Metodi Computazionali per la Fisica - Secondo Modulo: C++

Gianluca Merlo

11 febbraio 2008

Sommario

Il secondo modulo (C++) del corso in *Metodi Computazionali per la Fisica* ha avuto come argomento principale un'introduzione alla programmazione ad oggetti attraverso l'implementazione di codice per la *gestione dei dati*, per la parametrizzazione di questi ultimi con *reti neurali* e per la *minimizzazione* degli errori nel processo di parametrizzazione. Parti minori del corso sono inoltre state dedicate all'utilizzo di sistemi per lo sviluppo collaborativo ed il controllo di versione come *Subversion* e strumenti per la scrittura di documentazione quale *Doxygen*.
Gli argomenti sopra elencati sono quindi stati utili per lo sviluppo di un progetto, che verrà descritto in questo documento.

1 Introduzione

Per lo sviluppo del progetto, si è deciso di orientarsi sull'utilizzo delle reti neurali per la parametrizzazione di dati, scegliendo di applicarle all'*OCR*, ovvero al *Riconoscimento Ottico di Caratteri*. L'argomento è infatti di notevole interesse, in quanto ad oggi non esistono metodi completamente affidabili per consentire ad un essere umano di comunicare con una macchina in una maniera intuitiva e diffusa come la scrittura. Il problema è imputabile essenzialmente alla mancanza di *elasticità* che un calcolatore per sua stessa definizione possiede, ossia nella sua incapacità nel poter immagazzinare e interpretare dati se non gli è noto un preciso schema di formattazione degli stessi. La scrittura risulta un campo interessante di applicazione proprio perchè la forma che i medesimi caratteri assumono è fortemente variabile in base alla calligrafia, pur rimanendo essa stessa sufficientemente ben definita.

Una soluzione apparentemente ovvia al problema sarebbe fornire al calcolatore questa capacità di adattamento e generalizzazione di cui difetta, e un metodo per farlo si presenta se si immagina di sfruttare le reti neurali. Queste sono potenti mezzi di parametrizzazione che si sono dimostrati in grado di forti capacità di generalizzazione e predizione, e vengono comunemente impiegate come valida soluzione a problemi di *interpolazione di dati* o persino alla *predizione di eventi pseudocasuali*. Una rete neurale ben costruita potrebbe, in linea di massima, arrivare a costruire una mappa sufficientemente raffinata che colleghi la distribuzione dei colori in un'immagine ad una lettera dell'alfabeto, la quale potrebbe anche tenere conto di piccole variazioni quali la calligrafia o lo stile di scrittura, superando dunque il riconoscimento dell'immagine punto per

punto in favore di modelli più generali.
Una struttura simile consentirebbe peraltro di ridurre il processo di adattamento che un essere umano deve affrontare per comunicare con una macchina, poichè se una rete neurale venisse utilizzata per il riconoscimento di un set di caratteri sufficientemente regolari scritti da una persona, potrebbe a tutti gli effetti adattarsi all'utente senza che quest'ultimo debba compiere particolari sforzi.

Pertanto, alla luce di quanto detto, risulta chiaro che per lo sviluppo del progetto vi sono tre punti chiave:

1. Acquisizione e organizzazione dei dati.
2. Costruzione di una rete neurale adatta ai dati da trattare.
3. Allenamento della rete neurale su insiemi di dati.

Seguendo queste tre essenziali fasi necessarie per il progetto, si passerà quindi ad illustrare le informazioni generali alla base del problema, con cenni agli schemi di implementazione, per poi presentare alcuni dei risultati ottenuti e trarre le debite conclusioni.
Per quanto riguarda la documentazione tecnica inerente al codice, si rimanda a quella allegata, che è stata redatta secondo gli standard avvalendosi dell'aiuto di *Doxygen*, e nella quale sono presenti tutti i dettagli e le descrizioni opportune.

2 Illustrazione del Progetto

In questa sezione verranno descritte le componenti principali del progetto da un punto di vista generico ma sufficientemente dettagliato, per dare la possibilità al lettore di comprendere meglio la documentazione del codice e il codice stesso anche nel caso gli argomenti citati gli fossero estranei. Gli strumenti utilizzati verranno ovviamente introdotti limitatamente ai casi particolari riguardanti loro compito e uso all'interno del progetto, senza ambire a descrizioni formali complete per le quali si consiglia vivamente di riferirsi a testi specifici.

2.1 Gestione dei Dati

2.1.1 Formato dei dati

L'obiettivo del progetto è quello di giungere ad un riconoscimento sufficientemente preciso di *singoli caratteri*, e pertanto è innanzitutto necessario definire con precisione i tipi di dati su cui esso potrà operare.

Un *singolo carattere* si può definire banalmente come un'immagine di dimensione prefissata. La forma più semplice per questo tipo di immagine rispecchia quello dei vecchi *caratteri raster* usati nelle più semplici e vecchie applicazioni, ma comuni ancora oggi (ad esempio nel server *X.Org* e nei terminali). Tali immagini sono banalmente *bitmap*, in cui per ogni *pixel* viene registrato un valore numerico corrispondente al colore. Un esempio di formato in grado di gestire questo tipo di immagini è il formato *PBM* (*Portable Bit Map*), o più precisamente il formato *PGM* (*Portable Gray Map*) da esso derivato.
Un'immagine PBM è descritta dal suo header, che è composto da:

- Un *magic number* che identifica il tipo di PBM. Per quanto riguarda la codifica *ASCII*, *P1*, *P2* e *P3* corrispondono rispettivamente a immagini monocrotmatiche, in scala di grigi a 8 bit e a colori a 24 bit. Altri magic number sono invece riferiti a immagini codificate *raw*, che non consideriamo perchè sarebbero più complesse da leggere.
- La dimensione dell'immagine, in larghezza e altezza
- Il massimo valore di colore, che dovrebbe essere posto predefinitamente a 1 per immagini monocromatiche e a 255 per immagini in scala di grigi o colore.

A questi campi segue quindi l'elenco dei valori dei pixel separati da un qualsiasi caratter *whitespace*, elencati sequenzialmente percorrendo l'immagine da sinistra verso destra e dall'alto verso il basso. Nel caso delle immagini a colori, ogni pixel viene identificato in *RGB* da tre valori numerici.

Il tipo di immagine PBM che è stato scelto è la *PGM P2*, la quale corrisponde ad una mappa in scala di grigi, di profondità 8 bit e codificata in ASCII. La codifica è fondamentale perchè consentirà al programma di leggere con estrema semplicità i dati in essa contenuti, mentre la scelta della scala di grigi rispetto alla pura monocromaticità è dovuta al fatto che si è voluto dare al codice la possibilità di essere espandibile oltre quanto svolto durante il progetto[1].

Regular ASCII Chart (character codes 0 - 127)

000		(nul)	016	►	(dle)	032	sp	048	0	064	@	080	P	096	`	112	p
001	☺	(soh)	017	◄	(dc1)	033	!	049	1	065	A	081	Q	097	a	113	q
002	☻	(stx)	018	↕	(dc2)	034	"	050	2	066	B	082	R	098	b	114	r
003	♥	(etx)	019	‼	(dc3)	035	#	051	3	067	C	083	S	099	c	115	s
004	♦	(eot)	020	¶	(dc4)	036	$	052	4	068	D	084	T	100	d	116	t
005	♣	(enq)	021	§	(nak)	037	%	053	5	069	E	085	U	101	e	117	u
006	♠	(ack)	022	▬	(syn)	038	&	054	6	070	F	086	V	102	f	118	v
007	•	(bel)	023	↨	(etb)	039	'	055	7	071	G	087	W	103	g	119	w
008	◘	(bs)	024	↑	(can)	040	(	056	8	072	H	088	X	104	h	120	x
009	○	(tab)	025	↓	(em)	041	)	057	9	073	I	089	Y	105	i	121	y
010		(lf)	026	→	(eof)	042	*	058	:	074	J	090	Z	106	j	122	z
011	♂	(vt)	027	←	(esc)	043	+	059	;	075	K	091	[	107	k	123	{
012	♀	(np)	028	∟	(fs)	044	,	060	<	076	L	092	\	108	l	124	¦
013		(cr)	029	↔	(gs)	045	-	061	=	077	M	093	]	109	m	125	}
014	♫	(so)	030	▲	(rs)	046	.	062	>	078	N	094	^	110	n	126	~
015	☼	(si)	031	▼	(us)	047	/	063	?	079	O	095	_	111	o	127	⌂

Extended ASCII Chart (character codes 128 - 255)

128	Ç	143	Å	158	×	172	¼	186	‖	200	╚	214	í	228	õ	242	≥
129	ü	144	É	159	ƒ	173	¡	187	╗	201	╔	215	Î	229	Õ	243	¾
130	é	145	æ	160	á	174	«	188	╝	202	╩	216	Ï	230	µ	244	¶
131	â	146	Æ	161	í	175	»	189	¢	203	╦	217	┘	231	Þ	245	§
132	ä	147	ô	162	ó	176	░	190	¥	204	╠	218	┌	232	Þ	246	÷
133	à	148	ö	163	ú	177	▒	191	┐	205	═	219	█	233	Ú	247	¸
134	å	149	ò	164	ñ	178	▓	192	└	206	╬	220	▄	234	Û	248	°
135	ç	150	û	165	Ñ	179	│	193	┴	207	¤	221	¦	235	ù	249	¨
136	ê	151	ù	166	ª	180	┤	194	┬	208	ð	222	Ì	236	ý	250	·
137	ë	152	ÿ	167	º	181	Á	195	├	209	Ð	223	▀	237	Ý	251	¹
138	è	153	Ö	168	¿	182	Â	196	─	210	Ê	224	ó	238	¯	252	³
139	ï	154	Ü	169	®	183	À	197	┼	211	Ë	225	ß	239	´	253	²
140	î	155	ø	170	¬	184	©	198	ã	212	È	226	Ô	240	-	254	■
141	ì	156	£	171	½	185	╣	199	Ã	213	ı	227	Ò	241	±	255	
142	Ä	157	Ø														

Figura 1: Mappa dei caratteri *ASCII* e *ASCII estesa*.

[1] Tecnicamente il progetto è già in grado di operare su immagini in scala di grigi, ma non sono state fornite mappe nel pacchetto ed il convertitore da me scritto opera esclusivamente da *GIF* monocromatiche a *PGM P2*. (NdA)

Le immagini utilizzate dovranno inoltre essere riconducibili al carattere che rappresentano. La scelta che è stata fatta in questo caso è quella di porre nel nome del file *l'ordinale associato al carattere nella mappa ASCII*, per il quale rimandiamo alla figura 1.

2.1.2 Organizzatione dei Dati

I dati presenti nelle immagini utilizzate devono necessariamente essere inseriti in un contenitore comprensibile alle applicazioni, e questo ruolo è demandato alla classe *OCRData*. Tale classe descrive tre tipi di contenitori per questi dati:

- Singolo carattere (*Character*).
- Collezioni di caratteri (*CharMap*).
- Immagini di caratteri (*CharImage*).

I punti in comune tra tutti i dati risiedono in:

- Immagazzinamento delle informazioni dell'header dei dati di origine.
- Capacità di gestire *input* e *output* multipli, consentendo la gestione di funzioni $\mathbf{N}^n \rightarrow \mathbf{N}^m$.[2]
- Metodi di lettura, scrittura e manipolazione.

Le differenze invece, riguardano l'organizzazione dei dati:

- Per i singoli caratteri, un'unico punto con tanti input quanti i pixel nell'immagine, e con l'ordinale associato al carattere come unico output.
- Per le collezioni di caratteri, più punti costruiti analogamente a quelli di un singolo carattere.
- Per le immagini di caratteri, più punti, in numero pari al numero dei pixel nell'immagine, i cui input sono le due coordinate e il cui output è il valore di colore per l'input relativo. L'ordinale corrispondente al carattere è memorizzato a parte.

2.2 Reti Neurali

2.2.1 Cenni Generali

Le *reti neurali artificiali*, come già anticipato, sono utilizzate nel progetto come mezzo di parametrizzazione dei dati a causa della loro buona adattabilità e della relativa semplicità di implementazione.

Una rete neurale è composta, in analogia con il corrispettivo biologico, da una serie di elementi fondamentali detti *nodi* o *neuroni*, organizzati in uno schema a più strati (*layers*). Vi sono ovviamente molti tipi di reti neurali, differenti per struttura a seconda dei campi in cui vengono utilizzate. Non è l'obiettivo di questo testo trattare in generale l'argomento, e pertanto ne presentiamo un modello base, che consiste in tre o più strati:

[2] Il campo di impiego è limitato a $\mathbf{N}$ poichè sarebbe inutile lasciarlo esteso alla doppia precisione visti i valori tipici contenuti in un'immagine.

- Lo *strato di input* (*input layer*), nei cui nodi vengono introdotti i dati in ingresso.
- Lo *strato nascosto* (*hidden layer*), costituito da più strati di nodi interposti tra l'input e l'output, i quali hanno la funzione di processare i dati.
- Lo *strato di output* (*output layer*), i cui nodi contengono i valori finali di uscita.

Questo tipo di rete neurale è molto semplice, poichè solo gli strati di input e di output sono connessi con l'esterno, e soltanto i nodi di strati adiacenti sono collegati tra loro. I dati viaggiano in un'unica direzione, dallo strato di input allo strato di output, e per questo tale schema viene definito *rete neurale feed-forward*. Come appare dunque sufficientemente ovvio, ciò che caratterizza una

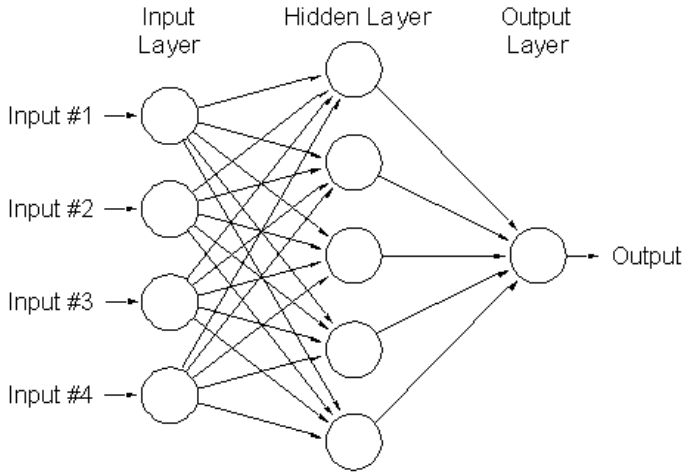

Figura 2: Generico schema di *rete neurale feed-forward*

rete neurale oltre al numero e alla disposizione dei suoi strati e dei suoi nodi è la quantità e il tipo di interconnessioni tra i questi ultimi, le quali vengono a determinarne la risposta e persino, con se si considerano strutture più avanzate di quella descritta, le capacità di adattabilità e di apprendimento. Questi concetti chiave alla base del successo delle reti neurali sono legati al fatto che le interconnessioni tra i nodi, e il modo con cui i dati vengono elaborati di livello in livello, possono essere opportunamente modificate attraverso algoritmi detti appunto di *apprendimento*, in cui si cerca in genere di minimizzare una determinata funzione di errore calcolata sulla base delle risposte della rete neurale e di dati noti attraverso metodi esterni o implementazioni interne allo schema della rete neurale.

2.2.2 *Perceptron* e *Multi-Layer Perceptron*

Vista l'importanza delle interconnessioni tra i nodi, è necessario scendere nel particolare per comprendere meglio il funzionamento di una rete neurale.

L'esempio più comune e semplice è quello del *perceptrone* (*perceptron*), come unità fondamentale per il funzionamento di una rete neurale. Il concetto di

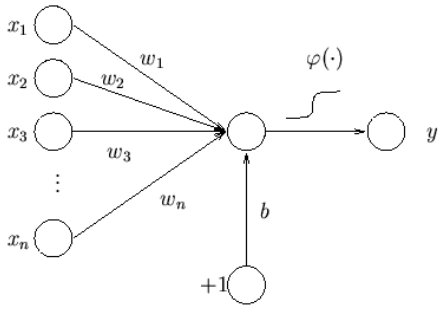

Figura 3: Schema di *perceptrone*

perceptrone fu formulato nel 1957 come unità base di classificazione e calcolo, e consiste nella definizione di un singolo nodo, il quale è connesso direttamente ad un certo numero di input, ed è in grado di restituire un output. Tale valore viene determinato attraverso la somma pesata degli input, aggiungendo un valore di soglia e valutando il tutto attraverso una opportuna *funzione di attivazione* φ. Con riferimento alla figura 3:

$$y = \varphi\left(\sum_i \omega_i x_i + b\right) \tag{1}$$

I dati in questo tipo di struttura viaggiano in una sola direzione, dall'input all'output, e pertanto esso rientra nella categoria della reti neurali feed-forward. Un perceptrone non è autonomamente in grado di compiere alcun tipo di apprendimento, e ciò va quindi demandato a un algoritmo di minimizzatione esterno che sia in grado di variare i pesi ω_i di modo che l'errore tra l'output y e dei valori noti venga ridotto. Il set di pesi così ottenuto può quindi essere provato su dati ignoti.
La funzione di attivazione φ può essere scelta dall'utente a seconda delle esigenze, ma è di norma appartenente a determinate classi di funzioni, quali la *sigmoide logistica*,le *radial basis functions* o persino (limitatamente a situazioni ben precise) una semplice funzione *lineare*. Di queste funzioni, la più comune è la sigmoide, che è definita come:

$$\varphi(x) = \frac{1}{1 + e^{-x}} \tag{2}$$

Risulta sufficientemente chiaro, osservando la funzione di attivazione in figura 4, il significato del termine b. Esso prende il nome da *bias* (*rumore*) e assume il ruolo di *soglia* per la funzione di attivazione, motivo per il quale viene comunemente indicato anche come θ (*threshold*). Nella traduzione in algoritmo del concetto di perceptrone il termine di soglia viene spesso trattato aggiungendo *un singolo input di valore unitario*, il che eguaglia il contributo di tale termine a quello del peso associato.

L'evoluzione naturale del concetto di perceptrone è un sistema costituito da più perceptroni, e lo schema più semplice in cui essi possono essere disposti è quello già visto per le reti neurali feed-forward in figura 2. Essendo questa una

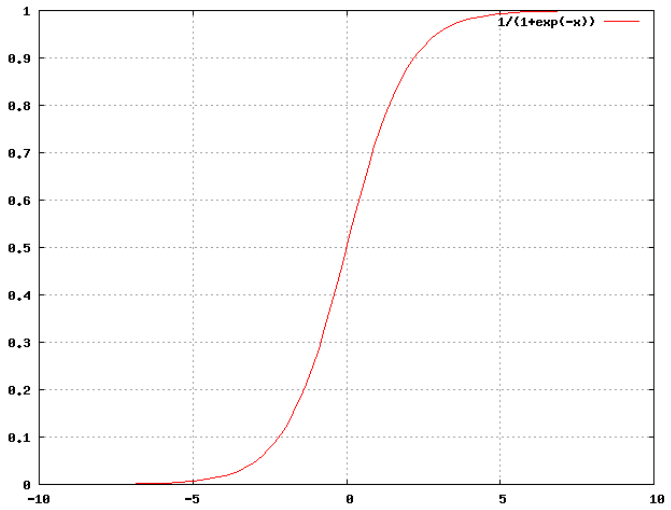

Figura 4: Sigmoide Logistica

disposizione di perceptroni su più strati, viene comunemente definita *multi-layer perceptron* (*MLP*).
In un MLP, ogni nodo è un perceptrone che ha per input tutti i nodi dello strato precedente. Pertanto, seguendo la notazione:

- x_i^l: L'i-esimo perceptrone dello strato l.
- ω_{ij}^l: Il peso di connessione dall'i-esimo perceptrone dello strato l al j-esimo dello strato $l+1$.
- θ_i^l: La soglia dell'i-esimo perceptrone dello strato l.
- N_l: Il numero di perceptroni sullo strato l.

Si ha:

$$x_i^l = \varphi\left(\sum_{j=0}^{N_{l-1}-1} \omega_{ji}^{l-1} x_j^{l-1} + \theta_j^l\right) \tag{3}$$

Nel caso venga utilizzato un valore di soglia pari al peso associato alla connessione, non è complesso convincersi che è conveniente, specie per quanto riguarda l'implementazione e l'algoritmo relativo, introdurre nello schema un ulteriore neurone di valore unitario per ogni strato. Pertanto, ponendo $x_{N_l}^l = 1$, la 3 diventa

$$x_i^l = \varphi\left(\sum_{j=0}^{N_{l-1}} \omega_{ji}^{l-1} x_j^{l-1}\right) \tag{4}$$

Un MLP possiede quindi un numero di pesi pari al numero di interconnessioni tra nodi, ossia

$$N_w = \sum_{l=0}^{L-1} (N_l + 1)\, N_{l+1} \tag{5}$$

nel caso di un MLP con L strati.
Questo numero di pesi influisce direttamente sulle capacità di parametrizzazzione di una rete neurale ed è quindi una variabile da tenere in considerazione nella costruzione di una simulazione. Se ad esempio l'immagine che si desidera riconoscere fosse 10x10 e la mappa di caratteri fosse da 26 caratteri, una rete neurale con 2600 parametri fornirebbe una rappresentazione 1:1 della collezione di caratteri e dunque eccederebbe in *rappresentazione*, mentre al contrario se venisse scelto un numero troppo limitato di pesi si potrebbe eccedere in *generalizzazione* e perdere dunque le capacità di riconoscimento.

2.2.3 Implementazione e Collegamento con i Dati

Avendo introdotto il tipo di rete neurale che verrà utilizzata, è ora possibile chiarire il suo collegamento con i dati dal paragrafo 2.1.

La rete neurale creata dovrà essere adatta al tipo di dati di analizzare, poichè come si è visto un MLP è connesso con l'esterno attravero i suoi layer di input e output. La rete neurale dovrà dunque essere di almeno 3 strati, e gli strati di input e output dovranno essere dimensionati in modo da rispecchiare quelli del tipo di dato in esame. La scelta dell'aggiunta di ulteriori strati nascosti e del numero di nodi in questi ultimi modificherà il numero di pesi associati e dunque il comportamento della rete. Le caratteristiche della rete possono essere modificate attraverso un file di configurazione.
Essendo un MLP incapace di effettuare un apprendimento senza un azione esterna, una rete neurale sarà dotata di metodi per caricare e salvare e consentire all'utente di accedere ai propri pesi. Il calcolo vero e proprio sarà effettuato da un'opportuno metodo in grado di popolare il layer di input ed eseguire l'algoritmo descritto nel paragrafo precedente, per restituire quindi gli output. Viene lasciata libertà di scelta e di espansione per quanto riguarda la funzione di attivazione, che è quindi implementata a parte e facilmente modificabile per l'introduzione di altri tipi di parametri.

La struttura scelta per l'implementazione del MLP consiste in una classe generica *NeuralNetwork* contenente gli elementi generali comuni a reti neurali organizzate secondo le descrizioni fatte, implementando a parte la classe *MLP* in cui vengono specializzati gli algoritmi di calcolo. In queso modo dovrebbe essere possibile descrivere tipi differenti di rete neurale appoggiandosi per ereditarietà alla classe base e implementando solo il codice necessario, nel caso si dovesse procedere ad espandere il codice.

2.3 Minimizzazione

2.3.1 Cenni Generali

Lo schema del MLP, come visto, necessita di un algoritmo esterno che provveda a fornire dati in ingresso alla rete neurale, recuperarne i dati in uscita e compiere modificazioni allo schema dei pesi al fine di provvedere al processo di apprendimento della rete stessa. Questo tipo di compito viene svolto generalmente da un *algoritmo di minimizzazione* applicato ad una debita *funzione di errore* che sia in grado di fornire un indicatore della qualità dei dati in uscita dalla rete neurale rispetto a risultati noti.
Un algoritmo di minimizzazione pertanto è banalmente un modo per esplorare

lo spazio dei parametri della rete neurale alla ricerca di risultati verosimili, attraverso il continuo confronto con dati di riferimento. Tale confronto non viene eseguito unicamente alla ricerca della massima precisione di rappresenzatione dei dati forniti, ma utilizza in genere uno schema *training/validation* al fine di garantire che tale tipo di minimizzazione salvaguardi la generalità della parametrizzazione fornita. I dati in questo modo vengono in genere divisi in due insiemi:

- *Insieme di training* (*training set*), sul quale viene eseguita l'effettiva minimizzazione, alla ricerca della migliore rappresentazione e dunque di errori più bassi.
- *Insieme di validation* (*validation set*), sul quale vengono provati gli effetti delle variazioni apportate dalla minimizzazione, per fare in modo che l'errore su dati esterni al set di training ma rappresentanti lo stesso fenomeno o situazione in esame non cresca eccessivamente.

Pertanto una minimizzazione sufficientemente buona dovrebbe produrre errori bassi sul set di training senza che quelli sul set di validation crescano oltre determinati valori. In questo modo è possibile controllare dall'esterno la quantità di rappresentazione o generalizzazione che si desidera ottenere.

I metodi specifici per ottenere questo tipo di risultato sono molti, poichè vi sono molti tipi di funzioni di errore definibili e molti metodi per esplorare lo spazio dei parametri con maggiore o minore cognizione di causa alla ricerca dei minimi di tale funzione, tenendo spesso conto di problemi di resa e di tempo di finalizzazione, piuttosto che di onerosità dei calcoli relativi. Senza pretese, dunque, verrà illustrato un metodo semplice e relativamente economico per implementare un algoritmo di questo tipo, con alcuni spunti per ottenere varianti più raffinate.

2.3.2 L'Algoritmo Genetico

Il metodo utilizzato in questo progetto è il cosiddetto *algoritmo genetico*. Tale metodo deriva il suo nome esattamente dall'analogo processo di selezione naturale delle specie esistente in natura. Lo spazio dei parametri, in altre parole, viene esplorato in maniera casuale variando un insieme limitato di questi ultimi, al fine di causare nella popolazione la comparsa di individui di errore minore di quelli presenti in partenza.

L'algoritmo genetico può essere riassunto in sostanza nelle seguenti fasi:

1. Generazione di N_{clones} cloni a partire dai migliori individui a disposizione.
2. Generazione di $N_{mutants}$ mutanti per ogni clone esistente.
3. Applicazione delle mutazioni agli individui.
4. Valutazione della funzione di errore per ogni individuo.
5. Generazione di una lista ordinata secondo il valore d'errore degli individui.
6. Selezione degli N_{clones} individui con l'errore più basso, i quali passeranno alla generazione successiva.

Tali fasi vengono ripetute quindi fino al raggiungimento di determinate condizioni. Generalmente l'algoritmo viene inizialmente impostato di modo da poter eseguire un numero massimo di iterazioni, con l'accorgimento di arrestarlo nel caso l'errore del set di validation inizi a crescere anzichè diminuire. Una peculiarità notevole di questo tipo di procedimento, che dovrebbe essere chiara dopo l'ultima osservazione, è che invece *l'errore calcolato sul set di training può unicamente descrescere o al più restare costante.*

Scendendo più nel particolare, l'algoritmo di base utilizzato nel progetto può essere configurato con i seguenti parametri:

- Numero massimo di iterazioni.
- Numero di cloni.
- Numero di mutanti per ogni clone.
- Errore massimo, al di sopra del quale l'algoritmo non può essere arrestato.
- Numero di mutazioni da eseguire su ogni mutante.
- Entità delle mutazioni.
- Numero di iterazioni per lo *smearing*.
- Numero di iterazioni tra i *dump*.

Il significato dei primi tre parametri è già stato trattato. L'errore massimo è un valore della funzione di errore che corrisponde al minimo errore che si desidera ottenere dal'algoritmo: se l'errore del set di training resta sopra tale valore l'algoritmo non viene arrestato nemmeno se l'errore sul set di validation cresce.
I seguenti due parametri ci consentono di controllare come le mutazioni vengono effettuate. Si è scelto di mutare un certo numero di pesi della rete neurale, i quali vengono scelti in maniera casuale, e al pari a tali pesi sarà aggiunto o tolto un valore casuale e proporzionale al parametro di entità delle mutazioni.
Per *smearing* si intende un accorgimento utilizzato per prevenire le fluttuazioni dell'errore che potrebbero aver luogo data la natura dell'algoritmo. In altre parole, durante la valutazione della condizione di arresto, all'atto del valutare l'andamento crescente o decrescente delle funzioni di errore, non vengono utilizzati gli ultimi valori calcolati, ma la media di un certo numero di questi ultimi proprio per ridurre la possibilità che fluttuazioni dell'errore di validation portino all'arresto prematuro dell'algoritmo.
L'ultimo parametro indica invece unicamente l'intervallo di iterazioni che deve intercorrere tra i dump, ovvero la scrittura di informazioni relative all'algoritmo su terminale e il salvataggio dello stato di apprendimento su disco.

Questo tipo di implementazione statica dell'algoritmo genetico può tuttavia non risultare del tutto soddisfacente, specie nella ricerca di errori particolarmente bassi. Ciò che in genere accade è che ad errori alti statisticamente ci siano un numero non trascurabile di mutanti che muovendosi nello spazio dei parametri entro il raggio dato riescano ad avvicinarsi ulteriormente al minimo della funzione di errore. Ciò tuttavia perde drasticamente di validità con il decrescere dell'errore: accade che i migliori cloni siano già prossimi al minimo ricercato, e che i mutanti generati vengano distibuiti in un raggio troppo ampio per generare in tempi accettabili risultati migliori. Si assiste così a fenomeni di

saturazione del'errore, il quale rimane costante per lunghi gruppi di iterazioni, e pertanto i tempi necessari a generare risultati validi diventano molto più elevati. Una prima soluzione a questo tipo di problema è scegliere un numero di individui sufficientemente alto e impostare un numero limitato di mutazioni di entità ridotta, anche se non è difficile convincersi del fatto che questo tipo di soluzione rimanda solamente il verificarsi della condizione di saturazione, oltre che a rallentare lievemente la fase iniziale.

Soluzioni più efficaci sono invece quelle connesse alla modifica dell'algoritmo al fine di renderlo dinamicamente in grado di variare i suoi parametri in presenza di determinate condizioni. Queste sono:

- Ridurre il numero di mutazioni in presenza di saturazione.
- Ridurre l'entità delle mutazioni in presenza di saturazione.
- Mutare la popolazione con diversi numeri di mutazioni ed entità, secondo distribuzioni prefissate.
- Aumentare il numero di mutanti in presenza di saturazione.

Come si vede sono tutti accorgimenti sufficientemente semplici e di non difficile implementazione, che consentono di rendere l'algoritmo genetico molto più rapido e funzionale.

2.3.3 Implementazione

Nell'implementazione di metodi di minimizzazione si è cercato di seguire lo stesso schema già visto per le altri classi. Una classe *Minimization* generica è stata predisposta per l'ampliamento attraverso ereditarietà, mentre l'algoritmo genetico è stato implementato nei suoi aspetti particolari nella classe *GeneticAlgorithm*.

Lo schema di implementazione seguito è stato inizialmente quello classico mostrato nel paragrafo precedente. L'algoritmo è costruito per essere utilizzato unicamente con dei *CharMap* come dati ingresso, ma supporta qualsiasi tipo di rete neurale ben definita secondo gli standard del codice poichè al suo interno di avvale della logica a puntatori a *NeuralNetwork*. La rete fornita, attraverso i suoi metodi di output, lettura e scrittura dei suoi pesi è continuamente riconfigurata durante il processo di allenamento con i dati degli individui, mentre valuta i dati passatigli con il set di training e quello di validation. I parametri dell'algoritmo di minimizzazione sono modificabili attraverso un file di configurazione.
Questo schema è stato tuttavia variato nelle fasi finali di sviluppo a causa dei lunghi tempi (spesso superiori alle 48 ore) necessari per l'apprendimento ad errori bassi per fenomeni di saturazione, e sono stati introdotti i primi due accorgimenti citati nel paragrafo precedente. In particolare, il numero di mutazioni e la loro entità vengono variati osservando quale dei due ha il rapporto più alto con il numero totale dei parametri e i valori massimi dei pesi nella rete neurale, consentendo riduzioni dei tempi di calcolo anche superiori al 100%.

Le classi presentano, oltre a tutti i metodi necessari per compiere singole iterazioni dell'algoritmo e ripeterle secondo le condizioni descritte, metodi che agiscono sulla rete neurale nella fasi iniziale, finale e di dump intermedio di

modo da salvare i risultati ottenuti per poterli analizzare in tempo reale o riprendere l'apprendimento in una fase successiva.
Per quanto riguarda la funzione di errore, è stato scelto il metodo del χ^2 ponendo $\sigma = 0.4$. Tale scelta è stata fatta poichè la rete neurale produce valori decimali in doppia precisione, mentre i caratteri sono associati ad un ordinale ASCII da 0 a 255, secondo la mappa in figura 1. Porre dunque tale valore per σ equivale a fissare per i valori di output della rete neurale la possibilità di raggiungere l'intero più vicino. Si potrebbe obiettare che semplicemente sottraendo i valori ordinali dei caratteri nella mappa ASCII non si ha una buona stima dell'errore[3] e che quindi una migliore definizione degli errori sarebbe semplicemente il numero di caratteri sbagliati nel set. Ciò è in linea di massima vero, anche se essendo l'algoritmo genetico sostanzialmente casuale ed essendo il nostro obiettivo la minimizzazione della funzione di errore, questa definizione è del tutto equivalente[4].

3 Simulazioni e Risultati

3.1 Descrizione delle Simulazioni e Presentazione dei Risultati

In questo paragrafo presentiamo i risultati di alcune simulazioni svolte utilizzando le applicazioni scritte per il progetto. Tutti i test sono stati fatti utilizzando l'applicazione *ocrtrain* per il training delle reti neurali. Si è scelto di concentrarsi su un numero ridotto di caratteri, limitando l'analisi alle 26 lettere minuscole dell'alfabeto, ossia ai caratteri dallo 097 al 122 della mappa ASCII, e avvalendosi dei font Arial, Book Antiqua, Comic, Garamond, Century Gothic, Monotype Corsiva e Times, mostrati nella figura di seguito.

Figura 5: Font utilizzati nel progetto. Nell'ordine, dall'alto verso al basso: Arial, Book Antiqua, Comic, Garamond, Century Gothic, Monotype Corsiva e Times.

Tutti i font sono costituiti da 26 immagini *GIF* monocromatiche di dimensione 11x16, convertite a *PGM P2* attraverso un convertitore interno al progetto

[3] Ad esempio, il carattere *8* (056) e la *B* (066) distano 10 posizioni pur essendo molto simili.
[4] Facciamo notare che se si fosse usato un algoritmo come *gradient descent*, che sfrutta le variazioni della funzione di errore, con tutta probabibilità si sarebbero ottenuti pessimi risultati e sarebbe stata necessaria una ridefinizione della stessa.

che sfrutta *Python* e le *Python Imaging Libraries* (*PIL*). Tali caratteri sono forniti ai set di training e validation come blocchi di uno o più font attraverso un opportuno file di configurazione.

Si può ora elencare i test eseguiti dividendoli a seconda di alcune loro caratteristiche salienti. Tali test sono mostrati nella tabella 1, riportante un numero identificativo, i font utilizzati per il training[5], l'errore raggiunto, il numero di pesi della rete neurale e il tempo macchina impiegato sotto *Condor* per il training[6].

ID	**Fonts**	χ^2	$\mathbf{N_w}$	**Runtime**
1	Arial, Century Gothic, Times	4.819	2671	01:02:02
2	Arial, Century Gothic, Times	0.988	2671	05:45:43
3	Arial, Century Gothic, Times	0.098	2671	05:18:16
4	Arial, Century Gothic, Times	4.995	891	03:17:18
5	Arial, Century Gothic, Times	0.97	891	01:31:17
6	Arial, Century Gothic, Times	0.1	891	25:18:16
7	Arial, Garamond, Century Gothic, Times	4.538	2671	02:19:47
8	Arial, Garamond, Century Gothic, Times	0.983	2671	09:10:25
9	Arial, Garamond, Century Gothic, Times	0.097	2671	12:24:41
10	Arial, Garamond, Century Gothic, Times	4.483	891	01:03:20
11	Arial, Garamond, Century Gothic, Times	0.997	891	19:25:02
12	Arial, Garamond, Century Gothic, Times	0.1	891	14:50:56

Tabella 1: Simulazioni effettuate.

Per quanto riguarda gli effettivi risultati del test, è molto complesso rappresentarli in una forma che possa essere contemporaneamente compatta ed esaustiva. Nella tabella 2 si possono trovare i numeri di caratteri complessivi riconosciuti relativamente ad ogni coppia carattere/test. Questo non è di certo un parametro ottimale di valutazione, ma fungerà da base per le considerazioni finali. Lasciamo al lettore la possibilità di verificare di persona i dettagli dei risultati, qualora fosse suo interesse, ricordando che nel pacchetto del progetto è presente la directory *examples* nella quale si possono trovare tutti i log, gli output e i dati per le reti neurali relativi ai test svolti nell'ultima fase di sviluppo, e che con l'interfaccia grafica *gocrtry* l'analisi dovrebbe risultare abbastanza rapida e semplice.

3.2 Considerazioni sui Risultati

I test effettuati, come si può notare osservando la tabella 1, sono divisi variando i seguenti parametri:

- Caratteri nel set di training.
- Numero di pesi nella rete neurale.

[5]Per il set di validation è sempre stato usato unicamente il font Comic.

[6]Purtroppo indicare i runtime registrati da Condor non è un metodo particolarmente affidabile di benchmarking del codice, e pertanto si inserisce tale dato unicamente per dare un'idea delle scale temporali in gioco. Per citare un esempio, il test 6 su un *Pentium III* a 750 MHz è stato completato in circa 15 ore.

Carattere/Test	**1**	**2**	**3**	**4**	**5**	**6**
Arial	17	23	26	22	22	26
Book Antiqua	2	2	3	0	1	1
Comic	3	1	1	3	2	3
Garamond	0	2	0	1	1	1
Century Gothic	13	22	25	21	22	25
Monotype Corsiva	0	2	1	0	0	1
Times	18	25	25	23	25	25
Carattere/Test	**7**	**8**	**9**	**10**	**11**	**12**
Arial	18	23	25	17	23	26
Book Antiqua	3	2	2	2	2	0
Comic	2	1	1	2	2	3
Garamond	18	25	26	20	25	26
Century Gothic	16	18	26	15	24	26
Monotype Corsiva	0	0	0	1	2	0
Times	18	24	26	21	22	24

Tabella 2: Caratteri riconosciuti dalle simulazioni.

- Valore finale del χ^2.

È possibile dunque valutare i risultati a seconda di queste variazioni per ottenere alcune informazioni su come il riconoscimento avviene.

Innanzitutto, si nota che il paradigma training/validation non è praticamente di nessuna utilità per il tipo di problema analizzato. Il font di validation Comic è costante per tutti i test e i suoi errori partono e restano alti[7]. Parrebbe dunque che in ogni caso questo tipo di parametrizzazione sia valido in *rappresentazione* ma non in *generalizzazione*, e ciò si nota anche considerando il fatto che tutti i caratteri non inclusi nei set di training hanno bassissimi numeri di caratteri riconosciuti, spesso in maniera apparentemente casuale. Questo risultato indica dunque che il tipo di analisi che la rete neurale effettua, e la parametrizzazione che fornisce, sono in questo progetto ben lontante dal concetto di *carattere*, al punto che i dati del font Comic sono considerati assolutamente estranei a quelli di tutti gli altri font come se rappresentassero qualcosa di completamente diverso. Seguendo questo ragionamento, l'unico modo per utilizzare nella pratica il progetto è quello di fornire un buon numero di font di training, possibilmente tutti estremamente simili, ponendosi come limite unicamente i tempi di calcolo[8] e l'inevitabile raggiungimento della saturazione della capacità di rappresentazione della rete neurale scelta.

Per quanto riguarda il valore finale di χ^2, non ci sono sorprese di sorta. Al decrescere di tale valore si nota uno spiccato aumento del numero di caratteri riconosciuti per i font di training, e in generale ai limiti estremi di rappresentazione, l'attesa degradazione dei risultati ottenuti per i font non contenuti nel set di training, indice della perdita di capacità di generalizzazione.

[7]Per *alti* si intende ben al di sopra dei limiti accettabili di χ^2, considerando che questi hanno l'ordine di grandezza delle centinaia, raggiungendo spesso anche il migliaio.

[8]Si noti che vi è un drastico incremento dei tempi necessari al calcolo passando da tre a quattro font, spesso vicino al raddoppio dello stesso.

Test	% Riconoscimenti
1	61.5
2	89.7
3	97.4
4	84.6
5	88.5
6	97.4
7	76.9
8	86.5
9	99
10	70.2
11	90.4
12	98.1

Tabella 3: Percentuali di successo complessive sul set di training.

Qualcosa di interessante è osservabile per quanto riguarda i risultati ottenuti al variare del numero di pesi nella rete neurale. Il tipo di dati che si desidera descrivere è una mappa di 26 caratteri, ciascuno dei quali di dimensioni 11x16. Ciò significa che ci si attende che la rete neurale offra una rappresentazione punto per punto qualora abbia 4576 parametri. Per questo motivo in una prima serie di test[9] si è usato un numero pari a circa la metà di questo valore, nella speranza che influisse positivamente sulla capacità di generalizzazione. In una serie successiva di test[10] tale numero è stato variato a meno di un quarto del valore citato in precedenza, per verificare se, per quanto non in grado di generalizzare al punto di riconoscere font ignoti, la rete potesse effettivamente avere un qualche tipo di incremento nella resa.
Tale tesi non può tuttavia essere confermata dai dati vista la loro ridotta quantità, anche se si nota che a degradarsi è il risultato sui caratteri effettivamente "differenti", come Monotype Corsiva e Book Antiqua, mentre Comic pare avere una lievissimo aumento di riconoscimenti.

Avendo già citato il problema della saturazione della capacità di rappresentazione in più occasioni, facciamo notare come i tempi di run per le simulazioni con minor numero di pesi siano nettamente più alti di quelle con numero di pesi più maggiore. Questo indica indubbiamente che reti neurali eccessivamente sottodimensionate, pur essendo interessanti se si tratta di verificare le loro capacità di generalizzazione, potrebbero non essere in grado di risolvere il problema del riconoscimento generalizzato proprio a causa dei tempi di apprendimento molto elevati a bassi errori. Ciò è prevedibile poichè le reti neurali in pratica approssimano funzioni ignote con composizioni lineari di funzioni più semplici, e il limite di precisione che possono raggiungere in tale approssimazione è ovviamente dipendente dalla dimensione di tali composizioni lineari, ossia al numero dei pesi che la rete possiede.

Per quanto riguarda invece i caratteri non riconosciuti, si possono fare altre interessanti considerazioni. Innanzitutto, per i font inclusi nel set di trai-

[9] Per i test 1,2,3,7,8,9.
[10] Per i test 4,5,6,10,11,12.

ning, generalmente i pochi errori sono molto poco distanti dai risultati corretti. Portiamo l'esempio di quanto accade per il font Arial nel test 5:

Carattere	**Valore Atteso**	**Valore**	**Scarto**
c	099	100.949	1.949
o	111	110.256	0.744
r	114	113.478	0.522
w	119	118.406	0.594

Tabella 4: Caratteri non riconosciuti per il font Arial nel test 5.

Come si vede, pur essendo le lettere errate, l'errore non è grande. La rete neurale viene allenata a disporre le lettere lungo la mappa ASCII a seconda della loro forma, e tutte le lettere non riconosciute citate sopra si discostano infatti di poco dal valore atteso, specie considerando che gli errori sono frutto dell'arrotondamento all'intero più vicino. Questo in realtà non è per nulla una buona approssimazione del risultato ottenuto, poichè come si è già più volte sottolineato la disposizione dei caratter nella mappa ASCII non tiene conto dell'effettiva somiglianza tra gli stessi.

3.3 Conclusioni

Considerando il carattere del progetto e le sue finalità, ossia quello dello sviluppo di un codice sufficientemente ben organizzato e documentato in C++ e l'applicazione delle tecniche viste a lezione ad un caso complesso, i risultati sono da considerare un buon successo. Come si è potuto notare dalla tabella 3, sul set di training le percentuali ottenute sono molto alte. Nel limite dei piccoli errori, esse superano sempre il 97%, valore che secondo la letteratura scientifica definisce una buon limite inferiore di resa affinchè il sistema possa risultare utilizzabile.

Per quanto riguarda invece i cattivi risultati in termini di generalizzazione, vi sono tuttavia alcuni aspetti che potrebbe valere la pena di analizzare, come la risposta del sistema per mappe più grandi e in scala di grigi, o la possibilità di integrare variazioni dell'immagine di input attraverso il riconoscimento per mezzo di un rete neurale o semplici set di trasformazioni geometriche[11]. Sotto questo punto di vista il progetto è largamente ampliabile e proprio grazie alle proprietà del C++ è predisposto per essere modificato e riutilizzato con sufficiente immediatezza.

[11] In analoghi progetti, le immagini vengono replicate applicando dei *crop*, traslazioni, rotazioni e stiramenti al fine di generalizzare in principio.