Alma Mater Studiorum · Università di Bologna

DIPARTIMENTO DI **INFORMATICA - SCIENZA E INGEGNERIA**
Laurea Magistrale in **Ingegneria Informatica**
Tesi Magistrale in **Intelligent Systems**

Deep Learning Models with Stochastic Targets: an Application for Transprecision Computing

Relatrice:
Chiar.ma Prof.ssa
MILANO MICHELA

Correlatore:
Dr.
BORGHESI ANDREA

Candidato:
BOSCARINO ANDREA

Sessione IV
Anno Accademico 2018-2019

Indice

Introduzione **7**

1 Transprecision Computing **9**
1.1 Motivazioni . . . 10
1.2 Progetto OPRECOMP . . . 11
1.3 Approccio proposto . . . 14

2 Machine learning **17**
2.1 Regressori . . . 21
2.2 Reti neurali . . . 23
2.2.1 Deep neural networks . . . 28
2.2.2 Autoencoder . . . 29
2.2.3 Funzioni di loss e metriche qualitative . . . 31
2.3 Transfer Learning . . . 33

3 Strumenti utilizzati **35**
3.1 Environment di base . . . 35
3.2 Tensorflow . . . 37
3.3 Keras . . . 38
3.4 Keras Tuner . . . 40
3.5 TensorBoard . . . 43

4 Progetto **45**
4.1 Studio delle relazioni di dominanza tra configurazioni di bit . . . 46
4.2 Creazione di un modello stocastico di base . . . 49
4.2.1 Un nuovo modello con Maximum Likelihood Estimation . . . 50
4.2.2 Tuning del modello . . . 52
4.2.3 Esperimenti e risultati . . . 53
4.3 Miglioramento del modello attuale a partire da considerazioni teoriche . . 55
4.3.1 Considerazioni sulla distribuzione degli errori . . . 56
4.3.2 Esperimenti col filtraggio dei dataset . . . 58

4.3.3 Considerazioni sulla conoscenza del dominio 60
4.3.4 Esperimenti con l'aggiunta di feature del grafo al dataset 62
4.3.5 Combinazione dei due metodi e riepilogo dei risultati 64
4.4 Esperimenti di codifica degli Input Set in forma compressa 67
4.4.1 Utilizzo di un autoencoder per ridurre la dimensionalità dei vettori dell'Input Set 68
4.5 Una riflessione parallela: il modello di regressione *ML_mean* 72
4.5.1 Implementazione di *ML_mean* 73
4.5.2 Esperimenti e risultati 76
4.6 Un nuovo modello con feature statistiche ricavate degli Input Set: *ML_MLE* 78
4.6.1 Creazione delle feature aggiuntive e preprocessamento dei dataset 79
4.6.2 Implementazione e tuning di *ML_MLE* 84
4.6.3 Esperimenti e risultati 86

5 Risultati sperimentali **87**

Conclusioni **93**

Bibliografia **99**

Introduzione

L'elaborato di tesi che si va ad analizzare è parte di un ampio progetto finanziato dall'Unione Europea, sotto il programma Horizon 2020 per la ricerca e l'innovazione, Open Transprecision Computing (OPRECOMP). Il progetto, della durata di 4 anni, punta a superare l'assunto conservativo secondo cui ogni calcolo compiuto da sistemi e applicazioni computazionali debba essere eseguito utilizzando la massima precisione numerica. Tale assunto è finora risultato sensato in vista di un'efficienza computazionale sempre migliore col passare del tempo, secondo la legge di Moore. Com'è noto, nell'era attuale tale legge ha iniziato a perdere di validità con l'approssimarsi dei limiti fisici che impediscono ulteriori miglioramenti di grande ordine previsti di anno in anno, dando piuttosto spazio a miglioramenti marginali.

L'approccio proposto dal progetto OPRECOMP (il cui sviluppo vuole beneficiare un range di applicazioni che spaziano dai piccoli nodi computazionali per l'Internet-of-Things, fino ai centri computazionali di High Performance Computing) è basato sul paradigma del Transprecision Computing, che supera l'assunto della massima precisione in favore di calcoli approssimati; tramite tale paradigma si arriva ad un doppio vantaggio: computazioni più efficienti e brevi, e soprattutto, risparmio energetico.
Per fare ciò, OPRECOMP fa leva sul principio secondo cui quasi ogni applicazione utilizza nodi intermedi di calcolo, le cui precisioni possono essere "limate" (in modo controllato e limitabile) con conseguenze minime sull'affidabilità dei risultati successivi, in quanto l'utente finale è interessato solamente alla validità del risultato della computazione finale. In definitiva, il progetto punta a portare un'alternativa all'approccio di computazione classico, alternativa tramite la quale verranno pensati e progettati i sistemi del futuro nel campo del data-mining e di applicazioni human-consuming (in cui la percezione - imprecisa - umana gioca un ruolo fondamentale).

Nei capitoli successivi verranno applicati i suddetti concetti tramite un'esplorazione di soluzioni di machine learning con lo scopo di studiare il trade-off tra numero di bit utilizzati per ogni step della computazione complessa (su svariati benchmark) e il relativo errore rilevato rispetto alla stessa computazione eseguita a precisione massima.
In particolare si parlerà di problemi di regressione con un target stocastico: a differenza dei problemi di regressione "classici", in cui si utilizzano una label e un target numerico per il learning, nell'approccio in esame si è scelto di considerare l'errore generato tra i risultati dei benchmark ottenuti con approssimazione delle variabili di input e quelli con massima precisione (al quale verrà fatto successivamente riferimento semplicemente con la parola "errore") al fine di allenare un modello stocastico, ovvero una distribuzione probabilistica caratterizzata da errore medio e varianza.
Durante la parte di progetto, quindi, tramite un'accurata disamina degli approcci adottati per la creazione dei modelli stocastici, si cercherà di affrontare la questione fondamentale dell'elaborato, ovvero osservare come varia l'errore generato in base ai dati che vengono dati in input ai benchmark, data una certa configurazione di bit di precisioni per le relative variabili.
Verrà infine mostrata una comparazione riassuntiva dei risultati ottenuti durante il lavoro di tesi, evidenziando vantaggi e svantaggi delle soluzioni operative associate.

1 Transprecision Computing

Per parlare di Transprecision Computing bisogna partire dal paradigma dell'Aproximate Computing [1][2], ovvero un ampio insieme di tecniche che mirano a rilassare i vincoli di precisione della computazione con lo scopo di migliorare performance, consumo di energia e ulteriori metriche di interesse.
Fino a tempi recenti, la riduzione del tempo di computazione e le conseguenti implicazioni sul risparmio energetico sono state guidate dalla legge di Moore, la quale - come sappiamo - ha man mano perso di validità nel corso degli anni, costringendo quindi alla ricerca attiva di una strada alternativa.

Il paradigma dell'Aproximate Computing, sfrutta il fatto che svariate applicazioni di interesse, in particolare quelle legate ad ambiti multimediali e di machine learning, non hanno necessariamente bisogno di produrre risultati con una precisione massima.
Come accennato durante l'introduzione, è il caso ad esempio di applicazioni dove la percezione umana (in quanto imprecisa) gioca un ruolo fondamentale, come nel banale caso della compressione di file multimediali.
L'interesse nel paradigma dell'Aproximate Computing, però, è recentemente aumentato dal momento in cui tale approccio permette potenzialmente di poter ridurre il consumo energetico durante l'uso di applicazioni (sia hardware che software) di interesse. Basti pensare come, al giorno d'oggi, la maggior parte delle computazioni è eseguita sia da device mobili che da grandi data center, e nonostante la differenza nelle dimensioni e nell'applicazione di riferimento, entrambe le piattaforme sono sottoposte a consumo energetico.

1.1 Motivazioni

Di per sé l'approccio dell'Aproximate Computing punta ad essere applicato solo a problemi e applicazioni specifiche; inoltre l'approssimazione di calcolo è introdotta solo in alcune routine e manca quindi di una dimensione di automazione del processo; a causa di ciò e della conseguente mancanza di precisione adattiva, l'approccio porta tipicamente a una perdita qualitativa sostanziale del risultato finale; infine non c'è modo di controllare o porre dei limiti sull'errore di computazione ottenuto.

Passiamo quindi ad una specializzazione del suddetto approccio, arrivando quindi al paradigma del Transprecision Computing, il quale supera i difetti poc'anzi evidenziati; questo nuovo approccio infatti gestisce l'approssimazione nello spazio e nel tempo in maniera adattiva, attraverso controlli software e hardware, permettendo quindi una precisione flessibile, un'accuratezza dei risultati adeguata, e soprattutto una buona scalabilità (che si tratti di un'applicazione di approssimazione nel range dei mW o in quello dei MW).
L'obiettivo principale del Transprecision Computing, è infatti quello di ottenere dei miglioramenti di almeno un ordine di grandezza nell'efficienza energetica, riuscendo ad ottenere una scalabilità tale da poter essere utilizzato in dominii che spaziano dell'IoT, passando per i Big Data, per il Deep Learning, fino agli ambiti di High Performance Computing, il tutto sia da un lato hardware che da un lato software.

Il filone di ricerca da cui prende piede questo elaborato riguarda principalmente il lato software, in cui si va ad agire sul modo in cui vengono rappresentati i dati necessari per la computazione. È stato infatti dimostrato come una delle maggiori cause di consumo energetico durante le computazioni intensive sia l'utilizzo di numeri floating-point (FP) e le relative operazioni [3].
In particolare, si cerca di effettuare un tuning del numero di bit utilizzati per i calcoli, effettuando un'approssimazione in maniera controllata e dinamica, imponendo allo stesso tempo specifici vincoli di qualità del risultato.

1.2 Progetto OPRECOMP

Come anticipato durante l'introduzione, il progetto OPRECOMP, inserito nel programma europeo Horizon 2020, punta ad esplorare il paradigma del transprecision computing allo scopo di mantenere un aumento esponenziale dell'efficienza computazionale negli anni, in alternativa al classico schema basato su parallelismo di CPU multi-core e altre tipologie di architetture il cui progredire fa affidamento sulla legge di Moore (nonostante, come già detto, si stiano raggiungendo dei limiti fisici per cui questa legge sta perdendo di validità negli ultimi anni).
La missione di OPRECOMP è sia di dimostrare che questa idea di fondo possa reggere in un vasto scenario architetturale (dal dominio dell'IoT fino all'High Performance Computing), sia di mostrare come l'efficienza energetica raggiunta possa toccare nove ordini di grandezza (dai milli-Watt ai MegaWatt), il tutto sia in ambito hardware che software.

Per quanto riguarda l'ambito hardware, il bottleneck della computazione è solitamente dato dalla limitatezza della larghezza di banda di memoria; le memorie più utilizzate al giorno d'oggi sono le Dynamic Random Access Memories (DRAM), che sono le parti responsabili per una grandissima percentuale del consumo energetico, mentre il restante, modesto, consumo energetico è dato dalla computazione in sé. La soluzione proposta è quella di introdurre una versione approssimata delle DRAM, e di utilizzare degli appositi memory controller per pilotarle, portando così degli incrementi significativi in termini di efficienza energetica e banda di memoria disponibile.

All'interno dell'ambito software (dominio di questo elaborato) invece, l'azione è concentrata sulla modalità di rappresentazione dei dati utilizzati per la computazione. È stato infatti dimostrato come il bottleneck maggiore per l'efficienza energetica in ambito software sia l'esecuzione di operazioni floating-point (FP): alcuni studi hanno evidenziato come più del 50% dell'energia consumata durante la computazione dipenda solamente dal calcolo FP [3].
L'idea è quindi quella di adottare formati numerici che richiedano un numero minore di bit di precisione, cosa che di conseguenza permette la semplificazione dei circuiti aritmetici e la diminuzione del numero di passaggi dei dati da memoria a memoria, o da memoria a registro.

Purtroppo, nonostante questa opportunità di risparmio energetico, è ancora assente all'interno di contesti di sviluppo software una metodologia sicura per decidere quali variabili FP sono delle possibili candidate al processo di riduzione di precisione, e soprattutto quale formato (a precisione ridotta) è invece più conveniente usare.
Nonostante esistano già dei tool per il tuning della precisione di variabili, quali ad esempio PROMISE [5] e fpPrecisionTuning [6], e piattaforme di emulazione di FP a precisione ridotta, come Multiple Precision Floating-Point Reliably (MPFR), questi presentano diversi limiti, non essendo stati progettati tenendo conto dei molteplici scopi di questa ricerca; ad esempio, tali librerie non permettono di evidenziare l'evoluzione dell'errore accumulato durante il flusso del programma, o di stimare i costi/-benefici associati all'introduzione di tipi di dato a precisione ridotta.

A tale scopo è quindi importante citare il lavoro svolto da Tavaglini et al. [3], nella realizzazione di una piattaforma per l'esecuzione di operazioni FP a precisione variabile che colmi i vuoti delle suddette librerie, e risultando quindi perfetta per la ricerca nel campo del Transprecision Computing, ovvero FlexFloat.
Si tratta di una libreria software open-source, progettata espressamente per aiutare lo sviluppo di applicazioni Transprecision, e che espone delle API (per C/C++) per il supporto a formati FP multipli (sia standard, che definiti in modo custom).
A differenza delle sopracitate librerie, FlexFloat:

- Permette un'emulazione accurata di formati FP con una lunghezza arbitraria in bit di mantissa ed esponente (campi che definiscono interamente un numero FP);
- Utilizza una metodologia di emulazione che sfrutta i tipi di dato nativi della piattaforma sottostante per ridurre di molto i tempi richiesti per il tuning della precisione;
- Offre caratteristiche fondamentali per applicazioni Transprecision, come statistiche di runtime sulle operazioni effettuate e un sofisticato meccanismo di tracciamento degli errori definito a livello di singola variabile.

L'apporto di questa libreria è fondamentale per il presente elaborato, in quanto i vari esperimenti verranno condotti sui seguenti benchmark matematici, precedentemente riadattati durante il lavoro di Tagliavini et al. per essere eseguiti su piattaforma FlexFloat, qui rappresentati in ordine di cardinalità crescente delle variabili:

- *Fast Walsh Transform (FWT)* per i vettori reali; cardinalità 2;
- *saxpy*: addizione vettoriale che moltiplica un vettore di input, X, per uno scalare a, e somma tutto ad un secondo vettore, Y; cardinalità 3;
- *convolution*: convoluzione di una matrice con kernel 11x11; cardinalità 4;
- *Discrete Wavelet Transform (dwt)*, dalla teoria dei segnali; cardinalità 7;
- *correlation*: calcolo di una matrice di correlazione dell'input; cardinalità 7;
- *BlackScholes*: operazione in ambito finanziario che stima il prezzo per un set di opzioni applicando l'equazione differenziale parziale di Black-Scholes; cardinalità 15;
- *Jacobi*: algoritmo iterativo utilizzato per determinare la soluzione di un sistema di equazioni lineari diagonalmente dominanti; cardinalità 25.

All'interno dell'elaborato ci si riferirà agli insiemi di valori dati alle variabili di input, durante le sperimentazioni con i benchmark, con il termine "input set". I risultati di queste sperimentazioni (sotto forma tabulare), invece, saranno semplicemente indicati come "dataset", i quali conterranno gli errori generati utilizzando una certa configurazione di bit per le variabili, e uno specifico input set per popolare queste ultime.

1.3 Approccio proposto

Questo elaborato si inserisce all'interno di un ampio macro-ambito esplorato all'interno del progetto OPRECOMP, ovvero quello dell'applicazione della metodologia dell'Empirical Model Learning (EML) [4] per affrontare la ricerca del tuning ottimale del il numero di bit (che le variabili dei benchmark menzionati nel paragrafo precedente possono assumere) minimizzando l'errore prodotto.

Solitamente, il processo di decision-making all'interno di problemi del genere si basa sulla progettazione di modelli predittivi che operano su dati reali, e che vengono utilizzati nella cosiddetta what-if analysis: durante questa analisi vengono ripetutamente sottoposti degli scenari (o insiemi di decisioni) ai modelli predittivi allo scopo di estrarre i valori di certe variabili di interesse; inevitabilmente, seguendo questo approccio, viene analizzato solo un numero limitato di scenari, per poi scegliere quello che produce i risultati migliori; purtroppo, in problemi come quello di nostro interesse, lo spazio decisionale può potenzialmente essere così grande che non è raro doversi trovare a selezionare scenari che in realtà non sono ottimali.
L'EML rivoluziona l'approccio, in quanto utilizza una parte di Machine Learning (ML) per ottenere dei modelli predittivi che imparano una relazione approssimativa tra le decisioni e il loro impatto sul sistema, e una parte in cui queste relazioni vengono inserite all'interno di un problema di ottimizzazione (Mathematical Programming, MP).
In questo modo, un sistema basato su EML potrebbe essere capace di suggerire decisioni ottimali in problemi reali complessi, grazie anche ai recenti sviluppi sull'analisi dei dati e sulla progettazione dei modelli predittivi.

Nel caso esaminato all'interno di questo elaborato, ci si occuperà della parte di Machine Learning, con lo scopo di osservare in che relazione si trovano il numero di bit di precisione assegnato alle variabili dei benchmark, e gli errori di precisione dei risultati prodotti durante esecuzione dei suddetti.
In particolare, nei capitoli successivi, verrà innanzitutto eseguito uno studio volto a scoprire la presenza di relazioni di non-monotonia tra le configurazioni di bit delle variabili e l'errore osservato, vale a dire, si andrà a capire quali tra i benchmark indurranno il modello di ML ad apprendere una funzione non-monotona nei confronti dell'andamento dell'errore; questo è molto importante in quanto la presenza di non-monotonia in-

dica una maggiore difficoltà per l'allenamento del modello predittivo a imparare una funzione rappresentativa della relazione tra numero di bit e l'effetto di questa decisione sul sistema.
Dopo di ciò verrà preso in esame un modello di ML che apprende un modello statistico (le cui predizioni daranno in output parametri di media e varianza), utilizzando target stocastici, tramite il principio della Maximum Likelihood Estimation (che verrà descritto successivamente), e verranno effettuati degli esperimenti preliminari al fine di comprenderne l'effettiva validità.

Verranno successivamente esplorate ulteriori soluzioni che avranno come scopo principale quello di aggiungere delle feature che caratterizzino al meglio gli input set (visto che aggiungere questi ultimi come ulteriori colonne ai dataset potrebbe essere proibitivo a causa dell'alta cardinalità dell'insieme complessivo di valori di input per ogni benchmark):

- Si tenterà innanzitutto di aggiungere delle feature ricavate da grafi che contengono relazioni tra le variabili in forma di disequazione (a tal proposito si veda il lavoro effettuato da F. Livi et al. [7] sulle Graph Convolutional Networks);

- Successivamente si tenterà una strada caratterizzata dall'utilizzo di reti neurali Autoencoder (con lo scopo di apprendere una rappresentazione ridotta dell'insieme degli input set, così da ridurne la cardinalità);

- Si arriverà poi a calcolare un insieme di feature aggiuntive a partire da valori statistici che caratterizzano i vari input set (ad esempio, media, varianza, N-percentile, ecc.).

Infine, nel capitolo conclusivo, verrà fatto un confronto diretto tra i vari metodi impiegati, così da poter valutare tra questi approcci possa risultare una strada valida da perseguire in sviluppi futuri.

2 Machine learning

Con il termine Machine Learning ci si riferisce a quella branca dell'intelligenza artificiale in cui si studiano algoritmi e modelli statistici che i calcolatori possono utilizzare per eseguire svariati compiti senza istruzioni pregresse esplicite, basandosi piuttosto su pattern e tecniche di inferenza. Questo approccio si distacca in modo evidente da quello per cui una computazione debba essere definita in maniera completa ed esplicita a seconda dei dati in ingresso.

In particolare, in base al tipo di algoritmo scelto per la raccolta di pattern e inferenza sui dati da parte della macchina si possono diverse famiglie di sistemi di apprendimento automatico; le più tipiche sono:

- *Approcci supervisionati*: questi algoritmi costruiscono un modello matematico a partire da un insieme di dati che contiene sia gli input, sia gli output relativi desiderati; questo insieme di dati viene detto "training set", in quanto serve ad "allenare" la macchina a riconoscere i pattern che emergono dai dati; questo allenamento è svolto tramite un'ottimizzazione ripetuta di una funzione obiettivo, e, quando eseguito correttamente (e quindi presenta una buona accuratezza di risultati dopo diverse iterazioni), attribuisce alla macchina la capacità di dare output corretti per input non presenti nel training set.
  I principali algoritmi di apprendimento supervisionato sono la classificazione (che comprende il classificatore Naive Bayes, gli alberi decisionali, le reti neurali, le Support Vector Machines, e altri) e la regressione. Gli algoritmi di classificazione vengono utilizzati solitamente quando i possibili valori di output sono limitati ad un certo insieme, mentre i regressori vengono utilizzati quando i valori di output possono assumere un qualunque valore numerico all'interno di un certo range;

- *Approcci non supervisionati*: questi algoritmi utilizzano set di dati che contengono solo valori di input al fine di dedurne una struttura implicita (ad esempio, raggruppamento in cluster). La loro particolarità è quindi quella di imparare pattern da dati che non sono stati etichettati o categorizzati, al fine di ottenere modelli che reagiscano in base alla presenza o assenza di tratti comuni tra il nuovo dato in input e i dati di training.
  L'applicazione più comune di queste tecniche è quella del clustering, ovvero la ricerca di una struttura e pattern ricorrenti all'interno di un vasto insieme di dati; ciò è molto utile quando si vogliono cercare dei trend dall'analisi dei dati, e in particolare casi in cui è impossibile effettuare manualmente il raggruppamento adeguato dei vari dati;

- *Approcci semi-supervisionati*: si tratta di modelli ibridi in cui il set di dati possiede solo alcune delle label (e quindi output attesi) rispetto alla totalità di input riportati; diversi studi hanno mostrato come l'utilizzo di dati non etichettati, in congiunzione con una piccola quantità di dati etichettati, possono produrre un aumento non indifferente di accuratezza di apprendimento.

- *Reinforcement learning*: questo approccio prevede che la macchina sia dotata di sistemi e strumenti in grado di migliorare il proprio apprendimento in base all'ambiente circostante; in particolare, la macchina reagisce in maniera dinamica imparando dagli errori, che infliggono delle "punizioni", e guadagna "ricompense" da azioni corrette, con lo scopo di accumulare e massimizzare le ricompense.

Qualunque famiglia di algoritmi si scelga di utilizzare, la qualità e il corretto utilizzo dei dati di partenza ricopre un'importanza fondamentale. Per esempio, sebbene in un approccio supervisionato il corretto etichettamento dei dati possa apparire come cosa scontata e banale, non è raro dover lavorare con dati che necessitano di pulizia e pre-processamento, a causa di label errate, valori mancanti, e altri scenari che compromettono la qualità del dato.
È altresì importante decidere un'adeguata quantità di dati da utilizzare per l'allenamento del modello: è infatti possibile incorrere nel fenomeno di overfitting, ovvero una situazione in cui il modello si è adattato talmente tanto ai dati di training da non essere in grado di generalizzare e fare predizioni corrette per nuovi dati; il fenomeno contrario è detto underfitting, ovvero quando il modello non è in grado di descrivere adeguatamente la variabilità dei dati.

Al fine di migliorare la qualità dei dati prima dell'allenamento di un modello è quindi importante effettuare del pre-processamento tramite le seguenti attività:

- *Aggregazione*: combinazione di due o più feature (attributi) con lo scopo di ridurne la quantità, o cambiamento della scala dei valori attribuibili a questi (ad esempio, aggregazione di città in regioni, o di giorni in settimane);

- *Campionamento*: riduzione della quantità dei dati da processare utilizzando adeguate tecniche per la scelta di un campione; per la buona riuscita dell'apprendimento del modello è bene che il campione di dati scelto sia rappresentativo del set originale, ovvero che abbia le sue stesse proprietà;

- *Riduzione della dimensionalità*: effettuata quando vi è un numero molto alto di feature nel set di dati, per ridurre la complessità spaziale e temporale per l'utilizzo di questi dati nel modello, e soprattutto con lo scopo di evitare l'overfitting; una delle principali tecniche è la Principal Component Analysis, ovvero una procedura matematica che trasforma un numero di feature correlate in un insieme più piccolo di feature non correlate, rimuovendo quindi ridondanze tra gli attributi; altri metodi per la riduzione della dimensionalità;

- *Creazione di nuove feature*: tecnica che risulta utile quando le feature attualmente disponibili non descrivono al meglio certe caratteristiche dei dati;

- *Trasformazione delle feature dei dati*: insieme di tecniche matematiche che convertono i dati in altri formati o scale, con cui l'algoritmo di apprendimento può lavorare meglio (ad esempio normalizzazione e standardizzazione).

Una volta effettuata la pulizia dei dati, questi vengono suddivisi in tre insiemi:

- *Training set*, ovvero i dati utilizzati durante la fase di addestramento del modello;

- *Validation set*, ovvero i dati utilizzati per la calibrazione dei cosiddetti iperparametri dell'algoritmo di learning; possono essere uti-

lizzati, in particolare, per effettuare delle tecniche di regolarizzazione (che hanno lo scopo di evitare l'overfitting), come l'early stopping, che interrompe l'allenamento quando l'errore sul validation set aumenta rispetto ad un certo valore di soglia;

- *Test set*, ovvero i dati utilizzati per la valutazione delle performance del modello; intuitivamente, serve a valutare come il modello reagisce quando riceve in input dei dati che non ha mai visto durante il training.

Scegliere le giuste proporzioni per questi tre sottoinsiemi di dati è cruciale: con una bassa quantità di dati di training si avrà un'alta variazione dei parametri di modello, mentre con una bassa quantità di dati di training si avrà una stima imprecisa delle performance del modello.
Solitamente è consigliabile dividere i dati tra train+validation e test set utilizzando una proporzione $80 : 20$, per poi ri-dividere con una proporzione simile train e validation set, ottenendo una proporzione totale di $65 : 15 : 20$ ($train : validation : test$).

Come vedremo nei capitoli successivi (in particolare in quello di progetto), gran parte di queste tecniche di processamento di dati appena enunciate verranno adeguatamente applicate per la creazione di modelli di Machine Learning nel contesto del problema esaminato in questo elaborato.

2.1 Regressori

Come anticipato precedentemente, una particolare famiglia di processi di apprendimento supervisionato sono i regressori. Nell'ambito della statistica, l'analisi della regressione è un insieme di processi utilizzati per modellare le relazioni presenti tra una variabile dipendente (detta variabile di output) e una o più variabili indipendenti (dette feature).
Più precisamente, in ambito di Machine Learning, i modelli di regressione approssimano una funzione f che prende valori di input X e dà in output un valore Y, e sono utilizzati per la predizione di valori continui, ovvero valori che rappresentano quantità o grandezze, piuttosto che semplici "etichette" come nel caso dei classificatori.
Vi sono diversi motivi per cui possa essere utile utilizzare la regressione: in generale, questo tipo di analisi permette di comprendere, all'interno di un problema (ad esempio: quali tra diverse dozzine di fattori meteorologico attuali hanno maggiore impatto sul tempo che ci sarà domani? C'è una correlazione tra un grande utilizzo di sigarette e caffeina e la comparsa di disfunzioni cardiovascolari?), quali tra molte variabili indipendenti hanno impatto su un certo risultato atteso.

Vi sono diverse tipologie di regressori:

- *Regressione lineare semplice*: tipologia più comune di regressione; viene predetta una variabile target Y sulla base di una o più variabili di input X, e tra le due parti deve esistere una relazione di linearità, ovvero, la funzione che approssima la correlazione tra Y e X deve essere rappresentabile tramite una retta.
  Tale funzione sarà nella forma:

  $$Y = a + bX$$

  Durante l'allenamento di un modello di regressione, sono proprio i coefficienti a e b ad essere adattati in base ai dati di training; in particolare, l'obiettivo da raggiungere durante l'allenamento è quello di ottenere una retta tale da minimizzare una certa funzione di costo (delle quali si parlerà successivamente nell'elaborato, col nome di funzioni di loss), la quale misura l'errore ottenuto tra il dato reale e quello predetto;

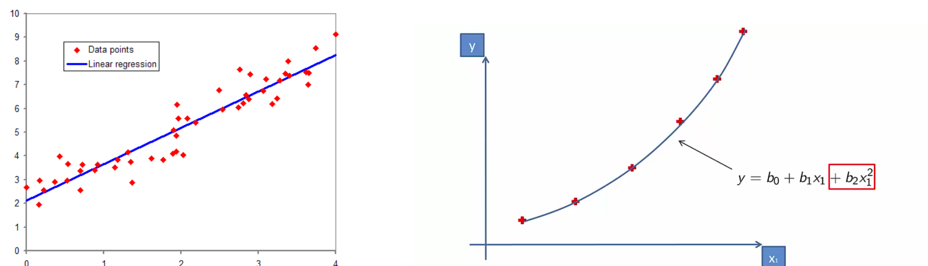


Figura 2.1: Regressione lineare semplice (sinistra), e polinomiale (destra)

- *Regressione polinomiale*: pvengono trasformate le feature originali in feature polinomiali di un determinato grado, per poi applicare la regressione lineare. L'effetto è quello di ottenere un modello lineare rappresentato però da una curva quadratica, la quale permette di approssimare in modo migliore la relazione tra i dati. Si noti però che l'aumento del grado ad un valore molto alto può portare a situazioni di overfitting, in quanto il modello impara così ad approssimare non solo i dati, ma anche il rumore;

- *Regressione con Support Vector*: si identifica un iperpiano con un certo margine massimo, tale che all'interno di esso risieda il massimo numero di punti che identificano i dati. L'obiettivo qui non è quello di minimizzare l'errore, bensì adattare l'errore entro certi limiti, ovvero quelli dell'iperpiano;

- *Regressione con alberi decisionali*: utilizzati sia per la classificazione che per la regressione, gli alberi decisionali sono costruiti partizionando i dati in sottoinsiemi contenenti istanze con valori simili. A differenza del caso della classificazione, in cui viene utilizzato il concetto di information gain, nella regressione viene utilizzata la deviazione standard per calcolare l'omogeneità del campione di dati, dove una deviazione standard uguale a zero indica una somiglianza massima all'interno del campione di dati;

- *Regressione con Random Forest*: metodo facente parte degli approcci ensemble (che costruiscono modelli a partire dall'accorpamento di sottomodelli più piccoli); questo metodo è molto robusto nei confronti dei fenomeni di overfitting (piuttosto comuni applicando metodi che utilizzano gli alberi decisionali), in quanto crea sottoinsiemi casuali di feature e costruisce degli alberi più piccoli con essi.

2.2 Reti neurali

Le reti neurali artificiali (ANNs) sono modelli computazionali composti di neuroni artificiali, ispirati quindi al funzionamento biologico del cervello umano, e sono molto utilizzati nell'ambito dell'intelligenza artificiale, in particolare nel Machine Learning. Questi sistemi imparano ad eseguire dei task a partire da esempi, generalmente senza venire precedentemente programmati con regole relative al problema specifico.

Sono ampiamente utilizzate per diverse applicazioni, ad esempio:

- *Riconoscimento e processamento di immagini e caratteri*: essendo in grado di ricevere input di cardinalità molto alta, le reti neurali riescono a processare immagini con lo scopo di inferire diverse relazioni non-lineari molto complesse e nascoste. In particolare, il riconoscimento di caratteri gioca un ruolo fondamentale nei processi di rilevamento delle frodi e persino di sicurezza nazionale, mentre il riconoscimento di immagini risulta un processo chiave per un vasto insieme di applicazioni che vanno dal riconoscimento facciale nei social media, fino al riconoscimento dei tumori in medicina;

- *Creazione di modelli predittivi*: il concetto di predizione gioca al giorno d'oggi un ruolo fondamentale nella formulazioni di decisioni di business, per esempio nella finanza e nella borsa; il problema però è che spesso questi task sottintendono diversi fattori complessi (alcuni persino sconosciuti). Le reti neurali offrono quindi un'ottima alternativa ai tradizionali modelli predittivi, i quali sono limitati dal non riuscire sempre a tener conto di relazioni complesse e non-lineari tra i dati.

Analizziamo adesso il componente principale e fondamentale per una rete neurale: il neurone artificiale; si tratta di un particolare elemento che ha la capacità di prendere in input dei dati, elaborarli e restituire un risultato in output:

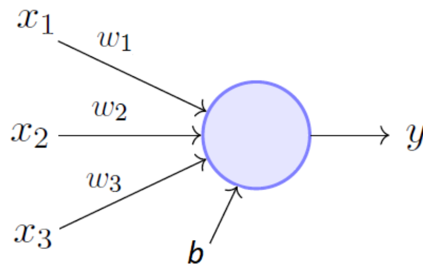


Figura 2.2: Neurone artificiale

Per ogni neurone si hanno $m + 1$ input, da x_1 a x_m più un input di bias b, e altrettante costanti moltiplicative w, dette pesi, uno per ogni connessione.
Nel dettaglio il componente opera come segue, considerando la figura:

1. Ogni dato di input viene moltiplicato per un numero, detto peso, il quale, intuitivamente, aumenta o diminuisce la potenza del segnale in quella particolare connessione:

$$x_1 \rightarrow x_1 * w_1$$

$$x_2 \rightarrow x_2 * w_2$$

$$x_3 \rightarrow x_3 * w_3$$

2. I nuovi input vengono sommati al bias:

$$(x_1 * w_1) + (x_2 * w_2) + (x_3 * w_3) + b$$

3. Infine, il risultato è passato ad una funzione detta "di attivazione", la quale è scelta a seconda dell'obiettivo del modello, in quanto ogni funzione può dare un risultato e una performance diversa; l'obiettivo di queste funzioni di attivazione è quello di determinare l'output del neurone, e, come osserveremo, anche quello della rete stessa. Più in particolare, esse determinano se il neurone deve essere attivato o meno e se deve quindi contribuire o meno all'output completo della rete.

La più semplice funzione di attivazione che ci può essere è detta funzione a step:

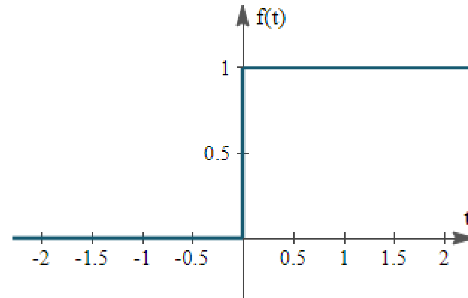


Figura 2.3: Funzione di attivazione a step

Questa funzione restituisce il valore 0 o 1 a seconda che una certa soglia sia superata o meno; questo approccio ha senso, ad esempio, negli scenari di classificazione binaria, dove l'output deve essere un'etichetta precisa.
Ci sono però dei casi in cui ciò non è sufficiente, e si necessita piuttosto di avere la probabilità per cui l'esempio da classificare appartenga ad una certa categoria tra tante; si utilizzano allora altre funzioni di attivazioni: le più utilizzate sono la sigmoide, la Rectified Linear Unit (ReLu) e la softmax, funzioni non lineari che sono necessarie per l'apprendimento di relazioni complesse tra dati complessi, o di grande dimensionalità.

Tramite l'aggregazione di neuroni, interconnessi in molteplici modi, otteniamo le reti neurali, le quali sono organizzate in layer (strati), ognuno dei quali contiene un determinato numero di neuroni; combinando tra loro layer con struttura diversa otteniamo infinite possibili topologie di rete.

I layer di una rete neurale sono sempre organizzati come segue:

- *Input layer*: trato che, ricevendo in input i dati, contiene un numero di neuroni esattamente pari al numero di feature dell'elemento presente nel training set;

- *Hidden layer*: strato intermedio che può comparire anche più volte (nel caso delle Deep Neural Network) all'interno della rete con lo stesso o diverso numero di nodi; quest'ultimo parametro è cruciale, in quanto un numero troppo alto di neuroni può portare a fenomeni di overfitting, mentre un basso numero diminuisce le performance del modello.
  In generale, diversi esperimenti in letteratura [14] hanno mostrato come la dimensione ottimale dell'hidden layer è solitamente compresa tra le dimensioni degli strati di input e di output, ma ovviamente ciò potrebbe non bastare per problemi molto complessi, e quindi potrebbe essere richiesto un numero di neuroni superiore a entrambe le suddette dimensioni;

- *Output layer*: strato finale del modello, che possiede un numero di neuroni solitamente dato dal numero possibili di label a cui associare un dato in input, oppure uguale ad 1 per buona parte delle applicazioni.

I neuroni di un layer si connettono solo ai neuroni dello strato immediatamente precedente e a quelli dello strato immediatamente successivo. Tra due layer è possibile effettuare le connessioni in diversi modi, ad esempio [13]:

- *Fully connected*: quando ogni neurone in un layer è connesso a tutti gli altri neuroni del layer successivo;

- *Pooling*: quando un gruppo di neuroni in uno layer è connesso ad un singolo neurone del layer successivo, riducendo così il numero di neuroni in quello strato.

Reti che utilizzano questi tipi di connessioni, e quindi possiedono una struttura a grafo aciclico, sono dette reti feedforward.
Al contrario, reti che permettono connessioni tra neuroni all'interno dello stesso strato, o connessioni in direzione dei layer precedenti, sono dette ricorrenti, e sono solitamente utilizzate per modellare una memoria di stato.

Una rete neurale, prima di poter essere utilizzata per effettuare delle predizioni, deve prima essere *addestrata* per poter apprendere le relazioni tra i dati di esempio. L'addestramento di una rete neurale avviene in due fasi:

- *Fase forward*: gli esempi del training set (i quali sono suddivisi in "dati", ovvero l'input da cui ci si aspetta una previsione, e in "target"/"label", cioè il risultato numerico di regressione o l'etichetta di classificazione che si deve ottenere), vengono dati in input alla rete, per poi procedere in avanti lungo gli strati successivi; così facendo la rete andrà a calcolare l'output di ogni neurone in ciascun layer, come descritto precedentemente in questo paragrafo durante la descrizione del funzionamento dei neuroni artificiali;

- *Fase backward*: in questa fase ogni strato riceverà un gradiente di errore rispetto ai suoi output ($\frac{\partial L}{\partial out}$), e produrrà un gradiente di errore rispetto ai suoi input ($\frac{\partial L}{\partial in}$); tutto ciò viene eseguito per modificare il valore di ogni peso legato ad ogni connessione, con lo scopo di ridurre man mano l'errore ad ogni iterazione: questo processo viene fatto modificando i pesi seguendo la direzione opposta indicata dal gradiente, ovvero seguendo la decrescita della funzione di costo da minimizzare.

Si noti quindi come addestrare una rete neurale significa andare, man mano, a diminuire il valore della funzione di loss, fino a raggiungere un valore minimo o accettabile; l'argomento verrà approfondito in dettaglio all'interno del paragrafo 2.2.3.

Tenendo conto di questi concetti di base, è possibile creare potenzialmente infinite topologie di rete, ognuna adatta ad applicazioni differenti. Durante le sperimentazioni effettuate all'interno di questo elaborato verranno in particolare utilizzate le Deep Neural Network e gli Autoencoder.

2.2.1 Deep neural networks

Una deep neural network (DNN) è una rete neurale che comprende più di un hidden layer tra input e output layer, ed è in grado di apprendere relazioni lineari e non-lineari che trasformano l'input in output. Ogni layer rappresenta una manipolazione matematica dell'input, e in particolare, man mano che si passa dal primo strato all'ultimo, viene calcolata la probabilità di ogni output. Inoltre, i layer sono detti "densi", in quanto tutti i neuroni di un layer sono collegati a tutti quelli del layer successivo.

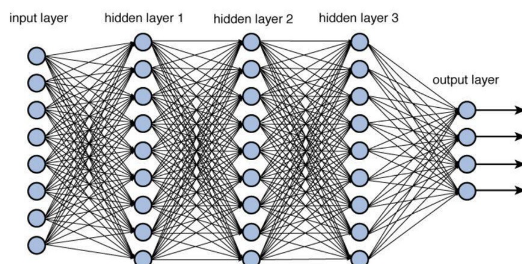


Figura 2.4: Esempio di architettura di DNN

Il numero di neuroni per ogni hidden layer, e il numero stesso di hidden layer sono entrambi parametri fondamentali per il tuning di questo tipo di rete; la decisione di quale debba essere il loro valore, insieme all'assegnamento delle varie funzioni di attivazione per i layer, rappresenta una delle più grandi difficoltà del processo di sviluppo di questo tipo di modelli predittivi, come si vedrà nel dettaglio all'interno del capitolo di progetto di questo elaborato.
Il vantaggio di questa particolare topologia di rete è quello di poter modellare relazioni non lineari molto complesse. Per questo motivo, diverse ricerche hanno evidenziato come le DNN siano in grado di svolgere compiti difficili, come il Natural Language Processing [9], e il riconoscimento vocale [10].

2.2.2 Autoencoder

Gli autoencoder sono tipi particolari di reti neurali utilizzate per ricavare dei modelli di codifica efficiente del dato in maniera non supervisionata. Più precisamente, l'obiettivo di un autoencoder è quello di imparare una rappresentazione per un set di dati, spesso con lo scopo di effettuare una riduzione del numero di feature, così da ridurre il rumore ed evitare overfitting, inserendo questa rappresentazione ridotta delle feature al posto delle feature originali. Il nome di questa particolare topologia di rete è dato dal fatto che, intuitivamente, questa impara a generare una codifica delle feature da cui è in grado di ricostruire le feature originali con un errore minimo.
Questa rete è costituita da due macroblocchi:

- Un *encoder* che mappa l'input ad una rappresentazione codificata;
- Un *decoder* che mappa la codifica ad una ricostruzione dell'input originario.

Se ci si limitasse, per ognuna di queste parti, ad avere degli hidden layer ognuno con ugual numero di neuroni si otterrebbe una rete che copia l'input e lo replica nell'output. Per questo motivo gli autoencoder spesso sono volutamente limitati in un modo tale da forzarli a ricostruire l'input in modo approssimativo, mantenendo solo gli aspetti più rilevanti di esso.

Vediamo nel dettaglio l'architettura di base di un autoencoder, ovvero una rete feedforward non ricorrente (ovvero che non modella una memoria di stato) dove input layer e output layer hanno lo stesso numero di neuroni, mentre gli hidden layer ne hanno un numero minore:

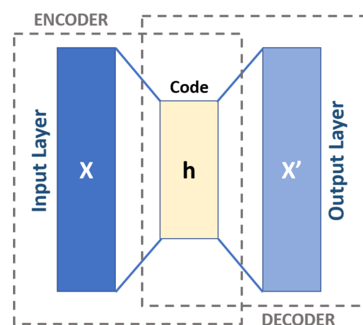


Figura 2.5: Esempio di architettura di Autoencoder

Come evidenziato dalla figura, e come accennato precedentemente, la rete è composta da due macroblocchi, encoder e decoder, che possono essere denominate come transizioni E e D [8]:

$$E : X \to F$$

$$D : F \to X$$

Dove X è il dominio dei dati originali, e F è il dominio delle possibili codifiche ottenibili nel layer centrale (h in figura).
L'obiettivo della rete è quindi quello di cercare i parametri Θ_E, Θ_D rispettivamente per i blocchi encoder e decoder che minimizzino la funzione di costo, nel seguente modo:

$$(\Theta_E, \Theta_D) = \underset{\Theta_E, \Theta_D}{\arg\min} \sum_x ||x - D(E(x))||^2$$

Nel caso più semplice, ovvero quello in cui tra input e output vi è un solo hidden layer, l'encoder prende l'input $\mathbf{x} \in X$ e lo mappa ad $\mathbf{h} \in F$:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Dove $\mathbf{x}$ è detto codice, o rappresentazione latente dell'input; nella formula, σ è una funzione di attivazione, ad esempio una sigmoide o un rettificatore, $\mathbf{W}$ è una matrice di pesi e $\mathbf{b}$ è il vettore di bias, e questi due componenti sono inizializzati in modo casuale per poi essere aggiornati iterativamente durante la fase di allenamento utilizzando la propagazione all'indietro dell'errore (essendo appunto una rete feedforward).

Come si vedrà nel capitolo dedicato alla parte di progetto, questa particolare topologia di rete neurale verrà utilizzata per tentare di effettuare un processo di riduzione della dimensionalità dell'insieme dei vari input set per i benchmark, al fine di poterli includere come feature, in forma compressa, nei dataset per l'allenamento stocastico delle reti.

2.2.3 Funzioni di loss e metriche qualitative

Come osservato precedentemente, l'addestramento di una rete neurale è coadiuvato dalla minimizzazione di una funzione di costo, detta funzione di loss. Questa ci permette di valutare quanto bene l'algoritmo di training sta modellando i dati di input; quando la predizione si discosta troppo dal risultato attuale, la funzione di loss produce un numero alto, ma gradualmente, durante le iterazioni, questa impara a ridurre l'errore fino ad arrivare ad un valore minimo o accettabile.
Esistono svariate funzioni di loss, utilizzate nel Machine Learning, e la scelta di quale sia meglio utilizzare per un determinato problema dipende da fattori come ad esempio l'algoritmo di apprendimento utilizzato, la facilità di calcolo delle derivate per indurre la decrescita del gradiente o la presenza di outlier tra i dati.
Generalmente le funzioni di loss sono suddivise in due grandi famiglie, a seconda del tipo di apprendimento che stiamo effettuando:

- *Funzioni per la regressione*;
- *Funzioni per la classificazione*;

All'interno del presente elaborato ci occuperemo esclusivamente di funzioni per la regressione, ma tra le più tipiche funzioni per la classificazione troviamo la Hinge Loss e la Cross Entropy Loss.

Le funzioni che andremo a utilizzare sono:

- *Root Mean Square Error*:

  $$\text{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2}$$

  Come suggerisce il nome, questa funzione è calcolata come la radice quadrata della media del quadrato della differenza tra predizione e valore reale. Si noti che, a causa dell'utilizzo della potenza di due, le predizioni che sono molto lontane dal valore reale vengono pesantemente penalizzate rispetto a predizioni migliori. Inoltre questa funzione ha delle proprietà matematiche tali da rendere facile il calcolo della sua derivata; infine, la funzione dà sempre un valore non negativo, almeno uguale a zero (cosa che indica una predizione perfetta).

- *Mean Absolute Error*:

$$\text{MAE} = \frac{\sum_{i=1}^{n} |\hat{y}_i - y_i|}{N}$$

  Questa funzione è calcolata come la media della somma del valore assoluto delle differenze tra valori predetti e valori reali. Come la RMSE, anche la MAE riporta la grandezza dell'errore di predizione senza considerarne la direzione. A differenza di essa però, per la MAE si utilizzano degli strumenti più complessi per computarne il gradiente; inoltre, a causa della mancanza dell'elevamento a potenza di due, essa è più tollerante nei confronti degli outlier.

- *Maximum Likelihood Estimation* [15]:

$$\max_{\beta} \left\{ \sum_{i} \ln \{p(y_i|\beta)\} \right\}$$

  In statistica, la MLE è un metodo per stimare il valore dei parametri di un modello statistico, cercando i valori che massimizzino una funzione di verosimiglianza, definita in base alla probabilità che si verifichino certe osservazioni sui dati. Il motivo per ci servirà questa funzione durante le sperimentazioni è che avremo a che fare con problemi di regressione con un target stocastico: a differenza dei problemi di regressione "classici", in cui si utilizzano una label e un target numerico singolo per il learning di ogni esempio, per il problema in esame si è scelto di allenare un modello stocastico, approssimando quindi (tramite la funzione imparata) una distribuzione probabilistica che per sua natura sarà caratterizzata da errore medio e varianza; si noti che il metodo MLE funziona correttamente solo in presenza di una distribuzione Gaussiana dei valori (che nel nostro caso sono gli errori).

Una volta allenato un modello predittivo, è importante capire quanto il suo funzionamento sia corretto, in base a delle metriche di performance. Esistono diverse metriche di valutazione, molte delle quali sono proprio funzioni di loss (la differenza è che non vengono utilizzate durante il training in questo caso, ma a posteriori), tra le quali RMSE e MAE, che verranno utilizzate per valutare le performance dei modelli creati durante le sperimentazioni; altre metriche spesso utilizzati sono la Logarithmic Loss e la curva ROC, nel caso dei classificatori.

2.3 Transfer Learning

Un importante vantaggio dei modelli predittivi creati con le reti neurali è quello di poterle riutilizzare all'interno di problematiche correlate a quelle originarie [11]. La tecnica del Transfer Learning riguarda proprio questa caratteristica dei modelli predittivi, e in particolare, parlando di reti neurali, ciò significa riutilizzare i pesi di uno o più layer provenienti da una rete già addestrata, all'interno di un nuovo modello.
Si tratta quindi, sostanzialmente, di un'ottimizzazione che permette rapidi progressi o addirittura performance migliori durante la creazione del nuovo modello. Si noti che però questo processo funziona soltanto se le feature apprese durante l'allenamento del modello originale sono generali, ovvero se sono adattabili sia al suddetto che al nuovo modello, piuttosto che essere specifiche rispetto uno dei due problemi.

Due approcci comuni per l'utilizzo del Transfer Learning sono:

1. **Approccio a sviluppo del modello**:
    (a) *Selezione del problema iniziale*: si seleziona un problema di modellazione predittiva dove c'è un'abbondanza di dati all'interno dei quali vi è una qualche relazione tra input e output;
    (b) *Sviluppo del modello iniziale*: viene allenato un modello (robusto e con buone performance) per questo problema iniziale, assicurandosi che abbia appreso le relazioni tra le feature e l'output;
    (c) *Riutilizzo del modello*: il suddetto modello iniziale può quindi essere utilizzato come punto di partenza per un modello predittivo per un nuovo problema simile, mantenendone tutte o solo alcune delle sue parti;
    (d) *Tuning del modello*: opzionalmente, il modello ottenuto può essere rifinito o adattato ai dati (coppie input-output) disponibili per il nuovo problema.

2. **Approccio a modello pre-allenato**:
    (a) *Selezione di un modello sorgente*: viene scelto un modello preallenato da un insieme di diversi modelli, messi a disposizione da diverse istituzioni di ricerca;
    (b) *Riutilizzo del modello*: il suddetto viene poi riutilizzato come punto di partenza per un nuovo modello relativo ad un nuovo problema;

(c) *Tuning del modello*: opzionalmente, il modello ottenuto può essere rifinito o adattato ai dati (coppie input-output) disponibili per il nuovo problema.

In letteratura [12] vengono solitamente descritti tre possibili benefici osservabili dal Transfer Learning:

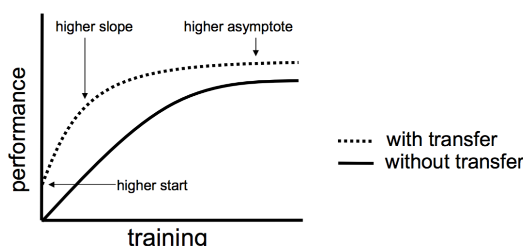


Figura 2.6: Effetti dei processi di Transfer Learning

- *Higher start*: la performance iniziale (ovvero all'inizio del processo di allenamento) sul modello sorgente è più alta di quanto sarebbe altrimenti;
- *Higher slope*: la velocità di miglioramento delle performance durante l'allenamento è maggiore (quindi si ha una curva più ripida) di quanto sarebbe altrimenti;
- *Higher asymptote*: le performance convergono, a fine allenamento, in un punto più alto di quanto sarebbe altrimenti.

Idealmente, un'applicazione ben riuscita del Transfer Learning dovrebbe portare a tutti e tre i benefici.

In definitiva, si tratta di un approccio da provare quando è possibile identificare un problema analogo a quello in esame, il quale dispone di un grande numero di dati, e si hanno le risorse per costruire su di esso un primo modello per poi riutilizzarlo come punto di partenza per proprio problema. Inoltre l'approccio è molto utile in casi in cui il problema in esame non dispone di molti dati per l'allenamento della rete, e quindi il Transfer Learning permette lo sviluppo di modelli predittivi con buone performance per esso che non sarebbero altrimenti possibili.

3 Strumenti utilizzati

Lo scopo di questo capitolo è quello di descrivere l'environment di sviluppo utilizzato per la realizzazione delle soluzioni software adoperate per le sperimentazioni sui modelli predittivi stocastici, con particolare attenzione alle motivazioni che hanno portato alla scelta di ogni tool o framework.

3.1 Environment di base

Durante il processo di sviluppo (e testing) delle applicazioni di supporto alle sperimentazioni, è stata utilizzata una macchina portatile con sistema operativo Windows 10 con il layer di compatibilità Windows Subsystem for Linux [16] (che utilizza Ubuntu 16.04.6 LTS) per permettere l'esecuzione di applicazioni e binari Linux e mantenere coerenza tra ambiente di sviluppo e di sperimentazione; su di essa è stato installato Anaconda [17] (in particolare la versione 4.8.0), ovvero una distribuzione open source dei linguaggi di programmazione Python e R per compiti di data science e machine learning, con una gestione semplificata e automatizzata di pacchetti e librerie software.

Per le sperimentazioni, non essendo la suddetta macchina adatta all'esecuzione di task complessi come l'allenamento di migliaia di reti neurali in un tempo ragionevole, è stata utilizzata una macchina fornita dal Laboratorio di Informatica Avanzata (LIA) del DISI (Dipartimento di Informatica - Scienza e Ingegneria, dell'Università di Bologna), la quale presentava già in partenza un environment di sviluppo basato su CentOS Linux 7.6.1810, e comprensivo sia di Python 3.6.8, sia delle relative librerie di interesse che vedremo nei prossimi paragrafi.

La scelta di utilizzare Python come linguaggio di programmazione è stata guidata dalla presenza, nella community open source, di un grande numero di librerie per manipolazione di dati e creazione di reti neurali. Alcune di queste sono:

- *NumPy* [18]: libreria per la manipolazione di grandi array multidimensionali e matrici, tramite l'ausilio di una collezione di funzioni matematiche di alto livello;

- *pandas* [19]: libreria per la manipolazione efficiente ad alto livello di dati strutturati (nel nostro caso, principalmente tabelle e serie temporali con dati numerici); offre supporto per tutti i classici processi di manipolazione, come le operazioni di join, merge e groupby, oltre a strumenti per preprocessamento e la pulizia dei dati;

- *scikit-learn* [20]: libreria di machine learning che offre supporto per la creazione di task di classificazione, regressione e clustering; utilizza NumPy per effettuare operazioni complesse di algebra lineare, ed è ben integrata con altre librerie, come le altre citate in questa lista. All'interno dell'elaborato, in particolare, sono state utilizzate alcune funzioni di loss messe a disposizione dalla libreria;

- *Matplotlib* [21]: libreria utilizzata per la produzione di rappresentazioni grafiche di dati; permette la creazione di istogrammi, grafici a curve o punti sparsi, grafici tridimensionali, e molto altro;

Le librerie poc'anzi menzionate rappresentano, nell'insieme, i componenti di base utilizzati per la costruzione dei due environment. In realtà, come anticipato, sono state utilizzate ulteriori librerie che hanno permesso lo svolgimento dei compiti di machine learning, e in particolare, la creazione assistita e la gestione dei modelli predittivi basati su reti neurali.

3.2 Tensorflow

TensorFlow [22] è un framework open source per il machine learning. Sviluppato inizialmente dal team Google Brain per utilizzo interno all'azienda, è stato successivamente rilasciato con Licenza Apache 2.0 [23] a fine 2015. Esso comprende un ecosistema flessibile di strumenti, librerie, e risorse provenienti da un'ampia community che permette un processo di sviluppo e deployment facilitato di applicazioni di machine learning; l'obiettivo è ottenuto fornendo delle API compatibili con Python e C++, le quali sono implementate a basso livello con quest'ultimo linguaggio.
Il vantaggio principale di TensorFlow, oltre a quello di essere diventato quasi uno standard de facto per lo sviluppo delle applicazioni di interesse per questo elaborato, è quello di essere altamente scalabile, e di poter essere quindi eseguito pressoché su qualsiasi dispositivo, a partire dalla propria macchina in locale, passando per un cluster nel cloud, fino ad addirittura dispositivi mobili, quali Android, Apple e hardware per operazioni di IoT.

In breve, in TensorFlow, una computazione è descritta da un Data Flow Graph [24], ovvero una struttura a grafo in cui ogni nodo rappresenta un'istanza di operazione matematica (moltiplicazione, addizione, divisione, e via dicendo), e ogni collegamento tra essi rappresenta degli insiemi multidimensionali di dati (detti tensori) sui quali le suddette operazioni vengono svolte; si noti quindi come la creazione di task di machine learning, seguendo il suddetto principio, venga eseguita ad un livello più alto rispetto alla programmazione pura con un qualsiasi linguaggio di programmazione, ma ad un livello ancora molto basso per quanto riguarda la semplicità d'utilizzo.

All'interno della tesi si è scelto di utilizzare in prima battuta TensorFlow in versione 1.15, in quanto versione ampiamente supportata durante i primi tempi di sviluppo dell'elaborato; successivamente, dovendo utilizzare Keras Tuner (descritto nel paragrafo 3.4) si è scelto di passare alla versione 2.0, distribuita dall'ottobre del 2019, e che ha rinnovato il framework in vari modi, basandosi principalmente sui feedback della comunità.

3.3 Keras

Come accennato nel paragrafo precedente, le API fornite da TensorFlow lavorano ancora ad un livello troppo basso per poter essere considerate "developer-friendly", ed è quindi consigliato, almeno per gli sviluppatori non esperti, l'impiego di uno strumento in più.
Keras [25], creato da un ricercatore di Google (Francois Chollet) nel 2015, era originariamente stato sviluppato per le proprie ricerche ed esperimenti; ciononostante, con l'esplosione della popolarità del concetto di deep learning, molti sviluppatori si ritrovarono ad utilizzarlo a causa delle sue API molto user-friendly.
All'epoca non vi erano molte librerie per il deep learning: tra le più popolari si trovavano Torch, Theano e Caffe; il problema di questi framework era la difficoltà di impiegarli per sviluppare applicazioni, in quanto costringevano all'utilizzo di pattern di programmazione di basso livello, che risultavano quindi tediosi e inefficienti.
Keras, d'altra parte, è subito risultato estremamente semplice da utilizzare, permettendo in particolare a ricercatori e sviluppatori di iterare sui propri esperimenti molto più velocemente.

Essendo solo un'API di alto livello, Keras richiede un backend per funzionare, ovvero un componente di fondo che si occupi di eseguire i calcoli di interesse a basso livello; originariamente il backend di default era Theano, anche se allo stesso tempo Google aveva rilasciato TensorFlow, e Keras aveva iniziato lentamente a supportarlo.
A partire dalla versione 1.1.0 di Keras, TensorFlow è ufficialmente il backend di default, essendo quest'ultimo diventato nel tempo il più popolare; la conseguenza di ciò è che, man mano, lo sviluppo dei due software è andato avanti di pari passo, fino ad arrivare all'integrazione completa delle API di Keras all'interno dell'installazione principale di TensorFlow (a partire dalla sua versione 2.0, col package di Python tensorflow.keras).

Fatta questa premessa, lo scopo di Keras, per quanto concerne l'elaborato, è fornire i "mattoncini" di base per la costruzione delle reti neurali; in particolare vengono messi a disposizione dei moduli che vanno a fare parte di un oggetto principale, ovvero il Model, astrazione di un modello di rete neurale, il quale una volta costruito deve essere compilato e poi allenato.

Alcuni dei moduli disponibili sono:

- *Layer*: astrazioni di uno strato di una rete neurale; tra i tipi possibili abbiamo `Input`, `Dense`, `Activation`, `Dropout` e `Flatten` come layer principali, `Conv1D` e `Conv2D` come esempi di strati convoluzionali, `MaxPooling1D` e `MaxPooling2D` come esempi di strati pooling, e via dicendo per tutte le altre tipologie di layer presenti in letteratura;

- *Losses*: modulo che contiene funzioni di loss pronte per l'utilizzo, specificabili come argomento alla funzione che si occupa di compilare il modello;

- *Metrics*: modulo che contiene funzioni per la valutazione delle performance del modello; vengono utilizzate allo stesso modo delle losses, ovvero inserite nel modello come argomento alla funzione `compile`;

- *Optimizers*: modulo che contiene gli ottimizzatori che devono essere utilizzati per l'aggiornamento dei pesi all'interno della rete durante l'allenamento; l'ottimizzatore scelto deve obbligatoriamente essere specificato come argomento all'interno della funzione `compile`;

- *Activation*: modulo che contiene le funzioni di attivazione per i neuroni artificiali; sono in realtà specificabili anche direttamente come argomento all'interno della definizione di un Layer;

- *Callbacks*: insieme di funzioni che possono essere applicate a determinati stadi della procedura di allenamento; solitamente vengono utilizzate per ottenere una visione degli stati interni e delle statistiche del modello; esempi di funzioni di callback sono la `EarlyStopping`, che ferma il training non appena si smette di evidenziare miglioramenti di una quantità monitorata (ad esempio il valore di loss), o `TensorBoard`, descritta nel paragrafo 3.5.

Una volta costruito e compilato il modello, è possibile allenarlo utilizzando il training set, fornendo alla funzione `fit` ulteriori parametri relativi alla modalità di allenamento, detti iperparametri, come ad esempio il numero di epoche di training (ovvero cicli di allenamento sull'intero dataset), la batch size (dimensione dei sottoinsiemi di dataset dati in ingresso alla rete durante l'allenamento, per evitare di fornire l'intero dataset che potrebbe essere troppo grande), la learning rate e il decay (parametri che regolano la velocità di discesa del gradiente della funzione di loss).

3.4 Keras Tuner

Durante lo svolgimento degli esperimenti sulle reti neurali per l'elaborato, è spesso capitato di dover cercare delle configurazioni di iperparametri tali da ottenere il migliore modello possibile per il problema in esame. Nonostante, come vedremo nel capitolo di progetto, spesso sia stato possibile effettuare dei tuning "controllati" (ovvero allenamenti di reti che utilizzano valori di iperparametri compresi in range definiti a mano, per problemi "semplici"), purtroppo per problemi complessi questo non è bastato: occorreva infatti esplorare più a fondo lo spazio di tutti i modelli ottenibili da tutte le combinazioni di iperparametri, spazio che, come immaginabile, può raggiungere dimensioni eccessive tali da non permettere materialmente l'allenamento di tutti quei modelli.

Si è scelto allora di utilizzare, per fasi selezionate di sperimentazioni, Keras Tuner [26], un framework per l'ottimizzazione di iperparametri di recente rilascio (la versione utilizzata, 1.0, è stata resa disponibile il 30 ottobre 2019, e richiede necessariamente TensorFlow 2.0).
Il framework mette a disposizione svariati algoritmi per la ricerca dei migliori iperparametri (e permette la creazione di algoritmi custom), tra cui:

- *Random Search* [29]: algoritmo tramite il quale vengono campionati valori casuali nello spazio degli iperparametri; nonostante abbia un costo computazionale moderato, l'approccio è inefficiente nella ricerca dei migliori candidati, in quanto non vi è nessuna forma di apprendimento dalle combinazioni di iperparametri precedentemente testate;

- *Hyperband* [28]: algoritmo che estende a sua volta una procedura detta SuccessiveHalving, proposta da Jamieson e Talwalkar (Jamieson et al., 2015), e che la utilizza come sottoprocedura. L'algoritmo si basa sulla seguente idea: quando un certo set di iperparametri genera, con poche epochs, un modello con basse performance, non ha più senso continuare ulteriormente ad allenarlo con quegli iperparametri; operativamente, vengono allenati diversi modelli per poche epochs, dopodiché vengono prelevati i modelli con le migliori performance, e il loro allenamento viene continuato per un altro numero di epochs (sempre moderato); questo ciclo viene continuato finché non si arriva ad una convergenza, e quindi, all'ottenimento del modello migliore;

- *Bayesian Optimization* [27]: come per la Random Search, anche in questo algoritmo l'ottimizzatore campiona un sottoinsieme di combinazioni di iperparametri; la differenza è che qui lo scopo non è testare le combinazioni senza mantenere informazioni sulla performance ottenuta, bensì si vuole apprendere una rappresentazione probabilistica della performance stessa; in particolare, all'inizio vengono scelte e testate alcune combinazioni di iperparametri, i cui risultati sono poi utilizzati per produrre un primo modello di una funzione obiettivo. Ottenuto quest'ultimo, si sceglie la combinazione di iperparametri successiva tramite uno dei seguenti modi:

  - Campionando "vicino" agli oggetti dello spazio che fanno osservare buone performance, in modo da cercare di farla crescere ulteriormente;
  - Campionando un ulteriore sottoinsieme dello spazio degli iperparametri non ancora esplorato, per il quale non vi sono informazioni e tramite cui si potrebbe ottenere un nuovo valore massimo di performance.

  Una volta testata la combinazione scelta, il modello probabilistico viene aggiornato, e il ciclo ricomincia; in sostanza la Bayesian Optimization utilizza le combinazioni già testate per poter campionare quelle successive.
  Nonostante sia migliore delle due tecniche menzionate precedentemente, è leggermente più costosa a livello computazionale, in quanto bisogna tener conto dell'aggiornamento del modello probabilistico ad ogni iterazione, prima di campionare una nuova combinazione; questo costo in realtà è alto solo all'inizio, perché dopo molte iterazioni, il modello possiede una confidenza della funzione obiettivo tale da andare a campionare nuove combinazioni in maniera efficiente.

Il processo di tuning con questo framework parte innanzitutto dalla definizione di un modello di rete neurale (comprensivo di istruzione di compilazione) con Keras, incapsulato in una funzione `build_model(hp)`, dove l'argomento hp rappresenta una lista di iperparametri, definiti all'interno di opportuni oggetti di Keras Tuner (e dovranno essere utilizzati dal modello in questa forma, piuttosto che utilizzare tipi primitivi di Python).

Fatto ciò, è necessario istanziare un tuner come istanza di un oggetto `RandomSearch`, `Hyperband`, o `BayesianOptimization`, ai cui costruttori vanno dati come argomenti la funzione `build_model`, l'obiettivo (ovvero la metrica da minimizzare o massimizzare), il numero massimo di configurazioni da testare, e ulteriori parametri.
Su questo oggetto tuner è possibile quindi effettuare un'operazione search, la quale ha esattamente gli stessi argomenti della funzione `fit` di Keras (sulla quale è quindi possibile ad esempio specificare training set, validation set, numero di epoch e callback); infine, tramite la funzione `get_best_models` è possibile ottenere i migliori modelli ottenuti, e con `get_best_hyperparameters` invece è possibile visualizzare le migliori combinazione di iperparametri che li hanno generati.

Come osservabile dalla suddetta descrizione, questo tool è utile per il tuning di iperparametri come il numero di hidden layer e i relativi neuroni, la scelta della funzione di attivazione da utilizzare, la scelta dei parametri di learning rate e decay, ma non dei parametri definibili a livello di funzione `fit` (o search nel caso del tuner), come ad esempio il numero di epochs e il batch size.
A causa di questa limitazione (e altre, relative a bug dovuti al fatto che è un tool nuovo, e quindi ancora acerbo), si è scelto di utilizzare l'ottimizzazione di Keras Tuner (sempre in modalità Bayesian) solo per parte della fase di tuning, servendosi quindi di una parte di tuning manuale ove necessario.

3.5 TensorBoard

TensorBoard [30] è un tool di visualizzazione (cui corrisponde una funzione di callback in Keras) compreso nella suite di TensorFlow. Lo scopo della callback è quello di scrivere un log dettagliato del processo di training e validazione della rete neurale, che viene poi interpretato dal tool principale (eseguibile da linea di comando e raggiungibile graficamente tramite una porta di rete dal browser) per creare delle rappresentazioni grafiche di tali metriche.

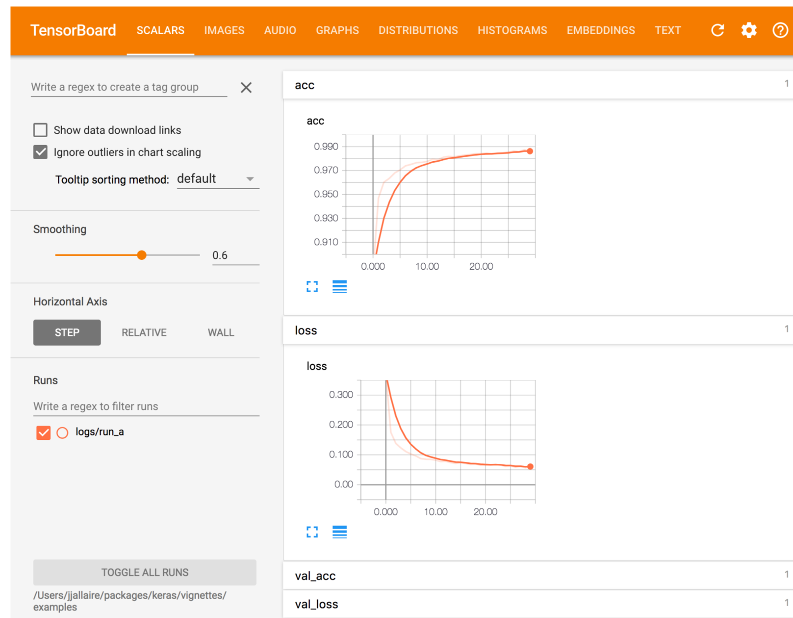


Figura 3.1: Interfaccia utente di TensorBoard, con esempi di grafici

In particolare, TensorBoard permette di tracciare e visualizzare metriche come loss e accuratezza, visualizzare il grafo del modello, osservare gli istogrammi del cambiamento dei pesi della rete nel tempo.
Per questa ragione, il tool è risultato essere di vitale importanza durante gli esperimenti dell'elaborato; infatti, è stato così possibile capire quali configurazioni di iperparametri, tra le migliaia di modelli allenati per ogni tipologia di esperimento, fossero le migliori.

4 Progetto

L'obiettivo di questo capitolo è descrivere nel dettaglio gli applicativi di supporto creati e utilizzati per effettuare gli esperimenti volti alla comprensione della questione: come cambia l'errore ottenuto da un benchmark che prende in ingresso un certo input set, data una una configurazione di bit di precisione per le variabili di input?
La questione, nel lavoro di tesi di F. Livi [7], era stata esplorata da un determinato punto di vista: si voleva evidenziare la correlazione presente tra configurazioni di bit ed errore generato, considerando un input set sempre fisso, utilizzando quindi dei modelli predittivi allenati ad imparare valori di errori precisi per ogni configurazione.

In questo elaborato, il problema viene affrontato da un punto di vista non ancora esplorato in letteratura: nello specifico, all'interno dello studio della correlazione configurazione-errore viene aggiunta un'ulteriore variabile, gli input set, rendendo il problema più generale e quindi più complesso. Verranno quindi studiati ed effettuati esperimenti mirati ad una rappresentazione efficiente di questa informazione per la sua codifica in un modello detto "stocastico"; questa nomenclatura è dovuta al fatto che, a causa delle suddette considerazioni sulla generalizzazione del problema, la relazione "imparata" dal modello dovrà approssimare una distribuzione statistica degli errori, fornendoci quindi in output valore medio e varianza dell'errore per ogni configurazione di bit data in input.
Va altresì tenuto presente che tutti gli esperimenti eseguiti durante la stesura dell'elaborato, per motivi di replicabilità, sono stati effettuati utilizzando un insieme di dataset appositamente generati all'interno del LIA; ognuno di questi set comprende, per ogni benchmark, 5000 esempi per il training, 500 esempi per la validation, e 1000 esempi per il test; si è scelto, nello specifico, di scegliere un set per ogni benchmark e mantenerlo fisso durante tutta la durata degli esperimenti.

4.1 Studio delle relazioni di dominanza tra configurazioni di bit

Il primo passo da compiere, ancor prima di selezionare gli strumenti teorici da utilizzare per la costruzione dei modelli stocastici, è stato quello di effettuare un'analisi sul dataset di ogni benchmark; ciò che si vuole verificare, è l'eventuale esistenza di comportamenti non monotonici dell'andamento dell'errore in una direzione, rispetto all'andamento quantitativo dei bit utilizzati nelle configurazioni nella direzione opposta.
Il motivo per l'esecuzione di questa analisi è da ricondurre al fatto che i comportamenti non monotonici rappresentano un'ulteriore sfida per l'allenamento di reti neurali, e in particolare per l'approssimazione della funzione che cercano di modellare. In sostanza, se si dovessero evidenziare casi di non monotonia, le nostre reti neurali avranno più difficoltà a modellare il comportamento desiderato.

In particolare, preso un determinato benchmark, la questione da evidenziare è: diminuendo da una configurazione all'altra il numero di bit di precisione utilizzati per ogni variabile, l'errore aumenta sempre, oppure si possono evidenziare dei casi di non monotonia, ovvero la presenza contemporanea all'interno del dataset sia di crescita degli errori, sia di decrescita, per configurazioni di bit con valori man mano decrescenti?
Si è scelto quindi di scrivere uno script in Python che, dato un dataset in forma

var_0	...	var_N	err_ds_0	...	err_ds_M	err_avg
12	...	3	1.785	...	0.326	2.542
54	...	63	1.201	...	1000.0	255.743
36	...	15	0.149	...	160.44	56.351

prendesse ogni configurazione e la confrontasse con tutte le altre, alla ricerca di relazioni "di dominanza" (ovvero, una configurazione $X = \{x_0, x_1, .., x_N\}$ è detta dominante su una configurazione $Y = \{y_0, y_1, ..., y_N\}$ quando ogni singolo valore in X è maggiore del rispettivo valore in Y).

Una volta trovate tali relazioni (e quindi coppie di configurazioni per cui vale quella proprietà), lo script deve, per ognuna delle coppie, verificare che dalla configurazione dominante a quella dominata vi sia una crescita dell'errore.

In caso di esito negativo, abbiamo a che fare con un caso di non monotonia, e quindi lo script deve aumentare un contatore (relativo appunto al numero di coppie non monotone).
Si noti che, disponendo di più colonne di errori (ognuno relativo ad un input set diverso) lo script dispone di un array di contatori per le situazioni di non monotonia di cardinalità analoga. Terminato tale processo su tutto il dataset, lo script calcola la percentuale di casi di non monotonia per ogni colonna d'errore, relativamente a tutte le coppie di configurazioni generabili nel dataset.

È riportato, nel seguito, un estratto di codice dello script che concretizza quanto detto appena detto a parole:

```
a = None
b = None
length = 0 # counter of already filled rows
for i in range(0, df_length):
    a = df.values[i, 0:var_num]
    for j in range(i+1, df_length):
        b = df.values[j, 0:var_num]
        print("Comparing configuration at row", i, "with configuration at row", j)
        if np.all(a >= b):
            print("--- FOUND DOMINANT CONFIGURATION A > B ---")
            dom_table_df.values[length] = [i, j, 'A']
        elif np.all(b >= a):
            print("--- FOUND DOMINANT CONFIGURATION B > A ---")
            dom_table_df.values[length] = [i, j, 'B']
        else:
            dom_table_df.values[length] = [i, j, 'none']
        length += 1
dom_table_df = dom_table_df[dom_table_df.DOMINANT_SET != 'none']
dom_table_df.reset_index(drop=True, inplace=True)
non_monotony_score = np.zeros(err_IS_num+1, dtype=int)
dom_table_length = dom_table_df.shape[0]

for i in range(0, dom_table_length):
    print("dom_table iteration", i, "of", dom_table_length-1)
    if dom_table_df.loc[i].values[2] == 'A':
        check_monotony(dom_table_df.loc[i].values[0],
            dom_table_df.loc[i].values[1], df, non_monotony_score, err_IS_num,
     var_num)
    elif dom_table_df.loc[i].values[2] == 'B':
        check_monotony(dom_table_df.loc[i].values[1],
            dom_table_df.loc[i].values[0], df, non_monotony_score, err_IS_num,
     var_num)
    else:
        print("ERROR: row", i, "of dom_table_df has an invalid DOMINANT_SET value"
     )

non_monotony_percentage = (non_monotony_score/len(dom_table_df.index)) * 100
print("The non monotony percentage for each err_IS is", non_monotony_percentage)
```

Dove la funzione `check_monotony` è implementata nel seguente modo:

```
def check_monotony(dominant, dominated, df, non_monotony_score, err_IS_num,
 var_num):
    for i in range(0, err_IS_num+1):
        if (df.iloc[dominant].values[var_num+i] > df.iloc[dominated].values[
 var_num+i]):
            if i < err_IS_num:
                print("POINT ADDED FOR err_ds_",i)
            else:
                print("POINT ADDED FOR err_avg")
            non_monotony_score[i] += 1
```

Implementato lo script, sono stati eseguiti degli esperimenti sui dataset relativi ai benchmark *BlackScholes*, *convolution*, *correlation*, *dwt*, *Jacobi* e *saxpy*; sono stati osservati i seguenti risultati (in percentuali):

	BlackScholes	convolution	correlation	dwt	Jacobi	saxpy
Input Set 0	6.842	0.421	6.352	0.163	0.000	0.437
Input Set 1	5.664	0.410	5.023	0.283	0.000	0.437
Input Set 2	5.596	0.405	5.686	0.314	0.000	0.437
Input Set 3	5.165	0.440	6.562	0.144	0.000	0.437
Input Set 4	7.046	0.435	7.194	0.282	0.000	0.437
Input Set 5	5.143	0.451	5.301	0.206	0.000	0.437
Input Set 6	4.961	0.460	4.685	0.138	0.000	0.437
Input Set 7	5.777	0.448	6.682	0.184	0.000	0.437
Input Set 8	5.188	0.476	4.719	0.165	0.000	0.437
Input Set 9	6.208	0.449	5.360	0.161	0.000	0.437
Input Set 10	5.800	0.486	5.818	0.217	0.000	0.437
Input Set 11	6.978	0.407	4.459	0.222	0.000	0.437
Input Set 12	6.004	0.456	4.564	0.143	0.000	0.437
Input Set 13	5.913	0.456	7.410	0.198	0.000	0.437
Input Set 14	6.615	0.414	5.951	0.154	0.000	0.437
Input Set 15	5.415	0.492	4.460	0.268	0.000	0.437
Input Set 16	5.822	0.460	5.605	0.192	0.000	0.437
Input Set 17	6.072	0.418	4.784	0.177	0.000	0.437
Input Set 18	5.301	0.495	4.785	0.099	0.000	0.437
Input Set 19	5.029	0.439	9.766	0.294	0.000	0.437
Input Set 20	6.570	0.440	5.562	0.192	0.000	0.437
Input Set 21	5.369	0.460	6.571	0.260	0.000	0.437
Input Set 22	6.140	0.466	5.234	0.179	0.000	0.437
Input Set 23	6.751	0.407	5.311	0.156	0.000	0.437
Input Set 24	5.868	0.441	8.161	0.185	0.000	0.437
Input Set 25	6.932	0.434	5.054	3.362	0.000	0.437
Input Set 26	5.324	0.490	5.367	0.286	0.000	0.437
Input Set 27	6.253	0.419	4.919	0.213	0.000	0.437
Input Set 28	6.479	0.458	8.847	0.202	0.000	0.437
Input Set 29	5.981	0.451	5.901	0.152	0.000	0.437
Media Input Set	5.913	0.539	9.447	0.286	0.000	0.437

Come si può osservare, la percentuale di casi di non monotonia dipende da caso a caso, variando da valori molto vicini all'1%, fino ad un massimo del quasi 10% di occorrenze, nel caso del benchmark correlation. Questi risultati, certamente non trascurabili per la maggior parte dei benchmark testati, ci fanno intuire che i nostri modelli predittivi stocastici dovranno modellare una funzione più complessa del previsto, e che è probabile che i risultati che osserveremo nei capitoli successivi possano essere "penalizzati" da questa caratteristica del dominio del problema.

4.2 Creazione di un modello stocastico di base

Eseguita la precedente analisi, si entra nel fulcro del problema: la creazione di un regressore che metta in una relazione matematica la scelta di configurazioni di bit e gli errori associati generati. Come menzionato durante l'introduzione del capitolo, il problema (in una versione più semplice) era stato affrontato all'interno del lavoro di tesi F. Livi et al. (2019); in particolare, fissato un certo Input Set, si era allenato dapprima un regressore lineare modellato con *scikit-learn*, e poi un regressore modellato con *Keras* come DNN, utilizzando la funzione di loss Mean Squared Error (corrispondente al quadrato della RMSE analizzata nel capitolo 2.2.3). I dataset per l'allenamento e per il test di entrambi i erano così strutturati:

Dati (configurazioni di bit per ogni variabile)			Target (errori associati)
conf_0_var_0	conf_0_var_...	conf_0_var_M	err_conf_0
conf_..._var_0	conf_..._var_...	conf_..._var_M	err_conf_...
conf_N_var_0	conf_N_var_...	conf_0_var_M	err_conf_N

In sostanza, i modelli così generati (uno per ogni Input Set, per ogni benchmark) imparano l'andamento dell'errore unicamente sulla base della configurazione di bit relativa, ed effettuano quindi la regressione di un punto singolo.

L'approccio che si vuole seguire in questa tesi è differente, e in quanto tale, cerca di osservare il problema da un lato ancora inesplorato: come si può generalizzare il modello per evitare di dover allenare tante istanze di esso quanti sono gli Input Set?

4.2.1 Un nuovo modello con Maximum Likelihood Estimation

Prima di affrontare il problema, si fa un importante assunto: si suppone che per ogni benchmark, al variare degli Input Set, gli errori generati utilizzando diverse configurazioni di bit seguano una distribuzione normale (Gaussiana), la quale sarà quindi pienamente descritta da una media μ e da una varianza σ^2.

Passiamo adesso alla questione chiave: come generalizzare il modello? Osservando il modello di F. Livi, si nota che (per scelta progettuale), come già detto, non viene tenuto conto degli Input Set, mentre in realtà potrebbe essere utile poter codificare il loro contributo all'interno dei modelli predittivi. Potremmo quindi pensare di utilizzare un'unica struttura dati contenente tutte le informazioni sugli errori relativi ai vari input set (nel nostro caso 30), e partire quindi da dataset del tipo:

Dati (configurazioni di bit per ogni variabile)			Target (errori associati per ogni input set)		
conf_0_var_0	conf_0_var_...	conf_0_var_M	err_conf_0_IS_0	err_conf_0_IS_...	err_conf_0_IS_29
conf_..._var_0	conf_..._var_...	conf_..._var_M	err_conf_..._IS_0	err_conf_..._IS_...	err_conf_..._IS_29
conf_N_var_0	conf_N_var_...	conf_0_var_M	err_conf_N_IS_0	err_conf_N_IS_...	err_conf_N_IS_29

Come si evince dalla nuova struttura (adottata nei dataset utilizzati per la replicabilità degli esperimenti), utilizzare una regressione di un singolo punto non è più sufficiente, visto che adesso abbiamo un target con più valori per un singolo esempio; in particolare, potrebbe essere comodo avere un modello che impari, non a fare la regressione di un punto singolo, bensì ad approssimare una distribuzione statistica, e arrivare quindi al concetto di modello stocastico.
Per fare ciò, vista l'assunzione fatta sulla distribuzione degli errori, si è scelto di allenare un regressore che utilizzi la Maximum Likelihood Estimation come funzione di loss; in realtà, per utilizzare tale metodo come funzione da minimizzare, è utile riformularla come minimizzazione della *negative logarithmic likelihood*, di cui è riportata di seguito l'implementazione come funzione di loss di Keras in Python:

```
def neg_log_likelihood_loss(y_true, y_pred):
    n_dims = int(int(y_pred.shape[1])/2)
    mu = y_pred[:, 0:n_dims]
    logsigma = y_pred[:, n_dims:]

    mse = -0.5*K.sum(K.square((y_true-mu)/K.exp(logsigma)),axis=1)
    sigma_trace = -K.sum(logsigma, axis=1)
    log2pi = -0.5*n_dims*np.log(2*np.pi)
    log_likelihood = mse+sigma_trace+log2pi
    return K.mean(-log_likelihood)
```

Come si evince dall'implementazione, questa funzione di loss (per sua natura) manipola al suo interno la media μ dei target e la loro varianza σ^2, facendo sì che il modello apprenda, di fatto, una distribuzione caratterizzata da quei parametri. Possiamo allora pensare di sviluppare una rete neurale che, utilizzando la MLE come funzione di loss, riesca a dare in output i valori di media e varianza della distribuzione appresa durante il training.
Per fare ciò, si è partiti da una topologia di rete molto semplice, implementata in Keras, comprendente un layer di Input, uno Hidden, e un Output con due neuroni (uno per la media e uno per la varianza):

```
# Definizione modello sequenziale
model = Sequential()

# Input Layer subito seguito da un Hidden Layer
# con numero di neuroni pari a tre volte il numero di feature
model.add(Dense(n_features * 3, activation='relu',
   activity_regularizer=regularizers.l1(1e-5),
   input_shape=input_shape))

# Output Layer con due neuroni
model.add(Dense(1 + 1, activation='linear'))

def neg_log_likelihood_loss(y_true, y_pred):
    n_dims = int(int(y_pred.shape[1])/2)
    mu = y_pred[:, 0:n_dims]
    logsigma = y_pred[:, n_dims:]

    mse = -0.5*K.sum(K.square((y_true-mu)/K.exp(logsigma)),axis=1)
    sigma_trace = -K.sum(logsigma, axis=1)
    log2pi = -0.5*n_dims*np.log(2*np.pi)
    log_likelihood = mse+sigma_trace+log2pi
    return K.mean(-log_likelihood)

# Definizione callback
early_stopping = EarlyStopping(
        monitor='val_loss', patience=10, min_delta=1e-5)
reduce_lr = ReduceLROnPlateau(
        monitor='val_loss', patience=5, min_lr=1e-5, factor=0.2)

# Compilazione modello con ottimizzatore Adam
adam = optimizers.Adam(lr=0.001, decay=0.005)
model.compile(optimizer=adam, loss=neg_log_likelihood_loss)

# Training modello
history = model.fit(x_train, y_train, epochs=100, batch_size=128, shuffle=True,
        validation_data=(x_val, y_val), callbacks=[early_stopping, reduce_lr])

# Test performance del modello
predicted = model.predict(x_test)
actual = y_test
MAE_test, RMSE_test = evaluate_predictions(predicted, actual, scaler, True)
```

Prima di poter utilizzare questo modello, è stato necessario trovare la combinazione ottimale, per ogni benchmark, degli iperparametri (quali: numero hidden layer, numero neuroni per hidden layer, numero epoch, learning rate e decay di Adam, funzione di attivazione e batch size), e si è quindi effettuata una fase di tuning, utilizzando (come anticipato) *Keras Tuner* per testare le varie combinazioni.

4.2.2 Tuning del modello

Si è scelto di utilizzare *Keras Tuner*, e in particolare la *Bayesian Optimization*.
Come anticipato nel capitolo 3.4, per effettuare il tuning con *Keras Tuner* è necessario un refactoring del codice; è stata quindi modificata la parte di codice della definizione di modello per essere inclusa all'interno di una funzione `build_model(hp)` (con relativa sostituzione dei parametri da tarare), per permettere la generazione dei file di log (necessari per i grafici) relativi ad ogni singolo modello allenato, ed è stato aggiunto tutto il codice di contorno per *Keras Tuner*.
Inoltre, per sopperire alla mancanza di un modo diretto per ottimizzare i parametri di epoch e batch size, è stato definito un tuner custom che incapsulasse al suo interno l'oggetto `kt.tuners.BayesianOptimization` e che fornisse la possibilità di effettuare il tuning di tali parametri; vengono di seguito riportate le parti importanti di questo refactoring:

```
def build_model(hp):
    pred_model = Sequential()
    # ...
    dense_layers = hp.Int('dense_layers', 1, 10)
    for n in range(dense_layers):
        pred_model.add(Dense(n_features_glob * hp.Int('node_multiplier_'+str(n),
     min_value=1, max_value=15),
        activation=hp.Choice('act_' + str(n), ['tanh', 'relu', 'sigmoid', '
     hard_sigmoid', 'exponential', 'linear', 'softmax', 'softplus']),
        activity_regularizer=regularizers.l1(1e-5)))

    # ...

    l_r = hp.Choice('l_r', values=[1e-2, 1e-3, 1e-4, 1e-5])
    dec = hp.Choice('dec', values=[5*(1e-2), 5*(1e-3), 5*(1e-4), (1e-5)])

    # ...

    adam = optimizers.Adam(lr=l_r, decay=dec)
    pred_model.compile(optimizer=adam, loss=neg_log_likelihood_loss)

    return pred_model

class MyTuner(kt.tuners.BayesianOptimization):
  def run_trial(self, trial, *args, **kwargs):
    kwargs['batch_size'] = trial.hyperparameters.Int('batch_size', 32, 256, step
     =32)
    kwargs['epochs'] = trial.hyperparameters.Int('epochs', 15, 75, step=15)
    super(MyTuner, self).run_trial(trial, *args, **kwargs)

def main(argv):
    # ...

    tuner = MyTuner(build_model, objective='val_loss', max_trials=256,
            executions_per_trial=1, directory='tunerlogs', project_name=benchmark)
    tuner.search(x_train, y_train, shuffle=True, validation_data=(x_val, y_val),
            callbacks=[early_stopping, reduce_lr])
    models = tuner.get_best_models(num_models=10)
    tuner.results_summary()
```

4.2.3 Esperimenti e risultati

Eseguito (per i benchmark *BlackScholes*, *correlation*, *Jacobi*, *dwt*, e *FWT*) il processo di tuning sull'environment della macchina di laboratorio, si sono ottenute le seguenti combinazioni di iperparametri:

- *BlackScholes*:
  - Hidden Layer: 3
    1. Numero neuroni: $7 * num_features$;
       Funzione di attivazione: `softplus`
    2. Numero neuroni: $2 * num_features$;
       Funzione di attivazione: `softmax`
    3. Numero neuroni: $11 * num_features$;
       Funzione di attivazione: `linear`
  - Batch size: 64
  - Epoch: 45
  - Learning rate e decay: 0.01, 0.0005
- *correlation*:
  - Hidden Layer: 9
    1. Numero neuroni: $15 * num_features$;
       Funzione di attivazione: `linear`
    2. Numero neuroni: $2 * num_features$;
       Funzione di attivazione: `tanh`
    3. Numero neuroni: $3 * num_features$;
       Funzione di attivazione: `exponential`
    4. Numero neuroni: $15 * num_features$;
       Funzione di attivazione: `tanh`
    5. Numero neuroni: $5 * num_features$;
       Funzione di attivazione: `hard_sigmoid`
    6. Numero neuroni: $13 * num_features$;
       Funzione di attivazione: `relu`
    7. Numero neuroni: $6 * num_features$;
       Funzione di attivazione: `tanh`
    8. Numero neuroni: $8 * num_features$;
       Funzione di attivazione: `exponential`
    9. Numero neuroni: $8 * num_features$;
       Funzione di attivazione: `softplus`
  - Batch size: 64
  - Epoch: 30
  - Learning rate e decay: 0.01, 0.0005
- *FWT*:
  - Hidden Layer: 5
    1. Numero neuroni: $10 * num_features$;
       Funzione di attivazione: `relu`
    2. Numero neuroni: $13 * num_features$;
       Funzione di attivazione: `linear`
    3. Numero neuroni: $3 * num_features$;
       Funzione di attivazione: `tanh`
    4. Numero neuroni: $3 * num_features$;
       Funzione di attivazione: `softplus`
    5. Numero neuroni: $3 * num_features$;
       Funzione di attivazione: `softplus`
  - Batch size: 64
  - Epoch: 45
  - Learning rate e decay: 0.01, 0.005

- *dwt*:
  - Hidden Layer: 8
    1. Numero neuroni: $7 * num_features$;
       Funzione di attivazione: `linear`
    2. Numero neuroni: $15 * num_features$;
       Funzione di attivazione: `relu`
    3. Numero neuroni: $12 * num_features$;
       Funzione di attivazione: `relu`
    4. Numero neuroni: $12 * num_features$;
       Funzione di attivazione: `softmax`
    5. Numero neuroni: $10 * num_features$;
       Funzione di attivazione: `exponential`
    6. Numero neuroni: $8 * num_features$;
       Funzione di attivazione: `linear`
    7. Numero neuroni: $14 * num_features$;
       Funzione di attivazione: `sigmoid`
    8. Numero neuroni: $11 * num_features$;
       Funzione di attivazione: `tanh`
  - Batch size: 160
  - Epoch: 45
  - Learning rate e decay: 0.001, 0.00001
- *Jacobi*:
  - Hidden Layer: 2
    1. Numero neuroni: $13 * num_features$;
       Funzione di attivazione: `softplus`
    2. Numero neuroni: $13 * num_features$;
       Funzione di attivazione: `sigmoid`
  - Batch size: 160
  - Epoch: 60
  - Learning rate e decay: 0.00001, 0.00001

Ottenuti i modelli, sono stati eseguiti dei test (utilizzando i test set) per valutare le loro performance, e in particolare osservare l'andamento di media e varianza.

Purtroppo i risultati non sono stati quelli sperati: la predizione della media dell'errore avviene correttamente per alcuni dei benchmark (quali *correlation*, *dwt*) e con qualche incertezza per i restanti, mentre la varianza assume un comportamento imprevedibile e di conseguenza è praticamente sempre errata.
Utilizzando delle metriche qualitative per giudicare i modelli, è possibile osservare che mentre da *correlation* e *dwt* risultano valori di MAE e RMSE rispettivamente nell'ordine di 1.70-2.20 e 2.40-3.00, i valori delle metriche dei restanti benchmark li superano con ordini di rispettivamente 6.00 e 7.00.

4.3 Miglioramento del modello attuale a partire da considerazioni teoriche

Sebbene si sia ottenuta una predizione del valore medio dell'errore con una qualità decente, il modello appena creato ci ha comunque fornito risultati di qualità altalenante. Si può fare di meglio?
Per rispondere a questa domanda ci si è interrogati sui motivi per cui il modello attuale non riesce a garantire performance ottimali, e in particolare si sono ipotizzate due cause:

1. L'assunzione iniziale sulla distribuzione potrebbe in realtà essere errata: si tenga a mente infatti che si è scelto di utilizzare il metodo Maximum Likelihood Estimation, che presuppone, per il corretto funzionamento, valori distribuiti secondo una gaussiana;

2. I dataset attuali, data la loro struttura, potrebbero non fornire al modello la conoscenza completa del dominio, e quindi delle relazioni tra configurazioni e distribuzione dell'errore.

Valutiamo una ad una le ipotesi, partendo con la prima nel sottoparagrafo successivo.

4.3.1 Considerazioni sulla distribuzione degli errori

Abbiamo inizialmente assunto che gli errori generati per i vari input set ad ogni configurazione possiedano una distribuzione normale, e ciò è cruciale che avvenga affinché il metodo MLE funzioni correttamente.
Il fatto che il modello attuale non operi come ci si aspettava, fa pensare che questo assunto possa in realtà non essere valido. Per essere sicuri di ciò, si è andati a rappresentare degli istogrammi, per ogni benchmark e per varie configurazioni, che rappresenta la distribuzione degli errori in base a quanti Input Set riscontrano uno stesso valore di errore.
Sono riportati di seguito degli esempi di grafici ottenuti:

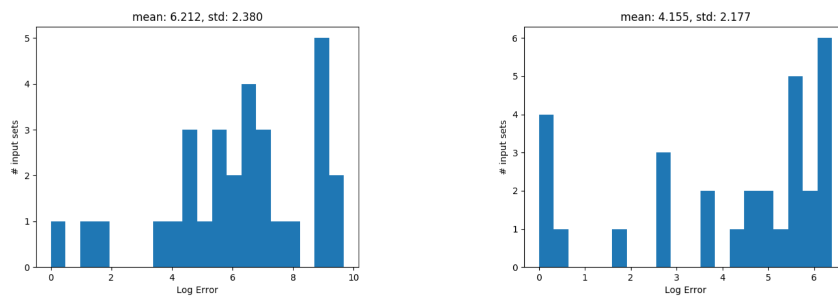
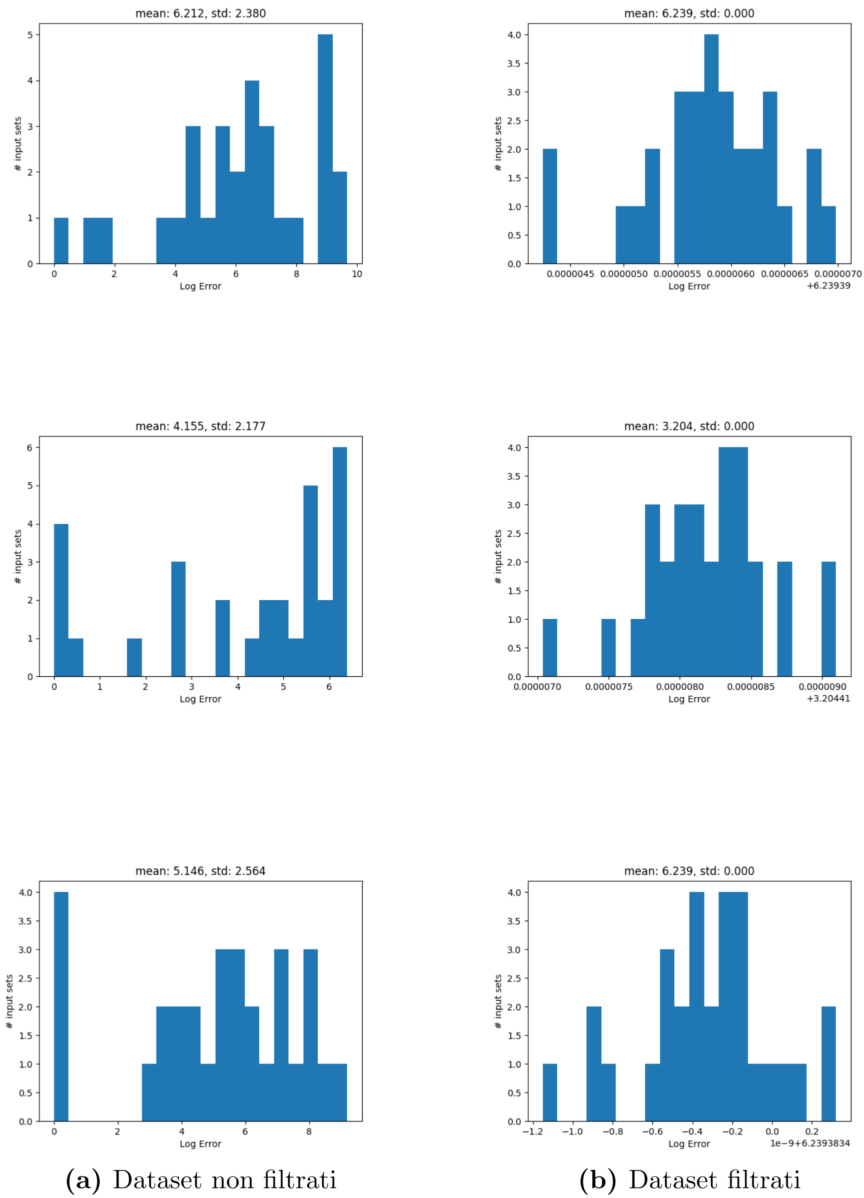


Figura 4.1: Esempi di istogrammi di *dwt* per diverse configurazioni di bit

In effetti, com'è possibile notare, il modo in cui sono distribuiti gli errori non approssima una campana gaussiana, e quindi possiamo dire che, per buona parte dei casi, gli errori (in forma logaritmica) non seguono una distribuzione normale, talvolta a causa di picchi di errori agli estremi dei grafici.
Si faccia particolare attenzione a quest'ultimo aspetto, perché trattandosi di errori logaritmici, più ci si allontana a destra dal valore 1, più il valore dell'inverso del logaritmo cresce esponenzialmente.
Data la suddetta osservazione, si è allora ipotizzato di poter filtrare i dataset, eliminando tutte quelle configurazioni cui corrisponde un errore maggiore di una certa soglia scelta arbitrariamente.

Vengono di seguito riportati degli esempi di grafici di dataset senza filtraggio, accanto ad altrettanti esempi ottenuti con dataset filtrati:



(a) Dataset non filtrati

(b) Dataset filtrati

Figura 4.2: Effetti del filtraggio degli errori per il benchmark *dwt*, con soglia massima 1.

4.3.2 Esperimenti col filtraggio dei dataset

Come si evince dai grafici preliminari, in effetti l'operazione di filtraggio di configurazioni con errori più alti di una certa soglia approssima leggermente di più la loro distribuzione ad una gaussiana.
Si è quindi ritenuto lecito aggiungere l'operazione di filtraggio dei dataset all'interno dello script del modello, in particolare durante fase di preprocessing dei dati.
In particolare, si è scelto di prendere gli script che allenano i modelli per ogni benchmark, tarati nel sottoparagrafo 4.2.3, e di modificarne il codice in maniera tale da eliminare dai dataset tutte le configurazioni con errore maggiore di 1, prima dell'allenamento della rete.

I miglioramenti, per quanto riguarda la predizione della media, sono notevoli, come si evince dalla seguente tabella[1]:

	Prima del filtraggio		Dopo il filtraggio	
	MAE	RMSE	MAE	RMSE
BlackScholes	4.755	6.493	3.045	4.054
correlation	1.533	2.690	1.607	2.878
dwt	1.807	3.170	1.659	2.091
FWT	6.257	7.423	2.504	3.168
Jacobi	6.085	7.003	1.708	2.325
Media	4.091	5.356	2.105	2.903

In particolare è possibile notare miglioramenti netti per *FWT*, *BlackScholes* e soprattutto *Jacobi*, che erano i tre benchmark maggiormente penalizzati negli esperimenti sul modello originale; per quanto riguarda *dwt* abbiamo un leggero miglioramento, comunque non trascurabile, mentre *correlation* non sembra reagire in modo particolarmente diverso dopo il filtraggio.

[1] Va precisato che le suddette performance non sono calcolate sull'intero modello, bensì solamente sulla capacità di predizione del valore medio della distribuzione, effettuando 6 esperimenti identici per ogni benchmark e utilizzando come valore della metrica la sua media lungo tali prove.

Per quanto riguarda invece la capacità di questo nuovo modello di saper predirre la varianza, purtroppo i risultati sono ancora molto variabili da esperimento a esperimento, quindi si è dovuto cercare un metodo diverso di valutazione; si è scelto di utilizzare l'*indice di correlazione di Pearson.* Detto anche *coefficiente di correlazione lineare*, questa metrica è un indice calcolato tra due variabili statistiche, che esprime un'eventuale relazione di linearità tra esse; il suo valore è compreso tra $+1$ e -1, dove il primo corrisponde alla perfetta correlazione lineare positiva, e il secondo corrisponde alla perfetta correlazione lineare negativa, mentre un valore 0 ci evidenzia una totale assenza di correlazione lineare.
L'indice è così calcolato:

$$\rho_{XY} = \frac{\sigma_{XY}}{\sigma_X \sigma_Y}, \quad -1 \leq \rho_{XY} \leq 1$$

Dove σ_{XY} è la covarianza tra X e Y, mentre σ_X e σ_Y sono le due deviazioni standard.
A livello implementativo, per il calcolo di questo coefficiente si è scelto di utilizzare la funzione `corrcoef(x,y)` fornita da *NumPy*, dove, nel nostro caso, `x` e `y` sono rispettivamente il vettore delle varianze predette dal nostro modello e il vettore di varianze reali degli errori calcolate sul dataset originale.

Osserviamo i risultati ottenuti durante gli stessi esperimenti effettuati per la media:

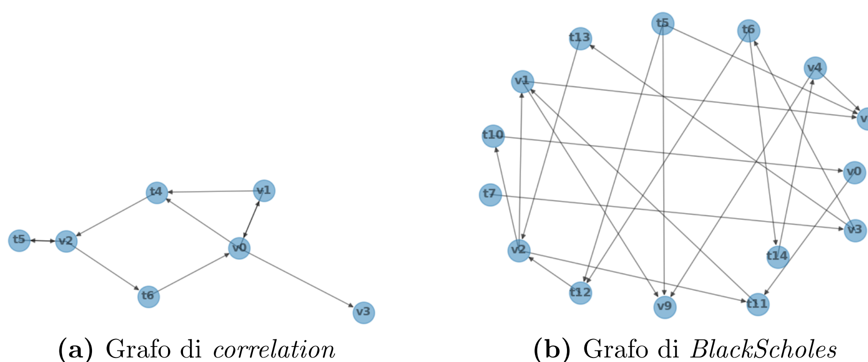
	Indice di correlazione di Pearson	
	Prima del filtraggio	Dopo il filtraggio
BlackScholes	0.6161	0.3516
correlation	0.5998	0.2974
dwt	0.8680	0.8668
FWT	0.8541	0.7707
Jacobi	0.3272	0.0554
Media	0.6530	0.4684

Purtroppo, come si evince dalla tabella, il filtraggio, nonostante abbia l'effetto di migliorare nettamente le performance di predizione della media, generalmente ha anche quello di peggiorare leggermente l'accuratezza della predizione della varianza. Per questa ragione, nel sottoparagrafo successivo verrà analizzata la seconda ipotesi data all'inizio del paragrafo 4.3, con lo scopo di cercare ulteriori miglioramenti.

4.3.3 Considerazioni sulla conoscenza del dominio

Uno degli obiettivi principali dell'elaborato, ricordiamolo, era quello di ottenere dei modelli stocastici che potessero essere quanto più generali e indipendenti possibile dai singoli Input Set, per dover evitare di allenare un modello per ognuno di questi ultimi.
Per cercare di raggiungere tale obiettivo, uno dei primi passi a livello implementativo è stato quello di utilizzare un dataset in una forma tale che ogni esempio contenesse l'errore ottenuto per ogni input set, quindi un target composto da un insieme di valori e non da un singolo valore.

Come ipotizzato all'inizio del paragrafo, è possibile che l'utilizzo di questo artificio non sia in grado, da solo, di fornire una conoscenza del dominio abbastanza completa per l'ottenimento di modelli affidabili.
In effetti, nel lavoro di F. Livi (capitolo 4.5 del suo elaborato di tesi [7]) si era evidenziato come esistessero delle relazioni tra le molteplici variabili di ogni benchmark: ad esempio, date tre variabili V_1, V_2 e V_3, messe tra loro in una relazione $V_1 = V_2 + V_3$, in fase di compilazione del benchmark, la libreria *FlexFloat* imporrà che la precisione in bit di V_1 sia minore o uguale di quella della somma delle precisioni di V_2 e V_3; intuitivamente, ciò significa che la precisione di V_1 dipende da quella delle altre due variabili.
Queste relazioni sono esprimibili come un grafo in cui ogni variabile è rappresentata da un nodo, e ogni arco che entra in un nodo indica una relazione di dipendenza della variabile di partenza con quella di arrivo.



(a) Grafo di *correlation* **(b)** Grafo di *BlackScholes*

Figura 4.3: Esempi di grafi delle dipendenze

Si è allora deciso di estrarre delle feature dal grafo (in forma di disequazione), tramite l'ausilio di tool sviluppati da F. Livi, e di estendere con esse i dataset attuali.
In sostanza, si è passati da dataset di tipo:

Dati	Target
conf_0	err_IS_*
conf_*	err_IS_*
conf_N	err_IS_*

A dataset di tipo:

Dati		Target
conf_0	feature_ext_*_0	err_IS_*
conf_*	feature_ext_*_*	err_IS_*
conf_N	feature_ext_*_N	err_IS_*

Dove le varie estensioni sono diverse per ogni configurazione, in quanto sono rappresentate nella pratica da numeri interi positivi (che indicano un apporto positivo sulla predizione dell'errore) o negativi (che indicano un apporto negativo), in quanto derivanti da disequazioni di tipo $X < Y$, (poste poi in forma $Y - X > 0$) dove X e Y sono precisioni in bit delle rispettive variabili, le quali si trovano nell'analoga relazione $V_X < V_Y$ evidenziata nel grafo.

Nel prossimo sottoparagrafo verranno eseguite sperimentazioni analoghe a quelle effettuate per il filtraggio degli errori, e ne verranno discussi i risultati.

4.3.4 Esperimenti con l'aggiunta di feature del grafo al dataset

Il motivo per cui si è scelto di aggiungere tali feature ai nostri dataset è che nell'elaborato di F. Livi si era notato un netto miglioramento durante l'allenamento del modello predittivo con regressione del singolo punto, rispetto all'utilizzo di dataset privi di quelle informazioni aggiuntive.
Ciò che si spera con questi esperimenti è di osservare un miglioramento analogo del nostro modello tramite l'aggiunta di queste feature, e in particolare si vuole capire se addestrare la rete stocastica aggiungendo della nuova conoscenza sul dominio possa rendere la rete più abile a generalizzare le predizioni su tutti gli Input Set (e di conseguenza ad apprendere meglio la varianza).

Per motivi di coerenza, si è scelto di effettuare gli esperimenti con le stesse identiche modalità di quelli sul filtraggio, aggiungendo quindi agli script una parte di preprocessing che aggiungesse al training set le informazioni del grafo, precedentemente estratte dai benchmark con un tool sviluppato da F. Livi.
Sono qui riportati i risultati per la predizione del valore medio:

	Dataset puro		Dataset esteso	
	MAE	RMSE	MAE	RMSE
BlackScholes	4.755	6.493	4.149	5.894
correlation	1.533	2.690	2.516	3.752
dwt	1.807	3.170	2.207	3.461
FWT	6.257	7.423	2.550	3.551
Jacobi	6.085	7.003	6.321	7.396
Media	4.091	5.356	3.549	4.811

Purtroppo i miglioramenti, a parte per *FWT*, sono marginali o addirittura assenti.

Osserviamo adesso gli indici di correlazione di Pearson per la varianza:

	Indice di correlazione di Pearson	
	Dataset puro	Dataset esteso
BlackScholes	0.6161	0.7106
correlation	0.5998	0.7133
dwt	0.8680	0.8878
FWT	0.8541	0.8747
Jacobi	0.3272	0.3482
Media	0.6530	0.6720

In effetti, a livello di coefficienti trovati, è possibile osservare un netto miglioramento nella maggior parte dei dataset.
Va inoltre segnalato che, durante l'allenamento delle reti, con questo metodo si è ottenuta una convergenza più rapida ad un valore di loss decisamente più basso rispetto a quello ottenuto nel modello originale (con il primo che osserva una loss di validazione più piccola di un range del $10 - 15\%$).
Osservati questi risultati, si è pensato di combinare i due nuovi metodi, con la speranza di osservare ulteriori miglioramenti: tali esperimenti verranno discussi nel prossimo sottoparagrafo, affiancati ad un riepilogo di tutti i risultati ottenuto fino ad adesso.

4.3.5 Combinazione dei due metodi e riepilogo dei risultati

Allo stato attuale, dopo alcune riflessioni a livello teorico che hanno portato allo sviluppo di due metodi di preprocessamento dei dataset, abbiamo ottenuto due varianti del modello originale:

1. Un modello allenato con dataset ottenuti dal filtraggio di errori grandi;
2. Un modello allenato con dataset estesi con feature ricavate dai grafi delle dipendenze dei benchmark;

Tramite la prima variante abbiamo potuto osservare come, tramite il filtraggio, la distribuzione degli errori potesse approssimarsi leggermente di più ad una gaussiana, permettendo così al metodo MLE di lavorare con i giusti prerequisiti, ottenendo grossi miglioramenti per quanto riguarda la predizione del valore medio dell'errore, ma ottenendo un leggero peggioramento per la varianza.
Utilizzando la seconda variante di modello abbiamo notato come l'aggiunta di nuova conoscenza del dominio (e quindi informazioni aggiuntive riguardo le relazioni tra le variabili del benchmark), come nel lavoro di F. Livi [7], potesse portare a miglioramenti non indifferenti per la varianza, e per la velocità di convergenza dell'allenamento della rete verso un valore di loss più piccolo rispetto al modello originale, affiancato però ad un leggero peggioramento della capacità di predizione del valore medio.

Si è allora pensato di provare ad unire le due metodologie, modificando il codice del modello affinché la fase di preprocessamento:

1. Elimini le configurazioni cui è associato un errore più grande di 1;
2. Aggiunga a tali esempi delle colonne contenenti le informazioni sulle relazioni tra le variabili;

Ciò che ci si augura di osservare, è un miglioramento sia per quanto riguarda la capacità di predizione della media, sia (soprattutto) per quanto riguarda la capacità di predizione della varianza; in caso contrario, visti i risultati precedenti, ci si aspetterebbe di trovare dei modelli con performance analoghe a quelle del modello base, piuttosto che un peggioramento.

Effettuati i consueti esperimenti, otteniamo i seguenti risultati:

	Dataset puro		Dataset esteso e filtrato	
	MAE	RMSE	MAE	RMSE
BlackScholes	4.755	6.493	2.934	3.970
correlation	1.533	2.690	1.932	3.077
dwt	1.807	3.170	1.233	1.734
FWT	6.257	7.423	1.802	2.310
Jacobi	6.085	7.003	1.554	2.063
Media	4.091	5.356	1.891	2.631

	Indice di correlazione di Pearson	
	Dataset puro	Dataset esteso e filtrato
BlackScholes	0.6161	0.3000
correlation	0.5998	0.3117
dwt	0.8680	0.9395
FWT	0.8541	0.7593
Jacobi	0.3272	0.0955
Media	0.6530	0.4812

Come è possibile notare, l'unione dei due metodi fa osservare un miglioramento molto grande delle performance di predizione del valore medio in tutti i benchmark tranne *correlation*, per il quale generalmente si ha una performance comparabile a quella del modello originale.
Purtroppo, lo stesso non si può dire per la predizione della varianza, la cui performance risulta essere peggiorata di molto in alcuni casi (*BlackScholes*, *correlation* e *Jacobi*), peggiorata marginalmente in *FWT*, e migliorata in *dwt*.

Vengono riportate di seguito delle tabelle riassuntive, che mostrano, testa a testa, i risultati ottenuti all'interno di questo paragrafo:

	Dataset puro		Dataset filtrato		Dataset esteso		Dataset esteso e filtrato	
	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE
BlackScholes	4.755	6.493	3.045	4.054	4.149	5.894	2.934	3.970
correlation	1.533	2.690	1.607	2.878	2.516	3.752	1.932	3.077
dwt	1.807	3.170	1.659	2.091	2.207	3.461	1.233	1.734
FWT	6.257	7.423	2.504	3.168	2.550	3.551	1.802	2.310
Jacobi	6.085	7.003	1.708	2.325	6.321	7.396	1.554	2.063
Media	4.091	5.356	2.105	2.903	3.549	4.811	1.891	2.631

	Indice di correlazione di Pearson			
	Dataset puro	Dataset filtrato	Dataset esteso	Dataset esteso e filtrato
BlackScholes	0.6161	0.3516	0.7106	0.3000
correlation	0.5998	0.2974	0.7133	0.3117
dwt	0.8680	0.8668	0.8878	0.9395
FWT	0.8541	0.7707	0.8747	0.7593
Jacobi	0.3272	0.0554	0.3482	0.0955
Media	0.6530	0.4684	0.6720	0.4812

Per un'analisi dettagliata dei suddetti risultati, si rimanda al Capitolo 5, dedicato all'aggregazione e al confronto dei risultati sperimentali ottenuti all'interno di questo capitolo di progetto.
Si noti però come il contributo maggiore sul cambiamento delle performance di modello sia dato dal filtraggio degli errori, osservazione che ci fa capire quanto l'assunto di avere una distribuzione gaussiana dei valori sia importante per la trattazione del problema con il metodo Maximum Likelihood Estimation.

4.4 Esperimenti di codifica degli Input Set in forma compressa

Ricapitolando quanto detto nei paragrafi precedenti, fino ad adesso si sono studiati e applicati metodi per:

1. Creare un modello stocastico che apprenda la distribuzione statistica dell'errore tramite MLE;

2. Migliorare il suddetto modello filtrando dai training set gli esempi i cui errori superavano una certa soglia, per approssimare la distribuzione degli errori ad una normale;

3. Migliorare il suddetto modello aggiungendo ad esso nuova conoscenza del dominio mutuata dalle informazioni estratte dei grafi delle dipendenze.

Nonostante le evidenti migliorie apportate dall'applicazione delle ultime due metodologie, si vorrebbe comunque cercare di ottenere un modello ancora più performante.

Essendo uno degli scopi dell'elaborato la generalizzazione del modello predittivo, e avendo per questa ragione indotto la nostra rete neurale ad apprendere la distribuzione degli errori a partire dai loro valori ottenuti per ognuno dei 30 Input Set, si potrebbe pensare di aggiungere questi ultimi come ulteriore conoscenza del dominio per il training della rete.
In effetti, una rete neurale allenata con queste modalità apprenderebbe senza grossi problemi la correlazione tra configurazione ed errore rispetto l'Input Set (almeno in teoria). Purtroppo però, non è possibile includere direttamente gli Input Set come feature all'interno dei nostri dataset. Gli Input Set sono infatti vettori di cardinalità variabile in base al benchmark, i quali potrebbero, per un solo benchmark e affiancati l'un l'altro, risultare in un vettore cumulativo contenente diverse centinaia di migliaia di elementi.
Chiaramente, è impensabile allenare una rete neurale con dataset del genere, in quanto risulterebbe lento, inefficiente, e decisamente poco pratico. È giocoforza quindi trovare una strada, e in particolare un automatismo, che ci permetta di ottenere una rappresentazione "compressa" degli Input Set, e di ridurre quindi la dimensionalità delle feature.

4.4.1 Utilizzo di un autoencoder per ridurre la dimensionalità dei vettori dell'Input Set

Nel capitolo 2, sottoparagrafo 2.2.2, sono stati descritti gli autoencoder, particolari reti neurali con una specifica architettura utilizzate per imparare una rappresentazione ridotta di un set di dati, riducendone così la dimensionalità, cercando di minimizzare l'errore di ricostruzione (quindi MAE e RMSE tra input e output dela rete).
Considerato quanto detto alla fine del sottoparagrafo precedente, un autoencoder potrebbe in effetti rappresentare una soluzione al nostro problema di riduzione di dimensionalità. Si potrebbe infatti costruire, per ogni benchmark, una rete costituita dai due blocchi *encoder* e *decoder*, che prenda in ingresso i vari Input Set, così da ottenere nell'Hidden Layer centrale (detto strato "latente") una loro rappresentazione compressa.

Questo approccio, a prima vista, presenta due possibili punti di fallimento indipendenti:

- Gli Input Set sono stati generati in modo randomico, e in quanto tali non sono comprimibili se non introducendo della perdita di informazione di una certa entità;
- È possibile che si possa trovare una dimensione dello strato latente tale da poter fornire una rappresentazione accettabile del vettore degli Input Set, ma tale anche da non essere abbastanza ridotta per consentire l'allenamento della rete con l'aggiunta di tali feature.

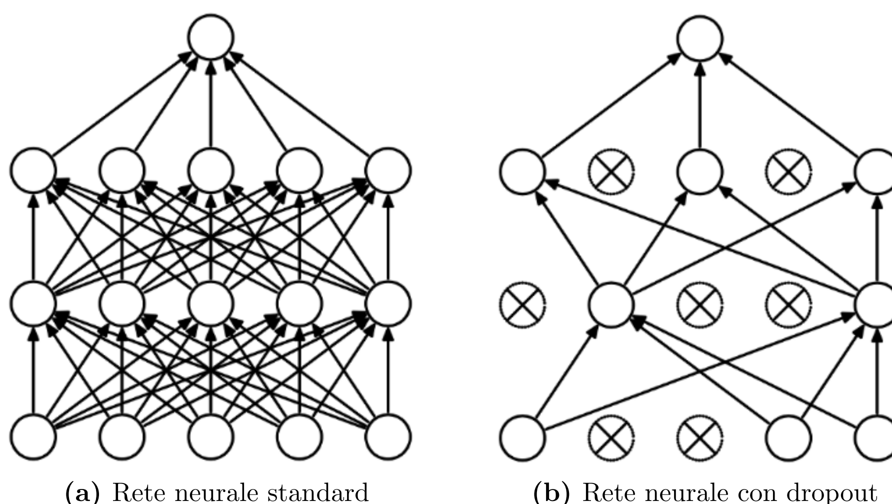
In ogni caso, si è ritenuto interessante effettuare degli esperimenti tentativi, per comprendere in che misura possono verificarsi i due fenomeni sopra descritti.
In particolare si è scelto di utilizzare, in prima battuta, soltanto il benchmark *dwt* per queste sperimentazioni preliminari, in quanto è un benchmark relativamente semplice che ha precedentemente reagito bene a tutte le metodologie applicate durante la costruzione del modello stocastico, e risulta quindi essere un test bed adeguato: in assenza di buoni risultati con questo benchmark, risulterebbe inutile proseguire con tutti gli altri.

Di seguito sono riportate le parti salienti dello script Python prodotto per l'esperimento:

```python
def autoencoder(benchmark, n_target_layers, n_target_neurons, epochs, dr_rate):
    # ... Caricamento dei file di input e scaling
    input_vector = Input(shape=input_shape)

    for i in range(n_target_layers):
        if i == 0:
            encoded = Dense(get_number_of_nodes(columns,n_target_layers,
 n_target_neurons,n_target_layers-i), activation='relu')(input_vector)
            encoded = Dropout(dr_rate)(encoded)
        else:
            encoded = Dense(get_number_of_nodes(columns,n_target_layers,
 n_target_neurons,n_target_layers-i), activation='relu')(encoded)
            encoded = Dropout(dr_rate)(encoded)

    if not n_target_layers == 0:
        encoded = Dense(n_target_neurons, activation='relu')(encoded)
    else:
        encoded = Dense(n_target_neurons, activation='relu')(input_vector)

    for i in range(n_target_layers):
        if i == 0:
            decoded = Dense(get_number_of_nodes(columns,n_target_layers,
 n_target_neurons,i+1), activation='relu')(encoded)
            decoded = Dropout(dr_rate)(decoded)
        else:
            decoded = Dense(get_number_of_nodes(columns,n_target_layers,
 n_target_neurons,i+1), activation='relu')(decoded)
            decoded = Dropout(dr_rate)(decoded)

    if not n_target_layers == 0:
        decoded = Dense(columns, activation='linear')(decoded)
    else:
        decoded = Dense(columns, activation='linear')(encoded)

    autoencoder = Model(input_vector, decoded)

    # Separazione dei due modelli
    # Encoder
    encoder = Model(input_vector, encoded)

    # Decoder
    for i in range(-int(len(autoencoder.layers)/2), 0, 1):
        if i == -int(len(autoencoder.layers)/2):
            decoder_layers = autoencoder.layers[i](encoded_input)
        else:
            decoder_layers = autoencoder.layers[i](decoder_layers)

    decoder = Model(encoded_input, decoder_layers)

    NAME = "AE_{}_{}-targetlayers_{}-targetneurons_{}-epochs_{}-dropout".format(
 benchmark, n_target_layers, n_target_neurons, epochs, dr_rate)
    tensorboard = TensorBoard(log_dir="logs/{}".format(NAME))

    autoencoder.compile(optimizer='adadelta', loss='mean_absolute_error', metrics
 =['mean_absolute_error', tf.keras.metrics.RootMeanSquaredError(name='rmse')])

    autoencoder.fit(train_data, train_data, epochs=epochs, batch_size=64,
                    shuffle=True, verbose=True,
                    validation_data=(val_data, val_data), callbacks=[tensorboard])
    os.mkdir("saved_models/{}/".format(NAME))
    autoencoder.save("saved_models/{}/autoencoder.h5".format(NAME))
    encoder.save("saved_models/{}/encoder.h5".format(NAME))
    decoder.save("saved_models/{}/decoder.h5".format(NAME))
    predicted = autoencoder.fit(test_data)
    actual = test_data
    MAE_test, RMSE_test = evaluate_predictions(predicted, actual)
```

Come si può notare, lo script contiene degli iperparametri, i quali sono stati tarati questa volta utilizzando un metodo di tuning manuale (uno script scritto in bash che allena diverse centinaia di modelli con diversi range di valori ragionevolmente scelti per ognuno degli iperparametri) affiancato ad un controllo delle performance dei modelli ottenuti con *TensorBoard*, con lo scopo di isolare il migliore tra questi.
Gli iperparametri in questione sono: numero di epoch di training, numero di hidden layer che precedono (e che seguono anche, quindi) lo strato latente (`n_target_layers`), numero di neuroni all'interno dello strato latente (`n_target_neurons`), e la *dropout rate* (`dr_rate`).

Particolare attenzione va fatta per l'ultimo di questi iperparametri; ricordiamo infatti di stare lavorando con un insieme di 30 Input Set per ogni benchmark, quindi nell'effettivo disponiamo di "pochi" dati per un allenamento ottimale dell'autoencoder.
In casi come questi è possibile incorrere in situazioni di overfitting e quindi è necessario correre ai ripari utilizzando tecniche che lo contrastino: per questo determinato caso si è scelto di utilizzare la tecnica del *dropout* [31], la quale consiste nel disattivare una frazione (data dalla *dropout rate*) de neuroni presenti all'interno degli strati della rete ad ogni iterazione del processo di allenamento.



(a) Rete neurale standard **(b)** Rete neurale con dropout

Figura 4.4: Effetti del dropout di neuroni dagli strati di una rete neurale

Si noti come questo aspetto è stato implementato nel nostro script: si è scelto di utilizzare un componente offerto da *Keras*, ovvero `Dropout(rate)`, dove il parametro `rate` è un numero float compreso tra 0 e 1, e rappresenta la frazione di unità di input da "spegnere"; questo componente è stato inserito subito dopo ogni strato `Dense` (intuitivamente così il Dropout layer raccoglie i risultati della precedente funzione di attivazione e filtra gli input da inviare al layer successivo, disattivando di fatto alcuni neuroni).

Una volta eseguito il processo di tuning otteniamo il seguente set di iperparametri:

- *Numero di hidden layer tra Input e strato latente* (replicati tra strato latente e Output): 1
- *Numero di neuroni nello strato latente*: 1200
- *Numero di epoch*: 1000
- *Dropout rate*: 0.1 (il 10% dei neuroni di ogni strato eccetto Input e Output viene spento)

L'autoencoder allenato per *dwt*, utilizzando i suddetti iperparametri presenta le seguenti performance sul test set (valori medi calcolati su 10 esperimenti identici):

- *MAE*: 0.3217
- *RMSE*: 0.3969

Purtroppo, questi risultati non sono soddisfacenti e, confermando le ipotesi di fallimento teorizzate all'inizio del sottoparagrafo, ci portano ad abbandonare l'idea di effettuare ulteriori esperimenti su questa strada. Sarà allora necessario, nei prossimi paragrafi, pensare ad una strada alternativa che utilizzi informazioni ricavate dagli Input Set, senza il bisogno di dover necessariamente utilizzare questi ultimi in forma pura, durante l'allenamento della rete.

4.5 Una riflessione parallela: il modello di regressione *ML_mean*

Fino ad adesso si è dato per scontato che un modello allenato con tecniche "classiche" (ad esempio utilizzando la MAE come funzione di loss) potesse non essere adeguato per la generalizzazione della rete rispetto agli Input Set, ovvero per il nostro problema principale.
Si è però pensato di eseguire un'analisi parallela agli studi ed esperimenti con MLE, in linea con il lavoro effettuato da F. Livi [7]: cosa succederebbe se noi allenassimo una rete neurale (per un compito di regressione lineare) utilizzando la MAE ad apprendere il valore medio dell'errore tra quelli ottenuti per i vari Input Set? Si noti la differenza rispetto allo studio di F. Livi, nel nostro caso la regressione non effettuata per errori specifici ad un determinato Input Set, bensì per il valore medio calcolato per ognuno di essi.

In particolare, quanto sarebbe *trasferibile* (a tal proposito si faccia riferimento al paragrafo 2.3 sul *Transfer Learning*) questo modello per ognuno degli Input Set? Ovvero, un modello allenato per ogni benchmark con tali modalità, risulta essere *generale* nei confronti di qualunque altro Input Set?
Per rispondere a questo quesito si è quindi scelto di aprire questa strada parallela e di effettuare degli adeguati esperimenti, descritti nel dettagli nel seguente sottoparagrafo ma che comprendono il preprocessing dei dati per l'adeguamento a tale approccio, la creazione e il tuning di una rete neurale (detta *ML_mean*) per ogni benchmark, e la valutazione delle performance su diversi test set (quello "medio" e quelli relativi ad ognuno degli Input Set).

4.5.1 Implementazione di *ML_mean*

Prima di procedere con la scrittura del codice, è stato necessario fare delle considerazioni riguardo i dataset da utilizzare per training, validazione e test; precedentemente avevamo utilizzato dataset in questa forma:

Dati			Target		
conf_0_var_0	conf_0_var_...	conf_0_var_M	err_conf_0_IS_0	err_conf_0_IS_...	err_conf_0_IS_29
conf_..._var_0	conf_..._var_...	conf_..._var_M	err_conf_..._IS_0	err_conf_..._IS_...	err_conf_..._IS_29
conf_N_var_0	conf_N_var_...	conf_0_var_M	err_conf_N_IS_0	err_conf_N_IS_...	err_conf_N_IS_29

Il problema è che adesso l'allenamento della rete non lo si vuole fatto utilizzando un insieme di valori come target, bensì utilizzando un valore unico per ogni esempio, e quel valore unico deve essere la media dei vari `err_conf_k_IS_*`, dove `k` rappresenta la `k`-esima riga del dataset.
Inoltre, adesso è necessario che vi siano diversi test set: uno costruito utilizzando come target la media dei valori d'errore, altri 30 costruiti utilizzando come target l'errore associato di volta in volta ad un Input Set distinto.

A causa di queste importanti considerazioni, si è scelto questa volta di scrivere da zero una funzione di preprocessing appositamente per il problema in esame (mentre precedentemente veniva utilizzata una libreria di funzioni scritta appositamente dal Dr. A. Borghesi durante i lavori per OPRECOMP al LIA).
Vengono qui riportate le parti salienti del codice della suddetta funzione:

```
def data_preprocessing(benchmark, train, test, i, flatten_large_errors=True):
    # Lettura dataset da file
    df_train, df_test, df_val = reader.read_replicable_dataset(5000, 1000,
     benchmark, 0)
    ninput = len(list(df_train.filter(regex='var_*')))

    # Costruzione degli N dataset separati per ogni IS (N = numero di Input Set)
    # Struttura:
    #
    # SET 0                                  SET N
    # config_0   err_IS_0                    config_0   err_IS_N
    # config_1   err_IS_0        ...         config_1   err_IS_N
    # ...        ...                         ...        ...
    # config_M   err_IS_0                    config_M   err_IS_N

    n_IS = 0
    separate_IS_datasets = []

    for c in df_test.columns:
        if 'err_' in c:
            n_IS += 1

    for i in range(0,n_IS):
        current_label = 'err_ds_' + str(i)
        IS_dataset = df_test.iloc[:,0:ninput]
        IS_dataset = IS_dataset.join(df_test[current_label])
        separate_IS_datasets.append(IS_dataset)

    # Continua...
```

```
# Continua...


# Costruzione training, test e validation set
# Struttura:
# config_0  err_mean
# config_1  err_mean
# ...       ...
# config_M  err_mean

label_target = 'err_mean'

errs_cols_train = []
errs_cols_test = []
errs_cols_val = []

for c in df_train.columns:
    if 'err_' in c:
        errs_cols_train.append(c)
        df_train[c] = df_train[c].mask(df_train[c] > flatten_value,
 flatten_value, inplace=False)
df_train['err_mean'] = np.mean(df_train[errs_cols_train], axis=1)
df_train = df_train.loc[:,~df_train.columns.str.startswith('err_ds_')]

for c in df_test.columns:
    if 'err_' in c:
        errs_cols_test.append(c)
        df_test[c] = df_test[c].mask(df_test[c] > flatten_value, flatten_value
 , inplace=False)
df_test['err_mean'] = np.mean(df_test[errs_cols_test], axis=1)
df_test = df_test.loc[:,~df_test.columns.str.startswith('err_ds_')]

for c in df_val.columns:
    if 'err_' in c:
        errs_cols_val.append(c)
        df_val[c] = df_val[c].mask(df_val[c] > flatten_value, flatten_value,
 inplace=False)
df_val['err_mean'] = np.mean(df_val[errs_cols_val], axis=1)
df_val = df_val.loc[:,~df_val.columns.str.startswith('err_ds_')]

result_separate_IS_datasets = []

# Normalizzazione con MinMaxScaler
scaler = preprocessing.MinMaxScaler()

x_train = scaler.fit_transform(df_train.iloc[:,0:ninput])
y_train = scaler.fit_transform(np.array(df_train[label_target]
        ).reshape((-1, 1)))
x_test = scaler.fit_transform(df_test.iloc[:,0:ninput])
y_test = scaler.fit_transform(np.array(df_test[label_target]
        ).reshape((-1, 1)))
x_val = scaler.fit_transform(df_val.iloc[:,0:ninput])
y_val = scaler.fit_transform(np.array(df_val[label_target]
        ).reshape((-1, 1)))

for i, df in enumerate(separate_IS_datasets):
    x = scaler.fit_transform(df.iloc[:,0:ninput])
    y = scaler.fit_transform(np.array(df.loc[:,
        ~df.columns.str.startswith('var_')]
        ).reshape((-1, 1)))
    result_separate_IS_datasets.append([x, y])

return x_train, y_train, x_test, y_test, x_val, y_val,
       result_separate_IS_datasets, scaler
```

Come si può notare, all'interno della funzione vengono creati tre dataset (training, validation e test) che utilizzano come target il valore medio dell'errore calcolato tra gli errori per ogni dataset, più altri 30 test set (uno per ogni Input Set), utilizzati per valutare come si comporta il modello in situazioni di transfer learning.

Una volta effettuato il preprocessamento dei dati in modo corretto, siamo pronti per allenare il modello, del cui codice vengono riportate, come di consueto, le parti salienti:

```
def ML_mean(benchmark, n_dense, node_mul, l_r, dec, epochs, batch_size):
    # Data preprocessing
    (x_train, y_train, x_test, y_test, x_val, y_val, IS_datasets, scaler) =
     data_preprocessing(benchmark, 5000, 1000, 0)

    # ...

    # Callback
    # ...
    tensorboard = TensorBoard(log_dir="logs/{}".format(NAME))

    # Creazione del modello
    n_samples, n_attributes = x_train.shape
    input_shape = (x_train.shape[1],)

    ML_mean_model = Sequential()
    ML_mean_model.add(Dense(int(n_attributes * node_mul),
        activity_regularizer=l1(1e-5), activation='relu',
        input_shape=input_shape))

    for n in range(n_dense):
        ML_mean_model.add(Dense(int(n_attributes * node_mul), activation='relu'))

    ML_mean_model.add(Dense(1, activation='linear'))

    # Compilazione e allenamento del modello
    adam = optimizers.Adam(lr=l_r, decay=dec)
    ML_mean_model.compile(optimizer=adam, loss='mean_absolute_error',
                metrics=['mean_absolute_error', tf.keras.metrics.
     RootMeanSquaredError(name='rmse')])

    history = ML_mean_model.fit(x_train, y_train, epochs=epochs, shuffle=True,
                validation_data=(x_val, y_val), batch_size=batch_size,
                callbacks=[early_stopping, reduce_lr, tensorboard], verbose=True)

    # Test del modello usando il test set medio
    predicted = ML_mean_model.predict(x_test)
    actual = y_test
    MAE_test, RMSE_test = evaluate_predictions(predicted, actual, scaler, True)

    # Test del modello per transfer learning con i 30 test set
    metrics_results = []

    for i, element in enumerate(IS_datasets):
        pr = ML_mean_model.predict(element[0])
        ac = element[1]
        MAE_IS, RMSE_IS = evaluate_predictions(pr, ac, scaler, False)
        metrics_results.append([MAE_IS, RMSE_IS])
```

Anche in questa occasione abbiamo a che fare con degli iperparametri, i quali, questa volta (per motivi di impossibilità tecnica con *Keras Tuner*, essendo un tool recente e ancora acerbo sotto alcuni punti di vista), verranno tarati con il tuning manuale (per un totale di 2000 possibilità), utilizzando anche *TensorBoard* per l'ottenimento della migliore combinazione con cui allenare il modello.

4.5.2 Esperimenti e risultati

Per questa fase del progetto si è scelto di utilizzare i benchmark *Black-Scholes*, *convolution*, *correlation*, *dwt*, *Jacobi*, *saxpy*; FWT è stato escluso dalla trattazione in questa sede, in quanto poco significativo (possiede solo due variabili di input).
Effettuato il tuning degli iperparametri, sono state trovate le seguenti migliori combinazioni per la creazione di modelli ottimali:

- *BlackScholes*:
  - Hidden Layer: 1
  - Numero neuroni per ogni hidden layer: $4 * num_features$;
  - Batch size: 32
  - Epoch: 119
  - Learning rate e decay: 0.01, 0.0005
- *convolution*:
  - Hidden Layer: 4
  - Numero neuroni per ogni hidden layer: $24 * num_features$;
  - Batch size: 256
  - Epoch: 14
  - Learning rate e decay: 0.005, 0.0005
- *correlation*:
  - Hidden Layer: 2
  - Numero neuroni per ogni hidden layer: $4 * num_features$;
  - Batch size: 8
  - Epoch: 65
  - Learning rate e decay: 0.01, 0.0005
- *saxpy*:
  - Hidden Layer: 4
  - Numero neuroni per ogni hidden layer: $2 * num_features$;
  - Batch size: 256
  - Epoch: 133
  - Learning rate e decay: 0.01, 0.005
- *dwt*:
  - Hidden Layer: 4
  - Numero neuroni per ogni hidden layer: $12 * num_features$;
  - Batch size: 8
  - Epoch: 71
  - Learning rate e decay: 0.01, 0.0005
- *Jacobi*:
  - Hidden Layer: 12
  - Numero neuroni per ogni hidden layer: $12 * num_features$;
  - Batch size: 32
  - Epoch: 106
  - Learning rate e decay: 0.0005, 0.0005

Una volta compilati tali modelli, questi sono stati adeguatamente testati con i test set ottenuti dalla fase di preprocessing, fornendo i seguenti risultati:

	Performance Test Set medio		Performance Test Set per gli Input Set			
Benchmark	MAE	RMSE	μ MAE	σ MAE	μ RMSE	σ RMSE
BlackScholes	0.060	0.180	0.728	0.003	0.834	0.002
convolution	0.025	0.101	0.025	0.001	0.100	0.006
correlation	0.049	0.115	0.315	0.017	0.479	0.019
dwt	0.009	0.031	0.120	0.144	0.238	0.145
Jacobi	0.053	0.205	0.920	0.013	0.938	0.011
saxpy	0.010	0.052	0.010	0.000	0.052	0.000
Media	0.034	0.114	0.353	0.030	0.440	0.030

Come si può osservare, nonostante i risultati sul test set medio siano spesso più che discreti, la capacità del modello di essere trasferibile purtroppo varia di caso in caso, con una tendenza alla non trasferibilità; questo significa che purtroppo non è possibile, con questa metodologia specifica, allenare un regressore "semplice" e renderlo generale rispetto ai vari Input Set.
All'interno del prossimo (e ultimo) paragrafo di questo capitolo, verrà analizzata la possibilità di creare un modello con metodo MLE che utilizzi delle feature aggiuntive create da zero a partire da parametri statistici ottenuti dai valori presenti all'interno degli Input Set, e ne verrà studiata la trasferibilità in modo analogo a come fatto precedentemente.

4.6 Un nuovo modello con feature statistiche ricavate degli Input Set: *ML_MLE*

In questa ultima parte di progetto, l'obiettivo è cercare una strada alternativa per fornire in input alla rete neurale una rappresentazione alternativa degli Input Set come feature dei dataset, in modo da non essere costretti a a doverli utilizzare in modo "grezzo" (a prescindere che siano puri o compressi).
Più precisamente, l'idea è quella di calcolare delle nuove feature aggiuntive da zero, partendo dai valori presenti all'interno degli Input Set.
In particolare, un Input Set è un vettore di valori (o una matrice, nel caso di *BlackScholes*, che per le nostre sperimentazioni è stato generato randomicamente, per cui non è compribimile; però, da essi è possibile estrarre dei valori statistici che li caratterizzino, in modo da ottenere delle feature che la rete cercherà di mettere in relazione con la distribuzione degli errori.
Si torna quindi adesso a ragionare in termini di Maximum Likelihood Estimation, e quindi in termini di modello stocastico, riprendendo in particolare quel filone di esperimenti che cercava la generalizzazione dei modelli rispetto agli Input Set utilizzando delle feature aggiuntive che apportassero una maggiore conoscenza del dominio del problema.

Anche per questo modello, come per *ML_mean*, bisogna parlare di preprocessing dei dati, in quanto è necessario calcolare le nuove feature come valori statistici calcolati per ognuno degli Input Set per ogni benchmark; come è possibile vedere nel sottoparagrafo successivo, la complessità di questa operazione ha richiesto la scrittura di uno script separato da quello di modello, per non sporcare troppo il codice di quest'ultimo.

4.6.1 Creazione delle feature aggiuntive e preprocessamento dei dataset

Il primo passo per la creazione di queste nuove informazioni è la scelta di quali metriche statistiche utilizzare per i calcoli. Al fine di caratterizzare al meglio ogni Input Set, si è scelto di utilizzare un insieme di 13 funzioni statistiche:

- Media aritmetica;
- Deviazione standard;
- Varianza;
- 5-percentile;
- 25-percentile;
- 50-percentile;
- 75-percentile;
- 95-percentile;
- Indice di asimmetria (skewness);
- Curtosi;
- Media armonica;
- Media geometrica;
- Coefficiente di variazione;

Ognuna di queste funzioni è stata calcolata per ognuno dei 30 Input Set, ottenendo infine un dataset con 30 righe, e 13 colonne.
Nella pagina successiva è riportato il codice relativo alla parte appena descritta, dallo script `create_feature_extension.py`.

```python
def main(argv):
    if not len(argv) == 1:
        print('\nERROR: not enough arguments!\n')
        print('USAGE: python create_feature_extensions.py <benchmark>\n')
        sys.exit()

    benchmark = argv[0]

    df = pd.read_csv("./csv_datasets/{}_input_sets.csv".format(benchmark))

    result_df = pd.DataFrame()
    result_df['mean'] = df.apply(lambda x: stats.tmean(x), axis=1)
    result_df['std'] = df.apply(lambda x: stats.tstd(x), axis=1)
    result_df['variance'] = df.apply(lambda x: stats.tvar(x), axis=1)
    result_df['5-percentile'] = df.apply(lambda x: np.percentile(x, 5), axis=1)
    result_df['25-percentile'] = df.apply(lambda x: np.percentile(x, 25), axis=1)
    result_df['50-percentile'] = df.apply(lambda x: np.percentile(x, 50), axis=1)
    result_df['75-percentile'] = df.apply(lambda x: np.percentile(x, 75), axis=1)
    result_df['95-percentile'] = df.apply(lambda x: np.percentile(x, 95), axis=1)
    result_df['skewness'] = df.apply(lambda x: stats.skew(x), axis=1)
    result_df['kurtosis'] = df.apply(lambda x: stats.kurtosis(x), axis=1)
    result_df['hmean'] = df.apply(lambda x: stats.hmean(x), axis=1)
    result_df['gmean'] = df.apply(lambda x: stats.gmean(x), axis=1)
    result_df['variation'] = df.apply(lambda x: stats.variation(x), axis=1)

    result_df.to_csv("{}_feature_extension.csv".format(benchmark), index=False)

if __name__ == '__main__':
    main(sys.argv[1:])
```

Come si può notare, le varie misure statistiche sono calcolate utilizzando *NumPy* e *SciPy* [32], una libreria open source di algoritmi e strumenti matematici per Python. Si noti come alla fine del processamento, viene salvato un file in formato `csv`, per evitare il ricalcolo delle informazioni ad ogni allenamento del modello.

La parte di preprocessing di dati sospesa con questo script, viene portata a termine all'interno dello script `ML_MLE.py`, nella funzione `data_preprocessing`, della quale sono di seguito riportate le parti salienti:

```python
def data_preprocessing(benchmark, i, flatten_large_errors=True):
    # ...
    # Lettura dei dataset preprocessati (se esistono):
    if os.path.isfile('./pr_sets/{}_{}_train_set.csv'.format(benchmark, i)):
        whole_train_set = pd.read_csv('./pr_sets/{}_{}_train_set.csv'.format(
 benchmark, i))
    else:
        print('./pr_sets/{}_{}_train_set.csv MISSING!'.format(benchmark, i))
        train_flag = True

    # ... operazione analoga per test set medio, test set individuale
    #     per ogni Input Set, e validation set

    # Lettura dei dataset se i dati preprocessati non esistono
    if train_flag or test_flag or test_is_flag or val_flag:
        if benchmark == "Jacobi":
            df_train, df_test, df_val = reader.read_replicable_dataset(2000, 1000,
  benchmark, i)
        else:
            df_train, df_test, df_val = reader.read_replicable_dataset(5000, 1000,
  benchmark, i)
        ninput = len(list(df_train.filter(regex='var_*')))

        # Lettura delle feature statistiche
        feature_extension = pd.read_csv("./feature_extensions/{}_feature_extension
 .csv".format(benchmark))
    # Continua...
```

```python
# Continua...
# Costruzione dei dataset con feature estese
# Struttura:
#
# config_0  ext_feat_0_0  ...  ext_feat_0_K  err_IS_0
# config_1  ext_feat_0_0  ...  ext_feat_0_K  err_IS_0
#                       ...                    ...
# config_M  ext_feat_0_0  ...  ext_feat_0_K  err_IS_0
#                       ...                    ...
# config_0  ext_feat_N_0  ...  ext_feat_N_K  err_IS_N
# config_1  ext_feat_N_0  ...  ext_feat_N_K  err_IS_N
#                       ...                    ...
# config_M  ext_feat_N_0  ...  ext_feat_N_K  err_IS_N

if train_flag:
    print("DEBUG: building train set...")

    n_IS = 0
    whole_train_set = pd.DataFrame()

    for c in df_train.columns:
        if 'err_' in c:
            n_IS += 1

    for i_NS in range(0,n_IS):
        current_label = 'err_ds_' + str(i_NS)
        IS_dataset = df_train.iloc[:,0:ninput]

        for feature_number in range(len(feature_extension.iloc[i_NS])):
            current_key = feature_extension.iloc[i_NS].keys().values[
feature_number]
            IS_dataset[current_key] = feature_extension.iloc[i_NS].get(
current_key)
            #print(feature_extension.iloc[i_NS].get(current_key))

        IS_dataset = IS_dataset.join(df_train[current_label])
        IS_dataset.columns = ['err' if x==current_label else x for x in
IS_dataset.columns]

        whole_train_set = pd.concat([whole_train_set, IS_dataset], axis=0)

    # Flattening big errors to flatten_value
    whole_train_set['err'] = whole_train_set['err'].mask(whole_train_set['err'
] > flatten_value, flatten_value, inplace=False)

    print(whole_train_set)
    print("DEBUG: train set DONE.")
    whole_train_set.to_csv("./pr_sets/{}_{}_train_set.csv".format(benchmark, i
), index=False)

# ... costruzione analoga del validation set

# Il test set   costruito in modo diverso;
#
# Costruzione dei 30 test set individuali:
# Struttura:
#
# SET 0:                                SET 29:
# config_0  ext_IS_0  err_IS_0          config_0  ext_IS_29  err_IS_29
# config_1  ext_IS_0  err_IS_0          config_1  ext_IS_29  err_IS_29
# config_2  ext_IS_0  err_IS_0    ...   config_2  ext_IS_29  err_IS_29
# ...                                   ...
# config_N  ext_IS_0  err_IS_0          config_N  ext_IS_29  err_IS_29
#
# Costruzione del set con errore medio:
# Struttura:
#
# config_0  ext_IS_0  err_mean_config_0
# ...
# config_N  ext_IS_0  err_mean_config_N
#                ...
# config_0  ext_IS_29  err_mean_config_0
# ...
# config_N  ext_IS_29  err_mean_config_N


# Continua...
```

```
# Continua...
if test_is_flag:
    print("DEBUG: building individual IS sets...")

    n_IS = 0

    for c in df_val.columns:
        if 'err_' in c:
            n_IS += 1

    for i_NS in range(0,n_IS):
        print("DEBUG: building individual IS set {}...".format(i_NS))
        current_label = 'err_ds_' + str(i_NS)
        IS_dataset = df_test.iloc[:,0:ninput]

        for feature_number in range(len(feature_extension.iloc[i_NS])):
            current_key = feature_extension.iloc[i_NS].keys().values[
 feature_number]
            IS_dataset[current_key] = feature_extension.iloc[i_NS].get(
 current_key)

        IS_dataset = IS_dataset.join(df_test[current_label])
        IS_dataset.columns = ['err' if x==current_label else x for x in
 IS_dataset.columns]
        # Appiattimento degli errori ad  flatten_value
        IS_dataset['err'] = IS_dataset['err'].mask(IS_dataset['err'] >
 flatten_value, flatten_value, inplace=False)
        separate_IS_datasets.append(IS_dataset)
        print("DEBUG: individual IS set {} DONE.".format(i_NS))
        IS_dataset.to_csv("./pr_sets/{}_{}_test_set_IS_{}.csv".format(
 benchmark, i, i_NS), index=False)

    print("DEBUG: individual IS sets DONE.")

if test_flag:
    print("DEBUG: building test set...")

    n_IS = 0
    whole_test_set = pd.DataFrame()

    errs_cols_test = []
    for c in df_test.columns:
        if 'err_' in c:
            n_IS += 1

    for c in df_test.columns:
        if 'err_' in c:
            errs_cols_test.append(c)
            df_test[c] = df_test[c].mask(df_test[c] > flatten_value,
 flatten_value, inplace=False)
    df_test['err_mean'] = np.mean(df_test[errs_cols_test], axis=1)

    for i_NS in range(0,n_IS):
        partial_df_test = df_test.loc[:,~df_test.columns.str.startswith('err_'
)]
        for feature_number in range(len(feature_extension.iloc[i_NS])):
                current_key = feature_extension.iloc[i_NS].keys().values[
 feature_number]
                partial_df_test[current_key] = feature_extension.iloc[i_NS].
 get(current_key)
        partial_df_test['err'] = df_test['err_mean']
        whole_test_set = pd.concat([whole_test_set, partial_df_test], axis=0)

    print (whole_test_set)

    print("DEBUG: test set DONE.")
    whole_test_set.to_csv("./pr_sets/{}_{}_test_set.csv".format(benchmark, i),
  index=False)

# ... normalizzazione dati

return x_train, y_train, x_test, y_test, x_val, y_val,
 result_separate_IS_datasets, scaler
```

Si noti innanzitutto come vi sia un semplice sistema di caching dei dati processati, che permetta di evitare la rilettura e la rimanipolazione ad ogni esecuzione dello script di modello; questo aspetto è stato implementato in quanto, com'è possibile notare dalla complessità del codice, l'esecuzione di queste operazioni su dataset sufficientemente grandi come quelli che abbiamo a disposizione è molto costosa a livello temporale.

Si faccia poi particolare attenzione a come viene gestito il test set: così come per il modello *ML_mean*, si vogliono ottenere un test set con i valori medi dell'errore e altri 30 test set individuali con i valori di errore corrispondenti a quelli ottenuti per ogni Input Set distinto; la differenza fondamentale rispetto al precedente modello è che non venivano considerate le feature estese, mentre in questo caso sì, ragion per cui questi nuovi dataset di test sono costruiti in maniera leggermente più complessa.
In particolare, mentre per ogni riga dei set individuali abbiamo la configurazione, l'estensione per un Input Set M e il relativo errore di M, per un totale di N righe (numero di configurazioni), per il test set medio abbiamo $N * M$ righe, dove ognuna di esse contiene la confgiurazione N-esima, l'estensione M-esima, e l'errore medio associato alla configurazione N. È stato necessario ottenere una struttura così composta per essere coerenti sia con le modalità di test utilizzate in *ML_mean*, sia con le modalità di test di un modello MLE.

Si noti infine la potenza della libreria *pandas*, che ci permette una gestione semplificata delle strutture dati in `DataFrame` e in particolare dei file in formato `csv`, il tutto utilizzando un numero di operazioni decisamente ridotto rispetto a quello che si avrebbe avuto effettuando una manipolazione di dati manuale.

4.6.2 Implementazione e tuning di *ML_MLE*

Come di consueto, sono qui riportate le parti salienti dell'implementazione del nuovo modello:

```
def regression_ML_MLE(benchmark, n_dense, node_mul, l_r, dec, epochs, batch_size):
    # ...
    n_samples, n_features = x_train.shape
    input_shape = (x_train.shape[1],)
    pred_model = Sequential()
    pred_model.add(Dense(n_features * 3, activation='relu',
       activity_regularizer=regularizers.l1(1e-5),
       input_shape=input_shape))

    for n in range(n_dense):
        pred_model.add(Dense(n_features * node_mul, activation='relu',
        activity_regularizer=regularizers.l1(1e-5)))

    pred_model.add(Dense(1 + 1, activation='linear'))

    adam = optimizers.Adam(lr=l_r, decay=dec)
    pred_model.compile(optimizer=adam, loss=neg_log_likelihood_loss,
                metrics=['mean_absolute_error', tf.keras.metrics.
     RootMeanSquaredError(name='rmse')])

    history = pred_model.fit(x_train, y_train,
            epochs=epochs, batch_size=batch_size, shuffle=True,
            validation_data=(x_val, y_val),
            verbose=True, callbacks=[early_stopping, reduce_lr, tensorboard,
     terminate_nan])

    # Test con test set medio
    predicted = pred_model.predict(x_test)
    actual = y_test
    MAE_test, RMSE_test = evaluate_predictions(predicted, actual, scaler, True)

    # Test con test set individuali
    metrics_results = []

    for i, element in enumerate(IS_datasets):
        print(len(IS_datasets))
        pr = pred_model.predict(element[0])
        ac = element[1]
        MAE_IS, RMSE_IS = evaluate_predictions(pr, ac, scaler, False)
        metrics_results.append([MAE_IS, RMSE_IS])
```

Come si può notare, il modello è stato implementato in maniera molto simile a *ML_mean*, utilizzando però ovviamente la funzione di loss MLE, e quindi dataset adeguatamente preprocessati; anche in questa occasione abbiamo gli stessi iperparametri dello scorso modello, tarati utilizzando il tuning manuale e selezionando la loro migliore combinazione per ogni benchmark tra 2000 varianti utilizzando *TensorBoard*.
Effettuato il tuning degli iperparametri, sono state trovate le seguenti migliori combinazioni per la creazione di modelli ottimali:

- *BlackScholes*:
  - Hidden Layer: 1
  - Numero neuroni per ogni hidden layer: $4 * num_features$;
  - Batch size: 128
  - Epoch: 103
  - Learning rate e decay: 0.005, 0.0005

- *convolution*:
  - Hidden Layer: 12
  - Numero neuroni per ogni hidden layer: $4 * num_features$;
  - Batch size: 32
  - Epoch: 21
  - Learning rate e decay: 0.01, 0.001
- *correlation*:
  - Hidden Layer: 4
  - Numero neuroni per ogni hidden layer: $8 * num_features$;
  - Batch size: 8
  - Epoch: 72
  - Learning rate e decay: 0.01, 0.0005
- *saxpy*:
  - Hidden Layer: 4
  - Numero neuroni per ogni hidden layer: $4 * num_features$;
  - Batch size: 8
  - Epoch: 97
  - Learning rate e decay: 0.001, 0.0005
- *dwt*:
  - Hidden Layer: 1
  - Numero neuroni per ogni hidden layer: $4 * num_features$;
  - Batch size: 64
  - Epoch: 71
  - Learning rate e decay: 0.01, 0.001
- *Jacobi*:
  - Hidden Layer: 8
  - Numero neuroni per ogni hidden layer: $2 * num_features$;
  - Batch size: 32
  - Epoch: 132
  - Learning rate e decay: 0.0005, 0.001

Come si può notare dai parametri trovati, possiamo ipotizzare che i modelli su cui andremo a fare gli esperimenti nel prossimo paragrafo avranno un tempo di allenamento decisamente maggiore rispetto a quelli trovati nel paragrafo precedente: questo è dovuto ad un maggiore numero di epoch necessarie per la convergenza del valore di loss al suo minimo, ma anche alla batch size (ovvero il numero di esempi consecutivi propagati durante ogni fase di discesa del gradiente, per evitare di propagare in una sola volta l'intero dataset e di saturare quindi la memoria RAM), che per molti dei benchmark è bassa (arrivando anche a 8). Questa ipotesi ha effettivamente trovato riscontro nel fatto che il tuning di Jacobi nella macchina di laboratorio del LIA è durato diverse settimane.

4.6.3 Esperimenti e risultati

Compilati i modelli derivati dalla fase di tuning degli iperparametri, questi sono stati testati con i $30 + 1$ test set, fornendo i seguenti risultati:

	Performance Test Set medio		Performance Test Set per gli Input Set			
Benchmark	MAE	RMSE	μ MAE	σ MAE	μ RMSE	σ RMSE
BlackScholes	0.138	0.241	0.180	0.005	0.283	0.007
convolution	0.026	0.105	0.039	0.001	0.153	0.005
correlation	0.082	0.142	0.235	0.055	0.388	0.067
dwt	0.034	0.081	0.409	0.067	0.554	0.054
Jacobi	0.053	0.192	0.059	0.006	0.222	0.014
saxpy	0.011	0.051	0.029	0.000	0.135	0.000
Media	0.057	0.135	0.158	0.022	0.289	0.024

Nel prossimo capitolo verranno messi a confronto tutti i risultati ottenuti durante la parte di progetto; quanto alla suddetta tabella, che verrà confrontata con i risultati di *ML_mean*, si osservano numeri di MAE e RMSE negli ordini rispettivamente di $0.01 - 0.14$ e $0.50 - 0.240$ per quanto riguarda il test set, cosa che indica performance più che discrete per la maggior parte dei benchmark, specie rispetto alle metriche (seppur parziali, calcolate sulla capacità di saper predirre valor medio) del regressore originale a cui erano stati aggiunti il filtraggio e le feature dal grafo.
Anticipiamo già che, come ci si aspettava purtroppo, le performance di questo modello che utilizza MLE, per quanto riguarda il test set, sono leggermente peggiori rispetto quelle di *ML_mean*, con il vantaggio però di avere una trasferibilità del modello migliore.
Come già detto, questi aspetti verranno però chiariti nel dettaglio, con misure quantitative e qualitative, nel capitolo successivo.

5 Risultati sperimentali

Una volta portata a termine la fase progettuale dell'elaborato, è certamente utile riassumere i risultati ottenuti, col fine di confrontarlii e trarre conclusioni sulla validità degli approcci utilizzati, quali:

1. Modello stocastico puro, allenato con MLE;
2. Arricchimento del precedente modello con filtraggio[1] del dataset e feature aggiuntive provenienti dal grafo delle dipendenze[2];
3. Esperimenti tentativi sull'utilizzo dell'autoencoder per la riduzione della dimensionalità degli Input Set;
4. *ML_mean*: performance e trasferibilità di un regressore per un punto semplice (errore medio)
5. *ML_MLE*: performance e trasferibilità di un modello stocastico con l'aggiunta di feature statistiche derivate dagli Input Set

Per effettuare tale riassunto, vengono di seguito riportate, messe a confronto e commentate le tabelle presenti in modo sparso nel capitolo di progetto, le quali (come già detto) riportano metriche qualitative per ogni approccio utilizzato all'interno della tesi, quali MAE, RMSE (e indice di correlazione di Pearson per i primi modelli che utilizzano MLE).

[1] Operazione tramite cui sono stati dai dataset utilizzati per il modello tutte le righe che presentavano, per una certa configurazione, errori più alti di una certa soglia, nel nostro caso uguale a 1.

[2] Aggiunta di feature ai dataset, sottoforma di numeri interi positivi o negativi ottenuti da disequazioni (che impongono relazioni tra variabili e le loro precisioni); tali disequazioni sono estratte dai grafi delle dipendenze (esistenti per ogni benchmark), ovvero strutture in cui ogni nodo è una variabile, e ogni linea orientata rappresenta una relazione tra la variabile di partenza e quella di arrivo.

Modello stocastico MLE puro, e aggiunta dei contributi del filtraggio e delle feature del grafo

Osserviamo la comparazione, testa a testa, delle quattro combinazioni di modello stocastico ottenute da queste metodologie.

	Dataset puro		Dataset filtrato		Dataset esteso		Dataset esteso e filtrato	
	MAE	RMSE	MAE	RMSE	MAE	RMSE	MAE	RMSE
BlackScholes	4.755	6.493	3.045	4.054	4.149	5.894	2.934	3.970
correlation	1.533	2.690	1.607	2.878	2.516	3.752	1.932	3.077
dwt	1.807	3.170	1.659	2.091	2.207	3.461	1.233	1.734
FWT	6.257	7.423	2.504	3.168	2.550	3.551	1.802	2.310
Jacobi	6.085	7.003	1.708	2.325	6.321	7.396	1.554	2.063
Media	4.091	5.356	2.105	2.903	3.549	4.811	1.891	2.631

Tabella 5.1: Performance aggregate per la predizione del valore medio.

	Indice di correlazione di Pearson			
	Dataset puro	Dataset filtrato	Dataset esteso	Dataset esteso e filtrato
BlackScholes	0.6161	0.3516	0.7106	0.3000
correlation	0.5998	0.2974	0.7133	0.3117
dwt	0.8680	0.8668	0.8878	0.9395
FWT	0.8541	0.7707	0.8747	0.7593
Jacobi	0.3272	0.0554	0.3482	0.0955
Media	0.6530	0.4684	0.6720	0.4812

Tabella 5.2: Performance aggregate per la predizione della varianza.

Si noti che per questi metodi le metriche sono di ordine molto più grande rispetto, ad esempio, a quelle presenti nelle tabella 5.4 e 5.3: ciò è dovuto al fatto che qui stiamo osservando valori denormalizzati, mentre per gli approcci successivi si è scelto di utilizzare metriche in forma normalizzata.

Come si può osservare, pare che il metodo che influisca maggiormente nei cambiamenti del comportamento del modello sia quello del filtraggio; infatti, è possibile notare come gli effetti del filtraggio ottenuti sulla media siano nettamente amplificati una volta combinati i due metodi; d'altra parte l'aggiunta di feature dà un contributo minimo, alzando leggermente gli indici di correlazione di Pearson della varianza.

In definitiva, sebbene non forniscano risultati eccellenti per la varianza, i nuovi modelli generati possono essere utilizzati per una predizione della media di discreta qualità. Sicuramente questa risulta essere una strada valida per la generalizzazione del modello per due motivi:

- Viene evidenziato come il rispetto dell'assunzione di distribuzione normale dei valori sia importantissimo per la riuscita di un modello che utilizza Maximum Likelihood Estimation; per questo motivo, l'utilizzo del filtraggio, in assenza di errori distribuiti su una gaussiana (situazione da cui, data la complessità del dominio del problema, ci si aspetta di partire), risulta essere fonte di miglioramenti non indifferenti;

- L'aggiunta di conoscenza del dominio al modello, in una situazione come quella del nostro problema, in cui la funzione che lega dati e target è non lineare (e talvolta non monotona come evidenziato dalla tabella 5.3 che riporta le percentuali medie rispetto agli Input Set di casi di non monotoni), "aiuta" la nostra rete neurale ad apprendere meglio tale relazione.

	BlackScholes	convolution	correlation	dwt	Jacobi	saxpy
Media Input Set	5.913%	0.539%	9.447%	0.286%	0.000%	0.437%

Tabella 5.3: Risultati aggregati dell'analisi sulla la quantità di relazioni di non monotonia riscontrate tra le configurazioni di ogni dataset, per ogni benchmark.

Esperimenti sulla fattibilità della compressione degli Input Set tramite un autoencoder

Essendo molto poco pratico (se non infattibile) cercare di aggiungere conoscenza del dominio al modello inserendo gli interi Input Set come feature all'interno dei dataset, si è scelto di provare a costruire un autoencoder per cercare di ottenere una loro rappresentazione a dimensionalità ridotta.
Già in partenza erano stati evidenziati due possibili punti di fallimento:

- Essendo gli Input Set generati in modo randomico, essi non possono essere compressi senza un errore di ricostruzione di un certo grado;
- La dimensione ottimale dello strato latente potrebbe essere ancora troppo grande per l'inserimento di tali feature nei dataset.

In poche parole, l'esito di tale esperimento è stata strettamente legata alla "qualità" del compromesso da attuare tra una situazione ideale (con errore di ricostruzione nullo e dimensione dello strato latente piccola) e una situazione "realistica" (dovuta ai due punti di fallimento evidenziati).

Una volta eseguito il processo di tuning della rete (solo per *dwt*) avevamo ottenuto il seguente set di iperparametri:

- *Numero di hidden layer prima e dopo lo strato latente*: 1
- *Numero di neuroni nello strato latente*: 1200
- *Numero di epoch*: 1000
- *Dropout rate*: 0.1

Come si può già notare, abbiamo un numero di neuroni nello strato latente pari a 1200: numero non molto basso, se si considera che ogni Input Set di *dwt* è composto da 3.328 valori. Inoltre, si sono osservati i seguenti errori di ricostruzione (valori medi calcolati su 10 esperimenti identici):

- *MAE*: 0.3217
- *RMSE*: 0.3969

Questi risultati non sono soddisfacenti e, confermando le ipotesi di fallimento teorizzate all'inizio del sottoparagrafo, ci hanno portato ad abbandonare l'idea di effettuare ulteriori esperimenti su questa strada.

ML_mean e ML_MLE a confronto

Scartata la possibilità di ottenere una rappresentazione a dimensionalità ridotta degli Input Set "grezzi" tramite un autoencoder, si è passati, con *ML_mean*, ad analizzare un aspetto differente del problema: la trasferibilità di un modello di regressione (costruito su una rete neurale che cerca di apprendere, con funzione di loss MAE, la relazione tra configurazione e un valore preciso, rappresentato dall'errore medio appreso direttamente dal dataset) da un test set che utilizza gli errori medi come target a molteplici test set che utilizzano gli errori rispettivamente dei vari Input Set come target.
Osserviamo i risultati ottenuti:

	Performance Test Set medio		Performance Test Set per gli Input Set			
Benchmark	MAE	RMSE	μ MAE	σ MAE	μ RMSE	σ RMSE
BlackScholes	0.060	0.180	0.728	0.003	0.834	0.002
convolution	0.025	0.101	0.025	0.001	0.100	0.006
correlation	0.049	0.115	0.315	0.017	0.479	0.019
dwt	0.009	0.031	0.120	0.144	0.238	0.145
Jacobi	0.053	0.205	0.920	0.013	0.938	0.011
saxpy	0.010	0.052	0.010	0.000	0.052	0.000
Media	0.034	0.114	0.353	0.030	0.440	0.030

Tabella 5.4: Performance di *ML_mean*.

Ciò che si evince da questo modello è una buona performance sul test set (calcolata a livello di intero modello, e non solo di valore medio); purtroppo, notiamo anche una trasferibilità del modello quasi assente (MAE e RMSE dei test set individuali si allontanano molto da quelle calcolate per il test set medio), portandoci ad una conclusione: un modello di regressione semplice, allenato con MAE a predirre il valore medio dell'errore come punto preciso e non come distribuzione, non è sufficiente per il nostro caso in quanto non generalizzabile.

Fatta questa riflessione parallela, si è pensato di tornare alla metodologia MLE, e in particolare si è allenato un nuovo modello stocastico utilizzando dataset arricchiti da feature aggiuntive, composte da metriche statistiche calcolate sui valori grezzi degli Input Set, con lo scopo di aggiungere nuova conoscenza del dominio.

Osserviamo i risultati ottenuti:

	Performance Test Set medio		Performance Test Set per gli Input Set			
Benchmark	MAE	RMSE	μ MAE	σ MAE	μ RMSE	σ RMSE
BlackScholes	0.138	0.241	0.180	0.005	0.283	0.007
convolution	0.026	0.105	0.039	0.001	0.153	0.005
correlation	0.082	0.142	0.235	0.055	0.388	0.067
dwt	0.034	0.081	0.409	0.067	0.554	0.054
Jacobi	0.053	0.192	0.059	0.006	0.222	0.014
saxpy	0.011	0.051	0.029	0.000	0.135	0.000
Media	0.057	0.135	0.158	0.022	0.289	0.024

Tabella 5.5: Performance di *ML_MLE*.

Da questi risultati si può notare come per la maggior parte dei benchmark le metriche per i test set individuali siano molto simili a quelle trovate per il test set medio.
Osserviamo adesso la tabella 5.6, che evidenzia le differenze percentuali per ogni metrica tra *ML_mean* e *ML_MLE*:

	Performance Test Set medio		Performance Test Set per gli Input Set			
Benchmark	MAE	RMSE	μ MAE	σ MAE	μ RMSE	σ RMSE
BlackScholes	130.00%	33.89%	-75.27%	66.67%	-66.07%	250.00%
convolution	4.00%	3.96%	56.00%	0.00%	53.00%	-16.67%
correlation	67.35%	23.48%	-25.40%	223.53%	-19.00%	252.63%
dwt	277.78%	161.29%	240.83%	-53.47%	132.77%	-62.76%
Jacobi	0.00%	-6.34%	-93.59%	-53.85%	-76.33%	27.27%
saxpy	10.00%	-1.92%	190.00%	0.00%	159.62%	0.00%
Media	67.65%	18.42%	-55.24%	-26.67%	-34.32%	-20.00%

Tabella 5.6: Differenze percentuali tra i due modelli, calcolate per ogni elemento della tabella tramite la formula $y = \frac{x_{MLE} - x_{mean}}{x_{mean}} * 100\%$.

Come ci si aspettava (a causa degli assunti parzialmente errati sulla distribuzione degli errori), purtroppo le performance di questo modello che utilizza MLE, per quanto riguarda il test set, sono leggermente peggiori rispetto quelle di ML mean. Si può però osservare un aspetto molto interessante: rispetto a *ML_mean*, in *ML_MLE* troviamo un grande miglioramento della trasferibilità del modello, probabilmente dovuta al contributo delle nuove feature statistiche aggiunte ai dataset.

Conclusioni

All'interno di questo elaborato di tesi sono stati implementati diversi modelli stocastici di machine learning, con diversa complessità, con lo scopo di approssimare al meglio la relazione non lineare che vi è tra numero di bit assegnato ad ogni variabile del benchmark e conseguente variazione dell'errore.

È innanzitutto stato necessario analizzare i dataset di partenza (contenenti esempi di configurazione con errore associato per ogni Input Set), alla ricerca di possibili comportamenti non monotoni dell'errore rispetto alla variazione delle configurazioni di bit. In effetti, seppure in una piccola percentuale, questi comportamenti esistono e non sono trascurabili: tutti gli esperimenti successivi hanno infatti tenuto in conto di tale problema (che rappresenta una sfida per l'apprendimento della relazione tra i dati da parte della rete neurale) nella valutazione della "bontà" dei risultati ottenuti.

A seguire, ci si è occupati di un primo modello di machine learning per la trattazione della questione principale dell'elaborato: trovare la relazione più generale possibile tra configurazioni di bit ed errori associati, evitando legami dovuti alla scelta dello specifico Input Set. Per fare ciò si è scelto di utilizzare non un semplice regressore, bensì un modello stocastico (allenato tramite Maximum Likelihood Estimation) che cercasse di apprendere una distribuzione statistica degli errori (totalmente caratterizzata da media e varianza) in base alla variazione delle configurazioni di bit.
I primi esperimenti su questo modello non sono stati pienamente soddisfacenti: mentre la predizione della media risultava essere discreta, la varianza era praticamente sempre inesatta. Il motivo per cui ciò accadeva era probabilmente riconducibile ad assunti errati formulati all'inizio della trattazione, e/o alla mancanza di informazioni sul dominio del problema.

Si è quindi deciso di andare a rappresentare graficamente la distribuzione dei valori degli errori dei dataset, permettendoci così di osservare che essi non sono distribuiti secondo una gaussiana (ipotesi necessaria per il corretto funzionamento del metodo MLE). Osservando che, per buona parte dei casi, la mancanza di una distribuzione gaussiana era dovuta alla presenza di configurazioni per cui l'errore era molto alto, si è deciso di provare a filtrare queste ultime dai dataset. I risultati, sebbene leggermente peggiorati per la varianza, sono certamente migliorati per quanto riguarda la capacità di predizione del valore medio.
A seguire, dai grafi delle dipendenze delle variabili di ogni benchmark (si veda il lavoro di tesi di F. Livi [7]), si sono ricavate delle feature aggiuntive per i nostri dataset che hanno effettivamente arricchito la conoscenza del dominio per il modello. Infatti, nonostante si siano osservati lievi peggioramenti per la predizione della media, si sono osservati anche dei miglioramenti non marginali per la predizione della varianza.
A conclusione di questa prima fase si è scelto di combinare i due approcci, ottenendo un modello stocastico allenato con dataset estesi e filtrati: dai risultati è stato possibile osservare come tra i due approcci, quello del filtraggio è il responsabile di ripercussioni (sia in positivo che in negativo) più evidenti sul modello puro.

Cercando di arricchire ulteriormente la conoscenza del dominio da parte del modello, si è voluto tentare di fornire una rappresentazione compressa degli Input Set (i quali hanno cardinalità che può arrivare fino a centinaia di migliaia di valori) tramite un autoencoder, con lo scopo di poter inserire questa nuova rappresentazione a dimensionalità ridotta come feature aggiuntiva dei nostri dataset. Purtroppo l'approccio si è rivelato fallimentare dopo aver tentato degli esperimenti con il benchmark *dwt*, il quale ha sempre reagito positivamente ai vari approcci tentati, e quindi non si sono effettuate ulteriori prove.

Nell'ultima fase del progetto si è tentata una strada alternativa: l'allenamento di un modello di regressione semplice (*ML_mean*) che trovasse una relazione tra configurazioni di bit ed errori medi (rispetto agli Input Set) associati; purtroppo questo modello, testato con buone performance su un test set con errori medi, è risultato essere *non trasferibile* rispetto ai test set con errori individuali per ogni Input Set.
Si è quindi tornati a ragionare in termini di MLE, alleando un ultimo modello stocastico (*ML_MLE*) che utilizzasse delle metriche statistiche calcolate sui valori grezzi degli Input Set come feature aggiuntive, per

proseguire sulla strada dell'arricchimento della conoscenza del dominio del problema. In effetti, questo approccio ha dato risultati soddisfacenti sia in termini di performance generali (sebbene leggermente peggiori rispetto quelle di *ML_mean*), sia soprattutto in termini di trasferibilità e quindi in termini di capacità di essere generalizzabile rispetto gli Input Set.

In conclusione abbiamo osservato come, nonostante gli ostacoli di partenza (quali non monotonia, e distribuzione degli errori non gaussiana), appare possibile (tramite le metodologie utilizzate) arrivare ad ottenere dei modelli stocastici generali rispetto agli Input Set che siano in grado di approssimare la relazione che vi è tra configurazione di bit di precisione per le variabili dei benchmark e conseguente errore riscontrato.
In particolare, per eventuali sviluppi futuri, potrebbe essere interessante analizzare la possibilità di combinare gli approcci, e quindi effettuare esperimenti su modelli MLE che applichino filtraggio ed estensione della conoscenza del dominio (con l'aggiunta di feature ottenute dai grafi delle dipendenze, e soprattutto dagli Input Set), magari esplorando nuove opzioni su come codificare le caratteristiche degli Input Set nel modello.

Bibliografia

[1] Mittal, S., *A Survey Of Techniques for Aproximate Computing* in "ACM Comput. Surv., Vol. a, No. b", Article 1, 2015

[2] Ammar Ben Khadra, M., *An Introduction To Aproximate Computing*, arXiv:1711.06115, 2017

[3] Tavaglini, G., Mach, S., Rossi, D., Marongiu, A., Benini, L., *A Transprecision Floating-Point Platform for Ultra-Low Power Computing*, arXiv:1711.10374, 2017

[4] Lombardi M., Milano M., Bartolini A., *Empirical Decision Model Learning*, 2016

[5] N.-M. Ho, E. Manogaran, W.-F. Wong, and A. Anoosheh, *Efficient floating point precision tuning for approximate computing* in "22nd Asia and South Pacific Design Automation Conference (ASP-DAC)", IEEE, 2017, pp. 63–68

[6] S. Graillat, F. Jezequel, R. Picot, F. Fevotte, and B. Lathuiliere, *Autotuning for floating-point precision with Discrete Stochastic Arithmetic*, 2016, [Online], Available https://hal.archives-ouvertes.fr/hal-01331917

[7] Livi F., *Supervised Learning with Graph Structured Data for Transprecision Computing*, Tesi Magistrale in Intelligent Systems, Alma Mater Studiorum, 2019

[8] Alasdair Newson, Andrés Almansa, Yann Gousseau, Saïd Ladjal. *Taking Apart Autoencoders: How do They Encode Geometric Shapes ?*. 2018. hal-01676326

[9] Ronan Collobert, Jason Weston, *A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning*, 2008

[10] Geoffrey Hinton et al., *Deep Neural Networks for Acoustic Modeling in Speech Recognition*, 2012

[11] Yosinski, Jason, et al. *How transferable are features in deep neural networks?*, Advances in neural information processing systems. 2014

[12] Olivas, Emilio Soria, et al., *Handbook of research on machine learning applications and trends: Algorithms, methods and techniques-2 volumes*, 2009

[13] Ciresan Dan, et al., *Flexible, High Performance Convolutional Neural Networks for Image Classification*, 2011

[14] Heaton Jeff, *Introduction to the Math of Neural Networks*, Heaton Research Inc, 2011

[15] Myung, In Jae, *Tutorial on maximum likelihood estimation*, Journal of mathematical Psychology 47.1 2003, pp. 90-100.

[16] *Windows Subsystem for Linux Documentation*, Microsoft, Available https://docs.microsoft.com/en-us/windows/wsl/about

[17] *Anaconda*, Anaconda Inc., Available https://www.anaconda.com

[18] *NumPy*, Available https://numpy.org

[19] *pandas*, Available https://pandas.pydata.org

[20] *scikit-learn*, Available https://scikit-learn.org

[21] *Matplotlib*, Available https://matplotlib.org

[22] *TensorFlow*, Available https://www.tensorflow.org

[23] *Apache Licence 2.0*, Apache Software Foundation, 2004, Available https://www.apache.org/licenses/LICENSE-2.0

[24] *TensorFlow Documentation: tf.Graph*, Available https://www.tensorflow.org/api_docs/python/tf/Graph

[25] *Keras Documentation*, Available https://keras.io

[26] *Keras Tuner Documentation*, Available https://keras-team.github.io/keras-tuner

[27] Snoek, Jasper, Hugo Larochelle, and Ryan P. Adams, *Practical bayesian optimization of machine learning algorithms*, Advances in neural information processing systems. 2012

[28] Li, Lisha, et al., *Hyperband: A novel bandit-based approach to hyperparameter optimization*, The Journal of Machine Learning Research 18.1 (2017): 6765-6816

[29] Bergstra, James, and Yoshua Bengio, *Random search for hyper-parameter optimization* Journal of machine learning research 13.Feb (2012): 281-305

[30] *TensorBoard*, Available https://www.tensorflow.org/tensorboard

[31] Srivastava, Nitish, et al., *Dropout: a simple way to prevent neural networks from overfitting.*, The journal of machine learning research 15.1 (2014): 1929-1958.

[32] *SciPy Documentation*, Available https://docs.scipy.org/doc/scipy/reference/