Lez17_RetiRicorrenti2

November 29, 2023

0.1 Cos'è un Hidden state?

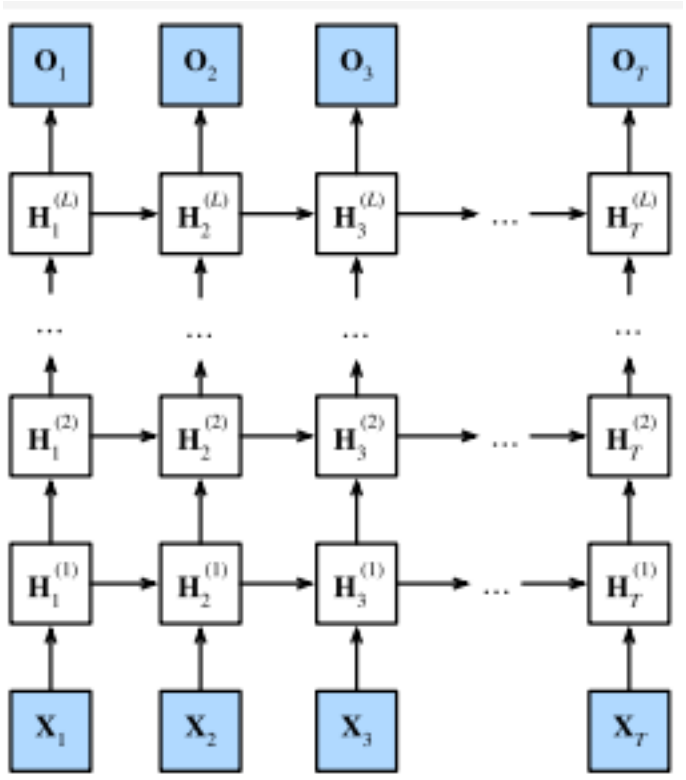
L'hidden state è output hidden layer. L'hidden state, usato per modellare una sequenza, che dimensioni avrà?

E' un vettore, o uno scalare?

E' un vettore, poichè ogni unità del livello ricorrente produce un output, che diventa un input per il neurone. Allora c'è coincidenza con la dimensione del livello. Nel caso di una serie temporale, per riassumere ciò che serve per una predizione, l'hidden state sarà un vettore, e quindi più unità nel livello ricorrente. Più è complessa la sequenza, più unità dovrò mettere nel livello nascosto.

0.2 Reti ricorrenti DEEP

Fino ad ora, avevamo un livello di input, di output e un livello nascosto. Possiamo parlare di rete *deep*? Sì. L'input che diamo in pasto, al primo istante, determina l'output della rete anche dopo 1000 step, c'è profondità nel *tempo*, non nel numero di livelli.



$H_1^{(1)}$, come vediamo, si propaga sia in verticale (quindi, fissato un istante, si propaga per formare l'output corrente), sia in orizzontale (che è l'istante successivo). La rete sarebbe quella verticale, più la ripetiamo verso destro, più istanti temporali abbiamo.

Stiamo solamente unendo più livelli *ricorrenti*.

Come può esserci utile?

Se pensiamo al disegno di una casa, in cui usiamo i movimenti della penna per capire la forma prodotta, con un solo livello non è facile capire la forma disegnata, bensì potrebbe servire un secondo livello che, partendo dalle coordinate, disegni le forme, un altro per capire se la forma è aperta o chiusa, e così via. Ogni livello lavora su *qualcosa*, producendo una *sequenza* per qualche altro livello successivo.

Non tutti i livelli devono essere ricorrenti, possono esserci anche livelli completamente connessi.

Se lavorassimo con tanti livelli, e avendo poche sequenze, quindi un input scarso, non abbiamo miglioramenti, perchè se il dato è scarso, non posso fare *magie*.

0.3 Esempio notebook rnn_timeseries.ipynb

La rete ricorrente in realtà non servirebbe.

```
tf.random.set_seed(42)
deep_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 1]),
    tf.keras.layers.SimpleRNN(32, return_sequences=True),
    tf.keras.layers.SimpleRNN(32),
    tf.keras.layers.Dense(1)
])
```

Quando si imposta `return_sequences=True`, significa che il livello RNN restituirà l'intera sequenza di output per ciascuna sequenza di input, e non solo alla fine.

```
fit_and_evaluate(deep_model, train_ds, valid_ds)
```

Ci si ferma con errore sul validation set del 6%, quindi non è stato utilissimo usarla *in questo caso*.

Come possiamo migliorare? Il dataset era composto da persone che usano i treni e gli autobus, consideriamo questi due input, e l'output e predire le persone che useranno i treni in futuro. Quindi in input avremo un vettore composto da due elementi, ovvero persone sul bus e sul treno.

```
df_mulvar = df[["bus", "rail"]] / 1e6  #usiamo sia bus sia treni, scala il numero
                                       #di passeggeri, per non avere numeri grandi.
df_mulvar["next_day_type"] = df["day_type"].shift(-1) #il giorno successivo è
                                                      #feriale o festivo? ce lo dice lui.
                                                      #E' legale? si, perchè è info nota.
df_mulvar = pd.get_dummies(df_mulvar, dtype=float)  # one-hot encode the day type
df_mulvar.head()
```

La rete potrebbe capire da sola i giorni che si succedono, ma se posso fornire una informazione, meglio farla!

date	bus	rail	next_day_type_A	next_day_type_U	next_day_type_W
2001-01-01	0.297192	0.126455	0.0	0.0	1.0
2001-01-02	0.780827	0.501952	0.0	0.0	1.0
2001-01-03	0.824923	0.536432	0.0	0.0	1.0

```
mulvar_train = df_mulvar["2016-01":"2018-12"]
mulvar_valid = df_mulvar["2019-01":"2019-05"]
mulvar_test = df_mulvar["2019-06":]


tf.random.set_seed(42)  # extra code - ensures reproducibility
mulvar_train.head()

train_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),  # usiamo tutte e 5 le colonne features
    targets=mulvar_train["rail"][seq_length:],  # forecast only the rail series
    sequence_length=seq_length,
    batch_size=32,
    shuffle=True,
    seed=42
)
valid_mulvar_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=mulvar_valid["rail"][seq_length:],
    sequence_length=seq_length,
    batch_size=32
)
```

Adesso l'errore è sceso a 4.8%, migliore, ce lo aspettavamo perchè abbiamo usato più informazioni.

Se facessimo la previsione anche delle persone che usano gli autobus?

```
tf.random.set_seed(42)

seq_length = 56
train_multask_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_train.to_numpy(),
    targets=mulvar_train[["bus", "rail"]][seq_length:],  # 2 targets per day
    sequence_length=seq_length,
    batch_size=32,
    shuffle=True,
    seed=42
)
```

```
valid_multask_ds = tf.keras.utils.timeseries_dataset_from_array(
    mulvar_valid.to_numpy(),
    targets=mulvar_valid[["bus", "rail"]][seq_length:],
    sequence_length=seq_length,
    batch_size=32
)

tf.random.set_seed(42)
multask_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(2)
])


fit_and_evaluate(multask_model, train_multask_ds, valid_multask_ds)
```

Abbiamo un errore del 4.6%, va bene!

Se avessimo fatto la stessa predizione in modo *naive*, ovvero basandoci sulle settimane precedente, l'errore ottenuto sarebbe stato dell'8%.

Se volessi splittare l'errore tra treno ed autobus:

```
Y_preds_valid = multask_model.predict(valid_multask_ds)
for idx, name in enumerate(["bus", "rail"]):
    mape = tf.keras.metrics.mean_absolute_percentage_error(
        mulvar_valid[name][seq_length:], Y_preds_valid[:, idx])
    print(name, float(mape))
```

Abbiamo un errore per i bus del 4.7% e per i treni del 5.1%.

L'uso della rete **deep** non porta vantaggi in questo caso. Ha senso perchè non dobbiamo fare operazioni eccessivamente complesse. Come è stata creata la rete deep? Così:

```
deep_multask_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.SimpleRNN(32, return_sequences=True),
    tf.keras.layers.SimpleRNN(32),
    tf.keras.layers.Dense(2)
])
fit_and_evaluate(deep_multask_model, train_multask_ds, valid_multask_ds)
```

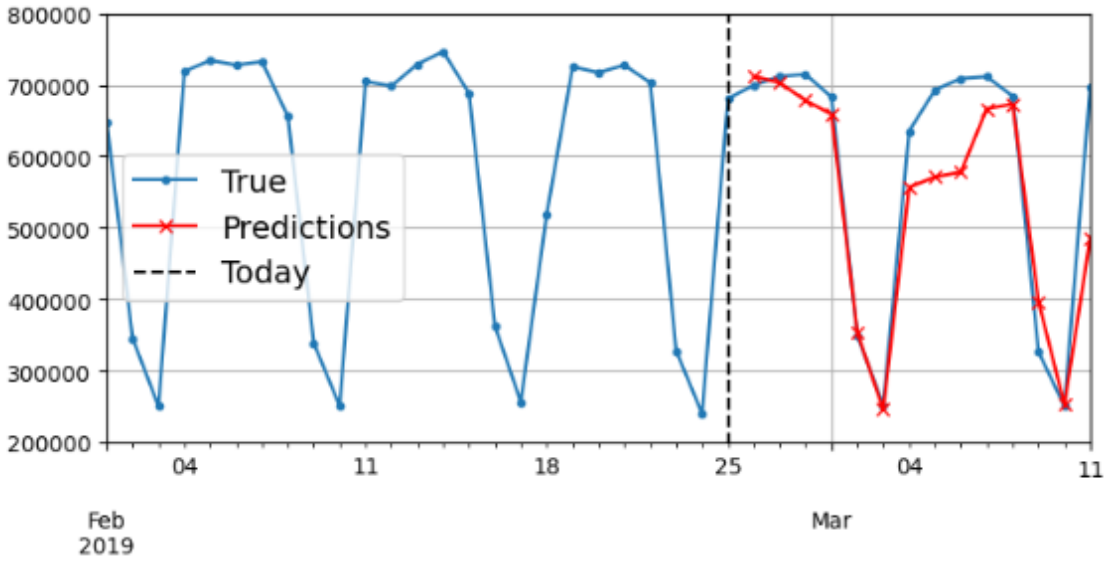
0.3.1 Se la predizione la volessi tra due settimane?

L'idea base è fare la predizione per il giorno successivo per 14 volte. Ma il problema è che l'errore si accumula!

```
import numpy as np

X = rail_valid.to_numpy()[np.newaxis, :seq_length, np.newaxis]
for step_ahead in range(14):
    y_pred_one = univar_model.predict(X)
```

```
X = np.concatenate([X, y_pred_one.reshape(1, 1, 1)], axis=1)
```



Non è male come predizione!

Con reti **RNN**, non facciamo più la predizione direttamente per il giorno dopo, ma direttamente per 14 giorni dopo. Devo creare un nuovo dataset con i valori target, lo shift non è più di 1 bensì di 14. Ciò viene fatto con lo snippet di codice identificato da:

```
def split_inputs_and_targets(mulvar_series, ahead=14, target_col=1):
    return mulvar_series[:, :-ahead], mulvar_series[:, -ahead:, target_col]
```

NB: Non è solo questo pezzo, è solo per identificarlo nel notebook.

```
tf.random.set_seed(42)

ahead_model = tf.keras.Sequential([
    tf.keras.layers.SimpleRNN(32, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
```

Perchè 14? Per ottenere tali predizioni con un'unica chiamata della rete. L'errore oscilla tra il 5% e l'8%.

```
MAE for +1: 26,383 (6.0)
MAE for +2: 27,742 (5.2)
MAE for +3: 30,237 (6.0)
MAE for +4: 32,245 (5.8)
MAE for +5: 36,581 (6.6)
MAE for +6: 32,990 (6.5)
MAE for +7: 35,386 (7.0)
MAE for +8: 37,795 (7.5)
MAE for +9: 32,449 (6.3)
MAE for +10: 35,460 (6.7)
MAE for +11: 39,183 (8.5)
MAE for +12: 39,800 (8.5)
MAE for +13: 34,058 (7.9)
MAE for +14: 34,050 (7.9)
```

Questa predizione è del tipo **sequence-to-vector**. Fornisco tutta la sequenza, solo alla fine ho predizione finale. Ovvero non specifico `return_sequences=True`. Potremmo fare **sequence-to-sequence**, ovvero la predizione dei 14 valori avviene ad ogni iterazione. Questo lo si fa con `return_sequences=True`.

0.4 Gestione delle sequenze lunghe

E' difficile addestrare queste reti per valori numerici, in quanto il gradiente può esplodere o annullarsi. Possiamo migliorare alcuni aspetti, come la scelta dei parametri, però non risolviamo definitivamente il problema. Altro *aiuto* è usare funzioni di attivazione come `tanh` invece che `ReLU`.

0.5 Reti Long Short-Tell Memory LSTM

Quando mandiamo input, attraverso rete ricorrente, una certa informazione (ad esempio il primo input) può essere perso dopo tante iterazioni.

E' sempre un male?

No, ad esempio, nel caso dei bus coi passeggeri, una previsione tra due mesi può non essere influenzata da ciò che avviene oggi.

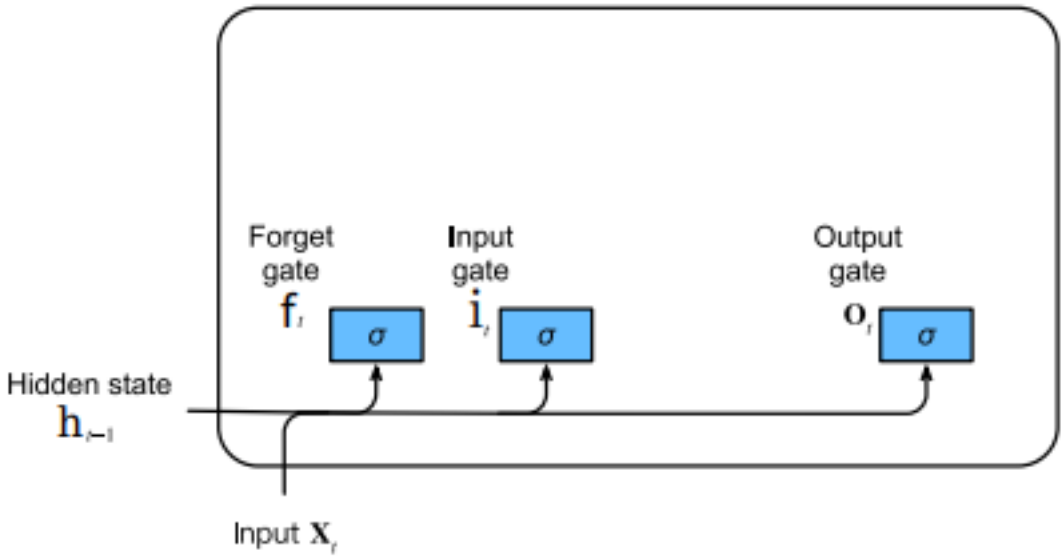
E' sempre un bene? No, ad esempio, in un insieme di parole, perchè le prime parole dovrebbero valere di meno?

Una rete ricorrente, semplice, ha memoria a lungo termine, ciò risiede nei pesi che non cambiano durante l'esecuzione. L'informazione in essi memorizzata, *non cambia*. La memoria a breve termine è invece dipendente dall'*hidden state*, che si aggiorna ad ogni istante temporale.

L'idea è combinarla. Si introduce il concetto di **memory cell**, posseduta da un **neurone ricorrente**, il quale la usa per la memorizzazione. Abbiamo anche dei **gate moltiplicativi**:

- **input gate**: l'input fornito al tempo t deve passare o meno dentro la memory cell? Cioè è importante da memorizzare?
- **forget gate**: Lo stato interno deve essere dimenticato?
- **outputgate**: Ciò che è memorizzato nello stato interno, deve essere mandato in output?

Graficamente:



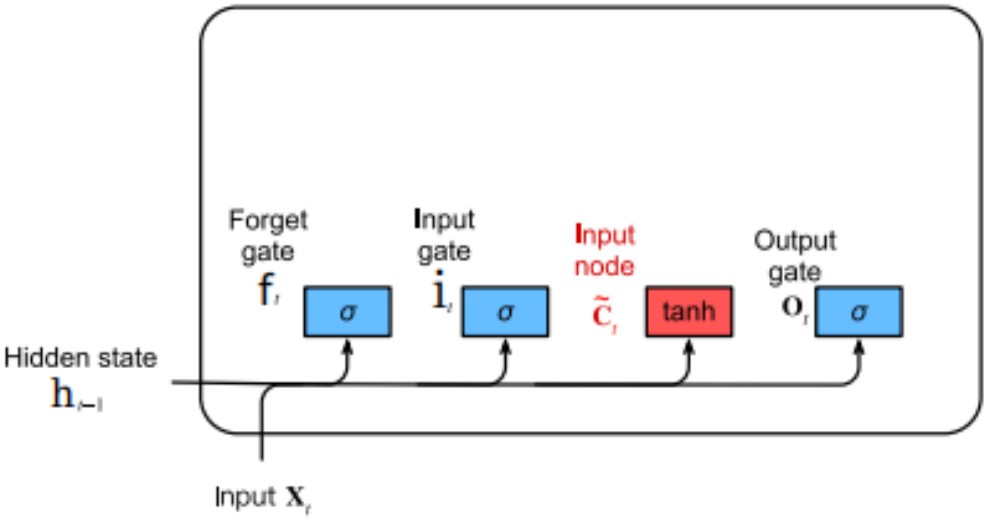
Questi gate si comportano come livelli completamenti connessi, che usano *sigmoid* (quindi valori nel range $(0, 1)$, ovvero funzionano come *interruttori*) in funzione dell'*hidden state* e dell'*input*.

Le formule sono:

$$\begin{aligned} \boldsymbol{i}_t &= \sigma(\boldsymbol{x}_t \boldsymbol{W}_{xi} + \boldsymbol{h}_{t-1} \boldsymbol{W}_{hi} + \boldsymbol{b}_i) \\ \boldsymbol{f}_t &= \sigma(\boldsymbol{x}_t \boldsymbol{W}_{xf} + \boldsymbol{h}_{t-1} \boldsymbol{W}_{hf} + \boldsymbol{b}_f) \\ \boldsymbol{o}_t &= \sigma(\boldsymbol{x}_t \boldsymbol{W}_{xo} + \boldsymbol{h}_{t-1} \boldsymbol{W}_{ho} + \boldsymbol{b}_o) \end{aligned}$$

Ogni componente ha una propria *matrice* e dei propri *bias*. La rete imparerà a gestire questi *interruttori*.

L'input portato alla cella di memoria al tempo t è rappresentato dal blocchetto rosso. E' l'**input node**, che ci aspettiamo sia connesso all'input gate.

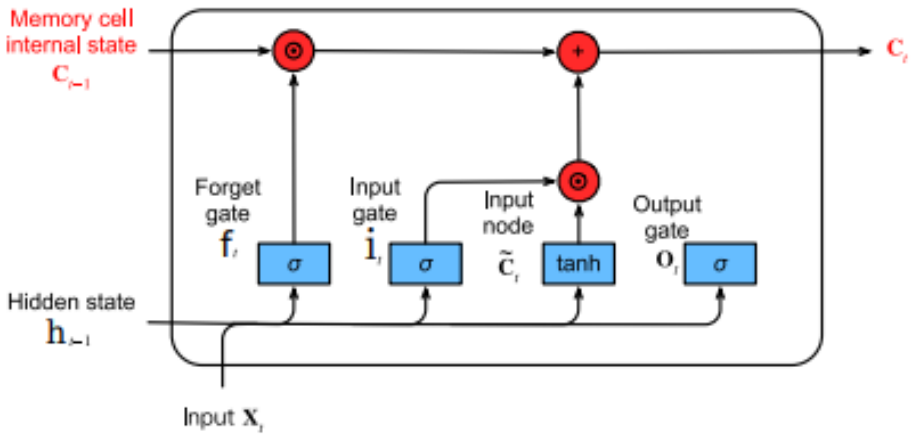


$\tilde{\boldsymbol{c}}_t \in (-1, 1)^h$ denotes the cell input node

$$\tilde{\boldsymbol{c}}_t = \tanh(\boldsymbol{x}_t \boldsymbol{W}_{xc} + \boldsymbol{h}_{t-1} \boldsymbol{W}_{hc} + \boldsymbol{b}_c)$$

L'input node sarà per h unità, per questo abbiamo $(-1, 1)^h$.

Manca la memoria, che ora introduciamo:



- $\boldsymbol{c}_t \in \mathbb{R}^h$ is the cell internal state

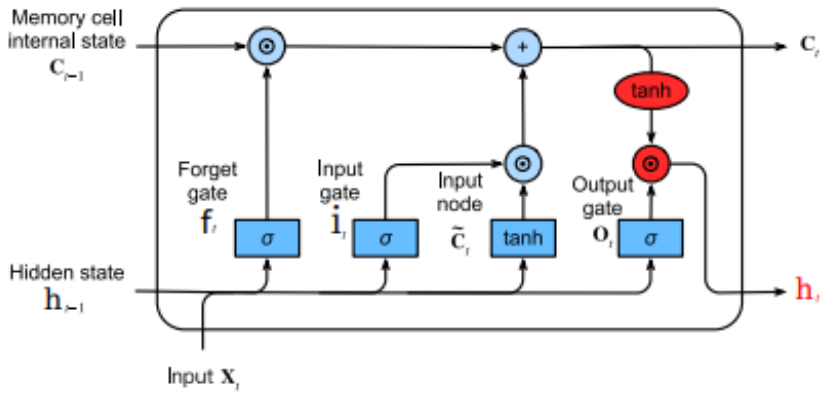
$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t$$

where $\odot$ denotes the element-wise product

Moltiplichiamo stato interno dello step precedente con *forget gate*, se quest'ultimo è nullo, dimentico ciò che ho messo nello stato interno. Se è pari ad 1, lo propago semplicemente.

Viene moltiplicato i per $\tilde{c}$. La somma di questo prodotto e ciò descritto prima, mi darà lo *stato interno*.

Come viene aggiornato l'*hidden state*? Entra in gioco l'*output gate*, che ha l'ultima parola per dire se bisogna mandare in output (se pari ad 1) o meno (output gate pari a 0).



- The hidden state $\boldsymbol{h_t} \in (-1, 1)^h$ is the output of the cell, as seen by next layer

$$\boldsymbol{h_t} = \boldsymbol{o_t} \odot tanh(\boldsymbol{c_t})$$

Se ho h unità nel livello LSTM, ciascun neurone ha tale comportamento. La dimensione dell'*hidden state* può essere visto come il numero di neuroni. Produce valori h_t, che normalmente vanno in un livello di *output*. Altrimenti, uscirebbe proprio h_t.

Adesso, possiamo avere un *hidden state* e potenzialmente non mandare nulla in output, perchè la rete ritiene meglio usarli dopo, mentre prima eravamo forzati nell'utilizzo.

0.6 Esempio notebook rnn_timeseries.ipynb

Le reti **LSTM** vengono usate per la predizione del carico di lavoro nei sistemi cloud.

La sua implementazione è semplice, basta dichiararla nel modello, come vediamo qui sotto. Il training è un po' più lento.

```
tf.random.set_seed(42)  # extra code - ensures reproducibility
lstm_model = tf.keras.models.Sequential([
    tf.keras.layers.LSTM(32, return_sequences=True, input_shape=[None, 5]),
    tf.keras.layers.Dense(14)
])
fit_and_evaluate(lstm_model, seq2seq_train, seq2seq_valid)
```

L'errore, giorno per giorno, è più o meno simile ai risultati del notebook precedente.

0.7 Wavenet

Abbiamo detto che è possibile usare anche reti ricorrenti. Ne è un esempio **WaveNet**, ottimale per le tracce audio, in grado di produrre audio in maniera sintetica. L'idea è sovrapporre 10 livelli convoluzionali e creare un *receptive field* per catturare i pattern a lungo termine nella traccia audio.

Con più livelli convoluzionali, ogni livello guarda una sua *parte*, ma alla fine i livelli più grandi riescono a processare parti sempre più grandi, fino a processarli tutti.

C'è un kernel di dimensione 2, e viene processata una parte *non contigua* (se vediamo `Conv1D`, i cerchi in neretto non sono vicini. Il parametro che definisce quanto devono essere separati è dato da *Dilation*).