Riconoscimento di Cifre Scritte a Mano

Gruppo di Lavoro:
Fabio Palmisano, [MAT. 720333], f.palmisano44@studenti.uniba.it

Link al Repository del Progetto:
Progetto

AA 2023-24

Indice

Capitolo 1) Introduzione

Capitolo 1.1) Ispirazione e Obiettivi

Il progetto nasce dal desiderio di comprendere il funzionamento di qualcosa che mi lasciò di stucco qualche anno fa: una lavagna intelligente, capace di risolvere equazioni ed effettuare calcoli autonomamente su qualsiasi cosa che noi decidessimo di scrivere a mano su di essa, tramite il solo dito o pennino.

Inoltre, questa idea si ricollega molto concettualmente alle app per smartphone molto scaricate ultimamente dagli studenti di materie scientifiche di tutte le età: app che permettono, tramite fotocamera, di riconoscere automaticamente una equazione scritta sul nostro quaderno, e di risolverla.

L'idea base che permette l'esistenza di queste due applicazioni è il riconoscimento di cifre, caratteri e operatori. Inoltre, il dettaglio più importante che mi ha spinto all'approfondimento su questo argomento, è che queste entità da riconoscere hanno la particolarità di essere scritte a mano.

Questa particolarità rende il compito del riconoscimento molto arduo: ogni persona ha un modo diverso di scrivere a mano, e una singola persona potrebbe scrivere gli stessi caratteri in modi totalmente diversi, anche a distanza di pochi secondi.

Quindi, lo scopo di questo progetto è l'implementazione di un riconoscitore di cifre, capace di prendere in input immagini contenenti una cifra scritta a mano, e classificarle in base alla cifra riconosciuta.

Capitolo 1.2) Requisiti Funzionali

Il progetto è stato realizzato in Python, linguaggio di programmazione che offre molte librerie che permettono di trattare e manipolare dei dataset oltre che per implementare i vari modelli supervisionati.

La versione utilizzata è Python 3.10 e l'IDE su cui è stato sviluppato e testato il progetto è PyCharm.

Librerie di Terze Parti Utilizzate:

- **numpy**: libreria per gestire array: molto utile per le nostre implementazioni
- **scikit_learn**: libreria che contiene implementazioni di modelli per l'apprendimento automatico. Non verrà utilizzato nell'implementazione del classificatore finale.

Installazione e avvio:

Una volta scaricato e aperto il progetto, bisogna eseguire la fase di installazione dei requisiti, che provvederà a installare le librerie necessarie sul virtual environment, tutto tramite il file "requirements.txt".

Capitolo 2) Creazione del Dataset

Capitolo 2.1) Scelta del Dataset

Tenendo in considerazione che il tipo di argomento di interesse per il progetto è un qualcosa di studiato frequentemente dalle persone, e considerando che il creare un dataset da zero con cifre scritte a mano richiederebbe un tempo assurdamente lungo, ho deciso di ricercare i dataset disponibili in rete che potrebbero fare a caso mio.

Per il mio sistema ho dunque deciso di usare il "MNIST Dataset".

Questo dataset contiene 70.000 immagini di cifre(da 0 a 9) scritte a mano, tutte ridotte alla stessa dimensione e centrate in una griglia quadrata di pixel. Ogni immagine è un array [28,28,1], dove le prime due grandezze sono le dimensioni in pixel dei lati del quadrato, e 1 è il canale del colore, in quanto tutte le immagini saranno solo bianco e nero, dove l'intensità dei pixel sarà 0 quando il loro colore è nero, e 1 quando il colore è bianco, e valori intermedi saranno varie intensità di grigio.

Inoltre, i dati target sono degli array di dimensioni 10 (ognuno di questi descrive una cifra), e ad ognuna di queste viene associato un valore tra 0 e 1, per quantificare con quanta certezza una specifica immagine rappresenta una determinata cifra.

Capitolo 2.2) Preprocessing del Dataset

Il preprocessing del dataset raccoglie quelle operazioni propedeutiche necessarie a rendere il dataset pronto ad essere utilizzato.

Il Dataset è già abbastanza pronto, l'unica operazione assolutamente importante è lo "split": bisogna dividere i dati in pattern di addestramento e pattern di test (rispettivamente training data e test data).

In questo modo, tramite split, abbiamo ottenuto su un totale di 70.000 immagini, 60.000 di addestramento, e 10.000 di test.

NB: Per raccomandazione, ho provato a usare lo split ulteriormente sull'insieme di dati di addestramento, sottraendone altri 10.000 per futuri test di convalida(risultando così un una suddivisione dei dati in gruppi da 50.000+10.000+10.000), ma per questioni di scala del progetto e di tempo, ho deciso di scartare l'idea.

Capitolo 3) Apprendimento Supervisionato: Parte 1

Capitolo 3.1) Introduzione all'Apprendimento Supervisionato

L'apprendimento supervisionato è una branca dell'apprendimento automatico all'interno di cui un modello viene addestrato su un insieme di dati in input con etichette(come nel nostro caso). Ne esistono due tipi:

- **Classificazione**: I possibili valori delle etichette appartengono ad un dominio finito numerico discreto. Un esempio è la classificazione booleana, dove l'etichetta può assumere valore *true* o *false*.
- **Regressione**: Le etichette possono assumere qualsiasi valore reale, dunque il dominio a cui appartiene è numerico e continuo.

Lo scopo dell'apprendimento supervisionato è far si che l'algoritmo definisca delle relazioni tra i valori dell'insieme degli input e degli output tramite etichette ad essi associate in modo che, una volta effettuato l'addestramento, il sistema possa fare previsioni su nuovi dati che non presentano alcuna etichetta.

Di solito, per addestrare un modello, è necessario seguire 4 fasi:

1. Scelta degli iper-parametri;
2. Fase di addestramento;
3. Fase di test;
4. Valutazione delle prestazioni.

Ma prima, bisognerebbe specificare dunque quale tipo di apprendimento supervisionato andremo ad applicare. E quello che rispecchia le nostre esigenze è la **Classificazione**, poiché il nostro sistema prevede un insieme finito di 10 classi possibili (cifre da 0 a 9), tutte ben distinte fra loro.

Capitolo 3.2) Riflessione su Modelli e Strategie

È importante scegliere il modello di apprendimento supervisionato più adatto al nostro progetto. I modelli presi in considerazione inizialmente, in teoria, sono:

- **Alberi Decisionali:** è un classificatore strutturato con un Albero in cui la radice e i nodi interni rappresentano delle condizioni sulle feature date in input, mentre le foglie rappresentano le classi di appartenenza dell'input o, almeno, le probabilità di appartenenza a tali classi. La classe viene scelta navigando l'albero, scegliendo il percorso in base a quali condizioni vengono rispettate o meno.
- **Random Forest:** è un classificatore ottenuto creanto tanti Alberi Decisionali. Il valore di output è ottenuto tramite media sulle predizioni di ogni albero della foresta.
- **SVM(Support Vector Machine):** è un classificatore che trova una linea o un piano ottimale che separa ogni classe nello spazio dimensionale, massimizzando la distanza fra questo separatore e le classi (prendendo i punti/dati più vicini all'altra classe).
- **ANN(Artificial Neural Network):** è un classificatore che per ora verrà tralasciato: entrerò nel dettaglio del suo utilizzo e della sua implementazione nel **Capitolo 4**.

L'ideale sarebbe valutare ogni singolo modello prima di scegliere quello che fa al caso nostro, ottimizzando gli iper-parametri tramite algoritmi per la loro scelta (tra cui Bayes Search, Grid Search o Random Search). Questo ci permette l'analisi delle prestazioni del modello.

Ma, avendo un personale bias verso le Reti Neurali(che ho deciso di sviluppare e approfondire), e per mancanza di tempo, ho fatto alcune veloci considerazioni:

- Dato che abbiamo un dataset di grandi dimensioni, composto da immagini(e quindi dati con molte informazioni), ho pensato che una SVM potesse fare a caso mio rispetto agli Alberi Decisionali (anche perché questi ultimi sembrano più suscettibili a fenomeni di overfitting causati dal grande numero di features).
- Le Reti Neurali possono rappresentare qualsiasi funzione su feature discrete, includendo tutte quelle implementabili con gli alberi di decisione, e questo mi permette di focalizzarmi più sulle prime.
- Generalmente le Reti Neurali sono più flessibili e scalabili delle SVM, e meno dispendiose da addestrare in confronto.

Dunque, ho deciso di implementare una semplicissima SVM di prova, per vedere che risultati mi avrebbe risultato di base, per poi focalizzare tutto il progetto su di una Rete Neurale, su cui si baserà il progetto principale.

Capitolo 3.3) Implementazione SVM e Risultati

Come accennato in precedenza, la SVM verrà implementata soltanto come prova.

Per l'implementazione, verrà utilizzata la libreria **sklearn** che permette una rapida implementazione di una SVM. Non focalizzando il progetto sulla SVM, è stata la scelta migliore e più rapida.

Per gli iper-parametri, sarebbe l'ideale operare il loro tuning tramite cross validation (k-fold), già implementata dalla libreria. Ma ho deciso di partire con quelli di default di sklearn, che funzionano bene su questo dataset:

- **Parametro di Regolarizzazione:** Valori di inversione della forza di regolarizzazione. Questo valore definisce la rigidità di un modello. Più il valore è basso (rigido), più ci si allontana dal problema dell'overfitting, avvicinandosi però all'underfitting. Più il valore è alto(flessibile), più il nostro modello tendera ad adattarsi troppo ai dati di addestramento, dunque causando l'overfitting. Di default è preso il valore 1.
- **Kernel e gamma:** Definiscono il tipo di kernel utilizzato nell'algoritmo. Quello di default è l'RBF(Radial Basis Function), conosciuto anche come kernel Gaussiano, con valore di gamma (che controlla l'influenza di ciascun elemento di training) pari a "scale", il quale ci dice che più la distanza euclidea tra due punti è grande, più la funzione kernel si avvicinerà a 0. Questo indicherebbe che due punti molto lontani hanno più probabilità di appartenere a classi diverse.

- **decision_function_shape:** essendo SVM un classificatore principalmente binario, è stato necessario sviluppare delle strategie per lavorare su multiple classi. Le strategie supportate dalla libreria di default sono: **OvO**(One-vs-One), dove viene addestrato un classificatore separato per ogni coppia di classi (per un totale di N*(N-1)/2 classificatori, con N numero di classi); **OvR**(One-vs-Rest), dove viene addestrato un classificatore per ogni classe, dove la classe principale di quel classificatore si troverà contrastata dal resto delle classi combinate in un'unica classe. Il classificatore con il valore più alto della funzione di decisione detterà la classe di appartenenza di un determinato elemento. La scelta adottata è molto semplice e si basa su due ragionamenti. In primis, per implementazione della libreria **sklearn,** la OvR è basata su OvO(dunque rendendo quest'ultima una scelta più leggera e meno ridondante). Ma in maniera più importante, considerando quel che abbiamo detto sul numero di classificatori per ogni strategia, possiamo giungere alla conclusione che avendo solo 10 classi come nel nostro caso, la OvO ci porterà al training di 6 classificatori contro i 10 della OvR, e questo ci fa preferire la OvO in quanto più conveniente.

Per la fase di training e test, essendo la SVM non l'argomento principale di interesse di questo progetto, ho deciso di prendere i risultati di una singola run del modello. Questa performance non deve essere considerata troppo "reale", in quanto non abbiamo addestrato questo modello più di una volta, e non abbiamo utilizzato esempi di training e test diverse, non utilizzando la K-Fold Cross Validation, anche se messa a disposizione della libreria.

Per completezza, facendo girare i questi comandi nella Python Shell di Pycharm:
“””
import mnist_data
training_data, validation_data, test_data = mnist_data.load_data()
import svm_prova
machine = svm_prova.SvmProva()
machine.svm_di_prova(training_data, test_data)
“””

```
>>>   import mnist_data
...     training_data, validation_data, test_data = mnist_data.load_data()
...     import svm_prova
...     machine = svm_prova.SvmProva()
...     machine.svm_di_prova(training_data, test_data)
```

Otterremo:

```
>>> machine.svm_di_prova(training_data, test_data)
Inizio Addestramento!
Attendere...
Addestramento Completato!
Inizio Fase di Test!
Attendere...
Fase di Test Completata!
9785 of 10000 valori corretti!.
```

Una accuracy del 97.85%, molto alta, ma per quanto detto poco prima, non troppo affidabile/veritiera.

Capitolo 4) Reti Neurali (ANN)

Capitolo 4.1) Idea alla Base

Prima di tutto, c'è bisogno di fare una premessa. Bisogna partire da un'idea: potremmo proviamo a risolvere il problema del riconoscimento delle cifre scritte a mano tramite una esplicitazione delle feature e dei pattern da riconoscere, dunque rifacendoci molto al metodo utilizzato da noi essere umani per il riconoscimento dei pattern: un 1 è rappresentato da una linea verticale, un 9 è rappresentato da una linea verticale e un cerchio nella parte alta e così via. Ma programmare un modello con delle regole così precise complicherebbe di molto l'implementazione, proprio perché ci sarebbero molte eccezioni e casi limite, considerando il fatto che ogni persona scrive in maniera completamente diversa(come accennato nel primo capitolo).

Però le Reti Neurali lavorano in maniera diversa. Esse utilizzano i dati di training per creare in modo **automatico** delle regole di inferenza per riconoscere le cifre, ed infatti queste saranno poco comprensibili a noi umani e non saranno in nessun modo dettate da noi.

La Rete Neurale è composta da **Neuroni** (nodi contenenti un valore da 0 a 1, detto **Activation**) e **Layer** (i quali raggruppano i Neuroni).

Detto questo, il tipo di Rete Neurale che andremo a sviluppare è un **CNN**(Convoluted Neural Network), usata per il riconoscimento di immagini, video o audio. Inoltre, la nostra Rete Neurale sarà di tipo **Feedforward**, il che significa che ci sarà una sola direzione del flusso di informazioni fra i layer, ovvero l'informazione non potrà tornare indietro.

I layer possono essere di 3 tipi:

- **Input Layer**: Il primo layer, contenente tutti i Neuroni di input, ovvero Neuroni a cui viene assegnato un valore di input che non dipende dal sistema in nessun modo;
- **Output Layer**: L'ultimo layer (quello dove l'informazione andrà a confluire), contenente tutti i Neuroni di output. In base ai valori dei Neuroni in questo layer, avverrà la classificazione del dato. Il numero di Neuroni di output dipende dal numero delle classi.
- **Hidden Layer(s)**: Layer compresi fra quelli di input e output, e sono quelli che principalmente si occupano del riconoscimento dei pattern. Sia il loro numero, che il numero dei Neuroni ad essi collegati possono essere considerati degli iper-parametri. NB: gli hidden layer sono facoltativi, ma la loro presenza migliora abbastanza la precisione di classificazione del sistema.

Dunque, la Rete **Feedforward** porterà e processerà le informazioni partendo dal layer di input, per poi passare fra quelli hidden, e poi confluire in quelli di output. Inoltre, layer vicini(sequenziali) sono collegati fra loro: ogni neurone di un layer è collegato tramite archi a tutti i singoli neuroni del layer precedente (tranne nel caso del layer di input, che non ha layer precedenti), e ognuno dei neuroni è anche collegato a tutti i singoli neuroni del layer successivo (tranne nel caso del layer di output, che non ha layer successivi).

Capitolo 4.2) Applicazione della Rete al Progetto

L'idea sarebbe quella di utilizzare l'intensità del colore (bianco e nero) di ogni pixel come valori di input. E ogni pixel verrà rappresentato da un neurone. Un neurone di input avendo attivazione di valore 0 indicherebbe che il determinato pixel ad esso associato è nero, un neurone di input avente attivazione di valore 1 indicherebbe invece che il pixel associato è bianco.

Sapendo che le dimensioni di tutte le immagini sono 28 pixel x 28 pixel, è facile dedurre che il numero di pixel(e dunque di neuroni di input) è 784.

Per il layer di output, sappiamo che le possibili classi di cifre sono 10 (ovvero i numeri da 0 a 9), e questo ci permette di confermare che il numero di neuroni nel layer di output sarà 10. Se per i neuroni di input l'attivazione indicava l'intensità del colore, per i neuroni di ouput essa indica la probabilità con il quale, secondo la rete, l'immagine appartenga a quella specifica classe(ovvero, la probabilità che l'immagine corrisponda alla specifica cifra rappresentata da quel neurone).

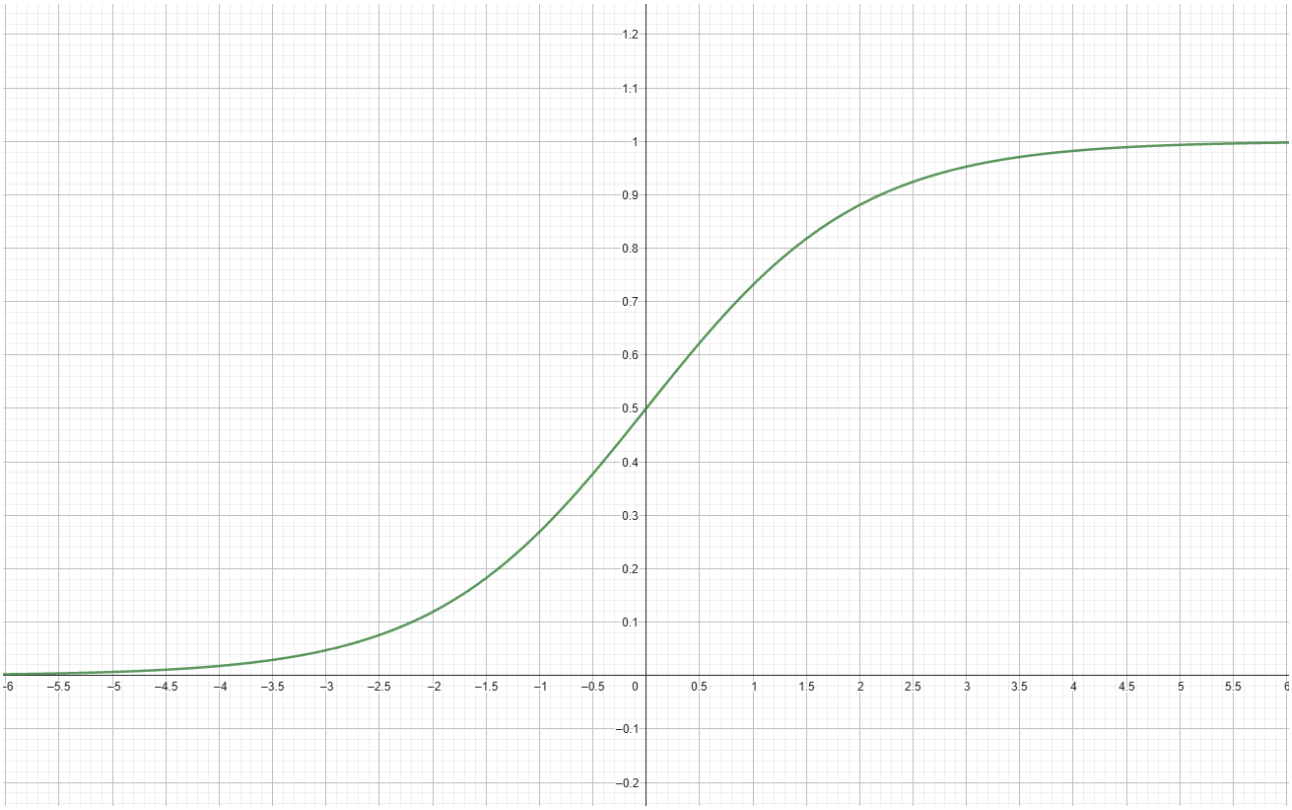
Gli hidden layer possono invece essere considerati come una black box: non si sa quel che accade precisamente ai valori di attivazione dei neuroni al loro interno, ma possiamo dedurre che per ogni hidden layer, verranno riconosciuti dalla rete specifici pattern con cui avverrà astrazione su quei dati ricevuti dal layer precedente. Quindi, ogni layer influenza il successivo: questo è rappresentabile tramite gli archi.

Ogni neurone, come detto in precedenza, è collegato a tutti i singoli neuroni del layer precedente tramite degli archi. Ad ognuno di questi archi, verrà assegnato un **peso** (**w**eight in inglese), il quale è solo un numero. Adesso, per determinare il valore del neurone, è possibile applicare una somma ponderata fra il valore di **a**ttivazione di ogni neurone, moltiplicato per il proprio peso.

w1*a1 + w2*a2 + w3*a3 + … + wn*an

Ma, facendo così, avremo un valore del neurone che molto probabilmente non sarà compreso fra 0 e 1. Per risolvere ciò, è possibile dare in pasto il risultato della somma ponderata alla funzione **Sigmoide:**

σ(x)=((1)/(1+*e*^(-x)))



Questa funzione schiaccerà il risultato della somma ponderata in un valore compreso tra 0 e 1: valori positivi della somma verranno schiacciati verso 1, valori negativi della somma verranno schiacciati verso 0, mentre valori della somma intorno a 0 vedrà la funzione sigmoide crescere notevolmente.

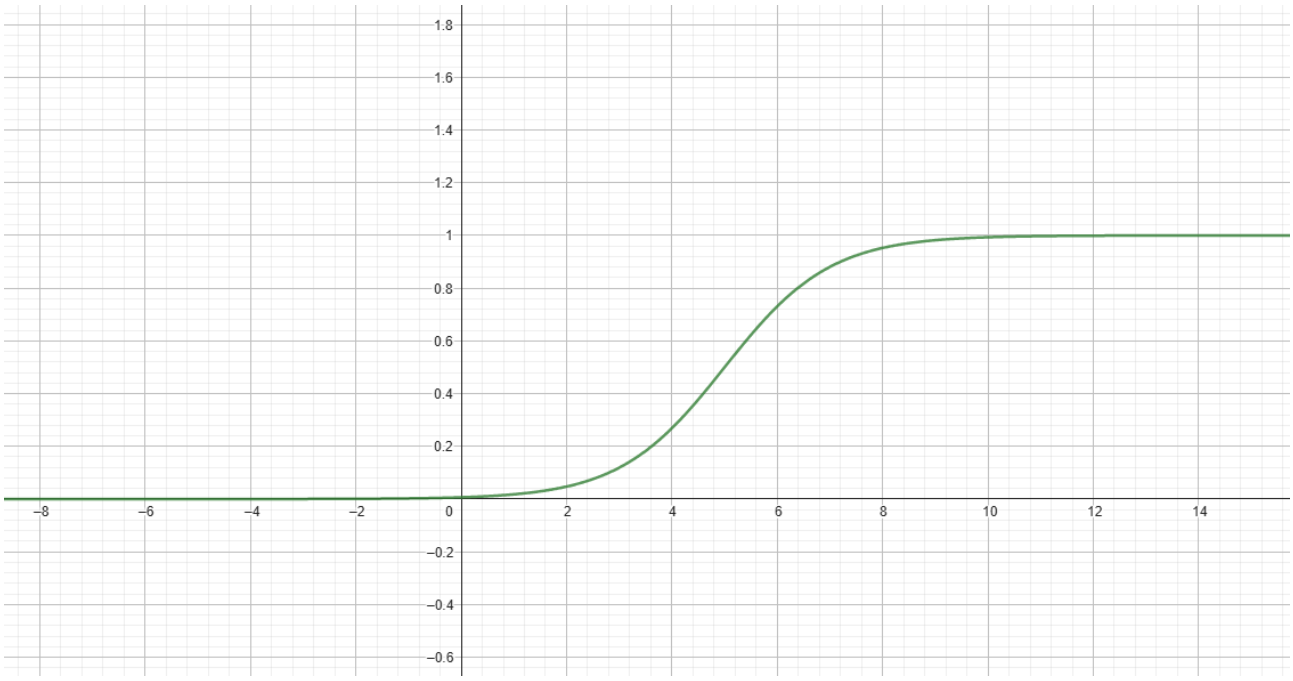
$$\sigma(w1*a1 + w2*a2 + w3*a3 + \ldots + wn*an)$$

Il valore di attivazione risultante del neurone non è altro dunque che un modo per definire quanto sia positiva(o negativa) la somma ponderata di tutti i neuroni del layer precedente.

Se volessimo però richiedere, ad esempio, che la somma non sia semplicemente positiva, ma molto positiva, è possibile introdurre dunque il concetto di **bias** che permette di spostare dunque la positività dell'input:

$$\sigma(w1*a1 + w2*a2 + w3*a3 + \ldots + wn*an + b)$$

Ad esempio, un bias di -5 vorrebbe significare che per essere significatamente positiva, la somma ponderata deve essere almeno superiore a 5. Questo è chiaro per analisi matematica: non è altro che una trasformazione della funzione sigmoide, precisamente una traslazione orizzontale:



E dunque, essendo sia i **pesi** che i **bias** dei parametri, possiamo dire che:

in una rete d'esempio con 4 layer, di cui quello di input avente 784 neuroni, il primo hidden layer avente 16 neuroni, il secondo hidden layer avente 12 neuroni e quello di output 10, è possibile calcolare il numero totale di parametri con cui è possibile modificare il comportamento dell'intera rete neurale.

La quantità di pesi è calcolata come la somma dei prodotti fra le quantità de neuroni del layer precedente e le quantità dei neuroni del layer successivo. Nel nostro esempio, avremo:

784x16 + 16x12 + 12x10 = 12.920 pesi

La quantità dei bias è calcolata come la quantità di tutti i neuroni non presenti nel layer di input. Nel nostro esempio, avremo:

16+12+10 = 38 bias

Dunque, la quantità totale di parametri modificabili è la somma di tutti i pesi e di tutti i bias:

12.920 + 38 = 12.958 parametri

E questo ci fa capire come sia necessario assolutamente trovare un modo che ottimizzi questi parametri in autonomia, in quanto questo è un compito impensabile per un essere umano.

Ed è proprio questo quello che verrà effettuato nella fase di addestramento: la Rete Neurale modificherà **passo per passo** i valori dei suoi bias e dei suoi pesi per riconoscere con più precisione le cifre.

Inoltre, per rappresentare il calcolo del valore di attivazione di un neurone, è possibile rappresentare la formula attraverso dei vettori e matrici:

$$a(i+1) = \sigma(Wa(i) + b)$$

Dove:

- **a(i+1)** è il vettore contenente i valori di attivazione del layer che vogliamo calcolare;
- **a(i)** è il vettore contenente i valori di attivazione del layer precedente a quello che vogliamo calcolare;
- **W** è la matrice contenente i pesi associati ad a(i) (con il quale viene applicata un prodotto fra matrici);
- **b** è il vettore contenente i bias dei neuroni.

Una rete neurale può dunque essere vista come una funzione che prende in input un vettore di 784 elementi(nel nostro caso), e ne restituisce uno con 10 elementi. Il valore più alto all'interno dell'array di output determina quale cifra è stata riconosciuta.

Capitolo 4.3) Costo e Gradient Descent Stocastica

Come inizio, è possibile assegnare ad ogni peso e bias della nostra rete neurale un valore completamente casuale: questo ci porterà quasi con certezza ad avere dei primi risultati anch'essi totalmente casuali: sul set di addestramento potremmo rivelare fin da subito molte classificazioni da parte della rete neurale che sono errate. Inoltre, i valori delle attivazioni di ogni singolo neurone di output potrebbero anch'esse risultare casuali: i valori di neuroni diversi potrebbero risultare anche molto vicini fra loro, essendo molto poco affidabili, e dimostrando che la rete non sappia in alcun modo essere certa della risposta da essa data.

Una risposta assolutamente certa da parte della rete dovrebbe essere, idealmente, una dove un neurone abbia valore di attivazione 1, e tutti gli altri abbiano valore 0, il che significherebbe che la rete è al 100% certa che la sua classificazione è corretta secondo la sua conoscenza.

Quindi, bisognerebbe introdurre il concetto di **funzione costo C**.

Sommando fra loro i quadrati delle differenze fra i valori di attivazione che abbiamo ottenuto e quelli che vorremmo, riusciremo ad ottenere il valore di costo di quel dato esempio di addestramento.

Ex: prendendo per semplicità una rete che riconosce solo le cifre 1 2 3 4, ovvero con 4 neuroni nel layer di output, potremmo calcolare il costo C assegnato un dato rappresentante un '3' con valori di attivazione dei neuroni in ordine: [0.17, 0.54, 0.81, 0.72] tramite la funzione costo:

$$C = (0.17 - 0.00)^2 + (0.54 - 0.00)^2 + (0.81 - 1.00)^2 + (0.72 - 0.00)^2$$
$$C = 0.0289 + 0.2916 + 0.0361 + 0.5184$$
$$C = 0.875$$

Con 0.875 costo per quell'esempio.

Chiaramente, questa somma (costo C) è bassa più la rete neurale sarà certa del risultato. Al contrario, più questa è alta, più la rete neurale non sarà affatto decisa sul risultato e quindi avremmo dei valori molto confusi o addirittura sbagliati!

Dunque è possibile calcolare la bontà di una rete neurale facendo la media dei costi su tutti i dati di addestramento.

Inoltre la funzione costo può essere espressa come un funzione che prende in input ogni singolo peso e bias (nell'esempio del capitolo 4.2, 12.958 parametri) e restituisce un solo valore (il costo).

Cercare di migliorare la nostra rete neurale vorrebbe dire trovare il **minimo della funzione costo**. Per semplicità, prendiamo una funzione con un solo parametro di input. Ricercare il minimo vuol dire trovare il punto del grafico della funzione più in basso, detto anche **minimo globale**.

Per scegliere un modo semplice per effettuare il tuning del parametro, è possibile calcolare la derivata nel punto attuale. Se la derivata (e dunque il coefficiente angolare della retta tangente al grafico nel punto) è positiva, allora vuol dire che bisognerebbe spostarsi a sinistra del grafico per trovare un minimo, in quanto la funzione è crescente in quell'intorno. Al contrario, se la derivata è negativa, vuol dire che bisognerebbe spostarsi a destra del grafico per trovare il minimo, dato che la funzione è decrescente in quell'intorno.

Dettaglio molto importante è che molte funzioni non sono semplici, e potrebbero presentare numerosi **minimi locali**. Questo vuol dire che, partendo da valori casuali di input, si potranno magari raggiungere punti di minimo completamente diversi, minimi magari migliori(come quello **globale**) o peggiori.

Se generalizziamo ad una funzione con più parametri di input(anche molto numerosi, come nel nostro caso), è il gradiente ad indicare la direzione di più veloce ascesa, ovvero la direzione che permette alla funzione di crescere il più velocemente possibile. Dunque, il negativo del gradiente ci offre la direzione di più veloce discesa, ovvero la direzione che permette alla funzione di decrescere il più velocemente possibile (che è esattamente quel che serve a noi).

Questo step è detto **Greedy Descent** e non consiste in altro che calcolare il valore del gradiente della funzione costo, e compiere un passo nella direzione opposta (il negativo del gradiente), e ripetere ciò fino a trovare un minimo.

Quello che realmente significa nella nostra applicazione e nel nostro progetto, è che l'algoritmo, detto di **backpropagation** (retropropagazione) ci dirà quali modifiche a tutti i pesi e bias causeranno la diminuzione più rapida della funzione di costo.

Ed essendo il valore del costo una media tra tutti i costi di tutti i dati di addestramento, trovare il minimo valore del costo vorrebbe dire rendere la rete neurale più sicura e precisa.

La **Greedy Descent**, considera dunque la media delle modifiche ai valori ottenute dalla backpropagation per ogni singolo elemento del set di addestramento. Ma dato che questo richiederebbe un grande costo computazionale, si preferisce adottare la **SDG (Greedy Descent Stocastica)**, la quale permette, dopo una suddivisione casuale dei dati in **mini-batch**, di aggiornare i valori per migliorare la rete neurale molto più velocemente e più spesso, diminuendo il costo man mano che si va avanti con i mini-batch.

Capitolo 5) Apprendimento Supervisionato: Parte 2

Capitolo 5.1) Implementazione Rete Neurale

Per l'implementazione, verrà utilizzata la libreria **numpy** la quale non ci prepara la rete neurale automaticamente(come nel caso della SVM del capitolo 3), ma ci offre degli strumenti per operare con gli array e le matrici, le quali ci permetteranno di lavorare più facilmente sulle reti neurali (per quanto accennato nel capitolo 4). È stata implementata la **SGD** e la **backpropagation**.

Per gli iper-parametri, abbiamo:

- **Numero di Hidden Layers e Numero di Neuroni per Hidden Layer:** Valori molto importanti e di difficile tuning. Più un sistema o task è complesso, più conviene avere una rete profonda. La profondità di una rete neurale non è altro che il numero di hidden layers. È da questo che ha origine il "deep" di deep learning;
- **Epochs (Epoche) :** Indica il numero di volte in cui viene dato in pasto il training set alla rete neurale per l'apprendimento. È da notare come valori troppo alti di Epochs tende ad aumentare i fenomeni di overfitting, e a questo è possibile arrivarci anche intuitivamente: la rete neurale inizierebbe a memorizzare i dati di addestramento piuttosto che riconoscerne i pattern (che è lo scopo principale del nostro modello!);
- **Mini_Batch_Size:** indica il numero di dati di addestramento casuali all'interno di un mini_batch, prima dell'aggiornamento dei valori del Gradient Descent Stocastico;
- **Learning Rate:** indica l'intensità dello step nella direzione decisa dal gradiente negativo. Valori troppo bassi ci porterebbero ad un apprendimento molto lento. Valori troppo alti ci porterebbero ad una oscillazione troppo forte fra le epoche, rendendo la rete neurale poco affidabile, oltre che poco performante.

Capitolo 5.2) Risultati e Conclusione

Per la fase di training e test, ho deciso di prendere i risultati di qualche run del modello, provando diversi iper-parametri manualmente. I motivi principali riguardano alcuni problemi (molto probabilmente delle ultime versioni dell'IDE open-source) con alcune librerie(tra cui **matplotlib** per i grafici e analisi dei dati oltre le semplici considerazioni che farò in seguito) o la mancanza di tempo, specialmente per portare il progetto su altre versioni. Le prove saranno più per verificare il funzionamento del ragionamento adottato che per analisi precise sui dati, le quali possono essere sviluppate in futuro.

Le varie prove:

1) Esempio Capitolo 4: 2 Hidden Layer, il primo da 16 neuroni, il secondo da 12, 30 epoche, 10 dimensione mini-batch, 1.0 learning rate.
   Facendo girare i comandi nella Python Shell di Pycharm:
   “””
   import mnist_data
   training_data, validation_data, test_data =
   mnist_data.load_data_wrapper()
   import reteneurale
   net = reteneurale.ReteNeurale([784, 16, 12, 10])
   net.SGD(training_data, 30, 10, 1.0, test_data)
   “””

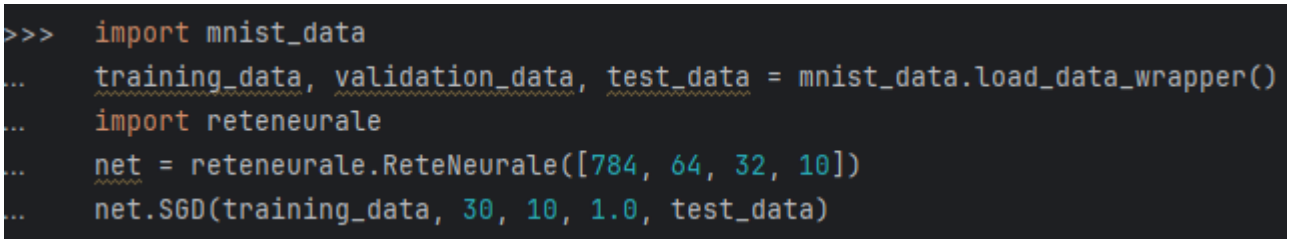
```
>>>   import mnist_data
...     training_data, validation_data, test_data = mnist_data.load_data_wrapper()
...     import reteneurale
...     net = reteneurale.ReteNeurale([784, 16, 12, 10])
...     net.SGD(training_data, 30, 10, 1.0, test_data)
```

Otterremo:

```
Epoca 0: 8602 / 10000, tempo impiegato: 3.77 secondi
Epoca 1: 8936 / 10000, tempo impiegato: 3.78 secondi
Epoca 2: 9035 / 10000, tempo impiegato: 3.80 secondi
Epoca 3: 9082 / 10000, tempo impiegato: 3.77 secondi
Epoca 4: 9107 / 10000, tempo impiegato: 3.78 secondi
Epoca 5: 9105 / 10000, tempo impiegato: 3.76 secondi
Epoca 6: 9116 / 10000, tempo impiegato: 3.77 secondi
Epoca 7: 9126 / 10000, tempo impiegato: 3.77 secondi
Epoca 8: 9159 / 10000, tempo impiegato: 3.76 secondi
Epoca 9: 9141 / 10000, tempo impiegato: 3.76 secondi
Epoca 10: 9174 / 10000, tempo impiegato: 3.76 secondi
Epoca 11: 9162 / 10000, tempo impiegato: 3.76 secondi
Epoca 12: 9200 / 10000, tempo impiegato: 3.81 secondi
Epoca 13: 9231 / 10000, tempo impiegato: 3.82 secondi
Epoca 14: 9219 / 10000, tempo impiegato: 3.79 secondi
Epoca 15: 9254 / 10000, tempo impiegato: 3.76 secondi
Epoca 16: 9232 / 10000, tempo impiegato: 3.78 secondi
Epoca 17: 9233 / 10000, tempo impiegato: 3.77 secondi
Epoca 18: 9268 / 10000, tempo impiegato: 3.78 secondi
Epoca 19: 9272 / 10000, tempo impiegato: 3.78 secondi
Epoca 20: 9218 / 10000, tempo impiegato: 3.76 secondi
Epoca 21: 9243 / 10000, tempo impiegato: 3.78 secondi
Epoca 22: 9257 / 10000, tempo impiegato: 3.80 secondi
Epoca 23: 9223 / 10000, tempo impiegato: 3.77 secondi
Epoca 24: 9264 / 10000, tempo impiegato: 3.76 secondi
Epoca 25: 9244 / 10000, tempo impiegato: 3.77 secondi
Epoca 26: 9267 / 10000, tempo impiegato: 3.78 secondi
Epoca 27: 9292 / 10000, tempo impiegato: 3.79 secondi
Epoca 28: 9292 / 10000, tempo impiegato: 3.77 secondi
Epoca 29: 9290 / 10000, tempo impiegato: 3.76 secondi
```

Accuracy massima raggiunta in epoca 27-28, 92.92%

2) Esempio precedente, ma con numero di neuroni negli hidden layer aumentato (a 64 e 32 rispettivamente).
   Facendo girare i comandi nella Python Shell di Pycharm:
   “””
   import mnist_data
   training_data, validation_data, test_data =
   mnist_data.load_data_wrapper()
   import reteneurale
   net = reteneurale.ReteNeurale([784, 64, 32, 10])
   net.SGD(training_data, 30, 10, 1.0, test_data)
   “””



Otterremo:

```
Epoca 0: 8905 / 10000, tempo impiegato: 7.37 secondi
Epoca 1: 9162 / 10000, tempo impiegato: 7.25 secondi
Epoca 2: 9233 / 10000, tempo impiegato: 7.23 secondi
Epoca 3: 9311 / 10000, tempo impiegato: 7.31 secondi
Epoca 4: 9373 / 10000, tempo impiegato: 7.38 secondi
Epoca 5: 9370 / 10000, tempo impiegato: 7.25 secondi
Epoca 6: 9417 / 10000, tempo impiegato: 7.23 secondi
Epoca 7: 9396 / 10000, tempo impiegato: 7.24 secondi
Epoca 8: 9437 / 10000, tempo impiegato: 7.25 secondi
Epoca 9: 9443 / 10000, tempo impiegato: 7.23 secondi
Epoca 10: 9464 / 10000, tempo impiegato: 7.28 secondi
Epoca 11: 9477 / 10000, tempo impiegato: 7.35 secondi
Epoca 12: 9463 / 10000, tempo impiegato: 7.34 secondi
Epoca 13: 9494 / 10000, tempo impiegato: 7.27 secondi
Epoca 14: 9485 / 10000, tempo impiegato: 7.27 secondi
Epoca 15: 9491 / 10000, tempo impiegato: 7.24 secondi
Epoca 16: 9488 / 10000, tempo impiegato: 7.26 secondi
Epoca 17: 9507 / 10000, tempo impiegato: 7.23 secondi
Epoca 18: 9513 / 10000, tempo impiegato: 7.22 secondi
Epoca 19: 9513 / 10000, tempo impiegato: 7.23 secondi
Epoca 20: 9486 / 10000, tempo impiegato: 7.22 secondi
Epoca 21: 9515 / 10000, tempo impiegato: 7.21 secondi
Epoca 22: 9508 / 10000, tempo impiegato: 7.23 secondi
Epoca 23: 9516 / 10000, tempo impiegato: 7.22 secondi
Epoca 24: 9499 / 10000, tempo impiegato: 7.24 secondi
Epoca 25: 9494 / 10000, tempo impiegato: 7.29 secondi
Epoca 26: 9518 / 10000, tempo impiegato: 7.33 secondi
Epoca 27: 9507 / 10000, tempo impiegato: 7.24 secondi
Epoca 28: 9508 / 10000, tempo impiegato: 7.22 secondi
Epoca 29: 9523 / 10000, tempo impiegato: 7.23 secondi
```

Accuracy massima raggiunta in epoca 29, 95.23%

Aumentare i neuroni degli hidden layer c'ha permesso di aumentare l'accuracy di qualche punto percentuale, con un raddoppio del tempo impiegato.

3) Esempio precedente, ma con un layer in più, un layer extra da 16 neuroni, 20 epoche invece di 30, ma un learning rate di 3.0. Mi aspetto un miglioramento delle prestazioni.

```
“””
import mnist_data
training_data, validation_data, test_data =
mnist_data.load_data_wrapper()
import reteneurale
net = reteneurale.ReteNeurale([784, 64, 32, 16, 10])
net.SGD(training_data, 20, 10, 3.0, test_data)
“””
```

```
>>>  import mnist_data
...  training_data, validation_data, test_data = mnist_data.load_data_wrapper()
...  import reteneurale
...  net = reteneurale.ReteNeurale([784, 64, 32, 16, 10])
...  net.SGD(training_data, 20, 10, 3.0, test_data)
```

Otterremo:

```
Epoca 0: 9053 / 10000, tempo impiegato: 6.94 secondi
Epoca 1: 9195 / 10000, tempo impiegato: 6.92 secondi
Epoca 2: 9318 / 10000, tempo impiegato: 6.95 secondi
Epoca 3: 9385 / 10000, tempo impiegato: 8.48 secondi
Epoca 4: 9438 / 10000, tempo impiegato: 8.41 secondi
Epoca 5: 9439 / 10000, tempo impiegato: 8.33 secondi
Epoca 6: 9406 / 10000, tempo impiegato: 8.31 secondi
Epoca 7: 9450 / 10000, tempo impiegato: 8.31 secondi
Epoca 8: 9458 / 10000, tempo impiegato: 8.54 secondi
Epoca 9: 9512 / 10000, tempo impiegato: 8.26 secondi
Epoca 10: 9527 / 10000, tempo impiegato: 8.67 secondi
Epoca 11: 9507 / 10000, tempo impiegato: 8.38 secondi
Epoca 12: 9479 / 10000, tempo impiegato: 8.37 secondi
Epoca 13: 9483 / 10000, tempo impiegato: 8.34 secondi
Epoca 14: 9499 / 10000, tempo impiegato: 8.32 secondi
Epoca 15: 9559 / 10000, tempo impiegato: 8.67 secondi
Epoca 16: 9571 / 10000, tempo impiegato: 8.41 secondi
Epoca 17: 9532 / 10000, tempo impiegato: 8.59 secondi
Epoca 18: 9564 / 10000, tempo impiegato: 8.33 secondi
Epoca 19: 9542 / 10000, tempo impiegato: 8.36 secondi
```

Accuracy massima raggiunta in epoca 16, 95.71%

Possiamo dedurre che con una valutazione e tuning dei parametri più approfondito, è possibile sicuramente superare l'accuracy raggiunta dalla SVM del capitolo 3.

Sviluppi Futuri:

La rete neurale funziona ed è stata implementata correttamente. Come detto in precedenza però, non è stato possibile per problemi di IDE, librerie e mancanza di tempo, poter fare valutazioni e tuning abbastanza preciso sugli iper-parametri. Sarebbe possibile dunque migliorare di molto le performance della rete neurale tramite cross validation o scelta ottimizzata in generale degli iper-parametri.

Altra miglioria da tenere in considerazione riguarda l'inizializzazione dei parametri (pesi e bias). Si potrebbe trovare un modo per non inizializzarli in maniera casuale, ma farli partire da qualche valore sensato, per raggiungere un minimo della funzione costo più facilmente.

Riferimenti Bibliografici

[1] D.Poole, A. Mackworth: Artificial Intelligence: Foundations of Computational Agents. 3rd ed. Cambridge University Press.

[2] I. Goodfellow, Y. Bengio, A. Courville: Deep Learning. MIT Press

[3] S. Shalev-Shwartz, S. Ben-David: Understanding Machine Learning: From Theory to Algorithms. Cambridge University Press.

[4] Scikit-Learn documentation