Università
degli Studi
di Ferrara

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA
ED INFORMATICA

ANALISI DELL'IMPATTO DI ERRORI HARDWARE SULL'ACCURATEZZA DI UNA RETE NEURALE SEMPLICE

Relatore:
Michele Favalli

Laureando:
Filippo Landi

ANNO ACCADEMICO 2018-2019

Sommario

In questa tesi si studia l'architettura di una rete neurale e di come errori hardware possano modificarne il comportamento. Inizialmente si analizza la teoria che sta alla base delle reti neurali. Successivamente implementiamo la rete neurale usando Python. La tipologia di rete implementata è una rete neurale in avanti e completamente connessa. Essa ha un solo strato nascosto (con numero ristretto di neuroni) e per questo la rete neurale viene definita 'semplice'. Essa viene allenata per il riconoscimento di numeri scritti a mano. In conclusione si eseguono alcuni test per analizzare statisticamente l'impatto di errori hardware sulla rete.

Indice

1 Capire le reti neurali artificiali **3**
1.1 L'architettura delle reti neurali artificiali 3
1.1.1 I neuroni artificiali 3
1.1.2 Collegare più neuroni insieme, le reti neurali 10
1.2 L'apprendimento nelle reti neurali artificiali 13
1.2.1 Il concetto di apprendimento 13
1.2.2 La retropropagazione dell'errore 15

2 Implementazione del codice in Python **27**
2.1 Il codice iniziale 27
2.1.1 Il codice di Michael Nielsen 27
2.1.2 Una rete neurale semplice per il riconoscimento di numeri scritti a mano 28
2.2 Il codice che ho scritto per implementare nuove funzionalità . 31
2.2.1 Il salvataggio dello stato allenato della rete 31
2.2.2 L'implementazione degli errori 32
2.2.3 Uno script flessibile per eseguire il codice 33

3 Analisi dell'impatto di errori sui pesi nella rete neurale **37**
3.1 Come ho eseguito i test e i risultati di essi 37
3.2 Conclusioni 43

A Informazioni utili sul codice e sui programmi da usare **45**

B Approfondimento sugli errori della rete neurale **47**

Introduzione

L'*apprendimento automatico*, o in inglese *machine learning*, è una branca dell'*intelligenza artificiale* in grande crescita negli ultimi anni. Le *reti neurali artificiali* sono un modello per l'apprendimento automatico e probabilmente il più popolare al giorno d'oggi. Le reti neurali sono sempre più presenti nella nostra vita quotidiana. Le troviamo in vari dispositivi digitali anche se non ne siamo completamente consapevoli. Per avere una idea di dove trovarle, basta pensare ai sempre più presenti assistenti virtuali: Siri, Alexa, Google Assistant, Cortana. Oppure al riconoscimento facciale di Facebook con *DeepFace*.

La scelta dell'argomento quindi è stata dettata dalla sua 'freschezza' e anche da una mia sfida personale, in quanto io non mi sono mai interessato all'argomento (in passato). Per questo motivo, l'obiettivo principale è stato imparare come funzioni una rete neurale artificiale. A questo scopo, durante le mie ricerche insieme a vari tutorial a sé stanti, ho trovato un libro online scritto da Michael Nielsen che struttura l'analisi dell'argomento molto bene e con diversi approfondimenti. Il primo capitolo della tesi affronta proprio quello che ho imparato da questa mia ricerca sulla architettura e il funzionamento delle reti neurali.

In allegato alla teoria affrontata nel suo libro, Nielsen ha scritto del codice piuttosto semplice, che si basa su di essa, e implementa una rete neurale per il riconoscimento di numeri scritti a mano. Questo è il codice da cui sono partito espandendolo poi con ulteriori funzioni. Il codice è scritto in Python e questa è stata un'altra sfida per me, non avendo mai scritto in questo linguaggio. In seguito a questa prima ricerca le reti neurali si sono dimostrate strutture piuttosto complesse: hanno svariati parametri che ne decidono il comportamento. L'idea, che caratterizza questa tesi, è quella di sfruttare l'implementazione di una rete neurale per simulare un dispositivo hardware che realizzi tale architettura. In questo modo si può osservare come errori iniettati all'interno di tale architettura possano impattarne il comportamento. Nel secondo capitolo affronteremo l'implementazione della rete in Python e l'implementazione del mio codice al fine di poter effettuare la simulazione di errori hardware.

L'implementazione di errori è una cosa interessante poiché, alla luce della struttura complessa delle reti neurali, si potrebbe pensare che generalmente

variare qualche parametro su tantissimi, non sia un grosso problema. Nei test del terzo e conclusivo capitolo, affronteremo il problema di errori iniettati sui, così detti, pesi della rete neurale. Vedremo come in media, bisognerà iniettare un numero considerevole di errori per impattare l'accuratezza della rete. Concludendo la tesi, noteremo quali sono i posti 'patologici' nella rete, dove anche pochi errori possono impattare la sua accuratezza in maniera decisa.

Visto l'argomento informatico, molti termini sarebbero in inglese. Ho fatto del mio meglio per tradurre più termini possibile in italiano (indicando nelle note a piè di pagina i corrispettivi termini inglesi). Alcuni termini però sono rimasti come nella letteratura inglese poiché non ho trovato una traduzione soddisfacente.

Alcune immagini e parte del codice che utilizzo sono parte del libro di Nielsen ma, come tutto il materiale del suo libro, sono utilizzabili liberamente (eccetto la vendita).

Capitolo 1

Capire le reti neurali artificiali

Per mostrare i concetti basilari delle reti neurali, divido questo capitolo in due parti:

- L'architettura delle reti neurali artificiali.
- L'apprendimento nelle reti neurali artificiali.

1.1 L'architettura delle reti neurali artificiali

Una rete neurale artificiale è un modello computazionale composto da *neuroni artificiali*, *collegati in maniera opportuna fra loro*, che si ispira vagamente alle reti neurali biologiche.[1]

Segue dalla definizione che i passi da compiere per capire le reti neurali saranno:

1. Capire cos'è e come funziona un neurone artificiale.
2. Capire come vengano collegati fra loro più neuroni.

1.1.1 I neuroni artificiali

Nel tempo, per ragioni che capiremo in seguito, sono stati sviluppati vari modelli di neurone artificiale. Partiamo da uno dei primi modelli che risulta anche semplice nella sua analisi: il *percettrone*.

Il percettrone

Questo modello è stato sviluppato negli anni 50' da Frank Rosenblatt.[2] Il modo in cui funziona è abbastanza semplice, infatti esso prende più ingressi[1] x_1, x_2, ..., e produce una singola uscita[2] di valore binario:

[1] In inglese *input*, di uso comune anche in italiano.
[2] In inglese *output*, di uso comune anche in italiano.

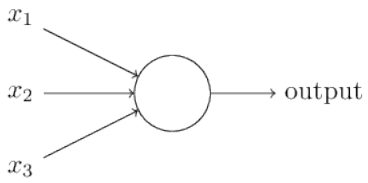

Figura 1.1: Un percettrone con tre ingressi.

Il percettrone in figura 1.1[3] ha tre ingressi x_1, x_2, x_3. In generale può avere più o meno ingressi.

Al fine di calcolare l'uscita, prima di tutto si introducono dei *pesi*[4] w_1, w_2,..., numeri reali che rappresentano in che misura gli ingressi la influenzino. Quindi l'uscita del neurone assume valore 0 oppure 1 a seconda se la somma pesata degli ingressi superi o meno una determinata *soglia*[5].

La soglia, come i pesi, è un numero reale ed è un parametro del neurone.

Definiamo il modello dell'uscita in maniera algebrica:

$$\text{uscita} \quad = \quad \begin{cases} 0 & \text{se } \sum_j w_j x_j \leq \text{ soglia} \\ 1 & \text{se } \sum_j w_j x_j > \text{ soglia} \end{cases} \tag{1.1}$$

Facciamo un esempio semplice per fissare le idee: pensiamo ad un percettrone capace di dirci se, in seguito a determinati fattori, dovremmo essere capaci di passare un esame (uscita a 1) oppure no (uscita a 0). Consideriamo di avere solo tre fattori fondamentali da considerare:

1. Il soddisfacimento delle propedeuticità.
2. Quante lezioni abbiamo seguito.
3. Quanto tempo abbiamo studiato.

Pensiamo a questi fattori come i tre ingressi x_1, x_2, x_3, della figura 1.1.

Ora supponiamo di poter fissare dei pesi che indicano l'incidenza sull'uscita di ogni ingresso: abbiamo un peso w_1 per le propedeuticità, un peso w_2 per quante lezioni abbiamo seguito e un peso w_3 per quanto abbiamo studiato. Supponiamo che i pesi siano tutti uguali e con valore unitario 1.[6] Supponiamo che ogni fattore sia un valore compreso tra 0 e 1, a significare una percentuale di quanto abbiamo soddisfatto quel fattore.

Faccio un esempio:

- Se non soddisfiamo alcuna propedeuticità, abbiamo valore 0 in quell' ingresso.

[3] Generalmente i neuroni vengono rappresentati come dei cerchi.
[4] In inglese *weights*.
[5] In inglese *threshold*.
[6] Questo significa che ogni ingresso ha lo stesso impatto sulla uscita.

- Se soddisfiamo tutte le propedeuticità, abbiamo valore 1 in quel ingresso.
- Se non le soddisfiamo appieno, abbiamo un valore tra 0 e 1 che indica in quale ipotetica percentuale le soddisfiamo.

Lo stesso vale per gli altri due fattori.

Soddisfando del tutto i requisiti, avremmo $\sum_j w_j x_j = w_1 x_1 + w_2 x_2 + w_3 x_3 = 3$, cioè la *somma pesata* dei tre ingressi equivale a 3. Rapportando questo 3 a una votazione in trentesimi (non considerando per semplicità la lode), possiamo per induzione fissare la ipotetica soglia di successo a 1,8 (rappresentante il 18, cioè la sufficienza).

Possiamo provare vari valori di ingresso, moltiplicarli per i rispettivi pesi, sommarli e vedere se la somma supera la soglia: se questo avviene allora il percettrone dirà che dovremmo essere in grado di passare l'esame, in caso contrario dirà che non dovremmo essere in grado di farlo.

Il modello di uscita che sfrutta la soglia non è comunemente usato nell'ambito delle reti neurali: applicando qualche modifica alla 1.1, possiamo ricavarci il modello più comune.

La prima modifica consiste nello scrivere $\sum_j w_j x_j$ come un prodotto scalare, dove appunto w e x sono vettori[7] le cui componenti sono rispettivamente i pesi e gli ingressi.

La seconda modifica è quella di muovere la soglia dall'altra parte della disuguaglianza e sostituirla con quello che viene chiamato *bias*[8] : $b = -soglia$.

Usando il bias invece della soglia, il modello per definire l'uscita del percettrone diventa:

$$\text{uscita} = \begin{cases} 0 & \text{se } w \cdot x + b \leq 0 \\ 1 & \text{se } w \cdot x + b > 0 \end{cases} \tag{1.2}$$

Il bias può essere visto come una misura di quanto sia facile avere l'uscita ad 1: un percettrone con un bias molto grande (molto positivo) facilmente l'avrà, un percettrone con un bias molto piccolo (molto negativo) difficilmente avrà uscita a 1.

Alla fine dei conti, l'introduzione del bias cambia marginalmente il modello, però si è notato che questo cambiamento generalmente semplifica la notazione.

È interessante notare che i percettroni possano implementare funzioni logiche elementari come `AND`, `OR` e `NAND`.[3]

Per esempio, supponiamo di avere un percettrone con due ingressi, ognuno con peso -2, un bias di 3:

[7] Indico i vettori senza notazioni particolari, non uso grassetto o 'frecce'.

[8] *Bias* è un termine inglese, non ho trovato una traduzione consona, quindi lo chiamerò sempre così.

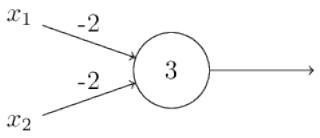

Figura 1.2: Implementazione con un percettrone di una porta `NAND`.

- Se in ingresso abbiamo $x = (x_1, x_2) = (0,0)$ in uscita abbiamo il valore 1, poiché $(-2) * 0 + (-2) * 0 + 3 = 3$ è positivo.[9]
- Se in ingresso abbiamo (0,1) e (1,0) in uscita abbiamo ancora il valore 1, poiché $(-2) * 0 + (-2) * 1 + 3 = 1$ è positivo.[10]
- Se in ingresso abbiamo (1,1) in uscita questa volta abbiamo 0, poiché $(-2) * 1 + (-2) * 1 + 3 = -1$ è negativo.

Quindi il percettrone rappresentato in figura 1.2 sta implementando una porta `NAND`.

Purtroppo il percettrone è capace di rappresentare solo funzioni logiche *linearmente divisibili* e quindi non può implementare, per esempio, la porta `XOR`.[4][11]

Proseguiamo la trattazione introducendo un altro modello comune di neurone artificiale.

Il neurone sigmoidale

Il neurone *sigmoidale* si rappresenta in maniera identica al percettrone:

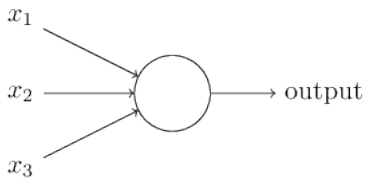

Figura 1.3: Neurone sigmoidale.

Come in un percettrone si mantengono i pesi per ogni ingresso w_1, w_2, ..., e il bias b. La differenza tra i due modelli sta nel fatto che un neurone sigmoidale permette di avere un valore compreso tra 0 e 1 in uscita, a differenza di un percettrone. Quindi per esempio '$0,432$' è un valore di uscita valido per un neurone sigmoidale.

[9] '$*$' è il simbolo che uso per le moltiplicazioni normali.

[10] Ho fatto il conto per (0,1), è uguale per (1,0).

[11] Questo limite si può sorpassare implementando una rete neurale composta da percettroni, ne tratterò nella sezione dedicata alle reti neurali.

Raggiungiamo questo risultato cambiando come viene computata l'uscita: invece di essere semplicemente 0 o 1, a seconda se la somma pesata più il bias '$w \cdot x + b$' supera un determinato valore di soglia, adesso abbiamo un valore dato da $\sigma(w \cdot x + b)$, dove $\sigma(z)$ è la *funzione sigmoidea*[12], ed è definita da:

$$\sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{1.3}$$

Rendendo più esplicita la dipendenza dagli ingressi x_1, x_2, ..., dai pesi w_1, w_2, ..., e il bias b, si ha:

$$\sigma(w \cdot x + b) \equiv \frac{1}{1 + \exp(-(w \cdot x + b))} \tag{1.4}$$

Prima ho usato z come semplice variabile, ha senso interpretarla come la variabile che rappresenta *la somma pesata degli ingressi più il bias* o il prodotto scalare tra il vettore di ingresso e il vettore dei pesi più il bias[13], cioè:

- Somma tra ingressi moltiplicati per i pesi, più il bias: $z = \sum_j w_j x_j + b$.
- Prodotto scalare tra gli ingressi e i pesi, più il bias: $z = w \cdot x + b$.

Posso farvi notare che si può far degenerare il neurone sigmoidale in un percettrone.

Osserviamo questi limiti: $\lim_{z \to \infty} \frac{1}{1+e^{-z}} = 1$ e $\lim_{z \to -\infty} \frac{1}{1+e^{-z}} = 0$.

Se quindi all'interno della funzione sigmoidea facciamo tendere z a $+\infty$ otteniamo come risultato 1, se invece facciamo tendere z a $-\infty$ otteniamo come risultato della funzione 0. Quindi con z molto grandi o piccoli il neurone sigmoidale tende a mimare un percettrone (uscita praticamente a 1 o 0 rispettivamente).

Rivediamo ulteriormente la situazione, però sotto un'ottica grafica.

Osserviamo in figura 1.4 il grafico della funzione sigmoidea[14], di cui abbiamo già visto la forma algebrica:

Osserviamo in figura 1.5 una semplice funzione che si incontra spesso in ambienti scientifici: la *funzione gradino*.[15]

$$\text{gradino(z)} = \begin{cases} 0 & \text{se } z \leq 0 \\ 1 & \text{se } z > 0 \end{cases} \tag{1.5}$$

[12] La $\sigma(z)$ detta *funzione sigmoidea* o *funzione sigmoide*, si può trovare in alcuni testi come *funzione logistica*, in realtà è un caso speciale di tale funzione. Comunque in seguito a questo, il neurone sigmoidale può essere chiamato anche *neurone logistico*.

[13] Spero che sia ovvio che rapprensentano lo stesso risultato.

[14] In inglese *sigmoid function*.

[15] In inglese *step function*. Ci sono diverse convenzioni per il valore nello 0 di questa funzione, ho scelto quella che si adatta meglio al mio caso.

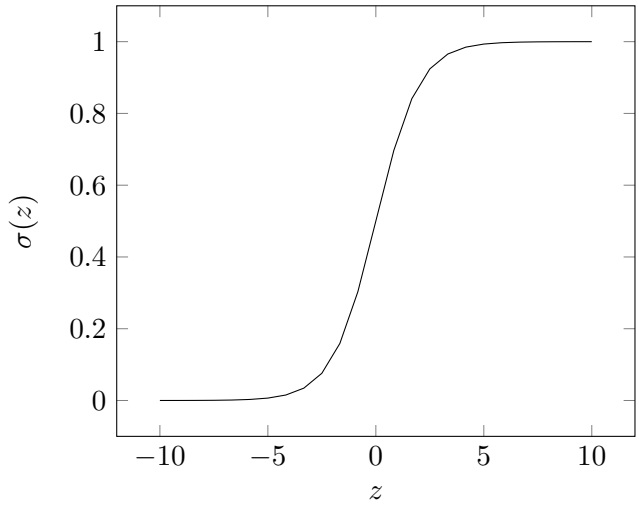

Figura 1.4: Funzione sigmoidea.

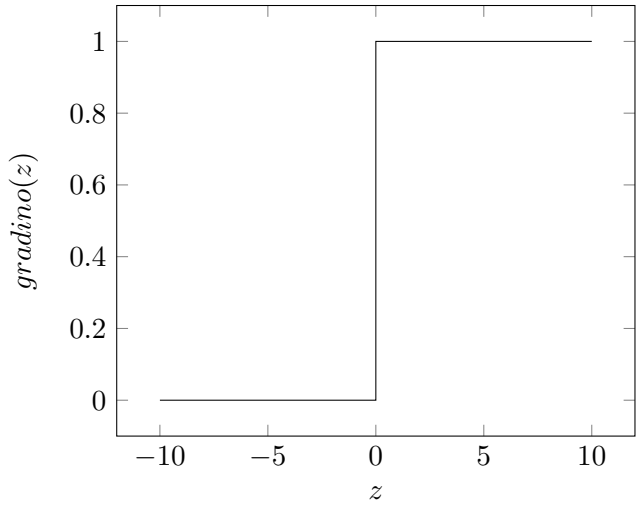

Figura 1.5: Funzione gradino.

Come da figura 1.6, possiamo vedere la funzione sigmoidea come una funzione gradino 'allungata', dove il passaggio da 0 ad 1 avviene in maniera graduale.

Come detto, z inserito nella funzione sigmoidea ci dà l'uscita del neurone sigmoidale. Inserendo lo stesso z nella funzione gradino abbiamo la risposta del percettrone. Da questo possiamo intuire una generalizzazione dove abbiamo una funzione $f(z)$ che prende in ingresso la z e restituisce un valore che sarà l'uscita del neurone. Questa funzione si chiama *funzione di*

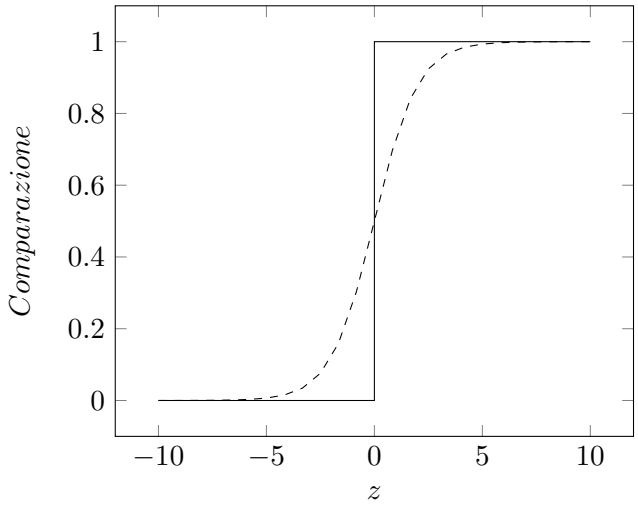

Figura 1.6: Comparazione tra sigmoide e gradino.

attivazione[16]. Variando questa funzione andiamo a variare direttamente il modello di neurone che stiamo utilizzando.

L'uscita di un neurone viene quindi chiamata in generale *attivazione*, a:

$$a = f(w \cdot x + b) \tag{1.6}$$

Vediamo un altro modello di neurone molto utilizzato di recente.

Unità lineare rettificata

La *funzione rampa* (rappresentata in figura 1.7) è un'altra funzione di attivazione divenuta importantissima di recente. Essa dà origine al neurone detto *unità lineare rettificata*[17].

$$rampa(z) = z^{+} = \max(0, z) \tag{1.7}$$

Il perché siano nati più modelli è legato al concetto chiave delle reti neurali: *l'apprendimento*.

Poichè il concetto di apprendimento si applica sia al singolo neurone che alle 'strutturate' reti neurali, introduco nella prossima sezione le reti neurali. L'apprendimento sarà trattato nella seconda parte del capitolo (come avevo già menzionato a inizio capitolo).

[16] In inglese *activation function*.
[17] In inglese *rectified linear unit*, abbreviato: *ReLU*.

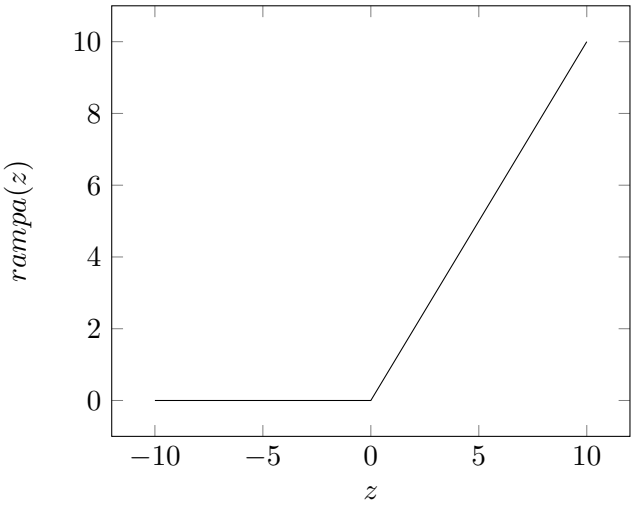

Figura 1.7: Funzione rampa.

1.1.2 Collegare più neuroni insieme, le reti neurali

Vediamo una rete neurale dove, per generalizzare l'analisi, non specifichiamo il modello di neurone utilizzato:

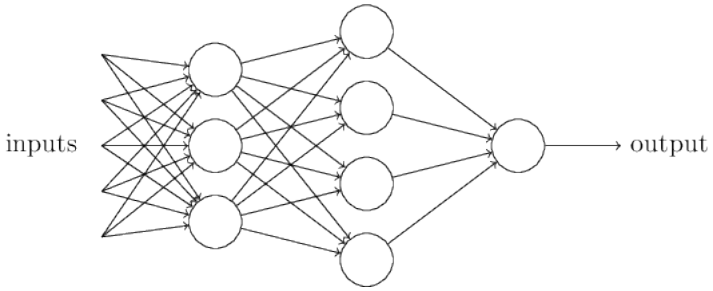

Figura 1.8: Rete neurale.

Osserviamo la figura 1.8: la prima colonna di neuroni, che chiamiamo per ora primo *strato*[18], calcola tre attivazioni, usando quindi gli ingressi e i rispettivi bias.[19] I quattro neuroni nel secondo strato calcolano quattro attivazioni assumendo come ingresso le tre attivazioni dello strato precedente. Possiamo dire che un neurone nel secondo strato fa calcoli a un livello più complesso e astratto rispetto a quelli nel primo strato (arriva a nuove conclusioni usando quelle precedenti). Per induzione, aggiungendo strati, si possono costruire reti sempre più complesse e capaci di diversi livelli di astrazione.

[18] In inglese *layer*.

[19] Faccio notare, se non fosse chiaro, che ogni neurone ha il suo bias.

Faccio notare che i neuroni sono sempre stati presentati con una singola uscita, però nell'immagine 1.8 sembrano averne di più. Questa è una scelta di stile per avere una immagine più chiara. Essenzialmente ogni neurone ha di fatto una sola uscita (che è la sua attivazione), però ogni neurone dello strato successivo la peserà in maniera diversa. Sarebbe più corretto fare una uscita e poi ramificarla successivamente nei vari neuroni ma una tale rappresentazione non sarebbe molto gradevole alla vista.

Inoltre, nelle rappresentazioni più comuni, anche gli ingressi vengono rappresentati come neuroni come vediamo in figura 1.9.

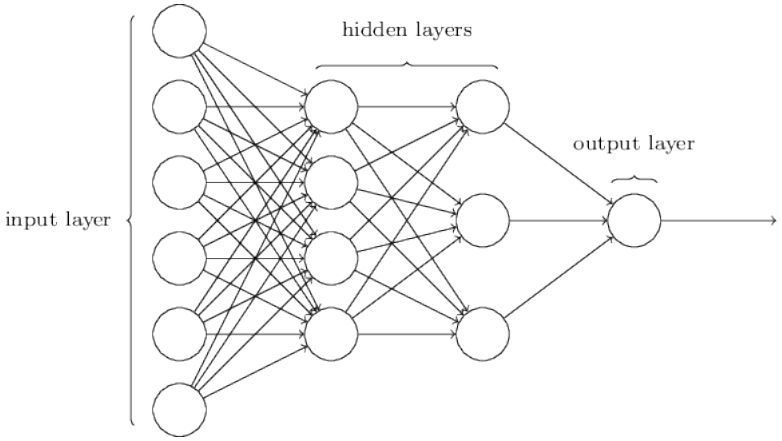

Figura 1.9: Rappresentazione classica di una rete neurale.

Il che è più ordinato da vedere ma potrebbe suggerire una idea sbagliata: gli ingressi sono gli ingressi stessi, non applicano la funzione di attivazione, semplicemente mettono in uscita determinati dati (gli ingressi stessi) che vengono processati dagli strati successivi.[20]

Con questa nuova rappresentazione dividiamo la rete in tre sezioni, già indicate (in inglese) in figura 1.9:

- Lo strato di ingresso.
- Gli *strati nascosti*[21], composti dai neuroni interni.
- Lo strato di uscita.

Una rete con un singolo strato nascosto viene detta *rete neurale*[22].

Una rete con più strati nascosti viene chiamata *rete neurale profonda*[23].

Notiamo queste due ulteriori caratteristiche dagli esempi precedenti:

[20] Quanto detto potrebbe non essere così intuitivo per alcuni, ho cercato di rendere l'idea il più chiara possibile.

[21] In inglese *hidden layers*.

[22] In inglese *neural network* oppure *shallow neural network*.

[23] In inglese *deep neural network*, abbreviato: *DNN*.

- Ogni neurone di uno strato manda la sua uscita come ingresso a *tutti* i neuroni dello strato successivo.

- La direzione in cui la computazione avviene va dallo strato di ingresso verso lo strato di uscita, strato per strato.

La prima caratteristica porta alla classificazione di rete *completamente connessa*, concetto che si spiega da sé.

La seconda caratteristica porta alla classificazione di rete *in avanti*[24], cioè dove il flusso di computazione avviene dagli ingressi verso le uscite, da uno strato a quello successivo, senza loop interni.

Esistono reti che non seguono queste regole: ci sono reti non completamente connesse dove alcuni collegamenti non risultano importanti,[25] oppure ci sono reti con loop interni.[26]

Non l'ho fatto notare direttamente ma una rete neurale può avere più uscite come mostrato in figura 1.10.

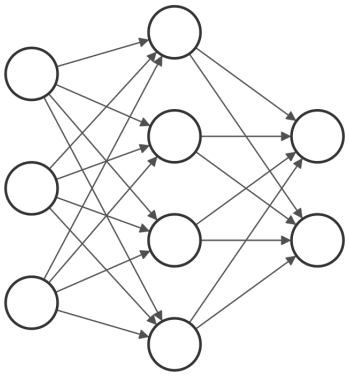

Figura 1.10: Rete neurale con più uscite.

Come ho fatto notare all'inizio di questa sezione, le reti neurali, essendo strutturate in strati, riescono a elaborare problemi più complessi ed astratti rispetto al singolo neurone (che adesso potremmo vedere come una 'mini' rete neurale).

Per esempio avevo detto che il singolo percettrone non fosse in grado di implementare una porta `XOR`. Abbiamo però visto insieme che fosse in grado di implementare una porta `NAND`. La porta `NAND` però è stata definita come una *porta logica universale*: l'unione di più porte `NAND`, fatta in maniera opportuna, può implementare ogni funzione logica. Ovviamente questo fatto si rispecchia nelle reti neurali, cioè si può costruire una rete neurale composta da più percettroni che implementi la funzione logica `XOR`.

[24] In inglese *feed forward*.

[25] Pensate ad un collegamento con peso vicino a zero.

[26] Le così dette *reti neurali ricorrenti*. In inglese *recurrent neural networks*.

Da quanto visto possiamo intuire che esistano differenti architetture di reti neurali, modellate per essere efficienti in diversi contesti.[27]

Per quanto affronterò nella mia trattazione userò una semplice rete neurale in avanti e completamente connessa.

Introdotte le reti neurali e la loro struttura, possiamo passare al concetto di *appredimento*.

1.2 L'apprendimento nelle reti neurali artificiali

1.2.1 Il concetto di apprendimento

"Si dice che un programma apprende dall'esperienza E con riferimento a alcune classi di compiti T e con misurazione della performance P, se le sue performance nel compito T, come misurato da P, migliorano con l'esperienza E."[5]

Questa è una definizione generale per l'*apprendimento automatico*[28], una branca dell'intelligenza artificiale. Le reti neurali sono un tipico approccio usato nell'apprendimento automatico.

In parole povere, adattando la frase al nostro problema: "una rete neurale *apprende* quando, svolgendo i compiti assegnati, migliora le sue prestazioni[29] nell'eseguirli."

L'obiettivo principale è che la nostra rete neurale impari, in un certo senso, a *generalizzare* dalla propria esperienza. Cioè che impari a svolgere ragionamenti per induzione al fine di portare a termine anche compiti che non ha mai affrontato (quindi di cui non ha avuto esperienza diretta).

Il punto da cui si parte nel voler costruire una rete neurale artificiale consiste nel raccogliere un *set di dati*[30]. Su di essi la nostra rete neurale deve eseguire dei calcoli. Generalmente si divide il set di dati in due parti, una parte che viene usata per l'apprendimento[31] della rete, l'altra parte per il test delle prestazioni.

Abbiamo tre possibili modi di affrontare l'apprendimento:

- Nel apprendimento *supervisionato* vengono dati alla rete esempi di ingressi e dei risultati desiderati da essi. Quindi l'obiettivo della rete è quello di elaborare una regola generale che associ l'ingresso al risultato (che esprimerà attraverso le sue attivazioni di uscita).

- Nel apprendimento *non supervisionato* vengono dati alla rete solo degli ingressi (senza il risultato desiderato) e la rete cerca di trovare una relazione in essi.

[27] Esempi sono le reti neurali convoluzionali, le reti neurali ricorrenti e altre.
[28] In inglese *machine learning*.
[29] Userò *performance* e *prestazioni* in maniera intercambiabile.
[30] In inglese *data set*.
[31] Si può chiamare, e lo farò spesso, anche *allenamento*. In inglese *training*.

- Nel apprendimento *per rinforzo* vengono dati alla rete gli ingressi e delle indicazioni se si stia comportando bene o male nell'eseguire il compito.[32]

A seconda di come si interpretano le uscite di una rete neurale si possono fare ulteriori considerazioni sul tipo di *compito* che essa stia eseguendo:

- Nella *classificazione*, le uscite della rete sono divise in una o più classi determinate a priori. L'obiettivo è classificare un ingresso in una di queste classi.

- Nella *regressione*, l'uscita è continua. L'obiettivo è valutare qualcosa in maniera continua rispetto l'ingresso.[33]

- Nel *clustering*, le uscite della rete sono divise ancora in classi ma, a differenza della classificazione, le classi non sono note a priori: sta alla rete crearle.

Per avvicinarci a come in pratica avvenga l'apprendimento, possiamo immaginarci che ogni volta che la rete esegue un determinato compito, essa vada a modificare dei suoi parametri interni al fine di migliorare le proprie performance. Questi parametri interni sono i pesi e i bias.

Tutto questo però deve avvenire in maniera graduale, al fine di non stravolgere la rete ad ogni passaggio ma avvicinarsi un po' alla volta al risultato voluto.

Se immaginiamo una rete di percettroni, possiamo capire che potremmo avere qualche problema con tutto ciò: anche un cambiamento minimo dei parametri interni della rete potrebbe portare a un cambio radicale di alcuni di questi neuroni, rendendo la rete difficilmente prevedibile nella risposta e ciò renderebbe difficile capire come modificare i parametri successivamente. Diversamente, possiamo immaginare che una rete composta da unità lineari rettificate o neuroni sigmoidali possa, a piccole variazioni dei parametri interni, portare a piccole variazioni nell'uscita. Questo ci permette di avere una rete più prevedibile e quindi di raggiungere risultati migliori nell'apprendimento.

Attualmente, le *unità lineari rettificate* risultano il modello di neurone più utilizzato nelle reti neurali profonde, in quanto risultano più facili da allenare delle reti che implementano i neuroni sigmoidali e i risultati sono comunque ottimi.[6]

Ora credo sia ben chiaro perché siano nati più modelli di neuroni, sono i componenti base della nostra rete neurale: modelli diversi hanno vantaggi (e svantaggi) diversi.

[32] Per esempio una rete che cerca di imparare a guidare un veicolo è come se avesse un assistente che le suggerisce se sta guidando bene o male.

[33] Per esempio il valore di una casa rispetto a determinati parametri.

L'*apprendimento automatico* è un argomento molto vasto e in continua evoluzione. Ci sono approcci all'apprendimento diversi dalle reti neurali. Come già accennato, le stesse reti neurali hanno diverse varianti a seconda dei compiti che devono eseguire. Non posso quindi trattare in generale di ogni possibile approccio nelle reti neurali, per ovvie ragioni mi dovrò limitare al mio specifico caso.

Nell'implementazione della nostra rete per riconoscere numeri scritti a mano, sfrutteremo la metodologia dell'*apprendimento supervisionato*. Il compito che la rete esegue è una *classificazione*, dove le classi sono i 10 numeri da 0 a 9 (compresi ovviamente).

Quindi le spiegazioni che seguono si incentrano su queste problematiche:

- Reti neurali in un contesto di *apprendimento supervisionato*.
- Reti neurali in un contesto di *classificazione*.

Nella prossima sezione spiegherò come si è arrivati e in cosa consiste l'algoritmo che ci permette di regolare i parametri interni della rete in seguito al confronto con i dati di allenamento del set di dati: la *retropropagazione dell'errore*[34].

È un argomento piuttosto complesso e per questo cercherò di seguire il filo del discorso seguito da Michael Nielsen nel suo libro online.[3]

Farò comunque delle aggiunte dove noterò che siano importanti o eliminerò sue considerazioni dove le credo superflue.

1.2.2 La retropropagazione dell'errore

Come detto la retropropagazione dell'errore è un argomento complesso: dobbiamo introdurre diversi concetti prima di poterlo comprendere.

La funzione di costo

Una rete neurale viene generalmente inizializzata con parametri interni casuali o ricavati tramite euristici. Questi ovviamente sono parametri iniziali e attraverso l'apprendimento si cerca di migliorarli.

In un contesto di apprendimento supervisionato, un set di dati per l'allenamento avrà, per ogni dato raccolto, il risultato che si vuole mostrare alla rete.[35] Difatto abbiamo una relazione $y(x)$ dove y indica il risultato del determinato dato x. I dati x saranno gli ingressi della nostra rete. Essa produrrà delle uscite, dipendenti dai suoi parametri interni, che identifichiamo nelle attivazioni di uscita a e queste dovranno essere confrontate con $y(x)$.

Per misurare di quanto la rete stia sbagliando, si introduce una *funzione di costo*[36].

[34] In inglese *backpropagation*.

[35] Nella letteratura inglese si dice che ogni dato ha una *label*, cioè una *etichetta*.

[36] In inglese *cost function* oppure *loss function*.

Noi come funzione di costo usiamo l'*errore quadratico medio*:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2 \tag{1.8}$$

La w denota la collezione di tutti i pesi nella rete, b i bias, n il numero totale di ingressi, a il vettore delle attivazioni delle uscite quando x è in ingresso, e la somma avviene per tutti gli ingressi x. Le attivazioni a sono funzione di x, w e b, non ho scritto la funzione completa per semplicità di notazione. La notazione $\|v\|$ indica il modulo di v.

Possiamo notare diverse cose:

- Abbiamo definito una funzione di costo che tiene conto di tutti gli ingressi del set di dati per l'allenamento.
- Questa funzione non è mai negativa, visto che ogni elemento sommato è positivo.
- $C(w,b)$ diventa piccola, cioè $C(w,b) \approx 0$, quando $a \approx y(x)$, cioè quando le uscite della nostra rete stanno approssimando $y(x)$.
- $C(w,b)$ diventa grande quando a non sta approssimando $y(x)$.

L'obiettivo dell'allenamento di una rete è ottimizzare questa funzione di costo, cioè di minimizzarla.[37]

Osservando la funzione di costo $C(w,b)$ faccio notare che $y(x)$ non viene vista come una sua variabile. Questo accade perché i dati del set sono *fissati* una volta raccolti e ad essi sono assegnati *risultati fissi*. Non avrebbe senso variare questi dati al fine di diminuire la funzione di costo: sarebbe come variare il set di dati perché si adatti alla nostra rete, cosa che ovviamente non si vuole fare.

Quindi i valori che si modificano per minimizzare la funzione di costo sono i parametri interni alla rete, cioè i pesi w e i bias b. Così facciamo in modo che sia la rete a dover generalizzare la relazione $y(x)$ definita dal set di dati.

Cercare di minimizzare questa funzione attraverso il semplice calcolo di massimi e minimi, come si fa nei problemi di analisi risulta difficile, soprattutto quando essa dipende da un numero molto alto di pesi e bias, che è una cosa normale nelle reti neurali.

La minimizzazione della funzione di costo quindi avviene con un metodo che attacca il problema in una maniera diversa: il metodo di *discesa del gradiente*.

[37] Praticamente stiamo parlando di un problema di *ottimizzazione*, come i famosi problemi *di massimo e minimo* nell'analisi matematica.

La discesa del gradiente

"In ottimizzazione e analisi numerica il metodo di discesa del gradiente (detto anche metodo del gradiente, metodo steepest descent o metodo di discesa più ripida) è una tecnica che consente di determinare i punti di massimo e minimo di una funzione a più variabili."[7]

Preso un 'punto di partenza' in una funzione a più variabili, l'algoritmo di discesa del gradiente consiste nel: valutare il gradiente della funzione nel punto, prenderlo negativamente, attraverso una proporzione decidere un passo di discesa, aggiornare il punto in cui ci si trova. Questo porta in genere alla scoperta di un minimo locale, non necessariamente globale.

Algoritmo: Discesa del gradiente
$k = 0$;
finché $\bigtriangledown f(x_k) \neq 0$ **fai**
  calcolare la direzione di discesa $p_k := - \bigtriangledown f(x_k)$;
  calcolare il passo di discesa α_k;
  $x_{k+1} = x_k + \alpha_k p_k$;
  $k = k + 1$;
fine

Per analizzare ulteriormente la questione, dimentichiamoci temporaneamente delle reti neurali, pensiamo semplicemente di ottimizzare una generica funzione a più variabili con il metodo della discesa del gradiente.

Immaginiamo di avere una funzione C dipendente da un certo v con $v = (v_1, v_2)$, quindi di fatto abbiamo $C(v) = C(v_1, v_2)$.[38]

A questo punto qualche ricordo di analisi matematica può suggerirci che:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{1.9}$$

Dobbiamo trovare un modo di variare Δv_1 e Δv_2 in modo che ΔC sia negativa. Per immaginarci meglio come fare questa scelta definiamo Δv come il vettore che indica la variazione di v, $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$, dove T è l'operazione di trasposizione, che qui trasforma righe in colonne.

Definiamo anche il *gradiente* di C come il vettore delle derivate parziali:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T \tag{1.10}$$

Con queste definizioni, l'espressione 1.9 per ΔC può essere riscritta come:

$$\Delta C \approx \nabla C \cdot \Delta v \tag{1.11}$$

Dove per '$\cdot$' si intende il prodotto scalare.

[38] Ho scelto due variabili ma potrebbero essere di più.

Il gradiente ∇C regola come varia C a seconda di come varia v.

Da questa espressione possiamo ricavare come scegliere Δv in modo da avere un ΔC negativo. In particolare se scegliamo:

$$\Delta v = -\eta \nabla C \tag{1.12}$$

Dove η è un parametro positivo e generalmente 'piccolo', chiamato *tasso di apprendimento*[39]. Sostituendo la nostra scelta in 1.11 possiamo vedere che:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2 \tag{1.13}$$

Poichè $\|\nabla C\|^2 \geq 0$, questo garantisce che $\Delta C \leq 0$, quindi variando v di Δv, C decrescerà soltanto.

Usando quindi 1.12 possiamo calcolare la nuova posizione v' in questo modo:

$$v \to v' = v - \eta \nabla C \tag{1.14}$$

Se usiamo questa regola in maniera iterativa, passo dopo passo, faremo decrescere C fino ad arrivare a un minimo, che sarà quanto meno un minimo locale.

Dobbiamo scegliere un η 'piccolo abbastanza' da non generare salti troppo grandi nella posizione[40], ma neanche un η troppo piccolo, poiché ogni passo modificherebbe di poco la posizione rendendo il processo più lungo del necessario.

Proviamo adesso ad applicare questo alle reti neurali e quindi alla funzione di costo. Abbiamo già tutto ma lo abbiamo affrontato in una forma più generica. La funzione C diventa la funzione di costo[41], essa ha come variabili i pesi w e i bias b, essi vengono inizializzati e ci indicano il punto da cui si parte, poi con le regole 1.15 e 1.16 aggiorniamo i loro valori:

$$w \to w' = w - \eta \frac{\partial C}{\partial w} \tag{1.15}$$

$$b \to b' = b - \eta \frac{\partial C}{\partial b} \tag{1.16}$$

Nell'applicazione pratica di questo metodo però abbiamo un problema computazionale: essenzialmente bisogna computare per ogni singolo ingresso una funzione di costo $C_x \equiv \frac{\|y(x)-a\|^2}{2}$, sommarle tutte e farne una media così da ottenere la funzione di costo totale $C = \frac{1}{n}\sum_x C_x$. In pratica per calcolare il gradiente ∇C, dobbiamo calcolare i gradienti ∇C_x separatamente per ogni ingresso x e poi farne una media, $\nabla C = \frac{1}{n}\sum_x \nabla C_x$. Purtroppo, quando il numero di ingressi diventa molto grande questa operazione può richiedere molto tempo, facendo diventare l'allenamento lento.

Vediamo successivamente una possibile soluzione al problema.

[39] In inglese *learning rate*.

[40] Potrebbero farci oscillare attorno ad un minimo, o addirittura allontanarci da esso.

[41] Avevo scelto di usare la lettera C per questo.

La discesa stocastica del gradiente

La *discesa stocastica del gradiente* opera in maniera simile alla *discesa del gradiente*, però stima ∇C attraverso una media di pochi ∇C_x. Facendo così, valutando quindi una *stima* piuttosto che il *valore esatto* di ∇C, possiamo accelerare di molto il processo e quindi l'apprendimento.

Per dare una idea numerica, la discesa stocastica del gradiente funziona prendendo un numero m di ingressi $X_1,\ldots,X_m$, che noi chiameremo un *mini-lotto*[42]. Prendendo un numero di ingressi non troppo alto ma neanche troppo basso,[43] possiamo pensare che il valore medio ∇C_{X_j} sia simile al valore medio su tutti i ∇C_x, cioè:

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C \tag{1.17}$$

Dove la prima somma è sui m valori scelti, mentre la seconda somma è su tutti i dati di allenamento.

Quindi 'invertendo' la lettura di 1.17:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j} \tag{1.18}$$

Quindi possiamo stimare il gradiente totale con il gradiente di un mini-lotto.

Prendiamo quindi w e b che denotano i pesi e i bias della nostra rete neurale. Con la discesa stocastica del gradiente otteniamo:

$$w \rightarrow w' = w - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w} \tag{1.19}$$

$$b \rightarrow b' = b - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b} \tag{1.20}$$

Dove le somme sono sui m ingressi di allenamento X_j del mini-lotto corrente. Successivamente si sceglie un'altro mini-lotto, prendendo altri m ingressi, si esegue l'aggiornamento dei parametri e così via finché non abbiamo finito gli ingressi di allenamento. Questo periodo in cui si ciclano tutti gli ingressi di allenamento si chiama *epoca*[44]. Finita una epoca si può incominciare con un'altra. La scelta di quante epoche affrontare è, come il *tasso di apprendimento*, un così detto *iperparametro*. Questi parametri sono scelti facendo vari test alla ricerca dei migliori, oppure con euristici, oppure, se si ha una certa esperienza con le reti neurali, con del buon senso.

È bene notare che ci sono diverse convenzioni su come eseguire le medie nella funzione di costo, i pesi e i bias. Nella 1.8 abbiamo eseguito la media

[42] In inglese *mini-batch*.

[43] Nel primo caso il calcolo tornerebbe ad essere troppo pesante, nel secondo caso diventerebbe una stima imprecisa del gradiente.

[44] In inglese *epoch*.

nella funzione di costo con un fattore $\frac{1}{n}$. Certe volte si omette il $\frac{1}{n}$ sommando tutti i costi degli ingressi individuali senza fare la media. Ciò è utile quando non si sa quanti siano gli ingressi di allenamento: per esempio questo accade quando questi dati sono generati in tempo reale. In un modo simile la 1.19 e 1.20 possono omettere $\frac{1}{m}$ davanti alle sommatorie. Concettualmente questo crea poca differenza in quanto è come modificare il fattore η.

Notare che abbiamo parlato del gradiente della funzione di costo ma non di come calcolarlo, questo sarà l'obiettivo delle sezioni successive.

Un approccio matriciale per calcolare l'uscita di una rete neurale

In questa sezione mostrerò che un approccio con calcoli matriciali, è un approccio efficiente per computare l'uscita di una rete neurale.

Per prima cosa indichiamo con w^l_{jk} il peso che connette il k-esimo neurone nel $(l-1)$-esimo strato al j-esimo neurone nel l-esimo strato. Questa notazione può essere un po' difficile da capire inizialmente. Useremo una notazione simile per il bias del j-esimo neurone nel l-esimo strato, b^l_j, e per l'attivazione del j-esimo neurone nel l-esimo strato a^l_j. Nelle spiegazioni che seguono (fino alla fine del capitolo) userò la funzione di attivazione σ.

Con queste notazioni è possibile mettere in relazione l'attivazione del singolo neurone j nello strato l, a^l_j, con le attivazioni dello strato precedente:

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right) \tag{1.21}$$

Dove la somma è su tutti i neuroni k dello strato $(l-1)$-esimo.

Modifichiamo il tutto in modo da essere visto come un calcolo tra matrici. Definiamo w^l per ogni strato l, i cui valori interni sono semplicemente w^l_{jk}, quindi è una matrice di dimensione $j \times k$. In maniera simile facciamo con b^l dove i valori interni saranno i b^l_j, quindi è una matrice di dimensione $j \times 1$, cioè un vettore colonna. Infine con a^l contenente i valori a^l_j quindi anche qui una matrice di dimensione $j \times 1$, cioè un vettore colonna.

In ogni caso è bene notare che non si considerano w^l, b^l, per lo strato di ingresso alla rete, in quanto non è propriamente un vero strato di neuroni (come affrontato precedentemente nella sezione 1.1.2).

L'ultima idea che ci manca è quella di *vettorizzare* la funzione σ, ciò significa semplicemente che essa si applicherà ad ogni elemento della matrice risultante, che sarà un vettore colonna.[45]

Con queste considerazioni possiamo riscrivere la 1.21 così:

$$a^l = \sigma(w^l a^{l-1} + b^l) \tag{1.22}$$

Con la 1.22 possiamo vedere come le attivazioni di uno strato siano legate a quelle dello strato precedente, in una maniera matriciale e piuttosto

[45] Lascio al lettore la verifica di questa semplice conclusione.

semplice. In pratica il vettore delle attivazioni del l-esimo strato si ottiene prendendo quello del $(l-1)$-esimo, moltiplicando quest'ultimo per la matrice dello strato attuale w^l, sommando i bias dello strato attuale , b^l, e infine imponendo la funzione sigmoidea per ogni elemento.

Questo è anche il modo in cui l'implementazione del nostro codice affronterà le cose.

Definiamo ora come z, la quantità intermedia che viene presa come ingresso dalla funzione sigmoidea:

$$z^l \equiv w^l a^{l-1} + b^l \tag{1.23}$$

Chiamiamo z: *ingresso pesato* allo strato l.[46] Quindi si può riscrivere la funzione così $a^l = \sigma(z^l)$. Ritornando un attimo alla visione per indici possiamo vedere che z^l ha come componenti $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$, che sono appunto l'*ingresso pesato* al neurone j nello strato l.

Le due condizioni che dobbiamo verificare sulla funzione di costo

L'obiettivo della retropropagazione dell'errore è quello di calcolare le derivate parziali $\partial C/\partial w$ e $\partial C/\partial b$ della funzione di costo C in relazione ad ogni peso o bias della rete.

Per fare in modo che la retropropagazione funzioni dobbiamo fare due considerazioni sulla forma della funzione di costo.

Ricordiamo la nostra funzione di costo quadratica:

$$C = \frac{1}{2n} \sum_x \|y(x) - a^L(x)\|^2 \tag{1.24}$$

Dove n, è il numero totale degli ingressi di allenamento, la somma è su tutti gli ingressi di allenamento x, $y(x)$ è l'uscita desiderata, L denota il numero di strati della rete e $a^L = a^L(x)$ è quindi il vettore delle attivazioni delle uscite quando x è l'ingresso (ho introdotto le ultime notazioni).

La prima considerazione da fare è che la funzione di costo possa essere riscritta come una media di una funzione di costo singola[47]. Cioè che $C = \frac{1}{n}\sum_x C_x$, con C_x che sono le funzioni di costo per i singoli ingressi di allenamento x. La funzione di costo quadratica rientra in questa cosiderazione, con la funzione di costo per singolo ingresso: $C_x = \frac{1}{2}\|y - a^L\|^2$. Questo ci serve perché con la funzione di costo per singolo ingresso calcoliamo $\partial C_x/\partial w$ e $\partial C_x/\partial b$ e facendo una media di questi $\partial C/\partial w$ e $\partial C/\partial b$.

Supponiamo che l'ingresso di allenamento sia fissato ad un particolare x, quindi non usiamo più il pedice x per le spiegazioni successive, scrivendo semplicemente C invece che C_x. È giusto per semplificare un po' la notazione, riporteremo la x in gioco solo alla fine, per concludere il discorso.

[46] Notare che la *somma pesata* presentata a inizio della trattazione non includeva il bias, qui *l'ingresso pesato* lo include per semplificare la notazione.

[47] Detta anche *funzione di errore*.

La seconda considerazione è che la funzione di costo singola possa essere scritta come una funzione delle attivazioni di uscita della rete. E abbiamo:

$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2 \tag{1.25}$$

Quindi è chiaramente funzione delle uscite a_j^L.

Ricordo che y non è considerato un parametro modificabile della funzione di costo, come visto a pagina 16.

Il prodotto puntuale o di Hadamard

Questa operazione ci sarà utile.

Essa introduce semplicemente un prodotto elemento per elemento $\odot$.

Per fare un esempio:

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1*3 \\ 2*4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix} \tag{1.26}$$

Le quattro equazioni fondamentali dietro alla retropropagazione

Con la retropropagazione vogliamo arrivare a capire come cambiare i pesi e i bias della rete per far diminuire la funzione di costo. Questo significa che dobbiamo calcolare le derivate parziali $\partial C/\partial w_{jk}^l$ e $\partial C/\partial b_j^l$. È utile introdurre una quantità intermedia, che chiamiamo *errore* nel neurone j dello strato l, δ_j^l. Con la retropropagazione calcoliamo questo δ_j^l e con alcune equazioni, partendo da esso ricaveremo $\partial C/\partial w_{jk}^l$ e $\partial C/\partial b_j^l$.

Per capire meglio l'errore cosa rappresenta, pensiamo ai neuroni negli strati nascosti della rete. Prendendo un singolo neurone, sappiamo che applicherà una funzione di attivazione a un ingresso pesato z. Se a questo z aggiungiamo un Δz, applicando la funzione sigmoidale per calcolare l'attivazione abbiamo $a_j^l = \sigma(z_j^l + \Delta z_j^l)$. Quindi questa perturbazione dell'ingresso pesato, perturba l'attivazione, la quale propaga la perturbazione negli strati successivi. Ciò ci porta a considerare che la funzione di costo sia cambiata, nei rispetti di questo cambiamento in questo modo: $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$. Come abbiamo già detto più volte, vogliamo minimizzare la funzione di costo. Quindi cercheremo di modificare Δz in modo da farla diminuire. Quanto diminuisce però dipende da $\frac{\partial C}{\partial z_j^l}$. Se questa derivata parziale è un numero piccolo allora potremmo fare poco con Δz, se invece risulta essere un numero piuttosto grande, possiamo diminuire di un valore significativo la funzione di costo attraverso Δz.

Quindi da questo definiamo l'*errore*:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} \tag{1.27}$$

Come nella convenzione già mostrata, vediamo δ^l come il vettore colonna contenente i δ^l_j.

Dobbiamo ora osservare quattro equazioni fondamentali che stanno dietro alla retropropagazione.

La prima equazione computa l'errore nello strato di uscita:

$$\delta^L_j = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j) \tag{1.28}$$

Questa equazione deriva direttamente dalla 1.27, sostituendo $l = L$ e usando la così detta *regola della catena*:

$$\delta^L_j = \frac{\partial C}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j}\frac{\partial a^L_j}{\partial z^L_j} = \frac{\partial C}{\partial a^L_j}\sigma'(z^L_j) \tag{1.29}$$

Il primo termine della 1.28, $\partial C/\partial a^L_j$, misura quanto velocemente cambi la funzione di costo nei rispetti della j-esima uscita. Se per esempio C non dipendesse particolarmente dalla uscita j, allora l'errore δ^L_j sarà piccolo. Il secondo termine $\sigma'(z^L_j)$ indica quanto velocemente cambi la funzione di attivazione del particolare neurone di uscita, rispetto all'ingresso pesato che gli viene sottoposto z^L_j.

Tutto questo può essere facilmente calcolato. Abbiamo già visto come calcolare il vettore z^l contenente i z^l_j per ogni possibile strato con la 1.23. Calcolare la derivata della funzione di attivazione non è un problema. Nel caso della funzione sigmoidea è $\sigma'(z^L_j) = \sigma(z^L_j)(1-\sigma(z^L_j))$. Anche la derivata della funzione di costo non è un problema: in questo caso considerando la funzione di costo del singolo ingresso $C = \frac{1}{2}\sum_j (y_j - a^L_j)^2$ diventa considerando il generico neurone j, $\partial C/\partial a^L_j = (a^L_j - y_j)$

Quindi, mettendo l'errore in forma matriciale, usando le ultime considerazioni e utilizzando il prodotto puntuale otteniamo:

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{1.30}$$

Dove $\nabla_a C$ è definito come il vettore colonna che indica il gradiente della funzione di costo rispetto al vettore delle attivazioni, quindi contiene i valori $\partial C/\partial a^L_j$, che abbiamo visto essere $(a^L_j - y_j)$. Quindi otteniamo:

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{1.31}$$

La seconda equazione mette in relazione l'errore allo strato l, δ^l con l'errore allo strato successivo δ^{l+1}:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{1.32}$$

Per farci una idea di perché funzioni questa equazione la dimostriamo. Partendo dal δ^l_j e usando la regola della catena possiamo dire che:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sum_k \frac{\partial C}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \frac{\partial z_k^{l+1}}{\partial z_j^l} \delta_k^{l+1} \tag{1.33}$$

Dove nell'ultima parte abbiamo invertito due termini e sostituito l'errore δ_k^{l+1} usando la sua defininizione.

Valutiamo z_k^{l+1} in questo modo:

$$z_k^{l+1} = \sum_j w_{kj}^{l+1} a_j^l + b_k^{l+1} = \sum_j w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1} \tag{1.34}$$

Quindi non abbiamo fatto altro che usare la sua definizione e nell'ultimo passaggio abbiamo esplicitato la funzione di attivazione con la sua definizione. Facendo la derivata parziale per z_j^l otteniamo:

$$\frac{\partial z_k^{l+1}}{\partial z_j^l} = w_{kj}^{l+1} \sigma'(z_j^l) \tag{1.35}$$

Sostituendo in 1.33 otteniamo:

$$\delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} \sigma'(z_j^l) \tag{1.36}$$

Che sono i componenti della 1.32 calcolati in maniera normale elemento per elemento invece che per calcolo matriciale, farete in fretta a ricondurvi all'altra notazione se siete arrivati fin qui.[48]

Capisco comunque che la dimostrazione possa essere un po' complicata, se non l'avete capita bene non è importantissimo. Quello che importa è capire che abbiamo un modo di calcolare il vettore degli errori dello strato l, δ^l, rispetto a quello degli errori dello strato successivo $l+1$, δ^{l+1}. Questo avviene usando i vari pesi che connettono i due strati e la derivata della funzione di attivazione, quindi tutti elementi che abbiamo già calcolato.

Combinando quindi 1.28 e 1.32 possiamo calcolare l'errore per ogni strato della rete, δ^l. Partiamo usando la 1.28 per calcolare l'errore nello strato finale e poi, applicando iterativamente 1.32 andiamo indietro di strato in strato.

La terza equazione ci mette in relazione la variazione della funzione di costo rispetto ad ogni bias della rete:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{1.37}$$

Quindi la variazione $\frac{\partial C}{\partial b_j^l}$ è esattamente uguale all'errore δ_j^l e quindi sappiamo già come calcolarla grazie a 1.28 e 1.32.

[48] La trasposizione in 1.32 avviene perché nella 1.36 abbiamo w_{kj} invece che w_{jk}.

La quarta e ultima equazione ci mette in relazione la variazione della funzione di costo rispetto a ogni peso nella rete:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{1.38}$$

Anche in questo caso sappiamo come calcolare a^{l-1}_k e δ^l_j.

Sia 1.37 e 1.38 si dimostrano con la regola della catena.

Per quanto riguarda la 1.37:

$$\frac{\partial C}{\partial b^l_j} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial b^l_j} = \frac{\partial C}{\partial z^l_j} = \delta^l_j \tag{1.39}$$

Poichè $\frac{\partial z^l_j}{\partial b^l_j} = 1$, basta guardare $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$ per vederlo.

Per quanto riguarda la 1.38:

$$\frac{\partial C}{\partial w^l_{jk}} = \frac{\partial C}{\partial z^l_j}\frac{\partial z^l_j}{\partial w^l_{jk}} = \delta^l_j a^{l-1}_k \tag{1.40}$$

Poichè $\frac{\partial z^l_j}{\partial w^l_{jk}} = a^{l-1}_k$, basta guardare la definizione $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$.

Possiamo ora vedere l'algoritmo della retropropagazione dell'errore.

L'algoritmo della retropropagazione dell'errore

Le quattro equazioni della retropropagazione dell'errore ci permettono di calcolare ∇C.

Scriviamo il tutto come un algorimo:

1. Mettere gli ingressi di allenamento x nella rete: essi quindi sono le attivazioni dello strato di ingresso a^1.
2. Fare la propagazione in avanti: per ogni $l = 2, 3, \ldots, L$ calcolare $z^l = w^l a^{l-1} + b^l$ e $a^l = \sigma(z^l)$.
3. Calcolare l'errore delle uscite δ^L: con il calcolo $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4. Retropropagare l'errore: per ogni $l = L-1, L-2, \ldots, 2$ calcolare $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5. Calcolare i componenti di ∇C: saranno dati da $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ e $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

Essenzialmente si chiama *retropropagazione* proprio perché: calcoliamo l'errore sull'ultimo strato della rete e quest'ultimo, sfruttando la regola della catena, viene poi usato per calcolare i vari vettori di errore della rete, in un movimento *propagato all'indietro*.

L'algoritmo descritto vale per la discesa del gradiente normale.

Applicando la retropropagazione dell'errore usando la discesa stocastica del gradiente otteniamo questo algoritmo:

1. Mettere gli ingressi x del mini-lotto nella rete.

2. Per ogni ingresso di allenamento x essi diventano le attivazioni dello strato di ingresso $a^{x,1}$ ed eseguiamo:

   - Una propagazione in avanti: dove per ogni $l = 2, 3, \ldots, L$ calcoliamo $z^{x,l} = w^l a^{x,l-1} + b^l$ e $a^{x,l} = \sigma(z^{x,l})$.
   - Calcoliamo l'errore di uscita $\delta^{x,L}$: con $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.
   - Retropropaghiamo l'errore: per ogni $l = L-1, L-2, \ldots, 2$ calcoliamo $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.

3. In fine applichiamo la discesa del gradiente: per ogni $l = L, L-1, \ldots, 2$ aggiorniamo i pesi in accordo con la regola $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$ e i bias in accordo con $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

Ovviamente nel codice tutto questo sarà accompagnato da un *loop* esterno che genera i mini-lotti prendendo m ingressi di allenamento e un loop ulteriormente esterno, che cicla il processo per il numero di epoche di allenamento che si è deciso di affrontare.

Abbiamo finalmente affrontato tutta la teoria matematica su cui si basa il nostro codice.

Spero di essere stato il più chiaro e preciso possibile nei passaggi. Purtroppo avere tutti questi indici e lettere può non sembrare proprio comodo... ma è necessario.

Passiamo alla parte più pratica della questione: l'implementazione della mia rete neurale per riconoscere numeri scritti a mano.

Capitolo 2

Implementazione del codice in Python

In questo capitolo mostrerò:

- Il codice iniziale per la realizzazione di una rete neurale semplice per il riconoscimento di numeri scritti a mano.
- Il codice che ho scritto per implementare nuove funzionalità al fine di iniettare degli errori e testare le successive performance della rete.

2.1 Il codice iniziale

2.1.1 Il codice di Michael Nielsen

Il codice da cui sono partito per effettuare il mio esperimento viene dato in allegato al libro online di Michael Nielsen.[3] Rimando all'appendice A, per ulteriori informazioni sul codice e i programmi che ho utilizzato.

Lo scopo del codice

Questo codice implementa molti concetti che abbiamo visto nella teoria. Esso consente di implementare una rete neurale semplice che possa riconoscere i numeri scritti a mano.

Lanciando da terminale alcuni comandi, si può inizializzare una rete ed allenarla, vedendo dopo ogni epoca di allenamento le sue prestazioni. Le prestazioni sono riportate in termini di: *quanti numeri sono stati correttamente classificati sul totale di numeri da riconoscere.*[1]

Nella prossima sezione analizzo con più dettaglio il problema che il codice di partenza si è posto di risolvere.

[1] Nel mio codice riporterò invece una percentuale di *accuratezza*.

2.1.2 Una rete neurale semplice per il riconoscimento di numeri scritti a mano

Come accennato nella sezione 1.2.1 riguardante l'apprendimento, quando si vuole fare una rete neurale bisogna avere un set di dati su cui farla allenare. In questo frangente i dati usati sono quelli del *database MNIST*.

Il database MNIST

L'archivio che verrà usato è *mnist.pkl.gz* ed è compreso con il codice iniziale di Nielsen.[2]

Questa è essenzialmente la sua struttura:

- Contiene 60.000 immagini di numeri scritti a mano per l'allenamento della rete e 10.000 immagini per il test.
- Il formato dell'immagine è un formato 'personalizzato': le immagini sono delle matrici 28x28 con all'interno i valori dei pixel, quindi abbiamo immagini con risoluzione 28x28 pixel.
- I numeri sono stati normalizzati nella grandezza e centrati nell'immagine.
- I valori dei pixel sono dati in `unsigned byte` quindi esprimono valori da 0 a 255.[3] Questi valori possono essere interpretati come dei colori in scala di grigi, che variano, rispettivamente con 0 e 255, dal bianco al nero.

Analizzato il set di dati, vediamo l'architettura della nostra rete.

L'architettura della nostra rete

Come dovremmo avere già in mente, vogliamo una rete neurale per *classificare* i numeri del set di dati, quindi la struttura sarà la seguente:

- *Uscite*: al fine di classificare gli ingressi nei numeri da 0 a 9, la rete avrà bisogno di uno strato di uscita con 10 neuroni: un neurone di uscita per ogni numero.
- *Ingressi*: le immagini sono 28x28 pixel, la rete le prende come ingresso e le dovrà valutare interamente quindi essa avrà 784 ingressi: un ingresso per ogni pixel.

[2] Rimando all'appendice A per ulteriori informazioni.

[3] Se non fosse chiaro, un byte è 8 bit da cui i numeri rappresentabili sono in totale $2^8 = 256$, che sono i numeri da 0 a 255.

- *Strati nascosti*: vogliamo una rete neurale 'semplice ma efficace' nel suo compito, quindi avrà un solo strato nascosto composto da 30 neuroni.[4]

Il modello di neurone utilizzato sarà il *neurone sigmoidale*.
La funzione di costo usata sarà la *funzione di costo quadratica media*.

Analizziamo ora come questa rete venga implementata nel codice.

L'implementazione della rete neurale nel codice

Per eseguire un allenamento della rete bisogna scrivere questi comandi dalla shell di Python[5]:

I comandi per eseguire l'allenamento.

```
import mnist_loader
training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
import network
net = network.Network([784, 30, 10])
net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

La loro analisi ci aiuterà a comprendere l'implementazione della rete neurale:

1. Il comando *import mnist_loader* importa il file *mnist_loader.py*, che implementa:

   - Le funzioni *load_data()* e *load_data_wrapper()* che servono a caricare i dati del MNIST dall'archivio *mnist.pkl.gz*.
   - La funzione *vectorize_result()* che serve a convertire un numero (la risposta nel set di dati) nella uscita desiderata dalla rete neurale.

2. Il comando *training_data, validation_data, test_data = \ ...mnist_loader.load_data_wrapper()* va a caricare i dati del MNIST da *mnist.pkl.gz* in tre *liste* di Python: divide il set di 60.000 immagini di allenamento in 50.000 per l'*allenamento* (training_data) e 10.000 per la *validazione*[6] (validation_data), e carica le altre 10.000 per il *test* (test_data).

3. Il comando *import network* importa *network.py* che implementa:

[4] Nielsen dice che questo è un buon numero per raggiungere buone prestazioni senza far diventare la rete più complicata del dovuto.

[5] Ovviamente dopo essersi spostati nella cartella contenente i file. Questo si può fare direttamente nella shell di Python importando la libreria *os* e usando la funzione *os.chdir()* per cambiare cartella.

[6] La validazione però non viene usata per i nostri scopi.

- La classe *Network* che tramite *__init__()* prende in ingresso i valori per inizializzare gli *strati* della rete, ed in seguito inizializza, grazie alla conoscenza degli strati, i valori iniziali dei *pesi* e i *bias* con valori *random*[7].
- La funzione *feedforward()* che permette dato l'ingresso della rete di computare l'uscita di essa in base ai pesi e i bias.
- La funzione *SGD()* che è la funzione per eseguire la *discesa stocastica del gradiente*, questa funzione fa più cose:
  (a) Crea i *mini-lotti*.
  (b) Per ogni mini-lotto, chiama la *update_mini_batch()* che dopo aver chiamato la retropropagazione dell'errore, va a modificare i *pesi* e i *bias* della rete.
  (c) Infine scrive a terminale come procede l'allenamento di epoca in epoca effettuando un test delle prestazioni.
- La funzione *backprop()* che implementa la *retropropagazione dell'errore*: esegue una feed forward interna, calcola l'errore δ^L e, con il metodo già affrontato nella teoria, gli *errori* nei vari strati.
- La funzione *cost_derivative()* ritorna la *derivata della funzione di costo* singola rispetto le uscite (serve alla retropropagazione dell'errore).
- La funzione *evaluate()* per testare le prestazioni della rete sui dati di test.
- Le funzioni *sigmoid()* e *sigmoid_prime()* che contegono rispettivamente: la definizione della funzione sigmoidea e della sua derivata.

4. Il comando *net = network.Network([784, 30, 10])* inizializza un oggetto *net*, di classe *Network* con 784 ingressi, 30 neuroni nello strato nascosto e 10 di uscita.

5. Il comando *net.SGD(training_data, 30, 10, 3.0, test_data=test_data)* esegue la discesa stocastica del gradiente e dice:
   (a) Di passare alla funzione la lista contenente i dati dell'allenamento *training_data*.
   (b) Di eseguire 30 *epoche* di allenamento.
   (c) Di creare *mini-lotti* da 10 ingressi di allenamento.
   (d) Di usare un *tasso di apprendimento* di 3,0.

[7] Questi valori random vengono inizializzati osservando una distribuzione gaussiana a media 0 e varianza 1.

(e) Di assegnare alla variabile *test_ data* all'interno del codice la lista contenente i dati per il test delle prestazioni *test_ data*.[8]

Come si è capito, con i tre file, *network.py*, *mnist_ loader.py* e *mnist.pkl.gz*, si riesce a costruire una rete neurale semplice.

Queste sono le fondamenta da cui sono partito.

Voglio far notare che ci sono diversi modi per creare delle reti neurali al giorno d'oggi. Per esempio avrei potuto creare una rete usando *Tensorflow* che è una libreria della Google, in coppia con le API *Keras*. Facendo così, avrei potuto creare una rete neurale per fare questo compito con molte meno righe di codice, ma avrei perso il controllo sulla *architettura interna* della rete.

Usando invece il codice che ho presentato come punto di partenza, seppur semplice e forse non super efficiente, posso regolare ogni ingranaggio della rete neurale. Questo mi permette, potenzialmente, di inserire ipotetici *errori hardware* in una qualsiasi parte della sua architettura.

Passiamo alla prossima sezione in cui spiego il codice che ho implementato personalmente.

2.2 Il codice che ho scritto per implementare nuove funzionalità

Come abbiamo visto, il codice di partenza permette di inizializzare una rete e di allenarla. Dopo l'allenamento, in seguito ad alcuni test, ho notato che la rete raggiunge in genere tra il 94% - 95% di numeri correttamente riconosciuti. La variabilità è data dalla scelta randomica dei pesi iniziali e dalla *SGD()* che quando viene eseguita mescola[9] gli ingressi di allenamento prima di formare i mini-lotti.

Volendo applicare degli errori hardware alla rete e volendo mantere le mie statistiche il più rilevanti possibile, ho capito di aver bisogno di salvare lo stato della rete una volta allenato, così da avere un singolo 'modello' di rete su cui operare.

2.2.1 Il salvataggio dello stato allenato della rete

Per implementare il salvataggio dello *stato allenato della rete*, ho inserito del codice all'interno di *network.py*, nella funzione *SGD()*, la quale, dopo aver eseguito l'allenamento, chiede all'utente se vuole salvarlo. Se si risponde in

[8] Questo può sembrare un po' strano ma serve a fare eseguire il test delle prestazioni. Se non fosse passato *test_ data* alla omonima variabile, la rete si allenerebbe soltanto e a console dei comandi comparirebbe un generico "*Epoch X complete*" dove X indica il numero della epoca.

[9] Esegue uno *shuffle*.

maniera affermativa, viene creato un array di NumPy[10], *backup*, contenente i pesi e i bias. Attraverso *pickle.dump()*[11] tutto ciò viene salvato all'interno di un file *save* che è di fatto una sequenza di byte[12].

Eseguo anche un salvataggio dei parametri su file testuale (*weights.txt* per i pesi e *biases.txt* per i bias) per renderli 'leggibili'. Il caricamento dal file *save* risulta però immediato con la *pickle.load()* quindi uso essa per ricaricarli.

2.2.2 L'implementazione degli errori

Prima di pensare a *dove* iniettare gli errori, bisogna pensare a *come siano* gli errori che vogliamo iniettare.

Notiamo che i pesi, i bias e le attivazioni, sono tutti valori indicati in *virgola mobile*[13] e, per semplicità d'uso, solitamente sono rappresentati in base decimale.

Un circuito digitale opera però con valori binari, quindi quello che vorremmo fare è iniettare errori *a livello bit*. Per questo bisogna riuscire a convertire i nostri valori dal decimale al binario, cioè come li utilizza il circuito digitale.

Attraverso il codice in *conversioni.py*, che sfrutta la libreria *struct*, viene resa possibile la conversione di un numero a virgola mobile dal decimale alla forma delineata dallo standard IEEE 754 (forma binaria), e viceversa.

Il formato a virgola mobile che usiamo è a *doppia precisione* quindi sfrutta 64 bit. Possiamo vedere ogni valore che convertiamo in questa struttura:

```
[1 bit di segno][11 bit di esponente][52 bit di mantissa]
```

Passiamo a ragionare su dove si possano inserire gli errori.

Dove inserire gli errori hardware

La scelta più semplice che possiamo pensare è quella di manipolare i valori di pesi e i bias, questi infatti sono contenuti all'interno di matrici e, indicando gli indici di esse, possiamo andare a modificare qualsiasi valore. Essendo pesi e bias valori 'salvati' dovranno stare in una qualche cella di memoria. Un errore ragionevole su di esse è il *bit flip*, cioè, che in seguito ad un qualche problema, un bit, contenuto in una cella, inverta il suo valore. Per esempio, applicando questo errore ad un bit a 1, esso andrebbe a 0 (e viceversa).

Ai fini di questa tesi abbiamo deciso di fermarci a questo tipo di errore nei confronti dei soli pesi della rete.

Per approfondimento, ho affrontato anche la possibilità di inserire gli errori dopo le varie operazioni che esegue la rete, *ma non ho eseguito test*

[10] NumPy è una libreria che usiamo. Rimando all'appendice A per ulteriori informazioni.

[11] Pickle è un modulo che implementa un algoritmo per serializzare e de-serializzare la struttura di un oggetto Python.

[12] In inglese *bytestream*.

[13] In inglese *floating point*.

su questa parte. Rimando all'appendice B per vedere le mie conclusioni a riguardo.

La funzione che implementa gli errori

Al fine di inserire gli errori nei pesi ho scritto una funzione *do_ err_ w()* all'interno del file *network.py*. Essa prende 5 variabili:

- La *m_ block* cioè il 'blocco di pesi' in cui si vuole inserire gli errori. La nostra rete ha essenzialmente due blocchi di pesi, quello da ingresso a strato nascosto e quello da strato nascosto alle uscite.
- La *rl0*[14] cioè il *range* dei 'neuroni' dello strato di ingresso che si vogliono coinvolgere, questo permette, di poter prendere un singolo pixel, oppure di prendere più pixel in serie o con un determinato step.
- La *rl1* cioè il *range* dei neuroni dello strato nascosto che si vogliono coinvolgere, con le stesse considerazioni fatte prima, cioè posso prendere un singolo neurone o più in serie o con step.
- La *rl2* cioè il *range* dei neuroni dello strato di uscita che si vogliono coinvolgere, con le stesse considerazioni fatte prima su come possa funzionare.
- La *err_ range* che indica il *range* su quali bit eseguire l'errore, con le stesse considerazioni fatte prima su come possa funzionare però ovviamente parlando di bit.

La funzione prende dei 'range di neuroni' perché essi indicano gli indici della matrice dei pesi e quindi i pesi in cui voglio inserire gli errori.

Nei test che seguiranno, i vari indici verrano inizializzati per iniettare nella rete un errore alla volta in maniera casuale.

Tutto questo è stato coadiuvato alla scrittura di uno script, che mi consente di eseguire flessibilmente varie operazioni, chiedendomi a terminale come comportarsi. Questo codice è contenuto in *test_ cippo.py*[15], analizziamolo.

2.2.3 Uno script flessibile per eseguire il codice

La prima forma di questo script è stata creata prendendo i comandi mostrati a pagina 29, inserendoli in un file *test_ cippo.py*, così da risparmiare l'inserimento dei vari comandi nella shell di Python ad ogni nuovo allenamento.

Aggiungendo le funzionalità che ho mostrato, ho dovuto integrare nello script il modo in cui usarle:

[14] *r* sta per *range*, *l* sta per *layer* e il *numero* indica lo *strato* della rete.

[15] *Cippo* è semplicemente il mio soprannome

- Aggiungendo l'implementazione del salvataggio, ho dovuto aggiungere la parte del ricaricamento.
- Aggiungendo *do_ err_ w*, ho dovuto pensare ulteriormente che modalità di inserimento di errori implementare e come gestire i test.

Il caricamento di uno stato allenato della rete

All'interno dello script viene chiesto se si voglia ricaricare l'ultimo salvataggio disponibile:

- In caso affermativo, se nella cartella corrente esiste un salvataggio *save* precedente, viene caricato tramite l'uso della funzione *pickle.load()*. Essa ricarica i dati in una variabile *backup* e attraverso gli indici di questa variabile si assegnano i pesi *net.weights* e i bias *net.biases* precedenti.
- In caso contrario viene chiesto se si vuole allenare la rete neurale:
  - Se si risponde affermativamente, si esegue l'allenamento e, al termine di esso, viene chiesto se si vuole salvare tale stato della rete.
  - In caso contrario si esce dal programma.

Modalità di inserimento di errore

Dopo l'allenamento o un caricamento di uno stato della rete precedente, lo script chiede:

1. Se si vogliono iniettare gli *errori transitori*, cioè degli errori singoli nella rete che, una volta inseriti ed eseguito il test della rete, vengono corretti da un ricaricamento dei pesi allo stato iniziale.
   - Se la risposta è affermativa viene chiesto:
     (a) In che blocco inserire gli errori.
     (b) Quanti errori di questo tipo effettuare.[16]
   - In caso contrario si passa agli errori additivi.
2. Se si vogliono iniettare gli *errori additivi*, che non ricaricano lo stato iniziale dei pesi durante l'iniezione sommando i loro effetti.
   - Se la risposta è affermativa allora viene chiesto:
     (a) In che blocco inserire gli errori.
     (b) Quanti errori devono essere aggiunti alla rete.
     (c) Quante volte eseguire questo test.[17]

[16] Chiamo i vari test singoli: *round* di errore.
[17] Anche in questo caso, chiamo *round* i vari test di errore.

- Se la risposta è negativa, non ho implementato ulteriori errori e quindi si esce dal programma.

Inserendo '3' alla richiesta di che blocco testare, il codice assume di dover testare l'intera rete.[18]

Come viene gestito il test degli errori

Durante i test, a terminale vengono mostrati ad *ogni errore inserito*:

- In formato binario a 64 bit (virgola mobile a doppia precisione):
  1. Il valore del peso originale.
  2. Il valore del peso con iniettato l'errore.
- In formato decimale, per capire in maniera semplice come cambia il valore:
  1. Il valore del peso originale.
  2. Il valore del peso con iniettato l'errore.
- Informazioni su:
  1. Il blocco selezionato.
  2. I neuroni selezionati, cioè gli indici della matrice di peso che individuano il singolo peso.
  3. L'indice del bit del numero in formato binario in cui ho iniettato il *bit flip*.
- La prestazione della rete con l'errore iniettato.

In più a seconda della modalità di errore selezionata, si ha:

1. Nel caso di errori transitori vengono mostrati:
   (a) La prestazione media totale dei vari casi di errore.
   (b) La prestazione media totale dei casi di errore con discostamento della accuratezza del 0,01% dal caso in assenza di errori.[19]
   (c) La prestazione minima raggiunta dalla rete durante il test.
2. Nel caso di errori additivi vengono mostrati:

[18] Quindi esegue una scelta random su $784 * 30 + 30 * 10 = 23820$ pesi, se il numero risultante è inferiore a 23519 (contando a partire da 0) allora si selezionano i pesi del primo blocco, in caso contrario si seleziona il secondo blocco. Questo ovviamente avviene ad ogni nuovo ciclo di errore.

[19] Dipende dallo stato di allenamento della rete, nel nostro caso abbiamo una prestazione del 94,97%.

(a) La prestazione media totale dei vari casi di errore.

(b) La prestazione minima raggiunta dalla rete durante il test.

In aggiunta all'uscita a terminale viene eseguito un dump sui file *dump_ err_ transitorio.txt* per gli errori transitori e *dump_ piu_ err.txt* per gli errori additivi.

Essi riportano:

- Il tipo di errore.
- Su quale blocco della rete sono stati iniettati gli errori.
- In caso di errori additivi, il numero di errori che si è scelto di aggiungere.
- Le prestazioni per ogni round che l'utente ha deciso di affrontare.
- Gli stessi valori statistici riportati a terminale a fine dei test.

I file di testo sono molto utili a rivedere i risultati statistici una volta effettuati i test. Facendo così ho potuto effettuare più test e confrontarne i comportamenti.

Spiegato il codice, passiamo al prossimo capitolo che mostra i test affrontati sulla rete.

Capitolo 3

Analisi dell'impatto di errori sui pesi nella rete neurale

3.1 Come ho eseguito i test e i risultati di essi

Come prima cosa, faccio notare che l'allenamento che ho salvato imposta la rete ad una accuratezza del 94,97%.[1]

Riguardo ai test, dovrebbe essere chiaro che non possiamo applicare ogni possibile scenario. Proprio per questo ci siamo fermati ad applicare errori nei pesi, con due possibili implementazioni di errore. Anche usando queste implementazioni non si può pretendere di effettuare ogni caso possibile: bisogna decidere alcuni casi che siano significativi.

Inizialmente il mio relatore e io abbiamo scelto di eseguire questi test:

- 1000 errori singoli random transitori sulla intera rete.
- 2 errori additivi random sulla intera rete per 100 round.
- 10 errori additivi random sulla intera rete per 100 round.

Eseguire questi test può prendere parecchio tempo col mio codice al completo. Questo avviene perché ad ogni errore viene valutata la rete e mostrato a schermo il risultato. Per gli errori singoli transitori è giusto così e poco si può fare. Per gli errori additivi invece commentando il codice[2] in modo che non ritesti la rete ad ogni inserimento di errore ma solo dopo averli inseriti tutti, si può velocizzare di molto l'esecuzione.

Per la questione di non metterci tempo, ho eseguito i test fermandomi a 100 round. Per avere dati ancora più consistenti, test più lunghi danno risultati migliori. Ho comunque eseguito i vari test almeno 3 volte per dare più certezza ai risultati.

[1] La rete in assenza di errori riconosce 9.497 numeri sui 10.000 di test.

[2] Attenzione poiché potrebbe servire anche una reindentazione del codice.

Con il mio codice comunque allego una cartella con i test effettuati.[3]

1000 errori singoli transitori

- Il primo test mi ha restituito:
  1. Una accuratezza media del 94,63%.
  2. Una accuratezza media 'fuori dal range di un 0,01%' del 81,73%.
  3. Una accuratezza minima del 9,80%.
- Il secondo test mi ha restituito:
  1. Una accuratezza media del 94,37%.
  2. Una accuratezza media 'fuori dal range di un 0,01%' del 78,69%.
  3. Una accuratezza minima del 9,74%.
- Il terzo test mi ha restituito:
  1. Una accuratezza media del 94,37%.
  2. Una accuratezza media 'fuori dal range di un 0,01%' del 74,25%.
  3. Una accuratezza minima del 9,80%.

La rete, già con un singolo errore, in alcuni casi (accuratezza minima) sbaglia completamente, arrivando a sbagliare 90% dei numeri.

Guardando le medie su tutti i round, abbiamo che l'accuratezza si discosta meno del 1% dall'accuratezza di partenza.

Facendo la media dei soli test che escono dal range dello 0,01%, notiamo che l'accuratezza è piuttosto bassa: si arriva fino a un 20,72% di distacco dalla accuratezza normale. Questo ci fa capire che, quando la rete non rimane in un certo range dalla sua accuratezza iniziale, in genere sbaglia significativamente (dalle medie totali però possiamo intuire che questo non capiti spesso).

Negli errori additivi abbiamo deciso di non usare più la media 'fuori dal range di un 0.01%' per accellerare l'analisi dei test.

2 errori additivi per 100 round

- Il primo test mi ha restituito:
  1. Una accuratezza media del 94,87%.
  2. Una accuratezza minima del 85,53%.
- Il secondo test mi ha restituito:

[3] Guardare l'appendice A per il link.

1. Una accuratezza media del 94,96%.
2. Una accuratezza minima del 94,59%.

- Il terzo test mi ha restituito:

  1. Una accuratezza media del 94,94%.
  2. Una accuratezza minima del 92,91%.

Analizziamo anche 10 errori additivi e facciamo qualche osservazione.

10 errori additivi per 100 round

- Il primo test mi ha restituito:

  1. Una accuratezza media del 91,25%.
  2. Una accuratezza minima del 9,80%.

- Il secondo test mi ha restituito:

  1. Una accuratezza media del 90,60%.
  2. Una accuratezza minima del 9,80%.

- Il terzo test mi ha restituito:

  1. Una accuratezza media del 92,39%.
  2. Una accuratezza minima del 9,80%.

  Osservo quindi che:

1. I 2 errori additivi, considerando la media totale, non discostano significativamente l'accuratezza dal caso con errore singolo: nei vari test che ho affrontato siamo sul 1% di errore. I valori minimi però sono cambiati, questo è perché si è testato per solo 100 round.[4]

2. I 10 errori additivi fanno perdere alla rete meno del 5% di accuratezza, qui notiamo però che abbiamo di nuovo valori minimi parecchio bassi.[5]

Questi errori, come detto, sono quelli che sono stati scelti di partenza.

Avendo un po' di tempo, ho comunque voluto affrontare ulteriori test per aggiungere altri risultati.

Prima però faccio una considerazione:

- Il numero totale di pesi nel primo blocco[6] è $784 * 30 = 23.520$ pesi.

[4] Affrontando il test con 1000 round ho avuto come minimo di accuratezza l'8,92%, come medio 93.68%, il test è riportato col codice.

[5] Questo perché, testando per 10 errori, anche solo 100 round sono sufficienti ad avere almeno un caso in cui la rete risente molto degli errori iniettati.

[6] Guardare *m_block* a pagina 33 per capire cosa intendo per *blocco*.

- Il numero totale di pesi nel secondo blocco è $30 * 10 = 300$ pesi.
- Sommandoli $23.520 + 300 = 23.820$ e moltiplicando questo numero per 64 (cioè i bit della loro rappresentazione binaria) ottengo $23.820 * 64 = 1.524.480$ bit che rappresentano tutti i bit dei pesi della rete.

Ora voglio applicare un errore ogni 100.000 bit e un errore ogni 10.000 bit per vedere come si comporta la rete. Per fare ciò devo testare di nuovo la rete per 15 e 153 errori additivi[7].

15 errori additivi per 100 round, un errore ogni 100.000 bit

- Il primo test mi ha restituito:
  1. Una accuratezza media del 88,10%.
  2. Una accuratezza minima del 9,80%.
- Il secondo test mi ha restituito:
  1. Una accuratezza media del 91,43%.
  2. Una accuratezza minima del 9,80%.
- Il terzo test mi ha restituito:
  1. Una accuratezza media del 87,19%.
  2. Una accuratezza minima del 9,80%.

Siamo quindi peggiorati un po' dalla condizione con 10 errori, il che è aspettabile, arriviamo a perdere fino a un 8% della accuratezza media della rete.

153 errori additivi per 100 round, un errore ogni 10.000 bit

In questo caso *consiglio vivamente* di commentare i test per ogni iniezione di errore ed effettuarne uno solo a fine iniezione.[8]

- Il primo test mi ha restituito:
  1. Una accuratezza media del 45,22%.
  2. Una accuratezza minima del 9,80%.
- Il secondo test mi ha restituito:
  1. Una accuratezza media del 62,75%.
  2. Una accuratezza minima del 9,80%.

[7] $1.524.480/100.000 \approx 15$, $1.524.480/10.000 \approx 153$.

[8] Con i test intermedi attivi il mio PC ci mette sulle 4 ore, senza test attivi 3-4 minuti.

- Il terzo test mi ha restituito:

    1. Una accuratezza media del 50,95%.
    2. Una accuratezza minima del 9,80%.

Quindi il caso di un errore ogni 10.000 fa crollare l'accuratezza media anche di un 49,75%.

Fatto questo, può essere interessante prendere *i blocchi singolarmente* e testare come errori isolati in essi possano creare problemi alla rete.

Scelgo di testare i blocchi nel caso di un errore ogni 10.000 bit. Per fare questo nel caso del primo blocco applico 151 errori additivi (sempre per 100 round), nel caso del secondo blocco applico 2 errori additivi.[9]

Errori sul primo blocco, un errore ogni 10.000 bit

- Il primo test mi ha restituito:

    1. Una accuratezza media del 59,57%.
    2. Una accuratezza minima del 9,80%.

- Il secondo test mi ha restituito:

    1. Una accuratezza media del 54,24%.
    2. Una accuratezza minima del 9,80%.

- Il terzo test mi ha restituito:

    1. Una accuratezza media del 51,11%.
    2. Una accuratezza minima del 9,80%.

Analizziamo gli errori sul secondo blocco.

Errori sul secondo blocco, un errore ogni 10.000 bit

- Il primo test mi ha restituito:

    1. Una accuratezza media del 87,56%.
    2. Una accuratezza minima del 9,59%.

- Il secondo test mi ha restituito:

    1. Una accuratezza media del 91,61%.
    2. Una accuratezza minima del 9,80%.

[9] Per il primo blocco abbiamo $23.520 * 64 = 1.505.280$ bit, quindi 151 errori additivi circa da testare per raggiungere un errore su 10.000. Nel caso del secondo blocco abbiamo $300 * 64 = 19.200$ bit quindi con 2 errori additivi raggiungiamo tale scopo.

- Il terzo test mi ha restituito:

    1. Una accuratezza media del 86,72%.
    2. Una accuratezza minima del 8,92%.

Da questi test vediamo che, generando errori con probabilità simile, il primo blocco vada ad impattare l'accuratezza della rete in maniera maggiore rispetto al secondo. Per questo, pensando all'implementazione di memorie con correzione di errore, risulterebbe più utile averle sul primo blocco che sul secondo.

Notiamo inoltre come il test sul primo blocco (con 151 errori) sia simile a quello sulla intera rete (con 153 errori). Questo è giusto poichè, come detto più volte, il primo blocco contiene 23.520 pesi mentre il secondo solo 300: il secondo blocco compone di fatto l'1,26% della rete.[10]

Per concludere i test, ho notato che spesso i valori minimi della rete convergono a determinati numeri, tipo 9,80%[11]. Il fatto che questi numeri escano con consistenza mi ha fatto pensare che magari la rete, in seguito ad errori, potesse ricadere in uno stato, dove non gli importasse del vero numero in ingresso ma classificasse quasi tutti i numeri come un particolare numero. Questo mi è venuto in mente perché, tecnicamente, si hanno 10 possibili numeri che la rete può indicare. Se la rete, per un qualche motivo, si 'bloccasse', classificando ogni numero che vede in un solo numero, circa il 10%[12] dei numeri lo dovrebbe azzeccare (supponendo dati di test equamente distribuiti per i 10 numeri).

Al fine di testare questo, ho modificato la *evaluate()* in *network.py*, dove, invece che calcolare l'indice del neurone con l'attivazione più alta, per determinare l'uscita gli passavo direttamente l'indice dell'unico numero che volevo che la rete riconoscesse.

Quindi ho ottenuto che:

1. Indicando come l'attivazione più alta quella del neurone in indice 0 (cioè la rete riconosce solo i numeri 0), ho ottenuto una accuratezza di 9,80%.

2. Indicando come l'attivazione più alta quella del neurone in indice 1 (cioè la rete riconosce solo i numeri 1), ho ottenuto una accuratezza di 11,35%.

3. Indicando come l'attivazione più alta quella del neurone in indice 2 (cioè la rete riconosce solo i numeri 2), ho ottenuto una accuratezza di 10,32%.

[10] $300/23.820 \approx 1,26$.

[11] In altri test che ho fatto, ho trovato altri valori tipo 8,92%, 9,74% etc.

[12] Le accuratezze minime viste prima sembrano stare vicine a questo valore.

4. Indicando come l'attivazione più alta quella del neurone in indice 3 (cioè la rete riconosce solo i numeri 3), ho ottenuto una accuratezza di 10.10%.

5. Indicando come l'attivazione più alta quella del neurone in indice 4 (cioè la rete riconosce solo i numeri 4), ho ottenuto una accuratezza di 9,82%.

6. Indicando come l'attivazione più alta quella del neurone in indice 5 (cioè la rete riconosce solo i numeri 5), ho ottenuto una accuratezza di 8,92%.

7. Indicando come l'attivazione più alta quella del neurone in indice 6 (cioè la rete riconosce solo i numeri 6), ho ottenuto una accuratezza di 9,58%.

8. Indicando come l'attivazione più alta quella del neurone in indice 7 (cioè la rete riconosce solo i numeri 7), ho ottenuto una accuratezza di 10,28%.

9. Indicando come l'attivazione più alta quella del neurone in indice 8 (cioè la rete riconosce solo i numeri 8), ho ottenuto una accuratezza di 9,74%.

10. Indicando come l'attivazione più alta quella del neurone in indice 9 (cioè la rete riconosce solo i numeri 9), ho ottenuto una accuratezza di 10,09%.

Come avevo intuito, ritrovo alcuni valori minimi già visti.

Sommando le percentuali ottengo 100%, cioè l'intero set di test (eliminando la notazione percentuale, i 10.000 numeri scritti a mano).

Quando vediamo un valore minimo con questi valori possiamo supporre che, in seguito a degli errori hardware, la rete pensi di avere in ingresso sempre lo stesso numero.[13]

È interessante notare che ci siano valori minimi discostanti da questi nei nostri test. Questo significa che, in determinati casi, non siamo ricaduti un una rete completamente 'fissata' su un numero.

3.2 Conclusioni

Abbiamo osservato nella sezione 3.1 come varia l'accuratezza della nostra rete in seguito a determinati errori. Abbiamo inoltre notato come anche inserendo pochi errori, in determinati casi, la rete abbia ripercussioni decise sulle proprie prestazioni.

[13] Usando questa interpretazione, sembra che, in seguito agli errori che ho riportato, la rete spesso pensi di vedere uno 0.

Con l'aiuto delle informazioni riportate a terminale durante i test, ci si accorge di quali siano gli errori sui pesi più problematici.

Ricordando la rappresentazione a 64 bit dei numeri a virgola mobile mostrata a pagina 32, abbiamo che:

- Gli errori che si applicano all'esponente, generalmente condizionano considerevolmente l'accuratezza della rete.
- Gli errori che si applicano al bit di segno possono condizionare l'accuratezza della rete, ma ciò dipende dalla grandezza iniziale dei pesi.
- Gli errori nella mantissa generano variazioni normalmente trascurabili nell'accuratezza della rete, in quanto non possono variare di troppo i pesi.

Infatti variare i bit di esponente alza o abbassa considerevolemente in valore assoluto un peso. Questo può influenzare in maniera decisa l'ingresso pesato di un neurone, quindi la sua stessa attivazione e gli strati successivi. Variare il bit di segno varia un peso da positivo a negativo (o viceversa), quindi il suo impatto sulla variazione dell'ingresso pesato dipende dalla grandezza iniziale del peso: se esso è un numero vicino allo zero invertire il segno non genera problemi significativi, diverso è il caso di un peso con valore assoluto grande. Variare i bit di mantissa difficilmente genera errori significativi nell'ingresso pesato, poiché, introdotti tali errori, i pesi generalmente variano poco.

Ci tengo a precisare che, quanto detto, vale per una rete strutturata come descritto nella trattazione: è lecito pensare che, reti neurali con architetture diverse, possano comportarsi in maniera diversa.

Appendice A

Informazioni utili sul codice e sui programmi da usare

Credo sia più utile indicarvi dove poter trovare il vario codice, piuttosto che riportarlo nella sua integrità all'interno di questa tesi. Nella stesura infatti ho inserito qualche comando e ho giusto spiegato il funzionamento del codice (senza inserirlo in mezzo alla trattazione).

Codice iniziale

Il codice di Nielsen per Python 2.7 si può trovare al link:

`https://github.com/mnielsen/neural-networks-and-deep-learning`

Il codice di Michal Daniel Dobrzanski per Python 3.5.2[1] si può trovare al link:

`https://github.com/MichalDanielDobrzanski/DeepLearningPython35`

Mio codice

Il mio codice può essere scaricato dal mio Google Drive:[2]

`https://drive.google.com/file/d/1_YifLfceINlv20oJYLTg9JeUCOxUWAsc/view?usp=sharing`

In caso non risultasse più disponibile, potete chiedermi il codice via email all'indirizzo: `cippo1995@gmail.com`.

[1] Funziona correttamente anche su Python 3.7.3.
[2] Devo ancora guardare GitHub come funziona.

Installazione programmi

Per scrivere ed eseguire il codice ho usato Python 3.7.3, la libreria NumPy, e Windows 10 1809.

Per installare Python basta andare sul suo sito e scaricare il programma nella sezione di download:

`https://www.python.it/download/`

Per l'installazione di NumPy su Windows ho seguito le istruzioni al seguente link:

`https://stackoverflow.com/a/28414059`

Installando la versione compatibile con la mia versione di Python, scaricabile a questo link:

`https://www.lfd.uci.edu/~gohlke/pythonlibs/#numpy`

Al momento della mia installazione NumPy era alla versione 1.16.2.

Pacchetto mnist.pkl.gz

Aggiungo un ulteriore link al pacchetto *mnist.pkl.gz*, indirizza alla stessa fonte da cui Nielsen lo ha preso. Il pacchetto contiene i dati del MNIST sistemati per essere usati efficientemente con pickle.

`http://www.deeplearning.net/tutorial/gettingstarted.html#mnist-dataset`

Informazioni ulteriori sul database MNIST

Se volete informazioni sul database MNIST potete andare a questo link:

`http://yann.lecun.com/exdb/mnist/`

Appendice B

Approfondimento sugli errori della rete neurale

Come detto non ho testato a fondo la questione, comunque volendo inserire gli errori nelle operazioni della rete abbiamo questo schema:

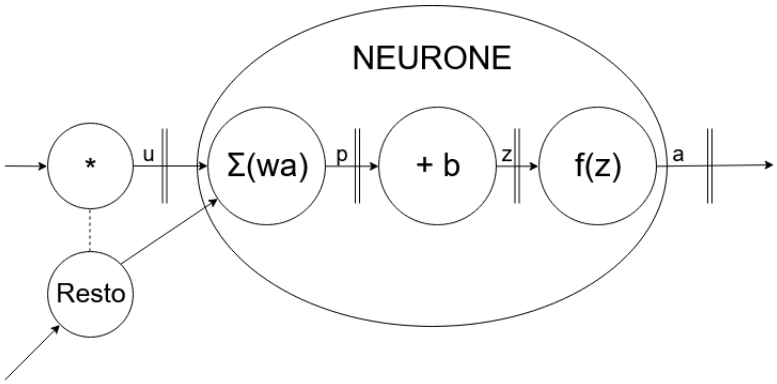

Figura B.1: Operazioni effettuate in una rete neurale.

Quindi possiamo pensare che un errore (indicato dalla doppia barretta) si possa introdurre:

- Dopo la moltiplicazione di una attivazione precedente per il peso, quindi in u.
- Dopo la somma pesata delle attivazioni per i pesi, quindi in p.
- Dopo la somma del bias, quindi in z.
- Dopo l'applicazione della funzione di attivazione, quindi in a.

Gli ultimi tre punti dovrebbero essere 'facilmente' attuabili creando una *feed forward*, modificata.

Feed forward

```
def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a)+b)
    return a
```

Infatti cambiando l'espressione di a, in una più 'modulare':

Feed forward

```
def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        p = np.dot(w, a)
        z = p + b
        a = sigmoid(z)
    return a
```

Ci siamo creati due nuovi vettori, p e z, che possiamo modificare. Ovviamente anche la stessa attivazione può essere modificata dopo la sua applicazione introducendo errori.

Questa procedura però verrebbe fatta ad ogni iterazione. Volendo applicare l'errore in una determinata iterazione (che identificherà determinati strati, pesi e bias), bisognerà aggiungere un contatore che, raggiunta l'iterazione voluta, permette di eseguire una feed forward con l'errore scelto.

Per quando riguarda gli errori dopo la moltiplicazione tra una attivazione e il rispettivo peso, quindi nei valori u, siamo obbligati a crearci un nuovo metodo di calcolo tra matrici.

In *matrixmul.py* abbiamo un metodo che applica il calcolo tra matrici nello stesso modo che lo applichiamo noi a mano, cioè con il metodo *riga per colonna*. È estremamente meno efficiente che usare la moltiplicazione tra matrici *dot* di NumPy, però funziona. Anche in questo caso scegliendo bene gli indici si può inserire un errore dove si vuole.

Spero che queste indicazioni siano utili per eventuali ricerche in questo ambito.

Bibliografia

[1] John J. Hopfield. Neural networks and physical systems with emergent collective computational abilities. 1982.

[2] Frank Rosenblatt. The perceptron a perceiving and recognizing automaton. 1957.

[3] Michael A. Nielsen. *Neural Networks and Deep Learning*. 2015.

[4] R. Rojas. *Neural Networks*. 1996.

[5] Tom M. Mitchell. *Machine Learning*. 1997.

[6] Y. Lecun, Y. Bengio, and G. Hinton. Deep learning. 521:436–444, May 2015.

[7] Wikipedia. Discesa del gradiente. 2019. [Online; ultimo accesso 01-Giugno-2019].