

UNIVERSITÀ DI BOLOGNA

Attività progettuale di Sistemi Intelligenti M

Reti neurali artificiali: dal MLP agli ultimi modelli di CNN

Autore:

Ali Alessio Salman

28 agosto 2017

Indice

1	Reti Neurali Artificiali: le basi	1
1.1	Breve introduzione	1
1.2	Multi-layer Perceptron	2
1.2.1	Strati Nascosti	3
1.3	Caso di studio: prevedere il profitto di un ristorante	3
2	Implementazione di un MLP	5
2.1	Dataset e Architettura	5
2.2	Forward Propagation	6
2.3	Backpropagation	8
2.4	Verifica numerica del gradiente	11
2.5	Addestramento	13
2.5.1	Apprendimento supervisionato	13
2.5.2	Discesa del gradiente	13
2.6	Ottimizzazione: diverse tecniche	14
2.7	Overfitting	18
2.7.1	Rilevare l'overfitting	18
2.7.2	Contromisure	19
2.8	Risultati	19
3	Reti Neurali Convolutionali	21
3.1	Breve introduzione	21
3.2	Architettura	22
3.2.1	Strato di Convoluzione	22
3.2.2	Strato di Pooling	22
3.2.3	Strato completamente connesso (FC)	22
3.3	Applicazioni e risultati	22
4	CNN: implementazione e addestramento	23
4.1	Il framework Torch	23
4.2	Modello di CNN basato su LeNet	23
4.2.1	Strato di Convoluzione	23
4.2.2	Strato di Pooling	23
4.2.3	Strato completamente connesso (FC)	23
4.3	Dataset d'esempio: CIFAR	23
4.4	Addestramento su CIFAR	23
5	Addestrare un modello allo stato dell'arte	25
5.1	ResNet	25
5.2	Implementazione di Resnet di Facebook	25
5.3	Addestramento su CIFAR	25
5.4	LeNet vs Resnet: confronto	25

6	Caso d'uso attuale: fine-tuning su dataset arbitrario	27
6.1	Dataset	27
6.2	Fine-Tuning	27
6.3	ModelZoo	27
6.3.1	Fine-tuning su Resnet	27
7	Conclusioni	29
A	MLP: Codice aggiuntivo	31
A.1	Classi in Lua	31
A.2	La classe Neural_Network	32
A.3	Metodi getter e setter	33
B	Il framework Torch	35
	Bibliografia	37

Capitolo 1

Reti Neurali Artificiali: le basi

1.1 Breve introduzione

Una rete neurale artificiale – chiamata normalmente solo rete neurale (in inglese *Neural Network*) – è un modello di calcolo adattivo, ispirato ai principi di funzionamento del sistema nervoso degli organismi evoluti che secondo l'approccio connessionista[1] possiede una complessità non descrivibile con i metodi simbolici. La caratteristica fondamentale di una rete neurale è che essa è capace di acquisire conoscenza modificando la propria struttura in base alle informazioni esterne (i dati in ingresso) e interne (tramite le connessioni) durante il processo di apprendimento. Le informazioni sono immagazzinate nei parametri della rete, in particolare, nei pesi associati alle connessioni. Sono strutture non lineari in grado di simulare relazioni complesse tra ingressi e uscite che altre funzioni analitiche non sarebbero in grado di fare.

L'unità base di questa rete è il neurone artificiale introdotto per la prima volta da McCulloch e Pitts nel 1943 (fig. 1.1).

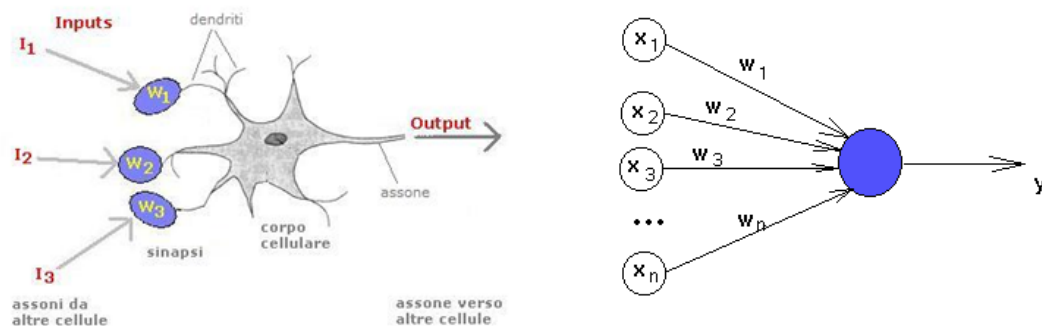


Figura 1.1: Modello di calcolo di un neurone (a sinistra) e schema del neurone artificiale (a destra)

Si tratta di un'unità di calcolo a N ingressi e 1 uscita. Come si può vedere dall'immagine a sinistra gli ingressi rappresentano le terminazioni sinaptiche, quindi sono le uscite di altrettanti neuroni artificiali. A ogni ingresso corrisponde un peso sinaptico w , che stabilisce quanto quel collegamento sinaptico influisca sull'uscita del neurone. Si determina quindi il potenziale del neurone facendo una somma degli ingressi, pesata secondo i pesi w .

A questa viene applicata una funzione di trasferimento non lineare:

$$f(x) = H\left(\sum_i (w_i x_i)\right) \quad (1.1)$$

ove H è la funzione gradino di Heaviside [2]. Vi sono, come vedremo, diverse altre funzioni non lineari tipicamente utilizzate come funzioni di attivazioni dei neuroni. Nel '58 Rosenblatt propone il modello di *Percettrone* rifinendo il modello di neurone a soglia, aggiungendo un termine di *bias* e un algoritmo di apprendimento basato sulla minimizzazione dell'errore, cosiddetto *error back-propagation*[3].

$$f(x) = H\left(\sum_i (w_i x_i) + b\right), \quad \text{ove } b = \text{bias} \quad (1.2)$$

$$w_i(t+1) = w_i(t) + \eta \delta x_i(t) \quad (1.3)$$

dove η è una costante di apprendimento strettamente positiva che regola la velocità di apprendimento, detta *learning rate* e δ è la discrepanza tra l'output desiderato e l'effettivo output della rete.

Il percettrone però era in grado di imparare solo funzioni linearmente separabili. Una maniera per oltrepassare questo limite è di combinare insieme le risposte di più percettroni, secondo architetture multistrato.

1.2 Multi-layer Perceptron

Il Multi-layer Perceptron (*MLP*) o percettrone multi-strato è un tipo di rete feed-forward che mappa un set di input ad un set di output. È la naturale estensione del percettrone singolo e permette di distinguere dati non linearmente separabili.

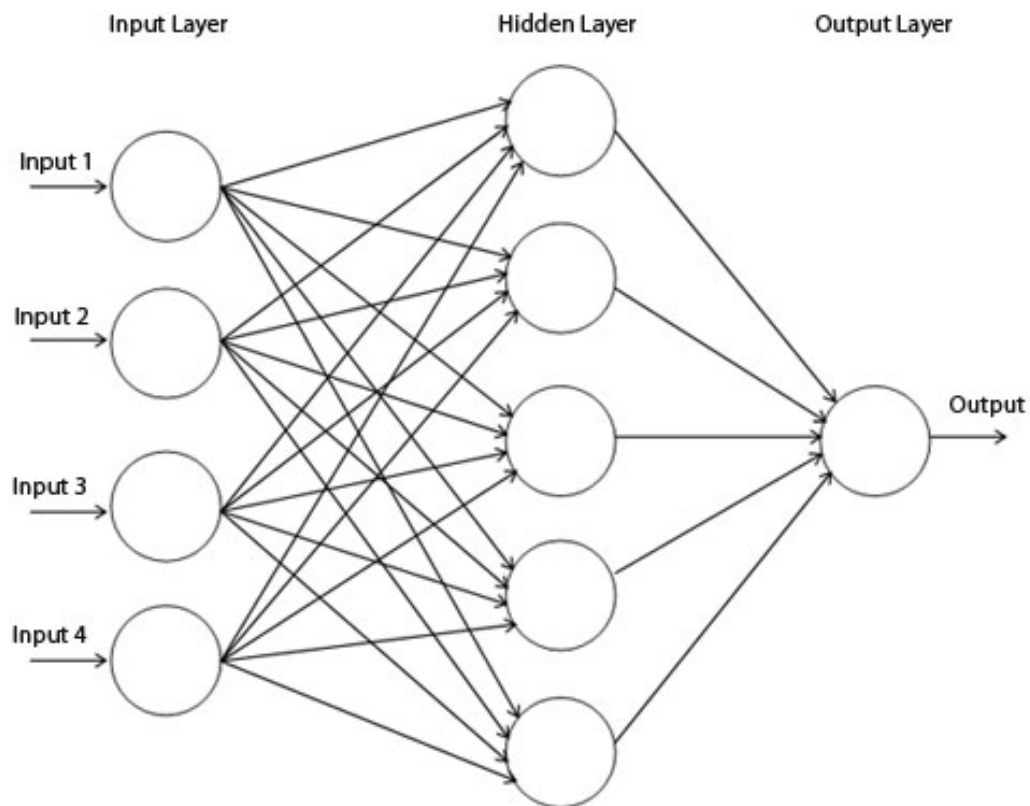


Figura 1.2: Struttura di un percettrone multistrato con un solo strato nascosto

Il *mlp* possiede le seguenti caratteristiche:

- ogni neurone è un percettrone come quello descritto nella sezione 1.1. Ogni unità possiede quindi una propria funzione d'attivazione non lineare.
- a ogni connessione tra due neuroni corrisponde un peso sinaptico w
- è formato da 3 o più strati. In 1.2 è mostrato un MLP con uno strato di input; un solo strato nascosto (o *hidden layer*; ed uno di output.)
- l'uscita di ogni neurone dello strato precedente è l'ingresso per ogni neurone dello strato successivo. È quindi una rete *completamente connessa*. Tuttavia, si possono disconnettere selettivamente settando il peso sinaptico w a 0.
- la dimensione dell'input e la dimensione dell'output dipendono dal numero di neuroni di questi due strati. Il numero di neuroni dello strato nascosto è invece indipendente, anche se influenza di molto le capacità di apprendimento della rete.

Se ogni neurone utilizzasse una funzione lineare allora si potrebbe ridurre l'intera rete ad una composizione di funzioni lineari. Per questo - come detto prima - ogni neurone possiede una funzione di attivazione non lineare.

1.2.1 Strati Nascosti

I cosiddetti *hidden layers* sono una parte molto interessante della rete. Per il teorema di approssimazione universale [4], [4] [1] una rete con un singolo strato nascosto e un numero finito di neuroni, può essere addestrata per approssimare una qualsiasi funzione continua su uno spazio compatto di \mathbb{R}^n . In altre parole, un singolo strato nascosto è abbastanza potente da imparare un ampio numero di funzioni. Precisamente, una rete a 3 strati è in grado di separare regioni convesse con un numero di lati \leq numero neuroni nascosti.

Reti con un numero di strati nascosti maggiore di 3 vengono chiamate reti neurali profonde o *deep neural network*; esse sono in grado di separare regioni qualsiasi, quindi di approssimare praticamente qualsiasi funzione. Il primo e l'ultimo strato devono avere un numero di neuroni pari alla dimensione dello spazio di ingresso e quello di uscita. Queste sono le terminazioni della "*black box*" che rappresenta la funzione che vogliamo approssimare.

L'aggiunta di ulteriori strati non cambia *formalmente* il numero di funzioni che si possono approssimare; tuttavia vedremo che nella pratica un numero elevato di strati migliori di gran lunga le performance della rete su determinati task, essendo gli hidden layers gli strati dove la rete memorizza la propria rappresentazione astratta dei dati in ingresso. Nel capitolo 4 vedremo un'architettura all'avanguardia con addirittura 152 strati.

1.3 Caso di studio: prevedere il profitto di un ristorante

Prendendo spunto dalla traccia d'esame di Sistemi Intelligenti M del 2 Aprile 2009:

"Loris è figlio della titolare di una famoso spaccio di piadine nel Riminese e sta tornando in Italia dopo aver frequentato con successo un prestigioso Master in Business Administration ad Harvard, a cui si è iscritto inseguendo il sogno di esportare in tutto il mondo la piadina romagnola. Nel lungo viaggio in prima classe, medita su come presentare alla mamma, che sa essere un tantino restia alle innovazioni, il progetto di aprire un ristorante a New York City."

Loris ha esportato con successo la piadina a NY, si veda [5] ma col passare degli anni ha notato alcuni problemi e vuole utilizzare di nuovo le sue brillanti capacità analitiche per migliorare il profitto del suo ambizioso ristorante.

I problemi sono 2:

1. il ristorante è conosciuto ormai - si sa che tutti vogliono mangiare italiano - ma il numero dei coperti era rimasto a 22, come quelli iniziali;
2. gli orari di apertura sono troppo lunghi e vi sono alcune zone morte dove il costo di mantenere aperto il ristorante è maggiore rispetto al ricavo dei pochi clienti che si siedono a mangiare durante quelle ore;

Secondo la National Restaurant Association[6] [7] il profitto medio lordo annuo di un ristorante negli Stati Uniti varia dal 2 al 6%. Così Loris ha collezionato alcuni dati riguardo gli ultimi anni e - attratto da tutta quest'entusiasmo attorno alle reti neurali - decide di provare ad utilizzarle per trovare il trade-off ottimale di coperti e di orari di apertura settimanali per massimizzare il profitto del suo ristorante.

Dati questi presupposti, si vedrà nel capitolo 2 come implementare da zero un multi-layer perceptron e addestrarlo al suddetto scopo.

Capitolo 2

Implementazione di un MLP

Per il caso di studio introdotto nel Capitolo 1 si è implementato da zero un percettrone multistrato a 3 strati come quello illustrato in sezione 1.2. Vediamo qui di seguito le varie parti da implementare passo passo per costruire un MLP, addestrarlo e verificare che l'addestramento sia stato eseguito in maniera corretta. Il progetto è realizzato in Lua, per utilizzare il framework per il *Machine Learning Torch* (si veda B) e mantenere le consistenza con i capitoli successivi, nei quali si userà ancora Torch per addestrare reti neurali molto più complesse.

2.1 Dataset e Architettura

Nei vari anni, Loris ha cambiato le due variabili in gioco annotando di volta in volta i risultati. Siccome i coperti erano troppo pochi e le file d'attesa erano troppo lunghe, il ristorante perdeva alcuni clienti. Quindi Loris ha portato i coperti a 25 e diminuendo le ore settimanali a 38, ed i profitti sono aumentati. Tuttavia, non era raro che ancora qualche cliente dovesse aspettare in piedi per troppo tempo (si sa la vita a NY è frenetica), finendo poi per scegliere un ristorante adiacente. Inoltre, aveva diminuito le ore di troppo; nel weekend i clienti arrivavano fino a tardi, quindi rimanere aperti un'ora in più sarebbe stato lungimirante. Così, dopo l'allargamento della sala principale, ha aggiunto altri coperti ed aumentato le ore settimanali a 40, segnando un record personale di 4.4% di profitti annui.

Quindi, i dati in ingresso ed in uscita, in X e Y rispettivamente sono:

$$X = \begin{pmatrix} 22 & 42 \\ 25 & 38 \\ 30 & 40 \end{pmatrix} \quad Y = \begin{pmatrix} 2.8 \\ 3.4 \\ 4.4 \end{pmatrix}$$

Osservando le dimensioni dei dati si nota che la rete deve avere 2 input e dare in uscita 1 output, che chiameremo \hat{y} , in contrapposizione a y che è l'uscita desiderata. Per quanto detto nella sezione 1.2, il MLP deve avere 2 neuroni nello strato di ingresso ed 1 solo in uscita. Inoltre, avrà uno strato nascosto con 3 neuroni. La dimensione di ogni strato fa parte di un insieme di parametri che viene deciso "a mano" sperimentando, i cosiddetti *hyperparameters*. Questi parametri non vengono aggiornati durante l'addestramento - come i pesi della rete - ma vengono decisi a priori.

In figura 2.1 è mostrata l'architettura generale della nostra rete.

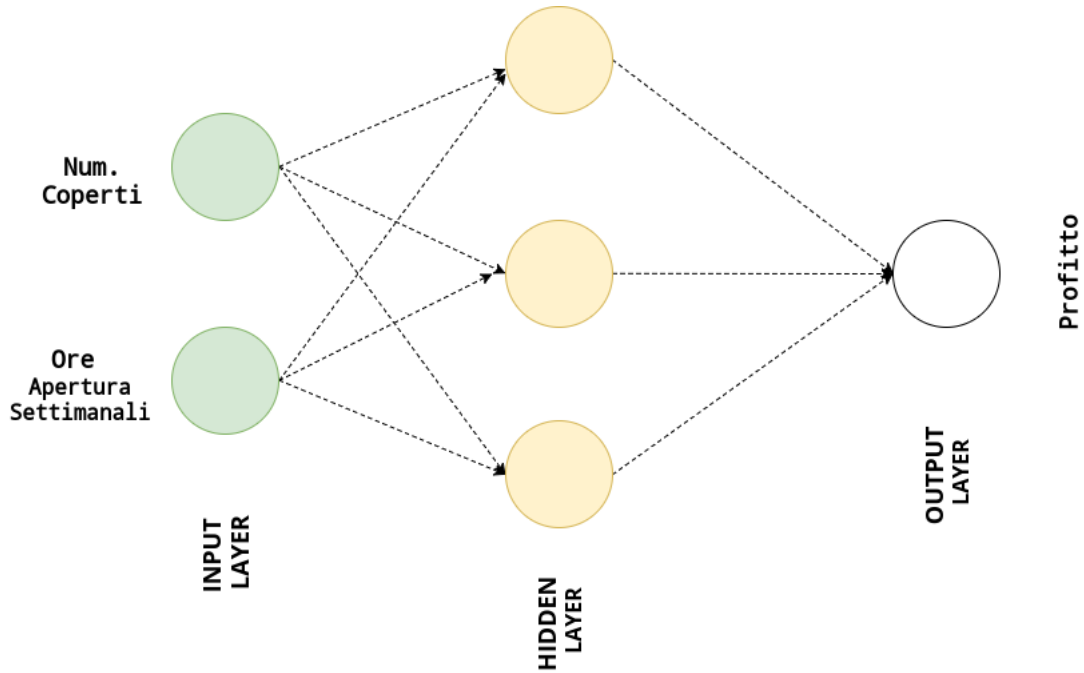


Figura 2.1: Architettura del MLP per la previsione dei profitti

Di seguito gli snippet di codice per la definizione dei dati e dell'architettura della rete secondo lo schema appena presentato.

```

1 ----- Part 1 -----
2 th = require 'torch'
3 bestProfit = 6.0
4 -- X = (num coperti, ore di apertura settimanali), y = profitto lordo
   annuo in percentuale
5 torch.setdefaulttensortype('torch.DoubleTensor')
6 X = th.Tensor({{22,42}, {25,38}, {30,40}})
7 y = th.Tensor({{2.8},{3.4},{4.4}})
8
9 --normalize
10 normalizeTensorAlongCols(X)
11 y = y/bestProfit

```

```

1 ----- Part 2 -----
2 --creating the NN class in Lua, using a nice class utility
3 class=require 'class'
4 local Neural_Network = class('Neural_Network')
5
6 function Neural_Network:__init(inputs, hidden, outputs)
7     self.inputLayerSize = inputs
8     self.hiddenLayerSize = hidden
9     self.outputLayerSize = outputs
10    self.W1 = th.randn(net.inputLayerSize, self.hiddenLayerSize)
11    self.W2 = th.randn(net.hiddenLayerSize, self.outputLayerSize)
12 end

```

2.2 Forward Propagation

Nella tabella 2.1 sono elencate le variabili della rete. Gli input dei layer indicati con z possono anche essere chiamati "attività dei layer" (indicando l'attività sulle loro

Tabella 2.1: Variabili usate nel testo e nel codice

Variabili			
S. Codice	S. Matematico	Definizione	Dimensione
X	X	Esempi, 1 per riga	(numEsempi, inputLayerSize)
y	y	uscita desiderata	(numEsempi, outputLayerSize)
W1	$W^{(1)}$	Pesi layer 1	(inputLayerSize, hiddenLayerSize)
W2	$W^{(2)}$	Pesi layer 2	(hiddenLayerSize, outputLayerSize)
z2	$z^{(2)}$	Input layer 2	(numEsempi, hiddenLayerSize)
a2	$a^{(2)}$	Uscita layer 2	(numEsempi, hiddenLayerSize)
z3	$z^{(3)}$	Input layer 3	(numEsempi, outputLayerSize)

sinapsi); e $a^{(2)}$ indica l'uscita del neurone dopo aver applicato la sommatoria e la funzione di attivazione sulle attività provenienti dal layer precedente.

Per muovere i dati in parallelo attraverso la rete si usa la moltiplicazione fra matrici, per questo è molto comodo usare framework che supportano operazioni fra matrici come *Torch*, *Numpy* o *Matlab*. Per prima cosa, gli input del tensore X devono essere moltiplicati e sommati con i pesi del primo layer $W^{(1)}$, ottenendo l'ingresso per l'hidden layer:

$$z^{(2)} = XW^{(1)} \quad (1)$$

Si noti che $z^{(2)}$ è di dimensione 3×3 , essendo X e $W^{(1)}$ di dimensione 3×2 e 2×3 rispettivamente.

Ora bisogna applicare la funzione di attivazione a $z^{(2)}$. Vi sono diverse funzioni di attivazione utilizzate per le reti neurali. Una delle prime a diventare popolare fu la funzione *sigmoide*[8], utilizzata per questa rete. Vedremo nei capitoli successivi funzioni più efficaci.

$$a^{(2)} = f(z^{(2)}), \quad \text{ove } f = \text{sigmoide} \quad (2)$$

Per completare la *forward propagation*, bisogna seguire lo stesso procedimento per lo strato di output: sommare i contributi provenienti dall'hidden layer ed applicare la sigmoide:

$$z^{(3)} = a^{(2)}W^{(2)} \quad (3)$$

$$\hat{y} = f(z^{(3)}) \quad (4)$$

Essendo $a^{(2)}$ di dimensione 3×3 e $W^{(2)}$ 3×1 l'output \hat{y} sarà anch'esso di dimensione 3×1 , risultando quindi in una previsione per ogni esempio in ingresso.

Si noti come la moltiplicazione fra matrici renda tutto esprimibile in poche righe di codice.

```

1 --Note: I didn't implement manually the sigmoid function as Torch has one
  built-in.
2 --define a forward method
3 function Neural_Network:forward(X)
4     --Propagate inputs through network
5     self.z2 = th.mm(X, self.W1) --matrix multiplication
6     self.a2 = th.sigmoid(self.z2)
7     self.z3 = th.mm(self.a2, self.W2)
8     yHat = th.sigmoid(self.z3)
9     return yHat
10 end

```

2.3 Backpropagation

Come si fa ad addestrare una rete multistrato con diversi neuroni per strato, ognuno dei quali con uscita non lineare? Tramite l'algoritmo di *backpropagation of errors*, ideato da Rumelhart-Hinton-Williams nel 1985.

Non si può introdurre l'algoritmo di "*backprop*" senza prima spiegare il concetto di funzione di costo (o *loss function*). Nelle reti neurali (e più specificatamente nell'apprendimento supervisionato[9], si veda anche sezione ??), la funzione di costo misura la discrepanza tra l'uscita desiderata e l'effettivo output della rete. È quindi una misura dell'errore della rete, per cui l'obiettivo dell'apprendimento è trovare il minimo di questa funzione (modificando la struttura interna della rete, ovvero i pesi sinaptici). Come per la funzione di attivazione, anche in questo caso ci sono ampie possibilità di scelta [10] a seconda del task su cui la rete viene addestrata.

E di nuovo, come per la funzione di attivazione, si è scelta una delle funzione di costo più popolari: l'errore quadratico medio.

$$J = \sum_{j=1}^n \frac{1}{2} (y - \hat{y})^2 \quad (5)$$

Da cui, il codice in Lua:

```

1 function Neural_Network:costFunction(X, y)
2     --Compute the cost for given X,y, use weights already stored in class
3     self.yHat = self:forward(X)
4     J = 0.5 * th.sum(th.pow((y-yHat), 2))
5     return J
6 end

```

La *backprop* ha alcuni requisiti:

- Reti stratificate
- Ingressi a valori reali $\in [0, 1]$
- Neuroni non lineari con funzione di uscita sigmoideale (o altra fz. di attivazione derivabile)

Sotto queste condizioni l'algoritmo sfrutta la regola della catena[11] per la derivazione di funzione composte, per calcolare il gradiente della *funzione di costo*. I pesi della rete vengono quindi aggiornati secondo la *discesa del gradiente* (figura2.2); ovvero variano in maniera tale da minimizzare la funzione di costo J .

Viene chiamato *backward propagation of errors* poiché l'errore calcolato a partire dall'output della rete viene distribuito in maniera proporzionale all'indietro, su tutti

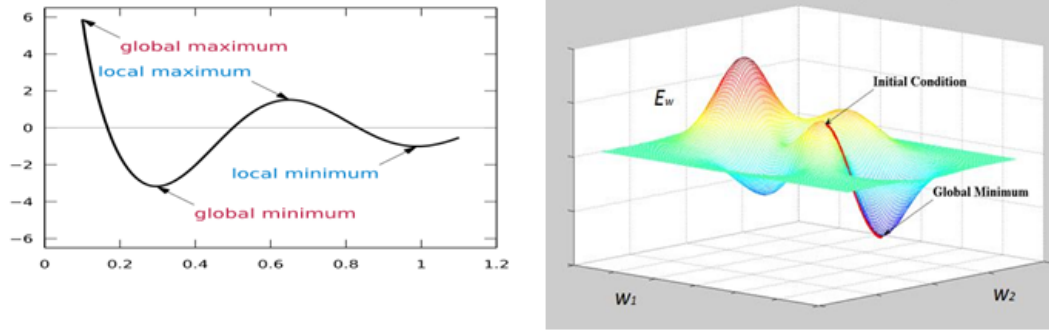


Figura 2.2: Cercare il minimo di una fz. seguendo la discesa del gradiente

i neuroni della rete. È importante quindi, spezzare il calcolo del gradiente dell'errore in derivate parziali, dall'ultimo strato fino al primo, e poi combinarle insieme.

Si noti che le equazioni (1-5) formano una un'unica equazione che lega J a $X, y, W^{(1)}, W^{(2)}$. Tenendo questo in mente, si applica la regola della catena. Partendo dal fattore riguardante lo strato di output si ha:

$$\frac{\partial J}{\partial W^{(2)}} = \sum \frac{\partial \frac{1}{2}(y - \hat{y})^2}{\partial W^{(2)}}$$

Sviluppando i calcoli si ottiene:

$$\frac{\partial J}{\partial W^{(2)}} = -(y - \hat{y}) \frac{\partial \hat{y}}{\partial W^{(2)}}$$

L'equazione (4) indica che \hat{y} è la funzione di attivazione di $z^{(3)}$. Da cui:

$$\frac{\partial J}{\partial W^{(2)}} = -(y - \hat{y}) \frac{\partial \hat{y}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial W^{(2)}}$$

Il 2° membro dell'equazione è semplicemente la derivata della funzione di attivazione sigmoide (fig. 2.3):

$$f(z) = \frac{1}{1 + e^{-z}}$$

$$f'(z) = \frac{e^{-z}}{(1 + e^{-z})^2}$$

Definiamola, quindi, nel codice:

```
1 function Neural_Network:d_Sigmoid(z)
2   --Derivative of sigmoid function
3   return th.exp(-z):cdiv( (th.pow( (1+th.exp(-z)), 2) ) )
4 end
```

L'equazione così ottenuta è:

$$\frac{\partial J}{\partial W^{(2)}} = -(y - \hat{y}) f'(z^{(3)}) \frac{\partial z^{(3)}}{\partial W^{(2)}}$$

Infine, dobbiamo trovare $\frac{\partial z^{(3)}}{\partial W^{(2)}}$, che rappresenta la variazione dell'attività del terzo

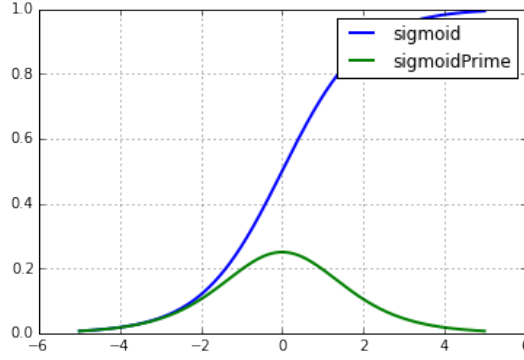


Figura 2.3: La funzione sigmoide e la sua derivata

layer rispetto ai pesi del secondo layer. Richiamando l'equazione (3):

$$z^{(3)} = a^{(2)}W^{(2)}$$

Tralasciando per un attimo la somma tra i vari neuroni, si nota una semplice relazione lineare fra i termini, con a^2 che rappresenta la pendenza. Indi:

$$\frac{\partial z^{(3)}}{\partial W^{(2)}} = a^{(2)}$$

Indicando con $\delta^{(3)}$, l'errore sullo strato di uscita, si ha:

$$\delta^{(3)} = -(y - \hat{y})f'(z^{(3)})$$

Ora bisogna moltiplicare l'errore con $a^{(2)}$. Come indicato nelle ottime dispense di CS231 di Stanford[12]: guardare alle dimensioni delle matrici può essere utile in questo caso. Infatti per fare combaciare le dimensioni, c'è solo una maniera di calcolare la derivata qui, ed è facendo la trasposta di $a^{(2)}$:

$$\frac{\partial J}{\partial W^{(2)}} = (a^{(2)})^T \delta^{(3)}$$

Si noti che la sommatoria che abbiamo tralasciato all'inizio del calcolo viene inclusa "automaticamente" dalle somme delle moltiplicazione fra matrici.

L'ultimo termine da calcolare è $\frac{\partial J}{\partial W^{(1)}}$. Il calcolo è inizialmente simile a quello precedente, iniziando sempre dalla derivata sull'ultimo strato ed utilizzando i risultati trovati in precedenza:

$$\frac{\partial J}{\partial W^{(1)}} = (y - \hat{y}) \frac{\partial \hat{y}}{\partial W^{(1)}}$$

$$\frac{\partial J}{\partial W^{(1)}} = (y - \hat{y}) \frac{\partial \hat{y}}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial W^{(1)}}$$

$$\frac{\partial J}{\partial W^{(1)}} = -(y - \hat{y})f'(z^{(3)}) \frac{\partial z^{(3)}}{\partial W^{(1)}}$$

$$\frac{\partial J}{\partial W^{(1)}} = \delta^{(3)} \frac{\partial z^{(3)}}{\partial W^{(1)}}$$

Ora rimane l'ultimo termine da calcolare, anch'esso da scomporre in diversi fattori

andando a ritroso nella rete:

$$\frac{\partial z^{(3)}}{\partial W^{(1)}} = \frac{\partial z^{(3)}}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial W^{(1)}}$$

Come prima, c'è una relazione lineare tra le sinapsi, ma stavolta la pendenza è data da $W^{(2)}$; anche in questo caso da trasporre.

$$\frac{\partial J}{\partial W^{(1)}} = \delta^{(3)} (W^{(2)})^T \frac{\partial a^{(2)}}{\partial W^{(1)}}$$

$$\frac{\partial J}{\partial W^{(1)}} = \delta^{(3)} (W^{(2)})^T \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial W^{(1)}}$$

$\frac{\partial a^{(2)}}{\partial z^{(2)}}$ è di nuovo la derivata della f di attivazione. Il termine finale del calcolo $\frac{\partial z^{(2)}}{\partial W^{(1)}}$, rappresenta quanto varia l'uscita del primo strato al variare dei pesi. Richiamando l'equazione (1) si nota subito che questo valore è dato dal vettore di input X - come prima - traposto:

$$\frac{\partial J}{\partial W^{(1)}} = X^T \delta^{(3)} (W^{(2)})^T f'(z^{(2)})$$

Chiamando $\delta^{(2)} = \delta^{(3)} (W^{(2)})^T f'(z^{(2)})$ diventa:

$$\frac{\partial J}{\partial W^{(1)}} = X^T \delta^{(2)}$$

Facendo un sommario:

$$\boxed{\frac{\partial J}{\partial W^{(2)}} = (a^{(2)})^T \delta^{(3)}} \quad (6)$$

$$\boxed{\frac{\partial J}{\partial W^{(1)}} = X^T \delta^{(2)}} \quad (7)$$

$$\boxed{\delta^{(2)} = \delta^{(3)} (W^{(2)})^T f'(z^{(2)})} \quad (8)$$

$$\boxed{\delta^{(3)} = -(y - \hat{y}) f'(z^{(3)})} \quad (9)$$

Implementando le equazioni sovrascritte in Lua, la classe `Neural_Network` è quindi completa (per i dettagli si veda l'appendice A).

```

1 function Neural_Network:d_CostFunction(X, y)
2   --Compute derivative wrt to W and W2 for a given X and y
3   self.yHat = self:forward(X)
4   delta3 = th.cmul(-(y-self.yHat), self:d_Sigmoid(self.z3))
5   dJdW2 = th.mm(self.a2:t(), delta3)
6
7   delta2 = th.mm(delta3, self.W2:t()):cmul(self:d_Sigmoid(self.z2))
8   dJdW1 = th.mm(X:t(), delta2)
9
10  return dJdW1, dJdW2
11 end

```

2.4 Verifica numerica del gradiente

Siccome la Backprop è notoriamente difficile da debuggare, una volta che la si usa per l'addestramento di una rete, bisogna controllare se l'implementazione della sezione

precedente è corretta prima di proseguire nel progetto. A questo scopo, è stata scritta una funzione per il calcolo *numerico* del gradiente che andrà poi confrontata con il calcolo computato dalla Backprop.

L'algoritmo[13] è basato sulla seguente definizione di derivata:

$$\frac{d}{d\theta} J(\theta) = \lim_{\epsilon \rightarrow 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon)}{2 * \epsilon}$$

Il gradiente che bisogna controllare è formato dai 2 vettori che contengono le derivate dei pesi di tutta la rete: $\frac{\partial J}{\partial W^{(1)}}$ e $\frac{\partial J}{\partial W^{(2)}}$.

Approssimando ϵ con un valore molto piccolo (*i.e.* 10^{-4}) è possibile perturbare singolarmente i singoli pesi della rete (contenuti nei vettori $W_{(1)}$ e $W_{(2)}$) e calcolare così il gradiente in maniera numerica.

Per poterlo fare, servono delle funzioni ausiliare (metodi getter e setter) per prendere e settare i pesi ed il gradiente della rete come singolo vettore "flat" di parametri (appendice A). Dopodiché basta un loop ed un array per memorizzare le derivate dei singoli pesi così calcolati:

```

1 function computeNumericalGradient(NN, X, y)
2     paramsInitial = NN:getParams()
3     numgrad = th.zeros(paramsInitial:size())
4     perturb = th.zeros(paramsInitial:size())
5     e = 1e-4
6
7     for p=1,paramsInitial:nElement() do
8         --Set perturbation vector
9         perturb[p] = e
10        NN:setParams(paramsInitial + perturb)
11        loss2 = NN:costFunction(X, y)
12
13        NN:setParams(paramsInitial - perturb)
14        loss1 = NN:costFunction(X, y)
15
16        --Compute Numerical Gradient
17        numgrad[p] = (loss2 - loss1) / (2*e)
18
19        --Return the value we changed to zero:
20        perturb[p] = 0
21    end
22
23    --Return Params to original value:
24    NN:setParams(paramsInitial)
25    return numgrad
26 end

```

Si può ora inizializzare una rete, eseguire la backpropagation e confrontare i valori con quelli calcolati numericamente:

```

1 --test if we actually make the calculations correctly
2 NN = Neural_Network(2,3,1)
3
4 print('Gradient checking...')
5 numgrad = computeNumericalGradient(NN, X, y)
6 grad = NN:computeGradients(X, y)
7 --[[
8 In order to make an accurate comparison of the 2 vectors
9 we can calculate the difference as the ratio of:
10 numerator --> the norm of the difference
11 denominator--> the norm of the sum
12 Should be in the order of 10^-8 or less
13 --]]

```



```

14 diff = th.norm(grad-numgrad)/th.norm(grad+numgrad)
15 print(string.format('The difference is %e',diff))

```

Per calcolare *quanto* siano effettivamente uguali i due gradienti si può usare un rapporto basato sulla norma della somma e della differenza dei gradienti (si veda il codice sopra). Se la backprop è stata implementata correttamente questa differenza dovrebbe essere nell'ordine di 10^{-8} o inferiore. Difatti, quando si esegue lo script si ottiene:

```

1 $ th 4_gradCheck.lua
2 Gradient checking...
3 The difference is 2.123898e-10

```

2.5 Addestramento

Una volta accertati che l'implementazione della Backprop è corretta si può procedere ad addestrare la rete. Quello che si vuole ottenere è una rete che guardando ai dati accumulati negli anni, riesca a prevedere l'andamento del profitto del ristorante. Nel nostro dataset, ogni esempio è formato da una coppia $\langle n. \text{ coperti}, n. \text{ ore settimanali} \rangle$ a cui è associata un'uscita *desiderata*. La rete cercherà di modificare la sua struttura interna (i.e. i pesi sinaptici) "creando" una funzione - la rete stessa rappresenta questa funzione - per riprodurre in maniera più precisa possibile quest'associazione. L'apprendimento sarà quindi di tipo *supervisionato*.

2.5.1 Apprendimento supervisionato

Con questo termine s'intende l'allenamento di un sistema tramite una serie di esempi ideali; l'insieme di questi esempi è chiamato *training set*. Il sistema impara quindi ad approssimare una funzione non nota a priori a partire da una serie di coppie ingresso-uscita: per ogni input in ingresso gli si comunica l'output desiderato. L'apprendimento consiste nella capacità del sistema - tramite la funzione generata con l'allenamento - di generalizzare a nuovi esempi: avendo in ingresso dati non noti deve poter predire in modo corretto l'output desiderato. Matematicamente parlando, questi esempi non noti sono punti del dominio che non fanno parte dell'insieme degli esempi di training. Esistono diversi algoritmi di apprendimento supervisionato ma tutti condividono una caratteristica: l'addestramento viene eseguito mediante la minimizzazione di una funzione di costo (si veda la sezione 2.3). Nella *Learning Rule* in figura 2.4 sono compresi 2 elementi:

1. Calcolo della discrepanza tra l'uscita della rete e l'uscita desiderata e backpropagation;
2. La strategia di aggiornamento dei parametri;

Riguardo all'ultimo punto ci sono diverse possibilità.

2.5.2 Discesa del gradiente

La loss function è una funzione che ha un numero di variabili pari al numero dei pesi della rete. Data la complessità - soprattutto in reti profonde molto più complesse di quella trattata in questo progetto - la ricerca del minimo non è semplice; si corre il rischio di rimanere bloccati in plateau o minimi locali. Per questo, si applicano metodi a raffinamento iterativo: si parte da una soluzione iniziale e si cerca di migliorarla ad

Fase di addestramento

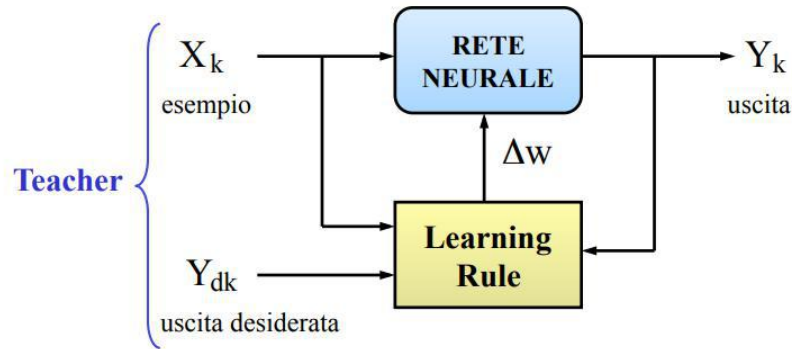


Figura 2.4: Apprendimento supervisionato: schema generale

ogni ciclo. Questo metodo è conosciuto come *Stochastic Gradient Descent*[14]. Calcolando il gradiente ∇J si conosce la direzione di massima variazione quindi ci si sposta - lungo l'opposto di questa direzione, l'antigradiente - di una quantità pari a η . Questo parametro si chiama *learning rate* e regola appunto la velocità dell'apprendimento.

Tornando alla strategia di aggiornamento dei parametri, la più intuitiva è chiamata "*Vanilla*":

$$W_{t+1} = W_t - \eta \nabla J(W_t) \quad (2.1)$$

Questa regola di apprendimento però soffre di alcuni problemi:

- effettua uno spostamento pari a η sia per features frequenti che non. Questo problema è noto come "*sparsità delle features*"
- η è una costante e non è detto che garantisca la convergenza. Si potrebbe "saltare" da un lato all'altro del punto di minimo senza mai trovarlo.
- come detto sopra, i punti di sella e plateau causano problemi. In questo caso il gradiente è nullo e quindi l'aggiornamento dei pesi si azzerà, fermando l'apprendimento.

Per ovviare a questo ed altri problemi, sono stati studiati numerosi metodi. La prossima sezione ne elenca qualcuno utilizzato per questo progetto.

2.6 Ottimizzazione: diverse tecniche

L'ottimizzazione dell'apprendimento delle reti neurali è un argomento vasto ed impervio, ma molto importante. Con addestramenti che possono durare mesi a seconda del tipo di apprendimento sono nate, in relativamente breve tempo, moltissimi metodi di ottimizzazione. Qui si faranno solo dei cenni ai metodi utilizzati in questo progetto.

Prima di elencare tecniche più evolute dell'aggiornamento Vanilla, occorre precisare alcuni punti dello Stochastic Gradient Descent, essendo quest'ultimo il punto di partenza di ogni altro metodo.

- SGD: lo *Stochastic Gradient Descent* è, come lo descrive il nome stesso, una versione stocastica della discesa del gradiente. La discesa del gradiente standard non è scalabile, poiché il gradiente da calcolare tiene conto dell'errore quadratico calcolato *su ogni singolo esempio del dataset*. Come detto poco fa, la loss function ha già di per sé un numero di variabili proporzionale alla complessità della rete; quando il dataset è molto largo, ed è tipico per problemi reali di machine learning, questo calcolo diventa inefficiente. L'SGD risolve questo problema approssimando l'operazione "*gradiente - aggiornamento pesi*" da tutto il dataset al singolo esempio. Questo rende l'approssimazione molto inaccurata, ma molto veloce da elaborare. Nonostante le oscillazioni dovute all'inaccuratezza, questo metodo consente nella pratica, dopo molte iterazioni, di trovare il minimo globale. Questo è soprattutto vero per problemi su larga scala.

Un compromesso tra il metodo standard e quello completamente stocastico è l'utilizzo di "*mini-batch*", cioè piccoli sottoinsiemi del dataset su cui viene eseguita l'iterazione di apprendimento. Se il dataset è abbastanza eterogeneo ed è ordinato in maniera randomica, il mini-batch approssima abbastanza bene l'intero dataset; di conseguenza, l'approssimazione sarà più verosimile al calcolo dell'intero gradiente, aumentando quindi l'accuratezza senza intaccare la velocità del calcolo.

- MOMENTUM & NAG: Il "*Momentum*" controlla la quantità d'inerzia nella modifica dei pesi sinaptici, inserendo nell'equazione la variazione ΔW precedente. INSERIRE FORMULA In questo modo si riducono le oscillazioni nella ricerca della soluzioni permettendo di usare learning rate più alti. È ispirato dal momento nella Fisica, considerando il vettore dei pesi come una particella che viaggia in uno spazio parametrico.

Il *Nesterov Accelerated Gradient* è una versione più raffinata del Momentum update, che elabora una correzione della traiettoria calcolando il gradiente *dopo* aver fatto la somma con il gradiente accumulato precedentemente. Questo Per aiutare a capire la differenza tra i due, si guardi la figura 2.5.

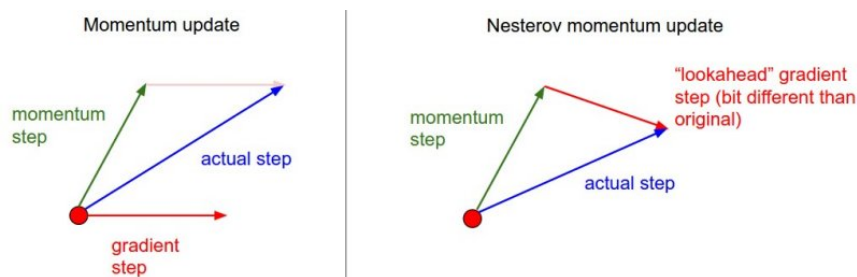


Figura 2.5: Confronto tra il momentum classico e NAG

- BFGS: l'algoritmo *Broyden-Fletcher-Goldfarb-Shanno* fa parte della famiglia "Quasi-Newton" [15]. Questo metodo supera i limiti della discesa del gradiente classica facendo una stima della matrice Hessiana, ovvero della curvatura della superficie della funzione di costo J . Utilizzando questa stima compie degli spostamenti più informati verso la discesa. Nella pratica si usa una versione efficiente che consuma meno memoria, detta Limited-BFGS[16].
- ADAM: l'*Adaptive Moment Estimation* cerca di ovviare ai problemi dell'aggiornamento vanilla modificando il learning rate in maniera diversa per ogni parametro e a seconda dello stadio dell'apprendimento. Fa parte quindi della

famiglia dei metodi *adattivi*. In particolare, l'algoritmo calcola la media con decadimento esponenziale del gradiente e del quadrato del gradiente; i parametri β_1 e β_2 controllano il decadimento di queste medie mobili. Gli autori forniscono i valori consigliati per questi parametri, che sono infatti i valori di default anche in ogni framework che supporta Adam[17].

Per riuscire effettivamente ad addestrare la rete, si è utilizzato un package di ottimizzazione di Torch: `optim`. Quest'ultimo fornisce il supporto a tutte (e più) le tecniche di ottimizzazione viste prima. Si può quindi procedere all'addestramento della rete. Utilizzando un *logger* per mantenere tutti i dati sul training, si possono successivamente plottare le diverse curve di apprendimento per confrontare i diversi metodi.

Definita quindi una classe `Trainer`, si definisce un metodo per addestrare la rete che astrae dalla tecnica di ottimizzazione utilizzata.

```

1 Trainer = class('Trainer')
2 function Trainer:__init(NN)
3     --Make Local reference to network:
4     self.N = NN
5 end
6
7 --Let's train!
8 function Trainer:train(X, y)
9     --variables to keep track of the training
10    local neval = 0
11    --get initial params
12    params0 = self.N:getParams()
13    -- create closure to evaluate f(X) and df/dX
14    -- this is requested by the API of the optim package
15    local feval = function(params0)
16        local f = self.N:costFunction(X, y)
17        print(f)
18        local df_dx = self.N:computeGradients(X, y)
19        neval = neval + 1
20        logger:add{neval, f} --,timer:time().real}
21        return f, df_dx
22    end
23    if optimMethod == optim.cg then
24        newparams,_,_ = optimMethod(feval, params0, optimState)
25    else
26        for i=1,opt.maxIter do
27            newparams,_,_ = optimMethod(feval, params0, optimState)
28            self.N:setParams(newparams)
29        end
30    end
31 end

```

Procedendo iterativamente con le diverse tecniche di ottimizzazione si ottiene il risultato in figura 2.6.

Dal grafico si evidenzia come le premesse teoriche dei vari metodi siano state pienamente rispettate. Nonostante il problema sia molto semplice se comparato ai reali problemi di *deep learning*, già su un dataset ed un numero di iterazioni così limitato si nota la superiorità di un metodo verso il precedente. In particolare, nel corso "CS231" di deep learning applicato alla visione artificiale di Stanford[18], sviluppato da Andrej Karpathy et. al, si raccomanda Adam come metodo di default per applicazioni di deep learning:

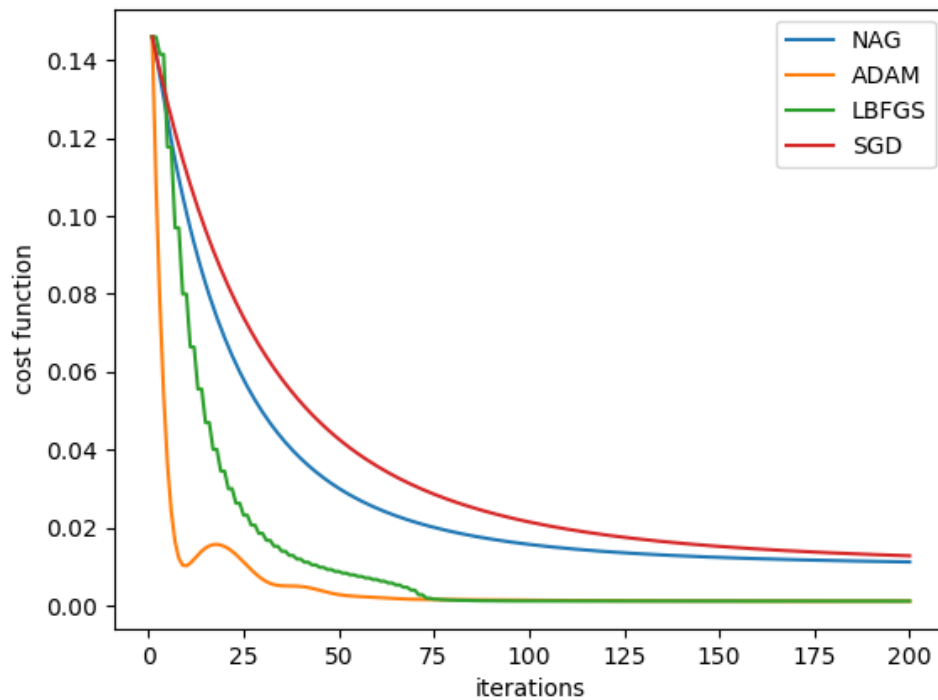


Figura 2.6: Confronto dei metodi di ottimizzazione durante il training

In practice Adam is currently recommended as the default algorithm to use. However, it is often also worth trying SGD+Nesterov Momentum as an alternative.

La rete è ora addestrata: dando in ingresso la matrice X di input si avranno in output previsioni molto più accurate, come mostrato in figura 2.7.

```
th> y
0.01 *
 2.8000
 3.4000
 4.4000
[torch.DoubleTensor of size 3x1]

[0.0003s]
th> nn.forward(X)
0.01 *
 2.7073
 3.5368
 4.3405
[torch.DoubleTensor of size 3x1]
```

Figura 2.7: L'output della rete \hat{y} è vicino all'output desiderato y

2.7 Overfitting

Uno dei problemi più comuni dell'apprendimento automatico è l'*Overfitting*. Esso si verifica laddove il modello creato per fare predizioni risulta troppo complesso e troppo "legato" al solo training set, di cui apprende anche i rumori: la rete è quindi incapace di *generalizzare* ad esempi ancora non visti e, nonostante l'accuratezza estremamente alta sul training set, darà delle pessime predizioni. Questo può essere dovuto a diversi fattori:

- il modello ha troppi parametri rispetto al numero di osservazioni;
- il dataset è costituito da troppi pochi esempi;
- l'addestramento è stato fatto troppo a lungo;

In figura 2.8 è mostrato un esempio di 2 modelli statistici che hanno entrambi errore nullo ma complessità molto diversa.

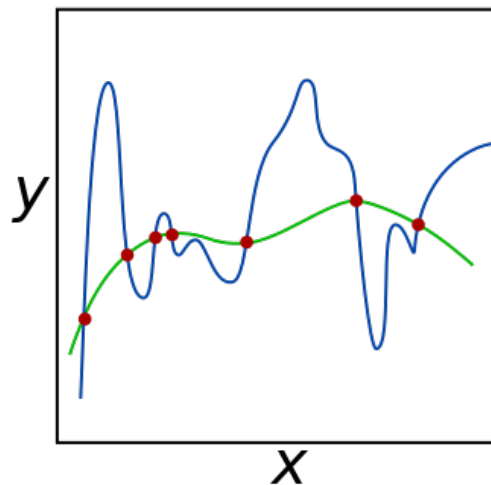


Figura 2.8: Entrambe le funzioni hanno un'errore nullo sul dataset. Tuttavia, la funzione blu è eccessivamente complessa ed affetta da rumore; avrà quindi cattive capacità di generalizzazione. La verde, al contrario, sarà presumibilmente più corretta rispetto ai punti sconosciuti della distribuzione sottostante.

2.7.1 Rilevare l'overfitting

Per rilevare se è avvenuto o meno overfitting durante l'apprendimento; si possono dare in ingresso al perceptrone esempi ancora non visti e controllare "ad occhio" se le predizioni hanno senso. Per averne la certezza però, bisogna dividere il dataset in un training set ed un *test set* su cui testare la rete durante l'apprendimento. Sugli esempi del test set non si farà backpropagation e non avverrà quindi apprendimento; si utilizzeranno solo per sapere quanto è corretto l'addestramento.

Testando la rete *durante* l'apprendimento si può capire il preciso momento in cui avviene l'overfitting. Per farlo, occorre plottare sullo stesso grafico l'errore sul training set e sul test set.

Dopo aver definito arbitrariamente alcuni esempi di testing, si modifica l'algoritmo di training:

```

1 --Need to modify trainer class a bit to check testing error during
  training:
2 function Trainer:train(trainX, trainY, testX, testY)
3     --variables to keep track of the training
4     local neval = 0
5
6     params0 = self.N:getParams()
7     -- create closure to evaluate f(X) and df/dX
8     local feval = function(params0)
9         local f = self.N:costFunction(trainX, trainY)
10        local test = self.N:costFunction(testX, testY)
11        --printing training and testing error
12        print(f..' '..test)
13        local df_dx = self.N:computeGradients(trainX, trainY)
14        neval = neval + 1
15        --logging both training and testing data
16        logger:add(neval, f) --,timer:time().real}
17        testLogger:add(neval, test)
18
19        return f, df_dx
20    end
21
22    if optimMethod == optim.cg then
23        newparams,_,_ = optimMethod(feval, params0, optimState)
24    else
25        for i=1,opt.maxIter do
26            newparams,_,_ = optimMethod(feval, params0, optimState)
27            self.N:setParams(newparams)
28        end
29    end
30 end

```

Si addestra la rete e si loggano i valori di training e di testing, con il seguente codice:

```

1 --now let's train and check where exactly the net is Overfitting
2 nn = Neural_Network(2,3,1)
3
4 init_params = nn:getParams()
5 logtrain = 'train.log'
6 logtest = 'test.log'
7 logger = optim.Logger(logtrain)
8 testLogger = optim.Logger(logtest)
9
10 trainer = Trainer(nn)
11 trainer:train(trainX, trainY, testX, testY)

```

Andando infine a plottare i dati così ottenuti:

2.7.2 Contromisure

2.8 Risultati

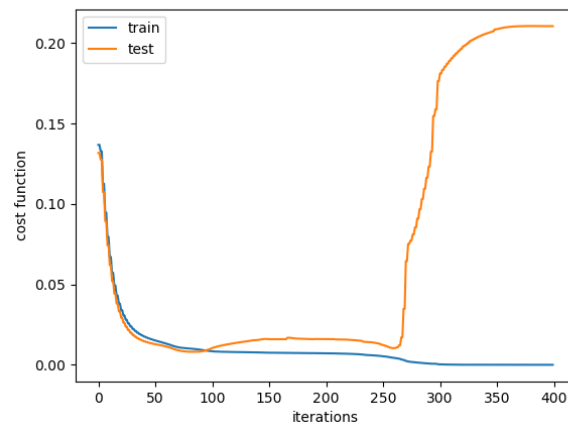


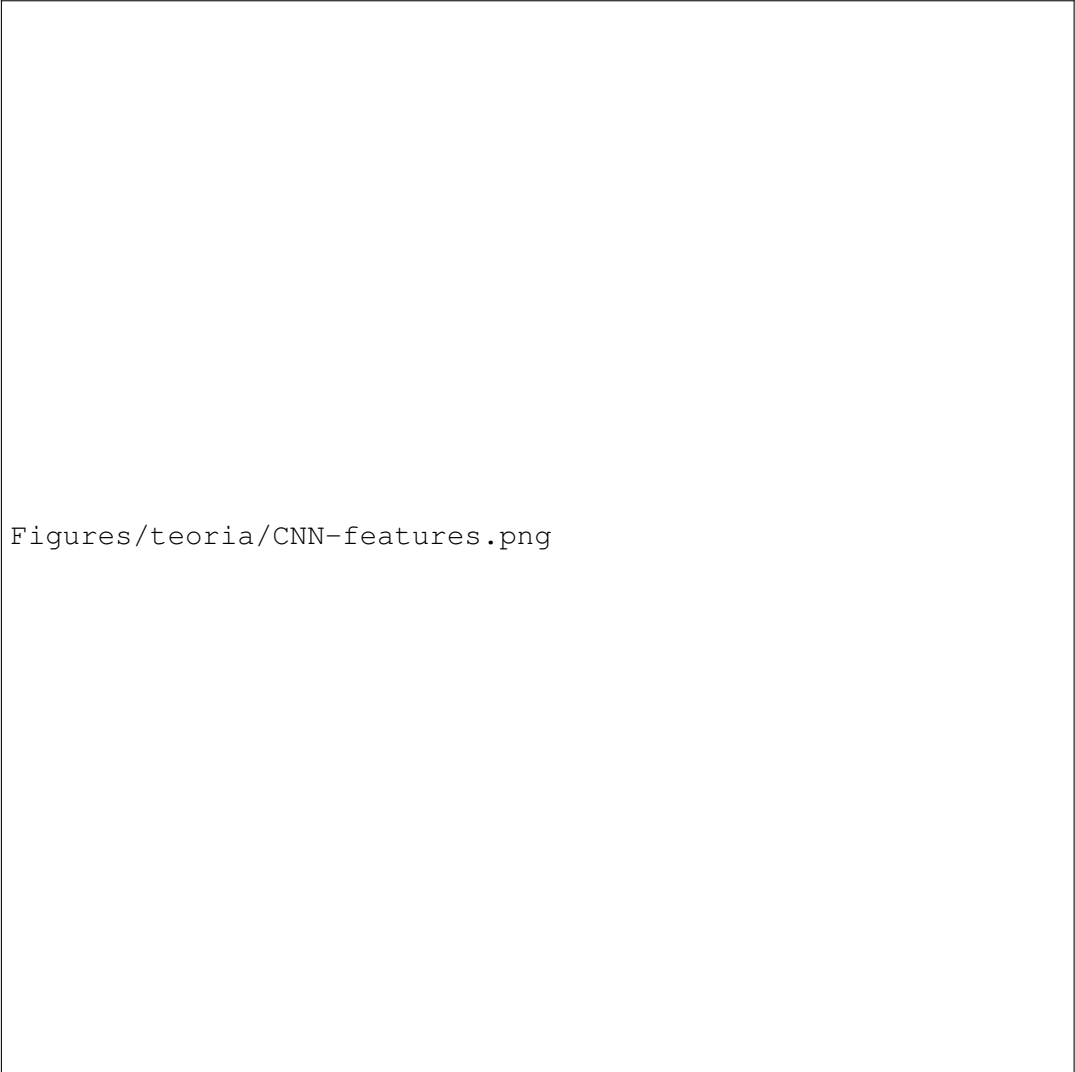
Figura 2.9: La rete mostra overfitting. Dopo l'iterazione 100 le performance sul test set iniziano ad essere sbagliate. Attorno alla 250 il modello diventa troppo complesso e le predizioni sono pessime

Capitolo 3

Reti Neurali Convoluzionali

3.1 Breve introduzione

Le reti neurali convoluzionali, alle quali ci riferiremo con l'abbreviazione *CNN* - dall'inglese *Convolutional Neural Network*, sono un'evoluzione delle normali reti artificiali caratterizzati da una particolare architettura che le ha rese negli anni molto efficaci su diversi compiti.



Figures/teoria/CNN-features.png

Figura 3.1: I diversi strati tipici di una CNN

3.2 Architettura

3.2.1 Strato di Convoluzione

3.2.2 Strato di Pooling

3.2.3 Strato completamente connesso (FC)

3.3 Applicazioni e risultati

Capitolo 4

CNN: implementazione e addestramento

4.1 Il framework Torch

4.2 Modello di CNN basato su LeNet

4.2.1 Strato di Convoluzione

4.2.2 Strato di Pooling

4.2.3 Strato completamente connesso (FC)

4.3 Dataset d'esempio: CIFAR

4.4 Addestramento su CIFAR

Vedremo nel capitolo seguente un confronto dei risultati di una CNN con un'architettura allo stato dell'arte addestrata sullo stesso dataset.

Capitolo 5

Addestrare un modello allo stato dell'arte

5.1 ResNet

5.2 Implementazione di Resnet di Facebook

5.3 Addestramento su CIFAR

5.4 LeNet vs Resnet: confronto

Capitolo 6

Caso d'uso attuale: fine-tuning su dataset arbitrario

Un caso di studio attuale è quello di prendere una rete allo stato dell'arte, già addestrata per mesi su un dataset enorme come quello di *Imagenet* per utilizzarla su un dataset arbitrario a nostra scelta, per scopi industriali o personali.

6.1 Dataset

6.2 Fine-Tuning

6.3 ModelZoo

6.3.1 Fine-tuning su Resnet

Facebook ha reso pubbliche le sue implementazioni di Resnet su github a tutti, cosicché possano essere utilizzate per qualsivoglia esperimento. Ci sono vari modelli, più o meno potenti a seconda degli strati della rete (da 18 a 140).

Capitolo 7

Conclusioni

Appendice A

MLP: Codice aggiuntionale

A.1 Classi in Lua

In Lua manca il costrutto delle classi. Si può tuttavia crearle utilizzando tables e meta-tables. Per realizzare il multi-layer perceptron si è utilizzato una piccola libreria, di seguito riportata.

```

1  -- class.lua
2  -- Compatible with Lua 5.1 (not 5.0).
3  function class(base, init)
4      local c = {} -- a new class instance
5      if not init and type(base) == 'function' then
6          init = base
7          base = nil
8      elseif type(base) == 'table' then
9          -- our new class is a shallow copy of the base class!
10         for i,v in pairs(base) do
11             c[i] = v
12         end
13         c._base = base
14     end
15     -- the class will be the metatable for all its objects,
16     -- and they will look up their methods in it.
17     c.__index = c
18
19     -- expose a constructor which can be called by <classname>(<args>)
20     local mt = {}
21     mt.__call = function(class_tbl, ...)
22         local obj = {}
23         setmetatable(obj, c)
24         if init then
25             init(obj, ...)
26         else
27             -- make sure that any stuff from the base class is initialized!
28             if base and base.init then
29                 base.init(obj, ...)
30             end
31         end
32         return obj
33     end
34     c.init = init
35     c.is_a = function(self, klass)
36         local m = getmetatable(self)
37         while m do
38             if m == klass then return true end
39             m = m._base
40         end
41         return false
42     end
43     setmetatable(c, mt)

```

```

44     return c
45 end

```

A.2 La classe Neural_Network

Nel capitolo 2 si sono mostrati i vari snippet di codice man mano che si introducevano i concetti teorici che stanno alla base di questa implementazione. Di seguito è presentata l'intera classe Neural_Network :

```

1  --creating class NN in Lua, using a nice class utility
2  class = require 'class'
3
4  Neural_Network = class('Neural_Network')
5
6  --init NN
7  function Neural_Network:__init(inputs, hidden, outputs)
8      self.inputLayerSize = inputs
9      self.hiddenLayerSize = hidden
10     self.outputLayerSize = outputs
11     self.W1 = th.randn(net.inputLayerSize, self.hiddenLayerSize)
12     self.W2 = th.randn(net.hiddenLayerSize, self.outputLayerSize)
13 end
14
15 --define a forward method
16 function Neural_Network:forward(X)
17     --Propagate inputs through network
18     self.z2 = th.mm(X, self.W1)
19     self.a2 = th.sigmoid(self.z2)
20     self.z3 = th.mm(self.a2, self.W2)
21     yHat = th.sigmoid(self.z3)
22     return yHat
23 end
24
25 function Neural_Network:d_Sigmoid(z)
26     --derivative of the sigmoid function
27     return th.exp(-z):cdiv( (th.pow( (1+th.exp(-z)),2) ) )
28 end
29
30 function Neural_Network:costFunction(X, y)
31     --Compute the cost for given X,y, use weights already stored in class
32     self.yHat = self:forward(X)
33     --NB torch.sum() isn't equivalent to python sum() built-in method
34     --However, for 2D arrays whose one dimension is 1, it won't make any
35     --difference
36     J = 0.5 * th.sum(th.pow((y-yHat),2))
37     return J
38 end
39
40 function Neural_Network:d_CostFunction(X, y)
41     --Compute derivative wrt to W and W2 for a given X and y
42     self.yHat = self:forward(X)
43     delta3 = th.cmul(-(y-self.yHat), self:d_Sigmoid(self.z3))
44     dJdW2 = th.mm(self.a2:t(), delta3)
45
46     delta2 = th.mm(delta3, self.W2:t()):cmul(self:d_Sigmoid(self.z2))
47     dJdW1 = th.mm(X:t(), delta2)
48
49     return dJdW1, dJdW2
50 end

```

A.3 Metodi getter e setter

Nella sottosezione 2.4 del capitolo 2 si è dimostrato come calcolare numericamente il gradiente. Si è fatto cenno ai *getter e setter* per ottenere dei *flattened gradients*, ovvero "srotolare" i tensori dei gradienti in vettori monodimensionali. I metodi, qui mostrati, necessitano di una comprensione dei comandi di Torch. Data la maggiore popolarità di Python *Numpy*, nel caso il lettore fosse più familiare con quest'ultimo, nell'appendice B è mostrata anche una tabella di equivalenza dei metodi fra i due.

```

1  --Helper Functions for interacting with other classes:
2  function Neural_Network:getParams()
3      --Get W1 and W2 unrolled into a vector
4      params = th.cat((self.W1:view(self.W1:nElement()), (self.W2:view(
5          self.W2:nElement()))))
6      return params
7  end
8
9  function Neural_Network:setParams(params)
10     --Set W1 and W2 using single parameter vector.
11     W1_start = 1 --index starts at 1 in Lua
12     W1_end = self.hiddenLayerSize * self.inputLayerSize
13     self.W1 = th.reshape(params[{ {W1_start, W1_end} }],
14         self.inputLayerSize, self.hiddenLayerSize)
15     W2_end = W1_end + self.hiddenLayerSize*self.outputLayerSize
16     self.W2 = th.reshape(params[{ {W1_end+1, W2_end} }],
17         self.hiddenLayerSize, self.outputLayerSize)
18 end
19
20 --this is like the getParameters(): method in the NN module of torch, i.e.
21     compute the gradients and returns a flattened grads array
22 function Neural_Network:computeGradients(X, y)
23     dJdW1, dJdW2 = self:costFunctionPrime(X, y)
24     return th.cat((dJdW1:view(dJdW1:nElement()), (dJdW2:view(dJdW2:
25         nElement()))))
26 end

```


Appendice B

Il framework Torch

[19]

Bibliografia

- [1] Wikipedia. (2016). Connessionismo Wikipedia, L'enciclopedia libera, indirizzo: <http://it.wikipedia.org/w/index.php?title=Connessionismo&oldid=82972690> (visitato il 17/08/2017).
- [2] —, (2016). Funzione gradino di Heaviside, indirizzo: https://it.wikipedia.org/wiki/Funzione_gradino_di_Heaviside (visitato il 20/08/2017).
- [3] —, (2016). Percettrone, indirizzo: <http://it.wikipedia.org/w/index.php?title=Percettrone&oldid=88722444> (visitato il 20/08/2017).
- [4] —, (2016). Universal Approximation Theorem, indirizzo: https://en.wikipedia.org/w/index.php?title=Universal_approximation_theorem&oldid=796183757 (visitato il 20/08/2017).
- [5] (2016). Gradisca New York, indirizzo: www.gradiscanyc.com (visitato il 20/08/2017).
- [6] A. Central. (2016). The Average Profit Margin for a Restaurant, indirizzo: <http://yourbusiness.azcentral.com/average-profit-margin-restaurant-13113.html> (visitato il 20/08/2017).
- [7] R. R. Group. (2016). The Restaurant Financial Red Flags, indirizzo: http://rrgconsulting.com/ten_restaurant_financial_red_flags.htm (visitato il 20/08/2017).
- [8] Wikipedia. (2016). The Sigmoid Function, indirizzo: https://en.wikipedia.org/wiki/Sigmoid_function (visitato il 20/08/2017).
- [9] —, (2016). Supervised Learning, indirizzo: https://en.wikipedia.org/wiki/Supervised_learning (visitato il 20/08/2017).
- [10] StackExchange. (2016). List of cost function used in Neural Networks applications, indirizzo: <https://stats.stackexchange.com/questions/154879/a-list-of-cost-functions-used-in-neural-networks-alongside-applications> (visitato il 20/08/2017).
- [11] Wikipedia. (2016). Regola della catena, indirizzo: https://it.wikipedia.org/wiki/Regola_della_catena (visitato il 20/08/2017).
- [12] CS231n-Stanford. (2016). Vector, Matrix, and Tensor Derivatives, indirizzo: <http://cs231n.stanford.edu/vecDerivs.pdf> (visitato il 23/08/2017).
- [13] Stanford. (2016). Gradient checking and advanced optimization, indirizzo: http://ufldl.stanford.edu/wiki/index.php/Gradient_checking_and_advanced_optimization (visitato il 20/08/2017).
- [14] Wikipedia. (2016). Stochastic Gradient Descent, indirizzo: https://en.wikipedia.org/wiki/Stochastic_gradient_descent (visitato il 20/08/2017).
- [15] —, (2016). The Quasi-Newton method, indirizzo: https://en.wikipedia.org/wiki/Quasi-Newton_method (visitato il 20/08/2017).

- [16] —, (2016). Limited-Memory BFGS, indirizzo: https://en.wikipedia.org/wiki/Limited-memory_BFGS (visitato il 20/08/2017).
- [17] M. Mastery. (2017). A gentle introduction to Adam, indirizzo: <https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/> (visitato il 25/08/2017).
- [18] CS231n-Stanford. (2017). CNN for Visual Recognition, indirizzo: <http://cs231n.github.io/neural-networks-3/> (visitato il 25/08/2017).
- [19] (2017). Torch: a scientific computing framework for LuaJit, indirizzo: torch.ch (visitato il 20/08/2017).