

Intelligenza artificiale: Riproduzione parziale dell'attività intellettuale propria dell'uomo (con particolare riguardo ai processi di apprendimento, di riconoscimento, di scelta) realizzata o attraverso l'elaborazione di modelli ideali, o, concretamente, con la messa a punto di macchine per fare quello che gli esseri umani non fanno bene: processare tante informazioni in pochissimo tempo.

Machine Learning: Il Machine Learning è il campo di studi che fornisce ai computer la capacità di imparare a risolvere i problemi senza essere esplicitamente programmati.

Deep Learning: Il deep learning, o deep neural learning, è un sottoinsieme del machine learning, che utilizza le reti neurali per analizzare diversi fattori con una struttura simile al sistema neurale umano.

Storia dell'intelligenza artificiale, Origini, I due Inverni, Tempi moderni: 2011 ad oggi

1940-1974 “La nascita e gli anni d’oro”: nascita dei primi calcolatori elettronici (relè e valvole termoioniche), Test di Turing per valutare se una macchina dimostra un comportamento intelligente, primo modello di neurone artificiale con Pitts e McCulloch, grande entusiasmo e predizioni troppo ottimistiche.

1974-1980 “Il primo inverno”: risultati non all'altezza delle aspettative con conseguente riduzione dei finanziamenti. I problemi erano dovuti ai computer lenti e poco potenti, gli algoritmi si bloccavano di fronte a problemi complessi ed era difficile risolvere problemi reali in tempi accettabili. Inoltre i sistemi avevano pochi dati su cui allenarsi, quindi imparavano poco e male.

1980-1987 “Nuova primavera”: Nacquero i sistemi esperti, cioè programmi che usavano conoscenze umane combinate con regole logiche (tipo: "se succede X, allora fai Y"). Venne pubblicato un algoritmo chiamato Backpropagation. Questo metodo permetteva di allenare meglio le reti neurali, che tornarono così a essere interessanti e più potenti rispetto al passato. Inoltre ci fu un finanziamento dal governo giapponese per la Quinta Generazione di Calcolatori.

1987-1993 “Il secondo inverno”: Flop «Quinta generazione». Nuovo stop finanziamenti. I sistemi esperti funzionavano solo in contesti molto limitati e specifici, non erano flessibili, e non riuscivano ad adattarsi a situazioni nuove o più generali.

1993-2011 “Tempi moderni”: Hardware sempre più potente, raccolta di grandi quantità di dati (Big Data) sempre più semplice e meno costosa e il recupero dell'algoritmo di backpropagation portarono alla rinascita del Deep Learning con successi in molte discipline.

Paradigma del machine Learning vs Paradigma di programmazione tradizionale.

Nel paradigma del machine learning, il sistema impara dai dati, in particolare ci sono 4 passaggi fondamentali:

- 1) Acquisizione dati: i dati sono l'elemento fondante di qualsiasi applicazione correlata al ML.
- 2) Data processing: tutte quelle tecniche con cui vengono elaborati i dati per adattarli al meglio al modello ML in questione
- 3) Modello: l'insieme delle tecniche matematiche e statistiche, in grado di apprendere da una certa distribuzione di dati forniti in input e di generalizzare su nuovi dati.
- 4) Predizione: l'output del modello può assumere molte forme a seconda dell'applicazione sviluppata.

Nel paradigma tradizionale, si scrivono regole esplicite (istruzioni) per ottenere un output dato un input.

Preparazione dei dati (Training Set, Validation Set, Testing Set)

1. **Training set**: i dati sui quali il modello apprende automaticamente durante la fase di apprendimento.
2. **Validation set**: parte del training set. Su questi dati, vengono messi a punto gli iperparametri.
3. **Testing set**: dati su cui il modello viene testato durante la fase di test

Task del Machine Learning: Classificazione, Regressione e Clustering

Nel machine learning, i task principali sono:

1. **Classificazione**: Dato un input specifico, il modello(classificatore) emette una classe: se ci sono solo 2 classi, chiamiamo il problema classificazione binaria; se ci sono più classi (>2), chiamiamo il problema classificazione multiclasse (es Dati: immagini a raggi X Classi/etichette: tumore maligno / benigno)
2. **Regressione**: modella la relazione tra le variabili indipendenti e la variabile dipendente. Emette un valore continuo (es: Stima dell'altezza di una persona in base al peso)
3. **Clustering**: identificazione di gruppi (cluster) di dati con caratteristiche simili

Classificazione degli algoritmi di Machine Learning: Algoritmi Supervisionati (Regressione e classificazione), Algoritmi non supervisionati (Clustering)

Se ho dati annotati(o etichettati), sono in un contesto di learning supervisionato parliamo dunque di classificazione(definiamo una classe/etichetta per i dati in input) e regressione(emettiamo un dato continuo a partire da un certo dato etichettato). Se non ho dati annotati, sono in un contesto di learning non supervisionato. Il clustering è spesso applicato in un ambiente di apprendimento non supervisionato, in cui i dati non sono etichettati.

Neurone Artificiale: Neurone Biologico e Neurone artificiale. Funzioni di attivazione. Percettrone a soglia e limiti.

Neurone biologico: riceve segnali elettrici da altri neuroni tramite i dendriti, li elabora nel soma e invia un segnale attraverso l'assone se la stimolazione supera una certa soglia.

Neurone artificiale: riceve in ingresso dei valori numerici(input), ciascuno associato a un peso che ne rappresenta l'importanza. Somma tutti questi input pesati e aggiunge un valore chiamato bias, che serve a spostare la funzione di attivazione. Il risultato finale passa attraverso una funzione di attivazione, che trasforma il valore grezzo in un output

Funzione di attivazione: determina se un neurone deve essere attivato o meno. Si tratta di alcune semplici operazioni matematiche per determinare se l'input del neurone alla rete è rilevante o meno nel processo di previsione. Le funzioni di attivazione devono essere non lineari (per la comprensione delle connessioni complesse tra i dati in input) e derivabili.

Percettrone a soglia: È un modello di neurone artificiale semplice: Calcola la somma pesata degli input e applica una funzione a soglia (step): output 1 se sopra la soglia, 0 altrimenti.

Limiti del Percettrone a soglia: il percettrone a soglia non è in grado di modellare funzioni non lineari. Per ovviare a questo problema sono state introdotte le reti neurali feedforward

Reti Neurali artificiali: Input Layer, Hidden Layer, Output Layer. Reti FeedForward, Reti ricorrenti

Una rete neurale è composta da tre tipi di livelli. Il livello di input riceve i dati grezzi, quello di output produce il risultato finale della rete neurale, mentre i livelli nascosti (hidden layer) elaborano i dati ricevuti in input e trasmettono il segnale ai neuroni dello strato successivo.

Reti FeedForward: è un tipo di rete neurale artificiale in cui i segnali si muovono in una sola direzione, dallo strato di input allo strato di output.

Reti ricorrenti: è un tipo di rete neurale artificiale in cui i segnali si muovono in entrambe le direzioni, dallo strato di input allo strato di output e viceversa.

MultiLayer Perceptron (MLP)

Il MultiLayer Perceptron (MLP) è una rete neurale composta da almeno tre livelli: uno di input, uno o più hidden layer e uno di output. Ogni neurone di un livello è collegato a tutti quelli del livello successivo(FeedForward). L'MLP usa funzioni di attivazione non lineari nei livelli nascosti, permettendo di risolvere anche problemi complessi. L'addestramento avviene tramite l'algoritmo di backpropagation, che regola i pesi per ridurre l'errore tra output previsto e output reale.

Training di una rete neurale. Forward Propagation e Backward Propagation.

Training di una rete neurale: è il processo di addestramento di una rete neurale artificiale. Iterativamente, la rete neurale viene esposta a un insieme di dati di addestramento e viene aggiornata in base all'errore tra l'output previsto e l'output desiderato con l'obiettivo di minimizzarlo. Durante la forward propagation, i dati di input attraversano la rete, passando da un livello all'altro. Ogni neurone calcola una somma pesata dei suoi input, aggiunge il bias e applica una funzione di attivazione, producendo l'output finale.

Nella backward propagation, l'errore tra l'output prodotto e quello desiderato viene propagato all'indietro. La rete aggiorna i pesi e i bias usando tecniche di ottimizzazione in modo da ridurre progressivamente l'errore. Questo processo si ripete per molte epochhe finché la rete non apprende una rappresentazione efficace dei dati.

Iperparametri di una rete neurale.

Gli iperparametri di una rete neurale sono valori impostati prima dell'addestramento e influenzano il comportamento e le prestazioni del modello. Esempi comuni sono: il numero di layer e di neuroni per layer, la funzione di attivazione, il learning rate (velocità di apprendimento), il numero di epochhe, la dimensione del batch e la funzione di ottimizzazione. Questi parametri non si apprendono dai dati ma si scelgono tramite test e validazione

Loss Function e Funzione costo nell'apprendimento supervisionato: caso del task della regressione; caso del task della classificazione.

Nel contesto dell'apprendimento supervisionato, la Loss Function e la Funzione Costo sono fondamentali per valutare l'errore della rete neurale e guiderne l'addestramento. Secondo il documento fornito, le funzioni di costo variano in base al tipo di task:

Caso della regressione

Nel caso della **regressione**, si adotta tipicamente la **funzione di perdita (loss function)** definita come **somma dei quadrati degli errori** tra l'output prodotto dalla rete e l'etichetta reale:

$$L(y, \hat{y}) = \frac{1}{2} \sum_{i=1}^s (y_i - \hat{y}_i)^2 = \frac{1}{2} \|y - \hat{y}(\mathbf{w})\|^2$$

Dove:

- Y è il valore del target (etichetta reale)
- Y[^] è l'output della rete
- W è il vettore dei pesi della rete

Caso della classificazione

Solitamente si usa il metodo Cross-entropy loss se l'output è probabilistico, tuttavia anche l'errore quadratico può essere usato in forma semplificata soprattutto in esempi introduttivi

Addestramento e minimizzazione della funzione di costo -> Durante l'addestramento, l'obiettivo è minimizzare la funzione di costo tramite la discesa del gradiente, aggiornando i pesi della rete in modo iterativo. Questa procedura permette alla rete di migliorare la sua capacità di previsione riducendo l'errore. Nel contesto **supervisionato**, la funzione costo si costruisce come media della loss su tutto il training set

$$C(\mathbf{w}) = \frac{1}{n_T} \sum_{j=1}^{n_T} L(y^{(j)}, \hat{y}^{(j)}(\mathbf{w}))$$

L'obiettivo è trovare i pesi ottimali w* che minimizzano la seguente funzione:

$$\mathbf{w}^* \in \arg \min_{\mathbf{w}} C(\mathbf{w})$$

Reti neurali Convoluzionali: Architettura di una rete CNN: parte convoluzione e parte fully-connected La parte convoluzionale consiste di strati convoluzionali seguite da funzioni di attivazione non lineare tipo(RELU) e di pooling. Questa parte costituisce il componente essenziale dell'estrazione di feature • La parte fully-connected consiste in un'architettura di rete neurale completamente connessa. Questa parte esegue il compito di classificazione in base all'input dalla parte convoluzionale.

Le reti neurali convoluzionali (CNN) sono progettate specificamente per l'elaborazione delle immagini. La loro architettura si divide in due parti principali:

1. Parte convoluzionale (Feature Extraction)

- Composta da layer convoluzionali, che estraggono caratteristiche visive dai dati in input.
- Ogni layer convoluzionale utilizza kernel (o filtri) che eseguono operazioni di convoluzione, applicando una trasformazione locale sull'immagine per generare feature maps.
- Dopo la convoluzione, si applicano funzioni di attivazione non lineare, come ReLU (Rectified Linear Unit), per introdurre non linearità nel modello e migliorare la capacità di apprendimento.

- I layer di pooling riducono la dimensionalità delle feature maps, conservando le informazioni essenziali e aumentando l'invarianza alla traslazione.

2. Parte fully-connected (Classificazione)

- Dopo l'estrazione delle caratteristiche, il volume di feature viene appiattito (flatten layer) e passato a una rete fully-connected (MLP).
- Questa parte è responsabile della classificazione, basandosi sulle feature estratte dalla parte convoluzionale.
- I neuroni completamente connessi aggregano le informazioni e generano la predizione finale della CNN.

Questa architettura consente alle CNN di apprendere automaticamente rappresentazioni gerarchiche delle immagini, con feature più semplici nei primi layer convoluzionali e caratteristiche più complesse nei layer profondi

Algoritmo di backpropagation per il calcolo delle derivate parziali della funzione costo rispetto ai pesi di tutti i layer . (Ricavare la formula di aggiornamento dei pesi nel caso di una rete MLP con 1 layer di input con un solo nodo, 2 layer nascosti ciascuno con un solo nodo ed un nodo di output)

L'algoritmo di backpropagation è una tecnica fondamentale per il calcolo delle derivate parziali della funzione costo rispetto ai pesi di una rete MLP (Multi-Layer Perceptron). Per una rete composta da:

- 1 nodo nel layer di input
- 2 layer nascosti, ciascuno con 1 nodo
- 1 nodo nel layer di output

l'aggiornamento dei pesi si basa sulla discesa del gradiente e la propagazione dell'errore all'indietro.

Passaggi principali dell'algoritmo di backpropagation.

1) Primo passaggio, fare scorrere l'input attraverso la rete

Forward Pass (senza bias):

1. Layer 1:

$$a^{(1)} = w^{(1)}x, \quad z^{(1)} = f(a^{(1)})$$

2. Layer 2:

$$a^{(2)} = w^{(2)}z^{(1)}, \quad z^{(2)} = f(a^{(2)})$$

3. Output:

$$a^{(3)} = w^{(3)}z^{(2)}, \quad \hat{y} = z^{(3)} = f(a^{(3)})$$

- 2) Secondo passaggio usare una loss function per calcolare l'errore tra l'output ottenuto e quello ideale

La loss Function (viene scelto l'Errore quadratico):

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

- 3) Applichiamo la chain rule per derivare la loss rispetto ai pesi. Serve per sapere in che direzione (e di quanto) dobbiamo aggiornare ogni peso (fase di backward propagation)

1. Output layer:

$$\frac{\partial L}{\partial w^{(3)}} = (\hat{y} - y) \cdot f'(a^{(3)}) \cdot z^{(2)}$$

2. Hidden layer 2:

$$\frac{\partial L}{\partial w^{(2)}} = (\hat{y} - y) \cdot f'(a^{(3)}) \cdot w^{(3)} \cdot f'(a^{(2)}) \cdot z^{(1)}$$

3. Hidden layer 1:

$$\frac{\partial L}{\partial w^{(1)}} = (\hat{y} - y) \cdot f'(a^{(3)}) \cdot w^{(3)} \cdot f'(a^{(2)}) \cdot w^{(2)} \cdot f'(a^{(1)}) \cdot x$$

- 4) Aggiornamento dei pesi, si usa il metodo della discesa del gradiente. Ad ogni iterazione, il peso viene spostato nella direzione opposta al gradiente della loss

$$w^{(l)} := w^{(l)} - \eta \cdot \frac{\partial L}{\partial w^{(l)}}$$

Per ogni layer $l = 1, 2, 3$, con learning rate η .

Tecniche di Ottimizzazione:

Nel contesto dell'addestramento delle reti neurali, l'ottimizzazione della funzione di costo avviene attraverso diverse varianti del metodo di discesa del gradiente:

1. Batch Gradient Descent:

- Utilizza tutti i campioni del training set per calcolare la funzione di costo.
- L'aggiornamento dei parametri avviene una sola volta per epoca, dopo aver elaborato l'intero dataset.

- È stabile ma può essere computazionalmente costoso per dataset di grandi dimensioni.

2. Stochastic Gradient Descent (SGD):

- Utilizza una singola osservazione per calcolare la funzione di costo e aggiornare i parametri.
- L'aggiornamento avviene ad ogni iterazione, rendendo il processo più veloce ma anche più rumoroso.
- Può aiutare a superare minimi locali, ma introduce una maggiore variabilità nei passi di aggiornamento.

3. Mini-batch Gradient Descent:

- Suddivide il dataset in piccoli gruppi (mini-batch) e calcola la funzione di costo su ciascun gruppo.
- Combina i vantaggi di Batch e Stochastic Gradient Descent: è più stabile di SGD e più efficiente di Batch Gradient Descent.
- È la tecnica più comunemente utilizzata nelle moderne applicazioni di deep learning.

Per la convergenza -> Il metodo di discesa del gradiente con passo fisso converge a un punto stazionario della funzione di costo sotto le seguenti condizioni:

- **La funzione di costo deve essere differenziabile:** La discesa del gradiente si basa sul calcolo del gradiente, quindi la funzione deve essere almeno continuamente differenziabile.
- **Il learning rate deve essere adeguatamente scelto:** Se il passo è troppo grande, l'algoritmo può oscillare senza convergere; se è troppo piccolo, la convergenza sarà molto lenta.
- **La funzione di costo deve essere convessa per garantire il minimo globale:** Se la funzione è convessa, ogni punto stazionario è un minimo globale, quindi la discesa del gradiente garantisce la convergenza al miglior valore possibile.
- **Assenza di minimi locali:** Se la funzione non è convessa, la discesa del gradiente può fermarsi in un minimo locale invece di raggiungere il minimo globale.

Non convessità della funzione di costo

La funzione di costo in un problema di ottimizzazione ideale dovrebbe essere convessa, perché in questo caso la discesa del gradiente garantisce la convergenza verso il minimo globale. Tuttavia, nei problemi di addestramento delle reti neurali, la funzione di costo non è convessa a causa della non linearità introdotta dagli strati nascosti.

Questo comporta la presenza di:

- **Minimi locali**, dove l'algoritmo di discesa del gradiente può bloccarsi senza raggiungere il minimo globale.
- **Regioni piatte**, chiamate *vanishing gradient*, dove il gradiente è quasi nullo e l'aggiornamento dei pesi diventa inefficace.

- **Punti sella**, in cui il gradiente è zero, ma il punto non corrisponde né a un minimo né a un massimo, creando difficoltà nell'ottimizzazione.

quasi tutti i problemi di ottimizzazione nelle reti neurali sono non convessi a causa dell'introduzione della non linearità. In particolare, aggiungendo strati nascosti, la funzione di costo diventa più complessa e presenta molteplici minimi locali, rendendo più difficile trovare il vero minimo globale.

Importanza del Learning Rate nei metodi di discesa

Il *learning rate* (η) gioca un ruolo cruciale nella discesa del gradiente, perché determina quanto i pesi vengono aggiornati a ogni passo. Una scelta non appropriata del learning rate può compromettere la convergenza dell'algoritmo:

- **Learning rate basso:**
 - Richiede molti passi prima di raggiungere un buon minimo.
 - Può far sì che i pesi rimangano bloccati in un minimo locale sub-ottimale.
- **Learning rate alto:**
 - Permette di superare i minimi locali, favorendo la ricerca del minimo globale.
 - Può causare instabilità e oscillazioni eccessive, impedendo la convergenza.

L'ottimizzazione del learning rate è quindi fondamentale e può richiedere sperimentazioni per trovare il valore ottimale. In pratica, si usano tecniche di *learning rate scheduling* e *adaptive learning rate* per adattare il valore dinamicamente durante l'allenamento.

Metodo di ottimizzazione del gradient descent con momento. Perchè è stato studiato e formula di aggiornamento dei pesi.

Il metodo del Gradient Descent con **Momento** (Momentum) è stato introdotto per superare i limiti del gradient descent classico, in particolare:

1. Lentezza nelle aree piatte del paesaggio della funzione di costo (plateau).
2. Oscillazioni in direzioni con variazioni ripide (valli strette).
3. Possibilità di rimanere bloccati in minimi locali non ottimali.

Il Momentum aiuta a velocizzare la convergenza e a rendere più stabile il percorso nella discesa del gradiente, accumulando le direzioni precedenti del gradiente, in modo da seguire con più decisione le direzioni coerenti e smorzare quelle instabili.

Nel metodo con **momento**, si introduce un termine ausiliario v_t , che rappresenta una sorta di "velocità" dei pesi:

$$v_{t+1} = \mu v_t - \eta \nabla C(w_t)$$

$$w_{t+1} = w_t + v_{t+1}$$

Effetti pratici

- Quando i gradienti puntano nella stessa direzione, $\nabla \text{loss} \cdot \nabla \text{loss}$ cresce \rightarrow accelerazione del processo.
- Quando cambiano direzione frequentemente, si smorzano a vicenda \rightarrow riduzione dell'oscillazione.
- Risulta particolarmente utile nelle reti profonde o con dati rumorosi.

Aggiornamento del learning rate programmato (learning rate scheduling) : step decay, decadimento esponenziale, decadimento dipendente dal tempo

Il Learning Rate (LR) scheduling: La regolazione del learning rate durante l'allenamento è spesso importante tanto quanto la selezione dell'ottimizzatore. Un Learning Rate alto è auspicabile all'inizio poiché i pesi sono lontani dai valori per cui la loss function raggiunge il suo minimo.

Un Learning rate basso è più appropriato nella fase finale dell'apprendimento perché i pesi sono già vicini ai valori per cui la loss function raggiunge il suo minimo, (aumentando la possibilità di raggiungere il minimo) Per regolare il Learning rate , ci sono una serie di aspetti da considerare:

- **Grandezza:** se il learning rate è troppo grande, l'ottimizzazione diverge, se è troppo piccolo ci vuole troppo tempo per l'allenamento o si finisce con un risultato non ottimale
- **Tasso Di Decadimento:** se il Learning Rate rimane grande potremmo rimbalzare intorno al minimo senza raggiungerlo

Tra le strategie comuni per il decadimento del learning rate abbiamo:

1. **STEP DECAY:** riduce il learning rate iniziale η_0 di un fattore gamma ogni numero predefinito di epoch s

$$\eta = \eta_0 \cdot \delta^{\lfloor \frac{n}{s} \rfloor}$$

- dove η_0 e δ e s sono iperparametri e n è l'iterazione corrente

$\lfloor \cdot \rfloor$ = restituisce il numero di volte che il learning rate è già stato ridotto

Il learning rate rimane costante per un certo numero di epoch , e poi "salta" bruscamente a un valore più basso in punti specifici dell'addestramento.

2. DECADIMENTO ESPONENZIALE

$$\eta = \eta_0 \cdot e^{-\delta n}$$

dove η_0 e δ sono iperparametri e n è l'iterazione

corrente

η_0 è il learning rate iniziale.

δ è un iperparametro positivo che controlla la velocità con cui il learning rate diminuisce. Un δ più grande significa un decadimento più rapido.

Il decadimento esponenziale fa sì che il learning rate diminuisca in modo non lineare, riducendosi sempre più velocemente all'inizio e poi rallentando la sua diminuzione man mano che l'addestramento procede

3. DECADIMENTO BASATO SUL TEMPO:

modifica il learning rate iniziale (η_0) in funzione del numero di iterazioni eseguite (n)

$$\eta = \frac{\eta_0}{1 + \delta \cdot n}$$

Il decadimento è più rapido all'inizio dell'addestramento (quando n è piccolo e il denominatore cresce in modo relativamente significativo per ogni incremento di n), e poi rallenta man mano che n diventa grande (poiché l'aggiunta di un'unità a n ha un impatto proporzionalmente minore su un denominatore già grande).

Learning rate adattivo per ogni peso (durante il processo di ottimizzazione) : Adagrad, RMSProp, Adadelta, Adam. (formula di aggiornamento dei pesi e discussioni)

Sono stati sviluppati metodi di aggiornamento adattivo del learning rate che modificano dinamicamente il learning rate per ciascun peso durante l'allenamento. Tra questi metodi abbiamo

1. **ADAGRAD** : adatta il learning rate ai parametri, eseguendo aggiornamenti più grandi per i parametri poco frequenti e aggiornamenti più piccoli per quelli frequenti (es. problema del riconoscimento delle forme geometriche semplici). Adatta il learning rate per l'aggiornamento individuale di ciascun parametro $w_j^{(k+1)}$ in proporzione alla sua cronologia di aggiornamento

$$s_j^{(k)} = s_j^{(k-1)} + \left(\nabla C(w_j^{(k)}) \right)^2$$

$$w_j^{(k+1)} = w_j^{(k)} - \frac{\eta}{\sqrt{s_j^{(k)} + \epsilon}} \nabla C(w_j^{(k)})$$

Minore è il gradiente accumulato, minore sarà il valore s_k , e ciò porta ad un learning rate maggiore. Uno dei principali vantaggi è che Adagrad elimina la necessità di regolare manualmente il learning rate. Il principale punto debole è l'accumulo dei gradienti al quadrato: durante l'addestramento la somma accumulata cresce, il learning rate diminuisce diventando infinitesimalmente piccolo fino a quando la rete non è più in grado di acquisire ulteriore

conoscenza. Funziona bene in presenza di gradienti sparsi o feature sparse, tuttavia può diminuire troppo velocemente il tasso di apprendimento rendendolo inefficiente nella fasi successive

2) RMSProp: è un algoritmo di ottimizzazione per l'apprendimento automatico che deriva da Adagrad e ne migliora alcune limitazioni. Adatta il learning rate individualmente per ciascun parametro di una rete neurale durante l'allenamento. RMSProp è stato introdotto per ridurre la diminuzione aggressiva del learning rate di Adagrad in particolare modifica la parte di accumulo del gradiente di Adagrad con una media ponderata esponenziale dei gradienti al quadrato invece della somma dei gradienti al quadrato.

$$s_j^{(k)} = \gamma s_j^{(k-1)} + (1 - \gamma) \left(\nabla C(w_j^{(k)}) \right)^2$$

$$w_j^{(k+1)} = w_j^{(k)} - \frac{\eta}{\sqrt{s_j^{(k)} + \epsilon}} \nabla C(w_j^{(k)})$$

Tra gli effetti abbiamo:

- Riduzione del decadimento aggressivo: La media ponderata esponenziale attenua l'effetto dei gradienti passati
- Maggiore stabilità
- Convergenza più affidabile

Questi effetti tuttavia portano ad un potenziale di accumulo di errore e alla scelta del fattore di smorzamento γ che deve essere scelto attentamente per ottenere il miglior bilanciamento tra stabilità e velocità di convergenza

3. ADAM: L'obiettivo principale di ADAM è quello di combinare i vantaggi di due altri algoritmi di ottimizzazione: RMSprop e Momentum

Utilizza la media pesata esponenziale dei gradienti ai passi precedenti, per ottenere una stima del momento del gradiente per ogni parametro

$$v_j^{(k)} = \beta_1 v_j^{(k-1)} + (1 - \beta_1) \nabla C(w_j^{(k)})$$

e del momento secondo del gradiente

$$s_j^{(k)} = \beta_2 s_j^{(k-1)} + (1 - \beta_2) \left(\nabla C \left(w_j^{(k)} \right) \right)^2$$

Si noti che se si inizializzano $v(0) = 0$ ed $s(0) = 0$, i momenti iniziali sono "sbilanciati" all'inizio del processo di apprendimento, per compensare questo sbilanciamento si usano le seguenti normalizzazioni

$$\hat{v}_j^{(k)} = \frac{v_j^{(k)}}{1 - (\beta_1)^k} \quad \hat{s}_j^{(k)} = \frac{s^{(k)}}{1 - (\beta_2)^k}$$

allora l'equazione di aggiornamento del j-esimo parametro diventa

$$w_j^{(k+1)} = w_j^{(k)} - \frac{\eta}{\sqrt{\hat{s}_j^{(k)} + \epsilon}} \hat{v}_j^{(k)}$$

il metodo ADAM adatta in modo dinamico il learning rate per ciascun parametro del modello utilizzando le stime dei momenti del primo e del secondo ordine. Questo approccio consente ad ADAM di convergere più velocemente e di essere più robusto rispetto ad altri metodi di ottimizzazione tradizionali

TEORIA ESERCIZI, RIASSUNTO ESAME METODI

MATRICE A DIAGONALE STRETTAMENTE DOMINANTE:

Con n dimensione della matrice e A la matrice stessa

```
np.all(np.abs(A.diagonal()) > np.sum(np.abs(A), axis=1) - np.abs(A.diagonal()))
```

ESERCIZI ZERI della FUNZIONE

Se ho una equazione es.

$$f(x) = x^3 - 6x^2 - 4x + 24$$

- 1) Creare la lambda $f = \lambda x: (equazione)$
- 2) Creare un range di punti con $xx = np.linspace(a, b, 100)$
- 3) Poi creare il grafico con $plt.plot(xx, f(xx), np.zeros_like(xx))$
- 4) Vedere dove sono gli zeri della funzione e prendere un nuovo range li vicino
- 5) Applicare la definizione

- 6) Se lo zero non è nullo calcolare errore usando `err=abs(xk - alpha)` dove `xk` è un vettore di zeri cioè `xk=np.array(v_xk)` (con `vx_k` gli iterati per arrivare alla soluzione della funzione)
- 7) Stampare con `plt.semilogy(range(it), err)`

ESERCIZI MATRICI SPECIFICHE

- Matrice di Vandermon: `np.vander(x, increasing=true)`
- Matrice di Hilbert: `scipy.linalg.hilbert(n)` con `n` ordine della matrice

Per trovare l'indice di condizionamento si fa o con `np.linalg.cond(A)` dove `A` è la matrice e se voglio un indice di condizionamento dato da un tipo di norma glielo scrivo di fianco

es. `np.linalg.cond(A, np.inf)` in questo modo ho l'indice di condizionamento della norma infinito

Altrimenti si può calcolare la norma infinito della matrice e il condizionamento è poi dato dal **PRODOTTO** (normale NO scalare) della matrice `A` per la sua inversa

Es. `cond = norma_inf(A)*norma_inf(np.linalg.inv(A))`

Per trovare invece errore sui dati e sulla soluzione si fa

`Err = np.linalg.norm(b_pert-b)/np.linalg.norm(b)`

Stessa cosa nel caso ci fosse `A` e stessa cosa per trovare l'errore relativo nella soluzione

1) FATTORIZZAZIONE PA=LU

La matrice deve essere a Rango massimo, utile quando devo risolvere un sistema lineare, trovare il determinante di `A` o l'inversa della matrice

`np.linalg.matrix_rank(M) <= min(m,n)`

`PT, L, U = lu(A)`

È un metodo per scomporre $A = LU$ ovvero rispettivamente in triangolare inferiore (valori sopra nulli e sulla diagonale = 1) e triangolare inferiore (valori della diagonale fino a sotto nulli)

Tuttavia per rendere `A` più stabile si usa una matrice di permutazione `P` (che è in grado di cambiare l'ordine delle righe e delle colonne di `A`)

Quindi si fa così:

prendo la prima colonna di `A` e vedo quale tra i valori che ci sono è il maggiore. Se il più grande è sotto scambio le righe (e faccio la stessa operazione in `P` che parte della matrice identità).

Poi cerco `U`, devo rendere 0 gli elementi sotto la diagonale quindi trovo un fattore moltiplicativo che mi permette ciò. Una volta trovato salvo questo fattore moltiplicativo nella riga di `L`

Es.

$$A = \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix}$$

Parto da A e verifico che 3 è il valore più grande quindi scambio la prima riga con la seconda

$$A = \begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix}$$

questo scambio che faccio mi serve per capire in realtà come scambiare le righe di P che parte da essere una matrice IDENTITA'

$$PA = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \cdot \begin{bmatrix} 2 & 1 \\ 3 & 4 \end{bmatrix} = \begin{bmatrix} 3 & 4 \\ 2 & 1 \end{bmatrix}$$

Ora scelgo U che deve avere valori nulli sotto la diagonale principale. Qui mi si annullano se dalla seconda riga tolgo 2/3 della prima

$$U = \begin{bmatrix} 3 & 4 \\ 0 & -\frac{5}{3} \end{bmatrix}$$

L invece si crea mettendo sulla diagonale principale valori pari a 1 e sotto nella riga che in U ho cercato di far diventare nulla ci devo mettere il fattore moltiplicativo usato per U ovvero 2/3

$$L = \begin{bmatrix} 1 & 0 \\ \frac{2}{3} & 1 \end{bmatrix}$$

In questo modo è possibile verificare che $PA = LU$

NEI SISTEMI LINEARI posso risolvere $Ax=b$ sfruttando $PA=LU$ facendo $Ly = pb$ e $Ux = y$

Quindi $x = P@b/L@U$

Per trovare il DETERMINANTE di una matrice dato che so che $PA = LU$ mi basta fare

$$\det(A) = \det(L) * \det(U) / \det(P)$$

dove $\det(U) = \text{prodotto elementi in diagonale}$, $\det(L) = 1$, $\det(P) = -1$

Per trovare l'inversa di una matrice $A^{-1} = PL^{-1}U^{-1}$

2) FATTORIZZAZIONE CHOLESKY

La matrice deve essere **SIMMETRICA** (se `np.array_equal(A, A.T)`) e **DEFINITA POSITIVA** (se `np.all(np.linalg.eigvals(A) > 0)`)

Con `scipy.linalg.cholesky(A)` mi restituisce L ovvero la matrice triangolare inferiore di A e io so che

$$A = L^T L$$

3) FATTORIZZAZIONE QR

Fatt = qr(A) ottengo Q matrice ortogonale e R matrice triangolare superiore con A =QR, qui e in cholesky va poi applicata alla matrice triangolare inferiore o superiore la Usolve o la Lsolve

ESERCIZI GAUSS, JACOBI

per trovare il vettore di termini noti b tale per cui x sia un vettore con valori pari a 1, b deve avere come valori la somma degli elementi su ogni riga di A:

```
n = A.shape[0]  
x0 = np.zeros(A.shape[0]).reshape(n,1)  
b = np.sum(A, axis=1).reshape(n,1)  
o np.zeros_like(b)
```

ESERCIZI MINIMI QUADRATI e RETTA con POLINOMI

```
m = x.shape[0]  
n = grado di ciò che si vuole rappresentare (1 -> retta, 2 -> parabola)  
n1 = n+1  
A = np.vander(x, increasing=true)[:, :n1]  
residuo_EQN = np.linalg.norm(A@sol_eqN-y.reshape(m,1))**2
```

per stampare la retta a minimi quadrati

```
xv = np.linspace(np.min(x), np.max(x), 100)  
pol_EQN=np.polyval(np.flip(alpha_EQN),xv)  
pol_QR=np.polyval(np.flip(alpha_QR_LS),xv)  
pol_SVD=np.polyval(np.flip(alpha_SVDLS),xv)  
plt.plot(x,y,'ro',xv,pol_QR,xv,pol_SVD)  
plt.show()
```

IMPORTANTE -> il residuo negli altri due algoritmi (SVDLS e QRRLS) è già negli scheletri

ESERCIZI POLINOMIO di INTERPOLAZIONE

```
x = np.linspace(a, b, quanti_valori_dice il problema)  
y = funzione
```

```
xx = np.linspace(a, b, 100 o 200 ecc)
```

```
pol = intrpl(x, y, xx)  
plt.plot(xx,pol,'b--',x,y1,'r*',xx,f(xx),'m-');
```

Se chiede poi di calcolare la stima o il valori in determinati punti:

```
L = np.array([40])
```

Pol2 = interpl(x, y, L) in questo modo metto nel grafico il nuovo punto che voglio calcolare sulla retta di interpolazione

PASSAGGI ESERCIZIO 1 ESAME

1) calcolo la grandezza della matrice

una matrice è piccola se le sue dimensione vanno da 10 a 100

per dire che è grande le sue dimensioni vanno da 300 a 500

2a) calcolo la sparsità

la sparsità se la matrice è $n*m$ è dato dal numero dei non_zeros, fare il rapporto sul totale degli elementi (rapporto percentuale che varia tra 0 e 1)

se è $sp < 0.33$ è sparsa

se $sp > 0.33$ è densa

sparsa e di grandi dimensioni -> metodi iterativi

3)calcolo se la matrice è a diagonale strettamente dominante (se lo è posso usare jacobi)

$|a_{ij}| >$ (somma) degli elementi che stanno nella sua stessa riga (formula sopra)

4) A simmetrica e definita positiva (uso Gauss-seidel, metodi iterativi, Gauss-Sor se voglio studiarlo per aumentare la velocità)

si può verificare con il criterio di Sylvestre oppure con $\text{np.all}(\text{np.linalg.eigvals}(A) > 0)$

2b) Se la matrice è piccola usiamo i metodi di fattorizzazione (LU, QR, LLT)

3) se la matrice è simmetrica e definita positiva fattorizziamo con cholensky $A = LL^T$, più stabile rispetto a gauss ($1/6 n^{**3}$) (il medio stabile), mal condizionata

4) matrice simmetrica e mal condizionata si usa QR ($2/3n^{**3}$) (il più costoso e il più stabile)

5) Guass è stabile in senso debole ($1/3n^{**3}$) (meno stabile), ben condizionata

2c) Se la matrice risulta non quadrata $m > n$ ci sono i sistemi sovradeterminati

3) uso eqnorm se la matrice è bene condizionata e a rango massimo ($ATAx = ATb$)

4) qr_LS se la matrice è mediamente condizionata

5) se matrice a non rango massimo e mal condizionata: SVDLS -> $A=UsigmaVT$

ben condizionata -> potenza bassa del 10 fino a 100

mal condizionata -> da 10^{**3} (1000)

altamente mal condizionata -> da 10^{**5} in avanti

TEOREMI ESERCIZIO 2 (i teoremi per l'es 1 sono sul foglio stampato)

Zeri di funzioni non lineari

Teorema da sapere -> quello di bisezione (dato un intervallo $[a,b]$ in cui la funzione è continua se $f(a)*f(b) < 0$ allora esiste almeno un punto c in cui la funzione si annulla tale che $c = b-a/2$) + quello che è sul foglio stampato sulla loro convergenza

Inoltre, da sapere che:

- 1) **REGOLA FALSI:** come il metodo di bisezione, con la differenza che l'iterato non è il punto mediano ma la retta passante per i punti $(a, f(a))$ e $(b, f(b))$. Metodo a convergenza SUPERLINEARE
- 2) **NEWTON:** calcola l'iterato successivo usando la derivata prima della funzione ed ha convergenza QUADRATICA poiché la differenza tra la soluzione esatta e l'iterato ad ogni iterazione si dimezza di 4 ma solo se l'iterato iniziale viene scelto vicino alla soluzione
- 3) **CORDE:** come il metodo di newton però usa per aggiornare l'iterato il coefficiente angolare della retta passante per (a,b) quindi il valore è fisso e per questo ha una convergenza lineare
- 4) **SECANTI:** ha due iterati iniziali e per trovare l'iterato successivo si usa la retta secante passante per i due punti, qui la convergenza è SUPERLINEARE

Equazioni di sistemi non lineari

Il metodo di newton rapson vuole trovare la soluzione del sistema ad equazioni non lineari trovando X^* , ovvero $F(X^*) = 0$. Dove $F()$ è una funzione vettoriale con n componenti. Per farlo si sfrutta l'approssimazione lineare del polinomio di taylor da cui deriva che $F(X_k) + J(X_k) * (X_{k+1} - X_k) = 0$. $J(X_k)$ è la jacobiana ovvero una matrice formata dalle derivate parziali della funzione rispetto ai valori.

Se l'iterato iniziale è vicino alla soluzione la convergenza è quadratica questo perché

$|X_{k+1} - X^*| = C * |X_k - X^*|^2$. Quindi la distanza tra il valore esatto e il valore calcolato si riduce quadraticamente ad ogni iterazione questo perché lo jacobiano viene ricalcolato in base al risultato precedente

Negli altri due metodi si ha una convergenza più lenta perché lo Jacobiano viene ricalcolato meno frequentemente (il più lento corde perché c'è un solo aggiornamento)

Minimo di una funzione in più variabili

Per trovare il minimo di una funzione si usa il gradiente della funzione $f : \nabla f(X^*) = 0$ con X^* la soluzione del sistema.

Si utilizza il gradiente perché esso rappresenta il punto di massima variazione di una funzione, quando è uguale a 0 siamo in un punto stazionario (con particolarità che se la funzione è convessa il punto trovato è un minimo globale). Dato che non sappiamo poi qual è il tipo di punto, utilizziamo l'hessiana per definirlo, infatti a seconda di come essa è definita ci sarà un punto di sella, un minimo o un massimo locale. Il minimo deriva dal fatto che il determinante della matrice hessiana deve essere positivo insieme all'elemento nel punto (1,1).

Ora prendendo la direzione opposta del gradiente (perché punta alla massima variazione) e l'hessiana gli applichiamo il metodo di neton rapson che ricalcola il valore dell'hessiana ad ogni iterazione

Approssimazione di dati sperimentali

Quando un sistema lineare è sovradeterminato potrebbe non avere una soluzione esatta. In questi casi si introduce il vettore residuo $r(x) = Ax - b$. L'obiettivo diventa trovare il vettore x^* che minimizza la norma 2 al quadrato del residuo

$$x^* = \arg \min_{x \in R^n} \|Ax - b\|_2^2$$

Questa formulazione garantisce un problema ben posto, ovvero una soluzione unica e stabile. Il problema dei minimi quadrati può essere usato anche per approssimare dati sperimentali, ad esempio per trovare la retta di regressione $P_1(x) = a_0 + a_1x$, dove i coefficienti a_0 e a_1 vengono determinati minimizzando la somma dei quadrati degli errori rispetto ai punti sperimentali

Interpolazione di dati sperimentali

L'errore nell'interpolazione polinomiale viene analizzato con il TEOREMA dell'ERRORE, che afferma:

Se abbiamo una funzione $f(x)$ definita su un intervallo $[a,b]$ e i suoi valori nei nodi x_i sono $y_i = f(x_i)$, allora il polinomio interpolatore $P_n(x)$ di grado n che passa per questi punti ha un errore dato da:

$$E(x) = f(x) - P_n(x) = \frac{1}{(n+1)!} f^{(n+1)}(\xi) \omega_{n+1}(x)$$

Dove ε è un valore nell'intervallo $[a,b]$ e $w_{n+1}(x)$ è il prodotto $(x-x_0)(x-x_1)(x-x_2)\dots(x-x_n)$. Questo errore tende a zero se $f(x)$ è un polinomio di grado n , poiché in tal caso la derivata di ordine $n+1$ è nulla

Costante di Lebesgue e Nodi di Chebushev

Misura la sensibilità dell'interpolazione polinomiale alle perturbazioni nei dati. La funzione è:

$$\Lambda_n(x) = \sum_{i=0}^n |L_i(x)|$$

dove $L_i(x)$ sono i polinomi base di Lagrange. La costante di Lebesgue viene definita come:

$$\Lambda_n = \max_{x \in [a,b]} \Lambda_n(x)$$

Questa costante rappresenta il numero di condizionamento del problema di interpolazione. Se i nodi di interpolazione sono equidistanti, la crescita della costante è esponenziale, mentre se si usano i nodi di chebushev la crescita è logaritmica, rendendo l'interpolazione più stabile (perché minimizzano $w_{n+1}(x)$). Questi nodi si distribuiscono in maniera più densa agli estremi e meno densa al centro dell'intervallo.

Stabilità di algoritmi - condizionamento

Un algoritmo può essere stabile o instabile in base al condizionamento. Si parla di algoritmo ben condizionato se piccole perturbazioni nei dati portano piccole perturbazioni nei risultati, viceversa si dice che l'algoritmo è mal condizionato.

L'indice di condizionamento K è dato dal rapporto dell'errore relativo sui risultati con l'errore relativo sui dati:

$$\frac{\|f(x_p) - f(x)\|}{\|f(x)\|} \leq k * \frac{\|x_p - x\|}{\|x\|}$$

Con x_p e $f(x_p)$ valori perturbati.

Passaggi per trovare questa formula:

Essendo la perturbazione δ_x piccola,

$$\begin{aligned} f(\tilde{x}) - f(x) &\approx (\tilde{x} - x)f'(x) \\ \frac{f(\tilde{x}) - f(x)}{f(x)} &\approx \frac{(\tilde{x} - x)f'(x)}{f(x)} \\ \left| \frac{f(\tilde{x}) - f(x)}{f(x)} \right| &\approx \left| \frac{f'(x)x}{f(x)} \right| \left| \frac{\tilde{x} - x}{x} \right| \end{aligned}$$

Poniamo $K = \left| \frac{f'(x)x}{f(x)} \right|$, allora

$$\left| \frac{f(x + \delta_x) - f(x)}{f(x)} \right| \approx K \left| \frac{\tilde{x} - x}{x} \right|$$

Dove K è detto indice di condizionamento del problema della valutazione di una funzione

Per la stabilità di un algoritmo guardare se nella formula data ci sono dei valori che se vicini tra loro (a livello di valore) creano una differenza molto piccola (es. del tipo 0.00000000000003), che può essere persa a causa dell'arrotondamento macchina e quindi causare un'instabilità nell'algoritmo. Per trovare una versione stabile solitamente si fattorizza in modo da eliminare il problema.

RISPOSTE CROCETTE

- 1) **Quale affermazione è falsa riguardo alle reti MLP** -> le reti MLP sono in grado di apprendere solo relazioni lineari tra dati di input e output
- 2) **Qual è la definizione corretta di learning rate** -> un parametro che controlla la velocità con cui i pesi della rete vengono aggiornati durante il training
- 3) **Qual è il compito della funzione di attivazione in una rete neurale** -> introdurre non linearità nel flusso di informazioni della rete
- 4) **Che cosa rappresenta un'epoca** -> il numero di volte in cui l'intero dataset di dati di training viene esposto alla rete
- 5) **Qual è l'affermazione FALSA riguardo alla suddivisione del dataset** -> il set di validation viene utilizzato per trovare le etichette dei dati di input

Altra Teoria per approfondire i primi es:

GAUSS-JACOBI-GAUSS_SOR

TEOREMA 1 – CONVERGENZA e SOLUZIONE UNICA

Se il sistema $Ax = b$ ha un'unica soluzione x e il metodo iterativo

$$x^{(k)} = Tx^{(k-1)} + q$$

È convergente allora il limite della successione coincide con la soluzione esatta x

Spiegazione:

se l'iterazione converge, allora per ogni x_0 , gli x_k si avvicinano sempre di più a x . Il metodo iterativo ricostruisce la soluzione tramite una successione. La convergenza garantisce che si raggiunga il valore corretto nel limite

TEOREMA 2 – condizione necessaria e sufficiente alla convergenza

Sia $T = M^{-1}N$ la matrice di iterazione del metodo iterativo Il metodo converge per ogni vettore iniziale x_0 se e solo se $p(T) < 1$ dove $p(T)$ è il raggio spettrale, ovvero il modulo massimo degli autovalori T

Spiegazione:

Perché un metodo iterativo converga, gli errori devono ridursi ad ogni iterazione. Questo accade se la "potenza" della matrice T va a zero. Condizione chiave: tutti gli autovalori devono avere modulo < 1

TEOREMA 3 – condizione sufficiente alla convergenza

Se per una qualsiasi norma risulta $\|T\| < 1$ allora il metodo iterativo

$$x^{(k)} = Tx^{(k-1)} + M^{-1}b$$

Converge per ogni x_0

Spiegazione:

poiché gli autovalori sono sempre minori o uguali alla norma della matrice, se la norma è < 1 anche tutti gli autovalori lo saranno. È condizione più semplice da verificare

TEOREMA 4 - convergenza con matrice a diagonale strettamente dominante

Se la matrice A è a diagonale strettamente dominante allora sia il metodo di jacobi sia Gauss-seidel convergono inoltre

$$\|T_G\| \leq \|T_J\| < 1.$$

Spiegazione:

Una diagonale dominante garantisce che ogni equazione sia “ben bilanciata”, cioè che la componente x_i dipenda principalmente da sé stessa. Questo favorisce la stabilità e la convergenza del metodo

TEOREMA 5 – convergenza con matrice simmetrica e definita positiva

Se la matrice A è simmetrica e definita positiva allora il metodo di gauss seidel è convergente

Spiegazione:

Le matrici simmetriche definite positive hanno proprietà molto buone in analisi numerica: tutte le loro componenti sono ben comportate e le iterazioni si stabilizzano velocemente.

SISTEMI DIRETTI

TEOREMA ROUCHè – CAPELLI

Il sistema lineare $Ax = b$ ammette soluzioni se e solo se il rango della matrice dei coefficienti A coincide con il rango della matrice completa $[A|b]$ cioè:

$$\text{rank}(A) = \text{rank}([A \mid b])$$

Spiegazione:

Serve per capire se un sistema ha almeno una soluzione. Se il rango della matrice dei coefficienti cambia quando aggiungiamo il termine noto, vuol dire che il termine noto “rompe” l’equilibrio e il sistema non ha soluzioni. È il criterio fondamentale per decidere la compatibilità di un sistema

TEOREMA UNICA – SOLUZIONE

Il sistema $Ax = b$ ha una e una sola soluzione per ogni b se e solo se la matrice A è invertibile (cioè ha rango massimo). Allora $x = A^{-1}b$

Spiegazione

Questo è il criterio per sapere quando un sistema quadrato ha soluzione unica. Se A è invertibile ($\det(A) \neq 0$) possiamo calcolare l'inversa e risolvere direttamente. Vale anche per il caso omogeneo se $b = 0$ e l'unica soluzione sarà $x=0$

TEOREMA FATTORIZZAZIONE DI GAUSS, CHOLENSKY E QR -> sono i teoremi sulla divisione in sistemi

METODI DI DISCESA

TEOREMA ORTOGONALITÀ E CONIUGATEZZA DELLE DIREZIONI

Nel metodo del gradiente coniugato, i vettori

- $\mathbf{r}^{(k)}$ (residui) sono **mutuamente ortogonali**, cioè:

$$\langle \mathbf{r}^{(k)}, \mathbf{r}^{(j)} \rangle = 0 \quad \text{per ogni } k \neq j$$

- $\mathbf{p}^{(k)}$ (direzioni di discesa) sono **mutuamente coniugate rispetto ad A**, cioè:

$$\langle A\mathbf{p}^{(k)}, \mathbf{p}^{(j)} \rangle = 0 \quad \text{per ogni } k \neq j$$

Spiegazione:

Questa proprietà è la più importante per il metodo del gradiente coniugato: a ogni passi si generano direzioni NON solo diverse ma OTIMALI perché si “interferiscono” tra loro rispetto alla matrice A. Inoltre i residui sono perpendicolari tra loro, il che permette un’evoluzione più stabile e veloce nel metodo

TEOREMA CONVERGENZA IN N PASSI IN ASSENZA DI ERRORI

Se si lavora in aritmetica esatta (senza errori numerici) allora il metodo del gradiente coniugato converge esattamente alla soluzione $\mathbf{x}_{\text{star}} = \mathbf{A}^{-1}\mathbf{b}$ in AL PIU’ n passi, dove n è la dimensione del sistema

Spiegazione:

poiché il metodo costruisce una base di direzioni coniugate dopo n passi si è esplorato tutto lo spazio, si è quindi raggiunta esattamente la soluzione

Nella pratica a causa degli errori di arrotondamento, il metodo viene usato come iterativo ma spesso converge comunque molto rapidamente

TEOREMA VELOCITA’ DI CONVERGENZA

Nel caso di minimizzazione di una funzione quadratica

$$F(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x}$$

Con A simmetrica e definita positiva, il metodo del gradiente coniugato ha convergenza lineare

Spiegazione

La convergenza del metodo dipende da quanto è buona la matrice A. Se è ben condizionata (cioè con $k(A)$ piccolo) converge velocemente. Se è mal condizionata (cioè con $k(A)$ maggiore di 1) può servire un numero maggiore di iterazioni. Comunque il CG è più veloce del metodo di discesa del gradiente semplice (lo steepest)