Università degli Studi di Padova

Dipartimento di Matematica
"Tullio Levi-Civita"

Corso di Laurea in Informatica

DBNsim: un software per analizzare *Deep Belief Networks*

Relatore: prof.ssa Silvia Crafa
Laureando: Giorgio Giuffrè

Anno Accademico 2016-2017

INDICE

1 INTRODUZIONE

Questa relazione presenta *DBNsim*, un'applicazione che ho sviluppato durante i due mesi di tirocinio che concludono il Corso di Laurea in Informatica a Padova. Ho realizzato questa applicazione per il *Computational Cognitive Neuroscience Lab* del Dip. di Psicologia Generale, che mi ha ospitato durante il tirocinio e mi ha arricchito di mille spunti su numerosi temi affascinanti.

Ho contattato il laboratorio qualche mese dopo aver seguito il corso di Intelligenza Artificiale dei proff. Marco Zorzi e Alberto Testolin. Volevo approfondire il campo dell'I.A. seguendo l'approccio "connessionista" che i due professori ci avevano fatto conoscere a lezione: secondo questo approccio le reti neurali non sono soltanto strumenti da "ottimizzare" per risolvere problemi specifici, bensì soprattutto dei modelli generali che dovrebbero essere *biologicamente plausibili* — degli strumenti per modellare il cervello e cercare di capirlo.

Il prof. Testolin, che è stato mio tutor durante questi due mesi, mi ha chiesto di sviluppare un software per allenare un particolare tipo di rete neurale (che introdurrò tra poco). Il software, a uso principalmente didattico, doveva permettere di analizzare visivamente queste reti e di far capire nel modo più chiaro possibile il loro funzionamento interno.

Allenare una rete neurale vuol dire avere a che fare con una struttura apparentemente caotica: solo un'interfaccia grafica accortamente pensata può chiarire il funzionamento della rete, nonché il suo significato dal punto di vista biologico. DBNsim è quindi uno strumento che permette di **capire** e **analizzare** reti neurali.

STRUTTURA DEL DOCUMENTO Nel resto del capitolo introdurrò le *Deep Belief Networks* (DBN; il tipo di rete neurale su cui si concentra il mio software) ed esporrò gli obiettivi personali che ho voluto raggiungere con questo tirocinio. Verranno poi i seguenti capitoli:

- il capitolo 2 presenta i risultati ottenuti da una prima analisi generale del problema.
- il capitolo 3 espone la soluzione progettuale che ho escogitato dopo una breve analisi ulteriore;
- il capitolo 4 descrive come ho sviluppato il software e presenta alcuni aspetti particolari dello sviluppo;

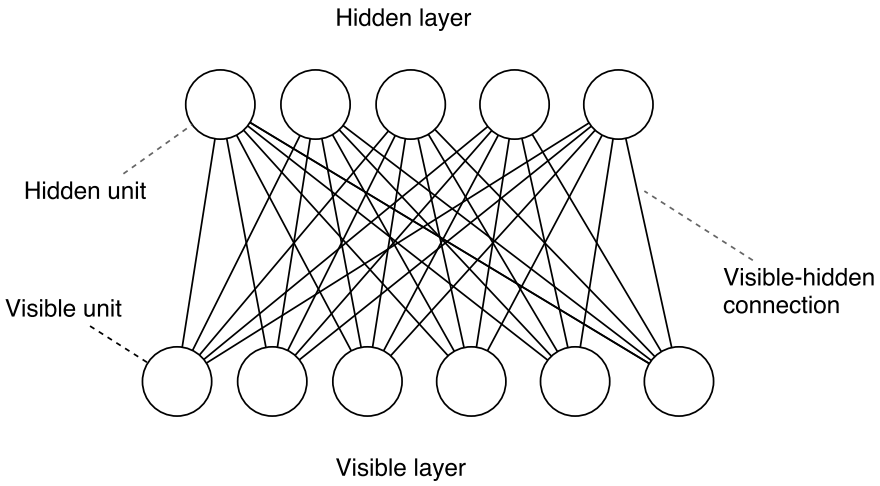

Figura 1: Esempio di *Restricted Boltzmann Machine*.

- con il capitolo 5 condivido gli aspetti più importanti che il mio software fa emergere e tiro le somme di ciò che, a mio avviso, un informatico può imparare dal campo delle reti neurali.

1.1 COS'È UNA DBN

Deep Belief Network è il nome che si dà a una particolare **architettura** di rete neurale, cioè a una particolare disposizione di neuroni all'interno di una rete. Per capire qual'è questa architettura, bisogna prima spiegare che cos'è una **RBM**, il mattone fondamentale di ogni DBN.

1.1.1 *Restricted Boltzmann Machine*

RBM sta per *Restricted Boltzmann Machine*, un tipo di rete neurale che corrisponde a un grafo bipartito pesato e non orientato. Cerchiamo ora di capire cos'è e a cosa serve.

STRUTTURA DI UNA RBM Essendo un grafo bipartito, una RBM contiene due gruppi di unità; questi due gruppi vengono chiamati "livelli". Come si può vedere in figura 1, uno di questi livelli è nascosto all'esterno (cioè chi interagisce con la rete non se sa nulla). L'altro, invece, è visibile all'esterno: le sue unità vengono attivate da input esterni, ad esempio il pixel di un'immagine.

Oltre ad essere un'interfaccia per i dati in ingresso, tuttavia, il livello visibile è anche usato per emettere degli output; quindi input e output sono dati dello stesso tipo. Facciamo un esempio: se una RBM osserva delle immagini da 30x30 pixel, possiamo considerare queste immagini come vettori da 900 unità, per semplicità; facendo osservare questo vettore alla RBM, ognuno dei 900 neuroni di input

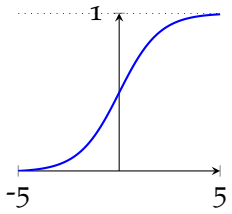

Figura 2: La funzione sigmoide.

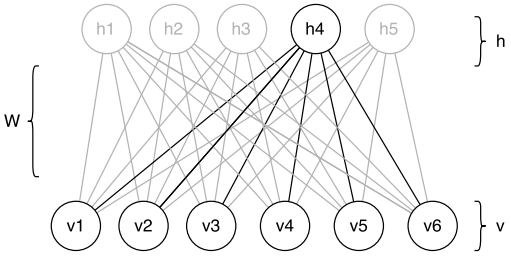

Figura 3: Calcolo dell'attivazione del neurone h_4, a partire dai neuroni del livello v.

della rete riceverà quindi un pixel e, dopo che i due livelli della RBM avranno interagito (nel modo che vedremo tra poco), i 900 neuroni visibili avranno assunto ognuno un nuovo valore; il livello visibile della RBM sarà quindi un nuovo vettore da 900 unità, cioè una nuova immagine da 30x30 pixel.

FUNZIONAMENTO DI UNA RBM Ci rimane da capire *qual è* il senso dell'output che una RBM sintetizza a partire da un input e *come* lo sintetizza. Oguno dei collegamenti tra due neuroni i e j di una RBM è **pesato**, cioè ha un coefficiente $W_{ij} \in \mathbb{R}$ che modifica i dati che "viaggiano" su quel collegamento. Questi collegamenti modellano le sinapsi del cervello, con i pesi negativi corrispondenti alle sinapsi inibitorie e quelli positivi alle sinapsi eccitatorie.

Quando il livello visibile osserva un input, ogni neurone h_i del livello nascosto assume valore 0 oppure 1, con probabilità

$$p(h_i \leftarrow 1) = \sigma(\sum_{j=1}^{m} W_{ij} * v_j) \tag{1}$$

dove:

- h è un vettore di n neuroni nascosti (*hidden*) — come si può dedurre dalla formula 1, i neuroni nascosti hanno valori **binari**.
- v è un vettore di m neuroni visibili (*visible*) — si attivano anch'essi con valori binari ma torna spesso utile osservare il valore della probabilità con cui si attivano, anziché il valore binario.
- W è la matrice dei pesi del grafo: W_{ij} è il peso del collegamento tra h_i e v_j; in pratica, ogni riga corrisponde a un neurone nascosto e ogni colonna a un neurone visibile.
- σ è la funzione sigmoide, definita come $\sigma(x) = \frac{1}{1+e^{-x}}$ (vedi figura 2).

Dopo aver calcolato l'attivazione dei neuroni nascosti, si può calcolare il nuovo valore dei neuroni visibili con una formula analoga, sempre basata su una somma pesata:

$$p(v_j \leftarrow 1) = \sigma(\sum_{i=1}^{n} W_{ij} * h_i) \tag{2}$$

Se i pesi della RBM sono casuali, l'output non ha alcun significato. Ma se i pesi vengono opportunamente impostati, una RBM è in grado di *inferire* un output completo a partire da un input incompleto o rumoroso; inoltre (sempre se i pesi vengono opportunamente impostati) le attivazioni del livello nascosto possono rivelare **caratteristiche non esplicite** nei dati — caratteristiche *astratte*.

APPRENDIMENTO IN UNA RBM Si chiama "apprendimento" il processo svolto da una RBM per modificare i propri pesi in modo tale da conoscere un certo dominio applicativo, ad esempio immagini di volti umani. L'apprendimento, basato su un insieme di esempi, si svolge in modo **non supervisionato**, cioè senza che ogni esempio sia associato ad un preciso valore da imparare come nell'apprendimento supervisionato; qui, piuttosto, la RBM cerca di capire qual è il "senso" che sta dietro ad ogni esempio, cos'hanno in comune i vari esempi — qual è un buon modo di sintetizzare i dati osservati. La sezione 1.1.3 spiega meglio la differenza tra apprendimento supervisionato e non.

L'algoritmo di apprendimento si chiama *Contrastive Divergence* (divergenza contrastiva); verrà introdotto tra poco e spiegato più in dettaglio in appendice B.

1.1.2 *Deep Belief Network*

Conoscendo struttura e funzionamento delle RBM, capire le DBN è semplice. Una DBN è una catena (una pila) di RBM, dove il livello nascosto della n-esima RBM coincide con il livello visibile della $n+1$-esima. La figura 4 illustra una DBN composta da tre RBM. La terminologia è la stessa delle RBM: questa rete ha un livello visibile (il livello visibile della prima RBM) e tre livelli nascosti. Inoltre, si dice "profondità" di una DBN il numero di livelli nascosti che essa ha.

MOTIVAZIONI Il senso di concatenare delle RBM è il seguente: se il livello nascosto di una singola RBM è in grado di rivelare caratteristiche astratte, una RBM costruita sopra un'altra RBM sarà in grado di rivelare caratteristiche ancora più astratte.

Tali caratteristiche vengono evidenziate dal **campo recettivo** di ogni neurone; il campo recettivo di un neurone è quell'input che, osservato dalla rete, produce la massima eccitazione nel neurone. In pratica,

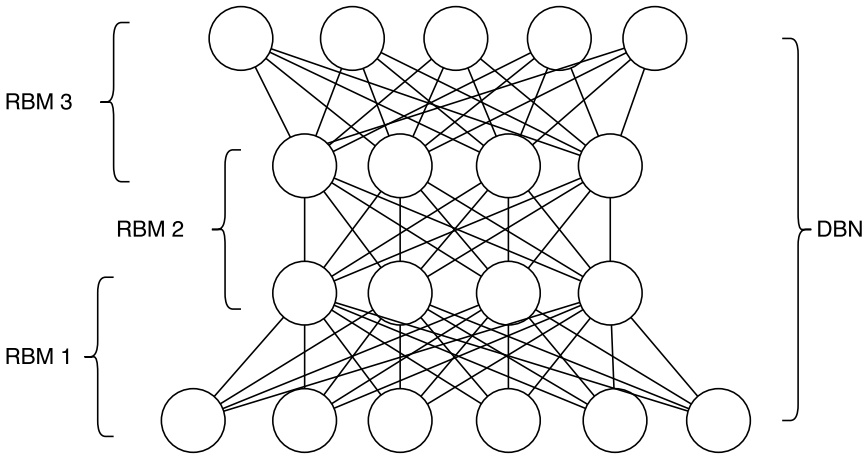

Figura 4: Esempio di *Deep Belief Network*.

man mano che si sale nella DBN, i campi recettivi dei neuroni nascosti rivelano caratteristiche di livello sempre più alto (vedi figura 5).

Un esempio biologico chiarirà il senso di tutto ciò. L'occhio umano, possiamo dire, riceve in input delle immagini; i dati raccolti da un occhio vengono passati a dei neuroni che rilevano caratteristiche di **basso livello** (se una forma è orizzontale o verticale, se è a destra o a sinistra...); questi neuroni passano tali caratteristiche ad altri neuroni, che rilevano caratteristiche più astratte (ad esempio semplici forme orizzontali, linee, punti, cerchi...), fino ad arrivare a neuroni che si attivano su input di alto livello quali ad esempio la faccia di un amico (come in fig. 6), un particolare tipo di oggetto o il simbolo di qualche alfabeto.

APPRENDIMENTO IN UNA DBN L'apprendimento, in una DBN, si svolge lasciando che ogni RBM apprenda a partire da ciò che la precedente ha imparato: la prima RBM (quella il cui livello visibile coincide con quello dell'intera DBN) apprende dai dati; la seconda apprende dalle attivazioni dei neuroni nascosti della prima; la terza apprende dalla seconda e così via.

Nello specifico, ogni RBM osserva un certo numero di esempi; per ogni esempio, il livello nascosto si attiverà con determinati valori binari: i valori binari del livello nascosto costituiranno un esempio da cui la successiva RBM imparerà.

Riassumo qui l'algoritmo di apprendimento *Contrastive Divergence*, per poi spiegarlo più in dettaglio in appendice B. Assumiamo che i livelli visibile e nascosto (v e h) di una RBM siano vettori "verticali", così come ogni esempio del *training set*. Assumiamo, inoltre, che la funzione sigmoide σ applicata su un vettore x restituisca un vettore y i cui elementi sono $y_i = \frac{1}{1+e^{-x_i}} \forall x_i \in x$. Data una **DBN** con matrice dei pesi W e dato un **insieme di esempi** $\{E_i\}_{i=1..k}$, per ogni RBM della DBN:

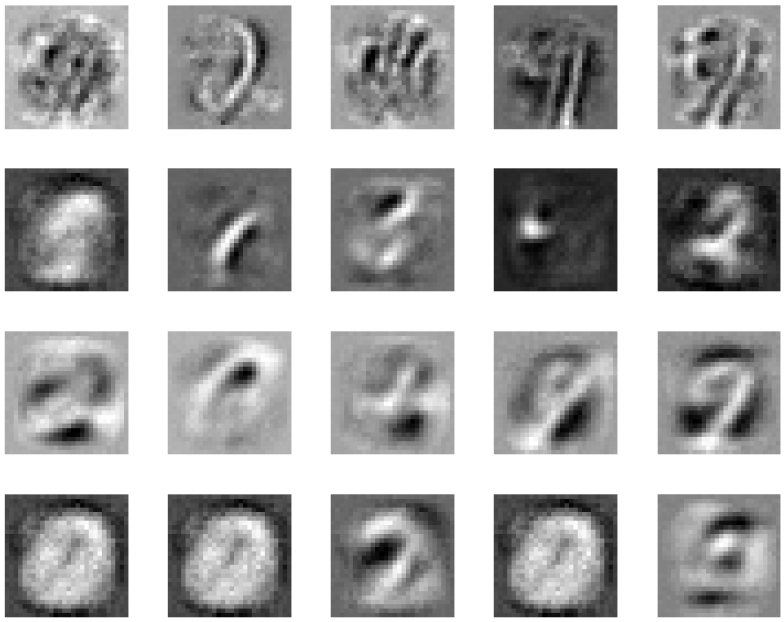

Figura 5: Campi recettivi di neuroni selezionati a caso dai livelli nascosti di una DBN allenata su immagini di numeri manoscritti; i campi recettivi della prima riga appartengono a cinque neuroni del primo livello nascosto, quelli della seconda riga al secondo livello e così via.

1: **while** non viene raggiunto il massimo numero di epoche **do**
2: **for all** esempio $e \in E$ **do**
3: $h \leftarrow \sigma(W \cdot e)$
4: $C_+ \leftarrow e \cdot h^T$
5: **if** siamo all'ultima epoca **then**
6: $e \leftarrow h$
7: **end if**
8: $v \leftarrow \sigma(W^T \cdot v)$
9: $C_- \leftarrow h \cdot v^T$
10: $W \leftarrow W + \eta \cdot (C_+ - C_-)$
11: **end for**
12: **end while**

Spiegato a parole:

- la riga 3 calcola, a partire dall'esempio e, le attivazioni h del livello nascosto;
- la riga 4 calcola la matrice delle correlazioni tra l'esempio e tali attivazioni;
- le righe da 5 a 7 (se siamo giunti all'ultima epoca) costruiscono il *training set* da cui si allenerà la prossima RBM, a partire dalle attivazioni del livello nascosto;

- la riga 8 calcola, a partire dalle attivazioni h del livello nascosto, le attivazioni v del livello visibile;
- la riga 9 calcola la matrice delle correlazioni tra le attivazioni del livello nascosto e quelle del livello visibile appena calcolate;
- infine, la riga 10 aggiorna la matrice dei pesi (in questo momento, davvero, la rete "apprende"); $\eta \in [0, 1] \cap \mathbb{R}$ è il coefficiente di apprendimento (*learning rate*), cioè la facilità con cui è possibile modificare i pesi della RBM.

1.1.3 Apprendimento non supervisionato

Prima di chiudere la parte teorica introduttiva, è d'obbligo spiegare un concetto generale dell'apprendimento automatico: la differenza tra apprendimento supervisionato e non supervisionato.

L'apprendimento automatico si preoccupa di fare in modo che un programma impari *da solo* un compito. Ci sono due modi con cui un programma può imparare:

- Con l'aiuto di un supervisore che ad ogni esempio associa una risposta corretta (apprendimento supervisionato) — il programma dovrà poi associare da solo, ad ogni esempio, questa risposta.
- Senza alcun supervisore (apprendimento non supervisionato). Questo implica che un esempio non ha una risposta corretta e, anzi, che non ha affatto una risposta; piuttosto, ogni esempio è visto come il "fotogramma" di un ambiente a cui il programma deve abituarsi, adattarsi.

Le DBN, come si è detto, imparano in modo **non** supervisionato: vedono ogni esempio di input come una sorta di immagine, che influenza i pesi delle loro connessioni. Da questi esempi una rete deve riuscire a ricavare dei buoni indicatori, cioè deve riuscire a capire quali sono dei buoni concetti che *generalizzano* ciò che essa sta osservando. Ad esempio, una DBN che osserva delle fotografie di volti umani dovrà riuscire a trovare degli indicatori abbastanza astratti e generali da non perdersi nella miriade di dettagli del volto umano: i campi recettivi di una tale DBN (vedi ad esempio la figura 6) evidenzieranno la sagoma del volto o la presenza di due occhi, ad esempio, non certo il numero di capelli o il colore degli occhi.

1.2 MOTIVAZIONI PER QUESTO TIROCINIO

Il motivo principale che mi ha spinto a pianificare questo tirocinio è stato un mio interesse non soltanto verso il campo delle reti neurali

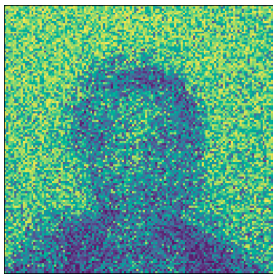

Figura 6: Esempio di campo recettivo in una DBN che ha osservato dei volti umani.

ma, più in generale, verso le architetture parallele e distribuite. Una rete di sensori indipendenti e interconnessi, infatti, non solo è un potente strumento ma può anche essere un vero e proprio **paradigma** per capire il mondo attuale.

Il mondo della *share economy*, dell'IoT, della Modernità Liquida, del *peer-to-peer* e del consumismo presenta dei temi ricorrenti che caratterizzano anche le reti neurali. Se il calcolo su GPU e le reti *peer-to-peer* decentralizzano il controllo, così fa anche una rete neurale, che preferisce tante semplici unità di controllo (i neuroni) a un'unica potente unità centrale. Se il consumismo preferisce la quantità alla qualità, gli algoritmi stocastici e "approssimativi" delle DBN sono abbastanza d'accordo: meglio tanti dati approssimativi piuttosto che pochi ma buoni. Infine, volendo spingerci fin nel campo della sociologia, un mondo che cambia di continuo richiede agli individui di adattarsi in fretta, così come l'aprendimento di una DBN non è altro che l'adattamento a un particolare *training set*.

Benché queste idee siano vaghe, sono utili almeno come domande. Percui, dopo aver studiato per tre anni (con grande interesse) gli aspetti più classici dell'informatica, ho voluto conoscere alcuni aspetti più sperimentali.

2 ANALISI INIZIALE

Questo capitolo analizza il problema così come l'ho affrontato inizialmente. Questa analisi, partita dal progetto di un'applicazione desktop, arriva poco a poco a una soluzione diversa: creare una *web app*; mentre l'idea della *web app* verrà sviluppata nel prossimo capitolo, il presente si concentra su quei **risultati dell'analisi iniziale** che sono rimasti validi nel passaggio da applicazione desktop ad applicazione web.

2.1 REQUISITI PRINCIPALI

Una prima analisi degli obiettivi principali da soddisfare ha portato al progetto di un'applicazione desktop. Parlerò di questa analisi in modo discorsivo; per un elenco formale e completo dei requisiti, si veda l'appendice A.

Il prodotto richiesto dal laboratorio era un'applicazione per **creare**, **allenare** e **analizzare** *Deep Belief Networks* (DBN); in particolare, queste reti dovevano essere allenate con l'algoritmo *Contrastive Divergence* (vedi appendice B) e doveva essere possibile analizzare una rete neurale sia durante il suo allenamento sia una volta allenata.

I requisiti appena elencati, dunque, individuano (almeno) una sequenza principale di tre azioni compiute da un utente del prodotto:

1. creazione di una DBN;
2. allenamento della DBN creata;
3. analisi della DBN (eventualmente non ancora allenata).

Il resto del capitolo analizza brevemente queste tre azioni, facendo dunque emergere gli elementi principali dell'interfaccia utente.

2.2 CREAZIONE DI UNA DBN

Per creare una DBN, sono necessari e sufficienti i seguenti parametri:

- numero di RBM che formano l'architettura della DBN;
- numero di nodi che formano ogni livello;
- intervallo entro cui inizializzare i pesi dei collegamenti della DBN.

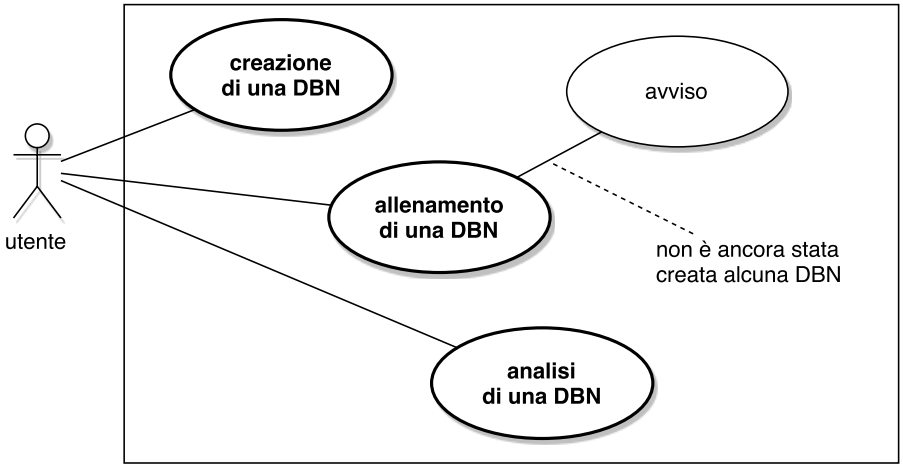

Figura 7: I tre principali casi d'uso di DBNsim.

Una DBN è una "pila" di *Restricted Boltzmann Machines* (RBM; cfr. sez. 1.1); è quindi necessario conoscere il numero di tali RBM, cioè la **profondità** della DBN. Oltre alla profondità della rete, bisogna anche specificare la **larghezza** (cioè il numero di nodi) di ogni suo livello.

Questi parametri specificano in modo univoco l'architettura della rete; rimane solo da specificare il valore a cui ogni arco pesato dev'essere inizializzato. Per fare ciò, l'uso è di inizializzare i pesi in modo casuale secondo una distribuzione normale centrata nello zero. Viene lasciata all'utente la scelta su quale sia la **deviazione standard** di tale distribuzione.

I diversi parametri necessari per creare una DBN (e i parametri per allenarla, come vedremo nella prossima sezione) suggeriscono di dialogare con l'utente tramite un *form* (modulo), con un campo dati per ogni parametro.

2.3 ALLENAMENTO DI UNA DBN

2.3.1 Parametri di apprendimento

Per allenare una DBN (la cui architettura sia stata definita) sono necessari e sufficienti i seguenti parametri:

- numero massimo di epoche;
- dimensione di ogni *mini-batch* in cui suddividere i dati di allenamento;
- coefficiente di apprendimento (*learning rate*);
- coefficiente di inerzia (*momentum*);
- coefficiente di decadimento dei pesi (*weight decay factor*).

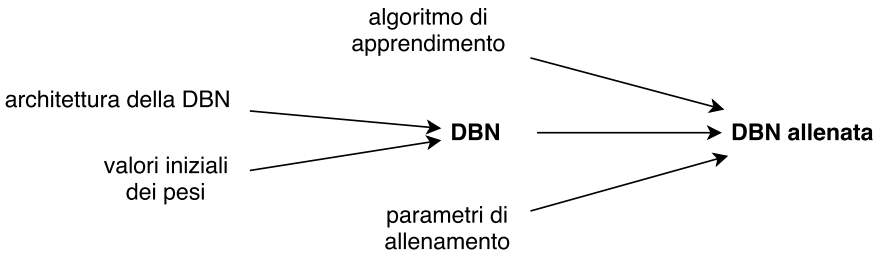

Figura 8: Creazione e allenamento di una DBN.

Ogni rete neurale (anche il nostro cervello) apprende tramite l'osservazione di esempi. Nel caso delle reti artificiali, questi esempi vengono raggruppati in un *training set*, il quale può essere osservato dalla rete un certo numero di volte; si dice **epoca** l'osservazione di un intero *training set*. Il numero massimo di epoche determina quindi quante opportunità avrà la DBN di osservare ciascun esempio.

Inoltre, per ridurre i tempi di allenamento, gli esempi mostrati alla rete vengono spesso raggruppati in piccoli **sottoinsiemi** (detti *mini-batch*). La dimensione di questi sottoinsiemi è un divisore del numero totale di esempi e dev'essere decisa dall'utente; dimensioni banali sono il numero totale di esempi (apprendimento *batch* o *one shot*) e 1 (apprendimento *online*).

Infine, l'algoritmo di allenamento di una DBN è parametrizzato su tre coefficienti: il coefficiente di apprendimento definisce la **plasticità** delle connessioni della rete (cioè la facilità con cui gli archi pesati della DBN si lasciano cambiare); il coefficiente di inerzia rappresenta l'**accelerazione** data all'allenamento quando molti esempi consecutivi sono simili tra loro; il coefficiente di decadimento dei pesi rappresenta il **grado di saturazione** a cui è soggetto l'aumento dei pesi (che altrimenti potrebbero aumentare a dismisura).

2.3.2 Algoritmo di apprendimento

Data l'architettura di una DBN, un *training set* e i parametri necessari per allenarla, la rete può iniziare ad apprendere, per mezzo dell'algoritmo *Contrastive Divergence*.

L'implementazione di questo algoritmo, come specificato nei requisiti in appendice A, è in Python. L'analisi preliminare che espongo in questo capitolo è stata preceduta dall'implementazione (sempre in Python) di un piccolo prototipo di algoritmo, che è stato poi il punto di partenza per l'implementazione attuale dell'algoritmo.

L'utilizzo di Python come linguaggio di implementazione è comodo e naturale: ha una sintassi concisa e le sue librerie sono efficienti. Invece, la prossima sezione evidenzierà come Python non sia il candidato migliore per creare un'interfaccia grafica.

2.4 ANALISI DI UNA DBN

Per analizzare una DBN (allenata o in fase di allemento) alcuni tra i migliori indicatori [vedi 7, sez. 15] sono:

- visualizzare l'architettura della rete;
- visualizzare la matrice dei pesi di ogni RBM nella DBN[1];
- visualizzare uno o più esempi di input con cui viene allenata la rete (utile soprattutto quando gli esempi sono immagini);
- visualizzare i campi recettivi di uno o più nodi della rete;
- visualizzare un istogramma dei pesi della rete.
- visualizzare l'andamento della distanza tra gli esempi del *training set* e la loro ricostruzione da parte della rete, man mano che essa apprende.

Notiamo che tutti questi sono indicatori grafici, visuali. Il motivo di ciò, a mio parere, è che una rappresentazione grafica (piuttosto che numerica, ad esempio) semplifica di molto l'enorme complessità che caratterizza le reti neurali e permette quindi di analizzarle con maggior facilità. Ad esempio, è poco interessante il valore numerico della distanza tra gli esempi del *training set* e la loro ricostruzione; piuttosto, è utile disegnare un grafico dell'andamento di tale distanza durante l'allenamento, per assicurarsi che essa diminuisca anziché crescere.

L'importanza della grafica è stata uno dei principali motivi che mi hanno spinto a trasformare il software in un'applicazione web; oltre a questo, come vedremo tra poco, un altro motivo è stata la naturale orientazione agli eventi di JavaScript.

DBNsim implementa tutti i punti appena elencati, tranne la matrice dei pesi — che è comunque molto meno eloquente di una rappresentazione "a grafo" della DBN. La scelta di implementare questi indicatori procede direttamente dai requisiti specificati in appendice A. Secondo questi, un utente doveva poter:

1. visualizzare schematicamente la struttura della rete;
2. monitorare l'apprendimento della rete;
3. visualizzare alcuni esempi di input forniti alla rete;
4. visualizzare i campi recettivi.

Nei prossimi paragrafi tratto di questi quattro punti, concentrandomi più sulla progettazione che sull'implementazione.

1 Una RBM è un grafo bipartito, rappresentabile quindi con una matrice le cui righe sono i neuroni nascosti e le cui colonne sono i neuroni visibili.

2.4.1 Struttura della rete

La struttura della rete viene decisa a partire dai parametri inseriti dall'utente. Per visualizzare dunque tale struttura, sembrava una buona idea aggiornare automaticamente, man mano che l'utente riempiva o modificava i campi, un **grafo** che rappresentasse l'architettura della DBN. Questa funzionalità (poi implementata) è un esempio di come l'orientazione agli eventi di JavaScript abbia spinto a convalidare l'idea della *web app*, a discapito di un'applicazione desktop.

2.4.2 monitoraggio dell'apprendimento

Per monitorare l'apprendimento della DBN, la soluzione più semplice per l'utente è visualizzare un **grafico** che riporta l'andamento dell'errore di ricostruzione della rete, man mano che la rete apprende. Ho quindi messo in conto di implementare un tale grafico, in cui tracciare l'errore di ricostruzione in funzione del numero di epoche trascorse.

Benché non menzionato nei quattro requisiti che ho appena elencato, il mio tutor mi ha chiesto di mostrare anche un **istogramma** dei pesi della rete. Il motivo di questa richiesta è che la distribuzione dei pesi di una rete che apprende bene (come spiegato anche in [7]) dovrebbe diventare poco a poco più stretta e più alta.

2.4.3 Esempi di input e campi recettivi

Per quanto riguarda gli esempi di input da cui apprende la rete, dobbiamo notare che essi non sono altro che dei lunghi vettori di n numeri dove ogni numero viene presentato a uno degli n neuroni visibili della DBN. Farsi un'idea di un tale vettore potrebbe sembrare complicato ma, non a caso, la stragrande maggioranza dei *training set* da cui apprendono le DBN sono insiemi di **immagini**. È quindi possibile mostrare all'utente un vettore di input rappresentandolo come immagine; ad esempio, un vettore da 900 unità può essere mostrato come un'immagine quadrata larga (e alta) 30 pixel.

I campi recettivi del punto 4, infine, sono un indicatore tanto interessante quanto sorprendente. Si dice **campo recettivo** di un neurone il vettore di input che lo stimola maggiormente. Ogni neurone può attivarsi con probabilità da 0 a 1 e il suo campo recettivo lo farà attivare con probabilità molto vicina a 1. Il bello è che i campi recettivi, come i vettori di input, possono essere rappresentati come immagini. Per questo, una DBN che apprende da un insieme di immagini contenenti numeri svilupperà dei campi recettivi che assomigliano a dei numeri; un'altra che osserva un gran numero di facce umane avrà dei campi recettivi che sembrano dei fantasmi. Sono spesso molto belle,

oltre che interessanti, queste “idee platoniche” di numero, di faccia o di ciò che la rete apprende.

3 DAL DESKTOP AL WEB

In questo capitolo analizzo ulteriormente il problema, concentrandomi sugli aspetti che hanno spinto a progettare una ***web app*** anziché un'applicazione desktop. Infine, introduco il progetto scaturito da questa analisi ed espongo brevemente l'implementazione attuale del prodotto.

3.1 INDIPENDENZA DAL SISTEMA OPERATIVO

Oltre ai requisiti funzionali appena analizzati, il laboratorio era interessato anche ad essere il più possibile slegato dalle specifiche del sistema operativo — cioè ad avere un'applicazione *cross-platform*.

Mentre la parte logica dell'applicazione non dava nessun problema di compatibilità, l'interfaccia utente richiedeva di trovare un *framework* grafico per Python che astraesse il più possibile dai vari sistemi operativi. I risultati della mia ricerca su un tale *framework* hanno trovato solo prodotti poco documentati o troppo poco stabili.

Mi venne quindi l'idea di implementare l'applicazione come *web app* — naturalmente con un server in Python, per rispettare i requisiti e per mantenere tutti i vantaggi che questo linguaggio offre in termini di efficienza e di calcolo scientifico.

3.1.1 Dal desktop al web

Passare dal progetto di un'applicazione desktop a quello di un'applicazione web non è affatto difficile se, nel progettare la versione desktop, si è prestato attenzione a tenere separati l'interfaccia utente e la *business logic* "algoritmica". Inoltre, se si progetta sin dall'inizio seguendo un approccio *top-down*, viene naturale tenere separati questi due aspetti. Ecco perché lo sviluppo del mio prodotto si è giovato di questo cambio, apparentemente repentino.

Di contro, per inciso, è interessante notare che non vale l'opposto. Un progetto che partisse da subito con l'idea di usare il browser per poi voler passare al desktop potrebbe trovarsi in seria difficoltà: ad esempio, distinguere un *front end* e un *back end* partendo da un groviglio di selettori *jQuery* non è compito facile.

3.1.2 Vantaggi di una *web app*

Nel mondo attuale, l'architettura che sta dietro alla maggior parte delle applicazioni web è il modello *client-server*. Questo modello non è affatto recente e potrebbe lasciare presto il posto ad altri modelli più giovani, basati sulla distribuzione e sul *peer-to-peer*; tuttavia, un'architettura *client-server* ha ancora molti pregi da offrire a chi la usa: di seguito espongo i principali vantaggi che ne ho ricavato durante la progettazione di DBNsim come *web app*.

VINCOLI Quando l'interfaccia utente e la logica sottostante sono molto diversi tra loro, tenerli separati è una scelta obbligata. Nel mio caso l'interfaccia utente faceva ampio uso di librerie grafiche, mentre il *back end* era puramente algoritmico (faceva solo calcoli matriciali); separare il *back end* dal *front end* equivaleva quindi a realizzare un sano disaccoppiamento tra due componenti scorrelate. Il fatto di progettare un'applicazione web non solo mi ha suggerito ma, addirittura, mi ha **vincolato** a tenere sempre separati l'interfaccia utente e i calcoli matriciali: quand'anche progettavo una funzionalità in modo da rompere questo disaccoppiamento, il modello *client-server* ne impediva subito l'**implementazione**.

CREAZIONE DI UN PROTOTIPO Altro vantaggio della programmazione web è la rapidità con cui si può sviluppare un prototipo dell'applicazione. Si può partire da un semplice scheletro in HTML e studiarlo finché non diventa necessario gestire eventi o creare animazioni; a questo punto si può introdurre JavaScript — senza dover toccare il codice HTML, dato che si può scrivere il codice JavaScript in un file separato. Infine, poco a poco, CSS viene in aiuto per i dettagli grafici e per farsi un'idea dell'apparenza del prodotto finale.

Questo scheletro di applicazione può accompagnare la progettazione e può essere il punto di partenza per implementare il progetto.

JAVASCRIPT Python è senz'altro uno dei linguaggi più belli per scrivere algoritmi: è semplice (sembra pseudocodice) e dispone di una miriade di librerie efficienti. Tuttavia non è certamente la prima scelta per chi vuole creare un'interfaccia grafica con elementi complessi (grafi, grafici...), per giunta a eventi e *cross-platform*.

Il fatto di sviluppare un'applicazione web, invece, mi ha messo a disposizione quello che è forse lo strumento più usato, in un sistema operativo, per la grafica e gli eventi: il *web browser*. Il suo linguaggio nativo, JavaScript, offre numerose librerie grafiche e gestice gli eventi dell'utente in modo naturale.

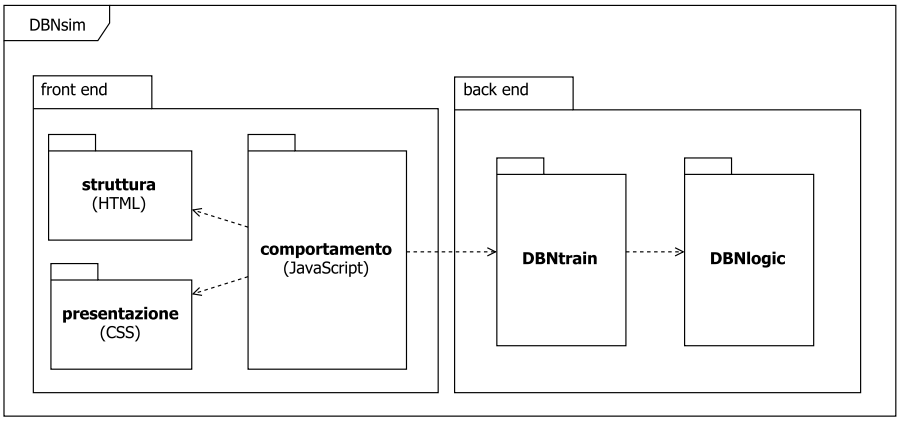

Figura 9: Architettura, ad alto livello.

3.2 COME SEMPLIFICARE L'INTERFACCIA?

La creazione e l'allenamento di una DBN necessitano di non pochi parametri; inoltre, l'analisi di tale DBN si può compiere osservando un gran numero di indicatori. L'interfaccia utente può quindi presto diventare disordinata — disordinata per l'utente ma anche per lo sviluppatore.

Per quanto riguarda l'utente, un modo semplice per avere un'interfaccia grafica ordinata e intuitiva è di progettare l'applicazione web come *pagina singola*. Questo può essere dannoso per lo sviluppatore, che deve gestire più codice in una sola pagina anziché distribuirlo in molte pagine. Tuttavia, se chi sviluppa sta attento, questo può essere una sorta di vantaggio poiché obbliga ogni singolo problema ad essere affrontato in modo analitico e a mettere in discussione le scelte precedenti.

Ad ogni modo, la separazione tra struttura, stile e comportamento incoraggiata dal modello HTML+CSS+JavaScript è un ottimo strumento per superare le difficoltà della pagina singola.

3.3 DESCRIZIONE DELL'ARCHITETTURA ATTUALE

DBNsim si compone dunque, nell'implementazione attuale, di due sotto-programmi: un *back end* scritto in Python e un *front end* in JavaScript. Questi due programmi possono girare su macchine indipendenti; in particolare, la macchina ospitante il *back end* può essere equipaggiata con una GPU *NVIDIA*™ se si vuole accelerare l'apprendimento di grandi reti neurali.

Prima di spiegare l'architettura di queste due componenti, è bene osservare la figura 9 per avere un'idea generale di com'è organizzato il tutto: l'interfaccia utente è divisa secondo la classica triade

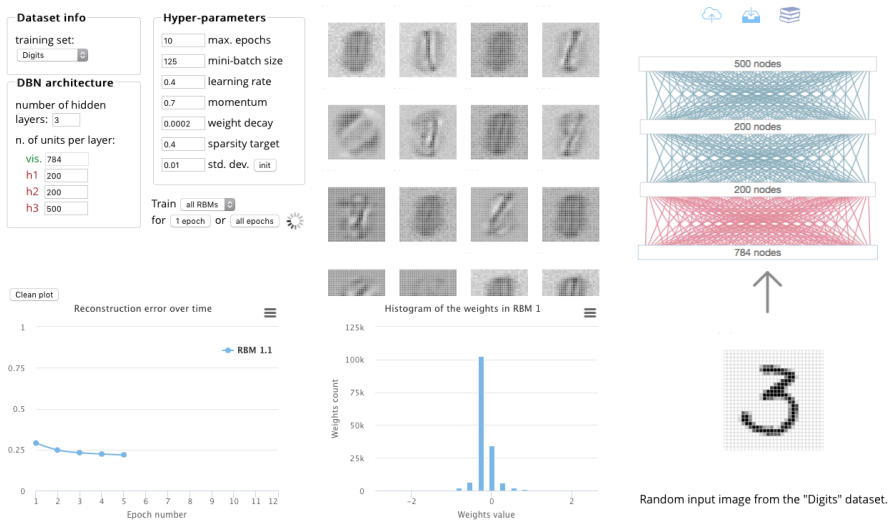

Figura 10: Interfaccia utente.

dei linguaggi web[1], mentre il *back end* (scritto in Python) si compone di un'interfaccia HTTP e di un modulo dedicato alla manipolazione delle DBN.

3.3.1 Interfaccia utente

Il *front end* di DBNsim è una singola pagina web scritta in XHTML e decorata con un foglio di stile CSS. Lo stile della pagina segue un *layout* fisso diviso in "quadranti", ognuno contenente dei campi dati, dei grafici o delle immagini interattive. La figura 10 mostra che l'interfaccia è divisa in sei quadranti (due righe da tre colonne); browser più piccoli visualizzano uno stile diverso, non fisso ma altrettanto completo.

ELEMENTI DELL'INTERFACCIA I sei quadranti dell'interfaccia corrispondono a sei elementi con funzionalità distinte. Tra questi elementi, il primo con cui interagisce l'utente è l'insieme dei **campi dati** in alto a sinistra: modificando questi campi l'utente può creare una rete e impostare i parametri per allenarla. Mentre l'utente modifica i campi, DBNsim aggiorna automaticamente il **grafo** in fig. 11.

Una volta lanciato un allenamento, entra in gioco il grafico dell'**errore di ricostruzione** (fig. 12), che viene aggiornato ad ogni epoca di allenamento. Questo grafico disegna una curva *per ogni RBM*; quindi ad ogni DBN corrispondono più curve, una per ogni sua RBM.

Durante l'allenamento vengono aggiornati anche i due elementi che stanno in centro alla pagina:

1 HTML per strutturare il contenuto, CSS per presentarlo, JavaScript per renderlo interattivo con l'utente.

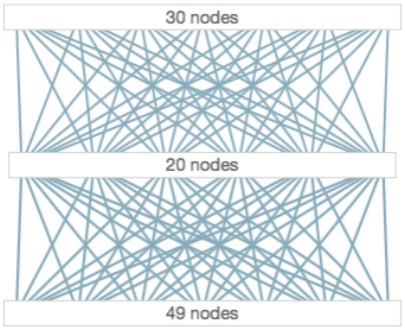

Figura 11: Grafo della DBN che l'utente crea.

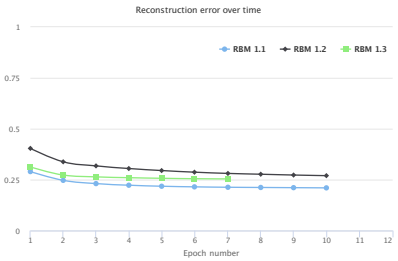

Figura 12: Grafico dell'errore di ricostruzione.

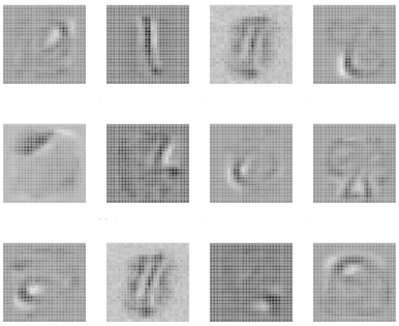

Figura 13: Riquadro con i campi recettivi di alcuni neuroni.

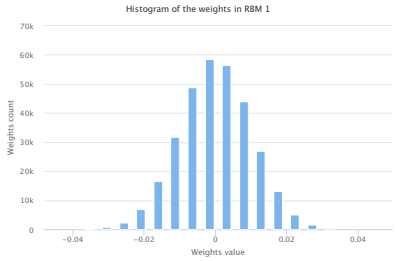

Figura 14: Istogramma dei pesi di una RBM.

- i **campi recettivi** di alcuni neuroni selezionati dalla RBM che sta apprendendo (ricordiamo che l'apprendimento di una DBN procede *una RBM per volta* — vedi sez. 1.1.2), come in fig. 13;
- l'**istogramma** dei pesi della RBM che sta apprendendo (fig. 14), cioè come sono disribuiti i pesi delle connessioni di una RBM.

Infine, ad ogni click dell'utente sul livello visibile del grafo (il livello più basso del grafo) DBNsim mostra un **esempio di input** preso a caso dal *training set* su cui si allena la DBN.

COMPORTAMENTO DELL'INTERFACCIA La parte più complessa dell'interfaccia utente è il suo comportamento. Questo viene gestito da una serie di funzioni JavaScript, come in figura 15.

Ma è evidente che la figura 15 non descrive in modo completo il comportamento del programma: infatti il grafo non è completamente connesso e ci sono addirittura dei nodi senza archi. Questo perché abbiamo usato un approccio esclusivamente **imperativo**; infatti la figura riporta soltanto le chiamate tra una funzione e un'altra, trascurando la componente ***event-driven*** del comportamento del programma, cioè le azioni dell'utente — giacché ogni funzione può essere chiamata non solo da altre funzioni ma anche da eventi dell'utente. La figura

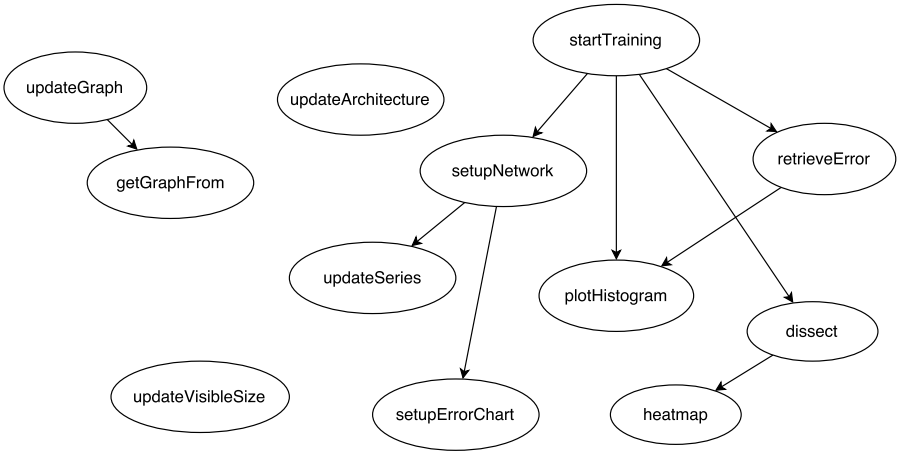

Figura 15: Interazioni tra le funzioni del *front end*: ogni elemento è una funzione e una freccia da A a B significa che A chiama B.

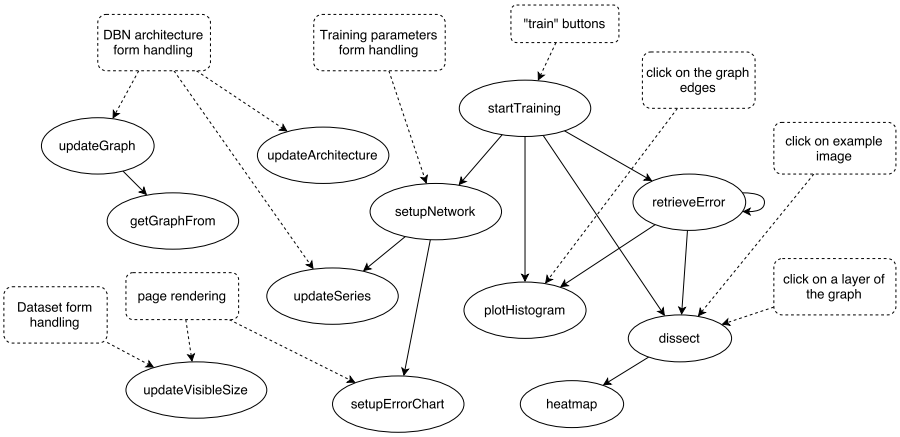

Figura 16: Interazioni tra gli eventi scatenati dall'utente (rettangoli tratteggiati) e le funzioni JavaScript (ovali).

16 completa quindi il nostro grafo, introducendovi le principali azioni dell'utente.

In quest'ultimo diagramma (non UML, in quanto il *front end* non è a oggetti) le principali funzioni JavaScript sono le seguenti:

- `updateGraph` aggiorna il grafo che rappresenta la DBN;
- `setupErrorChart` aggiorna il grafico dell'errore di ricostruzione;
- `startTraining` chiede a `setupNetwork` di creare una nuova DBN sul server e inizia ad allenare la rete chiamando `retrieveError`;
- `setupNetwork` chiede al server di creare una nuova DBN e di impostare i parametri con cui essa verrà allenata;
- `retrieveError` chiede al server di allenare la DBN per un'epoca.

- `plotHistogram` e `dissect` permettono di analizzare una rete mostrando un istogramma dei pesi, i campi recettivi e gli esempi di input.

TECNOLOGIE Il *front end* è implementato usando le seguenti tecnologie:

- XHTML 1 *strict*, per strutturare il contenuto della pagina.
- CSS3, per posizionare gli elementi della pagina in modo relativo alle dimensioni del browser.
- JavaScript, con le seguenti librerie:
  - *jQuery* [3], per semplificare l'interazione con gli elementi HTML e con le richieste asincrone verso il *back end*;
  - *Cytoscape.js* [5], per rappresentare la DBN come un grafo interattivo;
  - *Highcharts* [6], per disegnare grafici in modo dinamico.

3.3.2 *back end*

Il *back end* si appoggia sul *framework* Django [2], lo strumento più usato per scrivere applicazioni e siti web con Python, e si struttura quindi secondo le indicazioni di tale *framework*. In particolare, Django richiede di suddividere un sito in "applicazioni", le quali possono essere composte da più pagine web; nel mio caso, ho deciso di creare un'unica applicazione composta da un'unica pagina web.

Ogni applicazione è una mappa dove le chiavi sono URL e i valori sono risposte del server al client: ad ogni URL, in pratica, lo sviluppatore deve associare una funzione Python che accetta una richiesta (HTTP, ad esempio) come argomento e restituisce una risposta come valore. Se serve, una risposta può avvalersi di qualche *template* (cioè una pagina HTML inframmezzata di parole chiavi da sostituire con determinati valori) e di file statici (immagini, fogli di stile, script...).

Le funzioni che rispondono alle varie URL vanno messe tutte in **un solo file**, che Django controlla per capire come rispondere alle richieste. Naturalmente, questo file sarebbe diventato presto ingestibile se gli avessi fatto contenere tutta la *business logic* del *back end*. Ecco perché ho deciso di creare un *package* a parte, esterno alla mia applicazione Django ma interno al sito (quindi visibile alle varie funzioni che gestiscono le richieste di un client, di cui sopra).

Il *back end* di DBNsim è quindi diviso in due componenti:

- l'applicazione Django `DBNtrain`, che risponde alle richieste di un client — richieste per ottenere una pagina web, allenare una rete, recuperare informazioni relative ai dataset...

- il *package* `DBNlogic`, che offre un insieme di classi che implementano la *business logic* di DBNsim — ad esempio le classi `DBN`, `CDTrainer`, `DataSet`...

`DBNlogic` è, di proposito, indipendente dal resto del programma e può essere usato come libreria per creare e allenare DBN da riga di comando.

DBNTRAIN L'applicazione che fa da interfaccia con un client, quindi, è raggiungibile dall'indirizzo `/DBNtrain` del sito ospitante. Ad esempio `example.com/DBNtrain/index/` chiama la funzione `index` dell'applicazione `DBNtrain`; questa funzione popola il *template* index.html e lo restituisce come risposta HTTP. Invece, `example.com/DBNtrain/getReceptiveField/?layer=1&neuron=0` chiama `getReceptiveField` per chiedere il campo recettivo del primo neurone nel primo livello nascosto della DBN.

Ecco le richieste (tutte HTTP) a cui il server può rispondere:

- `index/` restituisce il *template* index.html, popolato con opportuni campi;
- `train/` costruisce una nuova DBN e imposta i parametri per allenarla;
- `getError/` allena una DBN (specificata nella richiesta) per una sola epoca;
- `getReceptiveField/` restituisce il campo recettivo di un particolare neurone;
- `getHistogram/` restituisce un istogramma di come sono distribuiti i pesi in una particolare RBM;
- `saveNet/` restituisce una particolare DBN, serializzata nel formato Pickle (un formato di Python);
- `getInput/` restituisce un particolare esempio di input da un particolare *training set*.

La figura 17 illustra come sia possibile inviare alcune richieste solo dopo averne eseguite altre; ad esempio, non è possibile richiedere il campo recettivo di un neurone con `getReceptiveField/` se il client non ha ancora chiesto di creare una DBN sul server con `train/`.

DBNLOGIC Il *package* che gestisce il funzionamento delle DBN è suddiviso in quattro moduli (vedi figura 18):

- `nets` espone una classe `DBN` e una classe `RBM`, i soggetti principali di DBNsim;
- `sets` offre la classe `DataSet`, un semplice contenitore di esempi di input;

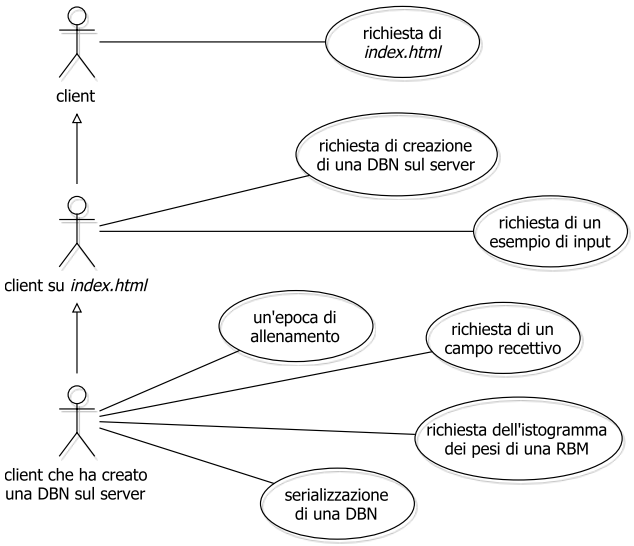

Figura 17: Casi d'uso del *back end* da parte di un client.

- `train` offre la classe *strategy* `CDTrainer` che esegue l'algoritmo di allenamento *Contrastive Divergence* su una RBM (è compito della DBN chiamare `CDTrainer.run` per ognuna delle proprie RBM);
- `util` definisce la classe `Configuration` (che raggruppa tutti i parametri necessari per l'apprendimento di una rete) e alcune funzioni di utilità (`sigmoid`, `activation` e `heatmap`).

Inoltre, `DBNlogic` presenta due directory per memorizzare dati:

- `data/` permette di immagazzinare e recuperare i *training set*, disponibili in tre formati: Pickle, CSV e MATLAB.
- `nets/` permette di salvare e recuperare le DBN, serializzate nel formato Pickle.

TECNOLOGIE Queste le principali tecnologie usate dal lato Python di DBNsim:

- il linguaggio Python 2;
- il *web framework* Django;
- le seguenti librerie Python:
  - `numpy`, per il calcolo scientifico;
  - `gnumpy` [14], una "traduzione" di `numpy` per GPU;
  - `cudamat` [10], per compiere calcoli matriciali su GPU che supportano CUDA;
  - `npmat`, un *fallback* usato da `gnumpy` quando non può usare la GPU e deve quindi appoggiarsi alla CPU;

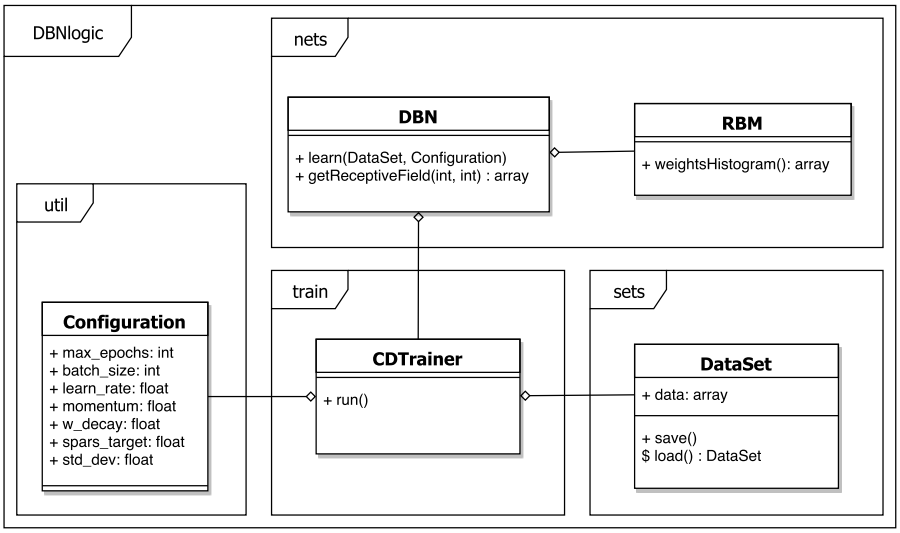

Figura 18: Diagramma UML delle classi per i quattro moduli di `DBNlogic`.

  - `pickle`, per serializzare e de-serializzare oggetti Python;
  - `json`, per codificare e decodificare stringhe JSON;
  - `random`, per generare numeri pseudo-casuali.
- una GPU, se si vuole gestire DBN di grandi dimensioni.

Perché Python 2 e non Python 3? Ebbene `gnumpy` è disponibile solo per Python 2; un mio tentativo iniziale di usare direttamente `cudamat` è fallito per mancanza di tempo. Nella sezione 4.3 esporrò le conseguenze di questo contrattempo.

4 MODELLO DI SVILUPPO DEL PRODOTTO

Il processo di sviluppo del prodotto è stato pianificato in base a precise scadenze temporali e ha seguito un **modello**. Espongo qui le motivazioni alla base della scelta di questo modello e aggiungo delle osservazioni su come si è svolto lo sviluppo del software.

4.1 SCELTA DEL MODELLO

Il rispetto di scadenze temporali (per il tirocinio) ha motivato la ricerca di un modello da seguire per lo sviluppo del prodotto.

La ricerca di un modello è partita dalla scelta tra le due grandi famiglie di modelli esistenti: il modello sequenziale e quello iterativo[1]. L'opportunità di poter dialogare di frequente con il mio tutor al laboratorio mi ha spinto a indagare esclusivamente la famiglia dei modelli iterativi. Inoltre, mentre i modelli sequenziali suddividono lo sviluppo per *attività*, quelli iterativi lo suddividono per *funzionalità* [4, cap. 2]. Una suddivisione per funzionalità si addiceva bene al mio prodotto, che è progettato in maniera modulare.

Difatti, questo ci offre uno spunto di riflessione sul perché i software moderni prediligano uno sviluppo iterativo a uno sequenziale: lo sviluppo per attività (sequenziale) va bene per un programma monolitico, com'erano un tempo tutti i software; ma la programmazione a oggetti, la separazione tra struttura e presentazione, il modello *client-server* rendono i software molto più modulari di quanto lo fossero un tempo, quindi molto più ricchi di tante piccole funzionalità separate. Questo giustifica quindi una tendenza a sviluppare per funzionalità, con un modello iterativo.

Il modello di sviluppo iterativo che ho seguito si ispira ai modelli di sviluppo **incrementale** e **Chaos**. Ho previsto di ripetere tre periodi: analisi, progettazione, implementazione. Man mano che il prodotto avanza, analisi e progettazione sono sempre più legate (cioè le soluzioni ai vari problemi che si incontrano sono sempre più immediate). In ogni momento, lo sviluppo cerca di seguire il motto del modello Chaos "always resolve the most important issue first" [15] e cerca di integrare continuamente il codice in un *repository* centralizzato. L'integrazione continua del codice esegue dei *test* sul codice Python, per assicurare che il cuore del programma sia corretto; una verifica automatizzata del codice JavaScript sarebbe stata certamente utile ma il poco tempo a disposizione mi ha spinto a eseguire solo prove manuali.

1 Martin Fowler parla di "iterative/waterfall dichotomy" [4, cap. 2].

4.2 UNA PARTICOLARITÀ EMERSA DURANTE LO SVILUPPO

Lo sviluppo del mio software è partito dalla modellazione delle DBN e del loro algoritmo di apprendimento; tuttavia, le DBN tendono ad essere molto difficili da analizzare con riga di comando, senza interfaccia grafica. Questo perché alcuni *bug* emergono solo con reti di grandi dimensioni, dove un piccolo errore si accumula e si manifesta come *overflow* numerico o come pesi troppo grandi.

Mi sono quindi trovato nella situazione di dover visualizzare grafici e campi recettivi per poter fare *debugging* di alcune DBN (con qualche libreria grafica in Python)... ma questo era esattamente il compito dell'interfaccia utente del mio software!

Ecco perché ho trovato molto comodo, anche se può sembrare strano, sviluppare l'interfaccia grafica **ancor prima** di avere un algoritmo di apprendimento corretto e validato. Inizialmente l'interfaccia grafica era, appunto, in Python. Tuttavia non è stato difficile trasporre i concetti (grafico, istogramma, campo recettivo...) dalle librerie Python a quelle JavaScript.

4.3 *downgrade* DA PYTHON 3 A PYTHON 2

Come ho accennato, DBNsim era inizialmente in Python 3 ma ha poi dovuto "scendere" a Python 2. Motivo del *downgrade* è stata la mia scarsa conoscenza delle GPU.

All'inizio, infatti, volevo usare direttamente la libreria `cudamat` che ho citato nella sezione 3.3.2; questa permetteva di tradurre codice Python in codice CUDA. Tuttavia, ben presto, ho dovuto capire che programmare bene per GPU è possibile solo se si conoscono anche i dettagli (o meglio i principi sottostanti) di queste architetture così diverse dal computer classico di Von Neumann.

Ho scelto allora di affidarmi alla libreria `gnumpy`, che si appoggia a sua volta a `cudamat` ma presenta un'interfaccia più ad alto livello; tra l'altro l'autore di questa libreria era interessato al Deep Learning, quindi `gnumpy` mi ha permesso di implementare con grande facilità una versione efficiente dell'algoritmo *Contrastive Divergence*.

Purtroppo `gnumpy` era scritta per Python 2, che non è *forward-compatible* con Python 3. Per questo ho dovuto attuare qualche accorgimento per far eseguire il mio programma da un interprete Python 2. Fortunatamente, le differenze sintattiche tra i due linguaggi sono pochissime.

5 CONCLUSIONI

Il presente capitolo spiega cosa apporta di nuovo DBNsim, tira le somme di quel che ho imparato e, più in generale, di quel che un informatico può imparare dallo studio del cervello con le reti neurali.

5.1 RISULTATI

DBNSIM COME STRUMENTO DIDATTICO La modellazione del cervello tramite reti neurali si trova al crocevia tra numerose branche del sapere: neurologia, psicologia, informatica, matematica, statistica... Ciò implica che chi si approccia a questo argomento non si sta semplicemente "specializzando" in una branca del proprio campo di studi, bensì deve assimilare una grande mole di concetti provenienti dalle discipline più disparate. Ad esempio, alcune architetture di rete neurale sono spiegate efficacemente come sistemi fisici; altre sono più facili da immaginare come algoritmi; altre ancora come strutture presenti in biologia.

È quindi necessario trovare un modo per spiegare concisamente ma chiaramente i concetti su cui si basano le reti neurali artificiali, tenendo presente che chi spiega si può trovare davanti a un **pubblico eterogeneo**. Proprio in questa direzione va DBNsim, che offre numerosi strumenti visivi e interattivi da affiancare a un corso o a una divulgazione che voglia introdurre le *Deep Belief Networks*.

Inoltre, la preponderanza della grafica rende DBNsim uno strumento utile per sconfiggere la complessità e l'apparente caos che caratterizza le DBN.

CREAZIONE DI PROTOTIPI Oltre a essere uno strumento didattico, DBNsim permette anche di creare facilmente prototipi di DBN. Questo può essere utile a un ricercatore che non disponga immediatamente di un proprio strumento per allenare DBN o che non conosca ancora una libreria efficiente adatta a tale scopo.

Facendo girare l'applicazione in locale e allenando qualche volta una DBN, uno studente o ricercatore può trovare in breve tempo quali sono i parametri migliori per l'apprendimento; quando li avrà trovati, potrà utilizzare questi parametri in qualsiasi altro strumento o libreria di *deep learning*, anche costruendo da sé un software per analizzare qualche dettaglio di suo interesse: usando DBNsim, non dovrà mescolare la ricerca dei migliori parametri di allenamento con il *debugging* del proprio software.

Inoltre, la facilità con cui si può allenare una rete su DBNsim viene incontro ai bisogni di chi, come psicologi e medici, sarebbero interessati a simulare il cervello con modelli computazionali ma non hanno tempo di imparare a scrivere programmi per GPU [13].

BASE PER UN'APPLICAZIONE PIÙ RICCA L'interfaccia grafica del programma è lungi dall'essere una guida completa al *deep learning*. Tuttavia, il *back end* implementa una versione ragionevolmente efficiente e completa dell'algoritmo di apprendimento di una DBN; questo significa che sviluppi ulteriori dell'applicazione sono limitati soltanto dalla fantasia e dagli interessi di chi vorrà contribuire.

In somma, *back end* e *front end* di DBNsim possono essere la base per un'applicazione più ricca.

5.2 LE SFIDE SCIENTIFICHE POSTE DAI MODELLI NEURALI

5.2.1 Reti neurali e interpretabilità

Con i grandi avanzamenti compiuti nel campo dell'intelligenza artificiale, iniziano a sorgere anche problemi etici. Ad esempio, ci si chiede quanto sia giusto che una macchina svolga gratuitamente un compito per cui un essere umano viene pagato.

Tra i vari aspetti etici, il più legato al mio tirocinio è forse quello dell'*interpretabilità* delle reti neurali. Con questo termine intendo la **facilità di comprensione** del funzionamento di una rete neurale da parte di un umano [9]. Una rete neurale, infatti, può essere tanto utile (e affascinante) quanto incomprensibile nel suo funzionamento.

Ad esempio, il *deep learning* è stato recentemente usato in campo medico per prevedere l'insorgere di patologie nei pazienti di un ospedale di New York [8]. I risultati dell'esperimento, giudicati ottimi dal gruppo di ricerca dell'ospedale che se ne è occupato, erano tuttavia inquietanti: anticipavano, tra l'altro, l'insorgere della schizofrenia, una malattia che i medici trovano difficoltà nel prevedere. È bene fidarsi della rete neurale e considerare i pazienti come a rischio? quali prove può addure, la rete, in favore delle sue previsioni? Ecco perché il campo dell'intelligenza artificiale sta iniziando a cercare dei modi per far sì che strumenti basati sull'intelligenza artificiale riescano anche a spiegare ciò che fanno e *perché* lo fanno.

5.2.2 Potenza *versus* stabilità/affidabilità

Sin dall'inizio, i computer sono serviti ad automatizzare funzioni tipicamente eseguite da esseri umani: memorizzare informazioni, comunicare con i propri simili, fare previsioni, fare calcoli matematici... In

breve tempo (dagli anni Quaranta a oggi) i computer sono diventati sempre più potenti e versatili.

Ma possiamo dire che la tecnologia è diventata anche più **stabile** di com'era prima dell'avvento dei computer? Senza dubbio i computer ci permettono di risolvere problemi più complicati e in modo più rapido; ma possiamo affermare, ad esempio, che lo facciano in modo numericamente più stabile di un cervello umano?

Esiste, a mio avviso, un compromesso tra **potenza** di uno strumento (la grandezza del lavoro che esso permette di compiere in un dato periodo di tempo) e **stabilità** di esso: quanto più importante è il lavoro (o la funzione) che uno strumento ci consente di compiere, tanto più è difficile far sì che questo strumento sia stabile, prevedibile. Per "stabilità" intendo il familiare concetto relativo al calcolo numerico: uno strumento è stabile quando una piccola differenza negli input non rischia di generare una grande differenza negli output.

Se questo compromesso tra potenza e stabilità è un problema, una (non certo l'unica) soluzione ci è offerta proprio dai modelli neurali, dato che esse sono sia potenti sia stabili:

1. Una rete neurale è "potente", cioè può imparare qualsiasi compito umano — questo è banale, poiché il cervello umano è esso stesso una rete di neuroni.

2. Le principali architetture di reti neurali finora conosciute sono estremamente stabili. Per farsi un'idea di quanto questo sia vero, basti pensare che le reti neurali vengono impiegate sempre più spesso per compiti come il riconoscimento facciale o lo *speech-to-text*, che mappano molti input diversi ma **simili** tra loro (ad es. varie forme della lettera "A" manoscritta) in un unico output (ad es. il carattere "A"): molti input con piccole differenze vengono associati allo stesso output.

5.2.3 Reti neurali e Scienza

In un mondo così ricco di dati — in un mondo in cui è così facile reperire informazione — il campo dei modelli neurali può dare a uno scienziato molti spunti di riflessione.

Una rete neurale, in quanto strumento che cerca di **modellare** dei dati, si scontra "abitualmente" con una mole imponenente di essi. Questi dati sono talvolta sbagliati, spesso ridondanti e quasi sempre rumorosi ("sporchi"). Qual'è la chiave con cui una rete neurale riesce a domare facilemente tutti questi dati approssimativi?

La risposta, sicuramente, non è univoca. Tuttavia, una delle caratteristiche che più di tutte sembra aiutare una rete neurale a costruirsi un buon modello a partire da una realtà approssimativa è il fatto

che essa non "analizza" i dati, bensì piuttosto li **sintetizza**[1]; cioè non compie un processo prettamente scientifico ma piuttosto analogico, che va per associazioni (analogie), anziché per cause ed effetti come farebbe uno scienziato rigoroso. In particolare, una DBN della mia applicazione scopre **correlazioni** tra i dati, mai causalità. Può sembrare azzardato preferire le correlazioni ai rapporti causa-effetto, ma la letteratura scientifica contemporanea non manca di critiche autorevoli al concetto di causalità (ad es. Pearl in [12]), nonché di innumerevoli dibattiti intorno al significato stesso del fare Scienza (ad es. [1]).

5.2.4 Tendenze nell'informatica

Con l'uso delle reti neurali artificiali, il computer emula il cervello per poter eseguire compiti che richiedono un alto grado di stabilità; ma una riflessione più attenta rivela che questo non è affatto l'unico punto di contatto tra computer e cervello. Consciamente o inconsciamente, le seguenti novità tecnologiche portano i calcolatori ad essere sempre più simili al nostro cervello:

- concorrenza;
- ridondanza.

CONCORRENZA Se il calcolatore di Von Neumann aveva una sola unità logica, i computer attuali ne hanno più di una e, talvolta, si appoggiano a strumenti speciali (le GPU) le cui unità logiche non sono tanto potenti quanto, piuttosto, **numerose**.

Questo rispecchia ciò che avviene in una rete neurale: i neuroni, pur essendo poco potenti e poco veloci a trasferire informazione, sono numerosissimi. (È interessante notare che i pacchetti Ethernet su un semplice cavo di rame viaggiano almeno 10^5 volte più velocemente di un impulso elettrico tra due neuroni.)

RIDONDANZA Si sente parlare sempre più spesso di *Internet of Things*, reti *peer-to-peer* e *filesystem* distribuiti. Un aspetto tra i tanti che accomunano queste tecnologie è il fatto di essere **basate sulla ridondanza**: cosa sarebbe *BitTorrent* se ogni file fosse custodito da uno e un solo utente?

Possiamo dire che la ridondanza rende la tecnologia più stabile, più affidabile. Come ho scritto più sopra (sez. 5.2.2), una rete neurale è spesso più stabile di un algoritmo "classico" che svolge lo stesso compito. Mi sembra che questa stabilità sia dovuta soprattutto al fatto che il riconoscimento di un determinato *pattern* (una particolare faccia, un particolare simbolo alfabetico...) non è compito di un singolo

1 Non a caso le DBN sono dette **modelli generativi**, in quanto generano rappresentazioni nello stesso dominio dei dati che osservano.

neurone specializzato ma di tutti quei neuroni che hanno imparato una sfaccettatura di quel *pattern*.

A SPECIFICA DEI REQUISITI DEL PRODOTTO

Questa appendice riporta i requisiti formali da cui è partito lo sviluppo del prodotto, suddivisi per importanza decrescente: obbligatori, desiderabili e opzionali. Il raggiungimento o meno di un obiettivo è segnato a margine.

A.1 REQUISITI OBBLIGATORI

1. Implementazione del software nel linguaggio Python. *sì*
2. Sviluppo di un'interfaccia grafica basilare per il simulatore dotata delle seguenti funzionalità:
   - addestramento di modelli di reti neurali basati su DBN; *sì*
   - possibilità di impostare graficamente i parametri del modello; *sì*
   - possibilità di monitorare l'apprendimento (errore di ricostruzione); *sì*
   - possibilità di scegliere due diversi *dataset* di apprendimento (MNIST, Numerosità); *sì*
   - possibilità di esecuzione su CPU e di esecuzione concorrente su singola GPU; *sì*
   - visualizzazione schematica della struttura della rete neurale; *sì*
   - visualizzazione delle matrici dei pesi[1] e dei campi recettivi; *sì*
   - visualizzazione di alcuni esempi di input forniti alla rete. *sì*
3. Robustezza e portabilità del simulatore:
   - test di funzionamento su ampio *set* di casistiche prototipiche; *sì*
   - test di funzionamento su diversi sistemi operativi (Windows, Linux, MacOS). *sì*
4. Modularità del software, per facilitare il successivo sviluppo di funzioni aggiuntive. *sì, a mio avviso*

1 Per "matrici dei pesi" ci siamo poi accorti, con il mio tutor, che si intendeva semplicemente i campi recettivi e non la matrice di adiacenza della rete.

A.2 REQUISITI DESIDERABILI

1. Sviluppo di estensioni all'interfaccia grafica:
   - Possibilità di scegliere altri dataset di apprendimento (Immagini naturali, Monosillabi...). *sì*
   - Visualizzazione dell'attività dinamica della rete:
     - attivazione dei neuroni nascosti; *no*
     - sviluppo progressivo dei campi recettivi; *sì*
     - processamento *bottom-up* e *top-down* di stimoli visivi. *no*
   - Possibilità di visualizzare il risultato di analisi più sofisticate:
     - regressione sull'attività neurale; *no*
     - test di selettività per i neuroni nascosti. *no*
   - Possibilità di caricare matrici dei pesi precedentemente addestrate. *abbozzato*
2. Possibilità di effettuare *joint training* con informazione supervisionata. *no*
3. Possibilità di addestrare classificatori lineari a vari livelli di rappresentazione, con relativa visualizzazione delle prestazioni (accuratezza di classificazione, matrici di confusione). *no*
4. Compilazione e creazione di un file eseguibile a partire dagli script del simulatore. *no*

A.3 REQUISITI OPZIONALI

1. Portabilità dell'interfaccia grafica su ambiente di sviluppo alternativo (MATLAB). *no*
2. Possibilità di salvare i dati relativi al modello e alle analisi effettuate in un file strutturato (*database*). *sì*
3. Implementazione di algoritmi di apprendimento alternativi (modelli con connessioni laterali, *autoencoders*, *backpropagation*). *no*

B CONTRASTIVE DIVERGENCE

Descrivo, prima in forma sintetica e poi completamente, l'algoritmo *Contrastive Divergence* per allenare le RBM che compongono una DBN. Infine riporto l'implementazione in Python dell'algoritmo.

B.1 ALGORITMO

Con il seguente algoritmo in pseudocodice, introduco alcune parti che per semplicità avevo omesso nella sezione 1.1.2:

- I vettori di *bias* vengono sommati ai livelli della RBM; tralascio le parti relative al loro aggiornamento, per non introdurre troppe variabili.
- Il coefficiente di inerzia accelera l'apprendimento, quando si incontrano sequenze di esempi che hanno tra di loro un effetto simile sulla matrice dei pesi.
- Il coefficiente di decadimento dei pesi è un modo per evitare la crescita indefinita dei pesi più grandi.

Require:
$\mathtt{dataset}$ = *l'insieme degli esempi di input*
v = *livello visibile della RBM attuale*
h = *livello nascosto della RBM attuale*
W = *la matrice dei pesi della RBM attuale*
a = *i* bias *del liv. visibile*
b = *i* bias *del liv. nascosto*
$\mathtt{max\_epochs}$ = *numero massimo di epoche*
η = *coeff. di apprendimento*
$\mathtt{momentum}$ = *coeff. di inerzia*
$w\_\mathtt{decay}$ = *coeff. di decadimento dei pesi*

$W\_\mathtt{update} \leftarrow$ matrice di zeri
$a\_\mathtt{update} \leftarrow$ vettore di zeri
$a\_\mathtt{update} \leftarrow$ vettore di zeri

$W \leftarrow$ pesi casuali
$a \leftarrow$ vettore di zeri
$b \leftarrow$ vettore di zeri

$\mathtt{epoch} \leftarrow 1$
while $\mathtt{epoch} \leqslant \mathtt{max\_epochs}$ **do**
  for all $ex \in \mathtt{dataset}$ **do**

$h \leftarrow \sigma(W \cdot ex + b)$
$C_+ \leftarrow ex \cdot h^T$
if $epoch = max\_epochs$ **then**
$ex \leftarrow h$
end if
$v \leftarrow \sigma(W^T \cdot v + a)$
$C_- \leftarrow h \cdot v^T$
$W\_update \leftarrow momentum \cdot W\_update + \eta \cdot (C_+ - C_-) - (w\_decay \cdot W)$
$a\_update \leftarrow [\dots]$
$b\_update \leftarrow [\dots]$
$W \leftarrow W + W\_update$
$a \leftarrow a + a\_update$
$b \leftarrow b + b\_update$
end for
end while

B.2 IMPLEMENTAZIONE IN PYTHON

Riporto una semplificazione dell'implementazione in Python dell'algoritmo. Questa implemetazione introduce, rispetto al precedente algoritmo in pseudocodice, l'aggiornamento dei *bias* e la gestione dei *mini-batch*.

```
epoch = 1
while (epoch <= max_epochs):
    for batch_n in range(int(len(trainset) / batch_sz)):
        start = batch_n * batch_sz
        data = trainset[start : start + batch_sz].T

        # positive phase:
        pos_hid_probs = (gpu.dot(net.W, data) +
            net.b.tile(batch_sz)).logistic()
        hid_states = pos_hid_probs > pos_hid_probs.rand()
        pos_corr = gpu.dot(pos_hid_probs, data.T)
        pos_vis_act = data.sum(axis = 1)
        pos_hid_act = pos_hid_probs.sum(axis = 1)

        # build the training set for the next RBM:
        if epoch == max_epochs:
            next_rbm_data[start : start + batch_sz] =
                pos_hid_probs.T

        # negative phase:
        vis_probs = (gpu.dot(net.W.T, hid_states) +
            net.a.tile(batch_sz)).logistic()
```

```
reconstr = vis_probs > vis_probs.rand()
neg_hid_probs = (gpu.dot(net.W, reconstr) +
    net.b.tile(batch_sz)).logistic()
neg_corr = gpu.dot(neg_hid_probs, reconstr.T)
neg_vis_act = reconstr.sum(axis = 1)
neg_hid_act = neg_hid_probs.sum(axis = 1)

# updates:
W_update = momentum * W_update +
    learn_rate * ((pos_corr - neg_corr) /
    batch_sz - (w_decay * net.W))
a_update = (momentum * a_update) +
    (pos_vis_act - neg_vis_act).reshape(-1, 1) *
    (learn_rate / batch_sz)
b_update = (momentum * b_update) +
    (pos_hid_act - neg_hid_act).reshape(-1, 1) *
    (learn_rate / batch_sz)
net.W += W_update
net.a += a_update
net.b += b_update

# encourage sparse hidden activities:
if spars_target > 0 and spars_target < 1:
    q = pos_hid_act.reshape(-1, 1) / batch_sz
    if q.mean() > spars_target:
        b_update -= learn_rate * (q - spars_target)
```

BIBLIOGRAFIA

[1] Paul Feyerabend. «How to defend society against science». In: *Introductory readings in the philosophy of science* (1998), pp. 54–65.

[2] Django Software Foundation. *Django. the web framework for perfectionists with deadlines*. URL: https://www.djangoproject.com (visitato il 05/07/2017).

[3] jQuery Foundation. *jQuery: write less, do more.* URL: http://jquery.com (visitato il 09/07/2017).

[4] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.

[5] Max Franz et al. «Cytoscape.js: a graph theory library for visualisation and analysis». In: *Bioinformatics* 32.2 (2016), p. 309. URL: http://dx.doi.org/10.1093/bioinformatics/btv557 (visitato il 09/07/2017).

[6] Highsoft. *Highcharts*. URL: https://www.highcharts.com (visitato il 09/07/2017).

[7] Geoffrey Hinton. *A practical guide to training restricted Boltzmann machines*. technical report. University of Toronto, 2010.

[8] Will Knight. «The Dark Secret at the Heart of AI». In: *MIT Technology Review* (2017).

[9] Zachary C. Lipton. «The mythos of model interpretability». In: *arXiv preprint arXiv:1606.03490* (2016).

[10] Volodymyr Mnih. *CUDAMat: A CUDA-based matrix class for Python*. technical report. University of Toronto, 2009.

[11] Jacques Monod. *Le hasard et la nécessité. Essai sur la philosophie naturelle de la biologie moderne.* Éditions Points, 2014.

[12] Judea Pearl. «The art and science of cause and effect». In: *Causality: models, reasoning and inference* (2000), pp. 331–358.

[13] Alberto Testolin et al. «Deep unsupervised learning on a desktop PC: a primer for cognitive scientists». In: *Frontiers in psychology* 4 (2013).

[14] Tijmen Tieleman. *Gnumpy: an easy way to use GPU boards in Python*. technical report. University of Toronto, 2010.

[15] Wikipedia. *Chaos model*. URL: https://en.wikipedia.org/wiki/Chaos_model (visitato il 07/07/2017).

[16] Marco Zorzi, Alberto Testolin e Ivilin P. Stoianov. «Modeling language and cognition with deep unsupervised learning: a tutorial overview». In: *Frontiers in psychology* 4 (2013).