Università degli Studi dell'Aquila
Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

---

Corso di Machine Learning

Homework 3: Report

Angelo Damiani

Anno Accademico 2017/2018

Indice

Capitolo 1

Struttura del software

1.1 Linguaggio, librerie e repository utilizzate

Per la realizzazione di questo homework è stato utilizzato come linguaggio di programmazione il Python con le seguenti librerie aggiuntive:

- Tensorflow: una libreria open source originariamente sviluppata dai ricercatori di Google per la ricerca sul deep learning. Fornisce delle API per diversi linguaggi di programmazione (Python, C++, Java, ...). È stata utilizzata per la compatibilità con la libreria Keras;
- Keras: è una high level API scritta in Python per diverse piattaforme (TensorFlow, CNTK e Theano). Originariamente è stata sviluppata per testare e sperimentare reti neurali nel più breve tempo possibile;
- Numpy: è una libreria matematica principalmente utilizzata per calcoli vettoriali e matriciali. Come per tensorflow, anche questa libreria è stata utilizzata per la compatibilità con Keras.

Il dataset utilizzato per il testing e la sperimentazione è la repository UCI Letter mentre per la stesura del manuale è stato utilizzato il dataset UCI Iris.

1.2 Organizzazione del codice

Per la realizzazione di questo homework sono state implementate 4 classi software:

- `MLP`: wrapper di Keras che si occupa di istanziare un multilayer perceptron, salvarlo, caricarlo da file, allenarlo e interrogarlo.

- `Encoder`: modulo che si occupa di generare la discretizzazione a partire da un dataset, salvarla, caricarla da un file, discretizzare un singolo campione, dediscretizzare un singolo vettore discretizzato, convertire un vettore discretizzato in un vettore composto da vettori caratteristici e viceversa. In più questo modulo si occupa anche di convertire un campione costituito da vettori caratteristici in un campione "pseudo-originale" in cui i vettori caratteristici corrispondenti alle stringhe (quindi valori simbolici) sono convertiti nella loro discretizzazione mentre quelli corrispondenti ai valori numerici sono convertiti nel loro formato originario nel dataset; ciò è stato fatto per permettere una metrica significativa per il calcolo della distanza di un risultato di una query con il vettore che ci si aspetta come risultato.

- `DatasetProcessor`: modulo che si occupa di lettura e salvataggio di un dataset, discretizzazione del dataset e trasformazioni in dataset con vettori caratteristici (utilizzando la classe `Encoder`). Anche per questa classe è stata definita una conversione del dataset in una versione "pseudo-originale".

- `DataAnalyzer`: modulo che si occupa, generalmente, dell'estrazione di statistiche (basilari) tra due dataset (uno contenente le predizioni e uno contenente ciò che ci si aspetta dalla predizione).

Infine c'è un file di main che si occupa di orchestrare queste classi.

Capitolo 2

Flusso implementativo

Il main, come scritto nel manuale, è composto da tre funzioni per l'utente:

- `queryMain`: funzione per l'interrogazione, mediante file, della rete neurale;
- `testMain`: funzione per il testing di una rete neurale pre-esistente;
- `trainMain`: funzione per il primo allenamento di una rete neurale;

In questo capitolo si tratterà quest'ultimo al fine di segnalare le principali scelte implementative. Nel seguito si daranno per effettuate le fasi di inserimento degli input come il nome della rete neurale (utilizzato per salvare la rete su file) o l'inserimento del path del dataset.

2.1 Lettura del dataset e preprocessing

Trovandoci nella situazione in cui si allena la rete (per una data repository) per la prima volta, non si conosce alcuna discretizzazione. Allo stato attuale si istanzia un oggetto della classe `Encoder` e si istanzia un oggetto della classe `DatasetProcessor` passandogli l'encoder (appena definito) e il path della repository che si vuole utilizzare.

Fatto ciò, il dataset deve essere letto e se ne occupa il dataset processor con la funzione `read()`. Tale funzione legge riga per riga il dataset salvandolo in una lista. Una volta che il dataset è nella lista, quest'ultima viene mischiata per evitare regolarità all'interno della distribuzione dei campioni (un esempio per cui è necessario

questo approccio sta nella repository MNIST per il riconoscimento dei caratteri che presenta i campioni in ordine alfabetico, prima tutte le "a", poi tutte le "b", ...).

Dopo aver letto il dataset "grezzo", questo viene dato all'encoder; il quale genererà la discretizzazione necessaria tramite la funzione `generateDiscretization()`

La discretizzazione avviene, all'interno dell'encoder, con l'ausilio di due attributi. Uno di questi (`typefeaturelist`) mantiene per ogni feature il tipo di quest'ultima (stringa, se simbolica, o float, se numerica); l'altra (`discretizationList`) mantiene la vera discretizzazione. `discretizationList` è una lista ordinata di liste; la i-esima di queste liste contiene la discretizzazione della i-esima feature. La discretizzazione della singola feature dipende dal tipo di questa; se il tipo è **stringa**, allora si tratta di una feature simbolica, già discreta, e dunque la lista conterrà tutti i possibili valori diversi assumibili da quella feature (e se possibile ordinarli, lo saranno). Se il tipo è **float**, allora andrà effettuata la discretizzazione; per effettuarla si cercano i valori minimi e massimi assumibili dalla singola feature e, una volta trovati, si costituisce una lista (poi aggiunta a `discretizationList`) contenente tutti i numeri dal minimo al massimo con passo pari a $\frac{massimo-minimo}{massimo}$. Quindi, l'encoder leggerà il dataset grezzo campione per campione e feature per feature; all'inizio assume che ogni feature sia numerica ma alla prima occorrenza di un valore non numerico cambierà la considerazione di questa feature da float a stringa.

Una volta generata la discretizzazione viene discretizzato il dataset. Ciò avviene discretizzando campione per campione, feature per feature. Si consideri di voler discretizzare l'i-esima feature del campione s; questa viene convertita selezionando l'indice j di `discretizationList[`i`]` tale che `discretizationList[`i`][`j`] - s[`i`]` sia minimo (ovviamente per le feature simboliche si avrà `discretizationList[`i`][`j`] - s[`i`] = 0`).

Una volta discretizzato il dataset si passa alla generazione del training set e del test set.

2.2 Selezione e costituzione del training set e test set

A questo punto va scelto il training set. Si supponga che sia stato già selezionato il numero di campioni, diciamo m, con il quale si vuole allenare la rete e le features obiettivo; ora si considera una porzione del dataset discretizzato (nell'implementazione è stato scelto il primo 90%) e si genera il training set. Il training è generato dal seguente algoritmo:

1: $trainingset \leftarrow$ lista vuota
2: $it \leftarrow 0$
3: **for** $it < m$ **do**
4: $sample \leftarrow$ campione casuale nel primo 90% del dataset discretizzato
5: $l \leftarrow$ numero casuale tra 1 e il numero di features obiettivo
6: scegli l features (tra le features obiettivo) da *sample* e sostituiscile con "?"
7: aggiungi *sample* a trainingset
8: $it \leftarrow it + 1$
9: **end for**
10: **return** *trainingset*

Il risultato di questo algoritmo è un training set discretizzato. Ciò che si desidera è un training set in cui ogni feature di ogni campione è rappresentato con un vettore caratteristico.

Ciò avviene sempre con l'ausilio dell'encoder. L'encoder, conoscendo il numero di possibili valori per ogni feature (ossia la lunghezza di ogni singola lista all'interno di `discretizationList`) conosce quanto deve essere lungo ogni campo del vettore caratteristico. Dunque il dataset processor chiederà all'encoder la conversione di ogni singolo campione. L'encoder, allora, convertirà il campione s feature per feature; si consideri l'i-esima feature, già discretizzata, l'encoder costituisce un vettore cv di lunghezza pari a `len(discretizationList[i])` (la lunghezza della discretizzazione per quella feature) inizializzato per avere tutti 0 tranne che per $cv[s[i]]$, dove avrà 1.

Ciò presupponendo che $s[i]$ sia in `discretizationList[i]`, se ciò non fosse (come per il caso del "?") il vettore avrà tutti 0.

Generato il training set va generato il test set. La procedura è la stessa con la differenza che viene scelto l'ultimo 10% del dataset discretizzato.

2.3 Rete neurale

Le unità per gli hidden layer della rete neurale sono user defined mentre quelli di input e output layer sono uguali e dipendenti dalla dimensione dei campioni formati dai vettori caratteristici.

Nel caso della repository UCI Letter si hanno 17 features (16 numeriche e 1 simbolica). Ciascuna delle features numeriche ammette valori tra 0 e 15, dunque la discretizzazione è una lista contenente i numeri da 0 a 15 con passo pari a $\frac{15-0}{15} = 1$. Per questo motivo, la lista sarà lunga 16 e così anche il singolo vettore caratteristico di ogni feature numerica. Per quanto riguarda la feature simbolica, essa può assumere i valori da 0 a 25 (dove 0 = "a" e 25 = "z"), ossia 26 valori diversi (e quindi un vettore caratteristico di stessa lunghezza).

Dato che le features numeriche sono 16, il campione codificato con i vettori caratteristici avrà lunghezza: $16 \cdot 16 + 26 = 282$.

Per quanto riguarda la funzione di attivazione, per ciascuna unità dell'input layer e degli hidden layer della rete neurale è stata scelta come funzione di attivazione una rectifier ($f(x) = max(0, x)$) mentre per l'output layer è stato scelto un sigmoide ($f(x) = \frac{1}{1+e^{-x}}$). Infine è stato scelto come metodo di ottimizzazione un metodo di stochastic gradient descent (nel particolare Adaptive Moment Estimation, ADAM).

A questo punto la rete è pronta per essere allenata, viene lasciata la scelta del numero di epoch all'utente.

Nelle sperimentazioni, nel capitolo successivo, si mostreranno le varie scelte nel particolare.

2.4 Postprocessing: dediscretizzazione e interpretazione dell'output

Dopo che la rete è stata allenata, questa viene testata con il test set definito in precedenza. Nonostante ci si aspetti come output una lista di campioni codificati con vettori caratteristici, quello che si ottiene (per ogni feature originale) è un vettore contenente numeri compresi tra 0 e 1 (poichè ogni unità dell'ultimo layer ha come funzione di attivazione un sigmoid). La prima cosa da effettuare, quindi, è il ricondursi a un campione codificato con vettori caratteristici. È stato scelto di convertire il valore più alto di ciascun vettore caratteristico "grezzo" con un 1 e ogni altro valore con uno 0. La ricostruzione della lunghezza del vettore caratteristico avviene con l'ausilio della lunghezza delle liste di discretizzazione all'interno dell'encoder.

Una volta effettuato ciò, si ha a disposizione una lista di risultati codificati con vettori caratteristici da ricondurre alla discretizzazione e poi al formato originale. Conoscendo le liste di discretizzazione ciò è realizzato dall'encoder.

Per ricondursi alla discretizzazione basta, per ogni vettore caratteristico (la cui lunghezza è conosciuta dall'encoder), trovare l'indice j in cui tale vettore ha un 1. Conoscendo l'indice j per la i-sima feature ci si può riportare al formato originale accedendo a `discretizationList[i][j]` che contiene il valore nel formato originale per quel dato indice.

L'interpretazione degli output viene delegata alla classe `DataAnalyzer`. Nell'implementazione del main è stato scelto di passare a questa classe il test set (convertito in un formato pseudo-originale in cui ogni valore numerico è convertito col suo valore nel formato originale mentre ogni valore simbolico è convertito nella sua discretizzazione) e il risultato del test (anch'esso convertito allo stesso modo). Il data analyzer, nel main, è utilizzato per calcolare la distanza di Minkowsky di ordine 1 (quindi la distanza 1) per ogni coppia <testsample,risultato> e salvarlo in una lista. Una volta che sono state analizzate tutte le coppie viene fatta una media degli errori, viene calcolato l'errore medio per singola feature e, per completezza, viene vista quanto è la percentuale di uguaglianza perfetta tra risultato e testsample.

Anche in questo caso, i risultati sperimentali verranno visti nel capitolo successivo.

Capitolo 3

Sperimentazione

Le sperimentazioni effettuate sono state realizzate allenando la rete su un sottoinsieme delle features (nel particolare si sono considerate le seguenti features del dataset "UCI Letter": 0 (la lettera), 1, 4, 8, 13, 16) per un malinteso sulla spiegazione; nonostante tale malinteso non condizionasse la fruizione del software (poichè bastava inserire tutte le feature come feature obiettivo durante la definizione del training) è stato ritagliato del tempo per l'ultima sperimentazione su tutte le variabili. Quest'ultima è l'ultima del capitolo.

3.1 Features di input come variabili simboliche

Nel corso dello sviluppo di questo homework sono state fatte diverse sperimentazioni, molte delle quali dovute a degli errori di sviluppo. Questo è il caso; all'inizio dello sviluppo dell'homework era stato sviluppato l'algoritmo considerando ogni feature come puramente simbolica. In questo contesto, la discretizzazione non ordinava neanche i simboli e infatti i risultati delle predizioni erano scadenti. L'architettura selezionata per questa sperimentazione presenta 6(+1) layer. Il "+1" tra parentesi indica l'input layer poichè la libreria Keras non definisce il layer d'ingresso (ciò vale per ogni altra sperimentazione e non sarà ripetuto). I layer hanno rispettivamente il seguente numero di unità: 300, 400, 500, 400, 300, 282. Come anticipato il risultato della predizione non era dei migliori. Ciò anche dovuto al fatto che i simboli delle feature non erano state ordinate; infatti quello che poteva essere un

errore unitario (invece di predire "7" magari predire "8") sulla singola feature veniva trasformato in un errore maggiore di 5 o 10. Dunque, la distanza 1 calcolata con il data analyzer riportava un errore medio di 49.3333. Purtroppo, essendo stato un primo esperimento in fase di sviluppo ho dimenticato di effettuare screenshot dei risultati.

3.2 Features numeriche ordinate e features simboliche senza ordinamento

Una volta accortomi dell'errore ho costituito una discretizzazione per i valori numerici. Inoltre, per avere risultati migliori, ho aggiunto complessità alla rete aumentando il numero di unità per layer: 400, 600, 700, 600, 400, 282. L'allenamento è stato effettuato con un training set di 420000 campioni e con 60 ripetizioni (epochs). Il risultato è dato dall'immagine 3.1. Si può notare che più di $\frac{1}{5}$ delle query risultano

```
Please insert the neural network name (it's only used to LOAD the neural network): UCI-letter
2017-11-28 22:15:03.527699: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
2017-11-28 22:15:03.527726: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
2017-11-28 22:15:03.527735: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
2017-11-28 22:15:03.527742: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
2017-11-28 22:15:03.527749: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow
Insert the dataset path for the test: files/testsets/UCI-letter/testset
Insert the number of test samples: 20000
On how many features do you want to test the neural network? 6
Insert the objective feature number 0: 0
Insert the objective feature number 1: 1
Insert the objective feature number 2: 4
Insert the objective feature number 3: 8
Insert the objective feature number 4: 13
Insert the objective feature number 5: 16
Test performed
Exact predictions = 4351
Exact predictions percentage = 21.755%
Mean distance = 3.83165
Mean error on single feature = 0.22539117647058823
```

Figura 3.1: Risultati dell'esperimento

perfettamente esatte a ciò che ci si aspettava (4351 su 20000). Per quanto riguarda la distanza media tra i vettori originali e i vettori predetti, risulta essere di 3.832 circa. Ciò significa che la somma delle differenze (in modulo) di ciascuna feature predetta, con l'originale, è inferiore a 4.

3.3 Features ordinate (simboliche e non)

Non accontentandomi del risultato, ho voluto sperimentare ordinando, quando possibile, anche le features simboliche. La rete mantiene la stessa struttura, a meno di retraining, e così anche l'allenamento (sia numero di campioni che numero di epochs). Sorprendentemente tale sperimentazione non ha portato a risultati rilevanti. La percentuale di predizioni perfette e la distanza media tra predizione e originale risultano essere molto simili all'esperimento precedente:

```
Please insert the neural network name (it's only used to LOAD the neural network): ordered-letter
2017-11-28 22:13:34.248394: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libra
2017-11-28 22:13:34.248425: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libra
2017-11-28 22:13:34.248434: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libra
2017-11-28 22:13:34.248441: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libra
2017-11-28 22:13:34.248448: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libra
Insert the dataset path for the test: files/testsets/ordered-letter/testset
Insert the number of test samples: 20000
On how many features do you want to test the neural network? 6
Insert the objective feature number 0: 0
Insert the objective feature number 1: 1
Insert the objective feature number 2: 4
Insert the objective feature number 3: 8
Insert the objective feature number 4: 13
Insert the objective feature number 5: 16
Test performed
Exact predictions = 4496
Exact predictions percentage = 22.48%
Mean distance = 3.89695
Mean error on single feature = 0.22923235294117647
```

Figura 3.2: Risultati dell'esperimento

3.4 Confronto con l'homework 2

Un altro esperimento che si è voluto effettuare riguarda il confronto dell'errore con l'homework 2. Si è utilizzata la rete allenata nell'esperimento precedente ma si è posta come unica variabile obiettivo del test la lettera da predire. L'interpretazione dell'output è stata fatta due volte; la prima valutando il perfect matching solo sulla variabile obiettivo (figura 3.3); la seconda valutando il perfect matching su tutto il vettore risultato 3.4.

I risultati mi hanno stupito per due diversi motivi.

Il primo è la grande accuratezza per quanto riguarda la singola lettera. Rispetto all'homework 2 la percentuale d'errore è diminuita sulla singola lettera portando ad

```
Please insert the neural network name (it's only used to LOAD the neural network): ordered-letter
2017-11-28 23:09:31.289415: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libı
2017-11-28 23:09:31.289435: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libı
2017-11-28 23:09:31.289440: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libı
2017-11-28 23:09:31.289443: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libı
2017-11-28 23:09:31.289447: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow libı
Insert the dataset path for the test: files/testsets/UCI-letter/testset
Insert the number of test samples: 20000
On how many features do you want to test the neural network? 1
Insert the objective feature number 0: 0
Test performed
Exact predictions = 19705
Exact predictions percentage = 98.525%
```

Figura 3.3: Risultati dell'esperimento (considerando il matching solo per la lettera)

```
Please insert the neural network name (it's only used to LOAD the neural network): ordered-letter
2017-11-28 23:11:20.951412: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
2017-11-28 23:11:20.951432: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
2017-11-28 23:11:20.951437: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
2017-11-28 23:11:20.951441: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
2017-11-28 23:11:20.951445: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow lib
Insert the dataset path for the test: files/testsets/ordered-letter/testset
Insert the number of test samples: 20000
On how many features do you want to test the neural network? 1
Insert the objective feature number 0: 0
Test performed
Exact predictions = 6038
Exact predictions percentage = 30.19%
Mean distance = 3.7176
Mean error on single feature = 0.21868235294117647
```

Figura 3.4: Risultati dell'esperimento (considerando il matching per tutto il campione)

un'accuratezza maggiore del 98%. Si tratta di un risultato inatteso poichè mi aspettavo, a causa dell'esplosione della dimensione dell'input da 16 a 282, una perdita di prestazione. Perdita di prestazione dovuta al fatto che la rete non è aumentata esponenzialmente come, invece, è accaduto all'input.

Il secondo è la grande differenza che c'è se si assume il matching perfetto su tutte le variabili nonostante quella obiettivo sia solo la prima. Si passa da un 98% a un 31% scarso. Ciò è dovuto alla "supponenza" della rete neurale nel considerare errate le feature di input nonostante non siano variabili obiettivo. A dispetto di ciò, la distanza tra predizione e originale non è grandissima e si assesta su un 3.7, indice che in media le variabili della predizione differiscono al massimo di 4 unità, circa, dall'originale.

3.5 Ultima sperimentazione: tutte le features sono obiettivo

Infine si è sperimentato utilizzando tutte le variabili come variabili obiettivo. Presupponendo una caduta nelle prestazioni è stata definita un'architettura della rete leggermente più complessa: 500, 600, 800, 600, 500, 282. La rete è stata allenata per 45 epochs (a causa dei lunghi tempi ho dovuto diminuire il numero di epochs) su un training set di 500000 campioni. Una volta allenata tutta la rete, è stato effettuato il testing e il risultato è rappresentato in figura 3.5.

```
Please insert the neural network name (it's only used to LOAD the neural network): all-feature-letters
2017-11-28 22:31:08.583473: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library \
2017-11-28 22:31:08.583493: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library \
2017-11-28 22:31:08.583498: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library \
2017-11-28 22:31:08.583502: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library \
2017-11-28 22:31:08.583505: W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library \
Insert the dataset path for the test: files/testsets/all-feature-letters/testset
Insert the number of test samples: 20000
On how many features do you want to test the neural network? 17
Insert the objective feature number 0: 0
Insert the objective feature number 1: 1
Insert the objective feature number 2: 2
Insert the objective feature number 3: 3
Insert the objective feature number 4: 4
Insert the objective feature number 5: 5
Insert the objective feature number 6: 6
Insert the objective feature number 7: 7
Insert the objective feature number 8: 8
Insert the objective feature number 9: 9
Insert the objective feature number 10: 10
Insert the objective feature number 11: 11
Insert the objective feature number 12: 12
Insert the objective feature number 13: 13
Insert the objective feature number 14: 14
Insert the objective feature number 15: 15
Insert the objective feature number 16: 16
Test performed
Exact predictions = 1909
Exact predictions percentage = 9.545%
Mean distance = 12.6471
Mean error on single feature = 0.7439470588235294
```

Figura 3.5: Risultati dell'esperimento

Come ci aspettava le prestazioni sono diminuite ma ancora buone. La percentuale di matching perfetto è diminuito al 9.5% e la distanza media tra predizione e campione originale è aumentata al 12. Nonostante ciò e nonostante un numero di campioni del training set esiguo rispetto alle possibili configurazioni delle combinazioni delle variabili obiettivo, tale distanza non è esageratamente grande se si considera che mediamente una feature differisce dalla rispettiva feature originale meno di 1 (0.744 circa).