UNIVERSITÀ DEGLI STUDI DI MILANO

FACOLTÀ DI SCIENZE E TECNOLOGIE

Laurea in Informatica

Apprendimento e compressione di strutture dati indicizzate

Relatore:
Prof. Dario MALCHIODI
Correlatore:
Dr. Marco FRASCA

Tesi di Laurea di:
Paolo GALLI
Matricola: 893396

Anno Accademico 2019/2020

Indice

Introduzione

Negli ultimi anni è aumentato l'interesse per lo studio e per l'utilizzo di algoritmi per l'apprendimento automatico (machine learning). Tali procedure hanno la peculiarità di apprendere informazioni aggiuntive dai dati, continuando a migliorare le prestazioni in maniera adattiva con l'aumentare degli esempi presi in considerazione. Infatti, cambia l'approccio rispetto ai classici programmi: non è più il programmatore che attraverso il codice sorgente dà le istruzioni da eseguire alla macchina, ma la macchina stessa che, elaborando un insieme di dati (dataset) tramite algoritmi di machine learning, sviluppa una propria logica per svolgere il compito richiesto. L'impiego che ne viene fatto è vasto: dalla sicurezza dei dati al trading finanziario, dal campo sanitario all'elaborazione del linguaggio naturale. Con l'aumentare delle prestazioni dei processori e delle dimensioni della memoria di dispositivi mobili, questi algoritmi hanno cominciato a essere impiegati anche su dispositivi mobili e nell'IoT [1]; oggi infatti negli smartphone ci sono programmi che, facendo uso di tecniche di machine learning, riescono ad esempio a classificare immagini o a fungere da assistente interagendo tramite comandi vocali con l'utente. In questo elaborato verrà preso in esame un particolare strumento usato nel machine learning, noto come reti neurali artificiali. Tale strumento sarà applicato al problema della regressione.

[1] Internet Of Things: termine che indica l'esistenza di una varietà di oggetti (things) che, attraverso la rete (internet) sono capaci di interagire tra di loro e collaborare con altri oggetti per creare nuovi servizi/applicazioni [1].

Nel primo capitolo verrà spiegato cosa sia effettivamente una rete neurale, cominciando da alcuni accenni storici, fino a illustrarne l'architettura e le modalità di utilizzo. Un grande problema delle reti neurali è la quantità di memoria che viene richiesta per memorizzare le strutture dati occorrenti al loro funzionamento. A tal proposito vengono proposti due algoritmi di compressione, di cui si parlerà nel secondo capitolo, per rimediare a tale problema. L'interrogativo a cui questo elaborato si pone di rispondere è infatti quanto l'azione di compressione delle reti possa inficiare sull'accuratezza delle loro previsioni, e in merito a ciò il terzo capitolo riporta gli esiti di alcuni esperimenti di compressione effettuati su una rete che predice l'indice di inserimento di un elemento all'interno di una lista ordinata (problema del predecessore). La parte conclusiva è dedicata all'analisi dei risultati ottenuti negli esperimenti proposti nel capitolo precedente.

Capitolo 1

Le reti neurali artificiali

1.1 Definizione e origini storiche

L'intuizione di replicare l'attività neurale che avviene nel sistema nervoso centrale umano quando apprende risale agli inizi degli anni '40, quando venne teorizzato un primo modello di neurone da McCulloch e Pitts [2]. Il parallelo è molto stretto: così come il compito di un neurone è quello di ricevere, elaborare e ritrasmettere impulsi elettrici all'interno del sistema nervoso centrale, quello di un neurone artificiale è ricevere un certo dato in input e attivarsi o meno a seguito della sua elaborazione. È interessante notare che, a differenza di un classico software, la costruzione di un neurone artificiale non richiede la scrittura di un programma esplicito ma avviene a seguito di un processo assimilabile a quello dell'apprendimento [3]. Inoltre, continuando l'analogia con il sistema nervoso centrale, non ci si è limitati a usare un singolo neurone ma più neuroni e più strati di neuroni collegati tra loro: questo ha portato allo sviluppo delle reti neurali.

1.2 Il neurone artificiale

La Figura 1.1 illustra il classico modello del neurone artificiale, che è composto da:

- uno o più input x_1, x_2, ..., x_n;
- uno o più pesi w_1, w_2, ..., w_n;
- funzione d'attivazione f;
- un output.

In particolare un neurone può essere attivato in base alla propria funzione d'attivazione: questa può essere una funzione binaria, come nel caso di quella a scalini [2] che attiva o disattiva il neurone, o generare un certo grado d'attivazione, come nella funzione sigmoide [3] o ReLu, che verrà spiegata nel Paragrafo 3.2.4.

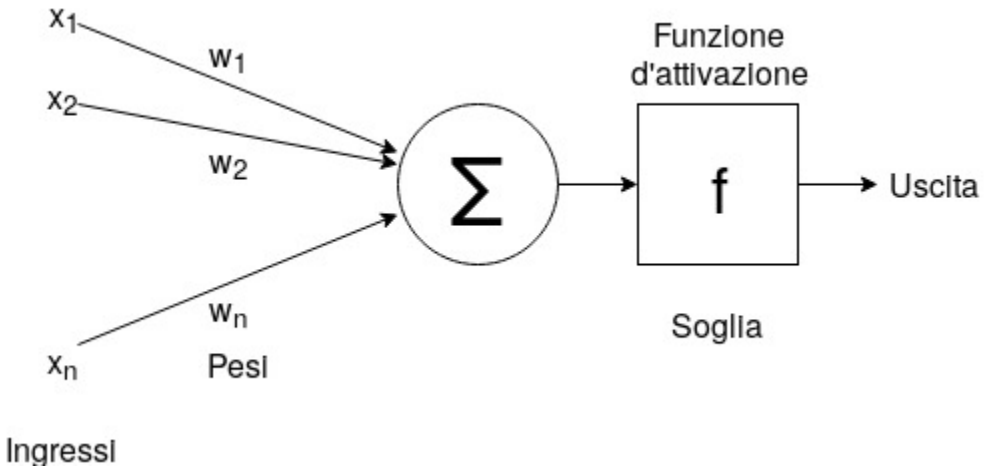


Figura 1.1: Modello di un neurone artificiale

[2] $\text{step}(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$

[3] $\text{sigmoid}(x) = \dfrac{1}{1 + e^{-x}}$

In particolare, il valore di uscita indicato nella Figura 1.2 viene calcolato moltiplicando ogni input (x_1, x_2, ..., x_n) per il corrispondente peso (w_1, w_2, ..., w_n), sommandone poi i risultati; il risultato costituirà l'argomento della funzione d'attivazione (f). Ad esempio, considerando l'immagine seguente:

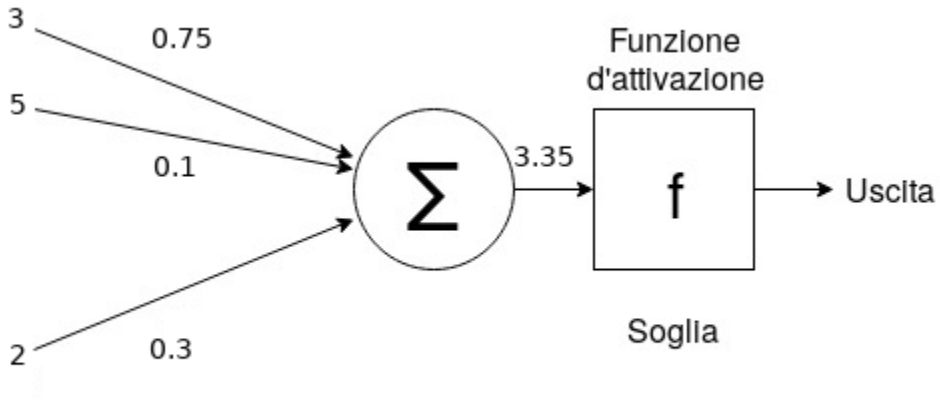


Figura 1.2: Modello di un neurone artificiale con esempi di valori

viene calcolato:

$$3 \times 0.75 + 5 \times 0.1 + 2 \times 0.3 = 3.35$$

che verrà successivamente utilizzato come argomento nella funzione d'attivazione

$$\text{Uscita} = f(3.35)$$

ottenendo così il valore Uscita. In linea generale, si esprime quindi l'output del neurone descritto in Figura 1.1 come $f(\mathbf{x}, \mathbf{w})$, dove $\mathbf{x} = (x_1, x_2, \ldots, x_n)$, $\mathbf{w} = (w_1, w_2, \ldots, w_n)$ e $\cdot$ è il prodotto interno allo spazio vettoriale.

1.3 Architettura di una rete neurale

Come detto precedentemente, una volta costruito un neurone artificiale, il passo successivo è stato quello di usare più neuroni in modo da formarne degli strati e connettere più strati tra di loro, in modo da permettere di affrontare problemi più complessi, compito impossibile per un neurone singolo. L'organizzazione di tale struttura può essere fatta in modi diversi, come ad esempio le reti di Hopfield [4] o quelle convoluzionali [5]; l'elaborato si concentrerà sull'organizzazione multistrato. Una rete, per essere in grado di apprendere teoricamente qualsiasi funzione, deve essere composta da almeno tre strati, illustrati in Figura 1.3, che prendono il nome di:

1. input;
2. hidden;
3. output.

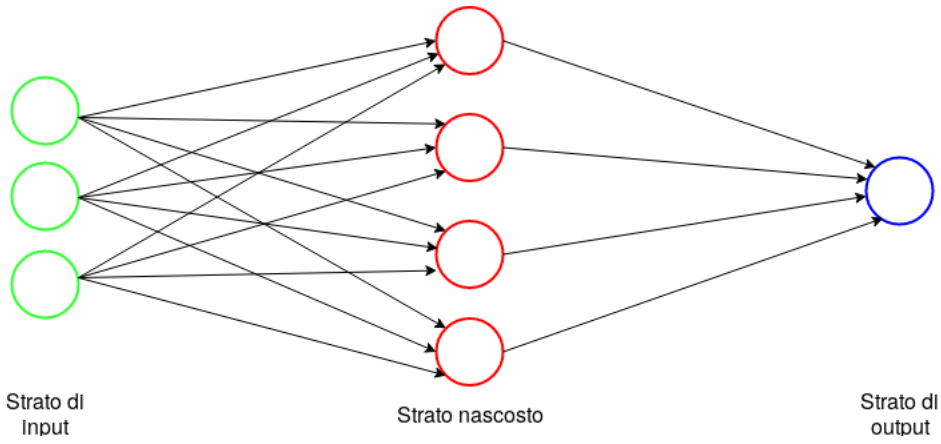


Figura 1.3: Architettura di una rete neurale multistrato

Il ruolo dello strato di input è quello di ricevere le feature da elaborare, lo strato nascosto si occuperà di costruire una rappresentazione intermedia delle feature, che

verrà poi elaborata dallo strato di output per fornire il risultato; va tuttavia specificato che lo strato di output può avere anche più neuroni (ad esempio nel caso di reti per classificazione), la Figura 1.3 è solo esemplificativa.

Inoltre, in base a come i neuroni dei vari strati sono collegati tra loro, si possono distinguere diverse configurazioni di rete come quella densamente connessa (fully connected), convoluzionale (convolutional) [4] usata principalmente per il riconoscimento di pattern in immagini o ancora quella ricorrente [5] . In questo elaborato ci si concentrerà sulla tipologia densamente connessa, in cui ogni neurone di uno strato sarà collegato a ogni neurone di quello successivo, come mostrato in Figura 1.3.

Come detto sopra, la rete è composta da strati; gli strati, a loro volta, sono un insieme di neuroni, ognuno dei quali ha associato un peso (w) per ogni collegamento in entrata e un bias (b). Questi devono essere prima inizializzati [7]; esistono diversi modi di farlo (ad esempio estraendo valori da una distribuzione uniforme o Gaussiana): negli esperimenti condotti sono stati inizializzati casualmente, seguendo una distribuzione uniforme in $[0, 1)$. Questi sono gli unici componenti che, una volta inizializzati, vengono modificati automaticamente dalla rete durante il training, sono detti parametri. Le caratteristiche rimanenti, come il numero di neuroni in uno strato hidden o il numero di strati, che rimangono tali durante tutte le fasi di addestramento del modello, sono dette *iperparametri*; l'approccio tipicamente usato per individuare gli iperparametri è quello empirico, come descritto nel Paragrafo 1.7.1.

[4] a differenza di quelle densamente connesse, i neuroni di uno strato sono collegati solo a una piccola parte dello strato precedente [5].

[5] la particolarità è che l'architettura prevede che siano presenti delle connessioni tra neuroni di uno strato e quello precedente, perdendo così la caratteristica di grafo aciclico; è usata quando è importante l'ordine dei dati, ad esempio nel riconoscimento del parlato o per riconoscere sequenze di stringhe o frame di un video [6].

1.4 Applicazioni

Le reti neurali vengono impiegate principalmente nell'attività di previsione (regressione) o classificazione. Nel primo caso rientrano, ad esempio, le previsioni di vendita: riuscire a prevedere il prezzo di un immobile in base a fattori che lo descrivono (i metri quadrati, il numero di stanze, la posizione centrale o meno in un centro abitato, ecc...) o prevedere l'andamento di un titolo azionario in borsa. Altro problema è quello invece della classificazione: si richiede in questo caso che la rete, dato un oggetto, ne restituisca la classe di appartenenza; tipico esempio è quello della classificazione di numeri scritti a mano o, data una serie di immagini, riconoscere quelle in cui è presente un particolare soggetto. In questo elaborato si andrà ad affrontare il problema legato alla regressione.

1.5 Esecuzione di una rete feed-forward

Le connessioni tra i neuroni di uno strato e quelli del successivo sono monodirezionali, configurazione che porta ad avere un grafo diretto e aciclico. Quindi il flusso di dati che scorre nella rete viaggia dallo strato di input a quello di output senza che ci siano cicli, il che implica anche che non ci siano interazioni tra neuroni dello stesso strato e che il tempo necessario per far scorrere i dati da input a output sia costante, dato che i vari neuroni elaborano parallelamente i segnali che ricevono.

1.5.1 Il calcolo tra strati

Nell'implementazione pratica i pesi delle connessioni vengono rappresentati da matrici e i pesi dei bias da vettori. Siano:

- $\mathbf{W_{ih}}$ la matrice dei pesi di dimensione $r \times c$, dove r è il numero di neuroni in input, c il numero di neuroni dello strato hidden e i pedici i e h stanno, rispet-

tivamente, per input e hidden; ogni componente di questa matrice rappresenta il peso di una connessione tra un neurone dello strato input e uno dello strato hidden;

- $\mathbf{W_{ho}}$ analogamente al punto precedente, la matrice dei pesi che rappresenta le connessioni tra gli strati nascosto e di output;
- $\mathbf{b_h}$ il vettore colonna dei bias dello strato hidden di dimensione $r \times 1$ dove r è il numero di neuroni dello strato hidden; ogni componente rappresenta il bias di uno dei neuroni dello strato hidden;
- $\mathbf{b_o}$ analogamente al punto precedente, il vettore dei bias dello strato di output;
- x il vettore di input;
- f la funzione d'attivazione scelta per lo strato.

Per eseguire l'algoritmo di feed-forward occorre calcolare

$$\alpha = f\left(x \cdot \mathbf{W_{ih}} + \mathbf{b_h}\right),$$

procedendo poi analogamente con lo strato successivo

$$\text{prediction} = f\left(\alpha \cdot \mathbf{W_{ho}} + \mathbf{b_o}\right).$$

Da notare che, in entrambi i casi sopra, la funzione d'attivazione viene applicata a ogni componente del vettore. Inoltre, per semplicità, nell'esempio è stata usata la stessa f per ogni strato; nulla vieta però di applicare una funzione di attivazione diversa per ogni strato. Le reti considerate in questo lavoro vengono utilizzate per calcolare una regressione, e dunque hanno un solo neurone di output: prediction sarà quindi un valore scalare che rappresenterà la previsione della rete in base all'input ricevuto.

1.6 Addestramento di una rete neurale

Per continuare il parallelo con il sistema nervoso centrale umano, così come per l'attività di apprendimento occorre studiare, anche una rete neurale artificiale ha bisogno di "studiare": questa fase viene detta fase di addestramento (training). Una volta selezionato il dataset contenente un certo numero di esempi [6], si procede a dividere tale insieme in due sottoinsiemi: per fare questo, negli esperimenti che verranno presentati più avanti, si è utilizzato un holdout di 80% - 20%, rispettivamente per training set, che verrà usato per l'attività di apprendimento, e per il testing set, per valutare la bontà della rete su esempi che non ha ancora visto. Una soluzione alternativa sarebbe stata quella di fare una cross-validation, concetto spiegato nel Paragrafo 1.8.1.

1.6.1 Fase di training

In questa fase le feature degli esempi nel training set vengono copiate nello strato di input della rete. Utilizzando la procedura descritta nel Paragrafo 1.5.1, la rete propaga il risultato fino al neurone di output. È in questo momento che la rete impara confrontando la previsione con l'effettivo target: in base alla loro differenza, la rete stessa correggerà i pesi degli strati usando un algoritmo noto come back-propagation (spiegato nel Paragrafo 1.7), con l'obiettivo di minimizzare localmente una funzione di perdita (loss function) [7]. L'operazione di aggiornamento dei pesi viene in realtà eseguita solo dopo che un certo numero di esempi è stato mostrato alla rete: tale numero prende il nome di batch size. Normalmente è necessario effettuare vari cicli di aggiornamento che richiedono di mostrare gli esempi del training set più volte, questo concetto prende il nome di epoca, che indica quante volte la rete ha visto

[6] un esempio è rappresentato da un insieme di feature e un target.

[7] in questo caso è la funzione legata all'errore di predizione fatto dalla rete sui dati del training set.

ogni esempio. Se per esempio si avessero 200 esempi nel training set e un batch size pari a 10, ciò significherebbe che l'aggiornamento dei pesi avverrebbe ogni volta che sono stati presentati alla rete 10 esempi, e quindi un'epoca corrisponderebbe a $200 \div 10 = 20$ aggiornamenti. Ipotizzando che l'intero processo di addestramento richieda cento epoche, si ottiene quindi che l'aggiornamento dei pesi è stato effettuato $20 \times 100 = 2000$ volte.

Inoltre esistono diversi criteri per stabilire quando terminare l'apprendimento; in particolare negli esperimenti condotti in questo elaborato ne sono stati usati due basati su:

- epoche: il processo di apprendimento prosegue fino al raggiungimento del numero di epoche predefinito;
- valore della loss function: vengono fissate due soglie, s_1 e s_2; l'apprendimento termina quando la loss function non supera s_1 per s_2 volte consecutive.

1.6.2 Fase di testing

Terminata la fase di training occorre verificare il comportamento della rete su dati nuovi rispetto a quelli usati per addestrarla. Questo permette di valutare la reale capacità di previsione della rete e di valutare se durante l'addestramento si sia verificato uno dei problemi descritti di seguito.

- Si parla di *overfitting* quando il modello si adatta così tanto agli esempi del training set da perdere la capacità di generalizzare su nuovi esempi. Sintomo dell'overfitting è un'alta accuratezza durante la fase di training e una scarsa accuratezza in fase di testing; la rete ha imparato "a memoria" i casi di training in modo tale da non riconoscerne di nuovi;

- l'*underfitting* è l'opposto del caso precedente. Si verifica quando il modello non riesce a imparare la relazione tra feature e target, risultando in previsioni poco accurate.

1.7 Algoritmo di back-propagation

L'apprendimento consiste nel determinare i pesi delle connessioni e i bias che permettono alla rete di fare predizioni accurate sul training set. In altre parole bisogna minimizzare l'errore quadratico medio di predizione, che è la loss function citata nel Paragrafo 1.6.1. Il numero di argomenti della funzione da ottimizzare è molto elevato, e inoltre questa funzione è non-lineare, quindi non si possono utilizzare le tecniche classiche di ottimizzazione. La soluzione a questo problema consiste nell'utilizzare una tecnica locale e iterativa, nota come back-propagation [8], che regola i pesi e i bias associati ai collegamenti. Quello che permette di fare è legare l'errore ottenuto in output da una rete neurale a un algoritmo di ottimizzazione locale di una funzione di errore, in questo caso l'errore quadratico medio di predizione degli esempi del training set. L'algoritmo utilizzato è quello del gradiente discendente che consiste nel modificare i parametri della rete sottraendo loro una frazione del gradiente [8]. Per l'aggiornamento del peso viene usata la seguente formula:

$$W'_x = W_x - \eta \left(\frac{\partial \text{Loss}}{\partial W_x} \right)$$

dove W'_x è il nuovo valore del peso, W_x il vecchio peso, η il learning rate [9] e $\frac{\partial \text{Loss}}{\partial W_x}$ la derivata parziale dell'errore rispetto ai pesi.

[8] data una funzione f e i suoi parametri x e y, il gradiente della funzione $f(x, y)$, indicato con $\nabla f(x, y)$, è dato dal vettore $[\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}]$.

[9] valore solitamente vicino allo 0; è il grado con cui la rete adatta i propri pesi, più è alto il learning rate e più impatto avrà l'errore propagato e viceversa.

Una volta fatto questo calcolo con la matrice dei pesi strato hidden/output, si fa la stessa cosa per quella input/hidden. Il procedimento descritto viene iterato per tutta la durata dell'allenamento della rete al termine della fase della visione di un intero batch.

Un problema legato a questo algoritmo è che è possibile rimanere bloccati in un punto di minimo locale, dato che l'algoritmo di ottimizzazione usato è locale. Questo succede quando la funzione presenta un punto di minimo locale in un intorno sufficientemente grande da impedire ai passi compiuti nella discesa del gradiente di spostarsi da quel bacino. Non sempre però è un grave problema, può accadere ad esempio che minimo locale sia poco peggiore di un minimo globale, come in Figura 1.4. Per evitare di rimanere bloccati su un minimo locale si possono provare diverse soluzioni come aumentare il valore del learning rate, incrementare il numero di strati nascosti o ancora provare diverse funzioni d'attivazione o algoritmi di ottimizzazione diversi dalla discesa del gradiente [9].

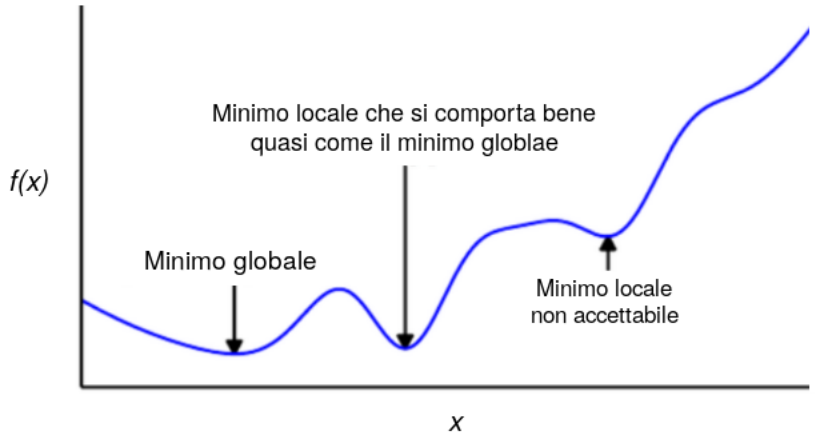


Figura 1.4: Diversi punti di minimo in una funzione

1.7.1 Iperparametri

Gli iperparametri, di cui si è già accennato nel Paragrafo 1.3, sono caratterizzati dal fatto che il loro valore viene impostato prima che l'addestramento abbia inizio e che non cambiano nel corso dello stesso. Negli esperimenti presentati in questo elaborato, questi valori vengono selezionati empiricamente attraverso model selection (vedi Paragrafo 1.8). In questa categoria rientrano:

- learning rate: indica il grado con cui la rete modifica i propri pesi durante la fase di back-propagation;
- numero di epoche: indica il numero di volte che la rete ha visto tutti gli esempi del training set;
- batch size: indica dopo quanti esempi mostrati alla rete si aggiornano i pesi;
- numero di neuroni per strato nascosto [10]: indica il numero di neuroni per singolo strato;
- funzione d'attivazione: funzione che determina l'intensità d'attivazione del singolo neurone.

Ogni volta che in questo elaborato verranno introdotti nuovi esperimenti ne verranno anche indicati gli iperparametri relativi.

1.8 Model selection

Come indicato nel Paragrafo 1.7.1, per trovare il valore degli iperparametri bisogna basarsi su prove empiriche. In particolare, negli esperimenti condotti sono stati

[10] in questo elaborato verranno usate reti neurali con zero, uno e due strati nascosti: per casi più complessi si può arrivare ad un numero maggiore di strati nascosti.

fissati a priori i valori per numero di epoche, batch size, numero di strati nascosti, learning rate e funzione d'attivazione. Il numero di neuroni dello strato nascosto è stato dimensionato usando la tecnica di model selection e ne sono state stimate le prestazioni usando uno la cross-validation che verrà spiegata nel prossimo paragrafo.

1.8.1 Cross validation

Dopo aver diviso il training set dal testing set, viene utilizzato esclusivamente il primo per la cross-validation. In particolare questo viene partizionato in k sottoinsiemi (fold [11]) di cui uno di questi prenderà il nome di validation set e gli altri di training set. Per ogni valore possibile dell'iperparametro (o per ogni possibile configurazione se ci sono più iperparametri) vengono ripetuti k processi di addestramento, ogni volta escludendo dal training set uno dei fold e utilizzando quet'ultimo per calcolare l'errore; si ottengono così k errori di cui si calcola la media.

Se per esempio il training set venisse suddviso in tre fold, le permutazioni prese in considerazione saranno quelle descritte nella Figura 1.5.

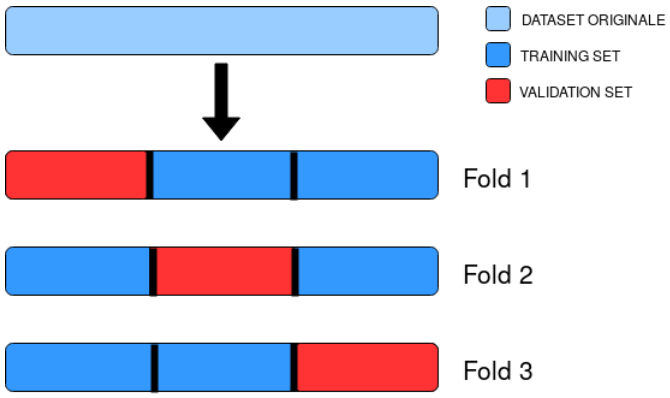


Figura 1.5: Divisione di un dataset in 3 fold

[11] l'idea è quella di suddividere gli m esempi in k gruppi di grandezza approssimativamente uguale, per semplicità si è deciso di mettere i primi $\frac{m}{k}$ elementi del training set nel primo fold e così via.

Viene infine scelto il valore dell'iperparametro (o degli iperparametri) che corrisponde all'errore medio minore, per poi riaddestrare la rete e valutare la bontà di generalizzazione di quest'ultima utilizzando il test set.

Di seguito è presentato l'algoritmo di cross validation in pseudocodice; per semplicità se ne presenta una versione in cui si fa cross validation su un solo iperparametro, ma lo stesso algoritmo può essere usato per più iperparametri.

Algoritmo 1 Cross Validation

input:
$H \leftarrow$ *insieme dei valori di un iperparametro*
$S \leftarrow$ *training set*
$K \leftarrow$ *numero di fold*
procedura:
$L[h] \leftarrow 0 \; \forall \; h \; \in \; H$

for $h \in H$ **do**
    for $k \in \{1, \ldots, K\}$ **do**
        $s_k \leftarrow$ *k-esimo fold di* S
        alleno la rete su $S \setminus s_k$
        valuto la rete su s_k
        $L[h] \leftarrow L[h] +$ *valore di loss per il k-esimo fold per h-esimo iperparametro*
    $L[h] \leftarrow L[h] \div K$
$h_{best} \leftarrow arg\_min(L[h] \, \forall \, h \in H)$
return h_{best}

Capitolo 2

Algoritmi di compressione per reti neurali

2.1 Compressione di una rete

Uno degli svantaggi più evidenti delle reti neurali è lo spazio che queste occupano in memoria e i tempi elevati per il loro addestramento. Questo aspetto ha assunto un ruolo sempre più importante con il diffondersi di macchine con ridotte capacità di computazione e di memoria e in generale dell'IoT. Per ridurre questo impatto negativo si stanno studiando diverse soluzioni tra cui comprimere le strutture dati alla base di una rete neurale in modo da ridurne le dimensioni in memoria intaccandone il meno possibile l'accuratezza. Questo elaborato si prefigge di verificare l'efficacia di due tecniche di compressione note come "pruning" e "weight sharing" su una rete neurale feed-forward utilizzata per problemi di regressione.

2.2 Pruning

L'idea di fondo del pruning è semplice: ogni collegamento tra neuroni ha associato un peso che ne determina l'importanza (avrà letteralmente un peso maggiore nell'influenzare il risultato finale della rete quando il valore della connessione è molto alto o molto basso); da qui l'intuizione per cui, ignorando i collegamenti con peso tendente allo zero, non si altera significativamente il risultato della rete che si otterrebbe considerandoli [10]. Questo risultato si può ottenere azzerando tali connessioni, ossia mettendo a zero il valore nella matrice dei pesi relativo alla connessione che si intende eliminare. Facendo così però si ottiene solo una parte del risultato che si vuole raggiungere usando il pruning; essendo il valore a zero si potrà verificare l'accuratezza della rete prunata di una percentuale di connessioni, ma non si godrà della riduzione di memoria, in quanto la matrice avrà lo stesso numero di elementi e quindi bisogno dello stesso numero di byte in memoria. Per questo viene in aiuto una particolare struttura dati adatta alla rappresentazione di matrici sparse [12]. Due scelte possibili per la struttura dati che accoglie la matrice sparsa sono indicate di seguito.

- ***compressed sparse row*** [11]: vengono usati 3 vettori contenenti rispettivamente i valori non nulli della matrice letti riga per riga, gli indici di colonna e i puntatori ai primi valori non nulli delle righe descritti nel modo seguente:

$$ptr = \begin{cases} ptr[0] = 0 \\ ptr[i] = prt[i-1] - nz \end{cases}$$

  dove nz è il numero di elementi non nulli nella riga $i-1$;

- ***compressed sparse column***: simile alla precedente con la differenza che nel primo vettore i valori vengono letti per colonna, vengono memorizzati gli indici di riga per ogni valore e i relativi puntatori alle colonne.

[12] per matrice sparsa si intende una matrice con un numero consistente di valori nulli.

Tuttavia questa operazione non è del tutto gratuita: l'operazione di moltiplicazione tra matrice e vettore con queste nuove strutture dati compresse è più lenta. Ne viene proposta l'implementazione in pseudocodice:

Algoritmo 2 Moltiplicazione CSC per vettore

1: *Siano*:
2: $val \leftarrow$ *vettore contenente i valori non nulli*
3: $col \leftarrow$ *vettore contenente gli indici di colonna*
4: $row\_ptr \leftarrow$ *vettore contenente i puntatori alle righe*
5: $x \leftarrow$ *vettore da moltiplicare*
6: $N \leftarrow$ *numero di colonne del vettore* x
7: $y \leftarrow$ *vettore risultate*
8: **for** $i \in N$ **do**
9: **for** $j \in \{row\_ptr[i], \ldots, row\_ptr[i+1]\}$ **do**
10: $y[i] \leftarrow y[i] + val[j] \times x[col[j]]$

2.2.1 Tasso di compressione

Il tasso di compressione ottenuto mediante pruning si può calcolare notando che quello che cambia è il numero di connessioni che vengono mantenute rispetto a quelle della rete originale. Denotando quindi con τ la soglia utilizzata [13], s_c lo spazio occupato dalla rete compressa composta dai pesi $\{w_{ij} \mid w_{ij} \geq \tau\}$ e s lo spazio occupato dalla rete originale, la percentuale di memoria risparmiata può essere espressa tramite l'equazione

$$\text{ratio} = \frac{s_c}{s} \tag{2.1}$$

Tuttavia, come si vedrà alla fine del Paragrafo 3.3.4, questo è un risultato teorico che non tiene conto del variare della struttura dati utilizzata. Occorre infatti notare che al di sotto di una certa percentuale di pruning l'uso di una struttura dati, quale la CSR, non è conveniente in termini di memoria occupata. D'altronde, altre strutture dati più efficienti per memorizzare matrici sparse potrebbero essere adottate; si lascia però questo approfondimento a studi futuri.

[13] In questo elaborato la soglia per scartare i pesi è stata calcolata come n-esimo percentile rispetto alla distribuzione dei pesi nella matrice, per $n \in \{10, \ldots, 90\}$ (quindi per $n = 10$ verranno scartate il 10% circa delle connessioni tra due strati comunicanti e così via).

2.3 Weight Sharing

La tecnica del weight sharing prende spunto dal flyweight pattern [12]: questo è uno strumento di ingengeria del software che permette di ottimizzare la memoria utilizzata facendo condividere un oggetto usato da più istanze piuttosto che crearne più copie. Un esempio concreto potrebbe essere quello riportato nel testo citato sopra, ossia quello di un editor di testo. Occorre distinguere però i due stati dell'oggetto condiviso, quello *interno*, che fa riferimento alle qualità intrinseche dell'oggetto indipendenti dall'utilizzo che ne verrà fatto, e quello *esterno*, che fa riferimento a quelle caratteristiche dipendenti dal contesto che possono mutare senza che muti l'oggetto condiviso. Tornando all'esempio dell'editor di testo, lo stato interno dell'oggetto condiviso è il carattere, mentre quello esterno la posizione all'interno del testo. Una volta creato un oggetto che raggruppi tutte le lettere dell'alfabeto, occorre interrogare tale oggetto condiviso per avere un riferimento alla lettera da inserire nel testo, operazione che chiede meno spazio rispetto alla creazione di un nuovo oggetto carattere ogni volta.

Nel caso pratico di una rete neurale l'oggetto condiviso è un array e le lettere dell'alfabeto sono i *centroidi*. Il centroide è un valore che rappresenta alcuni valori simili delle matrici dei pesi che vengono raggruppati in questo singolo valore. Questo valore viene usato in sostituzione ai pesi raggruppati per le operazioni matriciali. Come per l'esempio dell'editor di testo anche qui si possono distinguere uno stato interno, il valore numerico del centroide, e uno stato esterno, la posizione all'interno della matrice, o in altre parole il peso della connessione. Per stabilire i centroidi si possono usare algoritmi di clustering; in particolare quello usato in questo elaborato è K-Means [13], che verrà spiegato nel dettaglio nel Paragrafo 2.3.2.

2.3.1 Tasso di compressione

Come strutture dati sono stati utilizzati un vettore per ogni strato in cui vengono memorizzati i centroidi usando valori float a 32 bit e una matrice in cui memorizzare l'indice del centroide relativo; gli indici sono rappresentati da interi a 16 bit se la dimensione della matrice è superiore a 256, altrimenti da interi a 8 bit.
Il tasso di compressione, η, ottenuto tramite la tecnica del weight sharing è espresso dalla seguente formula:

$$\eta = \frac{nm \times f(n,m) + 32k}{32 \times nm}$$

$$\text{dove} \quad \begin{array}{l} k = \text{numero di centroidi,} \\ n, m = \text{dimensione delle matrici} \end{array}, f(n,m) = \begin{cases} 16 & nm > 256 \\ 8 & \text{altrimenti} \end{cases}$$

Pertanto, il tasso di compressione per gli esperimenti trattati nell'elaborato, dopo aver fissato il numero di cluster [14] per strato e dopo averne trovato i centroidi, le matrici dei pesi sono state sostituite con le matrici di indici che puntano al relativo centroide.

[14] per cluster si intende uno degli insiemi di pesi che sono stati raggruppati: il centroide che li rappresenterà si ottiene calcolandone il valor medio.

2.3.2 Algoritmo K-Means

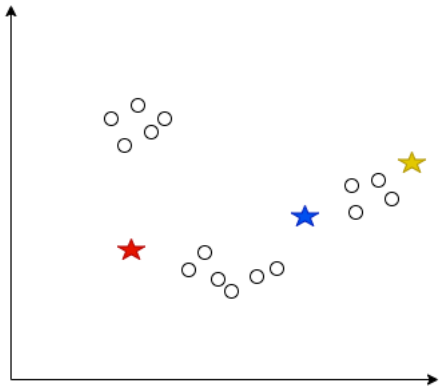
K-Means è un algoritmo di clustering iterativo. Si supponga di avere dei punti rappresentati su un piano cartesiano; l'obiettivo dell'algoritmo è quello di trovare dei sottogruppi di punti omogenei rispetto a una particolare misura, in questo caso la distanza euclidea [14]. Esistono diverse implementazioni di questo algoritmo, tra cui l'euristica di Hartigan [15], quella di Lloyd [16]; in questo elaborato viene usata quest'ultima, che in particolare prevede i seguenti passi:

1. vengono scelti k centroidi casualmente, dove k è il numero di cluster che si vogliono creare;
2. ogni osservazione viene assegnata al centroide in base alla minor distanza euclidea;
3. i centroidi vengono aggiornati in base alla media dei valori rientranti nel cluster;
4. se i centroidi sono stati aggiornati si ripete iterativamente dal punto 2, altrimenti l'algoritmo ha trovato i k centroidi.

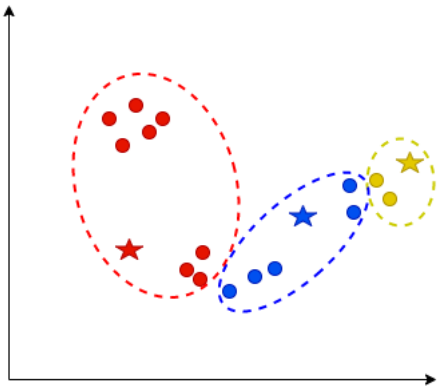
L'algoritmo usato è quello della libreria sklearn, MiniBatchKMeans [17] [15], una versione ottimizzata della precedente euristica di Lloyd che permette di lavorare, a ogni iterazione, su un sottoinsimene preso casualmente dei dati totali.

[15] https://scikit-learn.org/stable/modules/generated/sklearn.cluster.MiniBatchKMeans.html.

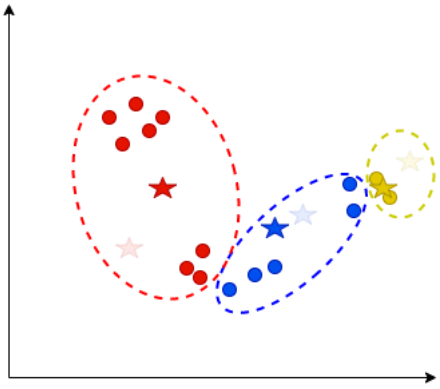
In Figura 2.1 viene presentato un esempio grafico in due dimensioni di come l'algoritmo K-Means viene eseguito in ogni singola iterazione; l'obiettivo è trovare tre cluster in cui raggruppare i punti.



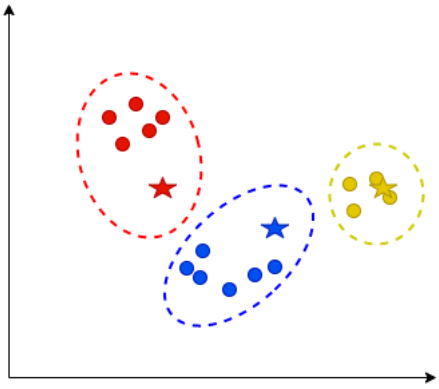
(a) Vengono scelti 3 centroidi casualmente rappresentati dalle stelline; un colore diverso per centroide. I pallini trasparenti sono i dati non ancora raggruppati



(b) Ad ogni osservazione assegno il centroide più vicino in base alla distanza euclidea, raggruppando i singoli cluster con un'ellissi del colore del relativo centroide



(c) Aggiorno i centroidi in base alla distanza media tra tutti i punti del cluster



(d) Ripeto quello fatto al punto (b) cambiando gli elementi all'interno dei cluster

Figura 2.1: Iterazioni algoritmo KMeans

Capitolo 3

Esperimenti

3.1 Dataset utilizzati

In questo paragrafo vengono descritti i dataset per i due problemi affrontati nella tesi: predizione del valore dei calciatori Fifa 2019, trattato nel Paragrafo seguente, e la predizione del predecessore in una sequenza ordinata, trattato nel Paragrafo 3.1.2. Si è deciso di cominciare usando un dataset con informazioni relative alle caratteristiche dei giocatori di calcio Fifa 2019 [16], con il problema di predire il valore di mercato del giocatore. Per tale problema è stata usata una rete neurale che sfrutta le API di Keras [17].

Una volta conclusa questa parte si è affrontato il problema della compressione per il problema del predecessore usando un dataset descritto nel dettaglio nel Paragrafo 3.1.2. Per quest'ultimo problema, invece, è stata creata una rete implementata senza l'utilizzo di librerie esterne al fine di poter sfruttare le nuove strutture dati introdotte per eseguire pruning, ovvero le matrici sparse, e weight sharing, con un vettore e una matrice per ogni strato.

[16] https://www.kaggle.com/karangadiya/fifa19, ultimo controllo 01/03/2020.
[17] https://keras.io/models/model/.

D'ora in poi si utlizzerà il termine *feature* per indicare una caratteristica che la rete usa durante la fase di training per predire il risultato e *target* per indicare il valore da predire in base alle feature in input.

3.1.1 Fifa 2019

Il primo dataset utilizzato è quello dei giocatori Fifa, composto da 89 feature e circa 18.2k esempi; il target da predire sarà il loro valore di mercato. Una volta eliminate le feature non necessarie all'esperimento (nome del giocatore, immagine del club, numero di maglia, ecc...) si è proceduto a convertire tutte le feature rappresentate come stringhe (come "Value", "Release clause", "Height", ecc...) in float e a rimuovere dal dataset tutti gli esempi il cui "Value" risultasse superiore o uguale al valore minimo tra gli outlier trovati [18], questo per evitare una prestazione peggiore in fase di training e testing [18]. Inoltre i nomi delle squadre di appartenenza sono stati inclusi come feature significative e codificati in valori interi.

Terminata questa fase preliminare si è deciso di considerare diverse alternative: la prima in cui vengono considerate tutte le feature a disposizione, la seconda in cui vengono selezionate solamente quelle il cui indice di correlazione [19] calcolato rispetto al target fosse maggiore di 0.55. È stata considerata anche una terza opzione: selezionare la metà delle feature tramite la Principal Component Analysis (PCA) che permette di ridurre la dimensione di un insieme di dati costituito da un elevato numero di variabili interconnesse, mantenendo il più possibile la variabilità presente in esso [19].

[18] tali outlier sono stati trovati selezionando tutti i valori standardizzati maggiori di una soglia impostata a 2.

[19] Calcolato tramite il metodo corr di pandas che utilizza il coefficiente di correlazione di Pearson $\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$ dove cov è la covarianza, σ_X la deviazione standard di X e σ_Y la deviazione standard di Y.

3.1.2 Problema del predecessore in sequenze ordinate

Se si vuole mantenere una lista di elementi ordinati per poi riuscire ad accedervi in poco tempo esistono delle strutture dati dedicate come ad esempio gli alberi binari bilanciati, tra cui i B-tree e i B+-tree [20]. Il tempo di accesso a queste strutture dati è dell'ordine di $\lceil O(log_2 m) \rceil$, dove n è il numero totale di elementi. Nonostante questo sia uno dei risultati migliori usando le strutture dati attualmente a disposizione, si è pensato di sfruttare le reti neurali per predire la posizione di un elemento all'interno di una sequenza ordinata di elementi con l'obiettivo di predire una posizione nella sequenza, data una chiave da cercare, più velocemente ma ammettendo anche un margine d'errore.

Formalmente, siano $x_1 \leq x_2 \leq \dots x_n$ gli n elementi di input, con $x_i \in \mathbb{N}$.

Il problema considerato è quindi il seguente: dato $x \in X = \{x_1, x_2, \dots, x_m\}$, restituire $\text{pred}(x) \in \{1, 2, \dots m\}$, posizione del predecessore di x nella sequenza. Il problema può essere risolto mediante una stima $\hat{F}$ della funzione di ripartizione empirica F rispetto a X. La stima di F viene ottenuta addestrando una rete sugli esempi $\{x_i, F(x_i)\}$, $i \in \{1, 2, \dots, m\}$; l'indice del predecessore di $x \in X$ può essere stimato mediante la formula seguente:

$$\text{pred(x)} = \lceil \hat{F}(x) \times m \rceil \tag{3.1}$$

Per questo problema sono stati utilizzati tre dataset, rispettivamente con 512, 8192 e 1048576 esempi. Gli elementi x_i sono stati rappresentati in codifica binaria. In questa versione si considereranno solamente valori di x già presenti nella lista e il problema non si occuperà di operazioni di aggiornamento della lista quali inserimento ed eliminazione.

3.2 Architettura delle reti

3.2.1 Rete Keras per analizzare il dataset di Fifa19

La prima rete considerata è stata costruita sfruttando delle API di Keras [20]. La rete è composta da uno strato di input con un numero di neuroni pari al numero di feature considerate, pari a $\lceil \log_2 m \rceil$, uno strato hidden con n neuroni e uno strato di output con un singolo neurone. Per gli iper-parametri si è deciso di fissare learning rate, numero di epoche, batch size e funzione d'attivazione; per trovare il numero di neuroni per il singolo strato hidden, invece, si è usata una tecnica di model selection, modalità descritta al Paragrafo 1.8.1.

3.2.2 NNPred: rete neurale per il problema del predecessore

Per il secondo problema sono state usate tre reti differenti:

- rete 1 con 0 strati hidden;
- rete 2 con 1 strato hidden di 256 neuroni;
- rete 3 con 2 strati hidden di 256 neuroni ciascuno.

Per tutte e tre le versioni gli iperparametri usati sono:

- momentum update[21]: 0.9;
- λ[22]: 10^{-5};
- batch size: 64;

[20] https://keras.io/models/model/

[21] valore che è compreso tra 0 e 1 che accelera la discesa del gradiente verso il minimo globale [21].

[22] usato per nella regolarizzazione l2, valore aggiunto alla loss function che forza pesi a scalare, penalizzando i pesi più grandi, riducendo così il problema di overfitting [22].

- epoche: 20000;
- stop function: vengono fissate due soglie, s_1 e s_2; l'apprendimento termina quando il valore della loss function non è inferiore a s_1 per s_2 volte consecutive; s_1 viene fissato inizialmente a 100 e poi aggiornato con il valore di loss più basso ottenuto e $s_2 = 4$.

Per la rete 1 è stato usato un learning rate di 5×10^{-4}, mentre per le altre due di 3×10^{-3}; questi valori sono stati trovati eseguendo un tuning sull'iperparametro notando che, per la rete senza strati nascosti, un valore più basso di learning ratge dava risultati con errore medio minore. I valori tra cui si è effettuato il tuning sono $\{5 \times 10^{-4}, 5 \times 10^{-3}, 3 \times 10^{-3}, 1 \times 10^{-3}, 1 \times 10^{-2}, 1 \times 10^{-1}\}$.

D'ora in poi ci si riferirà a queste reti come **NNX**, dove X indica il numero di strati.

3.2.3 Oragnizzazione dei dati

Per quanto riguarda il dataset Fifa, una volta mescolati gli esempi in maniera casuale, sono stati estratti l'80% degli esempi come train set e il restante 20% come test set. Ogni valore null all'interno dei set è stato poi sostituito con la media della relativa colonna e i valori scalati usando StandardScaler della libreria python sklearn [23] che utilizza la seguente formula:

$$z = \frac{x - u}{std} \tag{3.2}$$

dove u è la media del campione e std è la sua deviazione standard.

Per il secondo dataset vengono usati tutti gli esempi nel training, dopo essere stati permutati casualmente; i target invece vengono ricavati tramite la funzione numpy

[23] https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

linspace di python [24] che, dati due estremi di un intervallo e un intero x, restituisce un array di x valori ordinati equamente spaziati compresi tra gli estremi indicati.

3.2.4 Addrestramento

Per scegliere il giusto numero di neuroni per lo strato hidden della rete Keras al fine di predire il valore di mercato dei calciatori, si è proceduto con una cross-validation (Paragrafo 1.8.1). Di seguito vengono riportati gli iper-parametri scelti per la rete:

- funzione d'attivazione:
  - ReLu [25] per i neuroni dello strato hidden;
  - identità per il neurone d'output;
- loss function: errore quadratico medio [26];
- ottimizzatore: Adam (componente Keras) con
  - learning rate: 0.001;
  - beta [27]: 0.9;
  - decay [28]: 0.0;
- numero epoche: 100;

[24] https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html.

[25] $f(x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$

[26] $\frac{\sum_{i=1}^{n} (x_i - y_i)^2}{n}$ *dove x è l'i-esimo valore predetto, y è l'i-esimo target.*

[27] rappresenta il tasso esponenziale di decay per la stima del momenutm.

[28] quantità che viene utilizzata per abbassare il valore del learning rate, soprattutto quando si parte da un valore elevato. Questo fa in modo che, più ci si avvicini al minimo globale della funzione di loss, più i passi fatti verso tale punto sono piccoli e precisi.

- batch size: 64 esempi.

Il numero di fold è stato impostato a 5, provando per l'unico hidden layer un numero di neuroni $n \in \{10, 20, 30, \ldots, 200\}$.

3.3 Risultati

3.3.1 Risultati addestrando la rete mediante utility della libreria Keras per il problema di regressione sul valore di mercato dei giocatori

I risultati per ogni fold sono riportati in tabella 3.1, mentre i risultati dei test usando il numero di neuroni in corrispondeza dell'accuracy massima sono riportati in tabella 3.2.

Tutte le futures			Selezione per correlazione			Selezione per PCA		
Fold	Accuratezza	h	Fold	Accuratezza	h	Fold	Accuratezza	h
1	0.9937	190	1	0.9723	200	1	0.9955	200
2	0.9938	180	2	0.9691	170	2	0.9948	190
3	0.9934	180	3	0.9696	200	3	0.9955	200
4	0.9920	180	4	0.9801	180	4	0.9946	180
5	0.9928	190	5	0.9698	190	5	0.9944	180
Media	accuratezza:	0.9931	Media	accuratezza:	0.9721	Media	accuratezza:	0.9949
Media	neuroni:	184	Media	neuroni:	188	Media	neuroni:	190

Tabella 3.1: Tabella raggruppativa dei risultati per tutte le feature, quelle selezionate tramite indice di correlazione e quelle tramite PCA

Tutte le futures			Selezione per correlazione			Selezione per PCA		
Acc	Errore medio	h	Acc	Errore medio	h	Acc	Errore medio	h
0.9933	0.03	180	0.9591	0.05	190	0.9927	0.03	190

Tabella 3.2: Risultati dei test utilizzando il numero medio di neuroni ottenuto in Tabella 3.1

Osservando che il risultato migliore si è ottenuto in corrispondenza del numero massimo di neuroni, si è deciso di inserire un ulteriore strato hidden. Inoltre si è deciso di proseguire considerando tutte le feature e quelle selezionate tramite PCA.

I risultati per ogni fold, considerando tutte le feature e quelle selezionate tramite PCA, sono riportati in tabella 3.3; i risultati degli esperimenti in tabella 3.4

Tutte le futures			Selezione per PCA		
Fold	Accuratezza	h	Fold	Accuratezza	h
1	0.9960	70, 90	1	0.9964	100, 90
2	0.9961	100, 80	2	0.9962	100, 90
3	0.9962	100, 90	3	0.9964	80, 100
4	0.9959	80, 80	4	0.9962	100, 100
5	0.9961	90, 70	5	0.9963	100, 60
Media	accuratezza:	0.9960	Media	accuratezza:	0.9963
Media	neuroni:	88, 82	Media	neuroni:	96, 88

Tabella 3.3: Tabella con i risultati ottenuti considerando un numero di neuroni compreso tra 10 e 100 per entrambi gli strati nascosti

Tutte le futures			Selezione per PCA		
Accuratezza	Errore medio	h	Accuratezza	Errore medio	h
0.9951	0.0327	90, 80	0.9945	0.0342	100, 90

Tabella 3.4: Risultati dei test utilizzando il numero medio di neuroni ottenuti in Tabella 3.3

3.3.2 Risultati addestrando la rete NNPred per il problema di regressione sul valore di mercato dei giocatori

Purtroppo la rete costruita usando le API di Keras non si presta alla manipolazione necessaria per la compressione, né per il pruning, non c'è modo per mantenere inattive le connessioni tagliate, né per il weight sharing, non c'è compatibilità con le nuove strutture dati necessarie alla tecnica stessa. Come per la rete precedente si sono usati gli stessi iper-parametri eccezion fatta per il numero di neuroni degli strati nascosti: per questi si è proceduto con una cross-validation i cui risultati sono riportati nelle Tabelle 3.5 e 3.6

Fold	Accuratezza	h
1	0.9951	30, 30
2	0.9959	40, 40
3	0.9958	40, 30
4	0.9953	40, 40
5	0.9955	40, 40
Media	accuratezza:	0.9955
Media	neuroni:	38, 36

Tabella 3.5: Numero di neuroni strati hidden $h \in \{10, \ldots, 40\}$

Il risultato usando il test set usando un numero di neuroni pari alla media in tabella 3.5 ha raggiunto un'accuratezza di 0.9948 e un errore medio di 0.0357.

Fold	Accuratezza	h
1	0.9958	80, 90
2	0.9963	100, 80
3	0.9961	100, 90
4	0.9952	80, 90
5	0.9963	90, 90
Media	accuratezza:	0.9959
Media	neuroni:	90, 88

Tabella 3.6: Numero di neuroni strati hidden $h \in \{10, \ldots, 100\}$

Il risultato usando il test set usando un numero di neuroni pari alla media in Tabella 3.6 ha raggiunto un'accuratezza di 0.9953 e un errore medio di 0.0325.

3.3.3 NNPred: rete neurale per il problema del predecessore

In questo paragrafo vengono riportati i risultati riguardanti il problema del predecessore. La struttura della rete è quella presentata nel Paragrafo 3.2 e i dataset utilizzati sono quelli descritti al Paragrafo 3.1.2. Per ogni tabella riportata, la prima riga indica i risultati ottenuti dalla rete presentata nel Paragrafo 3.2.2 non compressa, in modo da poter effetture un confronto con le righe successive indicanti i risultati di pruning e weight sharing.

3.3.4 Pruning rete per il problema del predecessore

In questo paragrafo verranno illustrati i risultati ottenuti comprimendo la rete utilizzata nel sottoparagrafo 3.2.2 con la tecnica del pruning. Di seguito vengono spiegati i valori delle colonne che valgono sia per le tabelle relative al pruning che per quelle relative al weight sharing, a esclusione del punto "pruning".

- NN1: rete con 0 strati hidden;
- NN2: rete con uno strato hidden con 256 neuroni;
- NN3: rete con due strati hidden, entrambi con 256 neuroni;
- pruning: percentuale di connessioni eliminate; il valore 0 indica la rete non compressa;
- space overhead: la frazione di spazio usata dalla rete per memorizzare la rete rispetto alla dimensione del dataset, misurato in kilobytes tramite la formula

$$\frac{\sum_{i=1}^{n} (|\text{strato}_i|) \times \text{bytes} \times 100}{1024 \times |\text{dataset}|}$$

  dove per strato_i viene presa in considerazione la cardinalità delle strutture dati utilizzate per memorizzare i pesi per l'i-esimo strato, bytes rappresenta il

numero di bytes utilizzato per rappresentare i pesi all'interno delle strutture dati;

- training time: tempo in secondi impiegato dalla rete per il training dopo la compressione;
- ϵ: errore massimo sugli esempi di training;
- error %: errore massimo rispetto alla dimensione del dataset considerato;
- mean error: errore medio.

NN1 dataset 3					
Pruning %	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
0	8.93×10^{-2}	-	9	1.76	4.74×10^{-3}
10	8.56×10^{-2}	7.0×10^{-3}	8	1.56	4.58×10^{-3}
20	8.01×10^{-2}	7.6×10^{-3}	8	1.56	4.58×10^{-3}
30	7.10×10^{-2}	7.5×10^{-3}	8	1.56	4.59×10^{-3}
40	6.03×10^{-2}	7.2×10^{-3}	9	1.76	4.82×10^{-3}
50	5.11×10^{-2}	8.9×10^{-3}	14	2.73	8.91×10^{-3}
60	4.20×10^{-2}	8.9×10^{-3}	21	4.1	1.60×10^{-2}
70	3.13×10^{-2}	1.2×10^{-2}	36	7.03	3.13×10^{-2}
80	2.21×10^{-2}	1.1×10^{-2}	66	1.28×10	6.25×10^{-2}
90	1.30×10^{-2}	1.1×10^{-2}	65	1.27×10	6.25×10^{-2}

NN1 dataset 7					
Pruning %	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
0	3.09×10^{-3}	-	53	6.4×10^{-1}	1.62×10^{-3}
10	5.58×10^{-3}	8.1×10^{-2}	46	5.6×10^{-1}	1.55×10^{-3}
20	5.01×10^{-3}	8.1×10^{-2}	43	5.2×10^{-1}	1.52×10^{-3}
30	4.43×10^{-3}	8.9×10^{-2}	44	5.4×10^{-1}	1.53×10^{-3}
40	3.77×10^{-3}	9.3×10^{-2}	55	6.7×10^{-1}	1.77×10^{-3}
50	3.19×10^{-3}	1.2×10^{-1}	73	8.9×10^{-1}	2.41×10^{-3}
60	2.62×10^{-3}	1.2×10^{-1}	107	1.31	4.17×10^{-3}
70	1.96×10^{-3}	1.7×10^{-1}	107	1.31	4.17×10^{-3}
80	1.38×10^{-3}	2.4×10^{-1}	165	2.01	7.97×10^{-3}
90	8.11×10^{-4}	2.2×10^{-1}	165	2.01	7.97×10^{-3}

È interessante notare come, non comprimendo abbastanza, ovvero non più della metà delle connessioni, non si ottenga una matrice abbastanza sparsa da convernire l'utilizzo di strutture dati come le CSC; queste infatti occuperanno più spazio di una matrice densa più è alto il numero di valori non nulli in quanto useranno tre vettori, come descritto nel Paragrafo 2.2. In questo caso si può notare come lo spazio occupato dalla rete compressa fino al 50% risulti maggiore di quello occupato dalla rete non compressa, sia per quanto riguarda la tabella sopra che per quelle successive. Ad esempio una matrice dei pesi, senza valori nulli, di dimensione 3×3 in cui i pesi sono rappresentati da float32, occuperà 36 bytes; usando invece una csc si occuperanno 88 bytes in memoria. Tuttavia, con percentuali di pruning più elevate e con dataset più numerosi, il vantaggio della compressione si fa evidente, sia in termini di di spazio che di accuratezza, come si vedrà per il dataset più numeroso

(dataset 10).

NN1 dataset 10					
Pruning %	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
0	2.42×10^{-5}	-	905	8.6×10^{-2}	1.85×10^{-4}
10	4.35×10^{-5}	1.1×10	770	7×10^{-2}	1.52×10^{-4}
20	3.91×10^{-5}	1.6×10	770	7×10^{-2}	1.52×10^{-4}
30	3.46×10^{-5}	1.1×10	770	7×10^{-2}	1.52×10^{-4}
40	2.94×10^{-5}	1.1×10	770	7×10^{-2}	1.52×10^{-4}
50	2.49×10^{-5}	1.4×10	770	7×10^{-2}	1.52×10^{-4}
60	2.05×10^{-5}	1.2×10	734	7×10^{-2}	1.50×10^{-4}
70	1.53×10^{-5}	1.1×10	736	7×10^{-2}	1.51×10^{-4}
80	1.08×10^{-5}	1.1×10	748	7×10^{-2}	1.51×10^{-4}
90	6.32×10^{-6}	1.1×10	732	7×10^{-2}	1.48×10^{-4}

NN2 dataset 3					
Pruning %	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
0	1.289×10	-	4	7.8×10^{-1}	1.60×10^{-3}
10	2.324×10	2.1×10	2	3.9×10^{-1}	6.12×10^{-4}
20	2.070×10	2.3×10	2	3.9×10^{-1}	6.21×10^{-4}
30	1.817×10	2.1×10	3	5.8×10^{-1}	6.93×10^{-4}
40	1.563×10	2.0×10	3	5.8×10^{-1}	7.93×10^{-4}
50	1.309×10	2.1×10	3	5.8×10^{-1}	8.62×10^{-4}
60	1.055×10	2.3×10	3	5.8×10^{-1}	9.85×10^{-4}
70	8.01	2.1×10	4	7.8×10^{-1}	1.32×10^{-3}
80	5.47	1.7×10	5	9.7×10^{-1}	1.95×10^{-3}
90	2.93	1.2×10	6	1.17	2.78×10^{-3}

NN2 dataset 7					
Pruning %	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
0	8.06×10^{-1}	-	33	4.0×10^{-1}	1.12×10^{-3}
10	1.45	1.31×10^{2}	35	4.3×10^{-1}	5.94×10^{-4}
20	1.29	1.23×10^{2}	35	4.3×10^{-1}	5.98×10^{-4}
30	1.13	1.18×10^{2}	35	4.3×10^{-1}	6.04×10^{-4}
40	9.76×10^{-1}	1.18×10^{2}	35	4.3×10^{-1}	5.97×10^{-4}
50	8.18×10^{-1}	1.15×10^{2}	34	4.2×10^{-1}	5.99×10^{-4}
60	6.59×10^{-1}	1.11×10^{2}	35	4.3×10^{-1}	6.11×10^{-4}
70	5.00×10^{-1}	1.24×10^{2}	35	4.3×10^{-1}	6.12×10^{-4}
80	3.42×10^{-1}	2.2×10	33	4.0×10^{-1}	8.56×10^{-4}
90	1.83×10^{-1}	5	34	4.1×10^{-1}	1.19×10^{-3}

NN2 dataset 10					
Pruning %	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
0	6.29×10^{-3}	-	1031	9.0×10^{-2}	2.33×10^{-4}
10	1.13×10^{-2}	1.74×10^{2}	713	6.8×10^{-2}	1.46×10^{-4}
20	1.01×10^{-2}	1.25×10^{2}	713	6.8×10^{-2}	1.46×10^{-4}
30	8.87×10^{-3}	1.25×10^{2}	713	6.8×10^{-2}	1.46×10^{-4}
40	7.63×10^{-3}	1.24×10^{2}	713	6.8×10^{-2}	1.46×10^{-4}
50	6.39×10^{-3}	1.22×10^{2}	713	6.8×10^{-2}	1.46×10^{-4}
60	5.15×10^{-3}	1.22×10^{2}	713	6.8×10^{-2}	1.46×10^{-4}
70	3.91×10^{-3}	1.22×10^{2}	713	6.8×10^{-2}	1.46×10^{-4}
80	2.67×10^{-3}	1.22×10^{2}	715	6.8×10^{-2}	1.46×10^{-4}
90	1.43×10^{-3}	1.23×10^{2}	699	6.7×10^{-2}	1.45×10^{-4}

NN3 dataset 3					
Pruning %	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
0	6.308×10	-	4	7.8	1.83×10^{-3}
10	1.13×10^{2}	6.1×10	2	3.9×10^{-1}	6.63×10^{-4}
20	1.01×10^{2}	6.3×10	2	3.9×10^{-1}	6.86×10^{-4}
30	8.855×10	5.6×10	2	3.9×10^{-1}	7.66×10^{-4}
40	7.601×10	5.6×10	3	5.8×10^{-1}	8.79×10^{-4}
50	6.348×10	6.6×10	3	5.8×10^{-1}	8.82×10^{-4}
60	5.094×10	6.6×10	3	5.8×10^{-1}	1.02×10^{-3}
70	3.840×10	7.1×10	3	5.8×10^{-1}	1.15×10^{-3}
80	2.586×10	6.9×10	3	5.8×10^{-1}	1.50×10^{-3}
90	1.332×10	8.1×10	4	7.8×10^{-1}	2.03×10^{-3}

NN3 dataset 7					
Pruning %	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
0	3.94	–	40	4.8×10^{-1}	1.14×10^{-3}
10	7.10	1.97×10^{2}	36	4.3×10^{-1}	7.80×10^{-4}
20	6.31	2.27×10^{2}	36	4.3×10^{-1}	7.58×10^{-4}
30	5.53	9.5×10	32	3.9×10^{-1}	8.18×10^{-4}
40	4.75	1.04×10^{2}	33	4.0×10^{-1}	8.18×10^{-4}
50	3.97	9.6×10	33	4.0×10^{-1}	8.23×10^{-4}
60	3.18	1.25×10^{2}	34	4.1×10^{-1}	8.12×10^{-4}
70	2.40	9.9×10	34	4.1×10^{-1}	8.29×10^{-4}
80	1.61	9.3×10	34	4.1×10^{-1}	8.20×10^{-4}
90	8.32×10^{-1}	2.9×10	32	3.9×10^{-1}	1.24×10^{-3}

NN3 dataset 10					
Pruning %	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
0	3.08×10^{-2}	-	1270	1.2×10^{-1}	2.40×10^{-4}
10	5.54×10^{-2}	3.02×10^{2}	708	6.8×10^{-2}	1.47×10^{-4}
20	4.93×10^{-2}	3.03×10^{2}	708	6.8×10^{-2}	1.47×10^{-4}
30	4.32×10^{-2}	3.04×10^{2}	708	6.8×10^{-2}	1.47×10^{-4}
40	3.71×10^{-2}	3.05×10^{2}	707	6.8×10^{-2}	1.47×10^{-4}
50	3.09×10^{-2}	3.04×10^{2}	707	6.7×10^{-2}	1.47×10^{-4}
60	2.49×10^{-2}	3.00×10^{2}	707	6.7×10^{-2}	1.47×10^{-4}
70	1.87×10^{-2}	3.00×10^{2}	706	6.7×10^{-2}	1.47×10^{-4}
80	1.26×10^{-2}	3.02×10^{2}	700	6.7×10^{-2}	1.46×10^{-4}
90	6.50×10^{-3}	3.00×10^{2}	660	6.3×10^{-2}	1.45×10^{-4}

Osservando i risultati riportati si può notare come non convenga usare sempre la tecnica di pruning su una rete neurale priva di strati nascosti: ad esempio nella prima tabella sembra esserci un errore, sia medio che assoluto, minore, ma che non comporta una riduzione di spazio occupato significativa; dalla seconda tabella risulta addirittura svantaggioso comprimere, in termini di memoria occupata. D'altro canto si nota un miglioramento all'aumentare dei dati rappresentati; infatti, dalla tabella relativa al dataset 7, risulta un errore minore mantenendo un tasso di compressione fino al 30% e nel dataset 10 addirittura si ottengono risultati migliori sia in termini di accuratezza che di spazio occupato. Il discorso cambia quando si aggiungono uno o due strati nascosti: tenendo in considerazione il discorso fatto precedentemente sul rapporto tasso di compressione - spazio occupato, il pruning consente di avere dei buoni margini di risparmio di memoria e buone prestazioni anche in termine di errore medio e assoluto, risultando sempre conveninete rispetto ad una rete non compressa. Per esempio, sul dataset 10, quello più numeroso e quindi più significativo, si ottiene

una riduzione dello spazio da 2.42×10^{-5} KB a 6.32×10^{-6} KB con un errore medio che scende da 1.85×10^{-4} a 1.48×10^{-4} per la rete NN1; una riduzione di spazio occupato da 6.29×10^{-3} KB a 1.43×10^{-3} KB e un errore che passa da 2.33×10^{-4} a 1.45×10^{-4} per la rete NN2; infine, per la rete NN3, lo spazio si riduce da 3.08×10^{-2} KB a 6.50×10^{-3} KB e l'errore da 2.40×10^{-4} a 1.45×10^{-4}.

3.3.5 Weight sharing rete per il problema del predecessore

In questo paragrafo verranno illustrati i risultati ottenuti comprimendo la rete utilizzata nel sottoparagrafo 3.2.2 con la tecnica del weight sharing. Di seguito vengono spiegati i valori delle colonne.

- η: proporzione dello spazio originario occupato da quella compressa; il valore 1 indica la rete non compressa;
- cluster: indica, per ogni matrice dei pesi della rete, il numero di cluster utilizzati.

Si noti come per la rete a zero strati non siano riportati tassi di compressione inferiori al 30% e per le singolo e doppio strato inferiori al 60%. Questo è dovuto alla scelta di determinare il numero di cluster per strato usando la formula (3.3).

$$\text{cluster} = \left(\frac{c_p \times 32d - db}{32} \right) \tag{3.3}$$

dove $c_p \in [0, \ldots, 1]$ è il tasso di compressione che si vuole ottenere, d è la dimensione della matrice dei pesi e b è la quantità di byte usata per rappresentare i puntatori ai centroidi.

NN1 dataset 3						
η	Clusters	Space Overhead (KB)	Training Time (s)	Eplison	Error %	Mean Error
1	-	4.95×10^{-2}	-	9	1.75	4.74×10^{-3}
30	3	1.60×10^{-2}	8.07×10^{-2}	2427	4.74×10^{2}	3.15
40	9	2.06×10^{-2}	8.69×10^{-2}	14373	2.80×10^{3}	1.78×10
50	16	2.59×10^{-2}	9.27×10^{-2}	5714	1.11×10^{3}	7.32
60	22	3.05×10^{-2}	1.93×10^{-1}	5296	1.03×10^{3}	6.80
70	28	3.51×10^{-2}	1.93×10^{-1}	8159	1.59×10^{3}	1.03×10
80	35	4.04×10^{-2}	2.08×10^{-1}	542	1.05×10^{2}	5.83×10^{-1}
90	41	4.50×10^{-2}	2.44×10^{-1}	8	1.56	4.60×10^{-3}

NN1 dataset 7						
η	Clusters	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
1	-	3.09×10^{-3}	-	53	6.47×10^{-1}	1.62×10^{-3}
30	3	1.00×10^{-3}	2.71×10^{-1}	57×10^{7}	69.62×10^{5}	4.27×10^{3}
40	9	1.29×10^{-3}	2.71×10^{-1}	37×10^{4}	46.34×10^{2}	2.71×10
50	16	1.62×10^{-3}	3.02×10^{-1}	10×10^{4}	12.54×10^{2}	7.20
60	22	1.91×10^{-3}	3.55×10^{-1}	21×10^{4}	25.76×10^{2}	1.46×10
70	28	2.19×10^{-3}	3.55×10^{-1}	5528	6.74×10	2.44×10^{-1}
80	35	2.52×10^{-3}	3.98×10^{-1}	63	7.7×10^{-1}	2.26×10^{-3}
90	41	2.81×10^{-3}	4.71×10^{-1}	56	6.8×10	1.82×10^{-3}

NN1 dataset 10						
η	Clusters	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
1	-	2.42×10^{-5}	-	905	8.6×10^{-2}	1.85×10^{-4}
30	3	7.82×10^{-6}	3.8×10	2.22×10^{5}	2.12×10^{2}	1.11
40	9	1.00×10^{-5}	1.8×10	1.51×10^{4}	1.44×10	4.75×10^{-2}
50	16	1.26×10^{-5}	1.9×10	2.29×10^{4}	2.19×10	7.54×10^{-2}
60	22	1.49×10^{-5}	1.8×10	717	6.8×10^{-2}	1.49×10^{-4}
70	28	1.71×10^{-5}	1.9×10	715	6.8×10^{-2}	1.48×10^{-4}
80	35	1.97×10^{-5}	1.9×10	706	6.7×10^{-2}	1.48×10^{-4}
90	41	2.20×10^{-5}	1.9×10	706	6.7×10^{-2}	1.48×10^{-4}

NN2 dataset 3						
η	Clusters	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
1	-	1.28×10^{1}	-	4	7.8×10^{-1}	1.6×10^{-3}
60	$1638, 89$	8.01	2.8×10	5	9.7×10^{-1}	2.21×10^{-3}
70	$3276, 115$	9.28	3.4×10	2	3.9×10^{-1}	5.70×10^{-4}
80	$4915, 140$	1.05×10	4.3×10	2	3.9×10^{-1}	5.43×10^{-4}
90	$6553, 166$	1.18×10	4.8×10	2	3.9×10^{-1}	5.48×10^{-4}

NN2 dataset 7						
η	Clusters	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
1	-	8.05×10^{-1}	-	33	4.0×10^{-1}	1.12×10^{-3}
60	$1638, 89$	5.00×10^{-1}	1.71×10^{2}	33	4.0×10^{-1}	6.13×10^{-4}
70	$3276, 115$	5.80×10^{-1}	2.34×10^{2}	33	4.0×10^{-1}	5.95×10^{-4}
80	$4915, 140$	6.59×10^{-1}	2.80×10^{2}	33	4.0×10^{-1}	5.82×10^{-4}
90	$6553, 166$	7.38×10^{-1}	3.17×10^{2}	34	4.1×10^{-1}	5.90×10^{-4}

NN2 dataset 10						
η	Clusters	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
1	-	6.29×10^{-3}	-	1031	9.8×10^{-2}	2.33×10^{-4}
60	$1638, 89$	3.91×10^{-3}	2.64×10^{2}	292	2.8×10^{-2}	5.11×10^{-5}
70	$3276, 115$	4.53×10^{-3}	2.44×10^{2}	753	7.2×10^{-2}	1.49×10^{-4}
80	$4915, 140$	5.15×10^{-3}	2.82×10^{2}	714	6.8×10^{-2}	1.45×10^{-4}
90	$6553, 166$	5.77×10^{-3}	3.17×10^{2}	684	6.5×10^{-2}	1.45×10^{-4}

NN3 dataset 3

η	Clusters	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
1	-	6.31×10	-	4	7.8×10^{-1}	1.83×10^{-3}
60	$1638, 6553, 89$	3.84×10	2.00×10	3	5.8×10^{-1}	1.25×10^{-3}
70	$3276, 13107, 115$	4.46×10	1.43×10^{2}	2	3.9×10^{-1}	6.28×10^{-4}
80	$4915, 19660, 140$	5.09×10	1.87×10^{2}	4	7.8×10^{-1}	1.80×10^{-3}
90	$6553, 26214, 166$	5.72×10	2.03×10^{2}	2	3.9×10^{-1}	6.73×10^{-4}

NN3 dataset 7

η	Clusters	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
1	-	3.94	-	40	4.9×10^{-1}	1.14×10^{-3}
60	$1638, 6553, 89$	2.40	2.43×10^{2}	34	4.1×10^{-1}	6.01×10^{-4}
70	$3276, 13107, 115$	2.79	2.15×10^{2}	39	4.7×10^{-1}	9.25×10^{-4}
80	$4915, 19660, 140$	3.18	7.46×10^{2}	36	4.4×10^{-1}	7.40×10^{-4}
90	$6553, 26214, 166$	3.57	1.26×10^{3}	35	4.3×10^{-1}	6.88×10^{-4}

NN3 dataset 10

η	Clusters	Space Overhead (KB)	Training Time (s)	ϵ	Error %	Mean Error
1	-	3.08×10^{-2}	-	1270	1.2×10^{-1}	2.40×10^{-4}
60	$1638, 6553, 89$	1.87×10^{-2}	6.16×10^{2}	771	7.4×10^{-2}	1.66×10^{-4}
70	$3276, 13107, 115$	2.18×10^{-2}	8.28×10^{2}	678	6.5×10^{-2}	1.47×10^{-4}
80	$4915, 19660, 140$	2.49×10^{-2}	1.03×10^{3}	709	6.8×10^{-2}	1.47×10^{-4}
90	$6553, 26214, 166$	2.79×10^{-2}	2.00×10^{3}	675	6.4×10^{-2}	1.45×10^{-4}

Osservando i risultati ottenuti usando la tecnica di weight sharing si può notare come, per la rete senza strati nascosti, non convenga sempre applicare tale compressione: infatti, nonostante l'errore massimo diminuisca all'aumentare dei cluster, il tasso di compressione rimane minore rispetto al pruning; fa eccezione il dataset 10 che, pur mantenendo dei tassi di compressione ridotti, ha un'accuratezza ancora migliore della rete non compressa. La situazione cambia diametralmente in termini di errore medio e assoluto: infatti sulle reti compresse con uno e due strati nascosti si ottiene più precisone nella previsione rispetto alla rete originale. Tuttavia non si verifica un analogo miglioramento sul fronte della memoria; nonostante lo spazio occupato da una rete clusterizzata sia inferiore rispetto a quella originale, non se ne apprezza uno scarto significativo (l'unico caso degno di nota può essere la riga relativa a un tasso di compressione, η, pari a 60, in cui la rete compressa occupa poco più della metà dello spazio originario).

Conclusioni

In questo elaborato si è affrontato il problema di come comprimere una rete neurale usata per fare regressione impatti su diversi aspetti della stessa, principalmente sul compromesso tra accuratezza e dimensione occupata in memoria. Si è partiti da un caso particolare di regressione, la previsione del valore di mercato di giocatori di calcio, permettendo la comprensione della struttura di una rete neurale adatta a trattare tale problema. Una volta osservati i risultati ottenuti si è passati al problema del predecessore decidendo di affrontarlo usando tre diversi modelli di rete; a zero, singolo e doppio strato nascosto. Dai risultati ottenuti si può evincere che, sia con la tecnica di pruning che weight sharing, si ottengono dei buoni risultati considerando il rapporto accuratezza e memoria occupata. In particolare si può notare che eliminando fino al 60% di connessioni tramite pruning si continua a ottenere un errore massimo e medio inferiore alla rete originaria, il che si può leggere anche come un risparmio fino a più della metà dello spazio occupato dalle matrici della rete. C'è tuttavia da considerare anche il fattore tempo; usare una struttura dati come le matrici sparse richiede più tempo in fase di training in quanto le operazioni di moltiplicazione tra CSC diventano più dispendiose in termini di tempo. Ciònonostante questa problematica si presenta in fase di training, tempo che può essere ammortizzato su tutte le più numerose operazioni di previsione che si richiederanno alla rete. Anche la tecnica di weight sharing ha dato buoni risultati: infatti, escludendo il caso della rete senza strati nascosti, la rete compressa ha sempre dato risultati migliori o

uguali in termini di errore medio, massimo e percentuale rispetto a quella originale. Si ripropone un pattern simile a quello del pruning; sono state ottenute delle reti con accuratezza simile o addirittura migliori come accuracy e meno onerose in termini di memoria ma con un training time più lungo, anche in questo caso dato dalle strutture dati impiegate (per le moltiplicazioni matriciali bisogna, per ogni elemento, accedere al relativo centroide, avendo quindi un doppio accesso), ma anche in questo caso il tempo di addestramento è ammortizzabile.

In conclusione, l'obiettivo di ridurre la memoria è stato raggiunto, almeno fino a tassi di compressione del 60%, mantenendo le stesse accuratezze di predizione del modello base o addirittura migliorandole. Tuttavia, per poter essere applicate a contesti reali in cui si ha un limite di memoria prefissato, occorrerebbe un'ulteriore analisi che sviluppi un algoritmo che automaticamente, dato il limite di memoria da non superare, scelga la configurazione migliore per i metodi di compressione che rispetti quel limite. Questo può sicuramente essere un argomento per sviluppi futuri.

Bibliografia

[1] Ovidiu Vermesan and Peter Friess. *Internet of Things – From Research and Innovation to Market Deployment*, page 8. River Publisher, 2014.

[2] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*, pages 727,728. Prentice Hall, 2010.

[3] Stuart J. Russell and Peter Norvig. *Artificial Intelligence A Modern Approach*, pages 761, 762. Prentice Hall, 2010.

[4] J J Hopfield. Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences*, 1982.

[5] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *ArXiv e-prints*, pages 2–4, 11 2015.

[6] L. R. Medsker and L. C. Jain. *Recurrent Neural Networks – Design and Applications*. CRC Press, 2001.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[8] David E. Rumelhart, Richard Durbin, Richard Golden, and Yves Chauvin. *Backpropagation: The basic theory*, pages 1–34. 2008.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[10] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. *Published as a conference paper at ICLR 2016*, 2 2016.

[11] Youcef Saad. Sparskit: a basic tool kit for sparse matrix computations - version 2. 1994.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, pages 218–232. Addison-Wesley Professional, 1994.

[13] Wang Lei, Huawei Chen, and Yixuan Wu. Compressing deep convolutional networks using k-means based on weights distribution. 07 2017.

[14] Youguo Li and Haiyan Wu. A clustering method based on k-means algorithm. *Physics Procedia*, 25, 12 2012.

[15] Noam Slonim, Ehud Aharoni, and Koby Crammer. Hartigan's k-means versus lloyd's k-means: is it time for a change? *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, 08 2013.

[16] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. An efficient k-means clustering algorithm. *Proc First Workshop High Performance Data Mining*, 04 2000.

[17] D. Sculley. Web-scale k-means clustering. 2010.

[18] Azme Khamis, Zuhaimy Ismail, Haron Khalid, and Ahmad Mohammed. The effects of outliers data on neural network performance. *Journal of Applied Sciences*, 5, 01 2005.

[19] I.T. Jolliffe. *Principal Component Analysis*, pages 1–6. Springer, 1986.

[20] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2009.

[21] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. *PMLR 28(3):1139-1147*, 2013.

[22] Twan van Laarhoven. L2 regularization versus batch and weight normalization. *ArXiv*, abs/1706.05350, 2017.