Università degli studi di Ferrara

Dipartimento di Ingegneria
Corso di Laurea in Ingegneria Elettronica e Informatica

Tesi di Laurea Triennale

Fondamenti di Deep Learning: realizzazione di un sito Web di esempi

Candidato: *Alessio Morselli*
Relatore: *Prof. Evelina Lamma*
Correlatore: *Aranud Fadja*

ANNO ACCADEMICO 2016/2017

Contents

Contents

Abstract

In questo lavoro, ho voluto presentare i principi fondamentali di funzionamento del *deep learning*, a partire dalle reti neurali. Il primo capitolo, infatti, presenta come lavorano i neuroni artificiali e le diverse architetture di reti neurali, nonché i diversi paradigmi di apprendimento.

Successivamente, mi sono concentrato sulla spiegazione dei fondamenti teorici delle reti neurali più diffuse, le reti *feedforward* che usano il *supervised learning*. Ho presentato l'algoritmo che permette ad una rete di apprendere, per poi soffermarmi sui problemi che presenta tale processo di apprendimento e su alcuni modi per migliorarlo.

Ho poi descritto brevemente il funzionamento dell'apprendimento per rinforzo, presentando le sue caratteristiche principali. Ho anche descritto sinteticamente l'algoritmo di *learning* maggiormente utilizzato, il *Q-Learning*.

Quindi, ho riportato una breve descrizione del sito Web realizzato per presentare proprio l'argomento del *deep learning* in modo semplificato, soffermandomi su come funzionano le diverse demo che sono affiancate alla parte di teoria.

Per finire, ho voluto riportare una piccola riflessione di confronto tra reti neurali artificiali e reti neurali biologiche, in cui ho sottolineato le loro principali analogie e differenze.

URL del sito: https://alessiomorselli.github.io/DeepLearning/

Chapter 1

Reti neurali

Una rete neurale (artificiale) è un sistema computazionale la cui struttura e il cui funzionamento traggono ispirazione dalle reti neurali biologiche. Ogni rete neurale è composta da più neuroni, che sono organizzati in diversi strati o *layer*.

Ad una rete neurale è affidato un compito, che deve essere portato a termine senza alcun intervento diretto del programmatore: l'idea è infatti quella che le reti neurali dovrebbero essere in grado, similmente ad un sistema nervoso naturale, di apprendere autonomamente la soluzione del problema e di riuscire ad applicarla correttamente al di fuori del processo di apprendimento.

1.1 Neuroni artificiali

Il neurone artificiale è l'unità elementare che compone le reti neurali. Ogni neurone riceve uno o più input da altri neuroni, che vengono pesati e successivamente sommati. A tale valore viene aggiunto una costante, caratteristica del neurone, chiamata *bias*. Infine il neurone applica a questa somma una funzione, il cui risultato viene inviato come output del neurone ad altri neuroni della rete.

Tale funzione viene chiamata funzione di attivazione e può essere scelta in base alle esigenze o al tipo di output che un neurone può produrre. Inizialmente si usava una funzione a gradino e il neurone veniva chiamato *perceptron*. Esso ha la peculiarità di generare come output solo due valori, ovvero zero o uno.

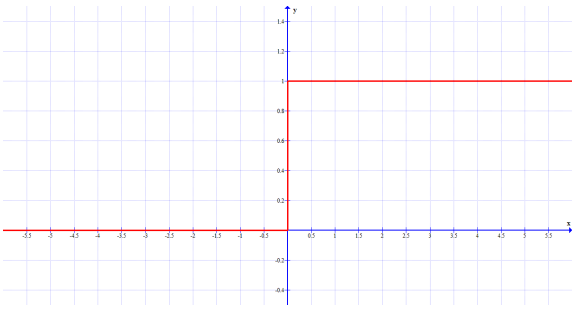

Figure 1.1: Il grafico della funzione gradino: l'output del *perceptron* sarà quindi 0 se la somma pesata degli input più il *bias* è negativa, in caso contrario assumerà il valore 1.

Nonostante si possano costruire reti neurali costituite da soli *perceptron* in grado di calcolare qualsiasi funzione logica, neuroni con un output reale e non intero consente di realizzare reti più flessibili e anche più precise.

Perciò attualmente si usano diverse funzioni di attivazione che siano continue, in modo che l'output del neurone sia un numero qualsiasi compreso tra 0 ed 1. Una delle funzioni più spesso usate è la funzione sigma, che dà il nome ai neuroni che ne fanno uso, detti appunto simgoidi. Il suo uso è diffuso in quanto la sua espressione coinvolge l'esponenziale, una funzione di cui è facile calcolare la derivata.

Talvolta, tuttavia, si preferisce l'uso di neuroni che hanno la tangente iperbolica (*tanh*), in quanto tali neuroni hanno un output sia positivo, che negativo, in quanto può variare tra -1 e 1. Questa caratteristica consente loro di avere performance talora migliori dei neuroni sigmoidi, anche se i miglioramenti sono minimi: infatti, poiché la funzione di attivazione è dispari (simmetrica rispetto all'origine), gli output positivi equivalgono quelli negativi.

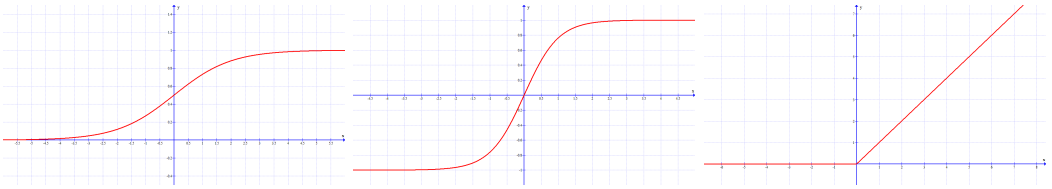

Figure 1.2: Il grafico di alcune delle funzioni più usate come funzioni di attivazione. Da sinistra: la funzione sigma, la funzione tangente iperbolica e la funzione lineare.

1.2 Struttura di una rete neurale

In una rete neurale, i neuroni sono divisi in tre grandi categorie: input, *hidden* e output.

I neuroni di input sono neuroni speciali, che rappresentano l'input della rete. Spesso infatti non si tratta di neuroni, ma nei diagrammi rappresentanti le reti neurali si usano solo come segnaposto, per usare una rappresentazione omogenea degli elementi. Questi neuroni sono caratterizzati solo da un output (coincidente con l'ingresso della rete) e non ricevono alcun input. Costituiscono il *layer* di input.

I neuroni nascosti o *hidden* si trovano nel mezzo della rete e formano gli *hidden layer*, così chiamati appunto perché non sono né di input né di output. Sono la parte centrale della rete e costituiscono il vero fulcro dei calcoli della rete. Solitamente, più *hidden layer* possiede una rete, maggiori saranno le sue potenzialità in quanto potenza di calcolo, ma si incontreranno anche maggiori difficoltà nella fase di allenamento.

Infine, l'ultimo strato è quello di output: l'uscita dei neuroni di questo *layer* rappresenta i risultati della rete.

Esistono principalmente due tipologie di architetture per costruire una rete neurale. La più diffusa è l'architettura detta *feedforward*, in cui i neuroni di un *layer* ricevono gli input dallo strato precedente e mandano il proprio output al *layer* successivo. In questa struttura non sono possibili collegamenti circolari, ma l'informazione attraversa la rete in una solo direzione, dallo strato di input a quello di output.

La seconda architettura, invece, permette collegamenti che formino dei loop: le reti così costruite sono chiamate ricorrenti. Proprio in virtù di tale caratteristica, le reti ricorrenti permettono comportamenti più dinamici. Nonostante ciò, le reti *feedforward* sono state

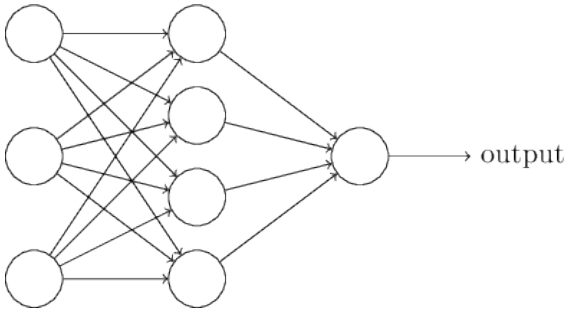

Figure 1.3: La rappresentazione di una rete neurale *feedforward*. Essa è composta da tre neuroni di input, quattro neuroni nascosti e uno solo di output. Come si può vedere, i collegamenti sono solo tra neuroni di *layer* adiacenti, con direzione dall'ingresso verso l'uscita.

più utilizzate, in quanto i loro algoritmi di apprendimento si sono dimostrati maggiormente performanti. Tuttavia, le reti neurali ricorrenti rimangono, da un certo punto di vista, più interessanti, poiché esse lavorano in modo simile a come funziona il cervello umano, più di quanto non facciano le reti *feedforward*.

1.3 Paradigmi di *learning*

Le reti neurali non si differenziano solo per il tipo di funzione di attivazione e per la loro architettura, ma anche per la tipologia di apprendimento a cui vengono sottoposte.

Esistono diversi metodi di apprendimento, che si differenziano principalmente per i dati di allenamento che ha a disposizione la rete in ingresso. L'apprendimento si dice supervisionato (*supervised learning*) se i dati di allenamento sono già catalogati, ovvero il supervisore che allena la rete conosce già a priori l'output che la rete dovrebbe mostrare quando riceve in ingresso ognuno dei dati di allenamento. Usando quindi l'output desiderato come misura della bontà del proprio lavoro, la rete aggiusta pesi e *bias* in modo di avvicinarsi sempre di più alle uscite indicate dal supervisore. Il termine che misura le prestazioni della rete è l'accuratezza con cui riesce a trovare le soluzioni corrette. La rete, quindi, dovrebbe essere in grado, una volta allenata sui dati di allenamento, di generalizzare ciò che ha appreso e applicare le soluzioni anche su input non facenti parte di quelli di allenamento.

Una possibile applicazione di questo metodo è la classificazione di dati, come ad esempio il riconoscimento delle cifre scritte a mano.

Viceversa, l'apprendimento non supervisionato (*unsupervised learning*) prevede invece di fornire alla rete input non catalogati, con l'unico suggerimento da parte del programmatore di alcune categorie di dati. Compito della rete sarà raggruppare in modo corretto i dati riconoscendo le caratteristiche comuni, non solo usando le categorie inizialmente suggerite, ma correggendo quelle ritenute sbagliate e aggiungendone nel caso ne avesse necessità. Il risultato finale saranno quindi diversi gruppi di dati, in ognuno dei quali si trovano input con caratteristiche simili tra loro. In questo caso, le prestazioni non possono essere misurate usando l'accuratezza, in quanto non si conoscono gli output relativi ai dati di allenamento.

L'apprendimento non supervisionato è utilizzato, ad esempio, nella ricerca dei motori

di ricerca, dove, dati i link delle pagine web, essi vanno raggruppati per le parole chiave che contengono.

Mentre questi due metodi di apprendimento prevedono dati statici, l'apprendimento per rinforzo (*reinforcement learning*) è utile nel caso in cui si abbia a che fare con un ambiente dinamico, in continuo mutamento. Nel *reinforcement learning*, il protagonista dell'azione è un agente, che non è altro che una macchina a stati: esso infatti si muove da uno stato all'altro, usando le azioni che ha a disposizione in ogni stato in cui si trova. Tale agente è controllato da una rete neurale. Questo tipo di apprendimento aggiunge il concetto di ricompensa (chiamato anche "rinforzo", da cui il nome "apprendimento per rinforzo"): l'obiettivo della rete neurale è stabilire quale azione in ogni stato non solo cerca di aumentare la ricompensa nel prossimo stato, ma che garantisce all'agente la maggior ricompensa possibile dopo un numero finito di mosse.

Il *reinforcement learning* è stato applicato con successo, per esempio, nell'addestrare una rete neurale a giocare ad alcuni giochi dell'Atari 2600, usando come unico input i pixel dello schermo, che rappresentano un ambiente d'azione costantemente dinamico.

Chapter 2

Reti neurali *feedforward* nel *supervised laerning*

Le reti *feedforward* sono le reti più studiate, in quanto la loro struttura, insieme al loro funzionamento, è concettualmente semplice e non richiedono grandi capacità di calcolo per essere allenate. Combinate con il metodo del *supervised learning* permettono quindi di realizzare compiti piuttosto complessi, come il riconoscimento di immagini, anche su macchine non particolarmente potenti. Inoltre consentono di ottenere accuratezze molto elevate dopo poche epoche di allenamento.

2.1 Algoritmo del gradiente decrescente

Per far sì che una rete neurale possa avere l'output corretto dato un input, i suoi parametri, ovvero pesi e *bias*, dovranno avere valori tali da garantire un'uscita della rete corretta. In primo luogo, si deve definire una quantità che mi permetta di definire quanto la rete sia lontana dal risultato corretto. Questo compito è svolto dalla funzione costo, una funzione che dipende da tutti i pesi e da tutti i *bias* della rete. La funzione costo, per essere considerata tale, deve rispettare anche due proprietà fondamentali: deve essere sempre non negativa e, se il neurone fornisce un'uscita simile a quella desiderata, deve assumere un valore piccolo (vicino allo zero).

L'obiettivo della rete è aggiustare i propri parametri per minimizzare la funzione costo e quindi avvicinarsi al valore d'uscita desiderato per gli input di allenamento. In altre parole, si deve calcolare il minimo della funzione costo. Poiché le variabili coinvolte sono molto numerose, trovarlo in maniera analitica è impensabile, in quanto risulterebbe computazionalmente pesante.

Si usa quindi l'algoritmo del gradiente decrescente, che consiste nel:

1. Calcolare il gradiente della funzione costo, che ci dice come essa cresce in funzione di ogni peso e di ogni *bias* della rete:

$$\nabla C = \frac{1}{n} \sum_X \nabla C_x$$

2. Si stabilisce un passo (la cui lunghezza è determinata dal tasso di apprendimento η) di variazione di ogni peso e *bias*, in modo tale che la rispettiva derivata sia negativa:

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$

vogliamo quindi far variare il peso/*bias* in modo tale che la funzione costo diminuisca;

3. Determinare quindi la nuova funzione costo:

$$C = \frac{1}{n} \sum_X C_x$$

ovvero calcoliamo la funzione costo per ogni input e quindi ne facciamo la media, ricavando la funzione costo complessiva. Applicando l'operazione gradiente, tale espressione non cambia: troviamo il gradiente della nuova funzione costo calcolando la media dei singoli gradienti per ogni input, come al passo 1, e cominciamo una nuova iterazione.

Il problema di questo algoritmo è proprio nel calcolo del gradiente: spesso, il numero di dati di ingresso è molto elevato e il tempo di computazione può diventare davvero molto elevato, rendendo inutilizzabile questo metodo per un'applicazione pratica. Si è perciò pensato di applicare il procedimento appena descritto non su tutti i dati di allenamento, bensì solo su un gruppo di essi, chiamato *mini-batch* composto da dati scelti casualmente tra quelli disponibili.

Questo nuovo algoritmo, detto del gradiente stocastico decrescente, aggiunge quindi un passo nel quale viene scelto il *mini-batch*. Si completa un'iterazione dell'algoritmo quando si sono usati tutti i dati di input e si è completata quindi un'epoca di allenamento. Quindi si riprende dall'inizio con una nuova epoca di allenamento.

L'algoritmo del gradiente stocastico decrescente da solo non è tuttavia sufficiente alla rete per imparare a trovare gli output corretti. Infatti in questo modo non si riesce a conoscere l'errore che commette ogni neurone, rendendo difficile sistemare in modo corretto i pesi e i *bias* per i *layer* nascosti della rete. Si deve affiancare un altro strumento, che prende il nome di *backpropagation*.

2.2 *Backpropagation*

La *backpropagation* (letteralmente "propagazione all'indietro") è un modo di calcolare l'errore che si basa sull'idea di propagare tale errore dall'uscita all'ingresso della rete. Infatti, una rete che si alleni con l'algoritmo del gradiente decrescente combinato con la *backpropagation* impara proprio così: dato un input di allenamento, lo propaga fino all'uscita, dove confronta il proprio output con quello desiderato dal supervisore; una volta fatto questo, tramite l'uso del gradiente, corregge i pesi e i *bias* scorrendo i neuroni dal *layer* di output fino a quello di input, calcolando l'errore di ogni strato con le equazioni della *backpropagation*.

La *backpropagation*, oltre a fornire un modo per calcolare l'errore di ogni neurone di ogni strato della rete, consente anche di calcolare le derivate parziali della funzione costo rispetto a pesi e *bias*.

L'algoritmo di *backpropagation* si basa principalmente su quattro equazioni fondamentali che vengono poi usate durante il suo svolgimento:

1. La prima equazione lega l'errore del *layer* di output con due termini: la derivata della funzione costo rispetto all'output dello strato di uscita, che misura quanto velocemente il costo sta cambiando rispetto all'uscita della rete, e alla derivata della

funzione di attivazione, che indica la velocità di crescita o decrescita in base agli input pesati che arrivano all'ultimo *layer* della rete.

Questo errore è tanto più grande, quindi, tanto più siamo lontani dal minimo della funzione costo e quindi tanto più l'output della rete è diverso da quello desiderato.

2. La seconda equazione è quella da cui deriva l'idea della propagazione all'indietro dell'errore: infatti esprime l'errore di uno strato della rete in funzione dell'errore dello strato successivo. Si ha quindi un movimento dall'uscita verso l'ingresso.

   Combinando le prime due equazioni, si possono calcolare gli errori di tutti gli strati.

3. La terza equazione esprime la relazione tra la derivata parziale del costo rispetto ad un *bias* e l'errore del neurone corrispondente: in particolare, questi due valori sono esattamente identici.

4. Infine, la quarta equazione dice che la derivata parziale del costo rispetto ad un peso è pari al prodotto tra l'errore di un neurone e la sua attivazione.

   Da quest'ultima equazione in particolare deriva il fatto che un neurone con un output molto basso impara molto lentamente, ovvero i pesi cambiano in maniera molto lieve a causa del gradiente decrescente. Ma non solo: ad esempio, usando un neurone sigmoide, che usa quindi la funzione sigma come funzione di attivazione, con attivazioni molto basse (quasi nulle) o molto alte (molto vicine a 1), si avrà un errore pressoché uguale a zero poiché il grafico della funzione sigma tende a diventare orizzontale (si veda la seconda equazione). In questo caso, si dice che il neurone si è saturato. La saturazione avviene sempre se il grafico della funzione di attivazione tende ad essere orizzontale (con derivata nulla, quindi).

Una volta definite queste relazioni si può applicare l'algoritmo della *backpropagation*, che consiste di cinque passaggi:

1. Si calcola l'attivazione per il *layer* di input per un dato ingresso;
2. Si applica quindi il procedimento di *feedforward* e si calcolano a ruota le attivazioni di tutti i neuroni, fino all'uscita della rete;
3. Si usa la prima equazione per calcolare l'errore dell'uscita;
4. Si propaga l'errore all'indietro, fino al primo *hidden layer*: è questo il fulcro della *backpropagation*;
5. Infine si applica il gradiente usando la terza e la quarta equazione.

Combinando quindi la *backpropagation* con l'algoritmo del gradiente stocastico decrescente la rete può aggiustare i propri pesi e quindi imparare a restituire i risultati corretti. Si ottiene quindi l'algoritmo spesso usato per allenare reti *feedforward* che usano il paradigma dell'apprendimento supervisionato:

1. Si sceglie il *mini-batch* per l'iterazione corrente;
2. Per ogni dato di allenamento del *mini-batch* si applica l'algoritmo della *backpropagation*;

3. Si applica il gradiente decrescente e si aggiornano i parametri della rete secondo la regola:

$$w_k \to w'_k = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \to b'_l = b_l - \eta \frac{\partial C}{\partial b_l}$$

Tale procedimento si applica per ogni *mini- batch* di un'epoca, quindi si continua finché la rete non mostri di aver aggiustato i pesi e i *bias* nella maniera corretta. Si può tenere d'occhio il valore della funzione costo, ma spesso si osserva l'accuratezza della rete per determinare se ha imparato o meno, ovvero si osserva la percentuale di dati di allenamento per cui la rete dà un'uscita corretta. L'accuratezza, tuttavia, non è misurata solo sugli ingressi usati per allenare la rete, ma anche su un gruppo di dati chiamati di test: tali input hanno spesso una sorgente diversa da quelli di allenamento, in modo tale che la rete abbia a che fare con dati che non ha mai visto, e servono esclusivamente per valutare le prestazioni della rete in termini di accuratezza.

2.3 Migliorare l'apprendimento di una rete neurale

Durante lo sviluppo delle reti neurali e il loro studio si sono rilevati diversi problemi, che rallentano l'apprendimento della rete o diminuisce le sue prestazioni in termini di accuratezza. Tutt'oggi, sia i problemi, sia le soluzioni non hanno un fondamento teorico solido. Quindi, ci si basa spesso solo sui risultati che si ottengono dalla pratica: anche le soluzioni sono state in molti casi trovate con metodi euristici, provando diverse idee direttamente nella pratica, senza una dimostrazione teorica alle spalle.

Nonostante ciò, si sono fatti molti progressi nel campo delle reti neurali e le prestazioni nel tempo sono andate sempre aumentando, usando diverse tecniche che si sono messe a punto nel corso delle ricerche.

2.3.1 La scelta della funzione costo

La funzione costo ricopre un ruolo fondamentale nel processo di apprendimento della rete. Infatti, compare sia nell'algoritmo del gradiente decrescente che nella *backpropagation* e può influenzare la velocità di apprendimento, in particolar modo con l'espressione delle sue derivate parziali.

Molto frequentemente, infatti, in tale espressione compare la derivata della funzione di attivazione. Ed è proprio questo termine a dare origine molto spesso a rallentamenti nell'apprendimento della rete. Può accadere infatti che gli input ricadano in zone dove il grafico dell'attivazione risulta orizzontale o quasi, annullando la derivata e diminuendo di molto la velocità di cambiamento dei pesi e dei *bias*.

Si può quindi pensare di scegliere una funzione costo in modo tale che, derivandola, il termine di derivata dell'attivazione venga eliso. Questa funzione è stata individuata nella funzione di *cross-entropy*. Tale funzione rispetta le due condizioni necessarie a interpretarla come funzione costo e, in più, evita il fenomeno del rallentamento, in quanto nell'espressione della sua derivata è mancante la derivata dell'attivazione.

Non solo, possiede anche l'ottima proprietà di avere le derivate parziali rispetto a pesi e *bias* proporzionali all'errore commesso dalla rete: quindi, più la rete sbaglia, più velocemente i suoi parametri cambiano, rendendo di fatto l'apprendimento più veloce. Tali

caratteristiche, inoltre, non dipendono da come è stato scelto il tasso di apprendimento, che potrà quindi essere deciso in maniera completamente indipendente dalla funzione costo.

In generale, la funzione di *cross-entropy* è quasi sempre la scelta migliore: infatti, quando si inizializza una rete spesso si scelgono pesi e *bias* in maniera casuale e perciò può accadere che la rete si sbagli di molto per alcuni input inizialmente. Grazie alle proprietà della *cross-entropy* si avrà quindi una fase iniziale di apprendimento molto veloce.

Si può pensare, per finire, alla funzione di *cross-entropy* ad una misura di "sorpresa": infatti, intuitivamente, è come se tale funzione misurasse misurasse la "sorpresa" della rete quando impara l'uscita desiderata per un dato input.

2.3.2 *Overfitting* e sovrallenamento

L'obiettivo di una rete neurale è massimizzare l'accuratezza dei suoi risultati e quindi raggiungere la massima percentuale di risposte corrette. Tuttavia, capita spesso che tale accuratezza smetta di crescere durante l'allenamento ed inizi ad oscillare. La principale causa di questo fenomeno è molto spesso l'*overfitting*: durante l'allenamento la rete non generalizza in maniera efficace durante le prove sui dati di test e le sue prestazioni ne risentono.

L'*overfitting* è un problema molto frequente nell'allenamento delle reti neurali e si deve prevenire per ottenere delle buone prestazioni. Per evitare il sovrallenamento serve un modo per individuare quando tale fenomeno inizia, in modo che si possa fermare il processo di allenamento. Un modo intuitivo per farlo è tenere traccia dell'accuratezza sui dati di test mentre la rete si allena: quando essa smette di migliorare, si ferma l'allenamento. Un altro modo è usare lo stesso metodo, ma non sui dati di test, ma su quelli di validazione. Questi dati sono un gruppo di input scelti tra quelli di allenamento, ma il cui scopo è verificare l'accuratezza della rete. Sono a metà tra i dati di allenamento, che servono per, appunto, allenare la rete ed aggiustare i suoi parametri, e quelli di test, che verificano solo le prestazioni della rete. Poiché, però, i dati di validazione derivano da quelli di allenamento, si deve guardare quando l'accuratezza su questi input satura, ovvero quando smette di crescere. Questo metodo viene chiamato *early stopping*.

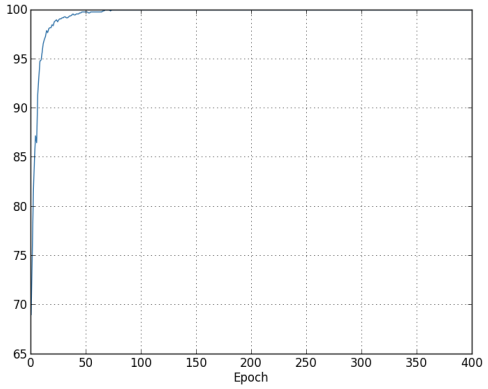

Figure 2.1: Il grafico dell'accuratezza sui dati di validazione. Come si vede, intorno all'epoca 70-80, l'accuratezza raggiunge il 100%: l'accuratezza si è saturata e lì ha inizio l'*overfitting*.

Questo procedimento fa parte di una strategia più generale che sfrutta i dati di validazione per valutare differenti scelte degli iper-parametri. Infatti, usando i dati di test, potrebbe accadere che questi parametri finiscano per abituarsi alle peculiarità di questo set di dati e quindi non generalizzerebbero più efficacemente altri dati. Usando i dati di validazione, invece, si decide un valore per gli iper-parametri e poi si svolge una valutazione finale con i dati di test. Questo metodo è chiamato di *held out*, in quanto i dati di validazione sono presi e separati dai dati di allenamento.

2.3.3 Regolarizzazione

Queste non sono le uniche soluzioni per ridurre l'*overfitting*. Un'alternativa è ridurre la complessità della rete e quindi togliere neuroni e *hidden layer*. Più semplice è la rete, minore è la probabilità che si verifichi il sovrallenamento, ma ha anche meno potenzialità di calcolo di reti complesse.

Esistono anche diverse tecniche che permettono di ridurre gli effetti dell'*overfitting* senza dover per forza rinunciare alla complessità della rete. Si tratta delle tecniche di regolarizzazione. Principalmente, tali tecniche consistono nel sommare alla funzione costo un termine dipendente dai pesi della rete:

$$C = C_0 + \lambda R(w)$$

C_0 è la funzione costo originale. Ad essa, viene aggiunto un termine composto da una funzione di regolarizzazione moltiplicato per il tasso di regolarizzazione λ. In questo contesto, la regolarizzazione deve essere vista come un compromesso tra la minimizzazione del costo e trovare dei pesi piccoli. In particolare, è λ a stabilire quale dei due obiettivi è preponderante durante l'allenamento: se il tasso di regolarizzazione è piccolo, si preferirà minimizzare il costo; viceversa, se λ è grande, si tenderanno a mantenere bassi i valori dei pesi.

Una rete che ha dei piccoli pesi tende a generalizzare meglio e ad imparare gli schemi ricorrenti nei dati di allenamento. Infatti il suo comportamento non sarà affetto dalle variazioni nei dati di allenamento, anche se alcuni di essi dovessero cambiare: reti regolarizzate riconoscono gli schemi, ma sono resistenti ad imparare le peculiarità dei dati, diminuendo le probabilità che si verifichi *overfitting*.

La regolarizzazione è uno di quegli aspetti delle reti neurali che non si ha ancora ben chiaro: la pratica mostra miglioramenti notevoli nella generalizzazione durante il suo uso, ma non si ha ancora una conoscenza sistematica soddisfacente che spieghi il perché di questi miglioramenti. In realtà, il problema è ben più profondo, in quanto non si sa ancora spiegare esattamente come funziona la generalizzazione. Al momento, la regolarizzazione viene largamente usata per via dei benefici che apporta all'apprendimento, ma basandosi solo sulle prove empiriche per usarla in modo corretto.

Esistono, oltre alla regolarizzazione sulla funzione costo, altre due tecniche che riguardano una la struttura della rete, l'altra il set di dati di allenamento.

La prima è la tecnica di *dropout*: generalmente, per allenare una rete si usa l'algoritmo della *backpropagation* insieme a quello del gradiente decrescente, ma in questo caso prima di applicare tali procedimenti si va a cancellare una certa percentuale di neuroni in ogni *hidden layer* della rete (scelti casualmente), lasciando intoccati gli strati di input e di output. Tale procedimento viene ripetuto ad ogni *mini-batch*, dopo aver aggiornato i parametri della sotto-rete. In questo modo, si va ad operare con reti meno complesse, che sono meno soggette a subire gli effetti dell'*overfitting*. Quando la rete ha terminato la fase di allenamento, viene presa nella sua completezza, modificando in modo appropriato i pesi

(ad esempio, se si cancellassero metà dei neuroni ad ogni epoca, i pesi verrebbero dimezzati durante la fase di funzionamento della rete). La tecnica del *dropout* può essere vista anche come se si stessero facendo lavorare tante reti diverse, per poi considerare come risultato la media di tutti gli output delle diverse reti virtuali. Tale processo spesso si rivela vincente nell'evitare l'*overfitting*, poiché reti diverse si sovrallenano in momenti e in modi diversi.

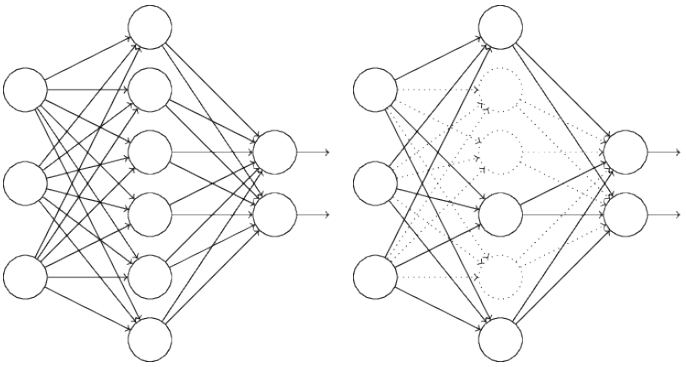

Figure 2.2: La figura mostra come funziona il *dropout* su una semplice rete: a sinistra, la rete completa, a destra la rete dopo il *dropout*. Il procedimento viene ripetuto ogni volta che il *mini-batch* di dati viene cambiato, quindi ad ogni iterazione dell'algoritmo di apprendimento.

La seconda tecnica consiste invece nell'ampliare il set di allenamento. Tuttavia, aggiungere nuovi dati risulta spesso molto costoso o, addirittura, impossibile da metter in pratica. Perciò si ricorre ad un ampliamento artificiale del set, ovvero creando nuovi dati da quelli già in possesso: si applicano su questi diverse operazioni che non ne modificano la natura, ma che riflettono quelle che sono le variazioni che si possono trovare nella pratica. Un esempio, nel riconoscimento di immagini, si possono applicare alle immagini di input piccole rotazioni o distorsioni per ottenere nuovi dati da fornire alla rete.

Figure 2.3: Nonostante la differenza sia minima, per l'analisi svolta dalla rete sono due immagini completamente diverse.

2.3.4 Inizializzazione di pesi e *bias*

I valori iniziali dei parametri della rete influenza come la rete impara e il tempo che l'accuratezza impiega a raggiungere il suo massimo. Diventa quindi importante inizializzare in modo intelligente pesi e *bias* per consentire alla rete di imparare più velocemente.

Un approccio iniziale può essere quelli di scegliere i valori di pesi e *bias* usando variabili aleatorie gaussiane indipendenti, normalizzate in modo da avere media 0 e deviazione standard 1. Questo metodo non è il migliore, però. I neuroni nascosti così risultanti, infatti, saranno maggiormente soggetti a saturazione e il loro apprendimento sarà rallentato.

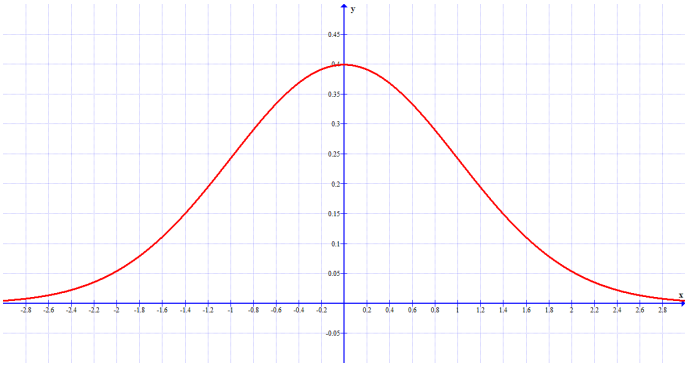

Figure 2.4: Il grafico della distribuzione di probabilità gaussiana. Come si vede, è più probabile avere parametri il cui valore è compreso tra -1 e 1.

Mentre la scelta di una buona funzione costo permette di evitare la saturazione nello strato di output, l'inizializzazione dei parametri consente i neuroni nascosti di saturare meno facilmente.

Una buona inizializzazione prevede di cambiare solo il modo con cui si scelgono i pesi: supponendo di avere un neurone con n_in input pesati: tali pesi andranno inizializzati con una variabile aleatoria gaussiana con media 0 e deviazione standard pari a $\frac{1}{\sqrt{n_in}}$.

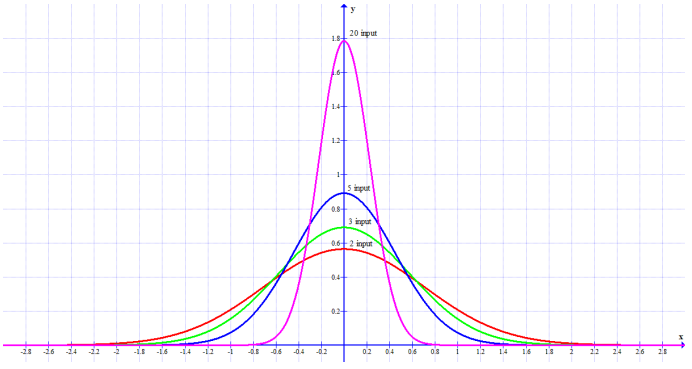

Figure 2.5: Diversi grafici della gaussiana al variare di n_in. Man mano che tale numero sale, il grafico si stringe e si alza, aumentando la probabilità che i pesi assumano valori sempre più vicini allo 0.

Il modo di inizializzare i *bias* invece non influenza più di tanto la velocità di apprendimento.

Inizializzare in modo diverso i pesi, quindi, permette alla rete di raggiungere più velocemente il massimo dell'accuratezza, ma spesso non cambia il valore di tale massimo,

ovvero non va a migliorare la performance finale della rete, benché esistano alcuni casi in cui può accadere che la rete migliori anche l'accuratezza.

2.3.5 La scelta degli iper-parametri

Pesi e *bias* non sono gli unici parametri che influenzano l'apprendimento di una rete neurale. Ci sono anche molti parametri secondari, che vengono detti iper-parametri, che vanno a modificare sia la velocità che le performance della rete. Perciò, scegliere i parametri corretti si rivela fondamentale se si vogliono massimizzare le prestazioni. Una strategia efficace nella scelta dei valori per questi parametri consiste nel semplificare la rete e quindi provare diversi valori di iper-parametri: questa tecnica (detta *broad strategy*) consente quindi di velocizzare la fase di verifica. Si ripete quindi il procedimento, raffinando sempre di più i valori selezionati, con reti sempre più complesse, fino a tornare alla rete originale.

Uno degli iper-parametri fondamentali è sicuramente il tasso di apprendimento η, che determina quanto stiamo cambiando pesi e *bias* ad ogni iterazione dell'algoritmo di apprendimento: troppo grande e i passi saranno talmente ampi da superare il minimo del costo; troppo piccolo e l'algoritmo del gradiente decrescente diventa troppo lento.

Durante l'allenamento, un buon approccio prevede di variare il tasso di apprendimento. Per prima cosa, si determina il suo valore iniziale: si stima inizialmente una valore-soglia (anche solo semplicemente l'ordine di grandezza) per il quale il costo inizia subito a diminuire; successivamente, si aggiusta il valore e si inizia l'allenamento. Poiché è più probabile che i pesi siano molto sbagliati all'inizio, è meglio mantenere un valore piuttosto alto del tasso di apprendimento, per poi abbassarlo durante lo svolgimento per raffinare meglio i valori di pesi e *bias*. Un approccio naturale è usare una tecnica analoga all'*early stopping*: si mantiene η costante fino a che l'accuratezza sui dati di validazione non inizia a diminuire; a quel punto si diminuisce di una piccola quantità il tasso di apprendimento e si ripete il procedimento.

Poiché lo scopo primario del tasso di apprendimento è controllare la grandezza del passo di allenamento, si usa la funzione costo per determinare il suo valore iniziale e non la tecnica di *held-out*, ovvero valutando le performance della rete sui dati di validazione.

Un altro iper-parametro è il numero di epoche, ovvero per quanto tempo la rete debba essere allenata. In questo caso, determinarlo non risulta complicato: si usa la tecnica dell'*early stopping* e quando l'accuratezza smette di crescere, si ferma l'allenamento. L'accuratezza, però, spesso oscilla anche quando il suo trend è in crescita: bisogna terminare il processo di apprendimento quando l'accuratezza massima non viene superata per qualche epoca.

Anche la grandezza del *mini-batch* influisce sulle performance. Può sembrare che usarne uno più largo aumenti la velocità di apprendimento, ma si deve fare attenzione, in quanto se lo si considera eccessivamente grande i pesi non saranno aggiornati abbastanza frequentemente. Per determinare, quindi, il valore migliore, si provano diverse misure del *mini-batch*, graficando sempre l'accuratezza della rete, e si sceglie quella che dà i miglioramenti più rapidi. Durante queste prove si usano valori accettabili per gli altri iper-parametri, attuando anche la variazione del tasso di apprendimento.

L'ultimo iper-parametro più importante è il tasso di regolarizzazione λ, che determina se ha maggior peso la funzione costo o il termine di regolarizzazione. Prima di determinare λ, bisogna prima di tutto determinare il tasso di apprendimento senza applicare regolarizzazione ($\lambda = 0$). Usando quindi tale valore per η si usano quindi i dati di validazione per determina un buon valore del tasso di regolarizzazione, iniziando da $\lambda = 1$ e quindi aumentando o diminuendo l'ordine di grandezza, sempre per migliore l'accuratezza della rete.

Una volta fatto, si raffina il suo valore, quindi si trova un nuovo tasso di apprendimento applicando la regolarizzazione appena calcolata.

Nonostante questi metodi funzionino e siano anche stati automatizzati con successo, l'ottimizzazione degli iper-parametri rappresenta ancora uno scoglio per la comprensione completa delle reti neurali, in quanto non è ancora un problema che è stato completamente risolto.

2.3.6 Variazioni sull'algoritmo del gradiente decrescente

Analizzando a fondo l'algoritmo del gradiente decrescente si è scoperto che può essere migliorato se, nell'approssimazione di Taylor della funzione costo, si tiene conto anche del termine di secondo grado. In particolare, durante un passo Δw si ha:

$$C(w + \Delta w) \approx C(w) + \nabla C + \Delta w + \frac{1}{2}\Delta w^T H \Delta w$$

La matrice H è la matrice Hessiana, che incorpora informazioni su come cambia il gradiente: una sorta di derivata seconda. La matrice dà anche il nome all'algoritmo, detto tecnica Hessiana. Infatti se si sceglie un passo pari a:

$$\Delta w = -\eta H^{-}1 \nabla C$$

la funzione costo si muove verso il proprio minimo. L'algoritmo prevede quindi di applicare tale passo per trovare il costo minimo. Tuttavia, tale metodo è di difficile applicazione pratica: il problema risiede nelle eccessive dimensioni della matrice Hessiana, che rallentano considerevolmente i calcoli.

Da tale intuizione è però stato tratto un algoritmo, che unisce la tecnica Hessiana all'algoritmo del gradiente decrescente: si tratta dell'algoritmo del gradiente decrescente basato sul momento, che vuole sfruttare il vantaggio di incorporare informazioni sia sul gradiente che su come cambia, evitando matrici di secondo grado eccessive.

Tale metodo introduce due concetti fondamentali: una nozione di "velocità" per il parametro che si sta ottimizzando (il gradiente ne modificherà non più la posizione, ma la velocità) ed una sorta di "attrito", che riduce la velocità.

Quindi si introduce una velocità associata a ciascun peso della rete e si seguono quindi questi cambiamenti:

$$v \to v' = \mu v - \eta \nabla C$$

$$w \to w' = w - v'$$

dove μ viene chiamato coefficiente del momento e controlla l'attrito. I due suoi valori limite sono 1 e 0. Quando vale 1 non c'è attrito e la velocità aumenta solamente controllata dal gradiente: quindi ci si muove sempre più velocemente verso il minimo e l'algoritmo si rivela più rapido di quello del gradiente decrescente, anche se una volta raggiunto il minimo, lo supereremo sicuramente. Il valore 0 corrisponde all'attrito massimo e si ricade nel caso dell'algoritmo decrescente. Usando quindi un valore tra 0 e 1 possiamo aumentare la velocità in modo controllato, riducendo quindi il rischio che si superi il minimo una volta raggiunto. Un valore accettabile di μ può essere determinato tramite *held-out* in modo simile agli altri iper-parametri.

2.3.7 Difficoltà di allenamento

È appurato che reti maggiormente complesse hanno una maggiore capacità di calcolo rispetto a reti semplici: infatti, se una rete possiede due o anche più *hidden layer* è capace di di creare astrazioni sempre maggiori. Ma reti più complesse portano con sé anche un problema non indifferente: i diversi strati imparano a velocità. In particolare, quando gli ultimi strati stanno imparando molto velocemente,i primi procedono in maniera molto lenta durante l'allenamento. Viceversa, se sono i primi *layer* ad allenarsi rapidamente, gli ultimi faticano a stare al passo.

Questo perché c'è una instabilità associata all'apprendimento tramite gradiente decrescente nelle reti più complesse, ovvero con più *hidden layer*. Ed è proprio il gradiente a generare tale instabilità: quando sono i primi strati ad essere lenti si parla di problema del gradiente evanescente, viceversa il problema prende il nome di gradiente divergente. Il gradiente tende, quindi, a diventare, rispettivamente, molto basso o molto alto nei primi strati.

Il problema risiede, in particolare, nel calcolo del gradiente per un neurone. Tale calcolo consiste in un prodotto di diversi valori: compare la derivata della funzione di attivazione per ogni neurone successivo nella rete (compreso il neurone preso in considerazione), un peso per ogni collegamento della rete e un termine che rappresenta la funzione costo nel *layer* di output. Considerando un neurone sigmoide, per esempio, troviamo che la sua derivata assume questa forma:

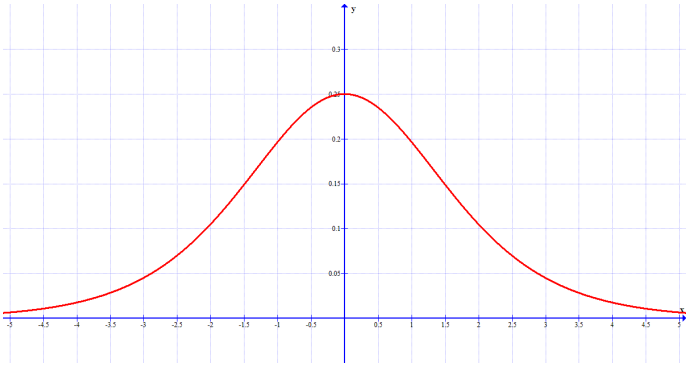

Figure 2.6: L'andamento della derivata della funzione sigma (immagine 1.2, primo grafico).

Poiché il suo valore non supera mai 0.25 e poiché anche i pesi sono quasi sempre compresi tra -1 e 1, risulta palese che moltiplicando tutti termini minori di 1 otterremo un valore sempre più piccolo. E poiché l prodotto prevede sempre più fattori man mano che si procede verso l'ingresso della rete, il gradiente dei primi neuroni tenderà ad essere sensibilmente minore rispetto a quello degli ultimi neuroni (problema del gradiente evanescente). Se anche si facesse in modo che un peso compensasse il valore della derivata dell'attivazione, si otterrebbe il problema inverso: il gradiente continuerebbe ad aumentare esponenzialmente ad ogni strato (problema del gradiente divergente). Inoltre, in questo caso, bisogna tenere conto del fatto che anche la derivata dell'attivazione dipende dal peso: aumentando troppo il peso tale derivata potrebbe diventare ancora più piccola. Bisogna quindi essere molto cauti quando si opera sui pesi, per fare in modo di non diminuire troppo

il valore della derivata. Questo spiega anche il perché il problema del gradiente evanescente è più frequente di quello divergente.

Il problema fondamentale è quindi proprio tale prodotto, che comprende termini da tutti gli strati successivi della rete: vi è un problema di gradiente instabile. Bisogna fare in modo che tutti i termini in qualche maniera si bilancino a vicenda, cosa che non è mai di facile realizzazione.

2.4 Un'alternativa all'architettura *fully-connected*

Finora si è considerata l'architettura *feedforward* più semplice, ovvero quella che prende il nome di completamente connessa o *fully-connected*. In questo tipo di architettura, un neurone di uno strato riceve un input pesato da ogni neurone dello strato precedente ed invia il proprio output ad ogni neurone dello strato successivo. Tuttavia, esiste un'architettura che si adatta particolarmente bene al riconoscimento ed alla classificazione di immagini, in quanto cerca di trarre vantaggio dalla disposizione spaziale di un input (nel caso della classificazione di immagini, dalla disposizione dei pixel). Le reti neurali che ne fanno uso vengono chiamate convoluzionali.

Queste reti si basano su tre idee fondamentali:

- Consideriamo un'immagine di risoluzione *nxn*. Normalmente, si collegherebbero ognuno dei pixel di input ai neuroni del primo *hidden layer*. Nelle reti convoluzionali si scelgono piccole porzioni dell'immagine, ognuna delle quali avrà un neurone nascosto del primo strato di riferimento. Queste porzioni, che in genere sono quadrate, vengono chiamate campi recettivi locali. Ogni connessione del campo impara quindi un peso, mentre ogni neurone impara un *bias* complessivo del campo. Quindi si procede a spostare la finestra del campo recettivo a destra (o in basso, nel caso si sia terminata una riga) di una numero di pixel chiamato passo, che di solito prende il valore di 1, anche se può essere variato a seconda delle necessità.

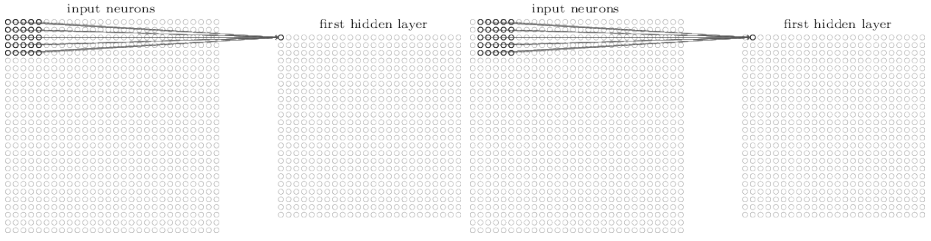

Figure 2.7: Il funzionamento del campo recettivo locale e di come viene spostato: tale procedimento viene ripetuto finché non si scorre l'intera immagine.

- Inoltre, ogni neurone del primo *hidden layer* condivide lo stesso peso e lo stesso *bias*. Questo significa che tutti i neuroni stanno ricercando nell'immagine esattamente la stessa caratteristica (ad esempio, l'angolo di un quadrato), ovvero un certo tipo di schema in ingresso causerà l'attivazione dei neuroni. Questo gruppo di neuroni viene chiamato mappa. Poiché, però, per il riconoscimento spesso non è sufficiente cercare una sola caratteristica, si affiancano diverse mappe, ognuna con il proprio peso, il proprio *bias* e la propria caratteristica da ricercare. la condivisione dei parametri

porta con sè anche il vantaggio di ridurre notevolmente il numero di parametri da calcolare durante l'allenamento, velocizzando l'apprendimento e rendendo più facile la costruzione di reti complesse.

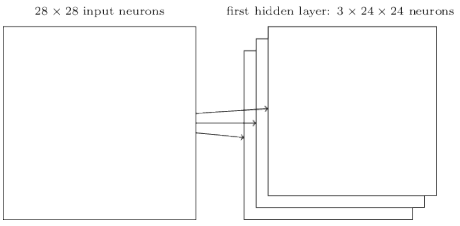

Figure 2.8: L'archiettura della rete finora.

- Immediatamente dopo i *layer* convoluzionali si trovano gli strati di *pooling*. Questi strati prendono le mappe precedentemente trovate e le condensano ulteriormente: infatti un neurone di *pooling* copre una regione della mappa, in modo simile a quello che avviene nei *layer* convoluzionali con l'immagine di input. Il tipo di *pooling* può variare: la più usata è il *max-pooling*, ovvero il neurone restituisce semplicemente l'attivazione massima della regione di mappa che analizza; questo tipo di *pooling* è un modo della rete per chiedere se una certa caratteristica è stata rilevata dalla mappa: l'informazione quindi che restituisce il *layer* di *pooling* è la locazione approssimativa di tale caratteristica, che poi verrà confrontata con la posizione di altre caratteristiche, in modo che la rete possa farsi un'idea di quello che sta analizzando. Un'altra tecnica piuttosto utilizzata è il *L2-pooling*, che fa la radice quadrata della somma dei quadrati delle attivazioni dei neuroni nella regione di mappa che si sta analizzando.

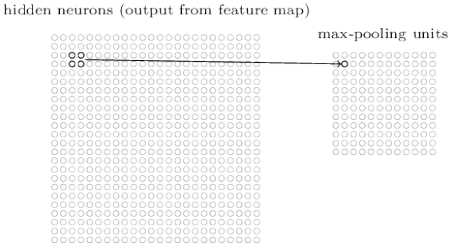

Figure 2.9: L'idea del *pooling layer* è proprio quella di condensare le informazioni che arrivano dalle mappe degli strati convoluzionali.

L'ultimo strato è completamente connesso e vuole quindi calcolare l'output considerando tutti gli strati di *pooling*. Si noti che per allenare la rete non si devono trovare altri algoritmi, ma si usa sempre l'algrotimo del gradiente decrescente e della *backpropagation*, facendo solo piccole modifiche proprio su quest'ultima per adattarla all'architettura convoluzionale e non più *fully-connected*.

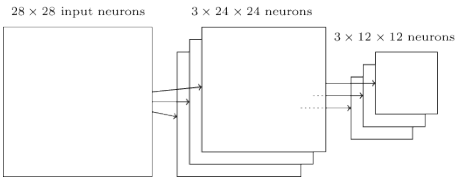

Figure 2.10: L'archiettura della rete finora.

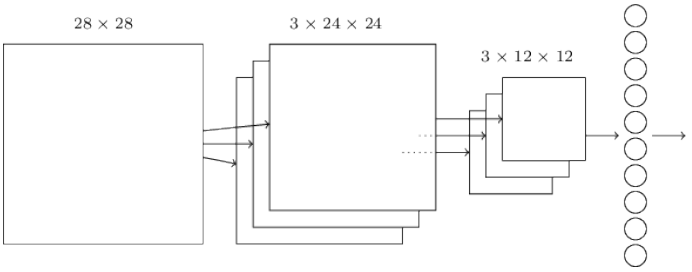

Figure 2.11: L'archiettura finale della rete.

Chapter 3

Reinforcement Learning

L'apprendimento supervisionato e l'apprendimento per rinforzo sono molto diversi l'uno dall'altro, fin dalla natura degli input che ricevono: mentre le reti che si allenano in modo supervisionato ricevono input statici e catalogati, una rete che segua il modello per rinforzo deve muoversi in un ambiente dinamico e in continuo mutamento. Da questo ambiente, essa deve trarre le informazioni necessarie a perfezionare il proprio comportamento. Infatti, ad ogni azione è associata una ricompensa (detta "rinforzo"), che deve essere mantenuta il più alto possibile dalla rete.

Nel *reinforcement learning*, infatti, non esiste il concetto di accuratezza, in quanto non vi è nessun supervisore che possa dire se la mossa è giusta o sbagliata: qui tutto si basa su quanta ricompensa riceve la rete ad ogni azione. A differenza dell'apprendimento supervisionato, invece, dove l'accuratezza è spesso presa a riferimento non solo per valutare le prestazioni della rete, ma anche per determinare diversi dei suoi parametri per allenarla al meglio. In pratica, il *supervised learning* si basa sull'apprendimento tramite esempi di cui un supervisore mostra il significato, mentre il *reinforcement leraning* sull'interazione tra un agente (comandato da una rete neurale) e l'ambiente in cui si trova ad agire.

Ovviamente, i due paradigmi di apprendimento avranno campi d'applicazione che siano consoni alle loro potenzialità: ad esempio, mentre una rete supervisionata sarà un ottimo punto di partenza per la classificazioni di immagini, una rete che apprende per rinforzo sarà una soluzione efficace nello sviluppo di un'intelligenza artificiale che giochi a scacchi.

3.1 Principi di funzionamento

I due elementi fondamentali nel *reinforcement learning* sono l'agente, che prende decisioni ed effettua azioni, e l'ambiente, in cui l'agente può muoversi e che offre diverse azioni da intraprendere. L'agente rappresenta una macchina a stati, che ha a disposizione un set di azioni in ogni stato che può compiere e che lo portano in stati diversi.

La prima caratteristica fondante di questo paradigma è il compromesso tra lo sfruttamento delle conoscenze in possesso dell'agente e l'esplorazione di nuove azioni. Infatti, l'agente, per ottenere una ricompensa notevole, deve ripetere le azioni che in passato gli hanno consentito un guadagno considerevole, ma deve anche esplorare nuovi comportamenti per individuare tali azioni: quindi, l'agente deve sfruttare ciò che già conosce per aumentare il proprio guadagno, ma anche esplorare per migliorare le decisioni che compierà in futuro.

Un'altra importante caratteristica è la considerazione del problema nella sua interezza: non si scompone il problema in numerosi sotto-problemi per poi risolverli uno indipendentemente dall'altro. Gli agenti hanno, invece, sempre uno scopo ben preciso e si muovono

nell'ambiente non passivamente, ma in maniera attiva, compiendo azioni per influenzarlo. Inoltre, dovranno agire anche se l'ambiente in cui si trovano risulta incerto e non predicibile: dovranno basarsi sulle informazioni che estraggono da esso sia per pianificare le proprie azioni che per compiere azioni che decidono *real-time*, istantaneamente.

A fianco di agente ed ambiente, si trovano quattro elementi altrettanto importanti nell'apprendimento per rinforzo: una linea di condotta, una funzione di ricompensa, una funzione di valore e, eventualmente, un modello dell'ambiente.

La linea di condotta definisce come l'agente si comporta al passare del tempo. Si tratta, metaforicamente, di una funzione che associa ad ogni stato dell'ambiente l'azione da intraprendere, una specie di associazione stimolo-risposta. La linea di condotta può essere semplicemente una tabella da consultare per stabilire l'azione da intraprendere o una fonte di ricerca su cui basare la decisione.

La funzione di ricompensa definisce l'obiettivo dell'agente. Associa ad ogni stato dell'ambiente (o ad ogni coppia stato-azione) un numero (il "rinforzo") che indica la desiderabilità di quello stato. Poiché l'obiettivo dell'agente è massimizzare la ricompensa totale a lungo termine, la funzione di ricompensa definisce quali azioni sono buone e quali cattive per l'agente. Nel campo biologico, può essere paragonato a ciò che dice quali azioni porteranno piacere e quali dolore. Definendo il problema e le sue regole, la funzione di ricompensa non può essere modificabile dall'agente, ma deve servirgli come base per modificare la sua linea di condotta.

Mentre, quindi, la funzione di ricompensa indica ciò che è buono nell'immediato, la funzione di valore definisce ciò che è buono a lungo termine (basandosi sulle conoscenze passate). Il valore può essere visto come la ricompensa totale che l'agente può aspettarsi di ricevere a partire da uno stato. La ricompensa dice quale stato successivo scegliere, il valore vuole cercare di pianificare una serie di stati e di indicare, basandosi su di essa, quale stato successivo scegliere, sempre considerando la ricompensa che deriva da ciascuno stato preso in considerazione. Anche se la ricompensa assume un ruolo primario rispetto al valore (la prima è certa, mentre il secondo è solo una previsione), è il valore ad essere coinvolto nel processo decisionale dell'agente: è preferibile seguire un'azione che sul momento dà poca ricompensa, ma che poi prevede maggiori guadagni finali, che non una che dà molta ricompensa subito e poi poca a lungo termine.

L'ultimo elemento, il modello dell'ambiente, non è sempre necessario: infatti serve all'agente per simulare l'ambiente e prevedere eventi che potrebbero accadere, ma di cui l'agente non ha mai avuto esperienza. L'agente può comunque imparare anche senza un modello, anche se, ovviamente, aggiungere un modello dell'ambiente contribuisce senza dubbio ad un apprendimento più veloce.

3.2 *Q-Learning*

Il *Q-Learning* è un modello di apprendimento per rinforzo che non fa uso di un modello dell'ambiente. Si basa sullo scegliere la linea di condotta migliore, utilizzando lo studio della funzione di valore composta da un numero finito di azioni da analizzare. In particolare, la forza del *Q-Learning* è il poter calcolare la ricompensa futura di un'azione senza la necessità di un modello dell'ambiente e senza dover prima aver sperimentato tale azione. Inoltre, non richiede adattamenti particolari per un problema specifico, ma può di qualsiasi problema che prevede un sistema di ricompense e di transizioni tra stati.

3.2.1 Algoritmo del *Q-Learning*

Il *Q-Learning* si basa esclusivamente sul valore per decidere l'azione migliore: ne consegue che, in ogni stato, sarà migliore l'azione che prevede una ricompensa a lungo termine maggiore. In particolare, tale ricompensa è la somma pesata di tutte le ricompense attese di ogni azione futura, dove il peso è un numero compreso tra 0 e 1. Tale peso, indicato con γ, viene anche chiamato fattore di sconto e serve per stabilire l'importanza della ricompensa passata rispetto a quella futura.

L'algoritmo prevede anche una funzione che calcola la quantità della combinazione tra stato ed azione, ovvero la ricompensa futura attesa:

$$\mathbf{Q} : S \times A \to \mathbb{R}$$

dove S ed A indicano, rispettivamente, l'insieme degli stati e delle azioni per ogni stato. Inizialmente, tale funzione ritorna un valore prestabilito. Successivamente, ogni volta che l'agente esegue un'azione ed osserva la nuova ricompensa e il nuovo stato (che dipendono dall'azione intrapresa e dallo stato di partenza), $\mathbf{Q}$ viene aggiornata. La base dell'algoritmo è, quindi, un semplice aggiornamento di $\mathbf{Q}$ ad ogni iterazione.

Basandosi quindi su $\mathbf{Q}$, che viene costantemente aggiornata con le nuove informazioni, l'agente può stimare le ricompense che riceve di volta in volta e decidere quindi l'azione da intraprendere in base alle sue stime.

Chapter 4

Sito Web di esempi

A completamento dello studio sui fondamenti del *deep learning*, sono stati realizzati tre esempi che mostrano il funzionamento dell'apprendimento di una rete neurale e di come essa lavora una volta che ha imparato.

A fianco di questi esempi, sul sito Web realizzato si trova anche una parte di teoria, che spiega in maniera semplificata i concetti fondamentali per comprendere a fondo il funzionamento delle demo, in modo, anche, di introdurre il lettore all'argomento del *deep learning*. Inoltre, si trovano anche due demo realizzate da un altro studente.

Il primo esempio è stato realizzato tramite una libreria di TensorFlow, mentre per le altre due demo è stata usata la libreria javascript messa a disposizione da ConvNetJs.

4.1 Classificazione di dati 2D

Nella prima demo, la rete neurale ha come scopo quello di classificare in modo corretto una serie di dati: tali dati sono distribuiti in una regione del piano (in particolare, nella regione: $[-6, 6] \times [-6, 6]$) e assumono un valore o positivo o negativo. La rete deve quindi dividere questa regione di piano in una zona positiva, contenente solo i valori positivi (contraddistinti dal colore blu), e in una zona negativa, che racchiude solo i valori arancioni, ovvero quelli negativi.

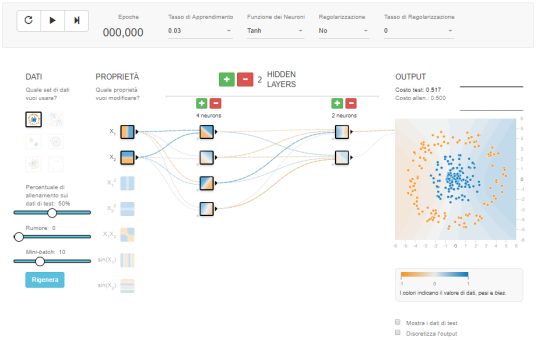

Figure 4.1: La schermata della demo: tutti i parametri sono impostati sui valori di default.

I primi parametri modificabili sono il tasso di apprendimento, di cui sono disponibili

alcuni valori tra cui scegliere, la funzione di attivazione dei neuroni (tra cui lineare rettificata, tangente iperbolica, sigma e lineare semplice), se usare o meno la regolarizzazione e, in caso affermativo, quale tipo utilizzare e il suo tasso. Inoltre, è disponibile anche un altro iper-parametro di cui scegliere il valore, la grandezza del *mini-batch*. L'utente può quindi modificare il valore di qualsiasi iper-parametro.

Si può modificare anche la distribuzione dei dati, scegliendo tra sei possibilità: circolare, XOR, gaussiano, spiraleggiante, casuale e casuale semplice. Inoltre è possibile scegliere il rapporto tra dati di allenamento rispetto a dati di test e il rumore durante la generazione dei dati.

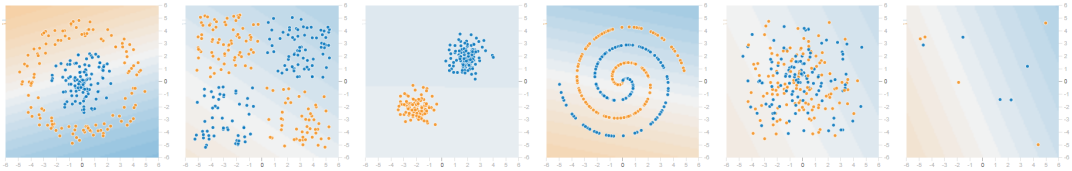

Figure 4.2: Tutte le possibili disposizioni dei dati.

La rete viene mostrata nella sua interezza. Lo strato di input presenta le funzioni che possono essere scelte come ingresso alla rete: determina quali forme possono essere usate dalla rete. I neuroni degli *hidden layer* mostrano il loro output. L'utente può anche scegliere il numero di strati nascosti e, per ognuno di essi, il numero di neuroni. L'output viene mostrato sull'estrema destra. Inoltre, vengono anche visualizzati tutti i pesi e tutti i *bias*: non solo, è anche possibile modificarli.

Infine, possono essere mostrati i dati di test e si può scegliere di visualizzare l'output discretizzato.

Quando la demo viene fatta partire, si visualizzano anche i valori e i grafici delle funzioni costo sia per i dati di allenamento, che per i dati di test. Se la rete sta riuscendo nel su intento, tali valori dovrebbero diminuire e l'output dovrebbe mostrare una zona arancione sotto i valori negativi e una zona blu sotto quelli positivi.

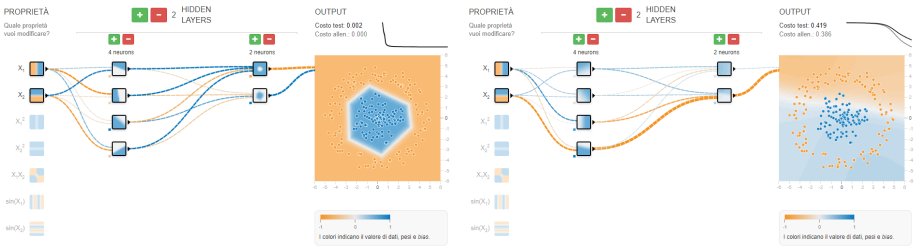

Figure 4.3: I risultati di alcune prove della demo dopo 300 epoche: a sinistra si sono usati neuroni lineari rettificati, mentre a destra neuroni sigmoidi. Si può vedere come reti uguali con attivazioni diverse abbiano prestazioni notevolmente diverse.

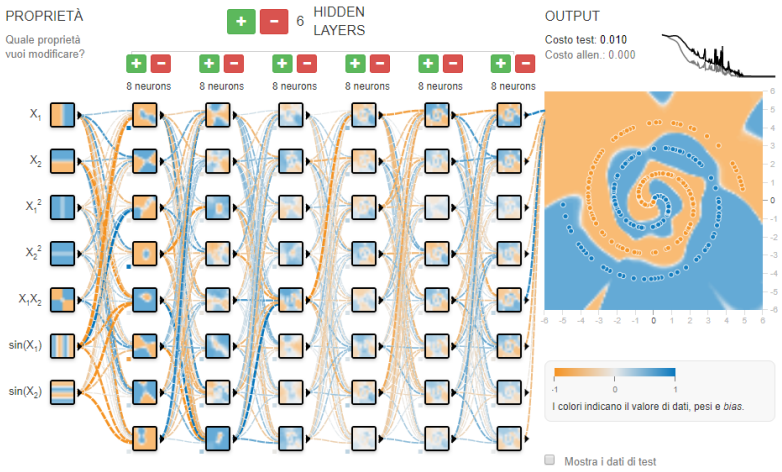

Figure 4.4: Esempio dell'apprendimento della rete neurale più complessa che si può realizzare nella demo dopo 300 epoche. In questo caso, si è usata la tangente iperbolica come attivazione. L'allenamento di una rete complessa come questa ha richiesto, ovviamente, molto più tempo di una rete come quella in figura 4.3.

4.2 *Reinforcement Learning*

Sono due le demo che trattano di apprendimento per rinforzo. La prima mostra sia il processo di apprendimento che quello di funzionamento effettivo della rete. La seconda, invece, è più un gioco che non una demo esemplificativa: l'utente dovrà sfidare il computer al gioco *Breakout*, completando per primo i sei livelli a disposizione.

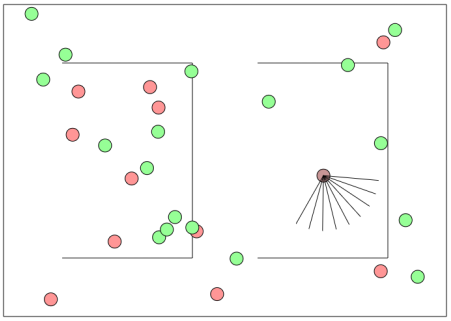

Figure 4.5: Il campo di gioco della prima demo: è questo l'ambiente dove si deve muovere l'agente, che è il pallino rosa.

Nella prima demo, l'agente deve muoversi nel campo di gioco e raccogliere le "mele", ovvero le palline rosse, ed evitare il "veleno", le palline verdi. L'agente ha a disposizione nove sensori, che rappresentano il suo campo visivo, ognuno dei quali calcola tre diversi valori (fino alla distanza massima di visibilità): la distanza da un muro, la distanza da una mela e la distanza da un veleno. Le azioni che può intraprendere sono cinque movimenti,

differenziate dall'angolo di rotazione.

Ciò che l'agente deve imparare è che le mele sono associate ad una ricompensa positiva e che quindi sono da raccogliere, mentre il veleno comporta una ricompensa negativa ed è perciò da evitare. L'algoritmo utilizzato è quello del *Q-Learning*.

I parametri modificabili dall'utente sono: la forma del campo di gioco, il numero di *hidden layer*, il numero di neuroni negli strati nascosti, tasso di apprendimento, la grandezza del *mini-batch*, tasso di regolarizzazione e il tasso di esplorazione ϵ, che rappresenta la probabilità che l'agente esegua una mossa nuova, ovvero che non ha mai eseguito prima.

I valori mostrati comprendono il grafico della rete, quello della ricompensa, il mondo di gioco e alcuni valori, come l'esperienza che l'agente ha accumulato (il massimo è 30.000), il tasso di esplorazione, il numero di epoche, il valore della funzione **Q** e il valore della ricompensa attuale. Il tasso ϵ viene mostrato perché durante l'allenamento assume un valore elevato, non costante e diverso da quello indicato: infatti, l'agente, in questa fase, deve imparare quali mosse sono quelle migliori, perciò dovrà compiere spesso azioni nuove. Sarà poi durante il gioco vero e proprio che ϵ assume il valore indicato.

Infine, si può scegliere la velocità di esecuzione della demo, se la rete debba allenarsi o giocare e se caricare una rete già allenata (in modo da poter vedere in azione una rete che sappia giocare senza dover allenarne una da zero).

Nella scelta dei parametri, sono stati applicati alcuni controlli sui valori, in modo che l'utente non possa usare valori esagerati, ad esempio, costruendo reti eccessivamente complesse.

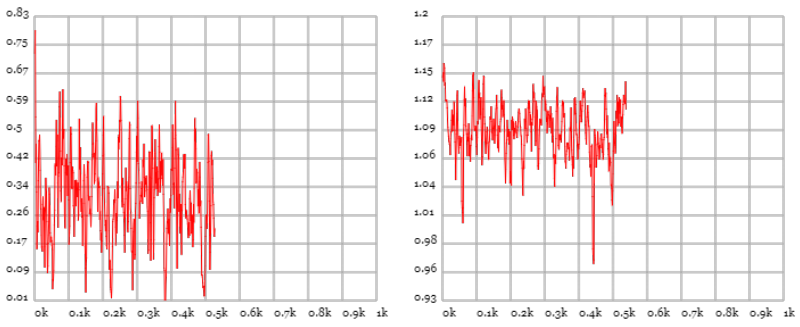

Figure 4.6: Due grafici di ricompensa: a sinistra, quello di una rete che gioca senza allenamento, a destra, quello di un agente già allenato. La differenza è notevole.

Il secondo esempio mostra invece le potenzialità di una rete neurale già allenata. Anche in questo caso, è stato usato javascript per realizzare il gioco. Si tratta del gioco *Breakout*, progettato da Atari negli anni Settanta.

La rete che comanda il giocatore del computer è stata allenata sempre tramite l'apprendimento per rinforzo, usando in particolare il *Q-Learning*, in modo analogo a quello che accade nel caso precedente. Qui, però, la rete è stata già allenata in precedenza con un totale di circa 15 milioni di mosse. Infatti, in questa demo non viene previsto la parte di allenamento della rete.

I due giocatori hanno gli stessi comandi a disposizione (muovere a destra e sinistra la barra) e ricevono le stesse informazioni, cioè ciò che viene mostrato sullo schermo. In questo caso, l'obiettivo del giocatore è terminare i cinque livelli che sono stati proposti

Figure 4.7: La demo come si presenta all'inizio.

prima che lo faccia il giocatore comandato dal computer.

Chapter 5

Reti neurali artificiali e reti neurali biologiche

Le reti neurali artificiali che si sono analizzate nei capitoli precedenti sono state costruite seguendo la struttura delle reti neurali che compongono il sistema nervoso degli esseri umani e, più n generale, degli esseri viventi. Le peculiarità sono molteplici, anche se le reti artificiali rappresentano una semplificazione di quelle biologiche, che risultano essere molto più complesse e raffinate.

5.1 Il neurone biologico: analogie con la sua controparte artificiale

Un neurone artificiale rispecchia la struttura di un neurone biologico. In entrambi possono essere infatti osservate tre parti principali: l'input, il corpo centrale e l'output. Nel neurone biologico, infatti, sono presenti i dendriti, sottili estensioni di forma tubulare che si ramificano a partire dal corpo cellulare. Essi svolgono una funzione di ricezione dei segnali in arrivo da altre cellule nervose, che comunicano i loro messaggi in zone specializzate, le sinapsi, zona di contatto tra dendrite ed assone.

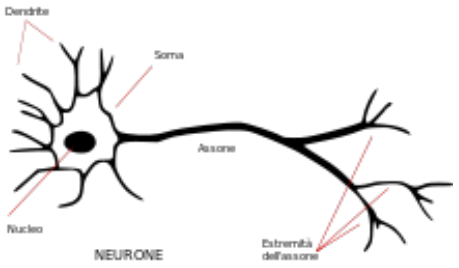

Figure 5.1: Lo schema della struttura di un neurone. Il neurone artificiale possiede una struttura simile.

Il centro del neurone biologico è il soma, ovvero il centro metabolico del neurone stesso. Qui si trovano il nucleo e l'apparato che produce, oltre che i costituenti della membrana cellulare, anche enzimi e altre sostanze indispensabili per svolgere le sue funzioni.

Dal lato opposto dei dendriti, si allunga l'assone, un sottile prolungamento della cellula. Esso ha il compito di di trasportare gli impulsi generati dal neurone sotto forma di potenziale d'azione verso le cellule bersaglio.

Il neurone artificiale riprende il funzionamento del neurone biologico: infatti, quest'ultimo riceve dagli altri neuroni una serie di segnali, di cui esegue la somma pesata. Se tale somma supera un valore soglia, il neurone si attiva ed invia un potenziale d'azione (un evento di breve durata in cui il potenziale elettrico di membrana di una cellula aumenta rapidamente e scende) a sua volta tramite il suo assone, altrimenti rimane in uno stato a riposo e non invia alcun segnale. In questo modo, è il *perceptron* ad essere più simile al neurone biologico, in quanto esso si attiva o meno se la somma pesata (a cui viene aggiunto il *bias*) supera lo zero. In generale, invece, un neurone che abbia come attivazione una funzione continua produrrà sempre un'uscita di qualche valore, anche se piccolo, benché anche per essi possa essere introdotto una soglia sopra la quale ritenere il neurone attivo.

5.2 Analogie e differenze tra i due tipi di reti neurali

L'unità fondamentale di entrambe le tipologie è quindi molto simile. Ma la struttura di una rete neurale biologica è talmente complessa da essere di difficile riproduzione per le tecnologie attuali. Le reti artificiali cercano di riprodurre il funzionamento delle reti biologiche semplificando perciò la loro struttura.

Un esempio lampante sono le reti *feedforward*: è impensabile, infatti, immaginare che il sistema nervoso possa assumere un'architettura a strati in cui vi una solo direzione delle informazioni. Le reti *feedforward* vogliono simulare una rete neurale artificiale, semplificando non solo la struttura, ma anche il loro funzionamento: in una rete biologica, infatti, le informazioni scorrono sempre in entrambe le direzioni. Anche se è vero che i neuroni possono specializzarsi in diverse funzioni: distinguere tra neuroni di input, di output e di passaggio (gli *hidden layer*) non è sbagliato, anche se per i neuroni biologici si usano nomi diversi (rispettivamente neuroni di senso, neuroni di moto e interneuroni).

Le reti neurali artificiali che, come struttura, si avvicinano maggiormente alle reti biologiche sono sicuramente e reti ricorrenti, in cui il concetto di *layer* viene a mancare e dove i neuroni non devono rispettare determinate regole per ricevere o inviare informazioni.

Per quanto riguarda i paradigmi di apprendimento, quelli analizzati non sono dissimili da quelli che avvengono durante il processo di apprendimento di una persona. Esiste anche in natura un processo supervisionato: imparare a partire da esempi già catalogati è molto diffuso (ad esempio, un bambino può imparare a distinguere cani e gatti perché gli viene detto tramite delle immagini). Anche l'apprendimento per rinforzo riprende un processo già esistente: ad esempio, un sistema piacere-dolore non si discosta molto dal concetto di ricompensa utilizzato in tale paradigma.

Per concludere, quindi, le reti neurali artificiali, insieme ai neuroni ed ai paradigmi di apprendimento, non sono altro che riproduzioni di sistemi già presenti a livello biologico: a cambiare non è tanto il principio di funzionamento, quanto la loro complessità.

Biblio-sitografia

- Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller, "Playing Atari with Deep Reinforcement Learning", 2013
- Richard S. Sutton, Andrew G. Barto, "Reinforcement Learning: An Introduction", The MIT Press, 2005
- In Enciclopedia Treccani: neurone
- In Wikipedia: rete neurale, potenziale d'azione, apprendimento per rinforzo, Q-Learning, test set

Per realizzare la demo sulla classificazione di dati 2D:
http://playground.tensorflow.org

Per realizzare le demo sul *reinforcement learning*:
http://cs.stanford.edu/people/karpathy/convnetjs/demo/rldemo.html
http://codeincomplete.com/posts/javascript-breakout/
http://users.eecs.northwestern.edu/ msn688/breakout_project/