Relazione Progetto Classificazione Binaria Grafi

Mattia Matucci

1 Introduzione

Il progetto prevede la realizzazione di una neural network per la classificazione binaria di grafi e la sua validazione attraverso il dataset NCI1.

Il dataset a disposizione appartiene al dominio chimico informatico. Ciascun grafo in input viene usato per descrivere un composto: i vertici corrispondono agli atomi, gli archi rappresentano i legami tra gli atomi. Ad ogni grafo è assegnato un valore binario in output, per esprimere il valore non tumorale o tumorale del composto. Per indicare il tipo di atomo a ciascun vertice è associata un etichetta, codificata attraverso un vettore one-hot.

Per risolvere il task assegnato, la rete concatena un classificatore, tipicamente un layer MLP (*Multi Layers Perceptron*), a un modello che scambia le informazioni tra i vertici del grafo, in questo caso una sequenza di layers GIN (*Graph Isomorphism Network*). La definizione del layer GIN è riportata nell'equazione seguente.

$$\mathbf{x}'_i = h_{\mathbf{\Theta}}\left((1+\epsilon)\cdot \mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \mathbf{x}_j\right)$$

La rete e i layers sono implementati estendendo la classe *Module* della libreria *Pytorch*, fa eccezione il layer GIN che invece estende la classe *MessagePassing* della libreria *Pytorch Geometric*. Quest'ultima libreria è fondamentale per il progetto perché aggiunge a *Pytorch* le funzioni per lavorare con i grafi.

Lo scopo principale della rete è quello di connettere i layers, delegandogli le operazioni e raccogliendo i loro output. A causa della procedura readout-concat, la fase di raccolta dei risultati assume ancor più importanza, poiché il classificatore riceve in input un vettore che contiene l'output di ciascun layer GIN.

Ho speso molto tempo per la validazione della rete, in particolar modo per selezionare la migliore configurazione degli iperparametri ed evitare l'overfitting del training set. Per ridurre il tempo d'esecuzione complessivo ho validato solo un sottoinsieme d'iperparametri, mentre gli altri sono stati scelti in accordo al peaper di riferimento. Per evitare l'overfitting ho applicato la procedura di early stopping, la sua introduzione nel codice non è stata semplice, ma ne parlerò meglio nella sezione 2.4.

2 Implementazione

Questo capitolo descrive l'implementazione della rete neurale e delle funzioni utili al suo training e alla sua validazione. Insieme a ciò sono riportate tutte le difficoltà incontrate e le soluzioni trovate durante lo sviluppo del codice.

Prima di scendere nei dettagli del codice penso che sia utile mettere in chiaro che il task della rete è la classificazione binaria di grafi, che è stata allenata con la procedura di early stopping e che i valori degli iperparametri sono stati validati con la cross validation.

2.1 Gestione dataset

La prima operazione per utilizzare il dataset NCI1 è scaricarlo, ciò è risultato semplice dal momento che *Pytorch Geometric* ne contiene una copia internamente.

Ciascun grafo del dataset è rappresentato dalle etichette associate ai vertici, dagli archi e dall'etichetta binaria da classificare. È inoltre interessante osservare che i grafi sono rappresentati in forma diretta anche se sono indiretti.

Per valutare correttamente la neural network è fondamentale partizionare il dataset. La tipologia di split dipende dalla procedura di allenamento e dalla metodologia di model selection. Poiché inizialmente non avevo definito nessuna delle due procedure, durante la scrittura del codice è stato necessario modificare più volte la funzione di suddivisone del dataset.

La versione finale della funzione prevede training, validation e test set. Nella fase di model selection viene usato solo il training set mentre l'intero dataset è utilizzato durante la fase di model assessment. Il validation set è impiegato per calcolare il criterio di arresto dell'early stopping. A seguito di queste scelte ho effettuato lo split secondo queste proporzioni $(\frac{2}{3}, \frac{1}{6}, \frac{1}{6})$.

Considerato che è possibile maneggiare il dataset come una lista di grafi, la funzione di suddivisone può essere scritta sfruttando lo slice. Tale rappresentazione è invece un problema durante il training della rete, poiché richiede un for esplicito. Per ovviare al problema emerso il dataset può essere trasformato in un grafo non connesso con componenti i grafi del dataset. Sul grafo ottenuto è quindi possibile invocare le stesse funzioni scritte per manipolare un singolo grafo. La funzione di *Pytorch Geometric* che implementa questa trasformazione si chiama *DataLoader*.

2.2 Layers

I layer implementati per questa neural network sono: GIN, MLP e Linear. Il layer GIN gestisce le interazioni tra i vertici di un grafo, il layer MLP rimuove la linearità alle interazioni tra i vertici del modello GIN e classifica i grafi, infine il layer Linear applica le trasformazioni lineari ai vettori.

Il layer GIN è definito a partire dalla classe *MessagePassing* di *Pytorch Geometric* mentre gli altri sono definiti a partire dalla classe *Module* di *Pytorch*. Questa differenza non si riflette nel codice, infatti tutti i layers sono delle classi i cui metodi sono: il costruttore per dichiarare i parametri del layer e il forward per definire le operazioni.

Un aspetto cruciale della definizione dei parametri è la loro inizializzazione, la scelta di un metodo piuttosto che un altro può portare ad avere reti completamente diverse. Ho riscontrato questa problematica all'inizio, quando l'inizializzazione dei parametri del layer Linear creava un rete che produceva pessimi risultati. Per individuare e risolvere il problema ho costruito una rete con i layer di *Pytorch*, dopo di che gli ho sostituiti progressivamente con quelli che ho scritto io.

Nei layers implementati il metodo che si occupa di definire i parametri può prendere in input delle flag, che sono utili a cambiare il comportamento della rete neurale. Le flag sono degli iperparametri, di conseguenza durante la model selection verrà individuata la loro migliore combinazione.

2.2.1 Linear

Il layer Linear applica la trasformazione lineare al vettore in input $\mathbf{x}$.

$$\mathbf{y} = \mathbf{W}^\mathsf{T}\mathbf{x} + \mathbf{b}$$

$\mathbf{W}$ e $\mathbf{b}$ sono i parametri del layer. Indicando con n la lunghezza del vettore in input e m la lunghezza del vettore in output, $\mathbf{W} \in \mathbb{R}^{n \times m}$ è la matrice dei pesi inizializzata estraendo dall'intervallo $(\frac{-1}{\sqrt{n}}, \frac{1}{\sqrt{n}})$ e $\mathbf{b} \in \mathbb{R}^m$ è il vettore bias inizializzato estraendo dall'intervallo $(\frac{-1}{\sqrt{m}}, \frac{1}{\sqrt{m}})$.

2.2.2 MLP

Il layer MLP è ottenuto concatenando a dei neuroni un layer Linear. Tipicamente un neurone è formato dal layer Linear e dalla funzione di attivazione, tuttavia in questo caso è risultato utile normalizzare l'output della trasformazione lineare prima di applicare la funzione di attivazione. A causa di ciò il layer Linear non applica direttamente la funzione di attivazione.

2.2.3 GIN

Il layer GIN preso in input un grafo (eventualmente non connesso) per ogni vertice $\mathbf{x}_i$ aggrega, tipicamente sommando, le informazioni del vicinato $\mathcal{N}(i)$. In un grafo diretto il vicinato è l'insieme dei vertici $\mathbf{x}_j$ per cui esiste un arco $(\mathbf{x}_j, \mathbf{x}_i)$. A questo risultato sono aggiunte le informazioni contenute dal vertice $\mathbf{x}_i$ eventualmente con un peso diverso determinato da ϵ. Una volta aggregate, le informazioni contenute in ogni vertice, vengono passate al layer MLP con la funzione di attivazione ReLU, per applicare la non linearità $h_{\mathbf{\Theta}}$.

$$\mathbf{x}'_i = h_{\mathbf{\Theta}}\left((1+\epsilon)\cdot \mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \mathbf{x}_j\right)$$

2.3 Neural Network

Lo scopo principale della classe Neural Network è quello di connettere i layers, delegandogli le operazioni e raccogliendo i loro output. Considerato che ogni rete può essere usata come layer di un rete più complessa, ho sviluppato la classe estendendo quella base di *Pytorch*. Di conseguenza i metodi definiti sono uguali a quelli delle classi layers.

Il costruttore specifica la struttura della rete, vale a dire una sequenza di layers e un classificatore, lasciando però all'utente il compito di definirli. In questo progetto la chiamata al costruttore della classe Neural Network viene eseguita solo all'interno della funzione *model definition*. All'interno di tale funzione sono pertanto dichiarati la sequenza di layers GIN e il classificatore. Nel resto del codice quando è necessario creare un'istanza della neural network viene chiamata la funzione appena descritta.

Il metodo forward oltre a gestire le connessioni tra i vari layer della rete, implementa la procedura readout-concat. Già descritta nell'introduzione questa tecnica prevede di passare al classificatore un vettore che contiene al suo interno un immagine del grafo dopo ogni layer della sequenza. Questa procedura è stata scritta sfruttando le funzioni di *Pytorch*.

La procedura di readout-concat è una scelta di design di questa rete che non può essere aggirata, così come il compito che svolge, cioè la classificazione binaria. Per implementare la classificazione binaria l'output del classificatore viene passato alla funzione Sigmoide, che ha come immagine l'intervallo $[0, 1]$. In questo modo la previsione della rete è la probabilità che il grafo abbia etichetta 1. Se il valore associato al grafo è superiore a 0.5, durante il calcolo dell'accuracy al grafo sarà associata l'etichetta 1, altrimenti 0.

2.4 Model Training

Similmente a quanto accaduto per il metodo di split del dataset, anche il metodo di training ha subito numerosi cambiamenti. Con l'ultima modifica è stata introdotta la procedura di early stopping. Nonostante questa introduzione abbia modificato il lifecycle della funzione, questa ha lasciato inalterato il suo scopo, cioè allenare il modello sul training set. L'attuale lifecycle prevede la continua alternanza tra la fase di training e di validazione della rete.

Aver definito chiaramente l'obiettivo di questa fase è fondamentale, poiché delega all'esterno la decisione riguardo il tipo di allenamento. In questo modo è possibile chiamare la stessa funzione

durante le fasi di model selection e di model assessment, che in questo progetto sono sviluppate con le tecniche di cross validation e hold out, rispettivamente.

I parametri in input della funzione sono: il modello, il training set, il validation set e altri parametri per modificare l'allenamento, tra questi riporto il learning rate e la patience, in quanto iperparametri da validare. La funzione restituisce in output: la miglior validation accuracy, il valore dei parametri al momento del calcolo della migliore accuracy, e lo sviluppo dell'accuracy sul training set e sul validation set.

La prima operazione prevista della funzione è la trasformazione della lista di grafi nell'equivalente grafo non connesso. Tale trasformazione è applicata ad entrambi gli insiemi di dati, visto che voglio tener traccia dell'evoluzione dell'accuracy su entrambi gli insiemi, effettuarla in questo momento evita la necessità di doverla ripetere ad ogni iterazione. Conclusa la fase di set up, la funzione inizia il proprio lifecycle controllando la sua condizione d'arresto, cioè aver raggiunto la patience. Tale condizione è ottenuta quando la validation accuracy non migliora per un numero di epoche consecutive pari al valore indicato dalla patience. La scelta della patience è dunque fondamentale per evitare l'underfitting e l'overfitting dei dati.

Valutata la condizione d'uscita inizia la fase di training. La funzione *DataLoader* permette di specificare la tipologia di discesa del gradiente. In questo progetto ho scelto la tecnica mini-batching, poiché aggiorna i parametri della rete più volte durante un'epoca. Un altro vantaggio della funzione è quello di restituire le batch nella forma desiderata, ovvero in grafi non connessi. Sequenzialmente ogni batch viene passata alla rete, dopo di che viene calcolata la loss, che in questo caso è la *Binary cross entropy*, e successivamente vengono aggiornati i parametri. Concluso l'allenamento su ogni batch, la funzione procede a valutare la rete, confrontando l'attuale accuracy con quella salvata. Se il valore attuale è maggiore, allora viene posto a 0 il contatore della patience e viene salvato lo stato dei parametri.

Gli output della funzione sono pensati per permette di utilizzare la funzione durante le fasi di model selection e model assessment, questo può portare ad avere dei valori in output che non sono utilizzati dal chiamante, per esempio nella model selection l'unico output usato è l'accuracy massima.

2.5 Model Eval

La valutazione della rete è realizzata misurando la metrica di riferimento sul test set, in questo caso l'accuracy. Dopo aver trasformato il test set nel equivalente grafo non connesso, quest'ultimo viene passato alla rete per predire le probabilità di ogni componente del grafo di assumere valore 1. Le probabilità sono convertite nelle etichette binarie secondo la regola: 1 se la probabilità è maggiore di 0.5, 0 altrimenti. I valori ottenuti sono poi confrontati con quelli reali per calcolare l'accuracy.

2.6 Model Selection

La fase di model selection determina la migliore configurazione di iperparametri per il task, selezionando quella che massimizza la validation accuracy. La model selection è realizzata mediante la cross validation sui grafi contenuti nel training set. Per ciascuno dei folder viene definita una rete allenata con la funzione di *model training*. L'unico output della funzione usato è il valore massimo dell'accuracy, che viene mediato con quello di tutti i folder associati alla combinazione d'iperparametri. La combinazione restituita dalla funzione di Model Selection è quella con la media massima.

Inizialmente per ogni combinazione veniva definita un'unica rete e per ogni folder venivano resettati i parametri della rete. In termini di tempo questa soluzione è risultata simile al definire una nuova rete per ogni folder, pertanto ho deciso di semplificare il codice e costruire una nuova rete ogni volta.

2.7 Model Assessment

Durante la fase di model assessment viene allenata la rete finale, sulla migliore combinazione d'iperparametri. Come per la model selection l'allenamento avviene con la procedura di early stopping. Terminato l'allenamento la rete viene valutata sul test set, prima di farlo è necessario ripristinare la configurazione dei parametri che massimizza la validation accuracy. Per ripristinare i parametri passo alla funzione *load_state_dict* lo stato dei parametri restituito dalla funzione di training.

Applicare la procedura di early stopping non è stato semplice perché inizialmente allenavo la rete con il dataset ottenuto dall'unione del training set e del validation set, mentre adesso il validation set viene usato per decidere quando fermare l'allenamento.

2.8 Integrazione con CUDA

Un aspetto interessante di *Pytorch* è che permette di eseguire con semplicità il proprio codice su GPU, con il beneficio di ridurre il tempo per addestrare una rete neurale. In questo caso specifico il guadagno è stato superiore al 50%. Il tempo risparmiato è stato speso per validare un maggior numero di iperparametri durante la fase di model selection.

L'unica difficoltà rilevata per utilizzare la GPU è la riproducibilità dell'esecuzione, che richiede di scrivere qualche linea di codice in più quando si scelgono i seed per la generazione casuale. Visto che richiedere la riproducibilità comporta un tempo d'esecuzione maggiore, ho introdotto un parametro per abilitarla quando necessario.

È possibile eseguire il codice sulla GPU solo se CUDA è disponibile sulla propria macchina, altrimenti l'esecuzione avverrà sulla CPU. La scelta del dispositivo di esecuzione viene fatta a runtime all'inizio del codice.

3 Selection e Assessment

In questo capitolo vengono esaminate le scelte fatte per la fase di model Selection e i risultati prodotti da quella di Assessment. Analizzerò gli iperparametri e le prestazioni della rete, insieme alla capacità del modello di risolvere il task assegnato.

Per questa fase è stato fondamentale l'articolo di riferimento, l'ho usato per determinare il valore degli iperparametri che non ho validato, e per confrontare i risultati che ho ottenuto con la rete sviluppata.

3.1 Iperparametri

Scegliere gli iperparametri corretti è fondamentale per ottenere dei buoni risultati. Purtroppo per tempistiche e risorse hardware disponibili non è stato possibile validare la maggior parte degli iperparametri, di conseguenza anche decidere quali validare è stato importante.

Nella tabella 1 sono riportati gli iperparametri della rete suddivisi secondo l'ambito di riferimento, cioè la rete, i primi sette, e la fase di training, gli ultimi quattro. Inoltre è possibile osservare come soltanto quattro degli undici iperparametri siano stati validati, ciò ha comunque richiesto di considerare 24 configurazioni durante la fase di model selection.

Gli iperparametri non validati relativi alla rete sono stati determinati in accordo a quanto proposto nell'articolo, perché ciò ha permesso di mettere a confronto i miei risultati con loro. Invece per quanto riguarda gli iperparametri validati, questi sono la dimensione dell'hidden layer e decidere se allenare il valore di ϵ. Entrambi gli iperparametri sono validati nell'articolo di riferimento, tuttavia i valori scelti per l'hidden layer sono diversi, poiché in un caso ho lasciato la stessa dimensione del vettore in input. Decidere di allenare ϵ significa lasciare apprendere alla rete il peso da attribuire alle informazioni del vertice durante la fase di aggregazione. Viceversa nell'altro caso il valore del parametro è determinato

Hyperparameters	Value
Num. Layers	5
Num. Hidden Layers	1
Hidden Size	16, 37
Batch Normalization	True
MLP Classifier	True
Initial ϵ	0
Train ϵ	False, True
Num. Folders	6
Patience	10, 20, 40
Learning Rate	0.005, 0.01
Batch Size	113

Tabella 1: Iperparametri con relativi valori, suddivisi secondo l'ambito di riferimento. In grigio sono evidenziati quelli che ho validato

dal valore iniziale che è 0, cioè non c'è alcuna differenza tra le informazioni del vicinato e quelle che il vertice già possiede. In ambedue i casi la situazione iniziale prevede che il valore sia inizializzato a 0.

Gli iperparametri associati alla fase di training della rete sono fondamentali per determinare il tempo di esecuzione complessivo. a causa di ciò non è stato possibile utilizzare quelli proposti nell'articolo. Per ridurre il tempo di esecuzione sono stati usati meno folder per la cross validation, dal momento che ogni folder in più richiede di allenare una rete in più per ogni configurazione d'iperparametri.

Relativamente agli iperparametri validati in questa fase ci sono la patience e il learning rate. Il primo è fondamentale per evitare l'overfitting sul training set, in quanto determina quando fermare la fase di training. Dunque la scelta del valore della patience è fondamentale, perciò validarla è un metodo per ridurre la possibilità di overfitting e underfitting, quest'ultimo caso si riscontrerebbe qualora il valore scelto fosse troppo piccolo.

Infine Il learning rate viene validato perché, a differenza dell'articolo, non è implementata una decadenza dei pesi all'aggiornamento dei parametri, a causa di ciò si rende necessario determinare la velocità di apprendimento della rete.

3.2 Risultati

In quest'ultima sezione vengono esaminati i risultati ottenuti dal modello sul task assegnato, concentrando l'attenzione sulla migliore configurazione d'iperparametri e sulla rete finale. Visto che il tempo totale di esecuzione è stato di 3 ore, ho allegato al progetto un file di log contenente le informazioni prodotte dal modello durante l'intera esecuzione, così da non doverla ripetere. Sfortunatamente il tempo richiesto per eseguire il codice compromette un'analisi approfondita del modello, che invece richiederebbe di eseguire più volte il codice per eliminare esecuzioni i cui esiti sono troppo favorevoli e sfavorevoli. Questo tipo d'analisi non è dunque fattibile eseguendo il codice localmente sul computer.

Analizzando i valori dell'accuracy ottenuti con tutte le configurazioni d'iperparametri è possibile concludere che la patience è l'iperparametro che influenza maggiormente i risultati. Questa conclusione era prevedibile dal momento che introdurre la patience serve a ridurre il rischio di overfitting sul training set. Malauguratamente i valori per questo iperparametro influenzano il tempo di esecuzione, pertanto credo di averli selezionati troppo piccoli, visto che i migliori risultati sono stati ottenuti quando la patience era massima.

Gli altri iperparametri sembrano influenzare meno i risultati della rete. Sono tuttavia rimasto sorpreso per il mancato allenamento del parametro ϵ nella configurazione finale, anche se questa possibile situazione era già stata fatta notare nell'articolo di riferimento.

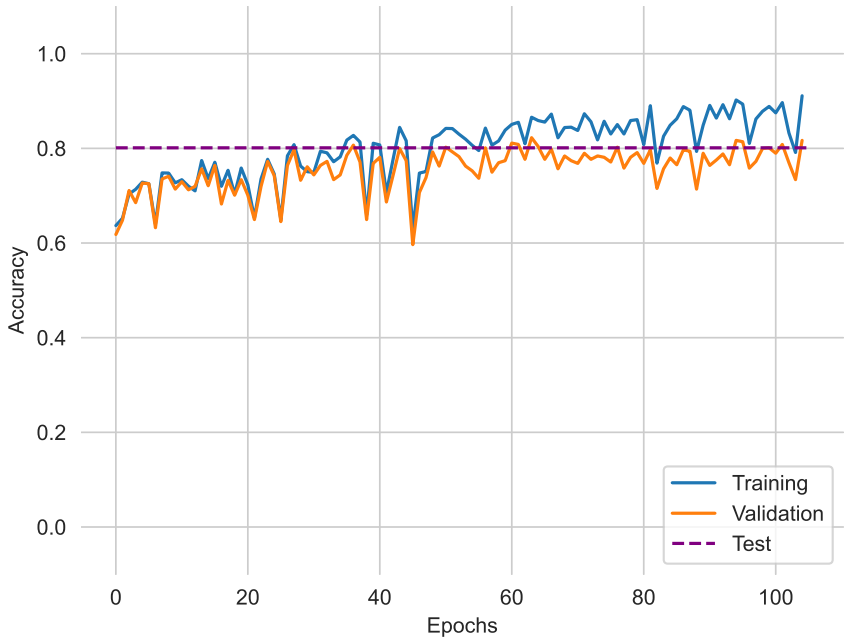

Figura 1: Evoluzione dell'accuracy sul Training set e sul Validation set della neural network finale

La figura 1 riporta l'evoluzione della training accuracy e della validation accuracy della rete finale, insieme al valore ottenuto sul test set, il quale è stato calcolato al termine della fase di training. Nonostante il grafico riporti l'evoluzione dell'accuracy fino al termine dell'allenamento, per il test set sono stati ripristinati i parametri al momento della massima validation accuracy, che è stato ottenuta intorno all'epoca 60.

La validation accuracy ottenuta per la rete finale è pari 0.8226 mentre l'accuracy sul test è uguale a 0.8011. Questi valori possono far concludere che non ci sia stato underfitting e overfitting, dato che le due accuracy differiscono di poco.

Infine nonostante i risultati siano piuttosto simili a quelli ottenuti nell'articolo credo che aumentando i valori della patience sia possibile ottenere risultati migliori. Pertanto avendo più risorse a disposizione potrei validare valori diversi della patience e eseguire più volte il codice per ottenere una migliore stima delle capacità del modello.