Università degli Studi di Napoli Federico II

Facoltà di Scienze MM.FF.NN.

Corso di Laurea in Informatica

Tesi sperimentale di Laurea Triennale

Calcolo ad alte prestazioni basato su GPU. Un modello ibrido neurale-genetico per data mining in astrofisica

Relatori
Prof. Guido Russo
Dr. Massimo Brescia

Candidato
Andrea Solla
matr. 566/2867

Anno Accademico 2011-2012

Ai miei genitori, senza i quali non sarei dove sono, e a te, che hai sempre creduto in me....

Indice generale

Indice delle illustrazioni

1. Introduzione

I settori della ricerca legati all'ICT (Information & Communication Technology) hanno rapidamente consolidato il proprio stato di strumento essenziale per molteplici ambiti di speculazione scientifica, al punto da identificare la scienza informatica come un elemento comune e integrante per la moderna ricerca multi-disciplinare. Il calcolo distribuito ad alte prestazioni ha quindi avviato un processo irreversibile di mutazione tecnologica, condizionando di fatto il modo con cui compiere la ricerca scientifica moderna. In molti settori scientifici di base, fra cui l'astronomia e l'astrofisica, la ricerca si basa ormai sulla necessità di esplorare enormi quantità di dati, spesso eterogenei e derivanti sia da osservazioni reali che da simulazioni complesse e computazionalmente onerose. In breve, molte scienze sono ormai considerate dato-centriche, ponendo cioè l'analisi dei dati al centro della speculazione teorica e sperimentale. Il problema odierno è dunque imperniato sull'eterogeneità e interoperabilità tra grandi archivi di dati. Gli obiettivi di questo nuovo paradigma della scienza moderna sono dunque focalizzati sulla creazione di strumenti informatici in grado di permettere agli scienziati di esplorare, in modo semplice, affidabile ed efficiente i dati, rendendo al contempo trasparente e asincrono l'uso di infrastrutture di calcolo geograficamente delocalizzate. Le moderne infrastrutture computazionali richiedono dunque l'esigenza, da parte degli scienziati, di superare qualunque barriera tra il calcolo e l'acquisizione di conoscenza sui dati (Fabbiano et al. 2010).

Il nostro convincimento è che molti aspetti del calcolo ad alte prestazioni cresceranno al punto da rispettare la legge di Moore (Moore 1965), in termini di dimensioni dei sistemi di memorizzazione e di potenza di calcolo, ma causando a breve un invalicabile gap relativo alla velocità di trasferimento di informazioni tra unità centrale (CPU) e memoria. Tale limitazione potrà quindi essere superata solo attraverso nuove tecnologie e architetture computazionali.

Certamente la legge di Moore permetterà l'evoluzione di infrastrutture di calcolo

parallelo, basato sul concetto di sistemi MPI (Message Passing Interface) su singoli chip basati su architetture multi-core, ma la velocità del clock di sistema rimarrà limitato dall'effetto collaterale dell'effetto termico, dovuto alla dissipazione del calore prodotto da questi processori (Sutter 2005). La diretta conseguenza sarà l'aumento dei costi di tali sistemi, inevitabilmente destinati a grossi centri di calcolo, spesso problematici per un uso intensivo e semplice da parte dell'utenza scientifica. Il risultato di un simile scenario sarà che si produrrà un esiguo numero di supercomputer a costi elevatissimi. Da un punto di vista applicativo ciò richiederà un drastico passaggio dall'attuale paradigma di programmazione parallela o sequenziale, applicato in ambito scientifico, ad un sistema ibrido, che mescoli la programmazione parallela con quella sequenziale, in grado di sfruttare a proprio vantaggio i tempi di latenza delle CPU multi-core.

Il presente lavoro di tesi va proprio in questa direzione. L'argomento discusso riguarda la progettazione ed implementazione di un modello di Soft Computing, ossia di un algoritmo ibrido di machine learning, basato su una rete neurale addestrata da un algoritmo genetico, dedicato al data mining su grandi archivi di dati. L'originalità del presente lavoro riguarda sia l'applicazione di un tale modello a dati di tipo astrofisico, sia l'architettura computazionale scelta per la sua implementazione. Sono infatti state realizzate due versioni dell'algoritmo, una sequenziale ed una parallela in ambiente CUDA (Compute Unified Device Architecture) C, su piattaforma GPGPU (General Purpose Graphics Processing Unit) della NVIDIA corporation (NVIDIA Corp. 2012).
Il modello ibrido deriva dal paradigma “supervisionato” del Machine Learning, dedicato cioè alla risoluzione di problemi di ottimizzazione nell'ambito di classificazione e regressione ed applicato direttamente su problemi di natura astrofisica (classificazione di oggetti astronomici peculiari estratti da cataloghi osservativi e regressione su oggetti astronomici al fine di individuarne la funzione, non nota analiticamente che sottintende al background noise, di

correlazione tra le loro caratteristiche fisiche osservate.

Il modello ottenuto, denominato FMLPGA (*Fast Multi Layer Perceptron trained by Genetic Algorithms*), si basa su un analogo modello (MLPGA), implementato nel recente passato in C++ e reso disponibile per la comunità attraverso la web application DAMEWARE, uno dei progetti afferenti alla Collaborazione internazionale DAME (Data Mining & Exploration), di cui l'Università Federico II è uno dei tre partner ufficiali (insieme all'INAF Osservatorio Astronomico di Capodimonte ed al California Institute of Technology della California). Maggiori dettagli sul progetto DAMEWARE sono descritti in Djorgovski et al. 2012.

Su tale piattaforma web il modello MLPGA è stato testato e validato scientificamente su dati astrofisici, ma nella sua versione originaria in C++, presenta una grossa limitazione: risulta poco scalabile rispetto a grandi volumi di dati. Il presente lavoro è dunque consistito nel riprogettare il modello in ambiente GPU, onde realizzarne una versione altamente performante, in grado di ottimizzare i tempi necessari all'esecuzione, pur garantendone le medesime prestazioni da un punto di vista scientifico.

2. Data Mining su grande volumi di dati

Negli ultimi anni, i progressi nel campo dell'acquisizione di dati digitali e l'incremento delle prestazioni dei dispositivi di immagazzinamento hanno portato alla crescita di grandi database. I dati contenuti in questi database vanno dalle transazioni bancarie e tabulati telefonici fino ad arrivare a dati tecnico/scientifici come cartelle cliniche, dati astronomici…ecc

Partiamo da un presupposto reale e fondamentale: viviamo in un mondo sommerso da un'infinità di dati.

I dati possono essere di molti tipi: tabelle, immagini, grafici, osservati, simulate, calcolate da statistiche o acquisiti da diversi tipi di sistemi di monitoraggio. Inoltre la recente esplosione del World Wide Web e altre risorse ad alte prestazioni stanno rapidamente contribuendo alla proliferazione di enormi database di dati. Questa grande quantità di dati porta ad una domanda importante: in che modo possiamo gestire, capire e usare in modo efficiente e completo?

E' ormai ampiamente riconosciuto lo squilibrio cronico tra la crescita dei dati disponibili e la capacità di gestirli (Hey et al. 2009) e nella maggior parte dei casi, i dati acquisiti, non sono direttamente interpretabili e comprensibili; questo o perché sono oscurati da alcune informazioni ridondanti o fonti di rumore, oppure perché hanno bisogno di essere fusi con altri dati.

Il "quarto paradigma della scienza" (Hey et al., 2009), pone in primis, ancora prima della rappresentazione o della strategia di memorizzazione, il problema della comprensione dei dati. Questo infatti pone in termini scientifici una metodologia per ricavarsi informazioni utili a partire da una conoscenza senza pregiudizi e da datasets di qualsiasi tipo (Brescia & Longo, 2012). Di norma questo paradigma impone l'uso di strumenti informatici efficaci e versatili, in grado di colmare il divario tra le limitate capacità umane (in termini di tempo di elaborazione) ed una crescita costante della quantità e della complessità dei dati oggi in nostro possesso. In altre parole si necessita di un software in grado di

replicare le alte capacità di apprendimento e adattamento del cervello umano, e allo stesso tempo riuscire a gestire un numero molto elevato di dati, caratteristica dei moderni calcolatori.

Queste due prerogative, sono, i punti cardine sui quali si basano due discipline di apprendimento: ***Data Mining (DM)*** e ***Machine Learning*** (ML).

Andando nel particolare, ci sono due concetti chiave che devono essere chiariti:

- Cosa si intende tecnicamente con il termine "apprendimento"?
- Come sfruttare le capacità computazioni delle macchine per eseguire esperimenti di apprendimento?

Generalmente l'aspetto importante non è tanto se un computer riesca ad apprendere o meno, bensì se sia in grado di rispondere in maniera corretta a specifiche domande. Questa capacità non è però sufficiente per affermare che un computer abbia imparato, soprattutto se si considera che il vero apprendimento è legato alla capacità di generalizzazione di un problema. In altre parole, il fatto che una macchina dia risposte corrette a domande note, utilizzate durante la fase di addestramento, è solo la fase preliminare della sua formazione completa. Quello che più interessa è il comportamento della macchina in situazioni imprevedibili, cioè in quelle domande mai poste durante la fase di "training".

Il Machine Learning è una disciplina scientifica che si occupa della progettazione e dello sviluppo di algoritmi che consentono ai computer di far evolvere i propri comportamenti basandosi su dati empirici.

Un algoritmo può usufruire di esempi (dati) per cogliere caratteristiche di interesse della loro distribuzione statistica. Questi dati costituiscono la cosiddetta Knowledge Base (KB), ovvero un insieme sufficientemente grande di esempi da utilizzare per l'addestramento dell'algoritmo, e per testarne le prestazioni.

I metodi di DM, d'altro canto, sono molto utili per ottenere informazioni a partire

da piccoli dataset e, pertanto, possono essere utilizzati in maniera efficace per affrontare problemi di scala molto più piccola (Brescia et al., 2012A).

Il DM su Massive Data Sets (MDS) pone due sfide importanti per quanto riguarda le infrastrutture computazionali: l'accesso asincrono e la scalabilità. Infatti, tramite le operazioni sincrone, tutte le entità che concorrono al funzionamento dell'attività devono rimanere attive per tutta la durata dell'attività: se qualche entità si interrompe, tutto il lavoro svolto dall'attività verrà perso. Per quanto riguarda la scalabilità, il miglior approccio per gestire una grande quantità di dati è quello di suddividere il problema in parti più piccole e farli eseguire da CPU differenti (parallelizzazione) ed infine ricombinare insieme i risultati ottenuti dai singoli sotto-problemi. Finora la tecnologia di calcolo parallelo scelta per questo scopo è stata la GPGPU (General-Purpose computing on Graphics Processing Units)

Nello scenario del DM, la scelta del modello di ML deve essere sempre associata ad un dominio delle funzionalità. Per essere più precisi, alcuni modelli di ML possono essere usati nello stesso dominio delle funzionalità, perché questo rappresenta il contesto funzionale nel quale avviene l'esplorazione dei dati.

Alcuni esempi di domini sono:

- Riduzione dimensionale
- Classificazione
- Regressione
- Clustering
- Segmentazione
- Previsioni
- Filtraggio dei modelli di Data Mining
- Analisi statistica

Nel presente lavoro si vuole focalizzare l'attenzione sulla classificazione e la regressione perché queste sono direttamente correlate con i domini funzionali del modello di Machine Learning presentato (MLPGA).

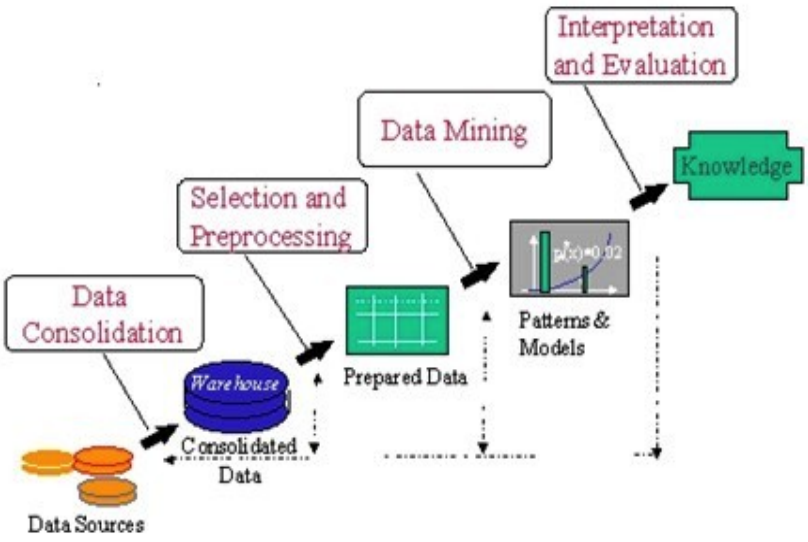


Illustrazione 1: Schematizzazione di un esperimento di Data Mining

2.1. Classificazione

La classificazione è una procedura nella quale singoli oggetti vengono assegnati a gruppi definiti, grazie a dataset contenenti informazioni inerenti agli oggetti stessi e sulla base di un training set di voci precedentemente etichettate (Kotsiantis 2007). Si intende con "classificatore" un sistema che esegue una mappatura tra uno spazio di caratteristiche X e un set di etichette Y. Fondamentalmente un classificatore assegna un etichetta predefinita per ogni campione.

Un problema di classificazione può essere formalmente formulato come segue: dati i seguenti dati di training {(x_1, y_1),.,(x_n, y_n)}, (dove x_i sono vettori), un classificatore h:X->Y mappa un oggetto x ε X sulla sua etichetta di classificazione y ε Y.

Durante la classificazione possono verificarsi le seguenti situazioni:

- ***"Classificazione esatta"***: dato un pattern 'x' in input, il classificatore ritorna la sua etichetta 'y' (scalare).
- ***"Classificazione probabilistica":*** Dato un pattern 'x' in input, il classificatore restituisce un vettore 'y' che contiene la probabilità che 'y_i' sia l'etichetta giusta per 'x'. In altre parole cerchiamo, per ogni vettore, la probabilità che esso sia un membro della classe y_i (qualunque sia 'y_i').

Entrambi i casi possono essere applicati in esperimenti che prevedono due o più classi di classificazione. Le fasi di classificazione sono sostanzialmente tre:

- ***Fase di Training:*** durante questa fase l'algoritmo viene allenato a restituire delle valutazione di qualche tipo, a partire da un insieme di pattern di input.
- ***Fase di Testing:*** prevede una serie di test durante i quali l'algoritmo "allenato" durante la fase di training prende in ingresso pattern e vettori di "target" e restituisce dati statistici, matrici di confusione, errore globale, nonché le etichette di classificazione ricavate per ogni pattern).
- ***Fase di valutazione:*** In questa fase viene dato all'algoritmo un insieme di dati di cui non si conosce il valore di output. L'algoritmo restituisce

  le etichette di classificazione per ogni pattern di input.

A causa della natura "supervisionata" degli esperimenti di classificazione, le prestazioni del sistema possono essere misurate durante la fase di "testing" tramite l'utilizzo di un insieme di vettori di "target", i quali vengono confrontati con le etichette generate dall'algoritmo. Il tasso di errore di classificazione, dato un dataset input, può non essere considerato rappresentativo della qualità del classificatore stesso. Infatti, nel caso in cui il dataset sia “sbilanciato”, il tasso di

errore potrebbe essere elevato anche se il classificatore, da parte sua, risulta performante.

Per una corretta visualizzazione delle prestazioni di classificazione può essere calcolata una matrice di confusione (Provost et al 1998.): ogni colonna della matrice rappresenta le istanze di una classe prevista, mentre ciascuna riga rappresenta le istanze di una classe reale. Uno dei principali benefici della matrice di confusione è quello di verificare in modo semplice se il sistema stia mischiando due classi. In alternativa, si può utilizzare una procedura di validazione (molti modelli di classificazione non necessitano questa procedura).

La procedura di validazione è un processo che ha il compito di controllare la genuinità del classificatore. Questo può essere utilizzato per evitare un eccessivo "overfitting" e fermare la formazione del modello in base ad alcuni criteri oggettivi. Con "criteri oggettivi" si intendono criteri che non si basano sugli stessi dati utilizzati durante la fase di training. Se il sistema non soddisfa i criteri previsti, il sistema può essere modificato, dopodichè viene rieseguita la procedura di validazione, finchè non si raggiungono i criteri richiesti (ad esempio il numero massimo di epoche). Sono diverse le procedure di valutazione utilizzabili: Una possibile soluzione è quella di utilizzare un intero dataset creato appositamente per la convalida; questo dataset può essere preparato direttamente dall'utente, o tramite metodi automatici. In altri casi (quando ad esempio il dataset di training ha una dimensione limitata) invece, è possibile utilizzare una tecnica di "validazione incrociata", ovvero, tramite un partizionamento di un campione di dati, in due sotto insiemi dei quali, uno viene utilizzato per la fase di training, mentre il secondo viene utilizzato per confermare e convalidare il classificatore (Mosteller et al. 1968). Esistono diversi tipi di "validazione incrociata" , ad esempio k-fold,leave-one-out...ecc

Possiamo concludere riassumendo che un comune esperimento di classificazione necessita di:

- Un dataset di training per "addestrare" il modello;
- Un dataset di test, che viene utilizzato per ottenere un giudizio finale sulla qualità del classificatore. I dati di questo dataset devono essere dati "nuovi", cioè non precedentemente utilizzati nelle fasi precedenti;
- Un dataset di validazione che può essere fornito dall'utente, estratto casualmente dalla KB, oppure generato automaticamente tramite una procedura di validazione incrociata.

2.2. Regressione

Lo scopo dei metodi di regressione è quello di mettere in evidenza le relazioni esistenti tra le variabili, in particolare quando le relazioni tra le variabili sono "imperfette" (cioè quando non esiste una precisa 'y' per ogni dato 'x').

Il termine "regressione" proviene dalla biologia, e viene utilizzato nello studio delle trasmissione di caratteri genetici di generazione in generazione. Lo scienziato Galton definisce "regressione", o meglio, "regressione verso la media" la tendenza a trasmettere "in media" le caratteristiche genetiche, ma non esattamente sempre nella stessa quantità (Galton 1877). Ma in realtà, cos'è la regressione? E' difficile trovare una definizione precisa. Esistono due significati diversi per il termine "regressione" (Hastie et al. 2005), la quale può essere intesa come una correlazione tra tabelle di dati statistici e come qualità di fitting di una funzione. Per quanto riguarda il primo significato, possiamo proporre un semplice esempio: supponiamo di avere due variabili, x ed y, e per ogni piccolo intervallo di x vi è una distribuzione di y corrispondente. Possiamo sempre calcolare una sintesi dei valori y per quell'intervallo. La sintesi può essere per esempio la media, mediana o la media geometrica. Fissiamo i punti $(x_{i,} \overline{y_l})$, dove

x_i è il centro dell'intervallo i-esimo e $\overline{y_l}$ la media per quell'intervallo. Quindi i punti fissati andranno a cadere nelle vicinanze della curva che li sintetizza, possibilmente vicino ad una linea retta. Questa curva "smussata" si avvicina alla curva di regressione chiamata "regressione di 'y' su 'x'. La generalizzazione di questo esempio si ha quando l'utente ha una tabella (composta da una serie di pattern che provengono da esperienze e osservazioni pregresse) e alcune corrispondenze tra gli intervalli di x (le righe della tabella) e qualche distribuzione di y (le colonne della tabella), rappresentando una correlazione generica non ben conosciuta tra queste.

Una volta che abbiamo una tabella del genere, possiamo chiarire o accentuare la relazione che intercorre tra alcuni valori di una variabile e quelli di altre variabili. Se vogliamo ottenere ad esempio una media, dovremmo analizzare la media o la mediana per ogni colonna. Dopodiché per ottenere una regressione, dovremmo intrecciare queste medie con le medie degli intervalli delle classi.

Seguendo l'esempio precedente, possiamo ricavarci una definizione formale di regressione. In ambito matematico, quando per ogni valore di 'x' esiste una distribuzione di 'y', con densità f(y|x) e un valore di media (o mediano) di 'y' per un 'x' dato da:

$$\int yf(y|x)dy$$

allora la funzione definita da coppie ordinate di dati del tipo ($(x, \overline{y}(x))$) è chiamata regressione di 'y' su 'x'.

Dipendentemente dall'operatore statistico utilizzato, la linea o curva di regressione ottenuta da uno stesso dataset potrebbe presentare piccole differenze.

Alcune volte partendo da note forme funzionali non si ottengono popolazioni continue. Ma I dati possono essere molto grandi (come nel caso dei dati astrofisici). In questi casi è possibile suddividere una delle variabili in intervalli più piccoli e analizzare le medie di ognuno di questi. Poi, senza effettuare grandi ipotesi sulla forma della curva, ci ricaviamo la curva di regressione.

Essenzialmente, quello che la curva fa è dare, quello che chiamano "stima sommaria" delle distribuzioni corrispondenti al dataset delle 'x'.

Inoltre, si può andare oltre, e calcolare differenti curve di regressione corrispondenti ai vari punti di percentuale della distribuzione, ottenendo così una conoscenza più completa del dataset di input. Spesso, si ottengono (ovviamente) informazioni incomplete della distribuzione". Nella prima definizione di regressione, quando i dati sono molto pochi, possiamo verificare che la variazione di campionamento è poco pratica se si vuole ottenere una curva di regressione affidabile tramite il metodo descritto in precedenza (Menard 2001).

Da questa assunzione, ne consegue la seconda definizione di regressione.

Normalmente, è possibile introdurre una procedura di affinamento, applicandola o alla colonna delle somme o sui valori iniziali di 'y' (ovviamente, dopo un ordinamento dei valori di 'y' in termini di valori di 'x' crescenti).

In altre parole, assumiamo una forma per la curva descrivente il dato, ad esempio lineare, quadratica, logaritmica...ecc. Infine andremo ad adattare la curva attraverso un qualsiasi metodo (es. quadrati minimi). In pratica, non si pretende che la curva abbia la stessa identica forma della curva di regressione ottenibile se avessimo un numero di dati infinito, ma semplicemente cerchiamo di ottenere una approssimazione accettabile.

In altre parole, qui intendiamo la regressione di dati in termini di adattamento forzato di una forma funzionale.

I dati reali presentano condizioni intrinseche che fanno si che questa seconda definizione sia quella ufficiale.

Di norma scegliamo una forma della curva con un numero di parametri relativamente piccolo, e quindi dobbiamo scegliere un metodo per adattare al meglio i risultati.

A volte, in diversi manuali è possibile un'ulteriore definizione probabilmente non

formalmente perfetta, ma molto chiara: Effettuare una regressione su una veriabile 'y' su una variabile 'x' significa trovare un vettore per 'x'.

Questo introduce nuovi possibili (e molto più complicati) scenari, nei quali possono essere trovati più di un vettore.

In questi casi, come vantaggio, si ha il fatto che le geometrie possono essere ottenute con tre dimensioni (con due vettori) fino ad arrivare a spazi n-dimensionali (n>3, con più di due vettori).

Possiamo concludere che gli usi della regressione sono:

- Valutazione di caratteristiche sconosciute all'interno del dataset;
- Predizione di fenomeni come quelli meteorologici o astronomici;
- Esclusione: Supponiamo di dare per certo che 'x' influenza 'y', e supponiamo di voler sapere se anche un dato evento 'z' sia associato con 'y' attraverso un possibile meccanismo casuale. In questo caso un approccio potrebbe prendere gli effetti di 'x' su 'y' e vedere se ciò che rimane è associato a 'z'. In pratica, questo può essere fatto da una procedura di adattamento iterativo la quale valuta ad ogni step i residui del precedente adattamento.

Le informazioni date fin'ora non sono sufficienti per descrivere in maniera esaustiva la regressione, ma ha il loro scopo è offrire una semplice comprensione del termine “regressione” e la possibilità di estrarre specifiche di base per la caratterizzazione del caso d'uso in fase di progettazione.

3. Machine Learning

Con il termine ***"Machine Learning"*** si intende una delle discipline fondamentali dell'intelligenza artificiale che si occupa di elaborare algoritmi di ottimizzazione auto-adattivi, ossia in grado di plasmare se stessi a partire da una fase di addestramento su dati noti per lo specifico problema (Bishop 2006). Nel particolare questa scienza prevede che una macchina possa modificare il suo modo di interagire con il mondo esterno grazie all'analisi approfondita di eventi noti. Da questa definizione possiamo dedurre che il ML rappresenta una forma di adattamento analoga a quella che avviene per gli organismi biologici i quali (in tempi ovviamente più lunghi), si adattano all'ambiente in cui si trovano.
Per effettuare effettuare l'apprendimento la macchina ha bisogno di due componenti:

- Un dataset contenente informazioni riguardanti il dominio di applicazione:
- Un algoritmo di apprendimento in grado di estrarre le conoscenze dal dataset in analisi.

Il primo esperimento di ML fu effettuato negli anni '50, dallo scienziato americano "Arthur Lee Samuel",il quale progettò un programma in grado giocare a scacchi ad alti livelli (Samuel, 1990). La cosa sorprendente era che Samuel non era assolutamente un bravo giocatore di scacchi; egli infatti progettò un programma che effettuava centinaia di partite contro se stesso, durante le quali la macchina imparava quali fossero le mosse più o meno opportune da fare durante una partita.

Esistono due tipologie di modelli di ML:

- ***Modelli supervisionati***: Approccio "top-down" applicabile quando è noto il dominio del problema;
- ***Modelli non supervisionati***: Approccio "botton-up" che, diversamente dall'apprendimento supervisionato, costruisce modelli a partire da dati non appartenenti a classi predefinite.

3.1. Modelli supervisionati

Nei modelli di Machine Learning supervisionati abbiamo un set di dati o osservazioni per i quali si conosce l'output desiderato, espresso in termini di classi categoriali, variabili numeriche o logiche, oppure come una generica descrizione di un qualsiasi problema reale. L'output desiderato fornisce infatti un certo livello di controllo perché viene utilizzato dal modello di apprendimento per adattarsi e correggere i propri output affinché questi siano il più corretti possibile. Infine, quando l'algoritmo diventa in grado di predire in maniera corretta le osservazioni, questo verrà definito "classificatore". Alcuni classificatori sono inoltre in grado di fornire risultati in un senso più probabilistico, ad esempio, la probabilità che un dato evento appartiene ad una determinata classe. Normalmente ci riferiamo a questo tipo comportamento del modello con il termine "regressione". Nel diagramma sottostante osserviamo una schematizzazione del lavoro di un modello di Machine Learning supervisionato.

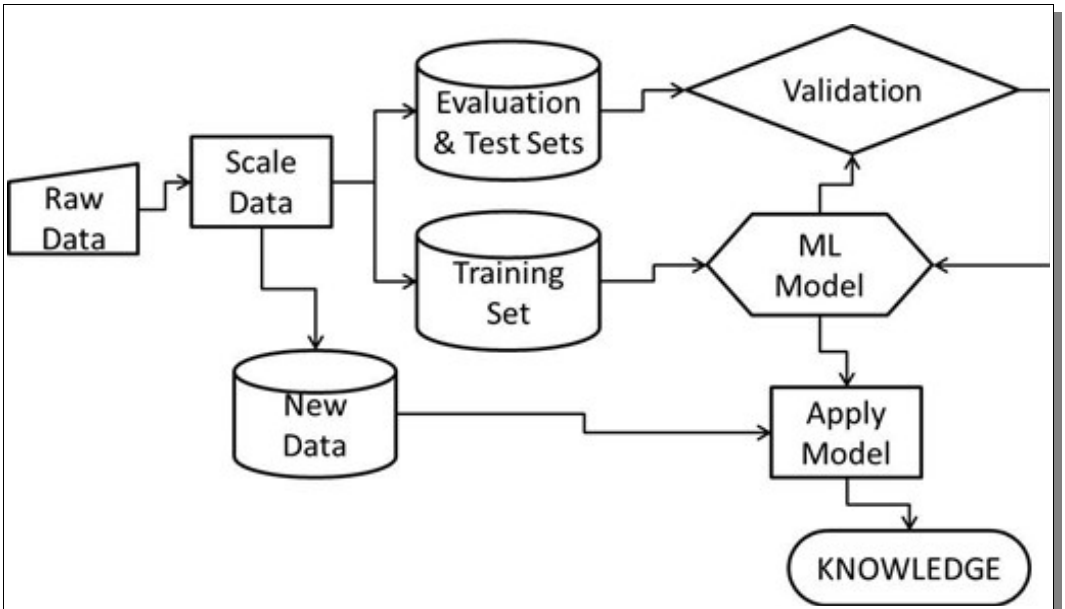


Illustrazione 2: Un workflow basato su un modello di ML supervisionato

Per un apprendimento supervisionato, abbiamo i seguenti step:

- ***Analisi preliminare dei dati:*** Prima di tutto dobbiamo costruire dei

patterns di input appropriati da dare "in pasto" al nostro algoritmo. In questa fase avviene il ridimensionamento e la preparazione dei dati;

- ***Creazione del dataset per il training e la validazione:*** Questa fase è effettuata suddividendo in maniera casuale lo spazio dei data patterns. Il dataset di training contiene i dati utilizzati dal classificatore per apprendere le loro correlazioni interne, mentre il set di validazione viene utilizzato per validare il modello precedentemente addestrato, in modo da calcolare il tasso di errore, che può aiutare l'utente ad identificare le performance e l'accuratezza del classificatore;
- ***Addestramento del modello:*** In questa fase eseguiamo il modello sul dataset di training. Il risultato ottenuto in questa fase è un modello che (in caso di successo) è in grado di prevedere il giusto risultato di un pattern anche quando nuovi dati sconosciuti vengono inseriti;
- ***Validazione:*** Dopo aver creato il modello, bisogna eseguire un test delle sua precisione, prestazioni, completezza e contaminazione. E' molto importante che il dataset utilizzato in questa fase non debba mai essere stato utilizzato in questo modello. Questo è il motivo principale per il quale, nelle fasi precedenti, abbiamo suddiviso il dataset in una parte dedicata al training e una da utilizzare in questa situazione. Lo scopo della validazione è verificare e misurare le capacità di generalizzazione del modello. Se l'errore di classificazione ottenuto durante questa fase è maggiore rispetto all'errore ottenuto nella fase di training, significa che il modello non è stato settato correttamente, e bisogna ritornare alle fase di training e modificare i parametri. Il motivo potrebbe essere che il modelle ha semplicemente memorizzato le risposte vista durante l'addestramento, fallendo così il processo di generalizzazione. Questo è un tipico comportamento di overfitting, e ci sono varie tecniche per superarlo;
- ***Utilizzo:*** Se la fase di validazione è andata a buon fine, il modello è

addestrato a dovere. Possiamo quindi utilizzare il modello per classificare/predire nuovi dati.

3.2. Modelli non supervisionati

Le tipologie di problemi adatte per un modello di ML non supervisionato possono sembrare molto simili a quelli visti nel caso supervisionato, ma in realtà sono molto differenti. Invece di provare a predire un insieme di dati appartenenti a classi già conosciute, con i modelli non supervisionati si cerca di identificare le classi dei pattern a partire esclusivamente dalle osservazioni. In altre parole, la differenza principale tra le due tipologie di modelli è che in quello non supervisionato non viene fornito un dataset di target. Questa è una fondamentale differenza che varia anche il modo in cui l'algoritmo opera. Se quindi da un lato abbiamo algoritmi supervisionati che cercano di minimizzare l'errore di classificazione, dall'altro abbiamo un algoritmo non supervisionato che non beneficia di questi vantaggi, perchè non ha a disposizione alcun valore di "target" sui quali basarsi. Gli algoritmi non supervisionati cercano di creare dei "cluster" (sovra-densità), di dati che hanno caratteristiche simili. In alcuni casi non necessitiamo conoscere cosa rende simili questi dati, ma gli algoritmi sono capaci di trovare relazioni tra questi dati, e raggrupparli in qualche modo. Differentemente dagli algoritmi supervisionati, il cui obiettivo è minimizzare l'errore di classificazione, gli algoritmi non supervisionati cercano di creare gruppi di dati nei quali i punti appartenenti ad un gruppo hanno molte caratteristiche in comune, e cercare di separare il più possibile questi gruppi tra loro (Haykin 1998).

Un'altra differenza fondamentale è che nei problemi non supervisionati, il concetto di dataset di training non viene applicato allo stesso modo che nei modelli supervisionati. Tipicamente abbiamo a disposizione un dataset utilizzato per trovare le relazioni tra i dati ed inserirli in precisi cluster. Un modus operandi

dei modelli non supervisionati è mostrato in maniera schematizzata nel diagramma sottostante.

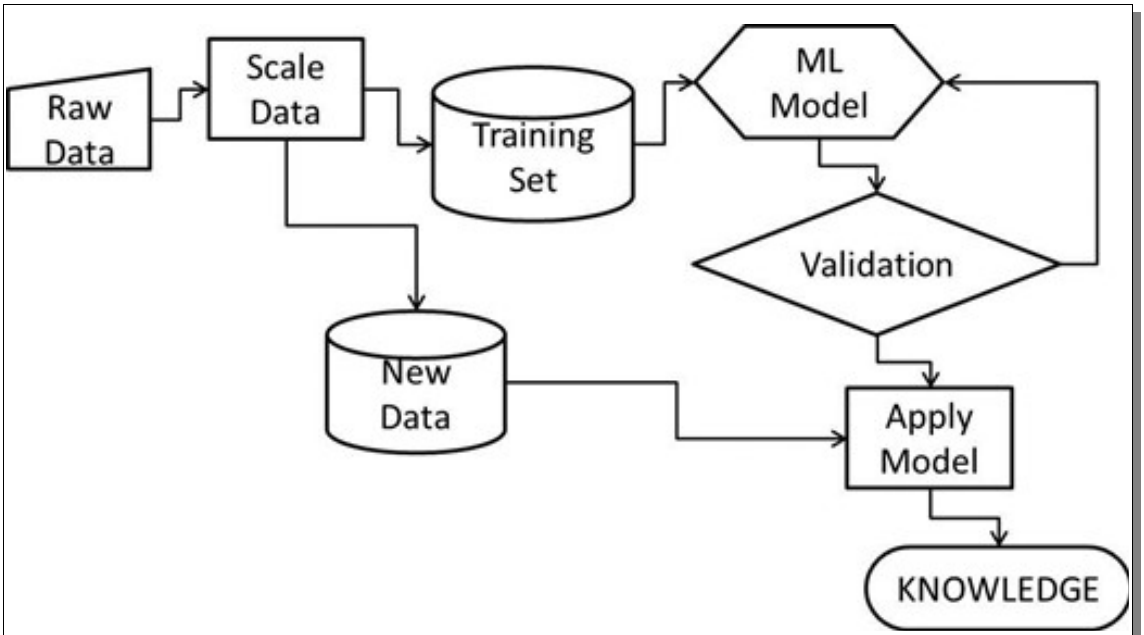


Illustrazione 3: Schematizzazione di un modello di apprendimento non supervisionato

Per un apprendimento non supervisionato, abbiamo i seguenti step:

- **Analisi preliminare dei dati**. Così come accade con i modelli supervisionati, in questo step viene effettuata la selezione delle caratteristiche da dare all'algoritmo, tramite il ridimensionamento dei dati in modo da creare un adeguato dataset di training;
- **Addestramento del modello**. Viene eseguito l'algoritmo non supervisionato sul dataset appena creato in modo da ricavarci i cluster di dati;
- **Validazione**. Dopo aver ottenuto i gruppi di dati, dobbiamo verificare se questi sono stati concepiti nel giusto modo. Per far ciò in questa fase verranno effettuati calcoli di dati statistici sui risultati del modello, così come analisi basate sul dominio di conoscenza, dove si può misurare come certe caratteristiche si comportano quando inserite in un gruppo.

Una volta soddisfatti dei risultati ottenuti tramite la creazione dei gruppi, il

modello può essere utilizzato per l'esecuzione di esperimenti con dati nuovi e mai utilizzati prima durante il training.

3.3. Rappresentazione dei dati per il machine learning

Prima di addentrarci nei meccanismi dei sistemi di Machine Learning, è utile focalizzare la nostra attenzione sui modelli e tipologie di dati offerti in input ai modelli.

Negli esperimenti di Machine Learning, le performance dipendono fortemente da i dati utilizzati durante la fase di training, è quindi la scelta dei dati utilizzati durante l'addestramento è una fase cruciale.

Generalmente, il dataset di input è fornito sotto forma di tabelle o matrici, nelle quali le righe identificano un esempio, mentre le colonne sono i parametri (features), e il loro valore rappresentano gli attributi dei parametri.

Molto spesso le tabelle possono essere composte da valori "vuoti" (matrice sparse) o mancante (mancanza di una informazione per una determinata osservazione). Può inoltre capitare che le informazione di una singola tabella non siano omogenee, ad esempio gli attributi possono essere di diverse tipologie, come numeri e caratteri.

A questa tipologia di diversità si può aggiungere la diversità di formato dei dataset, come ad esempio le tabelle salvate in codice ASCII (ANSI et al. 1977), CSV (Comma Separated Values), (Repici 2010) o FITS (Intestazione testuale seguita dal codice binario di una immagine), (Wells et al. 1981).

Al fine di raggiungere una efficiente e omogenea rappresentazione dei dataset da dare al modello, è obbligatorio effettuare un controllo dei formati dei dati in modo da renderli leggibili e utilizzabili dal sistema.

In altre parole, prima di passare i dataset al modello, dobbiamo trasformare le informazioni dei pattern in modo tale che questi assumano una tipologia di rappresentazione uniforme del dato.

In questo meccanismo le situazioni reali potrebbero essere molto diversi. Basta pensare alle sequenze temporali, dove i dati sono collezionati in una singola e lunga sequenza, non semplicemente divisibili, o ai dati "grezzi", come le immagini ottenute dalle osservazioni astronomiche, le quali possono essere affette da rumori o altri disturbi di fondo.
Questa tipologia di eventi necessita sempre di una fase di "pre-processing", atta alla pulizia e alla preparazione dei dataset in modo tale da renderli utilizzabili per qualunque esperimento di DM. Ovviamente questa fase preliminare deve tener conto anche degli scopi finali dell'esperimento da eseguire sui dati.
In pratica, tenendo presente la tassonomia funzionale descritta nei paragrafi precedenti, ci sono essenzialmente quattro tipologie di apprendimento in relazione con il ML per il DM:

- Apprendimento per associazione;
- Apprendimento per classificazione;
- Apprendimento per predizione (regressione);
- Apprendimento per aggregazione (clustering);

L'apprendimento per associazione, consiste nell'identificazione delle relazioni nascoste che intercorrono tra i dati. Questo non significa identificare l'appartenenza dei pattern a classi specifiche, bensì predire i valori di ogni attributo ricordando il suo valore negli esempi precedenti.
L'apprendimento per classificazione, chiamato più semplicemente "Apprendimento supervisionato", prevede che il training sia sempre effettuato tramite la supervisione di un "oracolo", il quale fornisce sempre il risultato corretto delle osservazioni note. L'efficacia di questa tipologia di apprendimento è valutata solitamente tramite l'utilizzo di un dataset del quale conosciamo i risultati, ma che non è stato mai utilizzato durante la fase di training.
Leggermente differente dalla classificazione è la regressione (Apprendimento per

predizione). In questo caso l'output del modello consiste in valori numeri piuttosto che in classi. La previsione numerica è ovviamente connessa ad un risultato quantitativo, perché il valore predetto è molto più interessante della struttura concettuale dietro il risultato stesso.
L'apprendimento per aggregazione (clustering), infine, è un modello di training utilizzato nel caso in cui non esiste alcuna informazione riguardante le classi di attribuzione.
Il clustering è usato per raggruppare pattern di dati che contengono caratteristiche simili. Ovviamente, la sfida degli esperimenti di clustering è trovare questi gruppi e assegnare ad essi i dati presi in input.

Tornando alla rappresentazione dei dati e alla loro preparazione per i vari modelli, qualunque sia l'analisi funzionale scelta, tutti gli esperimenti richiedono una standardizzazione dei dati. Questo processo, in linea di principio non è sempre possibile. Ad esempio, cosa succede se patterns differenti contengono caratteristiche differenti? Per esempio, nella categoria molto generale degli animali, una caratteristica potrebbe essere il numero di gambe/zampe. Ma questo è una caratteristica tipica delle specie del mondo terrestre…ma non per quelle del mondo marino.
Da un punto di vista pratico, viene utilizzato il valore speciale "irrilevant" o "NULL" come flag per indicare che una caratteristica particolare in alcuni pattern non esiste. Gli attributi di un elemento possono, in linea di principio, assumere sia valori numerici che nominali. Gli attributi nominali assumono valori limitati in un campo ben predefinito di categorie. Al contrario, gli attributi numerici sono semplici misure numeriche (intere o reali) (Witten et. al. 2005).
Questa dicotomia mostra tutti i suoi limiti intrinseci, quando si fa riferimento a problemi reali in cui le funzionalità dei dati devono essere considerate in senso statistico. In questi casi, infatti, dobbiamo trattare valori in termini di "livello di confidenza". Questi possono essere qualitativi simbolici, senza alcuna relazione

tra loro, nei quali non esiste alcuna distanza metrica. Ma in alcuni casi, anche quantità ordinali. Questa categoria si differenzia dai valori nominali, perché rende possibile la classificazione per categoria.

Altre due tipologie di livelli di confidenza sono intervallo e rapporto.

Gli intervalli sono abbastanza intuitivi e semplici da definire, avendo valori ordinati e misurati in una unità fissa e ben identificata.

I rapporti di quantità sono quelli più interessanti e direttamente collegati alle statistiche. Da un punto di vista matematico, i rapporti implicano uno schema di misura che definisce intrinsecamente un punto zero. Ad esempio, quando misuriamo una distanza tra oggetti, la distanza di un oggetto da se stesso è inequivocabilmente zero.

In conclusione, risulta evidente che in tutti i casi gli esperimenti di Data Mining richiedono sempre una profonda analisi e conoscenza dei dati. In caso di grandi dataset, l'utente può avvicinarsi "all'indagine" estraendo casualmente un sottoinsieme di dati e prendendo decisioni sui pattern, sulle caratteristiche, e sui singoli attributi in modo da organizzarli per gli esperimenti futuri.

La presenza di esperti del settore può, naturalmente, semplificare e ridurre il tempo di questa analisi preliminare. Ma in ogni caso, bisogna tener conto del fatto che per l'elaborazione dei dati, un esperimento di Machine Learning impiegherà sempre una quantità considerevole di tempo. Ci sono infine, articoli interessanti che mostrano che almeno il 60% del tempo di una applicazione di Data Mining è dedicata alla preparazione e verifica dei dati (Cabena et. al. 1998)

4. Architetture Computazionali

Con l'aumentare della complessità dei problemi nel calcolo scientifico e nella vita di tutti i giorni, le singole macchine come ad esempio le moderne CPU multi-core, anche se molto performanti, risultano essere insufficienti per l'esecuzione di esperimenti di ML con modelli particolarmente e strutturalmente complessi.

E' proprio da questa necessità che negli ultimi anni si è cercato di implementare

architetture computazionali costituite da reti di singoli calcolatori, all'interno delle quali il lavoro viene distribuito sui singoli nodi.

Oggi giorno le architetture più utilizzate nel calcolo scientifico risultano essere afferenti a tre macro-categorie: Grid, Cloud e HPC (High Performance Computing).

4.1. GRID

L'architettura "GRID" usa singoli computer ed altre risorse interconnesse con l'unico scopo di raggiungere alte prestazioni di calcolo. Questa architettura è stata sviluppata inizialmente nella metà degli anni 90, ma solo negli ultimi anni, grazie alle reti ad alta velocità e ad Internet che consentono interconnessioni anche a grandi distanze, le GRID sono diventate una delle più importanti tecniche computazionali ad alte prestazioni, utilizzate soprattutto per la condivisione di dati scientifici, tecnologici, ingegneristici ed economici.

Grazie all'utilizzo del web che permette reti intercontinentali ad alte prestazioni, sono le architetture preferite dai ricercatori per eseguire collaborazioni collettive atte alla soluzione di un problema di grandi dimensioni (vedi esperimenti del CERN).

I benefici di questa struttura sono i seguenti:

- I problemi che non potevano essere risolti dall'uomo a causa delle limitate capacità di calcolo, possono ora essere affrontati. Un esempio di questi problemi è la comprensione del genoma umano;
- Squadre specializzate, provenienti da ogni parte del mondo, possono cooperare per affrontare problemi che richiedono conoscenze interdisciplinari;
- Se un dispositivo specializzato alla risoluzione di un esperimento viene inserito nella griglia, questo può essere utilizzato da remoto ed in maniera collettiva da chiunque abbia accesso alla griglia.

L'enorme quantità di risorse condivise dalle GRID ha bisogno di regole precise per quanto riguarda:

- le risorse condivise;
- gli utenti autorizzati ad utilizzarle;
- le condizioni sul loro utilizzo.

In risposta a questa necessità, ci vengono incontro le Virtual Organization (o più semplicemente VO).
Le Virtual Organization sono gruppi di persone, la maggior parte delle volte appartenenti ad un unica istituzione, con determinate autorizzazione di accesso alle risorse della GRID; queste possono essere di diverse dimensioni e un utente può appartenere a più di una sola VO nello stesso momento. Per fare un esempio di VO, si può pensare ad un insieme di fisici in diversi campi di ricerca coinvolti nello stesso esperimento.
Per autenticarsi ed accedere alla Grid l'utente deve disporre di alcuni precisi requisiti.
Esistono degli enti di terza parte pubblici o privati (trusted third party), le CA (Certification Authorities), che si occupano di rilasciare un certificato digitale tramite procedura di certificazione che segue standard internazionali. Viene utilizzata la crittografia a doppia chiave, o asimmetrica, in cui una delle due chiavi viene resa pubblica all'interno del certificato (chiave pubblica), mentre la seconda, univocamente correlata con la prima, rimane segreta e associata al titolare (chiave privata). Una coppia di chiavi può essere attribuita ad un solo titolare. L'autorità dispone di un certificato con il quale sono firmati tutti i certificati emessi agli utenti. In aggiunta, ciascuna VO si può dotare di politiche per l'accesso dei propri utenti alle risorse appartenenti a domini differenti.

Un'altro aspetto altrettanto fondamentale nell'implementazione di una GRID è

sicuramente lo scheduling; infatti, mentre nei sistemi tradizionali le risorse e i processi vengono controllati direttamente dallo scheduler del sistema, nelle GRID le risorse sono eterogenee e distribuite su più macchine che gestiscono in locale lo scheduling, i tempi di accesso e la disponibilità delle risorse.

La mancanza di uno scheduler centralizzato e della presenza di utenti che generano processi diversi tra loro, rendono il management del sistema molto più complicato che in sistemi tradizionali. Per buona parte del lavoro di questa tesi, si è utilizzata la GRID del progetto PON S.Co.P.E. Dell'Università Federico II di Napoli.

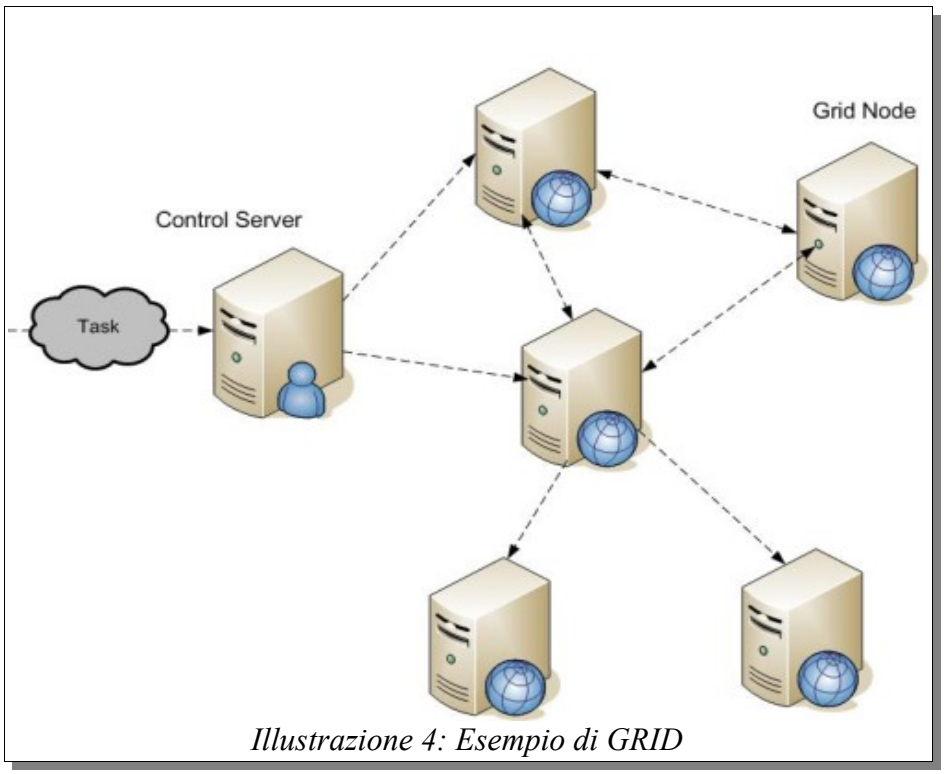


Illustrazione 4: Esempio di GRID

4.2. Cloud

L'architettura Cloud è nata per la distribuzione di servizi (commerciali e non) eterogenei, ed è gestita da un singolo individuo, chiamato “host”, il quale ha il pieno controllo del sistema e del suo hardware. La particolarità del Cloud

Computing è quella di fornire una collezione eterogenea di risorse implementate come un unico grande sistema.

L'utente che vuole usufruire dei servizi di questa architettura ha esclusivamente bisogno di un dispositivo con accesso ad internet, questo perchè i dati e le applicazioni per gestirle sono già situati sul Cloud, ed è il Cloud stesso ad occuparsi dell'esecuzione di questi.

In generale i servizi offerti dal Cloud possono essere suddivisi in tre distinti classi:

- ***Software as a Service (SaaS):*** Nei servizi che ricadono in questa categoria, l'utente acquista una licenza per l'utilizzo di un software presente sul Cloud. Tramite il software acquistato l'utente può utilizzare ed elaborare i dati presenti sul proprio dispositivo. Tramite questa tipologia di servizi l'utente può utilizzare software richiedente grandi risorse computazionali senza dover in alcun modo nuovo hardware per il proprio dispositivo locale. Un chiaro esempio di SaaS è "Office 365", una versione online di Microsoft Office che offre editing di documenti senza dover installare nulla sulla propria macchina;

- ***Data as a Service (DaaS):*** E' una tipologia di servizio molto simile alla precedente, ma con alcune piccole variazioni. I servizi di questa classe si occupano di fornire dati il cui calcolo risulta essere, computazionalmente parlando, complesso e non facilmente raggiungibile utilizzando il semplice hardware del dispositivo dell'utente. Un classico esempio di DaaS è "Google Maps", il quale offre, tramite incroci di dati provenienti da immagini satellitari, mappe digitali, e database di attività commerciali informazioni multiple riguardanti la maggior parte dei luoghi presenti sul globo;

- ***Hardware as a Service (HaaS):*** Questa tipologia di servizio prevede il

"noleggio" di un sistema ad alte prestazioni, al quale, l'utente, può accedervi tramite la rete, e tramite il quale può effettuare calcoli con una rapidità e precisione impossibili con l'hardware a sua disposizione.

- ***Platform as a Service (PaaS):*** Infine questa tipologia di servizio, (a differenza del SaaS che prevede un singolo programma), prevede la fornitura di una intera piattaforma software che contiene applicazioni, librerie e framework utili all'utente. Un esempio di servizio PaaS è "AppScale": una piattaforma che permette all'utente di sviluppare applicazioni scritte per "Google App Engine" direttamente sui propri server.

Il modello di Cloud Computing risulta essere molto diverso da quello della GRID; infatti, nel Cloud, le risorse sono condivise da tutti gli utenti allo stesso tempo, a differenza delle risorse dedicate delle Grid il cui accesso è gestito da un sistema di code). Questo dovrebbe consentire alle applicazioni sensibili alla latenza (es. real time applications) di operare in modo nativo all'interno del sistema, anche se, a causa della crescita esponenziale dei sistemi Cloud e dei loro utenti, garantire un livello di QoS (Quality of Service) sufficiente non è banale, ed è una delle principali sfide di questa architettura.

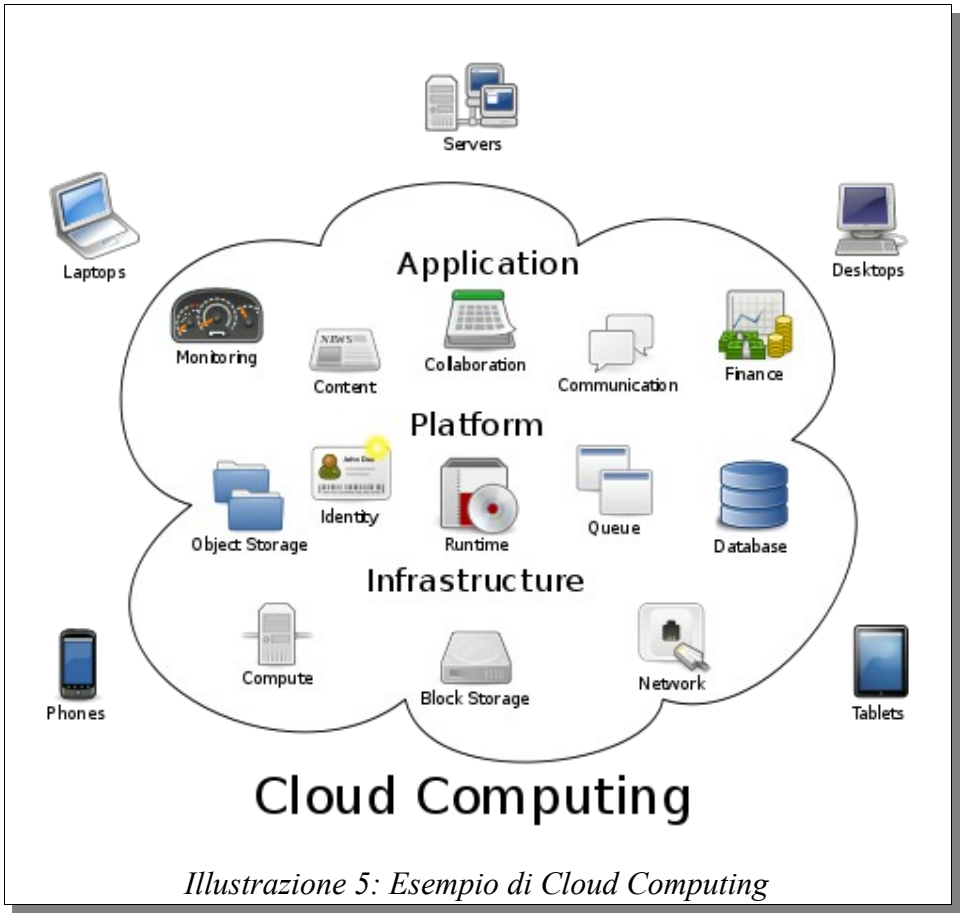


Illustrazione 5: Esempio di Cloud Computing

4.3. HPC

Con HPC (High Performance Computing) si intende una architettura computazionale orientata al clustering atta all'esecuzione distribuita di processi sui nodi del cluster.

Lo scopo di un cluster è quello di distribuire una elaborazione molto complessa tra i vari computer componenti il cluster. In sostanza un problema che richiede molte elaborazioni, per essere risolto, viene scomposto in sotto-problemi separati i quali vengono risolti in parallelo. Questo ovviamente aumenta la potenza di calcolo del sistema. I processi dialogano fra di loro implementando anche il parallelismo.

Gli attuali sistemi di calcolo più diffusi, che sfruttano le tecnologie HPC, sono installazioni che richiedono rilevanti investimenti e la cui gestione richiede

l'utilizzo di personale specializzato di alto livello.

L'intrinseca complessità e rapida evoluzione tecnologica di questi strumenti richiede, inoltre, che tale personale interagisca profondamente con gli utenti finali (gli esperti dei vari settori scientifici nei quali questi sistemi vengono utilizzati), per consentire loro un utilizzo efficiente degli strumenti.

Un esempio di cluster può essere il Beowulf[1]: un sistema multi-computer utilizzato per il calcolo parallelo, e costituito da singoli nodi controllati da un nodo principale il quale si collega a questi tramite cavi Ethernet. Beowulf viene utilizzato all'interno del progetto accademico "Beowulf", e il suo scopo e quello di effettuare elaborazioni parallele di immagini astronomiche. Librerie di programmazione parallela, come MPI e OpenMP, consentono infine la creazione di programmi ad alte prestazioni eseguibili su cluster.

Oggi giorno i clusters vengono utilizzati per numerosi scopi, come criptaggio, simulazioni, decodifiche e, più in generale, in tutte quelle occasioni dove la velocità di calcolo risulta essere un elemento fondamentale del sistema.

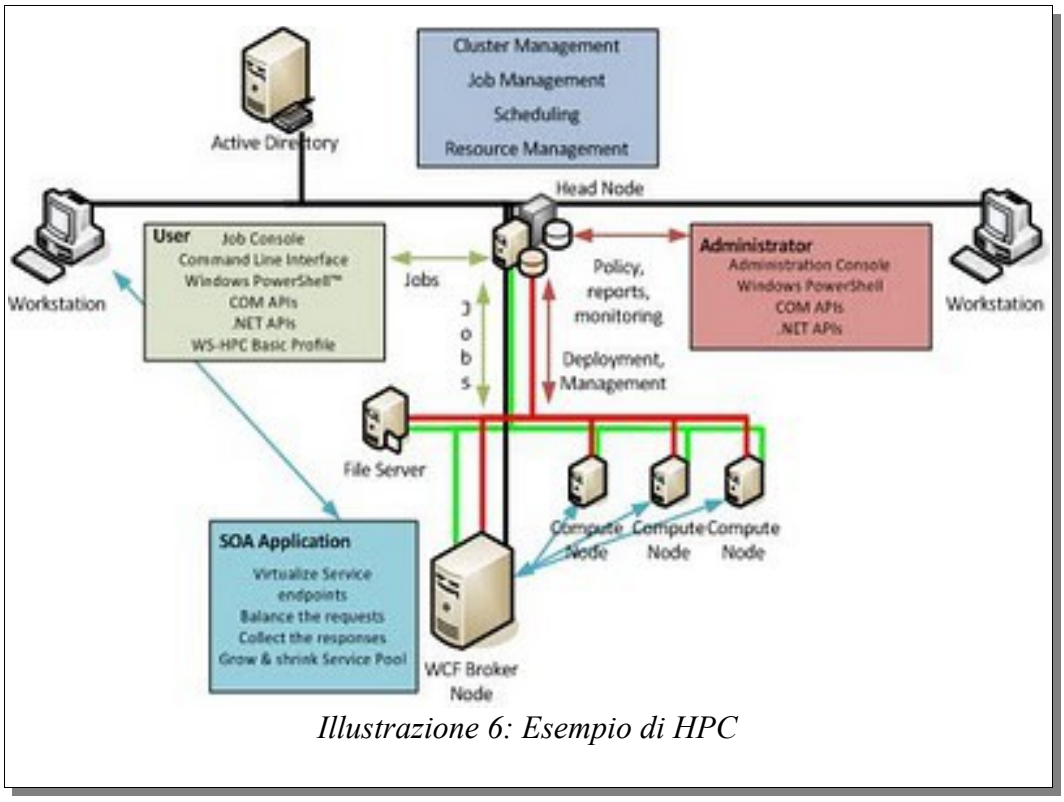


Illustrazione 6: Esempio di HPC

1 http:\\www.beowulf.org

5. Tecniche di calcolo parallelo

Negli ultimi anni a causa dell'eccessiva complessità dei software di nuova generazione (vd. Editor foto/video, renders, programmi di crittografia...ecc) e grazie alla commercializzazione di hardware "multi-core" capace di realizzare operazioni parallele, si è iniziato a prendere in considerazione la possibilità di parallelizzare il lavoro di un software in modo da ridurre (fino a 200 volte in alcuni casi) il tempo di esecuzione di questo.
I linguaggi per la programmazione parallela permettono la scrittura di algoritmi paralleli come insieme di azioni concorrenti eseguite su differenti processori (o cores) e la cooperazione tra due o più azioni concorrenti può avvenire in modi diversi, secondo il linguaggio scelto per codificare l'algoritmo.
In questo testo verranno mostrate le principali librerie utilizzate per la scrittura di programmi paralleli su architetture CPU e GPU.

5.1. CPU: MPI e OpenMP

Per quanto riguarda la programmazione parallela su CPU, i protocolli più utilizzati sono MPI e OpenMP:

MPI (**M**essage **P**assing **I**nterface), è una libreria (implementata in linguaggi come C e Fortran) per la gestione delle comunicazioni tra processi che compongono un programma parallelo su memoria distribuita (Queen, 2004).
Il protocollo prevede l'utilizzo di threads con memoria autonoma che comunicano tra di loro mediante l'utilizzo di messaggi inviati tramite l'uso delle funzioni *send* e *recv*.
Nell'immagine sottostante possiamo vedere un programma "Hello World!" scritto in C tramite l'uso del protocollo MLP, dal quale possiamo ricavarci le prime funzioni MLP :

```c
/* C Example */
#include <stdio.h>
#include <mpi.h>


int main (argc, argv)
     int argc;
     char *argv[];
{
  int rank, size;

  MPI_Init (&argc, &argv);      /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);        /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);        /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```

Illustrazione 7: Codice di "Hello World" per il protocollo "MPI"

Come prima cosa vediamo l'inclusione della libreria “mpi.h” che contiene le dichiarazioni di tutte le funzioni MPI, dopodichè all'interno del main vediamo un trittico fondamentale per la gestione dei thread: ***“MPI_Init”*** ***“MPI_Comm_rank”*** e ***“MPI_Comm_size”;***
Il primo si occupa della generazione dei threads e passa ad ognuno di loro gli argomenti del programma; al termine dell'esecuzione di questa funzione ogni operazione sarà eseguita in contemporanea da ogni thread appena creato.
“MPI_Comm_rank” copia nella variabile “rank” l'ID del thread corrente, mentre *“MPI_Comm_size”* ritorna il numero dei threads del programma.

Una volta terminata l'esecuzione di queste tre funzioni si può iniziare a scrivere un programma parallelo, al termine del quale si possono “uccidere” tutti i thread tramite l'utilizzo della funzione “***MPI_Finalize”.***

Come detto in precedenza, ogni thread ha un proprio spazio di memoria inaccessibile dagli altri processi; dunque per poter comunicare tra loro, i threads hanno bisogno di un canale di comunicazione che in MPI è dato dalle due

funzioni, ***send*** e ***recv*** le quali sono dichiarate in questo modo:

- `MPI_Send(void* data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator)`
- `MPI_Recv(void* data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status*status)`

dove:

- ***"data"*:** puntatore alla variabile da inviare (o ricevere).
- ***"count"*:** dimensione del dato
- ***"datatype"*:** tipo del dato trasferito
- **"*destination*":** ID del thread al quale inviare (o dal quale ricevere) il dato.
- ***"tag"*:** valore che identifica univocamente la trasmissione.
- ***"communicator":***identifica il canale di comunicazione attraverso il quale i threads si scambiano l'informazione (il communicator globale è "MPI_COMM_WORLD").
- ***"status"*:** contiene le informazioni generali riguardanti la trasmissione (mittente,destinatario,esito della trasmissione...ecc)

OpenMP è invece una libreria (implementata per C, C++ e Fortran) che contiene le API per lo sviluppo di applicazioni multi-thread con memoria condivisa (Queen, 2004).

La caratteristica di OpenMP, oltre alla memoria condivisa tra processi, è la sua semplicità d'uso;

Al programmatore che intende parallelizzare una porzione di codice di un programma, basta solo utilizzare la direttiva di preprocessore "*#pragma"* seguita dalle clausole che caratterizzano il tipo di parallelismo (forking); Al termine della porzione di codice scelta dal programmatore, il programma tornerà ad eseguirsi

con un singolo thread (joining).

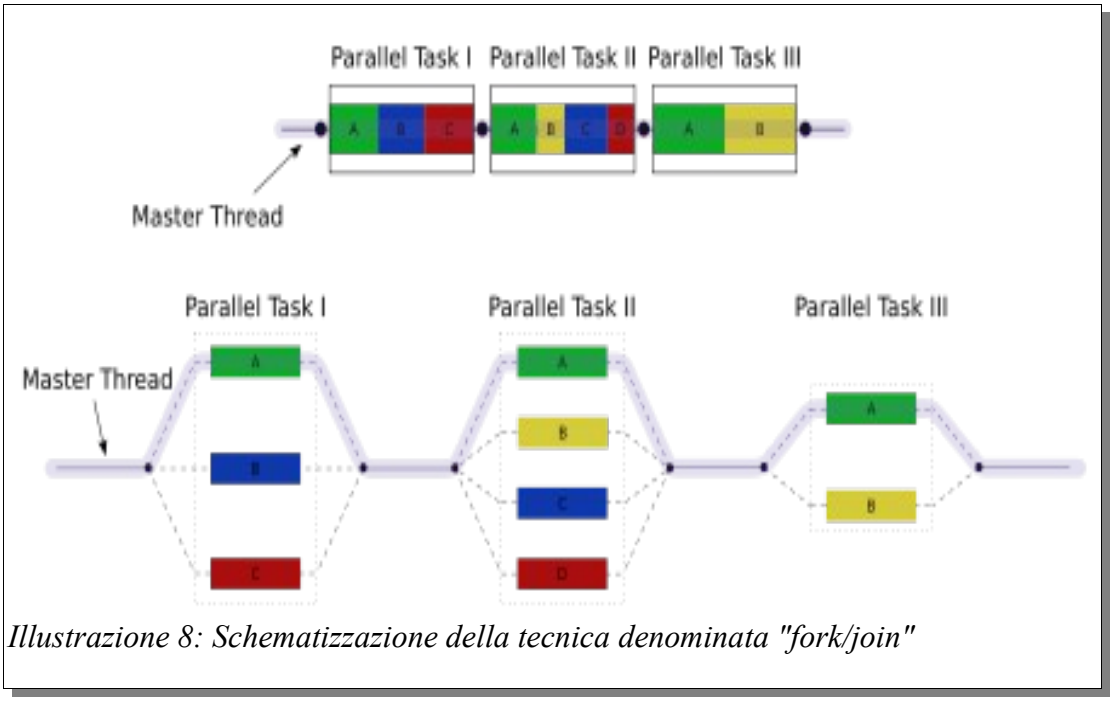


Illustrazione 8: Schematizzazione della tecnica denominata "fork/join"

Così come accade con MPI, anche in OpenMP ogni thread ha un valore identificativo che si può ottenere tramite l'uso della funzione "omp_get_thread_num()).

Di seguito possiamo vedere un Hello World scritto in C dove viene utilizzata la libreria OpenMP.

```c
#include <stdio.h>

int main(void)
{
  #pragma omp parallel
    printf("Hello, world.\n");
  return 0;
}
```

Illustrazione 9: Codice di "Hello World" per il protocollo "OpenMP"

Come possiamo vedere il codice è molto simile al classico "Hello World" per C; l'unica differenza è nella chiamata della direttiva di preprocessore "*#pragma*" il quale, in questa situazione crea un numero prefissato di thread (che può essere definito durante la chiamata del programma tramite variabile d'ambiente, o settato in fase di esecuzione tramite la funzione "omp_set_num_threads") che eseguirà la stampa a schermo della "celebre frase".

Una caratteristica fondamentale di OpenMP è, come detto in precedenza, l'utilizzo delle clausole all'interno della direttiva "#pragma" che hanno il compito di impostare le seguenti caratteristiche:

- La visibilità delle variabili che vengono usate durante la fase parallela (condivise o private);
- La tipologia di sincronizzazione dei threads;
- Lo scheduling che deve essere adottato per gestire i threads;
- Come devono essere gestiti i risultati ottenuti da ogni thread (es. *reduction*).

5.2. GPGPU

Anche se la programmazione parallela su CPU risulta essere nella maggior parte dei casi il miglior metodo di ottimizzazione di un algoritmo grazie anche alla sua semplicità di programmazione, la produzione delle moderne GPU multi-core ha indotto, negli ultimi anni, i ricercatori a sfruttare questa tipologia di architettura per ridurre drasticamente il tempo di calcolo dei propri algoritmi.

Ai giorni nostri infatti, le GPU hanno complesse architetture parallele che permettono il processo di numerosi poligoni in maniera indipendente (Nvidia Corp. 2012). Questi device appartengono a quella categoria di processori che implementano il paradigma delle SIMD (Single Instruction, Multiple Data), e il loro uso, grazie al grande numero di core, permette una elevata velocità di computazione rispetto alle moderne CPU (anche se con qualche limitazione che

vedremo in seguito).

L'ultima generazione di GPU favorisce l'esecuzione di programmi definiti "general purpose" ("a utilizzo generico"), permettendo agli sviluppatori di creare software eseguibile su GPU, da qui l'acronimo GPGPU, ovvero General Purpose computation on Graphic Processing Units.

Per quanto riguarda la tecnologia GPGPU, esistono diversi protocolli per la programmazione su processori grafici, ma in questo testo ci soffermeremo principalmente su quello utilizzato nei nostri esperimenti: CUDA.

5.2.1. CUDA

CUDA (Compute Unified Device Architecture) è l'architettura rilasciata da Nvidia per la realizzazione ed esecuzione di programmi general purpose paralleli su GPU Nvidia.

CUDA è, insieme, un modello architetturale, una Application Programming Interface (API) per sfruttare tale architettura e una serie di estensioni (C for CUDA) al linguaggio C per descrivere un'applicazione parallela in grado di girare sulle GPU che adottano quel modello.

CUDA, lavora sul modello architetturale riportato nell'immagine sottostante:

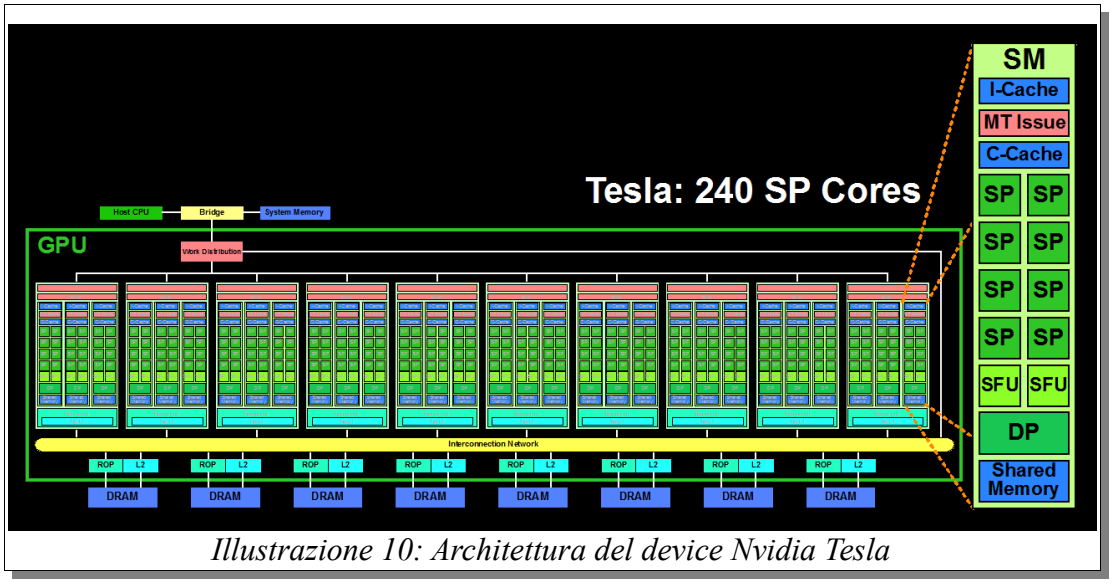


Illustrazione 10: Architettura del device Nvidia Tesla

Ogni GPU è composta da una serie di processori chiamati "Streaming MultiProcessor" (SMP). Il numero di questi SMP è dovuto alla fascia di mercato del chip grafico (es. GeForce 320M 6 SMP, GeForce GTX 560 42 SMP).
Ogni SMP è a sua volta composto da 8 Streaming Processor (SP), i quali fanno capo ad un unica unità di controllo, che ha il compito di decodificare le istruzioni che dovranno essere eseguite dai vari SP. Il fatto che ogni SMP contenga solo una unità di controllo rappresenta un collo di bottiglia non indifferente dell'architettura CUDA. Ci troviamo infatti in una tipica situazione di SIMD (Single Instruction Multiple Data), che impone l'esecuzione della stessa istruzione, con dati diversi, a tutti i thread. Un ulteriore limite dell'architettura CUDA è la "lentezza" dell'unità di controllo nella decodifica di una singola istruzione: Ogni CPU riesce a codificare una istruzione ad ogni ciclo di clock, mentre le unità di controllo delle GPU CUDA riescono a decodificare una istruzione ogni 4 cicli di clock.

5.2.1.1. I kernel

Un programma scritto per CUDA è composto da parti seriali che verranno eseguite dalla CPU (o host), e funzioni parallele denominate kernel, che verranno invece eseguite dalla GPU (o device).

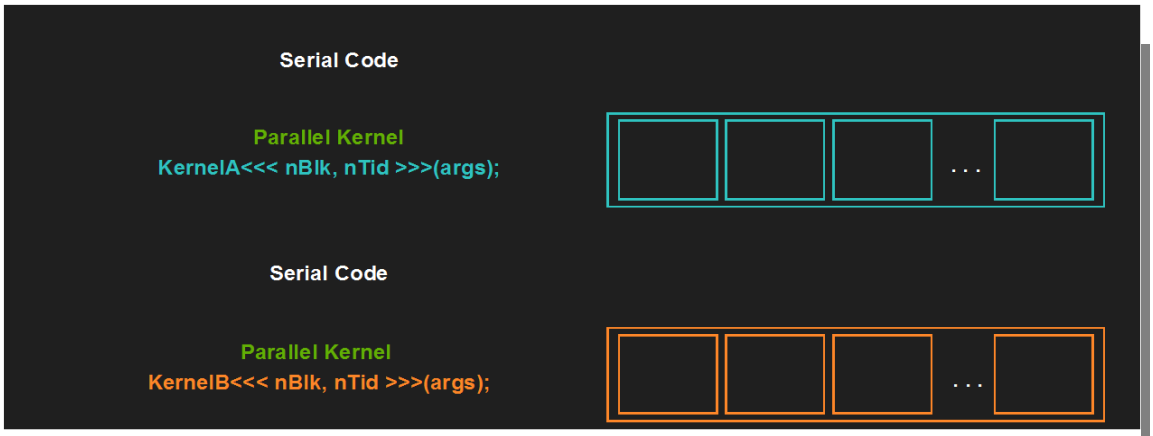


Illustrazione 11: Schematizzazione di un esempio di "fork/join" in ambiente CUDA

Le funzioni Kernel astraggono l'architettura fisica del device e la rendono

unificata e semplificata per il programmatore: Dal punto di vista del programmatore il device viene visto come una griglia composta da singoli blocchi (la dimensione dei blocchi viene impostata dal programmatore), che a loro volta sono composti da threads (il cui numero viene anch'esso definito dal programmatore). Ogni funzione Kernel può essere eseguita in maniera sequenziale su una GPU, mentre i vari threads all'interno della griglia vengono eseguiti ovviamente in parallelo. Il framework CUDA infine rende scalabile l'esecuzione di programmi paralleli in base alle caratteristiche del dispositivo GPU che deve eseguirlo; infatti il numero di threads fisicamente in esecuzione dipenderà esclusivamente dal numero di cores effettivamente disponibili. A causa delle limitazioni di CUDA sopracitate (che non sono le uniche), questa architettura tende a dare il meglio di se solo con un numero di threads molto elevato (sopra le migliaia), questo perché il parallelismo deve sfruttare i tempi morti dell'architettura, dovuti all'elevato tempo di decodifica delle istruzioni, nonché dalla lentezza stessa dei core, il cui clock è estremamente basso se paragonato a quelli delle moderne CPU.

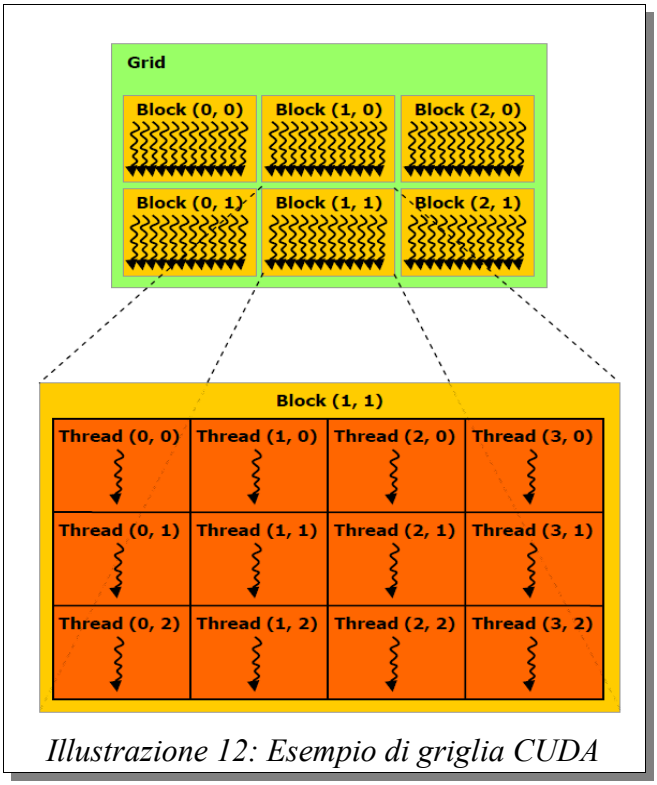


Illustrazione 12: Esempio di griglia CUDA

5.2.1.3. Tipologie di funzioni

CUDA prevede tre tipologie di funzioni alle quali viene assegnato uno specifico qualificatore da anteporre al nome della funzione stessa.

I qualificatori sono:

- ***__global__***: identifica i kernel, cioè le funzioni parallele chiamabili dall'host. Queste funzioni devono avere void come tipo di ritorno, non possono essere ricorsive e non possono utilizzare variabili statiche;

- ***__device__***: identifica una classica funzione C chiamabile dal codice che gira sulla GPU. E' quindi esente dai limiti del kernel, ma invocabile esclusivamente da un'altra funzione __device__ o da un kernel;

- ***__host__***: è una normale funzione C eseguibile esclusivamente sull'host.

I qualificatori __device__ e __host__ possono essere utilizzati insieme per indicare una funzione che viene eseguita sia su CPU che su GPU.

5.2.1.2. Gestione della memoria

Per quanto riguarda la memoria, CUDA la suddivide in quattro tipologie:

- ***Global memory***: contiene variabili condivise da tutti i blocchi e da tutti i threads. E' accessibile anche dall'host (CPU) tramite l'uso di funzioni come cudaMalloc e cudaMemcpy. La lettura e scrittura da questa memoria risulta molto lenta, ma nelle ultime versioni delle GPU NVIDIA il problema è stato attenuato mediante uso intensivo di memoria cache;

- ***Constant Memory***: contiene le variabili delle costanti comuni a tutti i blocchi. E' la memoria dove risiedono gli argomenti del kernel. La velocità di accesso a questa memoria è lenta come quella della global memory, ma

a differenza di questa, possiede sempre una cache di 8KB;

- ***Texture Memory***: è una cache ottimizzata per la grafica 2D. La lettura su questa memoria risulta essere vantaggiosa grazie alla tipologia di indirizzamento e l'interpolazione che può essere utilizzata senza alcuna spesa extra a livello di tempo;

- ***Shared Memory***: contiene le variabili condivise da tutti i threads appartenenti ad un determinato blocco. A differenza della Global Memory, questa è una memoria "On-Chip", e quindi molto più veloce. I dati presenti in questa memoria vengono deallocati appena il kernel termina.

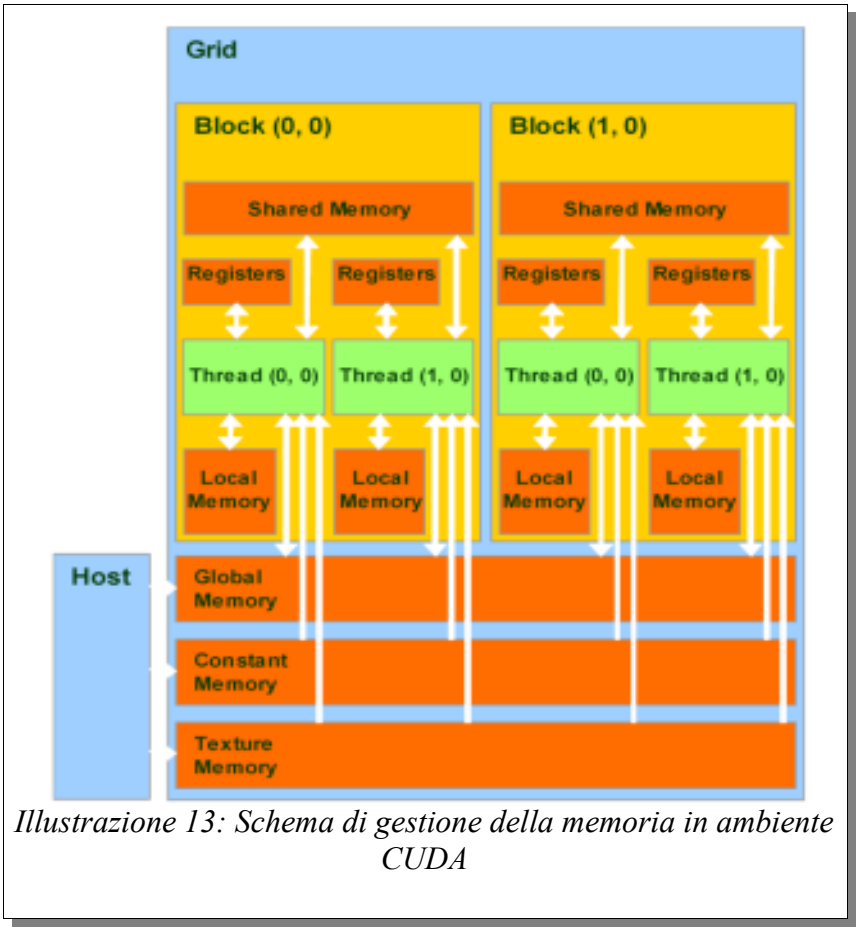


Illustrazione 13: Schema di gestione della memoria in ambiente CUDA

5.2.1.4. Limitazioni

Oltre alle limitazioni precedentemente mostrate, CUDA riporta molte altre limitazioni alle quali Nvidia sta cercando una soluzione, e che elencheremo in questo paragrafo:

- La copia/lettura tra host e device è molto lenta a causa della grande latenza di banda da parte del BUS di comunicazione;

- Durante l'esecuzione del kernel i threads appartenenti ad un blocco vengono inviati ad una Streaming Processor (SP) in blocchi da 32; ciò significa che bisogna eseguire un numero di thread multiplo di 32 se si vogliono ottenere performance elevate;

- Non è permesso l'uso di alcune caratteristiche delle classi C++ (es. membri virtuali);

- La doppia precisione non è assolutamente supportata sui devices con "CUDA Capability" 1.* mentre è in parte supportate (non rispetta lo standard IEEE 754) sui devices con "CUDA Capability" 2.* (questa restrizione sarà rimossa in un prossimo futuro)

Anche se con molteplici limitazioni, l'architettura CUDA risulta essere una delle migliori piattaforme di calcolo sul mercato, e se opportunamente utilizzata, riesce produrre risultati con una velocità centinaia di volte più elevata rispetto alle CPU.

6. Il Modello MLPGA

In questo capitolo si descrive l'architettura e le caratteristiche progettuali del modello ibrido denominato Multi Layer Perceptron con Algoritmo Genetico

(MLPGA). Tale modello si basa sulla coniugazione delle proprietà architetturali della rete neurale MLP con quelle algoritmiche di un algoritmo genetico (GA). Quest'ultimo viene impiegato come sistema di addestramento (nel seguito training) del modello MLP, in modo da aggiornare i pesi della rete neurale. Il GA viene quindi a sostituire il classico algoritmo di training noto come Back Propagation (BP), comunemente usato per la rete MLP (Bishop 2006). Come verrà approfondito nel seguito, l'addestramento con un GA introduce un meccanismo di ottimizzazione più basato su proprietà statistiche che analitiche, fondando il processo di training sul concetto di evoluzione genetica piuttosto che sulle proprietà del gradiente discendente per minimizzare la funzione errore del paradigma supervisionato.

6.1. Architettura MLP

Non esiste una singola e precisa definizione di "**Rete Neurale**", questo perchè è lo stesso concetto di Rete Neurale ad essere molteplice; In ogni caso possiamo ricavarci una definizione abbastanza generalizzata che afferma che: "Una Rete Neurale è un circuito composto da un elevato numero di oggetti elementari che hanno funzione di neurone. Ogni neurone opera su una singola informazione, e il loro lavoro congiunto (ma asincrono) determina l'output della rete."

Esistono diverse versioni di reti neurali che variano a seconda dell'interconnessione tra i neuroni, e le operazioni che i neuroni stessi applicano alle singole informazioni; In questo testo ci occuperemo in particolare del modello di rete neurale che caratterizza MLPGA: il MultiLayer Perceptron (MLP).

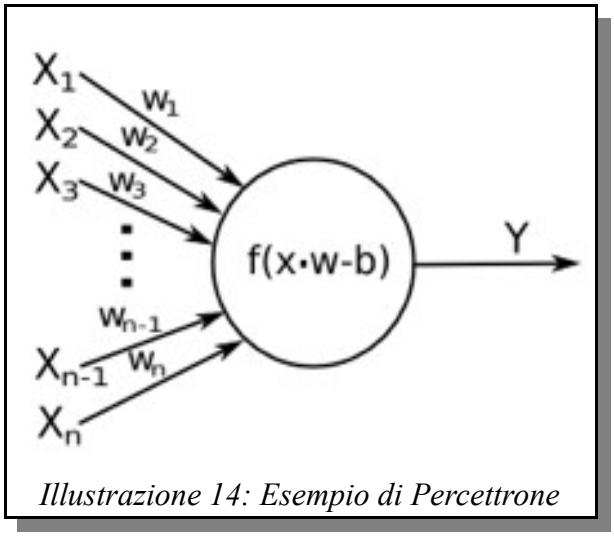


Illustrazione 14: Esempio di Percettrone

Nel 1957, lo studioso Frank Rosenblatt, inventò il ***Percettrone*****:** un modello di rete neurale basato sul paradigma di apprendimento supervisionato del machine learning (Rosenblatt 1957). Questo modello era composto da uno o più "neuroni": unità elementari in grado di elaborare un numero arbitrario di input, restituendo un valore di output.

Anche rappresentando una grande rivoluzione nel mondo del machine learning, questo modello fu messo subito da parte quando si realizzò che il Percettrone fosse in grado di distinguere e classificare esclusivamente classi completamente disgiunte tra loro nello spazio degli ingressi (ad esempio venne dimostrato che funzioni come AND e OR potevano essere facilmente classificate, mentre altre funzioni come XOR risultavano inclassificabili con questo modello).

Per questo motivo il Perceptron fu accantonato per diverso tempo finché, verso la fine degli anni sessanta, dopo diverse ricerche si scoprì che l'utilizzo di due o più livelli di Perceptrons (MultiLayer Perceptron), se opportunamente "addestrati", avrebbe potuto approssimare anche le funzioni prima inclassificabili (Rumelhart et al. 1986) .

Queste nuove scoperte non convinsero però i ricercatori, che ritenevano eccessivamente complessa la fase di training per questa struttura.

Solamente negli ultimi decenni, con l'aumentare della potenza computazionale dei nuovi calcolatori, si è iniziato a rivalutare questo modello, facendolo

diventare uno dei più noti e largamente diffusi. Identicamente alle reti neurali biologiche, anche i MLP sono costituiti da un numero finito di neuroni organizzati in livelli (chiamati layer); inoltre ogni neurone è "completamente connesso" con quelli del layer precedente.

Le reti MLP grazie alla loro versatilità e duttilità possono essere utilizzate per l'approssimazione di qualsiasi funzione a n variabili, e la loro "universalità" viene confermata dal teorema di Kolmogorov (Lorenz et. al.,1996) che afferma appunto che qualsiasi funzione reale continua definita in un sistema n dimensionale può essere rappresentata come somma di funzioni che hanno come loro argomento somme di funzioni continue a variabile singola (successivamente vedremo che la scomposizione in funzioni ad unica variabile è proprio il "modus operandi" del nostro modello).

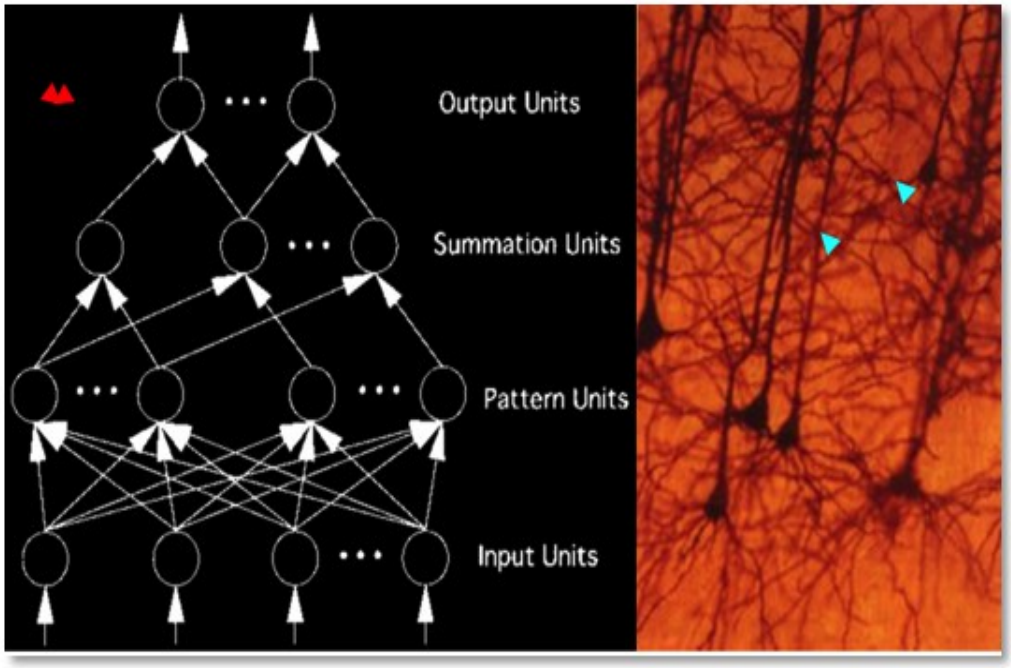


Illustrazione 15: MultiLayer Perceptron e rete neurale biologica a confronto

Così come avviene per la struttura, anche il funzionamento della rete MLP

riprende quello della classica rete neurale biologica. Infatti, così come l'impulso elettrico viaggia lungo gli assoni attraverso i neuroni, così nel MLP l'input (chiamato anche "input layer") viene propagato lungo tutta la rete, passando per ogni neurone, che lo elabora secondo regole che verranno illustrate a breve. L'output dei neuroni di ogni singolo layer verrà poi propagato ai neuroni del layer successivo, o rappresenterà l'output finale nel caso in cui il layer in questione sia l'ultimo della rete (output layer).

La generazione degli output dei neuroni, oltre che dagli output dei neuroni del layer precedente, dipende da altri due fattori altrettanto importanti (se non più importanti): le funzione di attivazione e i pesi.

6.1.1. Funzioni di attivazione

Le funzioni di attivazioni sono dei metodi intrinsechi di ogni neurone e rappresentano un punto focale nella propagazione del dato all'interno della rete neurale. Inoltre la giusta scelta e combinazione di queste può influenzare in maniera rilevante l'output della rete.

Le principali tipologie di funzioni di attivazione (usate nel nostro modello) sono quattro:

- **Funzione lineare**: E' la più semplice funzione di attivazione applicabile ad una rete neurale. La sua formula può essere scritta in questo modo:

  $$y = D * x$$

  dove 'x' è l'input del neurone, 'y' è l'output e 'D' è una costante che nella maggior parte dei casi è impostata a 1. Le funzioni lineari, da sole sono inappropriate per la giusta approssimazione della maggior parte delle funzioni, è dunque consigliato il suo uso insieme con altre funzioni non lineari.

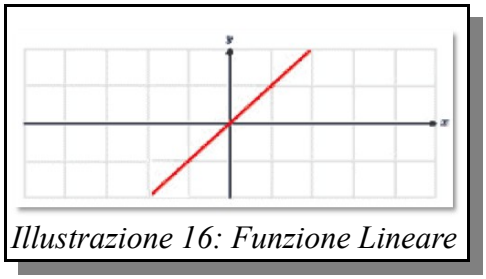


Illustrazione 16: Funzione Lineare

- **<u>Funzione a scalino</u>:** E' la funzione di attivazione che più si avvicina ai comportamenti dei neuroni biologici quando questi vengono a contatto con stimoli esterni. Questa può essere scritta in questo modo:

  y=x (se x>0) y=0(altrimenti)

  Questa tipologia di funzione di attivazione viene scelta per la risoluzione di problemi che prevedono la classificazione tra classi ben definite.

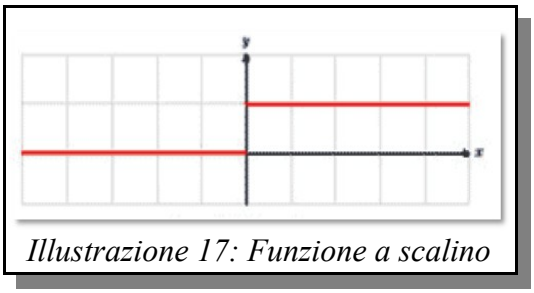


Illustrazione 17: Funzione a scalino

- **<u>Funzione sigmoidea:</u>** E' una delle funzioni più utilizzate nelle reti neurali, ed è caratterizzata da una curva smussata e restituisce valori compresi nell'intervallo [0,1].

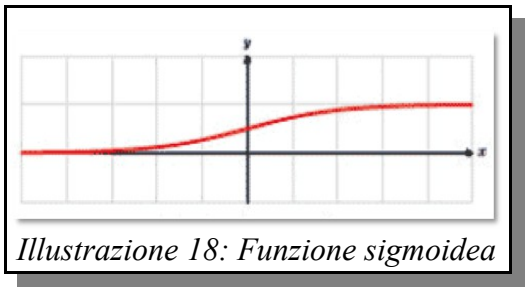


Illustrazione 18: Funzione sigmoidea

- **Funzione tangente iperbolica:** Questa funzione è molto simile alla sigmoidea tranne che per il fatto che restituisce valori inclusi nell'intervallo [-1,1] (può dunque essere usata per l'approssimazione di funzioni i cui input cadono esclusivamente all'interno del suddetto intervallo).

$$y = \tanh(x)$$

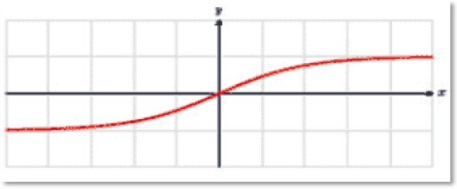


Illustrazione 19: Funzione tangente iperbolica

6.1.2. Pesi

I pesi di una rete MLP sono valori numerici normalizzati, solitamente compresi nell'intervallo [-1,1] e costituiscono un punto fondamentale, se non il più importante nel funzionamento di una rete MLP. Questi valori caratterizzano gli archi di collegamento tra due neuroni ed hanno il compito di calibrarne l'output. Il ruolo dei pesi è quello di fungere da "filtro di comunicazione" tra i neuroni, e osservando l'immagine sottostante possiamo intuire come vengono utilizzati questi "filtri": I valori di output dei neuroni del layer 0 vengono propagati al neurone del layer 1 e durante la propagazione ogni valore viene moltiplicato per il peso dell'arco che unisce i due neuroni.

Il valore di output del neurone nel layer 1 sarà dunque il valore di output della propria funzione di attivazione il cui input sarà dato dalla sommatoria degli input ricevuti dal layer precedente.

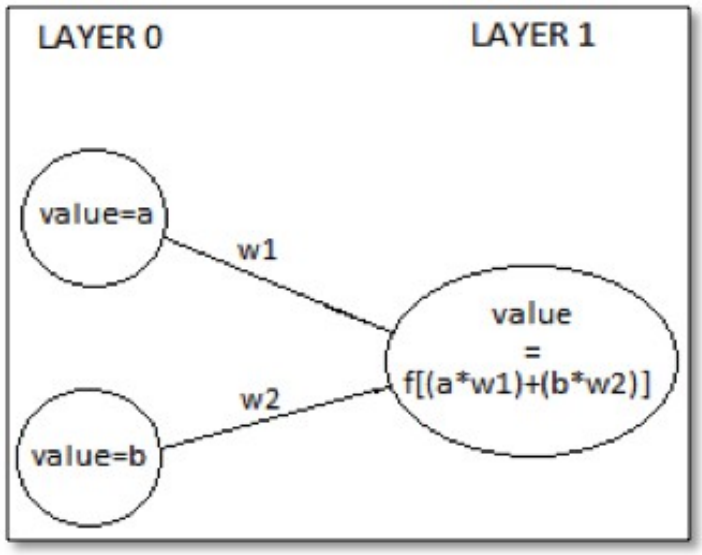


Illustrazione 20: Esempio di propagazione dell'informazione tra layers

6.2 Apprendimento con algoritmo genetico

L'algoritmo genetico è un metodo di computazione ed ottimizzazione dei dati ispirato al processo evolutivo darwiniano. Grazie alle sue capacità "evolutive", questo algoritmo è molto spesso utilizzato in modelli di data mining, o più in generale per determinare la soluzione di un problema quando il suo codominio non è noto a priori. Come detto in precedenza, questo algoritmo di ottimizzazione si basa sul modello evolutivo presentato da Charles Darwin nella sua opera "*L'origine delle specie"* nel 1859 (Darwin 1859). Darwin affermava che gli individui di una popolazione evolvono a causa di due semplici fattori: il caso e la necessità. E' infatti noto che si hanno delle piccole mutazioni naturali casuali dovute a piccoli errori genetici durate la riproduzione, ed inoltre le necessità ambientali (clima,alimentazione..ecc) effettuano una naturale selezione degli individui con migliori capacità di sopravvivenza.

Esempio: L'esempio più classico è il collo della giraffa: Per Darwin infatti gli

antenati delle giraffe erano delle semplici antilopi con il collo corto che pascolavano semplicemente l'erba dei prati. Con un aumento esponenziale di questi mammiferi il cibo a disposizione per questa specie iniziò a scarseggiare, e diventò sempre più difficile procurarsi del cibo; nel contempo a causa di casuali errori genetici avvenuti durante la riproduzione, nasce un antilope con il collo più lungo. Nella competizione del cibo questo nuovo esemplare sarà è molto più avvantaggiato, perché può mangiare le foglie più alte, li dove le altre antilopi non arrivano. Essendo il collo lungo un caratteristica vantaggiosa essa si trasmette ai figli creando così in breve tempo un gruppo sempre più esteso di esemplari dal collo lungo, andando a soppiantare la precedente popolazione dal collo corto, la quale a causa della propria incapacità di arrivare al cibo è destinata all'estinzione.

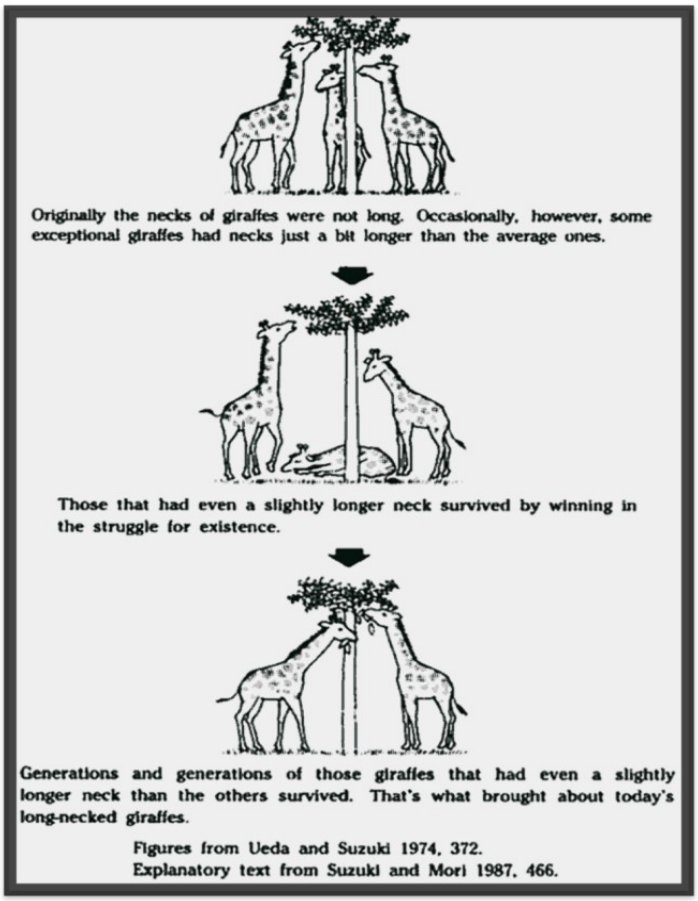


Illustrazione 21: L'esempio dell'evoluzione delle giraffe secondo Darwin

Così come accade con l'evoluzione biologica proposta da Charles Darwin, anche il modello proposto prevede una popolazione composta da esemplari, chiamati cromosomi, che evolvono durante tutta l'esecuzione dell'algoritmo. Ogni cromosoma è costituito a sua volta da un set di "geni" che costituiscono il DNA del cromosoma stesso.

I DNA costituiscono il componente cardine dell'algoritmo genetico; ogni singolo DNA rappresenta infatti un possibile risultato finale, ed è proprio la modifica dei singoli geni all'interno dei DNA a determinare l'evoluzione degli individui della popolazione.

6.2.2. Funzionamento

Il funzionamento dell'algoritmo genetico è sostanzialmente molto semplice, e può essere sintetizzato in quattro fasi:

- **L'inizializzazione della popolazione:** Il primo passo che viene fatto nell'utilizzo dell'algoritmo genetico consiste nella generazione casuale di una popolazione di cromosomi usando ad esempio un qualsiasi metodo di distribuzione statistica;

- **Selezione degli output:** Durante questa fase sono analizzati i DNA di ciascun cromosoma tramite un modello di calcolo (nel nostro caso il MultiLayer Perceptron), ricavando un valore di output;

- **Confronto dei risultati:** In questa fase vengono calcolate le differenze tra i risultati ottenuti tramite i DNA dei cromosomi, e il risultato aspettato. Il valore ottenuto è chiamato "*fitness*";

- **Evoluzione degli individui:** Durante questa fase vengono applicate (secondo regole che verranno spiegate in seguito) i cosiddetti "*operatori genetici"*, cioè funzioni che hanno lo scopo di modificare i DNA dei cromosomi. Al termine di questa fase otteniamo una nuova popolazione nata dall'evoluzione della precedente.

La seconda, terza, e quarta fase saranno eseguite ciclicamente sulle varie generazioni della popolazione, ed ogni ciclo è chiamato "*epoca"*.
L'algoritmo genetico potrebbe essere eseguito all'infinito, ma di norma si predispone che questo termini quando uno dei seguenti eventi si verifica:

- Si trova un cromosoma il cui fitness è minore o uguale un valore predeterminato.
- Si è raggiunto il numero massimo di epoche.

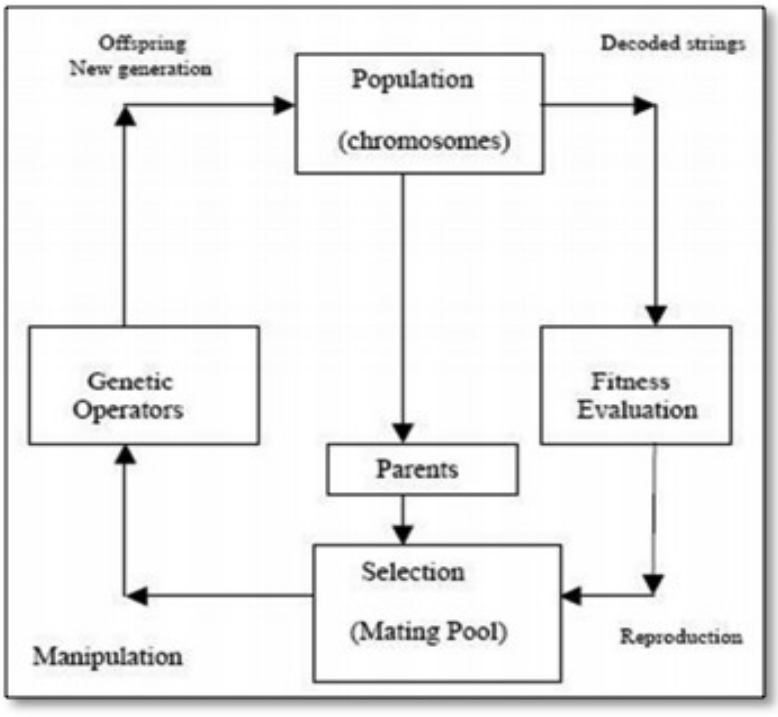


Illustrazione 22: Schematizzazione di un algoritmo genetico

6.2.3. Operatori genetici

Come detto in precedenza le operazioni di modifica dei DNA avviene tramite particolari funzioni chiamate "operatori genetici".

L'algoritmo genetico prevede due tipi di operatori:

- *Il crossover*
- *La mutazione*

La funzione di ***crossover*** riprende il concetto di "crossover biologico" che prevede la generazione di un nuovo individuo partendo da un mix dei geni provenienti dal DNA di due individui diversi (chiamati "*genitori*").

Nell'algoritmo genetico il crossover può essere implementato in diversi modi, tra i principali ci sono:

- **Single point crossover**: risulta essere il metodo di crossing-over più semplice, e consiste nel scegliere due individui, tagliare il loro DNA in un punto a caso chiamato "*crossing point*" (il punto deve essere lo stesso), e creare un nuovo individuo che abbia la "testa" del primo individuo, e la "coda" del secondo.

  Nell'immagine sottostante viene mostrato un esempio in cui il "crossing point" è uguale ad 1.

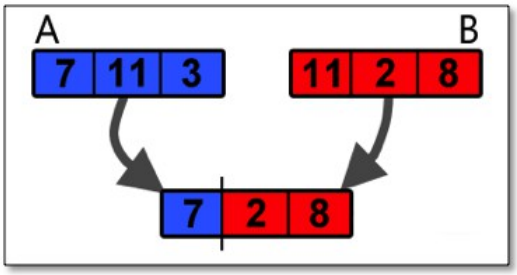


Illustrazione 23: Esempio di crossover "single point"

- **Double point crossover:** Questo metodo è una piccola variante della versione "*Single point crossover*" mostrata in precedenza; l'unica differenza consiste nel numero di *crossing point* che passa da uno a due.
  Nell'immagine sottostante è stata utilizzata la tecnica del Double point crossover in cui i due "crossing point" sono impostati rispettivamente a 1e2.

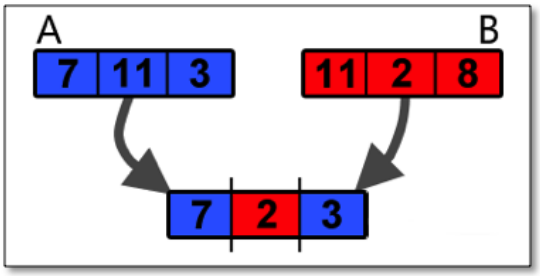


Illustrazione 24: Esempio di crossover "double point"

- **Mixing Crossover:** E' un'evoluzione delle precedenti tecniche, e nasce dalla consapevolezza statistica che è molto improbabile che, all'interno di un DNA, due geni "utili" si trovino in posizioni contigue. La tecnica proposta, infatti, invece di selezionare segmenti di geni contigui, prevede il DNA risultante sia la copia esatta di uno dei due genitori al quale sono stati modificati n geni (il numero viene scelto casualmente) scelti a caso. Nell'esempio riportato in basso viene illustrata il meccanismo della tecnica appena spiegata.

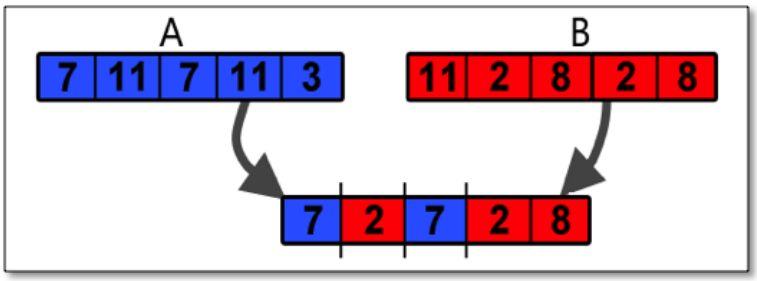


Illustrazione 25: Esempio di Mixing Crossover

A differenza del *crossover* che prevede l'utilizzo di due individui genitori, la ***mutazione*** invece esegue una piccola modifica di uno (o più) geni all'interno del DNA sottoposto. Anche se negli anni sono state implementate tantissime versioni di questo operatore, in questo testo verrà illustrata esclusivamente la più semplice e più usata versione di mutazione: la "single gene unbiased mutation" (conosciuta anche come **mutazione standard**).

La *mutazione standard* prevede la selezione di un gene casuale all'interno del DNA in analisi e la sua sostituzione con uno generato tramite un generatore di valori casuali.

Illustrazione 26: Esempio di Mutazione

6.2.4. Regole dell'evoluzione

Una volta definiti gli operatori genetici, bisogna ora vedere come questi vengono utilizzati all'interno del modello.

Le regole che gestiscono le modalità di applicazione del crossover e della mutazione sono molteplici, ma in questo testo descriveremo esclusivamente le principali che sono:

- Fitting
- Roulette
- Ranking

Prima però di introdurre queste regole bisogna prima chiarire un concetto comune a queste tre regole: **l'elitismo**

L'*elitismo* si basa sul concetto di "élite" ed è una strategia evolutiva che va a completare il processo di sostituzione della popolazione iniziato dall'operatore selezione. Quando creiamo una nuova popolazione con crossover e mutazione

abbiamo una grande probabilità di perdere il miglior cromosoma. L'elitismo semplice è un metodo che prima copia il miglior cromosoma (o i pochi migliori) nella nuova popolazione e poi prosegue con la logica espressa in precedenza. L'elitsmo può far crescere rapidamente le performance dell'algoritmo genetico, perché evita la perdita della migliore soluzione trovata.

Il **fitting** è una delle più semplici regole ed al contempo riesce a generare risultati abbastanza soddisfacenti.

Il suo modus operandi è il seguente:

- Considera "*elitico*" il 10% della popolazione con il fitness più basso
- Il resto della popolazione viene incrociato (crossover) con uno degli individui elitici preso a caso, dopodiché, al risultato di questo incrocio viene applicato l'operatore di *mutazione*.

Roulette è un metodo di selezione proporzionale al valore di fitness che prevede che gli accoppiamenti tra gli individui della popolazione avvengano tramite l'utilizzo di una roulette (virtuale) che restituisce un altro individuo della popolazione secondo due fattori precisi:

- Il fitting dell'individuo
- Il caso

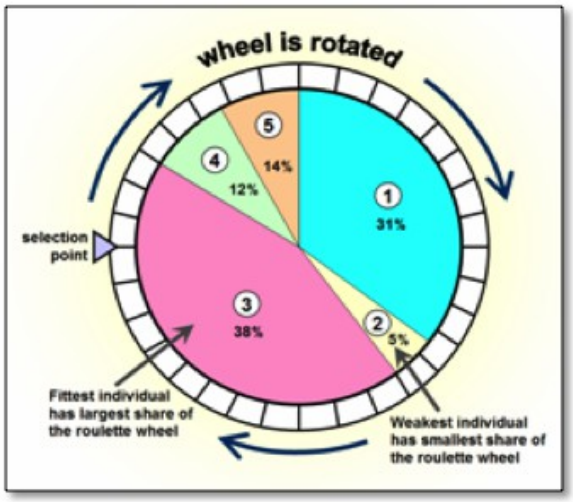


Illustrazione 27: L'algoritmo della Roulette

Ad ogni elemento della popolazione vengono assegnate un determinato numero di caselle della ruota, che è dovuto dalla sua fitness e da quelli degli altri individui.

La ruota assegnerà quindi un maggior numero di caselle all'elemento della popolazione con il valore di fitness migliore, ed il minor numero di caselle (che può essere anche zero!) all'elemento con la peggior fitness.
In breve questa regola effettua i seguenti passi:

- Considera "elitico" i primi x elementi con il valore di fitness più basso (il valore 'x' viene impostato in fase di 'starting' dell'algoritmo)
- Il resto della popolazione viene incrociato (crossover) con l'individuo risultate dall'utilizzo della ruota della roulette, dopodiché, al risultato di questo incrocio, viene applicato l'operatore di *mutazione*.

Infine il metodo **ranking** prevede semplicemente accoppiamenti tra i vari individui della popolazione e l'elemento con il fitness più basso.

3.1.3. Modello ibrido: MLPGA

Tenendo presente le considerazione fatto per il MultiLayer Perceptron e per l'Algoritmo genetico, mostriamo ora come questi vengono combinati insieme ottenendo così il modello **MLPGA.** In pratica i DNA dei cromosomi della popolazione rappresentano i possibili pesi della rete neurale, e l'evoluzione di questi cromosomi ha lo scopo di trovare il set di pesi (il DNA) che, innestato all'interno della rete, restituisce un output con il minor errore possibile.

L'esecuzione di questo algoritmo comporta i seguenti passi:

- Generazione di una popolazione random di cromosomi;
- Viene eseguita la "Forward propagation" su tutti i patter del dataset preso in input generando un risultato di output. Questo passaggio viene ripetuto per ogni set di pesi (ogni singolo cromosoma della

popolazione);

- Calcolo dell'errore di training;
- Generazione di una nuova popolazione di cromosomi applicando gli operatori genetici e i metodi precedentemente mostrati;
- Ripetizione dei passi da 2 a 4 finchè non si verifichi un evento di terminazione (si è raggiunto il numero massimo di *epoche* oppure si è trovato un cromosoma con un fitness minore di un valore inizialmente settato).

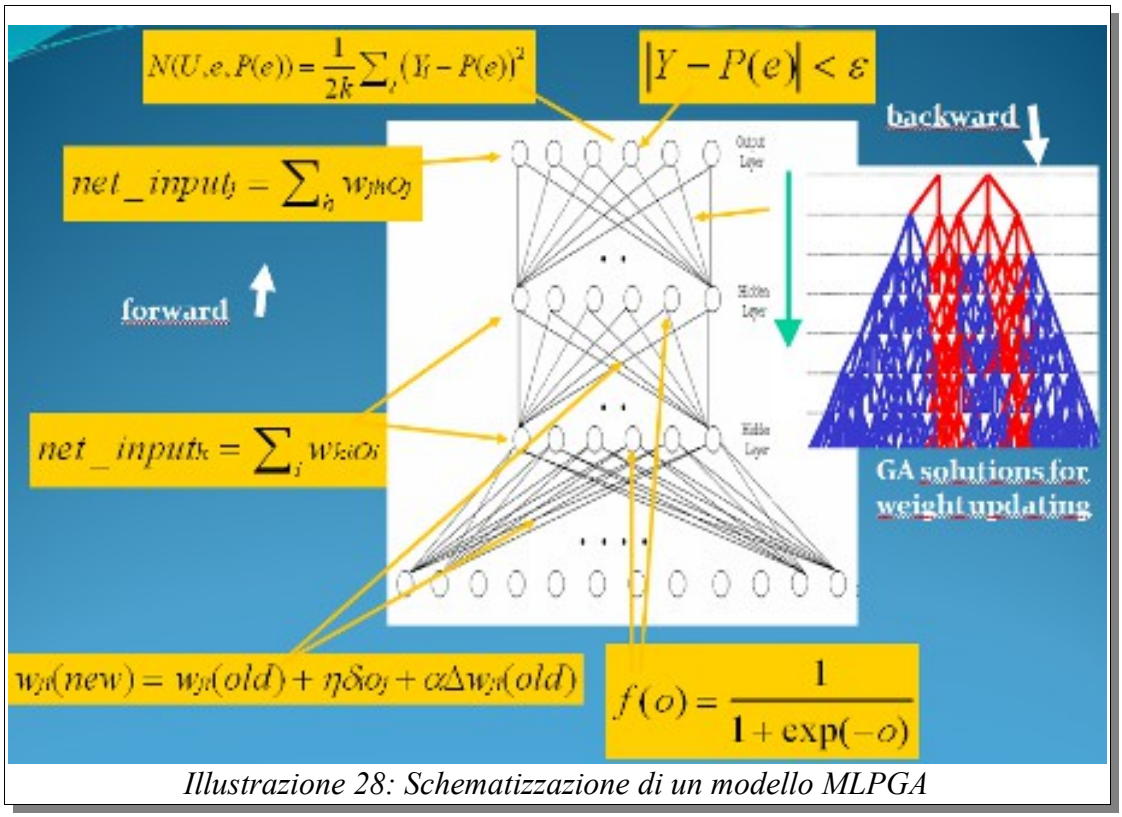


Illustrazione 28: Schematizzazione di un modello MLPGA

6.3 Implementazione seriale in C

L'implementazione del modello è stata realizzata tramite determinate strutture dati atte all'emulazione della morfologia del problema in analisi.

Andando nel particolare possiamo suddividere l'implementazione in due sotto sistemi: il sottosistema "MLP" e il sottosistema "GA".

6.3.1. Le strutture del MLP

Il sottosistema MLP si occupa appunto dell'emulazione del MultiLayer Perceptron, ed è costituito dalle seguenti strutture:

- Neuron
- NeuralLayer
- MLPNet

La loro cooperazione è illustrata nel seguente Class Diagram:

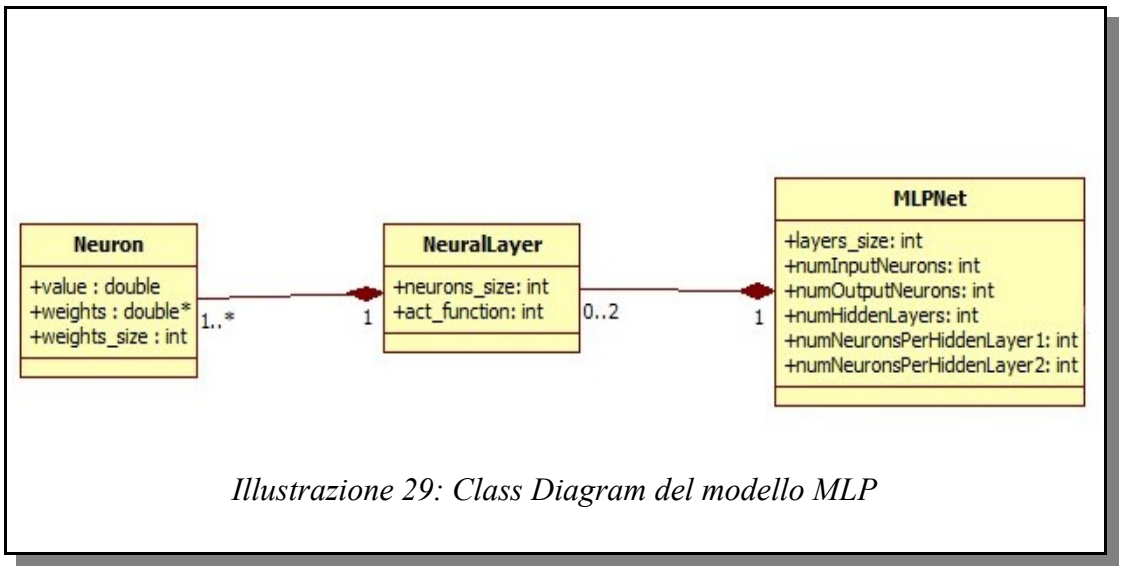


Illustrazione 29: Class Diagram del modello MLP

Scendendo ancora più nel dettaglio, andiamo a esaminare le singole struttura.

La struttura dati base, sulla quale si base il modello MLP, è *"**Neuron**"*; questa emula un singolo neurone ed è composto dalle seguente variabili:

- ***value:*** rappresenta il valore di output del neurone;
- ***weights:*** è un array contenente i pesi degli archi che collegano il neurone in questione con il neurone del layer precedente;
- ***weights_size***: contiene il numero di pesi contenuti nell'array *weights*.

Come detto in precedenza, ogni neurone è parte di un livello neurale (neural layer) il quale viene emulata dalla struttura "***NeuralLayer***" che è composta dalle seguenti variabili:

- ***neurons:*** è un array contenente i puntatori a tutti i neuroni del layer in questione;
- ***neurons_size:*** rappresenta la dimensione del layer (il numero di neuroni contenuti al suo interno);
- ***act_function:*** contiene l'identificativo della funzione di attivazione per tutti i neuroni contenuti nel layer.

Infine tutti i layer costituiscono la rete finale che viene rappresentata dalla struttura "***MLPNet***", composta dalle seguenti variabili:

- ***layers:*** è un array contenente i puntatori a tutti i layer (variabili NeuralLayer) della rete;
- ***layers_size:*** rappresenta il numero di layer di cui è composta la rete;
- ***numInputNeurons:*** rappresenta la dimensione del layer di input (l'input della rete);
- ***numOutputNeurons:*** rappresenta la dimensione del layer di output (l'output della rete);
- ***numHiddenLayers:*** numero di layer interni delle rete;
- ***NumNeuronsPerHiddenLayer1:*** numero di neuroni contenuti nel primo layer interno;
- ***NumNeuronsPerHiddenLayer2:*** numero di neuroni contenuti nel secondo layer interno;

6.3.2. Le strutture del GA

Il sottosistema "GA" emula il modello dell'algoritmo genetico, ed è costituito dalle seguenti strutture:

- Chromosome
- Population

e la loro (semplice) cooperazione è illustrata nel seguente Class Diagram:

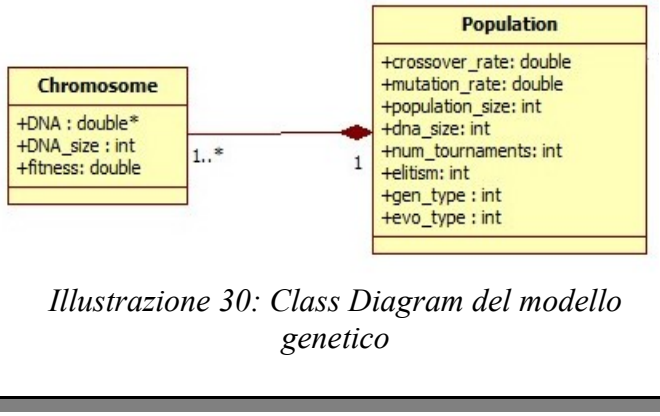


Illustrazione 30: Class Diagram del modello genetico

Esaminiamo ora le due strutture in analisi:

La struttura "***Chromosome***" rappresenta il singolo individuo della della popolazione, e le sue variabili interne sono:

- ***DNA:*** è un array e contiene (come suggerisce il nome) l'intero DNA dell'individuo;
- ***DNA_size:*** indica la dimensione del DNA;
- ***fitness:*** rappresenta l'errore di training dell'individuo (cioè quanto il DNA dell'individuo diverge dal risultato ottimale).

Così come spiegato in precedenza, gli individui (Chromosome) sono organizzati in una popolazione che viene rappresentata all'interno del sistema dalla struttura "***Population"***, la quale è composta dalle seguenti variabili:

- ***popv:*** è un array contenente i puntatori agli individui della popolazione (Chromosome);
- ***crossover_rate:*** rappresenta la probabilità di esecuzione dell'operatore di crossover durante l'evoluzione della popolazione;
- ***mutation_rate:*** rappresenta la probabilità di esecuzione dell'operatore di mutazione durante l'evoluzione della popolazione;
- ***population_size:*** indica la dimensione della popolazione, cioè il numero di individui all'interno della popolazione;
- ***dna_size:*** contiene la dimensione del DNA degli individui;
- ***num_tournaments:*** numero di individui concorrenti per il modello evolutivo "RANKING";
- ***elitism:*** numero di individui "elitici", ovvero i migliori individui destinati a passare intatti alla nuova popolazione;
- ***gen_type:*** contiene l'identificativo della funzione di generazione utilizzata per la creazione della popolazione iniziale;
- ***evo_type:*** contiene l'identificativo del metodo di evoluzione (FITTING, RANKING...ecc).

3.3. La struttura "MLPGA_Ctrl"

I due sottosistemi appena mostrati costituiscono i componenti del modello MLPGA che viene rappresentato dalla struttura dati denominata MLPGA_Ctrl che ne facilità la gestione.

La cooperazione dei sottosistemi mostrata nel seguente Class Diagram:

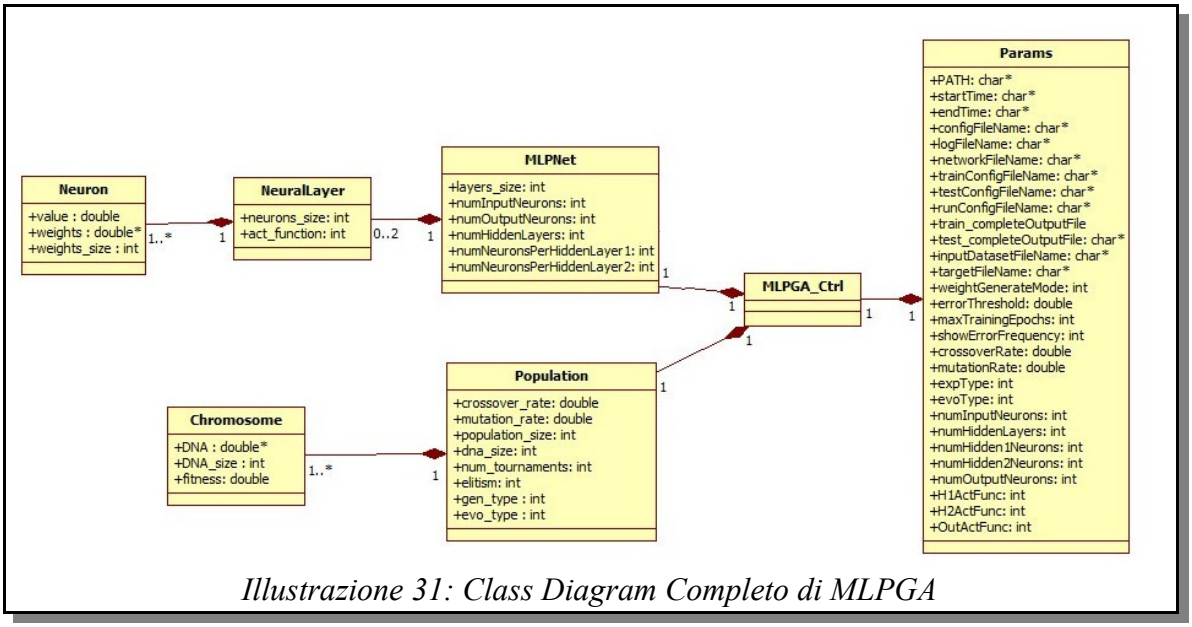


Illustrazione 31: Class Diagram Completo di MLPGA

Le variabili che costituiscono questa struttura sono:

- ***Net:*** un puntatore alla rete MLP;
- ***Pop:*** un puntatore alla popolazione corrente;
- ***Par:*** un puntatore ad una variabile di tipo "***Params***", una semplice struttura il cui unico scopo è quello di conservare i dati provenienti dai parametri di input.

6.3.4. Casi d'uso

L'MLPGA prevede complessivamente quattro casi d'uso:

- *Train*
- *Test*
- *Run*
- *Full*

Il caso d'uso "***Train***" prevede l'addestramento del modello tramite l'utilizzo di un dataset di input e uno di target. Al termine di questa fase, viene salvato all'interno di un file, i miglior risultato ricavato, ovvero la matrice dei pesi tramite la quale il MLP restituisce l'output con il minor valore di fitness.

Nel caso d'uso "***Test***" l'utente ha la possibilità di verificare la qualità dei pesi ricavati durante la fase di Train.

In questa fase è possibile utilizzare o lo stesso input dataset utilizzato durante l'addestramento, oppure usare un input dataset composto da pattern mai utilizzati (scelta consigliata); Gli output ricavati dalla rete, infine, verranno confrontati infine con un target dataset.

Dopo aver addestrato e testato la nuova rete MLP, l'utente potrà eseguire il caso d'uso "***Run***". A differenza dei casi precedenti, in questa fase non è previsto l'uso di un target dataset, perchè il suo unico scopo è semplicemente quello di eseguire il modello come se questo fosse una funzione analitica standard.

L'ultimo caso d'uso è "***Full***", il quale non è altro che l'unione dei due casi "Train" e "Test". Questo caso d'uso può essere utilizzato come alternativa all'esecuzione sequenziale dei due casi di cui è composto.

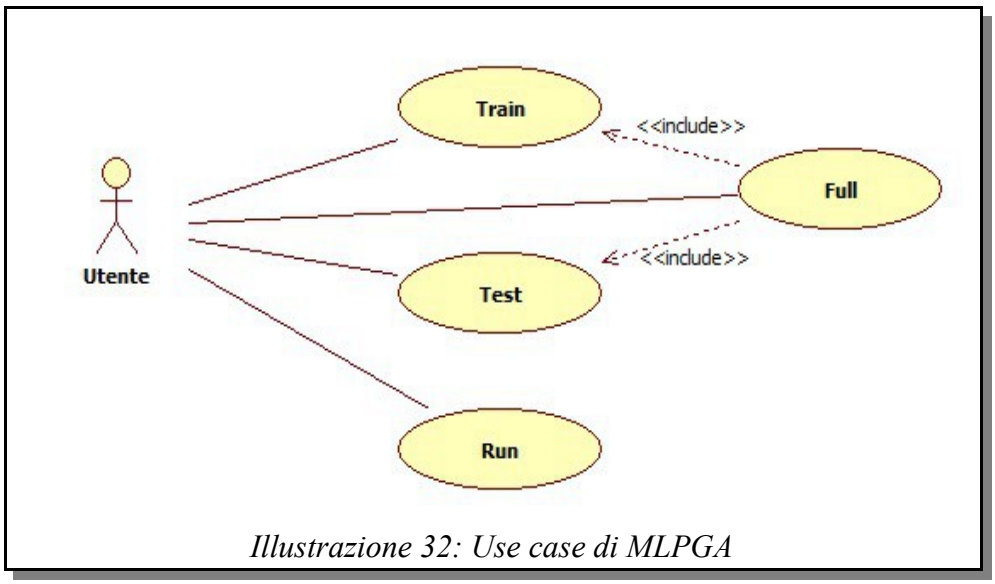


Illustrazione 32: Use case di MLPGA

6.3.5 Input/Output

Gli input e gli output del software implementato sono file di testo che possono essere suddivisi in:

- File di configurazione
- File di input
- File di output

I file di configurazione contengono le direttive generali per quanto riguarda la scelta del caso d'uso, la morfologia della rete MLP e la costituzione della popolazione.

Questa tipologia di file si suddivide a loro volta in tre categorie:

- ***Network configuration file***, contenente le informazioni per la configurazione della rete MLP;
- ***Experiment configuration file,*** contenente informazioni dettagliate riguardanti il caso d'uso da eseguire;

- ***General configuration file***, contenente il caso d'uso da eseguire (TRAIN, TEST...ecc), e i nomi del "*Experiment configuration file*" e del "*Network configuration file*".

I file di input contengono i dataset di input e di target utili all'esecuzione del modello.
I file di output contengono i risultati ottenuti durate il caso d'uso eseguito.

Esistono sostanzialmente tre file di output:

- ***Weights file***: contenente la matrice dei pesi ricavati durante il "training";
- ***Error file***: contenente l'andamento dell'errore durante tutta la fase di "Train";
- ***Log file***: contenente le informazioni generali riguardanti l'esperimento appena concluso;
- ***Output file:*** contiene l'output della rete MLP ottenuto tramite l'utilizzo dei pesi contenuti nel "Weights file";
- ***Statistical file:*** sono file contenenti dati statistici come la "Confusion Matrix" per la classificazione e la "Regression Matrix" per quanto riguarda invece la regressione.

6.4. Caratteristiche e limitazioni

Il modello implementato risulta essere in media discretamente efficiente, anche se in presenza di dati particolarmente "rumorosi" o complessi, le sue prestazioni possono risultare degradate rispetto a modelli MLP con differenti regole di apprendimento (Brescia et al. 2012a). In realtà la vera limitazione di questo modello è, come detto in precedenza, l'eccessivo tempo impiegato per effettuare la fase di training. E' stato proprio questo il problema principale che ci ha spinto a cercare un' alternativa alla classica implementazione seriale in modo da ridurre i tempi computazionali della fase di training.

7. Il modello Fast MLPGA (FMLPGA)

Per risolvere i problemi di performance illustrati nel capitolo precedente, abbiamo deciso di utilizzare la tecnologia Nvidia CUDA per parallelizzare il lavoro del modello MLPGA, riducendo così i tempi di esecuzione dello stesso.
Dalla parallelizzazione di MLPGA nasce il modello, da noi chiamato FMLPGA (Fast MLPGA).

7.1. Il processo di parallelizzazione tramite framework CUDA

Facendo alcuni test su MLPGA abbiamo notato che la fase più lenta del caso training è il calcolo dei valori di fitness dei singoli individui della popolazione; il calcolo di questi valori è infatti affidato ad un doppio loop (sulla dimensione della popolazione e sul numero di pattern del dataset) all'interno del quale viene eseguita la funzione di forward della rete.
Questo approccio porta all'esecuzione di un elevatissimo numero di istanze seriali della funzione di forward (anche oltre 100000), che penalizza di molto le performance del modello.
Per risolvere questo problema, abbiamo pensato alla riprogettazione dell'algoritmo di training di MLPGA in modo tale da sostituire il doppio loop precedentemente illustrato con un unica funzione che effettui il forwarding in parallelo su N reti, e dai loro risultati ricavarci i valori di fitness.

7.1.1. MLP4Cuda

Osservando i numerosi limiti che caratterizzano CUDA, e effettuando vari test, ci siamo accorti che l'implementazione della rete MLP precedentemente mostrata non era idonea per essere utilizzata su ambiente CUDA a causa dei numerosi puntatori di cui è composta; E' infatti molto semplice intuire che leggere un dato tramite una successione di puntatori risulta molto svantaggioso in un ambiente come CUDA, dove l'alta latenza nell'accesso alla memoria è una delle

"caratteristiche" principali (soprattutto se la memoria utilizzata è la "Global Memory").

Abbiamo dunque deciso di creare una nuova implementazione del MultiLayer Perceptron utilizzando un unica struttura dati, al cui interno l'utilizzo dei puntatori è minimo, il suo nome è *MLP4Cuda*.

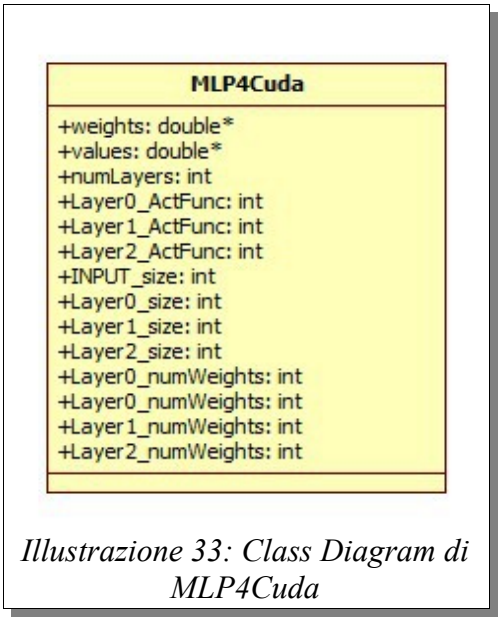


Illustrazione 33: Class Diagram di MLP4Cuda

Nell'immagine soprastante vediamo un Class Diagram di MLP4Cuda, il quale è composto dalle seguenti variabili:

- ***weights:*** Puntatore ad un array dinamico contenente i pesi di tutta la rete. Per semplificare l'accesso ad uno specifico peso sono state implementate funzioni ad-hoc che offrono una semplice interfaccia all'array;
- ***values:*** Puntatore ad un array dinamico contenente tutti valori dei singoli neuroni. Anche in questo caso, così come per l'array "weights", sono state implementate funzioni con lo scopo di creare una semplice interfaccia all'array;
- ***numLayers:*** contiene il numero di layers (livelli) presenti nella rete;
- ***Layer0_ActFunc:*** contiene l'identificativo della funzione di attivazione del Layer0;

- ***Layer1_ActFunc:*** contiene l'identificativo della funzione di attivazione del Layer1;
- ***Layer2_ActFunc:*** contiene l'identificativo della funzione di attivazione del Layer2;
- ***INPUT_size:*** contiene la dimensione dell'input layer;
- ***Layer0_size:*** contiene la dimensione del Layer0;
- ***Layer1_size:*** contiene la dimensione del Layer1;
- ***Layer2_size:*** contiene la dimensione del Layer2;
- ***Layer0_numWeights:*** contiene il numero di pesi di cui è composto ogni singolo neurone del Layer0;
- ***Layer1_numWeights:*** contiene il numero di pesi di cui è composto ogni singolo neurone del Layer1;
- ***Layer2_numWeights:*** contiene il numero di pesi di cui è composto ogni singolo neurone del Layer2.

Cosa si intende con Layer0, Layer1, Layer2?

In questo modello il tipo di informazioni a cui fanno riferimento le etichette “Layer0”, “Layer1” e “Layer2” dipende dalla morfologia della rete. Possiamo però generalizzare il loro funzionamento dicendo che Layer0 rappresenta il primo layer successivo all'input layer (in una rete senza “hidden layer” rappresenta l'output layer), Layer1 rappresenta il primo layer successivo al layer rappresentato da Layer0 (in una rete con un solo “hidden layer” rappresenta l'output layer) ed infine Layer2 è il primo layer successivo al layer rappresentato da Layer1 (in una rete con due “hidden layer” rappresenta l'output layer).Nel caso in cui la rete da simulare abbia un numero di layer inferiore a 3, le variabili che fanno riferimento ai layer inesistenti verranno settate a 0.

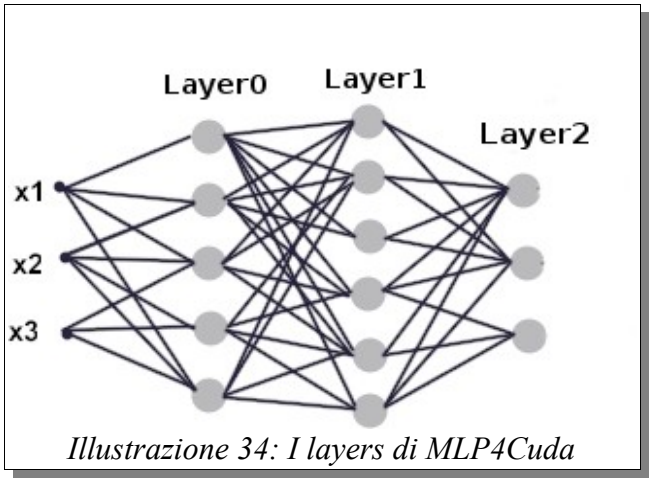


Illustrazione 34: I layers di MLP4Cuda

7.1.2. Allocazione delle risorse

La gestione dello spazio di memoria all'intero dell'ambiente CUDA è una fase delicata della programmazione e per evitare inutili allocazioni/deallocazioni di variabili e strutture dati che influenzerebbero negativamente le prestazioni del modello, abbiamo pensato di allocare tutte le strutture dati e le variabili utilizzate nell'ambiente CUDA nella fase di startup del modello.

Andando nel particolare, durante lo startup vengono allocate le seguenti risorse:

- Un array contenente tutti i pattern del dataset "input" (d_input);
- Un array contenente tutti i pattern del dataset "target" (d_output);
- Un array contenente tutti i dna della popolazione (d_dna);
- N reti MultiLayer Perceptron (variabili MLP4Cuda);
- Un array di dimensione N il cui scopo è quello di immagazzinare gli errori parziali delle singole reti. (d_batcherror).

Dove "N" è un valore intero equivalente al prodotto tra il numero di individui della popolazione, e il numero di pattern del dataset.

Tutte le variabili e strutture dati sono state allocate sulla GPU tramite la funzione *cudaMalloc*.

7.1.3. Preparazione della "CUDA Grid"

Una volta terminata l'allocazione delle variabili e delle strutture dati sulla GPU, bisogna configurare la "CUDA Grid".

Con il termine "CUDA Grid" intendiamo la griglia virtuale composta dai threads che eseguono il lavoro parallelo.

La "CUDA Grid", in questo modello, è costituita da un numero di blocchi pari al numero di pattern del dataset, ed ogni blocco sarà a sua volta composto da un numero di threads pari al numero di individui della popolazione.

Ogni threads eseguirà la funzione di forward su una specifica coppia pattern/dna attraverso una delle N reti allocate in fase di startup.

Nell'immagine sottostante vediamo la morfologia della CUDA Grid per dataset di dimensione 4 e una popolazione di dimensione 8.

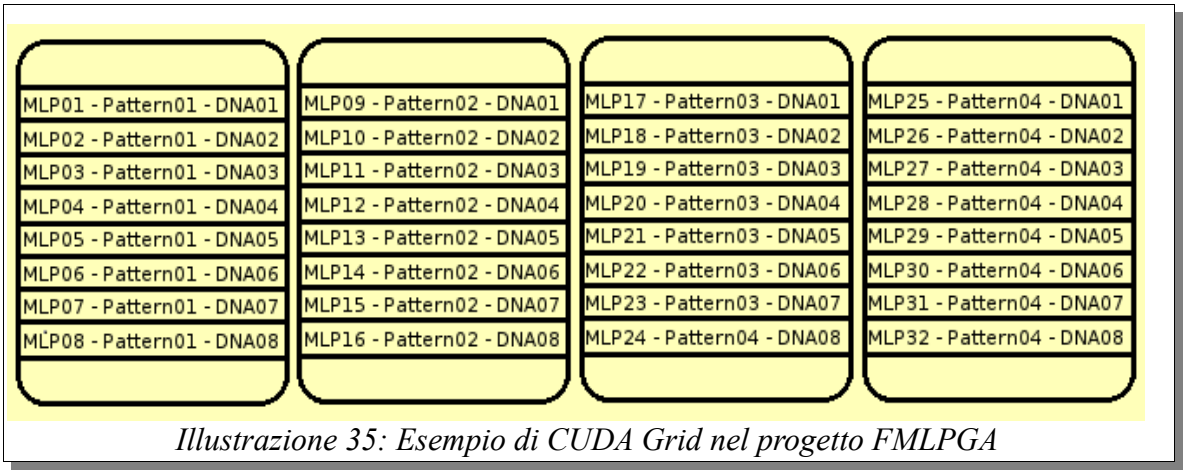


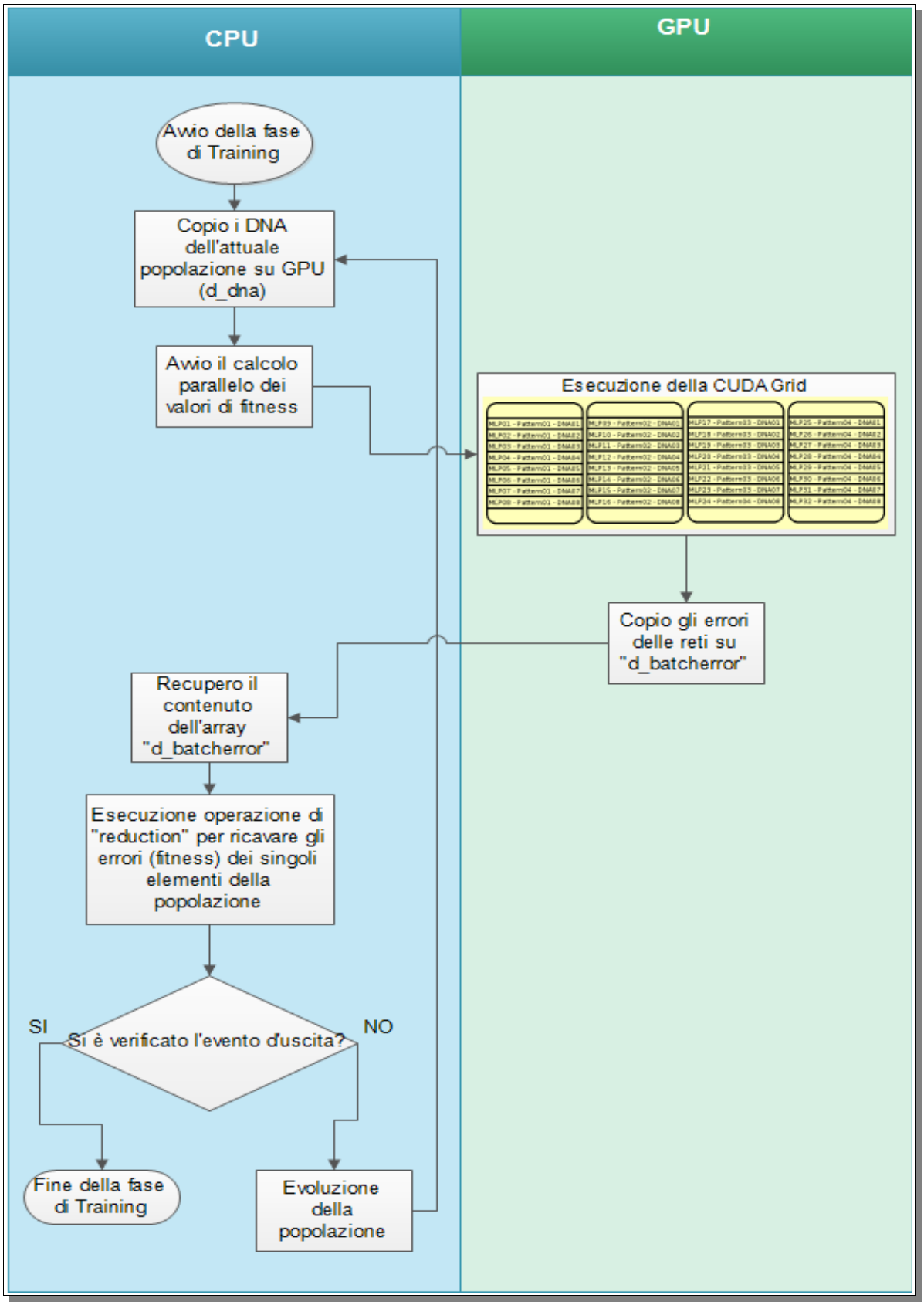
Illustrazione 35: Esempio di CUDA Grid nel progetto FMLPGA

7.2. Funzionamento del modello

Una volta allocate le variabili sulla GPU e settate le dimensioni della CUDA Grid, il modello è pronto per affrontare la fase di Training. Questa tipologia di implementazione del modello MLPGA (chiamata da noi FMLPGA), offre prestazioni fino a 10 volte superiori alla classica implementazione seriale, ma di questo argomento ci occuperemo nel prossimo capitolo.

Il modus operandi di FMLPGA può essere schematizzato e semplificato tramite il

seguente activity diagram:



8. Test prestazioni e confronto MLPGA-FMLPGA

Il presente capitolo è dedicato alla descrizione e discussione dei test di prestazioni qualitative del modello FMLPGA, utilizzando due casi d'uso scientifici in ambito astrofisico, nonché al confronto tra le due implementazioni, rispettivamente MLPGA seriale e FMLPGA parallela, in termini di velocità di esecuzione.

8.1. Piattaforme di sviluppo e test

Lo sviluppo e le fasi di messa a punto, debug e test preliminari sono stati realizzati sfruttando le risorse di calcolo del progetto PON S.Co.P.E., presso l'Università Federico II, consistenti in un sistema multi-core dotato di CPU Intel Xeon E5506 a 2.13 GHz 8-core, dotato di 4 GPU Fermi, modello NVIDIA TESLA S2050, con 448 core per ogni GPU, dedicate al calcolo parallelo.

I test prestazionali e di confronto tra le due versioni del modello MLPGA sono stati eseguiti su una macchina host, presso l'Osservatorio Astronomico di Capodimonte, dotata di una CPU Intel i7 a 3.4 GHz 2600 dual core e di una scheda NVIDIA GPU Geforce GTX 460 a 336 core.

8.2 Casi d'uso scientifici

Come già evidenziato, il modello MLPGA si basa sul paradigma supervisionato di Machine Learning. Si presta dunque a trattare problemi di classificazione e regressione. Nella fattispecie, l'algoritmo e le due relative implementazioni, seriale (MLPGA) e parallela (FMLPGA), sono stati impiegati per affrontare due casi scientifici reali, afferenti a problematiche note in ambito astrofisico. Il primo si riferisce alla classificazione di una tipologia peculiare di oggetti, denominati ammassi globulari, mentre il secondo al calcolo di un fondamentale parametro cosmologico, il redshift fotometrico, in grado di evidenziare le distanze cosmologiche di oggetti astronomici peculiari, denominati Quasar.

8.2.1 Classificazione: identificazione di ammassi globulari

Il dataset scelto come base dati per il test di classificazione del modello FMLPGA è denominato GCSearch. Il relativo problema scientifico consiste nello studio di popolazioni di ammassi globulari (in inglese Globular Clusters, GC), in galassie esterne a quella terrestre (Brescia et al. 2012a). Questo argomento è di notevole interesse in molti settori astrofisici: dalla cosmologia, all'evoluzione dei sistemi stellari, fino alla formazione ed evoluzione di sistemi binari. In generale lo studio dei GC extra-galattici richiede l'uso di fotometria a grande campo e multi-banda. Infatti, i GC in galassie lontane (a diversi MegaParsec, Mpc, da noi), appaiono come sorgenti luminose non risolte nelle immagini astronomiche da Terra. Ciò li rende indistinguibili dalle galassie circostanti, causando un altissimo tasso di contaminazione da parte di oggetti spuri. Per tale motivo la tecnica tradizionale consiste nella selezione dei GC sulla base dei colori e delle magnitudini delle sorgenti nelle diverse bande osservate. Tuttavia, per cercare di minimizzare l'effetto di contaminazione e misurare le proprietà dei GC, come dimensioni e parametri strutturali (raggio di core, concentrazione, tasso di formazione di stelle binarie ecc.), si richiedono dati supplementari, ad alta risoluzione, ottenuti da osservazioni con telescopi spaziali (come l'Hubble Space telescope, HST). Il dataset GCSearch, utilizzato nel presente esperimento consiste quindi in dati (pattern di oggetti osservati) a grande campo (wide field) ottenuti da HST e relativi alla galassia NGC1399, una gigante ellittica distante circa 20 Mpc. Questa galassia rappresenta l'ambiente ideale di studio dei GC, a causa della sua relativa distanza e della conseguente possibilità di coprire una grande porzione del suo sistema di GC tramite un relativamente ridotto numero di osservazioni. Inoltre a quella distanza i GC sono solo marginalmente risolti da HST, permettendo quindi di verificare le prestazioni dell'algoritmo di classificazione nel caso peggiore (cioè più conservativo, dato che i dati sono parzialmente contaminati e rumorosi). I dati sono caratterizzati da pattern (cioè oggetti) costituiti ciascuno da un certo numero di parametri, ottici e strutturali, relativi alle

sorgenti rivelate nelle immagini osservate. La sezione di cielo relativa alle osservazioni è evidenziata nella figura sottostante.

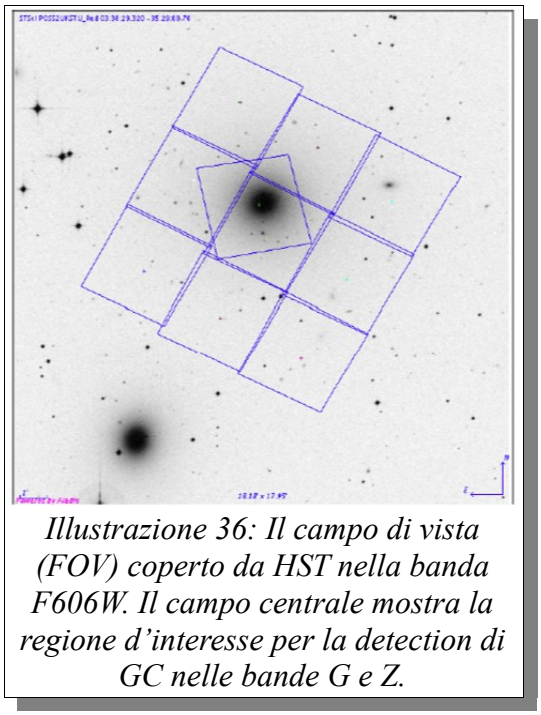


Illustrazione 36: Il campo di vista (FOV) coperto da HST nella banda F606W. Il campo centrale mostra la regione d'interesse per la detection di GC nelle bande G e Z.

La base di conoscenza costruita per l'esperimento di classificazione supervisionata con il modello FMLPGA è stata ottenuta attraverso l'identificazione di GC con I metodi statistici tradizionali (Chi quadro) applicati al modello fisico degli oggetti in esame. Tali tecniche, sebbene affidabili scientificamente, sono particolarmente onerose analiticamente e richiedono un grande numero di costose campagne osservative con strumenti spaziali e da terra di grandi dimensioni. L'obiettivo ultimo dell'esperimento con il modello FMLPGA è dunque la possibilità di addestrare un modello empirico sulla base di poche osservazioni reali, mettendolo in condizioni di classificare correttamente i GC con un minimo numero di osservazioni ed in ambienti caratterizzati da rumore e contaminazione. Il dataset finale è costituito da 2100 pattern (oggetti), ciascuno costituito da 11 parametri (features, 7 ottiche e 4 strutturali), più la label associata (0 non GC, 1 GC), relativa alla classificazione effettuata con tecniche tradizionali. Questo dataset costituisce l'input alla fase di addestramento e di

validazione del modello FMLPGA.

Di seguito si riporta una frazione del dataset, relativa ai primi 10 pattern (i parametri sono separati da virgole):

24.4753,26.7468,24.3789,0.0205,3.72,0.067,4.12,16.25,-0.1139,1.822,51.29,0,1
24.2342,26.5263,24.1632,0.0196,3.5,0.027,4.01,16.61,0.1321,1.856,35.38,0,1
23.1554,25.5964,23.1654,0.016,3.5,0.032,4.09,14.47,-0.3295,2.638,129.2,1,0
22.6316,25.3519,22.6808,0.0151,3.5,0.039,4.69,16.33,0.8065,5.002,80.45,1,0
22.4708,24.4951,22.4699,0.0216,3.5,0.066,3.45,12.81,-0.3912,-7.425,5.66,0,1
23.9033,27.5896,23.9168,0.0255,4.49,0.272,9.63,19.99,8.397,14.79,88.5,1,0
24.1972,26.4219,24.0978,0.0192,3.7,0.079,4.04,15.72,-0.1447,1.514,44.77,0,1
20.2423,22.1866,20.2963,0.017,3.5,0.03,3.23,6.68,-0.6999,-0.1492,1.899,0,1
23.5134,26.0983,23.511,0.0167,3.76,0.05,4.55,16.6,0.3777,4.75,105.8,1,0
22.5967,25.1807,22.6182,0.0147,3.5,0.021,4.45,15.98,0.6535,3.615,52.13,1,0

8.2.2 Regressione: calcolo del redshift fotometrico per oggetti Quasar

In Astrofisica il redshift (letteralmente spostamento verso il rosso) è il fenomeno per cui la frequenza della luce, quando osservata in determinate circostanze, è più bassa della frequenza che aveva quando è stata emessa. Ciò accade in genere quando la sorgente di luce si muove allontanandosi dall'osservatore (o equivalentemente, essendo il moto relativo, quando l'osservatore si allontana dalla sorgente). In particolare, si parla di redshift quando, nell'osservare lo spettro della luce emessa da galassie, quasar o supernovae lontane, questo appare spostato verso frequenze minori, se confrontato con lo spettro dei corrispondenti più vicini. Dato che nella luce visibile il rosso è il colore con la frequenza più bassa, il fenomeno ha preso questo nome. Tale parametro può essere misurato o spettroscopicamente o in maniera fotometrica. Nel primo caso la stima è generalmente molto precisa, sebbene molto più onerosa in termini di quantità di tempo osservativo da impiegare con costosi strumenti astronomici.

il redshift fotometrico (zphot) è una stima del redshift di sorgenti ottenuta usando la fotometria al posto della spettroscopia. Per tale motivo la qualità della stima fotometrica è alterata principalmente dalla forma dello spettro di emissione e da un limitato numero di forti caratteristiche spettrali, riconoscibili solo tramite l'integrazione con la Distribuzione Spettrale di Energia (SED) al di sotto del limite di trasmissione del filtro ottico utilizzato. Sebbene meno accurati, i zphot offrono tuttavia diversi vantaggi rispetto agli omologhi redshift spettroscopici (zspec): (i) essendo derivati da immagini, i zphot sono molto meno onerosi in termini di tempo di osservazione; (ii) possono permettere di individuare oggetti molto più lontani del limite di flusso spettroscopico; (iii) sotto specifiche condizioni i zphot permettono di superare/correggere il bias spettroscopico, come per esempio quello presente ad alto redshift, dove la spettroscopia è spinta al limite ed è limitata dal rapporto segnale/rumore estremamente basso. Quest'ultimo aspetto diviene cruciale quando i zphot si applicano a oggetti Quasar (contrazione di QUASi-stellAR radio source, cioè radiosorgente quasi stellare), oggetti peculiari, somiglianti ad una stella in un telescopio ottico (cioè è una sorgente puntiforme), e che mostra un grande spostamento verso il rosso (redshift) del suo spettro. In genere i quasar sono sempre stati identificati mediante una tecnica basata sull'estrazione di candidati da osservazioni a grande campo multi-banda e successivamente mediante la validazione spettroscopica di tali candidati. In pratica, a causa dell'onerosa quantità di dati spettroscopici necessari, la validazione viene limitata ad un'esigua quantità di oggetti, estrapolando poi i risultati al resto del campione di candidati. Risulta ovvio quindi che ottenere l'identificazione del redshift di quasar tramite dati fotometrici costituisce un enorme vantaggio, minimizzando i costi delle osservazioni e superando l'estrapolazione statistica nell'identificazione di tali oggetti.

La base di conoscenza costruita per l'esperimento di regressione supervisionata con il modello FMLPGA è stata ottenuta attraverso il calcolo dei redshift con il metodo spettroscopico su dati ottenuti da quattro differenti strumenti (vedi figura

sottostante) e relative campagne (survey) osservative a grande campo (Brescia et al. 2012b):

- GALEX: survey a due bande (fuv e nuv) nel lontano e vicino ultravioletto;
- SDSS: survey a 5 bande (u g r i z) nel visibile;
- UKIDSS: equivalente di SDSS nel vicino infrarosso (bande y j h k);
- WISE: survey a 4 bande nel medio infrarosso.

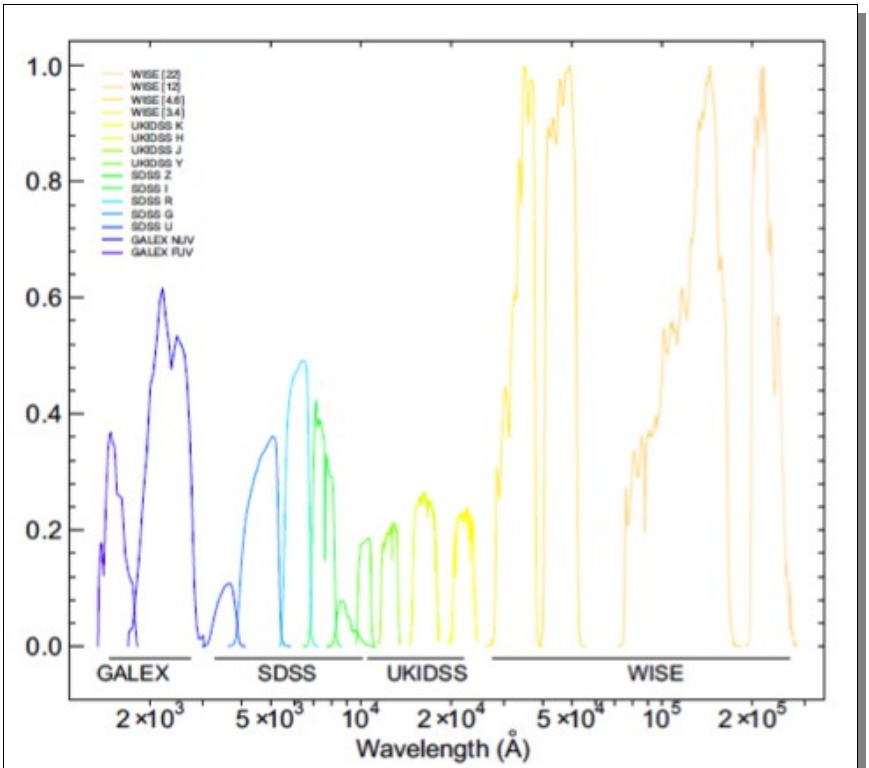


Illustrazione 37: curve di trasmissione delle quattro survey nelle rispettive bande spettrali.

L'obiettivo ultimo dell'esperimento con il modello FMLPGA è dunque la possibilità di addestrare un modello empirico sulla base di poche osservazioni reali, mettendolo in condizioni di calcolare correttamente il redshift fotometrico di oggetti quasar identificati dai quattro progetti suddetti. Il dataset finale è costituito da 1499 pattern (oggetti quasar), ciascuno costituito da 11 parametri (features, magnitudini nelle varie bande), più il valore noto del redshift

spettroscopico. Questo dataset costituisce l'input alla fase di addestramento e di validazione del modello FMLPGA.

Di seguito si riporta una frazione del dataset, relativa ai primi 10 pattern (i parametri sono separati da virgole):

23.0451,22.3865,21.3956,20.6516,20.4471,0.6733,0.9188,0.8195,0.1957,3.77,0,0.557
22.6303,21.5566,20.427,19.5619,19.4067,1.0808,1.1,0.8933,0.1545,3.594,0,0.4399
23.7177,23.3864,22.6248,21.8677,21.729,0.286,0.7801,0.8125,0.0926,3.049,0,0.7268
24.9936,24.9237,24.3248,24.2199,24.0675,0.0466,0.1922,0.3948,0.1973,3.458,0,0.7148
23.6051,22.7975,21.7195,20.9121,20.7008,0.8584,0.9947,0.8709,0.2072,4.137,0,0.5319
22.5319,21.9514,20.975,20.146,19.9494,0.586,0.9315,0.8727,0.194,4.708,0,0.5667
22.0164,21.3014,20.556,19.8579,19.683,0.7201,0.708,0.7435,0.1828,5.8,0,0.4342
23.7532,23.1459,22.8099,22.4685,22.4437,0.6347,0.1696,0.4065,0.0269,3.455,0,0.3245
23.7147,22.8983,22.1731,21.6817,21.65,0.7623,0.594,0.5945,-0.0149,3.706,0,0.4362
22.3038,21.3017,20.3543,19.589,19.4856,1.0017,0.9237,0.797,0.1023,5.158,0,0.438

8.3. Test di qualità

Relativamente ai due casi d'uso scientifici descritti in precedenza, sono stati effettuati sia test qualitativi, volti a validare scientificamente il modello FMLPGA, sia prestazionali, in termini di tempo computazionale, confrontando fra loro la versione seriale e quella parallela. In questo paragrafo discuteremo i risultati qualitativi.

8.3.1. *Classificazione*

In termini qualitativi, cioè di validazione scientifica dei risultati, il modello FMLPGA è stato testato in classificazione sul dataset dei globular clusters, variando opportunamente i parametri di configurazione disponibili, sia riguardo la topologia della rete MLP, sia riguardo le caratteristiche evolutive dell'algoritmo genetico.

I parametri di configurazione del modello, oggetto di variazioni sistematiche durante i test sono stati:

- Numero di livelli hidden della rete MLP;
- Numero di nodi del primo livello hidden;
- Numero di nodi del secondo livello hidden;
- Numero di iterazioni di training;
- Soglia di errore MSE di training;
- Numero di cromosomi della popolazione (numero di MLP generate ad ogni ciclo di training);
- Funzione di selezione evolutiva dei cromosomi (pesi delle MLP);
- Numero di cromosomi di elitismo;

Variando tali parametri, sono stati condotti circa un centinaio di test, al fine di individuare la migliore configurazione dei parametri. Per ogni esperimento, i risultati dell'addestramento sono stati valutati statisticamente attraverso la matrice di confusione.

Tale metodo si basa sulla disposizione in una matrice dei pattern in base alla loro classificazione, rispetto alle classi target. Gli elementi sulla diagonale principale, sommati, formano l'indice complessivo di accuratezza della classificazione, mentre il rapporto tra i valori delle diagonali secondarie rispetto al numero complessivo di oggetti noti appartenenti alle rispettive classi, identificano la percentuale di contaminazione. Infine il valore duale della contaminazione permette di calcolare l'indice di purezza di classificazione per ognuna delle classi.

Un esempio di matrice di confusione generata, nel caso migliore, cioè con le migliori prestazioni ottenute, è riportato di seguito.

Classification Confusion Matrix on 2100 patterns

	Not GC	GC
Target no GC	867	14
target GC	65	1154

classification accuracy (total number of well classified objects): 96.24%

classification purity for class 0 (not GC) = 98.41 %

classification purity for class 1 (GC) = 94.67 %

The contamination for each class is the dual of purity = 1.59% for not GC and 5.33% for GC objects

Il risultato riportato nella matrice di confusione si riferisce al caso migliore, ottenuto con funzione di selezione FITTING, rete MLP con doppio strato hidden e su 50000 iterazioni. La tabella seguente mostra, per confronto diretto, i risultati qualitativi, in termini dei 3 indicatori statistici su menzionati, al variare della funzione di selezione.

Funzione	Iterazioni	Classe target	Accuratezza [%]	Purezza [%]	Contaminazione [%]
Roulette	50000		94.89		
		Not GC		97.01	2.99
		GC		92.64	7.36
Ranking	50000		93.18		
		Not GC		95.32	4.68
		GC		91.45	8.55
Fitting	50000		96.24		
		Not GC		98.41	1.59
		GC		94.67	5.33

Illustrazione 38: Valutazione della qualità dei risultati del test di Classificazione

Dalla tabella si evince che nel caso FITTING, il modello FMLPGA rivela ottime prestazioni, superiori a quelle ottenibili con metodi tradizionali sui dati considerati.

8.3.2. Regressione

Con riferimento all'esperimento del calcolo del redshift fotometrico sulla base di conoscenza costituita da oggetti quasar, sono stati compiuti circa un centinaio di test, variando i parametri di configurazione del modello FMLPGA e valutando i risultati in termini numerici, attraverso il calcolo delle matrici di regressione, sia in termini grafici, attraverso i plot di distribuzione dei redshift fotometrici (zphot) rispetto a quelli spettroscopici (zspec).

I parametri di configurazione del modello, oggetto di variazioni sistematiche durante i test sono stati:

- Numero di livelli hidden della rete MLP;
- Numero di nodi del primo livello hidden;
- Numero di nodi del secondo livello hidden;
- Numero di iterazioni di training;
- Soglia di errore MSE di training;
- Numero di cromosomi della popolazione (numero di MLP generate ad ogni ciclo di training);
- Funzione di selezione evolutiva dei cromosomi (pesi delle MLP);
- Numero di cromosomi di elitismo;

La matrice di regressione, utilizzata per la valutazione statistica degli output forniti dal modello, consiste nel quantificare percentualmente il numero di oggetti aventi la distanza assoluta (valore assoluto di zphot calcolato – zspec noto)

inferiore a determinate soglie e valutando il valore dell'errore MSE (Mean Square Error) complessivo durante la fase di training.

Un esempio di matrice di regressione è quello riportato di seguito: si tratta del miglior esperimento in termini qualitativi, ottenuto utilizzando la funzione ROULETTE di selezione evolutiva dell'algoritmo genetico:

Regression Statistics results on 1499 patterns

In the regression case the matrix is an adapted confusion matrix

Regression matrix:

	Out 0	Out 1
target 0	1478	1
target 1	15	5

overall pattern prediction percentage = 98.93%, based on calculated threshold = 1.57

4.87% (73) of patterns within distance 0.00010 from target

10.21% (153) of patterns within distance 0.00100 from target

17.74% (266) of patterns within distance 0.01000 from target

66.11% (991) of patterns within distance 0.10000 from target

L'errore MSE del training è risultato pari a 0.091, dopo 50000 iterazioni. Dalla matrice di regressione riportata sopra, si evince quindi che per il 98.93% dei pattern, lo scostamento dei valori di zphot calcolati rimane al di sotto della distanza pari a 0.1, con discrete percentuali in cui tale distanza risulta ben più piccola. Questo esperimento è stato replicato variando in particolare la funzione di selezione evolutiva. I grafici seguenti mostrano le distribuzioni del redshift fotometrico in funzione di quello spettroscopico, al variare della funzione di selezione, ROULETTE, RANKING e FITTING. Il caso migliore, quello con la funzione ROULETTE, di cui abbiamo riportato sopra la matrice di regressione, è confermato anche graficamente.

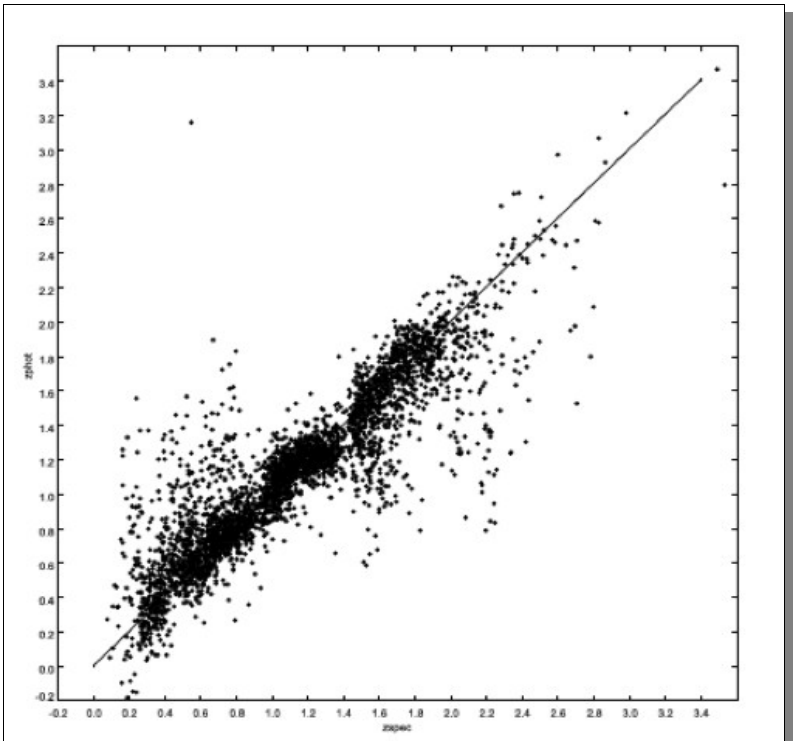


Illustrazione 39: Scatter plot (zspec vs zphot) sui 1499 oggetti quasar, ottenuto dal modello FMLPGA con funzione di selezione FITTING nel miglior esperimento.

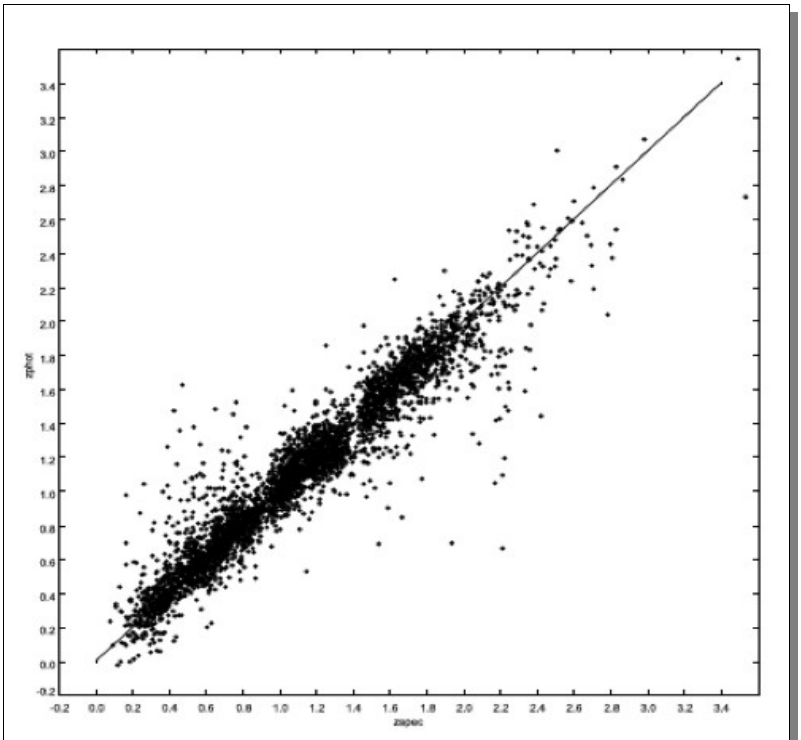


Illustrazione 40: Scatter plot (zspec vs zphot) sui 1499 oggetti quasar, ottenuto dal modello FMLPGA con funzione di selezione RANKING nel miglior esperimento.

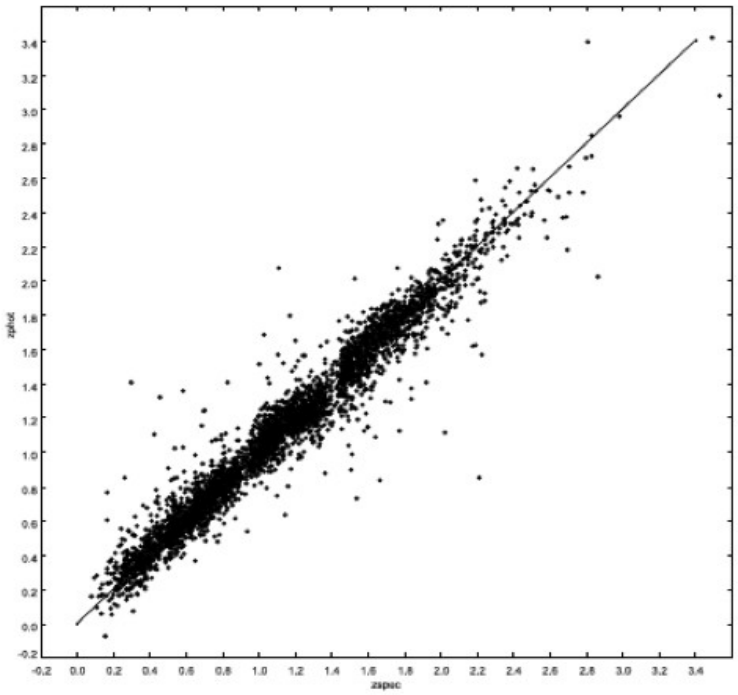


Illustrazione 41: Sscatter plot (zspec vs zphot) sui 1499 oggetti quasar, ottenuto dal modello FMLPGA con funzione di selezione ROULETTE nel miglior esperimento.

8.4. Test di prestazioni computazionali

Un test fondamentale riguardava il confronto diretto tra le due implementazioni, seriale e parallela, dell'algoritmo MLPGA in termini di tempo di esecuzione. Come detto in precedenza, lo scopo primario del presente lavoro consisteva infatti nel risolvere il principale ostacolo all'uso del modello, cioè l'eccessiva durata della fase di training e la conseguente scarsa scalabilità rispetto alla dimensione dei dati forniti per l'addestramento.

Il confronto è stato realizzato tenendo presente la variazione sistematica e oculata dei parametri del modello, già descritta nel paragrafo precedente e ovviamente utilizzando i due dataset utilizzati nei test di qualità. Si è quindi potuto valutare la variazione di performance sia nel caso di regressione che di classificazione.

Onde poter confrontare indirettamente le prestazioni nei due casi funzionali, si è

scelto di utilizzare due dataset, uno per classificazione ed uno per la regressione, perfettamente congruenti in termini di numero di pattern e di features. Nella fattispecie i due dataset hanno entrambi 1000 pattern e 11 features.

8.4.1. Test prestazionali per classificazione

I test qualitativi nel caso della classificazione, descritti in precedenza, hanno permesso non solo di validare scientificamente le prestazioni del modello FMLPGA, ma anche di identificare la migliore configurazione dei parametri del modello. Tali parametri sono riassunti di seguito:

- Numero di livelli hidden della rete MLP = 2;
- Numero di nodi del primo livello hidden = 23;
- Numero di nodi del secondo livello hidden = 4;
- Funzione di attivazione dei neuroni di ogni strato = tangente iperbolica;
- Numero di iterazioni di training = da 1000 a 50000;
- Soglia di errore MSE di training = 0.001;
- Distribuzione di probabilità per generazione popolazione iniziale = gaussiana;
- Numero di cromosomi della popolazione (numero di MLP per ogni ciclo di training) = 20;
- Funzione di selezione evolutiva = ROULETTE/RANKING/FITTING;
- Numero di cromosomi di elitismo = 2;
- Probabilità di crossover = 0.9;
- Probabilità di mutazione = 0.4;
- Numero di cromosomi per RANKING = 4;

Tale configurazione è stata dunque fissata per condurre l'analisi delle prestazioni,

in termini di tempo di esecuzione, variando solo due parametri: il numero di iterazioni di training e la funzione di selezione. Questi due fattori sono intuitivamente i principali parametri che possono incidere maggiormente sulla variazione di tempo di esecuzione del modello. Sicuramente anche il numero di neuroni della rete influenza tale fattore prestazionale, ma influenza anche direttamente le prestazioni qualitative del modello. Per cui si è preferito fissare il numero di neuroni, tenendo presente i risultati qualitativi ottenuti. La tabella seguente mostra il confronto diretto tra le versioni seriale (MLPGA) e quella parallela (FMLPGA) in termini di tempo di esecuzione ed in funzione dei due parametri citati in precedenza.

iterazioni	funzione	pattern	features	Tempo di esecuzione [sec]	
				MLPGA	FMLPGA
1000	ROULETTE	1000	11	104	10
2000				209	27
4000				428	55
8000				897	117
10000				1066	147
30000				3175	442
50000				5608	736
1000	RANKING	1000	11	106	14
2000				213	29
4000				436	58
8000				913	116
10000				1085	149
30000				3369	437
50000				5632	734
1000	FITTING	1000	11	98	13
2000				201	27
4000				416	58
8000				890	117
10000				1056	145
30000				3173	438
50000				5502	734

Illustrazione 42: Analisi prestazioni per il caso funzionale CLASSIFICAZIONE (MLPGA = seriale; FMLPGA = parallelo)

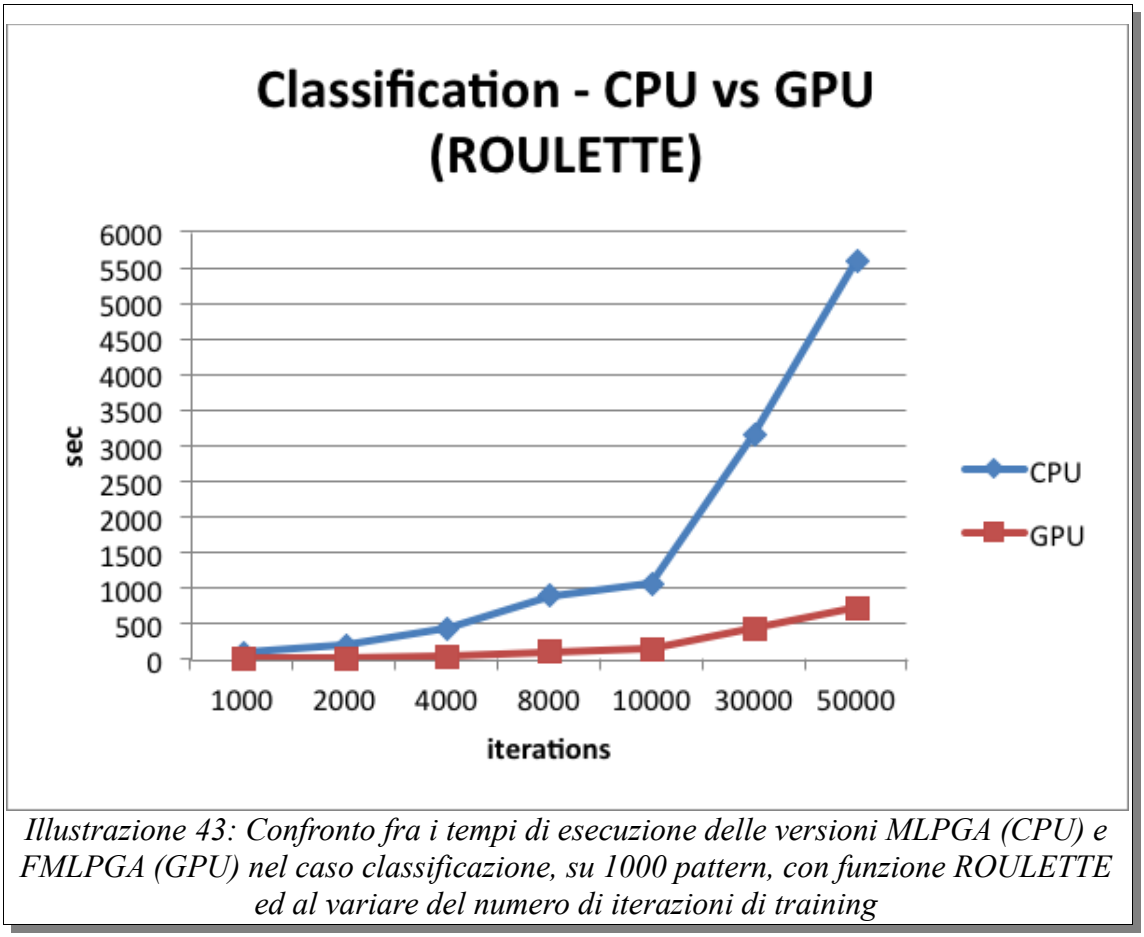


Illustrazione 43: Confronto fra i tempi di esecuzione delle versioni MLPGA (CPU) e FMLPGA (GPU) nel caso classificazione, su 1000 pattern, con funzione ROULETTE ed al variare del numero di iterazioni di training

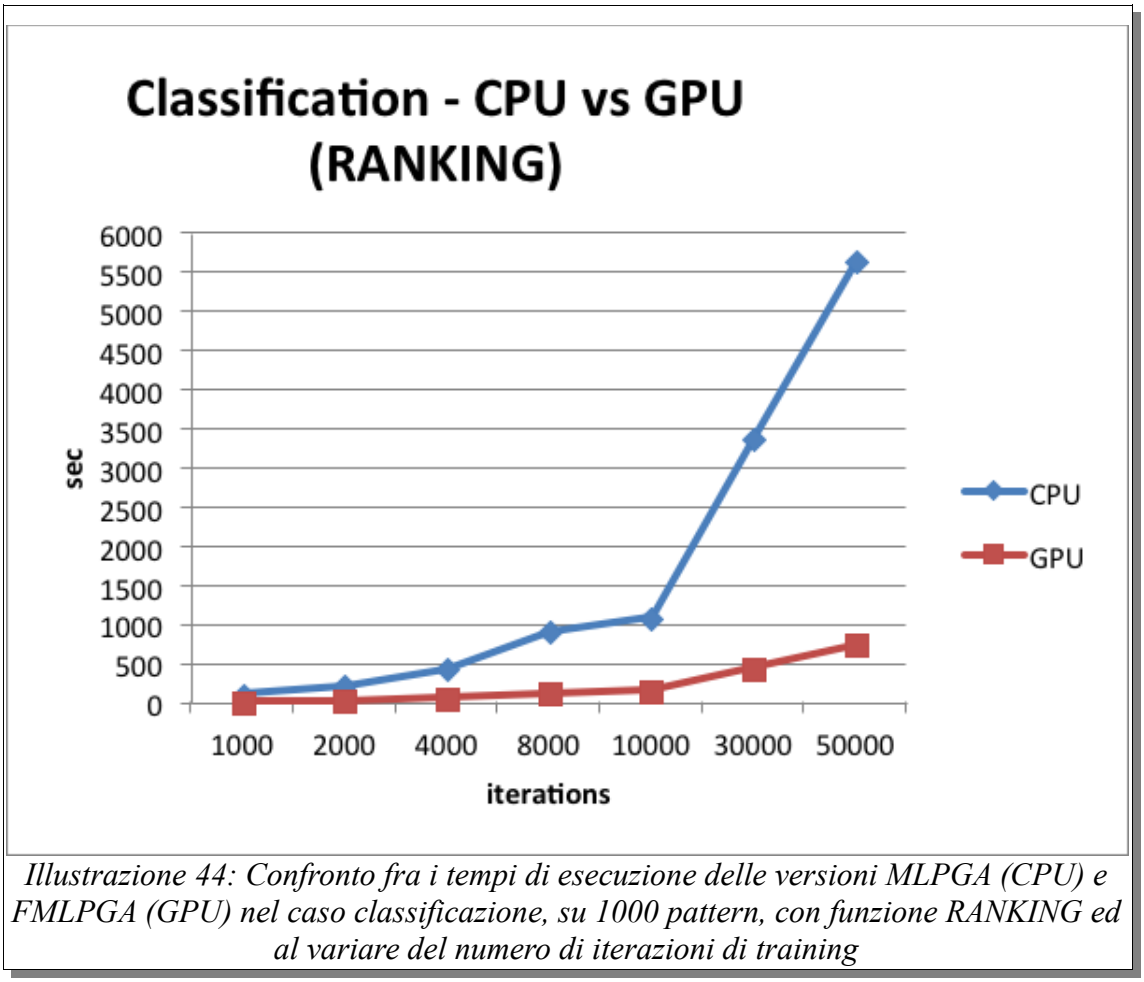


Illustrazione 44: Confronto fra i tempi di esecuzione delle versioni MLPGA (CPU) e FMLPGA (GPU) nel caso classificazione, su 1000 pattern, con funzione RANKING ed al variare del numero di iterazioni di training

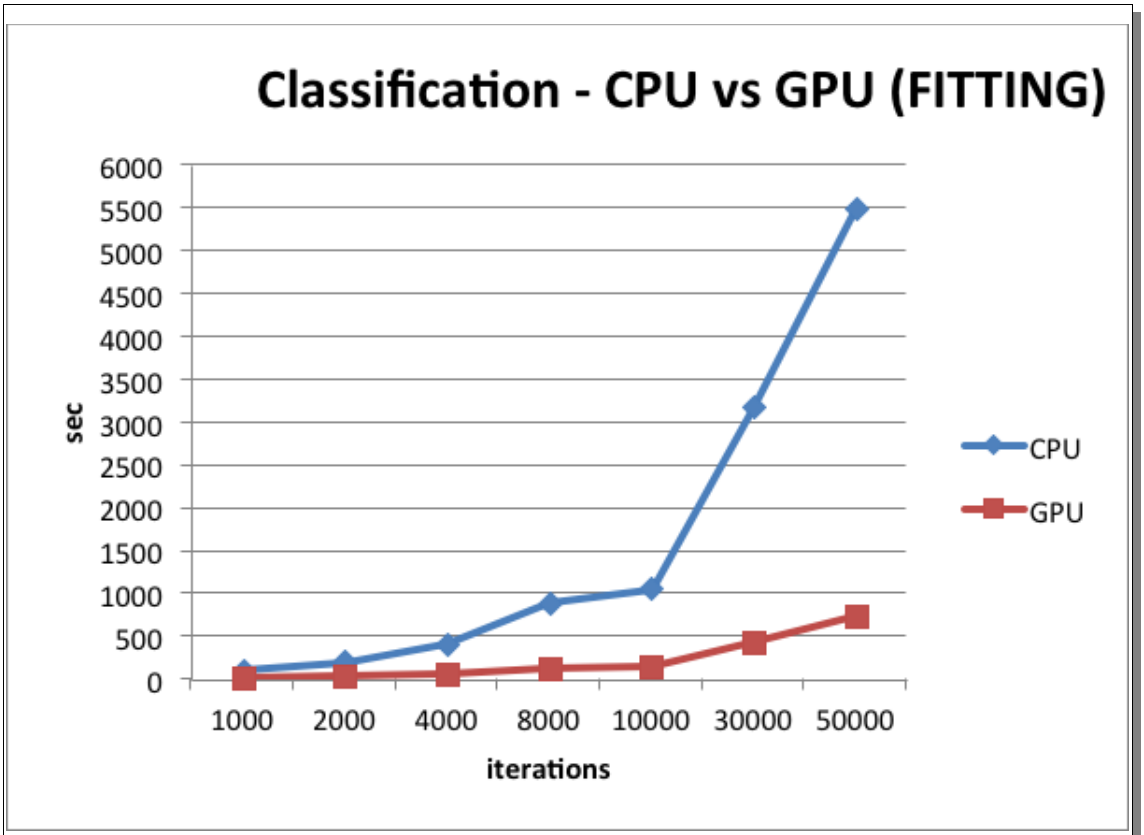


Illustrazione 45: Confronto fra i tempi di esecuzione delle versioni MLPGA (CPU) e FMLPGA (GPU) nel caso classificazione, su 1000 pattern, con funzione FITTING ed al variare del numero di iterazioni di training

8.4.2. Test prestazionali per regressione

I test qualitativi nel caso della regressione, descritti in precedenza, hanno permesso non solo di validare scientificamente le prestazioni del modello FMLPGA, ma anche di identificare la migliore configurazione dei parametri del modello. Tali parametri sono riassunti di seguito:

- Numero di livelli hidden della rete MLP = 2;
- Numero di nodi del primo livello hidden = 23;
- Numero di nodi del secondo livello hidden = 4;
- Funzione di attivazione dei neuroni di ogni strato = tangente iperbolica;
- Numero di iterazioni di training = da 1000 a 50000;
- Soglia di errore MSE di training = 0.0001;
- Distribuzione di probabilità per generazione popolazione iniziale = gaussiana;
- Numero di cromosomi della popolazione (numero di MLP per ogni ciclo di training) = 20;
- Funzione di selezione evolutiva = ROULETTE/RANKING/FITTING;
- Numero di cromosomi di elitismo = 2;
- Probabilità di crossover = 0.9;
- Probabilità di mutazione = 0.4;
- Numero di cromosomi per RANKING = 4;

Tale configurazione è stata dunque fissata per condurre l'analisi delle prestazioni, in termini di tempo di esecuzione, variando solo due parametri: il numero di iterazioni di training e la funzione di selezione.

Questi due fattori sono intuitivamente i principali parametri che possono incidere maggiormente sulla variazione di tempo di esecuzione del modello. Sicuramente anche il numero di neuroni della rete influenza tale fattore prestazionale, ma influenza anche direttamente le prestazioni qualitative del modello. Per cui si è preferito fissare il numero di neuroni, tenendo presente i risultati qualitativi ottenuti. La tabella seguente mostra il confronto diretto tra le versioni seriale (MLPGA) e quella parallela (FMLPGA) in termini di tempo di esecuzione ed in funzione dei due parametri citati in precedenza.

iterazioni	**funzione**	**pattern**	**features**	**Tempo di esecuzione [sec]**	
				MLPGA	**FMLPGA**
1000	ROULETTE	1000	11	111	10
2000				224	28
4000				471	57
8000				902	117
10000				1146	147
30000				3399	442
50000				5540	737
1000	RANKING	1000	11	114	14
2000				229	28
4000				490	58
8000				931	117
10000				1193	149
30000				3559	438
50000				5639	734
1000	FITTING	1000	11	103	13
2000				208	27
4000				435	58
8000				897	116
10000				1140	146
30000				3450	438
50000				5517	734

Illustrazione 46: Analisi prestazioni per il caso funzionale REGRESSIONE (MLPGA = seriale; FMLPGA = parallelo)

Dai valori in tabella è possibile riepilogare tutti i risultati del confronto attraverso i seguenti grafici.

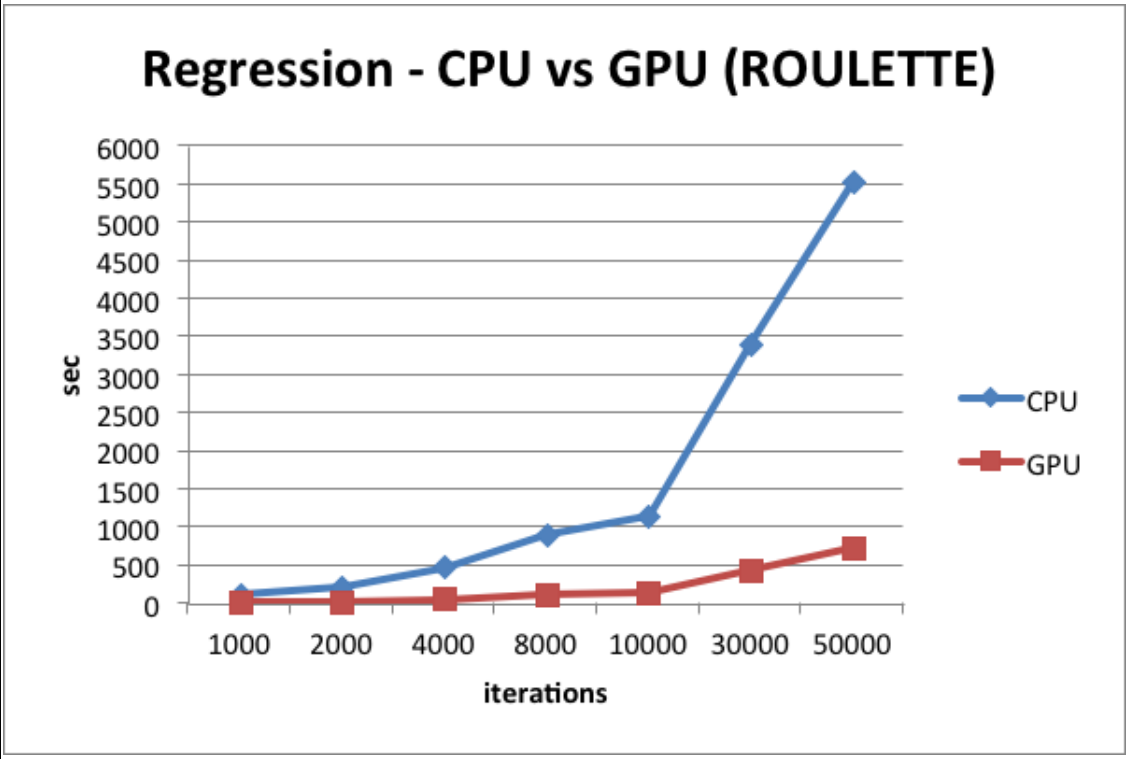


Illustrazione 47: Confronto fra i tempi di esecuzione delle versioni MLPGA (CPU) e FMLPGA (GPU) nel caso regressione, su 1000 pattern, con funzione ROULETTE ed al variare del numero di iterazioni di training

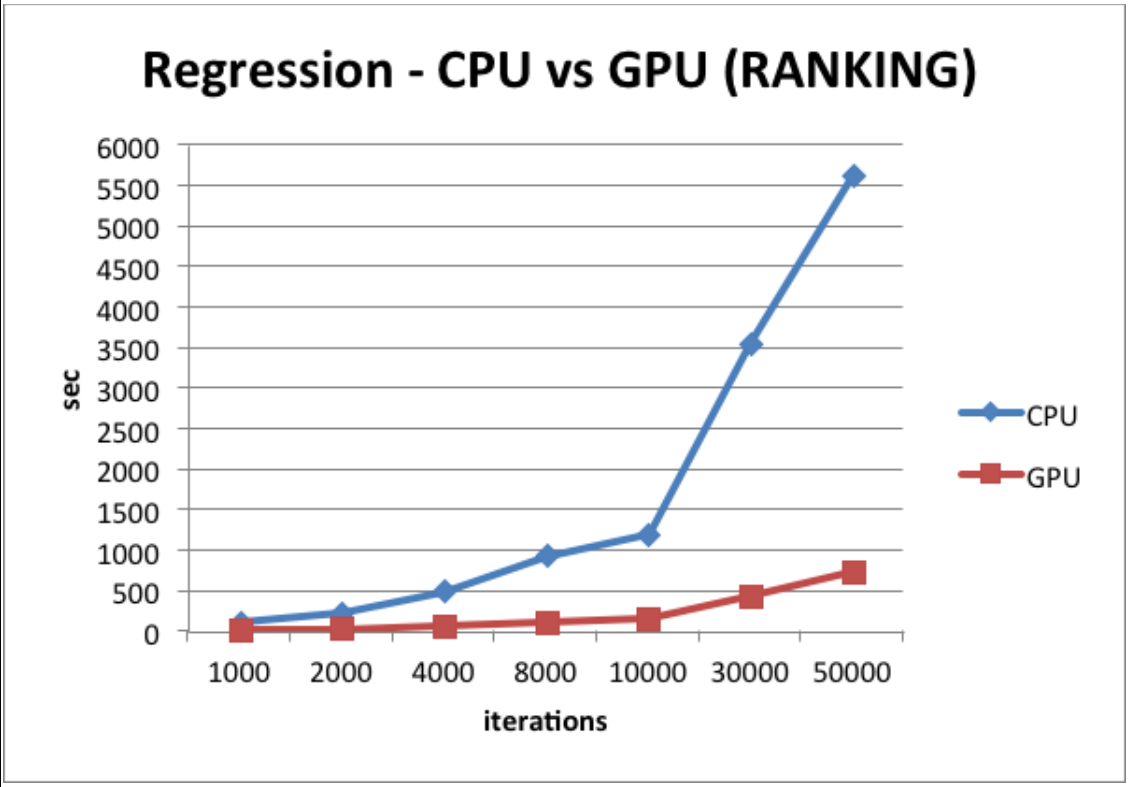


Illustrazione 48: Confronto fra i tempi di esecuzione delle versioni MLPGA (CPU) e FMLPGA (GPU) nel caso regressione, su 1000 pattern, con funzione RANKING ed al variare del numero di iterazioni di training

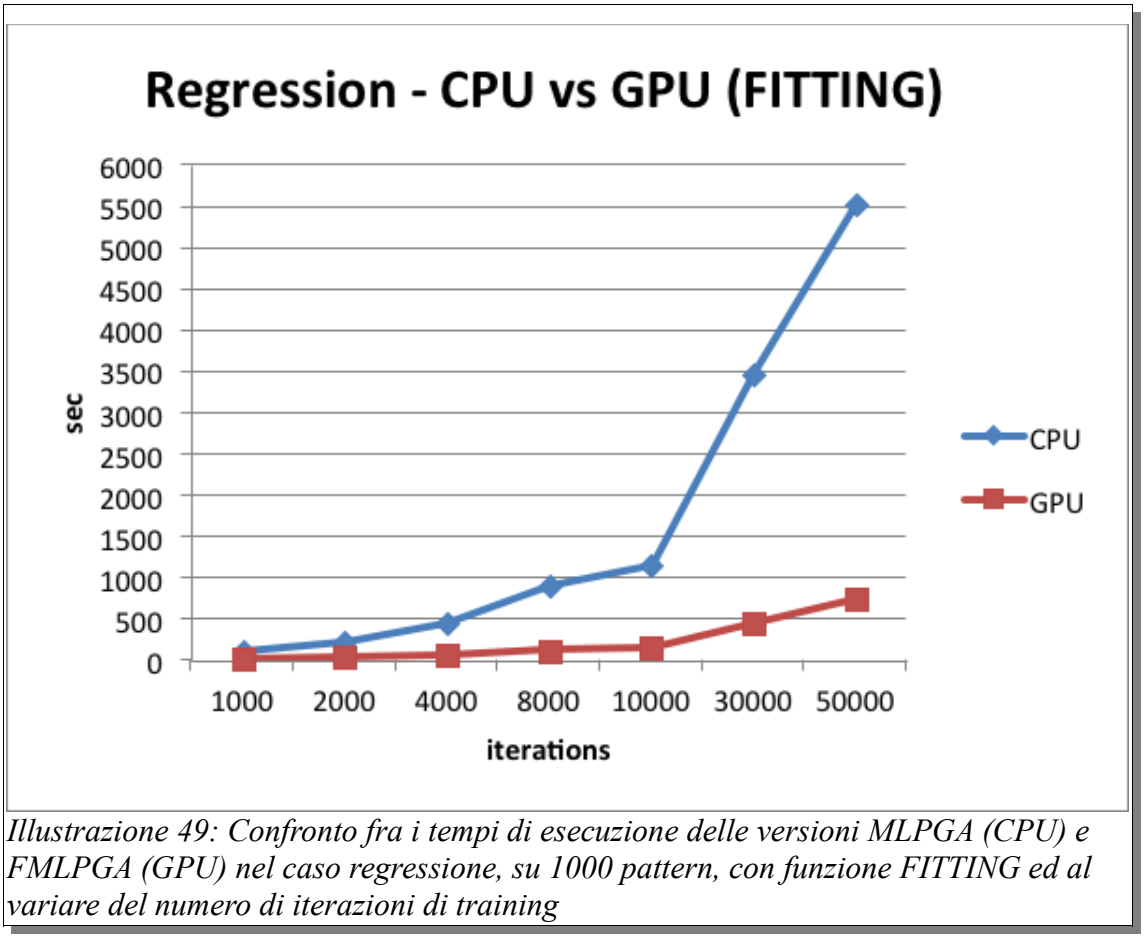


Illustrazione 49: Confronto fra i tempi di esecuzione delle versioni MLPGA (CPU) e FMLPGA (GPU) nel caso regressione, su 1000 pattern, con funzione FITTING ed al variare del numero di iterazioni di training

8.4.3. Valutazione finale delle prestazioni computazionali

I grafici riportati nelle sezioni precedenti evidenziano una sensibile diminuzione del tempo di esecuzione tra le due versioni seriale e parallela.

In termini di indice di speedup, ossia calcolando la media dei rapporti tra tempo di esecuzione dell'algoritmo seriale e tempo di esecuzione nel caso parallelo, otteniamo i valori indicati nei seguenti grafici.

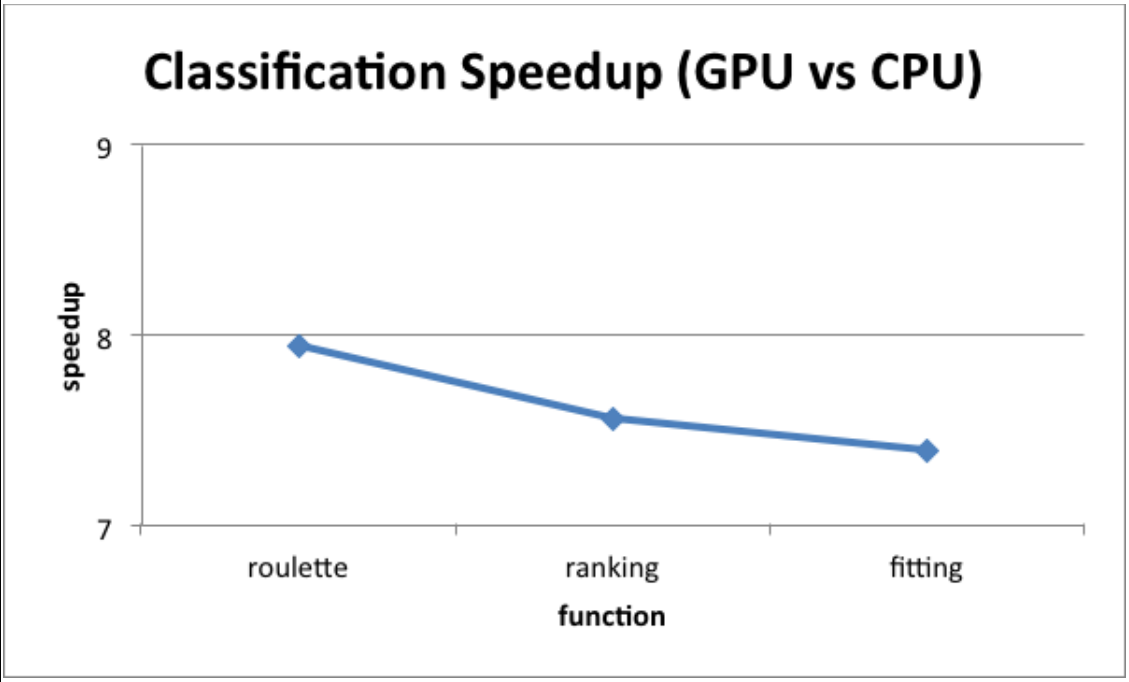


Illustrazione 50: Speedup dal confronto fra i tempi di esecuzione delle versioni MLPGA (CPU) e FMLPGA (GPU) nel caso classificazione, su 1000 pattern, al variare della funzione di selezione genetica

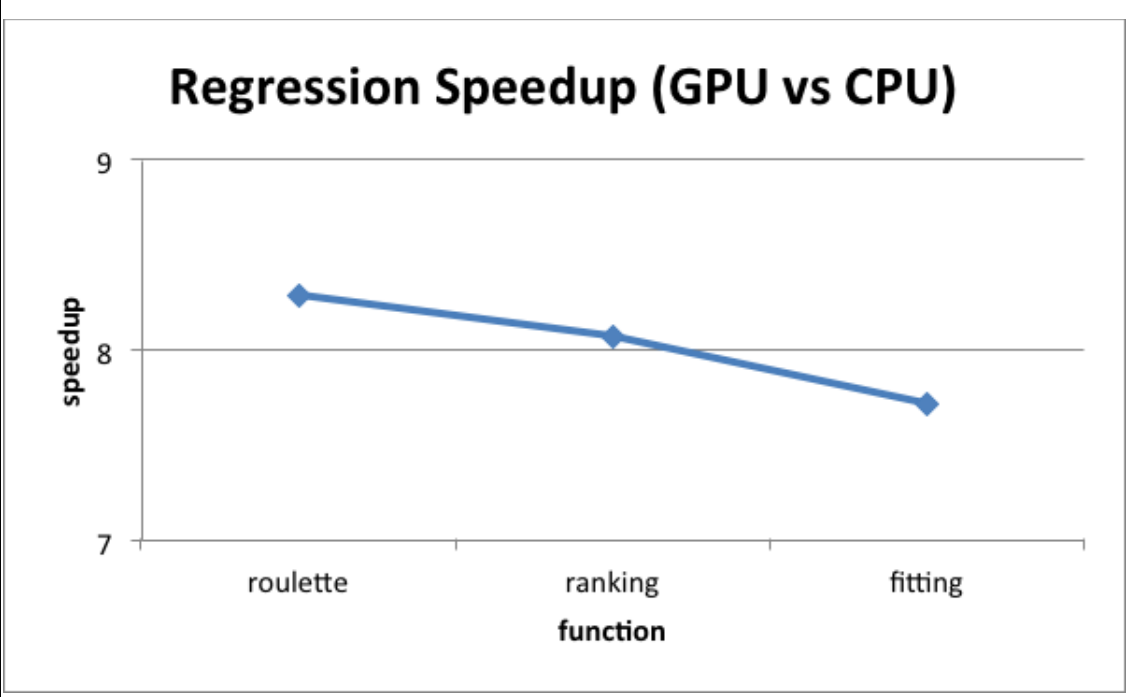


Illustrazione 51: Speedup dal confronto fra i tempi di esecuzione delle versioni MLPGA (CPU) e FMLPGA (GPU) nel caso regressione, su 1000 pattern, al variare della funzione di selezione genetica

Il guadagno in termini di minor tempo computazionale risulta pressochè simile in entrambi i casi di classificazione e regressione. La variazione in base alla funzione di selezione evolutiva genetica è dovuta alle minime differenze nei calcoli delle diverse funzioni. Considerando che la funzione ROULETTE è la più complessa fra le tre utilizzate, possiamo affermare che la versione parallelizzata del modello MLPGA ottiene uno speedup pari a 8x rispetto a quella seriale.

L'impiego del sistema di calcolo parallelo, basato su GPU+CUDA permette quindi di rendere pienamente efficiente e scalabile il modello MLPGA.

9. Conclusioni e sviluppi futuri

Il lavoro originale di questa tesi ha toccato diversi argomenti, integrando molteplici aspetti inter-disciplinari sia tecnologici che scientifici, spaziando dall'astrofisica al'ICT (Information & Communication Technology), passando per il data mining e l'intelligenza artificiale.

Prima di tutto è stato esplorato ed utilizzato lo stato dell'arte delle tecnologie computazionali, al fine di individuare e sfruttare le migliori soluzioni al problema di rendere efficiente e scalabile un modello ibrido di intelligenza artificiale per l'analisi auto-adattiva di grandi volumi di dati. In tal senso, si è proceduto alla progettazione e sviluppo di una rete neurale (Multi Layer Perceptron) addestrata con un algoritmo genetico, coniugando quindi le ben note capacità di generalizzazione di un modello neurale con le prerogative statistico-evolutive di modelli ispirati alla legge di selezione naturale, teorizzata da Darwin circa due secoli or sono.

La recente tecnologia informatica del calcolo parallelo a basso costo e ad alta granularità (Graphics Processing Unit) è stata poi applicata a problematiche scientifiche, nella fattispecie problemi aperti e complessi in ambito astrofisico, quali la classificazione di oggetti dell'Universo a partire da immagini e spettri osservati dai grandi telescopi, affetti da una notevole quantità di rumore di fondo ed alla possibilità di localizzare in modo preciso le distanze degli oggetti celesti a grande distanza dalla Terra, potendo estrapolare la terza dimensione (profondità) da dati (immagini) intrinsecamente bi-dimensionali.

Il connubio quindi tra tecnologia di calcolo parallelo e capacità di generalizzazione dell'intelligenza artificiale ha quindi permesso di affrontare e risolvere uno dei problemi più annosi della Scienza moderna: la capacità di esplorare ed analizzare efficientemente, in tempi rapidi ed in maniera pressochè automatica, enormi quantità di dati (dell'ordine di TB/giorno), acquisiti e memorizzati quotidianamente da strumenti e da simulazioni ogni giorno più avanzati e precisi, impossibili quindi da studiare per un cervello umano in tempi

ragionevoli. Il presente lavoro quindi s'inquadra nell'ambito della cosiddetta e-science, ossia della Scienza data-centrica, alla base del quarto paradigma della Scienza, dopo teoria, sperimentazione e simulazione.

Lo sfruttamento della tecnologia GPGPU+CUDA, ha portato alla generazione di un modello di data mining, basato sul paradigma supervisionato di Machine Learning, la cui parallelizzazione ha permesso di ottenere un guadagno medio di tempo computazionale (speedup) pari a circa 8x, rispetto all'omologa implementazione seriale su moderne CPU multi-core. Inoltre, tale modello è stato validato scientificamente su problemi reali di tipo astrofisico, raggiungendo quindi il duplice obiettivo originario di ottenere un sistema di data mining efficiente, economico e ad alta scalabilità rispetto ai dati da analizzare. Nondimeno l'intrinseca natura trasversale di un siffatto sistema di classificazione e regressione, rispetto alle analoghe problematiche presenti in svariate discipline scientifiche, mediche e/o sociali (cioè tutte le discipline umane basate sull'analisi di dati), rende il sistema realizzato un valido ed eclettico strumento di esplorazione dati in molteplici settori.

Nel prossimo futuro, si intende procedere all'ulteriore ottimizzazione dell'implementazione parallela del modello, nonché ad un suo intensivo sfruttamento nei settori dell'astrofisica sperimentale, laddove cioè vi sia la necessità di classificare ed estrapolare relazioni non note analiticamente tra grandi quantità di dati multi-banda e multi-epoca. Non ultimo vi è anche l'obiettivo di rendere il modello parallelo fruibile all'utenza esterna, integrandolo all'interno della web application DAMEWARE, una suite di data mining disponibile all'indirizzo http://dame.dsf.unina.it.

10. Appendice A: esempi di codice sorgente

In questo capitolo mostreremo le porzioni di codice delle principali funzioni che caratterizzano il modello, sia in versione seriale che in versione parallelizzata.

10.1. Funzione di Forwarding

In questo paragrafo verranno presentate le funzioni di forwarding per il modello seriale (MLPGA) e quello parallelo (FMLPGA)

10.1.1. Versione seriale

Di sotto viene mostrato il codice della funzione "run()", cioè la funzione che si occupa della fase di forwarding nel modello seriale

CPU void run(MLPNet* n,double* input, double* dna)

```
{
    int ws = 0;
    int i,j,k;
    //copies the DNA on MLP
    for (i=0; i<n->layers_size; i++)
    {
        for (j=0; j<n->layers[i]->neurons_size; j++)
        {
            for (k=0; k < n->layers[i]->neurons[j]->weights_size; k++)
            {
                n->layers[i]->neurons[j]->weights[k]=dna[ws++];
            }
        }
    }

    i=j=k=0;
    double totInput, bias;

    //execute the forward fase
    for (int i=0; i<n->layers_size; i++)
    {
        if (i == 0)
        {
            for (j=0; j<n->layers[i]->neurons_size; j++)
            {
                totInput = 0;
                for (k=0; k<(n->layers[i]->neurons[j]->weights_size)-
1; k++)
                {
                    totInput += n->layers[i]->neurons[j]->weights[k] *
```

```
input[k];
                }
                bias = n->layers[i]->neurons[j]->weights[(n-
>layers[i]->neurons[j]->weights_size)-1];
                double vt = executeActFunc(n->layers[i],totInput) -
bias;
                n->layers[i]->neurons[j]->value=vt;
            }
        }
        else
        {
            for (j=0; j<n->layers[i]->neurons_size; j++)
            {
                totInput = 0;
                for (k=0; k<(n->layers[i]->neurons[j]->weights_size)-
1; k++)
                {
                    totInput += n->layers[i]->neurons[j]->weights[k] *
n->layers[i-1]->neurons[k]->value;
                }
                // now calculate and subtract the neuron bias value
                bias = n->layers[i]->neurons[j]->weights[(n-
>layers[i]->neurons[j]->weights_size)-1];
                double vt2 = executeActFunc(n->layers[i],totInput) -
bias;
                n->layers[i]->neurons[j]->value=vt2;
            }
        }

    }
}
```

10.1.2. Versione parallela

Viene proposto ora una versione riadattata per CUDA della funzione "*run()*" appena mostrata. Come ovvio che sia, questa funzione è scritta per la struttura MLP4Cuda

```
GPU void MLP4Cuda_Run(MLP4Cuda MLP, double* input)
{
	int i;
	int j;
	double totInput;

	for(i=0;i<MLP.Layer0_size;i++)
	{
	totInput=0;
	for(j=0;j<MLP.Layer0_numWeights-1;j++)
		totInput += MLP4Cuda_getNeuronWeight(MLP,0,i,j)*input[j];

MLP4Cuda_setNeuronValue(MLP,0,i,MLP4Cuda_executeActFunc(MLP,0,totInput
) - MLP4Cuda_getNeuronWeight(MLP,0,i,j));
	}
	if(MLP.numLayers==1) return;

	for(i=0;i<MLP.Layer1_size;i++)
	{
		totInput=0;
		for(j=0;j<MLP.Layer1_numWeights-1;j++)
			totInput += MLP4Cuda_getNeuronWeight(MLP,1,i,j)*
MLP4Cuda_getNeuronValue(MLP,0,j);


MLP4Cuda_setNeuronValue(MLP,1,i,MLP4Cuda_executeActFunc(MLP,1,totInput
) - MLP4Cuda_getNeuronWeight(MLP,1,i,j));
	}
	if(MLP.numLayers==2) return;

	for(i=0;i<MLP.Layer2_size;i++)
	{
		totInput=0;
		for(j=0;j<MLP.Layer2_numWeights-1;j++)
			totInput += MLP4Cuda_getNeuronWeight(MLP,2,i,j)*
MLP4Cuda_getNeuronValue(MLP,1,j);


MLP4Cuda_setNeuronValue(MLP,2,i,MLP4Cuda_executeActFunc(MLP,2,totInput
) - MLP4Cuda_getNeuronWeight(MLP,2,i,j));
	}
	return;
}
```

10.2. Calcolo del fitness

In questo paragrafo mostreremo le differenze implementative dei due modelli in analisi per quanto riguarda la fase di calcolo del valore di fitness degli individui della popolazione.

10.2.1. Versione Seriale

Di sotto verrà mostrato una porzione di codice della funzione "train_mode()" che si occupa del calcolo dei valori di fitness.

```
...
for (j=0; j<M->Pop->population_size; j++)
{
      currentErr = 0.0;
      if(parallel==0)
      for (k=0; k<M->Par->numDatasetPatterns; k++)
      {
            run(M->Net,M->Par->input[k], M->Pop->popv[j]->DNA);
            currentErr += globalError(M->Net,M->Par->target[k]);
      }
            batch_error =sqrt(currentErr/M->Par->numDatasetPatterns);
            M->Pop->popv[j]->fitness=batch_error;

}
...
```

10.2.2. Versione Parallela

Verrà mostrato ora l'implementazione parallela del calcolo del valore di fitness. Questa operazione viene effettuata dalle funzioni "cudaTrainOnPop()" e parallelForward(), che si occupano rispettivamente di preparare le risorse necessarie e dell'esecuzione del parallelismo vero e proprio.

10.2.2.1. cudaTrainOnPop()

In questo paragrafo verrà mostrato il codice sorgente della funzione "cudaTrainOnPop()" che si occupa di:

- Preparare le varibili che utili alla fase parallela
- Avvia la funzione kernel "parallelForward()"
- Copia i risultati ottenuti da parallelForward() e situati sulla GPU in variabili allocate sull'host.

```
CPU void cudaTrainOnPop(dim3 grid,dim3 block,MLP4Cuda* nets,double*
input, double* output,Population* pop,Params* par,double*
d_batcherror)
{
            double* h_batcherror;
            double* d_dna;
            d_dna=putThisPopulationOnGPUArray(pop);

            h_batcherror=(double*)malloc(par->numDatasetPatterns*pop-
>population_size*sizeof(double));

      parallelForward<<<grid,block>>>(nets,input,output,d_dna,pop-
>population_size,par->numInputNeurons,par->numOutputNeurons,par-
>numHiddenLayers,par->numHidden1Neurons,par->numHidden2Neurons,par-
>H1ActFunc,par->H2ActFunc,par->OutActFunc,d_batcherror);

            if(cudaMemcpy(h_batcherror,d_batcherror,par-
>numDatasetPatterns*pop-
>population_size*sizeof(double),cudaMemcpyDeviceToHost)!=cudaSuccess)
            {
                  puts("ERROR 019:  Unable to copy batcherror on
GPU!\nThe program will be terminated\n");
                  exit(0);
            }
            int ws=0;
```

```
            for(int i=0;i<pop->population_size;i++)
            {
                  pop->popv[i]->fitness=0;

                  for(int j=0;j<par->numDatasetPatterns;j++)
                  {
                        pop->popv[i]->fitness+=h_batcherror[ws];
                        ws++;
                  }
                  pop->popv[i]->fitness=sqrt(pop->popv[i]-
>fitness/par->numDatasetPatterns);
            }
            free(h_batcherror);
            cudaFree(d_dna);
}
```

10.2.2.2. parallelForward()

La funzione presentata in questo paragrafo è il cardine del modello parallelo di MLP. *"parallelForward()"* si occupa della esecuzione parallela della fase di forward sui vari cores della GPU Nvidia.

```
KERNEL void parallelForward(MLP4Cuda* nets, double* input,double*
output, double* dna,int dimPop,int numInputNeurons,int
numOutputNeurons,int numHiddenLayers,int numHidden1Neurons,int
numHidden2Neurons,int H1ActFunc,int H2ActFunc,int OutActFunc,double*
batcherror)
{
	int idx2 = (threadIdx.x * gridDim.x) + blockIdx.x;
	int nWeights;
	nWeights=MLP4Cuda_CalcDimWeights(nets[idx2]);
	double* myoutput;
	if(nets[idx2].numLayers==1)
		myoutput=&output[(nets[idx2].Layer0_size)*blockIdx.x];
	if(nets[idx2].numLayers==2)
		myoutput=&output[(nets[idx2].Layer1_size)*blockIdx.x];
	if(nets[idx2].numLayers==3)
		myoutput=&output[(nets[idx2].Layer2_size)*blockIdx.x];
	__syncthreads();

	nets[idx2].weights=&dna[(nWeights*threadIdx.x)];

MLP4Cuda_Run(nets[idx2],&input[nets[idx2].INPUT_size*blockIdx.x]);

	batcherror[idx2]=MLP4Cuda_globalError(nets[idx2],myoutput);
}
```

10.3. Funzione di Training

Mostriamo infine la tunzione "train_mode()" che si occupa della fase di training sia per la modalità parallela, che per quella seriale.

```
CPU void train_mode(MLPGACtrl* M, int parallel)
{
	int j,k;
	double currentErr=0;
	int i=0;
	double batch_error = 0.0;
	double* d_input;
	double* d_output;
	double bestFit = 99999999;
	double* errors_list;
	dim3 block;
	dim3 grid;
	resetTrainErrorFile(M->Par->errorLogFilename);
	if(parallel==1)
	{
		printf("Setting CUDA parameters.............");
		d_input=putThisMatrixOnGPUArray(M->Par->input,M->Par-
>numDatasetPatterns,M->Par->numInputNeurons); //creates an array
version of the matrix token as input and put this copy into the device
		d_output=putThisMatrixOnGPUArray(M->Par->target,M->Par-
>numDatasetPatterns,M->Par->numOutputNeurons); //creates an array
version of the matrix token as input and put this copy into the device
		block.x=M->Par->popSize;;
		block.y=1;
		block.z=1;
		grid.x=M->Par->numDatasetPatterns; //sets the dimension of
the grid (number of blocks). It's always the same dimension of the
inputDataset Rows.
		grid.y=1;
		grid.z=1;
		errors_list=allocateErrorListOnGPU(M->Par-
>numDatasetPatterns,M->Par->popSize);
		puts("DONE!");
	}

	puts("\nI'm starting the Training mode...");
	while (bestFit > M->Par->errorThreshold)
	{
		if(parallel==0)
		{
			for (j=0; j<M->Pop->population_size; j++)
			{
				currentErr = 0.0;
				if(parallel==0)
					for (k=0; k<M->Par->numDatasetPatterns;
```

```
k++)
                                    {
                                          run(M->Net,M->Par->input[k], M-
>Pop->popv[j]->DNA);
                                          currentErr += globalError(M->Net,M-
>Par->target[k]);
                                    }
                              batch_error =sqrt(currentErr/M->Par-
>numDatasetPatterns);
                              M->Pop->popv[j]->fitness=batch_error;

                        }
                  }
                  else
                  {
                        cudaTrainOnPop(grid,block,M->NetsArray,d_input,
d_output,M->Pop,M->Par,errors_list);
                  }

                  bestFit = best(M->Pop)->fitness;

                  if (i % M->Par->showErrorFrequency == 0)
                  {
                        printf("%d - Best:%lf   Worst:
%lf\n",i,bestFit,worst(M->Pop)->fitness);
                        saveTrainError(M->Par->errorLogFilename,bestFit);
                        saveBestWeightsOnFile(M->Par-
>weightOutputFilename,best(M->Pop));
                  }
                  i++;

                  if(i>=M->Par->maxTrainingEpochs || bestFit<=M->Par-
>errorThreshold)
                  {
                        printf("%d - Best:%lf   Worst:
%lf\n",i,bestFit,worst(M->Pop)->fitness);
                        saveTrainError(M->Par->errorLogFilename,bestFit);
                        saveBestWeightsOnFile(M->Par-
>weightOutputFilename,best(M->Pop));
                        saveCompleteOutputOnFile(M->Par,M->Net,M->Par-
>input,best(M->Pop)->DNA,M->Par->train_completeOutputFile);
                        break;
                  }
                  else
                        nextPop(M->Pop);
            }
}
```

11.Appendice B: funzioni di statisica

In questa sezione si riporta il codice sorgente relativo alle funzioni di statistica utilizzate per valutare i risultati del modello, sia per la classificazione che per la regressione.

11.1. Creazione della "Confusion Matrix"

In questo paragrafo viene mostrato il codice della funzione "classificationMatrix()", che si occupa di ricavarsi la matrice di confusione, contenente i dati statistici di un esperimento di classificazione.

```
CPU int classificationMatrix(Params* p, MLPNet* n, char* fileOut)
{
    FILE* f;
      FILE* fout;

    int num_class;
      int ref_class, out_class;
      int** conf_matrix;
      double* targval;
      double* outval;

      // assign number of classes
      if(p->numOutputNeurons < 2)
      {
            printf("\n\nConfusion matrix not created because too low
number of classes!\n");
            printf("Please, specify two classes at least\n\n");
            exit(1);
      }
      else
            num_class = p->numOutputNeurons;


    // allocate and initialize the confusion matrix and other service
vectors
    /* num_class is the number of classes */
      if (( conf_matrix = ( int** )malloc( num_class*sizeof( int* )))
== NULL )
      {
            printf("\nerror allocating rows for conf_matrix\n");
            exit(1);
      }
```

```
	// now allocate columns of matrix
	for ( int i = 0; i < num_class; i++ )
	{
		if (( conf_matrix[i] = ( int* )malloc( num_class )) ==
NULL )
		{
			printf("\nerror allocating columns for
conf_matrix\n");
			exit(1);
		}
	}
	if (( targval = ( double* )malloc( sizeof(num_class) )) ==
NULL )
		{
			printf("\nerror allocating targval\n");
			exit(1);
		}
	if (( outval = ( double* )malloc( sizeof(num_class) )) == NULL )
		{
			printf("\nerror allocating outval\n");
			exit(1);
		}

	for(int r = 0; r < num_class; r++)
		for(int c = 0; c < num_class; c++)
		conf_matrix[r][c] = 0;


    // now process the output to evaluate regression results

	// open results file
	fout = fopen(fileOut, "r");

	// loop on all input patterns
	for(int i = 0; i < p->numDatasetPatterns; i++)
	{
		double tempval;

		// read input columns, to jump to target and output column
		for(int j = 0; j < p->numInputNeurons; j++)
		{
			fscanf(fout, "%lf%*c", &tempval);
			// check first values read
			//if( i < 4 )
			//	printf("\n %s, tempval[%d] = %lf\n", fileOut,
i, tempval);
			//if( i >= (p->inputOutputRows-4) )
			//	printf("\n%s, tempval[%d] = %lf\n", fileOut, i,
tempval);
		}
```

```
// now read the target values
//if( ( i < 4 ) || ( i >= (p->inputOutputRows-4) ) )
//    printf("\n %s, targval[%d] = ", fileOut, i);
for(int c = 0; c < num_class; c++)
{
      fscanf(fout, "%lf%*c", &tempval);
      targval[c] = tempval;

//if( ( i < 4 ) || ( i >= (p->inputOutputRows-4) ) )
//    printf("%lf  ", targval[c]);
}

// now read the output values
//if( ( i < 4 ) || ( i >= (p->inputOutputRows-4) ) )
//    printf("\n %s, outval[%d] = ", fileOut, i);
for(int c = 0; c < (num_class-1); c++)
{
      fscanf(fout, "%lf%*c", &tempval);
      outval[c] = tempval;

//if( ( i < 4 ) || ( i >= (p->inputOutputRows-4) ) )
//    printf("%lf  ", outval[c]);

}
// the last column is without final comma
fscanf(fout, "%lf", &tempval);
outval[num_class-1] = tempval;

//if( ( i < 4 ) || ( i >= (p->inputOutputRows-4) ) )
//    printf("%lf  ", outval[num_class-1]);

// locate the reference target class
// this is the maximum among targets
ref_class = 0;
for(int c = 0; c < num_class; c++)
{
      if(targval[c] > targval[ref_class])
            ref_class = c;
}

//if(i < 3)
//    printf("\nref_class = %d", ref_class);

// locate the best output class
// this is the maximum closest to selected target
out_class = ref_class;
for(int c = 0; c < num_class; c++)
{
      if(outval[c] > outval[out_class])
            out_class = c;
}
```

```
            //if(i < 3)
            //    printf("  and out_class = %d\n", out_class);

            // now update confusion matrix
            conf_matrix[ref_class][out_class] += 1;

            //if(i < 3)
            //    printf("  conf_matrix[%d][%d] = %d\n", ref_class,
out_class, conf_matrix[ref_class][out_class]);

      }

      fclose(fout);

      // now open file where to store confusion matrix
      f = fopen(p->conf_matrixOutputFilename, "w+");

      // fill in the header of the file
    fprintf(f, "Classification Confusion Matrix on %d patterns\n", p-
>numDatasetPatterns);

      fprintf(f,     "           ");
      for(int c = 0; c < num_class; c++)
            fprintf(f, "class %d   ", c);
      fprintf(f, "\n");

      for(int c = 0; c < num_class; c++)
      {
            fprintf(f, "target %d     ", c);
            for(int r = 0; r < num_class; r++)
                  fprintf(f, "%d       ", conf_matrix[c][r]);
            fprintf(f, "\n");
      }


    double final_stat = 0.0;
      double* class_stat;
      if (( class_stat = ( double* )malloc( sizeof(num_class) )) ==
NULL )
            {
                  printf("\nerror allocating class_stat\n");
                  exit(1);
            }
      for(int r = 0; r < num_class; r++)
            class_stat[r] = 0.0;

    for(int r = 0; r < num_class; r++)
      {
            class_stat[r] += conf_matrix[r][r];
        final_stat    += conf_matrix[r][r];
```

```
      }
      final_stat /= p->numDatasetPatterns;
    final_stat *= 100.0;
    fprintf(f, "\n\n");
    fprintf(f, "classification accuracy (total number of well
classified objects): %.2lf %% \n", final_stat);
    for(int r = 0; r < num_class; r++)
      {
            int num_objects = 0;
            for(int c = 0; c < num_class; c++)
            {
                  num_objects += conf_matrix[r][c];
            }
            class_stat[r] /= num_objects;
            class_stat[r] *= 100.0;
            fprintf(f, "classification purity for class %d = %.2lf %
% \n", r, class_stat[r]);
      }
      fprintf(f, "\nThe contamination for each class is the dual of
purity\n");

      // deallocate dynamic arrays
      for ( int i = 0; i < num_class; i++ )
      {
            free(conf_matrix[i]);
      }
      free(conf_matrix);
      free(targval);
      free(outval);
      free(class_stat);

    fclose(f);

      return 0;
}
```

11.2. Creazione della "Regression Matrix"

Di seguito viene proposto il codice della funzione "regressionMatrix()" che si occupa della creazione della matrice di regressione contenente le informazioni statistiche dell'esperimento di regressione appena eseguito.

CPU int regressionMatrix(Params* p, MLPNet* n, char* fileOut)

```
{
    FILE* f;
      FILE* fout;

    int matrix_dim;
      int** conf_matrix;
      double minval = 9000.0;
      double maxval = -9000.0;
      double threshold;
      double current_threshold = p->errorThreshold;
      int intervals = 0;
      int* regr_stat;
      double* regr_intervals;
      double perc = 0.0;

    matrix_dim = p->numOutputNeurons*2;

    // initialize the confusion matrix and other service vectors
    /* matrix_dim is the number of rows */
      if (( conf_matrix = ( int** )malloc( matrix_dim*sizeof( int* ) ))
== NULL )
      {
            printf("\nerror allocating rows for conf_matrix\n");
            exit(1);
      }

      // now allocate columns of matrix
      for ( int i = 0; i < matrix_dim; i++ )
      {
            if (( conf_matrix[i] = ( int* )malloc( matrix_dim )) ==
NULL )
            {
                  printf("\nerror allocating columns for
conf_matrix\n");
                  exit(1);
            }
      }

      for(int r = 0; r < matrix_dim; r++)
        for(int c = 0; c < matrix_dim; c++)
            conf_matrix[r][c] = 0;
```

```
    ////////////////////////////////////////////////////////////////
//
    // found min and max of target vectors to fix the output threshold

      for(int i=0; i < p->numDatasetPatterns; i++)
      {
            double t = p->target[i][0];
        if(minval >= t)
                  minval = t;
        else if(maxval <= t)
                  maxval = t;
    }

    threshold = (minval + maxval)/2.0;
    ////////////////////////////////////////////////////////////////

    // now process the output to evaluate regression results

      // here we calculate a more tuned evaluation of output
      // based on the distance from the error threshold
      // given as input parameter
      while(current_threshold <= 0.1)
      {
            intervals += 1;
            current_threshold *= 10.0;
      }


      //printf("\nintervals = %d\n", intervals);

      if (( regr_stat = ( int* )malloc( sizeof(intervals) )) == NULL )
            {
                  printf("\nerror allocating regr_stat\n");
                  exit(1);
            }

      for(int k = 0; k < intervals;k++)
            regr_stat[k] = 0;

      if (( regr_intervals = ( double* )malloc( sizeof(intervals) ))
== NULL )
            {
                  printf("\nerror allocating regr_intervals\n");
                  exit(1);
            }

      double tenpower = 1.;
      for(int k = 0; k < intervals; k++)
      {
            regr_intervals[k] = p->errorThreshold * tenpower;
            tenpower *= 10.;
```

```
            //printf("\nregr_intervals[%d] = %.2lf\n", k,
regr_intervals[k]);
      }

      // open results file
      fout = fopen(fileOut, "r");

      int loc = 0;
      for(int i = 0; i < p->numDatasetPatterns; i++)
      {
            double rm_out, rm_targ;
            double tempval;
            double targval, outval;

            // read input+target columns, to jump to output column
            for(int j = 0; j < p->numInputNeurons; j++)
            {
                  fscanf(fout, "%lf%*c", &tempval);
                  // check first values read
                  //if( i < 4 )
                  //    printf("\n %s, outval[%d] = %lf\n", fileOut, i,
tempval);
                  //if( i >= (p->inputOutputRows-4) )
                  //    printf("\n%s, outval[%d] = %lf\n", fileOut, i,
tempval);
            }

            // now read the target value
            fscanf(fout, "%lf%*c", &targval);
            // now read the output value
            fscanf(fout, "%lf", &outval);

            // check first values read
            //if( i < 4 )
            //    printf("\n %s, targval[%d] = %lf -- outval[%d] =
%lf\n", fileOut, i, targval, i, outval);
            //if( i >= (p->inputOutputRows-4) )
            //    printf("\n %s, targval[%d] = %lf -- outval[%d] =
%lf\n", fileOut, i, targval, i, outval);

            double diff = fabs(targval-outval);
            //if( i < 3 )
                  //printf("\n %s, diff(%d) = %lf\n", fileOut, i,
diff);
            for(int k = 0; k < intervals; k++)
            {
                  if( diff <= regr_intervals[k] )
                  {
                        regr_stat[k] += 1;
                        break;
                  }
```

```
            }


            if( outval <= threshold )
                  rm_out = minval;
          else if( outval > threshold  )
                  rm_out = maxval;

            if( targval <= threshold )
            rm_targ = minval;
          else if( targval > threshold )
            rm_targ = maxval;

          if( (rm_out == rm_targ) && (rm_targ == minval) )
                  conf_matrix[loc][loc] += 1;
          else if( (rm_out == rm_targ) && (rm_targ == maxval) )
            conf_matrix[loc+1][loc+1] += 1;
          else if( (rm_out != rm_targ) && (rm_targ == minval) )
            conf_matrix[loc][loc+1] += 1;
          else if( (rm_out != rm_targ) && (rm_targ == maxval) )
            conf_matrix[loc+1][loc] += 1;
        }

        fclose(fout);


        // now open file where to store confusion matrix
        f = fopen(p->conf_matrixOutputFilename, "w+");

        // fill in the header of the file
      fprintf(f, "Regression Statistics results on %d patterns", p-
>numDatasetPatterns);
      fprintf(f, "\nIn the regression case the matrix is an adapted
confusion matrix\n");
      fprintf(f, "Regression matrix:\n\n");
      fprintf(f, "          Out 0     Out 1");
        fprintf(f, "\n");

      // calculation of final statistics on recognized patterns
      // and dump it in the log result file
      for(int r = 0; r < matrix_dim; r++)
        {
          fprintf(f, "target %d  ", r);
          for(int c = 0; c < matrix_dim; c++)
            {
            fprintf(f, "%d       ", conf_matrix[r][c]);
          }
            fprintf(f, "\n");
        }

      double final_stat = 0.0;
```

```
    for(int r = 0; r < matrix_dim; r++)
        final_stat += (double)conf_matrix[r][r];
    final_stat /= p->numDatasetPatterns;
    final_stat *= 100.0;
    fprintf(f, "\n");
    fprintf(f, "predicted pattern percentage = %.2lf %% based on
calculated threshold = %.2lf\n\n", final_stat, threshold);

      for(int k = 0; k < intervals; k++)
      {
            perc = ((double)regr_stat[k] / (double)p-
>numDatasetPatterns) * 100.;
            //printf("\n k = %d -- patterns = %d -- regr_stat = %d --
perc = %.2lf\n", k, p->inputOutputRows, regr_stat[k], perc);
            fprintf(f, "%.2lf %% (%d) of patterns within distance
%.5lf from target\n", perc, regr_stat[k], regr_intervals[k] );
      }

      for ( int i = 0; i < matrix_dim; i++ )
      {
            free(conf_matrix[i]);
      }
      free(conf_matrix);
      free(regr_stat);
      free(regr_intervals);

    fclose(f);
    return 0;
}
```

Bibliografia

Aha, W.; Kibler, D.; Albert, M.K., *Instance-Based Learning Algorithms*. 1991, Machine Learning, Kluwer Academic Publishers, Boston MA, USA, Vol. 6, pp.37-66

American National Standards Institute, et al. 1977, *American National Standard Code for Information Interchange.* The Institute.

Bishop, C.M., *Pattern Recognition and Machine Learning*, 2006, Springer ISBN 0-387-31073-8.

Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; Maler, E.; Yergeau, F.; Cowan, J., XML 1.1 (Second Edition). 2006, W3C *Recommendation*, http://www.w3.org/TR/2006/REC-xml11-20060816/

Brescia, M., Cavuoti, S., D'Abrusco, R., Longo, G., Mercurio, A., 2012b. Zphot prediction on WISE-GALEX-UKIDSS-SDSS Quasar Catalogue, based on the MLPQNA model. Submitted to MNRAS

Brescia, M., Cavuoti, S., Paolillo, M., Longo, G., Puzia, T., 2012a. The Detection of Globular Clusters in Galaxies as a data mining problem. MNRAS, 421, 2, 1155-1165

Brescia, M., Longo, G., 2012. Astroinformatics, data mining and the future of astronomical research, 2012. To appear in the Proceedings of the 2-nd International Conference on Frontiers in Diagnostic Technologies, Nuclear Instruments and Methods in Physics Research A, NIMA Elsevier Journal, arXiv:1201.1867

Cabena, P.; Hadjinian, P.; Stadler, R.; Verhees, J. & Zanasi, A., *Discovering Data Mining: From Concepts to Implementation.* 1998, Prentice Hall.

Darwin, C., (1859). On the Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life, London: John Murray, pp. 502.

Djorgovski, S.G., Longo, G., Brescia, M., Donalek, C., Cavuoti, S., Paolillo, M., D'Abrusco, R., Laurino, O., Mahabal, A., Graham, M., 2012. Data Mining and Exploration (DAME): New Tools for Knowledge Discovery in Astronomy. American Astronomical Society, AAS Meeting 219, 145.12

Fabbiano, G., Calzetti, D., Carilli, C., Djorgovski, S.G., 2010. Recommendations of the VAO Science Council. E-print arXiv:1006.2168v1 [astro-ph.IM].

Galton, F. 1877, *Typical laws of heredity*, Nature 15

Guenther, R.; Radebaugh, J., *Understanding Metadata*. 2004, National Information Standards Organization (NISO) Press, Bethesda MD, USA

Hastie, T.; Tibshirani, R.; Friedman, J.; Franklin, J., *The elements of statistical learning: data mining, inference and prediction*. The Mathematical Intelligencer, 2005, Springer New York, Vol. 27, pp. 83-85

Haykin, S., 1998, *Neural Networks - A Comprehensive Foundation* (2nd. ed.). Prentice-Hall, Upper Saddle River, NJ USA

Hey, T.; Tansley, S.; Tolle, K., *The Fourth Paradigm: Data-Intensive Scientific*

Discovery; ISBN-10: 0982544200, 2009; Microsoft Research, Redmond Washington, USA, 2009

Lorentz, G.G., Golitschek,M, Makovoz Y., 1996. Constructive approssimation: Advanced problems, Vol.304, Springer, Berlin

Kotsiantis, S. B., *Supervised Machine Learning: A Review of Classification Techniques,* Proceeding of the 2007 conference on Emerging Artificial Intelligence Applications in Computer Engineering, IOS Press Amsterdam, The Netherlands, 2007, Vol. 160, pp. 3-24

Inmon, B., *Building the Data Warehouse*. 1992. John Wiley and Sons. ISBN 0471569607

Menard, S. W. 2001. *Applied logistic regression analysis,* Sage University Papers Series on Quantitative Applications in the Social Sciences, Thousand Oaks, CA, Vol. 106

Moore, G.E., 1965. Cramming more components onto integrated circuits. Electronics Magazine, p. 4

Mosteller F.; Turkey J.W., *Data analysis, including statistics.* In Handbook of Social Psychology. Addison-Wesley, Reading, MA, 1968

NVIDIA Corp., 2012. CUDA C Best Practices Guide. NVIDIA Corporation Distribution, ed. 4.1.

Pasian, F.; Ameglio, S.; Becciani, U.; Borgani, S.; Gheller, C.; Manna, V.; Manzat o, P.; Marseglia, L.; Smareglia, R.; Taffoni, G., *Interoperability and integration*

of theoretical data in the Virtual Observatory. 2007, Highlights of Astronomy, IAU XXVI General Assembly, Vol. 14, p.632.

Provost, F.; Fawcett, T.; Kohavi, R., *The Case Against Accuracy Estimation for Comparing Induction Algorithms,* Proceedings of the 15th International Conference on Machine Learning, 1998, Morgan Kaufmann. pp. 445-553

Quenn, M.J.,2004, Parallel Programmin in C with MPI and OpenMP, McGraw-Hill

Repici, J., (2010), *How To: The Comma Separated Value (CSV) File Format*. 2010, Creativyst Inc. http://www.creativyst.com/Doc/Articles/CSV/CSV01.htm

Rosenblatt, F., *The Perceptron - a perceiving and recognizing automaton*. 1957, Report 85-460-1, Cornell Aeronautical Laboratory

Rumelhart, D.; Hinton, G.; and Williams, R., *Learning internal representations by error propagation*. 1986, In Parallel Distributed Processing, MIT Press, Cambridge, MA, chapter 8.

Samuel, A.L., 1990, Memorial Resolution. Stanford University, Historical society

Sutter, H., 2005. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. Dr. Dobb's Journal, 3, 30.

Tozzi, M., 2010. *Il modello di Calcolo di un esperimento di Fisica*

Wells, D.C.; Greisen, E.W.; Harten, R.H., FITS: a *Flexible Image transport*

System. 1981, Astronomy & Astrophysics Supplement Series, Vol. 44, p. 363

Witten, I.H.; Frank, E., Data *Mining: Practical machine learning tools and techniques.* 2005, 2nd Edition, Morgan Kaufmann, San Francisco, USA

Ringraziamenti

Desidero ringraziare vivamente il prof. Guido Russo per avermi dato la possibilità di vivere questa esperienza formativa all'interno dello S.Co.P.E, ampliando così le mie conoscenze sulla materia.

Inoltre, un ringraziamento speciale va al dott. Massimo Brescia, per la sua infinita pazienza e per avermi sostenuto nella stesura di questa tesi.

GRAZIE