

Applied Machine Learning

Lecture 12: Time series and sequential models

Selpi (selpi@chalmers.se)

The slides are further development of Richard Johansson's slides

March 3, 2020

Example of tasks related to sequences

- ▶ sequence (sentiment/review) classification
- ▶ machine translation
- ▶ forecasting load of electricity
- ▶ recognising activities from videos
- ▶

overview

- ▶ how can we represent and predict **sequential** data using machine learning models?

overview

- ▶ how can we represent and predict **sequential** data using machine learning models?
 - ▶ use classical approaches
 - ▶ use tailored neural network approaches

sequence (review/sentiment) classification

I RECOMMEND this movie!



positive

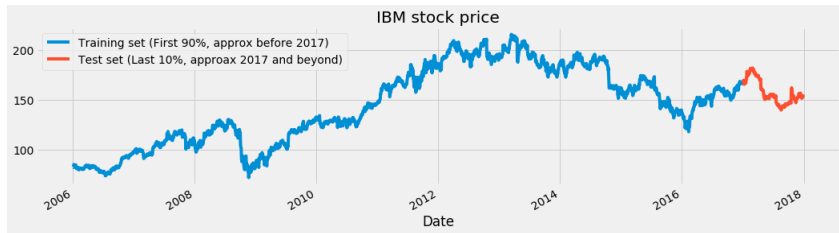
sequence prediction (1)

G O T H E N B



U

sequence prediction (2)



sequence tagging

United Nations official Ekeus heads for Baghdad .



B-ORG I-ORG O B-PER O O B-LOC O

A C A T G G T C T G A A



N N C C C C C C C C C N

Machine translation

This is a very nice school building.



Detta är en mycket trevlig skolbyggnad.

Overview

sequence prediction: classical approaches

neural networks for sequential problems

Different types of RNNs

Use "reduce problem" concept

- ▶ a **reduction** in machine learning means that we convert a complicated problem into (one or more) simpler problems
 - ▶ example: multiclass classification \rightarrow several binary
- ▶ let's see how we attack sequential problems as a series of classification or regression tasks

step-by-step prediction: general ideas

- ▶ break down the problem into a **sequence of smaller decisions**
 - ▶ think of it as a system that gradually consumes input and generates output
- ▶ use standard classifiers or regressors to guess the next step
 - ▶ use features from the **input** and from the **history**
 - ▶ might need to constrain predictions to keep output consistent

example: named entity tagging

- ▶ input: a sentence
- ▶ output: names in the sentence bracketed and labeled

United Nations official **Ekeus** heads for **Baghdad**.
[ORG] [PER] [LOC]

converting brackets into tags

- ▶ typical solution: **B**eginning/**I**nside/**O**utside coding

United Nations official Ekeus heads for Baghdad .

converting brackets into tags

- ▶ typical solution: **B**eginning/**I**nside/**O**utside coding

United Nations official Ekeus heads for Baghdad .
B-ORG

converting brackets into tags

- ▶ typical solution: **B**eginning/**I**nside/**O**utside coding

United Nations official Ekeus heads for Baghdad .
B-ORG I-ORG

converting brackets into tags

- ▶ typical solution: **B**eginning/**I**nside/**O**utside coding

United	Nations	official	Ekeus	heads	for	Baghdad	.
B-ORG	I-ORG	O					

converting brackets into tags

- ▶ typical solution: **B**eginning/**I**nside/**O**utside coding

United	Nations	official	Ekeus	heads	for	Baghdad	.
B-ORG	I-ORG	O	B-PER				

converting brackets into tags

- ▶ typical solution: **B**eginning/**I**nside/**O**utside coding

United	Nations	official	Ekeus	heads	for	Baghdad	.
B-ORG	I-ORG	O	B-PER	O			

converting brackets into tags

- ▶ typical solution: **B**eginning/**I**nside/**O**utside coding

United	Nations	official	Ekeus	heads	for	Baghdad	.
B-ORG	I-ORG	O	B-PER	O	O		

converting brackets into tags

- ▶ typical solution: **B**eginning/**I**nside/**O**utside coding

United	Nations	official	Ekeus	heads	for	Baghdad	.
B-ORG	I-ORG	O	B-PER	O	O	B-LOC	

converting brackets into tags

- ▶ typical solution: **B**eginning/**I**nside/**O**utside coding

United	Nations	official	Ekeus	heads	for	Baghdad	.
B-ORG	I-ORG	O	B-PER	O	O	B-LOC	O

- ▶ see Ratnov and Roth: *Design challenges and misconceptions in named entity recognition*. CoNLL 2009.

training step-by-step systems: the basic approach

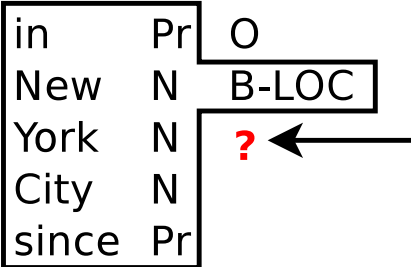
- ▶ how can we train the decision classifier in a step-by-step system?

training step-by-step systems: the basic approach

- ▶ how can we train the decision classifier in a step-by-step system?
- ▶ **imitation learning**: learn to imitate an “expert”
- ▶ naive approach to imitation learning:
 - ▶ force the system to generate the correct output
 - ▶ observe the states we pass along the way
 - ▶ use them as examples and train the classifier

example of features: named entity tagging

she	Pn	O
lives	V	O
in	Pr	O
New	N	B-LOC
York	N	?
City	N	
since	Pr	



what happens at prediction time?

- ▶ how will we use this system for prediction?

what happens at prediction time?

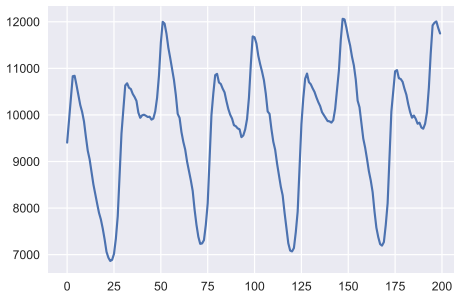
- ▶ how will we use this system for prediction?
- ▶ there is **difference between training and prediction**:
 - ▶ when training, “history features” are based on the gold standard
 - ▶ when predicting, they are based on automatic predictions

code example: named entity tagging

see <https://www.clips.uantwerpen.be/conll2003/ner/>

discussion

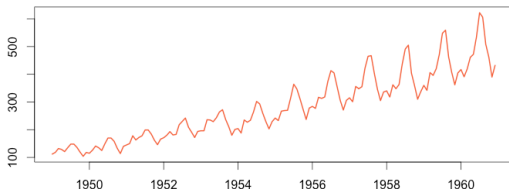
- ▶ how would you build a model to predict the demand of electricity load an hour from now?



see e.g. Chang et al. (2001) *EUNITE Network Competition: Electricity Load Forecasting*

time series prediction

- ▶ **time series** is a set of observation collected at specified time normally at equal interval (every 30minutes, every month, etc.)
- ▶ One usage is: to predict future values given the historical observed values (**predictive** perspective). This is the focus in ML.
- ▶ Note that the temporal aspect is important.
 - ▶ Useful info come not only from input features, but also from the changes in input/output over time.
 - ▶ make prediction is harder



Components of time series, in classic statistics approach

- ▶ trend (increase/decrease, something that happen for sometime and then disappear)
- ▶ seasonality (repeating pattern within a certain fixed period; e.g., sale is higher in December than other months)
- ▶ irregularity (noise, cannot be predicted); e.g., an outbreak happen, sale of maskers gone up, after that come down again
- ▶ cyclic (up/down movement), they keep on repeating, but harder to predict

time series analysis software in Python

► StatsModels:

<https://www.statsmodels.org/stable/tsa.html>



Table Of Contents

Time Series analysis tsa

- Descriptive Statistics and Tests
- Estimation
 - Univariate Autogressive Processes (AR)
 - Autogressive Moving-Average Processes (ARMA) and Kalman Filter
 - Exponential Smoothing
 - Vector Autogressive Processes (VAR)
- Vector Autogressive Processes (VAR)
- Vector Error Correction Models (VECM)
- Regime switching models
- ARMA Process

Time Series analysis `tsa`

`statsmodels.tsa` contains model classes and functions that are useful for time series analysis. Basic models include univariate autoregressive models (AR), vector autoregressive models (VAR) and univariate autoregressive moving average models (ARMA). Non-linear models include Markov switching dynamic regression and autoregression. It also includes descriptive statistics for time series, for example autocorrelation, partial autocorrelation function and periodogram, as well as the corresponding theoretical properties of ARMA or related processes. It also includes methods to work with autoregressive and moving average lag-polynomials. Additionally, related statistical tests and some useful helper functions are available.

Estimation is either done by exact or conditional Maximum Likelihood or conditional least-squares, either using Kalman Filter or direct filters.

Currently, functions and classes have to be imported from the corresponding module, but the main classes will be made available in the `statsmodels.tsa` namespace. The module structure is within `statsmodels.tsa` is

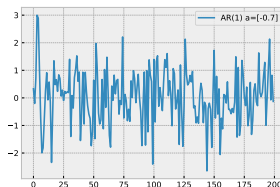
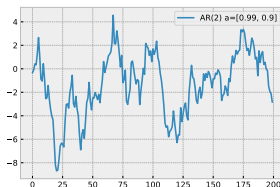
- `statsmodels`: empirical properties and tests, acf, pacf, granger-causality, adf unit root test, kpss test, bds test,jung-box test and others.
- `ar_model`: univariate autoregressive process, estimation with conditional and exact maximum likelihood and conditional least-squares
- `arma_model`: univariate ARMA process, estimation with conditional and exact maximum likelihood and conditional least-squares

autoregressive models: $AR(p)$

- an **autoregressive model** $AR(p)$ is defined by the equation

$$Y_t = c + \sum_{i=1}^p a_i Y_{t-i} + \varepsilon_t$$

where Y_t is the time series, c and a_i are constants, and ε_t is Gaussian noise



using an AR model for prediction

$$Y_t \approx c + \sum_{i=1}^p a_i Y_{t-i}$$

- ▶ for our purposes, this type of model can be seen as a type of linear regression model
- ▶ next step can be predicted from some previous steps

extensions to the basic autoregressive model

- ▶ AR models with **exogenous** variables (*ARX*)

$$Y_t = c + \sum_{i=1}^p a_i Y_{t-i} + \sum_{i=1}^q b_i X_{t-i} + \varepsilon_t$$

allow us to add more features, e.g. to model **seasonality**

- ▶ **nonlinear** models (*NARX*):

$$Y_t = F(Y_{t-1}, \dots, X_{t-i}, \dots) + \varepsilon_t$$

where F is any predictive model

Converting time series prediction/forecasting to supervised learning

- ▶ Convert data using sliding window method
 - ▶ one step forecast
 - ▶ multi-step forecast
- ▶ If using

Overview

sequence prediction: classical approaches

neural networks for sequential problems

Different types of RNNs

overview

- ▶ previously, we had an introduction to neural network classifiers
 - ▶ key attractions of this type of model is the reduced need of feature engineering
- ▶ today: **recurrent** neural networks, which we apply to sequential data
 - ▶ classifying sentences or documents (e.g., predict positive/negative review)
 - ▶ outputting sequences, such as time series prediction
 - ▶ “sequence-to-sequence”, such as machine translation

recap: feedforward NN

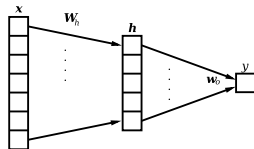
- ▶ a **feedforward neural network** or **multilayer perceptron** consists of connected layers of “classifiers”
 - ▶ the intermediate classifiers are called **hidden units**
 - ▶ the final classifier is called the **output unit**
- ▶ each hidden unit h_i computes its output based on its own weight vector w_{h_i} :

$$h_i = f(w_{h_i} \cdot x)$$

- ▶ and then the output is computed from the hidden units:

$$y = f(w_o \cdot h)$$

- ▶ the function f is called the **activation**



recap: representing words in NNs

- ▶ NN implementations tend to prefer short, dense vectors
- ▶ recall the way we code word features as one-hot sparse vectors:

$$\begin{aligned} \text{tomato} &\rightarrow [0, 0, 1, 0, 0, \dots, 0, 0, 0] \\ \text{carrot} &\rightarrow [0, 0, 0, 0, 0, \dots, 0, 1, 0] \end{aligned}$$

- ▶ the solution: represent words with low-dimensional vectors (**word embeddings**), in a way so that words with similar meaning have similar vectors

$$\begin{aligned} \text{tomato} &\rightarrow [0.10, -0.20, 0.45, 1.2, -0.92, 0.71, 0.05] \\ \text{carrot} &\rightarrow [0.08, -0.21, 0.38, 1.3, -0.91, 0.82, 0.09] \end{aligned}$$

- ▶ libraries such as `word2vec` help us build the vectors

word embeddings

- ▶ **word embeddings**: instead of high-dimensional sparse vectors, low-dimensional dense vectors

Gothenburg → $[0.010, -0.20, \dots, 0.15, 0.07, -0.23, \dots]$

Stockholm → $[0.015, -0.05, \dots, 0.11, 0.14, -0.12, \dots]$

- ▶ created as a by-product of training a neural network, or by separate **pre-training** using raw text
- ▶ this solution addresses the previously mentioned limitations:
 - ▶ low dimensionality, easy to plug into NNs
 - ▶ can encode some sort of “similarity”
 - ▶ to some extent, they mitigate the problem of rarity – if pre-trained

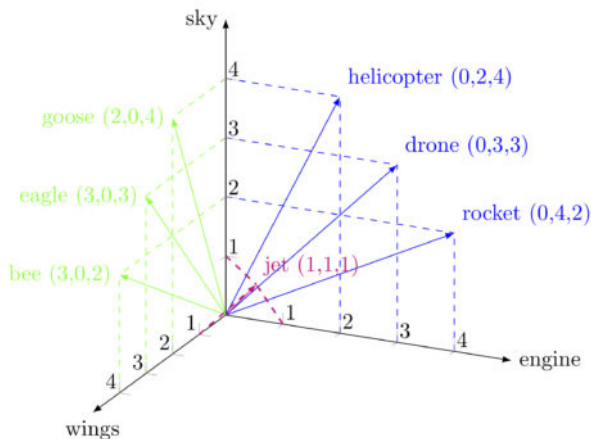
Using “contexts” to determine similarity

- ▶ the **distributional hypothesis**: the distribution of contexts in which a word appears is a good proxy of the meaning of that word. Assumption: two words probably have a similar “meaning” if they tend to appear in similar **contexts**
- ▶ example of a word–word co-occurrence matrix:
 - ▶ “I like NLP”
 - ▶ “I like deep learning”
 - ▶ “I enjoy flying”

$$X = \begin{array}{l} \begin{array}{c} I \\ like \\ enjoy \\ deep \\ learning \\ NLP \\ flying \\ . \end{array} \end{array} \begin{bmatrix} I & like & enjoy & deep & learning & NLP & flying & . \\ 0 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$

[source]

measuring the similarity between words: cosine



[source]

word2vec: basic assumptions

- ▶ each **word** w comes from a word vocabulary W
- ▶ from a large collection of text, we collect **contexts** such as
*they drink **coffee** with milk*

word2vec: “skip-gram with negative sampling”

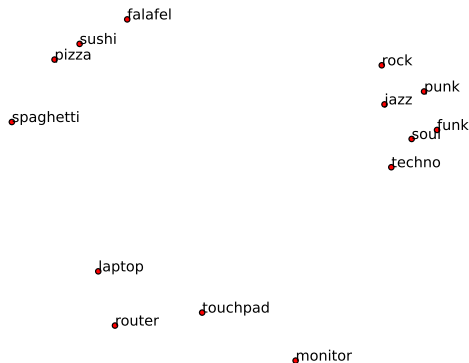
- ▶ collect word–context pairs occurring in the text data
coffee: drink
- ▶ for each such pair, randomly generate a number of synthetic pairs:
coffee: mechanic
coffee: contrary
coffee: persuade
- ▶ then fit the following model w.r.t. (V, V')

$$P(\text{true pair} | (w, c)) = \frac{1}{1 + \exp(-V(w) \cdot V'(c))}$$

$$P(\text{synthetic pair} | (w, c)) = 1 - \frac{1}{1 + \exp(-V(w) \cdot V'(c))}$$

inspection of the model

- ▶ after training the embedding model, we can inspect the result using nearest-neighbor lists
- ▶ for illustration, vectors can be projected to two dimensions using methods such as t-SNE or PCA



gensim example: training a model

```
import gensim

text_file_name = 'data/wikipedia.txt'
sentences = gensim.models.word2vec.LineSentence(text_file_name,
                                                  limit=100000)

model = gensim.models.Word2Vec(sentences, size=20, window=5,
                               min_count=5, workers=2)

...

model.most_similar('europe')
```

gensim example: using a trained model

```
from gensim.models import KeyedVectors

filename = 'GoogleNews-vectors-negative300.bin'
model = KeyedVectors.load_word2vec_format(filename, binary=True)

...
```


recap: sequential prediction

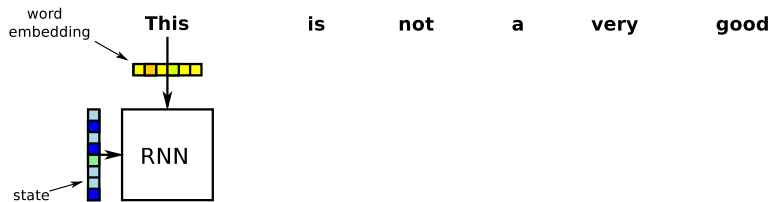
United	Nations	official	Ekeus	heads	for	Baghdad	.
B-ORG	I-ORG	O	B-PER	O	O	B-LOC	O

- ▶ a normal classifier is applied in each step: SVM, LR, decision tree, neural network, ...
- ▶ limitation: features are typically extracted from a small window!

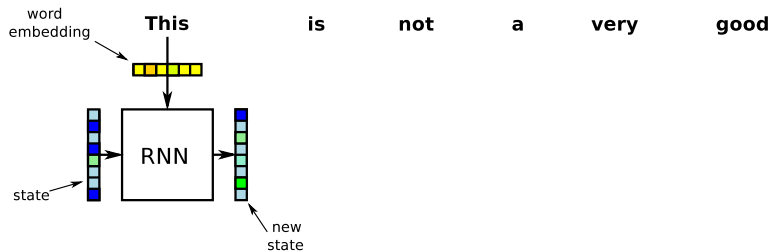
recurrent NN for processing sequences

- ▶ **recurrent** NNs (RNNs) are applied in a step-by-step fashion to a sequential input such as a sentence
- ▶ RNNs use a **state** vector that represents what has happened previously
- ▶ after each step, a new state is computed

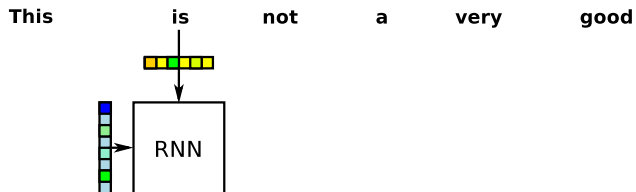
RNN example



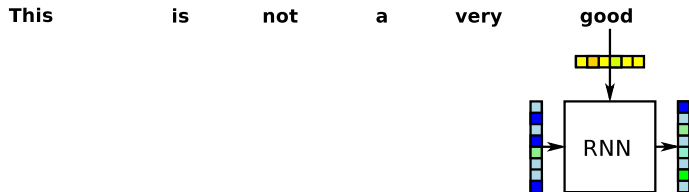
RNN example



RNN example



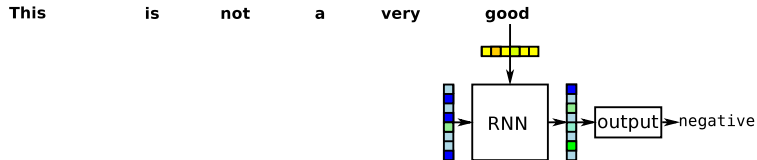
RNN example



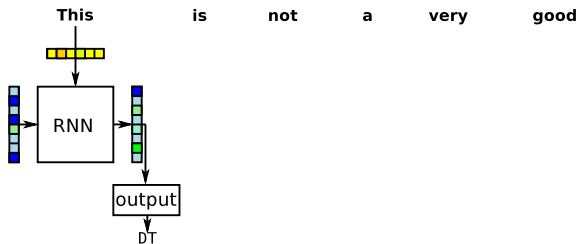
using the RNN output

- ▶ the RNN output isn't that interesting on its own ...
- ▶ we can use it
 - ▶ in sequence classifiers
 - ▶ in sequence predictors
 - ▶ in sequence taggers
 - ▶ in translation

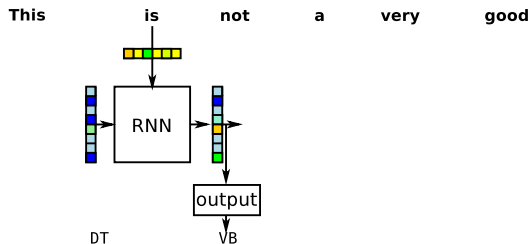
example: using the RNN output in a document classifier



example: using the RNN output in a part-of-speech tagger



example: using the RNN output in a part-of-speech tagger



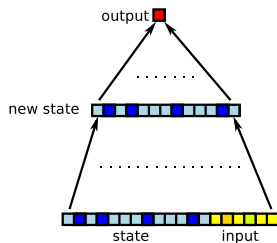
what's in the box? simple RNN implementation

- ▶ the simplest type of RNN looks similar to a feedforward NN
 - ▶ the next state is computed like a hidden layer in a feedforward NN

$$\text{state}_t = \sigma(W_h \cdot (\text{input} \oplus \text{state}_{t-1}))$$

- ▶ ...and the output from the next state

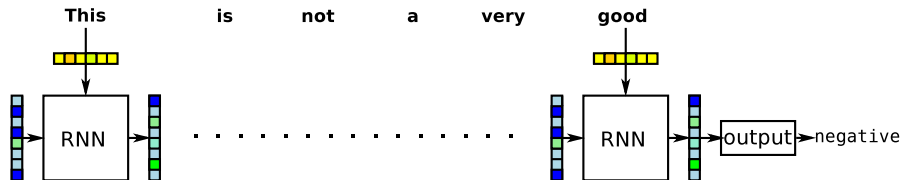
$$\text{output} = \text{softmax}(W_o \cdot \text{state}_t)$$



simple RNNs in Keras

```
model = Sequential()  
model.add(SimpleRNN(32))
```

training RNNs: backpropagation through time



Overview

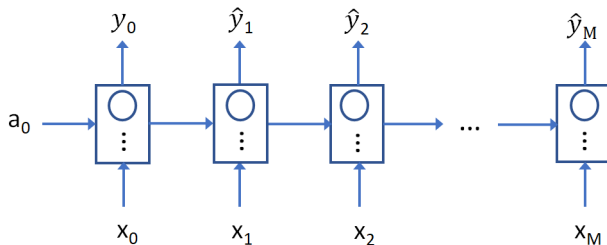
sequence prediction: classical approaches

neural networks for sequential problems

Different types of RNNs

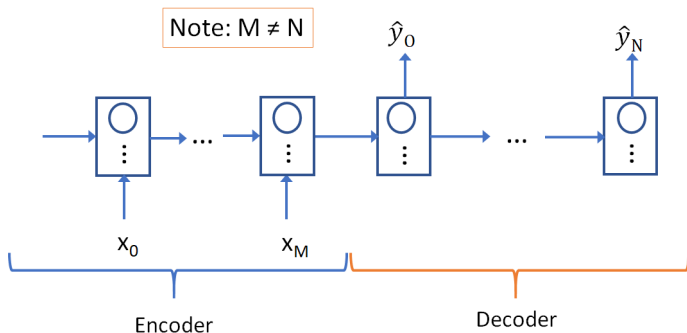
Many-to-many architectures(1)

- For input and output sequences of same length



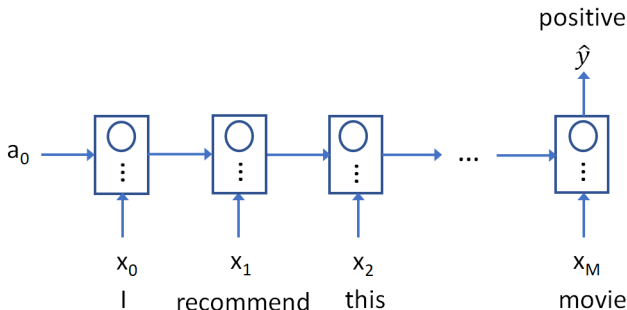
Many-to-many architectures(2)

- ▶ For input and output sequences of different lengths
- ▶ For example, machine translation for sentiment/document classification, input: sequence, output: positive/negative



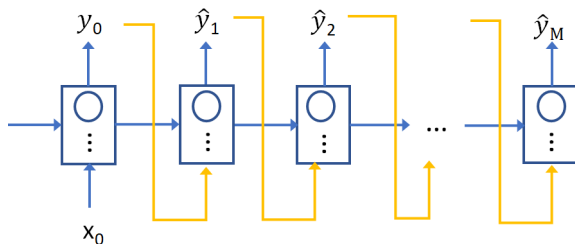
Many-to-one

- For example, for sentiment classification, input: sequence and output: positive/negative. For video-based activity recognition, input: sequence of video frames, output: type of activity



One-to-many

- For example, for music generation, input: genre, output: sequence of notes



simple RNNs have a drawback

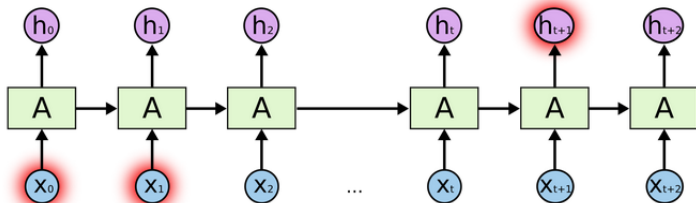


image borrowed from [C. Olah's blog](#)

long short-term memory: LSTM

- ▶ the **long short-term memory** is a type of RNN that is designed to handle long-term dependencies in a better way

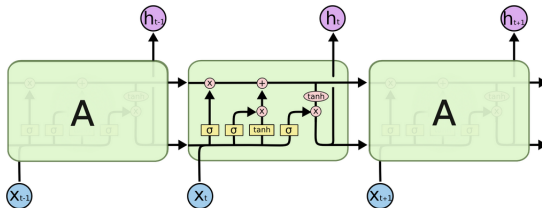
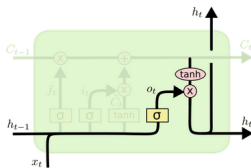


image borrowed from [C. Olah's blog](#)

- ▶ its state consists of two parts – a **short-term** and a **long-term** part (also known as the **cell state**)
- ▶ note that the exact implementation can differ slightly in literature!

LSTM: short-term part and output



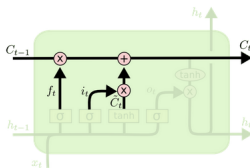
$$o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$$

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

image borrowed from [C. Olah's blog](#)

- short-term state is a bit similar to what we had in the simple RNN

LSTM: long-term part



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

image borrowed from [C. Olah's blog](#)

- ▶ the long-term state is controlled by a **forget gate** that decides if information should be passed along or discarded
- ▶ parts of the input and short-term state can be added to the long-term state

LSTMs in Keras

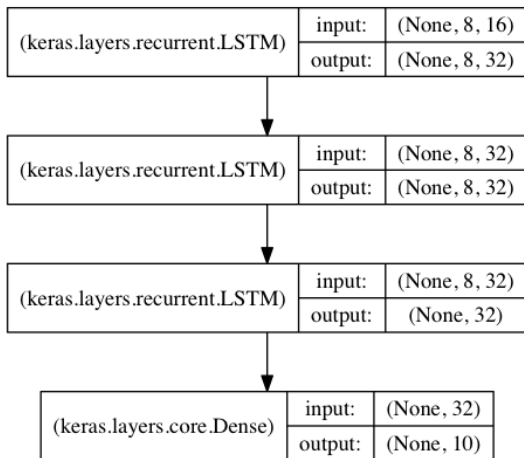
```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, output_dim=256))
model.add(LSTM(128))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='adam')

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

stacked LSTMs



stacked LSTMs in Keras

```
model = Sequential()

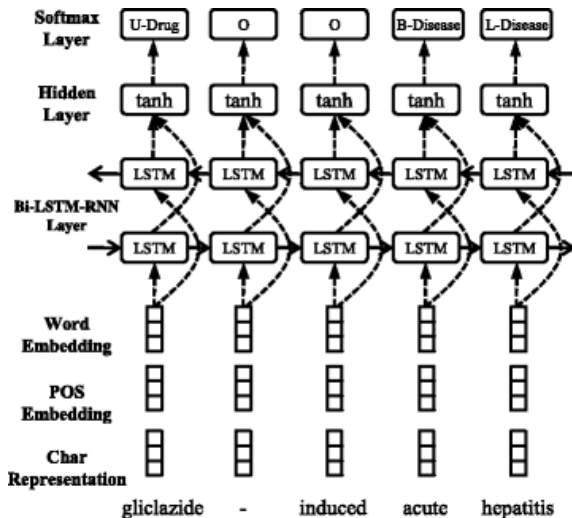
# returns a sequence of vectors of dimension 32
model.add(LSTM(32, return_sequences=True,
               input_shape=(timesteps, data_dim)))

# returns a sequence of vectors of dimension 32
model.add(LSTM(32, return_sequences=True))

# return a single vector of dimension 32
model.add(LSTM(32))

model.add(Dense(10, activation='softmax'))
```

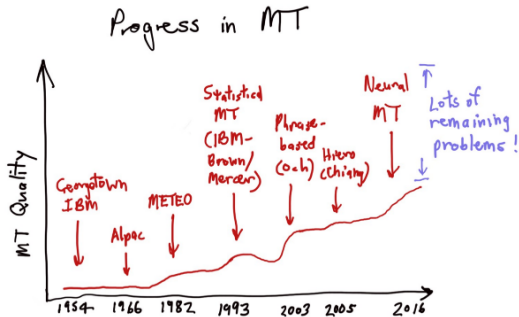
sequence tagging based on LSTMs



[source]

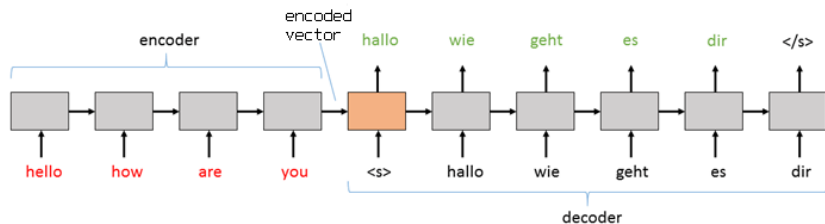
example: NER using LSTMs

recent developments in machine translation



[image by [Chris Manning](#)]

sequence-to-sequence learning for translation

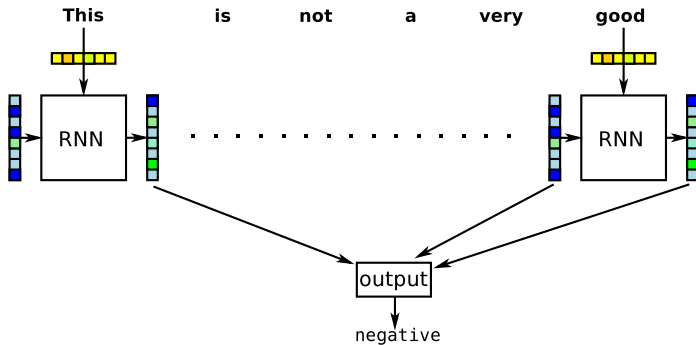


Sutskever et al. (2014) *Sequence to Sequence Learning with Neural Networks*.

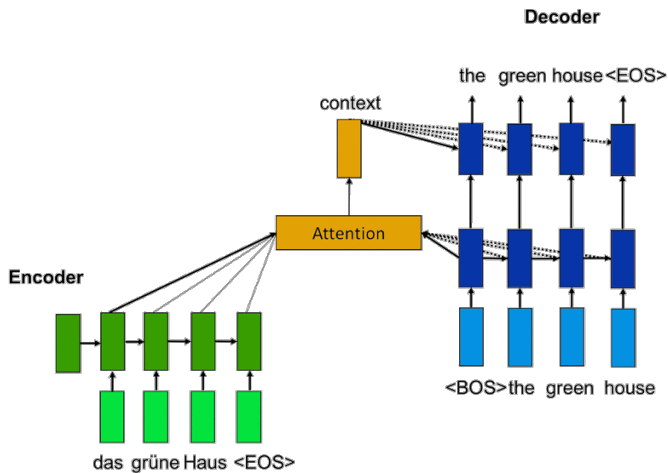
training data for machine translation systems



a final twist on RNNs: neural attention

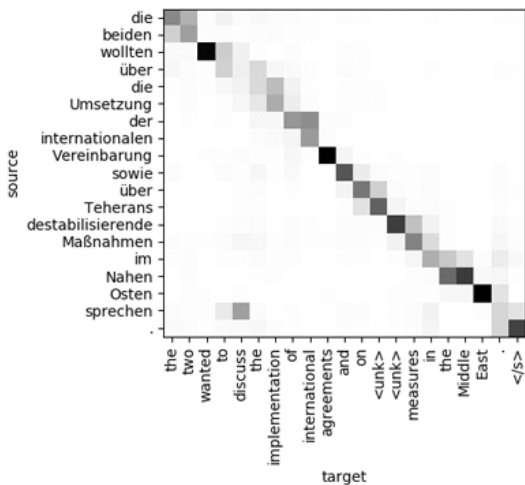


translation with attention



[[source](#)]

example: visualizing attention



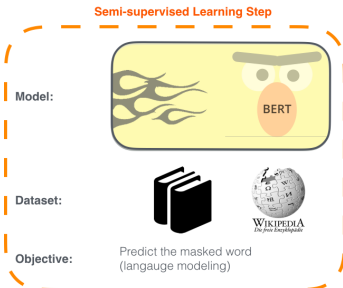
[source]

recent developments (2018)

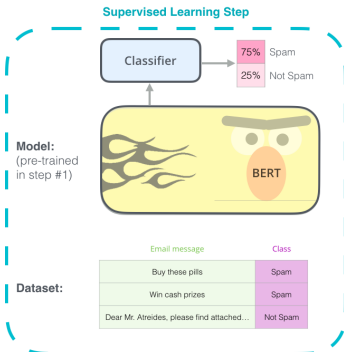
- ▶ We can use a **pre-trained** CNN for our own image recognition task to give us better performance, compared to when building own model using own training data
- ▶ can something similar be done for text?
- ▶ the **ELMo** model (by Allen AI) is a pre-trained LSTM model that can be “plugged” into other applications
 - ▶ <https://allennlp.org/elmo>
- ▶ the **BERT** model (by Google) uses a new architecture called the **transformer**
 - ▶ <https://github.com/google-research/bert>
- ▶ Many other pre-trained models derived from BERT
- ▶ XL-net

1 - Semi-supervised training on large amounts of text (books, wikipedia..etc).

The model is trained on a certain task that enables it to grasp patterns in language. By the end of the training process, BERT has language-processing abilities capable of empowering many models we later need to build and train in a supervised way.



2 - Supervised training on a specific task with a labeled dataset.



[source]

Next lectures

- ▶ 6 March: Guest lectures from two companies giving industrial aspects of machine learning
 - ▶ Solita
 - ▶ Ericsson
- ▶ 10 March: Semi-supervised learning, co-training/multi-view learning, self-learning, recommender systems, and more on LSTM