Università degli Studi di Padova
Dipartimento di Scienze Statistiche
Corso di Laurea Triennale in

STATISTICA PER L'ECONOMIA E L'IMPRESA

RELAZIONE FINALE

Deep learning per la classificazione di immagini: un approfondimento sulle reti neurali profonde

Relatore Prof. Erlis Ruli
Dipartimento di Scienze Statistiche

Laureando: Francesco Lollato
Matricola N 2009248

Anno Accademico 2022/2023

Indice

Introduzione

Il lavoro seguente ha l'obiettivo di esaminare e confrontare alcune tecniche e modelli statistici nell'ambito della classificazione di immagini, problema con il quale si indica il compito di assegnare un'etichetta ad un'intera immagine. Particolare rilevanza viene data all'approccio *deep* che caratterizza le reti neurali profonde e ai vantaggi che derivano dall'utilizzo di questo tipo di modello rispetto al problema in esame. Ciascun modello viene definito partendo da una base teorica per poi applicarlo ad un dataset reale ed infine confrontare i risultati ottenuti con gli altri modelli.

In particolare, nel Capitolo 1 si fornisce una breve introduzione al deep learning riguardo la sua importanza e agli ambiti di utilizzo. Nel Capitolo 2, invece, si descrive il modello di deep learning per eccellenza: la rete neurale. Di quest'ultimo vengono descritte, in modo generale, l'architettura e le tecniche di stima dei parametri e regolarizzazione del modello. Nel Capitolo 3 si descrivono, più nel particolare, i modelli in esame: viene contrapposto l'approccio machine learning tradizionale, caratterizzato da modelli quali *Support Vector Machines* e *Random Forest*, all'approccio deep tipico di modelli di rete neurale quali *Convolutional Neural Network*, *Recurrent Neural Network* e *Deep Clustering*. Infine, nel Capitolo 4, questi modelli vengono applicati ad un dataset di immagini di capi di abbigliamento e confrontati rispetto alla maggior precisione ottenuta nella classificazione di ogni capo. Le analisi sono svolte utilizzando i linguaggi Python e R.

Capitolo 1

Machine learning vs Deep learning

Questo capitolo presenta a grandi linee i concetti di machine learning e deep learning, il loro rapporto e i vantaggi e svantaggi dell'uno e dell'altro. L'attenzione è focalizzata principalmente sul deep learning, in quanto argomento centrale di questo lavoro.

1.1 Che cos'è il deep learning?

Il deep learning è un metodo di intelligenza artificiale che insegna ai computer ad elaborare dati in un modo che si ispira vagamente al cervello umano. Il termine deriva dal voler addestrare gli elaboratori a risolvere anche problemi intuitivi e automatici, ad esempio riconoscere parole, facce o immagini e non solamente problemi deterministici di calcolo. Il metodo per far questo è rappresentare il mondo in termini di una gerarchia di concetti, in cui ogni concetto è definito tramite relazioni con altri concetti. In parole povere, agli occhi del computer, il mondo è rappresentato come un enorme e profondo grafo.

Il deep learning, in particolare, permette al computer di rappresentare concetti complicati a partire da un insieme di concetti più semplici (ad esempio posso rappresentare una persona come insieme di relazioni tra concetti più semplici quali occhi, naso, angoli, contorni, forme, ...).

In sostanza, il deep learning è un particolare tipo di machine learning cioè di una tecnica che permette all'elaboratore di apprendere nuove informazioni dai dati. Approfondiremo meglio questo concetto nella sezione 1.2.

In questa sezione vedremo i fattori determinanti che hanno portato allo sviluppo e alla diffusione del deep learning in moltissimi ambiti di studio e di applicazione e i motivi del suo successo.

1.1.1 Perché il deep learning è importante?

Per capire il motivo del successo del deep learning e dell'importanza che ricopre al giorno d'oggi dobbiamo prima spiegare brevemente la storia del suo sviluppo.

Il nome deep learning ci fa pensare a qualcosa di nuovo, di recente, ma di fatto non lo è. Infatti, la sua storia inizia negli anni 50 del Novecento e durante il suo percorso è stato etichettato in vari modi: cibernetica, connessionismo, e infine, solo all'inizio dei primi anni 2000, deep learning. Di fatto il termine è salito alla ribalta solo recentemente grazie alla digitalizzazione

della società e al conseguente avvento dell'era dei "Big Data". L'enorme mole di dati che vengono collezionati al giorno d'oggi ci rende disponibili dataset sempre più grandi e di conseguenza ci permette di definire stime e previsioni statisticamente migliori anche utilizzando modelli meno complessi. Per questo, l'utilizzo e la popolarità di algoritmi di machine e deep learning è aumentata esponenzialmente negli ultimi anni tanto da indurci a pensare che sia una disciplina sviluppatasi solo negli ultimi 10-20 anni. Tuttavia, come detto in precedenza, gli albori del deep learning risalgono agli anni 50 del Novecento.

Un altro aspetto chiave nella diffusione e popolarità dei modelli di deep learning è che al giorno d'oggi disponiamo di strumenti computazionali eccezionali in grado di stimare ed allenare modelli sempre più complessi (e quindi più computazionalmente costosi) e con una mole di dati sempre più pesante, strumenti che fino a pochi anni fa non erano scontati. Da questo punto di vista, nel prossimo futuro ci aspettiamo ulteriori sviluppi sia in termini di complessità del modello che per quanto riguarda la mole di dati prodotta dalla popolazione, il tutto accompagnato da innovazioni in campo hardware e software.

I modelli di deep learning sfruttano algoritmi di apprendimento basati su reti neurali artificiali (ANN o *Artificial Neural Network*), cioè modelli matematici ispirati dal funzionamento delle reti neurali biologiche del cervello umano, ossia modelli costituiti da connessioni tra i vari livelli che li compongono. Parleremo delle reti neurali nel capitolo 2. Il deep learning moderno, tuttavia, va oltre questa prospettiva neurobiologica e, in modo più generico, definisce modelli basati sull'apprendimento a più livelli (strati) di composizione dei dati. Le caratteristiche dei dati vengono rilevate tramite apprendimento non supervisionato e scomposte in caratteristiche di basso e alto livello tramite i vari strati del modello in modo da creare una rappresentazione gerarchica dell'input. L'output viene poi ottenuto tramite una funzione non lineare delle caratteristiche.

Gli algoritmi di deep learning odierni sono in grado di riconoscere pattern complessi in immagini, testo, suoni e altri dati per produrre informazioni e previsioni accurate. In generale, questi metodi funzionano molto bene con dati non strutturati in quanto riescono ad analizzare il dato dividendolo automaticamente in parti più semplici senza dover effettuare questa estrazione manualmente, cosa che invece avviene nei modelli di machine learning.

Perciò, cercando di rispondere alla domanda che dà il titolo a questa sottosezione, possiamo dire che conoscere ed applicare algoritmi di deep learning è utile in quanto costituiscono un'arma in più nel nostro arsenale per l'analisi di dati. Inoltre, è stato dimostrato che nel loro campo di applicazione (vedi sezione 1.1.2) gli algoritmi di deep learning, con le dovute migliorie, sono decisamente più efficienti rispetto agli algoritmi classici di machine learning. Valuteremo questo confronto di performance con un esempio pratico nel capitolo 4.

1.1.2 In quali ambiti si utilizza il deep learning?

Come detto nella sottosezione precedente, i modelli di deep learning sono in grado di riconoscere strutture complesse in immagini, suoni, testi e altri dati per produrre informazioni e previsioni accurate. Si possono perciò utilizzare i metodi di deep learning per automatizzare le attività che in genere richiedono l'intelligenza umana, come la descrizione e classificazione di immagini, la trascrizione di un file audio in testo, l'elaborazione del linguaggio naturale, la computer vision, l'analisi di testi, la bioinformatica e analisi e previsione di serie storiche.

Quindi, data la potenza degli algoritmi di deep learning, è lecito chiedersi se valga la pena dimenticasi di tutti gli altri metodi e modelli e limitarci all'utilizzo delle reti neurali. La risposta è no! Dobbiamo ricordarci del principio del *Rasoio di Occam* secondo il quale tra due modelli ugualmente efficienti è preferibile scegliere il più semplice. Stimare un modello di rete neurale è complicato ed è insensato da utilizzare quando possiamo raggiungere lo stesso risultato con un modello più semplice e più facilmente interpretabile. Nella scelta del modello da utilizzare bisogna quindi valutare attentamente il compromesso tra performance e complessità. Non sempre il modello migliore è quello più potente.

Inoltre, bisogna considerare che gli algoritmi di deep learning forniscono risultati migliori quando vengono allenati su grandi quantità di dati e di buona qualità. I valori anomali o gli errori nel set di addestramento possono influire in modo significativo sulla stima del modello di deep learning. Per evitare tali imprecisioni, prima di addestrare i modelli, è necessario pulire ed elaborare una grande mole di dati. La pre-elaborazione dei dati di input richiede grandi capacità di archiviazione di dati. Perciò, tipicamente consideriamo il deep learning una scelta attraente per le nostre analisi quando la numerosità campionaria è molto elevata e quando l'interpretabilità del modello non è una priorità.

1.2 Rapporto tra machine learning e deep learning

Come accennato nella sezione 1.1, il deep learning è un particolare tipo di machine learning, termine con cui si indica un insieme di metodi, tecniche e algoritmi in grado di imparare dai dati. Il deep learning segue lo stesso obiettivo del machine learning ma lo fa in modo diverso. Mentre il machine learning dipende strettamente dal tipo di informazione e da come questa viene fornita dall'utente, cioè dalle caratteristiche (*features*) del caso in esame, il deep learning acquisisce e apprende queste caratteristiche in modo automatico, da un'immagine, da un documento o da un testo, rappresentando l'input fornito in un insieme gerarchico di concetti, ognuno definito tramite relazioni con altri concetti più semplici.

In questa sezione vedremo una piccola introduzione al machine learning per poi approfondire le differenze tra i due metodi e le migliorie apportate dal deep learning in alcuni ambiti di applicazione.

1.2.1 Le basi del machine learning

Il machine learning è un ramo dell'intelligenza artificiale che comprende un insieme di metodi, tecniche e algoritmi che riescono ad identificare in modo automatico strutture e relazioni nei dati, cioè ci permettono di imparare dai dati, e sfruttare questa conoscenza acquisita per predire dati futuri o prendere decisioni in condizioni di incertezza. Per capire cosa si intende con "imparare dai dati" cito una definizione popolare nell'ambito dell'informatica: "*A computer program is said to learn from experience E with respect to some class of task T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E*", cioè "imparare dai dati" nel senso di migliorare la performance nello svolgere certi obiettivi al crescere dell'esperienza, intesa come quantità di dati analizzati. Ora analizzeremo la definizione per andare un po' più nel dettaglio.

Con T (*tasks*) si indica tutte quelle operazioni, compiti e mansioni troppo complicate per un essere umano ma che necessitano di essere portate a termine in modo intelligente e con il miglior risultato possibile. Tra le *tasks* di un algoritmo di machine learning, perciò non vi è l'imparare dai dati, il processo di apprendimento, infatti, è il metodo per ottenere l'abilità di risolvere le *tasks*, e non una di esse. Tra le più comuni *tasks* del machine learning troviamo invece:

- Classificazione: l'algoritmo ha il compito di individuare a quale delle M categorie appartiene l'input che viene fornito.
- Regressione: all'algoritmo viene chiesto di prevedere un valore numerico dato un certo tipo di input. Questo tipo di lavoro è simile alla classificazione eccetto per il tipo di output che viene prodotto. Nella classificazione l'output è una delle M categorie mentre nella regressione è un valore numerico.
- Trascrizione: il compito dell'algoritmo è trasformare un certo tipo di input non strutturato in output di testo.
- Traduzione: in questo caso, l'input è una sequenza di simboli in una certa lingua e all'algoritmo viene chiesto di convertirlo in una sequenza di simboli in un'altra lingua.

La lettera P (*performance*) è una misura quantitativa per valutare le abilità e i risultati di un algoritmo di machine learning. Indica l'accuratezza del modello, cioè la proporzione di unità per cui il modello produce un output corretto. Nella pratica la performance dell'algoritmo viene misurata usando un set di dati (*test set*) diverso da quello utilizzato per adattare il modello (*train set*), questo perché la sfida principale del machine learning è di riuscire a generare buone previsioni su un insieme di dati che non sono ancora stati rilevati (sui dati di domani) usando il modello allenato con i dati a disposizione (i dati di oggi). Dato che i dati di domani non sono disponibili, i dati di oggi vengono suddivisi in due: train set e test set. Il train set viene utilizzato per stimare il modello e il test set per valutarne la performance. Nella fase di stima del modello è necessario non essere troppo ottimisti, cioè non pensare che i dati di domani abbiano lo stesso comportamento dei dati di oggi, infatti entra in gioco la componente casuale che non è prevedibile. Se non si tenesse in considerazione questo aspetto allora saremmo tentati di costruire un modello che sia il più potente possibile, in modo da prevedere con errore nullo i dati di oggi. Tale modello identificherebbe anche l'errore casuale come parte strutturale dei dati (quando invece non lo è) portando a dei grossi errori in caso di previsione futura. Tale fenomeno è detto *overfitting* o sovradattamento del modello rispetto ai dati. Per evitare ciò, è necessario ricercare il giusto compromesso tra varianza e distorsione delle stime (vedi figura 1.1). Queste due quantità sono infatti inversamente proporzionali: all'aumentare dell'una l'altra diminuisce. L'EQM (Errore Quadratico Medio) o MSE (*Mean Squared Error*) di previsione è dato da:

$$\text{EQM} = E[(\hat{\theta} - \theta)^2] = Bias(\hat{\theta})^2 + Var(\hat{\theta}) \tag{1.1}$$

ed è una misura dell'errore complessivo che lo stimatore $\hat{\theta}$ commette nel prevedere θ, cioè il vero valore oggetto della stima. L'EQM è utilizzabile solo nel caso di regressione. Nel caso di classificazione con M esiti possibili, la bontà della previsione effettuata è calcolata attraverso

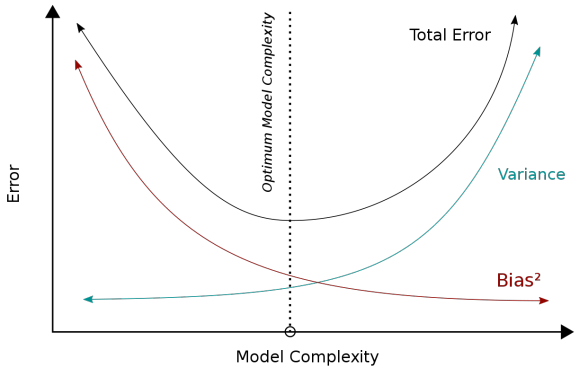


Figura 1.1: Trade-off tra varianza e distorsione in relazione alla complessità del modello

l'entropia incrociata categoriale (*categorical cross entropy*):

$$\text{CCE} = -\sum_{i=1}^{n}\sum_{m=1}^{M} \theta_{im} \log\left(\hat{\theta}_{im}\right) \tag{1.2}$$

Quindi, se lo scopo del modello è solo previsivo allora è sensato misurare la performance con l'EQM o con la CCE. Tuttavia, la bontà previsiva non è l'unico aspetto di cui dobbiamo tenere conto. La valutazione di un modello di machine learning non è un'operazione banale, infatti, è difficile ottenere una misura di performance che tenga conto di tutti gli aspetti importanti che un buon modello deve soddisfare. Per questo vengono definite diverse misure di bontà e, in base allo scopo dell'algoritmo, viene attribuito un peso diverso a ciascuna di esse.

Per quanto riguarda la lettera E (*experience*), gli algoritmi di machine learning vengono classificati in supervisionati e non supervisionati in base a quanta esperienza dispongono durante il processo di apprendimento. Nell'approccio supervisionato l'obiettivo è di apprendere la relazione tra un insieme di caratteristiche X e un'etichetta Y che le riassume. Nell'approccio non supervisionato disponiamo invece della sola lista delle caratteristiche X e l'obiettivo è quello di scovare strutture interessanti nei dati. Non è un tipo di problema ben definito, dato che non abbiamo una struttura precisa da identificare (un'etichetta), ma solamente un qualcosa che noi reputiamo interessante (*knowledge discovery*). In termini più matematici, l'apprendimento non supervisionato consiste in disporre di una serie di osservazioni di un vettore casuale X e provare ad imparare la funzione di probabilità $p(X)$ che segue X. Al contrario, nell'apprendimento supervisionato, oltre alla serie di osservazioni da X si osserva anche una serie di valori di un vettore casuale Y associato con X. Lo scopo in questo caso è di apprendere una funzione che possa mappare con precisione gli input X agli output Y e che consenta di effettuare predizioni accurate di Y dato un certo input X.

1.2.2 In cosa il deep learning è migliore del machine learning?

Come abbiamo già detto più volte, il deep learning è un sottoinsieme del machine learning. Gli algoritmi di deep learning sono emersi nel tentativo di rendere più efficienti le tecniche di machine learning tradizionali.

I modelli di deep learning, come le reti neurali profonde, sono preferiti rispetto al machine learning tradizionale in caso di:

1. Analisi da dati complessi: i modelli di deep learning sono in grado di apprendere rappresentazioni di alto livello dai dati, specialmente le reti neurali profonde.

2. Riconoscimento di pattern: le tecniche di deep learning primeggiano nel riconoscere ed estrarre pattern complessi e caratteristiche latenti dei dati.

3. Scalabilità: le reti neurali profonde sono in grado di gestire in modo efficiente grandi quantità di dati. Nel machine learning tradizionale, spesso è necessario identificare manualmente le caratteristiche rilevanti di un insieme di dati. Un modello di rete neurale può invece ricevere in input un numero enorme di variabili, la selezione avviene tramite la natura stessa del modello, senza richiedere interventi manuali specifici.

4. Generazione di contenuti: il deep learning viene ampiamente utilizzato per la generazione di nuovi dati e contenuti attraverso l'uso dei *Generative Adversarial Networks*. Questo speciale tipo di rete neurale è in grado di generare dati e contenuti più realistici rispetto alle tecniche di machine learning tradizionale.

D'altra parte, il deep learning non è sempre la soluzione migliore. I modelli di deep learning, infatti, forniscono risultati ottimi solo quando vengono addestrati su grandi quantità di dati e di alta qualità. Nei casi in cui i dati a disposizione siano scarsi, le risorse computazionali a disposizione siano limitate, il problema da risolvere sia semplice e l'interpretazione del modello sia importante, allora è preferibile utilizzare le più semplici tecniche tradizionali di machine learning.

Capitolo 2

Le reti neurali

I modelli di deep learning sfruttano algoritmi di apprendimento basati su reti neurali artificiali (*ANN* o *Artificial Neural Network*). Il nome stesso ci fa pensare a modelli ed algoritmi composti da "neuroni" artificiali ed ispirati vagamente ad una rete neurale biologica. Fin dagli albori, infatti, una delle idee alla base del deep learning è replicare computazionalmente il funzionamento del cervello umano in modo da addestrare un elaboratore ad agire in modo intelligente come farebbe un essere umano.

Una rete neurale artificiale è quindi un algoritmo deep learning di apprendimento supervisionato o non supervisionato che estrae combinazioni lineari degli input e le usa come argomento di una funzione non lineare per ottenere l'output desiderato. É un modello statistico non lineare applicabile allo svolgimento di molte mansioni, principalmente per regressione, classificazione ed elaborazione di dati. Nascono verso la metà degli anni '80 ma hanno raggiunto una certa fama solo recentemente grazie al miglioramento degli strumenti computazionali e alla sempre maggiore mole di dati immagazzinati al giorno d'oggi.

Questo capitolo presenta alcuni dettagli riguardo alla formalizzazione e al funzionamento di una rete neurale.

2.1 Architettura di una rete neurale

In questa sezione introduciamo un modello di rete neurale formato da uno strato di input, due strati nascosti (*hidden layers*) e uno strato di output. Il funzionamento della rete è basato sulla propagazione delle informazioni in avanti, seguendo cioè un flusso unidirezionale attraverso i vari strati. Tale modello di rete neurale si dice *feed-forward*.

2.1.1 Il modello di rete neurale feed-forward

Una rete neurale è definita come un modello di deep learning che, dato un input in p variabili $X = (x_1, x_2, \ldots, x_p)$ costruisce una funzione non lineare $f(X)$ per prevedere la risposta Y. La previsione viene fatta in più step: date le caratteristiche iniziali $X = (x_1, x_2, \ldots, x_p)$ (*input layer*), queste vengono combinate tramite un vettore di coefficienti $\omega^{(2)}$ e trasformate mediante

una funzione non lineare $g^{(1)}$ in K_1 *activations*: $A_k^{(1)}$ per $k = 1, \ldots, K_1$:

$$A_k^{(1)} = g^{(1)}(z) = g^{(1)}(\omega_{k0}^{(1)} + \sum_{j=1}^{p} \omega_{kj}^{(1)} x_j) \tag{2.1}$$

Queste ultime, verranno a loro volta utilizzate come input per stimare le *activations* dello strato successivo:

$$A_k^{(2)} = g^{(2)}(z) = g^{(2)}(\omega_{k0}^{(2)} + \sum_{j=1}^{K_1} \omega_{kj}^{(2)} A_k^{(1)}) \tag{2.2}$$

e così via. Il numero di neuroni K può variare da strato a strato e formano gli *hidden layers* della rete neurale. Con z si indica il generico input al neurone k che verrà poi trasformato mediante la funzione $g()$ per ottenere A_k. Nell'ultimo strato, lo strato di output, viene stimata la risposta Y tramite un modello di regressione lineare

$$Y = f(X) = \beta_0 + \sum_{k=1}^{K_S} \beta_k A_k^{(S)} \tag{2.3}$$

in cui le variabili esplicative sono le *activations* stimate all'ultimo strato nascosto S, K_S è il numero di neuroni dell'ultimo strato e β_k sono i coefficienti di regressione. Nel caso la risposta sia multivariata sarà necessario effettuare M regressioni, una per ogni esito possibile. La figura 2.1 mostra un esempio di rete neurale che rispecchia quanto descritto in questa sezione. Come detto precedentemente, questo tipo di rete neurale si dice *feed-forward* in quanto la propagazione delle informazioni tra gli strati avviene esclusivamente in direzione avanzata. Esistono anche altre varianti di rete neurale in cui la propagazione delle informazioni può avvenire anche all'indietro o attraverso dei loop. In particolare, il modello di rete neurale ricorrente (descritto nella sezione 3.3.2) è un tipo di rete neurale in cui, tramite dei loop, le informazioni si propagano non solo in avanti, ma anche all'indietro nel tempo.

2.1.2 Funzione di attivazione

La funzione $g()$ nelle equazioni 2.1 e 2.2 si dice funzione di attivazione (*activation function*) ed il suo compito è di introdurre la non linearità nel modello. La più comunemente utilizzata è la funzione *ReLU (Rectified Linear Unit)*:

$$g(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases} \tag{2.4}$$

con z che rappresenta l'input al neurone per cui si sta calcolando la rispettiva A_K, cioè z è una regressione sui parametri dello strato precedente (vedi le equazioni 2.1 e 2.2). Possono essere utilizzate anche altre funzioni di attivazione, l'unico vincolo è la loro non linearità. Ad esempio la funzione *sigmoid*:

$$g(z) = \frac{1}{1 + e^{-z}} \tag{2.5}$$

Tuttavia, quest'ultima si dimostra meno efficiente dal punto di vista computazionale rispetto alla *ReLU*. Infatti, l'utilizzo della *ReLU* rende più veloce il processo di discesa del gradiente nella

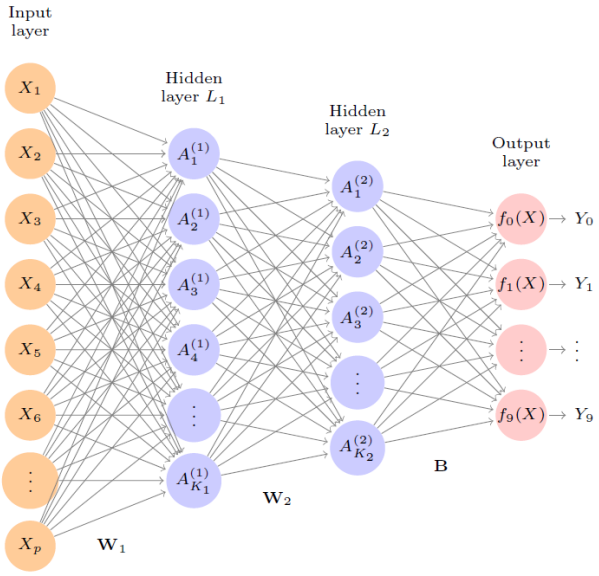


Figura 2.1: Diagramma di una rete neurale *feed-forward* con due *hidden layers* e una variabile risposta categoriale con $M = 10$ categorie possibili.

stima della rete neurale, in quanto il gradiente di g rimane costante (1 se $z > 0$, 0 se $z < 0$) e l'unica informazione che ci serve è il segno di z. La non differenziabilità in $z = 0$ è risolta attribuendo un valore arbitrario (0 o 1) per la derivata in quel punto. La discesa del gradiente è un metodo di stima della rete neurale tramite aggiornamento continuo dei parametri in base al gradiente della funzione di perdita e ad un parametro moltiplicativo che determina il passo di aggiornamento. Approfondiremo questi aspetti nella sezione 2.2.

Tornando alla funzione di attivazione $ReLU$, bisogna far presente che quest'ultima soffre nel caso in cui molti neuroni abbiano input negativo ($z < 0$). In questo caso, il gradiente di g è pari a zero e ciò porta ad uno stato stagnante in cui i parametri del neurone non vengono aggiornati. Se i neuroni con input negativo sono molti, il problema si propaga fino ad arrivare alla stagnazione del modello. I metodi di discesa stocastica del gradiente alleviano questo problema. Infatti, considerando una frazione abbastanza ampia di osservazioni (*batch*) possiamo ben sperare che almeno una di queste abbia gradiente positivo in modo che il gradiente totale della funzione di perdita, cioè la somma del gradiente calcolato per ogni osservazione, sia maggiore di zero. In alternativa possiamo considerare varianti della $ReLU$ come ad esempio la *Leaky ReLU* o la ELU. Lo scopo di queste varianti è di ovviare al problema di stagnazione della rete introducendo una specificazione diversa da zero anche nel caso in cui l'input al neurone sia negativo:

$$\textit{Leaky ReLU: } g(z) = \begin{cases} z & z \geq 0 \\ az & z < 0 \end{cases} \tag{2.6}$$

$$ELU: g(z) = \begin{cases} z & z \geq 0 \\ a(e^x - 1) & z < 0 \end{cases} \tag{2.7}$$

Entrambe definiscono un valore a come costante moltiplicativa positiva. Nella *Leaky ReLU* a viene scelto molto piccolo, solitamente 0.01 mentre nella ELU solitamente $a = 1$. É possibile, ed

è consigliabile, effettuare alcuni tentativi con valori diversi di a (ma sempre positivi) e scegliere il valore che fornisce prestazioni migliori. Un ulteriore miglioramento è determinare un parametro a per ogni strato durante l'addestramento del modello. In particolare, per ogni strato s viene determinato un valore a_s durante il processo di stima del modello. In questo modo la funzione di attivazione riesce anche ad adattarsi alle caratteristiche dello strato per cui è calcolata, apportando un leggero miglioramento delle performance. Quest'altra variante della $ReLU$ si chiama $PReLU$ ed è definita come:

$$PReLU\text{: } g(z) = \begin{cases} z & z \geq 0 \\ a_s z & z < 0 \end{cases} \tag{2.8}$$

Queste alternative però perdono un po' in efficienza computazionale (velocità del processo) in quanto il calcolo del gradiente non avviene in modo immediato come nella $ReLU$.

In un modello particolare di rete neurale, la rete neurale ricorrente, si preferisce utilizzare per gli strati ricorrenti una funzione di attivazione tangente iperbolica ($tanh$):

$$g(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2 \cdot sigmoid(2z) - 1 \tag{2.9}$$

Quest'ultima non è altro che una versione scalata e traslata della funzione $sigmoid$ all'intervallo $[-1; 1]$, infatti ne eredita le proprietà. La funzione di attivazione $tanh$ è spesso utilizzata per un modello di RNN in quanto, data la sua natura, riesce a catturare relazioni anche nei valori negativi, caratteristici di dati in serie temporale.

Per quanto riguarda lo strato di output, la scelta della funzione di attivazione da usare dipende dal problema in esame. Nel caso la risposta sia quantitativa allora lo strato di output sarà semplicemente formato da un numero di neuroni pari alla dimensionalità della risposta ma senza funzione di attivazione in quanto non è più necessario trasformare l'output. Lo stesso si fa nel caso in cui la risposta sia qualitativa binaria aggiungendo però una funzione di attivazione $sigmoid$ in modo da ottenere una misura di probabilità per determinare la classificazione. Infine, se la risposta è qualitativa divisa in $M > 2$ categorie, la funzione di attivazione più appropriata è la funzione $softmax$:

$$\sigma(Z)_m = \frac{e^{z_m}}{\sum_{l=1}^{M} e^{z_l}} \tag{2.10}$$

per $m = 1, \ldots, M$ dove $Z = (z_1, \ldots, z_M)$ è il vettore formato dalle M regressioni effettuate sullo strato precedente. Questa funzione mappa il vettore numerico di input in una distribuzione di probabilità formata da M misure di probabilità proporzionali all'input fornito. La classificazione stimata è il valore di m in corrispondenza del quale la 2.10 è massima.

2.1.3 Funzione di perdita

La funzione di perdita quantifica la differenza tra la previsione effettuata dal modello e il vero valore dell'output. Definendo con y il vero output e con $\hat{y}$ l'output predetto dal modello, una funzione di perdita $R(y, \hat{y})$ quantifica l'errore commesso dal modello. La forma della funzione R cambia in base al problema in esame: regressione o classificazione. Nel caso di un modello di rete neurale, il numero di coefficienti è molto elevato, perciò, indichiamo con θ il vettore di tutti i coefficienti della rete neurale.

Come funzione di perdita, in un problema di regressione su n osservazioni, possiamo utilizzare la funzione *errore quadratico medio* (MSE):

$$R(y, \theta) = \frac{1}{2}\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{2}\frac{1}{n}\sum_{i=1}^{n}(y_i - f_\theta(x_i))^2 \tag{2.11}$$

con $f(x_i)$ che è l'equazione del modello, cioè il predittore lineare formato dai coefficienti delle funzioni di attivazione contenuti nel vettore θ. Il termine $\frac{1}{2}$ serve solo per comodità nel calcolo della derivata di R (vedi sezione 2.2). Nel calcolo della 2.11, è da tenere in considerazione che l'MSE, data la presenza del termine quadratico, è molto sensibile alla presenza di valori anomali e questo potrebbe incidere nel processo di addestramento.

D'altra parte, in un problema di classificazione con M esiti possibili, possiamo utilizzare l'entropia incrociata categoriale (*categorical cross-entropy*) come funzione di perdita per misurare la diversità tra la distribuzione di y e la distribuzione di $\hat{y}$:

$$R(y, \theta) = -\sum_{i=1}^{n}\sum_{m=1}^{M} y_{im}\log\left(\hat{y}_{im}\right) = -\sum_{i=1}^{n}\sum_{m=1}^{M} y_{im}\log\left(f_{\theta m}(x_i)\right) \tag{2.12}$$

Infatti, in un problema di classificazione, y_{im} e $\hat{y}_{im}$ sono dei vettori M-dimensionali che rappresentano la distribuzione di probabilità dell'osservazione i di essere classificata come categoria m, per $m = 1, \ldots, M$. L'osservazione vera è un vettore di 0, con un 1 in corrispondenza della classificazione esatta, mentre la classificazione prevista attribuisce una probabilità per ogni classe. L'obiettivo è che queste due distribuzioni siano il più simili possibile.

2.2 Stima di una rete neurale

Stimare una rete neurale significa assegnare dei valori ai coefficienti in modo che il modello si adatti bene ai dati dell'insieme di addestramento e riesca a prevedere, con una buona precisione, i dati futuri.

Indicando con θ il vettore di tutti i coefficienti della rete neurale, la sua stima si ottiene minimizzando la funzione di perdita del modello. Ovvero, cercando di ridurre al minimo la distanza tra il vero valore osservato e la previsione del modello. La scelta della funzione di perdita varia in base al tipo di problema che si sta considerando (vedi sezione 2.1.3).

Come abbiamo visto nella sezione 2.1, un modello di rete neurale può avere un numero di coefficienti molto elevato. In particolare, le matrici dei pesi, o coefficienti del modello, sono indicate con $W_1, \ldots, W_s$ per gli strati nascosti, e con B per lo strato di output. Il numero di parametri per ogni strato è $(K_{s-1} + 1) \cdot K_s$ con s indice dello strato e il $+1$ che indica la presenza di intercetta. Le reti neurali profonde (*deep neural networks*) sono formate da molti strati nascosti e da molti neuroni per strato. Il numero di coefficienti totale del modello risulta quindi essere considerevole. Perciò, per risolvere in maniera efficiente il problema di minimizzazione della funzione di perdita e ottenere, allo stesso tempo, un modello con una buona accuratezza di previsione, dobbiamo utilizzare congiuntamente i metodi di discesa lungo il gradiente e regolarizzazione.

2.2.1 Discesa del gradiente

Il metodo di discesa lungo il gradiente è un algoritmo di ottimizzazione che consente di aggiornare iterativamente i coefficienti della rete neurale in modo da minimizzare la funzione di perdita. La procedura consiste nell'assegnare un valore iniziale θ_0 (scelto casualmente o attraverso stime preliminari) al parametro θ, calcolare il gradiente di R e valutarlo in θ_0. Se quest'ultimo è diverso da zero, allora aggiornare il valore di θ_0 lungo la direzione opposta a quella del gradiente in modo da spingere θ_0 verso un punto di minimo (il gradiente ci dice la direzione verso cui la funzione $R(y, \theta)$ aumenta più rapidamente). L'aggiornamento di θ_0 avviene perciò in questo modo:

$$\theta_{j+1} \leftarrow \theta_j - \delta \cdot \nabla R(y, \theta_j) \tag{2.13}$$

dove δ è una costante moltiplicativa che determina il passo di aggiornamento dei coefficienti. Una volta aggiornati i parametri, si stima nuovamente il modello con questi ultimi e si procede di nuovo all'aggiornamento. La procedura termina se $\nabla R(y, \theta_j) = 0$. Quando ciò accade, l'aggiornamento di θ_j non avviene più e l'algoritmo ci consegna il valore del parametro θ che individua un punto di minimo per $R(y, \theta)$.

Nella 2.13 il parametro δ prende il nome di *learning rate* ed è da scegliere con cura in quanto è determinante per l'efficacia della minimizzazione. Un valore di δ troppo elevato potrebbe generare oscillazioni troppo ampie del valore dei parametri e non riuscire ad identificare il punto di minimo. D'altro canto, un valore di δ troppo basso rende la procedura troppo lenta e computazionalmente costosa. Solitamente si prova il modello con alcuni valori di δ come $0.1, 0.05, 0.01, 0.001$.

Per θ invece, come valore iniziale θ_0 la scelta migliore è di inizializzare tutti i parametri con valori casuali vicini a zero. In questo modo inizializziamo la rete neurale a un modello quasi lineare e, man mano che procede l'aggiornamento dei parametri, il modello si allontana sempre più dalla linearità. In questo modo possiamo anche valutare se per modellare il fenomeno in esame è necessario introdurre la non linearità o se è sufficiente un più semplice modello lineare.

Il calcolo del gradiente nella 2.13 sfrutta la regola della catena per la derivazione di funzioni composte. Inoltre, dato che la funzione $R(y, \theta)$ è una somma, allora il suo gradiente è anch'esso una somma sulle n osservazioni:

$$\nabla R(y, \theta) = \sum_{i=1}^{n} \nabla R_i(y_i, \theta) \tag{2.14}$$

Perciò, per semplificare la notazione, dimostreremo il calcolo del gradiente solo per un'osservazione generica i e considereremo una rete neurale con un solo strato nascosto:

$$R_i(y_i, \theta) = \frac{1}{2}(y_i - \beta_0 - \sum_{k=1}^{K_1} \beta_k g^{(1)}(z_{ik}))^2 \tag{2.15}$$

con $z_{ik} = \omega_{k0} + \sum_{j=1}^{p} \omega_{kj} x_{ij}$ e $f_\theta(x_i) = \beta_0 + \sum_{k=1}^{K_1} \beta_k g^{(1)}(z_{ik})$. La derivata rispetto ad un generico

β_k é:

$$\begin{aligned}\frac{\partial R_i(y_i,\theta)}{\partial \beta_k} &= \frac{\partial R_i(y_i,\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial \beta_k} \\ &= -(y_i - f_\theta(x_i)) \cdot g(z_{ik}) \end{aligned} \tag{2.16}$$

e rispetto al generico ω_{kj}:

$$\begin{aligned}\frac{\partial R_i(y_i,\theta)}{\partial \omega_{kj}} &= \frac{\partial R_i(y_i,\theta)}{\partial f_\theta(x_i)} \cdot \frac{\partial f_\theta(x_i)}{\partial g(z_{ik})} \cdot \frac{\partial g(z_{ik})}{\partial z_{ik}} \cdot \frac{\partial z_{ik}}{\partial \omega_{kj}} \\ &= -(y_i - f_\theta(x_i)) \cdot \beta_k \cdot g'(z_{ik}) \cdot x_{ij} \end{aligned} \tag{2.17}$$

Notiamo che entrambe le espressioni contengono al loro interno un residuo $y_i - f_\theta(x_i)$. Perciò, durante l'aggiornamento 2.13, questo residuo viene propagato attraverso gli strati della rete per aggiustare la stima dei parametri nella direzione opposta al residuo, in modo da minimizzarlo. Il processo si ripete per ogni osservazione i, in questo modo si minimizza anche il residuo totale del modello. Quest'ultima proprietà prende il nome di retropropagazione dell'errore (*backpropagation*).

2.2.2 Scomparsa ed esplosione del gradiente

Nell'addestramento di una rete neurale profonda, formata quindi da molti strati, il processo di *backpropagation* può portare a dei problemi noti come scomparsa o esplosione del gradiente (*vanishing or exploding gradient problem*) che limitano le prestazioni del modello. In una rete neurale con un numero S di strati nascosti, la funzione $g(z_{ik})$ che viene derivata nella 2.17 è una composizione di S funzioni che, applicando la regola della catena per la sua derivazione, risulta una produttoria delle derivate di tutte le funzioni di attivazione utilizzate dallo strato s (quello del parametro per il quale stiamo derivando) in poi, fino all'ultimo strato nascosto S. Questa produttoria, insieme ad altri valori, viene poi moltiplicata per l'errore.

Nel problema di scomparsa del gradiente, per effetto della moltiplicazione delle derivate, l'errore propagato all'indietro durante il processo di aggiornamento dei parametri diminuisce esponenzialmente ad ogni strato, e ciò non permette ai coefficienti degli ultimi strati (quelli vicini a quello di input) di aggiornarsi. Questo tipo di problema è tipico nel caso si utilizzino funzioni di attivazione come la *sigmoid* o la *tanh*, in quanto sono caratterizzate da una derivata minore di 1 che quindi, nell'operazione di retropropagazione, spingono l'errore sempre più verso zero all'aumentare del numero di strati.

D'altra parte, il problema di esplosione del gradiente si verifica quando l'errore propagato all'indietro aumenta esponenzialmente, ciò porta ad un ampio aggiornamento dei parametri che rende il processo instabile. É il caso di funzioni di attivazione della famiglia $ReLU$. La derivata della funzione $ReLU$ è 1 per valori positivi e 0 per valori negativi, ciò può causare l'esplosione del gradiente nel caso in cui i coefficienti del modello siano grandi in valore assoluto e quindi la loro moltiplicazione per la derivata della $ReLU$ propagherebbe l'errore a infinito all'aumentare del numero di strati. Inoltre, sempre con la $ReLU$, nel caso di molti input negativi potremmo

incappare nel problema di stagnazione della rete (vedi sezione 2.1.2), che può essere visto come una specie di scomparsa del gradiente.

Per evitare i problemi di scomparsa ed esplosione del gradiente è quindi fondamentale inizializzare i coefficienti a valori vicini a zero e scegliere bene la funzione di attivazione da utilizzare.

2.2.3 Discesa stocastica del gradiente

Il metodo di stima dei parametri tramite discesa del gradiente descritto precedentemente è molto affidabile, tuttavia risulta essere lento in termini di tempo computazionale in quanto necessita di numerose iterazioni prima di arrivare a convergenza. Perciò, soprattutto con datasets di grandi dimensioni, è necessario velocizzare la procedura andando a rendere più efficiente il calcolo del gradiente. Infatti, come detto in precedenza, dato che la funzione $R(y, \theta)$ è una somma allora il suo gradiente è anch'esso una somma del gradiente calcolato per ogni osservazione. Perciò, per accelerare la procedura, ad ogni iterazione si sostituisce il valore esatto del gradiente della funzione di perdita con una sua stima ottenuta valutando il gradiente solo su un sottoinsieme casuale delle osservazioni (*minibatch*). Questo metodo è chiamato discesa stocastica del gradiente o *stochastic gradient descent* (SGD) e ad oggi è lo standard per la stima dei parametri di una rete neurale.

Tuttavia, come per il metodo di discesa del gradiente, anche lo SGD soffre dei problemi di scomparsa ed esplosione del gradiente, dipende fortemente dalla scelta iniziale dei parametri (inizializzazione) e spesso ci porta a identificare un punto di minimo che però è solo locale.

2.3 Regolarizzazione

Come detto in precedenza, le *deep neural networks* sono caratterizzate da molti strati e molti neuroni per strato in modo da ottenere performance migliori riguardo al problema in esame. Il rischio è quello di stimare un modello con un numero esagerato di parametri. É necessario perciò applicare un qualche metodo di regolarizzazione in modo da evitare il sovradattamento (*overfitting*) del modello ai dati. L'*overfitting* è una situazione in cui il modello è talmente complesso (cioè ha un numero di parametri esagerato rispetto ai dati in esame) che, oltre ad apprendere le caratteristiche strutturali del fenomeno, apprende anche le fluttuazioni aleatorie (il rumore) dei dati esaminati e le interpreta come caratteristiche strutturali quando in realtà non lo sono. Ciò produce un modello che si adatta perfettamente ai dati del train set, ma con scarsa accuratezza nel test set.

Ci sono diverse tecniche di regolarizzazione che possono essere applicate ad un modello di rete neurale, nel seguito ne verranno descritte quattro: contrazione dei coefficienti, *dropout regularization*, *batch normalization* e *dataset augmentation*.

2.3.1 Contrazione dei coefficienti

Per controllare la complessità del modello in modo da evitare il sovradattamento, possiamo aggiunge un termine di penalità alla funzione di perdita iniziale in modo da eliminare o comunque ridurre l'effetto di alcuni coefficienti. É infatti noto che coefficienti piccoli in valore assoluto

permettono al modello di generalizzare meglio i dati, riducendo il rischio di *overfitting*. Utilizzando come termine di penalità una funzione norma L_1 o norma L_2 sui coefficienti del modello e minimizzando la funzione di perdita penalizzata, i coefficienti vengono contratti verso lo zero proporzionalmente alla loro grandezza in valore assoluto. Vengono quindi contratti maggiormente i coefficienti distanti da zero, mentre per quelli vicini a zero la variazione è minima.

Indicando con $R(y, \theta)$ la funzione di perdita originale del modello, la funzione di perdita penalizzata tramite norma L_2 è:

$$R_1(y, \theta; \lambda) = R(y, \theta) + \lambda \sum_j \theta_j^2 \tag{2.18}$$

In alternativa è possibile usare la norma L_1: $\sum_j |\theta_j|$. Il parametro λ regola la forza della regolarizzazione. Un valore alto di λ genera un'elevata contrazione dei coefficienti verso zero. Viceversa, se $\lambda = 0$, la contrazione non c'è. La scelta di λ è effettuata provandone vari valori su un insieme di dati separato (*validation set*) e scegliendo il valore che minimizza l'errore di previsione sugli stessi dati. É anche possibile usare diversi valori di λ per gruppi di coefficienti di strati diversi. Per esempio possiamo decidere di penalizzare solo i coefficienti degli strati nascosti e non quelli dello strato di output.

2.3.2 Dropout

Un'altra tecnica di regolarizzazione è la *dropout regularization*. L'idea è di rimuovere casualmente una frazione ϕ (*dropout rate*) delle unità di uno strato durante l'operazione di stima del modello. Le unità rimaste dovrebbero quindi rappresentare anche quelle cancellate e i loro pesi vengono scalati di $1/(1-\phi)$ per compensare. Questa procedura è applicata separatamente ogni volta che viene passata un'osservazione del train set al modello. Il numero di neuroni della rete neurale è sempre lo stesso, solo che per ogni osservazione alcune unità vengono fissate a zero e quindi non considerate nel processo di apprendimento (vengono ignorate). Quindi, allenare una rete neurale utilizzando la dropout può essere pensato come stimare molte reti più "sottili" per poi fonderle in un'unica rete che raccoglie le proprietà chiave di ciascuna delle singole reti stimate. Questo processo diminuisce notevolmente il numero di coefficienti stimati e riduce quindi il rischio di *overfitting*.

2.3.3 Batch normalization

La *batch normalization* (BN), o normalizzazione del batch, è un metodo per combattere il problema di scomparsa o esplosione del gradiente e migliorare la stabilità del modello durante il processo di addestramento. Oltre alla scomparsa o esplosione del gradiente, una rete neurale può soffrire anche del problema di *internal covariate shift*. Tale problema si verifica soprattutto nelle reti neurali profonde e si riferisce al fatto che la distribuzione delle *activations* è varia all'interno di ogni strato nascosto e cambia ad ogni fase dell'addestramento. Tale problema è dovuto ad un qualche cambiamento nella distribuzione dei dati che causa un cambiamento dei parametri del modello durante l'addestramento. Il cambiamento continuo delle *activations* porta all'instabilità nel comportamento complessivo del modello. Questa instabilità implica che il processo di

addestramento della rete, per arrivare a risultati coerenti e accurati, richiederà più tempo del normale.

L'idea della BN è di aggiungere uno strato di normalizzazione tra due strati nascosti della rete neurale. Questi strati fungono da "normalizzatori", nel senso che vincolano le *activations* A_k di ogni strato ad avere media e varianza pari a media μ e varianza γ calcolate sul *minibatch* di osservazioni selezionato durante il processo di addestramento tramite SGD. Una volta normalizzate le unità, lo strato di BN trasferisce l'output al prossimo strato della rete, in modo che il processo di allenamento continui riducendo l'effetto della variazione delle distribuzioni e migliorando la stabilità del modello. Ciò rende la rete in grado di generalizzare meglio anche dati con distribuzioni diverse da quelle presenti nel train set, migliorando quindi le prestazioni del modello.

L'aggiunta di uno strato di BN introduce alcuni cambiamenti nell'algoritmo di discesa del gradiente e *backpropagation* in quanto è necessario tenere conto di due nuovi coefficienti: μ e γ. Il gradiente deve essere calcolato anche rispetto a questi due, permettendo il loro aggiornamento durante il processo. L'errore complessivo viene quindi propagato all'indietro anche attraverso lo strato di BN.

Come detto, la BN è efficacie anche contro il problema di scomparsa o esplosione del gradiente in quanto la normalizzazione delle unità di ogni strato impedisce a queste ultime di assumere valori troppo piccoli o troppo grandi che quindi, nelle moltiplicazioni successive, non avranno il rischio di far svanire o esplodere il gradiente. La normalizzazione rende il modello meno sensibile alla scelta iniziale dei coefficienti. Possiamo quindi interpretare la *batch normalization* come una tecnica di regolarizzazione del modello.

2.3.4 Dataset augmentation

Invece che concentrarsi sui coefficienti del modello, per cercare di evitare l'*overfitting* possiamo anche operare sul train set. Infatti, maggiore è la quantità di dati a disposizione, migliori sono le prestazioni del modello e minore è il rischio di sovradattamento.

Nel caso in cui non sia possibile collezionare nuovi dati, possiamo aumentare artificialmente la dimensione del train set distorcendo dati già esistenti. Nel caso di dati di immagini possiamo applicare trasformazioni quali zoom, rotazioni o traslazioni. Nel caso invece di dati di misura possiamo crearne di nuovi aggiungendo un errore casuale (ad esempio $\epsilon_i \sim \mathcal{N}(\mu = 0, \sigma^2 = 0.5)$) alle osservazioni già esistenti. L'etichetta dei dati generati sarà la stessa dei dati originali.

Inoltre, se la capacità di memoria a disposizione è limitata, possiamo anche distorcere direttamente sul momento il *minibatch* di dati selezionati durante il processo di discesa stocastica del gradiente. In questo modo i dati generati artificialmente vengono inseriti direttamente nel processo di stima e non è necessaria la loro memorizzazione.

Capitolo 3

Classificazione di immagini

La classificazione di immagini è una tecnica di intelligenza artificiale e machine learning che mira a identificare e assegnare un'etichetta o una categoria specifica a un'immagine digitale. La classificazione di immagini trova applicazioni in diverse aree, come l'analisi medica per diagnosticare patologie in modo automatico tramite immagini di risonanza magnetica o radiografie, l'analisi di immagini digitali da parte dei sistemi di sicurezza, il riconoscimento di oggetti, il riconoscimento facciale quando sblocchiamo il nostro cellulare e molto altro.

Questo capitolo inizia presentando il problema della classificazione di immagini e come queste vengono rappresentate tramite una matrice di pixel. Successivamente, presenta alcuni modelli di machine e deep learning adatti all'analisi e alla classificazione di immagini. Per ciascuno di questi, viene descritta l'architettura di base e il funzionamento generale.

3.1 Machine learning e deep learning nell'ambito della classificazione di immagini

Noi umani siamo in grado di assegnare un'etichetta ad un'immagine semplicemente guardandola. Tuttavia, data l'enorme quantità di dati da analizzare e il poco tempo a disposizione non possiamo etichettare manualmente milioni e milioni di immagini. Perciò dobbiamo addestrate un modello di machine o deep learning in modo che sia in grado, e con una buona precisione, di svolgere questo lavoro per noi.

Questi modelli ricevono in input un insieme di immagini e devono determinare, con la maggiore precisione possibile, la categoria di appartenenza di ognuna. Ogni immagine è rappresentata attraverso una struttura detta matrice di pixel o bitmap. Ogni pixel rappresenta una piccolissima parte dell'immagine, e contiene informazioni sul colore e sulla luminosità del punto corrispondente nell'immagine. La caratteristica del pixel varia in base al tipo di immagine: in scala di grigi o a colori. Nelle immagini in scala di grigio ogni pixel è rappresentato tramite un solo valore, che indica l'intensità luminosa di quella porzione di immagine. Questo valore varia da 0 a 255, dove 0 rappresenta il nero assoluto e 255 rappresenta il bianco assoluto. Nelle immagini a colori, invece, ogni pixel è rappresentato da una combinazione di tre valori di colore (o canali): rosso, verde e blu. Ogni canale rappresenta l'intensità del rispettivo colore tramite un numero da 0 a 255, e la

combinazione di questi tre valori crea il colore finale della porzione di immagine corrispondente al pixel. Quindi, in un'immagine a colori, ogni pixel è un vettore di 3 valori: (R,G,B).

Solitamente, ogni canale di colore ha una dimensione di 8 bit e perciò può assumere $2^8 = 256$ valori possibili. Tuttavia, in applicazioni in cui la qualità dell'immagine è fondamentale, la dimensione del canale potrebbe aumentare. Conoscere la dimensione del canale ci è utile in quanto, prima di passare in input un'immagine al modello, è consigliato standardizzare il valore dei pixel in modo velocizzare la procedura e rendere i dati più gestibili e facilmente confrontabili durante l'addestramento.

Le matrici di pixel ricavate dalle immagini possono quindi essere utilizzate come dati di input per i modelli di machine e deep learning nella classificazione di immagini. Abbiamo già spiegato nel capitolo 1 che, in generale, i modelli di deep learning riescono a fornire prestazioni migliori nel caso di analisi di dati complessi, come lo sono le immagini (vedi sezione 1.2.2). Le sezioni successive presentano alcuni modelli di machine e deep learning che possono essere impiegati con successo per un problema di classificazione di immagini.

3.2 Modelli di machine learning per la classificazione di immagini

Questa sezione presenta due tipici modelli di machine learning per la classificazione di immagini: *support vector machines* e *random forest*.

3.2.1 Support Vector Machines

Il modello di *support vector machines* (SVM) è una tecnica di classificazione e deriva da un'estensione del *support vector classifiers* (SVC) per includere la possibile presenza di non linearità della divisione in classi. L'idea è di aumentare lo spazio p-dimensionale delle variabili tramite qualche loro trasformazione, combinazione o forma polinomiale, in modo da catturare anche la non linearità. Il risultato di queste trasformazioni è un enorme valore di p, quello che fa il SVM è quindi introdurre la non linearità nel SVC e addestrarlo in modo computazionalmente efficiente usando i *kernels*.

Support vector classifier

Il SVC è un metodo basato sulla divisione (classificazione) delle osservazioni tramite un iperpiano $(p-1)$-dimensionale, dove p è il numero di variabili del modello. Questo metodo permette di ottenere solamente una classificazione binaria codificata in $\{-1, 1\}$ (l'osservazione sta sopra o sotto l'iperpiano). Parleremo alla fine di questa sezione di alcuni metodi per estendere questi concetti ad una classificazione multiclasse.

L'iperpiano viene determinato risolvendo un problema di ottimo. I termini $\beta_0, \ldots, \beta_p$ trovati definiscono un iperpiano che massimizza la distanza tra sé e l'osservazione a lui più vicina (massimizza il margine) e inoltre sbaglia appositamente la classificazione di una piccola parte delle osservazioni (determinata dal parametro C) per aumentare la precisione complessiva. Il margine è quindi collegato alla bontà di classificazione. Le osservazioni che hanno distanza dall'iperpiano pari al margine o che stanno nella parte sbagliata del margine rispetto alla loro vera classe di appartenenza, sono dette *support vectors*. Queste osservazioni sono determinanti nella definizione

del SVC (e quindi della banda di divisione tra le classi), in quanto una minima loro variazione comporta anche una variazione dell'iperpiano.

É dimostrato che la soluzione del problema di classificazione tramite il SVC (cioè la determinazione dell'iperpiano) richiede solo i prodotti scalari tra le osservazioni, perciò possiamo definire il *support vector classifier* come:

$$f(x) = \beta_0 + \sum_{i=1}^{n} a_i \langle x, x_i \rangle \tag{3.1}$$

dove i parametri da stimare sono gli a_i per $i = 1, \ldots, n$ e β_0, x è il nuovo punto da valutare, xi sono le osservazioni dell'insieme di train e $\langle x, x_i \rangle$ è il prodotto scalare tra due osservazioni. I parametri a_i sono definiti in modo che:

$$\beta = \sum_{i=1}^{n} a_i y_i x_i \tag{3.2}$$

dove $y_i = \{-1, 1\}$. Gli a_i sono anche collegati alle osservazioni che sono *support vectors*, cioè tutte le osservazioni che non sono *support vectors* hanno il corrispondente $a_i = 0$. Tale proprietà ci permette di ridurre notevolmente il numero di parametri e di prodotti scalare da calcolare e ci indica anche che il SVC è robusto rispetto ad osservazioni lontane dall'iperpiano (valori anomali).

Introduciamo ora nella 3.1 un *kernel* $K(x_i, x_{i'})$ al posto del prodotto scalare. Un *kernel* è una funzione che quantifica la similarità tra due osservazioni. Nel caso la relazione sia lineare allora il *kernel* adatto non è altro che il prodotto scalare tra le osservazioni. D'altra parte, nel caso in cui la relazione sia non lineare, possiamo considerare un *kernel* polinomiale di grado d:

$$K(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij} x_{i'j})^d \tag{3.3}$$

il che corrisponde, come detto in precedenza, ad addestrare il SVC includendo termini polinomiali di grado d estendendo quindi lo spazio p-dimensionale. Un'altra alternativa è il *kernel* radiale:

$$K(x_i, x_{i'}) = \exp(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2) \tag{3.4}$$

dove γ è una costante positiva da determinare. Questo tipo di *kernel* ha un comportamento molto locale, nel senso che la previsione per la nuova osservazione x sarà determinata quasi esclusivamente dalle osservazioni del train set a lei vicine in termini di distanza euclidea.

Il risultato della combinazione del SVC con un *kernel* non lineare è il modello di *support vector machines*:

$$f(x) = \beta_0 + \sum_{i \in S} a_i K(x, x_i) \tag{3.5}$$

dove S è l'insieme delle osservazioni che sono anche *support vectors*.

Come detto in precedenza, l'approccio basato sull'iperpiano di separazione descritto fino ad ora è però adatto solo al caso della classificazione binaria. In particolare, una volta determinata l'equazione 3.5, questa viene calcolata per l'osservazione x in esame e la classe di appartenenza

è determinata in base alla relazione $\hat{f}(x) < t$ o $\hat{f}(x) \geq t$, con t un certo valore soglia. Nel caso in cui la classificazione sia multiclasse, cioè con $K > 2$ classi, è necessario effettuare un altro passaggio. Una possibile soluzione è costruire $\binom{K}{2}$ modelli di SVM, ognuno dei quali confronta due classi. Vengono così confrontate tutte le possibili coppie di classi. La classificazione finale è determinata assegnando a x la classe che viene predetta più volte dai modelli. Questo approccio è chiamato *one-versus-one classification* e si contrappone al *one-versus-all classification* in cui si stimano K modelli di SVM confrontando una delle k classi (codificata come 1) con le restanti $K-1$ (codificate come -1). La classificazione finale di x è data dalla classe esaminata nel k-esimo modello che genera il più alto valore di $\beta_{0k} + \beta_{1k}x_1 + \cdots + \beta_{pk}x_p$.

3.2.2 Random Forest

Il *random forest* è un modello basato sugli alberi di decisione. Come suggerisce il nome stesso, è caratterizzato da un vasto insieme di alberi decisionali che vengono poi combinati tra di loro per generare una singola previsione. La caratteristica distintiva del *random forest* rispetto agli altri metodi basati sugli alberi è che questi ultimi sono incorrelati tra di loro, riducendo così la varianza di stima.

Più nel dettaglio, l'algoritmo consiste principalmente in due step: la fase di addestramento di ogni albero preso singolarmente e la fase di aggregazione delle previsioni.

Nella fase di addestramento vengono costruiti B alberi di decisione usando B diversi train sets ottenuti tramite bootstrap dal train set iniziale. Ogni train set è quindi ottenuto estraendo degli indici casuali dal singolo train set originale. Durante la crescita di ogni albero, vengono considerati come possibili candidati alla divisione (*split*) solo un sottoinsieme casuale m di tutti i p predittori (solitamente si pone $m \approx \sqrt{p}$). Lo *split* avviene su uno solo di questi m predittori, e ogni volta che è necessaria una nuova divisione si procede nuovamente all'estrazione casuale degli m predittori candidati per tale divisione. La motivazione di questa procedura discende dal fatto di voler ottenere alberi sostanzialmente diversi tra di loro (correlati negativamente), e quindi vincoliamo la procedura di divisione ad un sottoinsieme casuale di predittori in modo da evitare che le sequenza di *split* sia più o meno la stessa in ogni albero, ottenendo alberi diversi tra di loro. Questo processo è chiamato decorrelazione degli alberi.

Dopo aver ottenuto la previsione per ogni albero inizia la fase di aggregazione. Tutte le singole previsioni vengono combinate tra di loro tramite una media aritmetica generando così la previsione finale:

$$\hat{f}(x) = \frac{1}{B}\sum_{b=1}^{B} \hat{f}^b(x) \tag{3.6}$$

con $\hat{f}^b(x)$ che è la previsione ottenuta dall'albero b. Nell'ambito della classificazione di immagini, invece, la previsione aggregata è un vettore K-dimensionale in cui ogni elemento $p_k(x)$ è la proporzione di alberi che prevedono k come categoria di x. La previsione finale è la categoria k con la proporzione $p_k(x)$ più alta:

$$\hat{G}(x) = \arg\max_k [p_1(x), \ldots, p_K(x)] \tag{3.7}$$

L'incorrelazione tra le previsioni genera una diminuzione della varianza di stima e quindi rende la nostra previsione più affidabile. La scelta di B non è determinante per le sorti dell'algoritmo, ma è bene sceglierlo abbastanza grande.

3.3 Modelli di deep learning per la classificazione di immagini

In questa sezione vedremo tre modelli di deep learning basati su reti neurali profonde nell'ambito della classificazione di immagini: *convolutional neural networks*, *recurrent neural networks* e *deep clustering*.

3.3.1 Convolutional Neural Networks

Le reti neurali convoluzionali (CNN) sono un modello di rete neurale che utilizza in almeno uno strato l'operazione matematica di convoluzione. Le CNN sono particolarmente adatte per l'analisi di dati che presentano una struttura a griglia, come le immagini (griglia 2-D di pixel).

Una CNN lavora, in una certa misura, come farebbe un essere umano: riconosce specifiche caratteristiche (*features*) e pattern con i quali riesce a classificare accuratamente l'oggetto dell'immagine. In particolare, il processo di apprendimento avviene in due step: prima vengono identificate le *features* di più basso livello, come angoli, linee, bordi e colori, queste ultime vengono poi combinate tra loro per formare *features* di più alto livello, come occhi, orecchie, naso eccetera. Questo processo avviene tramite la combinazione di due particolari *hidden layers*: lo strato convoluzionale e lo strato di pooling (aggregazione). Le reti naurali convoluzionali profonde fanno uso di molti strati convoluzionali e di pooling in modo da catturare più dettagli possibili.

Uno strato convoluzionale è formato da un certo numero di filtri convoluzionali che hanno la funzione di determinare se una certa caratteristica è presente o meno in un'immagine. Questi ultimi vengono inizializzati casualmente, o con valori standard, e successivamente vengono aggiornati dalla CNN stessa durante il processo di addestramento del modello. Dal punto di vista matematico, un'operazione di convoluzione consiste nel moltiplicare la matrice bitmap (mappa dei pixel) dell'immagine per il filtro convoluzionale:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n \sum_d I(i+m, j+n, d) K(m,n,d) \tag{3.8}$$

dove S indica l'immagine convoluta, I è l'immagine originale e K è il filtro di dimensione $m \times n \times d$, cioè è una matrice $m \times n$ di coefficienti in cui ogni elemento è un vettore d-dimensionale. La dimensione del filtro va determinata in base alle caratteristiche delle immagini che si stanno analizzando. Solitamente si utilizzano filtri quadrati di dimensione 3,4 o 5. Un filtro di questa dimensione è in grado di analizzare bene l'immagine e ricavarne le caratteristiche più importanti. La terza dimensione del filtro, cioè l'indice d, indica la dimensione del singolo pixel, ovvero il numero di canali di colore presenti nell'immagine. Nelle immagini in scala di grigi $d = 1$, mentre nelle immagini a colori $d = 3$. La figura 3.1 mostra i dettagli dell'applicazione di un filtro convoluzionale 3x3x1 ad un'immagine (in scala di grigi). Il risultato, cioè l'immagine convoluta, ha dimensione pari al numero di sottomatrici 3x3 non uguali contenute nella bitmap

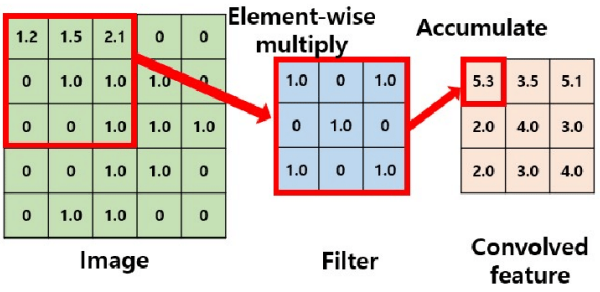


Figura 3.1: Applicazione di un filtro convoluzionale $3 \times 3 \times 1$ ad un'immagine.

dell'immagine originale e ne evidenzia le regioni che assomigliano al filtro applicato. Questa operazione viene ripetuta per ognuno dei k filtri dello strato convoluzionale in modo da catturare una grande varietà di dettagli. Il numero di coefficienti impiegato da uno strato convoluzionale per svolgere questa operazione è $k(m \cdot n \cdot d)$. Otteniamo quindi una serie di immagini convolute (una per ogni filtro), la *mappa delle caratteristiche* alla quale, prima di passare al *layer* successivo, verrà applicata la funzione di attivazione designata (solitamente una appartenente alla famiglia $ReLU$).

Lo strato di pooling, invece, permette di condensare un'immagine in una più piccola immagine riassuntiva. Prende come input l'output dello strato convoluzionale per ridurre la dimensione della mappa delle caratteristiche. Una delle possibili tecniche è utilizzare la funzione di *max pooling* che consiste nel riassumere ogni blocchetto della bitmap dell'immagine tramite il valore massimo al suo interno, restituendo quindi la caratteristica più rilevante presente in un certo blocchetto. Un'alternativa è utilizzare la funzione di *average pooling* che restituisce la media dal blocchetto considerato. Oltre alla riduzione della dimensionalità, l'operazione di pooling permette anche di conservare una certa invarianza di posizione dall'immagine iniziale a quella ridotta, nel senso che l'esatta locazione della caratteristica viene persa ma quello che è importante, cioè la locazione rispetto alle altre caratteristiche, viene conservata. Viene conservato anche il numero di canali d dell'immagine, dato che l'operazione di pooling riduce la dimensione dell'immagine solo in altezza e larghezza. La dimensione del blocchetto da riassumere e lo *stride* (passo) sono definiti a priori, solitamente si usano blocchetti 2x2 o 3x3. Lo *stride* indica di quanti passi il blocchetto si sposterà per determinare la prossima regione di pooling. Se voglio regioni non sovrapposte scelgo lo *stride* pari alla dimensione del blocco.

Oltre a diversi strati convoluzionali e di pooling, l'architettura di una CNN comprende anche uno strato di *flatten* (appiattimento) e uno o più strati completamente connessi che precedono lo strato di output. In particolare, uno strato convoluzionale è solitamente seguito da uno strato di pooling, tuttavia è anche possibile effettuare diversi strati convoluzionali in sequenza e poi uno strato di pooling in modo da catturare più caratteristiche e aumentare l'invarianza di posizione. Inoltre, una CNN con un numero elevato di strati convoluzionali ma con dimensione del filtro bassa, funziona meglio rispetto a una CNN con pochi strati convoluzionali e filtri grandi. Successivamente agli strati convoluzionali e di pooling, lo strato di *flatten* appiattisce le bitmap, ogni pixel viene quindi trattato come un'unità a sé stante per poi confluire in uno o più strati completamente connessi che integrano le informazioni provenienti dalle *feature map* estratte dagli strati convoluzionali e di pooling e le combinano per effettuare predizioni o classificazioni finali.

L'output dell'ultimo strato completamente connesso giunge infine allo strato di output che, in ambito di classificazione K-variata, è caratterizzato da una funzione di attivazione *softmax* con K livelli.

L'addestramento di una CNN avviene tramite l'algoritmo di discesa stocastica del gradiente e *backpropagation* descritti nella sezione 2.2. Tuttavia, una CNN arriva ad avere un numero di parametri molto elevato. É perciò utile applicare una qualche forma di regolarizzazione in modo da non incappare in *overfitting*. Possiamo utilizzare una delle tecniche descritte nella sezione 2.3. In particolare, nell'ambito di dati sotto forma di immagine, la *data augmentation* funziona molto bene in quanto per generare nuovi dati immagine è sufficiente applicare una piccola distorsione alle immagini già contenute nel dataset. Non è nemmeno necessario memorizzare le immagini distorte in quanto, durante la procedura di discesa stocastica del gradiente, vengono selezionate un *batch* casuale di immagini che quindi possiamo distorcere al volo senza doverle collezionare in memoria. Tipiche distorsioni sono zoom, rotazioni e piccoli spostamenti. L'etichetta delle immagini generate sarà la stessa dell'immagine originale.

3.3.2 Recurrent Neural Networks

Una rete neurale ricorrente (RNN) è un modello di rete neurale che utilizza al suo interno uno o più strati ricorrenti tramite la condivisione dei coefficienti tra istanti temporali differenti. Le RNN sono specializzate nell'analizzare dati di tipo sequenziale o con una dipendenza temporale. Tuttavia, è possibile impiegarle con successo anche per l'analisi di dati immagine trattando la sequenza dei pixel che compongono un'immagine come una serie storica. In particolare, ogni immagine viene considerata come un insieme di regioni successive, ogni regione è una sequenza di pixel, e quindi ogni immagine è una sequenza di regioni che rappresentano l'evoluzione spaziale dell'immagine (dei pixel nell'immagine), come fosse una serie storica. Lavorando quindi con dati sequenziali, introduciamo nella notazione un indice di tempo l. Ci sono due caratteristiche che identificano una rete neurale ricorrente:

1. la presenza di almeno uno strato ricorrente. Questo tipo di strato è caratterizzato dalla condivisione dei parametri tra istanti di tempo differenti. Cioè i neuroni di uno strato ricorrente al tempo l, oltre a ricevere un input dai neuroni dello strato precedente, ricevono un input anche dai neuroni dello stesso strato ma all'istante di tempo $l-1$. Perciò, l'output di un neurone dipende si dall'input che riceve dallo strato precedente ma anche dall'output dello stesso neurone ma al tempo precedente. É necessario, perciò, che la rete conservi un po' di memoria nel tempo. Questo tipo di strato permette al modello di catturare anche la sequenzialità temporale dei dati;

2. l'architettura degli strati è la stessa per ogni istante temporale, cioè i coefficienti del modello vengono condivisi e mantenuti costanti nel tempo durante l'analisi della sequenza temporale. Questo funziona in quanto ci aspettiamo che più osservazioni dello stesso dato si comportino in maniera simile e quindi è sensato utilizzare lo stesso insieme di coefficienti per predirne il comportamento. I coefficienti quindi, sono gli stessi per ogni istante temporale all'interno di una sequenza, ma variano da sequenza a sequenza.

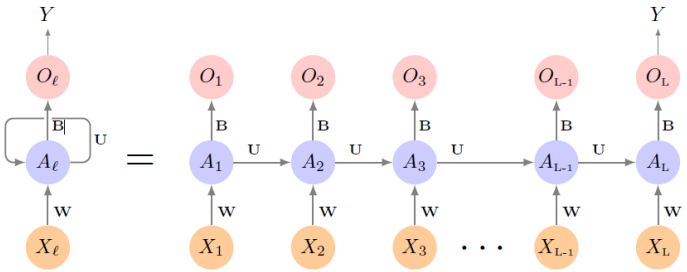


Figura 3.2: Architettura di una RNN con un solo strato ricorrente e con un solo neurone per strato. A sinistra in formato compatto, a destra in formato disteso. Nel caso di analisi di immagini, il generico input x_l rappresenta la regione l dell'immagine x.

In termini matematici, possiamo definire un neurone di uno strato ricorrente con:

$$A_l = g(b + Wx_l + UA_{l-1}) \tag{3.9}$$

con g funzione di attivazione non lineare, b l'intercetta, W è la matrice dei coefficienti tra lo strato di input e lo strato ricorrente e U è la matrice dei coefficienti che collegano gli strati ricorrenti a istanti temporali diversi. Per gli strati ricorrenti, la funzione di attivazione più utilizzata è la funzione tangente iperbolica: $g(z) = tanh(z)$ in quanto, dato il suo intervallo di variazione in $[-1; 1]$, permette di catturare relazioni anche nei valori negativi, caratteristici di dati in serie temporale.

Come fatto per gli strati ricorrenti, allo stesso modo possiamo definire lo strato di output del modello:

$$y_L = g(O_L) = g(d + BA_L) \tag{3.10}$$

con d intercetta, B matrice dei coefficienti tra lo strato nascosto e lo strato di output e g funzione di trasformazione di O_L nell'output finale y_L. Come detto in precedenza, l'architettura degli strati è sempre la stessa, quindi le matrici dei coefficienti W, U e B non cambiano nel tempo. Man mano che procede la sequenza, l'*activation* A_l apprende cos'è successo negli istanti precedenti, lo memorizza e usa poi questa conoscenza per la previsione finale. Nel caso di classificazione è necessario applicare la funzione *softmax* ad O_L in modo da ottenere un vettore di probabilità. Da notare che, per la previsione, l'output che ci interessa è solo quello all'ultimo istante temporale L, cioè dopo aver analizzato ogni elemento della sequenza. Gli output intermedi, invece, sono utili solo durante il processo di stima del modello, in quanto consentono di calcolare la funzione di perdita totale come somma delle funzioni di perdita ad ogni istante temporale. Inoltre, all'istante $l = 1$, per l'inizializzazione dell'operazione di ricorsione, definiamo A_0 come un vettore di zeri. In un modello di RNN, il numero di coefficienti di uno strato ricorrente è $(K_{s-1} + 1) \cdot K_s + K_s \cdot K_s$, in quanto è necessario tener conto anche dell'apporto dei coefficienti del tempo precedente allo strato s. Possiamo riassumere quanto detto attraverso la figura 3.2 che rappresenta un modello di RNN con un solo strato nascosto. Nelle applicazioni più avanzate, un modello di RNN è formato da numerosi strati nascosti, in modo da garantire una migliore performance. In questo caso, la ricorrenza temporale è presente solo allo stesso livello di strato e non tra neuroni di strati diversi.

L'addestramento di una RNN avviene attraverso l'algoritmo di *backpropagation through time* (BPTT) che non è altro che una versione dell'algoritmo di *backpropagation* modificato per tener conto della struttura ciclica della RNN (cioè della condivisione dei coefficienti). Questa procedura richiede l'espansione della RNN in formato disteso, come nella parte destra della figura 3.2. Il modello così ottenuto è essenzialmente una rete neurale classica, l'unica differenza è la ripetizione degli stessi coefficienti ad ogni istante temporale. In una RNN possiamo definire la funzione di perdita totale $R(y, \theta)$ come somma delle funzioni di perdita ad ogni istante temporale. Il gradiente totale è quindi la somma dei gradienti calcolati ad ogni passo della sequenza. Concentrandoci su un generico istante temporale l, la derivata di $R_l(y_l, \theta)$ rispetto ai parametri W e B avviene come nel processo di *backpropagation* classico. Nel calcolo della derivata rispetto a U, cioè rispetto ai coefficienti che collegano ciclicamente gli strati ricorrenti, è possibile, invece, individuare una forma ricorsiva nel calcolo della derivata di A_l rispetto a U:

$$\frac{\partial A_l}{\partial U} = g'(z_l)(A_{l-1} + U\frac{\partial A_{l-1}}{\partial U}) \tag{3.11}$$

con $z_l = b + Wx_l + UA_{l-1}$ e $A_l = g(z_l)$. Quest'ultima è necessaria per calcolare tramite la regola della catena la derivata di R_l rispetto a U. Perciò, anche la derivata completa assume una forma ricorsiva evidenziando il contributo che apportano gli istanti temporali precedenti al calcolo del gradiente per il tempo corrente. Durante la fase di aggiornamento dei parametri, l'errore totale del modello viene quindi propagato all'indietro anche attraverso il tempo (oltre che negli strati). Il nome dell'algoritmo di BPTT deriva da questa caratteristica distintiva delle reti neurali ricorrenti.

Nel caso di lunghe sequenze temporali, l'algoritmo di BPTT soffre del problema di scomparsa o esplosione del gradiente che limitano le prestazioni anche di un semplice modello di rete neurale ricorrente. Infatti, nella retropropagazione dell'errore nel tempo, i gradienti vengono moltiplicati ripetutamente con le matrici dei coefficienti della RNN, poiché le stesse matrici vengono utilizzate in ogni passo temporale. Questo può portare a una rapida diminuzione dei gradienti nel caso in cui i coefficienti siano compresi tra 0 e 1 o le derivate delle funzioni di attivazione siano minori di 1 (vedi sezione 2.2.1) con conseguente scarso apporto di informazioni all'istante l da parte degli istanti temporali più lontani nel tempo. L'output all'istante l è quindi determinato solo dalle connessioni a pochi istanti temporali precedenti e per nulla da connessioni di lungo termine. Di conseguenza, la RNN potrebbe avere difficoltà nel catturare dipendenze a lungo termine all'interno della sequenza e può limitare la capacità della rete di fare previsioni accurate. Lo stesso vale nel caso in cui i coefficienti o le derivate delle funzioni di attivazione siano grandi il valore assoluto. Ciò può portare all'esplosione del gradiente con conseguente instabilità del modello.

Sono state definite numerose varianti della RNN standard in modo da riuscire a preservare l'informazione per numerosi istanti temporali, la più popolare tra queste è la *long short term memory* (LSTM). La caratteristica chiave di questo modello è l'aggiunta di un'unità di memoria c_l: (*memory cell*), che riesce a mantenere informazioni per lunghi periodi di tempo. La struttura della LSTM include tre porte essenziali (*gating units*): la porta di input, la porta di output e la porta di dimenticanza. Queste porte controllano il flusso delle informazioni all'interno della cellula c_l, permettendo di decidere quali informazioni memorizzare, quali scartare e quali utiliz-

zare per previsioni future. In conclusione, un modello di LSTM riesce non solo a memorizzare informazioni per lunghi periodi temporali impedendo il problema di scomparsa del gradiente, ma anche a stabilire quali sono le informazioni importanti da conservare e quali non lo sono, in modo da migliorare la velocità e l'accuratezza di previsione.

3.3.3 Deep K-Means Clustering

Il *deep clustering* è una tecnica di apprendimento non supervisionato che combina clustering e modelli di deep learning in modo da migliorare le performance rispetto al solo impiego di un clustering tradizionale.

Il clustering è una tecnica di analisi dei dati non supervisionata in cui lo scopo è definire dei gruppi in modo che le osservazioni all'interno del gruppo siano simili tra loro e le osservazioni in gruppi diversi siano dissimili. Esistono vari tipi di clustering, uno di questi è il *k-means*. Quest'ultimo metodo, dopo aver definito a priori il numero di cluster K, determina la partizione delle osservazioni minimizzando la distanza euclidea tra l'osservazione e la media del gruppo (centroide). Il gruppo k per cui questa distanza è minima sarà il gruppo di appartenenza dell'osservazione. Le partizioni iniziali vengono definite casualmente e ne vengono calcolati i centroidi. Vengono poi aggiornati i cluster spostando le osservazioni nel gruppo per il quale la distanza è minima e vengono ricalcolati i centroidi. Il processo viene iterato fino a convergenza, cioè quando nessuna osservazione si sposta più.

Nell'ambito della classificazione di immagini non supervisionata, un algoritmo di clustering tradizionale non è la scelta migliore in quanto non riesce a catturare e sfruttare le relazioni complesse nelle immagini. L'uso congiunto di un modello di rete neurale profonda permette di catturare le informazioni più rappresentative delle immagini che consentono poi all'algoritmo di clustering di distinguere meglio le immagini e raggrupparle in gruppi più coerenti. L'idea è quindi di utilizzare un autocodificatore (*autoencoder*) per ridurre la dimensionalità e catturare le caratteristiche più significative delle immagini, migliorando così l'effaciacia del clustering.

Un autocodificatore è un tipo di rete neurale non supervisionata utilizzata per predire il suo stesso input in modo da catturare solo gli aspetti rilevanti di dati non etichettati. La struttura è divisa in due parti: un *encoder* o codificatore che mappa l'input x in una sua diversa rappresentazione h e un *decoder* o decodificatore che produce la ricostruzione dell'input r. L'autocodificatore deve essere ideato appositamente per non ricostruire perfettamente l'input ($r \neq x$), altrimenti l'intero processo sarebbe inutile. La parte del modello che ci interessa è h, ovvero la rappresentazione latente di x. Imponendo che la dimensione di h sia minore di quella di x, forziamo il modello a codificare solo le caratteristiche rilevanti e le proprietà distintive dell'input. Infatti, se alla fase di *encoder* e *decoder* viene concessa troppa capacità (in termini di dimensione), il modello finisce semplicemente con il copiare l'input, senza estrarre informazioni utili.

Il processo di addestramento avviene minimizzando la funzione di perdita $R(x, r)$, con $r = g(f(x))$, f e g sono rispettivamente le funzioni di codifica e decodifica e R è la funzione di perdita da utilizzare (errore quadratico medio o entropia incrociata). La minimizzazione avviene attraverso l'algoritmo di discesa stocastica del gradiente e *backpropagation*. Quando alleniamo questo particolare tipo di rete neurale, l'etichetta vera corrispondente all'output del modello è

l'input originale. Quindi, la funzione di perdita R corrisponde a quanto male l'input originale è ricostruito dal modello.

Un altro modo per impedire che il modello finisca con il copiare l'input è introdurre un termine di penalità nella funzione di perdita descritta sopra. Possiamo, per esempio, aggiungere alla funzione di perdita una penalità dovuta alla sparsità sulla rappresentazione h in modo da vincolare il modello ad apprendere caratteristiche salienti e proprietà utili dai dati anche senza ridurre la dimensione d h:

$$R_1(x, r) = R(x, g(f(x))) + \Omega(h) \tag{3.12}$$

con $h = f(x)$. In questo caso il modello prende il nome di *sparse autoencoder* o autocodificatore sparso. L'idea di sparsità è che l'attivazione di un minor numero di neuroni nella fase di codifica garantirebbe che l'autoencoder apprenda effettivamente rappresentazioni latenti anziché informazioni ridondanti nei dati di input. La penalità $\Omega(h)$ può essere costruita tramite la funzione norma L_1:

$$\Omega(h) = \lambda \cdot L_1(h) = \lambda \sum_i |h_i| \tag{3.13}$$

con λ parametro che indica la forza della penalità. La norma L_1 produce un generale rimpicciolimento verso zero delle *activations* presenti in h, con qualcuna che viene anche posta pari a zero. In tal modo alcuni neuroni non vengono attivati e il modello raccoglie solo le caratteristiche distintive dell'input.

Finora, per comodità, abbiamo introdotto il modello di *autoencoder* con un solo strato nascosto. Tuttavia, modelli più *deep* con molti strati nascosti forniscono una potenza di rappresentazione molto migliore. La presenza di molti strati permette al modello di catturare anche rappresentazioni gerarchiche nei dati che, nel caso di dati immagine, sono particolarmente utili.

Capitolo 4

Un esempio pratico

L'obiettivo di questo capitolo è applicare le tecniche descritte nei capitoli precedenti ad un caso reale. Viene analizzato il dataset *Fashion Mnist*. Quest'ultimo contiene immagini di dieci diverse categorie di prodotti di moda, spaziando tra capi di abbigliamento, accessori e calzature. Lo scopo è di ottenere la miglior precisione possibile nella classificazione di ogni immagine. Le analisi sono svolte utilizzando il linguaggio di programmazione Python.

4.1 Il dataset

Il dataset *Fashion Mnist* è composto da un insieme di 70.000 immagini in scala di grigi di dieci diverse categorie di prodotti di moda (T-shirt/Top, Pantaloni, Maglione, Abito, Cappotto, Sandalo, Camicia, Sneaker, Borsa, Stivaletto) e da altrettante etichette che identificano la vera classe di appartenenza dell'immagine. Ogni classe è associata a un numero intero da 0 a 9.

Ogni immagine ha una risoluzione di 28×28 pixel, ciò significa che la bitmap di ogni immagine è una matrice di dimensione 28×28 in cui ogni valore misura l'intensità luminosa del corrispondente pixel nell'immagine. La dimensione del canale è di 8 bit, quindi il valore di intensità luminosa del pixel può variare da 0 a 255.

Delle 70.000 immagini totali, 60.000 formano il train set e le altre 10.000 formano il test set.

4.2 Caricamento dei dati e analisi esplorativa

Il dataset *Fashion Mnist* è già preinstallato nella libreria Keras (che approfondiremo nella sezione 4.5) e per caricarlo è sufficiente importarlo dal modulo *keras.datasets* tramite il comando *fashion_mnist.load_data*. Con questo metodo di importazione, il dataset viene già suddiviso in train e test set e ogni immagine è già disponibile tramite la rispettiva bitmap.

L'analisi esplorativa viene effettuata sul solo train set e riguarda tecniche e metodi che siano in grado di fornire una panoramica iniziale dei dati. Le tecniche utilizzate sul dataset *Fashion Mnist* comprendono:

- Visualizzazione di alcune immagini: tramite delle funzioni contenute nel modulo *matplotlib.pyplot* vengono mostrate a schermo alcune immagini del train set in modo da capire come sono strutturate.

- Controllo del bilanciamento delle classi: tramite un grafico a barre viene verificato che il bilanciamento delle classi sia adeguato. Ci sono 6.000 immagini per ognuna delle 10 classi, sono quindi perfettamente bilanciate.

- Calcolo di alcune statistiche descrittive per ogni classe: vengono calcolate media, terzo quartile e deviazione standard per ogni classe e riportate nella tabella 4.1. Queste statistiche si riferiscono all'intensità luminosa dei pixel. Notiamo che le classi "Sandalo" e "Sneaker" sono le due con la luminosità media e valore del terzo quartile più bassi. Ciò significa che le immagini appartenenti a queste classi sono particolarmente "scure" rispetto alle altre, ma non è detto che saranno anche le peggio classificate. Infatti, alcune categorie di abbigliamento sono molto simili tra loro sia in termini di immagine che di statistiche (Maglione, Cappotto, Camicia) e il modello farà sicuramente più fatica a discriminare tra queste ultime.

Classe	**Media**	**Terzo Quartile**	**Deviazione Standard**
T-shirt/Top	83.03	172.00	89.44
Pantaloni	56.84	127.00	87.60
Maglione	96.06	187.00	91.46
Abito	66.02	152.00	90.33
Cappotto	98.26	198.00	95.96
Sandalo	34.87	28.00	67.09
Camicia	84.61	166.00	86.52
Sneaker	42.76	55.00	75.17
Borsa	90.16	188.00	93.14
Stivaletto	76.81	181.00	94.49

Tabella 4.1: Media, terzo quartile e deviazione standard calcolate per immagini della rispettiva classe all'interno del dataset *Fashion Mnist*.

4.3 Preprocessing dei dati

Il preprocessing è uno step importante nell'analisi di qualsiasi dataset. Consiste nell'applicare una serie di trasformazioni ai dati grezzi, in modo da renderli adatti ed efficaci per l'addestramento del modello di machine learning da utilizzare. Le tecniche di preprocessing possono quindi variare in base al modello adottato. Le operazioni di preprocessing applicate al dataset *Fashion Mnist* sono:

- Normalizzazione: viene scalato il valore dei pixel nell'intervallo $[0; 1]$ dividendo ogni pixel per 255 (il valore massimo possibile). Questa operazione aiuta a migliorare e velocizzare il processo di addestramento.

- Modellazione dei dati: ogni modello può richiedere una dimensione diversa per le immagini in input. Tale operazione viene effettuata tramite il comando *reshape* della libreria *numpy*. La modellazione delle dimensioni è specificata nelle sezioni corrispondenti ad ogni modello.

- Codifica *one-hot*: i modelli di rete neurale richiedono, nel caso di classificazione in M classi, una codifica di tipo "*one-hot*" per le etichette. Secondo tale codifica, ogni categoria viene

rappresentata da un vettore binario di lunghezza M in cui la categoria di appartenenza viene indicata con 1 e tutte le altre vengono indicate con 0. Questa operazione viene effettuata con il comando *to_ categorical* del modulo *keras.utils*.

- Creazione di un validation set: vengono estratte 6.000 immagini dal train set tramite la funzione *train_ test_ split* della libreria Scikit-learn. Queste formano l'insieme di validazione.

4.4 Applicazione di modelli di machine learning

I modelli di *support vector machines* e *random forest* descritti nei capitoli precedenti sono implementati nella libreria Scikit-learn di Python. Quest'ultima fornisce un'ampia varietà di strumenti per la creazione di modelli di classificazione, regressione e clustering, oltre a numerose tecniche di selezione di variabili, riduzione della dimensionalità, preprocessing dei dati e valutazione del modello. Questa libreria fornisce anche una ricca documentazione relativa ai comandi e ai parametri di ciascun modello che è possibile implementare con essa.

4.4.1 Support Vector Machines

L'implementazione del modello di SVM avviene tramite alcune funzioni della libreria Scikit-learn. In particolare, viene prima creato l'oggetto *svm_ classifier* tramite la funzione *SVC* del modulo *svm*, con parametri predefiniti. Viene poi definita una griglia di parametri sui quali viene effettuata un'analisi in cross-validation per identificarne il valore migliore per ognuno di questi parametri (provandone tutte le combinazioni possibili) rispetto all'insieme di validazione. I parametri specificati sono:

1. *kernel*: specifica il tipo di *kernel* da utilizzare. Le opzioni fornite sono: radiale (*rbf*), polinomiale (*poly*) o lineare (*linear*).

2. *gamma*: specifica il coefficiente del *kernel* per il tipo radiale e polinomiale (corrisponde a γ e d nella 3.4 e 3.3). Indicando con X l'insieme delle esplicative, se $gamma = scale$ allora $gamma = \frac{1}{p \cdot var(X)}$, se invece $gamma = auto$ allora $gamma = \frac{1}{p}$, con p che è il numero di esplicative.

3. C: è il parametro di regolarizzazione. La forza della penalità (norma $L2$) è inversamente proporzionale a C (vedi sezione 3.2.1).

Una volta individuato il valore migliore per ogni parametro, il modello viene addestrato sui dati del train set e testato sul test set, sul quale produce un'accuratezza pari al 89.74%.

Prima dell'implementazione è però necessario modellare la dimensione dei dati in modo da renderla compatibile con il modello (vedi punto 2 del paragrafo 4.3). In particolare, il modello di SVM richiede un appiattimento della dimensione dei dati. Ogni immagine diventa quindi un vettore di lunghezza $28 \cdot 28 = 784$.

4.4.2 Random Forest

Anche il modello di *random forest* è implementato nella libreria Scikit-learn di Python. Innanzitutto viene creato un oggetto *rf* tramite la funzione *RandomForestClassifier* che inizializza il classificatore *random forest* con i parametri predefiniti. Come nel SVM, viene poi definita una griglia di parametri per il modello. I parametri specificati sono:

1. *n_estimators*: è il numero di alberi all'interno della foresta. Corrisponde al parametro B indicato nella sezione 3.2.2.

2. *min_samples_leaf*: specifica il numero minimo di unità che un nodo deve contenere per essere considerato un nodo foglia. Perciò, uno *split* può avvenire solamente se crea due nodi con almeno *min_samples_leaf* unità ciascuno.

3. *criterion*: indica il criterio utilizzato per misurare la qualità di una divisione o, all'opposto, l'impurità di un nodo. Le opzioni possibili sono: indice di Gini (*gini*), entropia (*entropy*) o entropia incrociata (*log_loss*).

4. *min_impurity_decrease*: valore numerico che indica la diminuzione minima di impurità richiesta per effettuare una divisione. Se la divisione comporta una diminuzione di impurità del nodo minore di questo parametro, allora la divisione non viene effettuata.

5. *max_features*: indica il numero di predittori considerati durante la ricerca della miglior divisione possibile. Corrisponde al parametro m indicato nella sezione 3.2.2. Indicando con p il numero di predittori, se *max_features* = *sqrt* allora $m = \sqrt{p}$, se invece *max_features* = *log2* allora $m = \log_2(p)$. È anche possibile indicare un numero specifico come valore per m.

Una volta definita la griglia, viene individuato il valore migliore per ogni parametro tramite cross-validation. Il modello così definito viene addestrato sul train set e testato sul test set, sul quale produce un'accuratezza pari al 83.87%.

Come per il modello di SVM, anche il modello di *random forest* richiede un appiattimento della dimensione dei dati, passando da matrici 28×28 ad array di lunghezza 784.

4.5 Applicazione di modelli di deep learning

Per l'implementazione dei modelli di rete neurale viene utilizzato il modulo Keras della libreria TensorFlow. Quest'ultima è una libreria *open source* rilasciata nel 2015 da Google, che fornisce moduli e comandi utili nella realizzazione di algoritmi di machine learning. Keras, inizialmente sviluppata da François Chollet come libreria a sè stante, è stata poi integrata in TensorFlow nel 2017 come API (*Application Programming Interface*) ufficiale, per facilitare la creazione e l'addestramento di reti neurali. Keras è quindi un modulo all'interno della libreria TensorFlow che ci è utile per sviluppare modelli di deep learning in modo semplice ed intuitivo. In particolare, la classe *Sequential* del modulo *keras.models* permette di creare modelli di rete neurale in modo sequenziale, ovvero disponendo gli strati uno dopo l'altro in sequenza.

L'algoritmo di ottimizzazione usato per l'addestramento dei modelli di rete neurale è ADAM, o *adaptive moment estimation* che è una variante dell'algoritmo di SGD implementata in modo da migliorarne le performance. Il principale miglioramento è legato soprattutto all'adattamento del *learning rate* (δ). L'idea di ADAM è di adattare il *learning rate* in relazione all'importanza dei coefficienti del modello. Durante la *backpropagation*, oltre all'aggiornamento dei coefficienti, vengono aggiornati anche i momenti del primo e secondo ordine dei gradienti calcolati. Il *learning rate* per ogni coefficiente è calcolato dividendo il *learning rate* del passo precedente per una combinazione dei due momenti dei gradienti. Questo miglioramento velocizza e rende più stabile l'algoritmo di ottimizzazione.

Oltre alla scelta dell'algoritmo di ottimizzazione, durante il processo di addestramento del modello è necessario determinare anche altri parametri:

- la funzione di perdita utilizzata (parametro *loss*). Nel caso in esame, cioè classificazione multiclasse, la scelta ricade sulla funzione di entropia incrociata categoriale (vedi equazione 2.12);
- la metrica di valutazione delle prestazioni del modello (parametro *metrics*). Solitamente viene utilizzata l'accuratezza, ovvero la percentuale di previsioni corrette rispetto al totale delle previsioni effettuate;
- la dimensione del batch (parametro *batch_size*). Indica il numero di osservazioni del train set che vengono selezionate casualmente per calcolare il gradiente della funzione di perdita durante ogni iterazione dell'algoritmo di discesa stocastica del gradiente (SGD) o di un altro algoritmo di ottimizzazione;
- il numero di epoche di addestramento (parametro *epochs*). Un'epoca corrisponde al numero di iterazioni che effettua l'algoritmo di SGD per processare tutte le n osservazioni del train set. Perciò ogni epoca corrisponde a $\frac{n}{batch\_size}$ iterazioni dell'algoritmo di ottimizzazione scelto, in modo tale da processare tutto il train set. Il numero di epoche indica, quindi, il numero di volte che il modello viene allenato su tutto il train set. Impostare un numero di epoche troppo alto può portare ad *overfitting*. Viceversa, un numero di epoche troppo basso implica pochi aggiornamenti dei coefficienti ed è quindi causa di un modello poco preciso.

4.5.1 Convolutional Neural Network

L'implementazione del modello di CNN avviene tramite la classe *Sequential* di *keras.models*. Il modello è formato da strati convoluzionali e di pooling, intervallati da alcune tecniche di regolarizzazione quali *batch normalization* e *dropout*. Prima dello strato di output è anche presente uno strato completamente connesso, preceduto da uno di *flatten* per appiattire la dimensione dei dati.

In particolare, il primo blocco è costituito da uno strato convoluzionale (classe *Conv2D*) con 64 filtri di dimensione 3×3 e funzione di attivazione *LeakyReLU*, e da uno strato di pooling (classe *MaxPooling2D*) che utilizza la funzione *max pooling* per riassumere ogni blocchetto 2×2 dei filtri ricevuti in input dallo strato convoluzionale. Tra i due è presente uno strato di *batch*

normalization. Uno strato di *dropout* del 30% è inserito dopo lo strato di pooling. Il secondo blocchetto ricalca esattamente il primo, diminuendo però il numero di filtri convoluzionali a 32. Dopo questi due blocchetti, lo strato di *Flatten* appiattisce la dimensione dell'input per il successivo strato completamente connesso (classe *Dense*) da 128 neuroni che precede lo strato di output. Quest'ultimo è uno strato completamente connesso da 10 neuroni con funzione di attivazione *softmax* che restituisce la classificazione predetta dal modello.

Il primo strato del modello, in questo caso lo strato convoluzionale, deve specificare la dimensione dei dati in input attraverso il parametro *input_shape*. Uno strato convoluzionale della classe *Conv2D* richiede in input dati di dimensione $m \times n \times d$ (vedi sezione 3.3.1), perciò è necessario rimodellare i dati aggiungendo la terza dimensione d. Ciascuna immagine deve quindi avere dimensione $28 \times 28 \times 1$.

La funzione di perdita utilizzata è l'entropia incrociata categoriale, la metrica di valutazione è l'accuratezza e l'algoritmo di ottimizzazione è ADAM.

Terminata la costruzione, il modello viene addestrato sul train set per 30 epoche con una dimensione del batch di 128, e successivamente testato sul test set, sul quale fornisce un'accuratezza che oscilla tra il 91.50% e il 92.30%. Queste oscillazioni sono dovute alle componenti stocastiche del modello di rete neurale, quali l'inizializzazione casuale dei pesi, la selezione dei batch di addestramento, e la mescolazione casuale dei dati (parametro *Shuffle*).

4.5.2 Recurrent Neural Network

Anche l'implementazione del modello di RNN avviene tramite la classe *Sequential* di Keras. Il modello è formato da due strati ricorrenti e da altri due strati completamente connessi che precedono lo strato di output. Gli strati sono intervallati da tecniche di regolarizzazione quali *batch normalization* e *dropout*.

In particolare, gli strati ricorrenti sono definiti dalla classe *SimpleRNN*. Questa classe di *layer* implementa uno strato ricorrente base, come quello descritto dall'equazione 3.9. Il primo strato è definito con 400 neuroni e il secondo con 64, la funzione di attivazione utilizzata è la tangente iperbolica (parametro *activation*) ed entrambi sono definiti con un coefficiente di *dropout* del 10%. Nel primo dei due, è necessario impostare il parametro *return_sequences* = *True* in modo da passare al secondo strato ricorrente l'intera sequenza dei dati e non solamente l'output dello strato. Dopo i due strati ricorrenti, viene inserito uno strato di *batch normalization* e successivamente due strati completamente connessi (classe *Dense*) con funzione di attivazione *LeakyReLU*, intervallati da uno strato di *dropout* del 15%. Infine, lo strato di output, è uno strato completamente connesso di 10 neuroni con funzione di attivazione *softmax*, che restituisce la classificazione predetta dal modello.

Il primo strato ricorrente deve contenere l'informazione sulla dimensione dei dati in input, che si indica attraverso il parametro *input_shape*. In questo caso non è necessario rimodellare la dimensione dei dati, perciò ogni immagine mantiene la sua dimensione originale 28×28.

La funzione di perdita utilizzata è l'entropia incrociata categoriale, la metrica di valutazione è l'accuratezza e l'algoritmo di ottimizzazione è SGD. Infatti, per il modello in esame, è stato verificato che l'algoritmo di SGD fornisce prestazioni migliori rispetto all'algoritmo ADAM.

Una volta terminata la costruzione, il modello viene addestrato sui dati del train set per 30 epoche con una dimensione del batch di 128, e successivamente testato sul test set, sul quale fornisce un'accuratezza che oscilla tra l'88% e l'89%. Come per il modello di CNN, queste oscillazioni sono dovute alle componenti stocastiche del modello.

4.5.3 Deep K-Means Clustering

Per questo tipo di modello vengono confrontati due clustering effettuati su dati diversi: il primo viene calcolato su una rappresentazione latente dei dati ottenuta attraverso un modello di *autoencoder*, mentre il secondo è calcolato sui dati originali.

L'implementazione del modello di rete neurale *autoencoder* avviene attraverso la classe *Sequential* di Keras. Il modello è formato da un insieme di strati completamente connessi con funzione di attivazione *ReLU* implementati attraverso la classe *Dense*, ed è suddiviso in una parte di codifica (*encoder*) e una parte di decodifica (*decoder*). Le due parti (che sono specchiate) sono separate dallo strato latente, dal quale viene estratta l'omonima rappresentazione dell'input. Agli strati dell'*encoder* è stato anche aggiunto un termine di regolarizzazione: *activity_regularizer = regularizers.l1* che effettua una regolarizzazione di tipo L_1 alle *activations* dello strato sul quale viene applicata (vedi equazione 3.13). Lo strato di output, che fornisce la ricostruzione, deve essere formato da un numero di neuroni pari alla dimensione dell'input e con funzione di attivazione *sigmoid*, in quanto l'array da ricostruire è formato da valori compresi tra 0 e 1, e la *sigmoid* restituisce valori in questo stesso intervallo basandosi sull'input ricevuto dagli strati precedenti.

Uno strato della classe *Dense* richiede input unidimensionali, è perciò necessario rimodellare la dimensione dei dati ad un array di dimensione $28 \cdot 28 = 784$. La dimensione dell'input è quindi 784, mentre la dimensione dello spazio latente è impostata a 32. La riduzione è molto consistente e verranno estratte solo le caratteristiche più peculiari di ciascuna immagine.

L'algoritmo di ottimizzazione utilizzato è ADAM e la funzione di perdita è l'entropia incrociata binaria. Quest'ultima si utilizza al posto dell'entropia incrociata categoriale in quanto l'obiettivo è stabilire se la ricostruzione dell'input è avvenuta correttamente o meno (risposta binaria) e non determinare una classificazione multiclasse. Per lo stesso motivo non si utilizza alcuna metrica di valutazione, e la bontà del modello è valutata attraverso la funzione di perdita.

Terminata la definizione del modello, questo viene addestrato sul train set per 30 epoche e con una dimensione del batch di 256. Vengono poi definiti i modelli di *encoder* e *decoder*. Il primo viene testato sui dati del test set, ottenendone la rappresentazione latente (oggetto *embeddings*). Su quest'ultimo viene effettuato un clustering K-Means tramite la funzione *KMeans* della libreria Scikit-learn, ipotizzando che il numero di cluster sia noto a priori. Lo stesso metodo di clustering viene applicato sulle immagini originali contenute nel test set. La valutazione della bontà del clustering è determinata tramite la *silhouette*, una metrica che misura quanto bene le osservazioni sono state divise nei rispettivi gruppi attraverso una combinazione di misure di similarità all'interno del gruppo e dissimilarità rispetto agli altri gruppi. Tale valore è compreso nell'intervallo $[-1; 1]$ e, tanto più il valore è vicino a uno, tanto migliore a la divisione in cluster. La *silhouette* è calcolata attraverso la funzione *silhouette_score* della libreria Scikit-learn. Nel caso in esame, il punteggio di *silhouette* per il clustering tradizionale è di 0.16, mentre per il

deep clustering varia all'incirca tra 0.27 e 0.30, garantendo un discreto miglioramento rispetto al clustering tradizionale. Anche in questo caso l'oscillazione del valore della *silhouette* è causata dalle componenti stocastiche dal modello di rete neurale.

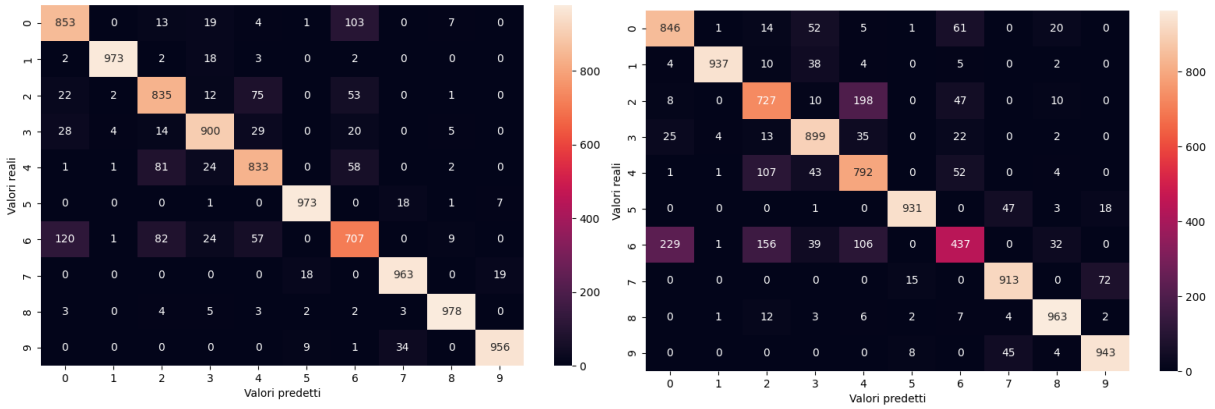
4.6 Valutazione e confronto

Nel contesto supervisionato, la valutazione e il confronto dei vari modelli viene effettuato tramite l'accuratezza di classificazione e la matrice di confusione (*confusion matrix*). Tale matrice evidenzia le discrepanze tra la previsione effettuata dal modello e la classificazione reale, per ogni osservazione appartenente al test set. In particolare, nelle colonne vengono rappresentati i valori predetti e nelle righe vengono rappresentati i valori reali. L'elemento in corrispondenza della riga i e della colonna j della matrice indica il numero di casi in cui il modello ha classificato la vera classe i come classe j. Perciò, tutti gli elementi che stanno fuori dalla diagonale sono gli errori di classificazione che commette il modello.

La figura 4.1 riporta le matrici di confusione per ognuno dei quattro modelli supervisionati analizzati. Notiamo che in tutti i modelli la classe peggio classificata è la numero 6, cioè la classe Camicia, che molte volte viene classificata come 0, 2 o 4, cioè come T-shirt/Top, Maglione o Cappotto. In effetti, guardando alcune immagini di questo tipo di capi d'abbigliamento, si nota una certa somiglianza. In più, la bassa risoluzione delle immagini agevola la "confusione" tra queste quattro classi. La CNN è il modello che riesce a discriminare meglio tra le classi Camicia, T-shirt/Top, Maglione e Cappotto. Infatti, analizzando piccole regioni di immagine alla volta tramite i filtri convoluzionali, il modello di CNN riesce a cogliere piccoli dettagli ma che sono significativi per distinguere queste classi apparentemente simili tra loro. All'opposto, la classe meglio classificata è la numero 8, che corrisponde alla categoria Borsa. È facile capire il motivo. La borsa è fisicamente la meno simile rispetto a tutte le altre classi di accessori d'abbigliamento presenti nel dataset.

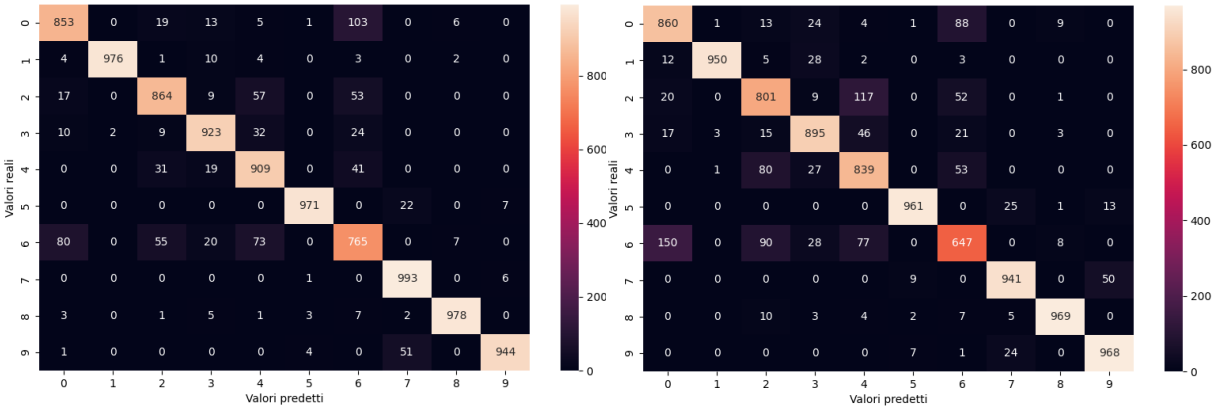
Dal punto di vista della precisione complessiva, il modello di CNN fornisce l'accuratezza di classificazione migliore, arrivando intorno al 92%. Infatti, come detto nella sezione 3.3.1, la struttura di tale modello (strati convoluzionali e di pooling) lo rende particolarmente adatto all'analisi di dati in formato immagine e, ad oggi, è lo standard per l'analisi di questo tipo di dati. Il modello di RNN fornisce invece un'accuratezza di classificazione di circa l'88%. Tuttavia, la potenzialità di questo modello non vengono sfruttate a pieno in un contesto di classificazione di immagini, in quanto il modello di RNN è ideale per l'analisi di dati che presentano una struttura sequenziale nel tempo. Riescono comunque a fornire un'ottima accuratezza anche in questo ambito. Di poco superiore alla RNN è il modello di SVM. Tale modello fornisce un'accuratezza che arriva quasi al 90% e rappresenta un'ottima alternativa alla CNN. L'ottimo risultato raggiunto dal SVM è sicuramente dovuto all'ottimizzazione dei parametri effettuata tramite cross-validation. Non si può dire lo stesso del modello di *Random Forest*, che fornisce un'accuratezza di poco inferiore all'84%. Si tratta comunque di una buona precisione di classificazione, ma inferiore del 10% circa rispetto alla CNN, il che non lo rende la scelta migliore per il problema in esame.

Per quanto riguarda il contesto non supervisionato, la valutazione e il confronto tra l'algoritmo di clustering *K-Means* tradizionale e il *deep K-Means* è effettuata sulla base della *silhouette*. Come detto nella sezione 4.5.3, l'aumento della *silhouette* nel *deep clustering* è considerevole



(a) SVM confusion matrix

(b) Random Forest confusion matrix



(c) CNN confusion matrix

(d) RNN confusion matrix

Figura 4.1: Matrici di confusione per i quattro modelli supervisionati analizzati: *Support Vector Machines*, *Random Forest*, *Convolutional Neural Network*, *Recurrent Neural Network*.

rispetto al metodo tradizionale (aumenta di circa 0.12) e ciò giustifica l'utilizzo dell'*autoencoder* per estrarre le caratteristiche più significative delle immagini. Essendo un metodo non supervisionato, la sua accuratezza sarà sempre inferiore rispetto ad un metodo supervisionato. Tuttavia, nel caso in cui sia necessario effettuare una classificazione non disponendo delle etichette reali, il *deep clustering* rappresenta comunque un valido metodo.

Conclusioni

L'obiettivo di questo lavoro era analizzare e confrontare alcune tecniche e modelli nell'ambito della classificazione di immagini, con particolare enfasi riguardo a modelli di rete neurale. A tale scopo, questo tipo di modelli viene descritto in dettaglio nei capitoli 2 e 3. Dal capitolo 4, invece, è possibile trarre alcune conclusioni riguardo la loro efficacia nel contesto della classificazione di immagini.

Dall'analisi e dal confronto delle diverse tecniche e modelli applicati al dataset *Fashion Mnist*, è emersa la superiorità, in termini di precisione di classificazione, del modello di *Convolutional Neural Network* rispetto a tutti gli altri. Questo risultato era ampiamente prevedibile dato che il modello di CNN è stato progettato appositamente per l'analisi delle immagini e per problemi di visione artificiale. Va anche sottolineato che un altro modello, il *Support Vector Machines*, ha raggiunto un livello di accuratezza di classificazione solo di poco inferiore alla CNN (circa un 2%). Questo risultato conferma l'efficacia del SVM come valida alternativa a modelli di rete neurale anche in un contesto di classificazione di immagini. Tuttavia le CNN, grazie allo loro capacità di catturare dettagli piccoli ma significativi e alla capacità di apprendere caratteristiche in modo gerarchico, restano il modello da preferire nell'ambito della classificazione di immagini.

Il modello di CNN analizzato in questo lavoro raggiunge un'accuratezza di circa il 92% sul dataset *Fashion Mnist*. Tuttavia, è importante notare che esistono modelli di CNN molto più complessi e computazionalmente costosi in grado di raggiungere livelli di precisione ancora più elevati, arrivando al 96% e oltre su questo stesso dataset.

In conclusione, questo lavoro ha analizzato l'efficacia di vari modelli nella classificazione di immagini, con le CNN che primeggiano grazie alla loro capacità di catturare dettagli significativi nelle immagini. Tuttavia, anche gli altri modelli presentati (in primis le SVM) non sono da scartare.

Appendice A

Codici Python

A.1 Analisi Esplorativa

```
from keras.datasets import fashion_mnist
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

# Carico il dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

# Definisco il vettore di etichette corrispondenti alla classificazione
labels = np.array(['T-shirt/top', 'Pantaloni', 'Maglione', 'Abito',
                   'Cappotto', 'Sandalo', 'Camicia', 'Sneaker',
                   'Borsa', 'Stivaletto'])

# 1) mostro a schermo qualche immagine
def plot_images(img1,img2,img3,img4,img5,img6):
    images = [img1, img2, img3, img4, img5, img6]
    for i in range(6):
        plt.subplot(3,2,i+1)
        plt.imshow(x_train[images[i]], cmap='gray')
        plt.title(labels[y_train[images[i]]])
        plt.axis('off')
    return(plt.show())

plot_images(22,456,8632,55867,22756,9745)

# 2) verifico il bilanciamento delle classi
def bar_plot():
    unique_labels, label_counts = np.unique(y_train, return_counts=True)
    plt.figure(figsize=(11,6))
    plt.bar(labels[unique_labels], label_counts, color='darkblue')
    plt.xlabel('Classe di abbigliamento')
    plt.ylabel('Frequenza')
```

```
    plt.ylim(0,6500)
    plt.title('Distribuzione delle etichette di classe')
    return(plt.show())

bar_plot()

# 3) calcolo media, terzo quartile e deviazione standard per ogni
# classe e le salvo in una lista di dizionari
statistiche = []
for class_label in np.unique(y_train):
    class_indices = np.where(y_train == class_label)
    class_images = x_train[class_indices]
    class_q3 = np.percentile(class_images, 75)
    class_mean = np.mean(class_images)
    class_std = np.std(class_images)
    class_stat_dict = {
        'Classe': labels[class_label],
        'Media': f'{class_mean:.2f}',
        'Terzo quartile': f'{class_q3:.2f}',
        'Deviazione Standard': f'{class_std:.2f}'}
    statistiche.append(class_stat_dict)

# Creo un DataFrame e lo esporto in formato LaTeX
df = pd.DataFrame(statistiche)
print(df)
latex_table = df.to_latex(index=False, escape=False)
```

A.2 SVM

```
from sklearn import svm
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import confusion_matrix
from keras.datasets import fashion_mnist
import matplotlib.pyplot as plt
import seaborn as sb
import numpy as np

# Carico il dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

#### Preprocessing
# Normalizzo il valore dei pixel in scala [0,1]
x_train = x_train / 255
x_test = x_test / 255

# Trasformo gli array da bidimensionali a unidimensionali
x_train = x_train.reshape(60000,28*28)
```

```
x_test = x_test.reshape(10000,28*28)

# Estraggo un insieme di 6000 immagini come insieme di validazione
val_size = 0.10
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
                                                  test_size=val_size,
                                                  random_state=11)
#### Training
svm_classifier = svm.SVC()
parameters = [{'kernel': ['rbf', 'poly'], 'gamma': ['scale', 'auto'],
               'C': [1, 10, 100]},
              {'kernel': ['linear'], 'C': [1, 10, 100]}]
grid = GridSearchCV(svm_classifier, parameters, cv=5, verbose=3)
grid.fit(x_val, y_val)

print(grid.best_params_)
print(grid.best_estimator_)

model = grid.best_estimator_
model.fit(x_train, y_train)

#### Testing
y_pred = model.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)*100
print("Precisione: {:.2f}%".format(accuracy))      # acc: 89.71%
print(classification_report(y_test, y_pred))

# Mostro a schermo la matrice di confusione
def plot_cm():
    cm = confusion_matrix(y_test, y_pred)
    plt.subplots(figsize=(10, 6))
    sb.heatmap(cm, annot = True, fmt = 'g')
    plt.xlabel("Valori predetti")
    plt.ylabel("Valori reali")
    plt.title("SVM Confusion Matrix")
    return(plt.show())

plot_cm()
```

A.3 Random Forest

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import confusion_matrix
from keras.datasets import fashion_mnist
import matplotlib.pyplot as plt
import seaborn as sb
```

```
# Carico il dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

#### Preprocessing
# Normalizzo il valore dei pixel in scala [0,1]
x_train = x_train / 255
x_test = x_test / 255

# Trasformo gli array da bidimensionali a unidimensionali
x_train = x_train.reshape(60000,28*28)
x_test = x_test.reshape(10000,28*28)

# Estraggo un insieme di 6000 immagini come insieme di validazione
val_size = 0.10
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
                                                  test_size=val_size,
                                                  random_state=11)
#### Training
rf = RandomForestClassifier()
parameters = {'n_estimators':[200,500,600], 'min_samples_leaf':[2,3],
              'criterion':['gini', 'entropy', 'log_loss'],
              'min_impurity_decrease':[0.01, 0.001],
              'max_features':['sqrt', 'log2']}
grid = GridSearchCV(rf, parameters, cv=2, verbose=3)
grid.fit(x_val,y_val)

print(grid.best_params_)
print(grid.best_estimator_)

model = grid.best_estimator_
model.fit(x_train, y_train)

#### Testing
y_pred = model.predict(x_test)
accuracy = accuracy_score(y_test, y_pred)*100
print("Precisione: {:.2f}%".format(accuracy))        # acc: 83.87%
print(classification_report(y_test, y_pred))

# Mostro a schermo la matrice di confusione
def plot_cm():
    cm = confusion_matrix(y_test, y_pred)
    plt.subplots(figsize=(10, 6))
    sb.heatmap(cm, annot = True, fmt = 'g')
    plt.xlabel("Valori predetti")
    plt.ylabel("Valori reali")
    plt.title("Random Forest Confusion Matrix")
    return(plt.show())
```

```
plot_cm()
```

A.4 CNN

```
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Flatten, Dropout, MaxPooling2D
from keras.layers import BatchNormalization, LeakyReLU
from keras.utils import to_categorical
from keras.datasets import fashion_mnist
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
import matplotlib.pyplot as plt
import seaborn as sb
import numpy as np

# Carico il dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

#### Preprocessing
# Normalizzo il valore dei pixel in [0:1]
x_train = x_train / 255
x_test = x_test / 255

# Estraggo 6000 immagini dal train set come insieme di validazione
val_size = 0.10
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
                                                  test_size=val_size,
                                                  random_state=11)
# Codifico i vettori risposta in 'one-hot' vector
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
y_val = to_categorical(y_val)

# Modifico la dimensione dei dati
x_train = x_train.reshape(54000,28,28,1)
x_test = x_test.reshape(10000,28,28,1)
x_val = x_val.reshape(6000,28,28,1)

#### Training
# Definizione del modello
model = Sequential([
    Conv2D(64, kernel_size=(3, 3), input_shape=(28, 28, 1)),
    LeakyReLU(alpha=0.01),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.30),
```

```
    Conv2D(32, kernel_size=(3, 3)),
    LeakyReLU(alpha=0.01),
    BatchNormalization(),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.30),

    Flatten(),

    Dense(units=128),
    LeakyReLU(alpha=0.01),
    Dense(units=10, activation='softmax')])

print(model.summary())
model.compile(optimizer='adam', loss='categorical_crossentropy',
              metrics=['accuracy'])
History = model.fit(x_train, y_train, validation_data=(x_val, y_val),
                    epochs=30, batch_size=128, shuffle=True)

#### Testing
test_loss, test_acc = model.evaluate(x_test, y_test)
print("Precisione: {:.2f}%".format(test_acc*100))      # acc: 91.96%
y_pred = model.predict(x_test)
# Ritrasformo le etichette one-hot in classi
y_test_class = np.argmax(y_test, axis=1)
y_pred_class = np.argmax(y_pred, axis=1)
print(classification_report(y_test_class, y_pred_class))

# Mostro a schermo la matrice di confusione
def plot_cm():
    cm = confusion_matrix(y_test_class, y_pred_class)
    plt.subplots(figsize=(10, 6))
    sb.heatmap(cm, annot = True, fmt = 'g')
    plt.xlabel("Valori predetti")
    plt.ylabel("Valori reali")
    plt.title("CNN Confusion Matrix")
    return(plt.show())

plot_cm()
```

A.5 RNN

```
from keras.models import Sequential
from keras.layers import Dense, SimpleRNN, BatchNormalization
from keras.layers import LeakyReLU, Dropout
from keras.utils import to_categorical
from keras.datasets import fashion_mnist
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
```

```
import matplotlib.pyplot as plt
import seaborn as sb
import numpy as np

# Carico il dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

#### Preprocessing
# Normalizzo i pixel nell'intervallo [0;1]
x_train = x_train / 255
x_test = x_test / 255

# Estraggo un insieme di 6000 immagini come insieme di validazione
val_size = 0.10
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
                                                  test_size=val_size,
                                                  random_state=11)
# Codifico i vettori risposta in 'one-hot' vector
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
y_val = to_categorical(y_val)

#### Training
image_size = x_train.shape[1]      # 28
time_steps = image_size
input_dim = image_size

# Definizione del modello
model = Sequential()
model.add(SimpleRNN(units=400, activation='tanh', dropout=0.1,
                    input_shape=(input_dim,time_steps),
                    return_sequences=True))
model.add(SimpleRNN(units=64, activation='tanh', dropout=0.1))
model.add(BatchNormalization())
model.add(Dense(units=64))
model.add(LeakyReLU(alpha=0.01))
model.add(Dropout(0.15))
model.add(Dense(units=32))
model.add(LeakyReLU(alpha=0.01))
model.add(Dense(units=10, activation='softmax'))

model.summary()
model.compile(loss='categorical_crossentropy', optimizer='sgd',
              metrics=['accuracy'])
History = model.fit(x_train, y_train, validation_data=(x_val,y_val),
                    epochs=30, batch_size=128, shuffle=True)

#### Testing
```

```
loss, acc = model.evaluate(x_test, y_test)
print("Precisione: %.2f%%" % (acc * 100))          # acc: 88.44%
y_pred = model.predict(x_test)
# Ritrasformo le etichette one-hot in classi
y_test_class = np.argmax(y_test, axis=1)
y_pred_class = np.argmax(y_pred, axis=1)
print(classification_report(y_test_class, y_pred_class))

# Mostro a schermo la matrice di confusione
def plot_cm():
    cm = confusion_matrix(y_test_class, y_pred_class)
    plt.subplots(figsize=(10, 6))
    sb.heatmap(cm, annot = True, fmt = 'g')
    plt.xlabel("Valori predetti")
    plt.ylabel("Valori reali")
    plt.title("RNN Confusion Matrix")
    return(plt.show())

plot_cm()
```

A.6 Deep K-Means Clustering

```
from keras.models import Sequential
from keras.layers import Dense
from keras import regularizers, Model
from keras.datasets import fashion_mnist
from sklearn.cluster import KMeans
from sklearn.model_selection import train_test_split
from sklearn.metrics import silhouette_score

# Carico il dataset
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_data()

#### Preprocessing
# Normalizzo i pixel nell'intervallo [0;1]
x_train = x_train / 255
x_test = x_test / 255

# Estraggo un insieme di 6000 immagini come insieme di validazione
val_size = 0.10
x_train, x_val, y_train, y_val = train_test_split(x_train, y_train,
                                                  test_size=val_size,
                                                  random_state=11)
# Modifico la dimensione dei dati
x_train = x_train.reshape((54000, 28 * 28))
x_test = x_test.reshape((10000, 28 * 28))
x_val = x_val.reshape((6000, 28 * 28))
```

```
#### Training e Testing
# Definizione dell'architettura dell'autoencoder
input_dim = 784     # Dimensione dei dati di input
latent_dim = 32     # Dimensione della reppresentazione latente
autoencoder = Sequential([
    # encoder
    Dense(256, activation='relu', input_shape=(input_dim,),
          activity_regularizer=regularizers.l1(10e-4)),
    Dense(128, activation='relu',
          activity_regularizer=regularizers.l1(10e-4)),
    Dense(64, activation='relu',
          activity_regularizer=regularizers.l1(10e-4)),
    Dense(latent_dim, activation='relu'),
    # decoder
    Dense(64, activation='relu'),
    Dense(128, activation='relu'),
    Dense(256, activation='relu'),
    Dense(input_dim, activation='sigmoid')])

print(autoencoder.summary())
autoencoder.compile(optimizer='adam', loss='binary_crossentropy')
History = autoencoder.fit(x_train,x_train, validation_data=(x_val,x_val),
                          epochs=30, batch_size=256, shuffle=True)

# Il modello encoder fornisce la rappresentazione embedded dell'input,
# il modello decoder fornisce la ricostruzione dell'input
encoder = Model(autoencoder.input, autoencoder.layers[3].output)
embeddings = encoder.predict(x_test)      #rappresentazione latente
decoder = Model(autoencoder.input, autoencoder.output)
decoddings = decoder.predict(x_test)      # ricostruzione dell'input

# Applicazione del clustering K-means sulle rappresentazioni embedded
num_clusters = 10
kmeans = KMeans(n_clusters=num_clusters, n_init=50,
                max_iter=1000, random_state=123)
kmeans.fit(embeddings)
cluster_labels = kmeans.predict(embeddings)
silhouette = silhouette_score(embeddings, cluster_labels)
print("Silhouette score: {:.2f}".format(silhouette))       # sil: 0.28


#### Confronto con il K-Means applicato sui dati originali
num_clusters = 10
kmeans_std = KMeans(n_clusters=num_clusters, n_init=50,
                    max_iter=1000, random_state=123)
cluster_labels_2 = kmeans_std.fit_predict(x_test)
silhouette2 = silhouette_score(x_test, cluster_labels_2)
print("Silhouette score: {:.2f}".format(silhouette2))      # sil: 0.16
```

Bibliografia

[1] Charu C Aggarwal et al. *Neural networks and deep learning*. Springer, 2018.

[2] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[3] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[4] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[5] Jure Leskovec, Anand Rajaraman, and Jeffrey David Ullman. *Mining of Massive Datasets*. Cambridge university press, 2020.

[6] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.