il farmaco sembra essere più efficace sia per gli uomini che per le donne, ma quando si combinano i dati di uomini e donne, il farmaco sembra essere meno efficace. Tale paradosso si verifica perché la variabile confondente (in questo caso il genere) influisce sui dati in modo diverso nei due gruppi. Ovvero il genere influisce sia sulla probabilità di essere trattati con il farmaco che sulla probabilità di guarire. Per calcolare quanto è realmente efficace il farmaco è necessario eliminare l'effetto della variabile confondente (genere) sulla probabilità di assumere il farmaco.

6.2 Structural Causal Model

Probemi di interpretazione della full joint probability e passaggio ai causal model

Analizzando la full joint probability $P(X_1, X_2, \dots, X_n)$ di un set di variabili aleatorie $X_1, X_2, \dots, X_n$ si osserva che tale distribuzione può essere fattorizzata in più modi diversi, ognuno dei quali esprime una certa struttura di dipendenza tra le variabili aleatorie. Ciò si verifica perché le correlazioni statistiche sono bidirezionali. Ogni struttura di dipendenza può essere rappresentata attraverso una bayesian network diversa, per cui esistono più bayesian network che rappresentano in maniera altrettando valida la stessa full joint probability.

Quando si esprimono, invece, le dipendenze tra variabili in termini di relazioni causali unidirezionali, allora è possibile identificare una sola struttura di dipendenza tra le variabili, e quindi una sola bayesian network che rappresenta la full joint probability. La rete formata si chiama causal model e permette di spiegare come sono generati i dati osservati, e permette di effettuare inferenze causali, ovvero di prevedere l'effetto di un intervento su una variabile osservata.

Variabili endogene ed esogene, causa e causa diretta

- **variabile esogena**: variabile che non dipende da nessun'altra variabile del modello; le variabili esogene rappresentano variabili esterne non osservate; spesso sono omesse nei grafi se mutualmente indipendenti tra di loro
- **variabile endogena**: variabile che dipende da almeno un'altra variabile del modello secondo una certa funzione deterministica; le variabili endogene rappresentano variabili osservate influenzate da altre variabili del modello
- **funzioni di causalità** $f_X(A, B, \dots)$: descrive il rapporto causale tra una variabile endogena X e le variabili da cui dipende $A, B, \dots$; siccome è un rapporto causale, la funzione è unidirezionale e ha senso invertirla (si potrebbe fare matematicamente, ma non ha interpretazione a livello causale)
- **causa diretta**: una variabile X si dice causa diretta di una variabile Y se la sua funzione f_Y dipende anche dalla variabile X
- **causa** (in generale): una variabile X si dice causa di una variabile Y se X è causa diretta di Y oppure se è causa diretta di una delle cause dirette di Y

Structural Causal Model - SCM

Un Structural Causal Model (SCM) è un insieme di variabili esogene U e variabili endogene V con un insieme di funzioni deterministiche F che descrivono le relazioni causali tra le variabili. Il tutto può essere rappresentato graficamente attraverso un grafo. Ad esempio:

$$U = \{A, B\}, \qquad V = \{C\}, \qquad F = \{f_C\} \qquad \text{con } C = f_C(A, B) = 2A + 3B$$

d-separation criterion

Il d-separation criterion è un criterio utilizzato per determinare se due variabili X e Y sono indipendenti condizionatamente a un insieme di variabili Z in un grafo causale: $X \perp\!\!\!\perp Y \mid Z$

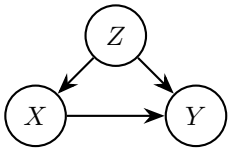
Due variabili X e Y sono d-separated condizionatamente ad un insieme di variabili Z se e solo se tutti i percorsi p tra le due variabili soddisfano almeno una delle seguenti condizioni:

- p contiene una chain $A \to B \to C$ o un fork $A \leftarrow B \to C$ tale che il nodo B è in Z
- p contiene un collider $A \to B \leftarrow C$ tale che il nodo B e i suoi discendenti non sono in Z

6.3 Interventions

Variabile confounder o causa comune

In alcuni casi si potrebbero avere due variabili X causa diretta di Y che sono a loro volta causate da Z come riportato nel grafo a destra. Z è causa comune sia di X che di Y ed è anche detta confounder.



Osservazione

Quando si vuole osservare come si distribuisce la variabile Y quando la variabile X assume un certo valore, si calcola la probabilità condizionata $P(Y \mid X)$ applicando le opportune regole di marginalizzazione, di fattorizzazione e di Bayes ($\mathbf{P}(X)$ si può portare dentro la sommatoria perché non dipende da Z):

$$\mathbf{P}(Y \mid X) = \frac{\sum_Z \mathbf{P}(Y, X, Z)}{\mathbf{P}(X)} = \frac{\sum_Z \mathbf{P}(Y \mid X, Z)\mathbf{P}(X \mid Z)\mathbf{P}(Z)}{\mathbf{P}(X)} = \sum_Z \mathbf{P}(Y \mid X, Z)\frac{\mathbf{P}(X \mid Z)\mathbf{P}(Z)}{\mathbf{P}(X)} = \dots$$

$$\mathbf{P}(Y \mid X) = \sum_Z \mathbf{P}(Y \mid X, Z)\mathbf{P}(Z \mid X)$$

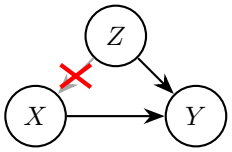
Dall'osservazione all'intervento

Tale probabilità condizionata indica le frequenze con cui Y assume ogni suo valore quando X assume un certo valore ed è basata sui dati utilizzati per costruire la rete. Ciò significa che se nel dataset, per una specifica istanza dell'ambiente, osservo il valore di X, allora posso stimare quale sarà la distribuzione di Y quando X assume quel valore.

Tuttavia, tale distribuzione non rappresenta specificatamente l'effetto causale di X su Y perché il valore di X è influenzato da Z. Ciò significa che quando si osserva (o condiziona) un certo valore di X, si sta indirettamente vincolando i valori che Z può assumere siccome Z ha causato X. Nel calcolo della probabilità condizionata, infatti, compare un fattore di peso $\mathbf{P}(Z \mid X)$ che rappresenta il vincolo che si impone a Z sapendo che X assume un certo valore.

Inolte, vincolando i valori di Z, l'effetto di Z su Y non è più generale, ma dipende dal valore osservato di X. Se ad esempio Z è il genere di una persona, X è la probabilità di essere trattati con un farmaco e Y è la probabilità di guarire, allora quando si osserva un certo valore di X (ad esempio essere trattati con il farmaco), si sta indirettamente alterando la distribuzione di Z siccome le donne sono più propense ad essere trattate con il farmaco rispetto agli uomini. Ciò significa che la probabilità di guarire Y in generale tiene conto maggiormente di come le donne rispondono al farmaco rispetto a come rispondono gli uomini, ovvero tiene conto sia dell'effetto diretto di X su Y che dell'effetto indiretto di Z su Y attraverso X.

Quando si vuole calcolare l'effetto causale di X su Y è necessario eliminare l'effetto indiretto di Z su Y attraverso X e per farlo è necessario intervenire sul grafo eliminando tutte le dipendenze che X ha con le altre variabili del modello. Ciò significa alterare il grafo come riportato a destra.



Intervento

Quando si vuole calcolare come si distribuisce Y quando si impone un certo valore a X (cioè intervenendo su X, eliminando tutte le dipendenze che X ha con le altre variabili del modello), si calcola la probabilità condizionata $\mathbf{P}(Y \mid do(X))$ applicando le opportune regole di marginalizzazione, di fattorizzazione e di Bayes, ottenendo la formula di back-door adjustment o adjustment formula:

$$\mathbf{P}(Y \mid X) = \frac{\sum_Z \mathbf{P}(Y, X, Z)}{\mathbf{P}(X)} = \frac{\sum_Z \mathbf{P}(Y \mid X, Z)\mathbf{P}(X)\mathbf{P}(Z)}{\mathbf{P}(X)} = \sum_Z \mathbf{P}(Y \mid X, Z)\frac{\mathbf{P}(X)\mathbf{P}(Z)}{\mathbf{P}(X)} = \dots$$

$$\mathbf{P}(Y \mid do(X)) = \sum_Z \mathbf{P}(Y \mid X, Z)\mathbf{P}(Z)$$

Per indicare che si sta intervenendo su X e non semplicemente osservando X, si utilizza la notazione do-operator $do(X)$. Si nota per l'appunto che il fattore di peso $\mathbf{P}(Z)$, ora non dipende più da X come invece accadeva prima, proprio perché è stata eliminata la dipendenza tra X e Z intervenendo su X.

Average Causal Effect - ACE

L'average Causal Effect (ACE) o causal effect difference è una misura di quanto il risultato di una variabile Y cambia in funzione di un intervento su una variabile X. In particolare, l'ACE è definito come la differenza tra la probabilità di Y quando si interviene su X imponendo un certo valore e la probabilità di Y quando si interviene su X.

$$\text{ACE} = P(Y = y \mid do(X)) - P(Y = y \mid do(\neg X))$$

Ad esempio applicando la formula di back-door adjustment e calcolando l'ACE per il caso del farmaco, si ottiene un valore di ACE postivo $\approx 5\%$ cioè significa che assumere il farmaco aumenta la probabilità di guarire del 5% rispetto a non assumere il farmaco.

$$\begin{aligned} P(Y = 1 \mid do(X = 1)) &= \sum_Z P(Y{=}1 \mid X{=}1,\, Z)P(Z) \\ &= P(Y{=}1 \mid X{=}1,\, Z{=}0)P(Z{=}0) + P(Y{=}1 \mid X{=}1,\, Z{=}1)P(Z{=}1) \\ &= 0.93 \cdot 0.51 + 0.73 \cdot 0.49 = 0.832 \end{aligned}$$

$$\begin{aligned} P(Y = 1 \mid do(X = 0)) &= \sum_Z P(Y{=}1 \mid X{=}0,\, Z)P(Z) \\ &= P(Y{=}1 \mid X{=}0,\, Z{=}0)P(Z{=}0) + P(Y{=}1 \mid X{=}0,\, Z{=}1)P(Z{=}1) \\ &= 0.87 \cdot 0.51 + 0.69 \cdot 0.49 = 0.7818 \end{aligned}$$

$$\text{ACE} = P(Y{=}1 \mid do(X{=}1)) - P(Y{=}1 \mid do(X{=}0)) = 0.832 - 0.7818 = 0.0502 \approx 5\%$$

6.4 Backdoor adjustment formula

Backdoor path

Un backdoor path è qualsiasi percorso tra due variabili X e Y che inizia con un arco entrante in X. Un backdoor path può essere naturalmente bloccato se contiene un collider oppure può essere bloccato condizionando sui nodi che costituiscono catene o fork.

Backdoor criterion

Un insieme di variabili S soddisfa il backdoor criterion se:

- blocca qualsiasi backdoor path tra X e Y
- non contiene nessun discendente di X

Eventualmente, $S = \varnothing$ se tutti i backdoor path sono naturalmente bloccati per opera di collider. Inoltre si nota che se c'è un collider $A \to B \leftarrow C$ che blocca un backdoor path tra X e Y, allora non si può condizionare solo su B perché A e C diventano dipendenti e il backdoor path non è più bloccato.

Adjustment formula with backdoor criterion

Se un insieme di variabili S soddisfa il backdoor criterion, allora è possibile calcolare la probabilità condizionata $\mathbf{P}(Y \mid do(X))$ con la adjustment formula generalizzata come segue:

$$\mathbf{P}(Y \mid do(X)) = \sum_S \mathbf{P}(Y \mid X,\, S)\mathbf{P}(S)$$

Model check with backdoor criterion

Nel caso in cui si abbia un modello causale in cui esistano più set S che soddisfano il backdoor criterion, allora è possibile calcolare più stime di $\mathbf{P}(Y \mid do(X))$ con la adjustment formula generalizzata. Se una delle stime differisce dalle altre, allora è possibile concludere che il modello causale è errato.

6.5 Unobserved confounders and front-door adjustment formula

Unobserved confounders

Alcune volte potrebbero esserci dei confounder non osservabili e di cui non si conosce nemmeno l'esistenza. In questi casi, non è possibile calcolare la probabilità condizionata $\mathbf{P}(Y \mid do(X))$ con la adjustment formula generalizzata perché non si conosce l'insieme di variabili S che soddisfa il backdoor criterion.

È possibile, però, che ci siano altre variabili osservabili Z comprese tra X e Y, per cui è possibile applicare il backdoor criterion ripetutamente tra X e Z e tra Z e Y. In questo modo è possibile calcolare le probabilità condizionate $\mathbf{P}(Z \mid do(X))$ e $\mathbf{P}(Y \mid do(Z))$ per poi calcolare la probabilità condizionata $\mathbf{P}(Y \mid do(X))$ combinando le due appena calcolate.

$$\mathbf{P}(Z \mid do(X)) = \mathbf{P}(Z \mid X) \text{ per collider } U \to Y \leftarrow Z \qquad \mathbf{P}(Y \mid do(Z)) = \sum_X \mathbf{P}(Y \mid Z, X)\mathbf{P}(X)$$

$$\begin{aligned}
\mathbf{P}(Y \mid do(X)) &= \sum_Z \mathbf{P}(Y \mid do(Z))\mathbf{P}(Z \mid do(X)) = \\
&= \sum_Z \left(\sum_{X'} \mathbf{P}(Y \mid Z, X')\mathbf{P}(X') \right) \mathbf{P}(Z \mid X) = \\
&= \sum_Z \mathbf{P}(Z \mid X) \sum_{X'} \mathbf{P}(Y \mid Z, X')\mathbf{P}(X') \qquad \text{front-door adjustment formula}
\end{aligned}$$

Front-door criterion

Un insieme di variabili S soddisfa il front-door criterion se:

- S blocca ogni percorso diretto da X a Y
- ogni backdoor path da X a S è bloccato
- ogni backdoor path da S a Y è bloccato condizionando su X

Front-door adjustment

Se un insieme di variabili S soddisfa il front-door criterion, allora è possibile calcolare la probabilità condizionata $\mathbf{P}(Y \mid do(X))$ con la front-door adjustment formula come segue:

$$\mathbf{P}(Y \mid do(X)) = \sum_S \mathbf{P}(S \mid X) \sum_{X'} \mathbf{P}(Y \mid S, X')\mathbf{P}(X')$$

6.6 Counterfactuals

Controlled direct effect - CDE

Il controlled direct effect (CDE) è una misura dell'effetto diretto di una variabile X su una variabile Y quando si modifica il valore di X (da x a x') e si impone un certo valore z a una variabile Z che media l'effetto di X su Y. Il risultato dipende sia dal valore z, sia dalla coppia di valori x, x'. Non è una proprietà della rete costante per qualsiasi valore, ma una misura per certi valori.

In particolare, il CDE si calcola come la differenza tra la probabilità di Y quando si interviene su X imponendo un certo valore x e la stessa probabilità quando si interviene su X imponendo un altro valore x', mantenendo fisso il valore z della variabile mediatrice Z.

$$\text{CDE}(x, x', z) = P(Y = y \mid do(X = x), do(Z = z)) - P(Y = y \mid do(X = x'), do(Z = z))$$

Imponendo un certo valore z alla variabile mediatrice Z, si elimina l'effetto indiretto di X su Y attraverso Z, e si misura solo l'effetto diretto di X su Y. Se ci sono più variabili mediatici, o in generale più percorsi non diretti da X a Y, allora è necessario imporre un certo valore a tutte le variabili mediatici o bloccare tali percorsi in altro modo per misurare solo l'effetto diretto di X su Y. Per questo scopo si formula il generalized controlled direct effect.

Generalized controlled direct effect

Il generalized controlled direct effect (GCDE) è una misura dell'effetto diretto di una variabile X su una variabile Y quando si modifica il valore di X (da x a x'), bloccando tutti i percorsi non diretti da X a Y attraverso altre variabili. È possibile calcolare il CDE per una rete generale se:

- $\exists\ \mathbf{S}_1$ che blocca ogni backdoor path da Z a Y
- $\exists\ \mathbf{S}_2$ che blocca ogni backdoor path da X a Y dopo aver rimosso ogni arco entrante in Z (per il punto precedente)

In questo caso, si può definire $\mathbf{S} = \{\mathbf{S}_1, \mathbf{S}_2\}$, allora il CDE si calcola come segue:

$$\text{CDE}(x, x') = \sum_{\mathbf{s} \in \mathbf{S}} P(Y = y \mid x, z, s) P(\mathbf{S} = \mathbf{s}) - \sum_{\mathbf{s} \in \mathbf{S}} P(Y = y \mid x', z, s) P(\mathbf{S} = \mathbf{s})$$

dopo aver definito $P(y \mid do(x), do(z)) = \sum_{\mathbf{s} \in \mathbf{S}} P(Y = y \mid x, z, s) P(\mathbf{S} = \mathbf{s})$

Definizione di counterfactuals

Un counterfactual è una proposizione ipotetica che descrive cosa sarebbe successo se una certa variabile X avesse assunto un certo valore x' in un dato contesto passato, quando invece X ha assunto un altro valore x. In pratica consiste nel chiedersi cosa sarebbe successo se si avesse fatto una scelta diversa in passato, conoscendo cosa è successo realmente.

Per indicare che Y sarebbe stata y (conseguenza) se X fosse stata x' (condizione), nel caso particolare in cui si è verifica l'evidenza e, si utilizza la notazione a pedice (potential outcome) come segue:

$$Y_{X=x'}(e) = y$$

I counterfactuals sono definiti a livello individuale di singolo evento. Indicano il risultato deterministico di un evento in determinate situazioni e non sono delle probabilità. Per essere calcolati, i counterfactuals richiedono la conoscenza di tutte le funzioni di causalità del modello, ovvero richiedono un SCM.

Calcolo deterministico del counterfactual

Il calcolo deterministico del counterfactual $Y_{X=x'}(e)$ si effettua in tre passi:

- conoscendo l'evidenza e, ovvero la situazione reale che si è verificata, si calcolano tutti i parametri della SCM compresi i parametri esogeni U omessi nei grafi
- si modifica la SCM intervenendo su X e imponendo il valore x' sulla variabile X (intervenire consiste nell'eliminare tutte le dipendenze che X ha con le altre variabili del modello)
- si calcola il valore di Y con la nuova SCM modificata mantenendo fissi tutti gli altri parametri calcolati al primo passo, escuso il valore imposto a X

Probabilità di un counterfactual - pre-intervention e post-intervention

Per passare da un counterfactual a un intervention, è necessario calcolare la probabilità del counterfactual:

$$\mathbf{P}(Y_x) \equiv \mathbf{P}(Y \mid do(X = x'))$$

Quando si vuole imporre anche l'evidenza $E = e$ che è stata osservata, serve fare maggiore attenzione:

$$\mathbf{P}(Y_x \mid E = e) \not\equiv \mathbf{P}(Y \mid do(X = x), E = e)$$

nel primo caso si sta effettuando prima l'intervento su X e poi si sta imponendo l'evidenza $E = e$, mentre nel secondo caso si sta prima imponendo $E = e$ e poi si sta effettuando l'intervento su X.

Calcolo probabilistico dei counterfactuals

Il calcolo probabilistico dei counterfactuals $P(Y_x \mid E = e)$ si effettua in tre passi:

- **abduction**: aggiornare la distribuzione di probabilità $\mathbf{P}(U)$ sapendo che è avvenuto un certo evento $E = e$, ovvero calcolare la distribuzione di probabilità $\mathbf{P}(U \mid E = e)$
- **action** o invtervention: modificare la SCM intervenendo su X (eliminandone le dipendenze) e imponendo il valore x, ovvero l'azione che si sarebbe voluto fare in passato
- **prediction**: usando il modello modificato, è possibile calcolare la probabilità del counterfactual $P(Y_x = y \mid E = e) = \sum_u P(U = u \mid E = e)$ per tutti i valori di u che soddisfano $Y_x(u) = y$, sfruttando le probabilità $\mathbf{P}(U \mid E = e)$ calcolate al primo passo

6.7 Linear SCM e Linear Gaussian models

Linear SCM - direct effects e total causal effects

Un linear SCM è un SCM in cui tutte le funzioni di causalità sono funzioni lineari. I coefficienti di causalità che compaiono nelle funzioni di causalità indicano gli effetti diretti tra variabili endogene.

In particolare:

- il **direct effect** di una variabile X su una variabile Y è rappresentato dal coefficiente di causalità associato alla variabile X nella funzione di causalità di Y.
- il **total causal effect** di una variabile X su una variabile Y è rappresentato dalla somma degli effetti di tutti i percorsi da X a Y calcolati moltiplicando i coefficienti di causalità che compaiono nelle funzioni di causalità lungo ogni percorso.

Linear Gaussian models

Un linear Gaussian model è un linear SCM in cui tutte le variabili esogene sono variabili aleatorie continue con distribuzione normale (gaussiana). Per un linear Gaussian model valgono le seguenti proprietà:

1. $X \perp\!\!\!\perp Y \mid Z \Rightarrow \begin{cases} P(Y \mid X, Z) = P(Y \mid Z) \\ E[Y \mid X, Z] = E[Y \mid Z] \end{cases}$

2. $\begin{aligned} \hat{Y} &= E[Y \mid X_1, X_2, \ldots, X_n] \\ &= r_0 + r_1 X_1 + r_2 X_2 + \cdots + r_n X_n \end{aligned}$

Dove i coefficienti $r_1, r_2, \ldots, r_n$ sono detti partial regression coefficients e rappresentano una retta/iperpiano che approssima al meglio la distribuzione dei dati, minimizzando la somma dei quadrati degli errori. Per calcolarli si ricorrono algoritmi iterativi come ad esempio l'algoritmo di gradient descent.

Si osserva che i coefficienti di regressione delle variabili endogene, "regrediti" sulle variabili genitori, rappresentano i coefficienti di causalità tra le variabili endogene. Inoltre il primo termine r_0 determina l'effetto delle variabili esogene sulla variabile endogena.

Calcolo dei causal effects con i linear Gaussian models

Dato un linear Gaussian model, è quindi possibile calcolare facilmente i direct effects e i total causal effects tra variabili endogene, semplicemente facendo regressione lineare dei dati del dataset tra le variabili endogene e i loro genitori.

6.8 Causal discovery and PC algorithm

Causal discovery and constraint-based methods

Il processo di causal discovery consiste nell'identificare la struttura causale che meglio rappresenta i dati osservati. Per farlo, si possono utilizzare diversi approcci, tra cui i constraint-based methods. Tali metodi si basano sulla misura delle dipendenze e indipendenze condizionate tra le variabili osservate per identificare la struttura causale del grafo. Alcuni esempi di constraint-based methods sono

- Peter Spirtes and Clark Glymour (PC) algorithm
- Fast Causal Inference (FCI) algorithm
- Greedy Equivalent Search (GES) algorithm

PC algorithm assumptions

Il PC algorithm ha quattro assunzioni fondamentali:

1. **acyclicity**: la struttura causale non contiene cicli
2. **causal sufficiency**: non ci sono confounder non osservati
3. **causal Markov condition**: variabili che soddisfano il d-separation criterion nel grafo, hanno distribuzioni indipendenti condizionate nel dataset
4. **faithfulness**: opposto alla causal Markov condition, se due variabili sono indipendenti condizionate su un insieme di altre variabili nel dataset allora sono d-separated nel grafo

Markov equivalent class and colliders

Si osserva che le chain $A \to B \to C$, $C \to B \to A$ e la fork $A \leftarrow B \to C$ implicano tutte la stessa indipendenza condizionata $A \perp\!\!\!\perp C \mid B$. Si dice che fanno parte della stessa Markov equivalence class (MEC), e risultano indistinguibili. Per questo motivo il PC algorithm utilizza i colliders per orientare gli archi del grafo in quanto $A \to B \leftarrow C$ implica $A \perp\!\!\!\perp C$, $A \not\perp\!\!\!\perp C \mid B$.

PC algorithm structure

Il PC algorithm è composto da quattro fasi:

1. si parte da un **fully connected undirected graph**, ovvero un grafo in cui ogni coppia di variabili è connessa da un arco non orientato
2. si identifica lo **skeleton del grafo**, ovvero si eliminano gli archi tra le variabili X e Y che risultano indipendenti condizionate $X \perp\!\!\!\perp Y \mid S$ su un certo insieme di variabili S; consiste in un classico test statistico come Fisher Z-test o BIC difference; in generale si parte con $S = \varnothing$ e si aumenta gradualmente la dimensione di S aggiungendo sempre più variabili fino a soddisfare la condizione di indipendenza, se tale condizione non si soddisfa mai, allora le due variabili sono dipendenti e si mantiene l'arco tra X e Y
3. si **orientano gli archi dei colliders**, ovvero le coppie di variabili X e Y che hanno $S = \varnothing$ in quanto formano un collider $X \to Z \leftarrow Y$ con altre variabili Z, le altre coppie di variabili X e Y che hanno $S \neq \varnothing$ fanno parte della stessa MEC e non possono essere orientate in quanto indistinguibili
4. si **orientano gli altri archi** propagando l'orientamento dei colliders, ad esempio ci sono partially directed path $X \to Z - Y$ e non c'è nessun collegamento tra X e Y, allora si orienta $Z \to Y$ per non creare un collider $X \to Z \leftarrow Y$ (che altrimenti sarebbe stato identificato al punto precedente)

6.9 Causal discovery for time series data

Time invariant data and time series data

Gli algoritmi di PC e FCI sono stati progettati per dataset time-invariant, ovvero datasetsenza correlazione temporale tra i samples. I dataset di questo tipo non hanno un ordine temporale, per cui è possibile scambiare i samples senza alterare la distribuzione dei dati.

Alcuni dataset, però, possono essere time-series o time-dependent, ovvero i samples hanno una dipendenza/correlazione temporale tra di loro. In questi casi gli algoritmi di PC e FCI non sono adatti in quanto non tengono conto di tale correlazione temporale.

PCMCI algorithm

Il PCMCI algorithm è un algoritmo di causal discovery progettato per time-series data. È basato sempre sulla causal discovery di un grafo causale, ma il grafo causale è ottenuto replicando le variabili del dataset per certo numero di time step indietro fino a $t - \tau$ (con τ massimo delay considerato). La rete è ottenuta con lo stesso principio dei grafi degli HMMs. In questo modo è possibile esprimere come le variabili di un certo time step influenzano le variabili dei time step successivi.

Si divide in due fasi:

- **PC algorithm**: si applica il PC algorithm per identificare soltanto lagged dependences, ovvero le dipendenze tra le variabili a diversi time step
- **Momentary Conditional Independence (MCI) test**: si applica un test di indipendenza condizionata per effettuare false positive rate optimization control

F-PCMCI algorithm

Il F-PCMCI algorithm è un algoritmo basato sul PCMCI algorithm, ma ottmizzato per la robotica. In particolare, prima di applicare il PCMCI algorithm, viene applicato un filtro (Transfer Entropy-based filter) per eliminare le variabili irrilevanti. In questo modo si riduce la complessità computazionale, migliorando sia i tempi di esecuzione, sia la qualità del grafo causale identificato.

CanDOIT algorithm

Il CanDOIT algorithm (Causal Discovery with Observational and Interventional Time-series data) è un algoritmo di causal discovery che si basa sull'idea di combinare dati osservazionali e dati interventional per effettuare causal discovery sul grafo ottenuto da un precedente algoritmo come F-PCMCI o L-PCMCI (più permmissivo dell'F-PCMCI). In questo modo, quando si hanno unobservable confounders che generano confusione all'L-PCMCI, si sfruttano le intervention per identificare correttamente il grafo causale, aggiungendo eventualmente variabili esogene (per rispettare la faithfullness).

ROS-Causal

Il ROS-Causal (Robot Operating System for Causal Discovery) è un framework software per la robotica che comprende una serie di sensori, attuatori e algoritmi di causal discovery come F-PCMCI e CAnDOIT. È possibile integrare ROS-Causal all'interno di robot che operano in ambienti dinamici, complessi e multiagente. In particolare, attraverso CAnDOIT, possono effettuare causal discovery in tempo reale, combinando dati osservazionali e dati interventional, per identificare la struttura causale dell'ambiente. In questo modo, possono adattare il loro comportamento in base alle osservazioni dell'ambiente, degli altri agenti e dei risultati delle loro azioni, migliorando la loro capacità di prendere decisioni informate e di interagire efficacemente con l'ambiente circostante.

7 Markov decision processes

7.1 Utility, policy and Markov models

Sequential decision problem

Un sequential decision problem è un problema in cui un agente deve prendere una serie di decisioni nel tempo su quale strada percorrere lungo una catena di Markov.

Utility

L'utilità è una funzione $U(s)$ che assegna un valore numerico a ogni possibile stato s del mondo. Rappresenta una misura di quanto è conveniente e desiderabile per l'agente essere in quello stato, immaginando che poi compierà sempre la scelta migliore possibile fino al raggiungimento dell'obiettivo. Di solito l'utilità viene normalizzata tra 0 e 1.

Expected utility

L'expected utility $EU(a)$ di un'azione a è la media pesata delle utilità degli stati raggiunti dopo aver compiuto l'azione pesata per l'utilità di ogni stato raggiunto. L'agente sceglie l'azione che massimizza l'expected utility.

$$EU(a) = \sum_{s'} P(\mathit{RESULT}(a) = s') \cdot U(s')$$

Policy

Una policy è una sequenza di azioni che un agente deve compiere per massimizzare la propria utilità. In un sequential decision problem viene anche detta Markov decision policy. Di solito si calcola semplificando il problema ricorsivamente in problemi più semplici, per poi riassemblare la soluzione finale.

Markov models

Un Markov model è un modello che soddisfa la proprietà di Markov, ovvero che uno stato s dipende soltanto dallo stato precedente. Esistono 4 classi di Markov model principali:

	passive (observation only)	**active** (possible actions)
fully observable	**MC** Markov Chain	**MDP** Markov Decision Process
partially observable	**HMM** Hidden Markov Model	**POMDP** Partially Observable Markov Decision Process

In base alla combinazione delle due caratteristiche si ottiene un modello diverso, con le seguenti proprietà:

- i modelli della prima colonna sono passivi, ovvero l'agente non può compiere azioni o in generale le azioni che compie non influenzano l'evoluzione dell'ambiente
- i modelli della seconda colonna sono attivi, ovvero l'agente può compiere azioni che influenzano l'evoluzione dell'ambiente
- i modelli della prima riga sono fully observable, ovvero l'agente è onniscente
- i modelli della seconda riga sono partially observable, ovvero l'agente deve fare inferenza per stimare lo stato del mondo

7.2 Markov Decision Processes - MDP

Struttura di un MDP

Un Markov Decision Process (MDP) è un modello che rappresenta un sequential decision problem in un ambiente fully observable e stocastico. Un MDP è definito da quattro elementi:

- un insieme di stati del mondo S con uno stato iniziale s_0;
- un insieme di azioni per ogni stato s, $ACTIONS(s)$
- un transition model $P(s' \mid s, a)$
- una reward function $R(s, a, s')$

Spesso si aggiunge anche un quinto elemento, il discount factor γ, che compare nella Bellman equation per calcolare l'utilità di ogni stato. Brevemente, indica quanto l'agente preferisce ricompense immediate rispetto a ricompense future. Verrà approfondito nei prossimi paragrafi.

Transition model

Il transition model $P(s' \mid s, a)$ è la probabilità di raggiungere lo stato s' dopo aver compiuto l'azione a dallo stato s. Rappresenta la dinamica dell'ambiente, ovvero come l'ambiente evolve in risposta alle azioni dell'agente. Siccome l'ambiente è stocastico, il risultato delle azioni non è deterministico. In generale viene usato per calcolare la probabilità di successo di un'azione usata per calcolare l'expected utility.

$$P(RESULT(a) = s') = \sum_s P(s' \mid s, a)P(s)$$

Reward function

La reward function $R(s, a, s')$ rappresenta la ricompensa immediata che l'agente riceve dopo aver compiuto l'azione a dallo stato s e aver raggiunto lo stato s'. Può essere sia positiva che negativa. Viene utilizzata per calcolare l'expected utility e guidare l'agente nella scelta delle azioni ottimali.

Le reward functions sono progettate in modo da guidare l'agente verso il raggiungimento dell'obiettivo desiderato in un piccolo numero di passi. Per fare ciò le ricompense sono generalmente negative in modo da penalizzare l'agente per ogni passo compiuto. Si osserva che più le reward sono negative, più l'agente è propenso a percorrere strade rischiose pur di arrivare all'obiettivo. Più le reward sono positive, più l'agente è propenso a percorrere strade più lunghe e sicure, ritardando il raggiungimento dell'obiettivo per massimizzare le ricompense immediate.

Nota: la reward è una ricompensa immediata che dipende localmente dallo stato corrente e dall'azione compiuta, la utility è globale e tiene conto anche delle scelte future.

Policy

Una policy è la soluzione a un MDP, ovvero una funzione che associa a ogni stato s la miglior azione a da compiere per massimizzare l'expected utility a lungo termine.

La qualità di una policy si misura in base alla sua expected utility della sequenze di stati raggiunti. La optimal policy è quella policy che massimizza l'expected utility.

L'optimal policy può essere determinata semplicemente scegliendo l'azione a che massimizza l'expected utility, ovvero che massimizza l'action-utility function $Q(s, a)$ (approfondita nei prossimi paragrafi):

$$\pi^*(s) = \arg\max_a Q(s, a)$$

7.3 Utilities over time

Evoluzione dell'utilità nel tempo

Esistono due modi per definire l'evoluzione temporare dell'ambiente:

- **finite horizon**: l'ambiente evolve per un numero finito di passi temporali dopo i quali non succede più nulla, la policy ottimale si dice non stazionaria in quanto può cambiare in base al numero di passi temporali rimanenti
- **infinite horizon**: l'ambiente evolve per un numero infinito di passi temporali, la policy ottimale si dice stazionaria in quanto è sempre la stessa, è il caso più semplice e che verrà trattato di seguito

Bellman equation

La Bellman equation è un'equazione che permette di calcolare l'utilità di ogni stato s in un MDP. Consiste in una media della somma della reward immediata $R(s, a, s')$ e dell'utilità dello stato raggiunto $U(s')$ pesata per la probabilità di raggiungere quello stato $P(s' \mid s, a)$, massimizzata per tutte le azioni possibili a da compiere dallo stato s. Scegliendo l'azione a ottima ed essendo ricorsiva, l'utilità di uno stato s tiene conto di tutte le ricompense future sapendo di compiere sempre la scelta migliore.

$$U(s) = \max_a \sum_{s'} P(s' \mid s, a) \cdot [R(s, a, s') + \gamma U(s')]$$

La somma da massimizzare viene anche detta action-utility function o Q-function:

$$U(s) = \max_a Q(s, a) \qquad \text{con } Q(s, a) = \sum_{s'} P(s' \mid s, a) \cdot [R(s, a, s') + \gamma U(s')]$$

L'utilità $U(s)$ dello stato s viene calcolata con processi iterativi come il value iteration algorithm o il policy iteration algorithm, approfonditi nei prossimi paragrafi.

Discount factor γ of the Bellman equation

Compare anche il discount factor $\gamma \in [0, 1]$ che rappresenta quanto l'agente preferisce ricompense immediate rispetto a ricompense future. Un buon trade-off si trova per $\gamma \in [0.9, 0.99]$.

- $\gamma \to 0$: l'agente è miope e preferisce ricompense immediate, anche se portano a vicoli ciechi o a situazioni pericolose nel medio termine, l'agente impiega quindi più tempo a raggiungere l'obiettivo
- $\gamma \to 1$: l'agente è lungimirante, preferisce ricompense future, anche al costo di sacrificare basse ricompense immediate, l'agente impiega quindi meno tempo a raggiungere l'obiettivo

Value iteration algorithm

Il value iteration algorithm è un algoritmo iterativo che calcola l'utilità di ogni stato s in un MDP basandosi sulla Bellman equation e su un processo iterativo di propagazione del calcolo. L'algoritmo inizia con tutte le utilità inizializzate ad un valore arbitrario (ad esempio 0) e ad ogni iterazione aggiorna le utilità di tuttigli stati attraverso la Bellman equation. L'algoritmo termina quando le utilità convergono, ovvero quando l'aggiornamento delle utilità è inferiore a una soglia di tolleranza ε.

0: **Algoritmo:** VALUE-ITERATION(mdp, ε)
Input: mdp: $\{S,\ A(s),\ P(s' \mid s, a),\ R(s, a, s'),\ \gamma\}$, ε: soglia di tolleranza
Output: vettore di utility function U per ogni stato del mondo

1: $U \leftarrow$ utility iniziale (ad esempio 0 per tutti gli stati)
2: **repeat**
3: $U' \leftarrow U$ ▷ salva l'utilità prima dell'aggiornamento per verificare la convergenza
4: **for all** $s \in S$ **do** ▷ per ogni stato del mondo
5: $U'[s] \leftarrow \max_a \sum_{s'} P(s' \mid s, a) \cdot [R(s, a, s') + \gamma U'[s']]$ ▷ Bellman equation update
6: **until** $\max_s |U[s] - U'[s]| < \varepsilon \cdot (1 - \gamma)/\gamma$ ▷ condizione di convergenza
7: **return** U

Policy iteration algorithm

Il policy iteration algorithm è un algoritmo iterativo che calcola l'optimal policy di un MDP basandosi sulla Bellman equation e su un processo iterativo come per il value iteration algorithm. Ogni iterazione dell'algoritmo consiste in due fasi:

- **policy evaluation**: calcola l'utilità di ogni stato considerando la policy corrente, basandosi sulla Bellman equation; siccome l'azione da compiere è già fissata, non si utilizza l'operatore di massimizzazione e le utilità di ogni stato possono essere calcolate analiticamente
- **policy improvement**: aggiorna la policy scegliendo per ogni stato l'azione che massimizza l'utilità calcolata nella fase di policy evaluation precedente

L'algoritmo termina quando i cambiamenti della policy non introducono più cambiamenti nell'utilità.

0: **Algoritmo:** POLICY-ITERATION(mdp, ε)
Input: mdp: $\{S,\ A(s),\ P(s' \mid s,a),\ R(s,a,s'),\ \gamma\}$, ε: soglia di tolleranza
Output: policy π: vettore con ogni azione da compiere per ogni stato del mondo

1: **repeat**
2: policy_stable $\leftarrow$ **true** ▷ inizializza controllo stabilità della policy
3: $U \leftarrow$ POLICY-EVALUATION(()mdp, π, U) ▷ fase 1 di policy evaluation
4: **for all** $s \in S$ **do** ▷ per ogni stato del mondo
5: $a^* = \arg\max_{a \in A(s)}$ Q-VALUE(()mdp, s, a, U) ▷ calcola l'azione ottima
6: **if** Q-VALUE(mdp, s, a^*, U) > Q-VALUE(mdp, s, $\pi[s]$, U) **then** ▷ se migliora la policy
7: $\pi[s] \leftarrow a^*$ ▷ aggiorna la policy
8: policy_stable $\leftarrow$ **false** ▷ la policy non si è ancora stabilizzata
9: **until** policy_stable ▷ condizione di convergenza
10: **return** U

Value iteration vs Policy iteration

Il policy iteration algorithm ha i seguenti vantaggi rispetto al value iteration algorithm:

- non utilizza l'operatore max per cui le equazioni sono lineari e più semplici da risolvere
- converge in un numero di iterazioni più basso rispetto al value iteration algorithm
- i risultati intermedi sono policy subottimali che possono essere utilizzate come soluzioni approssimate senza attendere la convergenza completa
- in generale è migliore per grandi spazi di stato

Viceversa, il value iteration algorithm ha i seguenti vantaggi rispetto al policy iteration algorithm:

- è più semplice da implementare
- le iterazioni da compiere sono più veloci (anche se ne servono di più)
- è migliore per piccoli spazi di stato

Esempio di applicazione di un MDP

Un esempio di applicazione di un MDP è il progetto di "Automatic hand-washing guidance for people with dementia". L'obiettivo è di guidare le persone con demenza (che fanno fatica a svolgere compiti sequenziali) a lavarsi le mani in modo corretto, attraverso un sistema di assistenza che riceve in input la situazione del processo di lavaggio delle mani da una fotocamera e fornisce in output dei suggerimenti vocali per guidare le persone a compiere i passi necessari per lavarsi le mani.

Lo spazio degli stati è rappresentato da tutte le possibili situazioni del processo di lavaggio delle mani e le azioni sono i suggerimenti vocali che il sistema può fornire per guidare le persone a compiere i passi necessari per lavarsi le mani e passare allo stato successivo.

7.4 Partially Observable Markov Decision Processes - POMDP

Struttura di di un POMDP

Un Partially Observable Markov Decision Process (POMDP) è un modello che rappresenta un sequential decision problem in un ambiente partially observable e stocastico. A differenza di un MDP, l'agente non è onnisciente e necessita di un sensor model per stimare lo stato in cui si trova. Un POMDP è definito dai quattro elementi di un MDP con l'aggiunta di un quinto elemento (il sensor model):

- un insieme di stati del mondo S con uno stato iniziale s_0;
- un insieme di azioni per ogni stato s, $ACTIONS(s)$
- un transition model $P(s' \mid s, a)$
- una reward function $R(s, a, s')$
- un **sensor model** $P(e \mid s)$

Belief state and belief estimation

Al posto di considerare uno stato s del mondo, in un POMDP si considera un belief state b che corrisponde ad alla distribuzione di probabilità di trovarsi in uno specifico stato, per ogni stato del mondo.

$$b(s_i) \equiv P(s_i)$$

Il belief state viene aggiornato ad ogni azione compiuta attraverso operazioni di filtering simili a quelle per gli HMMs. A differenza di questi, però, il transition model include anche l'azione compiuta. Inoltre il calcolo, seppur ricorsivo, è deterministico non probabilistico come negli HMMs siccome le variabili b, a, e sono tutte note.

$$b'(s') = \alpha \cdot P(e \mid s') \cdot \sum_s P(s' \mid s, a) b(s) \qquad b' = FORWARD(b, a, e)$$

Trasformare un POMDP in un Observable MDP

L'idea è di trasformare un POMDP in un Observable MDP per poter riutilizzare gli algoritmi e le tecniche già viste in precedenza per gli MDP. Per fare ciò, si ridefinisce sia il transition model che la reward function in funzione della distribuzione probabilistica dei belief state, al posto degli stati deterministici s e s'.

Transition model

La formula del transition model $P(b' \mid b, a)$ si ottiene marginalizzando per tutte le possibili osservazioni e, dove b è il belief state iniziale, b' è il belief state aggiornato dopo aver compiuto l'azione a e aver ricevuto l'osservazione e. Inoltre si espande $P(e \mid b, a)$ marginalizzando per tutti i possibili stati s e s'.

$$\begin{aligned}
P(b' \mid b, a) &= \sum_e P(b' \mid b, a, e) P(e \mid b, a) && \text{si marginalizza per } e \\
&= \sum_e P(b' \mid b, a, e) \sum_{s'} P(e \mid a, s', b) P(s' \mid a, b) && \text{si marginalizza per } s' \\
&= \sum_e P(b' \mid b, a, e) \sum_{s'} P(e \mid s') P(s' \mid a, b) && \text{dalla sensor Markov assumption} \\
&= \sum_e P(b' \mid b, a, e) \sum_{s'} P(e \mid s') \sum_s P(s' \mid s, a) b(s) && \text{dalla } 1^\circ \text{ order Markov assumption}
\end{aligned}$$

Si nota che, siccome il mondo è deterministico, se $P(b' \mid b, a, e) = 1$ allora vale $b' = FORWARD(b, a, e)$, altrimenti $P(b' \mid b, a, e) = 0$.

Reward function

La formula della reward function $\rho(b, a)$ si ottiene come la media della reward function $R(s, a, s')$ di un MDP moltiplicata per la probabilità di successo di un'azione a $P(s' \mid s, a)$, pesata per la probabilità di trovarsi nello stato s $b(s)$.

$$\rho(b, a) = \sum_s b(s) \sum_{s'} P(s' \mid s, a) R(s, a, s')$$

Optimal policy

Si osserva che una policy ottima per un MDP basato sui belief state è anche una policy ottima per il POMDP originale. Il problema è che ci potrebbero essere infiniti belief state per cui risulta notevolmente difficile calcolare la policy ottima. Per risolvere questo problema, si utilizzano algoritmi che portano a soluzioni approssimate.

Esempio di applicazione di un POMDP

Un esempio di applicazione di un POMDP è il progetto "ActiVis: Active Vision with Human in the Loop for People with Vision Impariments". L'obiettivo è di guidare le persone con disabilità visive a muoversi in modo sicuro e autonomo, attraverso un sistema di assistenza che riceve in input la situazione dell'ambiente da una fotocamera e fornisce in output dei suggerimenti vocali.

Si sfrutta un POMDP controller per suggerire all'umano quali azioni compiere per esplorare l'ambiente e classificare gli oggetti presenti, in modo da poter poi guidare l'umano a interagire con gli oggetti presenti. In particolare si parla di "Man on the Loop" POMDP controller, in quanto l'umano è parte integrante del processo decisionale e di esplorazione dell'ambiente e agisce insieme al sistema di assistenza per raggiungere l'obiettivo.

8 Machine learning

8.1 Types of machine learning and problem formulation

Types of machine learning

Sono stati teorizzati tre diversi approcci di machine learning:

- **supervised learning**: il dataset di addestramento è formato da coppie di input e output, e l'obiettivo dell'agente è di imparare le regole di mappatura tra input e output
- **unsupervised learning**: il dataset di addestramento è formato solo da input, e l'obiettivo dell'agente è di classificare l'input trovando strutture nascoste o pattern nei dati
- **reinforcement learning**: l'agente interagisce con un ambiente e riceve feedback sotto forma di ricompense o penalità, e l'obiettivo è di imparare una politica ottimale per massimizzare la ricompensa cumulativa

Problem formulation

In genere i problemi di machine learning vengono classificati in due categorie principali:

- **classification**: l'obiettivo è di classificare un certo input assegnandogli un'etichetta, i valori di output provengono da un set finito di classi
- **regression**: l'obiettivo è di trovare una funzione matematica che approssima al meglio una certa distribuzione dei dati, i valori di output sono continui

8.2 Supervised learning model

Ipotesi come best-fit-function

Dato un training set costituito da coppie (x_i, y_i), il processo di supervised learning consiste nel formulare una funzione ipotesi h, detta "model of the data", che deve approssimare al meglio la funzione f che ha generato i dati. Si cerca la best-fit-function affinché $h(x_i) \approx y_i = f(x_i)$, ma che allo stesso tempo generalizzi bene, ovvero che predice l'output corretto anche per i dati del test set.

La funzione ipotesi h è una funzione matematica parametrica, per cui può essere categorizzata in diverse classi come ad esempio lineare, sinusoidale, polinomiale di grado n, esponenziale, ...

Bias and underfitting

Il **bias** è la misura di quanto la funzione ipotesi h è in grado di rappresentare correttamente i dati dei training set. Un modello con un alto bias non si adatta bene ai dati di addestramento e ne sottostima la complessità (ad esempio un modello lineare per dati non lineari). Viceversa un modello con un basso bias rappresenta bene i dati di addestramento (quando la classe dell'ipotesi è appropriata ai dati).

Un modello è in **underfitting** quando ha un alto bias, ovvero quando non riesce a catturare la reale complessità dei dati di addestramento e di conseguenza avrà sempre basse performance sia sui training set che sui validation e test set.

Variance and overfitting

La **variance** è la misura di quanto la funzione ipotesi h è sensibile alle variazioni nei dati di addestramento. Un modello con un'alta variance si adatta troppo bene ai dati di addestramento (ad esempio un modello polinomiale di grado troppo elevato). Viceversa, uno con bassa variance riesce a generalizzare meglio e l'addestramento con dati diversi produce modelli simili.

Un modello è in **overfitting** quando si adatta troppo bene ai dati di addestramento, ma non riesce a generalizzare sui dati di test. Avrà alte performance nel training set, ma basse performance sul test set.

Bias-variance trade-off

Il bias-variance trade-off consiste nel scegliere opportunamente la classe e la complessità della funzione ipotesi h affinché abbia un bias sufficientemente basso per rappresentare bene e avere buone performance sui dati di addestramento, ma che allo stesso tempo abbia una variance sufficientemente bassa per generalizzare bene e avere buone performance anche sui dati del test set.

8.3 Model optimization and validation

Sviluppo di un supervised learning model

Lo sviluppo di un modello di supervised learning prevede i seguenti passaggi:

- **model selection**: scegliere la classe di funzioni ipotesi h da utilizzare per approssimare correttamente i dati di addestramento, ad esempio linear regression, polynomial regression, ...
- **model optimization**: trovare i parametri ottimali della funzione ipotesi h per approssimare al meglio i dati di addestramento, questo processo avviene iterativamente e ogni iterazione si divide in una fase di **training** dove si calcolano effettivamente i parametri ottimali, e una fase di **validation** dove si valutano le performance del modello durante ogni iterazione
- **model evaluation**: valutare le performance del modello sui dati di test, per verificare che il modello generalizzi bene e non sia overfitting

Si necessitano, quindi, di tre dataset distinti per sviluppare correttamente un modello di supervised learning:

- **training set**: utilizzato per addestrare il modello
- **validation set**: utilizzato per valutare e scegliere il modello migliore tra quelli addestrati
- **test set**: utilizzato per valutare le performance del modello

Tutti i dataset devono essere disgiunti tra loro, ovvero non devono contenere dati in comune, in modo che le valutazioni siano indipendenti e non influenzate da dati già visti durante l'addestramento. Quando si hanno pochi dati a disposizione, è possibile utilizzare tecniche come la cross-validation per massimizzare l'utilizzo dei dati disponibili.

Cross-validation

La cross-validation è una tecnica utilizzata ottimizzare l'utilizzo dei dati disponibili per la fase di addestramento e di validazione. L'idea è di suddividere il dataset in k parti (folds) di dimensioni approssimativamente uguali e di ripetere il processo di addestramento e validazione k volte, ogni volta ricombinando i dati in modo diverso. È sempre necessario riservare una parte di dataset per il test set per la valutazione finale del modello.

La principale tecnica di cross-validation è la **k-fold cross-validation**, in cui si utilizzano $k-1$ fold per addestrare il modello e il fold rimanente per validarlo, sempre diversi per ogni iterazione.

Se k è uguale al numero di campioni del dataset, si parla di **leave-one-out cross-validation**, in cui si utilizza un solo campione per validare il modello e tutti gli altri per addestrarlo, ripetendo il processo per ogni campione del dataset.

Model validation metrics

Le metriche di validazione a disposizione durante il training sono il training set error e il validation set error. In generale con l'addestramento il training set error diminuisce progressivamente perché il modello si adatta sempre meglio ai dati di addestramento. Il validation set error, invece, inizialmente diminuisce, ma dopo un certo punto inizia ad aumentare proprio quando il modello inizia ad andare in overfitting, perché si adatta troppo bene ai dati di addestramento e non riesce a generalizzare bene sui dati di validazione. Il punto in cui il validation set error inizia ad aumentare è il punto in cui si dovrebbe fermare l'addestramento per evitare l'overfitting.

Model evaluation

La model evaluation è la fase finale del processo di sviluppo di un modello di supervised learning, in cui si valutano le performance del modello sui dati di test. Serve per verificare che il modello sia in grado di generalizzare bene di fronte a dati nuovi e non visti durante l'addestramento.

In generale si avrà sempre una certa differenza tra le performance del modello sui dati di addestramento e sui dati di test. Se le performance sui dati di test, però, sono significativamente peggiori rispetto a quelle sui dati di addestramento, vuol dire che il modello è in overfitting.

8.4 Decision tree

Struttura di un decision tree

Un decision tree è un modello utile per risolvere problemi di classificazione, serve per mappare un vettore con i valori di una serie di attributi ad un singolo valore in output. Ha la stessa struttura di un albero binario di ricerca in cui ogni nodo interno rappresenta una condizione sul valore di un attributo e, in base alla risposta, si segue il branch (o ramo) dell'albero corrispondente fino a raggiungere una foglia che rappresenta l'output finale.

Learning a decision tree from data

Il processo di costruzione o addestramento di un decision tree è un processo ricorsivo in cui ogni iterazione si divide in due fasi principali:

- **feature selection**: si sceglie l'attributo più rilevante (o importante) come candidato ad essere la radice del futuro sottoalbero
- **data classification**: si divide il dataset in categorie in base ai valori dell'attributo scelto e si analizza la situazione di ogni categoria:
  - se la categoria non contiene campioni, si crea una foglia con il valore di output più comune della categoria padre
  - se la categoria contiene campioni con stesso output, si crea una foglia con quel valore di output
  - altrimenti, se la categoria contiene campioni con output diversi, si ripete il processo ricorsivamente scegliendo un nuovo attributo come radice del sottoalbero
  - infine se se la categoria contiene campioni con output diversi, ma si esauriscono gli attributi disponibili si crea una foglia con il valore di output più comune tra i campioni della categoria

Di seguito è riportato l'algoritmo in pseudocodice per costruire un decision tree:

0: **Algoritmo:** LEARN-DECISION-TREE(S_v, A, S)
Input: S_v: samples nella categoria da analizzare; A: attributi non ancora utilizzati; S: campioni della categoria padre
Output: decision tree T

1: **if** S_v è vuoto **then**
2: **return** crea foglia con il valore di output più comune tra S
3: **else if** tutti i campioni di S_v hanno lo stesso valore output **then**
4: **return** crea foglia con quel valore di output
5: **else if** A è vuoto **then**
6: **return** crea foglia con il valore di output più comune tra i campioni di S_v
7: **else**
8: $a \leftarrow \arg\max_{a \in A}$ IMPORTANCE(a, S_v) ▷ scegli l'attributo più rilevante
9: crea nodo interno N con l'attributo a ▷ crea nodo interno con l'attributo scelto
10: **for all** valore $v \in$ Values(a) **do**
11: $S_v \leftarrow \{c \in S \mid c[a] = v\}$ ▷ divide il dataset in base al valore v
12: $A' \leftarrow A - \{a\}$ ▷ aggiorna gli attributi disponibili
13: $T_v \leftarrow$ LEARN-DECISION-TREE(S_v, A', S) ▷ chiamata ricorsiva
14: collega il sottoalbero T_v al nodo N con un ramo etichettato con il valore v
return tree T con i nodi aggiunti

Importanza e rilevanza degli attributi - expected entropy e information gain

L'importanza o rilevanza di un attributo è una misura di quanto quell'attributo è utile per classificare i dati, ovvero quanti campioni del dataset riescono ad essere classificati in categorie pure (con campioni con lo stesso output) grazie a quell'attributo. Per misurare l'importanza di un attributo si utilizzano due concetti: l'expected entropy (o remainder) e l'information gain. L'attributo con l'information gain più alto è quello più rilevante, ovvero quello che viene scelto come radice del sottoalbero.

Expected entropy

Si definisce l'**expected entropy** (o remainder) di un attributo A come la media pesata delle entropie delle categorie S_v che si ottengono dividendo il dataset con i campioni ancora da classificare S in base ai valori v dell'attributo A. Per il calcolo si definiscono anche $P(c \in S_v)$ come la probabilità che un campione c appartenga alla categoria S_v associata al valore v e $\mathrm{B}(S_v)$ come l'entropia della categoria S_v.

$$\mathrm{Remainder}(A) = \sum_{v \in \mathrm{Values}(A)} P(c \in S_v) \cdot \mathrm{B}(S_v) \quad \text{con } P(c \in S_v) = \frac{|S_v|}{|S|}, \quad B(S_v) = -\sum_{x \in S_v} P(x) \log_2 P(x)$$

In un problema in cui l'output è binario (positive o negative), con p, n il numero di campioni positivi e negativi in S e p_v, n_v il numero di campioni positivi e negativi in S_v, si hanno le seguenti formule:

$$\mathrm{Remainder}(A) = \sum_{v \in \mathrm{Values}(A)} \underbrace{\frac{p_v + n_v}{p + n}}_{P(c \in S_v)} \cdot \underbrace{B\left(\frac{p_v}{p_v + n_v}\right)}_{B(S_v)}$$

Information gain

Si definisce l'**information gain** di un attributo A come la differenza tra l'entropia del dataset S con i campioni ancora da classificare e l'expected entropy di A:

$$\mathrm{Gain}(A) = \mathrm{B}(S) - \mathrm{Remainder}(A)$$

Decision tree performance

Come in un qualsiasi addestramento di un modello di supervised learning, analizzando le performance di un decision tree durante l'addestramento si può notare che il training set error e il test set error diminuiscono progressivamente all'aumentare della dimensione del decision tree fino a raggiungere un plateau, ovvero fino a saturazione. Progredendo oltre il plateau, il modello inizia ad andare in overfitting. Per impedire l'overfitting, è possibile utilizzare tecniche di pruning, il bagging o le random forests.

Pruning

Il pruning consiste nel rimuovere i nodi interni di un decision tree che non sono rilevanti per la classificazione. Per identificare i nodi da rimuovere, si utilizzano significance test come il chi-squared test, che confronta la deviazione standard Δ tra i campioni delle categorie di un attributo e un valore di riferimento Δ^* dato dalla chi-squared function χ^2. Se $\Delta < \Delta^*$, allora il nodo non è rilevante e viene rimosso, altrimenti viene mantenuto. Tale vincolo di irrilevanza di un nodo è chiamato **null hypothesis**.

In un problema in cui l'output è binario (positive o negative), con p, n il numero di campioni positivi e negativi in S, p_v, n_v il numero di campioni positivi e negativi in S_v e $\hat{p}_v$, $\hat{n}_v$ le stime dei campioni positivi e negativi in S_v, la total standard deviation Δ si calcola con la seguente formula:

$$\Delta = \sum_{v \in \mathrm{Values}(A)} \frac{(p_v - \hat{p}_v)^2}{\hat{p}_v} + \frac{(n_v - \hat{n}_v)^2}{\hat{n}_v}$$

Mentre il valore di riferimento Δ^* si ottiene dalla tabella della chi-squared function χ^2 con $d-1$ gradi di libertà, dove d è il numero di valori dell'attributo A, e con un livello di significatività α di solito scelto pari a 0.05 (5%).

Bagging

Il Bagging consiste nel suddividere il training set in k sottoinsiemi e costruire un decision tree per ogni sottoinsieme. Se si utilizzano i decision tree per classificazione, allora si prende il valore di output più comune tra i decision tree, se invece si utilizzano per regressione, allora si prende la media dei valori di output dei decision tree. In questo modo si riduce la variance del modello e si migliora la capacità di generalizzazione, evitando l'overfitting.

Il problema dei decision tree è il criterio di scelta degli attributi (rilevanza) è comune per tutti i K decision tree. È possibile, quindi, che più decision tree scelgano gli stessi attributi come radice dei sottoalberi e si riduca l'efficacia del bagging perché gli alberi sono fortemente correlati tra loro.

Random forests

Le random forests sono un'estensione del bagging in cui, oltre a suddividere il training set in k sottoinsiemi, si introduce anche una randomizzazione nella scelta degli attributi da utilizzare come radice dei sottoalberi. In questo modo si riduce la correlazione tra i decision tree e si migliora ulteriormente la capacità di generalizzazione del modello, evitando l'overfitting. Alcuni vantaggi delle random forests rispetto ai decision tree sono:

- effettuando la scelta dell'attributo con maggiore rilevanza tra un sottoinsieme casuale di attributi, il calcolo del candidato migliore è più efficiente
- non è necessario effettuare il pruning, perché la randomizzazione nella scelta degli attributi riduce già l'overfitting
- è possibile costruire i vari decision tree in parallelo su architetture parallelizzate, ottimizzando ulteriormente il processo di addestramento

Le random forests erano lo stato dell'arte per numerose applicazioni. Ora sono state superate da modelli più complessi come le reti neurali.

Decision tree applications

Un esempio di applicazione dei decision tree si trova nei chip "LSM6DS0X" di STMicroelectronics, detti anche MEMS chips, con il compito di riconoscere i gesti e le attività umane (ad esempio, camminare, correre, salire le scale, ...) attraverso i dati di accelerometri e giroscopi. Tali chip sono spesso installati negli smartphone e nei dispositivi indossabili.

8.5 Loss function

Loss function

Per misurare la performance di un modello di machine learning, si utilizza una loss function, ovvero una funzione matematica $L(y, \hat{y})$ con $\hat{y}$ la predizione del modello e y il valore reale del dataset, e restituisce un valore numerico che rappresenta la perdita di utility associata a quelle predizioni. Il learning agent, per massimizzare la sua utility, deve minimizzare la loss function. Esistono diverse loss function:

- **absolute value loss**: $L(y, \hat{y}) = |y - \hat{y}|$, per regressione
- **squared error loss**: $L(y, \hat{y}) = (y - \hat{y})^2$, per regressione (è il metodo dei minimi quadrati)
- **binary cross entropy o log loss**: basata sull'entropia*, per classificazione binaria
- **categorical cross entropy loss**: basata sull'entropia*, per classificazione multiclasse
- **hinge loss**: per classificazione binaria, utilizzata nelle support vector machines (SVM)

* le cross entropy loss verranno approfondite nella sezione dedicata alle reti neurali

Come notazione si utilizza:

- $Loss_j(\mathbf{w})$: loss function di un singolo campione j del dataset con $\mathbf{w}$ come parametri del modello
- $Loss(\mathbf{w}) = \sum_j Loss_j(\mathbf{w})$: loss function complessiva del dataset con $\mathbf{w}$ come parametri del modello

Gradient descent in generale

Il processo di gradient descent è un algoritmo iterativo simile all'hill-climbing, utilizzato per modificare i parametri del modello per minimizzare la loss function. L'idea è di aggiornare i parametri del modello per piccoli step nella direzione opposta al gradiente della loss function. La lunghezza dello step è controllata da un parametro chiamato learning rate α che può essere costante o decrescere con il tempo.

0: **Algoritmo:** GENERIC-GRADIENT-DESCENT($\mathbf{w}$, α)
Input: $\mathbf{w}$: parametri del modello; α: learning rate
Output: parametri ottimali w^*

1: **repeat**
2: **for all** parametro w_i in $\mathbf{w}$ **do** $w_i \leftarrow w_i - \alpha \dfrac{\partial Loss^*(\mathbf{w})}{\partial w_i}$ ▷ aggiornamento i parametri
3: **until** not converged

Varianti del gradient descent

In base a come viene calcolata la loss function $Loss^*(\mathbf{w})$, si distinguono diverse varianti:

- **batch gradient descent**: si utilizza l'intero dataset per calcolare il gradiente, la progressione risulta più stabile, ma è computazionalmente costosa per dataset di grandi dimensioni

$$Loss^*(\mathbf{w}) = \sum_{j \in \text{dataset}} Loss_j(\mathbf{w}) \quad \Rightarrow \quad w_i \leftarrow w_i - \alpha \frac{\partial \sum_{j \in \text{dataset}} Loss_j(\mathbf{w})}{\partial w_i}$$

- **stochastic gradient descent**: si utilizza un singolo campione j del dataset scelto a caso per calcolare il gradiente, la progressione risulta più rumorosa e non sempre converge correttamente, ma è computazionalmente più efficiente per dataset di grandi dimensioni

$$Loss^*(\mathbf{w}) = Loss_j(\mathbf{w}) \quad \Rightarrow \quad w_i \leftarrow w_i - \alpha \frac{\partial Loss_j(\mathbf{w})}{\partial w_i}$$

- **mini-batch gradient descent**: si utilizza un sottoinsieme del dataset per calcolare il gradiente, è un compromesso tra batch e stochastic gradient descent, con una progressione più stabile rispetto allo stochastic gradient descent e più efficiente rispetto al batch gradient descent

$$Loss^*(\mathbf{w}) = \sum_{j \in \text{partial dataset}} Loss_j(\mathbf{w}) \quad \Rightarrow \quad w_i \leftarrow w_i - \alpha \frac{\partial \sum_{j \in \text{partial dataset}} Loss_j(\mathbf{w})}{\partial w_i}$$

Nota: per la linearità della derivata, la sommatoria può essere portata dentro o fuori dalla derivata, in base a come si preferisce implementare il calcolo del gradiente. $\sum_c \frac{\partial f(x)}{\partial x} \Leftrightarrow \frac{\partial \sum_c f(x)}{\partial x}$

Multivariable linear regression

La regressione lineare su più variabili consiste nel trovare il vettore $\mathbf{w}$ che descrive l'iperpiano $(\mathbf{w} \cdot \mathbf{x}_j)$ che meglio rappresenta i dati di addestramento. L'iperpiano è la funzione ipotesi del modello.

$$h_{\mathbf{w}}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^T \mathbf{x}_j = \sum_i w_i x_{ji} \qquad \mathbf{w}^* = \arg\min_{\mathbf{w}} \sum_j L(y_j, h_{\mathbf{w}}(\mathbf{x}_j)) = \arg\min_{\mathbf{w}} \sum_j L(y_j, \mathbf{w} \cdot \mathbf{x}_j)$$

Multivariable classification

La classificazione su più variabili consiste nel trovare il vettore $\mathbf{w}$ che descrive l'iperpiano in grado di separare i dati di addestramento in classi distinte. L'iperpiano è chiamato decision boundary, o linear separator nel caso di classificazione lineare. La funzione ipotesi $h_{\mathbf{w}}(\mathbf{x}_j)$, per restituire un valore di output discreto, viene passata attraverso una funzione di attivazione detta threshold function.

$$h_{\mathbf{w}}(\mathbf{x}_j) = Threshold(\mathbf{w} \cdot \mathbf{x}_j) \qquad \text{con} \quad Threshold(z) = \begin{cases} 1 & \text{se } z \geq 0 \\ 0 & \text{altrimenti} \end{cases}$$

8.6 Support Vector Machines - SVM

Stuttura

Le Support Vector Machines (SVM) sono modelli di classificazione con le seguenti caratteristiche:

- cercano di trovare il maximum margin separator, ovvero il decision boundary con la distanza massima tra i campioni delle diverse classi
- sono lineari, ovvero il decision boundary è un linear hyperplane, ma con il kernel trick possono essere estese a classi di ipotesi non lineari
- non sono parametriche, in quanto il separating hyperplane è definito dai campioni di supporto del dataset e non da parametri del modello

Support vectors and decision boundary

I support vectors sono i campioni del dataset più vicini al decision boundary. Il decision boundary o maximum margin separator è definito come una linea che divide a metà l'area di confine tra le due classi, delimitata dai support vectors.

Ogni campione ha un parametro associato α_j che vale 0 se il campione non fa parte dei support vectors, e assume un valore positivo se il campione è un support vector. Il decision boundary si ottiene risolvendo il dual problem come segue:

$$\arg\max_{\alpha} \sum_j \alpha_j - 1/2 \sum_{j,k} \alpha_j \, \alpha_k \, y_i \, y_k \, (\mathbf{x}_j \cdot \mathbf{x}_k)$$

Una volta determinati i pesi α_j dei support vectors, la classificazione di un nuovo campione $\mathbf{x}$ si ottiene attraverso l'ipotesi con una 0-threshold function come la funzione *sign*:

$$h(\mathbf{x}) = \text{sign}\left(\sum_j \alpha_j y_j (\mathbf{x} \cdot \mathbf{x}_j) - b\right) \qquad \text{con} \quad \textit{sign}(z) = \begin{cases} 1 & \text{se } z \geq 0 \\ 0 & \text{altrimenti} \end{cases}$$

Kernel trick

Se i dati di addestramento non sono linearmente separabili, è possibile utilizzare il kernel trick, ovvero una rimappatura di un problema non lineare in uno spazio lineare di dimensione più elevata, in cui i dati possono essere linearmente separati. La rimappatura avviene attraverso una funzione di kernel $K(\mathbf{x}_j, \mathbf{x}_k)$ che si sostituisce al prodotto $\mathbf{x}_j \cdot \mathbf{x}_k$. Di seguito la formula di classificazione con il kernel trick ed alcuni esempi di kernel utilizzati comunemente:

$$h(\mathbf{x}) = \text{sign}\left(\sum_j \alpha_j y_j K(\mathbf{x}, \mathbf{x}_j) - b\right) \qquad \begin{array}{ll} \text{polynomial kernel :} & K(\mathbf{x}_j, \mathbf{x}_k) = (c + \mathbf{x}_j \cdot \mathbf{x}_k)^d \\ \text{gaussian kernel :} & K(\mathbf{x}_j, \mathbf{x}_k) = \exp(-\gamma \|\mathbf{x}_j - \mathbf{x}_k\|^2) \end{array}$$

9 Deep learning

9.1 Neuroni e funzioni di attivazione

Neurone artificiale

Un neurone artificiale è il singolo elemento di una rete neurale. È ispirato al neurone biologico e svolge le seguenti operazioni matematiche:

- riceve in ingresso un vettore di valori $\{a_1, a_2, a_3, \ldots, a_n\}$
- **input function**: effettua una combinazione lineare dei valori in ingresso, pesandoli attraverso un vettore i pesi $\{w_1, w_2, w_3, \ldots, w_n\}$ e aggiungendo un termine di bias $b = w_0$
- **activation function**: applica una funzione di attivazione al risultato della combinazione lineare, ottenendo un output y

$$y = g\left(w_0 + \sum_{i=1}^{n} w_i a_i\right) \qquad \text{con} \quad \begin{array}{ll} \text{input function:} & w_0 + \sum_{i=1}^{n} w_i a_i \\ \text{activation function:} & g(\cdot) \end{array}$$

Percettrone - neurone con threshold activation function

Un neurone artificiale che utilizza una threshold function come funzione di attivazione è detto percettrone. In origine si usavano solo percettroni, oggi si usano neuroni con funzioni di attivazione più complesse.

La threshold function è una funzione a scalino che, in funzione dei pesi assegnati agli input, permette di dividere i dati in due classi linearmente separabili.

$$y = Threshold(w_0 + w_1 x_1 + w_2 x_2) = \begin{cases} 1 & \text{per } w_0 + w_1 x_1 + w_2 x_2 \geq 0 \\ 0 & \text{altrimenti} \end{cases}$$

Ad esempio, rappresentando graficamente le possibili coppie di input di funzioni logiche elementari su un piano cartesiano a due variabili (x_1, x_2), si osserva che AND, OR e NOT sono linearmente separabili, per cui modellabili attraverso un percettrone, ma non la XOR, che è non linearmente separabile:

- AND: $w_1 = w_2 = 1$, $w_0 = -1.5$
- OR: $w_1 = w_2 = 1$, $w_0 = -0.5$
- NOT: $w_1 = -1$, $w_0 = 0.5$
- XOR: non linearmente separabile

Altre funzioni di attivazione

Oltre alla threshold function dei percettroni, esistono altre funzioni di attivazione più complesse, che permettono di modellare meglio valori continuti e facilitano l'addestramento delle reti neurali. Alcuni esempi di funzioni sono:

- sigmoid o logistic function: $g(x) = \sigma(x) = \dfrac{1}{1 + e^{-x}}$
- tanh (shifted and rotated sigmoid): $g(x) = \tanh(x) = \dfrac{e^{2x} - 1}{e^{2x} + 1}$
- ReLU (Rectified Linear Unit): $g(x) = \text{ReLU}(x) = \max(0, x)$
- softplus (rounded ReLU): $g(x) = \text{softplus}(x) = \ln(1 + e^x)$

L'addestramento di reti con percettori risulta complesso in quanto la funzione di attivazione threshold non è differenziabile. Per problemi complessi, conviene usare funzioni differenziabili come le sigmoidi.

È importante notare che le funzioni di attivazione non sono lineari, questo permette di risolvere anche problemi non lineari. Se si utilizzassero solo funzioni di attivazione lineari, si potrebbero risolvere solo problemi lineari, limitando enormemente ed inutilmente la potenza espressiva della rete neurale.

9.2 Multilayer networks and deep neural networks

Struttura di una rete multilayer

È possibile combinare più neuroni tra di loro in una rete multilayer. I neuroni sono organizzati in layer (verticalmente) in cui ogni neurone riceve input da tutti i neuroni del layer precedente e invia output a tutti i neuroni del layer successivo. Il primo layer è detto input layer e contiene i dati in ingresso (non ha neuroni propri), l'ultimo layer è detto output layer, mentre i layer intermedi sono detti hidden layer. Ogni neurone o percettrone della rete è detto unità.

Universal approximation theorem

Il teorema di approssimazione universale afferma che qualsiasi funzione continua può essere approssimata arbitrariamente bene da una rete neurale con un solo hidden layer, a condizione che il numero di neuroni in quel layer sia sufficientemente grande.

Ad esempio usando 2 neuroni nel hidden layer, è possibile approssimare una "ridge" (cresta) che risulta utile per modellare una XOR. Usando 4 neuroni, è possibile approssimare una "bump" (collina).

Rappresentazione grafica di una rete multilayer

Esistono due modi di rappresentare graficamente una rete neurale multilayer:

- **diagramma semplificato**: struttura a grafo con gli input e i singoli neuroni come nodi e con le connessioni tra i vari elementi come archi orientati a cui è associato il relativo peso
- **computational graph**: rappresentazione più dettagliata ed esplicita in cui ogni operazione (prodotto dell'ingressoper il peso, somme, funzioni di attivazione) è rappresentata come un nodo del grafo e i dati che fluiscono tra le operazioni sono rappresentati come archi orientati

Introduzione alle deep neural network

Una deep neural network è una rete neurale con più di un hidden layer. L'aggettivo "deep" si riferisce alla profondità della rete, ovvero al numero di hidden layer che contiene. Le deep neural networks sono in grado di apprendere rappresentazioni gerarchiche dei dati, in cui i layer più vicini all'input apprendono caratteristiche più semplici e hanno una vista locale, mentre i layer più vicini all'output apprendono caratteristiche più complesse e astratte e hanno una vista più globale.

Esistono due principali classi di deep neural networks:

- **feedforward neural networks**: i dati fluiscono in una sola direzione, dagli input agli output, senza cicli o retroazioni; un esempio sono le convolutional neural networks (CNN)
- **recurrent neural networks**: i dati, durante i processi intermedi e finali, possono essere reimmessi come input dello stesso layer o di layer precedenti, creando cicli e retroazioni; possiedono, quindi, uno stato interno ed una memoria; un esempio sono le long short-term memory (LSTM)

9.3 Convolutional Neural Networks - CNN

Struttura di una CNN

Una CNN è la prima e la più popolare classe di deep neural network. Fa parte delle feedforward neural networks e il nome deriva dalla presenza di hidden layer detti convolutional layers che svolgono operazioni di convoluzioni su parti dell'input. Tali layers seguiti da pooling layers che svolgono operazioni di downsampling, ed infine i dati sono processati da uno o più fully connected layers formati da neuroni come nelle reti neurali tradizionali.

Sono al momento lo stato dell'arte per l'elaborazione di immagini, ma possono essere applicate anche ad altri tipi di dati.

Receptive field e struttura gerarchica dei convolutional layers

I convolutional layers sono hidden layers che eseguono operazioni di convoluzione e sono organizzati gerarchicamente in una struttura che ricorda la corteccia visiva del cervello umano. I neuroni più vicini alla retina percepiscono segnali solo da una piccola area chiamata receptive field, elaborando caratteristiche semplici e locali come linee e bordi, senza una visione globale dell'oggetto. I neuroni del layer successivo ricevono input da più neuroni del layer precedente, ampliando la loro vista, ma mantenendo comunque una prospettiva relativamente locale. Con l'aumentare del numero di layer, i neuroni ricevono input sempre più ampi e complessi, acquisendo una visione sempre più globale, fino a elaborare caratteristiche complesse ed astratte come forme e oggetti.

Convolutional operation and kernel

L'operazione di convoluzione tra matrici è un processo iterativo che si basa sulla ripetizione di una **operazione fondamentale**. Tale operazione consiste nel sovrapporre una matrice più piccola chiamata kernel (o filtro) su una porzione dell'input in forma matriciale (ad esempio un'immagine) ed effettuare la combinazione lineare dei valori della parte dell'input sottesa dal kernel, ciascuno pesato per i valori corrispondenti del kernel.

L'operazione fondamentale viene ripetuta spostando il kernel su ogni possibile posizione dell'immagine, ottenendo così una nuova matrice chiamata **feature map** in cui ogni valore rappresenta l'output ottenuto da una posizione diversa del kernel. Il passo con cui si sposta il kernel sull'immagine è detto **stride** s.

Il risultato dell'operazione di convoluzione è una matrice di dimensioni inferiori in quanto il kernel non può essere sovrapposto nelle posizioni più esterne dell'immagine. Per evitare questo problema, si effettua un'operazione di **padding** che consiste nell'aggiungere un bordo di zeri attorno all'immagine, permettendo così al kernel di essere sovrapposto anche nelle posizioni più esterne.

Spesso si utilizzano più kernel diversi per processare lo stesso input, ottenendo così tante feature map quanti sono i kernel utilizzati. Ogni kernel è in grado di estrarre caratteristiche diverse dall'immagine.

Organizzazione dei neuroni in un convolutional layer

L'operazione fondamentale della convoluzione è implementabile attraverso un neurone artificiale in cui i pesi corrispondono ai valori del kernel. Tutti i neuroni di uno stesso convolutional layer condividono lo stesso kernel, ossia gli stessi pesi. Tale condivisione dei pesi viene definita **parameter sharing**.

Inoltre il **receptive field** (o input) di ogni neurone è una porzione dell'immagine (o dell'output del layer precedente) che ha stesse dimensioni del kernel. Impilando più convolutional layers si crea una struttura simile a quella della corteccia visiva spiegata in precedenza.

Pooling layers

Il pooling layer è un hidden layer che segue un convolutional layer e svolge operazioni di downsampling per ridurre le dimensioni dell'output del convolutional layer, mantenendo comunque le caratteristiche più importanti. Serve per semplificare l'input per il layer successivo. Esistono diversi tipi di pooling, tra cui:

- **max pooling**: prende il valore massimo all'interno di una finestra di dimensioni $p \times p$ e stride s
- **average pooling**: prende il valore medio all'interno di una finestra di dimensioni $p \times p$ e stride s

Tensor operations

Le operazioni di convoluzione tra matrici possono essere implementate in modo efficiente attraverso operazioni su tensori. Un tensore è una matrice multidimensionale utilizzata per rappresentare dati di input, pesi e output all'interno di una rete neurale. Le operazioni tra tensori sono eseguite in modo efficiente attraverso hardware dedicati altamente paralleli come TPU o GPU.

Di seguito un esempio di rappresentazione di alcuni tensori utilizzati in una CNN:

- un campione di input con 64 immagini RGB 256×256 è rappresentato da un tensore di dimensioni $64 \times 256 \times 256 \times 3$ (batch size, altezza, larghezza, canali RGB).
- 96 filtri composti da 3 kernel ciascuno di dimensioni 5×5 è rappresentato da un tensore di dimensioni $5 \times 5 \times 3 \times 96$ (altezza, larghezza, canali in ingresso, canali in uscita)
- l'output di un convolutional layer con stride $s = 2$ e padding $p = 2$ che utilizza i due tensori precedenti è rappresentato a sua volta da un tensore di dimensioni $64 \times 128 \times 128 \times 96$ (batch size, altezza, larghezza, canali in uscita).

Alcuni termini tecnici delle implementazioni delle CNN

- **batch**: insieme di campioni di dati in ingresso da un convolutional layer di dimensione costante, che vengono processati approssimativamente contemporaneamente per sfruttare al meglio le capacità di parallelizzazione dell'hardware; la dimensione del batch è detta **batch size** ed è costante durante l'addestramento
- **filtro**: insieme di più kernel che vengono applicati allo stesso input per estrarre una stessa feature; il numero di kernel in un filtro è detto **depth** del filtro ed è pari al numero di canali del tensore in ingresso del convolutional layer
- **feature map**: tensore di output dato dalla convoluzione tra il tensore di input e un solo filtro, ha solo un canale di uscita
- **output volume**: insieme di più feature map ottenute applicando più filtri allo stesso input; il numero di feature map in un volume è detto **depth** del volume ed è pari al numero di filtri utilizzati;

9.4 Input and output of a neural network

Input encoding

Le unità dell'input layer di una rete neurale sono direttamente collegate con i dati in ingresso. Serve dimensionare opportunamente l'input layer in modo che il numero di input units sia uguale alla lunghezza del vettore di input $\mathbf{x}$. In questo modo ogni unità dell'input layer riceve in ingresso un solo valore del vettore di input, che viene poi processato dai layer successivi. È possibile identificare tre principali tipi di input encoding che convertono i dati in ingresso in un formato numerico adatto per essere processato da una rete neurale:

- **boolean encoding**: false e true sono rappresentati rispettivamente da 0 e 1
- **numerical encoding**: i valori numerici sono rappresentati direttamente come input
- **categorical encoding**: i valori categorici sono rappresentati attraverso tecniche di encoding come **one-hot encoding**, in cui ogni bit del vettore di input è associato a una categoria e solo un bit è attivo (1) per indicare la categoria rappresentata, mentre gli altri sono inattivi (0)

Output layers

In base al tipo di problema richiesto, è necessario codificare opportunamente l'output layer di una rete neurale, in modo da renderlo interpretabile e sensato per il problema da risolvere. In base al tipo di problema si hanno diversi tipi di output layer:

- **regression**: i neuroni dell'output layer non possiedono activation function, l'output di ogni neurone si interpreta come una media pesata con varianza fissa dei valori in ingresso
- **single class classification**: l'output layer è formato da un solo neurone con una funzione di attivazione sigmoid che restituisce un valore compreso tra 0 e 1, interpretato come la probabilità che l'input appartenga alla classe positiva
- **multi-class classification**: l'output layer è formato da tanti neuroni quanti sono le classi, con una funzione di attivazione softmax che restituisce un vettore con valori comresi tra 0 e 1, interpretato come la pseudo-probabilità che l'input appartenga a ciascuna classe; la funzione non corrisponde a una vera distribuzione di probabilità, ma ne rispetta comunque gli assiomi:

$$\text{softmax}(\mathbf{z})_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

Loss function

La scelta opportuna della Loss function è importante per un corretto addestramento della rete neurale. Come già visto esistono diverse Loss function, che vanno scelte in base al tipo di problema da risolvere::

- **regression**: per problemi di regressione si utilizzano principalmente la absolute value Loss e la squared error Loss
- **single class classification**: per problemi di classificazione binaria si utilizza la binary cross entropy Loss o log Loss che penalizza le predizioni del modello che si discostano dai valori reali del dataset; a seguire la formula con $\hat{y}$ la predizione del modello e y il valore reale del dataset ottenuto come output di una sigmoid, ovvero la pseudo-probabilità che l'input appartenga alla classe positiva $x = 1$:

$$L(y, \hat{y}) = -\left[y \log(\hat{y}) + (1-y)\log(1-\hat{y})\right] = \begin{cases} -\log(\hat{y}) & \text{se } y = 1 \quad \rightarrow \text{ Loss bassa per } \hat{y} \approx 1 \\ -\log(1-\hat{y}) & \text{se } y = 0 \quad \rightarrow \text{ Loss bassa per } \hat{y} \approx 0 \end{cases}$$

- **multiclass classification**: per problemi di classificazione multiclasse, si utilizza la categorical cross entropy Loss che segue lo stesso ragionamento della binary cross entropy Loss, estesa a più valori di output, sapendo che y è un vettore one-hot e $\hat{y}$ è un vettore di pseudo-probabilità ottenuto da una softmax, si selezionerà solo il termine del vettore y associato alla classe reale c con $y_c = 1$:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) = -\log(\hat{y}_c) \approx -\log P(y = \hat{y}|x)$$

9.5 Learning neural networks

Learning neural networks

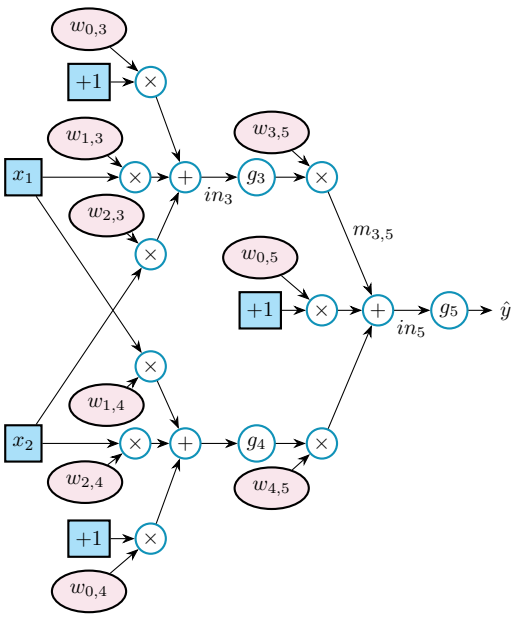
Il processo di addestramento di una rete neurale sfrutta l'algoritmo di (minibatch) stochastic gradient descent (SGD) illustrato in precedenza.

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla Loss(\mathbf{w}) \qquad \text{per ogni singolo peso } w_i: \quad w_i \leftarrow w_i - \alpha \frac{\partial Loss(\mathbf{w})}{\partial w_i}$$

Il problema che si incontra è il calcolo della derivata della Loss function rispetto ai pesi della rete neurale, in quanto risulta complesso scrivere analiticamente la funzione della Loss, per poter poi calcolarne la derivata parziale. Per risolvere questo problema, si utilizza l'algoritmo di back-propagation.

Back-propagation

L'algoritmo di back-propagation si utilizza per calcolare in modo efficiente la derivata della Loss function rispetto ad uno dei pesi della rete neurale. L'idea è di propagare all'indietro un certo messaggio (ovvero le derivate parziali della Loss, interpretabile anche come l'informazione dell'errore) dall'uscita della rete neurale fino al peso di interesse, basandosi sulla chain rule del calcolo differenziale.



Si vuole, ad esempio, calcolare la derivata della Loss function L rispetto al peso $w_{3,5}$:

$$\frac{\partial L}{\partial w_{3,5}} = \underbrace{\frac{\partial L}{\partial \hat{y}}}_{\text{passo 1}} \cdot \underbrace{\frac{\partial \hat{y}}{\partial g_5}}_{\text{passo 2}} \cdot \underbrace{\frac{\partial g_5}{\partial in_5}}_{\text{passo 3}} \cdot \underbrace{\frac{\partial in_5}{\partial m_{3,5}}}_{\text{passo 4}} \cdot \underbrace{\frac{\partial m_{3,5}}{\partial w_{3,5}}}_{\text{passo 5}}$$

1. derivata della Loss L rispetto all'output $\hat{y}$, si calcola analiticamente conoscendo la Loss function
2. derivata di $\hat{y}$ rispetto all'attivazione g_5, è sempre 1 in quanto $\hat{y} = g_5$
3. derivata di g_5 rispetto all'input in_5 del neurone 5, le derivate delle activation function sono tabulate
4. derivata di in_5 rispetto al moltiplicatore $m_{3,5}$, è sempre 1 in quanto $in_5 = m_{3,5} + m_{4,5} + m_{0,5}$
5. derivata di $m_{3,5}$ rispetto al peso $w_{3,5}$, corrisponde all'altro fattore della moltiplicazione

$$\frac{\partial L(y,\hat{y})}{\partial w_{3,5}} = \overbrace{\frac{\partial L(y,\hat{y})}{\partial \hat{y}}}^{\text{passo 1}} \cdot \overbrace{1}^{\text{passo 2}} \cdot \overbrace{g_5'(in_5)}^{\text{passo 3}} \cdot \overbrace{1}^{\text{passo 4}} \cdot \overbrace{g_3(in_3)}^{\text{passo 5}}$$

$$= \frac{\partial L(y,\hat{y})}{\partial \hat{y}} \cdot g_5'(in_5) \cdot g_3(in_3)$$

Se ci sono combinazioni di più nodi, si applicano anche le seguenti regole:

- se un nodo h ha più input provenienti da altri nodi f e g, i messaggi di errore che escono dal nodo h vengono moltiplicati per le derivate parziali di h rispetto a ciascuno dei suoi input f e g

$$\partial L/\partial f = \partial L/\partial h \cdot \partial h/\partial f_h \qquad \partial L/\partial g = \partial L/\partial h \cdot \partial h/\partial g_h \quad \rightarrow \quad \text{sono le regole usate sopra}$$

- se un nodo h ha più output diretti ad altri nodi j e k, i messaggi di errore multipli che entrano dai vari output $\partial L/\partial h_j$ e $\partial L/\partial h_k$ vengono sommati tra di loro

$$\partial L/\partial h = \partial L/\partial h_j + \partial L/\partial h_k$$

Pros and cons of back-propagation

- il costo della computazione è lineare nel numero di nodi del grafo computazionale, quindi è efficiente
- è possibile precalcolare le derivate delle activation function, rendendo il processo ancora più efficiente
- richiede la memorizzazione di tutti i risultati intermedi, rendendo il processo di addestramento molto dispendioso in termini di memoria
- può soffrire di vanishing gradient problem, ovvero quando $g'(\cdot) \ll 1$ (ad esempio con sigmoid o tanh) la propagazione degli errori ha molto poco effetto sull'aggiornanmento dei pesi

Deep networks and overfitting

A parità di parametri, una rete neurale con più hidden layer (deep network) è in grado di generalizzare meglio rispetto a una rete neurale con un solo hidden layer, ovvero tende ad andare meno in overfitting.

Adversarial attacks

Gli adversarial attacks sono attacchi che sfruttano la vulnerabilità delle reti neurali a piccoli cambiamenti impercettibili nei dati di input. Ad esempio l'aggiunta di un sottilissimo ed impercettibile (all'umano) layer di rumore ad una immagine può indurre la rete a fare predizioni errate, anche se i dati sembrano praticamente identici a quelli di input originali.

Dropout

Il dropout è una tecnica di regolarizzazione che consiste nel disattivare casualmente alcuni neuroni della rete durante l'addestramento, ad esempio dopo ogni minibatch. In questo modo si forza la rete a trovare soluzioni alternative e più robuste, evitando di creare corsie preferenziali che porterebbero a bassa generalizzazione e poco sfruttamento dell'intera rete.

9.6 Recurrent Neural Networks

Struttura di una RNN

Le RNN sono una classe di deep neural network in cui il computational graph contiene cicli, ognuno con un certo delay. Ciò significa che le unità di una RNN possono ricevere dati di input non solo dagli output layer precedenti, ma anche dal proprio stesso output o da quello di layer successivi proveniente da un precedente step temporale. In questo modo, le RNN sono in grado di mantenere uno stato interno e una memoria, che permette loro di processare sequenze di dati e di catturare dipendenze temporali.

Markov assumptions in RNNs

Le RNN possono soddisfare la Markov assumption se il loro stato interno $\mathbf{z}_t$ all'istante t dipende solo dallo stato interno $\mathbf{z}_{t-1}$ all'istante $t-1$ e dall'input $\mathbf{x}_t$ all'istante t.

$$\mathbf{z}_t = f_{\mathbf{w}}(\mathbf{z}_{t-1}, \mathbf{x}_t)$$

Backpropagation through time

L'output (o stato interno) del hidden layer $\mathbf{z}_t$ è determinato dall'input $\mathbf{x}_t$ dallo stato interno $\mathbf{z}_{t-1}$ e dai rispettivi pesi $\mathbf{W}_{z,z}$ e $\mathbf{W}_{x,z}$, passati attraverso la funzione di attivazione g_z. L'output $\hat{\mathbf{y}}_t$ si calcola con lo stato interno $\mathbf{z}_t$ ed i pesi $\mathbf{W}_{z,y}$ processati per la funzione di attivazione dell'uscita g_y. L'addestramento di una RNN deve aggiornare tutti i pesi del modello $\mathbf{W}_{x,z}, \mathbf{W}_{z,z}, \mathbf{W}_{z,y}$.

$$\mathbf{z}_t = g_z(\mathbf{W}_{z,z}\mathbf{z}_{t-1} + \mathbf{W}_{x,z}\mathbf{x}_t) \qquad \hat{\mathbf{y}}_t = g_y(\mathbf{W}_{z,y}\mathbf{z}_t)$$

Il processo di addestramento avviene attraverso l'algoritmo di **backpropagation through time**, che consiste nell'applicare l'algoritmo di back-propagation a una RNN "srotolata" su più step temporali (rappresentata come gli HMMs). Serve fare attenzione alla propagazione degli errori lungo la catena temporale: se $w_{z,z} < 1$ il gradiente tende a scomparire (vanishing gradient problem), mentre se $w_{z,z} > 1$ il gradiente tende ad esplodere (exploding gradient problem).

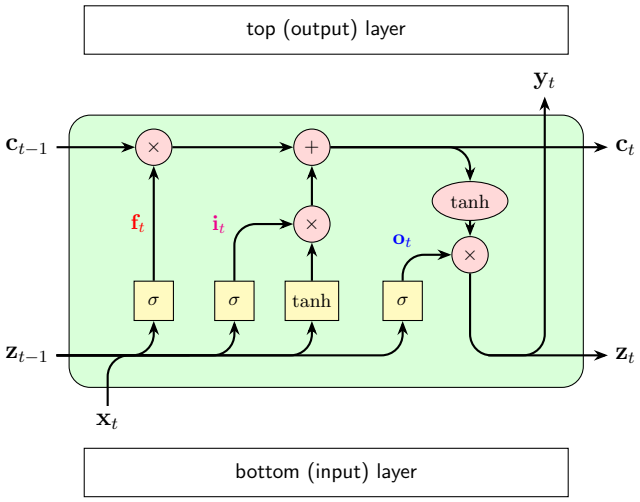
9.7 Long Short-Term Memory - LSTM

Struttura di una LSTM

Una LSTM è una particolare architettura di RNN che è stata progettata per risolvere il problema del vanishing gradient o dell'exploding gradient. Dispone sia di una working memory a breve termine, che di una long term memory a lungo termine. La long term memory è salvata in una memory cell c e, al posto di essere moltiplicata, viene copiata ad ogni iterazione, evitando così il problema del vanishing gradient o dell'exploding gradient.

Gates in a LSTM

La LSTM utilizza un'architettura a gate, ovvero un set di vettori di pesi che controllano il flusso di informazioni tra input stato interno e output. Di seguito lo schema di una LSTM, con i suoi gate:



- **forget gate $\mathbf{f}_t$**:
  determina quali termini della long term memory $\mathbf{c}_{t-1}$ devono essere dimenticati
- **input gate $\mathbf{i}_t$**:
  determina quali termini della long term memory $\mathbf{c}_{t-1}$ devono essere aggiornati additivamente
- **output gate $\mathbf{o}_t$**:
  determina quali termini della long term memory $\mathbf{c}_t$ devono essere usati per aggiornare lo stato interno $\mathbf{z}_t$ (anche detto short term memory) e l'output $\mathbf{y}_t$

Di seguito le formule che regolano il funzionamento di una LSTM:

$$
\begin{aligned}
\text{forget gate}: &\quad \mathbf{f}_t = \sigma(\mathbf{W}_{z,f}\mathbf{z}_{t-1} + \mathbf{W}_{x,f}\mathbf{x}_t) \\
\text{input gate}: &\quad \mathbf{i}_t = \sigma(\mathbf{W}_{z,i}\mathbf{z}_{t-1} + \mathbf{W}_{x,i}\mathbf{x}_t) \\
\text{output gate}: &\quad \mathbf{o}_t = \sigma(\mathbf{W}_{z,o}\mathbf{z}_{t-1} + \mathbf{W}_{x,o}\mathbf{x}_t) \\
\text{cell state}: &\quad \mathbf{c}_t = \underbrace{\mathbf{c}_{t-1} \odot \mathbf{f}_t}_{\text{forget}} + \underbrace{\mathbf{i}_t \odot \tanh(\mathbf{W}_{z,c}\mathbf{z}_{t-1} + \mathbf{W}_{x,c}\mathbf{x}_t)}_{\text{input}} \\
\text{hidden state}: &\quad \mathbf{z}_t = \underbrace{\mathbf{c}_t \odot \mathbf{o}_t}_{\text{output}}
\end{aligned}
$$

Applicazioni delle LSTM

Le LSTM sono utilizzate con successo in molti ambiti, tra cui:

- **natural language processing (NPL)**: modellazione del linguaggio, analisi del sentiment, traduzione automatica tra lingue diverse, ...
- **time series analysis**: previsione dei mercati azionari, previsione del meteo, ...
- **speech recognition**: trascrizione del linguaggio parlato in testo, ...
- **image and video captioning**: generazione di descrizioni testuali per immagini o video, ...
- **gesture recognition**: riconoscimento di gesti umani in video, ...
- **anomaly detection**: rilevamento di anomalie su dati sequenziali come frodi bancarie, intrusioni nella rete, ...
- **handwriting recognition**: riconoscimento di caratteri scritti a mano, ...

9.8 Unsupervised learning

Introduzione all'unsupervised learning

L'unsupervised learning è un paradigma di apprendimento automatico in cui il modello viene addestrato su dati non etichettati. Il dataset contiene solo input $\mathbf{x}$ senza output associati, e l'obiettivo del modello è quello di scoprire strutture nascoste, pattern o rappresentazioni significative nei dati per classificarli, raggrupparli o generare nuovi dati simili a quelli di input.

Il vantaggio dell'addestramento di modelli di unsupervised learning è che non richiede di effettuare labeling dei dati nel dataset che alcune volte risulta costoso o difficile, né di definire a priori ogni classe o categoria in casi ambigui (differenza tra bere e sorseggiare), poiché il modello può scoprire autonomamente le relazioni tra i dati.

Un modello di unsupervised learning è in grado di spiegare grandi dataset di input attraverso poche rappresentazioni in output. Il modello non fornisce un nome alle singole etichette o classi, ma ne conosce le proprietà di ognuna.

Obiettivi di un modello di unsupervised learning

Un modello di unsupervised learning viene utilizzato per diversi scopi, tra cui:

- **nuove rappresentazioni**: scoprire le key features dei dati di input che possono essere utilizzate per compiti di classificazione
- **generative modeling**: generare nuovi dati con le stesse caratteristiche o proprietà dei dati di input, ad esempio per generare nuove immagini, a partire da un dataset di immagini, ricombinando le caratteristiche apprese

Autoencoder - AE

Gli autoencoder sono una classe di modelli di unsupervised learning composti da una rete neurale divisa in due parti:

- **encoder**: insieme di layers collegati all'input, mappa i dati di input $\mathbf{x}$ in uno spazio latente di dimensione inferiore, detto bottleneck, ottenendo una rappresentazione compressa $\hat{\mathbf{z}} = f(\mathbf{x})$ contenente solo le features più rilevanti
- **bottleneck**: hidden layer centrale formato da poche unità, contiene la rappresentazione latente $\hat{\mathbf{z}}$ dei dati di input, con dimensione inferiore rispetto all'input
- **decoder**: insieme di layers collegati all'output, mappa la rappresentazione latente del bottleneck $\hat{\mathbf{z}}$ di nuovo nello spazio originale dei dati di input, cercando di ricostruire fedelmente i dati di input a partire dalla rappresentazione latente $\mathbf{x} \approx g(\hat{\mathbf{z}}) = g(f(\mathbf{x}))$

L'addestramento di un autoencoder consiste nel minimizzare la differenza tra i dati di input $\mathbf{x}$ e i dati di output $\hat{\mathbf{x}} = g(f(\mathbf{x}))$. Si nota che i dati di output corrispondono ai dati di input.

Linear autoencoders

Si definiscono i linear autoencoder come gli autoencoder in cui le due funzioni di compressione e decompressione f e g sono combinazioni lineari, (trascurando le funzioni di attivazione). Le formule risultano come segue:

$$\hat{\mathbf{z}} = f(\mathbf{x}) = \mathbf{W}\mathbf{x} \qquad \hat{\mathbf{x}} = g(\hat{\mathbf{z}}) = \mathbf{W}^T\hat{\mathbf{z}}$$

Separazione di encoder e decoder con applicazioni

Lo stadio di encoder e decoder possono essere separati e sfruttarli per compiti diversi e opposti.

Ad esempio, l'**encoder** può essere utilizzato per dimensionality reduction o feature extraction, ovvero per ridurre le dimensioni dei dati di input selezionando solo le features più rilevanti, per migliorarne la visualizzazione, per accelerarne i successivi processi di elaborazione o per compiti di classificazione.

Il **decoder**, invece, può essere utilizzato per ricostruire o generare nuovi dati simili a quelli di input, partendo da una rappresentazione latente con dati degradati o incompleti. Ad esempio, il decoder può essere utilizzato per image/audio denoising (riduzione del rumore ed aumento della qualità di immagini o audio degradati), image reconstruction (colorazioni di immagini, ...) o anomaly detection (analizzando le differenze tra i dati di input con imperfezioni e i dati di output ripuliti dalle imperfezioni).

Variational autoencoder - VAE

I variational autoencoder sono una particolari di autoencoder più complessi in cui la rappresentazione nel latent space è modellata come una distribuzione di probabilità $P(\mathbf{z} \mid \mathbf{x})$ (ad esempio gaussiana) e non più come un insieme discreto di punti $\mathbf{z}$ nello spazio latente come avveniva negli AE "classici".

L'aggiunta dell'aspetto probabilistico permette ai VAE di rappresentare maggiori informazioni con meno parametri, di essere più robusti a rumore e di **generare nuovi dati**. Infatti i VAE sono in grado di costruire nuovi dati campionando punti dallo spazio latente secondo la distribuzione appresa quando vengono perturbati da un rumore in ingresso o da dati incompleti. I "deterministic" autoencoder, invece, non sono in grado di generare nuovi dati, ma solo di ricostruire i dati di input partendo da una perfetta rappresentazione latente.

Generative adversarial networks - GAN

I generative adversarial networks sono una classe di modelli di unsupervised learning composti da due reti neurali in costante competizione tra di loro durante l'addestramento:

- **generator**: rete neurale che prende in input un vettore di rumore casuale e genera dati sintetici che assomigliano a quelli di input
- **discriminator**: rete neurale che prende in input sia dati reali (provenienti dal dataset) che dati sintetici costruiti dal generator, e cerca di distinguere tra i due, classificando correttamente i dati reali e quelli sintetici

Addestramento di una GAN

L'addestramento di una GAN consiste in un processo iterativo in cui il generator cerca di migliorare la qualità dei dati sintetici per ingannare il discriminator, mentre il discriminator cerca di migliorare la sua capacità di distinguere tra dati reali e sintetici. Durante il training possono verificarsi i seguenti probemi come:

- **mode collapse**: il generator converge ad una distribuzione limitata, specializzandosi nella generazione di pochi modi specifici, invece di generare una varietà di dati simili a quelli di input
- **mode hopping**: come il mode collapse, ma con il generator che alterna tra due o più modi specifici, invece di generare una varietà di dati simili a quelli di input

Per risolverli si può usare batch size maggiori, discriminatori più potenti, tecniche di regolarizzazione (ad esempio riducendo la complessità per migliorare la generalizzazione), ...

Applicazioni delle GAN

Esempi di applicazioni delle GAN includono la generazione di contenuti creativi e verosimili come immagini, musica o testo, l'upscaling di immagini a bassa risoluzione, la generazione di volti umani realistici (come in https://thispersondoesnotexist.com), ...

9.9 Transfer learning

Introduzione

Il trasfer learning è una tecnica di apprendimento automatico in cui un modello addestrato su un compito specifico viene riutilizzato come punto di partenza per addestrare un nuovo modello su un compito correlato, ma diverso. In pratica si cerca di trasferire la conoscenza acquisita risolvendo un compito ad un altro modello per affrontare un nuovo problema.

Esempi

Alcuni esempi di transfer learning includono:

- un pre-addestramento di un modello su ambienti simulati o sintetici, per poi trasferire la conoscenza acquisita su ambienti reali, utilizzando tecniche di domain adaptation per adattare il modello al nuovo dominio; tale processo risulta utile per contenere i costi di produzione se l'addestramento su ambienti reali è costoso o difficile
- un pre-addestramento di base language model per poi trasferire la conoscenza acquisita su compiti specifici per sviluppare modelli esperti in ambito medico o finanziario

Di solito si utilizzano modelli pre-addestrati popolari per compiti generali, come ad esempio "ResNet-50" per il riconoscimento di immagini.

Addestramento

L'addestramento di un modello con transfer learning si dividede in due fasi:

- **pre-addestramento**: addestramento del modello per risolvere un compito generale o di base, i primi layers di feature detection (sensibili all'ambiente in cui agisce) vengono temporaneamente disabilitati e ci si concentra sull'addestramento degli hidden layers di alto livello che caratterizzano il processo risolutivo del problema
- **fine-tuning**: addestramento fine e dettagliato del modello completo (con tutti i layers abilitati) sull'ambiente specifico in cui l'agente dovrà operare per adattare il modello al nuovo dominio e migliorare le sue prestazioni su compiti specifici

9.10 Language models

Introduzione ai language models

I language models sono modelli probabilistici che descrivono e rappresentano il linguaggio umano. Fino a prima dell'avvento delle reti neurali si usavano gli HMMs (basati sulla probabilità). Con la riscoperta delle reti neurali, si sono sviluppati modelli di language modeling sempre più complessi e potenti chiamati LLM o large language models che sfruttano per l'appunto deep neural networks molto grandi (large). Fanno parte, insieme ai LVM (large vision models) ed altri modelli, della categoria di "generative AI", ovvero modelli in grado di generare nuovi dati a partire da dati di input.

Word embedding and token classification

- **token**: unità di base del linguaggio, che può essere una parola o un simbolo (spazi e punteggiatura), vengono rappresentati come vettori numerici per essere processati dai modelli di language modeling
- **sentences**: sequenze di token che formano frasi o periodi, rappresentati come sequenze di vettori numerici
- **vocabulary**: insieme di tutti i token conosciuti dal modello, GPT-2 ad esempio ha un vocabolario di 50.000 token
- **word encoding**: processo di trasformazione delle parole e dei simboli del linguaggio naturale in numeri o vettori numerici per poter essere processati
- **word embedding**: classificazione o clustering dei token in spazi di dimensioni inferiori, basandosi sulle relazioni semantiche e sintattiche tra di essi

Token prediction and text generation

Il processo di token prediction consiste nel costruire una catena di token, partendo da un token di input e prevedendo successivamente il token successivo, fino a generare una sequenza di token che formano un testo coerente e significativo. L'algoritmo di token prediction può essere schematizzato come segue:

1. ricezione di un prompt o sentence di input (con annessa word encoding)
2. calcolo della distribuzione di probabilità per ogni token del vocabolario $\mathbf{P}(\text{next token} \mid \text{last tokens})$
3. scelta del token successivo secondo determinate strategie (greedy selection, random selection, ...)
4. concatenazione del nuovo token scelto alla sentence in ingresso
5. ripetizione del processo dal punto 2, fino a che non viene generato un "."

Sequence to sequence model with LSTM

Per implementare un language model si possono utilizzare le LSTM, in quanto, grazie alle loro capacità di memoria, permettono di comprendere il contesto dei vari token prima di generarne altri coerenti con il contesto. In particolare, si può utilizzare un sequence-to-sequence model basato su due reti LSTM (che "srotolate" formano una catena o sequenza di LSTM in cascata), in cui:

- la prima LSTM è detta **source**, riceve in ingresso i vari token progressivamente ad ogni istante di tempo e ne comprende il contesto e le relazioni tra di essi
- la seconda LSTM è detta **target**, riceve in input lo stato interno della prima LSTM e genera i token per costruire la sentence di output, coerente con il contesto compreso dalla prima LSTM

Il problema che si incontra è che le LSTM, a causa della loro struttura di memoria (le RNN rispettano la Markov assumption), rischiano di dare maggiore importanza ai token più vicini/recenti e perdono informazioni sui token più lontani, con conseguente progressiva perdita di contesto e coerenza nella generazione del testo. Per risolvere questo problema, si utilizzano le attention units.

Attention units e tecnica del self-attention

Le **attention units** sono componenti aggiuntivi della target LSTM che permettono alla target LSTM di attenzionare (catturare il contesto) di determinati layers della source LSTM. In questo modo, la target LSTM ha un contesto più ampio e completo, evitando di perdere le informazioni dei token più lontani. In particolare nei LLM sono richieste anche le **self-attention**, ovvero collegamenti tra i vari hidden layers della stessa LSTM. Per implementare le attention units e le self-attention, si utilizzano i transformers.

Transformers

I transformers sono una particolare struttura in grado di implementare le self-attentions. Sono l'elemento fondamentale degli LLM come GPT (Generative Pre-trained Transformer). Ogni transformer riceve in input un token e restituisce in output un vettore con la probabilità di ciascun token successivo.

Struttura a colonne e layers dei transformers

I transformers si organizzano in tante colonne verticali con un numero compreso tra 12 e 96 transformers ognuna, che vengono affiancate tra di loro in un numero prossimo alla dimensinone del vocabolario (ad esempio 128k). La struttura viene descritta attraverso due parametri:

- **context width**: numero di colonne affiancate tra di loro, indica la dimensione del contesto che il modello è in grado di catturare
- **context depth**: numero di transformers per colonna, indica quanto il modello è in grado di astrarre le informazioni del contesto, cioè il livello di astrazione del contesto del modello (i transformers con self-attention implementano concetti simili ai layers convoluzionali con perception-field)

La struttura a colonne è preceduta da un layer di input encoding ed è seguita da un layer di output decoding detto language modelling head.

Input encoding

L'input encoding è un layer che costruisce i vettori di input da passare al transformer block. Ogni elemento del vettore di input viene calcolato unendo la parola con il relativo indice di posizione all'interno della sentence.

È possibile che, se il modello utilizza un dataset di training con prompt piccoli, sia condizionato a dare maggiore importanza ai token iniziali, trascurando quelli più lontani. Per risolvere questo problema, si potrebbe usare un posizionamento relativo in base al contesto implementato anche dagli attentional layers dei transformers.

Residual stream all'interno dei transformers

La struttura interna di un transformer è definita con il nome di residual stream. In pratica i dati all'interno del transformer fluiscono dall'ingresso all'uscita in maniera diretta attraverso uno stream di dati chiamato residual stream. I blocchi del transformer lavorano su una copia dei dati dello stream e si limitano soltanto ad aggiungere informazioni additivamente nel residual stream, senza sovrascrivere quelle già esistenti. In reti molto profonde (senza residual stream) si rischierebbe di perdere informazioni importanti per via del vanishing gradient problem, mentre con questa tecnica le informazioni vengono preservate, arricchite e propagate fino all'ultimo transformer.

Un transformer è composto internamente da due layers principali, ciascuno preceduto da un layer di normalizzazione (per evitare problemi di vanishing o exploding gradient) e seguito dall'addizionamento dei dati nel residual stream. I due layers sono i seguenti:

1. **multi head attention layer**: implementa le self-attention, ricevendo in ingresso i dati normalizzati anche delle colonne vicine dello stesso layer, serve per espandere il contesto catturato man mano che si sale lungo la colonna di transformers
2. **feed forward layer**: fully connected 2-layer feed forward network con un hidden layer e un output layer per elaborare e processare i dati prima di passarli al transformer successivo

Language model head

Il language model head è un layer di output che riceve in ingresso i dati dell'ultimo transformer della colonna e restituisce un vettore con la probabilità di ciascun token successivo. È formato a sua volta da più fasi:

- **unembedding layer**: riconverte i dati dell'ultimo transformer al contrario di quanto fatto dall'input encoding utilizzando una matrice di pesi che è la trasposta di quella dell'input encoding, si ottiene così un vettore di logits lungo quanto il vocabolario del modello, che rappresenta in maniera grezza le probabilità di ciascun token successivo
- **softmax layer**: normalizza i logits in un vettore di probabilità (pseudo-probabilità), in modo da poter interpretare i valori come probabilità di ciascun token successivo

Training transformers

I transformers vengono addestrati con cross-entropy loss (o meglio negative log-likelihood loss) attraverso l'algoritmo di gradient descent e back propagation.

Come visto in precedenza per le cross-entropy loss, il valore della loss per il singolo elemento si calcola usando la probabilità che il modello attribuisce al token corretto (più alto è il valore della probabilità, più bassa è la loss). La loss totale del modello si calcola come la media della loss di tutti i token previsti. L'obiettivo dell'addestramento è di minimizzare la loss media.

9.11 Continual learning

Catastrofic forgetting problem

In un determinato contesto di deep learning, spesso capita di addestrare un modello per lavorare in un certo dominio di addestramento, per poi voler estendere la conoscenza rieseguendo un fine tuning su un nuovo dominio di test. Ad esempio si effettua un primo addestramento per ambienti indoor e diurni, per poi estendere le capacità del modello su ambienti outdoor e notturni. Quando si effettua il re-training del modello sul nuovo dominio, molto spesso la conoscenza acquisita durante il primo addestramento sul vecchio dominio viene dimenticata, in favore della nuova conoscenza. La loss rispetto al nuovo dominio diminuisce, ma la loss rispetto al vecchio dominio torna ad aumentare. Questo fenomeno è detto catastrofic forgetting problem, ed è un problema comune nella gestione di modelli di apprendimento che necessitano di continue evoluzioni.

Il catastrofic forgetting problem, però, può risultare vantaggioso perché permette di soddisfare il "Right to be forgotten" previsto dal GDPR (General Data Protection Regulation) dell'Unione Europea.

Continual learning paradigm

Il continual learning è un paradigma di apprendimento automatico che mira a risolvere il catastrofic forgetting problem permettendo ai modelli di continuare ad eseguire processi di apprendimento e acquisire nuove conoscenze senza dimenticare quelle già apprese o senza avere problemi di interferenza in cui il modello confonde le nuove informazioni con quelle vecchie. Bisogna trovare il plasticity-stability trade-off, ovvero un equilibrio tra la plasticità del modello (capacità di apprendere nuove conoscenze) e la stabilità del modello (capacità di preservare le conoscenze già acquisite).

Applicazioni del continual learning

Spesso si usa il continual learning quando non si hanno a disposizione subito grandi dataset di training per addestrare un modello con tutte le features desiderate, ma i dati arrivano progressivamente nel tempo. L'aumento di conoscenza del modello comprende:

- **domain incremental**: ovvero l'estensione della conoscenza su nuovi domini, ad esempio estendere le capacità di un modello addestrato su immagini reali, anche su disegni o cartoon-style
- **task incremental**: ovvero l'aggiunta di nuovi task, ad esempio estendere le capacità di classificazione del modello con nuove classi di oggetti che il modello prima non conosceva

Strategie di apprendimento

Esistono tre tipi di strategie di apprendimento (che possono essere anche combinate tra di loro) per implementare il continual learning:

- **architectural strategies**: consistono nel modificare l'architettura del modello originale per adattarla al nuovo dominio o task, ad esempio aggiungendo nuovi layers o aumentandone la dimensione, in modo da aumentare la capacità del modello di apprendere nuove conoscenze senza interferire con quelle già apprese, oppure aggiungendo piccoli modelli modulari in grado di risolvere task specifici
- **rehearsal strategies**: consistono nel mantenere un piccolo dataset di esempi rappresentativi del vecchio dominio o task e includerlo nel nuovo addestramento, oppure freezare alcuni layers del modello originale per preservare le conoscenze acquisite durante il primo addestramento, senza che vengano sovrascritte durante il re-training
- **regularization strategies**: consistono nell'aggiungere vincoli su addestramenti succesivi, ad esempio utilizzando la distillazione della conoscenza oppure penalizzando le modifiche ai parametri più importanti per performare correttamente nel vecchio dominio

Perché è utile avere algoritmi efficienti di continual learning

Dato il continuo evolversi del mondo reale, è importante che i modelli di apprendimento automatico siano in grado di adattarsi e aggiornare le proprie conoscenze in modo continuo ed in modo efficiente. Infatti la crescita del numero di parametri delle reti neurali sta superando la crescita della memoria all'interno dell'hardware dedicato. Inoltre ad oggi addestrare un modello da zero ha un'impronta ecologica estremamente elevata e poco sostenibile.