- **Varianti del Gradient Descent**

Batch Gradient Descent

Per il calcolo della funzione costo C(w) vengono usate tutte gli n_T campioni del training set .

$$T = \{(x^{(j)}, y^{(j)}), x^{(j)} \in \mathbb{R}^d, y^{(j)} \in \mathbb{R}^s, j = 1, \dots, n_T\},$$

$$\arg\min_w C(w) = \frac{1}{n_T} \sum_{j=1}^{n_T} L\left(y^{(j)}, \hat{y}^{(j)}\right)$$

Quindi:

- si considera l'intero set di addestramento,
- si esegue la Forward Propagation e si calcola la funzione di costo.
- si aggiornano i parametri usando il gradiente di questa funzione di costo rispetto ai parametri.

Epoca: l'intero set di addestramento viene passato attraverso il modello, vengono eseguite la propagazione in avanti e la propagazione all'indietro e i parametri vengono aggiornati.

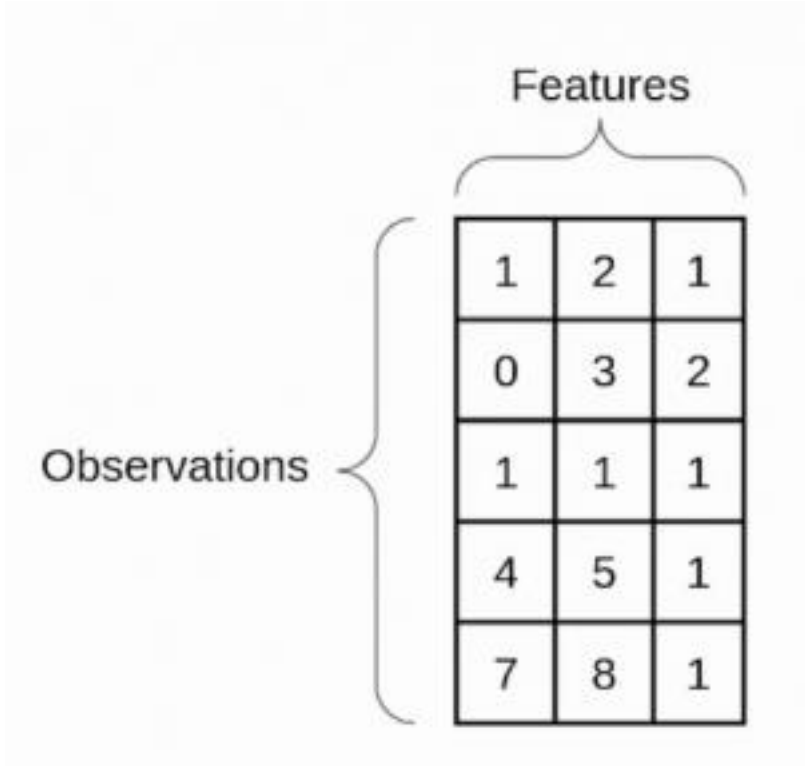
Nel **batch Gradient Descent** poiché stiamo utilizzando l'intero set di addestramento**, i parametri verranno aggiornati solo una volta per epoca.**

Discesa del gradiente stocastico (SGD)

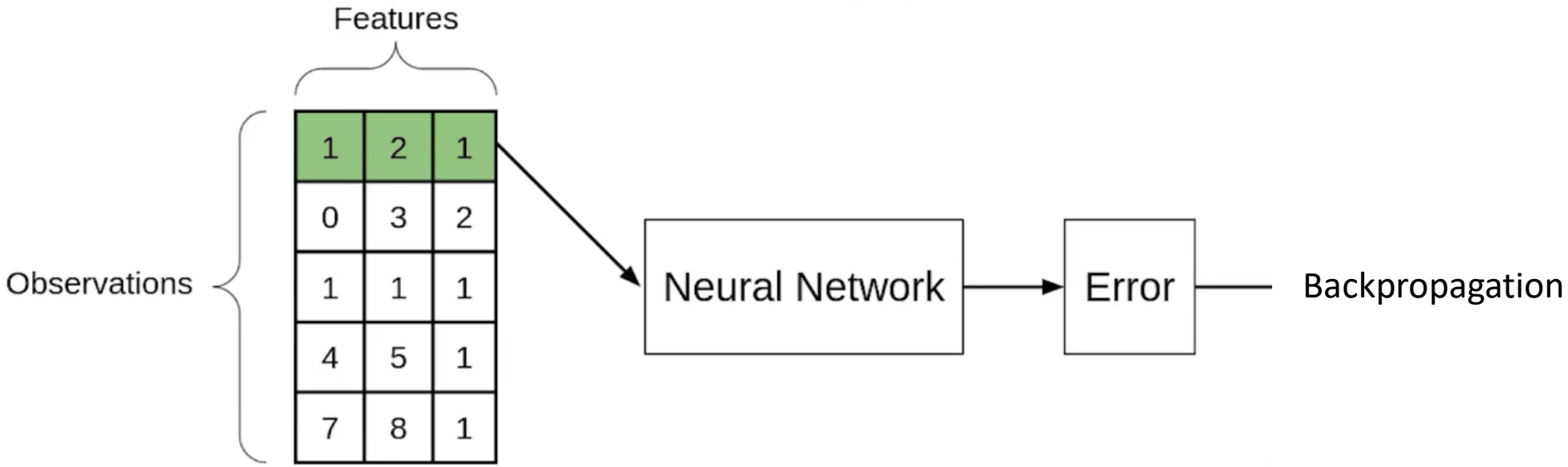
Se si utilizza una singola osservazione per calcolare la funzione di costo, si parla di **Stochastic Gradient Descent**, comunemente abbreviato **SGD**. Passiamo una sola osservazione alla volta, calcoliamo il costo e aggiorniamo i parametri.

Esempio:

Supponiamo di avere un dataset di training formato da 5 campioni, ognuno dei quali costituito da 3 input



Ora, se usiamo l'SGD, prenderemo la prima osservazione, poi la passeremo attraverso la rete neurale (**forward propagation**) , calcoleremo l'errore e quindi aggiorneremo i parametri.



Quindi entrerà in input la seconda osservazione, che **sarà elaborata tramite la forward propagation che utilizza i pesi che sono stati aggiornati mediante il gradiente della funzione costo calcolata sulla prima osservazione** ed saranno eseguiti passaggi simili su di essa. Questo passaggio verrà ripetuto fino a quando tutte le osservazioni non saranno passate attraverso la rete e i parametri non saranno stati aggiornati.

Ogni aggiornamento dei parametri, viene chiamato **Iterazione**.
In questo esempio, poiché abbiamo 5 osservazioni, i parametri verranno aggiornati 5 volte (sono state effettuate 5 iterazioni).

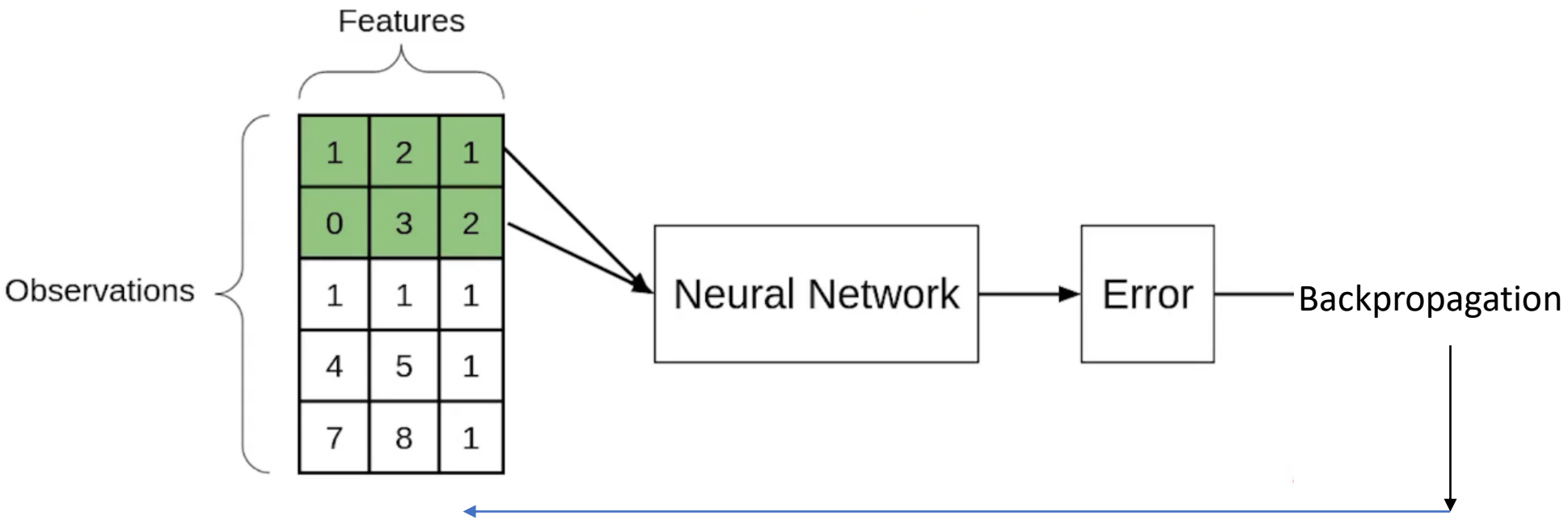
Nel caso del Batch Gradient Descent tutte le osservazioni insieme sarebbero state eleborate dalla rete ed i parametri sarebbero stati aggiornati solo una volta.
Nel caso di SGD, ci saranno n_T iterazioni per epoca, dove n_T è il numero di osservazioni nel dataset di training.

Se si utilizza l'intero set di dati per calcolare la funzione di costo si parla di **Batch Gradient Descent** e se si utilizza una singola osservazione per calcolare il costo si parla di **SGD**.

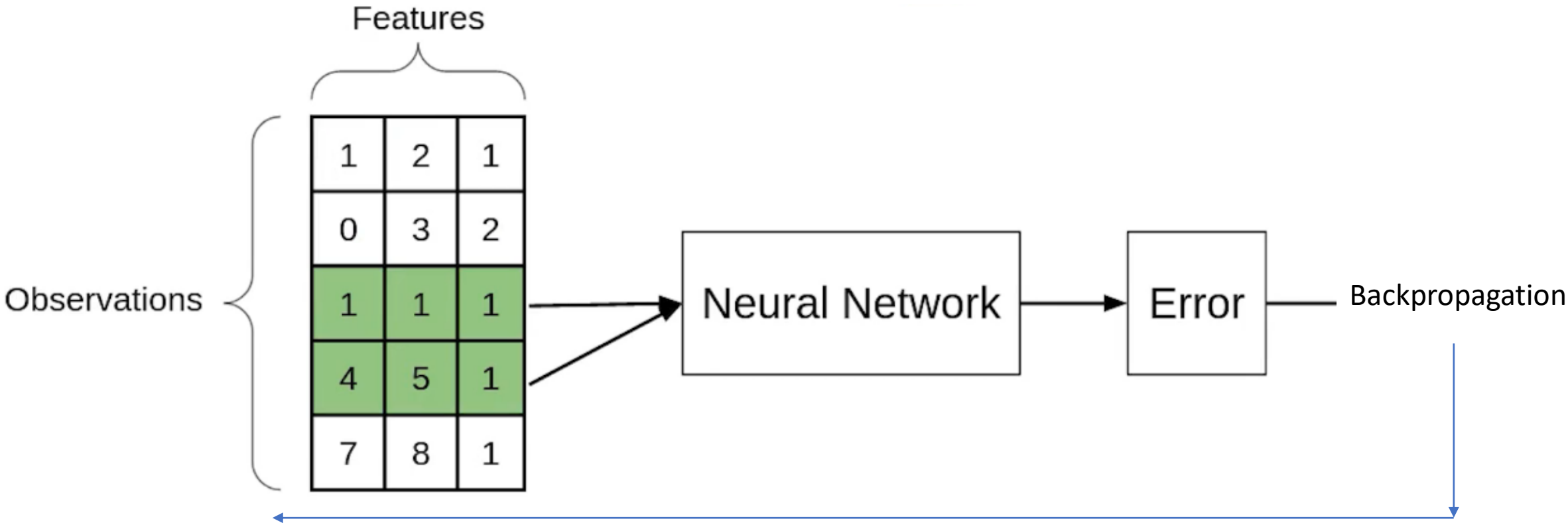
Mini-batch Stocastic Gradient Descent:

Per calcolare la funzione di costo si considera **un sottoinsieme dell'intero set di dati.** Quindi, se ci sono n_T osservazioni, il numero di osservazioni in ciascun sottoinsieme o mini-batch sarà maggiore di 1 e minore di n_T .

Ancora una volta prendiamo lo stesso esempio. Supponiamo che la dimensione del batch sia 2. Quindi prenderemo le prime due osservazioni, le «passeremo» attraverso la rete neurale (forward propagation), calcoleremo l'errore e quindi aggiorneremo i parametri.



Quindi, utilizzando i parametri aggiornati tramite il gradiente della funzione costo calcolata sul primo minibatch, considereremo le due osservazioni successive ed eseguiremo passaggi simili, forward propagation attraverso la rete, calcolo dell'errore ed aggiornamento i parametri.



Poiché ci rimane la singola osservazione nell'iterazione finale, ci sarà solo una singola osservazione e i parametri saranno aggiornati utilizzando questa osservazione.

Confronto tra Batch GD, SGD e Mini-batch SGD:

Segue un confronto tra le diverse varianti di Gradient Descent

Confronto: numero di osservazioni utilizzate per l'aggiornamento.

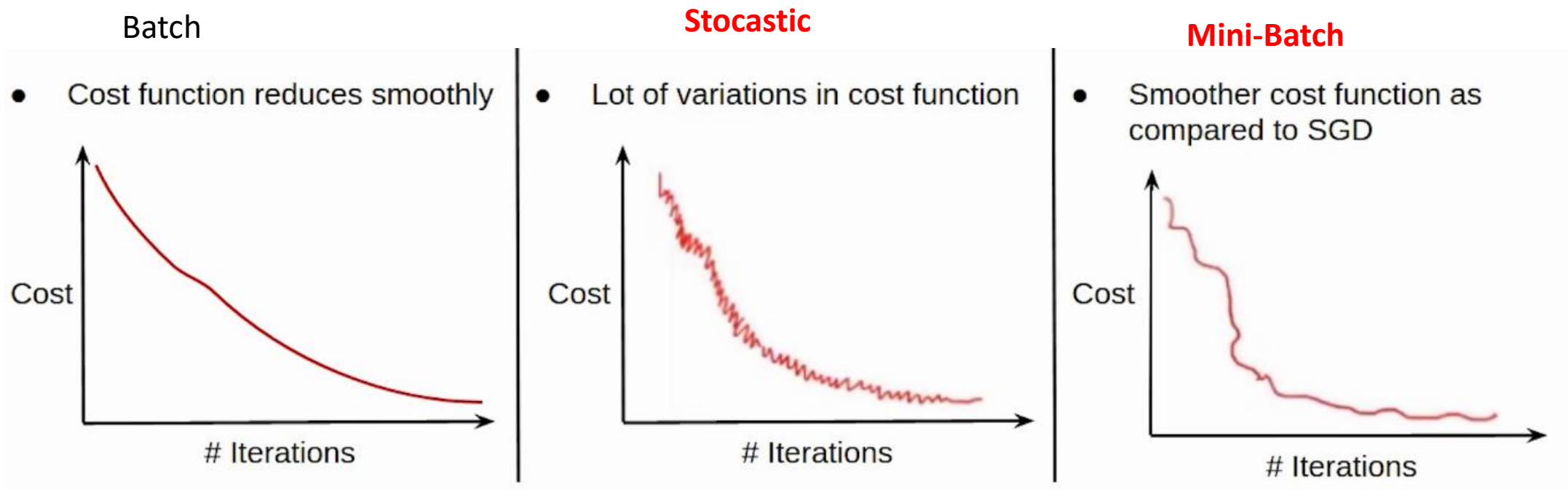
In **batch Gradient Descent**

- prendiamo l'intero set di dati
- calcoliamo la funzione di costo
- aggiorniamo i parametri

Nel caso di **Stochastic Gradient Descent**, aggiorniamo i parametri dopo ogni singola osservazione e sappiamo che ogni volta che i pesi vengono aggiornati si parla di iterazione.

Nel caso di **Mini-batch Stocastic Gradient Descent**, prendiamo un sottoinsieme di dati e aggiorniamo i parametri in base a ogni sottoinsieme.

Confronto Funzione costo



- Poiché nel caso del **Batch GD** aggiorniamo i parametri utilizzando l'intero set di dati la funzione costo, in questo caso, si riduce uniformemente.
- Questo aggiornamento nel caso di SGD non è così fluido. Poiché stiamo aggiornando i parametri sulla base di una singola osservazione, ci sono molte iterazioni. Potrebbe anche essere possibile che il modello inizi ad apprendere anche il rumore.
- L'aggiornamento della funzione di costo nel caso di **Mini-batch Gradient Descent** è più fluido rispetto a quello della funzione di costo in SGD. Dal momento che non aggiorniamo i parametri dopo ogni singola osservazione ma dopo ogni sottoinsieme dei dati

Veniamo ora al costo di calcolo e al tempo impiegato da queste varianti **di Gradient Descent**.

Poiché dobbiamo caricare l'intero set di dati alla volta, eseguire la propagazione in avanti su di esso e calcolare l'errore e quindi aggiornare i parametri, nel **caso Batch Gradient Descent** il costo di calcolo è molto elevato.

Il tempo di calcolo della funzione costo nel caso di **SGD** è inferiore rispetto a **Batch Gradient Descent** poiché dobbiamo caricare ogni singola osservazione alla volta, ma il tempo di calcolo qui aumenta poiché ci sarà un numero maggiore di aggiornamenti che si tradurrà in un numero maggiore di iterazioni .

Nel caso di **Mini-batch Stocastic Gradient Descent**, prendendo un sottoinsieme dei dati ci sono un numero minore di iterazioni o aggiornamenti e quindi il tempo di calcolo nel caso di mini-batch Gradient Descent è inferiore a SGD.

Inoltre, poiché non stiamo caricando l'intero set di dati alla volta mentre carichiamo un sottoinsieme dei dati, anche il tempo di calcolo della funzione costo è inferiore rispetto alla discesa del gradiente batch. Questo è il motivo per cui di solito si preferisce utilizzare il **Mini-batch Stocatic Gradient Descent.**

Iperparametri

Gli iperparametri sono parametri esterni al modello di machine learning che devono essere impostati prima dell'avvio del processo di addestramento. A differenza dei parametri del modello, che vengono appresi durante il processo di addestramento stesso, gli iperparametri influenzano il comportamento del processo di addestramento e la configurazione del modello.

Gli iperparametri determinano come avviene l'addestramento del modello e possono includere:

1.Learning rate : Determina quanto velocemente o lentamente il modello si adatta ai dati.

2.Numero di epoche: il numero di volte in cui l'intero set di dati di addestramento viene utilizzato per addestrare il modello. Un numero insufficiente di epoche può portare a un modello non addestrato adeguatamente, mentre un numero eccessivo di epoche può portare a un overfitting.

3.Dimensione del mini-batch: il numero di esempi di addestramento utilizzati in ciascuna iterazione dell'algoritmo di ottimizzazione (ad esempio, SGD o mini-batch GD). La dimensione del mini-batch può influenzare la velocità di apprendimento e la stabilità dell'addestramento.

1.Regolarizzazione: i parametri che controllano la regolarizzazione del modello, come il peso della regolarizzazione L1 o L2, che influenzano la complessità del modello e la tendenza all'overfitting.

2.Inizializzazione dei pesi: il metodo utilizzato per inizializzare i pesi del modello. Una buona inizializzazione può favorire una convergenza più rapida e una migliore performance.

3.Funzione di attivazione: la funzione utilizzata per calcolare l'output di un'unità nel modello. Le funzioni di attivazione comuni includono ReLU, sigmoid e tanh.

4. Architettura del modello: la struttura del modello, inclusi il numero di strati, il numero di unità in ciascun strato e le connessioni tra gli strati. La scelta dell'architettura dipende dal problema e dai dati specifici.

La scelta corretta degli iperparametri può influenzare significativamente le prestazioni del modello. Spesso, gli iperparametri vengono regolati attraverso tentativi ed errori, eseguendo iterazioni multiple di addestramento e valutando le prestazioni su un set di dati di validazione. L'esperienza, la conoscenza del dominio e le best practice possono guidare la scelta degli iperparametri ottimali per un determinato problema di machine learning.

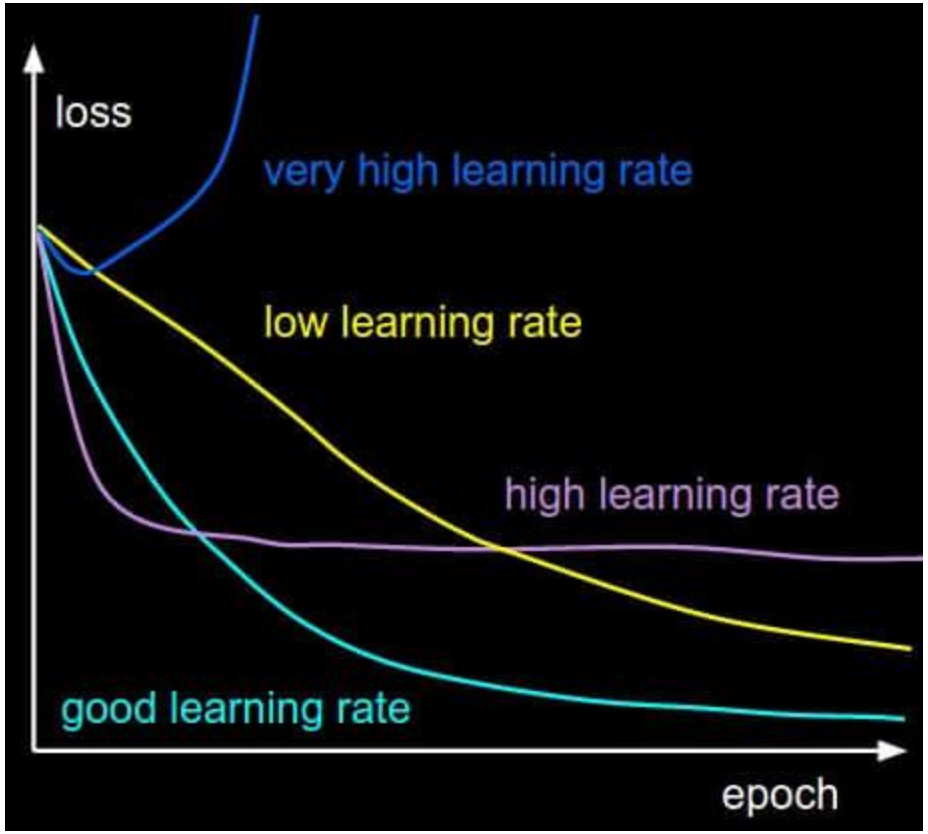
Nel seguente collegamento saranno analizzate Le proprietà della discesa del gradiente per minimizzare un funzionale e sarà introdotto il metodo di discesa con momento

GradientDescentMomentum.pdf

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**, marrone)
- La funzione costo raggiunge un valore migliore di quello iniziale,ma è ancora lontano da un valore ottimale (**Learning rate alto**: 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 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 minimi
- un Learning rate basso è più appropriato nella fase finale dell'apprendimento perché i pesi sono già vicini ai minimi (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 l'LR rimane grande potremmo rimbalzare intorno al minimo senza raggiungerlo

- Inizializzazione: come i parametri sono impostati inizialmente e come si evolvono? Utilizzare Learning rate iniziali grandi potrebbe non essere utile, poiché i parametri iniziali sono casuali le iniziali direzioni di aggiornamento potrebbero essere prive di significato

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

Sapere quando far decadere il Learning rate può essere complicato

- se si fa decadere lentamente si spreca tempo di calcolo rimbalzando con pochi miglioramenti per molto tempo
- decadimento troppo aggressivo e il sistema si raffredderà troppo rapidamente incapace di raggiungere la posizione migliore che può.

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

•**Step decay** : riduce il tasso di apprendimento 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 il numero di epoche eseguite

- Il **decadimento esponenziale** ha la forma matematica

$$\eta = \eta_0 \cdot e^{-kt}$$

dove η_0 e k sono iperparametri e t è il numero di iterazione corrente

- Il **decadimento basato sul tempo** divide il learning rate iniziale η_0 in funzione del numero di iterazioni eseguite (t)

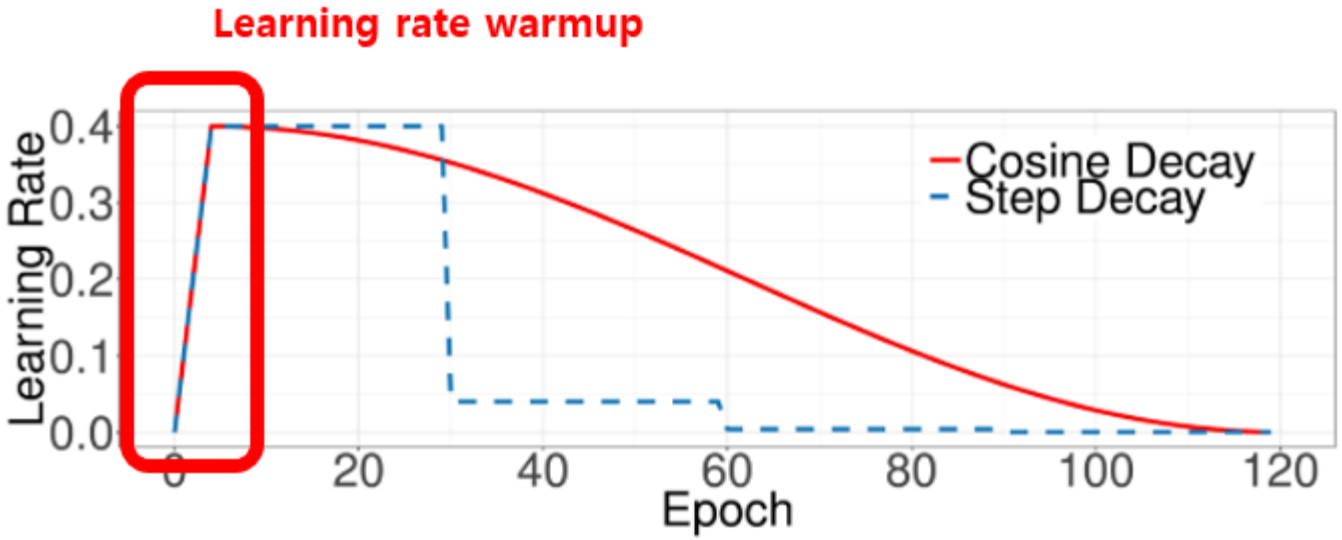
$$\eta = \frac{\eta_0}{1 + k \cdot t}$$

dove η_0 e k sono iperparametri

In alcuni casi, l'inizializzazione casuale dei parametri **non garantisce una buona soluzione** soprattutto se all'inizio viene utilizzato un **learning rate** grande che porta alla divergenza

Questo problema può essere affrontato scegliendo un **learning rate** sufficientemente piccolo per evitare divergenze all'inizio, purtroppo questo significa che il progresso è molto lento.

Una soluzione semplice consiste nell'utilizzare un periodo di **warm-up**: si inizia con un learning rate molto inferiore al learning rate "iniziale" e poi lo si aumenta in alcune iterazioni o epoche fino a raggiungere quel learning rate "iniziale" e poi lo si riduce fino alla fine del processo di ottimizzazione.



(a) Learning Rate Schedule

Learning rate adattivo

La sfida dell'utilizzo di un piano di aggiornamento dei **learning rate** è che gli iperparametri devono essere definiti in anticipo e dipendono fortemente dal tipo di modello e problema.

Un altro problema è che lo stesso **learning rate** viene applicato a tutti gli aggiornamenti dei pesi.

Se disponiamo di dati sparsi, potremmo invece voler aggiornare i pesi in misura diversa

Abbiamo bisogno di diminuire il valore del LR in modo diverso per ciascun peso man mano che l'allenamento procede

Ci sono quattro metodi rappresentativi che modificano in modo adattivo i LR

Adagrad
RMSProp
Adadelta
Adam

Adagrad

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

- Regola il tasso di apprendimento per i parametri in proporzione alla loro cronologia di aggiornamenti, più aggiornamenti più decadimento

$$s_k = s_{k-1} + \left(\nabla C(w_k)\right)^2$$

$$w_{k+1} = w_k - \frac{\eta}{\sqrt{s_k + \epsilon}} \nabla C(w_k)$$

dove $\epsilon > 0$ è un iperparametro scalare con valore molto piccolo (in genere $1e-7$) che serve per evitare che il denominatore si annulli

Minore è il gradiente accumulato, minore sarà il valore s_k , portando a 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

RMSProp

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

$$s_k = \gamma s_{k-1} + (1-\gamma)\big(\nabla C(w_k)\big)^2$$

$$w_{k+1} = w_k - \frac{\eta}{\sqrt{s_k + \epsilon}} \nabla C(w_k)$$

$\gamma > 0$ è un iperparmetro, il cui valore suggerito dagli autori è γ=0.9

$\epsilon > 0$ è un iperparametro scalare con valore molto piccolo (in genere $1e-7$) che serve per evitare che il denominatore si annulli

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

$$s_k = \gamma s_{k-1} + (1-\gamma)\left(\nabla C(w_k)\right)^2$$

$$\breve{\nabla}\, C = \frac{\sqrt{\Delta\omega_{k-1}+\epsilon}}{\sqrt{s_t+\epsilon}}\nabla C(w_k)$$

$$w_{k+1} = w_k - \breve{\nabla}\, C$$

$$\Delta w_k = \gamma \Delta w_{k-1} + (1-\gamma)\left(\breve{\nabla}\, C\right)^2$$

dove Δw_k è la media pesata esponenziale dei quadrati dei gradienti pesati $\left(\breve{\nabla}\, C\right)$

$\gamma > 0$ è un iperparmetro, il cui valore suggerito dagli autori è γ=0.9 5

ADAM

L'obiettivo principale di ADAM (Adaptive Moment Estimation) è quello di combinare i vantaggi di due altri algoritmi di ottimizzazione: RMSprop e Momento.

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

$$v_k = \beta_1 v_{k-1} + (1-\beta_1)\nabla C(w_k)$$

e del momento secondo del gradiente

$$s_k = \beta_2 s_{k-1} + (1-\beta_2)\big(\nabla C(w_k)\big)^2$$

dove β_1, β_2 iperparametri positivi ed gli autori consigliano di scegliere $\beta_1 = 0.9, \quad \beta_2 = 0.999,$

Si noti che se si inizializzano $v_0 = 0$ ed $s_0 = 0$, i momenti iniziali "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

$$w_{k+1} = w_k - \frac{\eta}{\sqrt{\hat{s}_k + \epsilon}}\,\hat{v}_k$$

dove $\epsilon > 0$ è un iperparametro scalare con valore molto piccolo (in genere $1e-7$) che serve per evitare che il denominatore si annulli

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.

Ci sono diversi metodi di ottimizzazione utilizzati nel machine learning, ognuno con le proprie caratteristiche e vantaggi. Di seguito ti fornisco un confronto tra alcuni dei metodi di ottimizzazione più comuni.

1.Gradient Descent (GD):

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 Methods (e.g., Momentum, Nesterov Accelerated Gradient):
    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.
3. RMSprop:
    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.
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