Appunti su Reti Neurali Artificiali

Loris Cro

20 giugno 2013

Approccio dell'esposizione

In questo documento vengono brevemente presentate le reti neurali artificiali.

Prima di procedere con la parte principale viene proposto un esempio utile a concretizzare la comprensione delle parti successive. L'esempio é implementato in Python e il codice é disponibile online come notebook di IPython, consultabile a questo indirizzo. Le uniche dipendenze richieste sono `numpy` (calcolo matriciale) e `PIL` (manipolazione di immagini), entrambi installabili facilmente tramite pip:

```
$ sudo pip install numpy PIL
```

Qualora non si avesse pip installato, é procurabile dai repository, ad esempio:

```
$ sudo apt-get install python-pip
```

0 Introduzione

Per introdurre cosa sono le reti neurali artificiali vediamo una versione estremamente semplificata di un problema tipicamente risolto con questa tecnica e di cui forniamo una soluzione di poche righe.

Il problema in questione é quello del riconoscimento di caratteri scritti a mano. Il riconoscimento dei CAP alle poste é un esempio di situazione in cui il problema si presenta in maniera simile a quanto viene ora proposto.

Nel nostro caso si tratta di riconoscere solo tre tipi di caratteri: `1`, `2` e `3`.

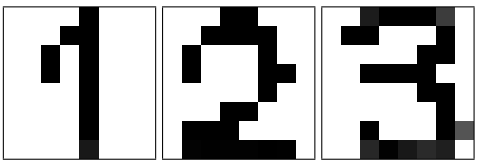

Figura 1: Tre esempi 8x8 di caratteri disegnati a mano.

Le immagini arrivano di dimensione 8x8 pixel su sfondo bianco, da cui noi ricaveremo una matrice a valori booleani (0 segna i pixel bianchi, 1 quelli pieni).

Supponiamo anche di avere una libreria di immagini di esempio classificate precedentemente, quindi di ogni esempio sappiamo giá qual' é la categoria giusta.

La struttura del metodo risolutivo é la seguente:

Il problema sostanzialmente consiste nel fatto che i caratteri sono versioni leggermente distorte o "sbavate" di quello che possiamo considerare come il carattere "ideale". I caratteri scritti a mano mantengono abbastanza caratteristiche da risultare riconoscibili, ma difficilmente si puó pensare ad una soluzione senza logiche di approssimazione.

In particolare potremmo assegnare dei pesi che indichino quanto sia rilevante lo stato dei pixel per riconoscere di che carattere si tratta. Ad esempio, una matrice per riconoscere gli `1` potrebbe avere dei pesi positivi a valori "alti" nella zona della colonna centrale, ad indicare che quando i pixel in quella zona sono pieni probabilmente si tratta di un `1`, e dei pesi negativi ai pixel nella parte destra dell'immagine ad indicare che quando quei pixel sono pieni probabilmente non si tratta di un `1`. Similmente si potrebbero costruire altre due matrici per riconoscere i `2` ed i `3`. A questo punto si potrebbe sommare i valori pesati dei pixel e decidere che se il valore ottenuto supera una certa soglia allora abbiamo riconosciuto il carattere.

Dobbiamo quindi trovare un modo automatizzato per generare le tre matrici dei pesi.

Iniziamo importando le librerie:

```
import random
import numpy as np
from PIL import Image
```

La prima operazione che ci serve é quella di caricare le immagini in memoria come matrici booleane. Quello che facciamo é semplicemente separare il colore bianco dal resto, supponendo che sia il colore di fondo.

```
def load_img(filename):
    img = Image.open(filename)
    return np.matrix([int(x != (255,255,255,0))
                          for x in img.getdata()]).transpose()
```

Un dettaglio: l'immagine viene caricata come una matrice formata da una sola colonna lunga 64 elementi. Questo é per semplificare poi i calcoli rendendoli come prodotti righe-colonne, idealmente conviene sempre pensarla come una matrice 8x8.

A questo punto possiamo caricare gli esempi in memoria. Su GitHub sono disponibili delle immagini adatte dentro la subdirectory `lettere/`. Siccome l'immagine verrá provata contemporaneamente su tutte le matrici di pesi, il risultato atteso é un vettore di tre bit di cui solo uno é alzato, ad indicare che quella é la categoria a cui appartiene l'immagine.

```
def risultato_atteso(indice):
    return np.array([int(x == indice) for x in range(0,3)])
    #esempio: [0, 0, 1] indica un 3

esempi = [{
            'immagine': load_img('lettere/%i%s.png' % (i, c)),
            'categoria': risultato_atteso(i-1)
          } for i in range(1,4) for c in 'abcd']
```

Ora vogliamo avere tre matrici di pesi per ognuna delle categorie. Per poter dopo risolvere il calcolo in un solo prodotto matriciale, le rappresentiamo come 3 righe da 64 elementi di una singola matrice.

```
pesi = np.zeros((3, 64))
```

Adesso entriamo nel succo del discorso:

Abbiamo le immagini di esempio, sappiamo come dovrebbero essere classificate ma ancora non abbiamo delle matrici dei pesi adatte. Ora vediamo come farle generare dal sistema. Il concetto di base é che facciamo provare a classificare un' immagine e, qualora la classificazione non sia corretta, le matrici che hanno sbagliato vengono alterate "indebolendo" i pesi dei pixel pieni nel caso di un falso positivo e "rafforzandoli" in caso di un falso negativo. Quanto si debba alterare il peso a fronte di un errore

é stabilito dal parametro `epsilon`.

```
epsilon = 0.2

runs = 0
errori = True

while errori:
    errori = False
    random.shuffle(esempi)
    for esempio in esempi:
        img, atteso = esempio['immagine'], esempio['categoria']
        prova = (pesi * img > 0).transpose()
        if not all(prova == atteso):
            errori = True

        delta = epsilon * img * (atteso - prova)
        pesi += delta.transpose()
        runs += 1
print 'Training completato in %i passi' % runs
```

Due righe meritano particolare attenzione:

```
prova = (pesi * img > 0).transpose()
```

Qui avviene il test dell'immagine sulle matrici di pesi. In particolare `pesi` é una matrice 3x64 e `img` é un vettore 64x1. Dopo il prodotto righe-colonne ogni elemento del vettore 3x1 risultante viene testato se maggiore di 0, che é la soglia da superare scelta per questo esempio. La trasposizione serve per far tornare dopo le operazioni matriciali.

```
delta = epsilon * img * (atteso - prova)
```

Qui vengono calcolate le variazioni da apportare alle matrici di pesi. Prima di tutto vengono calcolati gli errori di classificazione tramite la sottrazione tra `atteso` e `prova`. Per ogni matrice, il risultato della differenza puó essere -1 (falso positivo), 0 (classificazione corretta) o 1 (falso negativo); dopo di che viene stabilito se diminuire, lasciare invariati o aumentare, in base ai rispettivi casi, i pesi dei pixel pieni della quantitá stabilita da epsilon.

Possiamo anche provare a vedere come sono fatte le matrici dei pesi (richiede `pylab`):

```
import pylab as pl

for i in range(0, 3):
    print 'Classe %i:' % (i + 1)
    x = np.reshape(pesi[i], (8,8))
    x = np.flipud(x)
    pl.pcolor(array(x))
    pl.colorbar()
    pl.show()
```

I risultati differiscono tra run diverse in quanto l'ordine degli esempi con cui si allenano le matrici dei pesi é diverso per ogni esecuzione (la lista degli esempi viene mescolata ad ogni iterazione) e, appena una matrice riconosce correttamente gli esempi, questa non viene piú modificata, e, come si puó supporre, generalmente esistono piú matrici diverse in grado di classificare correttamente lo stesso tipo di caratteri (in particolare nel nostro caso le classi sono formate da solo 4 elementi). Un esempio che risulta abbastanza chiaro graficamente é il seguente:

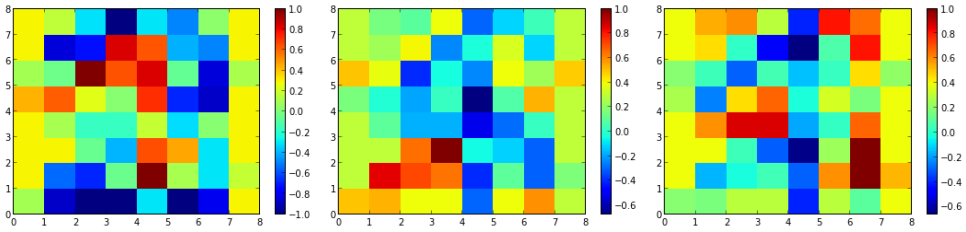

Figura 2: Una possibile configurazione delle tre matrici dei pesi.

A questo punto non rimane che generare una nuova immagine e vedere se siamo in grado di riconoscerla correttamente.

Se si vuole provare il brivido di ottenere risultati piú interessanti si suggerisce di trovare esempi a maggiore risoluzione e disponibili in maggiori quantitá, come ad esempio *Semeion Handwritten Digit Data Set*. Nel notebook di IPython é presente il codice per caricare anche questi formati e avendo una liberia di esempi abbastanza corposa (~1500 esempi) é anche possibile tenerne una parte "nascosta" (non usata per l' apprendimento) per valutare le prestazioni del sistema. I risultati sono sorprendentemente soddisfacenti, data la minimalitá della nostra implementazione, benché ovviamente molto lontani dai migliori risultati conosciuti.

1 Introduzione alle reti neurali artificiali

Le reti neurali artificiali sono modelli computazionali ispirati dalle controparti biologiche.

Il cervello umano conta circa 10^{11} neuroni. Ogni neurone riceve segnali elettro-chimici da altri neuroni a lui connessi e quando l'intensitá dei segnali in ingresso supera una certa soglia, il neurone si attiva e scarica a sua volta verso altri neuroni per un breve periodo.

Un singolo neurone non possiede complessi meccanismi computazionali, la forza espressiva deriva dalla struttura della rete, in particolare la capacitá di apprendimento di una rete neurale risiede nel poter variare il peso e la struttura delle connessioni tra neuroni. Nelle reti neurali artificiali solo la variazione dei pesi é permessa.

Le ANN (Artificial Neural Network) vengono organizzate in strati di neuroni che ricevono segnali dallo strato precedente e che a loro volta segnalano allo strato successivo:

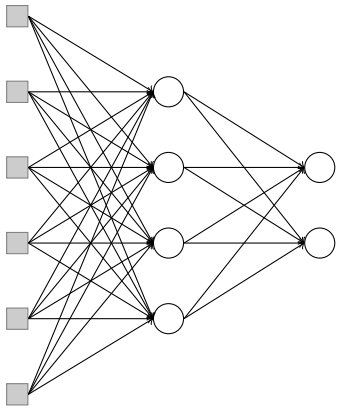

Figura 3: Esempio di struttura di una ANN.

Il primo strato viene chiamato "strato di input", quelli intermedi "strati nascosti" e quello finale "strato di output".

Entriamo ora nel dettaglio vedendo l'esempio piú semplice di ANN.

Percettrone

Un percettrone é il piú semplice. Quello che abbiamo costruito nell' esempio é effettivamente una rete di percettroni a singolo strato, con una sola differenza: nelle ANN (e quindi anche nei percettroni) anche la soglia ha un peso abbinato e quindi puó essere alterata a sua volta durante la fase

di apprendimento (come mostrato in Figura 5).

La struttura di una rete di percettroni a singolo strato é la seguente:

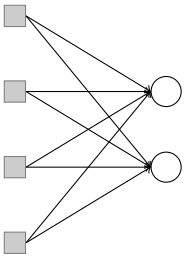

Figura 4: Esempio di struttura di una rete di percettroni a singolo strato (senza contare quello di input).

Un singolo percettrone al suo interno esegue due operazioni fondamentali: una somma pesata degli input e l'applicazione di una funzione di attivazione (anche detta *normalizzatore*).

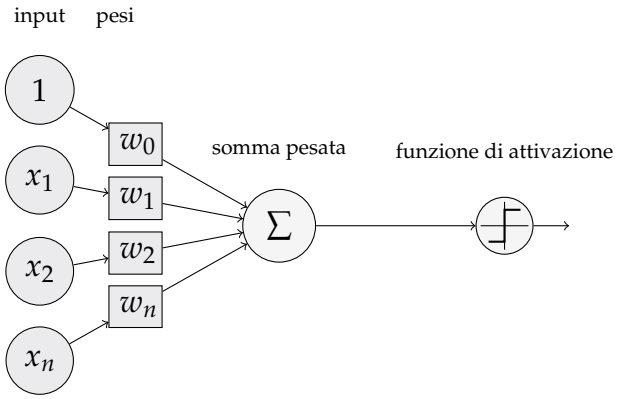

Figura 5: Struttura di un singolo percettrone.

Definizione formale di un percettrone:

$$P(x) = \text{Normalizzatore}(w \cdot x + b)$$

Dove w é la matrice pesi, x é il vettore colonna di input e il prodotto é quello righe-colonne. La funzione normalizzatrice puó essere di diverso tipo:

- Heaviside (come nell'esempio introduttivo): $H(n) = \begin{cases} 1 & \text{if } n > 0 \\ 0 & \text{else} \end{cases}$
- Signum: $\text{sgn}(n) = \begin{cases} 1 & n > 0 \\ 0 & n = 0 \\ -1 & n < 0 \end{cases}$
- Varie funzioni continue: cf. Funzioni di attivazione.
- Identitá (cf. ADALINE): $I(x) = x$

La regola di apprendimento per percettroni

La regola di apprendimento é la seguente:

$$w = w + \epsilon \cdot x \cdot (P(x) - E_x)$$

Dove ϵ é un parametro che regola la "ripiditá" dell'apprendimento e E_x é il risultato atteso dalla classificazione di x. Lo scopo dell' apprendimento é quello di minimizzare l' errore di classificazione commesso sugli esempi del training set.

Qualora esista una matrice w_i dei pesi tale da approssimare correttamente la funzione (a breve vedremo che non sempre esiste), allora la regola di apprendimento assicura che il percettrone convergerá ad una soluzione e:

- la soluzione ottenuta puó essere diversa da w_i.
- la convergenza si ottiene in un numero finito di passi.
- la scelta iniziale dei pesi non inficia la convergenza.

Limitazioni delle reti a singolo strato

Usando una rete semplice, come quella di percettroni a singolo strato, é possibile rappresentare una grossa quantitá di funzioni (di fatto ogni classificazione é una particolare funzione dall'insieme degli elementi all' insieme delle classi). Esistono peró alcuni tipi di funzioni che non é possibile rappresentare usando un solo strato.

Un esempio di questo genere di funzioni é lo XOR (or esclusivo):

X	Y	$X \oplus Y$
0	0	0
0	1	1
1	0	1
1	1	0

Il motivo di questa impossibilitá (provare per credere, si puó facilmente adattare il codice dell' esempio introduttivo) é dovuto al fatto che il percettrone é in grado di rappresentare solo funzioni linearmente separabili, cioé funzioni dove é possibile separare una classe dalle altre usando soltanto un iperpiano (vedi Figura 6).

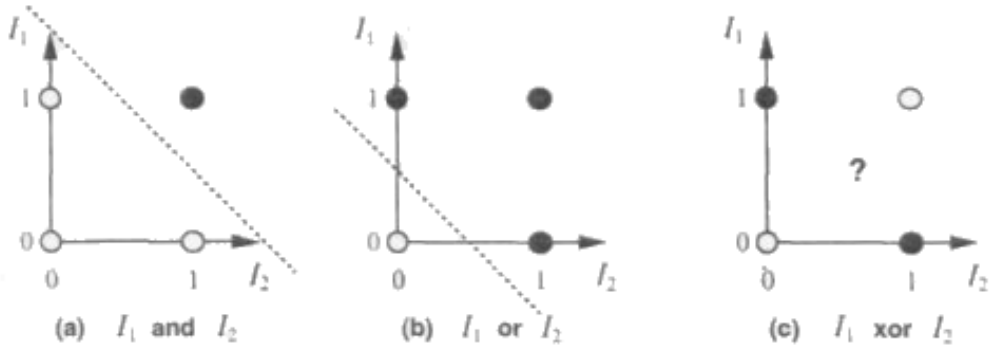

Figura 6: AND, OR e XOR di due variabili usando una retta per separare le due classi risultanti, laddove possibile.

Per poter rappresentare correttamente tutti i tipi di funzioni serve almeno uno strato nascosto.

2 Caratteristiche delle reti neurali artificiali

Vediamo ora quali sono piú in generale le caratteristiche delle ANN.

2.1 Numero di strati

Abbiamo visto che la potenza computazionale di una rete é ridotta se a singolo strato. Introducendo uno strato nascosto é possibile superare questo limite.

Superato il limite del singolo strato,quello che cambia ad aggiungere o togliere strati é prima di tutto il numero di neuroni. É difatti possibile ”ridurre” una rete a piú strati ad una con meno strati ma con un numero maggiore di neuroni per strato.

Generalmente gli strati aggiuntivi servono per introdurre “ragionamenti” di ordine superiore nella rete, in un modo simile a quello che si ottiene aumentando il grado dei polinomi usati per effettuare una regressione.

Questo permette quindi di approssimare funzioni piú “fluttuanti”. Come vedremo a breve, peró, aumentare il numero degli strati non implica un miglioramento delle prestazioni della rete in termini di riduzione dell' errore.

2.2 Funzioni di attivazione

É anche possibile usare neuroni con funzioni di attivazione diverse all'interno della rete. Un esempio puó essere quello di dover riconoscere oggetti dove alcune proprietá sono su un dominio continuo ed altre invece sono proprietá booleane.

2.3 Apprendimento

Abbiamo visto che prima di poter usare una ANN é necessaro eseguire un processo di apprendimento che corregga i pesi delle connessioni.

Back-propagating learning rule

La regola di apprendimento che abbiamo visto per il percettrone puó essere generalizzata per lavorare su ANN a piú strati. L'idea é sempre quella di “punire” e “premiare” gli input di un neurone in funzione del tipo di

errore commesso. Il nome é dovuto al fatto che la versione generalizzata inizia a calcolare l'errore dallo strato di output per poi ripercorrere gli strati a ritroso.
In generale l'apprendimento puó essere di quattro tipi:

Manuale

Nulla vieta di modificare manualmente i pesi delle connessioni.

Non supervisionato

Nel caso in cui non si abbiano a disposizione degli esempi giá classificati precedentemente, invece di usare la differenza tra il valore atteso e quello ottenuto, si usa una funzione di costo da minimizzare. Questa funzione di costo dipende da cosa si vuole modellare. In generale in questo caso la ANN viene usata per raggruppare i dati secondo certe caratteristiche.

Supervisionato

Quando si possiede un archivio di esempi con relative classificazioni giá conosciute. La regola di apprendimento é quella che abbiamo giá visto. Solitamente si applicano tecniche di validazione incrociata (cf. k-fold) per migliorare la qualitá dell' apprendimento.

Rinforzo

In questo caso normalmente alla ANN non viene fornito l'input direttamente, ma la rete (in quanto agente) lo ottiene a seguito di interazioni con l'ambiente. Ad ogni istante temporale l'agente esegue un'azione e l'ambiente genera un' osservazione ed un costo che l'agente "subisce". Lo scopo solitamente é di scoprire una politica di scelta delle azioni che minimizzi i costi subiti sul lungo termine.

2.4 Errore

Come abbiamo giá visto, la vita di una ANN si divide in due parti: apprendimento e utilizzo. L'obbiettivo della fase di apprendimento é quello di preparare al meglio la ANN per la seconda fase. Il punto é che la preparazione avviene su un insieme di esempi che é diverso da quello su cui poi la ANN andrá ad operare nella fase di utilizzo. Questo problema é definito come problema della generalizzazione. É abbastanza immediato

capire che il risultato finale dipende dalla dimensione e dalla qualitá del training set (vedi Figura 7).

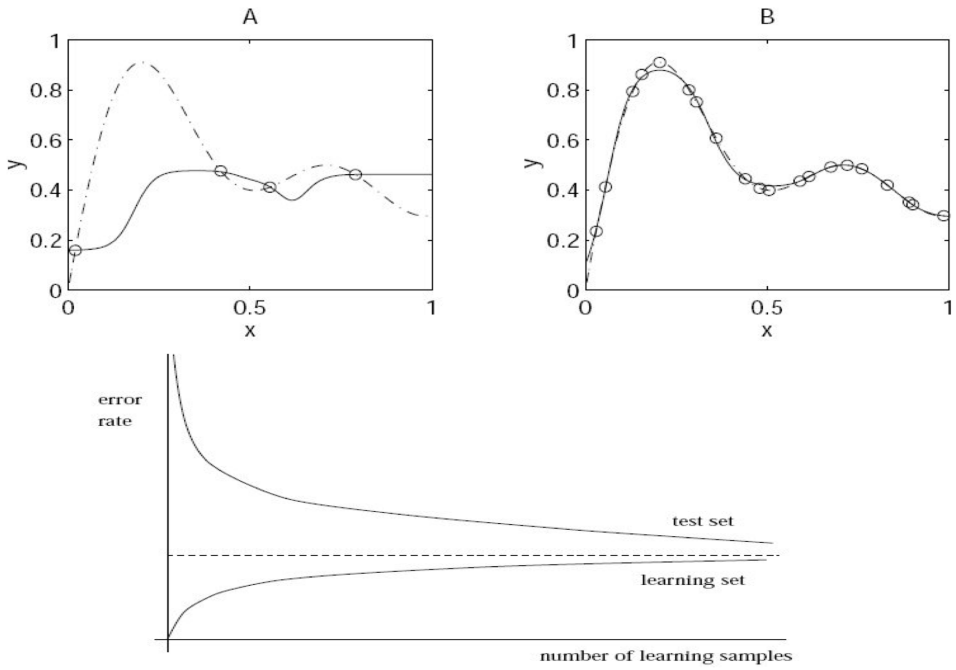

Figura 7: La differenza tra avere solo 4 esempi (A), averne molti e ben distribuiti (B) e l'errore in relazione alla dimensione del training set.

Un altro problema a cui sono soggette le ANN é quello del *overfitting*. Siccome il training set rappresenta solo un sottoinsieme dello spazio dei possibili input, il fatto che la ANN lo riconosca "troppo perfettamente" inficia le capacitá di generalizzazione della stessa. Questo solitamente avviene quando la rete ha molti neuroni negli strati nascosti e dove, a seguito della fase di apprendimento, la rete ha imparato ad "abusare" di alcune regolaritá presenti nel training set che non sono rispettate nell' insieme universo (vedi Figura 8).

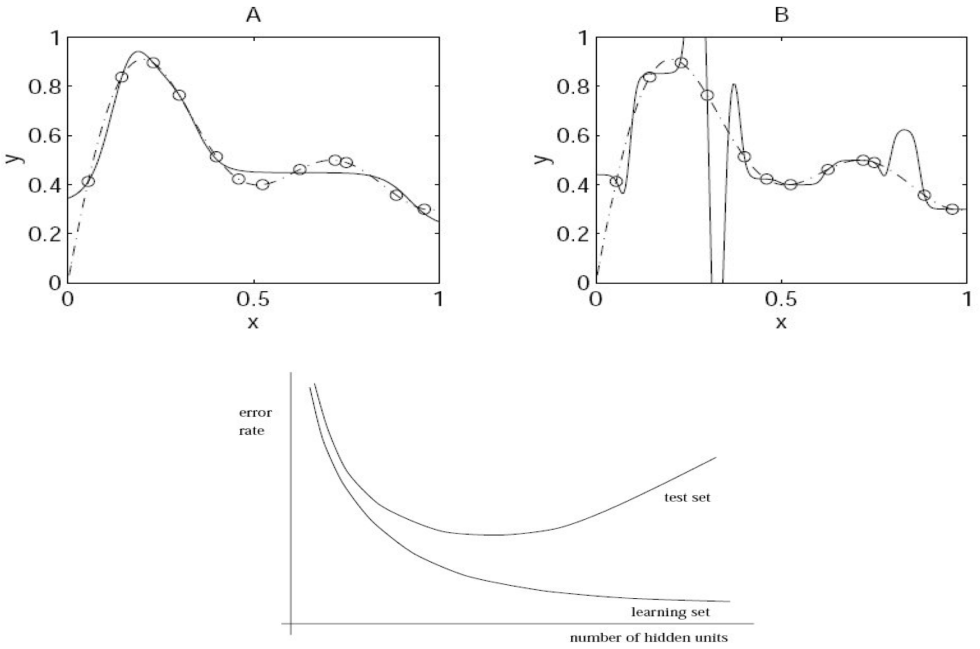

Figura 8: L'errore in relazione al numero di unitá nascoste.

Riassumendo, l'errore commesso da una ANN é riconducibile alle seguenti cause:

- L'algorimo di apprendimento: determina il livello di precisione raggiunto.
- Dimensione e qualitá del training set: defermina quanto si conosce a priori dell' insieme universo.
- Numero di unitá nascoste: determina il potere espressivo della rete.

3 Reti neurali ricorrenti

Le reti che abbiamo visto fino ad ora sono reti di tipo *feed-forward*, cioé reti che rispettano la seguenti proprietá:

- La struttura delle connessioni non contiene cicli.
- I neuroni di uno stesso strato non sono connessi tra loro direttamente.
- Lo strato di input comunica con il primo strato tra quelli nascosti.
- Il "flusso" dell'esecuzione passa da uno strato nascosto al successivo.
- L'ultimo livello nascosto comunica con lo strato di output.

Esistono altri tipi di reti dove queste regole non vengono rispettate. Il fatto di permettere connessioni cicliche non aumenta la capacitá di approssimazione della rete. Quello che si ottiene in compenso é una riduzione della dimensione della stessa.

L' introduzione di cicli permette di avere una "memoria" a corto termine utile per approssimare dinamiche temporali.

Apprendimento nelle reti ricorrenti

Per l'apprendimento é sempre possibile usare la regola di *back-propagation* che abbiamo visto per le reti feed-forward, generalizzata per questa superclasse di reti. Piú in generale la regola di apprendimento va applicata fino allo stabilizzarsi dei pesi.

3.1 Approcci principali

Vi sono due approcci principali alla realizzazione delle SRN (Simple Recurrent Network):

Reti di Elman

Le reti di Elman adotta una struttura a tre strati (contando anche lo strato di input) con l'aggiunta di uno strato extra di supporto allo strato nascosto. Questo strato extra viene definito *context layer* ed é collegato ciclicamente

allo strato nascosto, con la particolaritá che le connessioni dallo strato nascosto allo strato di contesto hanno il peso fissato ad 1. Lo strato di contesto viene utilizzato per tenere in memoria lo stato precedente dello strato nascosto.

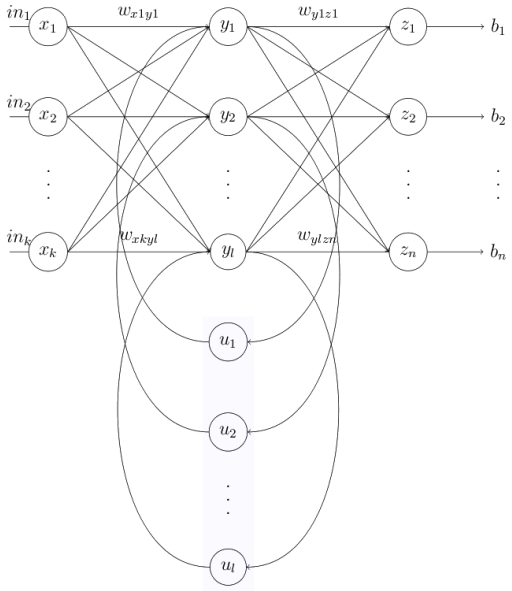

Figura 9: Struttura di una rete di Elman.

Reti di Jordan

Simili alle reti di Elman, lo strato di contesto riceve le informazioni dallo strato di output, anziché dallo strato nascosto.

Implementazione su reti feed-forward

Come giá accennato, l'uso tipico delle reti neurali ricorrenti é quello di approssimare dinamiche temporali. Lo strato extra permette di ricordare lo stato immediatamente precedente al fine di eseguire considerazioni di natura temporale. É comunque possibile implementare una rete feed-forward che risolva lo stesso problema. Il principio é quello di aumentare il numero di input $(x_1, x_2, \ldots, x_n)$ alla rete, assegnando ad ogni input aggiuntivo i valori della serie temporale nei successivi istanti $(x(t), x(t+1), \ldots, x(t+n))$ ottenendoli da un calcolo eseguito precedentemente.

4 Riferimenti

Per approfondire si consiglia di provare PyBrain che offre una implementazione semplice e completa delle reti neurali. Molti dataset interessanti sono disponibili su Kaggle. Per confrontare le prestazioni rispetto ad altre tecniche di machine learning, sempre per Python, c'é scikit-learn.
Letture per approfondire:

- Kevin Gurney, An introduction to Neural Networks, CRC Press, 1997
- Ben Kröse, Patrick van der Smagt, An introduction to Neural Networks, 1996