Time Series and Sequential Data

Volker Tresp

Summer 2019

Modelling of Time Series

- In some applications observations have a sequential order
- Observation $\{\mathbf{x}_t, \mathbf{y}_t\}_t$ have a sequential label t
- In some applications, in testing only the inputs $\mathbf{x}_t$ are measured, but past inputs $..., \mathbf{x}_{t-2}, \mathbf{x}_{t-1}$ or later inputs $\mathbf{x}_{t+1}, \mathbf{x}_{t+2}, ...$ might be relevant for predicting $\mathbf{y}_t$; a typical example is the labelling of words in a sentence
- In other applications, past outputs $..., \mathbf{y}_{t-2}, \mathbf{y}_{t-1}$ are available as well; this is the case in time series prediction; there are also applications in language modelling
- We start with the latter case
- The next figure shows a time series (DAX)
- Other interesting time-series: energy prize, energy consumption, gas consumption, copper prize, ...
- In a real application a number of preprocessing and normalization steps are perfroems (removal of linear or periodic trends, ...), which we do not discuss here

DAX Performance-Index

01.06.07 17:45 Uhr • 7.987,85 +1,33 % [+104,81]	Enthaltene Werte: **30** Tages-Vol.: **9,12 Mrd.**	Typ: **Index** Börse: XETRA

ARX Models / TDNN

Neural Networks for Time-Series Modelling

- Let $y_t, t = 1, 2, \ldots$ be the time-discrete time-series of interest (example: DAX)
- Let $x_t, t = 1, 2, \ldots$ denote a second time-series, that contains information on y_t (Example: Dow Jones)
- For simplicity, we assume that both y_t and x_t are scalar. The goal is the prediction of the next value of the time-series
- We assume a system of the form

$$y_t = f(y_{t-1}, \ldots, y_{t-T}, x_{t-1}, \ldots, x_{t-T}) + \epsilon_t$$

with i.i.d. random numbers $\epsilon_t, t = 1, 2, \ldots$ which model unknown disturbances.

Neural Networks for Time-Series Modelling (cont'd)

- We approximate, using a neural network,

$$f(y_{t-1}, \ldots, y_{t-T}, x_{t-1}, \ldots, x_{t-T})$$

$$\approx f_{\mathbf{w},V}(y_{t-1}, \ldots, y_{t-T}, x_{t-1}, \ldots, x_{t-T})$$

  and obtain the cost function

$$\text{cost}(\mathbf{w}, V) = \sum_{t=1}^{N} (y_t - f_{\mathbf{w},V}(y_{t-1}, \ldots, y_{t-T}, x_{t-1}, \ldots, x_{t-T}))^2$$

- The neural network can be trained as before with simple back propagation *if in training all y_t and all x_t are known*!

- This is a NARX model: **N**onlinear **A**uto **R**egressive Model with e**x**ternal inputs. Another name: TDNN (time-delay neural network).

- Note the "convolutional" idea in TDNNs: the same neural network is applied in all time instances

Prediction

- For single step prediction, we use

$$\widehat{y}_t = f(y_{t-1}, \ldots, y_{t-T}, x_{t-1}, \ldots, x_{t-T})$$

Mutiple-Step Prediction based on Single-Step Prediction

- Predicting multiple time steps in the future is not trivial for nonlinear models
- Consider the multivariate model

$$(y_t, x_t)^T = \mathbf{f_w}(y_{t-1}, \ldots, y_{t-T}, x_{t-1}, \ldots, x_{t-T}) + \epsilon_t$$

  where we model the progression of both time series models and where $\epsilon_t \sim \mathcal{N}(0, \Sigma)$
- In training we can learn $\mathbf{f_w}(\cdot)$ (now with two outputs) and Σ
- For multiple-step prediction, we can simulate (i.e., sample) the joint (y_t, x_t) for the desired number of time steps in the future (Monte-Carlo simulation) and can derive estimated means and covariances

Mutiple-Step Prediction based on Multiple Step Prediction

- Of course it is also possible to directly predict the value of the time series τ steps in the future

$$\widehat{y}_{t+\tau} = f(y_{t-1}, \ldots, y_{t-T}, x_{t-1}, \ldots, x_{t-T})$$

- This is done in system simulation: the prediction based on detailed system models might be computationally very expensive and cannot be done online; the idea is to train a neural network predictive model off-line and then use that one online instead of an expensive simulation

Representation Models and Language Models

Language Model

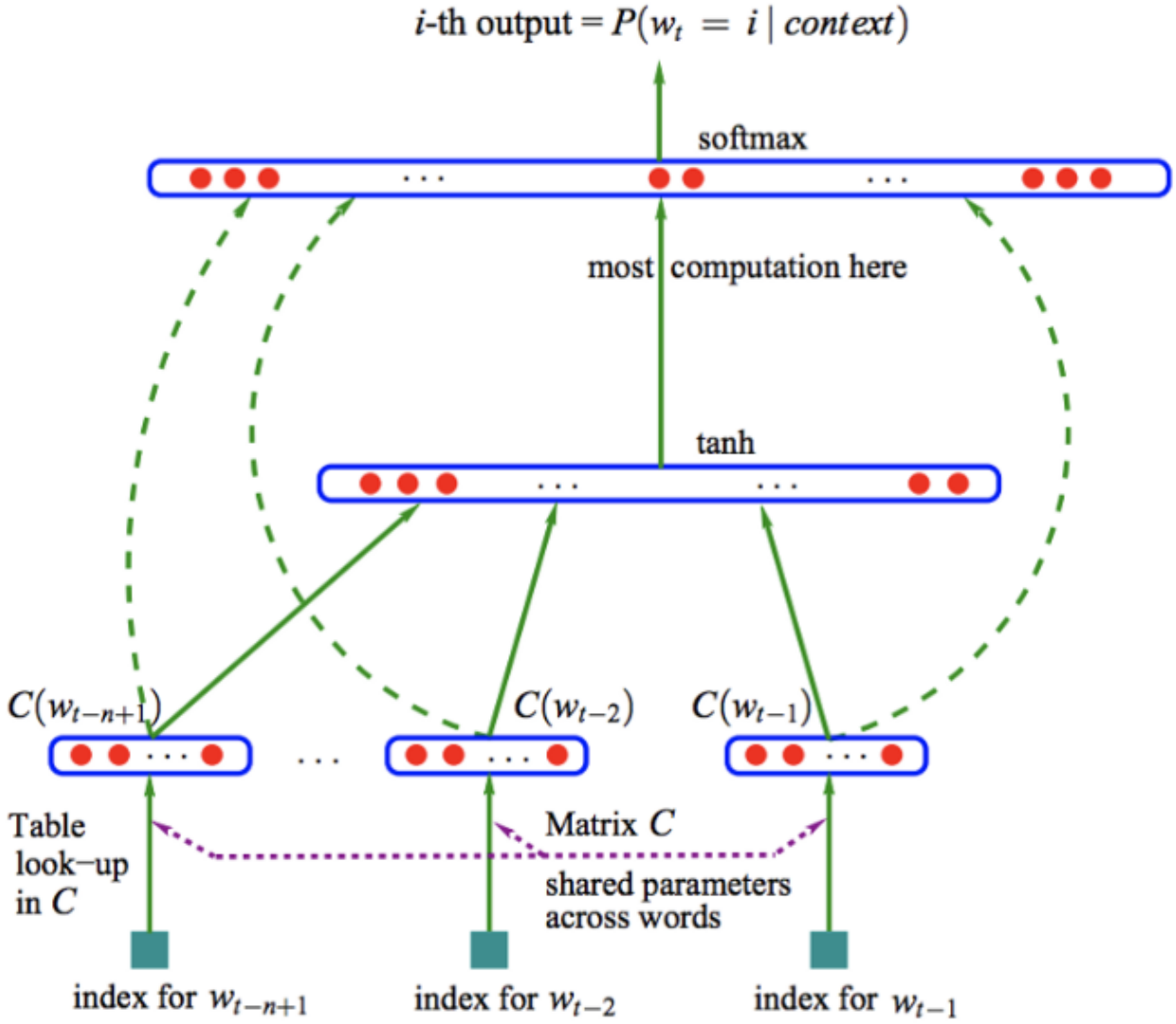
- A similar idea has been used in language modeling
- The ideas is to predict the next word (out off a vocabulary of N_w words) in a text, based on the last T words
- Consider we want to predict y_t: y_t has as N_w components, one for each word
- The inputs to the models are past words; the model assumption is that a word i is associated with an embedding vector $\mathbf{a}_i$ of dimension r
- Thus in a first step, a one-hot encoding word i is mapped to the embedding vector of word $\mathbf{a}_i$ which is then the input to a neural network
- We get

$$P(y_t = k|y_{t-1}, \ldots, y_{t-T}) = \mathit{softmax}_k \left(f_{\mathbf{w}}(\mathbf{a}_{i(t-1)}, \ldots, \mathbf{a}_{i(t-T)}) \right)$$

where $i(t-m)$ is the index of the word at position $t-m$ and where $f_w(\cdot)$ is a neural network with one hidden layer

Embeddings

- Training of the *word embeddings* and *the neural network parameters* can be done self-supervised on a huge corpus (without human labelling)
- After training, one obtains latent word representations (word embeddings) which are published and can be used in other applications
- State of the art are embeddings derived from language models like: ELMo, BERT, Word2vec, and GloVe
- The embedding idea is extremely powerful and one of the corner stones of modern machine learning
- In the next figure, the word embedding matrix is denoted as C


i-th output = $P(w_t = i \mid context)$
softmax
most computation here
tanh
$C(w_{t-n+1})$
$C(w_{t-2})$
$C(w_{t-1})$
Table look−up in C
Matrix C
shared parameters across words
index for w_{t-n+1}
index for w_{t-2}
index for w_{t-1}

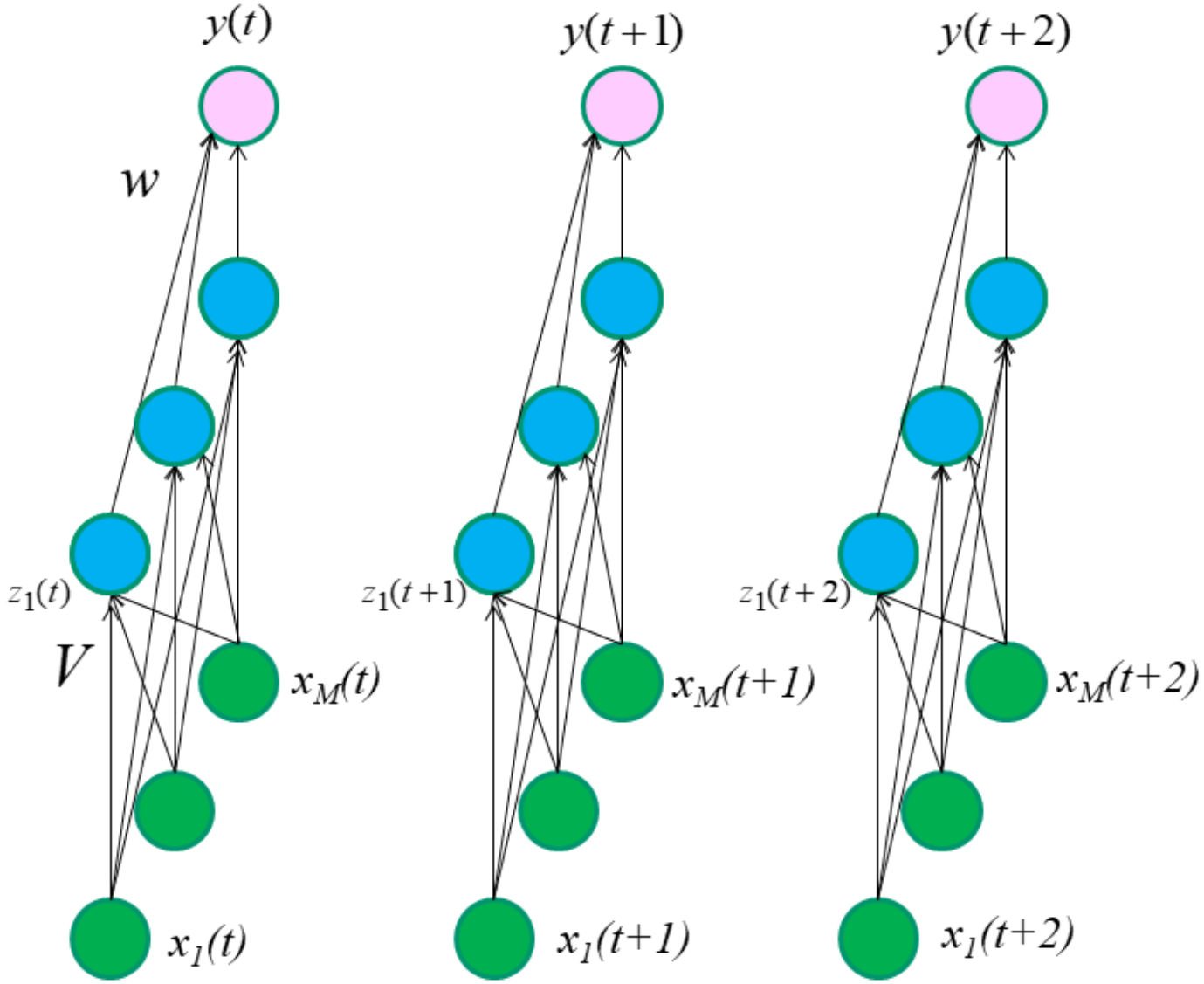
Recurrent Neural Networks

Recurrent Neural Network

- Recurrent neural networks (RNNs) are powerful methods for sequence modelling
- In their simplest form they are used to improve an output prediction by providing a memory for previous inputs

A Feedforward Neural Network with a Time Index

- We start with a normal feedforward neural network where the pattern is a sequential index t



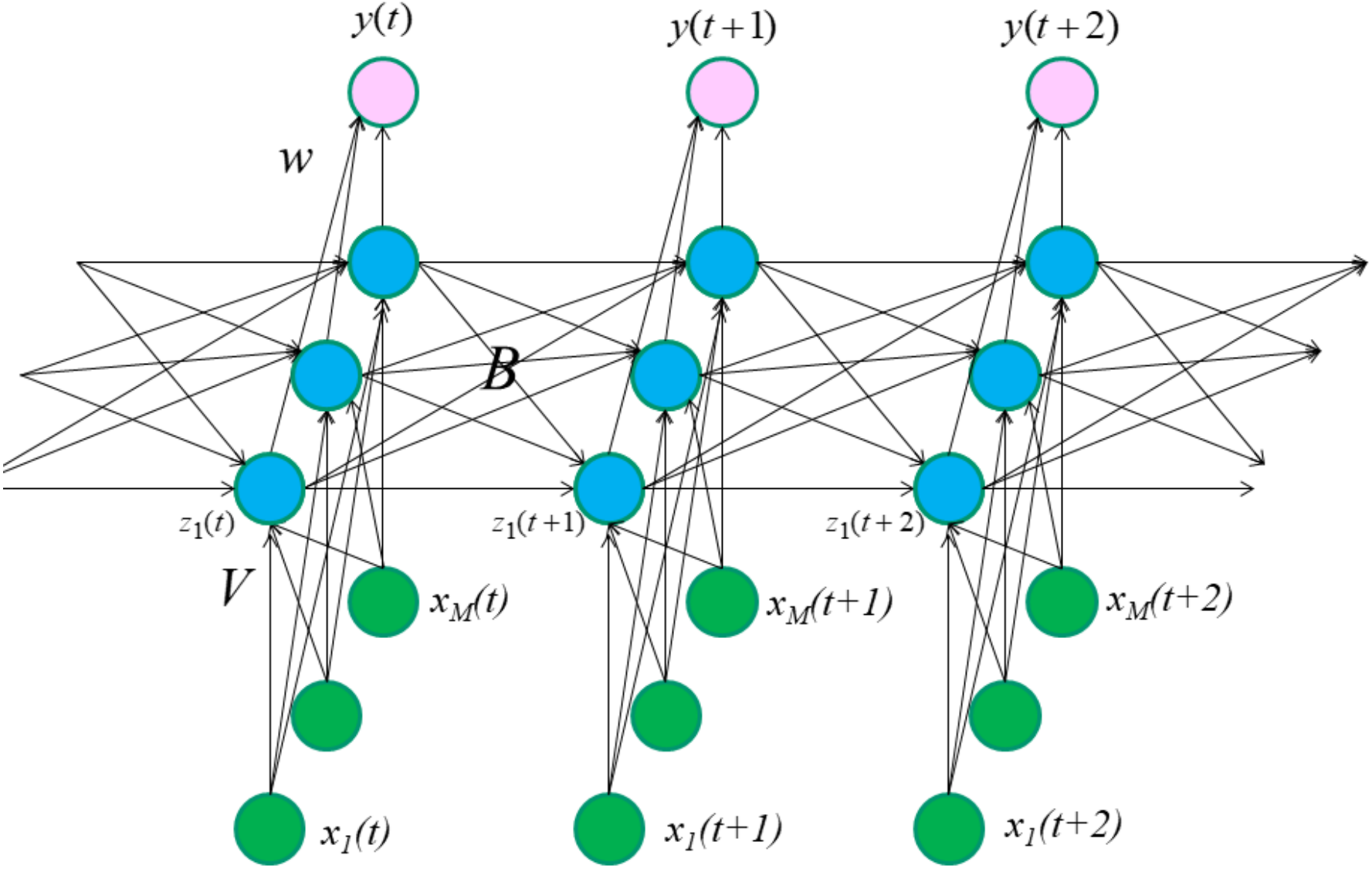
feed forward neural network at time step t

feed forward neural network at time step t+1

feed forward neural network at time step t+2

A Recurrent Neural Network Architecture unfolded in Time

- The hidden layer now also receives input from the hidden layer of the previous time step
- The hidden layer now has a memory function reflecting hidden inputs
- Thus a Recurrent Neural Network (RNN) is a nonlinear state-space model



Recurrent neural network, unfolded in time

A Recurrent Neural Network Architecture unfolded in Time (cont'd)

- In a compact notation, we write,

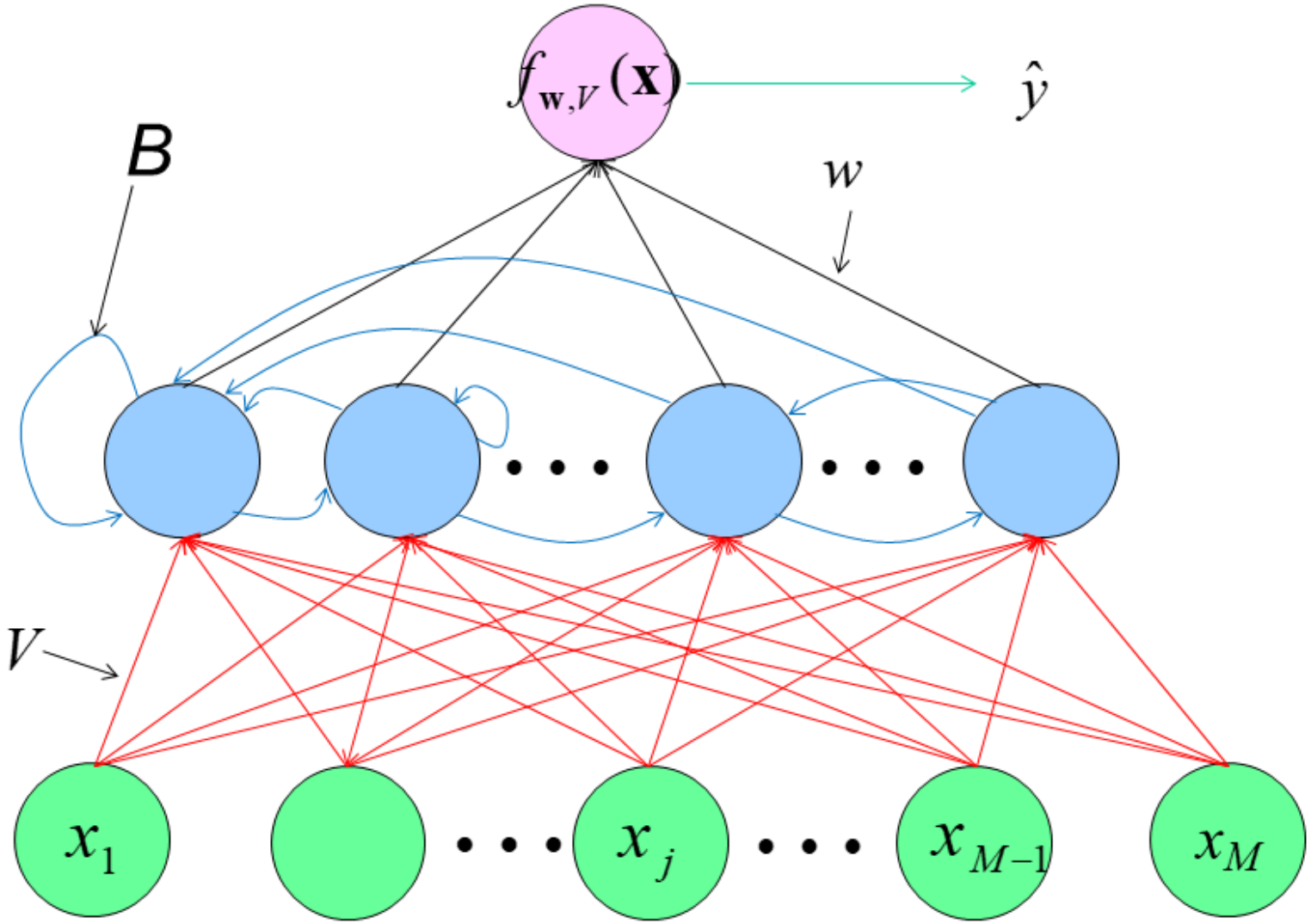
$$\mathbf{z}_t = \mathit{sig}(B\mathbf{z}_{t-1} + V\mathbf{x}_t)$$

$$\mathbf{y}_t = \mathit{sig}(W\mathbf{z}_t)$$

where we permit several outputs; also, in the last layer we might replace the *sig* with the *softmax*

Recurrent Representation

- The next slide shows an RNN as a recurrent structure



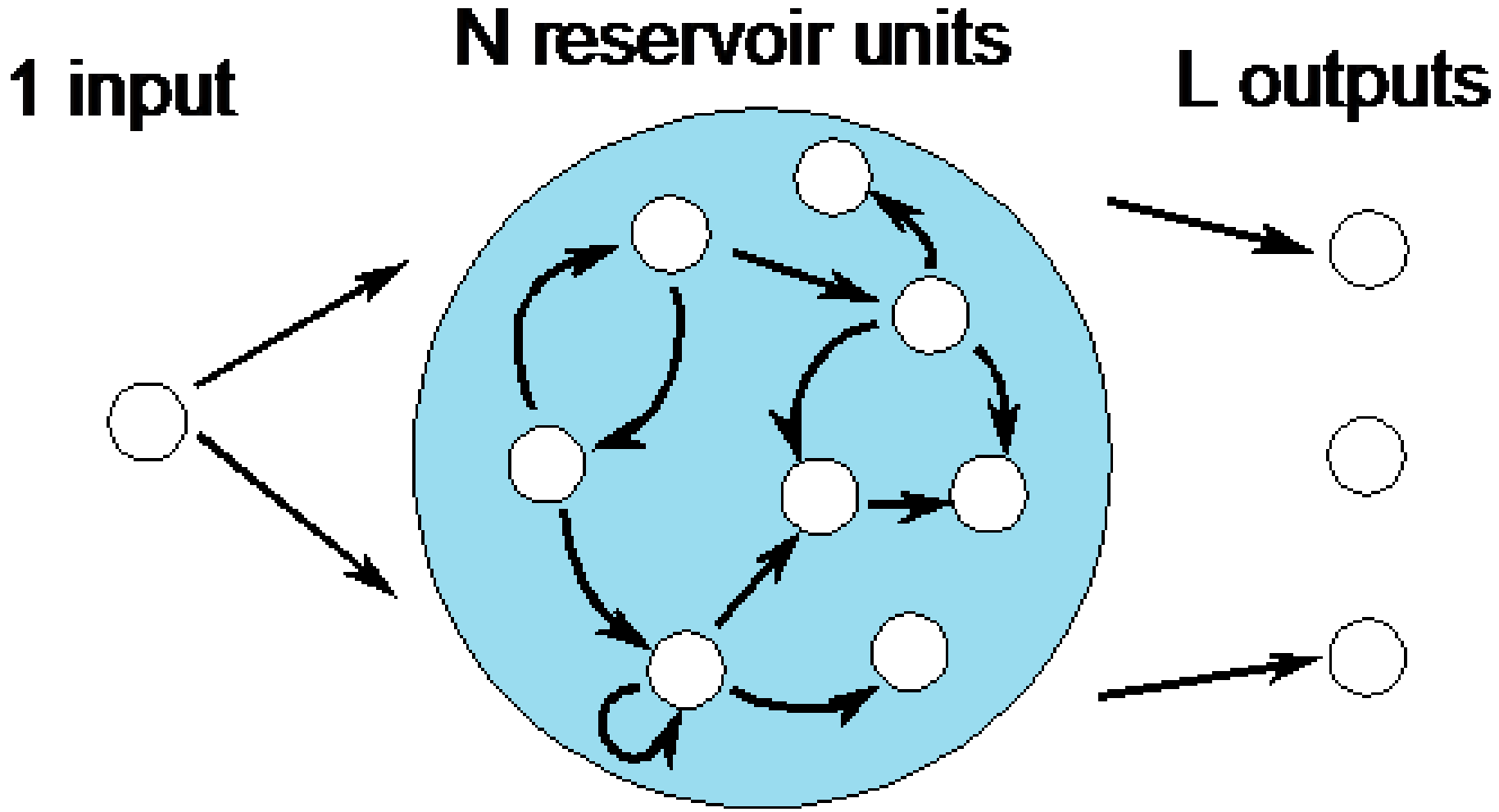
recurrent neural network showing recurrent connections

Backpropagation through time (BPTT)

- Training can be performed using backpropagation through time (BPTT), which is an application of backpropagation (SGD) to the unfolded network structure
- As an additional complexity, the error which occurs to the outputs at time t is not only backpropagated to the previous layers at time t, but also backward in time to all previous neural networks
- In principle, one would propagate back to $t = 1$; in practice, one typically truncates the gradient calculation

Echo-State Network

- Recurrent Neural Networks are sometimes difficult to train
- A simple alternative is to initialize B and V randomly (according to some recipe) and only train W
- W can be trained with the simple learning rules for linear regression or classification
- This works surprisingly well and is done in the Echo-State Network (ESN)
- ESN (and also liquid-state machines) are examples of so called *reservoir computing*


1 input
N reservoir units
L outputs

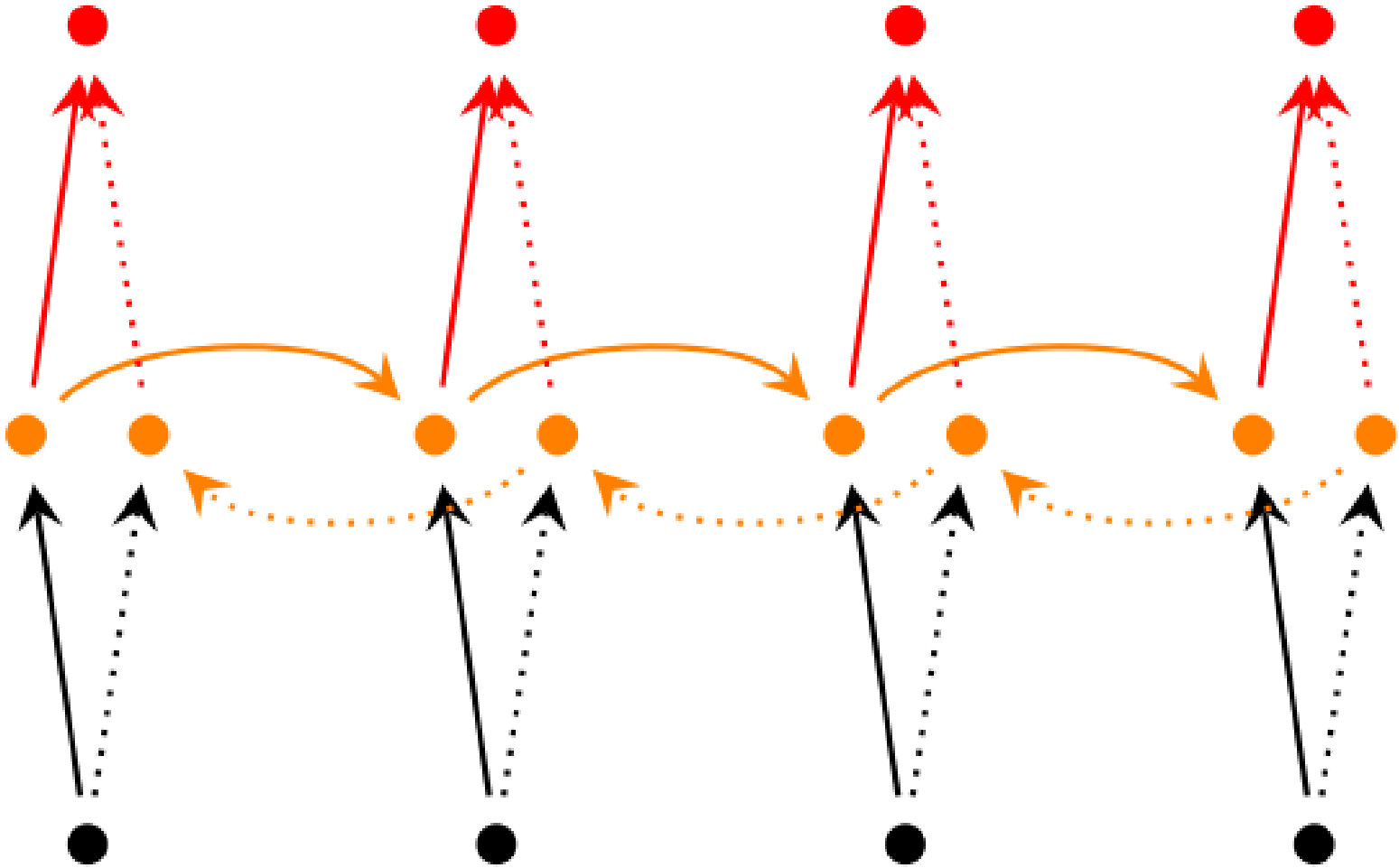
Issues in Prediction

- An RNN is typically used as predictive model in an iterative setting
- Due to the deterministic nature of the model: if the output $\mathbf{y}_t$ is predicted and then becomes available, it will not affect future predictions, since there is no information flowing back from $\mathbf{y}_t$ to $\mathbf{z}_t$
- This is in contrast to some probabilistic models such as hidden Markov models (HMMs), Kalman filters, stochastic state space models

Bidirectional RNNs

- The predictions in bidirectional RNNs depend on past and future inputs
- Useful for sequence labelling problems: handwriting recognition, speech recognition, bioinformatics, ...
- Bidirectional recurrent

$$\mathbf{z}_t = [\mathbf{z}_t^f ; \mathbf{z}_t^b] = \left[sig\left(V^f \mathbf{x}_t + B^f \mathbf{z}_{t-1}^f\right) ; sig\left(V^b \mathbf{x}_t + B^b \mathbf{z}_{t+1}^b\right)\right]$$



LSTMS

Issues in Prediction

- Although the RNN has a memory, it has difficulties remembering important information far in the past
- This can be attributed to the vanishing gradient problem
- Solutions are the long short-term memory (LSTM), and the gated recurrent units (GRUs)
- We now discuss the LSTM

We Start with a Feedforward Neural Network

- Consider a feedforward neural network

$$\mathbf{s}_t = \mathit{sig}(V\mathbf{x}_t) \quad \mathbf{z}_t = \tanh(\mathbf{s}_t)$$

$$\hat{\mathbf{y}}_t = \mathit{sig}(W\mathbf{z}_t)$$

- The transfer function of the hidden neuron is a bit strange, $\tanh(\mathit{sig}(V\mathbf{x}_t))$
- $\mathbf{s}_t$ is called the **cell state vector**, $\mathbf{z}_t$ is the **output vector** (of the units, not the neural network)
- In the following steps, each latent unit will become an LSTM unit; thus we will have H LSTM units in the network

We Enter Input and Output Gates

- We now use input and output gates which can turn on and off individual LSTM units
- With input gate vector $\mathbf{g}_t$ and output gate vector $\mathbf{q}_t$

$$\mathbf{s}_t = \mathbf{g}_t \circ \mathit{sig}(V\mathbf{x}_t) \qquad \mathbf{z}_t = \mathbf{q}_t \circ \tanh(\mathbf{s}_t)$$

  Here, $\circ$ is the elementwise (Hadamard) product. As before,

$$\hat{\mathbf{y}}_t = \mathit{sig}(W\mathbf{z}_t)$$

- Input gates and output gates are also functions of the inputs

$$\mathbf{g}_t = \mathit{sig}(V^g\mathbf{x}_t) \qquad \mathbf{q}_t = \mathit{sig}(V^q\mathbf{x}_t)$$

- Gates are commonly used in mixture of expert neural networks, if the function switches between modes of operations

With Feedback

- We add recurrent connections to the cell state vector and the gates

$$\mathbf{s}_t = \mathbf{g}_t \circ \mathit{sig}(V\mathbf{x}_t + B\mathbf{z}_{t-1}) \qquad \mathbf{z}_t = \mathbf{q}_t \circ \tanh(\mathbf{s}_t)$$

- Input Gate

$$\mathbf{g}_t = \mathit{sig}(V^g\mathbf{x}_t + B^g\mathbf{z}_{t-1})$$

- Output Gate

$$\mathbf{q}_t = \mathit{sig}(V^q\mathbf{x}_t + B^q\mathbf{z}_{t-1})$$

Cell State Vector with Self-recurrency and Forget Gate

- We add self-recurrency to the cell state vector, including a forget gate

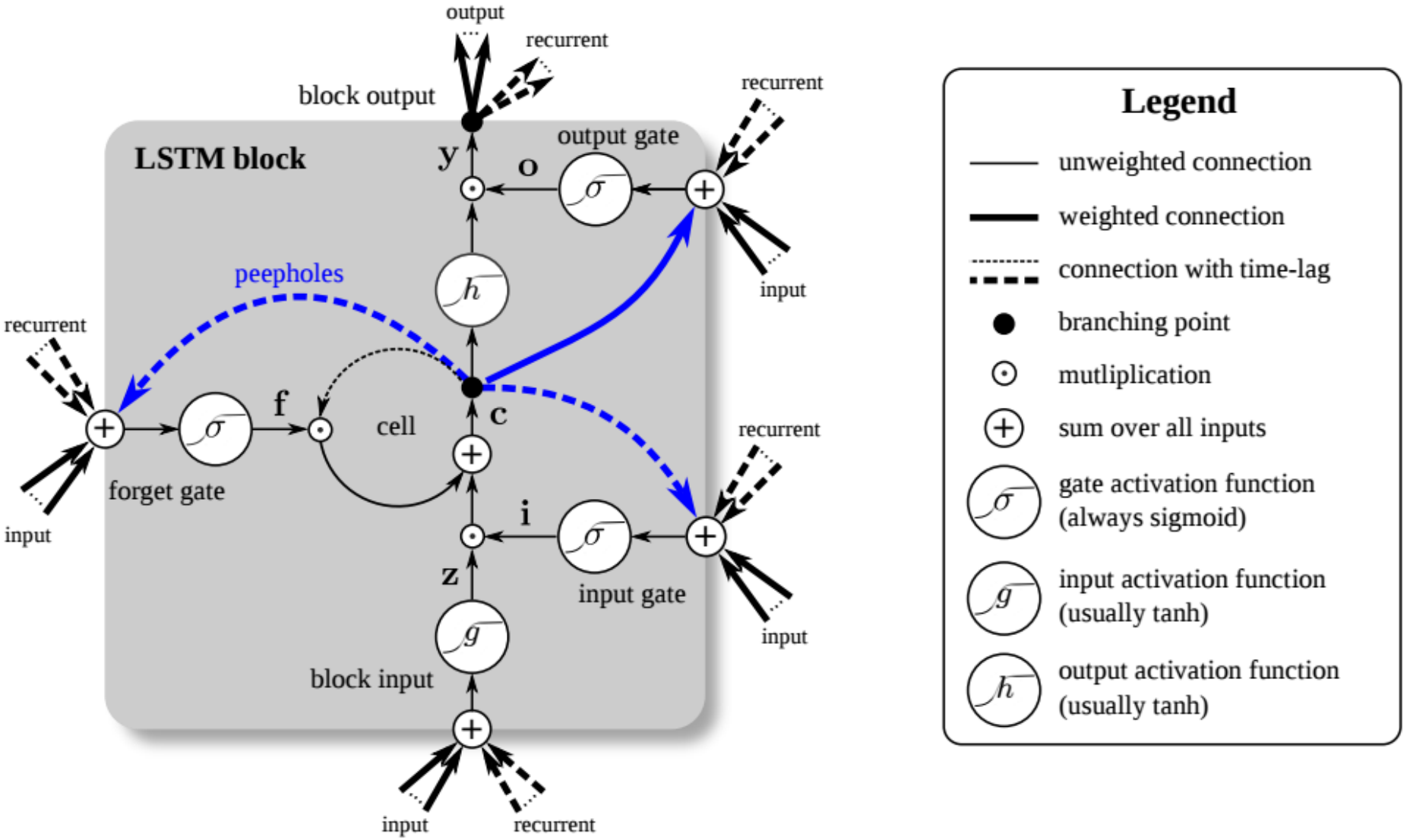
$$\mathbf{s}_t = \mathbf{f}_t \circ \mathbf{s}_{t-1} + \mathbf{g}_t \circ \mathit{sig}(V\mathbf{x}_t + B\mathbf{z}_{t-1})$$

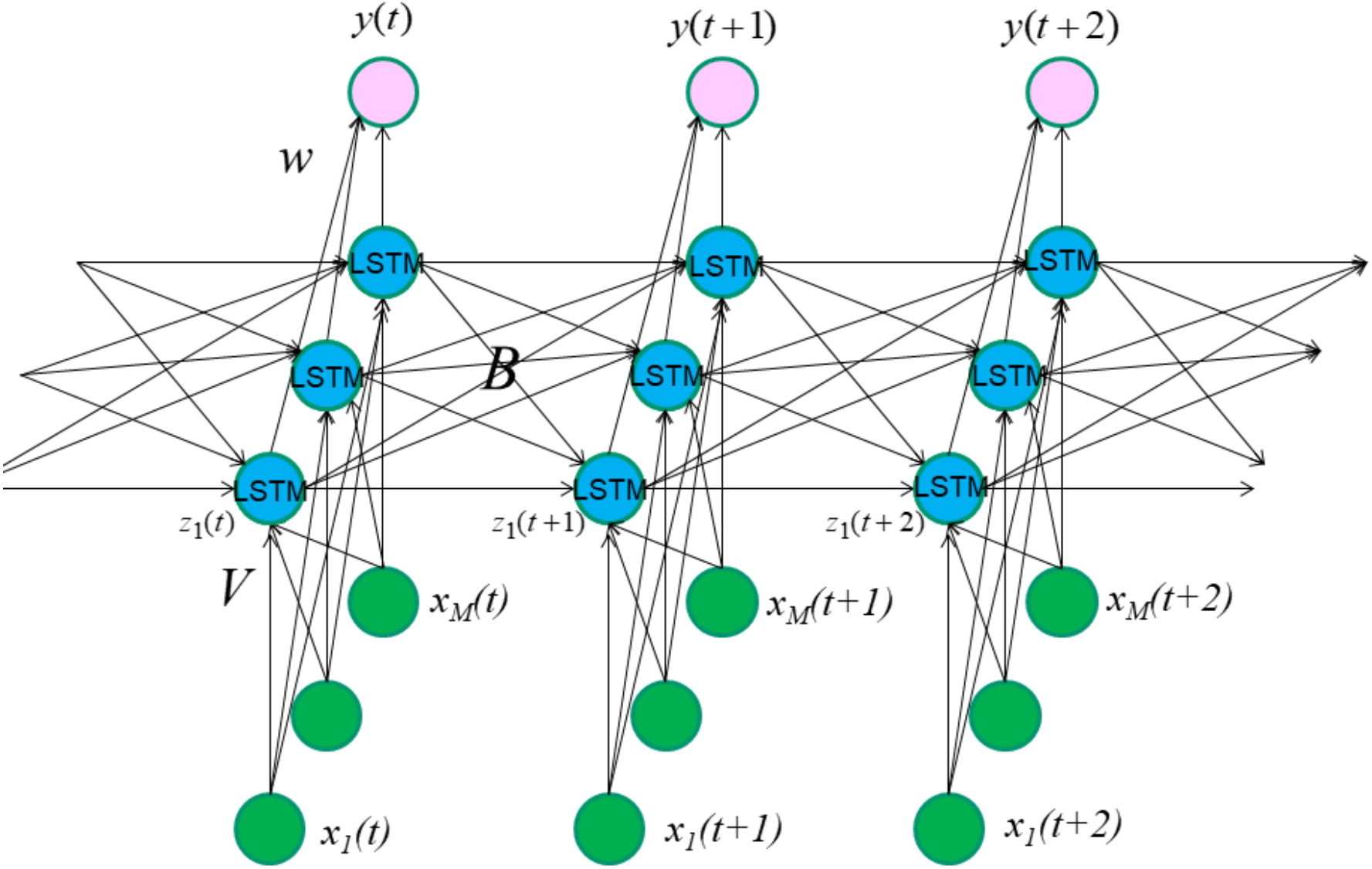
- Forget gate

$$\mathbf{f}_t = \mathit{sig}(V^f\mathbf{x}_t + B^f\mathbf{z}_{t-1})$$

Long Short Term Memory (LSTM)

- As a recurrent structure the Long Short Term Memory (LSTM) approach has been very successful
- Basic idea: at time t a newspaper announces that the Siemens stock is labelled as "buy". This information will influence the development of the stock in the next days. A standard RNN will not remember this information for very long. One solution is to define an extra input to represent that fact and that is on as along as "buy" is valid. But this is handcrafted and does not exploit the flexibility of the RNN. A flexible construct which can hold the information is a long short term memory (LSTM) block.
- The LSTM was used very successful for reading handwritten text and is the basis for many applications involving sequential data (NLP, machine translation, ...)
- For the rest of the network, an LSTM node looks like a regular hidden node


output
recurrent
block output
recurrent
output gate
LSTM block
y
o
peepholes
input
recurrent
h
recurrent
f
cell
c
forget gate
input
recurrent
i
input gate
z
input
g
block input
input
recurrent
Legend
unweighted connection
weighted connection
connection with time-lag
branching point
mutliplication
sum over all inputs
gate activation function (always sigmoid)
input activation function (usually tanh)
output activation function (usually tanh)



Recurrent neural network, unfolded in time

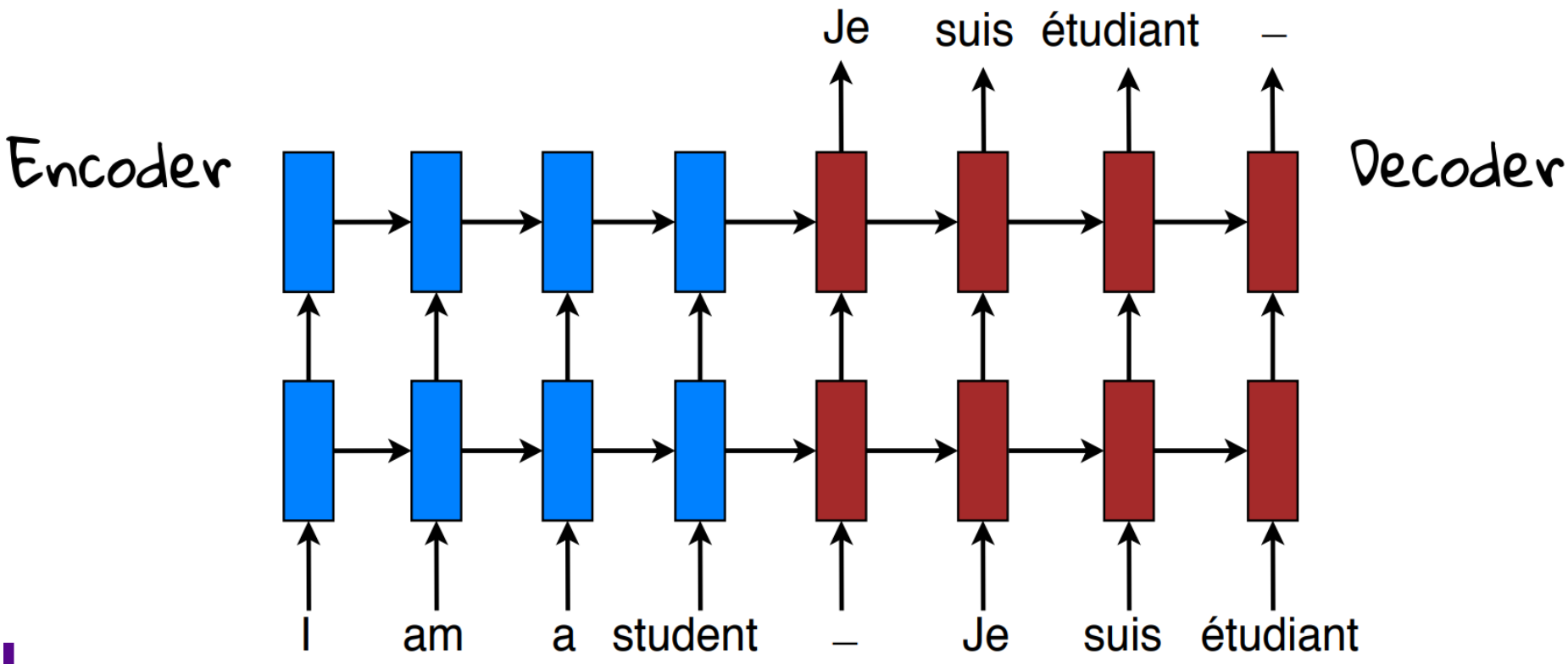
LSTM Applications

- Wiki: LSTM achieved the best known results in unsegmented connected handwriting recognition, and in 2009 won the ICDAR handwriting competition. LSTM networks have also been used for automatic speech recognition, and were a major component of a network that in 2013 achieved a record 17.7% phoneme error rate on the classic TIMIT natural speech dataset
- Applications: Robot control, Time series prediction, Speech recognition, Rhythm learning, Music composition, Grammar learning, Handwriting recognition, Human action recognition, Protein Homology Detection

Encoder-Decoder Networks for Machine Translation

Encoder Decoder Architecture

- Most machine translation systems rely on the encoder-decoder approach
- Neural Machine Translation (NMT)
- Typical numbers: embedding rank: $r = 1000$, and 1000 hidden units per layer


Encoder
Decoder
Je
suis
étudiant
–
I
am
a
student
–
Je
suis
étudiant

Encoder

- An **encoder** is an RNN (often an LSTM) with no output layer (no $\mathbf{y}_t$), but maybe several layers of recurrent units; as in the language model, *the inputs are latent embeddings of the words*
- The **encoder vectors** are the last hidden states (end-of-sentence)

Decoder

- The initial latent state of the decoder are the encoder vectors
- In its simplest form, the latent state of the decoder evolves as

$$\mathbf{z}_t = \mathit{sig}(B\mathbf{z}_{t-1} + V\mathbf{a}_{\mathbf{y}_{t-1}})$$

$$\mathbf{y}_t = \mathit{sig}(W\mathbf{z}_t)$$

- In training the input to the decoder is the *embedding of the previous word*; the output is the one-hot encoding of the current word
- Training is based on bilingual, parallel corpora; each hidden layer might consist of 1000 hidden units
- In testing one finds the most likely decoded sequence of words (e.g., using beam search)
- Often one uses two or more hidden layers of LSTM units

Encoder-Decoder Approach in NMT

- Neural Machine Translation (NMT) achieved state-of-the-art performances in large-scale translation tasks such as from English to French
- NMT has the ability to generalize well to very long word sequences.
- The model does not have to explicitly store gigantic phrase tables and language models as in the case of standard MT; hence, NMT has a small memory footprint.
- Implementing NMT decoders is easy unlike the highly intricate decoders in standard MT

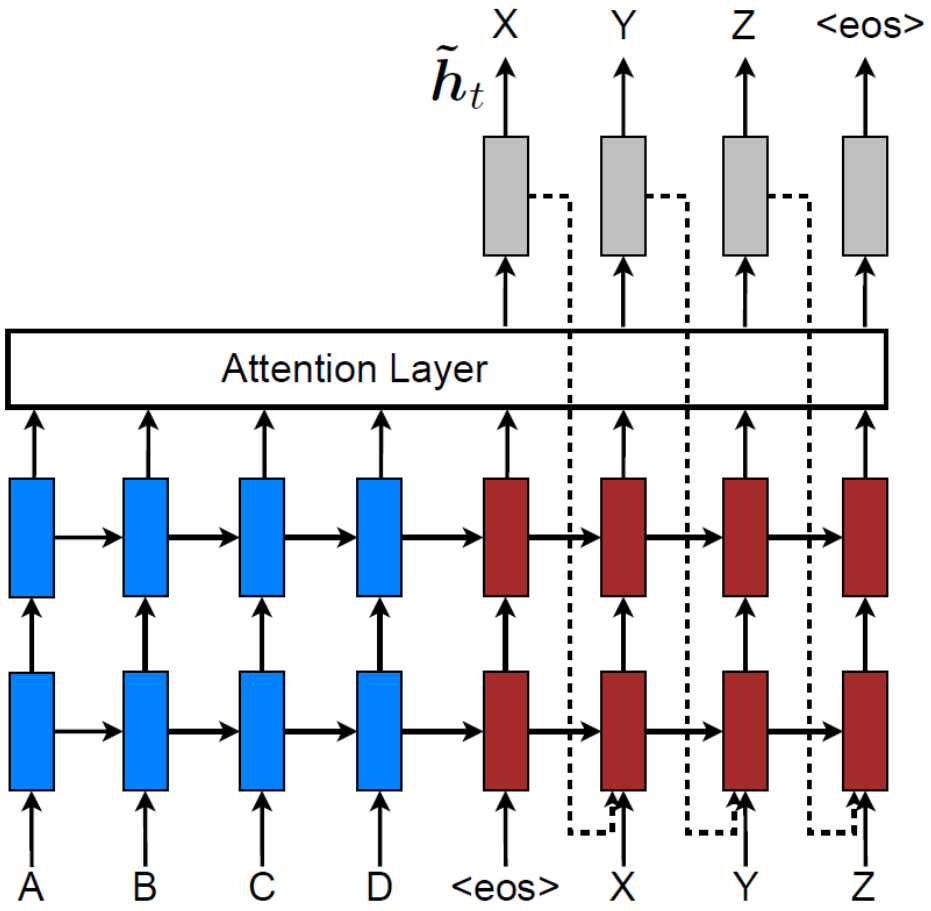
Attention

Introduction

- The concept of "attention" has gained popularity recently in training neural networks, allowing models to learn alignments between different modalities, e.g., between image objects and agent actions in the dynamic control problem, between speech frames and text in the speech recognition task, or between visual features of a picture and its text description in the image caption generation task
- Attention has successfully been applied to jointly translate and align words
- Attention-based NMT models are superior to non attentional ones in many cases, for example in translating names and handling long sentences
- We follow: Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. 2016. "Effective Approaches to Attention-based Neural Machine Translation"
- First work: D. Bahdanau, K. Cho, and Y. Bengio. 2015. "Neural machine translation by jointly learning to align and translate." In ICLR

Overall Architecture

- The next figure shows the overall architecture
- The attention layer sits on top of the normal encoder-decoder network
- Based on the neural activations in the encoder-decoder, it calculates new activations (grey boxes)
- Notation: $\mathbf{h}$ in the figures is our $\mathbf{z}$


X
Y
Z
<eos>
$\tilde{h}_t$
Attention Layer
A
B
C
D
<eos>
X
Y
Z

Attention

- Let $\mathbf{z}_t$ be a target hidden state vector of interest in the **decoder**
- Let $\mathbf{c}_t$ be the source-side **context vector** (derived further down)
- The attentional hidden state is

$$\tilde{\mathbf{z}}_t = \mathit{sig}\left(V\mathbf{z}_t + D\mathbf{c}_t\right)$$

- The *sig* is typically the tanh; note that this is a normal layer in a neural network where the layer $\mathbf{z}_t$ is the lower layer and $\tilde{\mathbf{z}}_t$ is the upper layer and where the lower layer is appended with $\mathbf{c}_t$
- The decoded word probability is then calculated as $\mathit{softmax}(W_s\tilde{\mathbf{z}}_t)$

Global Attention

- Let $\bar{\mathbf{z}}_s$ be any activation vector in the encoder (source hidden state) (often restricted to the top layer)
- The **alignment** of s for t is a scalar,

$$a_t(s) = \mathit{align}(\mathbf{z}_t, \bar{\mathbf{z}}_s) = \frac{\exp(\mathit{score}(\mathbf{z}_t, \bar{\mathbf{z}}_s))}{\sum_{s'} \exp(\mathit{score}(\mathbf{z}_t, \bar{\mathbf{z}}_{s'}))}$$

- The alignment score function calculates a similarity measure: A typical **score** is the dot product, $score(\mathbf{z}_t, \bar{\mathbf{z}}_s) = \mathbf{z}_t^T \bar{\mathbf{z}}_s$
- A **context vector** is then calculated as

$$\mathbf{c}_t = \sum_s a_t(s) \bar{\mathbf{z}}_s$$

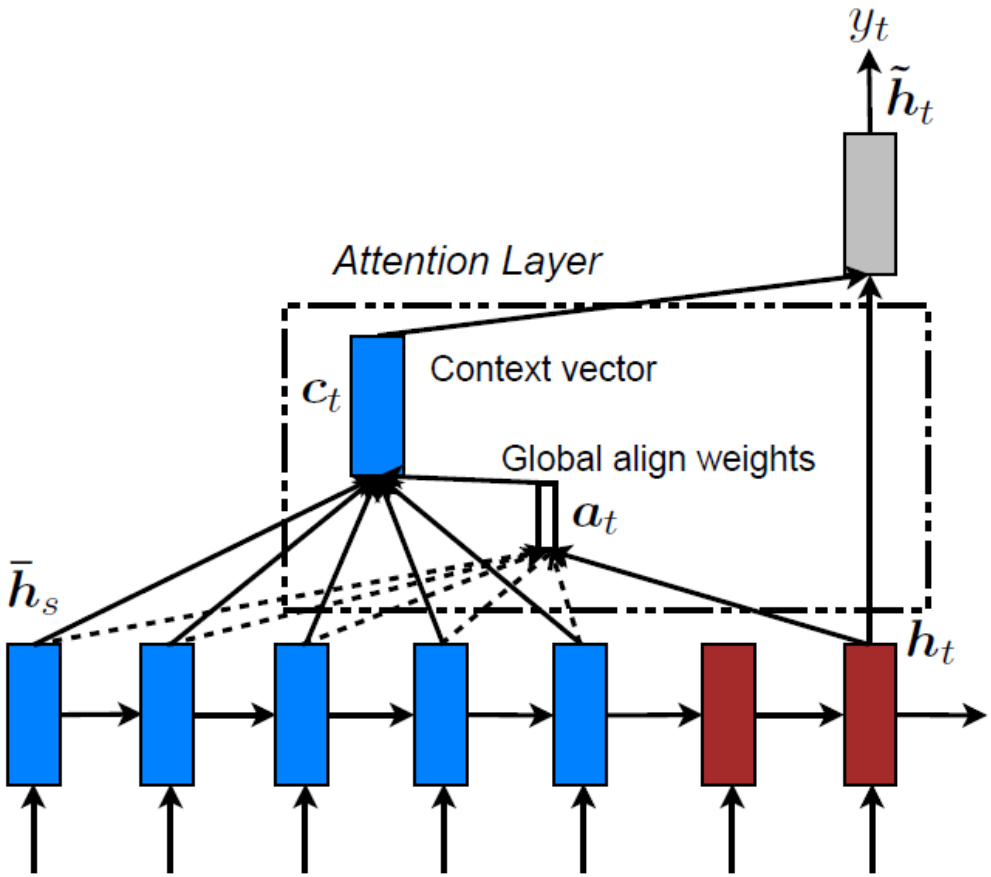


Figure 2: **Global attentional model** – at each time step t, the model infers a *variable-length* alignment weight vector $\boldsymbol{a}_t$ based on the current target state $\boldsymbol{h}_t$ and all source states $\bar{\boldsymbol{h}}_s$. A global context vector $\boldsymbol{c}_t$ is then computed as the weighted average, according to $\boldsymbol{a}_t$, over all the source states.

Local Attention Model / Position Encoding

- The global attention has a drawback that it has to attend to all words on the source side for each target word, which is expensive and can potentially render it impractical to translate longer sequences, e.g., paragraphs or documents
- To address this deficiency, we propose a local attentional mechanism that chooses to focus only on a small subset of the source positions per target word.
- The alignment becomes

$$a_t(s) = \mathit{align}(\mathbf{z}_t, \bar{\mathbf{z}}_s) \exp\left(\frac{(s - p_t)^2}{2\sigma^2}\right)$$

- p_t is the expected position in the input sequence predicted from $\mathbf{z}_t$ using a neural network

$$p_t = Ssig(v_p^T \tanh(W_p \mathbf{z}_t))$$

  S is the source sentence length

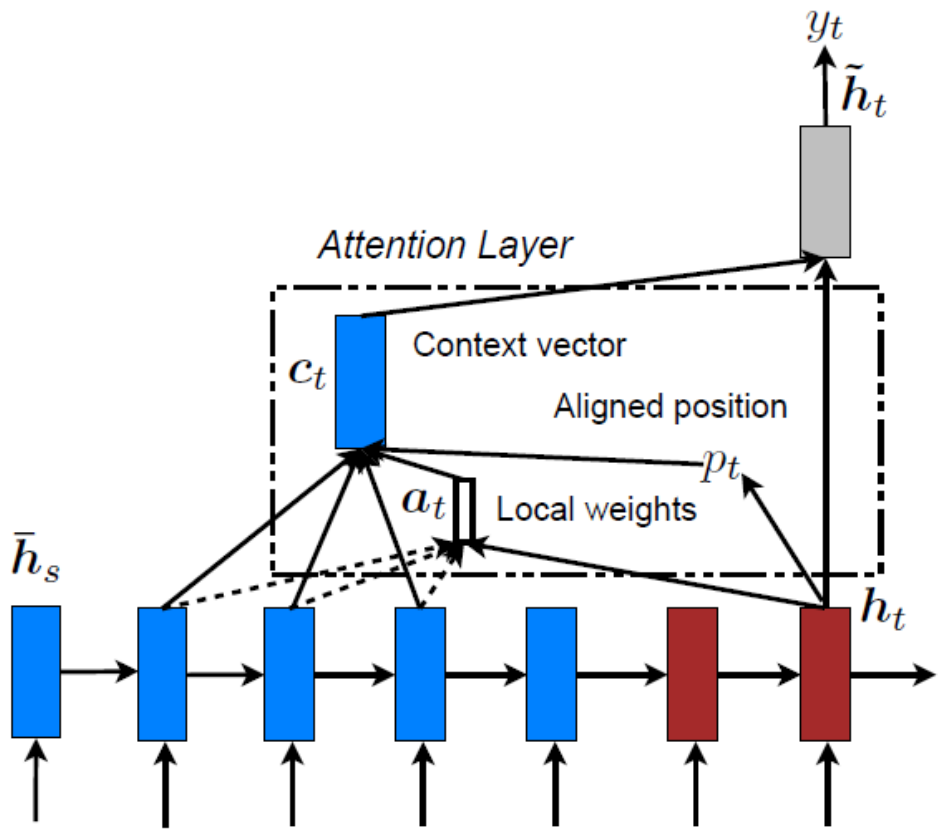
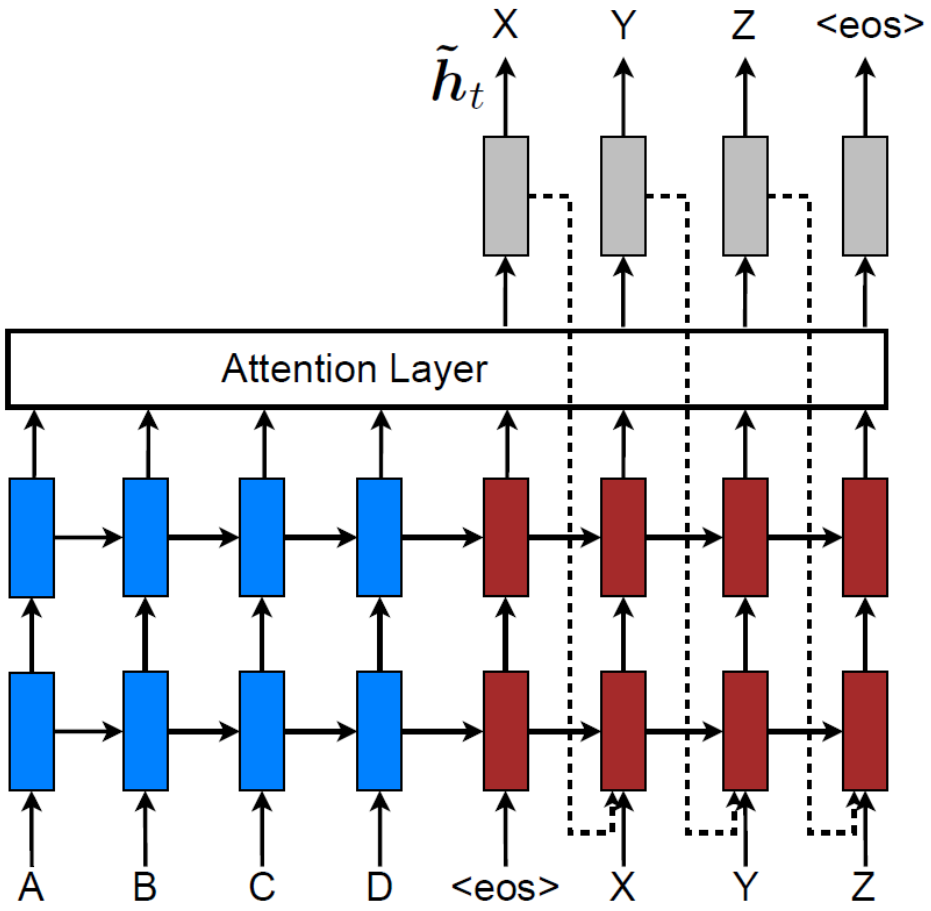


Figure 3: **Local attention model** – the model first predicts a single aligned position p_t for the current target word. A window centered around the source position p_t is then used to compute a context vector $\boldsymbol{c}_t$, a weighted average of the source hidden states in the window. The weights $\boldsymbol{a}_t$ are inferred from the current target state $\boldsymbol{h}_t$ and those source states $\bar{\boldsymbol{h}}_s$ in the window.

Overall Architecture

- The next figure shows again the architecture (ignore the dashed lines)


X
Y
Z
<eos>
$\tilde{h}_t$
Attention Layer
A
B
C
D
<eos>
X
Y
Z

Self-Attention

- An attention mechanism can be applied to any deep neural network
- Self-attention can replace convolutional and recurrent approaches ("attention is all you need")
- In self-attention, the activation of a hidden layer $\mathbf{z}_t$ is calculated based on a previous layer $\mathbf{x}_t$ of all entities/data points as

$$\mathbf{z}_t = \mathit{sig}\left(V\mathbf{x}_t + D\mathbf{c}_t\right)$$

- Here,

$$\mathbf{c}_t = \sum_{t'} a(\mathbf{x}_t, \mathbf{x}_{t'})\mathbf{x}_{t'}$$

- The *sig* is typically the tanh

Comparison

- **Feed forward neural network**

$$\mathbf{z}_t = sig\,(V\mathbf{x}_t)$$

so here each word label at position t is predicted separately; this i sthe i.i.d situation

- **Fully connected** (not used)

$$\mathbf{z}_t = sig\left(V\mathbf{x}_t + \sum_{t'} C_{t,t'}\mathbf{x}_{t'}\right)$$

The embeddings of all words are the inputs to one neural network; here one would need to use a standard length sentence (short sentences are dealt with by zero-passing); a problem with this approach is the huge number of parameters in the neural network

Comparison (cont'd)

- **Convolutional layer**

$$\mathbf{z}_t = sig\left(V\mathbf{x}_t + \sum_k \sum_{t'} C^k_{t-t'}\mathbf{x}_{t'}\right)$$

  Very powerful approach and very successful in NLP; needs zero padding at sentence boundaries

- **Recurrent neural networks**

$$\mathbf{z}_t = sig\left(V\mathbf{x}_t + B\mathbf{z}_{t-1}\right)$$

  Very powerful approach and very successful in NLP; often LSTM units are used

- **Bidirectional recurrent neural networks**

$$\mathbf{z}_t = [\mathbf{z}^f_t ; \mathbf{z}^b_t] = \left[sig\left(V^f\mathbf{x}_t + B^f\mathbf{z}^f_{t-1}\right) ; sig\left(V^b\mathbf{x}_t + B^b\mathbf{z}^b_{t+1}\right)\right]$$

Comparison (cont'd)

- **Self-Attention**

$$\mathbf{z}_t = sig\,(V\mathbf{x}_t + D\mathbf{c}_t)$$

$$\mathbf{c}_t = \sum_{t'} a(\mathbf{x}_t, \mathbf{x}_{t'})\mathbf{x}_{t'}$$

  (the *sig* is often the tanh) self-attention can replace convolutional or recurrent layers

- Compare to graph convolution networks (GCNs): here $\mathbf{c}_t$ is the average over the neighbors in the graph, including $\mathbf{x}_t$; $V = 0$;

- GCNs with attentions: Graph Attention Networks: enhances GCNs with an attention mechanism

APPENDIX: Transformer

Transformer

- Bottleneck of previous approaches in NMT: sequential processing at the encoding step
- The Transformer **dispensed the recurrence and convolutions** involved in the encoding step entirely and based models only on attention mechanisms to capture the global relations between input and output
- Each layer has two sub-layers comprising multi-head attention layer followed by a position-wise feed forward network.

Encoder

- Consider self-attention with

$$\mathbf{c}_t = \sum_{t'} a(W^Q \mathbf{x}_t, W^K \mathbf{x}_{t'})\, W^V \mathbf{x}_{t'}$$

  The W^Q, W^K, W^V, matrices are not necessarily quadratic: the dimension of $\mathbf{c}_t$ might be different from the dimension of $\mathbf{x}_t$

- **The transformer uses Multi-Head Attention,**

$$\mathbf{c}_t = \sum_{t'} \sum_{k} a(W_k^Q \mathbf{x}_t, W_k^K \mathbf{x}_{t'})\, W_k^O W_k^V \mathbf{x}_{t'}$$

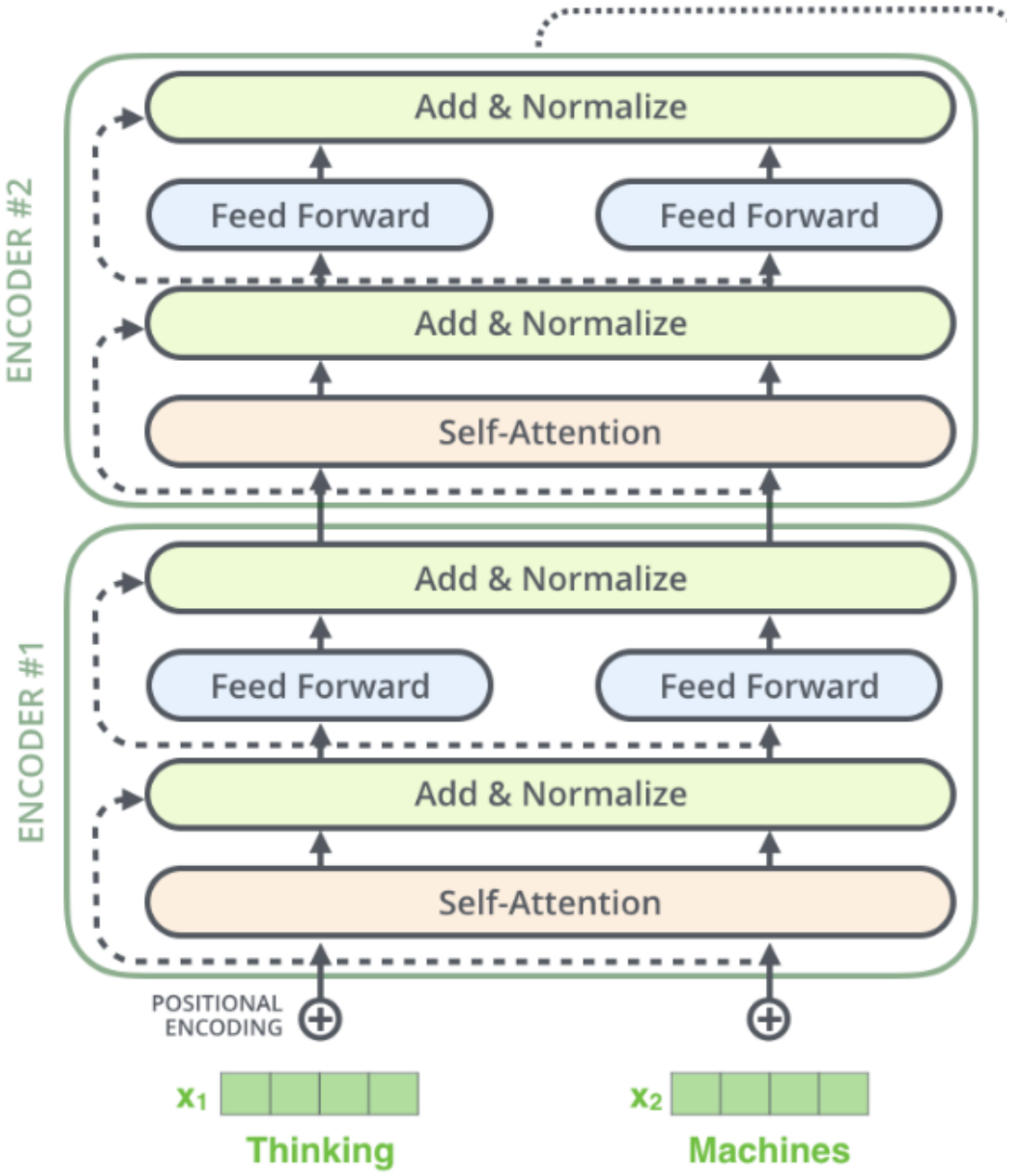
  The dimensions of W_k^O are such that the dimensions of $\mathbf{c}_t$ and $\mathbf{x}_t$ agree

- Then it uses a simple feedforward neural network $\mathbf{f}_\mathbf{w}(\cdot)$ to compute

$$\mathbf{z}_t = V\mathbf{x}_t + D\mathbf{c}_t + \mathbf{f}_\mathbf{w}\left(V\mathbf{x}_t + D\mathbf{c}_t\right)$$

Encoder (cont'd)

- In training masking operations are being used to hide the downstream target words
- It adds another normalization: Layer Normalization (related to batch normalization)
- Positional Encoding: To address this, the transformer adds a vector to each input embedding. These vectors follow a specific pattern that the model learns, which helps it determine the position of each word, or the distance between different words in the sequence.


ENCODER #2
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
POSITIONAL ENCODING
x1
x2
Thinking
Machines

Decoder

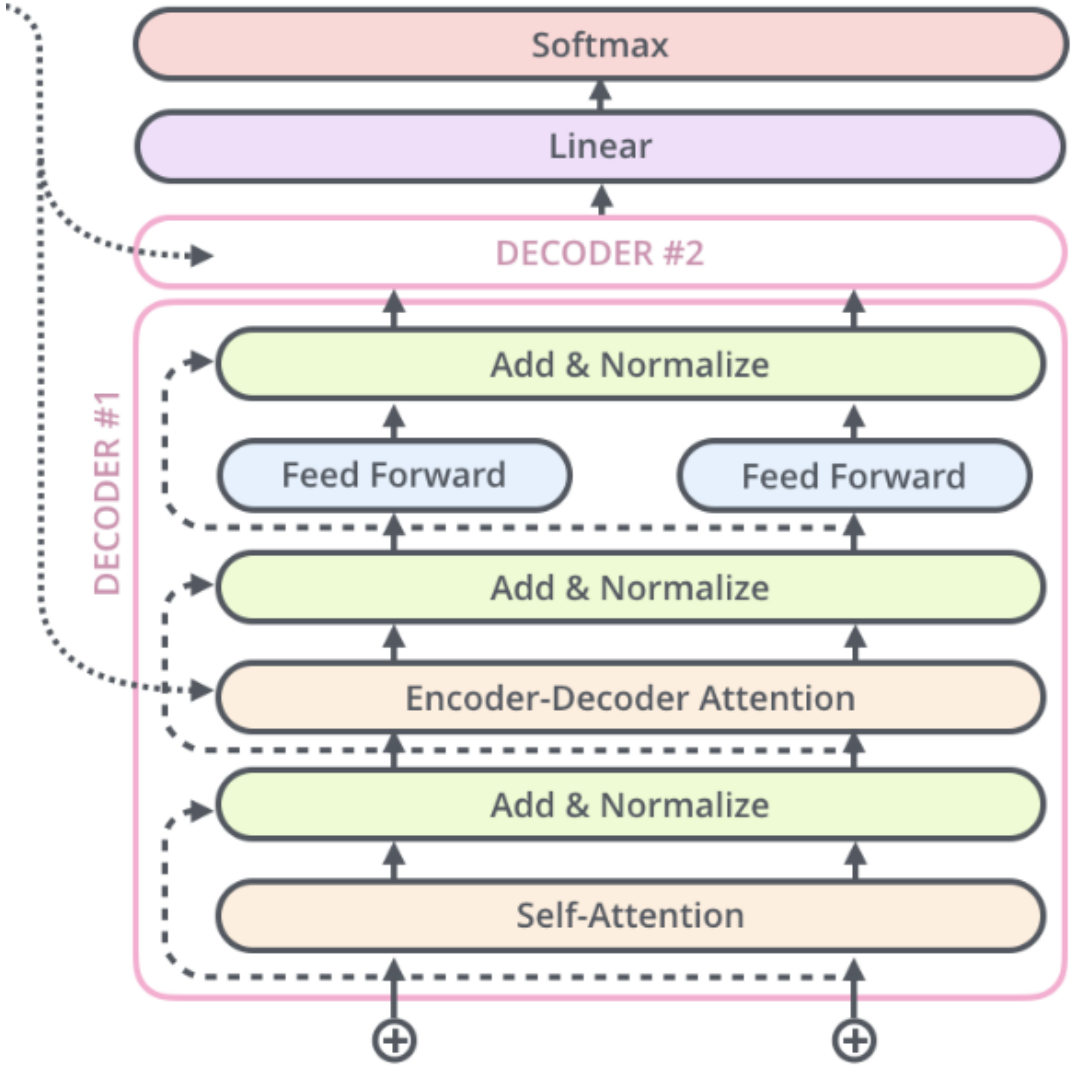
- The decoder is similar to the encoder but uses two more layers
- We start with the $\mathbf{z}_s$ from the encoder (the output from the top payer; encoder stack)
- The inputs are the previously decoded words; we start with no decoded word
- We calculate the $\mathbf{c}_t$, as before, with the decoded words as input and with Multi-Head Attention
- Then we do a multi-head attention with the **encoder latent representations** $\{\mathbf{z}_s\}_s$ (last layer)

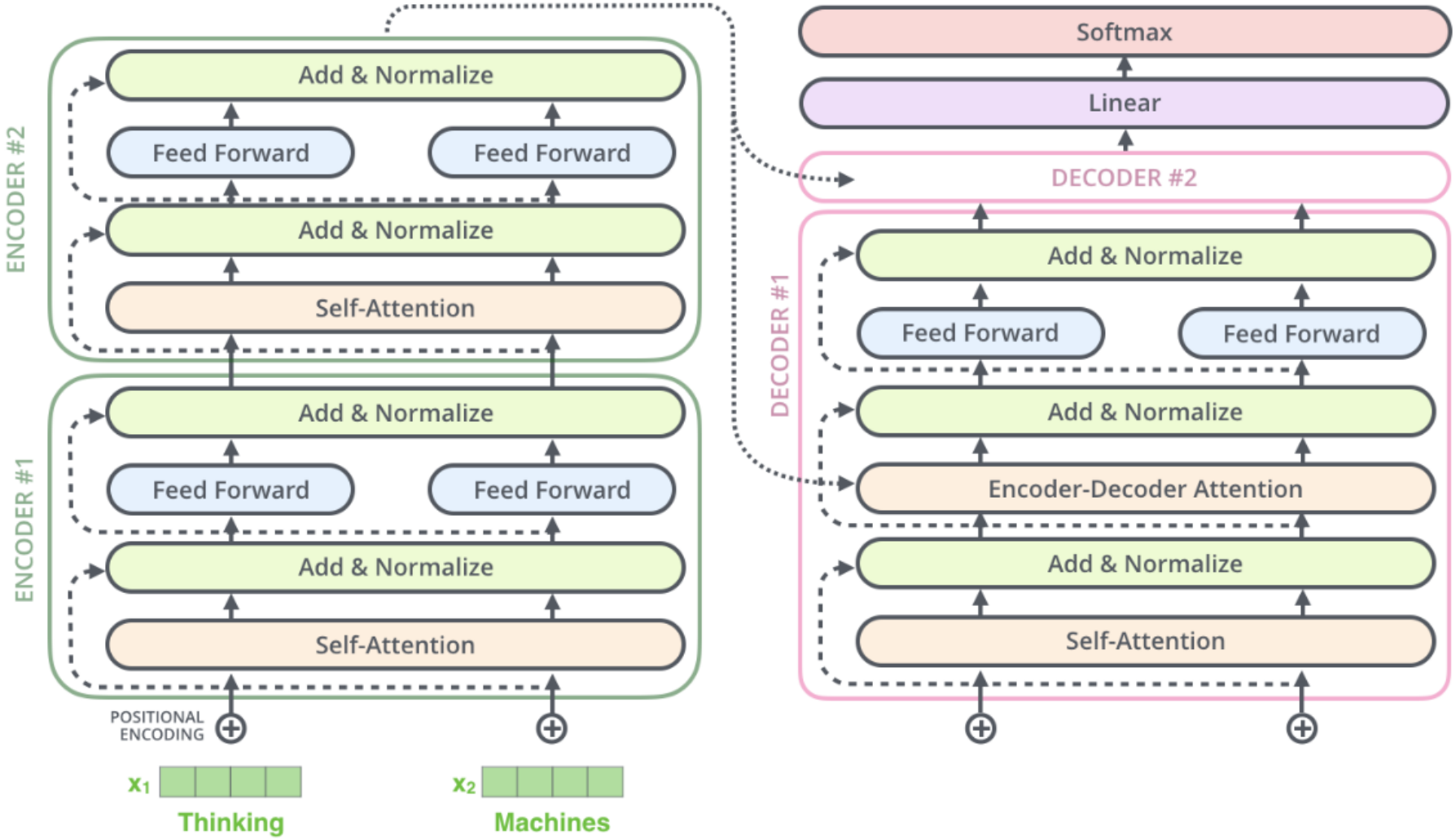
Decoder (cont'd)

- Multi-Head Attention:

$$\mathbf{c}_t = \sum_k \sum_s a(W_k^Q \mathbf{x}_t, W_k^K \mathbf{z}_s)\ W_k^O W_k^V \mathbf{z}_s$$

- Positional Encoding: To address this, the transformer adds a vector to each input embedding. These vectors follow a specific pattern , which helps to determine the position of each word, or the distance between different words in the sequence.


Softmax
Linear
DECODER #2
DECODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Encoder-Decoder Attention
Add & Normalize
Self-Attention


Softmax
Linear
DECODER #2
DECODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Encoder-Decoder Attention
Add & Normalize
Self-Attention
ENCODER #2
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
ENCODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
POSITIONAL ENCODING
x1
x2
Thinking
Machines