Alma Mater Studiorum
Università di Bologna
Campus di Cesena

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA
Corso di Laurea in Ingegneria e Scienze Informatiche

GEOMETRIC DEEP LEARNING PER IL DENOISING DI MESH 3D

Elaborato in:
Computer Graphics

Relatore:
Prof.ssa
Damiana Lazzaro

Presentata da:
Stefano Scolari

Sessione I
Anno Accademico 2021-2022

A chi mi è sempre
stato vicino.

Introduzione

La crescente disponibilità di scanner 3D ha reso più semplice l'acquisizione di modelli 3D dall'ambiente.
A causa delle inevitabili imperfezioni ed errori che possono avvenire durante la fase di scansione, i modelli acquisiti possono risultare a volte inutilizzabili ed affetti da rumore.
Le tecniche di *denoising* hanno come obiettivo quello di rimuovere dalla superficie della *mesh* 3D scannerizzata i disturbi provocati dal rumore, ristabilendo le caratteristiche originali della superficie senza introdurre false informazioni.
Per risolvere questo problema, un approccio innovativo è quello di utilizzare il *Geometric Deep Learning* per addestrare una Rete Neurale in maniera da renderla in grado di eseguire efficacemente il *denoising* di *mesh*.
L'obiettivo di questa tesi è descrivere il *Geometric Deep Learning* nell'ambito del problema sotto esame.
La tesi è così organizzata:

- `Capitolo I°`: Viene introdotto il concetto di Rete Neurale, elencandone tipi differenti e funzionamento, sia per quanto riguarda l'apprendimento che l'addestramento.

- `Capitolo II°`: Si analizzano le Convolutional Neural Networks, spiegandone applicazioni e struttura.

- `Capitolo III°`: Questo capitolo è dedicato a descrivere le *mesh*, elencandone distinzioni e caratteristiche.

- `Capitolo IV°`: È analizzato il *Geometric Deep Learning*, descrivendo come funziona e mostrando la differenza fra due tipi di operatori di convoluzione su grafo, ovvero le *Graph Convolutional Networks(GCNs)* e le *Graph Attention Networks(GATs)*.

- `Capitolo V°`: In questo capitolo vengono descritti due differenti possibili approcci al problema del *denoising* utilizzando il *Geometric Deep Learning*: con un metodo supervisionato oppure un metodo non supervisionato detto *Deep Mesh Prior*.
  Inoltre, sono descritte le effettive implementazioni ed architetture di entrambi gli approcci, ponendoli a confronto in base ai risultati ottenuti in seguito ad un insieme di esperimenti per testarne l'accuratezza.

Indice

Introduzione **i**

1 Reti Neurali **1**

1.1 Storia . . . 2
1.2 Il neurone . . . 3
1.3 Funzioni di attivazione . . . 4
1.4 Struttura . . . 5
1.5 Tipi di Reti Neurali . . . 6
1.5.1 Feedforward Neural Network . . . 6
1.5.2 Recurrent Neural Network . . . 7
1.5.3 Reti fully-connected . . . 7
1.6 Apprendimento . . . 7
1.6.1 Apprendimento supervisionato . . . 8
1.6.2 Apprendimento non supervisionato . . . 9
1.6.3 Apprendimento semi-supervisionato . . . 9
1.6.4 Apprendimento supervisionato con rinforzo . . . 9
1.7 Addestramento . . . 9
1.7.1 Funzione di loss . . . 10
1.7.2 Algoritmo di apprendimento . . . 11

2 Convolutional Neural Networks (CNN) **15**

2.1 Filtri . . . 15
2.2 Layer nelle CNN . . . 17
2.2.1 Convolutional Layer . . . 17

2.2.2 ReLU Layer 19
2.2.3 Pooling Layer 19
2.2.4 Fully Connected Layer 20

3 Mesh 21
3.1 Introduzione alle Mesh 21
3.2 Manifold 23
3.2.1 Manifold non orientabili 24
3.3 Mesh poligonali 25
3.3.1 Tri-mesh 26
3.3.2 Lista di triangoli 27
3.3.3 Lista di vertici e triangoli indicizzati 27
3.3.4 Risoluzione di una mesh 28
3.3.5 Struttura di una mesh 29
3.3.6 Formula di Eulero 31
3.3.7 Mesh di triangoli 32

4 Geometric Deep Learning 37
4.1 Graph Convolutional Networks (GCNs) 38
4.1.1 Funzionamento delle GCN ad alto livello 39
4.1.2 GCN distinte in base al metodo convoluzionale 40
4.1.3 Metodo Spettrale 41
4.1.4 Metodo spaziale 42
4.2 Graph Attention Networks (GATs) 45

5 Deep Mesh Denoising 49
5.1 Denoising 49
5.2 Approccio non supervisionato 50
5.2.1 Implementazione 54
5.2.2 Risultati 54
5.2.3 Esperimenti 55
5.2.4 Problematica 57

5.3 Approccio supervisionato 57
5.3.1 Implementazione 59
5.3.2 Risultati 61
5.3.3 Confronto 66
5.4 DMP e approccio non supervisionato: confronto diretto 67
5.4.1 Risultati 68
5.4.2 Ulteriori confronti 71
5.4.3 Analisi finale 75

Conclusioni **77**

Appendice **79**
5.5 Deep Mesh Prior 79
5.5.1 networks.py 79
5.5.2 denoise.py 85
5.6 Approccio supervisionato 89
5.6.1 networks.py 89
5.6.2 trainN.py 96
5.6.3 inference.py 104

Bibliografia **109**

Ringraziamenti **112**

Elenco delle figure

1.1 Struttura insiemistica dell'Intelligenza Artificiale 1
1.2 Neurone umano 3
1.3 Struttura di un nodo di una Rete Neurale 4
1.4 Funzione di attivazione *sigmoide* 5
1.5 Funzione di attivazione *tangente iperbolica* 5
1.6 Funzione di attivazione *ReLU* 5
1.7 Funzione di attivazione *Leaky ReLU* 5
1.8 Struttura a *layer* delle ANN 6

2.1 Operazione di convoluzione 16
2.2 Layer e struttura di una CNN 17
2.3 Layer Convoluzionale 18
2.4 Layer di Pooling 19
2.5 Layer di Fully Connected 20

3.1 Una mesh 3D composta da facce triangolari 22
3.2 Due *mesh* con stessa geometria ma topologia differente 22
3.3 Due *mesh* con stessa topologia ma geometria differente 23
3.4 Orientamento compatibile fra due facce 23
3.5 Differenza fra mesh di tipo manifold e non-manifold 24
3.6 Confronto fra *closed* e *open* fan 24
3.7 Nastro di Möbius 25
3.8 Bottiglia di Klein 25
3.9 Trasformazione di una *quad-mesh* in una *tri-mesh*. 26

3.10 Trasformazione di una *polygonal-mesh* in una *tri-mesh*. 26
3.11 Rappresentazione di una *mesh* come lista di triangoli 27
3.12 Rappresentazione di una *mesh* come lista di vertici e triangoli indicizzati 28
3.13 Esempio di *mesh* dotata di un livello di risoluzione adattivo. . 29
3.14 *Mesh* con struttura regolare. 30
3.15 *Mesh* con struttura semi-regolare. 30
3.16 *Mesh* con struttura irregolare. 30
3.17 Formula di Eulero per *mesh* non semplici che presentano *genus*. 31
3.18 Esempio di grafo. 32

4.1 Rappresentazione a grafo di una rete di network. 37
4.2 Una molecola. 37
4.3 Manifold. 38
4.4 Un grafo a forma di albero. 38
4.5 Struttura schematica e riassuntiva di una GCN. 38
4.6 Rappresentazione schematica di una *GAT*(*G*raph *AT*tention network). 45
4.7 Rappresentazione della *funzione di softmax*. 46

5.1 Esempio di mesh rumorosa. 50
5.2 Mesh risultato del *denoising*. 50
5.3 Esempi di *denoising* e *completion* in *DMP*. 50
5.4 Funzionamento del modello di *Deep Mesh Prior*. 51
5.5 *Mesh* rumorosa di input. 55
5.6 *Mesh* risultato dello *smoothing*. 55
5.7 *Mesh* di output risultato del *denoising*. 55
5.8 *Mesh* di *groundtruth*. 55
5.9 *Mesh* rumorosa di input. 56
5.10 *Mesh* risultato dello *smoothing*. 56
5.11 *Mesh* di output risultato del *denoising*. 56
5.12 *Mesh* di *groundtruth*. 56

5.13 *Mesh* di un bassorilievo di un angelo utilizzata come *groundtruth*. 57
5.14 *Mesh* di un bassorilievo di un angelo affetto da rumore utilizzata come input. 57
5.15 *Mesh* di un gargoyle utilizzata come *groundtruth*. 58
5.16 *Mesh* di un gargoyle affetto da rumore utilizzata come input. . 58
5.17 *Mesh* della faccia di Giulio Cesare utilizzata come *groundtruth*. 58
5.18 *Mesh* della faccia di Giulio Cesare affetta da rumore utilizzata come input. 58
5.19 *Mesh* di una mano utilizzata come *groundtruth*. 59
5.20 *Mesh* di una mano affetta da rumore utilizzata come input. . . 59
5.21 *Mesh* dell'oggetto complesso originale. 62
5.22 *Mesh* dell'oggetto complesso rumorosa. 62
5.23 *Mesh* dell'oggetto complesso di output. 62
5.24 *Mesh* della pompa originale. 62
5.25 *Mesh* della pompa rumorosa. 62
5.26 *Mesh* della pompa di output. 62
5.27 *Mesh* del peluche originale. 63
5.28 *Mesh* del peluche rumorosa. 63
5.29 *Mesh* del peluche di output. 63
5.30 *Mesh* della sfera originale. 63
5.31 *Mesh* della sfera rumorosa. 63
5.32 *Mesh* della sfera di output. 63
5.33 *Mesh* dell'oggetto complesso originale. 64
5.34 *Mesh* dell'oggetto complesso rumorosa. 64
5.35 *Mesh* dell'oggetto complesso di output. 64
5.36 *Mesh* della pompa originale. 65
5.37 *Mesh* della pompa rumorosa. 65
5.38 *Mesh* della pompa di output. 65
5.39 *Mesh* del peluche originale. 65
5.40 *Mesh* del peluche rumorosa. 65
5.41 *Mesh* del peluche di output. 65

5.42 *Mesh* della sfera originale. 66
5.43 *Mesh* della sfera rumorosa. 66
5.44 *Mesh* della sfera di output. 66
5.45 *Grayloc Mesh* originale utilizzata come *groundtruth*. 68
5.46 *Grayloc Mesh* rumorosa fornita in input. 68
5.47 Grafico che mostra l'andamento del *MAD* per il *DMP* in base al numero di epoche trascorse. 69
5.48 Zoom sul dettaglio della *Mesh* ottenuta dopo 5000 epoche di addestramento utilizzando l'approccio superivisionato. 71
5.49 Zoom sul dettaglio della *Mesh* ottenuta dopo 5000 epoche di addestramento utilizzando *DMP*. Livello di accuratezza superiore. 71
5.50 *Bumpy Mesh* originale. 72
5.51 *Bumpy Mesh* rumorosa. 72
5.52 *Bumpy Mesh* risultato del *denoising* utilizzando il metodo supervisionato. 72
5.53 *Bumpy Mesh* risultato del *denoising* utilizzando *DMP*. 72
5.54 *Carter Mesh* originale. 73
5.55 *Carter Mesh* rumorosa. 73
5.56 *Carter Mesh* risultato del *denoising* utilizzando il metodo supervisionato. 73
5.57 *Carter Mesh* risultato del *denoising* utilizzando *DMP*. 73
5.58 *Sharp Mesh* originale. 74
5.59 *Sharp Mesh* rumorosa. 74
5.60 *Sharp Mesh* risultato del *denoising* utilizzando il metodo supervisionato. 74
5.61 *Sharp Mesh* risultato del *denoising* utilizzando *DMP*. 74

Capitolo 1

Reti Neurali

In questo primo capitolo vengono introdotti gli aspetti fondamentali che riguardano le Reti Neurali, spiegandone inoltre il funzionamento.
Le Reti Neurali, conosciute anche come Reti Neurali Artificiali(ANN), rappresentano un sottoinsieme del Machine Learning, a sua volta sottoinsieme dell'Intelligenza Artificiale. In particolare, le Reti Neurali rappresentano l'elemento cardine del Deep Learning [8].

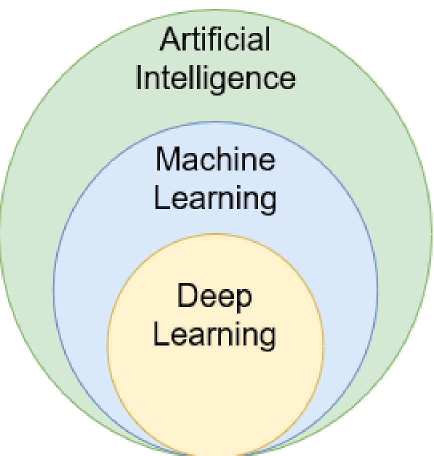


Figura 1.1: Struttura insiemistica dell'Intelligenza Artificiale

- Le reti neurali sono composte da strati di nodi

- Il funzionamento di ogni singolo nodo simula quello di un vero neurone presente nel cervello.
- Ogni nodo della Rete Neurale è adibito all'esecuzione di un determinato calcolo. Il risultato da lui calcolato è poi passato agli strati successivi della rete.

1.1 Storia

Lo studio riguardo il funzionamento dei neuroni può essere datato inizialmente al 1943, quando il neurofisiologo Warren McCulloch ed il matematico Walter Pitts pubblicarono un articolo scientifico tentando di spiegare come i neuroni potessero funzionare. Crearono inoltre un circuito elettronico così da simulare una semplice rete neurale per dimostrarne il funzionamento.
Dopo i lavori di McCulloch e Pitts del 1943, nel 1949 viene pubblicato *The Organization of Behaviour* di Donald Hebb [19]. Nel libro si evidenzia come i percorsi neurali vengano ad essere rinforzati ad ogni loro utilizzo, concetto che si avvicina molto a come l'essere umano essenzialmente *impara*.
È solo durante gli anni '50, grazie allo sviluppo tecnologico ed all'aumento di complessità e potenza dei computer, che fu finalmente possibile simulare un'ipotetica rete neurale. Ispirato dal lavoro di Walter Pitts e Warren McColluch, un ricercatore impiegato al Cornell Aeronautical Laboratory, Frank Rosenblatt, lavorò al *Perceptron*: un singolo strato di neuroni in grado di classificare delle immagini di alcune centinaia di pixel.
L'idea innovativa di Rosenblatt fu quella di implementare un algoritmo in grado di allenare i neuroni attraverso un dataset. Trasse ispirazioni da *The Organization of Behaviour* di Donal Hebb, soprattutto dall'idea che i collegamenti fra neuroni vengano rinforzati attraverso il loro utilizzo. Gli sviluppi rallentarono, a causa della struttura mono-stratificata del *Perceptron* che ne limitavano le capacità e potenzialità.
Nel 1974, Paul Werbos propose l'utilizzo di un algoritmo di *backpropagation* per ottimizzare le reti neurali. Il suo lavoro non fu però notato fino al 1986,

quando Rumelhart utilizzò la *backpropagation*, proposta da Werbos, come tecnica per addestrare le reti neurali [11].

1.2 Il neurone

Come precedentemente detto, i neuroni utilizzati nel deep learning ispirano il loro funzionamento ai neuroni del cervello umano.

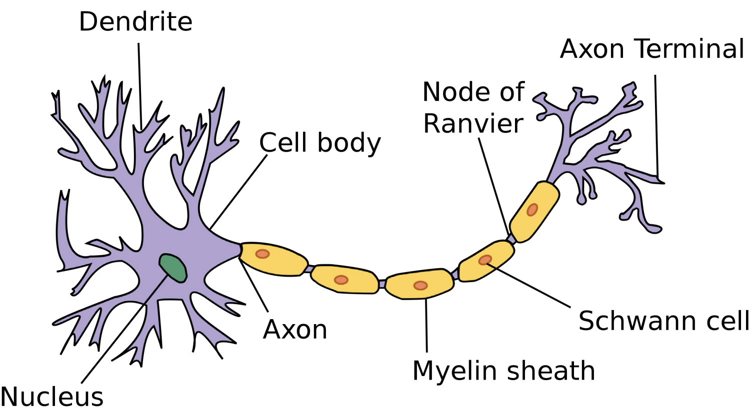


Figura 1.2: Neurone umano

Un singolo neurone risulterebbe inutile, il loro funzionamento è basato infatti sulla comunicazione che questi instaurano fra di loro.
Difatti, servono delle *reti* di neuroni per produrre qualche tipo di funzionalità. Questo perchè ricevono, elaborano e spediscono segnali.
I *dendriti* dei neuroni sono adibiti alla ricezione del segnale, che viene trasportato attraverso l'*assone*. I dendriti di un neurone sono collegati tramite le *sinapsi* all'*assone* di un altro neurone.
Nell'ambito del *deep learning*, questi concetti sono stati generalizzati.
Nelle reti neurali artificiali, i neuroni sono delle unità computazioni che hanno il compito di prendere un `input`, elaborarlo attraverso una *funzione di attivazione* con lo scopo di produrre un `output` se la *soglia di attivazione*

viene superata.
L'`input`, come si evince dalla figura 1.3, non è altro che la somma pesata di n input provenienti da altri neuroni di *layer* precedenti della rete ad esso connessi.
L'`output` è detto *attivazione* del neurone.

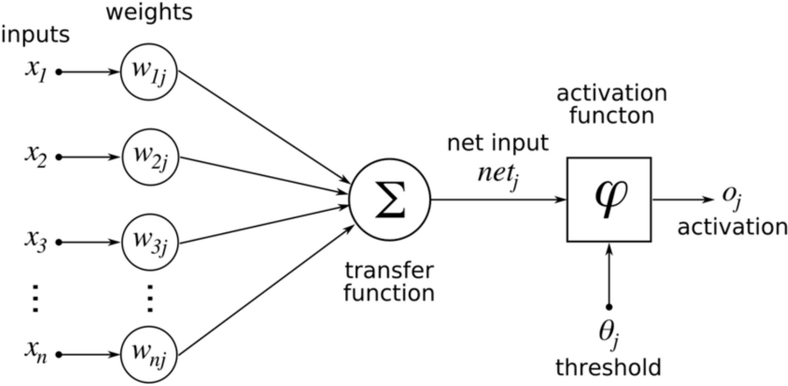


Figura 1.3: Struttura di un nodo di una Rete Neurale

1.3 Funzioni di attivazione

Lo scopo di una funzione di attivazione è di decidere se un neurone debba essere *attivato* oppure no. Questo significa che deciderà se l'`input` del neurone sia di rilevanza in quella determinata fase del processo predittivo.
Le principali funzioni di attivazioni (conosciute anche come *transfer functions*) utilizzate nelle Reti Neurali sono [20]:

- Funzione *sigmoide* o *logistica* $\boxed{f(x) = \frac{1}{1+e^{-x}}}$
- Funzione *tangente iperbolica* $\boxed{f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}}$
- Funzione *Rectified Linear Unit (ReLU)* $\boxed{f(x) = max(0, x)}$
- Funzione *Leaky ReLU* $\boxed{f(x) = max(ax, x)}$ dove a è piccolo, solitamente $\simeq 0.05$

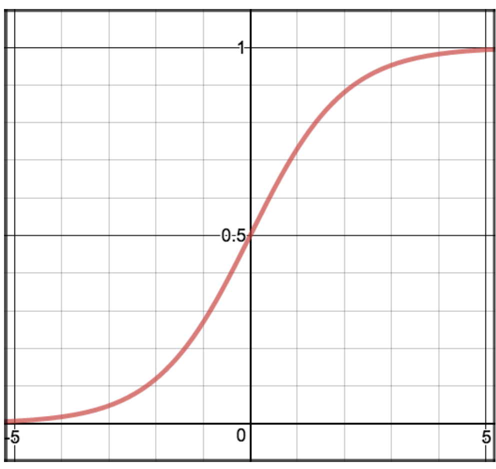


Figura 1.4: Funzione di attivazione *sigmoide*

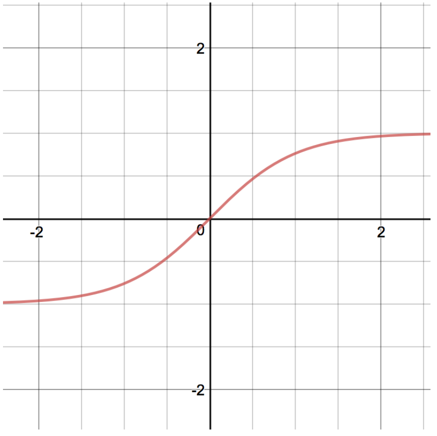


Figura 1.5: Funzione di attivazione *tangente iperbolica*

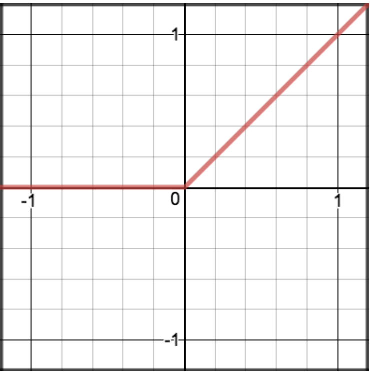


Figura 1.6: Funzione di attivazione *ReLU*

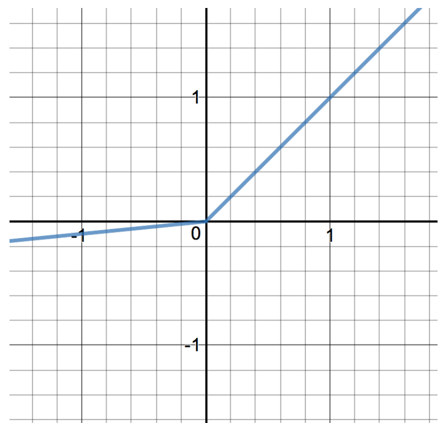


Figura 1.7: Funzione di attivazione *Leaky ReLU*

1.4 Struttura

Le reti neurali possiedono una struttura stratificata. Ogni strato è detto *layer*, ed è composto da un determinato numero di neuroni.
Il primo strato della rete è chiamato `input` *layer*, l'ultimo è il *layer* di `output` e quelli intermedi sono chiamati *hidden layers*.

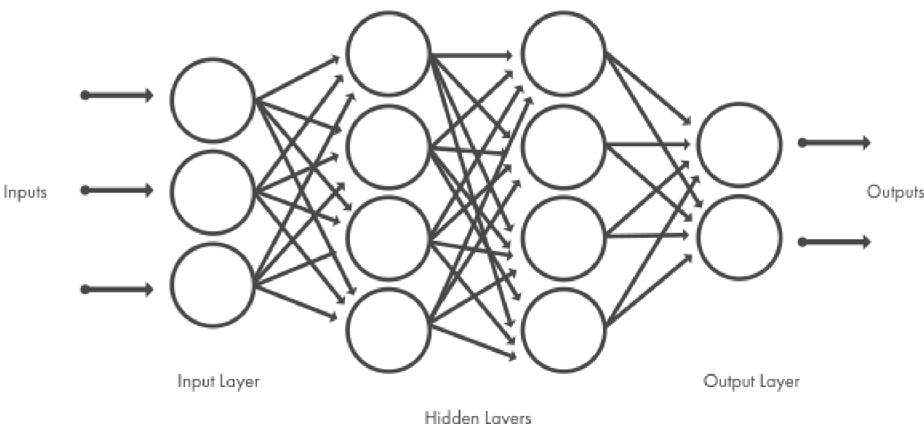


Figura 1.8: Struttura a *layer* delle ANN

Per comprendere la struttura di una ANN ci si può ricondurre alla teoria dei grafi.
I collegamenti fra nodi possono essere visti come degli *edge* a ciascuno dei quali è assegnato un *weight* che controlla il segnale fra due neuroni della rete.

1.5 Tipi di Reti Neurali

A condizionare fortemente le strutture delle reti sono le interconnessioni sinaptiche fra i nodi.
La disposizione dei neuroni gioca quindi un ruolo determinante nella classificazione delle reti e nel loro scopo di utilizzo.
Generalmente si distinguono due tipi principali di ANN [16]:

- *Feedforward Neural Network*
- *Recurrent Neural Network*

1.5.1 Feedforward Neural Network

Fu il primo modello di ANN implementato ed ha la caratteristica che le connessioni fra i suoi nodi non formano cicli. In questo tipo di rete le infor-

mazioni sono propagate soltanto in avanti (*forward*), partendo dai nodi di `input`, passando per i nodi negli *hidden layer* per poi giungere a quelli di `output`.

1.5.2 Recurrent Neural Network

L'archittettura implementata da questi tipi di reti è completamente differente dalle *Feedforward Neural Network*. Nelle *Recurrent Neural Network* le informazioni non sono propagate soltanto in avanti, infatti le informazioni provenienti da input precedenti vengono utilizzate per modificare gli attuali `input` ed `output`. In più, differiscono anche per la presenza di cicli nelle connessioni inter-nodali fra neuroni.

1.5.3 Reti fully-connected

In una rete *fully-connected* ogni nodo di uno strato l è collegato ad ogni nodo dello strato $l+1$. Il vantaggio di reti di questo tipo è la loro capacità di essere estremamente generali e fondamentalmente "agnostiche" al tipo di input.
Nonostante risultino vantaggiose per la loro vastissima possibilità applicativa, soffrono di performance minori se paragonate a reti progettate in maniera più specifica e verticale rispetto al problema in esame.
Le *reti fully-connected*, inoltre, possono essere distinte in *stratificate* e *non-stratificate*.
Nel caso di reti *reti fully-connected non-stratificate*, ogni nodo della rete è collegato con ogni altro nodo della rete.

1.6 Apprendimento

Gli algoritmi di Machine Learning si distinguono in base al tipo di paradigma di apprendimento utilizzato per addestrarli. In base alla funzione ed all'ambito applicativo si sceglie un differente tipo di paradigma di apprendimento [5]. Questi possono essere distinti in:

- Apprendimento supervisionato
- Apprendimento non supervisionato
- Apprendimento semi-supervisionato
- Apprendimento supervisionato con rinforzo

Prima di proseguire a spiegare le distinzioni e caratteristiche dei diversi paradigmi, è importante comprendere come sono strutturate le informazioni date in pasto a questi algoritmi.
Si distinguono in:

- `Labeled data`: può essere visto come un insieme di coppie di dati già etichettati, dove per ogni dato di input vi è il corrispondente output desiderato.
- `Unlabeled data`: insieme di dati privi di qualsiasi tipo di etichettatura.

1.6.1 Apprendimento supervisionato

Questi tipi di algoritmi utilizzano dati etichettati in maniera tale da essere in grado di classificare correttamente i dati o di predire un risultato accuratamente.
Man mano che all'algoritmo sono forniti dati in input, questo modifica i *pesi* dei collegamenti così da approssimare accuratamente il modello.
La precisione ed accuratezza viene misurata per mezzo di una *loss function*, e l'addestramento prosegue finchè l'errore calcolato non è stato sufficientemente minimizzato.
Negli algoritmi di tipo supervisionato, i dati sono suddivisi in:

- *Training set*: utilizzato per addestrare la rete in fase di *training*.
- *Validation set*: utilizzato per valutare l'accuratezza della rete durante l'addestramento in fase di *validation*.
- *Testing set*: utilizzato per valutare l'accuratezza della rete terminato l'addestramento in fase di *testing*.

1.6.2 Apprendimento non supervisionato

Nel caso di addestramento non supervisionato, l'ottimizzazione non tiene conto dell'errore. Questi tipi di algoritmi sono in grado di cercare e trovare autonomamente correlazioni fra i dati, anche in maniere prima impensabili per i progettisti.
Risolve quindi la problematica legata alla necessità di etichettare i dati.

1.6.3 Apprendimento semi-supervisionato

Questo tipo di algoritmo si pone a metà fra quelli supervisionati e quelli non supervisionati.
Nell'addestramento semi-supervisionato si utilizza un piccolo numero di dati etichettati ed una grande quantità di dati non etichettati. Combinando entrambe le tecniche viste in precedenza è in grado di sfruttare le potenzialità di entrambi, non avendo però il problema di dover etichettare una grande quantità di dati in input per funzionare [14].

1.6.4 Apprendimento supervisionato con rinforzo

Il funzionamento di questo tipo di algoritmi si basa sul concetto di *ricompensa*(*reward*). Vengono infatti ricompensati comportamenti voluti e puniti quelli non voluti. L'algoritmo viene in questo caso considerato un *agente*, in grado di interpetare l'ambiente in cui si trova, svolgere azioni ed imparare sbagliando.
Lo scopo è quello di incoraggiare l'*agente* a comportarsi nella maniera voluta ricompensandolo o punendolo in base alle sue azioni, in questo modo l'*agente* è indirizzato a cercare la soluzione finale migliore tramite tanti piccoli *reward*.

1.7 Addestramento

Dopo aver compreso la differenza fra i vari tipi di paradigmi di apprendimento per una rete neurale, è ora importante comprendere come sia possibile

addestrare una ANN in maniera tale che sia in grado di ristrutturare il proprio stato interno per ottenere risultati migliori.

1.7.1 Funzione di loss

Definiamo la *loss function* come $L = L(D, w)$, dove D rappresenta il dataset mentre w un insieme di parametri, detti comunemente pesi. Questa funzione ci fornisce una misura della differenza tra il target e l'output determinato dall'algoritmo, fornendo quindi una stima dell'attuale accuratezza con cui è in grado di descrivere il set di dati. L'obbiettivo è quindi quello di minimizzare il risultato della *loss function* modificando i parametri del modello.
Generalmente, per minimizzare la *loss function* si utilizza la tecnica della *discesa del gradiente* che è un'algoritmo di ottimizzazione utilizzato per trovare un minimo locale di una funzione differenziabile.
I parametri w sono inizializzati in maniera casuale e viene poi calcolato il gradiente della *loss function* ed i parametri sono quindi modificati seguendo la direzione verso cui il gradiente decresce [15]:

$$w_{l+1} = w_l - r\nabla_w L$$

dove:

- `r`: è il *learning rate* ed indica quanto il modello debba essere modificato in risposta al calcolo dell'errore. Più nello specifico indica quanto spostarsi lungo la direzione indicata dal gradiente.
- $\nabla_w[L(w)] = (\partial L(w)w_1, ..., \partial L(w)w_n)$

Queste operazioni vengono ripetute fino a che la funzione di *loss* raggiunge il suo minimo.
Una delle funzioni di loss più comuni è la *cross-entropy loss function* [3].
Per problemi di classificazione multi-classe in cui gli output del modello sono valori compresi fra 0 ed 1 risulta infatti una delle più convenienti da utilizzare.
Nel caso di una classificazione multi-classe, ovvero in cui si tenta di classificare input appartenenti a più di 2 classi, si calcola la loss per ciascuna classe e si

sommano i risultati:

$$-\sum_{c=1}^{M} y_{o,c} log(p_{o,c})$$

dove:

- `M`: rappresenta il numero di classi.
- `y`: vale 0 oppure 1 se c è la corretta classe di classificazione per il dato osservato o.
- `p`: predizione probabilistica riguardo l'appartenenza di o alla classe c.

Un'alternativa alla *cross-entropy loss function*, nel caso di classificazioni di tipo binario, è la *hinge loss function* [3]. Viene utilizzata per classificazioni binarie dove i valori di target si trovato appartenenti al set $(-1, 1)$.
La *hinge loss function* assegna un errore maggiore quando il segno tra il risultato atteso e quello in output differiscono, favorendo quindi risultati in cui il segno sia concorde.

$$L(x) = max(0, 1 - y \cdot P(x))$$

La divergenza di Kullback-Leibler tra due distribuzioni di probabilità P e Q, rappresenta invece una misura della differenza tra le due.

$$H(P, Q) = \sum_{x}^{X} P(x) \cdot log(Q(x))$$

dove P tipicamente può essere vista come la reale distribuzione dei dati mentre Q è una distribuzione di probabilità discreta che solitamente rappresenta l'approssimazione di P stessa. Essendo una misura non simmetrica, ricordiamo che $H(P, Q) \neq H(Q, P)$.

1.7.2 Algoritmo di apprendimento

Uno degli algoritmi di apprendimento più utilizzato per reti di tipo *Feedforward Neural Networks* con apprendimento supervisionato è l'*Error Backpropagation(EBP)*, ovvero l'algoritmo di retropropagazione dell'errore.

Essendo lo scopo principale delle reti con apprendimento supervisionato quello di aggiustare i pesi del proprio modello in maniera tale da minimizzare l'errore del proprio output rispetto al *target* desiderato, risulta ora necessario spiegare come avvenga questa modifica dei valori dei *weight* [18].
Si possono distinguere due fasi principali:

- Forward propagation
- Backward propagation

Forward Propagation

In questa fase dell'algoritmo di *EBP* avviene il calcolo dei valori di attivazione dei neuroni per ogni *layer*, spostando il flusso in avanti. Spostandoci quindi dall'input, attraverso gli *hidden layers* ottenendo i risultati dell'output.

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$$

dove:

- `l`: rappresenta l-esimo *layer*.
- `j`: rappresenta il j-esimo neurone nel *layer* l.
- σ: rappresenta la funzione di attivazione. Solitamente si utilizza la *ReLU activation function* 1.3.
- a_j^l: è il valore dell'attivazione del j-esimo neurone del layer l. Ricordandoci che l'output del layer l equivale all'input del layer successivo $l+1 \Rightarrow a_j^l = x_j^{l+1} \Rightarrow a_j^{l-1} = x_j^l$.
- w_{jk}^l: il peso associato alla connessione presente fra il k-esimo neurone del *layer* $l-1$ con il j-esimo neurone del *layer* l.
- b_j^l: bias associato al neurone j del *layer* l.

che nella forma matriciale può essere scritto come:

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

Il procedimento viene ripetuto per ogni *layer* fino all'`output` *layer*, in cui l'ultima attivazione sarà il risultato dell'output della rete.

Backward Propagation

Durante questa fase, si procede all'indietro, partendo dal *layer* di `output` per arrivare fino a quello di `input`, calcolando progressivamente l'errore relativo ad ogni neurone, modificando di conseguenza i *weight* associati.
Come detto in precedenza, nella fase di *forward propagation*, l'attivazione dell'ultimo *layer* rappresenta il risultato del nostro output. Quest'ultimo viene utilizzato come input per la funzione di loss. Si ricorda che *funzione di loss* e *funzione di costo* vengono di seguito utilizzate e citate in maniera intercambiabile.
Dato un dataset D, la funzione di loss sarà $L = loss(s, y)$ dove s è l'output calcolato dalla nostra rete, mentre y è l'output corretto associato a quell'input.
Scriviamo L come funzione di costo generica.
$\frac{\partial L}{\partial x} = [\frac{\partial L}{\partial x_1}, \frac{\partial L}{\partial x_2}, ..., \frac{\partial L}{\partial x_m}]$ rappresenta la direzione del gradiente lungo cui spostarsi, in modo tale da poter sapere come il parametro x debba cambiare in maniera tale da minimizzare L.
Utilizzando la funzione di costo otteniamo l'errore totale sull'output, applicando poi il *gradient descent* 1.7.1 siamo in grado di aggiornare e modificare i valori dei vari weight della nostra rete:

$$w_{ij}^{l'} = w_{ij}^l - \eta \frac{\partial L(s,y)}{\partial w_{ij}^l}$$

ricordiamo che w_{ij}^l è il *weight* tra l-esimo nodo del *layer* $l-1$ e l'i-esimo nodo del *layer* l.
Inoltre, η rappresenta il parametro di *learning rate*, mantenendo questo valore costante, dopo un determinato numero di iterazioni si raggiunge un plateau

rispetto alla minimizzazione della funzione di costo, ecco perchè questo termine viene modificato per riuscire a minimizzare l'errore il più possibile.
Da notare che la modifica di η risulta un procedimento molto delicato, infatti, se assegnato con un valore troppo elevato porta il modello a convergere verso una soluzione sub-ottima, mentre un valore troppo piccolo rischia di rallentare eccessivamente la computazione.
I valori di tutti i weight vengono aggiornati alla fine dell'*EBP*.

Capitolo 2

Convolutional Neural Networks (CNN)

In questo capitolo introduciamo il concetto di Convolutional Neural Network. Ci soffermiamo ad analizzare e spiegare il funzionamento di questo tipo di rete neurale.
Le Reti Neurali Convoluzionali sono un tipo di reti neurali di tipo *feedforward*. Il loro utilizzo principale si ha nell'analisi di immagini la cui struttura ha una topologia a griglia. Sono utilizzate principalmente per l'identificazione e classificazione [4].
Operando su dati strutturati a griglia, questo tipo di reti implementa delle metodologie volte all'ottimizzazione delle operazioni. Solitamente l'input è rappresentato da una matrice a 3 dimensioni del tipo $W \cdot H \cdot C$, dove W ed H rappresentano larghezza ed altezza della matrice, mentre C sono i 3 diversi canali di colore dell'immagine.

2.1 Filtri

Il filtro, detto anche *kernel* della rete ricopre un ruolo di primaria importanza per le CNN. Il filtro viene fatto scorrere lungo tutti i pixel, mantenendo il suo elemento centrale posizionato sopra il pixel dell'immagine di cui si vuole

calcolare il nuovo valore. Quest'ultimo viene sostituito dalla somma pesata di se stesso ed i pixel sottostanti il *kernel* [4].

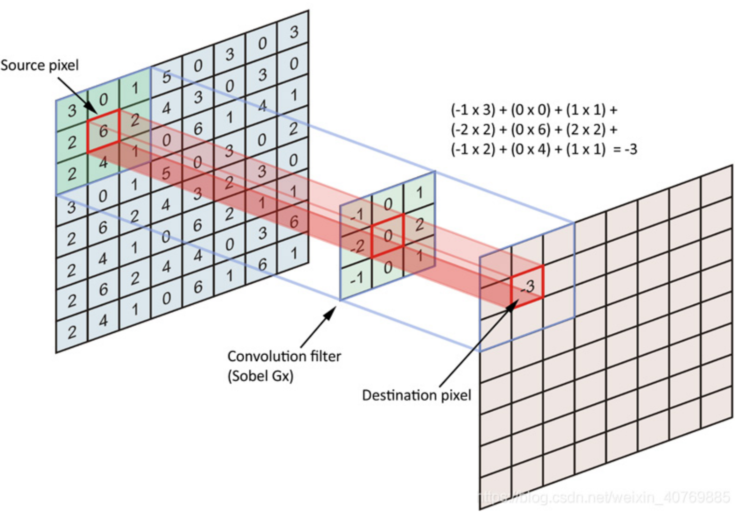


Figura 2.1: Operazione di convoluzione

Nella convoluzione, il valore del neurone non è influenzato da tutti i neuroni dello strato precedente, ma solamente da quelli adiacenti facenti parte della regione sottostante il filtro.
Sia I l'immagine sottoposta ad operazione di convoluzione mediante un filtro F di dimensione d. I nuovi valori dell'immagine saranno calcolati come:

$$I'[y,x] = \sum_{i=-d}^{d} \sum_{j=-d}^{d} F[i,j] \cdot I[y-i, x-j]$$

In più, tutti i neuroni condividono lo stesso filtro e sono soggetti agli stessi pesi nel medesimo *layer*.

2.2 Layer nelle CNN

Una Rete Neurale Convoluzionale è composta da diversi *hidden layer*, in grado di estrarre informazioni utili da un'immagine, i 4 tipi principali di layer sono [22]:

- Convolutional Layer
- ReLU layer
- Pooling layer
- Fully Connected layer

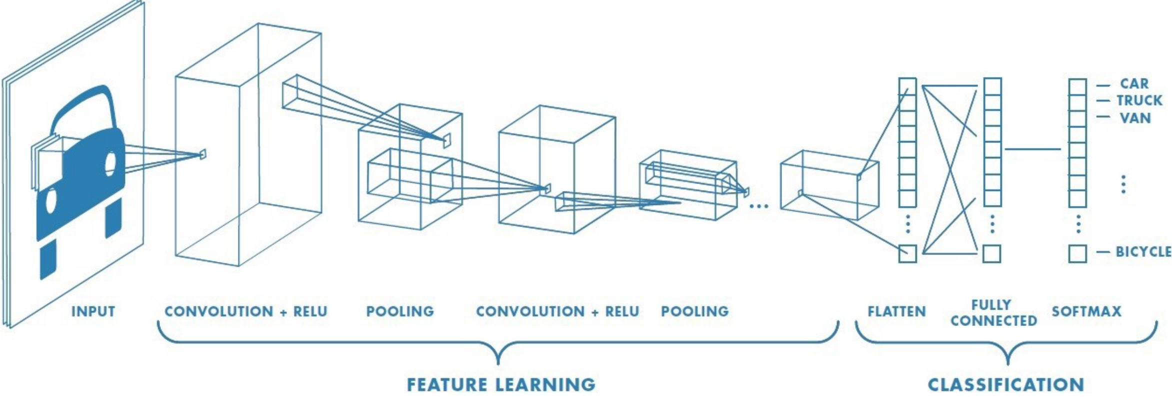


Figura 2.2: Layer e struttura di una CNN

2.2.1 Convolutional Layer

Rappresenta il primo passo nell'estrazione delle feature dall'immagine originale. In questo tipo di *layer* sono presenti diversi filtri adibiti ad eseguire diverse operazioni di convoluzione.
La convoluzione consiste nell'applicazione di un filtro ad un input, che risulta in un'*activation*. La ripetuta applicazione del filtro permette di ottenere un'*activation map*, che mostra le posizioni ed intensità delle feature ricercate in un input.

Lo scopo di un *convolutional layer* è quello di estrarre le informazioni principali dell'input, evidenziando le feature individuate e diminuendo la complessità dell'input iniziale, rendendolo maggiormente semplice da elaborare.
La novità introdotta dalle CNN è la capacità di imparare automaticamente a creare i filtri più appropriati ai propri fini, è quindi in grado di apprendere metodologie di riconoscimento autonomo in modo da identificare pattern noti.

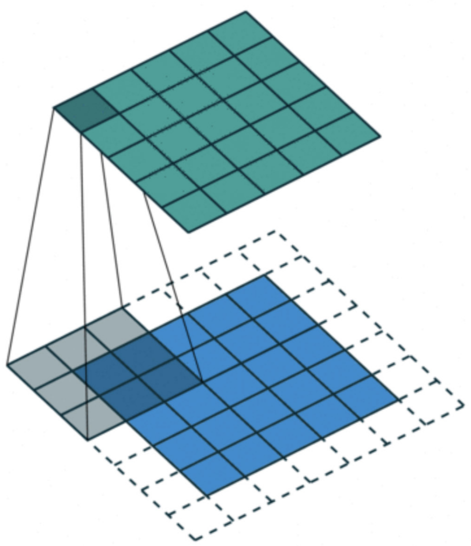


Figura 2.3: Layer Convoluzionale

Un *Convolutional layer* è caratterizzato dai seguenti parametri [4]:

- *Kernel*: rappresenta la dimensione del filtro utilizzato.
- *Stride*: indica di quanto debba spostarsi il filtro mentre scorre lungo l'immagine.
- *Padding*: indica il numero di pixel aggiunti all'immagine per permettere l'applicazione del filtro lungo i bordi.
- *Depth*: rappresenta il numero di filtri diversi che si stanno utilizzando. Il valore del depth è equivalente al valore della profondità dell'input.

2.2.2 ReLU Layer

Quello di ReLU in realtà non è un *layer* indipendente, ma rappresenta il passo successivo nell'applicazione dell'operazione di convoluzione. Il motivo per cui risulta importane applicare funzioni di attivazione come quella di ReLU è di aumentare la non-linearità del nostro input, generalmente un'immagine. Questo perchè le immagini sono naturalmente non-lineari.
Viene quindi eseguita un'operazione di *threshold* su ogni elemento dell'input, di solito quindi sui singoli pixel dell'immagine.
Vengono quindi evidenziati maggiormente i dettagli che stiamo ricercando.

2.2.3 Pooling Layer

Questi layer sono utilizzati per diminuire la dimensione spaziale dell'input, in modo tale da ridurre il costo computazionale per elaborare l'immagine. Oltre alla riduzione spaziale, hanno lo scopo di estrarre le caratteristiche dominanti, invarianti per rotazione e traslazione, in grado quindi di non alterare l'addestramento del modello.
Esistono due tipi di *Pooling layer*:

- *Max Pooling*: estrae il valore massimo dalla regione coperta dal filtro.
- *Average Pooling*: calcola il valore medio della regione coperta dal filtro.

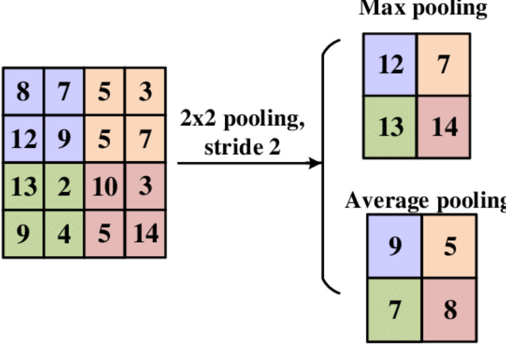


Figura 2.4: Layer di Pooling

Assieme ai layer di convoluzione e ReLU, il pooling layer forma l'i-esimo layer di una CNN, infatti spesso una CNN è composta da diverse terne di questi layer in grado di catturare dettagli di basso livello. Da notare che aumentando il numero di questi layer, il costo computazionale aumenterà considerevolmente.

2.2.4 Fully Connected Layer

Dopo le fasi di Convoluzione/ReLU/Pooling ed una fase di *flattening* dei dati, è possibile introdurre i *layer fully-connected* grazie alla riduzione della dimensione dell'input eseguita dai *layer* precedenti. In questo *layer*, infatti, tutti i neuroni di un *layer* sono connessi con quelli del *layer* precedente.
Rappresenta un modo per permettere alla rete di apprendere caratteristiche e dettagli di tipo non-lineare. Solitamente in una CNN sono presenti diversi *layer* di tipo *fully-connected*. L'ultimo di questi è composto da un numero di neuroni uguale al numero delle classi degli elementi nel dataset. Questi *layer* vengono allenati utilizzando l'algoritmo di *Error Back Propagation* 1.7.2.

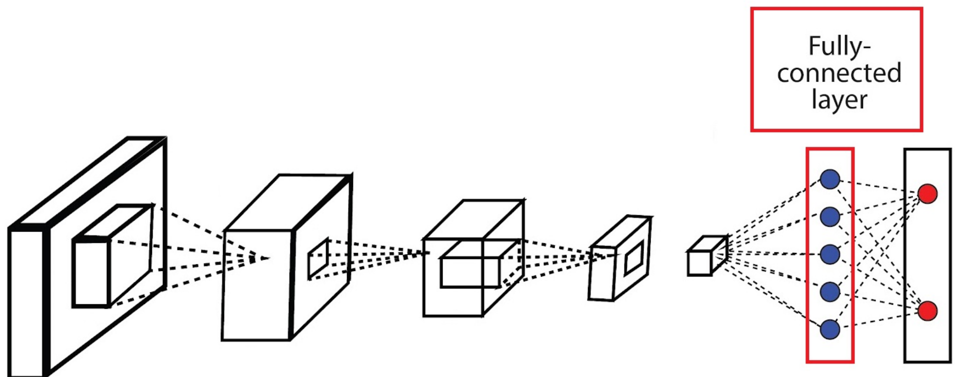


Figura 2.5: Layer di Fully Connected

Capitolo 3

Mesh

Prima di passare al *Geometric Deep Learning*, in questo capitolo verranno introdotti i concetti necessari a comprendere le *mesh poligonali*, di fondamentale importanza per comprendere le applicazioni che successivamente introdurremo.

3.1 Introduzione alle Mesh

Una *mesh poligonale* è costituita da tre tipi di elementi:

- `vertice`: è una posizione nello spazio ed ha inoltre informazioni relative al colore ed al vettore normale.
- `lato(edge)`: rappresenta la connessione fra due vertici.
- `faccia`: è un insieme chiuso di lati.

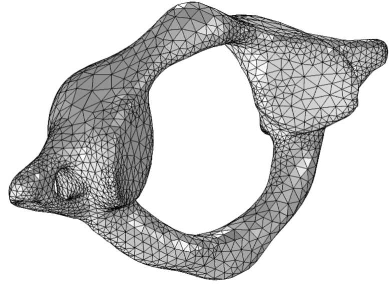


Figura 3.1: Una mesh 3D composta da facce triangolari

Importanti per descrivere le *mesh* sono la *connettività* e la *geometria* delle stesse.
Con *connettività* o *topologia* si intende la relazione di incidenza tra gli elementi della mesh.
La *geometria* invece specifica la posizione effettiva ed altre caratteristiche geometriche di ogni vertice.
Le immagini seguenti permettono di capire al meglio la differenza fra *geometria* e *topologia*.

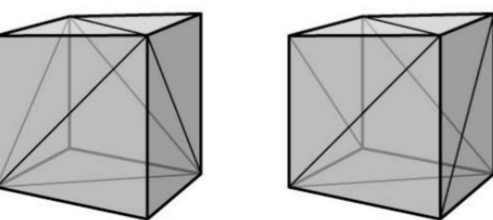


Figura 3.2: Due *mesh* con stessa geometria ma topologia differente

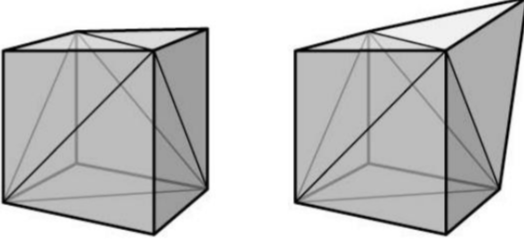


Figura 3.3: Due *mesh* con stessa topologia ma geometria differente

3.2 Manifold

Una *mesh* viene definita *manifold* se ogni lato è incidente con solo una o due facce e se le facce incidenti in ogni vertice formano un *open fan*(nel caso in cui il vertice sia di bordo) oppure un *closed fan*(nel caso in cui il vertice sia interno).
Si definisce come *orientamento* di una faccia l'ordine ciclico dei vertici incidenti. In più, l'*orientamento* di due facce adiacenti è *compatibile* se i due vertici del lato comune si trovano in ordine differente.

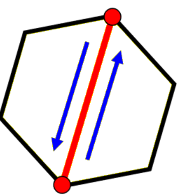
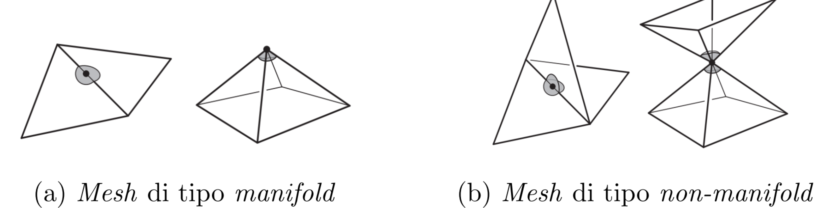


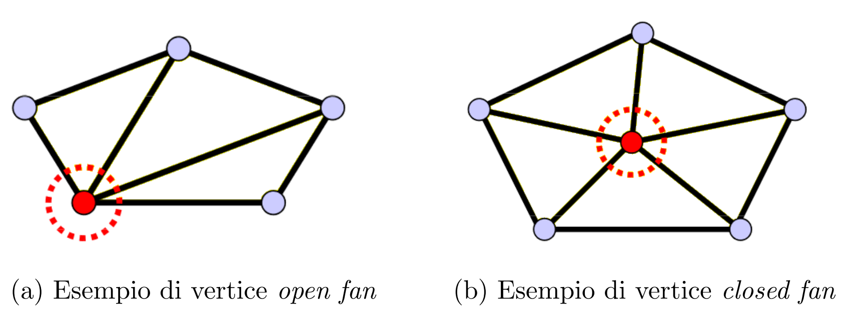
Figura 3.4: Orientamento compatibile fra due facce



(a) *Mesh* di tipo *manifold* (b) *Mesh* di tipo *non-manifold*

Figura 3.5: Differenza fra mesh di tipo manifold e non-manifold

Se ogni vertice della *mesh* ha un *closed fan*, allora il *manifold* è definito *senza boundary*. Invece, i lati che sono incidenti solo con una faccia formano il *boundary* del *manifold*. Possiamo inoltre vedere il *boundary* come l'unione di poligoni semplici.



(a) Esempio di vertice *open fan* (b) Esempio di vertice *closed fan*

Figura 3.6: Confronto fra *closed* e *open* fan

3.2.1 Manifold non orientabili

Non tutti i *manifold* sono *orientabili*, tra i più famosi troviamo il nastro di Möbius e la bottiglia di Klein.
Con *orientabilità* si intende infatti la caratteristica di un *manifold* di poter definire coerentemente su di esso un verso orario ed antiorario.
Il nastro di Möbius è un *manifold* unilaterale con *boundary*.

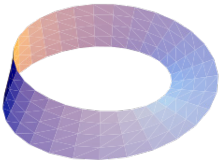


Figura 3.7: Nastro di Möbius

La bottiglia di Klein invece è un manifold privo di *boundary*, tagliando appropriamente una bottiglia di Klein si ottengono due nastri di Möbius.

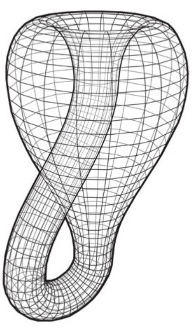


Figura 3.8: Bottiglia di Klein

3.3 Mesh poligonali

Le *mesh* vengono anche classificate in base ai poligoni di base che ne compongono la struttura.

- Tutti triangoli: *Triangular mesh* o *Tri-mesh.*
- Tutti quadrilateri: *Quad-mesh.*
- Quasi tutti quadrilateri: *Quad-dominant mesh.*

3.3.1 Tri-mesh

Le *mesh* composte da soli triangoli, presentano il vantaggio di possedere facce sempre planari, infatti tre punti nello spazio (ovvero i vertici di un triangolo), risultano sempre co-planari. In più, risulta semplice interpolare gli attributi di questo tipo di mesh. Possiamo inoltre vedere una *mesh poligonale* come una discretizzazione lineare a tratti di una superficie continua immersa in R^3.

In Computer Graphics le *mesh triangolari* sono l'unico tipo di mesh in grado di essere renderizzato ed elaborato direttamente delle GPU(Graphics Processing Unit).

Per questo motivo, *mesh* composte da poligoni differenti vengono scomposte in triangoli mediante un processo di triangolarizzazione.

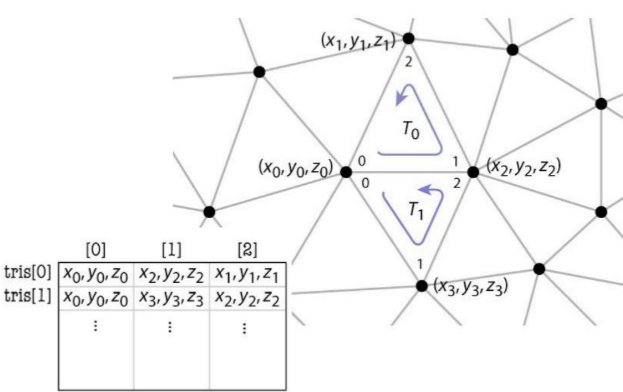
Figura 3.9: Trasformazione di una *quad-mesh* in una *tri-mesh*.

Figura 3.10: Trasformazione di una *polygonal-mesh* in una *tri-mesh*.

Per descrivere la struttura di una *mesh* è possibile utilizzare alcuni approcci differenti.

3.3.2 Lista di triangoli

Risulta il metodo più semplice, anche se soggetto a ridondanza di informazioni.



	[0]	[1]	[2]
tris[0]	x_0, y_0, z_0	x_2, y_2, z_2	x_1, y_1, z_1
tris[1]	x_0, y_0, z_0	x_3, y_3, z_3	x_2, y_2, z_2
	⋮	⋮	⋮

Figura 3.11: Rappresentazione di una *mesh* come lista di triangoli

3.3.3 Lista di vertici e triangoli indicizzati

Potendo in questo modo condividere i vertici tra triangoli differenti si riduce l'utilizzo della memoria. In più, viene garantita l'integrità della *mesh*, spostando un vertice per quel triangolo, la modifica riguarderà anche tutti i triangoli che condividono quel determinato vertice.

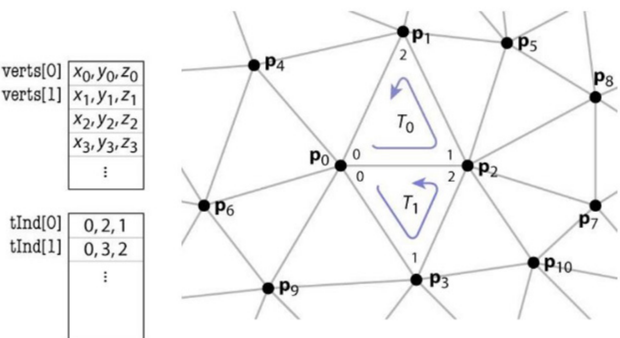


Figura 3.12: Rappresentazione di una *mesh* come lista di vertici e triangoli indicizzati

3.3.4 Risoluzione di una mesh

Con risoluzione si intende il numero di facce (oppure di vertici) che compongono la *mesh*. Se si sta ricercando un grado di accuratezza maggiore si farà utilizzo di *high-resolution mesh*. Alternativa a queste sono le *low-poly mesh*, ovvero delle *low-resolution mesh* dotate di un grado di accuratezza minore, ma adatte ad utilizzi che richiedono efficienza maggiore.
Inoltre, la risoluzione di una *mesh* può essere adattiva, ovvero dotata di un campionamento e tassellamento più fine dove necessario, per esempio dove nella mesh è presente un grado di curvatura più alto. Dove la *mesh* è più piatta basta un numero di triangoli inferiore.

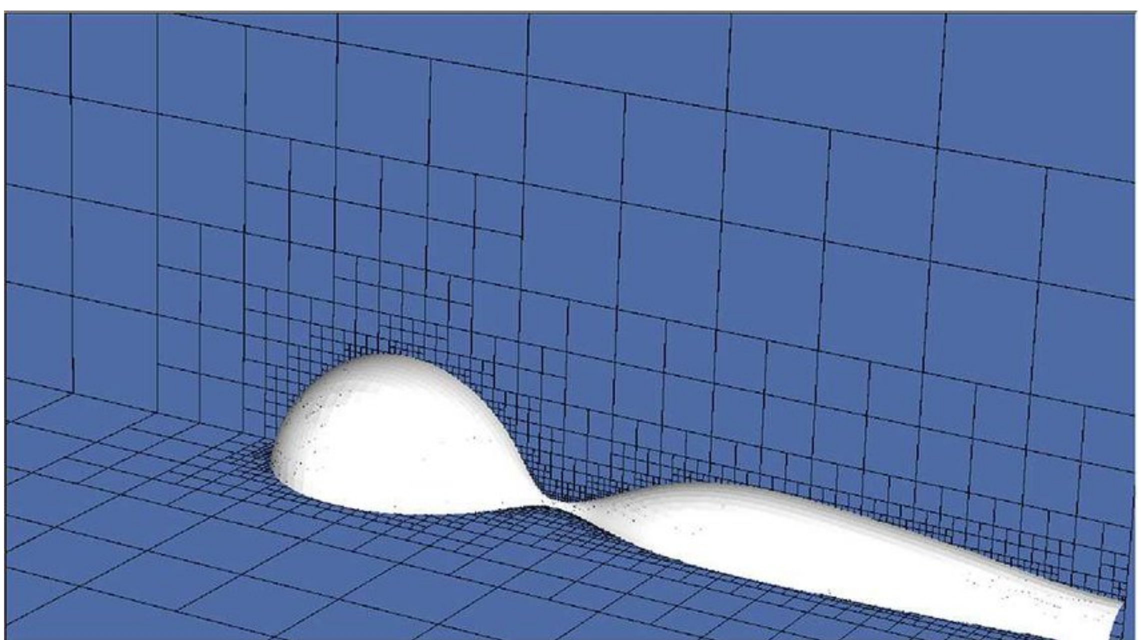


Figura 3.13: Esempio di *mesh* dotata di un livello di risoluzione adattivo.

3.3.5 Struttura di una mesh

Viene definita *valenza* di un vertice il numero di facce o di lati adiacenti a quel vertice.
Un vertice interno è detto *regolare* se possiede una valenza 4 nel caso di *quad-mesh* oppure valenza 6 nel caso di *tri-mesh*.
Di conseguenza:

- `Mesh regolare`: se tutti i suoi vertici interni sono regolari.
- `Mesh irregolare`: se solo pochi dei suoi vertici sono regolari.
- `Mesh semi-regolare`: ottenuta attraverso una suddivisione regolare di una mesh irregolare, in cui solo un piccolo numero di vertici non sono regolari.

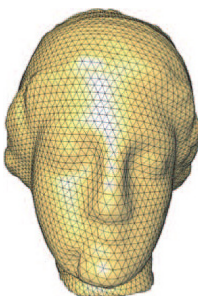


Figura 3.14: *Mesh* con struttura regolare.

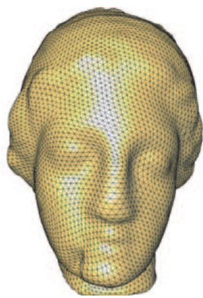


Figura 3.15: *Mesh* con struttura semi-regolare.

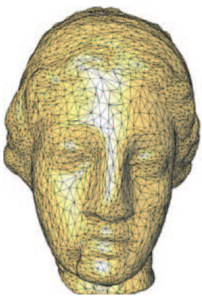


Figura 3.16: *Mesh* con struttura irregolare.

Le mesh irregolari permettono un'adattività maggiore della risoluzione. Alcuni metodi per generare mesh tipicamente producono *tri-mesh* irregolari, tra questi per esempio metodi di acquisizione 3D oppure lo *sculpting*.

3.3.6 Formula di Eulero

Sia M una *mesh* manifold priva di *boundary*, la caratteristica di Eulero-Poincarè di M è $\chi(M) = |V| - |E| + |F|$, dove $|V|$, $|E|$ ed $|F|$ sono rispettivamente il numero di vertici, lati(edge) e facce. Questa quindi fornice una relazione fondamentale tra il numero di facce, vertici e lati per una *mesh* chiusa e connessa.
Per una *mesh* connessa, semplice e solida $\Rightarrow \chi(M) = |V| - |E| + |F| = 2$, altrimenti per una *mesh* non semplice $\chi(M) = |V| - |E| + |F| = 2(1 - g)$, dove g è il numero di *genus*, ovvero di buchi che ne attraversano la superficie.

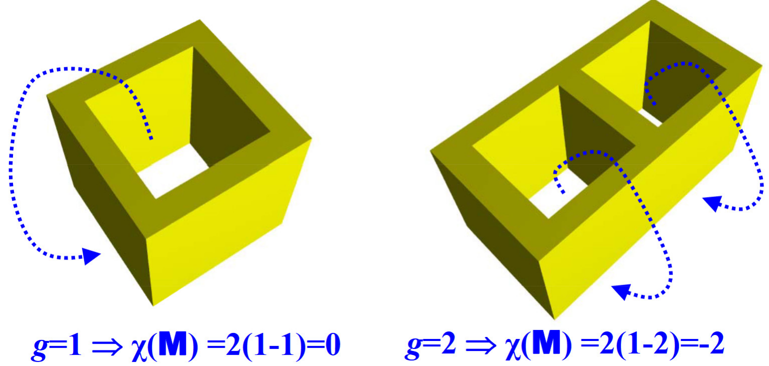


Figura 3.17: Formula di Eulero per *mesh* non semplici che presentano *genus*.

Il *boundary* di un *manifold* orientabile non è che l'unione di un insieme di poligoni semplici. Ognuno di questi poligoni delimita una faccia, e queste possono essere aggiunte assieme di nuovo a formare una *manifold* senza *boundary* così da poter applicare la caratteristica di Eulero.
La caratteristica di Eulero per un *manifold* con *boundary* è:

$$\chi(M) = 2(1 - g) - \delta$$

dove δ è il numero di poligoni di *boundary*.

3.3.7 Mesh di triangoli

Descriviamo ora una *mesh* di triangoli nelle sue componenti *geometrica* e *topologica*.
La componente *topologica* può essere rappresentata attraverso una struttura a grafo della forma (V, E, F), in cui:

- `Vertici`: V=$\{1, ..., N_v\}$
- `Lati`: E=$\{(i, j) \in V \times V : x_j \in N(X_i)\}$ tra due vertici.
- `Facce`: F=$\{(i, j, k) \in V \times V \times V : (i, j), (i, k), (k, j) \in E\}$ comprese tra 3 lati.

Definita ora una *mesh* dal punto di vista *topologico*, possiamo a questo punto rappresentare un insieme di informazioni riguardo la connettività attraverso un insieme di matrici.
Le matrici A, D ed L sono rispettivamente la *matrice di grado*, la *matrice di adiacenza* e la *matrice laplaciana*.
Prendendo per esempio il seguente grafo:

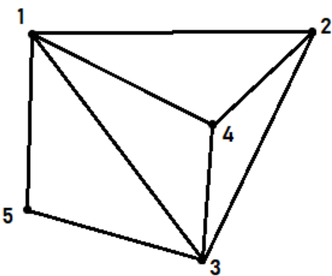


Figura 3.18: Esempio di grafo.

Matrice di adiacenza

La matrice di adiacenza contiene informazioni riguardo la connessione di vertici per mezzo di lati, in cui:

$$A_{ij} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{altrimenti} \end{cases} \tag{3.1}$$

$$\tag{3.2}$$

Per cui nel nostro caso, prendendo come riferimento il grafo 3.18, la sua matrice di adiacenza è:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{bmatrix} \tag{3.3}$$

Nel nostro caso, essendo il grafo non orientato, la matrice risulta simmetrica.

Matrice di grado

La matrice di grado di un grafo contiene informazioni riguardo il grado di ogni vertice, ovvero il numero di lati collegati ad ogni vertice. Questa è una matrice diagonale, che preso un grafo $G = (V, E)$ con $||V|| = n$ la matrice di grado D $n \times n$ è definita:

$$D_{ij} = \begin{cases} deg(v_i) & \text{se } i = j \\ 0 & \text{altrimenti.} \end{cases} \tag{3.4}$$

$$\tag{3.5}$$

inoltre ricordiamo che:

$$D_{ii} = \sum_{j=1}^{n} A_{ij}$$

Riferendosi sempre al grafo 3.18, la sua matrice di grado risulta essere:

$$D = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 \\ 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 0 & 2 \end{bmatrix} \tag{3.6}$$

Ricordando che gli indici vanno da $1, ..., ||V||$.

Matrice laplaciana

Dato un grafo $G = (V, E)$ la sua matrice laplaciana è data da:

$$L = D - A$$

dove D è la *matrice di grado* mentre A è la *matrice di adiacenza.*
Di conseguenza la *matrice laplaciana* è definita come:

$$L_{ij} = \begin{cases} -1 & \text{se } i \neq j \text{ e se } v_i \text{ è adiacente a } v_j \\ deg(\mathrm{v}_i) & \text{se } i = j \\ 0 & \text{altrimenti} \end{cases} \tag{3.7}$$

$$\tag{3.8}$$

Il grafo 3.18 può essere quindi espresso in forma matriciale come:

$$L_w = \begin{bmatrix} 4 & -1 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 & 0 \\ -1 & -1 & 4 & -1 & -1 \\ -1 & -1 & -1 & 3 & 0 \\ -1 & 0 & -1 & 0 & 2 \end{bmatrix} \tag{3.9}$$

Matrice laplaciana normalizzata

Una matrice che ci fornisce ulteriori informazioni riguardo il grafo è il *laplaciano normalizzato*, esprimibile come $L_w = D^{-1/2} L_w D^{-1/2}$. Dove L_w è il

laplaciano 3.3.7 e D è la *matrice di grado* 3.3.7.

$$L_{ij} = \begin{cases} -1 & \text{se } i = j \\ \lambda_{ij} & \text{se } (i,j) \in E \text{ (ovvero se sono vicini)} \\ 0 & \text{altrimenti} \end{cases} \tag{3.10}$$

$$\tag{3.11}$$

La cui matrice corrispondente è:

$$L_w = \begin{bmatrix} -1 & \lambda_{12} & \lambda_{13} & \lambda_{14} & \lambda_{15} \\ \lambda_{21} & -1 & \lambda_{23} & \lambda_{24} & 0 \\ \lambda_{31} & \lambda_{32} & -1 & \lambda_{34} & \lambda_{35} \\ \lambda_{41} & \lambda_{42} & \lambda_{43} & -1 & 0 \\ \lambda_{51} & 0 & \lambda_{53} & 0 & -1 \end{bmatrix} \tag{3.12}$$

dove λ_{ij} corrisponde al peso del lato fra i e j.

Capitolo 4

Geometric Deep Learning

Il *Deep Learning* solitamente viene utilizzato su dati strutturati di tipo euclideo, con dati in 1D oppure 2D.
Quella che analizzeremo di seguito è una parte del *Deep Learning* interessata all'applicazione delle *ANN* su dati non-euclidei come *grafi* e *manifold*.
I dati non-euclidei possono rappresentare informazioni complesse con un grado di accuratezza superiore rispetto alle rappresentazioni in forma euclidea.
Le *Graph Neural Network* sono una classe di metodi di *deep learning* utilizzati per lavorare su grafi [2].

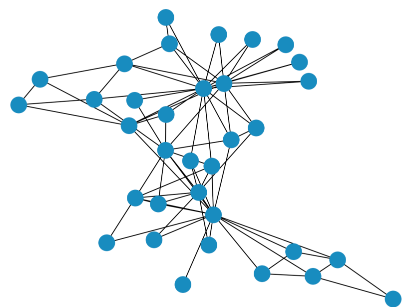


Figura 4.1: Rappresentazione a grafo di una rete di network.

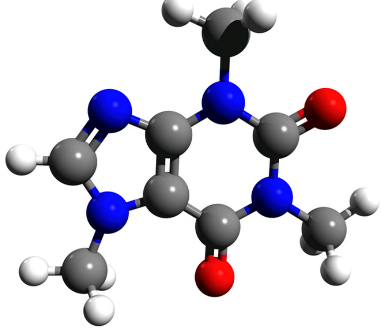
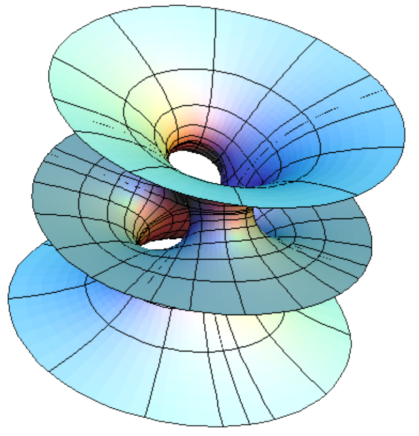


Figura 4.2: Una molecola.



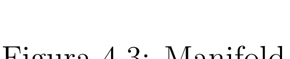


Figura 4.3: Manifold.

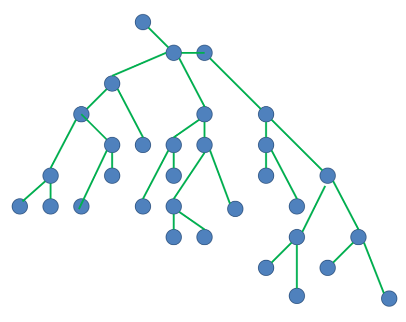


Figura 4.4: Un grafo a forma di albero.

4.1 Graph Convolutional Networks (GCNs)

L'aumento nella complessità ed irregolarità delle strutture dati da analizzare hanno portato ad un avanzamento nello studio delle *Graph Neural Network*, portando allo sviluppo di un tipo specifico di reti: le *Graph Convolutional Networks*.

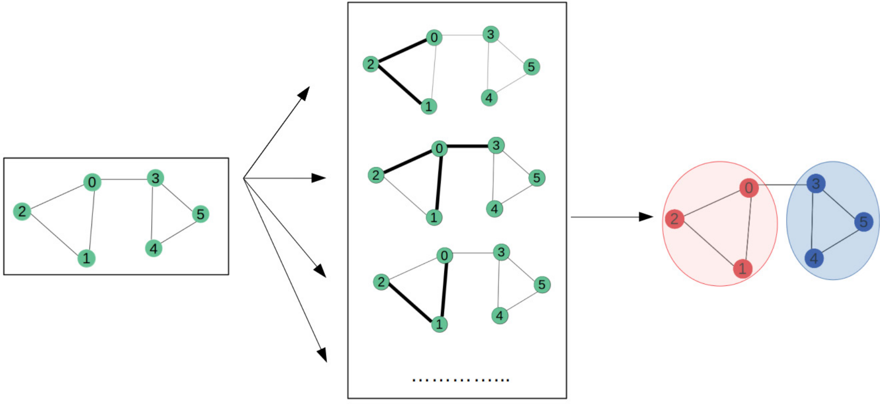


Figura 4.5: Struttura schematica e riassuntiva di una GCN.

4.1.1 Funzionamento delle GCN ad alto livello

Una GCN è sostanzialmente quindi una rete neurale che lavora su grafi. Dato un grafo $G = (V, E)$, una GCN assume in input:

- Una matrice X di dimensioni $N \times F^0$ dove N rappresenta il numero di nodi ed F^0 il numero di feature in input per ogni nodo.
- Una matrice di adiacenza A di dimensioni $N \times N$ rappresentante la struttura del grafo G.

Gli *hidden layer* H^i invece possono essere scritti come $H^i = f(H^{i-1}, A)$ dove $H^0 = X$ ed f è una funzione di propagazione. Ogni layer H^i corrisponde ad una matrice delle feature di dimensione $N \times F^i$ dove ogni riga è una rappresentazione delle feature di un nodo. In ogni *layer* le feature sono aggregate per formare le feature del *layer* successivo mediante la funzione di propagazione definita da f. Questo permette una progressiva astrazione delle *feature*, un *layer* dopo l'altro [24].
Una delle funzioni di propagazione più semplici è $f(H^i, A) = \sigma(AH^iW^i)$, dove W^i è la matrice dei pesi per il *layer* i, mentre σ è una funzione di attivazione non lineare, come per esempio la *funzione ReLU*. Da notare che la *matrice dei pesi* ha dimensione $F^i \times F^{i+1}$. La seconda dimensione della matrice dei pesi determina il numero di feature del layer successivo.
La propagazione nelle GCN è simile al *filtering* nelle CNN in quanto i pesi vengono condivisi tra i nodi del grafo, esattamente come per il *weight-sharing* nelle CNN [1].
La *funzione di propagazione* precedentemente introdotta $f(H^i, A) = \sigma(AH^iW^i)$ possiede due problemi principali, che di seguito illustriamo e per i quali è stata presentata in [13] una soluzione:

- Moltiplicando la matrice delle feature con quella delle adiacenze, consideriamo le feature dei nodi vicini, ma non quelle presenti sui nodi stessi. Per risolvere questo problema si aggiungono i *self-loop*, sommando alla *matrice di adiacenza* la *matrice identità*.

- La matrice A tipicamente non è normalizzata. Moltiplicando le *feature* con la matrice A si rischia di alterare completamente la scala delle *feature* stesse. Per risolvere questo problema la matrice A viene premoltiplicata per l'inversa della *matrice diagonale di grado* $\Rightarrow D^{-1}A$. Questo corrisponde a considerare la media delle feature dei nodi vicini. In pratica si è però scoperto che conviene introdurre una normalizzazione simmetrica sulla matrice $A \Rightarrow D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$.
Quindi:

$$f(H^i, A) = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}H^iW^i)$$

dove $\hat{A} = A + I$ (con I matrice identità) e $\hat{D}$ è la *matrice diagonale di grado* di $\hat{A}$.

4.1.2 GCN distinte in base al metodo convoluzionale

Per poter applicare e sfruttare il *deep learning* su dati non-euclidei si ha la necessità di astrarre in qualche modo l'operazione di *convoluzione*, precedentemente vista operante solo su dati 2D.
Nel caso di immagini, per esempio, che sono rappresentate da griglie euclidee 2D, il *kernel* può spostarsi in tutte le direzioni senza problemi. I grafi invece, assieme alle matrici precedentemente descritte, sono maggiormente astratti e ricchi di informazioni riguardo i dati. La chiave per generalizzare la convoluzione è il *kernel*.
Abbiamo visto come la convoluzione aggreghi informazioni dalle aree adiacenti o vicine all'entità di interesse. Queste informazioni aggregate sono utilizzate per creare delle *feature maps*, utilizzate poi per le predizioni.
Si vuole generalizzare questo concetto nel caso di dati non strutturati.
Le *GCN* possono essere suddivise in due sottoclassi a seconda del metodo che utilizzano:

- Metodo spettrale.

- Metodo spaziale.

4.1.3 Metodo Spettrale

Grazie alle ricerche di Michaël Defferrard, si ebbe una diffusione di un nuovo campo di studi nella teoria dei grafi e dell'analisi dei segnali: il *Graph Signal Processing*(GSP).
Il *GSP* rappresenta la chiave per generalizzare la convoluzione, permettendo di creare funzioni che tengano conto della generale struttura del grafo e delle caratteristiche dei singoli nodi del grafo stesso.
Vengono utilizzate funzioni di *signal processing* come la trasformata di Fourier, solitamente utilizzata per l'analisi di segnali e frequenze, applicate ai grafi. È grazie alla trasformata di Fourier sui grafi (*Graph Fourier Transform(GFT)*) che è possibile introdurre il concetto di "bandwidth" di un grafo. In senso spaziale con "bandwidth" si intende quanto siano vicini i valori di un insieme fra loro.
Dal punto di vista spettrale invece, un grafo che presenta proprietà di *clustering* possiederà un *bandwidth* abbastanza stretto nella *GFT*. Questo significa che un grafo *clustered* è invece sparso nel dominio delle frequenze, permettendo una maggiore precisione ed efficace rappresentazione dei dati.
Essendo le proprietà e caratteristiche dei nodi rappresentate tramite segnali, possiamo sfruttare il *signal processing* per apprendere informazioni da questi dati. Da notare che solitamente il segnale non rappresenta le proprietà del nodo di per sè, bensì il risultato di una funzione applicata a tali proprietà [17].
La convoluzione può essere calcolata ed eseguita dopo aver calcolato l'autodecomposizione del laplaciano relativo. Questa decomposizione permette di comprendere la struttura del grafo e di classificarne quindi anche i nodi.
Con *autodecomposizione* si intende la fattorizzazione di una matrice in forma canonica, ovvero in una forma in funzione dei suoi *autovettori* ed *autovalori*.
Data una matrice A quadrata $n \times n$ con n autovettori linearmente indipendenti q_i (con $i = 1, ..., n$) la si può fattorizzare nel seguente modo:

$$A = Q\Lambda Q^{-1}$$

dove:

- Q: la matrice $n \times n$ la cui i-esima colonna è l'autovettore q_i di A.
- Λ: la matrice diagonale i cui elementi sono i corrispondenti autovalori, $\Lambda_{ii} = \lambda_i$.

Quando la matrice da fattorizzare risulta essere una matrice *reale simmetrica* o *normale*, l'autodecomposizione è anche detta *decomposizione spettrale*, da cui il nome *Spectral Graph Convolutional Networks* [24].
Il calcolo degli autovettori del laplaciano restituisce la base di Fourier per il grafo, ed essendo eseguito sul laplaciano, avremo tanti autovettori quanti sono i nodi del grafo [13].
I passi principali per poter applicare la convoluzione tramite metodo spettrale sono:

- Trasformare il grafo nel dominio spettrale utilizzando l'autodecomposizione della matrice Laplaciano.
- Applicare l'autodecomposizione al *kernel* specifico di interesse.
- Moltiplicare il grafo in forma spettrale con il *kernel* anch'esso in forma spettrale, esattamente come si farebbe nella convoluzione classica.
- Riportare i risultati nel dominio spaziale originale, analogamente ad una *GFT* inversa.

4.1.4 Metodo spaziale

La filosofia principale dei metodi che fondano il loro funzionamento su di una rappresentazione spaziale è lo scambio di messaggi e metodi basati su sistemi di pseudo-coordinate in modo da aggregare le informazioni fra nodi, facendo poi le adeguate predizioni.

Uno dei problemi e la maggiore difficoltà degli approcci non-spettrali è quello di definire un operatore in grado di funzionare su gruppi di vicini di dimensione differente, mantenendo costante il valore del filtro per quel layer come nelle CNN.
Descriviamo il funzionamento del *metodo spaziale* a partire dalla rete *GraphSage* [9].
A partire da un'analisi ad alto livello, questa rete, presenta tre fasi principali:

Sampling dei vicini

Inizialmente si cercano i vicini diretti del nodo interessato. Poi, utilizzando il parametro di profondità k si ricercano i "vicini dei vicini" per un numero di volte pari a k. Questa è un'operazione ricorsiva.

Aggregazione

Dopo che ogni nodo del grafo ha eseguito il *sampling* dei propri vicini, bisogna "trasportare" tutte le informazioni e caratteristiche dei nodi facenti parte della rete dei vicini precedentemente individuati all'interno del nodo stesso.
In generale, è possibile definire la funzione di aggregazione per i singoli nodi come:

$$h_i^{(l+1)} = \sigma(\sum_{j \in \mathcal{N}(i)} \frac{1}{c_{ij}} W^{(l)} h_j^{(l)})$$

Dove:

- $\mathcal{N}$: rappresenta l'insieme dei nodi a distanza 1 di i. Se aggiunti i *self-loop* viene considerato anche il nodo i stesso.
- c: è una costante definita a partire dalla struttura del grafo, definisce una media isotropa.
- σ: è una funzione di attivazione, che introduce la non linearità nella trasformazione.
- W: rappresenta la matrice dei pesi composta da parametri apprendibili per la trasformazione delle feature.

Predizione

Il nodo di interesse utilizza le feature acquisite dai nodi vicini per fare delle predizioni attraverso la rete neurale, queste possono essere predizioni di tipo classificatorio oppure riguardare la determinazione di feature strutturali. È in questa fase che avviene il vero e proprio apprendimento. Questo processo a 3 step viene ripetuto per ogni nodo del grafo in maniera supervisionata.
È stato dimostrato che la funzione di aggregazione più efficace, oltre che efficiente, è quella di pooling.
GraphSage è attualmente utilizzato da Pinterest per predire quali siano le foto migliori da mostrare ai propri utenti in base ad interessi, ricerche ed interazioni [23].

4.2 Graph Attention Networks (GATs)

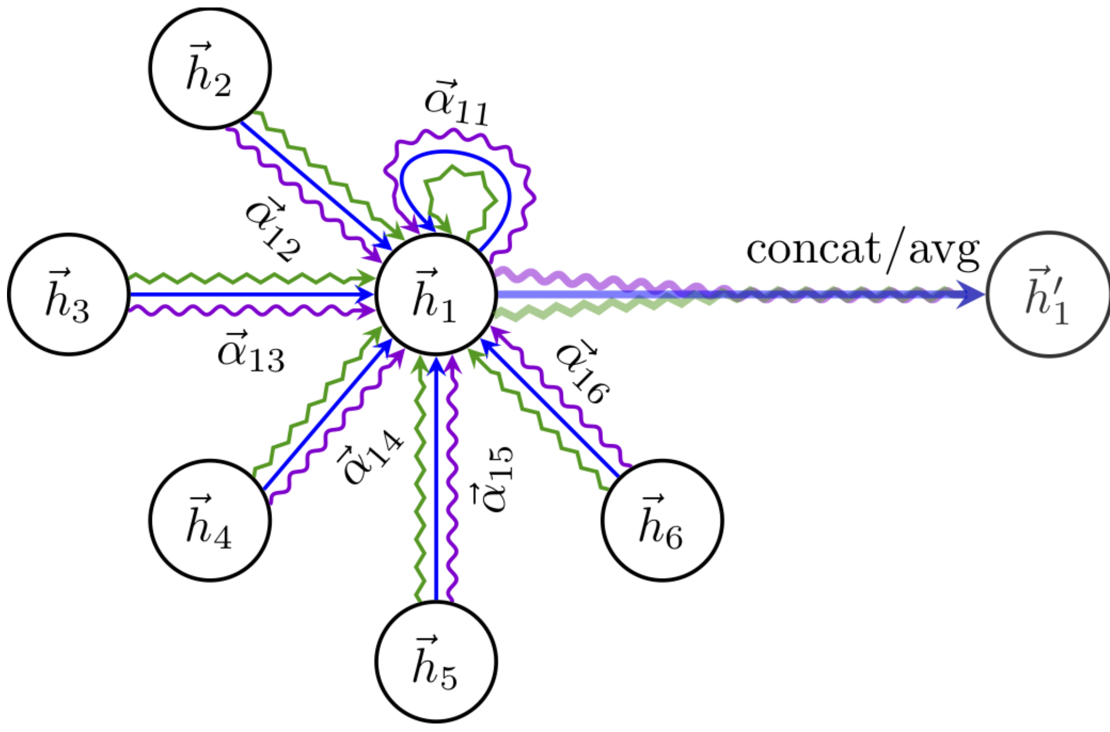


Figura 4.6: Rappresentazione schematica di una *GAT*(*G*raph *AT*tention network).

Le reti di tipo *GAT* espandono e migliorano la *funzione di aggregazione* definita nelle *GCN*. Infatti, le *GAT* sono in grado di assegnare importanza differente ai vari nodi attraverso l'*attention coefficient*, o *coefficiente di attenzione*. Questo indica quanta importanza ha un determinato nodo rispetto ad un altro, ovvero quanta attenzione un nodo debba prestare ad un altro [21].

1. $z_i^{(l)} = W^{(l)} h_i^{(l)}$,
2. $e_{ij}^{(l)} = LeakyReLU(\vec{a}^{(l)^T}(z_i^{(l)} || z_j^{(l)}))$,
3. $\alpha_{ij}^{(l)} = \frac{exp(e_{ij}^{(l)})}{\sum_{k \in \mathcal{N}(i)} exp(e_{ik}^{(l)})}$,

4. $h_i^{l+1} = \sigma(\sum_{j\in\mathcal{N}(i)} \alpha_{ij}^{(l)} z_j^{(l)})$,

1. Equazione (1): è una trasformazione lineare tra la matrice dei pesi W^l e le feature h^l di input. Questo passaggio permette di trasformare le feature di input in feature dense e di alto livello, aumentandone l'espressività.

2. Equazione (2): calcola l'*attention score* per la coppia di nodi vicini (i, j). Prima di tutto concatena le z dei due nodi calcolate mediante l'equazione al punto 1 (con $||$ si intende la concatenazione). Successivamente, esegue un prodotto scalare tra questa concatenazione ed un vettore di pesi apprendibili a. Infine, il risultato viene usato come input per la funzione di attivazione *LeakyReLU*. L'*attention score* risultante indica l'importanza del nodo vicino nel meccanismo di *message passing*.

3. Equazione (3): applica una *funzione di softmax* detta anche *funzione esponenziale normalizzata*, ovvero una specifica generalizzazione della funzione *logistica*. Viene utilizzata per convertire i valori di output precedenti in probabilità, permettendo un maggiore livello di confrontabilità tra i diversi nodi del grafo.

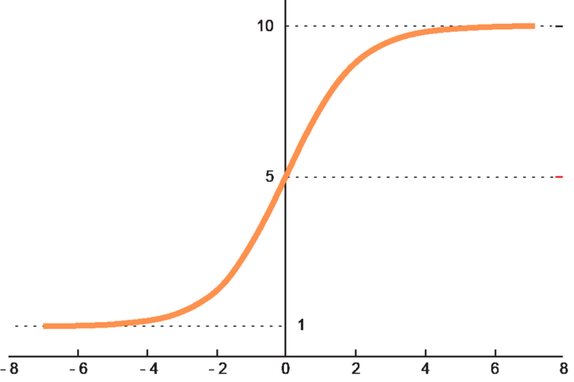


Figura 4.7: Rappresentazione della *funzione di softmax*.

4. Equazione (4) : simile alla *funzione di aggregazione* presente nelle *GCN*. In questo caso però, gli *embedding* calcolati per i vari nodi vicini, oltre ad essere aggregati, sono anche scalati in base all'*attention score*. Questo permette di differenziare i contributi dei vari nodi vicini.

Capitolo 5

Deep Mesh Denoising

In questo capitolo nell'ambito del problema del *mesh denoising* vengono messe a confronto due *Graph-Convolutional Neural Network*, una basata su un training di tipo non-supervisionato e l'altra su un approccio supervisionato.

5.1 Denoising

Il *Mesh denoising* è uno degli argomenti di ricerca principale del *geometry processing*. In particolare, negli ultimi anni, vi è stata una diffusione di un crescente numero di dispositivi in grado di eseguire scansioni 3D.
Le *mesh* acquisite sono però soggette ad un certo livello di rumore (*"noise"*) causato dall'inevitabile imperfezione che caratterizza i metodi, strumenti e tecniche di cattura e ricostruzione.
Il *denoising* di *mesh* si pone quindi come obiettivo la riduzione del rumore dal quale sono affette queste ultime, mantenendo il più possibile intatte le feature caratteristiche della *mesh* originale.

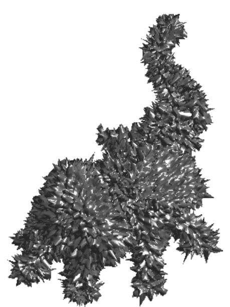


Figura 5.1: Esempio di mesh rumorosa.

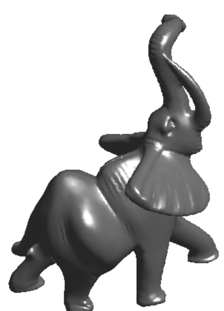


Figura 5.2: Mesh risultato del *denoising*.

5.2 Approccio non supervisionato

L'approccio *Deep Mesh Prior (DMP)* è stato introdotto in [10], si tratta di un metodo che utilizza le *GCN* in maniera non supervisionata ed ha come applicazione il *denoising* e la *completion* di *mesh 3D*. Noi ci soffermeremo ad utilizzarla solo per eseguire *denoising*.

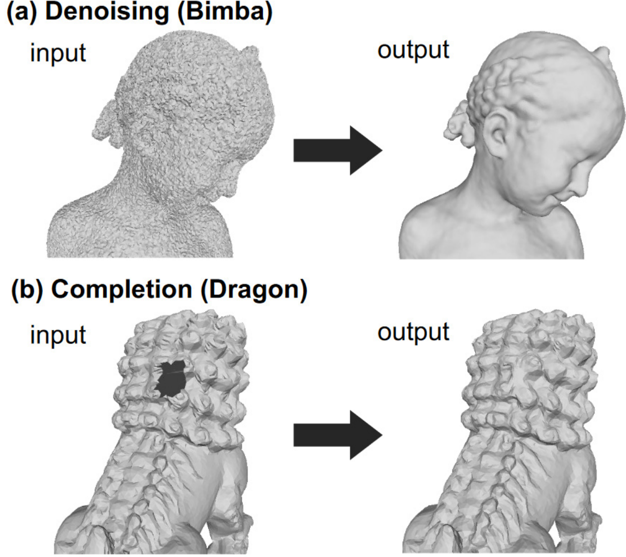


Figura 5.3: Esempi di *denoising* e *completion* in *DMP*.

Il *DMP* permette una ricostruzione e *denoising* della *mesh* senza l'utilizzo di un dataset iniziale, utilizza la sola *mesh* di input per addestrarsi ed apprendere. Basa il suo funzionamento sulle *feature* di *self-similarity* presenti nel modello [10].
Di seguito il framework di *DMP*:

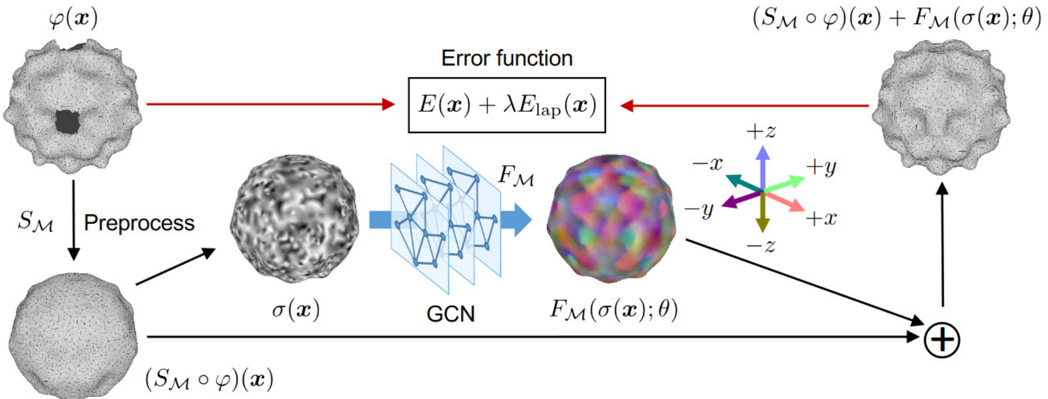


Figura 5.4: Funzionamento del modello di *Deep Mesh Prior*.

Dove:

- $\boxed{\varphi(x)}$ rappresenta la posizione dei vertici della *mesh* rumorosa. Nel caso di una *mesh* in cui sono presenti buchi, questi vengono riempiti in maniera tale da avere topologia uguale a quella della *mesh* che ci si aspetta in output.

- $\boxed{S_{\mathcal{M}}}$ è l'operazione di *smoothing*, in questo caso viene utilizzato uno *smooth laplaciano*.

- $\boxed{(S_{\mathcal{M}} \circ \varphi)(x)}$ sono le posizioni dei vertici della *mesh* in seguito all'operazione di *smoothing*.

- $\boxed{\sigma(x)}$ invece è l'insieme dei valori inizializzati casualmente che il modello utilizzerà come *displacement map* iniziale da utilizzare come input da fornire alla *GCN*.

- $\boxed{F_{\mathcal{M}}(\sigma(x);\theta)}$ rappresenta il risultato della funzione di *mapping* $F_{\mathcal{M}}$ definita dalla *GCN* a partire da un set di parametri θ.
- $\boxed{(S_{\mathcal{M}} \circ \varphi)(x) + F_{\mathcal{M}}(\sigma(x);\theta)}$ è la somma fra la *mesh* risultato dello *smooth* e la *displacement map* ottenuta come output dalla *GCN*.
- $\boxed{E(x) + \lambda E_{lap}(x)}$ la *GCN* è addestrata per rendere minimo l'errore fra le mesh di input e di output.

$E(x)$ è definito come segue:

$$\boxed{E(x) = ||\varphi(x) - \varphi'(x)||}$$

Dove $\varphi'(x) = (S_{\mathcal{M}} \circ \varphi)(x) + F_{\mathcal{M}}(\sigma(x);\theta)$

$E_{lap}(x)$ viene calcolato nel seguente modo:

$$\boxed{E_{lap}(x) = \left\| |\mathcal{N}(x)|\varphi'(x) - \sum_{y \in \mathcal{N}(x)} \varphi'(y) \right\|}$$

dove $\mathcal{N}(x)$ è un insieme di vertici vicini sulla *mesh*.

La *GCN* è quindi addestrata a risolvere:

$$\boxed{\theta^* = \arg\min_{\theta} \sum_{x \in \mathcal{M}} (E(x) + \lambda E_{lap}(x))}$$

dove λ è un parametro utilizzato per bilanciare l'errore di ricostruzione ed il *Laplacian loss*.

L'output finale prodotto dalla *GCN* viene confrontato con la *mesh* di *groundtruth* attraverso una metrica dell'errore, la *Mean Angle Difference (MAD)*. Si calcola confrontando per ogni faccia della *mesh* di output la propria normale con quella della corrispondente faccia nella *mesh* di *groundtruth*, eseguendo poi una media delle differenze calcolate.
La *MAD* può essere calcolato come segue:

$$Err^2 = \frac{1}{F}\sum_{i=1}^{F}\langle n_i, \hat{n_i}\rangle$$

dove:

- F: numero di facce della *mesh*.
- n_i: normale alla faccia f_i della *mesh* di *groundtruth*.
- $\hat{n_i}$: normale alla faccia $\hat{f_i}$ della *mesh* di output del *denoising*.
- $\langle ... \rangle$: rappresenta il prodotto scalare fra le due normali. Misura quindi l'angolo fra di esse formato.

A valori minori del *MAD* corrispondono ricostruzioni più accurate che mantengono le normali coerenti con quelle di *groundtruth*.

Di seguito l'implementazione in `Python` per il calcolo del *MAD* fra due *mesh*:

```
1  import numpy as np
2  from util.models import Mesh
3
4  def mad(mesh1, mesh2):
5      fn1 = Mesh.compute_face_normals(mesh1)
6      fn2 = Mesh.compute_face_normals(mesh2)
7      inner = [np.inner(fn1[i], fn2[i]) for i in range(fn1
          .shape[0])]
8      sad = np.rad2deg(np.arccos(np.clip(inner, -1.0, 1.0)
          ))
```

```
9     mad = np.sum(sad) / len(sad)
10
11    return mad
```

5.2.1 Implementazione

DMP è stato implementato utilizzando:

- **Linguaggio**: `Python`.
- **Framework**: `PyTorch`, `PyTorch.Geometric`.

I moduli `.py` principali e che di seguito verranno mostrati sono:

- `networks.py`: Dove è contenuta l'architettura della *GCN*.
- `denoise.py`: Utilizzato per avviare l'addestramento della rete nel caso si voglia effettuare il *denoising* sulla *mesh* fornita in input.

networks.py

La rete è composta da 15 *layer*, ognuno con una specifica profondità.
Per le operazioni di convoluzione viene utilizzato l'operatore `GCNConv`.
Per l'implementazione: 5.5.1.

denoise.py

In questo caso abbiamo fornito in input la *mesh* "dragon".
L'addestramento prosegue per 5000 epoche.
Per l'implementazione: 5.5.2.

5.2.2 Risultati

Al termine delle 5000 epoche, il risultato del *denoising* eseguito sulla *mesh* "dragon" è il seguente:

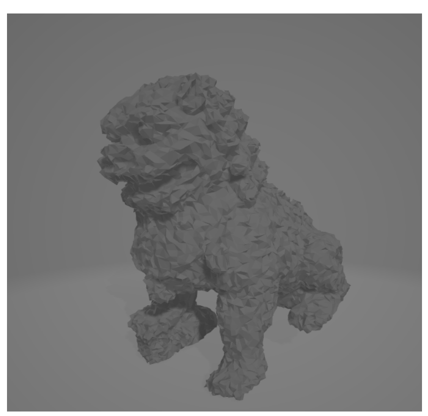


Figura 5.5: *Mesh* rumorosa di input.

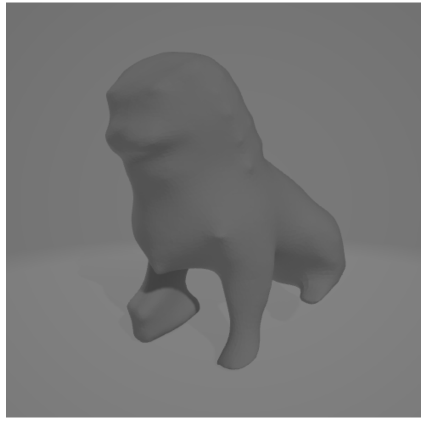


Figura 5.6: *Mesh* risultato dello *smoothing*.

Figura 5.7: *Mesh* di output risultato del *denoising*.

Figura 5.8: *Mesh* di *ground-truth*.

I risultati numerici sono i seguenti:

- `MAD` iniziale: 43.3320602
- `MAD` finale: 16.9128987

5.2.3 Esperimenti

Modificando l'operatore convoluzionale da `GCNConv` a `GATConv`, i risultati ottenuti sono invece insoddisfacenti. Questo perchè, non essendo presente

il *laplaciano* nella formulazione della *GAT*, il framework di *DMP* non è in grado di trovare ed identificare feature di *self-similarity*, soprattutto per l'assenza nel *GAT* dell'utilizzo della matrice di adiacenza, che permette una comprensione maggiore della struttura della *mesh*.

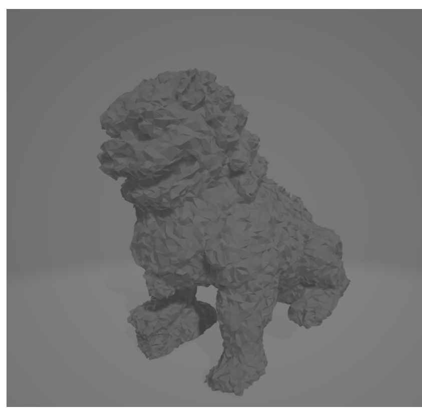


Figura 5.9: *Mesh* rumorosa di input.

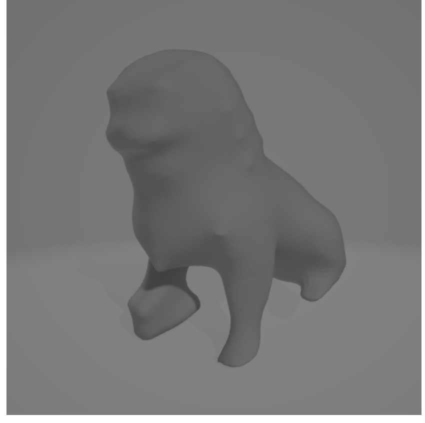


Figura 5.10: *Mesh* risultato dello *smoothing*.

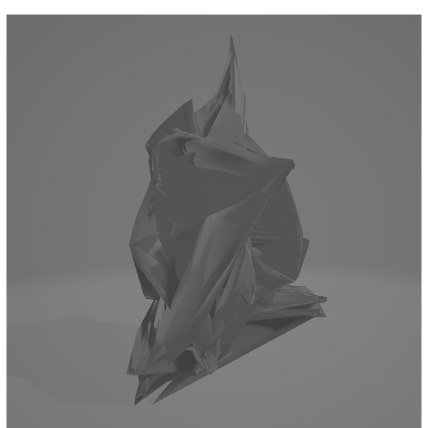


Figura 5.11: *Mesh* di output risultato del *denoising*.

Figura 5.12: *Mesh* di *ground-truth*.

I risultati numerici sono i seguenti:

- `MAD` iniziale: 43.3320602
- `MAD` finale: nan (dovuto all'estrema irregolarità della *mesh* di output)

5.2.4 Problematica

I risultati prodotti da questo metodo sono estremamente promettenti. Il problema primario sta nel fatto che questo non è un approccio che permette la creazione di un modello generalizzabile. La rete impara e si addestra su un singolo input, e produce un solo output. La rete finale non ha alcun utilizzo o applicazione al di fuori della *mesh* di cui si voleva fare il *denoising* o la *completion*.

5.3 Approccio supervisionato

Per lo sviluppo del modello supervisionato, è stato utilizzato sempre `Python` e come framework sia `PyTorch` che `PyTorch.Geometric`.
A causa di limitazioni computazionali il *training dataset* è composto da 24 *mesh*, il *validation dataset* da 4 *mesh* ed il *testing dataset* da 5 *mesh*.
Di seguito alcune delle *mesh* utilizzate in fase di *training*.

Figura 5.13: *Mesh* di un bassorilievo di un angelo utilizzata come *groundtruth*.

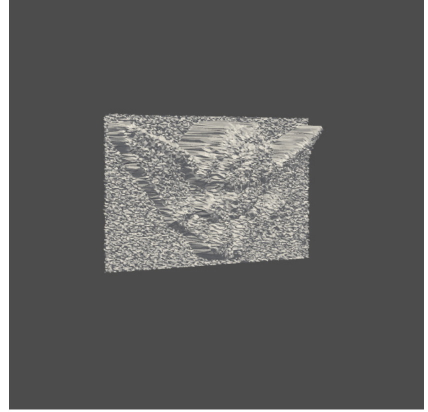


Figura 5.14: *Mesh* di un bassorilievo di un angelo affetto da rumore utilizzata come input.

Figura 5.15: *Mesh* di un gargoyle utilizzata come *groundtruth*.

Figura 5.16: *Mesh* di un gargoyle affetto da rumore utilizzata come input.

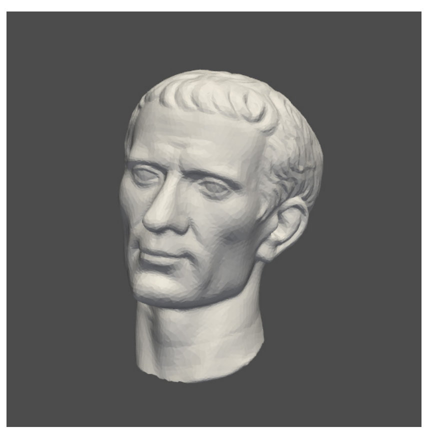


Figura 5.17: *Mesh* della faccia di Giulio Cesare utilizzata come *groundtruth*.

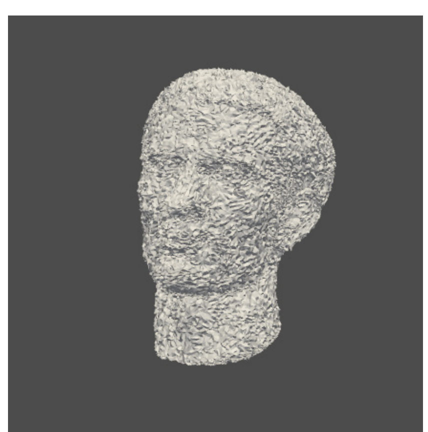


Figura 5.18: *Mesh* della faccia di Giulio Cesare affetta da rumore utilizzata come input.

Figura 5.19: *Mesh* di una mano utilizzata come *groundtruth*.

Figura 5.20: *Mesh* di una mano affetta da rumore utilizzata come input.

5.3.1 Implementazione

Durante la fase di *training*, necessitando di *mesh* affette da rumore su cui testare il modello, è stata implementata una funzione in grado di aggiungere artificialmente una determinata quantità di rumore alla *mesh* di *groundtruth* fornita. Questa funzione è la `mesh2Data` presente nel modulo `trainN.py`.
Ogni vertice della *mesh* originale viene scalato secondo un coefficiente risultato del prodotto fra il livello di rumore passato in input alla funzione ed uno scalare determinato casualmente. Questo valore viene poi moltiplicato per la normale presente in quel vertice, così da traslarlo nella stessa direzione.

```
newPointNoisy = newPointClean + np.random.randn(1, 3)*noiseLevel*normal
```

Nella nostra implementazione abbiamo impostato un *noise level* di 0.005.
La funzione di *loss* utilizzata in fase di *training* è la *Mean Squared Error (MSE)*. Ovvero:

$$\boxed{MSE = \frac{1}{N}\sum_{i=0}^{N}(y_i - \hat{y}_i)^2}$$

dove y_i rappresenta l'i-esima *mesh* di *groundtruth* mentre $\hat{y}_i$ è il corrispondente output della rete. N invece è il numero totale di *mesh* del *training set*. Per la fase di *backpropagation*, invece, è stato utilizzato l'ottimizzatore *Adam*, ovvero un algoritmo di ottimizzazione alternativo alla discesa stocastica del gradiente. *Adam* determina infatti degli *adaptive learning rates* specifici per ogni parametro del modello [12].
Nel nostro caso è stato deciso di impostare un *learning rate* iniziale di 0.0025.
I moduli `.py` di interesse sono tre:

- `networks.py`: contiene l'architettura della rete.
- `trainN.py`: contiene l'algoritmo di addestramento ed alcune funzioni utili alla generazione dei dataset.
- `inference.py`: contiene la fase di inference del modello prodotto dalla `trainN.py`.

networks.py

Per questo tipo di rete, è stato scelto di utilizzare 12 *layer*, questi ultimi di profondità ridotta rispetto alla struttura della rete utilizzata per *DMP*.
Per l'implementazione: 5.6.1.

trainN.py

Nella prima parte del modulo, nel caso non sia già stato fatto, vengono generati i dataset. Nella seconda metà invece avvengono le fasi di *training*, *validation* e *testing*.
L'addestramento, in questo caso, viene protratto per 201 epoche. Ogni 10 viene eseguita la fase di *validation* e calcolata la *loss* attuale. Se migliore rispetto alla migliore calcolata fino ad allora, si salva il modello ed il valore della *loss* dell'epoca attuale.
Per l'implementazione: 5.6.2.

inference.py

In questa modulo viene verificato e testato il funzionamento del modello generato dalla `trainN.py`.
Per l'implementazione: 5.6.3.

5.3.2 Risultati

Il funzionamento della rete è stato testato utilizzando sia l'operatore convoluzionale `GCNConv` sia `GATConv`, ottenendo risultati differenti.
Rispetto all'approccio non supervisionato, dove i risultati migliori sono stati ottenuti utilizzando la `GCNConv`, nel caso supervisionato, sono stati riscontrati risultati più accurati utilizzando la `GATConv`.
La differenza nell'efficacia dei due metodi è dovuta all'utilizzo della *matrice laplaciana*, che contiene la *matrice delle adiacenze*, nella `GCNConv`.
Questo permette un più alto grado di apprendimento per quanto riguarda le *feature* di *self-similarity* presenti nella *mesh* stessa, facendo però fatica a generalizzare tali informazioni in maniera da poter essere applicate a *mesh* con topologia e geometria anche molto differenti.

GCNConv

Di seguito le *mesh* di output ottenute della fase di *inference*:

Oggetto complesso originale

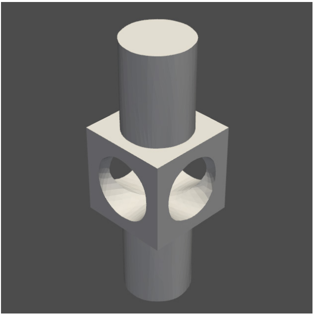


Figura 5.21: *Mesh* dell'oggetto complesso originale.

Oggetto complesso rumoroso

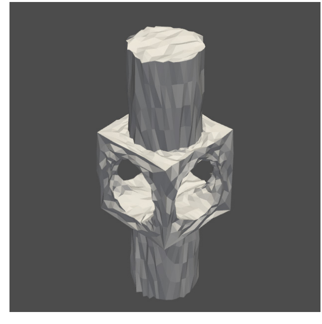


Figura 5.22: *Mesh* dell'oggetto complesso rumorosa.

Oggetto complesso output

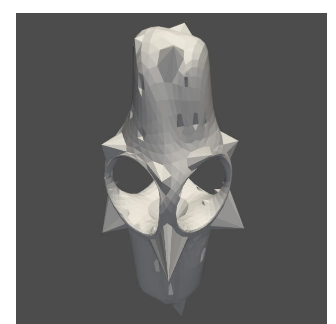


Figura 5.23: *Mesh* dell'oggetto complesso di output.

MAD iniziale *mesh* rumorosa: 21.9485704
MAD *mesh* di output con *GCNConv*: 33.7575704

Pompa dell'olio originale

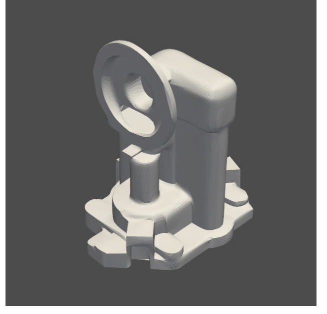


Figura 5.24: *Mesh* della pompa originale.

Pompa dell'olio rumorosa

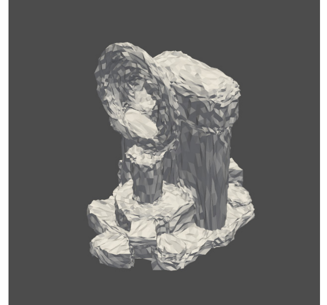


Figura 5.25: *Mesh* della pompa rumorosa.

Pompa dell'olio output

Figura 5.26: *Mesh* della pompa di output.

MAD iniziale *mesh* rumorosa: 31.9444668
MAD *mesh* di output con *GCNConv*: 29.2890290

Peluche originale

Figura 5.27: *Mesh* del peluche originale.

Peluche rumoroso

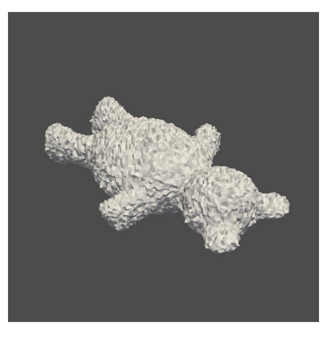


Figura 5.28: *Mesh* del peluche rumorosa.

Peluche output

Figura 5.29: *Mesh* del peluche di output.

MAD iniziale *mesh* rumorosa: 32.9576688
MAD *mesh* di output con *GCNConv*: 39.9913773

Sfera originale

Figura 5.30: *Mesh* della sfera originale.

Sfera rumorosa

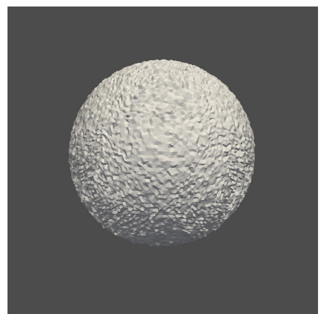


Figura 5.31: *Mesh* della sfera rumorosa.

Sfera output

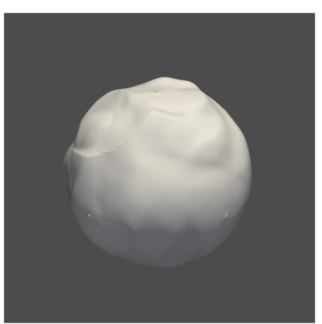


Figura 5.32: *Mesh* della sfera di output.

MAD iniziale *mesh* rumorosa: 26.0531445
MAD *mesh* di output con *GCNConv*: 6.3666344

GATConv

Dopo aver ri-addestrato il modello, questa volta con l'operatore convoluzionale *GATConv*, di seguito le *mesh* di output ottenute dalla nuova fase di *inference*:

Oggetto complesso originale

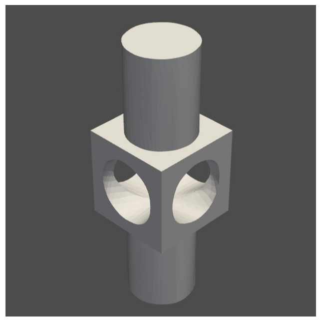


Figura 5.33: *Mesh* dell'oggetto complesso originale.

Oggetto complesso rumoroso

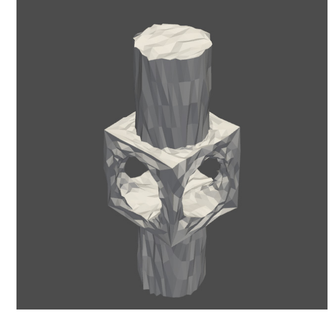


Figura 5.34: *Mesh* dell'oggetto complesso rumorosa.

Oggetto complesso output

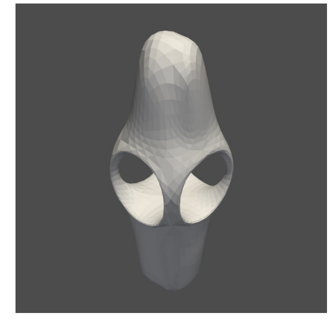


Figura 5.35: *Mesh* dell'oggetto complesso di output.

MAD iniziale *mesh* rumorosa: 21.9485704
MAD *mesh* di output con *GATConv*: 31.4856019

Pompa dell'olio originale

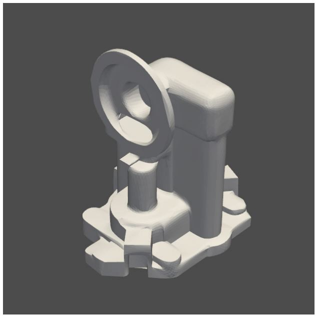


Figura 5.36: *Mesh* della pompa originale.

Pompa dell'olio rumorosa

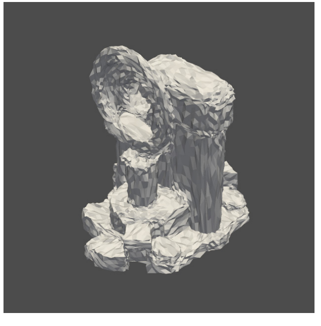


Figura 5.37: *Mesh* della pompa rumorosa.

Pompa dell'olio output

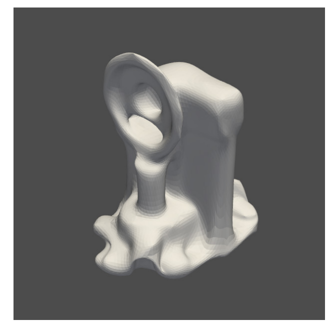


Figura 5.38: *Mesh* della pompa di output.

MAD iniziale *mesh* rumorosa: 31.9444668
MAD *mesh* di output con *GATConv*: 21.5410901

Peluche originale

Figura 5.39: *Mesh* del peluche originale.

Peluche rumoroso

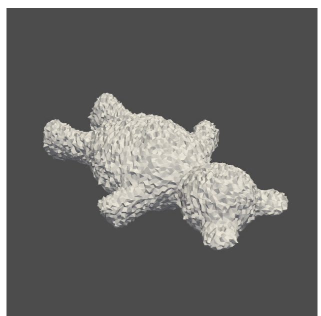


Figura 5.40: *Mesh* del peluche rumorosa.

Peluche output

Figura 5.41: *Mesh* del peluche di output.

MAD iniziale *mesh* rumorosa: 32.9576688
MAD *mesh* di output con *GATConv*: 5.3672317

Sfera originale

Figura 5.42: *Mesh* della sfera originale.

Sfera rumorosa

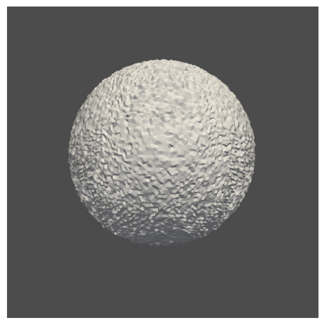


Figura 5.43: *Mesh* della sfera rumorosa.

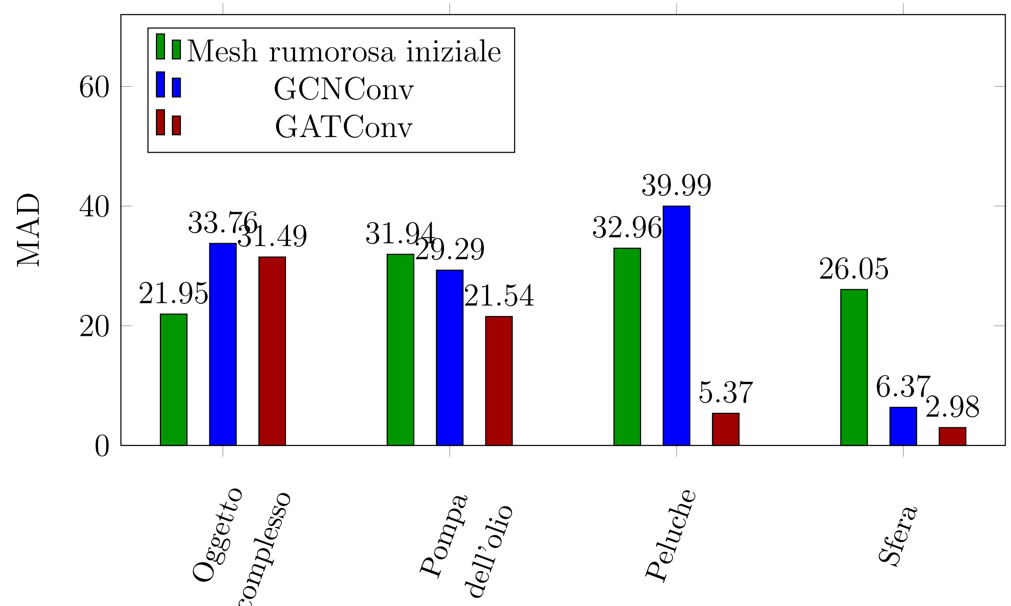
Sfera output

Figura 5.44: *Mesh* della sfera di output.

MAD iniziale *mesh* rumorosa: 26.0531445

MAD *mesh* di output con *GATConv*: 2.9809065

5.3.3 Confronto



Mesh	MAD mesh rumorosa	MAD GCNConv	MAD GATConv
Oggetto complesso	21.9485704	33.7575704	31.4856019
Pompa dell'olio	31.9444668	29.2890290	21.5410901
Peluche	32.9576688	39.9913773	5.3672317
Sfera	26.0531445	6.3666344	2.9809065

Come precedentemente anticipato, i risultati migliori sono stati ottenuti utilizzando l'operatore convoluzionale di *GATConv*.
Il caso dell' "Oggetto complesso" è interessante ed anomalo: la *mesh* rumorosa iniziale risulta infatti essere quella col valore di *MAD* minimo.
In tutti gli altri casi l'applicazione dell'operatore *GATConv* porta all'ottenimento di una *mesh* di output con un valore di *MAD* minimo rispetto agli alla *mesh* rumorosa ed alla *mesh* di output data dalla *GCNConv*.
Partendo da questi risultati, possiamo dire che la *GATConv* ottiene mediamente dei risultati del 43.24% migliori rispetto a quelli ottenuti con la *GCNConv*.

5.4 DMP e approccio non supervisionato: confronto diretto

Per effettuare un paragone diretto fra il *Deep Mesh Prior*, utilizzante `GCNConv` e l'approccio supervisionato utilizzante `GATConv` per valutarne efficacia, tempo di elaborazione e precisione eseguiamo il confronto utilizzando la stessa *mesh* in input. Il metodo supervisionato sarà addestrato a partire da un dataset di 34 *mesh*, nella fase di *inference* sarà poi utilizzata soltanto la stessa *mesh* fornita a *DMP*.
Inoltre, essendo due metodi estremamente differenti, risulta necessario trovare una metodologia per poterle mettere correttamente a confronto. Si è scelto di imporre il seguente vincolo, confrontando poi le prestazioni e risultati dei due metodi:

- Confronto a parità di epoche

Il confronto è stato eseguito utilizzando la seguente configurazione hardware:

- *CPU*: 12 Intel Core(TM) i7-9750H
- *Scheda Video*: NVIDIA GeForce GTX 1660Ti con Max-Q Design

Le *mesh* originale e rumorosa utilizzate per il confronto sono le seguenti(*Grayloc mesh*):

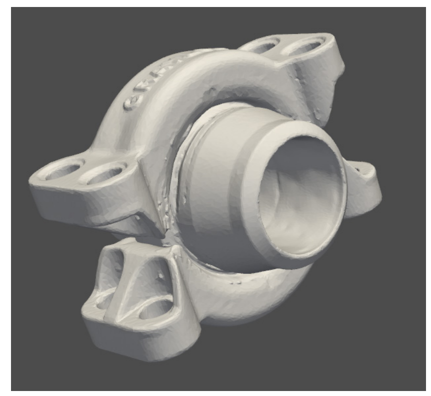


Figura 5.45: *Grayloc Mesh* originale utilizzata come *ground-truth*.

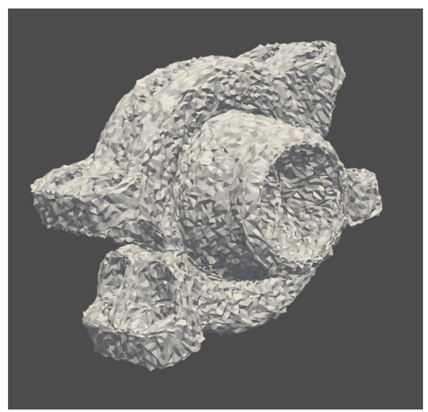


Figura 5.46: *Grayloc Mesh* rumorosa fornita in input.

5.4.1 Risultati

Di seguito sono messi a confronto i risultati numerici del *metodo supervisionato* e *DMP*:

Numero di epoche	MAD (supervisionato)	MAD (DMP)
200	9.9004784	13.7816579
500	9.7313256	9.7027045
1000	9.5620179	7.9934126
2000	9.4558426	7.8218710
5000	8.6083361	8.4261386

Interessante notare come nel caso di *DMP* con un numero di epoche di addestramento maggiore, il valore del *MAD* risulti peggiore rispetto ad output ottenuti con un numero di epoche inferiore. Questo risulta essere un problema di *overfitting*.
Come possiamo vedere dal seguente grafico, l'ottimo viene raggiunto tra la 1000-esima e la 2000-esima epoca.

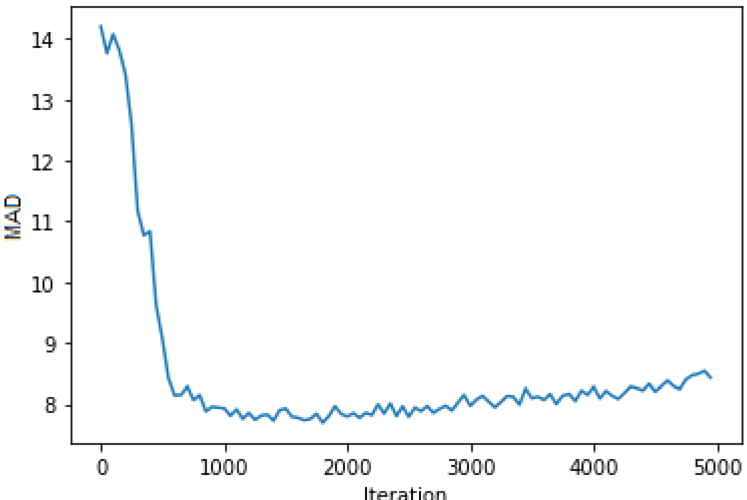


Figura 5.47: Grafico che mostra l'andamento del *MAD* per il *DMP* in base al numero di epoche trascorse.

I risultati grafici ottenuti in base al metodo utilizzato ed al numero di epoche trascorse sono i seguenti:

Numero di epoche	Mesh output (supervisionato)	Mesh output (DMP)
200		
500		
1000		
2000		
5000		

Da notare come il *DMP* sia maggiormente in grado di riconoscere che la scritta in bassorilievo non faccia parte del rumore ma sia una feature della *mesh* originale.

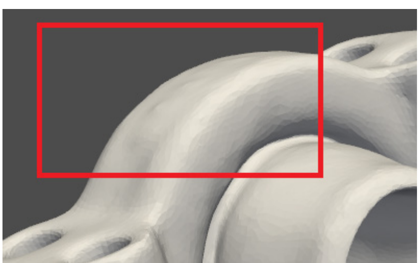


Figura 5.48: Zoom sul dettaglio della *Mesh* ottenuta dopo 5000 epoche di addestramento utilizzando l'approccio supervisionato.

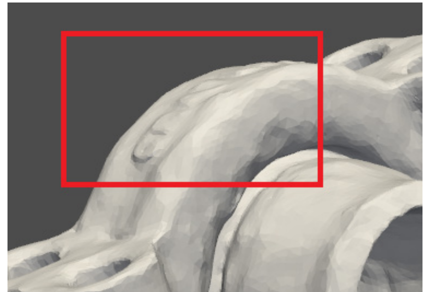


Figura 5.49: Zoom sul dettaglio della *Mesh* ottenuta dopo 5000 epoche di addestramento utilizzando *DMP*. Livello di accuratezza superiore.

Il metodo supervisionato invece tende a smussare tali dettagli.
Ricordiamo però che *DMP* produce un solo output, non si ottiene un modello generalizzabile. Questo significa che per eseguire il *denoising* su di un'altra *mesh*, sarebbe necessario ri-addestrare la rete sulla nuova *mesh*. Il metodo supervisionato, invece, avendo prodotto un modello di rete riutilizzabile, può effettuare il *denoising* di un numero arbitrario di *mesh*.

5.4.2 Ulteriori confronti

A partire da altre *mesh* è stata effettuata un'altra batteria di confronti fra i due metodi, in questo caso entrambi addestrati su 2000 epoche.

Bumpy Mesh

Originale

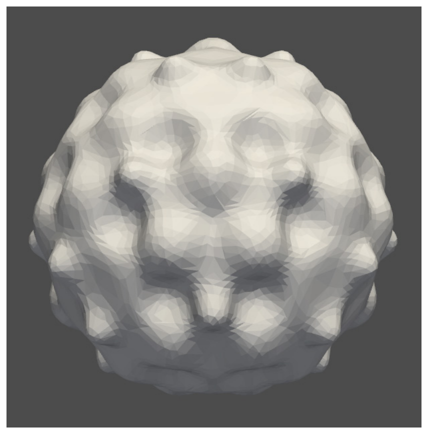


Figura 5.50: *Bumpy Mesh* originale.

Rumorosa

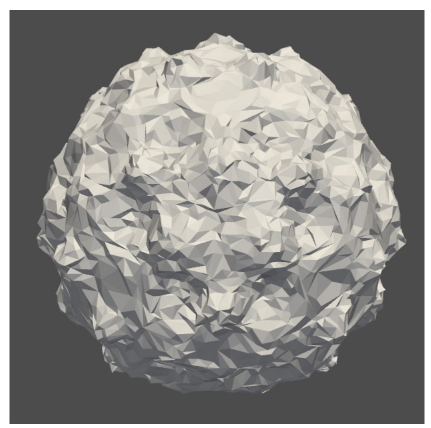


Figura 5.51: *Bumpy Mesh* rumorosa.

I risultati dei due metodi a confronto:

- *MAD con metodo supervisionato*: 11.99193901
- *MAD con DMP*: 7.04083234

Supervisionato

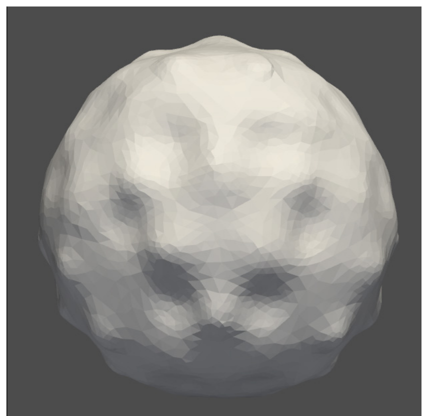


Figura 5.52: *Bumpy Mesh* risultato del *denoising* utilizzando il metodo supervisionato.

DMP

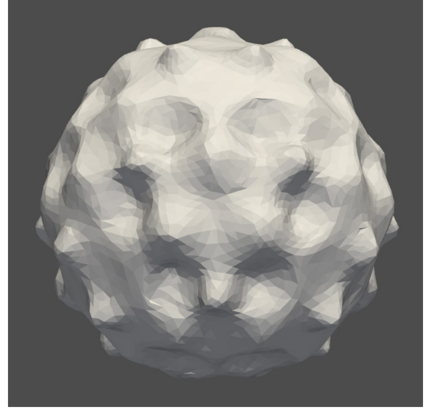


Figura 5.53: *Bumpy Mesh* risultato del *denoising* utilizzando *DMP*.

Carter Mesh

Originale

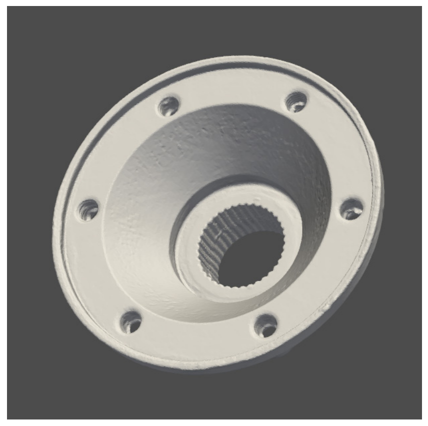


Figura 5.54: *Carter Mesh* originale.

Rumorosa

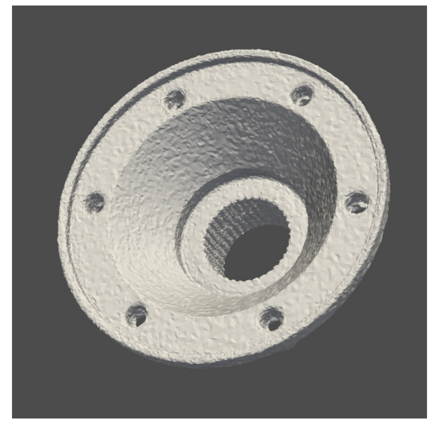


Figura 5.55: *Carter Mesh* rumorosa.

I risultati dei due metodi a confronto:

- *MAD con metodo supervisionato*: 9.6456375
- *MAD con DMP*: 5.8216894

Supervisionato

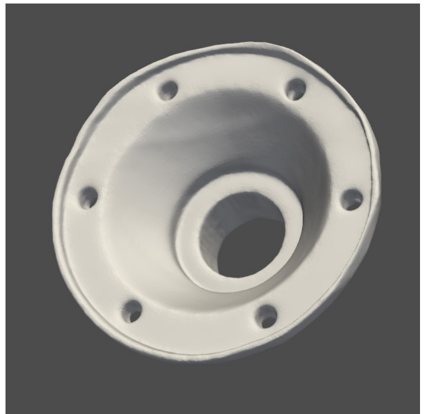


Figura 5.56: *Carter Mesh* risultato del *denoising* utilizzando il metodo supervisionato.

DMP

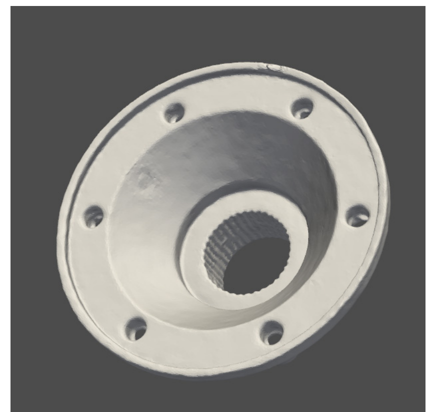


Figura 5.57: *Carter Mesh* risultato del *denoising* utilizzando *DMP*.

Sharp Mesh

Originale

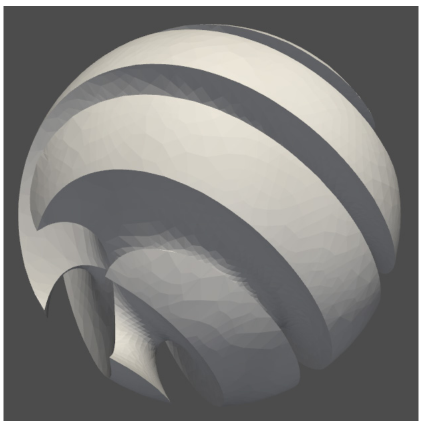


Figura 5.58: *Sharp Mesh* originale.

Rumorosa

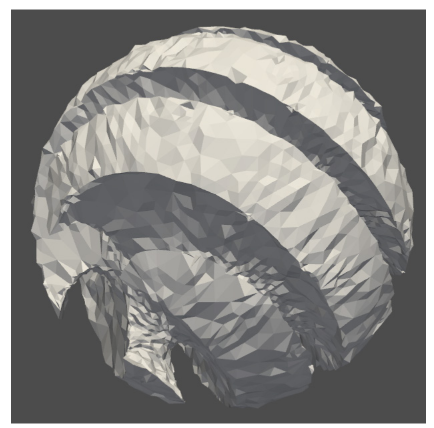


Figura 5.59: *Sharp Mesh* rumorosa.

I risultati dei due metodi a confronto:

- *MAD con metodo supervisionato*: 13.787774758
- *MAD con DMP*: 8.411802446

Supervisionato

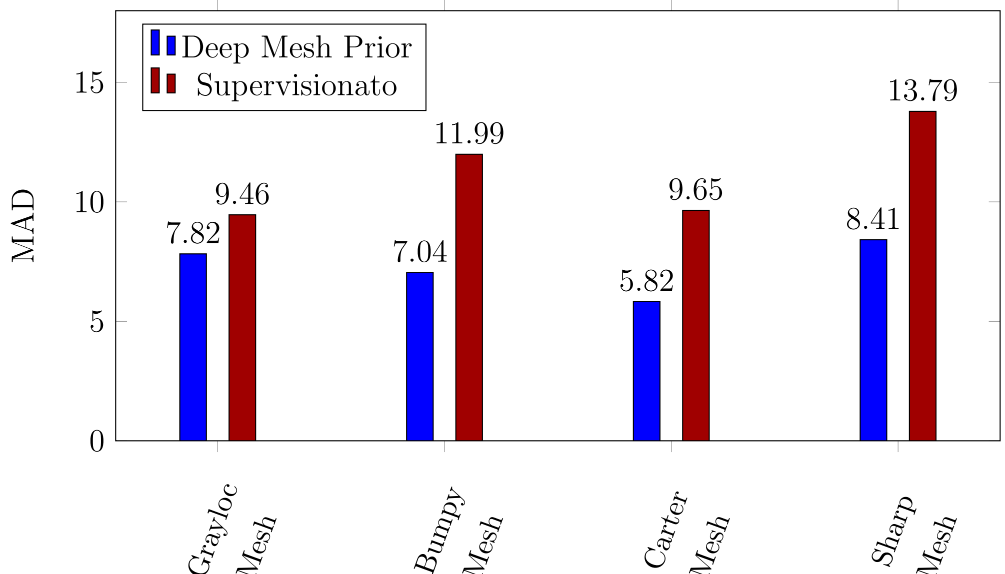
Figura 5.60: *Sharp Mesh* risultato del *denoising* utilizzando il metodo supervisionato.

DMP

Figura 5.61: *Sharp Mesh* risultato del *denoising* utilizzando *DMP*.

5.4.3 Analisi finale

Confrontando i due metodi, il *DMP* è quello ottenente i risultati più accurati. Ricordiamo però che le limitazioni riguardo la dimensione del *dataset* utilizzato nel metodo supervisionato sono da tenere in considerazione.
Possiamo comunque però affermare che sulla base degli esperimenti eseguiti, il *DMP* sia il metodo preferibile in termini di precisione ed accuratezza.



Conclusioni

Dopo aver confrontato un possibile approccio supervisionato ed uno non supervisionato per quanto riguarda il problema del *denoising* di *mesh* 3D utilizzando il *Geometric Deep Learning*, è possibile fare una riflessione.
I due metodi forniscono dei risultati paragonabili e non troppo distanti fra loro in termini di precisione nell'azione di *denoising* sulle *mesh*, mantenendone il più possibile invariate le caratteristiche originali.
Un approccio supervisionato per affrontare il *denoising* sembra ottenere risultati buoni, nonostante il *dataset* di partenza abbia dimensione ridotta, a causa delle limitazioni computazionali dell'hardware su cui è stato testato. Sarebbe stato interessante verificare quanto avrebbe potuto migliorare la precisione di questo approccio avendo avuto a disposizione un *dataset* più ampio.
Per quanto riguarda il metodo non supervisionato *Deep Mesh Prior*, è importante ribadire quanto questo metodo riesca ad ottenere dei miglioramenti in termini di precisione lavorando sulla *mesh* di input. Il tempo per addestrare questo tipo di rete risulta molto maggiore (3-4 volte superiore a parità di epoche) a causa della maggiore profondità della rete e numero di nodi per *layer*.
I due metodi possono però essere utilizzati in situazioni differenti, senza necessità di trattarli in maniera esclusiva.
DMP potrebbe essere utilizzato nel caso di lavori su un numero limitato di *mesh*, per le quali si vuole ottenere un livello superiore di accuratezza, senza badare a tempistiche e costi computazionali. L'utilizzo di *DMP* è sconsiglia-

to nel caso si voglia eseguire il *denoising* di centinaia di *mesh* differenti.
L'approccio supervisionato invece, risulta il giusto compromesso fra riutilizzabilità, precisione e tempo di addestramento.
Ricerche future si potrebbero focalizzare sullo sviluppo di un approccio semi-supervisionato, in grado di mediare accuratezza e generalizzabilità.

Appendice

5.5 Deep Mesh Prior

5.5.1 networks.py

```
import torch
import torch.nn as nn
import numpy as np
from torch_geometric.nn import GCNConv

class Net(nn.Module):
    def __init__(self, flags):
        super(Net, self).__init__()
        self.device = torch.device('cuda' if torch.cuda.
            is_available() else 'cpu')
        self.flags = flags
        h = [16, 32, 64, 128, 256, 256, 512, 512, 256,
            256, 128, 64, 32, 32, 3]
        if self.flags:
            # skip net
            self.conv1  = GCNConv(h[0], h[1])
            self.conv2  = GCNConv(h[1], h[2])
            self.conv3  = GCNConv(h[2], h[3])
            self.conv4  = GCNConv(h[3], h[4])
            self.conv5  = GCNConv(h[4], h[5])
            self.conv6  = GCNConv(h[5], h[6])
```

```
20            self.conv7   = GCNConv(h[6], h[7])
21            self.conv8   = GCNConv(h[7]+h[6], h[8])
22            self.conv9   = GCNConv(h[8]+h[5], h[9])
23            self.conv10  = GCNConv(h[9]+h[4], h[10])
24            self.conv11  = GCNConv(h[10]+h[3], h[11])
25            self.conv12  = GCNConv(h[11]+h[2], h[12])
26            self.conv13  = GCNConv(h[12]+h[1], h[13])
27            self.linear1 = nn.Linear(h[13], h[14])
28
29            self.bn1 = nn.BatchNorm1d(h[1])
30            self.bn2 = nn.BatchNorm1d(h[2])
31            self.bn3 = nn.BatchNorm1d(h[3])
32            self.bn4 = nn.BatchNorm1d(h[4])
33            self.bn5 = nn.BatchNorm1d(h[5])
34            self.bn6 = nn.BatchNorm1d(h[6])
35            self.bn7 = nn.BatchNorm1d(h[7])
36            self.bn8 = nn.BatchNorm1d(h[8])
37            self.bn9 = nn.BatchNorm1d(h[9])
38            self.bn10 = nn.BatchNorm1d(h[10])
39            self.bn11 = nn.BatchNorm1d(h[11])
40            self.bn12 = nn.BatchNorm1d(h[12])
41            self.bn13 = nn.BatchNorm1d(h[13])
42            self.l_relu = nn.LeakyReLU()
43
44        else:
45            # normal net
46            h = [16, 32, 64, 128, 256, 256, 512, 512,
                 256, 256, 128, 64, 32, 32, 16, 3]
47            self.conv1   = GCNConv(h[0], h[1])
48            self.conv2   = GCNConv(h[1], h[2])
49            self.conv3   = GCNConv(h[2], h[3])
50            self.conv4   = GCNConv(h[3], h[4])
51            self.conv5   = GCNConv(h[4], h[5])
```

```
            self.conv6  = GCNConv(h[5], h[6])
            self.conv7  = GCNConv(h[6], h[7])
            self.conv8  = GCNConv(h[7], h[8])
            self.conv9  = GCNConv(h[8], h[9])
            self.conv10 = GCNConv(h[9], h[10])
            self.conv11 = GCNConv(h[10], h[11])
            self.conv12 = GCNConv(h[11], h[12])
            self.conv13 = GCNConv(h[12], h[13])
            self.linear1 = nn.Linear(h[13], h[14])
            self.linear2 = nn.Linear(h[14], h[15])

            self.bn1 = nn.BatchNorm1d(h[1])
            self.bn2 = nn.BatchNorm1d(h[2])
            self.bn3 = nn.BatchNorm1d(h[3])
            self.bn4 = nn.BatchNorm1d(h[4])
            self.bn5 = nn.BatchNorm1d(h[5])
            self.bn6 = nn.BatchNorm1d(h[6])
            self.bn7 = nn.BatchNorm1d(h[7])
            self.bn8 = nn.BatchNorm1d(h[8])
            self.bn9 = nn.BatchNorm1d(h[9])
            self.bn10 = nn.BatchNorm1d(h[10])
            self.bn11 = nn.BatchNorm1d(h[11])
            self.bn12 = nn.BatchNorm1d(h[12])
            self.bn13 = nn.BatchNorm1d(h[13])
            self.l_relu = nn.LeakyReLU()


    def forward(self, data):
        x = np.random.normal(0, 0.1, size=(data.x.shape
            [0], 3))
        x, edge_index = data.x.to(self.device), data.
            edge_index.to(self.device)

```

```
83         if self.flags:
84             # skip net
85             dx = self.conv1(x, edge_index)
86             dx = self.bn1(dx)
87             dx = self.l_relu(dx)
88             skip1 = dx
89 
90             dx = self.conv2(dx, edge_index)
91             dx = self.bn2(dx)
92             dx = self.l_relu(dx)
93             skip2 = dx
94 
95             dx = self.conv3(dx, edge_index)
96             dx = self.bn3(dx)
97             dx = self.l_relu(dx)
98             skip3 = dx
99 
100            dx = self.conv4(dx, edge_index)
101            dx = self.bn4(dx)
102            dx = self.l_relu(dx)
103            skip4 = dx
104 
105            dx = self.conv5(dx, edge_index)
106            dx = self.bn5(dx)
107            dx = self.l_relu(dx)
108            skip5 = dx
109 
110            dx = self.conv6(dx, edge_index)
111            dx = self.bn6(dx)
112            dx = self.l_relu(dx)
113            skip6 = dx
114 
115            dx = self.conv7(dx, edge_index)
```

```
            dx = self.bn7(dx)
            dx = self.l_relu(dx)

            dx = torch.cat([dx, skip6], dim=1)
            dx = self.conv8(dx, edge_index)
            dx = self.bn8(dx)
            dx = self.l_relu(dx)

            dx = torch.cat([dx, skip5], dim=1)
            dx = self.conv9(dx, edge_index)
            dx = self.bn9(dx)
            dx = self.l_relu(dx)

            dx = torch.cat([dx, skip4], dim=1)
            dx = self.conv10(dx, edge_index)
            dx = self.bn10(dx)
            dx = self.l_relu(dx)

            dx = torch.cat([dx, skip3], dim=1)
            dx = self.conv11(dx, edge_index)
            dx = self.bn11(dx)
            dx = self.l_relu(dx)

            dx = torch.cat([dx, skip2], dim=1)
            dx = self.conv12(dx, edge_index)
            dx = self.bn12(dx)
            dx = self.l_relu(dx)

            dx = torch.cat([dx, skip1], dim=1)
            dx = self.conv13(dx, edge_index)
            dx = self.bn13(dx)
            dx = self.l_relu(dx)
            dx = self.linear1(dx)
```

```
149         else:
150             # normal net
151             dx = self.conv1(x, edge_index)
152             dx = self.bn1(dx)
153             dx = self.l_relu(dx)
154 
155             dx = self.conv2(dx, edge_index)
156             dx = self.bn2(dx)
157             dx = self.l_relu(dx)
158 
159             dx = self.conv3(dx, edge_index)
160             dx = self.bn3(dx)
161             dx = self.l_relu(dx)
162 
163             dx = self.conv4(dx, edge_index)
164             dx = self.bn4(dx)
165             dx = self.l_relu(dx)
166 
167             dx = self.conv5(dx, edge_index)
168             dx = self.bn5(dx)
169             dx = self.l_relu(dx)
170 
171             dx = self.conv6(dx, edge_index)
172             dx = self.bn6(dx)
173             dx = self.l_relu(dx)
174 
175             dx = self.conv7(dx, edge_index)
176             dx = self.bn7(dx)
177             dx = self.l_relu(dx)
178 
179             dx = self.conv8(dx, edge_index)
180             dx = self.bn8(dx)
181             dx = self.l_relu(dx)
```

```
            dx = self.conv9(dx, edge_index)
            dx = self.bn9(dx)
            dx = self.l_relu(dx)

            dx = self.conv10(dx, edge_index)
            dx = self.bn10(dx)
            dx = self.l_relu(dx)

            dx = self.conv11(dx, edge_index)
            dx = self.bn11(dx)
            dx = self.l_relu(dx)

            dx = self.conv12(dx, edge_index)
            dx = self.bn12(dx)
            dx = self.l_relu(dx)

            dx = self.conv13(dx, edge_index)
            dx = self.bn13(dx)
            dx = self.l_relu(dx)

            dx = self.linear1(dx)
            dx = self.l_relu(dx)
            dx = self.linear2(dx)

        return dx
```

5.5.2 denoise.py

```
import numpy as np

import torch
import copy
```

```
5  import datetime
6  import os
7  import sys
8  import glob
9  import argparse
10 import json
11 from util.objmesh import ObjMesh
12 from util.models import Dataset, Mesh
13 from util.networks import Net
14 import util.loss as Loss
15 
16 #from torch.utils.tensorboard import SummaryWriter
17 from torch_geometric.data import Data
18 
19 mesh_name = 'dragon'
20 
21 parser = argparse.ArgumentParser(description='Deep mesh
      prior for denoising')
22 #parser.add_argument('-i', '--input', type=str, required
      =True)
23 parser.add_argument('-i', '--input', type=str, default='
      datasets/d_input/'+mesh_name) #funge da percorso per
      la cartella giusta
24 parser.add_argument('--lr', type=float, default=0.01)
25 parser.add_argument('--iter', type=int, default=5000)
26 parser.add_argument('--skip', type=bool, default=False)
27 parser.add_argument('--lap', type=float, default=1.4)
28 FLAGS = parser.parse_args()
29 
30 for k, v in vars(FLAGS).items():
31     print('{:10s}: {}'.format(k, v))
32 
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


file_path = FLAGS.input
file_list = glob.glob(file_path + '/*.obj')
file_list = sorted(file_list) #perch dentro c'  la smooth e noise nella cartella di nome file_name

input_file = file_list[1] #smoothed mesh
label_file = file_list[0] #noisy mesh
gt_file = 'datasets/groundtruth/' + mesh_name + '.obj'

l_mesh = Mesh(label_file)
i_mesh = Mesh(input_file)
g_mesh = Mesh(gt_file)

# node-features and edge-index
np.random.seed(42)
x = np.random.normal(size=(l_mesh.vs.shape[0], 16))
x = torch.tensor(x, dtype=torch.float, requires_grad=True)
x_pos = torch.tensor(i_mesh.vs, dtype=torch.float)
y = torch.tensor(l_mesh.vs, dtype=torch.float)

edge_index = torch.tensor(l_mesh.edges.T, dtype=torch.long)
edge_index = torch.cat([edge_index, edge_index[[1,0],:]], dim=1)
init_mad = Loss.mad(l_mesh, g_mesh)

data = Data(x=x, y=y, x_pos=x_pos, edge_index=edge_index)
```

```
60 dataset = Dataset(data)
61 dataset.x = dataset.x.to(device)
62 dataset.y = dataset.y.to(device)
63 dataset.edge_index = dataset.edge_index.to(device)
64
65 # create model instance
66 model = Net(FLAGS.skip).to(device)
67 model.train()
68
69 # learning loop
70 min_mad = 1000
71 print("initial_mad_value: ", init_mad)
72 out_dir='datasets/d_output'
73 optimizer = torch.optim.Adam(model.parameters(), lr=
      FLAGS.lr)
74
75
76 for epoch in range(1, FLAGS.iter+1):
77     optimizer.zero_grad()
78     out = model(dataset)
79     loss1 = Loss.mse_loss(out, dataset.y)
80     loss2 = Loss.mesh_laplacian_loss(out, l_mesh.ve,
          l_mesh.edges)
81     loss = loss1 + FLAGS.lap * loss2
82     loss.backward()
83     optimizer.step()
84
85     if epoch % 10 == 0:
86         print('Epoch %d | Loss: %.4f' % (epoch, loss.
              item()))
87     if epoch % 50 == 0:
88         o_mesh = ObjMesh(input_file)
```

```
89            o_mesh.vs = o_mesh.vertices = out.to('cpu').
                  detach().numpy().copy()
90            o_mesh.faces = i_mesh.faces
91            o_mesh.save(out_dir + '/' + str(epoch) + '_' +
                  mesh_name + '_output.obj')
92            mad_value = Loss.mad(o_mesh, g_mesh)
93            vertex_difference = Loss.housdorff_dist(o_mesh,
                  g_mesh)
94            print("haus_dist: ", vertex_difference)
95            min_mad = min(mad_value, min_mad)
96            print("mad_value: ", mad_value, "min_mad: ",
                  min_mad)
97            #writer.add_scalar("mean_angle_difference",
                  mad_value, epoch)
98            print("mean_angle_difference", mad_value, "Epoca
                  ", epoch)
99
100 #writer.close()
```

5.6 Approccio supervisionato

5.6.1 networks.py

```
1  import torch
2  import torch.nn as nn
3  from torch_geometric.nn import GCNConv, GATConv
4  from torch.nn import ModuleList, ReLU
5
6  class Net(nn.Module):
7      def __init__(self, flags):
8          super(Net, self).__init__()
9          self.device = torch.device('cuda' if torch.cuda.
               is_available() else 'cpu')
```

```
10          self.flags = flags
11          h = [3, 6, 8, 16, 16, 32, 32, 16, 16, 8, 6, 3]
12          if self.flags:
13              # skip net
14              self.conv1   = GCNConv(h[0], h[1])
15              self.conv2   = GCNConv(h[1], h[2])
16              self.conv3   = GCNConv(h[2], h[3])
17              self.conv4   = GCNConv(h[3], h[4])
18              self.conv5   = GCNConv(h[4], h[5])
19              self.conv6   = GCNConv(h[5], h[6])
20              self.conv7   = GCNConv(h[5]+h[6],h[7])
21              self.conv8   = GCNConv(h[7]+h[4],h[8])
22              self.conv9   = GCNConv(h[8]+h[3],h[9])
23              self.conv10   = GCNConv(h[9]+h[2],h[10])
24              self.conv11   = GCNConv(h[10],h[11])
25
26              #self.linear1 = nn.Linear(h[10], h[11])
27
28              self.bn1 = nn.BatchNorm1d(h[1])
29              self.bn2 = nn.BatchNorm1d(h[2])
30              self.bn3 = nn.BatchNorm1d(h[3])
31              self.bn4 = nn.BatchNorm1d(h[4])
32              self.bn5 = nn.BatchNorm1d(h[5])
33              self.bn6 = nn.BatchNorm1d(h[6])
34              self.bn7 = nn.BatchNorm1d(h[7])
35              self.bn8 = nn.BatchNorm1d(h[8])
36              self.bn9 = nn.BatchNorm1d(h[9])
37              self.bn10 = nn.BatchNorm1d(h[10])
38
39
40              self.l_relu = nn.LeakyReLU()
41              #self.l_relu=ReLU(inplace=True)
42          else:
```

```
43              # normal net
44              h = [3, 6, 8, 16, 16, 32, 32, 16, 16, 8, 6,
                    3]
45              self.conv1  = GCNConv(h[0], h[1])
46              self.conv2  = GCNConv(h[1], h[2])
47              self.conv3  = GCNConv(h[2], h[3])
48              self.conv4  = GCNConv(h[3], h[4])
49              self.conv5  = GCNConv(h[4], h[5])
50              self.conv6  = GCNConv(h[5], h[6])
51              self.conv7  = GCNConv(h[6], h[7])
52              self.conv8  = GCNConv(h[7], h[8])
53              self.conv9  = GCNConv(h[8], h[9])
54              self.conv10  = GCNConv(h[9], h[10])
55              self.linear1 = nn.Linear(h[10], h[11])
56
57
58              self.l_relu = nn.LeakyReLU()
59              #self.l_relu=ReLU(inplace=True)
60      def forward(self, data):
61
62          x, edge_index = data.x.to(self.device), data.
               edge_index.to(self.device)
63
64          if self.flags:
65              # skip net
66
67              dx = self.conv1(x, edge_index)
68              #dx = self.bn1(dx)
69              dx = self.l_relu(dx)
70              skip1=dx
71
72              dx = self.conv2(dx, edge_index)
73              #dx = self.bn2(dx)
```

```
74            dx = self.l_relu(dx)
75            skip2=dx
76
77            dx = self.conv3(dx, edge_index)
78            #dx = self.bn3(dx)
79            dx = self.l_relu(dx)
80            skip3=dx
81
82            dx = self.conv4(dx, edge_index)
83            #dx = self.bn4(dx)
84            dx = self.l_relu(dx)
85
86            skip4=dx
87
88            dx = self.conv5(dx, edge_index)
89            #dx = self.bn5(dx)
90            dx = self.l_relu(dx)
91
92            skip5=dx
93
94
95            dx = self.conv6(dx, edge_index)
96            #dx = self.bn6(dx)
97
98            dx = self.l_relu(dx)
99
100
101           dx = torch.cat([dx, skip5], dim=1)
102
103           dx = self.conv7(dx, edge_index)
104           #dx = self.bn7(dx)
105           dx = self.l_relu(dx)
106
```

```
107                 dx = torch.cat([dx, skip4], dim=1)
108
109                 dx = self.conv8(dx, edge_index)
110                 #dx = self.bn8(dx)
111                 dx = self.l_relu(dx)
112
113
114                 dx = torch.cat([dx, skip3], dim=1)
115
116                 dx = self.conv9(dx, edge_index)
117                 #dx = self.bn9(dx)
118                 dx = self.l_relu(dx)
119
120                 dx = torch.cat([dx, skip2], dim=1)
121
122                 dx = self.conv10(dx, edge_index)
123                 #dx = self.bn10(dx)
124                 dx = self.l_relu(dx)
125
126                 #dx = self.conv11(dx,edge_index)
127                 dx=self.linear1(dx)
128
129
130
131
132             else:
133                 # normal net
134                 dx = self.conv1(x, edge_index)
135                 dx = self.l_relu(dx)
136
137
138                 dx = self.conv2(dx, edge_index)
139                 dx = self.l_relu(dx)
```

```
140
141
142            dx = self.conv3(dx, edge_index)
143            dx = self.l_relu(dx)
144
145
146            dx = self.conv4(dx, edge_index)
147            dx = self.l_relu(dx)
148
149
150            dx = self.conv5(dx, edge_index)
151            dx = self.l_relu(dx)
152
153
154            dx = self.conv6(dx, edge_index)
155            dx = self.l_relu(dx)
156
157            dx = self.conv7(dx, edge_index)
158            dx = self.l_relu(dx)
159
160            dx = self.conv8(dx, edge_index)
161            dx = self.l_relu(dx)
162
163            dx = self.conv9(dx, edge_index)
164            dx = self.l_relu(dx)
165
166
167            dx = self.conv10(dx, edge_index)
168            dx = self.l_relu(dx)
169
170            dx = self.linear1(dx)
171
172
```

```
173
174             return dx
175
176     class GAT_Simple(torch.nn.Module):
177         def __init__(self, in_channels: int, hidden_channels
                : int, output_channels: int, num_layers: int):
178             super().__init__()
179             self.device = 'cpu'#torch.device('cuda' if torch
                   .cuda.is_available() else 'cpu')
180             self.in_channels = in_channels
181             self.hidden_channels = hidden_channels
182             self.out_channels = output_channels
183             self.num_layers = num_layers
184             self.act = ReLU(inplace=True)
185
186             self.conv1 = GATConv(in_channels,
                   hidden_channels)
187             self.convs = ModuleList()
188             for _ in range(num_layers - 2):
189                 self.convs.append(
190                 GATConv(hidden_channels, hidden_channels))
191             self.lastConv = GATConv(hidden_channels,
                   output_channels)
192
193         def forward(self, data):
194             x, edge_index = data.x, data.edge_index
195
196
197
198             x = self.conv1(x, edge_index)
199             x = self.act(x)
200
201             for i in range(self.num_layers-2):
```

```
202            x = self.convs[i](x,edge_index)
203            x = self.act(x)
204
205        x = self.lastConv(x, edge_index)
206        # x = self.act(x)
207
208
209        return x
```

5.6.2 trainN.py

```
1  import os
2  import pickle
3  import torch
4  import numpy as np
5  import pyvista as pv
6  from mygatunet import *
7  from torch_geometric.data import Data
8  from torch_geometric.loader import DataLoader
9  from torch_geometric.nn import GCNConv, GMMConv, GATConv
10 from torch_geometric.nn import GAT, PNA, GIN, GCN
11 from torch_geometric.utils import degree
12 from torch.nn import ModuleList, ReLU
13 from networks import GAT_Simple, Net
14
15 import torch.nn.functional as F
16 from datetime import datetime as dt
17
18 np.random.seed(1)
19 #device = torch.device('cuda')
20 device = torch.device('cuda')
21
22 data_dir = "./graphset"
```

```



def mesh2Data(filename, noise_level):
    orig_mesh = pv.read(filename)

    # Centered and Scaled mesh
    scaled_orig = orig_mesh.copy()

    # Perturbed mesh
    new_mesh = orig_mesh.copy()

    points = orig_mesh.points #tensor of vertex coordinates
    normals = orig_mesh.point_normals # Normals of the wrong mesh?
    center = orig_mesh.center
    bounds = orig_mesh.bounds

    # Scale by the maximum range in x,y,z to preserve shape and avoid distortion
    bound_div = max([bounds[1]-bounds[0], bounds[3]-bounds[2], bounds[5]-bounds[4]])
    #  print(bound_div)

    new_points_noisy = []
    new_points_clean = []

    for point, normal in zip(points, normals):
        # Center and scale original mesh
        new_point_clean = (point - center)/bound_div
        # print(new_point_clean)
        # Perturb the scaled mesh
```

```
52          new_point_noisy = new_point_clean + np.random.
               randn(1, 3)*noise_level*normal #NOISE_LEVEL =
                0.005
53          # print(new_point_noisy)
54          new_points_noisy.append(new_point_noisy[0].
               tolist())
55          new_points_clean.append(new_point_clean.tolist()
               )
56
57      scaled_orig.points = np.asarray(new_points_clean)
58      new_mesh.points = np.asarray(new_points_noisy)
59
60      n_faces = new_mesh.n_faces
61      n_points = new_mesh.n_points
62      print('n_points: {}'.format(n_points))
63      print('n_faces: {}'.format(n_faces))
64      faces = new_mesh.faces.reshape(-1, 4)[:, 1:4]
65
66      TF = np.zeros([6*n_faces, 1])
67      TS = np.zeros([6*n_faces, 1])
68
69      ind = 0
70
71      for face in faces:
72          TF[ind] = face[0]
73          TS[ind] = face[1]
74          TF[ind+1] = face[1]
75          TS[ind+1] = face[0]
76
77          TF[ind+2] = face[0]
78          TS[ind+2] = face[2]
79          TF[ind+3] = face[2]
80          TS[ind+3] = face[0]
```

```
81
82          TF[ind+4] = face[1]
83          TS[ind+4] = face[2]
84          TF[ind+5] = face[2]
85          TS[ind+5] = face[1]
86
87          ind += 6
88      TF = np.squeeze(np.transpose(TF))
89      TS = np.squeeze(np.transpose(TS))
90      print(TF.shape)
91
92
93      # Convert to edge index tensor
94      edge_index = torch.tensor(np.array([TF.astype(int),
           TS.astype(int)]), dtype=torch.long)
95
96      x_in = torch.tensor(new_mesh.points, dtype=torch.
           float)
97      y_out = torch.tensor(scaled_orig.points, dtype=torch
           .float)
98
99      data = Data(x=x_in, y=y_out, edge_index=edge_index,
           pos=faces)
100
101     return data
102
103
104 def generateDataset(data_dir, noise_level, savename):
105
106     files = os.listdir(data_dir)
107     files = sorted(files)
108     print(files) # list of file names recorded in a
           tensor
```

```
109
110     datalist = []
111     for mesh in files:
112         filename = os.path.join(data_dir, mesh)
113         data = mesh2Data(filename, noise_level)
114         print('Data:')
115         print(data)
116         print('num node feats:' + str(data.
               num_node_features))
117         datalist.append(data)
118         print(mesh + " converted to Data object")
119
120     print(datalist)
121     with open(savename, 'wb') as f:
122         pickle.dump(datalist, f)
123
124 # Generate dataset
125 # generateDataset(data_dir, 0.005, 'MDset_cubo_faces.pkl
        ')
126 # generateDataset(data_dir, 0.001, 'myMDset.pkl')
127
128 # Load dataset from file
129 # with open('myMDset.pkl', 'rb') as f:
130 with open('MDset_cubo_faces.pkl', 'rb') as f:
131     datalist = pickle.load(f)
132
133 # Validation function
134 def vali(model, val_loader):
135     model.eval()
136     val_loss = 0
137     for batch in val_loader:
138         out= model(batch.to(device))
```

```
            # loss = F.cosine_embedding_loss(out, batch.y,
               torch.ones(out.size(0)))
            loss = F.mse_loss(out, batch.y)
            val_loss += loss.item()
    return val_loss

# Testing function
def test(model, test_loader):
    model.eval()
    test_loss = 0
    for batch in test_loader:
        out = model(batch.to(device))
        # loss = F.cosine_embedding_loss(out, batch.y,
           torch.ones(out.size(0)))
        loss = F.mse_loss(out, batch.y)
        test_loss += loss.item()
    return test_loss

bs = 1 # batch size

train_dataset = datalist[:24]
val_dataset = datalist[24:28]
test_dataset = datalist[28:29]

size_train = 24
size_val = 4
size_test = 4

train_loader = DataLoader(train_dataset, batch_size=bs)
val_loader = DataLoader(val_dataset, batch_size=bs)
test_loader = DataLoader(test_dataset, batch_size=bs)

# Num node features
```

```
170 nnf = datalist[0].num_node_features
171
172
173 model = Net(0).to(device)
174
175 # Compute the maximum in-degree in the training data.
176 max_degree = -1
177 for data in train_dataset:
178     d = degree(data.edge_index[1], num_nodes=data.
        num_nodes, dtype=torch.long)
179     max_degree = max(max_degree, int(d.max()))
180
181 # Compute the in-degree histogram tensor
182 deg = torch.zeros(max_degree + 1, dtype=torch.long)
183 for data in train_dataset:
184     d = degree(data.edge_index[1], num_nodes=data.
        num_nodes, dtype=torch.long)
185     deg += torch.bincount(d, minlength=deg.numel())
186
187 # model = PNAmodel(num_node_features=nnf, deg=deg)
188 model.to(device)
189
190
191 optimizer = torch.optim.Adam(model.parameters(), lr
      =0.0025, weight_decay=0)
192
193 print('Training..')
194 dat = dt.now()
195
196 best_val = 1e10
197 for epoch in range(201):
198     model.train()
199     epoch_loss = 0
```

```
200     R = 0
201     for batch in train_loader: #24 iterations
202         R += 1
203         optimizer.zero_grad()
204         out  = model(batch.to(device))
205         loss = F.mse_loss(out, batch.y)
206         epoch_loss += loss.item()
207         loss.backward() # e se lo mettessi fuori dal for
               ?
208         optimizer.step()
209 
210     print('Epoch: {} -- Training Loss: {:e}'.format(
           epoch, epoch_loss))
211     if (epoch > 1) and ((epoch % 10) == 0):
212         val_loss = vali(model, val_loader)
213         print('Validation Loss: {:e}'.format(val_loss /
               (size_val / bs)))
214         if val_loss < best_val:
215             print('BEST VAL!')
216             best_val = val_loss
217             torch.save(model.state_dict(), './graphW
                   {}-{}-{}_{}-{}_{}.pth'.format(dat.year,
                   dat.month, dat.day, dat.hour, dat.minute,
                    val_loss / (size_val/bs)))
218 
219 print('Testing Loss: {:e}'.format(test(model,
       test_loader) / (size_test/bs)))
220 
221 model.eval()
222 
223 out_test = model(datalist[30].to(device))
```

5.6.3 inference.py

```
import os
import matplotlib.pyplot as plt
import torch
import pickle
from torch_geometric.loader import DataLoader
import pyvista as pv
import numpy as np
from networks import Net

os.environ["KMP_DUPLICATE_LIB_OK"] = "TRUE"

device = torch.device('cpu')
data_dir = "./graphset"
plt.ion()

files = os.listdir(data_dir)
files = sorted(files)
print(files)

datalist = []
namelist = []

for mesh in files:
    filename = os.path.join(data_dir, mesh)
    namelist.append(filename)

# Load dataset from file
with open('MDset_cubo_faces.pkl', 'rb') as f:
    datalist = pickle.load(f)

bs = 1 # batch size

```

```
train_dataset = datalist[:24]
val_dataset = datalist[24:28]
test_dataset = datalist[28:]

size_train = 24
size_val = 4
size_test = 5

train_loader = DataLoader(train_dataset, batch_size=bs)
val_loader = DataLoader(val_dataset, batch_size=bs)
test_loader = DataLoader(test_dataset, batch_size=bs)

# Num node features
nnf = datalist[0].num_node_features

# Graph Unet model
model=Net(0)

# Load Weights
PATH="./graphW2022-6-2_15-35_7.172003279265482e-05.pth"
model.load_state_dict(torch.load(PATH))
model.eval()


def calcola_mad(vertici_gt, vertici_restored, facce):
    face_normals_restored = np.cross(vertici_restored[
       facce[:, 1]] - vertici_restored[facce[:, 0]],
       vertici_restored[facce[:, 2]] -vertici_restored[
       facce[:, 1]])
    norm_restored = np.sqrt(np.sum(np.square(
       face_normals_restored), 1))
    face_normals_restored /= np.tile(norm_restored, (3,
       1)).T
```

```
61
62      face_normals_gt = np.cross(vertici_gt[facce[:, 1]] -
            vertici_gt[facce[:, 0]], vertici_gt[facce[:, 2]]
            -vertici_gt[facce[:, 1]])
63      norm = np.sqrt(np.sum(np.square(face_normals_gt), 1)
          )
64      face_normals_gt /= np.tile(norm, (3, 1)).T
65
66      inner = [np.inner(face_normals_restored[i],
          face_normals_gt[i]) for i in range(
          face_normals_restored.shape[0])]
67      sad = np.rad2deg(np.arccos(np.clip(inner, -1.0, 1.0)
          ))
68      mad = np.sum(sad) / len(sad)
69
70      return mad
71
72
73  for ind, item in enumerate(test_loader):
74
75      out = model(item.to(device))
76
77      # size_train + size_val + ind
78      filename = namelist[size_train + size_val + ind]
79      #filename = namelist[ind]
80      orig_mesh = pv.read(filename)
81
82      # Centered and Scaled mesh
83      noisy_mesh = orig_mesh.copy()
84      restored_mesh = orig_mesh.copy()
85
86      noisy_mesh.points = np.asarray(item.x.to('cpu'))
```

```
87      restored_mesh.points = np.asarray(out.detach().to('
           cpu'))
88
89
90      faces=np.array(item.pos).squeeze()
91      vertices_noisy=np.array(item.x)
92
93      vertices_gt=np.array(item.y)
94      vertices_restored=  np.asarray(out.detach().to('cpu'
           ))
95
96      mad= calcola_mad(vertices_gt,vertices_restored,
           faces)
97      print("Mad Restored- GT= ",mad, filename)
98      EV= np.linalg.norm(vertices_gt-vertices_restored)/
           vertices_gt.size
99
100     mad_noisy= calcola_mad(vertices_gt,vertices_noisy,
           faces)
101     print("Mad Noisy - GT= ",mad_noisy, filename)
102     plotter = pv.Plotter(shape=(1, 3))
103
104     plotter.subplot(0, 0)
105     plotter.add_text("Original Mesh", font_size=10)
106     plotter.add_mesh(orig_mesh)
107
108     plotter.subplot(0, 1)
109     plotter.add_text("Noisy Mesh MAD " +str(mad_noisy),
           font_size=10)
110     plotter.add_mesh(noisy_mesh)
111
112     plotter.subplot(0, 2)
```

```
113     plotter.add_text("Restored Mesh GAT Fidelity solo
           vertici \n \n MAD  "+str(mad)+" \n \n EV   "+str(
           EV),font_size=12)
114     plotter.add_mesh(restored_mesh)
115
116     plotter.show()
```

Bibliografia

[1] Kipf, Thomas. Graph convolutional networks, 2016. https://tkipf.github.io/graph-convolutional-networks/, Last accessed on 2022-06-15.

[2] Michael, Bronstein. Geometric foundations of deep learning, 2021. https://towardsdatascience.com/geometric-foundations-of-deep-learning-94cdd45b451d, Last accessed on 2022-06-11.

[3] Ravindra, Parmar. Common loss functions in machine learning, 2018. https://towardsdatascience.com/common-loss-functions-in-machine-learning-46af0ffc4d23, Last accessed on 2022-06-07.

[4] Sumit, Saha. A comprehensive guide to convolutional neural networks, 2018. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks, Last accessed on 2022-06-02.

[5] Surbhi, Arora. Supervised vs unsupervised vs reinforcement, 2020. https://www.aitude.com/supervised-vs-unsupervised-vs-reinforcement/, Last accessed on 2022-06-10.

[6] Tobias, Skovgaard, Jepsen. How to do deep learning on graphs with graph convolutional networks part 2: Semi-supervised learning with spec-

tral graph convolutions, 2019. https://towardsdatascience.com/how-to-do-deep-learning-on-graphs-with-graph-convolutional, Last accessed on 2022-06-22.

[7] Abdel-Nasser Sharkawy. Principle of Neural Network and Its Main Types: Review. *Journal of Advances in Applied Computational Mathematics*, 7(August):8–19, 2020.

[8] Ayushi Chahal and Preeti Gulia. Machine learning and deep learning. *International Journal of Innovative Technology and Exploring Engineering*, 8(12):4910–4914, 2019.

[9] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in Neural Information Processing Systems*, 2017-December(Nips):1025–1035, 2017.

[10] Shota Hattori, Tatsuya Yatagawa, Yutaka Ohtake, and Hiromasa Suzuki. Deep Mesh Prior: Unsupervised Mesh Restoration using Graph Convolutional Networks. 2021.

[11] Jaspreet. A concise history of neural networks, 2016. https://towardsdatascience.com/a-concise-history-of-neural-networks-2070655d3fec2, Last accessed on 2022-05-30.

[12] Diederik P. Kingma and Jimmy Lei Ba. Adam: A method for stochastic optimization. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–15, 2015.

[13] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, pages 1–14, 2017.

[14] Yassine Ouali, Céline Hudelot, and Myriam Tami. An Overview of Deep Semi-Supervised Learning. pages 1–43, 2020.

[15] Sebastian Ruder. An overview of gradient descent optimization algorithms. pages 1–14, 2016.

[16] Seldon. A concise history of neural networks, 2022. `https://www.seldon.io/neural-network-models-explained`, Last accessed on 2022-05-06.

[17] Youngjoo Seo, Michaël Defferrard, Pierre Vandergheynst, and Xavier Bresson. Structured sequence modeling with graph convolutional recurrent networks. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11301 LNCS(2013):362–373, 2018.

[18] Sathyanarayana Shashi. A Gentle Introduction to Backpropagation. pages 1–15, 2014.

[19] G. L. Shaw. Donald hebb: The organization of behavior. In Günther Palm and Ad Aertsen, editors, *Brain Theory*, pages 231–233, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[20] Tomasz Szandała. Review and comparison of commonly used activation functions for deep neural networks. *Studies in Computational Intelligence*, 903:203–224, 2021.

[21] Petar Veličković, Arantxa Casanova, Pietro Liò, Guillem Cucurull, Adriana Romero, and Yoshua Bengio. Graph attention networks. *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, pages 1–12, 2018.

[22] Rikiya Yamashita, Nishio Mizuho, Kinh Gian Do Richard, and Kaori Togashi. Convolutional neural networks: an overview and application in radiology. pages 1–19, 2018.

[23] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 974–983, 2018.

[24] Si Zhang, Hanghang Tong, Jiejun Xu, and Ross Maciejewski. Graph convolutional networks: a comprehensive review. *Computational Social Networks*, 6(1), 2019.

Ringraziamenti

Mi sono trasferito a Cesena nel 2019 per studiare quello che pensavo sarebbe stato il corso perfetto per me. Fortunatamente non mi sbagliavo. Ho trascorso 3 anni bellissimi, fatti di alti e bassi, ma soprattutto di nuove esperienze ed amicizie.

Ci tengo a ringraziare in primis la prof.ssa Lazzaro per l'enorme aiuto e la costante disponibilità che ha dimostrato sia nel periodo di tirocinio che in quello di tesi. Inoltre, ci tenevo a ringraziare tutta la mia famiglia: mamma, papà e Federica. Grazie per avermi sopportato ma soprattutto supportato in questi anni.

Non sarebbe corretto concludere questa tesi senza mandare un grandissimo abbraccio e ringraziamento a tutti gli amici di Zucchero Sintattico, in ordine alfabetico: Ale, Alex, Gus, Gigi, Kel, Manu e Tommi.

Senza di voi sicuramente non sarei dove sono ora e non sarei chi sono oggi.

Di certo non dimentico gli amici di Padova, mando un caloroso abbraccio anche a voi e vi ringrazio per tutte le volte che mi siete stati vicino.

Il mio percorso a Cesena è ormai giunto al termine, è arrivato il momento di cambiare di nuovo città e trasferirmi nella lontana Stoccolma. Questo però non può che essere un arrivederci, perché a casa non si dice mai "addio".