Multichannel Convolutional Neural Networks for Multi-Step Time Series Forecasting

Javad Samieenia & Andre Surprenant

December 25, 2018

Abstract

We implement a one dimensional Convolutional Neural Network (CNN) to forecast time series data. As a metric for evaluation, we compare the performance of our model against a simple Feed-forward netwok (FF). We train and test our models on both simulated and real datasets.

1 Introduction

In recent years, Artificial Neural Networks (ANN) have made head-ways in terms of computational efficiency and applicability. ANNs have performed remarkably well on a variety of tasks, including computer vision, speech recognition, machine translation, social network filtering, and playing board games. However, research still needs to be done on how well ANNs might perform on predicting highly volatile time series data.

The most common and intuitive approach to forecasting with an ANN is to use a Recurrent Neural Network (RNN). Recurrent neurons, unlike feed forward neurons, receive input, x_t, as well as the output form the previous step, y_{t-1}. The most popular architecture which uses the recurrent structure is called LSTM for Long-Short-Term-Memory. It has been used, with great success, to predict text and speech patterns. Additionally, it has been used to for financial time series forecasting with mixed results. In order for the LSTM architecture to perform well it needs to acquire a good domain knowledge (which is difficult with economic data since it tends to be highly volatile and prone to structural breaks) and adequately set many hyper-parameters. Both of these conditions require very large datasets and, consequently, powerful computers to satisfy. However, most economic time series tend to be quite small, especially for academic research. For this reason, we have opted to use a one dimensional convolutional neural network (CNN) to perform the same task. CNNs have three features which make them very suitable for time series applications: sparse interactions, parameter sharing, and equivariant representations.

We train and test our network, for both univariate and multivariate cases, using simulated data as well as actual data concerning pollution levels in Beijing[1].

In order to assess the performance of the convolutional network, we wanted to create a simple benchmark ANN that could perform the same task. Our choice was a feedforward (FF) ANN that that reads input data sequentially in the same manner as the CNN. We then evaluate the performance of the CNN by comparing its prediction errors to those of a simple FF network using the same data.

2 Keras and TensorFlow

To build our network we use a high level API called Keras with a TensorFlow backend. Keras is widely used due to its approachability and relative efficiency. Keras simplifies our code greatly making it easier to follow so the reader can focus on the model implementation. Keras is also very well optimized for TensorFlow so it does not add to much too the computational burden. In fact, Keras is the official high level API for TensorFlow.

Within Keras, we use the `sequential` model to build our network. We start by listing each layer starting with the inputs and ending with the output. We then compile the model, specifying an optimizer and a loss function. Once the model is compiled we train it on our training data using the fit function; we also need to specify the number the batch size and number of epochs.

[1] Liang et al. (2015)

Here is an example of how to create a simple feedforward network with Keras:

```
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(1, activation='sigmoid'))
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(2, size=(1000, 1))

# Train the model, iterating on the data in batches of 32 samples
model.fit(data, labels, epochs=10, batch_size=32)
```

As you can see the model creation can be seperated into three distinct phases: The model building phase uses the `sequential()` function to initialize a model. We can then add layer and activation functions sequentially, starting with the inputs and ending with the outputs, by using the `model.add()` function. Once the model has been created we move to the compile phase where we can specify a loss function and an optimizer in our `model.compile()` function. Now that the model is compiled, we train it by passing our data into the `model.fit()` function along with a specified batch size and number of epochs.

3 Benchmark Model

In order to assess the performance of our convolutional network we need to compare it to a simpler point of reference. We want a model that is capable of both 1-step and multi-step forecasting and does not contain many hyperparameters. The optimal choice for this task is the simple feedforward network displayed in figure 1. When it comes to predicting time distributed data, one would intuit that a simple feed-forward model would not be able to forecast very well since, implicit in its structure is an invariance to shuffling the input data. However, The data treatment method we employ involves reshaping our data so that it is oriented towards making predictions m steps into the future given n lags for a particular time index t. With this alteration we become capable of forecasting with comparable accuracy to the more common RNN architectures.

In order to keep things simple, we limit this model to the univariate case. So, our inputs consist only of $\{y_{t-n}, y_{t-n+1}, ..., y_t\}$. Where t is the time index and n is the specified number of previous time steps we wish to take into account for the next prediction. Our outputs, $\{\hat{y}_{t+1}, \hat{y}_{t+2}, ..., \hat{y}_{t+m}\}$, try to predict the next m observations indexed by time. For example, the training data is reshaped as follows:

consider the case where $n = m = 7$ and let `dt` be a data point at time `t`. Our data takes the following shape after rearranging:

```
Input, Output
[d01, d02, d03, d04, d05, d06, d07],  [d08, d09, d10, d11, d12, d13, d14]
[d02, d03, d04, d05, d06, d07, d08],  [d09, d10, d11, d12, d13, d14, d15]
...
```

We build the model using leaky ReLu units to reduce the vanishing gradient problem and batch normalization to increase learning speed and add some regularization to the model.

Here is the code we used to implement this model:

```
def split_dataset(data): # data is np.array of size T*n_features=1
    T = data.shape[0]
    split_ratio = 0.7
    t_split = np.int(np.floor(split_ratio * T))
    # split into train and test
    train, test = data[0: t_split], data[t_split:]
    return train, test

# convert data (train or test) into inputs (X) and outputs (y)
def to_supervised(data, n_input, n_out):
    # X and y have shapes of (#samples, n_input) and (#samples, n_out) repectively.

    X = []
    y = []
    T_tr = data.shape[0]
```

```
    for i in range(T_tr - n_input - n_out + 1):
        X.append(data[i: i + n_input])
        y.append(data[i + n_input: i + n_input + n_out])
    X = np.array(X)
    y = np.array(y)
    return X, y



def build_model(train, n_input, n_out):
    # prepare data
    train_x, train_y = to_supervised(train, n_input, n_out)
    # create model
    model = Sequential()
    model.add(BatchNormalization())

    model.add(Dense(100, input_dim= n_input))
    model.add(LeakyReLU(alpha=0.1))
    model.add(BatchNormalization())


    model.add(Dense(n_out,))
    model.compile(loss='mean_squared_error', optimizer='RMSprop', metrics=['mse'])
    # Fit the model
    model.fit(train_x, train_y, epochs=100, batch_size=20)

    return model, train_x, train_y
```

As you can see, we start by splitting the data into training and testing sets. The `to_supervised()` function then arranges our data in a manner that can infer salient features from sequences. We then initialize our model using the `built_model()` function. It starts with a batch normalization layer, followed by a densely connected layers with leaky ReLu activation functions. Then we re-normalize the output from the first layer and pass it through our densely connected layer. We then compile the model specifying RMSprop as our optimizer and MSE as our loss function. Once the model is compiled we train it by running the `fit()` function on our training data.

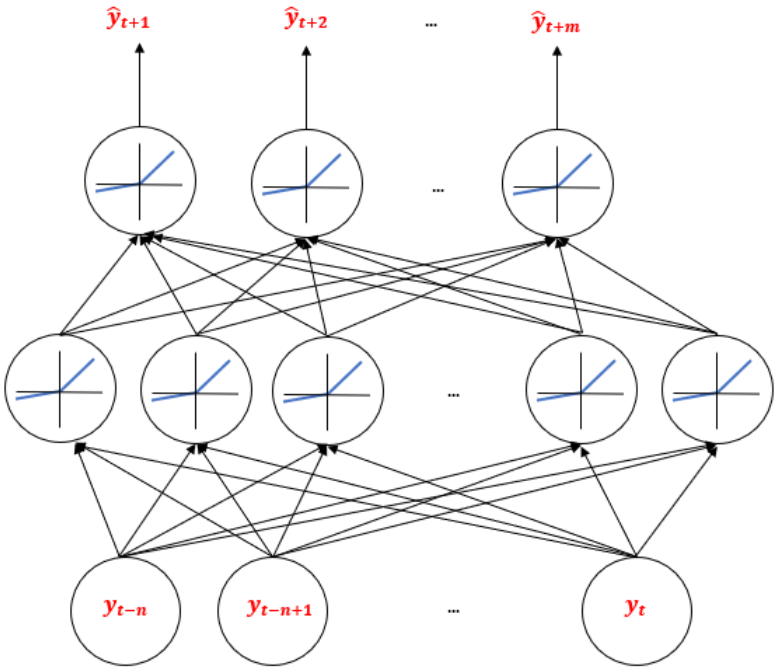

Figure 1: FeedForward Network Architecture.

4 Convolutional Network

4.1 Motivation

A convolutional neural network can be defined simply as a neural network that uses convolution in place of general matrix multiplication in at least one of its layers. The operation of discrete convolution most often implemented in CNNs is defined as follows:

$$s(t) = (X * \omega)(t) = \sum_{i=-\infty}^{\infty} X(i)\omega(t-i)$$

where t is time, x is our input matrix and ω is our kernal, whose dimensions must be specified as a model hyper-parameter. The output of this function, $s(t)$ is often referred to as a feature map.

One can think of each neuron in the second neuron in a convolutional layer as only being connected only to neurons located within a small time interval specified by our kernal. This structure allows our model to discern salient short-term features in the first convolutional layer and longer term features in the second layer. This aspect of CNNs is often referred to as sparse connectivity.

The next major motivation for using a CNN for forecasting its its use of parameter sharing. The term comes from the fact that the weights of several inputs are tied to one another. This happens since each member of the kernel is used at every position of the input. As a consequence of parameter sharing, our model has a reduced tendency to overfit the training data. Furthermore, CNNs are equivariant to temporal representations. This means that convolution produces a sort of timeline that shows when different features appear in the input. If we move an event later in time in the input, the exact same representation of it will appear in the output, just later.

Lastly, CNNs support multivariate data, and can directly output a vector for multi-step forecasting as well.

4.2 Multivariate Implementation

To build our multivariate CNN, we start by specifying the hyperparameter `n_input`, the number of input observations on which we base our forecast, and the hyperparameter `n_output`, the number of observations forecast.
Next, we split the whole dataset into a training set and a test set.

```
n_input = 20
n_out = 1

def split_dataset(data): # data is np.array of size T*K. It can be set as dataset.values
    T = data.shape[0]
    split_ratio = 0.7
    t_split = np.int(np.floor(split_ratio * T))
    # split into train and test
    train, test = data[0: t_split], data[t_split:]
    return train, test
```

After splitting the data into training and test sets we perform the same reshaping process that we did for the simple FF network.

```

# convert train into inputs (X) and outputs (y)
def to_supervised(train, n_input, n_out):
    X = []
    y = []
    T_tr = train.shape[0]

    for i in range(T_tr - n_input - n_out + 1):
        X.append(train[i: i + n_input])
        y.append(train[i + n_input: i + n_input + n_out])
    X = np.array(X)
    y = np.array(y)

    # X nad y have shapes of (#samples, n_input, n_feature) and (#samples, n_out,
    n_features) repectively.
    # y is always univariate (even if n_features>1), we should reshape it to (#samples,
    n_out)

```

```
    y = y[:,:, 0]

    return X, y
```

Now that our data is properly organized, we can build our CNN with the architecture outlined in figure 2.

The multi-step time series forecasting problem is an autoregression problem. For multivariate/multichannel input we use $\{y_{t-n}, ..., y_t\}$ to refer to the variable of interest and $\{X_{t-n}, ..., X_t\}$ to denote our exogenous control variables. That means it is likely best modeled where the next m days is some function of observations at prior time steps, $y_t = f(y_{t-1}, y_{t-2}, \ldots, X_{t-1}, X_{t-2,...})$. This means that a small model is required.

Our model uses two convolutional layers with 32 filter maps and a kernal size of 3 followed by a pooling layer, then another convolutional layer with 16 filters and pooling. The output of the final pooling layer is then flattened and represented as one long vector. This is then interpreted by a fully connected layer with 100 neurons before the output layer predicts the next `n_out` (or m) days in the sequence denoted by $\{\hat{y}_{t+1}, ..., \hat{y}_{t+m}\}$.

We will use the mean squared error loss function since it matches with our chosen error metric. We will use the efficient Adam implementation of stochastic gradient descent and fit the model for 70 epochs with a batch size of 16.

```
def build_model(train, n_input, n_out):
    # prepare data
    train_x, train_y = to_supervised(train, n_input, n_out)
    # define parameters
    verbose, epochs, batch_size = 1, 70, 16
    n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.
    shape[1]
    # define model
    model = Sequential()
    model.add(Conv1D(filters=32, kernel_size=3, activation='relu', input_shape=(
    n_timesteps,n_features)))
    model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Conv1D(filters=16, kernel_size=3, activation='relu'))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(100, activation='relu'))
    model.add(Dense(n_outputs))
    model.compile(loss='mse', optimizer='adam')
    # fit network
    model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
    return model, train_x, train_y
```

4.3 univariate implimentation

In order to compare the results of a CNN to our benchmark model, we need to construct a simpler univariate CNN. The arachitecture for this model is similar to the multivariate one but, trial and error revealed to us that the following model would perform better for a simple autoregression:

```
def build_model(train, n_input, n_out):
    # prepare data_ univarite case
    train_x, train_y = to_supervised(train, n_input, n_out)
    # define parameters
    verbose, epochs, batch_size = 1, 20, 4
    n_timesteps, n_features, n_outputs = train_x.shape[1], 1, train_y.shape[1] #
    n_features is 1 for a univariate case
    # define model
    model = Sequential()
    model.add(Conv1D(filters=16, kernel_size=4, activation='relu', input_shape=(
    n_timesteps,n_features)))
    model.add(MaxPooling1D(pool_size=2))
    model.add(Flatten())
    model.add(Dense(10, activation='relu'))
    model.add(Dense(n_outputs))
    model.compile(loss='mse', optimizer='adam')
    # fit network
    model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
    return model, train_x, train_y

```

4.4 Some Comments on Parameter Tuning

The convolutional network contains many hyperparameters which must be set according to our discretion. Due to computational limitations, we could not optimize this model to its full potential. However, the values which we did end up choosing represent, for the most part, "rule of thumb" values within the field as well as some guess work. For this reason, some of the results in following sections may seem to favour the simple FF network, which contains very few hyperparameters, a lack of hyperparameter optimization could be at fault.

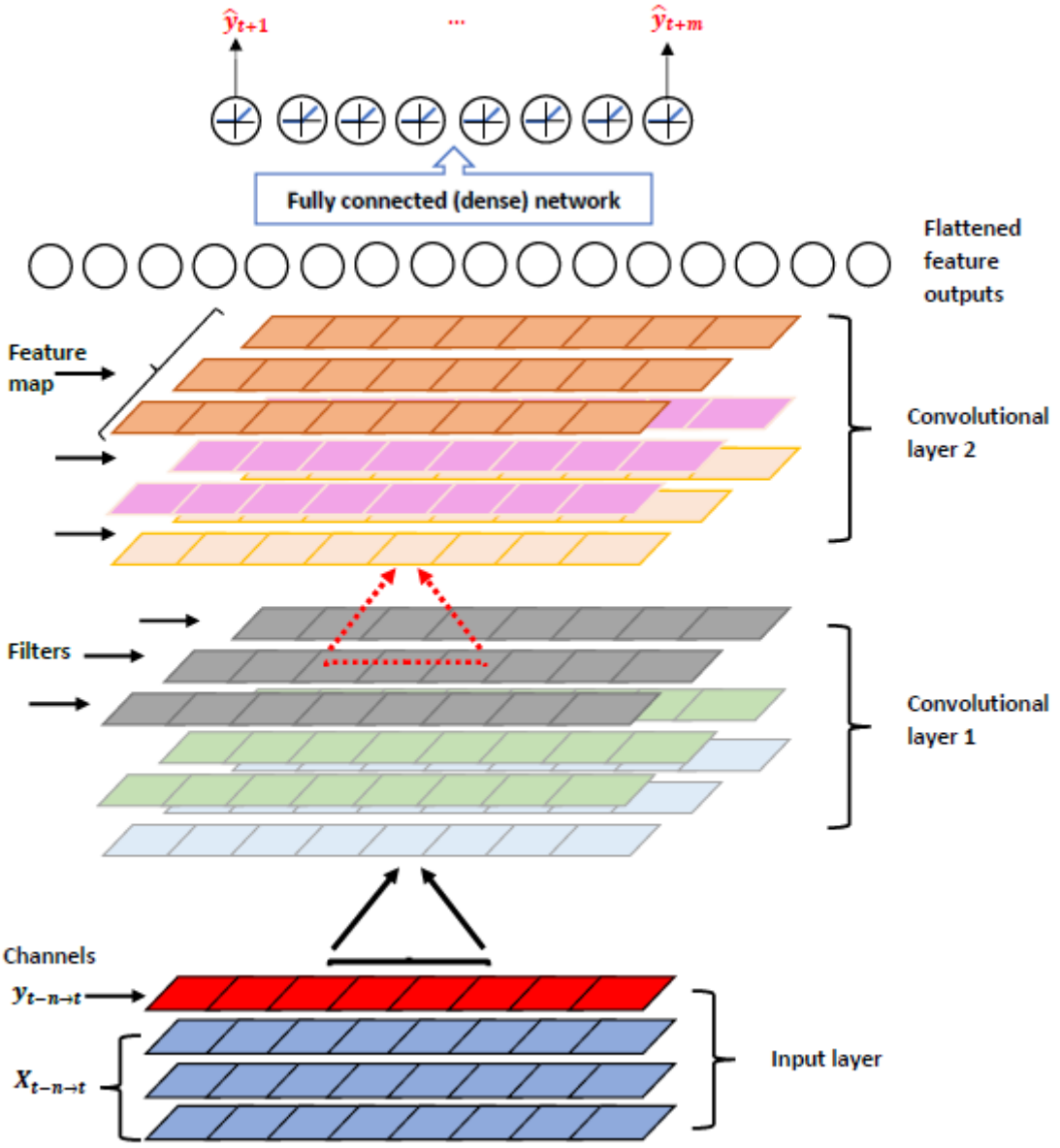

Figure 2: CNN Architecture.

5 Data

We test our models on two sets of data: a simulated one, and a real database. The simulated database consists of three time series of length 1000, TS1, TS2, and TS3.
Data generating process (DGP) of TS1 follows a *stationary* AR(5) (autoregressive process of order 5) of

the form:

$$y_t = 0.2y_{t-1} + 0.3y_{t-2} - 0.4y_{t-3} + 0.1y_{t-4} + 0.8y_{t-5} + \epsilon_t,$$

where ϵ_t is distributed as i.i.d $N(0, 1)$.
TS2 is also a mean-reverting (i.e. stationary) time series in which the cyclicality is pronounced. Its DGP is as follows:

$$y_t = 3\sin\left(\frac{t}{20}\right) + \epsilon_t, \quad \epsilon_t \sim i.i.dN(0, 1).$$

Finally, TS3 is also cyclical but not stationary anymore:

$$y_t = \frac{t}{5} sin\left(\frac{t}{20}\right) + \epsilon_t, \quad \epsilon_t \sim i.i.dN(0, \sqrt{t}).$$

Our main criterion to consider these DGPs is to possess a versatile set of sequential data, with diverse properties, in order to test our models in myriad of scenarios. After generating these series, they are stacked to each other and saved into a dataframe, and then into a csv file, which later will be read by our models:

```
np.random.seed(20181107)
y0 = 3 * np.random.normal(0,1)
y1 = -5 * np.random.normal(0,1)
y2 = 4 * np.random.normal(0,1)
y3 = 2 * np.random.normal(0,1)
y4 = -2 * np.random.normal(0,1)
Y_raw_1 = [y0, y1, y2, y3, y4]
T = 1000
a1 = .2
a2 = 0.3
a3 = -0.4
a4 = 0.1
a5 = 0.8
for t in range(T):
    yt_5 = Y_raw_1[-5]
    yt_4 = Y_raw_1[-4]
    yt_3 = Y_raw_1[-3]
    yt_2 = Y_raw_1[-2]
    yt_1 = Y_raw_1[-1]
    e = np.random.normal(0,1)
    yt = a1 * yt_1 + a2 * yt_2 + a3 * yt_3 + a4 * yt_4 + a5 * yt_5 + e
    Y_raw_1.append(yt)
Y_raw_1 = np.array(Y_raw_1[5:])
Y_raw_1 = Y_raw_1.reshape(T, 1)

Y_raw_2 = np.array([3 * np.sin(t/20) + np.random.normal(0,1) for t in range(T)])
Y_raw_2 = Y_raw_2.reshape(T, 1)

Y_raw_3 = np.array([t/5*np.sin(t/20) + np.random.normal(0,np.sqrt(t)) for t in range(T)
    ])
Y_raw_3 = Y_raw_3.reshape(T, 1)

data = np.stack((Y_raw_1, Y_raw_2, Y_raw_3), axis=1)
data = data.reshape(data.shape[0], data.shape[1] * data.shape[2])
data = pd.DataFrame(data, columns=['Y1', 'Y2', 'Y3'])
pd.DataFrame.to_csv(data, 'Simulated_Data.csv')
```

Figure 3 plots the simulated data, and table 1 provides the summary statistics of these simulated time series. Evidently, ($\rho_{12} = 0.02$), and between TS1 and TS3 ($\rho_{13} = 0.04$). This implies that TS2 nd TS3 have little to no predictive power for TS1. We will use this implication later on, in testing our models.

We eventually take our models to be tested on real data; we use Beijing PM2.5 Data Set[2]. This hourly data set contains the PM2.5 data of US Embassy in Beijing. Meanwhile, meteorological data from Beijing Capital International Airport are also included. It has 43824 instances. After cleaning the data (dropping the missing values), removing the irrelevant features, and dummy coding the categorical variables, we end up with a data set of the size (41757, 8). The features are:

- pm2.5: PM2.5 concentration (ug/m^3),

[2]Dheeru and Karra Taniskidou (2017)

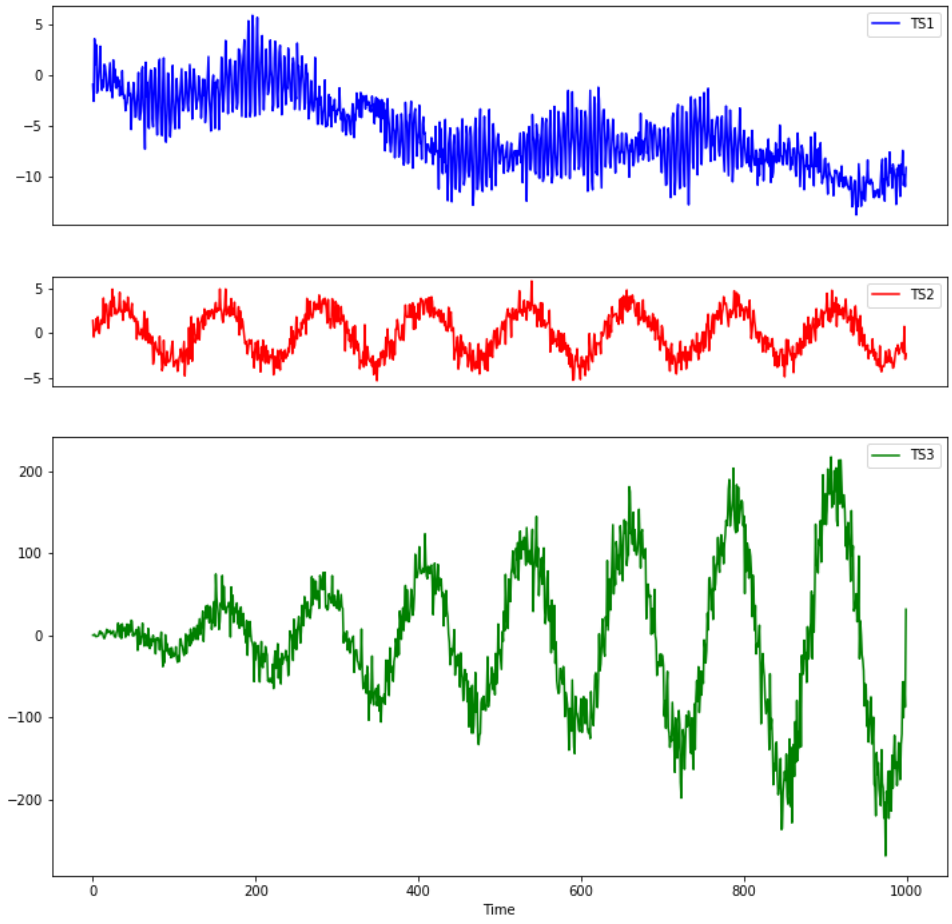

Figure 3: Simulated time series.

Table 1: Summary Statistics (simulated time series)

	Mean	Std Dev	AC(1)	AC(2)	ρ_{1j}	ρ_{2j}	ρ_{3j}
$TS1$	-5.53	3.75	0.8	0.61	1	0.02	0.04
$TS2$	0.0	2.36	0.82	0.82		1	0.75
$TS3$	-3.62	85.82	0.93	0.93			1

Sample mean, standard deviation, first and second autocorrelations (AC), and correlations (ρ_{ij}) are reported for three simulated time series.

- DEWP: Dew Point (F),
- TEMP: Temperature (F),
- PRES: Pressure (hPa) ,
- cbwd: Combined wind direction,
- Iws: Cumulated wind speed (m/s),

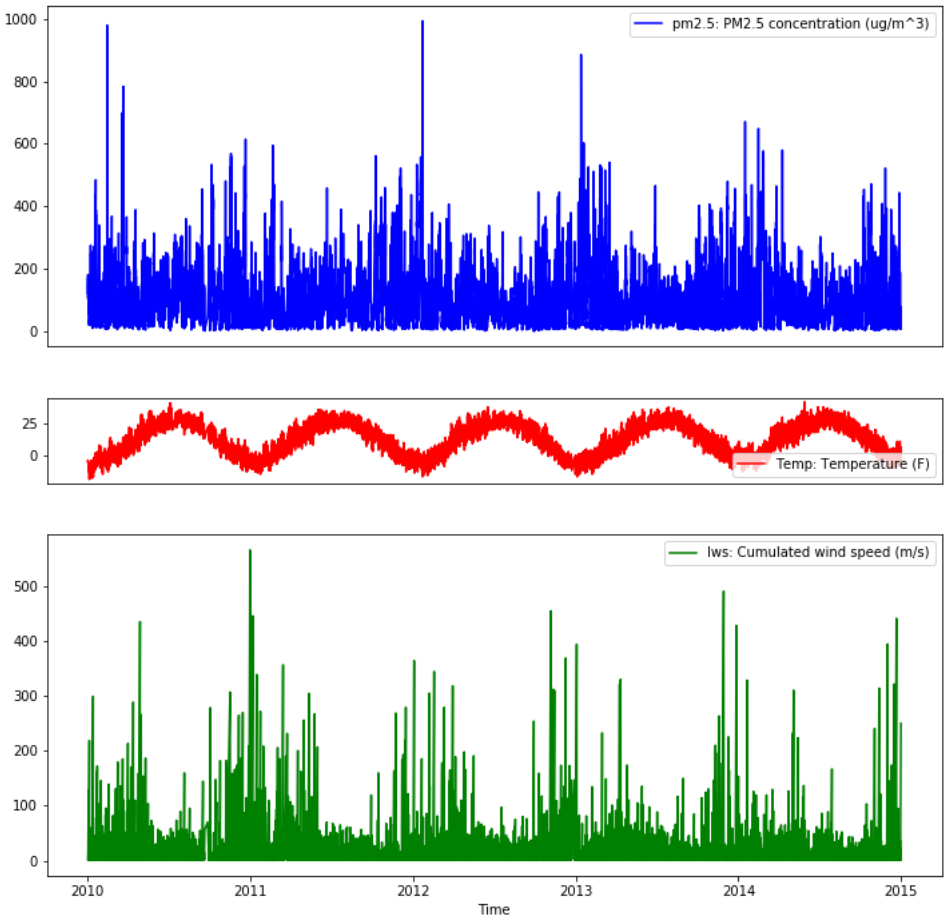

Figure 4: Beijing PM2.5 Data Set. PM2.5 concentration, along with two attributes, temperature and wind speed, are depicted.

- Is: Cumulated hours of snow,
- Ir: Cumulated hours of rain.

Figure 4 plots the features "pm2.5", "Temp", and "Iws", and table 2 summarizes some statistical properties of these series. These series are highly persistent (high AC(1) and AC(2)) and moreover, pm2.5 has a considerable amount of correlation with some of the features (e.g. $\rho_{12} = 0.17$, and $\rho_{15} = -0.25$)

6 Prediction Strategies

Since our models are designed to predict a univariate series, in both databases, we are eventually interested in one of the time series. The *time series of interest* in the simulated data set is TS1, and in the PM2.5 data set is pm2.5.; however at times we test our models on other times series of the data sets as well (e.g. on TS2 and TS3).
We follow two routes in making the predictions:

- univariate prediction: this is akin to a fitting an autoregressive process to the time series of interest: $y_t = f(y_{t-1}, y_{t-2}, \ldots)$.

Table 2: Summary Statistics (Beijing PM2.5 Data Set)

	Mean	Std Dev	AC(1)	AC(2)	ρ_{1j}	ρ_{2j}	ρ_{3j}	ρ_{4j}	ρ_{5j}
pm2.5	98.61	92.05	0.96	0.92	1	0.17	-0.09	-0.05	-0.25
DEWP	1.75	14.43	1.0	0.99		1	0.82	-0.78	-0.29
TEMP	12.4	12.18	0.99	0.98			1	-0.83	-0.15
PRES	1016.44	10.3	1.0	0.99				1	0.18
Iws	23.87	49.62	0.95	0.9					1

Sample mean, standard deviation, first and second autocorrelations (AC), and correlations (ρ_{ij}) are reported for five attributes of the Beijing PM2.5 Data Set.

- multivariate prediction: we are still forecasting one series, i.e. TS1 or pm2.5, however we include the other time series in our model, as exogenous variables: $y_t = f(y_{t-1}, y_{t-2}, \ldots, X_{t-1}, X_{t-2,\ldots})$.

In both aforementioned cases, we can perform a 1 step or multi-step prediction based on the value of the hyperparameter *n_out*, which specifies the number time periods into the future for which y_t is predicted. To help with the terminology, table 3 summarizes these distinct cases.
The reason we distinguish between 1-step and multi-step predictions is to have a better understanding of the performance of the models in short-term vs long-term prediction tasks.

Table 3: Prediction Strategies

	$y_t = f$(lags of y_t only)	$y_t = f$(lags of y_t and of X_t)
n_out = 1	1 step, univariate	1 step, multivariate
n_out > 1	multi-step univariate	multi-step multivariate

Classification of prediction strategies.

7 Results

7.1 1-step Univariate Time Series Forecasting: A Comparison between FF and CNN

In this section we try to answer this question:

- How do CNN and FF models perform, relatively, for *short-term* prediction of a time series based on only its own past.

We define "short-term" to be one time period ahead. In doing the comparison between the models, we attempt to keep the common parameters, such as the length of the block of past observations that is fed to the network, i.e. n_input, the same. We are aware that each model needs a separate, case-based fine-tuning, and hence comparing the performance of two models might not seem totally well-grounded; however, at the same time, one of our goals is that our models stay agnostic about the nature of the data and perform well in a general setting.
Moreover, given the stochastic nature of models (due to small batch size and employing stochastic gradient descent optimization algorithms), one needs to rerun the models in a Monte Carlo simulation setting, before making any positive recommendation regarding which model to be used in a certain setting. We are limited in terms of our computation power and, consequently, only make descriptive observations.

7.1.1 1-step Time Series Forecasting With a Univariate FF

The FF network is trained on 20 past observations (n_input = 20), and it predicts the next observation (n_out = 1). In the test stage, the trained model is applied on the test data and the prediction is compared with the actual observation. Repeating this procedure on the all samples of test data set, the mean squared error is calculated.

Table 4: Mean Squared Error- 1 step forecast of univariate FF on simulated TS

	$TS1$	$TS2$	$TS3$
mse	2.7322	1.3399	1223.2942

Mean Squared Error for 1 step prediction of the simulated time series, using the FeedForward network with n_input = 20.

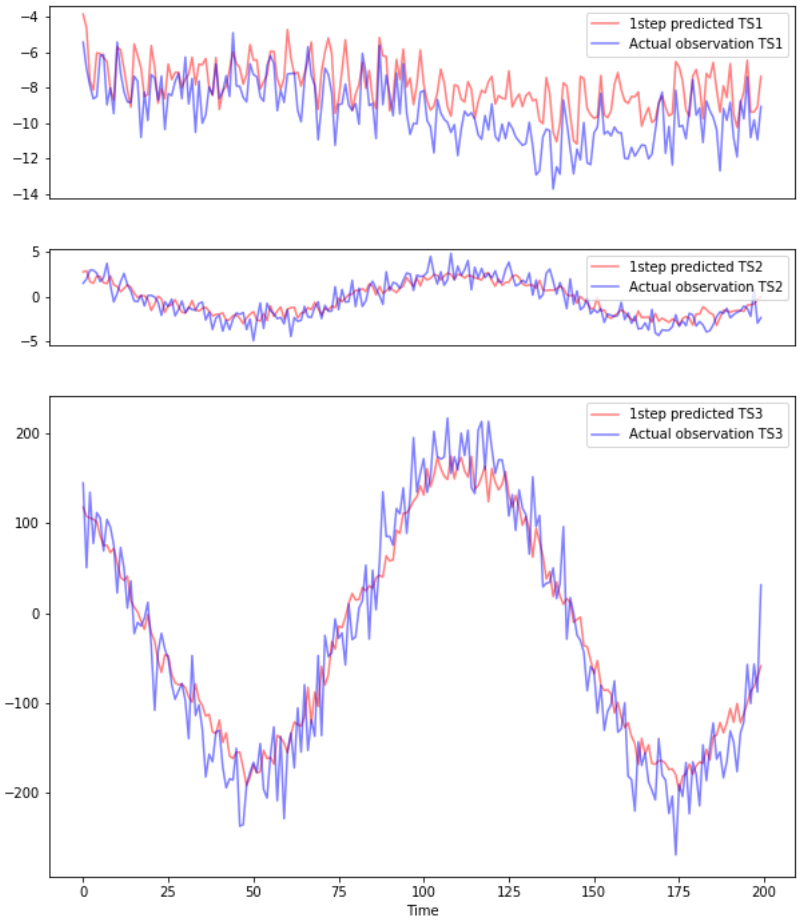

Figure 5: 1 step prediction of the simulated data using the univariate FeedForward network, along with the actual data (n_input = 20).

Table 4 reports the mean square error of 1 step prediction for TS1, TS2, and TS3, and figure 5 depicts the prediction vs actual data, on the test set, for each series.

The FF network seems to do a fine job of forecasting the simulated series. Although mse seems to be high for TS3, one should take into consideration that TS3 has a wide range and huge variability.

7.1.2 1-step Time Series Forecasting With a Univariate CNN

We repeat the previous analysis, this time using the convolutional network, and compare the results with the FF case. Table 5 and figure 6 are exact counterparts of table 4 and figure 5, respectively.

Table 5: Mean Squared Error- 1 step forecast of univariate CNN on simulated TS

	$TS1$	$TS2$	$TS3$
mse	1.4187	1.2105	1330.2607

Mean Squared Error for 1 step prediction of the simulated time series, using the Convolutional network with n_input = 20.

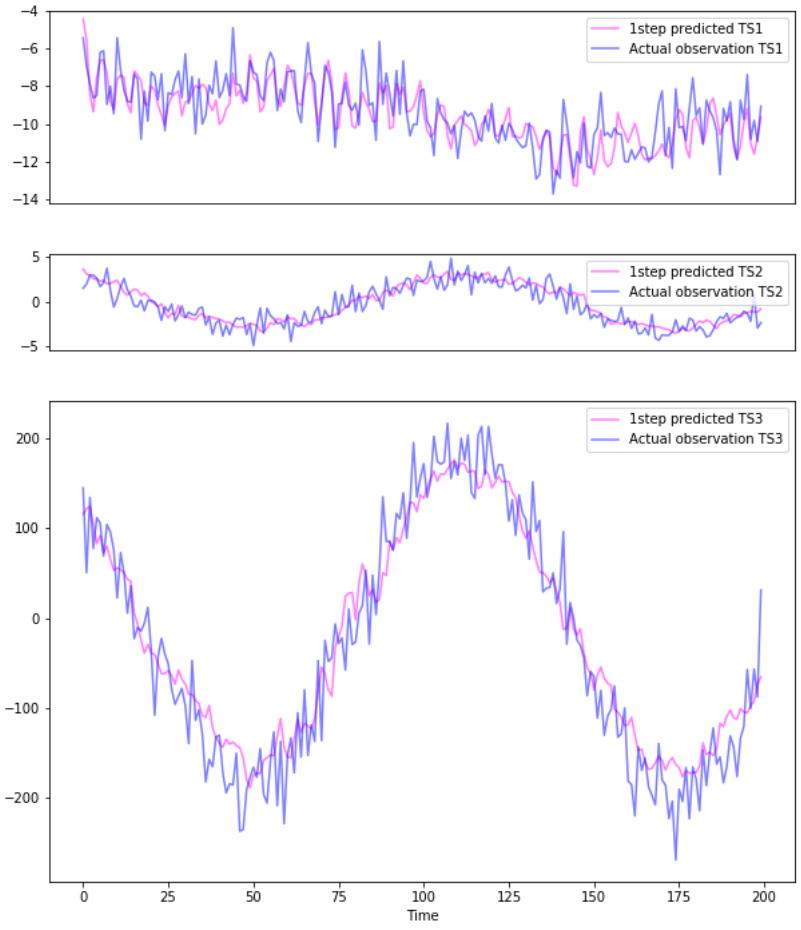

Figure 6: 1 step prediction of the simulated data using the univariate Convolutional network, along with the actual data (n_input = 20).

Comparing each model's mse, what's noticeable is that *CNN makes less prediction error on stationary series TS1 and TS2, but fares worse on the nonstationary and highly volatile TS3.* This might be partially

due to its relatively small kernel size (4).

7.2 Multi-step Univariate Time Series Forecasting: A Comparison between FF and CNN

In this section we aim to answer this question:

- How do CNN and FF models perform, relatively, for *long-term* prediction of a time series based on only its own past.

In particular, we predict 7 time periods into the future based on past 20 observations (i.e. n_out = 7 and n_input = 20).
We report the mean squared error for each lead time, and also an overall score, which is the simple average of all the mse's across the lead time periods. We restrict our attention only to the series TS1 when studying how well FF and CNN models perform.

7.2.1 Multi-step Time Series Forecasting With a Univariate FF

Table 6 reports the results. Not surprisingly, mse increases, on average, as the model predicts further into the future. Figure 7 visualizes this phenomenon: The prediction of 7 days into the future (bottom panel) gets closer to the average of the data, compared to the prediction of 1 day ahead (top panel). Nonetheless, the model seems to do a good job even for predicting far into the future: the prediction of the 7th day follows the troughs and peaks of the actual data pretty consistently.

Table 6: Mean Squared Error- multi-step forecast of univariate FF on TS1

	day 1	day 2	day 3	day 4	day 5	day 6	day 7	overall
mse	1.1213	1.0862	1.2027	1.3764	1.3397	1.7248	1.7888	1.3771

Mean Squared Error and overall score for multi-step prediction of simulated time series TS1, using the univariate FeedForward network with n_input = 20, n_out = 7.

7.2.2 Multi-step Time Series Forecasting With a Univariate CNN

Table 7 summarizes the results, and figure 8 compares the predictions of 1st and 7th days. Comparing the mse's between FF and CNN, it's evident that our FF model uniformly performs better.

Table 7: Mean Squared Error- multi-step forecast of univariate CNN on TS1

	day 1	day 2	day 3	day 4	day 5	day 6	day 7	overall
mse	1.4349	1.5325	1.6977	1.5528	1.7274	1.8442	2.4147	1.74350

Mean Squared Error and overall score for multi-step prediction of simulated time series TS1, using the univariate Convolutional network with n_input = 20, n_out = 7.

7.3 Multivariate Forecasting Using the CNN model

In this section we solely focus on the performance of the CNN model. Remember that a multivariate forecast is akin to fitting this functional form: $y_t = f(y_{t-1}, y_{t-2}, \ldots, X_{t-1}, X_{t-2,\ldots})$. As figure 2 depicts, multiple variables can be modeled as multiple channels of a convolutional network. Therefore, the terms multivariate CNN and multichannel CNN can be used interchangeably. One question that we try to answer is:

- compared to the univariate case, how will be the performance of multivariate CNN?

To answer this question we compare the results of multi-step (n_out>1) forecasting for a univariate and a multivariate CNN.

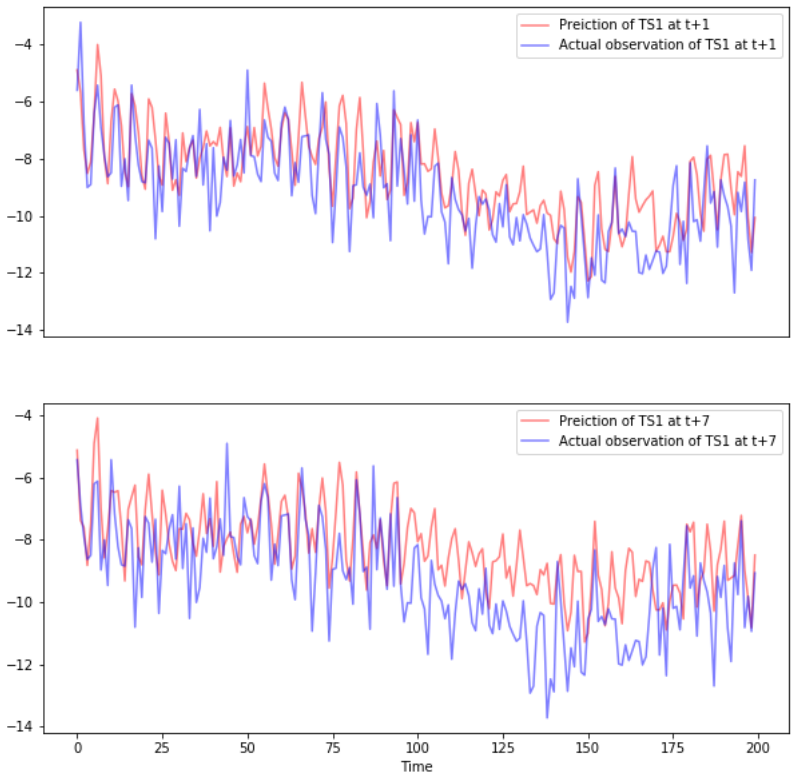

Figure 7: Multi-step prediction of the simulated series TS1 using the univariate FeedForward network, along with the actual data (n_input = 20, n_out = 7). The top graph shows the result of the prediction for 1 time period ahead, and the bottom graph shows the result of the prediction for 7 time periods ahead.

7.3.1 The Case of Simulated Data Set

Firstly, we study the case of simulated data set [TS1, TS2, TS3]. We have already reported the univariate results in the section 7.2.2. There, $y_t = TS1_t$ was forecast based the lagged values of TS1. In the multichannel version $y_t = TS1_t$ is forecast based on lagged TS1 and lagged $X_t = [TS2_t, TS3_3]$.
Table 8 reports the results. Comparing the results of this table with those reported in table 7 reveals that TS1 can be forecast better by using merely its own past values. Remember that TS1 had almost zero correlation with TS2 and TS3. Including TS2 and TS3 as exogenous variables should add no benefit in forecasting TS1, and might even hurt its forecastibilty. The results of the tables demonstrate that our CNN model conforms to this observation pretty well. Figure 9 also shows the deteriorating effect of inclusion of X = [TS2, TS3] in the prediction of TS1.

Table 8: Mean Squared Error- multi step forecast of multivariate CNN on TS1

	day 1	day 2	day 3	day 4	day 5	day 6	day 7	overall
mse	5.3115	6.3338	7.0165	5.3062	6.6480	6.2629	7.3654	6.3206

Mean Squared Error and overall score for multi-step prediction of simulated time series TS1, using the multivariate Convolutional network with n_input = 20, n_out = 7.

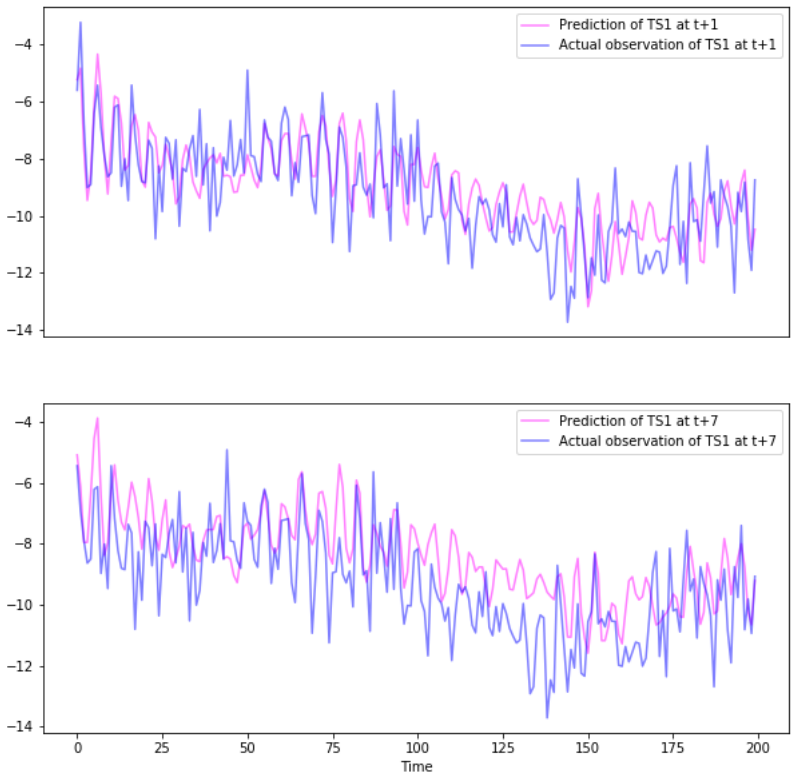

Figure 8: Multi-step prediction of the simulated series TS1 using the univariate Convolutional network, along with the actual data (n_input = 20, n_out = 7). The top graph shows the result of the prediction for 1 time period ahead, and the bottom graph shows the result of the prediction for 7 time periods ahead.

7.3.2 The Case of Beijing Pollution (PM2.5) Data Set

Here we have $y_t = pm2.5$ and $X_t = [DEWP_t, TEMP_t, PRES_t, cbwd_t, Iws_t, Is_t, Ir_t]$. Table 9 reports the result. To have a benchmark, we also report the result of a univariate forecasting of pm2.5 (i.e. $pm2.5_t = f(\text{lags of } pm2.5_t)$) in table 10. Since pm2.5 has a significant correlation with some of the features in the data set, one, intuitively, expects that a multivariate forecast to be more accurate than a univariate one. The reported mse for these two cases show that our CNN model is consistent with this intuition. Finally figures 10 and 11 give a visual comparison of the performance of multivariate and univariate CNN, for the 1st and 7th lead days, respectively.

Table 9: Mean Squared Error- multi step forecast of multivariate simulated data (pm2.5)

	hour 1	hour 2	hour 3	hour 4	hour 5	hour 6	hour 7,	overall
mse	588.84	1120.63	1671.39	2216.28	2778.57	3312.85	3795.91	2212.07

Mean Squared Error and overall score for multi-step prediction of time series pm2.5, using the multivariate Convolutional network with n_input = 20, n_out = 7.

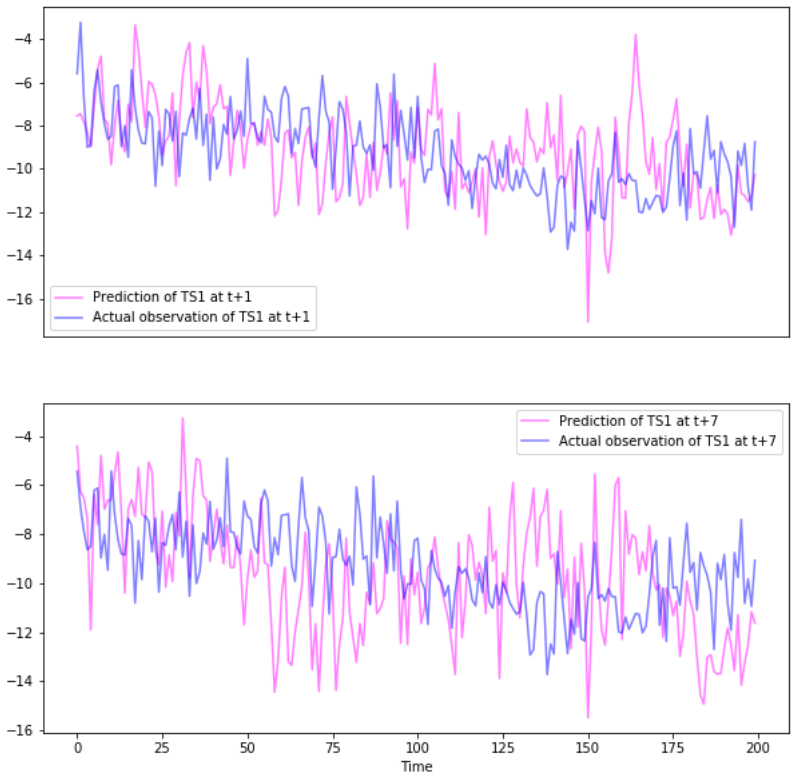

Figure 9: Multi-step prediction of the simulated series TS1 using the multivariate Convolutional network, along with the actual data (n_input = 20, n_out = 7). The top graph shows the result of the prediction for 1 time period ahead, and the bottom graph shows the result of the prediction for 7 time periods ahead.

Table 10: Mean Squared Error- multi step forecast of univariate simulated data (pm2.5)

	hour 1	hour 2	hour 3	hour 4	hour 5	hour 6	hour 7,	overall
mse	1090.61	1682.18	2276.93	2822.94	3339.95	3827.82	4245.13	2755.08

Mean Squared Error and overall score for multi-step prediction of time series pm2.5, using the univariate Convolutional network with n_input = 20, n_out = 7.

8 Conclusion

We designed and studied the time-series forecasting performance of two types of networks: a benchmark Feedforward network and a Convolutional network. By tweaking the input data representation, the FF network performs impressively well in forecasting time series of various types. This can be seen as a testimony to the merits of having a model with not many hyperparameters that is able to generalize easily. Our CNN model also performs well, especially in picking up short-term patterns.
Some more work can be done: The small batch size and the stochastic nature of the optimization algorithms mean that the same model will learn a slightly different mapping of inputs to outputs each time it is trained. This means results may vary when the model is evaluated. This can be exploited as a regularization strategy: one can run the model multiple times and calculates an average of model performance.

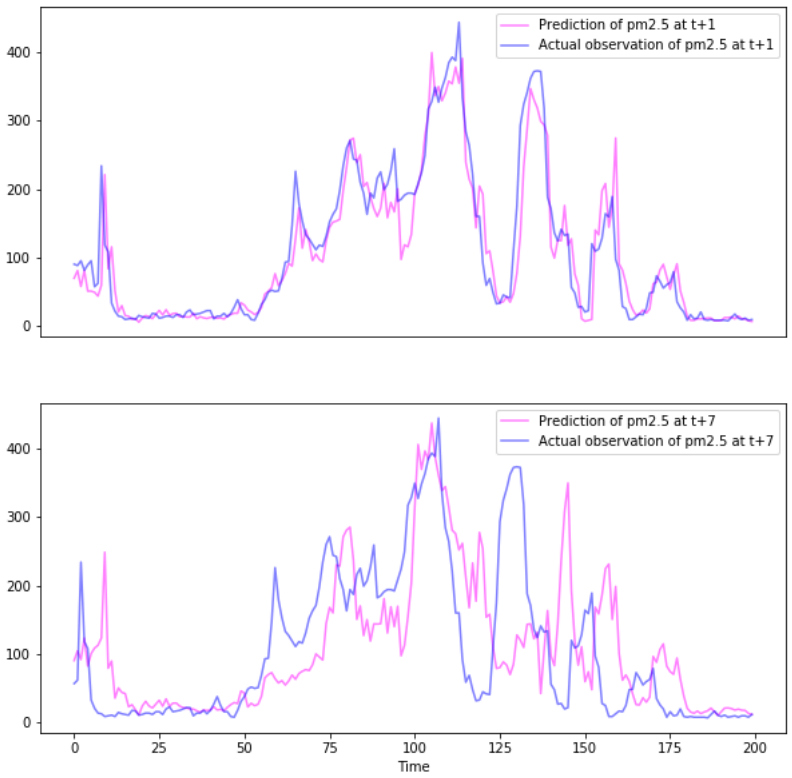

Figure 10: Multi-step prediction of the series pm2.5 using the multivariate Convolutional network, along with the actual data (n_input = 20, n_out = 7). The top graph shows the result of the prediction for 1 time period ahead, and the bottom graph shows the result of the prediction for 7 time periods ahead.

References

Dheeru, Dua and Efi Karra Taniskidou (2017). *UCI Machine Learning Repository*. URL: `http://archive.ics.uci.edu/ml`.

Liang, Xuan et al. (2015). "Assessing Beijing's PM2. 5 pollution: severity, weather impact, APEC and winter heating". In: *Proc. R. Soc. A* 471.2182, p. 20150257.

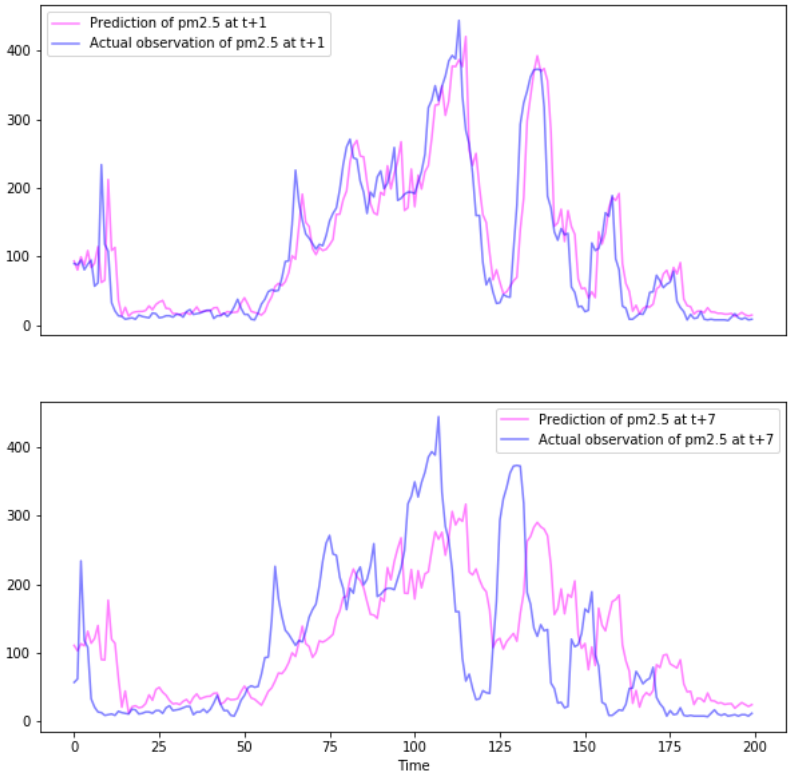

Figure 11: Multi-step prediction of the series pm2.5 using the univariate Convolutional network, along with the actual data (n_input = 20, n_out = 7). The top graph shows the result of the prediction for 1 time period ahead, and the bottom graph shows the result of the prediction for 7 time periods ahead.