Fondamenti di Machine Learning

Laurea Triennale in Ingegneria delle Comunicazioni

7: Reti neurali

Lecturer: S. Scardapane

SAPIENZA
UNIVERSITÀ DI ROMA

Reti neurali feedforward

Limitazioni dei modelli lineari

Per comprendere le limitazioni di un modello lineare $f(\mathbf{x})$, consideriamo un vettore di input $\widehat{\mathbf{x}}$, uguale a $\mathbf{x}$ eccetto $\widehat{x}_i = 2x_i$ (ad esempio, un cliente con il doppio del reddito).

L'output sui due input è dato da:

$$f(\widehat{\mathbf{x}}) = f(\mathbf{x}) + w_i x_i \,. \tag{1}$$

Gli effetti sono sovrapposti linearmente: è impossibile modellare qualche forma di interazione tra due caratteristiche (ad esempio, '**un cliente con un reddito di 10k non è affidabile, *a meno che* non sia molto giovane**').

Un altro esempio di grande interesse storico è lo **XOR** (exclusive OR): dati due input binari x_1 e x_2, l'output dello XOR è positivo solo se i due input hanno lo stesso valore:

$$f(0,0) = 1\,, \quad f(0,1) = 0\,, \quad f(1,0) = 0\,, \quad f(1,1) = 1 \tag{2}$$

Nonostante la sua semplicità, questo dataset non è **linearmente separabile**.

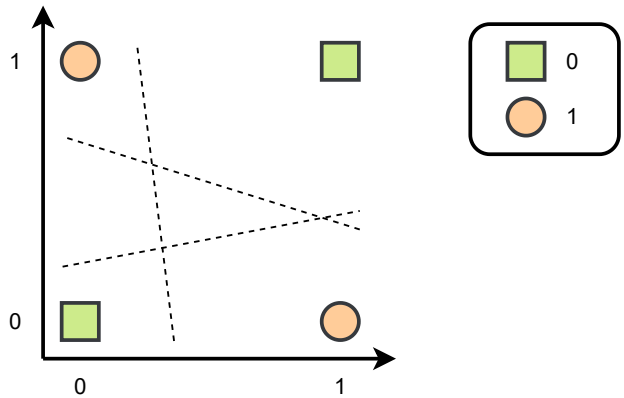


Figure 1: XOR dataset (i quadrati sono classi positive, i cerchi classi negative). E' impossibile trovare un modello lineare che li separi perfettamente.

Una osservazione interessante è che lo XOR diventa linearmente separabile se aggiungiamo la feature $x_1 x_2$ come input. Questo è un principio abbastanza comune in informatica: un problema all'apparenza complesso può diventare semplice se viene decomposto in sotto-problemi da risolvere in sequenza.

Nel caso generale, però, è impossibile costruire queste feature aggiuntive a mano, ed abbiamo bisogno che tutto venga ottimizzato in maniera automatica. Rimanendo nel campo di modelli differenziabili, questo porta alle **reti neurali**.

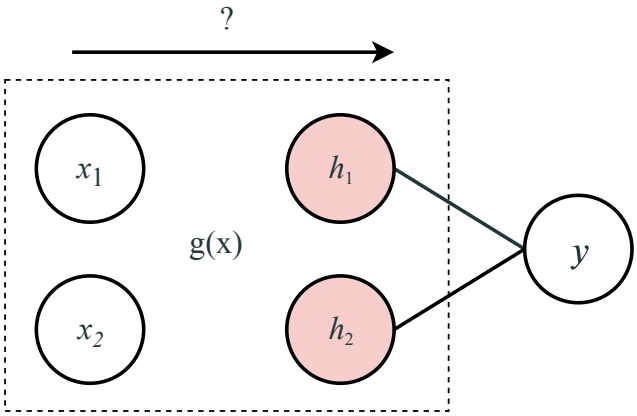


Figure 2: Come possiamo definire modelli strutturati su più livelli?

Reti neurali feedforward

Polynomial interpolation

Sulla base della discussione precedente, consideriamo un modello di questo tipo, un classificatore lineare applicato su una transformazione non-lineare $\phi : \mathbb{R}^d \to \mathbb{R}^e$ dell'input:

$$y = \mathbf{w}^\top \phi(\mathbf{x}) \tag{3}$$

Se ϕ è una trasformazione definita a priori, non allenabile, questo non cambia la convessità del problema di ottimizzazione o la soluzione. Il processo di costruzione di ϕ viene detto **feature engineering**.

Consideriamo nuovamente un dataset definito da $\mathbf{X} \in \mathbb{R}^{n \times d}$ e $\mathbf{y} \in \mathbb{R}^n$, dove n è la dimensione del dataset. Definiamo una nuova matrice contenente tutte le espansioni non-lineari dell'input:

$$\Phi = \begin{bmatrix} \phi(\mathbf{X}_1)^\top \\ \vdots \\ \phi(\mathbf{X}_n)^\top \end{bmatrix} \tag{4}$$

Supponendo un problema di ottimizzazione di tipo least-squares, possiamo scrivere la soluzione in forma chiusa come:

$$\mathbf{w} = \left(\Phi^\top \Phi\right)^{-1} \Phi^\top \mathbf{y} \tag{5}$$

A meno di una forte conoscenza di dominio, definire feature efficienti per un problema di ottimizzazione è molto difficile. In un caso 1D ($d = 1$) possiamo definire tutti i polinomi di ordine $1, \dots, M$, dove M è un iperparametro:

$$\phi(x) = \left[1, x, x^2, x^3, \dots, x^M\right] \tag{6}$$

Nel caso multidimensionale, possiamo considerare tutti i prodotti di feature $x_1^{i_1} x_2^{i_2} \cdots x_d^{i_d}$ tali che $\sum_j i_j \leq M$, ma la dimensionalità di ϕ cresce in maniera esponenziale.

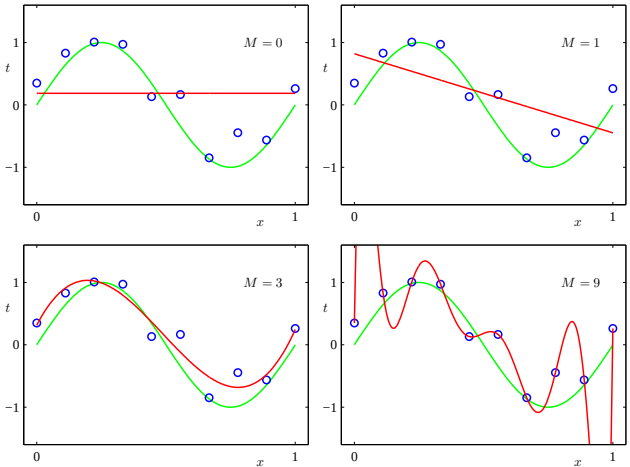


Figure 3: Esempio di regressione polinomiale (riprodotto da Bishop, 2006).

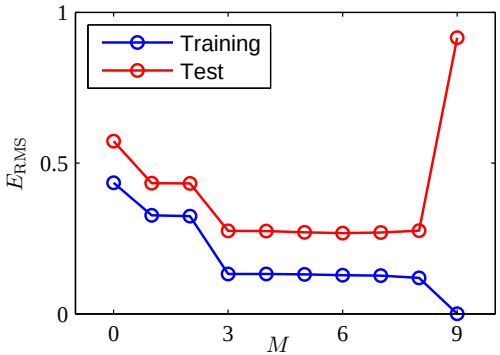


Figure 4: Grafico dell'errore su training set e test set nell'esempio di regressione lineare di prima (riprodotto da Bishop, 2006). Con $M = n$, la curva può interpolare *perfettamente* il training set (zero errore).

Possiamo notare come l'overfitting in questo caso corrisponda a valori molto alti nel vettore dei pesi $\mathbf{w}$. Informalmente, più grandi i pesi (sia in positivo che in negativo), maggiori le oscillazioni della funzione f.

Un modo comune di limitare questo fenomeno è *penalizzare* nella loss function questi valori:

$$L(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \mathbf{w}^\top \phi(\mathbf{X}_i)\right)^2 + \underbrace{\lambda\|\mathbf{w}\|^2}_{\sum_j w_j^2} \tag{7}$$

dove lo scalare $\lambda \geq 0$ è un nuovo iper-parametro che controlla quanto è forte questa penalizzazione. Con $\lambda = 0$ torniamo al classico least-squares, per $\lambda = \infty$ otteniamo $\mathbf{w} = \mathbf{0}$.

Questa tecnica viene detta **regolarizzare** il problema di ottimizzazione. Nel caso del least-squares, regolarizzare con la norma Euclidea viene detto **ridge regression** o **Tikhonov regularization**.

In questo caso, tra l'altro, possiamo scrivere la soluzione sempre in forma chiusa con una piccolissima variante:

$$\mathbf{w} = \left(\Phi^\top \Phi + \lambda \mathbf{I}\right)^{-1} \Phi^\top \mathbf{y} \tag{8}$$

Nel caso generale, scrivendo come $L(\mathbf{w})$ il problema non regolarizzato, possiamo riscrivere una versione regolarizzata come:

$$L_{\text{reg}}(\mathbf{w}) = L(\mathbf{w}) + \lambda R(\mathbf{w}) \tag{9}$$

A seconda di $R(\mathbf{w})$ possiamo ottenere diversi tipi di regolarizzazione, es., $R(\mathbf{w}) = |\mathbf{w}| = \sum_j |w_j|$ promuove la *sparsità* del vettore dei pesi.

La regolarizzazione non rientra nell'interpretazione maximum-likelihood, ma richiede una interpretazione probabilistica più ampia, che viene a volte detta **Bayesiana**.

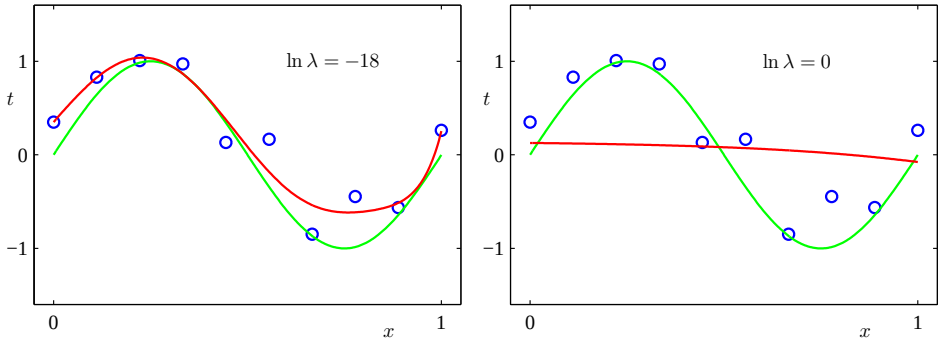


Figure 5: Ridge regressione nel caso polinomiale di prima, con $M = 9$ (riprodotto da Bishop, 2006).

Reti neurali feedforward

Fully-connected layers

Possiamo stratificare due modelli lineari?

$$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{b}\,, \tag{10}$$

$$f(\mathbf{h}) = \mathbf{w}^\top \mathbf{h} + c\,. \tag{11}$$

Potremmo, ma ciò è equivalente ad un singolo modello lineare:

$$f(\mathbf{x}) = \left(\mathbf{w}^\top \mathbf{W}\right)\mathbf{x} + \left(\mathbf{w}^\top \mathbf{b} + c\right)\,. \tag{12}$$

Abbiamo bisogno di un modo per separare le due operazioni lineari.

Possiamo evitare il 'collasso' dei due modelli lineari alternandoli con una semplice funzione (elementwise) ϕ:

$$\mathbf{h} = \phi(\mathbf{W}\mathbf{x}), \tag{13}$$

$$f(\mathbf{h}) = \mathbf{w}^\top \mathbf{h}. \tag{14}$$

Questo è il prototipo di una rete neurale **completamente connessa** (fully-connected, **FC**), a volte nota come **multilayer perceptron** (**MLP**).

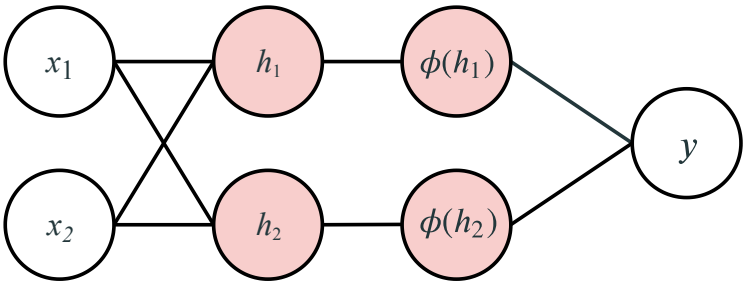


Figure 6: Visualizzazione di una semplice NN feedforward.

La ragione per cui chiamiamo questi modelli *reti neurali* è storica. Consideriamo la cosiddetta **step function**:

$$\mathbb{I}(x) = \begin{cases} 1 & \text{se } x \geq 0 \\ 0 & \text{altrimenti} \end{cases} \tag{15}$$

Il modello lineare $y = \mathbb{I}(\mathbf{w}^\top \mathbf{x} + b)$ fu uno dei primi modelli per il neurone biologico, nel quale i pesi di $\mathbf{w}$ prendono il ruolo delle sinapsi (a volte chiamato neurone di **McCulloch–Pitts**). Le reti neurali 'moderne' (a volte dette **artificiali**) si ottengono scegliendo un ϕ differenziabile per allenarle tramite discesa al gradiente.

Il contesto storico è rimasto anche in molti dei nomi che usiamo quando ci riferiamo alle reti neurali:

- Le due equazioni (13) e (14) vengono dette gli **strati** (**layer**) della rete (*hidden layer* e *output layer*).
- Gli elementi di $\mathbf{h}$ vengono detti le **attivazioni** del layer. Ogni elemento di $\mathbf{h}$ viene detto un **neurone**.
- La non-linearità ϕ viene detta **funzione di attivazione**.

Reti neurali feedforward

Addestramento e approssimazione universale

L'addestramento della rete può procedere in modo simile a un modello lineare. Ad esempio, dato un dataset $\{\mathbf{x}_i, y_i\}_{i=1}^n$ per la regressione, possiamo minimizzare il mean-squared error:

$$\mathbf{W}^*, \mathbf{w}^* = \arg\min \frac{1}{n} \sum_{i=1}^{n} \left(y_i - f(\mathbf{x}_i)\right)^2. \tag{16}$$

Per la classificazione, possiamo aggiungere una sigmoide e minimizzare la cross-entropy media. A differenza del caso lineare, questo problema di ottimizzazione è non convesso.

Un modello lineare può approssimare con precisione solo classi linearmente separabili. Cosa possiamo dire, invece, di un MLP? G. Cybenko è stato il primo a dimostrare (1989) che tali reti godono di una proprietà nota come **approssimazione universale**.

Informalmente, data una funzione $g(x)$ (sotto vincoli molto generici), possiamo trovare un MLP come in (13)-(14) tale per cui, per ogni $\varepsilon > 0$:

$$|f(\mathbf{x}) - g(\mathbf{x})| \leq \varepsilon \quad \forall \mathbf{x} \tag{17}$$

Cybenko lo ha provato per $\phi = \sigma$ (sigmoide), mentre autori successivi hanno generalizzato il risultato per ϕ di (quasi) ogni tipo.

Non dimostriamo il teorema di approssimazione universale, ma è utile avere una intuizione visiva di come strutturare tale dimostrazione.[1]

Al fine di visualizzare i risultati, ci concentriamo su funzioni con un singolo input ed un singolo output. Procediamo per gradi.

- Con un singolo neurone nello strato nascosto, possiamo approssimare qualsiasi *step function*.
- Con due neuroni, possiamo approssimare qualsiasi funzione costante su un intervallo (*bin function*).
- Con $2m$ neuroni possiamo costruire una funzione *stepwise constant* su m diversi intervalli.

[1] `http://neuralnetworksanddeeplearning.com/chap4.html`

Consideriamo un singolo neurone nello strato nascosto. In questo caso possiamo scrivere l'equazione della rete come:

$$f(x) = a\sigma(w(x - s)) \tag{18}$$

In questa formulazione, a controlla l'altezza della funzione, w lo slope, s il centro. In particolare, per w molto alti otteniamo funzioni che variano da 0 ad a molto rapidamente.

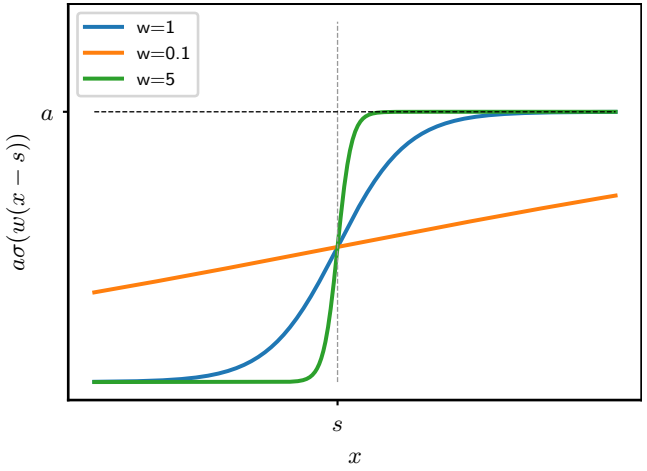


Figure 7: Una rete con un singolo neurone nello strato nascosto, visualizzata con a e s fissi, ma variando w.

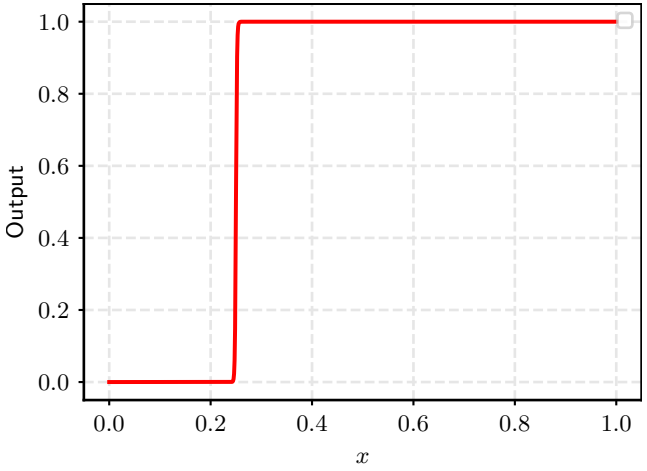


Figure 8: Fissando w ad un valore molto alto (es., 10^4) possiamo approssimare una step function con due parametri liberi (a ed s).

Aggiungendo un secondo neurone con ampiezza di segno opposto, possiamo approssimare una funzione costante su un intervallo. Definendo con Δ l'ampiezza dell'intervallo:

$$f(x) = a\sigma\left(w(x - s - \frac{\Delta}{2})\right) - a\sigma\left(w(x - s + \frac{\Delta}{2})\right) \tag{19}$$

Per semplicità nel seguito, possiamo denotare questo come $f(x; a, s, \Delta)$.

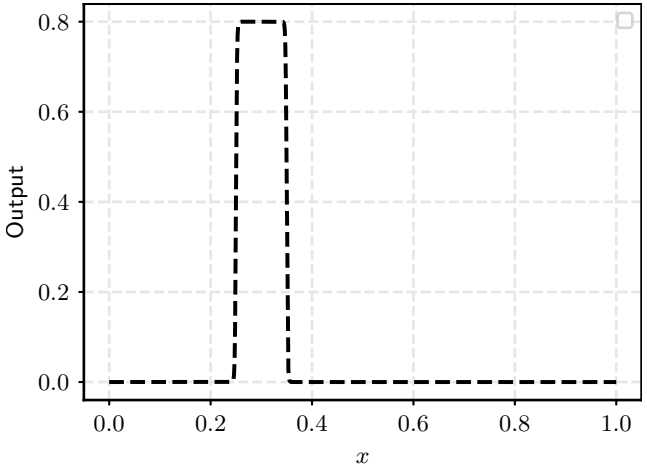


Figure 9: Con due neuroni, possiamo approssimare una funzione costante su un intervallo.

Per continuare, notiamo che la funzione è altamente localizzata, nel senso che $f_{a,s,\Delta}(x) = 0$ fuori dall'intervallo $\left[s - \frac{\Delta}{2}, s + \frac{\Delta}{2}\right]$.

Possiamo ottenere una funzione costante su *due* intervalli diversi con 4 neuroni:

$$f(x) = f(x; a_1, s_2, \Delta_1) + f(x; a_2, s_2, \Delta_2) \tag{20}$$

L'altezza del bin può anche essere negativa, come visualizzato nella prossima slide.

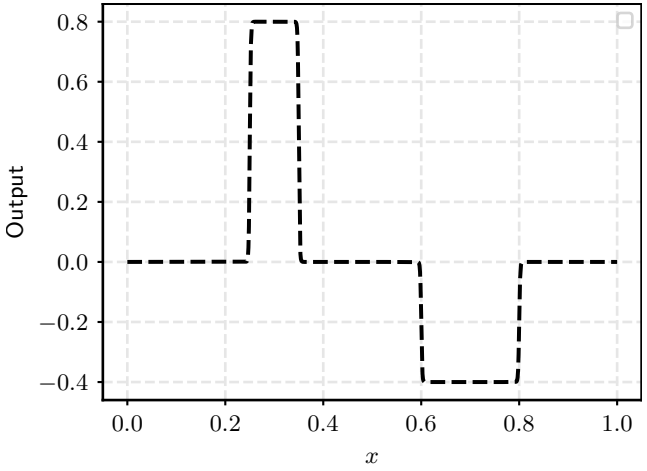


Figure 10: Con quattro neuroni possiamo approssimare due intervalli separati.

Consideriamo ora una funzione generica $g(x)$, che vogliamo approssimare sull'intervallo $[0, 1]$ (per semplicità). Dividiamo l'asse x in m bins equamente spaziati: l'i-esimo bin copre l'intervallo $B_i = \left[\frac{i}{m} - \frac{\Delta}{2}, \frac{i}{m} + \frac{\Delta}{2}\right]$, dove Δ è la distanza tra due bin. Definiamo il valore medio di g in ogni intervallo:

$$g_i = \frac{1}{\Delta} \int_{x \in B_i} g(x) dx \tag{21}$$

Definiamo una rete con $2m$ neuroni, ciascuno centrato in un bin:

$$f(x) = \sum_{i=1}^{m} f\left(x; g_i, \frac{i}{m}, \Delta\right) \tag{22}$$

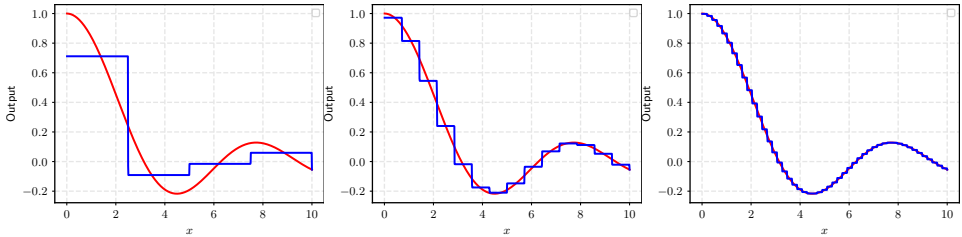


Figure 11: Approssimazione di $g(x) = \frac{\sin(x)}{x}$ con $m = 5$, $m = 15$, $m = 50$.

Quanto visto sopra si può estendere anche al caso multidimensionale o ad altre classi di funzioni di attivazione. Nella pratica, però, questo tipo di costruzione non è fattibile a causa del curse of dimensionality. In effetti, è facile costruire funzioni $g(x)$ dove l'approssimazione richiede un numero esponenziale di neuroni nella dimensionalità dell'input.

In pratica, la scelta del numero di neuroni (o del numero di layer, come vedremo) diventa un iper-parametro da ottimizzare, ed i pesi vengono selezionati tramite discesa al gradiente e non manualmente.

Reti neurali feedforward

Altre considerazioni

Poiché la NN può essere altamente non convessa, il suo problema di ottimizzazione ha molteplici minimi locali e/o punti di sella.

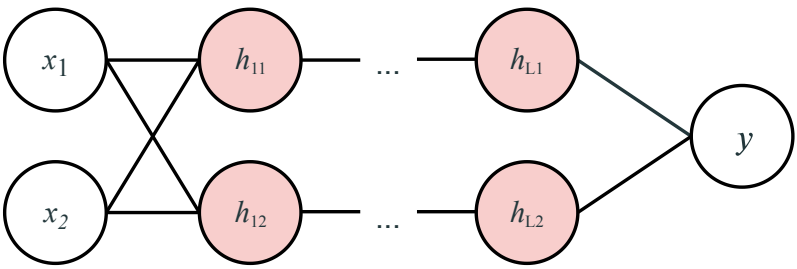
Infatti, addestrare una rete neurale *in maniera globalmente ottima* è **NP-hard**, anche per architetture molto semplici, e fortemente dipendente da una buona inizializzazione.

Trovare l'ottimo globale richiede di eseguire discesa al gradiente da *qualsiasi punto*: questo è simile a una ricerca esaustiva.

Blum, A. e Rivest, R.L., 1989. Training a 3-node neural network is NP-complete. In Advances in neural information processing systems (pp. 494-501).

Nulla ci impedisce di aggiungere strati 'nascosti' (intermedi) *aggiuntivi*:

$$f(\mathbf{x}) = \mathbf{w}^T \cdot \overbrace{\phi\big(\mathbf{Z} \cdot \underbrace{\phi\left(\mathbf{W} \cdot \mathbf{x}\right)}_{\mathbf{h}^1} + \mathbf{c}\big)}^{\mathbf{h}^2} \tag{23}$$



A differenza di un modello lineare, una NN presenta diverse scelte di progettazione su cui abbiamo libertà:

- La non linearità (a volte chiamata **funzione di attivazione**);
- La dimensionalità degli strati nascosti;
- Il numero di strati nascosti.

Chiamiamo questi **iper-parametri** per distinguerli dai parametri (pesi) da addestrare tramite GD.

Scegliere il set corretto di iper-parametri è chiamato problema di **selezione del modello** / **ottimizzazione degli iper-parametri**.

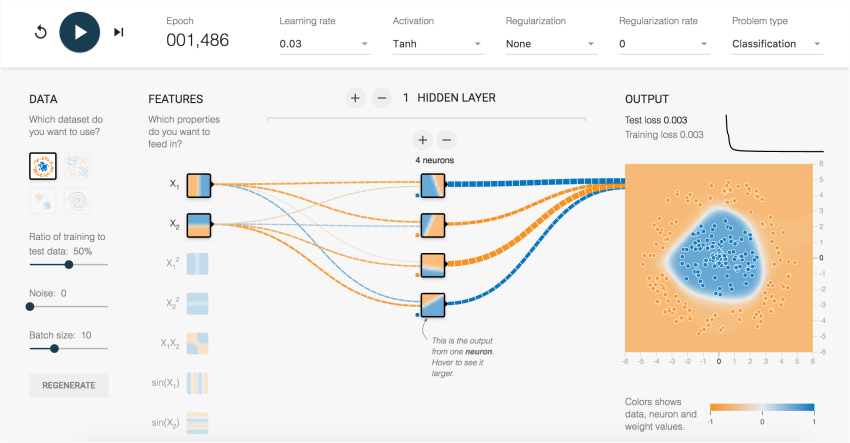


Figure 12: https://playground.tensorflow.org/.

Oltre alla sigmoide, quali tipi di non-linearità possiamo usare?

1. Tangente iperbolica (**tanh**) $\tanh(s) = 2\sigma(s) - 1$, che è solo una sigmoide scalata in $[-1, +1]$.
2. I polinomi $\phi(s) = s^p$ sono una pessima idea a meno che non si presti molta attenzione a causa di instabilità numeriche.
3. La cosiddetta **rectified linear unit** (ReLU) $\mathrm{ReLU}(s) = \max(0, s)$ è una buona scelta. Se consideriamo il bias dello strato precedente parte della funzione di attivazione, $\mathrm{ReLU}(s) = \max(0, s + b)$ è una funzione soglia con una soglia addestrabile.

Addestramento delle reti

Ottimizzazione stocastica

Consideriamo i passaggi necessari per una singola iterazione di GD:

1. Calcolare l'output della NN $f(\mathbf{x})$ su *tutti* gli esempi;
2. Calcolare il gradiente della funzione di costo rispetto ai pesi.

Entrambe queste operazioni scalano *linearmente* con il numero di esempi, il che è impraticabile quando abbiamo 10^5 o anche più esempi.

Nella pratica, per addestrare le NN usiamo versioni **stocastiche** di GD, che approssimano il gradiente reale da quantità minori di dati.

L'osservazione chiave è che il problema di addestramento nelle NN è un valore atteso rispetto a tutti i dati ($\boldsymbol{\theta}$ è l'insieme dei pesi):

$$J(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - f(\mathbf{x}_i)\right)^2. \tag{24}$$

Per limitare la complessità, possiamo usare solo un **mini-batch** $\mathcal{B}$ di M esempi dall'intero dataset:

$$J(\boldsymbol{\theta}) \approx \widetilde{J}(\boldsymbol{\theta}) = \frac{1}{M}\sum_{i\in\mathcal{B}}\left(y_i - f(\mathbf{x}_i)\right)^2. \tag{25}$$

Un GD in cui usiamo un mini-batch ad ogni iterazione è chiamato **discesa del gradiente stocastico** (stochastic gradient descent, SGD).

La complessità computazionale di un'iterazione di SGD è fissa rispetto a M (la **dimensione del batch**) e non dipende dalla dimensione del dataset.

Poiché abbiamo assunto che i campioni siano i.i.d., possiamo dimostrare che SGD converge a un punto stazionario, sebbene *solo in media*.

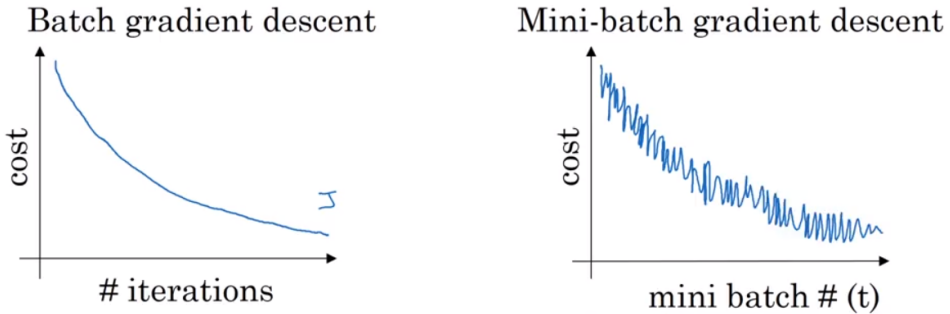


Figure 13: SGD convergerà in media a un punto stazionario, sebbene in modo apparentemente rumoroso (Fonte: EngMRK).

Invece di campionare casualmente i batch dal dataset di addestramento ad ogni iterazione, generalmente applichiamo la seguente procedura:

1. Mescoliamo (shuffling) l'intero dataset;
2. Suddividiamo il dataset in blocchi di M elementi e li elaboriamo sequenzialmente, batch per batch;
3. Dopo l'ultimo batch, torniamo al punto (1) e iteriamo.

Un passaggio completo sul dataset è chiamato una **epoca**. Questo è efficiente perché la maggior parte del tempo gestiamo dati memorizzati *contiguamente* in memoria. Nella pratica, anche l'operazione di mescolamento può essere approssimata.

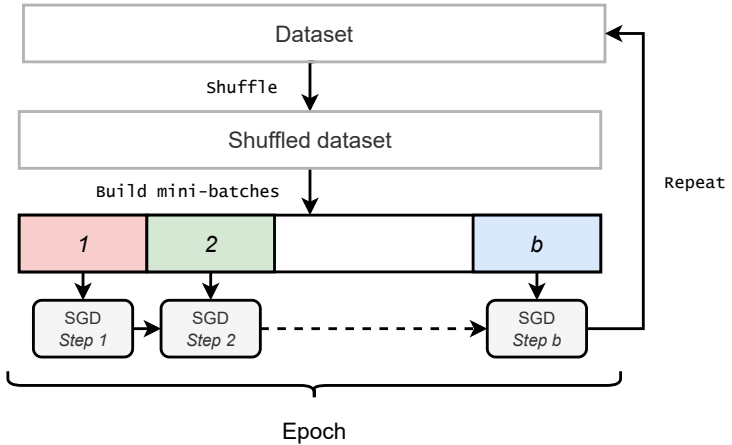


Figure 14: Suddividere il processo di ottimizzazione in epoche è utile anche per definire metriche e criteri di arresto.

Eseguire SGD richiede di selezionare la dimensione del mini-batch, che è un ulteriore iper-parametro:

- I mini-batch più piccoli sono più veloci, ma la rete potrebbe richiedere più iterazioni per convergere perché il gradiente è più rumoroso.
- I mini-batch più grandi forniscono una stima più affidabile del gradiente, ma sono più lenti.

In generale, è tipico scegliere dimensioni potenza di due (32, 64, 128, ...) a seconda della configurazione hardware e della memoria totale disponibile.

Addestramento delle reti

Automatic differentiation

L'ultimo aspetto da considerare è il calcolo dei gradienti della rete neurale. Questo porta a due sotto-problemi di interesse:

- Nonostante sia sempre possibile implementare i gradienti manualmente, ci piacerebbe avere tecniche *automatiche* per calcolarli (**automatic differentiation**).
- Tra tutti i possibili modi di calcolare il gradiente, ne vorremmo uno *efficiente*, ovvero che scali linearmente nel numero di parametri e nel numero di input alla rete stessa.

Baydin, A.G., Pearlmutter, B.A., Radul, A.A. and Siskind, J.M., 2018. **Automatic differentiation in machine learning: a survey**. *Journal of Machine Learning Research*, 18.

Per rispondere ai due quesiti, andiamo ad analizzare un caso concreto: una rete neurale con un solo strato nascosto, allenata per un problema di classificazione. In questo caso il problema di ottimizzazione si può scrivere (ignorando i bias) come:

$$L = \sum_{i=1}^{n} \mathrm{CE}\left(\mathbf{y}_i, \mathrm{softmax}\left(\mathbf{W}_2 \phi\left(\mathbf{W}_1 \mathbf{x}_i\right)\right)\right) \tag{26}$$

dove CE è la cross-entropy, $\mathbf{y}_i$ è one-hot encoded, e $\mathbf{W}_1$ e $\mathbf{W}_2$ sono i parametri allenabili.

Siamo interessati a calcolare $\nabla_{\mathbf{W}_1} L$ e $\nabla_{\mathbf{W}_2} L$ in maniera efficiente. Ricordiamo che se $\mathbf{h} = f(\mathbf{x})$ e $\mathbf{y} = g(\mathbf{h})$:

$$\partial_{\mathbf{x}} \mathbf{y} = [\partial_{\mathbf{h}} g(\mathbf{h})] \, \partial_{\mathbf{x}} f(\mathbf{x}) \tag{27}$$

Questa è la chain rule, estesa al caso dei Jacobiani: componendo due funzioni, lo Jacobiano risultante è il prodotto tra matrici dei due Jacobiani individuali. Per evitare troppi indici assumiamo che $n = 1$, ma il ragionamento si estende facilmente.

Andiamo quindi a decomporre la nostra rete neurale in operazioni di cui conosciamo (o possiamo calcolare) i singoli Jacobiani:

$$\begin{aligned}
\mathbf{h}_1 &= \mathbf{W}_1\mathbf{x} \\
\mathbf{h}_2 &= \phi(\mathbf{h}_1) \\
\mathbf{h}_3 &= \mathbf{W}_2\mathbf{h}_2 \\
L &= \mathrm{CE}\left(\mathbf{y}, \mathrm{softmax}(\mathbf{h}_3)\right)
\end{aligned}$$

facendo attenzione a distinguere matrici (lettere maiuscole in grassetto), vettori (lettere minuscole in grassetto), e scalari.

Iniziamo da:

$$\mathbf{h} = \mathbf{W}\mathbf{x} \tag{28}$$

Rispetto ad $\mathbf{x}$, abbiamo trivialmente $\partial_{\mathbf{x}}\mathbf{h} = \mathbf{W}$. Rispetto a $\mathbf{W}$, lo Jacobiano ha tre dimensioni, ed abbiamo:

$$\frac{\partial h_i}{\partial W_{ij}} = x_j \tag{29}$$

Questa operazione va effettuata per ogni elemento del dataset (o del mini-batch) per ottenere lo Jacobiano completo.

Consideriamo ora:

$$\mathbf{h} = \phi(\mathbf{x}) \tag{30}$$

In questo caso, lo Jacobiano è una matrice diagonale, e:

$$\frac{\partial h_i}{\partial x_i} = \phi'(x_i) \tag{31}$$

dove ϕ' è la derivata di ϕ. Anche in questo caso, nel caso l'input sia un mini-batch dobbiamo aggiungere una terza dimensione concatenando tutte le singole matrici diagonali.

Per concludere:

$$L = \mathrm{CE}(\mathbf{y}, \mathrm{softmax}(\mathbf{x})) \quad \rightarrow \quad \frac{\partial L}{\partial \mathbf{x}} = (\mathbf{y} - \mathbf{x}) \tag{32}$$

Possiamo quindi scrivere le equazioni dei due gradienti (ignorate i colori per ora):

$$\nabla_{\mathbf{W}_2} L = \left[\frac{\partial \mathbf{h}_3}{\partial \mathbf{W}_2}\right] (\mathbf{y} - \mathbf{h}_3) \tag{33}$$

$$\nabla_{\mathbf{W}_1} L = \left[\frac{\partial \mathbf{h}_1}{\partial \mathbf{W}_1}\right] \phi'(\mathbf{h}_1)\mathbf{W}_2 (\mathbf{y} - \mathbf{h}_3) \tag{34}$$

Le moltiplicazioni in (33) e (34) possono essere eseguite in qualsiasi ordine, però:

- Se le eseguiamo da sinistra a destra (**forward-mode**) ogni operazione richiede un prodotto matrice-matrice.
- Se le eseguiamo da destra a sinistra (**reverse-mode**) ogni operazione richiede un prodotto vettore-matrice.

Questo è sempre vero: eseguire le operazioni all'inverso è sempre un ordine di grandezza più efficiente che eseguirle nell'ordine delle operazioni originali. Questo algoritmo viene detto **reverse-mode automatic differentiation** o **back-propagation**.

Il termine rosso in (33) è ripetuto in (34). Se aggiungiamo un altro layer (provate), il termine verde sarebbe ripetuto nel nuovo gradiente. Questo permette una implementazione molto efficiente (sempre nel caso $n = 1$):

1. Si inizializza un vettore $\mathbf{c} = (\mathbf{y} - \hat{\mathbf{y}})$.
2. Per ogni operazione della rete neurale (all'incontrario):
   - 2.1 Se l'operazione ha parametri allenabili, si calcola il loro gradiente moltiplicando $\mathbf{c}$ per lo Jacobiano rispetto ai parametri.
   - 2.2 Si aggiorna $\mathbf{c}$ moltiplicandolo per lo Jacobiano rispetto all'input.

Una implementazione efficiente della back-propagation è il cuore di tutte le librerie di reti neurali, da TensorFlow a PyTorch e JAX, anche se il modo in cui viene implementata può variare.

Lo svantaggio della back-propagation è che richiede moltissima memoria, in quanto è necessario salvare in memoria tutti i valori intermedi della rete neurale durante il suo calcolo ($\mathbf{h}_1, \mathbf{h}_2, \ldots$). Di solito, il calcolo della loss viene detto **forward pass**, mentre il calcolo del gradiente **backward pass**.

Baydin, A.G., Pearlmutter, B.A., Radul, A.A. and Siskind, J.M., 2018. **Automatic differentiation in machine learning: a survey**. *Journal of Machine Learning Research*, 18.

Addestramento delle reti

Scelta di una funzione di attivazione

Generalizzando il ragionamento di prima, ogni volta che inseriamo una funzione di attivazione, nella back-propagation moltiplichiamo per la derivata della funzione.

Se abbiamo molti layer, eseguiamo numerose moltiplicazioni di questo tipo. Di conseguenza:

- Se $\phi'(\cdot) < 1$ sempre, il gradiente tenderà a zero in modo esponenziale nel numero di strati (**vanishing gradients**).
- Se $\phi'(\cdot) > 1$ sempre, il gradiente esploderà in modo esponenziale nel numero di strati (**exploding gradients**).

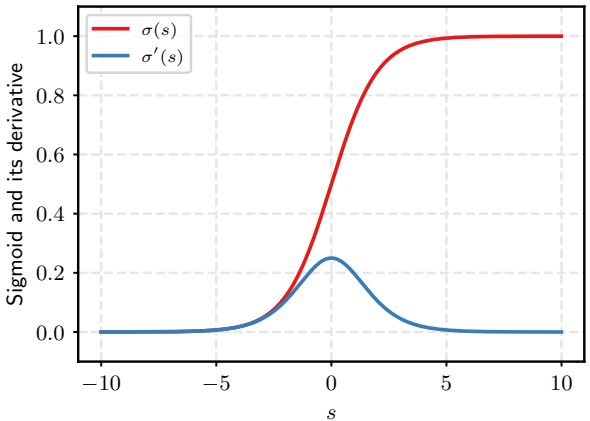


Figure 15: La funzione sigmoide $\sigma(\cdot)$ è una scelta errata come funzione di attivazione (per reti con molti strati), perché la sua derivata è bounded in $[0, 0.25]$.

Una scelta molto comune per le reti con molti strati è la **rectified linear unit** (ReLU), definita come:

$$\phi(s) = \max(0, s) \ . \tag{35}$$

La sua derivata è 1 (quando $s > 0$), altrimenti è 0. La ReLU è una buona scelta predefinita nella maggior parte delle applicazioni.

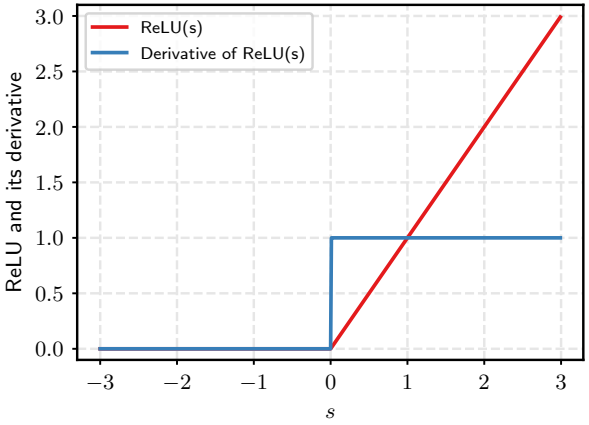


Figure 16: Un grafico della ReLU e della sua derivata.

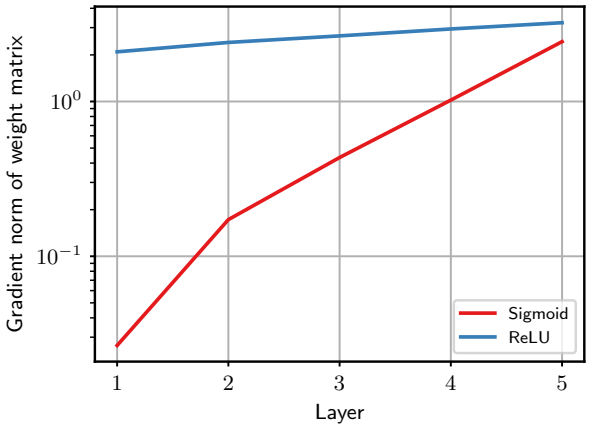


Figure 17: Consideriamo una NN con 5 strati nascosti. I pesi sono campionati dalla distribuzione uniforme su $[-0.5, 0.5]$. Mostriamo la norma di un tipico gradiente (cross-entropy) con funzioni di attivazione sigmoide e ReLU a confronto.