Università di Bologna

Attività progettuale di Sistemi Intelligenti M

Reti neurali artificiali: dal MLP agli ultimi modelli di CNN

Autore:
Ali Alessio Salman

30 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[**WConnessionismo**] 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).

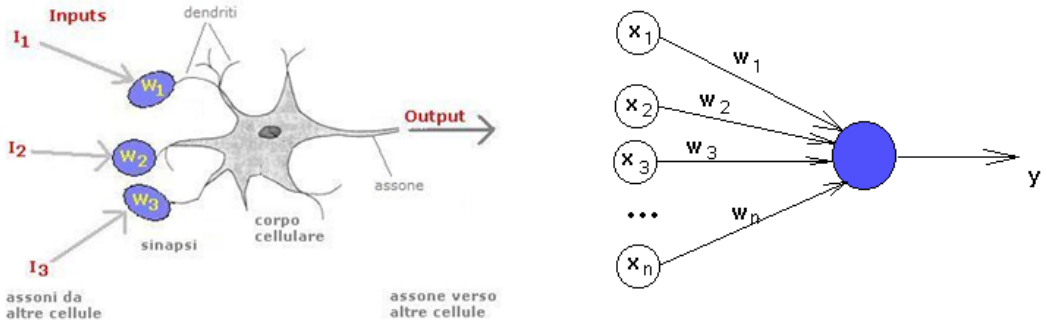


Figura 1.1: Modello di calcolo di un neurone (a sinistra) e schema del neurone artificiale (a destra)

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 [**WHeaviside**]. 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*[**WPercettrone**].

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

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

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.

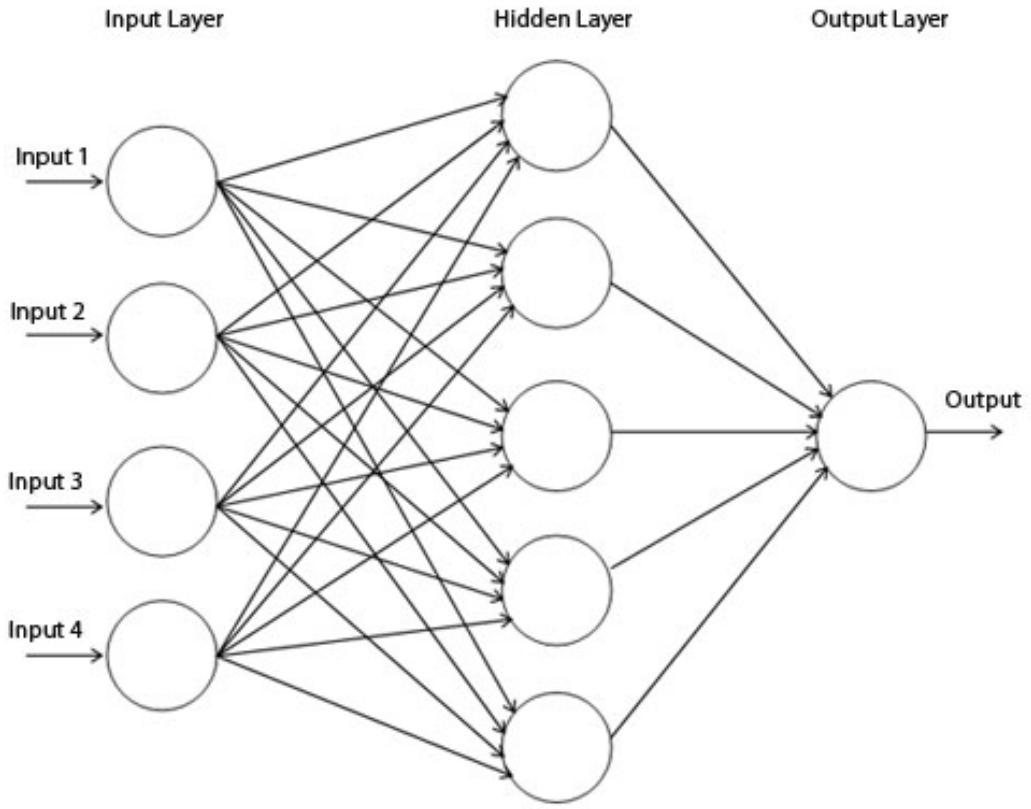


Figura 1.2: Struttura di un percettrone multistrato con un solo strato nascosto

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[**WApprox**], **WApprox** [**WConnessionismo**] 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 [**WGradisca**] 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[**WProfit**] [**WRRG**] 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 l'appendiceB) 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:

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

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.

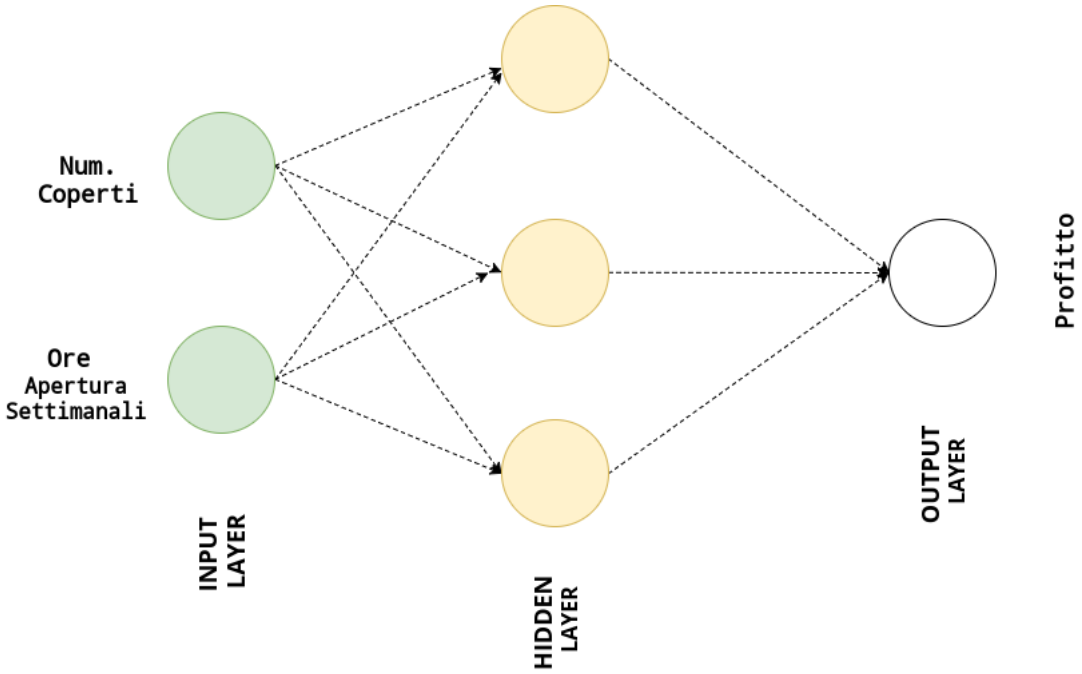


Figura 2.1: Architettura del MLP per la previsione dei profitti

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

```
----------------------- 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
```

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:

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

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*[**WSigmoid**], 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
```

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[**WSupervised**], 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 [**WLoss**] 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:

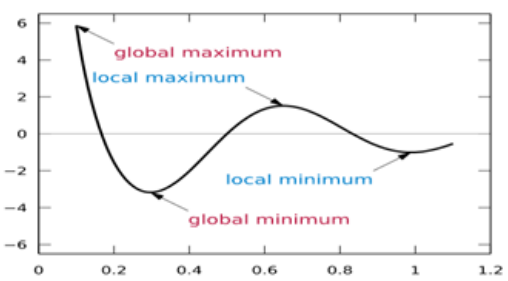
```
function Neural_Network:costFunction(X, y)
   --Compute the cost for given X,y, use weights already stored in class
   self.yHat = self:forward(X)
   J = 0.5 * th.sum(th.pow((y-yHat),2))
   return J
end
```

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[**WChain**] 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



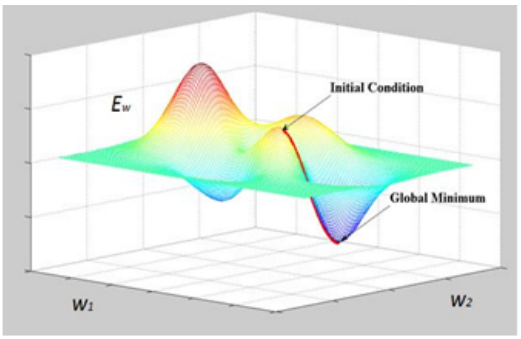


Figura 2.2: Cercare il minimo di una fz. seguendo la discesa del gradiente

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:

```
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

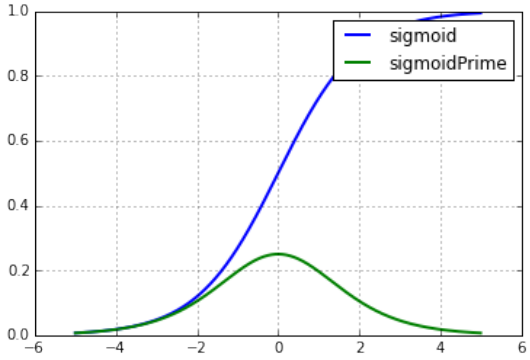


Figura 2.3: La funzione sigmoide e la sua derivata

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 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[**WCS231vec**]: 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
```

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[**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)
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:

```
$ th 4_gradCheck.lua
Gradient checking...
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 *<n. coperti, n. ore settimanali>* 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 ogni ciclo. Questo metodo è conosciuto come *Stochastig Gradient*

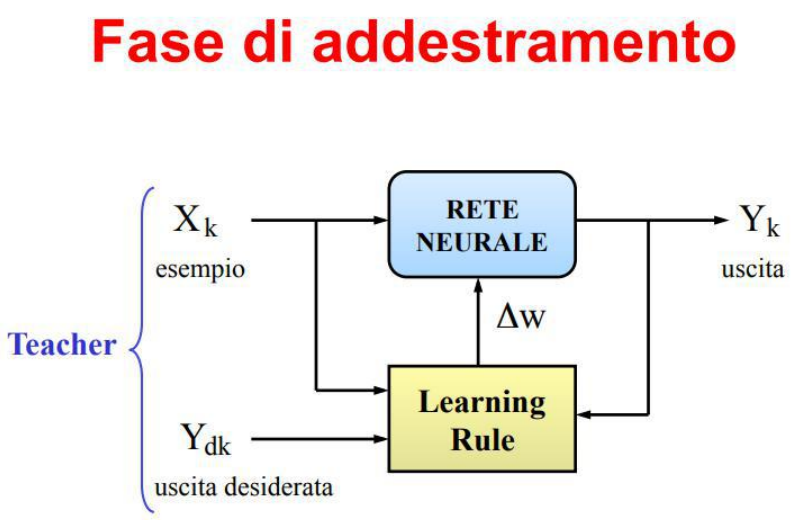


Figura 2.4: Apprendimento supervisionato: schema generale

Descent[**WSGD**]. 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"*:

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

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 azzera, 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 utlizzati 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, memorizzando nell'equazione la variazione ΔW precedente:

$$W_t = W_{t-1} - \eta \nabla J(W_t) - \underbrace{\mu \nabla J(W_{t-1})}$$

  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 e acquisice velocità nella discesa.
  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.

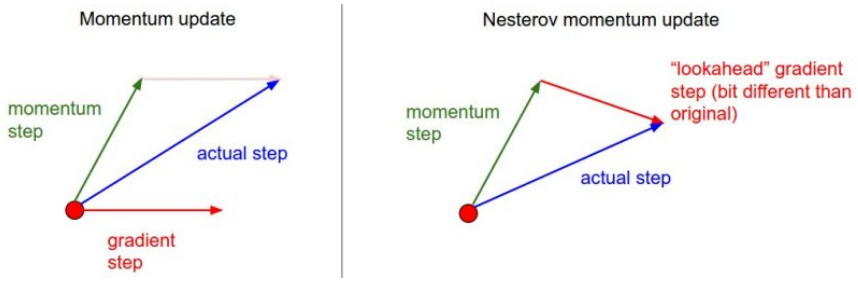


Figura 2.5: Confronto tra il momentum classico e NAG

- BFGS: l'algoritmo *Broyden-Fletcher-Goldfarb-Shanno* fa parte della famiglia "Quasi-Newton" [**Wquasinewton**]. 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[**WLBFGS**].

- 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[**WAdam**].

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.

```
Trainer = class('Trainer')
function Trainer:__init(NN)
  --Make Local reference to network:
  self.N = NN
end

--Let's train!
function Trainer:train(X, y)
   --variables to keep track of the training
   local neval = 0
   --get initial params
   params0 = self.N:getParams()
   -- create closure to evaluate f(X) and df/dX
   -- this is requested by the API of the optim package
   local feval = function(params0)
      local f = self.N:costFunction(X, y)
      print(f)
      local df_dx = self.N:computeGradients(X, y)
      neval = neval + 1
      logger:add{neval, f} --,timer:time().real}
      return f, df_dx
   end
   if optimMethod == optim.cg then
      newparams,_,_ = optimMethod(feval, params0, optimState)
   else
      for i=1,opt.maxIter do
         newparams,_,_ = optimMethod(feval, params0, optimState)
         self.N:setParams(newparams)
      end
   end
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[**WCS231adam**], sviluppato da Andrej Karpathy et. al, si raccomanda Adam come metodo di default per applicazioni di deep learning:

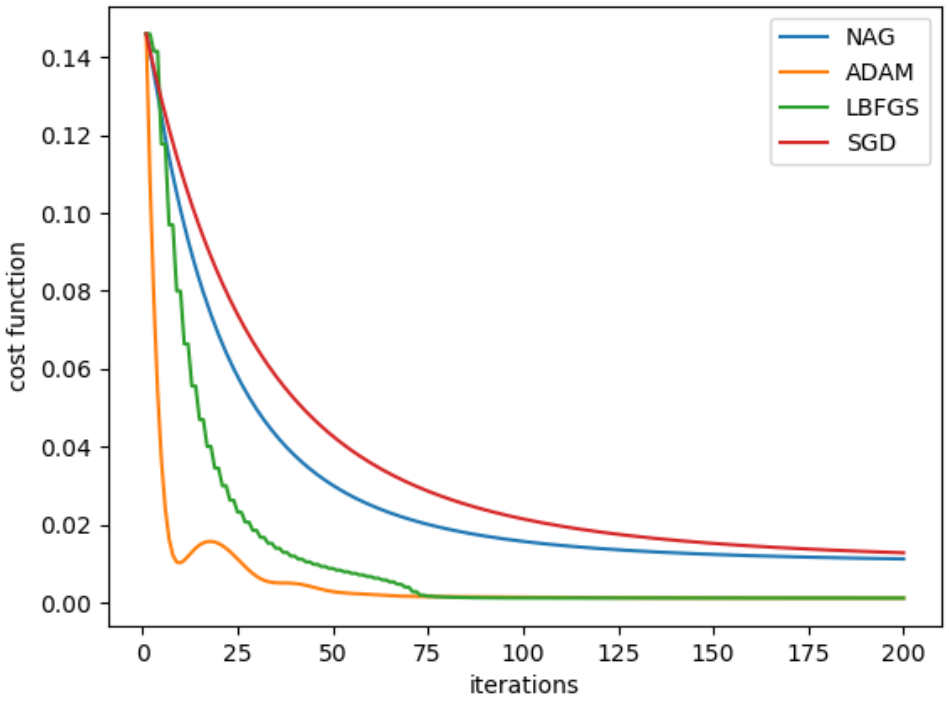


Figura 2.6: Confronto dei metodi di ottimizzazione durante il training

> *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, di cui i più classici sono:

- 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.

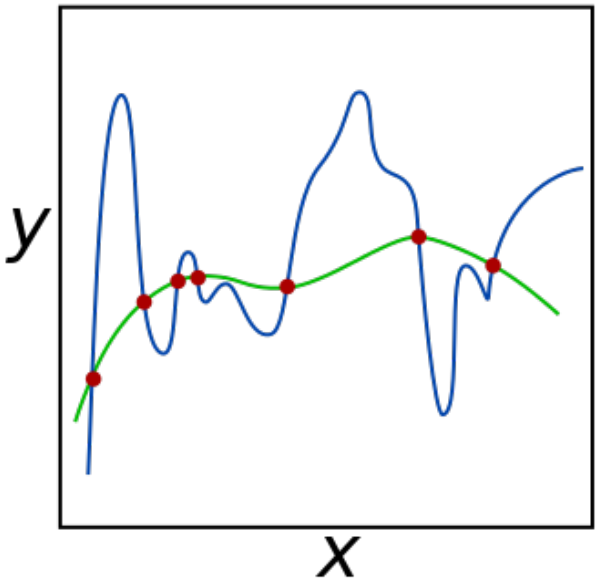


Figura 2.8: Entrambe le funzioni hanno un'errore nullo sul dataset. Tuttavia, la funzione blu è eccessivamente complessa ed affetta da rumore; avrà quindi cattive capacità di generalizzazione. La verde, al contrario, sarà presumibilmente più corretta rispetto ai punti sconosciuti della distribuzione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

2.7.1 Rilevare l'overfitting

Per rilevare se è avvenuto o meno overfitting durante l'apprendimento; si possono dare in ingresso al percettrone 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:

```
--Need to modify trainer class a bit to check testing error during
    training:
function Trainer:train(trainX, trainY, testX, testY)
   --variables to keep track of the training
   local neval = 0

   params0 = self.N:getParams()
   -- create closure to evaluate f(X) and df/dX
   local feval = function(params0)
      local f = self.N:costFunction(trainX, trainY)
      local test = self.N:costFunction(testX, testY)
      --printing training and testing error
      print(f..' '..test)
      local df_dx = self.N:computeGradients(trainX, trainY)
      neval = neval + 1
      --logging both training and testing data
      logger:add{neval, f} --,timer:time().real}
      testLogger:add{neval, test}

      return f, df_dx
   end

   if optimMethod == optim.cg then
      newparams,_,_ = optimMethod(feval, params0, optimState)
   else
      for i=1,opt.maxIter do
         newparams,_,_ = optimMethod(feval, params0, optimState)
         self.N:setParams(newparams)
      end
   end
end
```

Si addestra la rete e si loggano i valori di training e di testing:

```
--now let's train and check where exactly the net is Overfitting
nn = Neural_Network(2,3,1)

init_params = nn:getParams()
logtrain = 'train.log'
logtest = 'test.log'
logger = optim.Logger(logtrain)
testLogger = optim.Logger(logtest)

trainer = Trainer(nn)
trainer:train(trainX, trainY, testX, testY)
```

Andando infine a plottare i dati così ottenuti, si osserva la presenza di overfitting:

2.7.2 Contromisure

Vi sono diverse contromisure contro l'overfitting, che seguono le cause più comuni:

- dataset più ampio: una regola empirica consiste nell'avere circa 10 esempi per ogni parametro della rete. Nel caso del MLP di questo progetto i pesi sono 9, quindi secondo questa regola sono necessari 90 esempi. È ovvimente impossibile, poiché Loris ha collezionato solamente un limitato numero di esempi.
- *k-fold cross-validation*: si divide il dataset in K set di eguale misura, ad ogni iterazione uno dei set viene utilizzato come test-set ed il resto viene per formare

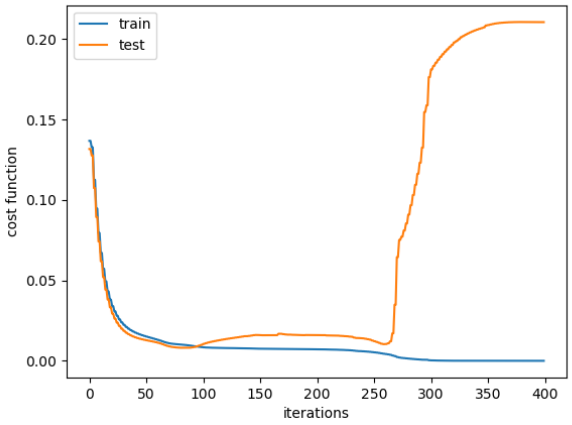


Figura 2.9: La rete mostra overfitting. Dopo l'iterazione 100 le performance sul test set iniziano ad essere sbagliate. Attorno alla 250 il modello diventa troppo complesso e le predizioni sono pessime

il training e validation set (si veda lez.1 del corso di Machine Learning per PhD di M. LippiWLippi). Anche questa procedura non è applicabile in questo caso.

- *Dropout*: il dropout è una tecnica di regolarizzazione maggiormente utilizzata nelle reti profonde, nella quale ad ogni iterazione si "spengono" alcuni neuroni (settando i pesi che li collegano a 0). Questo fa sì che neuroni vicini non si specializzino tutti a riconoscere le stesse caratteristiche, risultando in un minore overfitting. Una spiegazione approfondita esula dallo scopo di questo elaborato, per cui si suggerisce la consultazione del paper di Srivastava, Hinton et. al[**Dropout**].

- arresto anticipato: una volta rilevato quando avviene precisamente l'overfitting, si arresta l'apprendimento *prima* di quel momento. Osservando la figura 2.9 si potrebbe pensare di fermare le iterazioni alla n.100 (o comunque prima della n. 250) e funzionerebbe. Così facendo, si ha una maniera piuttosto semplice di risolvere il problema. C'è uno svantaggio però: se la rete non è ancora ben addestrata non c'è possibilità di migliorarla ulteriormente, poiché ogni nuova iterazione causerebbe overfitting. Per questo, è stato usato l'ultimo metodo qui presentato.

- Regolarizzazione: consiste nell'aggiungere un termine alla funzione di costo J che penalizza modelli troppo complessi. [**WLippi**]. Questo termine, $\lambda R(W)$ è regolato da un parametro λ che definisce l'intensità della regolarizzazione.

Un comune metodo di regolarizzazione è la *L2 Regularization*: si implementa aggiungendo alla funzione di costo il quadrato dei tensori dei pesi della rete. In questa maniera si tengono i valori di tutti i pesi piuttosto bassi evitando che si specializzino troppo sui dati del training set. Di seguito il codice dei metodi della rete che devono essere modificati:

```
1 --[[
2 ## Introducing a Regularization term to mitigate overfitting ##
3 Lambda will allow us to tune the relative cost:
4 higher values of Lambda --> bigger penalties for high model complexity
5 --]]
6
```

```
--so, the new Neural_Network class now becomes:
Neural_Network = class(function(net, inputs, hiddens, outputs, lambda)
      net.inputLayerSize = inputs
      net.hiddenLayerSize = hiddens
      net.outputLayerSize = outputs
      net.W1 = th.randn(net.inputLayerSize, net.hiddenLayerSize)
      net.W2 = th.randn(net.hiddenLayerSize, net.outputLayerSize)

      --regularization parameter
      net.lambda = lambda
   end)

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))/X:size()[1] +
            (self.lambda/2) * (th.sum(th.pow(self.W1,2)) + th.sum(th.pow(
    self.W2, 2)))

   return J
end

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))
   --Add gradient of regularization term:
   dJdW2 = th.mm(self.a2:t(), delta3)/X:size()[1] + self.lambda*self.W2

   delta2 = th.mm(delta3, self.W2:t()):cmul(self:d_Sigmoid(self.z2))
   --Add gradient of regularization term:
   dJdW1 = th.mm(X:t(), delta2)/X:size()[1] + self.lambda*self.W1

   return dJdW1, dJdW2
end
```

Eseguendo ora l'addestramento si ottengono risultati in figura 2.10. Si può osservare come la regolarizzazione sia stata efficace e le 2 curve siano pressoché identiche. Si noti anche che, data la modesta complessità del problema, dopo circa la 130esima iterazione si è già raggiunto un minimo ed il processo di apprendimento non migliora ulteriormente.

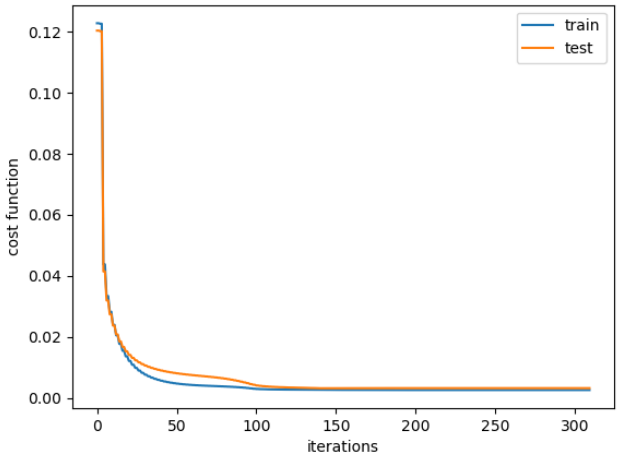


Figura 2.10: Dopo aver applicato la L2 regularization, la rete non è più affetta da overfitting.

2.8 Risultati

In questo capitolo si è visto come costruire da zero l'architettura di un percettrone multi-strato con paradigma OOP; come implementare la backpropagation e controllare numericamente che funzioni in maniera corretta. Successivamente si sono visti i principali algoritmi per ottimizzare l'apprendimento supervisionato di una rete neurale. Si è poi affrontato il problema dell'overfitting: come rilevarlo in maniera precisa ed un introduzione alle tecniche più comuni per risolverlo.
Infine, si è ottenuto una rete neurale capace di predire in maniera corretta il profitto di un ristorante in base al numero di coperti ed il numero di ore di apertura settimanali.

Capitolo 3

Reti Neurali Convoluzionali

In questo capitolo si introduce una panoramica generale sulle reti neurali convoluzionali. Essendo un argomento vasto, una trattazione teorica approfondita sarebbe materia di una tesi di laurea, ragion per cui gli argomenti sono introdotti con lo scopo di avere un'infarinatura per comprendere le applicazioni sviluppate nei capitoli successivi.

3.1 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 profonde caratterizzate da una particolare architettura estremamente vantaggiosa per compiti visivi (e non), che le ha rese negli anni molto efficaci e popolari. Sono state ispirate dalle ricerche biologiche di Hubel e Wiesel i quali, studiando il cervello dei gatti, avevano scoperto che la loro corteccia visiva conteneva una complessa struttura di cellule. Quest'ultime erano sensibili a piccole parti locali del campo visivo, detti campi recettivi *(receptive fields)*. Agivano quindi da filtri locali perfetti per comprendere la correlazione locale degli oggetti in un'immagine. Essendo questi sistemi i più efficienti in natura per la comprensione delle immagini, i ricercatori hanno tentato di simularli.

3.2 Architettura

Le CNN sono reti neurali profonde costituite da diversi strati che fungono da estrattori delle features ed una rete completamente connessa alla fine, che funge da classificatore, come raffigurato in figura 3.1.

Questi strati in cui si estraggono le caratteristiche delle immagini sono detti strati di convoluzione, e sono generalmente seguiti da una funzione non lineare e un passo di *pooling*. Vi possono poi essere degli strati di elaborazione dell'immagine, come quello di normalizzazione del contrasto, si veda ref:cnn2. Convoluzione e pooling hanno come scopo quello di estrarre le caratteristiche, mentre l'unità non lineare serve a rafforzare le caratteristiche più forti e indebolire quelle meno importanti, ovvero quelle che hanno stimolato meno i neuroni (si dice che fa da "squashing").
Sempre dalla figura 3.1, possiamo inoltre notare che, per ogni immagine in input, corrispondono nei vari strati, diversi gruppi di immagini, che vengono chiamate *feature maps*. Le feature maps sono il risultato dell'operazione di convoluzione svolta tramite un banco di filtri, chiamati anche kernel, che altro non sono che delle matrici con dei valori utili a ricercare determinate caratteristiche nelle immagini.
Infine, terminati i convolutional layers, le feature maps vengono "srotolate" in vettori e affidate ad una rete neurale "classica" che esegue la classificazione finale.

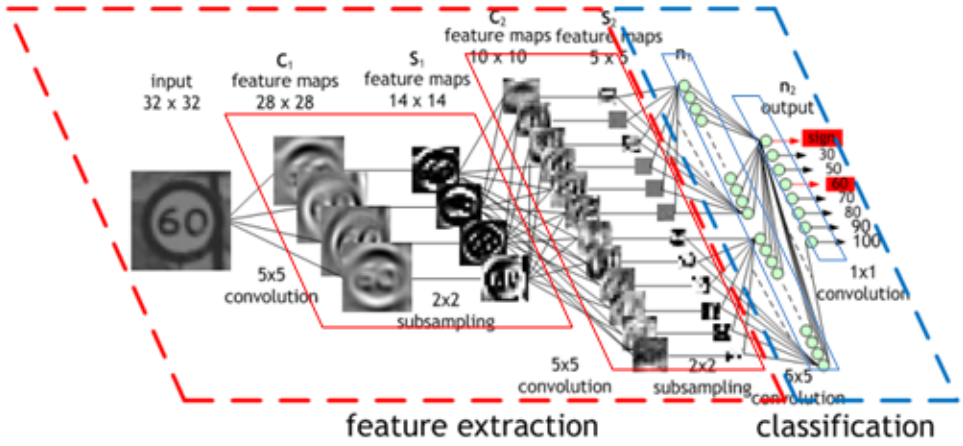


Figura 3.1: Architettura di una CNN che classifica segnali stradali: si evidenzia la divisione tra gli strati che fungono da feature extractor ed il classificatore finale

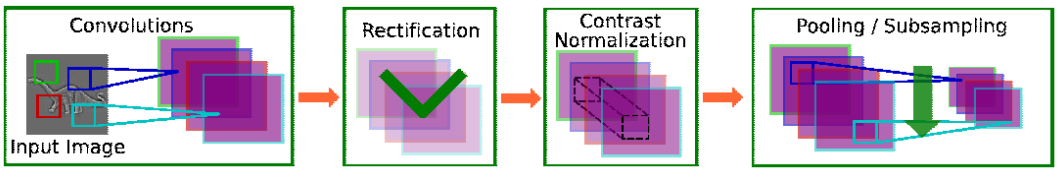


Figura 3.2: I diversi strati tipici di una CNN

Il numero di strati di convoluzione è arbitrario. Inizialmente, quando le CNN divennero famose grazie a Y.LeCun, che addestrò una CNN chiamata *"LeNet5"* al riconoscimento dei numeri [**lenet**], questo numero era compreso tra 2-5. Nel 2012, Alex Krizhevsky et al [**imagenet2012**] addestrarono una rete costituita da 5 strati di convoluzione, 60 milioni di parametri e 650 mila neuroni. Ottennero la migliore percentuale d'errore al mondo sul dataset ImageNet ILSVRC-2010, contenente 1,2 milioni di immagini divise in 1000 categorie.
Da allora le cose si sono evolute con una velocità disaramente, e l'ImageNet challenge del 2015, è stata vinta da una rete con 152 strati [**resnet**]. Nel capitolo 5 si farà un confronto tra quest'ultima rete, soprannominata *"ResNet"* e la capostipite LeNet5 su un task di classificazione.

3.2.1 Strato di Convoluzione

Per comprendere appieno quello che avviene in una CNN, occorre introdurre il concetto di convoluzione fra matrici, e capire come questo sia importante per applicare dei filtri ad un'immagine digitale.
Un'immagine digitale può essere considerata come una matrice A di dimensione MxN valori reali o discreti. Ogni valore della matrice prende il nome di pixel e i suoi indici sono anche chiamati coordinate: ogni pixel $A(m, n)$ rappresenta l'intensità nella posizione indicata dagli indici.
Si definisce "filtro" o "kernel" una trasformazione applicata ad un'immagine. Come detto prima, questi filtri sono a loro volta della matrici; la trasformazione quindi si effettua appunto tramite un'operazione di convoluzione tra l'immagine in ingresso ed

il filtro. La convoluzione, discreta nel caso di immagini digitali, si può definire come:

$$y[m,n] = x[m,n] \circledast h[m,n] = \sum^{m} \sum^{n} x[i,j] \times h[m-i, n-j]$$

Ogni pixel di $y[m,n]$ è così il risultato di una somma pesata tramite $h[m,n]$ della sottoregione che ha centro nel pixel indicato dalle coordinate m,n. Un esempio di convoluzione è rappresentato in figura 3.3.

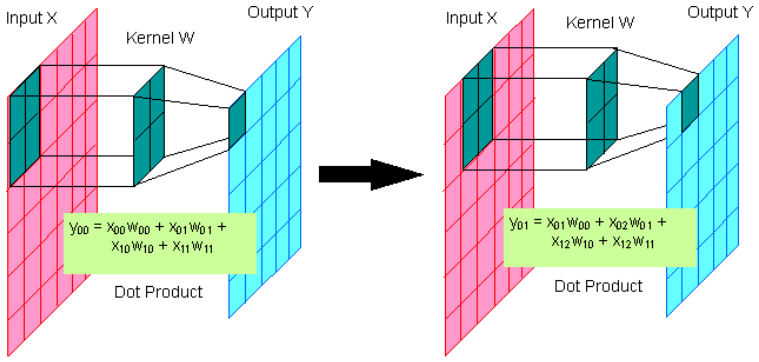


Figura 3.3: Convoluzione con un kernel: primi due step

Nei convolutional layers viene quindi fatta un'operazione di convoluzione tra l'immagine/i in ingresso e un numero arbitrario K di filtri. Questi filtri hanno valori tali da ottenere in uscita un riconoscimento di determinate caratteristiche.
I valori dei filtri sono all'inizio scelti casualmente, e vengono poi migliorati ad ogni iterazione mediante l'algoritmo di backpropagation, visto nel Capitolo 2. Così facendo la rete addestra i suoi filtri ad estrarre le features più importanti degli esempi del training set; cambiando training set i valori dei filtri saranno diversi. Ad esempio, i valori dei filtri di una rete allenata con immagini di pali verticali saranno diversi da quella allenata con immagini di palloni da calcio; nel primo caso i valori saranno calibrati per riconoscere lunghi orientamenti verticali, mentre nel secondo per riconoscere i confini sferici.

Nelle reti convoluzionali quindi, l'algoritmo di Backpropagation migliora i valori dei filtri della rete, è lì quindi che si accumula l'apprendimento. I neuroni, in queste reti, devono intendersi come i singoli filtri.

Vi sono diversi *hyperparameters* da settare manualmente negli strati di convoluzione:

1. la misura del filtro F: chiamato anche *receptive field*. Ogni filtro cerca una determinata caratteristica in un'area locale dell'immagine, la sua misura quindi è il campo recettivo del singolo neurone. Tipicamente sono 3x3, 5x5 o 7x7.

2. il numero K di filtri: per ogni strato, questo valore definisce la profondità dell'output dell'operazione di convoluzione. Infatti, mettendo una sopra l'altra le feature maps, si ottiene un cubo in cui ogni "fetta" è il risultato dell'operazione tra l'immagine in ingresso ed il corrispettivo filtro. La profondità di questo cubo dipende appunto dal numero dei filtri.

3. lo *"Stride"* S: definisce di quanti pixel si muove il filtro della convoluzione ad ogni passo. Se lo stride è settato a 2, il filtro salterà 2 pixel alla volta, producendo quindi un output più piccolo.

4. il *"padding"* P: definisce la misura con la quale si vuole aggiungere degli "0" all'input per preservare la dimensione in output. In generale, quando lo stride S=1, un valore di $P = (F-1)/2$ garantisce che l'output avrà le stesse dimensioni dell'input.

Quando si elaborano delle immagini con le CNN si hanno generalmente in ingresso degli input tridimensionali, caratterizzati dall'altezza H_1, l'ampiezza W_1 e il numero di canali di colore D_1. Conoscendo i parametri sopra specificati si può calcolare la dimensione dell'output di un layer di convoluzione:

$$H_2 = (H_1 - F + 2P)/S + 1$$
$$W_2 = (H_1 - F + 2P)/S + 1$$
$$D_2 = K$$

A questo proposito si osservi un esempio di "volume di neuroni" del primo strato di convoluzione in figura 3.4. Ogni neurone è collegato spazialmente solo ad 1 regione locale dell'input ma per tutta la profondità (i.e. i 3 canali del colore). Si noti che ci sono 5 neuroni lungo la profondità e tutti guardano alla stessa regione dell'input.

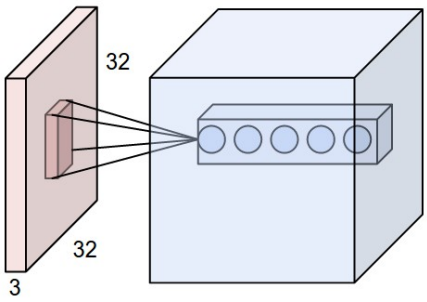


Figura 3.4: Ogni neurone è connesso a solo 1 regione locale dell'input ma a tutta la profondità (i.e. canali colori). La depth dell'output è data dal numero K di filtri, in questo caso 5

Gli strati di convoluzione mostrano molte proprietà interessanti. In primo luogo, se l'immagine in input viene traslata, l'output della feature map sarà traslato della stessa quantità ma rimarrà invariato altrove. Questa proprietà è alla base della robustezza rispetto alle traslazioni e alle distorsioni dell'immagine in ingresso; in secondo luogo, mettendo in fila diversi strati di convoluzione si ottiene una rete capace di avere una comprensione più "astratta" dell'immagine in ingresso. Il primo strato di convoluzione si occupa di estrarre features direttamente dai pixel grezzi dell'immagine e li memorizza nelle feature maps. Questo output diviene poi l'input di un successivo livello di convoluzione, il quale andrà a fare una seconda estrazione delle caratteristiche, combinando le informazioni dello strato precedente. Da questa astrazione a più livelli deriva una maggior comprensione delle features.

3.2.2 Strato di ReLU

Nel capitolo 2 si è detto che la funzione sigmoide non era la più efficiente. Difatti, negli anni si è stabilita con sicurezza la *Rectified Linear Unit* (ReLU) come funzione d'attivazione più efficace. La ReLU è più verosimile alla modalità di attivazione biologica dei nostri neuroni[**Relu**], ed è definita come:

$$f(x) = max(0, x)$$

Y. LeCun ha dichiarato che la ReLU è inaspettatamente *"l'elemento singolo più importante di tutta l'architettura per un sistema di riconoscimento"*. Questo può essere dovuto principalmente a 2 motivi:

1. la polarità delle caratteristiche è molto spesso irrilevante per riconoscere gli oggetti;
2. la ReLU evita che quando si esegue pooling (sezione 3.2.3) due caratteristiche entrambe importanti ma con polarità opposte si cancellino fra loro

3.2.3 Strato di Pooling

Un'altra proprietà che si vuole ottenere per migliorare i risultati sulla visione artificiale è il riconoscimento delle features indipendentemente dalla posizione nell'immagine, perché l'obiettivo è quello di rafforzare l'efficacia contro le traslazioni e le distorsioni. Questo si può ottenere diminuendo la risoluzione spaziale dell'immagine, il che favorisce una maggiore velocità di computazione ed è al contempo una contromisura contro l'overfitting, dato che si diminuisce il numero di parametri.
Lo strato di pooling ottiene in ingresso N immagini di una risoluzione e restituisce in uscita lo stesso numero di immagini, ma con una risoluzione ridotta in una certa misura, solitamente del 75%. Infatti, la forma più comune di pooling layer utilizza dei filtri 2x2, che dividono l'immagine in zone di 4 pixel non sovrapposte e per ogni zona scelgono un solo pixel.

I criteri con cui scegliere il pixel vincente sono diversi:

- average pooling: si calcola il valore medio sui pixel del pool;
- median pooling: si calcola la mediana dei valori dei pixel del pool;
- LP-pooling: si calcola la p-norma della matrice dei pixel;
- max pooling: si sceglie il pixel col valore più alto;

Di questi, quello che si è dimostrato più efficace è il *max pooling*, figura 3.5. Attraver-

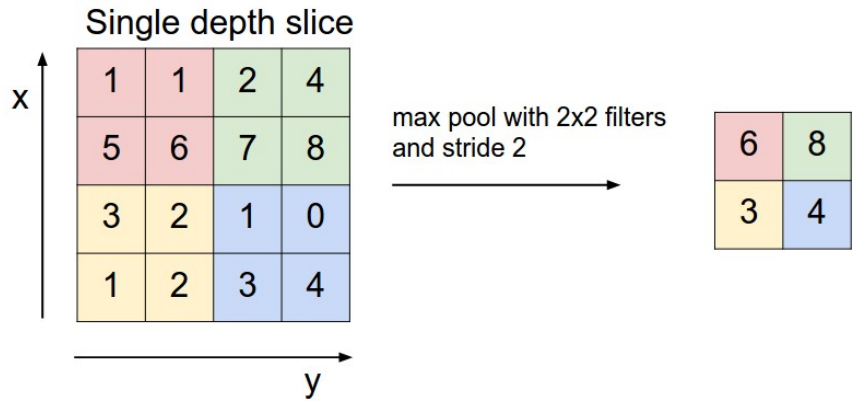


Figura 3.5: Max pooling: in uscita l'immagine avrà 1/4 dei pixel di partenza.

sando la rete si avrà gradualmente un numero di feature maps più alto e quindi della ricchezza della rappresentazione delle features; ed una diminuzione della risoluzione dell'input. Questi fattori combinati insieme donano un forte grado di invarianza alle trasformazioni geometriche dell'input.

3.2.4 Strato completamente connesso (FC)

Nello strato completamente connesso, tutti gli input provenienti dai layer di convoluzione vengono dati in ingresso ad una normale rete neurale completamente connessa che fungerà da classificatore. I calcoli in questa parte finale sono quindi uguali alle moltiplicazioni tra matrici visti nel Capitolo 2.
Ultimamente però, si è notato che, eccetto per la modalità di connessione, questi neuroni sono funzionalmente identici a quelli dei layer di convoluzione (entrambi computano moltiplicazioni fra matrici). Quindi si possono sostituire con degli strati di convoluzione che hanno un receptive field di 1x1[**WCS231layer**].

In figura 3.6 è rappresentata l'architettura completa di una CNN. Si noti come la risoluzione dell'immagine si riduca ad ogni strato di pooling (chiamato anche subsampling) e come ogni pixel delle feature maps derivi dal campo recettivo sull'insieme di tutte le feature maps del livello precedente.
In figura 3.7 invece, si può osservare una CNN nell'atto di classificazione di un'auto. Sono visualizzati i filtri della rete durante tutti i vari livelli di elaborazionde dell'input, per poi terminare in uno strato completamente connesso che da in output una probabilità. Questa probabilità è poi tradotta in uno score, da cui si sceglie la classe vincente.

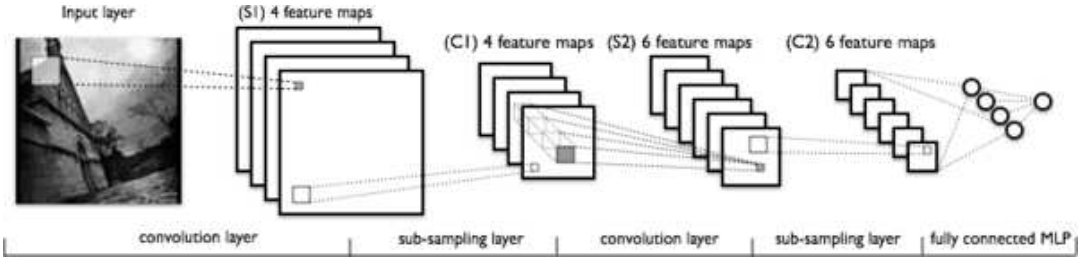


Figura 3.6: Architettura di una CNN

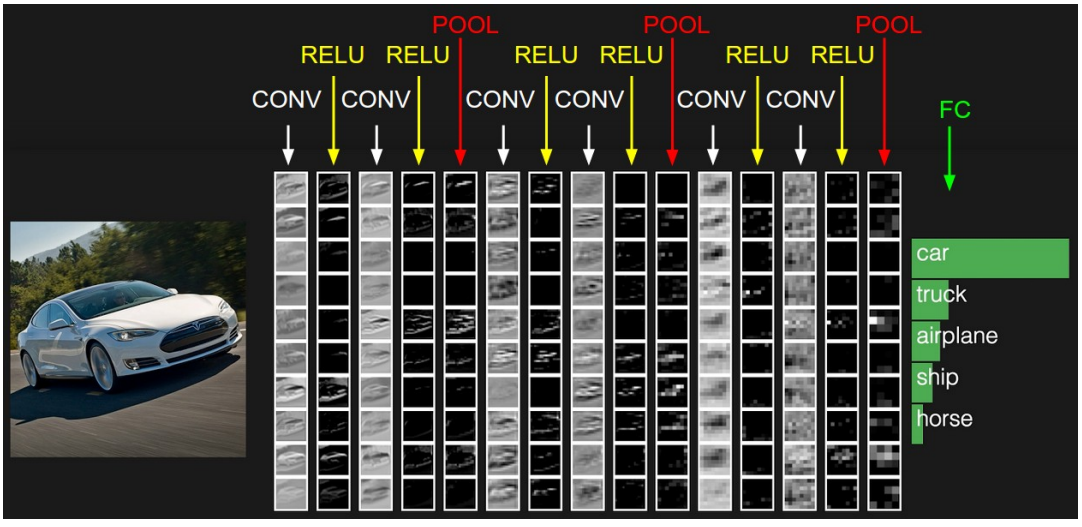


Figura 3.7: Tipica CNN in un task di classificazione; la classe vincente è quella con la probabilità più alta, indicata alla fine

3.3 Applicazioni e risultati

L'alta efficacia, la peculiare, vantaggiosa architettura insieme con l'enorme progresso tecnologico dell'hardware, hanno reso le CNN il sistema più promettente per compiti di visivo, con i più svariati ambiti applicativi come: riconoscimento e tagging facciale (si pensi a Facebook), ricerca intelligente di immagini (si pensi a Google Photos), automobili autonome, smartphone, robot, droni, (video)giochi ed altro. Le CNN hanno avuto eccellenti risultati anche nell'elaborazione naturale del linguaggio; nella scoperta di farmaci, poiché predicendo le interazioni tra determinate molecole e proteine hanno contribuito a scoprire potenziali biomolecole per il trattamento dell'ebola[**WCNN**], tanto per citarne altri.
Già 3 anni fa, in un articolo pubblicato dal dipartimento di visione artificiale del KTH [**Overfeat**], ha utilizzato "OverFeat", una CNN pubblica allenata per ILSVRC13, una competizione annuale di riconoscimento degli oggetti. L'articolo sottolinea come abbiano utilizzato questa CNN "off-the-shelf" ovvero già pronta e, senza allenarla ulteriormente, testandola contro altri metodi allo stato dell'arte finemente perfezionati sviluppati fino ad allora. Come test hanno scelto attività gradualmente sempre più lontane dall'originario compito per cui OverFeat è stata addestrata e, con enorme stupore, hanno verificato che OverFeat surclassa i suddetti metodi su qualsiasi dataset (si rimanda all'articolo per i dettagli) nonostante sia stata allenata solo mediante l'ImageNet. L'articolo si chiude con una frase che qui cito:

> *"Thus, it can be concluded that from now on, deep learning with CNN has to be considered as the primary candidate in essentially any visual recognition task."*

3.3.1 Confronto con l'uomo

Nel 2011, le CNN hanno la prima volta battuto l'uomo raggiungendo un errore di 0.56% contro l'1.16% degli umani sul riconoscimento dei segnali stradali nella competizione "German Traffic Sign competition run by IJCNN 2011".
Un altro compito che è sempre stato difficile per la visione artificiale era il riconoscimento di visi parzialmente occlusi, capovolti o spostati di diverse angolazioni. Tuttavia, nel 2015 un team del Yahoo Labs, è riuscito a far apprendere anche questo compito ad una CNN[**WMit**].

L'ultima pietra migliare in ordine cronologico della sfida "Uomo vs. Macchina" è senza dubbio quella di AlphaGo[**WAlphaGo**]. AlphaGo è il primo programma a riuscire a battere un giocatore umano professionista (Lee Sedol, 18 volte campione del mondo) all'antico gioco cinese di Go.
Go è conosciuto per essere computazionalmente estremamente complesso: vi sono 10^{170} possibili combinazioni della scacchiera, un numero più alto degli atomi dell'Universo conosciuto. È quindi inaffrontabile con un approccio a "forza bruta".
AlphaGo si basa su una combinazione di deep learning + tree search. In particolare, utilizza 3 CNN: 2 "policy network" per scegliere la strategia più vincente ed 1 "value network" come funzione euristica per valutare la bontà di un'ipotetica mossa. In più, l'output di queste reti è combinato con una "Montecarlo Tree Search" per avere una risposta finale sulla prossima mossa da giocare. Maggiori dettagli si trovano sul lungo paper pubblicato da DeepMind[**AlphaGo**].

Questi risultati bastano per comprendere la potenzialità delle convolutional neural networks.

Capitolo 4

CNN: implementazione e addestramento

In questo capitolo verranno messi in pratica i concetti introdotti nel Capitolo 3. In particolare, verrà addestrata una CNN su 2 dataset, MNIST e CIFAR. I risultati serviranno da confronto con ResNet, la rete utilizzata nel Capitolo 5.

4.1 Modello di CNN basato su LeNet

La rete implementata è basata sulla capostipite delle CNN, `LeNet5`, resa celebre da Y. LeCun per il riconoscimento di caratteri, anche scritti a mano[**lenet**]. L'architettura è rappresentata in figura 4.1. Si noti che la rete è appositamente progettata per avere in input immagini di 32x32 pixel, adatta ai dataset che si introduranno dopo.

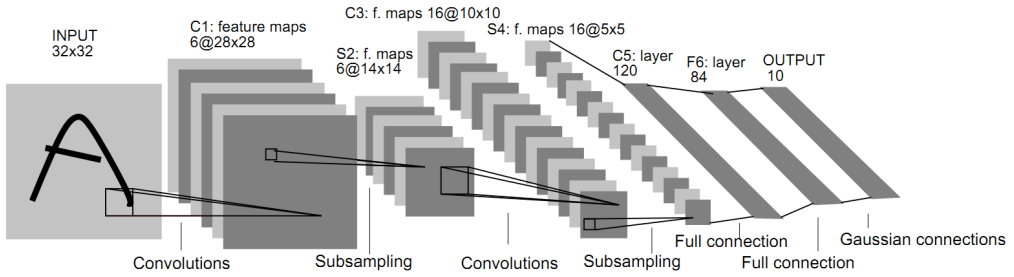


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Figura 4.1: Architettura di LeNet5, qui per il riconoscimento di caratteri scritti a mano

Per implementare la rete verrà impiegato il già citato framework Torch. Per saperne di più su Torch e sugli utilizzi basilari per le reti neurali si veda l'appendice B.

Si procede quindi, ad implementare la rete costituita da:

- 2x → Convolutional layers - ReLU - Pooling layers
- 1x → Multi-layer perceptron completamente connesso, con 1 hidden layer

4.1.1 Definire la loss function

Ora che si ha un modello, va definita una funzione di costo da minimizzare che, come visto nel capitolo 2, sta alla base dell'apprendimento. Nel capitolo 2 però, si era definito tutto "a mano", mentre ora si usano le librerie di Torch.
Con Torch si possono definire diverse funzioni di costo, a seconda del principio di

apprendimento che vogliamo utilizzare. Come ad esempio lo scarto quadratico che però non è in generale una scelta ottimale. Una delle più convienienti per i problemi di classificazione è la *negative-likelihood* (likelihood = funzione di verosimiglianza). La negative-likelihood necessita in ingresso di un vettori di probabilità logaritmiche normalizzate. Quindi, prima si trasforma l'output del classificatore in probabilità logaritmica aggiungendo un modulo chiamato `LogSoftMax` e poi si calcola l'errore sulla negative-likelihood. In Torch questo è semplicissimo:

```
--define the loss function
model:add(nn.LogSoftMax())

--criterion (i.e. the loss) will be the negative-likelihood
criterion = nn.ClassNLLCriterion()
```

4.2 Preprocessing del dataset MNIST

4.3 Addestramento su MNIST

4.4 Preprocessing del dataset CIFAR

4.5 Addestramento su CIFAR

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

Figura 4.2: Esempi di cifre classificate correttamente

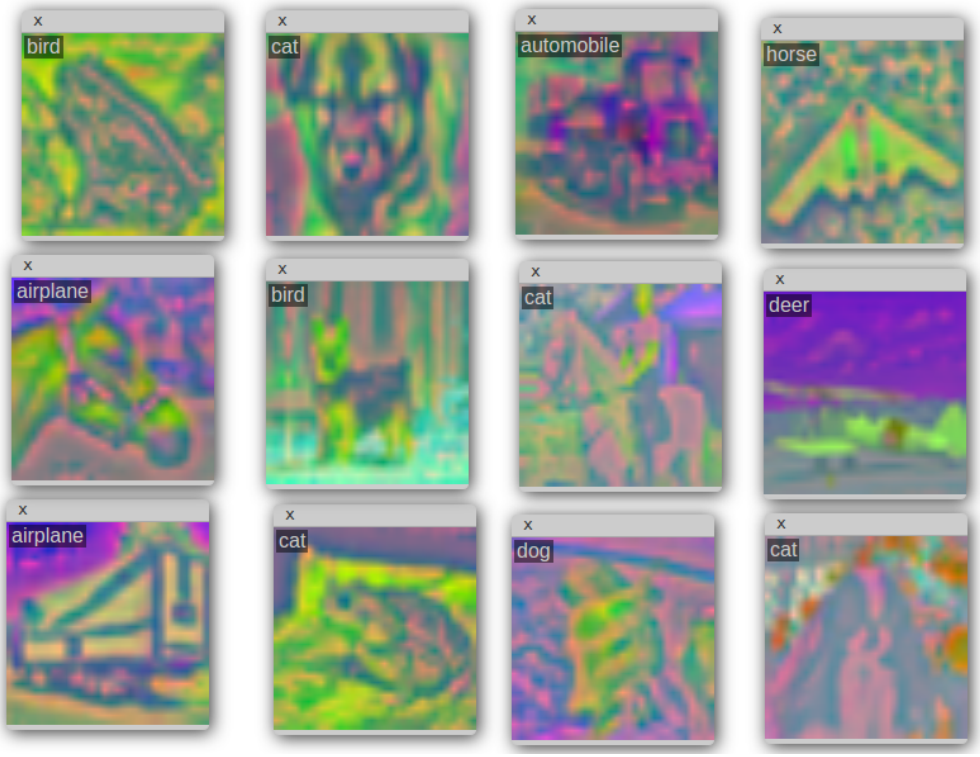


Figura 4.3: Esempi di animali classificati erroneamente

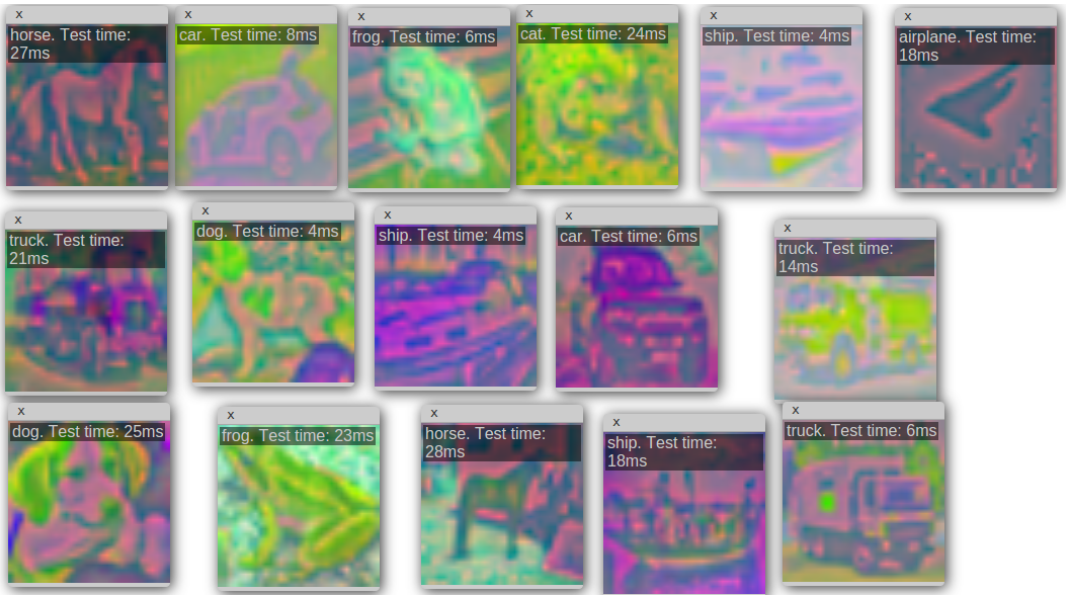


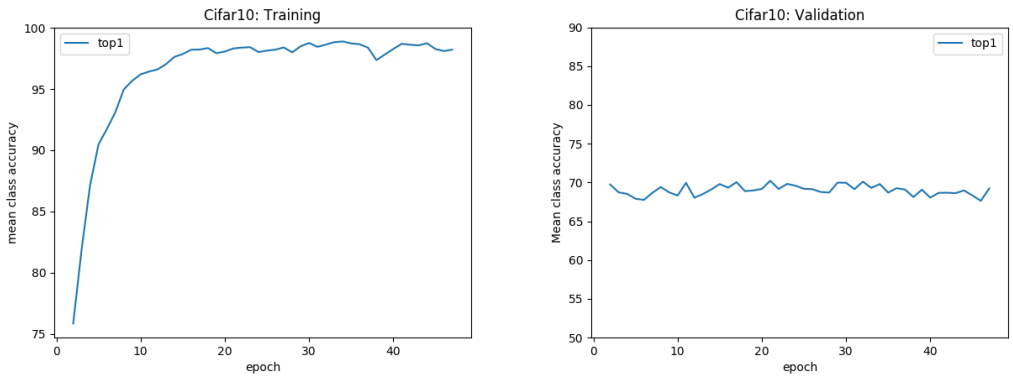
Figura 4.4: Esempi di animali classificati correttamente

```
<trainer> time to test 1 sample = 20.104405999184ms==========================>.] ETA: 155ms | Step: 17ms
ConfusionMatrix:
[[     190       0       0       0       0       0       1       0       0       0]   99.476%   [class: 0]
 [       0     217       1       1       0       0       0       1       0       0]   98.636%   [class: 1]
 [       0       0     197       0       0       0       0       1       0       0]   99.495%   [class: 2]
 [       0       0       0     189       0       0       0       1       0       1]   98.953%   [class: 3]
 [       0       0       0       0     212       0       1       0       0       1]   99.065%   [class: 4]
 [       0       0       0       0       0     180       0       0       0       0]   100.000%  [class: 5]
 [       1       0       0       0       0       0     199       0       0       0]   99.500%   [class: 6]
 [       0       0       0       0       0       0       0     224       0       0]   100.000%  [class: 7]
 [       0       0       0       0       0       0       0       0     171       1]   99.419%   [class: 8]
 [       2       0       0       1       0       0       0       0       0     207]]  98.571%   [class: 9]
 + average row correct: 99.311608672142%
 + average rowUcol correct (VOC measure): 98.63573372364%
 + global correct: 99.3%
```

Figura 4.5: Confusion Matrix MNIST: la matrice che mostra le percentuali di accuratezza ed errore sul validation set per ogni classe

```
 [================================ 10000/10000 ============================>] ETA: 0ms | Step: 8ms
<trainer> time to test 1 sample = 10.493553209305ms
ConfusionMatrix:
[[     775      24      36      23      15      12       7      16      63      29]   77.500%   [class: airplane]
 [      24     842       4       9       3       3       6       6      28      75]   84.200%   [class: automobile]
 [      64       4     611      79      83      59      52      27      12       9]   61.100%   [class: bird]
 [      25       8      51     583      43     171      60      21      19      19]   58.300%   [class: cat]
 [      18       4      48      50     701      48      50      62      11       8]   70.100%   [class: deer]
```

Figura 4.6: Confusion Matrix CIFAR: la matrice che mostra le percentuali di accuratezza ed errore sul validation set per ogni classe



(a) Accuracy sul training set

(b) Accuracy sul validation set

Figura 4.7: Performance della rete in training e validation

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 addizionale

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.

```
-- 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
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` :

```
--creating class NN in Lua, using a nice class utility
class = require 'class'

Neural_Network = class('Neural_Network')

--init NN
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

--define a forward method
function Neural_Network:forward(X)
   --Propagate inputs though network
   self.z2 = th.mm(X, self.W1)
   self.a2 = th.sigmoid(self.z2)
   self.z3 = th.mm(self.a2, self.W2)
   yHat = th.sigmoid(self.z3)
   return yHat
end

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

function Neural_Network:costFunction(X, y)
   --Compute the cost for given X,y, use weights already stored in class
   self.yHat = self:forward(X)
   --NB torch.sum() isn't equivalent to python sum() built-in method
   --However, for 2D arrays whose one dimension is 1, it won't make any
    difference
   J = 0.5 * th.sum(th.pow((y-yHat),2))
   return J
end

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
```

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(
       self.W2:nElement())))
5     return params
6  end
7
8  function Neural_Network:setParams(params)
9     --Set W1 and W2 using single paramater vector.
10    W1_start = 1 --index starts at 1 in Lua
11    W1_end = self.hiddenLayerSize * self.inputLayerSize
12    self.W1 = th.reshape(params[{ {W1_start, W1_end} }],
       self.inputLayerSize, self.hiddenLayerSize)
13    W2_end = W1_end + self.hiddenLayerSize*self.outputLayerSize
14    self.W2 = th.reshape(params[{ {W1_end+1, W2_end} }],
       self.hiddenLayerSize, self.outputLayerSize)
15 end
16
17 --this is like the getParameters(): method in the NN module of torch, i.e.
       compute the gradients and returns a flattened grads array
18 function Neural_Network:computeGradients(X, y)
19    dJdW1, dJdW2 = self:costFunctionPrime(X, y)
20    return th.cat((dJdW1:view(dJdW1:nElement())), (dJdW2:view(dJdW2:
       nElement())))
21 end
```

Appendice B

Il framework Torch

[**WTorch**]