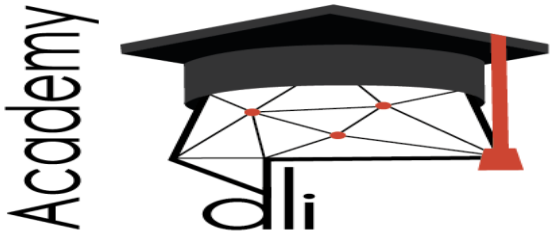


Deep Learning & Time Series Prediction

Francesco Pugliese, PhD

Italian National Institute of Statistics, Division "Information and Application Architecture", Directorate for methodology and statistical design

Email **Francesco Pugliese** :

f.pugliese@deeplearningitalia.com

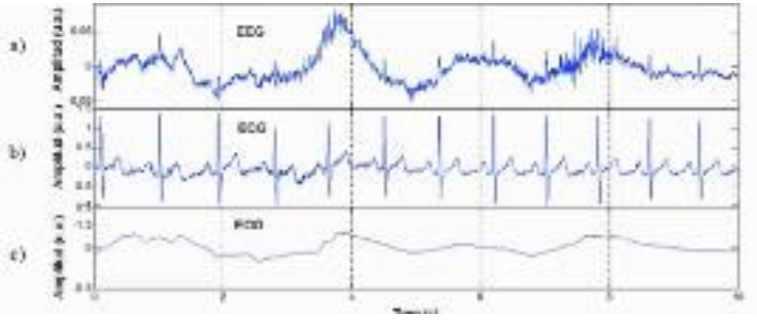
francesco.pugliese@istat.it

Time-Series

Time series are a sequence of data points: namely a Univariate time series is a sequence of simple numbers.

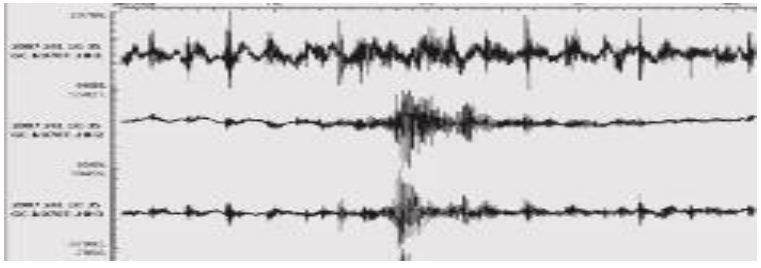
Time series can also take values from a vector space. In this case you they consist of multiple univariate time series: one for each vector component examples.

Example:
- Speed of a car

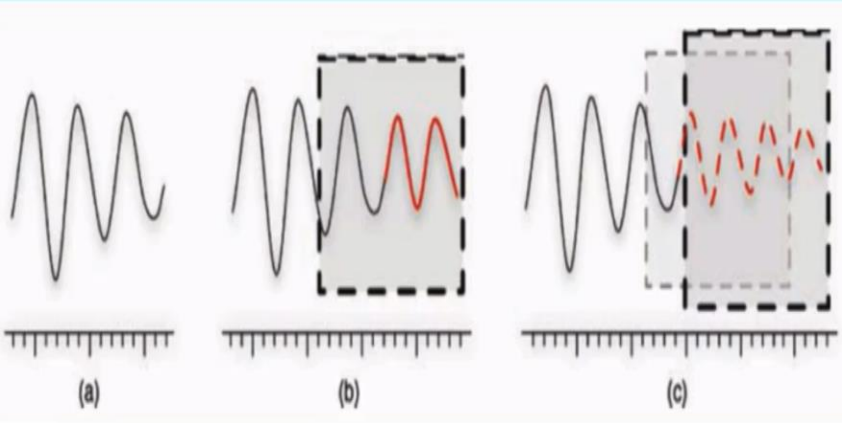


Example:

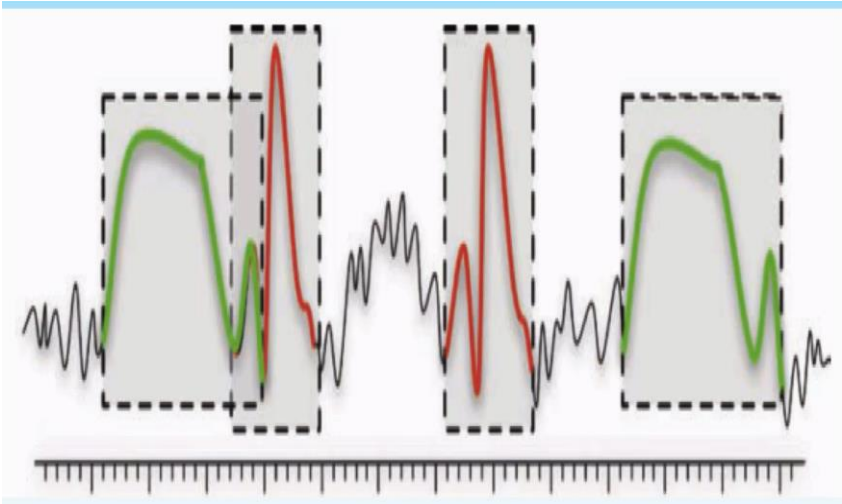
- Stock market
- Music
- Website monitoring
- Biosensors (EEG, ECG etc)
- IoT
- Energy Monitoring
- Eartquakes



Time-Series Prediction and Classification



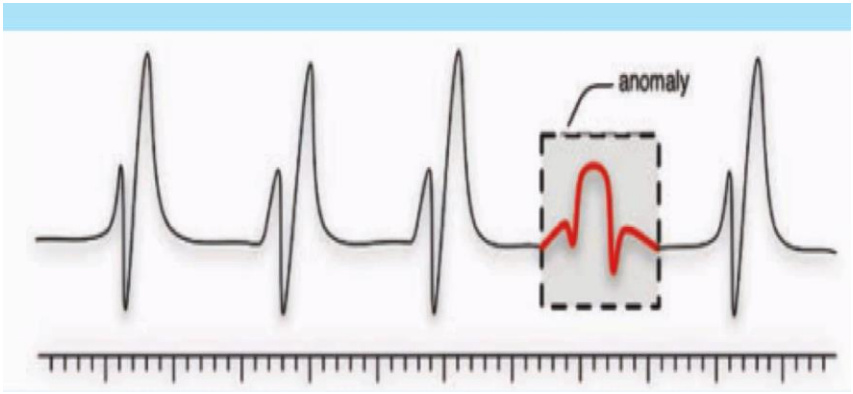
- We intend to predict the future values in the sequence (regression problem). We are going to predict the continuous quantity by exploiting features derived from the time series.



- We can identify regions of recurring patterns

Time-Series Analysis for Anomaly Detection

- In order to identify anomalies given a sequence with regular behavior we have to pick up where each anomaly deviates from the regularity.



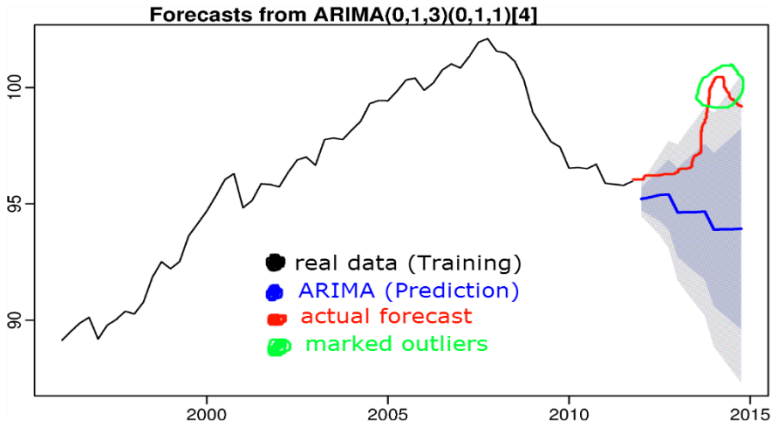
Deep Learning for Time-Series Prediction

- The application of **Deep Learning approaches** to time-series prediction has received a great deal of attention from both **entrepreneurs** and **researchers**. Results show that deep learning models outperform other statistical models in predictive accuracy **(Bao, et al., 2017).**

The application of classic time series models, such as **Auto Regressive Integrated Moving Average (ARIMA)**, usually requires strict assumptions regarding the distributions and stationarity of time series. For complex, non-stationary and noisy time-series it is necessary for one to know the properties of the time series before the application of classic time series models **(Bodyanskiy and Popov, 2006).** Otherwise, the forecasting effort would be ineffective.

$$X_t - \alpha_1 X_{t-1} - \cdots - \alpha_{p'} X_{t-p'} = \varepsilon_t + \theta_1 \varepsilon_{t-1} + \cdots + \theta_q \varepsilon_{t-q},$$

$$\left(1 - \sum_{i=1}^{p'} \alpha_i L^i\right) X_t = \left(1 + \sum_{i=1}^{q} \theta_i L^i\right) \varepsilon_t$$

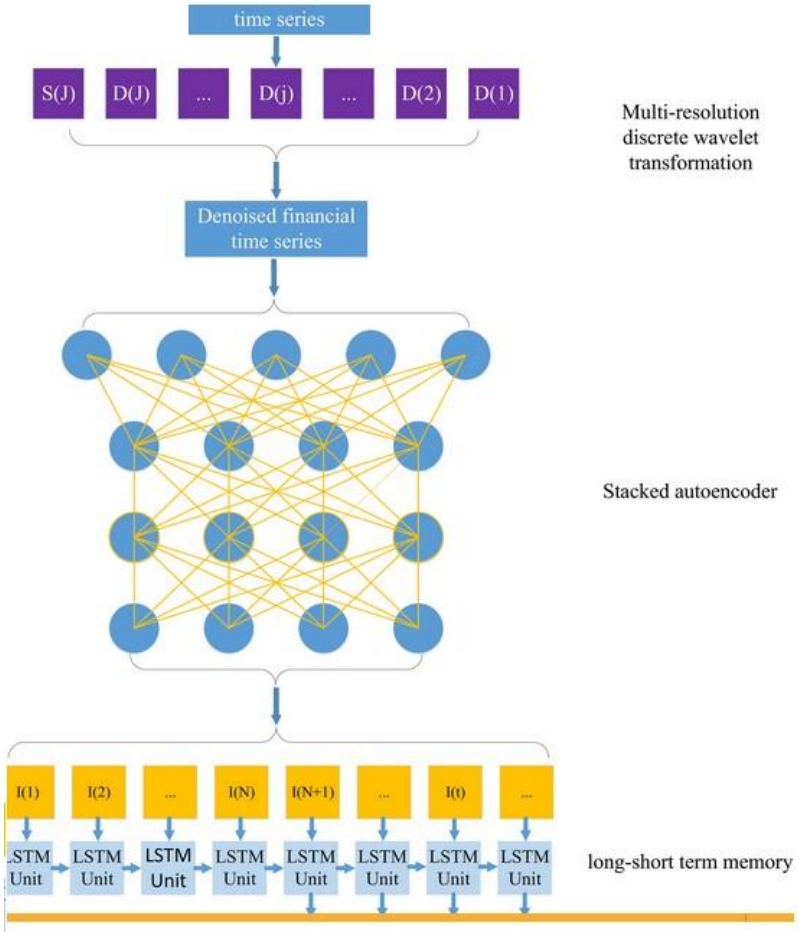


Advantages of Artificial Neural Networks (ANNs) in Time-Series Prediction

- However, by using ANNs, a priori analysis as ANNs do not require prior knowledge of the time series structure because of their black-box properties **(Nourani, et al., 2009)**.

- Also, the impact of the stationarity of time series on the prediction power of ANNs is quite small. It is feasible to relax the stationarity condition to non-stationary time series when applying ANNs to predictions **(Kim, et al., 2004)**.

- ANNs allow **multivariate time-series forecasting** whereas classical linear methods can be difficult to adapt to multivariate or multiple input forecasting problems.

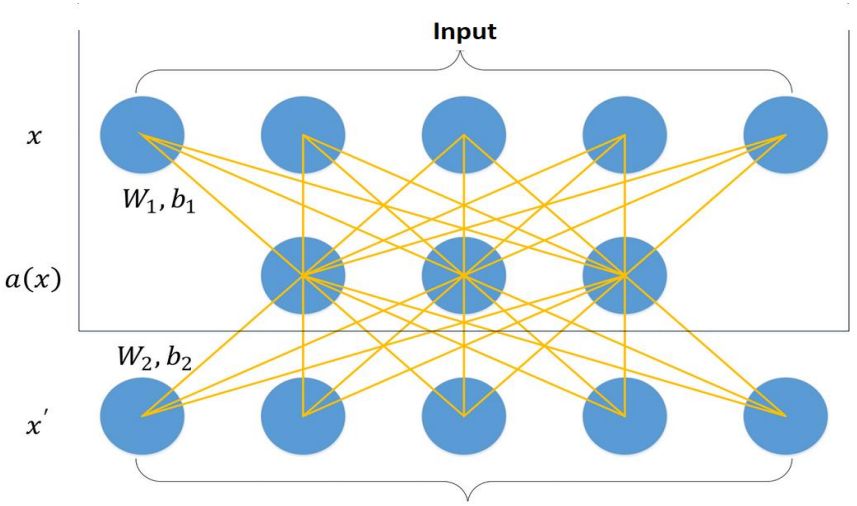


Stacked Auto-encoders (SAEs)

- According to recent studies, better approximation to nonlinear functions can be generated by stacked deep learning models than those models with a more shallow structure.

- A Single layer **Auto-Encoder (AE)** is a three-layer neural network. The first layer and the third layer are the input layer and the reconstruction layer with *k* units, respectively. The second layer is the hidden layer with *n* units, which is designed to generate the deep feature for this single layer AE.

- The aim of training the single layer AEE is to minimize the error between thee input vector and the reconstruction vector by **gradient descent**.



$$a(x) = f(\mathbf{W}_1 x + b_1)$$

$$x' = f(\mathbf{W}_2 a(x) + b_2)$$

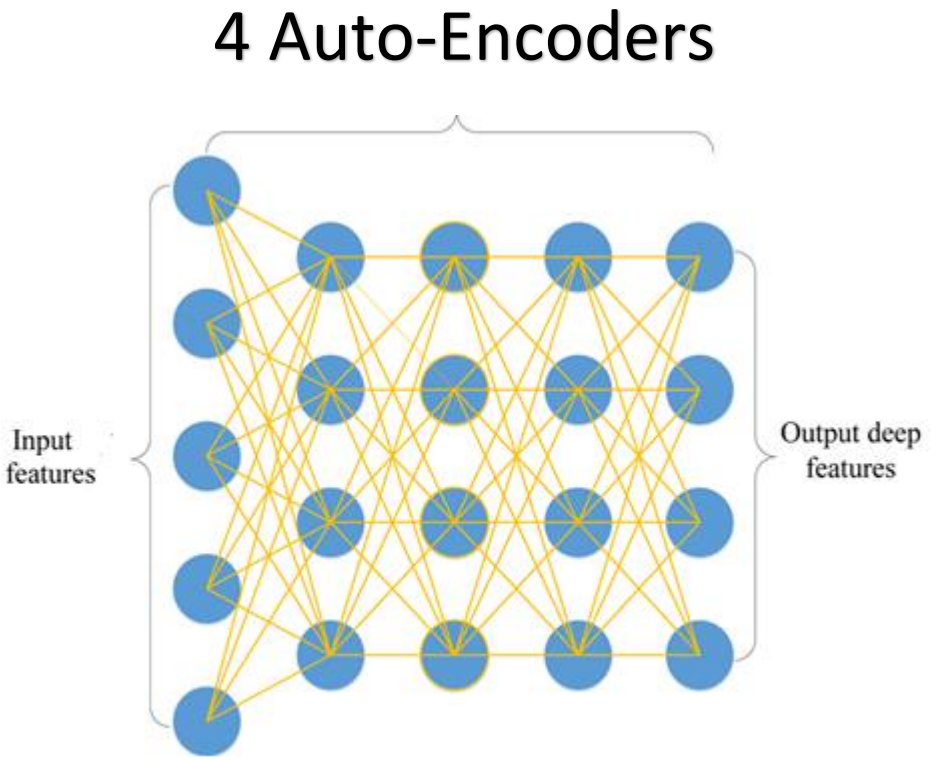
Stacked Auto-encoders (SAEs)

- Stacked auto-encoders (SAEs) are constructed by stacking a sequence of single-layer AEs layer by layer **(Bengio Y, et. Al. 2007)**.

- After training the first single-layer auto-encoder, the reconstruction layer of the first single layer auto-encoder is removed (included weights and biases), and the **hidden layer** is reserved as the input layer of the second single-layer auto-encoder.

- **Depth** plays an important role in **SAE** because it determines qualities like invariance and abstraction of the extracted feature.

- **Wavelet Transform (WT)** can be applied as input to SAEs to handle data particulary non-stationary (**Ramsey, (1999)** .



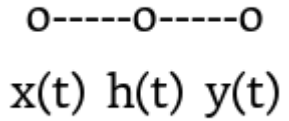
Recurrent Neural Networks (RNNs) : Elman's Architecture

- There exist seveal indicators to measure the predictive accuracy of each model (**Hsieh, et. al.,** 2011; **Theil**, 1973)

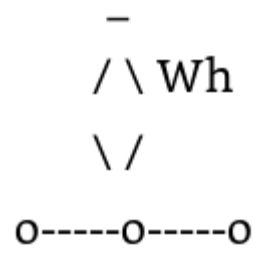
- **RMSE (Root Mean Square Error): R**epresents the sample standard deviation of the differences between predicted values and observed values.
- **MAPE (Mean Absolute Percentage Error):** Measures the size of the error in percentage terms. Most people are comfortable thinking in percentage terms, making the MAPE easy to interpret.
- Thanks to its recursive formulation, RNNs are not limited by the **Markov assumption** for sequence modeling:

$$p\{ x(t) \mid x(t-1), \ldots, x(1) \} = p\{ x(t) \mid x(t-1) \}$$

Simple Feed Forward Artificial Neural Network (MLP)



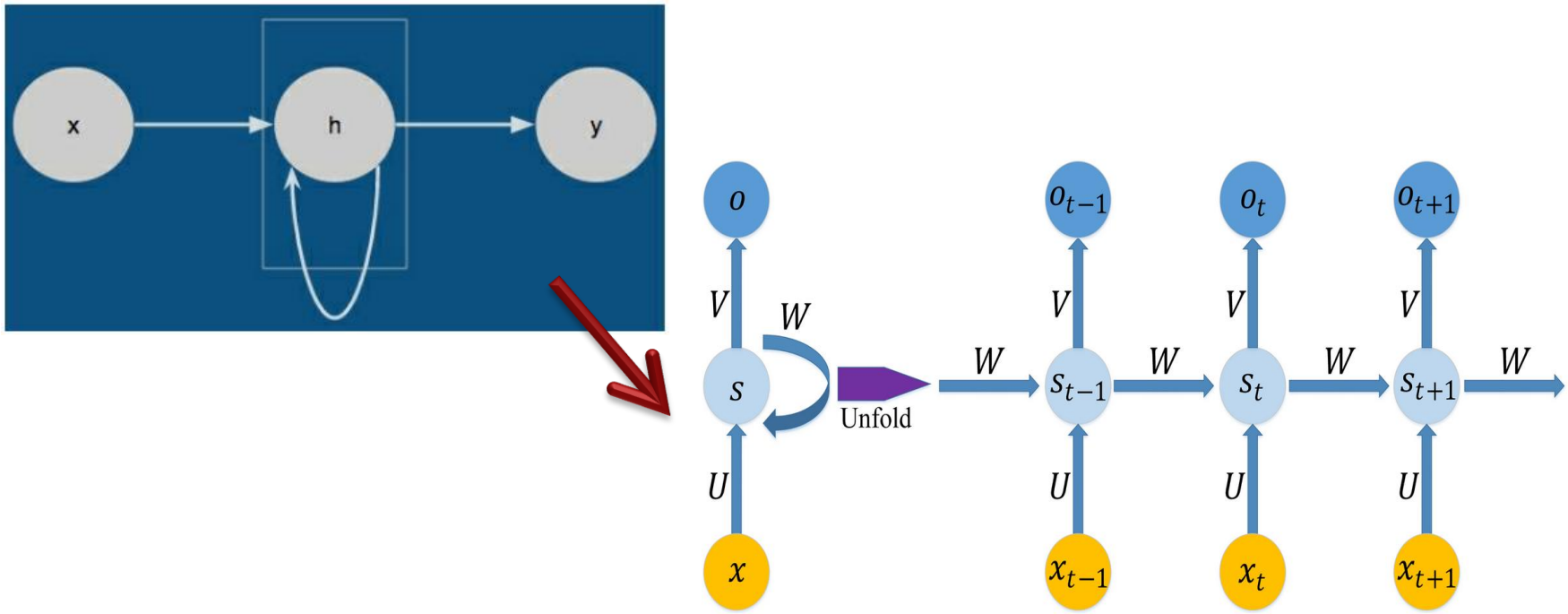
Recurrent Neural Network (Elman's Architecture)



$$h(t) = f(x(t)W_i + h(t-1)W_h + b_h)$$

Unfolding RNNs

- Although RNN models the time series well, it is hard to learn long-term dependencies because of the vanishing gradient problem in Back-Propagation Through Time (BPTT) **(Palangi H, et al., 2016)**



Back Propagation Through Time (BPTT)

- In BPTT updating weights is going to look exactly the same.
- **We can prove that the derivative of the loss function "Cross-Entropy" passes through the derivative of and the Softmax.** $p_j = \frac{e^{a_i}}{\sum_{k=1}^{N} e^{a}_{k}}$
- **Things are going to be multiplied together over and over again, due to the chain rule of calculus:**

$$d[W_h^T h(t-1)] / dW_h$$

- **The result is that gradients go down through the time (vanishing gradient problem) or they get very large very quickly (exploding gradient problem)**
- **RRNNs, GRUs, LSTMs solve the gradient problems with BPTT**

$$y(t)=\text{softmax}(W_o^T h(t))$$

$$y(t)=\text{softmax}(W_o^T f(W_h^T h(t-1)+W_x^T x(t)))$$

$$y(t)=\text{softmax}(W_o^T f(W_h^T f(W_h^T h(t-2)+W_x^T x(t-1))+W_x^T x(t)))$$

$$y(t)=\text{softmax}(W_o^T f(W_h^T f(W_h^T f(W_h^T h(t-3)+W_x^T x(t-2))+W_x^T x(t-1))+W_x^T x(t)))$$

- We drop the bias in ordcer to display things simply

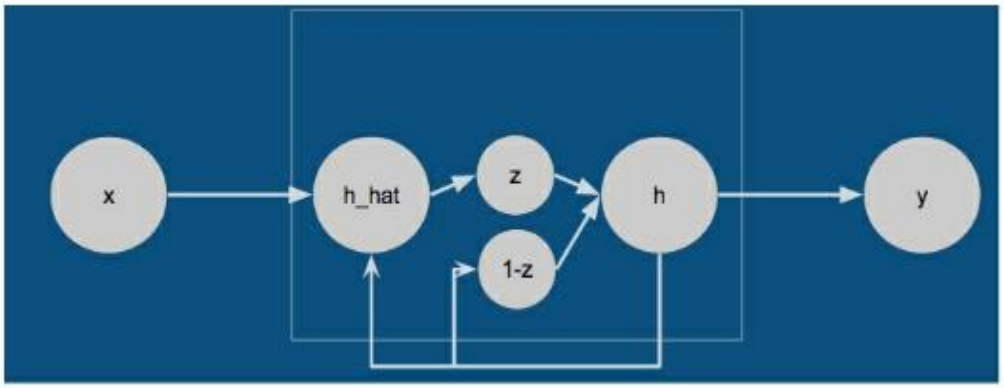
Rated Recurrent Neural Networks (RRNNs)

- The idea is to weight *f(x, h(t-1)),* which is the output of a simple RNN and *h(t-1)* which is the previous state **(Amari, et al., 1995)**.

- **We add a rating operation between what would have been the output of a simple RNN and the previous output value.**

- **This new operation can be seen as a gate since it takes a value between 0 an 1, and the other gate has to take 1 minus that value**

- **This is a gate that is choosing between 2 things: a) taking on the old value or taking the new value. As result we get a mixture of both.**



$$h\_hat(t) = f(x(t)W_x + h(t-1)W_h + b_h)$$

$$z(t) = \text{sigmoid}(x(t)W_{xz} + h(t-1)W_{hz} + b_z)$$

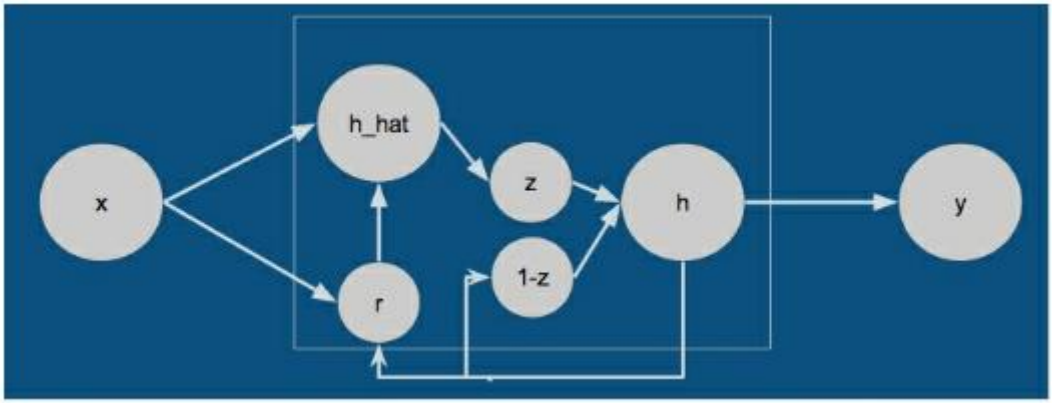
$$h(t) = (1 - z(t)) * h(t-1) + z(t) * h\_hat(t)$$

- **Z(t) is called the "rate"**

Gated Recurrent Neural Networks (GRUs)

- Gated Reccurrent Units were introduced in 2014 and are a simpler versione of LSTM. They have less parameters but same concepts **(Chung, et al., 2014).**

- **Recent research has also show that the accuracy between LSTM and GRU is comparable and even better with the GRUs in some cases.**
- **In GRUs we add one more gate with regard to RRNNs: the "reset gate r(t)" controlling how much of the previous hidden we will consider when we create a new candidate hidden value. In other words, it can "reset" the hidden value.**
- **The old gate of RRNNs is now called "update gate z(t)" balancing previous hidden values and new candidate hidden value for the new hidden value.**

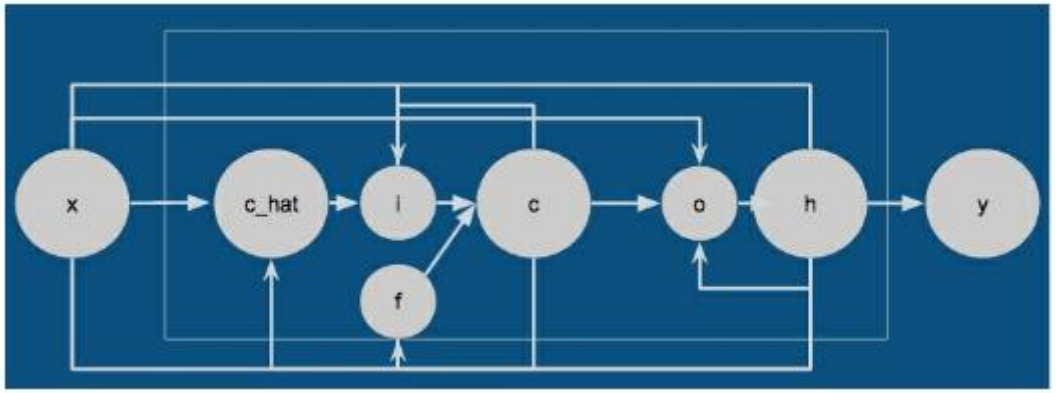


$$r_t = \sigma(x_t W_{xr} + h_{t-1} W_{hr} + b_r)$$
$$z_t = \sigma(x_t W_{xz} + h_{t-1} W_{hz} + b_z)$$
$$\hat{h}_t = g(x_t W_{xh} + (r_t \odot h_{t-1}) W_{hh} + b_h)$$
$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \hat{h}_t.$$

Long-Short Term Memories (LSTMs)

- LSTM is an effective solution for combating vanishing gradients by using memory cells **(Hochreiter, et al., 1997).**

- **A memory cell is composed of four units: an input gate, an output gate, a forget gate and a self-recurrent neuron**
- **The gates control the interactions between neighboring memory cells and the memory cell itself. Whether the input signal can alter the state of the memory cell is controlled by the input gate. On the other hand, the output gate can control the state of the memory cell on whether it can alter the state of other memory cell. In addition, the forget gate can choose to remember or forget its previous state.**

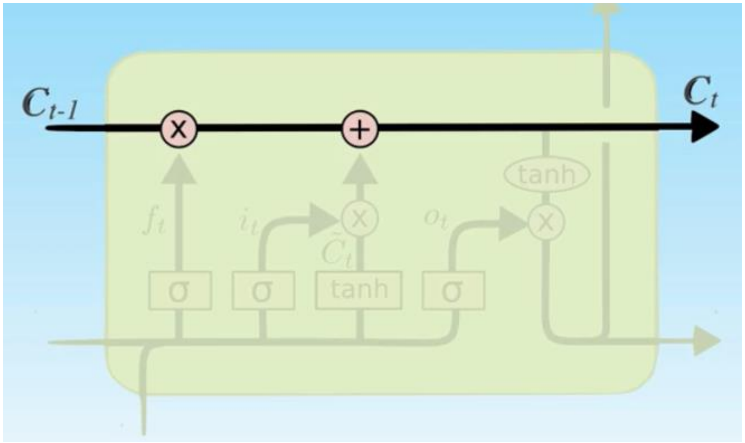


$$i_t = \sigma(x_t W_{xi} + h_{t-1} W_{hi} + c_{t-1} W_{ci} + b_i)$$
$$f_t = \sigma(x_t W_{xf} + h_{t-1} W_{hf} + c_{t-1} W_{cf} + b_f)$$
$$c_t = f_t c_{t-1} + i_t \tanh(x_t W_{xc} + h_{t-1} W_{hc} + b_c)$$
$$o_t = \sigma(x_t W_{xo} + h_{t-1} W_{ho} + c_t W_{co} + b_o)$$
$$h_t = o_t \tanh(c_t)$$

Cell State

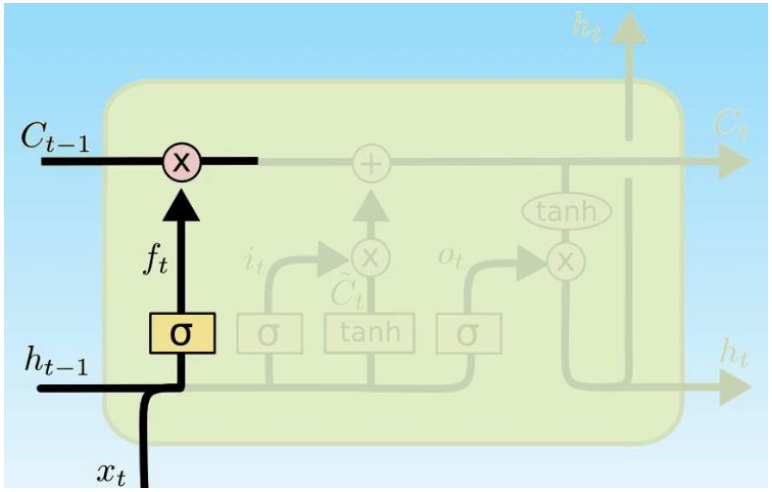
The first difference from **Vanilla** is the presence of internal state variable. This is passed from one cell to the next and it's modified by ***Operation gates.***

- Cell maintains state
- Gets modify information



Forget Gate

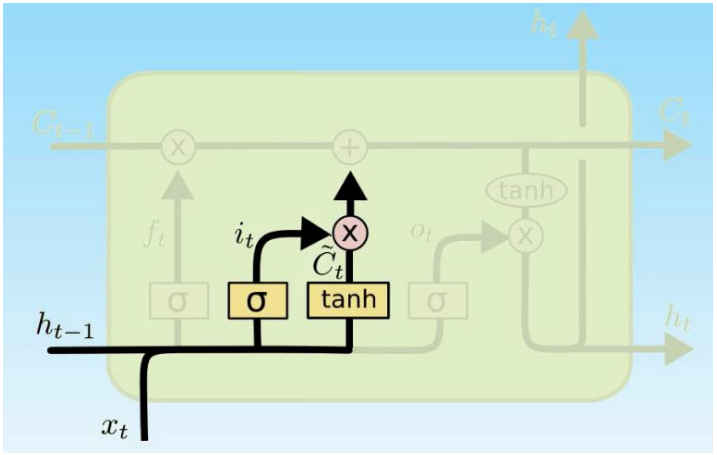
The first gate is called The ***Forget gate***. It's a sigmoid layer that takes the output at *t-1* and current input at times *c* concatenates them into a single tensor and than applies a linear transformation followed by a sigmoid because (output between 0 - 1). This number multiplies internal state for this reason the gate called ***Forget Gate***. If *ft* is 0 the previous internal state will be completely forgotten and if is 1 will be passed through an other.



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

Input Gate

The second gate is the **input gate**. The input gate takes the previous output and the new input and passes them through another sigmoid layer (this gate returns a value between 0 and 1). The value of the input gate is multiplied with the output of the candidate layer. This layer applies a tanh to the mix of the input and previous output returning a candidate vector to be added to the internal state.

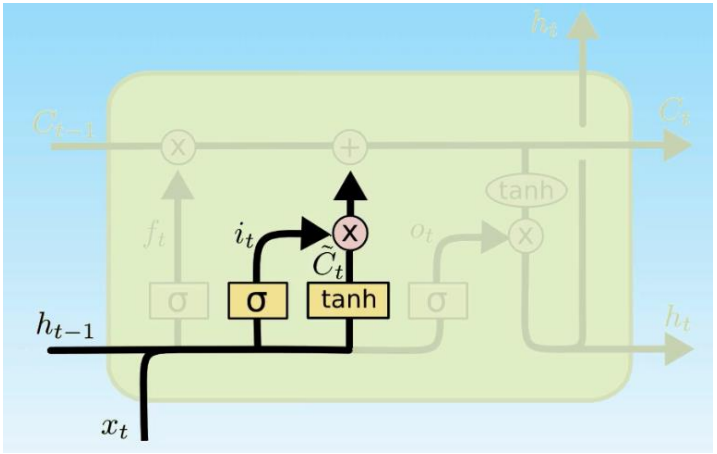


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

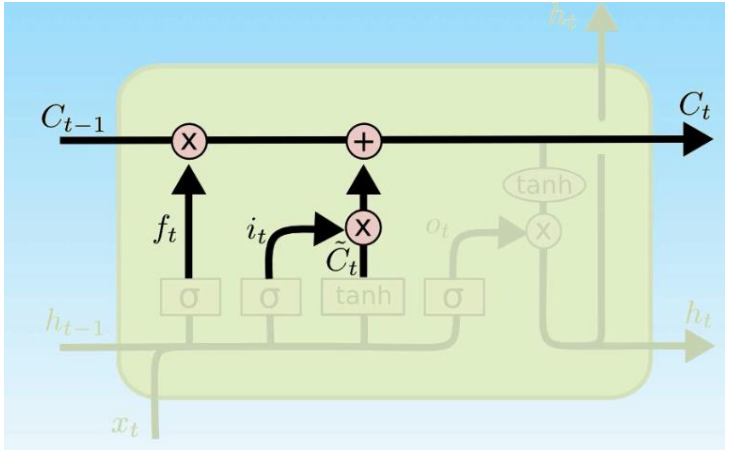
Input Gate

For example, if we are building a language model these gate would control which new relevant features to include in the internal state.



State Update

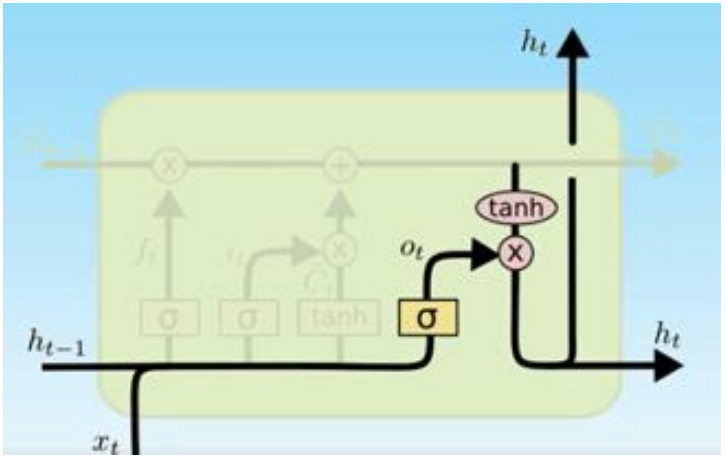
The internal state is update with this rule. The previous state is multiplied by the forget gate and then add in the fraction of the new candidate allowed by the input gate.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Output Gate

This gate control how much of the internal state is passed to the output.

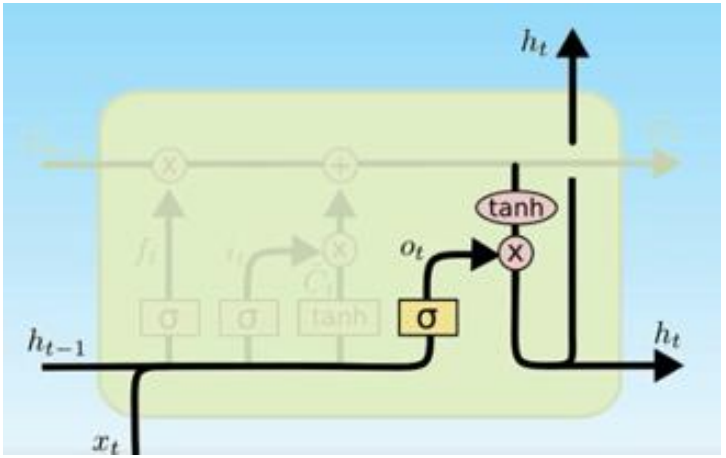


$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Output Gate

This gate control how much of the internal state is passed to the output.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

Metrics

- There exist several indicators to measure the predictive accuracy of each model (**Hsieh, et. al.,** 2011; **Theil**, 1973)

- **RMSE (Root Mean Square Error):** **R**epresents the sample standard deviation of the differences between predicted values and observed values.
- **MAPE (Mean Absolute Percentage Error):** Measures the size of the error in percentage terms. Most people are comfortable thinking in percentage terms, making the MAPE easy t interpret.
- **Theil U:** Theil U is a relative measure c the difference between two variable It squares the deviations to give mor weight to large errors and t exaggerate errors.

$$\text{RMSE} = \sqrt{\frac{\sum_{t=1}^{T}(\hat{y}_t - y_t)^2}{n}}$$

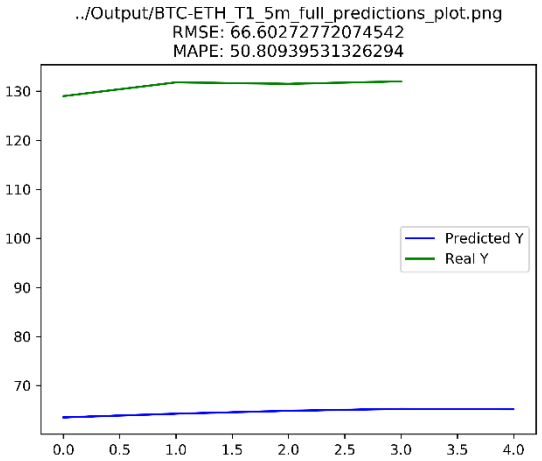
$$MAPE = \frac{\sum_{t=1}^{N} \left| \frac{y_t - y_t^*}{y_t} \right|}{N}$$

$$Theil\ U = \frac{\sqrt{\frac{1}{N}\sum_{t=1}^{N}(y_t - y_t^*)^2}}{\sqrt{\frac{1}{N}\sum_{t=1}^{N}(y_t)^2} + \sqrt{\frac{1}{N}\sum_{t=1}^{N}(y_t^*)^2}}$$

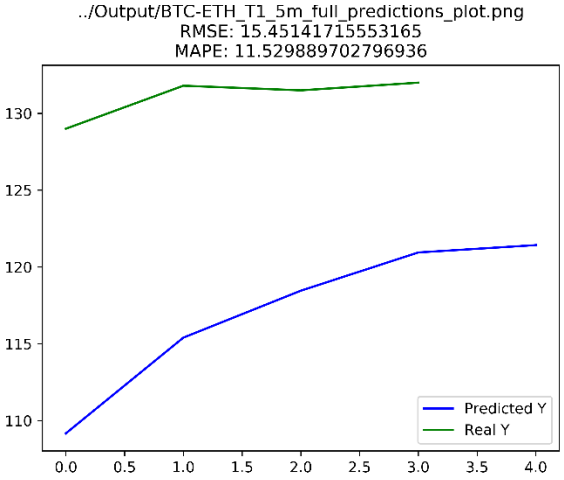
In these equations, yt is the actual value and is the predicted value.

Results: Italian GDP (Gross Domestic Production) Time Series Prediction – Yearly - 1980 – 2016 (Istat)

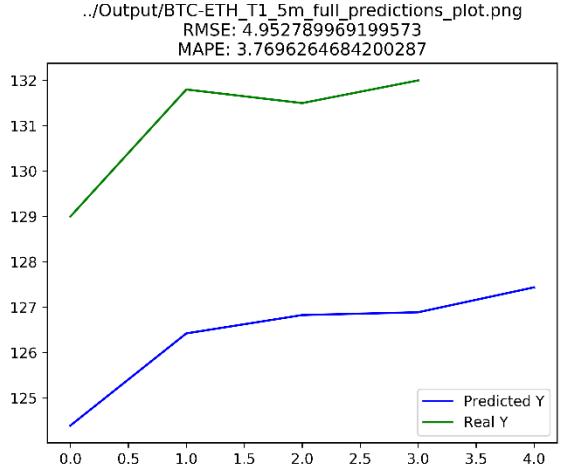
Test Set : 10%



1 Epoch



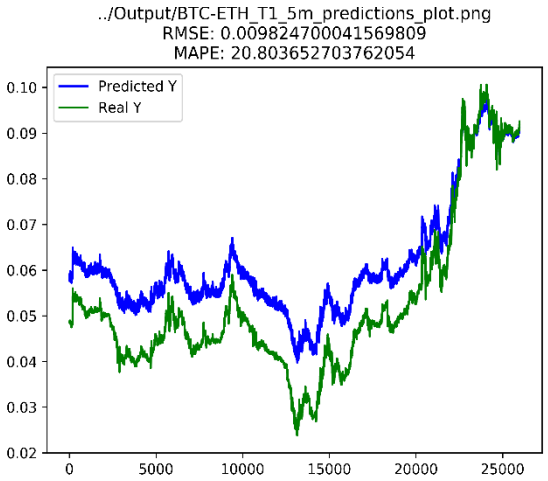
100 Epochs



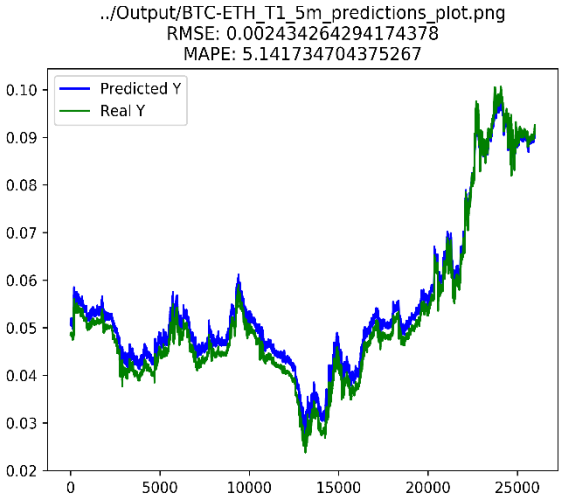
1000 Epochs

Results: Bitcoin BTC-ETH exchange Time Series Prediction – 5 mins (Poloniex)

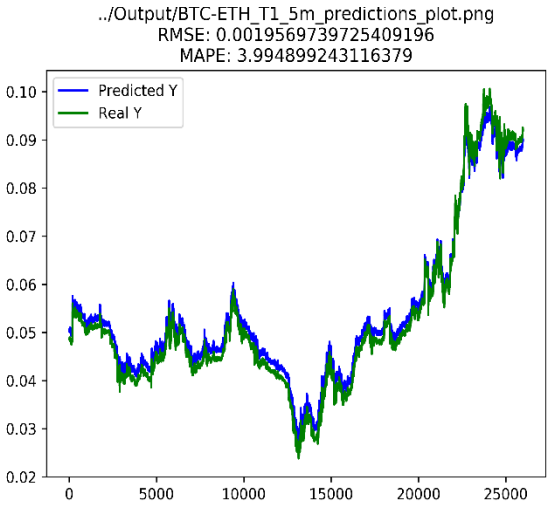
Test Set : 10%



1 Epoch



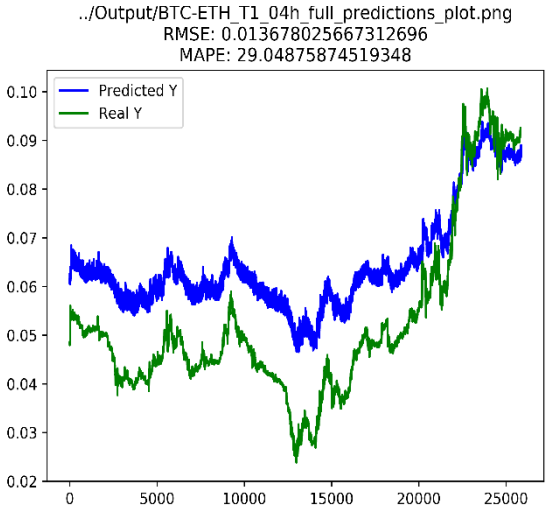
10 Epochs



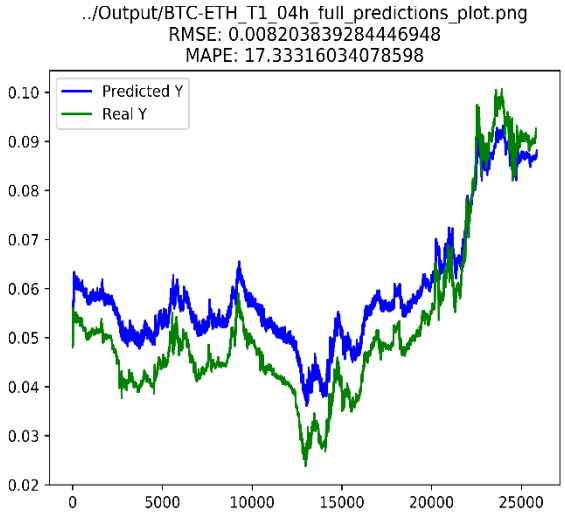
100 Epochs

Results: Bitcoin BTC-ETH exchange Time Series Prediction – 4 hours (Poloniex)

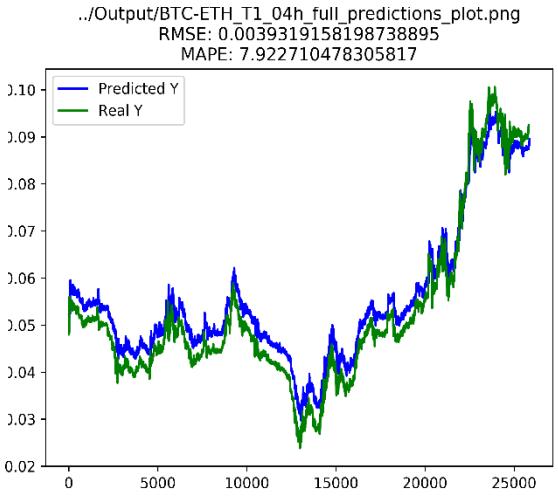
Test Set : 10%



1 Epoch

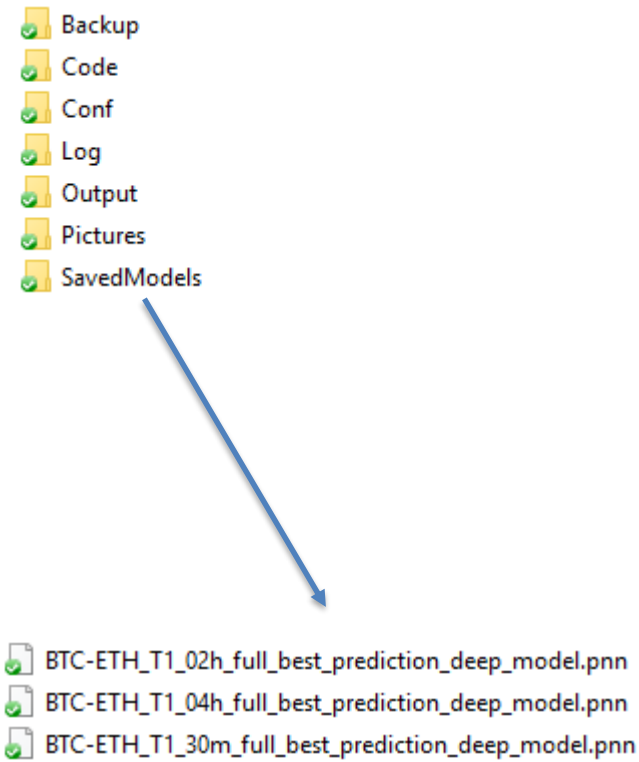


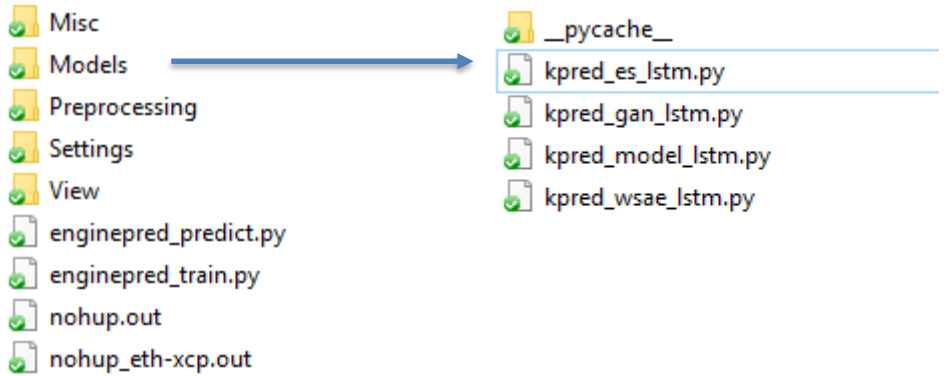
10 Epochs



100 Epochs

Anatomy of an advanced Text Classification Application by Deep Learning in Keras





Time Series Forecasting with Keras: Layers.Recurrent.RNN

RNN [source]

```
keras.layers.RNN(cell, return_sequences=False, return_state=False, go_backwards=False, stateful=False, unroll=Fals
```

Base class for recurrent layers.

Arguments

- **cell**: A RNN cell instance. A RNN cell is a class that has:
  - a `call(input_at_t, states_at_t)` method, returning `(output_at_t, states_at_t_plus_1)`. The call method of the cell can also take the optional argument `constants`, see section "Note on passing external constants" below.
  - a `state_size` attribute. This can be a single integer (single state) in which case it is the size of the recurrent state (which should be the same as the size of the cell output). This can also be a list/tuple of integers (one size per state).
  - a `output_size` attribute. This can be a single integer or a TensorShape, which represent the shape of the output. For backward compatible reason, if this attribute is not available for the cell, the value will be inferred by the first element of the `state_size`.

  It is also possible for `cell` to be a list of RNN cell instances, in which cases the cells get stacked on after the other in the RNN, implementing an efficient stacked RNN.
- **return_sequences**: Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **return_state**: Boolean. Whether to return the last state in addition to the output.
- **go_backwards**: Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.
- **stateful**: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch

Time Series Forecasting with Keras: Layers.Recurrent.RNN

- **unroll:** Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
- **input_dim:** dimensionality of the input (integer). This argument (or alternatively, the keyword argument `input_shape`) is required when using this layer as the first layer in a model.
- **input_length:** Length of input sequences, to be specified when it is constant. This argument is required if you are going to connect `Flatten` then `Dense` layers upstream (without it, the shape of the dense outputs cannot be computed). Note that if the recurrent layer is not the first layer in your model, you would need to specify the input length at the level of the first layer (e.g. via the `input_shape` argument)

Input shape

3D tensor with shape `(batch_size, timesteps, input_dim)`.

Output shape

- if `return_state`: a list of tensors. The first tensor is the output. The remaining tensors are the last states, each with shape `(batch_size, units)`.
- if `return_sequences`: 3D tensor with shape `(batch_size, timesteps, units)`.
- else, 2D tensor with shape `(batch_size, units)`.

Time Series Forecasting with Keras: Layers.Recurrent.SimpleRNN

SimpleRNN [source]

```
keras.layers.SimpleRNN(units, activation='tanh', use_bias=True, kernel_initializer='glorot_uniform', recurrent_ini
```

Fully-connected RNN where the output is to be fed back to input.

Arguments

units: Positive integer, dimensionality of the output space. activation: Activation function to use (see activations). Default: hyperbolic tangent (`tanh`). If you pass `None` , no activation is applied (ie. "linear" activation: `a(x) = x`). use_bias: Boolean, whether the layer uses a bias vector. kernel_initializer: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs (see initializers). recurrent_initializer: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state (see initializers). bias_initializer: Initializer for the bias vector (see initializers). kernel_regularizer: Regularizer function applied to the `kernel` weights matrix (see regularizer). recurrent_regularizer: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer). bias_regularizer: Regularizer function applied to the bias vector (see regularizer). activity_regularizer: Regularizer function applied to the output of the layer (its "activation"). (see regularizer). kernel_constraint: Constraint function applied to the `kernel` weights matrix (see constraints). recurrent_constraint: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints). bias_constraint: Constraint function applied to the bias vector (see constraints). dropout: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. recurrent_dropout: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. return_sequences: Boolean. Whether to return the last output in the output sequence, or the full sequence. return_state: Boolean. Whether to return the last state in addition to the output. go_backwards: Boolean (default False). If True, process the input sequence backwards and return the reversed sequence. stateful: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch. unroll:

Time Series Forecasting with Keras: Layers.Recurrent.GRU

GRU [source]

```
keras.layers.GRU(units, activation='tanh', recurrent_activation='hard_sigmoid', use_bias=True, kernel_initializer=
```

Gated Recurrent Unit - Cho et al. 2014.

There are two variants. The default one is based on 1406.1078v3 and has reset gate applied to hidden state before matrix multiplication. The other one is based on original 1406.1078v1 and has the order reversed.

The second variant is compatible with CuDNNGRU (GPU-only) and allows inference on CPU. Thus it has separate biases for `kernel` and `recurrent_kernel`. Use `'reset_after'=True` and `recurrent_activation='sigmoid'`.

Arguments

- **units:** Positive integer, dimensionality of the output space.
- **activation:** Activation function to use (see activations). Default: hyperbolic tangent (`tanh`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation:** Activation function to use for the recurrent step (see activations). Default: hard sigmoid (`hard_sigmoid`). If you pass `None`, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias:** Boolean, whether the layer uses a bias vector.
- **kernel_initializer:** Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs (see initializers).
- **recurrent_initializer:** Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state (see initializers).
- **bias_initializer:** Initializer for the bias vector (see initializers).
- **kernel_regularizer:** Regularizer function applied to the `kernel` weights matrix (see regularizer).

Time Series Forecasting with Keras: Layers.Recurrent.GRU

- **recurrent_regularizer:** Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer:** Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer:** Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint:** Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint:** Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint:** Constraint function applied to the bias vector (see constraints).
- **dropout:** Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout:** Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
- **implementation:** Implementation mode, either 1 or 2. Mode 1 will structure its operations as a larger number of smaller dot products and additions, whereas mode 2 will batch them into fewer, larger operations. These modes will have different performance profiles on different hardware and for different applications.
- **return_sequences:** Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **return_state:** Boolean. Whether to return the last state in addition to the output.
- **go_backwards:** Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.
- **stateful:** Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **unroll:** Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.
- **reset_after:** GRU convention (whether to apply reset gate after or before matrix multiplication). False = "before" (default), True = "after" (CuDNN compatible).

Time Series Forecasting with Keras: Layers.Recurrent.LSTM

LSTM [source]

```
keras.layers.LSTM(units, activation='tanh', recurrent_activation='hard_sigmoid', use_bias=True, kernel_initializer
```

Long Short-Term Memory layer - Hochreiter 1997.

Arguments

- **units:** Positive integer, dimensionality of the output space.
- **activation:** Activation function to use (see activations). Default: hyperbolic tangent (`tanh`). If you pass `None` , no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation:** Activation function to use for the recurrent step (see activations). Default: hard sigmoid (`hard_sigmoid`). If you pass `None` , no activation is applied (ie. "linear" activation: `a(x) = x`).
- **use_bias:** Boolean, whether the layer uses a bias vector.
- **kernel_initializer:** Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).
- **recurrent_initializer:** Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers).
- **bias_initializer:** Initializer for the bias vector (see initializers).
- **unit_forget_bias:** Boolean. If True, add 1 to the bias of the forget gate at initialization. Setting it to true will also force `bias_initializer="zeros"` . This is recommended in Jozefowicz et al. (2015).
- **kernel_regularizer:** Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer:** Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer:** Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer:** Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint:** Constraint function applied to the `kernel` weights matrix (see constraints).

Time Series Forecasting with Keras: Layers.Recurrent.LSTM

- **bias_constraint:** Constraint function applied to the bias vector (see constraints).
- **dropout:** Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout:** Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.
- **implementation:** Implementation mode, either 1 or 2. Mode 1 will structure its operations as a larger number of smaller dot products and additions, whereas mode 2 will batch them into fewer, larger operations. These modes will have different performance profiles on different hardware and for different applications.
- **return_sequences:** Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **return_state:** Boolean. Whether to return the last state in addition to the output.
- **go_backwards:** Boolean (default False). If True, process the input sequence backwards and return the reversed sequence.
- **stateful:** Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **unroll:** Boolean (default False). If True, the network will be unrolled, else a symbolic loop will be used. Unrolling can speed-up a RNN, although it tends to be more memory-intensive. Unrolling is only suitable for short sequences.

Time Series Forecasting with Keras: Layers.Recurrent.ConvLSTM2D

ConvLSTM2D [source]

```
keras.layers.ConvLSTM2D(filters, kernel_size, strides=(1, 1), padding='valid', data_format=None, dilation_rate=(1,
```

Convolutional LSTM.

It is similar to an LSTM layer, but the input transformations and recurrent transformations are both convolutional.

Arguments

- **filters:** Integer, the dimensionality of the output space (i.e. the number output of filters in the convolution).
- **kernel_size:** An integer or tuple/list of n integers, specifying the dimensions of the convolution window.
- **strides:** An integer or tuple/list of n integers, specifying the strides of the convolution. Specifying any stride value != 1 is incompatible with specifying any `dilation_rate` value != 1.
- **padding:** One of `"valid"` or `"same"` (case-insensitive).
- **data_format:** A string, one of `"channels_last"` (default) or `"channels_first"`. The ordering of the dimensions in the inputs. `"channels_last"` corresponds to inputs with shape `(batch, time, ..., channels)` while `"channels_first"` corresponds to inputs with shape `(batch, time, channels, ...)`. It defaults to the `image_data_format` value found in your Keras config file at `~/.keras/keras.json`. If you never set it, then it will be `"channels_last"`.
- **dilation_rate:** An integer or tuple/list of n integers, specifying the dilation rate to use for dilated convolution. Currently, specifying any `dilation_rate` value != 1 is incompatible with specifying any `strides` value != 1.
- **activation:** Activation function to use (see activations). If you don't specify anything, no activation is applied (ie. "linear" activation: `a(x) = x`).
- **recurrent_activation:** Activation function to use for the recurrent step (see activations).
- **use_bias:** Boolean, whether the layer uses a bias vector.
- **kernel_initializer:** Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers).

Time Series Forecasting with Keras: Layers.Recurrent.ConvLSTM2D

- **bias_initializer:** Initializer for the bias vector (see initializers).
- **unit_forget_bias:** Boolean. If True, add 1 to the bias of the forget gate at initialization. Use in combination with `bias_initializer="zeros"`. This is recommended in Jozefowicz et al. (2015).
- **kernel_regularizer:** Regularizer function applied to the `kernel` weights matrix (see regularizer).
- **recurrent_regularizer:** Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer).
- **bias_regularizer:** Regularizer function applied to the bias vector (see regularizer).
- **activity_regularizer:** Regularizer function applied to the output of the layer (its "activation"). (see regularizer).
- **kernel_constraint:** Constraint function applied to the `kernel` weights matrix (see constraints).
- **recurrent_constraint:** Constraint function applied to the `recurrent_kernel` weights matrix (see constraints).
- **bias_constraint:** Constraint function applied to the bias vector (see constraints).
- **return_sequences:** Boolean. Whether to return the last output in the output sequence, or the full sequence.
- **go_backwards:** Boolean (default False). If True, process the input sequence backwards.
- **stateful:** Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.
- **dropout:** Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs.
- **recurrent_dropout:** Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state.

Time Series Forecasting with Keras: Layers.Recurrent.CuDNNGRU

CuDNNGRU [source]

```
keras.layers.CuDNNGRU(units, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initial
```

Fast GRU implementation backed by CuDNN.

Can only be run on GPU, with the TensorFlow backend.

Arguments

units: Positive integer, dimensionality of the output space. kernel_initializer: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers). recurrent_initializer: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers). bias_initializer: Initializer for the bias vector (see initializers). kernel_regularizer: Regularizer function applied to the `kernel` weights matrix (see regularizer). recurrent_regularizer: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer). bias_regularizer: Regularizer function applied to the bias vector (see regularizer). activity_regularizer: Regularizer function applied to the output of the layer (its "activation"). (see regularizer). kernel_constraint: Constraint function applied to the `kernel` weights matrix (see constraints). recurrent_constraint: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints). bias_constraint: Constraint function applied to the bias vector (see constraints). return_sequences: Boolean. Whether to return the last output. in the output sequence, or the full sequence. return_state: Boolean. Whether to return the last state in addition to the output. stateful: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.

Time Series Forecasting with Keras: Layers.Recurrent.CuDNNLSTM

CuDNNLSTM [source]

```
keras.layers.CuDNNLSTM(units, kernel_initializer='glorot_uniform', recurrent_initializer='orthogonal', bias_initia
```

Fast LSTM implementation with CuDNN.

Can only be run on GPU, with the TensorFlow backend.

Arguments

units: Positive integer, dimensionality of the output space. kernel_initializer: Initializer for the `kernel` weights matrix, used for the linear transformation of the inputs. (see initializers). unit_forget_bias: Boolean. If True, add 1 to the bias of the forget gate at initialization. Setting it to true will also force `bias_initializer="zeros"`. This is recommended in Jozefowicz et al. (2015). recurrent_initializer: Initializer for the `recurrent_kernel` weights matrix, used for the linear transformation of the recurrent state. (see initializers). bias_initializer: Initializer for the bias vector (see initializers). kernel_regularizer: Regularizer function applied to the `kernel` weights matrix (see regularizer). recurrent_regularizer: Regularizer function applied to the `recurrent_kernel` weights matrix (see regularizer). bias_regularizer: Regularizer function applied to the bias vector (see regularizer). activity_regularizer: Regularizer function applied to the output of the layer (its "activation"). (see regularizer). kernel_constraint: Constraint function applied to the `kernel` weights matrix (see constraints). recurrent_constraint: Constraint function applied to the `recurrent_kernel` weights matrix (see constraints). bias_constraint: Constraint function applied to the bias vector (see constraints). return_sequences: Boolean. Whether to return the last output. in the output sequence, or the full sequence. return_state: Boolean. Whether to return the last state in addition to the output. stateful: Boolean (default False). If True, the last state for each sample at index i in a batch will be used as initial state for the sample of index i in the following batch.

Exercise 2: Multivariate Time-Series Forecasting in Keras

1. Air Pollution Forecasting

In this tutorial, we are going to use the Air Quality dataset.

This is a dataset that reports on the weather and the level of pollution each hour for five years at the US embassy in Beijing, China.

The data includes the date-time, the pollution called PM2.5 concentration, and the weather information including dew point, temperature, pressure, wind direction, wind speed and the cumulative number of hours of snow and rain. The complete feature list in the raw data is as follows:

1. **No**: row number
2. **year**: year of data in this row
3. **month**: month of data in this row
4. **day**: day of data in this row
5. **hour**: hour of data in this row
6. **pm2.5**: PM2.5 concentration
7. **DEWP**: Dew Point
8. **TEMP**: Temperature
9. **PRES**: Pressure
10. **cbwd**: Combined wind direction
11. **Iws**: Cumulated wind speed
12. **Is**: Cumulated hours of snow
13. **Ir**: Cumulated hours of rain

https://raw.githubusercontent.com/jbrownlee/Datasets/master/pollution.csv

```
No,year,month,day,hour,pm2.5,DEWP,TEMP,PRES,cbwd,Iws,Is,Ir
1,2010,1,1,0,NA,-21,-11,1021,NW,1.79,0,0
2,2010,1,1,1,NA,-21,-12,1020,NW,4.92,0,0
3,2010,1,1,2,NA,-21,-11,1019,NW,6.71,0,0
4,2010,1,1,3,NA,-21,-14,1019,NW,9.84,0,0
5,2010,1,1,4,NA,-20,-12,1018,NW,12.97,0,0
6,2010,1,1,5,NA,-19,-10,1017,NW,16.1,0,0
7,2010,1,1,6,NA,-19,-9,1017,NW,19.23,0,0
8,2010,1,1,7,NA,-19,-9,1017,NW,21.02,0,0
9,2010,1,1,8,NA,-19,-9,1017,NW,24.15,0,0
10,2010,1,1,9,NA,-20,-8,1017,NW,27.28,0,0
11,2010,1,1,10,NA,-19,-7,1017,NW,31.3,0,0
```

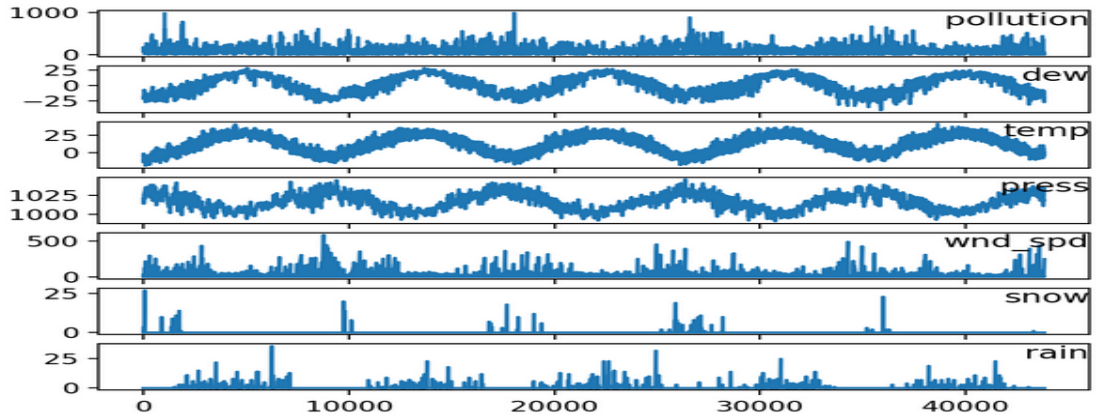
Time-Series: A pollution predictor with Keras – PM2.5 concentration

```
def pollution_prepare_data(pred_dataset_path='', pred_dataset_file_in='', pred_dataset_file_out=''):
    # load data
    def parse(x):
        return datetime.strptime(x, '%Y %m %d %H')
    dataset = read_csv(pred_dataset_path+'/'+pred_dataset_file_in,  parse_dates = [['year', 'month', 'day', 'hour']], index_col=0, date_parser=parse)
    dataset.drop('No', axis=1, inplace=True)
    # manually specify column names
    dataset.columns = ['pollution', 'dew', 'temp', 'press', 'wnd_dir', 'wnd_spd', 'snow', 'rain']
    dataset.index.name = 'date'
    # mark all NA values with 0
    dataset['pollution'].fillna(0, inplace=True)
    # drop the first 24 hours
    dataset = dataset[24:]
    # summarize first 5 rows
    print(dataset.head(5))
    # save to file
    dataset.to_csv(pred_dataset_path+'/'+pred_dataset_file_out)
```

	pollution	dew	temp	press	wnd_dir	wnd_spd	snow	rain
date								
2010-01-02 00:00:00	129.0	-16	-4.0	1020.0	SE	1.79	0	0
2010-01-02 01:00:00	148.0	-15	-4.0	1020.0	SE	2.68	0	0
2010-01-02 02:00:00	159.0	-11	-5.0	1021.0	SE	3.57	0	0
2010-01-02 03:00:00	181.0	-7	-5.0	1022.0	SE	5.36	1	0
2010-01-02 04:00:00	138.0	-7	-5.0	1022.0	SE	6.25	2	0

Time-Series: View Model

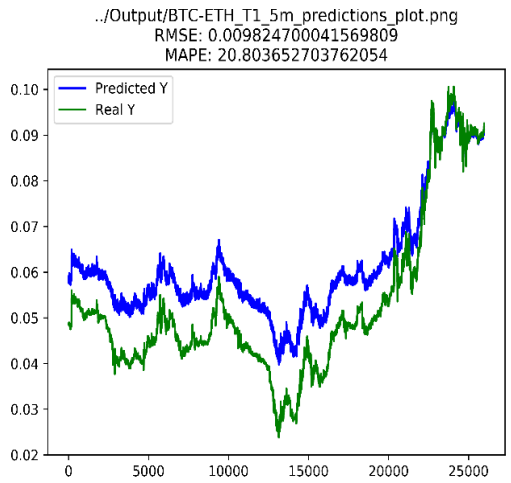
```
# Plot inputs
def plot_dataset(pred_dataset_path='', pred_dataset_file=''):
    # load dataset
    dataset = read_csv(pred_dataset_path+'/'+pred_dataset_file, header=0, index_col=0)
    values = dataset.values
    # specify columns to plot
    groups = [0, 1, 2, 3, 5, 6, 7]
    i = 1
    # plot each column
    pyplot.figure()
    for group in groups:
        pyplot.subplot(len(groups), 1, i)
        pyplot.plot(values[:, group])
        pyplot.title(dataset.columns[group], y=0.5, loc='right')
        i += 1
    pyplot.show(block=False)
    pyplot.pause(1)
```



Time-Series: View Model

```
# Plot the loss function chart
def plot_loss(history = None):
    pyplot.figure()                                          # generate a new window
    pyplot.plot(history['loss'], label='train')
    pyplot.plot(history['val_loss'], label='test')
    pyplot.legend()
    pyplot.show(block=False)
    pyplot.pause(1)

# Plot predictions and real y chart
def plot_predictions(predictions_view_path_file, rmse, mape, inv_yhat = None, inv_y = None, save = False, pause_time = 1):
    pyplot.figure()                                          # generate a new window
    pyplot.plot(inv_yhat, label='Predicted Y', color = 'blue')
    pyplot.plot(inv_y, label='Real Y', color = 'green')
    pyplot.legend()
    pyplot.title(predictions_view_path_file+"\nRMSE: "+str(rmse)+'\nMAPE: '+str(mape)
    pyplot.savefig(predictions_view_path_file, dpi = 600)
    pyplot.show(block=False)
    pyplot.pause(pause_time)
```



Multivariate LSTM Forecast Model

- Neural networks like Long Short-Term Memory (LSTM) recurrent neural networks are able to almost seamlessly model problems with multiple input variables.
- This is a great benefit in time series forecasting, where classical linear methods can be difficult to adapt to multivariate or multiple input forecasting problems.

```
class PredModelLSTM:
    @staticmethod
    def build(input_length, vector_dim, output_size, lstm_n_hiddens, summary):

        # initialize the model
        deepnetwork = Sequential()
        deepnetwork.add(LSTM(lstm_n_hiddens, input_shape = (input_length, vector_dim)))
        deepnetwork.add(Dense(output_size))

        if summary==True:
            deepnetwork.summary()

        return deepnetwork
```

Data Preparation : From Time-Series a Supervised Problem

```
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    n_vars = 1 if type(data) is list else data.shape[1]
    df = DataFrame(data)
    cols, names = list(), list()
    # input sequence (t-n, ... t-1)
    for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
    # forecast sequence (t, t+1, ... t+n)
    for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
    # put it all together
    agg = concat(cols, axis=1)
    agg.columns = names
    # drop rows with NaN values
    if dropnan:
        agg.dropna(inplace=True)
    return agg
```

Data Preparation : From Time-Series a Supervised Problem

```
0
1
2
3
4
5
6
7
8
9
```

```
X,  y
1   2
2,  3
3,  4
4,  5
5,  6
6,  7
7,  8
8,  9
```

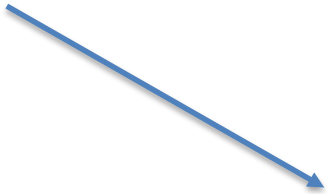
Data Preparation : Pandas shift function

```
1 from pandas import DataFrame
2 df = DataFrame()
3 df['t'] = [x for x in range(10)]
4 df['t-1'] = df['t'].shift(1)
5 print(df)
```

```
    t  t-1
0   0  NaN
1   1  0.0
2   2  1.0
3   3  2.0
4   4  3.0
5   5  4.0
6   6  5.0
7   7  6.0
8   8  7.0
9   9  8.0
```

Data Preparation : Multi-Step Univariate Analysis

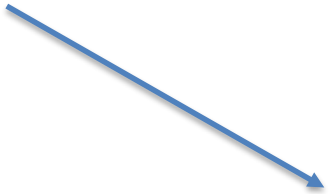
```
data = series_to_supervised(values, 3)
```



	var1(t-3)	var1(t-2)	var1(t-1)	var1(t)
3	0.0	1.0	2.0	3
4	1.0	2.0	3.0	4
5	2.0	3.0	4.0	5
6	3.0	4.0	5.0	6
7	4.0	5.0	6.0	7
8	5.0	6.0	7.0	8
9	6.0	7.0	8.0	9

Data Preparation : Multi-Step or Sequence Forecasting

```
1 data = series_to_supervised(values, 2, 2)
```



		var1(t-2)	var1(t-1)	var1(t)	var1(t+1)
2	2	0.0	1.0	2	3.0
3	3	1.0	2.0	3	4.0
4	4	2.0	3.0	4	5.0
5	5	3.0	4.0	5	6.0
6	6	4.0	5.0	6	7.0
7	7	5.0	6.0	7	8.0
8	8	6.0	7.0	8	9.0

Data Preparation : Multivariate Forecasting

		var1(t-1)	var2(t-1)	var1(t)	var2(t)
2	1	0.0	50.0	1	51
3	2	1.0	51.0	2	52
4	3	2.0	52.0	3	53
5	4	3.0	53.0	4	54
6	5	4.0	54.0	5	55
7	6	5.0	55.0	6	56
8	7	6.0	56.0	7	57
9	8	7.0	57.0	8	58
10	9	8.0	58.0	9	59

		var1(t-1)	var2(t-1)	var1(t)	var2(t)	var1(t+1)	var2(t+1)
2	1	0.0	50.0	1	51	2.0	52.0
3	2	1.0	51.0	2	52	3.0	53.0
4	3	2.0	52.0	3	53	4.0	54.0
5	4	3.0	53.0	4	54	5.0	55.0
6	5	4.0	54.0	5	55	6.0	56.0
7	6	5.0	55.0	6	56	7.0	57.0
8	7	6.0	56.0	7	57	8.0	58.0
9	8	7.0	57.0	8	58	9.0	59.0

Data Preparation : Data Encoding and Normalization

```
# load dataset
dataset = read_csv('pollution.csv', header=0, index_col=0)
values = dataset.values
# integer encode direction
encoder = LabelEncoder()
values[:,4] = encoder.fit_transform(values[:,4])
# ensure all data is float
values = values.astype('float32')
# normalize features
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(values)
# frame as supervised learning
reframed = series_to_supervised(scaled, 1, 1)
# drop columns we don't want to predict
reframed.drop(reframed.columns[[9,10,11,12,13,14,15]], axis=1, inplace=True)
print(reframed.head())
```

Data Preparation : Data splitting and scaling

```
   var1(t-1)  var2(t-1)  var3(t-1)  var4(t-1)  var5(t-1)  var6(t-1)  \
1   0.129779   0.352941   0.245902   0.527273   0.666667   0.002290
2   0.148893   0.367647   0.245902   0.527273   0.666667   0.003811
3   0.159960   0.426471   0.229508   0.545454   0.666667   0.005332
4   0.182093   0.485294   0.229508   0.563637   0.666667   0.008391
5   0.138833   0.485294   0.229508   0.563637   0.666667   0.009912

   var7(t-1)  var8(t-1)   var1(t)
1   0.000000        0.0  0.148893
2   0.000000        0.0  0.159960
3   0.000000        0.0  0.182093
4   0.037037        0.0  0.138833
5   0.074074        0.0  0.109658
```

```python
# split into train and test sets
values = reframed.values
n_train_hours = 365 * 24
train = values[:n_train_hours, :]
test = values[n_train_hours:, :]
# split into input and outputs
train_X, train_y = train[:, :-1], train[:, -1]
test_X, test_y = test[:, :-1], test[:, -1]
# reshape input to be 3D [samples, timesteps, features]
train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
test_X = test_X.reshape((test_X.shape[0], 1, test_X.shape[1]))
print(train_X.shape, train_y.shape, test_X.shape, test_y.shape)
```

Model Fitting and Prediction

```
# fit network
history = model.fit(train_X, train_y, epochs=50, batch_size=72, validation_data=(t
# plot history
pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')
pyplot.legend()
pyplot.show()

# make a prediction
yhat = model.predict(test_X)
test_X = test_X.reshape((test_X.shape[0], test_X.shape[2]))
# invert scaling for forecast
inv_yhat = concatenate((yhat, test_X[:, 1:]), axis=1)
inv_yhat = scaler.inverse_transform(inv_yhat)
inv_yhat = inv_yhat[:,0]
# invert scaling for actual
test_y = test_y.reshape((len(test_y), 1))
inv_y = concatenate((test_y, test_X[:, 1:]), axis=1)
inv_y = scaler.inverse_transform(inv_y)
inv_y = inv_y[:,0]
# calculate RMSE
rmse = sqrt(mean_squared_error(inv_y, inv_yhat))
print('Test RMSE: %.3f' % rmse)
```

REFERENCES

Bao, W., Yue, J., & Rao, Y. (2017). A deep learning framework for financial time series using stacked autoencoders and long-short term memory. PloS one, 12(7), e0180944.

Bodyanskiy Y, Popov S. (2006) Neural network approach to forecasting of quasiperiodic financial time series. Eur J Oper Res. 175(3):1357–66.

Nourani V, Komasi M, Mano A. (2009) A multivariate ANN-wavelet approach for rainfall-runoff modeling. Water Resources Management. 2009;23(14):2877.

Kim TY, Oh KJ, Kim C, Do JD. (2004) Artificial neural networks for non-stationary time series. Neurocomputing.

Hsieh TJ, Hsiao HF, Yeh WC. (2011) Forecasting stock markets using wavelet transforms and recurrent neural networks: An integrated system based on artificial bee colony algorithm. Applied Soft Computing. 2011;11(2):2510–25. 61(C):439–47.

REFERENCES

Sutskever I, Hinton GE (2008). Deep, narrow sigmoid belief networks are universal approximators. Neural Computation. 20(11):2629–36. pmid:18533819

Roux NL, Bengio Y. (2010) Deep Belief Networks Are Compact Universal Approximators. Neural Computation.22(8):2192–207.

.

Bengio Y, Lamblin P, Popovici D, Larochelle H. (2007) Greedy layer-wise training of deep networks. Advances in neural information processing systems. 19:153.

Ramsey JB. (1999) The contribution of wavelets to the analysis of economic and financial data. Philosophical Transactions of the Royal Society B Biological Sciences. 357(357):2593–606.

Palangi H, Ward R, Deng L. (2016) Distributed Compressive Sensing: A Deep Learning Approach. IEEE Transactions on Signal Processing. 64(17):4504–18.

Amari, S. I., Cichocki, A., & Yang, H. H. (1995, December). Recurrent neural networks for blind separation of sources. In *Proc. Int. Symp. NOLTA* (pp. 37-42).

REFERENCES

Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

Hochreiter S, Schmidhuber J. Long Short-Term Memory (1997). Neural Computation. 1997;9(8):1735–80. pmid:9377276

Bliemel F. Theil's (1973) Forecast Accuracy Coefficient: A Clarification. Journal of Marketing Research. 10(4):444.

AKNOWLEDGEMENTS

THANK YOU

FOR YOUR ATTENTION

Francesco Pugliese