Progetto Edge Computing

Anno accademico 2021/2022

Matteo Fresta e Matteo Pastorino Ghezzi

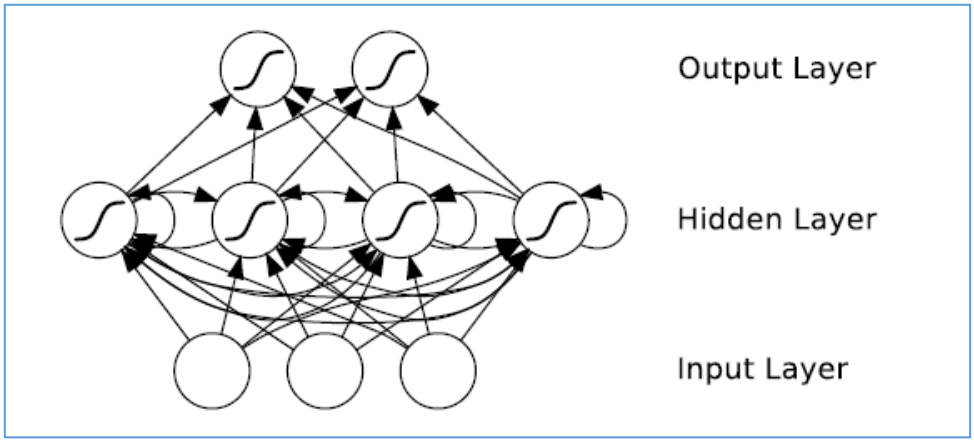
UNIVERSITÀ DEGLI STUDI
DI GENOVA

CONFRONTO TRA RETI NEURALI RICORRENTI (RNN) SU DATI INERENTI AL CONTAGIO DA COVID-19

1. Introduzione

Il presente elaborato è stato redatto con l'obiettivo di studiare e analizzare diverse reti neurali ricorrenti (RNN, *Recurrent Neural Network*) per predire le infezioni da Covid-19.

Le reti neurali ricorrenti sono una classe di architettura di reti neurali artificiali ispirata alla connettività ciclica dei neuroni nel cervello e utilizza loop di funzioni iterative per memorizzare informazioni.



Le connessioni ricorrenti consentono a una "memoria" di input precedenti di persistere nello stato interno della rete, influenzando così l'output della rete.

In totale sono state esaminate 6 reti neurali, tutte con differenti caratteristiche e proprietà: date le loro specificità, i risultati ottenuti hanno mostrato differenze che saranno discusse nei prossimi paragrafi.
Per testare la bontà di quanto ottenuto con un dataset (quello relativo alle positività giornaliere da inizio pandemia fino ad Agosto 2021), le stesse reti sono state utilizzate su altre due raccolte di dati, una delle quali rappresenta le nuove infezioni nel solo periodo estivo dell'anno corrente (2021), mentre la seconda le vaccinazioni in Italia.

2. Elaborazioni dati

Come accade per ogni problema di *Machine Learning*, uno dei passi fondamentali è l'acquisizione e manipolazione dei dati.
Il dataset di riferimento, almeno per la prima parte dell'analisi, è stato quello della Protezione Civile: all'interno della *repository* ufficiale di Github **(https://github.com/pcm-dpc/COVID-19),** è possibile trovare numerosi dati sull'andamento epidemiologico in Italia.
Per ogni RNN presa in esame, è stato letto lo stesso file csv, da cui sono state estratte le colonne *nuovi_positivi* e *data*; in seguito, i dati sono stati normalizzati (il motivo di questo passaggio sarà chiaro nel paragrafo riguardante la *LSTM*) e infine divisi in train e test set.

Ultimo passo prima della configurazione di una rete neurale, è la creazione del dataset secondo specifiche caratteristiche: la funziona che implementa ciò è la seguente:

```
# Create input dataset
# The input shape should be [samples, time steps, features]
def create_dataset (X, look_back = 1):
    Xs, ys = [], []

    for i in range(len(X)-look_back-1):
        v = X[i:(i+look_back)]
        Xs.append(v)
        ys.append(X[i+look_back])

    return np.array(Xs), np.array(ys)
```

Argomenti della funzione sono il dataset e *lookback*: questo parametro indica il numero di *time steps* precedenti che intendo usare come input per predire il prossimo periodo temporale.
Infine, siccome reti neurali come la *LSTM* richiedono una specifica forma in input (*[samples, time steps, features]*) è necessario effettuare un *reshape*, in quanto dalla funzione *create_dataset* otteniamo solo [*samples, features*].

Quanto specificato sopra sarà valido per ogni RNN univariata presente all'interno della relazione; dunque, il numero di *features* sarà pari a 1.
Prima di proseguire con il nostro studio, è necessario sottolineare che per ogni modello, è stato impostato un valore ottimo di unità, ovvero un numero di neuroni per ciascun *layer*, così come la variabile *batch_size* sarà sempre pari ad 1.
Di seguito, il numero di neuroni per ciascuna rete è:

- Vanilla LSTM: 8

- Stacked LSTM: 2
- BiDir LSTM: 1
- Vanilla GRU: 8
- Stacked GRU: 4
- BiDir GRU: 1

3. Vanilla LSTM

Definiamo *Vanilla LSTM* un modello *LSTM* che presenta un unico *hidden layer* di unità *LSTM*, e un *output layer* per la predizione. La funzione che crea il modello è la seguente:

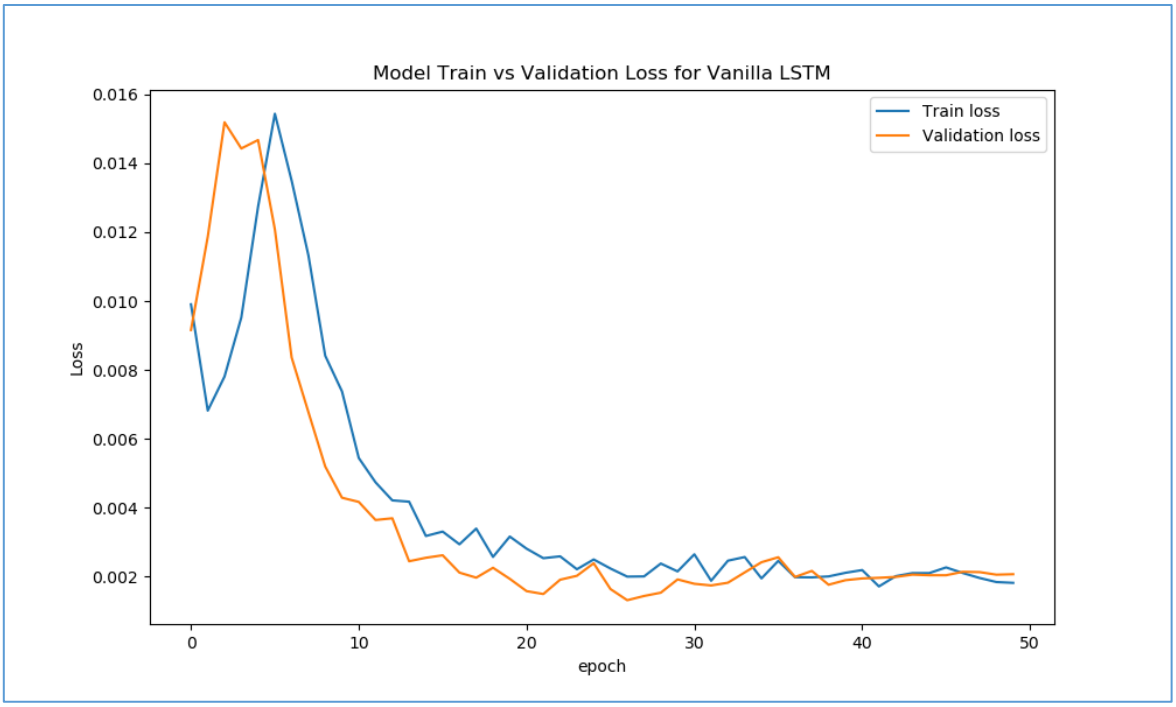
```
def create_lstm(units):
    model = Sequential()
    # Input layer
    model.add(LSTM(units = units, batch_input_shape=(n_batch, n_features,
lookback), stateful=True)) # return_sequences=True
    model.add(Dropout(0.01)) # Every LSTM layer should be accompanied by a Dropout
layer.
                             # This layer will help to prevent overfitting by
ignoring randomly selected neurons during training,
                             # and hence reduces the sensitivity to the specific
weights of individual neurons.
                             # 20% is often used as a good compromise between
retaining model accuracy and preventing overfitting.
    # Output layer
    model.add(Dense(1))
    #Compile model
    model.compile(optimizer='adam',loss='mse')
    return model
```

Con *Sequential()* (dalla libreria *Keras*) inizializzo la mia rete in cui andrò ad aggiungere con le funzioni *add()* i vari strati. *LSTM()* può prendere diversi argomenti in ingresso: nel nostro caso, il numero di unità è pari a 8 mentre all'interno di *batch_input_shape*, *n_batch* è pari a 1, *n_feature* = 1 (valori già menzionati in precedenza), e il *lookback* è 15 (numero di giorni che mi dovrebbe assicurare di non perdere la stagionalità all'interno del mio dataset).

È inoltre buona norma inserire un layer di *Dropout* per evitare l'overfitting: tale valore è stato settato sull'1% (questa percentuale sarà mantenuta su tutti i modelli all'interno della relazione).
Infine lo strato di uscita è identificato tramite la funzione *Dense()* e con *compile()*.

Definito il modello, è necessario operare il *fitting* sul training set: la funzione *fit()* di *Keras*, oltre a prendere in ingresso il training set, ha come argomenti il numero di iterazioni (*epochs*), *batch_size* (dove il suo valore ottimo è pari a 1), ed infine *validation_data*, che ci permette di analizzare il *loss* sul test set al termine di ogni iterazione.

Per *epochs=50*, i valori di *loss* sono i seguenti:

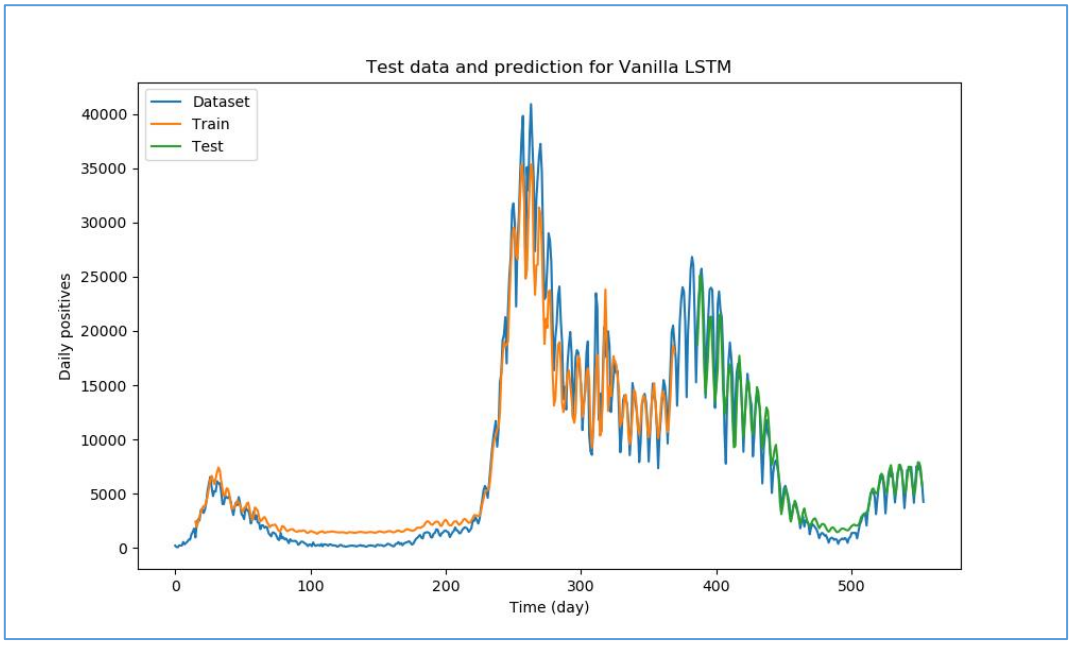


In generale l'errore è basso e stabile intorno alla 40° iterazione.

La predizione è calcolata dopo aver trasformato i dati nella loro forma originale: in questo modo sarà possibile visualizzare il grafico delle previsioni, oltre che calcolare l'errore e dunque le performance del modello.

Molta attenzione è stata dedicata sulla funzione di *plotting*, in quanto è stato necessario traslare i dati delle predizioni affinché fossero allineati con l'asse x del dataset originale.

In definitiva, il risultato è il seguente:



Come indicato dalla legenda, è stato deciso di disegnare, oltre al test set, anche il train per osservare come quest'ultimo si è comportato il modello sull'intero dataset.
I risultati ottenuti posso considerarsi soddisfacenti, seppur è chiaro che la *Vanilla LSTM* non sia il modello che meglio possa rappresentare questo problema; in termini di errore medio assoluto ed errore quadratico medio, otteniamo:

```
Vanilla LSTM:
Mean Absolute Error: 1626.2701
Root Mean Square Error: 1935.8952
```

I valori sono discretamente alti, sebbene siano stati utilizzati parametri ottimi, trovati dopo numerosi tentativi. Per provare a diminuire l'errore, è possibile aumentare il numero di iterazioni ma in questo modo non abbiamo la certezza che il modello migliori; per ovviare a ciò, *Keras* fornisce una funziona chiamata *EarlyStopping*, che può essere aggiunta come argomento durante il *fitting* sotto la voce *callbacks*.
Compito di questa *callback* è quello di fermare il *fitting* appena il parametro scelto sarà stabile per un numero di iterazioni consecutive decise dalla utente: nel nostro caso, il parametro è il *validation loss* con un'attesa pari a 10 *epochs*.

La funzione si presenta dunque così:

```
def fit_model(model):
    # This callback will stop the training when there is no improvement in the loss
for 'patience' consecutive epochs.
    early_stop = EarlyStopping(monitor = 'val_loss', patience = 10, verbose=1)
    history = model.fit(X_train, y_train, epochs = 200, validation_split = 0, #0.2
                    batch_size = 1, validation_data=(X_test, y_test), shuffle =
                   False, callbacks = [early_stop]) # batch_size = 16
    return history
```

Grazie alla callback, il fitting del modello si ferma dopo 34 iterazione e l'errore diminuisce:

```
Vanilla LSTM:
Mean Absolute Error: 1361.2967
Root Mean Square Error: 1654.0150
```

Visti i vantaggi che *callbacks* porta, questa verrà usata anche nei prossimi modelli che andremo ad analizzare.

4. Stacked LSTM

In generale, è possibile 'impilare' più *hidden layer* per creare una *Stacked LSTM*. Sebbene non ci siano limiti al numero di *layer* che l'utente può inserire, secondo Aurélien Géron nel suo libro *Hands-On Machine Learning with Scikit-Learn, Keras & TensorFlow*, di solito il numero di *hidden layer* varia da 1 a 5:

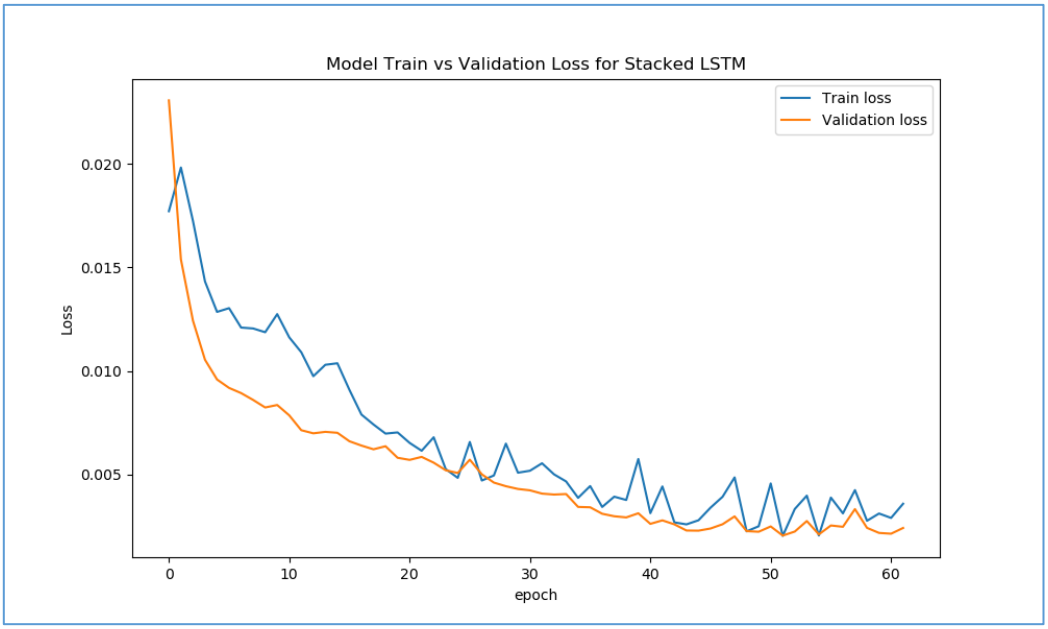
Table 10-1. Typical regression MLP architecture

Hyperparameter	Typical value
# input neurons	One per input feature (e.g., 28 x 28 = 784 for MNIST)
# hidden layers	Depends on the problem, but typically 1 to 5
# neurons per hidden layer	Depends on the problem, but typically 10 to 100
# output neurons	1 per prediction dimension
Hidden activation	ReLU (or SELU, see Chapter 11)
Output activation	None, or ReLU/softplus (if positive outputs) or logistic/tanh (if bounded outputs)
Loss function	MSE or MAE/Huber (if outliers)

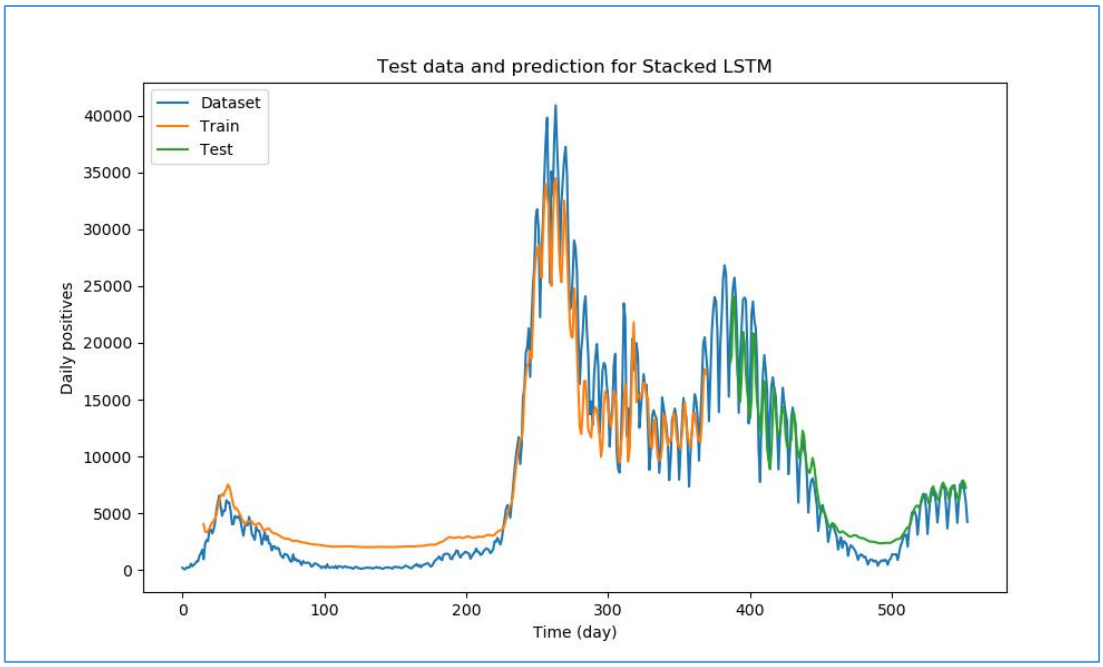
1 - Tabella presente nel libro

Nel definire il modello, è importante inserire *return_sequences=True* nell'*input layer*: questo ci permette di avere un output 3D dall'*hidden layer* come ingresso per il layer successivo.

Procedendo come con la *Vanilla*, dopo la 62° iterazione il fitting si interrompe in quanto non notiamo miglioramenti nel val_loss:



Il grafico delle predizioni è il seguente:



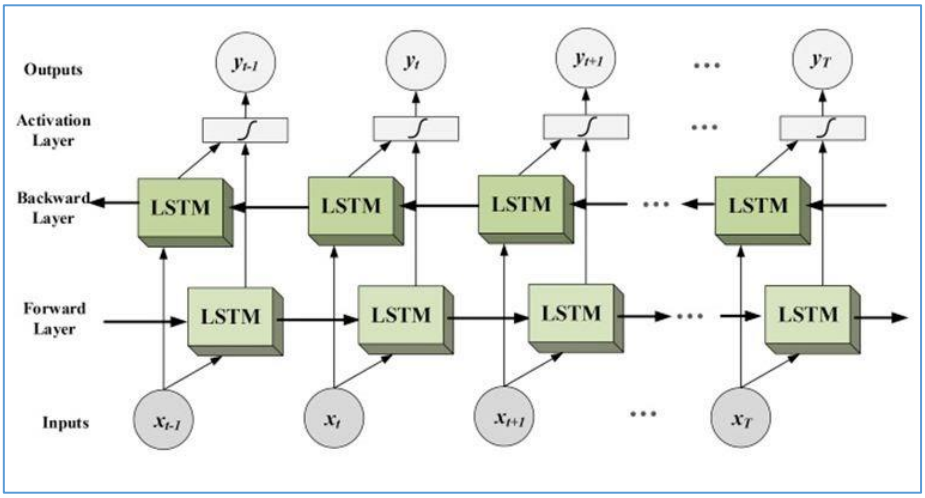
Osservando il solo grafico, ci aspettiamo di avere un errore superiore al caso *Vanilla*:

```
Stacked LSTM:
Mean Absolute Error: 1506.9942
Root Mean Square Error: 2002.0650
```

Come prevedibile, l'errore è più alto, dettato dall' *overfitting*, facilmente individuabile nel grafico delle *losses*.

5. Bidirectional LSTM

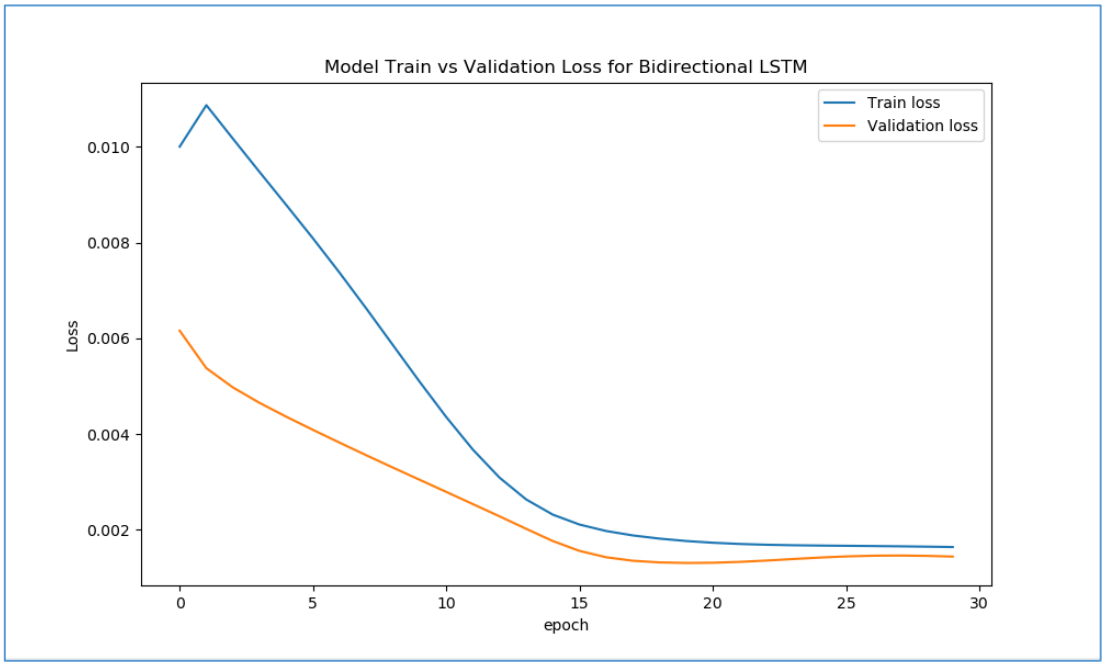
Ultimo modello LSTM analizzato in questa relazione è quello bidirezionale: qui il flusso degli ingressi viaggia sia in avanti (dal passato al futuro) che indietro (dal futuro al passato), permettendo di mantenere l'informazione del presente e del passato.



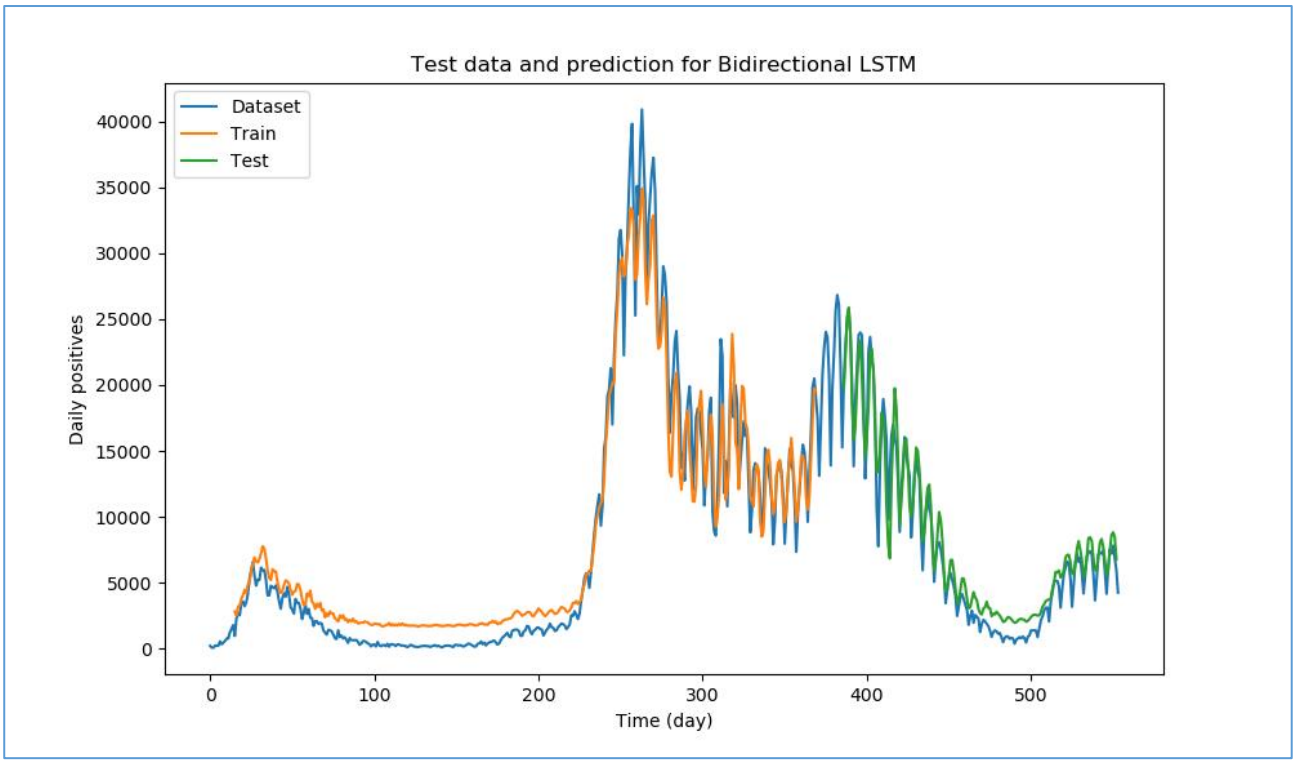
La funzione che implementa tutto ciò è molto semplice ed è simile a quanto già visto:

```
# Create BiLSTM model
def create_bilstm(units):
    model = Sequential()
    # Input layer
    model.add(Bidirectional(LSTM(units = units, input_shape=(X_train.shape[1],
X_train.shape[2]))))
    '''# Hidden layer
    model.add(Bidirectional(LSTM(units = units)))'''    # if we add the layer
again, put return_sequence = TRUE above
    # Output layer
    model.add(Dense(1))
    #Compile model
    model.compile(optimizer='adam',loss='mse')
    return model
```

I grafici delle *losses* qui sono molto più lisci e dopo 30 iterazioni otteniamo:



Da cui ricaviamo la seguente predizione:



Con errore:

```
Bidirectional LSTM:
Mean Absolute Error: 1221.6948
Root Mean Square Error: 1549.4605
```

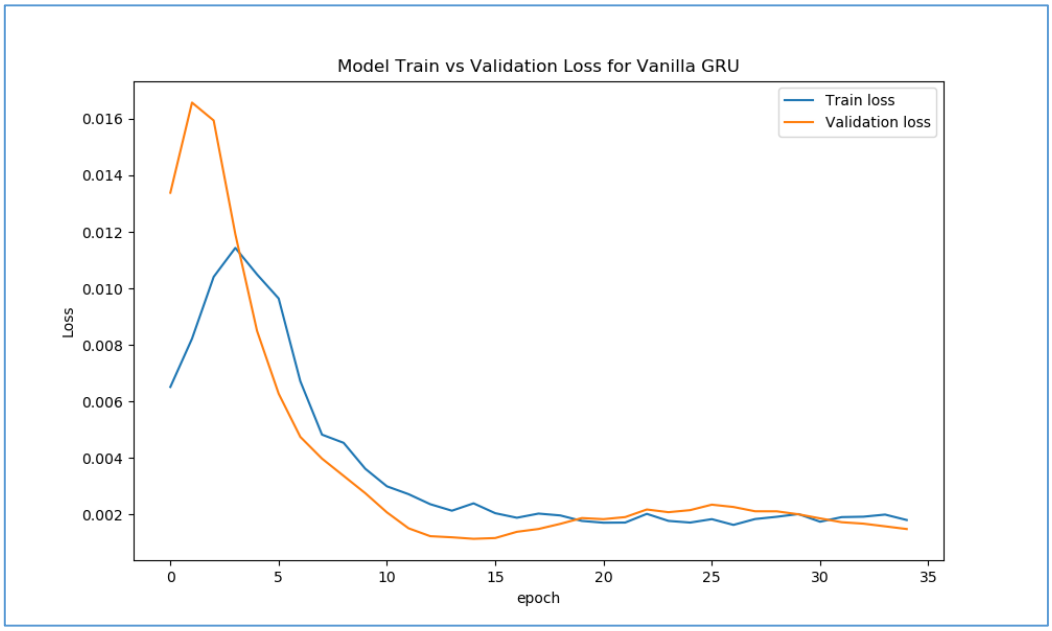
Per ora, la *LSTM* bidirezionale si dimostra tra le migliori reti viste, dovuto appunto al fatto che il *layer* di uscita riceve informazioni dagli stati futuri e passati contemporaneamente.

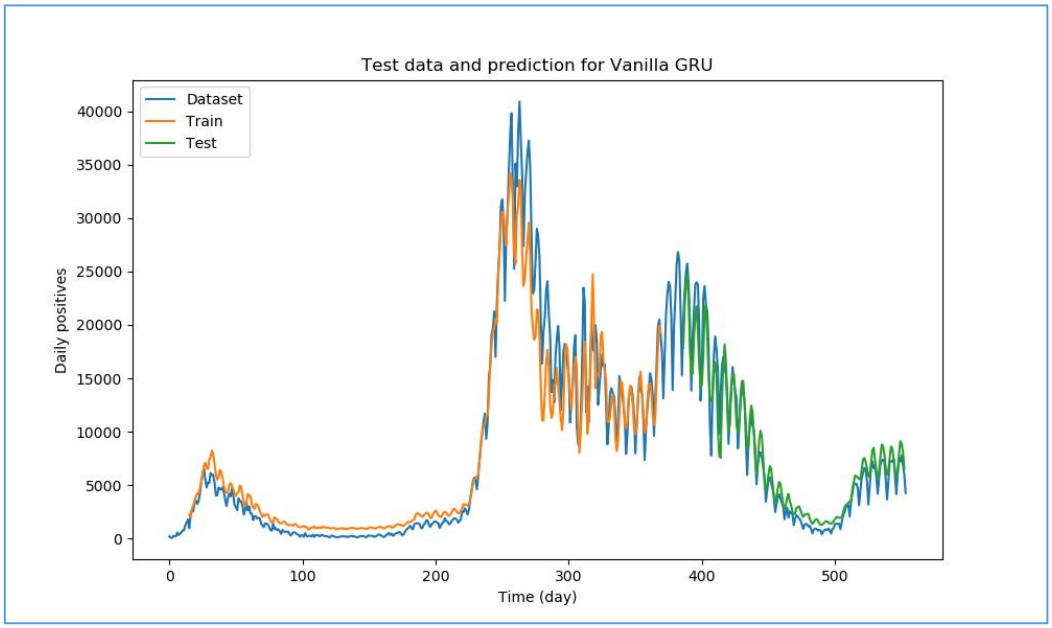
6. Vanilla GRU

Le unità *GRU* si differenziano dalle *LSTM* in quanto non usano un'unità di memoria per controllare il flusso di informazioni e quindi si servono subito di tutti gli *hidden states* senza alcun controllo. Date queste premesse, è chiaro che *GRU* utilizza meno parametri ed è dunque più veloce nel *training*.
La funzione è praticamente uguale alla *Vanilla LSTM*:

```
def create_gru(units):
    model = Sequential()
    # Input layer
    model.add(GRU (units = units, return_sequences = False,
                input_shape = [X_train.shape[1], X_train.shape[2]]))
    model.add(Dropout(0.01))
    # Output layer
    model.add(Dense(1))
    #Compile model
    model.compile(optimizer='adam',loss='mse')

    return model
```

Le prestazioni del nostro modello *GRU* sono:



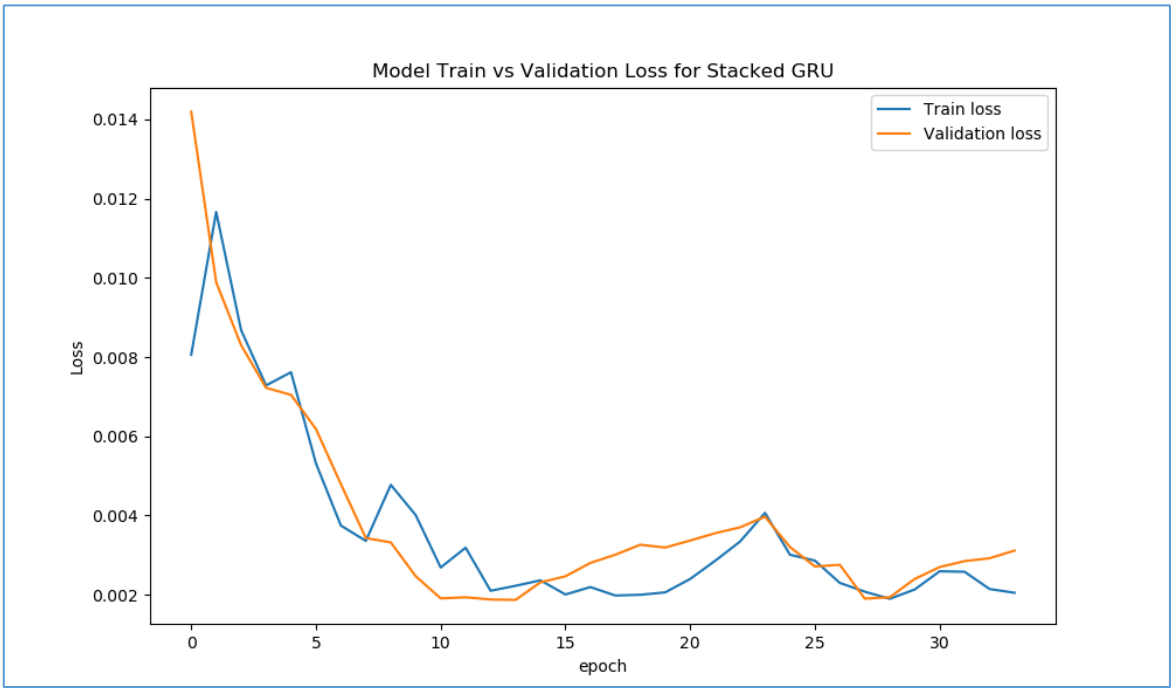


Si noti come le linee del grafico si avvicinino di più ai valori effettivi rispetto ai grafici ottenuti con la *LSTM*.
L'errore che otteniamo non si discosta dall'architettura precedente ma anzi si dimostra migliore:

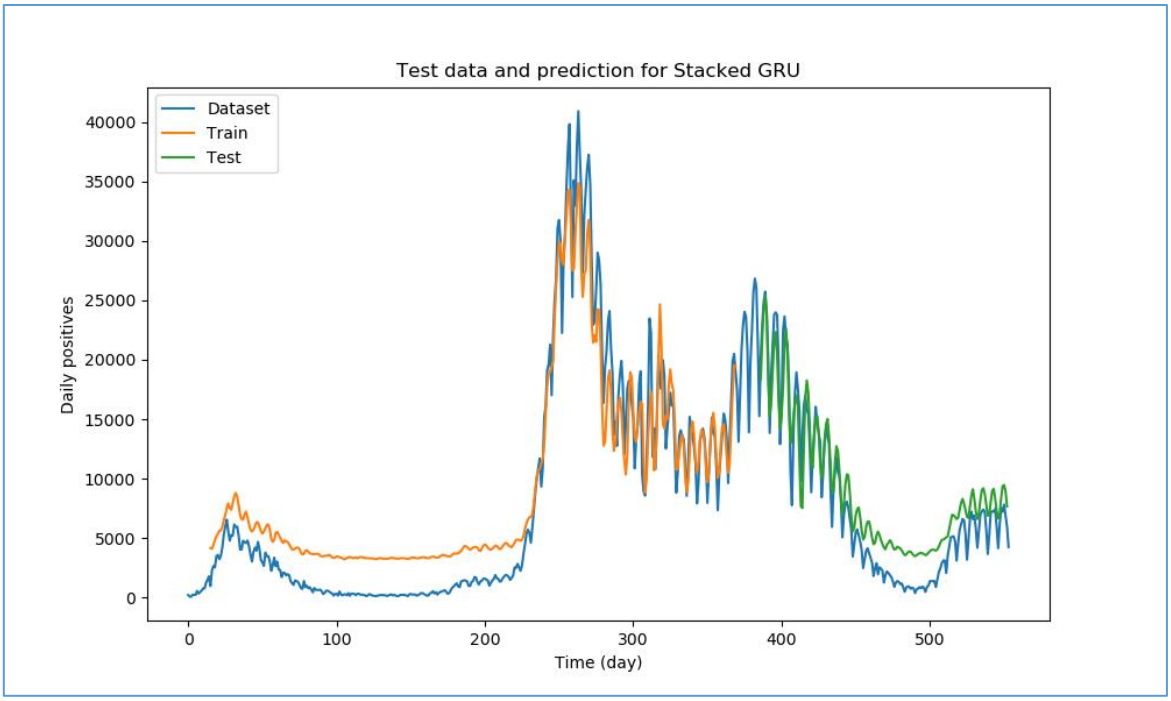
```
Vanilla GRU:
Mean Absolute Error: 1199.8497
Root Mean Square Error: 1573.1127
```

7. Stacked GRU

Se al modello precedente viene aggiunto un altro layer nascosto, otteniamo una *Stacked GRU*. Le considerazioni esposte sull'omologo LSTM sono valide anche qui.
Dopo la 34° iterazione, notiamo come il modello sia affetto da *overfitting*, fenomeno già osservato nella *Stacked LSTM*:



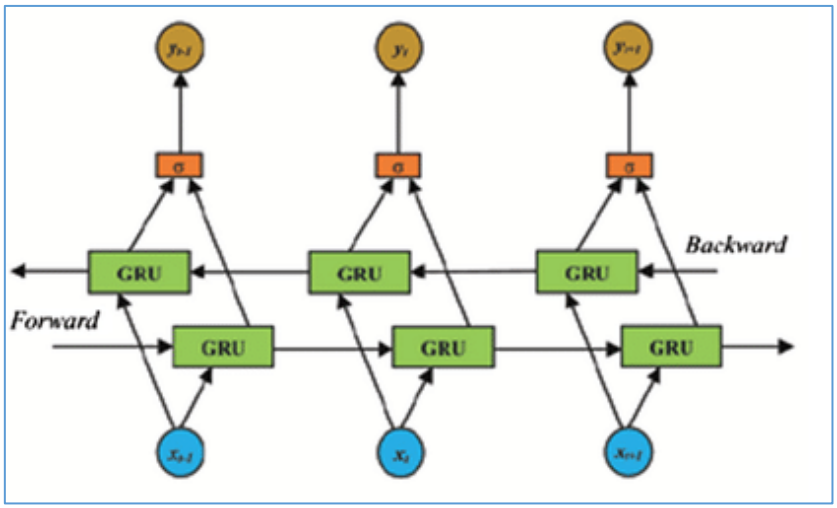
Le previsioni, dunque, non saranno precise e l'errore sulle previsioni sarà maggiore della sua versione *Vanilla*:



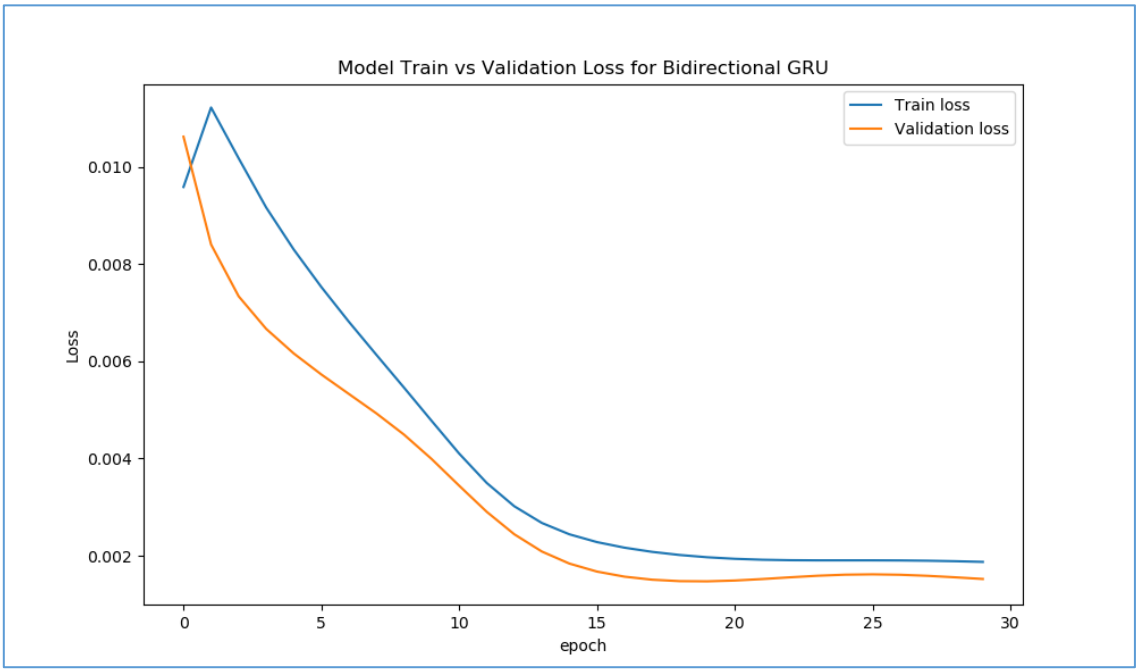
```
Stacked GRU:
Mean Absolute Error: 2058.7391
Root Mean Square Error: 2278.0517
```

8. Bidirectional GRU

Come ultimo modello univariato, ovvero quei modelli in cui la predizione è effettuata a partire da una singola variabile, consideriamo la *Bidirectional GRU*, il cui schema è:

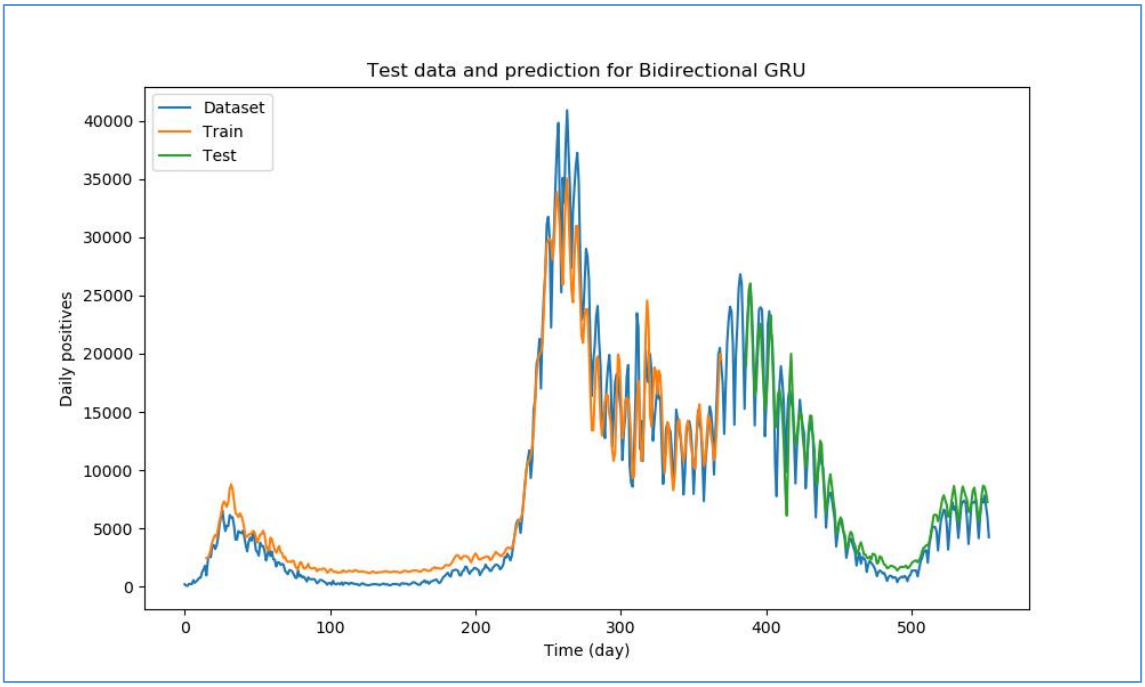


Dal grafico delle *losses,* il modello sembra comportarsi bene:



Già dopo circa 15 iterazioni, il modello si assesta su valori molto bassi.

Graficando come già visto in precedenza, otteniamo:



Gli errori commessi sono pari a:

```
Bidirectional GRU:
Mean Absolute Error: 1188.4613
Root Mean Square Error: 1593.0366
```

I vantaggi di utilizzare una rete bidirezionali vengono confermati anche qui: l'errore che otteniamo è il più basso tra quelli visti fin ora.

9. Multivariate LSTM

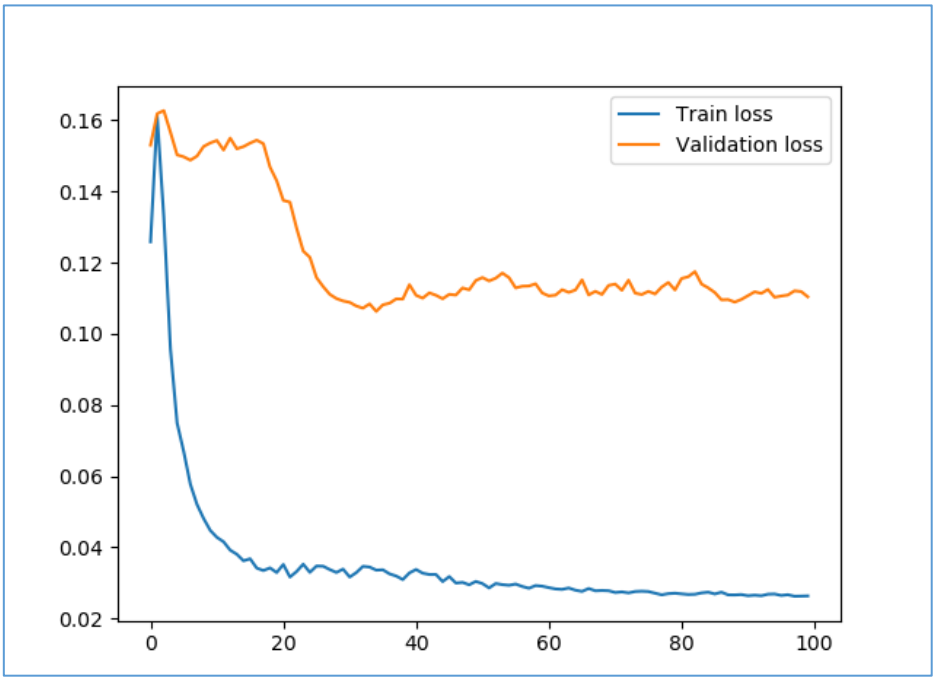
Reti neurali come la LSTM si adattano bene per studiare problemi con molteplici input. Per fare ciò, è necessaria un'elaborazione dei dati differente da quanto visto nel primo paragrafo: utilizzando lo stesso dataset fornito dalla Protezione Civile, notiamo come alcune colonne siano ininfluenti o con dati mancanti e dunque questi dati sono stati scartati (funzione *drop*).

La funzione che consente di lavorare con la rete neurale, dopo aver normalizzato i dati come in precedenza, crea un nuovo array con un numero di colonne pari al doppio del numero di variabili scelte per la predizione: all'interno dell'array infatti, sono presenti le sette variabili con le corrispettive sette future predizioni.
Poiché a noi interessa le predizioni della sola voce *nuovi_positivi*, le altre colonne (*ricoverati_con_sintomi, terapia_intensiva, totale_ospedalizzati, totale_positivi, deceduti* e *tamponi*) sono state ignorate.

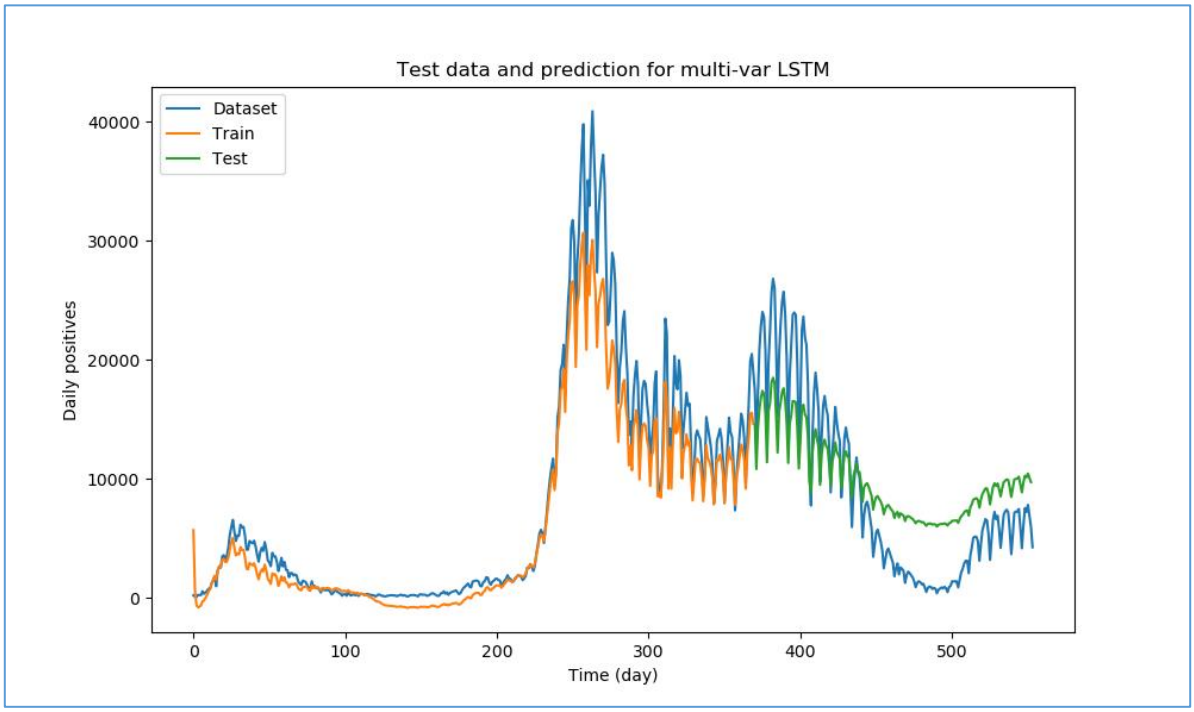
	var1(t-1)	var2(t-1)	var3(t-1)	var4(t-1)	var5(t-1)	var6(t-1)	var7(t-1)	var7(t)
1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.003503	0.000367
2	0.000376	0.002227	0.000599	0.000051	0.000112	0.000023	0.000367	0.000000
3	0.000780	0.002474	0.000964	0.000063	0.000204	0.000039	0.000000	0.004213
4	0.004249	0.007422	0.004612	0.000092	0.000455	0.000077	0.004213	0.003919
5	0.007053	0.009401	0.007348	0.000136	0.000745	0.000108	0.003919	0.003968

2 - Prime 5 righe dell'array creato dalla funzione; la colonna var7(t) è quella dei nuovi positivi

Anche qui, i grafici delle curve di apprendimento sono visualizzati per poter studiare il comportamento del modello: purtroppo, sebbene entrambe le curve si stabilizzino, il gap presente tra le due curve indica, con alta probabilità, un *training dataset* poco rappresentativo, dettato dalla scarsità di dati rispetto dal *validation dataset*.



Di conseguenza, le predizioni non saranno accurate e l'errore quadratico medio sarà molto alto:




```
Test RMSE: 4146.699
```

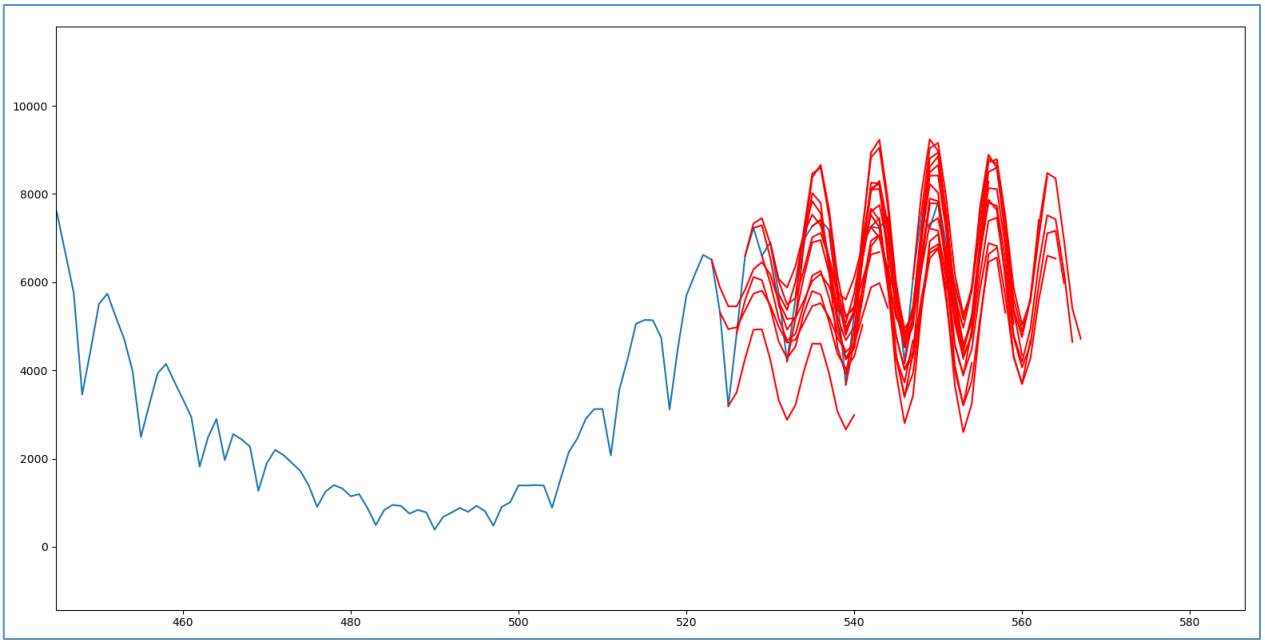
I risultati esposti non migliorano impiegando l'*EarlyStopping* o aumentando il numero di *epochs*.

10. Multi-step LSTM

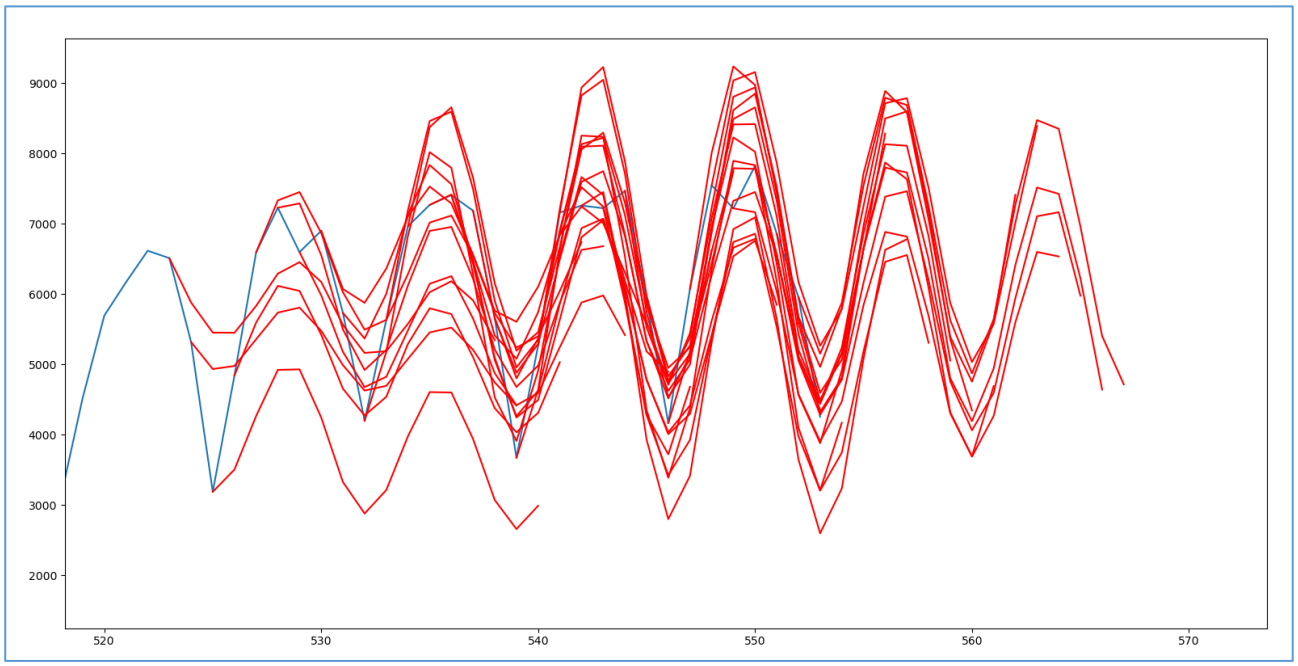
Oltre ad imparare lunghe sequenze di dati, *LSTM* è in grado di effettuare previsioni multi-step in un colpo solo (*one-shot multi-step forecast*).
Anche qui si sottolinei il fatto che la preparazione dei dati è fondamentale per avere buone performance: allo stesso modo del precedente paragrafo, la serie temporale viene convertita in un *supervised learning problem*; inoltre, per poter lavorare meglio con i dati, questi sono stati normalizzati e resi stazionari facendone la differenza.

Il *fitting* è stato fatto usando una *Bidirectional LSTM* in quanto si è dimostrata quella che meglio si adatta al nostro problema; i parametri ottimi sono già stati discussi ma sono stati aggiunti quanti giorni si desidera prevedere e da dove cominciare la predizione.
Anche qui l'*EarlyStopping* non si è rilevato soddisfacente, quindi il modello effettuerà il numero di iterazioni impostato dall'utente. Con *n_epochs = 200* i risultati sono:

```
t+1 RMSE: 789.056427
t+2 RMSE: 975.823004
t+3 RMSE: 912.020229
t+4 RMSE: 964.866943
t+5 RMSE: 1240.448042
t+6 RMSE: 1238.070919
t+7 RMSE: 1102.801443
t+8 RMSE: 1297.932263
t+9 RMSE: 1393.561394
t+10 RMSE: 1404.193590
t+11 RMSE: 1557.001755
t+12 RMSE: 1826.735764
t+13 RMSE: 1833.317460
t+14 RMSE: 1684.927898
t+15 RMSE: 1734.434259
```



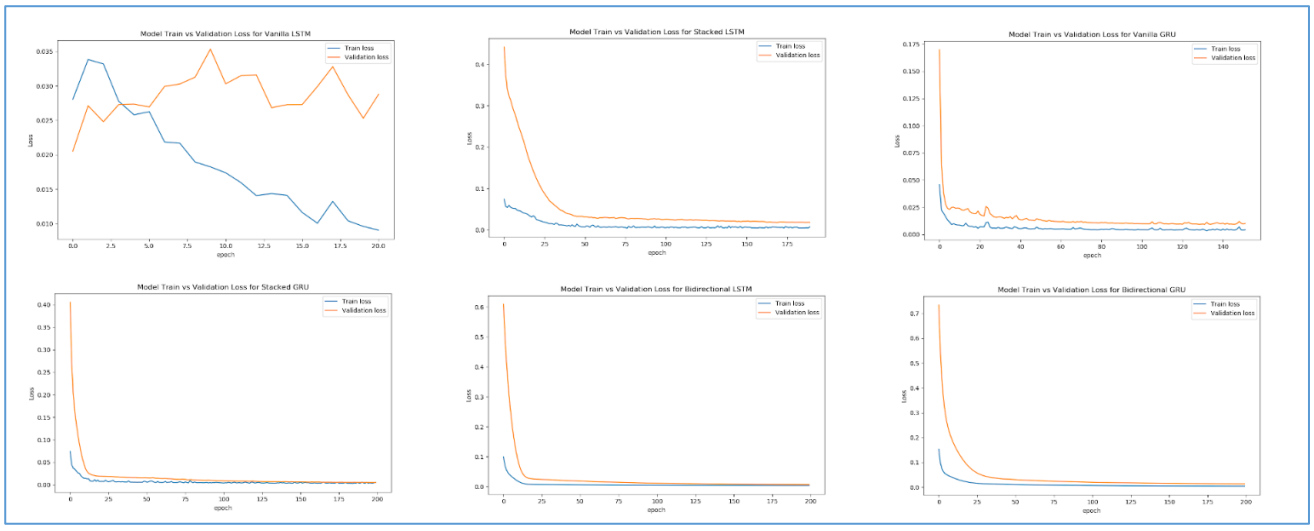
Nel dettaglio:



Gli errori commessi sono in linea, con i modelli precedentemente analizzati, se non più bassi; inoltre, osservando i grafici delle predizioni (il secondo è un ingrandimento del primo) l'andamento dei contagi è rispettato: il modello ha quindi colto la stagionalità presente nel dataset.

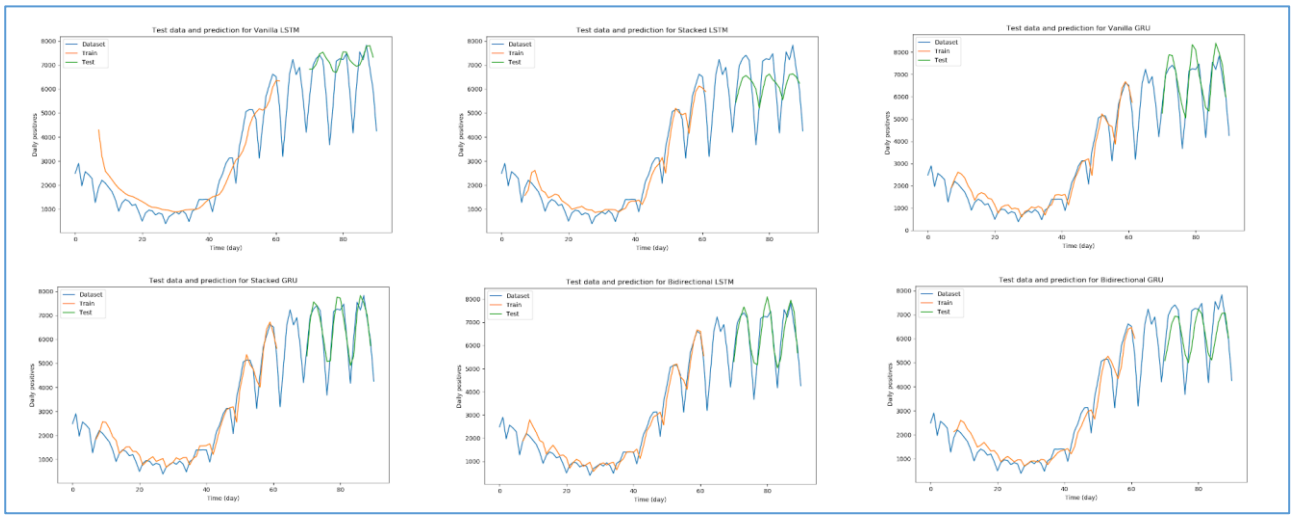
11. Confronto modelli esposti con dataset ridotto

Quanto segue è l'impiego delle stesse reti neurali con lo stesso dataset ma in versione ridotta: i dati ora riguardano il solo periodo estivo, da giugno ad agosto; i valori degli iperparametri sono rimasti uguali, così come è stato mantenuto l'utilizzo dell'*EarlyStopping*.
Raggruppando tutti i modelli in un unico codice, le prestazioni sono le seguenti:



Tutte le RNN sembrano avere buone prestazioni tranne la *Vanilla LSTM*, dove, anche togliendo l'*EarlyStopping*, è afflitta da un pesante *overfitting*.

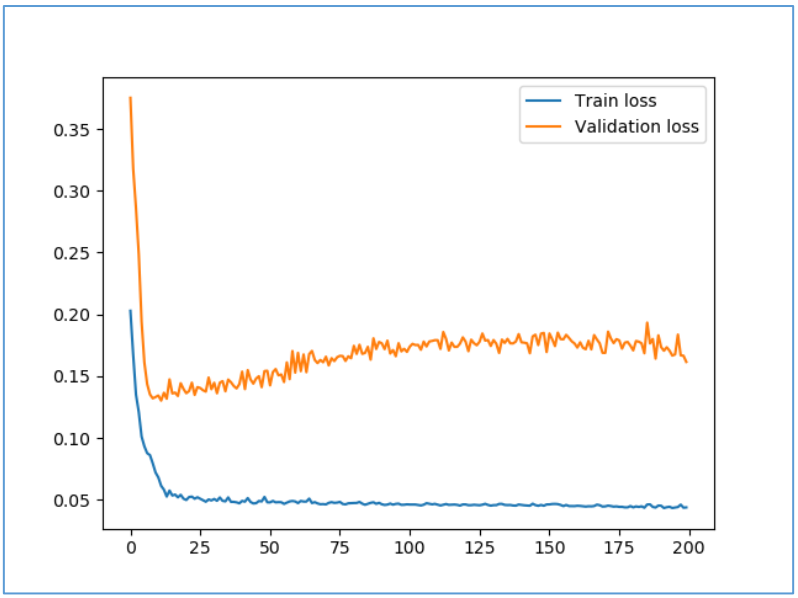
I grafici delle predizioni sono:

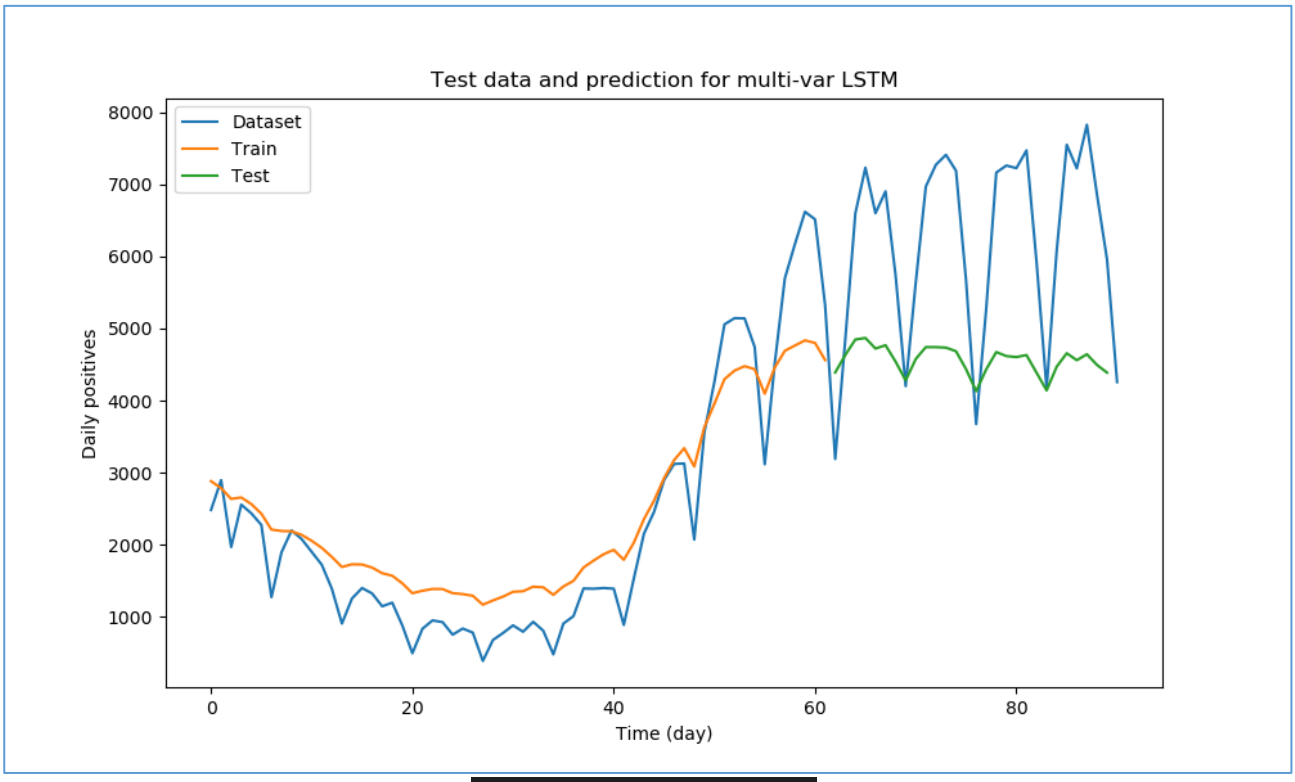


Il primo modello non è assolutamente rappresentativo e ciò era intuibile dall'immagine precedente; gli altri invece, sono in grado di capire l'andamento del problema, seppur non riescono a seguire i picchi presenti.
Ci aspettiamo dunque di avere un errore abbastanza alto per la *Vanilla LSTM*, mentre per le altre 5 reti neurali, l'*RMSE* dovrebbe essere contenuto:

```
Vanilla LSTM:
Mean Absolute Error: 879.2019
Root Mean Square Error: 1270.6018

Stacked LSTM:
Mean Absolute Error: 834.9478
Root Mean Square Error: 998.8062

Vanilla GRU:
Mean Absolute Error: 570.8257
Root Mean Square Error: 744.8046

Stacked GRU:
Mean Absolute Error: 417.5372
Root Mean Square Error: 519.0924

Bidirectional LSTM:
Mean Absolute Error: 441.9181
Root Mean Square Error: 586.6647

Bidirectional GRU:
Mean Absolute Error: 696.3651
Root Mean Square Error: 858.6186
```

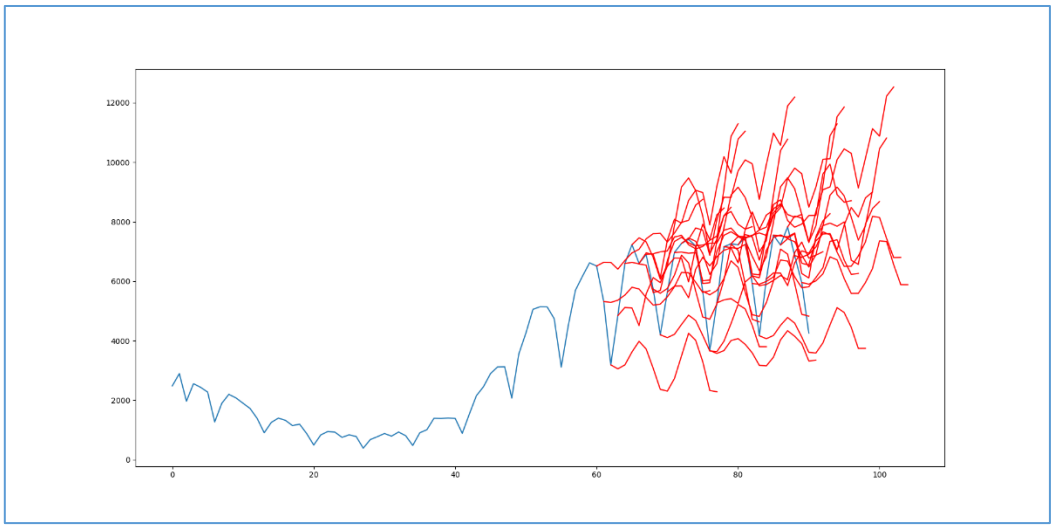
Con lo stesso nuovo dataset, abbiamo analizzato una *LSTM* multi-variabile e una *Bidirectional LSTM* multi-step: purtroppo nel caso multi-var, la rete è afflitta da *overfitting*, pertanto i risultati ottenuti sono di scarso valore:






```
Test RMSE: 929.363
```

Allo stesso modo, anche il modello multi-step non è di grande interesse: qui la previsione sull'andamento dei contagi è di gran lunga peggiore rispetto al dataset precedente.




```
t+1 RMSE: 1111.600228
t+2 RMSE: 1742.931983
t+3 RMSE: 1781.069573
t+4 RMSE: 1605.069457
t+5 RMSE: 1453.193490
t+6 RMSE: 1470.962522
t+7 RMSE: 1184.187134
t+8 RMSE: 1537.982191
t+9 RMSE: 2286.809982
t+10 RMSE: 2316.466542
t+11 RMSE: 2050.474390
t+12 RMSE: 1963.717776
t+13 RMSE: 1931.606528
t+14 RMSE: 1945.930234
t+15 RMSE: 2290.353056
```

Il motivo di questa discrepanza è dettato probabilmente dalla scarsità di dati in ingresso alla rete.

12. Applicazione RNN su dataset delle vaccinazioni

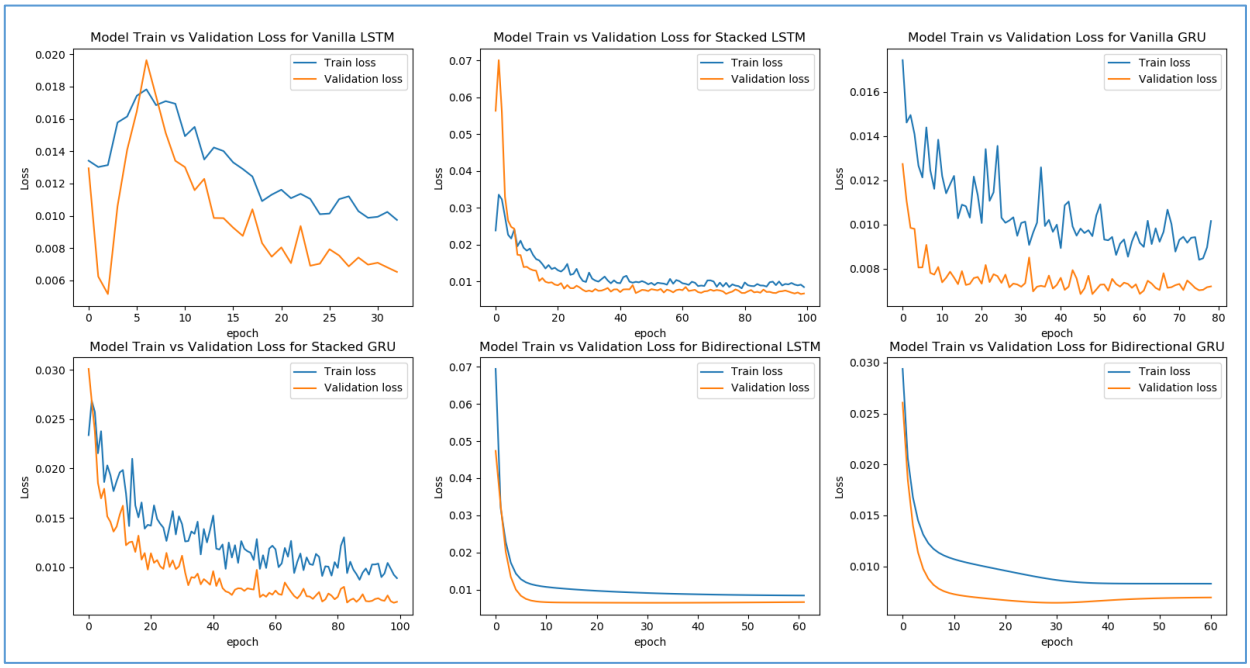
Per concludere questo lavoro, ci siamo chiesti come le reti neurali fin ora impiegate si comportino con un dataset totalmente diverso; pertanto, è stato utilizzato quello delle vaccinazioni in Italia. Dopo aver reperito il file csv **(**[**https://github.com/owid/covid-19-data/blob/master/public/data/vaccinations/vaccinations.csv**](https://github.com/owid/covid-19-data/blob/master/public/data/vaccinations/vaccinations.csv)**)** e studiato il suo contenuto, l'analisi si è incentrata sull'andamento delle vaccinazioni giornaliere; il dataset espone anche il numero di vaccinazioni giornaliere con un *rolling* su 7 giorni per quei paesi che non forniscono i dati di giorno in giorno ma per il caso italiano, non è stato necessario sfruttare ciò.

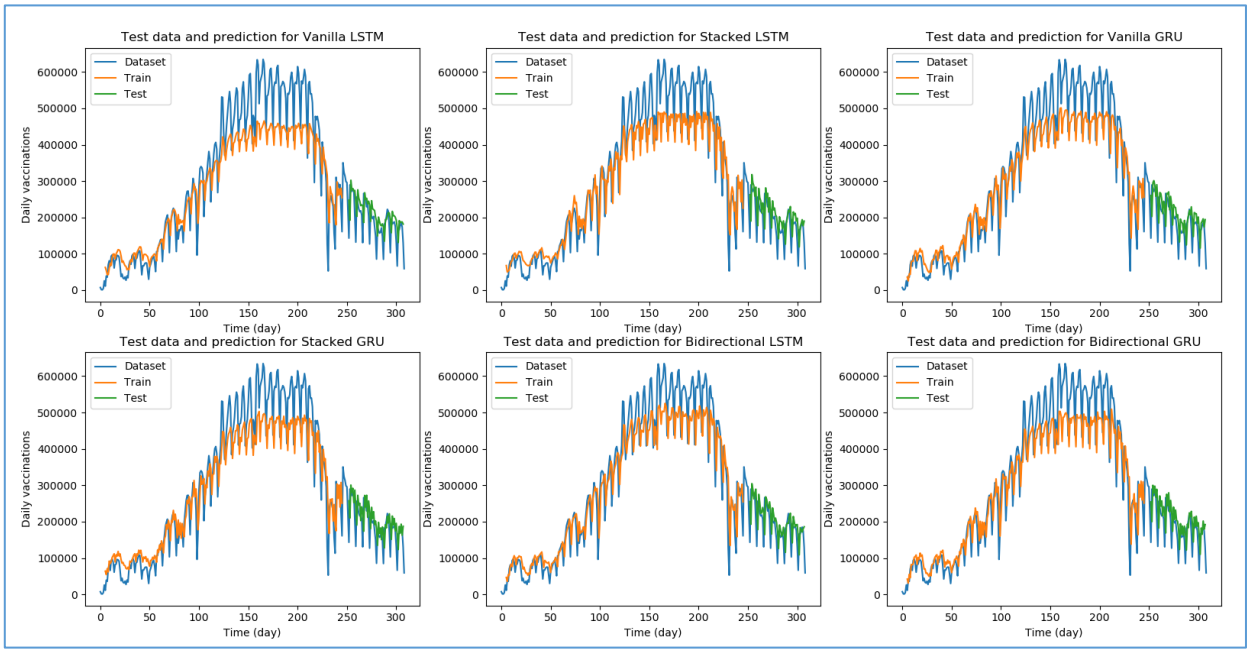
Passando al codice, lo split dei dati in *training* e *testing sets* e le funzioni per allenare le reti sono le stesse, a meno di alcuni valori: per esempio, essendo il dataset di dimensioni minori rispetto a prima, la divisione tra *training* e *testing sets* è pari a 80/20. Inoltre, il valore di *lookback* è pari a 5 e l' iperparametro *n_neurons* per ciascuna rete sono:

- Vanilla LSTM: 16
- Stacked LSTM: 4
- BiDir LSTM: 1

- Vanilla GRU: 4
- Stacked GRU: 2
- BiDir GRU: 1

Altra leggera differenza riguarda i grafici: qui è stato deciso di usare la funzione *subplot* per avere in un'unica schermata tutte le *losses* e le predizioni:



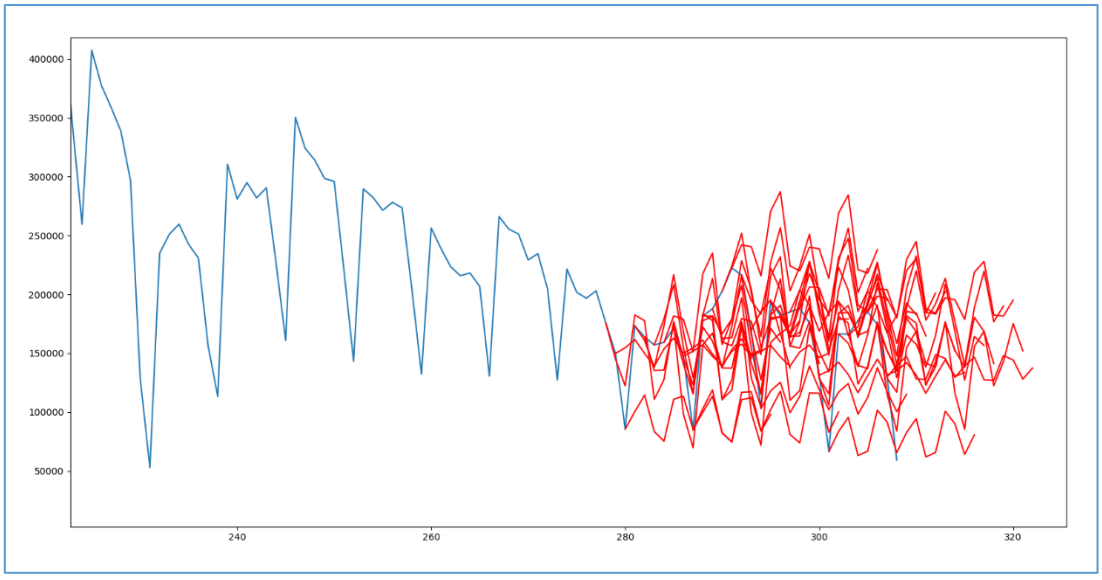


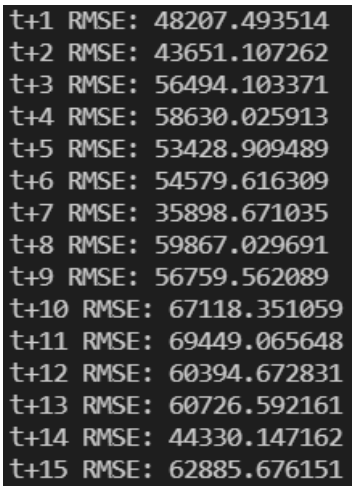
Anche in questo caso, si è ricorso all'utilizzo dell'*EarlyStopping*; nelle due reti *Vanilla*, i modelli non hanno prestazioni sufficienti, mentre nelle *Stacked*, individuiamo un leggero *overfitting*.
A conseguenza di quanto esposto, i grafici delle predizioni sembrano seguire bene gli andamenti generali ma non sono performanti nell'inseguire i picchi, influenzando di molto gli errori:

```
Vanilla LSTM:
Mean Absolute Error: 39052.9411
Root Mean Square Error: 50708.6985

Stacked LSTM:
Mean Absolute Error: 41639.4351
Root Mean Square Error: 52106.7840

Vanilla GRU:
Mean Absolute Error: 42766.2500
Root Mean Square Error: 53755.6543

Stacked GRU:
Mean Absolute Error: 40614.8670
Root Mean Square Error: 51066.4743

Bidirectional LSTM:
Mean Absolute Error: 39376.2408
Root Mean Square Error: 51556.7530

Bidirectional GRU:
Mean Absolute Error: 41441.0262
Root Mean Square Error: 52791.1825
```

Sebbene gli errori siano alti, è necessario rapportarli sui valori di picco: si tenga presente che spesso il numero di vaccinazioni superava il mezzo milione di dosi giornaliere e dunque, in percentuale, l'errore si rivela essere inferiore al 7%.

Poiché nel dataset non sono stati trovati altri dati rilevanti, la predizione multi-variabile è stata scartata; è stata però mantenuta la multi-step, implementata questa volta con una *Bidirectional GRU*. Le funzioni all'interno del codice sono invariate, così come i parametri.





I risultati ottenuti sono simili a quanto visto con gli altri modelli multi-step: la stagionalità del modello viene colta dalla rete e l'andamento previsto è coerente con la storia pregressa.

13. Conclusioni

Terminate le analisi su questo nuovo dataset, è necessario fare qualche commento: abbiamo notato come l'utilizzo di differenti reti neurali portino a risultati anche molto diversi tra loro; aspetto fondamentale nel corso di questa trattazione è stato dunque la scelta di parametri ottimi per supplire alle mancanze strutturali di certi modelli.
Sia i grafici delle *losses* sia quelli delle predizioni hanno rispettato le nostre attese; passando all'analisi degli errori, inizialmente si pensava di aver commesso delle imprecisioni, in quanto non ci aspettavamo numeri così importanti.
Tuttavia, è stato sufficiente notare come in percentuale, gli errori erano circa pari al 3% per il dataset sui contagi e il 7% per quello dei vaccini: risultato, dunque, che ha confermato la bontà del nostro lavoro.
È stato proprio l'ultimo dataset preso in esame la nostra prova definitiva: possiamo affermare con sicurezza che quanto svolto si adatti bene anche su dati diversi.

Con questo lavoro si chiude un ciclo iniziato lo scorso anno con l'analisi e previsione di serie temporali, sempre incentrato sui contagi da Covid-19 **(https://github.com/mattepasto/CPS_Covid):** abbiamo ritenuto di nostro grande interesse applicare quanto affrontato a lezione su un tema così delicato e di attualità, opportunità che raramente è possibile cogliere durante gli studi.
Nello specifico, quanto trattato durante questa relazione, ha sottolineato l'importanza di sapere leggere con attenzione i dati e studiare quanto proposto dalle librerie impiegate.
I risultati raggiunti possono considerarsi soddisfacenti, a patto che vengano interpretati con giudizio e che siano preceduti da attente riflessioni su di essi: confidiamo che ciò sia stato fatto grazie alla stesura di queste righe.