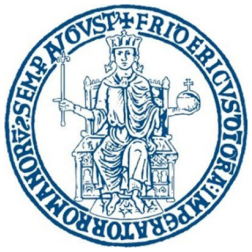
UNIVERSITÀ DEGLI STUDI DI NAPOLI
"FEDERICO II"



Scuola Politecnica e delle Scienze di Base

Area Didattica di Scienze Matematiche Fisiche e Naturali

Dipartimento di Fisica "Ettore Pancini"

Laurea Triennale in Fisica

Apprendimento di Reti Neurali attraverso la Particle Swarm Optimization

Relatori:
Dott.ssa Autilia Vitiello

Candidato:
Daniele Petrillo
Matr. $N85001328$

Anno Accademico 2020/2021

Indice

Introduzione **3**

1 Machine Learning **4**
1.1 Paradigmi di ML 4
1.2 Finalità del SL 4
1.2.1 Classificazione 4
1.2.2 Regressione 5
1.3 Workflow di algoritmo ML 6
1.3.1 Preprocessing 6
1.3.2 Allenamento 7
1.3.3 Valutazione 7

2 Reti Neurali Artificiali **8**
2.1 Neurone Artificiale 9
2.1.1 Funzioni di Attivazione 10
2.2 Apprendimento 12
2.2.1 Discesa del gradiente 12
2.3 Multi Layer Perceptron 14
2.3.1 Forward Propagation 16
2.3.2 Backward Propagation 17

3 Swarm Intelligence **20**
3.1 PSO 20
3.1.1 Basic PSO 21
3.1.2 Visualizzazione Geometrica 24
3.1.3 Limitazione delle Velocità e fattore di inerzia 25
3.1.4 Condizioni di arresto 26

4 Applicazione di PSO alle ANN **28**
4.1 Allenamento tramite PSO 28
4.1.1 Spazio di ricerca 28
4.1.2 Funzione di fitness 29
4.1.3 Termine dell'allenamento 30
4.2 Strumenti utilizzati 30

5 Esperimenti e risultati **32**
5.1 Workflow 32
5.2 Preprocessing 33
5.2.1 Dataset 33
5.2.2 Normalizzazione 34
5.2.3 Iperparametri Utilizzati 34
5.3 Risultati 35

Conclusione **38**

Introduzione

Da quando nel 1939 Alan Turing ideò la "Bomba Inglese", uno degli antenati più importanti del moderno computer, per decrittare messaggi cifrati tedeschi durante la seconda guerra mondiale, l'uomo si è spinto sempre più oltre nell' innovazione tecnologica affascinato dall'idea che un sistema automatico possa analizzare e comprendere informazioni al pari degli esseri umani.

Nel 1956 nasce l'Intelligenza Artificiale, disciplina che studia i fondamenti teorici, le metodologie e le tecniche che consentono di progettare sistemi hardware e software in grado di emulare i comportamenti cognitivi umani. Tre anni più tardi, il termine Machine Learning viene coniato dall'informatico Arthur Samuel, riferendosi ad una forma di Intelligenza Artificiale capace di apprendere conoscenza in modo autonomo e formulare soluzioni o eseguire azioni nei confronti di un problema specifico.

Le Reti Neurali Artificiali sono tra i più conosciuti algoritmi di Machine Learning, strutture complesse di neuroni artificiali interconnessi che, insieme, costituiscono un individuo in grado di imparare e prendere decisioni. Similarmente a ciò che accade per gli esseri viventi, una Rete Neurale deve allenarsi per divenire capace di compiere azioni correttamente. Negli anni sono stati formulati numerosi algoritmi di apprendimento, alcuni dei quali sono ormai considerati standard, tuttavia, la ricerca di metodi alternativi è sempre attiva.

Nel tentativo di ideare algoritmi di ottimizzazione capaci, tra le altre cose, di allenare Reti Neurali, è nata, nel 1989 la Swarm Intelligence: una famiglia di algoritmi ispirati al comportamento sociale di gruppi di animali. Tra questi vi è quello di Particle Swarm Optimization (PSO), ispirato al volo di stormi di uccelli.

In questo lavoro di tesi è stato realizzato ed utilizzato un algoritmo di Particle Swarm Optimization come metodo di allenamento di una rete neurale, confrontando le sue prestazioni con il metodo di BackPropagation.

Nel Capitolo 1 si riassume brevemente il concetto di Machine Learning, descrivendone tipologie, finalità e procedimenti.

Nel Capitolo 2 sono descritte in dettaglio le Reti Neurali Artificiali.

Nel Capitolo 3 è spiegato il significato di Swarm Intelligence e descitto nel dettaglio l'algoritmo di Particle Swarm Optimization.

Nel Capitolo 4 è illustrato il processo di applicazione del PSO all' apprendimento di Reti Neurali.

Nel Capitolo 5 sono infine descritte le modalità dell'esperimento e i risultati ottenuti.

1 Machine Learning

Negli ultimi tempi, la crescita inarrestabile di dati (strutturati o non) ha portato alla necessità di nuovi metodi in grado di gestirli e valutarli. Il **Machine Learning** (**ML**) si è proposto come soluzione al problema, evolvendosi come branca dell'**Intelligenza Artificiale** (**IA**), la quale sfrutta algoritmi automatici che acquisiscono conoscenza dai dati e la usano per effettuare azioni o predizioni.

Il Machine Learning ha rappresentato una grande novità nell'ambito delle IA, poiché non necessita di esseri umani che ricavino manualmente regole e costruiscano modelli per l'analisi dei dati; al contrario, gli algoritmi di ML propongono un modo efficace per acquisire autonomamente conoscenza ed aumentare gradualmente le proprie performance.

1.1 Paradigmi di ML

Esistono tre categorie principali di algoritmi di ML:

- **Supervised Learning** (**SL**): vengono mandati al modello come input una serie di dati (*istanze*) di training con *label*, essenzialmente etichette, che ne identificano la natura, stimolandolo a formulare una regola che attribuisca ad ogni dato il corrispondente label, nella previsione di utilizzare tale regola per la predizione di dati futuri di cui non si conosce la natura;
- **Reinforcement Learning** (**RL**): lo scopo è creare un modello (chiamato *agente*) che migliora le proprie performance interagendo con l'ambiente circostante. Il feedback positivo non è dato da label corretti (come per il SL) bensì da una *Reward Function*, la quale stabilisce una ricompensa come conseguenza di ogni azione dell'agente;
- **Unsupervised Learning** (**UL**): quando abbiamo a che fare con dati di cui non conosciamo label né struttura, si utilizzano tecniche di UL allo scopo di individuare pattern o sottogruppi (*cluster*) senza necessitare della guida di output conosciuti o funzioni di reward. Un'altra importante applicazione dell'UL è la riduzione della dimensionalità dei dati, grazie alla quale è possibile diminuire il tempo di esecuzione di un algoritmo, rimuovere rumori possibilmente dannosi per l' apprendimento e facilitare la visualizzazione dei dati.

1.2 Finalità del SL

Il mio lavoro di tesi si focalizza su approcci di **Supervised Machine Learning**, categoria che presenta due principali finalità per il modello: Classificazione e Regressione.

1.2.1 Classificazione

La **Classificazione** è uno dei sottogruppi del SL, il cui obiettivo consiste nel predire label di nuove istanze basandosi su osservazioni precedenti (effettuate

durante l'allenamento).

Ogni istanza appartiene ad una determinata *classe*, identificata dal proprio label; lo scopo del modello è predire il corretto label di ogni nuova istanza ad esso presentata, in modo tale da individuare la classe a cui appartiene senza l'aiuto di un essere umano.

La classificazione può essere *binaria* (Figura 1), se l'algoritmo deve distinguere unicamente due classi (e.g. stabilire se un tumore al seno sia maligno o benigno oppure se un paziente sia diabetico). Tuttavia l'insieme di label di classe può anche possedere una natura non binaria (*multiclasse*), in tal caso il modello deve distinguere tra più di due classi ed assegnare l'istanza a solo una di esse; ne è un esempio la classificazione della famiglia di fiori *Iris*, che contiene diverse specie tra cui *Setosa*, *Versicolor* e *Virginica*, le quali differiscono per attributi come lunghezza e larghezza sepale o petale; l'algoritmo prenderà quindi in input questi attributi e dovrà capire a quale delle tre specie appartiene il fiore in questione.

1.2.2 Regressione

La **Regressione** è il secondo tipo più importante di SL; mentre nei problemi di classificazione lo scopo è quello di individuare la classe di appartenenza di una istanza, in quelli di regressione non esistono label o classi, bensì l'obiettivo consiste nel predire valori di output continui.

Nell'analisi di Regressione viene dato in input un numero di variabili predittive (*explanatory*) ed una variabile continua di risposta (*outcome* o *target*), ed il modello prova ad individuare una relazione tra di esse.

Un modo per spiegare in semplici termini la regressione è il seguente: immaginiamo di avere una variabile di feature ed una di target e di posizionare le coppie corrispondenti di punti (feature-target) in un diagramma cartesiano. L'obiettivo del modello è disegnare una curva che minimizzi una certa metrica, di solito la distanza tra ogni punto e la curva stessa (vedi Figura 1). Una volta costruita questa curva durante la fase di allenamento, se ne possono verificare l'efficacia e la capacità di generalizzazione testando la sua risposta a nuove istanze di cui conosciamo il target o ad altre di cui non lo conosciamo.

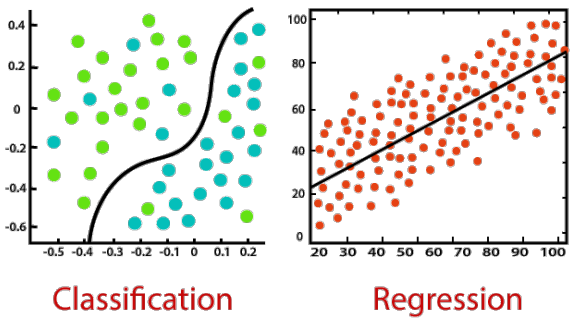


Figura 1: Esempi di classificazione e regressione [19]

1.3 Workflow di algoritmo ML

Praticamente ogni sistema di Machine Learning segue lo stesso workflow, illustrato dal seguente diagramma:

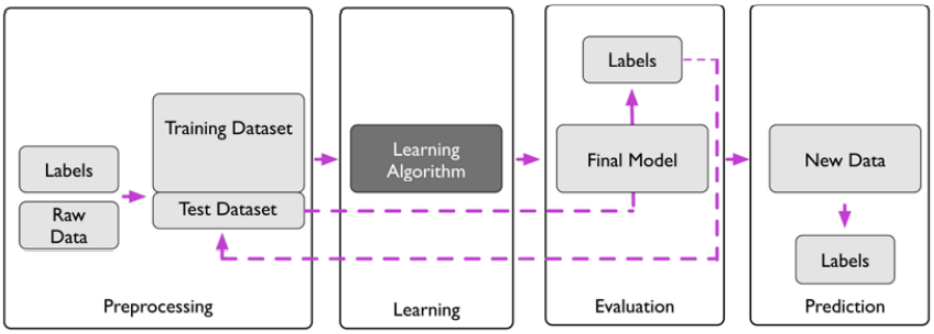


Figura 2: WF di un algoritmo di ML [13]

Le quattro fasi principali sono:

1. Preprocessing;
2. Allenamento sul train set;
3. Valutazione sul test set;
4. Predizione di dati ignoti.

1.3.1 Preprocessing

Uno degli step più importanti, solitamente anche quello che richiede più tempo, è il **Preprocessing**, cioè la preparazione dei dati prima di "darli in pasto" al modello. Spesso i dati sono raccolti da sensori o arrivano sotto forma di immagini, e non è possibile utilizzarli nella loro forma originale; in tutte le applicazioni di ML a sistemi di classificazione visiva (foto, video, etc.) i dati raccolti sono immagini di una certa risoluzione, indicata dal numero di pixel utilizzati dalla fotocamera per registrare il soggetto. Da queste immagini (contando i pixel) si possono ricavare attributi quali, ad esempio, l'altezza di una persona, il colore del petalo di un fiore o la forma di un ideogramma, in molti casi queste saranno le feature mandate al modello in input durante l'allenamento, in altri ci potrebbe essere la necessità di inviare le intere immagini (sempre trasformate in forma di vettore unidimensionale), accettando, però, un allenamento più lento e complesso.

Altri due aspetti ormai largamente considerati fondamentali per la corretta creazione di un modello sono la *normalizzazione* delle feature (nell'intervallo [0,1]) e la riduzione della *dimensionalità*, entrambe procedure dimostrate molto efficaci nel migliorare le performance, diminuire la quantità di memoria fisica necessaria e aumentare la velocità di allenamento ed esecuzione del modello.

Infine c'è bisogno di valutare se l'algoritmo è in grado di generalizzare ciò che ha imparato a dati mai visti; per fare ciò si divide casualmente il dataset in due insiemi: il training set (usato per l'allenamento) ed il testing set, su cui viene testato il modello dopo l'apprendimento.

1.3.2 Allenamento

Durante l'allenamento, il modello sfrutta algoritmi di apprendimento per aggiornare i propri parametri in modo da aumentare la propria performance nella predizione dei dati. In ciò è fondamentale la scelta della metrica da utilizzare per determinare se e quanto buoni sono i suoi risultati; una metrica molto utilizzata per la classificazione è l'*accuratezza* (*accuracy score*), la quale è definita come la porzione (o percentuale) di istanze classificate correttamente.

Oltre ai parametri modificati automaticamente durante l'allenamento esistono gli iperparametri, valori fissi e imposti a priori dal costruttore dell'algoritmo; nella maggior parte dei casi sono determinati in maniera euristica o ripresi dalla letteratura, ma talvolta sono il risultato di algoritmi (esterni) di ottimizzazione.

1.3.3 Valutazione

Dopo aver allenato il modello sul training set, possiamo utilizzare il test set per valutarne la capacità di generalizzazione a dati nuovi e mai visti; nel caso di un riscontro positivo su quest'ultimo, il modello può considerarsi pronto per l'applicazione a problemi reali.

È importante sottolineare che i parametri utilizzati per la normalizzazione e la riduzione di dimensionalità sono ricavati unicamente dal train set e successivamente utilizzati per la modulazione di entrambi gli insiemi e di dati futuri; questo perché, in generale, non ci è dato sapere che forma abbia la distribuzione dei dati che non abbiamo ancora prelevato (quelli su cui andremo a testare il modello una volta allenato), dunque il test set (che a tutti gli effetti li simula) non può essere determinante nel calcolo di suddetti valori.

2 Reti Neurali Artificiali

Il cervello è composto da centinaia di miliardi di cellule chiamate *neuroni*, formate da: un *nucleo*, una zona (il *corpo*) ad esso circoscritta ricca di *dendriti* (piccole protuberanze) e una lunga protuberanza che prende il nome di *assone*, come illustrato in Figura 3.

La grande forza del cervello, però, non è l'elevato numero di neuroni, bensì l'enorme e complesso sistema di connessioni tra di essi, realizzate attraverso il collegamento tra l'assone di un neurone ed un dendrite di un altro; le connessioni tra neuroni, che si aggirano intorno alle 60 trilioni di unità, prendono il nome di *sinapsi*, e rappresentano il motivo per il quale il cervello umano è ben superiore a qualsiasi macchina finora costruita, donandogli un'incredibile capacità di calcolo parallelo e differenziazione di mansioni.

Ogni segnale si propaga dai dendriti, attraverso il corpo della cellula fino all'assone e al nuovo neurone; tuttavia qualsiasi informazione può essere trasmessa solo se la cellula si "accende": possiamo immaginare il neurone come un interruttore che, in conduzione, trasmette il segnale, mentre, se aperto, lo blocca.

I meccanismi che determinano l'accensione sono nella realtà complessi e ignoti, mentre, nel caso di neuroni simulati, il ruolo è affidato a funzioni dette di *attivazione*.

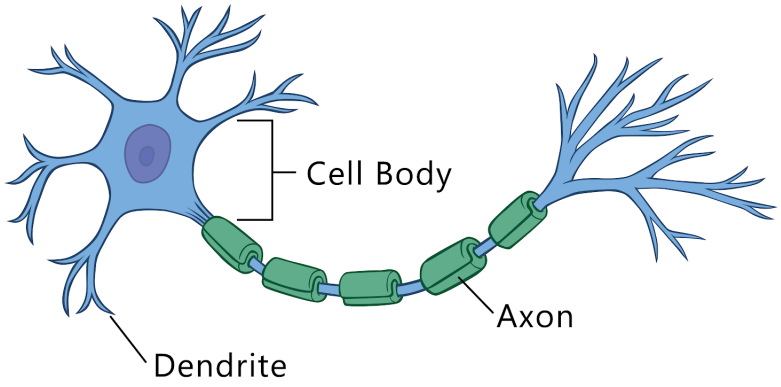


Figura 3: Schematizzazione di un neurone biologico [20]

Introduzione

Le **Reti Neurali Artificiali** (in inglese **Artificial Neural Network** o **ANN**) sono algoritmi di machine learning ispirati alle strutture neuronali del cervello.

Le ANN si sono dimostrate in più occasioni estremamente efficaci nella soluzione di particolari problemi, tra cui i più importanti sono:

- *Classificazione*;
- *Pattern Matching*: dove lo scopo è individuare un pattern associato ad un vettore di input;
- *Pattern Completion*: dove lo scopo è completare la parte mancante di un determinato vettore;
- *Ottimizzazione*;
- *Data Mining*: dove lo scopo è scoprire pattern nascosti in un gruppo (eventualmente molto ampio) di dati.

Una rete neurale è essenzialmente una mappa non lineare f_{NN} tra $\mathbb{R}^I \to \mathbb{R}^T$ dove I e T indicano rispettivamente la dimensione dello spazio di input e di target, ed f_{NN} è, in generale, una funzione composta di un insieme di funzioni non lineari, una per ogni neurone.

2.1 Neurone Artificiale

I primi ad ideare un neurone artificiale furono Warren McCullock e Walter Pitts, i quali, nel 1943, pubblicarono il primo studio su quello che fu successivamente denominato **McCullock-Pitts Neuron** (**MCP**) [1]. La loro idea era quella che il neurone fosse una semplice porta logica con output binario, dove la decisione della sua "accensione" era determinata dalla differenza tra la somma di tutti gli input ed un valore di soglia prestabilito.

Qualche anno più tardi, Frank Rosenblatt migliorò l' MCP donandogli la capacità di apprendere ed automigliorarsi, attraverso l'uso di un vettore di *pesi* che, moltiplicato scalarmente a quello di input, decide se accendere il neurone; questa nuova unità prese il nome di **Percettrone** [2].

In termini matematici un Percettrone equivale ad una mappa non lineare da:

$$\mathbb{R}^I \to [0, 1] \tag{2.1}$$

oppure:

$$\mathbb{R}^I \to [-1, 1] \tag{2.2}$$

dove I rappresenta la dimensione di input al neurone; questo vettore si presenta quindi nella forma:

$$\mathbf{x} = (x_1, \ldots, x_I) \tag{2.3}$$

e può provenire dall'ambiente circostante o da una collezione di output di un insieme di altri neuroni.

Il prodotto scalare tra il vettore dei pesi **w** e quello di input **x** prende il nome di *net input* (abbreviato *net*):

$$net = \sum_{i=1}^{I} x_i w_i \tag{2.4}$$

2.1.1 Funzioni di Attivazione

La **funzione di attivazione** del neurone artificiale f_P determina l'output mandato in avanti dall'unità, prendendo come input il *net* ed un valore aggiuntivo chiamato *bias* o *soglia*. Nella maggior parte dei casi, le funzioni di attivazione sono monotone crescenti con limiti all'infinito:

$$\lim_{x \to -\infty} f_P(x) = 0 \quad \text{oppure} \quad \lim_{x \to -\infty} f_P(x) = -1 \tag{2.5}$$

e:

$$\lim_{x \to \infty} f_P(x) = 1 \tag{2.6}$$

Nella formulazione originale del Percettrone la sua funzione di attivazione non era altro che una funzione a *gradino,* che prendeva in input il prodotto scalare tra il vettore dei pesi e quello degli input:

$$net = \mathbf{w}^T \mathbf{x} \tag{2.7}$$

e lo confrontava ad un valore di soglia θ:

$$\phi(net) = \begin{cases} 1 \quad se \quad net \geq \theta \\ -1 \quad (\text{oppure} \quad 0) \quad altrimenti \end{cases} \tag{2.8}$$

Questa tipologia di neurone si è però rivelata essere limitata e non adatta alla risoluzione di problemi non lineari. Nel 1960 Bernard Widrow e Tedd Hoff [3] pubblicarono un lavoro in cui veniva definita quella che poi sarebbe diventata l'unità standard al posto del Percettrone: l'**ADAptive LInear NEuron** (**Adaline**).

L'unica differenza tra lei ed il suo predecessore consisteva nella scelta di una funzione di attivazione lineare e non a scalino, in particolare venne scelta la funzione identità:

$$\phi(net) = net \tag{2.9}$$

Questa apparente piccola differenza aprì le porte ad un apprendimento basato sulla definizione e minimizzazione di funzioni di costo continue piuttosto che (come per il Percettrone) su aggiornamenti basati su differenze discrete tra output dell'unità e valore reale.

Nonostante la funzione di attivazione utilizzata per l'apprendimento sia diversa, Adaline fa ancora uso della funzione a gradino al momento della classificazione finale. La Figura 4 riassume le differenze tra i due tipi neuroni:

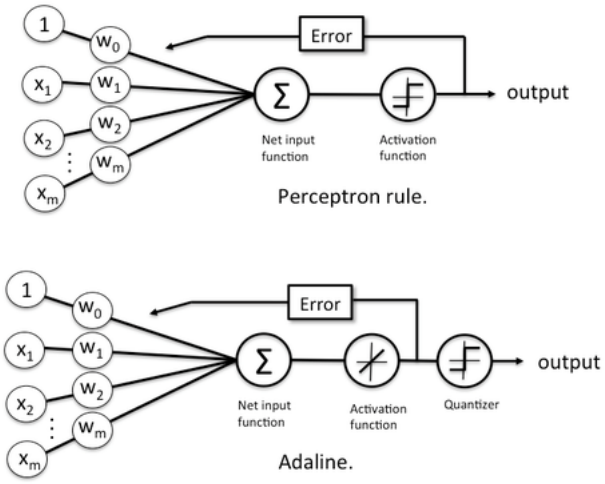


Figura 4: Confronto tra Percettrone e Adaline [22]

Di seguito, quando si parlerà di un neurone artificiale, ci si riferirà sempre ad un'unità Adaline.

Ad oggi sono state ideate ed implementate numerose funzioni di attivazione utilizzate a seconda del tipo di problema che si intende risolvere; tra le tante, una di quelle divenute "standard" prende il nome di **REctified Linear Unit** (**ReLU**) definita come la parte positiva del proprio argomento:

$$f(x) = x^+ = max(0, x) \tag{2.10}$$

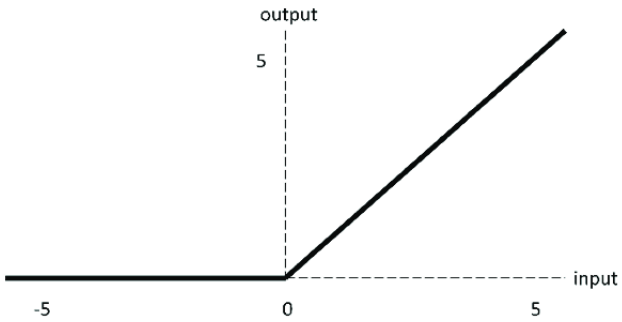


Figura 5: Funzione ReLU [21]

La funzione ReLU corrisponde alla naturale evoluzione della funzione identità, per cui è possibile scambiarle mutuamente (con piccoli accorgimenti) in maniera abbastanza naturale. ReLU si è dimostrata essere più performante

nell'allenamento delle reti neurali, se confrontata con le funzioni utilizzate in precedenza, e anche in grado di semplificare notevolmente i calcoli dei gradienti e diminuire i tempi di computazione.

2.2 Apprendimento

Finora abbiamo parlato delle caratteristiche di un Neurone Artificiale: i *pesi*, le *soglie* (*threshold*) e la funzione di *attivazione*, ma li abbiamo dati sempre per scontati e fissati nel tempo; la grande potenza di un neurone artificiale, però, è la sua capacità di migliorarsi e di imparare dai dati di input, modificando in modo dinamico i valori di *soglie* e *pesi* finché un determinato criterio di terminazione (o più di uno) non viene soddisfatto.

Nel caso di una rete neurale in *Supervised Learning* (come quella utilizzata in questo lavoro di tesi), ogni neurone ha lo scopo di adattare i propri *pesi* e le proprie *soglie* al fine di massimizzare la percentuale di successo di classificazione del modello, o, equivalentemente, minimizzare l'errore tra il suo output e quello (reale) di target.

Per semplificare la scrittura spesso si ridefinisce il vettore di input includendo, come valore $(I+1)$esimo il cosiddetto "valore di *bias*", posto sempre uguale a -1; inoltre, anche il vettore dei pesi viene modificato aggiungendo un'ultima componente w_{I+1} corrispondente al valore di soglia β.

In questo modo si rende la scrittura più pulita e sintetica:

$$\begin{aligned} z &= \sum_{i=1}^{I} x_i w_i - \beta \\ &= \sum_{i=1}^{I} x_i w_i + x_{I+1} w_{I+1} \\ &= \sum_{i=1}^{I+1} x_i w_i \end{aligned} \tag{2.11}$$

Ovviamente la funzione di attivazione dovrà essere riformulata per tenere conto della nuova definizione della sua variabile.

2.2.1 Discesa del gradiente

Uno dei processi più importanti nella definizione di un algoritmo di SL è la scelta di una cosiddetta **funzione obiettivo** da ottimizzare durante l'allenamento; nel caso di *minimizzazione* viene anche chiamata **funzione di costo**.

Tra le funzioni di costo più utilizzate c'è la **Sum of Squared Error** (**SSE**):

$$E(\mathbf{w}) = \frac{1}{2} \sum_i (y^i - \phi(z^i))^2 \tag{2.12}$$

dove ϕ è semplicemente la funzione identità (ma in generale rappresenta la funzione di attivazione scelta) e y^i è il *label* reale dell'istanza considerata (la

somma su i è da considerarsi come somma sull'insieme totale dei vettori di input, siano essi istanze del train set oppure provenienti da neuroni precedenti).

Aggiungere l'identità (al posto di una funzione a *gradino*) come funzione di attivazione permette alla funzione di costo di essere convessa ma sopratutto differenziabile, attributo cruciale nel processo di apprendimento tramite **Discesa del Gradiente** (**GD** dall'inglese).

Possiamo immaginare il concetto di GD come la effettiva discesa dal pendio di una montagna, fino al raggiungimento di una valle (il minimo della funzione), come illustrato in Figura 6:

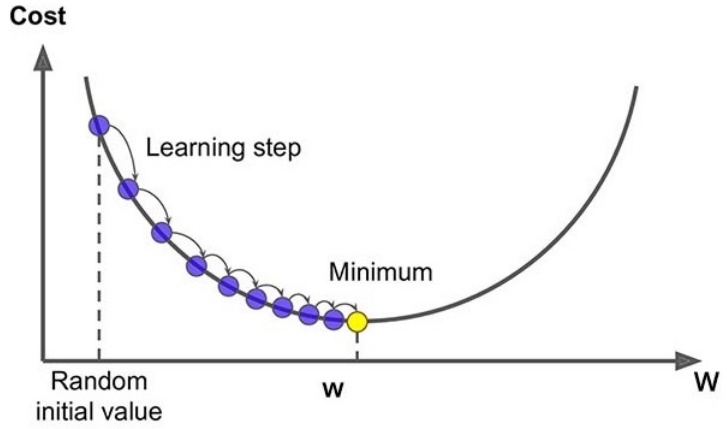


Figura 6: Discesa del gradiente [23]

Ad ogni step i pesi dell'unità sono aggiornati con in modo tale da "navigare" nel verso opposto rispetto al gradiente della funzione di costo. L'aggiornamento dei pesi corrisponde all'equazione:

$$\mathbf{w} := \mathbf{w} + \Delta\mathbf{w} \tag{2.13}$$

dove $\Delta\mathbf{w}$ è definito come:

$$\Delta\mathbf{w} = -\eta\nabla E(\mathbf{w}) \tag{2.14}$$

η è un iperparametro che prende il nome di *learning rate*; è molto importante ed il suo valore deve essere scelto con cura, questo perché:

- se troppo grande, tende a far compiere "balzi" eccessivamente ampi sulla funzione di costo, rischiando di saltare il minimo globale;
- se troppo piccolo, garantisce una migliore risoluzione a discapito della velocità di esecuzione.

Una via di mezzo tra i due casi è il giusto approccio per garantire un'efficacia ottimale.

La derivata parziale rispetto al peso $j$esimo vale:

$$\frac{\partial E}{\partial w_j} = -\sum_i (y^i - \phi(z^i))x_j^i \tag{2.15}$$

così che possiamo scrivere:

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j} = \eta \sum_i (y^i - \phi(z^i)) x_j^i \tag{2.16}$$

In totale:

$$\Delta \mathbf{w} = \eta \sum_i (y^i - \phi(z^i)) \mathbf{x}^i \tag{2.17}$$

In questa tecnica di apprendimento i pesi sono aggiornati tutti contemporaneamente basandosi sulla totalità dei vettori di input, per questo motivo viene anche chiamata **Batch Gradient Descent**. Una versione alternativa della discesa del gradiente è lo **Stochastic Gradient Descent** (**SGD**), dove, invece, l'aggiornamento dei pesi è effettuato in modo incrementale dopo ogni istanza:

$$\Delta \mathbf{w} = \eta (y^i - \phi(z^i)) \mathbf{x}^i \tag{2.18}$$

Tipicamente lo SGD raggiunge convergenza in meno tempo, poiché i pesi sono aggiornati con maggiore frequenza, inoltre, la superficie di errore contiene un rumore maggiore, consentendo all'algoritmo di uscire più agilmente dei minimi locali.

2.3 Multi Layer Perceptron

Sebbene il singolo neurone sia un buon strumento di classificazione, presenta dei difetti, tra cui la sua scarsa capacità di risolvere problemi complessi. Tuttavia, come sempre accade, l'unione fa la forza, ed è sufficiente mettere assieme anche solo poche unità in uno schema strutturato per notare grandi miglioramenti, nasce così il **Multi Layer Perceptron** (**MLP**).

L' MLP consiste in una rete *fully connected* suddivisa in livelli (*layer*), in cui tutti i neuroni appartenenti al medesimo layer sono connessi a tutti quelli nel successivo.

Contrariamente a ciò che suggerisce il nome, questo tipo di struttura non vincola la scelta del tipo di unità al semplice Percettrone, anzi, nella maggior parte dei casi, gli ingranaggi questa macchina virtuale sono neuroni come Adaline o più avanzati.

Per semplicità d'ora in poi tratteremo in modo esemplificativo una struttura a tre layer: denominati (in ordine) *input*, *hidden* e *output*, ma la teoria è applicabile anche a reti con un numero di layer interni superiore a due, in tal caso si parla di *deep learning*.

Chiamiamo a_i^l l'unità numero i del layer l. Trovandoci in una situazione semplice, d'ora in poi ci riferiremo ai layer di *input*, *hidden* e *output* rispettivamente con gli apici *in*, *h* e *out*.

In Figura 7 sono presenti anche le unità di *bias*, poste per convenzione uguali ad 1 (mentre saranno i pesi a variare). La regola sarà che ogni layer prima di quello di *output* avrà come primo neurone la cellula di bias, la quale sarà

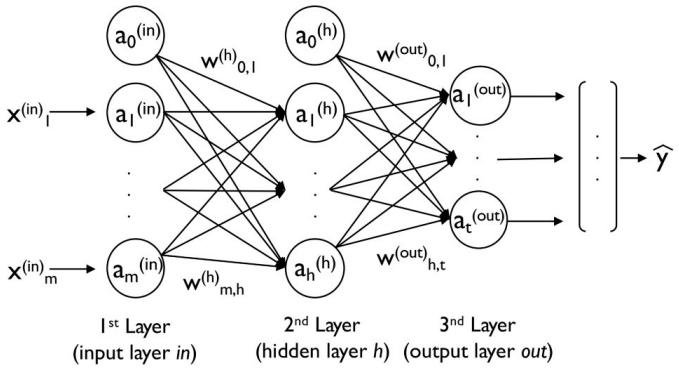


Figura 7: Schematizzazione MLP [13]

collegata a tutti i neuroni del layer successivo, escludendo le successive unità di bias. Il layer di *input* avrà quindi la forma:

$$\mathbf{a}^{in} = \begin{bmatrix} a_1^{in} \\ a_2^{in} \\ \vdots \\ a_m^{in} \end{bmatrix} = \begin{bmatrix} 1 \\ x_2^{in} \\ \vdots \\ x_m^{in} \end{bmatrix} \tag{2.19}$$

Ogni unità in un qualsiasi layer l è collegata a tutte quelle del layer successivo tramite un coefficiente (*peso*). In generale, il *peso* tra la $n$esima unità del layer l e la $m$esima del successivo si indica con $w_{n,m}^{l+1}$. Tornando ad una visione più ampia del sistema chiamiamo la matrice che collega il livello di input con quello di hidden $\mathbf{W}^h$ e quella che collega il livello hidden con quello di output $\mathbf{W}^{out}$.

Le dimensioni delle due matrici dipendono dal numero di neuroni in *input*, *hidden* ed *output*, infatti sarà $\mathbf{W}^h \in \mathbb{R}^{m \times h}$ dove m è il numero di neuroni in *input* e h quello dei neuroni in *hidden* (includendo in entrambi i casi il *bias*). Allo stesso modo l'altra matrice sarà $\mathbf{W}^{out} \in \mathbb{R}^{h \times t}$ con t numero di unità in output.

Il layer di *output* merita un breve approfondimento aggiuntivo. Nel caso di un singolo neurone l'unica tipologia possibile di classificazione è quella binaria; utilizzare un layer di output composto da molteplici neuroni ci permette di estendere il dominio della rete a problemi multiclasse, tramite la tecnica **One-Versus-All** (**OVA**), secondo la quale si effettua una classificazione binaria a coppie alterne per decidere in modo definitivo a quale delle classi appartiene una determinata istanza.

Disponendo di t neuroni nel layer di *output* pari al numero di classi distinte, il vettore di target contenente l'informazione sul label reale di un'istanza deve essere convertito in formato *One-Hot*, ponendo tutti i suoi elementi a 0 tranne quello la cui posizione corrisponde alla classe di appartenenza.

Prendendo come esempio il dataset *Iris* ed assegnando la seguente convenzione per le classi:

- $0 \longleftrightarrow$ *Setosa*
- $1 \longleftrightarrow$ *Versicolor*
- $2 \longleftrightarrow$ *Virginica*

le istanze possederanno un vettore di target scelto dall'insieme:

$$0 = \begin{bmatrix}1\\0\\0\end{bmatrix},\ 1 = \begin{bmatrix}0\\1\\0\end{bmatrix},\ 2 = \begin{bmatrix}0\\0\\1\end{bmatrix} \tag{2.20}$$

2.3.1 Forward Propagation

Un MLP prende in input un vettore di feature, lo processa internamente attraverso le matrici di peso e, infine, produce una risposta nel layer di *output*; il processo di analisi e trasmissione dell'input fino all'ultimo livello prende il nome di **Forward Propagation**, in cui ogni layer funge da input per quello successivo (si veda la Figura 8). Una rete neurale che fa uso di tale modalità prende il nome di **FeedForward Neural Network**.

Considerando il primo neurone (non *bias*) del livello *hidden*, il suo *net input* sarà:

$$z_1^h = a_0^{in} w_{0,1}^h + a_1^{in} w_{1,1}^h + \cdots + a_m^{in} w_{m,1}^h \tag{2.21}$$

da cui si ricava il valore dell'unità come risposta della funzione di attivazione:

$$a_1^h = \phi(z_1^h) \tag{2.22}$$

Riscrivendo in maniera più compatta:

$$\mathbf{z}^h = \mathbf{a}^{in}\mathbf{W}^h \tag{2.23}$$

$$\mathbf{a}^h = \phi(\mathbf{z}^h) \tag{2.24}$$

dove $\mathbf{a}^{in}$ è un vettore di attivazione $1 \times m$ definito in 2.19, mentre $\mathbf{W}^h$ è la solita matrice di pesi $m \times h$.

Dopo la moltiplicazione scalare vettore-matrice, riotteniamo un vettore $1 \times h$ (cioè $\mathbf{z}^h$) da cui ricaviamo $\mathbf{a}^h$. Generalizzando a tutti gli n vettori di input:

$$\mathbf{Z}^h = \mathbf{A}^{in}\mathbf{W}^h \tag{2.25}$$

Adesso $\mathbf{A}^{in}$ è una matrice $n \times m$ e $\mathbf{Z}^h$ una $n \times h$ così come $\mathbf{A}^h = \phi(\mathbf{Z}^h)$. Allo stesso modo si trova:

$$\mathbf{Z}^{out} = \mathbf{A}^h\mathbf{W}^{out}, \quad \mathbf{Z}^{out} \in \mathbb{R}^{n \times t} \tag{2.26}$$

$$\mathbf{A}^{out} = \phi(\mathbf{Z}^{out}), \quad \mathbf{A}^{out} \in \mathbb{R}^{n \times t} \tag{2.27}$$

2.3.2 Backward Propagation

Per l'allenamento di MLP di solito su utilizza una funzione di costo più complessa ed efficace della SSE, la **Logistic Cost Function** (**LCF**), che, nel caso di singolo neurone, assume la seguente forma:

$$J(\mathbf{w}) = -\sum_{i=1}^{n} y^i \log(a^i) + (1 - y^i) \log(1 - a^i) \tag{2.28}$$

dove a^i è l'attivazione dell'$i$esimo campione del training set. Al fine di limitare il fenomeno di *overfitting*, per il quale si tende ad adattare in modo eccessivo i pesi al train set, si aggiunge un termine di regolarizzazione L2:

$$L2 = \frac{\lambda}{2} \|\mathbf{w}\|_2^2 = \frac{\lambda}{2} \sum_{j=1}^{m} w_j^2 \tag{2.29}$$

Riformulando l'equazione precedente:

$$J(\mathbf{w}) = -\left[\sum_{i=1}^{n} y^i \log(a^i) + (1 - y^i) \log(1 - a^i) \right] + \frac{\lambda}{2} \|\mathbf{w}\|_2^2 \tag{2.30}$$

Nel caso di classificazione multiclasse il numero di neuroni nell'output layer è pari al numero di classi, per cui è necessario adattare la funzione di costo ed il termine L2 al caso di layer a più di un'unità:

$$J(\mathbf{w}) = -\left[\sum_{i=1}^{n} \sum_{j=1}^{t} y_j^i \log(a_j^i) + (1 - y_j^i) \log(1 - a_j^i) \right] + \frac{\lambda}{2} \sum_{l=1}^{L-1} \sum_{i=1}^{u_l} \sum_{j=1}^{u_{l+1}} (w_{j,i}^l)^2 \tag{2.31}$$

dove t è il numero di neuroni in output, L il numero di layer (in totale) e u_l quello di unità del layer l.

Come già visto l'allenamento corrisponde alla minimizzazione di tale funzione di costo $J(\mathbf{W})$, dove $\mathbf{W}$ è, stavolta, un tensore a tre dimensioni, che possiamo immaginare come un vettore di due elementi, pari rispettivamente alle matrici di pesi $\mathbf{W}^h$ e $\mathbf{W}^{out}$.

Mentre la predizione della classe di appartenenza di un input è eseguita tramite una propagazione *feedforward* (in avanti) del segnale, qualunque metodologia utilizzata ai fini dell'allenamento deve partire dal fondo, avendo come primo passo il confronto tra il valore di classe predetto dalla rete ed il valore reale di label.

Una volta stabilito l'errore di classificazione tramite la funzione di costo, l'informazione riguardante le modalità di aggiornamento dei pesi deve essere propagata in tutti i layer; è stato dimostrato che trasmettere l'informazione dal fondo della struttura alla cima è il metodo più efficace e che richiede meno sforzo computazionale, dovendo effettuare unicamente moltiplicazioni scalari matrice-vettore e non matrice-matrice (cosa che accadrebbe se ragionassimo analogamente al FF).

Per merito di questa modalità atipica "all'indietro", questo metodo di propagazione prende il nome di **Backward Propagation** o, ridotto, **Backpropagation** (**BP**) e rappresenta ormai la metodologia considerata globalmente standard per l'allenamento di reti neurali.

Il primo passo consiste nel *feedforward* della risposta generata dall'analisi di un vettore di input riassumibile nei passaggi:

1. $\mathbf{Z}^h = \mathbf{A}^{in}\mathbf{W}^h$
2. $\mathbf{A}^h = \phi(\mathbf{Z}^h)$
3. $\mathbf{Z}^{out} = \mathbf{A}^h\mathbf{W}^{out}$
4. $\mathbf{A}^{out} = \phi(\mathbf{Z}^{out})$

In sintesi, otteniamo un output trasmettendo in avanti le feature in input:

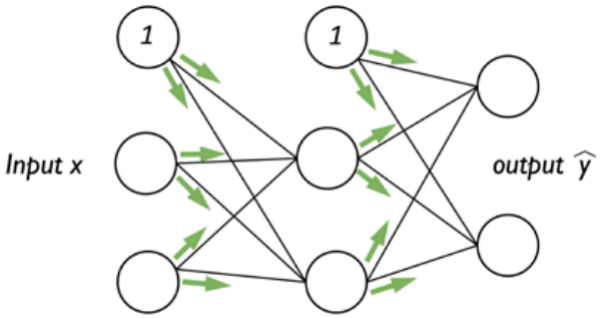


Figura 8: Forward Propagation [13]

Calcoliamo quindi il vettore di errore dell'ultimo layer:

$$\boldsymbol{\delta}^{out} = \mathbf{a}^{out} - \mathbf{y} \tag{2.32}$$

dove $\mathbf{y}$ è il vettore di label della classe reale. Da $\boldsymbol{\delta}^{out}$ ricavo l'errore nel layer interno:

$$\boldsymbol{\delta}^h = \boldsymbol{\delta}^{out}(\mathbf{W}^{out})^T \odot \frac{\partial\phi(z^h)}{\partial z^h} \tag{2.33}$$

dove ϕ è la solita funzione di attivazione e $\odot$ è il prodotto di *Hadamart*, che si esegue tra matrici (o vettori) di dimensioni uguali e restituisce un oggetto delle stesse dimensioni i cui elementi sono pari al prodotto di elementi che nelle due matrici di partenza occupano la medesima posizione.

Si può dimostrare che la derivata parziale della funzione di costo rispetto ad un peso rispettivamente della matrice di output e di quella interna equivale a:

$$\frac{\partial}{\partial w_{i,j}^{out}} J(\mathbf{W}) = a_j^h \delta_i^{out} \tag{2.34}$$

e:

$$\frac{\partial}{\partial w_{i,j}^{h}} J(\mathbf{W}) = a_j^{in} \delta_i^h \tag{2.35}$$

Ricordando che dobbiamo calcolare l'aggiornamento di pesi per ogni elemento del training set, riscriviamo i gradienti in forma vettoriale:

$$\mathbf{\Delta^h} = (\mathbf{A}^{in})^T \boldsymbol{\delta}^h \tag{2.36}$$

e:

$$\mathbf{\Delta}^{out} = (\mathbf{A}^{h})^T \boldsymbol{\delta}^{out} \tag{2.37}$$

Adesso possiamo aggiungere il termine di regolarizzazione (eccezion fatta per le unità di bias):

$$\mathbf{\Delta}^l := \mathbf{\Delta}^l + \boldsymbol{\lambda}^l \tag{2.38}$$

Per concludere aggiorniamo le matrici dei pesi andando in verso contrario al gradiente e moltiplicando per il *learning rate*:

$$\mathbf{W^l} := \mathbf{W}^l - \eta \mathbf{\Delta}^l \tag{2.39}$$

Il processo di *BackPropagation* è utilmente riassunto nella Figura 9:

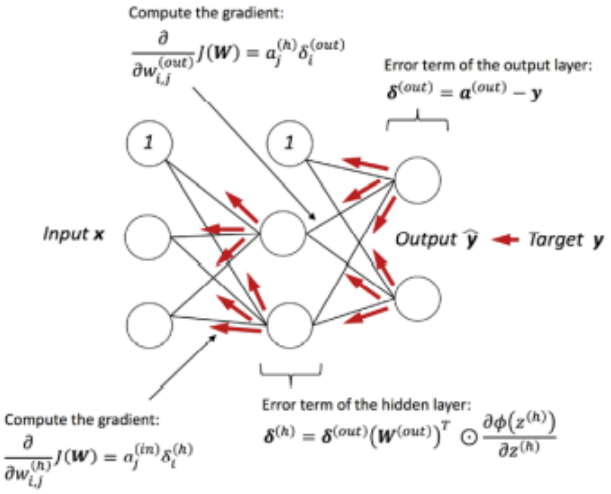


Figura 9: Schematizzazione BackPropagation [13]

3 Swarm Intelligence

Uno *sciame*, in inglese *swarm*, è definito come: "gruppo folto di animali alati di piccole dimensioni che volano insieme" [4]; in termini figurati, identifica un gruppo di individui semplici in continua interazione tra di loro e con l'ambiente circostante.

È possibile ammirare tali comportamenti in numerose specie animali tra cui formiche, api, uccelli e lucciole, specie che acquisiscono straordinarie capacità di esplorazione, costruzione e organizzazione sfruttando il semplice ma efficace potere della comunicazione. Se gli uccelli sono in grado di muoversi in modo coordinato o le formiche di individuare una fonte di cibo a centinaia di metri di distanza e ritornare al nido senza perdersi è sopratutto grazie alla costante comunicazione che ogni individuo sfrutta per ricevere nuove informazioni.

Le interazioni sociali della popolazione sono facilmente suddivisibili in due categorie:

- *diretta*, tramite stimoli visivi o, in generale, sensoriali: ne è un esempio il comportamento delle api operaie, le quali comunicano la posizione di una fonte di nettare mediante una specie di danza, denominata *Waggle Dance*;
- *indiretta*, attraverso una modifica dell'ambiente circostante: questo metodo di comunicazione prende il nome di *stigmergia*, e si può osservare nelle colonie di formiche o di termiti; sfruttando queste tracce è possibile coordinarsi per esplorare in modo sicuro le zone limitrofe in cerca di nutrimento (nel caso delle formiche) oppure costruire nidi di fango basati su complesse strutture come archi, pilastri, gallerie e camere (nel caso delle termiti).

Intorno alla fine degli anni '80, alcuni scienziati hanno cominciato ad accorgersi del potenziale di questi comportamenti collettivi e decentralizzati nell'ambito dell'Intelligenza Artificiale, finché, nel 1989, fu coniato da G. Beni e J. Wang il termine **Swarm Intelligence**(**SI**), ai tempi utilizzato per riferirsi ad una serie di algoritmi finalizzati al controllo di gruppi di robot [5]. Segue nel 1991 la pubblicazione dell'algoritmo di **Ant Colony Optimization** (**ACO**) ispirato al comportamento delle formiche [6] e, nel 1995, quello di **Particle Swarm Optimization** (**PSO**) nell'intento di replicare il movimento di stormi di uccelli alla ricerca di cibo [7].

Ad oggi sono stati ideati numerosi algoritmi ispirati alla natura dei gruppi sociali, i quali hanno dato prova del proprio valore nell'ambito di numerosi problemi di ottimizzazione e di ricerca come minimizzazione di funzioni, analisi di immagini o studio di percorsi ottimali, così come hanno trovato applicazione in diversi domini tra cui machine learning, bioinformatica, sistemi dinamici e finanza.

3.1 PSO

Il PSO è un algoritmo di ricerca *population-based* ispirato al movimento ed al comportamento di stormi di uccelli in volo. Sebbene l'intenzione originale

fosse quella di comprendere l'affascinante "danza sincronizzata" degli stormi, presto ci si rese conto che l'algoritmo aveva grandi potenzialità nei problemi di ottimizzazione [8].

L'idea di base è che ogni individuo, chiamato spesso anche *particella*, sia libero di esplorare liberamente il cosiddetto spazio di ricerca (spesso di un numero elevato di dimensioni), muovendosi nella direzione in cui ritiene sia locata la sua zona migliore, che nella realtà corrisponde ad un punto di minimo o di massimo di una **funzione obiettivo**.

La grande forza di questo algoritmo sta nel fatto che le decisioni (direzioni) prese da ogni individuo dipendono sia dall'esperienza personale che da quella collettiva, sfruttando al meglio il maggior numero di informazioni; questo comportamento dipendente dal gruppo fa sì che le particelle si muovano indipendentemente ma allo stesso tempo come una grande unità, una formula che si è rivelata vincente nella risoluzione di problemi di ottimizzazione e non solo.

3.1.1 Basic PSO

Il *basic* PSO è la versione più semplice e antica di PSO, trattandosi dell'algoritmo originale proposto da Eberhart e Kennedy. Ogni particella segue lo stesso comportamento di base, che si divide in due azioni:

- Ricordare e seguire i propri successi precedenti;
- Emulare il successo del resto del gruppo.

Lo scopo è quello di trovare le regioni ottimali (soluzioni nello spazio di ricerca) per il problema in esame; per farlo, si suddivide il tempo trascorso in maniera discreta (*epoche* o *iterazioni*) e, ad ogni iterazione, si aggiorna la posizione di ogni particella in base alla propria esperienza e a quella collettiva:

$$\mathbf{x}_i(t+1) = \mathbf{x}_i(t) + \mathbf{v}_i(t+1) \tag{3.1}$$

Le particelle (identificate dal pedice i) vengono inizializzate con posizioni casuali all'interno di una porzione dello spazio di ricerca, che, per motivi logistici, deve essere limitata a priori in maniera euristica. La velocità $\mathbf{v}_i(t+1)$ possiede due componenti:

1. componente *cognitiva*: riflette la conoscenza sperimentale e dipende dalla distanza tra la posizione attuale è quella del punto migliore finora incontrato;
2. componente *sociale*: riflette la conoscenza collettiva generata dallo stormo e dipende dalla distanza tra la posizione attuale e quella del punto migliore individuato da una frazione dell'intera popolazione, chiamata *neighborhood*.

Quando la componente sociale dipende dal miglior valore scoperto dall'intero stormo (*neighborhood* pari all'intera popolazione) si parla di *global best* PSO (o

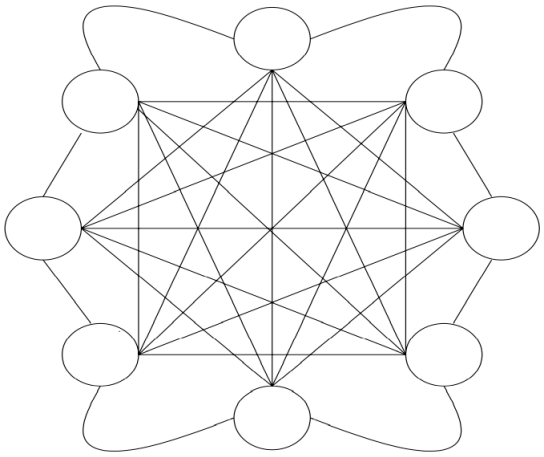


Figura 10: Topologia a Stella [14]

gbest PSO) e la rete sociale utilizzata ha una topologia a Stella (Figura 10), dove ogni ogni particella è connessa e comunica con tutte le altre. Questa era la topologia utilizzata nella prima implementazione del PSO.

La velocità della particella i è aggiornata secondo la seguente regola:

$$v_{ij}(t+1) = wv_{ij}(t) + c_1 r_{1j}(t)[y_{ij}(t) - x_{ij}(t)] + c_2 r_{2j}(t)[\hat{y}_j(t) - x_{ij}(t)] \quad (3.2)$$

dove:

- $v_{ij}(t)$ è la componente $j$esima della velocità della particella i al tempo t;
- $x_{ij}(t)$ è la componente $j$esima della posizione della particella i al tempo t;
- $\hat{\mathbf{y}}(t)$ è la migliore posizione finora trovata dallo stormo;
- $\mathbf{y}_i$ è la migliore posizione che la particella i ha visitato a partire dalla prima iterazione;
- w è un parametro chiamato *fattore di inerzia*, che stabilisce quanto la velocità al tempo t+1 debba dipendere da quella immediatamente precedente (al tempo t);
- c_1 e c_2 sono costanti positive di accelerazione che danno più o meno importanza rispettivamente alla componente cognitiva o sociale;
- r_{1j} e r_{2j} sono valori casuali tra 0 e 1 ricalcolati ad ogni iterazione, allo scopo di aggiungere un contributo stocastico.

Per quanto riguarda problemi di massimizzazione, la miglior posizione personale al prossimo step $t+1$ è calcolata come:

$$\mathbf{y}_i(t+1) = \begin{cases} \mathbf{y}_i(t) & se\ f(\mathbf{x}_i(t+1)) \leq f(\mathbf{y}_i(t)) \\ \mathbf{x}_i(t+1) & se\ f(\mathbf{x}_i(t+1)) > f(\mathbf{y}_i(t)) \end{cases} \tag{3.3}$$

dove f: $\mathbb{R}^{n_x} \to \mathbb{R}$ è la funzione di *fitness* (quasi sempre con valore tra 0 e 1) ed n_x è il numero di dimensioni dello spazio di ricerca. La funzione di **fitness**, similarmente alla funzione di costo delle ANN, misura quanto la soluzione corrispondente al suo input si avvicina al valore ottimale; in senso stretto ne quantifica la qualità.

Sempre per problemi di massimizzazione la miglior posizione globale $\hat{\mathbf{y}}(t)$ al tempo t è aggiornata secondo:

$$\hat{\mathbf{y}}(t) \in \{\mathbf{y}_0(t), \ldots, \mathbf{y}_{n_s}(t)\} | f(\hat{\mathbf{y}}(t)) = \max\{f(\mathbf{y}_0(t)), \ldots, f(\mathbf{y}_{n_s}(t))\} \tag{3.4}$$

dove n_s è il numero di particelle dello stormo.
Il *gbest* PSO può essere riassunto con il seguente pseudocodice:

Algoritmo 1 *gbest* PSO

```
Crea ed inizializza lo swarm;
repeat
    for ogni particella i do
        if f(x_i) > f(y_i) then                    ▷ aggiorna il pbest
            y_i = x_i
        end if
        if f(y_i) > f(ŷ_i) then                    ▷ aggiorna il gbest
            ŷ_i = y_i
        end if
    end for
    for ogni particella i do
        aggiorna velocità;
        aggiorna posizioni;
    end for
until stopping condition is True
```

Il calcolo della velocità in equazione 3.2 prevede tre termini:

1. velocità precedente, $wv_{ij}(t)$: funge da memoria a "breve termine", essendo essenzialmente la velocità al tempo precedente; a volte ci si riferisce ad essa con il termine *inerzia*. Il suo ruolo è di impedire cambi drastici di direzione da parte della particella, che potrebbero portare l'algoritmo ad un comportamento incontrollato e/o caotico;

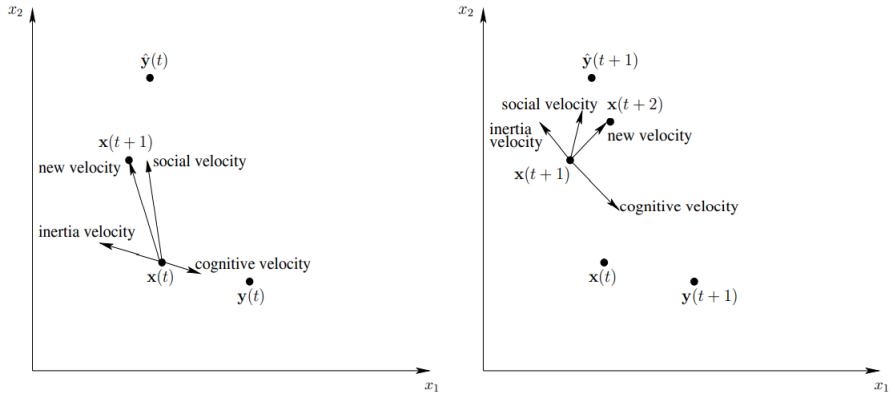
2. componente cognitiva, $c_1 r_{1j}(t)[y_{ij}(t) - x_{ij}(t)]$: analogamente alla precedente, svolge il ruolo di memoria a "lungo termine", definita originariamente da Kennedy ed Eberhart come "nostalgia". Grazie a questo termine le particelle hanno una tendenza a tornare verso la posizione migliore

incontrata fino a quel momento, riflettendo il comportamento degli esseri viventi i quali sono naturalmente spinti a ritornare nei luoghi (o situazioni) che li hanno soddisfatti di più in passato;

3. componente sociale, $c_2 r_{2j}(t)[\hat{y}_j(t) - x_{ij}(t)]$: il vero responsabile del successo del PSO, quantifica la qualità della particella i rispetto al gruppo. Concettualmente rappresenta una "forma" considerata dallo stormo migliore o standard, che ogni individuo punta ad ottenere. Il suo effetto esplicito è quello di trascinare le particelle nella direzione della migliore posizione mai trovata a livello globale.

3.1.2 Visualizzazione Geometrica

Spesso il PSO viene utilizzato per problemi di ottimizzazione a molte dimensioni, sarebbe dunque impossibile da visualizzare, ma, per comprendere meglio gli effetti dei vari termini di velocità, è utile considerare il caso di una singola particella in uno spazio di ricerca bidimensionale.



(a) Time Step t (b) Time Step $t+1$

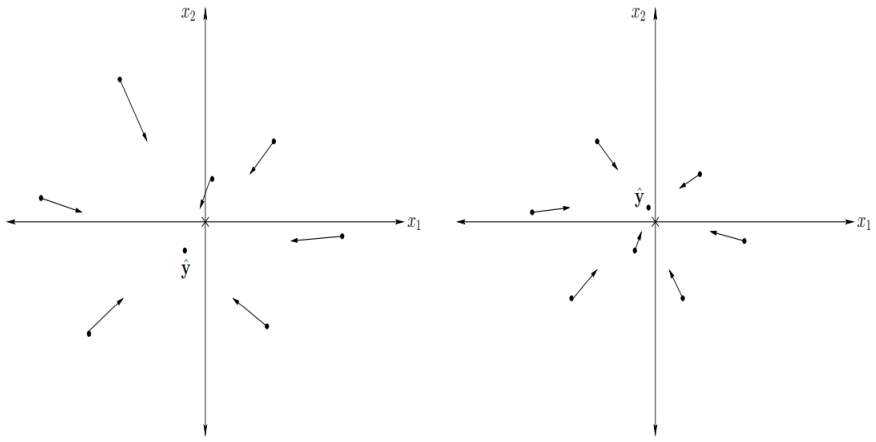
Figura 11: Illustrazione geometrica dell'aggiornamento di posizione e velocità di una particella in spazio bidimensionale [14]

Immaginiamo che, al tempo t, la particella abbia la posizione indicata in Figura 11(a), notiamo come essa si muova, in $x(t+1)$, verso il massimo globale $\hat{\mathbf{y}}(t)$. Nella Figura 11(b) vengono invece mostrate le successive velocità e posizione al tempo t+1.

Assumendo che il *personal best* non cambi, le 3 componenti di velocità, sommate, continuano a trascinare la particella nella direzione del *gbest*.

È possibile, anzi inevitabile, che un individuo "oltrepassi" il *gbest* a causa dell'inerzia; questo non è da intendere come un punto debole dell'algoritmo, anzi può rivelarsi una cosa positiva, risultando in due plausibili scenari:

1. la particella trova un nuovo *gbest*
2. la particella non trova un nuovo *gbest*, per cui, durante la successiva iterazione, viene rilanciata indietro



(a) At time $t = 0$ (b) At time $t = 1$

Figura 12: Illustrazione geometrica di gbest PSO a molte particelle [14]

Tornando al caso di più particelle, la Figura 12 mostra sette particelle in uno spazio bidimensionale in cui il punto ottimale è conosciuto a priori ed è locato nell'origine. Le particelle sono inizializzate uniformemente, al tempo $t = 0$, con velocità nulla (vedi Figura 12(a)). Non possedendo velocità iniziale né precedente, la loro nuova posizione (al tempo $t = 1$) dipenderà unicamente dalla componente sociale, testimoniato dal fatto che tutte le velocità puntano verso il *gbest* $\hat{\mathbf{y}}$.

La Figura 12(b) mostra, invece, la situazione al tempo $t = 1$. Assumendo che un nuovo *gbest* sia stato trovato, adesso tutte e tre le componenti giocano un ruolo nella determinazione della posizione successiva, sebbene la tendenza al seguire l'ultimo *gbest* conosciuto rimanga prioritaria.

3.1.3 Limitazione delle Velocità e fattore di inerzia

Nella definizione di un buon algoritmo di PSO, è importante bilanciare opportunamente due abilità fondamentali:

- *exploration*: la capacità di esplorare diverse regioni in cerca di un buon punto ottimale;
- *exploitation*: la capacità di concentrare la ricerca in una singola area promettente allo scopo di rifinire una posizione candidata a massimo globale.

Un metodo per tenere sotto controllo questi due fattori è la *limitazione delle velocità* (**velocity clamping**).

Durante le prime implementazioni del PSO, fu scoperto che, talvolta, le velocità esplodevano rapidamente fino a portare le particelle ben oltre i confini nei quali erano state inizializzate, portandole a divergere. Per risolvere il problema si è ricorso ad una limitazione perenne delle velocità, costrette a rimanere all'interno di intervalli specifici. Se una particella supera tali limiti, la sua velocità viene posta uguale al limite massimo.

Indicando con $V_{max,j}$ tale valore massimo nella dimensione j, questo controllo viene effettuato prima dell'aggiornamento delle posizioni:

$$v_{ij}(t+1) = \begin{cases} v'_{ij}(t+1) & se\ v'_{ij}(t+1) < V_{max,j} \\ V_{max,j} & se\ v'_{ij}(t+1) \geq V_{max,j} \end{cases} \tag{3.5}$$

dove v'_{ij} è calcolata normalmente secondo l'equazione 3.2. Scegliere un buon valore per $V_{max,j}$ (costante o variabile) è la base per ottenere un giusto compromesso tra particelle troppo veloci/lente ed *exploitation/exploration*; fortunatamente, arriva in soccorso il fattore di inerzia w, che, se opportunamente modificato nel corso dell'esecuzione, assicura una qualche forma di comportamento di convergenza.

Per valori di $w \geq 1$ le velocità aumentano con il tempo, accelerando verso la velocità massima, per $w < 1$ invece rallentano tendendo a zero. Le prime implementazioni di PSO utilizzavano valori fissi di w durante l'intera esecuzione, mentre, ultimamente, sembra essersi affermata la necessità di usare fattori di inerzia che cambiano dinamicamente (e.g. decadono linearmente con il numero di iterazioni da un w_{max} ad un w_{min}).

Questa scelta si basa sull'idea che:

1. nelle fasi iniziali si comincia con w grandi, permettendo alle particelle di esplorare lo spazio durante le prime iterazioni;

2. dopo aver esplorato abbastanza, la diminuzione di w sposta l'attenzione verso l' exploitation, quindi la rifinitura dei candidati a possibili ottimi globali.

3.1.4 Condizioni di arresto

L'ultimo aspetto importante riguardante gli algoritmi di PSO è il criterio usato per terminare il processo iterativo. Tra i molteplici criteri utilizzati i tre più frequenti sono:

- arresto quando si raggiunge un numero massimo di iterazioni: ovviamente, se questo limite è troppo piccolo, il processo potrebbe terminare prima che una buona soluzione sia stata trovata. Capita spesso che questo criterio sia affiancato come sorta di "piano B" nel caso un altro criterio non sia verificato in tempo. Usato da solo, è utile per valutare la migliore soluzione trovata in un tempo (o numero di iterazioni) ristretto;

- arresto quando una soluzione accettabile è stata trovata: è possibile terminare il processo nel caso in cui la differenza tra la soluzione trovata e quella ottimale sia inferiore ad un certo parametro fissato di errore. Questo criterio è utilizzabile soltanto quando si conosce a priori l'ottimo, cosa non banale e non sempre verificata;

- arresto quando non si osserva miglioramento dopo un certo numero di iterazioni: può essere verificato quando lo spostamento medio delle particelle diventa molto piccolo oppure quando le loro velocità tendono approssimativamente a zero.

4 Applicazione di PSO alle ANN

Fin dalla nascita dell'algoritmo di PSO, si è vista una grande potenzialità nella sua applicazione all'allenamento di Reti Neurali Artificiali [9]. Ogni rete neurale si basa essenzialmente su due categorie di coefficienti, i *pesi* ed i *bias*; l'idea centrale dell'applicazione consiste nel creare una corrispondenza biunivoca tra ogni particella ed una possibile configurazione di *pesi* e *bias* della ANN. In tale situazione, trovare un ottimo nello spazio di ricerca equivale (teoricamente) a trovare la rete neurale meglio performante.

4.1 Allenamento tramite PSO

Dopo aver preparato il dataset di riferimento su cui si intende allenare la rete, ha inizio l'esecuzione di un algoritmo di *basic* PSO di tipo *gbest*, nel quale una popolazione di n particelle viene inizializzata nello spazio di ricerca (descritto nel paragrafo successivo) e fatta evolvere durante un numero fissato di iterazioni (o fino alla verifica di una condizione di arresto). Terminato il processo di "movimento", l'individuo più performante è testato sul test set per verificarne la capacità di generalizzazione su dati mai visti prima.

4.1.1 Spazio di ricerca

Ogni particella ha lo scopo di rappresentare una struttura allenata di MLP, dunque deve contenere al proprio interno tutte le informazioni sulle matrici di *pesi* e sui vettori di *bias*, per questo motivo lo spazio di ricerca avrà una dimensionalità pari al loro numero totale.

Le dimensioni di tali matrici e vettori dipendono dal numero di neuroni (nodi) della rete; trattandosi di un'architettura MLP a tre layer possiamo distinguere due matrici e due vettori:

1. *pesi* tra i layer di *input* e *hidden*: matrice di dimensione $n_{features} \times n_{hidden}$, dove *feature* si riferisce alla dimensionalità del vettore in input (istanza) e *hidden* al numero di neuroni presenti nel layer interno, calcolato per ogni dataset come [15]:

$$n_{hidden} = (2 \times n_{features}) + 1 \tag{4.1}$$

2. *bias* del layer di *hidden*: vettore di dimensione n_{hidden};

3. *pesi* tra i layer di *hidden* e *output*: matrice di dimensione $n_{hidden} \times n_{classes}$, dove *classes* si riferisce alla dimensionalità del vettore in output (numero di classi);

4. *bias* del layer di *output*: vettore di dimensione $n_{classes}$.

La dimensionalità dello spazio di ricerca si ottiene "srotolando" questi quattro elementi e collegandoli fino a formare un unico vettore di lunghezza:

$$size = n_{hidden} \times (n_{features} + n_{classes}) + n_{classes} + n_{hidden} \tag{4.2}$$

Allo stesso modo, il vettore posizione nello spazio viene "ricomposto" nei suoi quattro elementi principali dopo aver ricalcolato le posizioni di ogni particella.

Prendendo come esempio il dataset *Iris*, che presenta 4 feature e 3 classi, le particelle si muoveranno in uno spazio a 75 dimensioni.

Al momento della loro creazione le particelle sono disposte uniformemente in un intervallo dello spazio scelto a priori su basi euristiche o riferendosi alla letteratura.

4.1.2 Funzione di fitness

Come funzione di *fitness* per valutare le prestazioni delle particelle è stata utilizzata una versione alternativa dell'*accuratezza* standard sul train set: la *Balanced Accuracy* [17]. L'*accuratezza* (*accuracy*) è definita come:

$$Accuracy = \frac{TP}{n_{set}} \tag{4.3}$$

dove n_{set} è il numero totale di istanze del set su cui stiamo calcolando la metrica e TP (*True Positive*) è il numero di istanze classificate correttamente (considerando tutte le classi).

Nonostante sia apparentemente la metrica più intuitiva e giustificata, attribuisce la stessa importanza ad ogni classe, tendendo a trascurare quelle con meno istanze a favore delle più popolate. Un'alternativa che tiene conto di questa problematica e la risolve è per l'appunto la *Balanced Accuracy*:

$$Balanced\ Accuracy = \frac{1}{n_{classes}} \sum_{i=1}^{n_{classes}} \frac{TP_i}{Total_i} \tag{4.4}$$

dove $Total_i$ e TP_i sono rispettivamente il numero di istanze realmente appartenenti alla classe i e il numero di quelle correttamente classificate.

Questa sua evoluzione è particolarmente utile quando il dataset su cui si lavora risulta non *bilanciato*, ossia quando non tutte le classi possiedono lo stesso numero di istanze.

Per evitare un'accuratezza inflazionata dal popolamento eterogeneo viene assegnato ad ognuna di esse un peso basato sull'inverso della loro prevalenza nel dataset. Trattandosi di una metrica che si intende massimizzare (il miglior scenario possibile è una *fitness* di 1, cioè il 100% di istanze classificate correttamente), la migliore particella sarà quella con la *fitness* più grande.

All'inizio di ogni iterazione viene valutata la *fitness* di ogni particella, trasformando la sua posizione corrente in matrici di *pesi* e vettori di *bias* e creando una rete neurale a cui vengono assegnati tali parametri.

Successivamente si confrontano i valori di *fitness* di ogni individuo prima con il *pbest* e poi con il *gbest*, aggiornandoli opportunamente; infine le particelle sono mosse nello spazio di ricerca in base alle velocità calcolate e si parte con l'iterazione successiva.

4.1.3 Termine dell'allenamento

Ci sono diversi criteri utilizzabili per la terminazione dell'algoritmo, il più frequente (quello scelto per il lavoro di tesi) è semplicemente stabilire un numero massimo di iterazioni, utile quando si vuole effettuare un confronto con altri algoritmi o si vuole limitare il tempo di esecuzione.

Un altro possibile criterio di terminazione potrebbe essere il raggiungimento di una *accuratezza* prestabilita, sebbene non sia scontato che l'algoritmo non rimanga ad un valore di best fitness inferiore, non essendo garantita a prescindere la convergenza nel punto ottimale reale.

4.2 Strumenti utilizzati

L'algoritmo è stato implementato in linguaggio Python e sono state utilizzate le librerie DEAP [24] e Scikit-learn [25] rispettivamente per la gestione del PSO e delle Reti Neurali. L'esperimento è stato realizzato su una macchina che monta un processore AMD Ryzen 7 3700x.

DEAP

Distributed Evolutionary Algorithms in Python (**DEAP**) è una libreria in linguaggio Python finalizzata a semplificare la costruzione di algoritmi evolutivi. In particolare fornisce strumenti molto utili per la gestione di:

- *Tipi*, o classi: permette di creare facilmente nuove classi attraverso il modulo `creator`; a differenza del procedimento standard di Python, ciò può essere realizzato in modo intuitivo e compatto, spesso richiedendo una singola linea di codice;
- *Inizializzazione*: dopo aver creato un individuo è necessario inizializzare i suoi valori e creare una popolazione prima dell'inizio del processo iterativo; il modulo `Toolbox` funge letteralmente da "cassetta degli attrezzi", in cui sono già collezionati numerosi metodi di inizializzazione e di gestione di popolazioni, lasciando comunque all'utente la piena libertà nel definirne facilmente di nuovi;
- *Operatori*: in generale è possibile aggiungere qualsiasi tipo di funzione al proprio `toolbox`, ognuna di esse acquisirà un nuovo nome, in modo tale da poter essere richiamata da algoritmi generici. Il modulo permette anche di semplificare la definizione di funzioni.

Scikit-learn

Scikit-learn (abbreviata *sklearn*) è una libreria Python rivolta alla creazione e gestione di algoritmi di Machine Learning, costruita compatibilmente alle librerie di analisi dati *NumPy*, *SciPy* e *Matplotlib*.

L'ambiente *sklearn* contiene numerosi modelli per l'analisi dati, tra cui algoritmi di Supervised e Unsupervised Learning, Clustering, Feature Extraction e

Selection e molti altri, in più include funzioni in grado di semplificare notevolmente la preparazione di dataset.

Nell'ambito delle Reti Neurali Artificiali, fornisce strumenti semplici ed efficaci per la costruzione, l'allenamento e la validazione di strutture più o meno complesse; ad esempio, la classe `MLPClassifier` consente di generare in modo rapido reti di tipo MLP, lasciando piena libertà all'utente per quanto riguarda parametri come funzioni di attivazione, learning rate e numero di neuroni.

L'allenamento della rete è affidato al metodo `fit`, mentre le sue performance possono essere misurate attraverso una delle tante possibili metriche offerte dalla libreria.

5 Esperimenti e risultati

In questo lavoro di tesi è stato utilizzato un algoritmo di PSO con lo scopo di allenare una rete neurale finalizzata alla classificazione; è stata scelta una rete del tipo Multi Layer Perceptron, avente la struttura più semplice possibile: un layer di *input*, uno di *hidden* ed uno di *output*. Il numero di neuroni nei layer di *input* ed *output* corrisponde rispettivamente a quello di feature e classi del dataset utilizzato per l'allenamento e la validazione, mentre il numero di neuroni nel layer *hidden* è imposto seguendo la Formula 4.1.

Le reti allenate tramite PSO sono state infine confrontate con reti di struttura identica ma allenate tramite BackPropagation.

5.1 Workflow

Il workflow dell'esperimento può essere riassunto nel seguente diagramma:

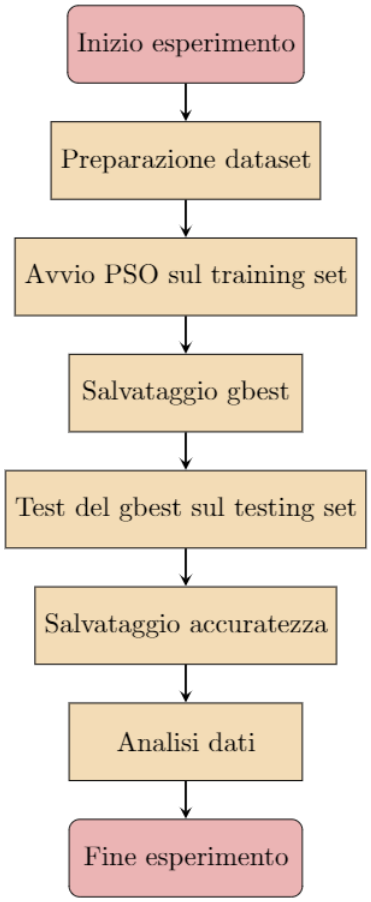


Figura 13: Flowchart dell'esperimento

5.2 Preprocessing

All'inizio di ogni test viene caricato il dataset su cui sarà allenata la rete e le relative informazioni di utilizzo (numero di feature e classi). Successivamente viene suddiviso in train set (70% delle istanze) e test set (30% delle istanze), insiemi che vengono "miscelati" internamente per eliminare un possibile pattern derivato dall'acquisizione originale dei dataset. Questo mescolamento dipende da un *seed*, il quale assicura che, ad ogni test, i dati vengano rimescolati allo stesso modo, eliminando la possibile dipendenza dei risultati dal loro ordine di ingresso.

5.2.1 Dataset

La rete neurale è stata allenata nella classificazione di 12 dataset, provenienti dai tre archivi gratuiti: UCI [10], Kaggle [11] ed OpenML [12]. È stata testata sia la capacità di classificazione binaria (primi 6 dataset) sia quella multiclasse (ultimi 6). Le Tabelle 1 e 2 riassumono le caratteristiche dei dataset:

Dataset	**Features**	**Classi**	**Istanze**
Blood	4	2	748
Breast Cancer	30	2	569
Diabetes	8	2	768
Liver	10	2	583
Parkinson	22	2	195
Tictactoe	9	2	958

Tabella 1: Descrizione dataset per la classificazione binaria

Dataset	**Features**	**Classi**	**Istanze**
Balance	4	3	625
Iris	4	3	150
Nursery	8	4	700
Seeds	7	3	210
Vertebral	6	3	310
Wine	13	3	178

Tabella 2: Descrizione dataset per la classificazione multiclasse

I dataset sono stati scelti mantenendo il numero di feature tendenzialmente inferiore a 15 e quello di classi tra 2, 3, e 4, al fine di presentare all'algoritmo un numero di dimensioni non troppo elevato, per non appesantire i tempi di computazione. Per il medesimo motivo, tutti i dataset possiedono un numero di istanze inferiore a 1000.

5.2.2 Normalizzazione

Come spesso accade quando si lavora con algoritmi di machine learning, la rimodulazione dei dati in input è un procedimento indispensabile se si ha intenzione di massimizzare l'accuratezza e minimizzare il costo computazionale.

Per questo motivo i valori di feature di ogni dataset sono stati normalizzati secondo la procedura:

$$x_{new} = \frac{x_{old} - \mu}{\sigma} \tag{5.1}$$

con μ valore medio della feature nel training set e σ corrispondente deviazione standard.

5.2.3 Iperparametri Utilizzati

I diversi iperparametri sono stati scelti seguendo la letteratura e/o in modo euristico [15], e sono stati mantenuti uguali per ogni dataset. Le Tabelle 3 e 4 illustrano i parametri utilizzati rispettivamente per l'apprendimento tramite PSO e BP:

Parametro	**Valore**
c_1	2.1
c_2	2.1
w_{max}	0.9
w_{min}	0.6
x_{max}	2
x_{min}	-2
v_{max}	1
v_{min}	-1
Popolazione	50
Iterazioni	200

Tabella 3: Parametri del PSO

Parametro	**Valore**
L2 (λ)	0.0001
Learning Rate (η)	0.001
Iterazioni	200
Attivazione	ReLU
Solver	SGD

Tabella 4: Parametri del BP

Le particelle sono inizializzate con posizioni e velocità (in ciascuna dimensione) compresi tra i valori massimi indicati in Tabella 3. Il fattore di inerzia w, è aggiornato in modo dinamico ad ogni iterazione, partendo da un valore iniziale di w_{max} e decrescendo con le iterazioni fino a w_{min}. Il suo valore durante la $i$esima iterazione sarà:

$$w(i) = w_{max} - \frac{(w_{max} - w_{min}) \times i}{Iterazioni} \tag{5.2}$$

Per il confronto con l'algoritmo di BackPropagation sono stati scelti parametri standard [25]. In Tabella 4, λ si riferisce al fattore di regolarizzazione L2 definito in Equazione 2.29. Per *Solver* si intende il metodo di ottimizzazione, in questo caso è stato utilizzato lo Stochastic Gradient Descent.

5.3 Risultati

Per ogni dataset sono state effettuate 15 run di allenamento tramite PSO e 15 tramite BP; nel caso del PSO, alla fine dell'ultima iterazione di ogni run, il migliore individuo è stato scelto come rete allenata e testato anche sul test set. Sebbene le due reti siano state allenate con metodi diversi, per garantire un giusto confronto finale riguardo le performance sul test set è stata utilizzata la stessa metrica (*accuracy*). Di seguito sono illustrati i risultati dell'esperimento sotto forma di valore medio (Avg), deviazione standard (Std), valore massimo (Max) e valore minimo (Min).

Accuratezze sul train set

		PSO	BP
Blood	Avg	0.736	0.760
	Std	0.004	0.009
	Max	0.745	0.776
	Min	0.732	0.751
Breast Cancer	Avg	0.989	0.966
	Std	0.003	0.004
	Max	0.993	0.975
	Min	0.983	0.957
Diabetes	Avg	0.794	0.763
	Std	0.008	0.017
	Max	0.809	0.803
	Min	0.784	0.726
Liver	Avg	0.746	0.719
	Std	0.006	0.003
	Max	0.759	0.723
	Min	0.734	0.711
Parkinson	Avg	0.958	0.848
	Std	0.010	0.022
	Max	0.975	0.890
	Min	0.935	0.809
Tictactoe	Avg	0.939	0.748
	Std	0.030	0.022
	Max	0.976	0.781
	Min	0.855	0.707

Tabella 5: Dataset binari

		PSO	BP
Balance	Avg	0.910	0.878
	Std	0.024	0.013
	Max	0.929	0.895
	Min	0.829	0.847
Iris	Avg	0.986	0.826
	Std	0.006	0.083
	Max	0.991	0.943
	Min	0.971	0.619
Nursery	Avg	0.969	0.917
	Std	0.012	0.008
	Max	0.983	0.927
	Min	0.940	0.898
Seeds	Avg	0.980	0.894
	Std	0.007	0.031
	Max	0.993	0.925
	Min	0.966	0.810
Vertebral	Avg	0.883	0.810
	Std	0.007	0.041
	Max	0.895	0.852
	Min	0.867	0.700
Wine	Avg	0.997	0.962
	Std	0.004	0.018
	Max	1.000	0.984
	Min	0.990	0.919

Tabella 6: Dataset multiclasse

Accuratezze sul test set

		PSO	BP
Blood	Avg	0.651	0.789
	Std	0.012	0.009
	Max	0.689	0.804
	Min	0.636	0.764
Breast Cancer	Avg	0.964	0.954
	Std	0.016	0.008
	Max	0.988	0.971
	Min	0.924	0.942
Diabetes	Avg	0.724	0.739
	Std	0.024	0.020
	Max	0.758	0.784
	Min	0.658	0.701
Liver	Avg	0.637	0.701
	Std	0.019	0.004
	Max	0.674	0.709
	Min	0.611	0.691
Parkinson	Avg	0.861	0.904
	Std	0.046	0.018
	Max	0.932	0.932
	Min	0.780	0.864
Tictactoe	Avg	0.915	0.708
	Std	0.042	0.024
	Max	0.969	0.760
	Min	0.795	0.674

Tabella 7: Dataset binari

		PSO	BP
Balance	Avg	0.895	0.887
	Std	0.019	0.010
	Max	0.915	0.904
	Min	0.851	0.862
Iris	Avg	0.953	0.846
	Std	0.040	0.092
	Max	1.000	0.956
	Min	0.867	0.600
Nursery	Avg	0.930	0.902
	Std	0.034	0.014
	Max	0.971	0.914
	Min	0.829	0.867
Seeds	Avg	0.919	0.848
	Std	0.028	0.036
	Max	0.968	0.905
	Min	0.841	0.762
Vertebral	Avg	0.771	0.700
	Std	0.037	0.034
	Max	0.817	0.763
	Min	0.677	0.634
Wine	Avg	0.952	0.951
	Std	0.031	0.026
	Max	1.000	0.981
	Min	0.907	0.889

Tabella 8: Dataset multiclasse

Dalle quattro tabelle si deduce che i due algoritmi tendono a performare in modo diverso a seconda della dimensionalità del problema di classificazione. Per giudicare la qualità dei due metodi sono state calcolate le accuratezze medie su tutti i dataset ed eseguito un test non parametrico chiamato test di *Mann-Whitney U* [26] scegliendo il livello di *significatività* α pari a 0.1, per determinare la validità statistica delle osservazioni.

Risultati su dataset binari

L'analisi dei valori di accuratezza su dataset di natura binaria hanno evidenziato un leggero vantaggio da parte del BP per quanto riguarda il test set, battendo il PSO 4 volte su 6, sebbene le differenze non siano schiaccianti; spostandoci sul train set, il PSO risulta comunque migliore 5 volte su 6 in modo convincente. Analizzando i dati si trova:

- accuratezza media PSO sul test set = 0.792;

- accuratezza media BP sul test set = 0.799;
- accuratezza media PSO sul train set = 0.860;
- accuratezza media BP sul train set = 0.801.

Il test di Mann-Whitney U non ha generato in alcun caso un p-valore inferiore a 0.1, quindi non si può strettamente parlare di differenza statistica tra i due algoritmi.

Risultati su dataset multiclassse

La situazione si fa molto diversa se si considerano unicamente dataset per classificazione multiclasse; in questo caso il PSO ottiene accuratezze decisamente superiori al BP su ogni singolo dataset sia nel train che nel test. L'analisi ai valori medi ha restituito:

- accuratezza media PSO sul test set = 0.903;
- accuratezza media BP sul test set = 0.856;
- accuratezza media PSO sul train set = 0.954;
- accuratezza media BP sul train set = 0.881.

A differenza del caso precedente il test non parametrico ha confermato la superiorità statistica del PSO, sia sul train set, con un p-valore = 0.02, che sul test set, con p-valore = 0.08.

Risultati sulla totalità dei dataset

Infine la stessa analisi è stata effettuata sull'insieme totale dei dataset, dove il punteggio favorisce indubbiamente il PSO, che vince 12 volte su 12 nel train e 8 volte su 12 nel test:

- accuratezza media PSO sul test set = 0.848;
- accuratezza media BP sul test set = 0.827;
- accuratezza media PSO sul train set = 0.907;
- accuratezza media BP sul train set = 0.841.

Il p-valore per quanto riguarda il test set è pari a 0.22, abbastanza da poter individuare una prevalenza del PSO sul BP ma formalmente non sufficiente; si può invece affermare con certezza la sua superiorità statistica riguardo i train set, con un p-valore di 0.03.

Conclusioni

Nella prima parte di questo elaborato di tesi sono stati approfonditi gli argomenti di Machine Learning, Reti Neurali Artificiali e Particle Swarm Optimization. Nella seconda, è stata discussa l'applicazione dell'algoritmo di Particle Swarm Optimization come metodo di apprendimento di Reti Neurali, mentre, nella terza ed ultima, tale modalità di allenamento è stata confrontata sperimentalmente con il noto algoritmo di BackPropagation.

L'analisi finale ha mostrato una superiorità generale del PSO, particolarmente accentuata nell'ambito di problemi di classificazione non binaria; le motivazioni per tale risultato non sono facili da individuare, ma è probabile che abbiano a che fare con la capacità dell'algoritmo di PSO di uscire dai minimi locali, problema caratteristico di algoritmi basati sulla discesa del gradiente come quello di BackPropagation.

Sebbene l'algoritmo di Particle Swarm Optimization sia stato ideato ormai quasi trent'anni fa, ancora dimostra la propria validità nei confronti di altre tecniche di ottimizzazione, nonché le sue potenzialità in applicazioni sempre nuove.

Bibliografia

[1] A Logical Calculus of the Ideas Immanent in Nervous Activity, W. S. McCulloch and W. Pitts, Bulletin of Mathematical Biophysics, 5(4): 115-133, 1943.

[2] The Perceptron: A Perceiving and Recognizing Automaton, F. Rosenblatt, Cornell Aeronautical Laboratory, 1957.

[3] An Adaptive "Adaline" Neuron Using Chemical "Memistors", Technical Report Number 1553-2, B. Widrow and others, Stanford Electron Labs, Stanford, CA, October 1960.

[4] https://www.treccani.it/vocabolario/sciame

[5] G. Beni and J. Wang, Swarm intelligence in cellular robotic systems. In NATO Advanced Workshop on Robots and Biological Systems, Il Ciocco, Tuscany, Italy, 1989.

[6] M. Dorigo, Optimization, learning and natural algorithms (in Italian), Ph.D. Thesis, Dipartimento diElettronica, Politecnico di Milano, Italy, 1992.

[7] J. Kennedy and R. C. Eberhart. Particle Swarm Optimization. In Proceedings of IEEE International Conference on Neural Networks, Perth, Australia, pp. 1942– 1948, 1995.

[8] R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In Proceedings of the Sixth International Symposium on Micro Machine and Human Science, Nagoya, Japan, pp. 39–43, 1995.

[9] J. Kennedy. The Particle Swarm: Social Adaptation of Knowledge. In Proceedings of the IEEE International Conference on Evolutionary Computation, pages 303–308, 1997.

[10] https://archive.ics.uci.edu/ml/datasets.php

[11] https://www.kaggle.com

[12] https://www.openml.org

[13] S. Raschka and V. Mirjalili. Python Machine Learning. Packt Publishing, 2017.

[14] A. P. Engelbrecht. Computational Intelligence, An Introduction. John Wiley and Sons, 2007.

[15] Bousmaha, Rabab, Reda Mohamed Hamou and Abdelmalek Amine. "Training Feedforward Neural Networks Using Hybrid Particle Swarm Optimization, Multi-Verse Optimization." CITSC 2019.

[16] Omid Tarkhaneh, Haifeng Shen. Training of feedforward neural networks for data classification using hybrid particle swarm optimization, Mantegna Lévy flight and neighborhood search. Heliyon 5 2019.

[17] Grandini, Margherita, Enrico Bagli and Giorgio Visani. "Metrics for Multi-Class Classification: an Overview." ArXiv abs/2008.05756 2020.

[18] Ahmed, Hazem and Glasgow, Janice. Swarm Intelligence: Concepts, Models and Applications, 2012.

[19] https://www.javatpoint.com/regression-vs-classification-in-machine-learning

[20] https://www.thepartnershipineducation.com/resources/nervous-system

[21] https://www.researchgate.net/figure/ReLU-activation-function_fig7_333411007

[22] https://sebastianraschka.com/images/faq/diff-perceptron-adaline-neuralnet/4.png

[23] https://miro.medium.com/max/600/1*iNPHcCxIvcm7RwkRaMTx1g.jpeg

[24] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau and Christian Gagné, "DEAP: Evolutionary Algorithms Made Easy", Journal of Machine Learning Research, pp. 2171-2175, no 13, jul 2012.

[25] Scikit-learn: Machine Learning in Python, Pedregosa et al., JMLR 12, pp. 2825-2830, 2011.

[26] Mann, Henry B., and Donald R. Whitney. "On a test of whether one of two random variables is stochastically larger than the other." The annals of mathematical statistics (1947): 50-60.