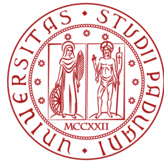

Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in
Ingegneria dell'Informazione

Reti Neurali Artificiali Singolo-Strato: Metodo della Discesa del Gradiente

Relatore:
Prof. Luca Schenato

Laureando:
Alessio Tezza
1189754

Anno Accademico 2021/2022

Abstract

L'obbiettivo di questo elaborato è quello di dare un'introduzione generale al concetto di reti neurali artificiali e successivamente applicare uno degli svariati algoritmi su un caso pratico per vederne effettivamente l'utilità.

La scelta di questo specifico argomento ricade sul fatto che negli ultimi anni, con l'avvento delle nuove tecnologie, si sente sempre più spesso parlare di numerose applicazioni che non sono altro che esempi di reti neurali, quali per esempio guida autonoma, intelligenza artificiale, riconoscimento della scrittura manuale, riconoscimento vocale ecc...

L'elaborato si suddivide in:

- Capitolo 1: Storia e introduzione alle reti neurali artificiali, con esempi pratici e cenni di concetti e teoremi fondamentali per la completa comprensione degli argomenti successivi.
- Capitolo 2: Teoria della classificazione tramite percettrone a singolo strato, ovvero la più semplice rete neurale artificiale che si possa progettare per avere una classificazione tramite una separazione lineare di due differenti classi. In questo capitolo si approfondirà anche il concetto di *training* della rete neurale con un apprendimento automatico supervisionato.
- Capitolo 3: Caso di studio pratico, classificazione dell'*Iris dataset* ovvero un insieme di dati che contiene quattro proprietà di questa tipologia di fiore, che in base a queste dovranno essere classificate in 3 specie differenti. Si discuterà come implementare un software in *Python* per realizzare l'inizializzazione dei dati e l'effettivo algoritmo con rete neurale a singolo strato per la classificazione di questo dataset.
- Capitolo 4: Analisi dei risultati ottenuti dal codice implementato nel capitolo 3, possibili estensioni dell'algoritmo e miglioramenti.

Indice

1 Introduzione alle reti neurali artificiali **1**

1.1 Riconoscimento del carattere 2
1.2 Cenni di classificazione 3
1.3 Adattamento polinomiale 4
1.4 Cenni di probabilità: Teorema di Bayes 6

2 Classificazione tramite Percettrone a Singolo-Strato **9**

2.1 Discrminante lineare 9
2.2 Discrminante non-lineare 10
2.3 Separazione lineare 12
2.4 Percettrone a singolo strato 13
2.5 Tecniche di esercitazione delle reti 14
2.6 Esercitazione del percettrone 17

3 Caso di studio: Iris Dataset **21**

3.1 Inizializzazione dei dati 22
3.2 Analisi del codice 23

4 Conclusioni **27**

4.1 Miglioramenti e possibili estensioni 30

Bibliografia **33**

Capitolo 1

Introduzione alle reti neurali artificiali

Negli ultimi anni la computazione grazie alle reti neurali è diventata un aspetto fondamentale della tecnologia quotidiana, con applicativi di successo in numerosi campi. La grande maggioranza di questi ultimi riguarda soprattutto l'ambito del "**pattern recognition**" (riconoscimento del modello), che comprende una grande varietà di elaborazione dei dati, partendo dal riconoscimento vocale, il riconoscimento di caratteri scritti a mano eccetera..

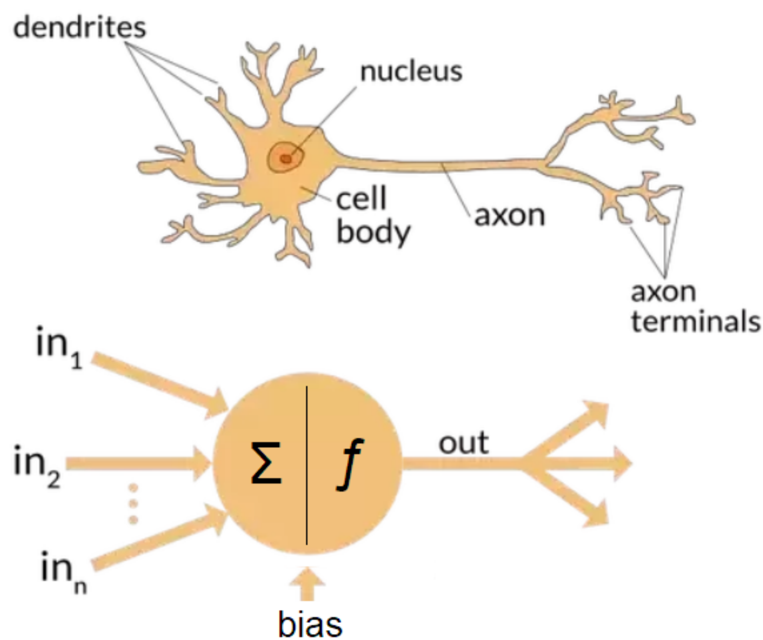

Figura 1.1: Ispirazione delle reti neurali da modelli biologici [8].

L'idea delle reti neurali deriva appunto da un modello di funzionamento dei neuroni del cervello umano (Fig.1.1). Il libro *Organization of Behaviour* (1949) propose un modello in cui i collegamenti neurali si rafforzano man mano che vengono utilizzati, specialmente quelli che collegano due neuroni che vengono 'utilizzati' nello stesso istante.

Da questo modello biologico vengono introdotti negli anni il concetto di *Machine Learning*, ovvero la piattaforma per automatizzare, sottoforma di algoritmi, modelli statistici per fare previsioni sui altri modelli; le *reti neurali artificiali* che sono quella porzione di modello che permette l'apprendimento da parte della rete; e le *Deep Networks* che sono un estensione più complessa delle reti neurali che permettono di fare classificazioni a problemi più complicati ed in maniera più accurata (reti neurali multi-strato).

La soluzione più generale a questo tipo di problema è l'**approccio probabilistico**, che riconosce la natura stocastica sia delle informazioni che si ricevono, sia del risultato finale. Questo tipo di approccio alla risoluzione del problema resta valido anche nei casi esposti in seguito, dove si utilizza un metodo di *apprendimento automatico supervisionato*, il quale consiste nell'allenare il sistema con un insieme prestabilito di dati per cercare di estrapolarne la regola ad essi sottesa, ed in seguito utilizzarla per fare delle previsioni su altri dati. Esistono poi anche altre tipologie di apprendimento che però non verranno trattate in questo documento.

1.1 Riconoscimento del carattere

Si potrebbe partire da qualsiasi esempio per introdurre il concetto di "pattern recognition", ma in questo caso si tratterà il caso del riconoscimento di due caratteri scritti a mano 'a' e 'b'. Il risultato desiderato è quello di sviluppare un algoritmo che sia in grado, data un immagine rappresentata da un vettore di pixel $\mathbf{x}$, di assegnarla ad una delle due classi $\mathcal{C}_k$ con $k = 1, 2$, dove $\mathcal{C}_1$ corrisponde al carattere 'a' e $\mathcal{C}_2$ corrisponde al carattere 'b'.

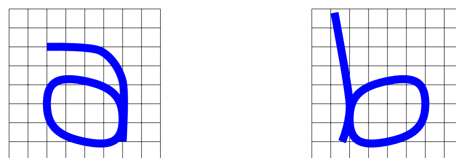

Figura 1.2: Vettori di input di un modello di *pattern recognition* di due caratteri.

In questo caso però non è possibile analizzare ogni volta tutti i pixel, ma si devono ricavare delle **features** da ogni vettore di input che permettano di analizzare più agilmente il problema di classificazione.

In questo preciso caso per il riconoscimento di questi due caratteri si potrebbe usare il rapporto tra l'altezza del carattere e la sua lunghezza, che viene denominato con $\widetilde{x_1}$, dal quale ci si può chiaramente aspettare che a valori più grandi di $\widetilde{x_1}$ corrisponda il carattere 'b', e viceversa, a valori più piccoli di $\widetilde{x_1}$ corrisponda il carattere 'a'. Ma anche in questo caso per valori intermedi di $\widetilde{x_1}$ non si riesce a distinguere bene i due caratteri, quindi si dovranno aggiungere altre *features* in modo da riuscire ad identificare in maniera corretta i due, sapendo comunque che non si potrà mai avere una classificazione perfetta di tutti i nuovi input.

1.2 Cenni di classificazione

Per quanto descritto nella sezione precedente, si può ipotizzare di avere un sistema con input $x_1, \ldots, x_d$ che rappresentano l'intensità dei pixel, ed avere come output una variabile y che sarà uguale ad 1 nel caso in cui l'immagine sia riconosciuta come di classe $\mathcal{C}_1$, e sarà uguale a 0 nel caso appartenga alla classe $\mathcal{C}_2$. Si riesce a sintetizzare una mappa di questi elementi con la funzione:

$$y_k = y_k(\mathbf{x}, \mathbf{w}) \tag{1.1}$$

dove $\mathbf{w}$ rappresenta un vettore di parametri, anche chiamati pesi (dall'inglese *weights*). L'importanza fondamentale delle reti neurali artificiali sta nel fatto che offrono un metodo generale per rappresentare mappe non-lineari partendo da diversi input fino a diversi output, dove i collegamenti di questa mappatura possono essere regolati da questi **pesi** $\mathbf{w}$.

Il processo di determinazione dei valori dei pesi si chiama *learning* oppure *training*, e per questo motivo generalmente un dataset di esempi per queste reti viene chiamato *training set*.

Pre-processing ed estrazione delle feature

Invece di rappresentare tutta la trasformazione da un vettore $\mathbf{x}$ di input fino ad un vettore $\mathbf{y}$ di output, si ha spesso un grande vantaggio nella separazione della mappatura in una sezione iniziale chiamata **pre-processing**.

Tenendo come esempio il caso di riconoscimento di caratteri si può chiamare *pre-processing* il cambiamento di valutazione dell'input, da un vettore rappresentante i pixel $\mathbf{x}$ ad una *feature* di esso, per esempio $\widetilde{x_1}$.

La distinzione tra pre-processing (oppure in questo caso 'estrazione della feature') non è mai chiaramente distinta dal resto della rete neurale, ma si può notare come il pre-processing sia una trasformazione finita dell'input, e invece la rete neurale è in continuo cambiamento e miglioramento. Vista la grande importanza del pre-processing si deve garantire che le *feature* che vengono estratte soddisfino sia l'**invarianza alla traslazione**, sia l'**invarianza alla scalabilità**.

1.3 Adattamento polinomiale

La maggior parte delle complessità che derivano dalle applicazioni delle reti neurali si può introdurre nel conteso più semplice dell'adattamento polinomiale. Si ha quindi il problema di adattare un polinomio a un numero N di punti con la tecnica della minimizzazione dell'errore. Si consideri un polinomio di ordine M definito come:

$$y(x) = w_0 + w_1 \cdot x + w_2 \cdot x^2 + \cdots + w_M \cdot x^M = \sum_{j=0}^{M} w_j \cdot x^j \tag{1.2}$$

Questo può essere inteso come un mappa non-lineare che prende x come ingresso e produce un'uscita y. Se si impone il polinomio di ordine N, avendo così un grado del polinomio per ogni punto desiderato per l'uscita y, e definendo t come il *target* di questo punto, si può scrivere la funzione di errore come la minimizzazione del quadrato dell'errore:

$$E = \frac{1}{2} \sum_{n=1}^{N} \left[y(x^n, \mathbf{w}) \; - \; t^n \right]^2 \tag{1.3}$$

La minimizzazione di questa funzione d'errore è chiamata *apprendimento supervisionato* siccome si conosce a priori la classificazione esatta dell'input t, che viene valutata con il risultato del nostro modello y per aggiustarne i valori.

A questo punto si può illustrare la tecnica del adattamento polinomiale generando dei punti (in questo caso sono punti equispaziati della funzione $\sin(x)$) e vedendo il comportamento di questa tecnica aumentando i gradi di libertà, possiamo quindi analizzare i risultati dalla figura seguente:

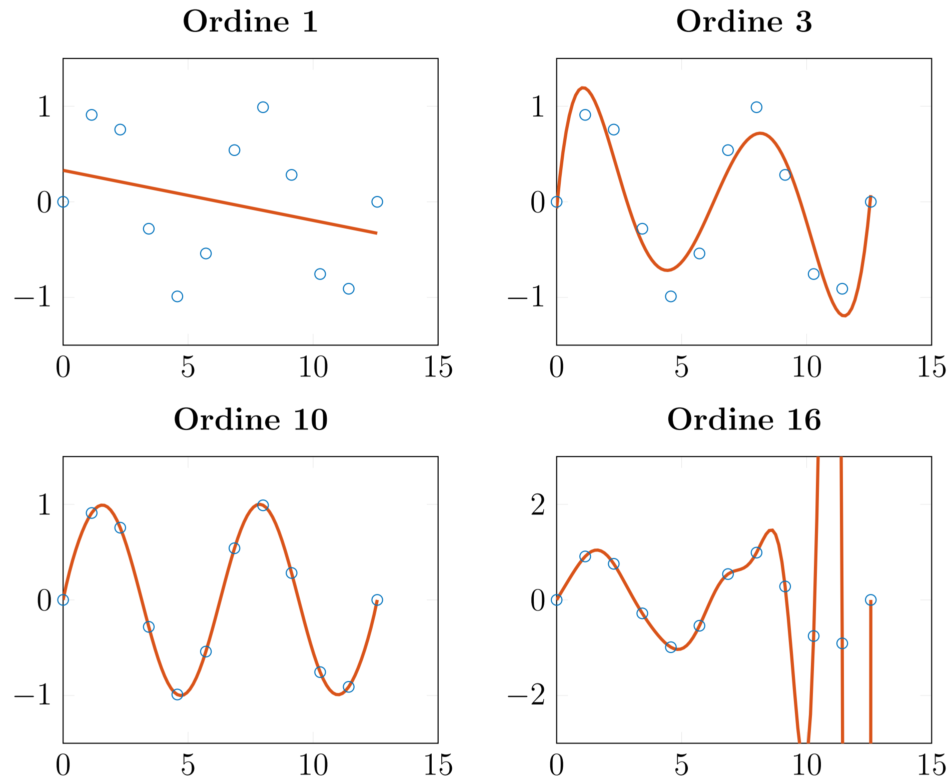

Figura 1.3: Adattamento polinomiale in base all'ordine crescente del polinomio.

Si può chiaramente notare come aumentando l'ordine del polinomio si ottenga un adattamento migliore ai punti dati, perché aumentando il grado di libertà della funzione si aumenta anche la sua flessibilità; ma da un certo punto in poi aumentando il grado di libertà si avrà sempre un adattamento perfetto a discapito di oscillazioni molto elevate.

Complessità del modello

Usando l'esempio di Fig.1.3 dell'adattamento polinomiale si può capire che la generalizzazione migliore si ha quando la complessità del modello non è ne troppo piccola ne troppo grande. Il problema di trovare la complessità ottimale fornisce un ottimo esempio del *Rasoio di Occam*, il quale principio afferma che:

> "*...tra più ipotesi per la risoluzione di un problema, indica di scegliere, a parità di risultati, quella più semplice...*" [2]

Per questo motivo quindi si deve modificare la funzione d'errore E e ricavare una *complessità effettiva* del modello, che sarà data da la funzione d'errore stessa alla quale si aggiungerà un termine di penalità Ω:

$$\widetilde{E} = E + v\Omega \tag{1.4}$$

nella quale questo termine Ω è un termine che fornirà le indicazioni per comprendere le oscillazioni elevate della funzione polinomiale, può essere calcolato per esempio con una semplice derivata di secondo grado della funzione $y(x)$:

$$\Omega = \frac{1}{2}\int \left(\frac{d^2y}{dx^2}\right)^2 dx \tag{1.5}$$

Invece il parametro v è una costante arbitraria e controlla l'efficacia del termine di penalità Ω, controllando a sua volta l'effettiva complessità del modello.

1.4 Cenni di probabilità: Teorema di Bayes

Si ipotizzi di dover classificare un nuovo carattere in input, senza però fare nessuna misura sui dati in ingresso (per ricavarne le ipotetiche *features*); l'obbiettivo sarà quindi di classificare quest'ultimo minimizzando la probabilità d'errore.

Si definisce quindi la **probabilità a priori** $P(\mathcal{C}_k)$ di un immagine in input, presupposto di aver ricevuto un numero abbastanza elevato di campioni, la frequenza con la quale questi simboli sono apparsi in precedenza nella rete. Questa rappresenta la miglior opzione di classificazione che si ha senza fare misure sui dati in ingresso.

Si suppone in seguito di aver fatto le misure sul nuovo carattere in input, dalle quali si estrae la *feature* $\widetilde{x_1}$. Si ipotizzi che la nostra caratteristica sia assegnata a un certo valore discreto $\{X^l\}$; si definisce quindi la **probabilità congiunta** $P(\mathcal{C}_k, X^l)$ come la probabilità che un carattere con $\widetilde{x_1}$ di valore X^l sia assegnato alla classe $\mathcal{C}_k$.

Come ultimo, si definisce la **probabilità condizionata** $P(X^l|\mathcal{C}_k)$ che specifica la probabilità che l'osservazione sia nella 'colonna' X^l (vedi Fig.1.4) dato il fatto che sia di classe $\mathcal{C}_k$.

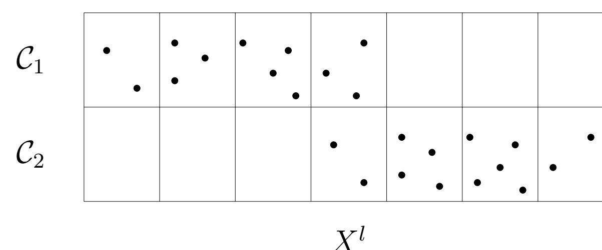

Figura 1.4: Probabilità condizionata di due classi, dati i valori discreti della *feature*.

Si può vedere che è valida la seguente uguaglianza per la probabilità congiunta:

$$P(\mathcal{C}_k, X^l) = P(X^l|\mathcal{C}_k)P(\mathcal{C}_k) \qquad \Leftrightarrow \qquad P(\mathcal{C}_k, X^l) = P(\mathcal{C}_k|X^l)P(X^l) \quad (1.6)$$

le due equazioni in (1.6) per la probabilità congiunta devono essere uguali, si può scrivere:

$$P(\mathcal{C}_k|X^l) = \frac{P(X^l|\mathcal{C}_k)P(\mathcal{C}_k)}{P(X^l)} \quad (1.7)$$

Questa espressione è detta *Teorema di Bayes*, la quantità sulla sinistra è chiamata **probabilità a posteriori**, siccome ci da la probabilità che la classificazione sia $\mathcal{C}_k$ dopo che abbiamo fatto la misura per $\widetilde{x_1}$.

A questo punto si differenziano due fasi importanti, la fase di *training* (esercitazione) dove si usano dei dati predefiniti per determinare i valori della probabilità a posteriori in Fig.1.4; e la seconda fase di *decisione* nella quale questi valori vengono usati per classificare nuovi punti nelle possibili classi.

Teorema di Bayes in generale

Fino ad ora la variabile $\widetilde{x_1}$ è sempre stata discretizzata in un serie di valori finiti. Ma in molteplici casi questo non è possibile e si ha un valore continuo, per questo si definisce la **densità di probabilità** come:

$$P(x \in [a, b]) = \int_a^b p(x)dx \quad (1.8)$$

Quindi dice che la densità di probabilità $p(x)$ specifica la probabilità di una variabile x di essere in un intervallo tra due punti $[a, b]$. Si riformula dunque il *Teorema di Bayes* con la densità di probabilità come:

$$P(\mathcal{C}_k|x) = \frac{p(x|\mathcal{C}_k)P(\mathcal{C}_k)}{p(x)} \quad (1.9)$$

Dalla quale si può dedurre facilmente l'uguaglianza:

$$p(x) = p(x|\mathcal{C}_1)P(\mathcal{C}_1) + p(x|\mathcal{C}_2)P(\mathcal{C}_2) \quad (1.10)$$

Regioni di decisione

La probabilità a posteriori $P(\mathcal{C}_k|\mathbf{x})$ fornisce la probabilità di un input di appartenere alla classe $\mathcal{C}_k$ dopo aver osservato le sue *feature* $\mathbf{x}$. La probabilità d'errore quindi viene minimizzata selezionando la classe $\mathcal{C}_k$ che ha la più grande probabilità a posteriori, in altre parole, un vettore di *feature* $\mathbf{x}$ è assegnato alla classe $\mathcal{C}_k$ se:

$$P(\mathcal{C}_k|\mathbf{x}) > P(\mathcal{C}_j|\mathbf{x}) \qquad \forall j \neq k \tag{1.11}$$

Si immagini quindi uno spazio diviso in c **regioni di decisione** da $\mathcal{R}_1$ fino a $\mathcal{R}_c$ (uno per ogni carattere da riconoscere), in modo che se un punto cade nella regione di decisione $\mathcal{R}_k$ sarà assegnato alla classe $\mathcal{C}_k$.

Bisogna quindi cercare un criterio che realizzi un metodo per ricavare queste regioni di decisione. Sapendo che avverrà un errore di decisione quando un nuovo input che sarebbe di classe $\mathcal{C}_1$ viene assegnato alla classe $\mathcal{C}_2$, si definisce quindi calcolare la probabilità di commettere un errore come:

$$\begin{aligned} P(\text{errore}) &= P(x \in \mathcal{R}_2, \mathcal{C}_1) + P(x \in \mathcal{R}_1, \mathcal{C}_2) \\ &= P(x \in \mathcal{R}_2|\mathcal{C}_1)P(\mathcal{C}_1) + P(x \in \mathcal{R}_1|\mathcal{C}_2)P(\mathcal{C}_2) \\ &= \int_{\mathcal{R}_2} p(x|\mathcal{C}_1)P(\mathcal{C}_1)\,dx + \int_{\mathcal{R}_1} p(x|\mathcal{C}_2)P(\mathcal{C}_2)\,dx \end{aligned} \tag{1.12}$$

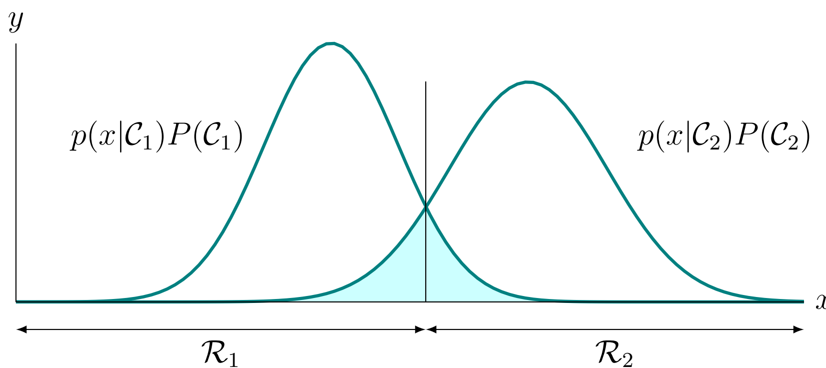

Figura 1.5: Scelta ottimale delle regioni di decisione, per la minimizzazione dell'errore.

Come si nota dalla Fig.1.5 la divisione ottimale delle regioni di decisione si ha quando si sceglie la regione di decisione $\mathcal{R}_k$ in modo che, ad ogni $\mathbf{x}$ sia assegnata la classe per la quale l'integrando è massimizzato; e si avrà una classificazione errata solo nel caso il punto corrisponda alla zona azzurra del grafico.

Capitolo 2

Classificazione tramite Percettrone a Singolo-Strato

Nel primo capitolo viene descritto il concetto di regione ottima di decisione per minimizzare la probabilità d'errore quando si deve classificare un nuovo simbolo in ingresso. Nella pratica però, vengono scelte accuratamente delle funzioni discriminanti per la classificazione dei nuovi simboli, che sono ricavate da un algoritmo di **esercitazione** della rete e un insieme di dati predefiniti.

La scelta più semplice della funzione discriminante è di prendere una combinazione lineare delle variabili di input, cosicchè i coefficienti di questa combinazione lineare siano i parametri del modello, chiamati **pesi**. Ma esistono anche altre tecniche per determinare questi pesi nelle reti a singolo strato quali *perceptron learning*, *least-square method* e il *discriminante di Fisher*.

2.1 Discrminante lineare

Si supponga di dover considerare un problema di classificazione tra due classi: in questo modo vorremo avere una funzione discriminante $y(\mathbf{x})$ che classifichi il vettore $\mathbf{x}$ alla classe $\mathcal{C}_1$ quando $y(\mathbf{x}) > 0$, e alla classe $\mathcal{C}_2$ quando $y(\mathbf{x}) < 0$, quindi la scelta più semplice per risolvere questo problema è scegliere una funzione discriminante che sia una combinazione lineare delle componenti del vettore $\mathbf{x}$ e sia quindi del tipo:

$$y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0 \tag{2.1}$$

dove $\mathbf{w}$ è un vettore di dimensione d, e il termine w_0 è detto *bias*. Combinando tutto questo, e riducendo l'espressione ad un vettore di dimensioni $(d+1)$ dove $\widetilde{\mathbf{x}} = (1, \mathbf{x})$ e $\widetilde{\mathbf{w}} = (w_0, \mathbf{w})$ e questo punto si riscrive la (2.1) come:

$$y(\mathbf{x}) = \widetilde{\mathbf{w}}^T \widetilde{\mathbf{x}} \tag{2.2}$$

Dunque si può dire che la il **bordo di separazione** delle regioni di decisione sarà un iperpiano di dimensione d che soddiferà l'equazione $y(\mathbf{x}) = 0$.

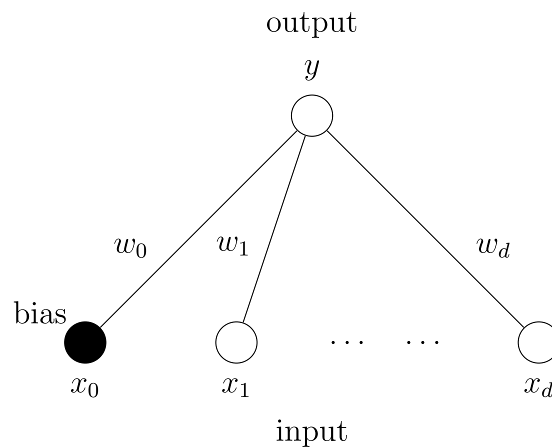

Figura 2.1: Rete neurale a singolo strato con discriminante lineare.

In Fig.2.1 viene rappresentata una rete a singolo strato e si vede come gli input sono rappresentati sotto forma di cerchi che sono connessi attraverso i corrispondenti pesi all'uscita y. Il *bias* w_0 è rappresentato come un peso che collega all'uscita un input x_0 che è sempre uguale ad uno.

2.2 Discrminante non-lineare

Sono state trattate fino a questo punto solo funzioni discriminanti che erano semplicemente combinazioni lineari delle variabili di input. Si può generalizzare meglio questo aspetto considerando l'utilizzo di una funzione non-lineare $g(\cdot)$ che trasforma la somma lineare e restituisce un discriminante per il problema di classificazione della forma:

$$y = g(\mathbf{w}^T \mathbf{x} + w_0) \tag{2.3}$$

dove $g(\cdot)$ è detta **funzione di attivazione** e generalmente è scelta in modo da essere monotona (crescente o descrescente).

Come motivazione della scelta di questo discriminante possiamo fare un esempio con un problema di classificazione di due classi, nel quale le densità della classe sono date da distribuzioni gaussiane con la stessa matrice di covarianza Σ:

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{1}{(2\pi)^{d/2}|\Sigma|^{1/2}} \cdot \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_k)^T \Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu}_k)\right] \tag{2.4}$$

A questo punto ricorrendo al teorema di Bayes,si ricava la probabilità a posteriori di un membro della classe $\mathcal{C}_1$ come:

$$\begin{aligned} P(\mathcal{C}_1|\mathbf{x}) &= \frac{p(\mathbf{x}|\mathcal{C}_1)P(\mathcal{C}_1)}{p(\mathbf{x}|\mathcal{C}_1)P(\mathcal{C}_1)+p(\mathbf{x}|\mathcal{C}_2)P(\mathcal{C}_2)} \\ &= \frac{1}{1+\exp(-a)} = g(a) \end{aligned} \tag{2.5}$$

dove $g(a)$ è la **funzione di attivazione sigmoidale** che viene disegnata in Fig.2.2, e si ottiene che:

$$a = \mathbf{w}^T\mathbf{x} + w_0 \tag{2.6}$$

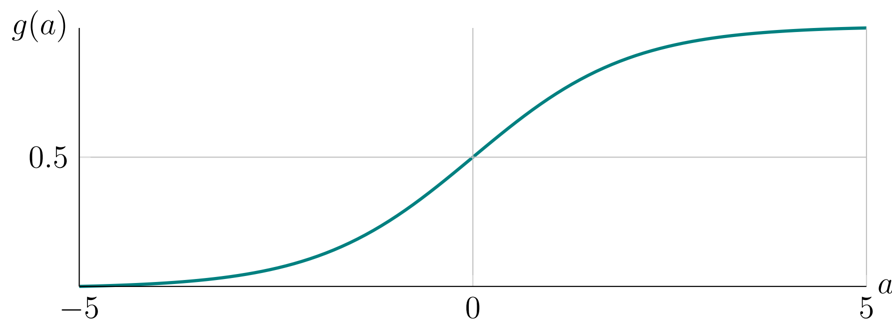

Figura 2.2: Grafico della funzione di attivazione sigmoidale.

Il termine *sigmoide* si riferisce al fatto che il grafico della funzione sia a forma di 'S', e questa funzione mappa l'intervallo $(-\infty, +\infty)$ in $(0, 1)$, otteniamo dunque che se $|a|$ è piccolo questa funzione può essere approssimata come una funzione lineare; e quindi si può dire che una rete con una funzione di attivazione sigmoidale 'contiene' una rete lineare come sotto-caso.

Un altra forma di discriminante lineare è stata introdotta da McCulloch e Pitts (1943) come un modello matematico per il comportamento di un singolo neurone in un sistema nervoso biologico.

Questa prende sempre la forma della (2.3) con una funzione di attivazione a soglia che è chiamata **Gradino di Heaviside**:

$$g(a) = \begin{cases} 0 \;\Rightarrow\; a < 0 \\ 1 \;\Rightarrow\; a \geq 0 \end{cases} \tag{2.7}$$

2.3 Separazione lineare

É stato esposto come le funzioni discriminanti abbiano bordi di decisione che sono lineari, o più in generale sono di una dimensione in meno rispetto alla nostra funzione d'uscita.

Un campo veramente ristretto di problemi possono essere risolti con questa soluzione; e questa quindi è la principale limitazione delle reti a singolo-strato, le quali lasciano posto alle reti multi-strato per la risoluzione di problemi che necessitano regioni di decisione con bordi non lineari.

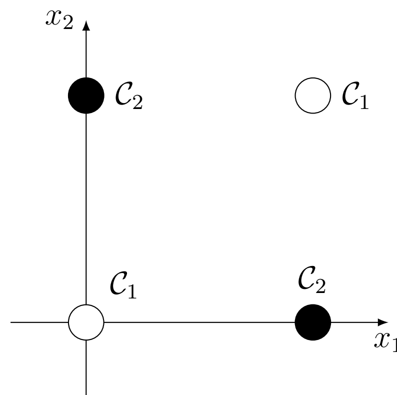

Figura 2.3: Problema di classificazione della porta XOR

Come si può notare dalla Fig.2.3 il problema di classificazione della porta XOR non può essere classificato correttamente da due regioni di decisione il quale bordo deve essere lineare. In caso contrario, cioè nel caso in cui si abbia un problema di classificazione per il quale si riesca a risolvere correttamente con un bordo di decisione lineare si dice che i punti di esso sono **linearmente separabili**.

Quindi la considerazione chiave di questo concetto è che per ogni problema bisogna capire quale sia la miglior funzione discriminante da utilizzare. Questo viene capito grazie alle conoscenze a priori del problema da risolvere e alla forma che deve avere la soluzione, messo insieme con un confronto delle prestazioni di modelli alternativi.

2.4 Percettrone a singolo strato

Le reti a singolo strato con funzioni di attivazione a soglia del tipo (2.7), studiate da Rosenblatt (1962), sono state denominate percettroni. Nello stesso periodo in cui Rosenblatt stava sviluppando i percettroni, Widrow e i suoi colleghi erano sulla sua stessa strada usando sistemi che chiamavano *adalines*, ovvero ADAptive LINear Element (elementi lineari adattivi), che erano essenzialmente uguali ai percettroni.

Discriminanti lineari generalizzati

Un metodo per generalizzare le funzioni discriminanti in modo da permettere un campo più elevato di possibili bordi di decisione, si ha trasformando il vettore di input $\mathbf{x}$ usando M predefinite funzioni non-lineari $\varphi_j(\mathbf{x})$, chiamate *basi*, e poi rappresentare l'output come combinazione lineare di queste funzioni:

$$y_k(\mathbf{x}) = \sum_{j=1}^{M} w_{kj}\varphi_j(\mathbf{x}) + w_{k0} \tag{2.8}$$

Questo rappresenta delle classi di funzioni più ampie $y_k(\mathbf{x})$, infatti scegliando accuratamente le basi questa funzione può approssimare qualsiasi trasformazione continua in una qualsiasi accuratezza.

Le rete multi-strato spesso possono essere viste come funzioni discriminanti generalizzate, ma nelle quali le basi possono essere modificate durante il processo.

Usando quindi la convenzione che il termine di bias sia associato ad una base extra chiamata φ_0 la quale funzione di attivazione è impostata sempre ad uno, si ha che l'output del *perceptron* sarà:

$$y = g\left(\sum_{j=0}^{M} w_j\varphi_j(\mathbf{x})\right) = g(\mathbf{w}^T\boldsymbol{\varphi}) \tag{2.9}$$

dove $\boldsymbol{\varphi}$ indica il vettore formato da $\varphi_0, \ldots, \varphi_M$. La funzione di attivazione dell'uscita di solito viene scelta come una versione anti-simmetrica della funzione di soglia a gradino di Heaviside, sarà quindi del tipo:

$$g(a) = \begin{cases} -1 \;\Rightarrow\; a < 0 \\ +1 \;\Rightarrow\; a \geq 0 \end{cases} \tag{2.10}$$

2.5 Tecniche di esercitazione delle reti

Sono state esposte in questo capitolo le varie forme delle reti a singolo-strato ed esplorate le loro proprietà, ora si analizzerà come trovare i vari pesi/parametri, e quindi, in altre parole, come fare il *training* (esercitazione) delle reti.

Funzione d'errore "somma dei quadrati"

La funzione d'errore *sum of squares* è la più semplice forma di funzione d'errore che si può trovare. Partendo da una funzione simile alla (2.8), e riscrivendola come:

$$y_k(\mathbf{x}) = \sum_{j=0}^{M} w_{kj}\varphi_j(\mathbf{x}) \tag{2.11}$$

in modo tale da implementare il *bias* all'interno della sommatoria ed associarlo alla base φ_0. Si può definire la funzione d'errore "somma dei quadrati" come la somma su tutti i pattern (tutte le coppie input/output) del *training set* e su tutti gli output, del tipo:

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}\left[y_k(\mathbf{x}^n;\mathbf{w}) \; - \; t_k^n\right]^2 \tag{2.12}$$

dove i termini rappresentano rispettivamente:

- $y_k(\mathbf{x}^n;\mathbf{w})$ rappresenta l'output dell'unità k-esima dato in ingresso un input $\mathbf{x}^n$ e il vettore di pesi attuali $\mathbf{w}$.
- N è il numero di pattern dati per l'esercitazione.
- c rappresenta il numero di unità di output.
- t_k^n è il *target*, ovvero il valore desiderato che dovrebbe avere la variabile $y_k(\mathbf{x})$ quando viene classificata correttamente.

Si nota chiaramente che la (2.12) è una funzione che dipende solo dai pesi e può essere minimizzata. Si procede, senza perdita di generalità, supponendo di abbassare la dimensione del problema in modo che la soluzione $\vec{y}$ sia nello stesso piano del target $\vec{t}$, a questo punto si può riscrivere la soluzione come:

$$\vec{y} = \sum_{j=0}^{M} w_j\vec{\varphi}_j \tag{2.13}$$

La (2.12) può essere riscritta come differenza della distanza euclidea data da:

$$E = \frac{1}{2} \left\| \sum_{j=0}^{M} w_j \vec{\varphi}_j - \vec{t} \right\|^2 \tag{2.14}$$

che, quando viene minimizzata con rispetto dei pesi si ricava:

$$\frac{\partial E}{\partial w_j} = 0 = \vec{\varphi}^{\,T} (\vec{y} - \vec{t} \,), \qquad j = 1, \ldots, M \tag{2.15}$$

questa equazione viene soddisfatta quando $\vec{y}$ è uguale a $\vec{t}$, risultato che ci si aspettava perché significa che l'errore sarà minimizzato quando la classificazione del pattern sarà quella corretta definita dal target.

Discesa del gradiente

La discesa del gradiente (dall'inglese *gradient descent*) è un metodo per ricavare i valori dei pesi, minimizzando la funzione d'errore (2.12). Questo metodo è utilizzato rispetto ad altri soprattutto perché si può utilizzare su una rete con una funzione di attivazione non lineare (vedi sigmoide Fig.2.2) e anche su reti che hanno una funzione d'errore diversa da quella della somma dei quadrati.

Si raggruppano inizialmente tutti i parametri (pesi e bias) per formare un unico vettore di pesi $\mathbf{w}$ in modo tale che la funzione d'errore possa essere espressa come $E = E(\mathbf{w})$. Un vincolo importante è che la funzione d'errore deve essere differenziabile come in questo caso, si inizia quindi dando un valore qualsiasi al vettore $\mathbf{w}$ per poi, passo dopo passo correggere i pesi apportando piccole modifiche ad ogni iterazione nella direzione dove l'errore decresce, ovvero nella direzione di $-\nabla_{\mathbf{w}} E$. Ripetendo questi passi più volte si ottiene una sequenza di vettori:

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} - \eta \cdot \left. \frac{\partial E}{\partial w_{kj}} \right|_{\mathbf{w}^{(\tau)}} \tag{2.16}$$

dove in questo caso η è un numero positivo piccolo chiamato *tasso di esercitazione* (dall'inglese 'learning rate'), la quale scelta rappresenta un punto fondamentale della progettazione dell'esercitazione della rete. Infatti scegliendolo troppo piccolo si avrà che la funzione d'errore sarà troppo lenta ad arrivare ad un minimo, al contrario, scegliendo η troppo grande si avranno oscillazioni nel sistema che porterà alla divergenza.

In generale si ha che la funzione d'errore è data dalla somma di più termini, ognuno dei quali rappresenta la funzione d'errore calcolata su un solo pattern n. In questo caso si può aggiornare il vettore dei pesi usando solo un pattern alla volta:

$$w_{kj}^{(\tau+1)} = w_{kj}^{(\tau)} \; - \; \eta \cdot \frac{\partial E^n}{\partial w_{kj}} \tag{2.17}$$

che viene poi ripetuta più volte, iterando per tutti i pattern del sistema.

A questo punto per implementare la *discesa del gradiente* non resta altro che trovare un espressione esplicita per la derivata della funzione d'errore rispetto ai pesi, considerando dunque la (2.17) per una rete lineare generalizzata le derivate sono date da:

$$\frac{\partial E^n}{\partial w_{kj}} = \left[y_k(\mathbf{x}^n) - t_k^n \right] \varphi_j(\mathbf{x}^n) = \delta_k^n \cdot \varphi_j^n \tag{2.18}$$

dove si abbrevia l'espressione imponendo:

$$\delta_k^n = y_k(\mathbf{x}^n) - t_k^n \tag{2.19}$$

quindi la variazione che avrà un peso dato un particolare pattern in ingresso sarà:

$$\Delta w_{kj} = -\eta \cdot \delta_k^n \cdot \varphi_j^n \tag{2.20}$$

Questa regola è detta **regola LMS**, ovvero regola *least mean square rule.*

Momentum learning

La tecnica del *momentum learning* è una delle varie tecniche di ottimizzazione dell'algoritmo della discesa del gradiente. Questo consiste nell'aggiungere 'inerzia' ogni volta che si modifica il valore dei pesi della rete, facendo in modo avere variazioni più elevate all'inizio, e smorzarle sempre di più verso la fine dell'esercitazione della rete. Quindi si può riscrivere la formula della discesa del gradiente come:

$$\Delta \mathbf{w}^{(\tau)} = -\eta \nabla E|_{\mathbf{w}^{(\tau)}} \; + \; \mu \Delta \mathbf{w}^{(\tau-1)} \tag{2.21}$$

dove chiamiamo μ il parametro di *momentum* ed è un numero positivo scelto tra $0 \leq \mu \leq 1$. Facendo il conto su tutte le iterazioni si ottiene il seguente risultato:

$$\begin{aligned} \Delta \mathbf{w} &= -\eta \nabla E \, \left[1 + \mu + \mu^2 + \dots \right] \\ &= -\frac{\eta}{1-\mu} \nabla E \end{aligned} \tag{2.22}$$

Questo comporta una maggior efficienza nel termine di *learning* portandolo da η a $\eta/(1-\mu)$, per questo, come si nota dalle immagini successive, il parametro di *momentum* tende ad una rapida convergenza verso il minimo senza causare oscillazioni che portano alla divergenza del sistema.

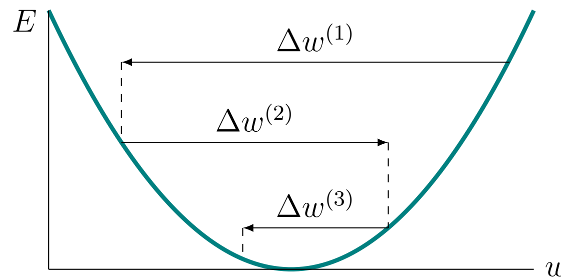

Figura 2.4: Discesa del gradiente con momentum learning.

2.6 Esercitazione del percettrone

Applicando la tecnica della discesa del gradiente per ogni pattern esposta in (2.17) al percettore si ottiene un espressione del tipo:

$$w_j^{(\tau+1)} = w_j^{(\tau)} + \eta \varphi_j^n t^n \tag{2.23}$$

Questo corrisponde ad un algoritmo di esercitazione relativamente semplice, che può essere interpretato come segue: si iteri su tutti i pattern del *training set*, e si associ ad ogni pattern l'attuale vettore dei pesi. Se il pattern viene classificato correttamente l'algoritmo non fa nulla, altrimenti aggiunge questo vettore di pattern (moltiplicato per η) al vettore di pesi se il pattern è stato classificato di classe $\mathcal{C}_1$, oppure sottrae nel caso sia classificato di classe $\mathcal{C}_2$.

Teorema di convergenza del percettrone

Un risultato molto importante nell'ambito delle reti neurali cita:

> "*Per ogni insieme di dati che risulta linearmente separabile, la tecnica di esercitazione della rete esposta in* (2.23) *garantisce di trovare una soluzione in un numero finito di passi.*" [3]

Questa scoperta prende il nome di **teorema di convergenza del percettrone**, al quale si darà una semplice dimostrazione, basata su quanto scritto da Hertz (1991).

Visto che si sta considerando un insieme di dati linearmente separabili è evidente che esiste almeno un vettore di pesi $\widehat{\mathbf{w}}$ per il quale tutti i vettori di esercitazione sono classificati correttamente, per cui:

$$\widehat{\mathbf{w}} \cdot \boldsymbol{\varphi}^n \cdot t^n > 0 \qquad \forall n \tag{2.24}$$

Il processo di esercitazione della rete inizia con un qualsiasi vettore dei pesi, che senza perdita di generalità si assume siano tutti zero. Ad ogni iterazione τ questo vettore viene aggiornato con:

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} \; + \; \boldsymbol{\varphi}^n t^n \tag{2.25}$$

dove $\boldsymbol{\varphi}^n$ è un vettore che viene classificato erroneamente dal percettrone. Dopo un numero τ^n di iterazioni, il vettore dei pesi sarà dato da:

$$\mathbf{w} = \sum_n \tau^n \boldsymbol{\varphi}^n t^n \tag{2.26}$$

Si prenda ora il prodotto scalare di questa equazione con $\widehat{\mathbf{w}}$ si ottiene:

$$\widehat{\mathbf{w}}^T \cdot \mathbf{w} \geq \tau \cdot \min_n (\widehat{\mathbf{w}}^T \varphi^n t^n) \tag{2.27}$$

dove τ è la somma di tutti i τ^n su n, e quindi è il numero totale di variazioni fatte ai pesi. Dalla (2.24) segue che il valore di $\widehat{\mathbf{w}}^T\mathbf{w}$ dipende da una funzione che aumenta linearmente con τ. Dall'equazione della variazione dei pesi ad ogni iterazione (2.20) si ricava che:

$$\Delta\|\mathbf{w}\|^2 = \left\|\mathbf{w}^{(\tau+1)}\right\|^2 - \left\|\mathbf{w}^{(\tau)}\right\|^2 \leq \|\boldsymbol{\varphi}\|^2_{\max} \tag{2.28}$$

e quindi, dopo τ iterazioni di aggiornamento del vettore dei pesi risulterà:

$$\|\mathbf{w}\|^2 \leq \tau \, \|\boldsymbol{\varphi}\|^2_{\max} \tag{2.29}$$

A questo punto la lunghezza del vettore $\mathbf{w}$ non può crescere più velocemente di $\sqrt{\tau}$, e dalla precedente osservazione nella quale si espone come $\widehat{\mathbf{w}}^T\mathbf{w}$ dipende linearmente da τ, per τ troppo elevati i due risultati non coinciderebbero. Quindi τ non può infinitamente crescere e questo porta a concludere che l'algoritmo deve convergere in un numero finito di passi.

Esempio di esercitazione del percettrone

Si espone nelle seguenti immagini una rappresentazione grafica dell'esercitazione di un percettrone. In questo caso per semplicità è stata scelta solo una base φ_1 da cui, con il bias incluso nel vettore dei pesi, si ottiene uno spazio in due dimensioni (φ_0, φ_1). É dato un insieme di dati dove i cerchi rappresentano gli elementi di classe $\mathcal{C}_1$ e i quadrati di classe $\mathcal{C}_2$.

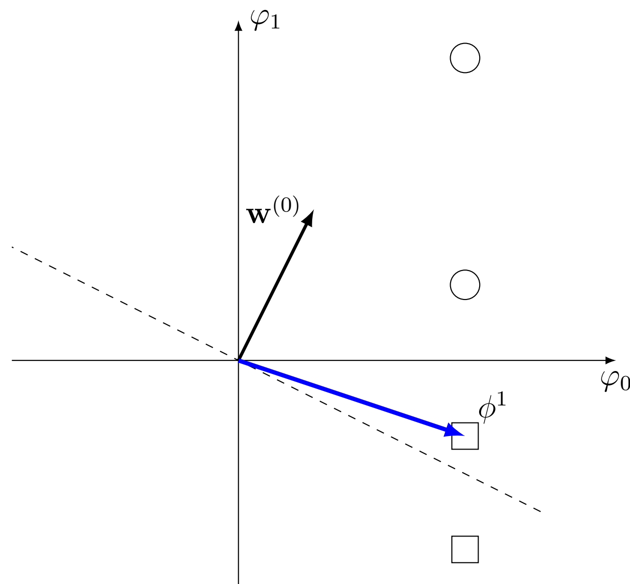

Figura 2.5: Prima iterazione dell'esercitazione del percettrone.

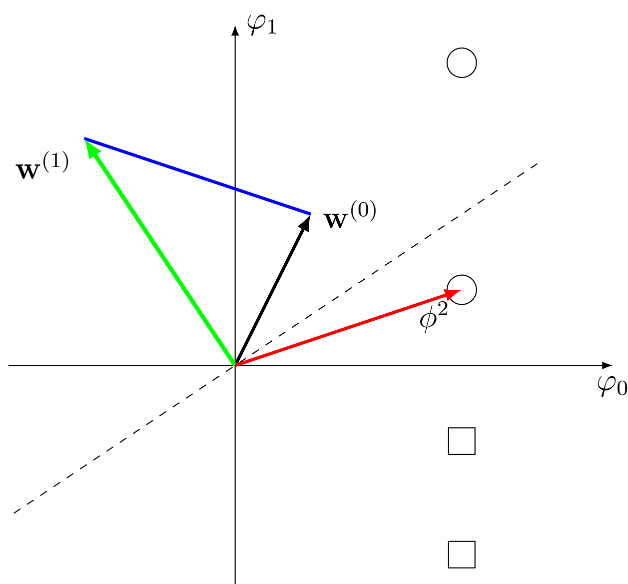

Figura 2.6: Seconda iterazione dell'esercitazione del percettrone.

In questo esempio il bordo della regione di decisione è rappresentato dalla linea tratteggiata, ortogonale al vettore dei pesi. Quest'ultimo si vede che si modifica con il vettore che è stato classificato erroneamente, che nel primo caso è rappresentato da ϕ^1 che sbaglia la classificazione di un quadrato, e nel secondo caso ϕ^2 sbaglia la classificazione di un cerchio.

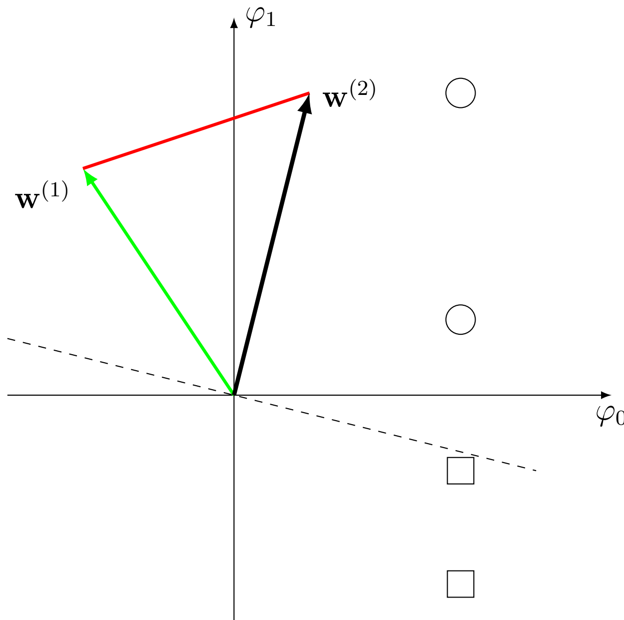

Figura 2.7: Terza ed ultima iterazione dell'esercitazione del percettrone.

Nell'ultimo caso si nota come, anche iterando su tutti i dati di esercitazione, vengano tutti classificati in maniera esatta, quindi l'algoritmo termina e il bordo della regione di decisione finale sarà quello rappresentato dalla linea tratteggiata.

Questo è un esempio di applicazione dell'algoritmo su un insieme di dati linearmente separabili. Se l'insieme di dati non fosse tale, l'algoritmo non si sarebbe concluso in un numero finito di passi (come dimostrato dal teorema) e continuerebbe ad iterare all'infinito.

Capitolo 3

Caso di studio: Iris Dataset

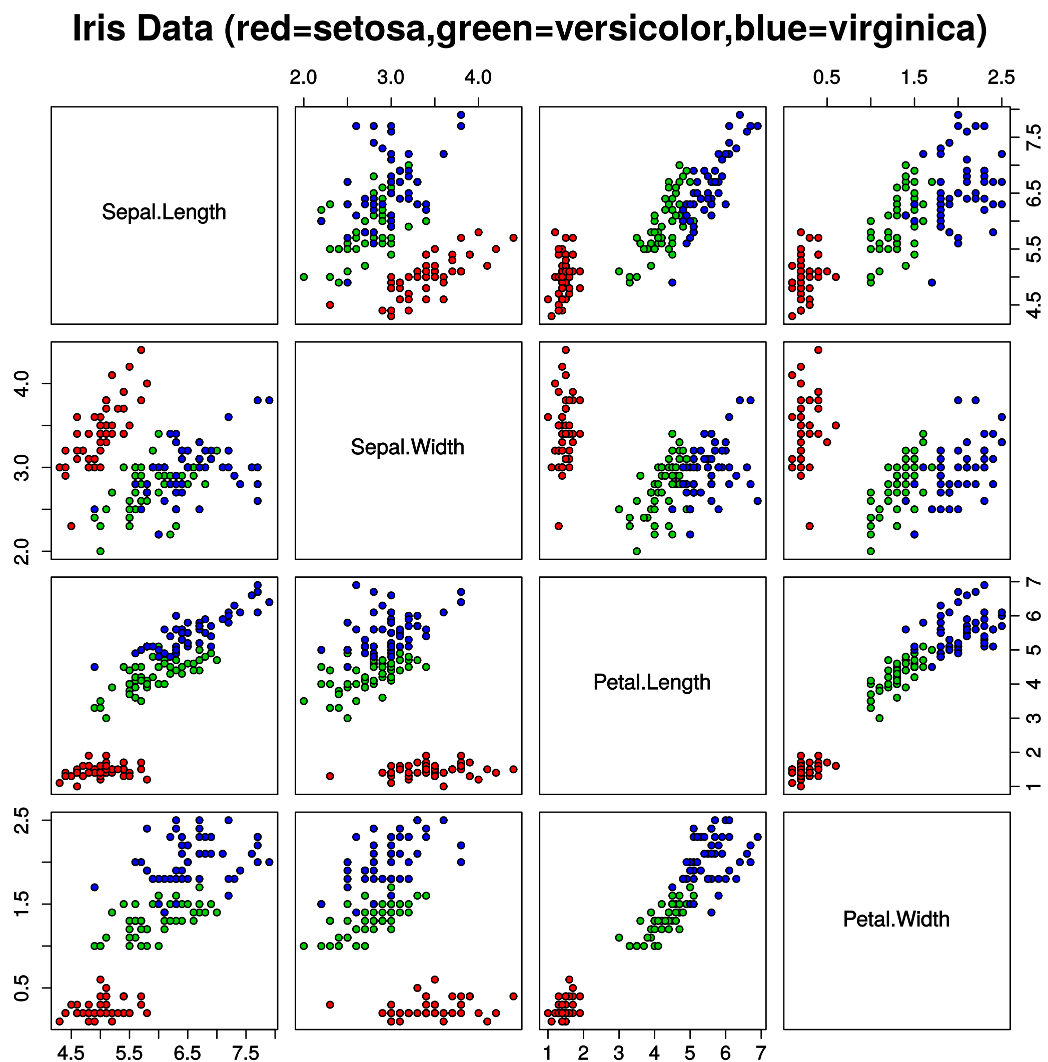

Figura 3.1: Rappresentazione di tutti i dati contenuti nell'Iris dataset [4].

In questo capitolo si testerà l'efficacia dell'algoritmo discesa del gradiente spiegato in precedenza sull'Iris dataset. Questo rappresenta un insieme di dati molto famoso per provare l'efficacia di nuovi algoritmi basati sulle reti neurali e consiste in tre classi differenti di fiori, che vengono classificate da 50 pattern ciascuna che contengono le quattro caratteristiche principali di questi fiori: lunghezza(cm) e larghezza(cm) di petalo e sepalo.

Come si vede chiaramente dalla Fig.3.1, tra le tre classi, solo una (Setosa) è linearmente separabile dalle altre (Versicolor e Virginica), che invece non risultano linearmente separabili. L'obbiettivo dunque sarà trovare un compromesso tra il numero di iterazioni dell'algoritmo (perché essendo non separabili linearmente, non si fermerebbe mai in modo autonomo) e la minimizzazione dell'errore di classificazione.

3.1 Inizializzazione dei dati

Come prima parte del codice si procede inizializzando i dati, ovvero facendone il *pre-processing*, che in questo caso comprende due parti principali:

```
1  from sklearn import datasets
2  import numpy as np
3
4  #load data
5  iris= datasets.load_iris()
6  shuffleMatrix=(np.column_stack((iris.data,iris.target)))
7  shuffleMatrix=np.take(shuffleMatrix,np.random.permutation(
       shuffleMatrix.shape[0]),axis=0,out=shuffleMatrix)
8
9  X=np.copy(shuffleMatrix[:,:4])
10 Y=np.copy(shuffleMatrix[:,4])
```

Codice 3.1: Pre-processing per mescolare la matrice di dati.

La prima parte (Codice 3.1) consiste nel caricare i dati dell'*Iris dataset* da una libreria Python chiamata *sklearn*, dopodichè è buona norma mescolare i dati in modo che l'algoritmo non sia dipendente dalla posizione dei dati, ma invece sia leggermente diverso per ogni volta che si esegue il programma. Infine si separa per comodita la matrice delle *feature* X dal vettore delle uscite Y.

```
#normalization
min,max = X.min(),X.max()
excursion=max-min
for i in range (len(X[0])):
    for j in range(len(X)):
        X[j,i] = (X[j,i]-min)/(excursion)

#SEPARATE TRAIN & TEST
X_train,Y_train=X[:100,:],Y[:100]
X_test,Y_test=X[100:,:],Y[100:]
```

Codice 3.2: Normalizzazione e separazione dei dati.

Il secondo passo (Codice 3.2) è quello di normalizzare i dati delle feature. Questa pratica viene fatta semplicemente per un fatto di prestazioni, ovvero per il semplice motivo che nella funzione di attivazione sigmoidale che si userà negli algoritmi seguenti è presente un termine esponenziale che si comporta in maniera migliore con valori "piccoli". Per questo motivo si fa una normalizzazione dei valori da 0 ad 1 definita come:

$$X[j,i] = \frac{X[j,i] - \min(X)}{\max(X) - \min(X)} \tag{3.1}$$

Dopo aver mescolato e normalizzato i dati, si dividono in due diverse variabili: i primi 100 elementi saranno quelli che serviranno all'esercitazione della rete e quindi al calcolo dei pesi utilizzando l'algoritmo della discesa del gradiente, ed i restanti 50 saranno usati per fare i test sulla rete allenata in precedenza per ricavarne i risultati e le conclusioni.

3.2 Analisi del codice

L'algoritmo sviluppato per applicare la tecnica della discesa del gradiente riceve i parametri di *learning rate* (che servirà per calcolare il parametro di *momentum*), e del numero di iterazioni che sono scelti in maniera arbitraria. Oltre a questi riceve le due classi (*target*) sulle quali applicare questo algoritmo, perché non è applicabile su tutte e tre le classi contemporaneamente.

Questo è dovuto al fatto che si divide lo spazio di decisione in 2 regioni di decisione separate da una linea. Si può capire come sia impossibile implementare un algoritmo che abbia 3 regioni di decisione in un unica volta.

```
def gd(lrate,iter,firstTarget,lastTarget):
    N=len(X_target)
    w=np.zeros((X_target.shape[1],1))
    b=0
    #function to minimize
    costs=[]
    for i in range(iter):
        momentumLearning=lrate*(iter-i)
        for j in range(N):
            XX=X_target[j,:]
            YY=Y_target[j]
            #w1x1+w2x2+w3x3+w4x4+w0
            Z=np.dot(w.T,XX.T)+b
            #Applicazione della funzione di attivazione
            Y_pred=sigm(Z)

            #Variazione dei pesi e del bias e errore
            db=Y_pred-YY
            dw= np.multiply(XX,db).reshape((4,1))
            cost+=pow(np.absolute(db),2)
            #Aggiornamento del vettore pesi e bias
            w= w-momentumLearning*dw
            b= b-momentumLearning*db
        costs.append(cost/N)
    w_final=np.squeeze(np.asarray(w))
    b_final=b[0]
    return(w_final,b_final,costs)
```

Codice 3.3: Algoritmo della discesa del gradiente.

Questo algoritmo agisce iterando tra i *pattern* uno per volta e provando a farne la combinazione lineare con i pesi (che vengono inizializzati a zero senza perdita di generalità). Dopo di questo, nel caso la classificazione sia errata verrano aggiornati i vettori dei pesi e del bias con coefficiente il parametro di *momentum*, e nel caso la classificazione sia esatta invece non verrà cambiato nulla dato che risulterebbe una variazione pari a zero.

Il parametro di *momentum* viene calcolato in modo da avere un massimo nella prima iterazione in cui vale 1, ed avere un minimo nell'ultima iterazione dove vale il valore che si è definito di *learning rate*.

Il vettore *costs* rappresenta la funzione di costo dell'algoritmo. Essa non è altro che la funzione d'errore di [2.12]. Viene calcolata dunque la funzione d'errore per ogni coppia di classi per poi valutarne l'efficacia.

Data la scelta di questo algoritmo, bisogna suddividere il nostro problema iniziale in 3 sotto-problemi di classificazione binaria, poi mediarne il risultato ed otterere cosi la classificazione di più classi con la strategia del *One-vs-One Multiclass Classification* [6].

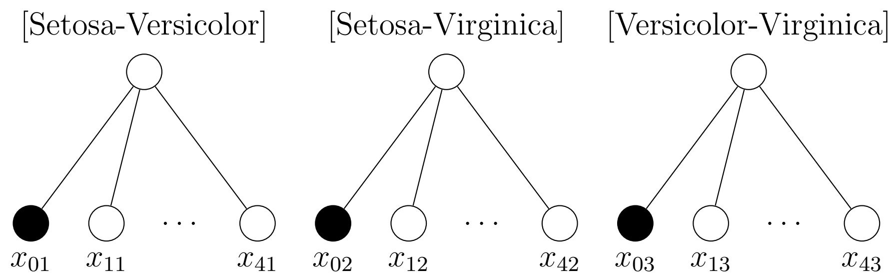

Figura 3.2: Suddivisione della rete in sottoproblemi.

Si capisce intuitivamente quindi che basterà ripetere 3 volte il Codice 3.3 passando i diversi target per ottenere i diversi pesi e bias con la tecnica della discesa del gradiente esposta nelle precedenti sezioni.

Per quanto riguarda la fase di test dei pesi e dei bias e verificare la correttezza di previsione dell'algoritmo si implementa questa funzione:

```
def predict(w,b,X):
    N=len(w)
    res=[]
    for i in range(N):
        wi=w[i,:]
        bi=b[i]
        Z=np.dot(wi,X)+bi
        #Yk=g(z)    yk=g(ak)
        res.append(sigm(Z))
    results=(np.sum(res))/3

    if (results<0.33):
        return 0
    if results>=0.67:
        return 2
    if (results>=0.33 and results<0.66):
        return 1
    else:
        return 10
```

Codice 3.4: Algoritmo per fare previsioni su un dato input.

In quest'ultimo algoritmo si applica semplicemente la formula generale di combinazione lineare, e la funzione di attivazione scelta per l'algoritmo discesa del gradiente:

$$y = \mathrm{sigm}(a) \qquad a = w_0 + \sum_{i=1}^{n} x_n \cdot w_n \tag{3.2}$$

in questo caso però si verifica la classificazione del pattern su ogni set di pesi che abbiamo ottenuto e infine, facendo la media aritmetica sui risultati otteniamo un ipotetico spazio di decisione nel quale esistono 3 regioni di decisione equivalenti. Possiamo quindi dire che se il risultato della media è basso (< 0.33) la previsione apparterrà alla classe $\mathcal{C}_1$ ovvero *Iris Setosa*, se sarà alto (> 0.67) apparterrà alla classe $\mathcal{C}_3$ ovvero *Iris Virginica* e nei casi intermedi sarà *Iris Versicolor*.

Questa suddivisione arbitraria è stata fatta dopo uno studio del dataset, precisamente vedendo che, sarebbe possibile suddividere quest'ultimo in maniera lineare con 3 differenti classi se non fosse per qualche caso limite che si sovrappone nelle altre classi. Quindi questa divisione è un'approssimazione semplificativa del nostro insieme di dati, ma risulta valida nella maggior parte dei casi.

Infine come ultima parte del codice c'è la funzione che permette di trovare dove saranno le regioni di decisione per graficarle in seguito insieme all'insieme di dati:

```
1 def findPoint(x,w,b,target):
2     w_target=np.squeeze(np.asarray(w[target,:]))
3     bias_target=b[target]
4     return ((-bias_target - w_target[xplot]*x)/w_target[yplot])
```

Codice 3.5: Algoritmo per trovare il bordo della regione di decisione.

Per limitazioni grafiche si visualizzeranno i risultati con massimo due *feature* alla volta, quindi questa funzione, dato un punto x_1 della prima feature, inverte la formula della combinazione lineare per trovare il secondo punto x_2:

$$x_2 = \frac{-w_0 - (w_1 \cdot x_1)}{w_2} \tag{3.3}$$

Questa funzione viene ricavata dal fatto che il massimo livello di indecisione nella funzione sigmoidale $g(a)$ (Fig.2.2) si ha quando il valore a è 0. Così iterando più volte questa funzione si otterrà un insieme di punti che delimita le regioni di decisione.

Capitolo 4

Conclusioni

In conclusione, utilizzando il codice mostrato nella sezione precedente si riesce ad allenare la rete e tutti pesi che servono per tracciare i 3 differenti bordi di decisione. Prendendo due *feature* a caso tra le quattro disponibili (in questo caso sono state analizzate la larghezza del sepalo in confronto con la lunghezza del petalo) si otterrà un grafico del tipo:

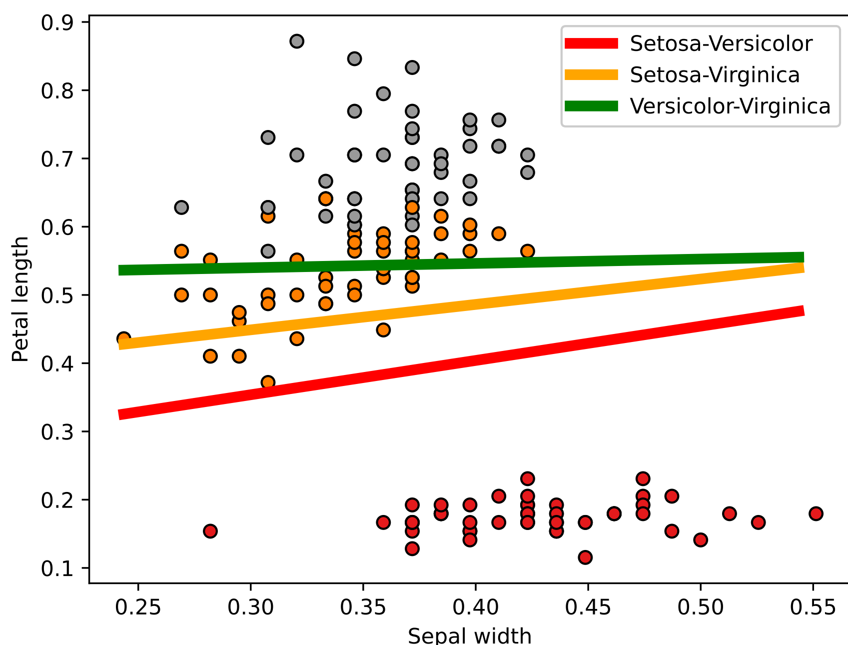

Figura 4.1: Esempio di regioni di decisione rispetto a due feature.

Dalla Fig.4.1 si può notare che gli elementi rossi (ovvero la specie *Setosa*) è stata separata perfettamente dalle altre due specie.

Come si era considerato in precedenza, questa specie è l'unica che risulta linearmente separabile dalle altre due. Per quanto riguarda le altre due specie si possono notare degli errori di classificazione e specialmente tra questi si nota che la specie arancione *Versicolor* è presente parecchie volte nella zona di classificazione degli elementi verdi *Virginica*.

Da questo modello, si può capirne la sua efficacia valutanto la **Confusion Matrix** che non è altro che la rappresentazione grafica di una matrice (in questo specifico caso sarà una matrice 3×3) che analizza i risultati del test che è stato sottoposto alla rete neurale:

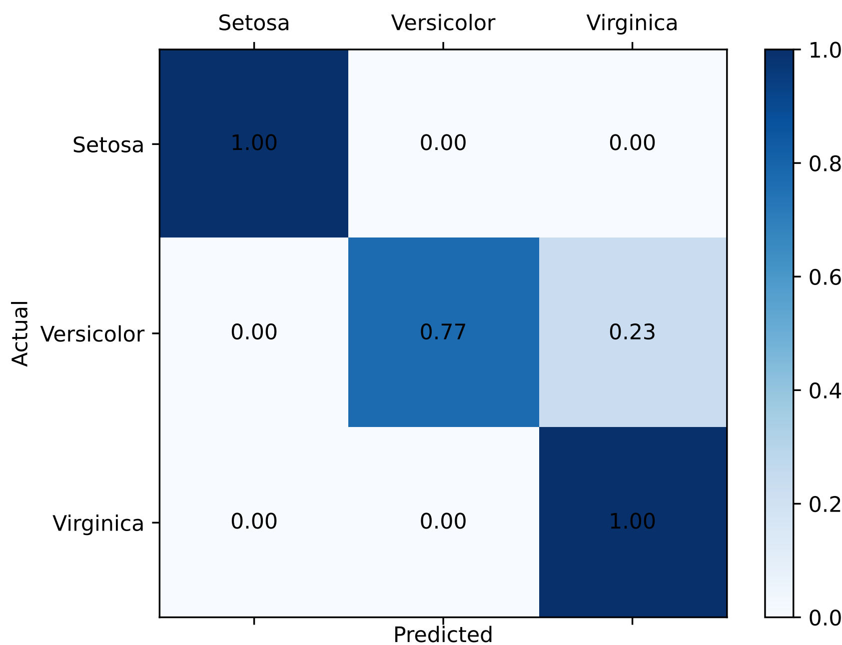

Figura 4.2: Confusion matrix

Questa matrice rappresenta sull'asse x le classificazioni predette dalla rete, e sull'asse delle y la classificazione esatta di queste ultime. Pertanto come ci si poteva aspettare dalla Fig.4.1 si avrà qualche errore di classificazione solo per la specie *Versicolor* che in qualche caso limite viene classificata erroneamente come *Virginica*. Il sistema ha un'accuratezza finale del 94% che significa che in media, con il vettore di test preparato nel codice si avrà una predizione di classificazione esatta da parte del sistema circa 47 volte su 50 (Codice 3.2).

Un'altro modo di valutare un sistema è quello di visualizzare l'andamento della funzione d'errore. Questa funzione, descritta in [2.12], calcola per ogni iterazione il valor medio tra tutti i pattern delle classificazioni errate.

Si avrà così il grafico della funzione d'errore, calcolato nelle iterazioni dell'algoritomo di allenamento della rete, come:

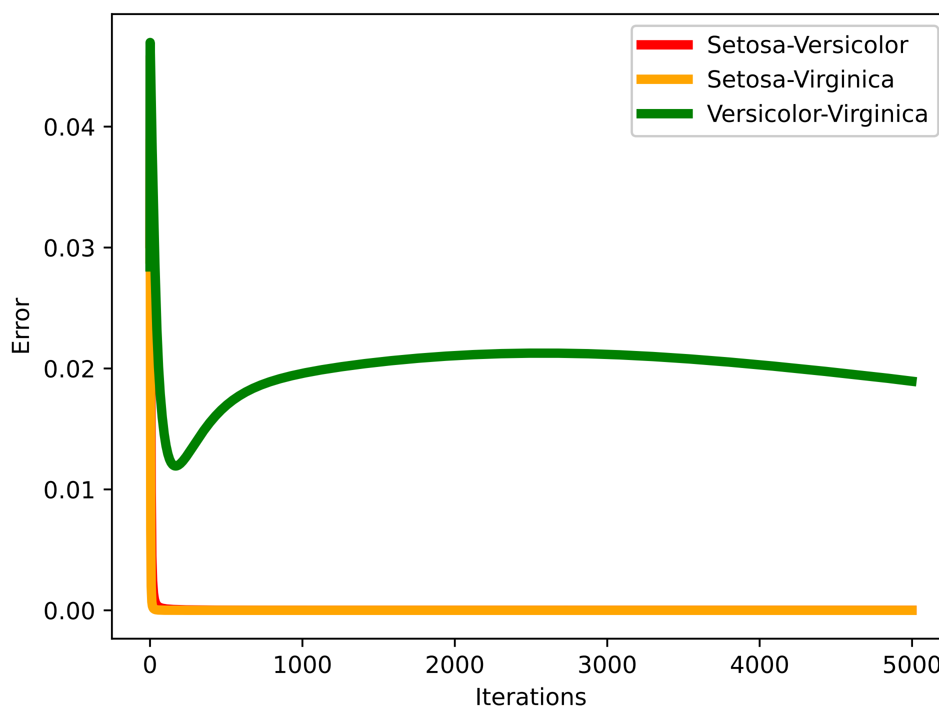

Figura 4.3: Andamento della funzione d'errore in 5000 iterazioni.

In questo grafico si nota che gli errori delle coppie gialla e rossa (ovvero quelli che comprendono la specie *Setosa*) tendono a zero dopo pochissime iterazioni. Questo è dovuto al fatto che, come detto in precedenza, questa specie è linearmente separabile dalle altre quindi dopo poche iterazioni si stabilizzano i bordi delle regioni di decisione e l'errore di classificazione si azzera.

Per quanto riguarda il grafico della funzione verde (della coppia di specie non linearmente separabili) si nota un minimo dopo esattamente 168 iterazioni. Dopo un numero così basso di iterazioni l'algoritmo non è ancora affidabile perchè, grazie al uso del parametro di *momentum* si capisce che questo è un minimo 'fittizzio', poichè il passo di variazione dei pesi è in continuo decremento quindi ci si aspetta una funzione monotona senza minimi/massimi locali.

Si può provare l'ultima frase facendo allenare il sistema per esattamente quel breve numero di iterazioni, e visualizzandone la confusion matrix:

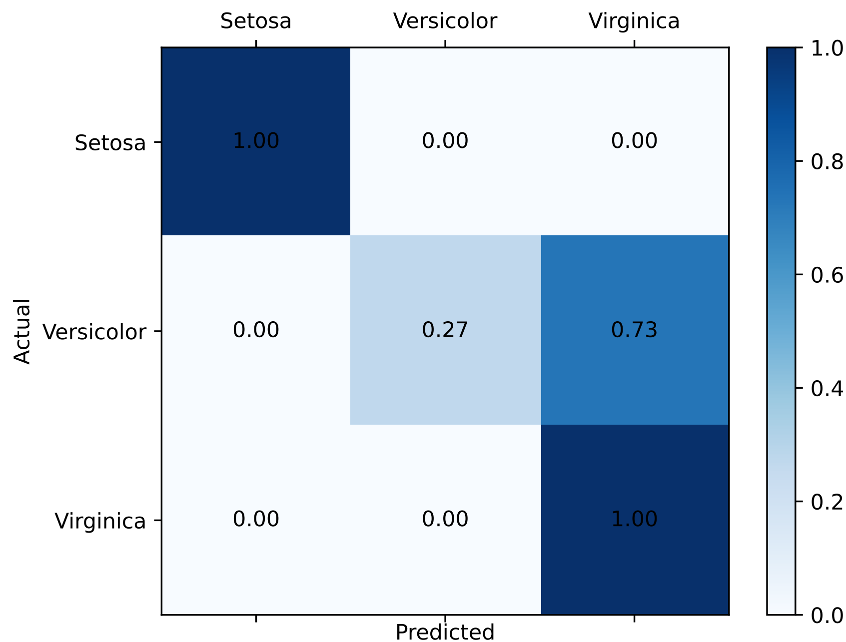

Figura 4.4: Confusion matrix

Allenando dunque il sistema per un numero cosi ristretto di iterazioni si ottiene una classificazione quasi sempre errata della seconda specie, abbassando l'accuratezza del sistema al 78%.

4.1 Miglioramenti e possibili estensioni

Le reti di questa tipologia, ovvero a singolo-strato, hanno la grossa limitazione di poter classificare al meglio solo dati che sono linearmente separabili tra di loro. Nei casi pratici non si ha quasi mai questa condizione, come si è esaminato nel caso dell'*Iris dataset*, per questo l'efficienza e l'accuratezza del sistema non potrà mai essere massima.

Per questo motivo negli anni sono stati fatti studi su come migliorare le capacità predittive di queste reti, arrivando a studiare le reti **multi-strato**, che non sono altro che reti a singolo strato con più passi intermedi e più set di pesi. Queste reti hanno la peculiarità di poter avere delle curve di separazione di qualsiasi forma, togliendo così il vincolo dell'obbligo di avere dati che devono essere linearmente separabili.

Tenendo sempre come esempio il caso del *Iris dataset*, si nota dalla figura successiva che quando esso viene analizzato da una rete multi strato le regioni di decisione hanno bordi arbitrari, facendo in modo cosi che anche i casi limite di ogni specie siano classificati in maniera esatta.

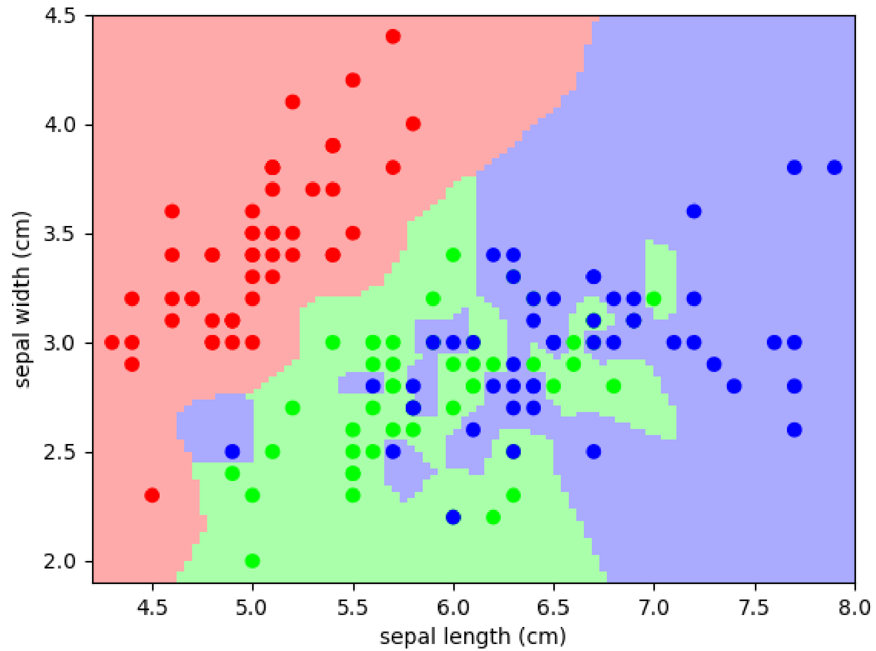

Figura 4.5: Curva di separazione di una rete multi-strato [7].

Questa tipologia di rete è ovviamente più dispendiosa rispetto alle reti a singolo-strato per via di calcoli da eseguire; ma a suo favore restituisce una classificazione quasi perfetta per qualsiasi insieme di dati se tarata in maniera ottimale.

Bibliografia

[1] Christopher M. Bishop, *Neural Network for Pattern Recognition.*

[2] *Rasoio di Occam, Wikipedia*, https://it.wikipedia.org/wiki/Rasoio_di_Occam, ultima consultazione: 03/12/2021.

[3] Rosenblatt, 1962; Block, 1962; Nilsson, 1965; Minsky e Papert, 1969; Duda e Hart, 1973; Hand, 1981; Arbib, 1987; Hertz, 1991; *Perceptron Convergence Theorem.*

[4] Di Nicoguaro, *Opera propria, CC BY 4.0,*, https://commons.wikimedia.org/w/index.php?curid=46257808, ultima consultazione: 10/12/2021.

[5] Kenneth J. McGarry, John Tait, Stefan Wermter, John MacIntyre, *Rule-Extraction from Radial Basis Function Networks*, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.127.8416&rep=rep1&type=pdf, ultima consultazione: 10/12/2021.

[6] *One-vs-One for Multiclass Classification*, https://machinelearningmastery.com/one-vs-rest-and-one-vs-one-for-multi-class-classification/, ultima consultazione: 13/12/21.

[7] *Multi-Layer plot of Iris Dataset*, https://scipy-lectures.org/packages/scikit-learn/auto_examples/plot_iris_knn.html, ultima consultazione: 20/12/21.

[8] *A biological and an artificial neuron*, https://www.quora.com/What-is-the-differences-between-artificial-neural-network-computer-science-and-biological-neural-network, ultima consultazione: 20/12/21.