

# Lezione 6 22/10/2024

## Optimization for training deep models

Per risolvere un problema di deep learning dobbiamo affrontare le seguenti questioni:

- Miglior batch size?
- Come evitare i minimi locali?
- Miglior algoritmo di addestramento?
- Tasso di apprendimento ottimale?
- Migliore politica di inizializzazione dei pesi?

Non c'è una regola per risolvere questi problemi, dipende dal modello, dal contesto...

In questa lezione vogliamo capire:

- Come la dimensione del batch possa influenzare i risultati
- I problemi dei minimi locali e dei punti sella
- Diversi algoritmi di addestramento
- Importanza del tasso di apprendimento
- Politiche di inizializzazione esistenti e le loro implicazioni

### Differenza tra learning e ottimizzazione pura

Il machine learning agisce indirettamente, a differenza dell'ottimizzazione.

Di solito, vogliamo ottimizzare una misura di performance  $P$ , basata sul set di test, e il problema potrebbe essere anche intrattabile (la natura non convessa del paesaggio della funzione di loss nel deep learning). Anche se i dati di training hanno la stessa distribuzione dei dati di test, la funzione obiettivo che vogliamo ottimizzare è molto complessa.

Visto che non sappiamo la funzione obiettivo reale e non possiamo derivarla, perchè non sappiamo i dati che verranno usati in futuro sul modello, facciamo la cost function. Il nostro obiettivo è di ottimizzare la misura di performance (accuracy, precision, recall...), ma questa è difficile da ottimizzare come funzione. Quindi lavoriamo sulla cost function, sperando che questa ottimizzerà anche la performance. Quindi ottimizziamo la misura di performance indirettamente.

Example: Application to recognize cats. You train on a set of pictures but you cannot know which picture you will be asked to recognize...



In real applications your model cannot be optimized with respect to the picture that it will be asked to recognize !!!



La misura di performance  $P$  è il numero di classificazioni corrette. Però non abbiamo queste immagini, quindi creiamo un training set e misuriamo la performance su questo set.

Minimizing  $J(\theta)$  does not necessarily minimize  $P$

Scegliamo una loss function sperando che questa rappresenti la performance. Un problema è che questa funzione spesso è una media semplice sul training set:

$$J(\theta) = E_{(x,y) \sim \hat{p}_{DATA}} L(f(x;\theta), y)$$

where

- $E$  is the expectation operator
- $L$  is the per-example loss function
- $f(x; \theta)$  is the predicted output for input  $x$
- $y$  is the target output
- $\hat{p}_{DATA}$  is the empirical distribution (notice the hat)

Of course we would prefer minimize the objective function with respect to the training set using

$$J^*(\theta) = E_{(x,y) \sim p_{DATA}} L(f(x;\theta), y)$$

Abbiamo una distribuzione empirica, non quella reale.

Ultimo pezzo: questo non è sempre possibile.

$$J^*(\theta) = E_{(x,y) \sim p_{DATA}} L(f(x;\theta), y)$$

Is called **Risk**

If we knew  $p_{DATA}$ , risk minimization will be reduced to a standard optimization task

Ma visto che si usa il training set, la loss function sta minimizzando il **rischio empirico**.

$$E_{(x,y) \approx \hat{p}_{DATA}} [L(f(x; \theta), y)] = \frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}; \theta), y^{(i)})$$

where m is the number of training examples

---

Qual è il problema con la minimizzazione del rischio empirico?

La minimizzazione del rischio empirico è soggetta a overfitting. Modelli con capacità sufficientemente elevata possono semplicemente memorizzare il training set.

Gli algoritmi più efficaci usano la discesa del gradiente. La loss function che misura la performance non è normalmente differenziabile, quindi dobbiamo scegliere un'altra loss function. Questo significa che nel contesto del deep learning non possiamo usare la minimizzazione empirica, ma dobbiamo cambiare approccio, perché la loss function deve riuscire ad approssimare e deve funzionare con l'algoritmo del gradiente.

Quindi per minimizzare il rischio, minimizziamo la **loss function surrogata**.

Questo è un proxy del rischio minimo e può essere ottimizzata efficientemente.

Example: the negative log-likelihood is a surrogate to the 0-1 loss for classification.

Quindi al posto di calcolare la 0-1 loss, dove 1 è un elemento correttamente classificato e 0 non correttamente classificato, visto che questa funzione non è continua, usiamo invece la negative log likelihood che permette al modello di stimare la probabilità condizionale delle classi, dato l'input, e se il modello è in grado di farlo bene, allora può scegliere le classi che producono la minore aspettativa di errore di classificazione.

Qual è la differenza tra log likelihood e cross entropy?

La log likelihood approssima la performance, non mi da 0-1, mi da un valore tra 0 e 1.

---

AI: La **log-likelihood** misura quanto bene un modello probabilistico spiega i dati osservati. È la probabilità logaritmica dei dati osservati data una distribuzione predetta dal modello.

La **cross-entropy** invece è una misura della distanza tra due distribuzioni probabilistiche: la distribuzione vera (reale) e quella predetta dal modello. È spesso utilizzata come funzione di perdita per classificazione, penalizzando fortemente predizioni errate.

In breve, la log-likelihood si concentra su quanto bene il modello spiega i dati, mentre la cross-entropy misura la differenza tra distribuzioni vera e predetta.

Quando alleniamo di solito usiamo procedure di early stopping, al posto di andare finchè troviamo un minimo (a differenza dei problemi di ottimizzazione normali).

Tipicamente, il criterio di early stopping si basa su una funzione di loss sottostante, come la 0-1 loss misurata sul set di validazione, ed è progettato per interrompere l'algoritmo prima che si verifichi overfitting.

L'halting deve accadere prima dell'overfitting, prima che l'errore sul validation set inizi a risalire.

Questa può essere vista come una tecnica di regolarizzazione che prova a reincorporare la vera loss function nel processo di learning.

## Batch and minibatch algorithms

Un aspetto degli algoritmi di apprendimento automatico che li differenzia dagli algoritmi di ottimizzazione generali è che la funzione obiettivo solitamente si decompone come una somma sugli esempi di addestramento:

$$J(\theta) = E_{(x,y)=\hat{p}_{DATA}} \log p_{MODEL}(x, y; \theta)$$

The gradient in this case is also an expectation over training data:

$$\nabla_{\theta} J(\theta) = E_{(x,y)=\hat{p}_{DATA}} \nabla_{\theta} \log p_{MODEL}(x, y; \theta)$$

Quindi anche il gradiente è un'expectation. Quanto ottimizziamo la true objective function non sempre questa può essere decomposta in somme, ma l'expectation sì? Questa caratteristica è sfruttata in questo algoritmo, basato sull'algoritmo del gradiente. È vero che la surrogate loss function è la somma sull'intero training set, ma visto che è un'expectation, posso incrementare man mano l'expectation usando subset del training set. Questo è importante perché se la funzione non fosse scomponibile, dovrei performare il training sull'intero dataset ogni volta, prima di un update, il che sarebbe impossibile. Questa è la caratteristica che rende la rete neurale allenabile

Quindi ad ogni iterazione, facciamo random sampling di un piccolo numero di esempi del training set.

Questo funziona perchè dalla definizione dell'expectation, sappiamo che:

Standard error of the mean estimated from m samples  $x$  is,

$$SE(\mu) = \sqrt{VAR\left[\frac{1}{m} \sum x^{(i)}\right]} = \frac{\sigma}{\sqrt{m}}$$

where  $\sigma$  is the standard deviation of the value of the samples.

Se aumentiamo il numero di samples, l'errore scende con la radice quadrata di questo numero. Non c'è una relazione lineare. Il denominatore fa vedere che c'è una meno di un ritorno lineare nell'usare più esempi per stimare il gradiente.

Se consideriamo anche che l'iterazione, l'update dei pesi, è molto più veloce con batches piccoli, allora gli algoritmi normalmente convergono più facilmente se possono computare la stima del gradiente più velocemente con pochi esempi, piuttosto che calcolarlo lentamente e più precisamente.

Questo significa che in teoria possiamo ottenere una buona stima considerando solo un esempio alla volta. Però visto che gli esempi possono variare, questo può introdurre troppa instabilità. Serve quindi un trade off tra batches troppo piccoli e batches troppo grandi.

Un'altra considerazione è che in un training set potremmo avere molti esempi simili che non aggiungono molta informazione. Se consideriamo il caso peggiore, dove abbiamo che tutti gli esempi sono identici, il gain che otteniamo calcolando il

gradiente sull'intero dataset è identico a quello dove uso un solo esempio. Questo è un caso limite, ma possiamo allo stesso modo trovare dei numeri grandi di esempi che non aggiungerebbero informazioni.

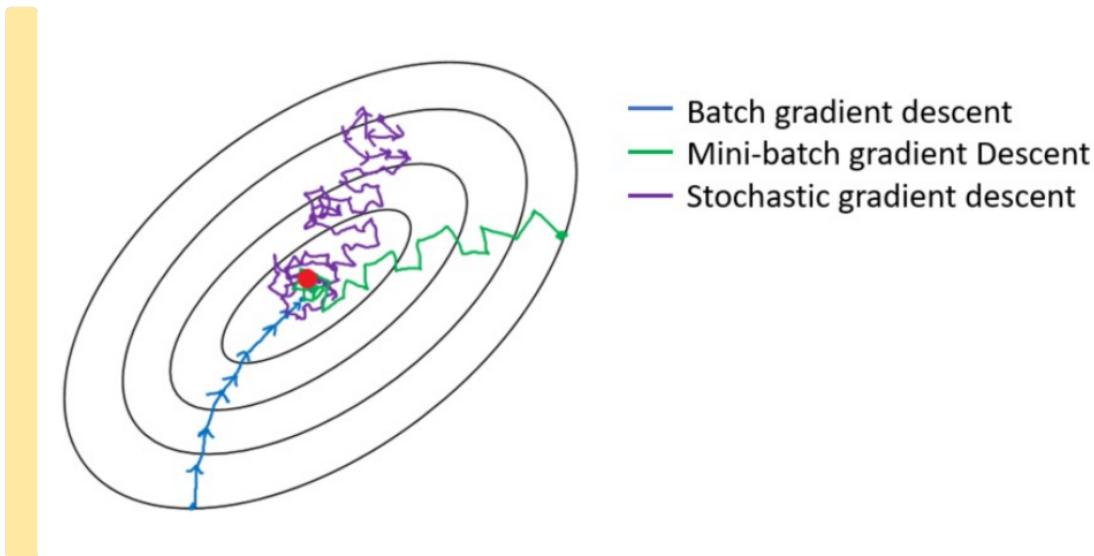
Gli algoritmi di ottimizzazione che usano l'intero training set per calcolare il gradiente sono chiamati **batch** o **metodi del gradiente deterministico**.

Questo può confondere perchè i **batch** sono i subset di solito. Però hanno anche questa definizione.

Se usiamo un singolo esempi di training invece, abbiamo un **metodo stocastico** oppure **online gradient method**.

La maggior parte dei metodi usati nel deep learning sono nel mezzo tra i due, e sono chiamati **minibatch** e **minibatch stochastic methods**. Noi li chiameremo metodi stocastici. WTF?

Qui possiamo vedere la differenza tra i metodi:



Nel metodo batch abbiamo la direzione esatta, perchè abbiamo il gradiente esatto. Quindi abbiamo meno iterazioni, ma ci mettono più tempo.

Nel metodo stocastico abbiamo tanta sensibilità al singolo esempio, ci sono molte più iterazioni ma ci mettono poco.

Il minibatch è una via di mezzo.

- Larger batches provide a more accurate estimate of the gradient, but with less than linear returns.
- Multicore architectures are usually underutilized by extremely small batches. This motivates using some absolute minimum batch size, below which there is no reduction in the time to process a minibatch.
- The batch size is usually set in the range of 8, 16, 32, 64, 128 and 256—to exploit parallel processing abilities in GPU's.

Batch più grandi forniscono una stima più accurata del gradiente, ma con rendimenti inferiori a quelli lineari.

Architetture multi-core sono solitamente sotto-utilizzate da batch estremamente piccoli. Questo motiva l'utilizzo di una dimensione minima assoluta per il batch, al di sotto della quale non c'è riduzione nel tempo per processare un mini-batch. La dimensione del batch è solitamente impostata nell'intervallo di 8, 16, 32, 64, 128 e 256 per sfruttare le capacità di elaborazione parallela delle GPU.

Batches piccoli possono avere un effetto di regolarizzazione. Questo perché il training set non è il real set, quindi se calcoliamo il gradiente esatto, non stiamo comunque calcolando quello reale e quindi il punto minimo che raggiungiamo potrebbe essere un punto di overfitting.

La generalizzazione migliore è spesso raggiunta con batches grandi 1, però il numero di step aumenta.

Di solito la soluzione migliore è qualcosa in mezzo.

Se abbiamo batch sizes intorno a 100 di solito abbiamo una buona stima.

I metodi che calcolano aggiornamenti basati solo sul gradiente sono solitamente relativamente robusti e possono gestire dimensioni di batch più piccole (come 100). Metodi del "secondo ordine" (usano anche la matrice hessiana, ovvero la matrice delle derivate della funzione obiettivo) tipicamente richiedono dimensioni di batch molto più grandi (come 10.000), specialmente se  $H$  ha un numero di condizionamento (misura della sensibilità di una matrice a piccole perturbazioni) scarso.

è importante campionare gli esempi dal training set in modo randomico. Perchè se li prendiamo andando in ordine, questo può essere un problema. Pensiamo per esempio ai frame di un video, oppure dei dati temporali. I dati devono essere quindi **indipendenti** per essere rappresentativi del dataset.

Di solito si mischiano gli esempi, prima di dividere in subsets. Di solito questo si fa una volta all'inizio, e al posto di fare sampling basta prenderli in ordine.

---

## Challenges in deep learning

L'ottimizzazione generale di per sé è un compito estremamente difficile.

Tradizionalmente, il machine learning ha evitato questa difficoltà progettando attentamente la funzione obiettivo e i vincoli per assicurarsi che il problema fosse convesso.

Però non possiamo garantire che la funzione obiettivo e i constrains siano convessi. Proxy functions che sono ottimizzabili più facilmente, ma le loss functions sono di solito non convesse. Quando abbiamo ottimizzazione non convessa è difficile applicare i metodi standard.

### III-conditioning

Se abbiamo autovalori molto alti, allora abbiamo numeri di condizionamento molto alti.

Il mal-condizionamento (ill-conditioning) si manifesta nella discesa del gradiente stocastica causando all'algoritmo di bloccarsi, nel senso che anche passi molto piccoli aumentano la funzione di costo.

Anche se l'algoritmo non si blocca, l'apprendimento procederà molto lentamente quando la matrice Hessiana ha un numero di condizionamento elevato.

Questo causa problemi quando applichiamo algoritmi di ottimizzazione, perchè piccoli cambiamenti nell'input possono cambiare l'output di molto. Quindi l'errore potrebbe propagare molto velocemente.

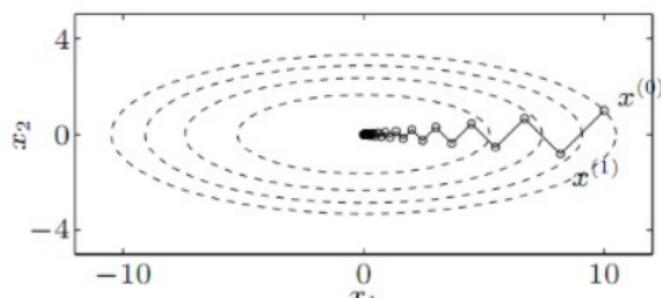
---

In più dimensioni, esiste una derivata seconda diversa per ogni direzione in un singolo punto. Il numero di condizionamento dell'Hessiana in questo punto misura quanto le derivate seconde differiscono l'una dall'altra.

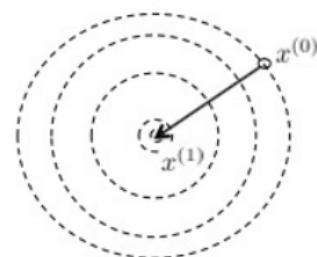
Quando l'Hessiana ha un numero di condizionamento elevato, la discesa del gradiente funziona male. Questo accade perché in una direzione la derivata aumenta rapidamente, mentre in un'altra direzione aumenta lentamente.

La discesa del gradiente non è consapevole di questo cambiamento nella derivata, quindi non sa che deve esplorare preferibilmente nella direzione in cui la derivata rimane negativa più a lungo.

Se abbiamo un condition number = 1, allora in uno step possiamo raggiungere il punto di ottimo. Mentre invece se è 10, tendi ad avere uno zig zag, con un gradiente instabile, e quindi il numero di step per raggiungere l'ottimo è alto.

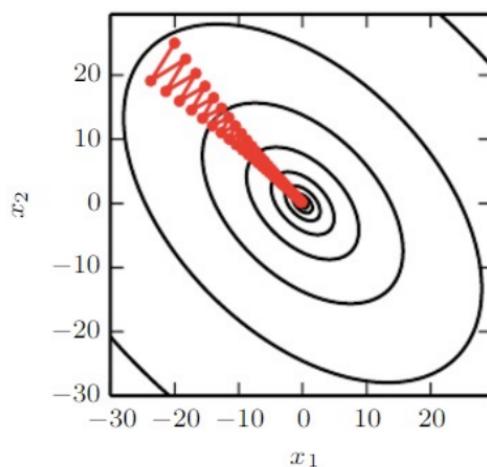


Condition number = 10



Condition number = 1

### ILL-CONDITIONING



## WHEN ILL-CONDITIONING IS A PROBLEM

- Approximate cost function by a quadratic Taylor expansion

$$f(x) \approx f(x^{(0)}) + (x - x^{(0)})^T g + \frac{1}{2} (x - x^{(0)})^T H (x - x^{(0)}) \quad (3)$$

where:

- $g$  is the gradient
- $H$  is the Hessian matrix

Ill-conditioning of the gradient becomes a problem when  $\frac{1}{2}\epsilon^2 g^T H g$  exceeds  $\epsilon g^T g$

- Updating using a learning rate of  $\epsilon$ :

$$f(x - \epsilon g) \approx f(x^{(0)}) - \epsilon g^T g + \frac{1}{2} \epsilon^2 g^T H g \quad (4)$$

- The first term is the squared gradient norm (positive)
- If second term grows too much, the cost increases
- Monitor both terms to see if ill conditioning is a problem

ADVANCED MACHINE LEARNING - ENZA MESSINA

Possiamo controllare se il secondo termine cresce troppo. Possiamo quindi monitorare per capire se abbiamo un problema. Questo non è fatto a priori, però se si sospetta che questo sia il problema, allora si può verificare, e poi restringere mettendo un upper bound al gradiente per evitare che salti troppo velocemente.

## Local minima

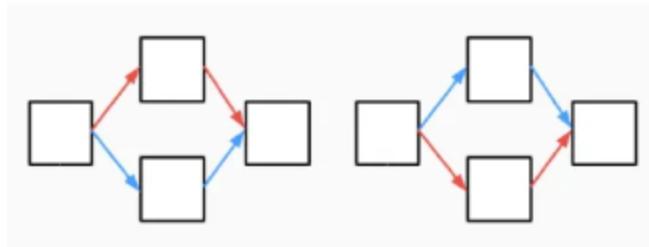
Se usiamo il metodo del gradiente, questo si ferma quando abbiamo 0, quindi anche in minimi locali.

I minimi locali non sono necessariamente un problema, a meno che c'è una regione molto piatta, perché questa rallenterà l'algoritmo di molto.

Un modello è identificabile se un ampio set di addestramento produce un insieme univoco di parametri. I modelli con variabili "latenti" spesso non sono identificabili.

Le reti neurali non sono identificabili:

- se abbiamo  $m$  strati con  $n$  unità nascoste ciascuno, ci sono  $n!^m$  modi di disporre le unità nascoste per ottenere modelli equivalenti. Questo è chiamato simmetria dello spazio dei pesi.



Per esempio le due immagini sono due network diverse, dove se inverti i 2 neuroni ottengo l'altro, e sono identici. Quindi non sempre il minimo locale è un problema.

Allo stesso modo, i pesi di input e output di ReLU o unità di maxout possono essere scalati per raggiungere lo stesso risultato.

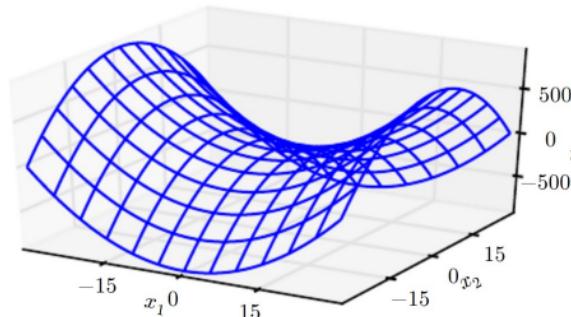
Se troviamo un buon minimo locale, per noi può andare bene, se ha un costo basso, ovvero se ho una regione di minimo locale a costo basso, allora le soluzioni sono equivalenti. I minimi locali sono un problema invece quando il costo è alto (la soluzione non è buona).

Di solito grazie alla stocasticità dei metodi, all'inizio esploro la regione, e quando rimango in un minimo locale è di solito un minimo buono.

## Saddle points

Sono punti dove se vado in una direzione trovo un punto di minimo, e nell'altra trovo un massimo. Il gradiente è 0 anche al massimo, ma l'algoritmo non lo sa.

We can think of a saddle point as being a local minimum along one cross-section of the cost function and a local maximum along another cross-section.



Many classes of random functions exhibit the following behaviour:

- In lower dimensional spaces, local minima and maxima are common.
- In higher dimensional spaces, local minima and maxima are rare, saddle points are much more common.

For a function  $f : R^n \rightarrow R$ , the expected ratio of the number of saddle points to local minima grows exponentially with  $n$ .

What is the intuition behind this phenomenon ?

Quindi più grande è la dimensione, più avremo saddle points rispetto ai minimi locali.

Un altro fenomeno che si verifica in molte funzioni casuali è che gli autovalori dell'Hessiana diventano più probabilmente positivi quando raggiungiamo regioni di basso costo.

Questo significa che i punti di minimo locale è più probabile che abbiano un costo basso, rispetto ad un costo alto.

Inoltre, i punti critici con costo alto è più probabile che siano saddle points, mentre quelli con costo estremamente alto è probabile siano massimi locali. Per gli algoritmi di ottimizzazione del primo ordine che utilizzano solo informazioni sul gradiente, la situazione non è chiara.

Il gradiente può diventare molto piccolo nei saddle points.

D'altra parte, la discesa del gradiente sembra empiricamente essere in grado di sfuggire ai punti di sella in molti casi.

Se rimaniamo fermi in un punto ad alto costo, possiamo cambiare punto di partenza e riprovare.

Per il metodo di Newton, i punti di sella costituiscono un problema importante.

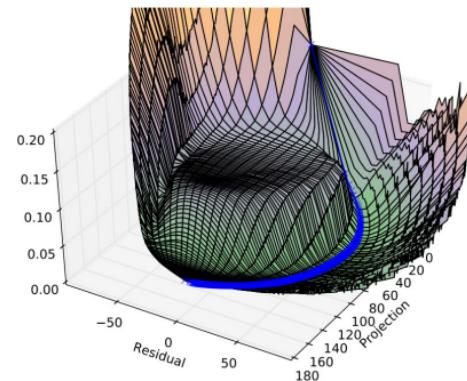
Questo perché, a differenza della discesa del gradiente che è progettata per muoversi verso il basso, il metodo di Newton (matrice hessiana) cerca attivamente soluzioni nei punti critici dove il gradiente è zero.

Questi metodi sono comunque poco usati, hanno molti problemi, non si riesce a scalarli molto al momento.

## Flat regions

Hanno un gradiente di 0 o quasi 0.

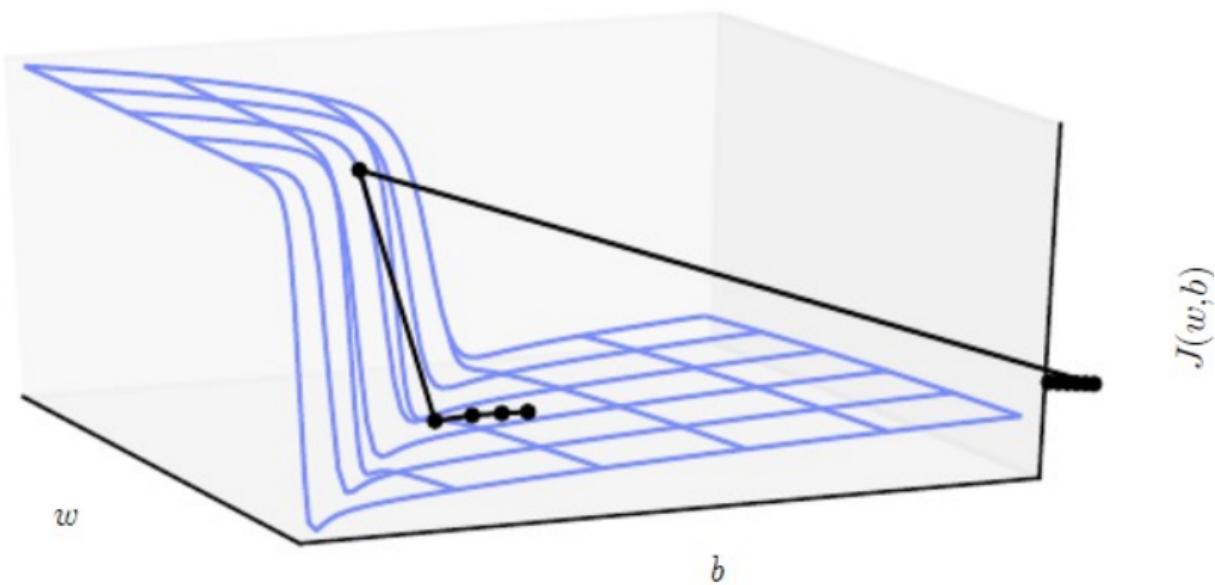
- Flat regions also have zero gradient
- It take a long time to traverse these regions
- Gradient wastes time circumnavigating tall mountains



Anche qui la soluzione è ripartire cambiando i pesi, partendo da una regione diversa.

## Cliffs and exploding gradients

Può capitare che il gradiente è molto sensibile a piccoli rumori.

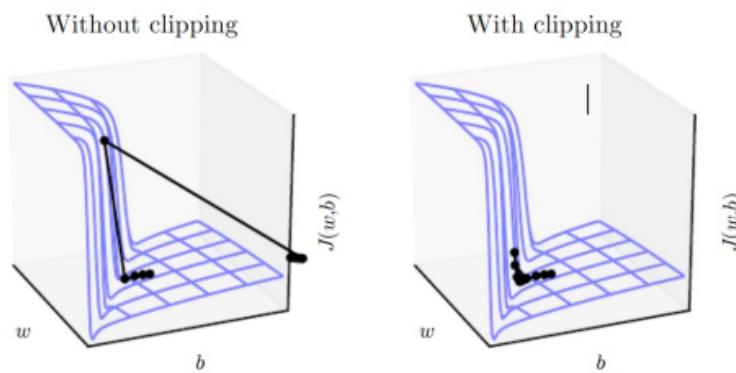


Il gradiente esplosivo è dato dal fatto che moltiplico pesi grandi, molte volte. E visto che non sto computando l'esatto step-size, potrei perdere il controllo.

In questo caso ci sono delle soluzioni semplici come il **clipping heuristic**. Viene aggiunto un upper bound al gradiente, e si vede se la fase di learning continua. Viene quindi ridotto lo step size.

I gradienti non specificano la dimensione ottimale del passo, ma solo la direzione ottimale all'interno di una regione infinitesimale.

Quando l'algoritmo di discesa del gradiente tradizionale propone di fare un passo molto grande, l'euristica di clipping del gradiente interviene per ridurre la dimensione del passo abbastanza da renderlo sufficientemente piccolo da essere meno probabile che vada al di fuori della regione dove il gradiente indica la direzione di discesa approssimativamente più ripida.

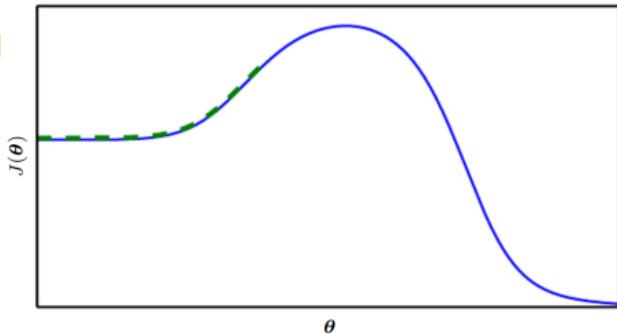


- **Dipendenze a lungo termine:** si verificano quando il grafo computazionale è molto profondo. Il risultato di questo problema è la scomparsa e l'esplosione dei gradienti.
- **Gradienti inesatti:** in pratica, di solito abbiamo solo una stima rumorosa o persino distorta del gradiente e dell'Hessiana. A volte, i gradienti per le nostre funzioni di perdita sono persino intrattabili. Non è un grande problema nell'addestramento delle reti neurali. Le funzioni di perdita surrogate tendono a funzionare abbastanza bene nella pratica.

**Scarsa corrispondenza tra struttura locale e globale:** è possibile superare tutti i problemi sopra citati in un singolo punto e comunque ottenere prestazioni scarse

se la direzione che porta al maggior miglioramento locale non punta verso regioni distanti di costo molto più basso.

Initialization is really important !



## CONCLUSIONS

- Optimizing cost functions for deep networks is really hard.
- We almost never arrive to a global minimum, our goal is to reduce the generalization error rather than the cost function itself.
- Theoretical analysis of whether an optimization algorithm can accomplish this goal is extremely difficult. Developing more realistic bounds on the performance of optimization algorithms therefore remains an important goal for machine learning research.
- Also, **initialization is very important** for stable performance of our optimization algorithms.

## Conclusioni

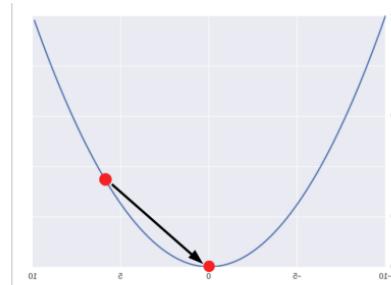
- Ottimizzare le funzioni di costo per le reti profonde è davvero difficile.
- Non arriviamo quasi mai a un minimo globale, il nostro obiettivo è ridurre l'errore di generalizzazione piuttosto che la funzione di costo stessa.

- L'analisi teorica per determinare se un algoritmo di ottimizzazione può raggiungere questo obiettivo è estremamente difficile. Sviluppare limiti più realistici sulle prestazioni degli algoritmi di ottimizzazione rimane quindi un obiettivo importante per la ricerca nel machine learning.
- Inoltre, l'inizializzazione è molto importante per ottenere prestazioni stabili dei nostri algoritmi di ottimizzazione.

## Basic algorithms

### • Stochastic Gradient descend

- Most used algorithm for deep learning
- Do not confuse with (deterministic) gradient descent
- SGD uses minibatches
- Algorithm is similar, but there are some important modifications



- Full training samples  $\{x^{(1)}, \dots, x^{(m)}\}$  with targets  $y^{(i)}$
- Compute gradient

$$g \leftarrow \frac{1}{m} \nabla_{\theta} \left( \sum_{i=1}^m L(f(x^{(i)}; \theta), y^{(i)}) \right) \quad (5)$$

- Apply update

$$\theta \leftarrow \theta - \epsilon g \quad (6)$$

where

- $\epsilon$  is the learning rate
- $\theta$  are the network parameters
- $L(\cdot)$  is the loss function

- **Minibatch** of training samples  $\{x^{(1)}, \dots, x^{(m)}\}$  with targets  $y^{(i)}$

- Compute gradient

$$\hat{g} \leftarrow \frac{1}{m} \nabla_{\theta} \left( \sum_{i=1}^m L(f(x^{(i)}; \theta), y^{(i)}) \right) \quad (7)$$

- Apply update

$$\theta \leftarrow \theta - \boxed{\epsilon_k} \hat{g} \quad (8)$$

Il learning rate è scelto, non computato, è un parametro.

- Learning rate  $\epsilon_k$  must be adaptive
  - Minibatches introduce noise that do not disappear along even at the minimum
- Sufficient condition for convergence:

$$\sum_{k=1}^{\infty} \epsilon_k = \infty \text{ and } \sum_{k=1}^{\infty} \epsilon_k^2 < \infty \quad (9)$$

- Implying  $\lim_{k \rightarrow \infty} \epsilon_k = 0$

Un modo comune di aggiornare il learning rate è quello di diminuirlo linearmente fino all'iterazione tau.

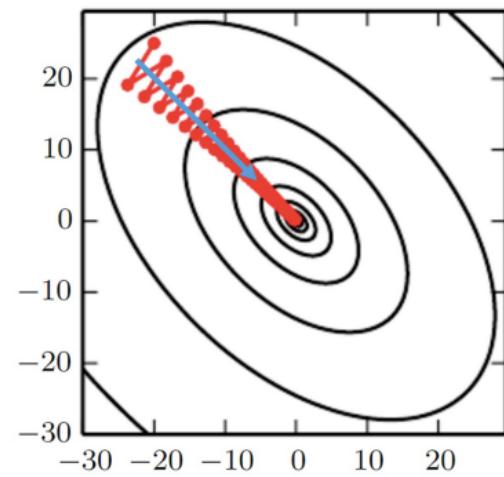
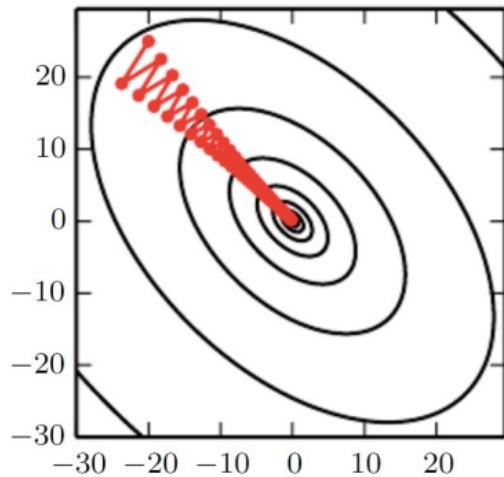
- It is common to decay the learning rate linearly until iteration  $\tau$ 
  - Can also be decayed at intervals
$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \quad (10)$$
- Three parameters to choose
- Usually:
  - $\tau$  should allow a few hundred passes through the training set
  - $\epsilon_\tau$  should be roughly 1% of  $\epsilon_0$

I problema principale è quello di come scegliere epsilon 0. Possiamo provare valori diversi, fare 100 iterazioni e scegliere il migliore risultato, e continuare con le iterazioni con quello. Valori troppo alti causano instabilità.

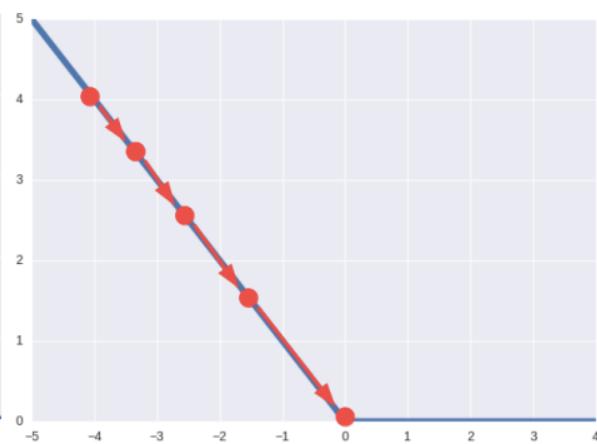
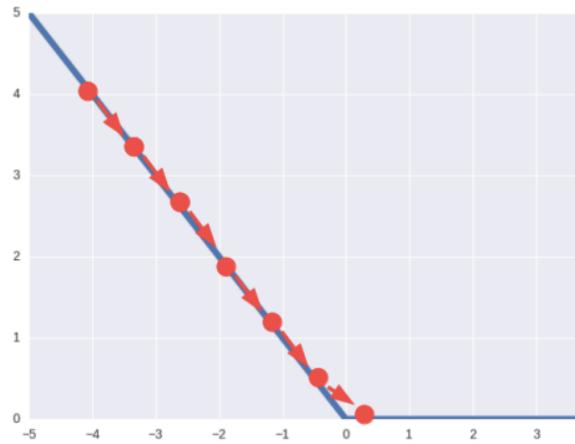
## Momentum

Il metodo del **momentum** è pensato per accelerare l'apprendimento, specialmente in fasi di alta curvatura della loss function, o se abbiamo un gradiente piccolo o rumoroso.

The method of Momentum (Polyak 1964) is designed to accelerate learning, especially in the face of high curvature, small or noisy gradients. It accumulates and exponentially decaying moving average of past gradients and continues to move in their direction



Il metodo arriva dalla fisica. Qui vediamo la differenza di velocità senza e con il momentum.



- Accumulates previous gradients

$$v \leftarrow \alpha v - \epsilon \nabla_{\theta} \left( \frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}; \theta), y^{(i)}) \right) \quad (11)$$

$$\theta \leftarrow \theta + v \quad (12)$$

where

- $\alpha \in [0, 1]$  is a hyperparameter
- $\nabla_{\theta}$  is the gradient
- $\theta$  are the network parameters

Questa funzione accumula l'informazione dei gradienti precedenti.

Procedura finale:

- Compute gradient

$$g \leftarrow \frac{1}{m} \nabla_{\theta} \left( \sum_{i=1}^m L(f(x^{(i)}; \theta), y^{(i)}) \right) \quad (13)$$

- Compute velocity update

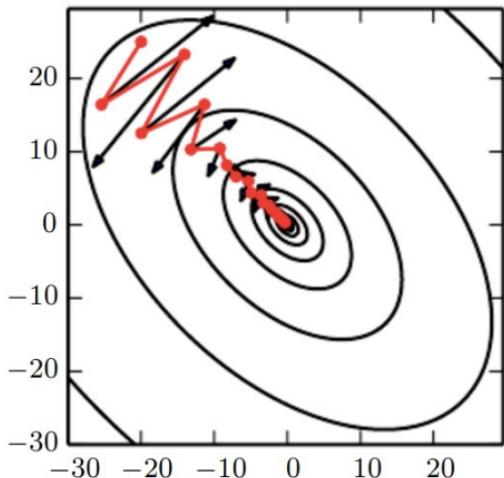
$$v \leftarrow \alpha v - \epsilon g \quad (14)$$

- Apply update

$$\theta \leftarrow \theta + v \quad (15)$$

where

- $\alpha$  is the momentum coefficient



Momentum aims primarily to solve two problems:

- poor conditioning of the Hessian matrix and
- variance in the stochastic gradient

La varianza viene ridotta, e il processo di ottimizzazione diventa più veloce.

La grandezza dello step dipende da quanto è larga e quanto è allineata la sequenza dei gradienti. Se sono allineati allora vengono accumulati.

- The step size is largest when many successive gradients point in exactly the same direction
- Common values used in practice for  $\alpha$  include 0.5, 0.9 and 0.99
- $\alpha$  may be adapted over time (start with a small value and then increase)

- Some variants of the Momentum methos exists:

**Nesterov Momentum** (Sutskever, 2013)

$$v \leftarrow \alpha v - \varepsilon \nabla_{\theta} \left[ \frac{1}{m} \sum_{i=1}^m L(f(x^{(i)}; \theta + \boxed{\alpha v}), y^{(i)}) \right]$$

It adds a correction factor to the standard momentum

In the convex batch gradient case, Nesterov momentum improves the rate of convergence

Unfortunately, this does not happens in in the stochastic gradient case !

## Algorithms with adaptive learning rate

Il learning rate impatta molto sul processo di apprendimento.

Il gradiente può essere molto sensibile in alcune direzioni e poco in altre. Ha quindi senso adattare il learning rate alla direzione, per dare diversi learning rate a diverse direzioni.

Metodo vecchio:

**Delta-bar-delta [Jacobs, 1988]**

Early heuristic approach

- Simple idea: if the partial derivative in respect to one parameter remains the same, increase the learning rate, otherwise, if that partial derivative change sign, decrease
- Must be used in batch methods (Why?)

Deve essere usato nei metodi di batch perchè qui potremmo avere più noise, e con questa procedura possiamo controllare il rumore.

## AdaGrad [Duchi et al., 2011]

Scale the gradient according to the historical norms

- Learning rates of parameters with high partial derivatives decrease fast
- Enforces progress in more gently sloped directions
- Nice properties for convex optimization
- But for deep learning decrease the rate in excess

- Accumulate squared gradients

$$r \leftarrow r + g \odot g$$

- Element-wise update

$$\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$$

- Update parameters

$$\theta \leftarrow \theta + \Delta\theta$$

where

- $g$  is the gradient
- $\delta$  is a small constant for stabilization

$\delta$ , is usually  $10^{-6}$  and is used to stabilize division by small numbers.

## RMSProp [Hinton, 2012] (Root Mean Square Propagation)

Modification of AdaGrad to perform better on non-convex problems

- AdaGrad accumulates since beginning, gradient may be too small before reaching a convex structure
- RMSProp uses an exponentially weighted moving average

It has been shown to be an effective and practical optimization algorithm for deep neural networks.

It is currently one of the go-to optimization methods employed routinely by deep learning.

Questo è il primo degli algoritmi che ha successo nel deep learning.

- Accumulate squared gradients

$$r \leftarrow \rho r + (1 - \rho)g \odot g$$

- Element-wise update

$$\Delta\theta \leftarrow -\frac{\epsilon}{\delta + \sqrt{r}} \odot g$$

$\delta$ , is usually  $10^{-6}$  and is used to stabilize division by small numbers.

- Update parameters

$$\theta \leftarrow \theta + \Delta\theta$$

where

- $\rho$  is the decay rate

$\rho$  is usually 0,9, it determines how much of the previous squared gradient is retained when computing the running average

**Adam** [Kingma and Ba, 2014] (Adaptive moments)

Adaptive Moments, variation of RMSProp + Momentum

- Momentum is incorporated directly as an estimate of the first order moment
- In RMSProp, momentum is included after rescaling the gradients
- Adam also add bias correction to the moments to account for their initialization at the origin

Viene aggiunto il momentum.

- Update time step:  $t \leftarrow t + 1$
- Update biased moment estimates

$$s \leftarrow \rho_1 s + (1 - \rho_1)g$$

$$r \leftarrow \rho_2 r + (1 - \rho_2)g \odot g$$

Suggested default values for  $\rho_1$  and  $\rho_2$  are 0.9 and 0.999 respectively

- Correct biases

$$\hat{s} \leftarrow \frac{s}{1 - \rho_1^t}$$

$$\hat{r} \leftarrow \frac{r}{1 - \rho_2^t}$$

- Update parameters

$$\Delta\theta \leftarrow -\epsilon \frac{\hat{s}}{\delta + \sqrt{\hat{r}}}$$

Questi sono gli algoritmi più usati. Sono tutti varianti della discesa del gradiente. Non c'è una regola per scegliere l'algoritmo.

Gli algoritmi con il learning rate adattivo sono solitamente migliori. Quindi è meglio iniziare con questi. Sono più robusti, al posto di imporre un learning rate. Non c'è un algoritmo migliore tra questi, dipende dal problema.

Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta and Adam.

The choice of which algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm (for ease of hyperparameter tuning).

---

## Autoencoders

Gli autoencoders sono reti neurali dove l'output è uguale all'input, è una copia.

La parte interessante non è l'output, ma per la **rappresentazione nascosta (code)** che si ottiene.

La rappresentazione nascosta contiene tutte le informazioni che ho bisogno per ricostruire l'output.

Quindi hanno due parti: un encoder e un decoder.

La forma più semplice di autoencoder è l'**undercomplete autoencoder**. Fanno in modo che il **code** abbia una dimensione più piccola dell'input.

Un hidden layer di solito è sufficiente.

## Regularization

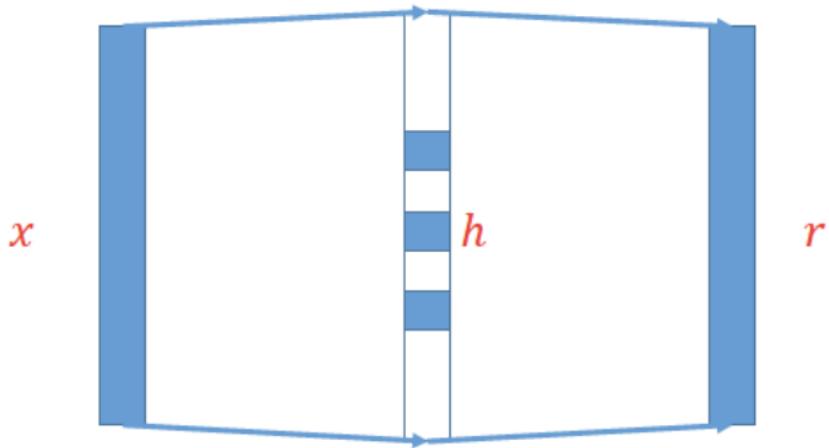
- ✧ Typically NOT Keeping the encoder/decoder shallow or  
Using small code size
  
- ✧ **Regularized autoencoders:** add regularization term that  
encourages the model to have other properties
  - Sparsity of the representation (sparse autoencoder)
  - Robustness to noise or to missing inputs (denoising-autoencoder)
  - Smallness of the derivative of the representation

## SPARSE AUTOENCODERS

Constrain the code to have sparsity

Training: minimize a loss function

$$L_R(x, g(f(x))) + \Omega(h)$$



è principalmente usato nell'image processing.

Un autoencoder che è stato regolarizzato per essere sparso deve rispondere a delle features statistiche del dataset, piuttosto che funzionare come una funzione identità.

---

Un altro modo è aggiungere noise all'input e poi ricostruire l'input senza noise.

# DENOISING AUTOENCODERS

Autoencoders encourage to learn  $g(f(x))$  as an identity

Denoising autoencoders minimize a loss function

$$L(x, r) = L(x, g(f(\tilde{x})))$$

Where  $\tilde{x} = x + \text{noise}$

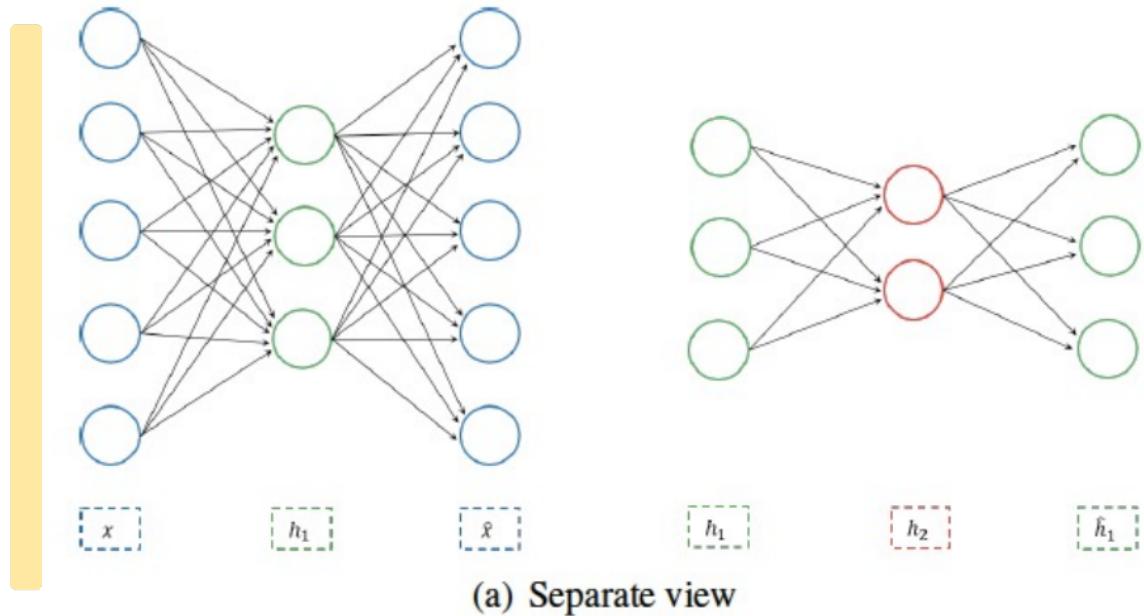
Denoising autoencoders must undo this corruption rather than simply copying their input

---

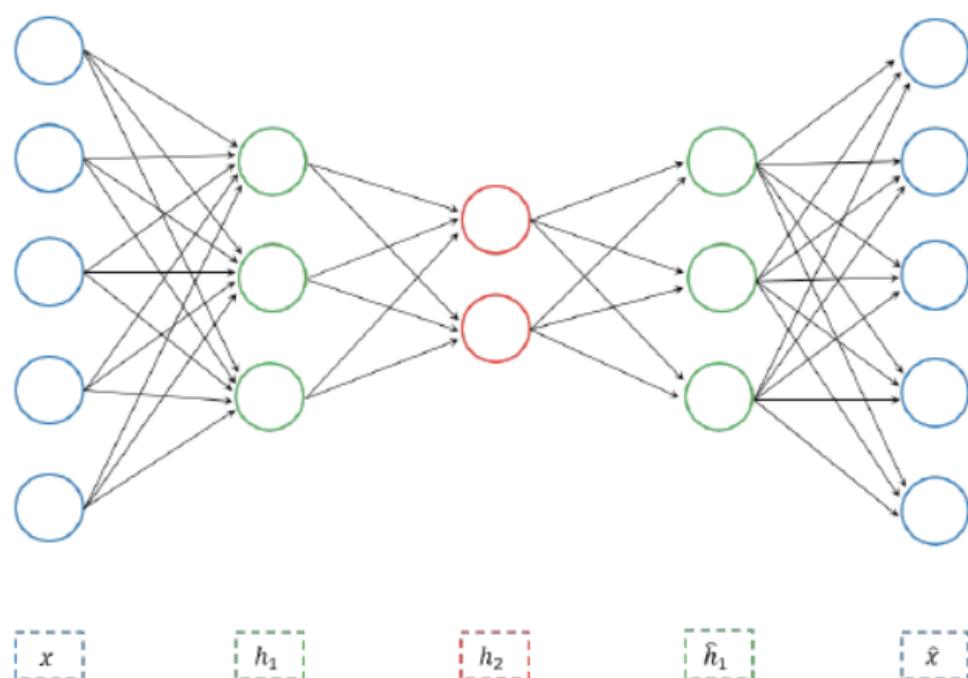
Usati quando i dati di input sono molto non lineari, non correlati? Sono un autoencoder dopo l'altro.



## STACKED AUTOENCODERS



# STACKED AUTOENCODERS



(b) Joint view

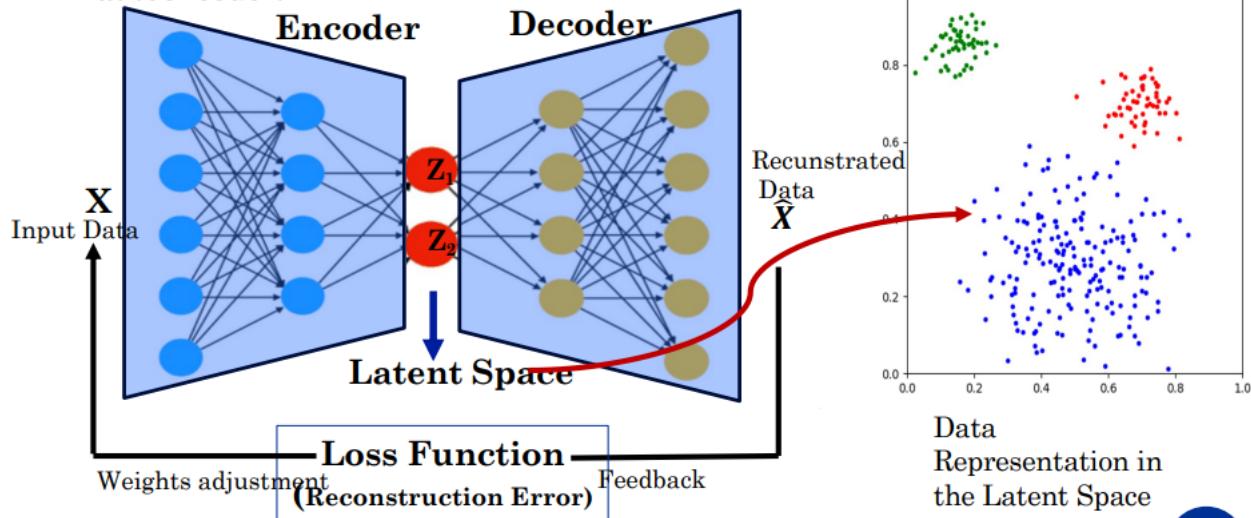
Sono diversi rispetto al feedforward neural network perché ogni layer è dato come input al prossimo stacked model.

Questa è la rappresentazione generale di un autoencoder. Il vettore centrale è quello che contiene la mia caratterizzazione dei dati.



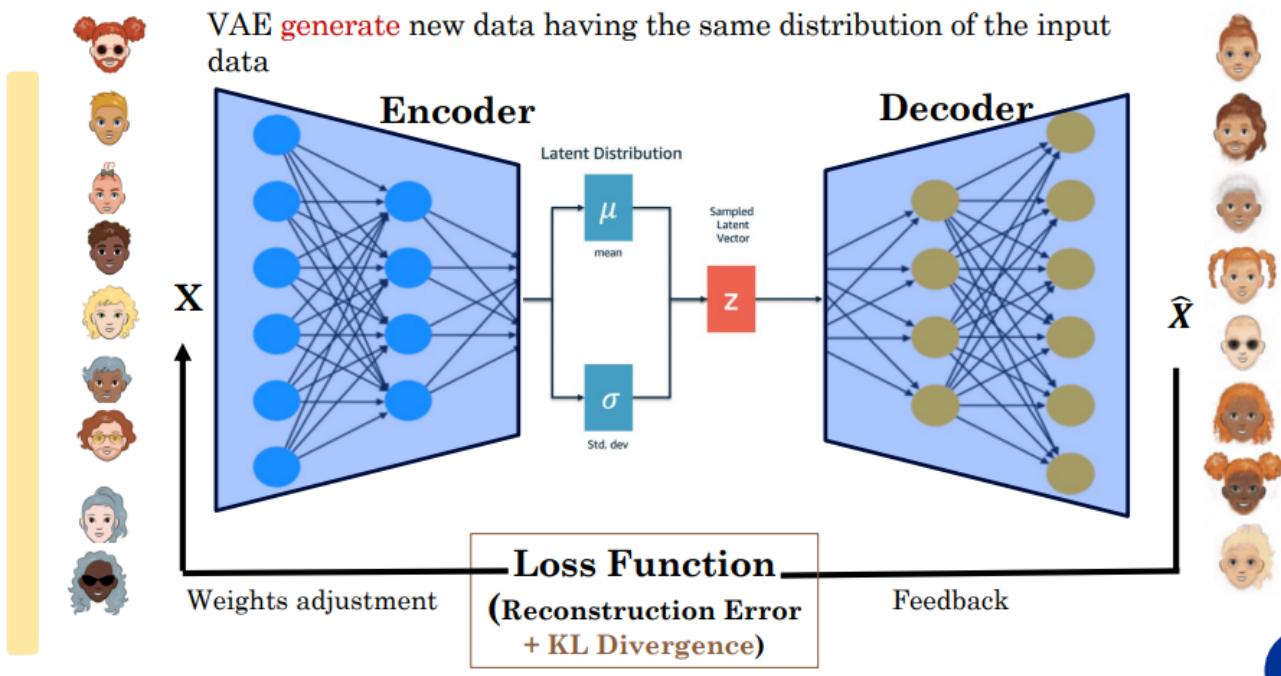
## AUTO-ENCODERS

The quintessential example of a representation learning algorithm is the **autoencoder**.



Se ho un hidden layer fatto da 2 unità posso metterli su un piano a 2 dimensioni e poi fare clustering.

Questo encoder impara un vettore singolo che vale per tutti gli input. In questo modo si impara la distribuzione dei dati. Imparo una distribuzione multivariata ogni elemento ha una media e una varianza.



Questi funzionano bene quando vogliamo ottenere un modello probabilistico dei miei dati, che posso usare anche per generare dati che sono simili all'input, ma non uguale.

Rispetto agli autoencoders, al posto di imparare un vettore, imparo 2 vettori che sono la media e la varianza dei dati ?penso?