Università degli studi di Padova
Dipartimento di Scienze Statistiche
Corso di Laurea Magistrale in

Scienze Statistiche

Relazione Finale

Proprietà statistiche di modelli per il deep learning

Relatore Prof. Bruno Scarpa
Dipartimento di Scienze Statistiche

Laureando Alessandro Aere
Matricola N 1089668

Anno Accademico 2016/2017

A mamma e papà

Sommario

Lo scopo di questa tesi è quello di analizzare le proprietà statistiche di modelli per il *deep learning* supervisionato, in particolare delle reti neurali multi-strato (o *deep neural networks*). Per prima cosa, sono studiati i punti di forza di questa classe di modelli, tra cui la capacità di approssimare una qualsiasi funzione matematica e di adattarsi ai dati con molta flessibilità. Per quest'ultimo aspetto, sono stati effettuati alcuni studi di simulazione, che mettono a confronto le reti neurali a singolo strato con le reti neurali multi-strato. Inoltre, viene proposta un'analisi per trovare le proprietà statistiche che la rete neurale ha in comune con il modello MARS. Sono studiati, in seguito, alcuni metodi di regolarizzazione, utili per evitare il sovradattamento ai dati di stima. I metodi principalmente utilizzati sono la penalità *weight decay* ed il *dropout*. È dimostrato come l'applicazione di questi metodi possa essere considerato, dal punto di vista statistico, come il risultato di un'inferenza di tipo bayesiano. Infine, sono state analizzate, attraverso due casi di studio, alcune situazioni in cui il *deep learning* ottiene generalmente risultati ottimi: il primo caso di studio è l'analisi di *big data*, in cui viene messa a confronto la capacità previsiva delle reti neurali multi-strato con quella di altri modelli statistici; il secondo caso di studio è la classificazione di immagini, in cui sono state applicate le *convolutional neural networks*. Infine, sono trattati gli aspetti computazionali: viene proposto un confronto tra l'utilizzo dell'architettura GPU con quella CPU, ed un confronto tra l'utilizzo di `Python` ed `R`.

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

"Non vince il più forte,
chi vince è il più forte."

L'irrefrenabile sviluppo tecnologico ha sicuramente caratterizzato gli ultimi decenni. L'utilizzo della tecnologia ha avuto un considerevole impatto sulla società moderna, entrando a far parte della nostra vita quotidiana e cambiando completamente le nostre abitudini. Il progresso tecnologico ha portato, inoltre, allo sviluppo di alcune *intelligenze artificiali*, ossia macchine che sono in grado di prendere autonomamente delle decisioni, sulla base delle conoscenze assunte.

Il *machine learning*, una branca in fortissima espansione nell'ambito dell'intelligenza artificiale, permette alle macchine di svolgere alcuni compiti, senza programmare esplicitamente il modo in cui farlo. Il *machine learning* utilizza una gamma di metodi, grazie ai quali è possibile estrarre alcune informazioni ricorrenti da un insieme di dati. La macchina è in grado di svolgere autonomamente il proprio compito, grazie all'estrazione di queste informazioni.

Recentemente, è stato studiato il modo in cui far svolgere alle macchine compiti legati all'ambito percettivo, e si è notato che i classici metodi per il *machine learning* non erano adeguati alla complessità del compito. Ad esempio, risultava molto difficile identificare l'oggetto contenuto in un'immagine: l'immagine, infatti, è composta da un insieme di *pixel*, da cui è molto difficile, per il modello statistico, risalire all'oggetto rappresentato. Si pensi che l'oggetto potrebbe anche trovarsi in un angolo, potrebbe mostrare colori differenti a seconda se è in luce o all'ombra, o potrebbe essere rappresentato da diverse angolazioni. Questi compiti risultano molto facili per l'essere uma-

no, ma impossibili per una macchina. Il *deep learning* fornì una soluzione a questo problema.

Il *deep learning* permette alla macchina di estrarre informazioni molto complesse da un insieme di dati, estraendo prima alcune informazioni semplici, per poi combinarle. In altre parole, viene raggiunta l'informazione finale elaborando diversi livelli di rappresentazione dei dati, corrispondenti a gerarchie di caratteristiche, fattori o concetti. Ad esempio, per identificare una persona in un'immagine, vengono prima estratte dai dati alcune informazioni semplici, come quelle riguardanti il contorno, il colore e la posizione. In seguito, queste informazioni vengono combinate per identificare la persona.

Al giorno d'oggi, grazie al *deep learning*, si è in grado di comprendere il contenuto di immagini, messaggi vocali e testi, in modo automatico. Inoltre, è stato utilizzato anche per prevedere l'effetto di farmaci, analizzare i dati dell'acceleratore di particelle, ricostruire circuiti cerebrali e prevedere gli effetti di mutazioni nel DNA non codificato sull'espressione genica e sulle malattie.

Il modello più utilizzato per il *deep learning* è la rete neurale multi-strato, o *deep neural network*. Questo modello riceve in input i dati, elabora l'informazione attraverso alcuni strati latenti e fornisce in output il risultato. Nonostante sia nata nel contesto del *machine learning*, una rete neurale multi-strato è un vero e proprio modello statistico. La rete neurale a singolo strato latente ebbe le sue prime applicazioni già nei primi anni $'60$ e negli anni successivi questo modello venne analizzato dal punto di vista statistico da molti autori, come ad esempio Ripley (1996) e Hastie et al. (2009). Le reti neurali multi-strato, al contrario, si diffusero solamente a partire dal 2010, quando lo sviluppo tecnologico lo permise. Quindi, questo modello innovativo, essendo di recente introduzione, non ha ancora trovato spazio nella letteratura statistica. Lo scopo di questa tesi è quello di studiare le reti neurali multi-strato, analizzando a fondo le loro proprietà statistiche.

Nel primo capitolo è descritta la struttura della rete neurale multi-strato, ponendo l'attenzione sulla procedura utilizzata per effettuare la stima dei parametri. Inoltre, vengono presentati, in modo approfondito, gli aspetti innovativi che caratterizzano questo modello.

Nel secondo capitolo sono studiati i punti di forza della rete neurale multi-

strato ed i suoi miglioramenti rispetto a quella a singolo strato, in termini di flessibilità e capacità approssimativa. Inoltre, al fine di analizzarne le proprietà, sono proposti alcuni confronti statistici tra la rete neurale ed altri modelli, quali la *projection pursuit regression* (PPR) ed il *multivariate adaptive regression splines* (MARS).

Nel terzo capitolo si cerca di stabilire il motivo per cui l'utilizzo degli strati multipli fornisca risultati previsivi molto soddisfacenti, nonostante abbiano un elevatissimo numero di parametri. Vengono, quindi, analizzate le proprietà dei metodi di regolarizzazione, e le loro capacità di risolvere il problema dell'*overfitting*. Inoltre, i risultati ottenuti dal *weight decay* ed dal *dropout* vengono reinterpretati come un'inferenza statistica di tipo bayesiano.

Infine, nel quarto capitolo sono presenti due casi di studio, con cui si valuta empiricamente la capacità previsiva delle reti neurali multi-strato, con particolari tipologie di dati. In particolare, vengono introdotte le *convolutional neural networks*, una classe di reti neurali particolarmente efficace nel classificare le immagini.

Capitolo 1

Un modello per il *deep learning*

Il *deep learning* è una branca del *machine learning*, che permette alle macchine di estrarre informazioni molto complesse da un insieme di dati, rendendo possibile lo svolgimento di compiti molto complicati, come quelli legati all'ambito percettivo. I modelli per il *deep learning* hanno la caratteristica di essere costituiti da diversi strati di elaborazione, ognuno dei quali estrae una rappresentazione dello strato precedente.

Nell'ambito del *deep learning* supervisionato, la classe di modelli più utilizzata è la rete neurale multi-strato, o *deep neural network* (DNN). Come viene suggerito dal nome stesso, questo è un tipo di modello costruito a rete, i cui principali componenti sono i *nodi*, detti anche neuroni. Esistono diverse classi di reti neurali, a seconda della tipologia dei nodi, e di come questi sono connessi fra loro. In questa tesi, si è deciso di approfondire la classe di reti neurali, sulla cui base si sono poi sviluppate tutte le tipologie di reti che negli ultimi anni stanno spopolando nel campo del *deep learning*. Le reti in questione sono le *feed-forward neural networks* (FFNN), dette anche "back-propagation" neural networks, di cui viene data una definizione formale in Ripley (1996, glossario):

Definizione 1 *La feed-forward neural network è una rete in cui i vertici possono essere numerati, cosicché tutte le connessioni vadano da un vertice ad un altro di numero maggiore. In pratica, i vertici sono strutturati in strati, con connessioni solamente verso strati più alti.*

Questa definizione fornisce precise indicazioni riguardo la struttura della rete. In particolare, viene posta l'attenzione su come i nodi siano raggruppati in *strati*, i quali vanno a formare una struttura gerarchica: lo strato più basso contiene i nodi di *input*, mentre quello più alto contiene i nodi di *output*. Tutti gli strati posti all'interno, invece, sono denominati *strati latenti*; si veda la Figura 1.1.

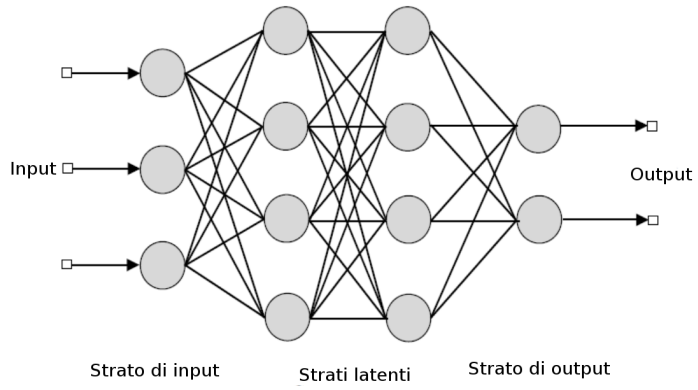

Figura 1.1: La struttura della *feed-forward neural network* : in questo caso la rete è composta da uno strato di input, due strati latenti ed uno strato di output.

Le connessioni sono unidirezionali e sono rappresentate da archi, ai quali è permesso il collegamento solamente tra nodi di uno strato e quelli dello strato successivo. Ad ogni arco è associato un *parametro*, che nel gergo del *machine learning* viene chiamato peso.

Il motivo per cui queste reti vengono definite "neurali" ha un fondamento biologico. Infatti sono state sviluppate inizialmente con lo scopo di modellare le connessioni tra i neuroni del cervello umano. Gli archi rappresentavano le *sinapsi*, ossia impulsi nervosi che si trasmettono da un neurone all'altro. Lo scopo di questi modelli era quello di individuare quali neuroni venivano attraversati da un segnale sufficientemente intenso. Venivano, quindi, omessi i neuroni, il cui segnale era inferiore ad una certa soglia. Una spiegazione più estesa è fornita in Ripley (1996, sez. 5.1).

Questo capitolo fornisce una descrizione approfondita della struttura di una rete neurale multi-strato e di come i suoi parametri vengano stimati, nel momento in cui è utilizzata come modello statistico.

La struttura di una rete neurale multistrato

Oltre allo scopo di fornire una dettagliata descrizione riguardo alla struttura di una rete neurale multi-strato, questo paragrafo mira a stabilire una notazione di base, a cui si farà riferimento in tutto il documento. La struttura e la notazione descritte in seguito si basano sull'analisi di una rete neurale multi-strato, contenuta in Efron e Hastie (2016, cap. 18).

Per semplicità, viene presentata, in prima battuta, una notazione di tipo univariato sulla relazione tra gli strati della rete, elaborando solo in un secondo momento una notazione vettoriale. Si definisca quindi:

- L: numero di strati della rete, composti da uno strato di input, uno strato di output e $L-2$ strati latenti;
- p_1: numero di nodi di input;
- p_l: numero di nodi presenti nello strato l-esimo;
- x_i: valore del nodo di input i-esimo;
- $a_j^{(l)}$: valore del nodo j-esimo dello strato l-esimo;
- $w_{ij}^{(l)}$: coefficiente associato all'arco che collega il nodo i-esimo dello strato l-esimo con il nodo j-esimo dello strato $(l+1)$-esimo;
- y_k: valore del nodo di output k-esimo.

La relazione tra lo strato di input ed il primo strato latente è

$$z_j^{(2)} = w_{0j}^{(1)} + \sum_{i=1}^{p_1} w_{ij}^{(1)} x_i,$$

$$a_j^{(2)} = g^{(2)}(z_j^{(2)}).$$

Si noti come il nodo j-esimo del primo strato latente assuma un valore pari a $g^{(2)}(z_j^{(2)})$, dove $g^{(2)}(\cdot)$ è una funzione non lineare, detta *funzione di attivazione*, mentre $z_j^{(2)}$ è la combinazione lineare dei nodi di input e i parametri $w_{\cdot j}^{(1)}$. A questa combinazione lineare viene sommato il termine $w_{0j}^{(l)}$, ovvero il parametro associato all'arco che collega un nodo costante e pari a 1 con il nodo j-esimo dello strato $(l+1)$-esimo. Questa quantità funge da intercetta nella combinazione lineare, e viene introdotta per modellare una eventuale distorsione. In generale, la relazione tra lo strato $(l-1)$-esimo e lo strato l-esimo è definita come:

$$z_j^{(l)} = w_{0j}^{(l-1)} + \sum_{i=1}^{p_{l-1}} w_{ij}^{(l-1)} a_i^{(l-1)},$$

$$a_j^{(l)} = g^{(l)}(z_j^{(l)}). \tag{1.1}$$

La funzione di attivazione $g^{(l)}(\cdot)$ è specifica per lo strato l-esimo, nonostante venga spesso usata, per l'intera rete, un'unica funzione di attivazione $g(\cdot)$ comune in tutti gli strati. Una descrizione delle funzioni di attivazione più utilizzate verrà fornita nel paragrafo 1.4. Infine, lo strato di output viene prodotto attraverso la relazione

$$z_k^{(L)} = w_{0k}^{(L-1)} + \sum_{i=1}^{p_{L-1}} w_{ik}^{(L-1)} a_i^{(L-1)},$$

$$y_k = g^{(L)}(z_k^{(L)}).$$

Sia il numero di nodi di output K, che la funzione di trasformazione $g^{(L)}(\cdot)$ dipendono dal problema in questione. Per un problema di regressione univariato, si ha tipicamente un solo nodo di output, quindi $K = 1$, mentre, una opportuna scelta della funzione di trasformazione è la funzione identità, $g^{(L)}(z_k^{(L)}) = z_k^{(L)}$. Per un problema di classificazione, il numero di nodi K coincide con il numero di classi della variabile risposta, che si vuole modellare. Ogni nodo k indica la probabilità di appartenenza alla k-esima classe. Come funzione di trasformazione, spesso conviene utilizzare la funzione logistica multinomiale,

$$g^{(L)}(z_k^{(L)}) = \frac{e^{z_k^{(L)}}}{\sum_{j=1}^{K} e^{z_j^{(L)}}}$$

che nel gergo del *machine learning* viene chiamata funzione *softmax*. Si ponga ora:

- $\mathbf{a}^{(1)} = \mathbf{x} = [1\ x_1 \cdots x_{p_1}]^T$;
- $\mathbf{a}^{(l)} = [1\ a_1^{(l)} \cdots a_{p_l}^{(l)}]^T$;
- $\mathbf{w}_j^{(l)} = [w_{0j}^{(l)}\ w_{1j}^{(l)} \cdots w_{p_l j}^{(l)}]^T$;
- $W^{(l)} = [\mathbf{w}_0^{(l)}\ \mathbf{w}_1^{(l)} \cdots \mathbf{w}_{p_{l+1}}^{(l)}]^T$;
- $\mathbf{W} = [W^{(1)} W^{(2)} \cdots W^{(L)}]$;
- $\boldsymbol{y} = [y_1 \cdots y_K]^T$.

Diventa così più semplice ed intuitivo adottare la notazione vettoriale per formulare la relazione tra due generici strati della rete:

$$\mathbf{z}^{(l)} = W^{(l-1)} \mathbf{a}^{(l-1)}, \tag{1.2}$$

$$\mathbf{a}^{(l)} = g^{(l)}(\mathbf{z}^{(l)}), \tag{1.3}$$

dove la funzione $g^{(l)}(\cdot)$ viene applicata elemento per elemento al vettore $\mathbf{z}^{(l)}$. Di conseguenza, la completa relazione che lega il vettore di input $\mathbf{x}$ con quello di output $\mathbf{y}$ risulta essere

$$\mathbf{y} = f(\mathbf{x}; \mathbf{W}) = g^{(L)}(W^{(L-1)} g^{(L-1)}(\cdots W^{(2)} g^{(2)}(W^{(1)} \mathbf{x}))). \tag{1.4}$$

La stima dei parametri via *backpropagation*

Per problemi di regressione, si ha generalmente una variabile risposta quantitativa $y = (y_1, ..., y_n) \in \mathbb{R}^n$, mentre per problemi di classificazione si ha una variabile risposta qualitativa $y = (y_1, ..., y_n) \in T(y)^n = \{t_1, ..., t_K\}^n$, dove $T(y)$ è l'insieme di modalità che può assumere y. Si consideri un insieme

di dati, composto da n osservazioni, per ognuna delle quali vengono rilevate p variabili esplicative, $x_i = (x_{i1}, ..., x_{ip}) \in \mathbb{R}^p$. L'interesse è quello di adattare all'insieme di dati una rete neurale, in modo da minimizzare una certa funzione di perdita $L[y, f(x; \mathbf{W})]$. Per farlo è necessario trovare quei valori dei parametri $\hat{\mathbf{W}}$, tali per cui

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \left\{ \frac{1}{n} \sum_{i=1}^{n} L[y_i, f(x_i; \mathbf{W})] \right\}. \tag{1.5}$$

Per problemi di regressione, le scelte più comuni per la funzione di perdita sono:

- l'**errore quadratico medio**, $\text{MSE}(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i; \mathbf{W}))^2$;
- la **radice dell'MSE**, $\text{rMSE}(\mathbf{W}) = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (y_i - f(x_i; \mathbf{W}))^2}$;
- l'**errore assoluto medio**, $\text{MAE}(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^{n} |y_i - f(x_i; \mathbf{W})|$.

Al contrario, per problemi di classificazione, le funzioni di perdita più usate sono:

- il **tasso di errata classificazione**, $\text{R}(\mathbf{W}) = \frac{1}{n} \sum_{i=1}^{n} I(y_i \neq f(x_i; \mathbf{W}))$;
- la **cross-entropia**, che corrisponde all'opposto della log-verosimiglianza della distribuzione multinomiale, $\text{H}(\mathbf{W}) = -\sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log f_k(x_i; \mathbf{W})$, dove $y_{ik} = 1$ se $y_i = t_k$, e 0 altrimenti; minimizzare la cross-entropia, quindi, corrisponde a massimizzare la log-verosimiglianza (Hastie et al. 2009, sez. 2.6.3).

L'algoritmo largamente più utilizzato per stimare le reti neurali, sia a strato singolo che multi-strato, è la *backpropagation* (ad esempio si veda Williams e Hinton (1986)). Questo algoritmo ha la capacità di risolvere l'equazione (1.5), con un basso costo computazionale. La maggior parte degli algoritmi di ottimizzazione numerica sono iterativi e necessitano del calcolo del gradiente di primo e secondo ordine della funzione di perdita, rispetto ai parametri. Molto spesso però, una rete neurale multi-strato ha un numero di parametri elevatissimo. Con L strati, la rete possiede L matrici di parametri

$W^{(l)}$, ognuna delle quali contiene $p_l \times p_{l+1}$ coefficienti, dove il numero di nodi p_l può raggiungere qualche migliaio. Per ogni iterazione dell'algoritmo, il calcolo del gradiente primo richiede un numero di operazioni pari al numero di coefficienti, mentre le operazioni richieste per il calcolo del gradiente secondo crescono in modo quadratico all'aumentare del numero dei parametri. L'algoritmo di *backpropagation*, invece, oltre a non necessitare del calcolo del gradiente secondo, richiede il calcolo del gradiente primo solamente nell'ultimo strato, il quale viene poi *propagato all'indietro*, per ottenere il gradiente negli altri strati. L'algoritmo 1 percorre in modo dettagliato tutti i passi dell'algoritmo di *backpropagation*, per una generica osservazione (x_i, y_i), con $i = 1, ..., n$.

L'algoritmo alterna due passi in modo iterativo: con il *passo in avanti* si ottiene $\hat{f}(x_i; \mathbf{W})$ attraverso la (1.4), tenendo fisso $\mathbf{W}$, mentre con il *passo all'indietro* si ottengono i gradienti e vengono aggiornati i parametri. Nel gergo del *machine learning*, ogni iterazione viene chiamata *epoca*.

Nel passo in avanti viene calcolato il valore dei nodi $\mathbf{a}^{(l)}$ per ogni strato $l = 2, ..., L$, utilizzando i valori correnti di $\mathbf{W}$ (punto 1 dell'algoritmo 1). Attraverso la formula (1.4), è possibile ottenere tutti i valori dei nodi, $\mathbf{a}^{(l)}$, e delle combinazioni lineari, $\mathbf{z}^{(l)}$, salvando le quantità intermedie in itinere. È necessario quindi inizializzare i parametri con valori scelti casualmente, preferibilmente vicini a 0.

Dopodiché si sviluppa il passo all'indietro. Questo comprende una *fase di propagazione* (punti 2-4) e una *fase di aggiornamento* (punto 5). Lo scopo della fase di propagazione è quello di calcolare tutte le derivate parziali $\partial L[y_i, \hat{f}(x_i; \mathbf{W})]$, rispetto ai parametri. La logica consiste nel ricavare alcune quantità, denominate $\delta_L, ..., \delta_2$, utili per calcolare le derivate parziali in modo iterativo, alleggerendo così il costo computazionale. Per farlo, è necessario ricavare il generico δ_l come $\partial L[y_i, \hat{f}(x_i; \mathbf{W})]$ rispetto a $\mathbf{z}^{(l)}$.

Per ricavare δ_L nell'ultimo strato, ossia quello di output, è opportuno

applicare la "regola della catena"; δ_L si può quindi scrivere come

$$\begin{aligned}\delta^{(L)} &= \frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial \mathbf{z}^{(L)}} \\ &= \frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial \hat{f}(x_i; \mathbf{W})} \frac{\partial \hat{f}(x_i; \mathbf{W})}{\partial \mathbf{z}^{(L)}} \\ &= \frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial \hat{f}(x_i; \mathbf{W})} \circ \dot{g}^{(L)}(\mathbf{z}^{(L)}),\end{aligned}$$

dove $\dot{g}^{(L)}(\mathbf{z}^{(L)})$ indica la derivata prima di $g^{(L)}(\mathbf{z}^{(L)})$, ed è facilmente ottenibile derivando l'espressione (1.3) per $l = L$; il simbolo $\circ$ indica il prodotto di Hadamard (o prodotto elemento per elemento).

Servendosi di quest'ultimo risultato, è possibile scrivere la quantità $\delta^{(l)}$ del generico strato l come

$$\begin{aligned}\delta^{(l)} &= \frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial \mathbf{z}^{(l)}} \\ &= \frac{\partial L[y, \hat{f}(x_i; \mathbf{W})]}{\partial \mathbf{z}^{(l+1)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \\ &= \delta^{(l+1)} \frac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} \frac{\partial \mathbf{a}^{(l)}}{\partial \mathbf{z}^{(l)}} \\ &= \left(W^{(l)'} \delta^{(l+1)}\right) \circ \dot{g}^{(l)}(\mathbf{z}^{(l)}),\end{aligned}$$

dove $\dfrac{\partial \mathbf{z}^{(l+1)}}{\partial \mathbf{a}^{(l)}} = W^{(l)'}$ è il gradiente primo, rispetto ad $\mathbf{a}^{(l)}$, dell'espressione (1.2). Questa espressione corrisponde alla formula (1.7) dell'algoritmo 1, e viene chiamata *equazione di backpropagation.*

Avendo $\delta_2, ..., \delta_L$, è possibile ricavare le derivate

$$\begin{aligned}\frac{\partial L[y_i, f(x_i; \mathbf{W})]}{\partial W^{(l)}} &= \frac{\partial L[y_i, f(x_i; \mathbf{W})]}{\partial \mathbf{z}^{(l+1)}} \frac{\partial \mathbf{z}^{(l+1)}}{\partial W^{(l)}} \\ &= \delta^{(l+1)} \mathbf{a}^{(l)'},\end{aligned}$$

dove $\dfrac{\partial \mathbf{z}^{(l+1)}}{\partial W^{(l)}} = \mathbf{a}^{(l)'}$ è il gradiente primo, rispetto ad $W^{(l)}$, dell'espressione (1.2).

Nella *fase di aggiornamento* vengono modificati i valori dei parametri, per mezzo della *discesa del gradiente* (*gradient descent*), la quale si serve univocamente delle derivate parziali di primo ordine, calcolate nella fase di

propagazione. La discesa del gradiente è una tecnica di ottimizzazione numerica che permette di trovare il punto di minimo di una funzione, utilizzando solamente le derivate prime.

Dopodiché si ricomincia l'algoritmo con una nuova iterazione, utilizzando i nuovi valori per i parametri $\mathbf{W}$.

Algoritmo 1 BACKPROPAGATION

1. Calcolare il valore dei nodi $a^{(l)}$ per ogni strato $l = 2, ..., L$, utilizzando i valori correnti di $\mathbf{W}$;
2. Per lo strato di output $l = L$, calcolare

$$\delta^{(L)} = \frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial \hat{f}(x_i; \mathbf{W})} \circ \dot{g}^{(L)}(\mathbf{z}^{(L)}); \tag{1.6}$$

3. Per gli strati latenti $(l = L - 1, .., 2)$ ricavare

$$\delta^{(l)} = \left(W^{(l)'} \delta^{(l+1)}\right) \circ \dot{g}^{(l)}(\mathbf{z}^{(l)}); \tag{1.7}$$

4. Avendo $\delta_2, ..., \delta_L$, è possibile ricavare le derivate parziali con

$$\frac{\partial L[y_i, \hat{f}(x_i; \mathbf{W})]}{\partial W^{(l)}} = \delta^{(l+1)} \mathbf{a}^{(l)'}; \tag{1.8}$$

5. Aggiornare i parametri $\mathbf{W}$ usando la discesa del gradiente;
6. Ricominciare con una nuova iterazione dal passo 1, utilizzando i nuovi valori per i parametri $\mathbf{W}$.

Nel prossimo paragrafo sarà approfondita la fase di aggiornamento dei parametri, effettuata attraverso la discesa del gradiente, ossia ciò che accade al punto 5 dell'algoritmo 1.

Metodi innovativi per la discesa del gradiente

Il metodo più comune ed immediato per attuare l'aggiornamento dei parametri $W^{(l)}$ (punto 5 dell'algoritmo 1) è la discesa del gradiente, basata sulla *regola delta*, approfondita in dettaglio in Mitchell et al. (1997, sez. 4.4). In questo caso, l'aggiornamento dei parametri, al passo t, avviene secondo la

formula

$$W_{t+1}^{(l)} = W_t^{(l)} - \eta \cdot \Delta L(W_t^{(l)}; x, y), \qquad \text{per } l = 1, ..., L-1$$

dove $\Delta L(W_t^{(l)}; x, y)$ è il gradiente rispetto a $W_t^{(l)}$ dell'argomento dell'espressione (1.5), ossia il gradiente di $\frac{1}{n}\sum_{i=1}^{n} L[y_i, f(x_i; W)]$. La quantità $\Delta L(W_t^{(l)})$ corrisponde quindi a

$$\Delta L(W_t^{(l)}; x, y) = \frac{1}{n}\sum_{i=1}^{n} \frac{\partial L[y_i, f(x_i; W)]}{\partial W_t^{(l)}}. \tag{1.9}$$

Nella formula (1.9) si fa dipendere ΔL da $(W_t^{(l)}, x, y)$, per semplicità di scrittura. Si tenga a mente che il risultato dell'equazione (1.8) è il gradiente per una singola e generica osservazione (x_i, y_i).

In sostanza, se il gradiente è negativo, la funzione di perdita in quel punto è decrescente, il che significa che il parametro deve spostarsi verso valori più grandi per raggiungere un punto di minimo. Al contrario, se il gradiente è positivo, i parametri si spostano verso valori più piccoli per raggiungere valori inferiori della funzione di perdita. Infine, il parametro $\eta \in (0, 1]$ viene chiamato *learning rate*, e determina la grandezza dello spostamento.

Mini-batch gradient descent

Questo metodo, tuttavia, presenta diversi problemi e limitazioni quando viene applicato alle reti neurali multi-strato, i quali vengono discussi in Ruder (2016). L'utilizzo di tutti i dati per effettuare un solo passo di aggiornamento comporta costi computazionali notevoli e rallenta di molto la procedura di stima. Inoltre, non è possibile stimare il modello qualora il *dataset* sia troppo corposo e non possa essere caricato interamente in memoria. A tal proposito viene introdotta la tecnica del *mini-batch gradient descent*. Ciò consiste nel suddividere il *dataset* in sottocampioni di numerosità fissata $m \ll n$, dopo una permutazione casuale dell'intero insieme di dati. L'aggiornamento viene quindi attuato utilizzando ciascuno di questi sottoinsiemi, attraverso la formula

$$W_{t+1}^{(l)} = W_t^{(l)} - \eta \cdot \Delta L(W_t^{(l)}; x^{(i:i+m)}, y^{(i:i+m)}),$$

dove $(i : i + m)$ è l'indice per riferirsi al sottoinsieme di osservazioni che vanno dalla i-esima alla $(i + m)$-esima. Quindi, per ogni epoca, anziché fare un solo aggiornamento utilizzando tutti i dati, vengono effettuati tanti aggiornamenti quanti sono i *mini-batch*, usando per ogni aggiornamento solo quel determinato sottoinsieme di dati. I vantaggi di questa tecnica sono:

- basandosi solo su una parte delle osservazioni, l'algoritmo permette un'esplorazione più ampia dello spazio parametrico, con la maggiore possibilità di trovare nuovi e potenzialmente migliori punti di minimo;

- effettuare un passo dell'algoritmo è computazionalmente molto più rapido, il che garantisce una convergenza più veloce verso il punto di minimo;

- si possono calcolare le stime dei parametri anche caricando in memoria solamente una parte del *dataset* alla volta, permettendo l'applicazione di questo metodo anche a grandi insiemi di dati;

Rimangono, tuttavia, ancora alcuni problemi irrisolti. Il primo di questi è la scelta del valore del *learning rate*: impostare valori troppo piccoli può portare ad una convergenza molto lenta, mentre valori grandi possono far fluttuare i parametri attorno al minimo senza portare l'algoritmo a convergenza. Inoltre, trattare questa quantità con i classici metodi di regolarizzazione (come la convalida incrociata) può essere computazionalmente troppo dispendioso. Infine, sembra inopportuno pensare che tutti i parametri necessitino dello stesso valore di *learning rate* per convergere in modo ottimale.

Il secondo problema è l'alta propensione dell'algoritmo a rimanere intrappolato in minimi locali molto lontani dal minimo assoluto. Poiché i modelli trattati sono altamente parametrizzati, le funzioni di perdita discusse nel paragrafo 1.2 sono generalmente convesse in $f(x; \mathbf{W})$, ma non in $\mathbf{W}$. Questo significa che $L[y, f(x; \mathbf{W})]$ ha un unico punto di minimo per $f(x; \mathbf{W})$, che è ovviamente il minimo assoluto. Al contrario, $L[y, f(x; \mathbf{W})]$ possiede diversi punti di minimo locale per $\mathbf{W}$, di cui solamente uno è assoluto. Quindi risolvere l'equazione (1.5) e trovare il minimo assoluto per $\mathbf{W}$ risulta alquanto

complesso, a causa dell'alto rischio di ottenerne un locale. La problematica dei molteplici minimi locali è affrontata in Hastie et al. (2009).

Adaptive moment estimation (Adam)

Il tentativo di risolvere i problemi appena citati ha permesso lo sviluppo di successivi miglioramenti del *mini-batch gradient descent*.

Duchi et al. (2011) proposero un miglioramento che incrementò di molto le prestazioni dell'ottimizzatore, soprattutto in presenza di grandi insiemi di dati, il quale venne denominato *Adagrad*. Questo metodo è in grado di adattare il *learning rate* ai parametri, regolando la grandezza del passo di aggiornamento in base alla loro frequenza: *Adagrad* effettua aggiornamenti più ampi per parametri poco frequenti e aggiornamenti più piccoli per parametri molto frequenti. In seguito, verrà utilizzata la notazione scalare per rendere l'esposizione meno impegnativa. Questa tecnica modifica il *learning rate* ad ogni passo t e per ogni parametro $w_{ij}^{(l)}$, basandosi sul gradiente dei passi precedenti. Per semplicità, si indichi $g_{t,ij} = \Delta L(w_{t,ij}^{(l)}; x^{(i:i+m)}, y^{(i:i+m)})$. L'aggiornamento dei parametri diventa

$$w_{t+1,ij}^{(l)} = w_{t,ij}^{(l)} - \frac{\eta}{\sqrt{G_{t,ij} + \varepsilon}} \cdot g_{t,ij},$$

dove $G_{t,ij}$ è la somma dei quadrati dei gradienti rispetto a $w_{t,ij}^{(l)}$, fino al tempo t, ossia $G_{t,ij} = \sum_{t=1}^{T} (g_{t,ij})^2$. La quantità ε, invece, è un termine di lisciamento che serve ad evitare un termine nullo a denominatore, ed è usualmente impostato a valori di ordine di 10^{-8}. Ciò permette di evitare la regolazione del parametro di *learning rate*, di cui viene solamente impostato un valore iniziale, di norma pari a 0,01.

Dal momento che, G_{ij} è una somma di termini positivi, questa quantità continua a crescere ad ogni epoca, e il *learning rate* decresce fino a tendere a 0. Questa problematica venne risolta da Zeiler (2012), ridefinendo iterativamente G_{ij} come una *media mobile esponenziale* (EWMA). La media al tempo t viene quindi definita come

$$\mathbf{E}[g^2]_{t,ij} = \gamma \mathbf{E}[g^2]_{t-1,ij} + (1 - \gamma) g_{t,ij}^2,$$

dove γ viene solitamente impostato attorno a 0,9. L'aggiornamento dei parametri diventa quindi

$$w_{t+1,ij}^{(l)} = w_{t,ij}^{(l)} - \frac{\eta}{\sqrt{\mathbf{E}[g^2]_{t,ij} + \varepsilon}} \cdot g_{t,ij}.$$

Un ulteriore miglioramento venne presentato in Kingma e Ba (2014), in cui si dimostra come si può raggiungere prestazioni più soddisfacenti, tenendo in memoria valori passati anche del termine $g_{t,ij}$, applicando anche a quest'ultimo una *media mobile esponenziale*. Questo metodo innovativo di discesa del gradiente viene denominato *Adam*. Si determini con $m_{t,ij} = \mathbf{E}[g]_{t,ij}$ e $v_{t,ij} = \mathbf{E}[g^2]_{t,ij}$. Sono quindi definite le quantità

$$m_{t,ij} = \beta_1 m_{t-1,ij} + (1 - \beta_1) g_{t,ij},$$

$$v_{t,ij} = \beta_2 v_{t-1,ij} + (1 - \beta_2) g_{t,ij}^2,$$

dove $m_{0,ij}$ e $v_{0,ij}$ vengono inizializzate a 0. Gli autori dimostrano come queste quantità siano distorte verso lo 0, rendendo opportuna la seguente correzione:

$$\tilde{m}_{t,ij} = \frac{m_{t,ij}}{1 - \beta_1^t}, \qquad \tilde{v}_{t,ij} = \frac{v_{t,ij}}{1 - \beta_2^t}.$$

L'aggiornamento dei parametri diventa quindi

$$w_{t+1,ij}^{(l)} = w_{t,ij}^{(l)} - \frac{\eta}{\sqrt{\tilde{m}_{t,ij} + \varepsilon}} \cdot \tilde{v}_{t,ij}, \tag{1.10}$$

dove opportuni valori iniziali per β_1 e β_2 sono rispettivamente 0,9 e 0,999. Inoltre, Kingma e Ba (2014) dimostrarono empiricamente come questo metodo sia il più efficiente nel processo di stima di una rete neurale multi-strato.

Nelle analisi empiriche presenti nei prossimi capitoli, il processo di *back-propagation* si appoggerà al metodo di discesa del gradiente *Adam*, con una porzione di *mini-batch* definita a seconda dell'ampiezza dell'insieme di dati.

Una funzione di attivazione flessibile ed efficace

La caratteristica principale di una rete neurale è la non linearità delle trasformazioni, che sono necessarie per mettere in relazione i nodi di input

con quelli di output. Questa non linearità è resa possibile dalle funzioni di attivazione. Come mostrato nell'equazione (1.1), la funzione di attivazione $g(\cdot) : \mathbb{R} \to \mathbb{R}$ è una trasformazione non lineare che è applicata a livello di nodo, dove la combinazione lineare z_j viene trasformata nel valore a_j. In Figura 1.2 sono raffigurate le funzioni di attivazione maggiormente utilizzate, descritte in modo più dettagliato da Heaton (2015).

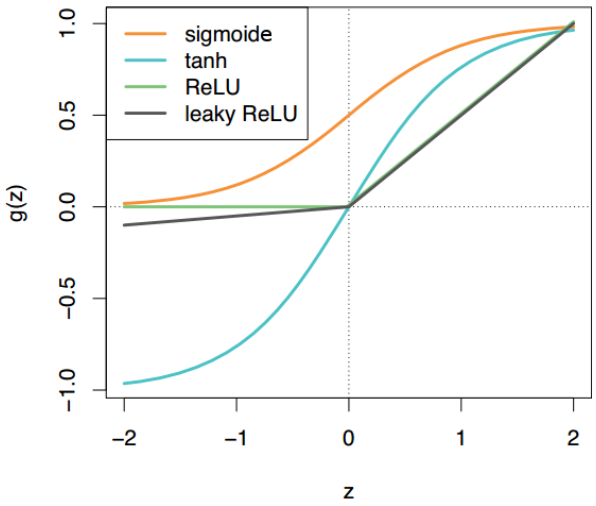

Figura 1.2: Le quattro funzioni di attivazione più usate per una rete neurale multistrato: logistica, tangente iperbolica, ReLU e *leaky* ReLU.

La funzione di attivazione **logistica**, o sigmoide, è stata per molti anni la scelta più comune come funzione di attivazione. Il suo codominio è $\mathrm{C} = (0, 1)$, ed è definita come

$$\text{logistica}(z) = \frac{1}{1 + e^{-z}}.$$

Successivamente, si è diffuso l'utilizzo di una funzione di attivazione antisimmetrica, legata in modo biunivoco alla funzione logistica, ossia la **tangente iperbolica** (tanh). Il suo codominio è $\mathrm{C} = (-1, 1)$, ed è definita come

$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} = 2 \cdot \text{logistica}(2z) - 1.$$

Nonostante l'ampio utilizzo di queste funzioni di attivazione, sorgono alcune problematiche, nel momento in cui vengono applicate a reti neurali

multi-strato. In primo luogo, affidare questo ruolo a funzioni di attivazione limitate può ostacolare la flessibilità del modello. Una funzione limitata ha la caratteristica di muoversi verso un valore ed, infine, di tendere a questo. Ad esempio, la tangente iperbolica tende a -1 al decrescere di z, ed a 1 al crescere di z. Questo significa che cambiamenti anche rilevanti di z, ma lontani dallo 0, corrispondono a variazioni quasi inesistenti di $\tanh(z)$. Questo fenomeno è accentuato dalla presenta di più strati latenti: i nodi di output rischiano di essere insensibili a variazioni dei valori dei nodi di input. Utilizzare una trasformazione non lineare illimitata ed estesa a tutto l'insieme dei numeri reali $\mathbb{R}$ può garantire una maggior flessibilità ed un maggior adattamento ai dati.

La seconda limitazione di queste funzioni di attivazione è la possibile presenza di problemi numerici nella procedura di stima. Il principale di questi è la cosiddetta "scomparsa del gradiente" (*vanishing gradient*), introdotta da Bengio et al. (1994). In sostanza, quando i valori di z si avvicinano agli asintoti orizzontali della tangente iperbolica, il gradiente di questa funzione di attivazione tende a 0. Questo significa che, a valori dei nodi anche non troppo lontani da 0, corrispondono valori molto bassi per i rispettivi gradienti. Nella *fase di propagazione*, quindi, l'equazione (1.7) produce risultati molto vicini a 0, così come il gradiente relativo ai parametri (equazione (1.8)). Nella *fase di aggiornamento*, questa osservazione non porterà il suo contributo. Di conseguenza, il passo di aggiornamento, compiuto nell'equazione (1.10), diventa piccolissimo, poiché vi contribuisce solo una minima parte delle osservazioni. Aggiornamenti infinitesimali dei parametri finiscono per far tendere questi ad una sorta di *plateau*, da cui non riescono più a muoversi. La manifestazione di questo fenomeno cresce all'aumentare della profondità della rete. Questo problema si presenta sia quando viene utilizzata la tangente iperbolica che la funzione logistica, come funzione di attivazione. Si veda Figura 1.3.

Rectified linear unit (ReLU)

Come soluzione ai problemi appena descritti, Glorot et al. (2011) proposero l'utilizzo della funzione *rectified linear unit* (ReLU), la quale è definita

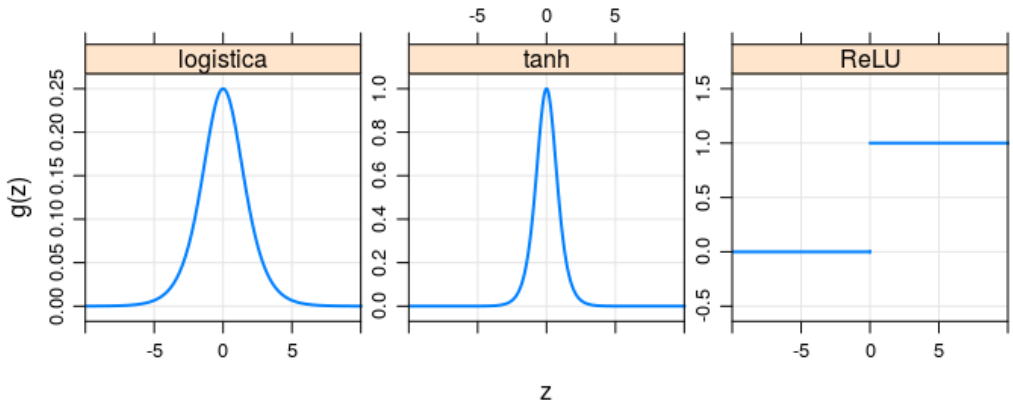

Figura 1.3: Gradiente primo delle funzioni di attivazione *logistica*, *tangente iperbolica* e ReLU.

come

$$\text{ReLU}(z) = z_+ = \max(0, z).$$

Questa funzione corrisponde alla funzione "parte positiva", che usualmente viene utilizzata nella *spline di regressione* lineare, come *funzione di base*. La ReLU possiede derivata prima pari a

$$\frac{\partial g(z)}{\partial z} = \begin{cases} 1, & \text{se } z > 0 \\ 0, & \text{se } z \leq 0. \end{cases}$$

Si veda Figura 1.3. Formalmente parlando, la funzione ReLU non è derivabile in 0. Per convenzione, si pone il gradiente pari a 0, quando $z = 0$.

La funzione ReLU ha dimostrato notevoli miglioramenti in termini di adattamento ai dati di stima. I suoi principali vantaggi sono i seguenti.

- In una parte del dominio, la funzione è lineare e non limitata; questa proprietà rende il modello più sensibile ai valori di input.

- La funzione ReLU non soffre del fenomeno della "scomparsa del gradiente". Circa il 50% dei nodi hanno valore esattamente pari a 0, e così anche il relativo gradiente. Quindi, nella fase di stima, non avviene

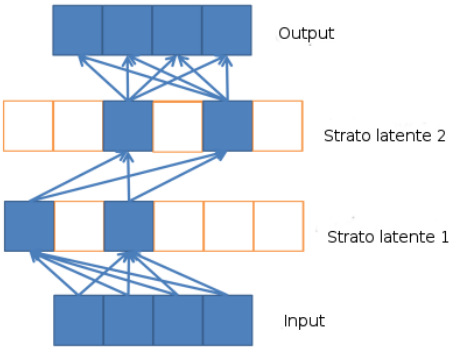

Figura 1.4: Relazione tra nodi in una rete neurale multi-strato con funzione di attivazione ReLU.

alcuna propagazione del gradiente attraverso questi nodi; si veda Figura 1.4. Di conseguenza, ai nodi che rimangono "attivi", il cui valore è strettamente positivo, corrisponde sempre un gradiente pari a 1. Questo garantisce la propagazione del gradiente fino agli strati più bassi, senza che quest'ultimo si riduca a valori molto vicini a 0.

- Grazie alla sua semplicità, i costi computazionali sono ridotti al minimo. Il gradiente può assumere solo i valori 0 o 1, e le trasformazioni tra nodi "attivi" sono sempre lineari.

- Viene introdotta sparsità nelle matrici di parametri $W^{(l)}$. Questa caratteristica è un vantaggio perchè rende flessibile l'ordine di complessità della relazione tra due strati. Avere tutti i nodi diversi da 0 obbliga uno strato ad assumere un numero di trasformazioni non lineari pari al numero dei suoi nodi. Alcune di queste trasformazioni possono essere superflue o ridondanti. Al contrario, porre esattamente a 0 il valore di alcuni nodi aiuta il modello ad assumere un numero di trasformazioni non lineari ottimale, ai fini di rappresentare la vera relazione tra i nodi di input e quelli di output.

Un potenziale problema della funzione di attivazione ReLU è il particolare caso in cui vengano "disattivati", al momento dell'inizializzazione dei parametri, un numero di nodi maggiore di quello che il modello ha bisogno

per rappresentare la relazione tra nodi di input e nodi di output. Questa situazione può accadere dal momento che una stima basata sul gradiente necessita di inizializzare i parametri con valori scelti casualmente. Ci sono varie proposte per risolvere questa problematica. La prima è quella suggerita da Achanta e Hastie (2015), ovvero quella di stimare il modello con diversi punti iniziali. Questo però aggrava di molto il peso computazionale della procedura di stima. La seconda proposta è presentata da Maas et al. (2013), e consiste nell'utilizzare un gradiente piccolo ma diverso da 0 quando un nodo viene inizializzato a 0, rendendo possibile la sua "attivazione". La funzione venne chiamata *leaky* ReLU , ed è definita come

$$\text{LeakyReLU}(z) = z_+ - \alpha z_- = \max(0, z) - \alpha \max(0, -z),$$

la cui derivata è

$$\frac{\partial g(z)}{\partial z} = \begin{cases} 1, & \text{se } z > 0 \\ \alpha, & \text{se } z \leq 0, \end{cases}$$

dove una scelta consigliata per α è 0,01 (Maas et al. 2013). Gli stessi autori, tuttavia, hanno dimostrato empiricamente che questa funzione non porta alcun vantaggio rispetto alla ReLU, in termini di adattamento del modello, se non un piccolo miglioramento dal punto di vista della velocità di convergenza del processo di stima.

Capitolo 2

I punti di forza di una rete neurale multi-strato

Le prestazioni di alcuni algoritmi utilizzati nel *deep learning* sono conosciute solo a livello euristico. Si sa, quindi, solo grazie a studi empirici che questi algoritmi hanno performance eccellenti con alcune tipologie di dati. Ad esempio, questi metodi vengono utilizzati con grandissimo successo per classificare immagini, frammenti video e frammenti audio, per tradurre e trascrivere il linguaggio vocale o per identificare il contenuto di testi e messaggi vocali (LeCun et al. 2015). Inoltre, questi metodi funzionano bene se si ha a disposizione un gran numero di osservazioni. Goodfellow et al. (2016) affermano che i metodi per il *deep learning*, utilizzati per la classificazione, raggiungono risultati soddisfacenti se si ha a disposizione almeno 5000 osservazioni per ogni categoria.

Nonostante le reti neurali multi-strato siano nate nel contesto *machine learning*, queste ultime sono, di fatto, modelli statistici. La rete neurale a singolo strato latente ebbe le sue prime applicazioni già nei primi anni '60. Nel corso degli anni, questo modello venne analizzato dal punto di vista statistico da molti autori, come ad esempio Ripley (1996) e Hastie et al. (2009). Le reti neurali multi-strato, al contrario, si diffusero molto più recentemente. Di conseguenza, non sono ancora state studiate in modo approfondito dal punto di vista statistico. Tuttavia, al fine di trovare i punti di forza e di debolezza delle reti neurali multi-strato, è fondamentale comprendere appieno le loro

proprietà statistiche. Conoscere, infatti, le caratteristiche di questi modelli, è importante sia per utilizzarli adeguatamente che per avere proposte su come migliorarli.

Per far ciò, un primo passo è quello di analizzare le analogie e le differenze tra le reti neurali e altre classi di modelli statistici. Alcune di queste verranno presentate in questo capitolo. Lo scopo di questo confronto è quello di comprendere maggiormente il funzionamento delle reti neurali come modello statistico. Inoltre, la ricerca delle proprietà statistiche che caratterizzano le reti neurali multi-strato, sarà supportata da alcuni teoremi matematici.

Verrà posta l'attenzione, infine, sul confronto tra reti a singolo strato e reti multi-strato, in modo da determinare quali siano i guadagni portati dall'incremento del numero di strati latenti.

Proprietà di una rete neurale a singolo strato

Il motivo principale, per cui si è diffuso l'impiego delle reti neurali, fu quello di riuscire a descrivere i fenomeni complessi, quando i modelli statistici classici risultavano inadeguati. Per molti anni si è fatto uso delle reti neurali a singolo strato, piuttosto che quelle multi-strato. La ragione, che ha maggiormente influito, fu l'inadeguatezza tecnologica, la quale rendeva impossibile la stima di modelli molto complessi. A causa dell'inapplicabilità delle reti neurali multi-strato, lo studio di strati latenti multipli, all'epoca, era limitato anche dal punto di vista teorico.

In questo paragrafo, verrà fornita una presentazione delle principali proprietà statistiche di una rete neurale a singolo strato. In particolare, sarà mostrato come una rete neurale sia un *approssimatore universale*, grazie al confronto con la *projection pursuit regression*. Verranno, in seguito, enunciati i teoremi di approssimazione di funzioni per reti neurali a singolo strato. Come supporto per l'esposizione di tali teoremi, verrà utilizzato quanto studiato da Ripley (1996, sez. 5.7).

Analogie con il modello PPR

Le reti neurali a singolo strato e quelle multi-strato condividono la stessa idea basilare: entrambe sono costruite ipotizzando che non ci sia una relazione lineare tra il fenomeno di interesse e le variabili che lo esplicano. In particolare, tale classe di modelli presuppone che esistano alcuni fattori latenti, i quali non sono osservati direttamente, ma influiscono sul fenomeno di interesse. Ad esempio, nelle reti neurali a singolo strato, la relazione tra variabili di interesse e variabili osservate si sviluppa in due passi: le variabili osservate influiscono i valori di questi fattori latenti, i quali, a loro volta, influiscono le variabili di interesse. Questa caratteristica allarga il numero di possibili relazioni che è possibile spiegare attraverso un modello statistico.

Un modello, in grado di rappresentare qualsiasi tipo di relazione tra variabili di interesse e variabili osservate, è la *projection pursuit*, introdotta da Friedman e Stuetzle (1981) e discussa, ad esempio, da Hastie et al. (2009, sez. 11.2) e Azzalini e Scarpa (2012, sez. 4.6).

La *projection pursuit* è un modello additivo applicato, non direttamente ai predittori, ma ad alcune loro trasformazioni non lineari, o meglio ad alcune loro proiezioni. Si assuma che la matrice delle variabili esplicative X sia p-dimensionale, mentre la variabile risposta y sia unidimensionale; il modello può essere scritto come

$$y = \beta_0 + \sum_{k=0}^{K} f_k(\beta_k^T X),$$

ed è chiamata *projection pursuit regression* (PPR). Il numero di proiezioni K è un parametro di regolazione, $\beta_k \in \mathbb{R}^p$ sono i vettori di proiezione, mentre $f_k(\cdot)$ sono chiamate funzioni *ridge* (o funzioni dorsale). Il processo di stima è iterativo, e alterna il metodo di Gauss-Newton, per la stima dei vettori di proiezione, con qualche metodo non parametrico di lisciamento, per ricavare le funzioni *ridge*. L'applicazione di funzioni non lineari a combinazioni lineari di X rende il modello altamente flessibile: infatti, se K viene fissato ad un valore arbitrariamente grande e le funzioni $f_k(\cdot)$ sono scelte in modo appropriato, il modello PPR, può approssimare qualsiasi funzione continua in $\mathbb{R}^p$. Modelli che presentano questa proprietà sono detti *approssimatori universali*.

La rete neurale a singolo strato può essere vista come un particolare caso del modello PPR appena descritto. Il numero di nodi della rete neurale corrisponde al numero di proiezioni del modello PPR. La principale differenza consiste nel fatto che il modello PPR fa uso di funzioni non parametriche $f_k(v)$, le quali hanno un ordine di complessità generalmente maggiore rispetto alle funzioni di attivazione $g(v)$, che hanno un numero finito parametri liberi, spesso solo tre. La formulazione (1.4), applicata ad una rete neurale a singolo strato, diventa

$$y = w_0^{(2)} + \sum_{k=1}^{K} \left(w_k^{(2)} g \left(w_{0k}^{(1)} + \boldsymbol{w}_k^{(1)^T} X \right) \right). \tag{2.1}$$

Riscrivendo la rete neurale come un modello PPR, una singola proiezione diventa

$$\begin{aligned} f_k(\beta_k^T X) &= w_k^{(2)} g \left(w_{0k}^{(1)} + \boldsymbol{w}_k^{(1)^T} X \right) \\ &= w_k^{(2)} g \left(w_{0k}^{(1)} + \|\boldsymbol{w}_k^{(1)}\| (\beta_k^T X) \right), \end{aligned}$$

dove $\beta_k = \boldsymbol{w}_k^{(1)} / \|\boldsymbol{w}_k^{(1)}\|$, in quanto i vettori di proiezione β_k sono vettori unitari, che determinano la direzione della proiezione. Le funzioni *ridge*, in questo caso, sarebbero completamente parametriche, e richiederebbero meno parametri liberi. La proiezione di una rete neurale, sotto forma di modello PPR, contiene infatti $p + 2$ parametri, mentre la proiezione di un classico modello PPR possiede un numero di parametri pari a $(p-1)+s$, dove s sono i gradi di liberà equivalenti necessari per stimare $f_k(\cdot)$ non parametricamente. Le analogie con il modello PPR rendono la rete neurale un modello statistico estremamente flessibile ed idoneo ad approssimare qualsiasi classe di funzioni ed a rappresentare qualsiasi relazione tra variabili.

Un approssimatore universale, tuttavia, è un modello di difficile interpretazione, in quanto la relazione tra variabili diventa molto complessa a causa dell'utilizzo di strati latenti o proiezioni. Per questo motivo, una rete neurale è utile quando lo scopo è prettamente quello di ottenere previsioni accurate, e non quello di fornire un modello comprensibile ed interpretabile.

Teoremi di approssimazione per reti neurali a singolo strato

Dal paragrafo precedente si è dedotto, in modo intuitivo, come una rete neurale a singolo strato sia un *approssimatore universale*; in questo paragrafo lo stesso concetto verrà mostrato dal punto di vista matematico. Si pensi alla relazione tra variabili osservate e fenomeno di interesse come una funzione matematica molto complessa. Alcuni teoremi matematici, presenti in letteratura, dimostrano che una rete neurale è un *approssimatore universale*, ovvero che, con una rete neurale a singolo strato dotata di un arbitrario numero di parametri, è possibile approssimare qualsiasi funzione matematica liscia e continua. Il primi, in ordine temporale, furono enunciati indipendentemente da Cybenko (1989) e Hornik et al. (1989), ma quella che segue è la versione di Ripley (1996):

Teorema 1 *Ogni funzione continua* $f : \mathbb{R}^n \to \mathbb{R}$ *può essere approssimata uniformemente su un insieme compatto da funzioni della forma* (2.1)*, con nodi di output lineari e funzioni di attivazione in grado di approssimare una funzione a gradini in* $(0, 1)$*.*

È facilmente dimostrabile che la funzione logistica e la tangente iperbolica approssimano una funzione a gradini. In Ripley (1996), viene fornita una dimostrazione del teorema per reti neurali con funzione di attivazione logistica.

In seguito, Barron (1993) fornì la relazione tra il numero di nodi e l'errore di approssimazione.

Teorema 2 *Si supponga che* $f : \mathbb{R}^n \to \mathbb{R}$ *possa essere rappresentata attraverso una trasformata di Fourier della forma*

$$f(x) = \int_{\mathbb{R}^n} e^{i\omega^T x} \tilde{f}(\omega) d\omega$$

con

$$C_f = \int_{\mathbb{R}^n} \|\omega\| \tilde{f}(\omega) d\omega < \infty$$

allora per ogni N, f può essere approssimato da una funzione della forma (2.1) *con N nodi nello strato latente, in* $L_2(\mu)$ *su* $B_r = \{\|x\| < r\}$*, con errore al più pari a* $(2\pi C_f)/\sqrt{N}$*.*

Quindi, il numero di nodi N e l'errore di approssimazione hanno una relazione quadratica ed inversamente proporzionale. Può sembrare che il risultato, prodotto dal teorema 2, non dipenda da n e che quindi non sia influenzato dalla *maledizione della dimensionalità*. Ciò non è vero: in primo luogo, l'insieme B_r, in cui è garantita l'approssimazione, diventa più piccolo al crescere di n; in secondo luogo, solitamente C_f cresce esponenzialmente con n.

Proprietà di una rete neurale multi-strato

Dal punto di vista logico, è naturale pensare che, per rappresentare relazioni tra variabili molto complesse attraverso una rete neurale, sia necessario un numero maggiore di trasformazioni non lineari. Questa fu l'idea che diede origine ai primi utilizzi di reti neurali multi-strato. La *profondità* del modello è il numero di trasformazioni non lineari applicate alle variabili osservate. La costruzione di una rete con diversi strati latenti genera un vero e proprio *modello gerarchico*, in cui i valori dei nodi dipendono unicamente dai nodi dello strato precedente.

Il primo punto di forza di una rete neurale multi-strato è che viene estratta, ad ogni strato, solamente l'informazione essenziale. Dal punto di vista probabilistico ciò si traduce con la dipendenza di ogni strato dal valore atteso degli strati precedenti. Questo concetto è introdotto da Mohamed (2015), ed in seguito verrà dimostrato con semplici operazioni di calcolo delle probabilità.

Si consideri una rete neurale a singolo strato, in cui non sono osservati i valori dei nodi. Si possono, quindi, considerare gli strati come variabili aleatorie Y, A ed X, le quali sono rispettivamente lo strato di output, quello latente e quello di input. Per la probabilità del valore atteso condizionato (Ross 2014, p. 435), si ha

$$\mathbb{E}_A[f(A)] = \mathbb{E}_X[\mathbb{E}_A[f(A)|X]].$$

Utilizzando questo risultato, è semplice ricavare il valore atteso di Y condizionato a X, che è alla base della modellazione statistica:

$$\begin{aligned}\mathbb{E}_Y[Y|X] &= \mathbb{E}_A[\mathbb{E}_Y[Y|A,X]] \\ &= \mathbb{E}_X[\mathbb{E}_A[\mathbb{E}_Y[Y|A,X]|X]] \\ &= \mathbb{E}_X[\mathbb{E}_A[\mathbb{E}_Y[Y|A]|X]],\end{aligned}$$

dove $\mathbb{E}_Y[Y|A,X] = \mathbb{E}_Y[Y|A]$ perché nelle reti neurali, uno strato dipende solo da quello precedente. Estendere il risultato alle reti neurali multi-strato è piuttosto immediato. Siano $A_1, ..., A_L$ le variabili aleatorie corrispondenti agli strati latenti. Il valore atteso condizionato diventa

$$\mathbb{E}_Y[Y|X] = \mathbb{E}_X[\mathbb{E}_{A_1}[\cdots \mathbb{E}_{A_L}[\mathbb{E}_Y[Y|A_L]|A_{L-1}]\cdots|X]].$$

Quest'ultimo risultato mette in evidenza diversi aspetti:

- la costruzione gerarchica della rete neurale multi-strato;
- la dipendenza markoviana di uno strato dagli strati precedenti;
- l'estrazione dell'informazione di ogni strato tramite il valore atteso.

Nel paragrafo 2.1.2, sono stati enunciati i principali teoremi matematici riguardo l'approssimazione di funzione, attraverso una rete neurale a singolo strato. Nel prossimo paragrafo, sarà affrontata l'estensione di tali teoremi a reti neurali multi-strato. In particolare, l'estensione riguarda l'incremento del numero degli strati latenti e l'applicazione della funzione di attivazione ReLU, caratteristica del *deep learning*.

Teoremi di approssimazione per reti neurali multi-strato

Come già annunciato ad inizio capitolo, lo studio di reti neurali multi-strato è esploso solo recentemente, ossia quando anche il progresso tecnologico ha reso possibile l'utilizzo di modelli così complessi. Delalleau e Bengio (2011) affermano che una rete neurale multi-strato può essere riscritta come una rete neurale a singolo strato con un arbitrario numero di nodi. Del resto, ciò viene già dimostrato nel Teorema 1, in quanto una rete neurale multi-strato rientra nella classe di funzioni continue f, approssimabili tramite una rete neurale a singolo strato. In seguito è dimostrato, tuttavia, che l'utilizzo di un singolo strato per approssimare una funzione non è conveniente.

Il primi due teoremi che confrontano reti neurali a singolo strato con quelle multi-strato, in termini di approssimazione di una funzione, furono proposti da Delalleau e Bengio (2011). Il primo è riservato a reti neurali con una profondità che dipende dal numero di nodi di input n.

Teorema 3 *Ogni rete neurale somma-prodotto a singolo strato, per approssimare $f \in \mathbb{F}$, necessita di almeno $O(2^{\sqrt{n}})$ nodi latenti. Una rete neurale con un numero di strati latenti di ordine $O(\log n)$ necessita, invece, di $O(n)$ nodi latenti.*

Il termine "somma-prodotto" si riferisce ad una rete neurale che può essere scritta come $\sum_j w_j \prod_t x_t^{\gamma_{jt}}$, con $w_j \in \mathbb{R}$ e $\gamma_{jt} \in \{0, 1\}$, per $t = 1, ..., n$. La rete neurale "somma-prodotto" ha una formulazione identica allo sviluppo in serie di potenze per più variabili (Boas e Khavinson 1997). Per il teorema di Stone-Weierstrass, una funzione continua nella forma (2.1) può essere approssimata da una serie di potenze.

Il secondo teorema rilassa la condizione di dipendenza del numero di strati latenti dal numero di nodi di input n.

Teorema 4 *Si consideri $k \in \mathbb{N}$. Ogni rete neurale somma-prodotto a singolo strato, per approssimare $f \in \mathbb{F}$, necessita di almeno $O((n-1)^k)$ nodi latenti, con k che dipende dalla complessità di f. Una rete neurale con un numero di strati latenti di ordine $O(k)$ necessita, invece, di $O(nk)$ nodi latenti.*

Le dimostrazioni di tali teoremi sono presenti nell'articolo originale.

Successivamente, sono stati pubblicati numerosi lavori riguardanti l'approssimazione di funzioni, attraverso reti neurali multi-strato. I principali sono i seguenti:

- Eldan e Shamir (2015) affermano che una funzione $f : \mathbb{R}^d \to \mathbb{R}$ è arbitrariamente approssimabile da una rete neurale a due strati latenti, ognuno con una numerosità di nodi *polinomiale* in d. La stessa funzione è approssimabile da una rete neurale con uno strato latente e con un numero di nodi *esponenziale* in d, con lo stesso errore di approssimazione ε.

- Telgarsky (2016) dimostra come una rete neurale con $O(k^3)$ strati latenti, $O(1)$ nodi per strato, non possa essere approssimata da una rete neurale con $O(k)$ strati latenti, se quest'ultima ha meno di $O(2^k)$ nodi.

- Liang e Srikant (2016) stabilirono la profondità e i numero di nodi sufficienti per garantire un errore di approssimazione ε di una funzione

continua e derivabile. Una rete neurale con $O(\log(1/\varepsilon))$ strati necessita di $O(\log poly(1/\varepsilon))$ nodi, mentre una rete con meno di $O(\log(1/\varepsilon))$ strati, richiede $O(poly(1/\varepsilon))$ nodi.

I teoremi 1 e 2, e gli studi appena citati portano tutti alla stessa conclusione: approssimare una funzione matematica con un errore ε fissato, utilizzando una rete neurale multi-strato, porta ad un guadagno esponenziale, in termini di numero di nodi, rispetto ad una rete neurale a singolo strato.

È facile intuire, comunque, che anche una rete neurale multi-strato mantenga la proprietà di approssimare arbitrariamente bene qualsiasi tipo di funzione continua, ma determinare il guadagno in termini di numero di nodi non è scontato dal punto di vista logico. I teoremi appena citati, però, valgono solamente se le funzioni di attivazione utilizzate non sono lineari. Se venisse utilizzata una funzione di attivazione lineare per connettere i nodi degli strati vicini, ciò equivale ad utilizzare un modello di *regressione lineare*, qualsiasi sia la profondità della rete.

Nel teorema 1, si fa riferimento a funzioni di attivazione in grado di approssimare una funzione a gradini, come ad esempio la funzione logistica o la tangente iperbolica. Nel prossimo paragrafo, verrà mostrato come questo teorema possa essere esteso alla funzione di attivazione più usata nel *deep lerning*, ossia la funzione ReLU.

Proprietà della funzione di attivazione ReLU

Nel paragrafo 1.4.1, sono stati presentati i principali vantaggi della funzione ReLU. Ora, verranno discusse le proprietà statistiche di questa funzione di attivazione e sarà fornita una motivazione per cui quest'ultima risulta un punto di forza, quando viene applicata ad una rete neurale multi-strato.

Nonostante la funzione di attivazione ReLU applichi trasformazioni lineari tra i nodi "attivi" di due strati consecutivi, la rete neurale mantiene la sua caratteristica principale, ossia la non linearità. Glorot et al. (2011) affermano che è possibile ottenere una funzione non lineare e limitata, simile alla logistica o alla tangente iperbolica, attraverso la combinazione di due funzioni ReLU con parametri condivisi. In seguito, verrà presentato un modo per dimostrare questa affermazione. Si considerino due combinazioni lineari dei

nodi di input

$$z_1 = w_{01} + \sum_{i=1}^{p} w_i x_i, \qquad z_2 = w_{02} + \sum_{i=1}^{p} w_i x_i,$$

dove i parametri scalari w_i sono condivisi tra le due formulazioni. Si ponga, per semplicità di notazione, $v = \sum_{i=1}^{p} w_i x_i$, e si applichi la funzione ReLU a z_1 e z_2:

$$a_1 = \max(0, w_{01} + v) \qquad a_2 = \max(0, w_{02} + v).$$

Combinando linearmente a_1 e a_2 si ottiene

$$\sigma(v) = \begin{cases} \dfrac{1}{w_{02} - w_{01}} \big(\max(0, w_{01} + v) - \max(0, w_{02} + v)\big) & \text{se } w_{01} < w_{02} \\ \dfrac{1}{w_{01} - w_{02}} \big(\max(0, w_{02} + v) - \max(0, w_{01} + v)\big) & \text{se } w_{01} > w_{02} \end{cases}$$

ricavando una funzione a tratti, limitata tra 0 e 1, con punti di non linearità in $-w_{01}$ e $-w_{02}$. Si veda la Figura 2.1.

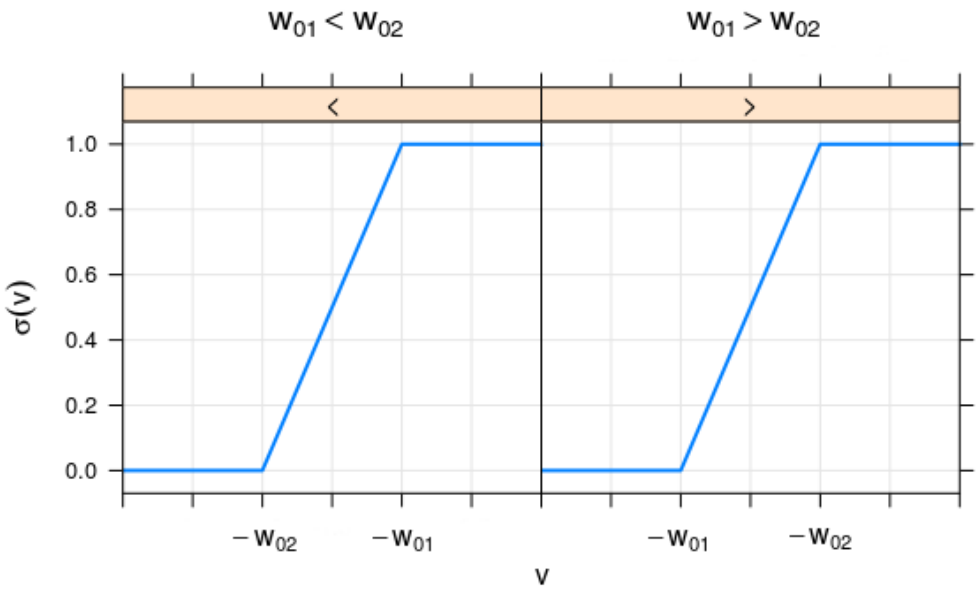

Figura 2.1: Combinazione lineare di due funzioni ReLU, nei casi in cui $w_{01} < w_{02}$ e $w_{01} > w_{02}$.

Per approssimare una funzione a gradini, una funzione deve essere limitata in $(0, 1)$ e monotona crescente. La combinazione di due funzioni ReLU, soddisfando queste proprietà, rispetta la condizione del teorema 1.

Quanto appena dimostrato conferma l'affermazione di Glorot et al. (2011), ossia che, se viene utilizzata la funzione ReLU, la rete neurale deve possedere almeno il doppio dei nodi, che avrebbe con una qualsiasi altra funzione di attivazione non lineare. Inoltre, è logico dedurre che, combinando linearmente un numero maggiore di funzioni ReLU e rilassando l'ipotesi di condivisione dei parametri, si possano ottenere trasformazioni non lineari più complesse, incrementando così la flessibilità del modello.

Analogie con il modello MARS

È immediato notare che la funzione ReLU ha la stessa formulazione di una *funzione di base* lineare, che usualmente viene utilizzata nella *spline di regressione* lineare. Quest'ultima ha la capacità di approssimare funzioni continue unidimensionali attraverso alcune funzioni lineari a tratti, con punti di non linearità determinati dai nodi. Una naturale estensione delle *spline di regressione* lineari nel contesto multi-variato sono le *Multivariate adaptive regression splines* (MARS), le quali furono introdotte da Friedman (1991) e discusse da Hastie et al. (2009). Per ogni variabile x_j, con $j = 1, ..., p$ e per ogni nodo ξ, con $\xi \in \{x_{1j}, ..., x_{nj}\}$, si consideri la coppia di funzioni di base $C = \{(x_j - \xi)_+, (\xi - x_j)_+\}$. Il modello è costruito attraverso una regressione lineare *stepwise forward*, in cui, al posto delle variabili esplicative, vengono utilizzate le coppie di funzioni di base. Il modello risulta

$$y = \beta_0 + \sum_{m=1}^{M} \beta_m h_m(X),$$

dove $h_m(X)$ è una funzione di base presente in C, oppure un prodotto tra due o più di esse. Non può essere inserita una singola funzione di base, ma solamente l'intera coppia. Le caratteristiche principali di un modello MARS sono:

- l'abilità di operare localmente, attraverso le spline di regressione, fornisce al modello MARS la capacità di adattarsi all'insieme di dati senza fare assunzioni parametriche;

- la possibilità di inserire prodotti di funzioni di base permette al modello di considerare la relazione tra variabile risposta e *interazioni* di variabili esplicative.

In seguito, è mostrato il modo in cui il modello MARS può essere riscritto come un caso particolare di rete neurale multi-strato con funzioni di attivazione ReLU, in cui vengono posti alcuni vincoli sui parametri. Questa reinterpretazione del modello MARS dimostra che anche la rete neurale multi-strato possiede le due proprietà appena citate, sotto l'ipotesi che un modello non vincolato abbia almeno le capacità di adattamento dello stesso modello, con vincoli nei parametri.

Si consideri, in prima battuta, il caso in cui vengano aggiunte nuove coppie di funzioni di base solo in modo addizionale, e non attraverso prodotti di basi. Si supponga, inoltre, che la m-esima funzione di base sia $(x_j - \xi)_+$. Riscrivendo il modello MARS come una rete neurale, il valore di un generico nodo dell'ultimo strato latente diventa

$$a_m = h_m(X) = \max(0, x_j - \xi). \tag{2.2}$$

Quindi, la quantità $x_j - \xi$ del modello MARS corrisponde, nella rete neurale, alla combinazione lineare che entra nel nodo m, a cui viene applicata la funzione ReLU. La funzione di base, quindi, può essere riscritta sotto forma di nodo:

$$(x_j - \xi)_+ = \left(w_0 + \sum_{i=1}^{p} w_i x_i \right)_+ ,$$

con $w_0 = -\xi$, $w_j = 1$ e $w_i = 0$ per $i \neq j$. Il procedimento e' molto simile per l'altra funzione di base della coppia, ossia la funzione parte negativa $(\xi - x_j)_+$. Si deduce, dunque, che una rete neurale con un solo strato latente e con funzioni di attivazione ReLU è sufficiente per rappresentare un modello MARS, in cui non ci sono prodotti di funzioni di base. Il numero di nodi della rete coincide con il numero di basi del MARS.

Si consideri, ora, il caso in cui il modello MARS contenga anche prodotti di due funzioni di base. Nella rete neurale, un generico prodotto tra due funzioni di base, $h_m(X)h_n(X)$, non può essere rappresentato come prodotto di nodi, $a_m a_n$, in quanto la rete neurale permette solo combinazioni lineari.

È possibile, tuttavia, approssimare un prodotto di nodi nel seguente modo:

$$a_m a_n \approx m(a_m, a_n) = (a_m + a_n)_+ + (-a_m - a_n)_+ - (a_m - a_n)_+ - (-a_m + a_n)_+. \tag{2.3}$$

Questo risultato deriva dal lavoro prodotto da Lin e Tegmark (2016).

In particolare, gli autori affermano che un prodotto di due nodi può essere approssimato da una rete neurale con uno strato e quattro nodi latenti. Si consideri la generica coppia di valori (u, v), e la generica funzione di attivazione non lineare $g(\cdot)$. Nell'articolo originale viene fornito il seguente risultato:

$$\frac{g(u+v) + g(-u-v) - g(u-v) - g(-u+v)}{4g''(0)} = uv[1 + O(u^2 + v^2)]. \tag{2.4}$$

dove $g''(0) \neq 0$ poiché la funzione di attivazione non è lineare.

Quindi una rete neurale a singolo strato, con quattro nodi latenti ed u e v come nodi di input, approssima al secondo ordine il prodotto tra u e v. Nell'equazione (2.3) la funzione di attivazione è la funzione ReLU, la cui derivata seconda in zero, però, non esiste. Al suo posto, si può usare la derivata seconda di una versione liscia della funzione ReLU, ossia la funzione *softplus*, introdotta da Glorot et al. (2011), e definita come

$$\text{softplus}(z) = \log(1 + e^z),$$

la cui derivata seconda è pari a

$$\frac{\partial^2 g(z)}{\partial z^2} = \frac{e^z}{(1+e^z)^2}.$$

Si utilizza, quindi, la funzione *softplus* per la derivata seconda a denominatore dell'espressione (2.4): l'approssimazione che si ricava è $g''(0) = 1/4$, con cui si ottiene la formulazione (2.3). Quindi, un prodotto di due funzioni di base, nel modello MARS, viene rappresentato dalla somma di quattro nodi, nella rete neurale. Il valore di ognuno di questi nodi è stabilito dall'applicazione della funzione ReLU alla combinazione lineare dei nodi dello strato precedente. Ad esempio, il primo nodo è

$$(a_m + a_n)_+ = \left(w_0 + \sum_{i=1}^{p} w_i a_i \right)_+,$$

con $w_0 = 0$, $w_m = w_n = 1$ e $w_i = 0$ per $i \neq \{m, n\}$. Per a_m e a_n vale l'equazione (2.2). Si noti, quindi, come sia necessaria una rete neurale con due strati latenti per ottenere una rappresentazione di un modello MARS con prodotti di due funzioni di base. Infatti, a_m non fa più parte della combinazione lineare che determina la variabile risposta, ma entra nella combinazione lineare di $m(a_m, a_n)$, il quale, a sua volta, viene combinato linearmente con altri nodi per determinare y.

In conclusione, grazie a quanto appena dimostrato, è stata provata la capacità di una rete neurale, con funzioni di attivazione ReLU, di adattarsi localmente e di essere in grado di stimare le interazioni tra variabili. Inoltre, è stato dimostrato, in modo "statistico", il motivo per cui una rete neurale multi-strato possiede caratteristiche di adattamento migliori rispetto ad una rete a singolo strato, nel caso in cui venga utilizzata la funzione di attivazione ReLU. Mantenendo i vincoli imposti dal modello MARS, una rete neurale a singolo strato è in grado di modellare solamente le relazioni marginali tra variabili esplicative e variabile risposta, mentre all'aumentare della profondità della rete, aumenta l'ordine di interazioni che la rete neurale riesce a rappresentare.

Questo ragionamento, naturalmente, vale solo per il caso particolare di rete neurale con parametri vincolati, come descritto in precedenza. Se la rete viene liberata dai vincoli sui parametri e sul numero di nodi, anche una rete neurale a singolo strato, con funzione di attivazione ReLU, riesce a modellare le interazioni tra variabili. Ciò è possibile dal momento che, come mostrato in precedenza, una rete a singolo strato è un *approssimatore universale*, anche quando la funzione di attivazione è la ReLU. Lin e Tegmark (2016), tuttavia, dimostrano che, per approssimare un prodotto di n termini con una rete neurale a singolo strato, occorrono $O(2^n)$ nodi. Al contrario, utilizzando una rete multi-strato, è possibile impiegare solamente $O(4n)$ nodi, disposti in $O(log_2 n)$ strati. Modellare un'interazione tra n variabili risulta, quindi, molto più conveniente se si sfrutta la profondità della rete.

Studio di simulazione

Si supponga che i dati provengano dal modello $Y = f(X)$. Adattando il modello ai dati x, si ottengono i valori stimati $\hat{y} = \hat{f}(x)$. In fase di previsione, un importante indice di bontà del modello è l'*errore quadratico medio* (MSE), il quale è definito come

$$\mathbb{E}\{[\hat{y} - f(x')]^2\} = [\mathbb{E}\{\hat{y}\} - f(x')]^2 + Var\{\hat{y}\} = \text{distorsione}^2 + \text{varianza},$$

dove x' sono le nuove osservazioni di cui si vuole fare le previsioni. Si può notare come questo indice sia scomponibile in *distorsione* e *varianza* del modello. Per approfondire la dimostrazione di questa scomposizione si veda Hastie et al. (2009, sez. 2.5).

La *distorsione* del modello è la differenza tra il vero valore della funzione in x e il valore atteso della previsione, al netto della casualità dell'insieme di dati di stima. I teoremi di approssimazione del paragrafo 2.1.2 e 2.2.1 fanno riferimento a questa quantità: determinano, infatti, il grado di approssimazione di una funzione raggiunto dalla rete neurale "ideale" con n nodi, la quale non dipende dai dati di stima. Questa indipendenza viene raggiunta applicando il valore atteso.

La *varianza* del modello è semplicemente la varianza delle previsioni, che si ottengono cambiando i dati di stima. Non esistono teoremi che determinano la dipendenza di questa quantità dal numero di nodi, nel caso delle reti neurali multi-strato. È noto, tuttavia, che la varianza dipende fortemente dalla complessità del modello: essa cresce al crescere del numero di parametri e viceversa. Di conseguenza, è considerato migliore un modello che raggiunge un certo adattamento con il minor numero di parametri, poiché la sua varianza è inferiore.

Per quantificare i vantaggi di una rete neurale multi-strato, rispetto a quella a singolo strato, anche su questo fronte, sono stati effettuati tre studi di simulazione. Lo scopo è quello di confrontare le capacità di adattamento, in termini di numero di parametri, delle reti neurali con uno, due e tre strati latenti. Per far ciò, sono fissate alcune soglie di errore; vengono poi confrontate le tre reti, sulla base del numero di nodi necessari per raggiungere un errore inferiore a queste soglie. I passi per effettuare la simulazione sono:

- generare le variabili esplicative casualmente;
- generare la variabile risposta dalle variabili esplicative, in modo deterministico, da una funzione $f(x)$;
- scegliere diverse soglie di errore di stima (come indice di errore si utilizza il *mean absolute error*);
- calcolare il numero minimo di nodi e parametri necessari per garantire un errore di stima inferiore a queste soglie, per una rete neurale a singolo strato;
- eseguire lo stesso procedimento del punto precedente anche per reti neurali a due e tre strati latenti, e confrontare i risultati.

Nel primo caso di studio si ha una sola variabile esplicativa, la quale è composta da 200 valori generati uniformemente nell'intervallo $(-10, 10)$. La funzione (Figura 2.2) da cui è stata simulata la variabile risposta è

$$y = \begin{cases} -2.186x - 12.864, & \text{per } -10 \leq x < -2 \\ 4.264x, & \text{per } -2 \leq x < 0 \\ 10e^{-0.05x-0.5}\sin(x(0.03x + 0.7)), & \text{per } 0 \leq x < 10 \end{cases}$$

Si scelgono 8 soglie di errore equi-spaziate, tra 0.5 e 1.55; i risultati ottenuti sono rappresentati in Figura 2.3 e Tabella 2.1.

Si noti come le reti neurali multi-strato necessitino di meno nodi e meno parametri per raggiungere la soglia di errore fissata. Ciò comporta una minor aumento di varianza, la quale è direttamente legata alla complessità del modello. La rete neurale migliore è quella con due strati latenti. Quindi, per reti neurali multi-strato, la capacità di approssimazione non sempre aumenta all'aumentare degli strati latenti.

Nel secondo caso di studio si hanno due variabili esplicative, ognuna delle quali è composta da 200 valori generati uniformemente nell'intervallo $(-1, 1)$. La funzione (Figura 2.4) da cui è stata simulata la variabile risposta è

$$y = 2\sin(\pi e^{-x_1-x_2}).$$

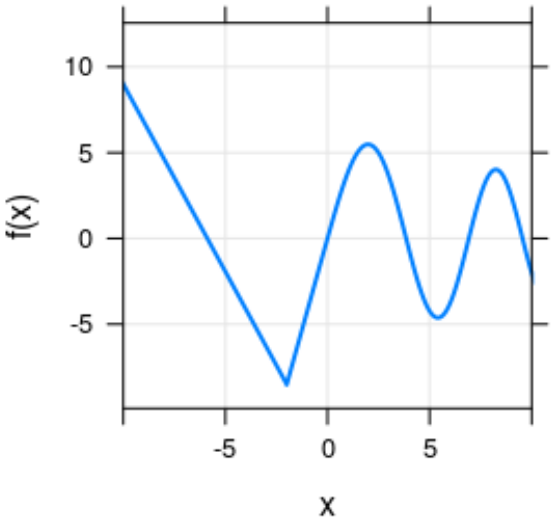

Figura 2.2: Grafico della funzione del primo studio di simulazione.

Tabella 2.1: Errori di approssimazione del primo studio di simulazione. Sono stati calcolati il numero di nodi necessari per reti a 1, 2 e 3 strati latenti.

MAE	nodi 1	nodi 2	nodi 3	param 1	param 2	param 3
1.55	18	10	9	55	46	34
1.40	18	10	9	55	46	34
1.25	21	10	9	64	46	34
1.10	21	10	9	64	46	34
0.95	27	10	15	82	46	76
0.80	27	10	15	82	46	76
0.65	27	10	15	82	46	76
0.50	87	18	21	262	118	134

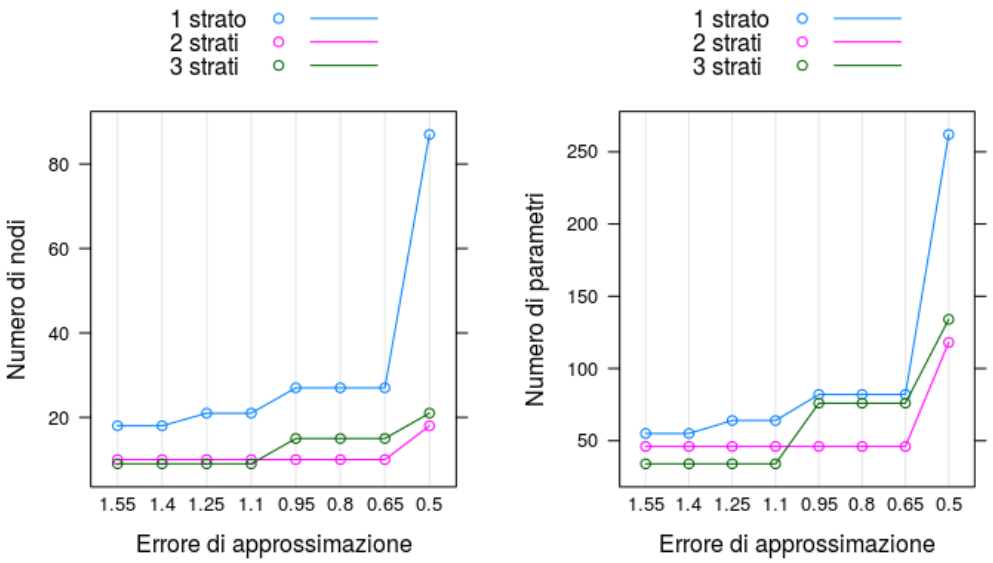

Figura 2.3: Errori di approssimazione del primo studio di simulazione.

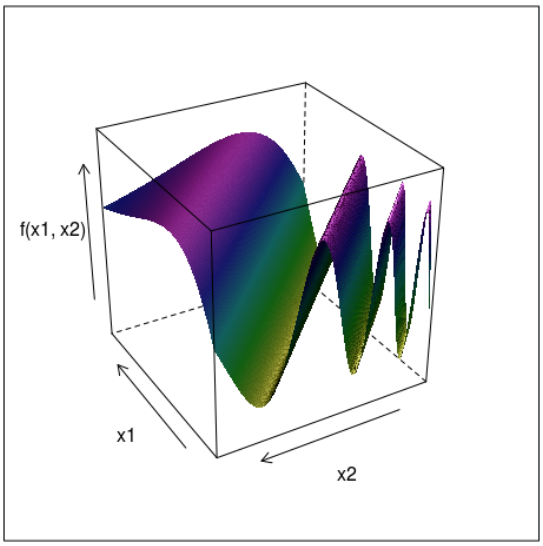

Figura 2.4: Grafico della funzione del secondo studio di simulazione.

Tabella 2.2: Errori di approssimazione del secondo studio di simulazione. Sono stati calcolati il numero di nodi necessari per reti a 1, 2 e 3 strati latenti.

MAE	nodi 1	nodi 2	nodi 3	param 1	param 2	param 3
0.8	27	6	6	82	22	19
0.7	45	14	30	136	78	251
0.6	51	26	30	154	222	251
0.5	51	26	30	154	222	251
0.4	78	26	30	235	222	251
0.3	183	38	45	550	438	526
0.2	420	38	54	1261	438	739
0.1	1164	62	57	3493	1086	818

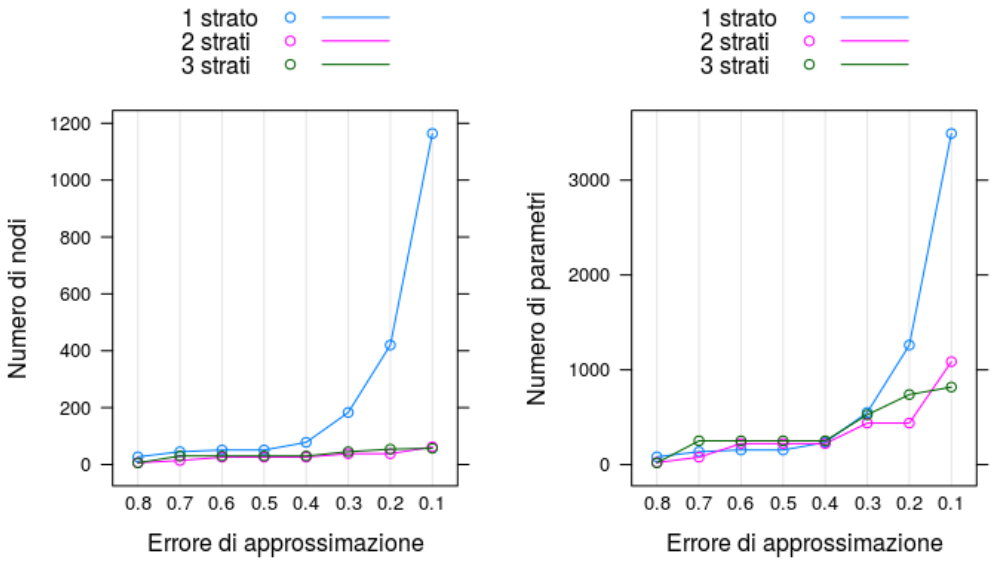

Figura 2.5: Errori di approssimazione del secondo studio di simulazione.

Si scelgono 8 soglie di errore equi-spaziate, tra 0.1 e 0.8; i risultati ottenuti sono rappresentati in Figura 2.5 e Tabella 2.2.

Si noti come, anche in questo caso, le reti neurali multi-strato necessitino di meno nodi e meno parametri per raggiungere la soglia di errore fissata. Le reti neurali a due e tre strati hanno pressoché la stessa capacità di adattamento.

Nel terzo caso di studio si hanno dieci variabili esplicative, ognuna delle quali è composta da 100 valori generati da una normale standard. La funzione da cui è stata simulata la variabile risposta è

$$y = \prod_{m=1}^{10} \log(\phi(X_m)) \text{ con } \phi(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}.$$

In Figura 2.6 sono rappresentate le marginali univariate e bivariate.

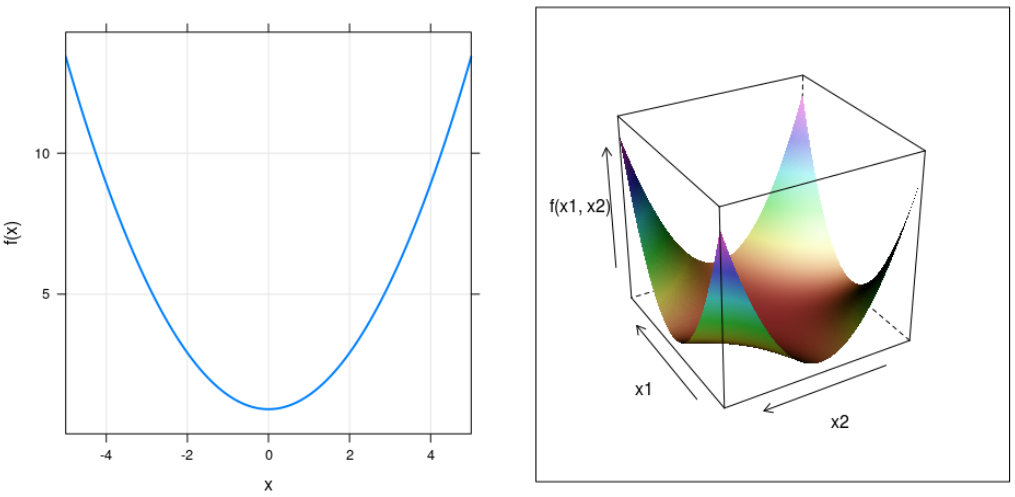

Figura 2.6: Grafico della funzione del terzo studio di simulazione.

Si scelgono 8 soglie di errore equi-spaziate, tra 3 e 24; i risultati ottenuti sono rappresentati in Figura 2.7 e Tabella 2.3.

Anche in questo caso, le reti neurali multi-strato necessitano di meno nodi e meno parametri per raggiungere la soglia di errore fissata. In questo caso la rete neurale a tre strati ha decisamente capacità di adattamento migliori.

Tabella 2.3: Errori di approssimazione del terzo studio di simulazione. Sono stati calcolati il numero di nodi necessari per reti a 1, 2 e 3 strati latenti.

MAE	nodi 1	nodi 2	nodi 3	param 1	param 2	param 3
24	9	10	12	28	46	53
21	18	10	12	55	46	53
18	27	14	15	82	78	76
15	45	18	21	136	118	134
12	54	18	21	163	118	134
9	90	26	21	271	222	134
6	153	26	24	460	222	169
3	246	38	30	739	438	251

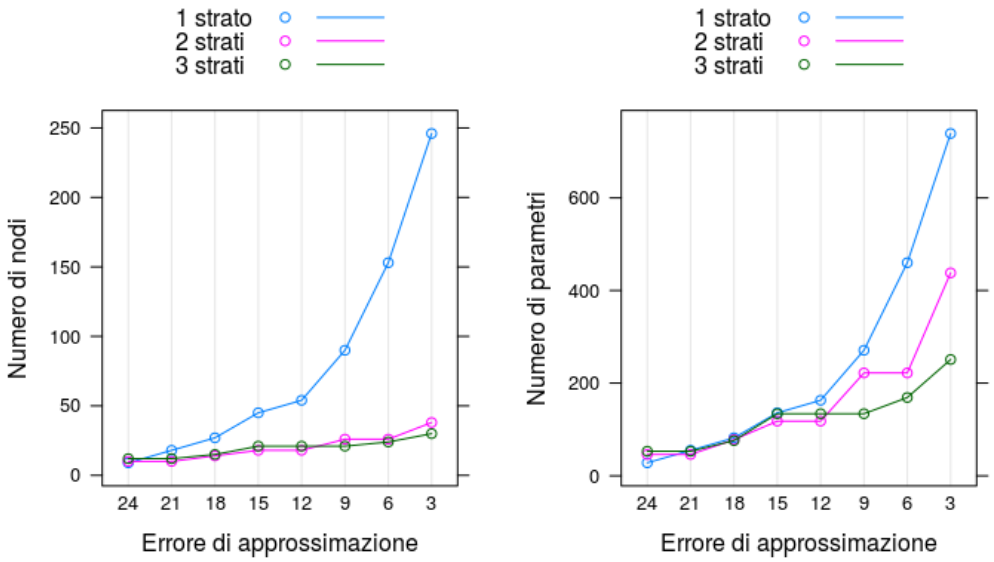

Figura 2.7: Errori di approssimazione del terzo studio di simulazione.

Riassumendo

In questo capitolo sono state confrontate le reti neurali a singolo strato e quelle multi-strato principalmente sotto due aspetti: la distorsione del modello e la capacità di adattamento all'insieme di dati di stima.

Riguardo alla distorsione del modello, i teoremi matematici, riportati in questo capitolo, dimostrano che approssimare una funzione, utilizzando una rete neurale multi-strato, porta ad un guadagno esponenziale, in termini di numero di nodi, rispetto ad una rete neurale a singolo strato. Spostando, invece, l'attenzione dal numero di nodi al numero di parametri del modello, utilizzare reti multi-strato rimane vantaggioso. All'aumentare del numero di nodi, la rete neurale multi-strato ha una crescita quadratica dei parametri, mentre in quella a singolo strato la crescita è lineare. Nonostante ciò, all'aumentare dei nodi di input, la crescita del numero di parametri è *quadratica* per una rete neurale multi-strato, ed *esponenziale* per una a singolo strato.

Riguardo alla capacità di adattamento all'insieme di dati, è stato mostrato, attraverso studi di simulazione, che per raggiungere un errore di stima fissato sono sufficienti meno parametri se vengono utilizzate reti neurali multi-strato. Sebbene il miglioramento rispetto alle reti a singolo strato sia evidente, il numero di strati latenti ottimale per reti multi-strato dipende dal problema in questione.

Infine, viene effettuato un confronto statistico tra la rete neurale multi-strato e il modello MARS. In particolare, è stato mostrato come il modello MARS sia un caso particolare di rete neurale multi-strato con funzioni di attivazione ReLU e con alcuni vincoli sui parametri. La rete neurale, costruita in questo modo, riesce a modellare solamente relazioni marginali se è composta da un solo strato latente, mentre è in grado di considerare interazioni tra variabili se è composta da più strati latenti.

Capitolo 3

Interpretazione statistica dei metodi di regolarizzazione

Il paragrafo 2.3 introduce l'errore quadratico medio (MSE), un importante indice che valuta l'accuratezza previsiva del modello. Come già annunciato, l'errore quadratico medio si può scomporre in *distorsione* e *varianza* del modello. Si supponga che i dati provengano dal modello $Y = f(X) + \varepsilon$, con $\mathbb{E}(\varepsilon) = 0$ e $Var(\varepsilon) = \sigma^2$. Tramite un modello di previsione, si ottengono i valori stimati $\hat{y} = \hat{f}(x)$ per i valori osservati x. La distorsione del modello misura la distanza tra la vera funzione $f(x)$ e la funzione stimata, al netto dell'insieme di dati, $b(\hat{y}) = f(x) - \mathbb{E}[\hat{y}]$. La varianza del modello corrisponde alla varianza delle previsioni, al cambiare dell'insieme di dati, $Var(\hat{y}) = \mathbb{E}[\hat{y}^2] - \mathbb{E}[\hat{y}]^2$. Queste due quantità sono caratterizzate da comportamenti opposti: la distorsione decresce se la complessità del modello aumenta, mentre la varianza cresce se la complessità del modello aumenta. La selezione del modello ottimale, in termini di accuratezza previsiva, deve essere condotta facendo un *compromesso* tra le due quantità appena descritte; questo è il motivo per cui, nel caso in cui si conosca il meccanismo generatore dei dati, $f(X)$, minimizzare l'MSE conduce alla scelta ottimale del modello. Nella pratica, tuttavia, non si è mai a conoscenza della vera relazione tra variabili osservate e fenomeno di interesse, rappresentata da $f(X)$. In questo caso, si può minimizzare l'*errore di previsione atteso*, o *expected prediction error*,

(Hastie et al. 2009, sez. 2.9), che è definito come

$$\begin{aligned}\text{EPE}(x) &= \mathbb{E}[(Y - \hat{f}(x))^2 | X = x] \\ &= (f(x) - \mathbb{E}[\hat{y}])^2 + Var(\hat{y}) + \mathbb{E}[(Y - f(x))^2] \\ &= \text{MSE}(x) + \sigma^2.\end{aligned}$$

Si può notare, tuttavia, che il termine ignoto σ^2 non dipende dai valori $\hat{y}$. Per questo motivo σ^2 viene chiamato *errore irriducibile* e determina il limite inferiore dell'EPE. Quindi, il vettore $\hat{y}$, che minimizza l'EPE, minimizza anche l'MSE. Una buona e sensata stima dell'EPE è l'*errore di previsione*, $\sum_{i=1}^{n}(y_i - \hat{y}_i)^2/n$, dove y sono i veri valori della variabile risposta e $\hat{y}$ sono i valori stimati (Hastie et al. 2009, sez. 2.9). Occorre, tuttavia, prestare attenzione ad un problema che può sorgere. Calcolare l'errore di previsione nello stesso insieme di dati, in cui sono state calcolate le stime, comporta un irreale ottimismo nelle previsioni (Hastie et al. 2009, sez. 7.4). Infatti, all'aumentare della complessità del modello, l'errore continua a scendere, poiché, più il modello è complesso, più questo riesce a cogliere piccole variazioni casuali dovute al rumore. In questo caso si parla di *sovradattamento* all'insieme di dati, in inglese "*overfitting*". La soluzione è quella di utilizzare un nuovo insieme di dati per calcolare l'errore di previsione. Nel nuovo insieme di dati, l'errore di previsione ed l'errore quadratico medio hanno lo stesso andamento, dal momento che l'errore irriducibile σ^2 rimane costante al variare della complessità del modello. Per approfondire il concetto di *selezione del modello* e *compromesso varianza-distorsione* si veda, ad esempio, Azzalini e Scarpa (2012, cap. 3) e Hastie et al. (2009, cap. 7).

Nel caso delle reti neurali multi-strato, la complessità del modello è stabilita dal numero di parametri, il quale è legato in modo deterministico al numero di nodi e di strati latenti. Nella pratica, infatti, vengono regolate queste ultime due quantità, perché risulta più semplice gestire la struttura della rete. Controllare la complessità del modello, tuttavia, non è l'unico modo per ottenere un compromesso tra varianza e distorsione. Per raggiungere tale scopo esistono altre tecniche, che sono chiamate *metodi di regolarizzazione* (Azzalini e Scarpa 2012, sez. 3.6.3), tra i quali saranno approfonditi, in questo capitolo, i *metodi di penalizzazione* ed il *dropout*, cioè i metodi più utilizzati nel *deep learning*.

Metodi di penalizzazione

Le reti neurali multi-strato sono una classe di modelli altamente parametrizzata; ciò significa che, non applicando alcun metodo di regolarizzazione, il modello tende ad avere una distorsione molto piccola e una varianza molto grande. Ridurre il numero di nodi, tuttavia, può compromettere la flessibilità della rete neurale. Una soluzione al problema è quella di stimare il modello imponendo dei vincoli ai parametri, con lo scopo di ridurre il loro valore e di comprimerli verso lo zero (Hastie et al. 2009, sez. 3.4). Un modo per ottenere le stime vincolate è quello di sfruttare la *forma lagrangiana* e penalizzare, così, la funzione di perdita. Operando in questo modo, si ottengono stime distorte dei parametri, a causa della penalità che è stata inserita, mentre la varianza del modello decresce, poiché le stime dei parametri hanno meno variazione. Questi sono chiamati, infatti, *metodi di penalizzazione* o di *shrinkage*. In questo caso, vengono considerati solamente i metodi di penalizzazione che effettuano una compressione dei parametri verso lo zero.

Sia $\mathbf{W}$ il tensore tridimensionale di parametri della rete neurale multistrato. Applicando il metodo di penalizzazione, la stima dei parametri $\hat{\mathbf{W}}$, rappresentata nell'equazione (1.5), diventa quindi

$$\hat{\mathbf{W}} = \arg\min_{\mathbf{W}} \left\{ \frac{1}{n} \sum_{i=1}^{n} L[y_i, f(x_i; \mathbf{W})] + \lambda J(\mathbf{W}) \right\}, \tag{3.1}$$

dove $J(\mathbf{W})$ è un *termine di regolarizzazione* non negativo, mentre $\lambda \geq 0$ è un *parametro di regolarizzazione*. Maggiore è il valore di λ, maggiore è la compressione dei parametri verso lo zero. Di conseguenza, aumentando il valore di λ, la distorsione aumenta e la varianza diminuisce. Il parametro λ viene scelto in modo da minimizzare l'EPE. Utilizzare tecniche come la convalida incrociata, in questo caso, è altamente sconsigliato, in quando comporterebbe un consumo computazionale esorbitante. Generalmente all'intercetta non viene applicato la compressione.

Dal momento che viene utilizzato il *mini-batch gradient descent* per l'aggiornamento dei parametri, ad ogni iterazione della *backpropagation* (par. 1.2) la funzione da minimizzare non è calcolata su tutte le n osservazioni, ma solamente su un suo sottoinsieme. Bengio (2012) consiglia, quindi, di pre-

moltiplicare il parametro λ per la quantità b/n, dove b è la numerosità del *mini-batch*. In questo modo, la stima di λ è indipendente dalla numerosità b. Se n non è divisibile per b e l'ultimo sottoinsieme ha numerosità $b' < b$, il parametro λ nell'ultimo sottoinsieme deve essere pre-moltiplicato per b'/n.

Esistono molti tipi di penalità (Mohamed 2015); quelle maggiormente utilizzate sono:

- *weight decay*, o penalità L_2, approfondita nel paragrafo 3.1.1;
- penalità L_1, o *lasso*, con $J(\mathbf{W}) = \|\mathbf{W}\|_1$;
- penalità L_p, con $J(\mathbf{W}) = \|\mathbf{W}\|_p^p$;
- *elastic net*, in cui ci sono due parametri di regolarizzazione, ed il termine da sommare è $\frac{\lambda_1}{2}\|\mathbf{W}\|_2^2 + \lambda_2\|\mathbf{W}\|_1$, dove λ_1 viene dimezzata solamente per semplicità di calcolo;
- *total variation*, con $J(\mathbf{W}) = \|\Delta\mathbf{W}\|_1$, dove Δ è l'*operatore Delta* che indica una variazione; in questo modo viene applicato il *lasso* a coppie di parametri, le quali saranno vincolati ad essere uguali, riducendo la dimensionalità dello spazio parametrico;
- *fused lasso*, in cui vengono combinati *lasso* e *total variation*, $\lambda_1\|\mathbf{W}\|_1 + \lambda_2\|\Delta\mathbf{W}\|_1$;
- *group lasso*, introdotto da Scardapane et al. (2016), applica il *lasso* ai nodi, ottenendo così una riduzione della struttura della rete; ciò può essere raggiunto vincolando a zero alcune righe delle matrici di parametri;
- *group sparse lasso*, è un estensione del *group lasso*, che mira a creare sparsità, sia a livello di parametri, che a livello di nodi; ciò è possibile combinando il *lasso* con il *group lasso*.

In seguito, verrà approfondita e discussa la penalità *weight decay*, la quale è molto utilizzata nelle reti neurali.

La penalità *weight decay*

La versione più popolare tra le penalità usate nelle reti neurali è il *weight decay*, o penalità L_2, introdotta da Krogh e Hertz (1991). In questo caso, la penalità è composta dalla norma quadratica di $\mathbf{W}$, il tensore tridimensionale dei parametri:

$$\begin{aligned} J(\mathbf{W}) &= \frac{1}{2}\|\mathbf{W}\|_2^2 \\ &= \frac{1}{2}\sum_{l=1}^{L-1}\|W^{(l)}\|_2^2 \\ &= \frac{1}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{p_l}\sum_{j=1}^{p_{l+1}}\left(w_{ij}^{(l)}\right)^2. \end{aligned}$$

La penalità, costruita in questo modo, è utilizzata anche in contesti diversi dalle reti neurali. Applicando, ad esempio, la penalità L_2 alla regressione lineare, si da origine alla *regressione Ridge* (Bishop 2006, cap. 3).

Nella fase di aggiornamento dei parametri della rete neurale (si veda par. 1.3), viene utilizzato il gradiente della funzione di perdita calcolato per n osservazioni. Con l'aggiunta della penalità *weight decay*, la funzione di perdita diventa

$$\frac{1}{n}\sum_{i=1}^{n} L\left[y_i, f(x_i;\mathbf{W})\right] + \frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{p_l}\sum_{j=1}^{p_{l+1}}\left(w_{ij}^{(l)}\right)^2, \tag{3.2}$$

il cui gradiente, rispetto al peso $w_{ij}^{(l)}$, è

$$\frac{1}{n}\sum_{i=1}^{n}\frac{\partial L[y_i, f(x_i;\mathbf{W})]}{\partial w_{ij}^{(l)}} + \lambda w_{ij}^{(l)}.$$

Utilizzando la forma matriciale, il gradiente dell'equazione (3.2), rispetto alla matrice di parametri $W^{(l)}$, è

$$\frac{1}{n}\sum_{i=1}^{n}\frac{\partial L[y_i, f(x_i;\mathbf{W})]}{\partial W^{(l)}} + \lambda W^{(l)}.$$

Nella fase di stima, l'aggiornamento dei parametri tramite la discesa del gradiente, formulato nell'equazione (1.9), diventa semplicemente

$$W_{t+1}^{(l)} = W_t^{(l)} - \eta\left(\Delta L(W_t^{(l)}) + \lambda W_t^{(l)}\right).$$

Usando l'ottimizzatore *Adam* per effettuare la discesa del gradiente, l'aggiornamento dei parametri, definito dall'equazione (1.10), diventa

$$w_{t+1,ij}^{(l)} = w_{t,ij}^{(l)} - \eta \left(\frac{\tilde{v}_{t,ij}}{\sqrt{\tilde{m}_{t,ij} + \varepsilon}} + \lambda w_{ij}^{(l)} \right).$$

Ciò significa che è possibile stimare il modello con la penalità *weight decay* senza costi computazionali aggiuntivi, ma semplicemente sommando un termine nell'equazione di aggiornamento dei parametri.

La stima dei parametri ottenuta minimizzando la funzione di perdita, penalizzata con il *weight decay*, ha un'interpretazione bayesiana. Nel prossimo paragrafo verranno fornite alcune informazioni riguardo ad un modello bayesiano, nel suo caso generale.

Un modello bayesiano

Si consideri un generico modello statistico, in cui θ è il parametro di interesse, $x = (x_1, ..., x_p)$ sono le p variabili esplicative, mentre $y = (y_1, ..., y_n)$ è la variabile risposta, con funzione di densità $p_Y(y|\theta; x)$. Supponendo che le osservazioni siano indipendenti ed identicamente distribuite, la verosimiglianza del modello è

$$\mathcal{L}(\theta; x, y) = p_Y(y_1, ..., y_n|\theta; x) = \prod_{i=1}^{n} p_{Y_i}(y_i|\theta; x).$$

In ambito bayesiano, i parametri sono variabili aleatorie e ammettono una loro distribuzione. Grazie al teorema di Bayes, è possibile ricavare la distribuzione *a posteriori*, ossia la distribuzione di θ condizionata ai dati y:

$$\pi(\theta|y, x) = \frac{\pi(\theta)p(y|\theta; x)}{p(y)},$$

dove $\pi(\theta)$ è la distribuzione *a priori* di θ, cioè una sintesi di θ prima di osservare i dati. La quantità $p(y) = \int_\Theta \pi(\theta)p(y|\theta; x)d\theta$ è un termine costante, in quanto non dipende dal parametro, perciò

$$\pi(\theta|y, x) \propto \pi(\theta)p(y|\theta; x).$$

La distribuzione a posteriori è, quindi, un compromesso tra la distribuzione a priori e la componente empirica. Spesso, per semplicità, viene utilizzata la *log-posteriori*, la quale è esprimibile come

$$\log(\pi(\theta|y,x)) \propto \log(\pi(\theta)) + l(\theta;x,y),$$

dove $l(\theta;x,y) = \log(\mathcal{L}(\theta;x,y))$ è la funzione di log-verosimiglianza.

In un contesto bayesiano, la previsione della nuova osservazione y', basata sulle variabili esplicative x', si ricava molto facilmente come

$$\begin{aligned} p(y'|x';x,y) &= \int_{\Theta} p(y',\theta|x')d\theta \\ &= \int_{\Theta} p(y'|\theta;x')\pi(\theta|x,y)d\theta, \end{aligned} \tag{3.3}$$

supponendo indipendenza condizionata tra y' e y dato θ (Gelman et al. 2014, sez. 1.3).

Interpretazione bayesiana del *weight decay*

In una rete neurale multi-strato, si ipotizzi l'utilizzo di una funzione di perdita che costituisca il nucleo di una funzione di verosimiglianza. Ad esempio, per un problema di regressione, l'utilizzo dell'errore quadratico medio (MSE) corrisponde a massimizzare la devianza della distribuzione normale, mentre per un problema di classificazione, l'utilizzo della cross-entropia corrisponde a massimizzare la devianza della distribuzione multinomiale (Hastie et al. 2009, sezione 2.6.3). Supponendo che le osservazioni siano indipendenti ed identicamente distribuite, vale, in questo caso, la seguente relazione di proporzionalità

$$l(\mathbf{W};y) \propto -\sum_{i=1}^{n} L[y_i, f(x_i;\mathbf{W})], \tag{3.4}$$

dove $l(\mathbf{W};y)$ è la funzione di log-verosimiglianza, rispetto ai parametri $\mathbf{W}$. Si può dimostrare che la stima dei parametri di una rete neurale, la quale utilizza la penalità *weight decay*, corrisponde alla moda a posteriori (MAP) di un opportuno modello bayesiano (Bishop 2006, sez. 5.5). Si supponga come distribuzione a priori per i parametri

$$\mathbf{W} \sim \mathcal{N}_{dim(\Omega)}\left(\mathbf{0}, \frac{1}{\lambda}\mathbf{I}_{dim(\Omega)}\right),$$

dove $dim(\Omega)$ è la dimensionalità dello spazio parametrico, ovvero la dimensionalità del *tensore* tridimensionale $\mathbf{W}$; perciò $\pi(\mathbf{W}) \propto \exp\left(-\frac{\lambda}{2}\|\mathbf{W}\|_2^2\right)$. Sfruttando l'informazione fornita dall'espressione (3.4), la log-posteriori risulta essere

$$\log(\pi(\mathbf{W}|y)) \propto -\sum_{i=1}^{n} L[y_i, f(x_i; \mathbf{W})] - \frac{\lambda}{2}\|\mathbf{W}\|_2^2,$$

la cui moda a posteriori è

$$\begin{aligned} \text{MAP}(\mathbf{W}) &= \arg\max_{\mathbf{W}} \left\{ -\frac{1}{n}\sum_{i=1}^{n} L[y_i, f(x_i; \mathbf{W})] - \frac{\lambda}{2}\|\mathbf{W}\|_2^2 \right\} \\ &= \arg\min_{\mathbf{W}} \left\{ \frac{1}{n}\sum_{i=1}^{n} L[y_i, f(x_i; \mathbf{W})] + \frac{\lambda}{2}\|\mathbf{W}\|_2^2 \right\}, \end{aligned}$$

ottenendo un caso particolare della formulazione (3.1), in cui la penalità è il *weight decay*.

Altri metodi di regolarizzazione

I metodi di penalizzazione, analizzati nel paragrafo 3.1, sono applicabili a molte classi di modelli statistici. Questi metodi, tuttavia, non sono l'unica soluzione possibile per trovare un compromesso tra varianza e distorsione, e risolvere, così, il problema dell'*overfitting*. In seguito verranno descritti brevemente altri metodi di regolarizzazione, applicabili alle reti neurali multistrato. Al *dropout* verrà dedicata, successivamente, un'intera sezione.

Batch normalization

Generalmente, nella procedura di *backpropagation*, i parametri vengono inizializzati in modo casuale: in genere vengono campionati da una distribuzione di media nulla e varianza unitaria. Dopo molte iterazioni, la media e la varianza dei valori dei parametri tende a cambiare. La *batch normalization* (Ioffe e Szegedy 2015) normalizza i parametri ad ogni *mini-batch*, in modo da riportarli ad avere media nulla e varianza unitaria. Così facendo si evita di ottenere stime molto lontane da zero, risolvendo, in parte, il problema dell'*overfitting*.

Early stopping

Dopo un certo numero di epoche della *backpropagation*, l'errore nell'insieme di stima non ha più miglioramenti sostanziali e l'errore nell'insieme di verifica comincia a risalire. In questo caso il modo più semplice, per risolvere il problema dell'*overfitting*, è bloccare la procedura di stima, in modo da ottenere l'errore di predizione minimo. Molti ebbero l'idea di applicare l'*early stopping* alla rete neurale, come, ad esempio, Finnoff et al. (1993).

Max-norm

Il metodo *max-norm* (Srebro e Shraibman 2005) consiste nel vincolare la norma quadratica del vettore di parametri, che entra in ogni nodo, ad essere limitata superiormente da un valore c fissato. Sia $\mathbf{w}_i^{(l)}$ la i-esima colonna della matrice di parametri che esce dall'l-esimo strato. Imponendo il vincolo $\|\mathbf{w}_i^{(l)}\|_2 < c$, si proietta $\mathbf{w}_i^{(l)}$ sulla superficie di un ipersfera di raggio c. Ciò implica che il valore massimo, che la norma del vettore di parametri può assumere, è c. La quantità c è un parametro di regolazione.

Il metodo *dropout*

Nell'ambito dell'analisi di dati e del *machine learning*, si è diffuso l'utilizzo della *combinazione di classificatori*. L'idea di base è quella di stimare più volte un modello di classificazione, apportando ogni volta delle modifiche all'insieme di dati di stima, ed utilizzare il *voto di maggioranza* per migliorare l'accuratezza previsiva. La combinazione è possibile, sebbene sia meno usata, anche per modelli di regressione. Per un approfondimento, consultare Azzalini e Scarpa (2012).

Generalmente, la combinazione risulta molto efficace se i modelli utilizzati hanno, tra loro, una struttura molto diversa; questo è il motivo per cui viene spesso applicata a classi di modelli, che sono molto sensibili a cambiamenti dei dati di stima. Un modello che utilizza una combinazione di classificatori è la *random forest* (Breiman 2001), che combina alberi di decisione. Ogni volta che viene stimato un albero, oltre ad essere effettuato un campionamento con reinserimento dei dati di stima, viene selezionato anche un sottoinsieme

delle variabili esplicative. Ciò accentua la differenza tra i classificatori, la cui combinazione risulta, quindi, più efficace. Uno dei punti di forza della *random forest* è che "non può sovra-adattarsi" ai dati di stima (Hastie et al. 2009, sezione 15.3.4). In altre parole, applicare la *random forest* a modelli sovra-adattati risolve il problema dell'*overfitting*. Questo accade perché l'utilizzo del voto di maggioranza, che coincide con la media delle probabilità predette, riduce la varianza delle previsioni.

La rete neurale multi-strato è una classe di modelli adeguata, per essere utilizzata nei metodi appena citati. Infatti, grazie all'elevato numero di parametri, piccoli cambiamenti nei dati modificano completamente la struttura della rete. In presenza di strati latenti multipli, tuttavia, l'utilizzo di combinazione della classificatori è purtroppo improponibile, a causa dell'elevato peso computazionale.

Ragionando, però, sulla struttura della rete e sulla sua procedura di stima, Hinton et al. (2012) introdussero un innovativo metodo di regolarizzazione: il *dropout*. Questo metodo è un'approssimazione del risultato che si otterrebbe attraverso la combinazione di classificatori ed è nato pensando alla logica del campionamento casuale delle variabili della *random forest*. La tecnica del *dropout* consiste nel porre, ad ogni iterazione della procedura di *backpropagation*, una porzione di nodi pari a zero. Questa porzione viene scelta casualmente. In questo modo, non solo viene risolto il problema dell'*overfitting*, ma si ha anche un netto miglioramento dell'accuratezza previsiva. Grazie a queste qualità, al *dropout* è attribuito gran parte del merito della diffusione del *deep learning*.

La logica del *dropout*

Il *dropout* è una tecnica che combina, in modo efficiente, molte reti neurali con diversa struttura. Ad ogni iterazione dell'algoritmo di *backpropagation* alcuni nodi, imponendoli a zero, vengono temporaneamente rimossi dal modello. Ciascun nodo viene conservato nel modello con una probabilità p fissata. Quindi, la parte di nodi rimossi e quella di nodi conservata è casuale e diversa ad ogni iterazione. La probabilità p è un parametro di regolazione

e può essere diverso per ogni strato, nonostante venga adottato molto spesso il caso più semplice, ossia p uguale in tutta la rete.

Con l'applicazione del *dropout*, il j-esimo nodo dell'$(l+1)$-esimo strato latente è ricavato nel seguente modo:

$$r_i^{(l)} \text{ realizzazione della v.c. } R \sim Bernoulli(p), \tag{3.5}$$

$$z_j^{(l+1)} = w_{0j}^{(l)} + \sum_{i=1}^{p_l} r_i^{(l)} w_{ij}^{(l)} a_i^{(l)},$$

$$a_j^{(l+1)} = g^{(l+1)}(z_j^{(l+1)}).$$

Applicando il *dropout*, quindi, viene selezionata una rete "ridotta", costituita solamente dai nodi non nulli. Si può pensare ad ognuna di queste reti "ridotte", come una diversa configurazione della rete originale. Nella procedura di stima, il gradiente si propaga solo attraverso la rete "ridotta", ossia attraverso i nodi non nulli. Questo perché, ai nodi nulli corrisponde un gradiente pari a zero e, di conseguenza, un contributo nullo nella fase di aggiornamento dei parametri.

Secondo la classica procedura di combinazione di classificatori, ognuna di queste reti "ridotte" dovrebbe essere stimata indipendentemente. È stato dimostrato empiricamente che si ottiene un risultato molto simile proponendo una diversa configurazione della rete, ad ogni passo di aggiornamento dei parametri. Così facendo, si gode di un vantaggio: stimando una singola rete vengono ridotti i costi computazionali.

Nella fase di previsione, si utilizza la struttura della rete originale senza il *dropout*, in cui i parametri stimati vengono pre-moltiplicati per il valore p. Questo viene effettuato in modo che, condizionatamente ai dati, il valore atteso delle previsioni, $\mathbb{E}[f(\mathbf{x}', \mathbf{W}_d)]$, coincida con il valore atteso dei valori stimati dal modello, $\mathbb{E}[f(\mathbf{x}, \mathbf{W}_d)]$. In questo caso, $\mathbf{x}$ è il vettore di dati dell'insieme di stima, $\mathbf{x}'$ indica il nuovo vettore di osservazioni, di cui si vuole fare la previsione, e $\mathbf{W}_d$ è il tensore di parametri, a cui è applicato il *dropout*. Il modello stimato si presenta come

$$f(\mathbf{x}; \mathbf{W}_d) = g^{(L)}(W^{(L-1)}(\mathbf{r}^{(L-1)} \circ g^{(L-1)}(\cdots W^{(2)}(\mathbf{r}^{(2)} \circ g^{(2)}(W^{(1)}(\mathbf{r}^{(1)} \circ \mathbf{x}))))))),$$

dove il vettore $\mathbf{r}^{(l)}$ è la realizzazione della variabile casuale $R \sim Bernoulli(p)$, con $\mathbb{E}[R] = p$. Se $\mathbf{r}^{(l)}$ non è osservata il valore atteso dei valori stimati dal

modello è

$$\begin{aligned}\mathbb{E}\left[f(\mathbf{x};\mathbf{W}_d)\right] &= \mathbb{E}\left[g^{(L)}(W^{(L-1)}(R\circ g^{(L-1)}(\cdots W^{(2)}(R\circ g^{(2)}(W^{(1)}(R\circ\mathbf{x}))))))\right]\\ &= g^{(L)}(W^{(L-1)}(\mathbb{E}\left[R\right]\circ g^{(L-1)}(\cdots W^{(2)}(\mathbb{E}\left[R\right]\circ g^{(2)}(W^{(1)}(\mathbb{E}\left[R\right]\circ\mathbf{x}))))))\\ &= g^{(L)}(pW^{(L-1)}g^{(L-1)}(\cdots pW^{(2)}g^{(2)}(pW^{(1)}\mathbf{x}))),\end{aligned} \tag{3.6}$$

con p scalare. Sostituendo $\mathbf{x}$ con $\mathbf{x}'$, il valore atteso delle previsioni diventa

$$\mathbb{E}\left[f(\mathbf{x}';\mathbf{W}_d)\right] = g^{(L)}(pW^{(L-1)}g^{(L-1)}(\cdots pW^{(2)}g^{(2)}(pW^{(1)}\mathbf{x}'))).$$

In fase di previsione, quindi, si utilizza la rete con tutti i nodi, scalando i parametri stimati per la probabilità p.

L'applicazione pratica del *dropout*

In un articolo successivo, Srivastava et al. (2014), gli autori del *dropout*, fornirono alcuni consigli pratici sull'utilizzo di questo metodo di regolazione:

- Il *dropout* esprime tutta la sua efficacia con un elevato numero di osservazioni; studi empirici hanno dimostrato che, se le osservazioni sono poche (qualche centinaio), la rete regolata con *dropout* può fornire un'accuratezza previsiva peggiore.

- Sia N il numero totale di nodi nella rete; è stato dimostrato empiricamente che, per regolare il parametro p, è conveniente tenere fissa la quantità $p \times N$: al variare di p è opportuno, quindi, cambiare il numero di nodi della rete, in modo che il numero di nodi non nulli attesi rimanga costante.

- L'operazione di regolazione del parametro p può risultare oneroso dal punto di vista computazionale e, a volte, conviene fissare *a priori* un valore per p; studi empirici, praticati dagli stessi autori, dimostrarono che un valore ottimale per p è compreso tra 0.5 e 1; spesso viene utilizzato $p = 0.5$.

- Per le *convolutional neural networks*, una classe di reti neurali che verrà descritta nel capitolo 4, gli autori consigliano di utilizzare valori diversi

di p per ogni strato; in particolare, spostandosi dai nodi di input a quelli di output, è opportuno usare valori di p decrescenti e compresi tra 0.5 e 1.

- Gli autori consigliano di applicare il *dropout* anche ai nodi di input, con probabilità p più grande rispetto ai nodi degli strati latenti; un valore consigliato è $p = 0.8$.

- Il *dropout* funziona particolarmente bene se è accompagnato dal metodo di regolarizzazione *max-norm* (sez. 3.2), con c tipicamente scelto tra 3 e 4;

- Questa tecnica non necessita dell'*early stopping* (sez. 3.2), perché un elevato numero di epoche non comporta *overfitting*.

- Gli autori consigliano l'utilizzo di valori più grandi del parametro *learning rate* (sez. 1.3), in modo da raggiungere regioni dello spazio parametrico che sarebbero, altrimenti, difficilmente esplorabili.

- Nella formula (3.6), è possibile, in alternativa, campionare $r_i^{(l)}$ da $R \sim \mathcal{N}(1, \sigma^2)$, con σ^2 parametro di regolazione.

Interpretazione bayesiana del *dropout*

Nel paragrafo precedente si è visto come il *dropout* sia un metodo efficace nel risolvere il problema dell'*overfitting* e nel migliorare l'accuratezza previsiva della rete neurale. Hinton et al. (2012), quando presentarono la prima applicazione di questa tecnica, ne dimostrarono l'efficacia solamente attraverso studi empirici, senza fornire una valida spiegazione statistica del funzionamento di questo metodo.

In seguito, Wager et al. (2013) reinterpretarono il *dropout* come un metodo di penalizzazione. In particolare, gli autori dimostrarono che le stime dei parametri, ottenute con il *dropout*, coincidono con quelle ricavate con la penalità *weight decay*, applicata dopo aver standardizzato le variabili esplicative. In questo caso, il parametro di regolazione λ del *weight decay* corrisponde a $\dfrac{1-p}{p}$, dove p è la probabilità che il *dropout* conservi un generico nodo.

Successivamente, Gal e Ghahramani (2015d) dettero ai parametri, stimati con il *dropout*, un'interpretazione statistica. In particolare, queste stime sono un'approssimazione della moda a posteriori (MAP) di un modello bayesiano, il quale utilizza, come distribuzione a priori dei parametri, una normale standard multivariata. Si tratta di un'approssimazione, perché la distribuzione a posteriori è ricavata attraverso l'*inferenza variazionale*. Il prossimo paragrafo sintetizza i punti principali di questo approccio.

La stima del modello via *variational Bayes*

Si consideri un modello bayesiano come quello presentato al paragrafo 3.1.1. Quando la distribuzione a posteriori è molto complessa, utilizzare metodi inferenziali bayesiani considerando direttamente $\pi(\theta|x,y)$ risulta molto difficile. Quindi, a volte, conviene considerare la cosiddetta *distribuzione variazionale* $q(\theta|\omega)$, cioè una distribuzione con una struttura molto più semplice, che approssima $\pi(\theta|x,y)$ (Blei et al. 2016). La funzione $q(\cdot)$ è scelta in modo da semplificare l'inferenza. Inoltre, $q(\theta|\omega)$ è dotata di un parametro ω, il quale viene ottimizzato in modo da approssimare $\pi(\theta|x,y)$ nel miglior modo possibile. L'idea di base consiste nel minimizzare la divergenza di Kullback-Leibler (KL) rispetto a ω (Kullback e Leibler 1951), la quale può essere definita come

$$\begin{aligned}
\mathrm{KL}(q(\theta|\omega)||\pi(\theta|x,y)) &= \int_\Theta q(\theta|\omega)\log\left(\frac{q(\theta|\omega)}{\pi(\theta|x,y)}\right)d\theta \\
&= \int_\Theta q(\theta|\omega)\log\left(\frac{q(\theta|\omega)p(y|x)}{\pi(\theta)\mathcal{L}(\theta;x,y)}\right)d\theta \\
&= \int_\Theta q(\theta|\omega)\log\left(p(y|x)\right)d\theta - \int_\Theta q(\theta|\omega)l(\theta;x,y)d\theta \\
&\quad - \int_\Theta q(\theta|\omega)\log\left(\frac{\pi(\theta)}{q(\theta|\omega)}\right)d\theta
\end{aligned} \tag{3.7}$$

e rappresenta una sorta di similarità tra due distribuzioni. L'espressione $\mathrm{KL}(q(\theta|\omega)||\pi(\theta|x,y))$ indica la distanza di Kullback-Leibler tra le distribuzioni $q(\theta|\omega)$ e $\pi(\theta|x,y)$. Grazie alla diseguaglianza di Jensen (Jensen 1906),

è possibile ottenere un limite inferiore per la log-verosimiglianza $l(\theta; x, y)$:

$$\begin{aligned} l(\theta; x, y) = \log(p(y|\theta; x)) &= \log \int_\Theta p(y, \theta|x) d\theta \\ &= \log \int_\Theta p(y, \theta|x) \frac{q(\theta|\omega)}{q(\theta|\omega)} d\theta \\ &= \log \mathbb{E}_{q(\theta|\omega)} \left[\frac{p(y, \theta|x)}{q(\theta|\omega)} \right] \\ &\geq \mathbb{E}_{q(\theta|\omega)} \left[\log \left(\frac{p(y, \theta|x)}{q(\theta|\omega)} \right) \right] = \mathcal{K}(\omega). \end{aligned}$$

Il limite inferiore della log-verosimiglianza, $\mathcal{K}(\omega)$, può essere definito come

$$\begin{aligned} \mathcal{K}(\omega) &= \mathbb{E}_{q(\theta|\omega)} \left[\log \left(\frac{p(y, \theta|x)}{q(\theta|\omega)} \right) \right] \\ &= \mathbb{E}_{q(\theta|\omega)} \left[\log(p(y, \theta|x)) \right] - \mathbb{E}_{q(\theta|\omega)} \left[\log(q(\theta|\omega)) \right] \\ &= \int_\Theta q(\theta|\omega) \log(p(y, \theta|x)) d\theta - \mathbb{E}_{q(\theta|\omega)} \left[\log(q(\theta|\omega)) \right] \\ &= \int_\Theta q(\theta|\omega) \log \left(\mathcal{L}(\theta; x, y) \pi(\theta) \right) d\theta - \mathbb{E}_{q(\theta|\omega)} \left[\log(q(\theta|\omega)) \right] \\ &= \int_\Theta q(\theta|\omega) l(\theta; x, y) d\theta + \int_\Theta q(\theta|\omega) \log \left(\pi(\theta) \right) d\theta - \int_\Theta q(\theta|\omega) \log(q(\theta|\omega)) d\theta \\ &= \int_\Theta q(\theta|\omega) l(\theta; x, y) d\theta + \int_\Theta q(\theta|\omega) \log \left(\frac{\pi(\theta)}{q(\theta|\omega)} \right) d\theta \\ &= \int_\Theta q(\theta|\omega) l(\theta; x, y) d\theta - \mathrm{KL}(q(\theta|\omega)||\pi(\theta)). \end{aligned} \tag{3.8}$$

Si noti, dalla terza uguaglianza di (3.7) e dalla sesta uguaglianza di (3.8), che vale la relazione

$$\mathrm{KL}(q(\theta|\omega)||\pi(\theta|x, y)) = -\mathcal{K}(\omega) + costante,$$

quindi, minimizzare $\mathrm{KL}(q(\theta|\omega)||\pi(\theta|x, y))$, rispetto a ω, equivale a massimizzare $\mathcal{K}(\omega)$. Di conseguenza, la distribuzione variazionale, che più si avvicina alla distribuzione a posteriori, si ottiene come

$$\tilde{q}(\theta|\omega) = \arg\max_{q(\theta|\omega) \in \mathcal{Q}} (\mathcal{K}(\omega)).$$

Applicando questo risultato all'equazione (3.6), si possono ottenere le previsioni approssimate per y' come

$$p(y'|x'; x, y) \approx \int_{\Theta} p(y'|\theta; x')\tilde{q}(\theta|\omega)d\theta. \tag{3.9}$$

Il *dropout* come approssimazione bayesiana

Le reti neurali stimate con metodi bayesiani sono state ampiamente studiate in letteratura; alcuni riferimenti utili, ad esempio, sono Neal (1996) e Denison (2002). Nell'ambito del *deep learning*, invece, l'utilizzo di metodi di stima bayesiani è, al momento, molto limitato, a causa dell'elevato costo computazionale che tali metodi comportano (Wang e Yeung 2016).

In seguito, sarà dimostrato che l'inferenza variazionale, applicata ad una rete neurale multi-strato, coincide con il processo di *dropout*. Così facendo, oltre a dare un'interpretazione statistica alle stime calcolate con il *dropout*, si dimostra che questo metodo di regolarizzazione approssima l'inferenza bayesiana, con un costo computazionale molto minore.

Come supporto per la stesura di questo paragrafo, verrà utilizzato il lavoro svolto da Gal e Ghahramani (2015b), i quali hanno dimostrato che l'inferenza variazionale applicata ad un processo gaussiano coincide con il *dropout*. In questa tesi, invece, l'applicazione verrà effettuata su una rete neurale multi-strato. Inoltre, si supponga che la log-verosimiglianza sia penalizzata con il *weight decay* (sez. 3.1.1).

In una rete neurale multi-strato, si ipotizzi l'utilizzo di un modello bayesiano come quello presentato in sezione 3.1.1. Si ponga una normale standard multivariata come una distribuzione a priori per i parametri, $\pi(\mathbf{W})$. Quindi, il generico parametro dell'l-esimo strato è distribuito come

$$w_{ij}^{(l)} \sim \mathcal{N}(0, 1).$$

La distribuzione a posteriori dei parametri, può essere definita secondo questa relazione:

$$\pi(\mathbf{W}|x, y) \propto \mathcal{L}(\mathbf{W}; x, y)\pi(\mathbf{W}).$$

La distribuzione $\pi(\mathbf{W}|x, y)$ è stata oggetto di studio da parte di molti autori, come, ad esempio, Graves (2011) e Hernández-Lobato e Adams (2015).

Verrà, ora, applicato il metodo *variational Bayes* per ricavare $\pi(\mathbf{W}|x, y)$. Per semplicità, verrà modificata la notazione usata finora, in modo da dividere l'intercetta dal resto dei parametri. Sia

- $\mathbf{W} = [W^{(1)}W^{(2)} \cdots W^{(L)}]$ il tensore tridimensionale di parametri della rete;
- $\mathbf{b} = [\mathbf{b}^{(1)}\mathbf{b}^{(2)} \cdots \mathbf{b}^{(L)}]$ il tensore bidimensionale delle intercette della rete;
- $W^{(l)} = [\mathbf{w}_1^{(l)}\mathbf{w}_2^{(l)} \cdots \mathbf{w}_{p_l}^{(l)}]$ la matrice di parametri che lega lo strato l allo strato $l + 1$;
- $\mathbf{b}^{(l)} = [b_1^{(l)}b_2^{(l)} \cdots b_{p_{l+1}}^{(l)}]^T$ il vettore di intercette che entra nello strato $l + 1$;
- $\mathbf{w}_i^{(l)} = [w_{i1}^{(l)}w_{i2}^{(l)} \cdots w_{ip_{l+1}}^{(l)}]^T$ l'i-esima colonna della matrice di parametri l.

Si assuma, inoltre, indipendenza tra le componenti della *distribuzione variazionale*, in modo da poter definire

$$q(\mathbf{W}) = \prod_{l=1}^{L}\prod_{i=1}^{p_l} q(\mathbf{w}_i^{(l)}); \qquad q(\mathbf{b}) = \prod_{l=1}^{L} q(\mathbf{b}^{(l)}); \qquad q(\mathbf{W}, \mathbf{b}) = q(\mathbf{W})q(\mathbf{b}).$$

Questa operazione viene effettuata spesso, perché semplifica molto la procedura inferenziale (Gelman et al. 2014, sez. 13.8).

La classe di distribuzioni scelta da Gal e Ghahramani (2015b) per $q(\cdot)$ è la mistura di due normali multivariate. Dunque, si può definire

$$r \text{ realizzazione della v.c. } R \sim Bernoulli(p),$$

$$\mathbf{w}_i^{(l)}|r = 0 \sim \mathcal{N}_{p_{l+1}}\left(\mathbf{0}, \sigma^2\mathbf{I}_{p_l}\right), \tag{3.10}$$

$$\mathbf{w}_i^{(l)}|r = 1 \sim \mathcal{N}_{p_{l+1}}\left(\mathbf{m}_i^{(l)}, \sigma^2\mathbf{I}_{p_l}\right), \tag{3.11}$$

mentre, per l'intercetta, $q(\cdot)$ è una normale multivariata, definita come

$$\mathbf{b}^{(l)} \sim \mathcal{N}_{p_{l+1}}\left(\mathbf{m}^{(l)}, \sigma^2\mathbf{I}_{p_l}\right). \tag{3.12}$$

Si indichi con $\mathbf{M}$ il tensore di tutti i parametri della distribuzione variazionale. Il limite inferiore della log-verosimiglianza, espresso dall'equazione (3.8), è in questo caso

$$\mathcal{K}(\mathbf{M}) = \int q(\mathbf{W}, \mathbf{b}) \log\left(p(y|\mathbf{W}, \mathbf{b}; x)\right) d\mathbf{W} d\mathbf{b} - \mathrm{KL}\left(q(\mathbf{W}, \mathbf{b})||\pi(\mathbf{W}, \mathbf{b})\right). \tag{3.13}$$

Massimizzare l'equazione (3.13), rispetto a $\mathbf{M}$, in modo analitico è impossibile, dal momento che la distribuzione variazionale è contenuta all'interno dell'integrale. Quindi, il procedimento di massimizzazione, che verrà attuato, consiste nell'iterare, fino a convergenza, la stima dell'integrale e la massimizzazione di $\mathcal{K}(\mathbf{M})$ (si veda l'algoritmo 2). Per effettuare la stima dell'integrale viene applicata l'integrazione Monte Carlo con un singolo campione, estraendo $\hat{\mathbf{W}}$ e $\hat{\mathbf{b}}$ dalla distribuzione $q(\mathbf{W}, \mathbf{b})$. Si ottiene, così una stima non distorta dell'equazione (3.13), la quale si presenta come

$$\begin{aligned} \mathcal{K}(\mathbf{M}) &= \log\left(p(y|\hat{\mathbf{W}}, \hat{\mathbf{b}}; x)\right) - \mathrm{KL}\left(q(\mathbf{W}, \mathbf{b})||\pi(\mathbf{W}, \mathbf{b})\right) \\ &\propto -\sum_{i=1}^{n} L[y_i, f(x_i; \hat{\mathbf{W}}, \hat{\mathbf{b}})] - \mathrm{KL}\left(q(\mathbf{W}, \mathbf{b})||\pi(\mathbf{W}, \mathbf{b})\right), \end{aligned} \tag{3.14}$$

ed è ricavata utilizzando l'equazione (3.4). Nel secondo passo, viene massimizzata la funzione (3.14).

Algoritmo 2 L'inferenza variazionale nella rete neurale

Ripetere fino a convergenza:

1. estrarre un campione da

$$r \text{ realizzazione della v.c. } R \sim Bernoulli(p),$$

$$\mathbf{w}_i^{(l)}|r = 0 \sim \mathcal{N}_{p_{l+1}}\left(\mathbf{0}, \sigma^2 \mathbf{I}_{p_l}\right),$$
$$\mathbf{w}_i^{(l)}|r = 1 \sim \mathcal{N}_{p_{l+1}}\left(\mathbf{m}_i^{(l)}, \sigma^2 \mathbf{I}_{p_l}\right);$$

2. massimizzare

$$\mathcal{K}(\mathbf{M}) \propto -\sum_{i=1}^{n} L[y_i, f(x_i; \hat{\mathbf{W}}, \hat{\mathbf{b}})] - \mathrm{KL}\left(q(\mathbf{W}, \mathbf{b})||\pi(\mathbf{W}, \mathbf{b})\right).$$

Si può facilmente notare che, se $\sigma^2 \to 0$, la procedura dell'algoritmo 2 coincide esattamente con il *dropout*. In questo caso, le equazioni (3.10) (3.11) (3.12) si possono riscrivere come

$$\mathbf{w}_i^{(l)} \approx R\mathbf{m}_i^{(l)} \qquad \mathbf{b}^{(l)} \approx \mathbf{m}^{(l)}.$$

Il campionamento Monte Carlo al passo 1 (algoritmo 2) viene effettuato semplicemente campionando r da $R \sim Bernoulli(p)$. Se $r = 0$ allora $\hat{\mathbf{w}}_i^{(l)} = \mathbf{0}$, altrimenti se $r = 1$ allora $\hat{\mathbf{w}}_i^{(l)} = \mathbf{m}_i^{(l)}$. Porre a 0 le colonne delle matrici di parametri corrisponde a vincolare a 0 i valori dei nodi dello strato precedente: questa è l'operazione che svolge il *dropout*. La massimizzazione di $\mathcal{K}(\mathbf{M})$ corrisponde esattamente alla massimizzazione dell'espressione (3.2), ossia all'ottimizzazione dei parametri della rete neurale, svolta ad ogni iterazione dell'algoritmo di stima. Il primo termine additivo dell'equazione (3.14) corrisponde alla funzione di perdita della rete neurale, mentre il secondo termine additivo può essere visto come una penalità *weight decay*. In realtà, non è possibile calcolare analiticamente la divergenza di Kullback-Leibler tra una mistura di normali e una distribuzione normale. Tuttavia, attraverso l'integrazione Monte Carlo, Gal e Ghahramani (2015a) provarono che

$$\mathrm{KL}\left(q(\mathbf{W}, \mathbf{b}) || \pi(\mathbf{W}, \mathbf{b})\right) \approx \frac{p}{2} \sum_{l=1}^{L} \sum_{i=1}^{p_l} \mathbf{m}_i^{(l)^T} \mathbf{m}_i^{(l)} + \frac{1}{2} \sum_{l=1}^{L} \mathbf{m}_i^T \mathbf{m}_i.$$

per $\sigma \to 0$. Si rimanda all'articolo originale la dimostrazione di questa espressione. Quindi, al passo 2 (algoritmo 2) si massimizza, rispetto a $\mathbf{m}^{(l)}$ e $\mathbf{m}_i^{(l)}$, l'espressione

$$\mathcal{K}(\mathbf{M}) = -\sum_{i=1}^{n} L[y_i, f(x_i; \hat{\mathbf{W}}, \hat{\mathbf{b}})] - \frac{p}{2} \sum_{l=1}^{L} \sum_{i=1}^{p_l} \mathbf{m}_i^{(l)^T} \mathbf{m}_i^{(l)} - \frac{1}{2} \sum_{l=1}^{L} \mathbf{m}_i^T \mathbf{m}_i, \quad (3.15)$$

con le colonne delle matrici di $\hat{\mathbf{W}}$ pari a $\hat{\mathbf{w}}_i^{(l)} = \hat{\mathbf{r}} \circ \mathbf{m}_i^{(l)}$. Il secondo e terzo termine dell'equazione (3.15) si approssimano ad una penalità *weight decay* con parametro di regolazione p, il quale stabilisce il compromesso tra varianza e distorsione del modello.

Dunque, è stato dimostrato che ogni iterazione dell'algoritmo 2 coincide con un aggiornamento dei parametri nella procedura di stima della rete neurale con *dropout*.

Ragionando, infine, sulla formulazione (3.9), Gal e Ghahramani (2015c) pensarono ad un modo alternativo di trattare il *dropout* nella fase di previsione. In particolare, pensarono di approssimare la stima dell'integrale con il metodo Monte Carlo, con T campioni, ottenendo la formulazione

$$p(y'|x';x,y) \approx \int p(y'|\mathbf{W},\mathbf{b};x')q(\mathbf{W},\mathbf{b})d\mathbf{W}d\mathbf{b} \approx \frac{1}{T}\sum_{i=1}^{T} p(y'|\hat{\mathbf{W}},\hat{\mathbf{b}};x').$$

Dal momento che, come è stato dimostrato in precedenza, ogni iterazione dell'algoritmo di stima coincide con un campionamento di $\mathbf{W}$ e $\mathbf{b}$, si possono ottenere le previsioni finali come media delle previsioni di ogni iterazione della *backpropagation*. Questo metodo è chiamato *Monte Carlo dropout* (MC-*dropout*) e generalmente porta a previsioni più accurate rispetto al *dropout* classico. Si ricorda che con quest'ultimo la previsione è ottenuta semplicemente scalando i parametri per la probabilità p, come mostrato nell'equazione (3.6).

Riassumendo

In questo capitolo è stata studiata un'interpretazione statistica dei metodi di regolarizzazione, ponendo l'attenzione principalmente su *weight decay* e *dropout*.

In particolare, al paragrafo 3.1.2 è stato dimostrato che la stima dei parametri di una rete neurale, quando è applicata la penalità *weight decay*, corrisponde alla moda a posteriori (MAP) di un opportuno modello bayesiano. La distribuzione a priori attribuita ad ogni parametro è una normale di media nulla e varianza pari al reciproco del parametro di *weight decay*.

Al paragrafo 3.3.2, invece, è stato dimostrato come le stime, ottenute con il *dropout*, siano un'approssimazione della moda a posteriori (MAP) di un modello bayesiano, il quale utilizza, come distribuzione a priori dei parametri, una normale standard. La procedura per ottenere la distribuzione a posteriori è il *variational Bayes*, con una mistura di normali come distribuzione variazionale.

Capitolo 4

Applicazioni pratiche

Nei capitoli precedenti, sono state studiate, dal punto di vista teorico, le reti neurali multi-strato. In particolare, sono stati analizzati la struttura, le proprietà statistiche ed i metodi di regolarizzazione. In questo capitolo, verranno presentate alcune applicazioni pratiche di questi modelli. La letteratura scientifica si è arricchita molto, negli ultimi anni, di articoli riguardanti le applicazioni pratiche del *deep learning*, il quale ha migliorato incredibilmente lo stato dell'arte, in alcune situazioni e con alcune tipologie di dati. Lo scopo di questo capitolo è quello di indicare, con alcuni casi di studio, quali sono i contesti in cui il *deep learning* fornisce risultati ottimali, rispetto ad altri modelli statistici supervisionati.

Analisi di *big data*

Goodfellow et al. (2016) affermano che il *deep learning* fornisce buoni risultati solamente se si ha a disposizione una grande quantità di osservazioni. In un'analisi, il cui scopo è quello di classificare le immagini in K categorie, i modelli per il *deep learning* forniscono un'accuratezza previsiva accettabile solo se si hanno a disposizione almeno 5000 osservazioni per ogni categoria. L'incremento del numero di osservazioni comporta un rilevante miglioramento, in termini di capacità previsiva: infatti, una rete neurale multi-strato riesce a pareggiare l'abilità umana nella classificazione d'immagini, se l'insieme di dati contiene qualche milione di osservazioni per ogni categoria.

Alla luce di quanto affermato da Goodfellow et al. (2016), in questo paragrafo verrà presentato un caso di studio di *big data*, ossia un insieme di dati con un elevato numero di osservazioni (n) e di variabili esplicative (p).

Caso di studio: *Reuters Corpus Volume I*

Reuters Corpus Volume I (RCV1) è una raccolta di circa 800 000 articoli di attualità, suddivisi manualmente per categorie. Il *dataset* è stato reso recentemente disponibile dalla *Reuters*, un'agenzia di stampa britannica, per scopi di ricerca. Ci sono 47 236 variabili esplicative, ognuna delle quali rappresenta una parola. Il *dataset* è caratterizzato da sparsità: il 99.84% dei dati è pari a zero. I dati non nulli sono rappresentati nella forma *term frequency–inverse document frequency* (TF-IDF): il loro valore aumenta proporzionalmente al numero di volte che il termine è contenuto nell'articolo, ma cresce in maniera inversamente proporzionale con la frequenza del termine in articoli della stessa categoria. L'idea alla base è di dare più importanza ai termini che compaiono nell'articolo, ma che in generale sono poco frequenti. La variabile risposta, invece, è costituita da 103 colonne, ognuna delle quali rappresenta una categoria. Gli elementi della variabile risposta possono assumere i valori "0" o "1", a seconda se quella specifica categoria è assente o presente. Ad ogni articolo può, quindi, corrispondere anche più di una categoria. È stato utilizzato il *dataset* contenuto nel pacchetto `sklearn` di `Python`. Per maggiori informazioni riguardo al *dataset* si veda Lewis et al. (2004).

Operazioni preliminari

Sono state fatte alcune operazioni preliminari prima di effettuare la fase di modellazione statistica. In primo luogo, si è scelto di considerare solamente una delle 103 categorie di articoli, e di conseguenza prendere solo una delle 103 colonne della variabile risposta. Si è scelto di considerare solamente la categoria più frequente, ossia "CCAT" (articoli che riguardano l'ambito aziendale/industriale); la variabile risposta quindi contiene "0" se questa categoria è assente e "1" se è presente. È inoltre bilanciata, in quanto la categoria è presente nel 47.4% di articoli. Infine, ogni variabile esplicativa è stata dicotomizzata in "presenza" o "assenza" della parola che rappresenta (ogni elemento

assume ora il valore "0" o "1"). Questa operazione è stata effettuata affinché i dati occupino un minor numero di *bit*, rendendo possibile così il caricamento dell'intero *dataset* in memoria RAM.

Per ovviare al problema della maledizione della dimensionalità, è stata effettuata la selezione delle variabili che più influenzano la risposta. L'elevata dimensionalità dell'insieme di dati, tuttavia, rende impossibile la selezione di variabili tramite modellazione statistica (ad esempio attraverso una regressione *stepwise* o *lasso*). Per tale motivo, si è scelto di effettuare la selezione valutando la relazione marginale di ogni variabile esplicativa con la variabile risposta. Per effettuare questa valutazione sono state utilizzate 40 221 osservazioni, corrispondenti al 5% dei dati. Dal momento che variabili esplicative e risposta sono variabili qualitative a due categorie, è stata valutata l'associazione tra la risposta e ogni singola esplicativa, applicando il test delle probabilità esatte di Fisher (Fisher 1935). La scelta di questo test è giustificata dal fatto che molte variabili esplicative contengono pochissimi valori pari a 1. Si è scelto di selezionare le 2000 variabili che presentavano il p-value più piccolo (in questo caso $p < 0.00298$), sintomo che le variabili considerate sono fortemente associate con la risposta. In questo caso i p-value sono utilizzati solamente per ricavare una classifica delle 2000 variabili più associate con la risposta.

Infine, tra le osservazioni non utilizzate per l'operazione precedente sono stati estratti casualmente l'*insieme di stima*, l'*insieme di convalida* ed l'*insieme di verifica*: il primo dedicato alla stima dei modelli, il secondo usato per scegliere i parametri di regolarizzazione ottimali ed il terzo per confrontare gli errori dei diversi modelli statistici. Questi tre insiemi di dati contengono rispettivamente 201 103, 80 441 e 80 441 osservazioni, corrispondenti al 25%, 10% e 10% dei dati. In tutto, sono stati utilizzati solamente il 50% dei dati totali, in quanto caricare in memoria insiemi di dati più grandi avrebbe reso computazionalmente impossibile la stima dei modelli. Nella fase di regolazione dei parametri, a volte si è ritenuto opportuno ridurre il numero di osservazioni dell'insieme di stima, in modo da diminuire i tempi computazionali.

Modellazione statistica

Sono stati stimati i seguenti modelli statistici:

- *Naive Bayes* (Zhang 2004), in cui ogni variabile esplicative, all'interno di una categoria, assume verosimiglianza normale.
- *Bagging* con 50 alberi di classificazione (Breiman 1996).
- *Random forest* con 400 alberi di classificazione; il numero di variabili campionate per ogni albero è stato impostato a 40 (scelto tra 5 valori contenuti tra 35 e 55), (Breiman 2001).
- *Adaboost* con 500 alberi di classificazione (scelto tra 5 valori contenuti tra 100 e 500), e con alberi di profondità 2 (valore scelto tra 2, 3 e 5), (Freund e Schapire 1995).
- *MARS* con funzione legame logistica, con un numero di basi fatto crescere fino a 100 e potato a 63 (Friedman 1991).
- *Regressione logistica* con funzione legame *logit*, ottenuta stimando i parametri in modo ricorsivo (algoritmo di Miller 1992).
- *Extreme gradient boosting* con 300 alberi di classificazione, con profondità massima pari a 10 (scelto tra 3, 5 e 10); il valore di *learning rate* è impostato pari a 0.3 (scelto tra 0.1, 0.3 e 0.5) e la porzione di colonne campionata per ogni albero è pari a 0.5 (scelti tra i valori 0.1, 0.2, 0.3, 0.5 e 1), (Friedman 2001, Chen e Guestrin 2016).
- *Rete neurale multi-strato* con 5 strati latenti e 1024 nodi per ogni strato; è stato applicato un'*early stopping* alla 50^a epoca.
- *Rete neurale multi-strato* con 2 strati latenti con 1024 e 512 nodi rispettivamente; È stato applicata una penalità *weight decay* pari a 0.001, ed un'*early stopping* alla 20^a epoca.
- *Rete neurale multi-strato* con 2 strati latenti e 2048 nodi per ogni strato; è stato applicato il *dropout*, con probabilità di conservare i nodi pari a 0.25 per gli strati latenti e 0.8 per l'input; inoltre, è stata applicata una penalità *weight decay* pari a 0.0001.

- *Rete neurale multi-strato* con 3 strati latenti e 2048 nodi per ogni strato; È stato applicato il *Monte Carlo dropout*, con probabilità di conservare i nodi pari a 0.25 per gli strati latenti e 0.8 per l'input; inoltre, è stata applicata una penalità *weight decay* pari a 0.1.

Si è deciso quindi di inserire nel confronto quattro reti neurali multi-strato, ognuna con diversi metodi di regolarizzazione. In particolare, sono state considerate

- una rete che non utilizza né *weight decay* né *dropout*;
- una rete che utilizza il *weight decay*;
- una rete che utilizza il *weight decay* e il *dropout*;
- una rete che utilizza il *weight decay* e il MC-*dropout*.

Ciò è stato fatto per avere modo di confrontare anche l'efficacia previsiva dei vari metodi di regolarizzazione. La struttura delle reti neurali multi-strato è stata scelta provando diverse architetture e scegliendo quella che forniva errore minimo nell'insieme di convalida. Sono state provate diverse reti variando da 1 a 5 strati latenti e da 128 a 2048 nodi. La probabilità di conservare un nodo nel *dropout* è stata scelta tra 0.25 e 0.5, per gli strati latenti, e tra 0.8 e 1, per l'input. Il parametro di *weight decay* è stato scelto tra i valori contenuti fra 0 e 0.1. Inoltre, è stata fissata una porzione di *mini-batch* pari a 1024. Infine, per le reti stimate con *dropout* sono state utilizzate 100 epoche. Le reti neurali usate sono le *feed-forward neural network* (FFNN).

Per valutare la capacità previsiva dei modelli statistici descritti sopra, si è utilizzato il tasso di errata classificazione, ossia la porzione di osservazioni classificate erroneamente. I risultati ottenuti sono rappresentati in Tabella 4.1.

Si noti come tre su quattro, tra le reti neurali stimate, forniscano un tasso di errata classificazione inferiore a quello di tutti gli altri modelli. La rete neurale multi-strato, quindi, è un modello particolarmente efficace, se utilizzato con un elevato numero di osservazioni. Si noti, inoltre, che il metodo di regolazione del *dropout* incrementa considerevolmente la capacità previsiva, rispetto al *weight decay*, che invece comporta pochi miglioramenti. Infine,

Tabella 4.1: Risultati sulle previsioni del *dataset* RCV1

Modello	**Errore di classificazione**
Naive Bayes	19.80 %
MARS logistico	12.44 %
GLM ricorsivo	8.15 %
Bagging	7.84 %
Adaboost	7.80 %
Random forest	6.28 %
FFNN	5.35 %
XGBoost	5.31 %
FFNN + L_2	5.30 %
FFNN + dropout	4.66 %
FFNN + MC-dropout	4.48 %

l'operazione di combinare le previsioni di ogni epoca (MC-*dropout*) fornisce un ulteriore miglioramento rispetto al *dropout* classico. I modelli presentati sono stati scelti in modo che possano essere stimati in un tempo ragionevole. Precisamente si è tenuto, come limite massimo, le tre ore di tempo.

Classificazione di immagini

Una caratteristica del *deep learning* è che funziona particolarmente bene per determinate tipologie di dati. Per alcune di queste, però, vengono utilizzate classi differenti di reti neurali, rispetto alle *feed-forward neural networks*, di cui si è parlato finora.

Le classi di reti neurali, che al momento sono le più utilizzate e che stanno riscontrando il maggior successo, sono le *convolutional neural networks* (LeCun et al. 1998a) e le *recurrent neural networks* (Elman 1990). Le prime sono un'innovazione nella classificazione di immagini, video e del riconoscimento vocale, le seconde eccellono particolarmente nell'elaborazione di testi e discorsi (Goodfellow et al. 2016).

In questo paragrafo, sarà affrontato un caso di classificazione di immagini, in cui verranno utilizzate le *convolutional neural networks*.

Convolutional neural networks

In seguito, verrà fornita una breve descrizione sulla struttura delle *convolutional neural network* (CNN). Poiché lo scopo di questo capitolo è quello di capire quali sono le situazioni in cui il *deep learning* funziona particolarmente bene, la descrizione di questa rete neurale verrà affrontata solo marginalmente.

Le CNN sono reti neurali costruite per elaborare dati nella forma di *array* multipli. Un immagine, ad esempio, possiede la struttura di un'*array* a tre dimensioni. Le prime due dimensioni rispecchiano la disposizione dei *pixel*, mentre la terza dimensione è la rappresentazione del colore. Se le immagini sono RGB, questa dimensione è composta da tre valori, uno per ogni colore (rosso, verde, blu), mentre è composta da solamente un valore se l'immagine è in bianco e nero.

Le CNN hanno in genere molti strati: la prima serie di strati dopo l'input alterna, spesso, uno strato *convoluzionale* con uno strato di *pooling*; la seconda serie di strati, invece, sono *fully-connected*, cioè esattamente come quelli delle *feed-forward neural networks*, descritte nel capitolo 1. (si veda Figura 4.1).

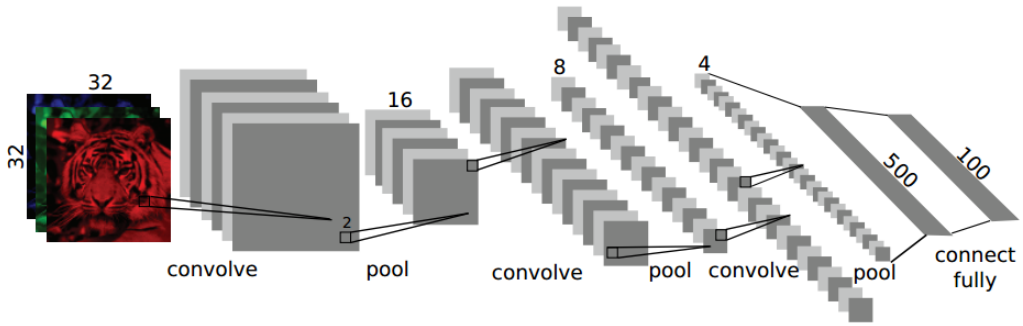

Figura 4.1: La struttura della *convolutional neural network*: alterna strati *convoluzionali* a strati di *pooling*; l'ultimo strato è *fully-connected*.

Lo strato *convoluzionale*, successivo allo strato di input, è costituito da p "versioni" differenti dell'immagine in entrata. Ognuna di queste "versioni" è

il risultato dell'applicazione di un *kernel* di trasformazione, chiamato *filtro*. Il filtro, spesso di dimensioni molto piccole, viene moltiplicato (attraverso il *prodotto-interno*) ad ogni sotto-immagine (delle stesse dimensioni del filtro) dell'immagine dello strato precedente. Quindi, se l'immagine x è di dimensioni $k \times k \times 3$, a cui viene applicato un filtro f di dimensioni $q \times q$, allora il generico elemento della "versione" trasformata dell'immagine si ricava come $\tilde{x}_{i,j} = \sum_{h=1}^{3} \sum_{l=1}^{q} \sum_{l'=1}^{q} x_{i+l,j+l',h} f_{l,l'}$. I valori del filtro sono i parametri della rete. Lo scopo dello strato convoluzionale è quello di rilevare *pattern* locali all'interno dell'immagine, che possono essere caratteristici dell'oggetto rappresentato.

Lo strato di *pooling*, invece, suddivide ogni immagine in piccole parti di dimensione $r \times r$, e di ognuna di queste prende il valore massimo. Con ciascun valore massimo ricostruisce un'immagine di dimensioni ridotte. Lo scopo dello strato di *pooling* è quello di unire caratteristiche simili (in quanto vicine) in una sola caratteristica.

Per maggiori informazioni sulle *convolutional neural networks*, si veda Goodfellow et al. (2016, cap. 9).

Caso di studio : *Mnist*

I dati provengono dal *dataset* del *MNIST* (*Modified National Institute of Standards and Technology*, LeCun et al. 1998b), di cui sono state prese in considerazione solamente 42 000 osservazioni, ovvero la parte scaricabile da Kaggle.com (*Kaggle: Your Home for Data Science*), il principale sito di competizioni nell'ambito dell'analisi di dati. Il *dataset* è composto da immagini in bianco e nero, ognuna delle quali rappresenta una cifra da "0" a "9" scritta a mano. Ogni immagine ha grandezza 28×28 pixel, ognuno dei quali assume un valore da 0 a 255, a seconda della tonalità di grigio.

Per evitare il sovra-adattamento, è stato diviso l'insieme dei dati in tre sottoinsiemi: l'insieme di stima, l'insieme di verifica e l'insieme di convalida, rispettivamente di 21 000, 10 500 e 10 500 unità. L'insieme di stima, come dice il nome, è stato usato per la stima dei modelli, l'insieme di verifica per confrontare l'adattamento di modelli diversi, mentre l'insieme di convalida per scegliere il valore ottimale dei parametri di regolarizzazione.

Modellazione statistica

Considerando ognuno dei *pixel* come una variabile esplicativa dell'insieme di dati, sono stati stimati i seguenti modelli:

- *K-Nearest Neighbors* con $K = 3$ osservazioni vicine (scelto tra 5 valori compresi tra 1 e 10).
- Albero di classificazione, potato a profondità 22; in fase di crescita è stata massimizzata la riduzione dell'indice di Gini, mentre in fase di potatura è stato minimizzato il tasso di errata classificazione.
- *Random forest* con 500 alberi; il numero di variabili campionate per ogni albero è impostato a 30 (scelto tra 5 valori contenuti tra 10 e 50), (Breiman 2001).
- *Extreme gradient boosting* con 300 alberi di classificazione, con profondità massima pari a 5 (scelto tra 3, 5 e 10); il valore di *learning rate* è impostato pari a 0.3 (scelto tra 0.1, 0.3 e 0.5) e la porzione di colonne campionata per ogni albero è pari a 0.3 (scelti tra i valori 0.1, 0.2, 0.3, 0.5 e 1), (Friedman 2001, Chen e Guestrin 2016).

Sono state estratte, in seguito, le componenti principali dalle variabili esplicative, allo scopo di ridurre il numero di variabili. Il numero di componenti da conservare nel modello è stato trattato come parametro di regolarizzazione, ed è stato scelto tra 26, 43 e 87, che corrisponde al 70%, 80% e 90% della varianza spiegata. Quindi, sulle componenti principali, sono stati stimati i seguenti modelli:

- *Naive Bayes* con verosimiglianza normale e 87 componenti (Zhang 2004).
- *Support vector machine* con *kernel polinomiale* e con 43 componenti. Il grado del polinomio è 3 (scelto tra 2, 3 e 4), mentre il costo di violazione della barriera è 1 (scelto tra 0.1, 1 e 10).
- *Support vector machine* con *kernel a basi radiali* e con 26 componenti. Il parametro di precisione è 10^{-1} (scelto tra 10^{-3}, 10^{-2} e 10^{-1}), mentre il costo di violazione della barriera è 1 (scelto tra 0.1, 1 e 10).

- *Feed-forward neural network* con 2 strati latenti da 1024 e 512 nodi rispettivamente; è stato applicato un'*early stopping* alla 20^a epoca ed una penalità *weight decay* pari a 0.001.

- *Feed-forward neural network* con 2 strati latenti e 1024 nodi per ogni strato; è stato applicato il *dropout* solo agli strati latenti, con probabilità di conservare i nodi pari a 0.5; inoltre, è stata applicata una penalità *weight decay* pari a 0.001.

- *Feed-forward neural network* con 3 strati latenti e 1024 nodi per ogni strato; è stato applicato il *Monte Carlo dropout* solo agli strati latenti, con probabilità di conservare i nodi pari a 0.5; inoltre, è stata applicata una penalità *weight decay* pari a 0.001.

Infine, considerando i dati come *array* bi-dimensionali sono state stimate le *convolutional neural networks*, con *weight decay* pari a 0.001 e

- *early stopping* alla 20^a epoca;

- *dropout*, con probabilità di conservare i nodi pari a 0.5, solo negli strati *fully-connected*;

- *MC-dropout*, con probabilità di conservare i nodi pari a 0.5, solo negli strati *fully-connected*.

Sono state confrontate reti neurali multi-strato con diversi metodi di regolarizzazione, in modo da avere un confronto anche su questo fronte. È stato dimostrato empiricamente che il *dropout* fornisce buoni risultati solo se applicato allo strato *fully-connected* (Hinton et al. 2012).

La CNN implementata è chiamata *Lenet*, la cui prima applicazione è stata attuata da LeCun et al. (1998a). Si tratta di una semplice CNN composta da due strati *convoluzionali* e due strati di *pooling*, disposti in sequenza alternata. A seguire, è stato posto uno strato latente composto da 500 nodi. Inoltre, è stata fissata una porzione di *mini-batch* pari a 256 e, per le reti stimate con *dropout*, sono state utilizzate 100 epoche.

Per valutare la capacità previsiva dei modelli statistici descritti sopra, si è utilizzato il tasso di errata classificazione. I risultati ottenuti sono rappresentati in Tabella 4.2.

Tabella 4.2: Risultati sulle previsioni del *dataset* Mnist

Modello	**Errore di classificazione**
Albero di classificazione	12.44 %
Naive Bayes	12.80 %
Random forest	3.94 %
KNN	3.75 %
XGBoost	2.87 %
FFNN	2.87 %
SVM polinomiale	2.73 %
SVM basi radiali	2.36 %
FFNN + dropout	2.08 %
FFNN + MC-dropout	1.73 %
CNN	1.36 %
CNN + dropout	1.23 %
CNN + MC-dropout	1.02 %

Si noti come le tre CNN si posizionino nelle prime tre posizioni. Inoltre, i metodi di regolazione del *dropout* e del MC-*dropout* incrementano considerevolmente la capacità previsiva. Le *convolutional neural network*, quindi, sono una classe di modelli particolarmente efficaci, se utilizzate con lo scopo di classificare immagini.

Aspetti computazionali

Nell'era dei *big data*, come viene definita da Goodfellow et al. (2016), raccogliere una grande quantità di dati non è più un problema. La loro elaborazione, al contrario, può causare delle complicazioni se la stima del modello utilizzato è computazionalmente pesante. Dal momento che possiedono un numero esorbitante di parametri, le reti neurali multi-strato necessitano di moltissime operazioni per ottenere la stima del modello. Questo, purtroppo, accade anche se si usa la procedura di *backpropagation* che, come spiegato nel capitolo 2, ottimizza la stima numerica dei parametri riducendo di molto i costi computazionali.

Grazie alla particolare struttura della rete neurale e al suo metodo di stima dei parametri, questo modello si presta alla *parallelizzazione*. Ciò significa che è possibile separare le operazioni computazionali e affidarle a più processori. Così facendo, i processori elaborano i calcoli simultaneamente, riducendo di molto i costi computazionali. Se un computer è dotato di c processori ed ognuno di questi elabora le operazioni di stima della rete neurale contemporaneamente, allora il tempo necessario per la stima della rete è nell'ordine di $1/c$, rispetto all'utilizzo di un singolo processore. La condizione necessaria per svolgere il calcolo parallelo, è che le operazioni debbano essere indipendenti l'una dall'altra. In seguito, si vedrà come la parallelizzazione può essere applicata alla rete neurale multi-strato.

Il calcolo parallelo nella *backpropagation*

In questo paragrafo verrà spiegato il modo in cui è possibile utilizzare il calcolo parallelo nelle *feed-forward neural networks*. Se si ha a che fare con altre classi di modelli come le *convolutional neural networks* o le *recurrent neural networks*, la procedura di parallelizzazione segue la stessa logica con alcune semplici modifiche, per maggiori informazioni si veda Hegde e Usmani (2016). Come spiegato in breve da Goodfellow et al. (2016, sez. 12.1), il calcolo parallelo nelle *feed-forward neural networks* può essere applicato in due modi:

- parallelizzando i dati;
- parallelizzando il modello.

La *parallelizzazione dei dati* è la più semplice ed intuitiva, ed è anche quella che è stata adottata per i modelli stimati in questa tesi. Come si può vedere nell'algoritmo 1, la fase di propagazione della procedura di *backpropagation* viene attuata per ogni osservazione indipendentemente. Quindi, può essere affidata una porzione di dati ad ogni processore, il quale ne calcola i gradienti, risparmiando così tempo computazionale. I gradienti vengono poi sommati tra di loro (equazione (1.9)) per essere impiegati nella fase di aggiornamento dei parametri. Questa seconda fase purtroppo non è parallelizzabile, in quanto ogni aggiornamento necessita del valore delle stime dell'iterazione precedente.

La *parallelizzazione del modello*, invece, è un'idea avanzata che consiste nell'utilizzare i processori per stimare parti diverse del modello, ovviamente dove l'indipendenza dei calcoli lo consente. In particolare, viene utilizzata nelle operazioni tra matrici.

L'utilizzo dell'unità di elaborazione grafica (GPU)

Alcune librerie di software statistici ed informatici permettono l'utilizzo dell'*unita di elaborazione grafica* (GPU). Rispetto al processore centrale (CPU), la GPU possiede un numero molto maggiore di processori: se si è in presenza di grandi insiemi di dati, utilizzare la GPU consente di stimare la rete neurale in tempi molto più brevi.

In questa tesi, per la stima delle reti neurali multi-strato si è fatto uso della libreria `MXNet`, la quale è stata implementata sia per `R` che per `Python`. Questa libreria permette di effettuare la stima del modello sfruttando il calcolo parallelo in GPU. Per l'elaborazione di questa tesi è stata utilizzata solamente una GPU, ma la libreria permette di usare più dispositivi in contemporanea. Per ulteriori informazioni a riguardo consultare l'indirizzo elettronico Mxnet.io (*Run MXNet on Multiple CPU/GPUs with Data Parallel*). L'utilizzo della GPU per l'esecuzione della libreria necessita di alcune operazioni preliminari, le quali comprendono

- la configurazione del CUDA, un'architettura hardware per l'elaborazione parallela, creata da NVIDIA;
- l'installazione della libreria CuDNN (CUDA Deep Neural Network), per incrementare ulteriormente la performance della GPU.

Queste operazioni sono state effettuate in un computer con sistema operativo LINUX UBUNTU 16.04 LTS e dotato di una GPU NVIDIA GEFORCE 820M. Questa GPU possiede 96 processori, ognuno dei quali ha 775MHz di velocità.

Librerie utilizzate

Per stimare le reti neurali multi-strato, come già annunciato al paragrafo precedente, è stata utilizzata la libreria `MXNet` (per maggiori informazioni si

veda Chen et al. (2015)). Per il modello GLM con stime ricorsive dei parametri è stata utilizzata la libreria `bigglm`, implementata in `R`. La stima e la previsione di tutti gli altri modelli statistici, invece, sono state implementate in `Python`, utilizzando le librerie `sklearn` e `xgboost`. La scelta di utilizzare `Python`, piuttosto che `R`, per questi modelli, è data dal fatto che il primo garantisce una miglior gestione della memoria e tempi computazionali inferiori. Infatti, l'utilizzo di `R` non ha portato solo a problemi di lentezza di esecuzione, ma spesso i processi di stima saturavano velocemente la memoria RAM.

Le reti neurali multi-strato sono state implementate sia in `R` che in `Python`. Il modello stimato in CPU, sfrutta comunque il calcolo parallelo: il computer utilizzato possiede 4 processori. La rete neurale è stata stimata sull'insieme di dati RCV1 (primo caso di studio) e possiede tre strati latenti, ognuno con 256 nodi. In Tabella 4.3, sono riportati i tempi computazionali della stima del modello, sia utilizzando la CPU che la GPU. Con un numero superiore di nodi, con `R` si ha un errore di memoria, mentre `Python` arriva a stimare anche 2048 nodi per strato.

Tabella 4.3: Tempi di esecuzione di una rete neurale con 3 strati da 256 nodi. Il confronto viene effettuato tra CPU e GPU e tra `R` e `Python`.

	R	Python
CPU	33 minuti	23 minuti
GPU	19 minuti	14 minuti

L'utilizzo della GPU porta un risparmio netto in termini di costo computazionale. In questo caso, il tempo impiegato viene quasi dimezzato ma si tenga a mente che i risultati possono essere ancora più evidenti: le GPU di nuova generazione possono avere anche migliaia di processori.

Si noti, inoltre, come `Python` offra performance migliori di `R`. I motivi possono essere due:

- Nonostante `MXNet` sia implementato in `C++` come linguaggio base, l'ambiente `R`, come linguaggio di collegamento, rallenta le operazioni computazionali;
- La cattiva gestione della memoria, da parte di `R`, comporta un rallentamento generale del computer.

È più probabile che il vero motivo sia il secondo, in quanto si è notato che, caricando in memoria meno dati, le performance di `Python` e `R` tendono ad essere più simili.

Conclusioni

Questa tesi offre un'analisi, dal punto di vista statistico, di un argomento che, al giorno d'oggi, sta rivoluzionando il mondo della tecnologia e di molte altre discipline, ad essa collegate: il *deep learning*.

Nell'ambito del *deep learning* supervisionato, il modello statistico più utilizzato è la rete neurale multi-strato, la quale può presentare strutture differenti a seconda del tipo di dati che si vuole analizzare (cap. 4). Questa tesi offre una sistematizzazione, dal punto di vista statistico, di quanto è presente in letteratura riguardo al *deep learning* ed in particolare alle *feed-forward neural networks*.

In seguito verranno riportati i contributi "teorici" originali, che per questa tesi rappresentano i risultati più rilevanti.

Il capitolo 2 presenta un confronto statistico tra la rete neurale multi-strato e il modello MARS. In particolare, è stato mostrato come il modello MARS sia un caso particolare di rete neurale multi-strato con funzioni di attivazione ReLU e con alcuni vincoli sui parametri. La rete neurale, costruita in questo modo, riesce a modellare solamente relazioni marginali se è composta da un solo strato latente, mentre è in grado di considerare interazioni tra variabili se è composta da più strati latenti. Questa è una dimostrazione, basata su un confronto statistico, del vantaggio che comporta l'utilizzo di strati latenti multipli.

Nel capitolo 2, inoltre, è riportata la dimostrazione di come la funzione di attivazione ReLU, molto utilizzata nel *deep learning*, renda la rete neurale multi-strato un *approssimatore universale*. In particolare, è stato dimostrato il modo in cui è possibile combinare due funzioni ReLU per garantire la non linearità del modello e soddisfare la condizione necessaria per far diventare

la rete neurale un approssimatore universale.

Nel capitolo 3 sono trattati i principali metodi di regolarizzazione per il *deep learning*: la penalità *weight decay* ed il *dropout*. In particolare è stato dimostrato come le stime, ottenute con il *dropout*, siano un'approssimazione della moda a posteriori (MAP) di un modello bayesiano, il quale utilizza, come distribuzione a priori dei parametri, una normale standard. La procedura per ottenere la distribuzione a posteriori è il *variational Bayes*, con una mistura di normali come distribuzione variazionale. Come supporto per questa dimostrazione è stato utilizzato il lavoro svolto da Gal e Ghahramani (2015d).

Il capitolo 4 è dedicato all'applicazione pratica del *deep learning*. Nella prima parte si vuole presentare, attraverso alcuni casi di studio, quali sono le situazioni in cui il *deep learning* riscuote maggior successo. In particolare, sono state considerate l'analisi di *big data* e la classificazione di immagini. In entrambi i casi, il *deep learning* ha fornito risultati molto più soddisfacenti, rispetto agli altri modelli statistici considerati. Nella seconda parte viene presentato un confronto, in termini di tempo computazionale impiegato per stimare il modello, tra l'utilizzo dell'architettura GPU e quella CPU. È stato proposto anche un confronto tra l'ambiente `R` e `Python`.

Un immediato sviluppo futuro è quello di analizzare le proprietà statistiche delle *convolutional neural networks* e delle *recurrent neural networks*, ovvero di altre classi di reti neurali che attualmente rappresentano lo stato dell'arte nell'analisi di determinate tipologie di dati. Dopodiché è possibile stilare un elenco infinito di possibili sviluppi futuri, per questo ambito che è attualmente in rapita evoluzione.

Appendice A

La libreria `MXNet`

Per l'implementazione dei modelli statistici utilizzati in questa tesi, è stato fatto largo uso di `MXNet`. Per una ricerca esaustiva utilizzare il sito web Mxnet.io (*MXNet Documents*) e l'articolo prodotto da Chen et al. (2015).

`MXNet` è una libreria destinata al *machine learning*, costruita al fine di massimizzare efficienza e flessibilità. Il creatore di tale libreria è il gruppo di ricerca DMLC, che collabora per sviluppare progetti *open-source* riguardanti il *machine learning*, allo scopo di rendere tale disciplina disponibile su larga scala e di facile utilizzo. Il gruppo è formato da ricercatori, studenti di dottorato e analisti di dati.

La libreria supporta sia una programmazione di tipo *dichiarativo*, ovvero finalizzata alla rappresentazione del contenuto di una certa entità, sia una programmazione *imperativa*, che si focalizza invece sul processo di creazione di tale entità. La programmazione dichiarativa si basa su espressioni simboliche e grafiche, che incrementano le opportunità di ottimizzazione del programma, in quanto ne ricostruiscono l'intera struttura prima di procedere al calcolo vero e proprio; al contrario, la programmazione imperativa si basa sul calcolo tensoriale e ne migliora la flessibilità, eseguendo il programma passo dopo passo, in modo semplice e lineare.

Le espressioni simboliche sono composte da operatori, che possono essere sia semplici operazioni matriciali, che strutture reticolari più complesse. Gli operatori hanno la caratteristica di ricevere in entrata le variabili di input, trasformarle ed elaborarle attraverso operazioni interne, e fornire come ri-

sultato alcune variabili di output. MXNet, inoltre, sfrutta la programmazione imperativa ed il calcolo tensoriale come collegamento tra le espressioni simboliche ed il linguaggio ospite. Infatti la libreria fa riferimento a C++ come linguaggio di base, ma può essere eseguita anche in molti altri linguaggi ad esso collegati come, ad esempio, Python, R, Julia, MATLAB, Go e Scala. Per ottenere i risultati presentati in questa tesi sono stati utilizzati R e Python come ambienti di programmazione.

Infine MXNet, per incrementare ulteriormente le prestazioni computazionali, può avvantaggiarsi del calcolo distribuito. Infatti, oltre ad utilizzare più di un *core* della CPU in parallelo, si avvale anche della potenza computazionale dell'*unità di elaborazione grafica* (GPU). Per l'elaborazione di questa tesi è stata utilizzata solamente una GPU, ma la libreria permette di usare più dispositivi GPU in contemporanea. Per ulteriori informazioni a riguardo consultare l'indirizzo elettronico Mxnet.io (*Run MXNet on Multiple CPU/GPUs with Data Parallel*). L'utilizzo della GPU per l'esecuzione della libreria necessita di alcune operazioni preliminari. In seguito sono riportate i passi principali per fare uso di MXNet attraverso la GPU.

- verificare di aver installato nel proprio computer il compilatore C++ che supporta C++11;
- installare il CUDA Toolkit 7.0 o maggiore, il quale permette ad MXNet di essere eseguito in una GPU NVIDIA che lo supporta; si veda una guida dettagliata al link Docs.nvidia.com (*CUDA Toolkit Documentation*);
- installare la libreria CuDNN (CUDA Deep Neural Network), per incrementare ulteriormente la performance della GPU (operazione non obbligatoria ma fortemente consigliata);
- scaricare il pacchetto MXNet dal sito Github.com (*GitHub - dmlc*);
- se, ad esempio, si vuole utilizzare R come *software* per la programmazione, ora è possibile installare la libreria nell'ambiente R.

In questo caso, è stata utilizzata una scheda video NVIDIA GEFORCE 820M, ed un sistema operativo LINUX UBUNTU 16.04 LTS. La libreria è però supportata dalla maggior parte dei sistemi operativi, quali Windows, Ubuntu, OS X,

`Android` e `iOS`. Per una guida dettagliata sulla procedura di installazione consultare il sito web Mxnet.io. (*Overview — mxnet 0.9.3 documentation*).

Come indicato nell'articolo di Chen et al. (2015), la libreria `MXNet` è stata confrontata con altri sistemi, quali `TensorFlow`, `Caffe` e `Torch7`, che permettono di eseguire le reti neurali e altri modelli adatti al *machine learning*. Il confronto fornisce indicazioni positive sulle prestazioni di `MXNet`, i cui costi computazionali sono sempre minori o uguali rispetto a quelli degli altri sistemi. Un ulteriore fattore che collabora nel fornire delle prestazioni computazionali notevoli è che `MXNet` è ottimizzata per ridurre al minimo l'uso di allocazione in memoria.

In seguito è riportato il codice `R` e `Python`, utile per fare la stima e la previsione di una rete neurale multi-strato con MC-*dropout*.

Codice A.1: Codice R per la previsione di una DNN con MC-dropout

```
library(mxnet)

# Costruzione dell'architettura della DNN
get_mlp_drop <- function(n1, n2, n3, n_class, p_input, p_hidden) {
  data = mx.symbol.Variable('data')
  data2 = mx.symbol.Dropout(data, p = p_input)
  fc1  = mx.symbol.FullyConnected(data = data2, name='fc1', num_hidden =
      n1)
  act1 = mx.symbol.Activation(data = fc1, name='relu1', act_type = "relu")
  drop1 = mx.symbol.Dropout(act1, p = p_hidden)
  fc2  = mx.symbol.FullyConnected(data = drop1, name = 'fc2', num_hidden =
       n2)
  act2 = mx.symbol.Activation(data = fc2, name='relu2', act_type = "relu")
  drop2 = mx.symbol.Dropout(act2, p = p_hidden)
  fc3  = mx.symbol.FullyConnected(data = drop2, name = 'fc3', num_hidden =
       n3)
  act3 = mx.symbol.Activation(data = fc3, name='relu2', act_type  ="relu")
  drop3 = mx.symbol.Dropout(act3, p = p_hidden)
  fc4  = mx.symbol.FullyConnected(data = drop3, name = 'fc4', num_hidden =
       n_class)
  mlp  = mx.symbol.SoftmaxOutput(data = fc4, name = 'softmax')
```

```
  return(mlp)
}

net <- get_mlp_drop(n1 = 256, n2 = 256, n3 = 256, n_class = 2,
                    # (256/256/256 con risposta a 2 categorie)
                    p_input = 0.2, p_hidden = 0.75)
                    # prob. di dropout 0.5 per strati latenti, 0.2 per la
                         risposta

# Stima del modello
model <- mx.model.FeedForward.create(
  symbol = net,
  X = X.train,
  y = y.train,
  ctx = mx.gpu(0), # mx.cpu() -> per utilizzare la CPU
  num.round = 100,
  optimizer = "adam",
  initializer = mx.init.normal(0.01),
  eval.metric = mx.metric.accuracy,
  verbose = F,
  array.batch.size = 1024,
  array.layout = "rowmajor",
  wd = 0.1,
  epoch.end.callback = mx.callback.save.checkpoint('checkpoint/chkpt')
)

# Calcolo della previsione con Monte Carlo dropout
MCdropout <- function(y.test, prefisso, n_epoche_tot, n_epoche_mc){
  p <- matrix(NA, length(y.test), n_epoche_mc)
  e <- rep(NA, n_epoche_mc)
  start <- n_epoche_tot - n_epoche_mc + 1
  for(i in start:n_epoche_tot){
    m <- mx.model.load(prefix = prefisso, iteration = i)
    p[, n_epoche_tot - i + 1] <- predict(m, X.test,  array.layout = "
        rowmajor")[2, ]
  }
```

```
  voto_max <- rowMeans(p) > 0.5
  e <- mean(y.test != voto_max)
  return(e)
}
MCdropout(y.test = y.test, prefisso = 'checkpoint/chkpt', n_epoche_tot =
    100, n_epoche_mc = 100)
```

Codice A.2: Codice Python per la previsione di una DNN con MC-dropout

```
import numpy as np
import mxnet as mx
from scipy import *

# Costruzione dell'architettura della DNN
def get_mlp_drop(n1, n2, n3, p_input, p_hidden):
  data = mx.symbol.Variable('data')
  data2 = mx.symbol.Dropout(data, p = p_input)
  fc1  = mx.symbol.FullyConnected(data = data2, name='fc1', num_hidden=n1)
  act1 = mx.symbol.Activation(data = fc1, name='relu1', act_type="relu")
  drop1 = mx.symbol.Dropout(act1, p = p_hidden)
  fc2  = mx.symbol.FullyConnected(data = drop1, name = 'fc2', num_hidden =
       n2)
  act2 = mx.symbol.Activation(data = fc2, name='relu2', act_type="relu")
  drop2 = mx.symbol.Dropout(act2, p = p_hidden)
  fc3  = mx.symbol.FullyConnected(data = drop2, name = 'fc3', num_hidden =
       n3)
  act3 = mx.symbol.Activation(data = fc3, name='relu2', act_type="relu")
  drop3 = mx.symbol.Dropout(act3, p = p_hidden)
  fc4  = mx.symbol.FullyConnected(data = drop3, name='fc4', num_hidden=2)
  mlp  = mx.symbol.SoftmaxOutput(data = fc4, name = 'softmax')
  return mlp

net = get_mlp_drop(256, 256, 256, 0.2, 0.75)
# (256/256/256 con risposta a 2 categorie)
# prob. di dropout 0.5 per strati latenti, 0.2 per la risposta
```

```
# Stima del modello
model = mx.model.FeedForward(
              symbol = net,
              ctx = mx.gpu(0), # mx.cpu() -> per utilizzare la CPU
              num_epoch = 100,
              optimizer = "adam",
              initializer = mx.initializer.Normal(0.01),
              numpy_batch_size = 1024,
              wd = 0.1)

model.fit(
              X = X_train,
              y = y_train,
              eval_metric = 'acc',
              epoch_end_callback = mx.callback.do_checkpoint('checkpoint/
                  chkpt')
              )

# Calcolo della previsione con Monte Carlo dropout
n_epoche_tot = 100
n_epoche_mc = 100
MCdropout = []
eval_errors = []
for i in range(n_epoche_mc):
    m = mx.model.FeedForward.load(prefix='checkpoint/chkpt', epoch=n_
        epoche_tot-i)
    p = m.predict(X_test)
    p = p[:, 1]
    MCdropout.append(p)
    MCDROPOUT = np.array(MCdropout)
    preds = mean(MCDROPOUT, 0) > 0.5
    eval_errors.append([n_epoche_tot-i, sum(preds != y_test)/len(y_test)])

eval_errors[n_epoche_mc-1][1]
```

Bibliografia

Achanta, Rakesh e Trevor Hastie. «Telugu OCR Framework using Deep Learning». In: *arXiv preprint arXiv:1509.05962* (2015).

Azzalini, Adelchi e Bruno Scarpa. *Data analysis and data mining: An introduction*. OUP USA, 2012.

Barron, Andrew R. «Universal approximation bounds for superpositions of a sigmoidal function». In: *IEEE Transactions on Information theory* 39.3 (1993), pp. 930–945.

Bengio, Yoshua. «Practical recommendations for gradient-based training of deep architectures». In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 437–478.

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

Bishop, Christopher M.. *Pattern recognition and machine learning*. Springer, 2006.

Blei, David M, Alp Kucukelbir e Jon D McAuliffe. «Variational inference: A review for statisticians». In: *arXiv preprint arXiv:1601.00670* (2016).

Boas, Harold e Dmitry Khavinson. «Bohr's power series theorem in several variables». In: *Proceedings of the American Mathematical Society* 125.10 (1997), pp. 2975–2979.

Breiman, Leo. «Bagging predictors». In: *Machine learning* 24.2 (1996), pp. 123–140.

— «Random forests». In: *Machine learning* 45.1 (2001), pp. 5–32.

Chen, Tianqi e Carlos Guestrin. «Xgboost: A scalable tree boosting system». In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2016, pp. 785–794.

Chen, Tianqi, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang e Zheng Zhang. «Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems». In: *arXiv preprint arXiv:1512.01274* (2015).

Cybenko, George. «Approximation by superpositions of a sigmoidal function». In: *Mathematics of Control, Signals, and Systems (MCSS)* 2.4 (1989), pp. 303–314.

Delalleau, Olivier e Yoshua Bengio. «Shallow vs. deep sum-product networks». In: *Advances in Neural Information Processing Systems*. 2011, pp. 666–674.

Denison, David GT. *Bayesian methods for nonlinear classification and regression*. Vol. 386. John Wiley & Sons, 2002.

Docs.nvidia.com. *CUDA Toolkit Documentation*. URL: `http://docs.nvidia.com/cuda/#axzz4aC4fnHkp`.

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

Efron, Bradley e Trevor Hastie. *Computer Age Statistical Inference*. Vol. 5. Cambridge University Press, 2016.

Eldan, Ronen e Ohad Shamir. «The power of depth for feedforward neural networks». In: *arXiv preprint* (2015).

Elman, Jeffrey L. «Finding structure in time». In: *Cognitive science* 14.2 (1990), pp. 179–211.

Finnoff, William, Ferdinand Hergert e Hans Georg Zimmermann. «Improving model selection by nonconvergent methods». In: *Neural Networks* 6.6 (1993), pp. 771–783.

Fisher, Ronald A. «The logic of inductive inference». In: *Journal of the Royal Statistical Society* 98.1 (1935), pp. 39–82.

Freund, Yoav e Robert E Schapire. «A desicion-theoretic generalization of on-line learning and an application to boosting». In: *European conference on computational learning theory*. Springer. 1995, pp. 23–37.

Friedman, Jerome H. «Greedy function approximation: a gradient boosting machine». In: *Annals of statistics* (2001), pp. 1189–1232.

— «Multivariate adaptive regression splines». In: *The annals of statistics* (1991), pp. 1–67.

Friedman, Jerome H e Werner Stuetzle. «Projection pursuit regression». In: *Journal of the American statistical Association* 76.376 (1981), pp. 817–823.

Gal, Yarin e Zoubin Ghahramani. «Dropout as a Bayesian Approximation: Appendix». In: *arXiv preprint arXiv:1506.02157* (2015).

— «Dropout as a Bayesian approximation: Insights and applications». In: *Deep Learning Workshop, ICML*. 2015.

— «Dropout as a Bayesian approximation: Representing model uncertainty in deep learning». In: *arXiv preprint arXiv:1506.02142* 2 (2015).

— «On modern deep learning and variational inference». In: *Advances in Approximate Bayesian Inference workshop, NIPS*. 2015.

Gelman, Andrew, John B Carlin, Hal S Stern e Donald B Rubin. *Bayesian data analysis*. Vol. 2. Chapman & Hall/CRC Boca Raton, FL, USA, 2014.

Github.com. *GitHub - dmlc*. URL: `https://github.com/dmlc/mxnet`.

Glorot, Xavier, Antoine Bordes e Yoshua Bengio. «Deep Sparse Rectifier Neural Networks.» In: *Aistats*. Vol. 15. 106. 2011, p. 275.

Goodfellow, Ian, Yoshua Bengio e Aaron Courville. *Deep learning*. MIT Press, 2016.

Graves, Alex. «Practical variational inference for neural networks». In: *Advances in Neural Information Processing Systems*. 2011, pp. 2348–2356.

Hastie, Trevor, Jerome Friedman e Robert Tibshirani. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York, NY: Springer-Verlag New York, 2009.

Heaton, Jeff. *Artificial Intelligence for Humans: Deep Learning and Neural Networks*. Vol. 3. Heaton Research, Inc., 2015.

Hegde, Vishakh e Sheema Usmani. «Parallel and Distributed Deep Learning». In: (2016).

Hernández-Lobato, José Miguel e Ryan Adams. «Probabilistic Backpropagation for Scalable Learning of Bayesian Neural Networks.» In: *ICML*. 2015, pp. 1861–1869.

Hinton, Geoffrey E, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever e Ruslan R Salakhutdinov. «Improving neural networks by preventing co-adaptation of feature detectors». In: *arXiv preprint arXiv:1207.0580* (2012).

Hornik, Kurt, Maxwell Stinchcombe e Halbert White. «Multilayer feedforward networks are universal approximators». In: *Neural networks* 2.5 (1989), pp. 359–366.

Ioffe, Sergey e Christian Szegedy. «Batch normalization: Accelerating deep network training by reducing internal covariate shift». In: *arXiv preprint arXiv:1502.03167* (2015).

Jensen, Johan Ludwig William Valdemar. «Sur les fonctions convexes et les inégalités entre les valeurs moyennes». In: *Acta mathematica* 30.1 (1906), pp. 175–193.

Kaggle.com. *Kaggle: Your Home for Data Science*. URL: `https://www.kaggle.com/`.

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

Krogh, Anders e John A Hertz. «A simple weight decay can improve generalization». In: *NIPS*. Vol. 4. 1991, pp. 950–957.

Kullback, Solomon e Richard A Leibler. «On information and sufficiency». In: *The annals of mathematical statistics* 22.1 (1951), pp. 79–86.

LeCun, Yann, Yoshua Bengio e Geoffrey Hinton. «Deep learning». In: *Nature* 521.7553 (2015), pp. 436–444.

LeCun, Yann, Léon Bottou, Yoshua Bengio e Patrick Haffner. «Gradient-based learning applied to document recognition». In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

LeCun, Yann, Corinna Cortes e Christopher JC Burges. *The MNIST database of handwritten digits*. 1998.

Lewis, David D, Yiming Yang, Tony G Rose e Fan Li. «Rcv1: A new benchmark collection for text categorization research». In: *Journal of machine learning research* 5.Apr (2004), pp. 361–397.

Liang, Shiyu e R Srikant. «Why Deep Neural Networks?» In: *arXiv preprint arXiv:1610.04161* (2016).

Lin, Henry W e Max Tegmark. «Why does deep and cheap learning work so well?» In: *arXiv preprint arXiv:1608.08225* (2016).

Maas, Andrew L, Awni Y Hannun e Andrew Y Ng. «Rectifier nonlinearities improve neural network acoustic models». In: *Proc. ICML*. Vol. 30. 1. 2013.

Miller, Alan J. «Statistical Algorithms: Algorithm AS 274: Least squares routines to supplement those of Gentleman». In: 41.2 (1992). See correction Miller 1994, pp. 458–478. ISSN: 0035-9254 (print), 1467-9876 (electronic). URL: http://lib.stat.cmu.edu/apstat/274.

Mitchell, Tom M et al. *Machine learning. WCB*. 1997.

Mohamed, Shakir. *A statistical view of deep lear ning*. 2015.

Mxnet.io. *MXNet Documents*. URL: http://mxnet.io/.

Mxnet.io. *Overview — mxnet 0.9.3 documentation*. URL: http://mxnet.io/get_started/setup.html.

Mxnet.io. *Run MXNet on Multiple CPU/GPUs with Data Parallel*. URL: http://mxnet.io/how_to/multi_devices.html.

Neal, Radford M. «Bayesian learning for neural networks». In: (1996).

Ripley, Brian D. *Pattern Recognition and Neural Networks*. Cambridge University Press, 1996.

Ross, Sheldon M. *Introduction to probability models*. Academic press, 2014.

Ruder, Sebastian. «An overview of gradient descent optimization algorithms». In: *arXiv preprint arXiv:1609.04747* (2016).

Scardapane, Simone, Danilo Comminiello, Amir Hussain e Aurelio Uncini. «Group Sparse Regularization for Deep Neural Networks». In: *arXiv preprint arXiv:1607.00485* (2016).

Srebro, Nathan e Adi Shraibman. «Rank, trace-norm and max-norm». In: *International Conference on Computational Learning Theory*. Springer. 2005, pp. 545–560.

Srivastava, Nitish, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever e Ruslan Salakhutdinov. «Dropout: a simple way to prevent neural networks from overfitting.» In: *Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

Telgarsky, Matus. «Benefits of depth in neural networks». In: *arXiv preprint arXiv:1602.04485* (2016).

Wager, Stefan, Sida Wang e Percy S Liang. «Dropout training as adaptive regularization». In: *Advances in neural information processing systems*. 2013, pp. 351–359.

Wang, Hao e Dit-Yan Yeung. «Towards Bayesian Deep Learning: A Survey». In: *arXiv preprint arXiv:1604.01662* (2016).

Williams, DRGHR e GE Hinton. «Learning representations by back-propagating errors». In: *Nature* 323.6088 (1986), pp. 533–538.

Zeiler, Matthew D. «ADADELTA: an adaptive learning rate method». In: *arXiv preprint arXiv:1212.5701* (2012).

Zhang, Harry. «The optimality of naive Bayes». In: *AA* 1.2 (2004), p. 3.