UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica e dell'Automazione

---

Euristiche di ottimizzazione guidate da tecniche di Machine Learning

Optimisation heuristics driven by Machine Learning techniques

Relatore:

Prof. Domenico Potena

Correlatore:

Prof. Alex Mircoli

Laureando:

Enrico Pio Martino

Anno Accademico 2021/2022

A zio Lino

Indice

1 Introduzione **5**
1.1 Obiettivo 6
1.2 Struttura della Tesi 7

2 Sorgente Dati **9**
2.1 Dataset 9
2.1.1 Initial Solution 11
2.1.2 Moves 12
2.1.3 OF_Diff 13
2.2 EDA 14
2.2.1 Initial Solution 14
2.2.2 Moves 17
2.2.3 OF_Diff 18

3 Architetture **21**
3.1 Long Short Term Memory 21
3.1.1 Struttura di una rete LSTM 23
3.1.2 Problemi delle LSTM 24
3.2 Transformer 25
3.2.1 Struttura di una rete Transformer 26
3.3 Graph Neural Network 28
3.3.1 Caratterizzazione di un grafo 28
3.3.2 Struttura di una Graph Neural Network 31

4 Implementazione **33**
4.1 LSTM 34
4.2 Transformer 35
4.3 GNN 36

5 Risultati **39**
5.1 Setup sperimentale 39
5.1.1 Dataset utilizzati 39
5.1.2 Iperparametri 41
5.1.3 Metriche di valutazione 44

5.1.4 Tecniche di validazione 45
5.2 Risultati ottenuti 46
5.2.1 Risultati Long Short Term Memory 46
5.2.2 Risultati Transformer 48
5.2.3 Risultati Graph Neural Network 51
5.3 Confronto 54

6 Conclusione e sviluppi futuri 57

7 Appendice 59

Bibliografia e Sitografia 63

Elenco delle figure 65

Elenco delle tabelle 68

Capitolo 1

Introduzione

In questa tesi verrà esplorato l'utilizzo di reti neurali per la predizione delle mosse da utilizzare in euristiche per l'ottimizzazione del routing di veicoli elettrici. Le reti neurali sono una classe di algoritmi di apprendimento automatico che hanno dimostrato di essere molto efficaci in una vasta gamma di applicazioni, tra cui la classificazione, la regressione e la generazione di contenuti.
Inoltre, nel campo dell'ottimizzazione, le reti neurali possono essere utilizzate per supportare le euristiche, ovvero metodi approssimativi per trovare soluzioni ottimali in problemi in cui la soluzione esatta è troppo costosa o impossibile da calcolare.
La combinazione di reti neurali ed euristiche di ottimizzazione permette di risolvere problemi complessi in maniera efficiente ed efficace.
Nella tesi verranno valutate diverse architetture di reti neurali per trovare un metodo valido per classificare se una certa mossa migliora o peggiora una certa funzione obiettivo. Riuscire a classificare in maniera opportuna se una mossa accresce o riduce il valore di una funzione obiettivo permette di evitare di dover provare tutte le possibili combinazioni, restringendo il campo di test alle sole mosse che incrementano la funzione obiettivo, essendo le mosse dell'algoritmo precedentemente classificate come "buone" o "cattive".
L'idea è quella di sfruttare le tecniche di classificazione per sostenere l'euristica, affinché si ottimizzi la consegna di merce che avviene utilizzando dei veicoli elettrici. Questi ultimi partono da un deposito e devono raggiungere dei clienti, passando per delle stazioni di ricarica.

1.1 Obiettivo

L'obiettivo del lavoro di tesi è quello di studiare e testare varie architetture di rete neurale al fine di riuscire a classificare se un dato algoritmo di *destroy-repair*, applicato a una soluzione iniziale, migliora o peggiora la funzione obiettivo. A tal fine è stato necessario suddividere il lavoro in più fasi.
In particolare, la prima fase riguarda l'esplorazione del dataset, affinché sia possibile conoscere in maniera approfondita i dati e calcolare alcune statistiche come: media, varianza, deviazione standard, il minimo e il massimo di alcuni attributi delle istanze del dataset.
Nella seconda fase si vanno a studiare dal punto di vista teorico le varie architetture di rete neurale da utilizzare. In particolare, si andranno a studiare tre reti neurali:

- LSTM: Long Short Term Memory;
- Transformer;
- GNN: Graph Neural Network.

La terza e quarta fase riguardano invece l'implementazione delle architetture appena elencate e l'addestramento delle stesse suddividendo il dataset in *training* e *testing*, valutando le metriche di prestazione del modello come F1-Score e Accuracy nei diversi test effettuati. Infine, a partire dai risultati ottenuti, si andrà a definire quale modello di rete neurale si presta meglio al task di classificazione.

1.2 Struttura della Tesi

Il lavoro di tesi è strutturato nel seguente modo: nel secondo capitolo viene inizialmente illustrato il dataset impiegato nell'addestramento delle reti. Successivamente segue una fase di **EDA**, *Exploratory data analysis*, che è un approccio per l'analisi di dati orientato a riassumerne le caratteristiche principali.
Il terzo capitolo riguarda la parte più teorica della tesi. Vengono esplorate le tre architetture utilizzate, puntando il focus sulle differenze sostanziali tra di esse.
Il quarto capitolo è quello relativo all'implementazione in *Python* delle differenti reti neurali, riportando le versioni software delle librerie utilizzate e i punti salienti del codice implementativo.
Il quinto capitolo contiene una sezione chiamata *Setup Sperimentale*, nella quale vengono definiti: i dataset utilizzati, le combinazioni di iperparametri, le metriche di valutazione e le tecniche di validazione delle 3 reti neurale. Successivamente vengono valutati i risultati dei vari test effettuati per le 3 reti.
L'ultimo capitolo riporta le conclusioni e offre degli spunti per degli sviluppi futuri.
Inoltre in appendice vengono mostrate alcune rappresentazioni della struttura dei dati su cui si è lavorato.

Capitolo 2

Sorgente Dati

Nel seguente capitolo si andrà ad approfondire il set di dati utilizzato in tutti gli esperimenti eseguiti ai fini della tesi.
In generale il dataset contiene informazioni riguardanti lo spostamento di veicoli elettrici per il trasporto di merce da un deposito a dei clienti, passando per delle stazioni di ricarica. Come vedremo nel dettaglio nella sezione 2.1, in un campo del dataset è presente una serie di nodi di tre tipi: **C** cliente, **D** deposito e **S** stazione di ricarica. I percorsi del veicolo saranno del tipo: partenza dal deposito **D**, consegna a vari clienti $\mathbf{C_n}$, passando per le stazioni di ricarica $\mathbf{S_n}$.
L'obiettivo sarà quello di classificare se una serie di mosse *destroy-repair*, applicata a una soluzione iniziale, migliori la funzione obiettivo di partenza. Tale classificazione può affiancare un'euristica di ottimizzazione, la quale non applicherà una cancellazione/creazione di un nodo in maniera casuale, ma sceglierà la mossa che è classificata come migliorante. Tutto ciò permette di agevolare l'euristica nella scelta dei nodi, rendendo più efficiente il raggiungimento dell'ottimo.
Nelle seguenti sezioni, si andranno prima a elencare e spiegare tutti i campi presenti nel file *csv* disponibile al seguente link *GitHub:* DB-Output.csv, in secondo luogo seguirà una sezione di *EDA* (Exploratory data analysis), che è fondamentale per conoscere profondamente il dataset tramite metriche statistiche e operazioni sul dataframe.

2.1 Dataset

Il dataset utilizzato ai fini della tesi è stato generato da un gruppo di ragazzi per il progetto relativo al corso di *Big Data Analytics e Machine Learning*. Il set di dati contiene 21157 istanze, ognuna con 24 attributi. Sono dunque presenti 21157 righe e 24 colonne. Di seguito vengono riportati i nomi degli attributi e il relativo significato:

1. **Instance's Name**: nome del file di input;

2. **Initial Solution**: array rappresentante l'insieme delle rotte;

3. **OFIS**: funzione obiettivo iniziale;
4. **Moves**: algoritmi destroy-repair utilizzati (station destroy, station repair, customer destroy, customer repair);
5. **OFFS**: funzione obiettivo a seguito delle mosse applicate;
6. **OF_Diff**: differenza tra funzione obiettivo finale ed iniziale per l'iterazione corrente;
7. **Exe_Time_d-r**: tempo di elaborazione per applicare le mosse;
8. **Avg_Battery_Status**: media del consumo di batteria dei veicoli;
9. **Avg_SoC**: media degli stati della batteria con i quali i veicoli tornano in deposito;
10. **Avg_Num_Charge**: media del numero di cariche effettuate dai veicoli;
11. **Avg_Vehicle_Capacity**: media della capacità dei veicoli;
12. **Avg_Customer_Demand**: media delle domande dei customer in termini di package weight;
13. **Num_Vehicles**: numero di veicoli utilizzati;
14. **Avg_Service_Time**: media con la quale ogni customer viene servito;
15. **Avg_Customer_TimeWindow**: media delle differenze tra DueDate e ReadyTime dei customer;
16. **Var_Customer_TimeWindow**: varianza delle differenze tra DueDate e ReadyTime dei customer;
17. **Avg_Customer_customer_min_dist**: media delle distanze tra ciascun cliente e quello più vicino;
18. **Var_Customer_customer_min_dist**: varianza delle distanze tra ciascun cliente e quello più vicino;
19. **Avg_Customer_station_min_dist**: media delle distanze tra ciascun cliente e la stazione più vicina;
20. **Var_Customer_station_min_dist**: varianza delle distanze tra ciascun cliente e la stazione più vicina;
21. **Avg_Customer_deposit_dist**: media delle distanze tra ciascun cliente e il deposito;

22. **Var_Customer_deposit_dist**: varianza delle distanze tra ciascun cliente e il deposito;

23. **CounterD_R**: vettore composto dai 17 possibili algoritmi che si possono applicare, in cui si conta quante volte ciascun algoritmo ha permesso di migliorare il valore della soluzione precedente;

24. **CounterD_Rlast**: vettore composto dai 17 possibili algoritmi che si possono applicare, in cui si conta quante volte ciascun algoritmo è stato l'ultimo, in ciascun ciclo, ad aver migliorato la soluzione (quindi ad aver portato alla funzione obiettivo migliore).

Per il lavoro di tesi, il focus è andato su alcuni dei 24 campi appena elencati. Questi sono: **Initial Solution**, **Moves** e **OF_Diff**.

2.1.1 Initial Solution

Il campo "Initial Solution" è di tipo *String Object*, cioè un oggetto Python di tipo *Stringa*.
Tale stringa rappresenta una lista principale che ha al suo interno N liste secondarie. Queste ultime contengono a loro volta una sequenza di nodi di tre tipologie:

- **D**: Deposito;
- **C**: Cliente;
- **S**: Stazione di ricarica;

Ogni nodo avrà una certa cardinalità, in base al numero di depositi, clienti e stazioni di ricarica presenti nel dataset. Il numero di elementi per ogni tipologia verrà riportata e approfondita nella sezione 2.2.
Di seguito è riportato un porzione di un'istanza del dataset relativo al campo *Initial Solution*:

"[['D0', 'C21', 'C26', 'D0'], ['D0', 'C24', 'C50', 'D0'], ['D0', 'C32', 'S11', 'D0'], ['D0', 'C9', 'D0'], ['D0', 'C90', 'C85', 'D0']]".

Ogni lista, contenuta nella lista primaria, è quindi una sequenza di nodi che rappresenta un percorso che parte e termina nel nodo $\mathbf{D_0}$, cioè il deposito, passando per i Clienti $\mathbf{C_n}$ e stazioni di ricarica $\mathbf{S_n}$.

2.1.2 Moves

Il campo "Moves" è anch'esso di tipo *String Object*. La stringa contiene un'unica lista nella quale è presente la sequenza di *mosse* attuate sui nodi del campo *Initial Solution*. Ogni mossa è un algoritmo di *destroy-repair*, cioè distruzione e ricostruzione di un nodo del grafo. Tale approccio è classico in problemi di **ALNS**, *Adaptive Large Neighborhood Search*. L'idea generale è quella di rimuovere ripetutamente delle istanze dalla soluzione attuale e reinserirle in una posizione più vantaggiose o più redditizie.
Un esempio di tale funzionamento è mostrato nella seguente figura:

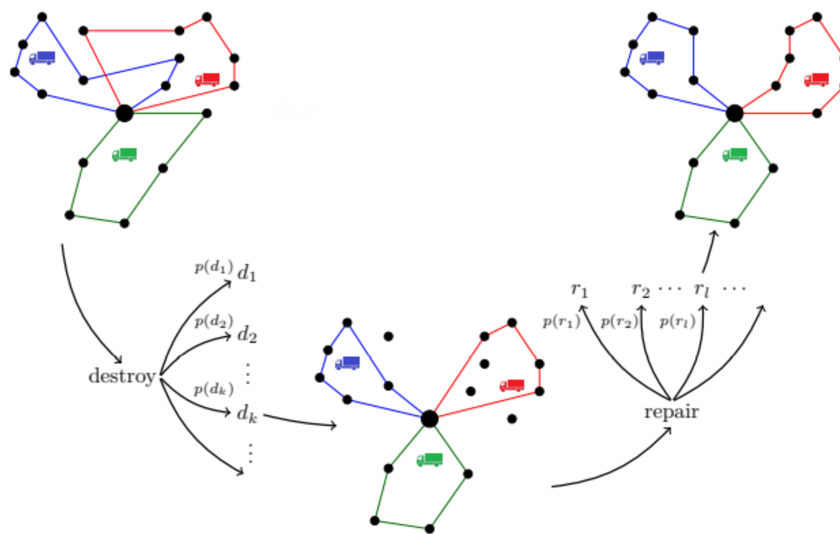


Figura 2.1: La figura illustra il funzionamento di un algoritmo di *destroy-repair*. Fonte: [1]

Come mostrato in figura 2.1, una mossa di *destroy-repair* distrugge o ricostruisce gli archi che collegano i nodi in modo da trovare una soluzione migliore.
Per modificare gli archi, quindi i tragitti da compiere, si ha una combinazione di 4 mosse per ogni istanza del dataset.
Le possibili mosse sono le seguenti:

1. null;
2. GreedyRouteRemoval.

Per i nodi di tipo **Customer**:

1. ShawDestroyCustomer;
2. TimeBasedDestroyCustomer;
3. GreedyRepairCustomer;

4. WorstDistanceDestroyCustomer;
5. RandomRouteDestroyCustomer;
6. DemandBasedDestroyCustomer;
7. ProximityBasedDestroyCustomer;
8. ProbabilisticWorstRemovalCustomer;
9. GreedyDestroyCustomer;
10. WorstTimeDestroyCustomer;
11. ZoneDestroyCustomer.

Per i nodi di tipo **Station**:

1. LongestWaitingTimeDestroyStation;
2. DeterministicBestRepairStation;
3. ProbabilisticBestRepairStation;
4. RandomDestroyStation.

Si hanno quindi delle mosse più generali (null e GreedyRouteRemoval) e delle mosse specifiche per i nodi di tipo Cliente (*Customer*) e per le Stazioni di ricarica (*Station*).

2.1.3 OF_Diff

Il campo "OF_Diff" è di tipo *Float*, cioè un numero in virgola mobile. Questo valore rappresenta la differenza tra i due campi del dataset: "OFFS" e "OFIS", ovvero la funzione obiettivo a seguito delle mosse applicate meno la funzione obiettivo iniziale. *OF_Diff* è quindi una cifra fondamentale per misurare e classificare se una mossa porta a un miglioramento o meno della funzione obiettivo. Non è da escludere che una certa sequenza di mosse possa lasciare invariata la funzione obiettivo, questo perché l'eliminazione o ricostruzione di un nodo potrebbe non influenzare il grafo in considerazione.

2.2 EDA

Il termine *EDA* sta per "Exploratory Data Analysis". L'analisi esplorativa dei dati si riferisce al processo critico di esecuzione di indagini iniziali sui dati per scoprire modelli, individuare anomalie, testare ipotesi e verificarle con l'aiuto di statistiche di sintesi [2]. L'esplorazione dei dati permette di conoscere a fondo le caratteristiche del dataset e dei suoi campi. In particolare anche per questa fase il focus è sui seguenti campi: **Initial Solution**, **Moves** e **OF_Diff**.

2.2.1 Initial Solution

Il primo studio eseguito sul dataset è quello relativo al campo "Initial Solution". Da una parte è noto che il numero di **Depositi** sia **1** ed è chiamato $\mathbf{D_0}$, d'altra parte si vanno a calcolare il numero di Clienti e di Stazioni di ricarica. Il codice *Python* per calcolare la cardinalità massima dei nodi *Customer* e *Station* è il seguente:

```
1 import pandas as pd
2 from numpy import var
3 import statistics
4 df = pd.read_csv('1-EDA\DB-Output.csv')
5 list_of_nodes=[]
6 for i in df.index:
7     clean=df['Initial Solution'][i].replace("[","").replace
    ("]","").replace("'","")
8     list_of_strings = clean.split(", ")
9     list_of_nodes.extend(list_of_strings)
10 customers=[node for node in list_of_nodes if node[0]== "C"]
11 numbers_customers = [int(num[1:]) for num in customers]
12 stations=[node for node in list_of_nodes if node[0]== "S"]
13 numbers_stations=[int(num[1:]) for num in stations]
14
15 print(max(numbers_customers),max(numbers_stations))
```

L'output del precedente codice è il seguente:

```
1 100 20
```

Dunque il numero di **Clienti** (nodi *Customer*) è **100** e il numero di **Stazioni** (nodi *Station*) è **20**.
Il secondo studio è stato quello di misurare il **numero di liste** di nodi contenute nella lista principale e il **numero di nodi** in ogni lista. Come visto nella sezione 2.1.1, ogni istanza del campo "Initial Solution" è del tipo:

"[['D0', 'C21', 'C26', 'D0'], ['D0', 'C24', 'C50', 'D0'], ['D0', 'C32', 'S11', 'D0'], ['D0', 'C9', 'D0'], ['D0', 'C90', 'C85', 'D0']]".

Del campo "Initial Solution" si è calcolato:

- Minimo numero di liste: 5;
- Massimo numero di liste: 57;
- Media del numero di liste: 24.572;
- Deviazione standard del numero di liste: 14.645;
- Minimo numero di nodi in una lista: 2;
- Massimo numero di nodi in una lista: 15;
- Media del numero di nodi in una lista: 4.842;
- Deviazione standard del numero nodi in una lista: 1.337.

Da questi risultati si evince che ogni istanza del campo "Initial Solution" contiene un numero variabile di liste che va da 5 a 57. In media sono presenti circa 25 liste di nodi per ogni istanza del dataset.
Per quanto riguarda il numero di nodi presenti in ogni lista, questi vanno da un minimo di 2 elementi (nessun percorso compiuto, $D_0 \rightarrow D_0$) a un massimo di 15 nodi, con una media di circa 5 nodi per ogni lista. Ogni lista contenuta nella lista principale indica un percorso che inizia dal deposito **D0**, attraversa i nodi *Customer* e *Station* e termina nel nodo **D0**. Un percorso è rappresentabile da un grafo, argomento del quale parleremo nella sezione 3.3.1. Per rappresentare la soluzione iniziale tramite un grafo si è utilizzata la libreria *NetworkX* di Python. Di seguito viene riportata la funzione che genera la rappresentazione del grafo:

```
1 def draw(G):
2     pos=nx.kamada_kawai_layout(G)
3     plt.figure(figsize=(10,8),dpi=200.0)
4     color_map = []
5     node_labels=nx.get_node_attributes(G, 'attribute')
6     for label in node_labels.values():
7       if label[0]=='D':
8         color_map.append('#ED173A')
9       if label[0]=='C':
10        color_map.append('#00AAD9') #1f77b4 dafault
11      if label[0]=='S':
12        color_map.append('#9AB550')
13    nodes = nx.draw_networkx_nodes(G, pos,node_size=500,
   node_color=color_map)
14    labels = nx.draw_networkx_labels(G, pos,labels=node_labels
   )
15    edges = nx.draw_networkx_edges(G, pos,width=0.5,arrowsize
   =6,node_size=500)
```

```
16     pt='/content/drive/MyDrive/Tirocinio/0_NO_attrib_mio/
     results/draw/'+file_name
17     isExist = os.path.exists(pt)
18     if not isExist:
19       os.mkdir(pt)
20     path_draw=pt+'/'+str(i)+'.png'
21     plt.savefig(path_draw)
22     plt.axis('off')
23     plt.show()
```

Per ogni istanza del dataset quindi possiamo creare un grafo che rappresenti la soluzione iniziale. Esso sarà dato da un insieme di percorsi, uno per ogni lista presente nel campo "Initial Solution". Di seguito viene mostrato l'output del *plot*:

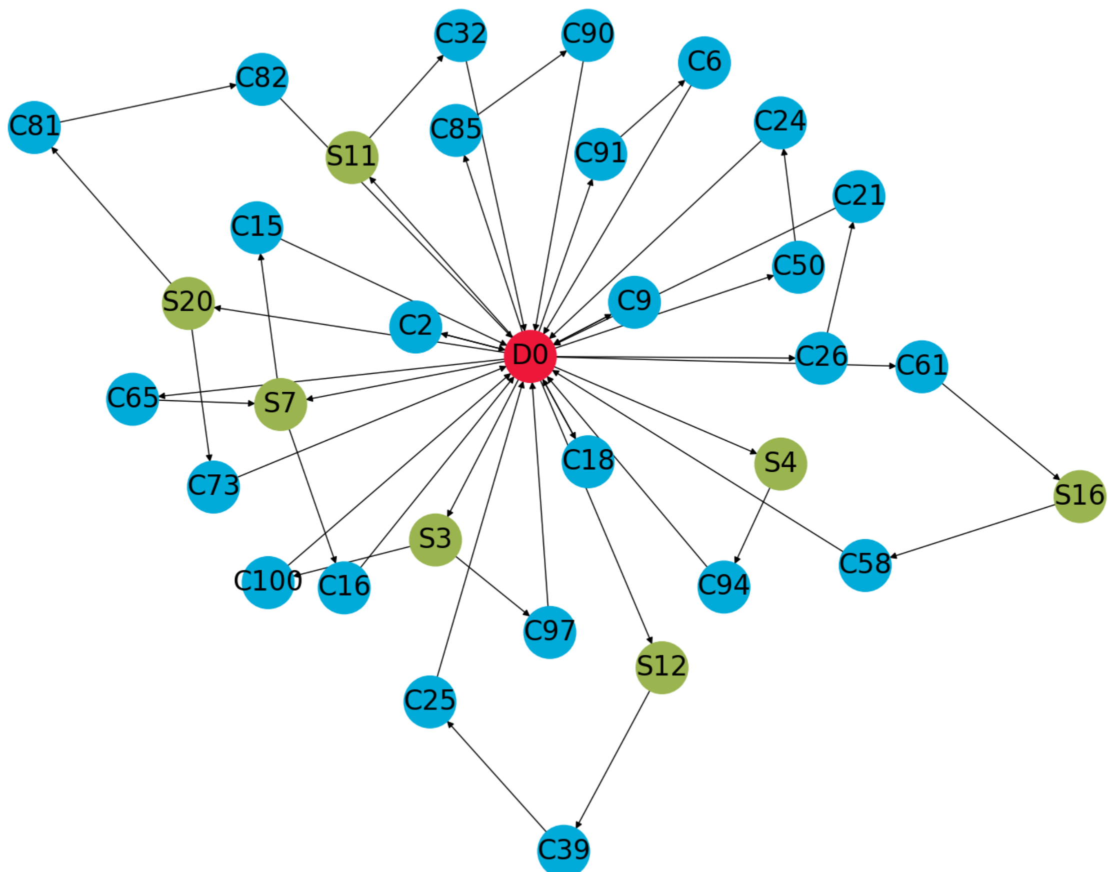


Figura 2.2: Figura di esempio di un grafo ottenuto dalla funzione **draw(G)**. Essa rappresenta graficamente l'insieme percorsi relativi a un'istanza del dataset, quindi a una riga del file DB-Output.csv. Il dataset contiene 21157 grafi, ognuno dei quali è più o meno complesso come quello in figura. Nell'Appendice verranno mostrate altre rappresentazioni di grafi presenti nel dataset.

2.2.2 Moves

Per quanto riguarda il campo "Moves" del dataset, si è voluto determinare il numero di volte in cui una mossa è presente in tutte le istanze del dataset. Il risultato è mostrato nella seguente tabella:

GreedyRepairCustomer	21157
null	16960
DeterministicBestRepairStation	2157
RandomDestroyStation	2116
LongestWaitingTimeDestroyStation	2081
ProbabilisticBestRepairStation	2040
TimeBasedDestroyCustomer	2008
ShawDestroyCustomer	1967
RandomRouteDestroyCustomer	1940
DemandBasedDestroyCustomer	1938
GreedyDestroyCustomer	1935
ZoneDestroyCustomer	1923
GreedyRouteRemoval	1919
ProximityBasedDestroyCustomer	1899
WorstTimeDestroyCustomer	1895
WorstDistanceDestroyCustomer	1871
ProbabilisticWorstRemovalCustomer	1862

La tabella riporta la frequenza, in ordine decrescente, delle mosse presenti nelle istanze del set di dati. Si evince che in tutte le istanze è presente la mossa "GreedyRepairCustomer", al secondo posto la mossa più frequente è "null" con 16920 presenze. Inoltre, ogni volta in cui la mossa "null" si presenta, essa è presente per due volte consecutive, di seguito è riportato un esempio:

"['null', 'null', 'GreedyDestroyCustomer', 'GreedyRepairCustomer']"

L'ultima caratteristica verificata del campo "Moves" è il numero di mosse per ogni istanza:

- Media: 4;
- Deviazione Standard: 0;
- Min: 4;
- Max: 4.

Quindi in definitiva sono sempre 4 le mosse applicate alla soluzione iniziale ("Initial Solution").

2.2.3 OF_Diff

Per quanto riguarda il campo "OF_Diff" del dataset, sono state calcolate anche in questo caso alcune statistiche. In particolare è stata calcolata la frequenza di volte in cui la differenza tra la funzione obiettivo finale e iniziale è positiva (maggiore di 0), neutra (uguale a 0) e negativa (minore di 0).

OF_Diff positiva	4947
OF_Diff neutra	15968
OF_Diff negativa	242

La frequenza di volte in cui si assiste a una *OF_Diff* uguale a 0 è quella prevalente, con 15968 istanze. Questo è dovuto al fatto che, applicando algoritmi *destroy-repair*, spesso l'eliminazione e ricostruzione di nodi nelle euristiche non influenzano i nodi di interesse, perché avvengono in maniera *Greedy*. Di fatto, un metodo euristico come quello *Greedy*, non condurrà all'ottimo del problema, ma ad una soluzione "ottima" secondo l'euristica che si è scelti di seguire [3].
In seguito, sono stati separati gli elementi con "OF_Diff" positiva e negativa. Di questi ultimi sono state calcolate le seguenti statistiche:

- Media positivi: 749.316
- Media negativi: -41.176
- Deviazione standard: positivi: 1150.919
- Deviazione standard negativi: 158.306
- Max positivo: 9174.788
- Min positivo: 3.637
- Max negativo: -0.00017
- Min negativo: -1361.0447

In seguito sono stati raggruppati gli elementi del campo "OF_Diff" in 5 classi:

much_improved	valore superiore alla media dei posivi	1963
little_improved	valore inferiore alla media dei posivi e maggiore di 0	2984
neutral	valore 0	15968
little_worse	valore minore di 0 e superiore alla media dei negativi	20
much_worse	valore inferiore alla media dei negativi	222

La suddivisione in 5 classi del campo "OF_Diff" è stata eseguita con il seguente criterio.

- La classe *much_improved* contiene gli elementi positivi che hanno un valore di *OF_Diff* superiore alla media dei positivi (749.316);
- La classe *little_improved* contiene gli elementi positivi ma con un valore inferiore alla media dei positivi (749.316)
- La classe *neutral* contiene gli elementi con valore pari a 0
- La classe *little_worse* contiene gli elementi con un valore minore di 0 ma superiore alla media degli elementi negativi (-41.176)
- La classe *much_worse* contiene gli elementi con un valore inferiore alla media degli elementi negativi (-41.176)

Tale suddivisione in 5 classi sarà necessaria alla classificazione multi-classe che si vedrà nel dettaglio nei prossimi capitoli.

Capitolo 3

Architetture

In questo capitolo esploreremo le diverse architetture delle reti neurali. Dalle reti *Recurrent* alle reti neurali a grafo vedremo come ognuna di queste architetture offre soluzioni per problemi specifici. Inizieremo esplorando le reti *Recurrent*, che sono molto utilizzate per problemi di elaborazione del linguaggio naturale e generazione di testo. Infine, esploreremo alcune architetture più avanzate, come le *Graph Neural Network*. In ogni caso, vedremo come ogni architettura è progettata per risolvere problemi specifici e come sfruttare al meglio le loro caratteristiche.
Affinché si raggiunga l'obiettivo di addestrare un classificatore che riesca a definire se una mossa, applicata a una soluzione iniziale, migliori o meno la funzione obiettivo, sono state analizzate dal punto di vista teorico le seguenti architetture di reti neurali:

1. *Long Short Term Memory* (**LSTM**);
2. *Transformer*;
3. *Graph Neural Network* (**GNN**).

3.1 Long Short Term Memory

Le reti neurali *Long Short Term Memory* (LSTM) sono un tipo di rete neurale di tipo *Recurrent*, progettate per gestire problemi di elaborazione del linguaggio naturale e generazione di testo. A differenza delle *Recurrent Neural Network* (RNN), le LSTM sono in grado di mantenere una memoria a lungo termine e di gestire informazioni con dipendenze a lungo termine.
Una LSTM è composta da una serie di "celle" che contengono tre tipi di porte: *forget*, ingresso e uscita. Le porte forget controllano quali informazioni devono essere cancellate dalla memoria a lungo termine, le porte di ingresso controllano quali informazioni devono essere aggiunte alla memoria a lungo termine, e le porte di uscita determinano quali informazioni devono essere passate al livello successivo della rete.
La rete LSTM quindi incorpora una "memoria interna" che permette di mantenere una memoria a lungo termine delle informazioni. Ciò significa che può "ricordare"

delle informazioni anche dopo che trascorre un lungo periodo, consentendo di gestire dipendenze a lungo termine tra le informazioni.
La rete LSTM è stata utilizzata con successo in una varietà di problemi di elaborazione del linguaggio naturale, tra cui la generazione di testo, il riconoscimento del linguaggio e la traduzione automatica.
Per l'obiettivo di questo lavoro di tesi, si è pensato di poter utilizzare questa rete per un task di classificazione binaria, ovvero verificare se una stringa contenente una sequenza di mosse porta a un miglioramento o meno della funzione obiettivo.
Prima di vedere nel dettaglio il funzionamento di una rete *LSTM*, verrà illustrata brevemente l'architettura su cui essa è basata, ovvero la *Recurrent Neural Network*.
La **RNN** è una generalizzazione della rete neurale feedforward dotata di una memoria interna. La RNN è "ricorrente" in quanto esegue la stessa funzione per ogni ingresso di dati, mentre l'uscita relativa all'ingresso attuale dipende dal calcolo precedente. Dopo aver prodotto l'output, questo viene copiato e inviato nuovamente alla rete ricorrente. Per prendere una decisione, considera l'input attuale e l'output che ha appreso dall'input precedente.
A differenza delle reti neurali feedforward, le RNN possono utilizzare il loro stato interno (memoria) per elaborare sequenze di input. Questo le rende applicabili a compiti come il riconoscimento di testo complesso. Nelle altre reti neurali, tutti gli ingressi sono indipendenti l'uno dall'altro. Nelle RNN, invece, tutti gli input sono correlati tra loro.

3.1.1 Struttura di una rete LSTM

Le reti LSTM (Long Short-Term Memory) sono una versione modificata delle reti neurali ricorrenti, che facilita la memorizzazione dei dati passati in input. Le reti LSTM sono adatte a classificare, elaborare e prevedere serie temporali. L'addestramento del modello avviene mediante la *back-propagation*. In una rete LSTM sono presenti tre porte:

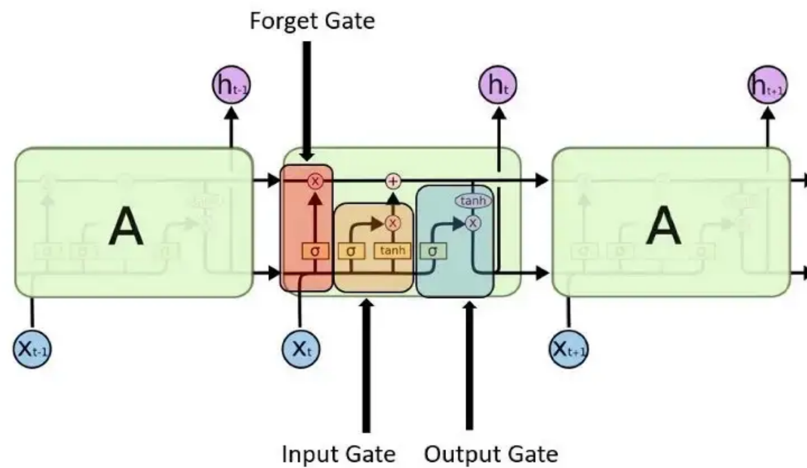


Figura 3.1: Rappresentazione dell'architettura di una LSTM. Fonte: [4]

1. **Input Gate**: Definisce quale valore dall'input dovrebbe essere utilizzato per modificare la memoria. La funzione *Sigmoid* decide quali valori far passare da 0 a 1. La funzione *Tanh* assegna un peso ai valori che gli vengono passati, decidendo il loro livello di importanza compreso tra -1 e 1.

2. **Forget gate**: Definisce quali dettagli scartare dal blocco. Viene deciso dalla funzione *Sigmoid*. Essa esamina lo stato precedente (h_{t-1}) e l'input (X_t) e genera un numero compreso tra 0 (se deve omettere l'informazione) e 1 (se deve conservare l'informazione) per ogni valore nello stato della cella C_{t-1}.

3. **Output gate**: L'input e la memoria del blocco vengono utilizzati per decidere l'output. La funzione *Sigmoid* decide quali valori lasciar passare da 0 a 1. La funzione *Tanh* assegna un peso ai valori che vengono passati, decidendo il loro livello di importanza da -1 a 1 e moltiplicandolo con l'output della *Sigmoid*.

3.1.2 Problemi delle LSTM

Lo stesso problema che generalmente accade alle RNN, accade anche con le LSTM, cioè quando le frasi sono troppo lunghe, le LSTM non sono molto efficaci. La ragione è che la probabilità di mantenere il contesto di una parola lontana dalla parola attualmente in elaborazione diminuisce esponenzialmente con la distanza da essa. Ciò significa che quando le frasi sono lunghe, il modello spesso dimentica il contenuto delle posizioni lontane nella sequenza. Un altro problema con le RNN e le LSTM è che è difficile parallellizzare il lavoro di elaborazione delle frasi, poiché è necessario elaborare le parole una per una. Per riassumere, LSTM e RNN presentano 2 problemi principali [5]:

- Limitata parallelizzazione dovuta alla computazione sequenziale;
- Non sono modellate in maniera concreta le dipendenze a lungo e breve termine;

3.2 Transformer

Le reti neurali Transformer sono un tipo di architettura di rete neurale sviluppata nel 2017 e progettata per problemi di elaborazione del linguaggio naturale, come la traduzione automatica, il riconoscimento del linguaggio e la generazione di testo. A differenza delle reti neurali *Recurrent*, come le LSTM, le reti Transformer utilizzano un meccanismo chiamato *attention* per elaborare il testo.
Il meccanismo dell'*attention* consente alle reti Transformer di concentrarsi su particolari parole o frasi nel testo mentre elaborano l'informazione. Ciò significa che possono prendere in considerazione il contesto globale del testo, anziché essere limitate dalla sequenza temporale delle parole.
L'architettura Transformer consiste in due blocchi principali: una serie di layer *multi-head* di *attention* e una serie di layer *feed-forward neural network* (FNN). I layer *multi-head* di attention utilizzano più *head* di attention per catturare le relazioni tra parole diverse nel testo, mentre i layer FNN sono utilizzati per elaborare le informazioni raccolte dagli strati di *attention*.
Anche questa architettura potrebbe prestarsi bene al task di classificazione delle "Moves". Infatti, potrebbe classificare se una stringa di testo rappresentante una mossa di *destroy-repair*, applicata a una soluzione iniziale, porta o meno a un miglioramento della funzione obiettivo.

3.2.1 Struttura di una rete Transformer

La struttura di una rete Transformer è mostrata nella seguente figura:

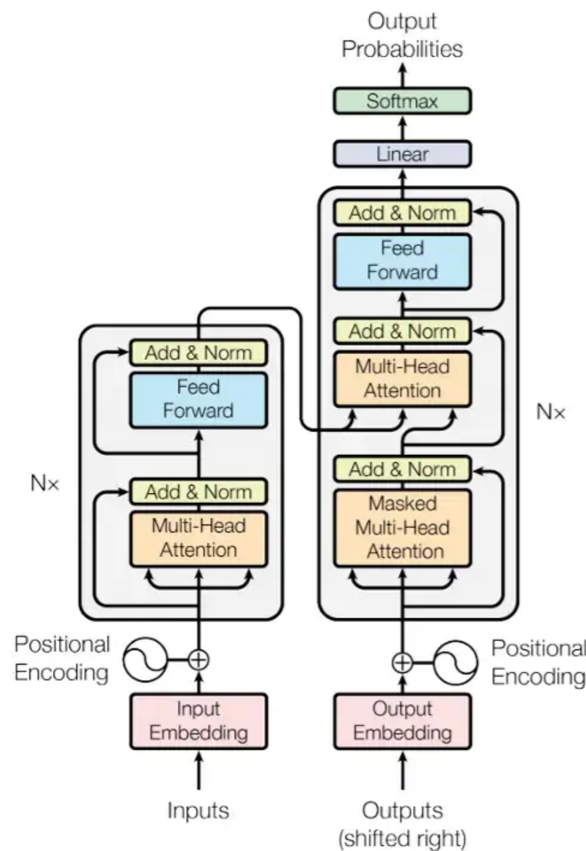


Figura 3.2: Architettura di una rete Transformer. Fonte: [6]

Analizziamo ora i singoli blocchi che compongono l'architettura.

Encoder Block L'encoder block è visibile in figura 3.2, nel blocco di sinistra nell'immagine.
Il comportamento delle singole componenti è il seguente:

- **Input embedding**: le stringhe con significati simili sono raggruppate in uno spazio chiamato **Embedding Space**. In seguito vengono convertite le parole in vettori;
- **Positional Encoding**: è un vettore che fornisce il contesto di una stringa in base alla sua posizione nella sequenza di stringhe;
- **Multi-Head Attention**: riporta quanto sia rilevante una determinata parola rispetto alle altre parole nel contesto. Viene rappresentato come un vettore di *attention*. Per ogni parola si può avere un vettore di attention che rappresenta il contesto della parola nella sequenza;

- **Feed Forward**: rete neurale feed-forward che viene applicata a ogni vettore di attenzione. Il suo scopo principale è quello di trasformare i vettori di attention in un formato compatibile con il prossimo strato di encoder o decoder.

Decoder Block Il decoder block è visibile in figura 3.2, in particolare è rappresentato dal blocco di destra nell'immagine. Il comportamento delle singole componenti è riassunto come segue:

- **Output Embedding e Positional Encoding**: trasformano le parole nei corrispettivi vettori, ha infatti un funzionamento simile all'Encoder Block
- **Masked Multi-Head Attention**: il funzionamento è simile al Multi-Head Attention, con la differenza che alcune parole vengono "mascherate" ponendole come *0* in modo che la rete non può utilizzarle
- **Multi-Head Attention**: i risultati dei vettori di *attention* provenienti dagli strati precedenti e i vettori provenienti dall'Encoder Block sono passati in questo blocco
- **Feed Forward**: viene passato ogni vettori di *attention* in tale blocco, esso farà in modo che i vettori di output siano in un formato compatibile con un altro Decoder Block o con un *Linear Layer*
- **Linear Layer**: è un altro strato feed-forward che viene utilizzato per espandere il numero di parole
- **Softmax Layer**: trasforma l'input in una distribuzione di probabilità, che è più facile da interpretare. L'output sarà quello con la probabilità più alta

In conclusione la rete Transformer inizia generando un embedding per ogni parola. In seguito, utilizzando l'attention, aggrega le informazioni ottenute da tutte le parole, generando una nuova rappresentazione per ogni parola, arricchita dall'intero contesto. Questo passaggio viene quindi ripetuto più volte in parallelo per tutte le parole, generando successivamente nuove rappresentazioni.
Il decoder opera in modo simile, ma genera una parola alla volta. Si riferisce non solo alle altre parole precedentemente generate, ma anche alle rappresentazioni finali generate dal blocco di encoder.

3.3 Graph Neural Network

Le Graph Neural Networks (GNNs) sono una classe di reti neurali che consentono di elaborare e analizzare dati strutturati in forma di grafi. Questi modelli si sono dimostrati particolarmente utili per la risoluzione di problemi di elaborazione dei dati che implicano relazioni complesse tra gli elementi. In questa sezione, esploreremo la teoria base dei grafi, le tipologie e le rappresentazioni, i principi fondamentali delle GNN, nonché la struttura dell'architettura di questa rete neurale e alcune applicazioni tipiche. Dopo una caratterizzazione dei grafi, inizieremo esaminando la struttura di base delle GNN e le tecniche utilizzate per la loro costruzione, quindi esploreremo alcuni esempi di come questi modelli possono essere utilizzati per la classificazione dei nodi, la predizione di archi e la classificazione globale di un grafo. In sintesi, questo capitolo fornirà una panoramica completa sui grafi e sulle GNN.

3.3.1 Caratterizzazione di un grafo

Un **grafo** rappresenta le relazioni tra un insieme di entità. Una relazione è chiamata *arco* e un'entità è chiamata *nodo*.

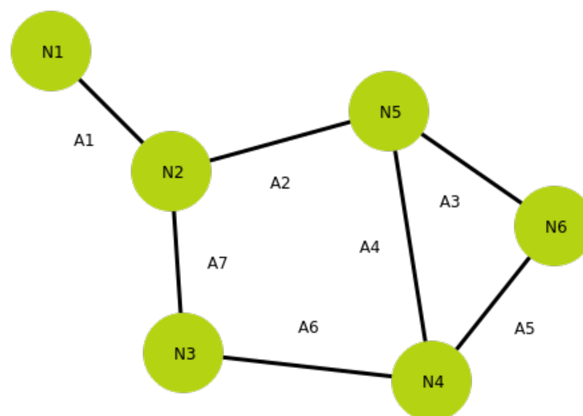


Figura 3.3: Immagine rappresentante un grafo semplice con 6 nodi e 7 archi.
Fonte: [7]

I grafi hanno molteplici applicazioni. In particolare, possono essere utilizzati per rappresentare:

1. Immagini
2. Testo
3. Molecole
4. Social Network

Esistono tre tipi generali di task di predizione sui grafi: a livello di grafo (**Graph-level task**), a livello di nodo (**Node-level task**) e a livello di arco (**Edge-level task**).
In un task a livello di grafo, vogliamo predire una singola proprietà per un intero grafo. In un task a livello di nodo, vogliamo predire una proprietà per ogni nodo del grafo. Per un task a livello di archi, si vogliono predire le proprietà o la presenza di archi in un grafo.
Per il lavoro di tesi è stato posto il focus sulla predizione a livello di grafo, perché in input alla nostra rete vogliamo passare l'intero grafo e giudicare se una mossa applicata al grafo migliora o peggiora la funzione obiettivo. La proprietà del grafo che vogliamo estrarre è quindi la bontà o meno di una mossa applicata ad esso.

Rappresentazione di un grafo

Il primo passo è rappresentare i grafi per renderli compatibili con le reti neurali.
I modelli di apprendimento automatico prendono in genere come input matrici rettangolari o a griglia. Non è quindi immediatamente intuitivo come rappresentarli in un formato compatibile. I grafi hanno fino a quattro tipi di informazioni che potenzialmente si possono usare per fare previsioni, questi sono: nodi, archi, contesto globale e connettività. Le prime tre sono relativamente semplici: per esempio, con i nodi possiamo formare una matrice di caratteristiche dei nodi N assegnando a ogni nodo un indice **i** e memorizzando la caratteristica del $\mathbf{nodo_i}$ in **N**.
Tuttavia, rappresentare la connettività di un grafo è più complicato. Forse la scelta più ovvia sarebbe quella di utilizzare una matrice di adiacenza, in quanto si presta meglio ad essere trasformata in un tensore. Questa rappresentazione presenta alcuni svantaggi. Il numero di nodi in un grafo può essere dell'ordine di milioni e il numero di archi per nodo può essere molto variabile. Spesso questo porta a matrici di adiacenza molto sparse, che sono poco efficienti dal punto di vista dell'occupazione di memoria. Un altro problema è che ci sono molte matrici di adiacenza che possono codificare la stessa connettività e non c'è garanzia che queste diverse matrici producano lo stesso risultato in una rete neurale.

Rappresentazione mediante liste di adiacenza Un modo efficiente in termini di memoria per rappresentare le matrici sparse è quello delle liste di adiacenza. Queste descrivono la connettività dell'arco (edge) e_k tra i nodi n_i e n_j come una tupla (i , j) nella k-esima voce di una lista di adiacenza. Dato che il numero di archi è spesso molto inferiore rispetto al numero di campi di una matrice di adiacenza ($n^2{}_{\text{nodes}}$), in questo modo si evita di calcolare e memorizzare le parti disconnesse del grafo. Avremo quindi n nodi: $[0..n_{-1}]$ e k archi, dove k è anche la dimensione delle liste di adiacenza. Quindi salviamo nelle liste di adiacenza i soli nodi connessi tra loro tramite l'arco e_k. Di seguito è riportata un'immagine che rappresenta il concetto appena espresso:

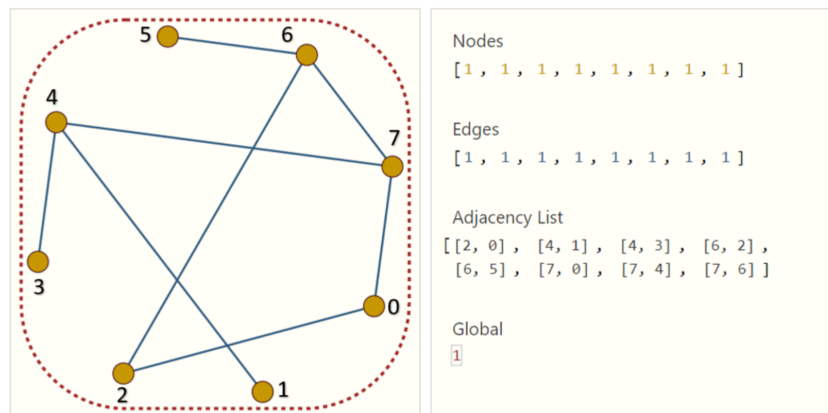


Figura 3.4: Rappresentazione di un grado con 8 nodi e 8 archi. La matrice di adiacenza sarà una lista contenente 8 liste, una per ogni arco del grafo.
Fonte: [8]

Nella figura vengono utilizzati valori scalari per nodo/arco, ma la maggior parte delle rappresentazioni tensoriali usano dei vettori per ogni attributo del grafo. Invece di un tensore di nodi di dimensione $[n_{nodes}]$, avremo a che fare con tensori di nodi di dimensione $[n_{nodes}, nodes_{dim}]$. Il discorso è analogo per gli archi del grafo.

3.3.2 Struttura di una Graph Neural Network

Ora che la descrizione del grafo è in un formato matriciale invariante alle permutazioni, descriveremo l'uso delle reti neurali a grafo (GNN) per risolvere task di predizione del grafo. Una GNN è ottimizzabile su tutti gli attributi del grafo (nodi, archi, contesto globale) che preserva le simmetrie del grafo (invarianza di permutazione). Una GNN è costruita utilizzando il framework "message passing neural network" proposto da [9] e rappresentando i grafi con vettori invece che con scalari. Di seguito viene illustrata una GNN semplice, quella in cui vengono appresi nuovi *embeddings* per tutti gli attributi del grafo (nodi, archi, globale), ma non vengono considerate le connettività del grafo. Questa GNN utilizza un multilayer perceptron (MLP) separato su ogni componente di un grafo. Questo è chiamato strato GNN. Per ogni vettore di un nodo viene applicato il MLP affinché si ottenga l'apprendimento di un *vettore nodo*. Lo stesso procedimento avviene per ogni arco, creando un embedding per ogni arco, e per il vettore del contesto globale, definendo un singolo embedding per l'intero grafo:

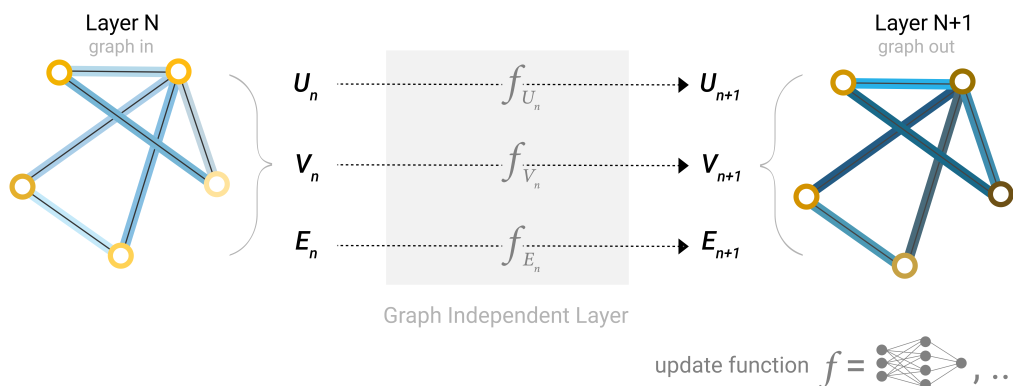


Figura 3.5: Un singolo strato di una GNN. L'input è un grafo e ogni componente (V,E,U) viene aggiornato da una MLP per produrre un nuovo grafo. Ogni pedice della funzione indica una funzione separata per un diverso attributo del grafo all'n-esimo strato di un modello GNN.
Fonte:[8]

Come accade spesso con i moduli o gli strati delle reti neurali, si uniscono gli n strati GNN. Poiché una GNN non aggiorna la connettività del grafo di ingresso, il grafo di uscita di una GNN viene descritto con lo stesso elenco di adiacenze e lo stesso numero di vettori di *feature* del grafo di ingresso. Il grafo di uscita quindi presenta embeddings aggiornati, poiché è stata aggiornata ciascuna delle rappresentazioni dei nodi, degli archi e del contesto globale.

Predizione con GNN

Si considera il caso di una classificazione binaria, ma tale struttura può essere facilmente estesa al caso multi-classe o alla regressione. Nel nostro caso di studio abbiamo informazioni a livello di nodo e di arco e dobbiamo predire una proprietà globale binaria, dobbiamo quindi raccogliere tutte le informazioni disponibili sui nodi e aggregarle. Questo comportamento è simile allo strato di *Global Average Pooling* utilizzato nelle *Convolutional neural network* (CNN).

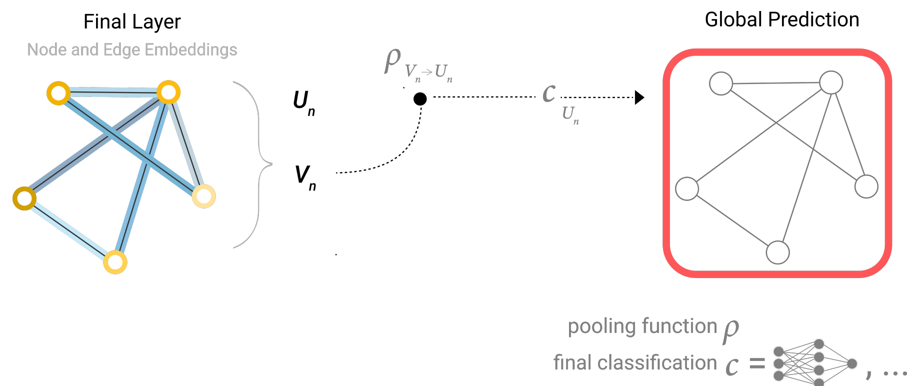


Figura 3.6: Predizione globale

Il *Pooling* procede in due fasi:

- Per ogni elemento da mettere insieme, si raccolgono tutti gli embeddings e li si concatena in una matrice;
- Gli embeddings raccolti vengono poi aggregati tramite un'operazione di somma.

Il funzionamento di una Graph Neural Network è riassunto come segue.

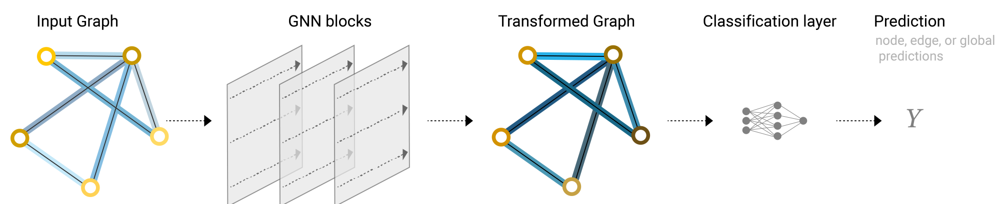


Figura 3.7: Recap della struttura di una Graph Neural Network

É possibile quindi fare previsioni binarie passando le informazioni tra le diverse parti del grafo. Se si hanno nuovi attributi del grafo, si può semplicemente definire come passare le informazioni da un attributo all'altro. Non si utilizza la connettività del grafo all'interno dello strato GNN. Infatti, ogni nodo viene elaborato in modo indipendente, così come ogni arco e contesto globale. Viene utilizzata la connettività solo quando si mettono insieme tutte le informazioni per ottenere una predizione.

Capitolo 4

Implementazione

In questo capitolo, esploreremo l'implementazione in Python di tre diverse tipologie di reti neurali utilizzate per il task di classificazione: la Long Short-Term Memory (LSTM), la rete Transformer e la Graph Neural Network (GNN).
Prima di passare alle specifiche implementazioni, vengono riportate nelle seguenti due tabelle le versioni dei tool e delle librerie necessarie per eseguire le reti.

Python	3.7.1
pip	22.3.1
tensorflow	2.7.0
pandas	1.3.5
numpy	1.21.6

Tabella 4.1: Versione dei software utilizzati per le reti **LSTM** e **Transformer**

Python	3.8.10
pip	22.0.4
pytorch	1.13.1
CUDA	11.2
networkx	2.6.3
torchmetrics	0.11.0

Tabella 4.2: Versione dei software utilizzati per la **Graph Neural Network**

Le seguenti sezioni riportano, per ogni architettura utilizzata, i pezzi salienti di codice Python con la relativa spiegazione. Il codice completo è invece disponibile al seguente link della repository *GitHub*: workspaces. La repository riporta tutti i codici relativi alle varie fasi del lavoro:

1. EDA-Exploratory data analysis

2. LSTM-Long Short-Term Memory

3. Transformer

4. GNN-Graph Neural Network

Il codice relativo ai punti 1-3 è eseguibile sul PC in locale scaricando il file *Zip* dalla repository *GitHub* e installando le librerie viste nella tabella 4.1. Per quanto riguarda invece l'esecuzione del codice relativo alle GNN è necessario utilizzare *Google Colab*. Per eseguire il notebook Python è sufficiente aprire ed eseguire i file con estensione **ipynb** presenti nella cartella disponibile al link: GNN-Drive.

4.1 LSTM

La prima implementazione riguarda la rete LSTM (Long Short-Term Memory). Vengono di seguito riportati i punti salienti dell'implementazione.

```
1 moves_string=[]
2 label=[]
3 moves=[]
4 with open("input\input_1_balanced.csv", 'r') as csvfile:
5     reader = csv.reader(csvfile, delimiter=',')
6     next(reader)
7     for row in reader:
8         moves_string.append(row[0])
9         label.append(row[1])
```

In questa prima fase si va a leggere il file di input che riguarda la combinazione di mosse applicata alla soluzione iniziale (campo "Moves" del dataset) e la *label* assegnata a ogni istanza del dataset. La *label* è un numero intero che ha valore 0 oppure 1. Ha valore 0 se la combinazione di mosse non migliora la funzione obiettivo, 1 altrimenti. In particolare nel codice si vanno a popolare due liste, una per le mosse e l'altra per le labels. Una volta ottenute le liste, ogni elemento andrà "tokenizzato". Di seguito è riportata la porzione di codice che permette di ottenere dei token (numeri *integer*) per ogni stringa di testo relativa alla combinazione di mosse. La trasformazione in token è fondamentale perché la rete prende in input dei numeri e non del testo.

```
1 tokenizer = Tokenizer(num_words = vocab_size, oov_token=
    oov_tok)
2 tokenizer.fit_on_texts(train_moves)
3 word_index = tokenizer.word_index
4 dict(list(word_index.items())[0:10])
5 train_sequences = tokenizer.texts_to_sequences(train_moves)
```

Il *Tokenizer* prende in input due argomenti: **num_words** e **oov_token**, che rispettivamente rappresentano il numero massimo di parole uniche (settato a 17 come

il numero di mosse applicabili) e la sostituzione delle parole fuori vocabolario durante le chiamate **text_to_sequence**. Dopo aver trasformato le stringhe in token, vengono trasformate in sequenze di token, una per ogni mossa. Il procedimento è analogo per il dataset di *testing*.
La rete vera e propria è implementata come segue:

```
def create_model(input_length):
    print ('Creating model...')
    model = Sequential()
    model.add(Embedding(input_dim = 50, output_dim = 50,
   input_length = input_length))
    model.add(LSTM(units=2, activation='relu' ,
   return_sequences=True))
    model.add(Dropout(0.5))
    model.add(LSTM( units=2,activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='relu'))

    print ('Compiling...')
    model.compile(loss='binary_crossentropy',
                  optimizer='rmsprop',
                  metrics=[tf.keras.metrics.BinaryAccuracy(),
   tf.keras.metrics.Precision(),tf.keras.metrics.Recall(),tf.
   keras.metrics.FalseNegatives(),tf.keras.metrics.
   TrueNegatives(),tf.keras.metrics.TruePositives(),tf.keras.
   metrics.FalsePositives()])

    return model
```

La rete implementa i vari strati: Embedding, LSTM, Dropout, Dense. Tutti gli strati hanno degli iperparametri da scegliere. Un iperparametro di una rete neurale è un parametro che non viene appreso durante il processo di addestramento, ma viene impostato dall'utente prima di iniziare il processo di addestramento. Questi possono includere il numero di strati nella rete, il numero di unità in ogni strato, il tasso di apprendimento e il metodo di regolarizzazione. La scelta appropriata degli iperparametri può influire sull'efficacia della rete neurale nell'apprendimento delle funzioni desiderate. Questo argomento verrà approfondito nella sezione 5.1.2.

4.2 Transformer

La seconda implementazione riguarda la rete Transformer. Anche qui vengono riportati i punti salienti dell'implementazione. Per quanto riguarda la generazione di *token* a partire dalle stringhe di mosse, l'implementazione è identica a quella della rete LSTM. Ciò che cambia è cosa prende in input il Tokenizer. Infatti in questo caso prende in input sia la soluzione iniziale che la combinazione di mosse. La stringa "tokenizzata" sarà più lunga rispetto al token delle LSTM. Per tale motivo il

parametro "num_words" è settato a 138, perché il dataset presenta: 100 clienti, 20 stazioni di ricarica, un deposito e 17 mosse applicabili. Di seguito è riportato il codice implementativo dell'architettura.

```
1 inputs = layers.Input(shape=(maxlen,))
2 embedding_layer = TokenAndPositionEmbedding(maxlen, vocab_size
    , embed_dim)
3 x = embedding_layer(inputs)
4 transformer_block = TransformerBlock(embed_dim, num_heads,
   ff_dim)
5 x = transformer_block(x)
6 x = layers.GlobalAveragePooling1D()(x)
7 x = layers.Dropout(0.1)(x)
8 x = layers.Dense(20, activation="relu")(x)
9 x = layers.Dropout(0.1)(x)
10 outputs = layers.Dense(2, activation="softmax")(x)
11 model = keras.Model(inputs=inputs, outputs=outputs)
12 model.compile(
13     optimizer="Adam", loss="sparse_categorical_crossentropy",
   metrics=["accuracy",precision_m,recall_m,f1_m])
14 history = model.fit(
15     x_train, y_train, batch_size=32, epochs=10,
   validation_data=(x_val, y_val))
```

La rete implementa i vari strati: TransformerBlock, GlobalAveragePooling1D, Dropout, Dense. Anche qui sono settati diversi iperparametri la cui scelta sarà discussa nella sezione 5.1.2.

4.3 GNN

La terza e ultima implementazione riguarda la rete GNN-Graph Neural Network. Vengono di seguito riportate le funzioni e le classi Python più rilevanti del codice con la relativa spiegazione di ciò che esse svolgono.

- **create_graph()**: prende in input il file relativo alla soluzione iniziale, aggiunge i nodi e gli archi al grafo. Ogni grafo è poi aggiunto alla lista dei grafi. Inoltre crea un file relativo agli attributi dei grafi. Un attributo è sostanzialmente il nome di ogni singolo nodo, per esempio **C21**, cioè il Cliente numero 21. Infine la funzione ritorna la lista dei grafi.

- **dictattr(PATH,file)**: prende in input il file degli attributi creato in **create_graph()** e crea un dizionario le cui chiavi sono il nome dei nodi e il valore è una lista di 1 o 0 (dove il valore è 1 se il nodo è presente, 0 altrimenti).

- **class TraceDataset(InMemoryDataset)**: si fa override della classe Python andando a definire un tensore per ogni grafo presente nella lista. Il tensore avrà 3 dimensioni: **x** (nodi del grafo), **edge_index** (gli archi del grafo) e **y** che è

la label. Infine ritorna la lista di tutti i tensori 3-dimensionali. Ogni tensore è una riga del dataset di partenza.

- **class SortPool(torch.nn.Module)**: definisce tutti i layer (strati) della rete. Comprende diverse funzioni come: __init__(), reset_parameters(), forward().
- **Split(G, per)**: prende in input il grafo G e la percentuale di training. Esegue quindi uno split del grafo G tra training e testing. Prepara quindi il dataset per la fase di addestramento e testing.
- **train()**: funzione di addestramento della rete.
- **test(loader)**: funzione per il test della rete.

Definite le funzioni e le classi, l'addestramento e il test della rete sono avviati dalla seguente porzione di codice:

```
1    for epoch in range(start_epoch, start_epoch + delta_ep):
2      train_loss, train_acc, cmt_train,f1_train = train()
3      test_loss, test_acc, cmt_test,f1_test =test(test_loader)
4      lossTr.append(train_loss / len(
5          train_loader.dataset))
6      lossTe.append(test_loss / len(test_loader.dataset))
7      accTr.append(train_acc / len(train_loader.dataset))
8      accTe.append(test_acc / len(test_loader.dataset))
9
10     print(f'Epoch: {epoch:03d}, Train Acc: {(train_acc / len
   (train_loader.dataset)):.4f},Train F1: {f1_train}, Test Acc
   : {(test_acc / len(test_loader.dataset)):.4f},Test F1: {
   f1_test} , Train Loss: {(train_loss / len(train_loader.
   dataset)):.4f}, Test Loss: {(test_loss / len(test_loader.
   dataset)):.4f} ',file=f)
```

Per quanto riguarda l'implementazione della rete GNN, è stato opportunamente modificato e trasformato il dataset in un formato con estensione **.g**. La procedura di trasformazione del dataset e la scelta degli iperparametri è illustrata nel prossimo capitolo, nella sezione 5.1.

Capitolo 5

Risultati

In questo capitolo verranno presentati i risultati ottenuti dall'addestramento delle tre seguenti reti neurali: LSTM, Transformer e Graph Neural Network. Verrà descritto il metodo utilizzato per l'addestramento e verranno mostrati i risultati ottenuti in termini di precisione e performance. Inoltre, verrà discusso il contesto in cui queste reti neurali sono state utilizzate e le loro potenziali applicazioni future. L'obiettivo di questo capitolo è quello di fornire una panoramica completa sull'efficacia di questi modelli nell'elaborazione dei dati e nella risoluzione di problemi specifici.

Prima di mostrare i risultati degli esperimenti, verranno fornite le seguenti informazioni nel *Setup Sperimentale*:

- Descrizione del dataset utilizzato;
- Combinazioni di parametri/hyperparameters che sono stati testati;
- Metriche di valutazione utilizzate;
- Tecniche di validazione utilizzate.

5.1 Setup sperimentale

In questa sezione si vanno a descrivere i dataset utilizzati nelle varie reti, le combinazioni di iperparametri che sono stati testati, le metriche di valutazione utilizzate e infine le tecniche di validazione utilizzate.

5.1.1 Dataset utilizzati

Per quanto riguarda i dataset da dare in input alla rete neurale bisogna fare una distinzione tra le varie reti utilizzate.

LSTM e Transformer Le reti LSTM e Transformer prendono in input un unico file **csv**. Sono stati eseguiti due test con due input diversi. Uno è il file csv completo, DB-Output.csv, l'altro è invece bilanciato ed è generato dallo script presente nella

repository disponibile al link: workspaces. Lo script è raggiungibile al seguente percorso: "2-LSTM\input\input_1_balanced.py". Tale codice genera un file in formato csv che comprende due campi, uno relativo alla combinazione di mosse da applicare alla soluzione iniziale e uno relativo alla label. La label è binaria, con valore 0 se la combinazione di mosse non migliora la funzione obiettivo o con valore 1 se la migliora. Inoltre bilancia il dataset con 50% di valori "0" e 50% di valori "1". Abbiamo quindi ottenuto un dataset, a partire da quello completo di partenza, che ha 4947 istanze per ognuna delle due classi, con un totale di 9894 elementi in input. Ogni istanza del dataset bilanciato avrà quindi la combinazione di 4 mosse applicate all'istanza iniziale e la label. Di seguito è riportato un esempio.

```
1 moves,label
2 "['null', 'null', 'WorstDistanceDestroyCustomer', '
    GreedyRepairCustomer']",0
3 "['null', 'null', 'RandomRouteDestroyCustomer', '
    GreedyRepairCustomer']",1
```

GNN La Graph Neural Network avrà due input, uno relativo al campo "Initial Solution" e l'altro relativo alla label. Per questa rete è stata utilizzata una strategia diversa rispetto alle due precedenti. Di fatto qui non abbiamo riferimenti alla combinazione di 4 mosse perché si è eseguito uno split del dataset completo in 17 file. Ogni file è relativo a una sola mossa. É stato quindi eseguito un raccoglimento di tutte le istanze del dataset completo che abbiano una specifica mossa nel campo "Moves". Avremo quindi 17 file relativi ai nodi del grafo (campo "Initial Solution") e 17 file contenente la label associata alla soluzione iniziale. Ogni file è relativo a una singola mossa. Vediamo ora nello specifico come sono i due file in input. In particolare, il primo file contiene i nodi e gli archi del grafo di partenza, il secondo contiene le label. Di entrambi abbiamo 17 file, uno per ogni mossa.
Il primo input non è in formato csv come per le altre reti, bensì è stato trasformato in un formato con estensione **.g**. Questo file contiene tutti i nodi e gli archi del grafo con una formattazione particolare. Infatti ogni riga del file è popolato da due tipi di elementi: **v** ed **e**, vertici e archi. I vertici hanno associato il nome del nodo, gli archi invece la coppia di nodi che esso lega. Lo script che genera i 17 file in formato **.g** è presente nel file: "5-GNN\ETL\2_generate_punto_g.py" all'interno della repository (workspaces). Per quanto riguarda il secondo input della rete, relativo alla label, per la GNN si è pensato di usare delle label multi-classe, e non binaria come per LSTM e Transformer. In questo caso le label sono 5 e sono: "migliora molto", "migliora poco", "neutro", "peggiora poco" e "peggiora molto", numerate da 4 a 0. Il file che esegue questa suddivisione è il seguente: "5-GNN\ETL\4_generate_5_labels.py". Abbiamo quindi definito i due file che la rete GNN prende in input. Per ogni coppia di input (soluzione iniziale e label) abbiamo 17 file, uno per ogni mossa. L'obiettivo è quello di addestrare 17 classificatori che possano stabilire quanto la mossa,

applicata a una soluzione iniziale, migliora o peggiore la funzione obiettivo.

5.1.2 Iperparametri

In questa sezione vengono illustrati gli iperparametri scelti per l'addestramento delle tre reti neurali. Come descritto in [10], per iperparametri si intende un insieme di parametri che non possono essere aggiornati durante l'addestramento di apprendimento automatico di una rete neurale, ma sono scelti dall'utente. Possono essere coinvolti nella costruzione della struttura del modello, come ad esempio il numero di strati nascosti e la funzione di attivazione, o nel determinare l'efficienza e l'accuratezza dell'addestramento del modello. Alcuni esempi sono il tasso di apprendimento (Learning Rate-LR) della discesa stocastica del gradiente (Stochastic Gradient Descent-SGD), la dimensione del batch (Batch Size) e l'ottimizzatore. Vediamo ora, per ogni rete addestrata, le combinazioni di iperparametri scelte per l'addestramento.

LSTM Partendo dalla rete LSTM, viene prima riportata una tabella che elenca gli iperparametri utilizzati per la fase di addestramento della rete, successivamente viene illustrato il significato di ognuno.

id	embedding_dim	activation LSTM	activation Dense	epochs	batch_size
1	18	relu	relu	10	32
2	**50**	relu	relu	**20**	**16**
3	18	relu	**tanh**	**20**	32
4	**50**	relu	relu	**20**	32
5	18	**tanh**	relu	10	32

Tabella 5.1: Iperparametri testati per l'addestramento della rete LSTM

Vediamo ora singolarmente cosa rappresentano gli iperparametri soggetti a test:

- **embedding_dim**: dimensione del vettore di embedding. Il vettore di embedding è il vettore ottenuto trasformando le stringhe relative alle mosse dell'algoritmo in vettori di numeri interi che sono poi dati in input alla rete;
- **activation LSTM**: funzione di attivazione da utilizzare per lo strato LSTM. Di default è la tangente iperbolica (tanh). Se si passa "None", non viene applicata alcuna attivazione;
- **activation Dense**: funzione di attivazione da utilizzare per lo strato denso;
- **epochs**: un'epoca significa addestrare la rete neurale con tutti i dati di addestramento per un ciclo. In un'epoca, tutti i dati vengono utilizzati esattamente una volta. Un forward e un backward vengono contati come un unico passaggio;

- **batch_size**: numero di elementi di addestramento utilizzati in un'iterazione. In genere, maggiore è la dimensione del batch, più velocemente il modello completerà ogni epoca durante l'addestramento.

Transformer Anche per la rete Transformer è stato effettuato un *Parameter Tuning*. Viene di seguito riportata una tabella che elenca gli iperparametri utilizzati per l'addestramento della rete. Successivamente viene illustrato il significato di ognuno.

id	embed_dim	num_heads	ff_dim	batch_size	epochs	activation
1	32	2	32	32	20	relu
2	32	2	32	32	20	**tanh**
3	**64**	2	32	32	20	relu
4	**64**	**4**	32	32	20	**tanh**
5	32	**4**	32	32	**10**	relu
6	32	2	**64**	32	**10**	relu
7	32	2	32	**16**	**10**	relu

Tabella 5.2: Nella tabella vengono riportati gli iperparametri testati per l'addestramento della rete Transformer

Vediamo ora singolarmente cosa rappresentano gli iperparametri sottoposti a test: [1]

- **num_heads**: numero di "Attention Heads" per ogni livello di attention del decoder della rete Transformer;

- **ff_dim**: dimensione dello strato intermedio di *Feed Forward* in ogni token.

[1]Gli iperparametri: embed_dim, batch_size, epochs e activation sono in comune con la rete LSTM, per cui il significato è visionabile nel paragrafo precedente (Iperparametri LSTM).

GNN Per quanto riguarda la Graph Neural Network gli iperparametri sottoposti a test sono i seguenti:

id	k	batch_size	dropout	lr	num_ly	delta_epoch
1	7	64	0.1	0.01	3	50
2	**15**	64	0.1	0.01	3	50
3	7	**32**	0.1	0.01	3	50
4	7	64	0.1	0.01	**5**	50
5	7	64	0.1	0.01	**2**	50
6	7	64	**0.5**	0.01	3	50
7	7	64	**0.05**	0.01	3	50
8	7	64	0.1	**0.1**	3	50
9	7	**128**	0.1	0.01	3	50
10	7	64	0.1	**0.005**	3	**100**

Tabella 5.3: Nella tabella vengono riportati gli iperparametri testati per l'addestramento della Graph Neural Network

Vediamo ora singolarmente cosa rappresentano gli iperparametri soggetti a test:[2]

- **k**: dimensione del kernel. Il kernel è un sottoinsieme dell'input che a ogni iterazione considera una porzione dell'input intero;
- **dropout**: quantità di unità (neuroni) da ignorare durante la fase di addestramento. Per "ignorare" si intende che queste unità non vengono considerate durante un particolare passaggio Forward o Backward;
- **lr**: regola la dimensione del passo con cui la rete aggiorna o apprende i valori della stima di un parametro
- **num_ly**: numero di strati del modello "SortPool";

[2]Gli iperparametri: batch_size e delta_epoch sono analoghi rispettivamente a "batch_size" e "epochs" visti per le LSTM, per cui non vengono ripetute le relative definizioni.

5.1.3 Metriche di valutazione

Le metriche di valutazione di una rete neurale sono utilizzate per misurare la performance del modello e verificare se esso è in grado di effettuare previsioni accurate. Tra le metriche più comuni utilizzate in questo contesto ci sono: Accuracy, Precision, Recall e F_1-score. Nello studio di tesi, i vari test sono valutati tramite Accuracy e F_1-score, quest'ultimo è una combinazione di precision e recall. Queste due metriche permettono di valutare se, le tre reti utilizzate, svolgono correttamente il task di classificazione. Prima di definire singolarmente le 4 metriche di valutazione, viene riportata la definizione di: *True Positive* (**TP**), *True Negative* (**TN**), *False Positive* (**FP**) e *False Negative* (**FN**).

1. *True Positive*: elemento Positivo e classificato come Positivo;
2. *True Negative*: elemento Negativo e classificato come Negativo;
3. *False Positive*: elemento Negativo ma classificato come Positivo;
4. *False Negative*: elemento Positivo ma classificato come Negativo.

Accuracy L'accuracy rappresenta il numero di istanze di dati classificate correttamente rispetto al numero totale di istanze di dati. La formula per il calcolo è la seguente:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

L'accuratezza può non risultare una buona metrica se il dataset non è bilanciato, cioè se entrambe le classi, negativa e positiva, hanno un numero diverso di istanze. Per tale motivo con ci si può basare solo sull'accuracy per valutare un classificatore.

Precision Rappresenta l'accuratezza con cui il classificatore prevede le classi positive. La formula è la seguente:

$$Precision = \frac{TP}{TP + FP}$$

La precision dovrebbe essere idealmente pari a 1 per un buon classificatore. La precisione diventa 1 solo quando il numeratore e il denominatore sono uguali, cioè $TP = TP + FP$, il che significa che FP è zero. Quando FP aumenta, il valore del denominatore diventa maggiore del numeratore, di conseguenza il valore di Precision diminuisce.

Recall Anche chiamata *sensitivity* o *true positive rate*: indica il rapporto di istanze positive correttamente individuate dal classificatore. La formula è la seguente:

$$Recall = \frac{TP}{TP + FN}$$

La recall dovrebbe essere idealmente pari a 1 per un buon classificatore. La recall diventa 1 solo quando il numeratore e il denominatore sono uguali, cioè $TP = TP + FN$, il che significa anche che FN è zero. Quando FN aumenta, il valore del denominatore diventa maggiore del numeratore, di conseguenza il valore di Recall diminuisce.

F_1-Score Idealmente, in un buon classificatore, vogliamo che sia la precision che la recall siano pari a uno, il che significa che FP e FN sono pari a zero. Pertanto, abbiamo bisogno di una metrica che tenga conto sia della precision che della recall. L'F_1-Score è una metrica che tiene conto sia della precision che della recall ed è definito come segue:

$$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall}$$

L'F_1-Score diventa 1 solo quando Precision e Recall sono entrambi pari a 1. L'F_1 è la media armonica di precision e recall ed è una misura più attendibile dell'accuratezza nel caso di dataset con classi sbilanciate.

5.1.4 Tecniche di validazione

I modelli sono stati validati mediante la tecnica di *Hold-out*. Quest'ultima è una tecnica frequentemente utilizzata per assicurarsi che un modello non produca un eccessivo livello di **overfitting** [3] rispetto ai campioni sui quali è sviluppato. Con tale tecnica, un modello viene costruito e adattato utilizzando solo una parte del campione di dati a disposizione, utilizzando la restante porzione per valutarne l'effettivo potere predittivo. In condizioni ideali, il modello funzionerà ugualmente bene su entrambe le porzioni di dati. In caso contrario, è probabile la presenza di un livello di overfitting non trascurabile. I campioni sono stati suddivisi in proporzione 80-20, in cui l'80% viene utilizzato per addestrare il modello (training dataset) e il restante 20% (validation dataset) viene impiegato per valutarne la capacità di predizione.

[3] L'overfitting si verifica quando un modello è troppo complesso rispetto ai dati di addestramento a disposizione. In altre parole, il modello impara troppo bene i dati di addestramento, finendo per memorizzare anche i rumori e le varianze presenti nei dati. Questo rende il modello molto adattato ai dati di addestramento e meno in grado di generalizzare su nuovi dati, cioè di fare previsioni accurate su dati mai visti in precedenza.

5.2 Risultati ottenuti

Una volta discussi i vari setup sperimentali, in questa sezione vengono riportati i risultati ottenuti dai vari addestramenti delle tre reti proposte: LSTM, Transformer, Graph Neural Network. L'obiettivo di questa sezione è di mostrare i valori relativi alle misure di valutazione discusse nella sezione 5.1.3. In particolare, per misurare le performance di un addestramento, si andranno a valutare l'accuracy e l'F_1-Score (che è una combinazione di *precision* e *recall*) di test di ogni addestramento. Per ogni addestramento verrà riportata la relativa accuracy e F_1-Score di test migliore. Per migliore si intende il valore in percentuale più alto registrato in una certa epoca prima che la rete subisca **overfitting**. Verranno quindi ignorate accuracy/F_1 di test nelle quali esse si discostano molto da quelle di training, il che è un indice della presenza di sovradattamento di un modello.

5.2.1 Risultati Long Short Term Memory

In questa sezione vengono riportati i risultati relativi all'addestramento della rete LSTM. I test seguono la politica della scelta degli iperparametri di rete mostrati nella sezione: Iperparametri LSTM. La seguente tabella riporta nella prima colonna l'id del test e come ultime due colonne i migliori risultati ottenuti in termini di accuracy e F_1-Score di test.

id	embed_dim	act LSTM	act Dense	epochs	batch_size	**Accuracy**	**F_1-Score**
1	18	relu	relu	10	32	**69%**	**10%**
2	**50**	relu	relu	**20**	**16**	**62%**	**8%**
3	18	relu	**tanh**	**20**	32	**63%**	**12%**
4	**50**	relu	relu	**20**	32	**63%**	**11%**
5	18	**tanh**	relu	10	32	**69%**	**10%**

Tabella 5.4: La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi alle combinazioni di iperparametri discussi nel paragrafo 5.1.2. La prima riga della tabella, con $id = 1$, riporta la combinazione di iperparametri iniziali. Le successive righe, con $id >= 2$, riportano le modifiche effettuate agli iperparametri per i test successivi, modificando 1 o più iperparametri alla volta (le modifiche sono evidenziate in grassetto).

Il dataset dato in input alla rete LSTM è stato bilanciato. In particolare è presente lo stesso numero di istanze che migliorano la funzione obiettivo e che non la migliorano. Solo per questa rete non vengono riportati i risultati dei test effettuati sul dataset completo, perché essi hanno esiti in termini di F_1-Score molto scarsi, inferiori al 10%. Per tale motivo si è preferito lavorare maggiormente sul dataset bilanciato.

Tuttavia, anche post bilanciamento, i risultati mostrano come la rete non riesca a rilevare correttamente i valori *True Positives* e *False Positives*, questo implica valori

di *Accuracy* accettabili, intorno al 60/70%, perchè la rete riesce a riconoscere bene i *True Negatives* e *False Negatives* e valori dell'F_1 score molto bassi, perchè il numero di "Positive" è piccolo, di conseguenza Precision e Recall sono bassi e così anche l'F_1.
Di seguito vengono illustrate le modifiche agli iperparametri rispetto alla combinazione di default, ovvero quella con $id = 1$ della tabella.
N.B. : per ogni $id >= 2$, vengono evidenziate le differenze con la prima combinazione di iperparametri ($id = 1$).

- $id = 2$: aumentando la dimensione di embedding a 50, il numero di epoche a 20 e diminuendo la dimensione del *batch* a 16, non porta a un miglioramento percepibile, infatti entrambi i valori delle metriche di valutazione sono leggermente inferiori rispetto alla combinazione di iperparametri di default.

- $id = 3$: cambiando la funzione di attivazione dello strato *Dense* in *tanh* e incrementando il numero di epoche a 20, si assiste a un peggioramento dell'accuracy e un miglioramento dell'F_1-Score.

- $id = 4$: aumentando la dimensione dell'embedding a 50 e il numero di epoche a 20, si assiste a un peggioramento dell'accuracy e a un lieve miglioramento dell'F_1-Score.

- $id = 5$: cambiando la funzione di attivazione dello strato LSTM in *tanh* e lasciando invariati gli altri parametri, si ottengono gli stessi risultati di partenza. Questo evidenzia come la funzione di attivazione non influenza la bontà del classificatore.

I valori di F_1 così scarsi hanno forzato l'utilizzo di un'altra tipologia di rete, detta Transformer, i cui risultati sono discussi nella successiva sezione.

5.2.2 Risultati Transformer

In questa sezione vengono riportati i migliori risultati relativi all'addestramento della rete Transformer. I test seguono la politica della scelta degli iperparametri di rete mostrati nella sezione: Iperparametri Transformer. In questo caso, a differenza della rete LSTM, vengono riportate due tabelle, una relativa al dataset di input non bilanciato, la seconda con il dataset bilanciato. La seguente tabella, relativa al dataset completo, riporta nella prima colonna l'id del test e come ultime due colonne i migliori risultati ottenuti in termini di accuracy e F_1-Score di test.

id	embed_dim	num_heads	ff_dim	batch_size	epochs	act	**Accuracy**	**F_1-Score**
1	32	2	32	32	20	relu	**76%**	**36%**
2	32	2	32	32	20	**tanh**	**74%**	**35%**
3	**64**	2	32	32	20	relu	**76%**	**36%**
4	**64**	**4**	32	32	20	**tanh**	**76%**	**35%**
5	32	**4**	32	32	**10**	relu	**76%**	**36%**
6	32	2	**64**	32	**10**	relu	**76%**	**35%**
7	32	2	32	**16**	**10**	relu	**75%**	**35%**

Tabella 5.5: La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi alle combinazioni di iperparametri discussi nel paragrafo 5.1.2. Il dataset dato in input è quello **completo**, non bilanciato. La prima riga della tabella, con $id = 1$, riporta la combinazione di iperparametri iniziali. Le successive righe, con $id >= 2$, riportano le modifiche effettuate agli iperparametri per i test successivi, modificando 1 o più iperparametri alla volta (le modifiche sono evidenziate in grassetto).

In generale possiamo dire che il classificatore si comporta meglio che con approccio LSTM. In particolare, l'accuracy e l'F_1-Score della rete Transformer sono superiori rispetto all'approccio LSTM.
Di seguito vengono illustrate le modifiche, e i relativi risultati, agli iperparametri rispetto alla combinazione di default, ovvero quella con $id = 1$ della tabella.
N.B. : per ogni $id >= 2$, vengono evidenziate le differenze con la prima combinazione di iperparametri ($id = 1$).

- $id = 2$: cambiando la funzione di attivazione in *tanh* si assiste a un lieve miglioramento dell'accuracy e un leggero peggioramento dell'F_1-Score.
- $id = 3$: incrementando la dimensione dell'embedding a 64 si ottengono gli stessi risultati di partenza.
- $id = 4$: aumentando la dimensione dell'embedding a 64, portando il numero di *heads* a 4 e utilizzando la funzione di attivazione *tanh*, non si assiste a un miglioramento percepibile, infatti i risultati sono il linea con quelli di partenza.
- $id = 5$: aumentando il numero di *heads* a 4 i risultati sono gli stessi di partenza. Quindi tale parametro non influenza la bontà del classificatore.

- *id* = 6: incrementando la dimensione dello strato *Feed Forward* a 64 i risultati sono approssimabili a quelli di partenza.

- *id* = 7: diminuendo la dimensione dei *batch* si ha un lieve calo delle metriche, tuttavia sono pressoché simili a quelle di partenza.

La seguente tabella riporta gli stessi test mostrati nella precedente tabella, con la differenza che l'input è stato bilanciato in termini di classi 0 e 1 (non migliorano/-migliorano).

id	embed_dim	num_heads	ff_dim	batch_size	epochs	act	**Accuracy**	F_1**-Score**
1	32	2	32	32	20	relu	**62%**	**50%**
2	32	2	32	32	20	**tanh**	**64%**	**51%**
3	**64**	2	32	32	20	relu	**60%**	**51%**
4	**64**	**4**	32	32	20	**tanh**	**62%**	**50%**
5	32	**4**	32	32	**10**	relu	**61%**	**49%**
6	32	2	**64**	32	**10**	relu	**58%**	**48%**
7	32	2	32	**16**	**10**	relu	**60%**	**49%**

Tabella 5.6: La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi alle combinazioni di iperparametri discussi nel paragrafo 5.1.2. Il dataset dato in input per i 5 test è **bilanciato**. In particolare, è stato considerato lo stesso numero di istanze cui le mosse migliorano (e non migliorano) la funzione obiettivo.
La prima riga della tabella, con $id = 1$, riporta la combinazione di iperparametri iniziali. Le successive righe, con $id >= 2$, riportano le modifiche effettuate agli iperparametri per i test successivi, modificando 1 o più iperparametri alla volta (le modifiche sono evidenziate in grassetto).

In generale possiamo dire che con un input bilanciato i valori di accuracy calano del circa 10%, ma si assiste a un notevole miglioramento dell'F_1-Score di circa 15% in tutti i test eseguiti.
Di seguito vengono illustrate le modifiche, e i relativi risultati, agli iperparametri rispetto alla combinazione di default, ovvero quella con $id = 1$ della tabella.
N.B. : per ogni $id >= 2$, vengono evidenziate le differenze con la prima combinazione di iperparametri ($id = 1$).

- $id = 2$: cambiando la funzione di attazione in *tanh* si assiste a un lieve miglioramento dell'accuracy e dell'F_1-Score.

- $id = 3$: incrementando la dimensione dell'embedding a 64 si ottiene un lieve peggioramento dell'accuracy e un leggero aumento dell'F_1-score.

- $id = 4$: aumentando la dimensione dell'embedding a 64, portando il numero di *heads* a 4 e utilizzando la funzione di attivazione *tanh*, si ottiene un risultato pressoché identico a quello di partenza.

- $id = 5$: aumentando il numero di *heads* a 4 i risultati sono lievemente peggiorati. Quindi l'aumentare di tale parametro non aumenta la bontà del classificatore.

- $id = 6$: incrementando la dimensione dello strato *Feed Forward* a 64 i risultati sono inferiori a quelli di partenza.
- $id = 7$: diminuendo la dimensione dei *batch* si ha un lieve calo delle metriche.

In generale possiamo affermare con certezza che l'approccio Transformer si mostra più efficace di LSTM nel classificare una sequenza di mosse. I risultati, anche con input sbilanciato, sono nettamente migliori di quelli con rete LTSM.
Inoltre, è apprezzabile una differenza concreta tra i risultati ottenuti con dataset completo e bilanciato nella rete Transformer. In particolare l'F_1-Score è nettamente più alto, mentre l'accuracy decresce ma ha comunque un valore accettabile. Tuttavia, nonostante il netto miglioramento, l'F_1 si assesta su un 50%, che non è un buon valore, soprattutto se si parla di una classificazione binaria. In definitiva i risultati altalenanti ottenuti con le reti LSTM e Transformer hanno spinto all'utilizzo di una rete ad hoc per la gestione dei grafi, che possa rispondere alla complessità del task in maniera più efficace. Quest'ultima è la **Graph Neural Network**, i cui risultati vengono discussi nella prossima sezione.

5.2.3 Risultati Graph Neural Network

In questa ultima sezione vengono riportati i risultati relativi all'addestramento della rete GNN. Anche per ques'ultima architettura sono state provate diverse combinazioni di iperparametri per verificare l'andamento di accuracy e F_1-Score. In questo caso però, avendo 17 classificatori, la scelta dell'accuracy e F_1 migliore segue la seguente logica. Per ognuna delle 17 mosse viene riportata la combinazione di iperparametri che ha portato accuracy e F_1 migliori.
Anche per la rete GNN la politica della scelta degli iperparametri si allineano con quanto detto nella sezione Iperparametri GNN. Per quanto riguarda le classi, in questo caso ne abbiamo 5, a differenza di LSTM e Transformer che erano di classe binaria. Le 5 classi, come visto nella sezione Dataset GNN, sono "migliora molto", "migliora poco", "neutro", "peggiora poco" e "peggiora molto".
I vari test sono stati eseguiti sul dataset **completo** e sul dataset **bilanciato**, contenente lo stesso numero di elementi per ognuna delle 5 classi.
La seguente tabella riporta i risultati relativi al dataset **completo**, non bilanciato, per ogni mossa di *destroy-repair*.

moves	id	k	batch	out	lr	ly	epoch	Accuracy	F_1
WorstTimeDestroyCustomer	10	7	64	0.1	0.05	3	100	74%	67%
ZoneDestroyCustomer	8	7	64	0.1	0.1	3	50	76%	66%
GreedyRouteRemoval	4	7	64	0.1	0.01	5	50	81%	78%
LongestWaitingTimeDestroyStation	1	7	64	0.1	0.01	3	50	84%	82%
DeterministicBestRepairStation	7	7	64	0.05	0.01	3	50	62%	61%
GreedyDestroyCustomer	1	7	64	0.1	0.01	3	50	64%	65%
ProbabilisticBestRepairStation	1	7	64	0.1	0.01	3	50	71%	70%
ShawDestroyCustomer	7	7	64	0.05	0.01	3	50	65%	65%
DemandBasedDestroyCustomer	1	7	64	0.1	0.01	3	50	69%	67%
ProximityBasedDestroyCustomer	6	7	64	0.5	0.01	3	50	64%	63%
null	10	7	64	0.1	0.05	3	100	69%	69%
ProbabilisticWorstRemovalCustomer	2	15	64	0.1	0.01	3	50	71%	70%
GreedyRepairCustomer	2	15	64	0.1	0.01	3	50	72%	72%
WorstDistanceDestroyCustomer	9	7	128	0.1	0.01	3	50	68%	68%
RandomRouteDestroyCustomer	9	7	128	0.1	0.01	3	50	75%	73%
RandomDestroyStation	10	7	64	0.1	0.05	3	100	77%	76%
TimeBasedDestroyCustomer	5	7	64	0.1	0.01	2	50	67%	65%

Tabella 5.7: La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi a ogni mossa di *destroy-repair*. In particolare per ogni mossa viene riportata la migliore combinazione di iperparametri. La migliore combinazione è quella che presenta metriche di valutazione più alte. La colonna *id* rappresenta la combinazione vista nella sezione 5.1.2, le successive colonne descrivono gli iperparametri e le metriche di valutazione.

I risultati mostrano come l'efficacia del classificare sia pressoché simile tra tutte le mosse tranne che per alcune mosse che vengono classificate più correttamente di altre. Questo perché, non essendo il dataset bilanciato, alcune mosse contengono

molte istanze appartenente alla stessa classe, per cui risultano meno complesse da classificare. A conferma di ciò si è eseguito lo stesso test ma con dataset bilanciato, con pari numero di istanze in ognuna delle 5 classi, affinché sia possibile valutare differenze nelle metriche di valutazione.

moves	id	k	batch	out	lr	ly	epoch	Accuracy	F_1
WorstTimeDestroyCustomer	1	7	64	0.1	0.01	3	50	64%	65%
ZoneDestroyCustomer	5	7	64	0.1	0.01	2	50	55%	52%
GreedyRouteRemoval	7	7	64	0.05	0.01	3	50	64%	62%
LongestWaitingTimeDestroyStation	4	7	64	0.1	0.01	5	50	67%	65%
DeterministicBestRepairStation	5	7	64	0.1	0.01	2	50	66%	64%
GreedyDestroyCustomer	3	7	32	0.1	0.01	3	50	65%	64%
ProbabilisticBestRepairStation	1	7	64	0.1	0.01	3	50	70%	68%
ShawDestroyCustomer	6	7	64	0.5	0.01	3	50	64%	62%
DemandBasedDestroyCustomer	3	7	32	0.1	0.01	3	50	62%	62%
ProximityBasedDestroyCustomer	5	7	64	0.1	0.01	2	50	66%	65%
null	10	7	64	0.1	0.05	3	100	63%	61%
ProbabilisticWorstRemovalCustomer	4	7	64	0.1	0.01	5	50	68%	66%
GreedyRepairCustomer	2	15	64	0.1	0.01	3	50	62%	60%
WorstDistanceDestroyCustomer	2	15	64	0.1	0.01	3	50	66%	66%
RandomRouteDestroyCustomer	5	7	64	0.1	0.01	2	50	61%	61%
RandomDestroyStation	7	7	64	0.05	0.01	3	50	64%	61%
TimeBasedDestroyCustomer	4	7	64	0.1	0.01	5	50	58%	56%

Tabella 5.8: La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi a ogni mossa di *destroy-repair*. In particolare per ogni mossa viene riportata la migliore combinazione di iperparametri. La migliore combinazione è quella che presenta metriche di valutazione più alte. La colonna *id* rappresenta la combinazione vista nella sezione 5.1.2, le successive colonne descrivono gli iperparametri e le metriche di valutazione.

I risultati mostrano come le mosse di *destroy-repair* vengano classificate con una simile efficacia. Le metriche di valutazione riportano differenze da mossa a mossa. Sul dataset **completo** l'accuracy va da un minimo del **62%** per la mossa *DeterministicBestRepairStation* a un massimo dell'**84%** per la mossa *LongestWaitingTimeDestroyStation*. L'F_1-Score invece va da un minimo del **61%** per la mossa *DeterministicBestRepairStation* a un massimo dell'**82%** per la mossa *LongestWaitingTimeDestroyStation*.
Sul dataset **bilanciato**, ovvero con un pari numero di istanze per ognuna delle 5 classi, le metriche presentano delle differenze con i test effettuati con dataset completo. In questo caso l'accuracy va da un minimo del **55%** per la mossa *ZoneDestroyCustomer* a un massimo del **70%** per la mossa *ProbabilisticBestRepairStation*. Per quanto riguarda l'F_1-Score, questo va da un minimo del **52%** per la mossa *ZoneDestroyCustomer* a un massimo del **68%** per la mossa *ProbabilisticBestRepairStation*.

L'accuracy media con il dataset **completo** è del **71%**. Per quanto riguarda la media degli F_1-Score, questa è del **69%**.
L'accuracy media con il dataset **bilanciato** è del **63.82%**. La media degli F_1-Score è del **62.35%**. I risultati sono riassunti nella seguente tabella:

Dataset	Worst Accuracy	Best Accuracy	Worst F_1	Best F_1	**Mean Accuracy**	**Mean F_1**
Completo	62%	84%	61%	82%	**71%**	**69%**
Bilanciato	55%	70%	52%	68%	**63.82%**	**62.35%**

La rete GNN si comporta in maniera diversa pre e post bilanciamento del dataset. In particolare, le metriche sul dataset bilanciato mostrano un calo di Accuracy e F_1-Score. Tale situazione è giustificabile dal fatto che, dopo il bilanciamento delle 5 classi, il dataset ha molte meno istanze rispetto a quello di partenza, infatti si passa da **67668** istanze del dataset **completo** a **16313** istanze del dataset **bilanciato**. Per un task di classificazione multi-classe è auspicabile che la rete venga addestrata su un numero di istanze numeroso per apprendere in maniera più efficace. La rete addestrata con dataset bilanciato è infatti, dopo l'epoca 25, soggetta a **overfitting**. Tale fenomeno è percepibile dal fatto che l'accuracy e F_1-Score di training sale molto (prossima al 90%), a differenza delle metriche di test che sono in media del 63%.
D'altra parte, un minor numero di istanze significa anche minor tempo di addestramento. Di fatto, con il dataset bilanciato, la rete impiega 12 minuti e 32 secondi per effettuare l'addestramento e il test, invece, il dataset completo impiega 32 minuti e 23 secondi (su 17 dataset). I tempi sono calcolati facendo una media tra le tempistiche dei 10 test effettuati.

Dataset	**Numero di istanze**	**Tempo impiegato**
Balanced	16313	$\simeq 12minuti$
Unbalanced	67668	$\simeq 32minuti$

In definitiva i risultati ottenuti su dataset bilanciato sono più attendibili rispetto a quelli ottenuti su dataset completo.
In generale, bilanciare le classi di un dataset prima di addestrare una rete neurale è importante perché i modelli di apprendimento tendono a prestare più attenzione alle classi con maggiore presenza nei dati di addestramento. In un dataset con classi sbilanciate, il modello potrebbe essere "ingannato" a privilegiare la classe dominante, migliorando l'accuratezza su quella classe ma peggiorando la capacità di riconoscere le classi meno frequenti. Bilanciando le classi del dataset, fornisce al modello una rappresentazione più equilibrata delle diverse classi e si aumenta quindi la possibilità che il modello riconosca tutte le classi con una certa precisione.

5.3 Confronto

I tre approcci utilizzati sono molto differenti, sia per quanto riguarda i dataset forniti in input, sia per risultati ottenuti.

Input

- LSTM prende in input un unico file csv con istanze di classe **binaria** 0 o 1 e valuta se la **combinazione di 4 mosse** applicata alla soluzione iniziale non migliora/migliora la funzione obiettivo. Inoltre il test è effettuato post bilanciamento delle istanze di classe 1 e 0.
- Transformer prende in input un unico file csv con istanze di classe **binaria** 0 o 1 e valuta se la **combinazione di 4 mosse** applicata alla soluzione iniziale non migliora/migliora la funzione obiettivo. In questo caso il test è effettuato sia per il dataset completo (sbilanciato), sia per il dataset con classi bilanciate.
- GNN prende in input una coppia di file: **.g** per le informazioni relative alla soluzione iniziale e un **csv** per le label associate. Ogni coppia di file è dato in pasto a ognuno dei 17 classificatori, uno per ogni tipologia di mossa. Quindi ogni mossa viene classificata **singolarmente** e non in sequenza di 4 come nel caso di LSTM e Transformer.

Risultati

- LSTM ottiene un'accuracy di test massima del **69%** con la combinazione di iperparametri con $id = 1$ o con $id = 5$. Il discorso cambia per l'F_1-Score, che ha un valore massimo del **12%** per la combinazione con $id = 3$.
- Transformer raggiunge un'accuracy di test massima del **76%** per il dataset completo e del **64%** per il dataset con classi bilanciate. L'F_1-Score massimo è invece del **36%** per il dataset completo e **51%** per il dataset con classi bilanciate.
- GNN ottiene un'accuracy di test massima dell'**84%** con il dataset completo e del **70%** per il dataset bilanciato. L'F_1-Score più alto per il dataset completo è dell'**82%**, mentre per il dataset con classi bilanciate raggiunge il **68%**.

In conclusione non si può fare un paragone concreto tra i tre approcci utilizzati perché, come abbiamo visto, gli input sono diversi e la classificazione non è sempre binaria. Tuttavia si può affermare che la rete LSTM, con lo stesso input di classe binaria bilanciato, ha una capacità predittiva peggiore dell'approccio con rete Transformer. Questi ultimi due approcci sono gli unici davvero confrontabili perché adottano la stessa strategia per la gestione degli input e delle classi. Per quanto riguarda la GNN, per quest'ultima si è ragionato in maniera differente. Infatti vengono addestrati 17 classificatori, relativi a ogni mossa presente nel campo "Moves" del dataset. Questa volta la classificazione è multi-classe, con 5 classi: "migliora molto", "migliora poco", "neutro", "peggiora poco" e "peggiora molto", numerate da 4 a 0. Nonostante le classi passino da 2 a 5 rispetto ai due approcci precedenti, il classificatore ottiene delle buone metriche di valutazione, maggiori del 60% sul un dataset bilanciato. Se si dovesse scegliere un classificatore che possa assistere un'euristica di ottimizzazione con una soluzione iniziale come per questo task, verrebbe naturale scegliere una GNN. Tale scelta non dipenderebbe solo dalle metriche che in generale si mostrano migliori rispetto agli altri approcci, ma anche dalla natura del dataset. Infatti nel caso delle LSTM e Transformer i nodi del grafo erano considerati come stringhe di testo, invece nelle GNN si va a creare, tramite la conversione in file **g**, una struttura che rappresenta un vero e proprio grafo. La scelta dell'architettura migliore è quindi strettamente legata alla configurazione iniziale dei dati. Per questo task l'utilizzo della GNN si è essenzialmente dimostrato più adeguato.

Capitolo 6

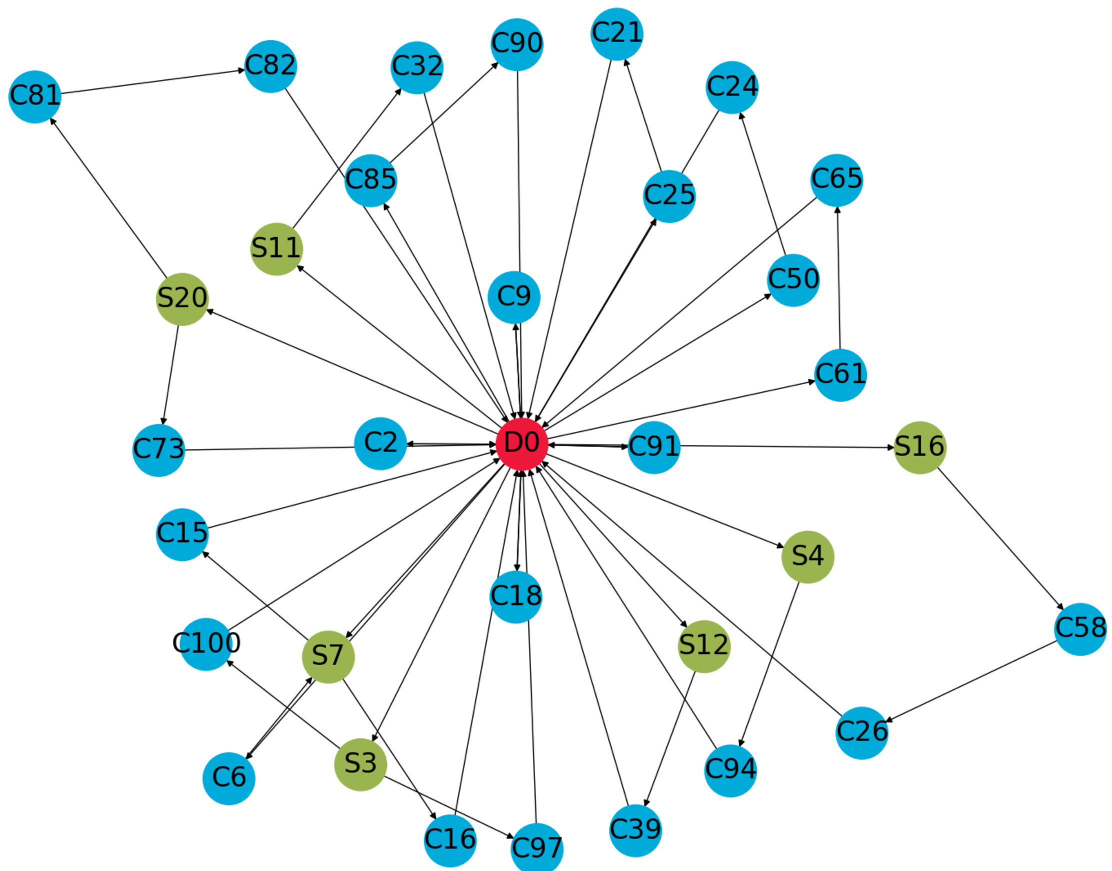
Conclusione e sviluppi futuri

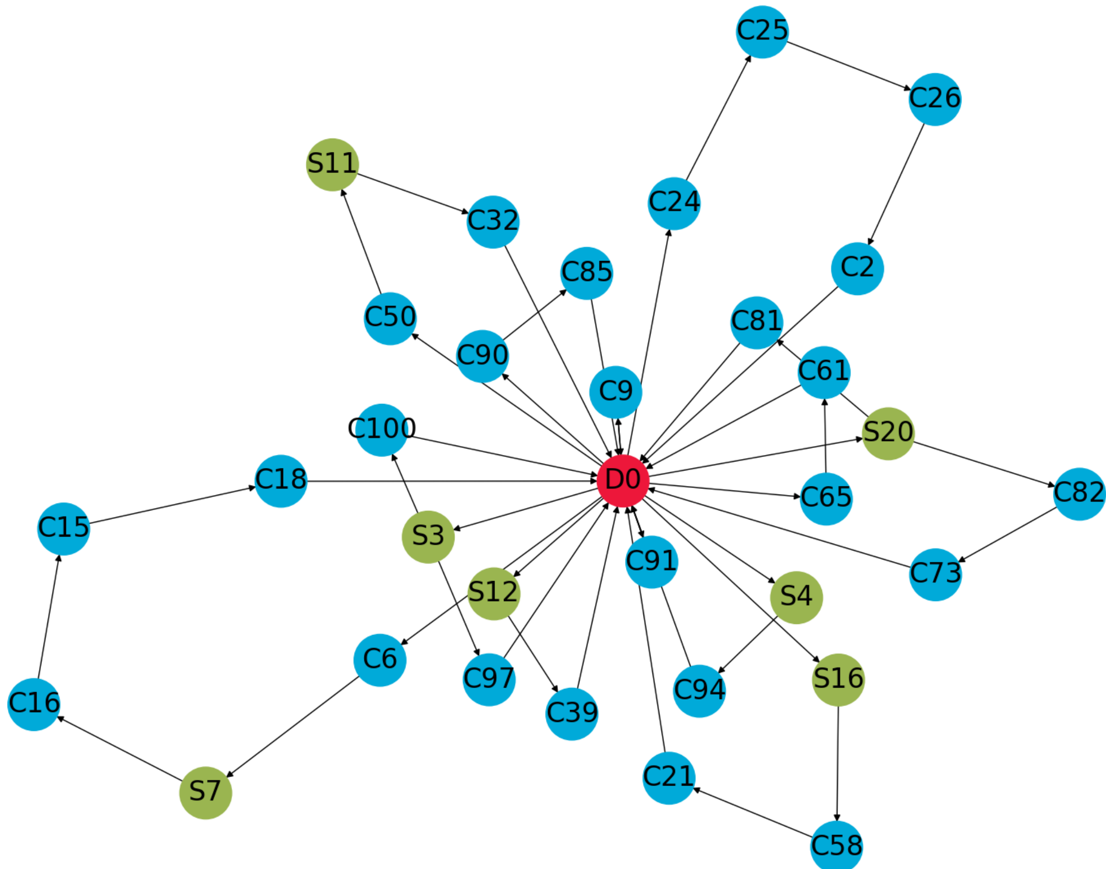
In questa tesi sono state esplorate varie tecniche di Machine Learning che possono fornire un supporto alle euristiche di ottimizzazione.
Si è partiti analizzando la sorgente dati, esplorando l'intero dataset e individuando le feature più promettenti per l'addestramento delle reti neurali. La fase esplorativa dei dati ha previsto analisi statistiche, come media e varianza, minimo e massimo, per tutti i campi di interesse del dataset.
La fase successiva è stata quella di sviluppare modelli predittivi sulla base di tre architetture di reti neurali: la LSTM (Long Short Term Memory), i Transformer e le GNN (Graph Neural Network). È stata effettuata una fase di valutazione sperimentale che ha mostrato come le GNN riescano ad ottenere i risultati migliori, raggiungendo in media $Accuracy = 71\%$ e $F_1 - score = 69\%$ nel caso di dataset completo e $Accuracy = 63.82\%$ e $F_1 - score = 62.35\%$ con il dataset con classi bilanciate. Si ritiene dunque di aver centrato l'obiettivo di addestrare un buon classificatore, ottenendo delle metriche di test discretamente alte. Si potrebbe pensare di utilizzare i modelli pre-addestrati per sostenere delle euristiche di ottimizzazione.
A tal proposito, come sviluppi futuri, si propone di integrare il codice che sceglie le mosse di *destroy-repair* da applicare alla soluzione iniziale, con i modelli ottenuti dagli addestramenti delle GNN. In particolare, l'idea è quella di creare un algoritmo che, prima di applicare una certa mossa, ne esegue la classificazione, se la mossa viene classificata come migliorante allora viene eseguita, altrimenti passa alla successiva. L'integrazione dei due approcci potrebbe fornire un solido strumento per problemi di ottimizzazione dei grafi, dove solitamente la soluzione viene cercata con metodi *Greedy*.

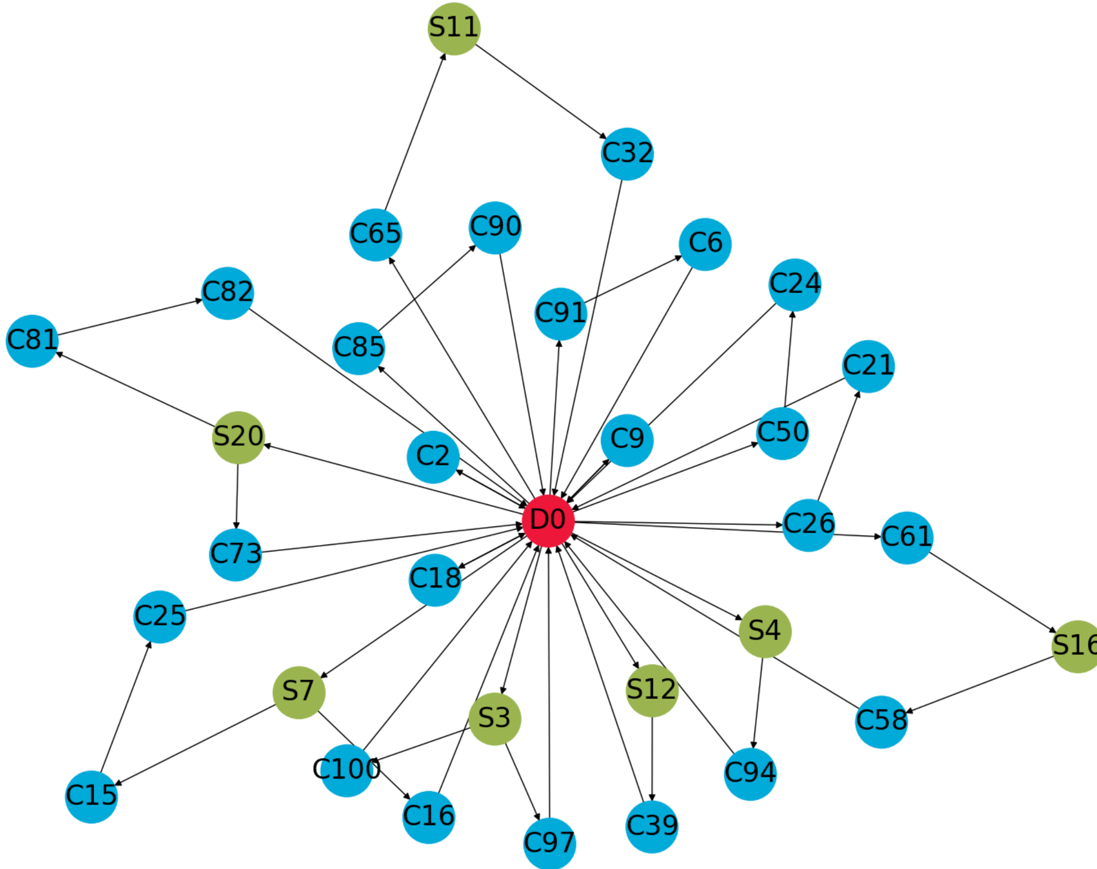
Capitolo 7

Appendice

Di seguito vengono riportate le rappresentazioni di alcuni grafi contenuti nel dataset. Per il codice Python per la generazione dei grafi si fa riferimento alla sezione 2.2.1.

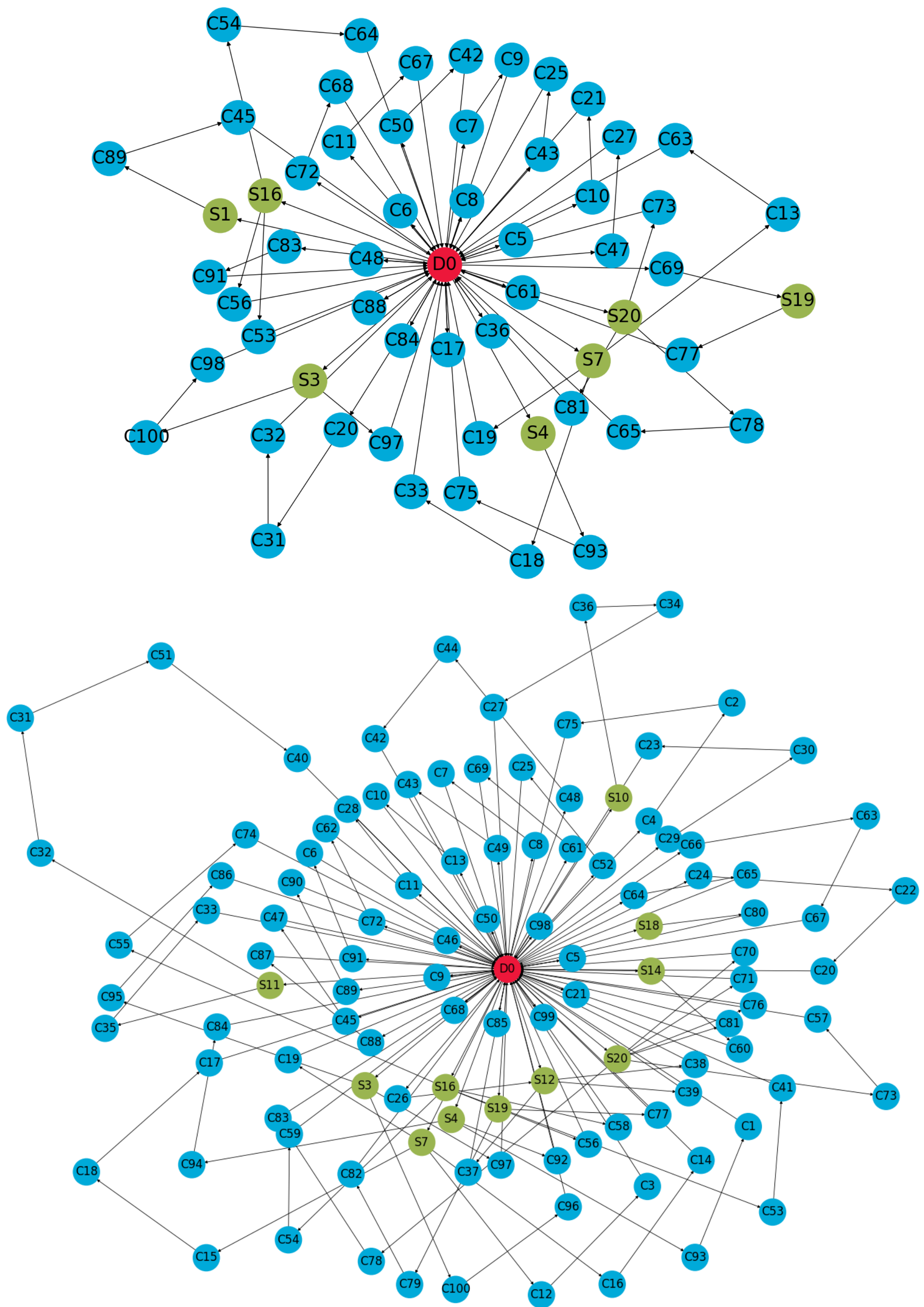



C25
C26
S11
C24
C32
C85
C2
C50
C81
C90
C61
C9
C100
S20
C18
D0
C82
C15
S3
C65
C91
C73
S12
S4
C6
C97
C94
S16
C16
C39
C21
S7
C58


S11
C32
C65
C90
C6
C24
C82
C91
C81
C85
C21
S20
C9
C50
C2
D0
C26
C61
C73
C18
C25
S4
S16
S7
S12
S3
C58
C100
C94
C15
C16
C39
C97

C15
C18
C9
C16
C26
S20
C6
C81
C82
C85
S7
C2
C73
C90
C100
D0
S3
C65
S11
C97
C21
C61
C50
C58
C91
C25
C24
C94
S16
S4
C39
C32
S12

C16
S7
C15
C9
C18
C6
S20
C81
C82
C85
C100
C2
C91
C73
C90
C97
D0
S3
C94
C65
C50
S11
C61
C58
C21
C24
S16
C26
C32
C25
S12
C39



Bibliografia e Sitografia

[1] Roman Lutz. Adaptive large neighborhood search. *Ulm University*, page 107, 08 2014.

[2] https://en.wikipedia.org/wiki/Exploratory_data_analysis, urldate = 2006-10-01.

[3] https://didattica-2000.archived.uniroma2.it//MMOD1bis/deposito/AlgEuristici.pdf.

[4] https://aditi-mittal.medium.com/understanding-rnn-and-lstm-f7cdf6dfc14e.

[5] https://towardsdatascience.com/transformers-141e32e69591.

[6] Ashish Vaswani. Attention is all you need. *31st Conference on Neural Information Processing Systems*, page 15, 12 2017.

[7] https://www.matteotroia.it/che-cose-un-grafo-e-come-si-rappresenta/.

[8] https://distill.pub/2021/gnn-intro/.

[9] J. Gilmer. Neural message passing for quantum chemistry. *Proceedings of the 34th International Conference on Machine Learning*, 2017.

[10] Tong Yu. Hyper-parameter optimization: A review of algorithms and applications. *Department of AI and HPC*, page 56, 03 2020.

Elenco delle figure

2.1 La figura illustra il funzionamento di un algoritmo di *destroy-repair*. Fonte: [1] . . . 12
2.2 Figura di esempio di un grafo ottenuto dalla funzione **draw(G)**. Essa rappresenta graficamente l'insieme percorsi relativi a un'istanza del dataset, quindi a una riga del file DB-Output.csv. Il dataset contiene 21157 grafi, ognuno dei quali è più o meno complesso come quello in figura. Nell'Appendice verranno mostrate altre rappresentazioni di grafi presenti nel dataset. . . . 16

3.1 Rappresentazione dell'architettura di una LSTM. Fonte: [4] . . . 23
3.2 Architettura di una rete Transformer. Fonte: [6] . . . 26
3.3 Immagine rappresentante un grafo semplice con 6 nodi e 7 archi. Fonte: [7] . . . 28
3.4 Rappresentazione di un grado con 8 nodi e 8 archi. La matrice di adiacenza sarà una lista contenente 8 liste, una per ogni arco del grafo.
Fonte: [8] . . . 30
3.5 Un singolo strato di una GNN. L'input è un grafo e ogni componente (V,E,U) viene aggiornato da una MLP per produrre un nuovo grafo. Ogni pedice della funzione indica una funzione separata per un diverso attributo del grafo all'n-esimo strato di un modello GNN.
Fonte:[8] . . . 31
3.6 Predizione globale . . . 32
3.7 Recap della struttura di una Graph Neural Network . . . 32

Elenco delle tabelle

4.1 Versione dei software utilizzati per le reti **LSTM** e **Transformer** . . 33
4.2 Versione dei software utilizzati per la **Graph Neural Network** . . . 33

5.1 Iperparametri testati per l'addestramento della rete LSTM 41
5.2 Nella tabella vengono riportati gli iperparametri testati per l'addestramento della rete Transformer 42
5.3 Nella tabella vengono riportati gli iperparametri testati per l'addestramento della Graph Neural Network 43
5.4 La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi alle combinazioni di iperparametri discussi nel paragrafo 5.1.2. La prima riga della tabella, con $id = 1$, riporta la combinazione di iperparametri iniziali. Le successive righe, con $id >= 2$, riportano le modifiche effettuate agli iperparametri per i test successivi, modificando 1 o più iperparametri alla volta (le modifiche sono evidenziate in grassetto). 46
5.5 La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi alle combinazioni di iperparametri discussi nel paragrafo 5.1.2. Il dataset dato in input è quello **completo**, non bilanciato. La prima riga della tabella, con $id = 1$, riporta la combinazione di iperparametri iniziali. Le successive righe, con $id >= 2$, riportano le modifiche effettuate agli iperparametri per i test successivi, modificando 1 o più iperparametri alla volta (le modifiche sono evidenziate in grassetto). 48
5.6 La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi alle combinazioni di iperparametri discussi nel paragrafo 5.1.2. Il dataset dato in input per i 5 test è **bilanciato**. In particolare, è stato considerato lo stesso numero di istanze cui le mosse migliorano (e non migliorano) la funzione obiettivo.
La prima riga della tabella, con $id = 1$, riporta la combinazione di iperparametri iniziali. Le successive righe, con $id >= 2$, riportano le modifiche effettuate agli iperparametri per i test successivi, modificando 1 o più iperparametri alla volta (le modifiche sono evidenziate in grassetto). 49

5.7 La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi a ogni mossa di *destroy-repair*. In particolare per ogni mossa viene riportata la migliore combinazione di iperparametri. La migliore combinazione è quella che presenta metriche di valutazione più alte. La colonna *id* rappresenta la combinazione vista nella sezione 5.1.2, le successive colonne descrivono gli iperparametri e le metriche di valutazione. 51
5.8 La Tabella riporta i risultati in termini di Accuracy e F_1-Score relativi a ogni mossa di *destroy-repair*. In particolare per ogni mossa viene riportata la migliore combinazione di iperparametri. La migliore combinazione è quella che presenta metriche di valutazione più alte. La colonna *id* rappresenta la combinazione vista nella sezione 5.1.2, le successive colonne descrivono gli iperparametri e le metriche di valutazione. 52