Lez9_regolarizzazione_ottimizzazione

November 9, 2023

1 Lez_9 Regolarizzazione e ottimizzazione

Il training necessita sia della **regolarizzazione** sia della **ottimizzazione**.

La *Regolarizzazione* mitiga un problema chiave del Machine Learning, ovvero come fare in modo che l'algoritmo si comporti bene sia su training set che su nuovi input. Idea ridurre errore su test set. Costo di perdere un po' di prestazione su training set. Esistono diverse tecniche, per esempio:

- *Tecniche basate su regole*: limitare numero di livelli nella rete.
- *Inclusione di termini aggiuntivi*: Nella funzione obiettivo dell'addestramento, che influiscono sulla capacità del modello di generalizzare.
- *Ensamble*: L'uso di metodi di ensemble, che combinano più modelli per ridurre la varianza e migliorare la generalizzazione. (ho più modelli e aggrego il risultato di tutti i modelli).

In generale si accetta di *avere più bias per diminuire la varianza dati nuovi input.*

1.1 Penalità relative alla norma dei parametri

Quando devo ottimizzare una funzione costo, la definisco come una funzione di loss a cui aggiungo un termine di regolarizzazione.

Abbiamo la **nuova funzione obiettivo regolarizzata** $\bar{J}(\theta; X, y) = \mathcal{L}(\theta; X, y) + \lambda\Omega(\theta)$

dove:

- λ è un iperparametro che dà un peso a $\Omega(\theta)$, se è piccolo non ha impatto. Può essere specifico per ogni livello, incrementando il numero di hyperparametri.
- $\Omega(\theta)$ funzione dei parametri del modello e cattura la penalità che ho in base ai termini del modello. Può assumere diverse forme a seconda della tecnica di regolarizzazione utilizzata, come ad esempio la norma L1 o L2, che sono utilizzate nella regolarizzazione L1 e L2 rispettivamente. Solitamente è riferito ai soli pesi W, non ai bias, poichè ciò può portare ad *underfitting.*
- $\mathcal{L}(\theta; X, y)$ è la funzione di perdita (loss function) che misura l'errore del modello sui dati di addestramento.

Questo approccio consente di controllare l'equilibrio tra la minimizzazione dell'errore di addestramento e la riduzione della complessità del modello attraverso la penalizzazione dei parametri . L'iperparametro λ regola l'intensità della regolarizzazione: valori più alti di λ portano a una maggiore enfasi sulla penalità e quindi a modelli più semplici, mentre valori più bassi di λ permettono al modello di adattarsi meglio ai dati di addestramento a costo di una possibile overfitting.

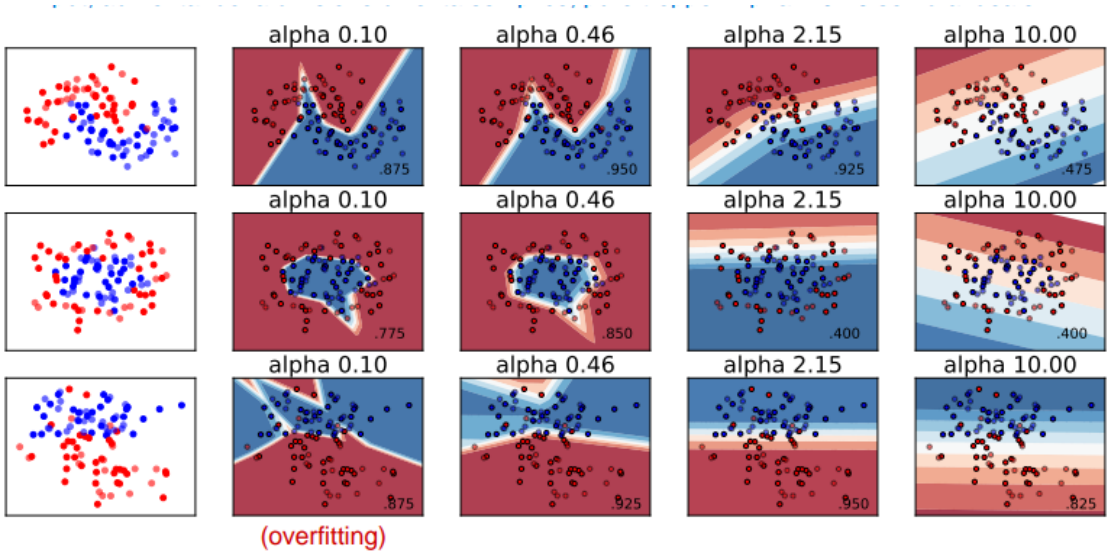
Quando vado a regolarizzare, usiamo la **norma L2** : $\Omega(\theta) = \frac{1}{2}\|w\|_2^2$ oppure la **norma L1** :

$$\|w\|_1$$

Così sto dicendo all'algoritmo di ottimizzazione che non voglio valori troppo grandi per i parametri W. Il valore ottimale sarebbe 0, meglio di così non posso fare ma la *loss* sarà molto grande. Bisogna trovare un compromesso, quindi l'algoritmo cerca di mettere a 0 più pesi possibili. Questo vuol dire che sto incentivando l'algoritmo di training a capire che se *alcuni neuroni non hanno necessità per il livello successivo può mettere a 0 quel peso* (sto chiedendo di eliminare connessioni nella rete, semplificando il più possibile il modello per avere meno rischi di overfitting).

1.2 Esempio

Preso da scikit-learn. Dati tre dataset fa vedere il diverso impatto di lambda (alpha in figura).



Viene addestrato un semplice classificatore. Per il valore di lambda più piccolo ($\lambda = 0.10$) il modello impara perfettamente come sono distribuiti i dati in input (*overfitting*). All'aumentare di lambda, il modello impara una cosa sempre più semplice, fino ad arrivare ad $\lambda = 10.00$, in cui apprende una cosa troppo semplice e non corretta (underfitting).

Regolarizzazione utile (evita overfitting) ma bisogna ottimizzare anche l'iperparametro lambda.

Per implementare questa cosa ci sono parametri che possono essere passati al kernel di keras, in particolare regularizers (hai due parametri primo per kernel che minimizza i pesi e scegli se minimizzare norma o errore quadratico medio con lambda, il secondo è per il bias però di solito non si usa). Specificabile per ogni livello.

[ ]:
```
from keras import regularizers
layer = layers.Dense(5, kernel_regularizer=regularizers.L1(0.01),␣
  ↪bias_regularizer=regularizers.L2(0.01))
```

1.3 Dataset Augumentation

Il modo migliore per far sì che un modello generalizzi meglio è addestrarlo su un maggior numero di dati. Tuttavia, nella pratica, la quantità di dati a disposizione è spesso limitata. Per affrontare questa limitazione, a volte è possibile generare dati sintetici o falsi (**fake data**) e aggiungerli al set di addestramento. La fattibilità e la facilità di farlo dipendono in gran parte dalla natura del compito.

Ad esempio, in compiti come il riconoscimento di oggetti, è possibile generare nuovi dati di input applicando trasformazioni alle immagini esistenti, come rotazioni, traslazioni o altre distorsioni. Questo può aiutare il modello a imparare a gestire le variazioni nei dati di input e, di conseguenza, migliorare la sua capacità di generalizzazione.

Tuttavia, l'efficacia della generazione di dati sintetici dipende dalla natura del compito e dalla qualità dei dati generati. A volte, la generazione di dati sintetici può introdurre bias o non rappresentare adeguatamente la distribuzione dei dati del mondo reale, quindi è importante essere cauti nell'uso di questa tecnica. Inoltre, in alcuni casi, è più semplice generare dati sintetici rispetto ad altri, e la qualità dei dati generati è fondamentale per il successo del modello. Quindi potremmo creare un modello più robusto, oppure sbilanciare troppo il training set.

1.4 Noise injection

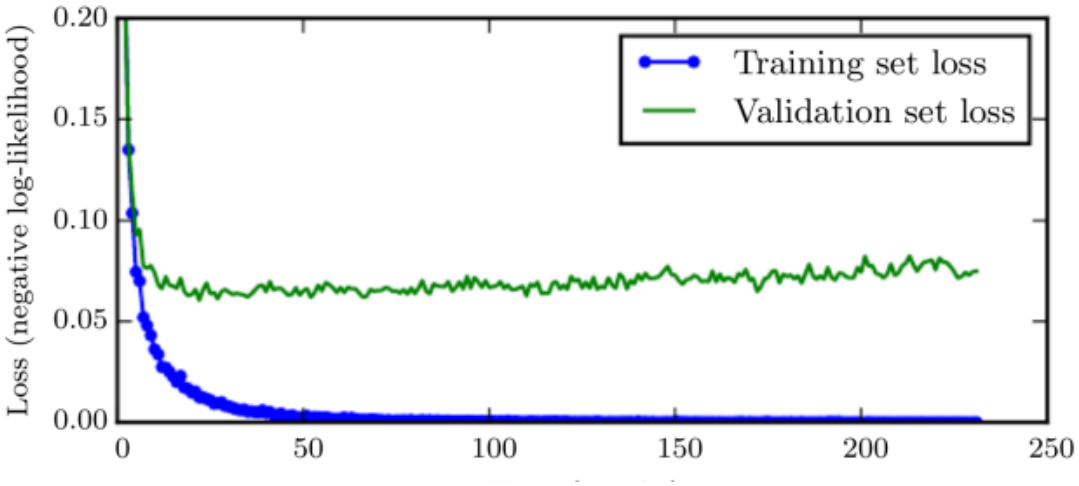
Per migliorare la capacità di generalizzazione di una rete neurale e renderla più robusta al rumore nell'input, è possibile utilizzare diverse tecniche. Queste tecniche aiutano la rete a diventare più resiliente alle variazioni e alle interferenze nei dati. (esempio: una foto che ritaglio o scurisco leggermente). Ciò che possiamo fare è:

1. **Aumento dei dati con Rumore nell'Input:** Un modo per migliorare la robustezza è aggiungere rumore ai dati di input durante l'addestramento. Si introducono intenzionalmente variazioni nei dati di addestramento per aiutare il modello a generalizzare meglio alle situazioni reali. Ad esempio, è possibile introdurre perturbazioni casuali nei dati di input aggiungendo piccole quantità di rumore gaussiano. Questo può essere particolarmente utile in compiti come la classificazione delle immagini, dove è possibile applicare trasformazioni casuali alle immagini di input, come rotazione, ridimensionamento o ribaltamento. Coincide con **Dataset Augmentation**.
2. **Regolarizzazione dei Pesi con Rumore:** Un'altra tecnica consiste nell'aggiungere rumore ai pesi della rete durante l'addestramento. Quindi lavoriamo sulla matrice W. Questo può essere visto come una forma di regolarizzazione dei pesi. Introducendo rumore ai pesi, si impedisce alla rete di adattarsi troppo ai dati di addestramento e la rende più robusta alle variazioni. Quindi non solo bisogna trovare il set di parametri che si adattano meglio ai dati in input, ma devo essere ben adattabili anche a piccoli cambiamenti. Ad esempio, è possibile utilizzare layer come `keras.layers.NoisyDense` per aggiungere rumore ai pesi di specifici strati nella rete neurale.
3. **Rumore sulle Unità Nascoste:** È anche possibile aggiungere rumore alle attivazioni o alle unità nascoste all'interno della rete. Strati come `keras.layers.GaussianNoise` possono essere applicati alle unità nascoste della rete per introdurre rumore casuale. Ad esempio, aggiungiamo rumore all'output di un neurone nascosto $h^{(n)}, e quindi passiamo questo neurone

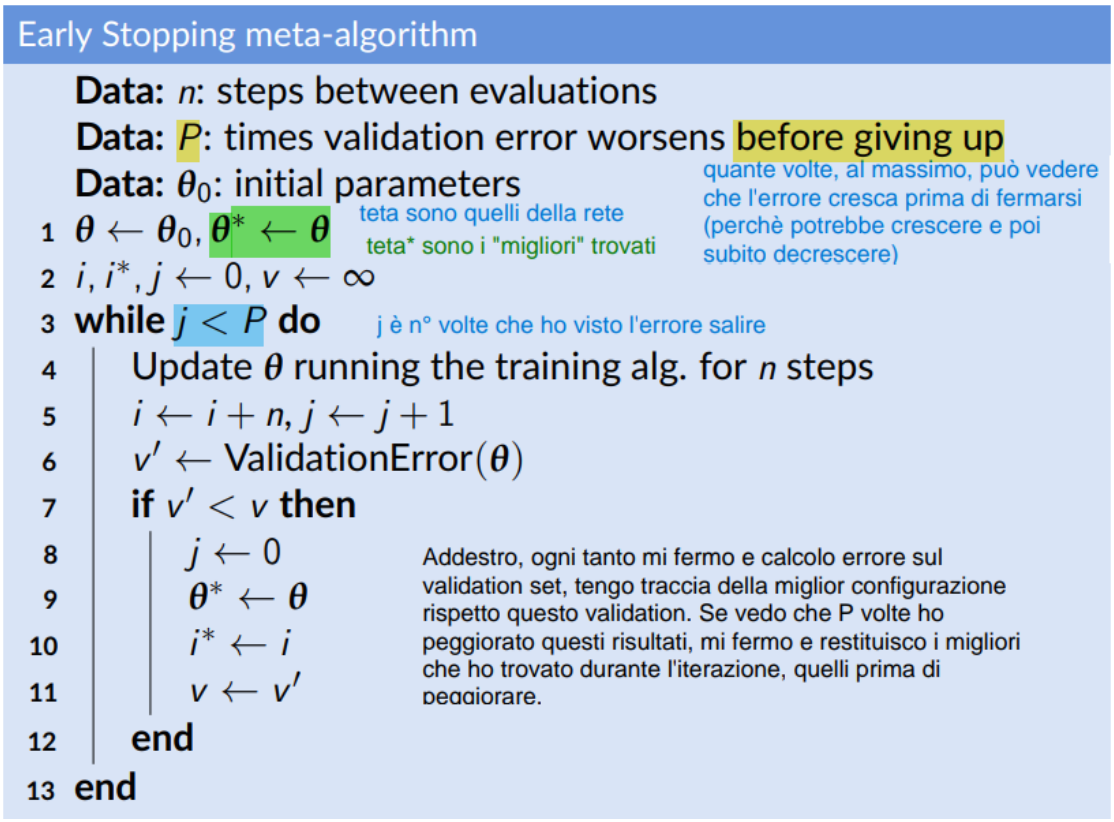
1.5 Early Stopping

Quando si addestra una rete, soprattutto se è grande, il modello definito presenta il rischio di cadere in *overfitting*, perché non sappiamo esattamente il numero ideale di unità dei neuroni e dei livelli per risolvere un task. Quindi bisogna identificare un modello abbastanza complesso per comprendere quel task, in più gli algoritmi di addestramento non hanno la certezza di trovare ottimo globale.

Se osservo *loss su training* e *validation* può succedere che la *loss* scenda sempre (altrimenti sarebbe un problema!). Al contrario, la *loss su validation set* scende all'inizio; poi potrebbe cominciare a salire (overfitting, la rete ha imparato quello che c'era da imparare e si sta specializzando su training set alle spese dei nuovi dati).



In questo caso potrei fermarmi prima, questa è l'idea dell'*early stopping* (prendere i parametri migliori, se vedo che le cose peggiorano mi fermo).



L'algoritmo richiede 3 parametri:

- n : Quanti step di addestramento voglio fare tra 2 verifiche.
- P : Pazienza dell'early stopping (quante volte errore su validation set peggiora prima di fermarsi).
- θ_0: Vettore di parametri iniziali. Si mantengono due coppie di parametri, θ_0 e θ_0^*, questi ultimi sono i migliori parametri trovati fino ad ora.

Abbiamo anche j, che rappresenta il numero di volte che ho visto il *validation error* aumentare. Se supera il valore limite di P finisco il training.

Durante il training: - aggiorno i parametri θ, facendo n iterazioni, con n solitamnete multiplo delle epoche.

- Incremento i, che conta le iterazioni; j lo incremento di 1.

Se l'errore sul validation set si è ridotto: - j lo pongo a 0.

- θ_0^* lo metto pari a θ_0.
- i^* diventa pari a i.
- v diventa pari a v'.

Addestro, ogni tanto mi fermo, calcolo l'errore sul validation set, e mantengo la configurazione migliore. I parametri che restituisco sono i migliori che avevo visto rispetto al validation error, dopo P peggioramenti di fila mi fermo.

Questa tecnica è la forma di regolarizzazione più usata, è una cosa di buon senso evitare di spendere ore per addestrare una rete mandandola in overfitting. Il problema è anche quello di ottimizzare l'iperparametro del numero di epoche (con early stopping metto un valore grande, l'algoritmo si fermerà da solo quando necessario).

Ho un beneficio in termini computazionali poichè riduco il tempo di addestramento. L'altra faccia della medaglia è che comunque sto aggiungendo overhead (anche se posso velocizzare le cose tramite *parallelismo*). C'è overhead anche rispetto alla memoria perché devo tenere in memoria anche una copia dei parametri migliori, raddoppia la richiesta in memoria.

L'*Early stopping* ha bisogno della creazione di un *validation_set*, ma ci sono casi in cui ho pochi dati per l'addestramento e toglierne altri per il validation_set diventa difficile. Posso mitigare in questo modo: completato il training, continuo ad addestrare il modello con i dati del *validation_set*.

Quindi, dopo aver trovato con early stopping il numero giusto di iterazioni, addestro il modello con tutti i dati e con quel numero di iterazioni. Devo addestrare due volte la rete. Finito l'early stopping, faccio qualche altra iterazione (numero a tentativi, potrebbe essere un problema deciderlo) con tutti i dati.

Per utilizzare early stopping devo utilizzare callback (funzioni usate durante il training).

```
callback = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=3
)
# ...
history = model.fit(
    ...,
    epochs=10,
    batch_size=16,
    callbacks=[callback]
)
```

L'early stopping verifica le condizioni e se necessario ferma il training. Bisogna indicare **monitor** (metriche da monitorare) e **min_delta**, ossia il cambiamento minimo per definire un miglioramento.

1.6 Bagging

In generale, invece di addestrare una rete, ne addrestro n, e poi vedo tutte le reti. Vedo ciò che dice la maggioranza, ossia un errore viene compensato dagli altri. Nel caso delle reti, è semplice ottenere modelli diversi, perchè sensibili, cambio una cosa e la differenza è tangibile.

1.7 Dropout

Tecnica recente, è una approssimazione del bagging, con un numero esponenziale di reti diverse. Modo efficiente di avere tantissime reti diverse nella stessa rete. Scartiamo alcuni neuroni a caso della rete mentre la uso, basta mettere uscita corrispondente a tale unità a 0. Lo faccio ad ogni step,è come se generassi una rete diversa per ogni mini batch nel mio training, ma sempre nella stessa rete.

Supponiamo che $p(l)$ sia la probabilità di dropout per il livello l, con valori compresi tra 0 e 1.

- Di solito 0.8 per il livello di input e 0.5 per le unità nascoste. Queste sono probabilità di *tenerlo*. Il complementare è probabilità di toglierlo.
- Definiamo $h(l)$ come l'output del livello l (dove $h(0) = x$). Per l'unità i nei livelli da 0 a $l-1$, ad ogni passaggio di addestramento abbiamo: $\bar{h}(l)_i = \begin{cases} h(l)_i & \text{con probabilità } p \\ 0 & \text{con probabilità } (1-p) \end{cases}$

È come se stessimo utilizzando una diversa rete neurale (con parametri condivisi) per ogni esempio nel set di addestramento. Ciò vuol dire non usare alcuni parti dell'input (cioè feature), forzando la rete ad apprendere in maniera robusta con l'input fornito. Tali variazioni, rispetto la singola feature, non dovrebbero essere cosi impattanti, e quindi la fortifico.

Dropout SOLO nel training. Dopo training, scalo i pesi moltiplicando per le probabilità, questo per avere valori in range sensati.

Esempio Prendiamo $y_1 = W^{(2)} \cdot h = \sum h_i$, se i vari h_1 sono 1, abbiamo circa 3 (per semplicità). Se tolgo un'unità, avrei somma 2, quindi range di valore diverso, potrebbe alterare le predizioni fatte, perchè y si aspetterà 2 e non 3. Se usassi gli stessi parametri, y avrebbe valori più vicini a 3 che a 2, per risolvere, prendo i pesi W e li moltiplico per p, è un fattore di correzione.

Dopo l'addestramento, possiamo utilizzare la rete neurale senza dropout ridimensionando i pesi: $W(l)_{test} = p(l)W(l)$.

1.7.1 Dropout in TensorFlow

Il Dropout è disponibile come layer in Keras. Il layer Dropout imposta casualmente le sue unità di input a 0 con una frequenza di rate ad ogni passaggio durante l'addestramento. Gli input non impostati a 0 vengono ridimensionati di un fattore di $\frac{1}{(1-rate)}$ in modo che la somma su tutti gli input rimanga invariata.

Keras esegue anche il ridimensionamento degli input durante l'addestramento. Questo è leggermente diverso dal ridimensionamento dei pesi solo dopo l'addestramento (come descritto in precedenza), ma è equivalente in pratica.

Ecco il codice Keras per creare un layer Dropout con rate 0.5 e una forma di input di (2,):

```
layer = tf.keras.layers.Dropout(0.5, input_shape=(2,))
```

Quindi Keras applica la correzione nel training, ma utilizzando il reciproco delle probabilità.

Caso d'uso - Notebook Spotify Dopo 10 epoche, si nota un caso di overfitting dei dati nel grafico. Per mitigare questo problema, si apportano due modifiche al modello. Innanzitutto, si introduce un layer Dropout subito dopo il livello di input (il primo strato della rete). In secondo luogo, si aggiunge un secondo layer Dropout dopo il secondo livello nascosto. Queste modifiche hanno contribuito a migliorare le prestazioni del modello e a evitare l'overfitting dei dati.

1.8 Algoritmi di ottimizzazione per NN

Il machine learning si basa su principi di ottimizzazione matematica, in particolare sull'ottimizzazione di una funzione di perdita (loss function). L'obiettivo è trovare i parametri

del modello che minimizzano questa funzione di perdita, il che rappresenta un problema di ottimizzazione. La discesa del gradiente è uno dei principali algoritmi utilizzati per raggiungere questa ottimizzazione.

Tuttavia, quando ci si trova ad affrontare task complessi, addestrare una rete neurale di grandi dimensioni può richiedere molto tempo, anche giorni o mesi. In queste situazioni, non è sensato utilizzare il primo algoritmo disponibile. Pertanto, è importante studiare e comprendere gli algoritmi di ottimizzazione più avanzati.

1.8.1 Learning vs pure optimization

La differenza fondamentale tra ottimizzazione tradizionale e machine learning risiede nella natura indiretta dell'ottimizzazione nel machine learning. Nella ricerca operativa classica, l'obiettivo è trovare una combinazione di variabili che minimizzino o massimizzino una funzione obiettivo specifica. Questa funzione obiettivo è chiaramente definita, e l'obiettivo è trovare la soluzione ottimale in base a essa.

Nel machine learning, l'ottimizzazione avviene in modo indiretto. L'obiettivo non è minimizzare o massimizzare una funzione obiettivo predeterminata, ma piuttosto addestrare un modello che possa fare previsioni accurate su nuovi dati. La misura di prestazione del modello, come ad esempio l'accuratezza su un set di test, può essere una metrica intrattabile da ottimizzare direttamente. Inoltre, il costo (loss) ottimizzato durante l'addestramento della rete può differire dalla metrica di valutazione, come l'errore quadratico medio.

Nel machine learning, l'algoritmo di ottimizzazione lavora per minimizzare la funzione costo (loss function) associata al modello, e non è necessariamente interessato a minimizzare direttamente la metrica di valutazione del modello. L'obiettivo principale è addestrare il modello in modo che si comporti bene sulla metrica desiderata, ma l'algoritmo di ottimizzazione agisce in base alla funzione costo.

In sintesi, nel machine learning, non si sta minimizzando direttamente la metrica di valutazione del modello, ma si sta ottimizzando la funzione costo per addestrare il modello in modo che si comporti bene sulla metrica desiderata, anche se la funzione costo può differire dalla metrica di valutazione effettiva.

1.8.2 Sfide nell'ottimizzazione NN

L'ottimizzazione è una sfida; molti modelli di machine learning sono stati attentamente progettati per affrontare problemi di ottimizzazione **convessa** (che possono comunque essere complessi!). In molti studi, trovare un minimo locale è sufficiente (sarà anche un minimo globale).

Nell'addestramento delle reti neurali però, dobbiamo affrontare il caso generale non convesso.

1.8.3 Minimi Locali

Le reti neurali profonde possono presentare numerosi minimi locali, ed è una questione complessa. Non è garantito che un minimo locale sia simile al minimo globale, e spesso possono essere molto diversi. In reti neurali di grandi dimensioni, i minimi locali tendono ad essere vicini ai minimi globali, anche se non esiste un teorema che possa garantire questa affermazione; è una scoperta empirica.

Uno dei problemi principali legati ai minimi locali è che in tali punti, il gradiente della funzione di perdita è nullo, e ciò può portare ad una "stagnazione" dell'ottimizzazione in queste zone. Tuttavia, l'utilizzo di tecniche di discesa del gradiente stocastica risulta efficace nel superare questi punti di sella.

1.8.4 Vanishing & exploding gradients

Il fenomeno dei "vanishing gradients" (gradiente che tendono a sparire) e degli "exploding gradients" (gradiente che tendono a esplodere) è un problema comune in reti neurali profonde.

Nei casi dei "vanishing gradients," quando calcoliamo i gradienti all'indietro attraverso la rete, se gli autovalori delle matrici dei pesi sono inferiori a 1 in valore assoluto, i gradienti diventano sempre più piccoli man mano che ci spostiamo all'indietro nella rete. Questo porta a gradienti molto vicini a zero, che possono causare un rallentamento dell'addestramento, rendendo la convergenza lenta o addirittura impossibile.

D'altra parte, negli "exploding gradients," quando gli autovalori delle matrici dei pesi sono superiori a 1 in valore assoluto, i gradienti crescono esponenzialmente man mano che ci spostiamo all'indietro nella rete. Questo può portare a gradienti molto grandi che possono causare problemi di instabilità numerica e rendere l'addestramento instabile.

Questi eventi interessano reti molto profonde, partendo dall'ultimo livello verso il primo livello.

La buona notizia è che questa problematica riguarda **reti ricorrenti**, in cui le matrici dei pesi sono condivise tra i diversi passaggi temporali. Noi lavoriamo con una matrice W per ogni livello, mitigando tali effetti.

[ ]: