Artificial Intelligence

University of Milan
Master's Degree in Computer Science
A.Y. 2022/2023

Stefano Vallodoro

Indice

1 Neural networks

1.1 Background biologico

Il nostro cervello è in grado di analizzare in modo sofisticato l'ambiente circostante per agire in modo efficace, come il riconoscimento e la reazione ad un pericolo. Lo studio di questi processi coinvolge biologia, medicina e psicologia. Ciò ha portato alla creazione di modelli che simulano l'attività cerebrale, utilizzati dall'informatica per offrire strumenti di predizione, ottimizzazione e problem-solving in vari campi applicativi. Il successo di questi modelli dipende dalla capacità di elaborare grandi quantità di dati in parallelo.

Il sistema nervoso degli animali è composto dal cervello (che nelle forme di vita "inferiori" viene spesso chiamato solo "sistema nervoso centrale"), dai diversi sistemi sensoriali che raccolgono informazioni dalle diverse parti del corpo e dal sistema motorio che controlla i movimenti. La maggior parte dell'elaborazione delle informazioni avviene nel cervello/sistema nervoso centrale. Il cervello è composto da miliardi di cellule dette *neuroni*. Un neurone è una cellula che raccoglie e trasmette attività elettrica ed è costituito da:

- *soma*, corpo cellulare del neurone che contiene il nucleo
- *dendriti*, sono filamenti raggiunti dalle terminazioni di altri neuroni e che gli permettono di raccogliere informazioni grazie a processi biochimici originati dai così detti neuro-trasmettitori
- *assone*, un lungo filamento (da alcuni millimetri a un metro) che si estende dal corpo principale della cellula nervosa e trasmette segnali elettrici che attivano altri neuroni attraverso il rilascio di neuro-trasmettitori. E' il percorso fisso lungo il quale i neuroni comunicano tra di loro. Alla sua estremità, l'assone si ramifica e possiede delle estremità chiamate bottoni terminali. Ogni pulsante terminale tocca quasi una dendrite o il corpo cellulare di un altro neurone. (sinapsi)

Secondo stime comuni, ci sono circa 100 miliardi (10^{11}) neuroni nel cervello umano, di cui una parte abbastanza grande è attiva in parallelo. I neuroni elaborano le informazioni interagendo tra di loro.

Sinapsi Un luogo in cui un assone e una dendrite si toccano quasi. C'è un piccolo spazio tra 10 e 50nm di larghezza, una miliardesima di metro (10^{-9}mt).

E' una struttura specializzata che permette la comunicazione tra neuroni. Consente il trasferimento da un neurone all'altro o verso una cellula effettore (come un muscolo o una ghiandola). Il trasferimento dell'informazione avviene attraverso segnali chimici chiamati neuro-trasmettitori, che vengono rilasciati dalle terminazioni nervose del neurone pre-sinaptico e si legano ai recettori della membrana del neurone post-sinaptico, generando una risposta elettrica o chimica sul neurone destinatario.

Comunicazione tra neuroni La forma più comune di comunicazione tra neuroni è che un bottone terminale dell'assone rilascia determinate sostanze chimiche, chiamate neuro-trasmettitori, che agiscono sulla membrana della dendrite ricevente e ne modificano la polarizzazione[1] (il suo potenziale elettrico). Quando questo impulso nervoso raggiunge la fine dell'assone, provoca il rilascio di neuro-trasmettitori nei bottoni terminali, trasmettendo così il segnale alla cellula successiva, dove il processo si ripete.

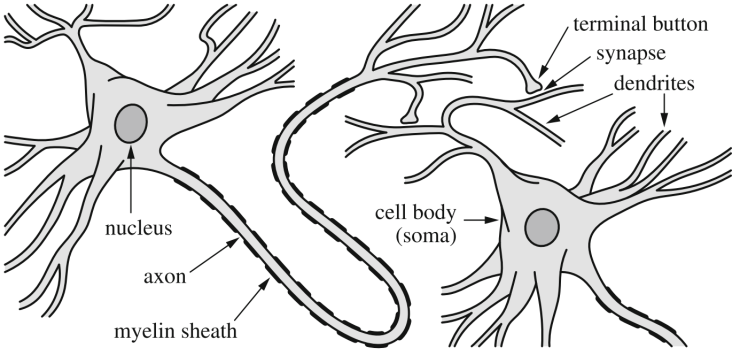


Figura 1: Struttura prototipica dei neuroni biologici

1.2 Threshold Logic Unit

La descrizione delle reti neurali biologiche rende naturale modellare i neuroni come unità logiche a soglia: se un neurone riceve un'input eccitante sufficiente che non viene compensato da un'input inibitorio altrettanto forte, diventa attivo e invia un segnale ad altri neuroni[2]. Un tale modello è stato già esaminato molto dettagliatamente da McCulloch e Pitts (1943). Di conseguenza, le Threshold Logic Unit sono anche conosciute come *neuroni di McCulloch-Pitts* o *perceptron*.

Per creare una rete neurale artificiale, è necessario trovare un analogo artificiale del neurone biologico. Questo ruolo è svolto dalle Threshold Logic Unit, o *TLU*. Una TLU è composta da n variabili di input, indicate con $x1, x2, ..., xn$, e un output y. Ad ogni variabile di input è associato un peso, indicato con w_i, che rappresenta la sua importanza ai fini del calcolo dell'output della TLU. Inoltre, viene assegnato un threshold θ, che rappresenta la soglia minima di attivazione necessaria per produrre un output positivo.

L'output della TLU viene calcolato mediante una funzione che somma i prodotti tra i pesi delle variabili di input e i loro valori, e confronta il risultato con il threshold θ. Se la somma dei prodotti supera la soglia, l'output della TLU sarà

[1] La polarizzazione si riferisce al livello di carica elettrica presente sulla membrana cellulare del neurone. La membrana cellulare è polarizzata quando vi è una differenza di potenziale elettrico tra l'interno e l'esterno della cellula

[2] Nel contesto della descrizione delle reti neurali biologiche, gli input eccitanti e inibitori rappresentano gli impulsi provenienti da altri neuroni che influenzano l'attività e il comportamento di un neurone specifico. La somma di questi impulsi eccitatori e inibitori determina se il neurone diventerà attivo e invierà un segnale agli altri neuroni.

positivo, altrimenti sarà negativo.

$$y = \begin{cases} 1 & \text{se } \sum w_i x_i \geq \theta \\ 0 & \text{altrimenti} \end{cases}$$

AND logico Con una TLU, possiamo simulare alcune funzioni booleane, come ad esempio l'AND logico tra due input $x1$ e $x2$. Ciò si ottiene assegnando pesi e threshold in modo da soddisfare un sistema di disequazioni, che definisce una regione nello spazio degli input in cui l'output sarà 1, e una regione dove sarà 0

1.2.1 Interpretazione geometrica

La formula per calcolare l'output di una TLU è simile all'equazione di un *iperpiano*, che rappresenta un piano in uno spazio di n dimensioni. Un'unità logica a soglia calcola la funzione

$$\sum w_i x_i + \theta = 0 \quad \textit{oppure} \quad \sum w_i x_i - \theta = 0$$

Spesso gli input vengono combinati in un vettore di input $x = (x1, \ldots, xn)$ e i pesi in un vettore di peso $w = (w1, \ldots, wn)$. Con l'aiuto del prodotto scalare, la condizione testata da una TLU può essere scritta anche come $wx \geq \theta$. E' un interpretazione del vettore normale[3]

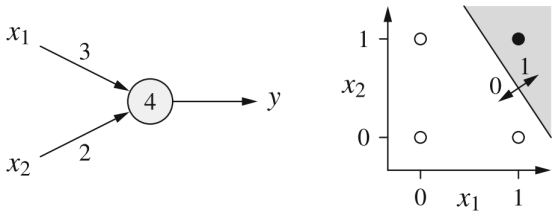


Figura 2: Geometria dell'unità logica a soglia per $x1 \wedge x2$. La retta mostrata nel diagramma a destra ha l'equazione $3x1 + 2x2 = 4$; per il semipiano grigio, invece, è $3x1 + 2x2 \geq 4$.

Linearmente separabili Sappiamo dall'interpretazione geometrica delle loro computazioni che le Threshold Logic Unit possono rappresentare solo funzioni che sono, come si dice, *linearmente separabili*, cioè funzioni per le quali i punti con output 1 possono essere separati dai punti con output 0 mediante una funzione lineare, ossia mediante una retta, un piano o un iperpiano.

Purtroppo, però, non tutte le funzioni sono linearmente separabili. Un esempio è la doppia implicazione, ossia $x1 \iff x2$. Due insiemi di punti X e Y si dicono linearmente separabili se e solo se i loro gusci convessi sono tra loro disgiunti.

[3]Il vettore normale è perpendicolare a tutti i vettori contenuti nella superficie o nello spazio considerato. E' spesso utilizzato per descrivere la direzione e l'orientamento di una superficie o per calcolare il prodotto scalare tra vettori

Convesso Un insieme di punti X in uno spazio euclideo si dice convesso se e solo se non è vuoto, è connesso e ogni coppia di punti può essere congiunta da un segmento

Convex hull Il più piccolo insieme convesso che contiene X

Doppia implicazione Non è possibile trovare pesi e una soglia tali che una TLU rappresenti la bi-implicazione. In generale, al crescere del numero di argomenti delle funzioni booleane, diminuisce il numero di funzioni che possono essere simulate da una singola Threshold Logic Unit. Tuttavia, è possibile costruire network di unità logiche a soglia per ovviare a questo problema

Networks of Threshold Logic Unit Il potere delle unità logiche a soglia è che, se le combiniamo insieme, possiamo risolvere alcuni compiti che una singola unità logica a soglia non può risolvere da sola. Consideriamo una possibile soluzione per il problema della bi-implicazione utilizzando tre unità logiche a soglia organizzate in due strati. Questa soluzione sfrutta l'equivalenza logica $x1 \iff x2 \equiv (x1 \to x2) \wedge (x2 \to x1)$, che suddivide la doppia implicazione in tre funzioni. Sappiamo già che l'implicazione x2 $\to$ x1 è linearmente separabile.

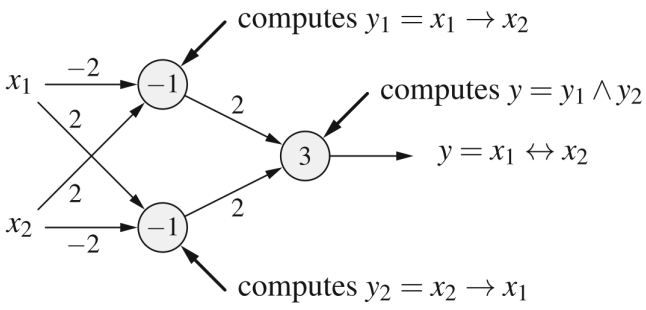


Figura 3: Combina diverse unità logiche a soglia

La linea di separazione $g1$ corrisponde all'unità logica a soglia superiore e descrive l'implicazione $y1 = x1 \to x2$: per tutti i punti sopra questa linea l'output è 1, per tutti i punti sotto di essa l'output è 0. La linea di separazione $g2$ appartiene all'unità logica a soglia inferiore e descrive l'altra implicazione

Tutte le funzioni booleane con input arbitrari possono essere calcolate utilizzando reti di unità logiche a soglia, sfruttando equivalenze logiche per suddividere le funzioni in sottofunzioni linearmente separabili.

1.2.2 Training TLU

La combinazione di più unità logiche a soglia in una rete per calcolare la doppia implicazione può essere interpretata geometricamente. Questo metodo funziona

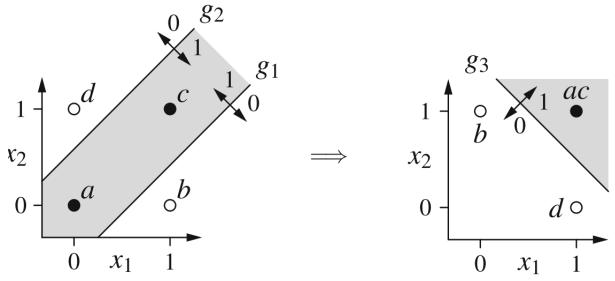


Figura 4: Interpretazione geometrica della combinazione di più Threshold Logic Unit in una rete per calcolare la doppia implicazione

bene per funzioni con 2 o 3 variabili, in cui è possibile trovare facilmente una linea, un piano o un iperpiano che separi i punti per i quali l'output è 1 da quelli per cui l'output è 0. Dai parametri dell'equazione che descrive questa linea, piano o iperpiano, è possibile ricavare facilmente i pesi e la soglia. Tuttavia, questo metodo diventa difficile e infine impossibile per funzioni con più di tre argomenti, poiché non siamo in grado di visualizzare lo spazio di input corrispondente. Inoltre, automatizzare questo metodo è impossibile poiché richiede una valutazione visiva dei set di punti da separare, un processo che non può essere replicato direttamente da un computer.

Per far evolvere una TLU in modo autonomo e automatizzato, possiamo utilizzare un algoritmo che procede nel seguente modo:

1. Iniziamo assegnando valori randomici ai pesi e alla threshold
2. Calcoliamo l'errore nell'output per un insieme di input di controllo. L'errore è una funzione dei pesi e della threshold, $e(w_1, \ldots, w_n, \theta)$ che indica quanto bene, dati i pesi e la soglia, la funzione calcolata coincide con quella desiderata
3. Aggiorniamo i pesi e la threshold in modo da correggere l'errore
4. Iteriamo questo processo finché l'errore si annulla, ovvero finché l'output si avvicina il più possibile all'output desiderato

L'algoritmo permette alla TLU di apprendere autonomamente, adattando i suoi pesi e il threshold per minimizzare l'errore nell'output. Questa lenta e graduale adattazione dei weights e della threshold (stepwise adaptation) viene anche chiamata learning/training. L'obiettivo è determinare quei valori in modo che la funzione abbia un errore pari a 0. Per raggiungere questo obiettivo, cerchiamo di ridurre il valore della funzione di errore ad ogni passo.

Negazione booleana Nell'esempio considerato si vuole addestrare una TLU con un unico input, un unico peso associato, e un threshold per calcolare la negazione booleana. Sia x l'input, w il peso associato e θ la soglia, allora

l'output y sarà definito come:

$$y = \begin{cases} 1 & \text{se } 0w = 0 \geq \theta \\ 0 & \text{se } 1w = w \geq \theta \end{cases}$$

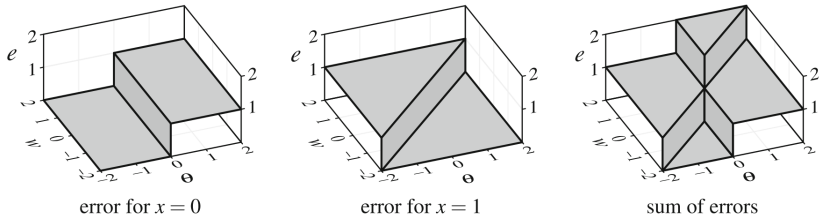


Figura 5: Errore di calcolo della negazione rispetto alla soglia

La funzione di errore viene definita inizialmente come il valore assoluto della differenza tra l'output desiderato e l'output effettivo. Il diagramma a sinistra mostra l'errore per l'input $x = 0$, per il quale si desidera un output di 1. Poiché la TLU calcola un 1 se $xw \geq \theta$, l'errore è 0 per una soglia negativa e 1 per una soglia positiva. (Il peso non ha alcuna influenza, poiché viene moltiplicato per l'input, che è 0).

Il diagramma centrale mostra l'errore per l'input $x = 1$, per il quale si desidera un output di 0. Qui sia il peso che la soglia hanno un'influenza. Se il peso è inferiore alla soglia, abbiamo $xw \leq \theta$ e quindi l'output, e di conseguenza l'errore, è 0. Il diagramma a destra mostra la somma di questi errori individuali.

Dal diagramma a destra, un essere umano può facilmente capire come scegliere il peso e la soglia in modo che la TLU calcoli la negazione: i valori di questi parametri devono trovarsi nel triangolo nella parte inferiore sinistra del piano $w - \theta$, in cui l'errore è 0. L'adattamento automatico dei parametri utilizzando l'ispezione visiva della funzione di errore non è possibile direttamente con un computer. È necessario dedurre la direzione di modifica del peso e della soglia osservando la forma della funzione nell'attuale punto al fine di ridurre l'errore.

Per risolvere questo problema, modifichiamo la funzione di errore. Quando l'unità logica a soglia produce un output errato, consideriamo quanto il limite viene superato (per un output desiderato di 0) o quanto viene sottoutilizzato (per un output desiderato di 1). Intuitivamente, possiamo dire che il calcolo è "più sbagliato" quanto più il limite viene superato (exceeded) per un output desiderato di 0, o quanto più viene sottoutilizzato (underrun) per un output desiderato di 1.

Ci muoviamo semplicemente nella direzione in cui la funzione di errore ha la pendenza più ripida verso il basso. Cerchiamo cioè di *descent in the error landscape*.

Processo di allenamento Ci sono due modi

- *Online learning*, correggiamo l'errore per ogni scelta dell'input

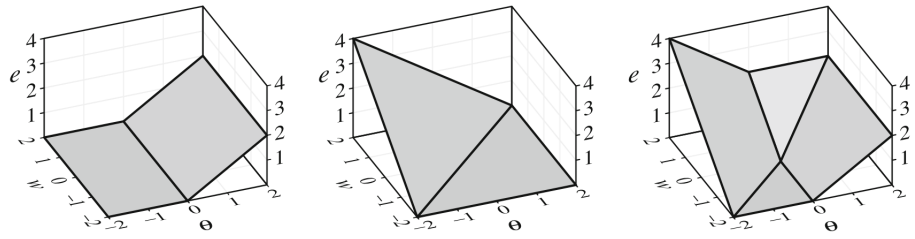


Figura 6: Funzione di errore differenziabile

- *Batch learning*, prendiamo in considerazione l'errore cumulato su una sequenza di input prima di applicare le correzioni

Delta rule o procedura di Widrow-Hoff Possiamo definire il seguente metodo generale di addestramento per le unità logiche a soglia.

Sia $\vec{v} = (x_1, \ldots, x_n)$ il vettore di input di una TLU, a l'output atteso e y il valore attuale. Se $a = y$, abbiamo finito. Al contrario, per ridurre l'errore computeremo nuovi valori per il threshold e i pesi nel seguente modo:

$$\theta^{(new)} = \theta^{(old)} + \Delta\theta \text{ con } \Delta\theta = -\eta(a - y)$$

$$\forall i \in \{1, \ldots, n\} : w_i^{(new)} = w_i^{(old)} + \Delta w_i \text{ con } \Delta w_i = \eta(a - y)x_i$$

dove η è il *learning rate*[4]. Più è alto, più i cambiamenti sui pesi e sui threshold sono elevati.

L'algoritmo utilizza la differenza tra l'output desiderato e l'output effettivo della rete per calcolare l'errore di predizione. Quindi, i pesi vengono aggiornati proporzionalmente a questo errore utilizzando un coefficiente di apprendimento. L'obiettivo è minimizzare l'errore di predizione adattando iterativamente i pesi della rete.

Convergence Theorem for Delta Rule Esiste un teorema che ci garantisce che applicando la *delta rule* l'algoritmo converge ad una soluzione.

Sia $L = \{(\vec{v_1}, a_1), \ldots, (\vec{v_n}, a_n)\}$ una sequenza di pattern di allenamento per la TLU, dove $\vec{v_i}$ sono i vettori di input e a_i l'output atteso. Siano inoltre $L_0 = \{(\vec{v}, a) \in L | a = 0\}$ e $L_1 = \{(\vec{v}, a) \in L | a = 1\}$ rispettivamente gli insiemi delle coppie di pattern che hanno come output atteso 0 e quelle che hanno come pattern atteso 1. Se L_0 e L_1 sono linearmente separabili, allora esiste un $\vec{w}$ vettore di pesi e un θ threshold t.c.:

$$\forall(\vec{v}, 0) \in L_0 : \vec{w}\vec{v} < \theta$$

$$\forall(\vec{v}, 1) \in L_1 : \vec{w}\vec{v} \geq \theta$$

[4]Parametro utilizzato negli algoritmi di apprendimento delle reti neurali che determina la dimensione dei passi di aggiornamento dei pesi durante il processo di adattamento del modello

ADAptive LINear Element Nell'utilizzo delle Threshold Logic Unit per la risoluzione di problemi booleani, si è soliti rappresentare il valore falso come 0 e il valore vero come 1. Questo approccio può presentare un problema nel caso in cui il valore falso venga moltiplicato per un peso, poiché il risultato della moltiplicazione sarebbe sempre 0 e non ci sarebbe modo di modificare il peso per correggere l'errore. Per ovviare a questo, si utilizza una diversa codifica chiamata ADALINE (ADAptive LINear Element), in cui falso viene rappresentato con il valore -1 e vero con 1. Questo consente la modifica dei pesi anche in caso di valore falso.

È importante notare che la regola delta o la procedura Widrow-Hoff sono state originariamente sviluppate per il modello ADALINE (Widrow e Hoff 1960). Pertanto, possiamo correttamente riferirci a questa solo quando si utilizza il modello ADALINE.

1.3 Artificial Neural Network

Grafo diretto Un grafo (orientato) è una coppia $G = (V, E)$ composta da un insieme (finale) V di vertici o nodi e un insieme (finale) $E \subseteq V \times V$ di archi. Diciamo che un arco $e = (u, v) \in E$ è diretto dal vertice u al vertice v

Per descrivere le reti neurali abbiamo bisogno solo di grafi orientati, poiché le connessioni tra i neuroni sono sempre dirette.

Artificial neural network Una rete neurale artificiale è un grafo orientato $G = (U, C)$ i cui vertici $u \in U$ sono chiamati neuroni o unità e i cui archi $c \in C$ sono chiamati connessioni. L'insieme dei nodi U può essere partizionato in tre sottoinsiemi

- U_{in}, nodi di input, ricevono l'informazione dall'ambiente
- U_{out}, nodi di output, comunicano con l'esterno
- U_{hidden}, nodi interni, propagano la computazione

Structure of neural networks Il grafo di una rete neurale viene spesso descritto da una matrice di adiacenza che, invece dei valori 1 (connessione) e 0 (nessuna connessione), contiene i pesi delle connessioni (se un peso è zero, la corrispondente connessione non esiste)

Questa matrice deve essere letta dall'alto verso destra: le colonne corrispondono ai neuroni da cui si originano le connessioni, le righe ai neuroni a cui le connessioni portano. Questa matrice e il grafo pesato corrispondente, ossia il grafo con archi pesati, sono chiamati struttura di rete.

In base alla struttura di rete, possiamo distinguere due tipi fondamentali di reti neurali: se il grafo è aciclico, la rete viene chiamata ***feed forward network***, in

questa rete i calcoli seguono di solito un ordinamento topologico[5] dai neuroni di input progressivamente verso i neuroni di output. Se il grafo contiene cicli viene chiamata ***recurrent network***, in questo caso l'output finale può dipendere dall'ordine in cui i neuroni ricalcolano i loro output.

Ogni neurone possiede quattro variabili:

1. net_u, l'input totale che il neurone riceve dalla rete
2. act_u, la funzione di attivazione applicata all'input
3. out_u, l'output prodotto dal neurone dopo aver applicato la funzione di attivazione
4. ext_u, eventuali input esterni al neurone

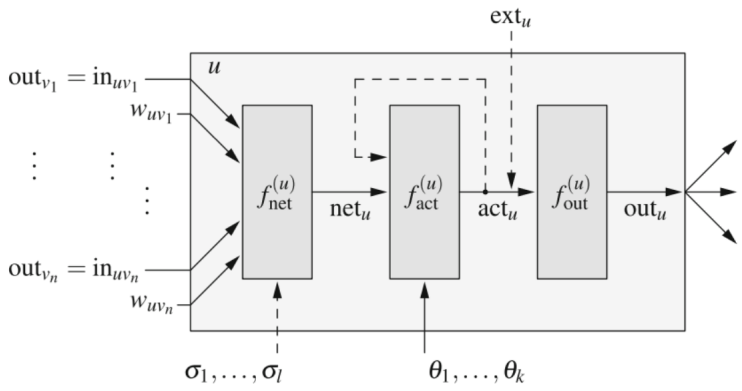


Figura 7: Structure of Neural Network

Le prime tre variabili vengono calcolate in ogni momento dell'evoluzione dell'Artifical Neural Network grazie a tre funzioni associate

- *Network input function* f^u_{net}, calcola la somma pesata dell'input
- *Activation funciont* f^u_{act}, vari modelli. Analizza l'input di rete dalla funzione precedente e genera lo stato di eccitazione del nostro neurone. Questo ci dirà se il neurone è sufficientemente eccitato.
- *Output function* f^u_{out}, definisce l'output a seconda dell'attivazione del neurone. Prende lo stato di eccitazione del neurone e elabora lo stato finale del neurone per consegnarlo ai neuroni successivi

[5]Un ordinamento topologico è una numerazione dei vertici di un grafo diretto, tale che tutti gli archi siano diretti da un vertice con un numero inferiore a un vertice con un numero superiore

Processi interni Dividiamo i calcoli di una rete neurale in due fasi:

- *Input phase*, gli input esterni vengono acquisiti dai neuroni di input. Questa fase ha lo scopo di inizializzare la rete. Le attivazioni dei neuroni di input vengono impostate ai valori degli input esterni corrispondenti. Le attivazioni dei neuroni rimanenti vengono inizializzate arbitrariamente, di solito impostandole semplicemente a 0. Inoltre, la funzione di output viene applicata alle attivazioni inizializzate, in modo che tutti i neuroni producano output iniziali. Se un neurone non riceve alcun input di rete perché non ha predecessori, viene definito che mantiene semplicemente la sua attivazione (e quindi anche il suo output).
- *Work phase*, viene calcolato l'output della rete neurale. Cioè gli input esterni vengono disattivati e le attivazioni e gli output dei neuroni vengono ricalcolati (eventualmente più volte). I ricalcoli vengono interrotti se la rete raggiunge uno stato stabile, ovvero se ulteriori ricalcoli non modificano più gli output dei neuroni oppure se è stato effettuato un numero predeterminato di ricalcoli.

1.3.1 Training Artificial Neural Network

La delta rule ci consente di allenare automaticamente una singola TLU, questo non è generalizzabile alle ANN. Nonostante ciò, i principi su cui si basa la delta rule sono gli stessi utilizzati per allenare le Artificial Neural Network: calcolare le correzioni da applicare ai pesi e alla threshold dei singoli neuroni, aggiornandoli di conseguenza.

Tipi di apprendimento

- *Fixed learning task*, supervisionato
  Avremo un insieme $L = (i_1, o_1), \ldots, (i_n, o_n)$ di coppie che assegnano ad ogni input un output desiderato. Una volta completato l'apprendimento, la ANN dovrebbe essere in grado di restituire l'output adeguato rispetto all'input presentato. In pratica, questo accade raramente e bisogna accontentarsi di un risultato approssimativo. Per giudicare in che misura una ANN si avvicina alla soluzione della fixed learning task si adotta una funzione di errore. Tale funzione viene calcolata come il quadrato della differenza tra l'output desiderato e quello attuale

  $$e = \sum_{l \in L} \sum_{v \in U_{(out)}} e_v^l \rightarrow e_v^l = (o_v^l - out_v)^2$$

  e_v^l è l'errore individuale per una particolare coppia l e un neurone di output v. La scelta di utilizzare il quadrato delle differenze come funzione di errore in primo luogo permette di tener conto degli errori positivi e negativi, evitando che si annullino a vicenda. In secondo luogo, questa funzione è ovunque derivabile, semplificando il processo di aggiornamento dei pesi e dei threshold, e consentendo di calcolare la direzione di miglioramento del modello in ogni punto.

- *Free learning task*, non supervisionato
  Avremo solo una sequenza di input $L = i_1, \ldots, i_n$. Non è possibile definire un output atteso per ogni input. In questo tipo di apprendimento l'obiettivo è quello di produrre un output simile per input simili, senza avere un output preciso da raggiungere.

Normalizzare il vettore di input Stiamo assumendo che gli input e gli output di una rete neurale siano numeri reali. Tuttavia, a volte i dati non sono in questa forma. Nella pratica, incontriamo attributi che hanno diversi tipi di scala, che possono richiedere una fase di pre-processing.

Comunemente lo si scala in modo tale che abbia media uguale a 0 e varianza uguale a 1. Per fare questo si calcola per ogni neurone $uk \in U_{in}$ la media aritmetica μk e la deviazione standard σk degli input esterni

$$\mu_k = \frac{1}{|L|} \sum_{l \in L} ext^l_{u_k} \qquad \sigma_k = \sqrt{\frac{1}{|L|} \sum_{l \in L} (ext^l_{u_k} - \mu_k)^2}$$

1.4 Multi-layer Perceptron

Le Multi-Layer Perceptron sono un tipo di rete neurale feed-forward composta da percettroni organizzati in diversi layer. Questa architettura prevede che ogni layer abbia connessioni solo con il layer successivo, garantendo così una struttura a flusso unidirezionale dell'informazione e riducendo le ricomputazioni durante la propagazione del segnale. I neuroni sono collegati in gruppi, ciascun gruppo riceve l'input solo dallo strato precedente o dall'input esterno. L'output viene inviato al gruppo successivo e ci sono diversi tipi di strati: strato di input, strato nascosto, strato di output. Le MLP rappresentano una delle prime tipologie di Artificial Neural Network sviluppate.

La network input function di ogni neurone nascosto e di ogni neurone di output $u \in U_{hidden} \cup U_{out}$ viene calcolata come la somma pesata (ponderata con i pesi delle connessioni) degli input:

$$f^u_{net}(w_u, i_u) = \sum_{v \in pred(u)} w_{uv} out_v$$

Nel caso delle Threshold Logic Units, abbiamo utilizzato la funzione a gradino come attivazione. Con i Multi-Layer Perceptron possiamo utilizzare funzioni sigmoide[6] come attivazione. Una funzione sigmoide è una funzione monotona non decrescente con:

$$f : \mathbb{R} \to [0, 1] \quad \text{con} \lim_{x \to -\infty} f(x) = 0 \quad \text{e} \lim_{x \to \infty} f(x) = 1$$

Ha una forma che permette un valore minimo e massimo, in modo che lo stato netto possibile dei neuroni sia all'interno del dominio continuo tra 0 e 1 (questo

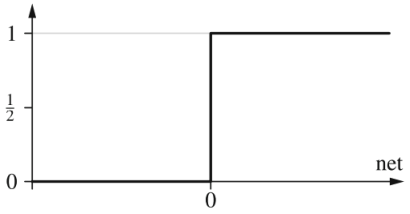
[6]Comunemente nei percettroni multistrato si utilizzano funzioni sigmoide, ma con gli sviluppi più recenti, come nel deep learning, vengono utilizzate diverse funzioni di attivazione. In particolare, la Rectified Linear Unit (ReLU).

è chiamato "squishification" dello stato della rete). Se ho abbastanza eccitazione nello stato della rete, la funzione sigmoide darà un'eccitazione positiva al neurone, viceversa, indicherà un valore più basso nella funzione.

Ci sono molti tipi di funzioni sigmoide per rappresentare il comportamento generale, la più semplice è la funzione gradino. Ha un valore 0 fino a quando l'eccitazione della rete raggiunge il valore Θ, quindi l'uscita dell'eccitazione salta allo stato più forte (il valore di attivazione minimo e massimo è generalmente 0 e 1).

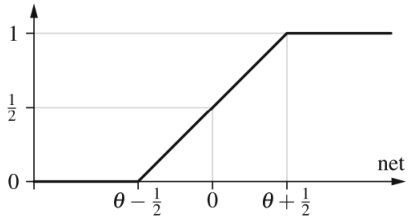
(Heaviside or unit) step function:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} \geq \theta, \\ 0 & \text{otherwise.} \end{cases}$$



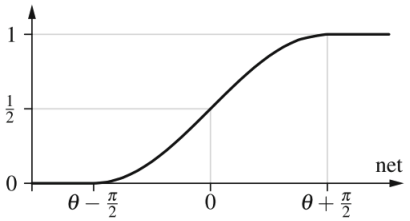
semi-linear function:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} > \theta + \frac{1}{2}, \\ 0 & \text{if net} < \theta - \frac{1}{2}, \\ (\text{net} - \theta) + \frac{1}{2} & \text{otherwise.} \end{cases}$$



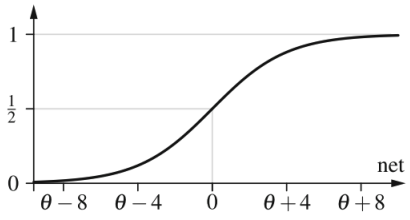
sine up to saturation:

$$f_{\text{act}}(\text{net}, \theta) = \begin{cases} 1 & \text{if net} > \theta + \frac{\pi}{2}, \\ 0 & \text{if net} < \theta - \frac{\pi}{2}, \\ \frac{\sin(\text{net} - \theta) + 1}{2} & \text{otherwise.} \end{cases}$$



logistic function:

$$f_{\text{act}}(\text{net}, \theta) = \frac{1}{1 + e^{-(\text{net} - \theta)}}$$



La funzione di output può essere una sigmoide o una funzione lineare.

Matrice di peso: tutti i pesi tra due strati La struttura a layer suggerisce che il network possa essere descritto da una matrice dei pesi, semplificando la computazione attraverso la moltiplicazione tra matrici e vettori. Siano $U_1 = v_1, \ldots, v_n$ e $U_2 = u_1, \ldots, u_m$ due layer consecutivi di neuroni. I pesi delle loro connessioni sono codificati in una matrice W di dimensioni $n \times m$:

$$W = \begin{pmatrix} w_{u_1 v_1} & w_{u_1 v_2} & \cdots & w_{u_1 v_n} \\ w_{u_2 v_1} & w_{u_2 v_2} & \cdots & w_{u_2 v_n} \\ \vdots & \vdots & \vdots & \vdots \\ w_{u_m v_1} & w_{u_m v_2} & \cdots & w_{u_m v_n} \end{pmatrix}$$

Se due neuroni u_i e v_j non sono connessi, è sufficiente porre $w_{u_i v_j} = 0$. Il vantaggio di questa matrice sta nel fatto che è possibile scrivere il network input di un layer come:

$$net_{U2} = w_{in_{U2}} = w_{out_{U1}}$$

dove $net_{U2} = (net_{u1}, \ldots, net_{um})^T$ e $in_{U2} = out_{U1} = (out_{v1}, \ldots, out_{vn})^T$

Basta prendere il vettore dei neuroni di input del secondo strato (che sono l'output dello strato precedente) e moltiplicarlo per il peso tra gli strati. Poi si applica la funzione sigmoide σ per comprimere i valori tra $[0, 1]$,

1.4.1 Function approximation

Capiamo cosa guadagniamo rispetto alle TLU, ovvero neuroni con la funzione di attivazione a gradino, se consentiamo l'utilizzo di altre funzioni di attivazione. Finora abbiamo visto funzioni booleane, ma ora vogliamo ampliare le dimensioni di una rete feed-forward per considerare non solo la funzione booleana dei TLU, ma vogliamo considerare in un perceptron multistrato qualsiasi tipo di numero reale (ampliare la capacità di descrivere il mondo). Il principio è

- Approssimare una data funzione con una funzione a gradino
- Costruire una rete neurale che calcoli la funzione a gradino
- L'errore è misurato come l'area tra le funzioni

- Per ogni valore di confine del gradino x_i, creiamo un neurone nel primo livello nascosto di un percettrone multistrato a quattro strati. Questo neurone serve a determinare da quale lato del valore di confine del gradino si trova il valore di input
- Nel secondo livello nascosto creiamo un neurone per ogni gradino, che riceve input dai due neuroni nel primo livello nascosto che si riferiscono ai valori x_i e $x_i + 1$ che segnano i confini di questo gradino. I pesi e la soglia sono scelti in modo tale che il neurone sia attivato se il valore di input non è inferiore a x_i, ma è inferiore a $x_i + 1$, cioè se il valore di input si trova nell'intervallo del gradino. Si noti che in questo modo solo un neurone nel secondo livello nascosto può essere attivo, ovvero quello che rappresenta il gradino in cui si trova il valore di input
- Le connessioni dai neuroni del secondo livello nascosto al neurone di output sono ponderate con i valori della funzione dei gradini che sono rappresentati dai neuroni. Poiché solo un neurone può essere attivo nel secondo livello nascosto, il neurone di output riceve in input l'altezza del gradino in cui si trova il valore di input

Anche se l'errore di approssimazione può essere ridotto aumentando il numero di neuroni, ciò non garantisce che la differenza tra l'output del percettrone multistrato e la funzione da approssimare sia inferiore a un certo limite di errore in

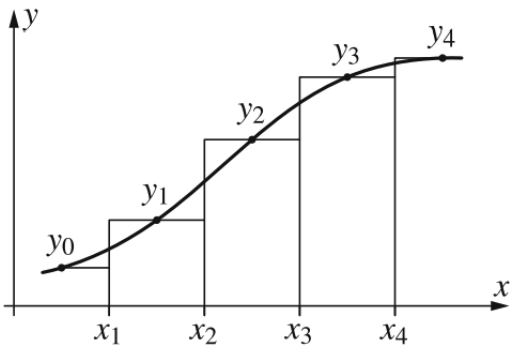


Figura 8: Approssimare una funzione continua con funzioni a gradino. L'errore è l'area tra la funzione da approssimare e la step function

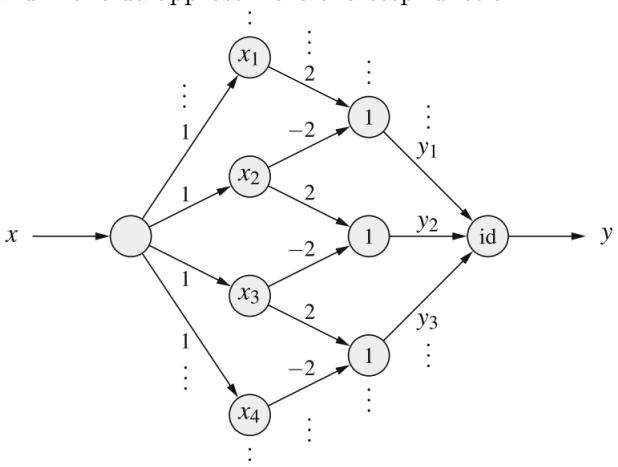


Figura 9: Una rete neurale che calcola la funzione a gradino

ogni punto. Questo perché potrebbero esserci picchi molto sottili nella funzione che non vengono catturati dai gradini.

Non è la forma della funzione di attivazione, ma la struttura stratificata della rete feedforward che rende i percettroni multistrato approssimatori universali.

Delta approximation approach Possiamo effettivamente semplificare l'errore considerando un trucco invece di usare l'eccitazione combinata. Posso suddividere il dominio della funzione per la mia rete in parti come prima, quindi anziché definire il valore della funzione per ogni intervallo, utilizzo iterativamente la variazione Δ rispetto al valore considerato.

La struttura della rete sarà più semplice perché distribuirò l'input dei neuroni nascosti, che mi diranno se il valore x è prima di x_i o maggiore di x_i (non mi interessa dove). Ad esempio, se il valore si trova tra x_1 e x_2, il primo neurone genererà Δ_{y1} e ciò sarà fornito in uscita. Se ho un valore compreso tra x_2 e x_3, avrò comunque x_1 che sarà abbastanza eccitato e genererà il contributo per

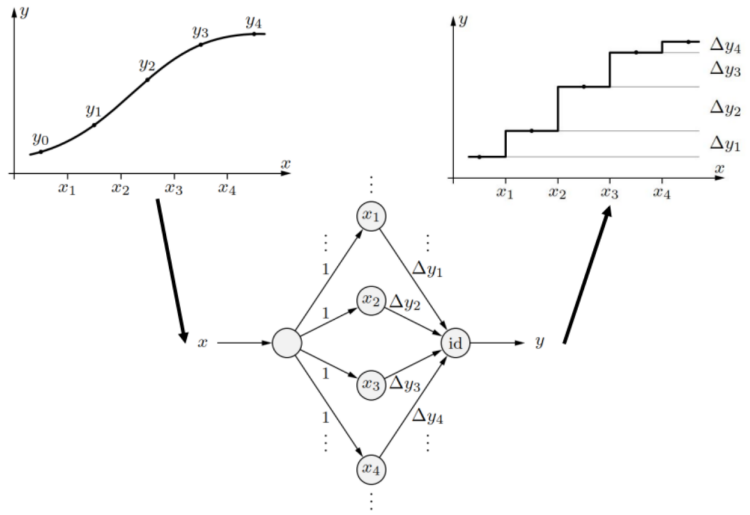


Figura 10: Delta approximation approach

l'output (x_1 e x_2 saranno entrambi eccitati, il loro contributo verrà sommato e quindi fornito in output). Questo renderà più semplice la struttura della rete.

Funzione Reimann Tutte le funzioni Reimann-integrabili possono essere approssimate con precisione arbitraria da percettroni a quattro strati, a condizione che il neurone di output abbia come funzione di attivazione l'identità invece di una funzione a gradino.

Questo teorema richiede solo che la funzione da rappresentare sia Riemann-integrabile, non è necessario che sia continua. In altre parole, la funzione da rappresentare può avere "salti". Tuttavia, può avere solo un numero finito di salti di altezza finita nella regione in cui deve essere approssimata. La funzione deve essere continua "quasi ovunque".

Il teorema mostra che i percettroni multistrato hanno elevata espressività ma nella pratica questo teorema ha scarso utilizzo. Per ottenere un'approssimazione accurata, infatti, richiede la creazione di MLP con un numero eccessivamente elevato di neuroni.

1.5 Regressione

Per allenare una rete neurale artificiale, si deve minimizzare la funzione di errore, che viene spesso calcolata come il quadrato della differenza tra l'output attuale e l'output atteso. Questo processo di apprendimento è simile al concetto più ampio di regressione utilizzato in analisi e statistica per estrarre la linea migliore (o, in generale, il polinomio) che approssima la relazione tra i dati di input e di output. La regressione ci aiuta a trovare i parametri della funzione che

descrive meglio il dataset. A seconda del tipo di funzione che stiamo cercando di approssimare, esistono diverse forme di regressione.

1.5.1 Lineare

Per ottenere una relazione lineare tra due variabili x e y, è necessario trovare i parametri a e b che definiscono la retta $y = g(x) = a + bx$. Tuttavia, potrebbe non essere possibile trovare una singola retta che passi attraverso tutti i punti nel nostro dataset. Pertanto, utilizzeremo una tecnica di regressione per trovare la retta che si avvicina il più possibile ai punti nel dataset. Questo viene fatto minimizzando l'errore, che è calcolato come la somma dei quadrati delle differenze tra i valori predetti e i valori effettivi di y per ogni punto nel dataset

$$F(a,b) = \sum (g(x_i) - y_i)^2 = \sum (abx_i - y_i)^2$$

1.5.2 Polinomiale

Il metodo precedente può essere esteso a polinomi di ordine arbitrario. In questo caso, si prende come ipotesi che la funzione indotta dal dataset approssimi un polinomio di ordine n

$$y = p(x) = a_o + a_1 x + \cdots + a_n x^n$$

E si cercherà di minimizzare la funzione F tale che

$$F(a_1, \ldots, a_n) = \sum (p(x_i) - y_i)^2 = \sum (a_o + a_1 x + \cdots + a_n x^n - y_i)^2$$

Come nel caso della regressione lineare, la funzione potrà essere minimizzata solo se le derivate parziali rispetto ai parametri a_i si annullano

Regressione multilineare E' possibile utilizzare la regressione per calcolare funzioni che dipendono da più di un parametro. L'idea di base rimane quella di trovare i parametri che minimizzano l'errore tra i valori attesi e quelli predetti.

1.5.3 Logistica

Quando ci troviamo in una situazione in cui la relazione tra le quantità nel nostro dataset non può essere approssimata con sufficiente precisione da una funzione polinomiale, potremmo dover utilizzare funzioni di altro tipo, ad esempio

$$y = ax^b$$

Possiamo trasformare questa funzione in una equazione lineare applicando il logaritmo naturale ad entrambi i membri dell'equazione

$$ln(y) = ln(a) + b \cdot ln(x)$$

Il motivo è che il calcolo di un polinomio di best fit può essere utilizzato anche per determinare altre funzioni di best fit. Nel caso delle Artificial Neural Network ci interessiamo in particolare alla funzione logistica

$$y = \frac{Y}{1 + e^{a+bx}}$$

dove Y, a e b sono costanti, esiste anche una trasformazione con la quale il problema può essere ridotto alla computazione di una linea di regressione, chiamata regressione logistica. La funzione logistica viene utilizzata molto frequentemente come funzione di attivazione.

Se una rete neurale utilizza come funzione di attivazione dei neuroni la funzione logistica, è possibile utilizzare il metodo della regressione per determinare i parametri della rete. In particolare, si può applicare il metodo della regressione alla funzione logistica e ottenere i valori dei parametri a e b. In questo modo, si possono determinare i parametri di qualsiasi rete a due layer con un singolo input. Il valore a nella funzione corrisponderebbe alla threshold del neurone di output e la b al peso dell'input

Logit transformation Se la f_{act}^{u} è una logistica si può applicare la regressione per determinare i parametri della rete a, b. Dove a è la treshold del neurone di output, e b è il peso dell'input. Questo significa linearizzare la funzione logistica

Formiamo il valore reciproco dell'equazione logistica e prendiamo il logaritmo di questa equazione

$$y = \frac{Y}{1 + e^{a+bx}} \leftrightarrow \frac{1}{y} = \frac{1 + e^{a+bx}}{Y} \leftrightarrow \frac{Y - y}{y} = e^{a+bx} \leftrightarrow ln(\frac{Y - y}{y}) = a + bx$$

La somma degli errori quadratici può essere calcolata solo per i neuroni di output, limitando così l'uso di questo metodo ai percettroni a due strati, che hanno solo uno layer di input, uno di output senza hidden layer

1.6 Backpropagation

Concettualmente pensiamo che ogni neurone sia collegato a ogni neurone del livello precedente e la somma pesata di questi neuroni esprima la forza di tali connessioni:

$$\sigma(w_1u_1 + w_2u_2 + \ldots w_nu_n + b)$$

Se consideriamo tutti i pesi e i bias inizializzati random, la rete MLP si comporterà piuttosto male perchè agirà casualmente. Quello che si fa è definire una funzione di costo che possa esprimere se il risultato della NN è buono o cattivo per il nostro obiettivo. Maggiore è il valore della funzione, maggiore è l'errore della NN rispetto all'output desiderato.

$$e_v^{(l)} = (o_v^{(l)} - out_v^{(l)})^2$$

Quindi, ciò che si fa è considerare la funzione di costo media su tutti gli esempi all'interno della NN. Vogliamo utilizzare questo errore in modo tale da poter eseguire meglio durante la prossima esecuzione, consideriamo una funzione $C(w)$ che rappresenta solo un costo, se vogliamo trovare l'input che minimizza il costo di questa funzione dobbiamo semplicemente prendere la derivata $dC(w) = 0$, ma ciò non è davvero praticabile per funzioni complesse. Si possono trovare diversi minimi locali della funzione, quindi ciò che si trova non è il valore di costo migliore possibile che si può trovare. Trovare il minimo globale è piuttosto difficile. La nostra funzione sarà molto più complicata, poiché stiamo lavorando con molti parametri (come i pesi e i bias), immaginiamo ora una funzione con due variabili in cui la funzione di costo è rappresentata graficamente come una superficie sopra il piano xy. Ora, per minimizzare la funzione, mi chiedo in quale direzione la funzione $C(x, y)$ diminuisce più rapidamente? Il calcolo multivariato ha un concetto di gradiente, che è la direzione dell'aumento più ripido $\vec{\nabla}C(x, y)$, e inoltre la lunghezza di quel vettore esprime quanto ripido è l'aumento. L'algoritmo sarà essenzialmente:

I) Calcola $\vec{\nabla}C(x, y)$

II) Piccolo passo nella direzione $-\vec{\nabla}C(x, y)$

III) Ripeti fino alla convergenza

Il gradiente negativo della funzione di costo ti dirà come cambiare tutti i pesi e i bias per tutte le connessioni, per ridurre efficientemente il costo. La retropropagazione è un algoritmo per calcolare quel gradiente

1.6.1 Gradient descent

Consideriamo il metodo della discesa del gradiente per determinare i parametri di un percettrone multistrato. Consiste nell'utilizzare la funzione di errore per calcolare la direzione in cui modificare i pesi e la threshold dei neuroni in modo da minimizzare l'errore complessivo della rete. Vogliamo trovare il minimo locale di una funzione di errore per trovare il giusto insieme di pesi

Considerando questa superficie di errore in figura, se ci troviamo nel punto centrale possiamo osservare le derivate della funzione di errore nelle due direzioni, il gradiente ∇ è il vettore tangente alla superficie che ci indicherà la direzione in cui la superficie sta aumentando o diminuendo di più. Se vogliamo trovare la soluzione per il nostro apprendimento, dobbiamo cercare il minimo di questa funzione di errore, quindi in realtà dobbiamo guardare nella direzione opposta al gradiente e cercare di raggiungere il minimo

Per poter utilizzare il gradient descent è necessario che la funzione di errore sia *differenziabile*, ovvero che sia possibile calcolare la sua derivata in ogni punto. In questo modo, è possibile calcolare il gradiente della funzione di errore rispetto ai pesi e utilizzarlo per aggiornarli in modo iterativo fino a raggiungere un valore minimo di errore

1. Inizializzazione dei pesi, assegnati casualmente

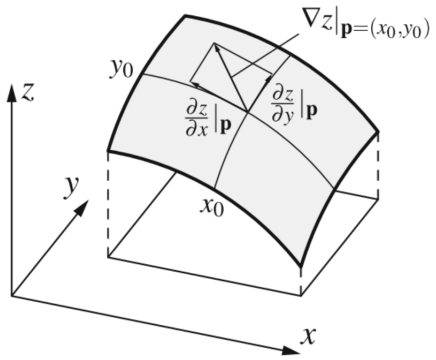


Figura 11: Interpretazione intuitiva del gradiente di una funzione a valori reali $z = f(x, y)$ in un punto (x_0, y_0)

2. Calcolo il gradiente della funzione di errore, rappresenta la direzione e l'entità del cambiamento necessario per ridurre l'errore della rete neurale
3. Aggiornamento dei pesi, facendo un passo nella direzione opposta al gradiente, poiché vogliamo minimizzare l'errore, e aggiornando il learning rate che determina quanto i pesi vengono modificati ad ogni iterazione
4. Ripetizione del processo, fino a quando l'errore della rete neurale viene ridotto a un livello accettabile o fino a quando viene raggiunto un numero massimo di iterazioni

Gradiente Il gradiente è un vettore espresso come n-dimensionale, dove n dipende dai pesi e dai bias, è di solito un numero enorme (impossibile da visualizzare). L'importante è che la magnitudine di ciascuna componente (quindi il valore) ti dirà quanto sensibile sarà l'output della funzione di costo rispetto al peso e al bias (quanto si sposta rispetto al valore precedente). Fondamentalmente, quando parliamo di addestramento o del processo di apprendimento di una NN, si tratta solo di minimizzare la funzione di costo, o funzione di errore.

Formalmente, il calcolo del gradiente restituisce un campo vettoriale. Ciò significa che il gradiente assegna a ogni punto del dominio della funzione un vettore, le cui componenti sono le derivate parziali della funzione rispetto ai suoi argomenti (conosciute anche come derivate direzionali). Questo vettore è spesso chiamato semplicemente il gradiente della funzione nel punto considerato. Punta nella direzione della pendenza più ripida della funzione in quel punto.

L'operatore differenziale ∇ (chiamato nabla o del) viene utilizzato per calcolare il gradiente di un punto o di una funzione. Il principio è partire da un punto scelto casualmente nello spazio di ricerca, fare piccoli passi nello spazio dei parametri sempre nella direzione del massimo aumento (o diminuzione) della funzione da ottimizzare, fino a raggiungere un massimo o minimo locale, almeno approssimativamente. Questa operazione consiste nell'identificare la direzione di massima pendenza (o variazione) in un dato punto, e quindi determinare

la direzione in cui la funzione sta aumentando o diminuendo più rapidamente. Nel caso delle Multiple Layer Perceptron, calcolare il gradiente della funzione di errore si traduce nel calcolare la derivata parziale della funzione di errore rispetto ai pesi e i threshold presi come parametri. Sia $w_u = (-\theta, w_{u_1}, \ldots, w_{u_k})$ il vettore dei pesi di un singolo layer esteso così da includere anche la threshold, calcoliamo il gradiente come segue

$$\nabla_{w_u} e = \frac{\partial e}{\partial w_u} = (-\frac{\partial e}{\partial \theta}, \frac{\partial e}{\partial w_{u_1}}, \ldots, \frac{\partial e}{\partial w_{u_k}})$$

Siccome l'errore totale e è dato dalla somma degli errori individuali rispetto a tutti i neuroni e tutti i training pattern l, otteniamo che

$$\nabla_{w_u} e = \frac{\partial e}{\partial w_u} = \frac{\partial}{\partial w_u} \sum_{l \in L} e^l = \sum_{l \in L} \frac{\partial e^l}{\partial w_u}$$

Error backpropagation Il processo che ci permette di calcolare la correzione necessaria per ogni peso e threshold di ogni singolo neurone dopo aver trovato l'errore.

I valori di errore di ogni layer (nascosto) di un percettrone multistrato possono essere calcolati dai valori di errore del layer successivo. Possiamo anche dire che un segnale di errore viene trasmesso dall'output layer all'indietro attraverso gli hidden layer.

Si assume che la funzione di attivazione sia la funzione logistica per ogni neurone $u \in U_{hidden} \cup U_{out}$ tranne che per quelli di input.

1. *Setting the input*, applichiamo l'input ai neuroni di input che lo restituiscono senza modifiche in output al primo dei layer hidden
2. *Forward propagation of the input*, calcolare per ogni neurone dei livelli successivi la somma pesata degli input e applicare al risultato la funzione logistica, producendo l'output che sarà propagato in tutta la rete fino ai neuroni terminali.
3. *Error factor*, calcoliamo la differenza tra l'output atteso e quello attuale. Utilizzando l'inversa della funzione di attivazione, possiamo risalire all'input che ha causato quell'errore
4. *Error backpropagation*, trasformiamo l'errore dell'output in un errore dell'input possiamo distribuire l'errore (con la correzione) in modo proporzionale indietro al neurone precedente, propagando l'errore fino ai neuroni di input.

L'adattamento dei pesi viene eseguito mediante la seguente formula (che mi dice come eseguire la correzione):

$$\forall w_{up} : \sigma w_{up} = \eta \sigma_u out_p$$

Variazioni gradient descent Uno dei problemi che possiamo incontrare è il "rimanere bloccati" in un minimo locale. Esistono varie sofisticazioni della tecnica del gradient descent che permettono un più veloce apprendimento

I) *Manhattan training*, utilizza al posto del valore del gradiente solo il segno per calcolare la direzione. E' veloce ma non riesce ad ottenere l'ottimo

II) *Flat spot elimination*, è una tecnica utilizzata per affrontare il problema dei plateau, ovvero delle zone in cui la funzione di attivazione del neurone ha una derivata molto bassa o addirittura nulla. Quando ci si avvicina a un plateau, l'abbattimento della lunghezza degli step di apprendimento può limitare notevolmente l'efficienza dell'aggiornamento dei pesi. Per ovviare a questo problema, la tecnica del flat spot elimination solleva artificialmente la derivata della funzione di attivazione nel punto in cui la derivata è molto bassa, in modo da evitare che la lunghezza degli step di apprendimento si abbatta troppo. Questo sollevamento viene fatto aggiungendo un valore costante alla funzione di attivazione nel punto in cui la derivata è bassa.

III) *Momentum term*, viene aggiunto un termine che rappresenta la memoria del passato del cambiamento dei pesi. E' utile perché in molte situazioni il gradiente può avere una grande variazione tra le iterazioni e può essere instabile, quindi l'aggiunta di questo termine può aiutare a mantenere una direzione coerente per il cambiamento dei pesi

IV) *Self-Adaptive Error backpropagation: SuperSAB*, aggiunge un termine di adattamento ai learning rate dei singoli parametri. Il tasso di apprendimento di ogni parametro viene adattato in base al suo gradiente locale. Se il gradiente è grande, viene ridotto, in modo da limitare il rischio di oscillazioni indesiderate. Al contrario, se il gradiente è piccolo, il tasso di apprendimento viene aumentato, in modo da favorire il rapido apprendimento.

V) *Resilient error backpropagation*, combina il manhattan training con il self-adaptive

VI) *Quick propagation*, al posto di utilizzare il gradiente approssimo la funzione con una parabola e salto direttamente all'apice della parabola. Se la funzione di errore è "benigna", il training può avvicinarsi molto al minimo effettivo in un singolo passo di allenamento.

VII) *Weight decay*, tecnica utilizzata per evitare una crescita eccessiva dei pesi con conseguente overfitting del modello

Cross validation Per capire se abbiamo una buona qualità per il nostro apprendimento, dobbiamo dividere casualmente i nostri set di dati in due parti: training set; validation set. Possiamo fare la suddivisione in due modi:

- *Cross validation*, divide casualmente i dati in due sottoinsiemi, addestriamo l'MLP con diversi numeri di neuroni nascosti sui dati di addestramento

e li valutiamo sui dati di convalida. Quindi ripetiamo la suddivisione dei dati, l'addestramento e la valutazione molte volte e facciamo la media dei risultati. Alla fine scegliamo il numero di neuroni nascosti con l'errore medio migliore.

- *k-fold cross validation*, i dati vengono divisi in k sottoinsiemi di dimensioni uguali. Di questi k sottoinsiemi, vengono formati k coppie di set di dati di addestramento e convalida utilizzando 1 fold come convalida e $k-1$ fold per l'addestramento. Il campione viene suddiviso in gruppi di uguale dimensione, un gruppo alla volta viene escluso per verificare la bontà del modello di previsione utilizzato. Interrompiamo l'addestramento quando l'errore di convalida è sufficientemente basso

1.6.2 Overfitting e underfitting

Di quanti neuroni ho bisogno per avere un buon network? Come regola di massima si dovrebbe scegliere il numero di neuroni negli hidden layer secondo la seguente formula:

$$hidden\ neurons = \frac{input\ neurons + output\ neurons}{2}$$

Il numero di neuroni negli hidden layer deve essere valutato in modo da evitare l'*underfitting* o l'*overfitting*. Un numero troppo basso di neuroni può portare ad un modello insufficientemente complesso, mentre un numero troppo elevato può far sì che il modello si adatti troppo ai dati di training, generando un'alta varianza e quindi un alto errore di generalizzazione. Per questo motivo, è importante dividere il dataset in due sottoinsiemi: training set e validation set. In questo modo, si può valutare l'accuratezza del modello su dati mai visti durante il processo di apprendimento e scegliere il numero ottimale di neuroni negli hidden layer per minimizzare l'errore di generalizzazione.

1.6.3 Sensitivity analysis

Le Artificial Neural Network presentano uno svantaggio in termini di comprensione della conoscenza acquisita durante il processo di apprendimento. Questa viene rappresentata in matrici a valori reali che risultano di difficile interpretazione per l'utente. Esiste un'interpretazione geometrica dei processi interni alle ANN ma offre poco aiuto all'intuizione quando lo spazio degli input supera le tre dimensioni. Per superare questo problema, una soluzione è quella di eseguire una *sensitivity analysis*, che determinerà l'influenza dei vari input sull'output del network. Per eseguirla, occorre calcolare la somma delle derivate parziali degli output rispetto agli input esterni per ogni neurone di output e ogni training pattern. Questa somma viene infine divisa per il numero di training pattern, per rendere la misura indipendente dalla grandezza del dataset.

$$\forall u \in U_{(in)}: \quad s(u) = \frac{1}{|L|} \sum_{l \in L} \sum_{\nu \in U_{(out)}} \frac{\partial out_\nu^l}{\partial ext_u^l}$$

Il valore $s(u)$ indica quanto importante fosse l'input assegnato al neurone u per la computazione del Multi-Layer Perceptron. Potremmo decidere di semplificare il network eliminando i nodi con i valori di $s(u)$ più bassi.

1.7 Deep learning

La rete di apprendimento profondo è essenzialmente un MLP, ma con una differenza: ciò che vogliamo fare è non avere alcun tipo di conoscenza sul problema e cercare di forzare la NN a configurarsi con un numero limitato e piccolo di neuroni.

La formula di Reinmann ha mostrato come un MLP con un solo hidden layer può approssimare ogni funzione continua su $\mathbb{R}^n$ con una precisione arbitraria. Può non essere semplice conoscere a priori il numero esatto di neuroni necessari per approssimare una data funzione. Inoltre, a seconda della funzione, questo numero potrebbe assumere dimensioni considerevoli.

Un esempio è quello della funzione che calcola la parità su una parola di n-bit. L'output sarà 1 se e solo se nel vettore di input che rappresenta la parola saranno ad 1 un numero pari di bit. Nel caso scegliessimo di utilizzare un Multi-Layer Perceptron con un solo hidden layer questo avrà al suo interno 2^{n-1} neuroni, in quanto la forma normale disgiuntiva della funzione di parità su n-bit è una disgiunzione di 2^{n-1} congiunzioni.

L'utilizzo di più di un hidden layer promette di ridurre il numero di neuroni necessari. Questo è il focus dell'area del deep learning, dove la "profondità" di una rete neurale è quella che separa i neuroni di input da quelli di output. Una maggiore profondità del network in cambio di un miglioramento delle risorse utilizzate nel calcolo e nella costruzione. Il deep learning oltre ad offrire vantaggi porta con se alcune problematiche: *Overfitting*, l'aumento dei neuroni dati dagli ulteriori layer nascosti può moltiplicare i parametri in modo sproporzionato; *Vanishing gradient*

Soluzioni overfitting

- *Weigth decay*, limita i valori che possono assumere i pesi della rete neurale, ponendo un tetto massimo alla grandezza dei pesi. In pratica, durante l'addestramento, viene aggiunto un termine alla funzione di costo che penalizza i pesi più grandi, in modo che i pesi non raggiungano valori troppo elevati.
- *Sparsity constraint*, limite al numero di neuroni negli hidden layer, oppure limite sul numero di quelli attivi
- *Dropout training*, alcuni neuroni degli hidden layer vengono omessi durante l'evoluzione del network

1.7.1 Vanishing gradient

Il vanishing gradient è un problema che si verifica in una rete neurale quando il gradiente utilizzato nell'algoritmo di backpropagation dell'errore diminuisce significativamente in valore man mano che ci si avvicina ai primi layer della rete. Questo problema si verifica spesso nelle reti neurali con funzioni di attivazione non lineari e derivate che assumono valori compresi tra 0 e 1. Nel caso della funzione di attivazione logistica, la sua derivata raggiunge al massimo il valore di $\frac{1}{4}$. Quando il gradiente diventa molto piccolo, l'aggiornamento dei pesi diventa meno efficace, e quindi la rete neurale impiega più tempo per apprendere e può anche avere difficoltà a convergere

Modificare la funzione di attivazione Una soluzione è quella di modificare leggermente la funzione di attivazione in modo che sia sempre crescente. Alcuni candidati proposti in letteratura sono la *ramp function* e la *softplus function*

Autoencoder Un approccio completamente diverso consiste nel costruire il perceptron multistrato strato per strato, addestrando solo il nuovo strato aggiunto in ogni passaggio. Una procedura molto popolare per fare ciò è costruire la rete come *stacked autoencoder*. Un autoencoder è un perceptron multistrato che mappa i suoi input in un'approssimazione di questi input, utilizzando un hidden layer.

Lo strato nascosto (e le connessioni che lo collegano allo strato di input) forma un encoder che trasforma gli input in una forma di rappresentazione interna, che viene poi decodificata dallo strato di output (e dalle connessioni dallo strato nascosto allo strato di output). Un tale autoencoder viene addestrato con la back-propagation standard, o una delle sue varianti, che non soffre molto del problema del vanishing gradient, poiché c'è solo uno strato nascosto.

1. *Sparse autoencoder*, prevede di utilizzare un numero molto minore di neuroni nell'hidden layer, rispetto a quelli di input in modo che l'autoencoder sia costretto ad imparare alcune caratteristiche effettivamente rilevanti invece di propagare solo i dati.
2. *Sparse activation scheme*, disattivare casualmente alcuni neuroni durante la fase di calcolo
3. *Denoising autoencoder*, si aggiunge rumore all'input

Layer-wise training: addestramento strato per strato Per costruire un MLP con molti strati, è possibile utilizzare una tecnica chiamata layer-wise training. Inizialmente si allena un autoencoder con un singolo strato nascosto utilizzando la backpropagation per minimizzare l'errore di ricostruzione tra l'input e l'output dell'autoencoder. Una volta addestrato, il decoder dell'autoencoder viene rimosso e viene conservato solo il layer nascosto, che viene utilizzato come

input per l'addestramento di un nuovo autoencoder con un altro strato nascosto. Questo processo viene ripetuto fino a raggiungere il numero desiderato di strati nascosti.

Questa tecnica è stata utilizzata con successo per la classificazione di immagini, ad esempio nel riconoscimento di numeri scritti a mano. Tuttavia, per applicazioni più complesse in cui le feature da riconoscere non sono localizzate in una specifica parte dell'immagine, sono necessarie architetture più avanzate come le Convolutional Neural Network (CNN).

Convolutional Neural Network Questo tipo di rete è ispirato alla struttura della retina umana, in cui i neuroni sensoriali hanno un cosiddetto campo recettivo, cioè una regione limitata in cui rispondono a uno stimolo visivo. Questo viene imitato in una CNN connettendo ogni neurone nello strato nascosto (primo) solo a un piccolo numero di neuroni di input contigui (nell'immagine). I pesi della rete sono condivisi tra le diverse regioni, in modo che la stessa feature possa essere rilevata in posizioni diverse dell'immagine. Durante la computazione, il campo ricettivo viene spostato su tutta l'immagine e viene eseguita una convoluzione tra la matrice dei pesi e l'immagine in input per produrre una mappa delle feature. Questa mappa viene quindi sottoposta ad un pooling, ovvero ad una riduzione della dimensione, per ridurre la complessità del modello e prevenire l'overfitting. Questo processo viene ripetuto per tutti gli strati nascosti della rete fino a raggiungere l'output finale.

1.8 Radial Basis Function Network

Le RBFN hanno diversi use cases, come function approximation, time series prediction, classificazione. E' strettamente limitata ad avere un solo strato nascosto, che chiamiamo *feature vector*. L'input layer si limita a ricevere i dati di ingresso e feedarli nell'hidden layer, il calcolo che avviene nello strato nascosto è molto diverso da quello della maggior parte delle reti neurali. Lo strato di output esegue operazioni di previsione come classificazione o regressione.

- *Input layer*: i neuroni sono completamente connessi con lo strato nascosto, non fanno nessuna computazione ma solo feeding dei dati
- *Hidden layer*: prende in input un modello che potrebbe non essere linearmente separabile e lo trasforma in un nuovo spazio più linearmente separabile, per questo il numero di neuroni è maggiore rispetto a quello di input per il teorema di Cover
- *Outuput layer*: funzione di attivazione lineare

I Radial Basis Function Network sono reti neurali feed-forward con tre layer di neuroni, questo significa che c'è esattamente un layer nascosto. Rispetto ai Multi-Layer Perceptron, hanno una funzione di attivazione diversa, chiamata funzione radiale di base. Nei neuroni di output, la somma pesata degli input avviene come nei MLP. Nel layer nascosto, invece, la funzione *net* è una funzione

di distanza tra il vettore di input e il vettore dei pesi. Questa distanza deve rispettare tre assiomi geometrici per essere considerata una metrica.

$$d(w,v) = 0 \leftrightarrow w = v \ \ (equality)$$

$$d(w,v) = d(v,w) \ \ (simmetry)$$

$$d(w,e) + d(e,v) \geq d(w,v) \ \ (triangle\ inequality)$$

La funzione di input della rete per i neuroni di output è la somma pesata degli input. La funzione di attivazione di ogni neurone nascosto è una funzione radiale. La funzione di attivazione di ogni neurone di output è una funzione lineare. La funzione di input di rete e le funzioni di attivazione di un neurone nascosto descrivono una sorta di "zona di influenza" di questo neurone. I pesi delle connessioni dallo strato di input a un neurone dello strato nascosto indicano il centro di questa regione, poiché la distanza (funzione di input di rete) viene misurata tra il vettore dei pesi e il vettore di input. Il tipo di funzione di distanza determina la forma della zona di influenza

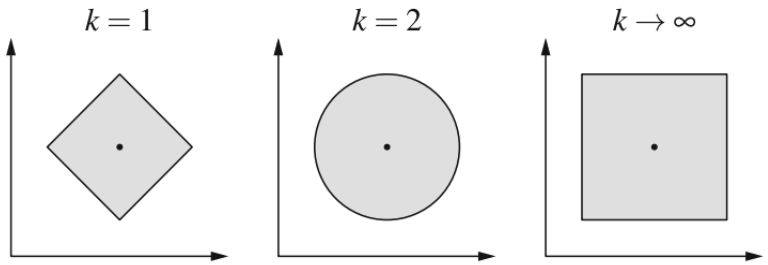


Figura 12: Cerchi per diverse funzioni di distanza. Tutti i cerchi hanno lo stesso raggio.

In modo intuitivo, la funzione di attivazione di un hidden neuron e i suoi parametri determinano la "dimensione" della zona di influenza del neurone specificando quanto sia forte l'influenza di un vettore di input in base alla sua distanza dal vettore dei pesi. Chiamiamo questa funzione di attivazione una funzione radiale, perché è definita lungo un raggio da un centro, che è descritto dal vettore dei pesi e assegna a ciascun raggio (cioè, a ciascuna distanza dal centro) un'attivazione.

Una famiglia di funzioni usate spesso nelle applicazioni è quella formulata dal matematico Minkowski e rinominata in suo onore famiglia di Minkowski. Tale famiglia è definita come

$$d(w,v)_k = (\sum (w_i - v_i)^k)^{\frac{1}{k}}$$

Alcune funzioni appartenenti alla famiglia

$$k = 1 : Manhattan\ distance$$

$$k = 2 : Euclidian\ distance$$

$$k = \infty : Maximum\ distance, ovvero\ d(w,v)_\infty = max|w_i - v_i|$$

La funzione di attivazione di ciascun hidden neuron è una funzione radiale cioè una funzione monotona non crescente

$$f : \mathbb{R}^+ \to [0,1] \quad con \quad f(0) = 1 \quad e \quad \lim_{x \to \infty} f(x) = 0$$

Questa funzione calcola l'area in cui il neurone focalizza la propria attenzione definita dal raggio di riferimento σ.

Approximation functions Un RBFN ha la capacità di approssimare funzioni con un errore arbitrariamente piccolo, allo stesso modo di un MLP, e quindi può essere considerato come un approssimatore universale. Questo significa che può approssimare qualsiasi funzione Riemann-integrabile.

1.9 Learning Vector quantization

LVQ è un algoritmo di classificazione supervisionato basato su prototipi. E' un precursore delle Self-Organizing Maps (SOM). Addestra la sua rete attraverso un algoritmo di Competitive Learning simile alla Self-Organizing Map

Fino ad ora abbiamo descritto l'apprendimento delle Aritificial Neural Network in relazione ai fixed learning task, dove l'obiettivo è di adattare la rete per approssimare gli output desiderati per un determinato insieme di input. Tuttavia, ci sono situazioni in cui non siamo in grado di conoscere gli output desiderati per tutti i nostri dati in ingresso. In questi casi, l'obiettivo della rete neurale può essere quello di raggruppare o classificare i dati in modo automatico, senza alcuna indicazione su quale sia l'obiettivo specifico. Una tecnica che aiuta nel clustering automatico dei dati è la Learning Vector Quantization.

Quantizzazione vettoriale dell'apprendimento (Kohonen 1986). L'obiettivo di questo metodo è organizzare i dati in cluster, i singoli cluster sono rappresentati da un centro. Questo centro viene posizionato in modo che si trovi approssimativamente al centro del gruppo di punti dati che costituisce il cluster.

1.9.1 Vector quantization network

La learning vector quantization utilizza una rete feed-forward a due layer chiamata Learning Vector Quantization Network (LVQN). Questa rete può essere vista come una variante della Radio Basis Function Network, in cui il layer di output sostituisce il layer nascosto. Come nelle RBFN, la funzione di input del layer di output dipende dalla distanza tra il vettore di input e i pesi associati ai neuroni e la funzione di attivazione dei neuroni di output è una funzione radiale.

La differenza principale rispetto alle RBFN è che la $f_{(out)}$ non è l'identità, ma viene propagata solo se l'attivazione del neurone è la massima tra tutti i neuroni di output.

Winners takes all Il neurone con la maggiore attivazione produce l'output 1, tutti gli altri neuroni producono output 0

Competitive learning L'apprendimento competitivo nella LVQ comporta la competizione tra i neuroni di output per attivarsi in risposta a un punto di input, con il neurone vincitore che viene adattato per avvicinarsi al punto e gli altri neuroni che vengono spinti lontano. Questo processo si ripete per ogni punto di input nel dataset, consentendo alla rete neurale di adattare i suoi vettori di riferimento per una migliore rappresentazione dei dati di input

Attraction rule Sposta il vettore di riferimento verso il punto di input

$$r^{new} = r^{old} + \eta(x - r^{old})$$

Repulsion rule Sposta gli altri vettori di riferimento associati ai neuroni non vincenti lontano dal punto di input

$$r^{new} = r^{old} - \eta(x - r^{old})$$

dove x è l'input, r è il vettore di riferimento per il neurone vincitore e η è il learning rate

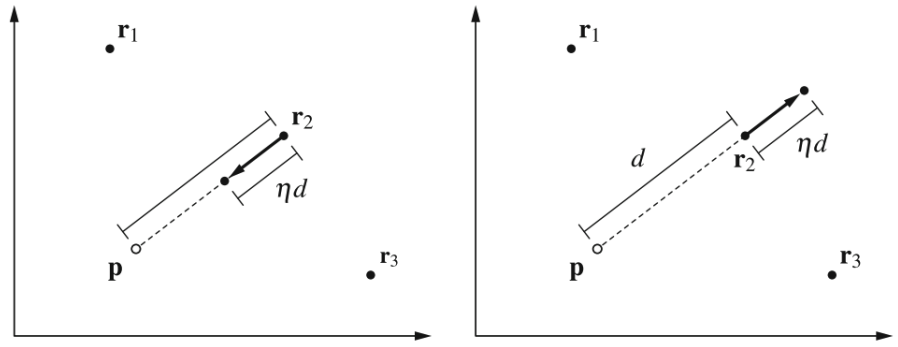


Figura 13: Sinistra attraction rule, destra repulsion rule

Dopo la competizione, i vettori di riferimento associati al neurone vincente vengono adattati in modo che il vettore di riferimento sia mosso più vicino al punto di input, mentre gli altri vettori di riferimento associati ai neuroni non vincenti vengono allontanati dal punto

Time dependent learning rate Finora abbiamo dato per scontato che η rimanesse costante durante l'intero processo di apprendimento, ma ci sono alcune situazioni in cui un tasso di apprendimento costante può causare problemi. Infatti, se è troppo elevato, la rete neurale potrebbe oscillare attorno al punto di ottimizzazione senza mai raggiungerlo, mentre se è troppo basso, l'apprendimento potrebbe essere troppo lento e richiedere molto tempo per convergere verso un minimo globale

Per risolvere il problema del η costante, un metodo è quello di far decrescere il learning rate al crescere delle iterazioni. In questo modo, il movimento circolare collassa col passare del tempo in una spirale, facendo così convergere l'algoritmo

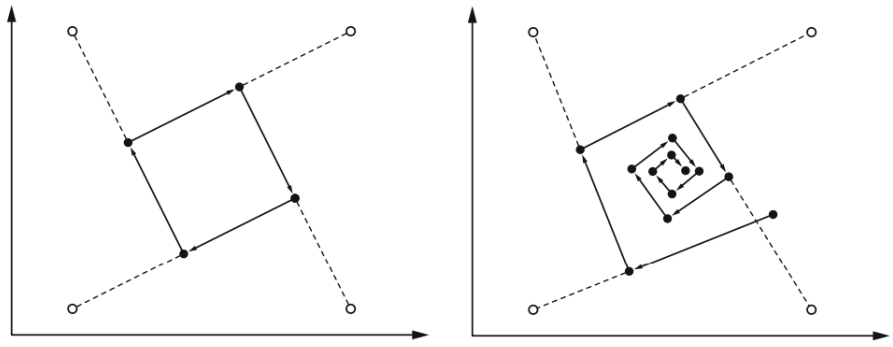


Figura 14: Learning rate costante a sinistra, decrescente a destra

1.10 Self-organizing maps

Una Self-Organizing Map è una tecnica di apprendimento automatico non supervisionato utilizzata per produrre una rappresentazione a bassa dimensione (tipicamente bi o tri dimensionale) di un insieme di dati di dimensioni più elevate, preservando la struttura topologica dei dati. E' un tipo di ANN trainata utilizzando competitive learning invece della back-propagation con il gradient descent. Come la maggior parte delle reti neurali artificali, operano in due modalità:

- *Training*, utilizza un insieme di dati di input (input space) per generare una rappresentazione a bassa dimensione (map space)
- *Mapping*, classifica ulteriori dati di input utilizzando la mappatura generata

Le SOM, o mappe delle caratteristiche di Kohonen, sono strettamente correlate alle Radial Basis Function Network. Possono essere considerate reti a funzione di base radiale senza uno strato di output, o, più precisamente, l'hidden layer di una Radial Basis Function Network è già lo strato di output di una Self-Organizing Maps. I neuroni sono disposti in una griglia.

Una SOM è una rete neurale a due strati senza hidden neuron. La sua struttura corrisponde essenzialmente allo strato di input e allo strato nascosto delle Radial Function Base Network

Le self-organizing maps, o *mappe delle caratteristiche di Kohonen*, sono una tipologia di rete neurale feed-forward a due strati che possono essere considerate come una generalizzazione delle Learning Vector Quantization. A differenza di quest'ultime, le connessioni tra i neuroni nascosti e quelli di output sono limitate solo ai neuroni "vicini". In pratica, ogni neurone di output è associato a un'area

della mappa bidimensionale e, durante la fase di apprendimento, i neuroni della mappa vengono organizzati in modo tale da riflettere la struttura dei dati in input. In questo modo, i neuroni della mappa vicini a quelli attivati da un particolare input si attivano anch'essi, formando una sorta di cluster o regione di attivazione. Le SOM possono essere utilizzate per la visualizzazione dei dati, la classificazione e la riduzione della dimensionalità.

La funzione di output di ciascun neurone di output è l'identità. L'output può essere discretizzato secondo il principio del *winners takes all*.

Come nel caso dei LVQN, la $f_{(net)}$ dei neuroni di output è una funzione di distanza tra il vettore di input e quello dei pesi, e la $f_{(act)}$ è una funzione radiale[7].

Sui neuroni dello strato di output è definita una relazione di vicinato, descritta da una funzione di distanza

$$d_{\text{neurons}} : U_{\text{out}} \times U_{\text{out}} \to \mathbb{R}^+$$

Questa funzione assegna un numero reale non negativo a ciascuna coppia di neuroni di output.

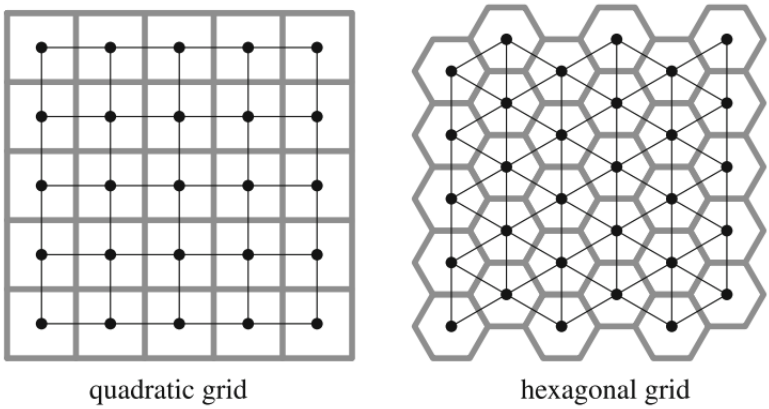


Figura 15: Esempi di disposizioni dei neuroni di output di una Self-Organizing Map. Ogni punto corrisponde a un neurone di output. Le linee sono intese per rendere più chiara la struttura di vicinato

In analogia alle reti a funzione di base radiale, nelle mappe auto-organizzanti i pesi delle connessioni dall'input ai neuroni di output rappresentano le coordinate di un centro, chiamato *reference vector*, rispetto al quale viene misurata la distanza di un pattern di input. Più un pattern di input si avvicina al vettore di riferimento, maggiore sarà l'attivazione del neurone corrispondente. Questo significa che il neurone di output che ha il vettore di riferimento più simile al pattern di input avrà un'attivazione più alta rispetto agli altri neuroni di output.

[7]Funzione che restituisce un valore basato sulla distanza tra l'input e un centro specifico nello spazio

Quando si parla di quantizzazione vettoriale dello spazio di input, si intende che la mappa auto-organizzante suddivide lo spazio di input in diverse regioni, ognuna delle quali è associata a un neurone di output. Ogni regione rappresenta un insieme di input simili che sono mappati sul neurone di output corrispondente. Quindi, la mappa auto-organizzante consente di raggruppare i pattern di input in base alle loro caratteristiche simili, creando una rappresentazione compressa dello spazio di input.

Proiezioni di Robinson La self-organizing map è una funzione che preserva la topologia, cioè conserva la posizione relativa tra i punti del dominio. Un esempio di funzione che preserva la topologia sono le proiezioni di Robinson, utilizzate per costruire le mappe del globo. Queste proiezioni conservano la posizione relativa tra i punti anche se la proporzione della distanza tra due punti nell'originale e nella proiezione è maggiore quanto più ci si allontana dall'equatore. Utilizzando queste funzioni, è possibile mappare spazi multidimensionali in spazi con dimensioni minori, offrendo un vantaggio in termini di efficienza computazionale.

Applicato alle Self-Organizing Maps, i punti di intersezione delle linee della griglia sulla sfera potrebbero indicare la posizione dei vettori di riferimento nello spazio di input, mentre i punti di intersezione delle linee della griglia nella proiezione indicano la posizione dei neuroni di output e la loro relazione di vicinato.

Competitive training Nelle SOM, l'apprendimento si basa sul competitive training, dove ogni input viene assegnato al neurone con l'attivazione più alta. Non solo il neurone vincitore viene aggiornato, ma anche i suoi vicini, sebbene in misura minore. Questo evita che i vettori di riferimento dei neuroni vicini si muovano arbitrariamente lontani l'uno dall'altro, mantenendo così la topologia dello spazio di input. La regola di apprendimento utilizzata per trovare la corretta funzione che preservi tale topologia è una *generalizzazione della attraction rule.*

$$r^{new} = r^{old} + \eta(t) f_{nb}(d_{neuroni}(u, u_*), \rho(t))(x - r^{old})$$

dove u_* è il neurone vincitore e f_{nb} è una funzione radiale. Il learning rate η è parametrizzato rispetto al tempo perché varierà con il numero delle iterazioni

1.11 Hopfield network

E' un tipo di rete neurale artificiale nota per essere il modello di rete che simula le capacità del cervello umano di ricordare le cose o di ricostruire le immagini distorte. Questo modello è classificato come apprendimento non supervisionato, la rete impara senza avere esempi, soltanto con l'uso del concetto di *energia.*

In precedenza ci siamo concentrati esclusivamente sui modelli di reti neurali feed-forward, ovvero modelli di reti neurali rappresentati da grafi aciclici. Tuttavia, esistono anche modelli di *recurrent network*, il cui grafo di connessione presenta dei cicli diretti. Tra questi, uno dei modelli più semplici è quello degli

Hopfield Network, che ha avuto origine come modelli fisici per descrivere il magnetismo. Rispetto agli altri tipi di Artifial Neural Network, gli HN presentano alcune differenze: tutti i neuroni sono contemporaneamente neuroni di input e di output, non ci sono hidden neuron e ogni neurone è connesso con tutti gli altri neuroni (sono eslusi cappi) con pesi di connessione simmetrici.

La rete può essere rappresentata come un grafo completamente connesso, con pesi simmetrici tra i neuroni e archi a loop con peso 0. Poiché lo strato di input e lo strato di output sono fusi nello stesso strato, il numero di input e output della rete è uguale al numero di neuroni.

La funzione di input di ogni neurone è determinata dalla somma pesata degli output degli altri neuroni.

$$f^u_{(net)}(w, i) = \sum_{v \in U - \{u\}} w_{uv} out_v$$

La funzione di attivazione è una threshold function

$$f^u_{(act)}(net_u, \theta_u) = \begin{cases} 1 & \text{se} \quad net_u >= \theta_u \\ -1 & \text{se} \quad net_u < \theta_u \end{cases}$$

Possiamo rappresentare un Hopfield Network attraverso la sua matrice dei pesi che contiene la forza delle connessioni tra i neuroni, visto che i neuroni non si connettono a se stessi, i pesi sulla diagonale sono nulli.

$$W = \begin{bmatrix} 0 & w_{u_1 u_2} & \dots & w_{u_1 u_n} \\ w_{u_2 u_1} & 0 & \dots & w_{u_2 u_n} \\ \vdots & \vdots & \vdots & \vdots \\ w_{u_n u_1} & w_{u_n u_2} & \dots & 0 \end{bmatrix}$$

La funzione di output è la funzione identità. Per ogni neurone, l'output viene calcolato con una funzione che confronta la somma pesata degli input con la soglia e restituirà 1 se il valore è superiore o uguale alla soglia, altrimenti 0. Il problema di questo tipo di rete è che l'output di ogni neurone si diffonde a tutti gli altri neuroni: se l'aggiornamento dello stato viene fatto parallelamente, non sarà possibile garantire uno stato finale stabile della rete, ma cambierà continuamente. La soluzione è eseguirlo sequenzialmente (considerando il teorema di convergenza) e convergerà a uno stato finale in al massimo $n \cdot 2^n$ passaggi.

Teorema convergenza Se i neuroni di un HN sono aggiornati in modo asincrono allora uno stato stabile viene raggiunto al massimo in $n \cdot 2^n$ passi, dove n è il numero dei neuroni. La prova del teorema si basa sul calcolo dell'energia del sistema

$$E = -\frac{1}{2} \sum_{u,v \in U, u \neq v} w_{uv} act_u act_v + \sum_{u \in U} \theta_u act_u$$

Gli HN sono in grado di raggiungere uno stato stabile che corrisponde ad un minimo locale della funzione energia del sistema. Ciò significa che il network può solo evolvere verso uno stato con energia minore rispetto a quello attuale. Questo

fatto può essere sfruttato per utilizzare gli HN come *memorie associative*, dove si collega un dato allo stato stabile raggiunto dopo averlo processato attraverso il network. Inoltre, gli HN possono essere impiegati per risolvere *problemi di ottimizzazione*, dove la ricerca della soluzione ottimale corrisponde alla ricerca del minimo globale della funzione energia del sistema

Use cases

- Memoria associativa, un esempio è la nostra capacità di riconoscere un'immagine anche quando questa non è esatta o riconosere un vecchio amico rivedendolo dopo anni. La memoria associativa è un sistema in cui il richiamo della memoria è avviato dall'associabilità di un modello di input a uno memorizzato. Questo ci permette di recuperare e completare un ricordo utilizzando solo una parte incompleta o rumorosa di esso
- Risoluzione di problemi di ottimizzazione
- Simulated annealing, è un'evoluzione del gradiente che accetta di esplorare soluzioni che sono peggiori di quella corrente, permettendo di sfuggire agli ottimi locali. Con una rete di Hopfield questa tecnica viene implementata controllando, per ogni transizione di stato, se lo stato raggiunto avrà un'energia più bassa o più alta. Nel primo caso, la transizione viene sempre accettata perché descrive una soluzione migliore, nel secondo caso la transizione viene accettata solo con una certa probabilità proporzionale al peggioramento dell'energia che sarebbe ottenuto.

1.12 Boltzmann machines

Le macchine di Boltzmann sono simili alle Hopfield Network in quanto entrambi utilizzano una funzione energia associata ad ogni stato per risolvere problemi di ottimizzazione. Tuttavia, le macchine di Boltzmann sono più complesse, poiché possono contenere hidden neuron e hanno una procedura di aggiornamento differente.

In questa fase di aggiornamento viene selezionato casualmente un neurone e, con i suoi input, vengono calcolati i delta di energia con cui viene calcolata la probabilità di attivazione. Questa procedura viene ripetuta diverse volte selezionando casualmente i neuroni. Alla fine degli aggiornamenti, lo stato finale è indipendente dallo stato iniziale e dai neuroni iniziali aggiornati, e questo stato finale è chiamato Equilibrio Termico.

La procedura di aggiornamento descritta sopra non è sempre utilizzabile:

- È computazionalmente pesante e quindi non è applicabile a reti di grandi dimensioni
- Funziona bene con dati distribuiti secondo una distribuzione di Boltzmann, ma non così bene con altre distribuzioni. Una soluzione a questo problema è utilizzare la cosiddetta "Restricted Boltzmann Machine"

La funzione energia viene utilizzata per definire una distribuzione di probabilità (di Boltzmann) rispetto agli stati del network

$$P(\overrightarrow{s}) = \frac{1}{c} e^{-\frac{E(\overrightarrow{s})}{kT}}$$

dove s è l'insieme degli stati, c una costante di normalizzazione, E la funzione energia, T la temperatura del sistema, k la costante di Boltzmann ($k \simeq 1,38 \cdot 10^{-23}$). I possibili valori di attivazione dei neuroni all'interno del sistema rappresentano gli stati del sistema stesso. Per definire la distribuzione di probabilità associata ai vari stati del sistema, si utilizza una funzione energia. La probabilità di attivazione di un neurone viene calcolata utilizzando una funzione logistica, la quale tiene conto della differenza di energia tra il caso in cui il neurone è attivo e quello in cui è inattivo.

$$P(act_u = 1) = \frac{1}{1 + e^{-\frac{\Delta E_u}{kT}}}$$

dove

$$\Delta E_u = E_{act_u=1} - E_{act_u=0} = \sum_{v \in U - \{u\}} w_{uv} act_v - \theta_u$$

Markov-chain Monte Carlo E' la procedura di aggiornamento che prevede di scegliere randomicamente un neurone e calcolare il suo differenziale energetico e, con questo, la probabilità di attivazione. Questa stessa procedura viene ripetuta varie volte fino alla convergenza del sistema

1.12.1 Training Boltzmann machines

L'obiettivo dell'apprendimento in una macchina di Boltzmann è quello di adattare i pesi delle connessioni tra i neuroni e i loro threshold in modo tale che la distribuzione di probabilità implicita nel dataset di addestramento sia approssimata dalla distribuzione rappresentata dai neuroni visibili della BM. Per fare ciò, si definisce una funzione di costo che descrive la differenza tra le due distribuzioni e si utilizza la tecnica del *gradient descent* per minimizzarla

Kullback-Leibler

$$KL(p1, p2) = \sum_{\omega \in \Omega} p1(\omega) ln \frac{p1(\omega)}{p2(\omega)}$$

dove $p1$ si riferisce alla distribuzione del dataset e $p2$ a quella della macchina di Boltzmann. Ogni passo di apprendimento viene suddiviso in due fasi

1. *Positive phase*
2. *Negative phase*

Se distinguiamo la probabilità che un neurone u sia attivato nella positive phase p_u^+ e quella che lo stesso neurone sia attivato nella negative phase p_u^- e la probabilità che due neuroni u e v siano attivati simultaneamente nella positive phase p_{uv}^+ e quella che gli stessi due neuroni siano attivati nella negative phase p_{uv}^-, possiamo definire la regola di update dei pesi e della threshold

$$\Delta w_{uv} = \frac{1}{\eta}(p_{uv}^+ - p_{uv}^-) \quad e \quad \Delta\theta_u = -\frac{1}{\eta}(p_u^+ - p_u^-)$$

Se un neurone viene sempre attivato quando viene presentato lo stesso input, il suo threshold verrà ridotto. Allo stesso modo, se due neuroni vengono spesso attivati assieme, il peso della loro connessione verrà aumentato

1.12.2 Restricted Boltzmann machines

Non ammettono connessioni intra-layer, cioè non c'è connessione tra unità visibili-visibili e tra unità nascoste-nascoste.

Le restricted Boltzmann machines sono state introdotte per ridurre il dispendio di risorse necessario per allenare le BM di dimensioni medie. La differenza principale tra una RBM e una BM classica è che il grafo del network di una RBM è bipartito. I neuroni sono essenzialmente bipartiti in neuroni visibili e nascosti. I neuroni visibili sono quelli che ricevono l'input esterno e sono completamente connessi solo con i neuroni nascosti, i neuroni nascosti prendono in input l'output dei neuroni visibili e sono completamente connessi solo con loro.

Grazie a questa struttura, il processo di apprendimento di una RBM può essere svolto in modo efficiente attraverso tre passaggi

1. Le unità di input vengono fissate rispetto ad un pattern scelto casualmente e quelle nascoste vengono aggiornate in parallelo ottenendo quello che si chiama *positive gradient*
2. Si invertono le parti, si fissano i neuroni nascosti e si aggiornano quelli visibili, ottenendo il *negative gradient*
3. Si aggiornano pesi e threshold con la differenza tra positive e negative gradient

1.13 Recurrent network

Sia gli Hopfield Network che le Boltzmann Machines sono esempi di recurrent network. In questi network, l'output viene prodotto solo quando il sistema raggiunge uno stato stabile nella sua elaborazione.

Training Questa rete può essere addestrata con una backpropagation modificata che gestisce i loop.

Use cases Rappresentazione di sistemi di equazioni differenziali.
La dinamica di questi sistemi può essere descritta attraverso l'utilizzo di *equazioni differenziali*, che vengono rappresentate in forma ricorsiva.

$$x(t_i) = x(t_{i-1}) + \Delta y_1(t_{i-1})$$

$$y_1(t_i) = y_1(t_{i-1}) + \Delta y_2(t_{i-1})$$

$$\vdots$$

$$y_{i-1}(t_i) = y_{i-1}(t_{i-1}) + f(t_{i-1}, x(t_{i-1}), \ldots, y_{n-1}(t_{i-1}))$$

Queste equazioni possono essere utilizzate per calcolare il valore successivo di una variabile in funzione della sua derivata nell'istante di tempo precedente. In questo modo, è possibile creare un network ricorrente in cui ogni variabile è associata a un nodo e le connessioni tra i nodi rappresentano il valore del differenziale.

2 Fuzzy system

2.1 Introduzione

La logica fuzzy è un approccio al calcolo basato sul "grado di verità" anziché sulla solita logica booleana vera o falsa (1 o 0) con risposte a mutua esclusione

La logica classica si basa su una visione binaria del mondo, dove ogni proposizione può essere vera o falsa. Molte proposizioni sul mondo reale non sono né vere né false. Questa visione può essere inadeguata quando si tratta di modellare concetti complessi, che non hanno definizioni precise e possono presentare sfumature e *gradi di verità* diversi. In questi casi, la logica fuzzy e la teoria degli insiemi fuzzy possono essere utilizzate per rappresentare e ragionare. È importante notare che l'imprecisione non deve essere confusa con l'incertezza, che si riferisce alla possibilità che un evento si verifichi o meno espressa attraverso la probabilità. La probabilità rimane un fenomeno booleano, mentre l'appartenenza fuzzy si riferisce a quanto un oggetto soddisfa una determinata proprietà.

Esempio Ad esempio, una persona non avrà problemi ad accelerare lentamente mentre avvia una macchina, se gli viene chiesto di farlo. Se vogliamo automatizzare questa azione, non sarà affatto chiaro come tradurre questo consiglio in un'azione di controllo ben definita. È necessario determinare una dichiarazione concreta basata su un valore non ambiguo, ad esempio "premi l'acceleratore alla velocità di mezzo centimetro al secondo". D'altra parte, questo tipo di informazione non sarà adeguata o molto utile per una persona. In questo caso, un problema principale sarà la traduzione della descrizione verbale in valori concreti, ad esempio assegnando "premi l'acceleratore lentamente" a "premi l'acceleratore alla velocità di un centimetro al secondo".

Fuzzy set A differenza degli insiemi tradizionali, in cui un elemento può o non può appartenere all'insieme, in un insieme fuzzy ogni elemento ha un grado di appartenenza compreso tra 0 e 1. Questo grado di appartenenza rappresenta il livello di membership o la probabilità di appartenenza dell'elemento all'insieme

Definizione 1 *Dato un dominio del discorso X, un insieme fuzzy μ è una funzione $\mu : X \to [0, 1]$ che assegna ad ogni elemento un grado di appartenenza $\mu(x)$ rispetto all'insieme μ.*

I fuzzy set rappresentano un'interfaccia tra il linguaggio naturale e le sue corrispondenti rappresentazioni numeriche. E' possibile associare un valore numerico ad un'espressione linguistica che descrive un concetto sfumato o non preciso, in modo da poter utilizzare tali valori per l'analisi e la manipolazione dei dati.

Ad esempio, vogliamo dare un modello formale alla proprietà "essere alto per un bambino di 4 anni". Per farlo definiremo un insieme fuzzy μ_{tall} attraverso una funzione sigmoide come in figura, tale per cui risulteranno sicuramente

nell'estensione della proprietà i bambini più alti di $1.5m$ centimetri e sicuramente fuori dall'estensione quelli più bassi di $0.7m$. Tutti gli altri apparterranno all'insieme con un certo grado.

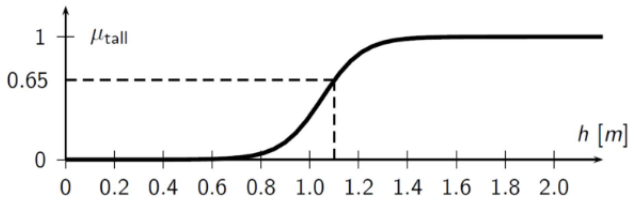


Figura 16: L'insieme fuzzy μ_{tall} che descrive il predicato "essere alto per un bambino di 4 anni"

Semantiche della funzione di appartenenza Esistono diverse interpretazioni possibili per la funzione di appartenenza fuzzy, a seconda dell'applicazione in cui viene utilizzata. In particolare

1. *Somiglianza*, la funzione di appartenenza viene interpretata come il *grado di prossimità* di un elemento rispetto ad un prototipo, e viene utilizzata in problemi di classification, clustering e regression

2. *Preferenza*, negli insiemi fuzzy, la preferenza viene rappresentata attraverso la funzione di appartenenza, che indica il grado di soddisfazione o di preferenza per un determinato oggetto o un dato valore. La funzione di appartenenza viene interpretata come l'intensità della preferenza associata all'oggetto. Più il grado di appartenenza di un oggetto a un insieme fuzzy è elevato, maggiore sarà l'intensità della preferenza per quell'oggetto. Il grado di appartenenza, nell'ambito della preferenza, fornisce un modo per quantificare e misurare l'intensità o la forza della preferenza per gli oggetti o le decisioni prese

3. *Possibilità*, la funzione di appartenenza viene interpretata come il *grado di possibilità* che un elemento sia il valore di un parametro X. Questa interpretazione viene utilizzata per quantificare lo stato epistemico di un agente, distinguendo ciò che è "sorprendente" da ciò che è "tipico" o "aspettato"

Funzioni triangolari e trapezoidali Di solito gli insiemi fuzzy vengono utilizzati per modellare espressioni, a volte chiamate anche espressioni linguistiche, al fine di sottolineare la relazione con il linguaggio naturale, ad esempio "circa 3", "di altezza media" o "molto alto", che descrivono un valore impreciso o un intervallo impreciso. Gli insiemi fuzzy associati a tali espressioni dovrebbero aumentare in modo monotono fino a un certo valore e diminuire in modo monotono da tale valore. Tali insiemi fuzzy sono chiamati *convessi*. Le funzioni che rappresentano insiemi convessi sono dette *funzioni triangolari*, rappresentate

come

$$\Lambda_{a,b,c} : \mathbb{R} \to [0,1], \quad x \mapsto \begin{cases} \frac{x-a}{b-a} \text{ if } a \leq x < b \\ \frac{c-x}{c-b} \text{ if } b \leq x \leq c \\ 0 \quad \text{altrimenti} \end{cases}$$

Le funzioni triangolari possono essere considerate un caso particolare delle *funzioni trapezoidali*

$$\Pi_{a,b,c,d} : \mathbb{R} \to [0,1], \quad x \mapsto \begin{cases} \frac{x-a}{b-a} \text{ if } a \leq x \leq b \\ 1 \quad \text{if } b \leq x \leq c \\ \frac{d-x}{d-c} \text{ if } c \leq x \leq d \\ 0 \quad \text{altrimenti} \end{cases}$$

Definizione 2 *Sia μ un fuzzy set definito rispetto al dominio del discorso X e sia $\alpha \in [0,1]$. L'insieme*

$$[\mu]_\alpha = \{x \in X \mid \mu(x) \geq \alpha\}$$

è chiamato α-cut o α level set dell'insieme μ.

Vertical view La rappresentazione di un insieme fuzzy come una funzione dall'universo del discorso all'intervallo unitario, che assegna un grado di appartenenza a ciascun elemento, viene chiamata visione verticale. Il fuzzy set può essere visto come un inviluppo superiore dei suoi alpha cut

Horizontal view Un'altra possibilità per descrivere un insieme fuzzy è la visione orizzontale. Per ogni valore α dell'intervallo unitario, consideriamo l'insieme degli elementi che hanno un grado di appartenenza almeno α all'insieme fuzzy

Gli insiemi di livello di un insieme fuzzy hanno l'importante proprietà di caratterizzare univocamente l'insieme fuzzy stesso. Quando conosciamo gli insiemi di livello $[\mu]\alpha$ di un insieme fuzzy μ per tutti gli $\alpha \in [0,1]$, possiamo determinare il grado di appartenenza $\mu(x)$ di qualsiasi elemento x a μ tramite l'equazione

$$\mu(x) = \sup_{\alpha \in [0,1]} \{x \in [\mu]_\alpha\}$$

2.1.1 Definizioni alpha-cut

Avendo introdotto gli alpha-cut come uno strumento per rappresentare i fuzzy set, ora li sfrutteremo definendo alcuni concetti molto utili

Supporto Il supporto $S(\mu)$ di un insieme fuzzy μ è l'insieme booleano che contiene tutti e soli gli elementi del dominio del discorso che hanno un grado di appartenenza non nullo rispetto a μ. Tutti gli elementi del dominio

$$S(\mu) = [\mu]_{\bar{0}} = \{x \in X | \mu(x) > 0\}$$

Centro Il centro $C(\mu)$ di un insieme fuzzy μ è l'insieme booleano che contiene tutti e soli gli elementi del dominio del discorso che hanno un grado di appartenenza uguale a 1 rispetto a μ

$$C(\mu) = [\mu]_1 = \{x \in X | \mu(x) = 1\}$$

Altezza L'altezza $h(\mu)$ di un insieme fuzzy μ è il più alto grado di appartenenza ottenibile da un elemento di μ

$$h(\mu) = \sup_{x \in X} \{\mu(x)\}$$

Un insieme fuzzy μ è definito normale sse $h(\mu) = 1$, altrimenti sub-normale

Convesso Un insieme fuzzy μ è definito convesso[8] sse i suoi alpha-cut/level set sono convessi per ogni scelta di $\alpha \in [0, 1)$

Numero fuzzy Un insieme fuzzy μ è un numero fuzzy sse μ è normale e $[\mu]\alpha$ è chiusa, limitata e convessa per ogni scelta di $\alpha \in [0, 1)$. Va a generalizzare l'aritmetica che si fa con i numeri reali

2.2 Logica fuzzy

La logica classica e la teoria degli insiemi finiti sono equivalenti e possono essere rappresentate tramite un'algebra booleana finita. Gli operatori logici di congiunzione, disgiunzione e negazione possono essere utilizzati per definire gli operatori insiemistici. Nella logica fuzzy, l'intervallo reale $[0, 1]$ rappresenta l'insieme dei valori di verità e gli operatori logici booleani vengono adattati per aderire alla nuova semantica. Quindi, è possibile costruire una teoria degli operatori insiemistici "fuzzy" utilizzando gli stessi operatori logici della logica classica.

I connettivi logici più importanti sono il logico AND $\wedge$ (congiunzione), il logico OR $\vee$ (disgiunzione), la negazione NOT $\neg$ e l'IMPLICAZIONE $\rightarrow$. Le funzioni di verità più comunemente usate per la congiunzione e la disgiunzione nella logica fuzzy sono il minimo o il massimo. Ciò significa che

1. $\neg\mu \doteq 1 - \mu(x)$
2. $\mu \wedge \mu' \doteq \min\{\mu(x), \mu'(x)\}$
3. $\mu \vee \mu' \doteq \max\{\mu(x), \mu'(x)\}$

[8] Un insieme convesso è un insieme in cui, presi due punti qualsiasi all'interno dell'insieme, la linea che li congiunge è completamente contenuta all'interno dell'insieme stesso. In altre parole, tutti i punti sulla linea che collega due punti dell'insieme sono anch'essi contenuti nell'insieme

Negazione stretta e forte Esistono vari modi di definire la negazione in una logica fuzzy. L'unico requisito è che la definizione rispetti tre proprietà che, intuitivamente, ogni negazione deve possedere:

1. $\neg(0) = 1$
2. $\neg(1) = 0$
3. $x \leq y \implies \neg x \geq \neg y$

In aggiunta a queste proprietà, una negazione può soddisfarne altre. Per esempio, si può chiedere che sia *strettamente decrescente*

$$x < y \implies \neg x > \neg y$$

Che sia *continua*, se soddisfa la proprietà di continuità, che afferma che piccole variazioni nei valori di verità delle proposizioni corrispondono a piccole variazioni nei valori di verità delle loro negazioni. In altre parole, se due proposizioni fuzzy sono molto simili, le loro negazioni dovrebbero essere anche molto simili.

O *involutiva*:

$$\neg\neg x = x$$

Definizione 3 *Una negazione si dice stretta sse è strettamente decrescente e continua*

Definizione 4 *Una negazione si dice forte sse è stretta e involutiva*

T-norme e t-conorme Le t-norme (triangular norms) e le t-conorme (triangular conorms) sono utilizzate nella logica fuzzy per definire le operazioni di congiunzione e disgiunzione tra le proposizioni fuzzy.

Le t-norme sono funzioni che prendono in input due valori di verità (compresi tra 0 e 1) e restituiscono un valore di verità che rappresenta la congiunzione delle due proposizioni fuzzy. Una funzione $\top : [0,1]^2 \to [0,1]$ si dice *t-norma* sse soddisfa le seguenti proprietà:

1. Commutativa: $\top(x, y) = \top(y, x)$
2. Associativa: $\top(x, \top(y, z)) = \top(\top(x, y), z)$
3. Monotonicità: $x \leq z \implies \top(x, y) \leq \top(x, z)$
   cioè il valore di verità della congiunzione $x \wedge z$ non dovrebbe essere inferiore al valore di verità della congiunzione $x \wedge y$, se x ha un valore di verità inferiore a z.
4. Identità: $\top(x, 1) = x$
   richiediamo che il valore di verità di una proposizione x sia lo stesso del valore di verità della congiunzione di x con qualsiasi proposizione vera y.

Nel contesto della logica fuzzy, è conveniente utilizzare una t-norma come funzione di verità per la congiunzione. La proprietà 4 implica che per ogni t-norma t, abbiamo $t(1,1) = 1$ e $t(0,1) = 0$. Inoltre, grazie alla proprietà 1, otteniamo $t(1,0) = 0$ a partire da $t(0,1) = 0$. Infine, grazie alla proprietà di monotonicità 3 e $t(0,1) = 0$, otteniamo $t(0,0) = 0$. Pertanto, ogni t-norma, quando limitata ai valori 0 e 1, coincide con la funzione di verità della congiunzione usuale data dalla tabella di verità.

Le t-conorme sono funzioni che prendono in input due valori di verità e restituiscono un valore di verità che rappresenta la disgiunzione delle due proposizioni fuzzy. Una funzione $\perp : [0,1]^2 \to [0,1]$ si dice *t-conorma* sse soddisfa le seguenti proprietà:

1. Commutativa: $\perp(x,y) = \perp(y,x)$
2. Associativa: $\perp(x, \perp(y,z)) = \perp(\perp(x,y), z)$
3. Monotonicità: $x \leq z \implies \perp(x,y) \leq \perp(x,z)$
4. Identità: $\perp(x,0) = x$
   ciò significa che il valore di verità di una proposizione x sarà lo stesso del valore di verità della disgiunzione di y con una proposizione falsa x

Implicazione fuzzy Nell'ambito della logica fuzzy, le regole fuzzi sono state usate per definire come le variabili di input influenzano quelle di output. Ogni regola è composta da due parti: l'antecedente (if) e il conseguente (then). Le implicazioni sono operatori matematici che collegano l'antecedente al conseguente.

Come nel caso booleano avremo che un insieme fuzzy è contenuto in un altro se tutti gli elementi del primo sono contenuti nel secondo. Sfruttando l'isomorfismo tra operatori logici e insiemistici, inoltre potremo definire il concetto di sottoinsieme a partire da quello di implicazione come segue

$$I(a,b) = \neg a \vee b$$

A seconda della semantica che daremo ai nostri operatori logici fuzzy avremo varie classi di implicazioni:

1. *S-implication*: $I(a,b) = \perp(\neg a, b)$
2. *Residuated implication*: $I(a,b) = \sup\{x \in [0,1] | \top(a,x) \leq b\}$, basato sul concetto di residuo nella teoria delle reti di Kleene
3. *Quantum Logic-implication*: $I(a,b) = \perp(\neg a, \top(a,b))$, meccanica quantistica

Linguistic Modifiers Normalmente, un fuzzy set rappresenta un concetto impreciso come "velocità elevata", "giovane" o "alto". Da tali concetti possiamo derivare altri concetti imprecisi applicando dei modificatori linguistici (linguistic hedges) come "molto" o "più o meno".

Una velocità molto alta è anche una velocità elevata, ma non viceversa. Pertanto, il grado di appartenenza di una specifica velocità v al fuzzy set μ_{vhv} non dovrebbe superare il suo grado di appartenenza al fuzzy set μ_{hv}. Possiamo ottenere ciò interpretando il modificatore linguistico "molto", simile alla negazione, come un operatore unario e assegnando una funzione di verità adeguata ad esso, ad esempio $w_{very(\alpha)} = \alpha^2$. In questo modo otteniamo $\mu_{vhv(x)} = (\mu_{hv(x)})^2$. Ora, una velocità che è in misura di 1 una velocità elevata è anche una velocità molto alta. Una velocità che non è una velocità elevata (grado di appartenenza 0) non è nemmeno una velocità molto alta. Se il grado di appartenenza di una velocità a μ_{hv} è compreso tra 0 e 1, essa è anche una velocità molto alta, ma con un grado di appartenenza inferiore.

Nello stesso modo possiamo assegnare una funzione di verità al modificatore "più o meno". Questa funzione di verità dovrebbe aumentare il grado di appartenenza, ad esempio $w_{moreorless(\alpha)} = \sqrt{\alpha}$.

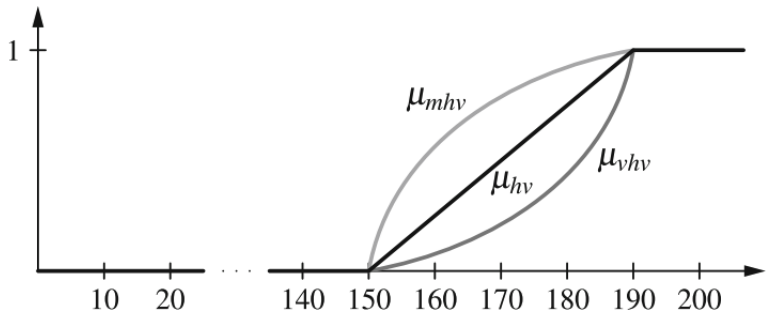


Figura 17: I fuzzy set μ_{hv}, μ_{vhv} e μ_{mhv} di velocità elevate, molto elevate e più o meno elevate

2.3 Teoria della logica fuzzy

Principio di estensione Permette di generalizzare una qualsiasi funzione G da insiemi classici ad insiemi fuzzy

Un insieme fuzzy è caratterizzato da gradi di appartenenza, che indicano in che misura gli elementi dell'universo del discorso appartengono all'insieme fuzzy. Il principio di estensione ci fornisce una regola per calcolare il grado di verità degli elementi dell'insieme immagine basandoci sui gradi di verità degli elementi originali dell'insieme fuzzy

Il principio di estensione si basa sull'idea fondamentale di associare a ciascun elemento dell'insieme immagine un grado di appartenenza che rappresenta la massima appartenenza possibile tra tutti gli elementi dell'insieme di partenza che vengono mappati su quell'elemento specifico

In altre parole, il principio di estensione stabilisce che il grado di appartenenza di un elemento all'immagine[9] di un insieme fuzzy è determinato dalla massima

[9]L'immagine di un insieme attraverso una funzione è un concetto che descrive l'insieme dei

appartenenza tra tutti gli elementi dell'insieme di partenza che sono mappati su quell'elemento specifico dell'insieme di arrivo. Il principio di estensione, nel contesto degli insiemi fuzzy, ci consente di estendere il concetto di immagine anche agli insiemi fuzzy.

Alcuni insiemi fuzzy rilevanti Esistono diversi tipi di insiemi fuzzy, ma quelli definiti sull'insieme dei numeri reali sono particolarmente importanti perché hanno un significato quantitativo che può essere utilizzato per rappresentare variabili fuzzy. Queste variabili sono fondamentali in molte applicazioni come il controllo fuzzy, il ragionamento approssimato e l'ottimizzazione. In letteratura, ci sono alcune classi di $F(\mathbb{R})$, l'insieme degli insiemi fuzzy sui reali, che sono spesso citate, tra cui:

1. Normal fuzzy set:
   $F_N(\mathbb{R}) = \{\mu \in F(\mathbb{R}) | \exists x \in \mathbb{R} : \mu(x) = 1\}$
2. Upper Semi-continuous fuzzy set:
   $F_C(\mathbb{R}) = \{\mu \in F_N(\mathbb{R}) | \forall \alpha \in (0, 1] : [\mu]_\alpha \text{ è compatto } \}$
3. Fuzzy interval:
   $F_I(\mathbb{R}) = \{\mu \in F_N(\mathbb{R}) | \forall a, b, c \in \mathbb{R} : c \in [a, b] \implies \mu(c) \geq \min\{\mu(a), \mu(b)\}\}$

Particolare interesse rivestono i *fuzzy interval* anche detti *fuzzy numbers* perché permettono di definire variabili fuzzy quantitative. Queste variabili assumono come valore numeri fuzzy. Quando le quantità fuzzy rappresentano concetti linguistici (come piccolo, grande, etc.) si parla di variabili linguistiche. Ogni variabile linguistica è definita da un quintupla *(v, T, X, g, m)*, dove v è il nome della variabile, T è l'insieme dei termini che coprono v, X è il dominio del discorso, gè la grammatica per generare i termini ed m la semantica che assegna ad ogni termine un fuzzy number. Per processare questo genere di variabili occorrerà estendere le operazioni insiemistiche e aritmetiche originalmente utilizzate per i numeri.

Rappresentazione per insiemi Abbiamo visto come definire le operazioni aritmetiche sui fuzzy set nell'ambito di $F(\mathbb{R})$ utilizzando il principio di estensione. Tuttavia, calcolare direttamente queste funzioni sugli insiemi fuzzy può essere molto costoso, specialmente se si utilizza la rappresentazione verticale invece di quella orizzontale. Pertanto, sarebbe utile ridurre l'aritmetica fuzzy all'ordinaria aritmetica sugli insiemi booleani e applicare alcune semplici operazioni sugli intervalli per ottenere il risultato. Questo è possibile utilizzando la rappresentazione per insiemi di un insieme fuzzy.

---

valori che la funzione assume quando viene applicata agli elementi dell'insieme di partenza. Più precisamente, data una funzione $f : X \to Y$, dove X è l'insieme di partenza e Y è l'insieme di arrivo, l'immagine di un insieme $A \subseteq X$, indicata come $f[A]$, è l'insieme dei valori che la funzione f assume quando viene applicata agli elementi di A. In altre parole, $f[A] = y \in Y | \exists x \in A : f(x) = y$. Può aiutarci a studiare la relazione tra gli elementi di due insiemi e a identificare proprietà importanti della funzione stessa

Definizione 5 *Una famiglia* $(A_\alpha)_{\alpha\in(0,1)}$ *è una rappresentazione per insiemi di* $\mu \in F_N(\mathbb{R})$ *se*

1. $0 < \alpha < \beta < 1 \implies A_\alpha \subseteq A_\beta \subseteq \mathbb{R}$
2. $\mu(t) = \sup\{\alpha \in [0,1] | t \in A_\alpha\}$

Definizione 6 *Sia* $\mu \in F_N(\mathbb{R})$. *La famiglia* $(A_\alpha)_{\alpha\in(0,1)}$ *è una rappresentazione per insiemi di* μ *sse*

$$[\mu]_{\bar{\alpha}} = \{t \in \mathbb{R} | \mu(t) > \alpha\} \subseteq A_\alpha \subseteq \{t \in \mathbb{R} | \mu(t) \geq \alpha\} = [\mu]_\alpha$$

è valida per ogni $\alpha \in (0,1)$.

Questa definizione ci assicura che una rappresentazione per insiemi è una fedele immagine dell'insieme fuzzy che raffigura.

Relazioni fuzzy Una relazione booleana tra insiemi $X_1, X_2, \ldots, X_n$ rappresenta un sottoinsieme del prodotto cartesiano di questi insiemi. In altre parole, ogni possibile combinazione di elementi degli insiemi è associata a un valore booleano di vero o falso, a seconda che la combinazione appartenga o meno alla relazione. Questa relazione può essere descritta attraverso una funzione chiamata *funzione caratteristica*, che associa ogni combinazione di elementi ad un valore booleano

$$R(x_1, \ldots, x_n) = \begin{cases} 1 \text{ se e solo se } (x_1, \ldots, x_n) \in R \\ 0 \text{ altrimenti} \end{cases}$$

Nel contesto delle relazioni fuzzy, la funzione caratteristica può essere estesa per includere valori fuzzy che indicano il grado di forza della relazione tra i membri di una tupla. Consideriamo n insiemi fuzzy $A_1, \ldots, A_n$ definiti su n domini del discorso $X1, \ldots, Xn$. Il prodotto cartesiano degli insiemi fuzzy $A_1 \times \cdots \times A_n$ è una relazione fuzzy nello spazio prodotto $X_1 \times \cdots \times X_n$. Questa relazione è definita attraverso la funzione di partecipazione

$$\mu_{A_1 \times \cdots \times A_n}(x_1, \ldots, x_n) = \top(\mu_{A_1}(x_1), \ldots, \mu_{A_n}(x_n))$$

Dove $\top$ è una t-norma, solitamente il minimo o il prodotto.

Definizione 7 *Siano* $w = (x_1, \ldots, x_n)$ *e* $v = (y_1, \ldots, y_m)$ *due tuple.* w *è chiamato sottosequenza di* v *(in simboli,* $w \prec v$*) sse* $\forall j \in \{1, \ldots, n\}, w_j = v_j$

Definizione 8 *Data un relazione* $R(x_1, \ldots, x_n)$ *e un sottoinsieme dei domini del discorso* $Y \subseteq \{X_1, \ldots, X_n\}$, *denotiamo con* $[R \downarrow Y]$ *la proiezione di* R *su* Y *definita come:*

$$[R \downarrow Y](v) = \max_{v \prec w} R(w)$$

Definizione 9 *Data una proiezione* $[R \downarrow Y]$, *una estensione cilindrica che denotiamo come* $[R \uparrow X - Y]$ *è la relazione* R *di partenza salvo che ogni valore diverso da quello della proiezione viene sosituito con quello stesso valore*

$$[R \uparrow X - Y](v) = R(w)$$

Relazioni binarie Le relazioni binarie, che sono una generalizzazione delle funzioni matematiche, sono di particolare interesse tra le relazioni n-dimensionali. A differenza delle funzioni, una relazione $R(X, Y)$ può associare un elemento di X a uno o più elementi di Y. Possiamo estendere le operazioni tipiche sulle funzioni, come la composizione e l'inversa, anche alle relazioni fuzzy.

Definizione 10 *Data una relazione $R(X, Y)$ il suo dominio, denotato $domR$, è definito come*

$$domR(x) = \max_{y \in Y}\{R(x, y)\}$$

Definizione 11 *Data una relazione $R(X, Y)$ il suo codominio, denotato $ranR$, è definito come*

$$ranR(x) = \max_{x \in X}\{R(x, y)\}$$

Definizione 12 *Data una relazione $R(X, Y)$ la sua altezza, denotata hR, è definito come*

$$hR(x) = \max_{x \in X} \max_{y \in Y}\{R(x, y)\}$$

Definizione 13 *Data una relazione $R(X, Y)$ la sua inversa, denotata R^{-1}, è definito come quella relazione su $Y \times X$ tale che*

$$R^{-1}(y, x) = R(x, y) \quad \forall x \in X, \forall y \in Y$$

Definizione 14 *Data una relazione $R(X, Y)$ e una relazione $Q(Y, Z)$ la loro composta, denotata $R \circ Q(X, Z)$, è definito come quella relazione su $X \times Z$ tale che*

$$R \circ Q(x, z) = \sup_{y \in Y}\{\min\{P(x, y), Q(y, z)\} \quad \forall x \in X, \forall z \in Z\}$$

Definizione 15 *Data una relazione $R(X, Y)$ e una relazione $Q(Y, Z)$ il loro join, denotata $R \star Q(X, Y, Z)$, è definito come quella relazione su $X \times Y \times Z$ tale che*

$$R \star Q(x, y, z) = \min\{P(x, y), Q(y, z)\} \quad \forall x \in X, \forall y \in Y, \forall z \in Z$$

Definizione 16 *Data una relazione $R(X, X)$ si definisce una relazione di equivalenza sse soddisfa le seguenti proprietà*

1. *riflessività:* $\forall x \in X \quad R(x, x) = 1$
2. *simmetria:* $\forall x, y \in X \quad R(x, y) = R(y, x)$
3. *transitività:* $\forall (x, z) \in X^2 \quad R(x, z) \geq \max_{y \in X} \min\{R(x, y), R(y, z)\}$

2.4 Fuzzy controller

Si basa sulla definizione di transizioni non-lineari tra gli stati di un sistema, evitando di specificare un insieme di equazioni differenziali per ogni variabile. Ciò consente di modellare sistemi complessi che altrimenti sarebbero difficili da analizzare in modo preciso attraverso metodi matematici tradizionali. Il fuzzy control sfrutta i concetti del fuzzy logic e del fuzzy set per costruire un controllore che possa gestire situazioni incerte o ambigue

Un sistema di controllo fuzzy è basato su regole ed è costituito da tre fasi: fuzzificazione, inferenza e de-fuzzificazione. La prima consiste nel generare gli insiemi fuzzy necessari, poi si applicano delle regole. Infine l'insieme fuzzy viene tradotto in una opportuna quantità fisica in modo da agire sul sistema da controllare

- *fuzzyfication interface*, riceve gli input e si occupa di convertirli in un dominio adeguato (termini linguistici o fuzzy set)
- *knowledge base*, consiste di dati che contengono informazioni riguardo intervalli, trasformazioni di dominio e a quali insiemi fuzzy corrisponderanno i termini linguistici; e regole che contengono i controlli del tipo if-then
- *decision logic*, rappresenta l'unità processore, si occupa di computare l'output in base all'input e alla knowledge base
- *defuzzification interface*, si occupa di mappare i valori fuzzy in valori booleani che sono inviati come segnali al controllo del sistema

Defuzzification La mappatura dei segnali fuzzy interni al controller in segnali booleani utili a controllare il sistema può essere operata in svariati modi. I più comuni sono:

1. *MCM: Max Criterion Method*, sceglie un valore arbitrario y per raggiungere il massimo valore di appartenenza
2. *MOM: Mean of Maxima*, prende Y come intervallo e ne calcola l'insieme Y_{MAX} tale che l'output in quei punti è massimo. Il valore di output sarà calcolato come la media su Y_{MAX}
3. *COG: Center of Gravity*, come il MOM, preso Y come un intervallo restituisce in output il centro dell'area

Mamdani controller Il Mamdani controller è il primo tipo di fuzzy controller sviluppato da Mamdani e Assilian nel 1975 per controllare una combinazione di motore a vapore e caldaia da un insieme di regole di controllo linguistico.

Questo controller utilizza una serie di regole del tipo "if X is M_n, then Y is N_m", dove M_n e N_m sono intervalli che rappresentano termini linguistici. Queste regole definiscono parzialmente una funzione e possono essere rappresentate

come un'unione di intervalli nello spazio S

$$S = \cup M_i \times N_i$$

Takagi–Sugeno controller Il Sugeno controller è una modifica del Mamdani controller. Anche in questo caso i valori di input nelle regole sono descritti da insiemi fuzzy. Tuttavia, utilizzando un modello TS, il conseguente di una singola regola non è più un insieme fuzzy, ma determina una funzione con le variabili di input come argomenti. Ad esempio una funzione lineare

$$R: \text{ if } x_1 \text{ is } \mu_1 \text{ and } \ldots \text{ and } x_n \text{ is } \mu_n, \text{ then } y = f(x_1, \ldots, x_n)$$

Questa funzione viene considerata una buona funzione di controllo per la regione descritta dall'antecedente. Per evitare sovrapposizioni tra le varie regioni descritte nell'antecedente delle regole, si cerca di mantenere la leggibilità del modello. Inoltre, poiché l'output è già crisp[10], non è necessario defuzzificarlo[11]

Similarity-based reasoning Questa tipologia di controller utilizza il concetto di *relazione di somiglianza*, l'analogo fuzzy delle relazioni di equivalenza.

Definizione 17 *Una funzione $E : X^2 \to [0, 1]$ è definita relazione di somiglianza rispetto ad una T-norma se e solo se soddisfa le seguenti condizioni:*

1. $E(x, x) = 1$
2. $E(x, y) = E(y, x)$
3. $\top(E(x, y), E(y, z)) = E(x, z)$

Questo genere di relazioni viene utilizzato per tradurre l'informazione degli esperti in modo che le varie tuple coprano tutti i possibili comportamenti del sistema. Dalle classi di somiglianza possiamo poi estrarre regole in tutto uguali a quelle per il Mamdani controller

2.5 Fuzzy data analysis

Il termine "fuzzy data analysis" può essere interpretato in due modi distinti. Nel primo caso, si riferisce all'utilizzo di tecniche di ragionamento fuzzy per analizzare dati non fuzzy o crisp, ad esempio tramite l'applicazione di algoritmi di clustering. In questo contesto, l'obiettivo è quello di gestire l'incertezza e l'imprecisione presenti nei dati non fuzzy.

Nel secondo caso, "fuzzy data analysis" si riferisce all'analisi di dati rappresentati come fuzzy set, ovvero insiemi fuzzy. Questa forma di analisi coinvolge l'elaborazione di dati che sono intrinsecamente incerti o parzialmente definiti, come ad esempio i random set e le random fuzzy variables.

[10] L'output del controller è un valore esatto e non un insieme fuzzy

[11] La defuzzificazione, in generale, è il processo di convertire un insieme fuzzy in un valore crisp, rappresentativo dell'output del sistema di controllo

2.5.1 Fuzzy clustering

Il fuzzy clustering è un metodo di apprendimento non supervisionato che suddivide un dataset in gruppi (cluster) in modo che gli oggetti all'interno di uno stesso cluster siano il più simili possibile tra loro, quelli in cluster diversi siano il più possibile dissimili. La similitudine è misurata in termini di una funzione di distanza. Nel caso dell'algoritmo *hard c-means*, si sceglie un numero di cluster e si procede all'assegnamento dei punti ai rispettivi cluster. Successivamente, si aggiornano le posizioni dei centri in base ai punti assegnati. L'algoritmo viene ripetuto fino a quando la posizione dei centri si stabilizza. Il problema di questo approccio è che la partizione è *crisp* e quindi non è possibile esprimere la partecipazione di un punto a più di un cluster.

Il fuzzy clustering risolve questo problema introducendo un concetto di appartenenza continuo in $[0, 1]$, che consente di esprimere l'appartenenza di un punto a più di un cluster. Il risultato sarà una partizione del dataset in fuzzy set, rappresentata da una matrice che assegna ad ogni componente u_{ij} il grado di appartenenza del punto x_j al fuzzy set Γ_i, in simboli $u_{ij} = \mu_{\Gamma_i}(x_j)$

Esistono due tipi di fuzzy clustering: quello probabilistico e quello possibilistico, a seconda delle condizioni imposte alla funzione di appartenenza.

Caso probabilistico

1. $\sum_{j=1}^{n} u_{ij} > 0, \quad \forall i \in \{1, \ldots, c\}$, ogni cluster con almeno un elemento
2. $\sum_{i=1}^{c} u_{ij} = 1, \quad \forall j \in \{1, \ldots, n\}$, grado di appartenenza totale uguale a 1

La prima condizione significa che ogni cluster deve avere almeno un elemento. La seconda condizione indica che l'appartenenza di ogni elemento del dataset deve essere espressa attraverso la somma dei gradi di appartenenza a tutti i fuzzy set che costituiscono la partizione, in modo tale che il grado di appartenenza totale sia uguale a 1.

Caso possibilistico Nel caso possibilistico si mantiene solo la prima assunzione. L'interpretazione possibilistica è da preferire quando si ha a che fare con dati rumorosi, con molti outlier

Problemi Per valutare se la partizione in cluster prodotta dal nostro algoritmo è accurata e riflette l'informazione implicita nei dati, dobbiamo utilizzare una *misura di qualità* del clustering. Ci sono diversi criteri che possiamo utilizzare per questo scopo, tra cui una chiara separazione tra i cluster, un volume minimo per ciascun cluster e un massimo numero di punti concentrati vicino al centro. L'obiettivo è trovare il numero ottimale di cluster per il dataset in esame

1. *Partition coefficient*: $PC = \frac{1}{n} \sum_{i=1}^{c} \sum_{j=1}^{n} u_{ij}^2$

2. *Average partition density*: $APD = \frac{1}{c}\sum_{i=1}^{c}\frac{\sum_{j\in Y_i} u_{ij}}{\sqrt{|\sum_i|}}$

3. *Partition entropy*: $PE = \sum_{i=1}^{c}\sum_{j=1}^{n} u_{ij}\log u_{ij}$

Varianti La distanza euclidea è la misura di distanza più comune utilizzata nell'algoritmo di clustering, ma può essere limitante perché permette solo la formazione di cluster sferici. Per ovviare a questo problema, sono state proposte alcune varianti.

I) *Gustafson-Kessel*, la distanza euclidea viene sostituita con quella di Mahalanobis definita rispetto ad un cluster Γ_i. Questa misura di distanza considera anche la covarianza degli attributi, rendendo possibile la formazione di cluster ellittici

$$d^2(x_j, C_j) = (x_j - c_i)^T \sum_{i}^{-1}(x_j - c_i)$$

Dove $\sum_i$ è la matrice covariante. Questo permette di rilassare i vincoli sulla forma dei cluster e di estrarre maggiori informazioni rispetto all'algoritmo standard. Tuttavia, l'algoritmo è più sensibile ad una corretta inizializzazione e può essere utile eseguire alcune iterazioni dell'algoritmo standard per decidere una buona inizializzazione. E' più costoso in termini computazionali rispetto a quello standard e può essere difficile da applicare a dataset di grandi dimensioni a causa dell'inversione della matrice covariante.

II) *Algoritmi di shell clustering*, permettono cluster di forma non convessa. Sono particolarmente utili per il riconoscimento di immagini e la loro analisi

III) *Kernel-based clustering*, è utile qualora si abbia a che fare con dati non-vettoriali come sequenze, alberi o grafi. Questo metodo si basa su una mappa $\phi : \chi \to \mathbb{H}$, dove $\mathbb{H}$ è uno spazio di Hilbert e χ è lo spazio degli input. La funzione kernel k definisce la relazione di somiglianza tra i dati mappati, senza utilizzare prototipi per i singoli cluster.

$$k : \chi \times \chi \to \mathbb{R}, \forall x, x' \in \chi :< \phi(x), \phi(x') >= k(x, x')$$

I centri dei cluster sono combinazioni lineari dei dati mappati in $\mathbb{H}$.

$$c_i^{\phi} = \sum_{r=1}^{n} a_{ir}\phi(x_r)$$

Uno svantaggio di questo approccio è la difficoltà nella scelta di una funzione kernel adeguata e dei parametri ottimali. Inoltre, non c'è una rappresentazione esplicita dei singoli cluster, il che può rendere difficile l'interpretazione dei risultati ottenuti.

IV) *Noise clustering*, aggiungono un cluster speciale, chiamato outlier cluster c, che rappresenta tutti quei dati che non possono essere assegnati ad

alcun altro cluster a causa del rumore. Il centro di questo cluster è scelto in modo tale che abbia una distanza costante da tutti i punti del dataset. In questo modo, gli outlier possono essere facilmente identificati e gestiti separatamente dagli altri dati durante l'analisi.

2.5.2 Random set

Nella statistica tradizionale, l'analisi dei dati si basa sul concetto di variabili casuali, ovvero funzioni che mappano uno spazio di probabilità ad un insieme U (spesso $\mathbb{R}$). Si cerca di estendere questi concetti per poter gestire descrizioni di dati fuzzy. Un random set è una generalizzazione del concetto di variabile casuale: anziché assegnare un valore a un dato, un random set assegna un sottoinsieme dell'insieme di valori possibili. In questo modo, un dato viene descritto in termini di una serie di possibili valori. Questo permette di modellare meglio la natura incerta dei dati in alcuni contesti.

Un'estensione naturale che consente l'imprecisione sono i cosiddetti insiemi casuali, che sono variabili casuali a valore insiemistico e possono quindi essere mappati, ad esempio, su sottoinsiemi di un insieme finito o su intervalli dei numeri reali. Mentre nel caso di una variabile casuale $X : \Omega \to A$ il risultato di un esperimento è un elemento dell'insieme A, nel caso di un insieme casuale $X : \Omega \to 2^A$ il risultato di un esperimento è un sottoinsieme di A, cioè un elemento dell'insieme delle parti di A.

Consideriamo il seguente insieme di lingue L = Inglese, Tedesco, Francese, Spagnolo. Supponiamo di selezionare casualmente una persona ω dall'insieme Ω degli impiegati di un gruppo di lavoro e chiediamo quali lingue l'impiegato sa parlare. Possiamo modellare l'esito di questo esperimento mediante una funzione a valore insiemistico $\Gamma : \Omega \to 2^{L} - \emptyset$, dove $\Gamma(\omega) \subseteq L$ è l'insieme delle lingue che la persona ω sa parlare. Supponiamo che ogni persona sia in grado di parlare almeno una lingua in L. Se ogni impiegato ha la stessa probabilità di essere selezionato, utilizziamo la distribuzione di probabilità uniforme P definita sull'insieme finito Ω. La terna (Ω, P, Γ) descrive adeguatamente l'insieme casuale. Tipiche domande di interesse in un tale contesto sono le seguenti:

- Qual è la proporzione $P1$ di impiegati che sanno parlare tedesco e inglese e non sanno parlare nessun'altra lingua?
- Qual è la proporzione $P2$ di impiegati che sanno parlare tedesco o inglese ma nessun'altra lingua?
- Qual è la proporzione $P3$ di impiegati che sanno parlare tedesco o inglese?
- Qual è la proporzione $P4$ di impiegati che sanno parlare almeno tre lingue?

Tali domande possono essere risposte attraverso un'analisi formale dell'insieme casuale corrispondente rispettivamente come segue:

$P1 = P(\omega \in \Omega : \Gamma(\omega) = \{Inglese, Tedesco\})$
$P2 = P(\omega \in \Omega : \Gamma(\omega) \subseteq \{Inglese, Tedesco\})$

$P3 = P(\omega \in \Omega : \Gamma(\omega) \cap \{Inglese, Tedesco\} \neq \emptyset)$
$P4 = P(\omega \in \Omega : |\Gamma(\omega)| \geq 3)$

Limite superiore di probabilità Indica la proporzione di elementi la cui immagine "tocca" un certo sottoinsieme di U

$$P^*(A) = P(\{\omega \in \Omega | \Gamma(\omega) \cap A \neq \emptyset\})$$

Limite inferiore di probabilità Indica la proporzione di elementi la cui immagine è interamente contenuta in un dato sottoinsieme di U

$$P_*(A) = P(\{\omega \in \Omega | \Gamma(\omega) \subseteq A \text{ e } \Gamma(\omega) \neq \emptyset\})$$

Attraverso questi strumenti possiamo analizzare dati descritti in modo fuzzy. Possiamo associare, infatti, ad ogni elemento della mappa una probabilità attesa $E(\Gamma)$ di modo che

$E(\Gamma) = \{E(X) | X(\omega) \in \Gamma(\omega)$ e la X è una variabile randomica tale che $E(X), \forall \omega \in \Omega\}$

3 Evolutionary computing

3.1 Introduzione

Gli algoritmi evolutivi sono una classe di tecniche di ottimizzazione che imitano i principi dell'evoluzione biologica. Essi appartengono alla famiglia delle metaeuristiche. Il principio fondamentale degli algoritmi evolutivi è quello di applicare principi evolutivi come la mutazione e la selezione a popolazioni di soluzioni candidate al fine di trovare una soluzione (sufficientemente buona) per un dato problema di ottimizzazione.

Optimization problems Un *problema di ottimizzazione* può essere descritto da una tripla $(\Omega, f, \prec)$ dove Ω è lo spazio di ricerca, f è una funzione di valutazione della forma $f : \Omega \to \mathbb{R}$ e $\prec$ un preordine. L'insieme $H \subseteq \Omega$ tale che:

$$H = \{x \in \Omega | \forall x' \in \Omega : f(x) \succeq f(x')\}$$

è definito l'insieme degli *ottimi globali*. Dato un problema di questo genere la sua soluzione sta nel fornire un elemento che appartiene all'insieme H

Gli *algoritmi evolutivi* rispondono a questo problema adottando una strategia innovativa. Tali algoritmi sono direttamente ispirati alla teoria della evoluzione biologica

Principi fondamentali evoluzione biologica Gli algoritmi evolutivi sono tra le metaeuristiche più antiche e popolari. Sono essenzialmente basati sulla teoria dell'evoluzione biologica di Charles Darwin proposta nel libro "*Le origini delle specie*", che esprime la diversità e complessità di tutte le forme di vita. Il principio alla base dell'evoluzione biologica consiste in:

1. Tratti vantaggiosi che sono risultato di mutazioni casuali tendono ad essere favoriti dalla selezione naturale
2. Gli individui che mostrano questi tratti vantaggiosi hanno migliori opportunità di procreare e moltiplicarsi

È importante comprendere che un tratto non è benefico o dannoso di per sé, ma solo in relazione all'ambiente[12]

Elementi di un algoritmo evolutivo Un algoritmo evolutivo richiede i seguenti elementi

[12] Un pigmento della pelle scuro può essere utile in Africa per difendersi dai raggi ultravioletti ma altrove può condurre una carenza di vitamina D; oppure l'anemia (la ridotta capacità di trasportare ossigeno nel sangue) che però protegge dalla malaria, ed è quindi un vantaggio in alcuni regioni.

1. una ***codifica*** per i candidati di soluzione. Poiché vogliamo evolvere una popolazione di candidati di soluzione, abbiamo bisogno di un modo per rappresentarli come cromosomi, cioè dobbiamo codificarli, essenzialmente come sequenze di oggetti computazionali (bit, caratteri, numeri, ecc.).

2. Un metodo per creare una ***popolazione iniziale***. Poiché i cromosomi sono semplici sequenze di oggetti computazionali, una popolazione iniziale viene comunemente creata generando semplicemente sequenze casuali

3. Una ***funzione di fitness***, indica le probabilità di sopravvivere e riprodursi, dovuti alla capacità di adattarsi all'ambiente. Spesso è la funzione da ottimizzare

4. ***Metodi di selezione*** rispetto ai valori di fitness. Si scelgono così gli individui che dovranno procreare nella successiva generazione

5. ***Operatori genetici*** che modificano i cromosomi. Utilizzati negli algoritmi genetici per generare nuove soluzioni candidate attraverso la manipolazione dei geni o delle caratteristiche delle soluzioni esistenti. I nuovi tratti possono essere creati da diversi processi, i più utilizzati sono:

   - *Mutazione*, introduce una piccola perturbazione casuale in una soluzione candidata. Avviene modificando casualmente uno o più geni della soluzione.
   - *Crossover*, è un operatore che simula il processo di riproduzione sessuale. Nelle forme di vita con riproduzione sessuata vengono presi elementi dai DNA dei genitori per comporre il DNA della prole. Durante le prime fasi della moltiplicazione cellulare il DNA viene ripetutamente ricombinato (per questo è anche chiamato ricombinazione) creando così nuovi tratti.

   Nella maggior parte dei casi le mutazioni sono sfavorevoli o addirittura letali, ma c'è una piccola probabilità che siano invece vantaggiose, ovvero che aiutino la sopravvivenza. Ogni nuovo individuo è "testato" subito dall'ambiente che lo circonda, se il nuovo tratto è sfavorevole l'individuo non sopravvive e il tratto così sparisce dalla popolazione.

6. *Vari parametri*, come dimensione della popolazione, probabilità di mutazione. Ad esempio, la dimensione della popolazione da evolvere, la frazione di individui scelti da ogni popolazione per produrre discendenti, la probabilità che si verifichi una mutazione in un individuo, ecc

7. ***Condizione di terminazione***, numero di generazioni, approssimazioni all'ottimo

Per ogni problema di ottimizzazione occorre separare lo spazio dei *fenotipi* Ω, come l'individuo appare, ciò che interagisce con l'ambiente e a cui viene applicata la funzione di fitness, da quello dei *genotipi* Γ, come l'individuo è rappresentato dalla codifica scelta e su cui agiscono gli operatori genetici. La funzione di fitness sarà definita sui fenotipi, gli operatori genetici agiranno sui genotipi. Per valutare i cambiamenti nel genotipo sarà necessario provvedere una funzione di decodifica $dec : \Gamma \to \Omega$.

Individuo E' un organismo vivente in biologia, e corrisponde nel nostro caso ad un candidato alla soluzione del problema. Ogni *individuo* A è rappresentato da un tupla $(A.G, A.S, A.F)$ contenente il genotipo $(A.G \in \Gamma)$, informazioni e parametri addizionali $A.S \in Z$ e la valutazione dello stesso rispetto alla funzione di fitness $A.F = f(dec(A.G))$.

Gene E' una parte del cromosoma, è l'unità fondamentale che si va ad ereditare e definisce un tratto, come il colore degli occhi.

Allele Un allele è la possibile forma di un gene in biologia, se un gene rappresenta il colore degli occhi, ha degli alleli che codificano il colore blu, o marrone, o verde ecc. C'è un solo allele per ciascun gene.

Popolazione Una popolazione è un insieme di individui della stessa specie (di norma).

Generazione Una generazione si riferisce alla popolazione in un momento del tempo specifico.

Operatore di mutazione E' definito come una mappa

$$Mut^{\xi} : \Gamma \times Z \to \Gamma \times Z$$

dove ξ è un numero randomicamente generato.

Operatore di ricombinazione Avente $r \geq 2$ genitori e $s \geq 1$ figli è definito come una mappa

$$Rek^{\xi} : (\Gamma \times Z)^r \to (\Gamma \times Z)^s$$

dove ξ è un numero randomicamente generato

Operatore di selezione Ci permette di scegliere grazie ai valori di fitness tra una popolazione di r individui un numero s di individui che continueranno la specie. Sia $P = \{A_1, \ldots, A_r\}$ la popolazione di individui allora l'operatore di selezione avrà la forma:

$$Sel^{\xi} : (\Gamma \times Z \times \mathbb{R})^r \to (\Gamma \times Z \times \mathbb{R})^s$$

$$A_i \quad {\scriptstyle 1 \leq i \leq r} \mapsto A_{IS^{\xi}(c_1,\ldots,c_r)_k} \quad {\scriptstyle 1 \leq k \leq s}$$

dove la selezione ha la forma

$$IS^{\xi} : \mathbb{R}^r \to \{1, \ldots, r\}^s$$

Definizione formale di algoritmo evolutivo Un algoritmo evolutivo su un problema di ottimizzazione P è una tupla

$$(\Gamma, dec, Mut, Rek, IS_{genitori}, IS_{ambiente}, \mu, \lambda)$$

Dove *dec* è la decoding function, $IS_{genitori}$ è la selection function applicata ai genitori, $IS_{ambiente}$ è la selection function che identifica altri elementi nella popolazione la cui selezione deriva da fattori ambientali μ descrive il numero degli individui della generazione precedente e λ descrive il numero di figli da generare per la generazione successiva.

Si può fare una distinzione all'interno degli algoritmi evolutivi

- *Algoritmi genetici*
- *Algoritmi evolutivi*

3.2 Meta-euristiche

Le meta-euristiche sono tecniche computazionali piuttosto generali che vengono tipicamente utilizzate per risolvere problemi di ottimizzazione numerica e combinatoria in modo approssimato in più iterazioni. Le meta-euristiche vengono generalmente definite come una sequenza astratta di operazioni su determinati oggetti e possono essere applicate a problemi essenzialmente arbitrari. Tuttavia, gli oggetti su cui operano e i passaggi da effettuare devono essere adattati al problema specifico.

Le metaeuristiche vengono di solito applicate a problemi per i quali non è noto un algoritmo di soluzione efficiente, cioè problemi per i quali tutti gli algoritmi conosciuti hanno una complessità temporale (asintotica) esponenziale rispetto alla dimensione del problema. Nella pratica, tali problemi possono raramente essere risolti in modo esatto, a causa delle elevate richieste di tempo di calcolo e/o di potenza di calcolo. Di conseguenza, è necessario accettare soluzioni approssimate, e questo è ciò che le meta-euristiche possono fornire

Local search method E' un tipo di meta-euristica che cerca di trovare il massimo globale di una funzione di ottimizzazione attraverso l'ispezione di punti vicini a quelli scelti in fase di inizializzazione. Questo metodo si basa sull'idea che la funzione non abbia salti significativi tra i punti vicini. A differenza di altri algoritmi evolutivi, questo utilizza solo un singolo individuo e si concentra sulla mutazione piuttosto che sulla ricombinazione. Durante ogni iterazione, l'algoritmo decide se continuare a modificare l'individuo attuale o crearne uno nuovo per evitare minimi o massimi locali. Il gradient ascent/descent viene utilizzato per identificare il punto di massimo/minimo, con l'ampiezza dei passi che deve essere bilanciata per evitare oscillazioni o convergenza lenta.

Per evitare di rimanere bloccati in minimi o massimi locali, l'algoritmo viene eseguito su diversi punti.

Tutti questi metodi sono talvolta chiamati metodi di ricerca locale, perché compiono solo piccoli passi nello spazio di ricerca e quindi effettuano una ricerca locale di soluzioni migliori

Hill climbing Il termine Hill Climbing indica la capacità dell'algoritmo di scalare i nodi verso quelli con valori maggiori. Lo spazio di ricerca dell'algoritmo è limitato ai soli nodi vicino a quello corrente. Il rischio è quello di incappare in un massimo locale.

Se la funzione f non è differenziabile, la discesa o l'ascesa del gradiente non è un'opzione. Tuttavia, possiamo cercare di determinare una direzione in cui f aumenta valutando punti casuali nelle vicinanze del punto corrente. Il risultato è noto come *hill climbing*

Simulated annealing Il *simulated annealing* è una generalizzazione di questo approccio che prevede l'accettazione di soluzioni peggiori in base alla loro "qualità". Si basa sul principio che "la transizione da un massimo locale più basso ad uno più alto è più probabile della transizione in direzione opposta". Le soluzioni migliori sono sempre tenute in considerazione, mentre le soluzioni peggiori sono anch'esse considerate con una certa probabilità e un parametro *temperatura* che decresce nel tempo: soluzioni peggiori sono più accettabili all'inizio dell'esplorazione dello spazio invece che alla fine, per consentire una convergenza.

Tabu search La Tabu Search è un algoritmo di ricerca locale che tiene conto della storia delle generazioni precedenti durante la creazione di una nuova soluzione. L'obiettivo è evitare di ritornare su soluzioni già esplorate in passato e di superare i possibili ottimi locali. Durante l'esecuzione dell'algoritmo, viene mantenuta una lista Tabu, che funziona come una coda FIFO, in cui vengono registrati i candidati già esplorati. Questi candidati sono *tabù* e non vengono considerati nelle fasi successive della ricerca, in modo da evitare cicli ripetitivi

L'obiettivo principale della Tabu Search è quello di esplorare lo spazio delle soluzioni in modo più efficace, superando i possibili ottimi locali e cercando di raggiungere la migliore soluzione globale

3.3 Elementi di algoritmi evolutivi

Gli algoritmi evolutivi non sono procedure fisse, ma contengono diversi elementi che devono essere adattati al problema di ottimizzazione da risolvere

3.3.1 Codifica

Il modo in cui le soluzioni candidate al problema di ottimizzazione vengono codificate può avere un notevole impatto sulla facilità con cui un algoritmo

evolutivo trova una soluzione buona. In generale, è importante prestare attenzione all'interazione tra la codifica scelta e gli operatori genetici. Se la codifica riduce lo spazio di ricerca, ma si scopre essere difficile trovare operatori genetici che garantiscono che il risultato della loro applicazione sia in questo spazio di ricerca ridotto, potrebbe essere necessario gestire tali casi.

Non esiste una ricetta generale, il problema della codifica è specifico per ogni problema. Possiamo identificare le seguenti proprietà che una codifica dovrebbe avere

I) ***Hamming Cliffs***, rappresentare fenotipi simili con genotipi simili. Due genotipi sono ovviamente simili se differiscono di alcuni geni, perché servono poche mutazioni per passare da un genotipo all'altro. Possiamo definire la similarità dei genotipi con quanti operatori genetici servono per mutare da un genotipo all'altro.

   Gli Hamming cliffs si verificano quando due genotipi o sequenze di bit che rappresentano soluzioni simili a livello fenotipico (cioè soluzioni candidate simili) hanno una grande distanza di Hamming a livello genotipico, cioè differiscono per molti bit. In altre parole, gli Hamming cliffs si presentano quando due soluzioni simili richiedono molte mutazioni o modifiche dei bit per trasformare una sequenza nella sua controparte simile.

II) ***Epistasis***, la funzione di fitness deve restituire valori simili per candidati simili. In biologia l'epistasi è il fenomeno in cui l'allele di un gene sopprime gli effetti di tutti i possibili alleli di un altro gene. Negli algoritmi evolutivi descrive l'interazione tra due geni di un cromosoma: quanto cambia la fitness di una soluzione se si cambia un suo gene.

III) ***Closedness of the Search Space***, lo spazio Ω deve essere chiuso rispetto agli operatori genetici. Tutte le soluzioni possibili sono rappresentate, nessuna soluzione non possibile è rappresentata. Se non è chiuso ci possono essere soluzioni che non sono codificabili ed interpretate

3.3.2 Fitness and selection

Alla base della selezione c'è il principio che gli individui migliori hanno migliori probabilità di procreare e contribuire alla creazione delle nuove generazioni. Questo processo di selezione deve essere bilanciato attraverso l'uso della cosiddetta ***selective pressure***[13], che può essere bassa per favorire l'esplorazione dello spazio di ricerca o alta per sfruttare gli individui migliori vicini all'ottimo.

- Un'alta selective pressure indica che anche piccole differenze nella fitness tra individui danno probabilità molto diverse di procreare: sfruttiamo gli individui migliori, perché solo quelli con alta fitness hanno buone possibilità di riprodursi

[13]La selective pressure, o pressione selettiva, è un parametro chiave negli algoritmi genetici che influenza la probabilità di selezione degli individui durante il processo di riproduzione e creazione delle nuove generazioni

- Una bassa selective pressure indica che le probabilità di riprodursi degli individui dipendono poco dalle differenze nei valori di fitness: favoriamo l'esplorazione dello spazio di ricerca, anche individui in regioni con basso fitness hanno buone possibilità di riprodursi

La strategia migliore per scegliere la selective pressure giusta è time-dependent: nelle prime generazioni tenerla bassa, e alzarla mano a mano fino alle ultime generazioni. Possiamo usare metriche come:

- *Selection intensity*, valutare la differenza tra i valori di fitness prima e dopo la selezione
- *Time to takeover/tempo di convergenza*, indica il numero di generazioni necessarie affinché la popolazione converga verso una soluzione ottima o di alta qualità. Questo valore riflette la velocità con cui l'algoritmo genetico è in grado di individuare e sfruttare le soluzioni migliori presenti nello spazio di ricerca

$$f_{rel}(A_i) = \frac{A_i.F}{\sum_{j=1}^{|P|} A_j.F}$$

La probabilità per un individuo di essere selezionato per la riproduzione sarà proporzionale al suo valore di fitness relativo. Alcuni svantaggi

- La computazione del valore di fitness relativo è costosa e difficilmente parallelizzabile
- Scomparsa delle biodiversità, perchè gli individui con un alto valore di fitness potrebbero dominare la selezione
- Molto veloce a trovare ottimi locali, ma pessima esplorazione dello spazio

3.3.3 Selezione

Varie strategie disponibili in letteratura per operare la selezione degli individui che costituiranno il pool genetico per la successiva generazione

- *Roulette-wheel selection*, il funzionamento della roulette-wheel selection è simile a una ruota della roulette in cui ogni individuo ha un settore proporzionale alla sua probabilità di selezione. La somma delle probabilità di tutti gli individui corrisponde a un intero giro della ruota. Per selezionare un individuo, viene generato casualmente un numero tra 0 e la somma delle probabilità e viene eseguita una scansione dei settori della ruota fino a quando il numero casuale è compreso all'interno di un settore. L'individuo corrispondente a quel settore viene selezionato.

  Nella roulette-wheel selection, gli individui con un rank o un valore di fitness più alto avranno settori più ampi sulla ruota, aumentando così la loro probabilità di essere selezionati

- *Rank-based selection*, si ordinano gli individui in ordine di fitness decrescente. A seconda della posizione si assegna ad ogni individuo un rank e con esso si definisce la probabilità di essere selezionati. Si procede ad una selezione del tipo roulette-wheel. Lo svantaggio sta che occorre ordinare gli individui, complessità $O(n \log n)$ dove n è il numero degli individui

- *Tournament selection*, si estraggono k individui casualmente dalla popolazione. Tramite confronti individuali, basandosi su rank o valore di fitness si decide il migliore, il quale riceverà la possibilità di riprodursi nella prossima generazione. Si riesce così ad evitare il problema della dominanza[14] perché ogni scontro individuale offre una possibilità per gli individui meno dominanti di emergere come vincitori e avere la possibilità di riprodursi

- *Elitismo*, in tutti i casi precedenti non ci sono garanzie che i migliori passino da una generazione alla successiva. Con l'etilismo un numero selezionato di individui migliori passa direttamente alla generazione successiva per preservare l'elite. Il vantaggio è che la convergenza viene ottenuta rapidamente. Lo svantaggio è che c'è il rischio di rimanere bloccati in ottimi locali, poiché gli individui di élite vengono mantenuti senza subire modifiche, potrebbero non essere in grado di esplorare nuove regioni dello spazio di ricerca che potrebbero contenere soluzioni migliori.

- *Crowding*, individui delle generazioni successive rimpiazzano individui simili a loro. La densità locale in Γ non può crescere in modo indefinito, questo permette una migliore esplorazione dello spazio. Per implementare il crowding, è necessario definire una metrica di distanza o similarità tra gli individui, comunemente si utilizza la distanza euclidea o la distanza di Hamming, a seconda della codifica utilizzata. Gli individui più simili tra loro saranno quelli che avranno una distanza più piccola. Durante il processo evolutivo, quando si selezionano gli individui per la riproduzione, vengono scelti quelli che hanno una bassa similarità con gli individui già presenti nella popolazione. In altre parole, si preferiscono gli individui che sono "diversi" dagli altri.

3.3.4 Operatori genetici

Fungono da strumenti per esplorare lo spazio di ricerca. Tali operatori servono principalmente a produrre una soluzione candidata che sia (molto) simile al genitore. Se le soluzioni candidate sono codificate come stringhe di bit (ovvero, i cromosomi sono composti da zeri e uni), la mutazione bit (nota anche come inversione di bit) è comunemente scelta. Consiste nel capovolgere casualmente gli alleli selezionati, cioè trasformare un 1 in 0 e viceversa.

Gli operatori genetici sono comunemente categorizzati in base al numero di genitori in *mutation (one parent) operator*; *crossover (two-parent)*; *multi-parent operator*. Le ultime due categorie, cioè con più di un genitore, sono talvolta chiamate genericamente operatori di ricombinazione.

[14] Il problema della dominanza si riferisce alla situazione in cui un individuo con un rank o un valore di fitness molto alto domina gli altri individui, impedendo loro di essere selezionati o influenzando significativamente la diversità genetica della popolazione

Mutation/One parent operator Nella prima classe possiamo trovare l'operatore di mutazione. Introduce biodiversità e favorisce l'esplorazione dello spazio di ricerca. Esistono vari metodi per operare una mutazione

1. *Standard mutation*, sostituisce l'allele corrente di un gene scelto casualmente con un altro allele scelto casualmente

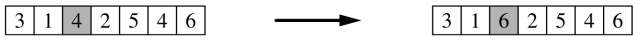


Figura 18: Esempio di standard mutation: un allele di un gene viene sostituito da un altro allele

2. *Pair swap*(trasposizione), swap della posizione di due geni

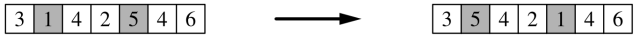


Figura 19: Esempio di trasposizione: due geni scambiano i loro alleli in un cromosoma

3. *Shift*, shifta a destra o sinistra un gruppo di geni
4. *Inversion*, invertire l'ordine degli alleli
5. *Arbitrary permutation*, permutazione arbitraria di un gruppo di geni

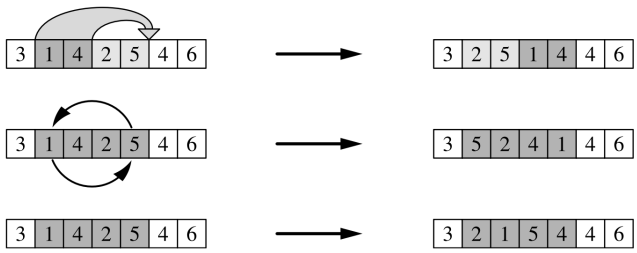


Figura 20: Operatori di mutation su sottosequenze: shift, inversion e arbitrary permutation

Crossover/Two parent operator Vari modi per ricombinare

1. *One-point crossover*, viene scelto casualmente un cut point e le sequenze di geni su un lato di questo punto vengono scambiate tra i due cromosomi

2. *Two-point crossover*, è un'estensione diretta del one-point crossover. In questo caso, vengono scelti due cut point casuali e la sezione compresa tra i due punti viene scambiata tra i cromosomi.

Figura 21: Esempio di one-point crossover

Figura 22: Esempio di two-point crossover

3. *N-point crossover*, un generalizzazione dei precedenti. Si scambiano le aree incluse nei punti selezionati casualmente
4. *Uniform crossover*, per ogni gene si determina se scambiarlo o meno a seconda di un certo parametro di probabilità p_x
5. *Shuffle crossover*, mescola casualmente i geni prima di applicare un operatore arbitrario a due genitori e quindi ripristina l'ordine originale dei geni. Prima una permutazione randomica, poi one-point crossover, poi unmixing
6. *Uniform order-based crossover*, operatore genetico utilizzato per combinare i geni di due genitori in modo flessibile e casuale. Utilizza una maschera di crossover per selezionare casualmente le posizioni dei geni da entrambi i genitori, creando discendenti con maggiore variabilità e potenzialmente esplorando una più ampia gamma di soluzioni nell'ottimizzazione e nella ricerca.
7. *Edge-recombination crossover*, il cromosoma è rappresentato come un grafo. Ogni gene è un vertice che ha archi verso i suoi vicini. Gli archi dei due grafi vengono mischiati. Si preserva l'informazione relativa alla vicinanza (permutazione).

Multiple parent operator Il *diagonal crossover* è un operatore di ricombinazione che può essere applicato a tre o più genitori ed è una generalizzazione del one-point crossover. Dati k genitori, si scelgono $k-1$ punti per il crossover e si procede shiftando diagonalmente le sequenze rispetto ai punti scelti. Il crossover diagonale è considerato un metodo molto efficace per esplorare lo spazio di ricerca, specialmente quando si hanno un gran numero di genitori (circa 10-15).

Caratteristiche operatori crossover Gli operatori di ricombinazione sono spesso categorizzati in base a determinate proprietà

- *Positional bias*, quando la probabilità che due geni vengano ereditati assieme dallo stesso genitore dipende dalla posizione (relativa) dei due ge-

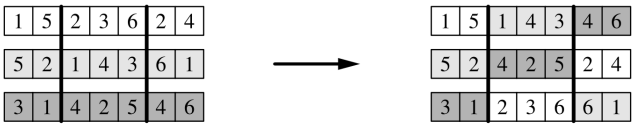


Figura 23: Esempio di diagonal crossover

ni nel cromosoma. Deve essere evitato perché può rendere la disposizione dei geni cruciale per la riuscita dell'algoritmo.

Un esempio semplice di un operatore di ricombinazione che mostra un bias posizionale è il one-point crossover: la probabilità che due geni vengano ereditati congiuntamente è più alta quando i geni sono più vicini nel cromosoma. Ciò avviene perché i geni vicini hanno solo pochi cut points possibili tra di loro. Solo se uno di questi punti viene scelto, i geni vengono separati. Poiché tutti i cut points hanno la stessa probabilità, la probabilità che due geni siano ereditati congiuntamente è inversamente proporzionale al numero di punti di taglio tra di loro e quindi alla loro distanza nel cromosoma. Un caso estremo è rappresentato dal primo e l'ultimo gene di un cromosoma, che non possono mai essere ereditati congiuntamente mediante one-point crossover poiché qualsiasi punto di taglio li separa. Due geni vicini in un cromosoma vengono separati dal one-point crossover con una probabilità di $\frac{1}{L-1}$, dove L è la lunghezza del cromosoma.

- *Distributional bias*, un operatore di ricombinazione presenta un bias distributivo se la probabilità che un certo numero di geni venga scambiato tra i cromosomi genitori non è la stessa per tutti i possibili numeri di geni. Il bias distributivo di solito è meno critico (più tollerabile) rispetto al bias posizionale. Un esempio semplice di un operatore di ricombinazione che presenta un bias distributivo è il crossover uniforme.

Un esempio di operatore di crossover che non presenta né un bias posizionale né un bias distributivo è il crossover shuffle basato sul one-point crossover.

Migliorare le performance Due strategie per migliorare le performance

- *Interpolating recombination*, avviene attraverso una combinazione lineare dei geni dei genitori. Ogni gene del discendente è calcolato come una media ponderata dei geni corrispondenti dei genitori. Ad esempio, se abbiamo due genitori A e B con i geni $[1, 3, 5]$ e $[2, 4, 6]$, rispettivamente, e utilizziamo un peso di 0.5 per entrambi i genitori, il discendente C avrà i geni $[\frac{1+2}{2}, \frac{3+4}{2}, \frac{5+6}{2}] = [1.5, 3.5, 5.5]$. Questo processo di combinazione lineare permette di creare discendenti con caratteristiche medie dei genitori originali.
- *Extrapolating recombination* oltre alla media ponderata dei geni dei genitori, viene introdotto un fattore di estensione che permette di generare nuovi valori al di là dei limiti dei genitori. Ad esempio, se abbia-

mo due genitori A e B con i geni $[1, 3, 5]$ e $[2, 4, 6]$, rispettivamente, e utilizziamo un fattore di estensione di 1.5, il discendente C avrà i geni $[1.5 \cdot (1-2), 1.5 \cdot (3-4), 1.5 \cdot (5-6)] = [-1.5, -1.5, -1.5]$. Questo processo permette di esplorare nuove regioni dello spazio delle soluzioni durante la ricerca e l'ottimizzazione, potenzialmente portando a soluzioni più diverse e innovative.

3.3.5 Swarm and population based optimization

Swarm intelligence La swarn intelligence è un'area di ricerca nell'intelligenza artificiale che si concentra sullo studio del comportamento collettivo di popolazioni di agenti semplici. I sistemi di swarm intelligence sono spesso ispirati al comportamento di alcune specie animali, in particolare insetti sociali come formiche e api, nonché animali che vivono e cercano cibo in gruppi come pesci, uccelli e lupi. Questi animali, cooperando tra loro, sono spesso in grado di risolvere problemi piuttosto complessi. Ad esempio, possono trovare i percorsi più brevi per fonti di cibo, costruire nidi, cacciare prede. Esistono varie tipologie di euristiche di questo genere

- *Particle swarm optimization*, ispirato al pattern biologico della ricerca del cibo in uccelli e pesci. Gli individui aggregano informazioni, creando un insieme di conoscenze comuni, al fine di presentare una sola soluzione. Ogni individuo è un candidato ad essere la soluzione
- *Ant colony optimization*, ispirato al pattern biologico delle formiche che cercano una strada che le conduca al cibo. Gli individui scambiano informazioni modificando l'ambiente, in modo che gli altri possano seguire (o meno) le loro tracce. Ogni individuo è un candidato ad essere la soluzione

Population based incremental learning In questa tecnica di ottimizzazione, gli individui sono generati casualmente utilizzando una distribuzione di probabilità. Non è necessario conservare esplicitamente ogni singolo individuo nella memoria, ma è sufficiente mantenere le statistiche della popolazione, come ad esempio la media e la deviazione standard delle soluzioni.

Per la *ricombinazione*, viene utilizzato l'operatore *uniform crossover*. La selezione degli individui si basa sul miglioramento delle statistiche della popolazione, cioè si scelgono gli individui che portano a una media migliore o a una deviazione standard più piccola.

La *mutazione*, invece, prevede solo un semplice *bit-flip*, ovvero un cambiamento casuale di un singolo bit nel genotipo. Una caratteristica distintiva di questa tecnica è che il tasso di apprendimento, ovvero il parametro che regola la velocità di movimento degli individui nello spazio delle soluzioni, cambia nel tempo e si riduce con il numero di iterazioni.

3.3.6 Fondamenti teorici

Per verificare la validità degli algoritmi evolutivi, è necessario considerare gli *schemata*, che sono cromosomi binari parzialmente specificati che codificano un comportamento particolare. Questi schemata possono essere utilizzati per studiare l'evoluzione nel tempo del numero di cromosomi che condividono lo stesso schema.

L'obiettivo è quello di fornire una stima statistica di come gli algoritmi evolutivi esplorano lo spazio di ricerca. In questo modo, è possibile valutare l'efficacia degli algoritmi e capire come essi si comportano nella ricerca di soluzioni ottimali.

Definizione 18 *Uno schema h è una stringa di caratteri di lunghezza L sull'alfabeto $0, 1$, cioè $h \in 0, 1, L$. Il carattere è chiamato carattere jolly o don't care symbol*

Definizione 19 *Un cromosoma c si dice che condivide lo schema h (in simboli, $c \triangleleft h$) se e solo se, escluse le posizioni in h aventi il simbolo $*$, h coincide con c.*

A titolo di esempio sia h uno schema di lunghezza $L = 10$ e $c1$, $c2$ due diversi cromosomi di questa lunghezza, che appaiono così:

$$h = *01110$$

$$c1 = 1100111100$$

$$c2 = 1111111111$$

Chiaramente, il cromosoma $c1$ corrisponde allo schema h, cioè $c1 \triangleleft h$, perché $c1$ differisce da h solo nelle posizioni in cui h è $*$. D'altra parte, il cromosoma $c2$ non corrisponde a h, cioè $c2 \not\triangleleft h$, perché $c2$ contiene 1 nelle posizioni in cui h è 0 (posizioni 3 e 9).

In un algoritmo genetico, la fitness rappresenta la capacità di un cromosoma di risolvere un determinato problema o di adattarsi a un ambiente specifico. Quando si utilizzano schemi specifici per analizzare la popolazione di cromosomi, la fitness viene calcolata solo per quei cromosomi che condividono lo schema in questione. Ad esempio, se lo schema specifico è rappresentato da una sequenza di bit "1101", allora solo i cromosomi che contengono questa sequenza esatta di bit verranno considerati nella valutazione della fitness. L'ordine di uno schema è importante perché indica quanti bit all'interno dello schema devono essere mantenuti costanti per preservare l'efficacia dello schema durante la mutazione.

Per calcolare l'influenza dell'operatore di mutazione su uno schema specifico, occorre valutare la probabilità che lo schema venga preservato dopo la mutazione. Ciò significa che, dopo che un cromosoma è stato sottoposto a mutazione, occorre valutare se lo schema specifico è ancora presente e, in caso contrario, determinare se la mutazione ha portato a un miglioramento o a una riduzione della fitness complessiva del cromosoma

3.4 Strategie evolutive

In una strategia evolutiva, consideriamo l'intero processo evolutivo, non solo l'ottimizzazione dei singoli individui. Prendiamo in considerazione parametri come riproduzione, mortalità, e la lunghezza media della vita degli individui, che dipendono dalle scelte che facciamo riguardo agli operatori genetici. Rappresentiamo il problema di ottimizzazione come una funzione $f : \mathbb{R}^n \to \mathbb{R}$ che vogliamo minimizzare, e utilizziamo array di reali come rappresentazione dei cromosomi. Utilizziamo solo l'operatore di mutazione per spostare i cromosomi all'interno dello spazio di ricerca aggiungendo un vettore randomico r ottenuto da una distribuzione normale.

Selection La selezione nelle strategie evolutive segue un rigoroso principio di élite: solo i migliori individui passano alla generazione successiva. Nonostante il principio di élite sia fisso, esistono due diverse forme di selezione, che si distinguono dal fatto che vengono considerati solo i discendenti o se i genitori e i discendenti insieme sono presi in considerazione nel processo di selezione. Sia μ il numero di individui nella generazione dei genitori e λ il numero di individui discendenti creati per mutazione:

- *Plus-strategy*, i genitori e i figli vengono raggruppati per il processo di selezione, ossia la selezione viene effettuata su $\mu + \lambda$ individui
- *Comma-strategy*, la selezione considera solo gli individui discendenti

3.5 Programmazione genetica

Con la genetic programming (GP) si cerca di evolvere espressioni simboliche o addirittura programmi informatici con determinate proprietà.

Finora abbiamo considerato solo cromosomi che erano array di lunghezza fissa, ad esempio array di bit per gli algoritmi genetici o array di numeri reali per le strategie evolutive. Per la programmazione genetica, abbandoniamo la restrizione a una lunghezza fissa e permettiamo cromosomi di lunghezza variabile.

La base formale è una grammatica che descrive il linguaggio dei programmi genetici. Seguendo l'approccio standard dei linguaggi formali, definiamo due insiemi, ovvero l'insieme $\mathcal{F}$ di simboli di funzione e operatori e l'insieme $\mathcal{T}$ di costanti e variabili. $\mathcal{F}$ e $\mathcal{T}$ dovrebbero essere limitati in dimensione al fine di ridurre lo spazio di ricerca a una dimensione fattibile, ma "abbastanza ricchi" per consentire una soluzione al problema.

A titolo di esempio, supponiamo di voler apprendere una funzione booleana che mappa n input binari in output binari associati. In questo caso, i seguenti insiemi di simboli sono una scelta naturale:

$$\mathcal{F} = \{\text{and}, \text{or}, \text{not}, \text{if} \ldots \textit{then} \ldots \textit{else} \ldots\}$$

$$\mathcal{T} = \{x1, \ldots, xn, 1, 0\} \text{ o } T = \{x1, \ldots, xn, \text{vero}, \text{falso}\}$$

Una proprietà desiderabile di $\mathcal{F}$ è che tutte le funzioni in esso siano complete nel dominio, ovvero dovrebbero accettare qualsiasi valore di input possibile. Semplici esempi di funzioni che non sono complete nel dominio sono la divisione, che provoca un errore se il divisore è zero, o il logaritmo, che di solito accetta solo argomenti positivi.

Un'altra proprietà importante è la completezza degli insiemi di funzioni $\mathcal{F}$ e $\mathcal{T}$ rispetto alle funzioni che possono rappresentare. La programmazione genetica può risolvere efficacemente un dato problema solo se $\mathcal{F}$ e $\mathcal{T}$ sono sufficienti per trovare un programma appropriato. Ad esempio, nella logica proposizionale booleana, $\mathcal{F} = \{\wedge, \neg\}$ e $\mathcal{F} = \{\rightarrow, \neg\}$ sono insiemi completi di operatori, perché qualsiasi funzione booleana con qualsiasi numero di argomenti può essere rappresentata da opportune combinazioni degli operatori in questi insiemi. Tuttavia, $\mathcal{F} = \{\wedge\}$ non è un insieme completo, perché non è possibile rappresentare nemmeno la semplice negazione di un argomento. Trovare il più piccolo insieme completo di operatori per un dato insieme di funzioni da rappresentare è (di solito) NP-hard. Di conseguenza, $\mathcal{F}$ di solito contiene più funzioni di quanto effettivamente necessario.

I programmi vengono rappresentati come alberi sintattici, dove i nodi interni rappresentano le operazioni e le foglie rappresentano le variabili o le costanti. Tuttavia, per risolvere un problema con successo, l'insieme di operazioni e simboli terminali deve essere *completo* e *sufficiente*

Parsing tree È conveniente rappresentare un'espressione simbolica mediante il cosiddetto albero di parsing. Gli alberi di parsing sono comunemente utilizzati, ad esempio, nei compilatori, soprattutto per le espressioni aritmetiche.

Inizializzazione Creare una popolazione iniziale è un po' più complesso nella programmazione genetica rispetto agli algoritmi evolutivi, perché non possiamo semplicemente creare sequenze casuali di simboli di funzione, costanti, variabili e parentesi. Data la complessità che esibiscono queste strutture, nel processo di creazione, bisogna considerare alcuni parametri quali l'*altezza massima degli alberi* e il *numero massimo di nodi*. Esistono vari sotto-algoritmi che si occupano dell'inizializzazione degli alberi sintattici:

- *Grow*, la probabilità di scegliere un nodo interno o uno terminale è distribuita in modo uniforme a qualsiasi livello di profondità. Questo permette di creare alberi sbilanciati
- *Full*, i nodi terminali possono occorrere solo al livello dell'altezza massima dell'albero. Questo permette di creare alberi bilanciati
- *Ramp-half-and-half*, i primi due metodi possono essere combinati creando metà della popolazione iniziale con il metodo grow e l'altra metà con il metodo full per avere più varianza nella forma esibita dagli alberi sintattici. Questo metodo ha il vantaggio di garantire una buona diversità nella popolazione, con alberi di altezza e complessità diverse

Operatori genetici La popolazione iniziale della programmazione genetica (come in qualsiasi algoritmo evolutivo) di solito ha un valore di fitness molto basso, perché è altamente improbabile che una generazione casuale di parsing tree produca anche una soluzione vagamente adeguata. Per ottenere candidati di soluzioni migliori, vengono applicati gli operatori genetici. I tre più importanti sono: *crossover*, consiste nello scambio di due sotto-espressioni (e quindi sotto-alberi nei parsing tree); *mutation*; *cloning* (duplicazione di un individuo)

È comune limitare la mutazione alla sostituzione di piccoli sotto-alberi (con un'altezza non superiore a tre o quattro), in modo da creare un individuo effettivamente simile. Una mutazione non limitata potrebbe sostituire l'intero albero di parsing, il che equivale a introdurre un individuo completamente nuovo. Tuttavia, se la popolazione è sufficientemente ampia in modo che il "materiale genetico" presente garantisca una diversità adeguata, la mutazione spesso viene abbandonata e il crossover diventa l'unico operatore genetico[15].

Il motivo è che il crossover, rispetto ad altri algoritmi evolutivi che lavorano con array di lunghezza fissa, è un operatore molto più potente. Ad esempio, se lo applichiamo a due individui identici, il semplice fatto che possano essere scelti diversi sotto-alberi crea la possibilità di ottenere due individui diversi.

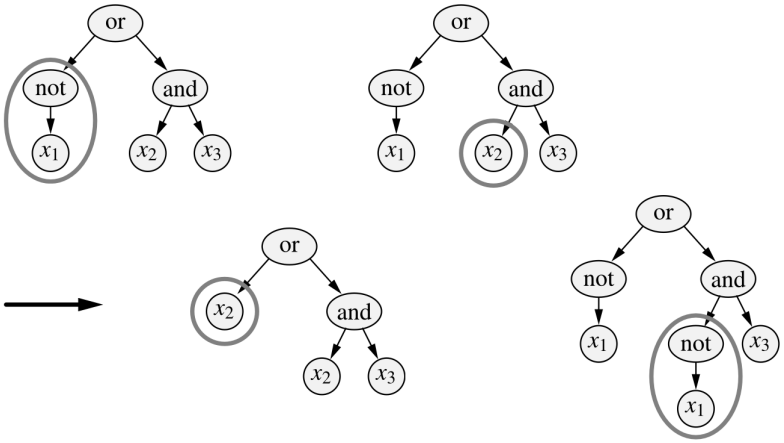


Figura 24: Vantaggio del crossover nella programmazione genetica rispetto agli operatori di crossover basati su array: applicato a due cromosomi identici, è possibile ottenere due figli diversi

Introni Durante il processo evolutivo gli individui tendono a sviluppare larghe porzioni di codice "inutile" ai fini della computazione. Un concetto simile viene dalla biologia: gli introni sono porzioni di DNA che non codificano alcuna informazione a livello del fenotipo (per questo vengono talvolta chiamati junk-DNA).

[15]Questo è esattamente l'opposto delle strategie evolutive, in cui spesso si abbandona il crossover e la mutazione è l'unico operatore genetico

Ad esempio, se vengono creati delle sotto-espressioni come (if $2 > 1$ then ... else ...), la parte else dell'istruzione condizionale è un introne, perché non può mai essere eseguita a meno che la condizione di test non venga modificata.

In generale, è consigliabile evitare la creazione di introni il più possibile, poiché gonfiano i cromosomi inutilmente, aumentano il tempo di elaborazione e rendono più difficile interpretare le soluzioni trovate.

Per evitare il verificarsi di questo fenomeno esistono alcune strategie:

- *Breeding recombination*, questo operatore crea molti figli dagli stessi due genitori applicando un operatore di crossover con parametri diversi. Solo il miglior figlio del gruppo entra nella generazione successiva
- *Intelligent recombination*, sceglie intenzionalmente i punti di crossover
- *Continuos slight changes* TODO ...

3.6 Multi-criteria optimization

Nella vita quotidiana, ci troviamo spesso di fronte a situazioni che non possono essere descritte nella semplice forma dei problemi di ottimizzazione. In particolare, spesso ci troviamo di fronte al compito di selezionare tra un insieme di opzioni che soddisfano diversi criteri in modo differente. Molto spesso, questi criteri sono addirittura contrastanti, ossia cercare di migliorare uno fa sì che un altro criterio sia meno soddisfatto. Consideriamo, ad esempio, il compito di trovare un appartamento o di acquistare un bene di consumo. In generale, qualità e prezzo sono criteri in conflitto.

Weight combination of criteria Formalmente, l'ottimizzazione multi-criterio può essere descritta da k funzioni obiettivo

$$f_i : \Omega \to \mathbb{R}, i = 1, \ldots, k$$

Il nostro obiettivo è trovare un elemento dello spazio di ricerca per il quale tutte le funzioni restituiscono un valore il più alto possibile. L'approccio più semplice a questo problema è combinare tutte le k funzioni obiettivo in una singola funzione, riducendo così il problema a un problema di ottimizzazione standard. Ad esempio, possiamo calcolare una somma ponderata

$$f(s) = \sum_{i=1}^{k} w_i \cdot f_i(s)$$

dove i valori assoluti dei pesi specificano l'importanza relativa che attribuiamo ai diversi criteri.

Sfortunatamente, un approccio basato sulla combinazione di diversi criteri in questo modo presenta gravi inconvenienti: oltre al fatto che potrebbe non essere facile scegliere pesi adeguati, perdiamo la possibilità di adattare le nostre preferenze relative in base alle proprietà delle soluzioni potenziali che otteniamo

(cosa che sicuramente facciamo nei processi decisionali come la ricerca di un appartamento). Tuttavia, il problema è ancora più fondamentale: in generale, qui ci troviamo di fronte al problema di dover aggregare le preferenze.

3.6.1 Pareto optimal solution

Un approccio alternativo per combinare criteri multipli nell'ottimizzazione è cercare di trovare tutte o almeno molte soluzioni Pareto-ottimali.

Definizione 20 *Un elemento $s \in \Omega$ viene chiamato Pareto-ottimale rispetto alle funzioni obiettivo f_i , $i = 1, \ldots, k$ se non esiste alcun elemento $s\prime \in \Omega$ per il quale valgono le seguenti due proprietà:*

$$\forall i, 1 \leq i \leq k : f_i(s\prime) \geq f_i(s)$$

$$\exists i, 1 \leq i \leq k : f_i(s\prime) > f_i(s)$$

In modo intuitivo, l'ottimalità di Pareto significa che la soddisfazione di un qualsiasi criterio non può essere migliorata senza danneggiarne un altro.

Chiaramente, un vantaggio della ricerca di soluzioni Pareto-ottimali è che le funzioni obiettivo non devono essere combinate (e quindi non c'è bisogno di specificare pesi o una funzione di aggregazione). Inoltre, preserviamo la possibilità di adattare la nostra visione di quanto sia importante un criterio rispetto agli altri in base alle soluzioni che otteniamo. Tuttavia, lo svantaggio è che raramente esiste una sola soluzione Pareto-ottimale e quindi non esiste una soluzione univoca al problema di ottimizzazione

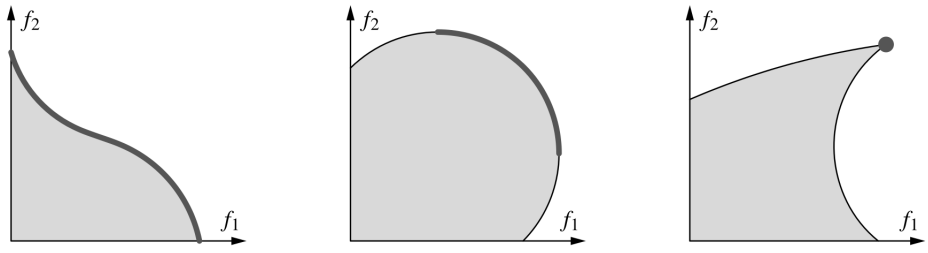


Figura 25: Illustrazione delle soluzioni Pareto-ottimali, ovvero la cosiddetta frontiera di Pareto. Tutti i punti dello spazio di ricerca sono situati nell'area grigia (con le funzioni f_1 e f_2 che forniscono le coordinate). Le soluzioni Pareto-ottimali si trovano nella parte del confine disegnata in grigio scuro. Ad eccezione del diagramma a destra, ci sono molteplici soluzioni Pareto-ottimali

.

Finding Pareto oprimal with Evolutionary Algorithms Fino ad ora abbiamo sempre supposto di avere una singola funzione da ottimizzare, gli algoritmi evolutivi possono essere applicati anche a problemi di ottimizzazione

multi-criterio. In questo caso, l'obiettivo è trovare una selezione di candidati alla soluzione sulla frontiera di Pareto o almeno vicino ad essa, che copra in modo sufficiente tale frontiera.

3.7 Algoritmi evolutivi paralleli

Rispetto ad altre tecniche di ottimizzazione, gli algoritmi evolutivi sono noti per produrre risultati ottimali, ma con un costo computazionale elevato. Un modo per migliorare questa situazione è la parallelizzazione, ovvero la distribuzione delle operazioni necessarie su diversi processori. Si può parallelizzare:

- La generazione iniziale è spesso facile da parallelizzare, poiché di solito i cromosomi vengono creati casualmente e in modo indipendente l'uno dall'altro
- Il calcolo del fitness degli individui
- La selezione se costituita da eventi indipendenti, come ad esempio tournament selection, diventa più complesso gestire etilismo
- L'applicazione degli operatori genetici
- Il controllo di raggiungimento del criterio di terminazione

Due architetture utilizzate sono: *island model*; *cellular evolution*

Island model and migration Facendo riferimento a una chiara analogia con la natura, ogni popolazione può essere vista come che abita un'isola, il che spiega il nome per questa architettura. Un modello di isole puro è equivalente all'esecuzione dello stesso algoritmo evolutivo più volte, che può essere fatto anche in modo seriale. Ogni isola avrà una popolazione, ed eseguirà il processo evolutivo.

Un altro processo è noto come *migration*. L'idea alla base di questo metodo è che il trasferimento di materiale genetico tra le isole migliori le proprietà esplorative delle popolazioni delle isole, senza necessità di ricombinazioni dirette tra cromosomi provenienti da isole diverse. Esistono diverse modalità di migrazione tra le isole. Nel modello casuale, le coppie di isole vengono selezionate casualmente e scambiano alcuni dei loro individui. In questo caso, qualsiasi coppia di isole può teoricamente scambiarsi individui. Nel modello di rete, invece, le isole sono organizzate in una struttura di rete o grafo, e gli individui possono migrare solo tra isole connesse da un lato del grafo.

Sfruttando l'architettura a isole e la migrazione, è possibile ottenere una parallelizzazione efficace degli algoritmi evolutivi.

Cellular evolution Gli algoritmi evolutivi cellulari sono una forma di parallelizzazione chiamata anche *isolation by distance*. Essi lavorano con un gran

numero di processori (virtuali), ciascuno dei quali gestisce un singolo individuo, o solo un piccolo numero di individui. L'intero spazio di ricerca viene diviso in una griglia rettangolare.

Per la selezione, ogni processore valuta i cromosomi dei propri vicini e calcola il massimo tra di essi. Gli operatori genetici, come la mutazione o la ricombinazione, sono applicati solo tra i cromosomi dei vicini. In questo modo, ogni processore lavora indipendentemente, ma condivide informazioni con i suoi vicini per influenzare la selezione e l'evoluzione. La mutazione di un cromosoma è gestita da ogni singolo processore