Neural Networks and Deep Learning

Recurrent Neural Networks

Sequential data

Speech recognition

“Alexa, self destruct”

Music generation

Sentiment analysis

“I hate this movie”

Machine translation

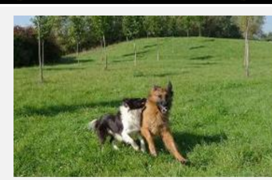
“The hat does not fit in the bag because it is small”

"El sombrero no hi cap a la bossa perque és petita"

Video action recognition

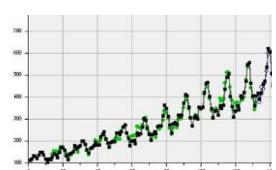
Running

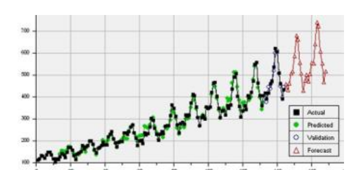
Image captioning



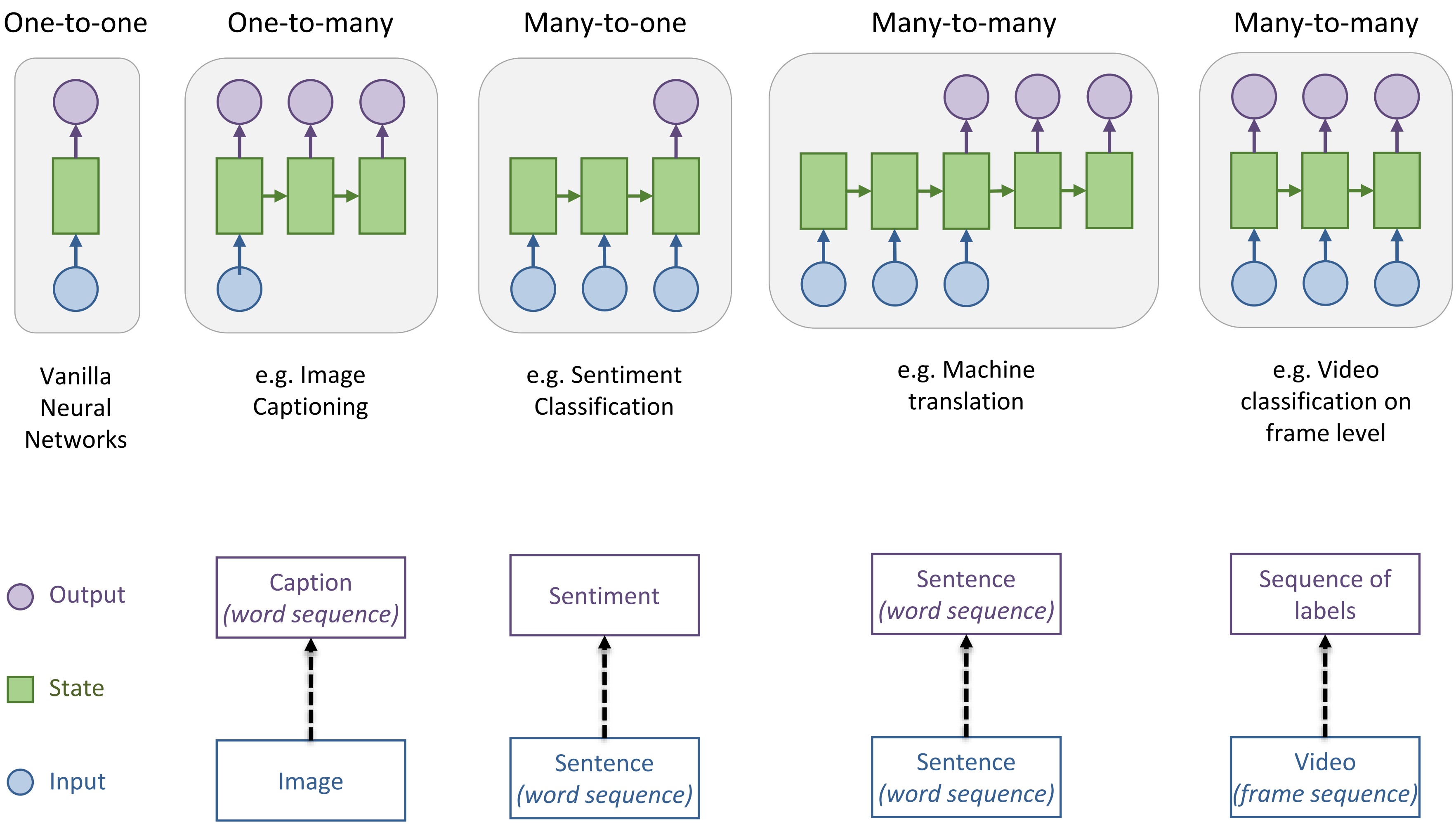
"Two dogs play in the grass"

Time series analysis



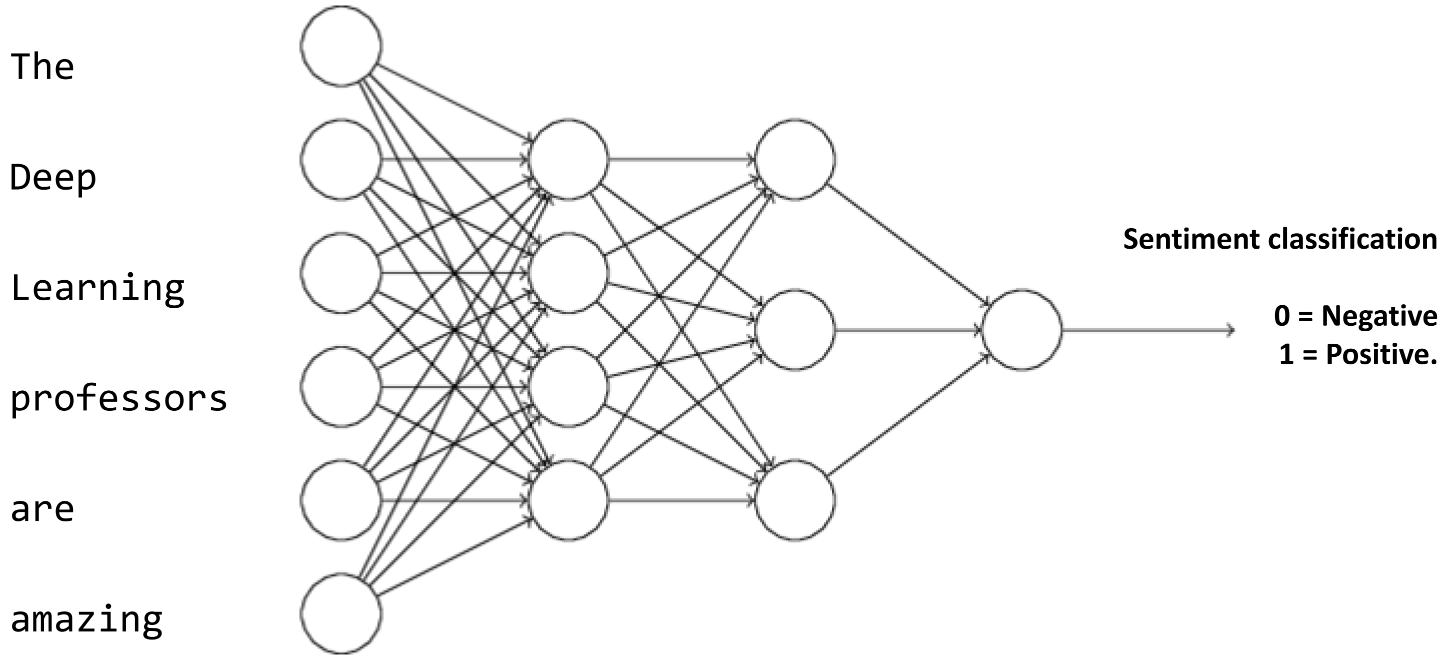


Modelling Sequences



Slide adapted from Convolutional Neural Networks course, Fei-Fei Li, Stanford University, 2017

Why not a standard MLP?

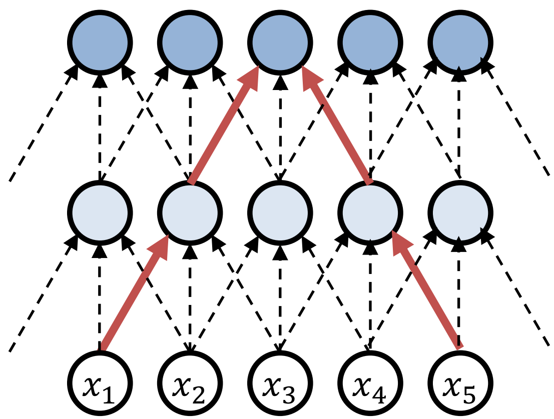


Inputs can be different lengths in different examples.

Does not share features learned across different positions.

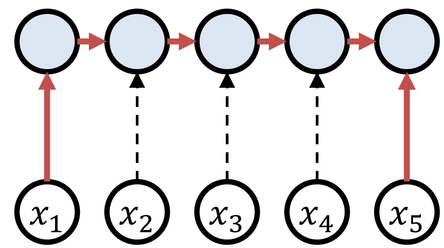
CNNs vs RNNs

CNN



Hierarchical: $O(n/k)$

RNN



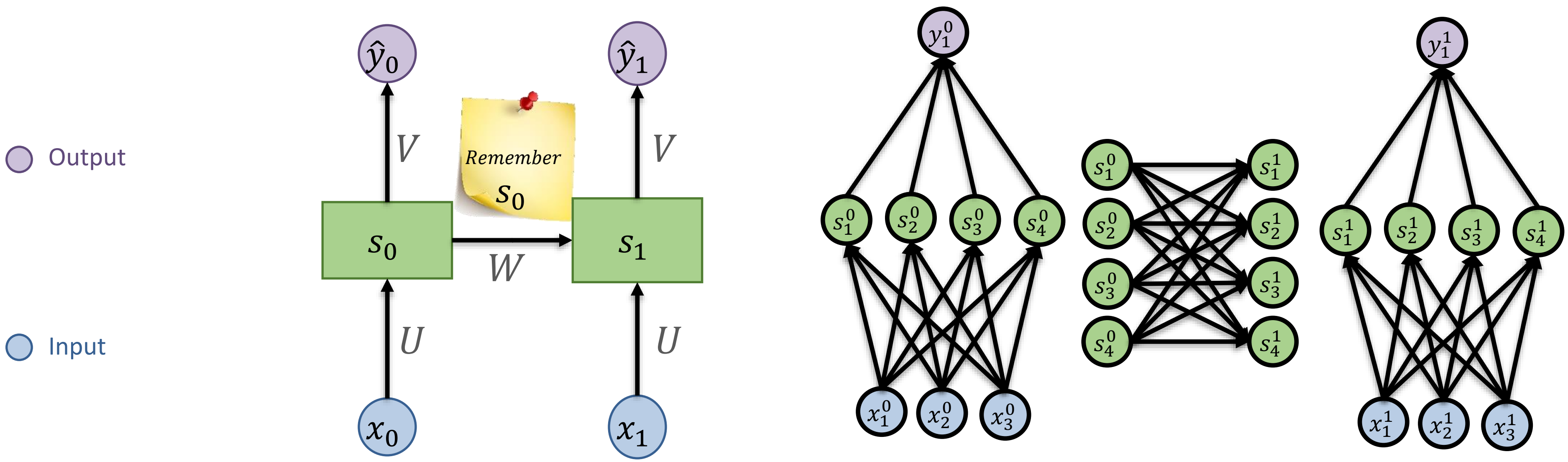
Sequential: $O(n)$

CNNs need less “steps” but more computation to “see” far apart. Notion of “spatial structure”, but no notion of “order”	RNNs need a linear number of steps but less computation to “see” far apart. Capture notion of ordered sequences
The further apart we need them to integrate information from, the more depth we need to add	In order to integrate information from far apart we need to introduce some kind of “long-term memory”
We cannot easily produce a sequential output of arbitrary size	We can easily deal with sequential inputs and outputs of arbitrary size

$O(\cdot)$ - maximum path length for a sequence of length n and filter size k

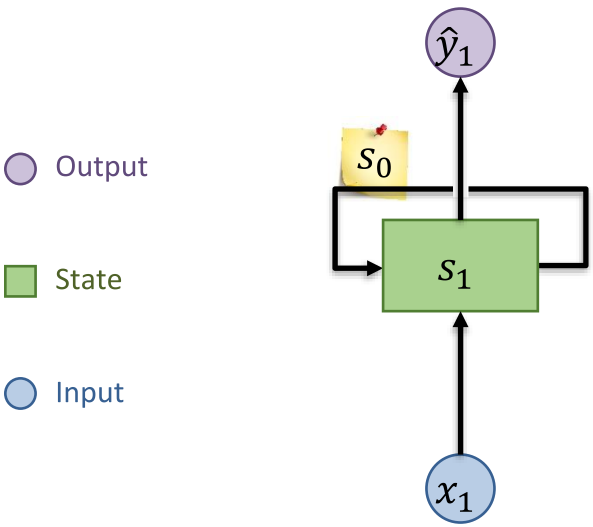
RNN BASICS

Recurrent Neural Networks (RNN)



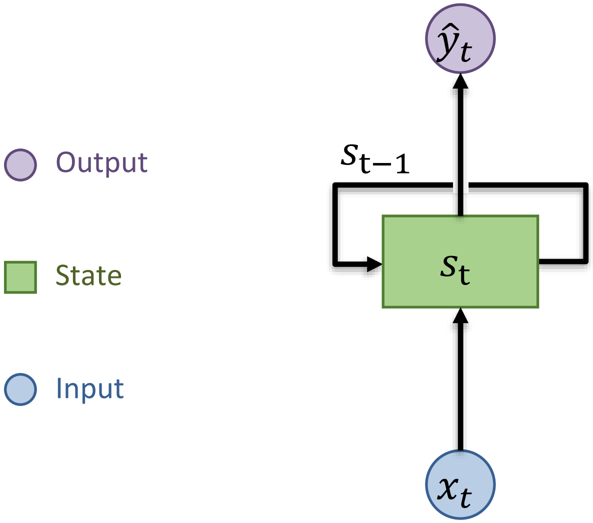
$$x_0 = < x_1^0, x_2^0, x_3^0 >$$
$$s_0 = < s_1^0, s_2^0, s_3^0, s_4^0 >$$
$$\hat{y}_0 = < y_1^0 >$$

Recurrent Neural Networks (RNN)



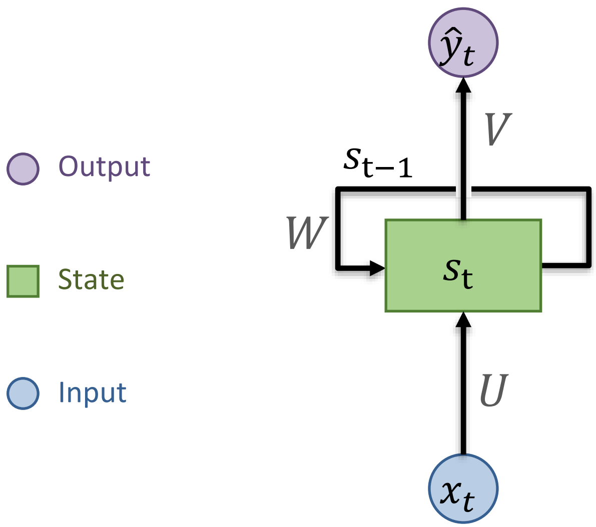
Recurrent Neural Networks (RNN)

RNNs are networks with loops, allowing information to persist over time.
RNNs have “states”.



At time step t the new state depends on both the input to the model (x_t) and the state of previous time step (s_{t-1})

Recurrent Neural Networks (RNN)



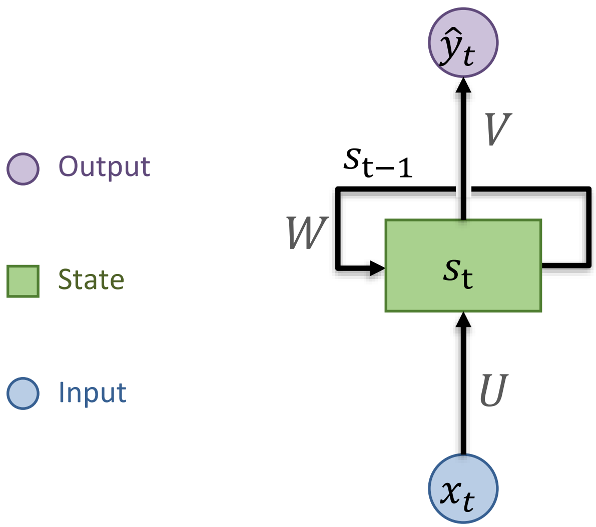
State update (transition function):

$$s_t = f_{U,W}(x_t, s_{t-1})$$

Output function:

$$\hat{y}_t = f_V(s_t)$$

Recurrent Neural Networks (RNN)



State update (transition function):

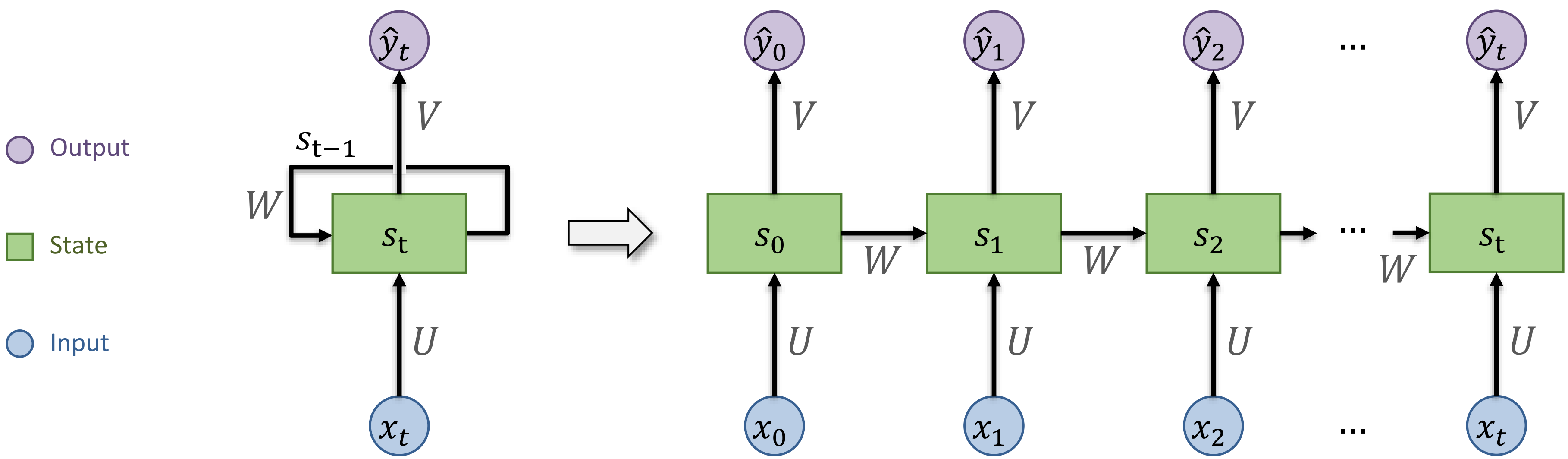
$$s_t = tanh(Ux_t + Ws_{t-1})$$

Output function:

$$\hat{y}_t = f(Vs_t)$$

Recurrent Neural Networks (RNN)

It is easier to think about the unravelled (over time) version of an RNN:



Note that parameters are shared over time

Simple RNN pseudocode

```
# model parameters (weight matrices)
W = np.random.random((hidden_features, hidden_features))
U = np.random.random((hidden_features, in_features))
V = np.random.random((out_features, hidden_features))

state = 0          # state at t=0
outputs = []       # output sequence

for input_t in input_sequence: # iterate over input sequence elements
         # update the state
         state = activation(np.dot(U, input_t) + np.dot(W, state))
         # Calculate the output
         output_t = np.dot(V, state)
         # Keep track of the outputs of all time steps
         outputs.append(output)
```

Notice that `dot(U, input_t)` and `dot(W, state)` are linear (fully connected) layers

Simple (Elman) RNN in PyTorch

Docs > torch.nn > RNNCell

RNNCELL

CLASS `torch.nn.RNNCell(input_size, hidden_size, bias=True, nonlinearity='tanh', device=None, dtype=None)` [SOURCE]

An Elman RNN cell with tanh or ReLU non-linearity.

$$h' = \tanh(W_{ih}x + b_{ih} + W_{hh}h + b_{hh})$$

If `nonlinearity` is *'relu'*, then ReLU is used in place of tanh.

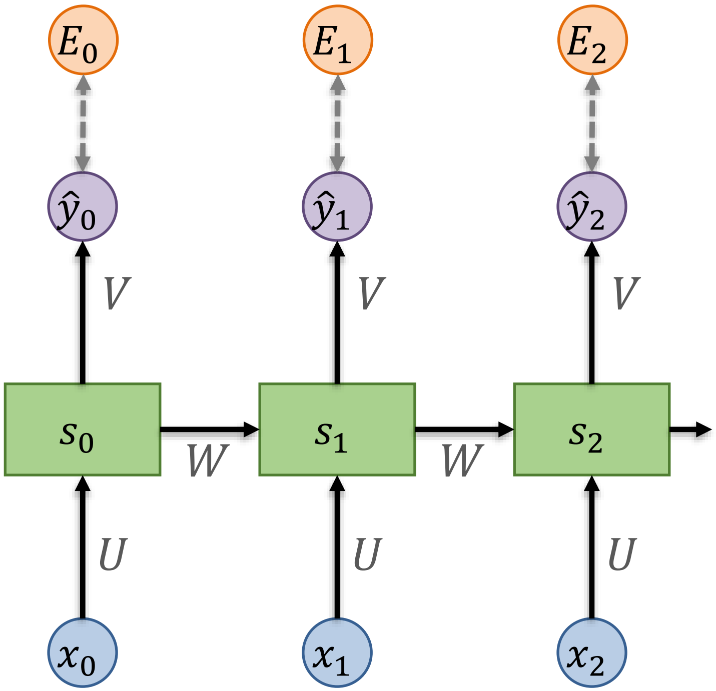
Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`

https://pytorch.org/docs/stable/generated/torch.nn.RNNCell.html#torch.nn.RNNCell

Elman, J.L. (1990), "Finding Structure in Time". Cognitive Science, 14: 179-211.

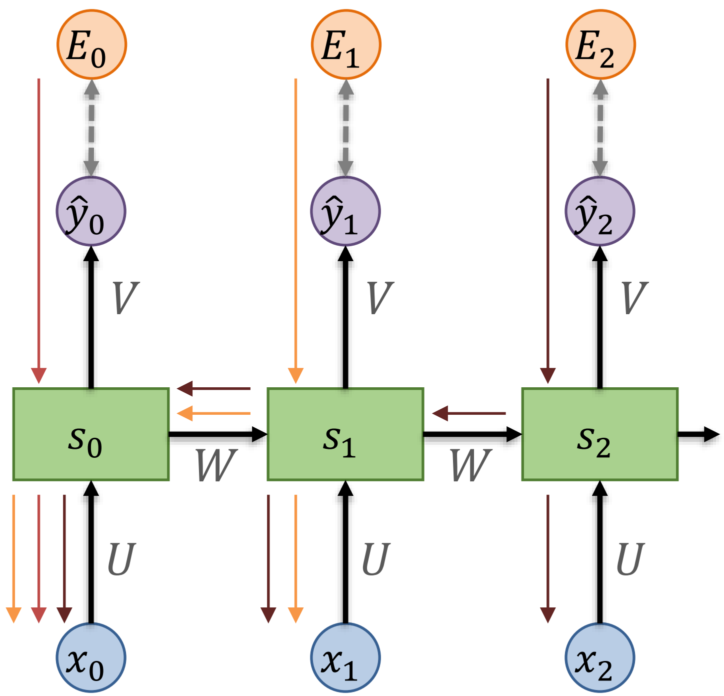
Backpropagation through time



Loss function:

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

Backpropagation through time

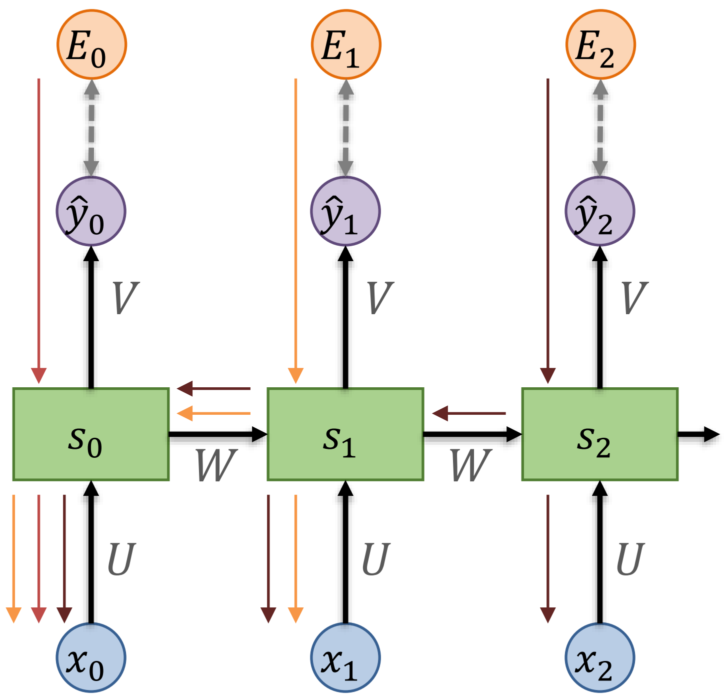


Loss function:

$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

Backpropagate (apply chain rule)

Backpropagation through time



Loss function:

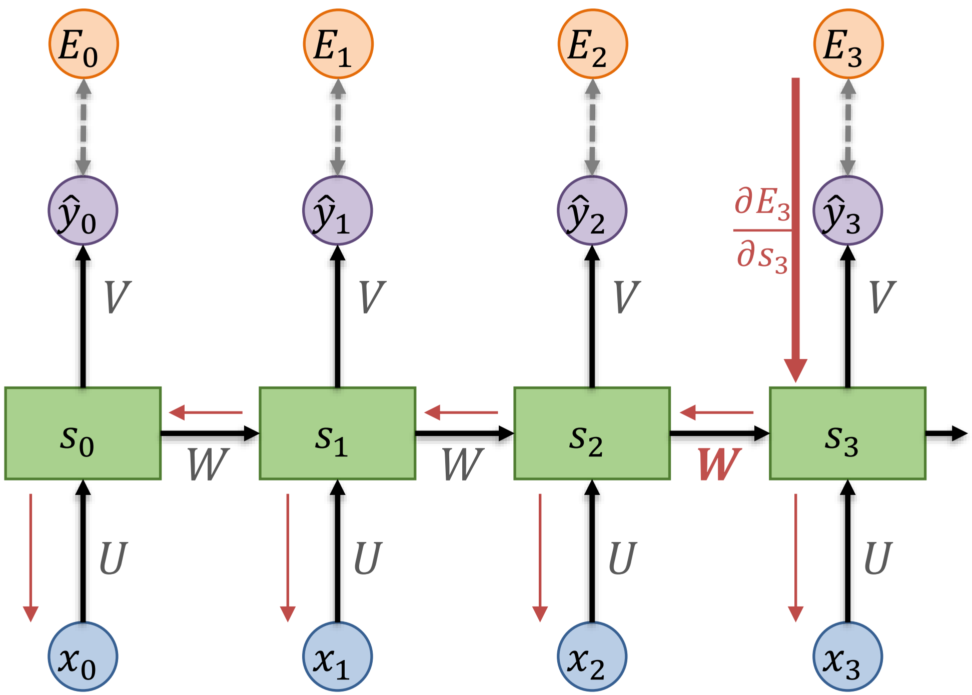
$$E(y, \hat{y}) = \sum_t E_t(y_t, \hat{y}_t)$$

Backpropagate (apply chain rule)

Accumulate gradients:

$$\frac{\partial E}{\partial U} = \sum_t \frac{\partial E_t}{\partial U} \qquad \frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W} \qquad \frac{\partial E}{\partial V} = \sum_t \frac{\partial E_t}{\partial V}$$

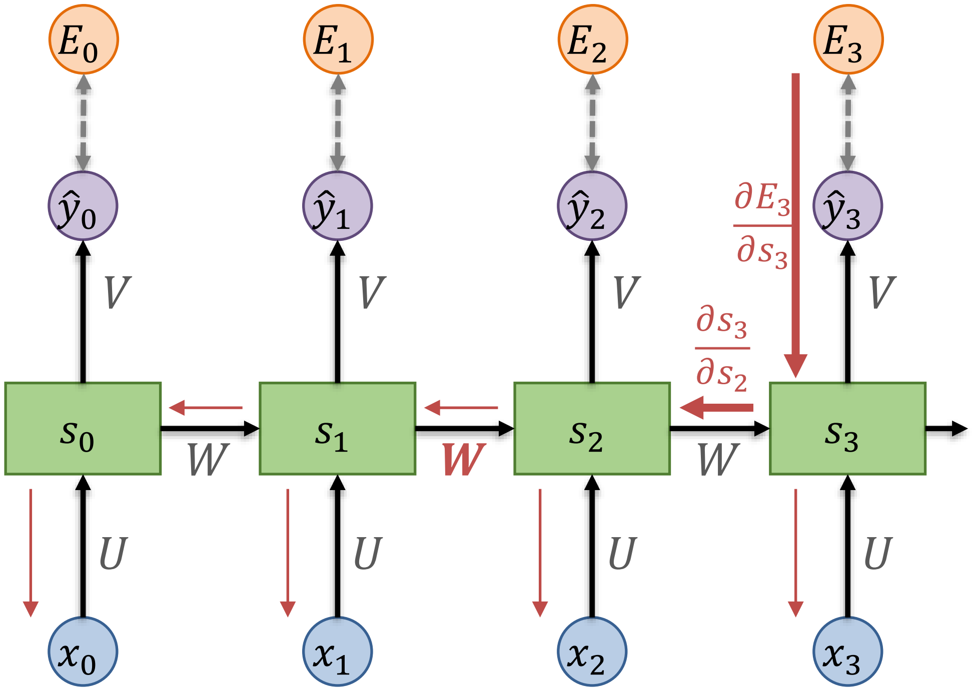
Temporal dependencies



Gradient through time from t = 3:

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial s_3}\frac{\partial s_3}{\partial W} +$$

Temporal dependencies

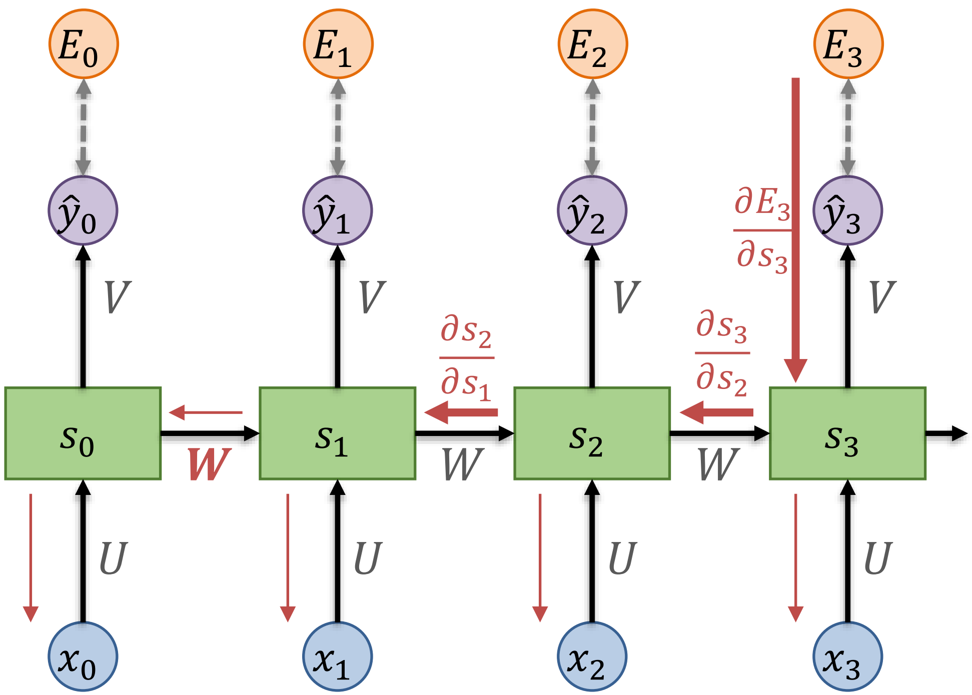


Note that weights are shared

Gradient through time from t = 3:

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial s_3}\frac{\partial s_3}{\partial W} + \frac{\partial E_3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial W} +$$

Temporal dependencies

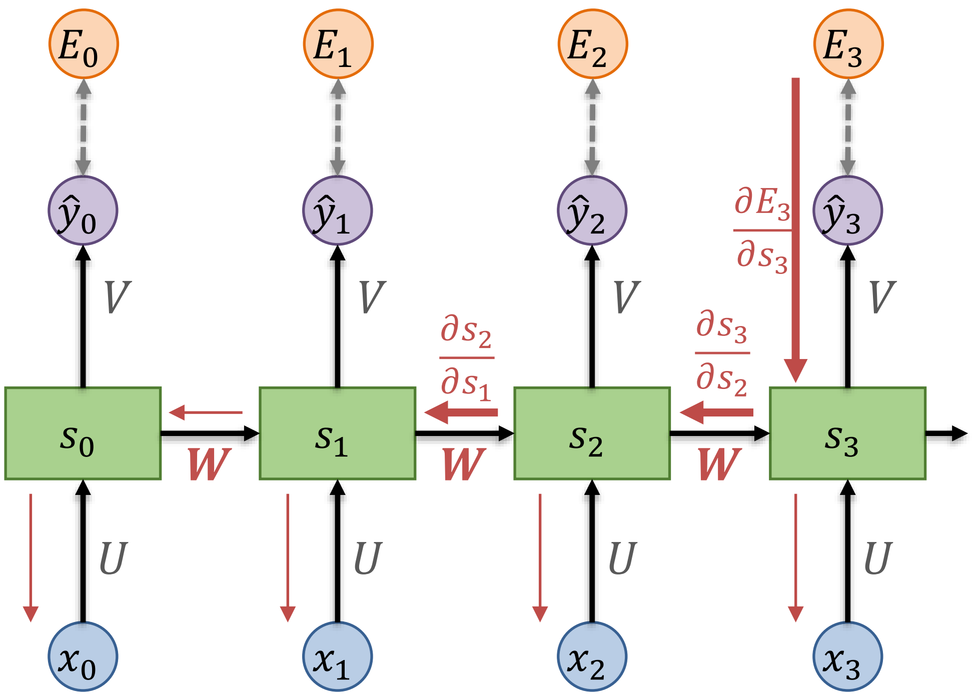


Note that weights are shared

Gradient through time from t = 3:

$$\frac{\partial E_3}{\partial W} = \frac{\partial E_3}{\partial s_3}\frac{\partial s_3}{\partial W} + \frac{\partial E_3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial W} + \frac{\partial E_3}{\partial s_3}\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W}$$

Temporal dependencies



Gradient in time from t back to k:

$$\frac{\partial E_t}{\partial W} = \sum_{k=1}^{t} \frac{\partial E_t}{\partial s_t} \frac{\partial s_t}{\partial s_k} \frac{\partial s_k}{\partial W}$$

$$\frac{\partial s_t}{\partial s_k} = \prod_{i=k}^{t} \frac{\partial s_i}{\partial s_{i-1}}$$

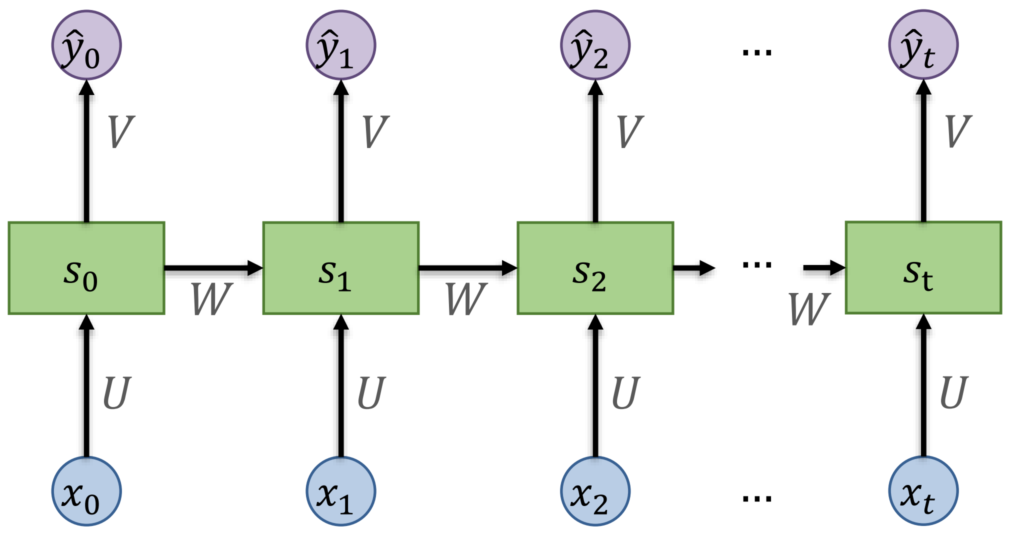
The term $\frac{\partial s_t}{\partial s_k}$ becomes exponentially small / large as time differences increase, leading to **exploding or vanishing gradients**

Exploding gradients: gradient norm clipping or element wise gradient clipping

Vanishing gradients: LSTM and GRU

LONG SHORT-TERM MEMORY (LSTM)

Up to now: Elman RNN



Up to now: Elman RNN

hidden state: Working memory capability that carries information from immediately previous events and overwrites at every step uncontrollably -present at RNNs and LSTMs

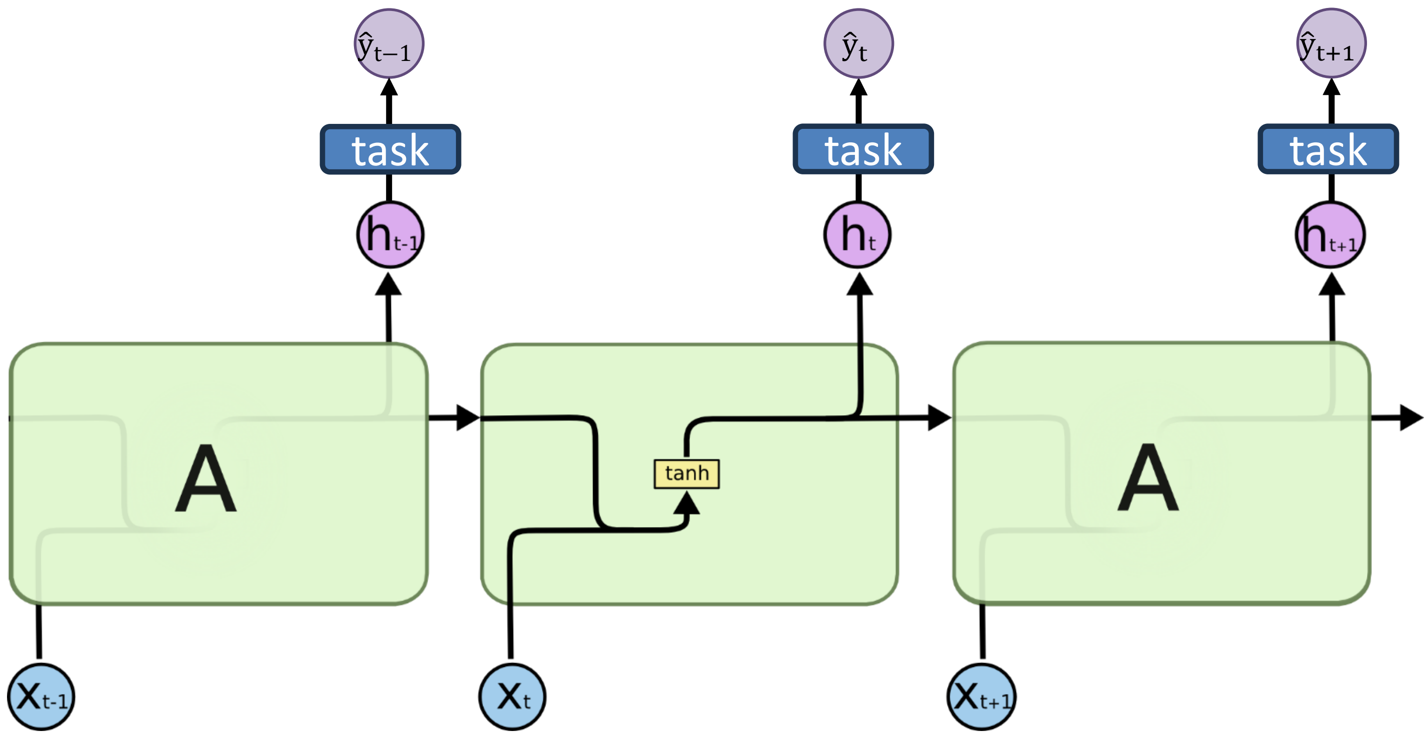
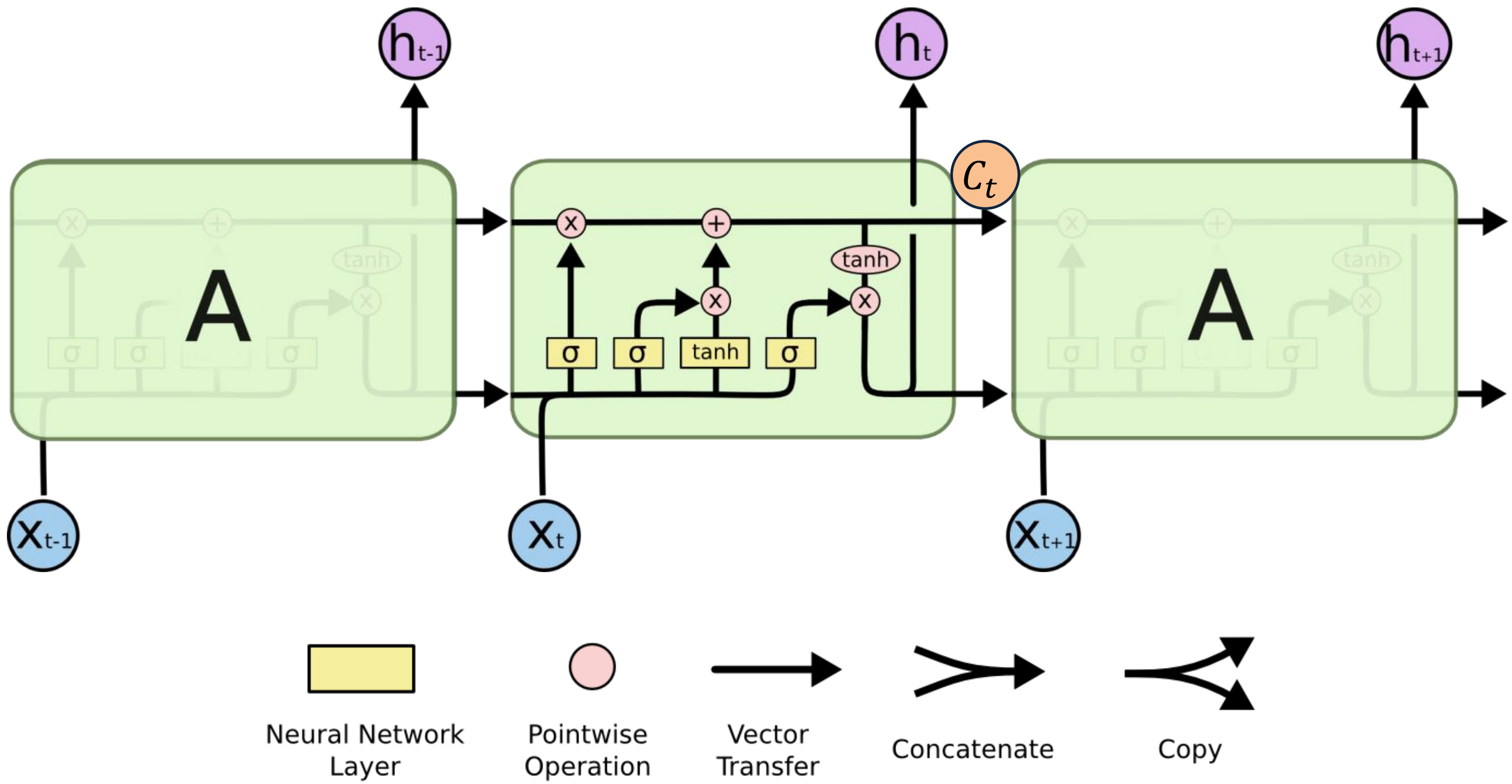


Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory (LSTM)

hidden state: Working memory capability that carries information from immediately previous events and overwrites at every step uncontrollably -present at RNNs and LSTMs

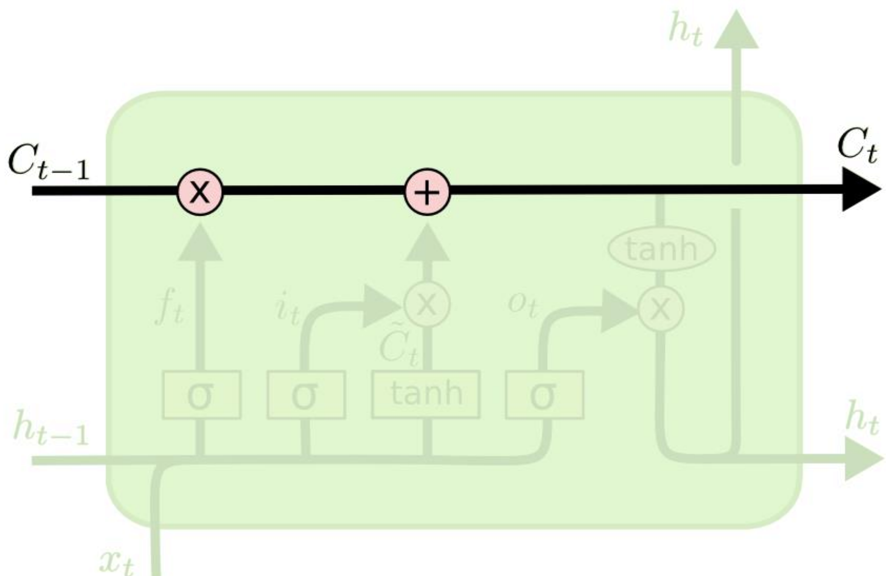
cell state: long term memory capability that stores and loads information of not necessarily immediately previous events



S. Hochreiter, J. Schmidhuber, "Untersuchungen zu dynamischen neuronalen Netz" 1991

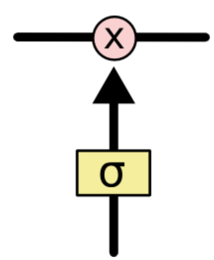
Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory (LSTM)



The cell state (C_t) is a straight path down the entire chain with minor linear interactions

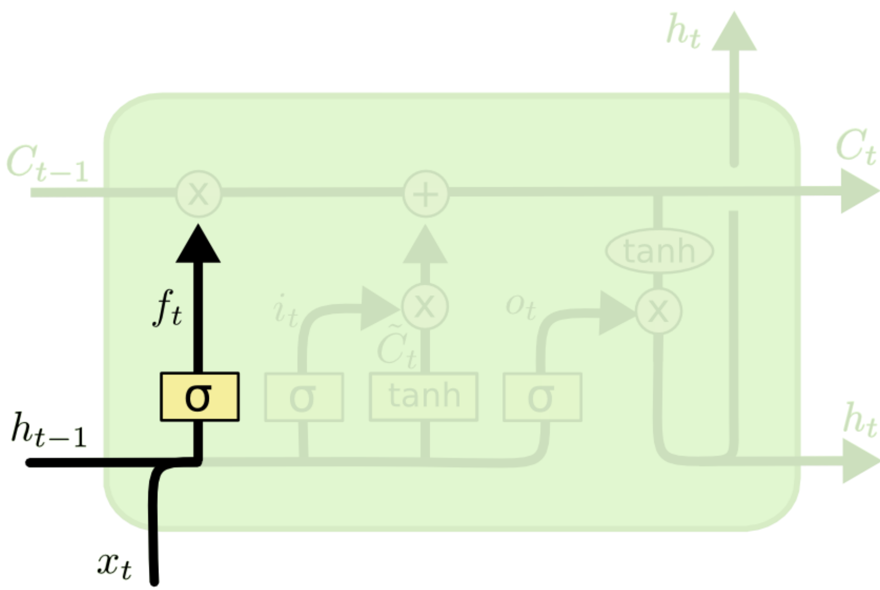
In every step we remove or add information to the cell state via **gates**



A **gate** defines how much information should be allowed to pass through

S. Hochreiter, J. Schmidhuber, "Untersuchungen zu dynamischen neuronalen Net" 1991
Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory (LSTM)



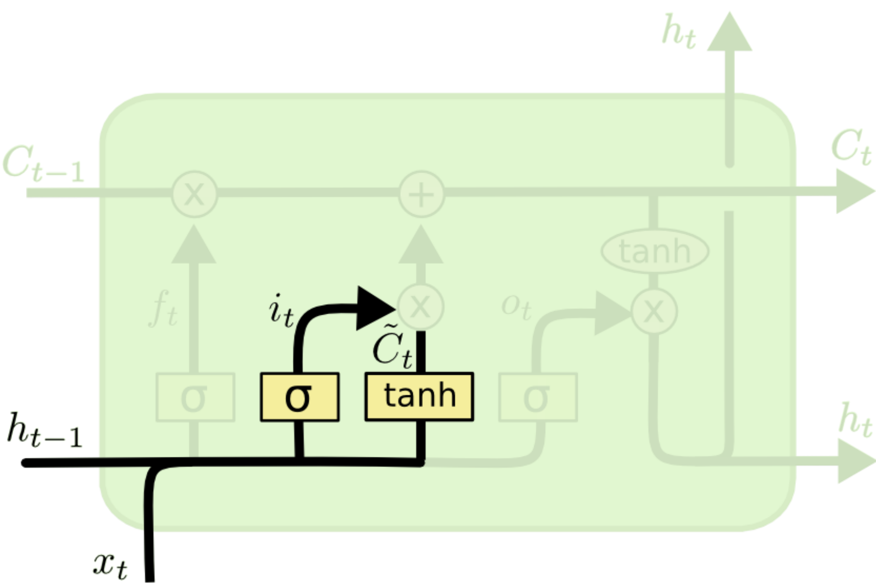
Forget gate: decide what info we should through away from the previous cell state

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

1: keep information
0: delete information

S. Hochreiter, J. Schmidhuber, "Untersuchungen zu dynamischen neuronalen Netz" 1991
Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory (LSTM)



Input gate: decide what new info we should add to the previous cell state

Create new candidate values for the cell state

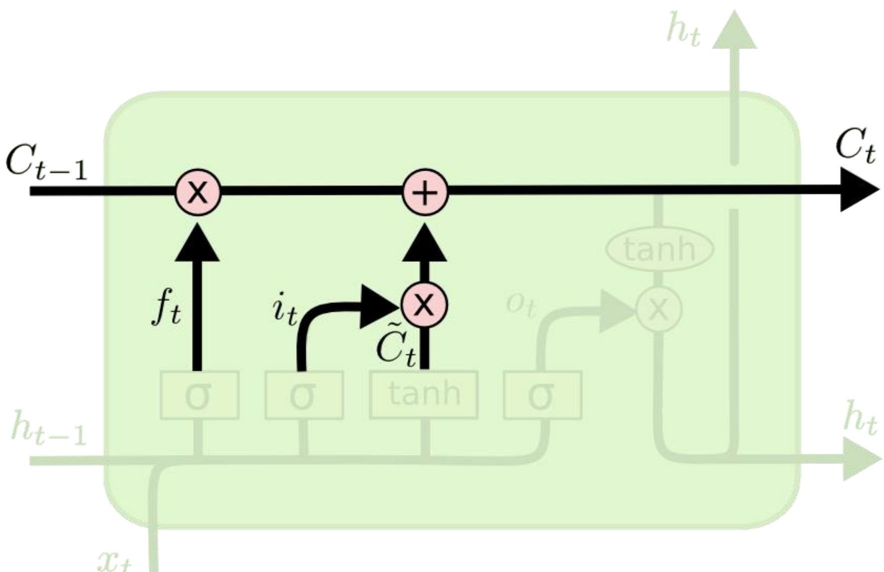
$$\tilde{C}_t = tanh(W_C[h_{t-1}, x_t] + b_C)$$

Decide what part of this new info we should add to the previous cell state

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

S. Hochreiter, J. Schmidhuber, “Untersuchungen zu dynamischen neuronalen Net“ 1991
Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory (LSTM)

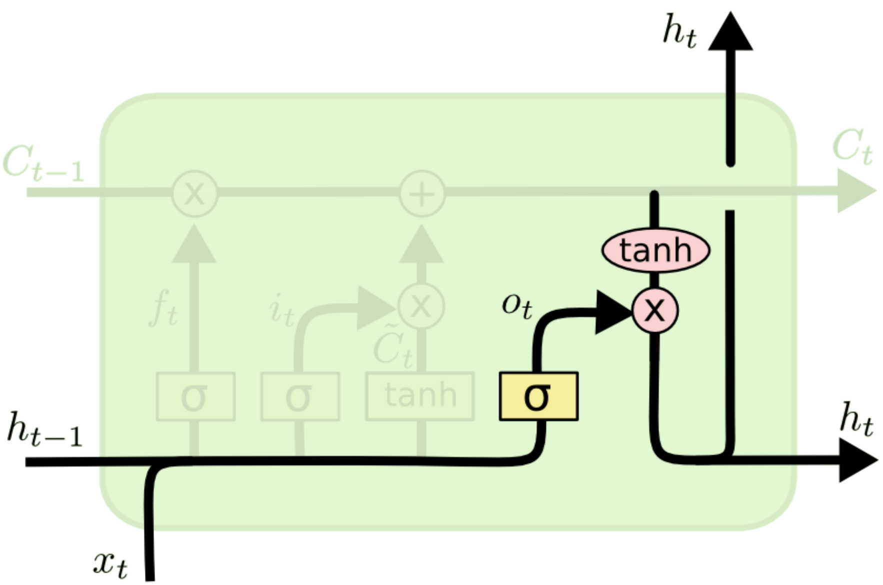


Update the cell state

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

S. Hochreiter, J. Schmidhuber, "Untersuchungen zu dynamischen neuronalen Net" 1991
Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short Term Memory (LSTM)



Calculate the output (h_t)

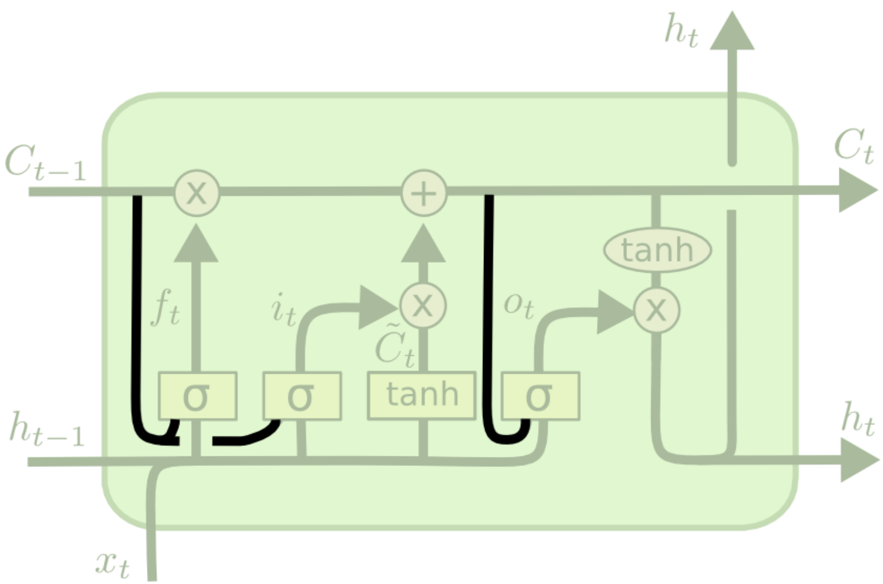
$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$$

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

S. Hochreiter, J. Schmidhuber, "Untersuchungen zu dynamischen neuronalen Netz" 1991
Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Peephole connections

Idea: Allow the gates to also peek at the previous cell state



$$f_t = \sigma\big(W_f[C_{t-1}, h_{t-1}, x_t] + b_f\big)$$

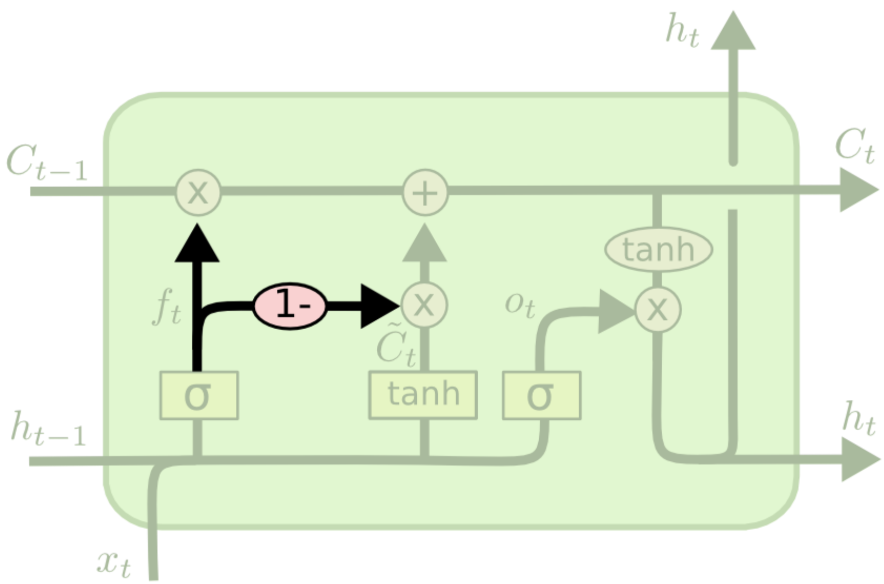
$$i_t = \sigma(W_i[C_{t-1}, h_{t-1}, x_t] + b_i)$$

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

F.A. Gers, J. Schmidhuber, "Recurrent nets that time and count" 2000
Image source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Coupled forget and input gates

Idea: Use a single gate to control both the forget and input gates



$$C_t = f_t C_{t-1} + (1 - f_t)\tilde{C}_t$$

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

LSTM in PyTorch

Docs > torch.nn > LSTM

LSTM

CLASS `torch.nn.LSTM(*args, **kwargs)` [SOURCE]

Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
i_t &= \sigma(W_{ii} x_t + b_{ii} + W_{hi} h_{t-1} + b_{hi}) \\
f_t &= \sigma(W_{if} x_t + b_{if} + W_{hf} h_{t-1} + b_{hf}) \\
g_t &= \tanh(W_{ig} x_t + b_{ig} + W_{hg} h_{t-1} + b_{hg}) \\
o_t &= \sigma(W_{io} x_t + b_{io} + W_{ho} h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned}
$$

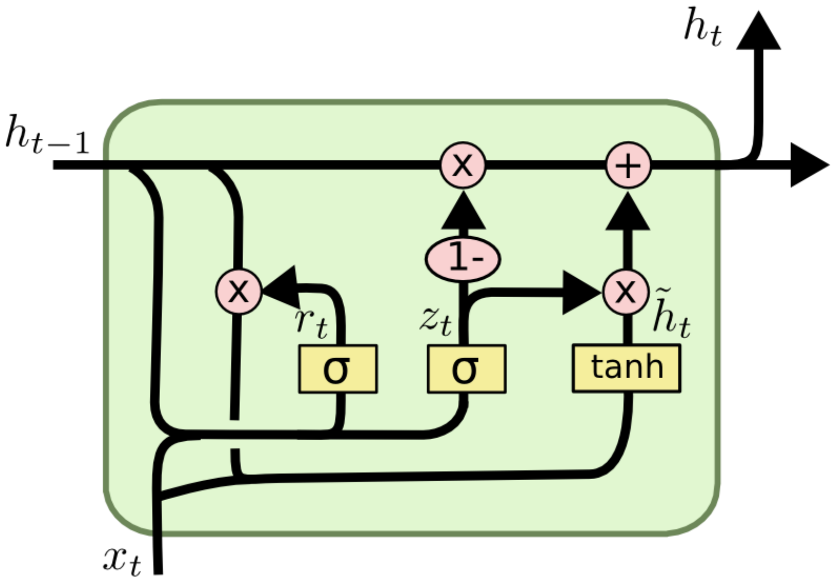
Parameters

- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two LSTMs together to form a *stacked LSTM*, with the second LSTM taking in outputs of the first LSTM and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each LSTM layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional LSTM. Default: `False`
- **proj_size** – If `> 0`, will use LSTM with projections of corresponding size. Default: 0

https://pytorch.org/docs/stable/generated/torch.nn.LSTM.html#torch.nn.LSTM

GATED RECURRENT UNITS (GRU)

Gated Recurrent Unit (GRU)



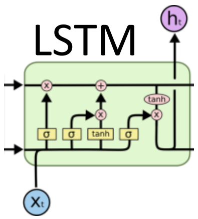
Update gate: decide what part of the previous state to keep and what to remove

$$z_t = \sigma(W_z[h_{t-1}, x_t])$$

Reset gate: decide what part of the previous state to combine with the new input

$$r_t = \sigma(W_r[h_{t-1}, x_t])$$

Input and forget gates are combined into a single update gate

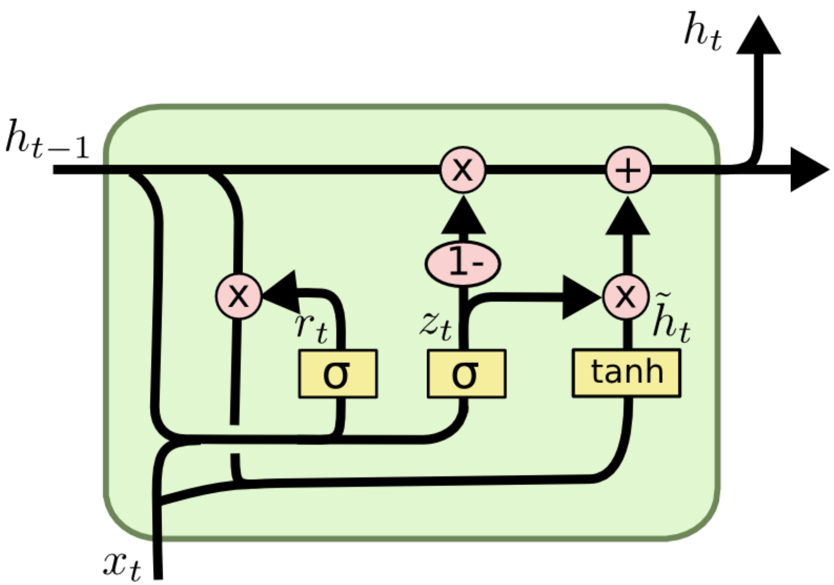


Internal memory and hidden state are merged into a single variable

Cho et al, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", 2014

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

Gated Recurrent Unit (GRU)



Create new candidature values for the hidden state

$$\widetilde{h_t} = \tanh(W[r_t h_{t-1}, x_t])$$

Compute the new hidden state

$$h_t = (1 - z_t)h_{t-1} + z_t\widetilde{h_t}$$

Cho et al, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation", 2014

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

LSTM vs GRU

LSTM	**GRU**
3 gates: Forget, Input, Output	2 gates: Update, Reset
Separate hidden state and cell state	A single hidden state variable
Cell state update	Hidden state update

GRU in PyTorch

Docs > torch.nn > GRU

GRU

CLASS `torch.nn.GRU(*args, **kwargs)` [SOURCE]

Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

For each element in the input sequence, each layer computes the following function:

$$
\begin{aligned}
r_t &= \sigma(W_{ir}x_t + b_{ir} + W_{hr}h_{(t-1)} + b_{hr}) \\
z_t &= \sigma(W_{iz}x_t + b_{iz} + W_{hz}h_{(t-1)} + b_{hz}) \\
n_t &= \tanh(W_{in}x_t + b_{in} + r_t * (W_{hn}h_{(t-} \\
h_t &= (1 - z_t) * n_t + z_t * h_{(t-1)}
\end{aligned}
$$

Parameters

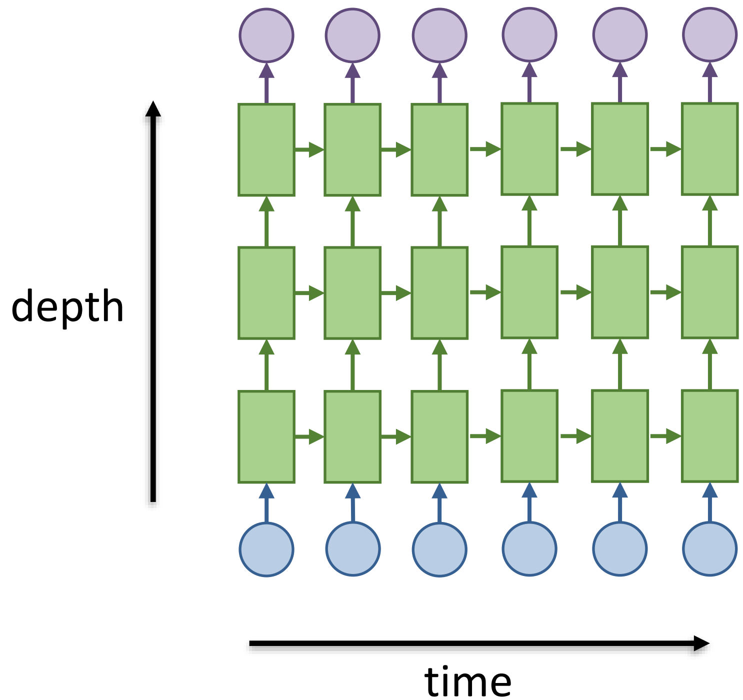
- **input_size** – The number of expected features in the input *x*
- **hidden_size** – The number of features in the hidden state *h*
- **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two GRUs together to form a *stacked GRU*, with the second GRU taking in outputs of the first GRU and computing the final results. Default: 1
- **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
- **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each GRU layer except the last layer, with dropout probability equal to `dropout`. Default: 0
- **bidirectional** – If `True`, becomes a bidirectional GRU. Default: `False`

https://pytorch.org/docs/stable/generated/torch.nn.GRU.html#torch.nn.GRU

DEEP AND BI-DIRECTIONAL RNNs

Deep RNN

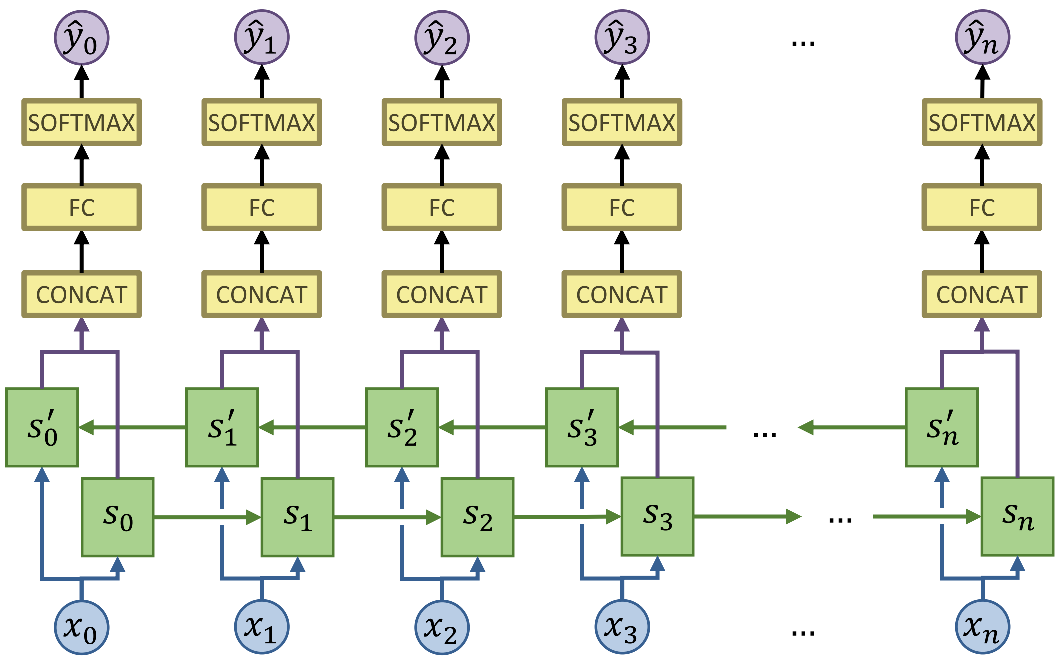
Idea: Stack together multiple layers of RNNs (rarely more than 3)



S. El Hihi, Y. Bengio, "Hierarchical recurrent neural networks for long-term dependencies", 1995
A. Graves, A. Mohamed, G. Hinton, "Speech recognition with deep recurrent neural networks", 2013
R. Pascanu, C Gulcehre, K Cho, Y Bengio, "How to construct deep recurrent neural networks", 2014

Bidirectional RNN

Idea: Incorporate both past and future context



State update (transition function):

$$s_t = f_{U,W}(x_t, s_{t-1})$$

$$s'_t = f_{U',W'}(x_t, s'_{t+1})$$

Left and right cells have different parameters

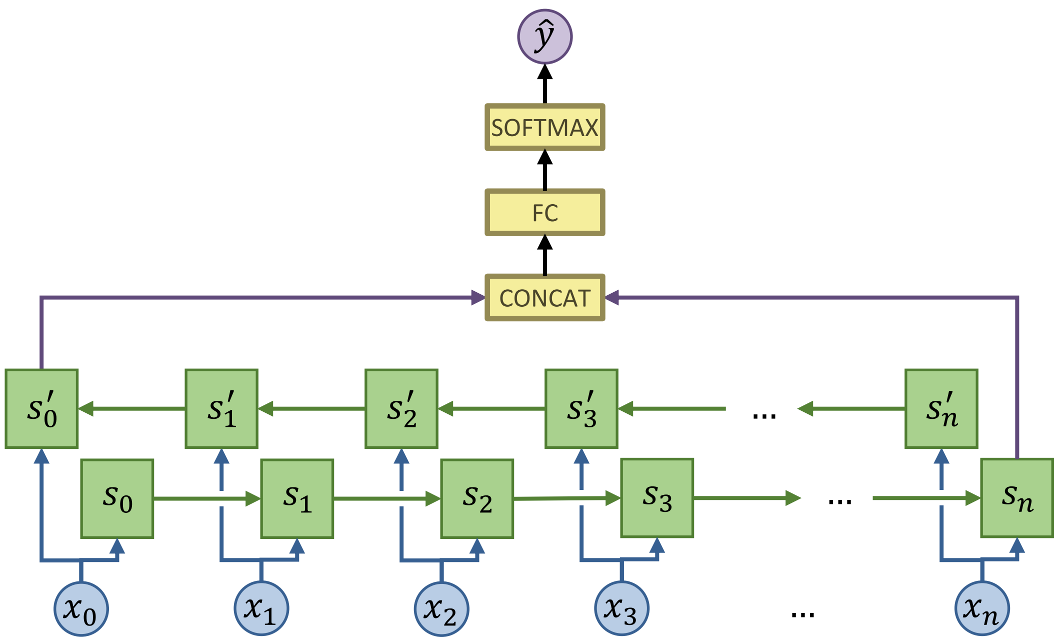
Output function:

$$\hat{y}_t = f_V([s_t, s'_t])$$

Concatenate

Bidirectional RNN

Careful: when a single output is required, we concatenate the "last" hidden states from each direction



Output function:

$$\hat{y}_t = f_V([s_t, s'_0])$$

Concatenate

Deep and Bidirectional RNNs in PyTorch

Docs > torch.nn

Recurrent Layers

`nn.RNN`	Applies a multi-layer Elman RNN with $\tanh$ or ReLU non-linearity to an input sequence.
`nn.LSTM`	Applies a multi-layer long short-term memory (LSTM) RNN to an input sequence.
`nn.GRU`	Applies a multi-layer gated recurrent unit (GRU) RNN to an input sequence.

```
# example bidirectional RNN with GRU cells and depth of 3 layers

bi_grus = torch.nn.GRU(input_size=1, hidden_size=1, num_layers=3, bidirectional=True)
```


https://pytorch.org/docs/stable/nn.html#recurrent-layers

WORKING WITH TEXT DATA

Natural Language Processing

Natural language processing (NLP) is a subfield of linguistics, computer science, and artificial intelligence that studies how to analyze natural language data

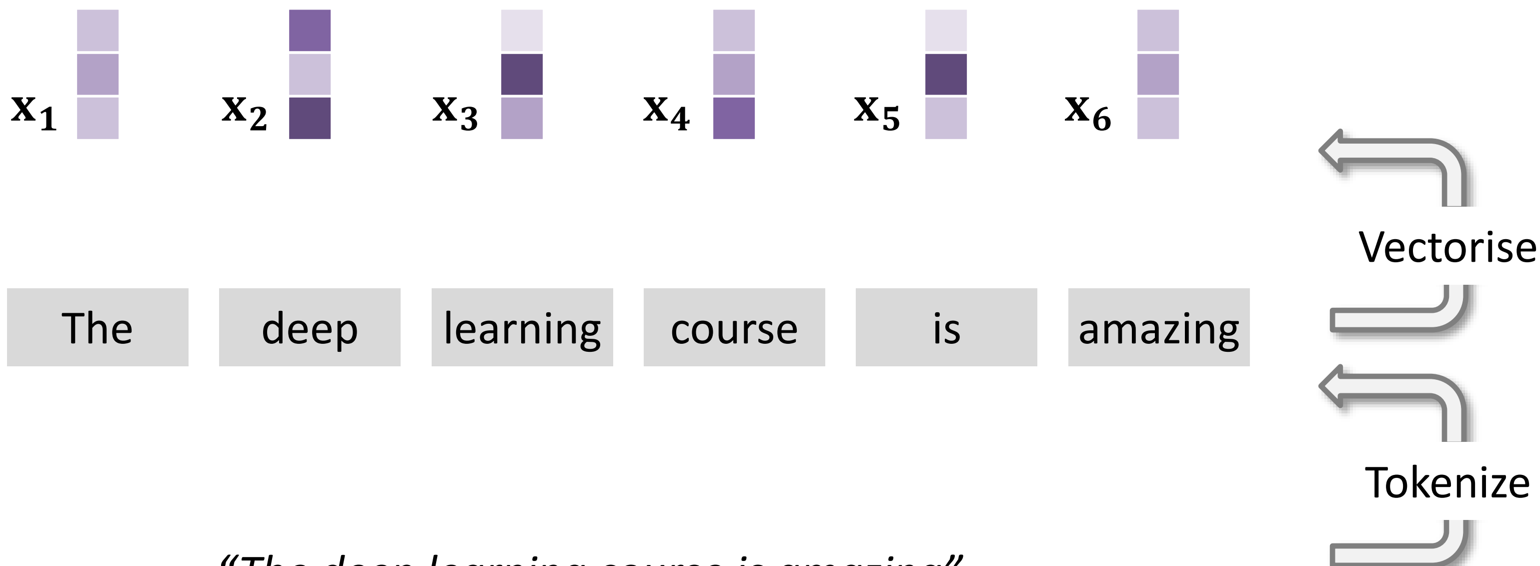
Common NLP tasks:
- Machine translation
- Document summarization
- Question Answering
- Topic modeling
- Part-of-speech tagging
- Sentiment analysis
- Language models for Optical Character Recognition
- Speech recognition
- etc.

Working with text data

Text is one of the most widespread forms of sequential data. It can be understood either as a sequence of characters or as a sequence of words.

Tokenizing text refers to the process of splitting it into a list of “tokens” (words, n-grams, characters)

Vectorizing text is the process of transforming text tokens into numeric values

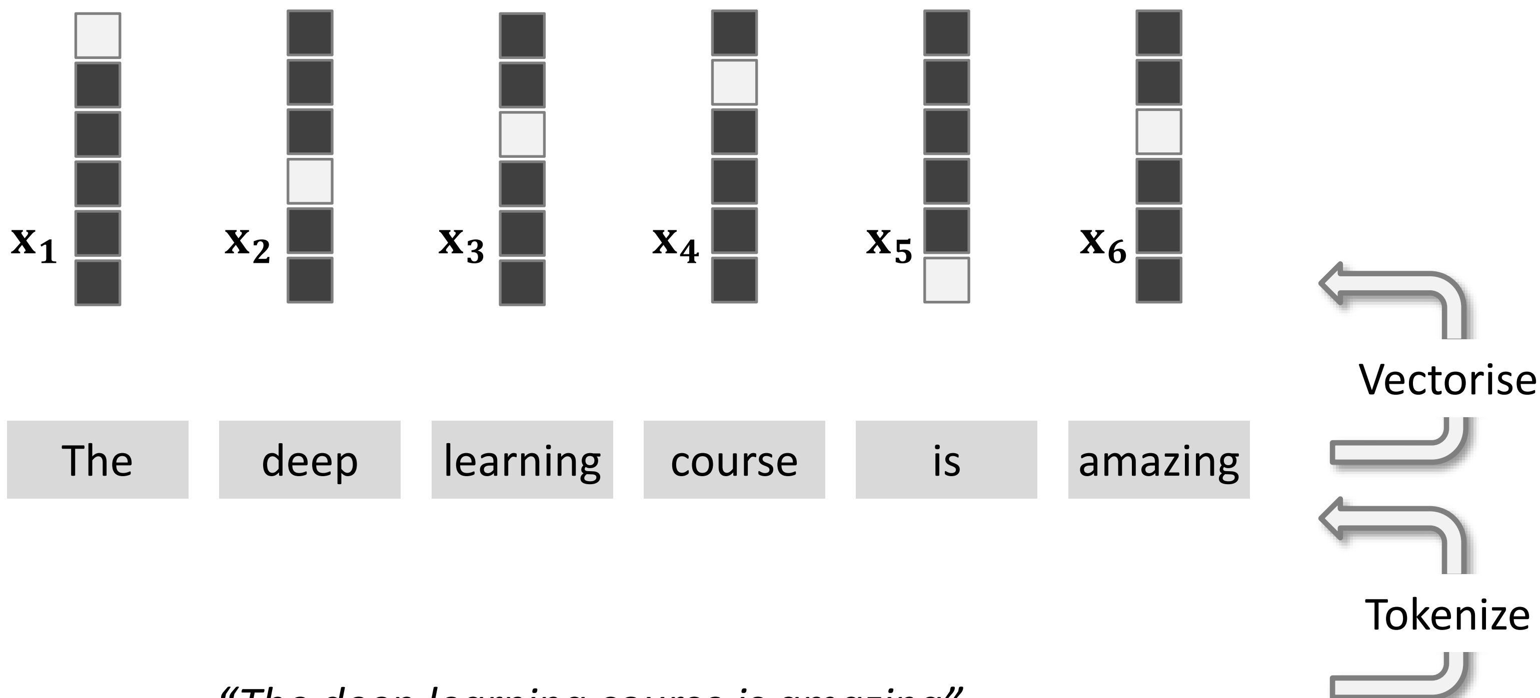


“The deep learning course is amazing”

One-hot encoding

The most common and basic way to turn a “token” into a vector is to use a 1-hot representation:

- Associate a unique integer to each word/character in a fixed vocabulary
- Convert each word into a one-hot vector of size N (the vocabulary size)

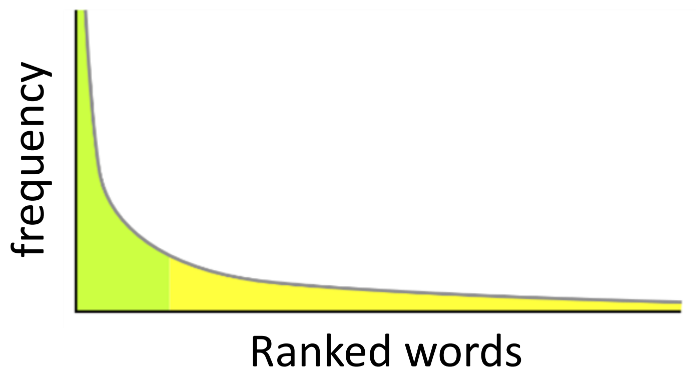


“The deep learning course is amazing”

One-hot encoding of words

How to choose the vocabulary?

Typically: use the top-N most frequent words in your training text corpus



Use a special *<UNK>* token to represent “unknown” out-of-vocabulary words

Depending on the task may also need to define special tokens for the start (*<START>)* and end (<EOS>) of the sequence

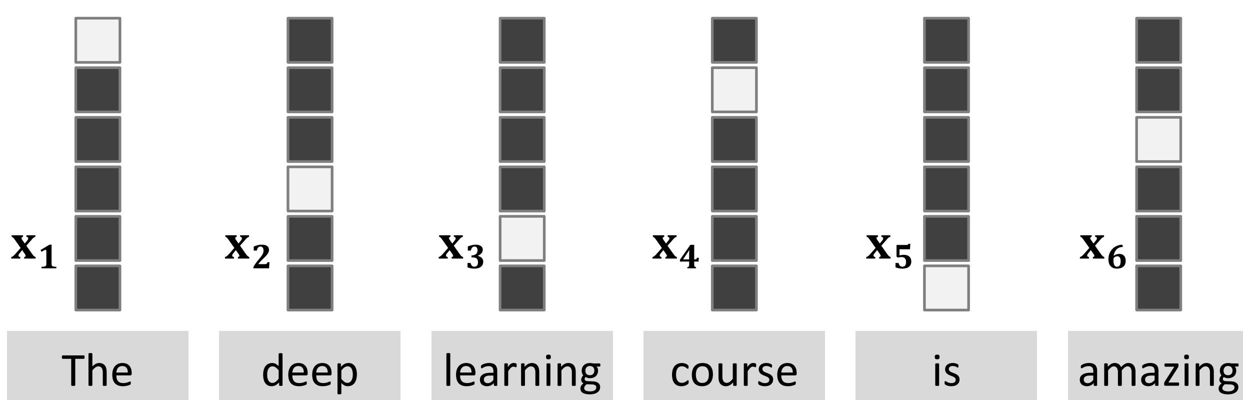
Depending on the task you may also need the tokens for punctuation symbols

Word embeddings

One-hot embeddings do not scale well, the size of the vectors is as big as the vocabulary

Adding and removing words changes all representations

Does not encode any useful semantic information (similar vectors do not imply “similar” words) (*actually, all vectors have the same distance…*)

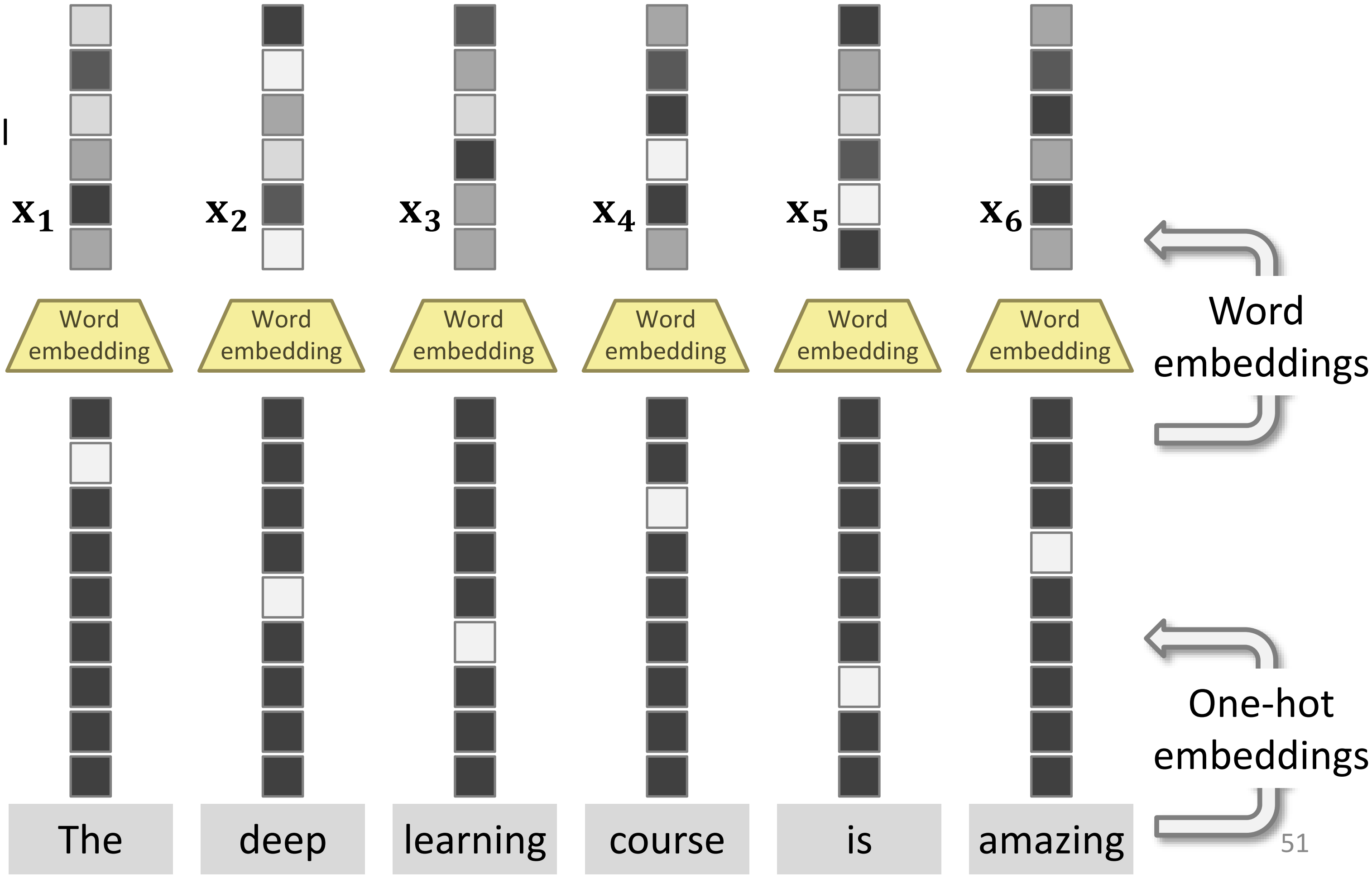


Word embeddings

Idea: project one-hot embeddings into lower dimensionality, dense (continuous), floating-point vectors

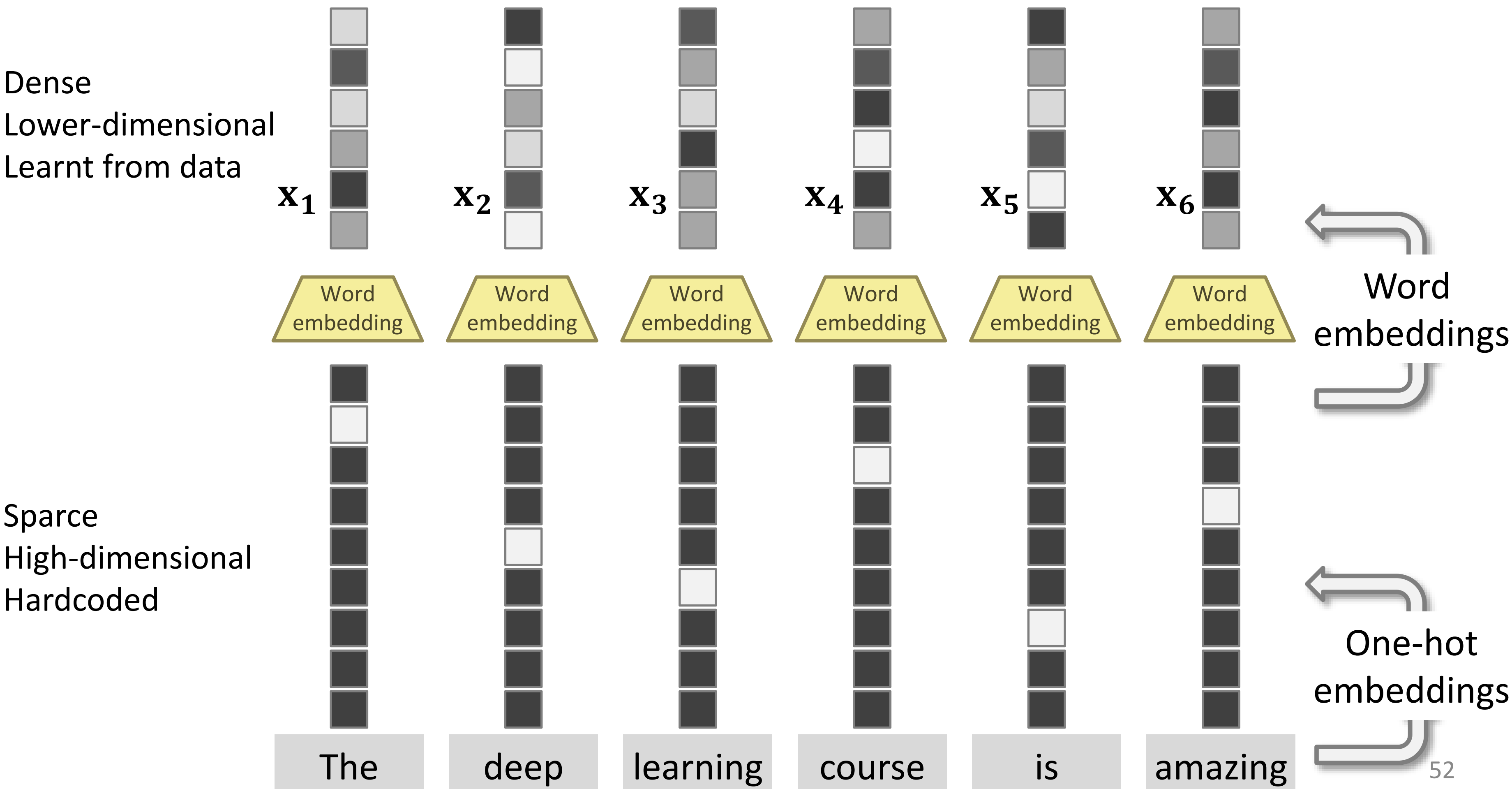
Dense
Lower-dimensional
Learnt from data

Sparse
High-dimensional
Hardcoded



Learning Word embeddings

Either learn them **jointly with the main task** you want to solve in your model or Learn them in **another NLP task (pre-training)** and re-use



Pre-trained word embeddings

Word2Vec is trained for the task of word prediction given its context.

Training set: large corpus of english text.

Sliding window (**self-supervised!**)

[The wide road shimmered] in the hot sun.

The [wide road shimmered in the] hot sun.

The wide road shimmered in [the hot sun].

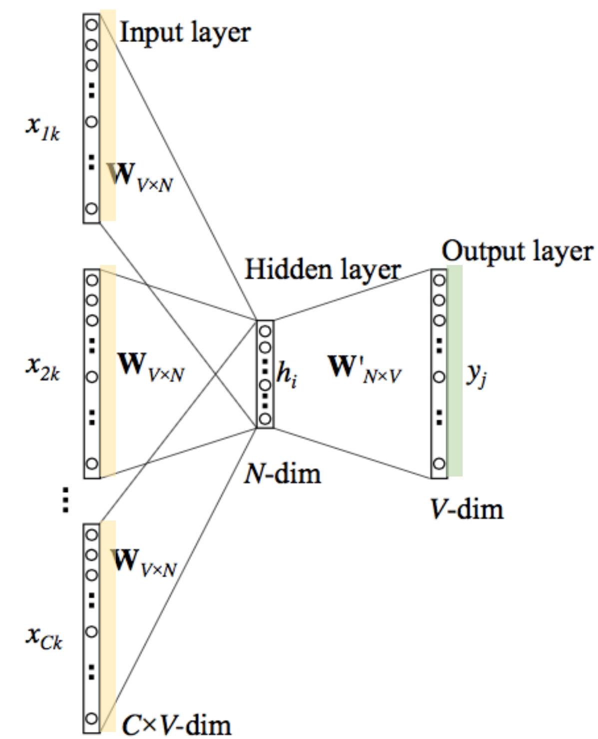
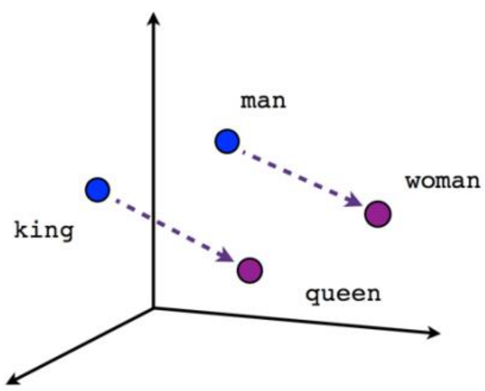


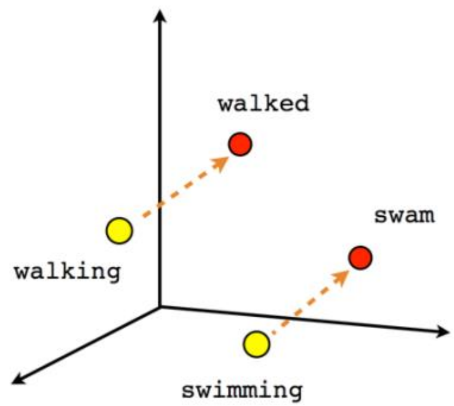
Image source: https://towardsdatascience.com/introduction-to-word-embedding-and-word2vec-652d0c2060fa

Pre-trained word embeddings

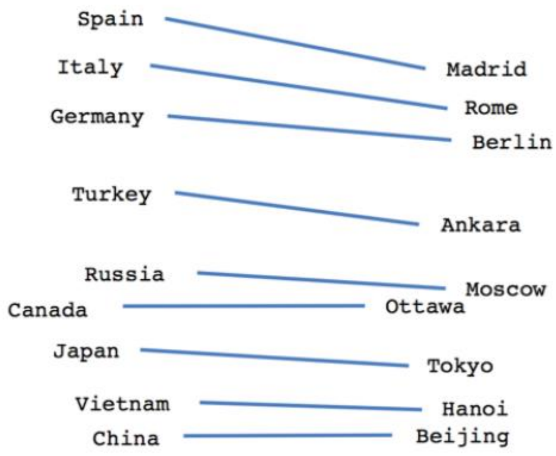
Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located close to one another in the space



Male-Female



Verb tense



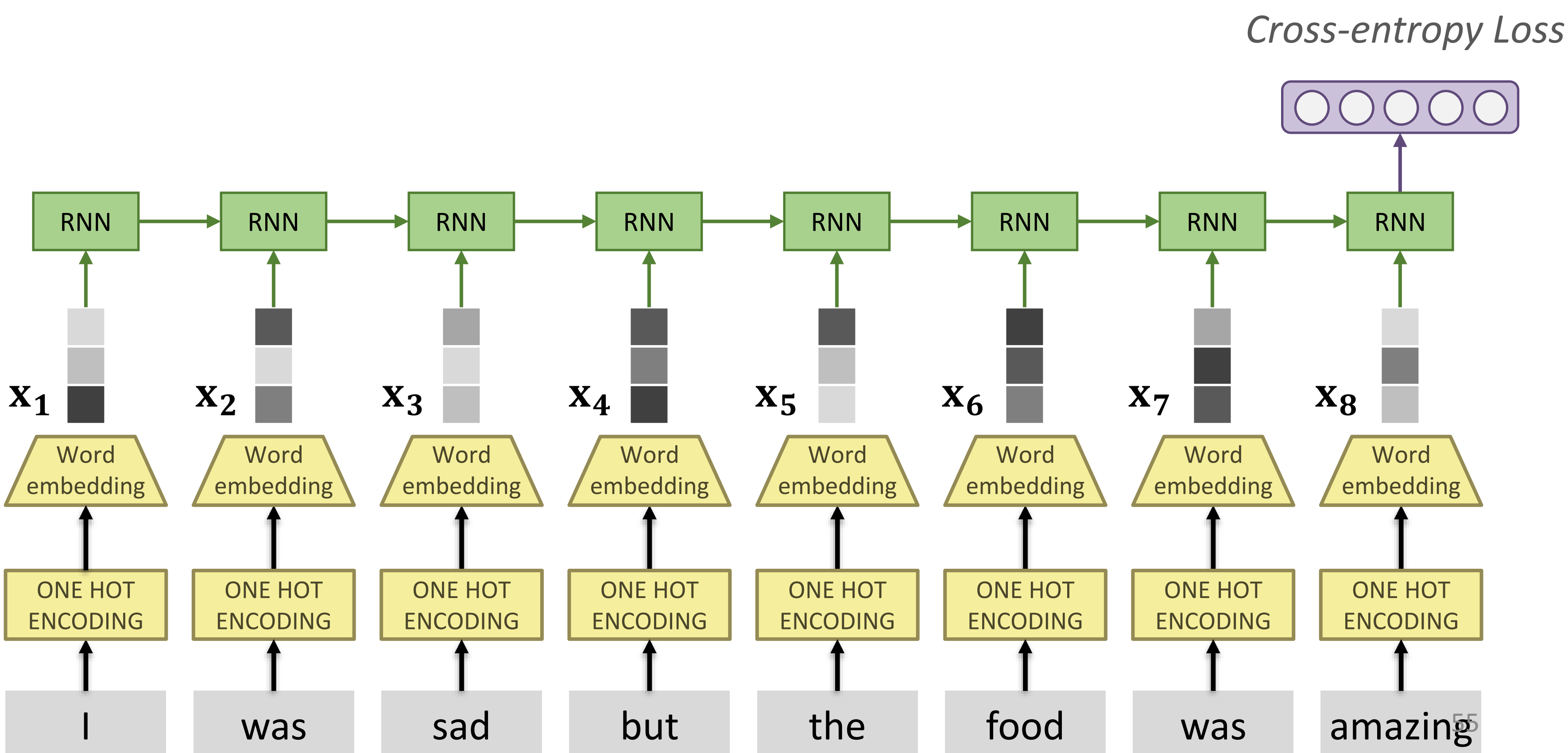
Country-Capital

Image source: https://towardsdatascience.com/introduction-to-word-embeddings-4cf857b12edc

Example: Sentiment Analysis

Model for sentence sentiment analysis

Classification task into five classes: “Very positive”, “Positive”, “Neutral”, “Negative”, “Very negative”



Example: Language Modelling

Speech recognition

"The city is on a **plain**"

"The city is on a **plane**"

It would make sense to use the most probably word, given the context

$$P(\text{“The city is on a plain”}) = \ 2.95 \times 10^{-10}$$

$$P(\text{“The city is on a plane”}) = 3.58 \times 10^{-13}$$

How can we calculate these probabilities?

$$P(w_1, w_2, w_3, \dots, w_n) = ?$$

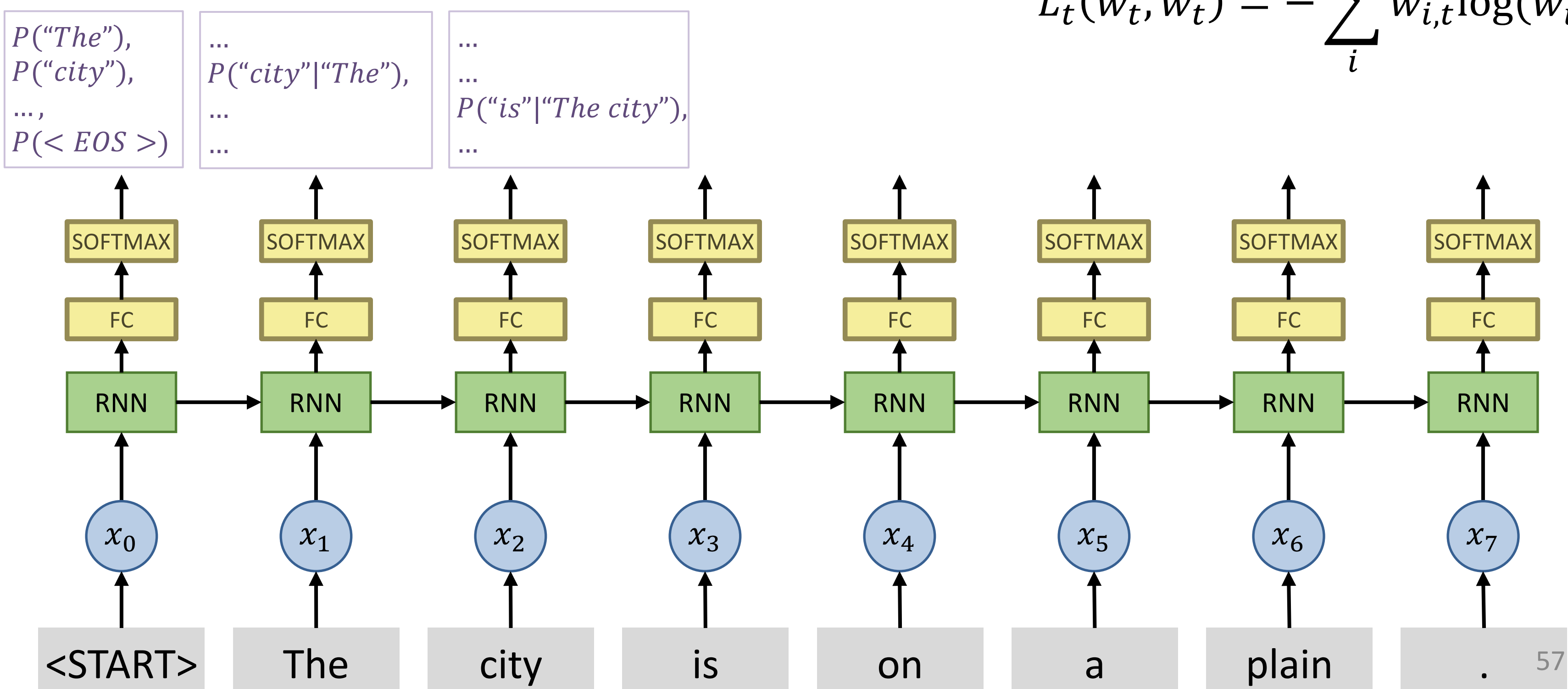
Language modelling with RNNs

`s = “The”, “city”, “is”, “on”, “a”, “plain”, “.”, <EOS>`

Loss Function:

$$L = \sum_t L_t(\widehat{w_t}, w_t)$$

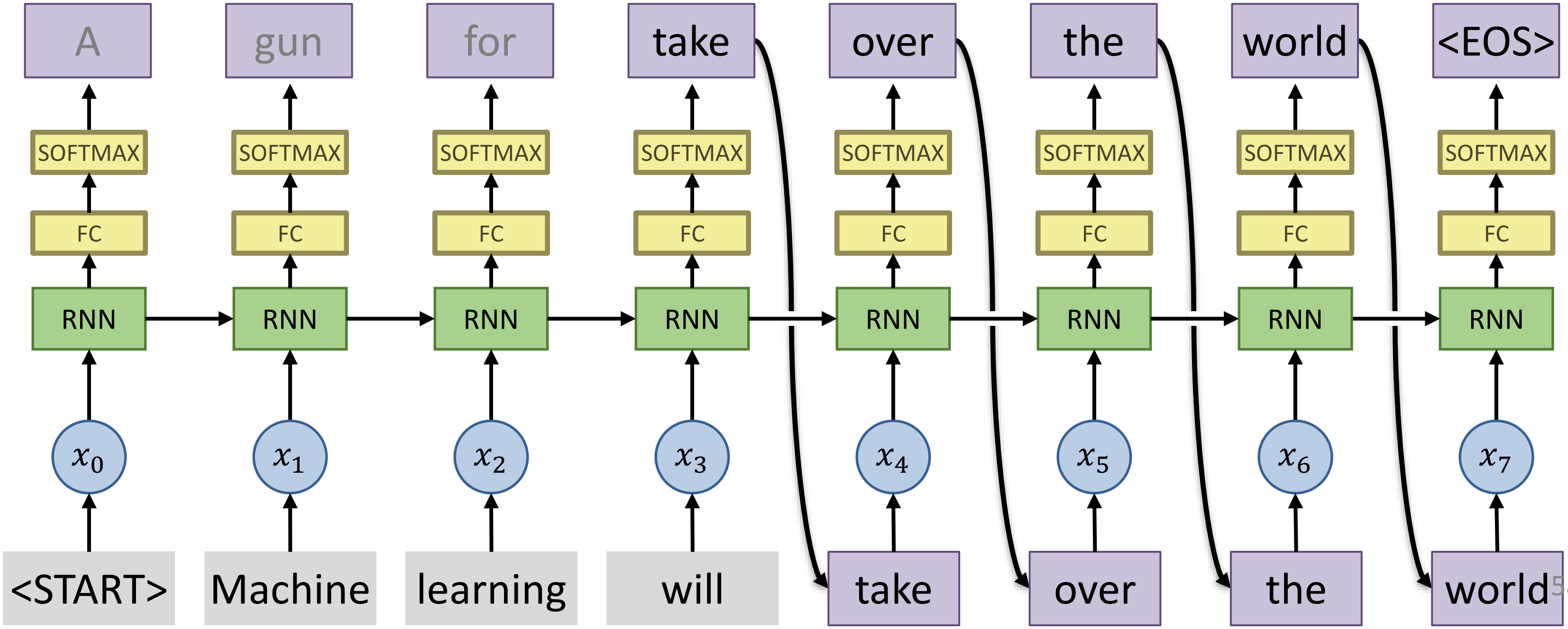
$$L_t(\widehat{w_t}, w_t) = -\sum_i w_{i,t} \log(\widehat{w_{i,t}})$$



Language Modelling with RNNs

Once the model is trained it can be used to generate new sentences!

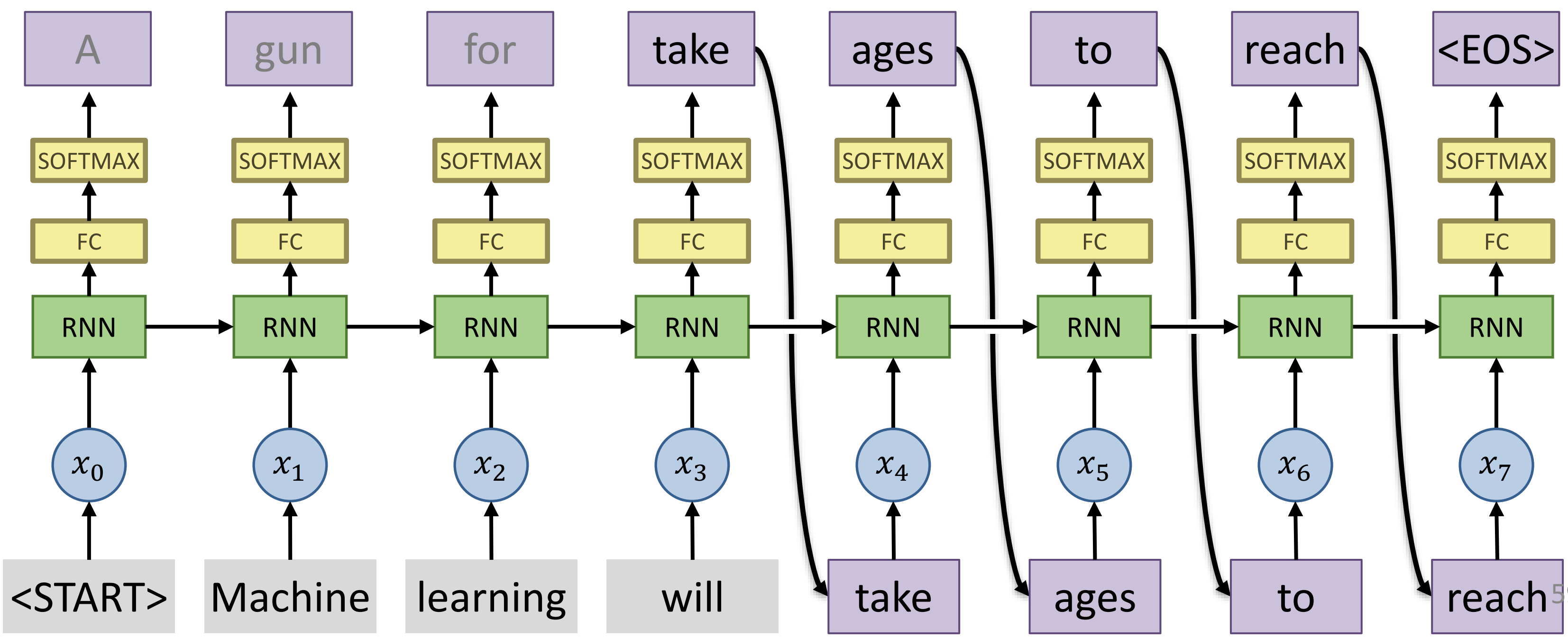
Initialize the model with some words and then feed the predicted word at time t ($\widehat{w_t}$) as the input at time step $t+1$ (x_{t+1}). Then repeat until the special token <EOS> is predicted.



Language Modelling with RNNs

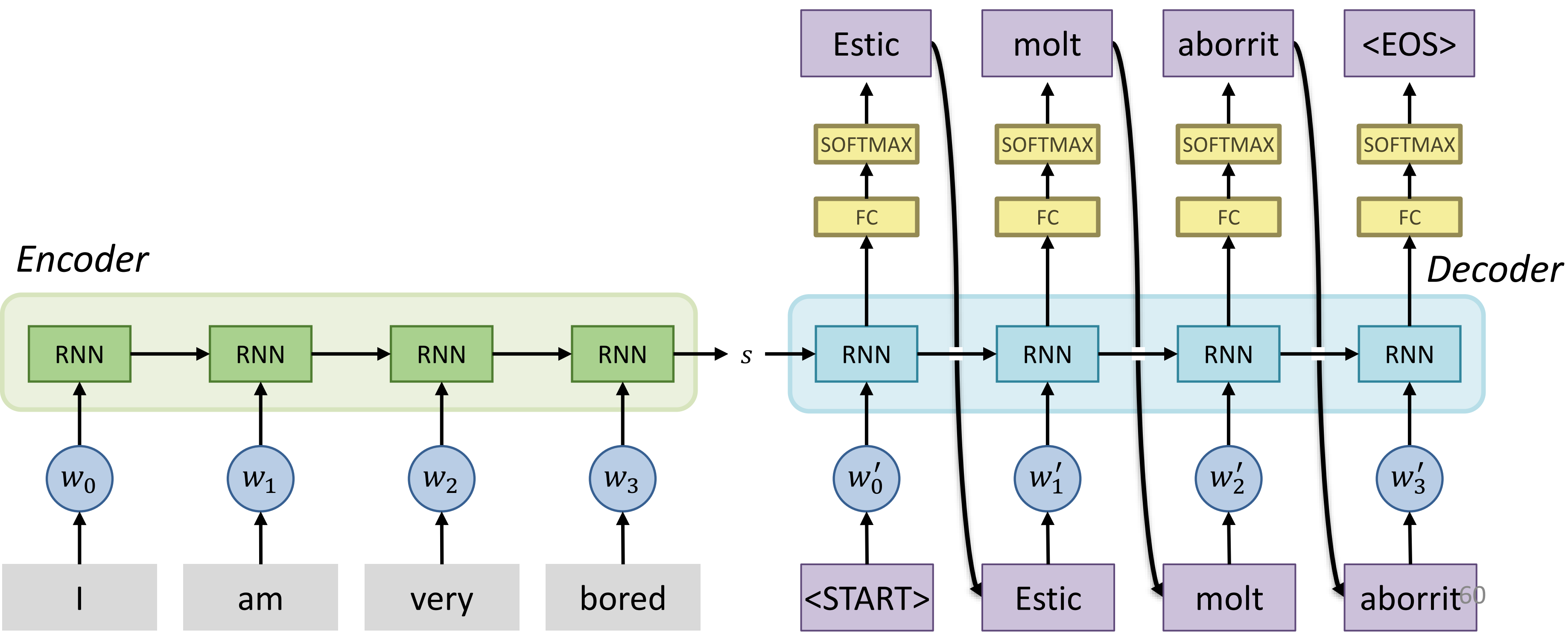
The generation is deterministic… starting with the same sentence will produce the same output

Sample from the softmax distribution instead of taking the argmax. This makes the generated text less predictable / less repetitive. Also reject <UNK> predictions.



Encoder / Decoder Architectures

Model for machine translation.
Classification task (Softmax). As many classes as “words” in target language vocabulary.
Input and output sequences may have different lengths!



Example: Composing music

Training sample:

```
X: 1
T:"Hello world in abc notation"
M:4/4
K:C
"Am" C, D, E, F,|"F" G, A, B, C|"C"D E F G|"G" A B e c
```

By Francesco Marchesani, Daniel Johnson

https://www.danieldjohnson.com/2015/08/03/composing-music-with-recurrent-neural-networks/

500 iterations

```
X::IW|Xn=|aI,X%2*c*nWWWn|,
VomIO"Jjmk"+g6fx24
50*/4nN=TkTwV]2m-|Ro
BJonudINC'"i/JW?
/690!kVk/mPoXkm:6,W<:dcaMgO%
J,I"IuO"A,WD/JIKXIk5suCs,f5 fp%f" JVW_V3@]Ji,9wim/NiOIg9,CNNnn:SNn,
[gwrWnIp)ic/-I,N:N"{V43siAlmLaNn)9B,n>3N]NN"RG"G0DW
VVh]%lBVo+On9"tBNcVkCgdomiNBm"#
,f%N4kk3k:cukX3siC%P)TVzuO/:9VOM,ug6IF f<F \XX*"",<9:"k*4W9I,0E**=2V/Epc4n "
NoXkV,NIn ~c%ZV:BWp'G/,WV}8s,X!mNEsXgBRfruNJf]dn s,,kfnyxlEO,
XC&6VO",koik r:!b%,i2MW3a 2Lg'.]Vf,n]30wW|
:o
Nd~cfP|9,fi /d|gOS!cX~ :_
iT" 5MD]cNs"m|/9mJS>nX"s"I >p :tN|cMii(W:"m/W
gV1sB:kp,W|t3:OR{W9)4p<d#omX"t0s/
```

1000 iterations

```
X:tNotimam Music Database
S:via PR
M:4/4
L:1/4
K:G
d  (3gb-B/2-|"C"d/2D/2d/2B/2A/2 Bz:||


X: 36
T:Raheolen-
```

7200 iterations

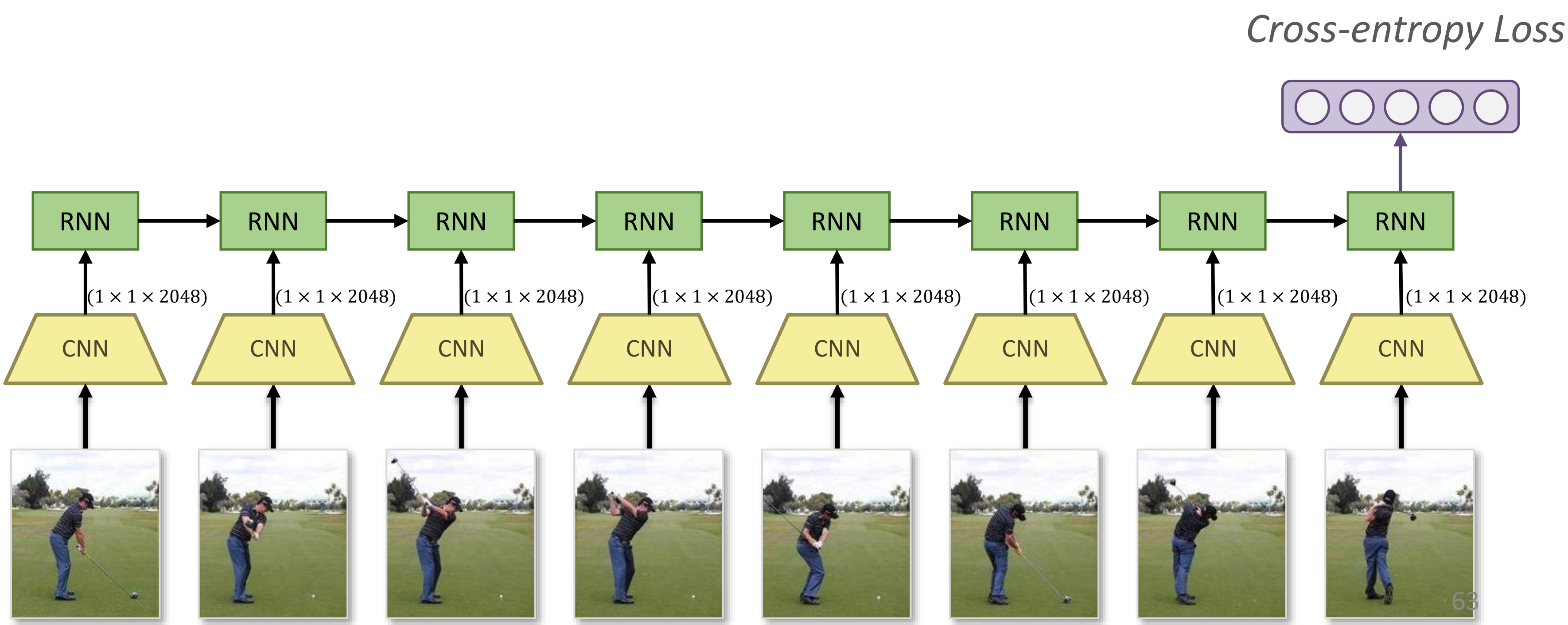
```
X: 26
T:Cherronge The Yenndens
% Nottingham Music Database
P:AAB
S:Gethor Cankied, via PR
M:4/4
L:1/4
K:D
"D"FF df/2a/2|"G"GB Bc/2d/2|"A"e3/2c/2 AE|"Em"EE E2|"A7"E/2E/2F/2G/2 "A7"AA/
2B/2|"D"Ad d::
F/2A/2|"D"d2 d2|"D"f3/2d/2 AA|"Em"Bc Bd/2B/2|"D"DF "A7"D2|"A
ubuntu@ip-172-31-9-240:/mnt/char-rnn-tensorflow$
```

RNNs IN COMPUTER VISION

Example: Video Classification

Use a CNN to extract feature vectors from each frame of the image

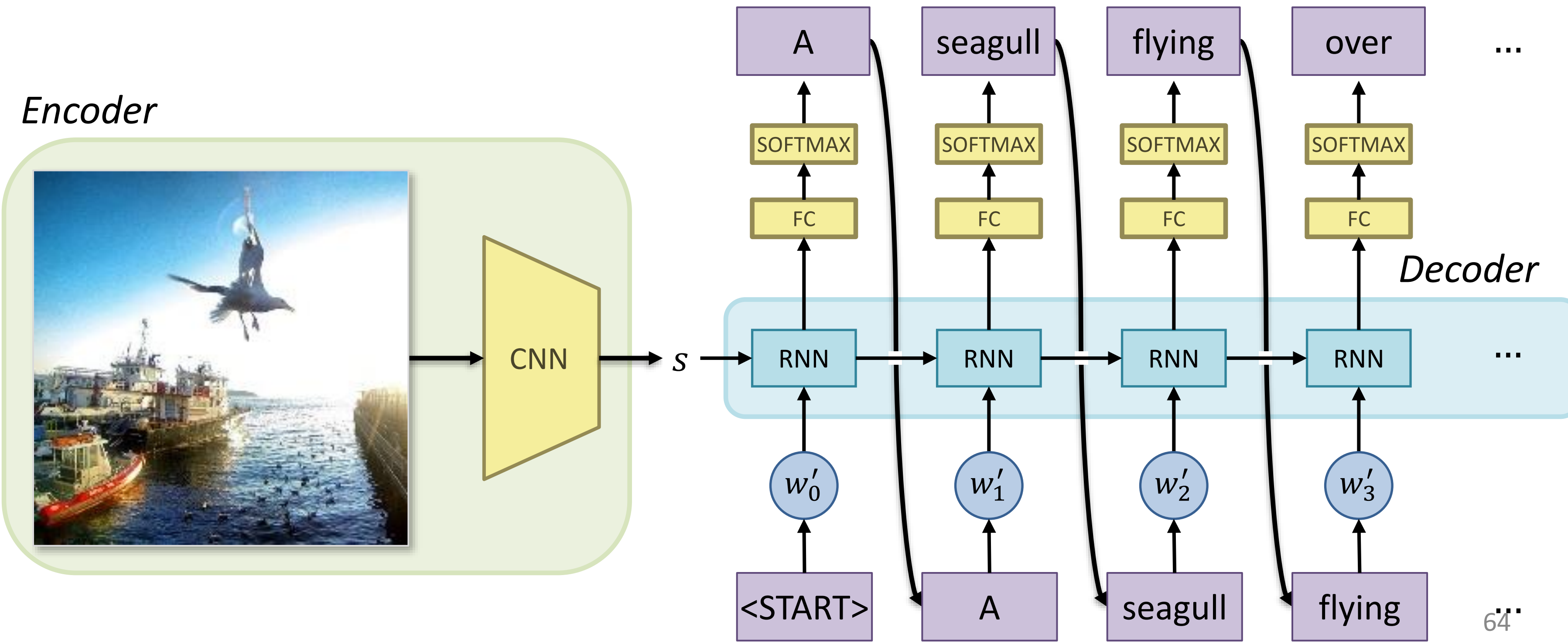
Pre-train the CNN!



Example: Image Captioning

Encode the image information using a pre-trained CNN, then use an RNN to decode it into natural language

Classification task. As many classes as words in target language vocabulary

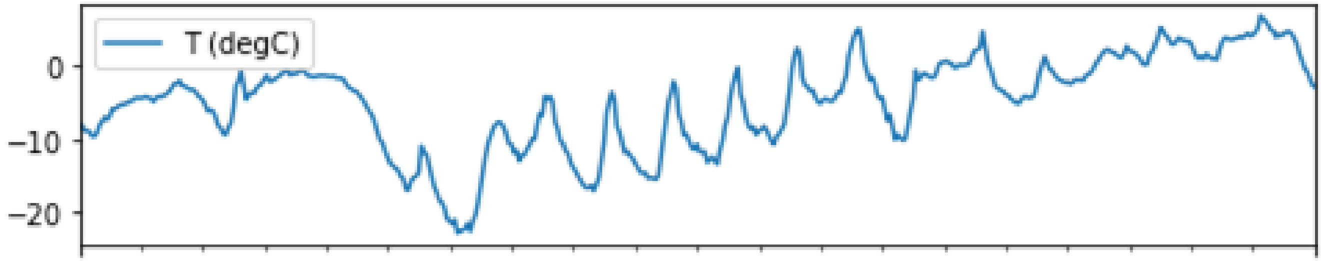


TIME SERIES FORECASTING

Time Series Forecasting

Time-series refers to an ordered series of data in time, and time-series models are used to forecast what comes next in the series by using some previous (ordered) observations.

Typical time-series applications include weather forecasting, stock prices forecasting, etc.



Contrary to typical regression tasks, time-series forecast is an **extrapolation** task while the regression tasks we have seen before are performing **interpolation**.

Image source: https://www.tensorflow.org/tutorials/structured_data/time_series

Example: weather prediction

Continuous prediction: Each time step we feed the measured input to the model, and predict the next step

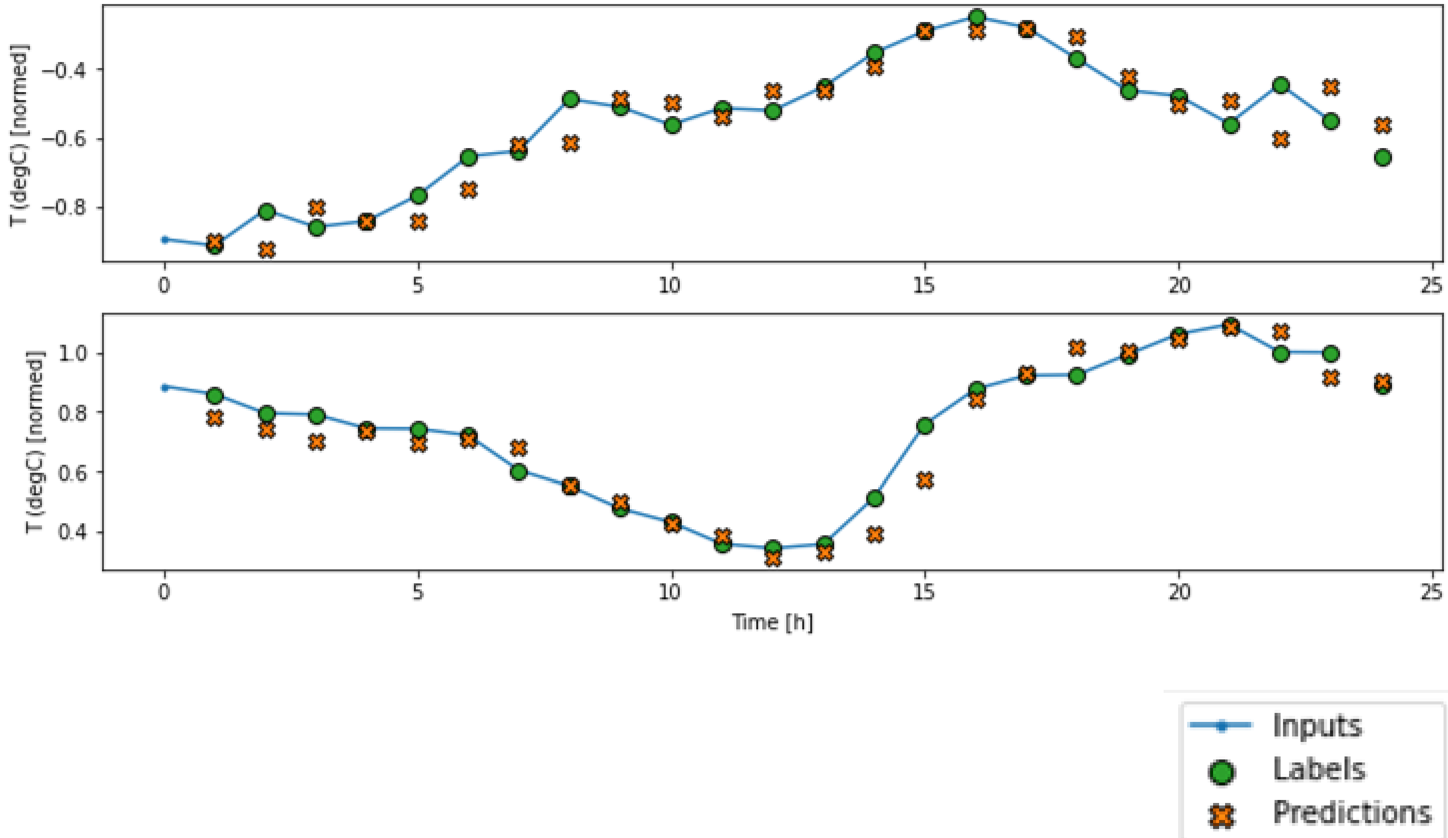


Image source: https://www.tensorflow.org/tutorials/structured_data/time_series

Example: weather prediction

Autoregressive prediction: After initialising the model, we do one prediction per step and feed the output back to the model

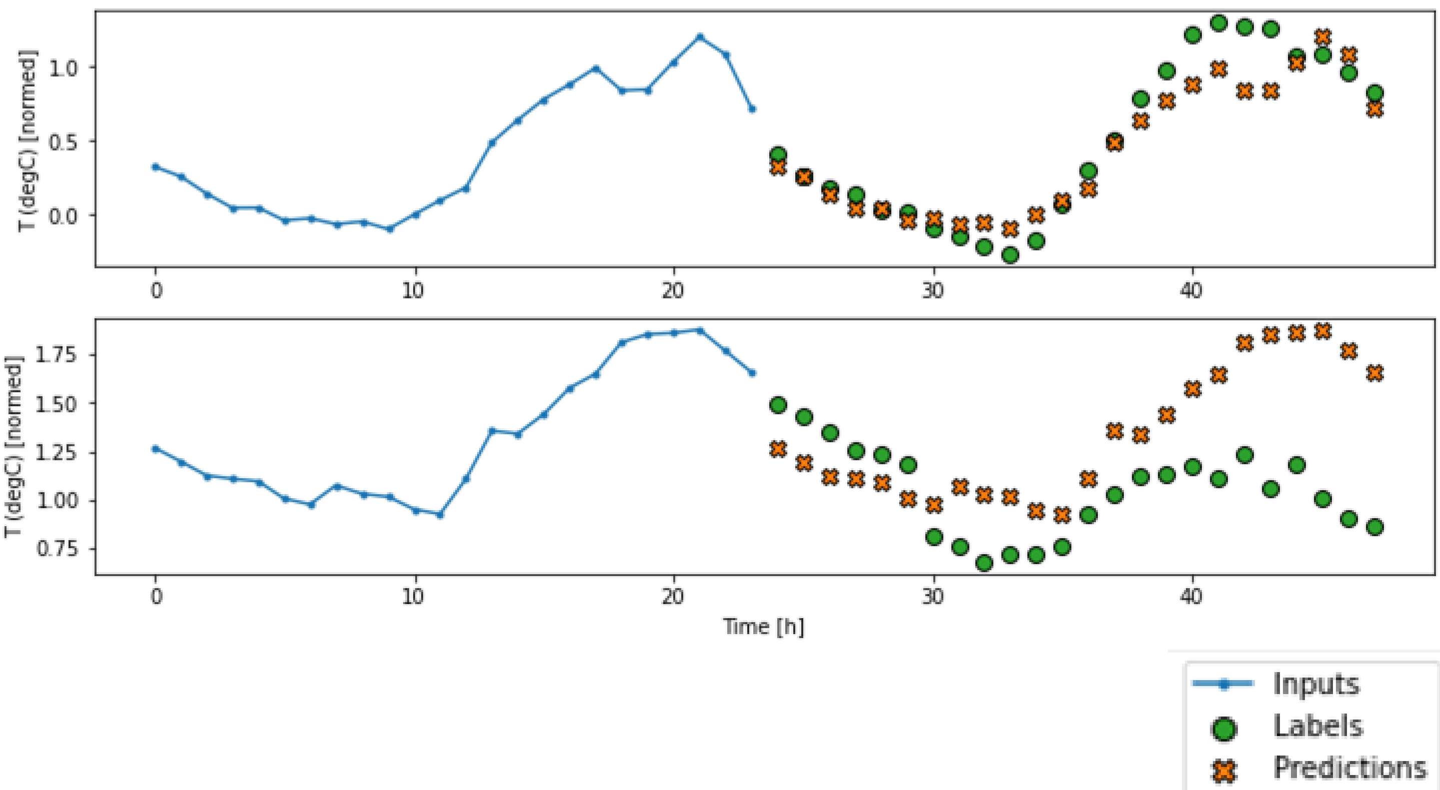
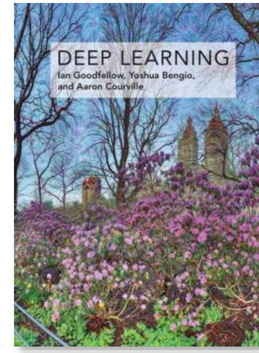
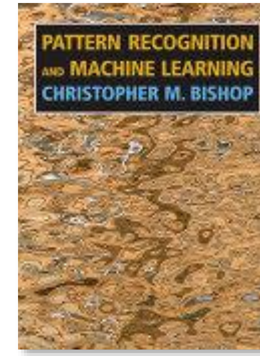


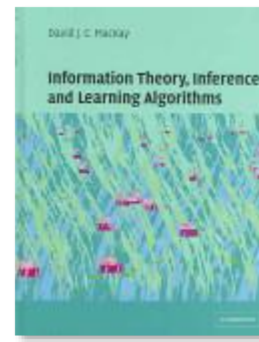
Image source: https://www.tensorflow.org/tutorials/structured_data/time_series

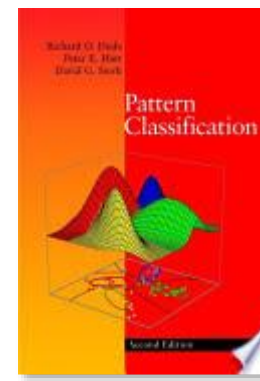
Resources

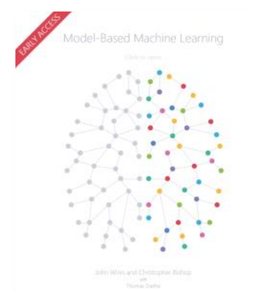
- I. Goodfellow, Y. Bengio, A. Courville, *"Deep Learning"*, MIT Press, 2016
  http://www.deeplearningbook.org/
- C. Bishop, *"Pattern Recognition and Machine Learning"*, Springer, 2006
  http://research.microsoft.com/en-us/um/people/cmbishop/prml/index.htm
- D. MacKay, *"Information Theory, Inference and Learning Algorithms"*, Cambridge University Press, 2003
  http://www.inference.phy.cam.ac.uk/mackay/
- R.O. Duda, P.E. Hart, D.G. Stork, *"Pattern Classification"*, Wiley & Sons, 2000
  http://books.google.com/books/about/Pattern_Classification.html?id=Br33IRC3PkQC
- J. Winn, C. Bishop, *"Model-Based Machine Learning"*, early access
  http://mbmlbook.com/
- A. Zhang, Z.C. Lipton, M. Li, A.J. Smola, *"Dive into Deep Learning"*, 2021
  https://d2l.ai/

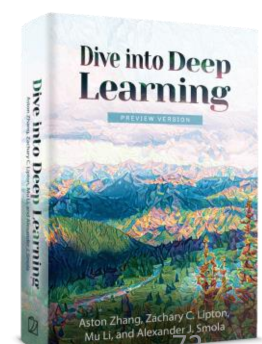












Further Info

- Many of the slides of these lectures have been adapted from various highly recommended online lectures and courses:
  - Andrew Ng's *Machine Learning Course*, Coursera
    https://www.coursera.org/course/ml
  - Andrew Ng's *Deep Learning Specialization,* Coursera
    https://www.coursera.org/specializations/deep-learning
  - Victor Lavrenko's *Machine Learning* Course
    https://www.youtube.com/channel/UCs7alOMRnxhzfKAJ4JjZ7Wg
  - Fei Fei Li and Andrej Karpathy's *Convolutional Neural Networks for Visual Recognition*
    http://cs231n.stanford.edu/
  - Geoff Hinton's *Neural Networks for Machine Learning, (*ex Coursera)
    https://www.youtube.com/playlist?list=PLiPvV5TNogxKKwvKb1RKwkq2hm7ZvpHz0
  - Luis Serrano's introductory videos
    https://www.youtube.com/channel/UCgBncpylJ1kiVaPyP-PZauQ
  - Michael Nielsen's *Neural Networks and Deep Learning*
    http://neuralnetworksanddeeplearning.com/
  - David Charte et al. A practical tutorial on autoencoders for nonlinear feature fusion:Taxonomy, models, software and guidelines
    https://arxiv.org/abs/1801.01586