Time Series and Sequential Data

Volker Tresp

Winter 2024-2025

Time Series Modelling

- I have to predict the total energy consumption of a city for tomorrow, based on certain inputs (weather forecast: temperature, precipitation, wind; then: working day/holiday, ...)
- It would help to also consider the energy consumption of today and maybe of yesterday as inputs
- Added complexity: If my goal is to predict the energy consumption two days in the future, my own prediction for tomorrow becomes an input for the prediction for two days in the future
- In time series modelling, outputs, and often also the inputs, are real numbers

DAX Performance-Index

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

Sequence Modelling

- Sequence classification (encoder): The input is a sentence, i.e., a sequence of words; the output classifies the sentiment of the sentence
- Decoder or Generator models
- Encoder-decoder (sequence-to-sequence) modelling: The input is a sentence, i.e., a sequence of words, in English; the output is the sentence translated into German
- In sequence modelling, inputs and outputs are typically discrete

I. Time Series Modelling: NARX Models

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 scalars. 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 the function, using a neural network,

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

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

- A reasonable cost function is

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

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

- It is important to note, that 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 model is called 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

$$\widehat{y}_t = f_{\mathbf{w},\mathbf{V}}(y_{t-1}, \ldots, y_{t-T}, x_{t-1}, \ldots, x_{t-T})$$

Self-supervised Learning

- The time-series provides its own labels
- No human labelling is necessary: self-supervised learning

Multiple-Step Prediction based on Multiple Step Prediction

- We can also train a model to predict τ time steps into the future; the prediction then becomes

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

- 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

Mutiple-Step Prediction based on Single-Step Prediction

- Why not just iterate the single-step prediction? One issue is that my prediction is uncertain, so I should consider that uncertainty; second: I do not have future inputs!
- One way is simulation; for y_t we have the model as before, $(t' = t, \ldots, t + \tau)$

$$y_{t'} = f_{\mathbf{w},\mathbf{V}}(y_{t'-1}, \ldots, y_{t'-T}, x_{t'-1}, \ldots, x_{t'-T}) + \epsilon_{t'}$$

- Using both we can generate samples for the future; for the noise I might assume a Gaussian distribution $\epsilon_t \sim \mathcal{N}(0, \sigma^2)$
- Future inputs $x_{t'}$, we either set to zero, or we develop a separate prediction model for those as well
- For multiple-step prediction, we can simulate (i.e., sample) for the desired number of time steps in the future (Monte-Carlo simulation) repeatedly and can derive estimated means, variances, and covariances

Residual Modeling

- We have

$$\widehat{y}_t = y_{t-1} + f_{\mathbf{w},\mathbf{V}}(y_{t-1}, \ldots, y_{t-T}, x_{t-1}, \ldots, x_{t-T})$$

- Realize the similarity to ResNet

Considering the Complete History?

- Consider a prediction model that uses the complete history,

$$\widehat{y}_t = f_{\mathbf{W},\mathbf{V}}(y_{t-1}, \ldots, y_1, x_{t-1}, \ldots, x_1)$$

  This means that the time window grows with t: $T := t - 1$

- Technical solutions:
- 1: Models with an internal memory: RNNs, LSTMs
- 2: Models with the ability to grow: Transformers

GPT-type Architecture

- GPT always considers the complete history

$$\widehat{y}_t = f_{\mathbf{w},\mathbf{V}}(y_{t-1}, \ldots, y_1)$$

- A generated sequence (text) is a simulation of the future
- The first K steps $y_1, \ldots, y_K$ are the input (prompt) from the user
- y is a discrete variable with as many states as there are tokens (words)

II. Sequence Modelling

Encoding Inputs and Outputs

- So far we considered that x_t is either binary $x_t \in \{0, 1\}$ or continuous, $x_t \in \mathbb{R}$
- How do we encode that $x_t \in \{0, 1, \ldots, N^{words}\}$, where N^{words} is the number of words in the vocabulary?
- We could consider that $x_t \in \mathbb{R}$ and encode it as a scalar (amplitude encoding): this is not commonly done
- Alternatively, we introduce N^{words} binary variables with $x_{t,j} = \mathbb{I}(word(t) \equiv j)$ (one-hot encoding) ($\mathbb{I}()$ is the indicator function which is equal to 1, when the argument is true, otherwise zero)

(A): One-hot Encoding

- This type of encoding is called one-hot encoding
- We train the input to hidden matrix $\mathbf{V}$, which is an $H \times N^{words}$ dimensional matrix

(B): Embedding Encoding

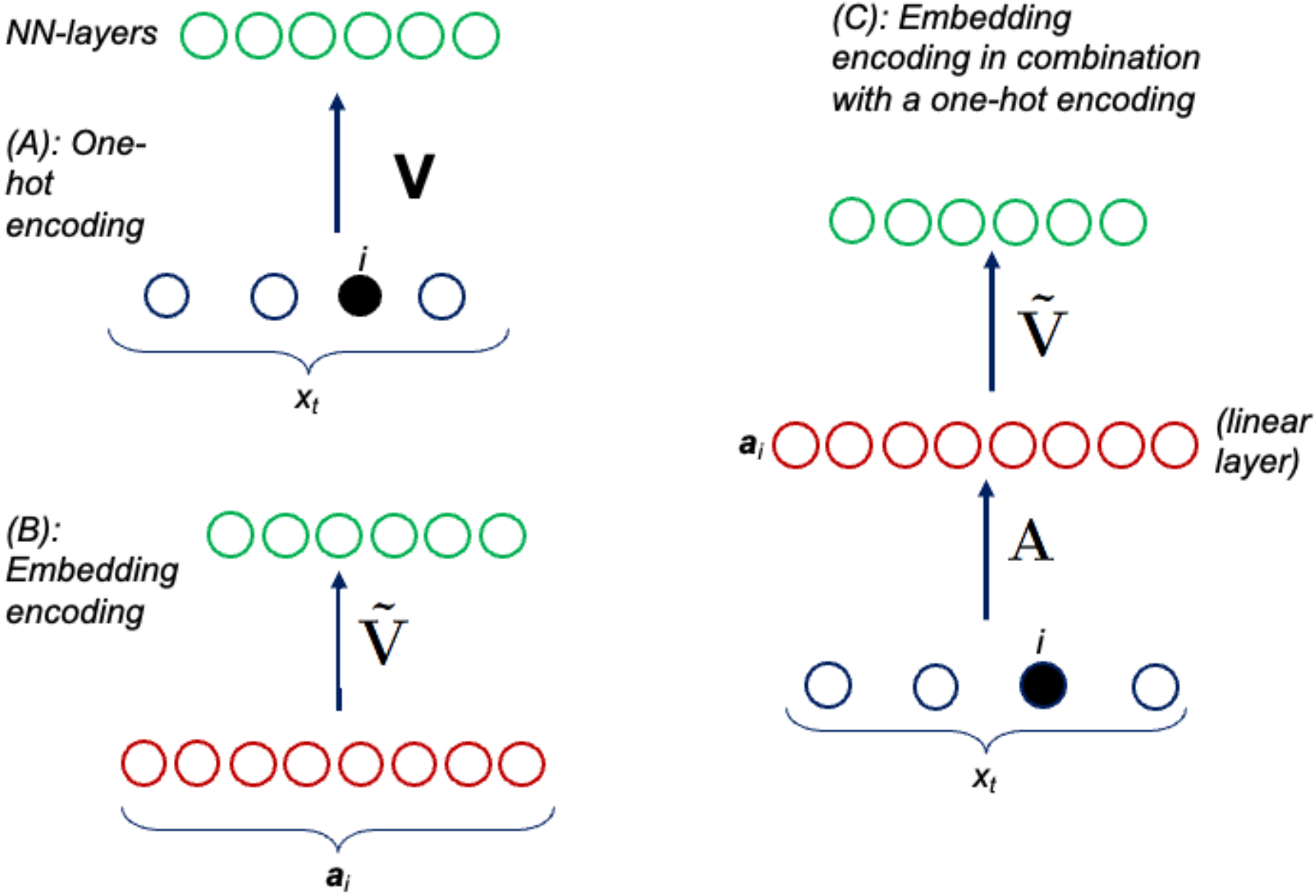
- Maybe we should represent a word by its attributes? But what attributes?
- An embedding vector $\mathbf{a}_i$ for word i is a vector of abstract attributes that represent the word and which might have been derived from a large vocabulary and is shared between applications
- This embedding vector might have been generated by some other research group and is simply a vector of real numbers of length r (rank)
- Now the input to hidden connection matrix is $\tilde{\mathbf{V}}$ which is an $H \times r$ dimensional matrix

(C): Embedding Encoding in Combination with One-hot Encoding

- Sometimes it is more intuitive to consider a matrix $\mathbf{A}$ connecting the one-hot encoded input with the first hidden layer
- The i-th column in matrix $\mathbf{A}$ contains $\mathbf{a}_i$

Relationship Between Encodings

- (C) is identical to (B)
- (A) is identical to (C) and (B), if we set $\mathbf{V} = \tilde{\mathbf{V}}\mathbf{A}$


NN-layers
(A): One-hot encoding
V
i
x_t
(B): Embedding encoding
$\tilde{V}$
a_i
(C): Embedding encoding in combination with a one-hot encoding
$\tilde{V}$
a_i
(linear layer)
A
i
x_t

IIa. Representation Models and Language Models

Language Model

- The ideas is to predict the next word (out off a vocabulary of N^{words} words) in a text, based on the last T words
- Consider we want to predict y_t: y_t has as N^{words} components, one for each word (one-hot encoding)
- 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 (embedding representation)
- 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 (embedding with one-hot)

Language Model (cont'd)

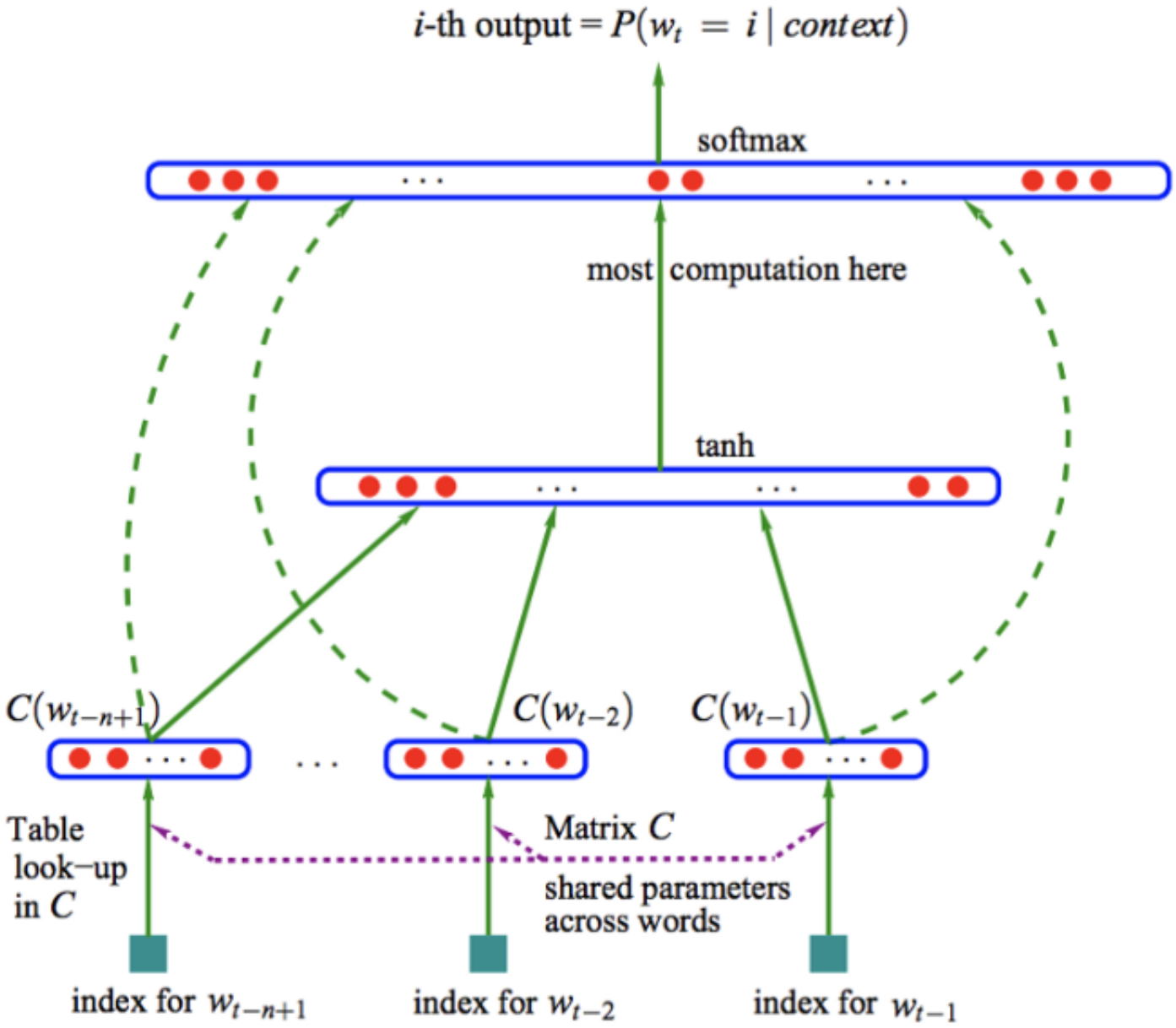
- We get

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

where $i(t-m)$ is the index of the word at position $t-m$ and where $\mathbf{f_w}(\cdot)$ is a neural network with one hidden layer and N^{words} output neurons

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 $\mathbf{A}$ 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}

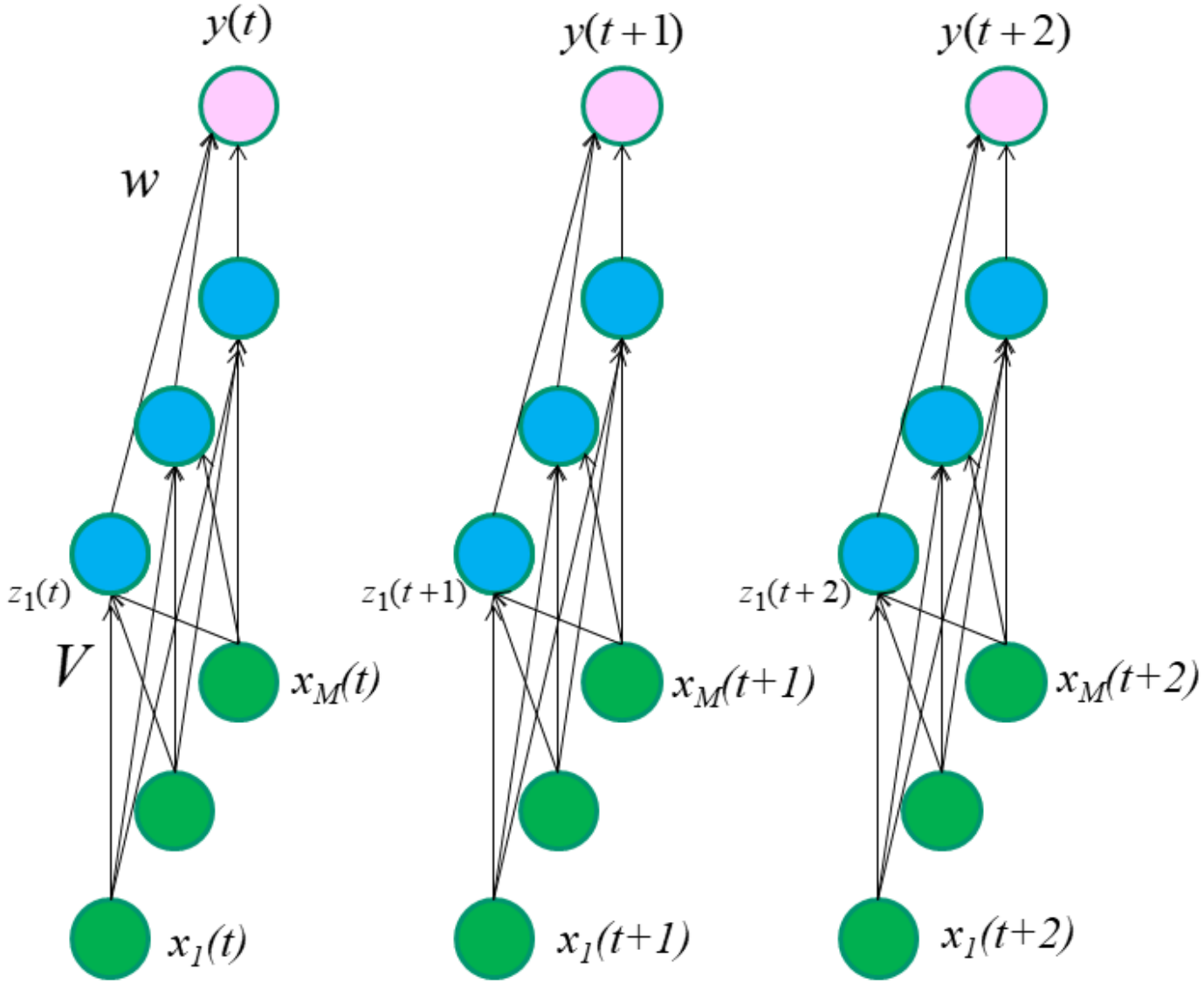
IIb. Recurrent Neural Networks

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
- We do not have to specify a time window T: an RNN can consider the whole history

A Feedforward Neural Network with a Time Index

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



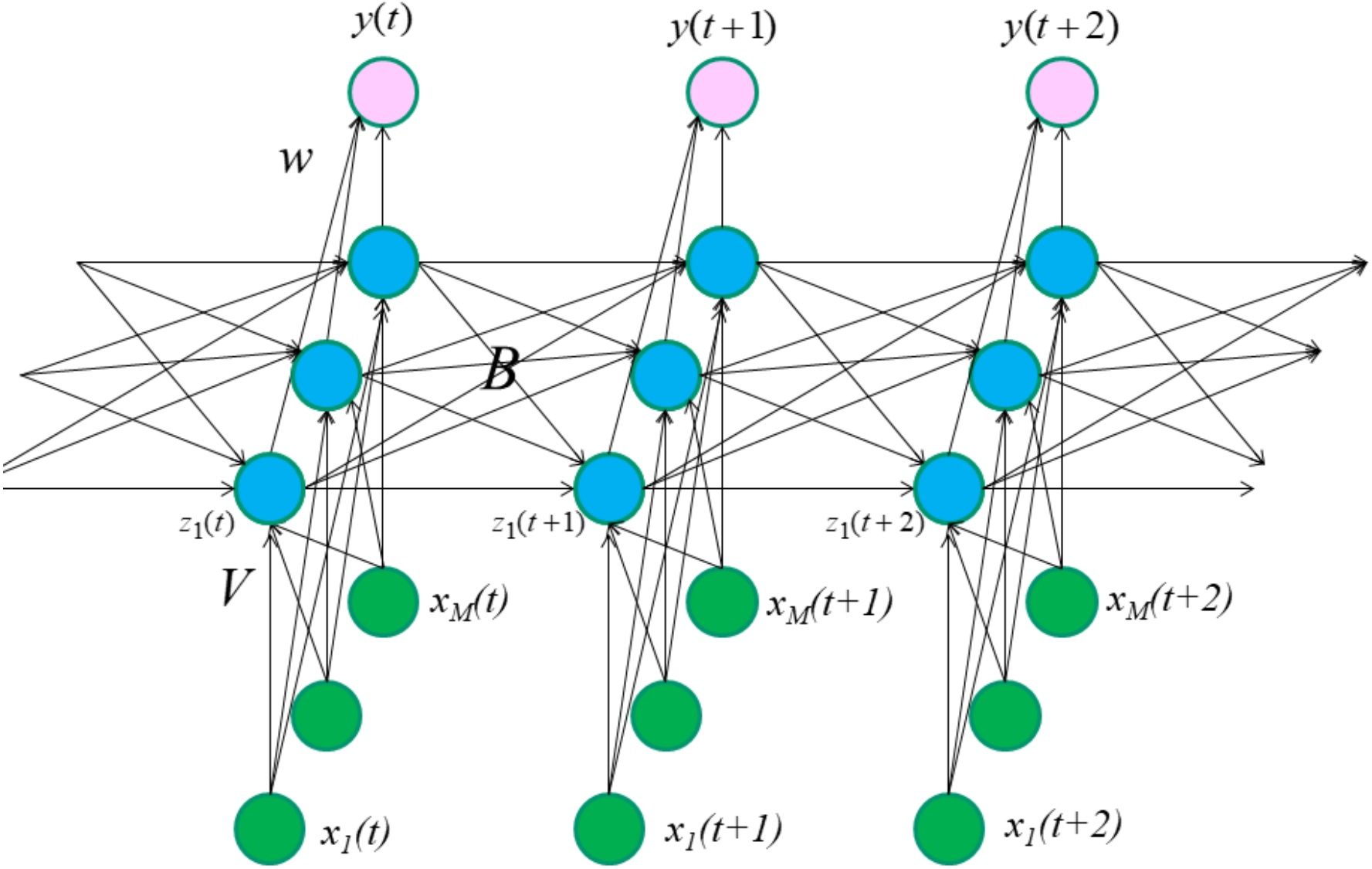
feed forward neural network at time step t

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,

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

$$y_t = \mathsf{sig}(\mathbf{w}^\top \mathbf{z}_t)$$

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

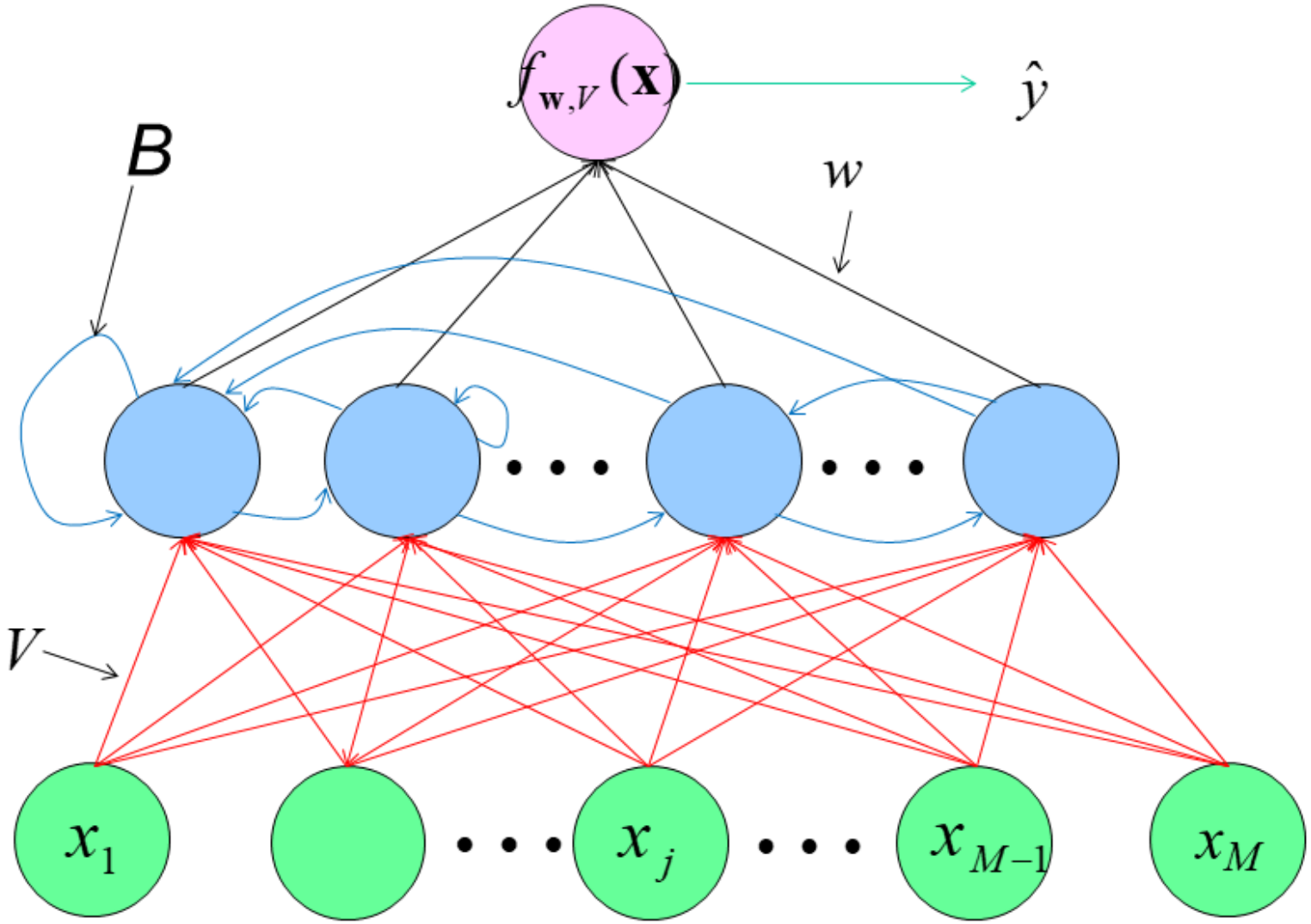
$$\mathbf{y}_t = \mathsf{softmax}(\mathbf{W}\mathbf{z}_t)$$

Temporal Representation

- $\mathbf{z}_t$ is the representation of the hidden state of the system (e.g., patient, plant, ...) at time t
- $\mathbf{x}_t$ can be the embedding of a thing which is present or active at time t (e.g., word, medication, ...)
- Word embedding: $\mathbf{x}_t = \mathbf{a}_{i(t)}$, where $i(t)$ is the word at t and with $M = r$
- This is a link to representation learning

Recurrent Representation

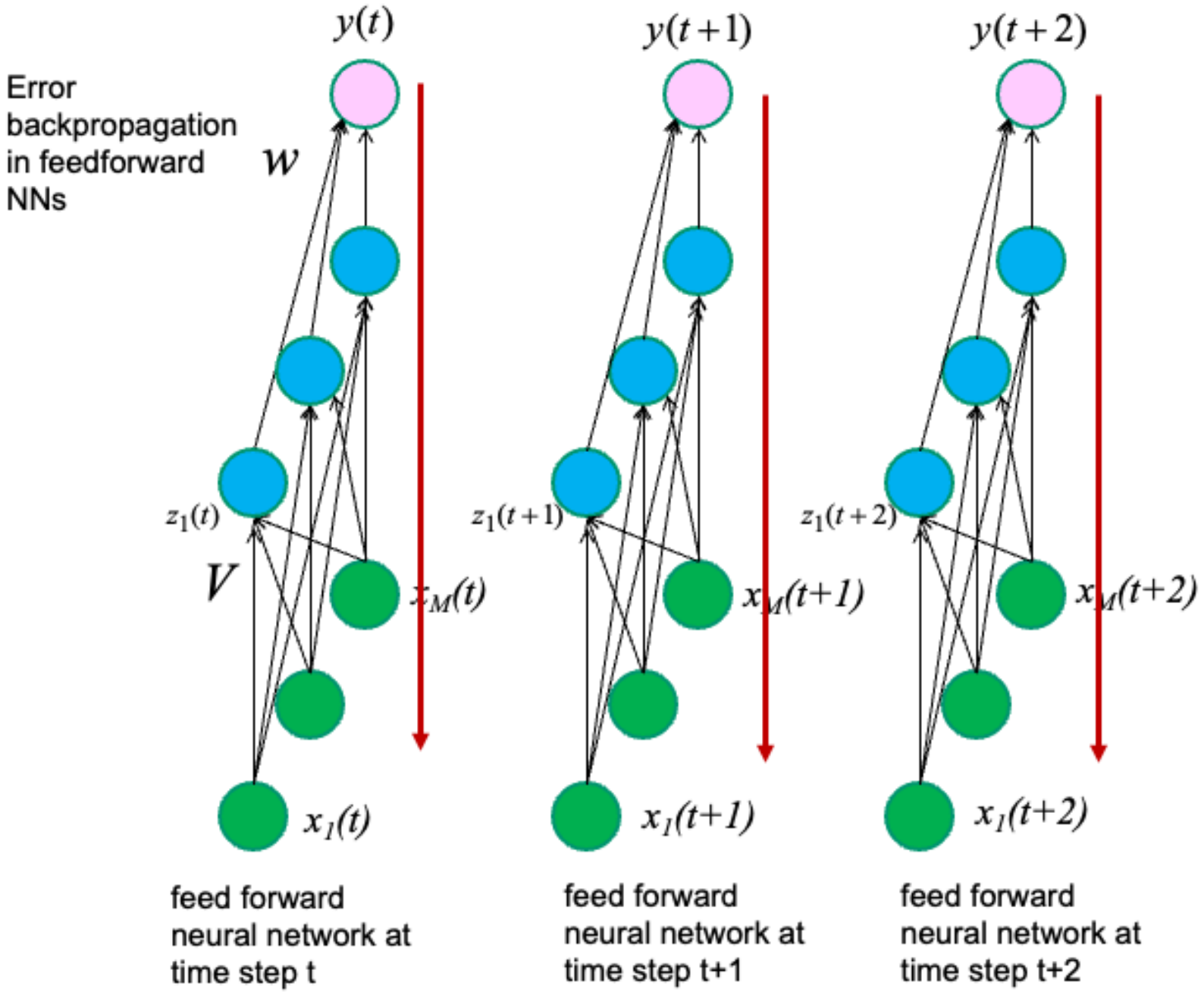
- The next slide shows an RNN as a recurrent structure
- If $\mathbf{V}$ is sparse (contains many zero-entries), the inputs directly only influence a small number or hidden neurons
- If $\mathbf{B}$ is sparse (contains many zero-entries), most hidden neurons are not directly coupled

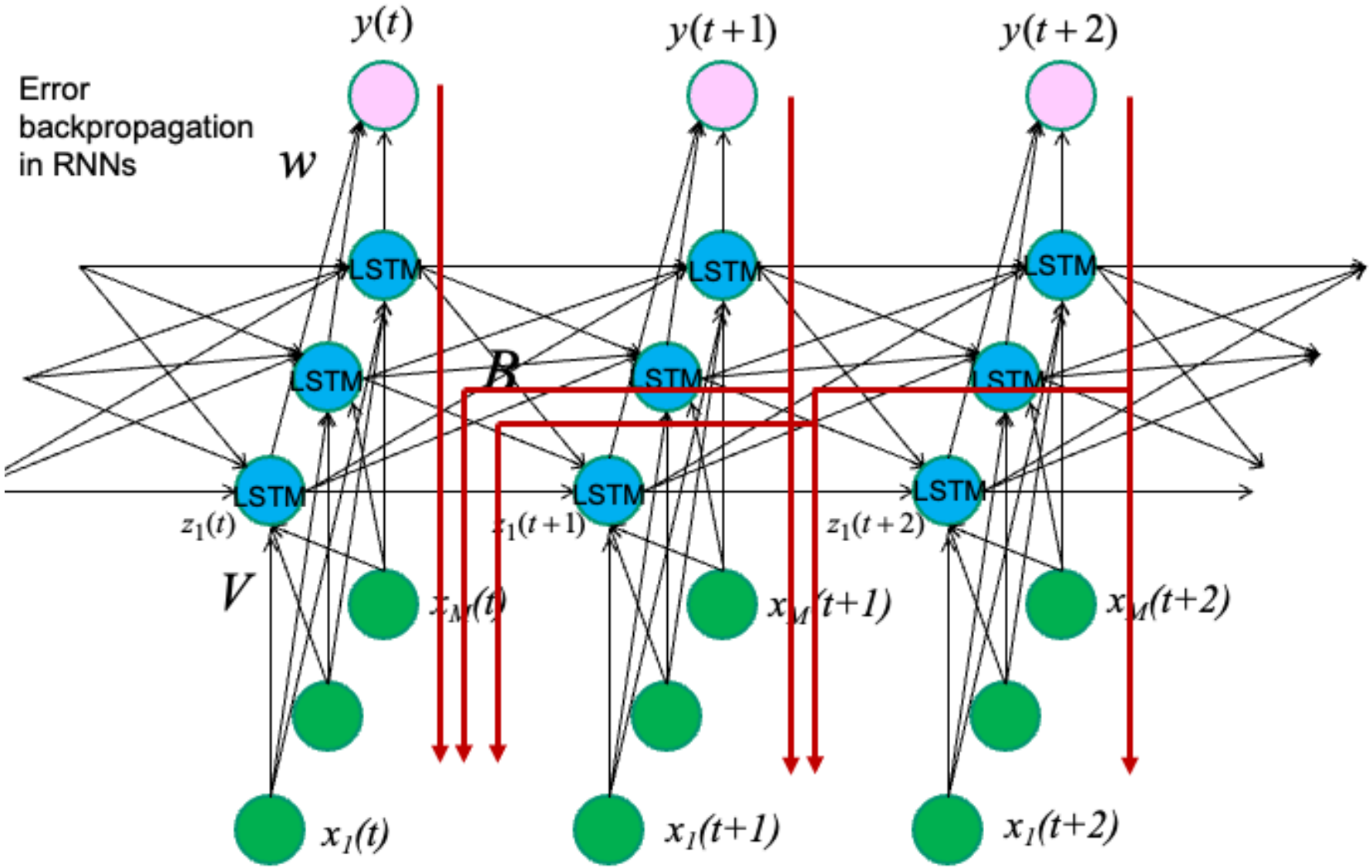


recurrent neural network showing recurrent connections

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

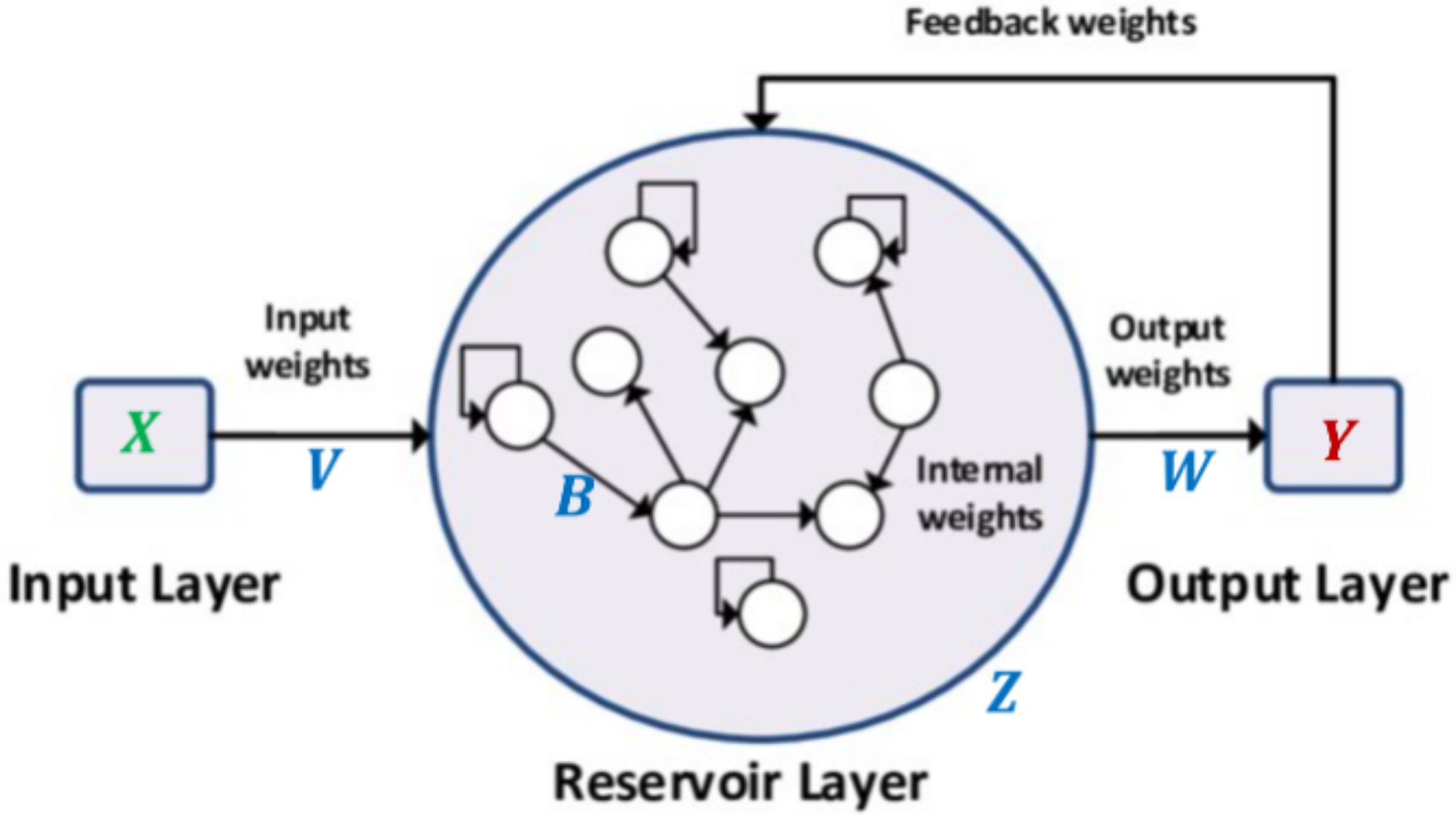

Error backpropagation in feedforward NNs
y(t)
y(t+1)
y(t+2)
w
V
z1(t)
z1(t+1)
z1(t+2)
xM(t)
xM(t+1)
xM(t+2)
x1(t)
x1(t+1)
x1(t+2)
feed forward neural network at time step t
feed forward neural network at time step t+1
feed forward neural network at time step t+2



Recurrent neural network, unfolded in time

Echo-State Network

- Recurrent Neural Networks are sometimes difficult to train
- A simple alternative is to initialize $\mathbf{B}$ and $\mathbf{V}$ randomly (according to some recipe) and only train $\mathbf{W}$
- $\mathbf{W}$ 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) (Herbert Jaeger, 2007)
- ESN (and also liquid-state machines) are examples of so called *reservoir computing*


Feedback weights
Input weights
Output weights
X
V
B
Internal weights
W
Y
Input Layer
Output Layer
Z
Reservoir Layer

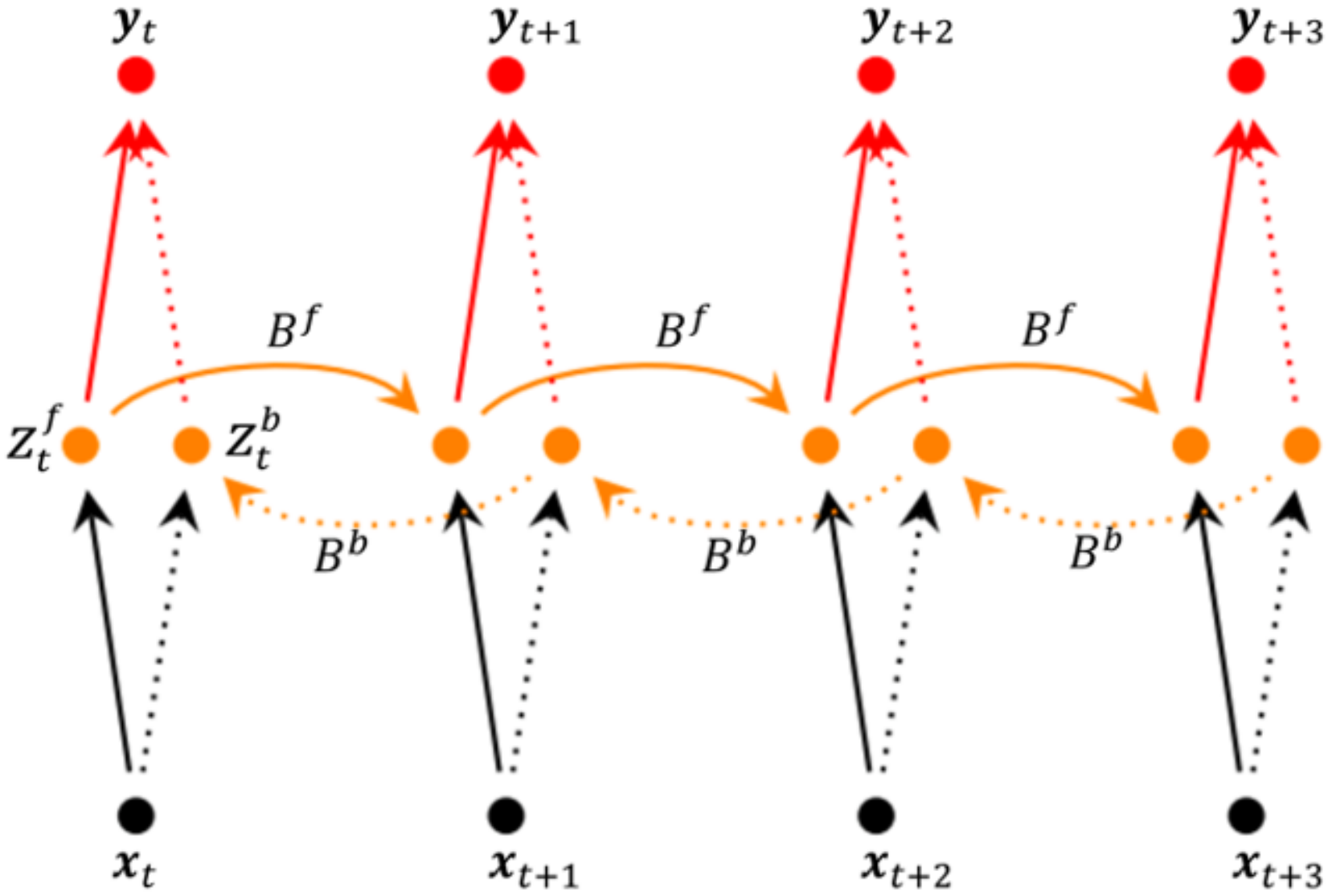
Issues in Prediction

- 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
- One reason that we can simply apply backpropagation is that RNNs are deterministic! To train HMMs and Kalman filters one can apply a form of EM learning (expectation maximization)

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

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


y_t
y_{t+1}
y_{t+2}
y_{t+3}
B^f
B^f
B^f
Z_t^f
Z_t^b
B^b
B^b
B^b
x_t
x_{t+1}
x_{t+2}
x_{t+3}

IIc. LSTMS

Issues in Prediction

- Although the RNN has a memory, it has difficulties remembering important information far in the past
- Bottleneck problem: the latent vector (hidden state) needs to encode information maybe 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

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

$$\hat{\mathbf{y}}_t = \text{sig}(\mathbf{w}^\top \mathbf{z}_t)$$

- The transfer function of the hidden neuron is a bit strange, $\tanh(\tanh(\mathbf{V}\mathbf{x}_t))$
- $\mathbf{s}_t$ is called the **cell state vector** (pre-activations), $\mathbf{z}_t$ is the **output vector** (of the units, not the neural network) (post-activation); $\mathbf{z}_t$ feeds to the next upper layer (could be the output $y((t))$ and to the next hidden layer in the sequence
- 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 $\mathbf{q}_t$

$$\mathbf{u}_t = \tanh(\mathbf{V}\mathbf{x}_t) \quad \mathbf{s}_t = \mathbf{g}_t \circ \mathbf{u}_t \quad \mathbf{z}_t = \mathbf{q}_t \circ \tanh(\mathbf{s}_t)$$

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

$$\widehat{y}_t = \text{sig}(\mathbf{w}^\top \mathbf{z}_t)$$

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

$$\mathbf{g}_t = \text{sig}(\mathbf{V}^g \mathbf{x}_t) \quad \mathbf{q}_t = \text{sig}(\mathbf{V}^q \mathbf{x}_t)$$

- 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

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

- Input Gate

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

- Output Gate

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

Cell State Vector with Self-recurrency and Forget Gate

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

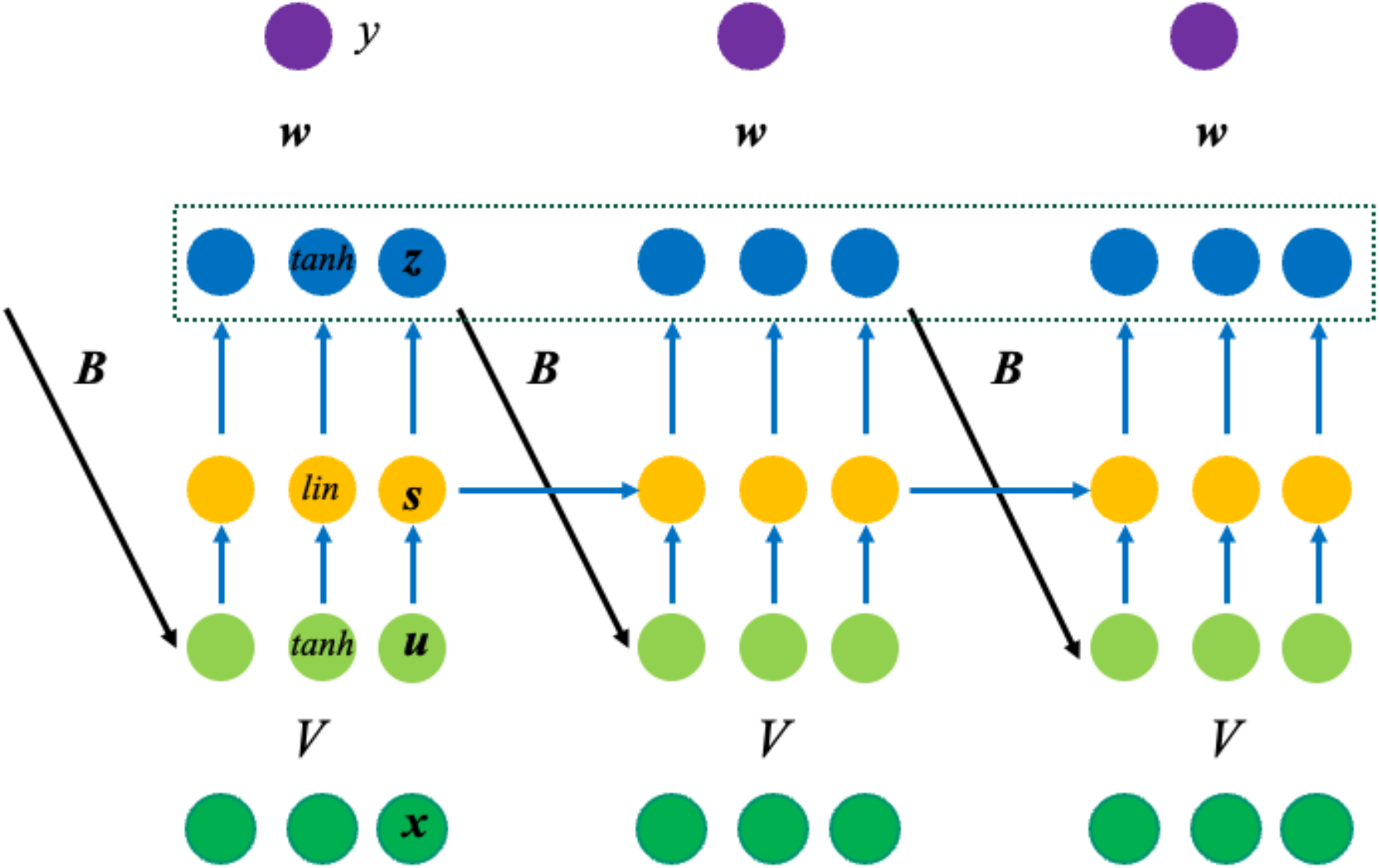
$$\mathbf{s}_t = \mathbf{f}_t \circ \mathbf{s}_{t-1} + \mathbf{g}_t \circ \mathbf{u}_t$$

- Forget gate

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

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, ...)
- Consider an LSTM without the gates; then $\mathbf{s}_t$ is the vector of pre-activations and $\mathbf{z}_t$ the vector of post-activations
- It is similar to a ResNet architecture where the link from $\mathbf{s}_{t-1}$ to $\mathbf{s}_t$ is the skip connection



LSTM without the gates

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

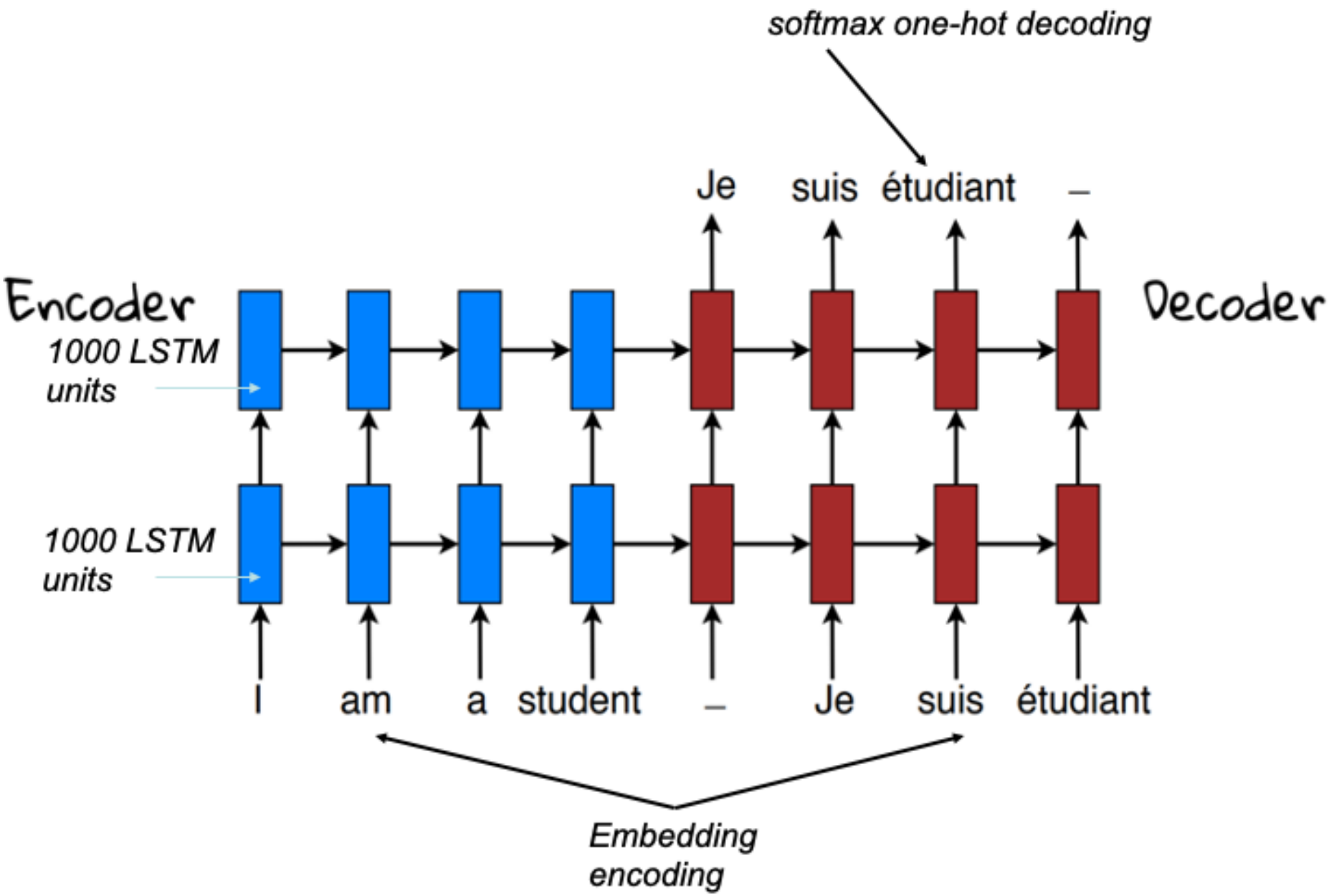
Comments on LSTM

- You cannot easily do transfer learning with LSTMs (does not work very well): thus you need a large data set for any new problem

IId. 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


softmax one-hot decoding
Je
suis
étudiant
–
Encoder
1000 LSTM units
1000 LSTM units
Decoder
I
am
a
student
–
Je
suis
étudiant
Embedding encoding

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 (two) embedding vectors (hidden states) of $(-)$, i.e., the end-of-sentence symbol

Decoder

- The initial latent states of the decoder are the encoder vectors (the first two red rectangles in the figure)
- In its simplest form, the latent state of the decoder evolves as

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

$$\hat{\mathbf{y}}_t = \text{softmax}(\mathbf{W}\mathbf{z}_t)$$

- In training, the input to the decoder is the *embedding of the previous word* $\mathbf{a}_{t-1}$; the output is the one-hot encoding of the current word: $\mathbf{y}_t$ is a one-hot vector
- Training is based on bilingual, parallel corpora; each hidden layer might consist of 1000 hidden units

Decoder: Prediction

- In prediction, one finds the most likely decoded sequence of words (e.g., using beam search); teacher forcing: the detected word appears at the input of the next instance
- In the simplest case: $\mathbf{a}_{t-1}$ is the embedding vector of the previous output token with index $i = \arg\max_{i'} \ \widehat{y}_{t-1,i'}$
- 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

IIe. Attention

Introduction

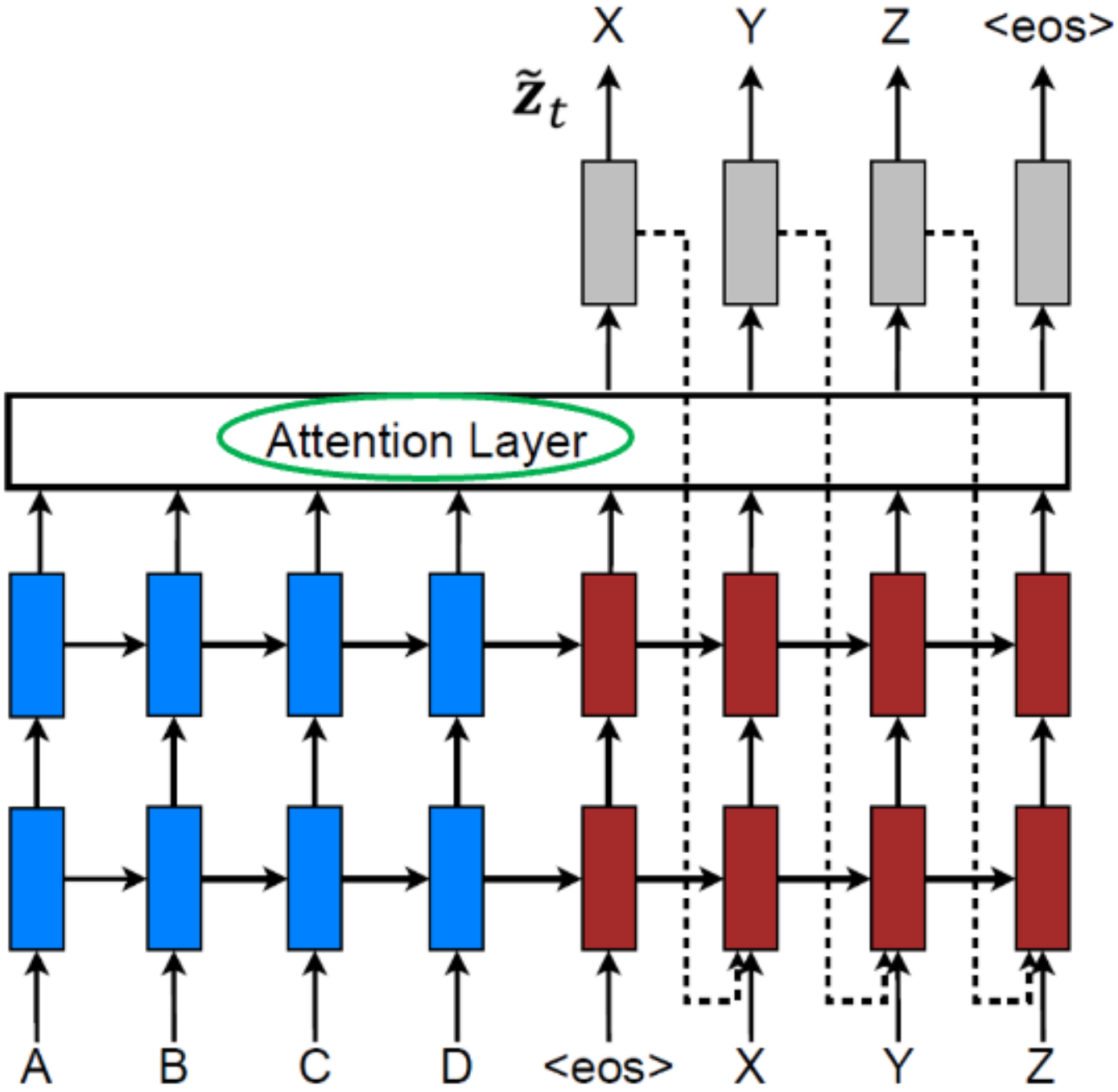
- The concept of "attention" has gained popularity recently in training neural networks, allowing models to learn alignments between different modalities, e.g., between image objects and agent actions in the dynamic control problem, between speech frames and text in the speech recognition task, or between 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

Bottleneck in the Encoder-Decoder Architecture

- In the encoder-decoder architecture, all information about the input sequence needs to be transported through the two encoding embedding vectors
- Information earlier in the sequence tends to get forgotten
- One needs short cuts: maybe earlier embedding vectors are important as well!
- In attention one provides information about the top embeddings (upper layers) to the decoder; attention does it in a way that avoids overfitting

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) in the fourth layer


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

Attention

- Let $\mathbf{z}_t$ (red) be a hidden state vector of interest in the **decoder** (so called target at t; also called the **query**)
- Let $\mathbf{c}_t$ be the source-side **context vector** (derived further down)
- The attentional hidden state (grey) is

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

  (Note that $\mathbf{V}$ is the connection matrix to the previous layer and not to the inputs)
- 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 $\mathbf{c}_t$
- $\tilde{\mathbf{z}}_t$ is then the top hidden layer: the decoded word probability at the target is calculated as *softmax*$(\mathbf{W}_s\tilde{\mathbf{z}}_t)$

Global Attention: What is the Context $\mathbf{c}_t$?

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

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

- The alignment score function calculates a similarity measure: A typical **score** is the dot product, $score(\mathbf{z}_t, \bar{\mathbf{z}}_s) = \mathbf{z}_t^T \bar{\mathbf{z}}_s$; here, $\mathbf{z}_t$ is the query, and $\bar{\mathbf{z}}_s$ is the key
- The already introduced **context vector** is then calculated as (here, $\bar{\mathbf{z}}_s$ on the right assumes the role of the **value**)

$$\mathbf{c}_t = \sum_s \mathit{align}(\mathbf{z}_t, \bar{\mathbf{z}}_s)\ \bar{\mathbf{z}}_s$$

- Note that each component of $\mathbf{c}_t$ is like a post-activation of a neuron
- One defines $\text{Attention}_t = \mathbf{D}\mathbf{c}_t$

From Attention to Self-Attention

- So far we calculated the attention of an element in the output sequence w.r.t all elements in the input sequence
- Let's consider another task, e.g., entity labelling
- We have a sequence of words/entities ($t = 1, 2, ...$) as inputs; the goal is to provide a label for each word/entity, or to provide a label for the whole sequence
- We now change notation: $\mathbf{z}_{t,l}$ is the activation vector at layer l
- Self-attention can be applied to any deep neural network
- Self-attention can replace convolutional and recurrent approaches ("attention is all you need")
- Whereas RNNs work left to right, self-attention (as convolutional NNs) work bottom up, in parallel

Self-Attention (cont'd)

- In self-attention, the activation of a hidden layer $\mathbf{z}_{t,l}$ is calculated based on other layer's $\mathbf{z}_{t,l-1}$ of all entities/data points as

$$\mathbf{z}_{t,l} = \text{sig}\left(\mathbf{V}_l \mathbf{z}_{t,l-1} + \mathbf{D}_l \mathbf{c}_{t,l-1}\right)$$

- Here, the context vector is

$$\mathbf{c}_{t,l-1} = \sum_{t'} \mathit{align}\Big(\mathbf{z}_{t,l-1}, \mathbf{z}_{t',l-1}\Big)\, \mathbf{z}_{t',l-1}$$

  The sum is over all elements in the sequence

- (Often the tanh is used instead of the sig)

- Self-attention can be applied to any layer (not just the top layer)

- Key advantage: memoryless modelling of far-range dependencies (beyond nearest-neighbour interactions)

Comparison

- **Feed forward neural network**

$$\mathbf{z}_{t,l} = \text{sig}\left(\mathbf{V}_l \mathbf{z}_{t,l-1}\right)$$

so here each word label at position t is predicted separately; embeddings are all independent; this is the i.i.d situation

- **Fully connected** (not used in practice)

$$\mathbf{z}_{t,l} = \text{sig}\left(\mathbf{V}_l \mathbf{z}_{t,l-1} + \sum_{t'} \mathbf{C}_{t,t',l} \mathbf{z}_{t',l-1}\right)$$

The embeddings of all words are considered; 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**

$$\mathbf{z}_{t,l} = \text{sig}\left(\mathbf{V}_l \mathbf{z}_{t,l-1} + \sum_k \sum_{t'} \mathbf{C}^k_{t-t',l} \mathbf{z}_{t',l-1}\right)$$

  Very powerful approach and very successful in NLP; needs zero padding at sentence boundaries; k is the index over different filter kernels

- In some approaches (e.g., graph convolution) simply the averages of the neighbor embeddings are calculated

Comparison (cont'd)

- **Recurrent neural networks**

$$\mathbf{z}_{t,l} = \text{sig}\left(\mathbf{V}_l \mathbf{z}_{t,l-1} + \mathbf{B}_l \mathbf{z}_{t-1,l}\right)$$

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

- **Bidirectional recurrent neural networks**

$$\mathbf{z}_{t,l} = [\mathbf{z}^f_{t,l}; \mathbf{z}^b_{t,l}]$$

$$= \left[\text{sig}\left(\mathbf{V}^f_l \mathbf{z}_{t,l-1} + \mathbf{B}^f_l \mathbf{z}^f_{t-1,l}\right); \text{sig}\left(\mathbf{V}^b_l \mathbf{z}_{t,l-1} + \mathbf{B}^b_l \mathbf{z}^b_{t+1,l}\right)\right]$$

Comparison (cont'd)

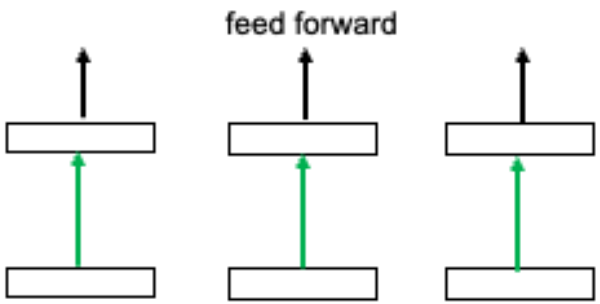
- **Self-Attention**

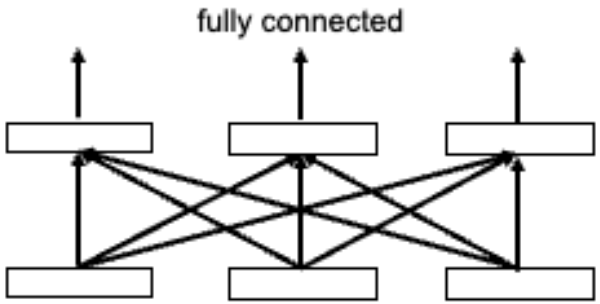
$$\mathbf{z}_{t,l} = \text{sig}\left(\mathbf{V}_l \mathbf{z}_{t,l-1} + \mathbf{D}_l \mathbf{c}_{t,l-1}\right)$$

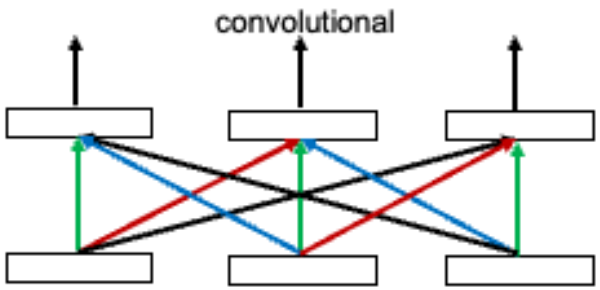
$$\mathbf{c}_{t,l-1} = \sum_{t'} align\left(\mathbf{z}_{t,l-1}, \mathbf{z}_{t',l-1}\right) \mathbf{z}_{t',l-1}$$

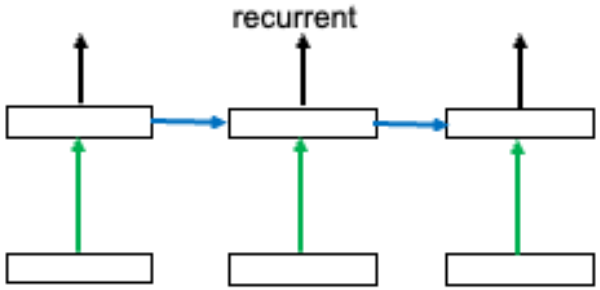
  (the sig is often the tanh) self-attention can replace convolutional or recurrent layers

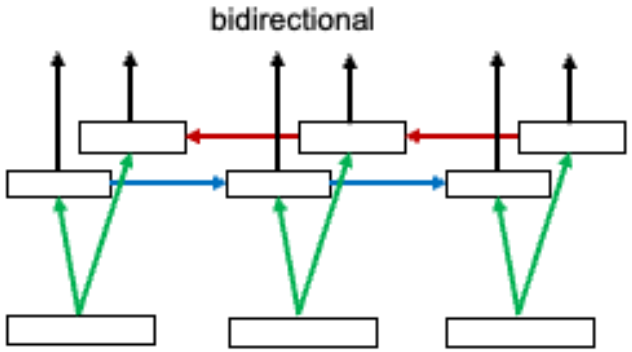
- $align\left(\mathbf{z}_{t,l-1}, \mathbf{z}_{t',l-1}\right)$ is like a (LSTM-like) gate for $\mathbf{D}_l \mathbf{z}_{t',l-1}$

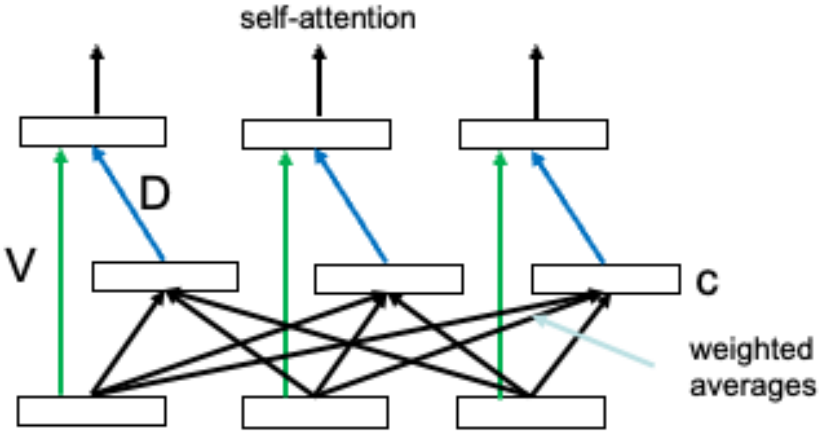

feed forward


fully connected


convolutional


recurrent


bidirectional


self-attention
D
V
C
weighted
averages

Conclusions

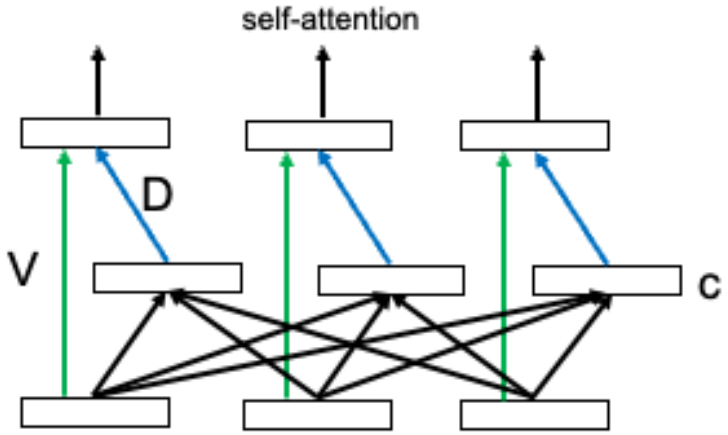
- The global attention has a drawback that it has to attend to all words on the source side for each target word, which is expensive and can potentially render it impractical to translate longer sequences, e.g., paragraphs or documents
- To address this deficiency, a local attentional mechanism has been proposed that chooses to focus only on a small subset of the source positions per target word
- We will discuss positional encoding in the context of the transformer
- Sequential models find many applications in natural language processing (NLP) applications, including machine translation
- Attention capture powerful inductive biases
- Attention mechanisms are the basis for state of the art machine translation (Transformer) and context sensitive embedding models

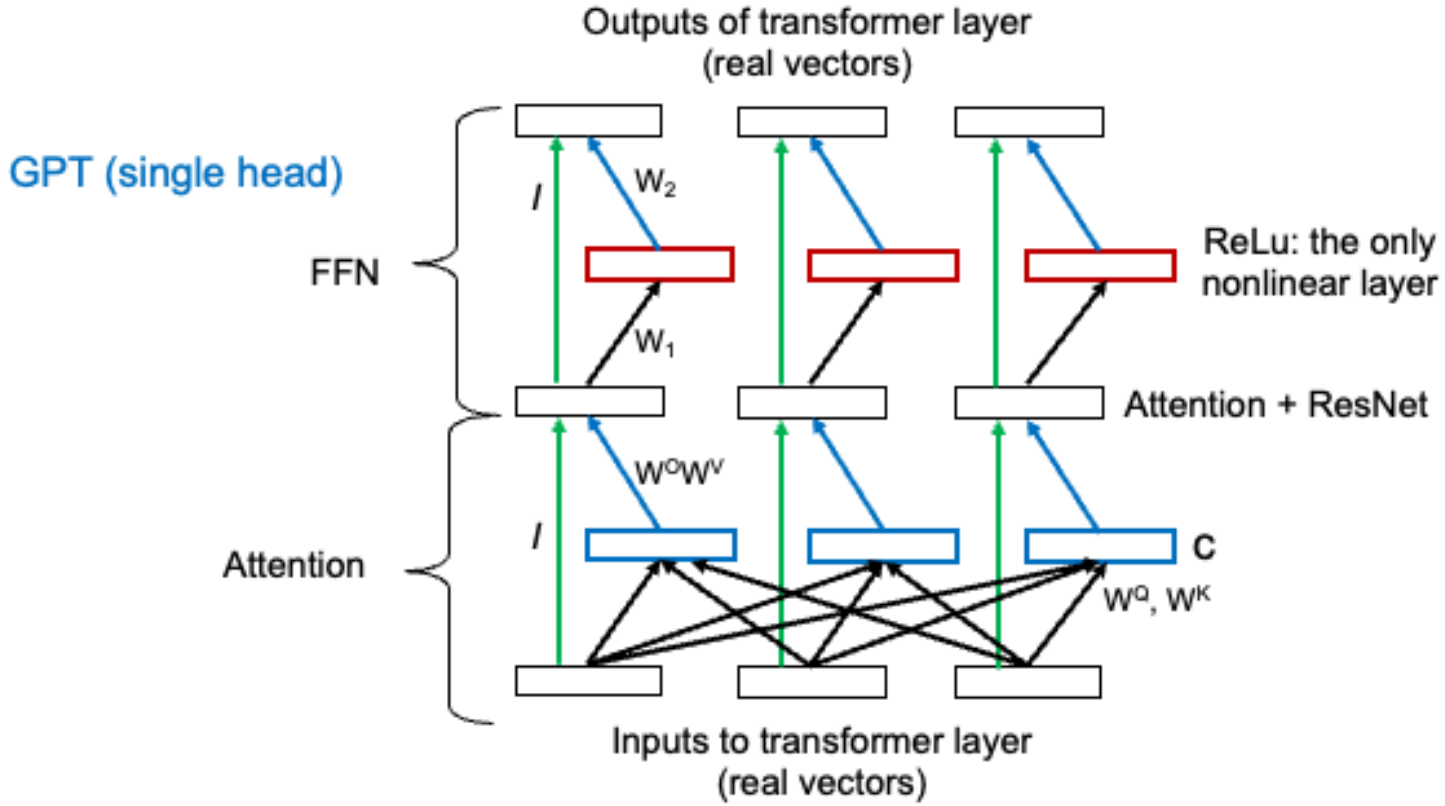
Transformer, BERT, GPT

Transformer Layer

Transformer Layer: Single-Head Attention

- We do not have a layer index: $\mathbf{x}_t$ is the layer and $\tilde{\mathbf{x}}_t$ the embedding at the next layer
- $\mathbf{z}_t = \mathsf{LayerNorm}(\mathbf{x}_t + \mathbf{W}^O \mathbf{W}^V \sum_{t'} \mathsf{align}_{t,t'}\, \mathbf{x}_{t'})$
- $\tilde{\mathbf{x}}_t = \mathsf{LayerNorm}(\mathbf{z}_t + \mathrm{FFN}(\mathbf{z}_t))$
- $\mathsf{align}_{t,t'} = \frac{\exp\left((\mathbf{W}^Q \mathbf{x}_t)^\top (\mathbf{W}^K \mathbf{x}_{t'})/\sqrt{r_{head}}\right)}{\sum_{t'} \exp((\mathbf{W}^Q \mathbf{x}_t)^\top (\mathbf{W}^K \mathbf{x}_{t'})/\sqrt{r_{head}})}$
- $\mathrm{FFN}(\mathbf{z}) = \mathbf{W}_2 \max(0, \mathbf{W}_1 \mathbf{z} + \mathbf{b}_1) + \mathbf{b}_2$
- Note the two ResNets which reduce a vanishing-gradient issue; $\mathrm{FFN}(\mathbf{z})$ is a neural network with one hidden layer and linear output neurons


self-attention
D
V
C


Outputs of transformer layer
(real vectors)
GPT (single head)
FFN
I
W2
W1
ReLu: the only
nonlinear layer
Attention + ResNet
WOWV
I
Attention
C
WQ, WK
Inputs to transformer layer
(real vectors)

Transformer Layer: Single-Head Attention: Dimensions

- All vectors are "embedding vectors" (pre-activation vectors): $\mathbf{z}_t, \mathbf{x}_t, \tilde{\mathbf{x}}_t \in \mathbb{R}^r$
- Layer normalization normalizes an embedding vector such that all entries in an embedding vector have zero mean and unit variance
- The $\mathbf{W}^Q, \mathbf{W}^K \in \mathbb{R}^{r_{head} \times r}$, $\mathbf{W}^V \in \mathbb{R}^{r_v \times r}$ are projection matrices; $\mathbf{W}^O \in \mathbb{R}^{r \times r_v}$; with $r_v < r$ and $r_{head} < r$, we get dimensionality reduction
- Having both $\mathbf{W}^O$ and $\mathbf{W}^V$ seems redundant: but with $r_v < r$ we get a dimensionality reduction! Having both $\mathbf{W}^Q$ and $\mathbf{W}^K$ seems redundant, as well (for dot-product attention): $(\mathbf{W}^Q \mathbf{x}_t)^\top (\mathbf{W}^K \mathbf{x}_{t'}) = \mathbf{x}_t^\top ((\mathbf{W}^Q)^\top \mathbf{W}^K) \mathbf{x}_{t'}$
- The projection matrices are shared in the same layer but are different in different layers
- Note: in the literature, all vectors are row vectors and all matrices are transposed!

Transformer Layer: Multi-Head Attention

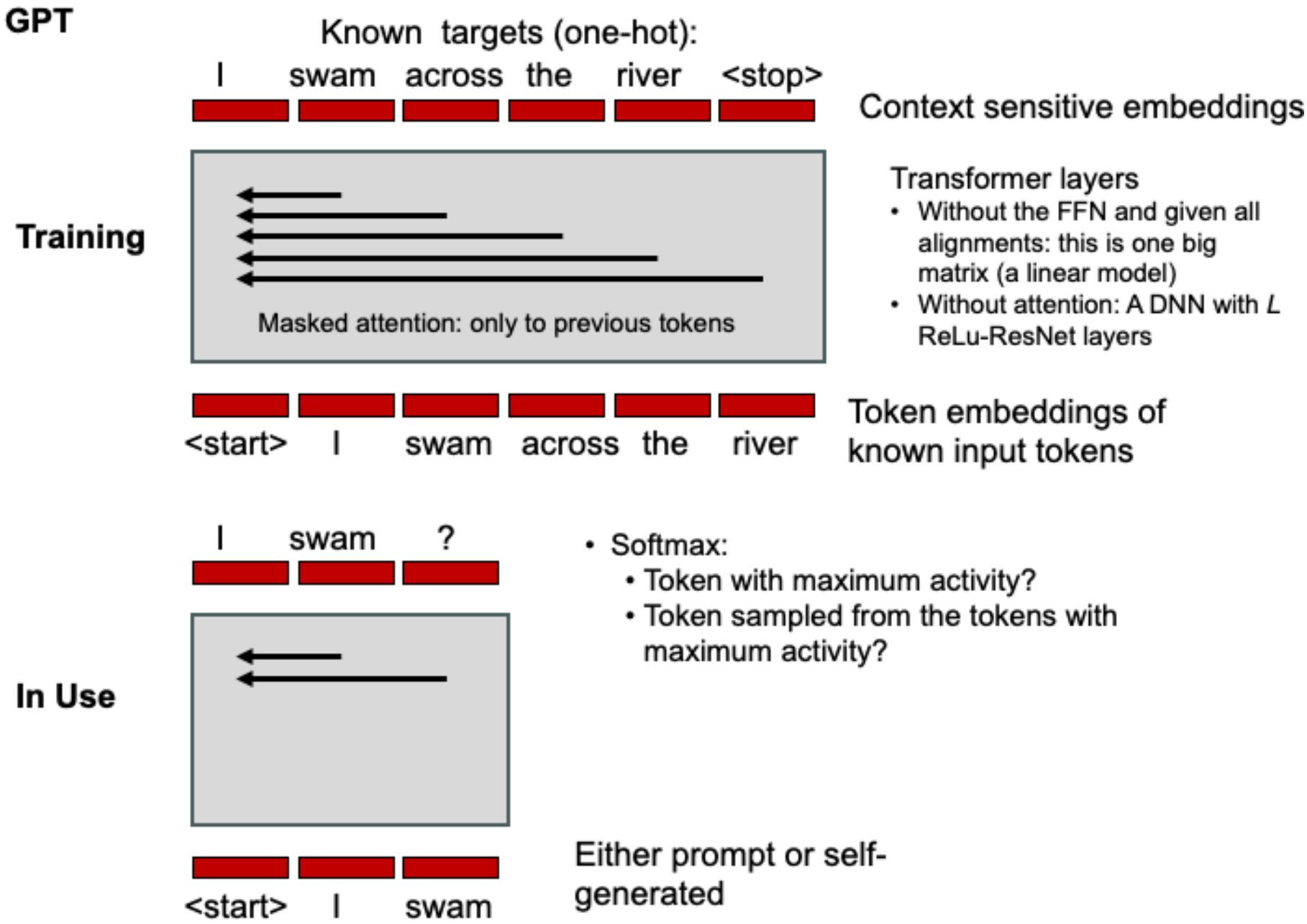
- We do not have a layer index: $\mathbf{x}_t$ is the layer and $\tilde{\mathbf{x}}_{t'}$ the embedding at the next layer
- $\mathbf{z}_t = \text{LayerNorm}(\mathbf{x}_t + \sum_{k=1}^{K} \mathbf{W}^{O,k}\mathbf{W}^{V,k} \sum_{t'} \text{align}_{t,t',k}\ \mathbf{x}_{t'})$
- $\tilde{\mathbf{x}}_{t'} = \text{LayerNorm}(\mathbf{z}_t + \text{FFN}(\mathbf{z}_t))$
- $\text{align}_{t,t',k} = \frac{\exp\left((\mathbf{W}^{Q,k}\mathbf{x}_t)^\top(\mathbf{W}^{K,k}\mathbf{x}_{t'})/\sqrt{r_{head}}\right)}{\sum_{t'}\exp((\mathbf{W}^{Q,k}\mathbf{x}_t)^\top(\mathbf{W}^{K,k}\mathbf{x}_{t'})/\sqrt{r_{head}})}$
- In the original publication, $r = 512, r_{head}, r_v = 64$;nNumber of heads: $K = 8$; number of transformer layers: $L = 6$
- Computational cost in use per layer: $\mathcal{O}(T^2 r)$ (attention layer) and $\mathcal{O}(Tr^2)$ for the FNN; T is the sequence length (number of columns)

GPT: Decoder Only

GPT

- Generative Pre-trained Transformer (GPT)
- It is decoder-only
- It is initialized with the prompt; a prompt can be a question, a task, ...; it is the initialization (prompt engineering)
- GPT-1 to GPT-3: main difference are: different forms of training schedules!
- GPT-4 can handle sequences of 32k tokens (under development: sequences of 100k or more tokens)
- Tokenization (byte pair encoding): Typically, most words will be encoded as a single token, while rare words will be encoded as a sequence of a few tokens, where these tokens represent meaningful word parts. This translation of text into tokens can be found by variants of byte pair encoding, such as subword units

GPT


Known targets (one-hot):
I swam across the river <stop>
Context sensitive embeddings
Training
Transformer layers
• Without the FFN and given all alignments: this is one big matrix (a linear model)
• Without attention: A DNN with L ReLu-ResNet layers
Masked attention: only to previous tokens
<start> I swam across the river
Token embeddings of known input tokens
I swam ?
• Softmax:
• Token with maximum activity?
• Token sampled from the tokens with maximum activity?
In Use
<start> I swam
Either prompt or self-generated

ChatGPT Training

- Training phases:
- 1: Pretraining (99% of effort); LLM training; leads to **base model**
- Data: internet crawls (CommonCrawl, C4), Github, Wikipedia, Books, ArXiv, StackExchange
- 2: Supervised fine-tuning; with ideal prompt-response pairs (the responses can be quite long); LLM model; leads to **SFT model**
- 3: Reward modeling; human annotators who rank different responses (completions); predicts (subjective) quality; leads to **RM model**
- 4: Reinforcement Learning; used RM; leads to RL model (RLHF: Reinforcement Learning with Human Feedback); improves the LLM further but difficult to get to work

GPT Research

- Research issues: fine-tuning, prompt engineering, providing access to external data like databases (vector databases, database queries)
- At this stage: increasing the number of parameters is not the most important factor; training set size/quality is currently more in focus
- Hallucination: An inherent feature of generative models?

GPT

word, but also to continue on writing whole paragraphs in the style of the initial prompt. For example, when I gave GPT-3 the prompt:

In 1958, Frank Rosenblatt introduced the Perceptron, an early version of a neural net that could only detect the location of a square on a printed card. Fifty years later, neural nets are capable of . . .

The software dutifully completed the thought, and then continued on, picking up on the historical framing of the initial text:

. . . detecting and classifying faces, finding patterns in data, and emulating the human brain. While the neural net matured in academia, it also found its way into the tech industry. In the late 1980s and early 90s, neural nets were used in speech recognition and character recognition applications. The growth in computing power and the ability to collect larger data sets allowed other neural net applications to emerge.

New York Times, 2022-04-22

GPT 175B

- $r = 12288$, $r_{head} = r_v = 128$; $L = 96$ layers; number of heads $K = 96$; number of tokens ("words") $N^{tokens} = 50257$; size of the hidden layer in the FFN: $12288 \times 4 = 49152$ (large!)
- $\mathbf{W}^{O,k}$, $\mathbf{W}^{Q,k}$, $\mathbf{W}^{K,k}$, $\mathbf{W}^{V,k}$: Each of them has 128×12288 parameters, and we have 96×96 (number of heads times number of layers) of them: **58B**
- $\mathbf{W}_1$ and $\mathbf{W}_2$ in the FFN: $2 \times 12288 \times 49152$ and we have 96 of them (one per layer): **116B**
- Token embedding: $50257 \times 12299 =$ **0.6B** (number of token types times rank)
- This is together: **173.6B**; we need to add the biases, other parameters to get to **175B**

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or "GPT-3"	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

Table 2.1: Sizes, architectures, and learning hyper-parameters (batch size in tokens and learning rate) of the models which we trained. All models were trained for a total of 300 billion tokens.

Then they say:

> Table 2.1 shows the sizes and architectures of our 8 models. Here nparams is the total number of trainable parameters, nlayers is the total number of layers, dmodel is the number of units in each bottleneck layer (we always have the feedforward layer four times the size of the bottleneck layer, dff = 4 ∗ dmodel), and dhead is the dimension of each attention head. All models use a context window of nctx = 2048 tokens. We partition the model across GPUs along both the depth and width dimension in order to minimize data-transfer between nodes. The precise architectural parameters for each model are chosen based on computational efficiency and load-balancing in the layout of models across GPU's. Previous work [KMH+20] suggests that validation loss is not strongly sensitive to these parameters within a reasonably broad range.

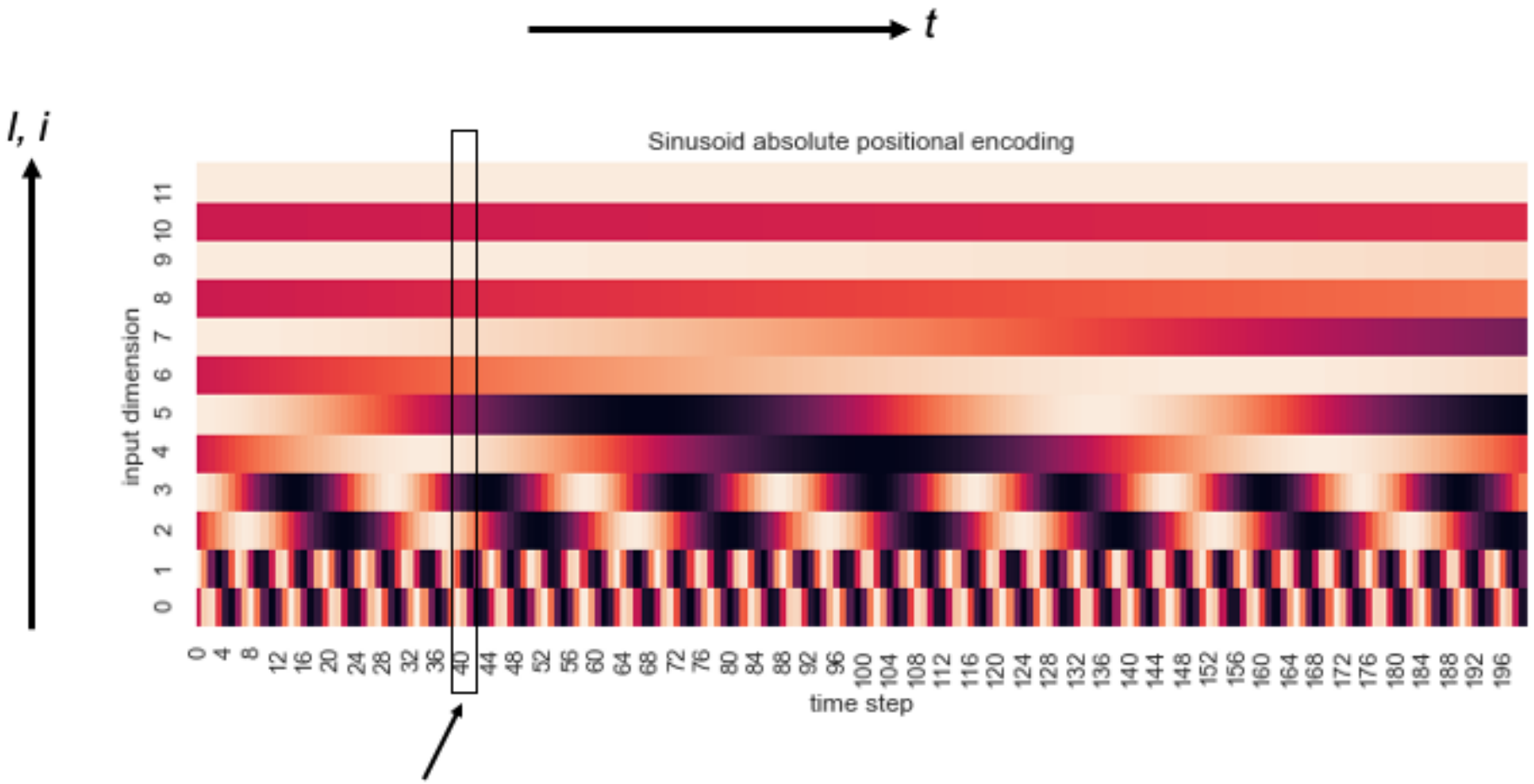
Position Encoding

- Consider that t is the position in the sequence; $PosEnc(t, :)$ is the position encoding vector, and $PosEnc(t, l)$ with $l = 1, \ldots, r$ is its l-th component
- We have, for $i = 0, \ldots, r/2$

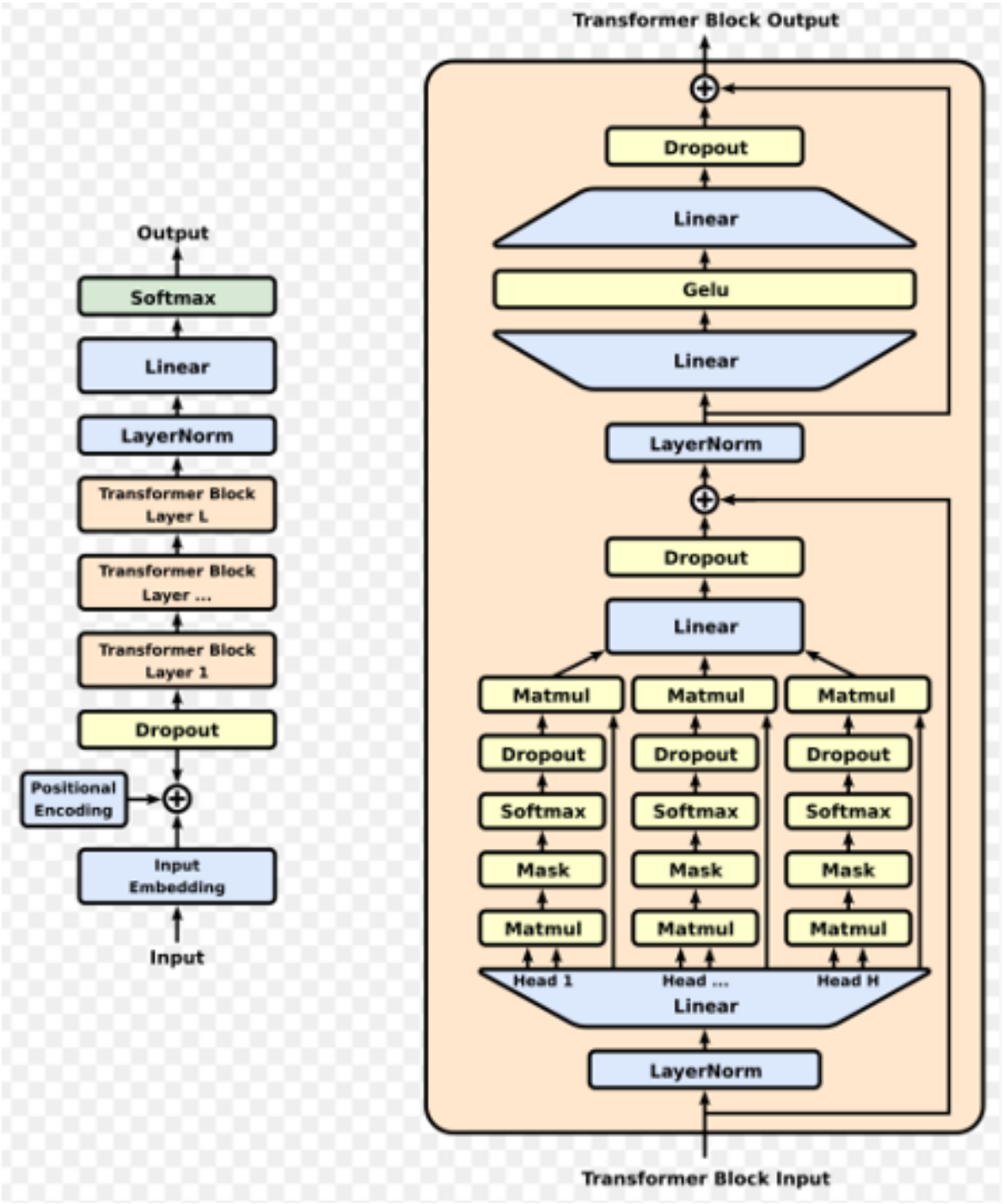
$$PosEnc(t, 2i) = \sin((t-1)/n^{2i/r})$$

$$PosEnc(t, 2i+1) = \cos((t-1)/n^{2i/r})$$

- Increasing t (left to right), we sample a sine/cosine wave pattern
- Increasing i (or i), the frequency decreases, from 1 to $1/n$
- n is a user specified parameter (often $n = 10000$)


t
l, i
Sinusoid absolute positional encoding
input dimension
0 1 2 3 4 5 6 7 8 9 10 11
0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80 84 88 92 96 100 104 108 112 116 120 124 128 132 136 140 144 148 152 156 160 164 168 172 176 180 184 188 192 196
time step
Position encoding (position embedding vector) for position t= 40

GPT


Output
Softmax
Linear
LayerNorm
Transformer Block Layer L
Transformer Block Layer ...
Transformer Block Layer 1
Dropout
Positional Encoding
Input Embedding
Input
Transformer Block Output
Dropout
Linear
Gelu
Linear
LayerNorm
Dropout
Linear
Matmul
Matmul
Matmul
Dropout
Dropout
Dropout
Softmax
Softmax
Softmax
Mask
Mask
Mask
Matmul
Matmul
Matmul
Head 1
Head ...
Head H
Linear
LayerNorm
Transformer Block Input

Making it Work in Practice

- Fine-tuning / model editing: **Instruction fine-tuning** is a process in the training or fine-tuning of large language models (LLMs) where the model is specifically trained to better understand and follow natural language instructions; Instruction tuning uses datasets consisting of instruction-response pairs
- LoRA (Low-Rank Adaptation): Take any weight matrix W and add in parallel an Autoencoder layer: adapt only that one for a new data set (replaces the adaptation of the last layers) LORA is a form of **adaptation or fine-tuning** used in **transfer learning**
- Mixture of Experts (MoE): specialized sub-LLMs, each one focusing on a top category. If you bundle 2000 of them together, you cover the entire human knowledge. The whole system, sometimes called mixture of experts, is managed with an LLM router
- MAMBA, xLSTM: replace attention with RNNs to reduce computational cost per layer to $(O)(Tr^2)$ and memory footprint to $\mathcal{O}(r^2)$ per layer

Prompt Engineering

- **Prefix prompting**: before entering the task/question provide additional instructions: "no offensive language"
- **In context learning** (ICL):
- Few-shot learning: Provide some QA examples in the prompt (ICLR)
- One-shot learning: Provide one QA example in the prompt (ICLR)
- Zero-shot learning: no examples are given
- **Chain of Thought (CoT)** prompting is a technique used with large language models (LLMs) to improve their reasoning capabilities by encouraging them to break down complex problems into intermediate steps, instead of directly generating an answer. This approach leverages the model's ability to process and reason through multi-step problems more effectively by following a structured, step-by-step reasoning path
- **Long-context methods** in large language models (LLMs) refer to techniques and approaches designed to extend the model's ability to handle and reason over significantly larger input texts or contexts than its default capacity.

Retrieval-Augmented Generation (RAG)

- Retrieval-Augmented Generation (RAG) is a hybrid approach used in large language models (LLMs) to improve their ability to generate accurate and relevant responses by integrating an external retrieval mechanism with the model's generative capabilities. It combines the strengths of retrieval systems and generative models to address challenges like knowledge cutoff, hallucinations, and domain-specific queries
- Sparse retrieval methods (e.g., TF-IDF, BM25).
- Dense retrieval methods: in a preprocessing step, embeddings are generated for the documents; the prompt also generates an embedding vector and similarity between prompt embedding and document embeddings are used in retrieval
- The retrieved documents are combined with the original query (appended) to create a context for the generative model
- New: MemoryLLM: combines static parameters with dynamic parameters representing new knwoledge

GPU

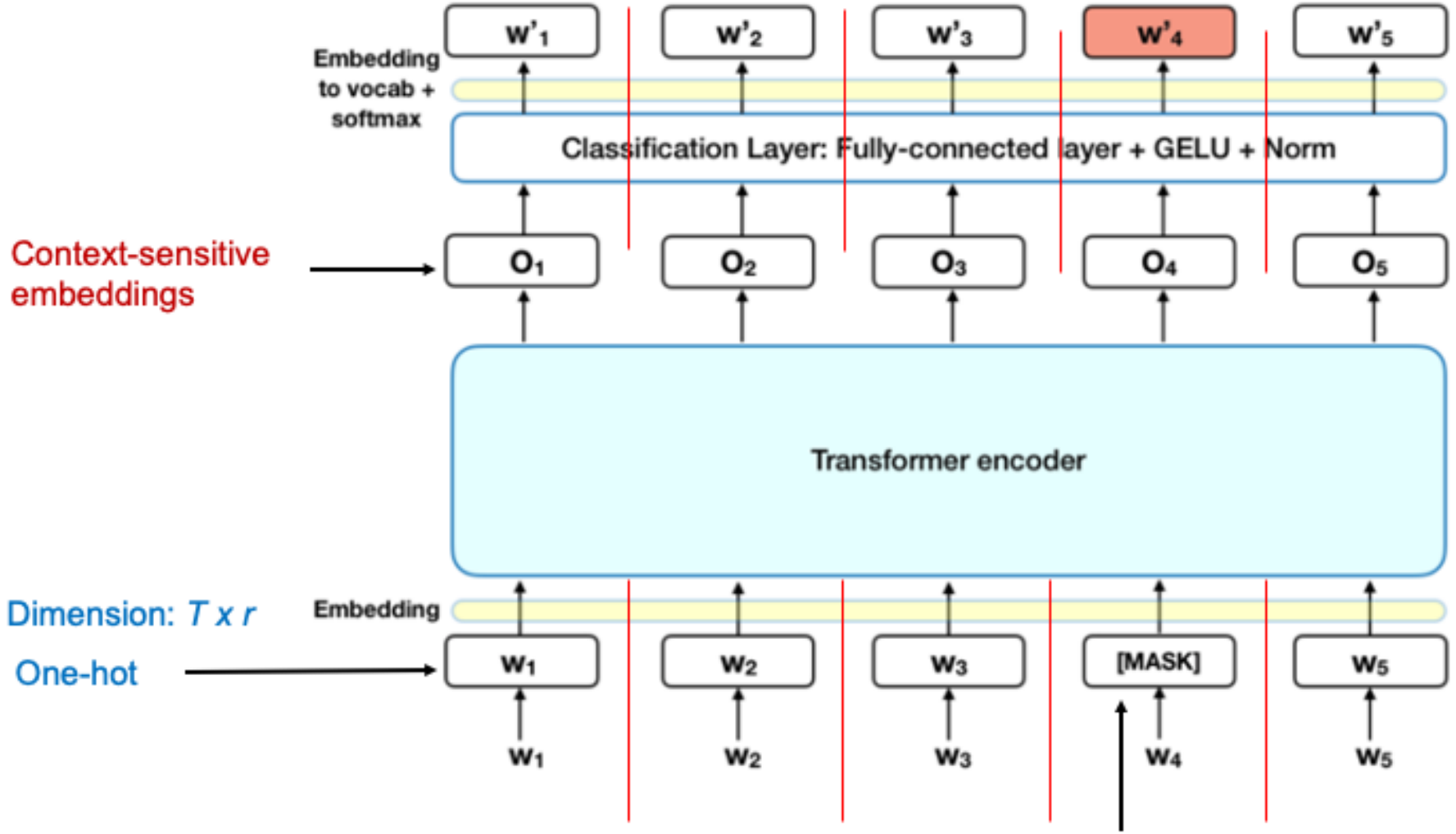
- Computations can be executed in parallel and map well to GPUs (in contrast to RNN, LSTM)

BERT: Encoder Only

BERT

- **BERT** (Bidirectional Encoder Representations from Transformers) from Google leverages attention mechanism and transformer to learn word contextual relations using a masked language model (MLM)
- It is an encoder-only model (it is not a generative model)

BERT: Masked LM

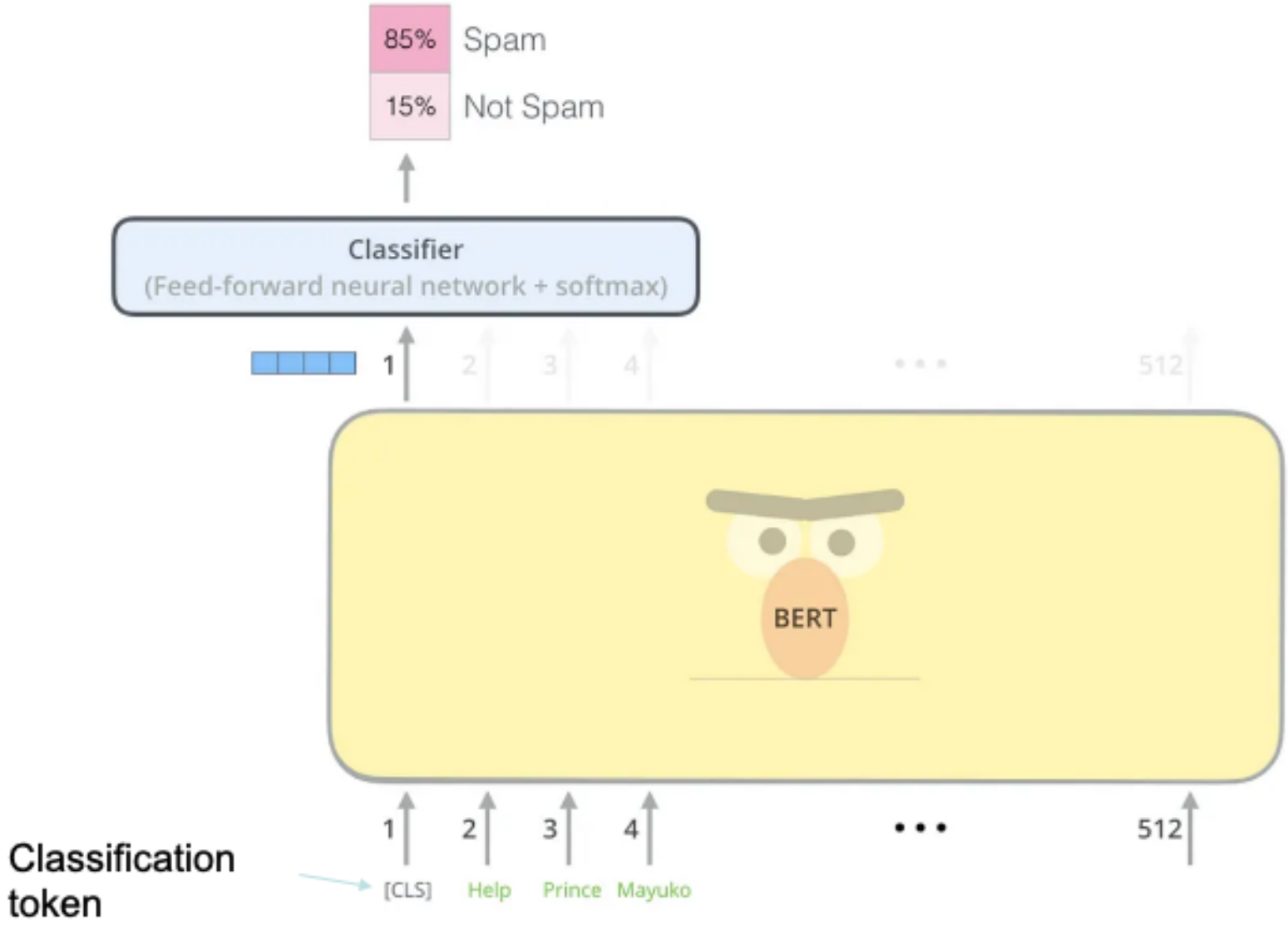


towardsdatascience

BERT

- BERT is almost an auto encoder: Inputs and outputs are identical
- But some tokens of the input sentence are removed (masked) and the network is trained to predict the correpsonding tokens at the output layer
- Masked language modelling (MLM)
- The context-sensitive word embeddings can be used for all sorts of tasks, like world labelling, NER, ...
- BERT is a pre-trained model which can be used for different purposes (fine-tuning): for example an additional column might be generated where the output reflects the class of the sentence. The input to that column is the token $< CLS >$

Fine-tuning Bert to Perform Sentence Classification



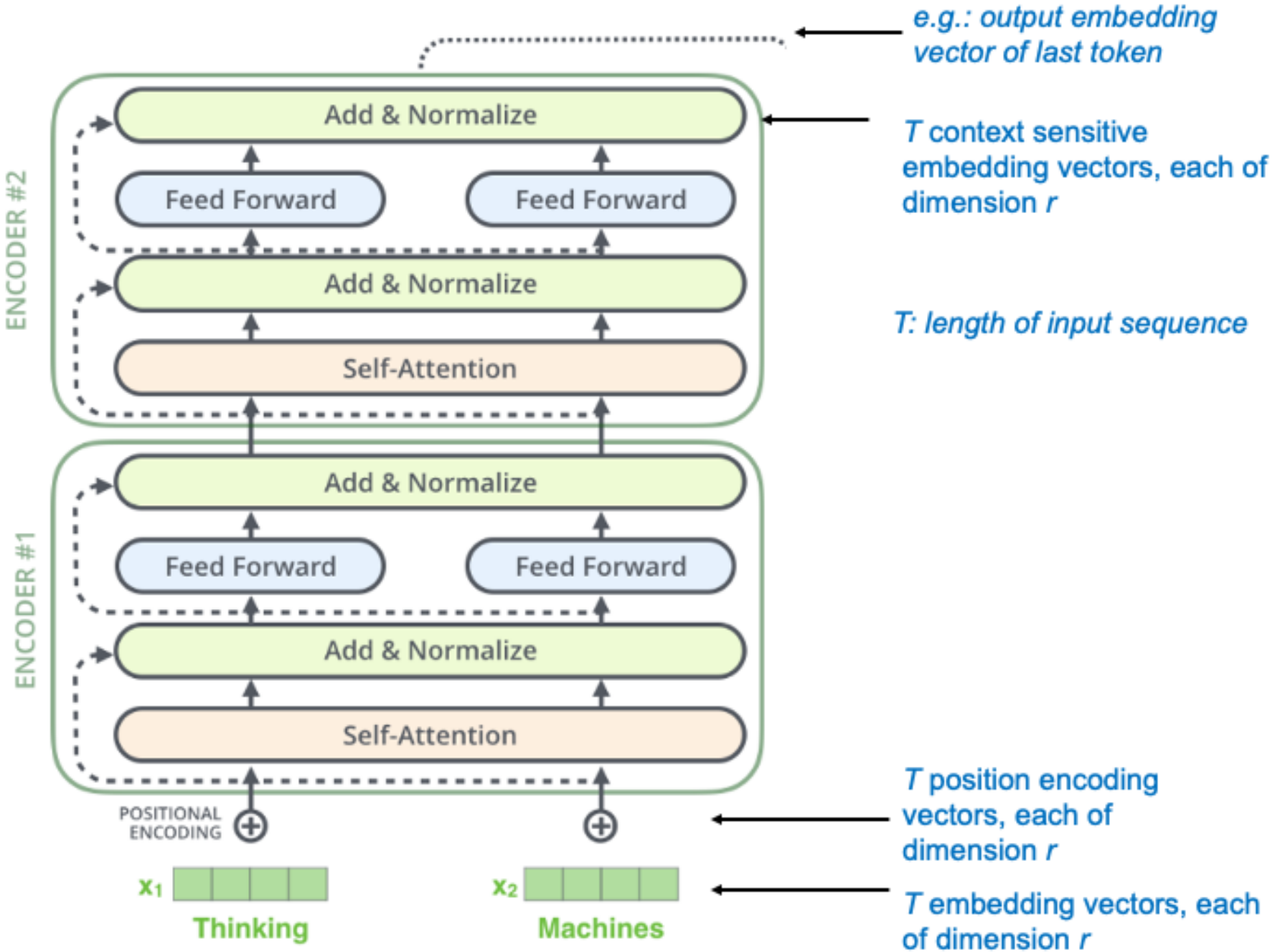
Transformer: Sequence-to-Sequence

Transformer

- This is the first transformer model and was developed for neural machine translation (NMT)
- Bottleneck of previous approaches in NMT using LSTMs: 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

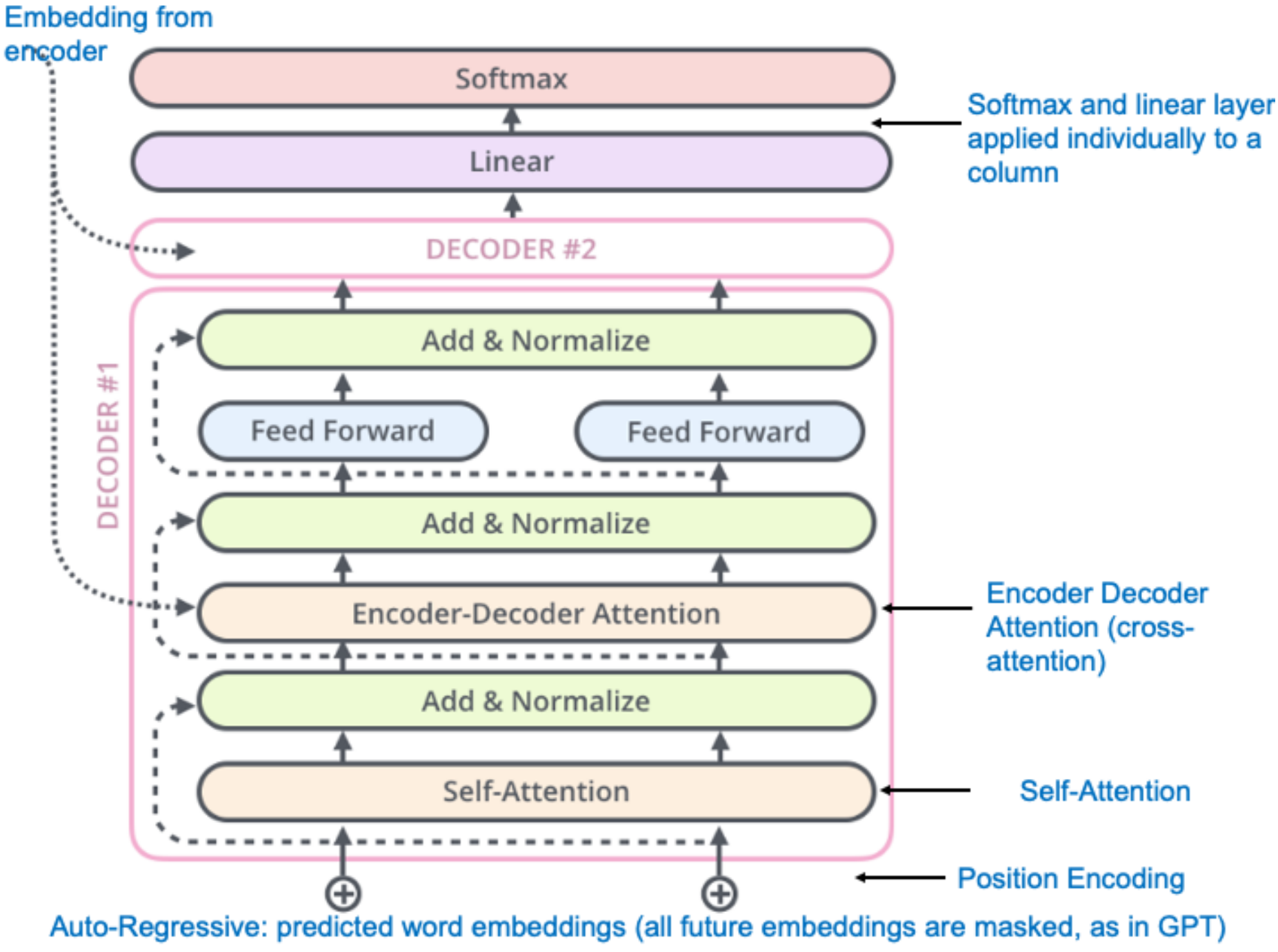
Encoder (BERT)

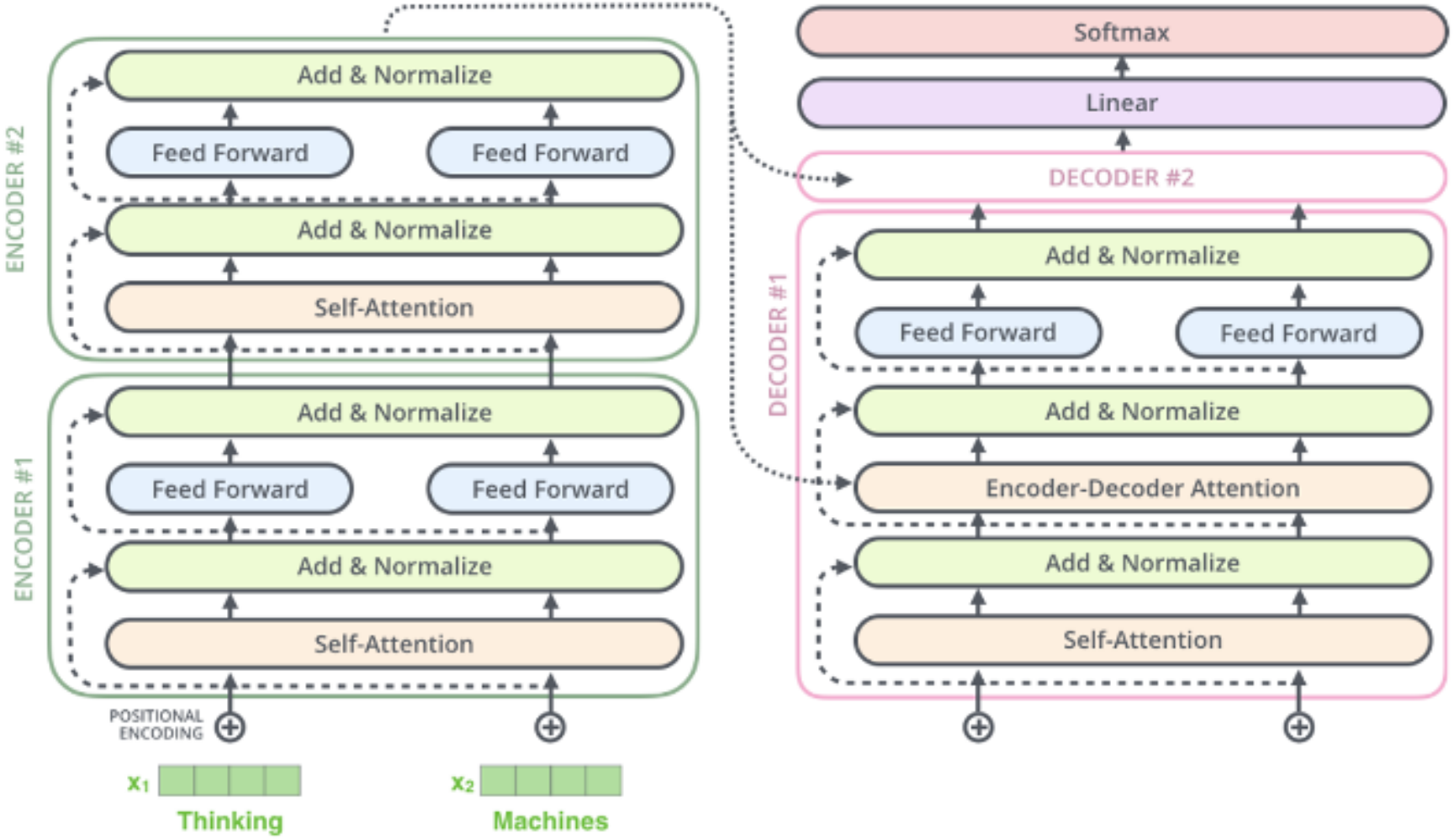
- The encoder uses **self-attention**
- No masking is required

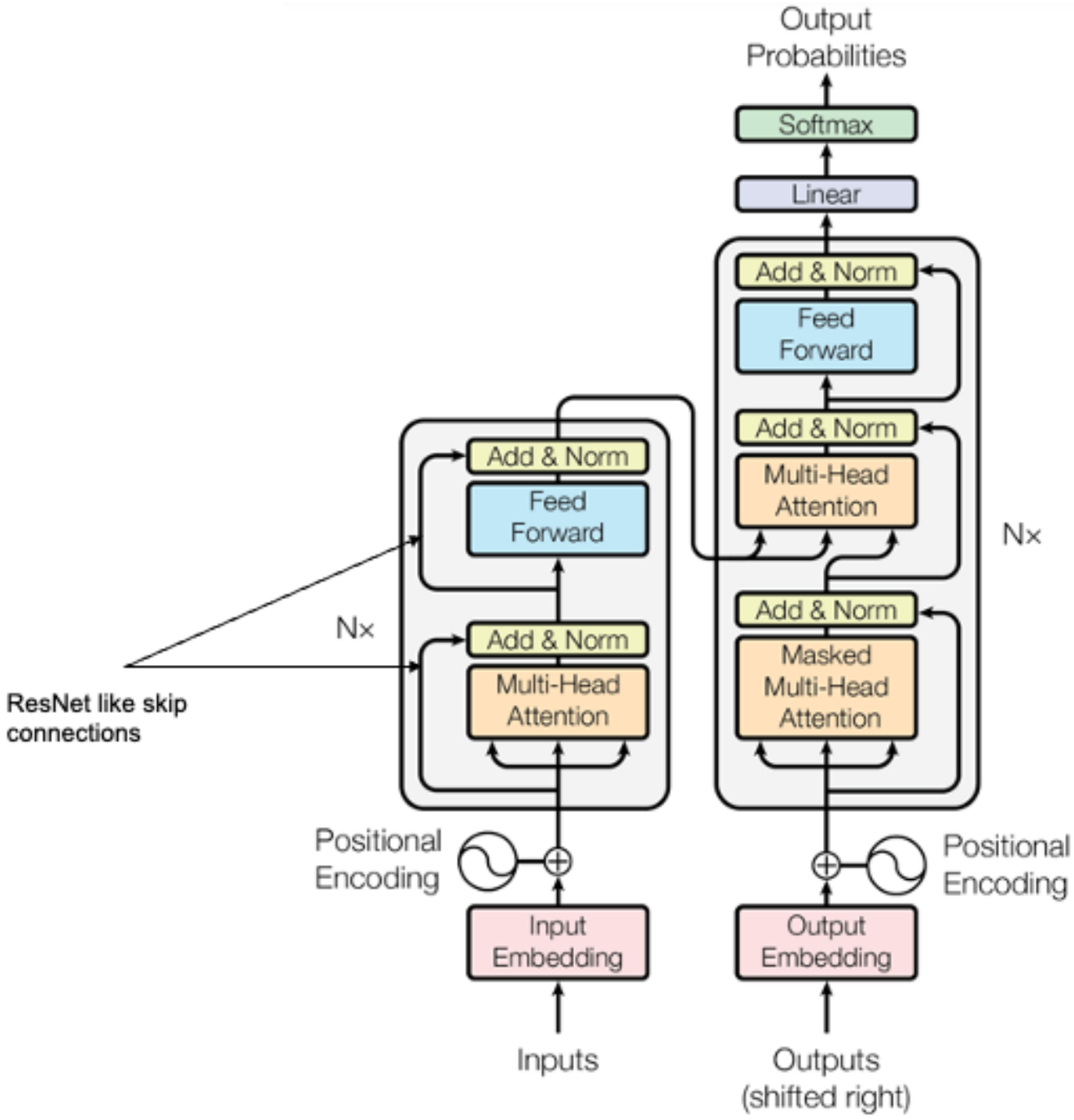

e.g.: output embedding vector of last token
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
ENCODER #2
T context sensitive embedding vectors, each of dimension r
T: length of input sequence
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Self-Attention
ENCODER #1
POSITIONAL ENCODING
T position encoding vectors, each of dimension r
x1
x2
Thinking
Machines
T embedding vectors, each of dimension r

Decoder (GPT)

- The decoder uses both **self-attention** and **cross-attention (encode-decoder attention)**


Embedding from encoder
Softmax
Linear
Softmax and linear layer applied individually to a column
DECODER #2
DECODER #1
Add & Normalize
Feed Forward
Feed Forward
Add & Normalize
Encoder-Decoder Attention
Encoder Decoder Attention (cross-attention)
Add & Normalize
Self-Attention
Self-Attention
Position Encoding
Auto-Regressive: predicted word embeddings (all future embeddings are masked, as in GPT)


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


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
ResNet like skip connections
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

Beyond NLP

- Vision Transformer (ViT): a token is a 16x16x3 image patch (3 colors); by transforming this tensor into a vector one can obtain the embedding vector of the patch;
- Alternative: (1) its embedding vector of dimension r is generated by a simple linear map (2): dimensionality reduction by a convolutional layer
- Multimodal Transformers: Language and Images
- Embodied AI (Robotics) and Multiagent Frameworks / World models: Palm-E (PaLM (Pathways Language Model)-embodied, Google), Generalist AI Model (RT-X), Genie
- AlphaFold (Nobel Price)

Vision Transformer