Corn

Giorgio Giuffrè

Abstract

Corn (COstruttore di Reti Neurali) è una piccola piattaforma che permette di progettare e allenare semplici reti neurali artificiali feedforward (cioè acicliche), e di collaudarle poi su input numerici.

1 Introduzione

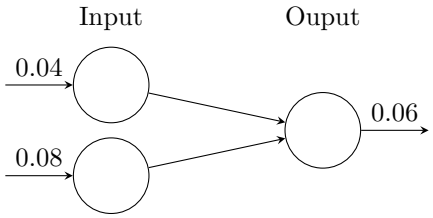
1.1 Cos'è una rete neurale?

Il miglior esempio di rete neurale è senza dubbio il cervello umano: una rete di cellule collegate tra loro — dette *neuroni* — che si scambiano informazione sotto forma di segnali elettrici. Alcuni neuroni si interfacciano con l'ambiente esterno: sono i neuroni sensoriali (di input) e quelli motori (di output); altri invece stanno "nascosti" all'interno, nei meandri della rete. Possiamo vedere i neuroni di input e quelli di ouput come l'*interfaccia* della rete con l'utente, mentre i neuroni nascosti costituiscono l'*implementazione* di un algoritmo — il comportamento umano, nel caso del nostro cervello. Nel cervello, ogni neurone manda un segnale a più neuroni e riceve segnali da neuroni diversi, determinando un'intricata catena parallela di segnali che termina con i neuroni motori, collegati ai muscoli.

Formalmente, una rete neurale è un grafo orientato in cui ogni nodo è un **neurone** e ogni arco è un **collegamento** che va da un neurone a un altro. Un neurone è una cellula che riceve uno o più segnali di input, li somma ed emette un solo segnale di output (quindi un neurone è una funzione $f : \mathbb{R}^n \to \mathbb{R}$, dove $n \geq 1$ è il numero di segnali in ingresso). In base al segnale totale di input x, ogni neurone emette quindi un certo segnale di output $f(x)$ che può poi ramificarsi, cioè può essere mandato a più di un neurone, a seconda di com'è disegnato il grafo. Le connessioni (gli archi) tra un neurone e l'altro sono pesate, cioè ogni input x_i viene moltiplicato per una costante reale w_i che può essere modificata dalla rete nel corso del tempo.

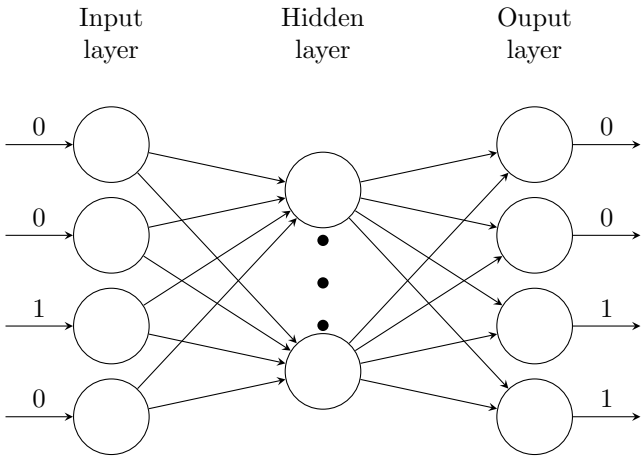
Tutti i neuroni della rete implementano la stessa semplice funzione f, detta **funzione di attivazione**. Chiaramente, l'output di due neuroni può essere diverso, per il fatto che gli archi della rete non hanno tutti lo stesso peso. La capacità della rete di modificare i pesi delle proprie connessioni fa sì che essa sia capace di associare ad ogni input un certo output desiderato. Ad esempio, una

rete con due neuroni di input e un neurone di output può modificare i propri pesi in modo da imparare a calcolare la media di due numeri che le vengono dati in input:



La rete apprende grazie ad una serie di **esempi** che le vengono presentati: $(0.04, 0.08 \rightarrow 0.06)$, $(0.05, 0.01 \rightarrow 0.03)$, $(0.02, 0.05 \rightarrow 0.035)$, $(0.09, 0.08 \rightarrow 0.085)$ e così via. Più esempi vengono forniti, più è preciso l'apprendimento. A questo proposito, è importante notare che la rete fornisce risposte *approssimate*, dovendo imparare da un insieme di esempi anziché da delle regole esplicite.

Oppure, una rete con 4 neuroni in ingresso e 4 in uscita (più un livello di neuroni "nascosti") potrebbe imparare a calcolare il successore di un numero in formato binario[1]:



1.2 Cosa può fare Corn?

Insomma, i compiti che una rete può imparare sono numerosissimi e Corn offre un'interfaccia semplice per specificare sia la configurazione della rete sia i compiti da farle imparare. La progettazione di una rete neurale artificiale si articola in tre fasi: definizione dell'architettura della rete; definizione degli esempi da presentare alla rete; allenamento della rete. Una volta allenata la rete, la si può interrogare su degli input numerici.

[1] I pallini neri nel disegno indicano la presenza di eventuali altri neuroni nascosti, oltre ai due disegnati

2 Guida per l'utente

CORN consiste di una sola finestra principale, dalla quale l'utente può lavorare su una rete alla volta. La finestra principale presenta, a sinistra, un elenco di cartelle contenenti le reti finora create (oltre a quelle disponibili di default). Ci sono tre cartelle, corrispondenti ai tre tipi di rete che il programma gestisce: le reti con funzione di attivazione sigmoide, quelle ad arcotangente e quelle a tangente iperbolica. Basta cliccare su una cartella e poi su una rete dell'elenco per potersi interfacciare con essa.

Come detto prima, la progettazione di una rete si articola in tre fasi:

- definizione dell'architettura della rete;
- definizione degli esempi su cui la rete si allenerà;
- allenamento vero e proprio.

La prima cosa da fare, quindi, è andare sul menù "Rete" e cliccare "Nuova Rete". La finestra principale contiene ora un pannello dal quale possiamo costruire la rete. Innanzitutto bisogna decidere il **nome** con cui battezzare la rete (senza l'estensione *.net*, aggiunta automaticamente per memorizzarla in un file); è utile darle il nome del compito che deve imparare: ad esempio, una rete che debba apprendere a calcolare la somma tra quattro numeri potrebbe chiamarsi "sum_4", per poterla ritrovare poi con più facilità. Si deve poi selezionare il **tipo di funzione di attivazione** dei neuroni (la stessa funzione per tutti i neuroni: ciò che cambia sono solo i pesi delle connessioni) e infine il **numero di livelli** della rete (includendo nel conto il livello di input e quello di output). Più il compito è elaborato, più livelli ci vogliono; generalmente, uno o due livelli nascosti vanno più che bene ma, se l'utente vuole sperimentare, può selezionare fino a 8 livelli. Cliccando "Prosegui...", l'utente potrà poi specificare il **numero di neuroni** per ogni livello; i livelli sono ordinati dall'input all'output e i più importanti sono il primo e l'ultimo, che dovranno contenere rispettivamente il numero di neuroni in ingresso e il numero di neuroni in uscita. Ora, basta cliccare "Crea" e la nuova rete comparirà in una cartella dell'elenco delle reti, a sinistra nella finestra principale.

Per scrivere gli esempi da presentare alla rete, dal menù "Dati" si seleziona "Nuovo foglio di esempi". Quello che viene presentato all'utente è un editor di testo diviso in due parti: nella prima va inserito l'input del singolo esempio; nella seconda l'output desiderato. Input e output devono essere sequenze di uno o più numeri floating point e bisogna tenere presente che i valori ideali per questi numeri sono tra 0 e 1 (per le reti a sigmoide, che hanno codominio $[0, 1]$) oppure tra -1 e 1 (per le reti ad arcotangente o a tangente iperbolica, con codominio $[-1, 1]$); va quindi benissimo anche una codifica binaria (solo 0 e 1) o bipolare (solo -1 e 1), per apprendere funzioni booleane o quant'altro. Ogni esempio va aggiunto con "Aggiungi" e dopo aver aggiunto l'ultimo basta

cliccare “Crea” e rispondere alla finestra di dialogo indicando il nome del foglio di esempi (aggiungendo anche l'estensione *.data*). Ancora, è opportuno dare il nome del compito da imparare: un foglio di esempi per la media tra due numeri si potrebbe intitolare “avg_2” o “media”. I file di allenamento sono reperibili nella cartella *logica/data/* e possono essere d'aiuto per imparare a creare nuove funzioni. Da notare che per semplicità nessuno di essi usa la codifica bipolare (-1 e 1) o una codifica continua in $[-1, 1]$; tuttavia, è possibile (e istruttivo) creare nuovi file con questa codifica: ad esempio un file *and_2.data* con gli esempi $(-1, -1 \mapsto -1)$, $(-1, 1 \mapsto -1)$, $(1, -1 \mapsto -1)$ e $(1, 1 \mapsto 1)$.

Dobbiamo ora allenare la nostra rete — i cui pesi sono ora casuali — con il foglio di esempi appena creato. Per fare ciò navighiamo nell'elenco di reti a sinistra fino a trovare la rete che cerchiamo. Si apre ora l'interfaccia utente-rete, dalla quale possiamo scegliere se allenare la rete o interrogarla su degli input: basta scegliere tra le due tab in alto. Può essere interessante collaudare la rete prima ancora di averla allenata: darà degli output insensati per via del fatto che i pesi delle sue connessioni sono stati inizializzati a caso. Per allenarla, invece, inseriamo il nome del **file di dati** da cui la rete imparerà, il **massimo numero di epoche**[2] che la rete avrà a disposizione per allenarsi e il massimo **errore** (medio, calcolato sugli esempi) tollerato. Schiacciando il pulsante “Allena”, la rete apprenderà il compito richiesto grazie all'algoritmo di *Backpropagation*[3]. L'algoritmo può impiegare pochi millisecondi come anche parecchi secondi: ciò dipende dalla difficoltà del compito che la rete deve imparare e, soprattutto, dall'architettura della rete: più neuroni ci sono, più l'algoritmo è lento.

Per testare la rete che abbiamo appena allenato, selezioniamo l'altra tab (“Collauda la Rete”) dell'interfaccia utente-rete. Qui l'utente può mettere in input dei numeri (interi o reali, ma la rete li interpreta come reali) e premere “Vai” per osservare il risultato della rete. Si potrà notare come una rete allenata fornisca, con un certo grado di approssimazione, proprio gli output che è stata allenata a calcolare. Da notare che un input troppo breve (eventualmente vuoto) è completato dalla rete con degli zeri; un input troppo lungo, invece, viene tagliato.

Possiamo anche collegare diverse reti tra loro creando una *inter-rete*. È sufficiente fare *ctrl-click* (*cmd-click* sul Mac) per aggiungere una rete dall'elenco a sinistra nell'inter-rete che stiamo creando. Dopo aver aggiunto le reti che ci servono, dobbiamo cliccare “Inter-rete” dal menù “Rete”. Ci viene presentata una tabella (la matrice di adiacenza dell'inter-rete): per collegare l'output della rete x a un input della rete y dovremo cliccare la casella x della riga y, che si colorerà di nero. Un successivo click annulla il collegamento tra le due reti. Il pulsante “Collauda...” porterà all'interfaccia per interrogare la rete appena creata, in cui

[2] Un'epoca è il periodo in cui la rete osserva tutti gli esempi di un foglio: dopo tre epoche, la rete avrà visto ogni esempio tre volte.

[3] `https://en.wikipedia.org/wiki/Backpropagation`

si dovrà inserire un input di lunghezza pari alla somma dei neuroni di input delle reti in ingresso. Ad esempio, aggiungiamo all'inter-rete *and.net*, *or.net* e *xor.net*; dalla tabella dell'inter-rete, coloriamo le prime due caselle della riga in basso (così l'output di *and* e *or* sarà l'input di *xor*) e premiamo e "Collauda..."; ricordando che abbiamo inserito nell'inter-rete prima *and* e poi *or*, scriviamo "0 1 1 1": (0 1) è l'input di *and*, mentre (1 1) è quello di *or*; l'input di *xor* sarà quindi $((0and1)(1or1)) = (01)$, per cui l'output totale sarà $(0xor1) = 1$... approssimato, dato che stiamo usando una rete neurale!

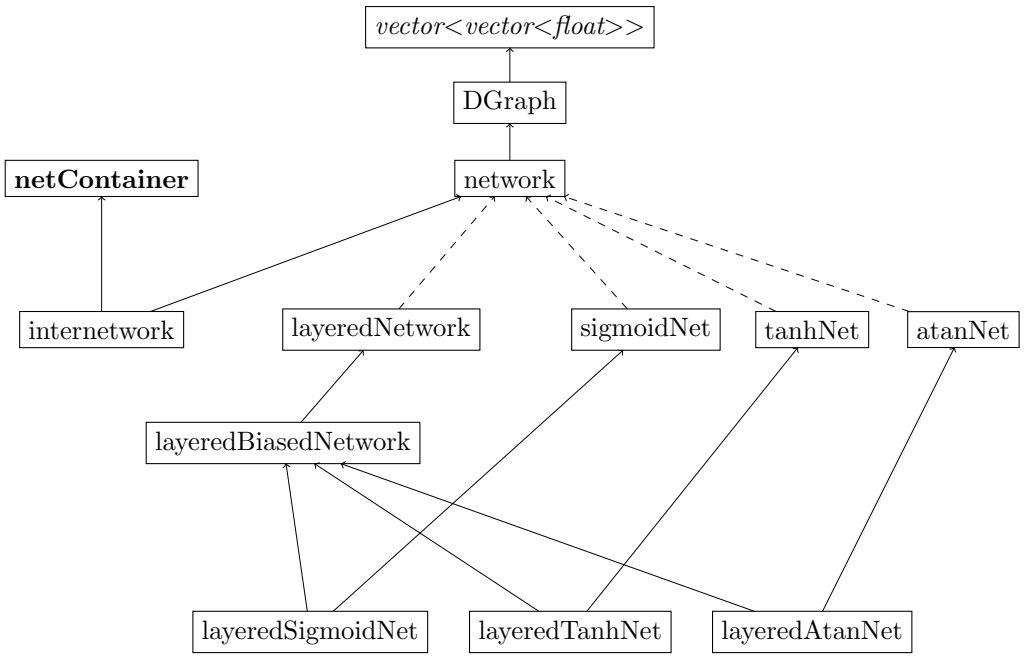
3 Implementazione

Il programma è stato progettato seguendo lo schema *Model-View* ed è diviso in due blocchi ben distinti: il modulo logico (nella cartella "logica") e quello grafico. Il modulo logico è indipendente da quello grafico. Mentre l'interfaccia è in italiano, l'implementazione è in inglese. Sono stati usati `gcc 4.6` (con supporto parziale per `C++11`), `Qt 5.3.2` e `GNU Make 3.81` su Ubuntu 12.04. Il file *progetto.pro* è stato generato manualmente.

3.1 Parte logica

Una rete neurale è una rete, che a sua volta è un grafo orientato. Un grafo orientato è rappresentabile per mezzo della sua matrice di adiacenza W, in cui l'elemento w_{ij} rappresenta la presenza o meno di un collegamento dal nodo j-esimo al nodo i-esimo. Per modellare una rete neurale, in cui i collegamenti tra i neuroni sono pesati, la matrice di adiacenza deve avere elementi reali (`float`): ogni segnale dal neurone j al neurone i verrà moltiplicato per w_{ij}.

Alla base della gerarchia che implementa una rete neurale (in figura) sta la classe `DGraph` (*Directed Graph*), sottotipo di `std::vector<std::vector<float> >`; è quindi una matrice di `float`. Non si sono scelti i `double` perché, anche biologicamente, una rete neurale non lavora a precisione elevata. `DGraph` implementa un grafo orientato ma *non* aciclico, dato che in futuro il programma potrebbe essere dotato di reti cicliche (un'architettura ben più potente ma anche più complicata); per ora, la classe `layeredNetwork` implementa un'architettura che garantisce assenza di cicli. texttttDGraph, per ereditarietà, è un contenitore. Sono stati introdotti due nuovi iteratori per il seguente motivo: di ogni nodo interessano solo i collegamenti con gli *altri* nodi; per questo, la classe `weights_iterator` itera sui collegamenti entranti (anche nulli) di un particolare nodo (specificato nel costruttore dell'iteratore) ma evita automaticamente i collegamenti del nodo con se stesso. Per completezza, questa classe è definita a partire da `nodes_iterator`, una semplice classe iteratore che rappresenta un indice da utilizzare con l'operatore di subscripting `[ ]` per iterare sui nodi.



Da `DGraph` eredita la classe `network`, che implementa alcune funzionalità specifiche delle reti come il passaggio d'informazione (con `activation_function`), l'inizializzazione dei pesi e il calcolo del flusso (`operator()`). Da `network` discendono varie classi:

- `internetwork`: una *rete di reti* — un contenitore che è esso stesso rete;
- `layeredNetwork`, che definisce l'**architettura** a strati tipica delle reti neurali che creeremo;
- `sigmoidNet`, `tanhNet` e `atanNet`, che definiscono le **funzioni di attivazione** dei neuroni della rete.

È necessaria una classe ulteriore: `layeredBiasedNetwork`, che aggiunge un dettaglio tecnico alla classe da cui eredita, cioè la presenza di neuroni "bias". Questi sono degli input costanti (non visibili all'utente) che servono alla rete per generalizzare meglio; ogni strato della rete (tranne quello di output) ne possiede uno.

Infine, partendo da queste classi possiamo facilmente definire quelle che effettivamente saranno le nostre reti neurali: `layeredSigmoidNet`, `layeredTanhNet` e `layeredAtanNet`. Dato che queste classi chiudono "a diamante" la gerarchia, l'ereditarietà della classe che definisce l'architettura e di quelle che definiscono le funzioni di attivazione è virtuale.

Descriviamo brevemente cos'è un `internetwork`. In pratica, ogni rete che abbia un solo nodo di output (oppure di cui ci interessi solo il primo nodo di output) può essere considerata come nodo di una rete. Infatti, in una rete normale, ogni nodo ha uno o più input, che vengono pesati e sommati prima di essere processati dalla funzione di attivazione; anche in un `internetwork`, ogni nodo ha uno o più input, che però non vengono sommati bensì considerati come input distinti del nodo, che è qui una rete. Le classi da cui eredita `internetwork` sono `netContainer`, una contenitore[4] di `network *` (per avvalersi del polimorfismo) e `network`, dato che vogliamo creare un contenitore di reti che sia esso stesso una rete. Di `network` viene reimplementata, tra altre, la funzione *store*, per permettere all'utente di inserire un input "totale" che vada a "distribuirsi" su tutti i nodi di input delle reti di input.

3.2 Interfaccia grafica

L'utente si interfaccia con una finestra unica di tipo `cornWindow`, chiamata nel *main*, che consiste di:

- una barra dei menù — membro puntatore a `QMenuBar`;
- un *dock* laterale — membro puntatore a `netsList`, che eredita da `QDockWidget`;
- un'area che cambia apparenza a seconda di ciò che l'utente vuole fare — membro puntatore a `QStackedWidget`, che si avvale degli altri membri di `cornWindow` (puntatori a `netInterface`, `netBuilderWidget`, `dataBuilderWidget` e `internetInterface`).

I membri a cui punta lo `QStackedWidget` ereditano tutti da `QWidget`, tranne `netInterface` che eredita da `QTabWidget`; infatti, quest'ultima classe rappresenta l'interfaccia dell'utente con la singola rete neurale (scelta dal *dock*) e dalle due tab si può scegliere se interrogare la rete (per mezzo di un `netRunnerWidget`) o allenarla (per mezzo di un `netTrainerWidget`).

[4] *Il* contenitore della consegna.