Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Global scaling predittivo di un'architettura a microservizi con tecniche di deep learning: gestione del flusso di email

Relatore:
Chiar.mo Prof.
Maurizio Gabbrielli

Presentata da:
Edoardo Procino

Correlatore:
Dott. Stefano Pio Zingaro

Correlatore:
Dott. Lorenzo Bacchiani

Sessione II
Anno Accademico 2020/2021

Indice

Introduzione

Nell'articolo *Microservice Dynamic Architecture-Level Deployment Orchestration* di Lorenzo Bacchiani et al.[3] viene discusso un nuovo approccio per lo scaling globale di applicazioni di microservizi. Più precisamente, è stato sviluppato un algoritmo per riconfigurare automaticamente il sistema in base al carico di lavoro in entratata in un determinato istante di tempo. Il funzionamento di questo approccio viene poi testato sul caso studio delle email dimostrando come, rispetto al classico scaling locale, eviti rallentamenti a cascata con conseguente aumento di perdita di messaggi e di latenza.

L'obiettivo di questa tesi è integrare, a questo sistema, una rete neurale, così da poter implementare un autoscaling predittivo che potrebbe portare ad avere performance migliori, dato che lo scaling avverrebbe prima dell'aumento del flusso in input, con conseguente diminuzione della latenza e dei messaggi persi.

L'idea è di far funzionare questo sistema anche in situazioni reali. Per far ciò, l'ideale sarebbe allenare la rete quando il carico di email è basso, cioè la notte, sui dati raccolti durante la giornata, così che il sistema sia sempre più affidabile ed efficiente.

Per implementare questo meccanismo, abbiamo sfruttato il dataset *Enron Mail Dataset* [7] con il quale abbiamo allenato una rete neurale e fatto delle predizioni, che abbiamo usato per fare delle simulazioni per verificare se il sistema funzionasse meglio.
In particolare, i risultati che ci aspettavamo di ottenere erano due:

- Capire come manipolare il dataset per ottenere delle predizioni con il più basso errore possibile;
- Verificare, se con il predictive scaling, le performance sarebbero migliorate.

Nell'elaborato abbiamo affrontato i seguenti punti:

- Preprocessing dei dati con trasformazione dei valori del campo "data" in valori nominali così da renderla più adatta all'allenamento della rete;

- Struttura, iperparametri e funzionamento della rete neurale profonda;
- Analisi dei vari modelli utilizzati per capire quali dataset — tra i tre proposti — e feature siano i migliori;
- Simulazione in linguaggio ABS (Abstract Behavioral Specification) del sistema con la rete neurale integrata e suo confronto con il sistema senza le predizioni.

1. Enron Email Dataset

1.1 Il dataset

Il dataset *Enron Email Dataset* è un dataset che contiene le email di 150 dipendenti dell'azienda Enron. È formato da 150 directory, ognuna delle quali corrisponde a un impiegato/a dell'azienda e in ogni cartella si trovano le email inviate e ricevute da quell'utente. In tutto sono presenti 517.402 email comprensive sia di testo che di metadati.

Anche se il dataset non contiene un numero sufficiente di email per simulare il sistema in ABS [3], dato che i microservizi (§ 4.1) sono stati modellati per gestire un numero di email molto superiore, è stato scelto perché è largamente utilizzato in letteratura e perché, la cosa importante per la simulazione finale, è la distribuzione delle email più che il loro numero.
Come spiegato nel capitolo dedicato (§ 4.2.1), per effettuare le simulazioni con dei valori adeguati, abbiamo moltiplicato i dati reali e le predizioni della rete per 2,5. In questo modo, abbiamo lasciato invariato il pattern delle email del dataset ma, allo stesso tempo, il numero di email elevato ha permesso di simulare un sistema reale.

1.2 Preprocessing dei dati

Il metadato su cui abbiamo lavorato per predire le email in arrivo è la data, presente nel dataset nel formato *%a, %d %b %Y %H:%M:%S %z (%Z)* [5].
Prima di poter utilizzare i dati per allenare la rete neurale (§ 2.2), li abbiamo dovuti rielaborare tramite alcuni script in python. In particolare, abbiamo prima estratto il metadato relativo alla data per poi elaborarlo in più passaggi per ottenere un dataset utilizzabile per il training della rete.
Nei seguenti paragrafi sono descritti nel dettaglio i vari passaggi.

1.2.1 Estrazione e modifica dei dati

Per prima cosa, utilizzando il modulo *Path* della libreria *os* [9] abbiamo estratto la data da ogni email per poi salvarla in un file csv, grazie all'omonima libreria [6]. Il file si presenta con due colonne (Tabella 1.1): una per l'id dell'email — rimosso in seguito — e una per la data, e verrà utilizzato nelle prossime fasi di prepocessing. Il numero delle righe del file è pari al numero di email presenti nel dataset.

Mail-ID	Data
<10018609.1075841067540.JavaMail.evans@thyme>	Sat, 14 Apr 2001 12:57:00 -0700
<30759990.1075841067518.JavaMail.evans@thyme>	Sat, 14 Apr 2001 14:30:00 -0700

Tabella 1.1: *La tabella mostra le prime due entry del file csv dopo l'estrazione dei dati*

Successivamente, abbiamo compresso il dataset unendo in un'unica riga più entry aventi la stessa data e ora, ottenendo così 224.122 elementi. Per fare ciò, è stata aggiunta al dataset una nuova colonna denominata "mail" che indica il numero esatto di email ricevute in quell'istante. Utilizzando la libreria *Pandas* [10], abbiamo poi ordinato cronologicamente le email per poter creare le finestre temporali (vedi paragrafo successivo). Oltre a questo, abbiamo anche apportato delle modifiche al dataset perché alcuni dati outlier avrebbero potuto compromettere la successiva fase di addestramento.
Di seguito, l'elenco di tutte le modifiche:

- Impostato 2021 alle cinque date con anni successivi a quello corrente.
- Pandas di default ha sostituito con 2001 tutti gli anni delle date come 0001, 0002, ecc.
- Rimozione di tutte le email dell'anno 1980 perché associate all'orario 00:00:00. Questo errore nel dataset faceva sì che risultassero più di 500 email nella data *1980/01/01* alle ore *00:00:00*.
- A causa di fusi orari diversi presenti nelle varie date (alcune in PST e altre in PDT), abbiamo trasformato il fuso orario di tutte in UTC (fuso orario scelto come riferimento globale), impostando l'omonimo flag a "True" nel metodo *to_datetime* [12] di Pandas. Tale metodo viene utilizzato per convertire le date da *string* a *datetime64*, così da poterle utilizzare per il sorting. Abbiamo scelto UTC come fuso orario per due motivi principali: (1) non dover considerare il fuso orario nella successiva fase di addestramento della rete neurale; (2) non fare supposizioni sull'ora, dato che stiamo considerando un sistema distribuito e non possiamo sapere a priori dove si trovano le varie macchine.

- Modifica del formato della data in *%Y-%m-%d %H:%M:%S* [5], così da rendere il dataset più semplice da leggere e capire.

1.2.2 Creazione delle finestre temporali e passaggio a dati nominali

Lo scopo della rete neurale è quello di predire le email che arriveranno nei prossimi m minuti basandosi sulle email arrivate precedentemente, per cui abbiamo modificato il dataset rimuovendo la colonna "mail" e aggiungendo due colonne nuove:

- "mail_count_past": per ogni entry del set, indica le email già ricevute in una finestra temporale di *window_past* minuti;

- "mail_count_future": per ogni entry del set, conta la email ricevute dalla data e ora correnti fino ad una finestra temporale di *window_future* minuti nel futuro.

Per semplicità e per dare maggiore coerenza ai dati, abbiamo scelto di settare le due finestre con valori uguali — *window_future* = *window_past* — e sono stati testati i valori di 10, 60 (Tabella 1.2) e 120 minuti (§ 3.2.2). Il risultato che ci aspettavamo di ottenere era di avere risultati migliori aumentando la dimensione delle finestre dato che, a questo aumento, sarebbe corrisposto un aumento anche della quantità di informazione presente in ogni entry del dataset.

Abbiamo realizzato, per far ciò, un semplice algoritmo in python, che per ogni entry del dataset, confronta la sua data con quelle precedenti e future, facendo una sottrazione in valore assoluto tra esse. Se la differenza è minore di *window_future* (o *window_past* nel caso si guardi al passato), allora le email di tutte le entry coinvolte vengono sommate e inserite nella colonna "mail_count_future" (o "mail_count_past") dell'entry corrente, altrimenti l'algoritmo passa alla data successiva.

Data	mail_count_past	mail_count_future
2001-05-07 04:00:00	0	9
2001-05-07 04:01:38	1	8
2001-05-07 04:06:58	2	7
2001-05-07 04:13:00	3	6
2001-05-07 04:22:00	6	5

Tabella 1.2: *La tabella mostra delle entry del dataset creato con le finestre impostate a 60 minuti.*

Il passo successivo è stato quello di rendere il campo "Data" utilizzabile per allenare la rete neurale. Per prima cosa, abbiamo suddiviso la colonna della data in 6 colonne per indicare mese, giorno del mese, ora, minuti, secondi e giorno della settimana. L'anno non è preso in consierazione, dato che la sua eventuale influenza sul numero di email non è rilevante ai fini del progetto, mentre lo è l'influenza, per esempio, del giorno della settimana, dato che presumibilmente il numero di email sarà più alto nei giorni lavorativi piuttosto che nel weekend.
Dopo queste manipolazioni il dataset ha l'aspetto rappresentato nella Tabella 1.3.

month	day	dOfWeek	hour	minute	second	mail_count_past	mail_count_future
5	7	0	4	0	0	0	9
5	7	0	4	1	38	1	8
5	7	0	4	6	58	2	7
5	7	0	4	13	0	3	6
5	7	0	4	22	0	6	5

Tabella 1.3: *La tabella mostra le stesse entry della tabella 1.2 dopo l'aggiunta delle nuove colonne. I giorni della settimana sono codificati con numeri da 0 a 6 dove lo 0 rappresenta il lunedi e il 6 la domenica.*

Infine, abbiamo trasformato i dati relativi a data e ora in dati nominali. Questo ha lo scopo di far tenere in considerazione al modello solo il fatto che periodi della giornata, del mese o dell'anno possano incidere sul numero di email, altrimenti avrebbe potuto imparare relazioni inutili tra un determinato valore della data e un numero di email particolarmente alto o basso.
Per esempio, non sarebbe stato corretto associare al ventottesimo giorno del mese un eventuale alto (o basso) numero di email, dato che il sistema avrebbe potuto intendere che quell'aumento (o decremento) nel valore da predire fosse dovuto all'alto (o basso) valore del giorno in esame.

Per fare ciò, utilizzando il metodo *get_ dummies* di Pandas [11], abbiamo suddiviso ogni colonna relativa ai valori di data e ora in più colonne: 12 per i mesi (da 1 a 12), 31 per i giorni (da 1 a 31), 24 per le ore (da 0 a 23), 7 per i giorni della settimana (da 0 a 6) e 60 per i minuti e 60 per i secondi (entrambe da 0 a 59). In ogni riga queste colonne hanno valore 0 tranne in quella relativa al giusto valore della data e ora in esame. Per esempio, la data *2001-05-07 04:01:38* avrà valore 1 solo nelle seguenti colonne: nella quinta dei mesi, nella settima dei giorni, nella prima del giorno della settimana (come visibile nella seconda riga della Tabella 1.3 la data in questione è di lunedì), nella quinta dell'ora, nella seconda dei minuti e nella trentanovesima dei secondi.

dOfWeek0	dOfWeek1	dOfWeek2	dOfWeek3	dOfWeek4	dOfWeek5	dOfWeek6
0	0	1	0	0	0	0
1	0	0	0	0	0	0
1	0	0	0	0	0	0

Tabella 1.4: *La tabella mostra una sezione del dataset finale. In particolare sono riportate solo le colonne relative al giorno della settimana con tre esempi. Nella prima riga la data è di mercoledì, mentre nella seconda e nella terza di lunedì.*

2. La rete neurale profonda

Una rete neurale profonda — *deep neural network* — è una rete neurale artificiale composta da un livello di input, un livello di output e più livelli nascosti — *hydden layer*. Ogni livello è un insieme di neuroni e ogni neurone è collegato a tutti i neuroni del livello successivo tramite dei collegamenti ai quali è associato un peso. Ogni neurone esegue una media ponderata dei suoi input (Formula 2.1) provenienti da tutti i neuroni del livello precedente, poi applica una funzione di attivazione per ottenere la non linearità e, quindi, essere in grado di risolvere anche problemi più complessi. Infine, trasmette l'output così calcolato a tutti i neuroni del livello seguente.

L'obiettivo del training della rete è minimizzare una determinata funzione di loss — o costo. Ciò è fatto iterativamente aggiustando ogni volta i pesi relativi ai collegamenti tra i vari neuroni (W nella Formula 2.1) basandosi sull'output ottenuto nell'iterazione corrente. Questo processo prende il nome di *backpropagation*: l'aggiornamento parte dai pesi che regolano il livello di output, poi, "muovendosi all'indietro", vengono aggiustati anche gli altri.

$$y = x * W^T + b \tag{2.1}$$

2.1: *L'input x viene moltiplicato per la matrice dei pesi trasposta e successivamente viene sommato un bias.*

2.1 Struttura della rete

Il modello utilizzato è una deep neural network che sfrutta l'apprendimento supervisionato — imparando quindi da dati che posseggono una label — e la regressione non lineare per cercare di predire in output "mail_count_future" — la label — sfruttando come input "mail_count_past" e i valori della data corrente strutturati in modo nominale come riportato nella Tabella 1.4.

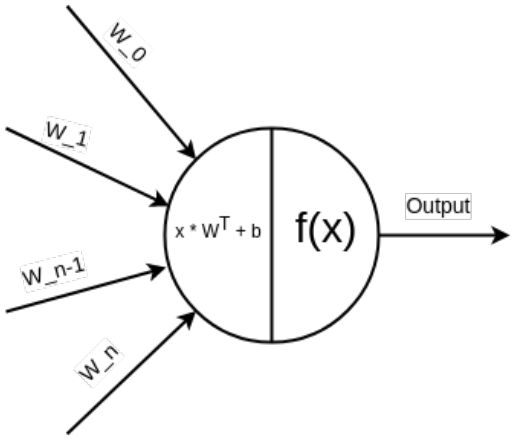


Figura 2.1: *Struttura di un singolo neurone*

La rete è caratterizzata da un input layer di 135 neuroni, 3 hidden layer rispettivamente di 375, 187 e 93 neuroni e, infine, un output layer con un singolo neurone. La struttura è quella riportata nella Figura 2.2.

Una volta calcolata la trasformazione lineare, ogni neurone applica una funzione di attivazione *ReLU* ($y = x^+ = Max(0, x)$, Figura 2.3) per aggiungere non linearità alla soluzione del problema.
Abbiamo utilizzato una funzione ReLU perché è più efficiente nel training delle reti neurali rispetto ad altre funzioni ampiamente usate in questo campo, come la *sigmoid* e la *tangente iperbolica* [14].

2.2 La fase di addestramento della rete

2.2.1 Insiemi di train e di test

Una volta strutturata la rete, per prima cosa abbiamo ordinato il dataset in modo randomico così da essere sicuri che la successiva divisione in training e test set garantisse ad entrabi gli insimi di essere rappresentativi del dataset originale [15].
Abbiamo quindi suddiviso il dataset in due parti:

- training set, corrispondente al'80% del totale, cioè l'insieme dei dati con i quali verrà addestrata la rete;

- test set, usato per testare il modello allenato e verificare l'errore sulle predizioni (Formula 2.2).

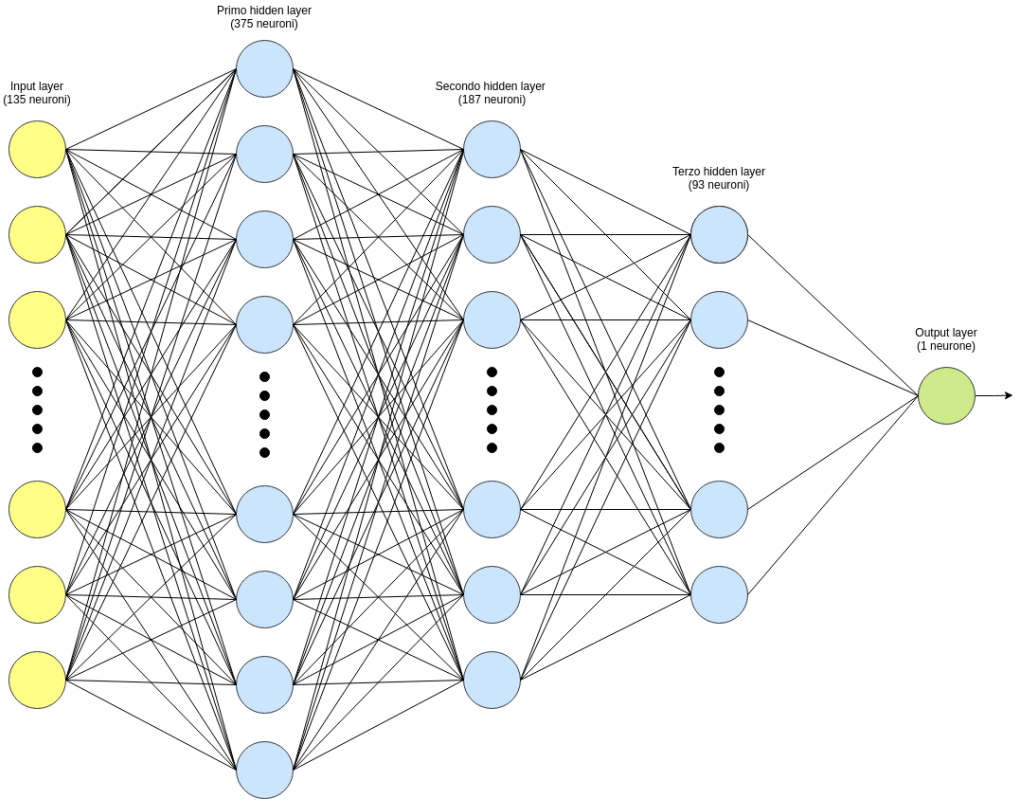


Figura 2.2: *Struttura della rete neurale*

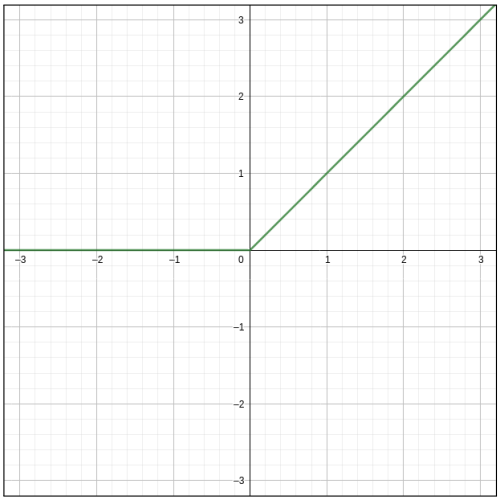


Figura 2.3: *Funzione di attivazione ReLU*

$$Errore = \frac{||y\_test - y\_predicted||_2}{||y\_test||_2} \cdot 100 \tag{2.2}$$

Come ultima modifica, abbiamo suddiviso ulteriormente il training set in dei sotto insiemi da 1024 elementi l'uno, così da poter utilizzare la tecnica *mini_ batch* durante la fase di training. Questa tecnica permette di avere ottimi risultati senza rinunciare all'efficienza durante la computazione [4], ed è migliore, sopratutto in caso di dataset molto grandi, rispetto al *Batch Gradient Descent* e allo *Stochastic Gradient Descent.*

2.2.2 Inizializzazione degli iperparametri e training della rete

Siamo poi passati alla parte di addestramento della rete. Come prima cosa, abbiamo settato gli iperparameti e inizializzato la rete (righe da 1 a 7 nel Listing 2.1 del codice riportato sotto). Il passaggio successivo è stato quello di inizializzare:

- la funzione di loss, necessaria per capire, durante il training, quanto la predizione attuale è distante dai valori effettivi;
- l'optimizer, cioè l'algoritmo di ottimizzazione utilizzato per cercare di minimizzare la funzione di loss.

Per quanto riguarda la funzione di loss, abbiamo utilizzato la *Mean Squared Error* (Formula 2.3) perché è una funzione largamente utilizzata in letteratura. Come optimizer abbiamo scelto l'*algoritmo di Adam* [8] perché è un algoritmo molto efficiente ed è adatto soprattutto per problemi con molti parametri.

$$MSE = \frac{1}{n} \cdot \sum_{i=1}^{n} (y\_test_i - y\_predicted_i)^2 \tag{2.3}$$

I risultati del modello allenato sul training set sono discussi nel capitolo 3, dove vengono analizzati 9 diversi modelli. In tutto abbiamo utilizzato tre dataset che differivano per la finestra temporale scelta e, per ognuna, sono state allenate tre reti con diversi input: la prima rete con input "mail_count_past", il mese, il giorno del mese, il giorno della settimana e l'ora, la seconda con l'aggiunta anche dei minuti e la terza con anche i secondi.

```
1 inputDim = 135
2 outputDim = 1
3 learningRate = 1e-4
4 epochs = 3000
5 n_hidden = 375
```

```
n_hidden2 = 187
n_hidden3 = 93

net = Net(inputDim, n_hidden,n_hidden2,n_hidden3, outputDim)
loss_f = torch.nn.MSELoss() #mean squared error
optimizer = torch.optim.Adam(net.parameters(), lr = learningRate)

loss=0
for epoch in range(0, epochs):

    for i,data in enumerate(trainloader,0):
        inputs, targets = data
        inputs, targets = inputs.float(), targets.float()
        targets = targets.reshape(((targets.shape[0], 1)))

        optimizer.zero_grad()
        outputs = net(inputs)

        loss = loss_f(outputs, targets)

        loss.backward()
        optimizer.step()
```

Listing 2.1: Inizializzazione degli ipermarametri e loop di training

3. Risultati del training e confronto dei diversi modelli

Le predizioni fatte con i vari modelli hanno mostrato come queste siano influenzate sia dalla finestra utilizzata nella creazione del dataset sia l'utilizzo di diverse feature in input. Allo stesso tempo, per ogni modello, i risultati sono stati molto diversi tra loro.

Nei paragrafi successivi, saranno confrontate prima la discesa della funzione di loss sui diversi dataset utilizzati e dopo le configurazioni, per poi analizzare, nel dettaglio, quale sia la migliore tra quelle confrontate, andando a vedere con quale dataset e con quali input è stata ottenuta la performance migliore.

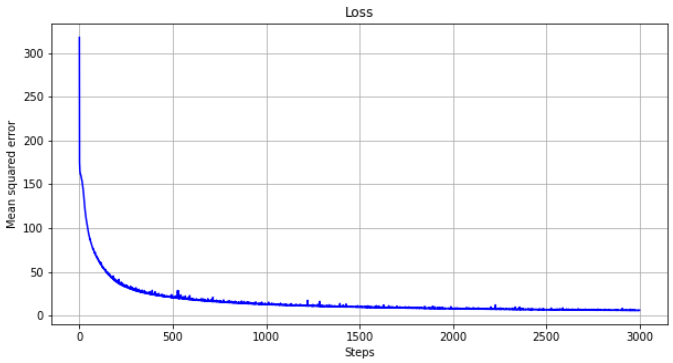
Prima del calcolo dell'errore (Tabella 3.1 e Tabella 3.2) e del plot dei grafici che confrontano le predizioni con dati reali (Figure 3.2, 3.3, 3.4, 3.5 e 3.6), abbiamo modificato le predizioni portando a 0 tutti i valori negativi.
Questa modifica ai dati predetti è stata fatta per due motivi:

- Non ha senso che in un dato momento arrivi un numero negativo di email;
- Facendo questa modifica l'errore si abbassa dato che, non potendo esistere istanti con un numero negativo di email, il valore 0 è sicuramente più vicino al dato reale rispetto a qualsiasi numero negativo.

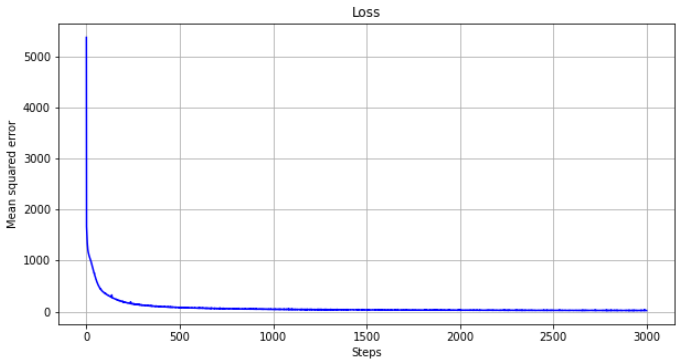
3.1 Risultati del training

Anche se con notevoli differenze, la rete è riuscita a minimizzare la funzione di loss con tutti e tre i dataset costruiti con le diverse finestre.
Come visibile nella Figura 3.1, andando ad aumentare la dimensione della finestra, la curva di discesa della funzione di loss diventa più omogenea e riesce ad arrivare a valori più prossimi allo 0.

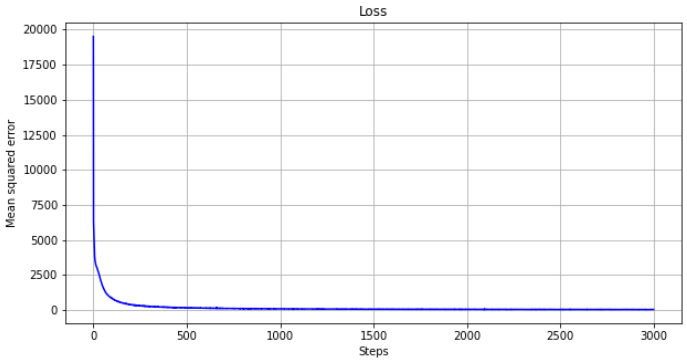
La vicinanza allo 0 delle diverse curve corrisponde a un migliore apprendimento e, quindi, a dei risultati migliori sulle predizioni (vedi paragrafi successivi).



(a) Finestre di 10 minuti



(b) Finestre di 60 minuti



(c) Finestre di 120 minuti

Figura 3.1: *Discesa della funzione di loss rispetto alle diverse finestre. Gli input dati ai vari modelli sono quelli di predittore_minuti (dettagli nel prossimo paragrafo)*

3.2 Confronto dei diversi modelli

Come accennato nel capitolo precedente, per riuscire a ottenere un modello con il più basso errore possibile sulle predizioni, ci siamo mossi in due direzioni:

- Capire quale fosse l'insieme di input migliore da utilizzare;
- Capire quale fosse il valore migliore per settare *window_past* e *window_future*, cercando eventuali correlazioni tra dei valori alti o bassi delle finestre e l'errore sulle predizioni.

Nei prossimi paragrafi analizzeremo i risultati ottenuti dai vari test cercando di capire quale sia la soluzione migliore.

3.2.1 Analisi degli insiemi di input

In questa sezione confronteremo tre predittori che differiscono per l'insieme di input utilizzato:

- *predittore_ore* ha come input "mail_cont_past" e i valori di mese, giorno, giorno della settimana e ora espressi in modo nominale (§ 1.2.2);
- *predittore_minuti* aggiunge agli input precedenti i minuti;
- *predittore_secondi* rispetto a *predittore_minuti* aggiunge anche i secondi.

Il dataset che utilizzeremo per questa prima analisi è quello con le finestre impostate a 120 minuti dato che è il set che permette di ottenere i risultati migliori.

Confronti sul dataset con finestra di 120 minuti

Come visibile nella Tabella 3.1, i predittori *predittore_minuti* e *predittore_secondi* hanno un errore pari al 5,4%, quindi minore rispetto al *predittore_ore* che non sfrutta gli input aggiutivi, il quale ha un errore del 6,1%. Ciò trova riscontro anche nei grafici della Figura 3.2 dove è evidente che i due predittori con errore più basso approssimano in modo migliore la funzione dei valori reali.

	predittore_ore	*predittore_minuti*	*predittore_secondi*
Errore	6,1%	5,4%	5,4%

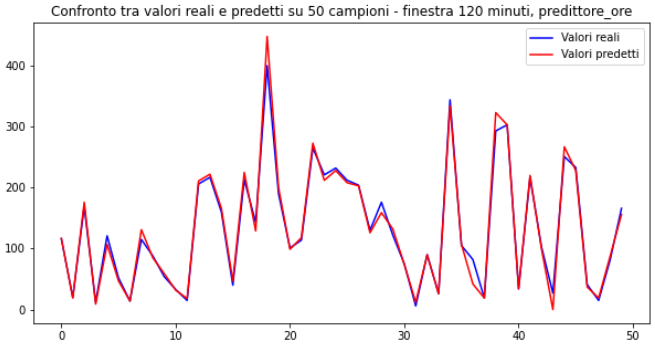
Tabella 3.1: *La tabella mostra l'errore (Formula 2.2) sulle predizioni rispetto ai dati reali utilizzando il dataset con finestra di 120 minuti.*

A fronte di questi risultati, la scelta migliore è ricaduta sull'utilizzo del *predittore_minuti*, dato che è quello con il miglior trade-off tra valore dell'errore e numero di feature di input.

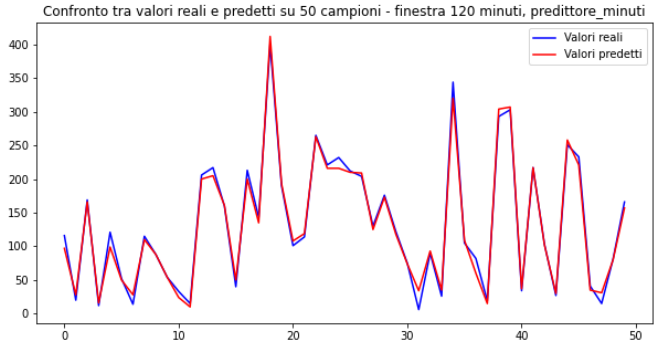
3.2.2 Analisi dei valori delle finestre

Una volta stabilito quali fossero le migliori feature di input, ci siamo concentrati sul capire il miglior valore per settere le due finestre *window_past* e *window_future*.

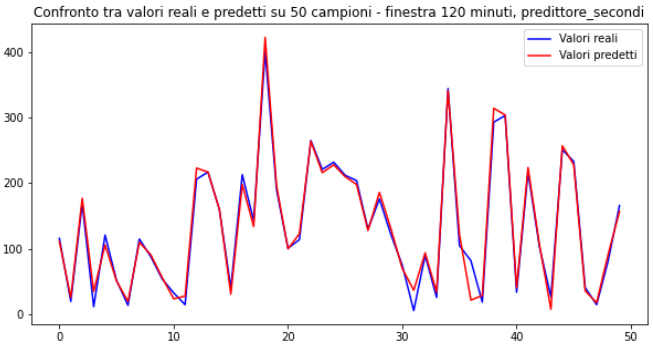
Per fare questo, abbiamo fissato il *predittore_minuti* ignorando gli altri predittori, perché questo era quello con i risultati migliori. Però, per completezza, abbiamo



(a) *predittore_ore*



(b) *predittore_minuti*



(c) *predittore_secondi*

Figura 3.2: *Confronti tra valori reali e predetti sul dataset con finestra di 120 minuti e diversi set di dati in input*

anche analizzato, nel dettaglio, tutte le combinazioni tra feature in input e finestre temporali (§ 3.3).

Finestre di 10 minuti

Utilizzando questo dataset come input al predittore, si ottengono i risultati peggiori dato che, con una finestra temporale piccola, il modello non riesce a trovare delle associazioni solide tra "mail_count_past" e "mail_count_future", arrivando a un errore del 39,1%.
Come visibile nella Tabella 3.2, questo dataset è quello che ha le performance peggiori tutti quelli analizzati.

Questo grande errore è facilmente visibile nel grafico sottostante dove viene mostrata la funzione reale e la funzione approssimatrice su un capione di 50 elementi del test set.

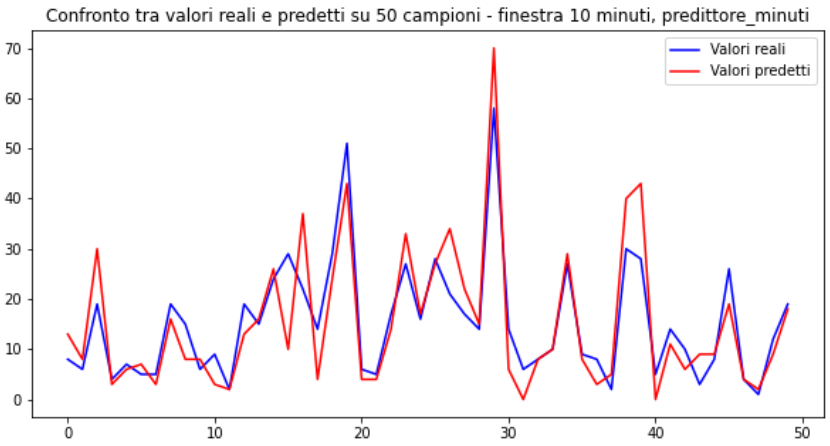


Figura 3.3: *Confronto tra valori reali e predetti sul dataset con finestra di 10 minuti con predittore predittore_minuti.*

Finestre di 60 minuti

Con questo valore delle finestre l'errore diminuisce sensibilmente rispetto al caso precedente, passando del 39,1% al 7,8%. Questo netto miglioramento è visibile confrontando il grafico nella Figura 3.3 con il grafico nella Figura 3.4 dove la funzione relativa ai valori predetti riesce ad approssimare molto bene la funzione dei valori reali.

Finestre di 120 minuti

Come supposto inizialmente, questo valore per le finestre è quello che garantisce i risultati migliori portando l'errore al 5,4%. Questo è dovuto al fatto che, grazie al valore maggiore delle finestre, ogni entry del dataset è più ricca di informazioni e, inoltre, influenza i valori di "mail_count_past" e "mail_count_future" di più entry rispetto ai casi precedenti.
Questa ricchezza di informazione porta il modello a trovare delle relazioni solide tra i dati e, quindi, a performare meglio rispetto ai casi discussi in precedenza.

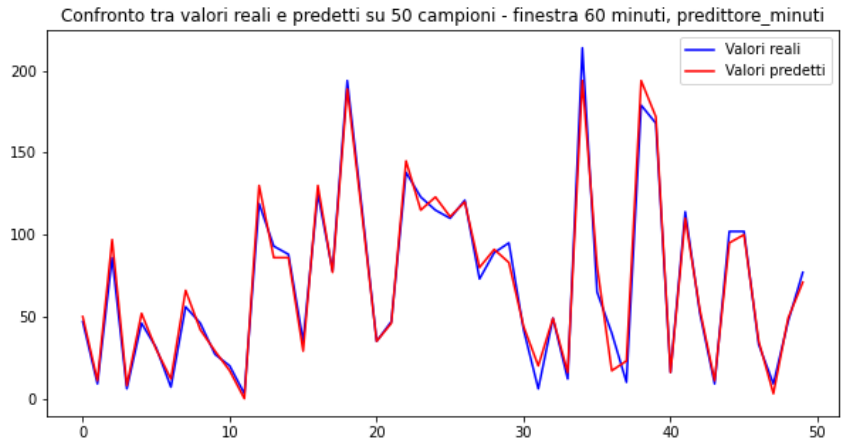


Figura 3.4: *Confronto tra valori reali e predetti sul dataset con finestra di 60 minuti con predittore predittore_ minuti.*

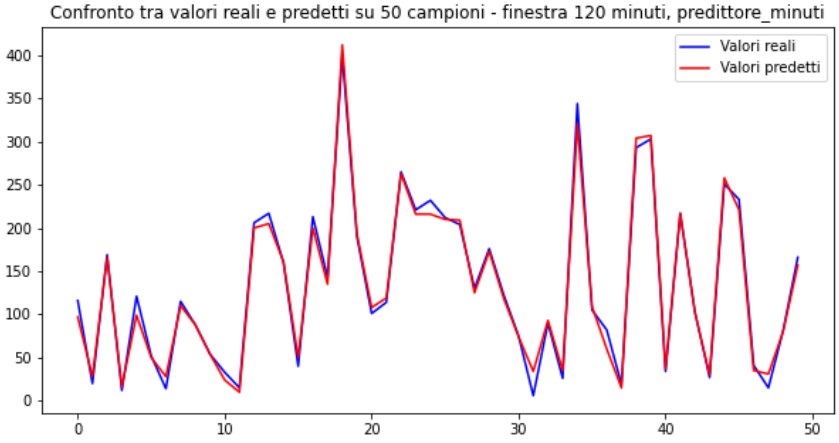


Figura 3.5: *Confronto tra valori reali e predetti sul dataset con finestra di 120 minuti con predittore predittore_ minuti.*

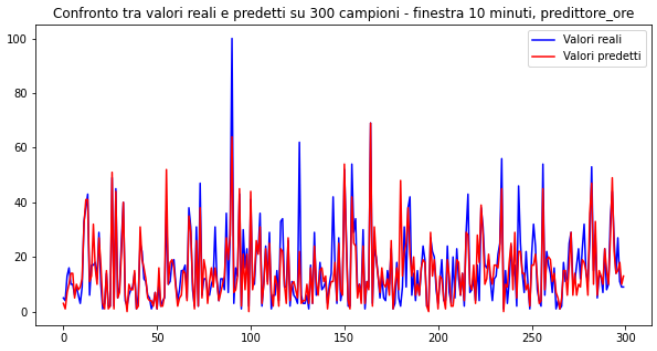
3.3 Risultati finali

Riassumendo, com'è visibile nella Tabella 3.2, i risultati migliori sono stati raggiunti utilizzando *predittore_ minuti* e il dataset con le finestre *window_ past* e *window_ future* impostate a 120 minuti. Questa combinazione è quella che offre il miglior trade-off tra il numero di feature in input e valore basso dell'errore.

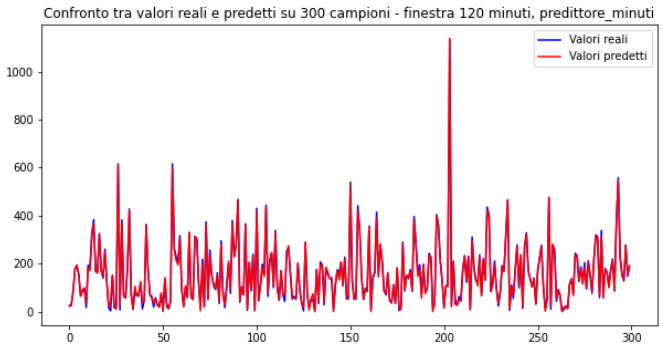
Nella Tabella 3.2, per completezza, sono riportati gli errori di tutte le nove possibili configurazioni, mentre, nel grafico della Figura 3.6, sono confrontate le predizioni su 300 campioni del modello peggiore (a sinistra) con quelle del modello migliore (a destra).

	Finestre di 10 minuti	Finestre di 60 minuti	Finestre di 120 minuti
predittore_ore	45,7%	11,4%	6,1%
predittore_minuti	39,1%	7,8%	5,4%
predittore_secondi	41,1%	7,8%	5,4%

Tabella 3.2: *Tabella riassuntiva degli errori dei vari predittori con in input i diversi dataset*



(a) *predittore_ore*,
dataset con finestre di 10 minuti



(b) *predittore_minuti*,
dataset con finestre di 120 minuti

Figura 3.6: *Confronto tra le predizioni del modello peggiore 4.3a e migliore 4.3b*

4. Global scaling e predictive global scaling

Il linguaggio utilizzato per le simulazioni è ABS, un linguaggio *actor-based* e *object-oriented* con una sintassi molto simile a Java. *Timed ABS* è un'estenzione del linguaggio che permette di introdurre la nozione di *tempo astratto*. Nello specifico, a *run-time* il tempo è espresso come il numero di unità di tempo passato dall'avvio del sistema. Nel caso delle simulazioni riportate di seguito, un'unità di tempo corrsiponde a $\frac{1}{30}$ di secondo.

Data la vastità dell'argomento e perché l'obiettivo di questa tesi non è spiegare il linguaggio, per approfondimenti sul tema e sul perché risulti utile per simulare questa tipologia di sistemi rimandiamo alla documentazione ufficiale di ABS [1] e alla tesi del dott. Lorenzo Bacchiani [2].

4.1 Email Pipeline Processing System

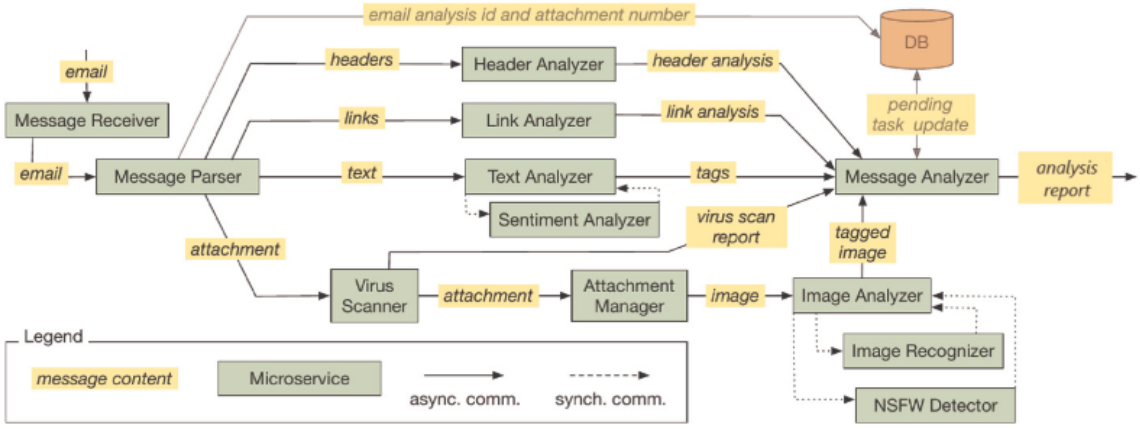


Figura 4.1: *Struttura della email pipeline. Immagine presa da [3]*

L'*email processing pipeline* è composta da 12 tipi di microservizi, ognuno dei quali ha il proprio *load balancer* che viene utilizzato per distribuire le richieste su un insieme di istanze che possono essere aumentate o diminuite al bisogno [2].
Per poter effettuare le simulazioni, i microservizi e i corrispondenti load balancer, sono stati rappresentati come classi di ABS.
Inoltre, nel progetto sono state create altre classi, tra le quali:

- *MailGenerator*: utile per inviare mail al sistema simulando uno specifico workload;
- *SetUpSystem*: utilizzato per creare e inizializzare il sistema;
- Il *Monitor*: usato per controllare periodicamente il sistema e scalare up o down quando necessario utilizzando l'algritmo di global scaling.

4.2 Refactoring del codice

Per poter implementare il *predictive global scaling*, sono state necessarie alcune modifice al codice che implementava il global scaling tradizionale:

- Modifica della lista del *real workload*, la lista delle email in entrata nel sistema
- Cambio del modo in cui il sistema gestisce l'inbound workload per fare predictive scaling
- Cambiamento del metodo di setup del sistema
- Modifica concettuale dell'architettura di sistema

4.2.1 Il nuovo real workload

I valori del real workload sono stati selezionati dalla colonna "mail_count_future" del test set. Nello specifico, abbiamo ordinato il test set in ordine cronologico, così da avere degli istanti di tempo continui e, successivamente, abbiamo fatto delle predizioni su questi dati utilizzando il modello allenato.

Per riuscire a simulare con dei valori consoni, abbiamo moltiplicato sia i valori reali che quelli predetti per 2,5 (§ 1.1). Inoltre, i nostri dati sono espressi in $\frac{mail}{ora}$ e il sistema richiede $\frac{mail}{secondo}$, ma questo non è un problema, perché con un tempo simulato possiamo supporre che il pattern delle email di Enron vada bene per le nostre simulazioni.

4.2.2 Modifica alla gestione dell'inbound workload

Nella versione di global scaling senza il predittore, l'inbound workload fungeva da contatore delle email in entrata nel sistema. In questa versione, invece, l'inbound workload è stato trasformato in una lista contenente i valori predetti e viene utilizzata per decidere in anticipo se il sistema deve scalare.

Ora lo scaling avviene una *time unit* prima che il workload che richiede lo scaling arrivi. Prima, invece, il sistema aspettava un determinato numero di email per effettuare lo scaling, rischiando o di avere prestazioni troppo elevate andando ad aumentare i costi o, viceversa, prestazioni troppo basse andando ad aumentare la latenza e i messaggi persi.

La scelta di utilizzare le predizioni come una lista hard-coded, invece che immettere dinamicamente i dati, per esempio tramite le API di ABS [1], è stata fatta perché si sarebbe potuta creare confusione facendo sembrare la simulazione real-time, quando invece il tempo era simulato.

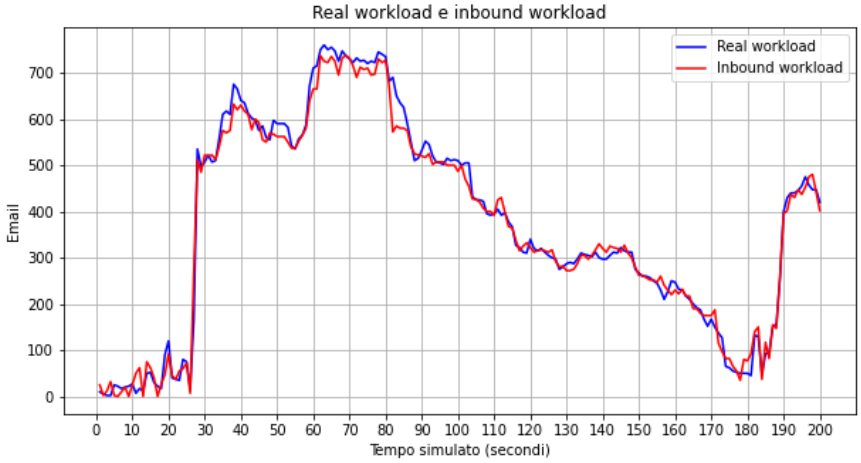


Figura 4.2: *Confronto tra inbound workload e real workload*

4.2.3 Modifiche al metodo di setup del sistema

Anche il setup del sistema è stato modificato per adattarsi al predictive global scaling. Infatti, ora il setup del sistema sfrutta la prima previsione per calcolare la prima orchestrazione, mentre nella versione precedente, come prima orchestrazzione, veniva utilizzata quella di base, contenente un'istanza per ciascun microservizio.

4.2.4 Modifica dell'architettura del sistema

Ora è presente un servizio apposito che implementa l'algoritmo di global scaling. Nella versione precedente, tale algoritmo era implementate dal monitor, che ora si limita a sfruttare questo servizio se rileva che è necessario effettuare lo scaling.

4.3 Global scaling vs Predictive global scaling

Per confrontare i risultati ottenuti dalle due diverse implementazioni verranno analizzate due metriche:

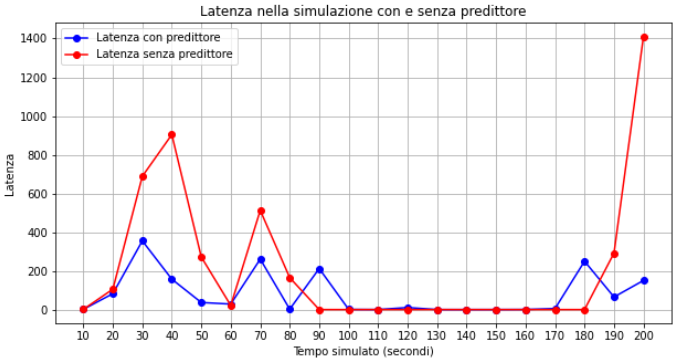
- *Latenza*, ovvero il tempo medio per processare completamente una email che entra nel sistema;
- *Messagi persi*.

Come visibile nella Figura 4.3, il predictive global scaling ha prestazioni migliori sia sulla *latenza* che sui *messaggi persi*. In particolare, da *tempo = 20* a *tempo = 60*, dove viene registrato un brusco aumento delle email (Figura 4.2), il sistema con il predittore riesce a tenere i due parametri analizzati nettamente più bassi.
La stessa cosa è notabile tra i tempi 190 e 200 nei quali, a fronte di un nuovo rapido aumento nel numero di email, il sistema senza predittore raggiunge il picco di latenza pari a 1411, mentre il sistema con il predittore la mantiene a 150, un valore molto più basso.

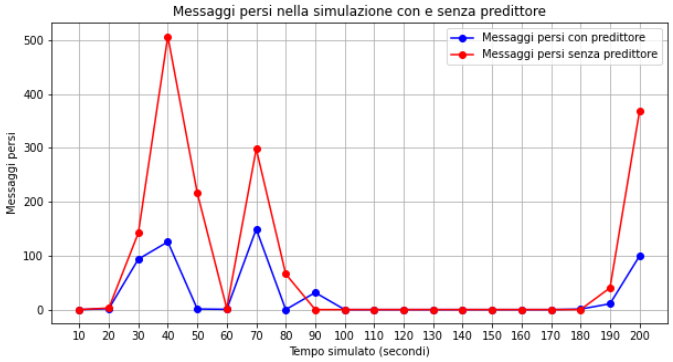
Nonostante ai tempi 90 e 180 per la latenza, e 90 per i messaggi, la simulazione senza predittore risulti migliore, la latenza media e il numero di messaggi persi medi risultano comunque migliori nel sistema con il predittore (Tabella 4.1)

	Latenza media	Media messaggi persi
Simulazione con predittore	80,95	25,73
Simulazione senza predittore	218,65	82,12

Tabella 4.1: *Tabella con latenza media e media messaggi persi da tempo 10 a tempo 200*



(a) *Latenza*



(b) *Messaggi persi*

Figura 4.3: *Confronto tra i sistemi con e senza predittore di latenza 4.3a e messaggi persi 4.3b*

Conclusioni

In questo elaborato, abbiamo ampliato il lavoro già discusso in [2, 3]. Nello specifico, abbiamo aggiornato il sistema di global scaling, inserendo delle previsioni ottenute tramite un modello di deep learning, per ottenere il predictive auto scaling. I risultati ottenuti, discussi nel capitolo 4, mostrano come il predictive global scaling performi meglio sia dal punto di vista della latenza, che dei messaggi persi, rispetto al global scaling senza predizioni.

Il codice, utile a fare le simulazioni e ad allenare la rete neurale utilizzata in questa tesi, è disponibile nel repository GitHub del dott. Lorenzo Bacchiani [13].

Personalmente, ho trovato questo lavoro una grande opportunita. Mi ha permesso in primis di studiare e utilizzare una rete neurale, un argomento non trattato durante la triennale, così da convincermi a preseguire gli studi specializzandomi in questo settore. Oltre a ciò, per la prima volta ho potuto contribuire a un progetto di ricerca, sperimentando varie soluzioni fino a trovare quella più adatta al problema, permettedomi di crescere e capire, almeno in parte, come si fa ricerca.

Bibliografia

[1] *Abs language*. URL: https://abs-models.org/.

[2] Lorenzo Bacchiani. "Microservice Dynamic Architecture-Level Deployment Orchestration". URL: http://amslaurea.unibo.it/21412/.

[3] Lorenzo Bacchiani et al. "Microservice Dynamic Architecture-Level Deployment Orchestration". In: *International Conference on Coordination Languages and Models*. Springer. 2021, pp. 257–275.

[4] *Batch, Mini Batch & Stochastic Gradient Descent*. URL: https://towardsdatascience.com/batch-mini-batch-stochastic-gradient-descent-7a62ecba642a.

[5] *Codici formato data*. URL: https://docs.python.org/3/library/datetime.html.

[6] *csv — CSV File Reading and Writing*. URL: https://docs.python.org/3/library/csv.html.

[7] *Enron Email Dataset*. URL: https://www.cs.cmu.edu/~./enron/.

[8] Diederik P Kingma e Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[9] *os.path — Common pathname manipulations*. URL: https://docs.python.org/3/library/os.path.html.

[10] *Pandas*. URL: https://pandas.pydata.org/docs/index.html.

[11] *pandas.get_dummies*. URL: https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html.

[12] *pandas.to_datetime*. URL: https://pandas.pydata.org/docs/reference/api/pandas.to_datetime.html.

[13] *Repository GitHub con codice per simulazioni e per allenare la rete neurale*. URL: https://github.com/LBacchiani/ABS-Simulations-Comparison.

[14] Tomasz Szandała. "Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks". In: *Bio-inspired Neurocomputing*. Springer, 2021, pp. 203–224.

[15] *Train-Test Split for Evaluating Machine Learning Algorithms*. URL: https://machinelearningmastery.com/train-test-split-for-evaluating-machine-learning-algorithms/.

Ringraziamenti

Vorrei prendere quest'ultima pagina per ringraziare tutte le persone che sono state per me fondamentali, non solo per riuscire a scrivere questo elaborato, ma per essere riuscito ad arrivare alla fine di questo percorso di tre anni che non sono stati un semplice percorso di studi, ma una vera crescita.

Voglio intanto ringraziare il prof. Maurizio Gabbrielli, per avermi dato l'opportunità di prendere parte a questo interessantissimo progetto, e i miei correlatori, il dott. Stefano Pio Zingaro e il dott. Lorenzo Bacchiani, per avermi dato il loro tempo, gli strumenti e tutte le spiegazioni di cui ho necessitato per lavorare.

Passo poi ai miei genitori, che ringrazio per aver creduto nella mie possibilità e per il sostegno economico necessario per poter pagare affitto e studi.

Ci tengo anche a ringraziare con tutto il mio cuore la mia amica Elena, per esserci sempre stata, per avermi fatto ritrovare il sorriso e la voglia di fare anche nei momenti più difficili di questi tre anni e per preoccuparsi sempre per me, anche quando è l'unica a capire che c'è qualcosa che non va.

Un ringraziamento anche ai miei due coinquilini Elia e Andrea, per avermi sopportato, aiutato, nello studio, come nelle difficoltà di tutti i giorni e per avermi scelto per fare insieme tutti i progetti di questa laurea.

Un grazie speciale anche a Clara, per avermi fatto capire l'importanza di lottare per i propri sogni, a Stefano, per avermi insegnato che non è mai troppo tardi per cambiare strada e a Eleonora, se ho passione per lo studio, in particolare per l'informatica, è anche grazie a te.

Per ultima, ma non per importanza, vorrei ringraziare la mia psicologa Claudia, per avermi sostenuto durante dei mesi difficili, per avermi fatto credere nelle mie capacità e per avermi fatto capire che ho tutte le risorse necessarie per riuscire a fare quello che voglio.