Università degli Studi di Trieste

Dipartimento di Scienze Economiche, Aziendali, Matematiche e Statistiche "Bruno De Finetti"

Corso di Laurea in Scienze Statistiche e Attuariali

Tesi Di Laurea Magistrale

Reti neurali per la stima della riserva di indennizzo di sinistri RC Auto al momento della loro denuncia

Laureando
Fabio Mardero

Relatore
Prof.ssa Patrizia Gigante

Correlatore
Prof. Renato Pelessoni

Anno Accademico 2018-2019

Capitolo 4

Reti Neurali

Le reti neurali sono un particolare modello di machine learning in grado apprendere in maniera supervisionata, non supervisonata, e per rinforzo. Le loro capacità di elaborazione si distinguono molto da tutti gli altri algoritmi di ML, tanto che nella loro accezione "più complicata" hanno guadagnato una loro precisa categorizzazione: il deep learning. Gli algoritmi di deep learning sono infatti in grado di elaborare tutti i tipi di dati compresi quelli testuali e grafici, singolarmente o anche in formato sequenziale (ad esempio video o audio).

Le prime reti neurali sono teorizzate già a metà degli anni Novanta, ma solo intorno al 2010 hanno ottenuto un enorme successo e interesse accademico e sopratutto industriale. Il motivo è legato alla loro struttura: si tratta di modelli dotati di un grandissimo numero di parametri e per questo sono computazionalmente molto onerosi. Come discusso nella Sottosezione 3.4.2, dagli anni 2000 l'architettura e le capacità dei calcolatori sono enormemente cresciute, tanto da permettere l'applicazione di modelli di machine learning via via sempre più complicati. Nvidia, uno dei maggiori produttori hardware mondiali, nel 2009 è stata coinvolta in quello che è noto come il big bang del deep learning: con il titolo «Large-scale Deep Unsupervised Learning using graphics processors», la pubblicazione (Raina, Madhavanand e A. Y. Ng, 2009) di alcuni dei più illustri data scientists contemporanei, ha dimostrato che l'applicazione delle reti neurali su GPUs anziché CPUs velocizzava il processo di training di ben 70 volte (esperimenti che richiedevano settimane di elaborazione ora impiegavano soltanto qualche ora). Con i progressi fatti in termini di software e hardware, nel 2009 i tempi erano maturi per l'applicazione del deep learning a problemi reali. Attualmente tutte le più grandi compagnie digitali sviluppano il loro modello di business proprio attorno alle reti neurali.

Per la stesura di questo capitolo ci si è basati sui testi *Machine Learning: a probabilistic perspective* (Murphy, 2012) e *Deep learning* (Goodfellow, Bengio e Courville, 2016).

4.1 Il percettrone

Nel 1957 Frank Rosenblatt (Rosenblatt, 1958) ideò un modello matematico, che chiamò *percettrone*, in grado di schematizzare le funzionalità di un neurone con l'intento di spiegare matematicamente il funzionamento del cervello umano. Un neurone (Figura 4.1) è una cellula in grado di essere

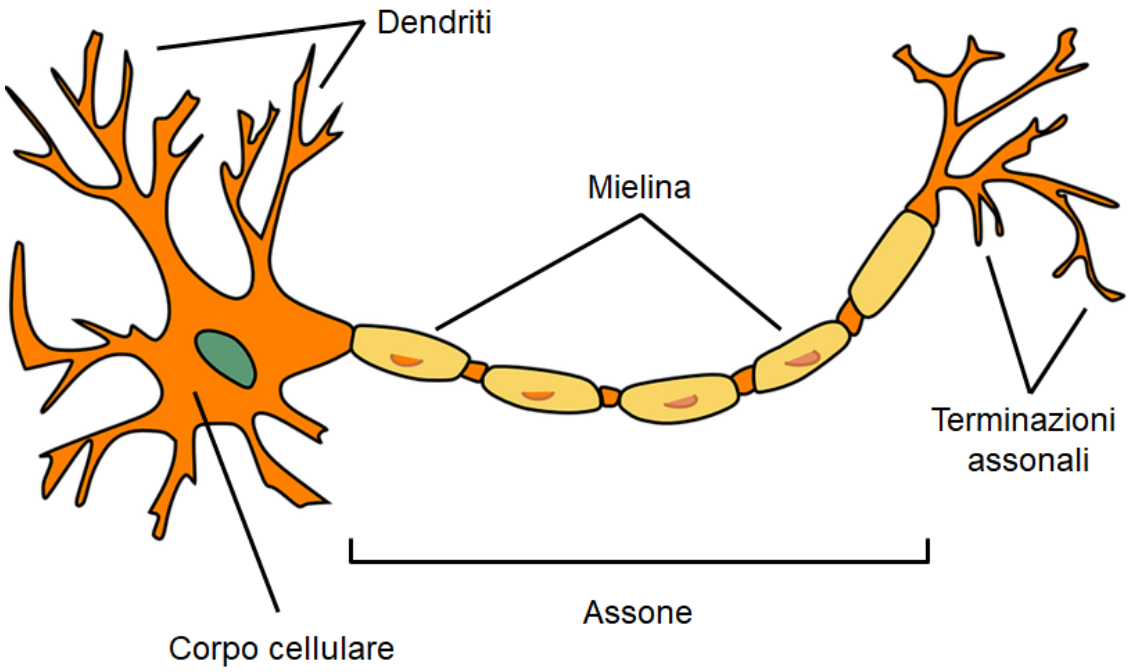


Figura 4.1: Visualizzazione semplificata di un neurone.

elettricamente eccitata al fine di ricevere, elaborare, e trasmettere informazioni sotto forma di segnali chimici ed elettrici. Ogni neurone, oltre al corpo centrale, comprende un assone e i dendriti. L'assone è un lungo canale con il quale il segnale elaborato è propagato alle cellule neuronali successive. I dendriti di un neurone invece ricevono segnali da molti assoni di altre cellule e li trasmettono al corpo centrale. Il collegamento assoni-dendriti è realizzato per mezzo della struttura chiamata sinapsi in cui il segnale elettrico proveniente dall'assone è trasformato in chimico e ricevuto dai dendriti che lo ritrasformato in elettrico. L'informazione trasmessa lungo l'assone è infatti un impulso elettrico il cui potenziale è chiamato potenziale d'azione e sarà indicato con $\hat{y}$. Quando un neurone è a riposo, cioè non è stimolato da segnali esterni, invece, tra l'interno e l'esterno della cellula è presente un potenziale b, detto di membrana, pari a circa -70 mV (milliVolt). Considerando un singolo neurone $\mathcal{N}$, i segnali che i suoi dendriti ricevono non hanno la stessa intensità con cui sono stati emessi dalle rispettive cellule (l'impulso "originale" dell'i-esimo neurone è indicato con x_i). Il motivo è dato dal fatto che il segnale propagandosi cambia intensità in base allo spessore dello strato di mielina che avvolge l'assone: più è spesso, minore sarà la dispersione $w > 0$ del segnale. Potenziali x_i se positivi si dicono eccitatori, altrimenti inibitori. Il segnale $v(x_1, \ldots, x_n)$, detto campo indotto locale, che il corpo cellulare di $\mathcal{N}$ riceve sarà quindi la somma degli n potenziali in ingresso,

$$v(x_1, \ldots, x_n) = \sum_{i=1}^{n} w_i x_i = \boldsymbol{w}^T \boldsymbol{x}.$$

Se il potenziale v è più grande in modulo di quello di membrana allora $\mathcal{N}$ libera a sua volta un segnale elettrico lungo l'assone. Al contrario, se la tensione che il corpo cellulare riceve è troppo

bassa, allora nessun segnale è propagato. Vi è inoltre indipendenza tra i segnali in input e quello in output. Il precedente meccanismo può essere esposto matematicamente considerando una funzione di attivazione h (*activation function*) che, dato $b < 0$, sia in grado di restituire un valore positivo se $v = \boldsymbol{w}^T\boldsymbol{x} > -b$, o nullo in caso contrario. Un esempio di activation function è la funzione gradino Θ di Heaviside:

$$h(u) = \Theta(u) = \begin{cases} 1 & u \geq 0 \\ 0 & u < 0 \end{cases} \tag{4.1}$$

Si può infatti assumere senza perdita di generalità che la tensione di emissione sia o unitaria o nulla (neurone "acceso" o "spento"), ottenendo quindi la seguente schematizzazione di $\mathcal{N}$.

$$\hat{y} = h(v + b) = \Theta(\boldsymbol{w}^T\boldsymbol{x} + b) \tag{4.2}$$

Nel corso degli anni sono state utilizzate diverse funzioni di attivazione h, come la sigmoide/logistica

$$h(u) = \sigma(u) = \frac{1}{1 + e^{-u}}, \tag{4.3}$$

la tangente iperbolica

$$h(u) = \tanh(u) \tag{4.4}$$

e la ReLU (*rectified linear unit*)

$$h(u) = ReLU(u) = \max(0, u) = \begin{cases} u & u \geq 0 \\ 0 & u < 0 \end{cases} \tag{4.5}$$

Nel seguito si considera il caso della activation function (4.1). Per come definito in (4.2), possedendo due stati di output[1], il percettrone è di fatto un modello di machine learning di classificazione binaria definito dalla famiglia di funzioni

$$\mathcal{F} = \left\{ f(\boldsymbol{x}; \boldsymbol{w}) = h(\boldsymbol{w}^T\boldsymbol{x});\ \boldsymbol{x}, \boldsymbol{w} \in \mathbb{R}^{n+1} \right\} \tag{4.6}$$

in cui si assume che $x_0 = 1$ e $w_0 = b$ (d'ora in poi sarà utilizzata questa notazione). Dato $\mathbb{R}^n$ l'insieme di tutte le possibili determinazioni degli input $(x_1, \ldots, x_n)$, un percettrone separa in due questo spazio n-dimensionale tramite un iperpiano $(n-1)$-dimensionale: in uno il percettrone rilascia il segnale, nell'altro rimane a riposo. Basta infatti considerare che il neurone artificiale cambia il suo output in base al fatto che $\boldsymbol{w}^T\boldsymbol{x}$ sia positivo o negativo. Il *decision boundary* (letteralmente "confine di decisione") che delimita le features per cui si ottiene $\hat{y} = 1$ da quelle con risultato nullo si ottengono, quindi, ponendo

$$s(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} = 0, \tag{4.7}$$

che è appunto l'equazione di un iperpiano $(n-1)$-dimensionale (la combinazione tra le variabili è affine). Nel caso in cui $n = 2$, ad esempio, presa x_2 come ordinata, si ricava che la retta di separazione ha equazione[2]

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}.$$

Il neurone artificiale è quindi in grado di risolvere solamente problemi linearmente separabili, cioè risolvibili con la definizione di un iperpiano di separazione in $\mathbb{R}^n$.

[1] Nel caso delle altre funzioni di attivazione aventi output continuo nell'intervallo $[0, 1]$, il percettrone restituisce la probabilità che l'output sia la classe "1", cioè di quanto "è sicuro" di dover rilasciare il segnale elettrico attraverso l'assone.

[2] Si assume che il modello associ una qualche significatività ad entrambe le covariate, cioè che $w_i \neq 0, i = 1, 2$.

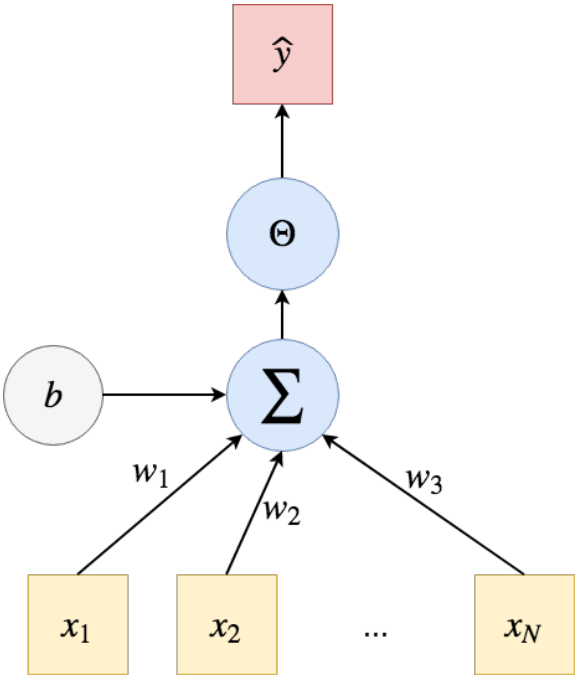


Figura 4.2: Computational graph di un percettrone.

Le operazioni svolte dal neurone artificale possono essere sintetizzate graficamente in quello che è noto come *computational graph* (vedi Figura 4.2). In questa forma di rappresentazione sono utilizzati nodi e linee: ogni nodo rappresenta una variabile o una creazione di essa attraverso un'operazione matematica, le linee indicano invece quali nodi interagiscono con gli altri. Il nodo Σ, ad esempio, rappresenta la valutazione di $s(\boldsymbol{x})$, quello Θ indica l'applicazione della funzione di attivazione. Il computational graph di f in Figura 4.2 ha la caratteristica di poter essere descritto come flusso ordinato di operazioni permettendo quindi di individuare input e output del modello.

Al di là dell'interesse nel rappresentare matematicamente il funzionamento dell'unità fondamentale del cervello umano, il percettrone è una particolare forma di regressione non lineare in quanto $\hat{y}$ non è lineare in $\boldsymbol{w}$. Ad esempio per $h = \sigma$ le funzioni in $\mathcal{F}$ possiedono la stessa struttura di quelle di una regressione logistica: ciò che manca sono le ipotesi probabilistiche che invece stanno alla base di quest'ultima. Il neurone artificale può quindi ritenersi una generalizzazione dei modelli GLM in cui, infatti, non sono specificate ipotesi sulla distribuzione della variabile target.

Il percettrone, infine, è allenato utilizzando metodi di gradient descent, in particolare sfruttando l'algoritmo di error backpropagation (3.12). Anche in natura, infatti, i neuroni adattano le loro caratteristiche allo scopo di ottimizzare il risultato finale. Si tratta dunque di determinare i punti

di annullamento delle seguenti equazioni.

$$\begin{aligned}\frac{\partial R_{emp}(\mathcal{D}, \boldsymbol{w})}{\partial w_i} &= \frac{1}{N}\sum_{k=1}^{N}\frac{\partial L\left(f(\boldsymbol{x}_k;\boldsymbol{w}), y_k\right)}{\partial w_i} \\ &= \frac{1}{N}\sum_{k=1}^{N}\frac{\partial L\left(t, y_k\right)}{\partial t}\,\frac{\partial f(\boldsymbol{x}_k;\boldsymbol{w})}{\partial w_i} \\ &= \frac{1}{N}\sum_{k=1}^{N}\frac{\partial L\left(t, y_k\right)}{\partial t}\,\frac{\partial h(s)}{\partial s}\,\frac{\partial s(\boldsymbol{x};\boldsymbol{w})}{\partial w_i} \\ &= \frac{1}{N}\sum_{k=1}^{N}\frac{\partial L\left(t, y_k\right)}{\partial t}\,\frac{\partial h(s)}{\partial s}\,x_i\ , \quad i = 0, \ldots, n \end{aligned} \tag{4.8}$$

Si noti che i parametri sono i pesi $w_0 = b, w_1, \ldots, w_n$.

4.2 Il percettrone multistrato

Dodici anni dopo la pubblicazione di Rosenblatt, nel 1969 Minsky e Papert (Minsky e Papert, 1969) hanno evidenziato le ampie limitazioni pratiche del percettrone. Questo tipo di modello, infatti, data la sua capacità di risolvere solamente problemi linearmente separabili, non è nemmeno in grado di rappresentare il funzionamento della funzione logica XOR (Tabella 4.1) definita come

$$XOR(x_1, x_2) = AND(NOT(AND(x_1, x_2)), OR(x_1, x_2)).$$

Come mostrato in Figura 4.3, il dominio di (x_1, x_2), dove $x_i \in \{0, 1\}$ per $i = 1, 2$, non può essere

$\boldsymbol{x_1}$	$\boldsymbol{x_2}$	$\boldsymbol{XOR(x_1, x_2)}$
0	0	0
0	1	1
1	0	1
1	1	0

Tabella 4.1: La funzione logica XOR.

separato da una retta in grado di dividere i punti che forniscono output $XOR(x_1, x_2) = 1$ (blu) da quelli che $XOR(x_1, x_2) = 0$ (rossi).

Le notevoli limitazioni all'uso del percettrone, soprattutto per problemi reali, placarono l'entusiasmo per il risultato ottenuto da Rosenblatt (Rosenblatt, 1958). La ricerca su questo tipo di modelli si arenò per molti anni finché nel 1986 Rumelhart, G. E. Hinton e Williams (Rumelhart, G. E. Hinton e Williams, 1986) presentarono la soluzione al problema XOR: il percettrone multistrato. Il *multi-layer perceptron* (MLP) non è altro che la combinazione sequenziale di più neuroni artificiali che vanno a comporre i cosiddetti *layer* (strati) del modello. Il percettrone diventa quindi l'unità fondamentale di modelli più complessi detti reti neurali (*neural networks*). Gli strati di una rete neurale possono essere distinti in *input layer* (il vettore degli input), *hidden layer* e *output layer* (il vettore degli output). Per ogni unità degli hidden layer (possono esserci più strati di questo

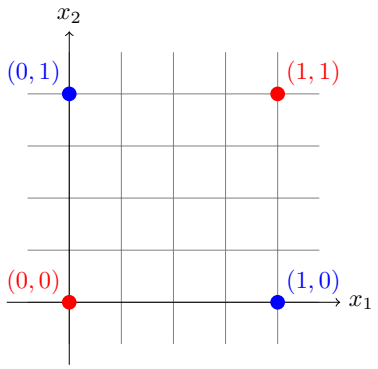


Figura 4.3: La funzione logica XOR non è linearmente separabile. Sono blu i punti per cui $XOR(x_1, x_2) = 1$, rossi gli altri.

tipo) o dell'output layer è calcolata la funzione di attivazione sulla somma pesata dei relativi input; sono appunto le stesse operazioni compiute da un neurone artificiale. Il percettrone infatti può essere visto come un caso particolare di MLP in quanto possiede un input layer ed un output layer composto da una singola unità che restituisce valori binari (Figura 4.2). I multi-layer perceptrons sono quindi una famiglia di funzioni non lineari composte, in cui la trasformazione non lineare (la funzione di attivazione) è applicata ripetutamente. Come trattato nella Sezione 4.1, essendo i percettroni rappresentabili con un computational graph dato da un flusso di operazioni dagli input verso l'output, le reti neurali allora si dicono *feed-forward* ("con flusso in avanti"). Nel caso in cui all'interno della rete neurale ogni unità sia connessa con tutti i percettroni del layer precedente, si parla di *fully-connected neural network* (Figura 4.4).

Come osservabile in Figura 4.5, il problema XOR può essere risolto usando un fully-connected multi-layer perceptron con un unico hidden layer composto da due unità (in tutto sono coinvolti tre percettroni).

$$\mathcal{F} = \left\{ f(\boldsymbol{x}; \boldsymbol{W}^{(1)}, \boldsymbol{w}^{(2)}) = h\left(\sum_{j=0}^{2} w_j^{(2)}\ h\left(\sum_{i=0}^{2} w_{ij}^{(1)} x_i \right) \right);\right.$$
$$\left. \boldsymbol{W^{(1)}} \in \mathbb{R}^3 \times \mathbb{R}^3,\ \boldsymbol{w^{(2)}} \in \mathbb{R}^{\mathbf{3}} \right\} \quad (4.9)$$

Con il risultato (4.7) è inoltre possibile visualizzare per ogni neurone del primo layer, $\mathcal{N}^{[L]}$ e $\mathcal{N}^{[R]}$, il relativo decision boundary, ovvero

$$\begin{aligned} \mathcal{N}^{[L]} &: \quad x_2 = x_1 - 0.01 \\ \mathcal{N}^{[R]} &: \quad x_2 = x_1 + 0.01 \end{aligned}$$

Per entrambi i confini è possibile ricavare quale regione dello spazio delle features sia assegnata all'output unitario e quale a quello nullo semplicemente ponendo $s(\boldsymbol{x}) \geq 0$ (Figura 4.6a e

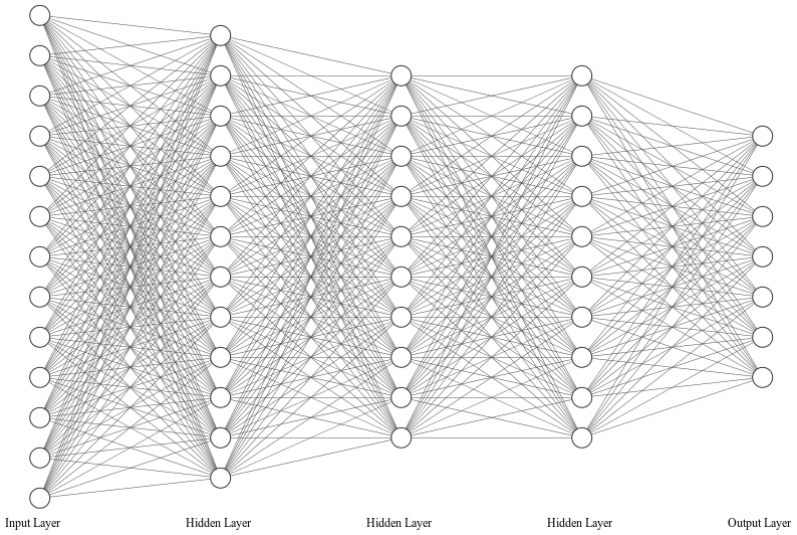


Figura 4.4: Esempio di multi-layer perceptron fully-connected con 13 input, 7 output e 3 hidden layer.

Figura 4.6b).

$$\mathcal{N}^{[L]} \quad : \quad \begin{cases} s^{[L]}(\boldsymbol{x}) \geq 0 \\ s^{[L]}(\boldsymbol{x}) = x_1 - x_2 - 0.01 \end{cases} \qquad \Longrightarrow \quad x_2 \leq x_1 - 0.01$$

$$\mathcal{N}^{[R]} \quad : \quad \begin{cases} s^{[R]}(\boldsymbol{x}) \geq 0 \\ s^{[R]}(\boldsymbol{x}) = -x_1 + x_2 - 0.01 \end{cases} \qquad \Longrightarrow \quad x_2 \geq x_1 + 0.01$$

Come appare evidente dalle considerazioni precedenti, nessuno dei due percettroni da solo è in grado di trovare una retta adeguata al compito; è grazie all'output layer che la rete neurale, aggregando in una qualche forma le due regioni (Figura 4.6c), è in grado di simulare quanto in Tabella 4.1. La regola per cui un neurone artificiale può risolvere solo problemi linearmente separabili vale anche per l'unità in output; ciò significa che è l'hidden layer a compiere una trasformazione sui dati tale da rendere, nello strato successivo, il problema risolvibile anche per un percettrone. Allo scopo di mostrare questa proprietà generale dei MLP (Alcorn, 2017), si assumerà che il dominio di X_1 e X_2 sia reale. Per verificare come opera l'hidden layer, basta infatti costruire una griglia di punti (x_1, x_2) e verificare quali sono le risposte fornite da $\mathcal{N}^{[L]}$ e $\mathcal{N}^{[R]}$. Trattandosi di soli due neuroni è possibile rappresentare i loro output $\left(x^{[L]}, x^{[R]}\right)$ in uno spazio bidimensionale. Come visibile in Figura 4.7, l'hidden layer collassa ogni datapoint nello spazio $\{(0,0),(0,1),(1,0)\}$. Il problema, con questa trasformazione delle features, risulta linearmente separabile; il decision boundary dell'unità di output ($w^{[L]} = w^{[R]} = 1$) è infatti in grado di dividere correttamente gli input simulando il funzionamento della funzione logica XOR.

$$\mathcal{N}^{[out]} \quad : \quad \begin{cases} s(x^{[L]}, x^{[R]}) \geq 0 \\ s(x^{[L]}, x^{[R]}) = x^{[L]} + x^{[R]} - 0.01 \end{cases} \qquad \Longrightarrow \quad x^{[R]} \geq -x^{[L]} + 0.01$$

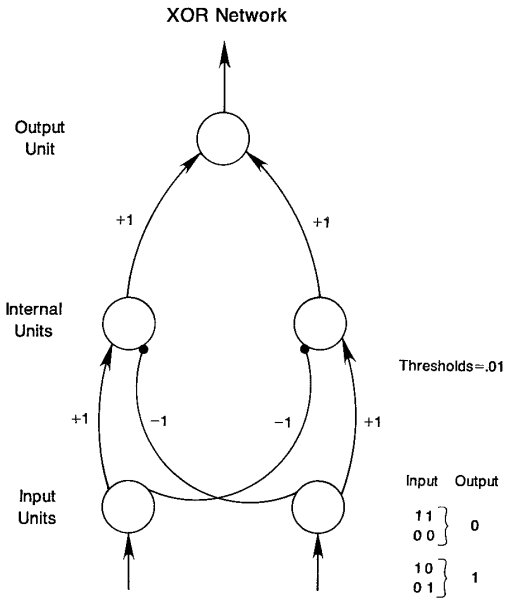
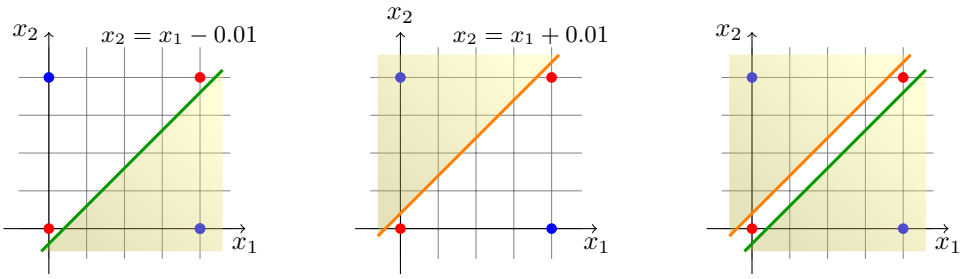


Figura 4.5: Grafico originale (Rumelhart, G. E. Hinton e Williams, 1986) del percettrone multistrato in grado di risolvere correttamente il problema XOR. Il valore a fianco di ogni connessione rappresenta il peso w applicato al relativo input. Per "treshold", rispetto a com'è definita (4.1), si intende invece il valore del bias $b = 0.01$ applicato alle "internal units" e alla "output unit".

Il problema XOR è stato trattato ripercorrendo la prima soluzione proposta (Figura 4.5), utilizzando (4.1) come funzione di attivazione. Si è parlato di decision boundaries proprio perché il compito del modello è di classificazione. La definizione dei confini è inoltre semplice in quanto la Θ di Heaviside assume valori binari. Utilizzando altre funzioni di attivazione che restituiscono invece valori nel continuo, è necessario individuare un valore di separazione (*treshold*) che distingue gli output di una classe da quelli di un'altra; ad esempio per la sigmoide si utilizza generalmente $\hat{y} = 0.5$. Qualora, invece, il problema sia di regressione è necessario utilizzare activation function con output continuo ed ha più senso parlare di curve di livello piuttosto che di treshold. I ragionamenti sopra esposti valgono anche in questo caso (intuitivamente basta pensare il problema di regressione come uno di classificazione con infiniti decision boundaries). La funzione di attivazione ReLU ne è un'eccezione in quanto è una particolare generalizzazione della Θ di Heaviside, non è limitata superiormente, è continua. Presenta, infatti, un boundary che separa gli output $h(\boldsymbol{w}^T\boldsymbol{x}) = s(\boldsymbol{x})$ da quelli $h(\boldsymbol{w}^T\boldsymbol{x}) = 0$. Questa particolarità permette quindi di visualizzare in modo analogo a Figura 4.6c il suo comportamento all'interno di uno strato della rete neurale.

4.2.1 Teorema di approssimazione universale

Come si è visto nel caso del problema XOR, il percettrone a singolo strato nascosto (4.9) è in grado di risolvere anche problemi non linearmente separabili semplicemente rimediando alle limitazioni del neurone artificiale attraverso l'applicazione di trasformazioni non lineari a trasformazioni non



(a) Perceptron $\mathcal{N}^{[L]}$. (b) Perceptron $\mathcal{N}^{[R]}$. (c) $\mathcal{N}^{[L]}$ e $\mathcal{N}^{[R]}$.

Figura 4.6: Rappresentazione del decision boundary dei due percettroni dell'hidden layer appartenenti al percettrone multistrato di Rumelhart applicato al problema XOR. Le regioni evidenziate rappresentano quelle per cui si ottiene dal neurone un output unitario.

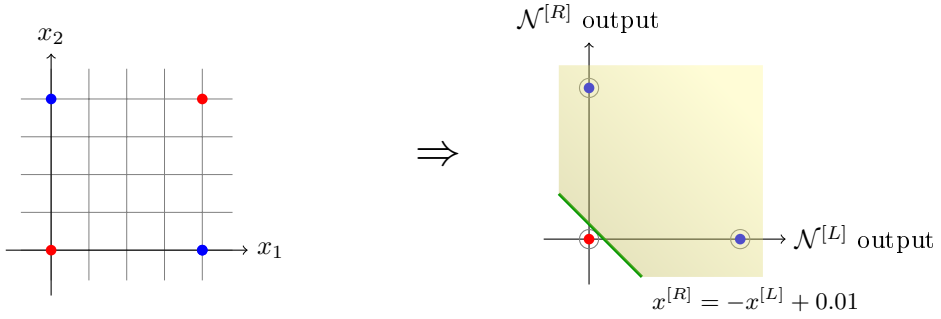


Figura 4.7: Le trasformazioni apportate dai percettroni dell'hidden layer alle covariate collassano ogni datapoint in uno dei tre punti in $\{(0,0),(0,1),(1,0)\}$. Il problema risulta così linearmente separabile, permettendo cioè di definire un decision boundary in grado di separare correttamente gli (x_1, x_2) tali che $XOR(x_1, x_2) = 1$ (area evidenziata) dagli altri.

lineari delle covariate. Questa proprietà è, nel caso di funzioni di attivazione continue, ancor più rafforzata dal teorema di approssimazione universale del percettrone multistrato (Hornik, 1991).

Teorema 4.2.1 (Teorema di approssimazione universale)**.** *Sia $I_n = [0,1]^n$, l'ipercubo unitario n-dimensionale, in grado di racchiudere l'insieme delle determinazioni $(x_1, \dots, x_n)$ delle features del modello. Sia $\mathcal{C}(I_n)$ l'insieme delle funzioni continue su I_n e h una funzione di attivazione non costante, limitata e continua. Si fissi quindi una funzione $\varphi \in \mathcal{C}(I_n)$ qualsiasi che si vuole approssimare. Allora,*

$$\forall \varepsilon > 0 \quad \exists \alpha \in \mathbb{N},\ \exists u_i \in \mathbb{R} \text{ e } \boldsymbol{w}_i \in \mathbb{R}^{n+1}, \qquad i = 1, \dots, \alpha,$$

posto

$$f(\boldsymbol{x}) = \sum_{i=1}^{\alpha} u_i\, h\left(\boldsymbol{w}_i^T \boldsymbol{x}\right), \tag{4.10}$$

si ha

$$|f(\boldsymbol{x}) - \varphi(\boldsymbol{x})| < \varepsilon, \quad \textit{per ogni } \boldsymbol{x} \in I_n. \tag{4.11}$$

Per esempio, per quanto detto nella Sezione 3.3, la funzione da approssimare potrebbe essere $\varphi(\boldsymbol{x}) = \mathbb{E}(Y|\boldsymbol{X} = \boldsymbol{x})$. Appare evidente che f, per com'è definita in (4.10), non è altro che un membro della famiglia dei percettroni multistrato ad un hidden layer e con α unità (caso generale di quanto è stato definito in (4.9)). Il teorema sostiene dunque che f è in grado di approssimare quanto bene si vuole una generica funzione continua φ. Si tratta di un risultato estremamente importante, ma di scarsa validità pratica. Infatti, da un punto di vista operativo determinare α, u_i e $\boldsymbol{w}_i$ $(i = 1, \ldots, \alpha)$ tali da soddisfare (4.11) può richiedere tempi estremamente lunghi. Per questo motivo, generalmente, per affrontare problemi complessi si preferisce complicare la struttura di f, detta *topologia* (rappresentazione del computational graph in cui i nodi sono o input o percettroni), al fine di velocizzare la ricerca dei parametri ottimali del modello. Ad esempio spesso si aumenta il numero di hidden layer presenti nella rete neurale; così facendo si va, tuttavia, incontro ad un trade-off tra la lentezza di stima dovuta all'introduzione dai nuovi parametri[3] e il tentativo di impiegare molto meno tempo della ricerca diretta di u_i e $\boldsymbol{w}_i$ su un percettrone monostrato. Si consideri inoltre che nella pratica si dispone di un dataset $\mathcal{D}$ limitato e che può non contenere tutte le determinazioni di $(\boldsymbol{X}, Y)$. Non è quindi possibile verificare se il modello con i parametri stimati soddisfa la relazione "esatta" (4.11).

4.3 Topologie avanzate

Le reti neurali, come si è in parte già visto, possono prevedere topologie estremamente complicate e ad un elevato numero di parametri (si superano facilmente le decine di migliaia di parametri). Per distinguere modelli così particolari, e dalle grandi potenzialità di previsione di Y, è stata dedicata loro una sotto-categoria del machine learning denominata *deep learning*. Esempi di reti neurali profonde sono i MLP a molti strati, le reti convoluzionali e quelle ricorrenti. Queste ultime due prevedono topologie specifiche in grado di risolvere particolari tipi di problemi che i normali percettroni a più strati trovano intrattabili o per cui sono altamente inefficienti. Le reti neurali convoluzionali (*convolutional neural network*, CNN) sono infatti in grado di elaborare input matriciali, indagando la relazione che lega il valore di una componente a quello assunto dalle circostanti; è il caso delle immagini, ad esempio, in cui il colore di un pixel dipende anche da quelli che lo circondano. Le reti ricorrenti (*recurrent neural network*, RNN), invece, elaborano serie storiche identificando un pattern tra gli input ricevuti in istanti diversi. Di seguito sarà approfondito quest'ultimo particolare tipo di rete in quanto le RNN saranno utilizzate e richiamate anche nei capitoli successivi.

4.3.1 Le reti ricorrenti

Se le reti convoluzionali forniscono un approccio alternativo all'elaborazione degli input matriciali, quello che manca ai multi-layer perceptrons è la capacità di gestire flussi di dati continui come quelli derivanti da serie temporali. Avendo a che fare con dati audio, testo o video, nasce quindi l'esigenza di costruire reti neurali capaci di comprendere, ai fini della previsione di una variabile target, la dipendenza di un input rispetto al precedente. Secondo queste considerazioni, quindi, in un flusso di dati l'ordine con cui le informazioni si presentano è importante. Si pensi ad esempio ad una frase

[3] L'aumento delle dimensioni della struttura della rete neurale implica un incremento dei parametri da stimare.

in cui ogni parola è ordinata o casualmente o rispettando le regole grammaticali: appare evidente che è estremamente più complicato comprenderne il significato nel primo caso. Il problema può essere formalmente espresso considerando lo stato del sistema[4] $\boldsymbol{s}$, da cui la variabile target dipende, che evolve nel tempo in funzione degli input $\boldsymbol{x}$. Indicando con $\boldsymbol{s}^{(t)}$ e $\boldsymbol{x}^{(t)}$ rispettivamente lo stato del sistema e l'informazione ricevuta al tempo t si può scrivere che

$$\begin{aligned} \boldsymbol{s}^{(t)} &= \varphi\left(\boldsymbol{s}^{(t-1)}, \boldsymbol{x}^{(t)}\right) \\ &= \Phi\left(\boldsymbol{s}^{(0)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, \ldots, \boldsymbol{x}^{(t-1)}, \boldsymbol{x}^{(t)}\right), \end{aligned} \tag{4.12}$$

dove φ è proprio la funzione che si vuole approssimare e Φ riassume la ripetuta applicazione di questa sugli input sequenziali. Si assume che lo stato iniziale $\boldsymbol{s}^{(0)}$ sia fissato e noto così che $\boldsymbol{s}^{(t)}$ dipenda solo da $\left(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\right)$:

$$\boldsymbol{s}^{(t)} = \Phi\left(\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}\right).$$

L'unico modo in cui un modello di machine learning può apprendere dai dati è attraverso i suoi parametri. L'idea è quindi di simulare (4.12) sostituendo a φ una funzione f, parte della rete neurale complessiva, che comprenda due matrici di parametri: $\boldsymbol{U}$ per $\boldsymbol{x}^{(t)}$, e $\boldsymbol{W}$ per $\boldsymbol{s}^{(t-1)}$. Così facendo quindi, il modello, cercando il minimo di R_{emp}, modifica anche le componenti di $\boldsymbol{U}$ e $\boldsymbol{W}$ allo scopo di spiegare il più accuratamente possibile la relazione tra la sequenza di input e lo stato del sistema/variabile target selezionata. Siccome l'informazione in uscita da f è solo l'approssimazione di $\boldsymbol{s}^{(t)}$ allora questa sarà indicata con $\boldsymbol{h}^{(t)}$, permettendo di scrivere

$$\boldsymbol{h}^{(t)} = f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}; \boldsymbol{U}, \boldsymbol{W}\right). \tag{4.13}$$

Il computational graph di (4.13) può essere visualizzato in Figura 4.8 sia in forma compatta, rappresentando le ricorsive applicazioni di f (a sinistra), che "srotolando" (*unfolding*) le operazioni svolte nel corso del tempo, consentendo di osservare ogni passaggio di stato (a destra). Si suppone che

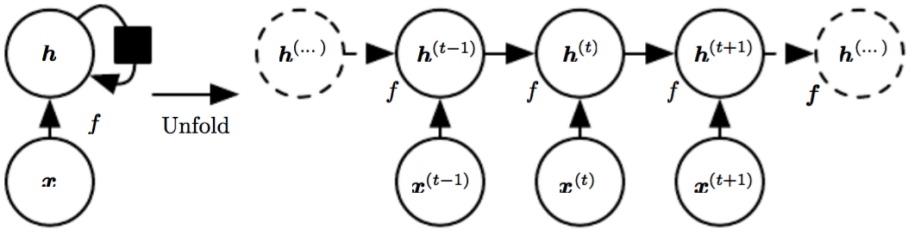


Figura 4.8: Computational graph delle operazioni iterative applicate ad input sequenziali (Goodfellow, Bengio e Courville, 2016).

la variabile target dipenda dallo stato del sistema in maniera funzionale tramite ψ, approssimata dalla rete neurale con g che coinvolge il vettore dei parametri $\boldsymbol{v}$.

$$\begin{aligned} y^{(t)} &= \psi\left(\boldsymbol{s}^{(t)}\right) \\ \hat{y}^{(t)} &= g\left(\boldsymbol{h}^{(t)}; \boldsymbol{v}\right) \end{aligned} \tag{4.14}$$

[4]Si suppone che lo stato del sistema sia esprimibile come entità vettoriale.

Generalmente, inoltre, la determinazione della variabile target osservata è riferita all'ultimo stato del sistema $s^{(T)}$ ottenuto dopo l'elaborazione dell'intero flusso di input disponibili. La rete neurale composizione di f e g si dice quindi rete neurale ricorrente (*recurrent neural network*, RNN), la cui famiglia $\mathcal{F}$ può essere scritta come[5]

$$\mathcal{F} = \left\{ F(\boldsymbol{x}; \boldsymbol{U}, \boldsymbol{W}, \boldsymbol{v}) = g \circ f\left(\boldsymbol{h}^{(T-1)}, \boldsymbol{x}^{(T)}\right);\ \boldsymbol{U},\ \boldsymbol{W},\ \boldsymbol{v} \right\} \tag{4.15}$$

che, nel caso molto semplice di una sequenza di input $\left(\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)$, diventa

$$\mathcal{F} = \left\{ F(\boldsymbol{x}; \boldsymbol{U}, \boldsymbol{W}, \boldsymbol{v}) = g\left(f\left(f\left(\boldsymbol{s}^{(0)}, \boldsymbol{x}^{(1)}; \boldsymbol{U}, \boldsymbol{W}\right), \boldsymbol{x}^{(2)}; \boldsymbol{U}, \boldsymbol{W}\right); \boldsymbol{v}\right) \right\}.$$

L'apprendimento della rete neurale ricorrente non richiede variazioni rispetto a quanto già esposto nella Sezione 4.2 e quindi è generalmente basato su algoritmi di stocastic gradient descent. Il procedimento di error backpropagation applicato ad un modello in grado di gestire sequenze in ingresso è, in particolare, detto *backpropagation through time* (BPTT). La BPTT è calcolata applicando la consueta regola di derivazione composta prestando, però, molta attenzione al fatto che $\boldsymbol{h}^{(T)} = f\left(\boldsymbol{h}^{(T-1)}, \boldsymbol{x}^{(T)}; \boldsymbol{U}, \boldsymbol{W}\right)$ dipende da $\boldsymbol{U}$ e $\boldsymbol{W}$ che intervengono sia nella funzione f che nel termine $\boldsymbol{h}^{(T-1)}$:

$$\begin{aligned}
\frac{\partial F(\boldsymbol{x}; \boldsymbol{U}, \boldsymbol{W}, \boldsymbol{v})}{\partial u} &= \frac{\partial g}{\partial \boldsymbol{h}^{(T)}} \frac{\partial \boldsymbol{h}^{(T)}}{\partial u}, & u &= \boldsymbol{U}_{ij} \\
\frac{\partial F(\boldsymbol{x}; \boldsymbol{U}, \boldsymbol{W}, \boldsymbol{v})}{\partial w} &= \frac{\partial g}{\partial \boldsymbol{h}^{(T)}} \frac{\partial \boldsymbol{h}^{(T)}}{\partial w}, & w &= \boldsymbol{W}_{ij} \\
\frac{\partial F(\boldsymbol{x}; \boldsymbol{U}, \boldsymbol{W}, \boldsymbol{v})}{\partial v} &= \frac{\partial g}{\partial v}, & v &= \boldsymbol{v}_i
\end{aligned} \tag{4.16}$$

Il calcolo di $\dfrac{\partial \boldsymbol{h}^{(T)}}{\partial u}$, ad esempio, può quindi essere completato solamente conoscendo la forma funzionale che lega lo stato in $(T-1)$ e i parametri. Durante l'allenamento, il numero delle componenti con cui rappresentare $\boldsymbol{h}^{(t)}$ è arbitrario e in quanto tale è un ulteriore iperparametro del modello. Maggiori sono le componenti del vettore di stato, maggiore è l'informazione trasportata dalla rete nel corso del tempo, ma di conseguenza aumenta anche il numero di parametri da stimare. Occorre quindi bilanciare le dimensioni di $\boldsymbol{h}^{(t)}$ con il costo computazionale richiesto per allenare $\mathcal{F}$.

Il modello più semplice di rete neurale ricorrente consiste nell'interpretare f e g come hidden layer[6]

$$\begin{aligned}
f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right) &= \sigma\left(\boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right) \\
g\left(\boldsymbol{h}^{(t)}; \boldsymbol{v}\right) &= \sigma\left(\boldsymbol{v}\, f\left(\boldsymbol{h}^{(t-1)}, \boldsymbol{x}^{(t)}\right) + \boldsymbol{b}_g\right)
\end{aligned} \tag{4.17}$$

così che

$$\begin{aligned}
\boldsymbol{h}^{(t)} &= \boldsymbol{\sigma}\left(\boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right) \\
\hat{y}^{(T)} &= \sigma\left(\boldsymbol{v}\, \boldsymbol{h}^{(T)} + \boldsymbol{b}_g\right).
\end{aligned}$$

[5] Si utilizza, in questo caso, la notazione F per indicare un membro della famiglia di $\mathcal{F}$ al fine di distinguerlo dalla funzione f coinvolta in (4.13).

[6] Per semplicità di notazione, in queste pagine, la funzione di attivazione applicata ad ogni componente del vettore in input è indicata con $\boldsymbol{\sigma}$.

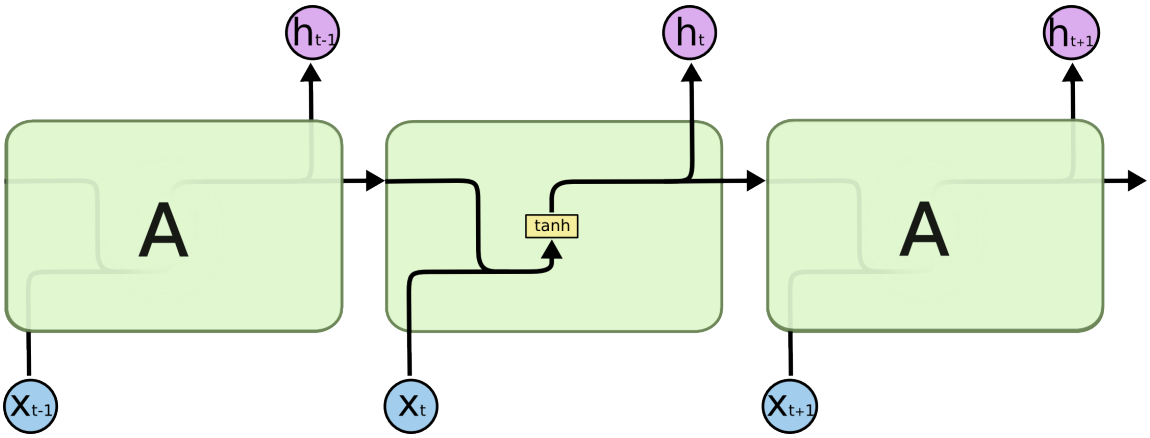
Generalmente la funzione di attivazione σ utilizzata è la tangente iperbolica (4.4). Nota f è quindi possibile completare il calcolo di backpropagation through time (4.16) utilizzando la regola di derivazione del prodotto di due funzioni. Ad esempio per $u = \boldsymbol{U}_{ij}$ si ottiene il seguente risultato:

$$\begin{aligned}
\frac{\partial F(\boldsymbol{x}; \boldsymbol{U}, \boldsymbol{W}, \boldsymbol{v})}{\partial u} &= \frac{\partial g}{\partial \boldsymbol{h}^{(T)}} \frac{\partial \boldsymbol{h}^{(T)}}{\partial u} \\
&= \frac{\partial g}{\partial \boldsymbol{h}^{(T)}} \sum_{t=1}^{T} \frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \bar{\boldsymbol{h}}^{(t)}}{\partial u} \\
&= \frac{\partial g}{\partial \boldsymbol{h}^{(T)}} \sum_{t=1}^{T} \left[\left(\prod_{\tau=t+1}^{T} \frac{\partial \boldsymbol{h}^{(\tau)}}{\partial \boldsymbol{h}^{(\tau-1)}} \right) \frac{\partial \bar{\boldsymbol{h}}^{(t)}}{\partial u} \right] \\
&= \left(\frac{\partial \sigma(z)}{\partial z} \boldsymbol{v} \right) \sum_{t=1}^{T} \left[\left(\frac{\partial \boldsymbol{\sigma}(z)}{\partial z} \boldsymbol{W} \right)^{T-t} \left(\frac{\partial \boldsymbol{\sigma}(z)}{\partial z} \boldsymbol{x}^{(t)} \frac{\partial \boldsymbol{U}}{\partial u} \right) \right], \qquad (4.18)
\end{aligned}$$

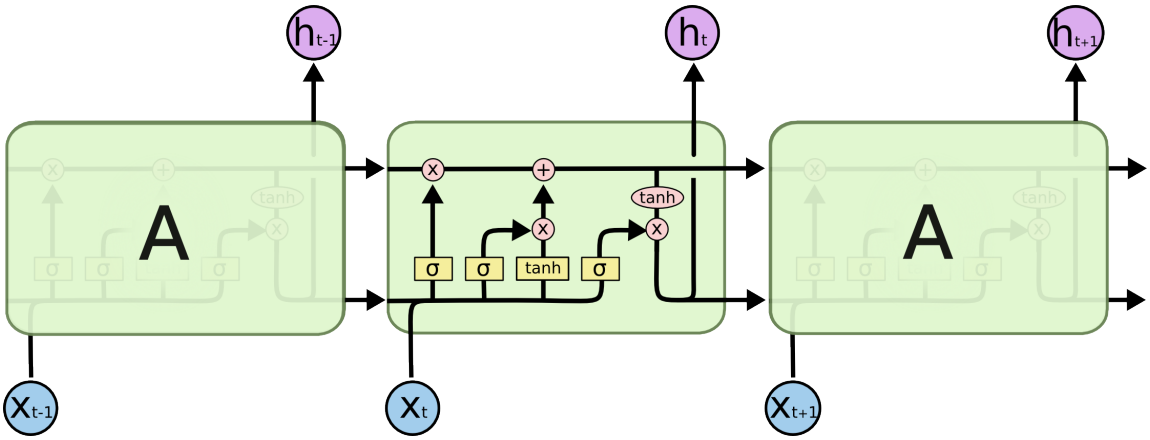
dove con $\frac{\partial \bar{\boldsymbol{h}}^{(t)}}{\partial u}$ si intende la derivata parziale di $\boldsymbol{h}^{(t)}$ rispetto a u, con $\boldsymbol{h}^{(t-1)}$ ritenuto costante rispetto al termine di derivazione. Il problema principale delle reti ricorrenti definite tramite (4.17) è che più la sequenza di input è lunga, minore è l'informazione dei primissimi valori in ingresso che il modello manterrà in memoria fino a T. Come visibile in (4.18), infatti, il gradiente di F, e quindi quello della funzione di rischio empirica, decade esponenzialmente in T se le componenti di $\boldsymbol{W}$ sono minori di 1. Ciò significa che gli addendi provenienti dai primi termini della sequenza di input risulteranno in un valore relativamente molto basso inficiando l'apprendimento della rete proprio rispetto alla significatività di questi (Sottosezione 4.4.1). Per questo motivo si dice che la versione classica di rete neurale ricorrente non è in grado di gestire le *long-term dependencies* (Bengio, Simard e Frasconi, 1994). Risulta necessario quindi modificare la topologia del modello allo scopo di rimediare a questa debolezza. La RNN sopra esposta (Figura 4.9a), infatti, possiede come unità fondamentale il percettrone, estremamente versatile quanto tuttavia generico rispetto al compito che la rete deve eseguire. A questo scopo sono stati ideati altri tipi di unità fondamentali specifiche per problemi di tipo ricorrente. Ne sono un esempio le *long-short term memory units* (LSTM cells), tra le prime ad essere proposte (1997) e attualmente le più popolari ed utilizzate.

Le reti neurali LSTM (Hochreiter e Schmidhuber, 1997) (Figura 4.9b), in seguito spiegate in dettaglio, differiscono da quelle più semplici in (4.17) per due motivi: introducono il cosiddetto stato della cella $\boldsymbol{c}^{(t)}$ e definiscono f in (4.13) non come un singolo hidden layer, ma bensì come la composizione di quattro strati in grado di interagire tra loro in modo particolare. Nel contesto delle celle LSTM, inoltre, la dimensione di $\boldsymbol{h}^{(t)}$ è detta, in maniera fuorviante, *number of units* della rete e sarà indicata con ν nelle pagine seguenti. Lo stato della cella $\boldsymbol{c}^{(t)}$ possiede lo stesso numero di componenti di $\boldsymbol{h}^{(t)}$ e si propaga da un'unità all'altra, nel corso del tempo, interagendo in maniera controllata, tramite i cosiddetti *gates*, con gli input e gli stati del sistema. I gates hanno lo stesso ruolo che un rubinetto ha nei confronti di un flusso d'acqua: attraverso una funzione sigmoidale consentono alla rete di aggiungere più o meno informazione proveniente da alcune variabili alla cella seguente. La cella LSTM possiede tre gates.

Il primo dei quattro strati che compongono la cella LSTM è il *forget gate layer* che ha lo scopo di determinare quanta nuova informazione in ingresso $\boldsymbol{x}^{(t)}$, rispetto alla conoscenza dello stato precedente $\boldsymbol{h}^{(t-1)}$, l'unità debba scartare/dimenticare in quanto inutile negli step successivi. Questo strato consiste nell'applicazione di una funzione di attivazione sigmoidale (4.3) sulla somma



(a) Rete neurale ricorrente semplice.



(b) Rete neurale ricorrente LSTM.

Figura 4.9: Rappresentazione schematica della struttura di due tipi di reti neurali ricorrenti (Olah, 2015).

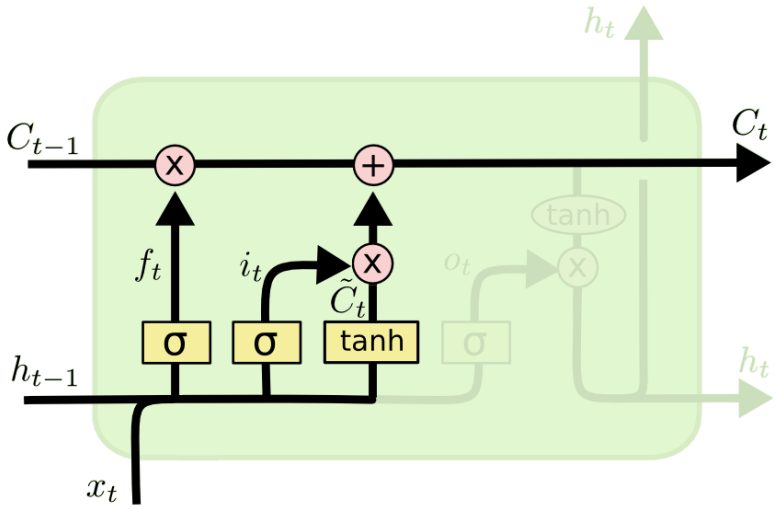


Figura 4.10: Rappresentazione grafica delle operazioni coinvolte all'interno di una cella LSTM per l'aggiornamento dello stato $\boldsymbol{c}^{(t-1)}$ (Olah, 2015).

pesata di $\boldsymbol{h}^{(t-1)}$ e $\boldsymbol{x}^{(t)}$: è appunto un gate. Il risultato, indicato con $\boldsymbol{f}_t$, è quindi un vettore di componenti ν, poiché $\boldsymbol{W}_f$ e $\boldsymbol{U}_f$ hanno dimensioni rispettivamente di $\nu \times \nu$ e $\nu \times n$, e aventi valori compresi tra 0 e 1:

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_f\, \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_f\, \boldsymbol{x}^{(t)} + \boldsymbol{b}_f\right) \tag{4.19}$$

Forget Gate Layer

Il secondo layer $\boldsymbol{i}_t$ detto *input gate layer*, applica le stesse operazioni del primo. Lo scopo però è opposto: in questo caso il modello utilizza $\boldsymbol{W}_i$ per decidere quali informazioni memorizzare:

$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_i\, \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_i\, \boldsymbol{x}^{(t)} + \boldsymbol{b}_i\right) \tag{4.20}$$

Input Gate Layer

Per aggiornare lo stato $\boldsymbol{c}^{(t-1)}$ è necessario costruirne un "prototipo", indicato con $\tilde{\boldsymbol{c}}^{(t)}$, basato sullo stato del sistema nell'istante precedente $\boldsymbol{h}^{(t-1)}$ e il nuovo input $\boldsymbol{x}^{(t)}$. Questo compito è realizzato dal terzo strato della cella LSTM il quale utilizza la tangente iperbolica come funzione di attivazione poiché lo stato può essere definito positivo o negativo, e per convenzione di modulo massimo unitario:

$$\tilde{\boldsymbol{c}}^{(t)} = \tanh\left(\boldsymbol{W}_c\, \boldsymbol{h}^{(t-1)} + \boldsymbol{U}_c\, \boldsymbol{x}^{(t)} + \boldsymbol{b}_c\right) \tag{4.21}$$

Tanh Layer

I risultati dei tre strati sono quindi combinati (Figura 4.10) per aggiornare lo stato della cella al tempo t:

$$\boldsymbol{c}^{(t)} = \boldsymbol{f}_t \circ \boldsymbol{c}^{(t-1)} + \boldsymbol{i}_t \circ \tilde{\boldsymbol{c}}^{(t)} \tag{4.22}$$

Cell State Update

Con la notazione "$\circ$" si intende il prodotto di Hadamard, effettuato cioè elemento per elemento. Si osservi che al secondo membro il primo termine non è altro che la scelta, frutto del forget gate, di quanto ricordare del vecchio stato $\boldsymbol{c}^{(t-1)}$; l'altro addendo esprime invece quanto il modello debba memorizzare del candidato costruito tramite i soli $\boldsymbol{h}^{(t-1)}$ e $\boldsymbol{x}^{(t)}$. Con la (4.22) si comprende bene qual è il ruolo svolto dallo stato della cella all'interno di una rete LSTM: con questa architettura $\boldsymbol{c}^{(t)}$ costituisce la "memoria" del modello che si distingue da $\boldsymbol{h}^{(t)}$, l'output e stato del sistema. Il

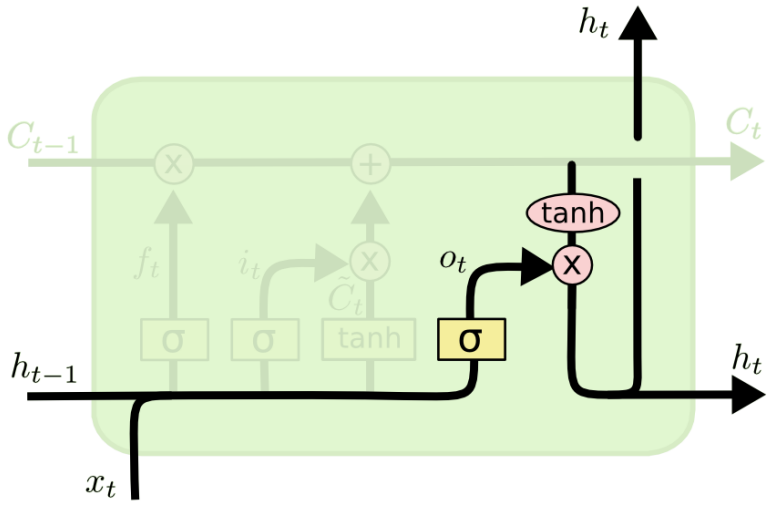


Figura 4.11: Rappresentazione grafica delle operazioni coinvolte all'interno di una cella LSTM per l'aggiornamento dello stato del sistema $\boldsymbol{h}^{(t-1)}$ (Olah, 2015).

quarto e ultimo layer ha infine il compito di aggregare lo stato della cella aggiornato $\boldsymbol{c}^{(t)}$, lo stato del sistema in $(t-1)$ $\boldsymbol{h}^{(t-1)}$ e il nuovo input $\boldsymbol{x}^{(t)}$ in modo da ottenere $\boldsymbol{h}^{(t)}$. Si procede analogamente a quanto fatto per $\boldsymbol{c}^{(t)}$ utilizzando un gate, detto di output $\boldsymbol{o}_t$, che può essere scritto analogamente agli altri due come

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_o\,\boldsymbol{h}^{(t-1)} + \boldsymbol{U}_o\,\boldsymbol{x}^{(t)} + \boldsymbol{b}_o\right). \tag{4.23}$$

Output Gate

Esso ha lo scopo di riassumere quanta e quale informazione di $\boldsymbol{h}^{(t-1)}$ e $\boldsymbol{x}^{(t)}$ debba essere considerata ai fini della definizione del nuovo stato del sistema. Si procede poi alla costruzione di un candidato $\tilde{\boldsymbol{h}}^{(t)}$ funzione solamente di $\boldsymbol{c}^{(t)}$ e definito come in (4.21) per mezzo della tangente iperbolica.

$$\tilde{\boldsymbol{h}}^{(t)} = \tanh\left(\boldsymbol{c}^{(t)}\right)$$

Si aggregano infine le due informazioni.

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}_t\tilde{\boldsymbol{h}}^{(t)} \tag{4.24}$$

System State Update

In questo caso non c'è la necessità di utilizzare un forget gate in quanto l'idea di stato della cella $\boldsymbol{c}^{(t)}$ è proprio quello di portare con sè, dipendentemente dai nuovi input, le informazioni sufficienti, e potenzialmente sovrabbondanti, necessarie per la definizione di $\boldsymbol{h}^{(t)}$. Una cella LSTM è in grado di memorizzare e scartare informazioni grazie all'utilizzo di numerosi parametri come $\boldsymbol{W}_f$, $\boldsymbol{U}_f$, $\boldsymbol{b}_f$ (Hochreiter e Schmidhuber, 1997). Nella fase di training la rete neurale tenta di assegnare loro i migliori valori al fine di minimizzare la funzione di rischio empirica e quindi permettere la sola memorizzazione delle informazioni che risultano utili a spiegare il fenomeno. Per quanto riguarda la funzione g in (4.14), che lega lo stato del sistema alla variabile target, questa è generalmente costruita come uno (come in (4.17)) o più fully-connected layer. La scelta della topologia della porzione terminale della rete neurale è, come accade per la dimensione di $\boldsymbol{h}^{(t)}$, un iperparametro.

Si calcola, infine, la backpropagation di una rete neurale ricorrente LSTM per u componente di $\boldsymbol{U}_f$ come

$$\begin{aligned}\frac{\partial F(\boldsymbol{x})}{\partial u} &= \frac{\partial g}{\partial \boldsymbol{h}^{(T)}} \frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{c}^{(T)}} \frac{\partial \boldsymbol{c}^{(T)}}{\partial u} \\ &= \frac{\partial g}{\partial \boldsymbol{h}^{(T)}} \frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{c}^{(T)}} \sum_{t=1}^{T} \frac{\partial \boldsymbol{c}^{(T)}}{\partial \boldsymbol{c}^{(t)}} \frac{\partial \bar{\boldsymbol{c}}^{(t)}}{\partial u} \\ &= \frac{\partial g}{\partial \boldsymbol{h}^{(T)}} \frac{\partial \boldsymbol{h}^{(T)}}{\partial \boldsymbol{c}^{(T)}} \sum_{t=1}^{T} \left[\left(\prod_{\tau=t+1}^{T} \frac{\partial \boldsymbol{c}^{(\tau)}}{\partial \boldsymbol{c}^{(\tau-1)}} \right) \frac{\partial \bar{\boldsymbol{c}}^{(t)}}{\partial u} \right]\end{aligned}$$

dove, indicando con $\boldsymbol{I}_\nu$ la matrice identità di dimensioni $\nu \times \nu$,

$$\begin{aligned}\frac{\partial \boldsymbol{c}^{(\tau)}}{\partial \boldsymbol{c}^{(\tau-1)}} &= \frac{\partial}{\partial \boldsymbol{c}^{(\tau-1)}} \left(\boldsymbol{f}_\tau \circ \boldsymbol{c}^{(\tau-1)} + \boldsymbol{i}_\tau \circ \tilde{\boldsymbol{c}}^{(\tau)} \right) \\ &= \boldsymbol{I}_\nu \, \boldsymbol{f}_\tau\end{aligned}$$

e

$$\begin{aligned}\frac{\partial \bar{\boldsymbol{c}}^{(t)}}{\partial u} &= \frac{\partial}{\partial u} \left(\boldsymbol{f}_\tau \circ \boldsymbol{c}^{(\tau-1)} + \boldsymbol{i}_\tau \circ \tilde{\boldsymbol{c}}^{(\tau)} \right) \\ &= \left(\boldsymbol{c}^{(t-1)} \circ \frac{\partial \boldsymbol{\sigma}(z)}{\partial z} \right) \frac{\partial \boldsymbol{U}_f}{\partial u}.\end{aligned}$$

Si ottiene allora il seguente risultato.

$$\frac{\partial F(\boldsymbol{x})}{\partial u} = \frac{\partial g}{\partial \boldsymbol{h}^{(T)}} \frac{\partial \mathbf{tanh}(z)}{\partial z} \sum_{t=1}^{T} \left[\left(\boldsymbol{I}_\nu \, \boldsymbol{f}_\tau \right)^{T-t} \left(\left(\boldsymbol{c}^{(t-1)} \circ \frac{\partial \boldsymbol{\sigma}(z)}{\partial z} \right) \frac{\partial \boldsymbol{U}_f}{\partial u} \right) \right] \tag{4.25}$$

La dipendenza della derivata parziale da un unico gate, in questo caso il forget gate, permette di gestire facilmente il modo in cui il modello memorizza le informazioni provenienti dai primi istanti t. Inizializzando i gate a 1, il modello LSTM è in grado di gestire più facilmente le long-term dependencies rispetto a (4.17) che ha backpropagation (4.18). Si osservi, infatti, che il fattore $\left(\boldsymbol{I}_\nu \, \boldsymbol{f}_\tau \right)^{T-t}$ non dipende da alcuna funzione di attivazione che costringe i suoi valori ad essere minori di uno (per maggiori dettagli si veda la Sottosezione 4.4.1).

4.4 Instabilità del gradiente

Tra gli algoritmi iterativi di ricerca del minimo della funzione di rischio empirica (3.9), i più utilizzati per quanto riguarda le reti neurali sono basati sulla discesa del gradiente di f. Spesso, per quanto discusso nella Sottosezione 3.3.2, per aggiornare i parametri del modello si preferisce utilizzare la tecnica dello stochastic gradient descent o alcune sue varianti ancora più sofisticate. L'elemento fondamentale di tutti questi algoritmi rimane comunque la backpropagation dell'errore lungo la rete neurale. Nel calcolo del gradiente di f è però necessario distinguere le sue componenti rispetto a quale layer il relativo parametro appartiene. Si consideri fissata una determinazione $\boldsymbol{x}$ delle feartures. Per un MLP ad un hidden layer (come quello in (4.9)), ad esempio, la componente i-esima di $\nabla_{(\boldsymbol{W}^{(1)},\boldsymbol{w}^{(2)})} f$ è calcolata, analogamente a quanto descritto per il percettrone in (4.8). Posto n il numero di variabili esplicative componenti di $\boldsymbol{x}$, k il numero di unità dell'hidden layer,

$$\begin{aligned} s_j^{(1)} &= \sum_{i=0}^{n} w_{ij}^{(1)} x_i, \\ \tilde{\boldsymbol{x}} &= (\tilde{x}_1, \dots, \tilde{x}_k)^T, \qquad \text{con } \tilde{x}_j = h\left(s_j^{(1)}\right), \\ s^{(2)} &= \sum_{j=0}^{k} w_j^{(2)} \tilde{x}_j, \end{aligned}$$

si ha[7]

$$\begin{aligned} \frac{\partial f\left(\boldsymbol{x}; \boldsymbol{W}^{(1)}, \boldsymbol{w}^{(2)}\right)}{\partial w_{ij}^{(1)}} &= \frac{\partial h(s^{(2)})}{\partial s^{(2)}} \frac{\partial s^{(2)}(\tilde{\boldsymbol{x}}; \boldsymbol{w}^{(2)})}{\partial \tilde{x}_j} \frac{\partial h(s_j^{(1)})}{\partial s_j^{(1)}} \frac{\partial s_j^{(1)}(\boldsymbol{x}; \boldsymbol{w}_j^{(1)})}{\partial w_{ij}^{(1)}} \\ &= \frac{\partial h(s^{(2)})}{\partial s^{(2)}} w_j^{(2)} \frac{\partial h(s_j^{(1)})}{\partial s_j^{(1)}} x_i \end{aligned} \tag{4.26}$$

nel caso in cui $w_{ij}^{(1)}$ sia componente della matrice dei pesi dell'hidden layer $\boldsymbol{W}^{(1)}$ o come

$$\begin{aligned} \frac{\partial f\left(\boldsymbol{x}; \boldsymbol{W}^{(1)}, \boldsymbol{w}^{(2)}\right)}{\partial w_i^{(2)}} &= \frac{\partial h(s^{(2)})}{\partial s^{(2)}} \frac{\partial s^{(2)}(\tilde{\boldsymbol{x}}, \boldsymbol{w}^{(2)})}{\partial w_i^{(2)}} \\ &= \frac{\partial h(s^{(2)})}{\partial s^{(2)}} \tilde{x}_i \\ &= \frac{\partial h(s^{(2)})}{\partial s^{(2)}} h\left(\boldsymbol{w}_i^{(1)} \boldsymbol{x}\right) \end{aligned} \tag{4.27}$$

se $w_i^{(2)}$ appartiene al vettore dei pesi dell'output layer $\boldsymbol{w}^{(2)}$. Come si vede in (4.26) e (4.27) il gradiente della funzione dipende dal valore dei pesi della rete neurale prima dell'iterazione e dalla specifica funzione di attivazione scelta. Questi due elementi influenzano quindi il calcolo dei

[7] Nel caso in cui il peso $w_{ij}^{(1)}$ o $w_i^{(2)}$ sia in realtà un bias, si ricordi che il rispettivo fattore è posto uguale a 1. Ciò significa che calcolando la derivata parziale di f rispetto a $w_{0j}^{(1)}$, $j = 1, \dots, k$ allora $x_0 = 1$, mentre per $w_0^{(2)}$ si ha che è $\tilde{x}_0$ ad essere unitario.

parametri ottimali. Per un'incorretta scelta dei parametri iniziali e/o della topologia (riferendosi all'activation function), è addirittura possibile che durante la fase di allenamento si verifichi un'instabilità del gradiente tale da impedire alla rete neurale di apprendere alcunché. Un simile comportamento può presentarsi in seguito alla "scomparsa" del gradiente o alla sua "esplosione". Tali problemi sono trattati nelle prossime due sottosezioni, rispettivamente.

4.4.1 Vanishing gradient problem

Il problema di scomparsa del gradiente (*vanishing gradient problem*) si verifica soprattutto nelle reti neurali molto profonde ed è imputabile alle matrici dei pesi associate ai primi hidden layer. Il gradiente svanisce quando, a causa dell'elevato numero di strati, dell'inizializzazione dei parametri e della funzione di attivazione scelta, questo assume valori prossimi a zero provocando una modifica impercettibile dei parametri nel corso delle epoche. Questo comportamento si traduce nel mancato apprendimento del modello ed è anche responsabile dell'inefficienza delle reti neurali ricorrenti semplici nella gestione delle long-term dependencies. Nonostante il multi-layer perceptron utilizzato sopra non sia particolarmente profondo, è possibile comprendere già da (4.26) che più grande è il numero di layer, maggiore è il numero di fattori in cui può essere scomposta la derivata parziale di f rispetto ad un parametro dei primi strati. Calcolando la derivata rispetto ad un peso $w_i^{(1)}$ del primo layer, ad esempio, si ottene un numero di fattori pari al doppio degli strati sui cui l'errore è retropropagato: in (4.26) infatti se ne ottengono quattro. Per un percettrone multistrato con L hidden layer se ne otterrebbero $2(L+1)$.

$$\frac{\partial f}{\partial w_i^{(1)}} = \frac{\partial h\left(s^{(L+1)}\right)}{\partial s^{(L+1)}}\, w^{(L+1)}\, \frac{\partial h\left(s^{(L)}\right)}{\partial s^{(L)}}\, w^{(L)}\, \dots\, \frac{\partial h\left(s^{(2)}\right)}{\partial s^{(2)}}\, w^{(2)}\, \frac{\partial h\left(s^{(1)}\right)}{\partial s^{(1)}}\, x_i \tag{4.28}$$

Poiché per ogni strato della rete è applicata una funzione di attivazione ad una somma di variabili, infatti, la derivazione composta sul suddetto layer produce due fattori, uno relativo all'activation function e l'altro alla somma. Questo è il motivo per cui il numero dei fattori è sempre pari. Se tutti, o la gran parte, dei (molti) fattori in cui è scomposta $\partial f \,/\, \partial w_i^{(1)}$ assumono valore inferiore all'unità allora per effetto moltiplicativo il risultato sarà molto vicino a zero. Lo stesso effetto si ottene considerando anche la BPTT (4.18) e (4.25). Per le considerazioni appena fatte, quindi, un simile risultato è dovuto all'utilizzo di funzioni di attivazione che hanno derivata sempre minore o uguale ad uno (come la funzione logistica o la tangente iperbolica) e ad un'inadeguata inizializzazione dei parametri.

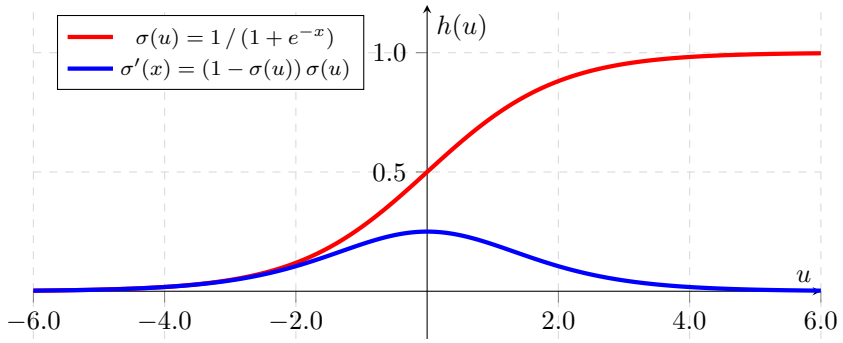
Tipicamente i parametri iniziali del modello sono ottenuti come determinazioni di una variabile aleatoria, nella maggioranza dei casi con distribuzione uniforme o normale. Importante nota è che i loro valori non devono essere mai inizializzati tutti a 0, altrimenti il gradiente sarebbe nullo e il modello non potrebbe apprendere. Non crea particolari problemi, invece, porre solo i valori di alcuni pesi, indicati con $\tilde{\boldsymbol{w}}$, prossimi a 0 (magari perché determinazioni "sfortunate" della variabile aleatoria): le derivate che non contengono quelle variabili sono positive e quindi permettono al modello di imparare e di modificare conseguentemente il valore di $\tilde{\boldsymbol{w}}$. Ad esempio, è possibile inizializzare come nulli i bias di ogni strato della rete senza causare problemi all'apprendimento. Utilizzare una variabile aleatoria da cui ricavare i valori iniziali dei parametri è un modo pratico per rompere la simmetria del modello, che invece si avrebbe assegnando una costante a tutti i suoi pesi e bias. Se infatti tutti i parametri fossero uguali tra loro, allora le unità di ogni layer sarebbero tra loro uguali. Si consideri infatti un MLP fully-connected: le unità del primo layer,

ricevendo tutte il medesimo input $\boldsymbol{x}$ e avendo la stessa inizializzazione, restituirebbero ognuna lo stesso output che si comporterebbe da ingresso per il secondo strato e così via. Questa forma di simmetria all'interno della rete limiterebbe notevolmente l'apprendimento in quanto (4.28) sarebbe uguale per ogni neurone dello specifico strato (indipendentemente dal valore assunto dalle features). Applicando invece un'inizializzazione randomizzata, ogni neurone può differenziarsi dagli altri ed apprendere pattern diversi migliorando le capacità predittive dell'intero modello. Inoltre, applicare valori casuali ai parametri all'inizio della fase training segue lo stesso principio per cui lo SGD permette di trovare minimi "migliori", se non quello globale, rispetto alla versione deterministica del gradient descent (Sottosezione 3.3.2). Eseguendo più volte l'allenamento del modello, infatti, non si otterrà indicativamente lo stesso set di migliori parametri; lanciando più run c'è quindi la possibilità di selezionare l'insieme dei valori che garantisce le migliori performance della rete neurale sul validation set.

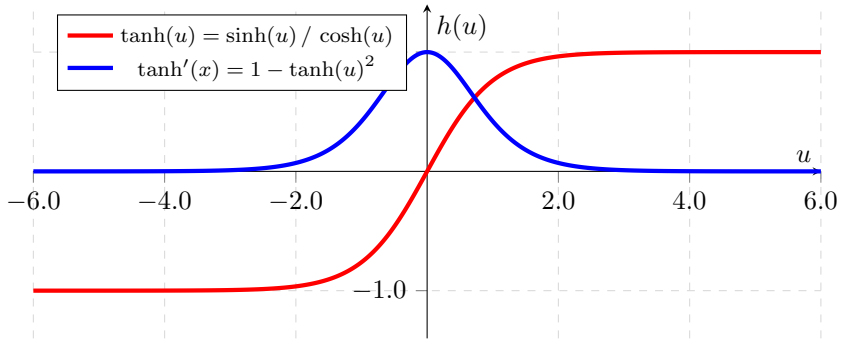
Nonostante gli ampi vantaggi di questa metodologia di inizializzazione, senza particolari accortezze, il suo utilizzo può portare al problema di vanishing del gradiente. Si pensi ad esempio ad un modello in grado di ricevere n input e inizializzato tramite variabili normali con varianza σ^2. Supponendo che $\boldsymbol{x} = \mathbf{1}$ e considerando una generica unità del primo hidden layer, allora $s^{(1)}(\boldsymbol{x})$ ha deviazione standard pari a $\sigma^{(1)} = \sqrt{n}\sigma$. Se n è grande allora lo è anche $\sigma^{(1)}$, implicando che, seppur con bassa probabilità (che però aumenta con n), i valori di $s^{(1)}$ possano assumere valori notevolmente grandi o piccoli. In questi casi, allora, funzioni con asintoto orizzontale possono restituire derivate molto piccole (Figura 4.12). Questo effetto è moltiplicato a cascata se anche tutti gli altri fattori $\partial h(s) \,/\, \partial s$ sono minori o uguali a uno. Ne risulta che, se il fenomeno è frequente perché n è grande, il modello non è in grado di apprendere poiché la retropropagazione dell'errore restituisce gradiente prossimo a 0. È esattamente il motivo per cui la RNN in (4.17) non è in grado di apprendere long-term dependencies (T molto grande): si osservi nella backpropagation through time (4.18), infatti, la presenza del termine $\left(\frac{\partial \boldsymbol{\sigma}(z)}{\partial z}\, \boldsymbol{W}\right)^{T-t}$. La rete neurale ricorrente LSTM, invece, non prevede per lo stesso fattore dipendenza dalla funzione di attivazione e quindi è in grado di evitare nella maggior parte dei casi il problema di scomparsa del gradiente. La soluzione più semplice al problema consiste nella scelta di $\sigma \ll 1$ e con valore dipendente dal numero di feature (o di input nel caso di layer più profondi). L'inizializzazione di Xavier, ad esempio, consiste nello scegliere i valori iniziali dei parametri come determinazioni di distribuzioni normali di varianza inversamente proporzionale a n. Con $\sigma^2 = c^2/n$, infatti, si ottiene $\sigma^{(1)} = c$ con c costante desiderata (di solito $c = 0.1$). Soluzioni più articolate coinvolgono non solo il numero di input n_{in} del neurone, ma anche quelli di output n_{out}, definendo $\sigma^2 = c^2/(n_{in} + n_{out})$. Quando possibile, per evitare l'instabilità del gradiente, conviene quindi non utilizzare funzioni di attivazione con asintoti orizzontali. Alcune volte, però, la scelta di un'activation function dell'output layer con questo comportamento asintotico è imprescindibile. Se infatti Y ha determinazioni in un intervallo chiuso è necessario utilizzare una funzione di output della rete che si comporti allo stesso modo qualsiasi sia il risultato di $s^{(L+1)}$. Esempi di funzioni utilizzabili a questo scopo sono quindi la funzione sigmoidale, la tanh o, nel caso di più output, la funzione esponenziale normalizzata/softmax:

$$softmax(u_j) = \frac{e^{u_j}}{\sum_{k=1}^{K} e^{u_k}}, \quad j = 1, \ldots, K.$$

Per questo motivo è buona norma inizializzare adeguatamente i parametri della rete neurale onde evitare problemi di scomparsa del gradiente. Per gli hidden layer, però, dove la scelta della funzione di attivazione è arbitraria conviene utilizzare la funzione rettificata ReLU che prevede un solo



(a) Funzione logistica.



(b) Funzione tangente iperbolica.

Figura 4.12: Esempi di funzioni di attivazione con derivata avente asintoto $x = 0$.

asintoto orizzontale. La sua derivata può infatti essere scritta come una Θ di Heaviside (4.1). Il fatto che per $s > 0$ la funzione restituisca valori unitari implica che non vi è degradazione del segnale durante la backpropagation nemmeno per le reti più profonde. Per contro, però, se s è minore di zero allora h' è nulla, annullando quindi l'intera derivata parziale di f. Questo inconveniente può essere risolto definendo una nuova funzione detta Leaky ReLU

$$LeakyReLU(u) = h(u) = \max(0.01\,u, u) = \begin{cases} u & u \geq 0 \\ 0.01\,u & u < 0 \end{cases} \tag{4.29}$$

avente derivata sempre positiva.

$$\frac{\mathrm{d}LeakyReLU(u)}{\mathrm{d}u} = h'(u) = \begin{cases} 1 & u > 0 \\ 0.01 & u < 0 \end{cases}$$

Con una corretta inizializzazione dei parametri, tuttavia, spesso non è necessario ricorrere a quest'ultima funzione. Generalmente basta infatti utilizzare la funzione di attivazione ReLU e l'inizializzazione di Xavier per non generare instabilità nel gradiente. Ai fini della backpropagation, inoltre, la velocità di esecuzione delle derivate di ReLU/LeakyReLU (basta effettuare un controllo

del segno) è estremamente più elevata che calcolare esponenziali come necessario fare se si utilizza una delle altre funzioni di attivazione citate. Per questo motivo la funzione ReLU è attualmente l'activation function più utilizzata negli hidden layer. Come già anticipato, il fenomeno della scomparsa del gradiente inficia pesantemente l'apprendimento di tutta la rete neurale ricorrente in quanto i pesi $\boldsymbol{W}$ sono condivisi lungo tutti gli strati. Se però per la RNN "semplice" (4.17) questo problema è difficilmente risolvibile in quanto è complesso calibrare inizializzazione e valori restituiti dalle funzioni di attivazione, così non è per le celle LSTM. Queste ultime, infatti, richiedono semplicemente che i gate siano inizializzati ad 1.

4.4.2 Exploding gradient problem

Il problema dell'esplosione del gradiente (*exploding gradient problem*) è generalmente meno comune della sua controparte e riguarda principalmente le reti neurali ricorrenti. Quando un gradiente assume valori estremamente grandi, l'algoritmo di gradient descent forza la rete neurale a compiere aggiornamenti spropositati sui parametri, impedendogli di trovare il minimo di R_{emp}. Di fatto quindi, l'esplosione del gradiente rende inefficace il processo di allenamento e ottimizzazione del modello. Il gradient exploding problem è indipendente dalla funzione di attivazione utilizzata in quanto è dovuto, nel calcolo della backpropagation, all'elevato numero di termini che assumono valori superiori all'unità rispetto a quelli con valori inferiori a 1. Le RNN, in particolare, presentano spesso questo sbilanciamento e per questo sono le più soggette ad esplosione del gradiente. In (4.18) si vede infatti che la matrice $\boldsymbol{W}$ è moltiplicata per se stessa per un numero elevato di volte. Se le sue componenti assumono valori significativamente maggiori dell'unità allora ciò può comportare all'esplosione il gradiente di f. I modi più comuni per limitare questo problema sono il cambiamento di topologia della rete neurale e il *gradient clipping*. Nel primo caso usualmente si passa da un modello ricorrente semplice ad uno basato sulle celle LSTM, in cui è $|\boldsymbol{f}_t| < 1$ ad essere moltiplicato molteplici volte, beneficiando oltretutto delle loro capacità di gestire le long-term dependencies. Il gradient clipping, invece, permette di limitare ad ogni iterazione dell'algoritmo di gradient descent quanto i parametri debbano essere aggiornati. Questa tecnica agisce sul valore di $\nabla_{\boldsymbol{\theta}} f$, fissandone una soglia massima b (*threshold*) raggiungibile. Se il gradiente eccede b allora l'algoritmo di ricerca del minimo esegue comunque un apprendimento basato su $\nabla_{\boldsymbol{\theta}} f = b$. Il valore di threshold costituisce un ulteriore iperparametro del modello.

L'esplosione del gradiente può essere contrastata anche tramite regolarizzazione. Applicare ad esempio un regularizer di tipo lasso o ridge, come esposto nella Sottosezione 3.3.2, obbliga la rete neurale a mantenere "piccoli" i valori dei parametri, e quindi anche il risultato delle relative derivate parziali su f.

4.4.3 Algoritmi iterativi stocastici avanzati

Per diminuire la probabilità di incorrere in fenomeni di instabilità del gradiente, sopratutto riguardo quanto discusso nella Sottosezione 4.4.2, e aumentare la velocità di ricerca del minimo sono stati ideati diversi algoritmi iterativi come evoluzione della più classica discesa stocastica del gradiente. Lo SGD è infatti molto lento nel trovare un (buon) minimo di R_{emp} perchè, come già detto nella Sottosezione 3.3.2, minimizzare la loss function su qualche punto non implica che valga lo stesso anche per la funzione di rischio; l'algoritmo può trovare parametri che deviano dal percorso più breve che, sul grafico $\mathcal{G}$ di R_{emp} in funzione di $\boldsymbol{\theta}$, collega lo starting point $\boldsymbol{\theta}_0$ al minimo globale $\hat{\boldsymbol{\theta}}$. Lo stochastic gradient descent di epoca in epoca fornisce valori che si avvicinano al minimo facendo

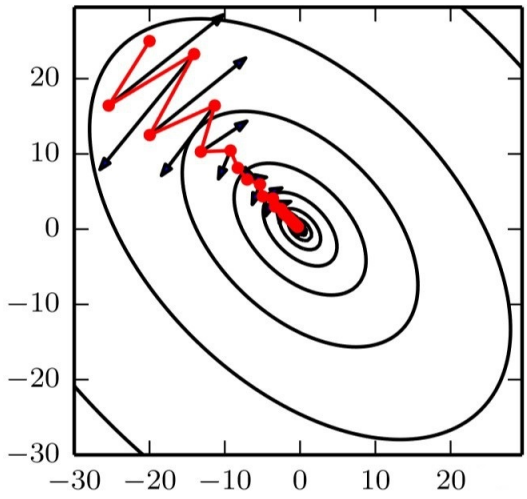


Figura 4.13: Esempio di come l'algoritmo (4.31) si muove, in rosso, lungo le curve di livello della funzione di rischio associata ad un modello a due parametri. Le frecce rappresentano il gradiente della loss function calcolato sul rispettivo punto (Goodfellow, Bengio e Courville, 2016). Se ne osservino le ampie oscillazioni laterali.

però oscillare $(\boldsymbol{\theta}, R_{emp}(\mathcal{D}, \boldsymbol{\theta}))$ in direzioni diverse da quella più breve su $\mathcal{G}$ (Figura 4.13). Parte del tempo di esecuzione dell'algoritmo è quindi "sprecato" nel muovere il punto $(\boldsymbol{\theta}, R_{emp}(\mathcal{D}, \boldsymbol{\theta}))$ in direzioni non volute. Nello SGD, inoltre, ogni parametro è allenato con una velocità diversa in quanto si utilizza lo stesso learning rate η. In una rete neurale, ad esempio, unità dello stesso layer ottengono in ingresso valori che possono essere significativamente diversi e più o meno sparsi; le derivate associate ai loro parametri possono dunque avere anche ordini di grandezza differenti. Applicare lo stesso learning rate modifica l'intensità dell'aggiornamento, ma non il rapporto relativo tra le componenti del gradiente. Così facendo però, il percorso ottenuto tramite l'applicazione dell'algoritmo stocastico risulta molto irregolare perchè dipendente più dalla posizione istantanea che dalla destinazione finale.

Gli algoritmi di ricerca del minimo che rispetto allo SGD migliorano la velocità di convergenza e limitano le problematiche di esplosione del gradiente sono molteplici, i più famosi sono però AdaGrad (Duchi, Hazan e Singer, 2011), RMSProp (G. Hinton, Srivastava e Swersky, 2012) e Adam (Kingma e Ba, 2014), e sono tutti basati sul concetto fondamentale di momento (*momentum*). Il momentum[8] $\Delta\boldsymbol{\theta}_{n+(j+1)/\kappa}$, riferito alla j-esima iterazione della n-esima epoca, è definito come la grandezza vettoriale tale che

$$\begin{aligned}\Delta\boldsymbol{\theta}_{n+(j+1)/\kappa} &= \alpha\,\Delta\boldsymbol{\theta}_{n+j/\kappa} + \eta\,\nabla_{\boldsymbol{\theta}} L\left(f(\boldsymbol{x}_j;\boldsymbol{\theta}_{n+j/\kappa}), y_j\right) \\ &= \eta \sum_{i=0}^{n\kappa+j} \alpha^{n\kappa+j-i}\ \nabla_{\boldsymbol{\theta}} L\left(f(\boldsymbol{x}_j;\boldsymbol{\theta}_{i/\kappa}), y_j\right), \quad (\boldsymbol{x}_j, y_j) = \gamma\left(\mathcal{D}_j\right), \end{aligned} \tag{4.30}$$

[8] Nel testo $\Delta\boldsymbol{\theta}_n$ sarà chiamato momentum, nella sua accezione inglese, per distinguerlo dalla definizione statistica di momento che sarà utilizzata di seguito.

essendo α un valore fissato, $\alpha < 1$, e $\Delta\boldsymbol{\theta}_0 = \mathbf{0}$. Il momentum rappresenta dunque l'aggiornamento dei parametri in funzione di quello prodotto nello step precedente (il cui effetto è controllato da α) e dal gradiente della loss function. Come si può osservare dalla seconda espressione, $\Delta\boldsymbol{\theta}_{n+(j+1)/\kappa}$ è esprimibile come la somma pesata di tutta la serie storica dei precedenti gradienti calcolati: più è "vecchio" il gradiente membro della somma, minore sarà la sua influenza ai fini di un nuovo aggiornamento dei parametri poiché α è inferiore all'unità. L'applicazione più semplice del momentum nella ricerca del minimo di R_{emp} (Figura 4.13) consiste nello riscrivere lo SGD (3.16) come

$$\boldsymbol{\theta}_{n+1} = \boldsymbol{\theta}_n - \sum_{j=1}^{\kappa} \Delta\boldsymbol{\theta}_{n+j/\kappa}, \tag{4.31}$$

dove ad ogni iterazione il singolo gradiente è sostituito dalla somma pesata con pesi esponenziali (4.30). L'adozione di (4.31) permette di raggiungere una velocità di convergenza superiore allo SGD. L'ipotesi fondante di un algoritmo stocastico è che la maggior parte dei gradienti della loss function puntino al set di parametri di minimo della funzione di rischio. Com'è noto, inoltre, ogni gradiente può essere rappresentato come un vettore e, per quanto già discusso, ci si aspetta che nel corso delle iterazioni questo non punti nella sola direzione di $\left(\hat{\boldsymbol{\theta}}, R_{emp}(\mathcal{D}, \hat{\boldsymbol{\theta}})\right)$, ma che possieda anche una componente laterale. L'adozione del momentum come somma pesata della serie storica

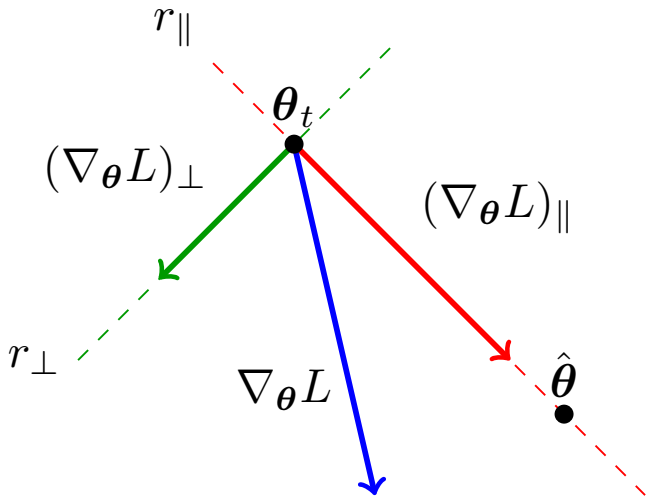


Figura 4.14: Scomposizione del vettore gradiente della loss function lungo le componenti perpendicolari $r_{\perp}$ e parallele $r_{\parallel}$ rispetto alla retta passante per $\boldsymbol{\theta}_t$ e $\hat{\boldsymbol{\theta}}$.

dei gradienti consente di mediare queste oscillazioni in modo da minimizzare epoca dopo epoca, ad effetto cascata, la componente laterale del nuovo aggiornamento dei parametri $\Delta\boldsymbol{\theta}_{n+(j+1)/\kappa}$. Ogni $\nabla_{\boldsymbol{\theta}} L$ può essere infatti scomposto tramite la distanza euclidea lungo la direzione del minimo ottenendo $(\nabla_{\boldsymbol{\theta}} L)_{\parallel}$ e quella ad essa perpendicolare $(\nabla_{\boldsymbol{\theta}} L)_{\perp}$ (Figura 4.14): di epoca in epoca in (4.30) la componente parallela acquista magnitudine, mentre quella perpendicolare, assunta come pressoché casuale, rimane contenuta. Nelle prime iterazioni, rispetto alle successive, generalmente il gradiente ha modulo $|\nabla_{\boldsymbol{\theta}} L|$ molto maggiore e per questo conviene usare un α relativamente basso

(ad esempio $\alpha = 0.5$) allo scopo di bilanciare il superamento dei minimi locali e il problema del mancato raggiungimento di un punto di minimo (*overshooting*). Al contrario, dopo molte iterazioni, i gradienti con $|\nabla_{\boldsymbol{\theta}} L|$ elevato hanno poca influenza in (4.30) in seguito al decadimento esponenziale, permettendo quindi l'utilizzo di valori di α più grandi ($\alpha = 0.9$ o anche $\alpha = 0.99$) (G. Hinton, Srivastava e Swersky, 2012).

Quanto presentato in (4.31) non risolve ancora però il problema per cui ogni parametro è allenato secondo lo stesso η. L'algoritmo RMSprop (*Root Mean Square Propagation*), ad esempio, per questo motivo introduce durante la ricerca del minimo un tasso di apprendimento adattivo basato su una versione modificata del concetto di momentum. Esso, infatti, non è utilizzato come delta di aggiornamento dei parametri $\Delta\boldsymbol{\theta}_{n+j/\kappa}$, ma è riferito al quadrato del gradiente della funzione di costo. Per questo motivo, nel seguito sarà rappresentato con il vettore $\boldsymbol{v}_t$ in cui per convenzione, al fine di semplificarne la notazione, si userà indicare con t l'iterazione $n + j/\kappa$ e con $t+1$ la successiva $n + (j+1)/\kappa$. Il metodo più semplice per uniformare l'aggiornamento dei parametri è quello di dividere ogni componente del gradiente per la norma euclidea della sua media. Risulta preferibile utilizzare una media mobile pesata esponenzialmente così da tenere sufficientemente conto dell'evoluzione temporale del modulo della derivata che, nel caso in cui essa cali bruscamente, indica la vicinanza ad un minimo. Si definisce quindi il momentum del quadrato del gradiente come

$$\begin{aligned} \boldsymbol{v}_{t+1} &= \beta \boldsymbol{v}_t + (1-\beta) \left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right)\right)^2 \\ &= (1-\beta) \sum_{i=0}^{t} (\beta)^{t-i} \left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_i\right)\right)^2 \end{aligned} \tag{4.32}$$

in cui $\beta < 1$ è il *forgetting factor* (detto anche *decay factor*), cioè il termine moltiplicativo che determina di quanto l'algoritmo deve "dimenticare" i valori assunti da $\boldsymbol{v}$ nelle iterazioni precedenti. Grazie alla particolare scelta dei fattori di ogni addendo, il momentum è proprio una media mobile esponenziale calcolata sul processo stocastico $\left\{\left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right)\right)^2\right\}_{t\in\mathbb{N}}$. Noto $\boldsymbol{v}$ al tempo $(t+1)$ è dunque possibile eseguire l'aggiornamento ai parametri nel seguente modo.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\eta}{\sqrt{\hat{\boldsymbol{v}}_{t+1}} + \boldsymbol{\varepsilon}} \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right) \tag{4.33}$$

L'applicazione della radice quadrata e del rapporto sono da considerarsi elemento per elemento (*element-wise*). Il vettore $\boldsymbol{\varepsilon}$ è definito avere tutte componenti uguali e pari ad un valore molto basso (ad esempio $\boldsymbol{\varepsilon}_i = 10^{-8}$ per ogni i) allo scopo di impedire che il denominatore diventi nullo. Riscrivendo (4.33) per una generica componente i di $\boldsymbol{\theta}_t$ risulta che $\sqrt{\hat{v}_i}$ è la radice quadrata della media (RMS, *root mean squared*) esponenziale del quadrato dei gradienti, la norma euclidea appunto. Applicata sulla derivata parziale calcolata in t, questa ne scala il valore in un intorno di 1 dipendentemente da qual è stata la dispersione registrata nella serie storica.

$$\theta_{t+1,i} = \theta_{t,i} - \eta \underbrace{\frac{\frac{\partial L(\boldsymbol{\theta}_t)}{\partial \theta_{t,i}}}{\sqrt{\hat{v}_{t+1,i}} + \varepsilon_i}}_{\substack{\text{aggiornamento} \\ \text{scalato}}}$$

L'algoritmo di RMSProp si dice avere un grado di apprendimento adattivo in quanto, rispetto a (3.16), la derivata parziale è moltiplicata per un fattore fisso η e uno che, per mezzo del momentum,

dipende dalle caratteristiche di convergenza mostrate dal parametro.

$$\theta_{t+1,i} = \theta_{t,i} - \underbrace{\frac{\eta}{\sqrt{\hat{v}_{t+1,i}} + \varepsilon_i}}_{\substack{\text{learning rate} \\ \text{adattivo}}} \frac{\partial L\left(\boldsymbol{\theta}_t\right)}{\partial \theta_{t,i}}$$

Fino ad ora si è vista una soluzione per accelerare la velocità di convergenza dello SGD e una per adattare l'apprendimento ad ogni singolo parametro. L'algoritmo che le fonde prende il nome di Adam (*ADAptive Momentum estimation* (Kingma e Ba, 2014)) e, proprio perché assolve alle principali esigenze di apprendimento delle reti neurali profonde, è attualmente la tecnica di ricerca del minimo più diffusa nel campo del deep learning. Questo algoritmo introduce un momentum sia per il momento primo che per il momento secondo di $\nabla_{\boldsymbol{\theta}} L$ che sono indicati rispettivamente con

$$\begin{aligned}
\boldsymbol{m}_{t+1} &= \beta_1 \boldsymbol{m}_t + \left(1 - \beta_1\right) \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right) \\
&= \left(1 - \beta_1\right) \sum_{i=0}^{t} \left(\beta_1\right)^{t-i} \nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_i\right)
\end{aligned}$$

e

$$\begin{aligned}
\boldsymbol{v}_{t+1} &= \beta_2 \boldsymbol{v}_t + \left(1 - \beta_2\right) \left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right)\right)^2 \\
&= \left(1 - \beta_2\right) \sum_{i=0}^{t} \left(\beta_2\right)^{t-i} \left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_i\right)\right)^2 .
\end{aligned} \tag{4.34}$$

Entrambe le variabili sono, però stimatori distorti delle relative quantità. Ad esempio, facendo uso della risoluzione della serie geometrica troncata, il valore atteso di $\boldsymbol{v}_{t+1}$ vale

$$\begin{aligned}
\mathbb{E}\left[\boldsymbol{v}_{t+1}\right] &= \mathbb{E}\left[\left(1 - \beta_2\right) \sum_{i=0}^{t} \left(\beta_2\right)^{t-i} \left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_i\right)\right)^2\right] \\
&= \left(1 - \beta_2\right) \sum_{i=0}^{t} \left(\beta_2\right)^{t-i} \mathbb{E}\left[\left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_i\right)\right)^2\right] \\
&= \mathbb{E}\left[\left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right)\right)^2\right] \left(1 - \beta_2\right) \left(\beta_2\right)^t \left[\sum_{i=0}^{t} \left(\beta_2^{-1}\right)^i\right] + \xi \\
&= \mathbb{E}\left[\left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right)\right)^2\right] \left(1 - \beta_2\right) \left(\beta_2\right)^t \frac{1 - \beta_2^{-(t+1)}}{1 - \beta_2^{-1}} + \xi \\
&= \mathbb{E}\left[\left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right)\right)^2\right] \left(1 - \left(\beta_2\right)^{t+1}\right) + \xi
\end{aligned} \tag{4.35}$$

dove ξ è nullo se il processo stocastico è stazionario, cioè non modifica la sua distribuzione di probabilità nel tempo. Questa condizione permette infatti di considerare le variabili aleatorie del processo stocastico $\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_i\right)$ e $\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right)$ come uguali in distribuzione. Risulta quindi che ξ è esprimibile come

$$\xi \propto \sum_{i=0}^{t} \left(\beta_1\right)^{t-i} \left[\left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_i\right)\right)^2 - \left(\nabla_{\boldsymbol{\theta}} L\left(\boldsymbol{\theta}_t\right)\right)^2\right] \tag{4.36}$$

e che la sua approssimazione a 0 è tanto più valida quanto è più piccolo $\beta_2 < 1$. In questo modo, infatti, gli errori che pesano di più ai fini del valore ultimo di ξ sono quelli che coinvolgono i gradienti agli istanti i più prossimi a t che sono ragionevolmente più "simili" loro. Si più procedere analogamente per $\boldsymbol{m}_t$. La distorsione è più evidente durante le prime iterazioni in cui la media mobile è fortemente influenzata dalla scelta del valore iniziale (che è ovviamente posto a zero). Conviene quindi definire i seguenti stimatori.

$$\hat{\boldsymbol{m}} = \frac{\boldsymbol{m}_{t+1}}{1 - (\beta_1)^{t+1}}$$
$$\hat{\boldsymbol{v}} = \frac{\boldsymbol{v}_{t+1}}{1 - (\beta_2)^{t+1}}$$

Questi, componendo quanto già trattato in (4.31) e (4.33), possono essere utilizzati per operare un aggiornamento dei parametri con tasso adattivo e con limitati movimenti su direzioni diverse da quella che collega $\boldsymbol{\theta}_0$ e $\hat{\boldsymbol{\theta}}$.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \eta \frac{\hat{\boldsymbol{m}}}{\sqrt{\hat{\boldsymbol{v}}} + \boldsymbol{\varepsilon}} \tag{4.37}$$

I valori $\beta_1 = 0.9$, $\beta_2 = 0.999$ e $\boldsymbol{\varepsilon}_i = 10^{-8}$, già proposti nella pubblicazione originale dell'algoritmo (Kingma e Ba, 2014), sono tutt'ora utilizzati di default.

Bibliografia

Rosenblatt, Frank (1958). «The perceptron: a probabilistic model for information storage and organization in the brain». In: *Psychological review* 65.6, p. 386.

Minsky, Marvin e Seymour A Papert (1969). *Perceptrons: An introduction to computational geometry*. MIT press.

Rumelhart, D. E., G. E. Hinton e R. J. Williams (1986). «Parallel distributed processing: explorations in the microstructure of cognition, Vol. 1». In: a cura di David E. Rumelhart, James L. McClelland e CORPORATE PDP Research Group. Cambridge, MA, USA: MIT Press. Cap. Learning internal representations by error propagation, pp. 318–362. ISBN: 0-262-68053-X.

Wejchert, Jakub e Gerald Tesauro (1990). «Neural network visualization». In: *Advances in neural information processing systems*, pp. 465–472.

Hornik, Kurt (1991). «Approximation capabilities of multilayer feedforward networks». In: *Neural networks* 4.2, pp. 251–257.

Vapnik, Vladimir (1992). «Principles of risk minimization for learning theory». In: *Advances in neural information processing systems*, pp. 831–838.

Mack, Thomas (1993). «Distribution-free calculation of the standard error of chain ladder reserve estimates». In: *ASTIN Bulletin: The Journal of the IAA* 23.2, pp. 213–225.

Bengio, Yoshua, Patrice Simard e Paolo Frasconi (1994). «Learning long-term dependencies with gradient descent is difficult». In: *IEEE transactions on neural networks* 5.2, pp. 157–166.

Jørgensen, Bent e Marta C Paes De Souza (1994). «Fitting Tweedie's compound Poisson model to insurance claims data». In: *Scandinavian Actuarial Journal* 1994.1, pp. 69–93.

Whitley, Darrell (1994). «A genetic algorithm tutorial». In: *Statistics and computing* 4.2, pp. 65–85.

Hochreiter, Sepp e Jürgen Schmidhuber (1997). «Long short-term memory». In: *Neural computation* 9.8, pp. 1735–1780.

Schuster, Mike e Kuldip K Paliwal (1997). «Bidirectional recurrent neural networks». In: *IEEE Transactions on Signal Processing* 45.11, pp. 2673–2681.

Francis, Louise (2001). «Neural networks demystified». In: *Casualty Actuarial Society Forum*.

England, Peter D e Richard J Verrall (2002). «Stochastic claims reserving in general insurance». In: *British Actuarial Journal* 8.3, pp. 443–518.

Bengio, Yoshua, Réjean Ducharme et al. (2003). «A neural probabilistic language model». In: *Journal of machine learning research* 3.Feb, pp. 1137–1155.

Dugas, Charles et al. (2003). «Statistical learning algorithms applied to automobile insurance ratemaking». In: *Intelligent And Other Computational Techniques In Insurance: Theory and Applications*. World Scientific, pp. 137–197.

Merz, Michael e Mario V. Wüthrich (2008). «Modelling The Claims Development Result For Solvency Purposes». In:

Dalkilic, Turkan Erbay, Fatih Tank e Kamile Sanli Kula (2009). «Neural networks approach for determining total claim amounts in insurance». In: *Insurance: Mathematics and Economics* 45.2, pp. 236–241.

Raina, R., A. Madhavanand e A. Y. Ng (2009). «Large-scale Deep Unsupervised Learning using graphics processors». In: *Proceedings of the 26th annual international conference on machine learning*, pp. 873–880.

Gigante, Patrizia, Liviana Picech e Luciano Sigalotti (2010). *La tariffazione nei rami danni con modelli lineari generalizzati*. EUT Edizioni Università di Trieste.

Duchi, John, Elad Hazan e Yoram Singer (2011). «Adaptive subgradient methods for online learning and stochastic optimization». In: *Journal of Machine Learning Research* 12.Jul, pp. 2121–2159.

Hinton, Geoffrey, Nitish Srivastava e Kevin Swersky (2012). «Neural networks for machine learning lecture 6a overview of mini-batch gradient descent». In:

Mano, Cristina e Elena Rasa (2012). «A Discussion of Modeling Techniques for Personal Lines Pricing». In: *Trans 27th ICA*.

Murphy, Kevin P. (2012). *Machine Learning: a probabilistic perspective*. 1ª ed. Adaptive computation and machine learning. MIT Press. ISBN: 978-0-262-01802-9.

Storkey, Amos (2013). «When training and test sets are different: characterizing learning transfer». In:

Cho, Kyunghyun et al. (2014). «Learning phrase representations using RNN encoder-decoder for statistical machine translation». In: *arXiv preprint arXiv:1406.1078*.

Kingma, Diederik P e Jimmy Ba (2014). «Adam: a method for stochastic optimization». In: *arXiv preprint arXiv:1412.6980*.

Srivastava, Nitish et al. (2014). «Dropout: a simple way to prevent neural networks from overfitting». In: *The Journal of Machine Learning Research* 15.1, pp. 1929–1958.

Aggarwal, Charu C. (2015). *Data Mining: the textbook*. Springer. ISBN: 978-3-319-14141-1.

Baur, C. e D. Wee (2015). *Manufacturing's next act*. McKinsey & Company. URL: `https://www.mckinsey.com/business-functions/operations/our-insights/manufacturings-next-act` (visitato il 04/10/2018).

Meyers, Glenn (2015). «Stochastic loss reserving using Bayesian MCMC models». In:

Olah, Christopher (2015). *Understanding LSTM Networks*. Colah's Blog. URL: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` (visitato il 18/11/2018).

Cao, Xi Hang, Ivan Stojkovic e Zoran Obradovic (2016). «A robust data scaling algorithm to improve classification accuracies in biomedical data». In: *BMC bioinformatics* 17.1, p. 359.

Evans, R. e J. Gao (2016). *DeepMind AI reduces Google data centre cooling bill by 40%*. DeepMind e Google. URL: `https://deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/` (visitato il 01/10/2018).

Goodfellow, Ian, Yoshua Bengio e Aaron Courville (2016). *Deep learning*. Vol. 1.

Guo, Cheng e Felix Berkhahn (2016). «Entity embeddings of categorical variables». In: *arXiv preprint arXiv:1604.06737*.

Hassabis, Demis (2016). *What we learned in Seoul with AlphaGo*. Google. URL: `https://blog.google/technology/ai/what-we-learned-in-seoul-with-alphago/` (visitato il 01/10/2018).

IBM (2016). *IBM at 100 - Deep Blue*. URL: http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/deepblue/ (visitato il 01/10/2018).

Jouppi, Norm (2016). *Google supercharges Machine Learning taskswith TPU custom chip*. Google. URL: https://cloudplatform.googleblog.com/2016/05/Google-supercharges-machine-learning-tasks-with-custom-chip.html (visitato il 06/10/2018).

McCormick, Chris (2016). *Word2Vec Tutorial - The Skip-Gram Model*. URL: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/ (visitato il 01/02/2019).

Ng, Andrew Y. (2016). *Nuts and bolts of applying Deep Learning*. Deep Learning School, 24-25 Settembre, Stanford. YouTube. URL: https://www.youtube.com/watch?v=F1ka6a13S9I (visitato il 12/10/2018).

Alcorn, Tom (2017). *Seeing like a Perceptron*. URL: https://tdb-alcorn.github.io/2017/12/17/seeing-like-a-perceptron.html (visitato il 05/11/2018).

Capizzi, Sirio e Andrea Neroni (2017). «Neural Networks in motor insurance». In:

Rupp, Karl (2017). *Transistors per microprocessor - Rupp & Horowitz*. URL: https://github.com/owid/owid-datasets (visitato il 01/02/2019).

Styrud, Lovisa (2017). «Risk premium prediction of car damage insurance using Artificial Neural Networks and Generalized Linear Models». In:

Ania (2018a). *L'assicurazione italiana 2017-2018*. Ania. URL: http://www.ania.it/it/pubblicazioni/ (visitato il 04/02/2019).

— (2018b). *Premi del lavoro diretto italiano 2017*. Ania. URL: http://www.ania.it/export/sites/default/it/pubblicazioni/rapporti-annuali/Volumi-Premi-lavoro-diretto-italiano/2017/PREMI-2017-x-WEB.pdf (visitato il 05/02/2019).

— (2018c). *Trends - Focus R.C. Auto*. Ania. URL: http://www.ania.it/export/sites/default/it/pubblicazioni/COLLANE-PERIODICHE/ANIA-Trends/ANIA-Trends-Focus-RC-Auto/2017/ANIA-TRENDS-RC-Auto-Dati-al-31-dicembre-2017.pdf (visitato il 04/02/2019).

IVASS (2018). *Bollettino Statistico Anno V - N. 17 - Dicembre 2018*. Ramo r.c. auto: dati tecnici 2017. IVASS. URL: https://www.ivass.it/pubblicazioni-e-statistiche/statistiche/bollettino-statistico/2018/n17/Bollettino_Statistico_Dati_Tecnici_RCA_2017.pdf (visitato il 05/02/2019).

Kuo, Kevin (2018). «DeepTriangle: a Deep Learning approach to loss reserving». In: *arXiv preprint arXiv:1804.09253*.

Richman, Ronald (2018). «AI in actuarial science». In: Presentato in occasione della Actuarial Society Convention del Sud Africa tenutasi 24-25 ottobre a Città del Capo.

Singh, Seema (2018). *Cousins of Artificial Intelligence*. URL: https://towardsdatascience.com/cousins-of-artificial-intelligence-dda4edc27b55 (visitato il 01/02/2019).

Wüthrich, Mario V. (2018a). «Machine learning in individual claims reserving». In: *Scandinavian Actuarial Journal*, pp. 1–16.

— (2018b). «Neural Networks applied to Chain-Ladder reserving». In:

IVASS (2019). *Bollettino Statistico - Anno VI - N. 1 - Febbraio 2019*. IPER: L'andamento dei prezzi effettivi per la garanzia RC Auto nel terzo trimestre 2018. IVASS. URL: https://www.ivass.it/pubblicazioni-e-statistiche/statistiche/bollettino-statistico/2019/n1/Bollettino_IPER_2018_3.pdf (visitato il 05/02/2019).