UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA ELETTRONICA ED INFORMATICA

Convolutional Autoencoders per il rilevamento di anomalie nei componenti elettronici

Tesi di laurea di:

Giuseppe Mantineo

Relatore:

Chiar.mo Prof.

Dario Bruneo

Anno accademico 2020/2021

“The people I've met and the places I've been
Are all what make me the man I so proudly am”

Ringraziamenti

Un ringraziamento a mia madre e mio padre per il continuo supporto, per avermi sempre sostenuto anche nei momenti più difficili ed essere i primi a credere in me, ad avermi insegnato cosa sia il sacrificio e la dedizione.
Un grazie a mio fratello Gabriele, sei speciale.
A Ghella per essere la mia sorella mancata, sempre presente nella mia vita.
A Bonny per essere la persona di cui ho bisogno nella mia vita e darmi l'affetto di cui ho bisogno.
Ad Angelo per essere il mio punto di riferimento, quello che ha sempre la soluzione a tutto.
Ad lenia per essere la mia compagnia di sessioni notturne, la mia geme.
A Davide, Cristian, Enrico e Giovanni per i bellissimi momenti passati assieme, per le mille risate.
Un grazie ad Alessandro e Pietro, la Triplice, miei compagni di vita insieme a Ciccio e Peppe e Claudio.
Un grazie a Mara, la mia sorellina, l'amore mio, sei una persona stupenda, grazie per far parte della mia vita.
Un grazie a Marts, per aver portato un po' di freschezza nella mia vita ed essere diventata un punto di riferimento nella mia vita.
Un ringraziamento al professore Bruneo e all'ingegnere De Vita per il continuo supporto durante questo lavoro di tesi.
Per ultimo, ma non meno importante, un grazie a me stesso, per aver continuato nonostante tutto, ad aver creduto in me stesso, per avercela fatta.

Indice

Elenco Delle Figure

Capitolo 1

Introduzione

1.1 Anomaly Detection

L' **Anomaly Detection**, ovvero il rilevamento di anomalie, è una tecnica che permette di identificare uno o più elementi "rari", che si distinguono particolarmente dal resto dei dati. [1] Per cui permette di rilevare quelle istanze dei dati che presentano delle anomalie, così da poter poi esser sistemate. Il notevole aumento di dati prodotti da ogni essere umano ha permesso a queste tecniche di essere ancora più praticabili. Studiando, infatti i dati che definiremo normali, si possono trovare dei pattern all'interno di essi che permetteranno poi di identificare le istanze anomale dei dati. Per cui, i dati i cui patterns si discostano notevolmente da quelli studiati dal sistema verranno identificati come anomali.

Il nostro campo di applicazione è quello del rilevamento di anomalie nei componenti elettronici, utilizzando tecniche di analisi delle immagini. Questo può essere utile nelle industrie 4.0, dove un sistema del genere può semplicemente indicare quali componenti di un macchinario hanno subito

un guasto, così da permetterne una riparazione o sostituzione e tornare immediatamente al processo di fabbricazione, permettendo così di avere una maggiore efficienza di produzione

La tecnica utilizzata per sviluppare un algoritmo in grado di attuare ciò è quella dei **Convolutional Autoencoders,** una particolare famiglia di algoritmi di machine learning che cercano di ricopiare l'input nel loro output sotto alcune restrizioni, in modo da dover imparare pattern e feature di essi. Prima di vedere più nel dettaglio cosa sono e come funzionano, diamo una breve introduzione della teoria che sta dietro a tutto ciò.

Capitolo 2

Il Machine Learning

In questo capitolo si introdurrà il Machine Learning, parlando di cosa sia Machine Learning, di quali siano le sue principali applicazioni e di come si costruisca ed alleni un algoritmo di Machine Learning. Lo scopo del machine learning è quello di realizzare algoritmi "autonomi", in grado cioè di imparare dall'esperienza in uno specifico compito:

"Un programma apprende da una certa esperienza E se: nel rispetto di una classe di compiti T, con una misura della prestazione P, la prestazione P misurata nello svolgere il compito T è migliorata dall'esperienza E" (T.M. Mitchell) [2]

2.1 Intelligenza Artificiale E Machine Learning

Il Machine Learning, ma più in generale il ramo dell'Intelligenza Artificiale, si pone l'obbiettivo di dotare le macchine di caratteristiche che vengono solitamente associate ad un tipo di intelligenza umana, ad esempio visiva, decisionale e così via. Si parla quindi di sistemi intelligenti in grado di ricreare una o più di queste differenti forme di intelligenza. Il machine learning cambia radicalmente l'approccio alla programmazione, difatti non è più sviluppato un algoritmo strutturato in

cui bisogna gestire le varie occorrenze, bensì l'algoritmo di machine learning attraverso l'analisi dei dati forniti si evolve imparando dai dati stessi. Il machine learning è quindi la scienza (ed arte) della programmazione di imparare dai dati, ovverosia di imparare senza essere esplicitamente programmato. [3] Per cui entrano in gioco concetti come il training del sistema stesso, l'*overfitting* dei dati ed altri. Il machine learning è un ottimo approccio risolutivo per tutti quei problemi in cui è richiesto al sistema un continuo aggiornamento (spam filters, ad esempio) oppure a problematiche a cui la programmazione tradizionale porrebbe soluzioni ottenendo dei risultati non ottimali, come ad esempio la computer vision, il natural processing Language ed altre applicazioni.

2.2 Classificazione dei Sistemi Di Machine Learning

Bisogna anzitutto distinguere fra diversi tipi di sistemi di Machine Learning, tra questi abbiamo il **Supervised Learning,** in cui al sistema vengono fornite, oltre che le istanze dei dati su cui allenarsi, anche le rispettive etichette (label), ovvero le "soluzioni". Cosicché il sistema possa rendersi conto se le sue conclusioni siano corrette o meno; solitamente questo tipo di sistemi viene utilizzato per applicazioni di Classificazione o di Regressione. Un'applicazione di Classificazione ha il compito di classificare un oggetto in una delle varie classi, ad esempio se un'immagine ritrae un gatto o un cane. Un'applicazione di regressione, invece, cerca di prevedere un valore a partire dai dati che gli sono forniti, come ad esempio il prezzo di un immobile in base alla zona dove esso è situato.

Nell' **Unsupervised Learning,** invece, vengono forniti solo i dati sui quali il sistema deve allenarsi, senza però le relative etichette (*label)*, per cui dovrà imparare senza un riscontro, questo tipo di sistema è generalmente utilizzato in applicazioni di Clustering, Dimensionality Reduction etc.

Esiste poi una via di mezzo fra questi due sistemi, ovvero il **Semisupervised learning,** in cui una parte del dataset è etichettata (*labeled)* ed una parte no. Questo capita perché il labeling dei dati è piuttosto costoso a livello tempistico. Questi sistemi trovano applicazione nei tag delle foto, ad esempio, dove in alcune foto i soggetti sono taggati e poi sta al sistema taggarli nelle altre foto.

Si ha poi una classificazione in base a come i dati vengono forniti durante il training, a tal proposito i sistemi basati sul **Batch Learning** hanno la peculiarità di dover essere allenati sull'intero set di dati disponibili, ciò vuol dire che se l'istanza dei dati venisse aggiornata sarebbe necessario allenare nuovamente il sistema sull'intera istanza aggiornata, un processo molto dispendioso. I sistemi che, invece, sono basati invece sull'**Online Learning** possono essere allenati su piccole istanze alla volta, permettendo così di poter essere allenati anche solo sulle nuove porzioni di dati, dette **mini-batches**. In questi sistemi è importante settare il parametro del **learning rate,** che determina quanto le nuove istanze siano influenti rispetto a quelle vecchie: con un learning rate elevato il sistema darà un'importanza elevata alle nuove istanze rispetto a quelle precedenti, viceversa le nuove istanze verranno quasi del tutto ignorate. È quindi importante trovare il giusto equilibro tra le due situazioni per avere un sistema performante. Inoltre, i sistemi di machine learning vengono

classificati in base a come poi generalizzano, una volta svolto il training, i sistemi **Instanced-Based** basano la loro generalizzazione su similarità rispetto agli esempi forniti, un sistema invece **Model-Based** crea invece un modello dagli esempi forniti ed è quindi in grado di generare delle *prediction.* In questo caso il modello verrà allenato a creare un modello che riesca quanto più possibile a generalizzare correttamente.

2.3 Principali Problematiche Dei Sistemi Di Machine Learning

I sistemi di Machine Learning imparano dai dati, i quali sono utilizzati nel processo di training per permettere poi al modello stesso di generalizzare. Se i dati forniti non sono di sufficiente qualità, ovvero dei dati che hanno al loro interno errori o che sono "rumorosi", porteranno il sistema a non generalizzare correttamente. Un' altra problematica può essere la non sufficiente quantità di dati, in questo caso il modello non potrà essere allenato su grandi istanze e non riuscirà a creare un modello che rappresenti concretamente la problematica in questione. Inoltre, i dati possono essere non rappresentativi per un qualche motivo, si parla di **Sampling Bias**, ed in anche in questo caso ciò porterà il sistema a non generalizzare efficacemente. È quindi essenziale che i dati siano della più alta qualità possibile; infatti, una gran parte del tempo nello sviluppo di un sistema di Machine Learning è dedicato alla raccolta, pulizia dei dati, ad estrarne da essi feature ed ottimizzarli, in modo che il sistema abbia i miglior dati possibili per un dato compito. Un' ulteriore problematica è introdotta dagli algoritmi stessi di machine learning, in quanto ognuno di essi non può coprire tutti i vari task possibili. Va quindi selezionato quello

ottimale in base al task in questione. Tipiche problematiche riguardo gli algoritmi sono quelle di *overfitting* ed *underfitting*.

Si parla di *overfitting* quando il modello ottiene ottimi risultati sui dati del training, ma non riesce poi concretamente a generalizzare correttamente, questo accade solitamente perché si sta usando ad esempio un modello molto complesso per il task o perché il set di dati del training ha molte features o presenta elevato rumore. Una soluzione può quindi essere, oltre quella di un'ottimizzazione del set di dati del training, quella di ridurre la complessità del modello, solitamente tramite tecniche di *regularization*. La *regularization* in pratica, restringe, limita il grado di libertà dei parametri di fatto riducendo la complessità del modello in modo che non si verifichi l'*overfitting*. L'*underfitting* è invece l'esatto opposto, ovvero il modello non performa bene nella generalizzazione ma nemmeno nei dati del training, sarà quindi necessario rendere il modello più complesso, ottimizzare il dataset ed anche ridurre la *regularization*. Ma come si valutano le prestazioni di un modello di Machine Learning?

2.4 Testing e Validation di un sistema

Per la valutazione di un sistema, la tecnica adottata è quella di dividere il dataset in due porzioni, solitamente seguendo la regola del 80-20. Infatti, l'80% del dataset è utilizzato per il training mentre il restante 20% è utilizzato per testing e di conseguenza vengono chiamati training set e test set. È importante notare che al fine di rendere il dataset il più distribuito possibile, prima di effettuare la suddivisione è buona norma randomizzare la posizione delle varie istanze così da renderlo il più indipendente ed

identicamente distribuito possibile. Ma se si vuole scegliere la versione migliore di un modello? In questo caso, la tecnica utilizzata è un po' differente, difatti si effettua una ulteriore suddivisione del training set, in una parte che sarà sempre delegata al training dei vari modelli con parametri differenti ed una piccola porzione invece, detta validation set, che servirà per valutare il migliore tra questi modelli. Quello che otterrà le metriche migliori sul validation set, sarà poi successivamente allenato sull'intero training set e testato definitivamente sul test set. Un modello che performa bene sul training set e male sul test set sta evidentemente *overfittando*. È importante che il validation set abbia le dimensioni corrette per offrire delle metriche valide. Un' altra tecnica, che pone una soluzione alla problematica della dimensione del validation test, è quella della *cross-validation*, che consiste nell'utilizzo di tanti piccoli validation set ed ogni modello dopo il training è valutato su ognuno di questi set, dopodiché le metriche vengono mediate, fornendo così una più accurata rappresentazione delle reali prestazioni dei vari modelli, ovviamente con questa tecnica il costo computazionale aumenta notevolmente.

2.5 Allenamento di un modello di Machine Learning

Un sistema di Machine Learning impara attraverso i dati, ma in realtà il processo di training è un po' più complesso. Un algoritmo di machine learning è composto da moltissimi parametri e lo scopo del training è quelli di trovare i valori migliori per ognuno di questi parametri affinché il modello generalizzi nel modo migliore possibile. Per fare ciò è necessario poter valutare le prestazioni del sistema analiticamente, per cui si utilizza una *cost function*, che è differente in base al tipo di applicazione

del sistema, ad esempio in un task di regressione lineare, una molto utilizzata è la Root Mean Square Error (RMSE), ovvero lo scarto quadratico medio, spesso noto anche come deviazione standard, o in egual modo la Mean Square Error (MSE):

$$MSE(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m}(\theta^T x_i - y_i)^2$$

In questa equazione **θ** è il vettore dei parametri del modello, **x** è il vettore dell'i-esima feature, il prodotto scalare dei due restituisce valore predetto dal sistema, è spesso anche chiamato come hypothesis function questo prodotto. Il vettore y contiene il corretto risultato dell'istanza corrente. Per cui il processo di training, cerca di trovare il **θ** che minimizzi questa cost function.

Esistono due approcci per la risoluzione di questa problematica, la prima calcola il **θ** ottimale attraverso la Normal Equation, ma è necessario calcolare l'inversa di una matrice che non sempre è possibile ed inoltre il costo computazionale di tutta la procedura è piuttosto elevato. Il metodo spesso utilizzato è quello del **Gradient Descent**, l'idea generale è quella di procedere per piccole iterazioni che modifichino leggermente i parametri fino a trovare il **θ** che minimizzi il più possibile la cost function. All'atto pratico quello che questa tecnica fa è calcolarsi il gradiente locale rispetto a θ, muovendosi nella direzione discendente, quando il gradiente varrà zero, la cost function è stata minimizzata. Partendo da valori di **θ** casuali, possiamo definire un training step come segue:

$$\boldsymbol{\theta}^{\boldsymbol{next\ step}} = \boldsymbol{\theta} - \eta\, \nabla \mathrm{MSE}(\boldsymbol{\theta})$$

dove:

- $\theta^{next\ step}$ è il vettore dei parametri del modello dello step successivo di training
- $\boldsymbol{\theta}$ è il vettore dei parametri del modello dello step corrente di training
- $\boldsymbol{\eta}$ è il learning rate
- $\boldsymbol{\nabla MSE(\theta)}$ è il gradiente della cost function calcolato rispetto a $\boldsymbol{\theta}$

La selezione del learning rate ottimale è di fondamentale importanza, in quanto se troppo piccolo l'algoritmo avrà bisogno di tantissime iterazioni, potendo anche non convergere ad una soluzione, se troppo grande invece potrebbe saltare il punto di minimo ed anche in questo caso non convergendo ma addirittura divergendo. Per trovare il learning rate ottimale si può utilizzare una *grid search,* limitando il numero di iterazioni escludendo così a priori tutti quei valori per i quali la convergenza viene ottenuta lentamente o non ottenuta completamente. Per far ciò solitamente, si setta un numero elevato di iterazioni, ma si fissa una tolleranza **ε,** quando la norma del vettore gradiente diventa più piccola di questa tolleranza, allora il Gradient Descent ha quasi raggiunto il minimo. Il grande problema di questa tecnica è l'onere computazionale, infatti per calcolare il gradiente è necessario farlo sull'intero dataset ad ogni iterazione, processo piuttosto costoso.

Per evitare ciò ci sono due soluzioni, **SGD (**Stochastic Gradient Descent) ed il **Mini Batch Gradient Descent.** Il primo seleziona una istanza casuale del training set ad ogni iterazione e si calcola il gradiente rispetto a questa istanza. Quindi la natura dell'algoritmo è stocastica, nonostante si sia trovata una soluzione al costo computazionale, la natura stessa dell'algoritmo non garantisce regolarità nel proseguo delle sue iterazioni.

Infatti, è molto probabile che raggiunga i pressi del minimo senza mai raggiungere il minimo stesso. Una soluzione all'irregolarità sia dell'algoritmo, che anche della cost function, poiché anche essa può essere non regolare, è quella di implementare un **learning schedule**, ovvero durante il training si scala gradualmente il **learning rate.** È imprescindibile che le istanze del training set siano **Indipendenti e Identicamente Distribuiti (**IID), per ottenere ciò è sufficiente randomizzare la disposizione del dataset.

Il Mini Batch GD segue la stessa idea, solo che invece di selezionare una singola istanza casualmente, seleziona una piccola porzione del dataset, detta appunto mini batch, le considerazioni sono più o meno le stesse, essendo anche questo algoritmo di natura stocastica. Nell'irregolarità però, è un po' più regolare, quindi otterrà risultati lievemente migliori.

2.6 Regularization

Come abbiamo visto, una delle grandi sfide nel processo di creazione di un sistema di Machine Learning è trovare il giusto compromesso tra le varie metriche in modo che esso non *overfitti*. Abbiamo anche visto che la **regularization** ci aiuta in questo, in quanto essa consiste nel limitare i gradi di libertà di un sistema, in modo da renderlo meno complesso e quindi meno tendente all'*overfitting* dei dati. La tecnica della regularization, in generale, consiste nell'aggiungere un termine chiamato regularizer alla cost function, ogni tecnica di regularization introduce un hyperparameter per specificare quanto si vuole regolarizzare un modello, è quindi importante anche qui trovare il giusto valore affinché il modello

performi al meglio delle possibilità. Per cui più in generale, il compito della regularization è quello di ridurre l'errore di generalizzazione senza modificare l'errore di training, infatti l'ulteriore contributo è utilizzato solo durante il training, la valutazione del modello sul test set è effettuata con la cost function originale. Esistono diverse tecniche di regularization, ognuna con i loro pro e contro, quale scegliere dipende dal tipo di applicazione del sistema e dai risultati che si vogliono ottenere. Infatti, non esiste un algoritmo di machine learning migliore di un altro.

Capitolo 3

Deep Learning

In questo capitolo si introdurrà il Deep Learning, una particolare classe di algoritmi di machine learning, in grado di riconoscere pattern molto complessi all'interno dei dati. Si esporranno le principali problematiche che questi algoritmi tentano di risolvere e le tecniche utilizzate nella costruzione, allenamento e gestione di un modello di Deep Learning.

3.1 Perché Il Deep Learning

Gli algoritmi di machine learning sono riusciti nell'intento di porre soluzioni a molte problematiche, ma allo stesso tempo alcune di esse come lo speech recognizing o l'object detection non vengono risolte perfettamente da questo tipo di algoritmi. Questo è dovuto principalmente alla complessità dei dati, infatti essi sono multidimensionali ed un algoritmo di machine learning non riesce a generalizzare bene con questa tipologia di dati e quindi costruire un modello efficace a partire da essi. Questo fenomeno è detto **curse of dimensionality.** [4] Per sopperire a ciò, nel tempo sono stati sviluppati algoritmi molto più complessi ed è questo ciò di cui si occupa il **Deep Learning.** Infatti**,** esso trova applicazioni in

tasks come Facial Recognition, Virtual Assistants e tutti quei problemi complessi a cui un semplice algoritmo di machine learning non riesce a proporre soluzioni efficaci.

3.2 Deep Neural Networks e Backpropagation

L'elemento principale di un algoritmo di Deep Learning è l'Artificial Neuron, questo prende spunto infatti dal neurone umano. Esso attiva il suo output quando più di un dato numero di input ad esso collegati si attivano, gli input e gli output sono binari (on/off). Con questa semplice struttura è possibile ricreare tutte le funzioni logiche, come è possibile vedere in figura 1.

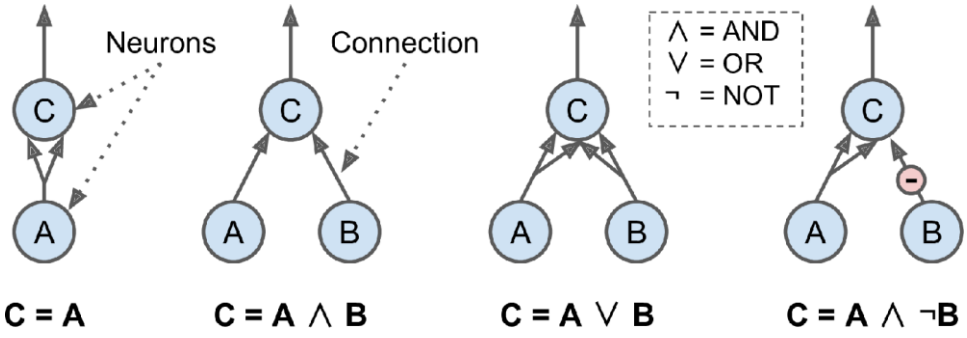


Figura 1 *- Esempi di funzioni logiche elementari riprodotte con Artificial Neural Networks [3]*

Uno dei primi modelli che implementa questa ideologia è il **Perceptron**, basato su un artificial neuron leggermente differente chiamato *threshold logic unit (TLU).* Input e output sono dei valori numerici e ad ogni input è associato un peso, il TLU computa la somma pesata degli input ($z =$

$w_1x_1 + w_2x_2 + \cdots + w_nx_n = \boldsymbol{x^tW}$**),** dopodiché applica una **step function** a questa somma e ne fornisce l'output:

$$h_w(\boldsymbol{x}) = \boldsymbol{step}(\boldsymbol{z})$$

La step function più comunemente utilizzata nei perceptron è il gradino di Heaviside oppure è utilizzata la funzione segno:

$$heaviside\ (z) = \begin{cases} 1\ se\ z \geq 0 \\ 0\ se\ z < 0 \end{cases} \qquad segno(z) = \begin{cases} 1\ se\ z > 0 \\ 0\ se\ z = 0 \\ -1\ sez < 0 \end{cases}$$

Un singolo TLU può essere utilizzato come un classificatore binario, infatti esegue il processo sopra descritto, dopodiché se il risultato supera una certa soglia da in output la classe positiva, altrimenti negativa. Un perceptron è composto da un singolo layer di TLU, con ogni TLU connesso a tutti gli input, ed in questo caso si parlerà di **Dense Layer.**

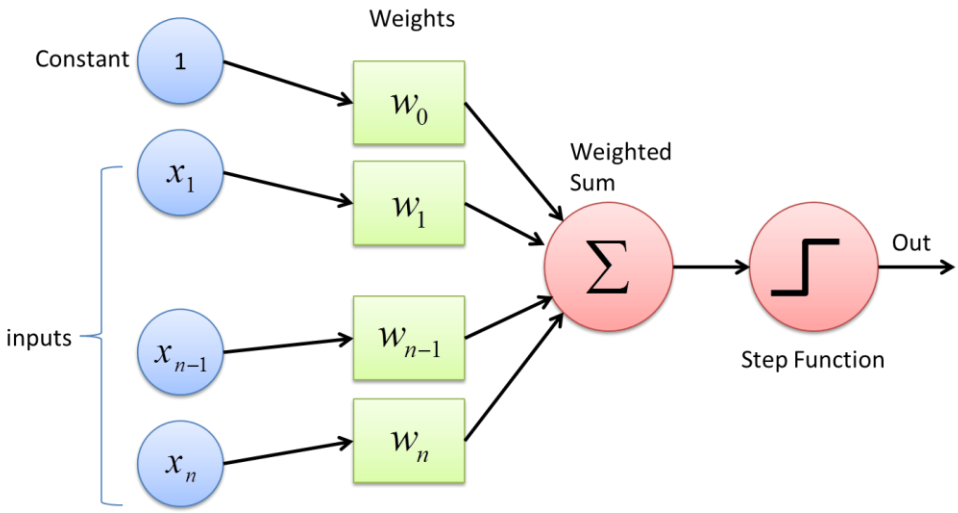


Figura 2 *- Un esempio di architettura di un layer di TLU. [5]*

Gli input sono fatti passare attraverso dei neuroni speciali, gli input neurons che semplicemente danno in uscita ciò che vi è in entrata,

l'insieme degli input neurons forma l'input layer, ma con l'aggiunta di un termine costante, detto *bias neuron*, che fornisce in output una costante (tipicamente 1). Il processo di allenamento per il Perceptron si basa sulla *Hebb's rule*, ovvero che i pesi delle connessioni tra due neuroni tendono ad incrementare quando si accendono simultaneamente. In particolare, i Perceptron utilizzano una variante di questa "regola", infatti l'algoritmo tiene conto dell'errore fatto dalla rete quando effettua una predizione, la **learning rule** rinforza le connessioni che aiutano a ridurre l'errore. Più nel dettaglio, al perceptron è fornita un'istanza di training per volta e per ognuna di essa viene effettuata una predizione, per ogni output neuron che ha fornito una errata predizione, si rinforzano le connessioni e quindi i pesi dagli input che avrebbero contribuito ad una corretta predizione. Questo si può vedere dalla seguente equazione:

$$W_{i,j}^{next} = W_{i,j} + \eta(y_j - \widehat{y_j})\, x_i$$

dove:

- $W_{i,j}^{next}$ è la matrice dei pesi delle connessioni dello step successivo di training
- $W_{i,j}$ è la matrice dei pesi delle connessioni dello step corrente di training
- η è il learning rate
- y_j è l'output desiderato del j-esimo output neuron per la corrente istanza di training
- $\widehat{y_j}$ è l'output predetto del j-esimo output neuron per la corrente istanza di training
- x_i è l'i-esimo valore di input della corrente istanza di training

Il modello è però poco potente con un solo layer, non può quindi imparare pattern complessi. A questo scopo sono state sviluppate le MLP (*Multilayer perceptron),* formate da un input layer, da più di un layer di TLUs, che sono detti *hidden layers*, ed un layer di output. I layer più vicini all'input sono detti *lower layers*, quelli più vicini all'output invece *higher layers.* Ogni layer, escluso quello di output introduce un bias term, ha quindi un bias neuron, inoltre tutti i layer sono **Dense layers,** che ricordiamo vuol dire che sono totalmente connessi. Questa struttura, con un grande numero di hidden layers è detta **Deep Neural Network (DNN).**

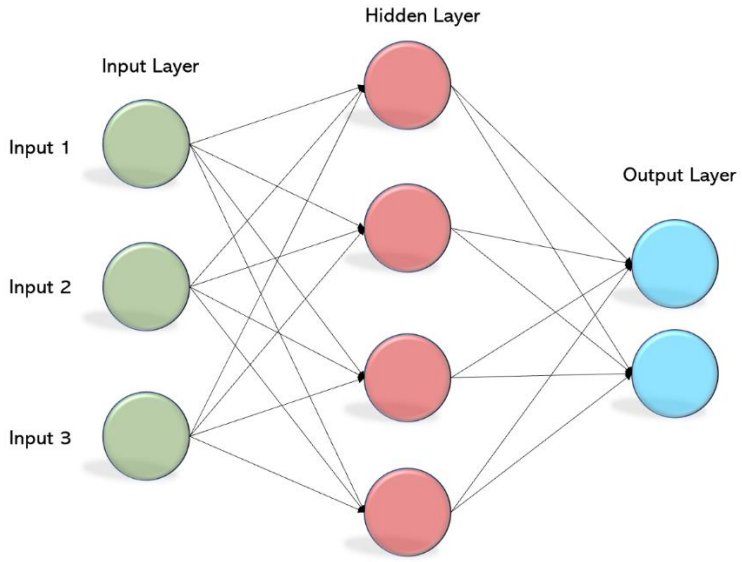


Figura 3 *- Architettura di una MLP, 3 input neuron, un hidden layer con 4 neuroni e 2 output layer. I bias term sono impliciti, per questo non mostrati in figura. [5]*

Il segnale va in una sola direzione, dall'input verso l'output, si parla quindi di una feedforward neural network (FNN). Il training di queste reti è effettuato con una tecnica detta, *batchpropagation*, funziona sostanzialmente come un Gradient Descent, ma utilizza una tecnica differente molto più efficace per il calcolo del gradiente. Il suo

funzionamento è il seguente, la rete maneggia un mini-batch per volta, facendo ciò per tutto il training set più volte, in tale contesto ogni iterazione sull'intero dataset è detta **epoca.** Ogni mini-batch è quindi fornito in pasto alla rete, per cui passa dall'input layer, nei vari hidden layer e infine arriva all'output layer, per ogni layer l'algoritmo computa l'output dei vari neuroni che andranno poi forniti in ingresso al layer successivo, questo è il così detto *forward pass*. L'algoritmo computa poi il network's output error (attraverso una loss function). Dopodiché computa quanto ogni output ha contribuito all'errore (attraverso la chain rule) e questo è ripetuto per tutti i layer a partire dall'output fino al input, il così detto backward pass. Una volta che ha calcolato il contributo all'errore di ogni connessione, l'algoritmo aggiorna i pesi delle connessioni utilizzando i gradienti calcolati nel backward pass. Sottolineiamo che questo viene eseguito per ogni mini-batch di ogni epoca. Affinché questa tecnica funzioni, sono necessarie due condizioni, la prima è che i pesi iniziali delle varie connessioni siano inizializzati casualmente, altrimenti il training fallirebbe. La seconda è il cambio della step function, anche chiama activation function. Difatti, essa è utilizzata per introdurre della non linearità durante il learning process, vengono quindi utilizzate funzioni più complesse come la logistic function, la tangente iperbolica o la ReLu(z).

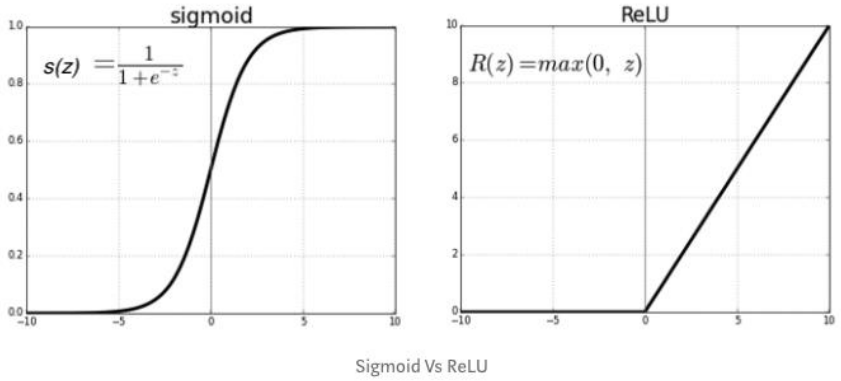


Figura 4 *- A sinistra il grafico della logistic function, detta anche sigmoid per via della sua forma simile ad una s. A destra il grafico della funzione Rectified Linear Unit (ReLU). [6]*

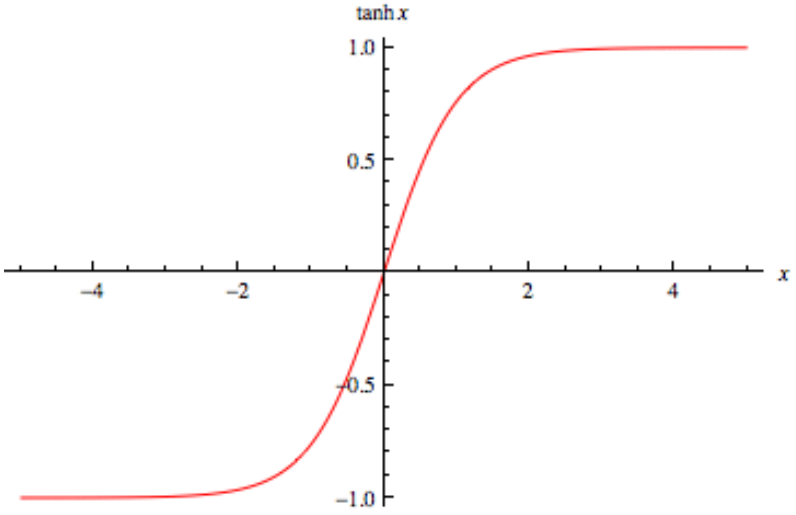


Figura 5 *- Grafico della funzione tanh(z), simile alla logistic solo che varia tra -1 ed 1. Questo aiuta a velocizzare la convergenza dell'algoritmo. [7]*

3.3 Training di una Deep Neural Networks

Il grande vantaggio delle DNN è la loro scalabilità, infatti si possono creare modelli ad hoc per ogni problema, impostando un sistema più o meno complesso. Ovviamente all'aumentare della complessità del problema, aumenterà di conseguenza la complessità del sistema che cerca

di modellarlo e quindi ci saranno molti più parametri che rendono il processo di training più complesso. Il processo di training è quello di backpropagation analizzato precedentemente, ma vediamo ora alcune problematiche che si possono presentare durante l'allenamento di un sistema di Deep Learning. Una di queste è il *vanishing gradient*, ovvero il gradiente diventa sempre più piccolo andando avanti con le iterazioni e quindi le modifiche effettuate al sistema diventano sempre più irrilevanti e l'algoritmo non converge ad una soluzione ottimale. Può accadere il problema opposto, detto *exploding gradient*, ovvero che il gradiente aumenti ad ogni iterazione portando l'algoritmo a divergere. Questa problematica è causata dalla activation function e dall'inizializzazione dei pesi. Infatti, essi vanno inizializzati casualmente con una distribuzione normale con media pari a zero e una varianza pari a

$$\boldsymbol{\sigma^2} = \frac{1}{fan_{avg}}$$

dove fan_{avg}è la somma tra il numero di ingressi ed il numero di uscite diviso 2

$$\boldsymbol{fan_{avg}} = \frac{fan_{in} + fan_{out}}{2}$$

Questa strategia di inizializzazione prende il nome di *Glorot Initialization,* sostituendo al denominatore della varianza fan_{in},ovvero il numero degli input, si ottiene la *LeCun Initialization.* Esistono anche altre tipologie di inizializzazioni, lo scopo di queste strategie è quello di permettere di velocizzare notevolmente i tempi di training. Per quanto riguarda invece le activation function, esse per non incidere in questa problematica del

gradiente non devono saturare. Per questo motivo sono state sviluppate alcune versioni particolari della ReLU specifiche per ottimizzare i processi di training. La ReLU standard soffre del problema che alcuni neuroni "muoiano" durante il processo di training. Tutte queste tecniche evitano che il problema del gradiente si presenti nelle fasi iniziali del training, ma ciò non esclude che possa presentarsi più in avanti nel processo. Per prevenire ciò è stata sviluppata una tecnica, detta **Batch Normalization** (BN). Essa consiste nell'aggiunta di un'operazione di centratura e normalizzazione dei livelli di input prima o dopo l'activation function, questo per evitare che si verifichi il fenomeno della *internal covariate shift.* Ogni layer riceve in input l'output del layer precedente, se questi input presentano una grande covarianza interna il processo di training sarà molto lento, se invece è normalizzato e centrato il processo sarà molto più veloce. Tale tecnica, funziona inoltre come una sorta di regularization, per cui ha notevoli vantaggi al fine di evitare fenomeni di overfitting. A livello pratico quello che fa è calcolare il valore medio e la varianza del batch corrente e normalizzare sottraendo il valore medio e dividendo per la varianza.

$$\widehat{x_\iota} = \frac{x_i - \mu_b}{\sqrt{\sigma_b^2 + \varepsilon}}$$

dove:

- ε è un termine di "smoothing" che serve ad evitare l'eventualità di divisioni per zero, assume infatti solitamente il valore di 10^{-5}.
- μ_b è il valore medio del mini batch di input
- σ_b^2 è la deviazione standard del mini batch di input
- x_i sono i valori di input del corrente mini batch
- $\widehat{x_\iota}$ sono i nuovi valori del mini batch centrati e normalizzati

I valori normalizzati vengono poi moltiplicati e sommati con due nuovi vettori di parametri che la rete apprenderà durante il training

$$y_i = \boldsymbol{\gamma}\widehat{\boldsymbol{x}_i} + \beta$$

dove:

- $\boldsymbol{\gamma}$ è il vettore di scalamento dell'output per il layer
- $\boldsymbol{\beta}$ è il vettore di offset dell'output per il layer
- y_i è l'i-esimo output scalato e shiftato

Per il testing poi, l'algoritmo calcola media e varianza dell'intero set durante il training e userà questi valori calcolati per normalizzare gli input del test set. Grazie a questa tecnica è possibile utilizzare learning rate più elevati ed anche activation function saturanti, come la tanh(z) o la logistic function, e come detto funge anche in parte da regularization.

Un'altra metodologia per velocizzare il processo di training è quello di utilizzare delle versioni ottimizzate del Gradient Descent, ne esistono di diversi tipi ognuno con i propri pro e contro. Anche il giusto valore del learning rate può notevolmente incrementare la velocità del processo, solitamente più che impostare un singolo valore si implementa un *Learning Rate Schedule*, ovvero si parte da un valore alto del learning rate e poi durante il training si diminuisce secondo una certa regola dipendente dall'algoritmo di schedule utilizzata, alcuni funzionano anche in maniera opposta, partendo da un valore basso ed aumentandolo. È importante notare che una rete ha una notevole quantità di parametri, che è uno dei

punti di forza in quanto permettono di essere notevolmente flessibili, ma allo stesso tempo è molto probabile che si verifichi il fenomeno dell'overfitting. Abbiamo detto che la tecnica di BN funge sotto un certo punto di vista da regularization, ma alcune volte ciò non basta ed è quindi necessario utilizzare delle tecniche più potenti. Una delle più comunemente utilizzate è il *Dropout.* L'idea di base è piuttosto semplice, ogni neurone ha una probabilità **p** di essere ignorato durante un training step, ma potrebbe essere attivo durante il prossimo step. **P** è un hyperparameter detto *dropout rate*, solitamente fissato tra il 10% - 50%. Questa tecnica rende la rete più robusta e meno sensibile a piccoli cambiamenti negli input, riuscendo così a generalizzare meglio. C'è solo un piccolo accorgimento da effettuare, quando viene settato **p**, ad esempio al 50%, allora i neuroni della rete saranno coinvolti al doppio delle connessioni, per bilanciare ciò bisogna moltiplicare tutti i pesi delle connessioni di un fattore 0.5 dopo il training, più in generale, dato **p,** bisogna moltiplicare per (1-p) i vari collegamenti o in modo del tutto uguale dividere ogni output dei neuroni per (1-p), ovvero la *keep probability*. Esistono anche ulteriori tecniche come la Max-Norm regularization, tutte con lo scopo di evitare che la rete overfitti. Un importante contributo alla buona riuscita di un modello di Deep Learning è il numero di hidden layers e di neuron in ogni layer, è quindi fondamentale scegliere un giusto numero per avere dei buoni risultati. In tal senso è molto rilevante trovare il giusto equilibrio fra parametri, regularization e learning rate affinché il sistema riesca a generalizzare nel modo migliore possibile. Alcune volte è possibile partire da sistemi che si occupano di problematiche simili e allenarli per altre applicazioni, questa procedura è chiamata **Transfer Learning.** Essa consiste appunto nel prendere un sistema adatto ad una determinata applicazione, ed usarlo per

una simile, più nel dettaglio quello si fa è rendere non allenabili i lower layers, poiché sono quelli che generalmente imparano i pattern meno specifici, ed allenare solo gli higher layers, utilizzando un learning rate piuttosto basso. Se si ottengono degli ottimi risultati, si può continuare così, altrimenti sarà necessario rendere allenabile qualche altro layer iterando questa procedura finché non si ottiene il risultato desiderato. Ovviamente il layer di input va rimpiazzato se le specifiche di input sono differenti e lo stesso vale anche per il layer di output. Anche con questa tecnica è importante possedere una elevata quantità di dati e di ottima qualità, in modo che il sistema possa generalizzare nel miglior modo possibile.

Capitolo 4

Computer Vision

Una delle principali sfide nella realizzazione di sistemi di machine learning è dotare questi sistemi dell'intelligenza visiva. Per noi esseri umani è facile riconoscere un oggetto, distinguerlo da un altro, così facile che descrivere come lo facciamo è davvero difficile. Per un computer invece un'immagine è solo una matrice di pixel e nulla di più. Lo scopo della **Computer Vision** è quello quindi di rendere questi pixel interpretabili per il sistema. Nel corso degli anni gli sviluppi in questo ambito sono stati incredibili ottenendo ottimi risultati, basti vedere applicazioni come *face verification* e *handwritten text recognition.* Per ottenere ciò si utilizzano delle speciali reti neurali, chiamate **Convolutional Neural Network,** spesso abbreviate con CCN. [8] Esse hanno la caratteristica di non essere più fully connected ed avere dei layer speciali detti Convolutional Layers che cercano di imitare il funzionamento della corteccia visiva. Si è notato che molti neuroni si occupano solo di un determinato **campo recettivo**, quindi reagiscono solo a determinati stimoli. Più neuroni posso condividere lo stesso campo recettivo, sovrapponendosi e ottenendo così una prospettiva più ampia. Si potrebbe pensare di utilizzare una deep neural network fully connected anche per queste applicazioni, ma la quantità di parametri richiesti, visto che i dati saranno tridimensionali (questo perché saranno immagini RGB,

quindi tre canali uno per ogni colore), è veramente spropositata. Mentre un approccio del tipo CNN riesce a diminuire questo numero. Infatti, ogni convolutional layer è collegato solo ad una piccola porzione dei neuroni vicini (inteso come posizione nella griglia) del precedente layer, ovvero al suo campo recettivo. Queste permette quindi di ridurre notevolmente i parametri del sistema e di avere una localizzazione delle feature nell'immagine, in quanto ogni neurone si occuperà solo di una determinata parte dell'immagine.

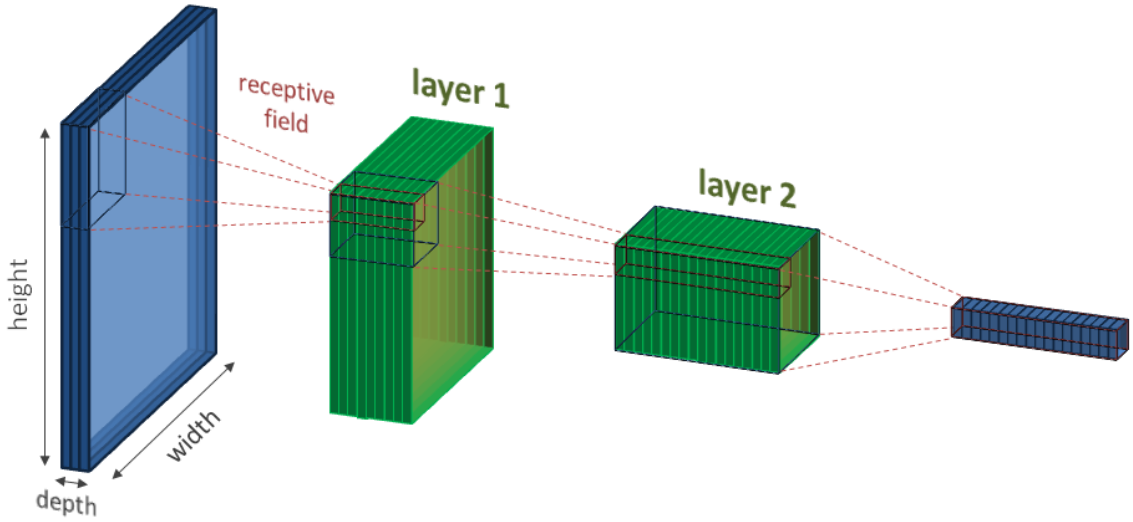


Figura 6 *– Illustrazione di un paio di convolutional layers, mostrando il campo recettivo da destra verso sinistra dei vari neuroni nei vari layers. [8]*

4.1 Convolutional Layers

Un convolutional layer è l'elemento centrale di una CNN, come detto i neuroni di un layer di CNN non sono fully connected, ma bensì sono connessi solo con i neuroni del loro campo recettivo del layer precedente. In particolare, il primo layer è connesso solo ai pixel del proprio campo

recettivo, a loro volta i neuroni del secondo layer sono connessi solo ad una porzione dei neuroni del primo livello. Quindi un neurone locato nella riga $\boldsymbol{i}$, colonna $\boldsymbol{j}$ di un certo layer è connesso agli output del layer precedente locati nelle righe i ad $i + f_h - 1$ e nelle colonne j a $j + f_w - 1$, dove f_w e f_h sono la larghezza e l'altezza del campo recettivo. Per ottenere layer delle stesse dimensioni dei layer precedenti si usa la tecnica dello **zero padding**, ovvero di aggiungere zeri sul bordo per ottenere la stessa dimensione. Questo è necessario poiché andando avanti nei layers, l'immagine diventa sempre più piccola.

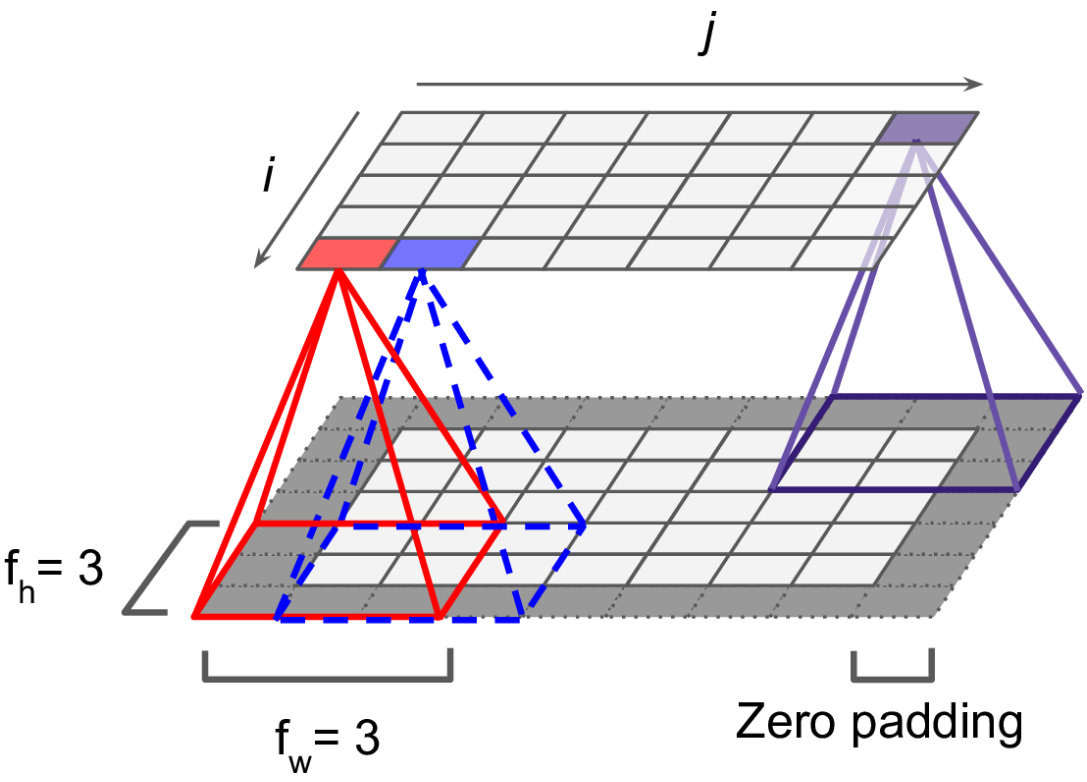


Figura 7 *– Input di 5x7 a cui viene applicato uno zero padding al solo bordo dell'immagine per renderla di una dimensione adatta. [3]*

È possibile connettere un input grande ad un layer più piccolo spaziando il suo campo recettivo, ovvero invece di muoversi un pixel per volta lo

farà di più pixel in base a quanto sia lo **stride.** come mostrato in figura 8. Fissato uno stride, un neurone situato nella riga ***i***, colonna ***j*** è connesso agli output del layer precedente locati dalla riga $i \times s_h$ alla $i \times s_h + f_h - 1$, e dalla colonna $j \times s_w$ alla $j \times s_w + f_w - 1$ dove s_w *e* s_h sono gli stride orizzontali e verticali, è possibile averne solo uno dei due o entrambi.

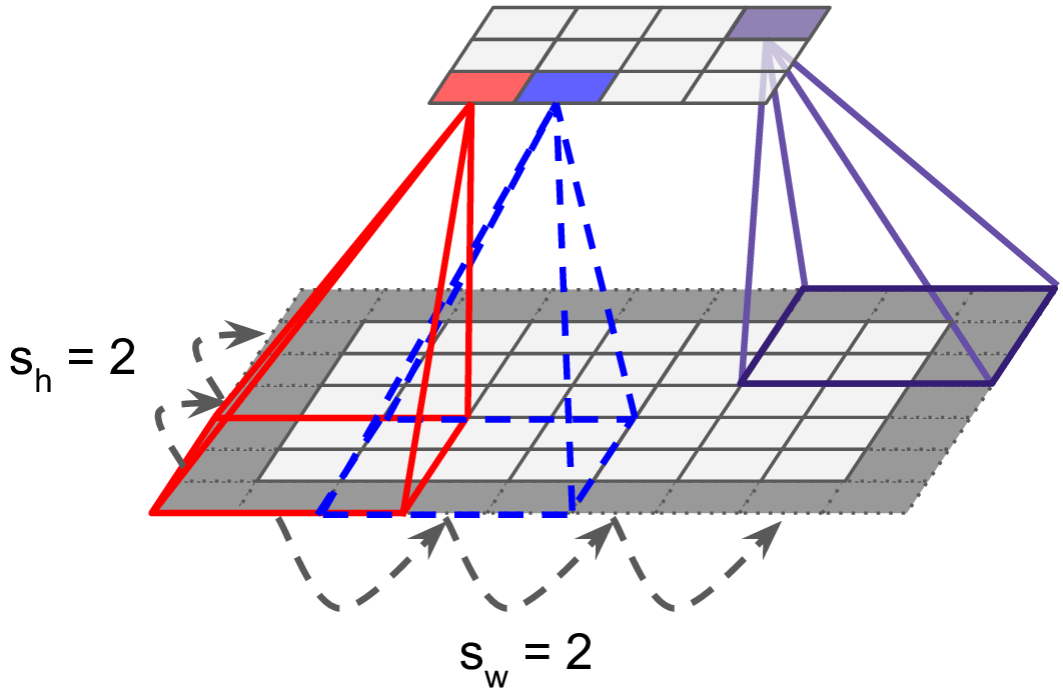


Figura 8 *– CNN in cui un input 5x7 è connesso ad un layer 3x4 usando campi recettivi 3x3 con stride pari a 2 in altezza e larghezza. Si può notare che si ottiene un'immagine più piccola in output dal layer rispetto alla figura 4.2. [3]*

Quella che svolge quindi un Convolutional Layer è a tutti gli effetti un'operazione di convoluzione, o in termini più matematici è un'operazione di correlazione, ma la intenderemo sempre come un'operazione di convoluzione fra l'input e i filtri. I filtri sono infatti la rappresentazione dei pesi, che sono multidimensionali in questo caso, rappresentati attraverso una matrice. L'operazione che si fa è quindi quella

di far scorrere il filtro su tutto l'input e calcolare la convoluzione. I filtri sono utilizzati per ricavare i pattern dell'immagine. Un layer che utilizza lo stesso filtro fornisce in output una *feature map*, cioè un'immagine in cui si evidenziano le aree in cui hanno attivato quei filtri. Per cui durante il training, i convolutional layer impareranno i filtri più utili per il task e i layer sovrastanti impareranno a combinarli per estrarre pattern più complessi. I neuroni di una certa feature map condividono gli stessi parametri, riducendone così il numero totale. Il campo recettivo è quindi l'estensione spaziale della connettività fra l'input e i neuroni localmente connessi, che ovviamente coincide con la dimensione del filtro. Questa rappresenta la proprietà della CNN di **sparse interaction,** ovvero di interazione sparsa. L'operazione di convoluzione è effettuata tra l'input ed il filtro lungo le tre dimensioni, ma la dimensione di profondità è pari al numero di filtri, lo si può immaginare come una convoluzione fra più matrici 2D (una per ogni livello di profondità) e il filtro, come si può notare nella figura 9.

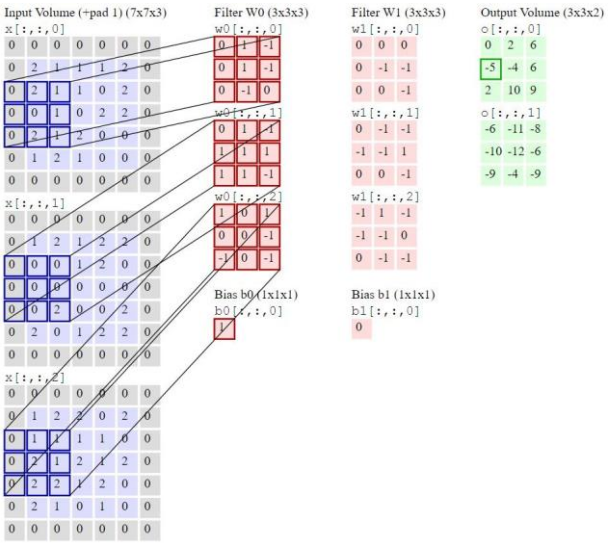


Figura 9 *- Operazione di convoluzione di due differenti filtri W0 e W1 di dimensione 3x3x3 su un volume di input 7x7x3. [9]*

I layer hanno più filtri, che indicheremo con N_f, tutti con la stessa dimensione. Durante il foward-pass ogni filtro è fatto convolvere lungo l'input, producendo ognuno una feature map. Per cui verranno prodotte N_f feature maps. Le quali concatenate lungo la terza dimensione producono l'output del layer, per cui la profondità dell'output è dipendente dal numero di filtri del layer. Tenendo conto di tutto ciò, l'equazione che permette il calcolo dell'output di un dato neurone in un layer è:

$$\boldsymbol{z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \times w_{u,v,k',k}}$$

dove:

- $i' = i \times s_h + u$;
- $j' = j \times s_w + v$
- $z_{i,j,k}$ è l'ouput del neurone della riga i, colonna j della feature map k del layer
- s_h e s_w sono gli stride orizzontali e verticali, $f_h e$ f_w sono l'altezza e la larghezza del campo recettivo ed $f_{n'}$ è il numero di feature map del layer precedente
- $\boldsymbol{x_{i',j',k'}}$ è l'output del neurone del layer precedente nella riga i', colonna j', feature map k'
- $\boldsymbol{w_{u,v,k',k}}$ è il peso della connessione tra ogni neurone della feature map k del layer e il suo input della riga u, colonna v e feature map k
- $\boldsymbol{b_k}$ è il bias term per la feature map k

Da questa formula si può notare che i neuroni collegati alle stesse feature map, condividono gli stessi parametri, ossia i pesi, infatti essi non dipendono da (i ,j), ma bensì dalle feauture maps K. Per cui si è dimostrato che permettono di ridurre notevolmente il numero dei parametri del sistema.

4.2 Pooling Layers

Un altro elemento fondamentale di una CNN è il **Pooling Layer**, il suo scopo è quello di ridurre la dimensione degli input del prossimo Convolutional Layer, così da diminuire il numero di parametri e l'onere computazionale della rete. Il pooling layer opera su ogni feature map applicando un filtro che esegue un'operazione deterministica, che solitamente è la media o il massimo, per cui non c'è la presenza di pesi. Il pooling layer garantisce la proprietà delle CNN di intolleranza alle piccole traslazioni, rendendo, infatti, la somiglianza tra due immagini molto più evidente. In alcuni casi però questa riduzione non va bene poiché diminuisce notevolmente le feature dell'immagine in analisi ed in alcune applicazioni un piccolo cambiamento dell'input deve corrispondere ad una variazione dell'output, come ad esempio in applicazioni di semantic segmentation (che consiste nella classificazione di ogni pixel di un'immagine).

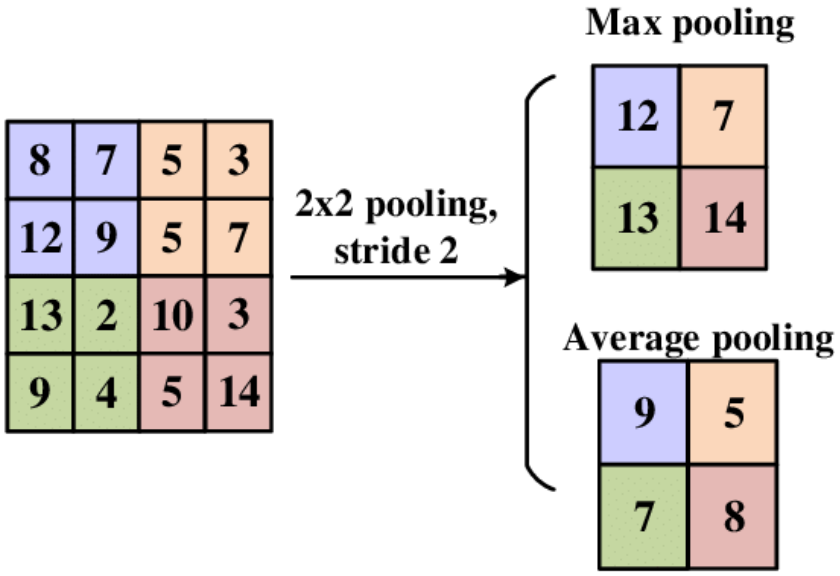


Figura 10 *– Esempio di Max Pooling e Average Pooling con filtro 2x2 applicato ad un input 4x4. [10]*

4.3 Architettura Generica Di Una Convolutional Neural Network

Dopo aver introdotto le due componenti principali di una CNN, vediamo ora un esempio di architettura generica. La struttura è piuttosto semplice, come si può vedere dalla figura 11.

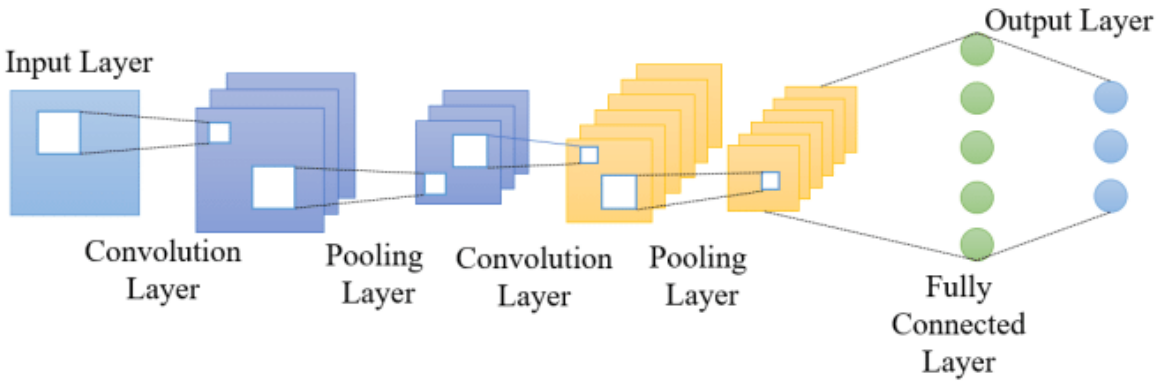


Figura 11 *– Architettura generica di una Convolutional Neural Network [11]*

Abbiamo un susseguirsi di Convolutional Layers e Pooling layers, seguiti poi da una fully connected network. L'immagine procedendo nella rete diventa via via sempre più piccola, ma vengono estratte feature sempre più complesse. Queste vengono poi elaborate dalla fully connected network che darà l'output finale del sistema. È importante scegliere la dimensione dei filtri in base all'input; quindi, man mano le dimensioni dei filtri nei layer diminuiranno, poiché la dimensione dell'input diminuirà. Inoltre, gli higher layers avranno un numero maggiori di filtri rispetto ai lower layers, cosicché possano essere estratte feauture più complesse. Esistono molte architetture di CNN, come ad esempio LeNet-5, AlexNet ed altre. Molte volte, infatti, si utilizzano dei modelli pre-allenati, come le architetture sopraelencate, ed utilizzate delle tecniche di transfer learning, piuttosto che partire da una rete nuova di zecca.

4.4 Data Augmentation

La **Data Augmentation** è una tecnica utilizzata spesso in applicazioni di computer vision per aumentare la mole di dati presenti in un dataset, questo poiché spesso la quantità di dati non è sufficiente. Questa tecnica consiste nel creare delle copie leggermente differenti delle istanze del dataset, ad esempio ruotando leggermente l'immagine, traslarla di qualche pixel ed altre modifiche simili. Ovviamente bisogna prestare attenzione a quali trasformazioni si adottano, poiché alcune di esse potrebbero non essere adatte all'applicazione, ad esempio in un compito di classificazione delle immagini ritraenti numeri, ruotare la cifra di 180° è una problematica

(in quanto trasformerebbe il ‘6’ in ‘9’ o viceversa) e non un ulteriore istanza del dataset.

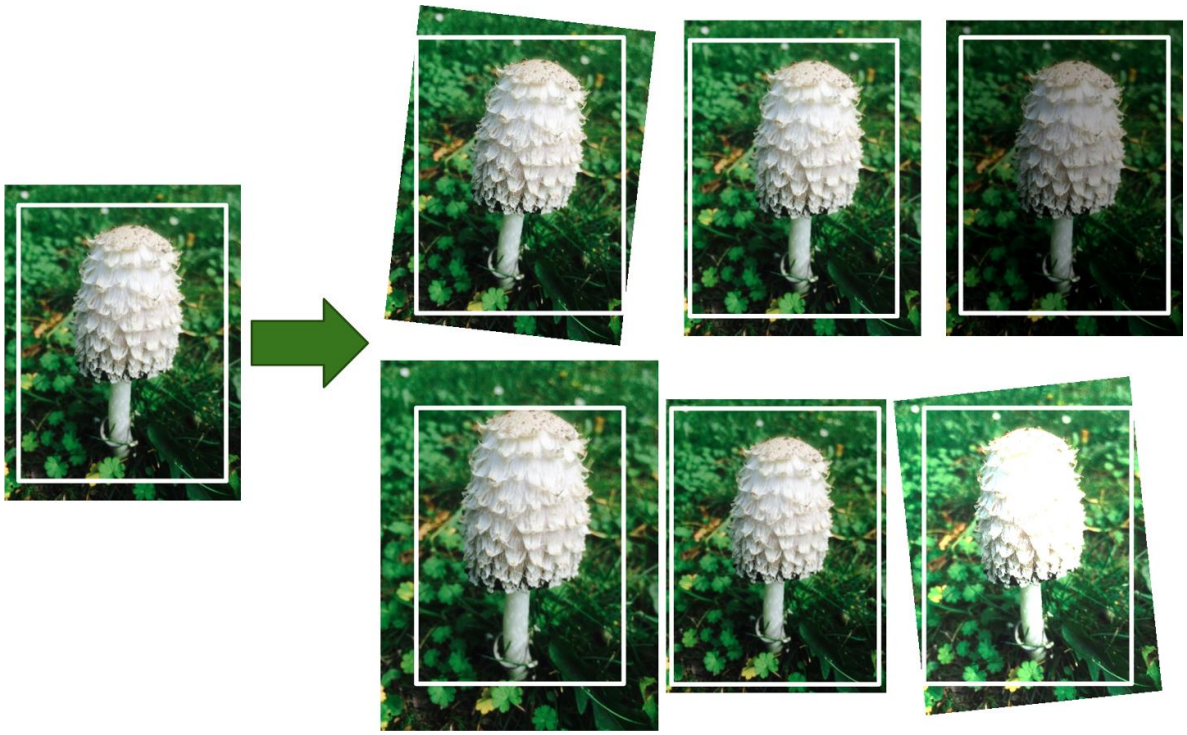


Figura 12 *- Esempio della creazione di nuove istanze di training tramite la tecnica di Data Augmentation. [3]*

Capitolo 5

Autoencoder

5.1 Apprendimento Non Supervisionato

L' **Unsupervised Learning** è una tipologia di algoritmi di Machine Learning che imparano senza una supervisione, i dati sopra cui lavorano sono infatti grezzi, senza alcuna etichettatura. Per cui l'algoritmo è forzato ad imparare le feature e i pattern dei dati in modo autonomo. Il vantaggio di questa tipologia di algoritmi è che non hanno bisogno di molta lavorazione sui dati, che sono solitamente operazioni molto dispendiose e molte volte non automatizzabili. Di contro però, richiedono una elevata quantità di dati ed in genere i costi computazionali sono maggiori. Algoritmi unsupervised vengono spesso utilizzati per applicazioni di **Clustering** o **Anomaly Detection**. Un algoritmo di **Clustering** cerca di classificare i dati in vari sottogruppi, quindi di trovare similarità tra i dati. Un algoritmo di **Anomaly Detection** invece cerca di indentificare quelle istanze dei dati che si discostano notevolmente da tutte le altre, e che potrebbero essere quindi anomale. Una famiglia di algoritmi utilizzati per l'anomaly detection è quella degli autoencoder, durante il loro training cercano di replicare nel loro output l'input in modo da poter impararne i pattern.

5.2 Autoencoder

Gli **Autoencoder** sono reti neurali in grado di imparare rappresentazioni complesse dell'input, dette **codings**, sono degli algoritmi unspervised, per cui i loro input sono senza labels. I codings hanno solitamente una dimensione minore rispetto all'input, infatti sono spesso utilizzati per applicazioni di **dimensionality reduction**. Gli autoencoder cercano di replicare nel loro output l'input ricevuto, ma limitando la rete in alcuni aspetti, costringendola così a dover trovare un modo efficace per fare ciò, riconoscendo pattern e non semplicemente copiando l'input. Più nello specifico, i codings sono una riproduzione dell'**encoder**, una delle due parti degli autoencoder, imparando la **identity function** dell'input sotto alcune restrizioni. Per cui un autoencoder visualizza un input, lo converte in una **latent representation,** ovvero una rappresentazione di ciò che la rete ha imparato, dopodiché da questa rappresentazione cerca di ricostruire l'input nel modo più fedele possibile, questa seconda parte è svolta dalla seconda componente dell'autoencoder, chiamata **decoder.** Un autoencoder è quindi composto da un **encoder**, che converte l'input in una latent representation ed un **decoder** che da questa rappresentazione ne ottiene un output.

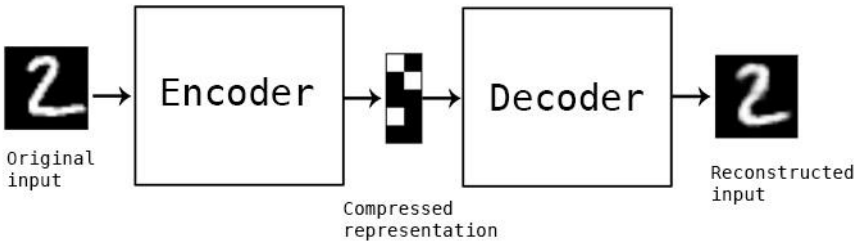


Figura 13 - Architettura di un autoencoder, nelle due sue parti encoder e decoder. [12]

La struttura di un autoencoder a livello di layers è molto simile a quella di una *Multilayer perceptron*, con l'unica differenza che gli output neurons sono in numero uguale agli input neurons. Questo genere di autoencoder è detto **stacked autoencoder** e si può notare dalla figura 14 che la loro struttura è simmetrica rispetto al **coding layer.**

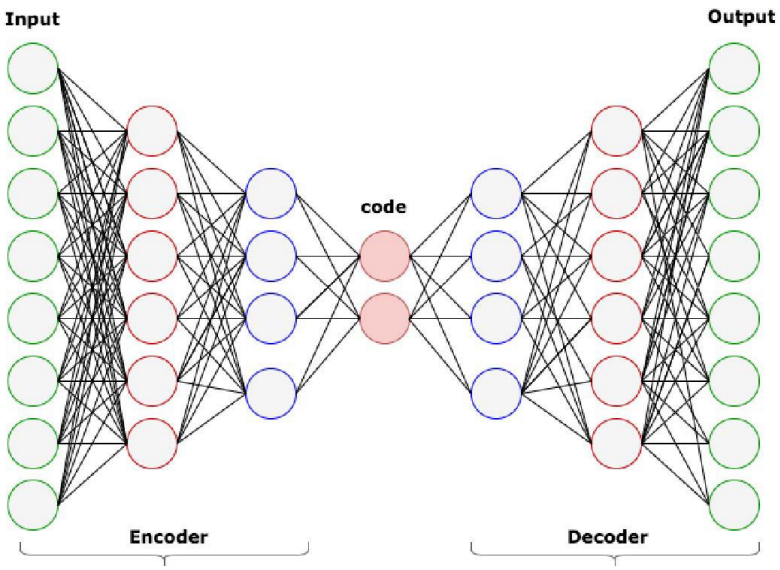


***Figura 14** - Struttura generica di uno Stacked Autoencoder, notare la simmetria della struttura rispetto al coding layer. [13]*

Più hidden layers sono presenti, più il sistema sarà capace di produrre coding complessi e quindi più rappresentativi dell'input, ma bisogna

sempre evitare l'overfitting, altrimenti il sistema non generalizzerà bene. Vista la sua natura simmetrica, una tecnica utilizzata spesso per velocizzare il processo di training è quella di legare i pesi dei layer simmetrici, ovvero il peso di un layer dell'encoder è legato al peso del corrispettivo layer del decoder.

5.3 Convolutional Autoencoder

Quando ci si ritrova a lavorare con immagini, nell'ambito del deep learning, si preferisce utilizzare i convolutional layers, che permettono di avere una maggiore efficienza e un costo computazionale minore. Allo stesso modo, i Convolutional Autoencoder permettono di avere gli stessi vantaggi.

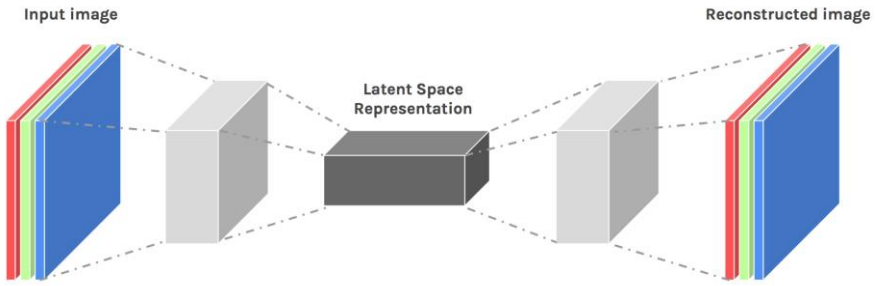


Figura 15 *- Esempio di generica architettura di un Convolutional Autoencoder. [14]*

Anche in questo caso notiamo la struttura simmetrica rispetto al coding layer, dove è presente la latent representation dell'input, ma questa volta sarà necessaria una piccola modifica rispetto ad una normale CNN nel decoder. Le CNN hanno la peculiarità, infatti, di ridurre l'estensione

spaziale dell'input (inteso come larghezza e altezza) ed aumentare la profondità, cioè le feature map, di conseguenza la latent representation avrà dimensioni più piccole. Per tornare alle dimensioni originali quindi un normale convolutional layer nel decoder non va bene, si utilizzano infatti dei convolutional layer trasposti, che effettuano una operazione di convoluzione inversa. Un' alternativa possibile è anche quella di utilizzare i normali convolutional layer, ma invece dell'operazione di MaxPooling, si effettua una operazione di Up Sampling, la cui funziona è quella di raddoppiare la dimensione del suo input [15].

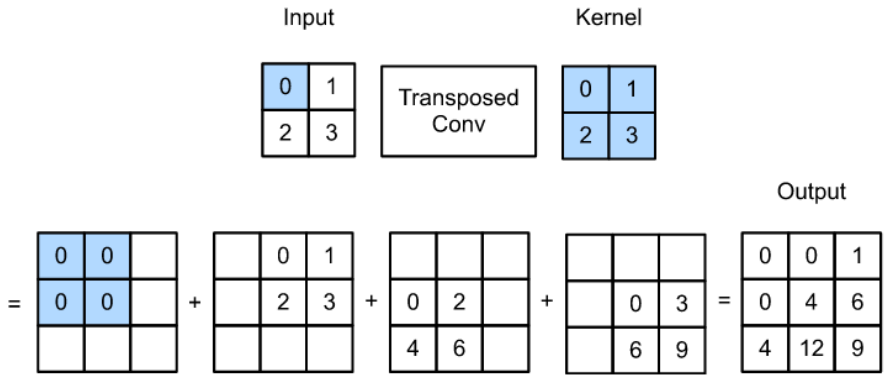


Figura 16 *- Esempio di operazione di convoluzione inversa tra un input 2x2 e un filtro 2x2 che da un output di dimensione 3x3. [16]*

5.4 Anomaly Detection utilizzando autoencoder

Gli autoencoder vengono allenati cercando di minimizzare il più possibile la **reconstruction loss,** ovvero la differenza tra l'input e l'output ricostruito. Si può sfruttare questa peculiarità per utilizzare gli autoencoder per il rilevamento di anomalie. Allenando la rete sui soli dati

normali, la rete imparerà a ricostruire quest'ultimi con un certo errore, quando invece si è in presenza di un'istanza anomala, l'errore di ricostruzione sarà maggiore. Calcolando il valore medio di questo errore di ricostruzione durante l'allenamento della rete, lo si può utilizzare come una soglia, che permetterà di discriminare tra un'istanza normale ed una anomala. Confrontando l'errore di ricostruzione di una data istanza con la soglia, infatti, se esso è maggiore della soglia sarà un'istanza anomala, altrimenti normale. Questo viene ancora di più enfatizzato nell'utilizzo dei convolutional autoencoder e permette di sviluppare algoritmi di anomaly detection attraverso l'analisi delle immagini. [17]

Capitolo 6

Rilevamento delle anomalie nei componenti elettronici

Il nostro campo di applicazione è quello del rilevamento di anomalie nei componenti elettronici, in particolare nei Transistor. Le anomalie nei transistor possono essere di vario genere, noi ci occuperemo solo di quelle che posso essere riconoscibili visivamente come ad esempio il case danneggiato, uno o più pin rotti o piegati ed altre problematiche simili. Per il training ed il test del nostro algoritmo utilizzeremo il dataset della MVTEC AD [18].

6.1 Il Dataset

Il dataset generale contiene 15 categorie per un totale di 3629 immagini, la parte di nostro interesse è la sola categoria dei transistor, composta da 313 immagini RGB, con dimensioni di 1024x1024 pixels. Il dataset della nostra categoria è già suddiviso in due parti, quella di train set e quella di test set. Il train set contiene 213 immagini RGB tutte normali come quella in figura 17.

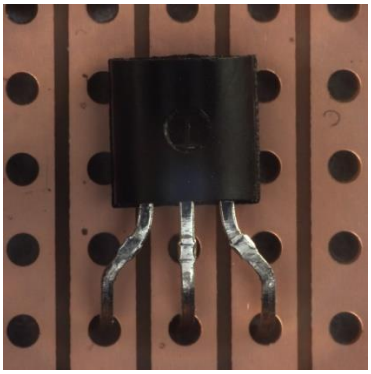
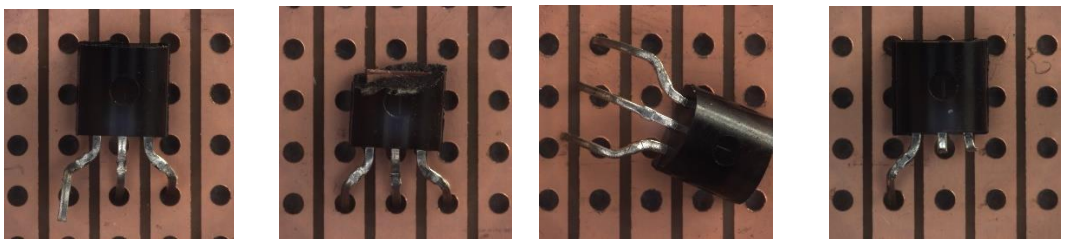


Figura 17 *- Un'istanza del train set con un transistor in buone condizioni.*

Il test set contiene 100 immagini RGB miste fra normali ed anomale, in particolare 60 sono normali e 40 sono anomale, come mostrato nelle figure 17 e 18. Nel dataset le anomalie sono suddivise per tipologia come: case danneggiato, mal posizionamento, pin rotto e pin scollegato; tuttavia, a noi interessa solo la distinzione tra un transistor anomalo avente uno o più dei problemi già citati ed uno nomale (case e pin integri, corretto posizionamento).



a **b** **c** **d**

Figura 18 *- Nella figura **a** è presente un transistor con un pin non connesso correttamente. Nella figura **b** un transistor con il case danneggiato. Nella figura **c** un transistor mal posizionato e nella figura **d** un transistor con due pin rotti.*

6.2 Costruzione Del Modello

Andiamo adesso ad analizzare il nostro autoencoder utilizzato per il rilevamento di anomalie nei transistor. Per la costruzione del modello abbiamo utilizzato python, il framework TensorFlow [19] e le API Keras [20].

```
input_layer = Input(shape=(128, 128, 3), name="INPUT")
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_layer)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same', name="CODE")(x)

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(16, (3, 3), activation='relu',padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu',padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu',padding='same')(x)
x = UpSampling2D((2, 2))(x)
output_layer= Conv2D(3, (3, 3), activation='linear', padding='same')(x)
```

Analizziamo nel dettaglio ogni layer, partendo dall'encoder:

- Un input layer di dimensione 128x128x3
- Un primo convolutional layer con 32 filtri di dimensione 3x3, activation function ReLu e zero padding
- Un MaxPooling layer di dimensione 2x2
- Un secondo convolutional layer con 32 filtri di dimensione 3x3, activation function ReLu e zero padding
- Un MaxPooling layer di dimensione 2x2
- Un terzo convolutional layer con 16 filtri di dimensione 3x3, activation function ReLu e zero padding
- Un quarto convolutional layer con 8 filtri di dimensione 3x3, activation function ReLu e zero padding
- Un MaxPooling layer di dimensione 2x2
- Un quinto convolutional layer con 8 filtri di dimensione 3x3, activation function ReLu e zero padding
- Un MaxPooling layer di dimensione 2x2, dove risiederà il coding dell'input

Il decoder sarà simmetrico:

- Un primo convolutional layer con 8 filtri di dimensione 3x3, activation function ReLu e zero padding
- Un UpSampling layer di dimensione 2x2
- Un secondo convolutional layer con 8 filtri di dimensione 3x3, activation function ReLu e zero padding

- Un terzo convolutional layer con 16 filtri di dimensione 3x3, activation function ReLu e zero padding
- Un UpSampling layer di dimensione 2x2
- Un quarto convolutional layer con 32 filtri di dimensione 3x3, activation function ReLu e zero padding
- Un UpSampling layer di dimensione 2x2
- Un quinto convolutional layer con 32 filtri di dimensione 3x3, activation function ReLu e zero padding
- Un UpSampling layer di dimensione 2x2
- Un output layer che è un convolutional layer con 3 filtri di dimensione 3x3, con activation function linear e zero padding

Compiliamo il modello, con il metodo .compile() [21]:

```
transistor_AE = Model(input_layer, output_layer)
transistor_AE.compile(optimizer='adam', loss='mse')
```

Utilizziamo come ottimizzatore adam, il quale provvederà ad aggiornare i pesi delle connessioni e loss function la MSE, così da calcolare la "distanza" tra l'immagine ricostruita e quella di input. Alla fine di questa procedura, il modello compilato è risultato avere 33,603 parametri allenabili.

6.3 Training Del Modello:

Una volta compilato il modello, possiamo allenarlo sul training set, chiamando il metodo .fit() [21]:

```
history = transistor_AE.fit(transistor_train, transistor_train,
                epochs=1000,
                batch_size=16,
                shuffle=True,
                callbacks= EarlyStopping(monitor = "val_loss", min_delta
= 0.0001, patience = 30, mode = "min"),
                validation_split=0.15
               )
```

Passiamo come dati solamente il train set essendo questo utilizzato anche come label ai fini del training. Fissiamo come *batch_size* 16 e la permutazione casuale del training set ad ogni epoca (*shuffle=True*). Il *validation_split* è stato settato a 0.15, così che il 15% del train set venga utilizzato come validation set. Utilizziamo, inoltre la callback **EarlyStopping,** in modo tale da fissare un numero elevato di epoche senza doversene preoccupare, il sistema finirà automaticamente il suo training quando una certa metrica smetterà di migliorare. Nel nostro caso fissiamo come metrica la *val_loss*, ovvero la loss function sul validation set che non dovrà migliorare più del *min_delta* per 30 epoche. Il modello ha terminato l'allenamento dopo 242 epoche, ottenendo in quest'ultima una loss di 0.0032 ed una val_loss di 0.0032. Visualizzando le **learning curves,** ovvero una rappresentazione grafica delle performance del sistema nel tempo (inteso come epoche)

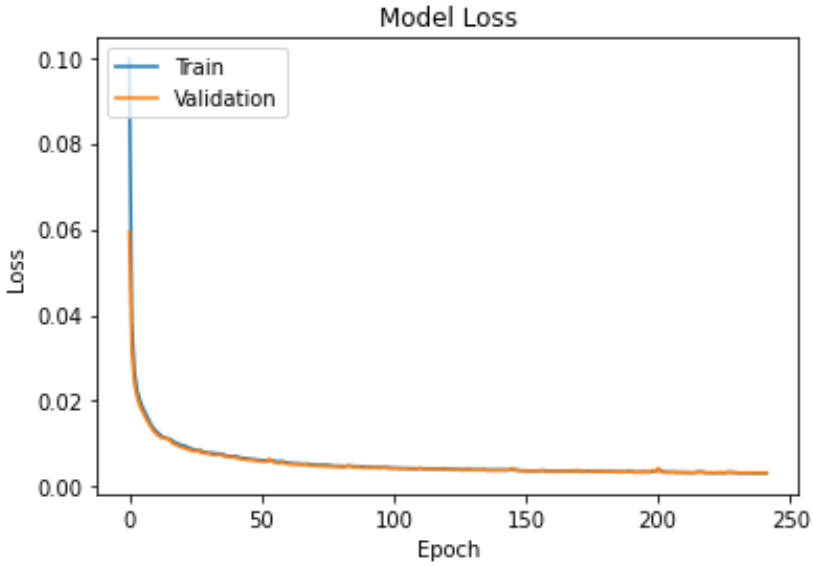


Figura 19 *- Learning Curve del nostro sistema.*

Si evince che il sistema riesce a generalizzare bene essendo l'errore nel validation set nello stesso range di quello del training, per cui non overfitta. Possiamo inoltre valutare la qualità del modello visualizzando le immagini originali e le relative immagini ricostruite come mostrato in figura 20.

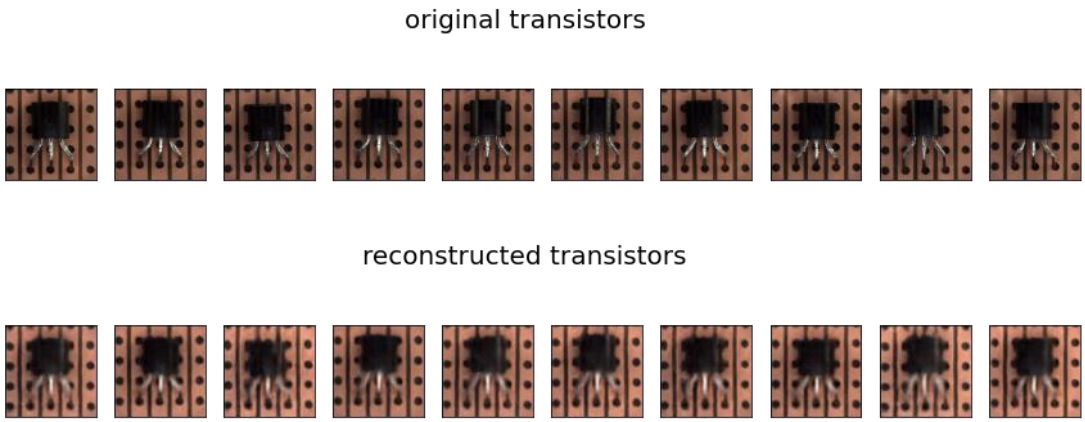


Figura 20 *- In alto le immagini originali del dataset, in basso le relative immagini ricostruite.*

6.4 Rilevamento delle Anomalie

Per rilevare le anomalie, computiamo l'MSE compiuto durante il training, che chiameremo *train_loss:*

```
reconstructed_transistor = transistor_AE.predict(transistor_train)

train_loss            =            mean_squared_error(transistor_train,
reconstructed_transistor)
```

Fissiamo adesso la soglia come la somma tra la media del *train loss* e la mediana dello stesso:

```
threshold =  np.mean(train_loss) + np.median(train_loss)
```

Questa soglia avrà il valore di:

0.0038157622

Computiamo l'MSE compiuto sul test set, quest'ultimo è stato diviso in due parti, una contenenti le sole immagini normali ed una le sole immagini anomale. Computando l'MSE per entrambi le parti, abbiamo ottenuto un valore di **0.0033998876** sui dati normali e di **0.0052148905** sui dati anomali. Si noti come l'errore di ricostruzione medi dei dati anomali sia

più grande di quello sui dati normali, il che permette di utilizzarlo per effettuare l'anomaly detection.

```
reconstructions = transistor_AE.predict(transistor_test_good)
test_loss_good                                                  =
mean_squared_error(transistor_test_good,reconstructions)
labels_predict0 =[]

#anomaly detection sulle normali
for i in range(len(test_loss_good)):
    tmp = np.mean(test_loss_good[i])
    if tmp >= threshold:
        print(i+1,"Anomaly")
        labels_predict0.append([1])
    else:
        print(i+1,"Normal")
        labels_predict0.append([0])
```

Si effettua infine il confronto tra il valore medio dell'errore e la soglia calcolata. Se è maggiore, l'immagine sarà rilevata come anomala, altrimenti come normale.

6.5 Risultati

Valutiamo adesso le prestazioni del sistema attraverso alcune metriche. La prima che prediamo in considerazione è la **confusion matrix** [22], essa ci dirà quante istanze della classe normale vengono classificate come classe anomala e viceversa. Ogni riga rappresenta una classe ed ogni colonna rappresenta la classe predetta. Nel nostro caso la classe positiva è quella

delle immagini anomale (seconda riga), quella negativa quella delle immagini normali (prima riga). La confusion matrix ottenuta è la seguente:

$$\begin{bmatrix} 49 & 11 \\ 16 & 24 \end{bmatrix}$$

Sulle 60 immagini normali del test set, il nostro modello ne classifica 49 come normali, dette **true negatives**, e 11 come anomale, dette **false positives.** Per una correttezza di classificazione del 81.6% circa**.** Sulle 40 immagini anomale, 16 vengono classificate come normali, dette **false negatives**, e 24 come anomale, dette **true positives.** Per una correttezza di classificazione del 60%. In totale sopra 100 immagini, ne vengono classificate correttamente 73, per cui una accuratezza totale del 73%.

Dalla confusion matrix possiamo calcolarci delle altre metriche molto interessanti. La prima detta **Precision**, è l'accuratezza del sistema nel non classificare come positiva un'istanza negativa.

$$precision = \frac{TP}{TP + FP} = \frac{24}{24 + 11} = 0.6857142857142857$$

La seconda è detta **Recall** ed è la precisione del classificatore di trovare tutte le istanze positive.

$$recall = \frac{TP}{TP + FN} = \frac{24}{24 + 16} = 0.6$$

Queste due metriche possono essere combinate, per ottenerne un'altra chiamata $\boldsymbol{F_1 score}$, definita come la media armonica della precision e recall.

$$F_1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}} = \frac{2}{\frac{1}{0.6857142857142857} + \frac{1}{0.6}} = 0.64$$

6.6 Conclusioni

Ricapitolando, il nostro sistema ottiene un'accuratezza dell'81.6% circa sulle immagini normali del test set e del 60% sulle immagini anomale. Sono dei risultati comunque ragionevoli, soprattutto se messi a paragone con altri studi effettuati utilizzando lo stesso approccio, come ad esempio quello effettuato dalla stessa MVTec AD [23], ha ottenuto un 97% sulle istanze normali, ma un 45% sulle anomale. Per cui il nostro sistema performa peggio sulle istanze normali, ma decisamente meglio sulle istanze anomale avendo una migliore accuratezza. Ad un primo sguardo può sembrare che i valori delle metriche di recall o $f_1 score$ siano basse, ma sono in realtà in linea con quelli ottenuti da altre tecniche simili. Si è quindi provato che anche gli autoencoder possono essere utilizzati nell'ambito dell'anomaly detection, con delle buone performance. Si potrebbe potenziare il modello utilizzando una maggiore quantità di dati (sia reali, sia che campionati), in modo da rendere lo stesso più capace di generalizzare correttamente ed essere in grado di distinguere meglio le due classi. Ovviamente con una maggiore quantità di dati andrebbe anche potenziato il modello stesso, cambiando la sua architettura in termini di layer e neuroni

Bibliografia

[1] «Anomaly Detection,» [Online]. Available: https://en.wikipedia.org/wiki/Anomaly_detection.

[2] T. Mitchell, Machine Learning, McGraw Hill, 1997.

[3] A. Geron, Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems, Oreilly & Associates Inc, 2019.

[4] Y. B. A. C. Ian Goodfellow, Deep Learning, The MIT Press, 2016.

[5] [Online]. Available: https://www.codenong.com/cs109675430/.

[6] M. Chaudhary, «Activation Functions: Sigmoid, Tanh, ReLU, Leaky ReLU, Softmax,» 2020. [Online]. Available: https://medium.com/@cmukesh8688/activation-functions-sigmoid-tanh-relu-leaky-relu-softmax-50d3778dcea5.

[7] [Online]. Available: https://mathworld.wolfram.com/HyperbolicTangent.html .

[8] E. A. Benjamin Planche, - Hands-On Computer Vision with TensorFlow 2: Leverage deep learning to create powerful image processing apps with TensorFlow 2.0 and Keras, Packt Publishing, 2019.

[9] «Parameters and flops in convolutional neural network CNN,» [Online]. Available: https://chowdera.com/2021/04/20210420120752555v.html.

[10] H. Y. A. A.-Y. Lee, «Deep Neural Networks on Chip - A Survey,» [Online]. Available: https://www.researchgate.net/publication/340812216_Deep_Neural_Networks_on_Chip_-_A_Survey/figures.

[11] H. G. W. H. Gui, «Blind Channel Identification Aided Generalized Automatic Modulation Recognition Based on Deep Learning,» [Online]. Available: https://www.researchgate.net/publication/335086346_Blind_Channel_Identification_Aided_Generalized_Automatic_Modulation_Recognition_Based_on_Deep_Learning/figures?lo=1.

[12] «Building Autoencoders in keras,» [Online]. Available: https://blog.keras.io/building-autoencoders-in-keras.html.

[13] M. B. Batouche, «Deep semi-supervised learning for DTI prediction using large datasets and H2O-spark platform,» [Online]. Available: https://www.researchgate.net/publication/325025951_Deep_semi-supervised_learning_for_DTI_prediction_using_large_datasets_and_H2O-spark_platform/figures?lo=1.

[14] S. I. Serengil, «Convolutional Autoencoder: Clustering Images with Neural Networks,» [Online]. Available: https://sefiks.com/2018/03/23/convolutional-autoencoder-clustering-images-with-neural-networks/.

[15] Tensorflow, «tf.keras.layers.Conv2DTranspose,» [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose.

[16] D. i. d. learning, «Transposed Convolution,» [Online]. Available: https://d2l.ai/chapter_computer-vision/transposed-conv.html.

[17] Tensorflow, «Autoencoder,» [Online]. Available: https://www.tensorflow.org/tutorials/generative/autoencoder.

[18] «MVTEC ANOMALY DETECTION DATASET,» [Online]. Available: https://www.mvtec.com/company/research/datasets/mvtec-ad.

[19] «Tensorflow,» [Online]. Available: https://www.tensorflow.org/.

[20] «Keras,» [Online]. Available: https://keras.io/.

[21] Keras, «Model Training API,» [Online]. Available: https://keras.io/api/models/model_training_apis/.

[22] J. Brownlee, «What is a Confusion Matrix in Machine Learning,» [Online]. Available: https://machinelearningmastery.com/confusion-matrix-machine-learning/.

[23] P. B. M. F. D. S. C. Steger, «MVTec AD — A Comprehensive Real-World Dataset for Unsupervised,» 2019. [Online]. Available: https://www.mvtec.com/fileadmin/Redaktion/mvtec.com/company/research/datasets/mvtec_ad.pdf.

Appendice

Codice della rete per intero

```
import numpy as np
import pandas as pd
import os
import matplotlib.pyplot as plt
from tensorflow import keras
from keras.layers import Input, Dense, Conv2D, MaxPooling2D,
UpSampling2D, Conv2DTranspose, Reshape
from keras.models import Model
from keras.preprocessing import image
from keras.callbacks import EarlyStopping
from keras.losses import mean_squared_error, mae
import tensorflow as tf
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from sklearn.metrics import confusion_matrix ,
precision_recall_fscore_support
#loading del training set

transistor_path = "C:/transistor/train/good/"

transistor_train = []
for filename in os.listdir(transistor_path):
     if filename.endswith(".png"):
        img = image.load_img(transistor_path+filename,
target_size=(128, 128))
        transistor_train.append(image.img_to_array(img))
transistor_train = np.array(transistor_train)
transistor_train = transistor_train/255
print ("transistor_train", transistor_train.shape)
#loading del test set good
transistor_test_good_path = "C:/transistor/dtest/good/"
```

```
transistor_test_good = []

for filename in os.listdir(transistor_test_good_path):
     if filename.endswith(".png"):
        img = image.load_img(transistor_test_good_path+filename,
target_size=(128, 128))
        transistor_test_good.append(image.img_to_array(img))
transistor_test_good = np.array(transistor_test_good)

transistor_test_good = transistor_test_good/255

print("transistor_test_good", transistor_test_good.shape)
#loading del test set notgood
transistor_test_notgood_path = "C:/transistor/dtest/notgood/"

transistor_test_notgood = []

for filename in os.listdir(transistor_test_notgood_path):
     if filename.endswith(".png"):
        img = image.load_img(transistor_test_notgood_path+filename,
target_size=(128, 128))
        transistor_test_notgood.append(image.img_to_array(img))
transistor_test_notgood = np.array(transistor_test_notgood)

transistor_test_notgood = transistor_test_notgood/255

print("transistor_test_notgood", transistor_test_notgood.shape)

#funzioni per la stampa delle immagini
def show_transistor_data(X, n=10, title=""):
    plt.figure(figsize=(15, 5))
    for i in range(n):
        ax = plt.subplot(2,n,i+1)
        plt.imshow(image.array_to_img(X[i]))
        ax.get_xaxis().set_visible(False)
        ax.get_yaxis().set_visible(False)
```

```
    plt.suptitle(title, fontsize = 20)
#labels

labels0= np.zeros(60, dtype=int)
labels1= np.ones(40, dtype=int)
labels= np.concatenate([labels0, labels1])
#creazione modello

input_layer = Input(shape=(128, 128, 3), name="INPUT")
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_layer)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same', name="CODE")(x)

x = Conv2D(8, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = Conv2D(16, (3, 3), activation='relu',padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu',padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu',padding='same')(x)
x = UpSampling2D((2, 2))(x)
output_layer= Conv2D(3, (3, 3), activation='linear', padding='same')(x)

transistor_AE = Model(input_layer, output_layer)
transistor_AE.compile(optimizer='adam', loss='mse')
transistor_AE.summary()
history = transistor_AE.fit(transistor_train, transistor_train,
                epochs=1000,
                batch_size=16,
```

```
                shuffle=True,
                callbacks= EarlyStopping(monitor = "val_loss",
min_delta = 0.0001, patience = 30, mode = "min"),
                validation_split=0.15
               )

#learning curves
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Validation'], loc='upper left')
plt.show()

#stampa dei risultati

transistor_AE.save("transistor_AE.h5")


reconstructed_transistor = transistor_AE.predict(transistor_train)
show_transistor_data(transistor_train, title="original transistors")
show_transistor_data(reconstructed_transistor, title="reconstructed
transistors")
#calcolo soglia

reconstructed_transistor = transistor_AE.predict(transistor_train)

train_loss = mean_squared_error(transistor_train,
reconstructed_transistor)

threshold =  np.mean(train_loss) + np.median(train_loss)
print("Threshold: ",threshold)

reconstructions = transistor_AE.predict(transistor_test_good)
test_loss_good =
mean_squared_error(transistor_test_good,reconstructions)
```

```
labels_predict0 =[]

#anomaly detection sulle normali
for i in range(len(test_loss_good)):
    tmp = np.mean(test_loss_good[i])
    if tmp >= threshold:
        print(i+1,"Anomaly")
        labels_predict0.append([1])
    else:
        print(i+1,"Normal")
        labels_predict0.append([0])



reconstructions = transistor_AE.predict(transistor_test_notgood)
test_loss_notgood =
mean_squared_error(transistor_test_notgood,reconstructions)
labels_predict1 =[]

#anomaly detection sulle anomali
for i in range(len(test_loss_notgood)):
    tmp = np.mean(test_loss_notgood[i])
    if tmp >= threshold:
        print(i+1,"Anomaly")
        labels_predict1.append([1])
    else:
        print(i+1,"Normal")
        labels_predict1.append([0])



labels_predict= np.concatenate([labels_predict0,
labels_predict1])

#Anomalie 1, Normali 0
conf_mx = confusion_matrix(labels, labels_predict)
conf_mx
```

```
precision_recall_fscore_support(labels, labels_predict,
average='binary')
plt.matshow(conf_mx , cmap=plt.cm.gray)
plt.show()
```