ieeta instituto de engenharia electrónica e telemática de aveiro

universidade de aveiro

Departamento de Eletrónica, Telecomunicações e Informática

LECTURE 11 Deep Learning
SEQUENCE MODELS & TIME SERIES FORECASTING

Petia Georgieva
(petia@ua.pt)

universidade de aveiro

Outline

1. Sequence models – motivation
2. Recurrent Neural Networks (RNN)
3. Backpropagation through time
4. Long-Short Term Memory (LSTM)
5. Time Series Forecasting

Examples of sequence data

	x (input)	y (output)
Speech recognition		**==> "The quick brown fox jumped over the lazy dog."**
Sentiment classification	**"There is nothing to like in this movie."**	**==>** 
DNA sequence analysis	**AGCCCCTGTGAGGAACTAG**	**==> AGCCCCTGTGAGGAACTAG**
Machine translation	**Voulez-vous chanter avec moi?**	**==> Do you want to sing with me?**
Video activity recognition		**==> Running**
Name entity recognition	**Yesterday, Harry Potter met Hermione Granger.**	**==> Yesterday, Harry Potter met Hermione Granger.**

x and y are both sequences, or only x is a sequence, or only y is a sequence.

Sequence Model Notation

Name Entity Recognition application is to find people's names, companies names, locations, countries names, currency names, etc. in text.

Given an input sequence of words (X), the sequence model has to automatically tell where are the proper names in this sentence.

x: Harry Potter and Hermione Granger invented a new spell.

$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ ... $x^{<9>}$

For this exampe the sequence length is 9 words (features) **=> Tx=9**

Target output y is binary vector with same length as the input
(1 if the word is name; 0 if not)

y= [1 1 0 1 1 0 0 0 0]
$y^{<1>}$ $y^{<2>}$ $y^{<3>}$ ..... $y^{<t>}$ $y^{<Ty>}$

Ty=9

In this problem every input $x^{<i>}$ has an output $y^{<i>}$ => **length Ty=Tx**
Each sentence (example) can have different sequence length.

NLP - representing individual words

Natural Language Processing (NLP). Create vocabulary or download existing dictionary.
For modern NLP, 10000 words is a small dictionary, 30-50 thousand is more common.
Large internet companies use 1 million words dictionary.
Each word is represented by a binary vector with # elements = dimension of dictionary where only the index of the word in the dictionary =1, all others are 0 (one-hot vector).

word index (in dictionary) **one-hot encoding**

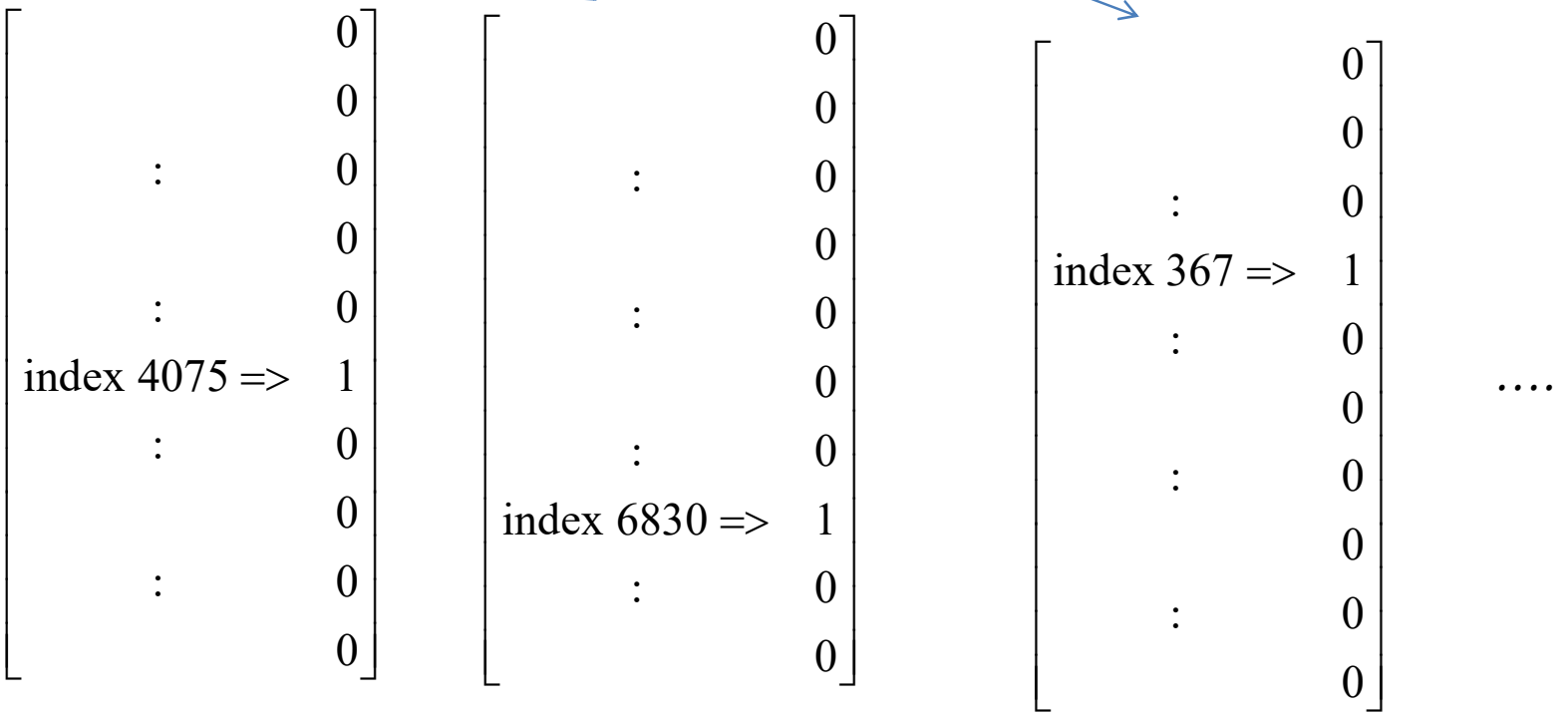
$$\begin{bmatrix} a => & 1 \\ aaron => & 2 \\ \vdots & \\ and => & 367 \\ \vdots & \\ Harry => & 4075 \\ \vdots & \\ Potter => & 6830 \\ \vdots & \\ zulu => & 10000 \end{bmatrix}$$

$$Harry = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

NLP – one hot encoding

x: Harry Potter and Hermione Granger invented a new spell.

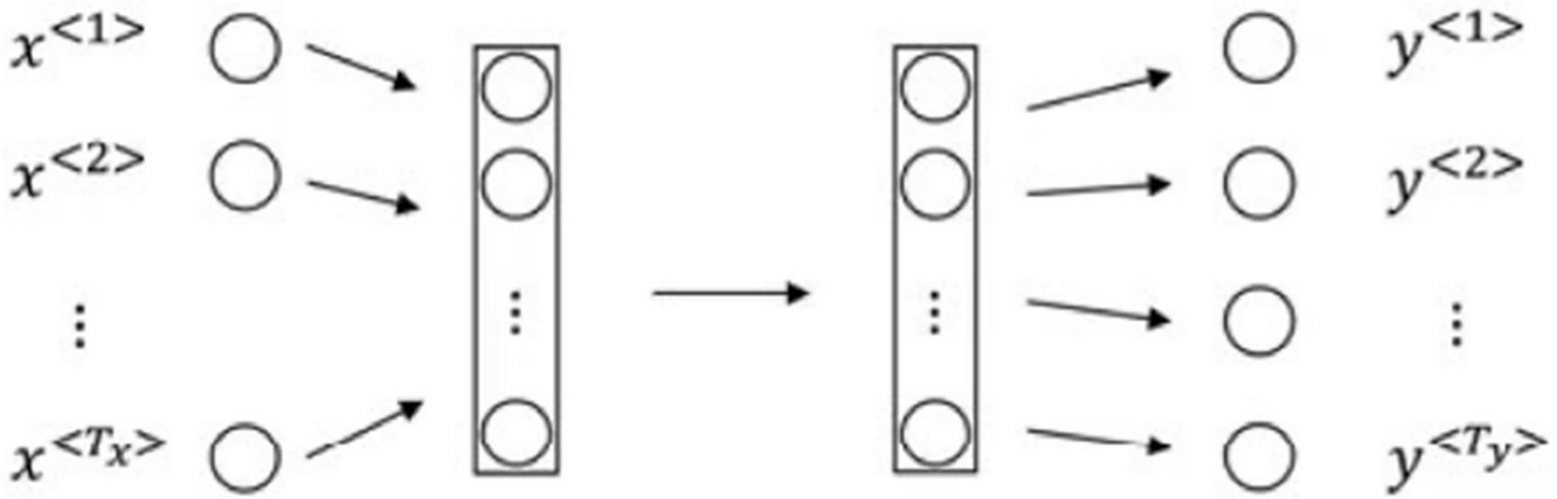
$x^{<1>}$ $x^{<2>}$ $x^{<3>}$ … $x^{<9>}$



<unk> - notation for words not in the dictionary. It can be added to encode all missing words that may appear in sentences.

The problem is formulated as supervised learning with labelled data (x,y).

Why not a standard neural network?



Problems:

1) Inputs and outputs can have different lengths in different examples.

2) Doesn't share features learned across different positions of text.

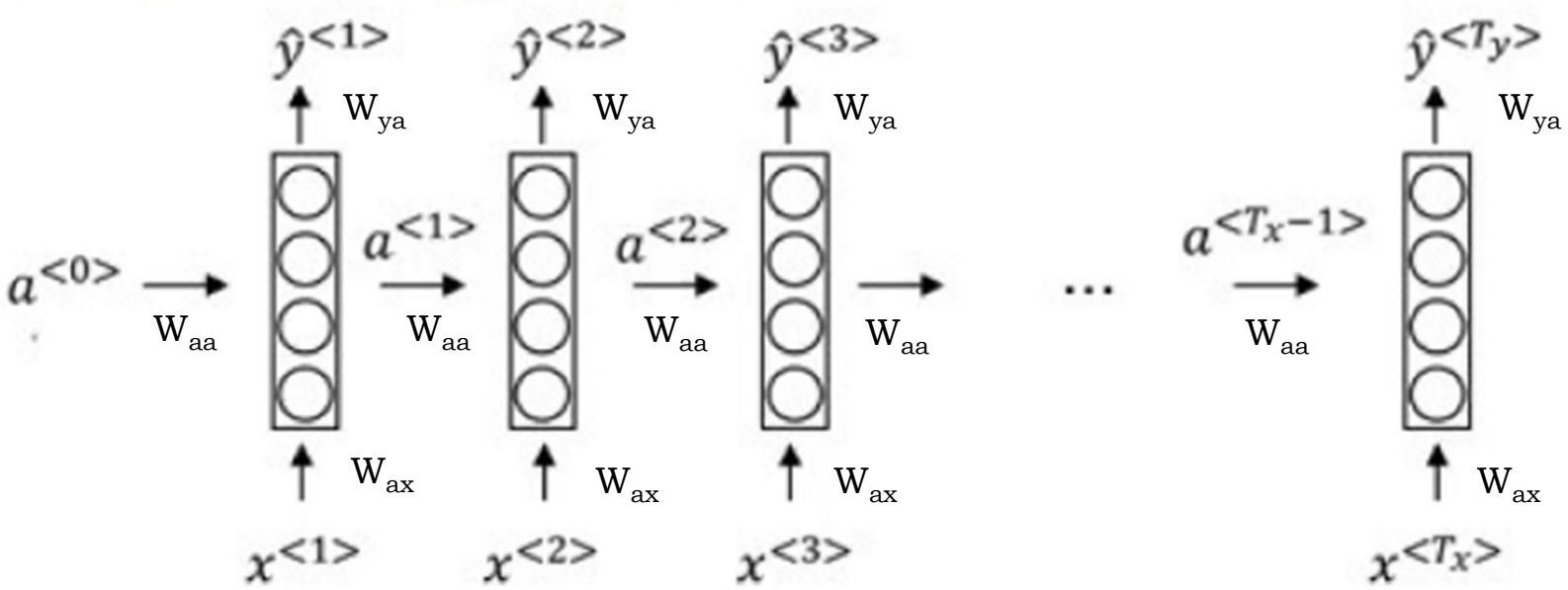
3) High input dimension (e.g. Tx* dimension of dictionary) => too many trainable parameters.

Recurrent Neural Networks (RNN) – better solution

Recurrent Neural Networks (RNN)

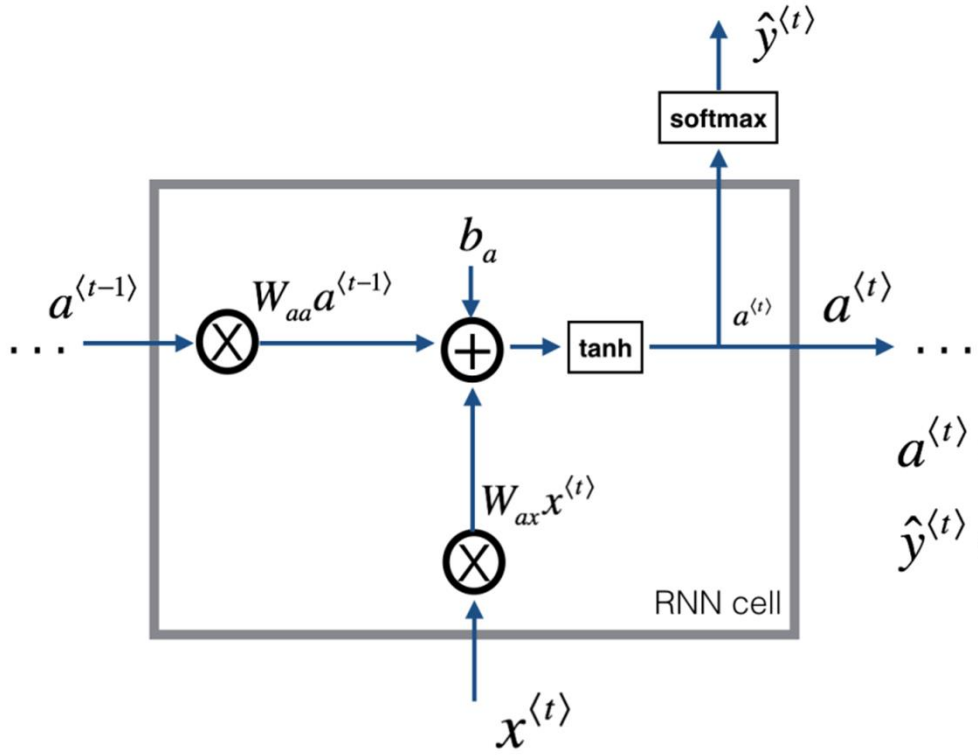
Read the sentence word by word (one time step per word).
Activation produced by 1st word is taken into account when read 2nd word.
Notation: inputs -$x^{<t>}$; outputs - $y^{<t>}$; hidden states - $a^{<t>}$
$a^{<0>}$ - initial state at time step 0, usually vector of zeros.
Previous results are passed as inputs => we get context !
A weakness of this type of uni-directional RNN is that the prediction at a certain time uses inf. that is earlier in the sequence but not latter.
The Bidirectional RNN (BRNN) overcome this problem.

Flow chart (unrolled/unfolded RNN diagram representation) :



Basic RNN unit

Takes $x^{<t>}$ (current input) and $a^{\langle t-1 \rangle}$ (activation from previous step) as inputs, and computes $a^{\langle t \rangle}$ (current activation). $a^{\langle t \rangle}$ is then used to predict $y^{\langle t \rangle}$ (current output) and passed forward to the next RNN unit (next time step). W_{aa}, W_{ax}, W_{ya}, b_a, b_y – the same RNN weights used in all time steps.

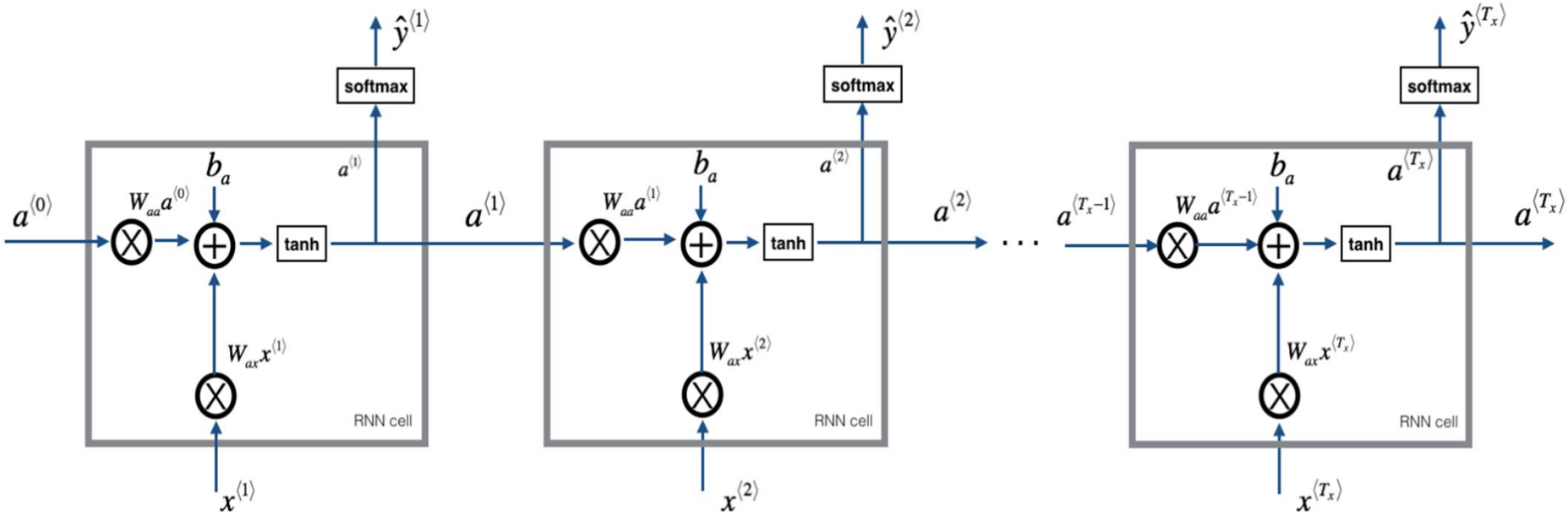


$$a^{\langle t \rangle} = \tanh(W_{ax}x^{\langle t \rangle} + W_{aa}a^{\langle t-1 \rangle} + b_a)$$

$$\hat{y}^{\langle t \rangle} = soft\max(W_{ya}a^{\langle t \rangle} + b_y)$$

RNN forward pass

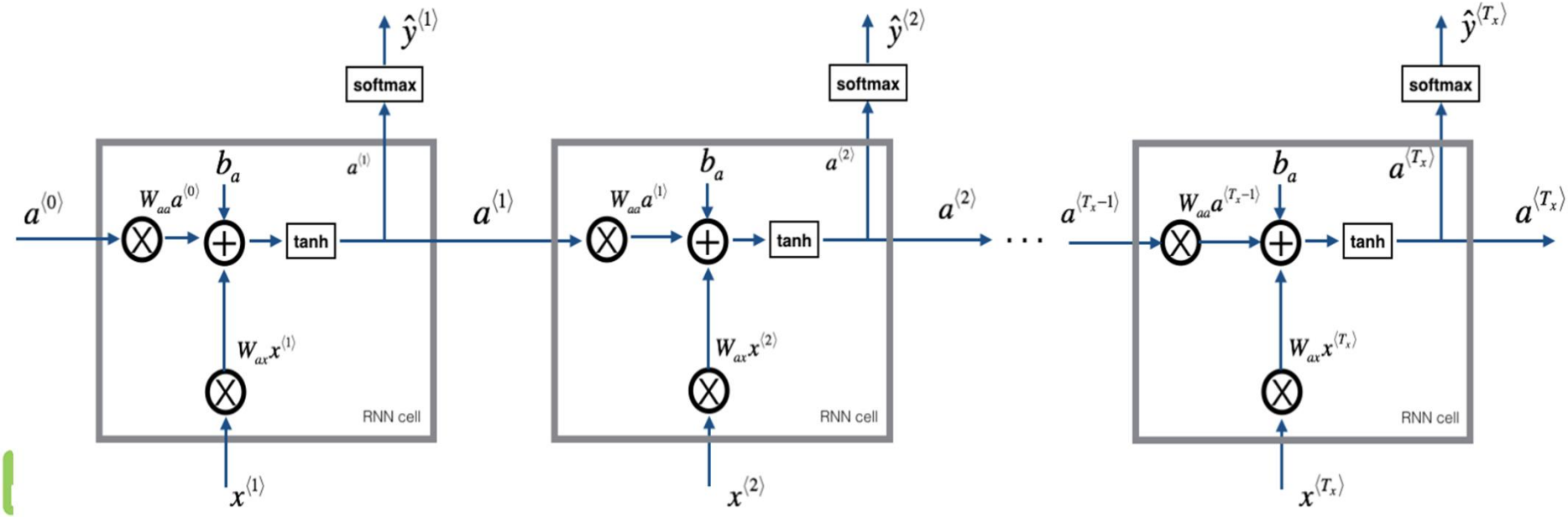
RNN is a sequence of basic RNN cells.
If input sequence is x=[$x^{<1>}$, $x^{<2>}$,...$x^{<Tx>}$] => RNN cell is copied T_x times.
RNN outputs y=[$y^{<1>}$, $y^{<2>}$,...$y^{<Tx>}$]



Backpropagation Through Time (BPTT)

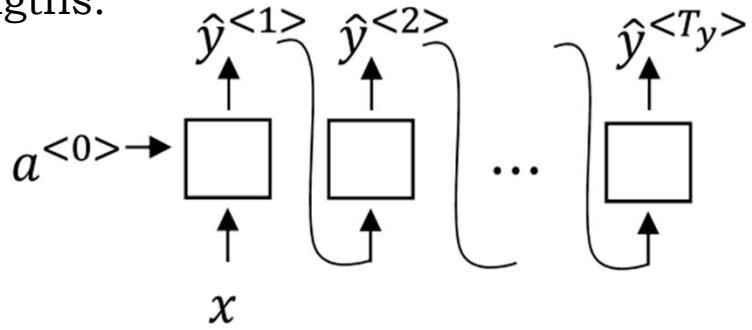
Forward propagation (for-prop): from time step 1 to time step Tx (from left to right) => compute RNN predictions.
Compute the cost (loss) function for each output $y^{\langle t \rangle}$ ($J^{\langle t \rangle}$)
Compute the sum of all cost (loss) functions (*J_all)*
Backward propagation (back-prop) : to update RNN parameters compute the gradients of *J_all*, starting from the last time step $y^{<Tx>}$ and going back through the previous time steps down to the first time step $y^{<1>}$ (from right to left) => This is called *Backpropagation Through Time (BPTT).*

The programing framework usually computes BPTT automatically.

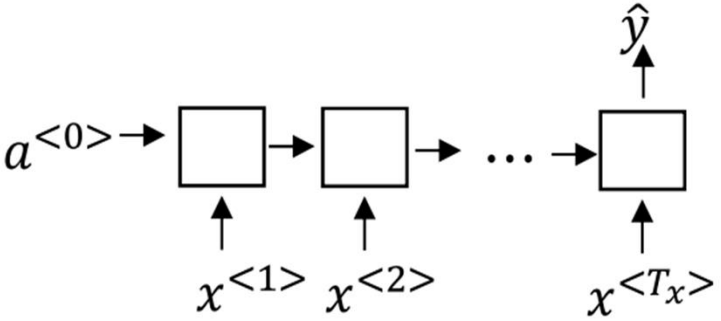


Different Types of RNN

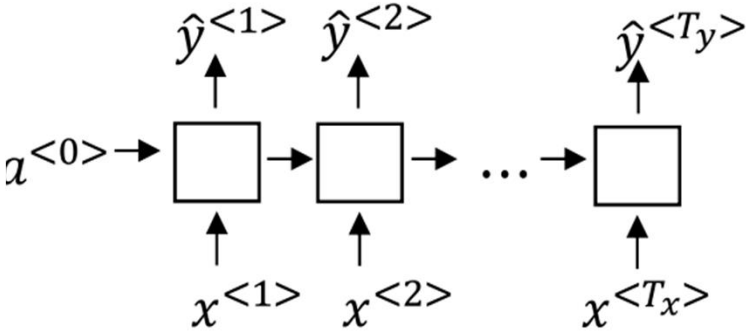
The input X and Y can be of many types and they do not have to be of the same lengths.



One to many (e.g. Music generation)

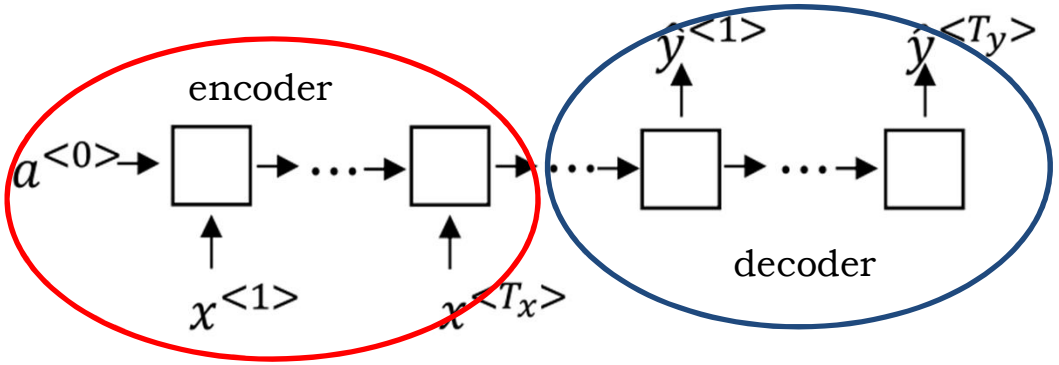


Many to one (e.g. Sentiment analysis)



Many to many

Tx=Ty (e.g. Name entity rec.)



Many to many

Tx different from Ty (e.g. machine translation)

Note: Time series forecasting (many to one, or many to many)

RNN – vanishing/exploding gradients

RNN works well when each output $y^{\langle t \rangle}$ can be estimated using "local" context (i.e. inf from inputs $x^{\langle t' \rangle}$ where t' is not too far from t).

Why ? – because RNN's suffers from vanishing gradient (similar to Deep NN).

Vanishing gradient - gradient values become very small. Layers that get small gradients stops learning. Those are usually the earlier layers in the sequence model. Because these layers don't learn, RNN's can forget what it seen in longer sequences, thus having a short-term memory.

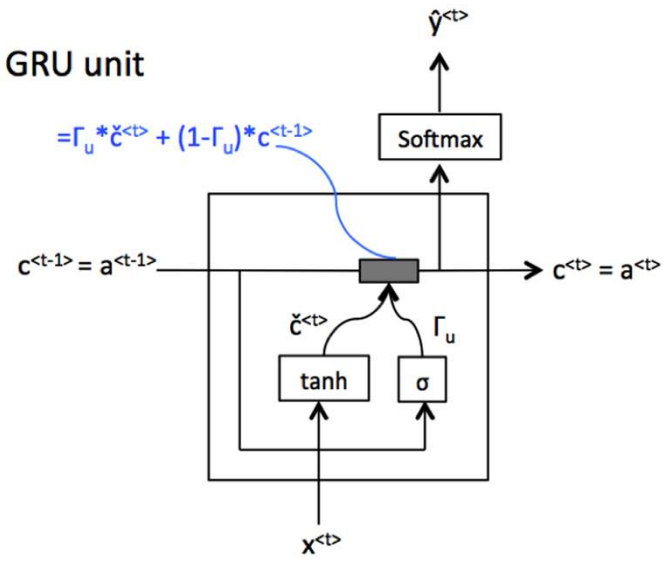
Standard RNN is not very good in capturing long-range dependencies, ex.:

*The **girl** that entered the coffee shop with many friends **was** happy.*
*The **girls** that entered the coffee shop with many friends **were** happy.*

Need to remember "**girl-was**" or "**girls-were**".

Long Short-Term Memory * (LSTM) and **Gated Recurrent Units** (GRU) – Solution for vanishing gradients.

Gated Recurrent Unit (basic architecture)



c - memory cell

$c^{<t>} = a^{<t>}$ (for LSTM they are different)

GRU outputs activation value = memory cell

$\tilde{c}^{<t>} = \tanh\left(w_c\left[c^{<t-1>}, x^{<t>}\right] + b_c\right)$ – candidate

$\Gamma_u = \sigma\left(w_u\left[c^{<t-1>}, x^{<t>}\right] + b_u\right)$ – update gate

$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + (1 - \Gamma_u) * c^{<t-1>}$ – update memory cell

Γ_u => values between (0,1)

if Γ_u => 0, don't update $c^{<t>}$, if Γ_u => 1, update $c^{<t>}$.

Γ_u=1 Γ_u=0 Γ_u=0 Γ_u=0 …… Γ_u=1

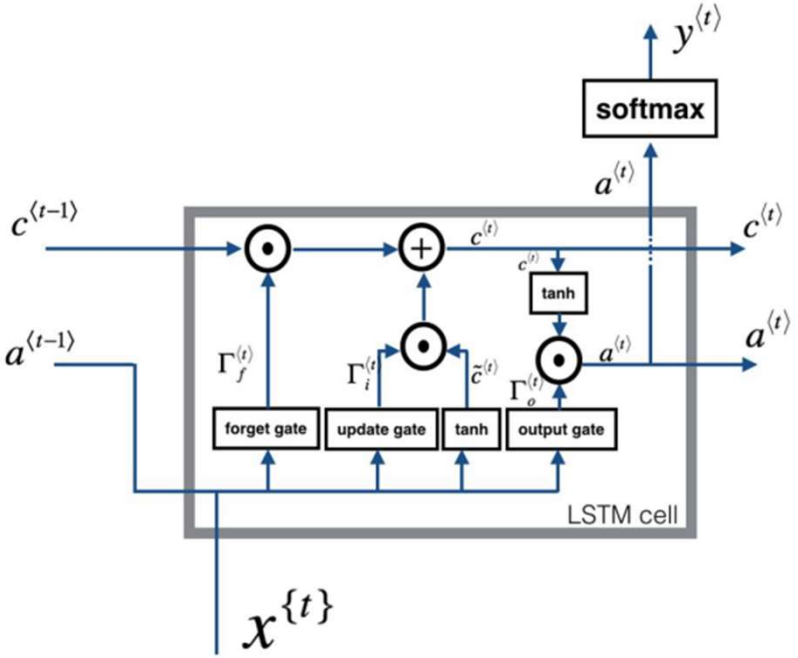
$c^{<t>}$= 1, keeps the same value of $c^{<t>}$ long time …….

Ex. The ***girl*** *that entered the coffee shop with many friends* ***was*** *happy.*

Memory cell c memorises if the girl was singular or plural even if **girl-was** are separated with many many words. Efficient against vanishing gradients.

$c^{<t>}$ and Γ_u are vectors (e.g. 100 elements of mostly 0 and 1, bits to update)

Long Short-Term Memory (LSTM) unit



$$\tilde{c}^{<t>} = \tanh\left(W_c\left[a^{<t-1>}, x^{<t>}\right] + b_c\right) - \text{candidate}$$

$$\Gamma_u = \sigma\left(W_u\left[a^{<t-1>}, x^{<t>}\right] + b_u\right) - \text{update gate}$$

$$\Gamma_f = \sigma\left(W_f\left[a^{<t-1>}, x^{<t>}\right] + b_f\right) - \text{forget gate}$$

$$\Gamma_o = \sigma\left(W_o\left[a^{<t-1>}, x^{<t>}\right] + b_o\right) - \text{output gate}$$

$$c^{<t>} = \Gamma_u * \tilde{c}^{<t>} + \Gamma_f * c^{<t-1>} - \text{update memory cell}$$

$$a^{<t>} = \Gamma_o * \tanh\left(c^{<t>}\right) - \text{activation}$$

LSTM outputs both activation value and memory cell

Gates (Γ_f , Γ_u , Γ_o)contain sigmoid activations => values between 0 and 1.

Forget gate decides what is relevant to keep from prior steps.
Inf from the previous hidden state and from the current input is passed through the sigmoid function. Output values between 0 and 1.
Closer to 0 means to forget, closer to 1 means to keep.

Update gate decides what inf is relevant to add from the current step.

Output gate determines what the next hidden state should be.

GRU & LSTM

LSTM and GRU are two variations of RNNs to capture better long range dependencies (connections) in sequences.

They mitigate short-term memory using mechanisms called memory cell and gates.

Gates remember (keep saved) bits of inf for many time steps.

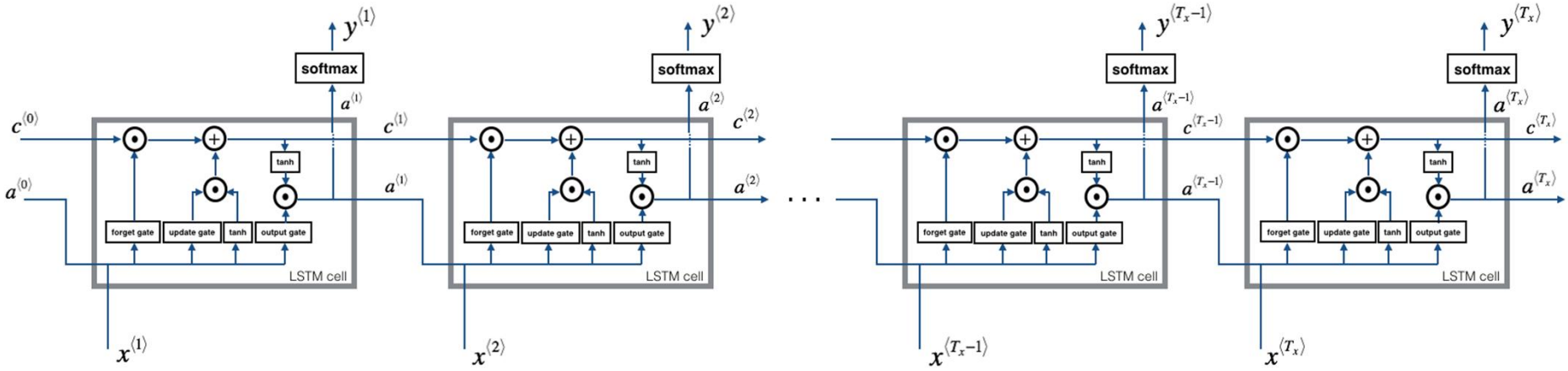
Ex.: Read text, and use LSTM/GRU to keep track of grammatical structures => if the subject is singular or plural. If the subject changes from a singular word to a plural word, forget the previously stored memory value of the singular state.

LSTM's and GRU's are successfully applied in speech recognition, speech synthesis, natural language understanding, time series forecasting, etc.

When to use LSTM or GRU ? - there is not a consensus.
LSTM appeared first (1997), GRU (2014) is a simplification of LSTM.
LSTM is more powerful (3 gates in LSTM, 1 gate in GPU).

** ref. Hochreiter and Schmidhuber 1997, "Long short-term Memory".*

LSTM network

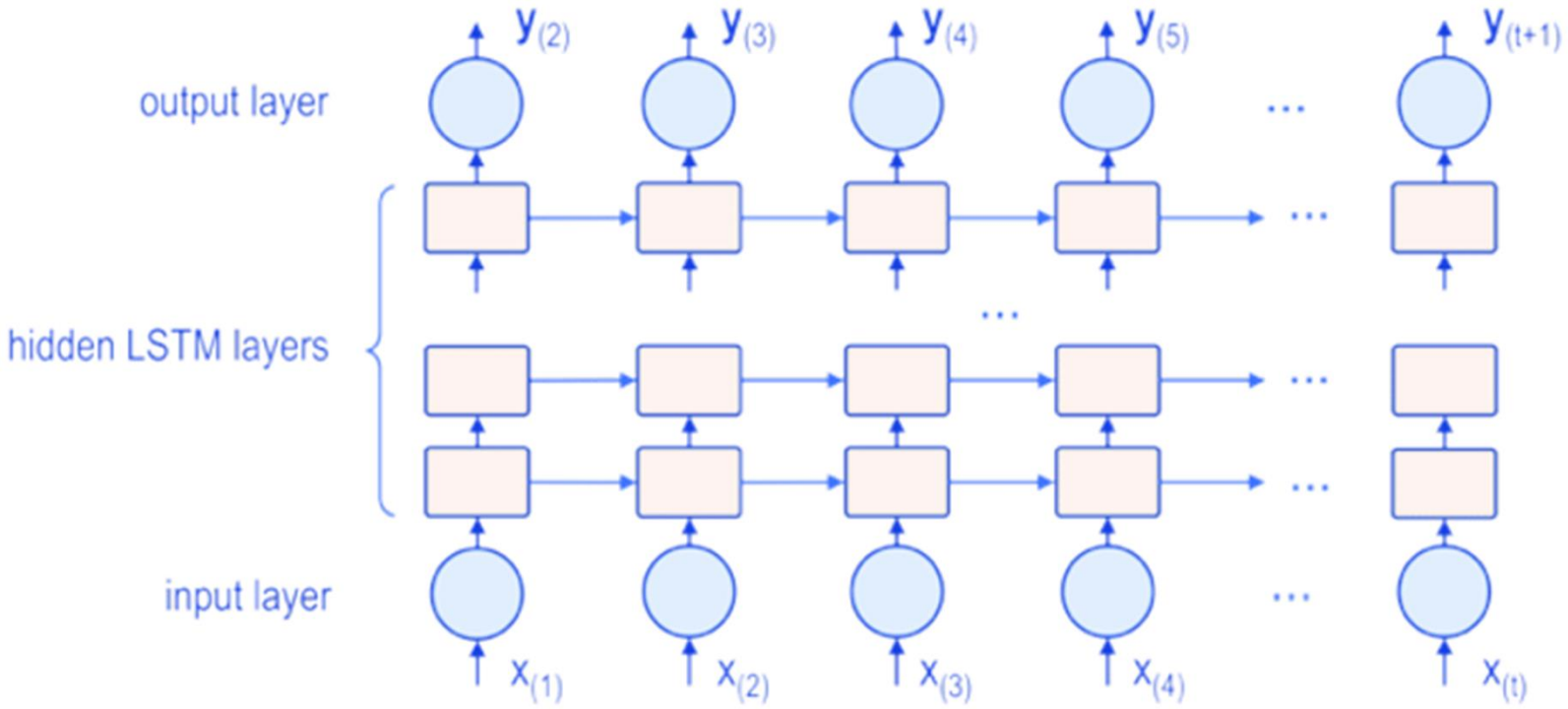


LSTM network is a temporal sequence of LSTM units.
There is a line at the top that shows how LSTM can memorise and pass certain values $c^{<t>}$ through several temporal steps.

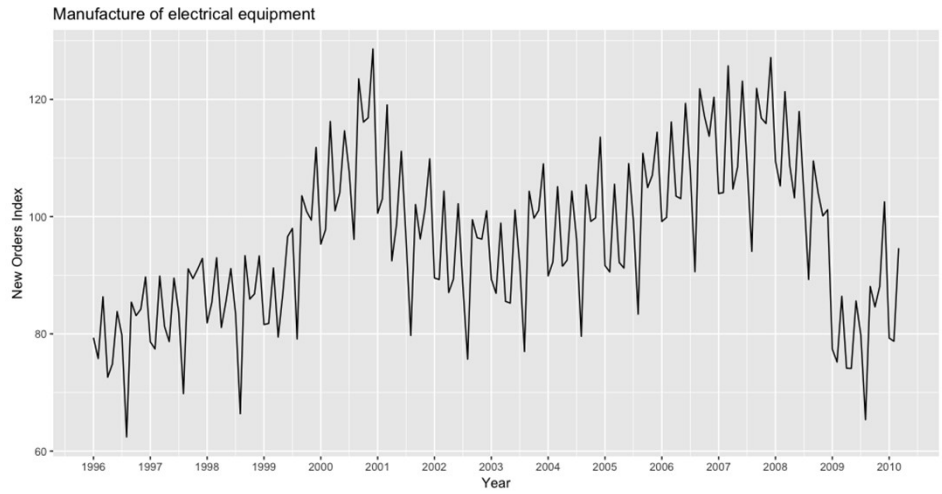
Other versions : the gate's values depend also on $c^{<t-1>}$. (peephole connection)

The gates are vectors (e.g. if 100 dimensional hidden memory cell units, the gates will have 100 elements).

Deep LSTM


output layer
hidden LSTM layers
input layer
y(2)
y(3)
y(4)
y(5)
y(t+1)
x(1)
x(2)
x(3)
x(4)
x(t)

Sequence Models for Time Series Data



Time Series (TS) - collection of data points indexed based on the time they were collected => TS models are naturally sequence models.
Most often, data is recorded at regular time intervals.

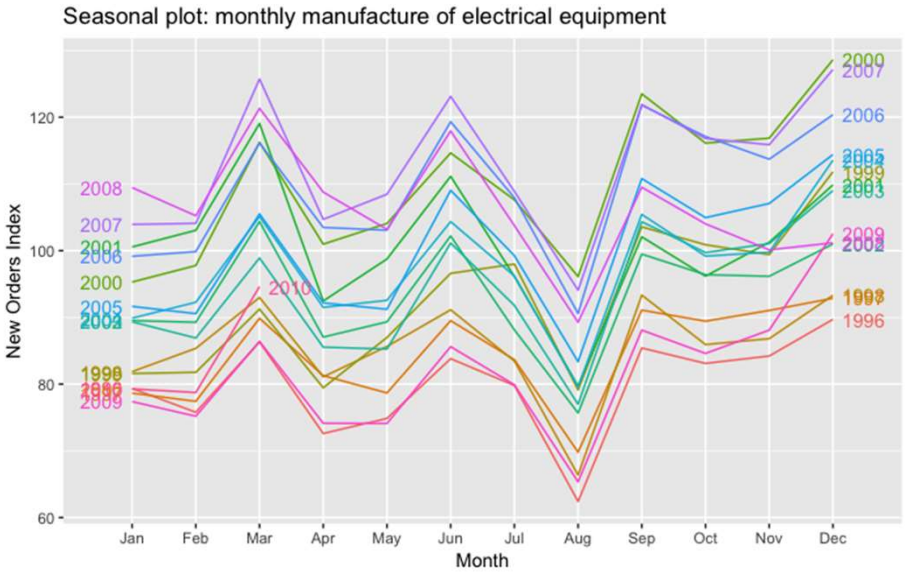
TS forecasting (prediction) is a hot topic with many applications in practice:

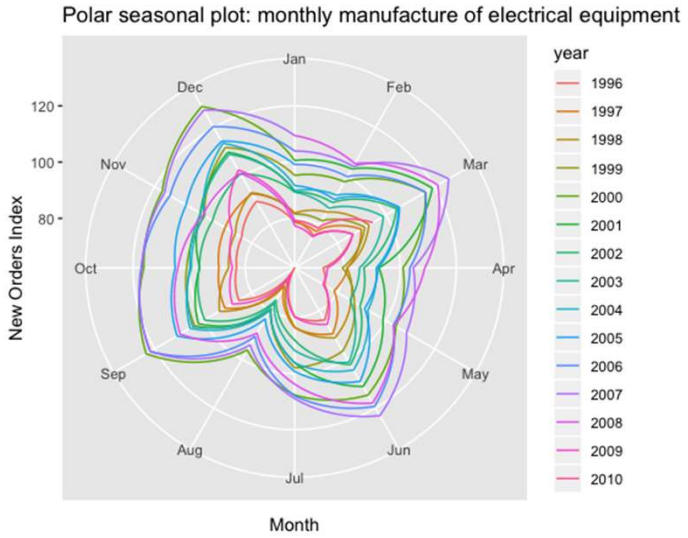
- Stock prices forecasting / Weather forecasting
- Bitcoin price forecasting
- Biological sensors (ECG, EEG) – predict health problems
- Network traffic prediction

Time Series Properties

Stationarity - TS is said to be stationary when the mean and variance remain constant over time.

Seasonality – TS variations at specific time-frames (i.e. people buy more Christmas trees during Christmas, people eat more ice-cream in summer).





Auto-correlation - correlation between the current value and the value from the previous times (lags).

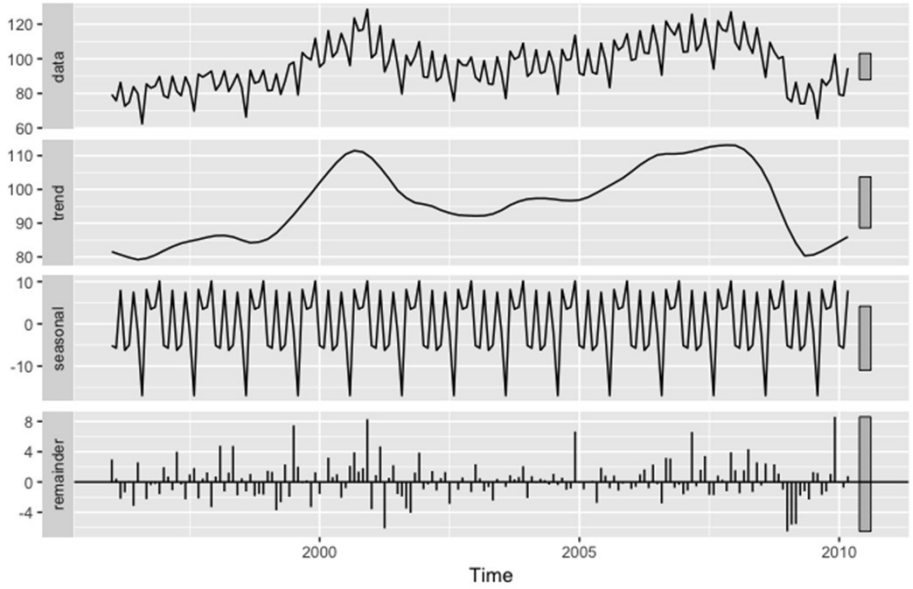
Time Series Decomposition

Y(t) =>

T(t) =>

S(t) =>

R(t) =>



If TS shows some seasonality (e.g. daily, weekly, yearly) it may be useful to decompose the original TS *Y(t)* into sum of 3 components:

Y(t) = S(t) + T(t) + R(t)

T(t) - trend component, estimate T(t) through the mean of the last *n* samples (rolling mean)

S(t) - seasonal component : **S(t) = Y(t)-T(t)**

R(t) - remaining component: **R(t)=Y(t)-T(t)-S(t)**

Trend component in TS

Trend is a long-term increase or decrease in the level of the time series (TS). Systematic change in TS that does not appear to be periodic is known as a trend.

Identify a Trend => Plot TS data to see if a trend is obvious or not.

Remove a Trend => TS with a trend is called non-stationary. Identified trend can be removed (TS de-trending).
If TS does not have a trend or the trend is successfully removed, the dataset is said to be trend stationary.
To get stationary TS, subtract the trend => S(t) = Y(t)-T(t)
To get the trend, subtract the seasonality => T(t)=Y(t)-S(t)
Y(t) – original time series

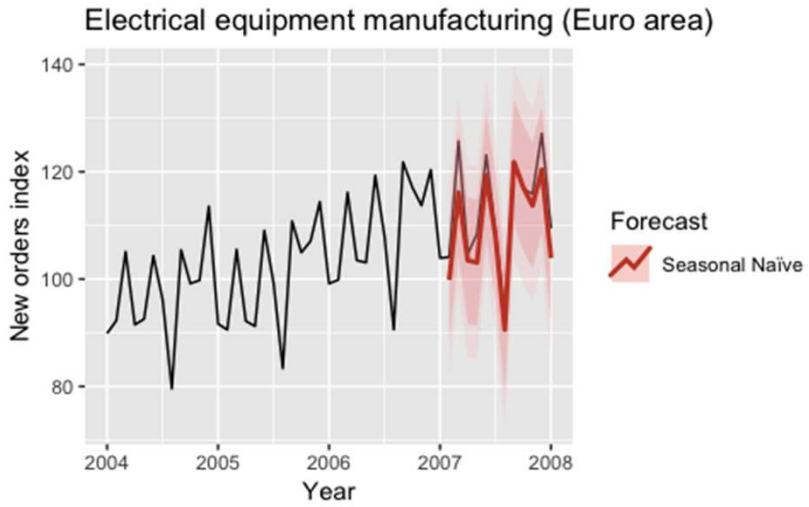
Before forecasting, it is helpful to remove both the trend and seasonality.

Time Series Forecasting – classical methods

TS forecasting can be seen as a supervised regression problem, however the methods need to consider the temporal nature of observations.

1) Naïve model : the forecasts for every horizon (h) correspond to the last observed value: $\hat{Y}(t+h|t) = Y(t)$

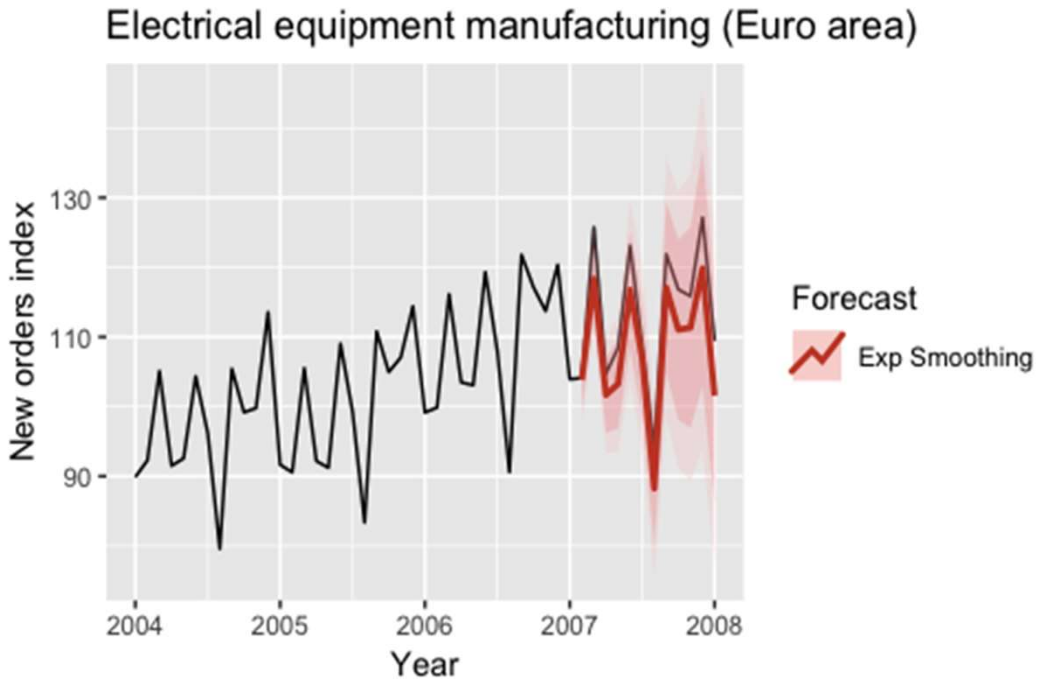
2) SNaïve (Seasonal Naïve) model: TS has a seasonal component with period of seasonality *T:* $\hat{Y}(t+h|t) = Y(t+h-T)$



Time Series Forecasting – classical methods

3) Exponential smoothing (very successful classical forecasting methods). The forecasts are equal to a weighted average of past observations and the corresponding weights decrease exponentially as we go back in time.

$\hat{Y}(t+h|t) = \alpha Y(t) + \alpha(1-\alpha)Y(t-1) + \alpha(1-\alpha)^2 Y(t-2) + \ldots$, $0<\alpha<1$ (basic model)



Time Series Forecasting – classical methods

4) ARIMA (among the most widely used methods for TS forecasting)

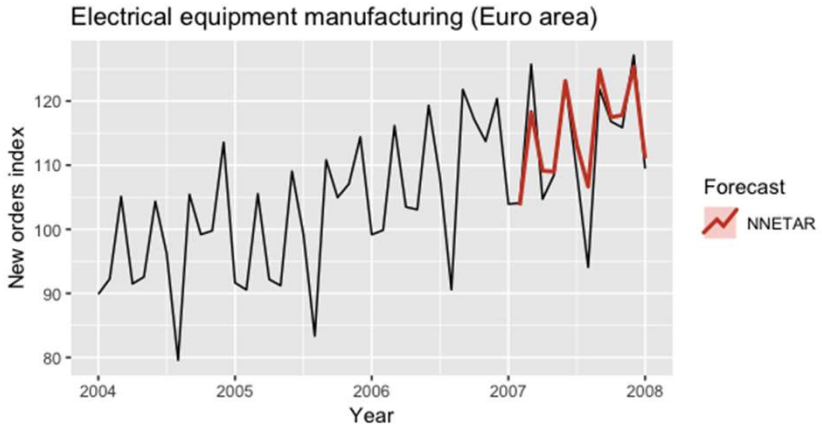
AutoRegressive model - linear combination of past values of the TS.

Moving Average model - linear combination of past forecasting errors.

ARIMA (AutoRegressive Integrated Moving Average) models combine the two models.
TS has to be stationary => instead of the original values use differences : $\hat{Y}(t) = Y(t)-Y(t-T)$

5) SARIMA model (Seasonal ARIMA) extends the ARIMA by adding a linear combination of seasonal past values and/or forecast errors.

Time Series Forecasting – sequence models



Forecasting with classical methods (ARIMA, etc.) => work better with features (e.g. extract trend, seasonality, etc.) than with raw TS data.

Forecasting with Sequence models (LSTM, GRU, etc.) => no need to extract features, work with raw data.

Cut TS into smaller sequences, and use Sequence models to forecast it.

Recommended further reading

- Understanding LSTM Networks – colah's blog :

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

- The Unreasonable Effectiveness of Recurrent Neural Networks - Andrej Karpathy blog:

http://karpathy.github.io/2015/05/21/rnn-effectiveness/