Università degli Studi di Padova
Dipartimento di Scienze Statistiche

Corso di Laurea in Statistica per le Tecnologie e le Scienze
Sistemi di Elaborazione 2
Aa. 2023-2024

RETI NEURALI A RETROPROPAGAZIONE: implementazione e applicazione in campo meteorologico

Giovanni Sequani
Matricola n° 2033244

Abstract

Uno degli algoritmi di machine learning più discussi degli ultimi anni è certamente quello delle reti neurali artificiali. I continui miglioramenti, favoriti dalla sempre più grande capacità di calcolo dei moderni computer, hanno portato le reti neurali ad essere considerate ad oggi lo stato dell'arte per il mondo dell'intelligenza artificiale. Questa relazione si propone di illustrare la struttura base e il funzionamento dal punto di vista matematico di una rete neurale che sfrutta la retropropagazione dell'errore come algoritmo di apprendimento. Tale rete verrà implementata in Python e verrà proposto un approfondito esempio di utilizzo in campo meteorologico. In particolare l'obiettivo che ci si porrà sarà la previsione della presenza di precipitazioni partendo da un dataset di comuni dati metereologici tramite l'utilizzo della rete neurale implementata.

Indice

1. TEORIA DELLE RETI NEURALI ARTIFICIALI

CONCETTI BASE

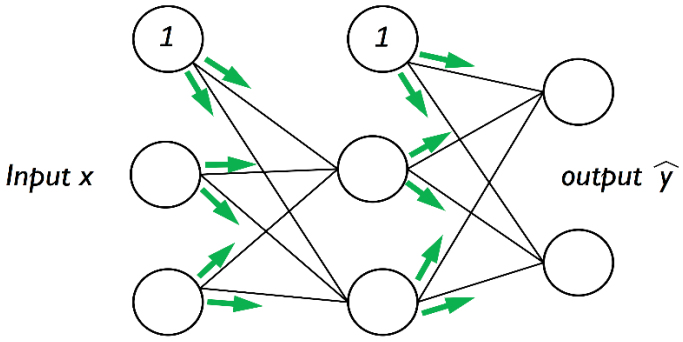
Le reti neurali artificiali sono una categoria di algoritmi di apprendimento automatico (*Machine learning*) che hanno riscosso grande successo e popolarità nell'ultimo decennio, nonostante i primi studi su di esse risalgono agli anni Quaranta (*A logical calculus of the ideas immanent in nervous activity*, McCulloch e Pitts, 1943). L'idea su cui le reti neurali artificiali si basano è quella di riprodurre in maniera semplificata il funzionamento dei neuroni del cervello umano attraverso algoritmi e operazioni matematiche. Un neurone biologico può essere visto come una porta logica che, tramite i dendriti, riceve dei segnali in input ed emette in output, attraverso i terminali dell'assone, una serie di output, i quali faranno da input ad altri neuroni.

A partire da questo concetto, numerosi studi hanno portato alla realizzazione di reti neurali artificiali sempre più complesse, in grado di svolgere compiti di classificazione, regressione, clustering, generazione di dati (testo, immagini, suoni), raccomandazione, ecc. Tuttavia in questa relazione mi concentrerò su reti neurali pienamente connesse (ogni neurone è connesso ad ogni neurone del layer precedente e successivo), dette anche *perceptron a layer multipli*, per la classificazione supervisionata; in particolare descriverò il funzionamento dell'algoritmo di apprendimento a retropropagazione basato sulla discesa stocastica del gradiente.

I passi per l'addestramento consistono nella generazione dell'output a partire dai *mini-batch* (dei sottoinsiemi di osservazioni) estratti dal dataset di addestramento, questa fase è chiamata propagazione in avanti. Successivamente l'algoritmo di retropropagazione, sulla base dell'output, calcola gli errori dei layer e aggiorna i pesi minimizzando il gradiente.

ALGORITMO DI PROPAGAZIONE IN AVANTI

La propagazione in avanti, come anticipato in precedenza, opera su mini-batch di osservazioni che io per semplicità considererò contenenti una sola osservazione, di conseguenza l'algoritmo verrà eseguito per ogni riga del dataset di addestramento. Esso consiste nella propagazione degli input di rete (le variabili esplicative dell'osservazione) ai successivi layer, fino a quello di output. Questa 'propagazione' si tratta della semplice combinazione delle variabili per i pesi, i quali sono uguali per ogni unità (o neurone) del layer:



Ad esempio, la prima unità del secondo layer viene calcolata nel seguente modo:

$$a_1^{(2)} = \phi\left(z_1^{(2)}\right) = \phi\left(w_{0,1}^{(1)} + a_1^{(1)} w_{1,1}^{(1)} + a_2^{(1)} w_{2,1}^{(1)}\right)$$

Dove $\phi()$ è la funzione di attivazione, che per questo tipo di rete utilizzerò la funzione sigmoidea:

$$\phi(z) = \frac{1}{1 + e^{-z}}$$

$w_j^{(l)} = \left(w_{0,j}^{(l)},\ w_{1,j}^{(l)},\ \dots,\ w_{m,j}^{(l)}\right)^T$ è il vettore di pesi relativo alla j-esima unità del layer l e $a_i^{(l)}$ è l'unità i del layer l.

L'algoritmo calcola così ogni layer, a partire dal precedente, fino al layer di output, ovvero l'output della rete.

ALGORITMO DI RETROPROPAGAZIONE

L'obiettivo dell'aggiornamento dei pesi è quello di minimizzare la funzione di errore, che in questo caso sarà l'errore quadratico:

$$J(W) = \frac{1}{2}\sum_{i}^{t}(y_i - \hat{y}_i)^2$$

Dove t è il numero di classi della variabile *target* e $\hat{y}_i$ è il valore predetto dalla rete di y_i cioè $a_i^{(L)}$ che dipende dai parametri *W*.

L'addendo $\frac{\lambda}{2}\sum_{l=1}^{L-1}\sum_{i=1}^{u_l}\sum_{j=1}^{u_{l+1}}(w_{j,i}^{(l)})^2$ è il termine di regolarizzazione L2, costituito dalla somma dei quadrati di tutti i pesi.

Per minimizzare questa funzione utilizzo la discesa stocastica del gradiente: iniziamo calcolando il gradiente della funzione di costo rispetto ai pesi del layer di output:

$$\Delta^{(L)} = \frac{\partial J(W)}{\partial w^{(L)}} = \frac{\partial J(W)}{\partial a^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}\frac{\partial z^{(L)}}{\partial w^{(L)}}$$

Dove:

$$\frac{\partial J(W)}{\partial a^{(L)}} = \frac{\partial}{\partial w^{(L)}}\left[\frac{1}{2}\sum_i \left(y_i - a_i^{(L)}\right)^2\right] = -\left(y - a^{(L)}\right)$$

$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \phi'\left(z^{(L)}\right)$$

$$\frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

Quindi il gradiente del layer di output è:

$$\Delta^{(L)} = -\left(y - a^{(L)}\right)\phi'\left(z^{(L)}\right)a^{(L-1)} = \delta^{(L)}a^{(L-1)}$$

dove $\delta^{(L)} = \frac{\partial J(W)}{\partial a^{(L)}}\frac{\partial a^{(L)}}{\partial z^{(L)}}$.

Per i layer nascosti vi è un problema: non conosciamo l'output ideale del layer, come invece si ha per il layer di output. L'algoritmo di retropropagazione permette però la propagazione dell'errore da destra verso sinistra, cioè dal layer di output fino a quello di input. In particolare si ha:

$$\Delta^{(l)} = \frac{\partial J(W)}{\partial w^{(l)}} = \frac{\partial J(W)}{\partial a^{(l+1)}}\frac{\partial a^{(l+1)}}{\partial z^{(l+1)}}\frac{\partial z^{(l+1)}}{\partial a^{(l)}}\frac{\partial a^{(l)}}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial w^{(l)}}$$

Dove:

$$\frac{\partial J(W)}{\partial a^{(l+1)}}\frac{\partial a^{(l+1)}}{\partial z^{(l+1)}} = \delta^{(l+1)}$$

$$\frac{\partial z^{(l+1)}}{\partial a^{(l)}} = w^{(l+1)}$$

$$\frac{\partial a^{(l)}}{\partial z^{(l)}} = \phi'\left(z^{(l)}\right)$$

$$\frac{\partial z^{(l)}}{\partial w^{(l)}} = a^{(l-1)}$$

e si propaga ai layer precedenti moltiplicandolo per i pesi relativi:

$$\Delta^{(l)} = \delta^{(l+1)} w^{(l+1)} \phi'\left(z^{(l)}\right) a^{(l-1)}$$

Allora si ha che il gradiente di un layer nascosto *l* qualsiasi è:

$$\Delta^{(l)} = \delta^{(l)} a^{(l-1)}$$

Dove $\delta^{(l)} = \delta^{(l+1)} w^{(l+1)} \phi'\left(z^{(l)}\right)$

e $\phi'\left(z^{(l)}\right)$ è la derivata della funzione di attivazione, nel nostro caso la funzione sigmoidea, e può essere calcolata semplicemente:

$$\begin{aligned}
\frac{\partial \phi(z)}{\partial z} &= \frac{\partial}{\partial z}\left(\frac{1}{1 + e^{-z}}\right) \\
&= \frac{e^{-z}}{(1 + e^{-z})^2} \\
&= \frac{1 + e^{-z}}{(1 + e^{-z})^2} - \left(\frac{1}{1 + e^{-z}}\right)^2 \\
&= \frac{1}{1 + e^{-z}} - \left(\frac{1}{1 + e^{-z}}\right)^2 \\
&= \phi(z) - (\phi(z))^2 \\
&= \phi(z)(1 - \phi(z)) \\
&= a(1 - a)
\end{aligned}$$

Il gradiente deve poi essere regolarizzato nel seguente modo:

$$\Delta^{(l)} = \Delta^{(l)} + \lambda \cdot W^{(l)}$$

dove λ è il parametro che regola la regolarizzazione L2.

Ora che ho ottenuto il gradiente regolarizzato della funzione di costo che voglio minimizzare, applico il metodo della discesa stocastica del gradiente e aggiorno i pesi compiendo un passo nella direzione opposta del gradiente:

$$W^{(l)} = W^{(l)} - \eta \Delta^{(l)}$$

dove η è il tasso di apprendimento.

Queste procedure che ho illustrato, ripetute per ogni mini-batch del dataset di addestramento, ottimizzano i pesi in modo da minimizzare la funzione di costo e quindi migliorano le capacità predittive della rete. Inoltre costituiscono un'*epoca* di addestramento della rete. Tramite un apposito parametro si può regolare il numero di epoche che l'addestramento verrà ripetuto, a vantaggio di migliori prestazioni della rete ma con il pericolo di cadere nel fenomeno dell'*overfitting*. Nel prossimo capitolo vedremo come implementare questi passaggi con Python in modo da creare una rete neurale artificiale.

2. IMPLEMENTAZIONE DI UNA RETE NEURALE ARTIFICIALE

STRUTTURA DI BASE

L'implementazione della rete neurale si baserà sul linguaggio di programmazione Python, questa scelta non è la migliore dal punto di vista della velocità di esecuzione (linguaggi come C o Fortran hanno migliori prestazioni) tuttavia è giustificata dalla semplicità e dall'interpretabilità, che hanno reso Python uno dei linguaggi di programmazione più diffusi. Farò uso di librerie, come *Numpy* e *Pandas*, scritte parzialmente in C o in CPython ed ottimizzate per il calcolo vettoriale e matriciale, fondamentale per gli algoritmi che ho illustrato. Inoltre l'uso di un linguaggio semplice come Python permette di focalizzarmi maggiormente sull'algoritmica piuttosto che su complicati aspetti di programmazione.

Il costrutto su cui baserò la rete neurale è la Classe, alla quale 'passeremo' gli iperparametri che definiranno le specifiche caratteristiche della rete. Definiamo gli iperparametri come quelle costanti numeriche e non, definite nel momento della creazione della rete, che non variano durante l'addestramento del modello. Nel nostro caso gli iperparametri sono il numero di layer nascosti e il loro numero di unità, il tasso di apprendimento, il parametro di regolarizzazione L2 e il numero di epoche. Si distinguono dai parametri per il fatto che questi ultimi sono variabili che vengono aggiornate durante il processo di addestramento. Nel nostro caso un esempio di parametri sono i pesi di ciascun layer.

A questo punto possiamo iniziale l'implementazione definendo innanzitutto la classe *NeuralNetwork* e il relativo metodo `.__init__()`:

```
class NeuralNetwork(ClassifierMixin, BaseEstimator):


    def __init__(self,
                 n_hidden_units: list | None = [10],
                 eta: float | None = 0.1,
                 l2: float | None = 0.01,
                 epochs: int | None = 100,
                 seed: int | None = 0) -> None:

        self.n_hidden_units = n_hidden_units
        self.eta = eta
        self.l2 = l2
        self.epochs = epochs
        self.seed = seed
```

I parametri del metodo `.__init__()` sono:

- `n_hidden_units`: è una lista di interi, i quali rappresentano il numero di unità per ciascun layer nascosto. Se ad esempio si passa la lista `[20, 30, 10]` allora la rete sarà composta da tre layer nascosti composti da 20, 30 e 10 unità.
- `eta`: è il tasso di apprendimento della rete.
- `l2`: è il parametro che regola l'intensità di regolarizzazione L2.
- `epochs`: è il numero di volte che viene ripetuto l'algoritmo di apprendimento e quindi l'iterazione delle osservazioni nel dataset di addestramento.
- `seed`: è il seme per la generazione di numeri casuali da assegnare inizialmente ai pesi. Semi uguali inizializzano i pesi con valori uguali ad ogni esecuzione dello script.

Per implementare i metodi che servono per addestrare la rete e da essa fare previsioni seguiranno i formalismi utilizzati dai modelli offerti dalla libreria *scikit-learn* (formalismi) un modulo di Python per il machine learning basato su *SciPy* (https://scikit-learn.org). Nello specifico, i modelli di classificazione *scikit-learn* (o *sklearn*)

possiedono un metodo `.fit()` per l'addestramento, un metodo `.predict()` per la previsione ed ereditano metodi dalle classi genitrici *ClassifierMixin* e *BaseEstimator*.

Le librerie che ci serviranno sono: *numpy*, *sys* e *time*.

IL METODO .FIT()

In questo paragrafo illustrerò l'implementazione del metodo `.fit()` che permette di addestrare la rete neurale. Esso avrà come parametri di input due dataset di addestramento, contenenti le variabili esplicative o *features* e la variabile risposta o *target*, i quali dovranno essere oggetti della classe `numpy.ndarray`.

Inizio la costruzione del metodo `.fit()` dividendo i dataset di addestramento nei sottoinsiemi per l'aggiornamento dei pesi (train) e per la stima delle capacità predittive (valid). Successivamente salvo informazioni utili sui dataset in apposite variabili locali:

```
    def fit(self, X_train: np.ndarray, y_train: np.ndarray) -> None:

        ## split del dataset
        ind = int(np.floor(0.7 * X_train.shape[0]))
        X_valid, X_train = X_train[ind:], X_train[:ind]
        y_valid, y_train = y_train[ind:], y_train[:ind]

        ## usefull info
         start = time.time()
        n_examples = X_train.shape[0] # number of obs.
        n_features = X_train.shape[1] # number of features
        n_output = np.unique(y_train).shape[0] # number of levels of the target variable
```

Nelle seguenti righe di codice inizializzo casualmente le matrici dei pesi di ciascun layer, salvandole in liste. Inoltre costruisco la matrice di variabili dummy relative alla variabile target:

```
        ## weights initialization
        random = np.random.RandomState(self.seed)
        self.weights = []
        self.intercepts = []
        dimensions = [n_features]+self.n_hidden_units+[n_output]
        for i in range(len(dimensions)-1):
            self.weights.append(random.normal(loc=0,scale=0.1,
                                              size=(dimensions[i],dimensions[i+1])))
            self.intercepts.append(np.zeros(dimensions[i+1]))

        ## encoding of y: returns matrix of dummy [n_examples]x[n_output]
        y_enc = np.zeros(shape=(n_examples, n_output))
        for ind, val in enumerate(y_train.astype(int)):
            y_enc[ind,val] = 1

        ## dict for accuracy
        self.acc = {'train': [], 'valid': []}
```

Tramite un primo ciclo for ripeto la procedura di apprendimento `self.epochs` volte e tramite un secondo for la ripeto per ogni mini-batch di addestramento costituito da una sola osservazione. L'apprendimento dei pesi viene eseguito richiamiamo i metodi interni per la propagazione in avanti, per la retropropagazione e per il calcolo dei gradienti dei pesi. Successivamente calcolo il termine di regolarizzazione L2 per ogni peso (i termini di intercetta solitamente non vengono regolarizzati) e infine aggiorno i pesi ad ogni iterazione. Inoltre calcolo ad ogni epoca le performance della rete in termini di accuratezza sulla base del sottoinsieme di validazione e la stampo come output tramite la libreria *sys*.

```
        # repeat [self.epochs] times
        for j in range(self.epochs):

            # repeat [n_examples] times
            for idx in np.arange(n_examples):

                X = X_train[[idx]] # select obs.

                a_hidden, z = self._forward_propagation(X)
                deltas = self._back_propagation(a_hidden, y_enc[idx])
                gradienti_weights, gradienti_intercepts = self._gradients(X, deltas, a_hidden)

                ## L2 regolarization
                delta_weights = []
                delta_intercepts = []
```

```
            for i in range(len(self.weights)):
                delta_weights.append(gradienti_weights[i] + self.l2 * self.weights[i])
                delta_intercepts.append(gradienti_intercepts[i]) # intercetta non
                                                                   regolarizzata

            ## weights update
            for i in range(len(self.weights)):
                self.weights[i] -= self.eta * delta_weights[i]
                self.intercepts[i] -= self.eta * delta_intercepts[i]


        ## final forward propagation
        a_hidden, z = self._forward_propagation(X)

        ## evaluate accuracy
        y_train_pred = self.predict(X_train)
        y_valid_pred = self.predict(X_valid)

        train_accuracy = (np.sum(y_train == y_train_pred)).astype(np.float32) / n_examples
        valid_accuracy = (np.sum(y_valid == y_valid_pred)).astype(np.float32) / len(y_valid)

        self.acc['train'].append(train_accuracy)
        self.acc['valid'].append(valid_accuracy)

        sys.stderr.write('\repoch: %d/%d | valid accuracy: %.2f' %
                         (j+1, self.epochs, valid_accuracy*100))
        sys.stderr.flush()

    s = time.time() - start
    print(f"\ntraining time: {int(s/60)} min {int(s%60)} s")

    return self
```

PREVISIONE

Rifacendomi ai formalismi di *scikit-learn*, per la previsione delle etichette di nuovi dati, implemento il metodo `.predict()`, il quale riceve in input un array *numpy* contenente le variabili esplicative e restituisce in output le etichette predette. La previsione di nuove etichette consiste nella propagazione in avanti dei valori di input (le variabili esplicative) e nello scegliere come classe predetta quella con valore maggiore fra i valori in uscita della rete. Riporto di seguito l'implementazione in Python:

```
def predict(self, X: np.ndarray) -> np.ndarray:

    a_hidden, z = self._forward_propagation(X)

    return np.argmax(z, axis=1)
```

3. PREVISIONE IN CAMPO METEOROLOGICO

DATI E PREPROCESSING

In questo capitolo mostrerò un esempio di utilizzo della rete neurale artificiale descritta e implementata nei precedenti capitoli. Il problema che voglio affrontare è quello della previsione se in una determinata fascia oraria vi saranno o meno precipitazioni. Vogliamo quindi addestrare una rete capace di classificare una fascia oraria come "precipitazioni" o "no precipitazioni" e per farlo è fondamentale innanzitutto costruire un dataset in grado di fornire all'algoritmo l'informazione meteorologica necessaria.

Reperirò i dati meteorologici tramite l'API di *WorldWeatherOnline* che fornisce per Python la libreria *wwo_hist*. È necessario disporre di una API key per poter scaricare dati, è possibile ottenerla in prova gratuita

di 30 giorni o sottoscrivendo un abbonamento mensile. Ho già reperito una API key valida fino al 29 giugno 2024. Per comodità opererò su un dataset già da me scaricato e salvato attraverso il seguente codice:

```
from wwo_hist_updated import retrieve_hist_data

frequency = 1
start_date = '1/1/2010'
end_date = '27/9/2023'
api_key = '61d88074833245ecbb9135659243004'
location_list = ['padova']
hist_weather_data = retrieve_hist_data(api_key,
                                location_list,
                                start_date,
                                end_date,
                                frequency,
                                location_label = False,
                                export_csv = False,
                                store_df = True)

hist_weather_data[0].to_csv("padova.csv")
```

Questo codice è contenuto nel file *retrieve_data.py*, nella cartella *datasets*. Si noti che in essa è presente anche una copia della libreria *wwo_hist* modificata e rinominata *wwo_hist_updated*, nella quale ho corretto alcuni metodi e parametri *pandas* deprecati. Non è quindi necessario scaricare la libreria *wwo_hist*.

Ho così ottenuto un dataset contente numerose variabili con frequenza oraria fra le quali estrarrò quelle più significative per la previsione di *y*. Inoltre uniamo le variabili "sunset" e "sinrise" in "sunmin", ovvero i minuti trascorsi fra l'alba e il tramonto.

Se ora passassi alla rete neurale questo dataset le variabili esplicative porterebbero informazioni solamente riguardo un'ora specifica, l'ultima disponibile; tuttavia è ragionevole pensare che possano essere utili agli scopi predittivi anche le informazioni di più fasce orarie passate. A tal scopo, aggiungo come variabili esplicative anche i valori di ciascuna variabile con un certo *lag*, ovvero i valori di ogni variabile ritardata.

Inoltre la *y* da prevedere sarà la variabile binaria "precipBin":

$$y = \begin{cases} 1 & se\ 'precipMM' > 0 \\ 0 & se\ 'precipMM' = 0 \end{cases}$$

È importante sottolineare che la variabile binaria y va anche traslata o 'shiftata' di un certo numero di fasce orarie per permettere la previsione futura. Se infatti non applicassimo alcuna traslazione la *y* si riferirebbe all'orario degli ultimi dati disponibili, tuttavia a noi non interessa prevedere se ci sono state precipitazioni nel passato (non è necessario nessun algoritmo di previsione) ma ci interessa farlo nel futuro. Se ad esempio ipotizziamo ora siano le 12:21 e volessimo prevedere se fra le 13:00 e le 14:00 ci saranno precipitazioni allora disporremmo dei dati fino alla fascia oraria 11:00-12:00 ma la variabile target *y* sarebbe quella della fascia 13:00-14:00. Quindi *y* va traslata 'all'indietro' di 2 righe del dataset. Regolando il parametro `SHIFT` si può aumentare la traslazione, quindi andando a prevedere fasce orarie "più future".

Infine elimino le righe contenenti valori nan e salvo il dataset in un file .csv. Questi passaggi di creazione della matrice di dati sono riportati nel file *preprocessing.py*.

Il dataset è ora pronto per essere diviso in *X*, insieme delle variabili esplicative o *features*, e *y*, variabile risposta o *target*. Uso inoltre la funzione *train_test_split()* del modulo *sklearn.model_selection* per suddividere il dataset in un sottoinsieme di addestramento (60%) e un dataset di test (40%):

```
y = data["precipBin"]
X = data.drop(columns=["date_time","precipBin"])
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, shuffle=False)
```

Prima di passare i dati di addestramento alla rete neurale è importate standardizzare le features, in modo da portarle sulla stessa scala, e per fare ciò utilizzo la classe *StandardScaler()* del modulo *sklearn.preprocessing*.

```
std = StandardScaler()
X_train_std = std.fit_transform(X_train)
X_test_std = std.transform(X_test)
```

Ora è tutto pronto per addestrare la rete neurale implementata:

```
rn = NeuralNetwork(n_hidden_units=[200, 100, 50], eta=0.001, l2=0.0001, epochs=30)
rn.fit(X_train_std, y_train)
```

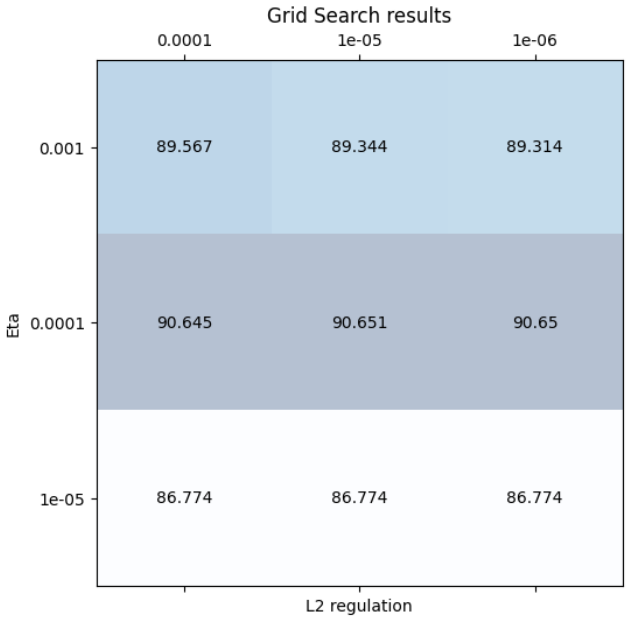
In questo primo esempio la rete sarà costituita da un layer di input, tre layer nascosti e un layer di output. I layer nascosti saranno costituiti rispettivamente da 200, 100 e 50 unità, come specificato dall'iperparametro `n_hidden_units`. Nel prossimo paragrafo illustrerò come ottimizzare gli iperparametri della rete, ovvero selezionare i valori ottimali che massimizzano l'accuratezza di previsione.

OTTIMIZZAZIONE IPERPARAMETRI

La scelta dei valori ottimali degli iperparametri è un processo molto oneroso dal punto di vista computazionale, infatti comporta l'addestramento della rete, già di per sé costoso, numerose volte, tante quante le combinazioni di valori degli iperparametri. *Scikit-learn* fornisce diversi metodi di ottimizzazione, come ad esempio il metodo della ricerca casuale o la ricerca a griglia; io userò quest'ultimo combinato al metodo della convalida incrociata utilizzando la classe *GridSearchCV* del modulo *sklearn.model_selection*.

Questa procedura è molto costosa e a seconda del numero di combinazioni di parametri da valutare può impiegare molto tempo, talvolta molte ore, perciò riporto di seguito il codice e i risultati ottenuti:

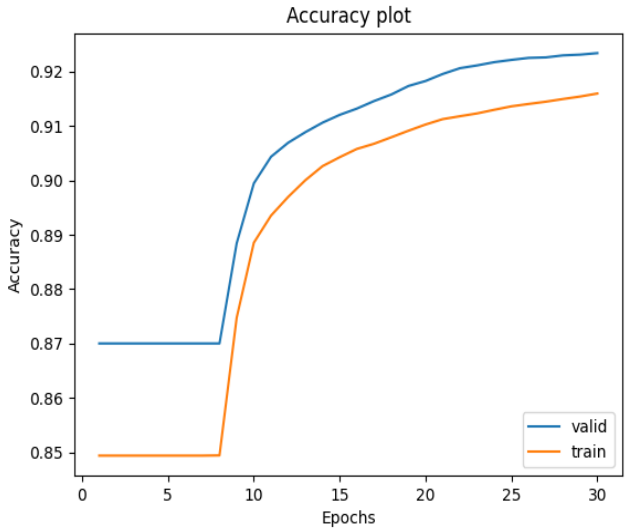
```
from sklearn.model_selection import GridSearchCV
import pickle as pkl

param_grid = {"n_hidden_units" : [[200,100,50]],
              "eta" : [0.001, 0.0001, 0.00001],
              "l2" : [0.0001, 0.00001, 0.000001],
              "epochs" : [30] }

gs = GridSearchCV(estimator=NeuralNetwork(),
                  param_grid=param_grid,
                  scoring=scorer, n_jobs=-1, cv=3, verbose=3)

gs.fit(X_train_std, y_train)
```

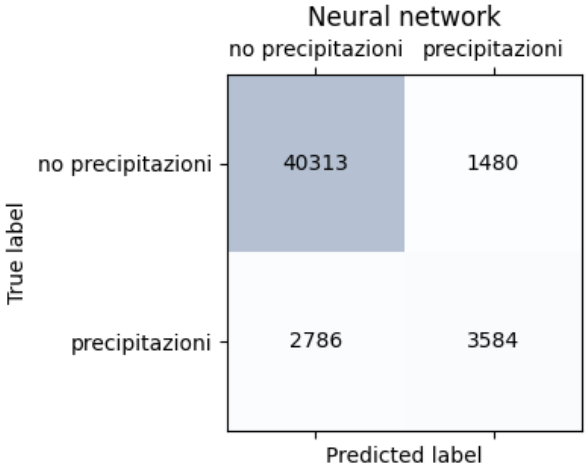


Otteniamo che il miglior modello ha come parametri:

```
{'epochs': 30, 'eta': 0.0001, 'l2': 1e-05, 'n_hidden_units': [200, 100, 50]}
```

e ha le seguenti prestazioni:





MODELLO FINALE

Il modello finale è quindi una rete neurale con tre layer nascosti con 200, 100 e 50 unità, il tasso di apprendimento è pari a 0.0001, il parametro di regolarizzazione L2 è 0.00001 e 30 epoche di addestramento. Le prestazioni su dati di test che il modello non "ha mai visto" sono buone con un'accuratezza che si attesta intorno al 91.1% e una specificità pari a 96.4%; tuttavia la sensibilità è bassa (56.3%), ciò significa che quasi la metà delle volte che vi sono precipitazioni, il modello prevede in modo errato.

4. CONCLUSIONI

LIMITI DEL MODELLO E PROBLEMI INCONTRATI

Come accennato nel capitolo 1, ogni passo di apprendimento viene eseguito su un mini-batch di k esempi di addestramento con $k = 1$, quindi l'apprendimento e il calcolo dei pesi aggiornati viene fatto per ogni esempio del dataset di training. Questo approccio permette di convergere più velocemente, tramite discesa del gradiente, al minimo della funzione di costo, d'altra parte aumenta sensibilmente la complessità computazionale del processo di addestramento. Questo, infatti, è stato uno dei principali problemi incontrati: l'eccessiva complessità computativa dell'algoritmo di addestramento rende difficoltoso per i comuni computer il fitting della rete. Di conseguenza, risulta più complicata anche la fase di ottimizzazione degli iperparametri, penalizzando perciò le prestazioni predittive del modello finale. Ho quindi modificato il metodo `.fit()` permettendo mini-batch composti da più osservazioni, ciò velocizza l'addestramento ma complica l'ottimizzazione degli iperparametri, infatti per la maggior parte delle combinazioni di essi la rete prevedeva sempre 'no precipitazioni' per ogni esempio del dataset.

Infine un altro problema incontrato è quello della bassa sensibilità del modello finale: essa è dovuta al fatto che il ristretto numero di esempi di addestramento etichettati come 'precipitazioni' non permette al modello di apprendere al meglio le caratteristiche di questi esempi, facendo così "sfuggire" molte osservazioni dalla classificazione 'no precipitazioni'.

La rete neurale da me descritta, cioè un percettrone a layer multipli, è una delle più semplici e dalla più facile interpretazione rispetto alle reti più utilizzate negli ultimi anni, come ad esempio le reti neurali convoluzionali profonde, le reti generative avversarie e le reti neurali ricorrenti. Queste, nella maggior parte dei casi, sono in grado di ottenere risultati predittivi migliori; nonostante ciò le metriche di performance per l'esempio del capitolo 3 indicano un buon adattamento. Si può quindi concludere che la rete neurale pienamente connessa implementata ha ottime prestazioni per scopi di classificazione, come quello presentato; tuttavia per problemi più complessi, come l'analisi di testo e immagini, i quali trattano migliaia di features, è preferibile adottare reti più elaborate.