Deep Neural Networks for Learned Indexes Data Structures

Ettore Tancredi Galante

January 2019

Introduction

Neural Network for a Uniform Distribution

Bozza 1

Sia X una variabile aleatoria distribuita uniformemente su U, possiamo dire che:

$$P\{X = x\} = \frac{1}{|X|} = \frac{1}{n}; \text{ con x} \in \text{X}$$
$$P\{X \leq x\} = \hat{F}(x) = \frac{I(x)}{n}$$
con I(x) = Indice del predecessore di x in X.

Possiamo quindi dire che: $P\{X \leq x\} = F(x)$ e che tale F(x) è stimata da $\hat{F}(x)$.

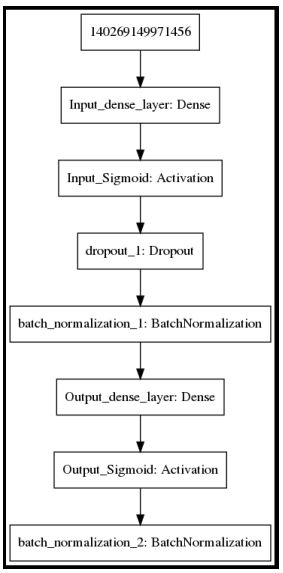
Esperimento 1

Si generi X $\subseteq$ Z tale che X sia uniformemente distribuito e che i suoi elementi siano equispaziati tra loro di un intervallo l $\in$ N.
Per stimare $\hat{F}(x)$, si necessita, nel contesto dell'esperimento, di apprendere una rete feed forward il cui output sia $\tilde{\text{F}}$(x).
La rete avrà uno strato di input composto da k neuroni, con:

$n = max(X)$;
$k = log_2 n$.

Lo strato nascosto di neuroni di tipo Dense, ossia con neuroni totalmente connessi a ciascun neurone di output, ha dimensione arbitraria k * 2 e funzione di attivazione Sigmoide, così come nel singolo neurone di output. Questo neurone restituirà $\tilde{\text{F}}$(x).
La funzione di loss è definita dall'errore quadratico medio.

La prima conformazione della rete



Define function to build the model

```
def build_model(neurons):
    """ Return keras network model """
    model = Sequential()
    model.add(Dense(neurons * 2, input_dim=neurons, name="Input_dense_layer"))
    model.add(Activation("sigmoid", name="Input_Sigmoid"))
    model.add(Dropout(0.2))
    model.add(BatchNormalization())
    model.add(Dense(1, name="Output_dense_layer"))
    model.add(Activation("sigmoid", name="Output_Sigmoid"))
    model.add(BatchNormalization())
    model.compile(
        loss='mean_squared_error', optimizer='sgd', metrics=['mean_absolute_error'])

    return model
```

```
history = model.fit(
    x_train,
    y_train,
    epochs=20,
    shuffle=True,
    batch_size=256,
    verbose=0,
    callbacks=[ktqdm(metric_format="{name}: {value:e}")],
    validation_data=(x_test, y_test)
)
```

Rappresentazione della prima rete costruita, con definizione del suo modello e dati per il fitting dello stesso

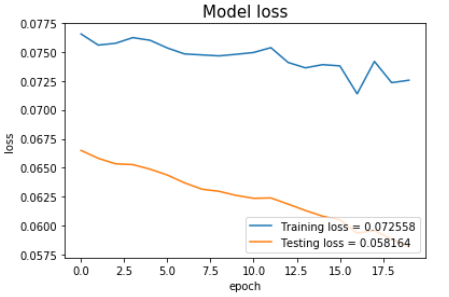
Si può notare, nella figura sovrastante, come sia costruito il modello della rete.
Il primo strato di input presenta k neuroni, con $n = max(X)$;
$k = log_2 n$.
Lo strato nascosto possiede un numero di neuroni pari a k * 2, valore scelto arbitrariamente, ed una funzione di attivazione del tipo sigmoide.
Dopo questo strato, prima dell'output a neurone singolo, si procede con un'operazione di dropout con valore 0.2, scelto arbitrariamente, e, successivamente ad una batch normalization. La seconda operazione, la batch normalization, è utile per minimizzare la funzione di loss mentre il dropout serve per fare in modo che la rete, ad ogni passo della fase di training, spenga con probabilità 1 - p, con p uguale al parametro passato alla funzione Dropout, ogni singolo neurone.
In tal modo si evita che i neuroni in una rete totalmente connessa sviluppino interdipendenze forti tra loro o, più generalmente, si evita di avere overfitting,

ossia quando un modello statistico complesso si adatta al campione osservato per via di un sovrannumero di parametri rapportato al numero di osservazioni.

Tale funzione è stata applicata con una probabilità p = 0.2 in quanto, su una rete con un totale di k * 3 neuroni, per iniziare ad eseguire esperimenti è parso un numero accettabile. Significa che ogni singolo neurone ha probabilità pari a 1 - p di venire disattivato. Ovviamente i risultati non sono stati affatto positivi e, come possiamo vedere nei grafici seguenti, le metriche di valutazione confermano la dichiarazione.

```
plot_history_graph(history, 'mean_absolute_error', 'mae')
```


```
plot_history_graph(history, 'loss', 'loss')
```

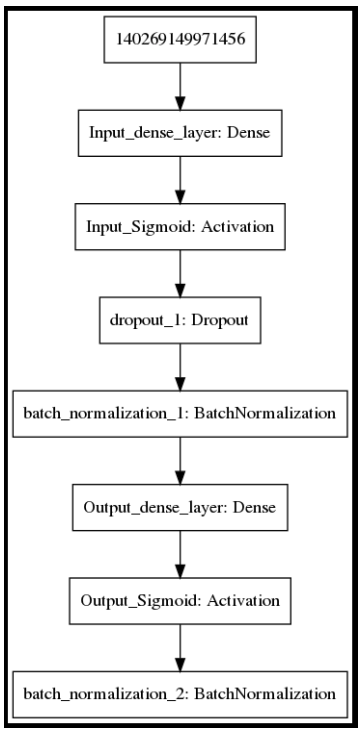



```
def prediction(num, width, model):
    bin = np.array([list(np.binary_repr(num, width))])
    print("Prediction on ", num, ":\t ", model.predict(bin))

(prediction(int(1), width, model)), (prediction(int(50 * 10001), width, model))
```

```
Prediction on  1 :       [[0.36054826]]
Prediction on  500050 :  [[0.54835874]]
```

I valori del grafico dell'errore assoluto medio, della funzione di loss presa in considerazione e delle conseguenti predizioni

Il secondo Esperimento

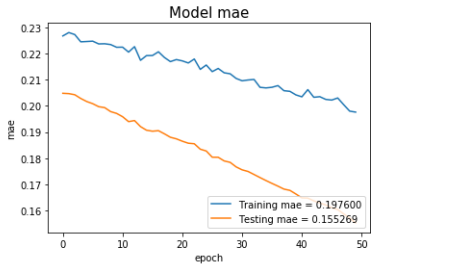



```
history = model.fit(
    x_train,
    y_train,
    epochs=50,
    shuffle=True,
    batch_size=256,
    verbose=0,
    callbacks=[ktqdm(metric_format="{name}: {value:e}")],
    validation_data=(x_test, y_test)
)
```

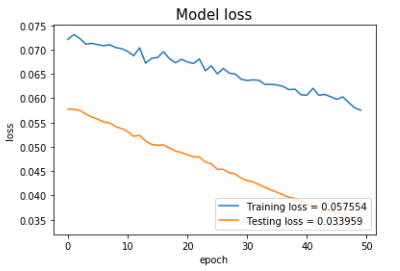
Il secondo modello della rete neurale

Dalla figura soprastante si può vedere che la conformazione della rete neurale non è cambiata.
Tuttavia, le epoche per il training sono state innalzate da 20 a 50 e, come si evince dai seguenti grafici e risultati delle predizioni, i risultati migliorano.

```
plot_history_graph(history, 'mean_absolute_error', 'mae')
```




```
plot_history_graph(history, 'loss', 'loss')
```



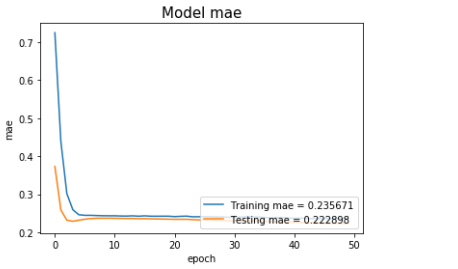

```
def prediction(num, width, model):
    bin = np.array([list(np.binary_repr(num, width))])
    print("Prediction on ", num, ":\t ", model.predict(bin))

(prediction(int(1), width, model)), (prediction(int(50 * 10001), width, model))
```

```
Prediction on  1 :        [[0.26790237]]
Prediction on  500050 :   [[0.6479814]]
```

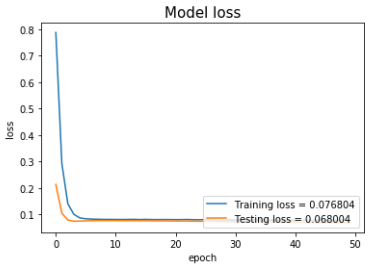
I grafici relativi allo scarto quadratico medio e allo scarto assoluto medio

Nel grafico relativo allo scarto assoluto medio si può notare come i valori risultino dimezzati rispetto al primo esperimento semplicemente con un numero più elevato di epoche. Il grafico della funzione di loss mostra un lieve miglioramento dei parametri sia in fase di testing che in fase di training.
Vediamo ora cosa accade cambiando il numero di neuroni nello strato nascosto, passando da k * 2 a $\frac{k}{2}$.

```
plot_history_graph(history, 'mean_absolute_error', 'mae')
```




```
plot_history_graph(history, 'loss', 'loss')
```




```
def prediction(num, width, model):
    bin = np.array([list(np.binary_repr(num, width))])
    print("Prediction on ", num, ":\t ", model.predict(bin))

(prediction(int(1), width, model)), (prediction(int(50 * 10001), width, model))
```

```
Prediction on  1 :        [[0.45117494]]
Prediction on  500050 :   [[0.5740772]]
```

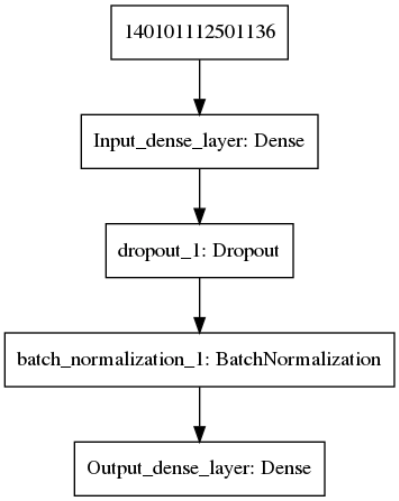
I grafici relativi ad un modello di rete con solo metà dei neuroni iniziali

Come è possibile vedere, i risultati tornano ad essere molto simili a quelli dell'esperimento numero 1.
Anche i risultati delle predizioni si allontanano nuovamente da quelli sperati. Dimezzare il numero di neuroni non è quindi la soluzione più efficiente per migliorare le prestazioni della rete.
Al contrario, sembra in realtà che quanti più neuroni sono presenti nello strato nascosto quanto più è precisa la predizione.

Terzo esperimento A - Rimozione di parametri inutili

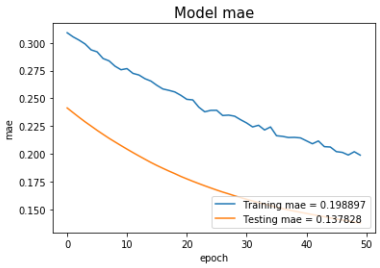
La rete è stata originariamente creata a immagine del primo modello con due batch normalization e droput con parametro 0.2.
Vedendo l'inadeguato tasso di miglioramento nei risultati si è deciso di procedere ad un approccio più drastico nella modifica della rete.
Modificando il modello in maniera non radicale si sono ottenuti risultati migliori che con l'inserimento di un secondo strato nascosto ed il mantenimento del dropout sul primo. I risultati in merito a questa dichiarazione saranno però discussi nella sezione seguente.
Gli esperimenti sono stati eseguiti aumentando anche il primo strato nascosto da k * 2 a k * 3 neuroni.
Per un miglioramento sulla terza cifra decimale, il numero di epoche è aumentato a 100 in maniera del tutto empirica. Oltre tale numero, per la dimensione del campione scelta, non paiono esserci miglioramenti significativi in relazione alla batch size.

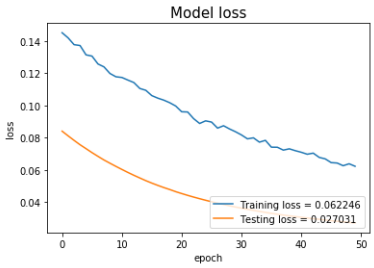
0.0.1 Rimozione Batch Normalization



Modello della rete senza Batch Normalization

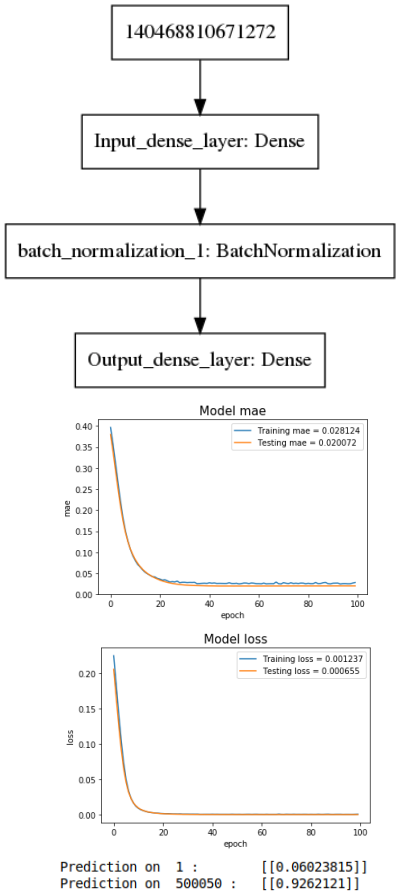
Dopo aver rimosso la batch normalization dal modello è possibile notare un deciso miglioramento nei grafici.
Le curve in fase di training e testing presentano un andamento decisamente più accettabile.





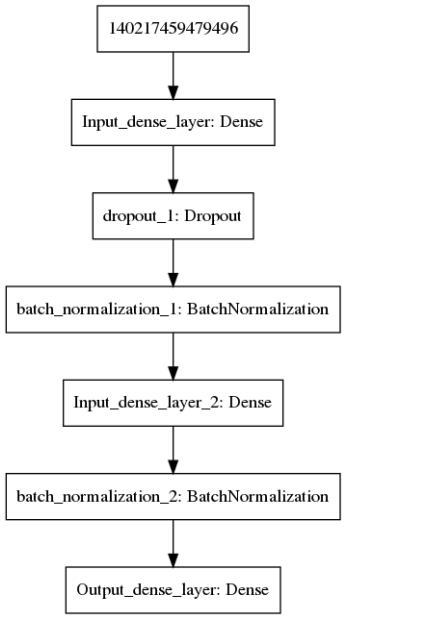
I risultati, tuttavia, si osservano ancor più notevoli se si rimuove anche il dropout dallo strato nascosto come si evince nel quarto esperimento.

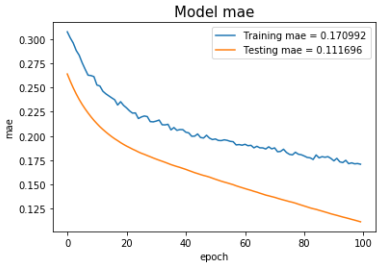
0.0.2 Rimozione dropout

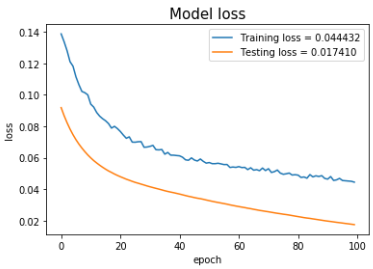


Il modello della rete al terzo esperimento A con grafici e predizioni

Terzo esperimento B - Inserimento di uno strato nascosto







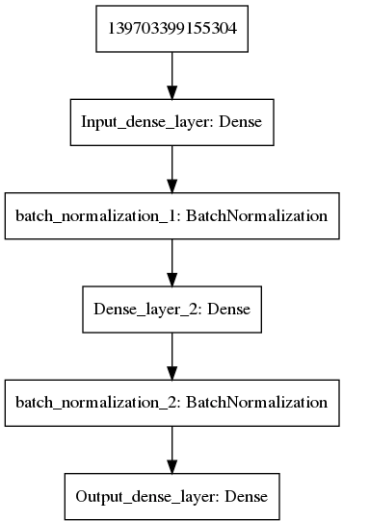

```
Prediction on  1 :        [[0.2519546]]
Prediction on  500050 :   [[0.7920837]]
```

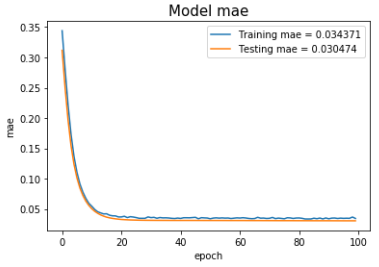
Il modello della rete al terzo esperimento B con grafici e predizioni

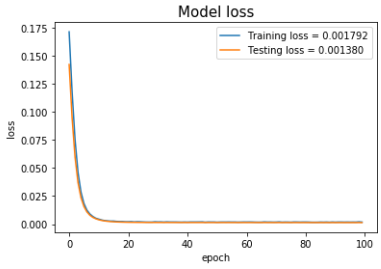
Si può osservare come l'aggiunta di un secondo strato nascosto non migliori la precisione quanto la rimozione del dropout.

Il terzo esperimento variante C mostra miglioramenti notevoli combinando esperimento A e B assieme.

Terzo esperimento C - Rimozione dropout ed inserimento di uno strato nascosto








```
Prediction on  1 :       [[0.04012125]]
Prediction on  500050 :  [[0.9235834]]
```

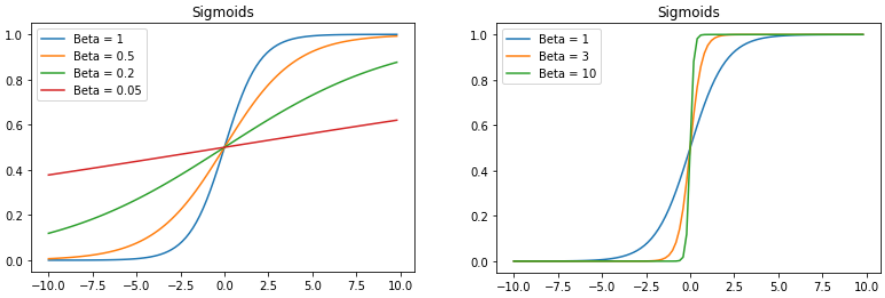
Il modello attuale della rete all'esperimento 3c

Quarta serie di esperimenti - Utilizzare una sigmoide personalizzata

Ottenuti dei risultati quantomento accettabili negli esperimenti precedenti, è ora necessario ottimizzare ulteriormente le predizioni. Una soluzione possibile può essere modificare la funzione di attivazione dei neuroni degli strati nascosti, utilizzando una funzione sigmoide differente da quella standard definita da Keras.

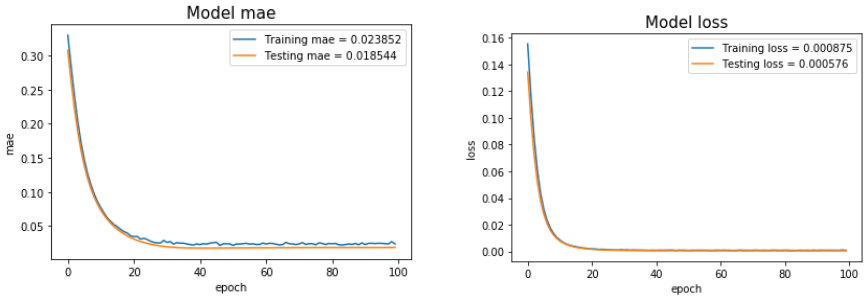
$$f(x) = \frac{1}{(1 + e^{(-x\beta)})} \tag{1}$$

Si veda quindi di seguito il comportamento della rete neurale quando si utilizzano funzioni con beta dal valore differente che, ovviamente, modificano lo slope della funzione. Si riportano i grafici in ogni sottosezione sia per la modifica del primo strato nascosto che per entrambi gli strati nascosti.



Le differenti funzioni sigmoidee utilizzate. L'unica differenza che presentano è il valore del parametro Beta. A sinistra vediamo Beta pari o minori ad 1, a destra pari o maggiori.

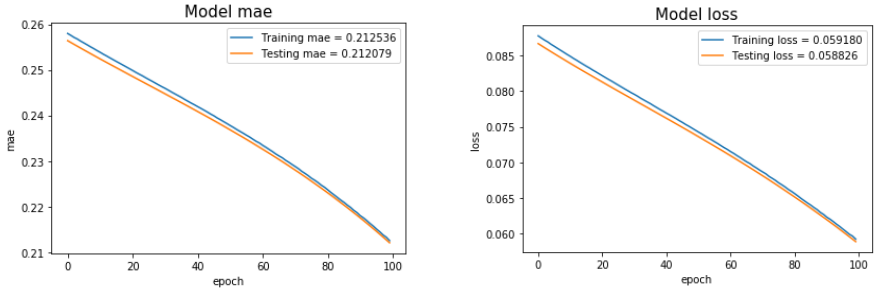
$\beta = \mathbf{0.05}$




```
Prediction on  500050 :    [[0.92871374]]
Prediction on  1 :         [[0.02720378]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 0.05 per il primo strato nascosto

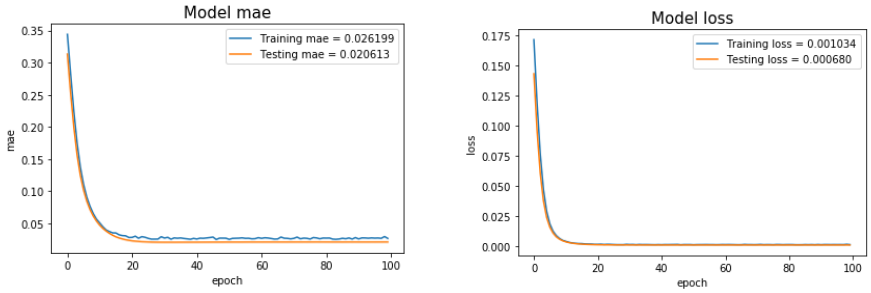
I risultati si mostrano qualitativamente superiori rispeto che all'uso della stessa sigmoide con slope basso in entrambi gli strati nascosti.




```
Prediction on  500050 :    [[0.5878973]]
Prediction on  1 :         [[0.3946688]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 0.05 per entrambi gli strati

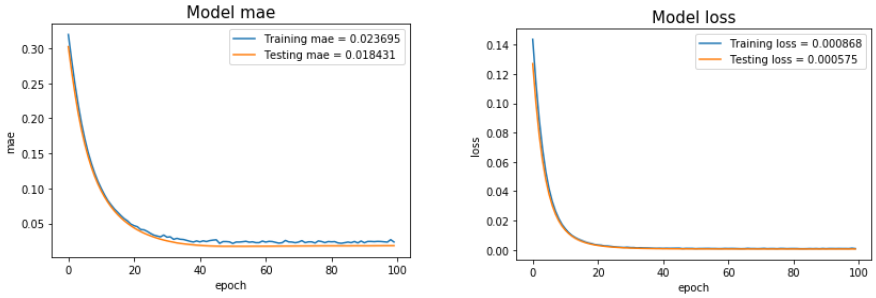
$\beta = \mathbf{0.2}$




```
Prediction on  500050 :    [[0.92808956]]
Prediction on  1 :         [[0.03202072]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 0.2 per il primo strato nascosto

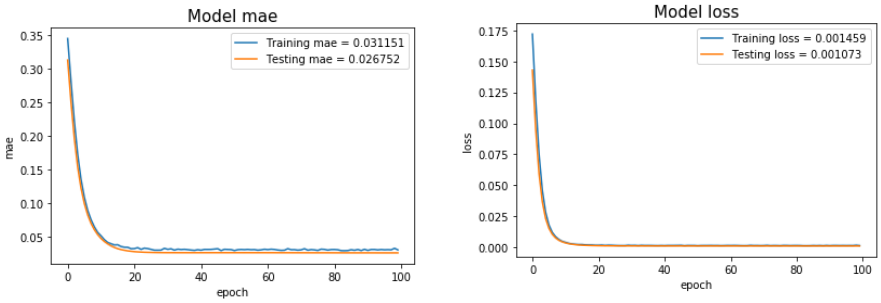
I risultati si dimostrano qualitativamente migliori nel secondo caso in questa situazione.




```
Prediction on  500050 :    [[0.92818606]]
Prediction on  1 :         [[0.02835259]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 0.2 per entrambi gli strati

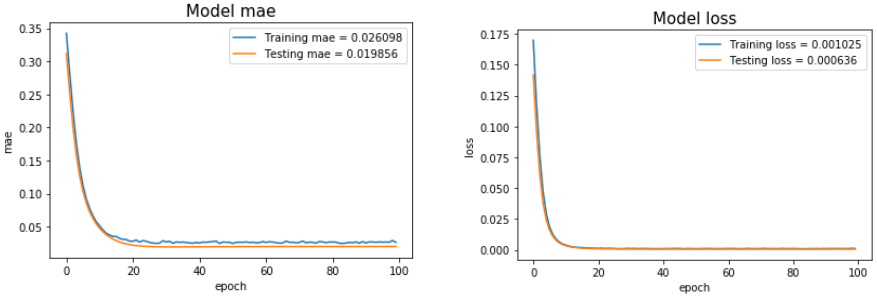
$\beta = \mathbf{0.5}$




```
Prediction on  500050 :    [[0.9265681]]
Prediction on  1 :         [[0.03952693]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 0.5 per il primo strato nascosto

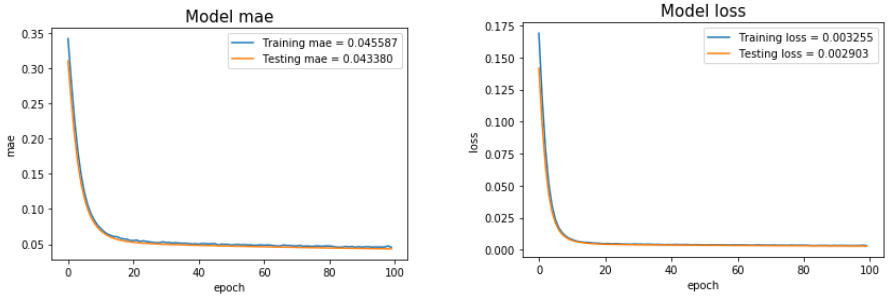
Come nel caso precedente, anche in questa situazione utilizzare la funzione personalizzata in entrambi gli strati si dimostra lievemente migliore come approccio.




```
Prediction on  500050 :    [[0.92725605]]
Prediction on  1 :         [[0.02775128]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 0.5 per entrambi gli strati

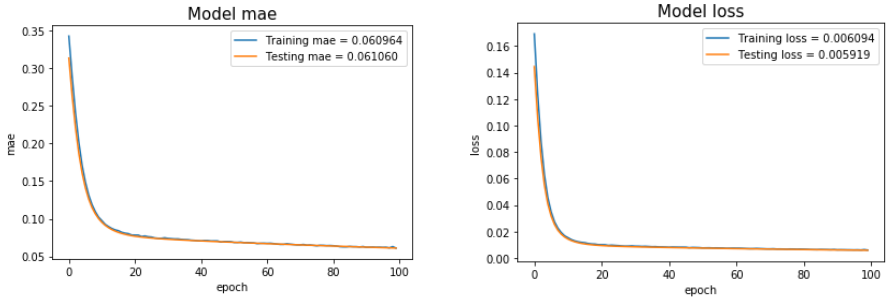
$\beta = \mathbf{3}$




```
Prediction on  500050 :   [[0.91189116]]
Prediction on  1 :        [[0.02870902]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 0.5 per il primo strato nascosto

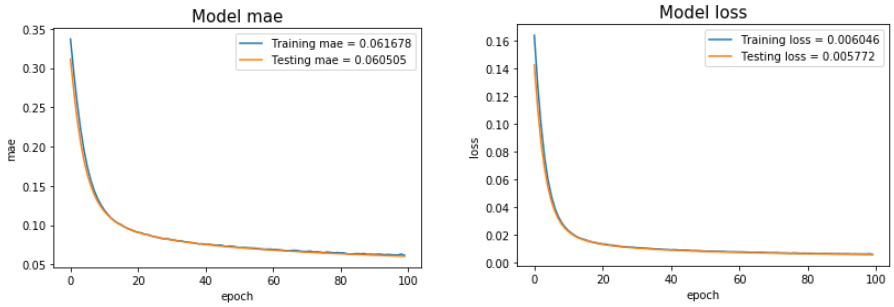
I risultati sembrano tornare come nei primi esperimenti di questa sezione. Usare la sigmoide personalizzata in entrambi gli strati peggiora le predizioni, anche se solo marginalmente.




```
Prediction on  1 :        [[0.03762117]]
Prediction on  500050 :   [[0.913077]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 0.5 per entrambi gli strati

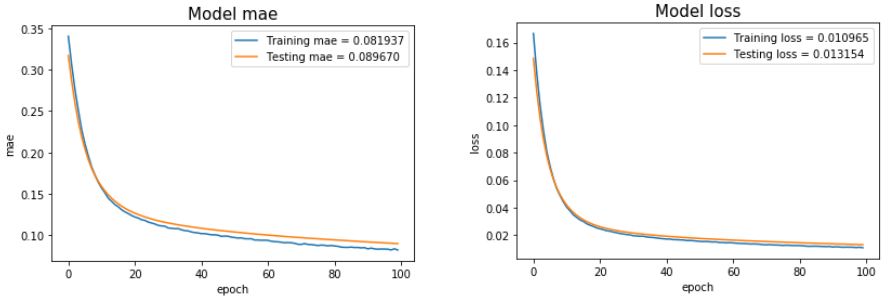
$\beta = 10$




```
Prediction on  500050 :    [[0.85885406]]
Prediction on  1 :         [[0.02257008]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 3 per il primo strato nascosto

Uno slope troppo ripido peggiora la situazione in entrambe i casi.




```
Prediction on  500050 :    [[0.865673]]
Prediction on  1 :         [[0.03714983]]
```

I grafici e le predizioni relativi ad un modello che utilizza una sigmoide con Beta = 10 per entrambi gli strati

Note per esperimenti futuri scritte in maniera parecchio rupestre e miglioramenti per il documento in sé

Ricorda di inserire le motivazioni sulla bontà delle predizioni. Perché non sono accettabili in alcuni punti? Quale dovrebbe essere il risultato sperato?
Spiegare il motivo della rimozione dei "Parametri inutili" ai fini dell'esperimento
Se non specificato, ricorda di specificare che si tratta di una distribuzione uniforme
Spiegare l'esperimento della ricerca del predecessore
La funzione sigmoide, come la funzione tangente iperbolica, dimostra di restituire valori troppo compressi, impedendo di ottenere predizioni accurate.
N.B.: Allega grafici con sigmoide e tanh.
L'idea adesso è definire una funzione sigmoide custom e sperimentare con valori di beta differenti da 1, minori o maggiori che siano.
N.B.: Allegare grafici con tale funzione con valori di beta diversi.
N.B.: Allegare grafici del mae e della loss della rete.
Modificare gli strati di neuroni nascosti.