UNIVERSITÀ DEGLI STUDI DI CATANIA

Dipartimento di Matematica e Informatica

Corso di Laurea Triennale in Informatica

Riccardo Raciti

Protein Classification Via Graph Neural Networks

Relazione Progetto Finale

Relatore: Giovanni Micale

Anno Accademico 2021 - 2022

Abstract

Lo scopo di questo progetto è la realizzazione di un modello Neurale in grado di classificare le proteine basandosi sulle loro caratteristiche morfologiche.
Per il dataset è stato svolto un lavoro di mapping tra due siti, di cui uno mette a disposizione la classificazione delle proteine e un altro la composizione, quindi gli aminoacidi che le compongono e la posizione spaziale di essi. Sono stati provati differenti modelli per la classificazione modificando la struttura interna della rete neurale e i valori di training.
Come risultato finale è stato ideato un classificatore OneVsAll poiché all'aumentare del numero di classi presenti nel dataset l'accuratezza diminuiva drasticamente.

Indice

Capitolo 1

Introduzione

L'obiettivo di questo progetto è quello di creare un classificatore di proteine, che riesca a discriminare una proteina basandosi sulla morfologia della sua struttura.

1.1 Le Proteine

Le proteine sono costituite da diversi aminoacidi collegati tra loro. Vi sono venti tipi diversi di aminoacidi comunemente presenti nelle piante e negli animali. Una proteina tipica è composta da 300 o più aminoacidi, e il numero e la sequenza specifica di aminoacidi sono unici per ciascuna proteina.
Come per l'alfabeto, le 'lettere' dell'aminoacido possono essere organizzate in milioni di modi diversi per creare delle 'parole' e un'intera 'lingua' proteica. A seconda del numero e della sequenza degli aminoacidi, la proteina risultante si ripiegherà in una forma specifica. Questa forma è molto importante perché determinerà la funzione della proteina. Esse possono essere immaginate come un grafo tridimensionale.

1.2 I Grafi

I grafi sono modelli matematici che permettono di codificare le interazioni tra le componenti di un sistema complesso. La scienza che studia le reti prende il nome di network science.
Formalmente, un grafo è una coppia $G = (V, E)$ dove V è l'insieme di vertici ed E è l'insieme degli archi, dove un arco è una coppia di vertici (u, v). I vertici rappresentano le componenti del sistema, mentre gli archi rappresentano le interazione tra le componenti. Se $(a, b) \in E$ allora diremo che b è adiacente ad a. Nel caso delle proteine si può immaginare di avere a che fare

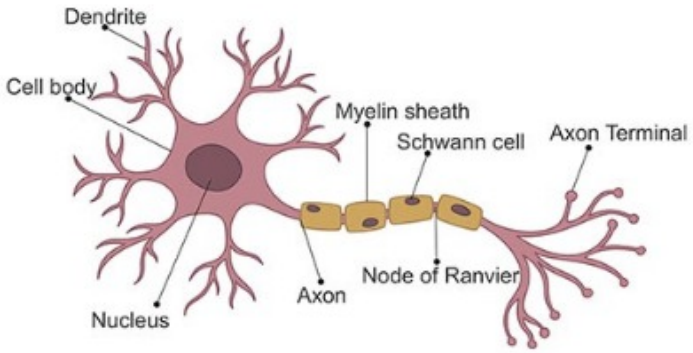
con grafi indiretti o non orientati, diremo che un grafo è non orientato se:

$$\forall(a,b) \in E \Leftrightarrow (b,a) \in E$$

In altri termini in un grafo non orientato ogni interazione tra due nodi è reciproca.

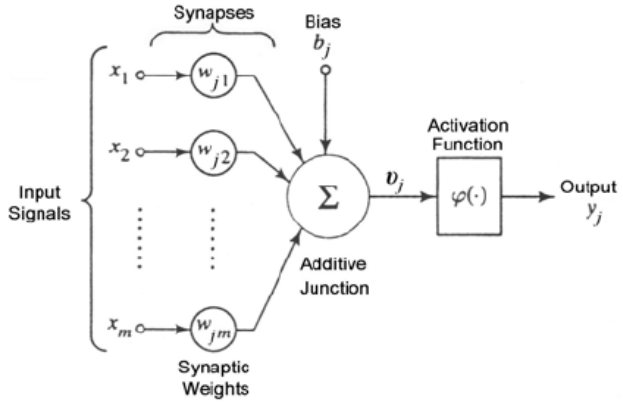
1.3 Reti Neurali

Per la risoluzione di questo problema si è optato per l'utilizzo delle reti neurali le quali si ispirano al funzionamento dei neuroni umani.
I neuroni sono le più importanti cellule del sistema nervoso. Le connessioni sinaptiche, o sinapsi, agiscono come punti di collegamento per il passaggio dell'informazione tra neuroni. I dendriti sono fibre minori che si ramificano a partire dal corpo cellulare del neurone, soma. Attraverso le sinapsi, i dendriti raccolgono input da neuroni afferenti e li propagano verso il soma. L'assone è la fibra principale che parte dal soma e si allontana da esso per portare ad altri neuroni l'output.



Il passaggio delle informazioni attraverso le sinapsi avviene attraverso processi elettro-chimici: il neurone presinaptico libera delle sostanze, i neuro neurotrasmettitori, che attraversano lo sapzio sinaptico e sono captati da appositi recettori, detti canali ionici, sulla membrana del neurone postsinaptico. L'ingresso di ioni attraverso i canali ionici determina la formazione di una differenza di potenziale tra il corpo del neurone postsinaptico e l'esterno. Quando questo potenziale supera una certa soglia, detta di attivazione, si produce uno spike o impulso: il neurone propaga un breve segnale elettrico detto potenziale d'azione lungo il proprio assone: questo potenziale determina il rilascio di neurotrasmettitori dalle sinapsi dell'assone.

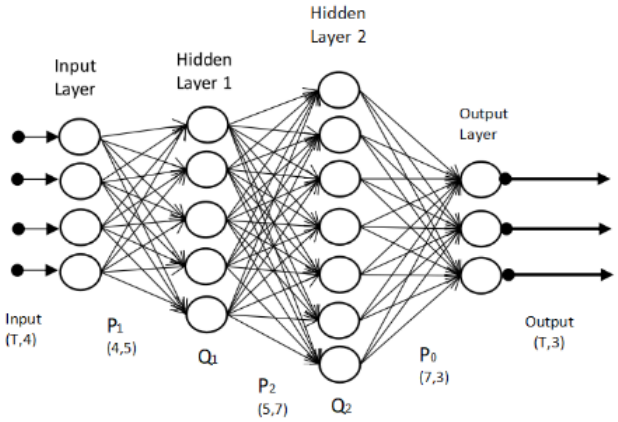
Il reweighting delle sinapsi, ovvero la modifica della loro efficacia di trasmissione, è direttamente collegata ai processi di apprendimento e memoria in accordo con la regola di Hebb.
Hebbian Rule: se due neuroni, tra loro connessi da una o più sinapsi, sono ripetutamente attivati simultaneamente allora le sinapsi che li connettono sono rinforzate.
Il cervello umano contiene circa 100 miliardi di neuroni, ciascuno dei quali connesso con circa altri 1000 neuroni (sinapsi). La corteccia celebrale (sede delle funzioni nobili del cervello umano) è uno strato laminare continuo di 2-4 mm, una sorta di lenzuolo che avvolge il nostro cervello formando numerose circonvoluzioni per acquisire maggiore superficie. Sebbene i neuroni siano disposti in modo ordinato in livelli consecutivi, l'intreccio di dendriti e assoni ricorda una foresta fitta ed impenetrabile.
Il primo modello di neurone artificiale fu progettato da McCulloch e Pitts: gli input e gli output erano binari ed erano in grado di eseguire delle computazioni logiche.
Un neurone artificiale moderno prende in ingresso n input $(x_1, \ldots, x_\mathrm{n})$, pesati rispettivamente con n pesi $(w_1, \ldots, w_\mathrm{n})$ che rappresentano l'efficacia delle connessioni sinaptiche dei dendriti. Tali valori varieranno durante il processo di apprendimento. Esiste un ulteriore peso, detto costante di *bias*, che si considera collegato ad un input fittizio con valore costante 1, questo peso è utile per tarare il punto di lavoro ottimale del neurone.



Il neurone somma i prodotti tra gli input ed i corrispettivi pesi, compresa la costante di *bias*, e produce un valore. Dopodiché, sulla base di una funzione di attivazione a cui viene passato il valore, produce un valore z di output.

1.3.1 Layer

Le reti neurali sono composte da gruppi di neuroni artificiali organizzati in livelli o *layer*. Tipicamente sono presenti un layer di input, un layer di output e uno o più layer intermedi o nascosti, tipicamente chiamati *hiddenlayer*. Ogni layer contiene uno o più neuroni.



Il layer di input è costituito da un vettore di n valori $(x_1, \ldots, x_n)$.
Gli hidden layer sono costituiti da uno o più nodi che prendono in input uno o più valori provenienti dal layer precedente.
Ogni nodo dell'hidden layer produrrà un output che verrà passato ad uno o più nodi del layer successivo. Il layer di output è costituito da uno o più nodi che restituiscono in output un valore. Il termine deep neural network indica una rete formata da molti layer nascosti.

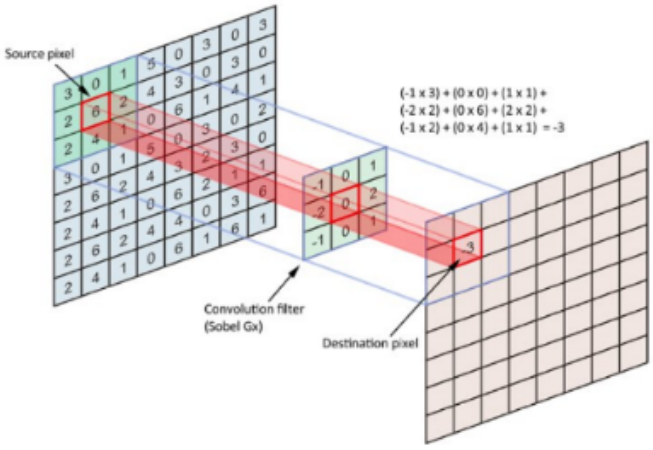
1.3.2 Tensori

Nel contesto delle reti neurali, un tensore è un array *n-dimensionale*, ovvero una generalizzazione di un vettore o di una matrice. Nel modello più generale di rete neurale sia l'input che l'output di un nodo della rete può essere un tensore. I nodi di un layer possono essere organizzati e disposti a formare una matrice, come nelle convolutional networks, o un tensore. Nel primo caso ritroviamo il tensore al livello dei dati, mentre nel secondo lo ritroviamo come disposizione dei nodi della rete.

1.3.3 Connessione tra layer

Una rete neurale in cui ogni nodo di un certo layer riceve tutti gli output del layer precedente è detta *densa*. In questo caso si parla di layer totalmente connessi.
Altre tipologie di connessioni tra layer possibili:

- *Connessione random*: fissato m, ogni nodo riceve output solamente da m nodi, generalmente casuali, del precedente layer.
- *Connessione pooled*: i nodi di un layer sono partizionati in k cluster. Il layer successivo, chiamato pooled layer, sarà formato da k nodi, uno per ogni cluster. Il nodo associato al cluster C_i riceverà tutti e soli gli output dei nodi del layer precedente appartenenti al cluster C_i.
- *Connessione convolutional*: i nodi di ogni layer sono visti come se fossero disposti su una griglia. Il nodo di coordinate (i, j) riceve tutti e soli gli input dei nodi del layer precedente che si trovano in una regione piccola della griglia intorno al punto (i, j).



Connessione Convoluzionale

1.3.4 Explanable AI

La rete neurale si presenta come un modello black-box: un osservatore esterno vede l'output prodotto dal modello a partire da un input, ma il modello non è in grado di giustificare il risultato ottenuto, ovvero non è in grado di spiegare il procedimento logico per cui si arriva a produrre quel risultato. Il termine Explanable AI indica una serie di tecniche a supporto di modelli di intelligenza artificiale per far sì che un risultato prodotto da tali modelli possa essere compreso da un essere umano.

1.4 Funzioni di attivazione

1.4.1 Definizione formale

Sia $\bar{x} = (x_1, ..., x_n)$ il vettore dei valori ricevuti da un nodo, $\bar{w} = (w_1, ..., w_n)$ il vettore dei pesi e b la costante di bias. La funzione di attivazione di un nodo i è la funzione F che determina la risposta $F(z)$ prodotta a partire da:

$$z = \bar{w}\bar{x} + b$$

Tutti i nodi di un layer hanno la stessa funzione di attivazione.

1.4.2 Proprietà desiderate

La scelta della funzione di attivazione si lega al metodo scelto per ottimizzare i pesi della rete basato sulla minimizzazione della funzione loss. Il metodo di minimizzazione più popolare è quello della discesa del gradiente (gradient descent). Affinché il gradient descend lavori al meglio, la funzione di attivazione deve avere delle proprietà desiderate:

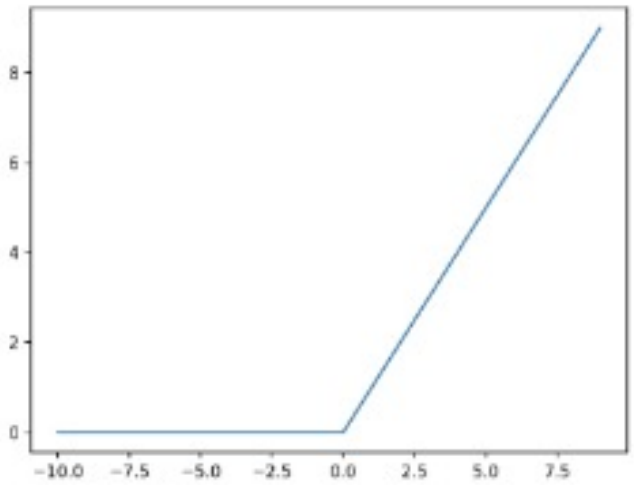
- La funzione deve essere *continua* e *differenziabile* in ogni punto, o quasi.
- La derivata della funzione non deve *saturare*, ovvero tendere a zero nel proprio dominio: questo potrebbe portare ad uno stallo nel processo di ricerca dei pesi ottimali.
- La derivata della funzione non deve *esplodere*, ovvero tendere all'infinito nel proprio dominio: questo potrebbe portare instabilità numerica nel processo di ricerca dei pesi ottimali.

1.4.3 Rectified Linear Unit (ReLU)

La funzione ReLU prende spunto dai raddrizzatori a singola semionda (half-wave rectifiers) usati in elettronica per trasformare un segnale alternato in un segnale unidirezionale, sempre positivo o sempre negativo, facendo passare solo semionde positive. È formalmente definita come segue:

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x \text{ if } x \geq 0 \\ 0 \text{ if } x < 0 \end{cases}$$

Rappresentabile con il seguente grafico:



La funzione ReLU non satura mai per valori di x positivi. Nella pratica, le reti neurali che utilizzano ReLU offrono uno speed-up significativo nella fase di training rispetto alle funzioni sigmoidee. Sia il calcolo della funzione che della sua derivata sono molto semplici e veloci da effettuare poiché non è richiesta l'esponenziazione. ReLU presenta problemi di saturazione della sua derivata quando x è negativo.

1.5 Funzioni Loss

La loss function, o funzione *costo*, è quella funzione utilizzata nel processo di learning dei pesi del modello. Essa quantifica la differenza tra le predizioni di un modello e i valori corretti di output osservati nel training set, quindi l'errore medio di predizione tra valori predetti e valori reali. I pesi ottimali sono quelli che minimizzano la funzione loss.
Distinguiamo due tipologie di funzioni loss in base al problema che affronta la rete neurale:

- Regression loss nei problemi di regressione, dà in output uno scalare o un vettore di valori reali.
- Classification loss nei problemi di classificazione, dà in output una distribuzione di probabilità, con valori che indicano la probabilità di appartenenza ad una classe.

1.5.1 Classification loss

Poniamoci in un generale problema di classificazione con k classi $C_1, ..., C_k$. Supponiamo di avere un training set $T = (\bar{x}_1, \bar{p}_1), ..., (\bar{x}_n, \bar{p}_n)$ dove $\bar{x}$ è il vettore degli input e $\bar{p} = (p_1, ..., p_k)$ è una distribuzione di probabilità. La componente p_i del vettore $\bar{p}$ indica la probabilità che $\bar{x}$ appartenga alla classe C_i.
Supponiamo che la rete neurale sia progettata, ad esempio attraverso una softmax, per produrre in output una distribuzione di probabilità $\bar{q} = (q_1, ..., q_n)$, dove q_i indica la probabilità predetta dal modello che $\bar{x}$ sia di classe C_i. Le funzioni di classification loss quantificano la distanza tra le distribuzioni di probabilità $\bar{p}$ e $\bar{q}$.

1.5.2 Entropia

Supponiamo di avere un alfabeto di n simboli. Si vuole trasmettere un messaggio utilizzando questi simboli attraverso un canale di informazione. Sia $\bar{p} = (p_1, ..., p_n)$ una distribuzione di probabilità.
Supponiamo che in ogni punto del messaggio la probabilità di osservare l'i-esimo simbolo sia p_i.
Il teorema di Shannon afferma che, in un sistema di codifica ottimale, il numero medio di bit per simbolo necessari per codificare il messaggio è dato dall'entropia $H(\bar{p})$:

$$H(\bar{p}) = -\sum_{i=1}^{n} p_i \log_2 p_i$$

Il termine $-\log p_i$ indica il numero di bit necessari a rappresentare l'i-esimo simbolo usando lo schema di codifica ottimale. Qualunque altro schema utilizza in media più bit, dunque è sub-ottimale.

1.5.3 Entropia incrociata

Nella teoria dell'informazione, l'entropia incrociata, o cross-entropy, tra due distribuzioni di probabilità p e q, relative allo stesso insieme di eventi, misura il numero medio di bit necessari ad identificare un evento estratto dall'insieme nel caso sia utilizzato uno schema alternativo q anziché la vera distribuzione p.
Si supponga di cambiare lo schema di codifica, usando una diversa distribuzione di probabilità $\bar{q} = (q_1, ..., q_n)$. Con questo nuovo schema occorrono $-\log q_i$ bit per rappresentare l'i-esimo simbolo. Quanti bit per simbolo occorrono in media se usiamo questo schema di codifica sub-ottimale? Dalla definizione di entropia incrociata possiamo calcolarlo come segue:

$$C(\bar{p}||\bar{q}) = -\sum_{i=1}^{n} p_i \log_2 q_i$$

1.6 Training di una rete neurale

Il processo di training consiste nel trovare i parametri della rete, detti pesi, che minimizzano l'average loss, quantificato mediante la funzione di loss scelta, su un training set di dati. L'obiettivo finale è costruire un modello che garantisca una loss media bassa su tutti i possibili input. Dato l'elevato numero di parametri di una rete neurale, specialmente se deep, il rischio di overfitting è alto. Concentriamoci inizialmente sulla minimizzazione della funzione di loss sul training set.

1.6.1 Derivate parziali e gradiente

È possibile estendere l'idea della derivata alle funzioni multivariate. Sia $y = f(x_1, x_2, \ldots, x_n)$ una funzione ad n variabili. La derivata parziale di y rispetto alla componente i-esima è:

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_n)}{h}$$

Per calcolare la derivata parziale di y rispetto alla componente i-esima è sufficiente trattare tutte le variabili meno che la i-esima come costanti e

calcolare la derivata di y. Le seguenti notazioni sono equivalenti:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = f_{x_i} = D_i f = D_{x_i} f$$

Raggruppando le derivate parziali di una funzione multivariata rispetto ad ognuna delle sue componenti otteniamo il vettore gradiente della funzione. Supponiamo che l'input $\bar{x}$ della funzione $f : R^n \to R$ sia un vettore n-dimensionale $x = [x_1, \ldots, x_n]^T$ e che l'output sia invece uno scalare. Il gradiente della funzione $f(x)$ rispetto ad x è un vettore di n derivate parziali:

$$\nabla_{\bar{x}} f(\bar{x}) = \left(\frac{\partial f(\bar{x})}{\partial x_1}, \frac{\partial f(\bar{x})}{\partial x_2}, ..., \frac{\partial f(\bar{x})}{\partial x_n} \right)$$

Sia x un vettore n-dimensionale, le seguenti regole sono spesso utilizzate nella differenziazione di funzioni multivariate:

- $\forall A \in R^{m \times n}, \nabla_{\bar{x}} Ax = A^T$.
- $\forall A \in R^{n \times m}, \nabla_{\bar{x}} x^T A = A$.
- $\forall A \in R^{n \times n}, \nabla_{\bar{x}} x^T Ax = (A + A^T)x$.
- $\nabla_{\bar{x}} ||\bar{x}||^2 = \nabla_{\bar{x}} x^T x = 2x$.

Similmente, per ogni matrice X, abbiamo che $\nabla_X ||X||_F^2 = 2X$.

1.6.2 Matrice jacobiana

Sia $\bar{x} = (x_1, ..., x_n)$ un vettore di n valori reali. Sia $f : R^n \to R^m$ ed $\bar{y} = f(\bar{x})$. La matrice jacobiana di $\bar{y}$ rispetto ad $\bar{x}$ è la matrice formata dalle derivate parziali prime di ciascuna componente di $\bar{y}$ rispetto a ciascuna componente di $\bar{x}$:

$$Jf(\bar{x}) = J(\bar{y}) = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \ldots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \\ \frac{\partial y_1}{\partial x_n} & \ldots & \frac{\partial y_m}{\partial x_n} \end{bmatrix}$$

1.6.3 Metodo di discesa del gradiente

Il metodo di discesa del gradiente, *gradientdescent*, è una tecnica atta ad individuare i punti di minimo, o di massimo, di una funzione di più variabili. Nel contesto delle reti neurali la funzione da minimizzare è la funzione loss calcolata sui parametri correnti del modello.
Partendo da un valore iniziale assunto dai parametri della funzione, il metodo iterativamente cerca la direzione di massima discesa del valore della funzione e aggiorna i valori dei parametri seguendo tale direzione. La direzione di

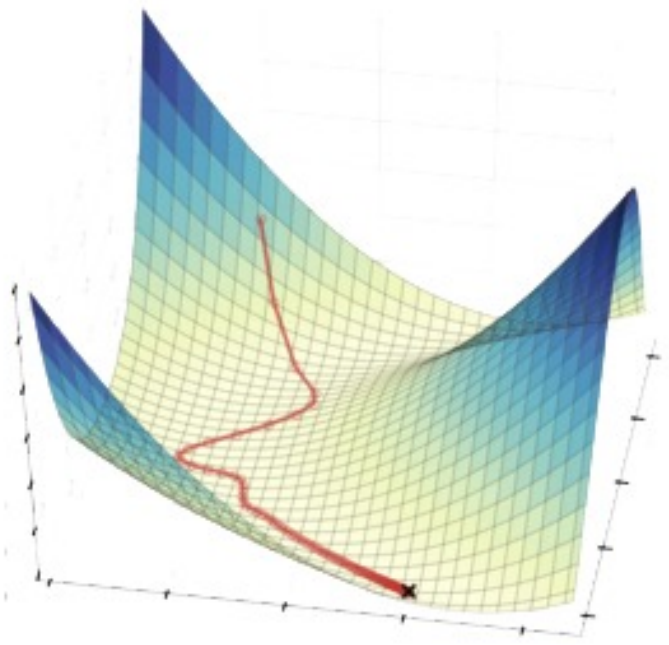
massima discesa è quella opposta al gradiente, o alla matrice jacobiana nel caso in cui il codominio della funzione sia multidimensionale.

1.6.4 Schema del metodo

Sia $f : R^n \to R$ la funzione loss da minimizzare. Indichiamo con W_t la matrice di parametri della rete neurale calcolato dall'algoritmo all'iterazione t.
La procedura generale dell'algoritmo di discesa del gradiente è la seguente:

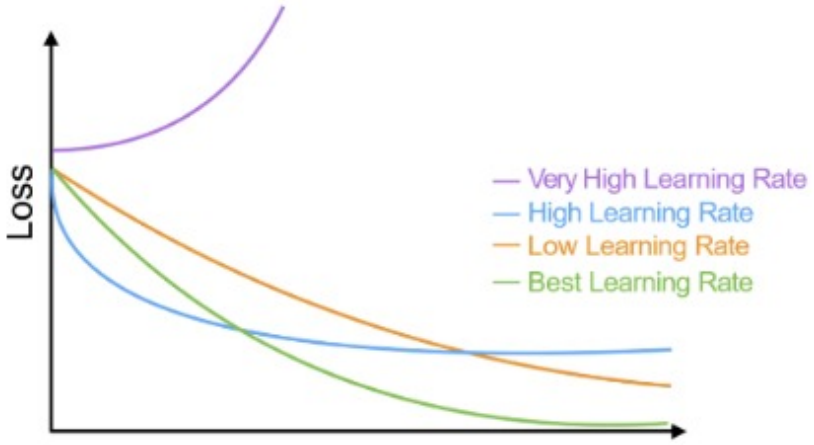
- Per $t = 0$ inizializzare la matrice dei pesi W_0 con valori casuali.
- Calcolare la funzione loss con i parametri W_t ed il gradiente $\nabla_{W_t} f$.
- Aggiornare la matrice dei pesi $W_{t+1} = W_t - \eta \nabla_{W_t} f$.
- Passare alla iterazione successiva $t = t + 1$ e ripartire dal secondo step.

Il metodo viene iterato sino a quando i valori del vettore non cambiano in maniera significativa. Analogo metodo per la matrice Jacobiana. Essendo basato su una scelta greedy, non garantisce l'individuazione di minimi assoluti, per cui può convergere ad un minimo locale. Il vettore $\bar{x}$ ottenuto dopo la convergenza dell'algoritmo contiene i parametri della rete che minimizzano la funzione loss.



1.7 Learning rate

Il parametro η (eta) è detto *learning rate* e determina la velocità con cui si desidera che il metodo converga al valore ottimale. Valori troppo bassi implicano che la convergenza richieda molte iterazioni. Dall'altra parte, valori molto alti possono causare grandi oscillazioni nei valori dei parametri della rete, impedendo di arrivare a convergenza. Un metodo per trovare il learning rate ottimale consiste nel partire da un valore alto di η e ad ogni passo moltiplicare η per un fattore β (con $0 < \beta < 1$) fino ad ottenere un valore di η che porti a convergenza.



1.7.1 Inizializzazione dei pesi della rete

Per applicare il metodo di discesa del gradiente occorre partire da un valore iniziale della funzione loss, poiché quest'ultima deve essere calcolata su una rete neurale già definita con dei pesi assegnati. Si pone quindi il problema di inizializzare i pesi della rete. Intuitivamente, se vogliamo che i nodi di un layer si comportino in maniera diversa, e riconoscano feature diverse sugli input ricevuti, occorre scegliere dei pesi diversi.
Vi sono due approcci:

- Scegliere casualmente pesi in $[-1, 1]$ seguendo una distribuzione uniforme.
- Scegliere casualmente secondo una distribuzione normale.

1.7.2 Stochastic gradient descent

Il metodo *stochastic gradient descent* è una variante della discesa del gradiente in cui, ad ogni iterazione del metodo, si lavora su un piccolo campione di dati del training set selezionato in maniera casuale. Tale variante risulta più veloce del gradient descent originale quando il training set è troppo grande.

1.7.3 Calcolo del gradiente

Una volta calcolata la funzione loss su un vettore di output, occorre calcolare il gradiente della funzione loss rispetto ai pesi della rete in quel determinato momento. Un algoritmo efficiente per il calcolo del gradiente è l'algoritmo di backpropagation, che sfrutta il concetto di grafo computazionale.

1.8 Backpropagation

1.8.1 Forward propagation

Il termine $forward\ propagation$, o forward pass, si riferisce al calcolo e all'archiviazione ordinata di variabili intermedie della rete neurale, dall'input layer all'output layer.
Per semplicità assumiamo che l'input d'esempio sia $x \in R^d$ e che vi sia un solo layer nascosto nella rete, che non includa alcun termine di bias. Sia z una variabile intermedia:

$$z = W^{(1)}x$$

Dove $W^{(1)} \in R^{h \times d}$ è la matrice dei pesi dell'unico hidden layer. Diamo in input tale variabile z alla funzione di attivazione ϕ e otteniamo un vettore di attivazione h di lunghezza h:

$$h = \phi(z)$$

Anche la variabile h è una variabile intermedia. Assumendo che l'output layer possegga una matrice di pesi $W^{(2)} \in R^{q \times h}$, otteniamo il risultato dell'output layer e poniamolo in una variabile temporanea o di lunghezza q:

$$o = W^{(2)}h$$

Assumendo che la funzione loss sia l e che la classe dell'esempio x sia y, possiamo calcolare la loss per la predizione di x come segue:

$$L = l(o, y)$$

Dalla definizione della regolarizzazione L_2, dato un parametro λ, il termine di regolarizzazione è:

$$s = \frac{\lambda}{2}\left(||W^{(1)}||_F^2 + ||W^{(2)}||_F^2\right)$$

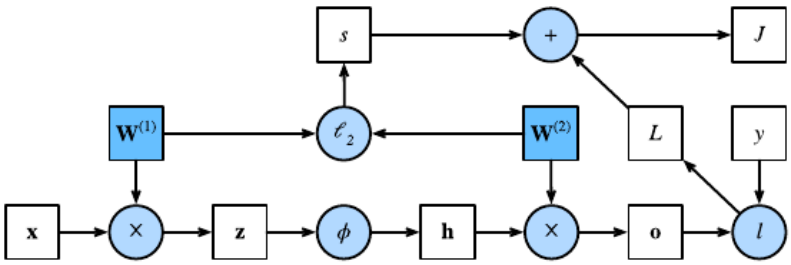
Dove la norma di Frobenius (o norma matriciale) è semplicemente la norma L_2 applicata dopo aver concatenato la matrice in un singolo vettore. Otteniamo quindi l'ultimo termine J, ovvero la loss regolarizzata:

$$J = L + s$$

Ci riferiremo a J con il nome di objective function o funzione loss regolarizzata.

1.8.2 Grafo computazionale

Un grafo computazionale è un grafo aciclico diretto (DAG) che permette di visualizzare il flusso di dati di una rete neurale. Ogni nodo può avere due forme: un nodo quadrato indica un valore, tensore di dimensione arbitraria, mentre un nodo circolare indica una operazione. La direzione indica che il nodo mittente è operando del nodo destinatario, o che il nodo destinatario è output del nodo mittente. Visualizziamo il grafo computazionale della rete neurale descritta dalla forward propagation:



1.8.3 Chain rule

La regola della catena "*chain rule*" è una regola di derivazione che permette di calcolare la derivata di una funzione composta da due funzioni derivabili. Supponiamo che le funzioni $y = f(u)$ ed $u = g(x)$ siano entrambe differenziabili, la regola della catena enuncia che:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

Nel caso più generale di funzioni multivariate, supponiamo che la funzione differenziabile y abbia $u_1, \ldots, u_m$ variabili, e che ogni funzione differenziabile u_i abbia $x_1, ..., x_n$ variabili. La regola della catena enuncia che per calcolare la derivata parziale di y rispetto ad x_i è sufficiente calcolare:

$$\frac{dy}{dx_i} = \frac{dy}{du_1}\frac{du_1}{dx_1} + \frac{dy}{du_2}\frac{du_2}{dx_1} + \cdots + \frac{dy}{du_m}\frac{du_m}{dx_1}$$

per $i = 1, 2, \ldots, n$.

1.8.4 Algoritmo di backpropagation

L'algoritmo di backpropagation è utilizzato nel calcolo del gradiente della funzione loss rispetto ai parametri, pesi, della rete neurale. In breve, il metodo percorre la rete neurale in verso opposto, dall'output layer all'input layer, e calcola il gradiente sfruttando la regola della catena.
L'algoritmo conserva le derivate parziali intermedie ad ogni iterazione e le ri-utilizza per calcolare altre derivate parziali andando indietro nel grafo computazionale. Ipotizziamo due funzioni $Y = f(x)$ e $Z = g(Y)$, in cui X, Y, Z sono tensori di dimensione arbitraria. Utilizzando la regola della catena, possiamo calcolare la derivata di Z rispetto ad X come segue:

$$\frac{\partial Z}{\partial X} = \text{prod}\left(\frac{\partial Z}{\partial Y}\frac{\partial Y}{\partial X}\right)$$

Dove l'operatore prod generalizza la chain rule in base alla dimensione dei tensori. Riprendiamo l'esempio visto nella forward propagation di rete neurale ad un solo hidden layer, in cui $W^{(1)}$ è la matrice dei pesi dell'hidden layer, mentre $W^{(2)}$ è la matrice dei pesi dell'output layer. Sia J la funzione costo regolarizzata, l'obiettivo della backpropagation è quello di calcolare i gradienti $\nabla_{W^1} J$ e $\nabla_{W^2} J$. Per ottenere ciò, calcoliamo a turno i gradienti rispetto ad ogni variabile intermedia utilizzando la chain rule.
Partendo in ordine inverso, il primo step consiste nel calcolare il gradiente

della funzione costo regolarizzata rispetto al termine di loss L e rispetto al termine di regolarizzazione s.

$$J = L + s \Longrightarrow \frac{\partial J}{\partial L} = 1 \text{ and } \frac{\partial J}{\partial s} = 1$$

Calcoliamo il gradiente della funzione loss regolarizzata J rispetto al risultato dell'output layer o, seconda la regola della catena:

$$\frac{\partial J}{\partial o} = \text{ prod}\left(\frac{\partial J}{\partial L}\frac{\partial L}{\partial o}\right) = \frac{\partial L}{\partial o} \in R^q$$

Calcoliamo il gradiente del termine di regolarizzazione s rispetto ad entrambe le matrici di parametri $W^{(1)}$ e $W^{(2)}$, ricordando di aver utilizzato la regolarizzazione L_2:

$$\frac{\partial s}{\partial W^{(1)}} = \lambda W^{(1)} \text{ and } \frac{\partial s}{\partial W^{(2)}} = \lambda W^{(2)}$$

È possibile adesso calcolare il gradiente (in questo caso una matrice Jacobiana) della funzione loss regolarizzata J rispetto ai parametri dell'output layer $W^{(2)}$ utilizzando la regola della catena:

$$\frac{\partial J}{\partial W^{(2)}} = \text{prod}\left(\frac{\partial J}{\partial o}\frac{\partial o}{\partial W^{(2)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}\frac{\partial s}{\partial W^{(2)}}\right) = \left(\frac{\partial J}{\partial o}h^T + \lambda W^{(2)}\right) \in R^{q\times h}$$

Per ottenere il gradiente di J rispetto ai parametri dell'hidden layer $W^{(1)}$ è necessario continuare la backpropagation dall'output layer all'hidden layer. Il gradiente della funzione loss regolarizzata J rispetto all'output dell'hidden layer h è dato da:

$$\frac{\partial J}{\partial h} = \text{prod}\left(\frac{\partial J}{\partial o}\frac{\partial o}{\partial h}\right) = {W^{(2)}}^T\frac{\partial J}{\partial o} \in R^h$$

Dato che la funzione di attivazione ϕ viene applicata ad ogni elemento di z, calcolare il gradiente di J rispetto a z richiede l'utilizzo dell'operatore di moltiplicazione element wise denotata dal simbolo $\odot$:

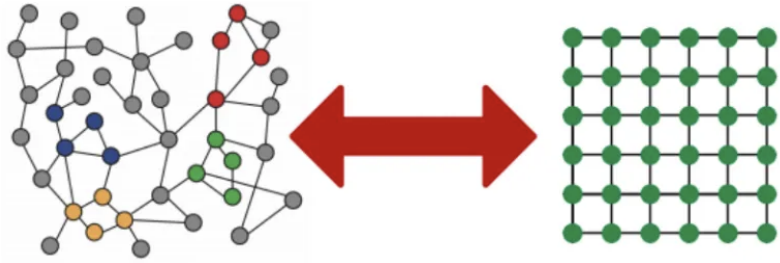
$$\frac{\partial J}{\partial z} = \text{prod}\left(\frac{\partial J}{\partial h}\frac{\partial h}{\partial z}\right) = \frac{\partial J}{\partial h} \odot \phi'(z)$$

In conclusione, è possibile ottenere il gradiente (anche in questo caso una matrice Jacobiana) della funzione J rispetto ai parametri dell'hidden layer $W^{(1)}$ utilizzando la regola della catena:

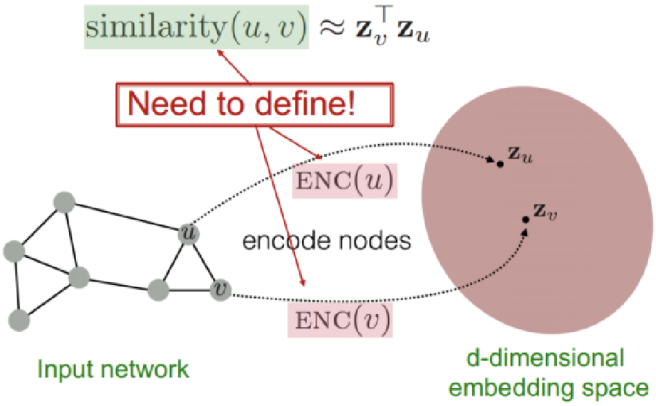
$$\frac{\partial J}{\partial W^{(1)}} = \text{prod}\left(\frac{\partial J}{\partial z}\frac{\partial z}{\partial W^{(1)}}\right) + \text{prod}\left(\frac{\partial J}{\partial s}\frac{\partial s}{\partial W^{(1)}}\right) = \left(\frac{\partial J}{\partial z}x^T + \lambda W^{(1)}\right) \in R^{h\times d}$$

1.9 GNN

In questo caso i dati in input saranno dei grafi per cui il tensore in input avrà una disposizione a forma di matrice.



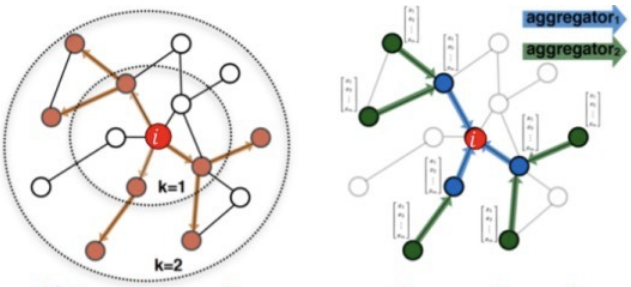
Le reti neurali che elaborano grafi sono chiamate Graph Neural Networks o GNNs, esse sono una classe di metodi di deep learning progettati per eseguire inferenze su dati descritti dai grafi.
Le GNN sono reti neurali che possono essere applicate direttamente ai grafi e forniscono un modo semplice per eseguire attività di previsione a livello di nodo, archi e sui grafi stessi.
Le GNN possono fare ciò che le reti neurali convoluzionali (CNN) non possono fare.
Nella teoria dei grafi, vi è il concetto di Node Embedding, il quale esprime il concetto di mappare i nodi da uno spazio a n dimensioni a uno spazio dimensionalmente più piccolo, solitamente i nodi vengono mappati in uno spazio bidimensionale, in modo che nodi simili nel grafo siano incorporati l'uno vicino all'altro.
L' obiettivo è mappare i nodi in modo che la similarità nello spazio di embedding si avvicini alla somiglianza nella rete.
Si definiscano u e v come due nodi in un grafo.
X_u e X_v sono due vettori contenenti le feature, o caratteristiche, dei nodi.
Definiremo ora la funzione encoder $Enc(u)$ e $Enc(v)$, che convertono i vettori delle caratteristiche in Z_u e Z_v.
La funzione di similarità utilizzata in questo progetto è Contact Map.



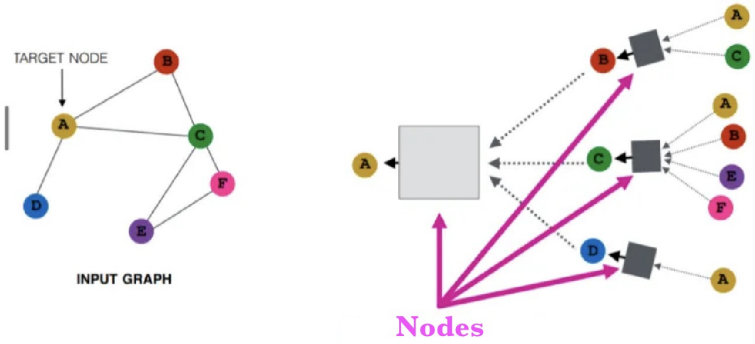
La funzione dell'encoder dev'essere in grado di estrapolare informazioni:

- Locali → relative ai singoli nodi.
- Aggregate → relative ai vicini dei nodi.
- Più livelli → relative alla struttura della rete.

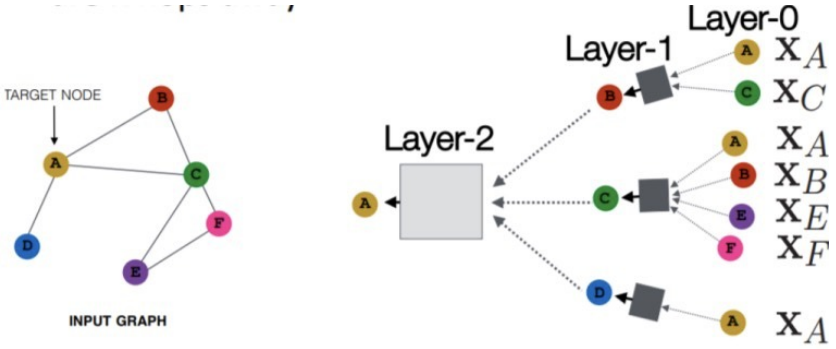
Le informazioni sulla località possono essere ottenute utilizzando un grafo computazionale. Come mostrato nel grafo sottostante, i è il nodo rosso in cui vediamo come questo nodo è connesso ai suoi vicini e ai vicini dei vicini. Vedremo tutte le possibili connessioni e formeremo un grafo di calcolo.
In questo modo, si acquisisce la struttura del grafo e allo stesso tempo si estrapolano le informazioni sulla struttura.



Dove per K si intende la distanza da il nodo i ai nodi vicini, anche chiamato cammino di lunghezza 1.
Una volta ottenute le informazioni sulla località dei nodi del grafo computazionale, si iniziano ad aggregare tali informazioni. Questo è fondamentalmente realizzato mediante le reti neurali.



I riquadri grigi rappresentano i nodi della rete neurale.
Le reti neurali richiedono che le aggregazioni siano invarianti per ordine, come somma, media, massimo, perché sono funzioni invarianti per permutazione. Questa proprietà consente di eseguire le aggregazioni.
Il forward propagation, o forward pass, nelle GNN si riferisce al calcolo e all'archiviazione ordinata di variabili intermedie della rete neurale, dall'input layer all'output layer.



Ogni nodo ha un vettore di caratteristiche contenente le feature del nodo.
Ad esempio, (X_A) è un vettore di feature del nodo A .
Gli input sono quei vettori di caratteristiche, posti a destra dell'immagine.
Nel layer-0, i vettori di caratteristiche X_A e X_C saranno gli input del primo nodo, il nodo li aggregherà e li passerà al layer successivo.
Tale introduzione alle GNN è un estratto di [1].

Capitolo 2

Metodi per la risoluzione del problema

Come scritto in precedenza le proteine possono essere rappresentate attraverso dei grafi. Le proteine in natura hanno una composizione tridimensionale ma le GNN elaborano solo grafi bidimensionali, per questo motivo è stata necessaria una fase di preprocessing per adattare le strutture 3D di ogni grafo in strutture bidimensionali; per fare ciò è stata utilizzata come metrica di distanza la contract map, che si basa sulla vicinanza spaziale dei punti.
Scelta una soglia minima x diremo che il punto P è collegato al punto W se la loro distanza è minore di x, considerando tutte e tre le dimensionalità dello spazio di partenza.
Per effettuare questa operazione è stata utilizzata la libreria <bio3d> [2], la quale mette a disposizione una funzione chiamata "cmap".

```
contact.map <- cmap(pdb, dcut=10, scut=0, inds=inds, mask.lower=F)
```

Le proteine sono rappresentate da differenti sequenze di aminoacidi che variano in base al percorso che si prende in esame, tale percorso prende il nome di catena. Per questo progetto si è scelto di utilizzare ed estrapolare le informazioni inerenti solo dalle Catene di tipo A, per fare ciò si è utilizzata la funzione "atom.select", dove la variabile CHAIN è settata con valore A.

```
inds <- atom.select(pdb,type="ATOM",chain=CHAIN)
```

Ogni nodo delle proteine sarà un aminoacido a se stante per cui è necessario salvare le informazioni dei nodi. Per ottenere un grafo più piccolo e senza informazioni ridondanti si considerano come nodi, solo una delle permutazione consecutive degli aminoacidi.

```
list.amino <- pdb$atom[inds$atom,c("resno","resid")]
by_resno <- list.amino %>% group_by(resno)
res.table <- by_resno %>% summarise(res=unique(resid))
list.amino <- res.table$res
```

La parte inerente alla mappatura dei grafi e la creazione delle matrici di adiacenza è stata svolta tramite il linguaggio Rstudio.
Una volta ottenute le mappe di contatto e le informazioni sui nodi, si hanno a disposizione tutte le informazioni necessarie per la creazione dei grafi. Queste informazioni in particolare saranno la matrice di adiacenza dei grafi e le etichette dei nodi, ogni nodo sarà etichettato dall'aminoacido che rappresenta.

2.1 DGL

Per la scelta del frameWork si è scelta la libreria DGL [3] che è una libreria Python basata sulla libreria Pytorch [4].

2.1.1 Strutture in DGL

DGL rappresenta un grafo diretto come oggetto DGLGraph. Si può costruire un grafo specificando il numero di nodi nel grafo e l'elenco dei nodi di origine e di destinazione.
DGL rappresenta ogni nodo con un numero intero univoco, chiamato Node ID, e ciascun arco con una coppia di numeri interi corrispondenti agli ID dei suoi nodi, nodo sorgente e nodo destinazione. DGL assegna a ciascun arco un numero intero univoco, chiamato edge ID , in base all'ordine in cui è stato aggiunto al grafo. La numerazione dei Node ID e degli archi inizia da 0. In DGL, tutti gli archi sono diretti e un arco (u, v) indica che la direzione va dal nodo u al nodo v.
Per specificare più nodi, DGL utilizza un tensore di interi 1-D (ovvero, il tensore di PyTorch, il tensore di TensorFlow o il ndarray di MXNet) contenente i Node ID. DGL chiama questo formato "node-tensors". Per specificare più archi, si utilizza una tupla di tensori di nodo (U,V).
Per accedere alle informazioni inerenti agli archi e ai nodi si utilizzano le interfacce $n.data$ e $e.data$, le quali restituiscono i tensori contenenti le feauture.
Inoltre DGL permette la creazione di grafi da fonti esterne, come la libreria <networkx> [5], bisogna attenzionare il fatto che i grafi DGL possono essere solo diretti mentre networx permette la creazione di grafi indiretti, questo comporterà che nella fase di mapping bisognerà inserire per ogni arco (u,v)

l'arco (v,u), oppure utilizzando gli edge-tensor U e V, dopo aver inserito gli archi (U,V) bisognerà inserire gli archi (V,U).
L'aggiunta e la definizione di collegamenti tra nodi definisce una matrice di adiacenza interna per ogni grafo, la matrice di adiacenza è una matrice sparsa poiché solo alcuni nodi sono collegati tra loro.

2.1.2 Matrici sparse

In matematica, in particolare in analisi numerica, una matrice sparsa è una matrice i cui valori sono quasi tutti uguali a zero. Concettualmente, la sparsità si collega ai sistemi accoppiati.
Quando delle matrici sparse vengono memorizzate e gestite su un computer, risulta proficuo, e spesso anche una necessità, utilizzare algoritmi specializzati e strutture dati che tengono conto della natura sparsa della matrice. Svolgere delle operazioni utilizzando le strutture e gli algoritmi matriciali usuali risulta un'operazione molto lenta, e porta anche a grandi sprechi di memoria, se la matrice da gestire è sparsa.
I dati sparsi sono, per loro natura, facilmente comprimibili, e la loro compressione comporta quasi sempre un utilizzo significativamente inferiore di memoria.
La struttura dati classica di una matrice è quella di un array bidimensionale. Ogni elemento dell'array rappresenta il generico elemento $a_{i,j}$, a cui si può avere accesso tramite i due indici i e j. Per una matrice $m \times n$ dev'essere disponibile almeno il minimo quantitativo di memoria necessario ad immagazzinare gli $(m \times n)$ valori della matrice.
Molti, se non tutti, i valori di una matrice sparsa sono pari a zero. Il principio di base che si segue per memorizzare una matrice sparsa è di salvare, invece di tutti i valori, soltanto quelli diversi da zero. A seconda del numero e della distribuzione dei valori diversi da zero, si possono utilizzare diverse strutture dati ed ottenere risparmi considerevoli in termini di memoria impossibili da ottenere con l'approccio usuale.
Un esempio di un formato per memorizzare matrici sparse è Yale Sparse Matrix Format, che memorizza una matrice sparsa $m \times n$, M, in una riga usando tre array monodimensionali.
Sia NNZ il numero di valori diversi da zero di M. Il primo array è A, di lunghezza NNZ, e memorizza tutti i valori diversi da zero di M, ordinati da sinistra a destra e dall'alto verso il basso. Il secondo array è IA, che è lungo $m+1$, ossia un valore per riga, più uno. $IA(i)$ contiene l'indice del primo elemento di A diverso da zero della riga i-esima. La lunghezza della riga i è determinata da $IA(i+1) - IA(i)$, per questo motivo IA dev'essere di lunghezza $m+1$. Il terzo array, JA, contiene l'indice della colonna di ciascun

elemento di A, quindi JA è lunga NNZ.
La definizione delle matrici di confusione come matrici sparse permette quindi di diminuire lo spazio in memoria utilizzato per rappresentare ogni grafo, questo approccio utilizzato da DGL è molto utile considerando la grande mole di dati con cui si lavorerà.

2.1.3 Batch

Un set di dati di classificazione dei grafi contiene due tipi di elementi: un set di grafi e le relative etichette, quindi l'etichetta associata al grafo stesso.
Come per le NN quando il set di dati da analizzare è molto grande è necessario allenare il modello tramite mini-batch, nel caso delle GNN per richiamare le batch si utilizza il metodo *GraphDataLoader*, poiché ogni elemento del dataset ha associato un grafo e un'etichetta, *GraphDataLoader* restituirà due oggetti per ogni iterazione. Il primo elemento è il grafo batch e il secondo elemento è un vettore di etichette che rappresenta l'etichetta di ciascun grafo della mini-batch.
In ogni mini-batch, i grafi campionati vengono combinati in un unico grafo batch più grande tramite *dgl.batch*. Il singolo grafo batch più grande unisce tutti i grafi originali come componenti collegati separatamente, con le caratteristiche dei nodi e degli archi concatenate. Anche questo grafo più grande è un *DGLGraph*, quindi lo si può ancora trattare come un normale *DGLGraph*. Contiene tuttavia le informazioni necessarie per recuperare i grafi originali, come il numero di nodi e di archi di ciascun elemento del grafo composto.

Capitolo 3

Creazione del dataset

Il dataset utilizzato per questo progetto non era pronto a disposizione, è stato reperito e creato tramite il sito *SCOPe* [6].

3.1 SCOPe

SCOPe (Structural Classification of Proteins — extended) è un database sviluppato presso il Berkeley Lab e l'UC Berkeley per estendere lo sviluppo e il mantenimento di SCOP.
SCOP è stato concepito presso l'MRC Laboratory of Molecular Biology ed è sviluppato in collaborazione con i ricercatori di Berkeley. I lavori su SCOP (versione 1) si sono conclusi nel giugno 2009 con il rilascio di SCOP 1.75.
SCOPe classifica molte strutture più recenti attraverso una combinazione di automazione e cura manuale e corregge alcuni errori in SCOP, con l'obiettivo di avere la stessa accuratezza delle versioni SCOP curate a mano. SCOPe incorpora e aggiorna anche il database Astral.
La versione attuale di SCOPe, nonché quella utilizzata per tale progetto, è la versione *SCOPe* 2.08.

3.2 Classificazione Orginale

Nel sito vi è una sezione dove sono mostrate tutte le proteine classificate negli anni dai biologi, la classificazione viene effettuata manualmente. SCOPe mette a disposizione la classificazione delle proteine basata sulla morfologia della loro composizione, classificando le proteine in 12 macro classi denominate con le lettere dell'alfabeto dalla lettera a alla lettera i, ogni classe si ramifica in micro classi seguendo una struttura ad albero. Per questo progetto si è optato per la classificazione più generale delle classi.

In una sezione del sito web vi è la possibilità di scaricare le informazioni riguardanti le proteine. Nel nostro caso specifico si scaricherà il file contenete le informazioni riguardanti la classificazione di esse, le informazioni a noi utili saranno: nome codificato della proteina, id interno della proteina e classificazione.

```
d1ux8a_ 1ux8    A:      a.1.1.1 113449
d1dlwa_ 1dlw    A:      a.1.1.1 14982
d1uvya_ 1uvy    A:      a.1.1.1 100068
d1dlya_ 1dly    A:      a.1.1.1 14983
d1uvxa_ 1uvx    A:      a.1.1.1 100067
```

Una volta scaricato il seguente file per ottenere le informazioni riguardanti la composizione morfologica di ogni proteina, quindi numero di aminoacidi, catena di sequenza e posizioni degli aminoacidi nello spazio, bisognerà accedere al dataset conservato nel sito web di WordWide Protein Data Bank.

3.2.1 WordWide Protein Data Bank

Dal 1971, l'archivio Protein Data Bank (PDB) [7] funge da archivio unico di informazioni sulle strutture 3D di proteine, acidi nucleici e insiemi complessi, inoltre l'organizzazione Worldwide PDB (wwPDB) gestisce l'archivio PDB e garantisce che il PDB sia disponibile gratuitamente e pubblicamente alla comunità globale.
Il loro obiettivo:
"Sostenere archivi core liberamente accessibili e interoperativi di dati strutturali e metadati per macromolecole biologiche come bene pubblico duraturo per promuovere la ricerca e l'istruzione di base e applicata attraverso le scienze."

3.2.1.1 Informazioni Proteine

La Protein Data Bank permette di scaricare le informazioni in vari formati. Scegliendo il formato *.pdb* la prassi per scaricare le informazioni delle proteine è la seguente:

- Recarsi nella sezione delle strutture divise per indice interno.
- Scaricare il file specifico di ogni proteina.

È importante considerare che nei file delle proteine non è presente l'informazione riguardante la classificazione della proteina stessa, per questo motivo bisognerà fare un mapping tra l'*id* della proteina scaricata e le informazioni ottenute da SCOPe.

Le informazioni relative alle proteine sono strutturare come nell'immagine sottostante.

```
ATOM      1  O5'   C A   1      -4.549   5.095   4.262  1.00 28.71           O
ATOM      2  C5'   C A   1      -4.176   6.323   3.646  1.00 27.35           C
ATOM      3  C4'   C A   1      -3.853   7.410   4.672  1.00 24.41           C
ATOM      4  O4'   C A   1      -4.992   7.650   5.512  1.00 22.53           O
ATOM      5  C3'   C A   1      -2.713   7.010   5.605  1.00 23.56           C
ATOM      6  O3'   C A   1      -1.379   7.127   5.060  1.00 21.02           O
ATOM      7  C2'   C A   1      -2.950   7.949   6.756  1.00 23.73           C
ATOM      8  O2'   C A   1      -2.407   9.267   6.554  1.00 23.93           O
ATOM      9  C1'   C A   1      -4.489   7.917   6.825  1.00 20.60           C
ATOM     10  N1    C A   1      -4.931   6.902   7.826  1.00 19.25           N
ATOM     11  C2    C A   1      -4.838   7.263   9.158  1.00 16.72           C
ATOM     12  O2    C A   1      -4.287   8.308   9.505  1.00 15.49           O
ATOM     13  N3    C A   1      -5.367   6.448  10.085  1.00 15.96           N
ATOM     14  C4    C A   1      -5.978   5.310   9.736  1.00 16.84           C
ATOM     15  N4    C A   1      -6.592   4.588  10.676  1.00 19.14           N
ATOM     16  C5    C A   1      -6.059   4.907   8.376  1.00 17.68           C
ATOM     17  C6    C A   1      -5.522   5.732   7.461  1.00 17.68           C
```

Atom identifica che la riga è relativa a un aminoacido, il numero identifica il progressivo dell'aminoacido nella sequenza della catena considerata, nella terza colonna sono presenti gli acronimi relativi agli aminoacidi, nella quinta colonna la catena esaminata, nella settima, ottava e nona colonna sono presenti le coordinate dell'aminoacido nello spazio.
Una volta ottenuto questi file è possibile creare le mappe di contatto, nonché le matrici di adiacenza, dei futuri grafi.
La parte inerente l'elaborazione del file contenente gli ID e la classificazione delle proteine è svolta in Python, l'ottenimento delle specifiche delle proteine e la creazione delle mappe di contatto è svolto tramite Rstudio.

3.2.2 Regole DGL per la creazione del Dataset

Per la creazione dei grafi si è optato per il metodo messo a disposizione dalla libreria DGL, che consiste nel conservare le informazioni inerenti i grafi all'interno di file CSV, ognuno formattato nel modo opportuno, più un file con estensione $YAML$ che servirà come file di configurazione.

I file da dover utilizzare per la creazione di un dataset di più grafi sono i seguenti:
File *meta.yaml*:

```
dataset_name: mini_multi_dataset
edge_data:
- file_name: edges.csv
node_data:
- file_name: nodes.csv
graph_data:
  file_name: graphs.csv
```

Questo file conterrà il nome del dataset, impostato nel campo dataset_name, più i nomi dei file che conterranno le informazioni riguardanti archi, nodi e label dei grafi.

File *edges.csv*:

```
graph_id,src_id,dst_id,feat
0,0,4,"0.39534097273254654,0.9422093637539785,0.634899790318452"
0,3,0,"0.04486384200747007,0.6453746567017163,0.8757520744192612"
0,3,2,"0.9397636966928355,0.6526403892728874,0.8643238446466464"
0,1,1,"0.40559906615287566,0.9848072295736628,0.493888090726854"
0,4,1,"0.253458867276219,0.9168191778828504,0.47224962583565544"
0,0,1,"0.3219496197945605,0.3439899477636117,0.7051530741717352"
0,2,1,"0.692873149428549,0.4770019763881086,0.21937428942781778"
0,4,0,"0.620118223673067,0.08691420300562658,0.86573472329756"
0,2,1,"0.00743445923710373,0.5251800239734318,0.054016385555202384"
0,4,1,"0.6776417760682221,0.7291568018841328,0.4523600060547709"
```

Questo file conterrà nella prima colonna un indice che indicherà il numero del grafo di cui si stanno descrivendo gli archi; nella seconda e terza colonna il nodo sorgente e destinazione dell'arco, mentre nell'ultima colonna le informazioni relative alle feature degli archi.

File *node.csv*:

```
graph_id,node_id,feat
0,0,"0.5725330322207948,0.8451870383322376,0.44412796119211184"
0,1,"0.6624186423087752,0.6118386331195641,0.7352138669985214"
0,2,"0.7583372765843964,0.15218126307872892,0.6810484348765842"
0,3,"0.14627522432017592,0.7457985352827006,0.1037097085190507"
0,4,"0.49037522512771525,0.8778998699783784,0.0911194482288028"
```

Questo file conterrà nella prima colonna un indice che indicherà il numero del grafo di cui si stanno descrivendo i nodi, nella seconda colonna un indice

progressivo che indicherà quale nodo si sta analizzando e nella terza colonna le feature dei nodi.

File graph.csv:

```
graph_id,feat,label
0,"0.7426272601929126,0.5197462471155317,0.8149104951283953",0
1,"0.534822233529295,0.2863627767733977,0.1154897249106891",0
```

Questo file conterrà nella prima colonna un indice che indicherà il numero del grafo di cui si stanno descrivendo le feature e le label, dove per Lebel si intende l'etichetta che verrà predetta in seguito dal modello.

Capitolo 4

Struttura della Rete neurale

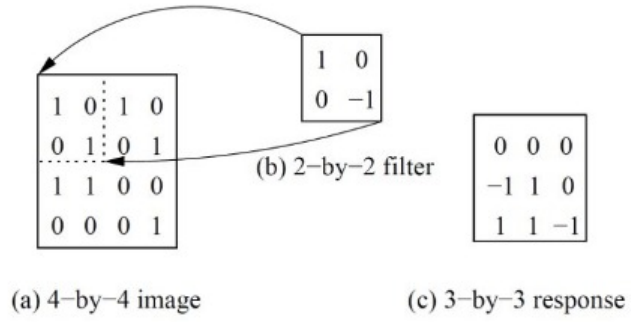
Per il modello di classificazione delle proteine sono stati scelti due Hidden layer, entrambi convoluzionali.

4.1 Convolutional Neural Network

Una rete neurale convoluzionale, CNN o ConvNet dall'inglese convolutional neural network, è un tipo di rete neurale artificiale feed-forward in cui il pattern di connettività tra i neuroni è ispirato dall'organizzazione della corteccia visiva animale, i cui neuroni individuali della retina, i fotorecettori, sono disposti in layer.
I nodi all'interno di un layer convoluzionale condividono gli stessi pesi per gli input. Generalmente si alternano i layer convoluzionali a dei layer pooled, o a volte densi, con un numero di nodi progressivamente minore.
Un layer convoluzionale è formato da una griglia di nodi.
Ogni nodo può essere immaginato come un filtro di dimensione $f \times f$ applicato in un punto della griglia di nodi del layer precedente, ed f è detta dimensione del filtro. Nell'ambito dell'image processing, questo equivale ad applicare un filtro $f \times f$ in un pixel di una immagine. Se il nodo della convolutional layer applica il filtro sul nodo $x_{i,j}$ della griglia di nodi del layer precedente, considera il quadrato di dimensione $f \times f$ il cui vertice in alto a sinistra è $x_{i,j}$ e si calcola il valore di output come segue:

$$z_{i,j} = \sum_{k}^{f} \sum_{l}^{f} w_{k,l} \times x_{i+k,j+l}$$

Se anziché considerare il vertice in alto a sinistra si considerasse il centro, la formula assumerebbe un'altra forma. Per calcolare l'output di tutti i nodi del layer convoluzionale bisogna scorrere il filtro, uguale per tutti i nodi, in lungo ed in largo sulla griglia del layer precedente ed applicarlo. Il risultato è una convoluzione del filtro sull'immagine prodotta dal layer precedente, da cui il nome.
Lo stride di un filtro indica di quante posizioni si deve scorrere una volta applicato il filtro. Se $stride = 1$ allora l'immagine in output avrà la stessa dimensione dell'immagine in input, se $stride > 1$ allora sarà più piccola.
A seconda delle dimensioni della matrice di input ricevuta, la matrice di output prodotta potrebbe avere dimensioni inferiori anche con stride pari ad 1. Nell'esempio sottostante, partendo dai pixel nell'angolo in basso a destra dell'input potrebbe essere impossibile costruire un quadrato di dimensione $f \times f$. Per ottenere un output delle stesse dimensioni dell'input, una tecnica semplice consiste nell'aggiungere alla matrice di input righe e colonne di 0. Questa tecnica prende il nome di zero padding.



4.2 Scelta parametri

La funzione di attivazione scelta è la $ReLu$, per i motivi spiegati in precedenza, analogamente la motivazione dell'utilizzo dell'entropia incrociata.
Come valore di learning rate si è scelto $0,01$ poiché esso permette la ricerca del minimo assoluto con maggiore precisione nonostante il rischio di imbattersi in un eventuale minimo locale.
Per scegliere il numero di epoche e la batch-size, cioè la dimensione della batch, sono state effettuate svariate prove al fine di trovare il miglior valore per entrambi.
La batch-size indica il numero di grafi che verranno caricati contemporaneamente in memoria e su cui verrà fatto il training e successivamente la fase di test. Un numero troppo basso potrebbe comportare ad una bassa accuratezza causata dal fatto che se la size è molto più piccola del numero di possibili classi presenti la batch non avrà sempre le n-calssi su cui fare comparazione durante il training creando un certo bias, nel caso speculare in cui la size è troppo grande si rischia di introdurre $overfitting$.
La definizione di overfitting è la seguente: la costruzione di un modello che funziona molto bene sul training set, ma non generalizzabile e funzionante su nuovi input. Le deep neural network sono particolarmente suscettibili all'overfitting.

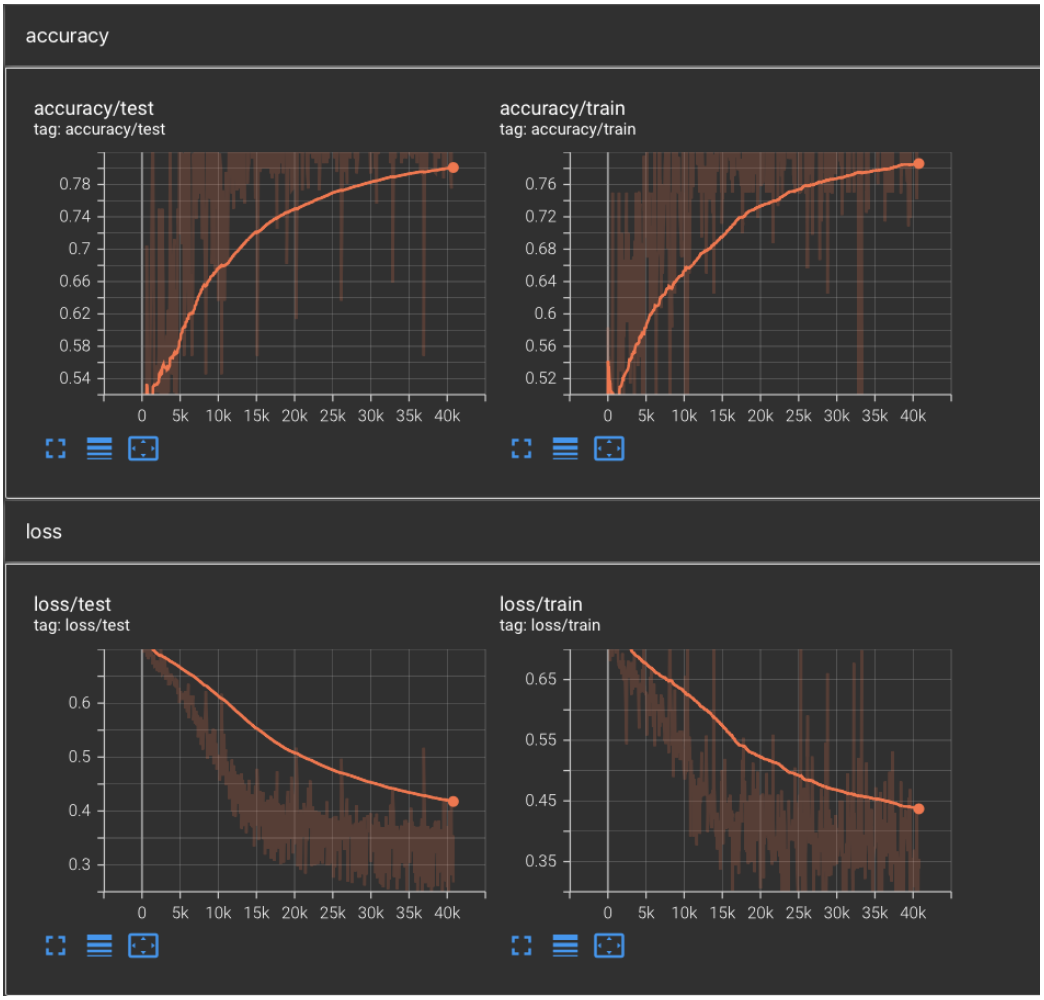
4.2.1 Training modello

Per epoche si intendono i seguenti passi effettuati durante il training di una rete neurale.
Data una rete neurale R ed un training set T:

- Si assegna ad R una matrice W di pesi iniziali scelti casualmente.
- Si addestra la rete neurale con la matrice attuale di pesi W su T.
- A partire dagli output prodotti dalla rete neurale su T e dagli output noti per T, viene calcolata la loss media su L, su tutti gli esempi del training set.
- Usando l'algoritmo di backpropagation si calcola il gradiente $\nabla_W L$ di L rispetto a W.
- Si effettua una iterazione del metodo di discesa del gradiente per aggiornare la matrice dei pesi.

- Si iterano i passi da 2 a 5 sino a quando la loss media non decresce più significativamente oppure dopo un numero fissato di iterazioni, nonchè dette epoche.

In questo progetto si è optato per bloccare il processo ad un numero di epoche prefissato, poiché durante il processo la loss non si stabilizza e non tende a convergenza.
Per la fase di test si è utilizzata l'*accuratezza* come misura per la valutazione della bontà del classificatore; dato un classificatore M e la sua matrice di confusione C, l'indice di accuratezza misura la percentuale di tuple classificate correttamente da M.
L'accuratezza e la loss sono state calcolate contemporaneamente sul training-set e sul test-set in modo da poter vedere l'andamento delle misurazioni.
Nel caso del classificatore binario con size-batch uguale a 4 e 200 epoche i grafi risultanti sono i seguenti.



4.3 Considerazione Modello

Come si può osservare dalle rette riguardanti l'accuratezza esse tendono ad avere un comportamento crescente nonostante i picchi dell'accuratezza, questo è stato il motivo che ha portato alla scelta di interrompere la fase di training a un numero fissato di epoche e non per una soglia. Nel caso della loss le rette sono decrescenti e seguono lo stesso ragionamento fatto per l'accuratezza.

4.3.1 Matrice di confusione

Per visualizzare meglio il comportamento del classificatore si è creata la matrice di confusione, la quale è una struttura utile a rappresentare l'accuratezza di un classificatore. Da tale matrice derivano varie metriche che analizzeremo in dettaglio. Sulle righe sono disposti i valori reali, mentre sulle colonne i valori predetti.
L'elemento $c_{i,j}$ contiene il numero di casi in cui il classificatore ha classificato l'osservazione nella classe j, quando la classe di appartenenza è i. Nel caso in cui $j = i$ allora la classe predetta è corretta: un buon classificatore ha valori alti nella diagonale principale e valori nulli, o molto bassi, nelle altre posizioni.
Nei casi dove si utilizza la matrice di confusione l'accuratezza viene calcolata come segue: vengono sommati tutti i valori della diagonale principale e viene rapportata la somma al numero totale di osservazioni classificate.
Al contrario, possiamo misurare la percentuale di errore, o error-rate, del classificatore sommando tutti gli elementi che non risiedono nella diagonale principale e rapportandoli al numero totale di osservazioni classificate. Tuttavia, l'error-rate è complementare all'accuratezza, per cui è possibile calcolare solo la prima e derivare l'errore come segue:

$$\text{ErrorRate}(M) = 1 - acc(M)$$

Nel nostro caso la matrice di confusione sarà di un classificatore binario, cioè il modello classificherà un item in sole due possibili classi papabili, nel caso di classificatori binari le considerazioni da fare sono le seguenti.
Supponiamo di avere un dataset con due sole classi, una indicata come P (positiva) ed una indicata come N (negativa). Indichiamo con Pos e Neg l'insieme delle osservazioni di classe P ed N rispettivamente.

Sulla base dell'esito della classificazione possiamo distinguere 4 sottoinsiemi di tuple:

Nome	Descrizione	Simbolo
True positive	tuple di classe P classificate come P	T_{pos}
True negative	tuple di classe N classificate come N	T_{neg}
False positive	tuple di classe N classificate come P	F_{pos}
False negative	tuple di classe P classificate come N	F_{neg}

4.3.2 Metriche di valutazione

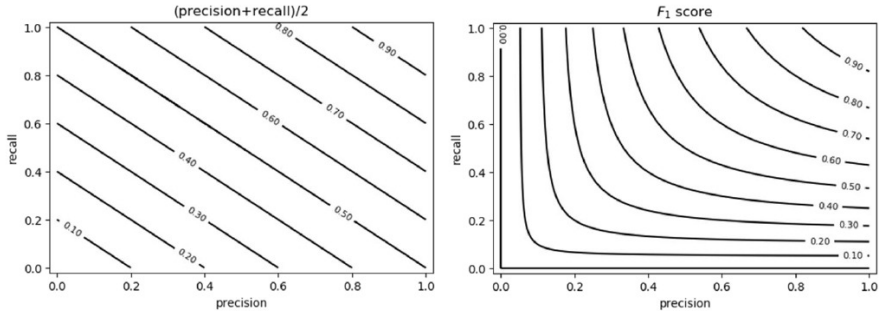
Metriche per classificatori binari:

- La recall, anche chiamata sensitività o più tecnicamente True Positive Rate, è la percentuale di osservazioni positive classificate correttamente:
$$\text{recall}(M) = \text{TPR}(M) = \frac{|T_{pos}|}{pos}$$
- La specificità, o più tecnicamente True Negative Rate, è la percentuale di osservazioni negative classificate correttamente:
$$\text{TNR}(M) = \frac{|T_{neg}|}{neg}$$
- La false positive rate è la percentuale di osservazioni negative classificate come positive. Ipotizziamo che il modello classifichi pazienti tra positivi ad un tipo di tumore o negativi: si vuole una FPR tendenzialmente bassa per evitare che al paziente siano provvisti risultati sgradevoli.
$$\text{FPR}(M) = \frac{F_{pos}}{neg}$$
- Il false discovery rate è una metrica che indica la percentuale di falsi positivi rispetto a tutte le osservazioni classificate come positive (corrette o meno che siano):
$$\text{FDR}(M) = \frac{F_{pos}}{F_{pos} + T_{pos}}$$
- La precision è la percentuale di osservazioni classificate correttamente come positive rispetto alle osservazioni classificate positive (corrette o meno che siano):
$$\text{precision}(M) = \frac{T_{pos}}{F_{pos} + T_{pos}}$$

- Sia la precision che la recall sono importanti quando si giudica un classificatore, è però possibile valutare entrambi attraverso una sola metrica, chiamata score F1. Lo score F1 è compreso tra 0 ed 1 e consiste in una media armonica tra le due metriche:

$$F_1 = 2 \times \frac{precision \times recall}{precision + recall}$$

  Se sia la precision che la recall hanno un valore alto, allora lo score F1 sarà anch'esso alto. Questa è una proprietà della media armonica, per cui si preferisce rispetto alla media canonica. Vediamo la differenza tra le due attraverso i seguenti grafici:



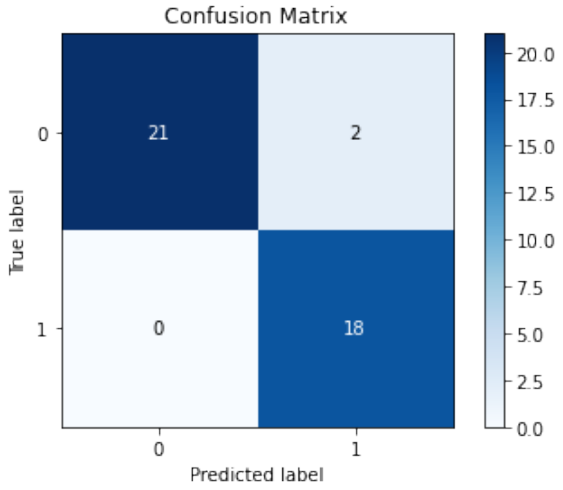
- Se il classificatore in analisi utilizza un valore soglia σ per effettuare la classificazione, allora le misure di performance vanno rivalutate al variare della soglia discriminativa.

4.4 Risultato modello

Tramite una libreria *Phython* è possibile ricavare queste informazioni tramite una sola funzione, la quale prende in input i valori predetti dal modello e i valori reali.
Ecco un esempio:

	precision	recall	f1-score	support
0	0.00	0.00	0.00	0
1	0.91	0.67	0.78	384
2	0.96	0.51	0.67	552
3	0.22	0.28	0.25	234
accuracy			0.52	1170
macro avg	0.52	0.37	0.42	1170
weighted avg	0.80	0.52	0.62	1170

Tramite la libreria scikit-plot è possibile, attraverso gli stessi input precedentemente citati, ottenere graficamente la matrice di confusione.



I plot riportati sono basati sul modello di classificazione binaria, delle proteine delle categorie A e B.

4.5 Predizione

Nel caso del modello OneVsAll è stato utilizzata una funzione logistica, tipicamente utilizzata dai regressori logistici.
La predizione è simile alla classificazione poiché costruisce un modello e usa il modello per predire valori per un dato input. La predizione è diversa rispetto alla classificazione poiché la classificazione predice valori categoriali, etichette, mentre la predizione modella funzioni a valori continui.

Diversi classificatori possono essere utilizzati come predittori (alberi decisionali, SVM), e viceversa (regressione logistica). La tecnica più importante di predizione è la regressione.
La regressione è una forma di apprendimento supervisionato che consente di apprendere una mappatura tra i dati di input e i corrispondenti output. Esempi di task legati alla regressione sono:

- Predizione del prezzo di prodotti date delle caratteristiche.
- Predizione del profitto di una certa compagnia.
- Contare il numero di automobili presenti in una immagine.

4.5.1 Regressore

Formalmente un regressore lo si definisce come come una funzione $f : R^n \rightarrow R^m$ dove n è la dimensionalità del dominio, features dei dati in input, ed m è la dimensionalità del codominio, features dei dati in output.
In generale, ci riferiremo ad un dato in input con $x \in R^n$, all'output vero legato ad x con $y \in R^m$ e all'output predetto dal regressore con $\hat{y} \in R^m$. Possiamo predire un valore a partire da x:

$$\hat{y} = f(x)$$

In questo contesto, spesso x è chiamata variabile indipendente, mentre y è chiamata variabile dipendente. Come visto nella classificazione, possiamo definire una funzione di rappresentazione $r : E \rightarrow R^n$ per mappare un input da $e \in E$ ad R^n. Utilizzando la funzione di rappresentazione, possiamo predire il valore per un input $e \in E$:

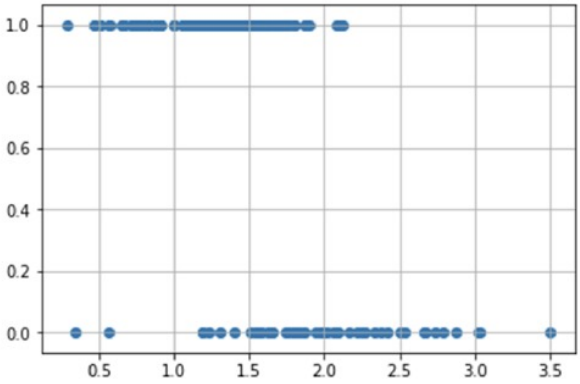
$$\hat{y} = f(r(e))$$

Per trovare la funzione di regressione f si possono utilizzare vari metodi a seconda del contesto.

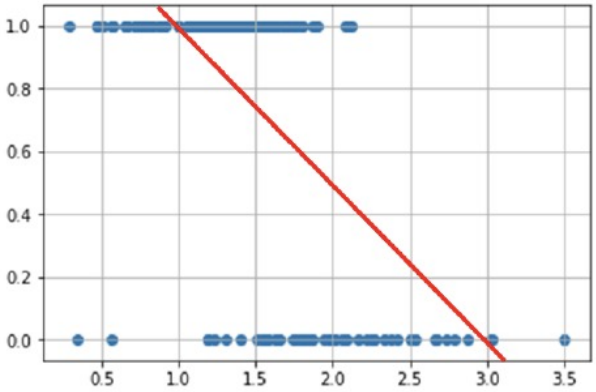
Un regressore lineare multiplo permette di associare un numero reale $\hat{y} \in R$ ad un vettore in input $x \in R^n$ attraverso una funzione parametrica f_Θ. L'obiettivo di un classificatore è simile: predire una classe $\hat{y} \in 0, ..., M-1$ a partire da un vettore in input $x \in R^n$. Vedremo com'è possibile costruire una funzione parametrica f_Θ che possa performare la classificazione $\hat{y} = f_\Theta(x)$.
Si ci concentrerà sulla task della classificazione binaria per iniziare, quindi avremo due sole classi $0, 1$. Potremmo provare ad utilizzare la regressione lineare per predire direttamente la classe binaria:

$$\hat{y} = f(x) = \Theta^T x$$

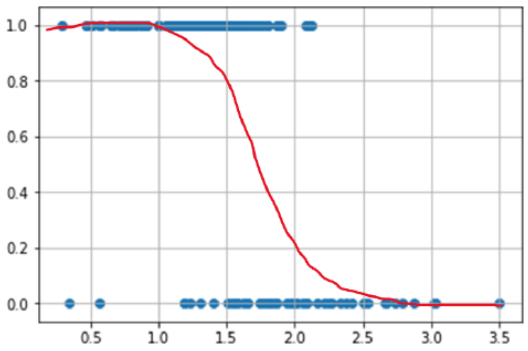
Tuttavia questo metodo risente di alcuni problemi fondamentali. Consideriamo un esempio ad una dimensione in cui degli input $x \in R$ sono classificati in due classi $0, 1$:



Idealmente vorremmo una funzione che associ tutti i punti "positivi" ad 1 e tutti i punti "negativi" a 0. Tale funzione sarebbe definita come $f : R \to 0, 1$, quindi il codominio sarà discreto. Se proviamo ad utilizzare la regressione lineare, otterremo con molta probabilità una linea del genere:



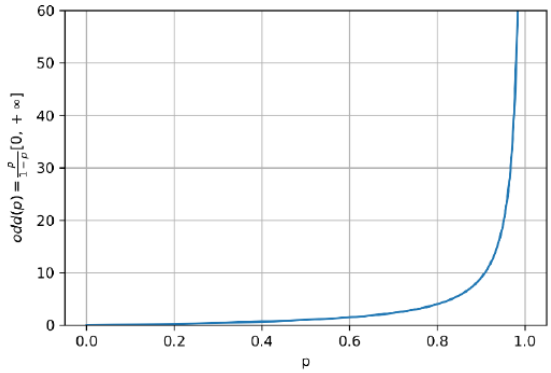
Possiamo vedere immediatamente che tale funzione non risulterà essere particolarmente accurata nella classificazione: cosa fare quando i punti stanno nel mezzo? O quando stanno sopra 1 / sotto 0? Potremmo migliorare la funzione mappando gli input x alla probabilità che essi assumano il valore 1: $P(y = 1|x)$. La funzione sarebbe definita come segue: $f : R \to [0, 1]$, che risolve i problemi sull'assegnazione dei valori tra 0 ed 1, ma risulta ancora imprecisa per input fuori dal range. Anziché una retta, vorremmo utilizzare una curva ad s che copra gli elementi come segue:



Questa funzione ideale mappa gli input nel range $0, 1$ e satura a 0 ed 1, il che è naturale poiché all'avvicinarsi ad uno dei due estremi densi aumenta anche la certezza di appartenenza ad una classe. Questa analisi suggerisce che non è possibile risolvere il problema della classificazione attraverso una funzione lineare.
Si vorrebbe trovare una trasformazione della probabilità che mappi i valori da $[0, 1]$ a $(-\infty, +\infty)$ e che renda possibile l'approssimazione delle probabilità attraverso una funzione lineare.
Sia $p = P(y = 1 \mid x)$ la probabilità che un esempio sia positivo. Considereremo la funzione dispari, odd function, di p la misura della proporzione tra probabilità che l'esempio sia positivo e quella che l'esempio sia negativo:

$$odd(p) = \frac{p}{1 - p}$$

Questa prima trasformazione risolve parte del problema, di fatto mappa la probabilità dall'intervallo $[0, 1)$ all'intervallo $[0, +\infty)$. Vediamo l'effetto della mappatura nel grafico sottostante:



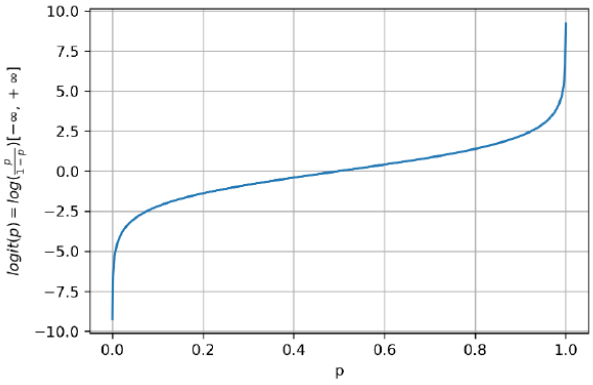
Si vuole che l'intervallo finale sia $(-\infty, +\infty)$, per cui si definisce la funzione logit come il logaritmo naturale della funzione dispari:

$$logit(p) = \log \frac{p}{1-p}$$

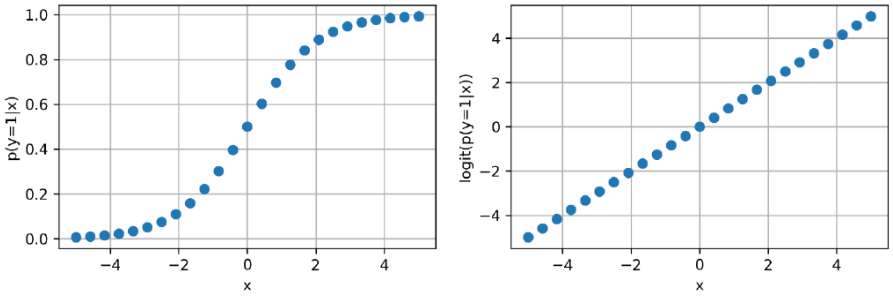
Osserviamo che:

$$logit : (0, 1) \to (-\infty, +\infty)$$

altresì confermato dal grafico della funzione:



Ci si aspetta che le probabilità siano pressoché non lineari (a forma di s), la funzione di logit permette di linearizzare i dati rispetto all'asse x, come mostrato in figura:



Ora che la funzione logit mappa le probabilità in uno spazio in cui si dispongono linearmente, si può utilizzare un regressore lineare per trovare una retta che approssimi la funzione:

$$logit(p) \approx \theta_0 x_0 + \theta_1 x_1 + ... + \theta_n x_n = \Theta^T x$$

Una volta trovati i coefficienti Θ per il regressore lineare, si deve trovare il metodo per estrapolare la probabilità dai risultati della funzione logit, quindi effettuare una mappatura inversa a quella iniziale: $(-\infty, +\infty) \rightarrow (0, 1)$.
Si può fare invertendo la funzione logit come segue:

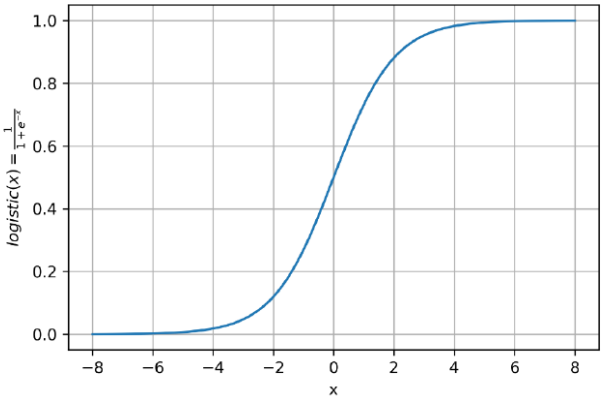
- partendo dall'espressione iniziale: $logit(p) = log\frac{p}{1-p}$
- utilizzando l'esponenziale: $e^{logit(p)} = \frac{p}{1-p}$
- moltiplicando entrambi i termini: $(1-p)e^{logit(p)} = p$
- si semplifica: $e^{logit(p)} - pe^{logit(p)} = p$
- raccogliamo per la probabilità: $p(1 + e^{logit(p)}) = e^{logit(p)}$
- estraendo la probabilità: $p = \frac{e^{logit(p)}}{(1+e^{logit(p)})}$
- si possono effettuare ulteriori semplificazioni una volta estratta la probabilità:

$$p = \frac{e^{logit(p)}}{(1 + e^{logit(p)})} = \frac{1}{(\frac{1}{e^{logit(p)}} + \frac{e^{logit(p)}}{e^{logit(p)}})} = \frac{1}{(1 + e^{-logit(p)})}$$

La funzione derivata prende il nome di funzione logistica o funzione sigmoide ed è definita in generale come segue:

$$logistic(x) = \frac{1}{1 + e^{-x}}$$

La curva disegnata dalla funzione logistica è a forma di s (da cui il nome sigmoide) come mostrato in figura:

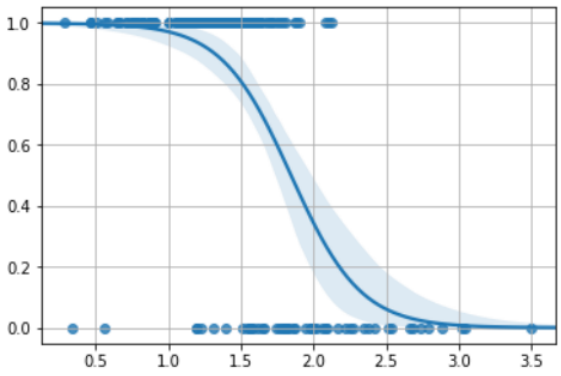


Possiamo finalmente definire il regressore logistico con la seguente funzione:

$$P(y = 1 \mid x) = f_{\Theta}(x) = \frac{1}{1 + e^{-\Theta^T x}}$$

Allenare il regressore logistico significa trovare i parametri Θ che riescano ad approssimare al meglio la probabilita $P(y = 1 \mid x)$.

La funzione approssimata non è più una retta, bensì un sigmoide:



Una volta che il modello è allenato, possiamo classificare le osservazioni attraverso la seguente regola:

$$\hat{y} = \Big\{ 1 \text{ if } x \geq 0.5 \; 0 \text{ otherwise}$$

Capitolo 5

Visualizzazione dei Vari modelli e confronto di essi

Per la realizzazione di questo progetto ci si è imbattuti nel problema riguardate la moltitudine di dati a disposizione. Nei modelli di machine learning più sono i dati a disposizione più saranno accurate le predizioni del modello, il problema è che ci si potrebbe imbattere nei problemi inerenti il Big Data. Per questo motivo si è optato per utilizzare in media 300 proteine per ogni macro classe.

5.1 Dataset

Sono stati prodotti 4 dataset ognuno per un modello di classificazione differente, li indicheremo con il nome che gli è stato dato nel file *.yaml*:

- Dataset_Binario → dataset di prova utilizzato per la creazione di un modello binario, contenente 200 grafi, cento per ogni classe.
- Dataset_quattro_classi → dataset contenente 1200 grafi, trecento per le quattro classi su cui si vuole fare la predizione.
- Dataset_sei_classi → dataset contente 1800 grafi.
- Dataset_otto_classi → dataset contenente 2400 grafi.

5.1.1 Prove sui Dataset

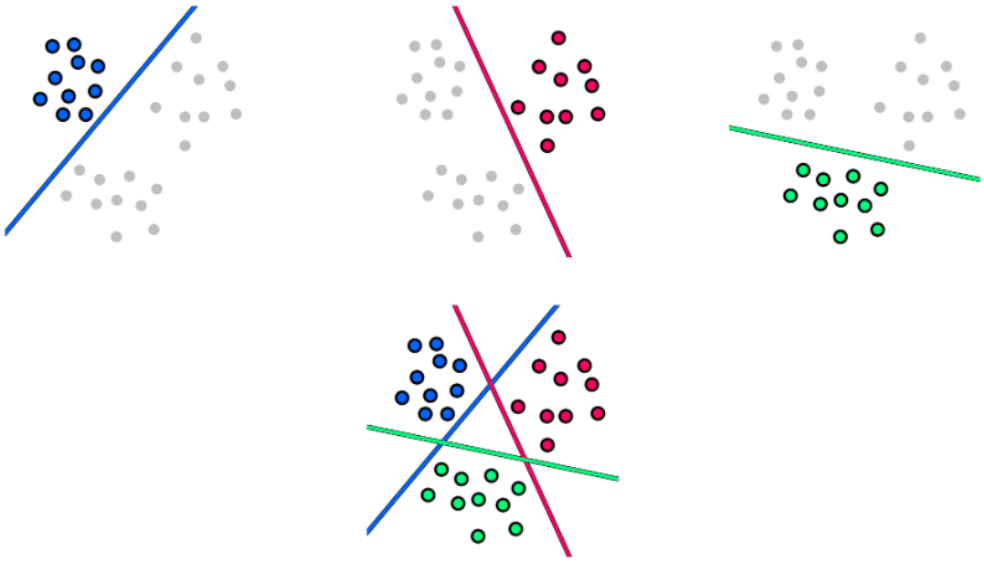
Tutti questi modelli sono stati provati molteplici volte, sia per la ricerca della batch-size ideale che per il numero di nodi interni per layer ideali. La batch-size è stato posta uguale al doppio del numero di classi presenti nel dataset. Di seguito sono riportati i valori di accuratezza riscontrati per ogni dataset alla modifica del numero di nodi per layer.

Numero di nodi per layer	Accuratezza Dataset_Binario	Accuratezza Dataset_quattro_classi	Accuratezza Dataset_sei_classi	Accuratezza Dataset_otto_classi
12	0.87804	0.55555	0.53276	0.36622
32	0.87804	0.62820	0.43874	0.44298
128	0.87804	0.56837	0.54415	0.44298
256	0.90243	0.24358	0.54415	0.36184

È importante considerare che all'aumento dei nodi per layer aumentano drasticamente i tempi di training del modello, considerando inoltre che generalmente non vi sono molte differenze tra i valori di accuratezza ottenuti si può optare per l'utilizzo di 32 nodi per layer, rendendo il modello più reattivo e veloce.
Come si può vedere dalla tabella all'aumento delle classi l'accuratezza precipita vertiginosamente, quindi ai fini di una possibile classificazione di tutte e dodici le classi questo protrebbe rivelarsi un problema. Per questo motivo si è implementato un'altro modello che utilizza la tecnica OneVsAll.

5.2 Modello OneVsAll

L'approccio one-vs-all permette di trasformare il problema della classificazione multiclasse (con più di 2 classi) in un insieme di problemi di classificazione binaria. Ogni sottoproblema può essere risolto attraverso un classificatore binario (es. il regressore logistico). Oltre al classificatore binario, è necessario che l'output contenga anche un valore di confidenza (confidence value), come una probabilità. Questo è necessario per confrontare i risultati di tutti i classificatori e capire quale potrebbe essere quello corretto.



Dato un problema di classificazione T con n classi, l'approccio one-vs-all consiste nei seguenti passi:

- Dividere T in n task di classificazione binaria T_i, dove si considera la classe i contro tutte le altre.
- Allenare i classificatori separatamente.
- Per classificare un input x utilizziamo ognuno dei classificatori f_i, poi assegniamo la classe del classificatore con il confidence value maggiore.

$$\hat{y} = arg\max_i [f_i(x)]$$

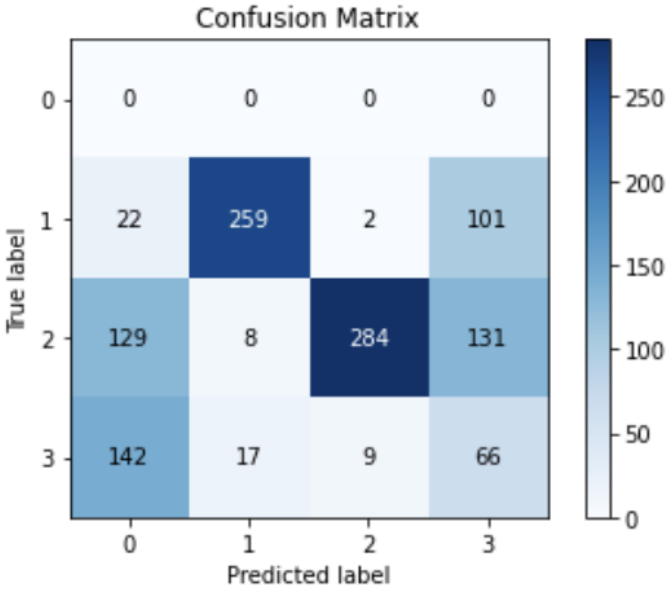
Questo modello è stato testato sul dataset contenente le informazioni riguardanti quattro tipologie di grafi con differenti label. In questo modello sono state allenate quattro reti neurali ognuna con classificazione binaria, utilizzando come batch-size 4, numero di epoche 350 e 256 nodi per layer. Una volta allenati i modelli sono stati utilizzati quattro funzioni logistiche una per ogni vettore di output dei modelli, questo è stato fatto per poter mappare i risultati in un range di valori che va da 0 a 1, in questo modo la predizione è data dal modello con probabilità più alta, il valore fornito dalla funzione logistica verrà quindi utilizzato come valore di confidenza.

5.2.1 Risultati OneVsAll

Una volta allenati tutti i modelli separatamente e classificato le proteine i risultati finali sono i seguenti:

	precision	recall	f1-score	support
0	1.00	0.91	0.95	23
1	0.90	1.00	0.95	18
accuracy			0.95	41
macro avg	0.95	0.96	0.95	41
weighted avg	0.96	0.95	0.95	41

Producendo tale matrice di confusione:



Come si può vedere dalla matrice di confusione il modello con la tecnica OneVaAll classifica correttamente la maggior parte delle proteine di classe B/1 e C/2, mentre ha difficoltà nel classificare le proteine di classe A/0, questo potrebbe essere causato dal fatto che il grafi delle proteine di tipologia A sono molto "semplici" e simili agli altri per cui il modello non riesce a cogliere le feature più importanti per classificarli correttamente.

Conclusioni

In questo progetto è stato interessante notare come il modello allenato sia in grado di classificare correttamente le proteine di classe B e di classe C, mentre non lo è stato per la classificazione delle proteine di classe A. Questo è dovuto al fatto che le proteine classificate da SCOPe come C e D sono proteine al cui interno contengono le catene di aminoacidi caratteristiche delle proteine di classe A e B. Per ovviare a questo problema si potrebbe scendere più in profondità nella classificazione delle proteine senza soffermarsi sulla prima classificazione fornita da SCOPe, sarebbe inoltre interessante provare il modello su proteine di classi indipendenti tra loro per vedere se l'accuratezza migliora o rimane costante come nel caso della classificazione di proteine di classi A, B, C e D. Il problema principale per la creazione di questo progetto è stato l'utilizzo di risorse limitate per il training del modello, a causa di ciò le fasi di training sono risultate molto lunghe e dispendiose computazionalmente, per questo motivo non è stato possibile allenare il modello OneVsAll per la classificazione di tutte e dodici le classi come si pensava inizialmente. Con i giusti mezzi sarebbe interessante valutare il modello nella classificazione di tutte e dodici le classi e scendere più in profondità nell'albero di classifciazione che offre SCOPe. Si consideri che tutt'oggi non vi sono presenti GNN che classificano proteine o molecole con una sola feature, che nel nostro caso è l'aminoacido che identifica il nodo.

Bibliografia

[1] Amal Menzli. GNN gnn. https://neptune-ai.translate.goog/blog/graph-neural-network-and-some-of-gnn-applications?_x_tr_sl=en&_x_tr_tl=it&_x_tr_hl=it&_x_tr_pto=op,sc. Accessed: 2010-09-30.

[2] Grant B.J., Rodrigues A.P.C., ElSawy K.M., McCammon J.A., and Caves L.S.D. Bio3d: An r package for the comparative analysis of protein structures. *Bioinformatics*, 22:2695–2696, Nov 2006.

[3] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

[4] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[5] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[6] Naomi K. Fox, Steven E. Brenner, and John-Marc Chandonia. SCOPe: Structural Classification of Proteins—extended, integrating SCOP and ASTRAL data and classification of new structures. *Nucleic Acids Research*, 42(D1):D304–D309, 12 2013.

[7] wwPDB consortium. Protein Data Bank: the single global archive for 3D macromolecular structure data. *Nucleic Acids Research*, 47(D1):D520–D528, 10 2018.

[8] Lemuel Puglisi. DiveIntoDataMining diveintodatamining. `https://github.com/LemuelPuglisi/DiveIntoDataMining`. Accessed: 2010-09-30.