Università di Bologna

Attività progettuale di Sistemi Intelligenti M

Reti neurali artificiali: dal MLP agli ultimi modelli di CNN

Autore:
Ali Alessio Salman

23 agosto 2017

Indice

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).

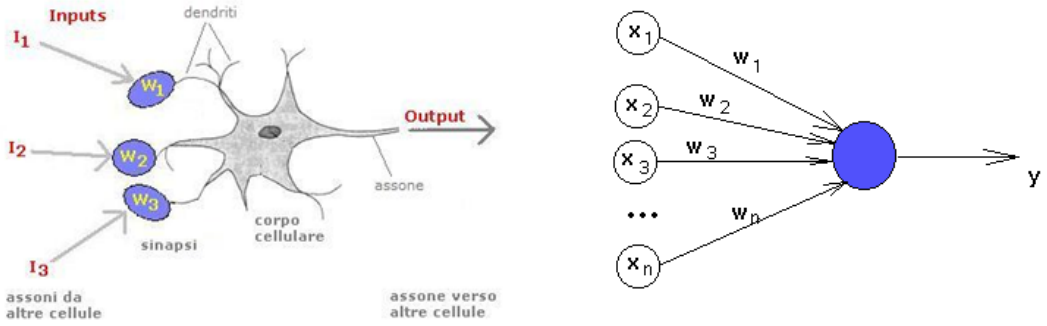

Figura 1.1: Modello di calcolo di un neurone (a sinistra) e schema del neurone artificiale (a destra)

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 funziona di trasferimento non lineare:

$$f(x) = H(\sum_i (w_i x_i)) \tag{1.1}$$

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].

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

$$w_i(t+1) = w_i(t) + \eta \delta x_i(t) \tag{1.3}$$

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.

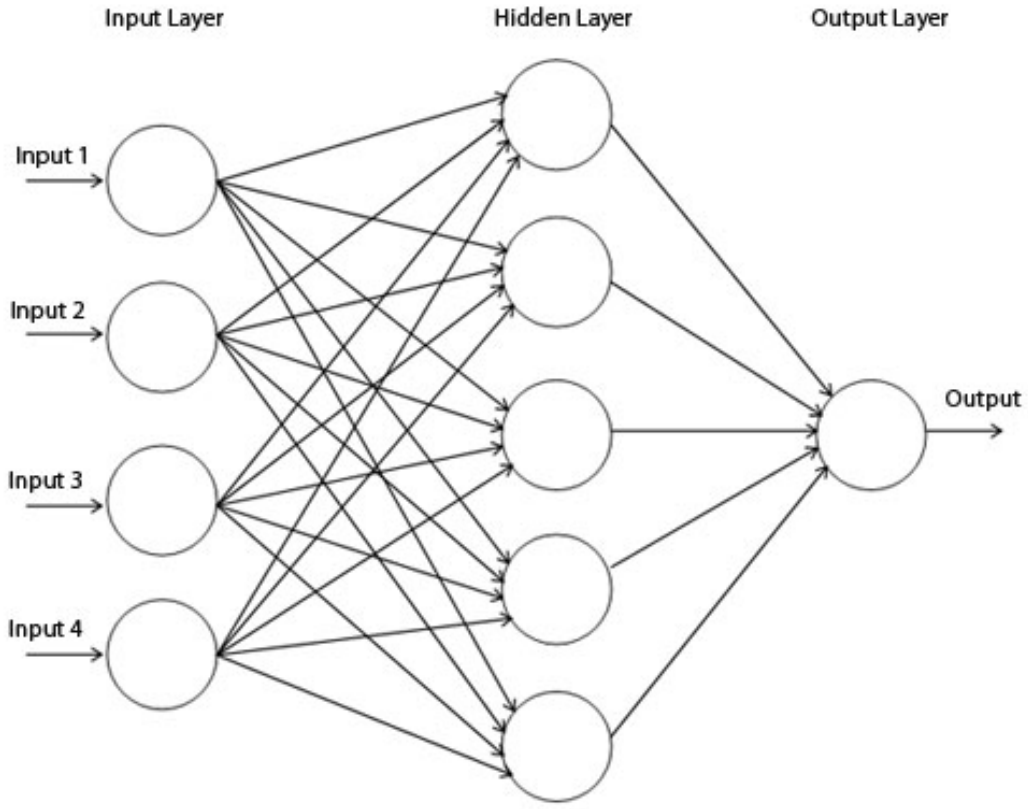

Figura 1.2: Struttura di un percettrone multistrato con un solo strato nascosto

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 finiti 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 $\leqslant$ 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 ristorantino 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.

1.4 Implementazione da zero di un MLP

Per il suddetto caso di studio 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** e mantenere le consistenza con gli altri capitoli, nei quali si userà ancora Torch per addestrare reti neurali molto più complesse.

1.4.1 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:

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

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 1.3 è mostrata l'architettura generale della nostra rete.

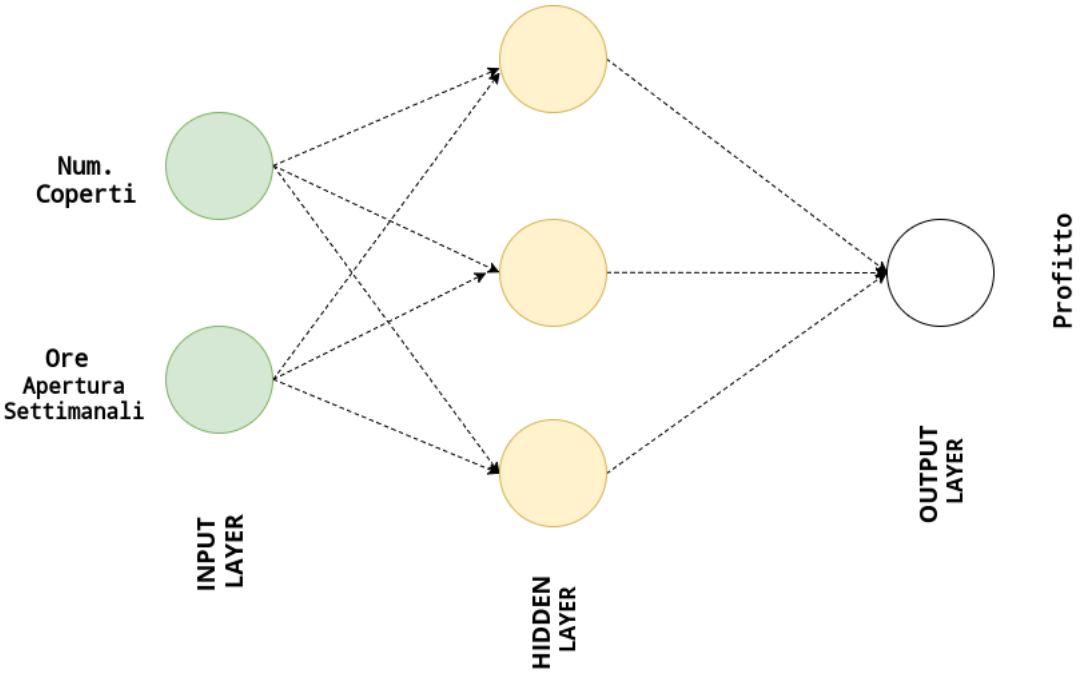

Figura 1.3: Architettura del percettrone multistrato per la previsione dei profitti

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

```
---------------------- Part 1 ----------------------------
th = require 'torch'
bestProfit = 6.0
-- X = (num coperti, ore di apertura settimanali), y = profitto lordo
    annuo in percentuale
torch.setdefaulttensortype('torch.DoubleTensor')
X = th.Tensor({{22,42}, {25,38}, {30,40}})
y = th.Tensor({{2.8},{3.4},{4.4}})

--normalize
normalizeTensorAlongCols(X)
y = y/bestProfit
```

```
---------------------- Part 2 ----------------------------
--creating the NN class in Lua, using a nice class utility
class=require 'class'
local Neural_Network = class('Neural_Network')

function Neural_Network:__init(inputs, hiddens, outputs)
      self.inputLayerSize = inputs
      self.hiddenLayerSize = hiddens
      self.outputLayerSize = outputs
      self.W1 = th.randn(net.inputLayerSize, self.hiddenLayerSize)
      self.W2 = th.randn(net.hiddenLayerSize, self.outputLayerSize)
end
```

Tabella 1.1: 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)

1.4.2 Forward Propagation

Nella tabella 1.1 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 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:

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

Si noti che $z^{(2)}$ è di dimensione 3x3, essendo X e $W^{(1)}$ di dimensione 3x2 e 2x3 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.

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

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:

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

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

Essendo $a^{(2)}$ di dimensione 3x3 e $W^{(2)}$ 3x1 l'output $\hat{y}$ sarà anch'esso di dimensione 3x1, 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.

```
--Note: I didn't implement manually the sigmoid function as Torch has one
    built-in.
--define a forward method
function Neural_Network:forward(X)
   --Propagate inputs though network
   self.z2 = th.mm(X, self.W1) --matrix multiplication
   self.a2 = th.sigmoid(self.z2)
   self.z3 = th.mm(self.a2, self.W2)
   yHat = th.sigmoid(self.z3)
   return yHat
end
```

1.4.3 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*, scoperto 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]), 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 diminuire questa funzione. 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.

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

Da cui, il codice in Lua:

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

La *backprop* ha alcuni requisiti:

- Reti stratificate
- Ingressi a valori reali $\in [0, 1]$
- Neuroni non lineari con funzione di uscita sigmoidale (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* (figura1.4); ovvero variano in maniera tale da minimizzare la funzione di costo J.

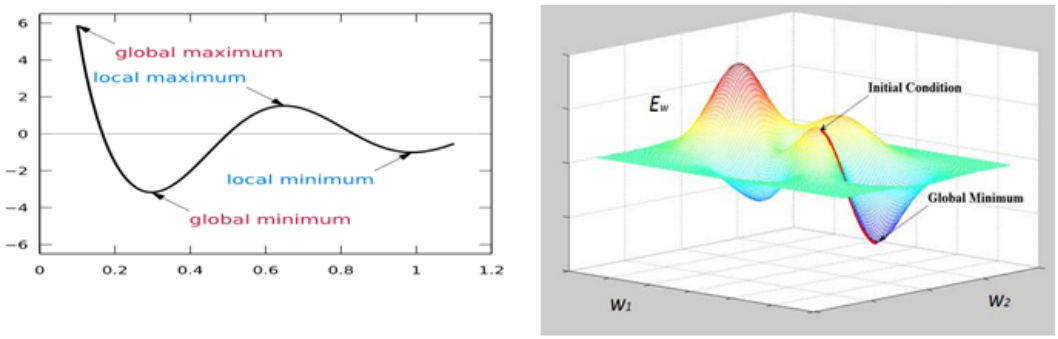

Figura 1.4: Cercare il minimo di una fz. seguendo la discesa del gradiente

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 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:

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

Definiamola, quindi, nel codice:

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

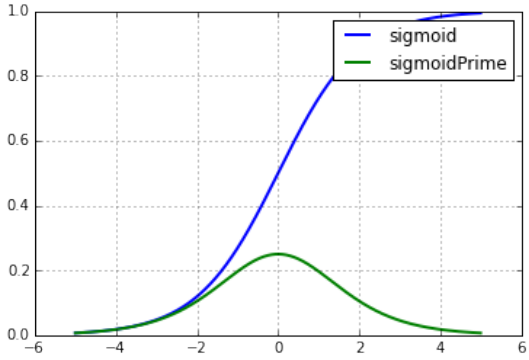

Figura 1.5: La funzione sigmoide e la sua derivata

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 a2 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:

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

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

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

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

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

```
function Neural_Network:d_CostFunction(X, y)
   --Compute derivative wrt to W and W2 for a given X and y
   self.yHat = self:forward(X)
   delta3 = th.cmul(-(y-self.yHat), self:d_Sigmoid(self.z3))
   dJdW2 = th.mm(self.a2:t(), delta3)

   delta2 = th.mm(delta3, self.W2:t()):cmul(self:d_Sigmoid(self.z2))
   dJdW1 = th.mm(X:t(), delta2)

   return dJdW1, dJdW2
end
```

1.4.4 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[**WGradCheck**] è basato sulla seguente definizione di derivata:

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

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:

```
function computeNumericalGradient(NN, X, y)
   paramsInitial = NN:getParams()
   numgrad = th.zeros(paramsInitial:size())
   perturb = th.zeros(paramsInitial:size())
   e = 1e-4

   for p=1,paramsInitial:nElement() do
      --Set perturbation vector
      perturb[p] = e
      NN:setParams(paramsInitial + perturb)
      loss2 = NN:costFunction(X, y)

      NN:setParams(paramsInitial - perturb)
      loss1 = NN:costFunction(X, y)

      --Compute Numerical Gradient
      numgrad[p] = (loss2 - loss1) / (2*e)

      --Return the value we changed to zero:
      perturb[p] = 0
   end

   --Return Params to original value:
   NN:setParams(paramsInitial)
   return numgrad
end
```

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

```
--test if we actually make the calculations correctly
NN = Neural_Network(2,3,1)

print('Gradient checking...')
numgrad = computeNumericalGradient(NN, X, y)
grad = NN:computeGradients(X, y)
--[[
In order to make an accurate comparison of the 2 vectors
we can calculate the difference as the ratio of:
numerator  --> the norm of the difference
denumerator--> the norm of the sum
Should be in the order of 10^-8 or less
--]]
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 norme 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:

1.4.5 Addestramento

1.5 Ottimizzazione: diverse tecniche

1.6 Overfitting: come rilevarlo e risolverlo

1.7 Risultati

Capitolo 2

Reti Neurali Convoluzionali

2.1 Breve introduzione

La reti neurali convoluzionali, alle quali ci riferiremo con l'abbrevazione *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.

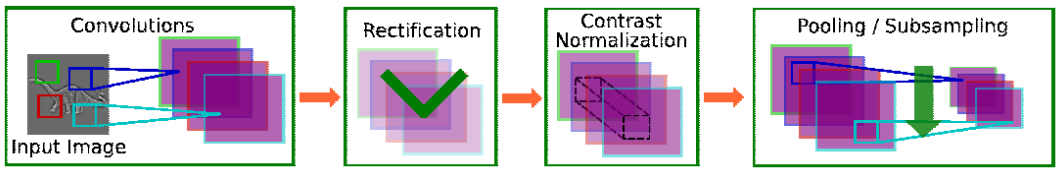

Figura 2.1: I diversi strati tipici di una CNN

2.2 Architettura

2.2.1 Strato di Convoluzione

2.2.2 Strato di Pooling

2.2.3 Strato completamente connesso (FC)

2.3 Applicazioni e risultati

Capitolo 3

CNN: implementazione e addestramento

3.1 Il framework Torch

3.2 Modello di CNN basato su LeNet

3.2.1 Strato di Convoluzione

3.2.2 Strato di Pooling

3.2.3 Strato completamente connesso (FC)

3.3 Dataset d'esempio: CIFAR

3.4 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 4

Addestrare un modello allo stato dell'arte

Capitolo 5

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.

5.1 Dataset

5.2 Fine-Tuning

5.3 ModelZoo

5.3.1 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 6

Conclusioni

Appendice A

Appendice A: Codice addizionale ed il framework Torch

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

```
-- class.lua
-- Compatible with Lua 5.1 (not 5.0).
function class(base, init)
   local c = {} -- a new class instance
   if not init and type(base) == 'function' then
      init = base
      base = nil
   elseif type(base) == 'table' then
      -- our new class is a shallow copy of the base class!
      for i,v in pairs(base) do
         c[i] = v
      end
      c._base = base
   end
   -- the class will be the metatable for all its objects,
   -- and they will look up their methods in it.
   c.__index = c

   -- expose a constructor which can be called by <classname>(<args>)
   local mt = {}
   mt.__call = function(class_tbl, ...)
      local obj = {}
      setmetatable(obj,c)
      if init then
         init(obj,...)
      else
         -- make sure that any stuff from the base class is initialized!
         if base and base.init then
            base.init(obj, ...)
         end
      end
      return obj
   end
   c.init = init
   c.is_a = function(self, klass)
      local m = getmetatable(self)
      while m do
         if m == klass then return true end
         m = m._base
      end
      return false
   end
   setmetatable(c, mt)
   return c
```

```
45 end
```

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 Netoworks 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] CS231-Stanford. (2016). Vector, Matrix, and Tensor Derivatives, indirizzo: http://cs231n.stanford.edu/vecDerivs.pdf (visitato il 23/08/2017).