

UNIVERSITÀ DI BOLOGNA

*Attività progettuale di Sistemi Intelligenti M*

---

# Reti neurali artificiali: dal MLP agli ultimi modelli di CNN

---

*Autore:*

Ali Alessio Salman

31 agosto 2017



# Indice

<b>1</b>	<b>Reti Neurali Artificiali: le basi</b>	<b>1</b>
1.1	Breve introduzione . . . . .	1
1.2	Multi-layer Perceptron . . . . .	2
1.2.1	Strati Nascosti . . . . .	3
1.3	Caso di studio: prevedere il profitto di un ristorante . . . . .	3
<b>2</b>	<b>Implementazione di un MLP</b>	<b>5</b>
2.1	Dataset e Architettura . . . . .	5
2.2	Forward Propagation . . . . .	6
2.3	Backpropagation . . . . .	8
2.4	Verifica numerica del gradiente . . . . .	11
2.5	Addestramento . . . . .	13
2.5.1	Apprendimento supervisionato . . . . .	13
2.5.2	Discesa del gradiente . . . . .	13
2.6	Ottimizzazione: diverse tecniche . . . . .	14
2.7	Overfitting . . . . .	18
2.7.1	Rilevare l'overfitting . . . . .	18
2.7.2	Contromisure . . . . .	19
2.8	Risultati . . . . .	22
<b>3</b>	<b>Reti Neurali Convolutionali</b>	<b>23</b>
3.1	Breve introduzione . . . . .	23
3.2	Architettura . . . . .	23
3.2.1	Strato di Convoluzione . . . . .	24
3.2.2	Strato di ReLU . . . . .	26
3.2.3	Strato di Pooling . . . . .	27
3.2.4	Strato completamente connesso (FC) . . . . .	28
3.3	Applicazioni e risultati . . . . .	29
3.3.1	Confronto con l'uomo . . . . .	29
<b>4</b>	<b>CNN: implementazione e addestramento</b>	<b>31</b>
4.1	Modello di CNN basato su LeNet . . . . .	31
4.1.1	Definire la loss function . . . . .	32
4.2	Procedura di training . . . . .	32
4.3	MNIST: preprocessing . . . . .	34
4.4	MNIST: addestramento . . . . .	34
4.5	CIFAR: preprocessing . . . . .	35
4.6	CIFAR: addestramento . . . . .	37
<b>5</b>	<b>Addestrare un modello allo stato dell'arte</b>	<b>39</b>
5.1	Deep Residual Network . . . . .	39
5.2	Implementazione di ResNet di Facebook . . . . .	41
5.3	Addestramento su CIFAR . . . . .	42

5.4	LeNet vs Resnet: confronto . . . . .	42
<b>6</b>	<b>Caso d'uso attuale: fine-tuning su dataset arbitrario</b>	<b>45</b>
6.1	Dataset . . . . .	45
6.2	Fine-Tuning . . . . .	45
6.3	ModelZoo . . . . .	45
6.3.1	Fine-tuning su Resnet . . . . .	45
<b>7</b>	<b>Conclusioni</b>	<b>47</b>
<b>A</b>	<b>MLP: Codice aggiuntivo</b>	<b>49</b>
A.1	Classi in Lua . . . . .	49
A.2	La classe Neural_Network . . . . .	50
A.3	Metodi getter e setter . . . . .	51
<b>B</b>	<b>Il framework Torch</b>	<b>53</b>
B.1	Introduzione . . . . .	53
B.2	Utilizzo base per reti neurali . . . . .	53
B.2.1	Supporto CUDA . . . . .	53
B.3	ResNet . . . . .	53

## 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).

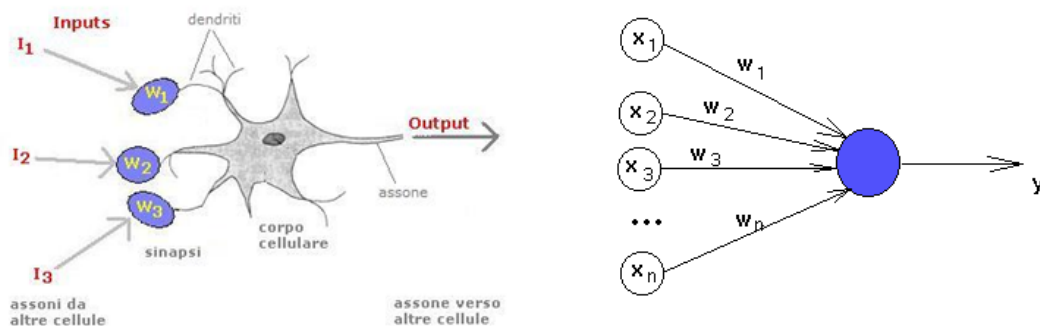


Figura 1.1: Modello di calcolo di un neurone (a sinistra) e schema del neurone artificiale (a destra)

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 funzione di trasferimento non lineare:

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

ove  $H$  è la funzione gradino di Heaviside [**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**].

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

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

dove  $\eta$  è una costante di apprendimento strettamente positiva che regola la velocità di apprendimento, detta *learning rate* e  $\delta$  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.

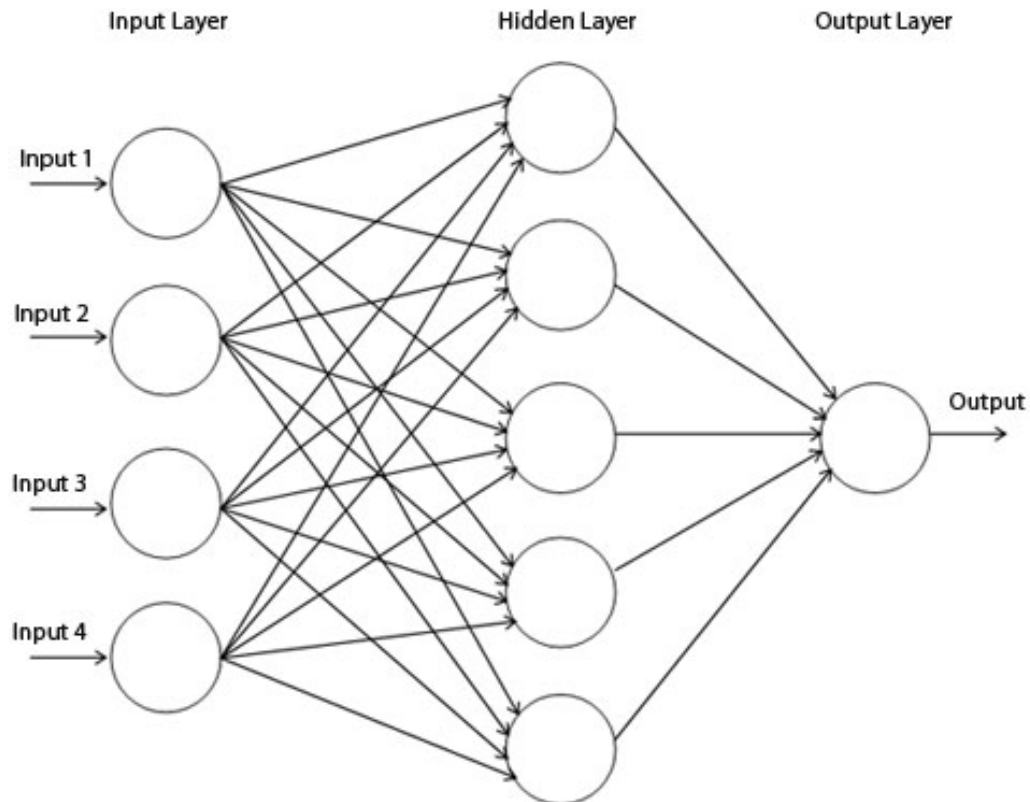


Figura 1.2: Struttura di un percettrone multistrato con un solo strato nascosto

Il *mlp* possiede le seguenti caratteristiche:

- ogni neurone è un percettore 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 finito di neuroni, può essere addestrata per approssimare una qualsiasi funzione continua su uno spazio compatto di  $\mathbb{R}^n$ . In altre parole, un singolo strato nascosto è abbastanza potente da imparare un ampio numero di funzioni. Precisamente, una rete a 3 strati è in grado di separare regioni convesse con un numero di lati  $\leq$  numero neuroni nascosti.

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

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

## 1.3 Caso di studio: prevedere il profitto di un ristorante

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

*"Loris è figlio della titolare di una famoso spaccio di piadine nel Riminese e sta tornando in Italia dopo aver frequentato con successo un prestigioso Master in Business Administration ad Harvard, a cui si è iscritto inseguendo il sogno di esportare in tutto il mondo la piadina romagnola. Nel*

*lungo viaggio in prima classe, medita su come presentare alla mamma, che sa essere un tantino restia alle innovazioni, il progetto di aprire un ristorante a New York City."*

Loris ha esportato con successo la piadina a NY, si veda [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'appendice B) e mantenere le consistenza con i capitoli successivi, nei quali si userà ancora Torch per addestrare reti neurali molto più complesse.

### 2.1 Dataset e Architettura

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

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

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

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

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

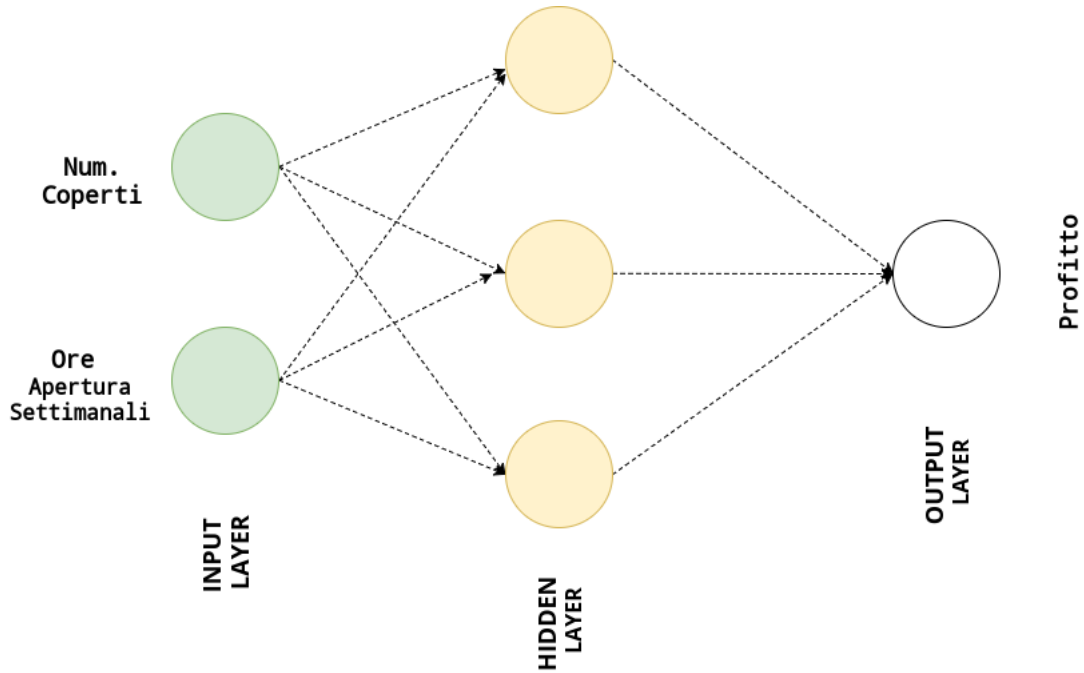


Figura 2.1: Architettura del MLP per la previsione dei profitti

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

```

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

```

```

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

```

## 2.2 Forward Propagation

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

Tabella 2.1: Variabili usate nel testo e nel codice

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

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

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

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

Si noti che  $z^{(2)}$  è di dimensione 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.

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

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

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

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

Essendo  $a^{(2)}$  di dimensione 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.

```

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

```

## 2.3 Backpropagation

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

Non si può introdurre l'algoritmo di "*backprop*" senza prima spiegare il concetto di funzione di costo (o *loss function*). Nelle reti neurali (e più specificatamente nell'apprendimento supervisionato [**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.

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

Da cui, il codice in Lua:

```

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

```

La *backprop* ha alcuni requisiti:

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

Sotto queste condizioni l'algoritmo sfrutta la regola della catena [**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* (figura 2.2); ovvero variano in maniera tale da minimizzare la funzione di costo  $J$ .

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

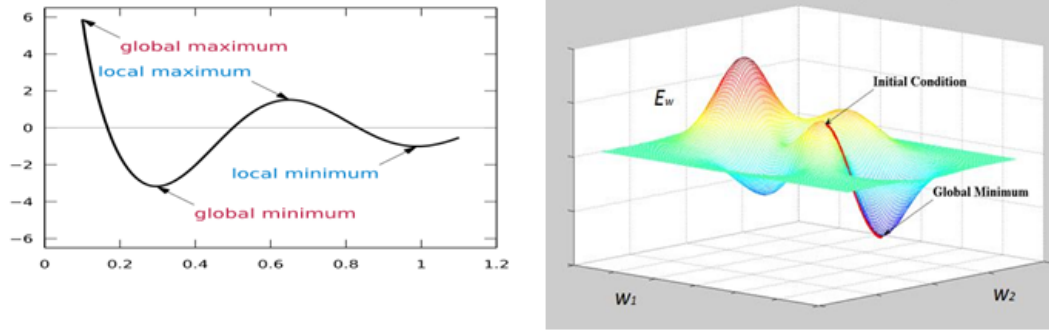


Figura 2.2: Cercare il minimo di una fz. seguendo la discesa del gradiente

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

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

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

Sviluppando i calcoli si ottiene:

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

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

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

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

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

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

Definiamola, quindi, nel codice:

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

L'equazione così ottenuta è:

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

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

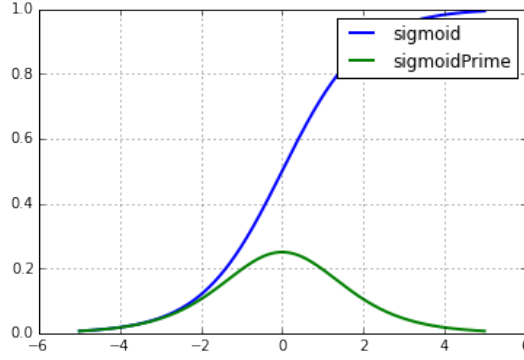


Figura 2.3: La funzione sigmoide e la sua derivata

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

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

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

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

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

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

Ora bisogna moltiplicare l'errore con  $a^{(2)}$ . Come indicato nelle ottime dispense di CS231 di Stanford[**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:

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

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

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

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

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

```

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

```

## 2.4 Verifica numerica del gradiente

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

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

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

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

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

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

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

```

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

```

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

```

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

```



```

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

```

Per calcolare *quanto* siano effettivamente uguali i due gradienti si può usare un rapporto basato sulla 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 $ th 4_gradCheck.lua
2 Gradient checking...
3 The difference is 2.123898e-10

```

## 2.5 Addestramento

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

### 2.5.1 Apprendimento supervisionato

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

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

Riguardo all'ultimo punto ci sono diverse possibilità.

### 2.5.2 Discesa del gradiente

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

## Fase di addestramento

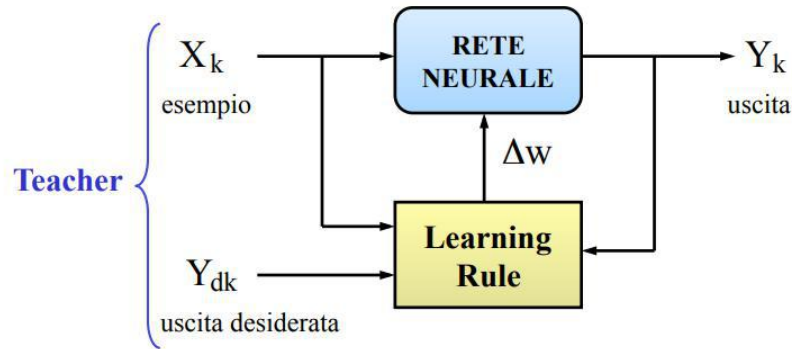


Figura 2.4: Apprendimento supervisionato: schema generale

*Descent*[WSGD]. Calcolando il gradiente  $\nabla J$  si conosce la direzione di massima variazione quindi ci si sposta - lungo l'opposto di questa direzione, l'antigradiente - di una quantità pari a  $\eta$ . Questo parametro si chiama *learning rate* e regola appunto la velocità dell'apprendimento.

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

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

Questa regola di apprendimento però soffre di alcuni problemi:

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

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

## 2.6 Ottimizzazione: diverse tecniche

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

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

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

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

- MOMENTUM & NAG: Il "*Momentum*" controlla la quantità d'inerzia nella modifica dei pesi sinaptici, memorizzando nell'equazione la variazione  $\Delta W$  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 acquisisce 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.

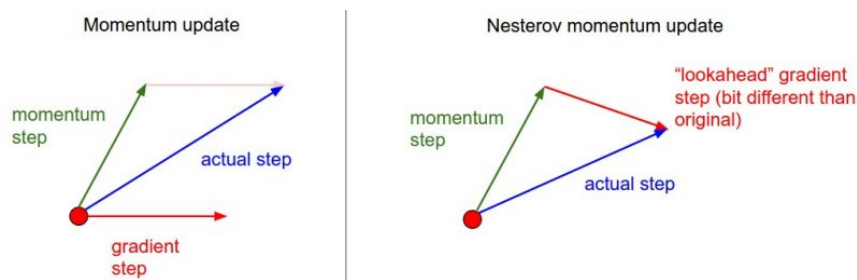


Figura 2.5: Confronto tra il momentum classico e NAG

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

- 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  $\beta_1$  e  $\beta_2$  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.

```

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

```

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

Dal grafico si evidenzia come le premesse teoriche dei vari metodi siano state pienamente rispettate. Nonostante il problema sia molto semplice se comparato ai reali problemi di *deep learning*, già su un dataset ed un numero di iterazioni così limitato si nota la superiorità di un metodo verso il precedente. In particolare, nel corso "CS231" di deep learning applicato alla visione artificiale di Stanford[**WCS231adam**], sviluppato da Andrej Karpathy et. al, si raccomanda Adam come metodo di default per applicazioni di deep learning:

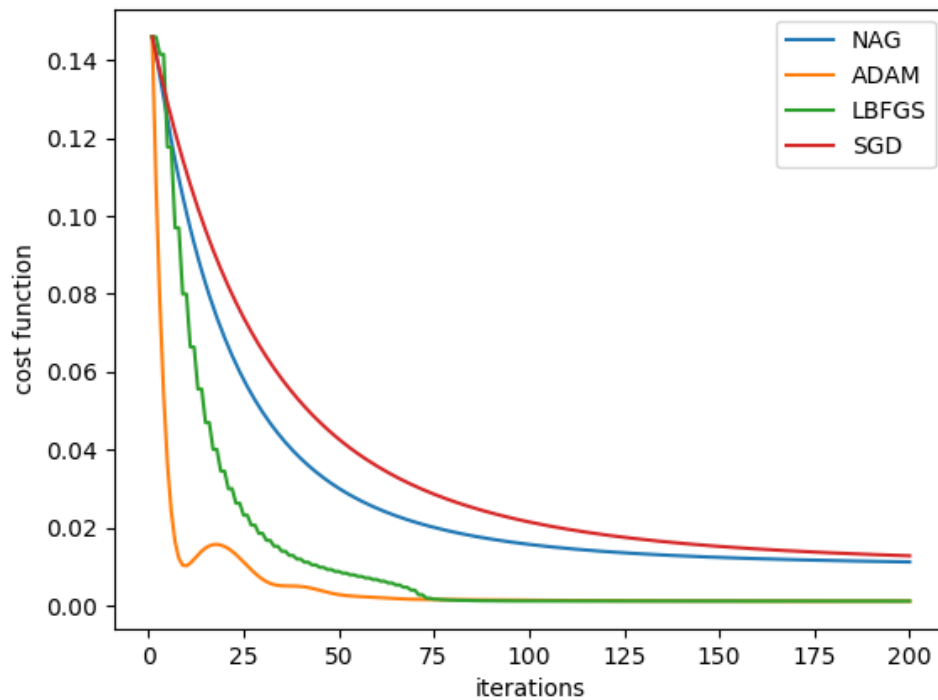


Figura 2.6: Confronto dei metodi di ottimizzazione durante il training

*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.

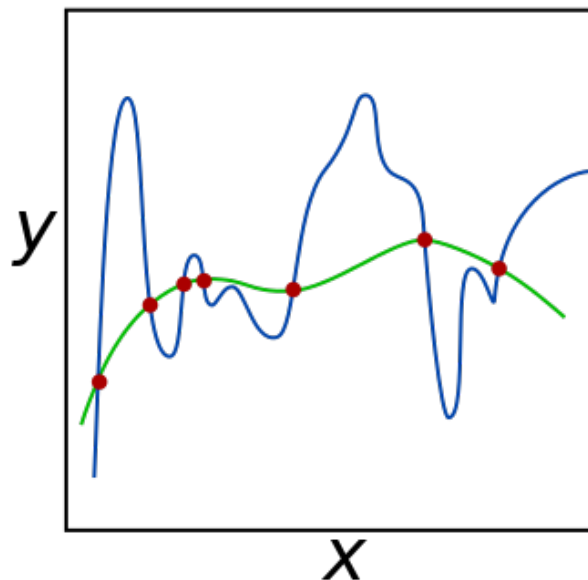


Figura 2.8: Entrambe le funzioni hanno un'errore nullo sul dataset. Tuttavia, la funzione blu è eccessivamente complessa ed affetta da rumore; avrà quindi cattive capacità di generalizzazione. La verde, al contrario, sarà presumibilmente più corretta rispetto ai punti sconosciuti della distribuzione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

### 2.7.1 Rilevare l'overfitting

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

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

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

```

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

```

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

```

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

```

Andando infine a plottare i dati così ottenuti, 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 ovviamente 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

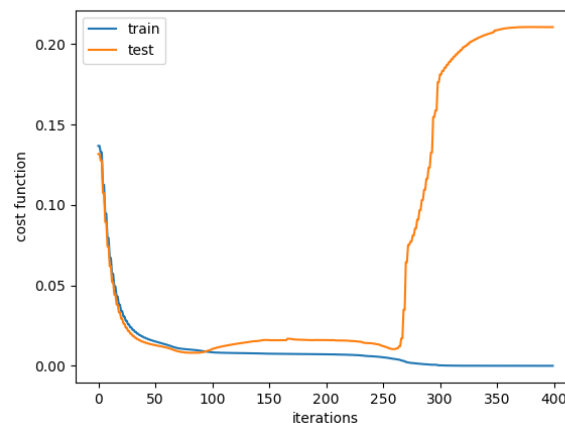


Figura 2.9: La rete mostra overfitting. Dopo l'iterazione 100 le performance sul test set iniziano ad essere sbagliate. Attorno alla 250 il modello diventa troppo complesso e le predizioni sono pessime

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 (setstando 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  $\lambda$  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

```



```

7  --so, the new Neural_Network class now becomes:
8  Neural_Network = class(function(net, inputs, hidden, outputs, lambda)
9      net.inputLayerSize = inputs
10     net.hiddenLayerSize = hidden
11     net.outputLayerSize = outputs
12     net.W1 = th.randn(net.inputLayerSize, net.hiddenLayerSize)
13     net.W2 = th.randn(net.hiddenLayerSize, net.outputLayerSize)
14
15     --regularization parameter
16     net.lambda = lambda
17 end)
18
19 function Neural_Network:costFunction(X, y)
20     --Compute the cost for given X,y, use weights already stored in class
21     self.yHat = self:forward(X)
22     J = 0.5 * th.sum(th.pow((y-yHat),2))/X:size()[1] +
23         (self.lambda/2) * (th.sum(th.pow(self.W1,2)) + th.sum(th.pow(
24             self.W2, 2)))
25     return J
26 end
27
28 function Neural_Network:d_costFunction(X, y)
29     --Compute derivative wrt to W and W2 for a given X and y
30     self.yHat = self:forward(X)
31     delta3 = th.cmul(-(y-self.yHat), self:d_Sigmoid(self.z3))
32     --Add gradient of regularization term:
33     dJdW2 = th.mm(self.a2:t(), delta3)/X:size()[1] + self.lambda*self.W2
34
35     delta2 = th.mm(delta3, self.W2:t()):cmul(self:d_Sigmoid(self.z2))
36     --Add gradient of regularization term:
37     dJdW1 = th.mm(X:t(), delta2)/X:size()[1] + self.lambda*self.W1
38
39     return dJdW1, dJdW2
40 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.

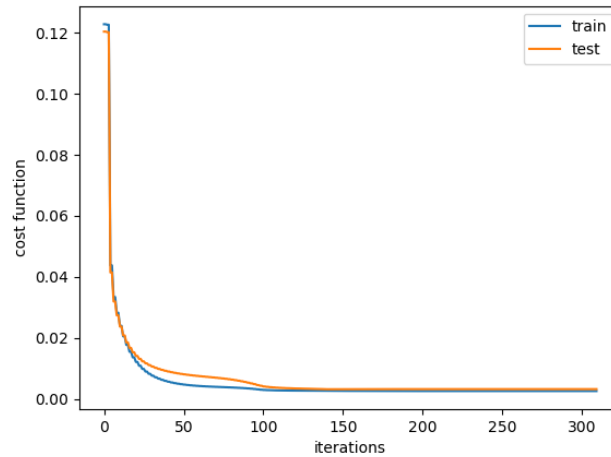


Figura 2.10: Dopo aver applicato la L2 regularization, la rete non è più affetta da overfitting.

## 2.8 Risultati

In questo capitolo si è visto come costruire da zero l'architettura di un perceptrone 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 ottenuta 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

Le reti neurali convoluzionali, alle quali ci riferiremo con l'abbreviazione *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.

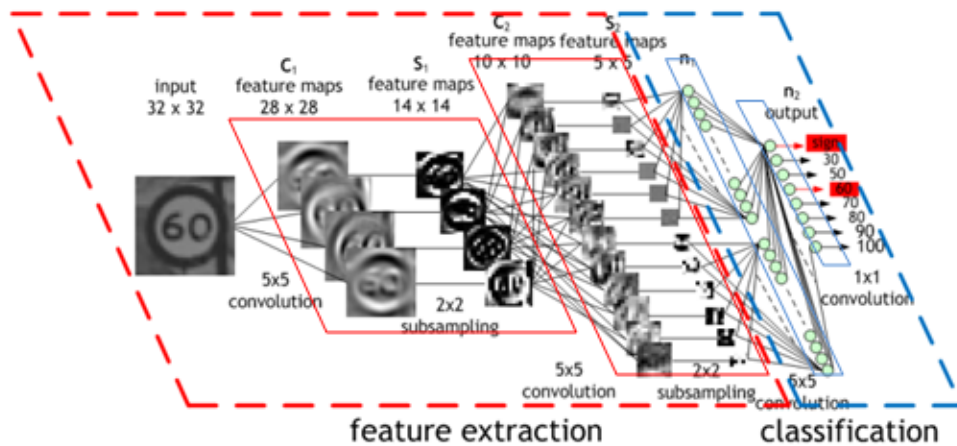


Figura 3.1: Architettura di una CNN che classifica segnali stradali: si evidenzia la divisione tra gli strati che fungono da feature extractor ed il classificatore finale

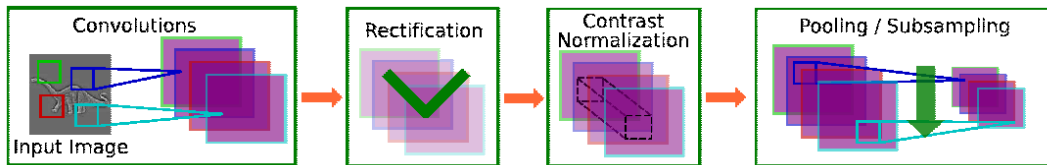


Figura 3.2: I diversi strati tipici di una CNN

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 disarmonica, 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  $M \times N$  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:

$$y[m, n] = x[m, n] \otimes h[m, n] = \sum_i^m \sum_j^n x[i, j] \times h[m - i, n - j]$$

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.

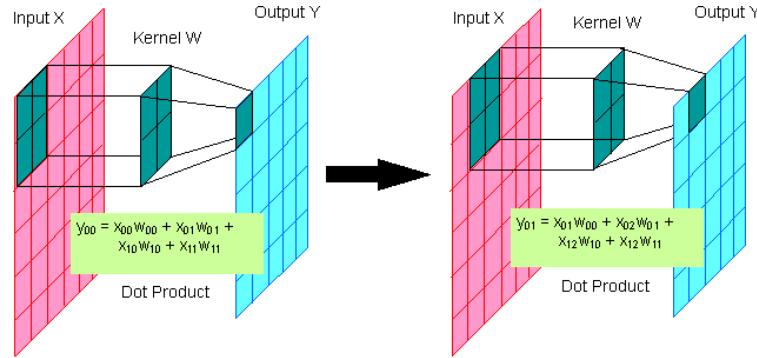


Figura 3.3: Convoluzione con un kernel: primi due step

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:

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

$$W_2 = (W_1 - F + 2P)/S + 1$$

$$D_2 = K$$

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.

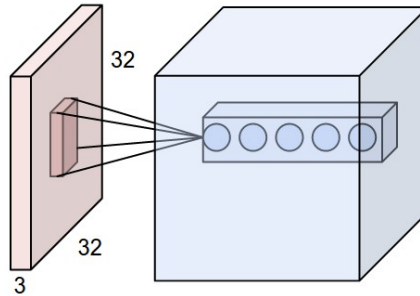


Figura 3.4: Ogni neurone è connesso a solo 1 regione locale dell'input ma a tutta la profondità (i.e. canali colori). La depth dell'output è data dal numero  $K$  di filtri, in questo caso 5

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  $2 \times 2$ , 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-

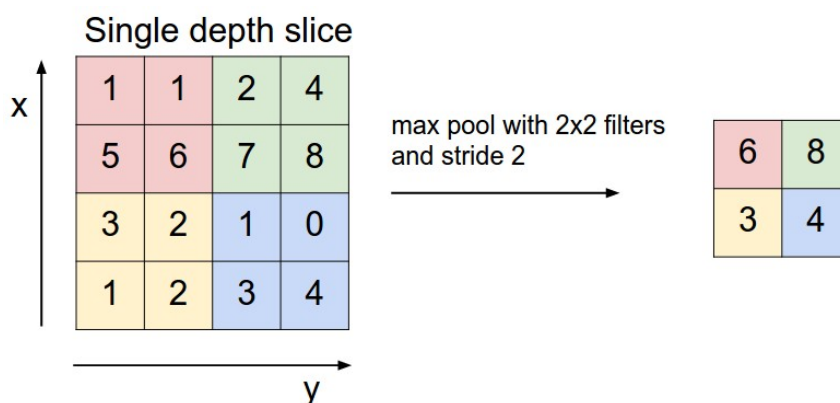


Figura 3.5: Max pooling: in uscita l’immagine avrà 1/4 dei pixel di partenza.

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  $1 \times 1$  [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 sub-sampling) 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 elaborazione 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.

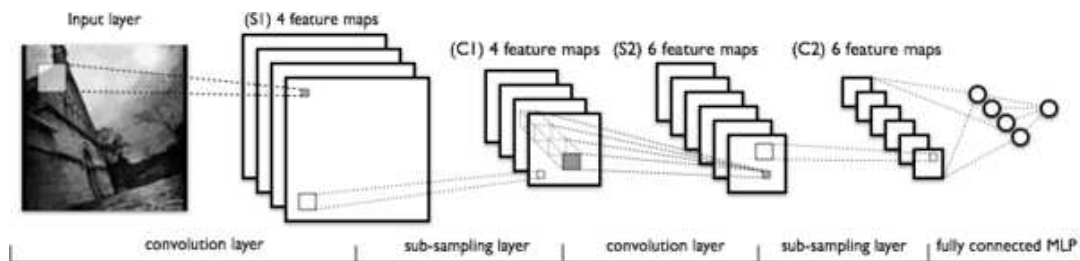


Figura 3.6: Architettura di una CNN

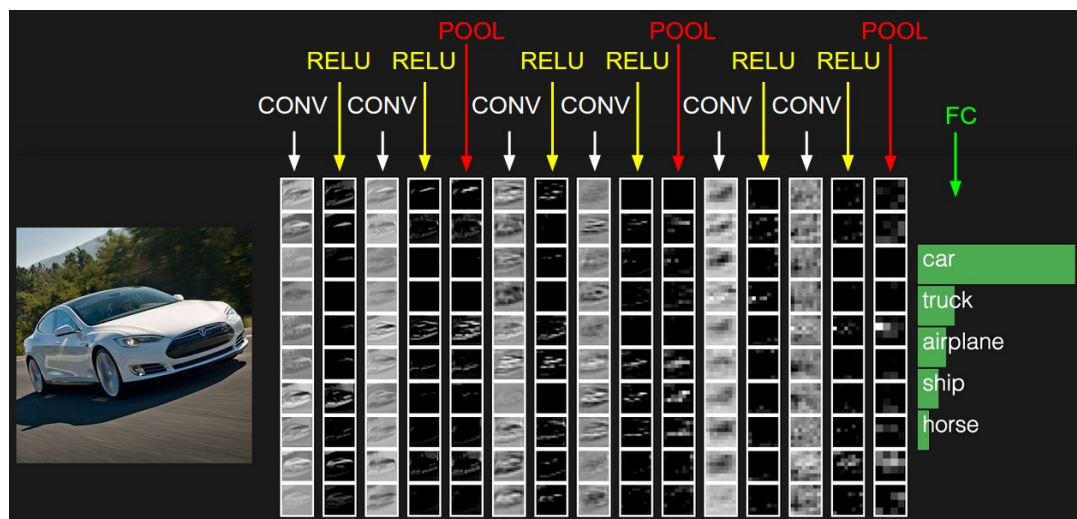


Figura 3.7: Tipica CNN in un task di classificazione; la classe vincente è quella con la probabilità più alta, indicata alla fine



### 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".

Due anni fa, nell'annuale competizione di ILSVRC, considerata ormai dalla comunità come le "Olimpiadi della Visione Artificiale", Microsoft Research Asia ha presentato ResNet[resnet]: una CNN a 152 strati che ha abbassato l'errore sulla classificazione (1000 classi) al *solo 3.6%*. Il risultato è impressionante, dato che un essere umano più o meno abile nel riconoscere tutte le classi ha un errore di circa 5-10%, si veda [Wkarpa].

Un altro compito storicamente 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 miliare 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 su quest'ultimo 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 introdurranno dopo.

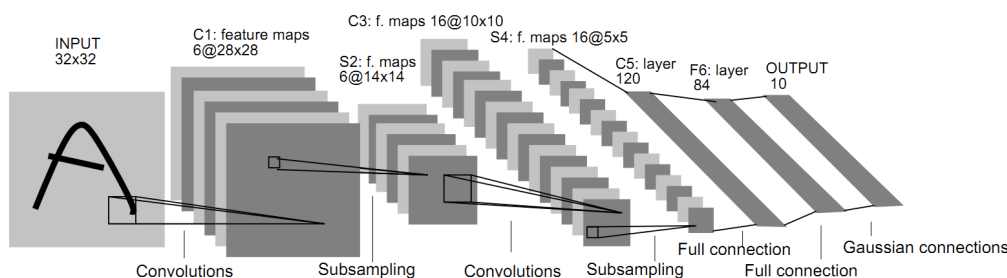


Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

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

```

1 -- define model to train
2 -- on the 10-class classification problem
3 ---
4 classes = {'airplane', 'car', 'bird', 'cat',
5           'deer', 'dog', 'frog', 'horse', 'ship', 'truck'}
6
7 normkernel = image.gaussian1D(13)

```

```

8
9 --first, we get a sequential model onto which we'll add every layer
10 model = nn.Sequential()
11 -----
12 -- convolutional network
13 -----
14 -- stage 1 : mean+std normalization -> filter bank -> squashing -> max
    pooling
15 model.add(nn.SpatialConvolutionMap(nn.tables.random(3,16,1), 5, 5))
16 model.add(nn.ReLU())
17 model.add(nn.SpatialLPPooling(16,2,2, 2, 2, 2))
18 model.add(nn.SpatialSubtractiveNormalization(16, normkernel))
19
20 -- stage 2 : filter bank -> squashing -> max pooling
21 model.add(nn.SpatialConvolutionMap(nn.tables.random(16, 256, 4), 5, 5))
22 model.add(nn.ReLU())
23 model.add(nn.SpatialLPPooling(256,2,2, 2, 2, 2))
24 model.add(nn.SpatialSubtractiveNormalization(256, normkernel))
25
26 -- stage 3 : standard 2-layer neural network
27 model.add(nn.Reshape(256*5*5))
28 model.add(nn.Linear(256*5*5, 128))
29 model.add(nn.ReLU())
30 model.add(nn.Linear(128, #classes))

```

#### 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 convenienti per i problemi di classificazione è la *negative-likelihood-log* (likelihood = funzione di verosimiglianza). La NLL necessita in ingresso di un vettore 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 NLL. In Torch questo è molto semplice:

```

1 --add a Log-SoftMax classifier at the end of the Net
2 model.add(nn.LogSoftMax())
3
4 --criterion (i.e. the loss) will be the negative-likelihood
5 criterion = nn.ClassNLLCriterion()

```

## 4.2 Procedura di training

Prima di implementare la procedura di training occorre definire con precisione la relativa terminologia:

- *Batch*: insieme di esempi che si dà in input alla rete e su cui si va a calcolare la media dell'errore tramite backpropagation. Se gli esempi non sono molto numerosi si chiama anche "mini-batch". Solitamente è a multipli di 8 (32,64,128,256). Si veda anche il Capitolo 2, sezione 2.3.

- *Pass*: un forward pass + un backward pass. In altre parole si dà qualcosa in input alla rete e si calcola l'errore tramite la backpropagation. Questo "qualcosa", può essere un solo esempio, o un mini-batch.
- *iterazioni*: un pass completo di tutti gli esempi di un batch. Nel caso di `batchSize = 1`, iterazione e pass assumono lo stesso significato.
- *Epoch*: con il termine epoch si intende un pass completo di tutti gli esempi del dataset.

Passiamo quindi alla definizione della procedura di training: il seguente snippet di codice mostra come implementare il training di una epoch, dopodiché basterà eseguirlo in loop per tutto il dataset.

```

1  -- do one epoch
2  print('<trainer> on training set:')
3  print("&<trainer> online epoch # " .. epoch .. ' [batchSize = ' ..
   opt.batchSize .. ']')
4  for t = 1, dataset:size(), opt.batchSize do
5      -- disp progress
6      xlua.progress(t, dataset:size())
7
8      -- create mini batch
9      local inputs = {}
10     local targets = {}
11     for i = t, math.min(t+opt.batchSize-1, dataset:size()) do
12         -- load new sample
13         local input = dataset.data[i]
14         local target = dataset.labels[i]
15         table.insert(inputs, input)
16         table.insert(targets, target)
17     end
18
19     -- create closure to evaluate f(X) and df/dX
20     -- i.e. forward and backward pass
21     local feval = function(x)
22         -- get new parameters
23         if x ~= parameters then
24             parameters:copy(x)
25         end
26
27         -- reset gradients
28         gradParameters:zero()
29
30         -- f is the average of all criterions
31         local f = 0
32
33         -- evaluate function for complete mini batch
34         -- #inputs == size of the mini-batch
35         for i = 1, #inputs do
36             -- estimate f
37             local output = model:forward(inputs[i])
38             local err = criterion:forward(output, targets[i])
39             f = f + err
40
41             -- estimate df/dW to minimize loss function
42             local df_do = criterion:backward(output, targets[i])
43             --backpropagation
44             model:backward(inputs[i], df_do)
45
46             -- update confusion
47             confusion:add(output, targets[i])

```

```

48
49     end
50
51     -- normalize gradients and f(X)
52     gradParameters:div(#inputs)
53     f = f/#inputs
54
55     -- return f and df/dX
56     return f,gradParameters
57 end
58
59 --configure optimizer parameters
60 config = config or {learningRate = opt.learningRate,
61                     weightDecay = opt.weightDecay,
62                     momentum = opt.momentum,
63                     learningRateDecay = 5e-7}
64 -- optimize on current mini-batch
65 --N.B. optimMethod = 'SGD' or others
66 optimMethod(feval, parameters, config)
67 end

```

### 4.3 MNIST: preprocessing

MNIST è un dataset di cifre scritte a mano, disponibile sul sito di Y. LeCun[Wlecun], costituito da un training set di 60000 esempi e un test set di 10000.

Le immagini sono B/N, tagliate in modo tale da avere la cifra esattamente al centro dell'immagine e una risoluzione di 32x32; per questo richiedono un pre-processing minimo. Basta utilizzare uno script per scaricare le immagini e caricarle nei tensori per il training ed il testing. Dopodiché si esegue uno step di *data normalization* per standardizzare i dati, sottraendo la media e dividendo per la deviazione standard.

$$Z = \frac{X - \mu}{\sigma}$$

Così si avrà un dataset con media nulla e varianza pari a 1.

### 4.4 MNIST: addestramento

Quindi, utilizzando il codice della sezione precedente e lanciando un addestramento su MNIST si avranno i risultati mostrati nella *confusion matrix* in figura 4.2.

```

<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    0   199    0    0] 99.500% [class: 6]
 [    0    0    0    0    0    0    0    0   224    0] 100.000% [class: 7]
 [    0    0    0    0    0    0    0    0    0   171] 99.419% [class: 8]
 [    2    0    0    1    0    0    0    0    0  207]] 98.571% [class: 9]
+ average row correct: 99.311608672142%
+ average rowCol correct (VOC measure): 98.63573372364%
+ global correct: 99.3%

```

Figura 4.2: Confusion Matrix MNIST: la matrice che mostra le percentuali di accuratezza ed errore sul test set per ogni classe

Come da aspettative, l'architettura divenuta popolare sulla classificazione di caratteri

raggiunge un'accuratezza vicina al 99.3% in poco meno di mezz'ora di training. Utilizzando una libreria per Lua basata su Javascript, si è scritto uno script per visualizzare alcune delle immagini in input durante il testing, con le rispettive classificazioni compiute dalla rete, in alto a sinistra (figura 4.3).

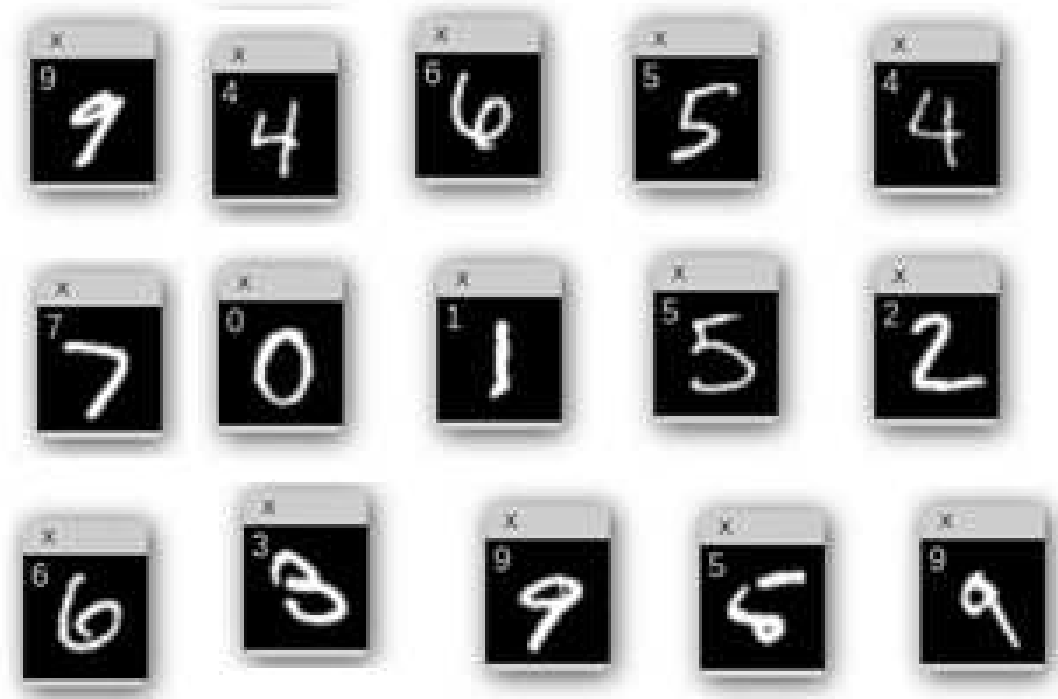


Figura 4.3: Esempi di cifre classificate correttamente

Come dimostrato, MNIST è un dataset piuttosto semplice per le CNN. Tuttavia, è rimarchevole come in poche linee di codice si possa addestrare una rete nella comprensione dei caratteri in pochissimo tempo, laddove i metodi precedenti richiedevano più sforzi e dovevano necessariamente essere adattati e rifiniti per ogni dataset diverso. All'opposto, questa stessa CNN è stata utilizzata con successo per l'addestramento sul dataset CIFAR.

## 4.5 CIFAR: preprocessing

Il dataset CIFAR10 è decisamente più impegnativo in quanto contiene immagini a colori di 10 classi di oggetti e animali del mondo reale: aeroplani, automobili, uccelli, cani, cavalli, renne, rane, navi e camion. Vi sono 60000 immagini a colori 32x32 divisi nelle suddette 10 classi, 6000 immagini per classe. Le immagini sono divise in 50000 per il training e 10000 per il testing. Più precisamente vi sono 5 batch di training e 1 di testing. Quello di testing contiene esattamente 1000 immagini per ogni classe, mentre alcuni batch di training contengono più immagini di una classe rispetto ad un'altra, ma nell'insieme contengono esattamente 5mila immagini per ogni classe. Gli oggetti di ogni classe variano molto fra di loro: ci sono diverse specie di uccello, in svariate pose diverse ed a volte parzialmente occluse e le pose si complicano ulteriormente per cani e gatti. Tutto ciò lo rende un dataset alquanto impegnativo per un sistema automatico, in particolare nella versione con 100 classi (CIFAR100). Il record è del 96% in quello da 10 classi e del 75% in quello da 100, stabilito nel

2015[Wcifar]. Qui verrà utilizzato quello da 10 classi anche per motivi di tempistiche di training.

Prima di eseguire il training, bisogna preparare il dataset. Essendo CIFAR un dataset a colori, la normalizzazione avviene in maniera appena diversa da MNIST. Il metodo più efficace consiste in 2 passi:

1. trasformare le immagini dallo spazio colore RGB a YUV per separare il canale della luminanza Y, da quelli dei colori U,V;
2. normalizzare Y localmente - i.e. per ogni esempio - e U,V globalmente;

Nei seguenti snippet di codice, si mostra come caricare il dataset e fare pre-processing.

```

1 -- load dataset
2 trainData = {
3   data = torch.Tensor(50000, 3072),
4   labels = torch.Tensor(50000),
5   size = function() return trsize end
6 }
7 for i = 0,4 do
8   subset = torch.load('cifar-10-batches-t7/data_batch_' .. (i+1) .. '.t7'
9     , 'ascii')
10   trainData.data[{ {i*10000+1, (i+1)*10000} }] = subset.data:t()
11   trainData.labels[{ {i*10000+1, (i+1)*10000} }] = subset.labels
12 end
13 --labels must go from 1 to 10
14 trainData.labels = trainData.labels + 1
15
16 -- reshape data
17 trainData.data = trainData.data:reshape(trsize,3,32,32)
18 testData.data = testData.data:reshape(tesize,3,32,32)

```

```

1 ----- preprocess/normalize train/test sets -----
2 print '<trainer> preprocessing data (color space + normalization)'
3
4 -- preprocess trainSet
5 normalization = nn.SpatialContrastiveNormalization(1, image.gaussian1D(7))
6 for i = 1,trainData:size() do
7   --rgb -> yuv
8   local rgb = trainData.data[i]
9   local yuv = image.rgb2yuv(rgb)
10
11   -- normalize Y locally:
12   yuv[1] = normalization(yuv[{{1}}])
13   trainData.data[i] = yuv
14 end
15 -- normalize U globally:
16 mean_u = trainData.data[{ {},2,{},{} }]:mean()
17 std_u = trainData.data[{ {},2,{},{} }]:std()
18 trainData.data[{ {},2,{},{} }]:add(-mean_u)
19 trainData.data[{ {},2,{},{} }]:div(-std_u)
20
21 -- normalize V globally:
22 mean_v = trainData.data[{ {},3,{},{} }]:mean()
23 std_v = trainData.data[{ {},3,{},{} }]:std()
24 trainData.data[{ {},3,{},{} }]:add(-mean_v)
25 trainData.data[{ {},3,{},{} }]:div(-std_v)
26
27 --same applies to test set...

```



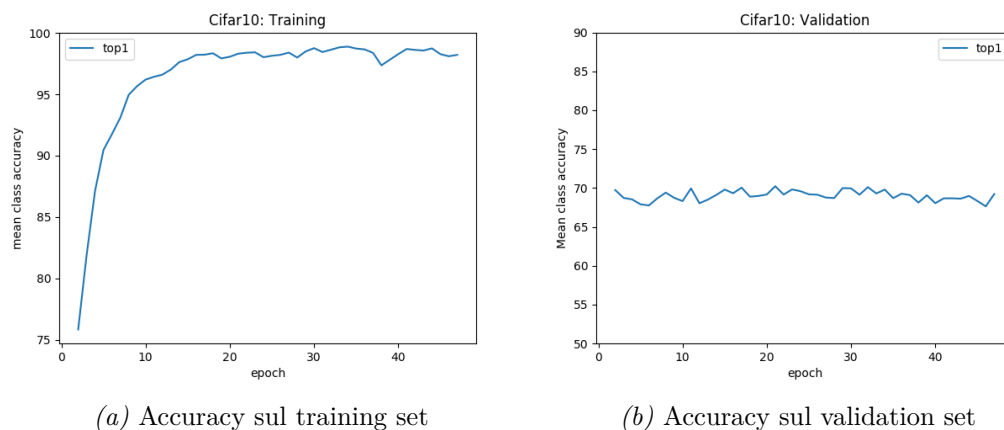


Figura 4.4: Performance della rete in training e validation

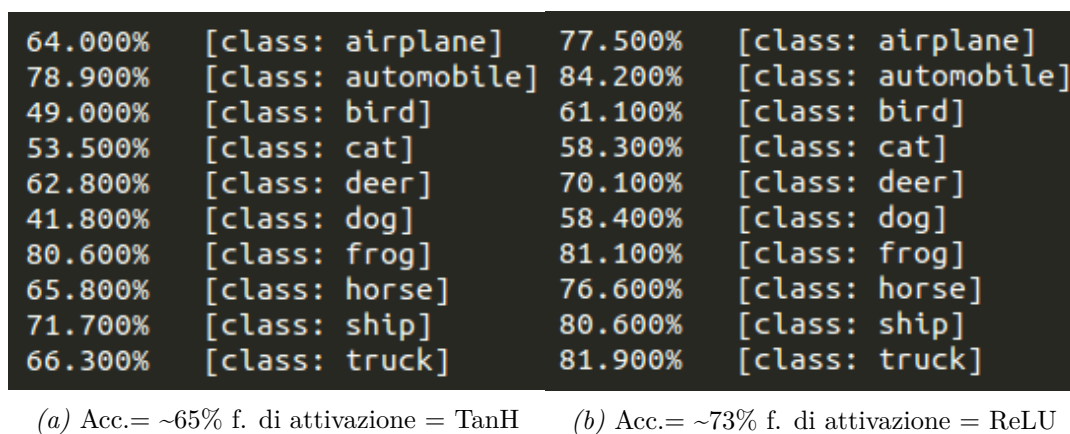


Figura 4.5: Percentuali di accuracy a seconda della funzione d'attivazione. La ReLU produce indubbiamente risultati migliori.

## 4.6 CIFAR: addestramento

Con il dataset pronto, si può procedere all'effettivo addestramento della rete e misurarne i risultati.

Dopo circa 4-5 epoche (1 ore di training) si hanno i risultati in figura 4.4. L'apprendimento è quasi ultimato e la percentuale di accuracy sul validation set si mantiene attorno al 70%. Continuando il training per 20 epoche, si arriva ad un'accuratezza sulla media delle classi del 72.7%.

Il tempo di training è alto se si usa la CPU, ma cala drasticamente se si addestrano le reti sulla GPU. Torch infatti, ha il supporto a CUDA[cuda] ed è molto semplice passare da CPU a GPU e viceversa: basta chiamare sul tensore il metodo :cuda(). Si veda un esempio in appendice B.

Sono stati effettuati diversi addestramenti per testare alcune scelte architettureali. È importante notare che cambiando solamente la funzione d'attivazione - da tangente iperbolica (tanh) a rectified linear unit - ed allenando la rete per lo stesso numero di epoch, si ha un miglioramento di circa 8 punti percentuali. Come menzionato prima nel corso di quest'elaborato infatti, la ReLU si è dimostrata più volte la funzione non lineare più efficace per l'apprendimento. In figura 4.5 il confronto fra i 2 risultati.

Come per MNIST, si sono visualizzati a titolo d'esempio alcune immagini su cui

la rete ha dato un responso corretto ed altre dove ha sbagliato. I colori strani delle immagini sono dovute al pre-processing, figura 4.6 e 4.7. In particolare, durante il training si è visto che buona parte degli errori è dovuta ad una confusione tra cani, gatti e uccelli, in figura si può vedere qualche esempio di ciò.

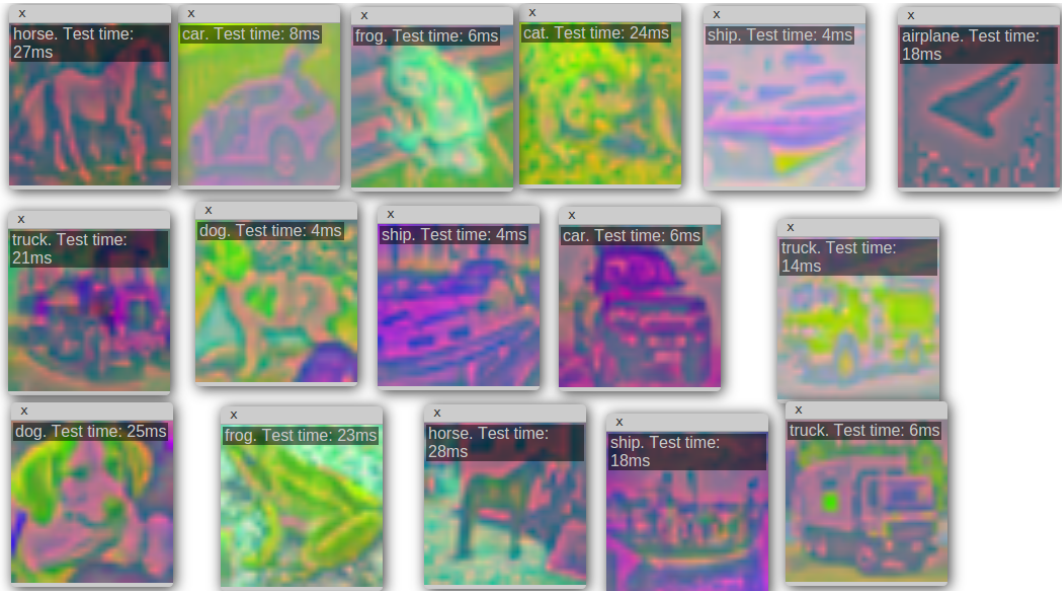


Figura 4.6: Esempi classificati correttamente

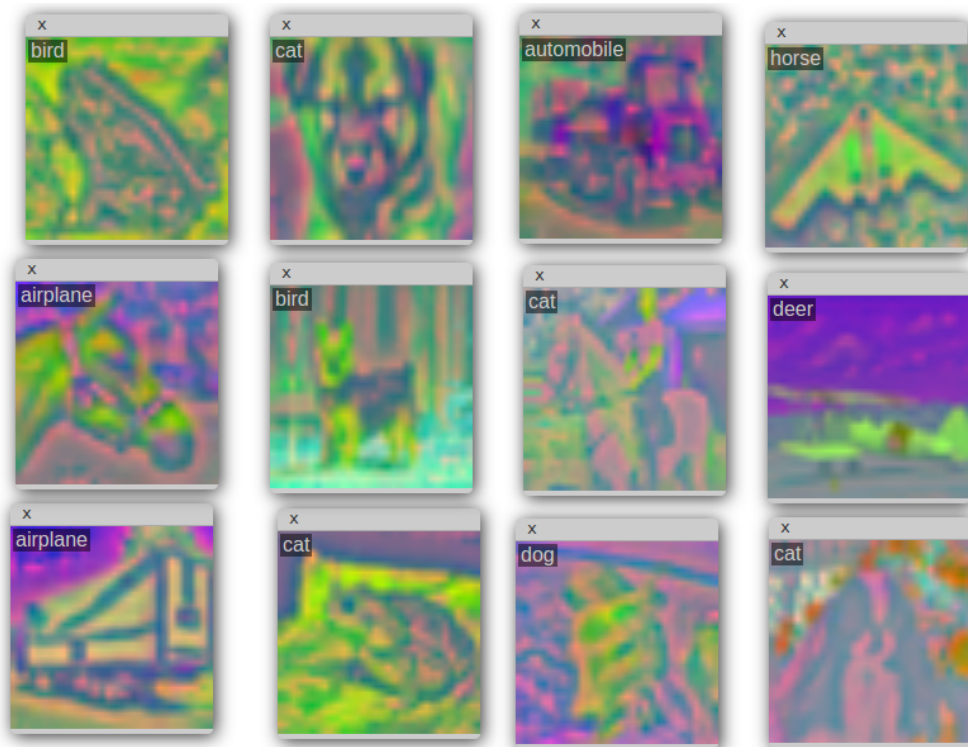


Figura 4.7: Esempi classificati erroneamente

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

## Capitolo 5

# Addestrare un modello allo stato dell'arte

### 5.1 Deep Residual Network

Menzionata nel Capitolo 3 a causa del clamore riscosso per l'incredibilmente bassa percentuale d'errore nella ImageNet Challenge del 2015, la *Deep Residual Network* - o ResNet - è una CNN *molto* profonda presentata da Microsoft Research Asia[**resnet**] che ha stravolto tutti i record in classification, detection e localization.

L'architettura è modulare: si basa su un blocco detto "*residual block*" (figura 5.1) che viene ripetuto N volte, prima dell'ultimo strato di classificazione. L'idea dietro a questo blocco viene chiamata *residual learning* e funziona così:

1. l'input  $x$  va attraverso una serie conv-relu-conv dando in output un certo  $F(x)$
2. a questo viene aggiunto l'input originale:  $H(x) = F(x) + x$

$F(x)$  è chiamato residual learning.

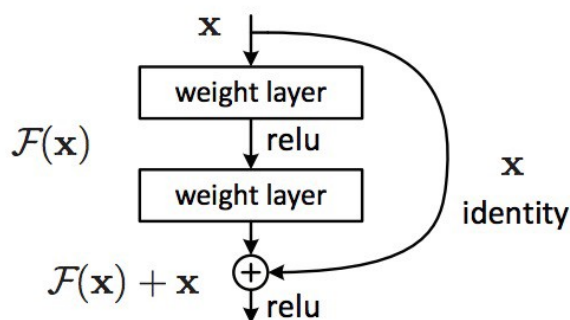


Figura 5.1: Un "Residual Block", all'input viene aggiunto  $F(x)$  che è il residual

Nelle CNN normalmente si passa direttamente da  $x$  a  $F(x)$ , la quale è una rappresentazione completamente nuova che non mantiene l'informazione dell'input originale. La novità qui, invece, consiste nel mantenere queste informazioni lungo tutta la rete; ogni blocco esegue una sorta di fine-tuning delle informazioni apprese negli strati precedenti, e secondo gli autori è più facile ottimizzare un "residual mapping" anziché un "unreferenced mapping".

Un esempio della profondità di ResNet è rappresentato in figura 5.2, dove viene confrontata la versione da 34 strati con la VGG-Network, la rete più profonda che venne presentata 2 anni prima.

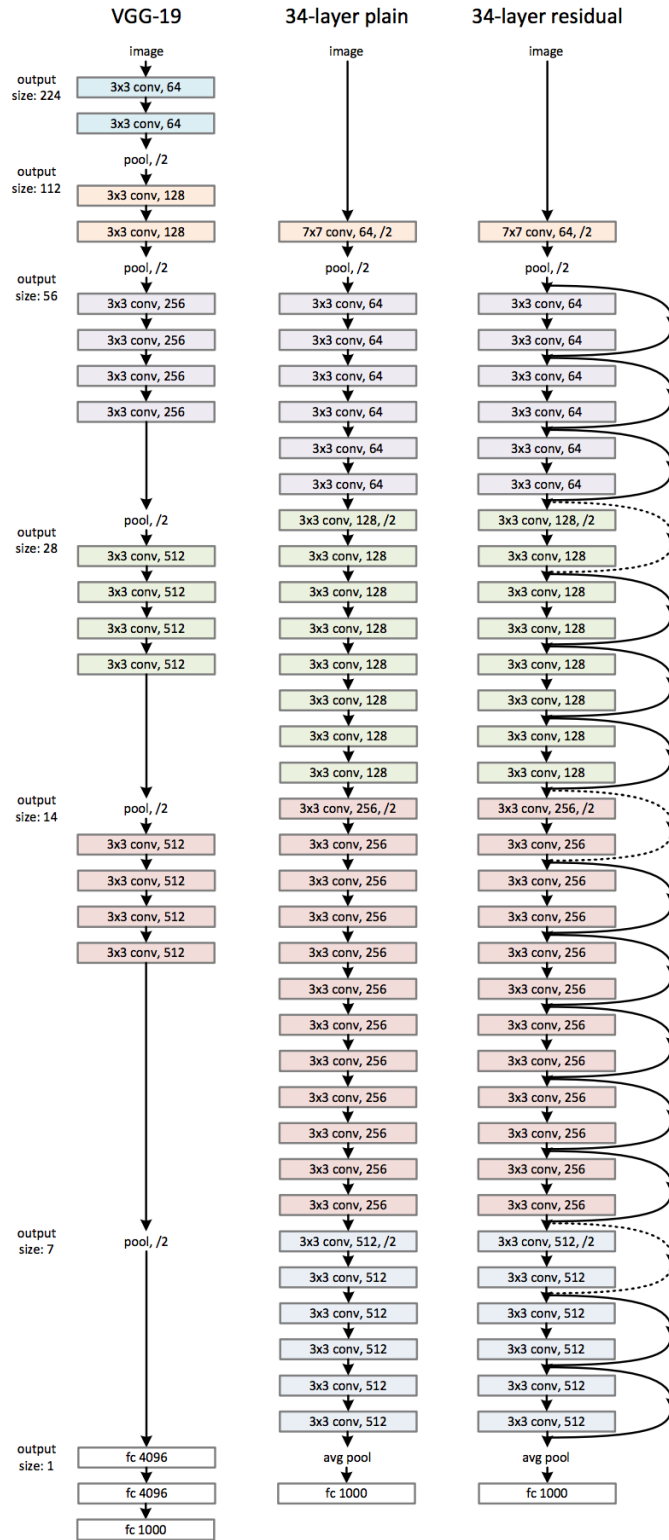


Figura 5.2: Confronto architetture: VGG vincitrice dell'ILSVRC 2013 (sinistra), rete classica a 34 strati (centro), Residual Network a 34 strati (sinistra).

Un'altra ragione per cui ResNet è stupefacente è che la backpropagation funziona meglio a causa delle somme di ogni layer che distribuiscono la derivata lungo la

rete. Inoltre, avere queste scorciatoie che non passano attraverso conv-relu aiuta contro il cruciale problema dei *vanishing gradients*. Quando si usano molti layer uno sull'altro si corre il rischio di non riuscire ad ottimizzare l'apprendimento perché l'errore calcolato con la backprop si perde tra i vari strati e dopo alcuni layer arriva un gradiente nullo che non aiuta nell'apprendimento. Per questa ragione anche, il paper di ResNet è importante: ha dimostrato come poter costruire CNN molto profonde (numero di strati  $> 100$ ) riuscendo a completare l'apprendimento ed avere inoltre risultati allo stato dell'arte.

Tuttavia, altre analisi sul successo di ResNet[**restudy**] sostengono il contrario: anche in essa il problema dei vanishing gradients si manifesta dopo 20 moduli, e quindi la profondità da sola non può essere la caratteristica principale della rete. Suggestiscono invece, che la rete si comporti come un gruppo di "sotto-reti" più piccole che contribuiscono insieme a generare la risposta corretta.

È sicuramente un argomento recente e complesso; per ulteriori dettagli si rimanda alla bibliografia.

## 5.2 Implementazione di ResNet di Facebook

L'implementazione di ResNet è decisamente complessa ma fortunatamente nella community si rilasciano spesso le implementazioni dei modelli più popolari per stimolare i ricercatori a riprodurre gli esperimenti e migliorare sempre di più lo stato dell'arte. A questo proposito Facebook ha rilasciato il codice per diversi modelli di ResNet da 18-34-50-101-152-200 strati.

Nei seguenti snippet, vi è l'implementazione della scorciatoia identità e del residual block:

```

1 -- The shortcut layer is either identity or 1x1 convolution
2 local function shortcut(nInputPlane, nOutputPlane, stride)
3     local useConv = shortcutType == 'C' or
4         (shortcutType == 'B' and nInputPlane ~= nOutputPlane)
5     if useConv then
6         -- 1x1 convolution
7         return nn.Sequential()
8             :add(Convolution(nInputPlane, nOutputPlane, 1, 1, stride,
9                 stride))
10            :add(SBatchNorm(nOutputPlane))
11    elseif nInputPlane ~= nOutputPlane then
12        -- Strided, zero-padded identity shortcut
13        return nn.Sequential()
14            :add(nn.SpatialAveragePooling(1, 1, stride, stride))
15            :add(nn.Concat(2))
16            :add(nn.Identity())
17            :add(nn.MulConstant(0))
18    else
19        return nn.Identity()
20    end
21 end
22
23 -- The basic residual layer block for 18 and 34 layer network, and the
24 -- CIFAR networks
25 local function basicblock(n, stride)
26     local nInputPlane = iChannels
27     iChannels = n
28
29     local s = nn.Sequential()
30     s:add(Convolution(nInputPlane, n, 3, 3, stride, stride, 1, 1))
31     s:add(SBatchNorm(n))

```

```
31     s:add(ReLU(true))
32     s:add(Convolution(n,n,3,3,1,1,1,1))
33     s:add(SBatchNorm(n))
34
35     return nn.Sequential()
36         :add(nn.ConcatTable()
37             :add(s)
38             :add(shortcut(nInputPlane, n, stride)))
39         :add(nn.CAddTable(true))
40         :add(ReLU(true))
41 end
```

Microsoft ha addestrato ResNet per circa 3 settimane su 8 GPU di ultima generazione. Dato l'evidente mancanza di hardware opportuno per un'"impresa" del genere si è deciso di utilizzare il modello più piccolo, quello a 18 strati. In appendice B è mostrata una rappresentazione testuale del modello, ove se ne può apprezzare la complessità e lunghezza.

### 5.3 Addestramento su CIFAR

Si è addestrato il modello ResNet-18 su CIFAR10 per circa 160 epoche. I risultati sul training e validation set sono presentati rispettivamente in figura e figura. Sull'asse X vi sono le epoch di training, sull'asse Y la percentuale di errore.

La rete da in uscita le 10 classi più probabili; con top1 s'intende quando la classe corretta è quella in uscita più probabile mentre con top5 quando la classe corretta è tra le prime 5 più probabili.

### 5.4 LeNet vs Resnet: confronto

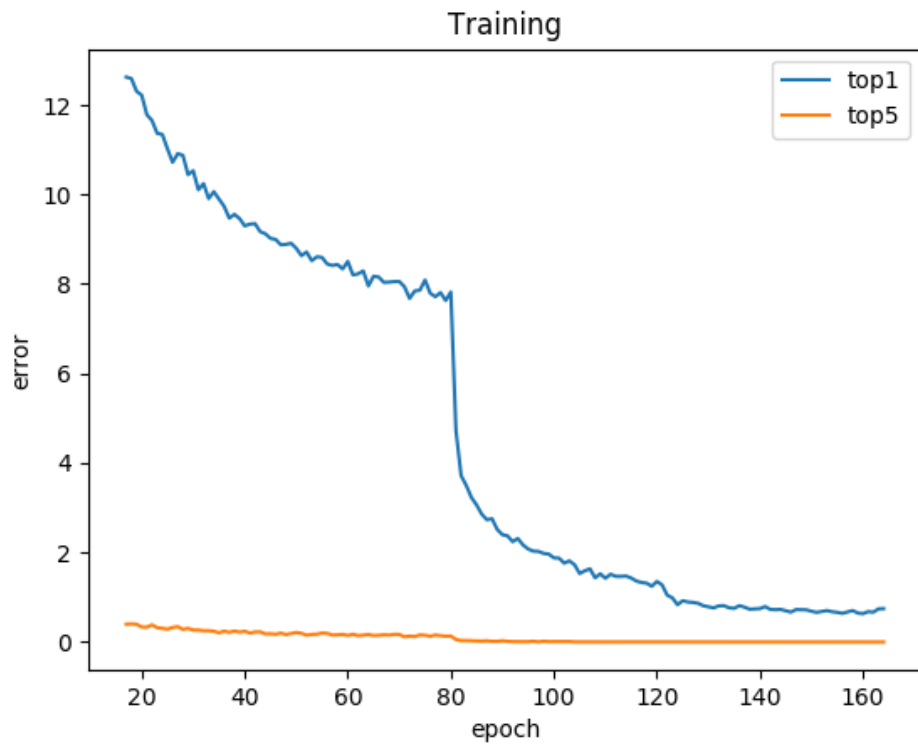


Figura 5.3: Top-1 e Top-5 training accuracy di ResNet sul dataset CIFAR10

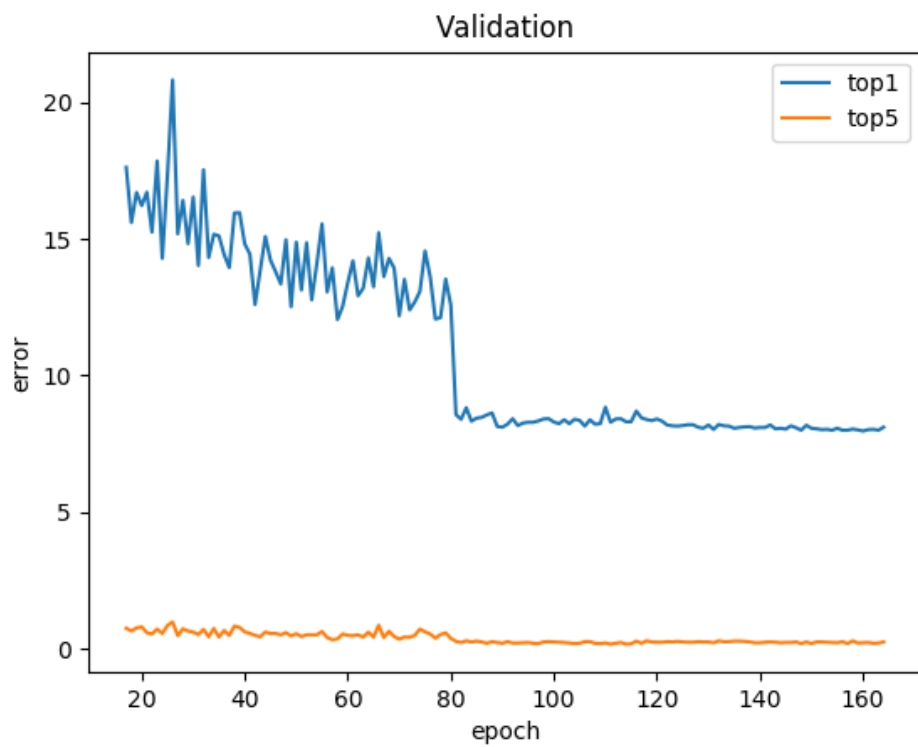


Figura 5.4: Top-1 e Top-5 validation accuracy di ResNet sul dataset CIFAR10





## 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

[Wzoo]

#### 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).

```
1 --add a Log-SoftMax classifier at the end of the Net
2 model.add(nn.LogSoftMax())
3
4 --criterion (i.e. the loss) will be the negative-likelihood
5 criterion = nn.ClassNLLCriterion()
```



## Capitolo 7

# Conclusioni



## Appendice A

# MLP: Codice aggiuntionale

### A.1 Classi in Lua

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

```

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

```

```

44     return c
45 end

```

## A.2 La classe Neural\_Network

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

```

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

```

## A.3 Metodi getter e setter

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

```

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

```





## Appendice B

# Il framework Torch

## B.1 Introduzione

[WTorch]

## B.2 Utilizzo base per reti neurali

### B.2.1 Supporto CUDA

## B.3 ResNet

Una rappresentazione testuale del modello a 18 strati di Residual Network.

```

1 nn.Sequential {
2   [input -> (1) -> (2) -> (3) -> (4) -> (5) -> (6) -> (7) -> (8) -> (9) ->
   output]
3   (1): cudnn.SpatialConvolution(3 -> 16, 3x3, 1,1, 1,1) without bias
4   (2): nn.SpatialBatchNormalization (4D) (16)
5   (3): cudnn.ReLU
6   (4): nn.Sequential {
7     [input -> (1) -> (2) -> (3) -> output]
8     (1): nn.Sequential {
9       [input -> (1) -> (2) -> (3) -> output]
10      (1): nn.ConcatTable {
11        input
12        | '-> (1): nn.Sequential {
13        |   [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
14        |   (1): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
15        |   without bias
16        |   (2): nn.SpatialBatchNormalization (4D) (16)
17        |   (3): cudnn.ReLU
18        |   (4): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
19        |   without bias
20        |   (5): nn.SpatialBatchNormalization (4D) (16)
21        |   }
22        |   '-> (2): nn.Identity
23        |   ... -> output
24      }
25      (2): nn.CAddTable
26      (3): cudnn.ReLU
27    }
28    (2): nn.Sequential {
29      [input -> (1) -> (2) -> (3) -> output]
30      (1): nn.ConcatTable {
31        input
32        | '-> (1): nn.Sequential {
33        |   [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
34        |   (1): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
35        |   without bias

```

```

33         |         (2): nn.SpatialBatchNormalization (4D) (16)
34         |         (3): cudnn.ReLU
35         |         (4): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
    without bias
36         |         (5): nn.SpatialBatchNormalization (4D) (16)
37         |         }
38         |         \-> (2): nn.Identity
39         |         ... -> output
40     }
41     (2): nn.CAddTable
42     (3): cudnn.ReLU
43 }
44 (3): nn.Sequential {
45     [input -> (1) -> (2) -> (3) -> output]
46     (1): nn.ConcatTable {
47         input
48         | \-> (1): nn.Sequential {
49         |     [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
50         |     (1): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
    without bias
51         |     (2): nn.SpatialBatchNormalization (4D) (16)
52         |     (3): cudnn.ReLU
53         |     (4): cudnn.SpatialConvolution(16 -> 16, 3x3, 1,1, 1,1)
    without bias
54         |     (5): nn.SpatialBatchNormalization (4D) (16)
55         |     }
56         |     \-> (2): nn.Identity
57         |     ... -> output
58     }
59     (2): nn.CAddTable
60     (3): cudnn.ReLU
61 }
62 }
63 (5): nn.Sequential {
64     [input -> (1) -> (2) -> (3) -> output]
65     (1): nn.Sequential {
66         [input -> (1) -> (2) -> (3) -> output]
67         (1): nn.ConcatTable {
68             input
69             | \-> (1): nn.Sequential {
70             |     [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
71             |     (1): cudnn.SpatialConvolution(16 -> 32, 3x3, 2,2, 1,1)
    without bias
72             |     (2): nn.SpatialBatchNormalization (4D) (32)
73             |     (3): cudnn.ReLU
74             |     (4): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
    without bias
75             |     (5): nn.SpatialBatchNormalization (4D) (32)
76             |     }
77             |     \-> (2): nn.Sequential {
78             |         [input -> (1) -> (2) -> output]
79             |         (1): nn.SpatialAveragePooling(1x1, 2,2)
80             |         (2): nn.Concat {
81                 input
82                 | \-> (1): nn.Identity
83                 |     \-> (2): nn.MulConstant
84                 |     ... -> output
85             }
86         }
87         ... -> output
88     }
89     (2): nn.CAddTable
90     (3): cudnn.ReLU

```

```

91     }
92     (2): nn.Sequential {
93       [input -> (1) -> (2) -> (3) -> output]
94       (1): nn.ConcatTable {
95         input
96         | '-> (1): nn.Sequential {
97         |       [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
98         |       (1): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
99         without bias
100        |       (2): nn.SpatialBatchNormalization (4D) (32)
101        |       (3): cudnn.ReLU
102        |       (4): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
103        without bias
104        |       (5): nn.SpatialBatchNormalization (4D) (32)
105        |       }
106        '-> (2): nn.Identity
107        ... -> output
108      }
109      (2): nn.CAddTable
110      (3): cudnn.ReLU
111    }
112    (3): nn.Sequential {
113      [input -> (1) -> (2) -> (3) -> output]
114      (1): nn.ConcatTable {
115        input
116        | '-> (1): nn.Sequential {
117        |       [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
118        |       (1): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
119        without bias
120        |       (2): nn.SpatialBatchNormalization (4D) (32)
121        |       (3): cudnn.ReLU
122        |       (4): cudnn.SpatialConvolution(32 -> 32, 3x3, 1,1, 1,1)
123        without bias
124        |       (5): nn.SpatialBatchNormalization (4D) (32)
125        |       }
126        '-> (2): nn.Identity
127        ... -> output
128      }
129      (2): nn.CAddTable
130      (3): cudnn.ReLU
131    }
132    (6): nn.Sequential {
133      [input -> (1) -> (2) -> (3) -> output]
134      (1): nn.Sequential {
135        [input -> (1) -> (2) -> (3) -> output]
136        (1): nn.ConcatTable {
137          input
138          | '-> (1): nn.Sequential {
139          |       [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
140          |       (1): cudnn.SpatialConvolution(32 -> 64, 3x3, 2,2, 1,1)
141          without bias
142          |       (2): nn.SpatialBatchNormalization (4D) (64)
143          |       (3): cudnn.ReLU
144          |       (4): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
145          without bias
146          |       (5): nn.SpatialBatchNormalization (4D) (64)
147          |       }
148          '-> (2): nn.Sequential {
149            [input -> (1) -> (2) -> output]
150            (1): nn.SpatialAveragePooling(1x1, 2,2)
151            (2): nn.Concat {
152              input

```

```

148         | '-> (1): nn.Identity
149         '-> (2): nn.MulConstant
150         ... -> output
151     }
152 }
153     ... -> output
154 }
155     (2): nn.CAddTable
156     (3): cudnn.ReLU
157 }
158 (2): nn.Sequential {
159     [input -> (1) -> (2) -> (3) -> output]
160     (1): nn.ConcatTable {
161         input
162         | '-> (1): nn.Sequential {
163         |     [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
164         |     (1): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
without bias
165         |     (2): nn.SpatialBatchNormalization (4D) (64)
166         |     (3): cudnn.ReLU
167         |     (4): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
without bias
168         |     (5): nn.SpatialBatchNormalization (4D) (64)
169         |     }
170         '-> (2): nn.Identity
171         ... -> output
172     }
173     (2): nn.CAddTable
174     (3): cudnn.ReLU
175 }
176 (3): nn.Sequential {
177     [input -> (1) -> (2) -> (3) -> output]
178     (1): nn.ConcatTable {
179         input
180         | '-> (1): nn.Sequential {
181         |     [input -> (1) -> (2) -> (3) -> (4) -> (5) -> output]
182         |     (1): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
without bias
183         |     (2): nn.SpatialBatchNormalization (4D) (64)
184         |     (3): cudnn.ReLU
185         |     (4): cudnn.SpatialConvolution(64 -> 64, 3x3, 1,1, 1,1)
without bias
186         |     (5): nn.SpatialBatchNormalization (4D) (64)
187         |     }
188         '-> (2): nn.Identity
189         ... -> output
190     }
191     (2): nn.CAddTable
192     (3): cudnn.ReLU
193 }
194 }
195 (7): cudnn.SpatialAveragePooling(8x8, 1,1)
196 (8): nn.View(64)
197 (9): nn.Linear(64 -> 10)
198 }

```