Sotto quali condizioni, il metodo di discesa del gradiente con passo fisso converge a un punto stazionario della funzione costo, che può essere un minimo globale se la funzione è convessa?

Le condizioni sono:
- La funzione costo è convessa e differenziabile.
- Il Learning Rate è adeguatamente scelto.
- Il gradiente della funzione costo è Lipschitz-continuo.

Abbiamo visto che il Gradient Descent aggiorna i parametri (pesi e bias) con

$$w^{(k+1)} = w^{(k)} - \eta \nabla C(w^{(k)})$$

w è il vettore dei parametri e η è il learning rate.

→ La discesa del gradiente trova il minimo globale se la funzione è convessa.

Vale il seguente risultato

Definizione: Una funzione $f: R^n \rightarrow R$ è detta **lipschitziana** se esiste una costante L > 0 tale che per ogni coppia di punti x, y in R^n valga la seguente disuguaglianza:

$$\left|\left|f(x) - f(y)\right|\right| \leq L \, ||x - y||$$

Quindi, funzione lipschitziana ha una "crescita limitata": la variazione del suo valore (la differenza tra i valori in due punti) non può mai superare un valore proporzionale alla distanza tra i due punti, moltiplicata per la costante L.

Lipschitzianità del gradiente:

Si dice che il gradiente di una funzione f è lipschitziano con costante L se la seguente disuguaglianza vale per ogni coppia di punti $x, y \; in \; R^n$:

$$\left||\nabla f(x) - \nabla f(y)|\right| \leq L \, ||x - y||$$

Teorema

della funzione costo

Ipotizziamo che $f: R^n \; -> \; R$ sia una **funzione convessa** e differenziabile con gradiente ∇f(x) continuo Lipschitz con costante L, e che x^* tale che $\nabla f(x^*)$=0. Consideriamo l'iterazione del metodo della discesa del gradiente (GD).

$$x^{(k+1)} = x^{(k)} - \eta \nabla f(x^{(k)})$$

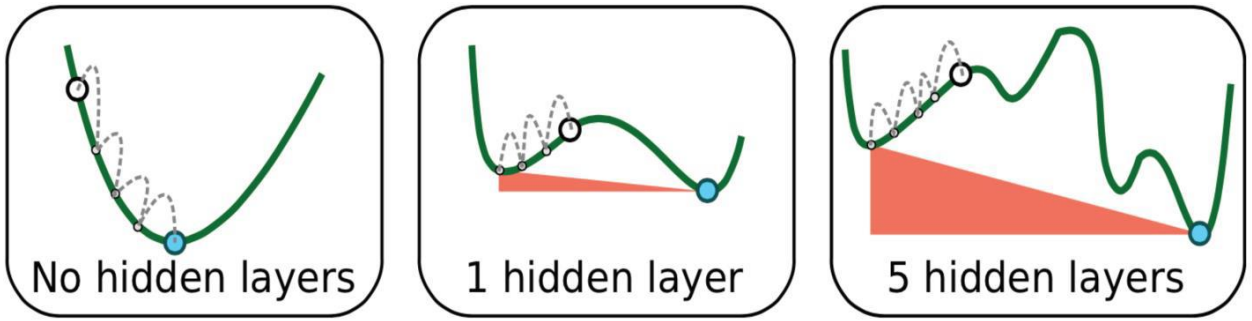
dove

$$0 < \eta < \frac{2}{L}$$

allora la sequenza di iterati $x^{(k)}$ converge ad x^*

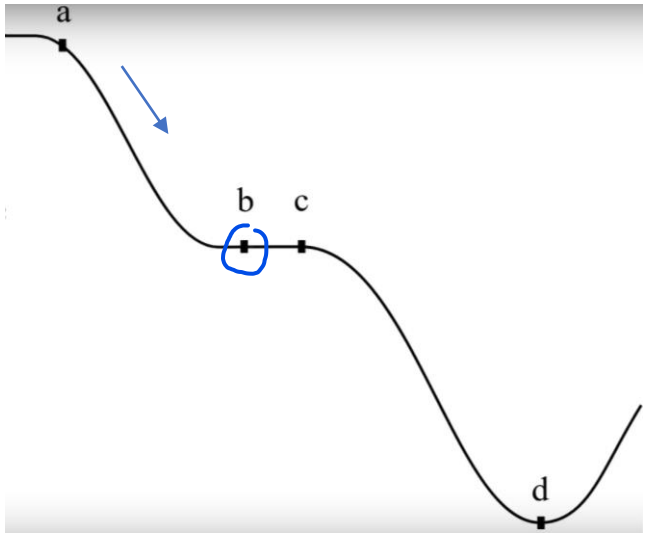
Sfortunatamente, quasi tutti i problemi di ottimizzazione che sorgono nell'allenamento di reti neurali sono non convessi

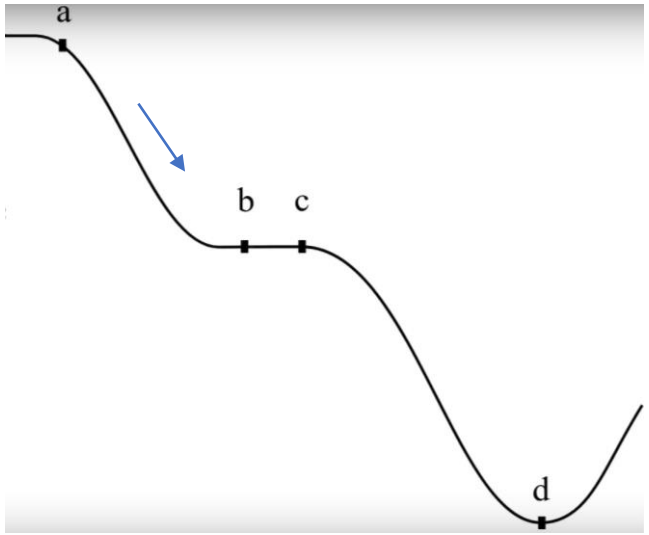
• introducendo la non linearità nella rete (aggiungendo strati nascosti), la funzione costo diventa non convessa e compaiono i minimi locali

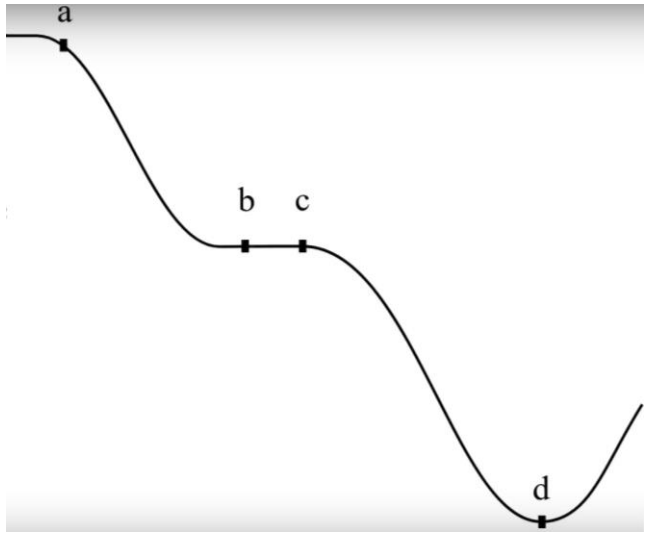


Consideriamo questa curva, (che rappresenta una funzione costo, per semplicità 1d)

Inizializziamo il peso w ad un punto a ed applichiamo il metodo iterativo del gradiente.





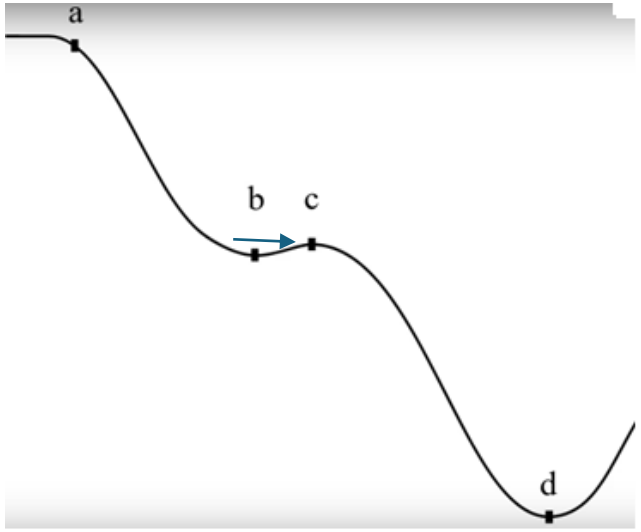


w raggiunge il punto b e si ferma, poiché in b il gradiente si annulla, la funzione è caratterizzata da una zona piatta, un *plateau* (*vanishing gradient*).

Se invece di una regione piatta, in b c'è un minimo locale, w non può superare il massimo locale nel punto c e rimane bloccato in b.

In entrambi i casi non può raggiungere il minimo locale in d.

Simili situazioni si possono verificare nelle superfici multidimensionali. Specialmente nei punti sella dove il gradiente è zero, ma il punto non è un minimo oppure un massimo.

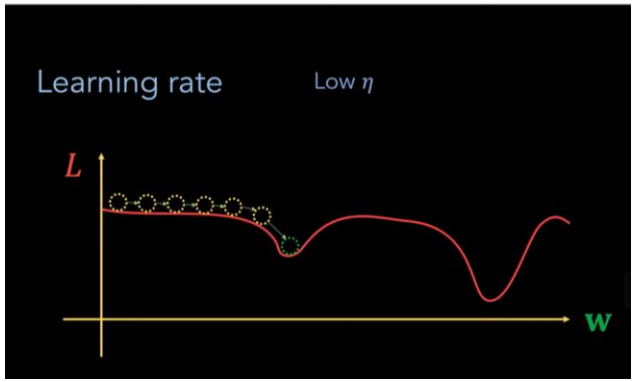


Importanza del Learning rate

Valori bassi di η

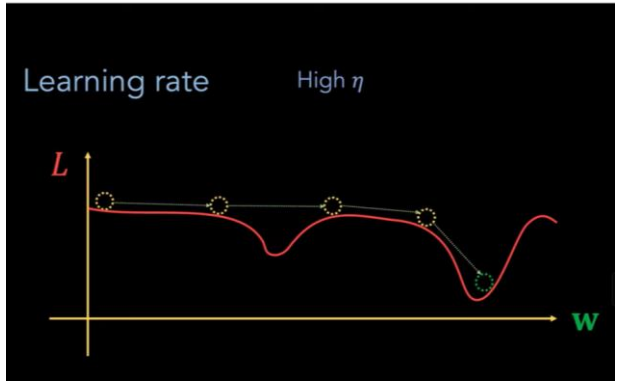
- possono far sì che sia necessario un numero elevato di passi prima che l'allenamento sia completato.

- possono far sì che i pesi rimangano bloccati in un minimo locale, che è sub-optimal se confrontato con il global minimum.

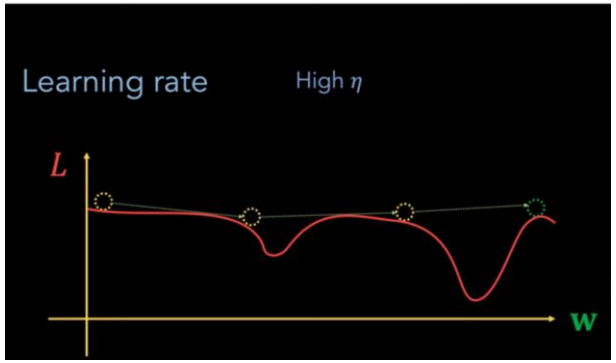


Valori alti di η

Valori alti di learning rate permettono di mitigare questi problemi passando oltre i minimi locali e raggiungendo il minimo in numero limitato di passi.



i pesi possono anche finire per superare il minimo target

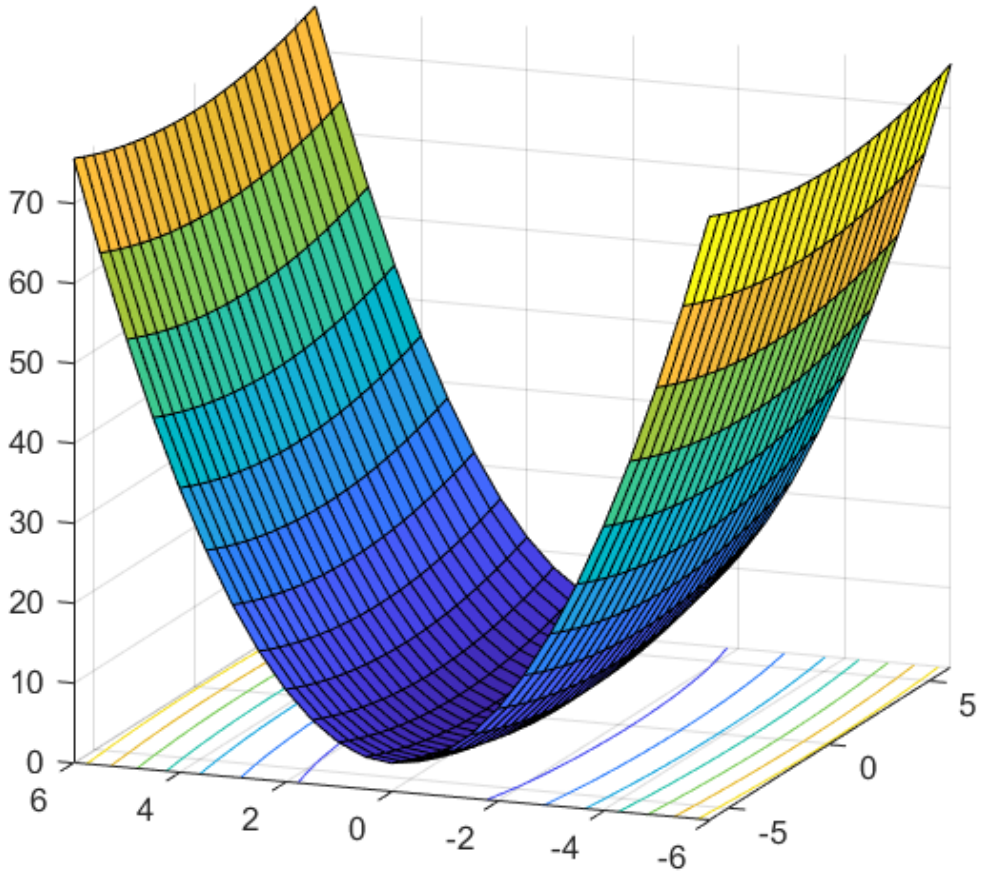


→ Queste osservazioni fanno intuire che la scelta del learning late è cruciale ed in pratica può richiedere molta sperimentazione per scoprire quale possa essere il valore ottimale del learning rate.

Consideriamo una loss function del tipo

$$f(w_1, w_2) = 0.1\, w_1^2 + 2\, w_2^2$$

- ha il suo minimo in (0,0) .
- è *molto* piatta nella direzione di w_1.
-
- $\nabla f(w_1, w_2) = \begin{bmatrix} \frac{\partial f(w_1,w_2)}{\partial w_1} \\ \frac{\partial f(w_1,w_2)}{\partial w_2} \end{bmatrix} = \begin{bmatrix} 0.2\, w_1 \\ 4 w_2 \end{bmatrix}$



Vediamo cosa succede quando eseguiamo la discesa del gradiente per calcolarne il minimo. Scegliamo un learning rate di $\eta = 0.4$.

$\boldsymbol{w_1^{(0)} = -5,\ w_2^{(0)} = -2,}$

$f\left(w_1^{(0)}, w_2^{(0)}\right) = -10.5$

$w_1^{(1)} = w_1^{(0)} - \eta \frac{\partial f}{\partial w_1}\left(w_1^{(0)}, w_2^{(0)}\right) =$

$= -5 - 0.4(0.2 \cdot -5) = -4.6$

$w_2^{(1)} = w_2^{(0)} - \eta \frac{\partial f}{\partial w_2}\left(w_1^{(0)}, w_2^{(0)}\right) =$

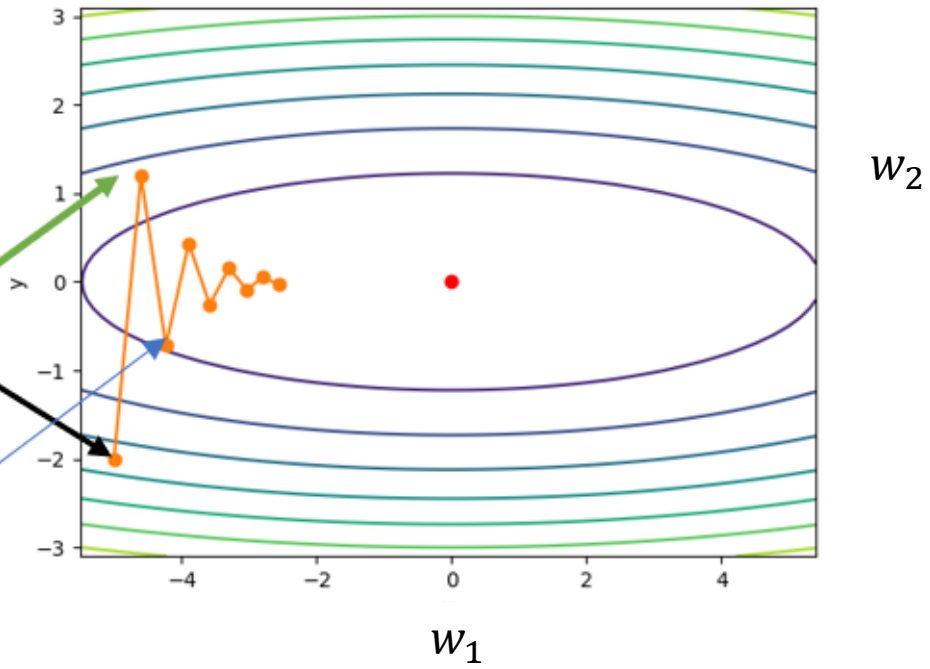
$-2 - 0.4\left(4 \cdot (-2)\right) = 1.2$

$\boldsymbol{w_1^{(1)} = -4.6,\ w_2^{(1)} = 1.2,}$

$f\left(w_1^{(1)}, w_2^{(1)}\right) = 5$

$\boldsymbol{w_1^{(2)} = -4.2,\ w_2^{(2)} = -0.7,}$

$f\left(w_1^{(2)}, w_2^{(2)}\right) = 2.7$



Per come è definita la funzione costo $f(w_1, w_2)$ il gradiente nella direzione w_2 è molto più alto e cambia molto più rapidamente rispetto alla direzione orizzontale.
Si possono verificare due situazioni in base alla scelta del learning rate:

- Se scegliamo un learning rate basso la soluzione non diverge nella direzione verticale w_2 , ma c'è una lenta convergenza nella direzione orizzontale w_1.

- Con un elevato learning rate progrediamo rapidamente nella direzione orizzontale w_1 , ma c'è divergenza nella direzione verticale w_2

Dopo 20 epoche w_1 : -0.943467,w_2: -0.000073

Per risolvere questi problemi si introduce **il gradient descent con momentum.**

Gradient Descent con Momentum
Il momentum aiuta a risolvere questi problemi aggiungendo una componente che tiene conto della direzione e della velocità di aggiornamento dei pesi dai passi precedenti. Questo consente di avere aggiornamenti più stabili e veloci.

Vantaggi del Gradient Descent con Momentum
- Accelera la Convergenza: Grazie all'accumulo delle variazioni passate, l'algoritmo può superare rapidamente le regioni piatte della funzione di costo, convergendo più velocemente lungo direzioni lente (come la direzione w1).
- Stabilizza gli Aggiornamenti: Il momentum aiuta a smorzare le oscillazioni lungo direzioni ripide della funzione di costo (come la direzione w2) (ridurre o eliminare le oscillazioni dei parametri attorno al minimo della funzione di costo), riducendo la probabilità di divergenza e oscillazioni.
- Superamento dei Minimi Locali: L'accumulo del momento può aiutare l'algoritmo a superare i minimi locali, fornendo una spinta nelle direzioni giuste.

Gradient Descent con Momentum

Il Gradient Descent (Discesa del gradiente) è un algoritmo di ottimizzazione ampiamente utilizzato per addestrare reti neurali e altri modelli di apprendimento automatico.
Tuttavia, l'aggiornamento dei parametri durante la discesa del gradiente può essere lento e presentare oscillazioni, soprattutto per funzioni di costo con curvature dolci o valli poco profonde.

Momentum è una tecnica utilizzata per accelerare la convergenza e ridurre le oscillazioni durante l'allenamento con Gradient Descent.

Formula di aggiornamento dei pesi :

$$\boldsymbol{w^{(k+1)} = w^{(k)} - \eta v^{(k)}}$$

$$\boldsymbol{v^{(k)} = \beta v^{(k-1)} + \nabla C\left(w^{(k)}\right)}$$

β prende il nome di **momentum** e può variare tra 0 ed 1. v_0 vettore inizializzato a zero.

Se si pone $\beta = 0$, si ricade nel **classico metodo del gradiente**. Un valore appropriato è tra 0.8 e 0.9.

Analizziamo il termine

$$\boldsymbol{v}^{(\mathrm{k})} = \boldsymbol{\beta v}^{(\boldsymbol{k-1})} + \nabla \boldsymbol{C}\left(\boldsymbol{w}^{(\boldsymbol{k})}\right)$$

$$v^{(1)} = \beta \boldsymbol{v}^{(\mathbf{0})} + \nabla C(w^{(1)})$$

$$v^{(2)} = \beta \boldsymbol{v}^{(\mathbf{1})} + \nabla C(w^{(2)})$$

$$v^{(2)} = \beta\left(\beta \boldsymbol{v}^{(\mathbf{0})} + \nabla C(w^{(1)})\right) + \nabla C(w^{(2)}) = \beta^2 \boldsymbol{v}^{(\mathbf{0})} + \beta \nabla C(w^{(1)}) + \nabla C(w^{(2)})$$

$$v^{(3)} = \beta \boldsymbol{v}^{(\mathbf{2})} + \nabla C(w^{(3)})$$

$$= \beta\left(\beta^2 \boldsymbol{v}^{(\mathbf{0})} + \beta \nabla C(w^{(1)}) + \nabla C(w^{(2)})\right) + \nabla C(w^{(3)}) = \beta^3 \boldsymbol{v}^{(\mathbf{0})} + \beta^2 \nabla C(w^{(1)}) + \beta \nabla C(w^{(2)} + \nabla C(w^{(3)})$$

$$v^{(k)} = \beta^k \boldsymbol{v}^{(\mathbf{0})} + \beta^{k-1} \nabla C(w^{(1)}) + \cdots + \beta \nabla C(w^{(k-1)}) + + \beta^0 \nabla C\left(w^{(k)}\right)$$

Posto $v_0 = 0$

$$v^{(k)} = \sum_{j=1}^{k} \beta^{k-j} \nabla C\left(w^{(j)}\right)$$

Si basa sullo stesso concetto di **momento** , *quantità di moto in fisica*. Un classico esempio del concetto è una palla che rotola giù da una collina che raccoglie abbastanza slancio per superare una regione di altopiano e raggiungere un minimo globale invece di rimanere bloccata in un minimo locale.

"Aumentano le dimensioni degli aggiornamenti" significa che il momentum, accumulando gradienti passati, amplifica l'aggiornamento complessivo dei pesi quando i gradienti sono consistenti.

Momentum

Il ~~termine momento~~ aumenta le dimensioni degli aggiornamenti quando i gradienti puntano nella stessa direzione mentre riduce le dimensioni degli aggiornamenti quando i gradienti cambiano direzione.

In questo modo si ottiene una più veloce convergenza e si riduce l'oscillazione.

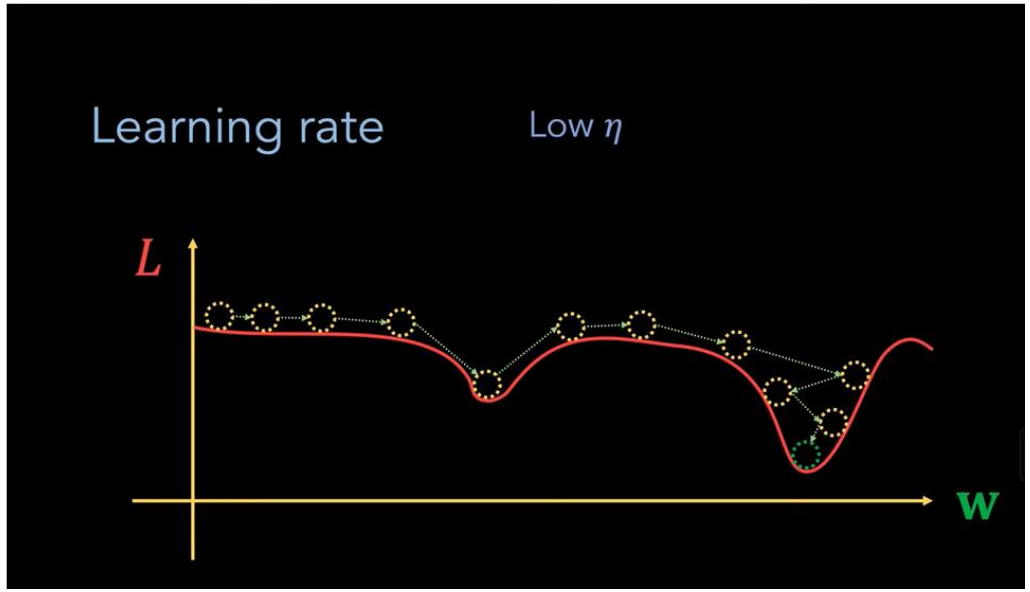
In questo modo la scelta del learning rate è meno cruciale poiché la procedura di training si adatta alla particolare loss function.

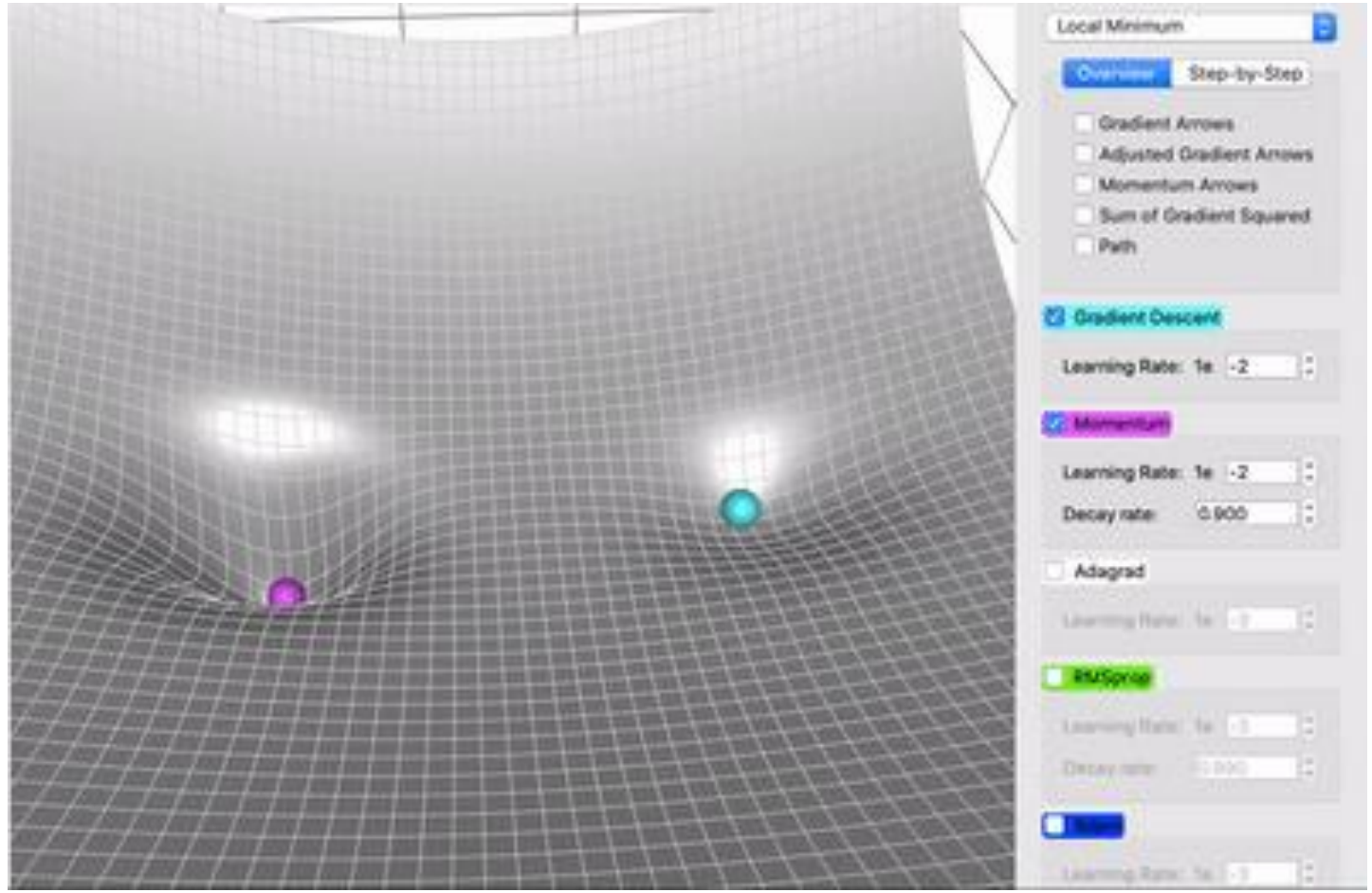
Quando i gradienti successivi puntano nella stessa direzione (cioè i segni dei gradienti non cambiano tra gli aggiornamenti consecutivi), il momentum accumula questi gradienti, aumentando l'aggiornamento complessivo dei pesi. Questo si traduce in passi più grandi lungo quella direzione.

Quando i gradienti cambiano direzione tra gli aggiornamenti consecutivi (cioè i segni dei gradienti cambiano), il termine del momentum agisce per smorzare questi cambiamenti, riducendo l'aggiornamento complessivo dei pesi. Questo si traduce in passi più piccoli lungo quella direzione.

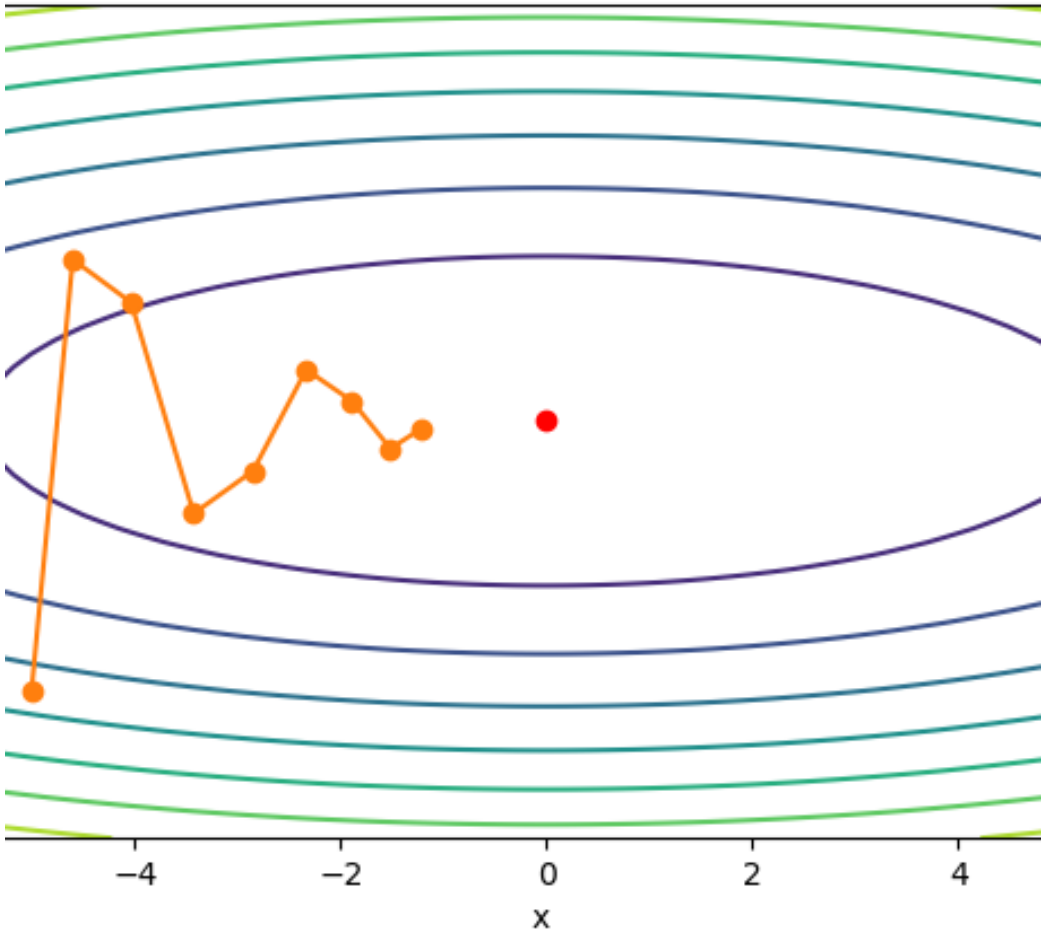
Frequenza degli Aggiornamenti
Questi passaggi si ripetono ad ogni iterazione del ciclo di ottimizzazione, che può essere:

Per ogni batch: Nel caso di mini-batch gradient descent, l'aggiornamento dei pesi avviene dopo aver processato ogni mini-batch.
Per ogni esempio: Nel caso di stochastic gradient descent (SGD), l'aggiornamento dei pesi avviene dopo aver processato ogni singolo esempio.
Per ogni epoca: Nel caso di batch gradient descent, l'aggiornamento dei pesi avviene dopo aver processato l'intero set di dati.




Local Minimum
Overview
Step-by-Step
Gradient Arrows
Adjusted Gradient Arrows
Momentum Arrows
Sum of Gradient Squared
Path
Gradient Descent
Learning Rate: 1e -2
Momentum
Learning Rate: 1e -2
Decay rate: 0.900
Adagrad
RMSprop

- Tornando al nostro esempio, il metodo del gradiente con momento produrrà le seguenti iterazioni:



- nella direzione w_1 il metodo del gradiente con momento accumulerà gradienti ben allineati, aumentando così la distanza che percorriamo ad ogni iterazione.
- Al contrario, nella direzione w_2 in cui i gradienti sono oscillanti, l'accumulo dei gradienti ridurrà la dimensione del passo a causa delle oscillazioni che si annullano a vicenda.

Direzione w_1 :

Gradienti ben allineati: Se i gradienti nella direzione w_1 sono ben allineati, l'accumulo di gradienti con Momentum amplifica la loro direzione, aumentando la distanza percorsa ad ogni passo.

Il termine di momentum conserva la "memoria" dei gradienti passati, favorendo la continuità del movimento nella stessa direzione. Se i gradienti sono allineati, l'accumulo li rafforza, permettendo al modello di avanzare più rapidamente lungo questa traiettoria.

Direzione w_2 :

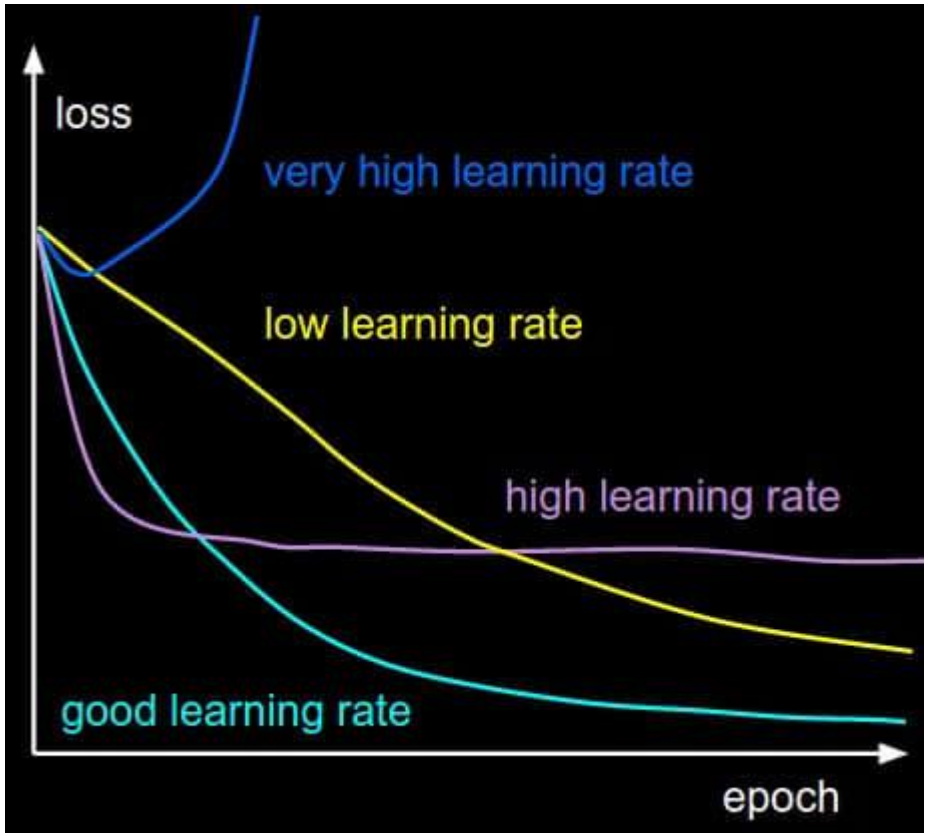
Gradienti oscillanti: Se i gradienti nella direzione w_2 oscillano, l'accumulo di gradienti con Momentum tende a ridurre la dimensione del passo, smorzando le oscillazioni.

Il termine di momentum media i gradienti passati, attenuando l'effetto di quelli che si annullano a vicenda a causa delle oscillazioni. Questo aiuta a stabilizzare il movimento e a impedire che il modello si sposti troppo in una direzione e poi torni indietro.

Learning rate adattivo

Uno degli iperparametri più difficili da regolare nelle reti neurali è il **learning rate.**

- La funzione costo diminuisce ma richiede molto più tempo per converge (**learning rate basso**, curva gialla)
- La funzione costo raggiunge un valore migliore di quello iniziale,ma è ancora lontano da un valore ottimale (**Learning rate alto**: curva viola)
- La funzione costo inizialmente diminuisce poi inizia ad aumentare (**Learning Rate molto alto**, curva blu)
- La funzione costo diminuisce costantemente fino a raggiungere il valore minimo possibile (**Learning rate buono**, curva azzurra)



Effetto di differenti learning rate sulla minimizzazione della funzione costo.

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 dai 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

Le **tecniche di aggiornamento del learning rate** cercano di regolarlo durante **l'allenamento riducendolo in accordo ad uno schema predefinito.**

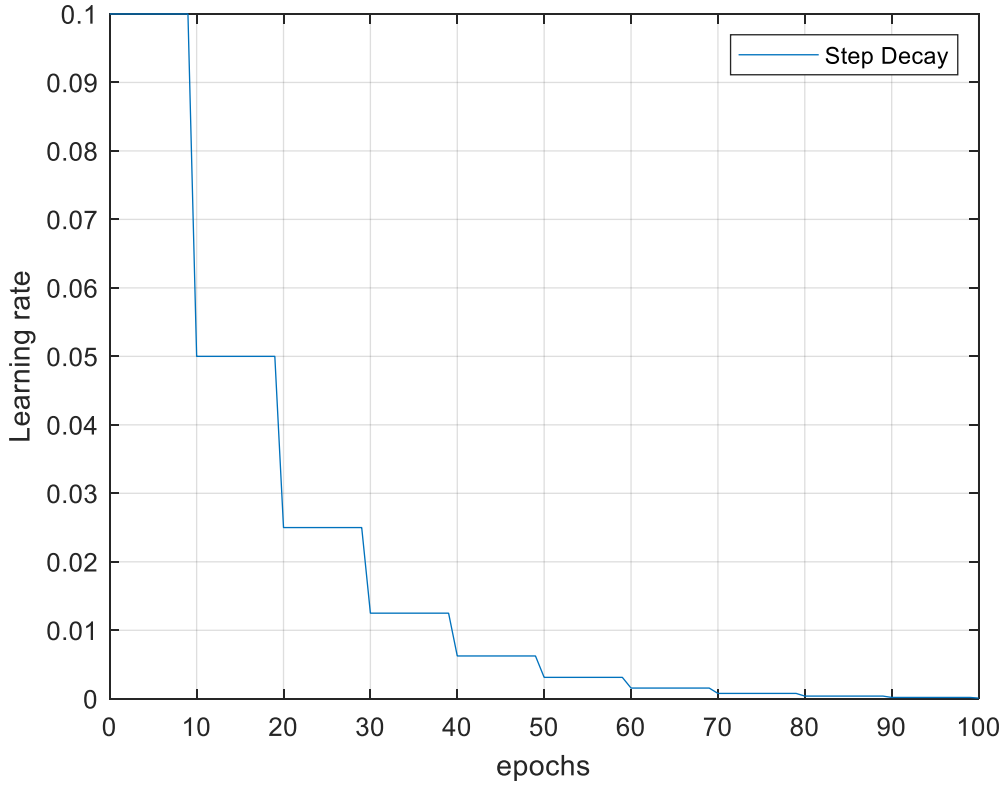
Un LR troppo alto può portare a oscillazioni instabili e impedire alla rete di convergere verso una soluzione ottimale.

Al contrario, un LR troppo basso può rallentare eccessivamente l'allenamento e richiedere un tempo di calcolo computazionalmente costoso.

Per questo motivo, **le tecniche di decadimento del learning rate (LR decay) sono impiegate per regolare il LR durante l'allenamento.** Queste tecniche riducono gradualmente il LR nel tempo, cercando un equilibrio tra velocità di convergenza e stabilità.

Sfide nella scelta del momento giusto per il decadimento (decay)

Decidere quando e come far decadere il LR è una sfida cruciale nell'allenamento delle rete neurali. Se il decadimento avviene troppo lentamente, si rischia di sprecare tempo di calcolo con piccoli miglioramenti incrementali. Al contrario, un decadimento troppo aggressivo può raffreddare prematuramente la rete, impedendole di raggiungere il minimo globale o una soluzione ottimale.



$\eta_0 = 0.1 \quad s = 10 \quad \delta = 0.5$

In questo caso, ogni 10 epoche, cambio il LR

Strategie comuni per il decadimento del learning rate:

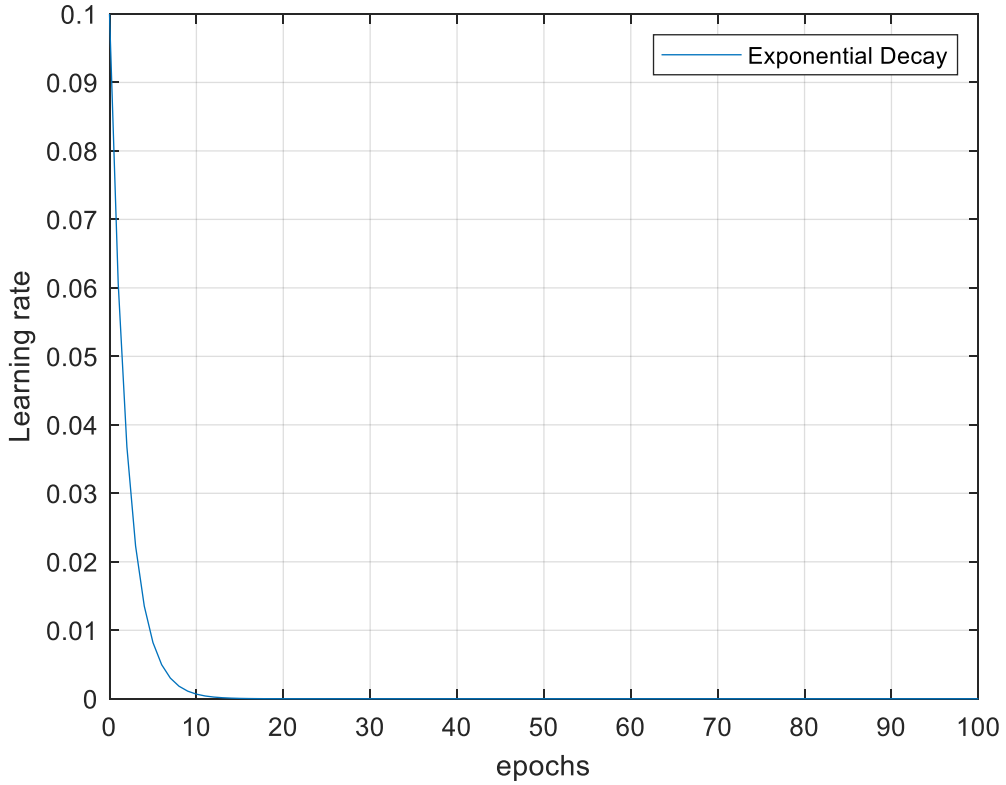
- Esistono tre tipi comuni di implementazione del decadimento del learning rate:

- **Step decay** : riduce il Learning rate iniziale η_0 di un fattore δ ogni numero predefinito di epoche s

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

- dove η_0 e δ e s sono iperparametri e n è l'iterazione corrente
- $\lfloor x \rfloor$ (**floor:** associa ad ogni numero reale x il più grande intero **minore o uguale a x**)

Avevamo visto il ceil che invece associava ad ogni numero reale x il piú grande intero maggiore o uguale a x



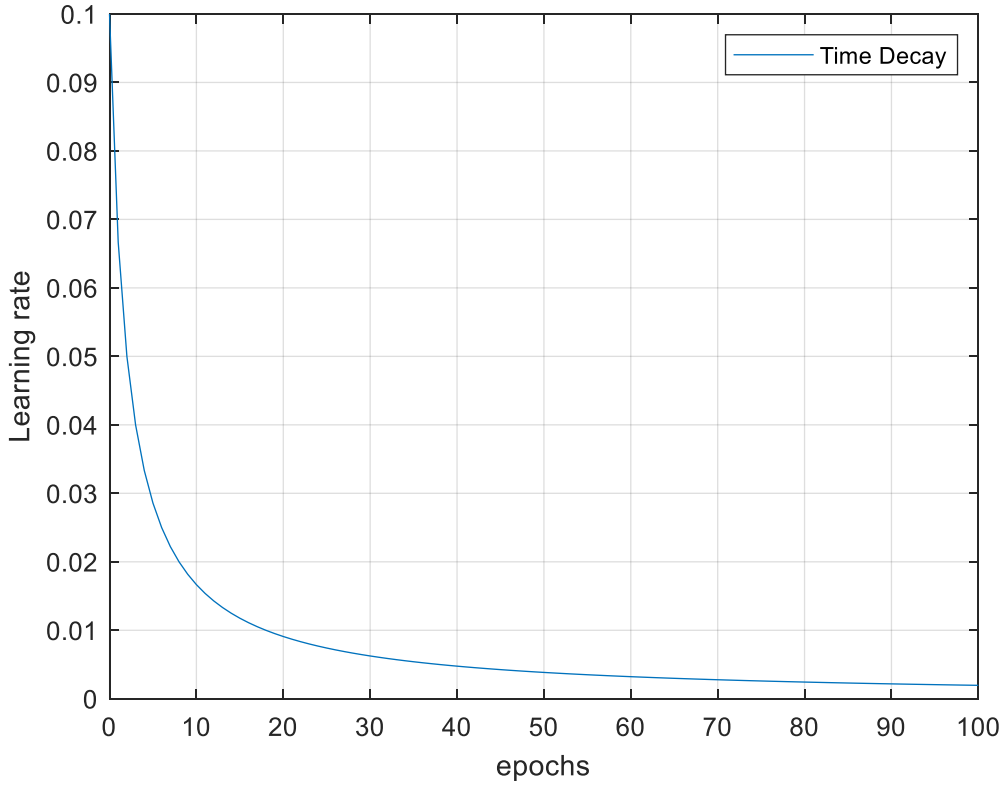
$\eta_0 = 0.1 \quad \delta = 0.5$

Strategie comuni per il decadimento del learning rate:

- **decadimento esponenziale** ha la forma matematica

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

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



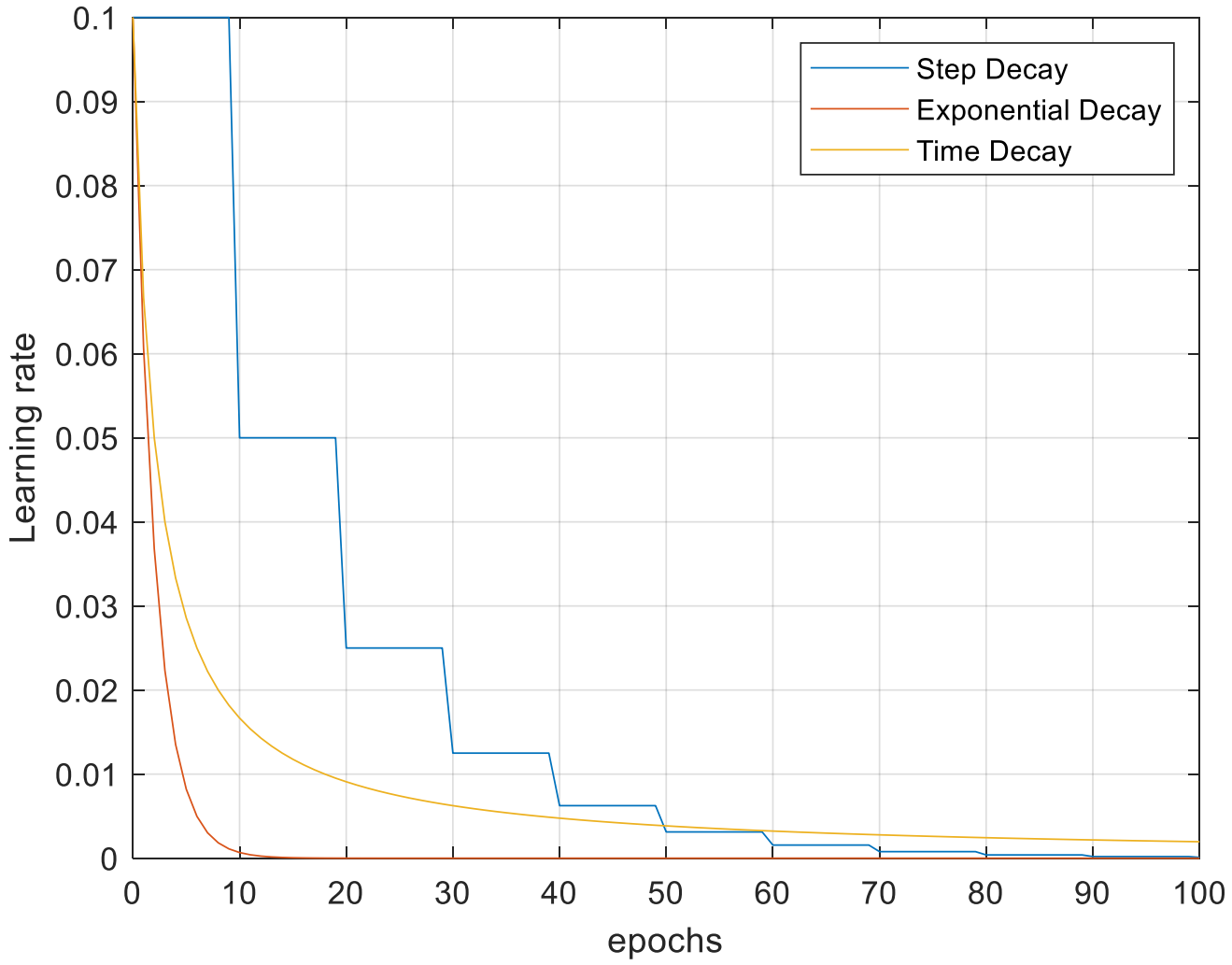
$\eta_0 = 0.1 \quad \delta = 0.5$

Strategie comuni per il decadimento del learning rate:

- **decadimento basato sul tempo:** modifica il learning rate iniziale η_0 in funzione del numero di iterazioni eseguite (n)

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

dove η_0 e δ sono iperparametri


Step Decay
Exponential Decay
Time Decay
Learning rate
epochs
0.1
0.09
0.08
0.07
0.06
0.05
0.04
0.03
0.02
0.01
0
0
10
20
30
40
50
60
70
80
90
100

L'utilizzo del warm-up per l'inizializzazione casuale dei parametri

L'inizializzazione **casuale dei parametri** è una tecnica comune per l'avvio dell'allenamento di una rete neurale. Tuttavia, questa tecnica può presentare alcuni problemi, in particolare quando si utilizzano learning rate elevati all'inizio dell'allenamento.

Problemi con l'inizializzazione casuale e learning rate elevati:

Divergenza: Un learning rate elevato può amplificare gli errori iniziali e causare una rapida crescita dei valori dei parametri, portando alla divergenza della rete neurale.

Lentezza del progresso: Un learning rate troppo basso, necessario per evitare la divergenza, può rallentare notevolmente il progresso dell'allenamento.

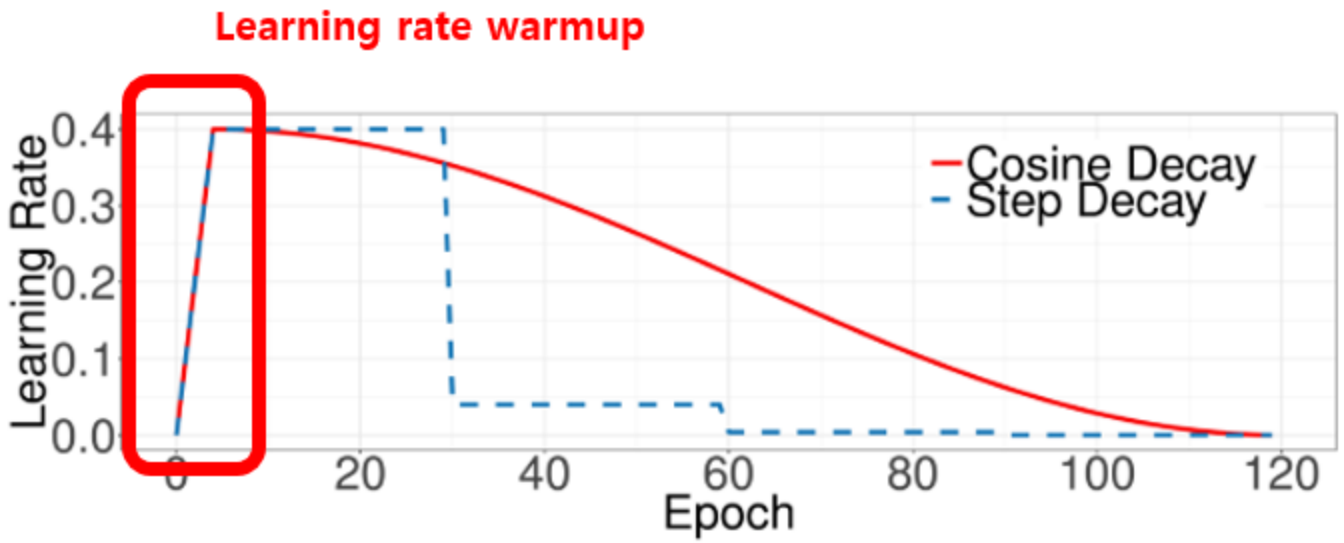
Soluzione: warm-up del learning rate

Il warm-up del learning rate è una tecnica che affronta i problemi sopracitati **introducendo un periodo iniziale di allenamento con un learning rate molto inferiore rispetto al valore finale desiderato**. Questo periodo di **warm-up** consente alla rete di adattarsi gradualmente ai parametri iniziali casuali senza causare divergenza e prepara il terreno per un allenamento più efficiente con un learning rate più alto.

Il Warm Up del Learning Rate è una tecnica utilizzata nell'addestramento delle reti neurali per iniziare con un learning rate basso e aumentarlo gradualmente durante le prime fasi dell'allenamento fino a raggiungere il learning rate desiderato. Questa strategia può migliorare la stabilità e la convergenza del modello.

Come funziona il warm-up del learning rate:

Inizializzazione: I parametri della rete neurale vengono inizializzati casualmente.
Warm-up: Si inizia con un learning rate molto basso (ad esempio, 10 volte o 100 volte inferiore rispetto al learning rate finale desiderato).
Aumento graduale del learning rate: Durante un periodo di warm-up predefinito (ad esempio, un certo numero di iterazioni o epoche), il learning rate viene gradualmente aumentato fino a raggiungere il valore finale desiderato.
Allenamento con learning rate e decay : Una volta raggiunto il learning rate finale, si riduce con una tecnica di decay fino alla fine del processo di ottimizzazione.



(a) Learning Rate Schedule

Le regole di aggiornamento del learning rate, come il decadimento a step o esponenziale, sono ampiamente utilizzati nell'allenamento delle reti neurali. Tuttavia, **presentano alcune limitazioni**:

- **Mancanza di adattabilità**: non si adattano alle caratteristiche specifiche del modello o dei dati, **richiedendo la definizione manuale di iperparametri che potrebbero non essere ottimali.**
- **Aggiornamenti uniformi:** applicano **lo stesso learning rate a tutti i parametri della rete**, non considerando la diversa importanza o velocità di convergenza di ciascun peso.
- **Inefficienza per dati sparsi:** Nei casi di dati sparsi, dove alcuni pesi interagiscono con pochi dati, un learning rate fisso può portare a un aggiornamento eccessivo di questi pesi, ostacolando la convergenza.

L'inefficienza per dati sparsi, quando si utilizza un learning rate fisso durante l'allenamento delle reti neurali, si riferisce ai problemi che possono sorgere in situazioni in cui solo una piccola porzione dei dati contribuisce all'aggiornamento di determinati pesi.

Metodi di aggiornamento adattivo del learning rate

Per superare queste limitazioni, sono stati sviluppati metodi di aggiornamento adattivo del learning rate che **modificano dinamicamente il learning rate per ciascun peso durante l'allenamento.** Questi metodi si basano su informazioni accumulate durante il processo di apprendimento per ottimizzare il ritmo di aggiornamento dei pesi.

Quattro metodi rappresentativi:

1. **Adagrad:**
2. **RMSProp**
3. **Adadelta**
4. **Adam:**

Adagrad (Adaptive Gradient Algorithm)

- Adagrad **adatta il Learning Rate ai parametri**, eseguendo aggiornamenti più grandi per i parametri poco frequenti e aggiornamenti più piccoli per quelli frequenti.

- Adatta il learning rate per l'aggiornamento individuale di ciascun parametro in proporzione alla sua cronologia di aggiornamento, più aggiornamenti più decadimento nel learning rate:

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

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{s^{(k)} + \varepsilon}} \nabla C\left(w^{(k)}\right)$$

ε>0 serve per evitare che il denominatore si annulli nel caso in cui $s^{(k)}$ sia nullo.

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 infinitesimamente piccolo fino a quando la rete non è più in grado di acquisire ulteriore conoscenza:

Dopo molte iterazioni, il learning rate effettivo può diventare così piccolo che il modello smette di apprendere.

I parametri meno frequenti si riferiscono a quei parametri di un modello di machine learning che vengono aggiornati con minore frequenza durante il processo di addestramento.

Questo può dipendere da diverse ragioni, tra cui:

Architettura del modello: In alcune architetture di reti neurali, alcuni parametri sono coinvolti in calcoli meno frequenti rispetto ad altri. Ad esempio, in una rete convoluzionale, i parametri dei layer convoluzionali tendono ad essere aggiornati più frequentemente rispetto ai parametri dei layer di pooling.

Dataset: In alcuni dataset, alcuni parametri sono associati a caratteristiche che appaiono meno frequentemente nei dati di training. Ad esempio, in un dataset di immagini di animali, i parametri associati a caratteristiche come le macchie sulle giraffe potrebbero essere aggiornati meno frequentemente rispetto ai parametri associati a caratteristiche più comuni come la forma del corpo o il colore del mantello.

Effetti di Adagrad:

Aggiornamenti più grandi per parametri rari: I parametri che vengono aggiornati meno frequentemente hanno un accumulatore di gradienti quadrati più piccolo, risultando in un learning rate normalizzato più alto e aggiornamenti più grandi.

Aggiornamenti più piccoli per parametri frequenti: I parametri che vengono aggiornati più frequentemente hanno un accumulatore di gradienti quadrati più grande, risultando in un learning rate normalizzato più piccolo e aggiornamenti più piccoli.

Riduzione del rischio di sovrastima: Adagrad aiuta a prevenire la sovrastima dei parametri, in particolare quelli con gradienti elevati, regolando il learning rate in base alla loro storia di aggiornamento.

Vantaggi di Adagrad:

- **Adattabilità:** Adagrad si adatta automaticamente alle caratteristiche dei dati e del modello, ottimizzando il learning rate per ciascun parametro.
- **Efficienza:** Può accelerare la convergenza in alcuni casi, soprattutto per modelli con parametri rari o con gradienti elevati.
- **Robustezza:** Può essere utile per problemi di ottimizzazione con vincoli di sparsità o con dati rumorosi.

Svantaggi di Adagrad:

- **Accumulo di gradiente quadrato:** L'accumulatore di gradiente quadrato può crescere senza limiti, portando a un learning rate troppo piccolo e a una convergenza lenta o stagnante.
- **Sensibilità ai gradienti iniziali:** Può essere influenzato dai valori iniziali dei parametri, portando a prestazioni subottimali se i parametri iniziali sono mal scelti.

RMSProp

RMSProp (Root Mean Square Propagation) è un algoritmo di ottimizzazione per l'apprendimento automatico che deriva da Adagrad e ne migliora alcune limitazioni.

Come Adagrad, RMSProp **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

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

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

$$w^{(k+1)} = w^{(k)} - \frac{\eta}{\sqrt{s^{(k)} + \varepsilon}} \nabla C\left(w^{(k)}\right)$$

Effetti di RMSProp:

•**Riduzione del decadimento aggressivo:** La media ponderata esponenziale attenua l'effetto dei gradienti passati, impedendo un decadimento troppo rapido del learning rate come può accadere con Adagrad.

•**Maggiore stabilità:** RMSProp tende ad essere più stabile di Adagrad e meno incline a oscillazioni o divergenze durante l'allenamento.

•**Convergenza più affidabile:** Può aiutare a ottenere una convergenza più affidabile e uniforme su un'ampia gamma di problemi di ottimizzazione.

Vantaggi di RMSProp:

- **Migliora la stabilità:** Rispetto ad Adagrad, RMSProp offre una maggiore stabilità e riduce il rischio di oscillazioni o divergenze.
- **Convergenza più affidabile:** Può facilitare una convergenza più uniforme e affidabile su diverse tipologie di problemi.
- **Meno sensibile ai gradienti iniziali:** È meno sensibile ai valori iniziali dei parametri rispetto ad Adagrad.

Svantaggi di RMSProp:

- **Potenziale accumulo di errore:** La media ponderata esponenziale può accumulare errori nel tempo, soprattutto per modelli con gradienti rumorosi.
- **Scelta del fattore di smorzamento:** Il fattore di smorzamento γ deve essere scelto attentamente per ottenere il miglior bilanciamento tra stabilità e velocità di convergenza.

ADADELTA

Adadelta è un'altra variante di Adagrad proposta per superare il suo principale inconveniente

Come RMSProp calcola l'accumulo del gradiente come una media ponderata esponenziale dei gradienti al quadrato ma, a differenza di RMSProp, **non richiede di impostare un learning rate** in quanto utilizza la quantità di cambiamento stessa come calibrazione per il cambiamento futuro

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

$$\breve{\nabla}\, C\left(w^{(k)}\right) = \frac{\sqrt{\Delta w^{(k-1)}+\epsilon}}{\sqrt{s^{(k)}+\epsilon}} \nabla C\left(w^{(k)}\right)$$ **Gradiente pesato (riscalato)**

$$w^{(k+1)} = w^{(k)} - \breve{\nabla}\, C\left(w^{(k)}\right)$$ **Aggiornamento dei pesi**

$$\Delta w^{(k)} = \gamma \Delta w^{(k-1)} + (1-\gamma)\left(\breve{\nabla}\, C\left(w^{(k)}\right)\right)^2$$ **Media ponderata esponenziale dei quadrati dei gradienti riscalati** (PESATI)

dove $\Delta w^{(k)}$ è la media pesata esponenziale dei quadrati dei gradienti pesati. $\left(\breve{\nabla}\, C\right)$

ADAM

L'obiettivo principale di ADAM (Adaptive Moment Estimation) 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

$$v^{(k)} = \beta_1 \mathrm{v}^{(k-1)} + (1-\beta_1)\nabla C(w^{(k)})$$

e del momento secondo del gradiente

$$s^{(k)} = \beta_2 \mathrm{s}^{(k-1)} + (1-\beta_2)\left(\nabla C\left(w^{(k)}\right)\right)^2$$

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

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

allora l'equazione di aggiornamento diventa

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

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.

In sintesi, il confronto tra i metodi di ottimizzazione visti.

1.Gradient Descent (GD): Batch Gradient Descent

1. È il metodo di base per l'ottimizzazione dei modelli di machine learning.
2. Richiede il calcolo del gradiente dell'intero set di dati di addestramento ad ogni iterazione, rendendolo computazionalmente costoso per grandi set di dati.
3. Può soffrire di problemi come il plateau dei gradienti o la convergenza lenta in presenza di curve di costo complesse o di minime locali.

2.Stochastic Gradient Descent (SGD):

1. Utilizza un campione casuale dal set di dati di addestramento per calcolare il gradiente ad ogni iterazione.
2. Rispetto al GD, SGD è più efficiente computazionalmente, soprattutto per grandi set di dati.
3. Tuttavia, la variabilità introdotta dall'utilizzo di un singolo campione può rendere l'ottimizzazione più rumorosa e richiedere una scelta attenta del tasso di apprendimento.

3.Mini-batch Gradient Descent:

1. È una variante di SGD in cui il gradiente viene calcolato su un piccolo sottoinsieme di campioni (mini-batch) a ogni iterazione.
2. Combina vantaggi di GD e SGD: più efficiente di GD e meno rumoroso di SGD.
3. Richiede una scelta appropriata della dimensione del mini-batch, che può influenzare la convergenza e la velocità di apprendimento.

1. **Momentum-based:**
    1. Utilizzano un momento che tiene conto dei gradienti precedenti per accelerare la convergenza.
    2. Sono particolarmente efficaci per ridurre l'effetto di oscillazioni o rumore nella direzione del gradiente.
    3. Possono aiutare a superare zone di plateau e minimi locali piatti.
2. **Adagrad:**
    1. Adatta il learning rate per ciascun parametro in base alla somma dei gradienti passati.
    2. Funziona bene in presenza di sparsi gradienti o feature sparse.
    3. Tuttavia, può diminuire troppo velocemente il tasso di apprendimento nel corso dell'addestramento, rendendolo inefficiente nelle fasi successive.

Quadrati

3. **RMSprop:**

Media Ponderata Esponenziale dei Quadrati dei Gradienti Passati

    1. Adatta il learning rate in base alla ~~media mobile delle varianze dei gradienti passati~~.
    2. Aiuta ad affrontare il problema di diminuzione eccessiva del tasso di apprendimento in Adagrad.
    3. È particolarmente utile quando i parametri del modello hanno diverse scale di aggiornamento.

Alcuni parametri possono avere gradienti molto grandi, mentre altri possono avere gradienti molto piccoli.

4. **ADAM:**
    1. Combina le idee di RMSprop e momento per adattare il learning rate in modo individuale per ciascun parametro.
    2. È efficace in diverse situazioni e spesso offre una convergenza più rapida rispetto ad altri metodi di ottimizzazione. Tuttavia, può richiedere una scelta accurata dei suoi iperparametri per ottenere prestazioni ottimali

MLPClassifier

class sklearn.neural_network.MLPClassifier(hidden_layer_sizes=(100,), activation='relu', *, solver='adam', alpha=0.0001, batch_size='auto', learning_rate='constant', learning_rate_init=0.001, power_t=0.5, max_iter=200, shuffle=True, random_state=None, tol=0.0001, verbose=False, warm_start=False, momentum=0.9, nesterovs_momentum=True, early_stopping=False, validation_fraction=0.1, beta_1=0.9, beta_2=0.999, epsilon=1e-08, n_iter_no_change=10, max_fun=15000)