Recurrent Neural Network

DEEP LEARNING COURSE

E. FATEMIZADEH, FALL 2023

Sharif University of Technology, School of Electrical Engineering

RNN Introduction

DEEP LEARNING, FALL 2023

SHARIF UNIVERSITY OF TECHNOLOGY

Brief

- Till now: Static mapping from input to output
  - Feedforward calculation
- RNN: Temporal/Dynamic Systems
  - Feedback Connection
- Good for sequential data analysis and prediction

Application

- Time series prediction/forecasting
- Natural Language Processing
- Speech/Signal Processing
- Question/Answering Machine
- Image Captioning
- ….

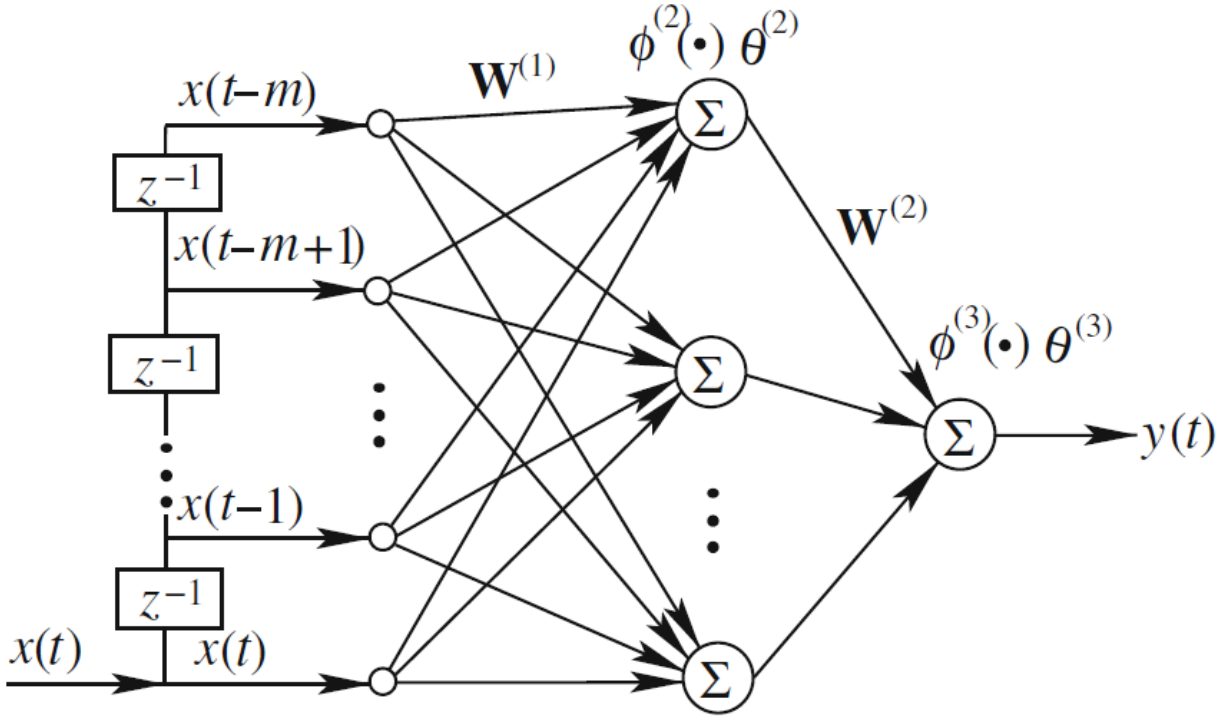
General Model

- Again an old fashion idea!
  - Start with a nonlinear dynamical system
- Nonlinear Auto Regressive eXogenous model (NARX):
  - y: output
  - x: input
  - n: time sample

$$y_{n+1} = F\left(y_n, y_{n-1}, \cdots; x_n, x_{n-1}, \cdots\right)$$

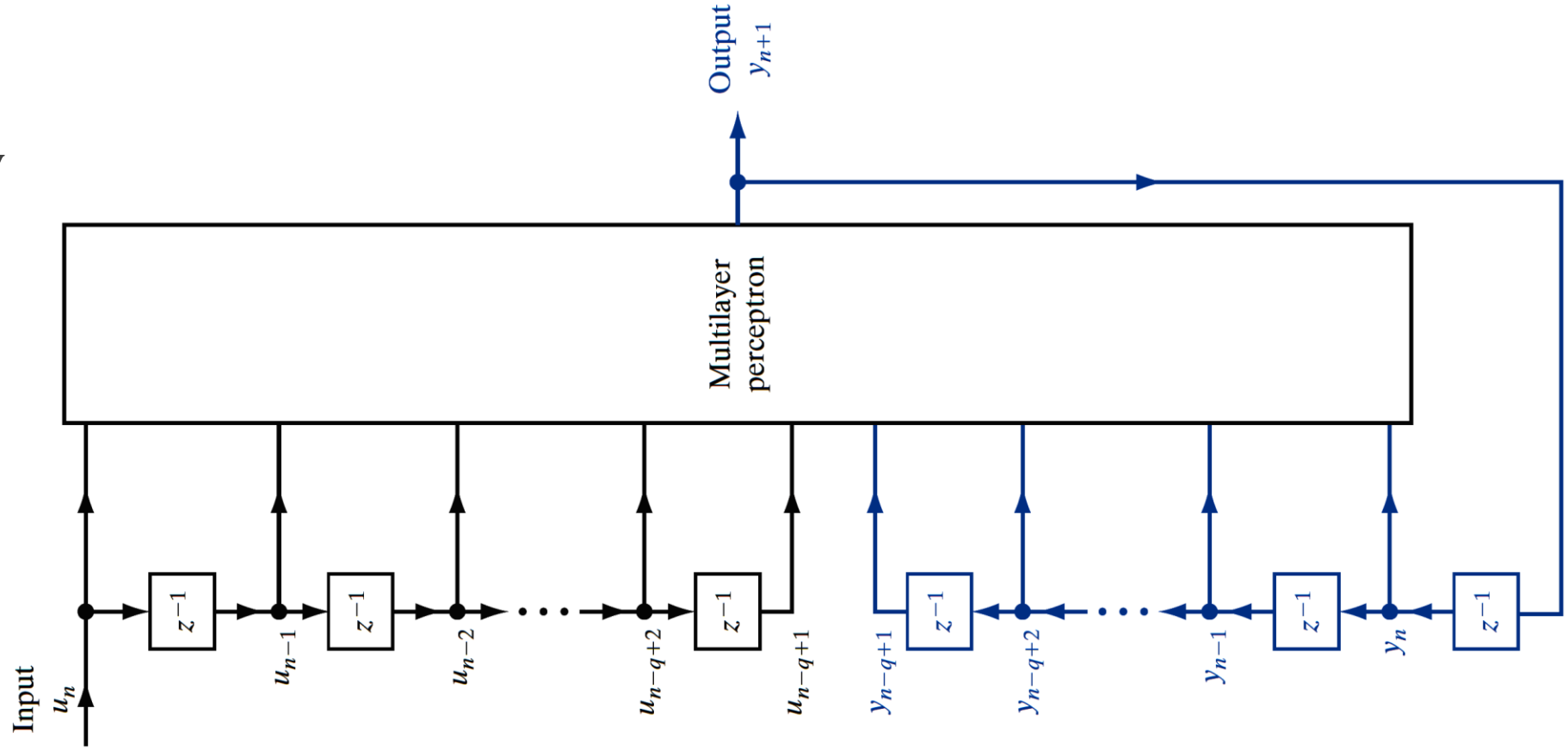
Time-Delayed Neural Network (TDNN)

- Time-Delayed Neural Network, TDNN (for dynamic system):



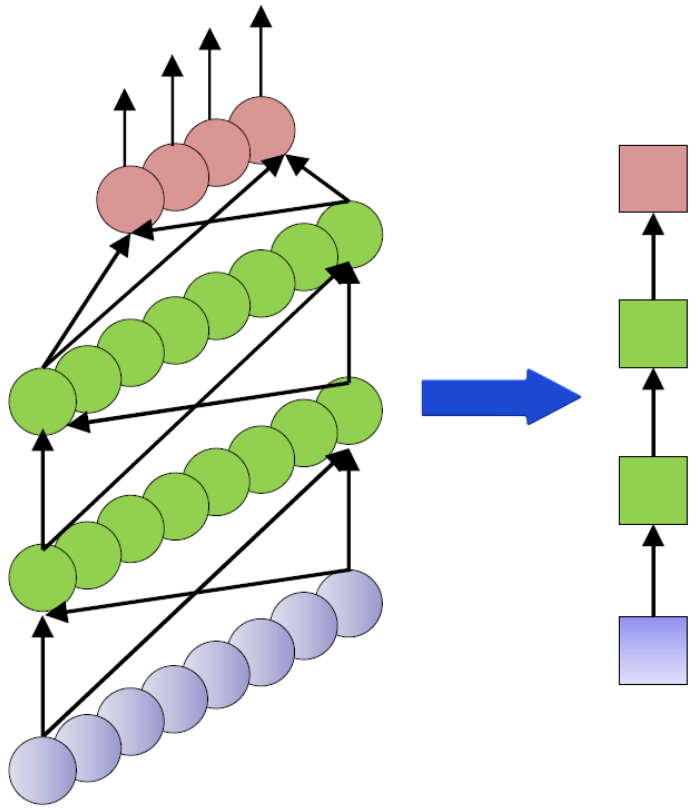
Nonlinear AutoRegressive network with eXogenous inputs (NARX)

- y: output
- u: input
- z^{-1}: time delay



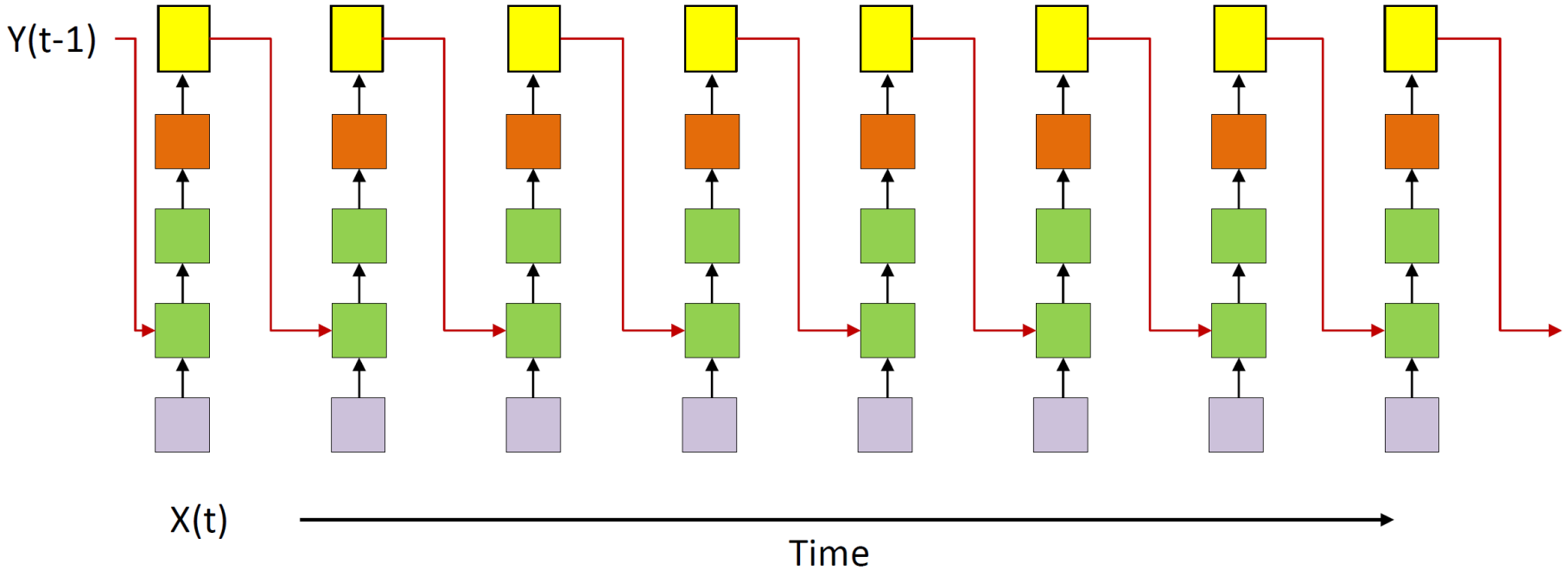
NARX Signal Flow

- Graphic notation:
  - Input at each time is a *vector*
  - Each layer has *many* neurons
  - Output layer too may have *many* neurons
  - Everything by simple boxes
  - Each box actually represents an entire layer with *many* units



NARX Signal Flow

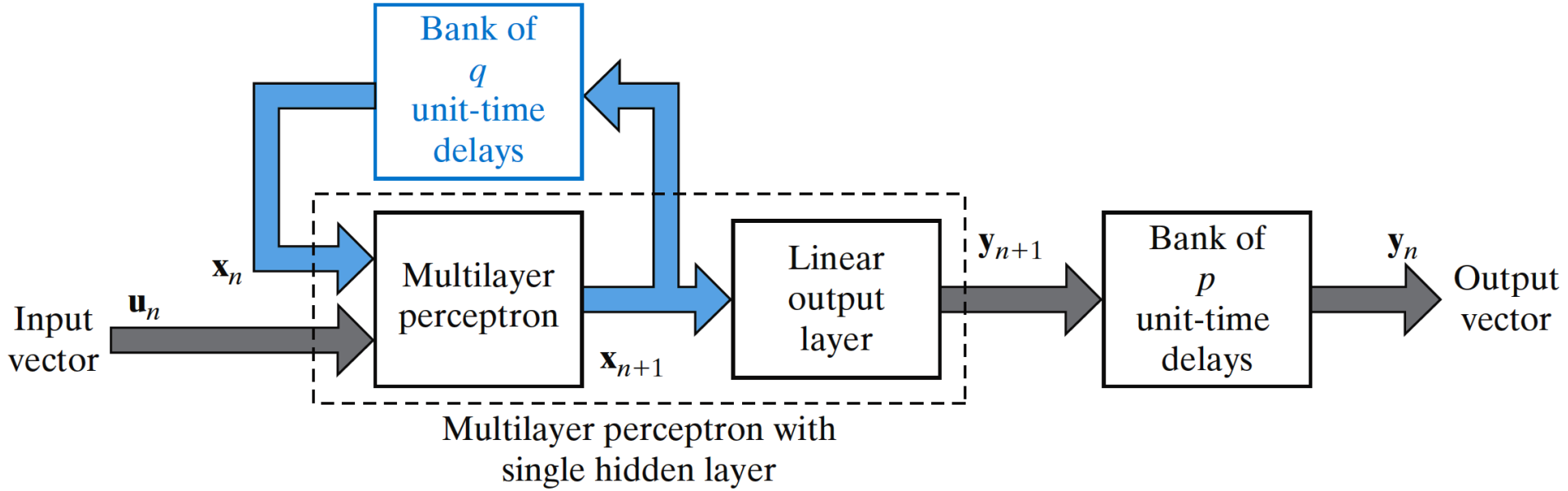
- All columns are identical
- NARX *memory* of the past is in the *output* itself, and *not* in the *network*



State-Space Model

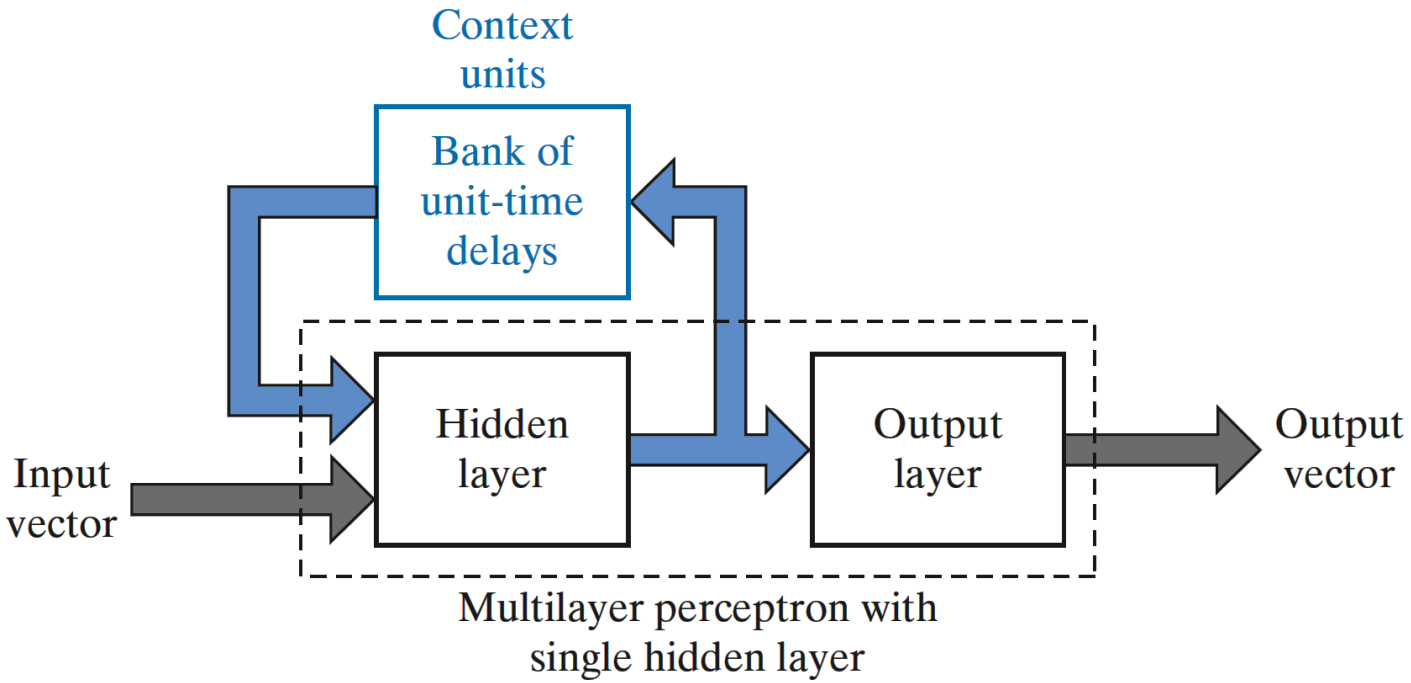
- $\mathbf{U} \in \mathbb{R}^m$: input
- $\mathbf{X} \in \mathbb{R}^q$: state (output of hidden neuron)
- $\mathbf{Y} \in \mathbb{R}^p$: output

$$\mathbf{X}_{n+1} = \mathbf{F}(\mathbf{X}_n, \mathbf{U}_n)$$
$$\mathbf{Y}_n = \mathbf{B}\mathbf{X}_n$$



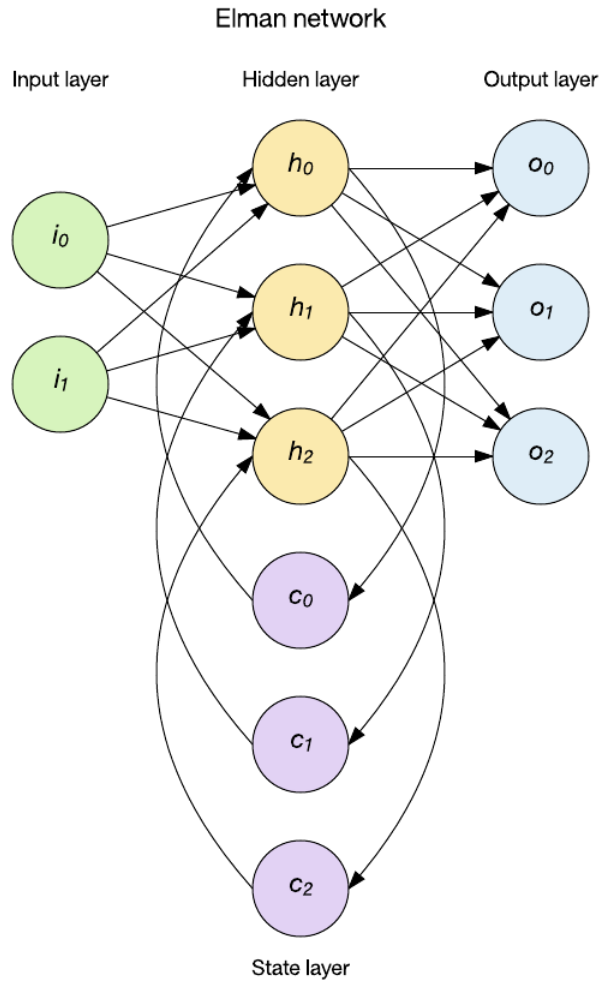
Simple Recurrent Network (SRN)

- Architecture:



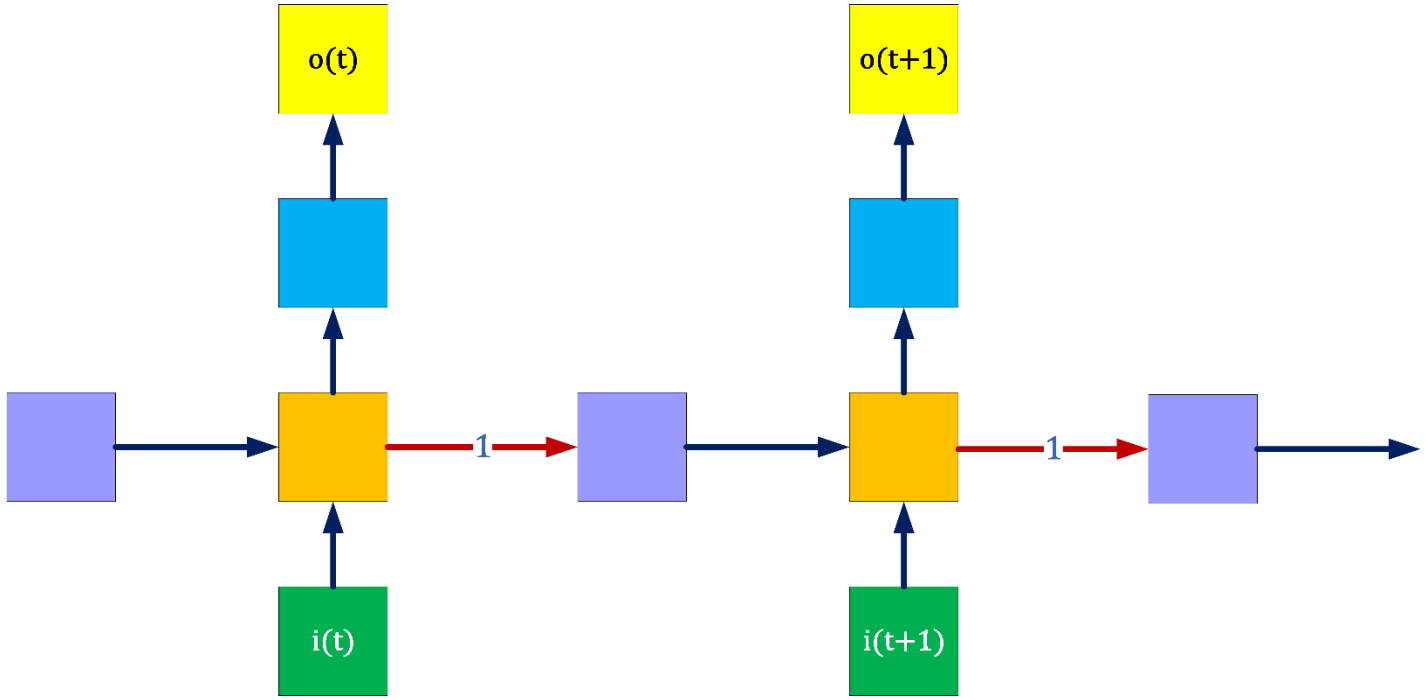
Elman (1990) Network

- Equations:
  - $\mathbf{h}_t = \boldsymbol{\sigma}_h(\mathbf{W}_i\mathbf{i}_t + \mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{b}_h)$
  - $\mathbf{o}_t = \boldsymbol{\sigma}_o(\mathbf{W}_o\mathbf{h}_t + \mathbf{b}_o)$
  - Context layer: unit-time delays



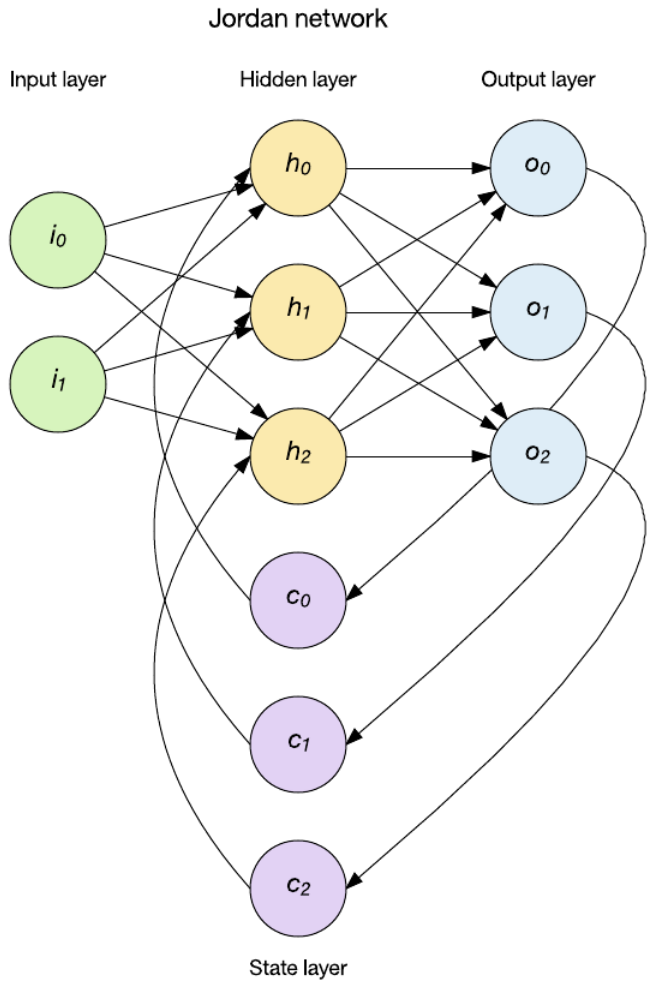
Elman (1990) Network

- Signal Flows



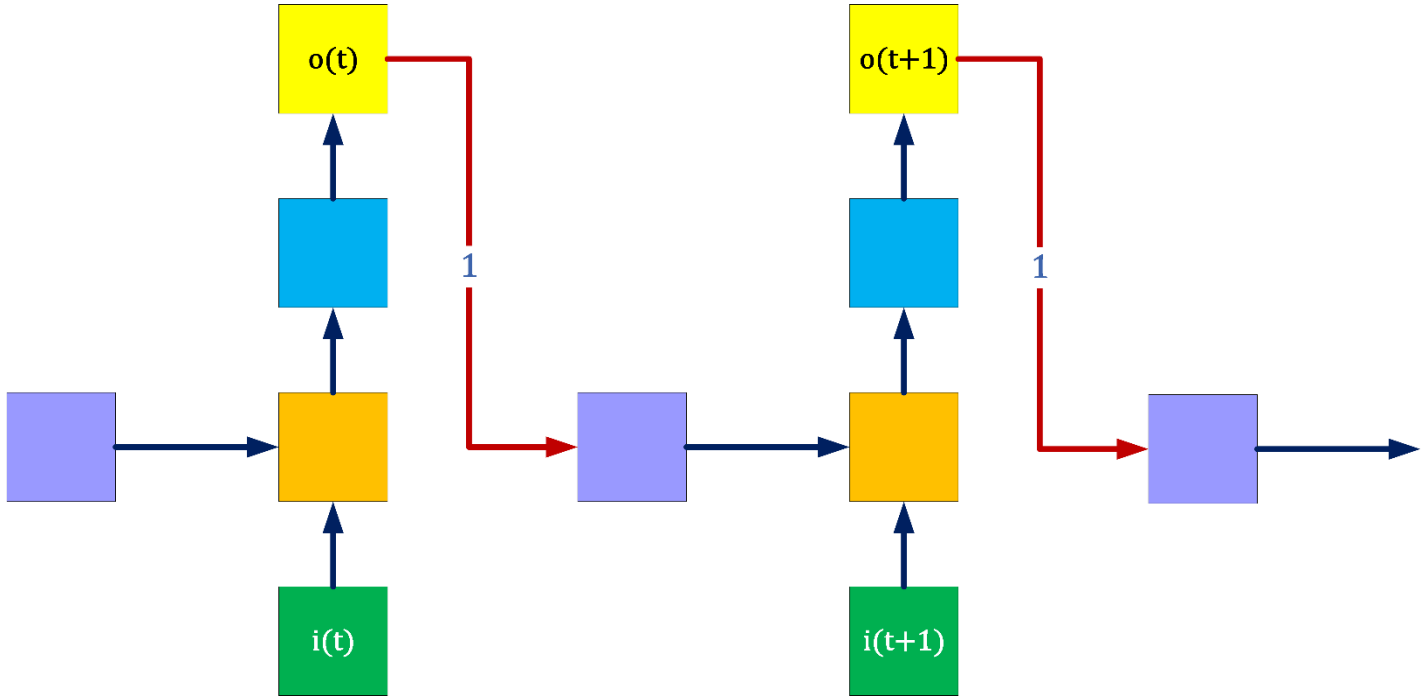
Jordan (1999) Network

- Equations:
  - $\mathbf{h}_t = \boldsymbol{\sigma}_h(\mathbf{W}_i \mathbf{i}_t + \mathbf{W}_h \mathbf{o}_{t-1} + \mathbf{b}_h)$
  - $\mathbf{o}_t = \boldsymbol{\sigma}_o(\mathbf{W}_o \mathbf{h}_t + \mathbf{b}_o)$
  - Context layer: unit-time delays



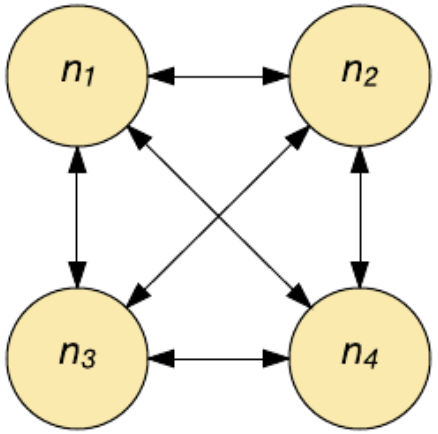
Jordan (1999) Network

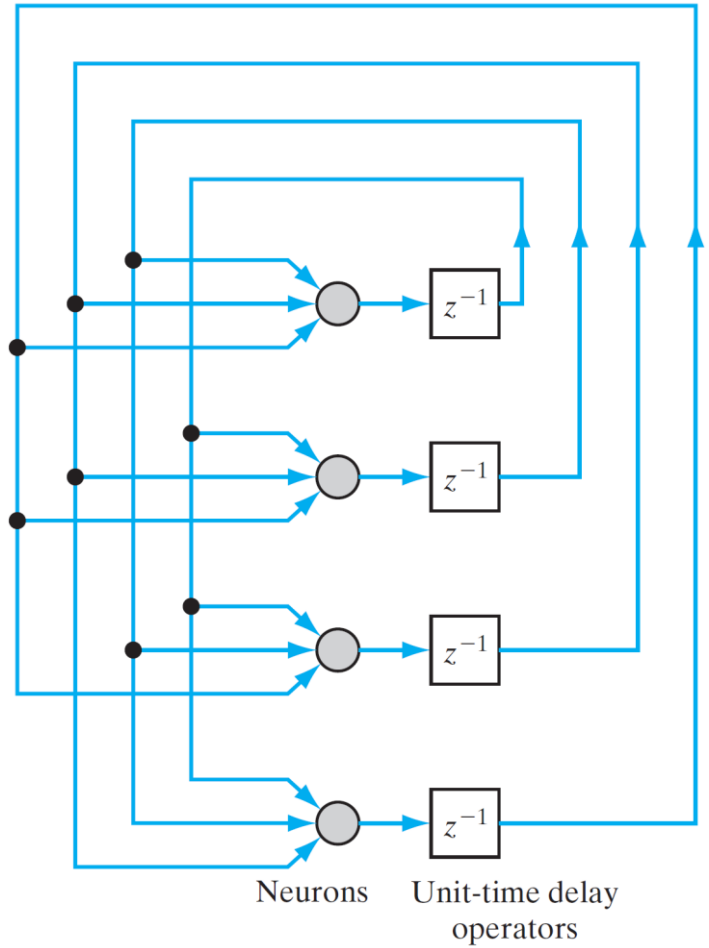
- Signal Flows



Hopfield Network (1982)

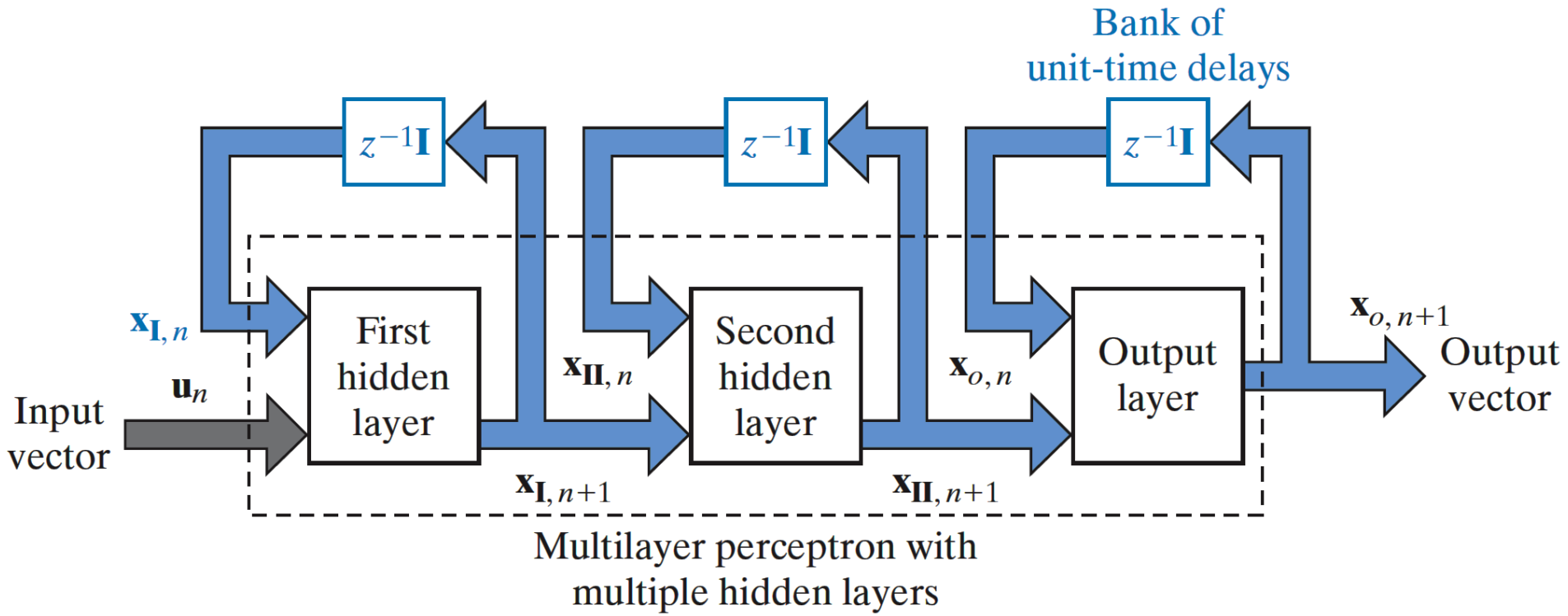
- Symmetric Weights ($w_{ij} = w_{ji}, w_{ii} = 0$)
- Fully Connected (except self connection)
- Working with initial condition
- $x_i(n+1) = sign\left(\sum_{j=1}^{N} w_{ij} x_j\right)$
- Output: fixed point of x_i
  - $x_i(n+1) = x_i(n)$





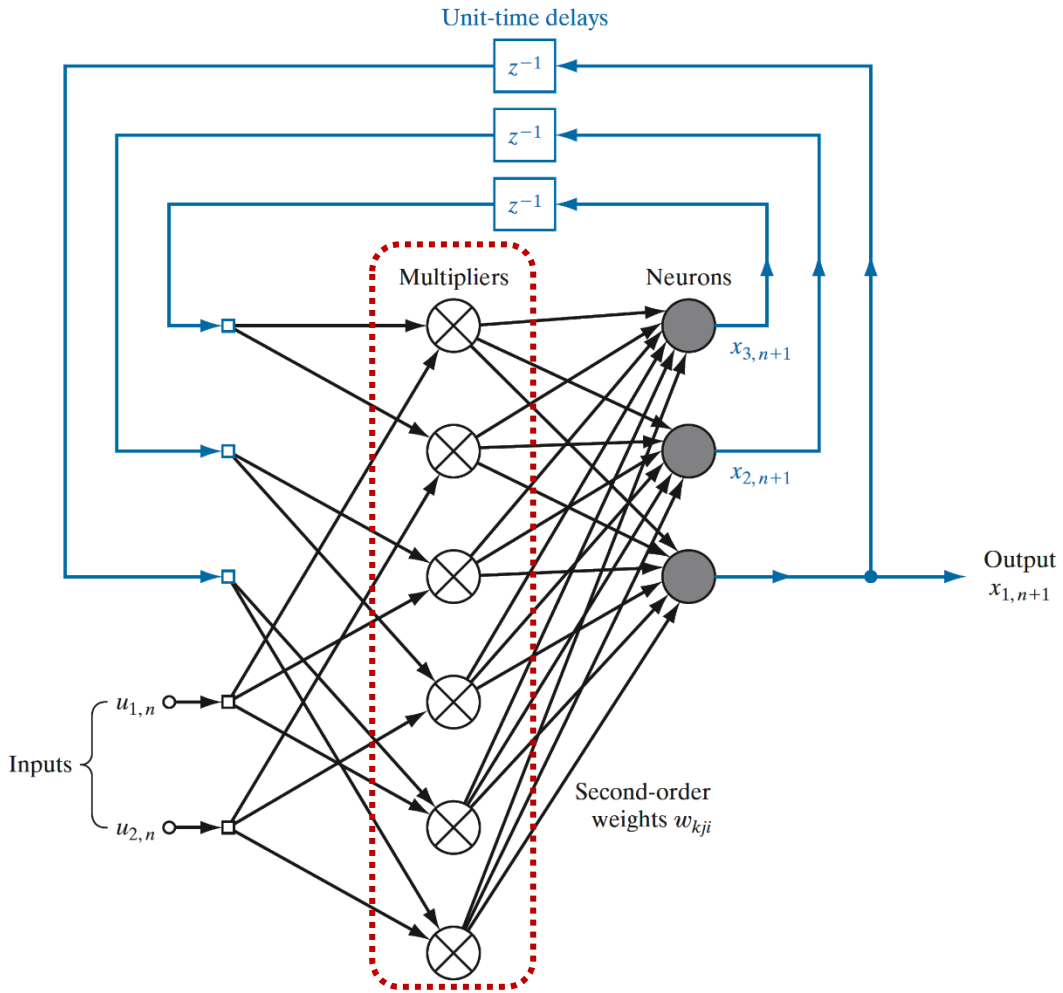
Recurrent Multilayer Perceptron (RMLP)

- Architecture:



Second-Order Recurrent Network

- Architecture:
- Neuron Equation:
- $x_{k,n} = \sum_i \sum_j w_{kij}\, x_{i,n} u_{j,n} + b_k$



Universal Approximation Theorem (Dynamic Systems)

- Consider following state-space model:

$$\mathbf{X}_{n+1} = \boldsymbol{\Phi}(\mathbf{W}_a \mathbf{X}_n + \mathbf{W}_b \mathbf{U}_n)$$
$$\mathbf{Y}_n = \mathbf{W}_c \mathbf{X}_n$$

- $\boldsymbol{\Phi}$: a vector activation function (sigmoid or tanh)

> *Any nonlinear dynamic system may be approximated by a recurrent neural network to any desired degree of accuracy and with no restrictions imposed on the compactness of the state space, provided that the network is equipped with an adequate number of hidden neurons.*

- Ref: Dynamical system identification by recurrent multilayer perceptrons, 1993

Recurrent Neural Network (RNN)

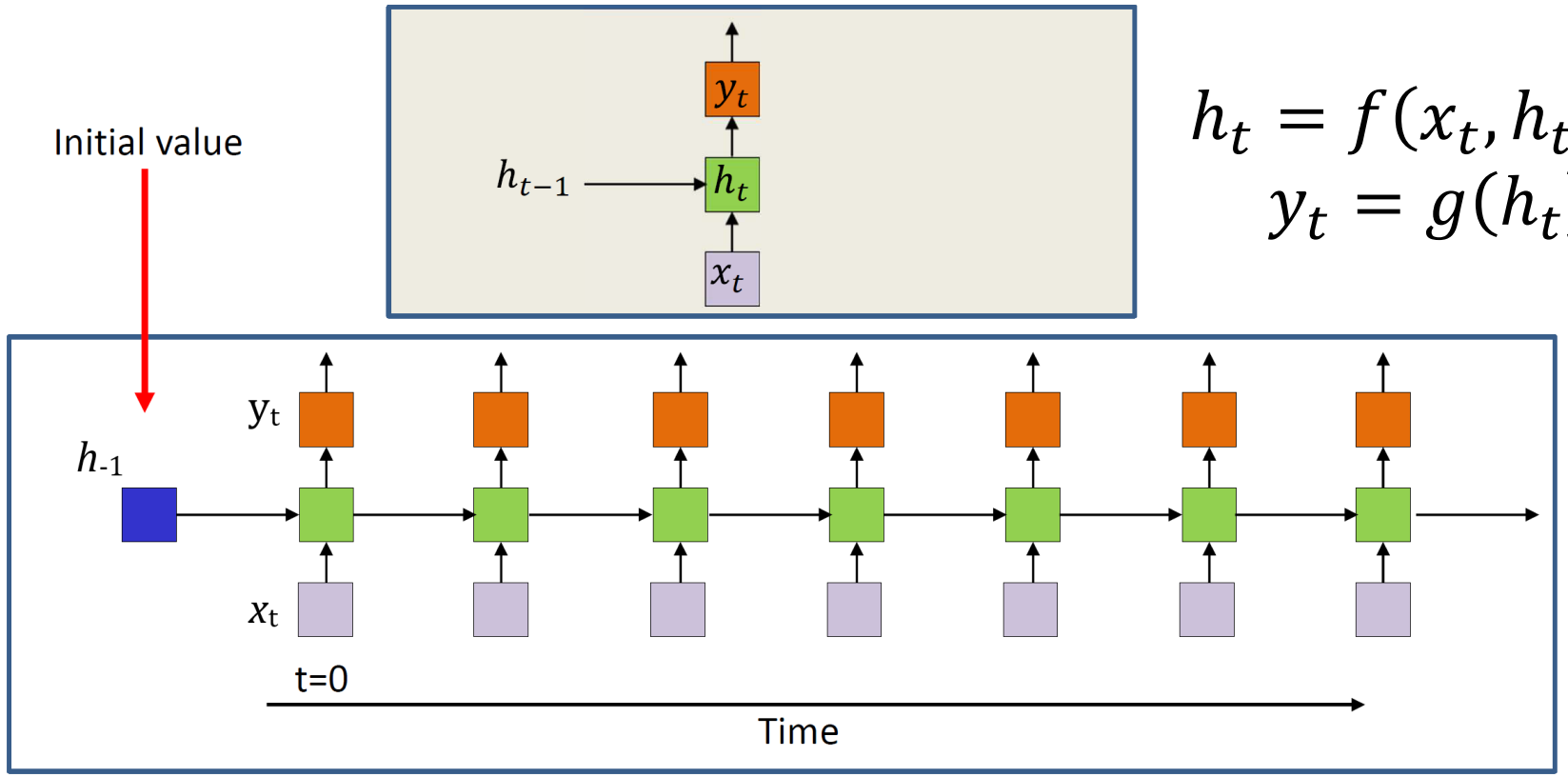
- Base equations for state-space model:

$$h_t = f(x_t, h_{t-1})$$
$$y_t = g(h_t)$$

- h_t is *state* of network and *summarizes* information about the *entire past*
- Model directly embeds the *memory* in the *state*
- Need to define initial state h_{-1}
- This is a *fully recurrent neural network* or simply a r*ecurrent neural network* (RNN)
- An *input* at $t = 0$ affects outputs *forever*

Simple RNN (State-Space Model)

- Signal and Computation Flow

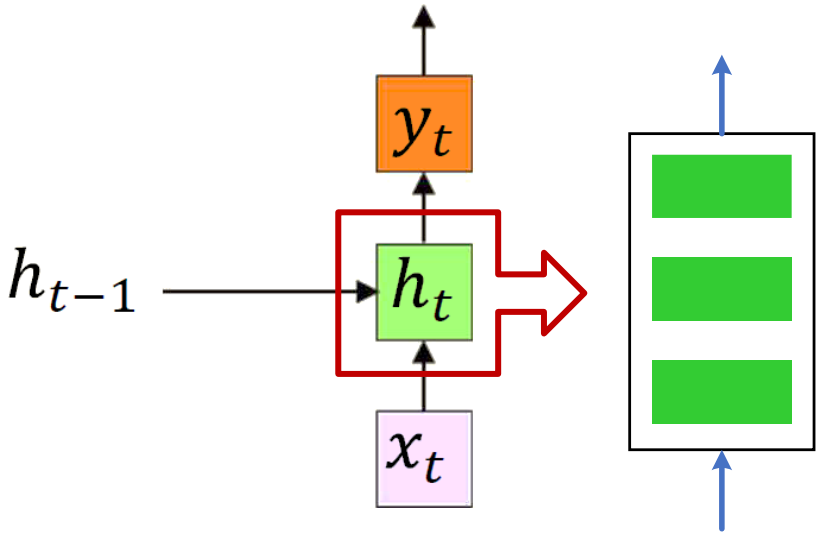


$$h_t = f(x_t, h_{t-1})$$
$$y_t = g(h_t)$$

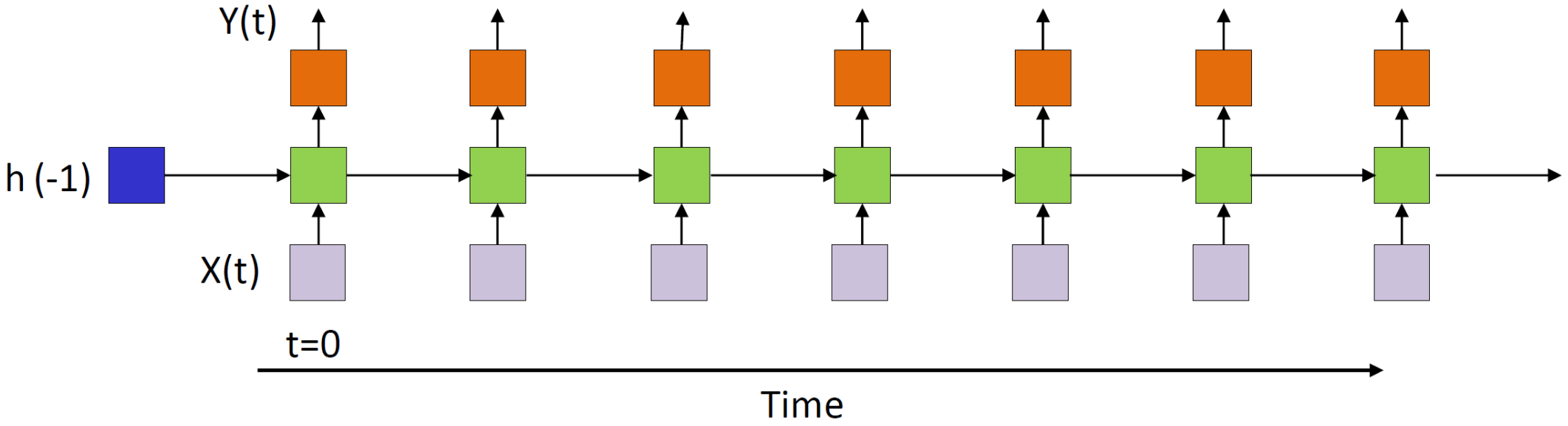
More Complex RNN

- State sub-network, $f(\cdot,\cdot)$, may be complex

$$h_t = f(x_t, h_{t-1})$$
$$y_t = g(h_t)$$

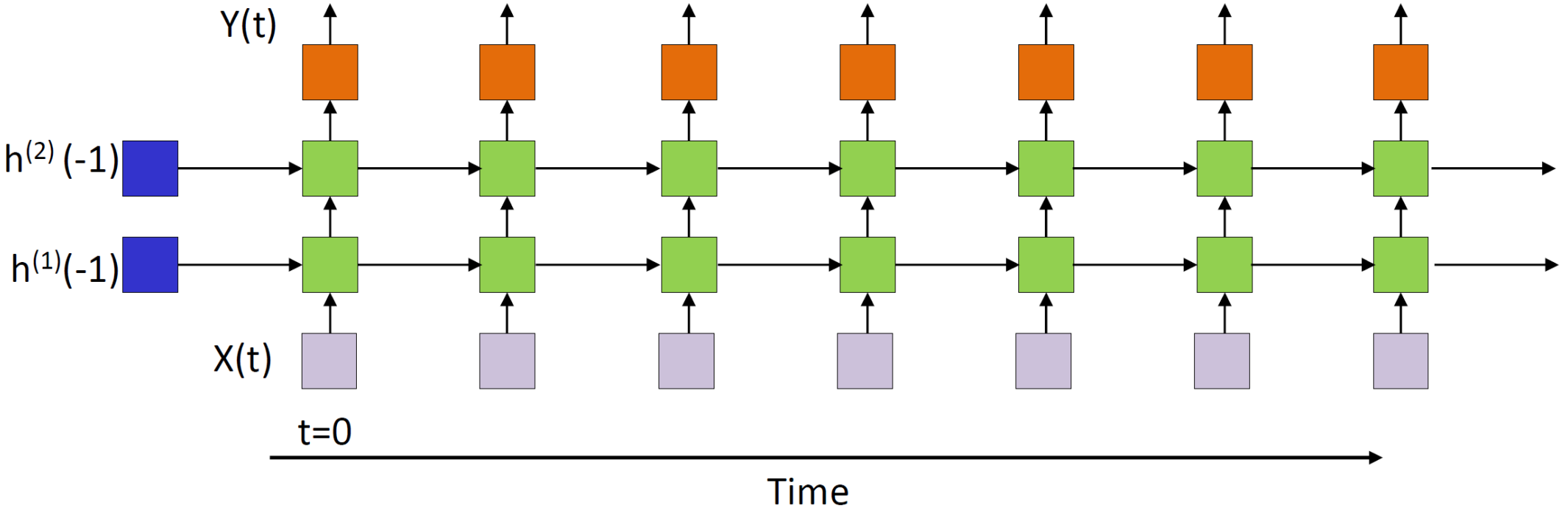


Single Hidden Layer RNN



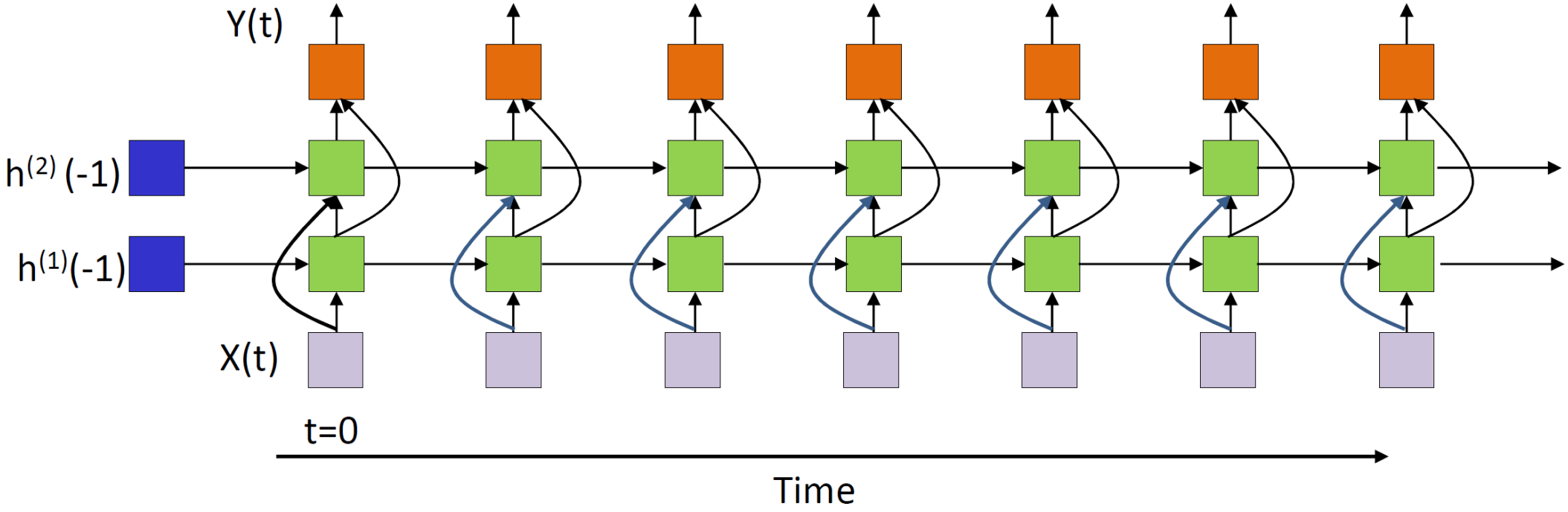
All columns are identical

Multiple Recurrent Layer RNN



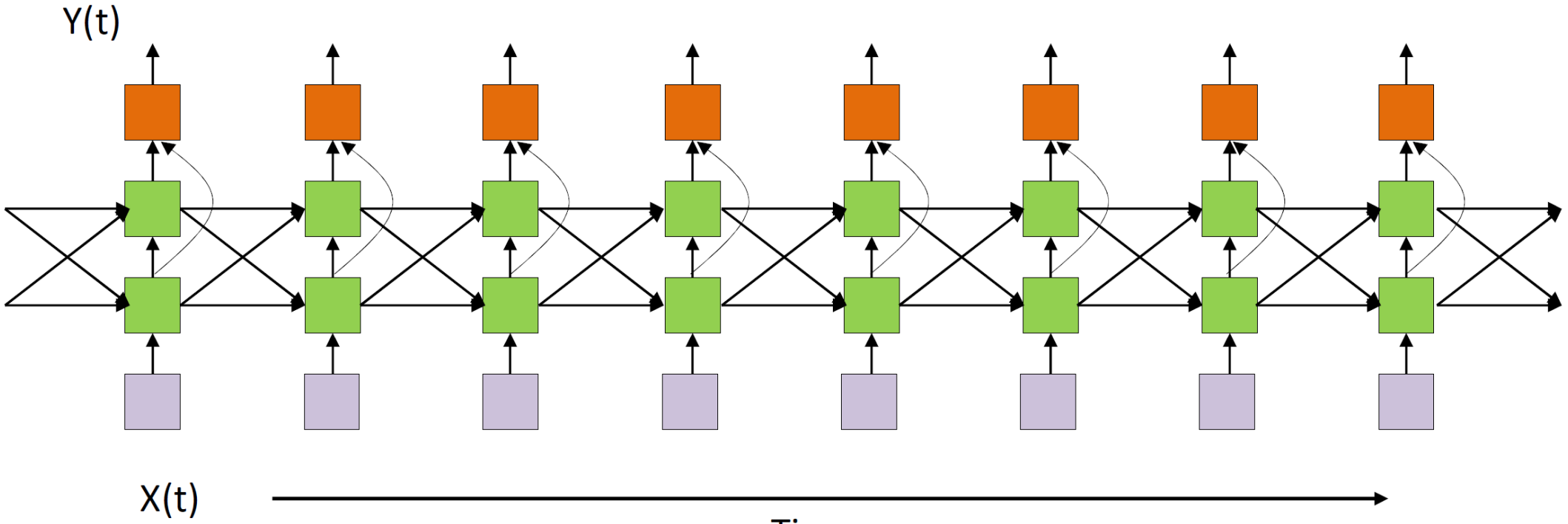
All columns are identical

More #1 Complex RNN (Skip Connection)

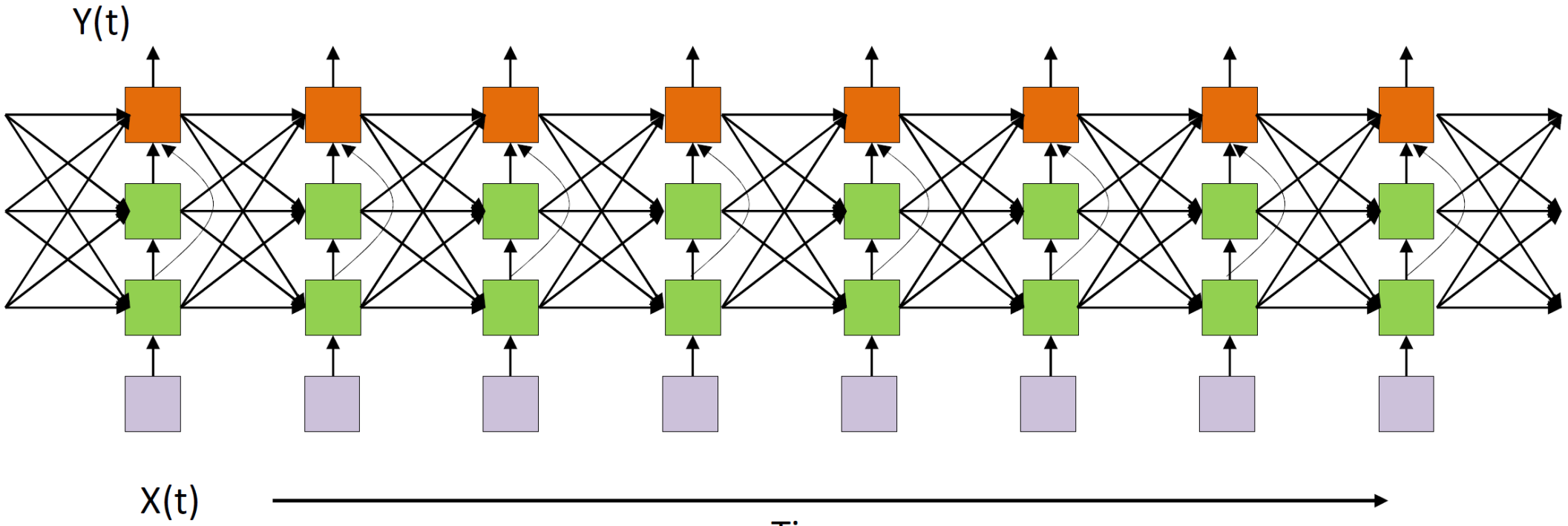


All columns are identical

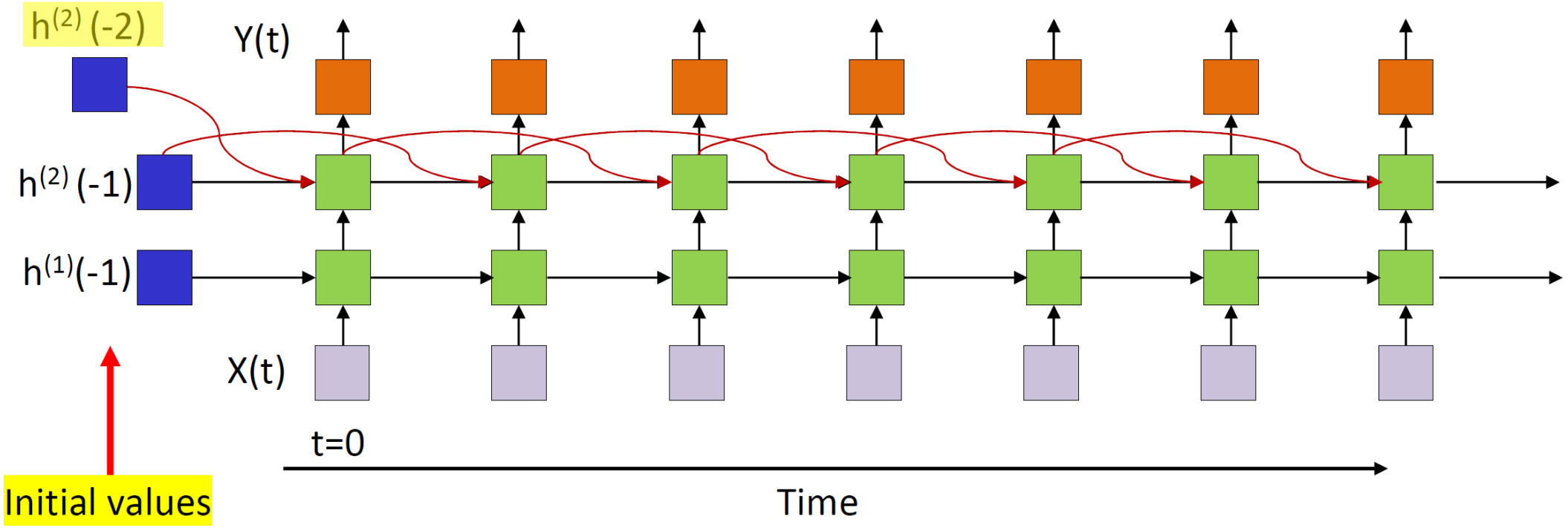
More #2 Complex RNN



More #3 Complex RNN (Skip Connection)

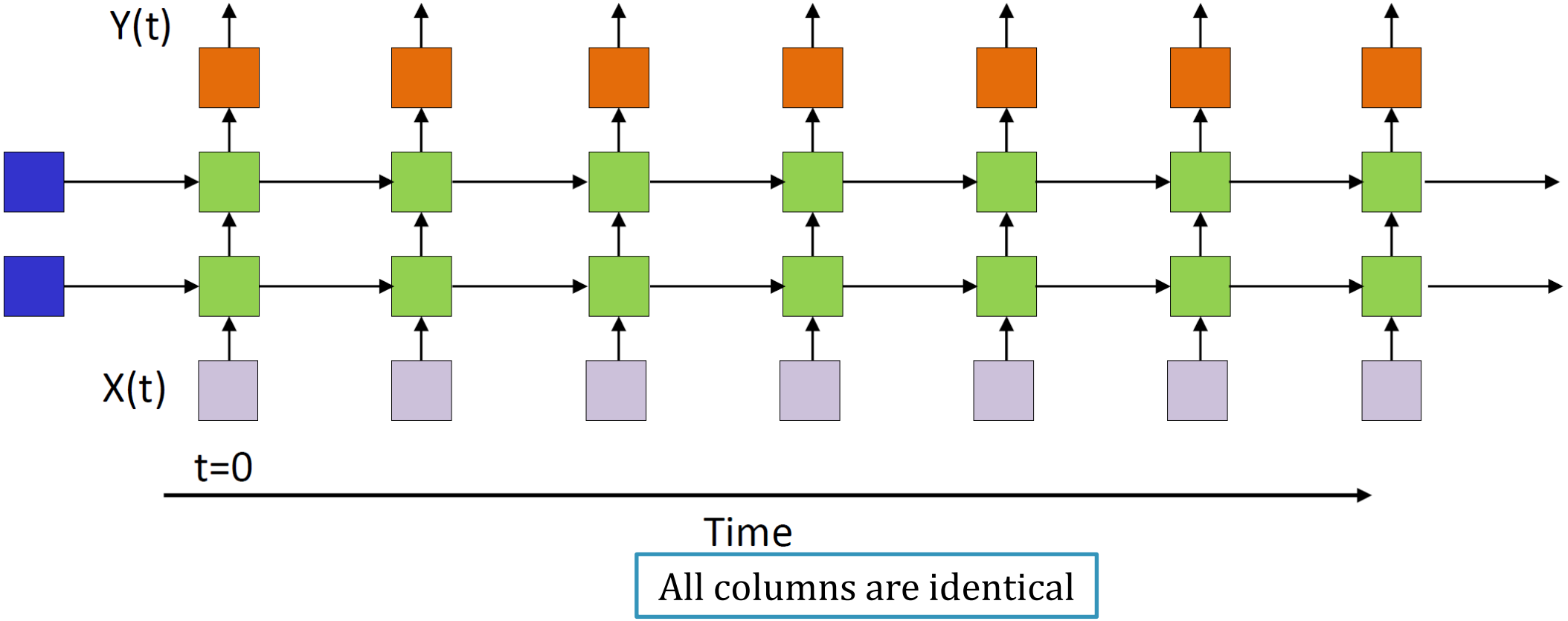


More #4 Complex RNN



All columns are identical

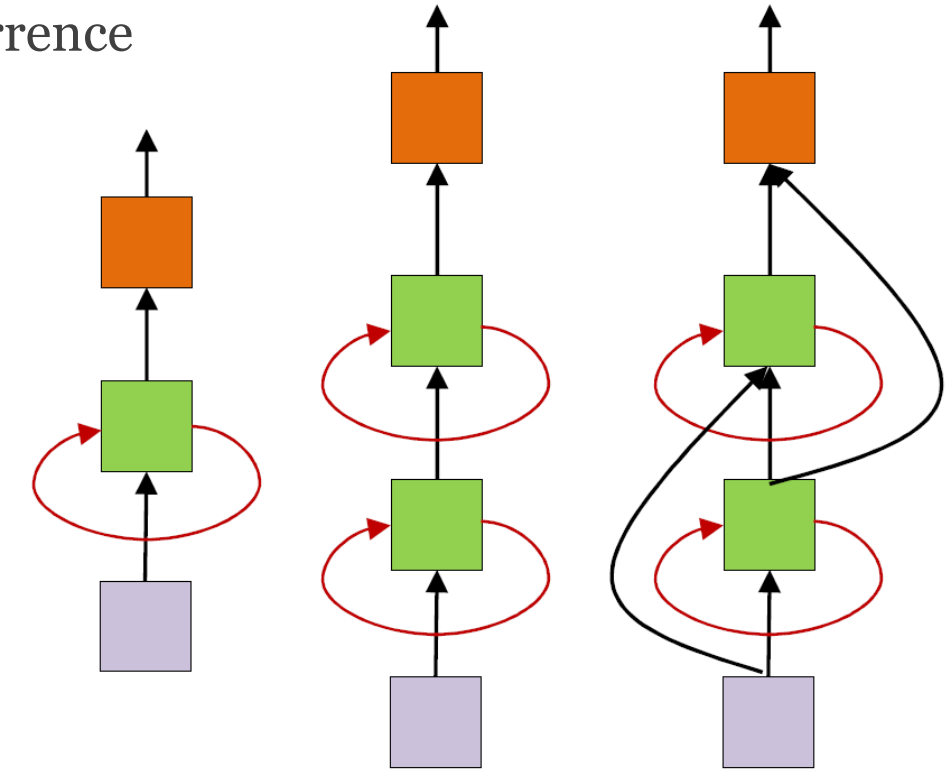
More Popular and Simple RNN



All columns are identical

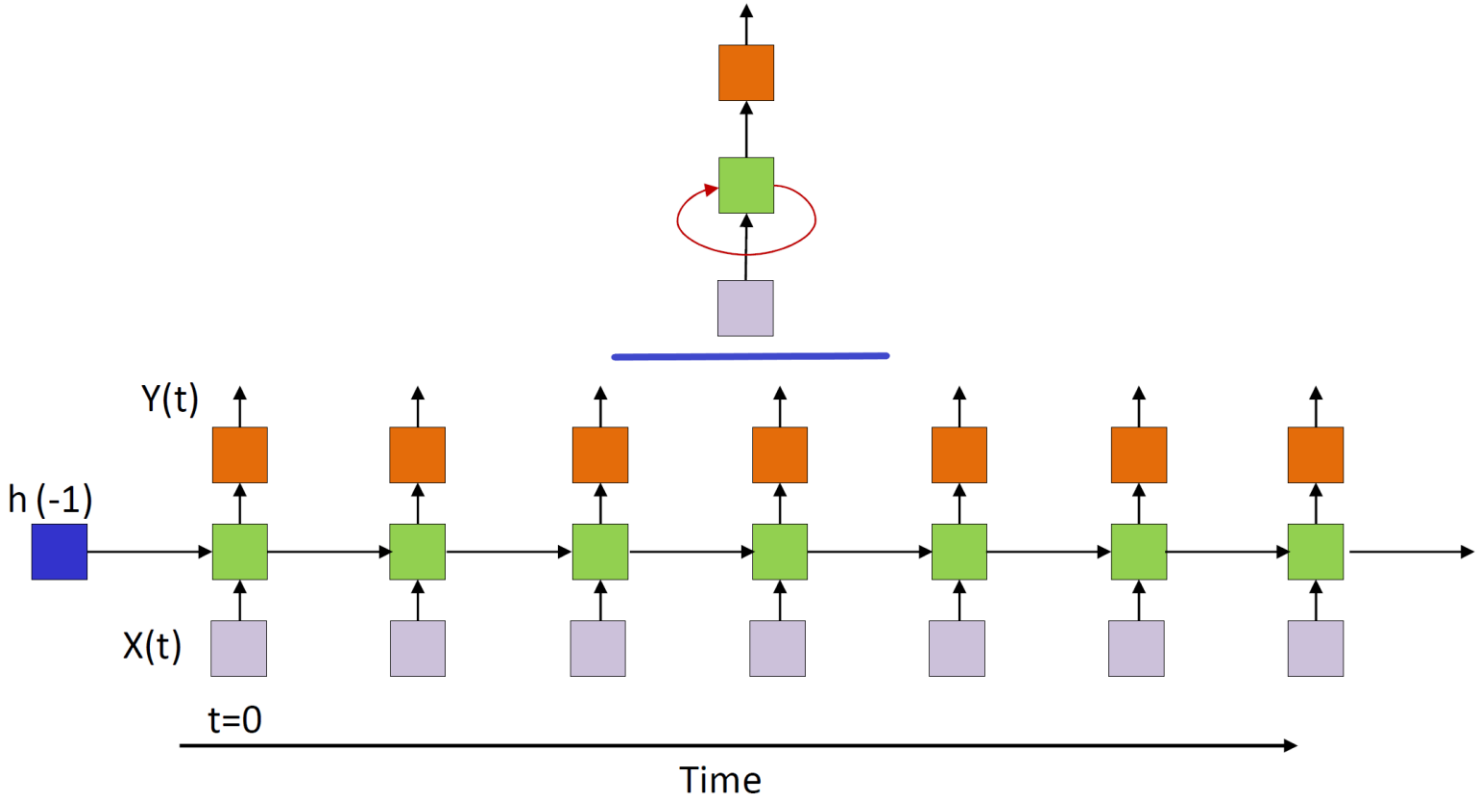
Simplified Illustration

- Simplified models often drawn
- The loops imply recurrence



Simplified Illustration

- Unfold Simplified Illustration

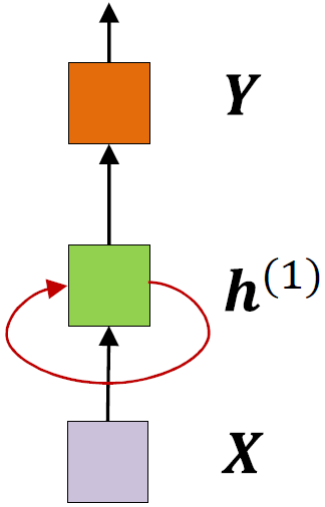


RNN Equations

- $\boldsymbol{h}^{(1)}(-1)$: Part of network parameters

$$\boldsymbol{h}^{(1)}(t) = f_1\left(\boldsymbol{W}^{(1)}\boldsymbol{X}(t) + \boldsymbol{W}^{(11)}\boldsymbol{h}^{(1)}(t-1) + \boldsymbol{b}^{(1)}\right)$$

$$Y(t) = f_2\left(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)}(t) + \boldsymbol{b}^{(2)}\right)$$



- Superscript indicates layer of network from which inputs are obtained
- Assuming vector function at output, e.g. softmax.
- The state node activation, $f_1(\cdot)$ is typically $\tanh(\cdot)$

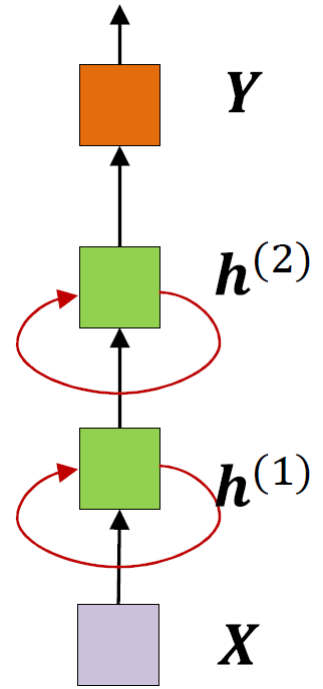
RNN Equations

- $\boldsymbol{h}^{(1)}(-1)$ and $\boldsymbol{h}^{(2)}(-1)$: Part of network parameters

$$\boldsymbol{h}^{(1)}(t) = f_1\left(\boldsymbol{W}^{(1)}\boldsymbol{X}(t) + \boldsymbol{W}^{(11)}\boldsymbol{h}^{(1)}(t-1) + \boldsymbol{b}^{(1)}\right)$$

$$\boldsymbol{h}^{(2)}(t) = f_2\left(\boldsymbol{W}^{(2)}\boldsymbol{h}^{(1)}(t) + \boldsymbol{W}^{(22)}\boldsymbol{h}^{(2)}(t-1) + \boldsymbol{b}^{(2)}\right)$$

$$\boldsymbol{Y}(t) = f_3\left(\boldsymbol{W}^{(3)}\boldsymbol{h}^{(2)}(t) + \boldsymbol{b}^{(3)}\right)$$



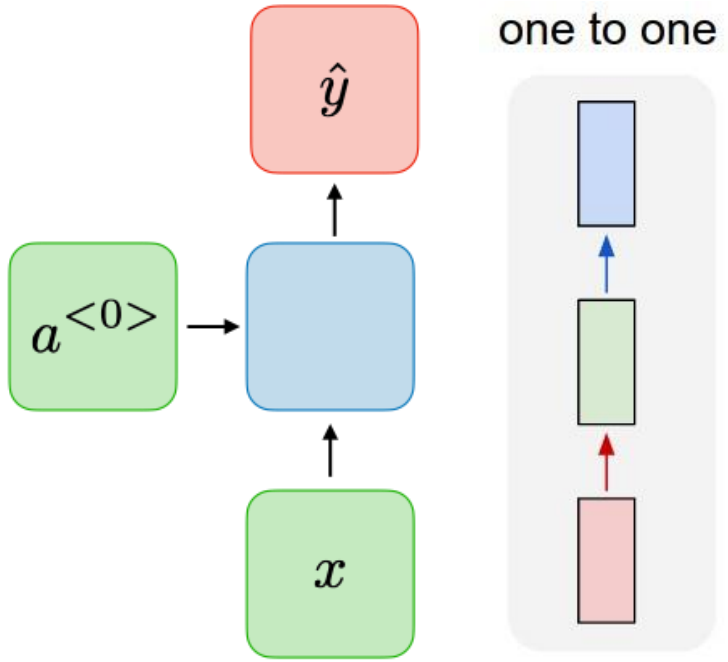
- Superscript indicates layer of network from which inputs are obtained
- Assuming vector function at output, e.g. softmax.
- The state node activation, $f_1(\cdot)$ and $f_2(\cdot)$ is typically $\tanh(\cdot)$

RNN variants

- Type of RNNs based on number of input(s)-output(2):
  - One-to-One
  - One-to-Many
  - Many-to-One
  - Many-to-Many
- Type of RNNs based on signal flow:
  - Deep RNN (DRNN)
  - Bidirectional (BRNN)

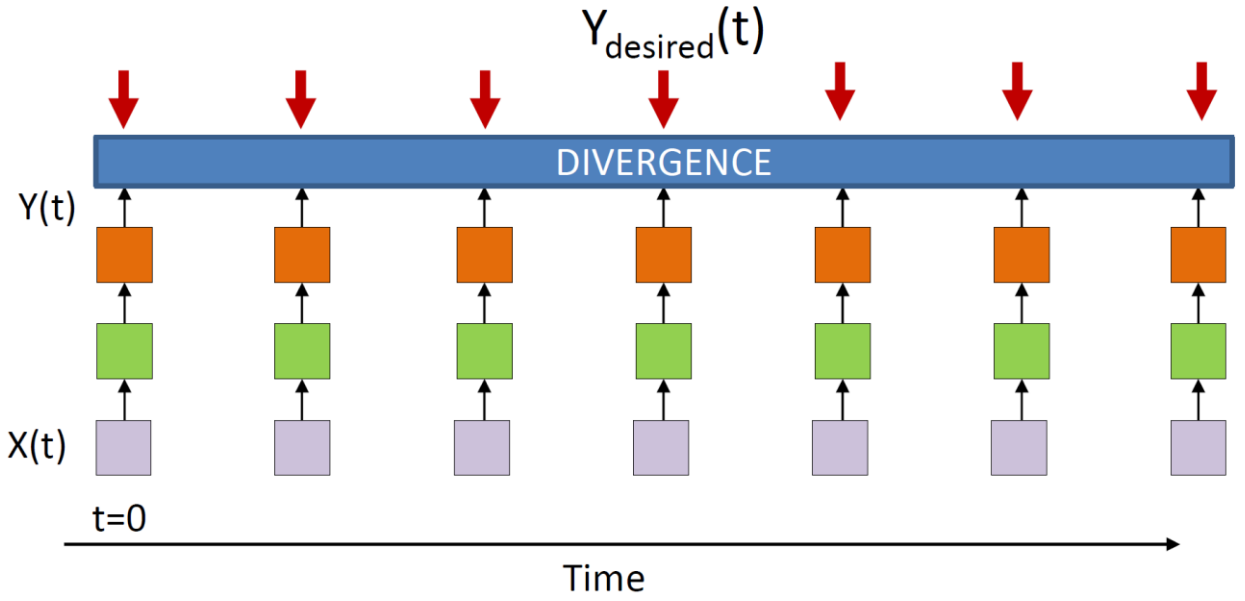
One-to-One RNN

- Conventional MLP (Vanilla/Traditional Neural Networks)
- Time-synchronous outputs (analyzing time series)
- $T_x = 1, T_y = 1$



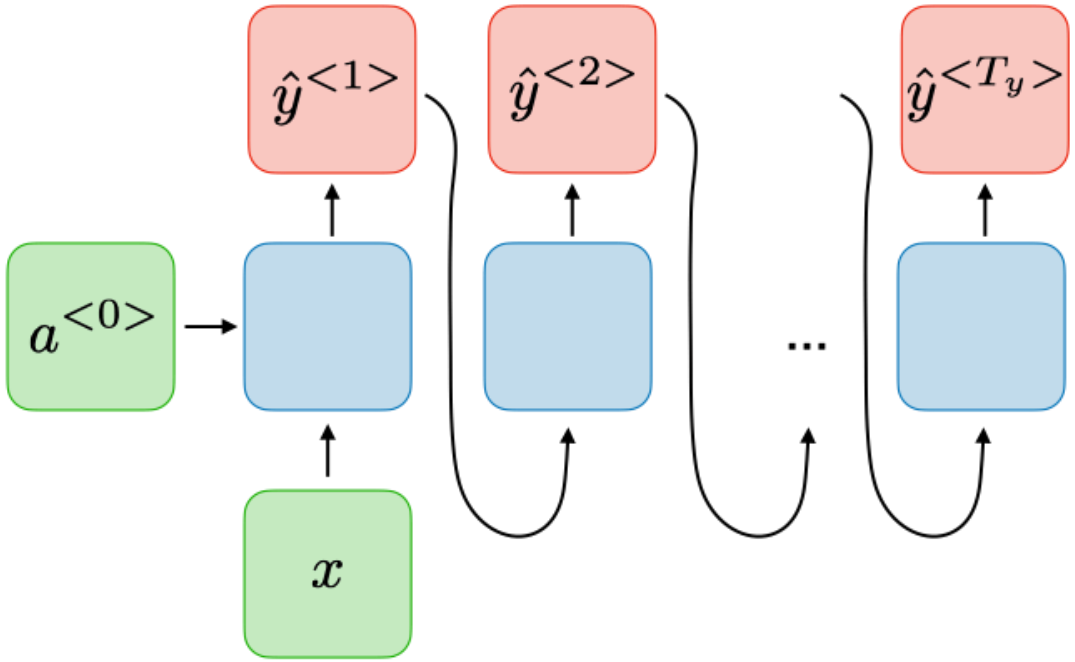
One-to-One RNN

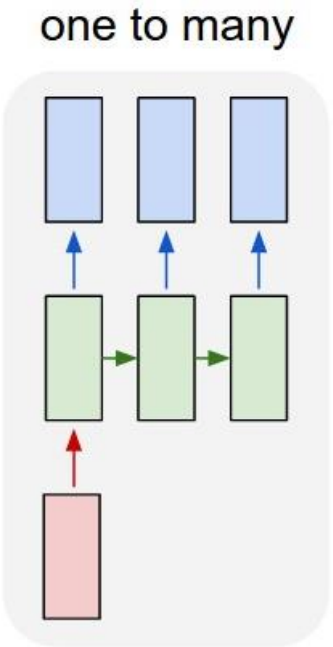
- The output at time is t unrelated to the output at $t' \neq t$
- In the context of analyzing time series, the divergence to minimize is still the divergence between two series. (one-to-one correspondence)
- Total divergence is the sum of local divergences at individual times



One-to-Many RNN

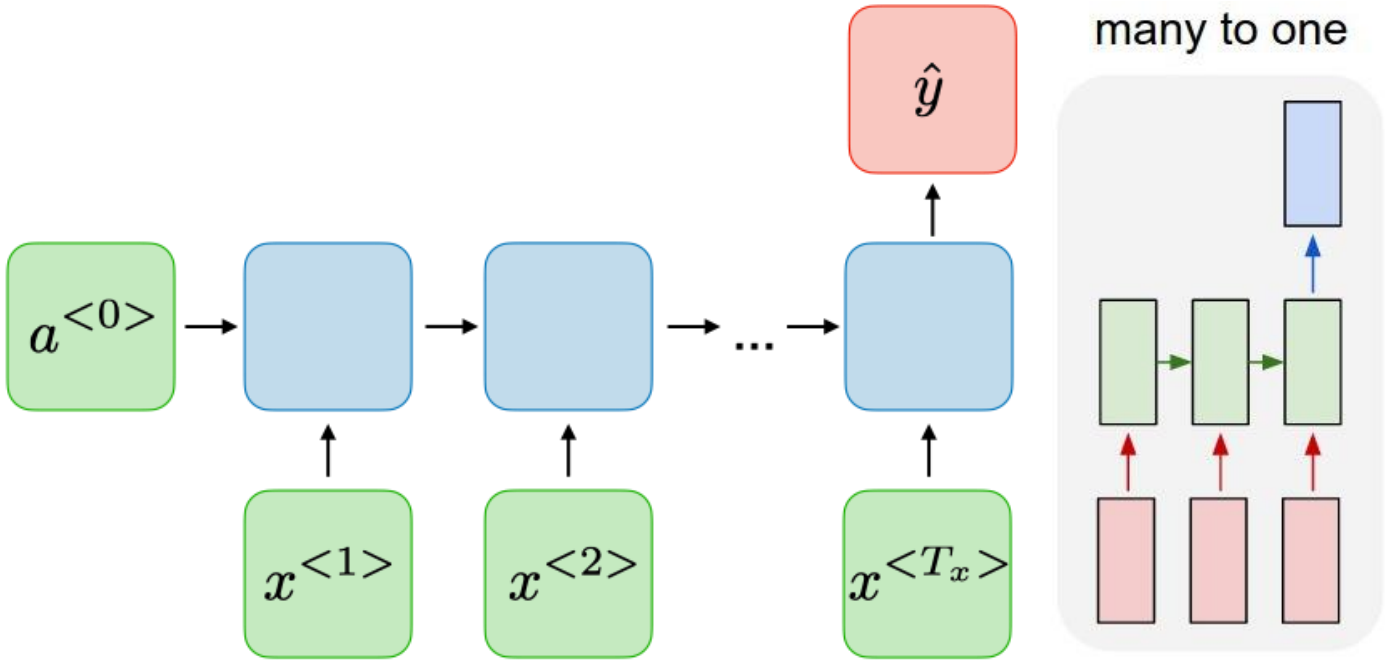
- Sequence generation
- $T_x = 1, T_y > 1$
- Applications:
  - Image Captioning
  - Music generation





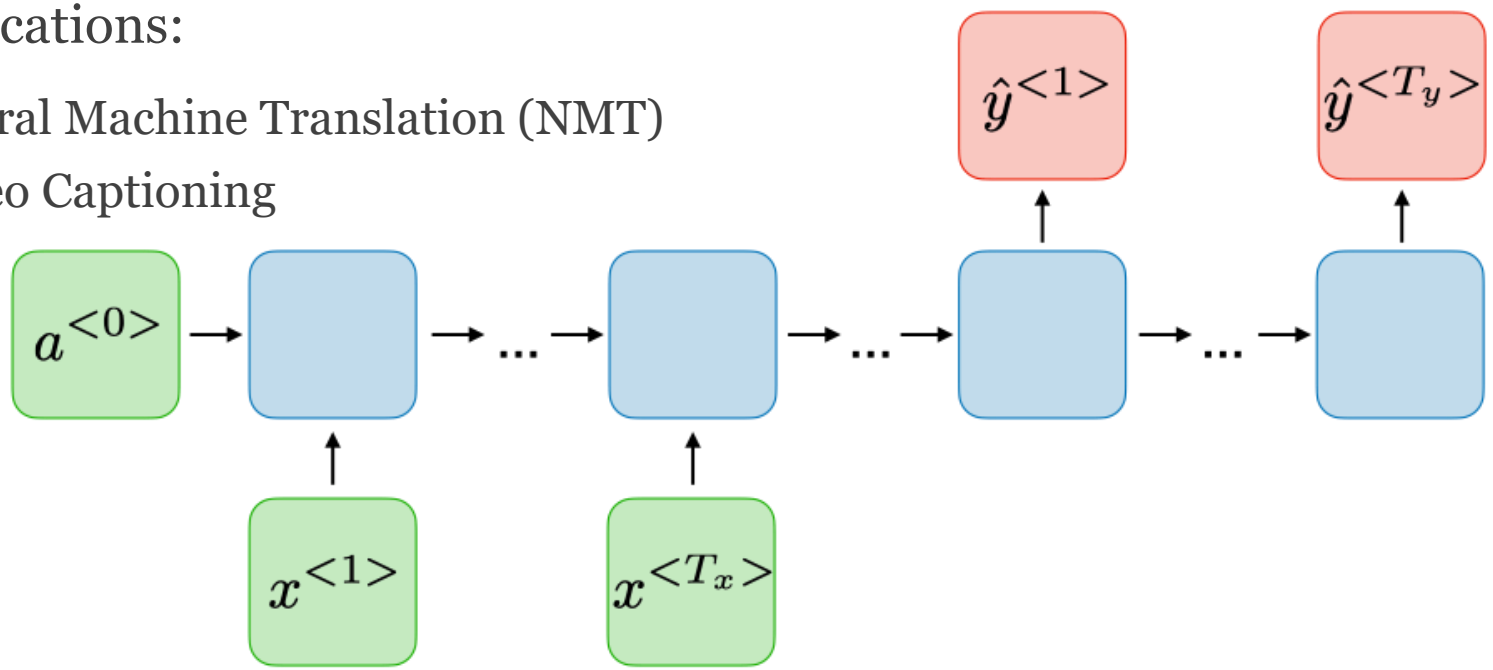
Many-to-One RNN

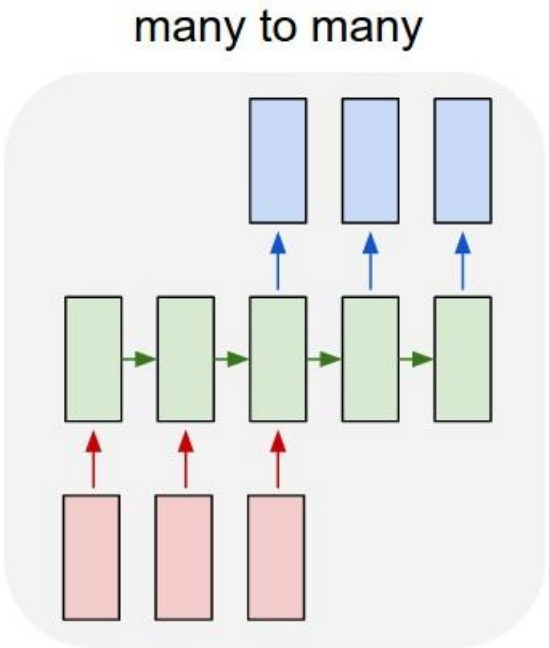
- Sequence based prediction or classification
- $T_x > 1, T_y = 1$
- Applications:
  - Speech Recognition
  - Music generation
  - Action Prediction
  - Sentiment Classification
  - Emotion Classification



Many-to-Many RNN

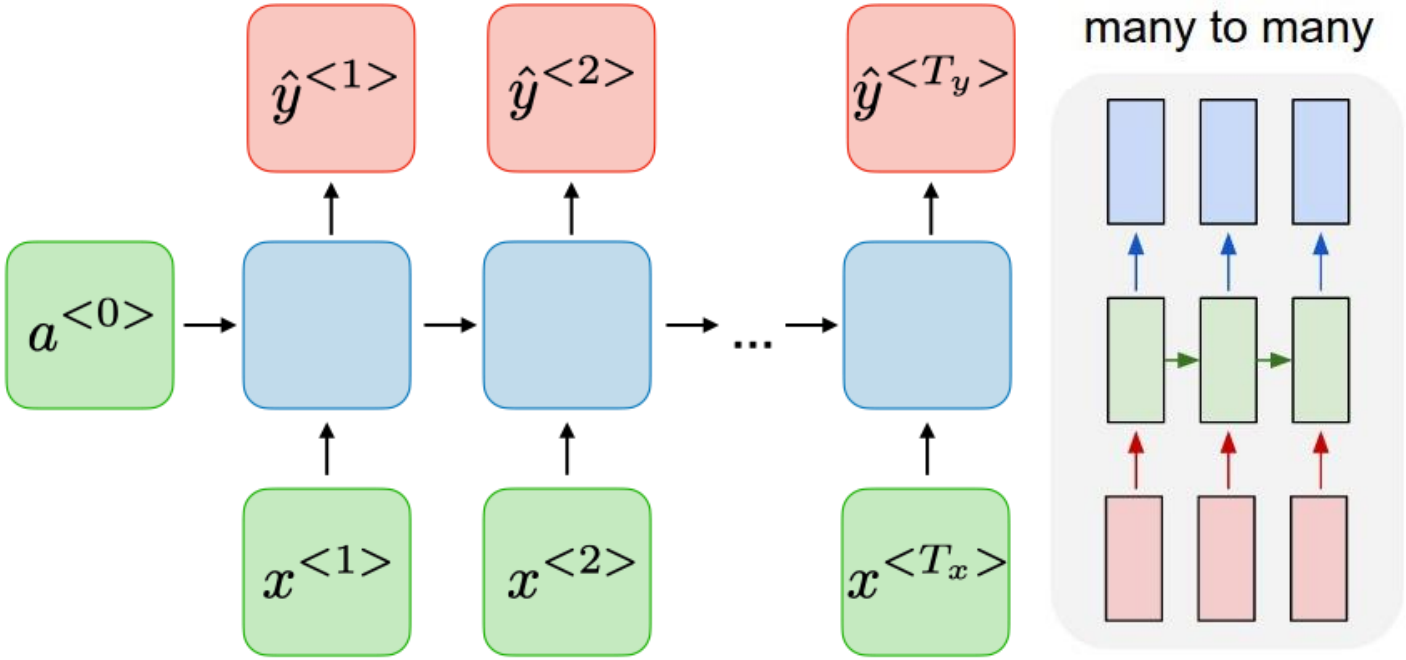
- Delayed sequence to sequence
- $T_x \neq T_y > 1$
- Applications:
  - Neural Machine Translation (NMT)
  - Video Captioning





Many-to-Many RNN

- Sequence to sequence
- $T_x = T_y > 1$
- Applications:
  - Stock problem
  - Label prediction
  - Video classification on frame level
  - Named entity recognition

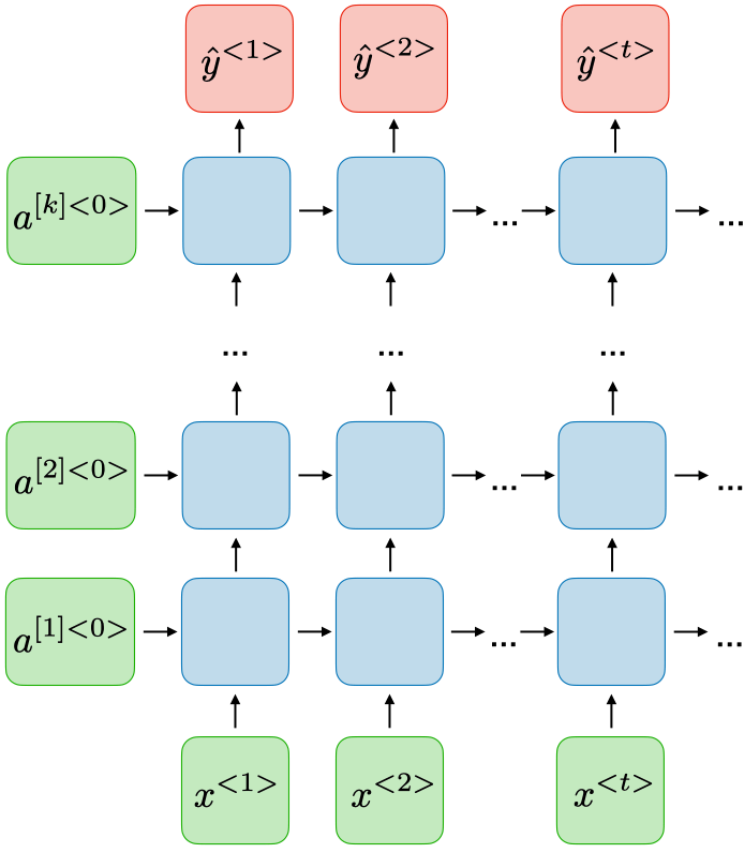


Many-to-Many RNN

- Named entity recognition

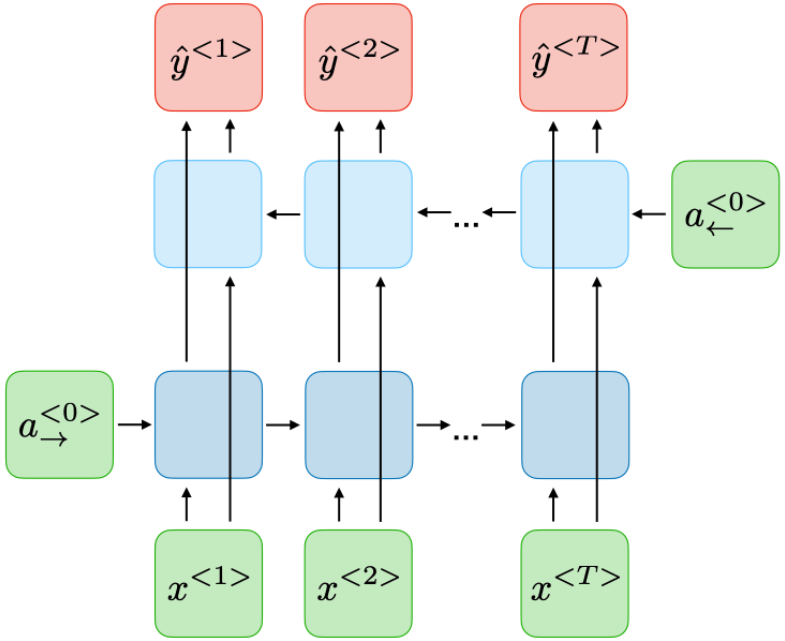
Deep RNN (DRNN)

- DRNN Architecture:



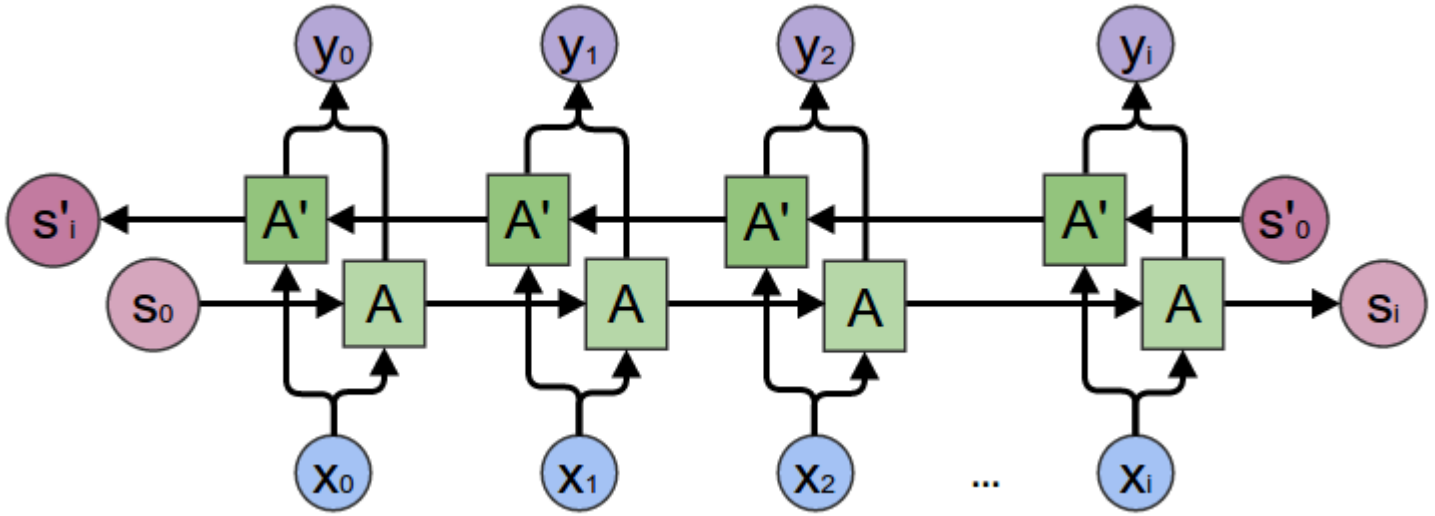
Bidirectional RNN (BRNN)

- BRNN Architecture:
- Non-Causal inference (Back to future ☺)



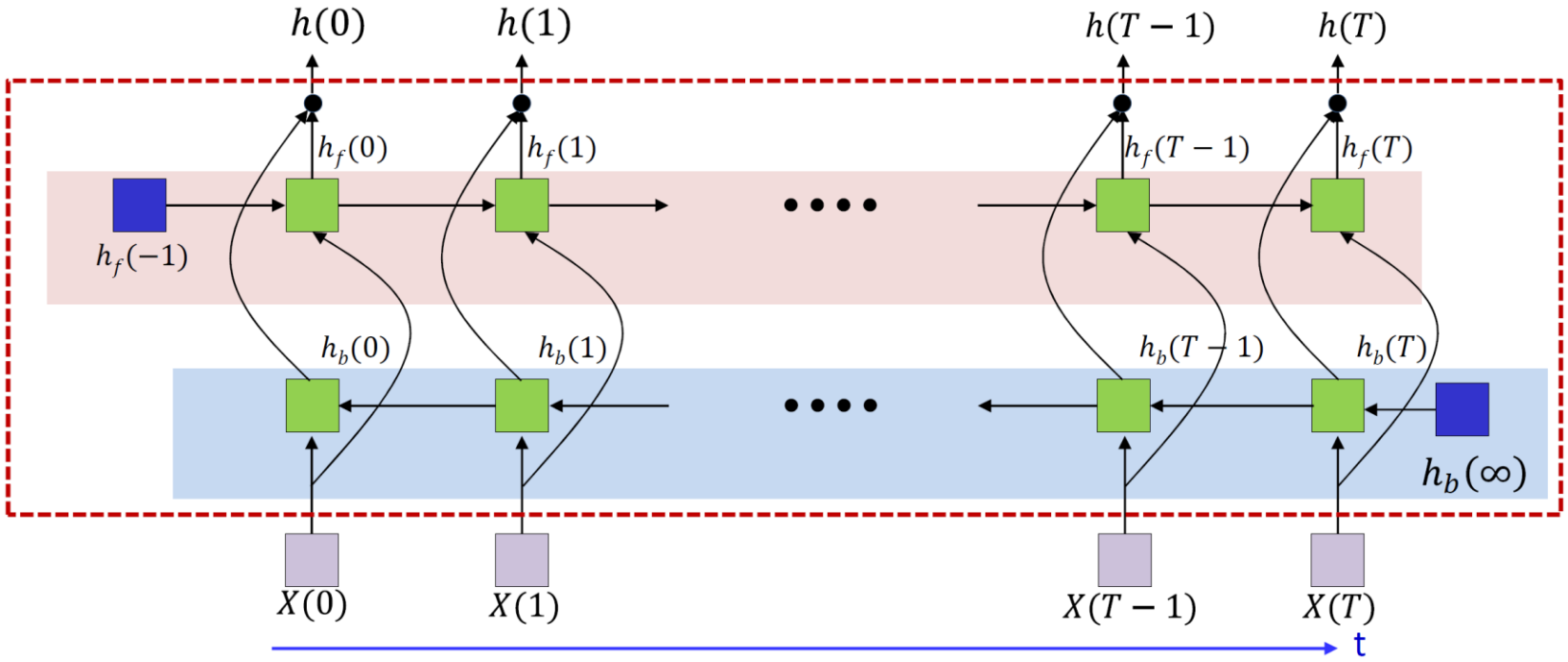
RNN Variations

- Bidirectional Recurrent Neural Network:
  - Non-Causal inference (Back to future ☺)
    - Merge to RNN (one forward through time 1→t and one backward through time t→1)



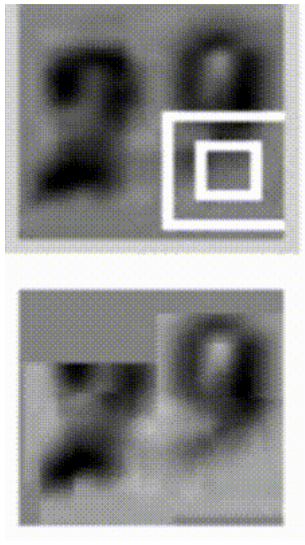
Bidirectional RNN (BRNN)

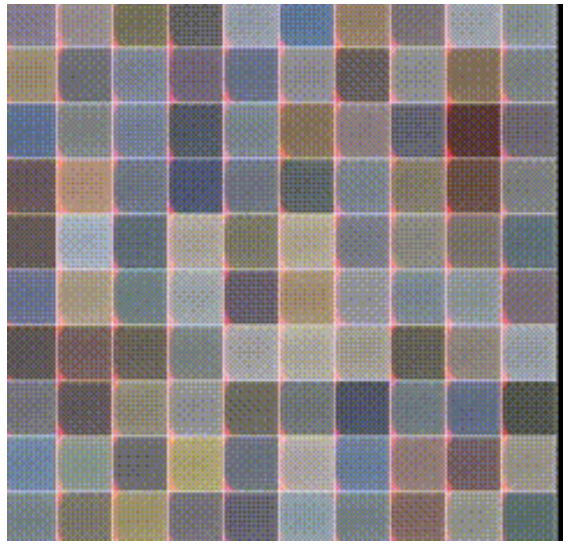
- The computed states of both networks are combined (concatenate) to give the output of the bidirectional block, $h(t) = [h_f(t); h_b(t)]$



Sequential Processing of Non-Sequence Data

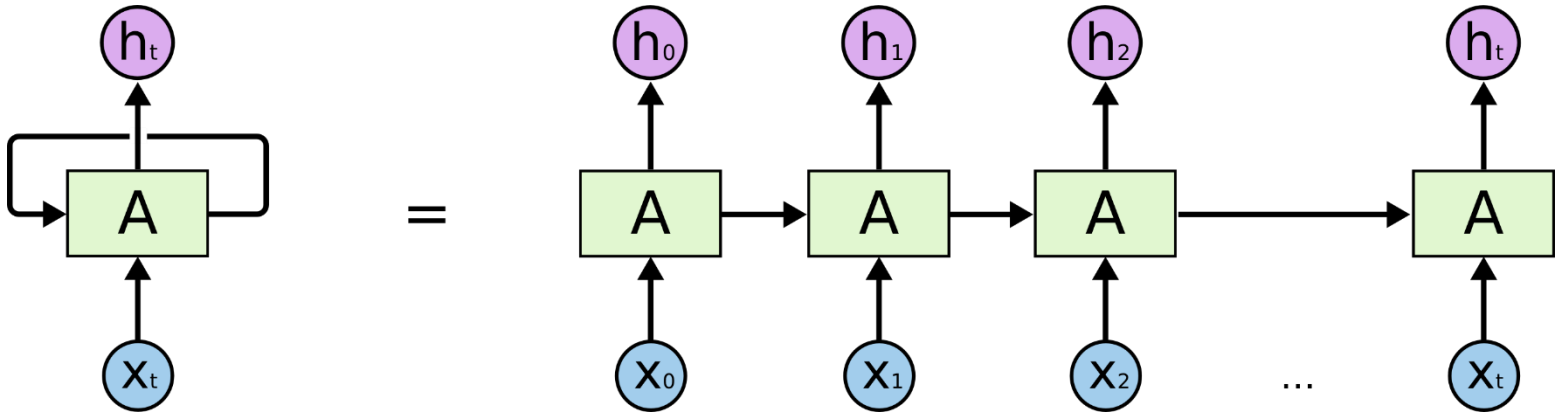
- Example:
  - Scan image (via patch) from left to right, top to bottom
  - Left (paper): Multiple Object Recognition with Visual Attention
  - Right (paper): DRAW: A Recurrent Neural Network For Image Generation





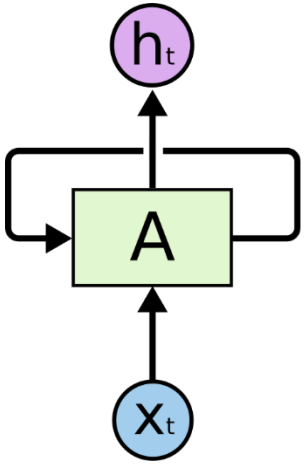
RNN Unfolding/Unrolling

- Unfolding/Unrolling Computational Graph (Simple RNN):



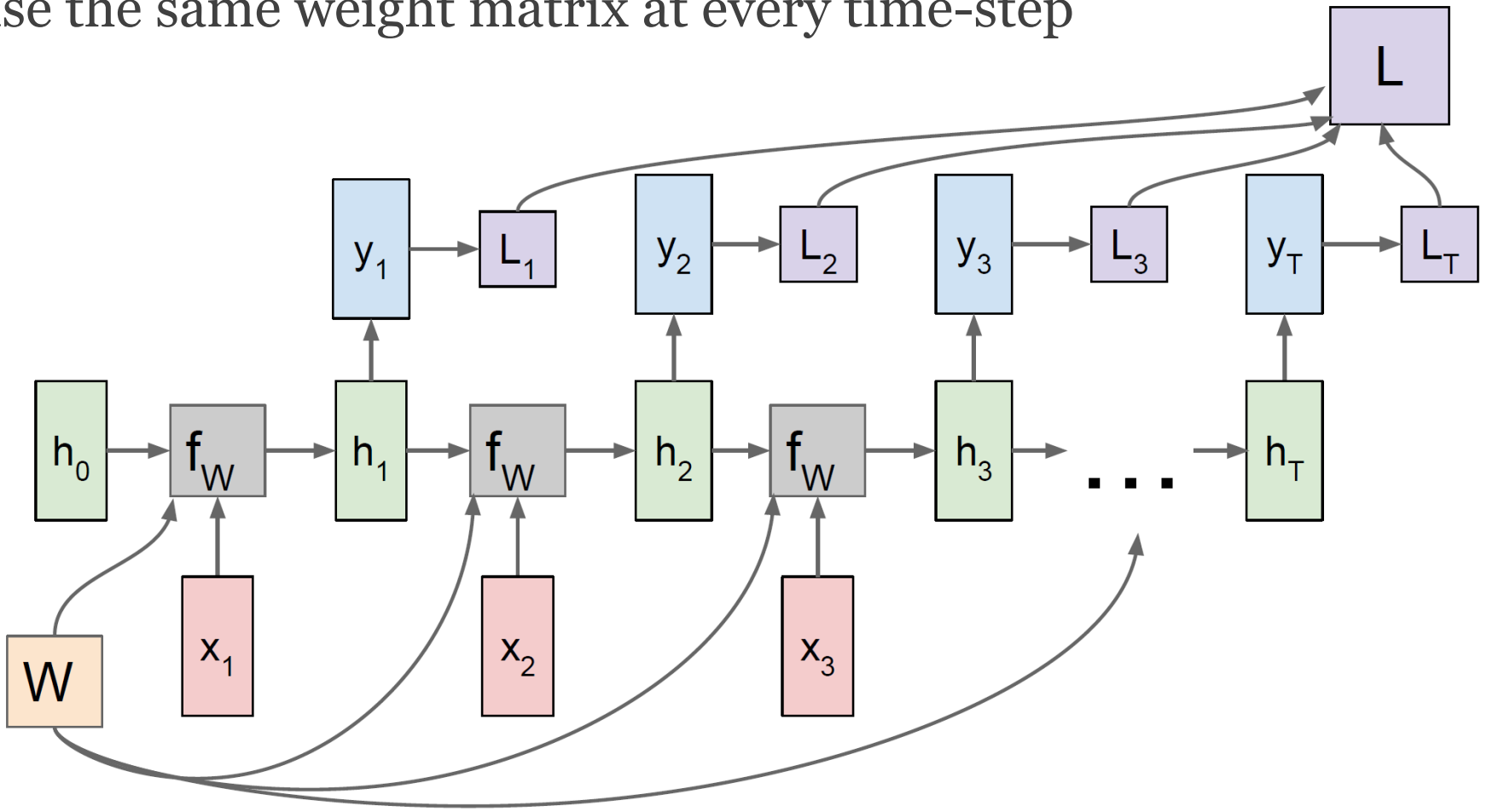
RNN Equations

- (Simple/Vanilla/Elman) RNN:
  - x_t: input sequence
  - y_t: output sequence
  - h_t: A single hidden state vector
- Equations:
  - $h_t = f_W(h_{t-1}, x_t)$
  - $h_t = g(W_{hh}h_{t-1} + W_{xh}x_t) = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$
  - $y_t = W_{hy}h_t$



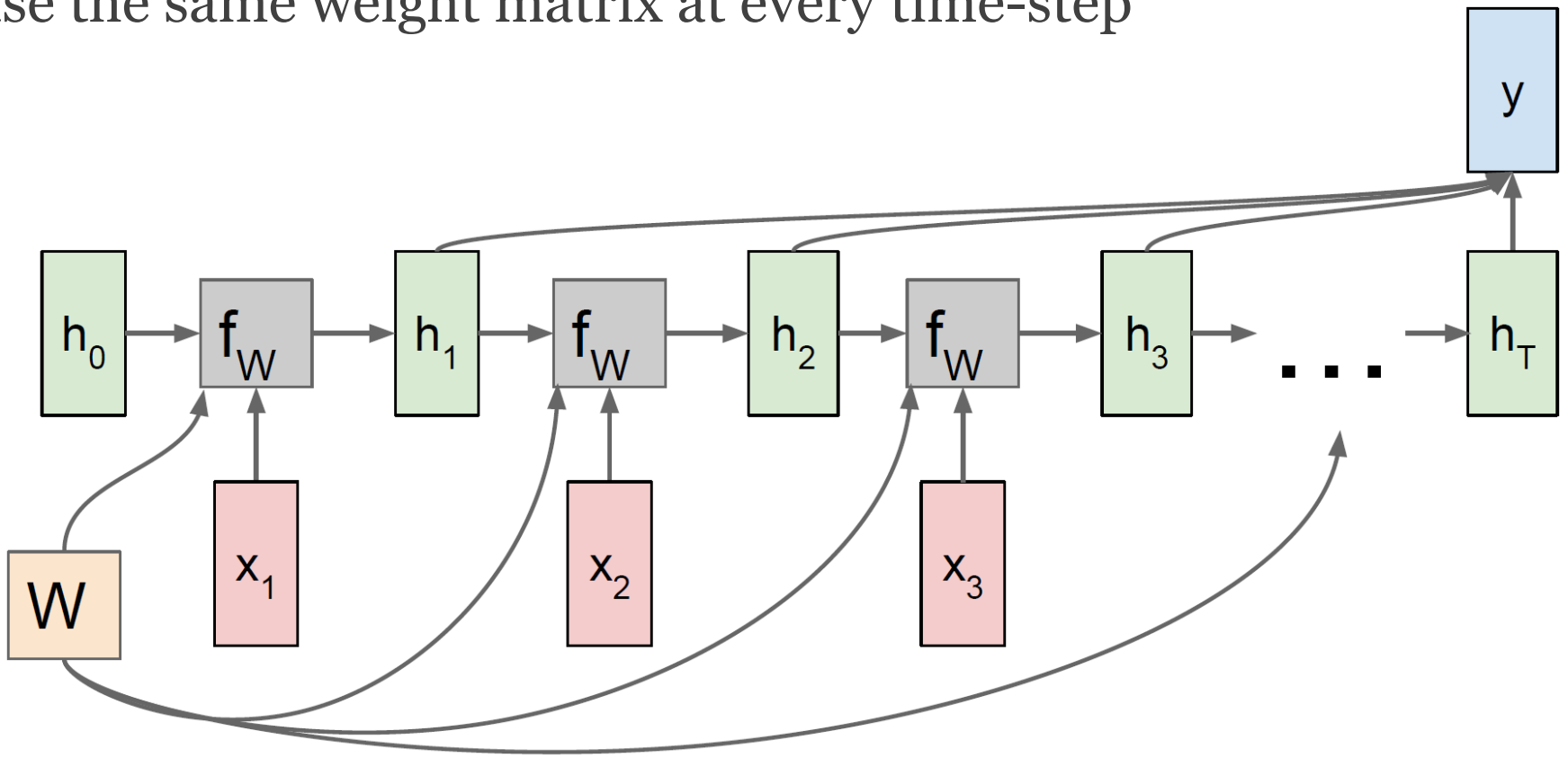
Computational Graph - Many-to-Many

- Re-use the same weight matrix at every time-step



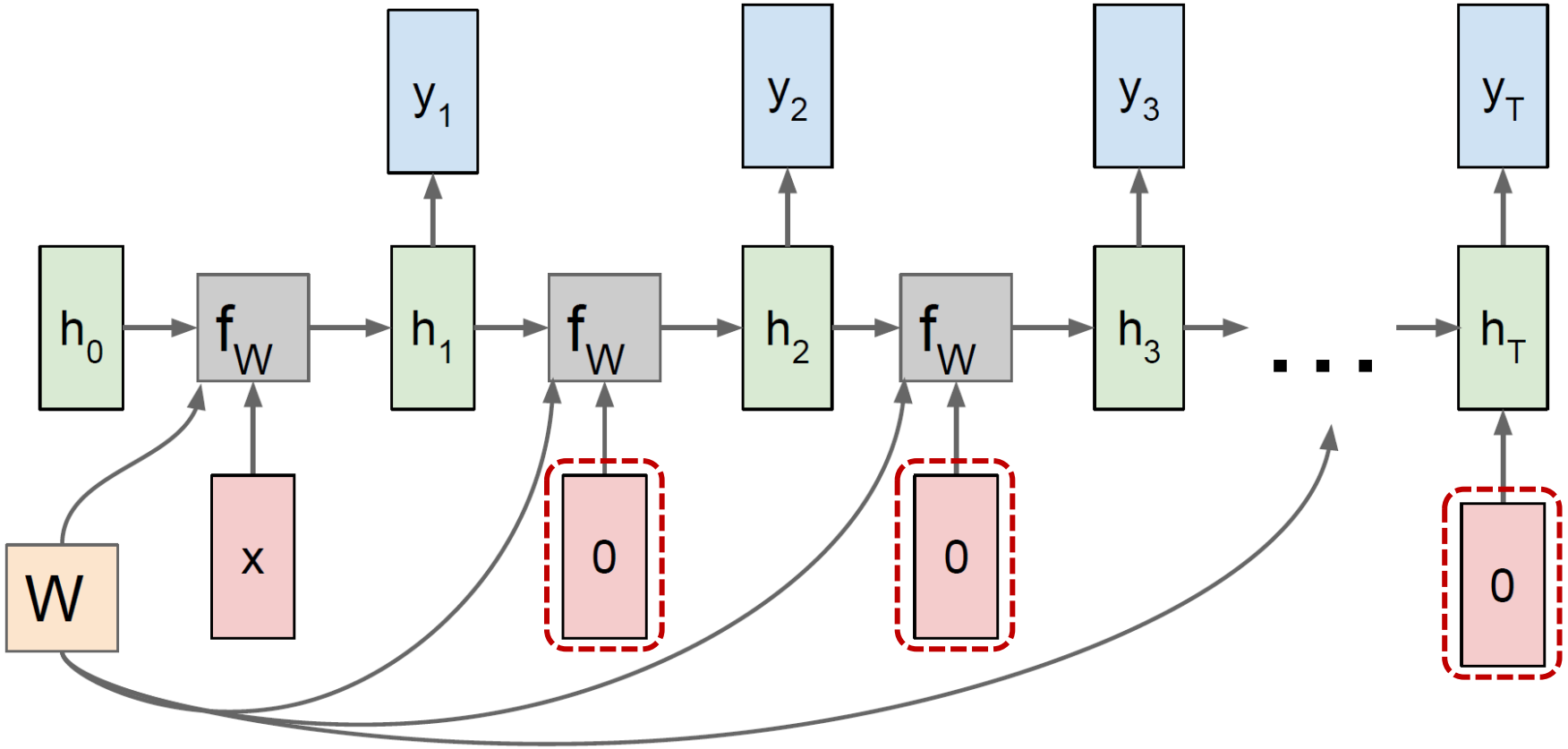
Computational Graph - Many-to-One

- Re-use the same weight matrix at every time-step



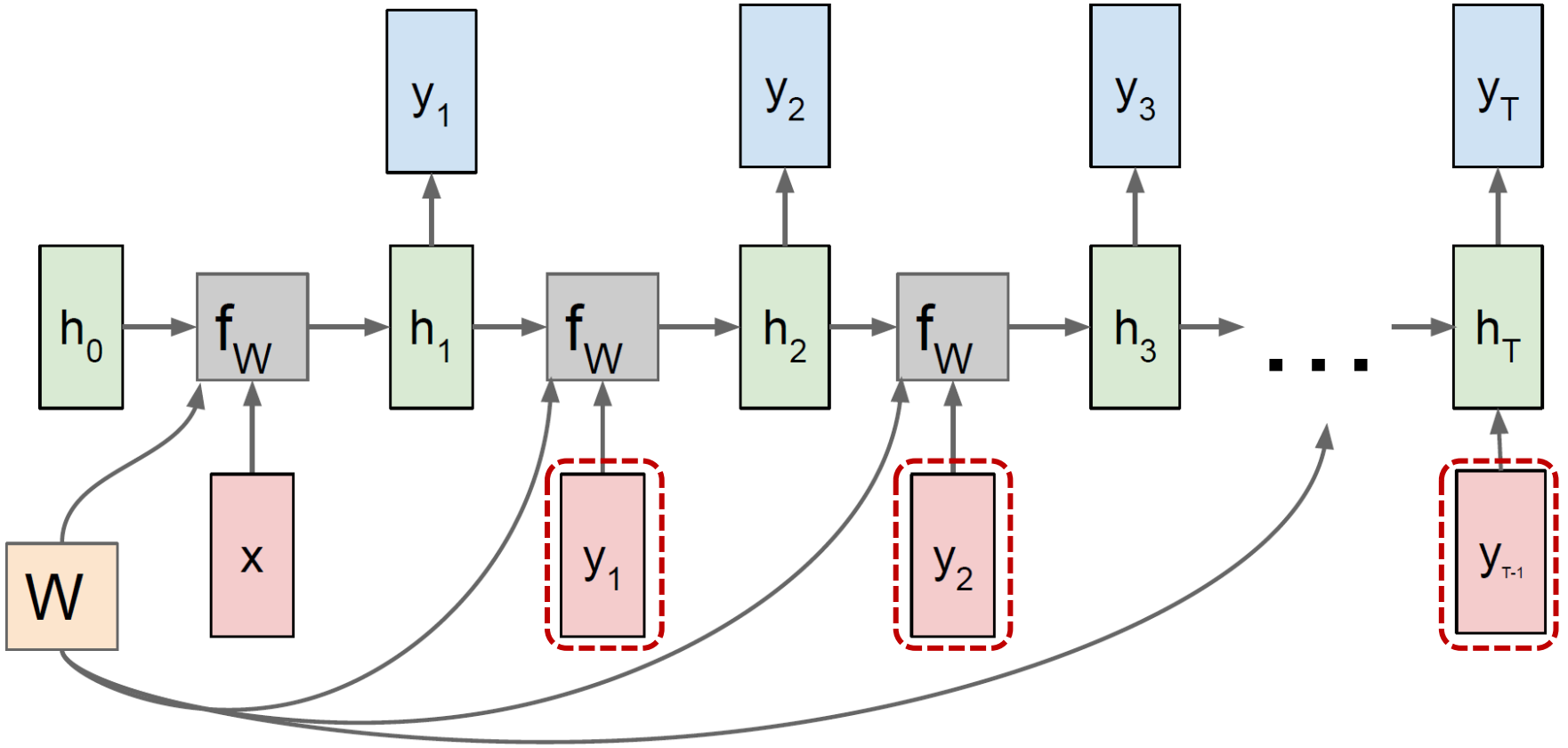
Computational Graph - One-to-Many

- Re-use the same weight matrix at every time-step



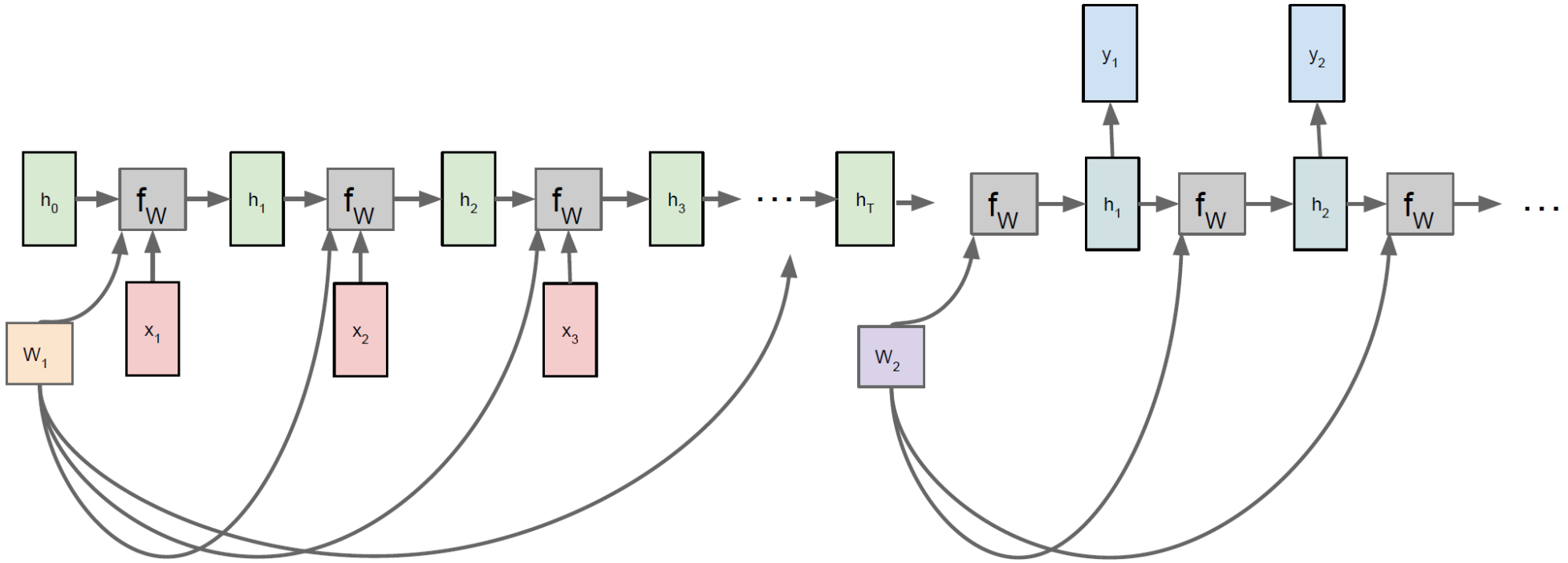
Computational Graph - One-to-Many

- Re-use the same weight matrix at every time-step



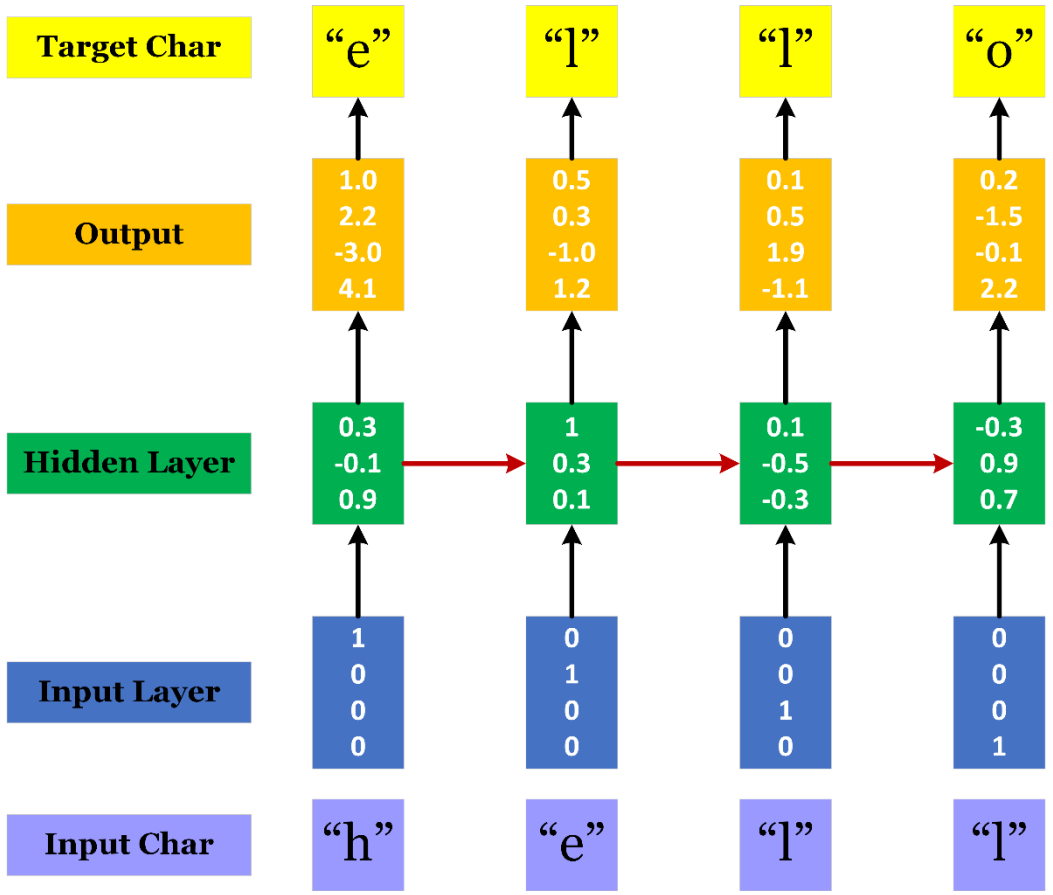
Sequence-to-Sequence (M2O+O2M)

- Many-to-One: *Encode* input sequence in a *single* vector.
- One-to-Many: Produce *output* sequence from *single* input vector.



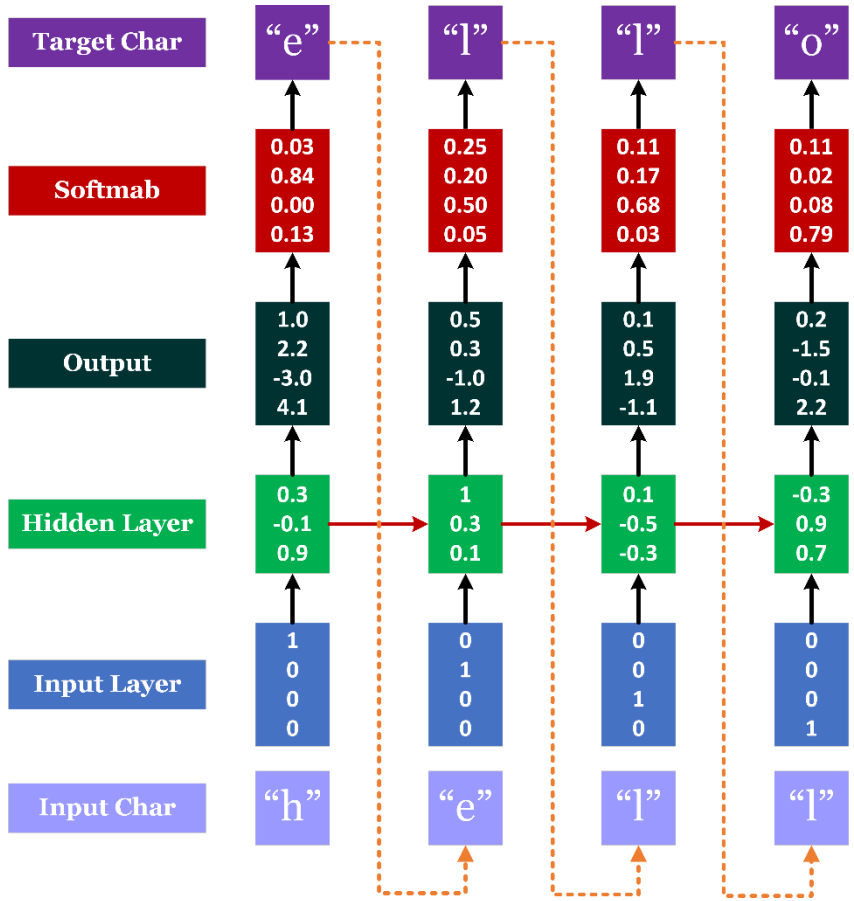
Seq2Seg Example - Train

- Character-Level Language Models:
- Vocabulary: [h,e,l,o]
- Example training sequence: "hello"



Seq2Seg Example - Inference

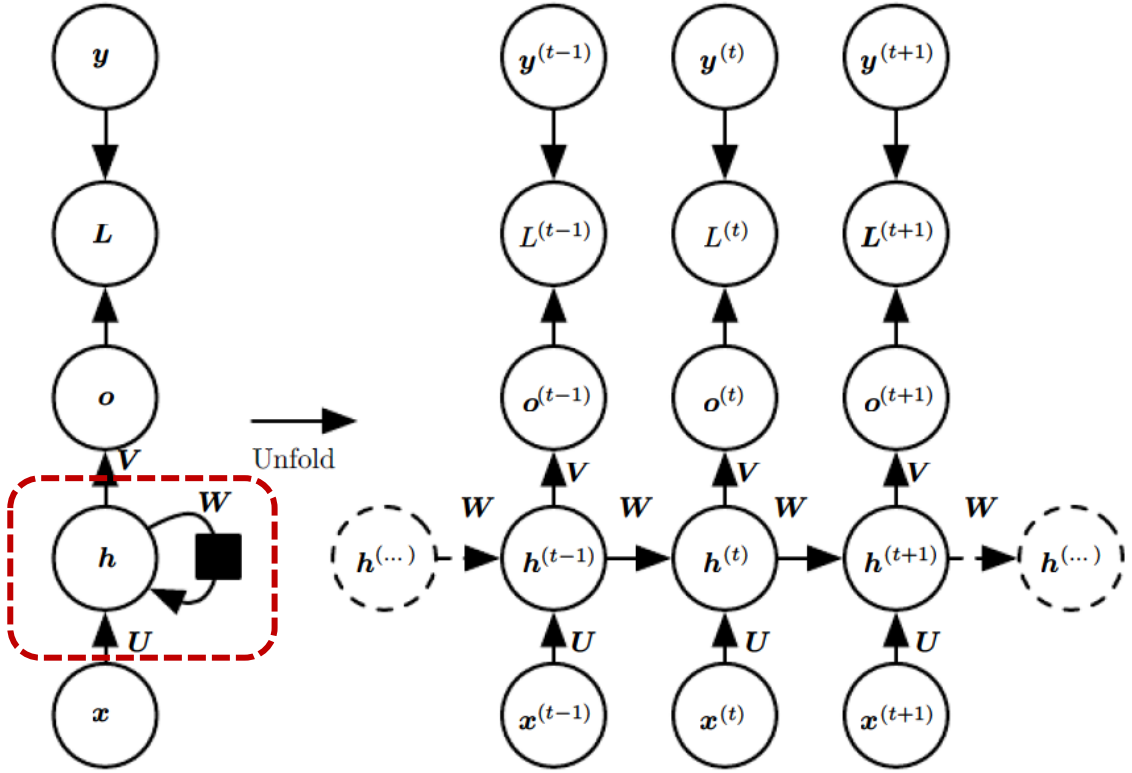
- Character-Level Language Models:
- Vocabulary: [h,e,l,o]
- At test-time sample characters one at a time *feedback* to model.



- See this interesting page!

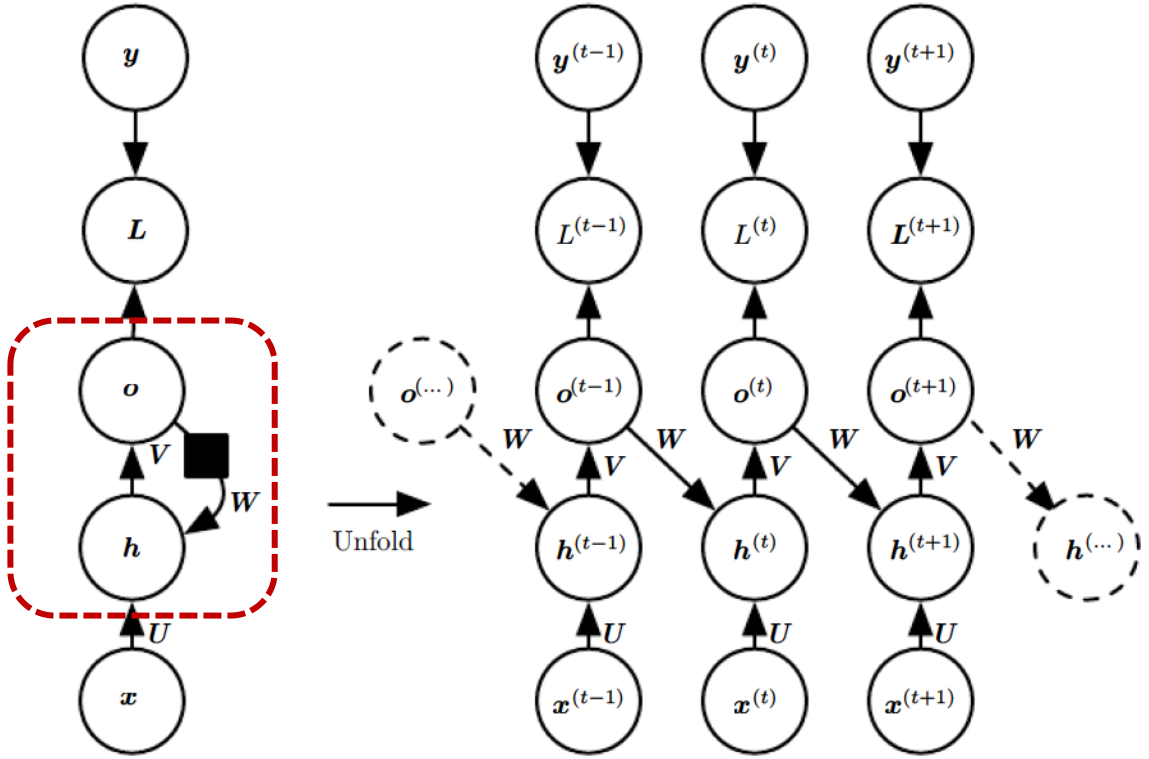
RNN - Conventional

- Unfolding



RNN – Teacher Force

- Unfolding



RNN – Loss

- Cumulative loss and update rules:

$$L\left(\{\boldsymbol{x}^{(1)},\dots,\boldsymbol{x}^{(T)}\},\{\boldsymbol{y}^{(1)},\dots,\boldsymbol{y}^{(T)}\}\right)=\sum_t L^{(t)}=-\sum_t logp_{model}\left(\boldsymbol{y}^{(t)}\middle|\{\boldsymbol{x}^{(1)},\dots,\boldsymbol{x}^{(t)}\}\right)$$

- For $t=1$ to $t=T$, we apply the following update equation:

$$\boldsymbol{a}^{(t)}=\boldsymbol{b}+\boldsymbol{W}\boldsymbol{h}^{(t-1)}+\boldsymbol{U}\boldsymbol{x}^{(t)}$$

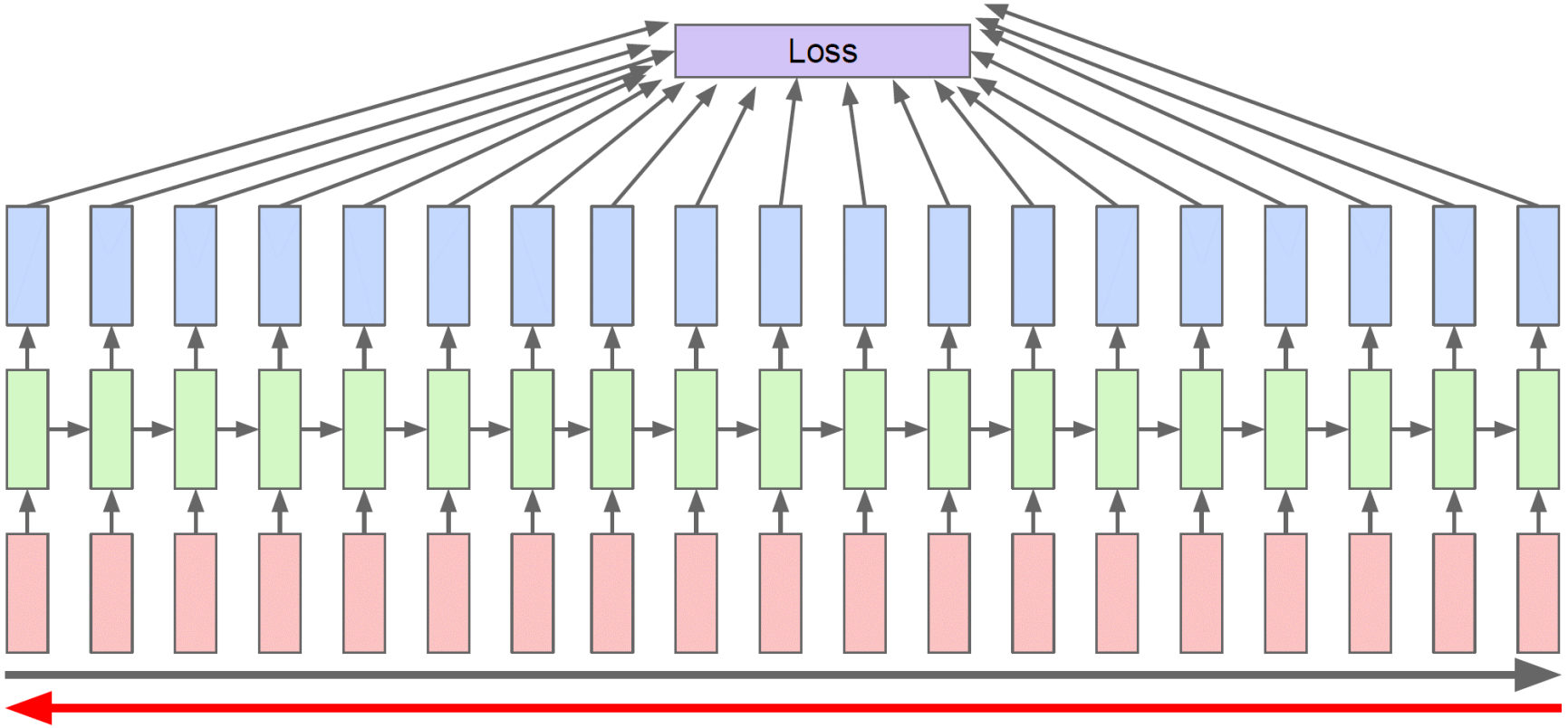
$$\boldsymbol{h}^{(t)}=tanh\left(\boldsymbol{a}^{(t)}\right)$$

$$\boldsymbol{o}^{(t)}=\boldsymbol{c}+\boldsymbol{V}\boldsymbol{h}^{(t)}$$

$$\hat{\boldsymbol{y}}^{(t)}=softmax\left(\boldsymbol{o}^{(t)}\right)$$

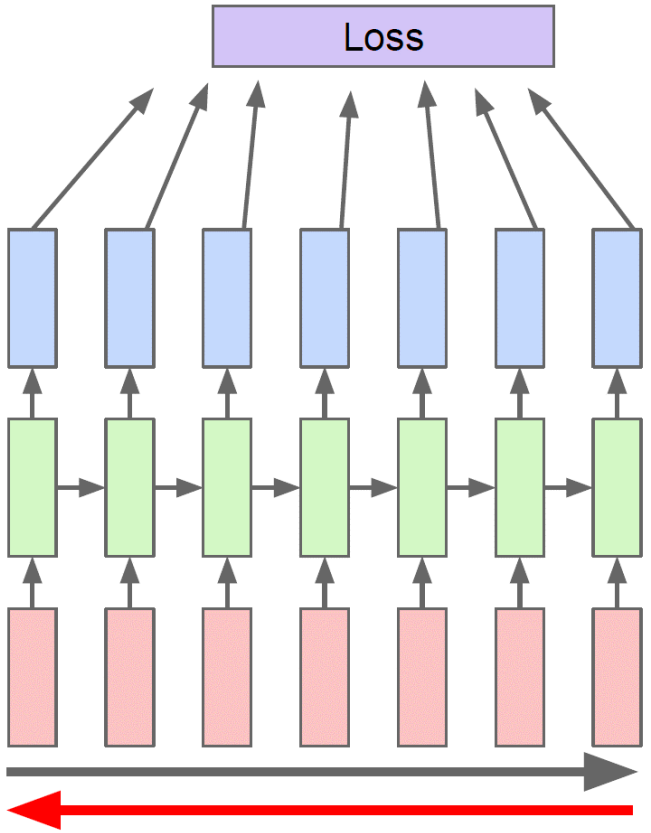
BPTT (Back Propagation Through Time)

- BTPP: Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient



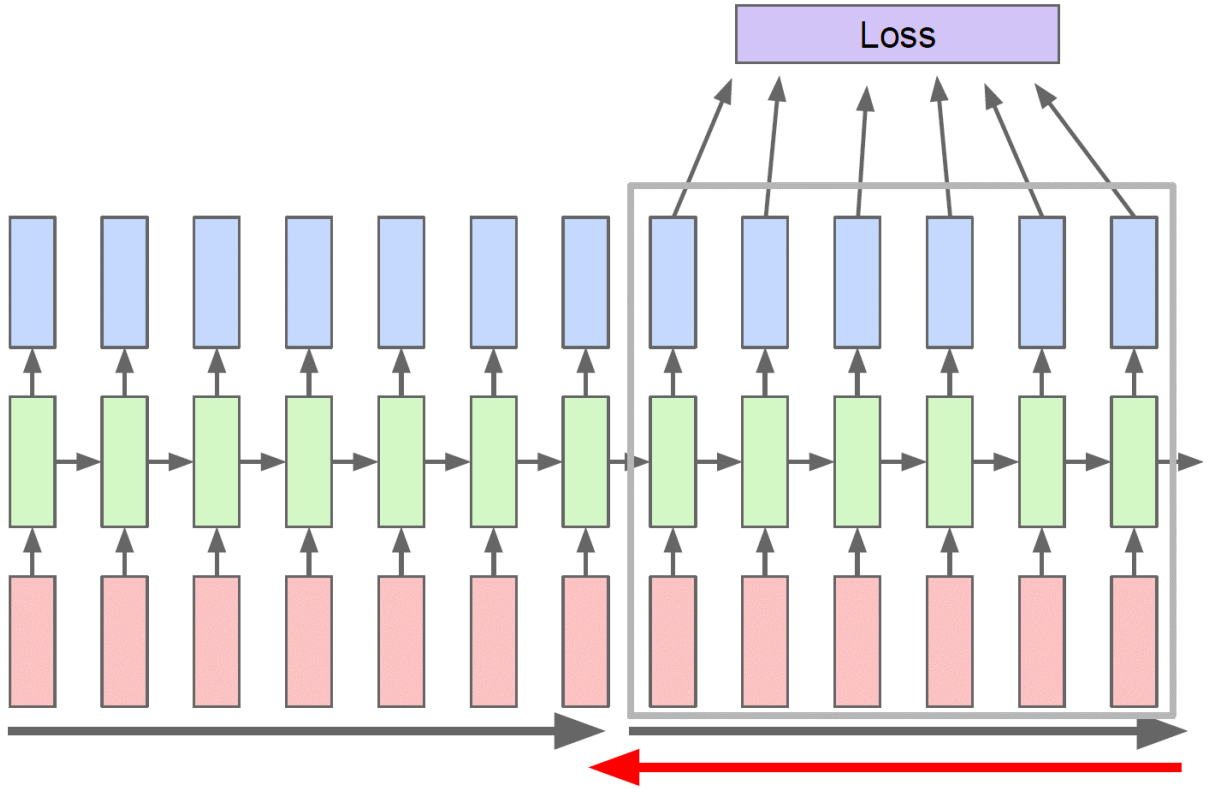
Truncated Backpropagation Through Time

- T-BPTT (option #1): Run forward and backward through pieces of the sequence instead of whole sequence



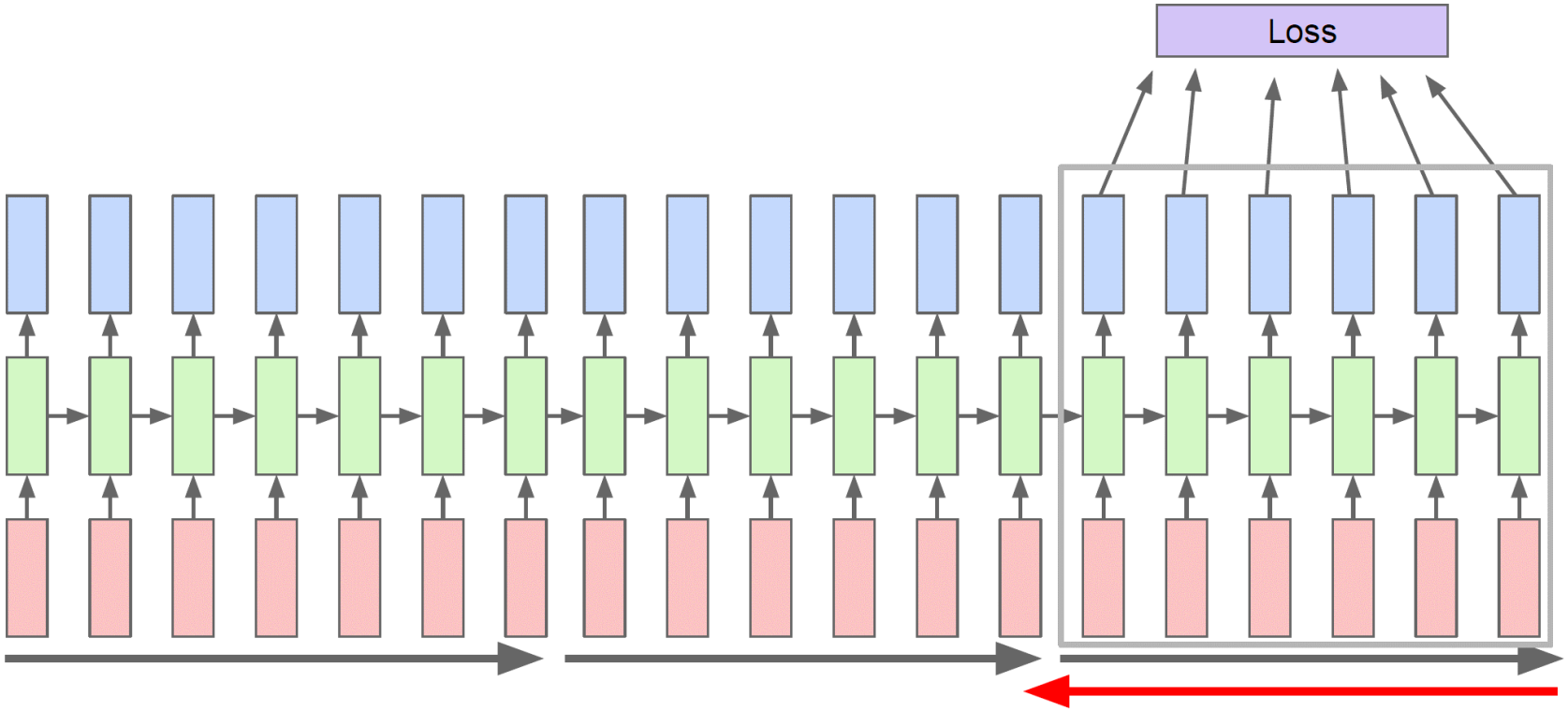
Truncated Backpropagation Through Time

- T-BPTT (option #2): Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps



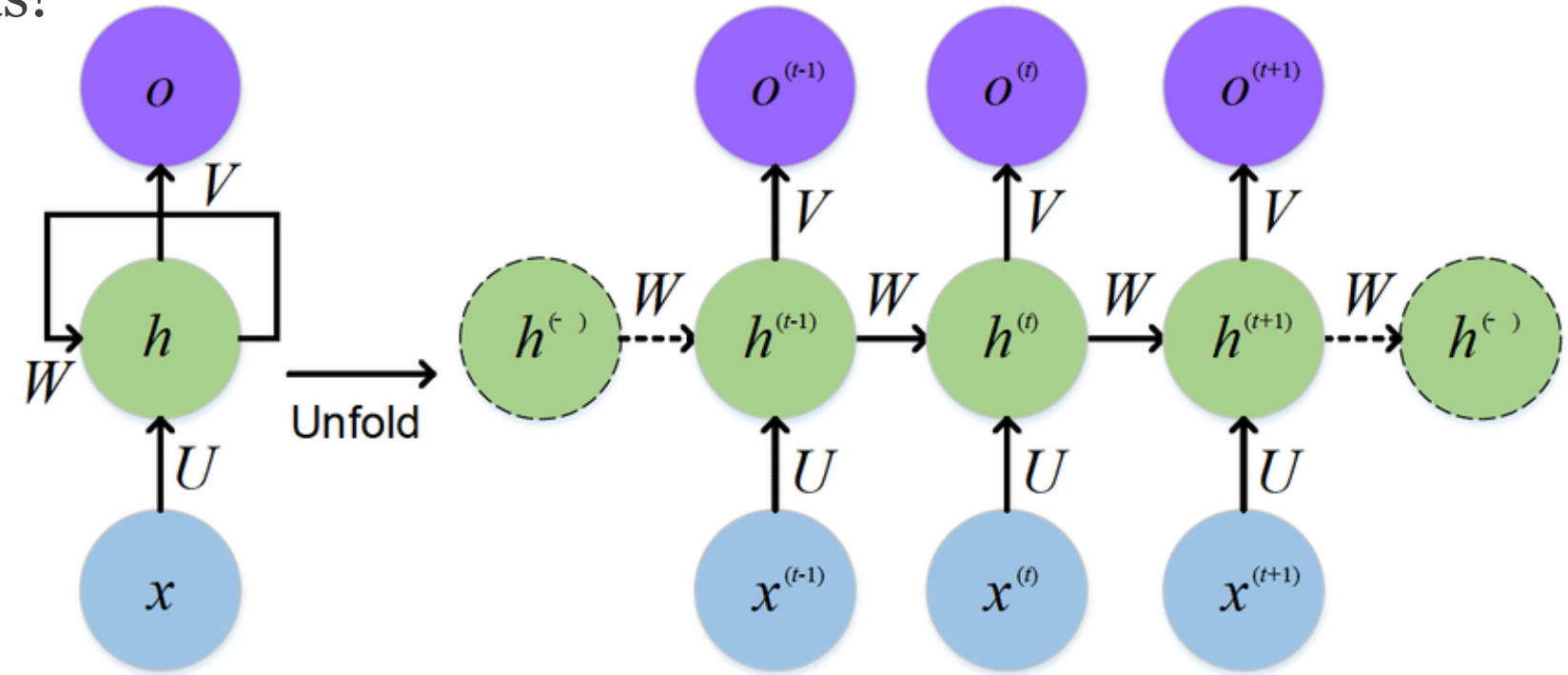
Truncated Backpropagation Through Time

- T-BPTT (option #3): Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps



BPTT (Backpropagation Through Time)

- MLP vs. Unfolded RNN:
- Shared Weights!



BPTT (Backpropagation Through Time)

- EBP in MLP (with previous notation):

$$\frac{\partial Loss(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial Loss(\boldsymbol{\theta})}{\partial z_{nj}^r}\frac{\partial z_{nj}^r}{\partial \boldsymbol{\theta}_j^r} = \frac{\partial Loss}{\partial z_{nj}^r}\boldsymbol{y}_n^{r-1}, \delta_{nj}^r \triangleq \frac{\partial Loss(\boldsymbol{\theta})}{\partial z_{nj}^r}$$

- Last Layer (L):

$$\delta_{nj}^L = f'\left(z_{nj}^L\right)L'\left(y_{nj}, f\left(z_{nj}^L\right)\right)$$

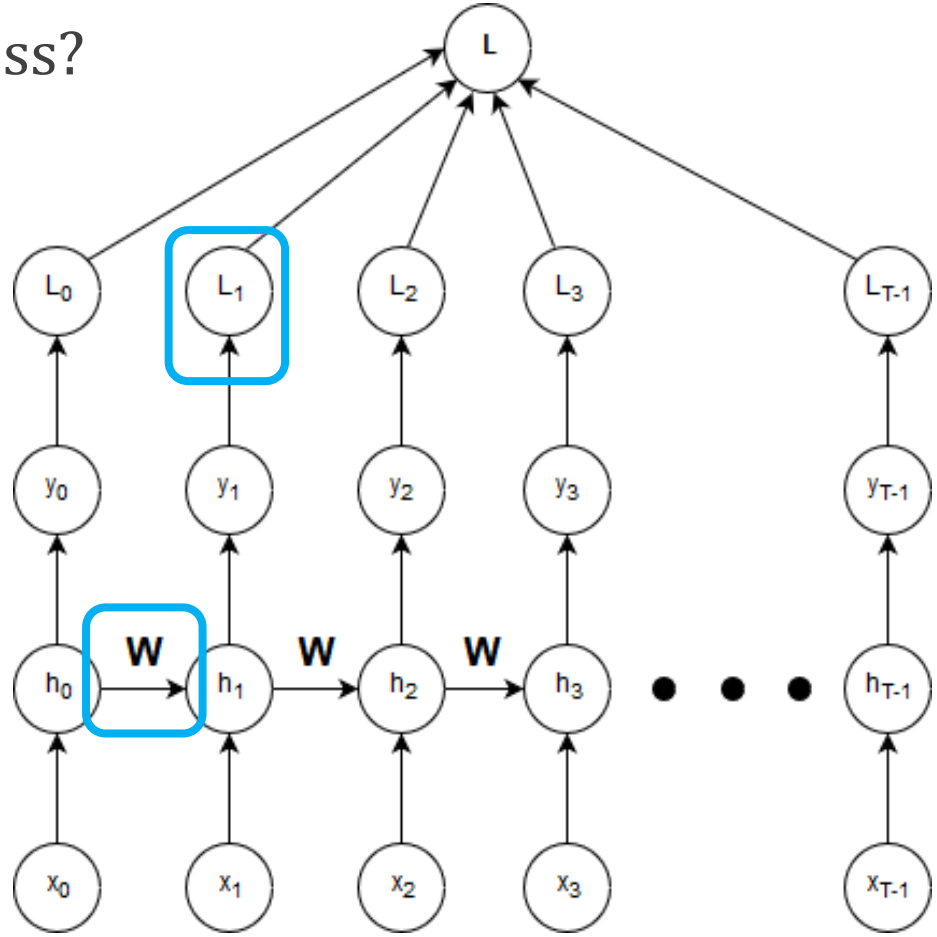
- Hidden Layer $r-1$, from r):

$$\delta_{nj}^{r-1} = \left(\sum_{k=1}^{k_r}\delta_{nk}^r\theta_{kj}^r\right)f'\left(z_{nj}^{r-1}\right),\ j = 1,2,\dots,k_{r-1}$$

- Just one backward path!

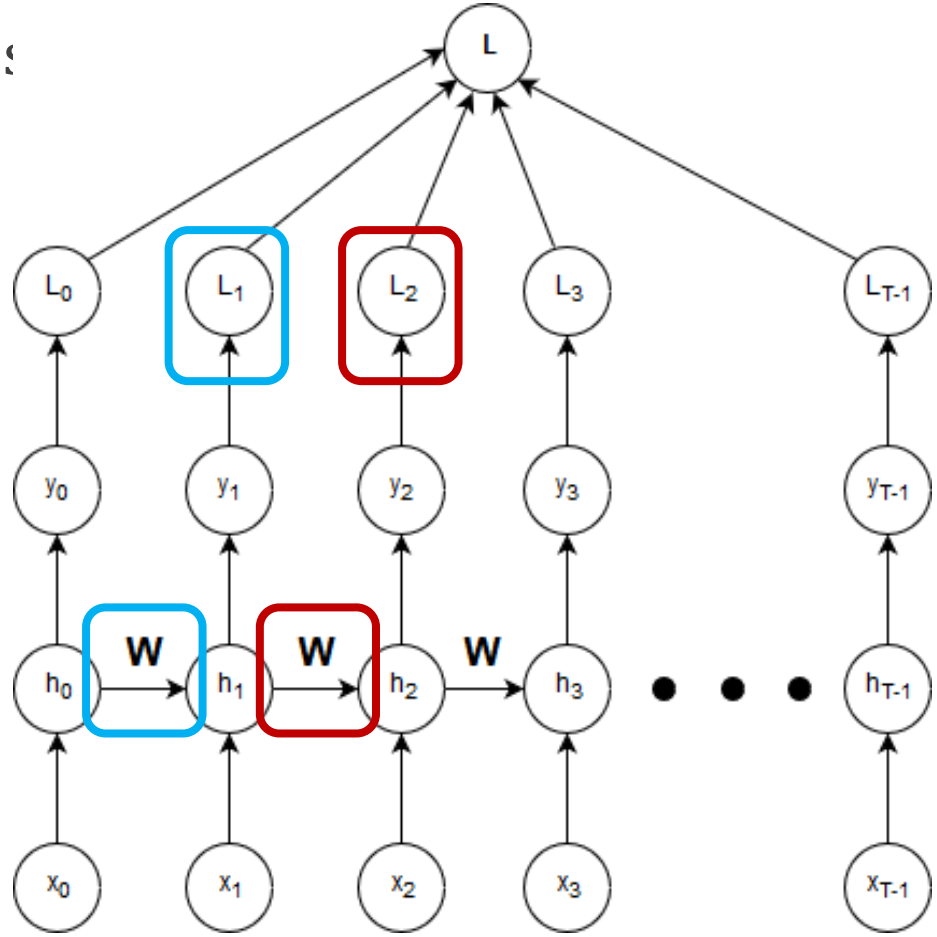
RNN - BPTT (Backpropagation Through Time)

- How Many path from weight(s) to temporal Loss?
  - Path #1 originate at h_0



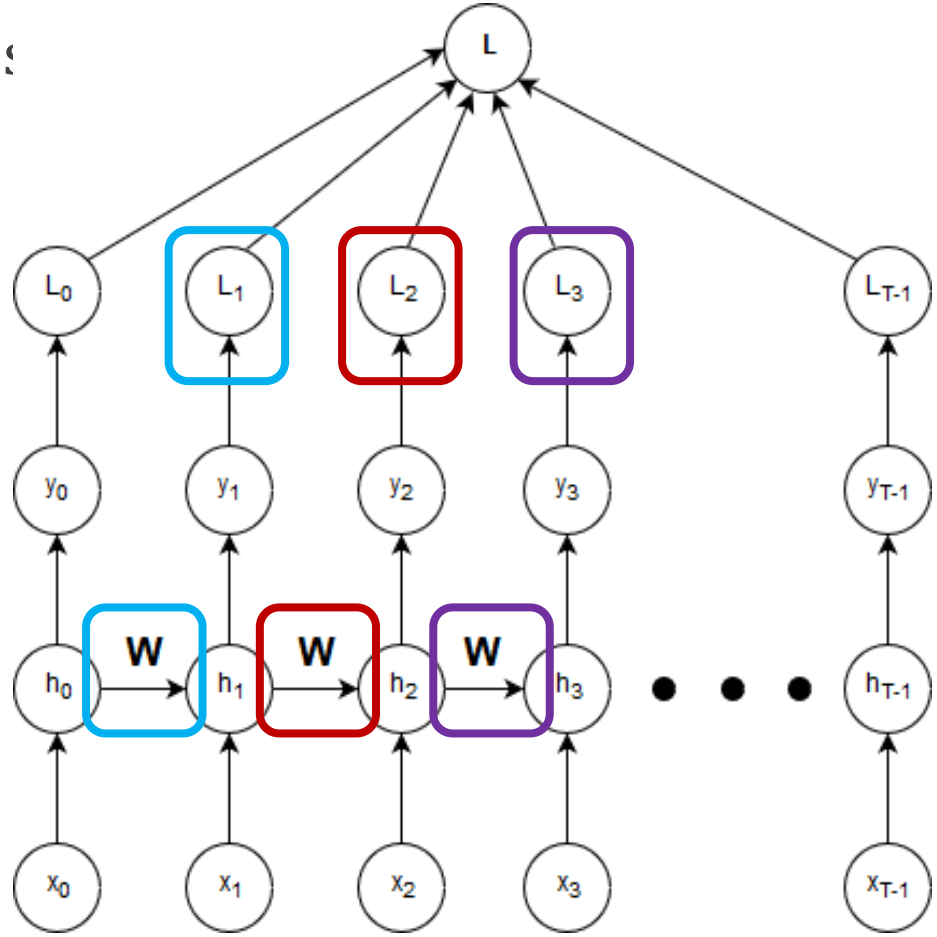
RNN - BPTT (Backpropagation Through Time)

- How Many path from weight(s) to temporal Los
  - Path #2 originate at h_0 and h_1



RNN - BPTT (Backpropagation Through Time)

- How Many path from weight(s) to temporal Los
  - Path #3 originate at h_0 , h_1 and h_2

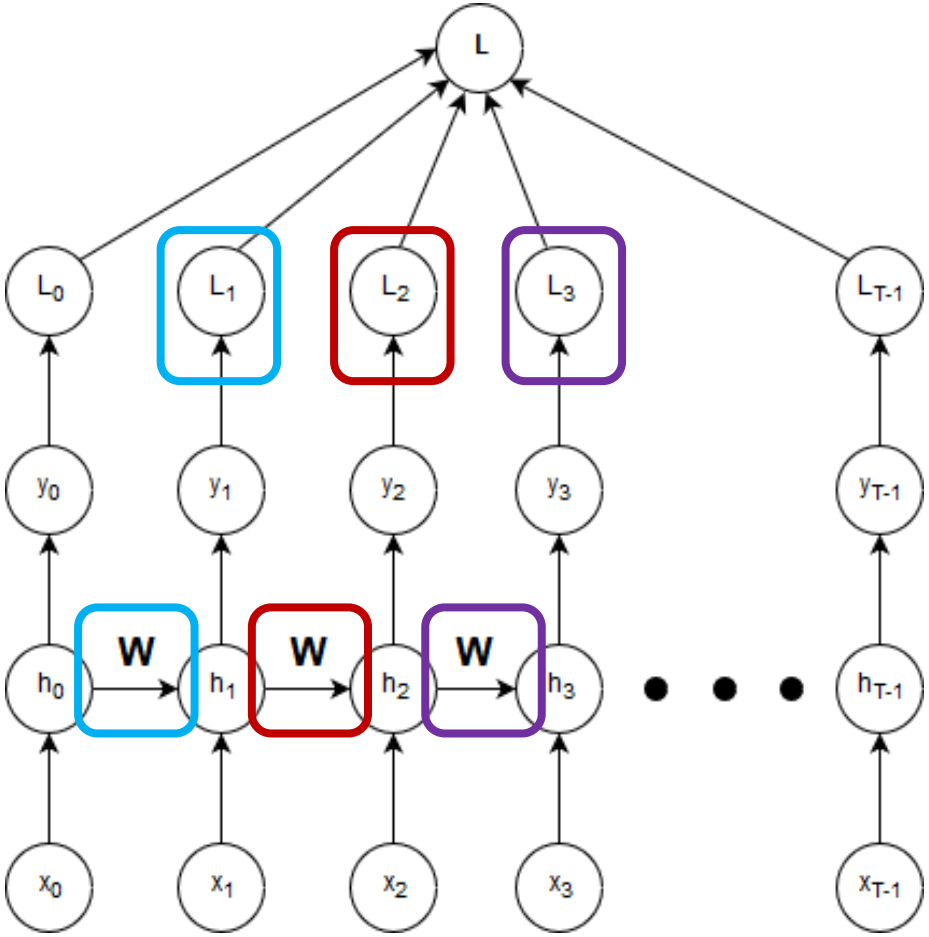


RNN - BPTT (Backpropagation Through Time)

- For EBP, we need

$$\frac{\partial L}{\partial W}$$

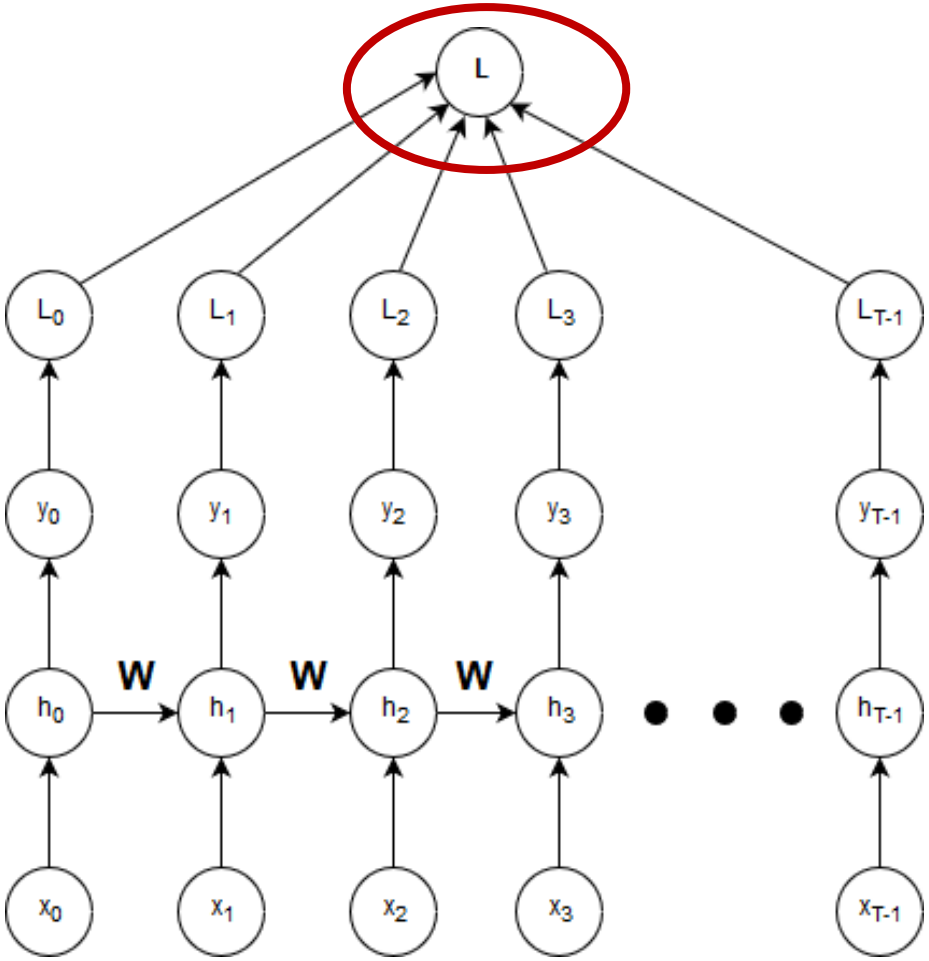
- There are two Summations:
  - Sum of temporal Losses
  - Sum of gradient path(s)



RNN - BPTT (Backpropagation Through Time)

- Temporal Loss(es)→ Total Loss:
- j: Temporal Index

$$\frac{\partial L}{\partial W} = \sum_{j=0}^{T-1} \frac{\partial L_j}{\partial W}$$

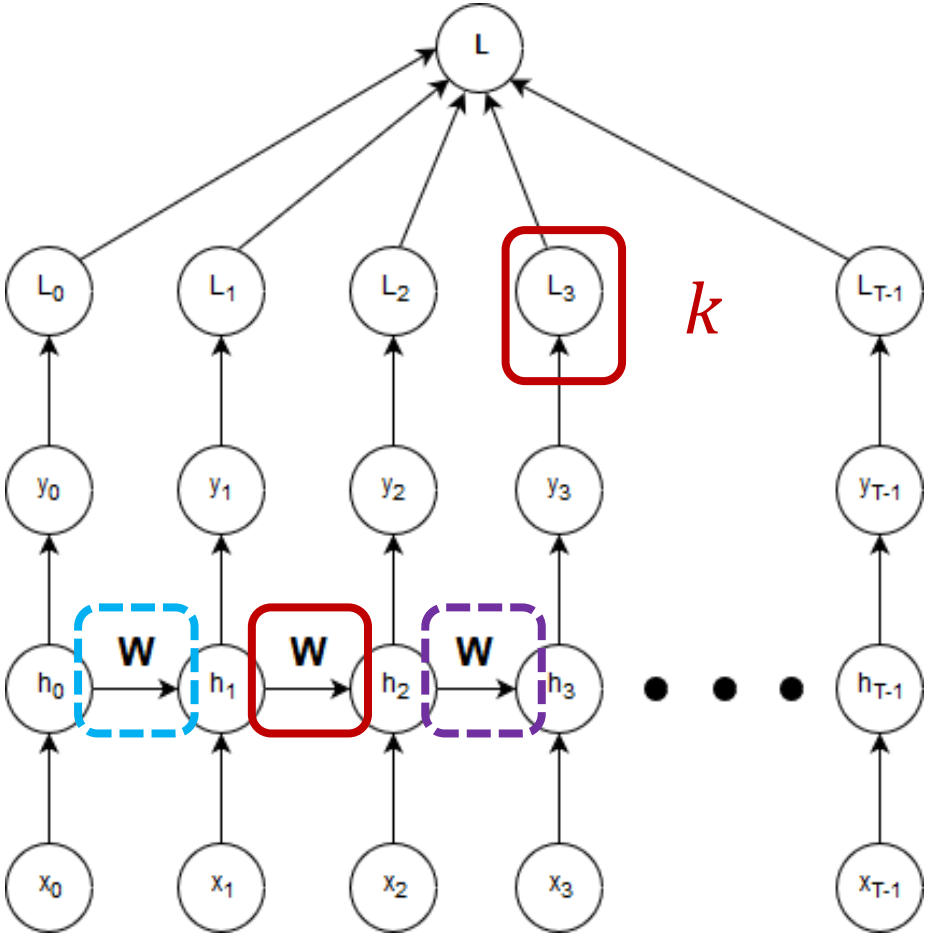


RNN - BPTT (Backpropagation Through Time)

- Gradient Summation:
  - Several dependency!
  - Indirect dependency!

$$\frac{\partial L}{\partial W} = \sum_{j=0}^{T-1} \frac{\partial L_j}{\partial W}$$

$$\frac{\partial L_j}{\partial W} = \sum_{k=1}^{j} \frac{\partial L_j}{\partial h_k} \frac{\partial h_k}{\partial W}$$



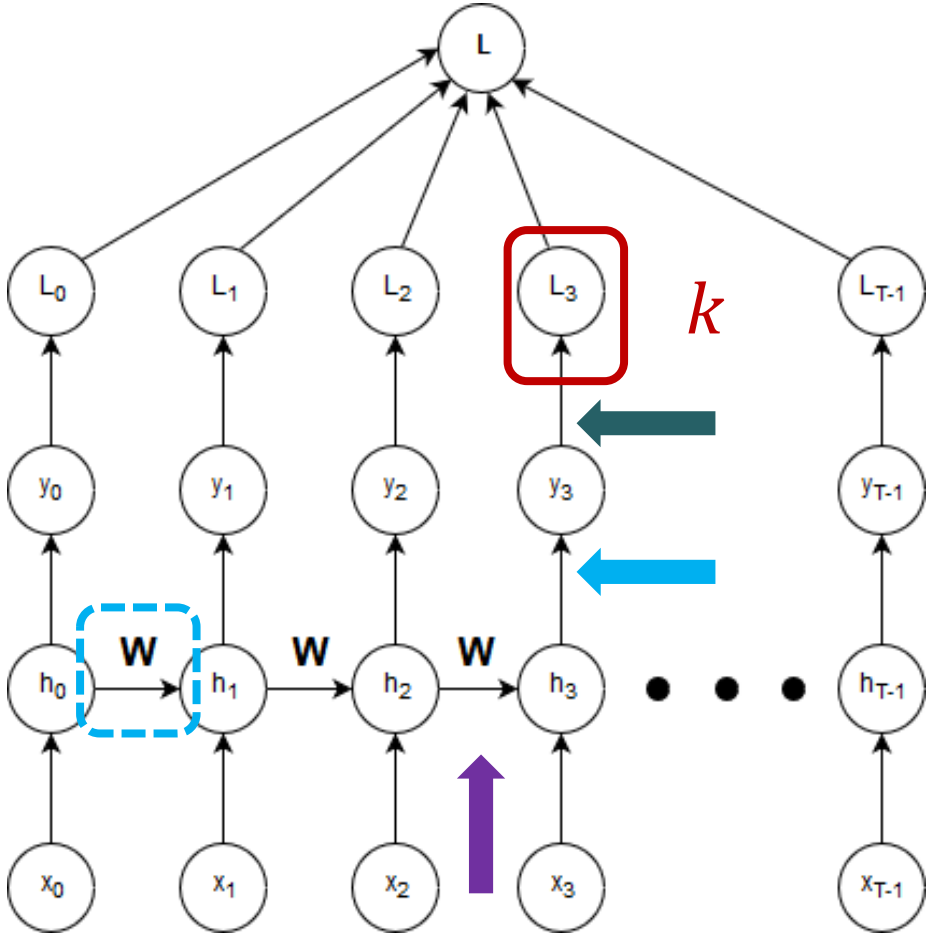
RNN - BPTT (Backpropagation Through Time)

- Chain Rule:
  - Convert to direct dependency!

$$\frac{\partial L_j}{\partial W} = \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \frac{\partial h_j}{\partial h_k} \frac{\partial h_k}{\partial W}$$

- Jacobian:

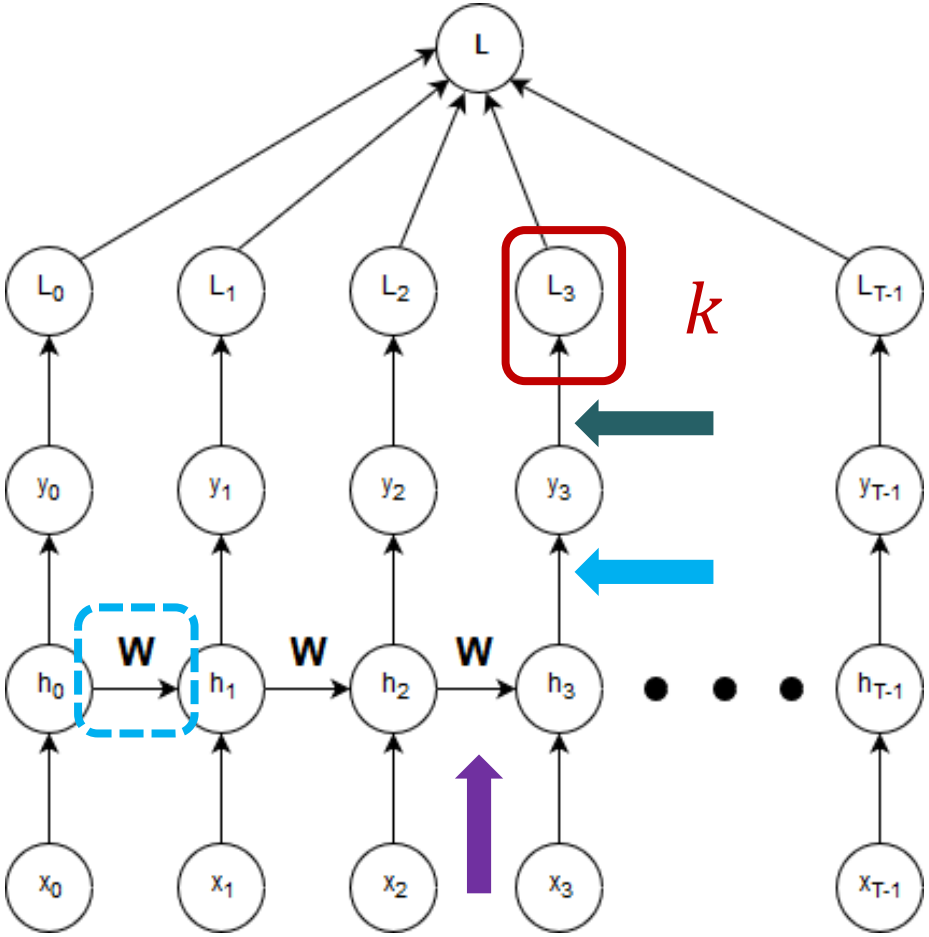
$$\frac{\partial h_j}{\partial h_k} = \prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}}$$



RNN - BPTT (Backpropagation Through Time)

- Final EBP:

$$\frac{\partial L}{\partial \boldsymbol{W}_h} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right) \frac{\partial h_k}{\partial \boldsymbol{W}_h}$$



RNN - BPTT (Backpropagation Through Time)

- Problem with considering all paths:
  - Memory
  - Gradient Flow (why?)

$$\frac{\partial L}{\partial \boldsymbol{W}_h} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right) \frac{\partial h_k}{\partial \boldsymbol{W}_h}$$

- Solution:
  - Truncate the network!

$$\frac{\partial L}{\partial \boldsymbol{W}_h} = \sum_{j=0}^{T-1} \sum_{k=j-p}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right) \frac{\partial h_k}{\partial \boldsymbol{W}_h}$$

RNN - BPTT (Backpropagation Through Time)

- Unwrap the state dependency:

$$\frac{\partial L}{\partial \boldsymbol{W}_h} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right) \frac{\partial h_k}{\partial \boldsymbol{W}_h}$$

$$h_m = f(\boldsymbol{W}_h h_{m-1} + \boldsymbol{W}_x x_m)$$

$$\frac{\partial h_m}{\partial h_{m-1}} = \boldsymbol{W}_h^T diag\left(f'(\boldsymbol{W}_h h_{m-1} + \boldsymbol{W}_x x_m)\right)$$

RNN - BPTT (Backpropagation Through Time)

- Dependency depth problem:

$$\frac{\partial L}{\partial \boldsymbol{W}_h} = \sum_{j=0}^{T-1} \sum_{k=1}^{j} \frac{\partial L_j}{\partial y_j} \frac{\partial y_j}{\partial h_j} \left(\prod_{m=k+1}^{j} \frac{\partial h_m}{\partial h_{m-1}} \right) \frac{\partial h_k}{\partial \boldsymbol{W}_h}$$

$$\frac{\partial h_j}{\partial h_k} = \prod_{m=k+1}^{j} \boldsymbol{W}_h^T diag\left(f'(\boldsymbol{W}_h h_{m-1} + \boldsymbol{W}_x x_m) \right)$$

- Long Term Dependency → Repeated Matrix Multiplication

Gradient Vanishing and Exploding ☹

RNN - BPTT (Backpropagation Through Time)

- For *ReLU* activation function:

$$\frac{\partial h_j}{\partial h_k} = \prod_{m=k+1}^{j} \boldsymbol{W}_h^T diag\left(f'(\boldsymbol{W}_h h_{m-1} + \boldsymbol{W}_x x_m)\right)$$

$$\frac{\partial h_j}{\partial h_k} = \left(\boldsymbol{W}_h^T\right)^n = \boldsymbol{Q}^{-1}\boldsymbol{\Lambda}^n\boldsymbol{Q}$$

- Long Term Dependency → Repeated Matrix Multiplication

Gradient Vanishing and Exploding ☹

RNN – Gradient Vanishing and Exploding

- A numerical example:

$$\left(\boldsymbol{W}_h^T\right)^n = \boldsymbol{Q}^{-1}\boldsymbol{\Lambda}^n\boldsymbol{Q}$$

$$\boldsymbol{\Lambda} = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix} \Rightarrow \boldsymbol{\Lambda}^{10} = \begin{bmatrix} 0.5 & 0 \\ 0 & 1.5 \end{bmatrix}^{10} = \begin{bmatrix} 0.00098 & 0 \\ 0 & 57.665039 \end{bmatrix}$$

- A Good reference: Bengio et al,. "On the difficulty of training recurrent neural networks." (2012)

RNN – Initialization

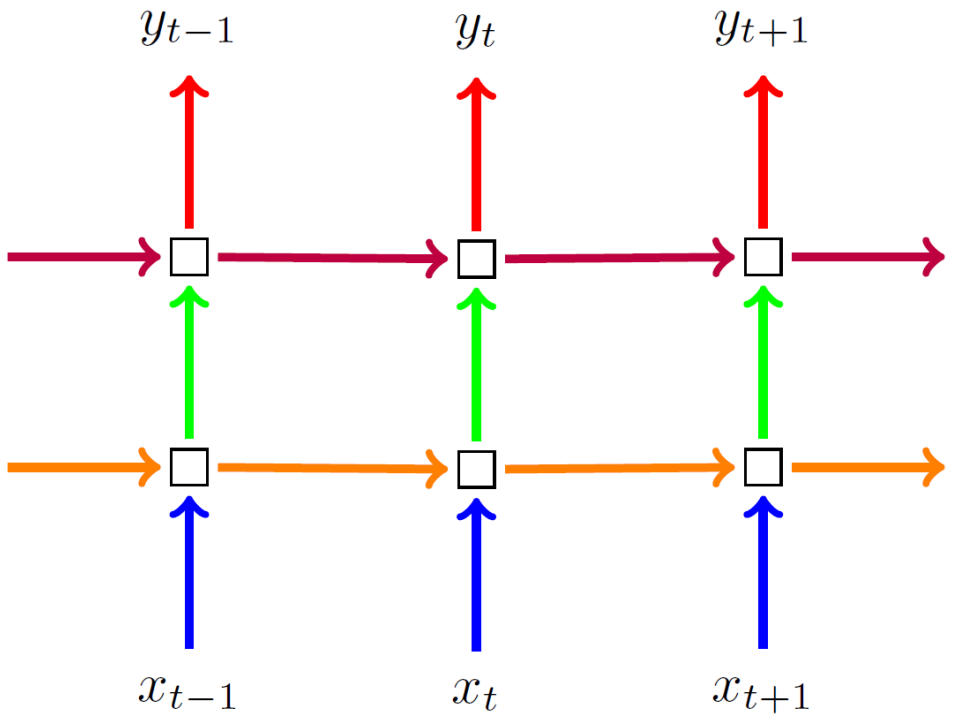
- Initialize with suitable eigenvalues.
- Most used initialization methods:
  - np-RNN: random pdm with all except the highest eigenvalue less than one (atleast one eigenvalue is equal to 1)
  - u-RNN: Based on unitary matrix
- Papers:
  - Improving Performance of Recurrent Neural Network With ReLu Nonlinearity, ICLR2016
  - Unitary Evolution Recurrent Neural Networks, ICML2016

RNN - Exploding

- Solution #1 (Gradient Clipping):
  - $\hat{\boldsymbol{g}} \leftarrow \frac{\partial Loss}{\partial \boldsymbol{W}}$
  - if $\|\hat{\boldsymbol{g}}\| > threshold$ then $\hat{\boldsymbol{g}} \leftarrow \frac{threshold}{\|\hat{\boldsymbol{g}}\|}\hat{\boldsymbol{g}}$
- Solution #2 (unity feedback gain, ResBlock idea):
  - $\boldsymbol{h}_t = \boldsymbol{h}_{t-1} + F(\boldsymbol{x}_t) \Longrightarrow \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{h}_{t-1}} = 1$

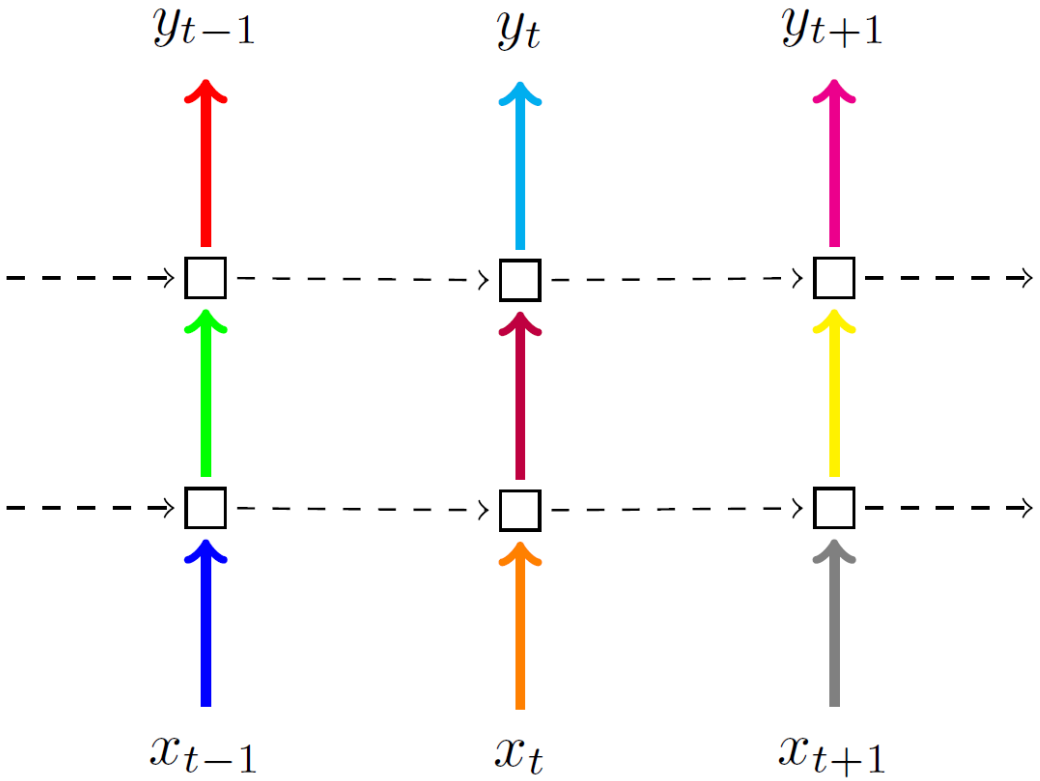
RNN Dropout - A Theoretically Grounded Application of Dropout in Recurrent Neural Networks – NIPS 2016

- Variational Dropout:
  - All Connection (including recurrent) but each timestep applies the same dropout mask, different colors corresponding to different dropout masks



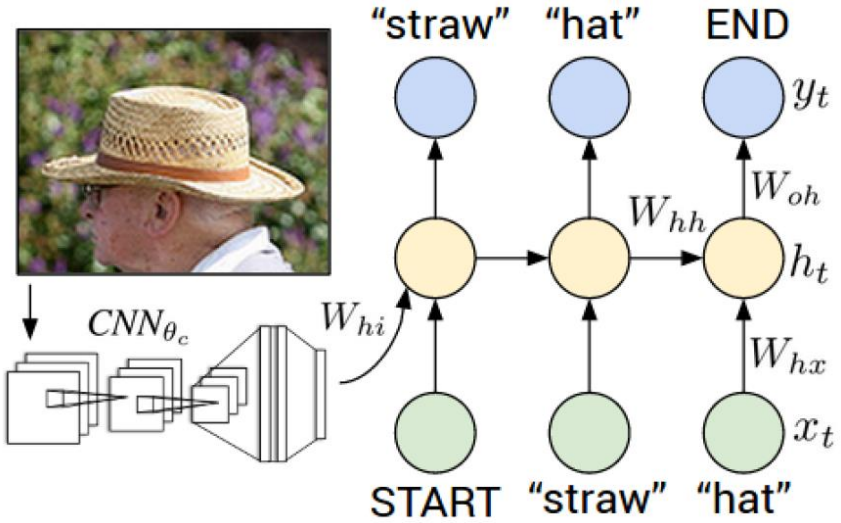
RNN Dropout - A Theoretically Grounded Application of Dropout in Recurrent Neural Networks – NIPS 2016

- Conventional RNN-Dropout to avoid loss memory:
  - Only Apply on non-recurrent connections (Solid/Color vs dashed connections)



Example – Image Captioning

- Training:
  - Input: An image (feature vectors,FC4094) and a sequence of input vectors $\{\boldsymbol{x}_1, \dots, \boldsymbol{x}_T\}$
  - It compute a sequence of hidden states $\{\boldsymbol{h}_1, \dots, \boldsymbol{h}_T\}$ and a sequence of outputs $\{\boldsymbol{y}_1, \dots, \boldsymbol{y}_T\}$
  - $\boldsymbol{b_v} = \boldsymbol{W}_{hi}[CNN_{\boldsymbol{\theta}_c}(I)]$
  - $\boldsymbol{h}_t = f(\boldsymbol{W}_{hx}\boldsymbol{x}_t + \boldsymbol{W}_{hh}\boldsymbol{h}_{t-1} + \boldsymbol{b}_n + \mathbb{I}(t=1) \odot \boldsymbol{b_v})$
  - $y_t = softmax(\boldsymbol{W}_{oh}\boldsymbol{h}_t + \boldsymbol{b}_o)$
  - Learnable parameters: $\boldsymbol{W}_{hi}, \boldsymbol{W}_{hx}, \boldsymbol{W}_{hh}, \boldsymbol{W}_{oh}, \boldsymbol{b}_n, \boldsymbol{b}_o$
  - Image context only proved in first step (t=1)
- Test:
  - A word sampled from the output (or pick the argmax) set its embedding vector as $\boldsymbol{x}_2$, and repeat this process until the END token is generated.



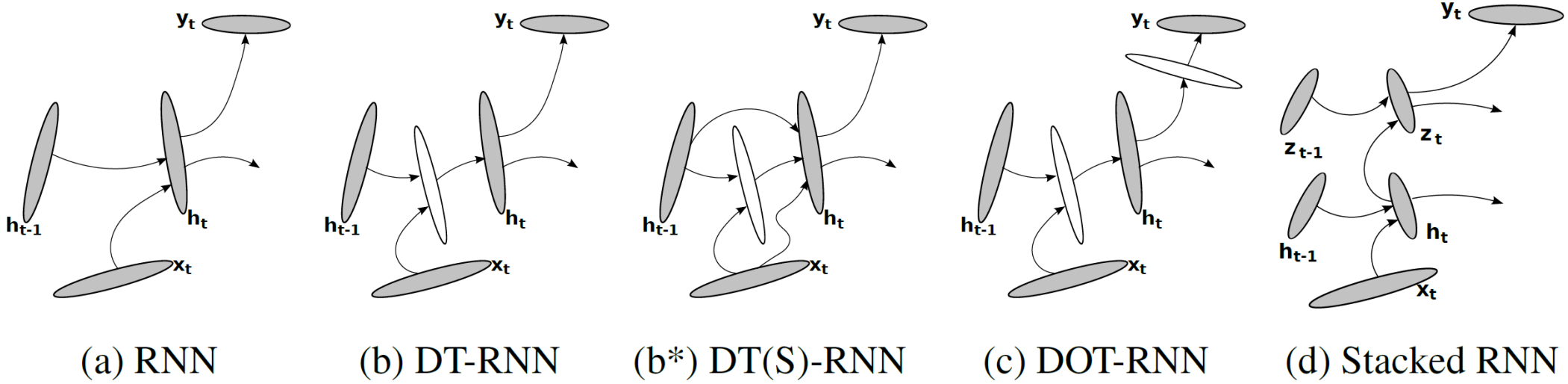
RNN – How to Deepen RNN

- Input to the Hidden state, (Convert weight matrix to a network), W_{xh}
  - Higher level representation
- Recurrent Layer, W_{hh}
  - Add high nonlinearity (+temporal dependency)
  - State and input information (+ to concatenate)
- Hidden state to Output layer, W_{hy}
  - Facilitate the output prediction,

$$h_t = f_W(h_{t-1}, x_t)$$
$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$y_t = W_{hy}h_t$$

RNN Deepening

- There are several possibility:

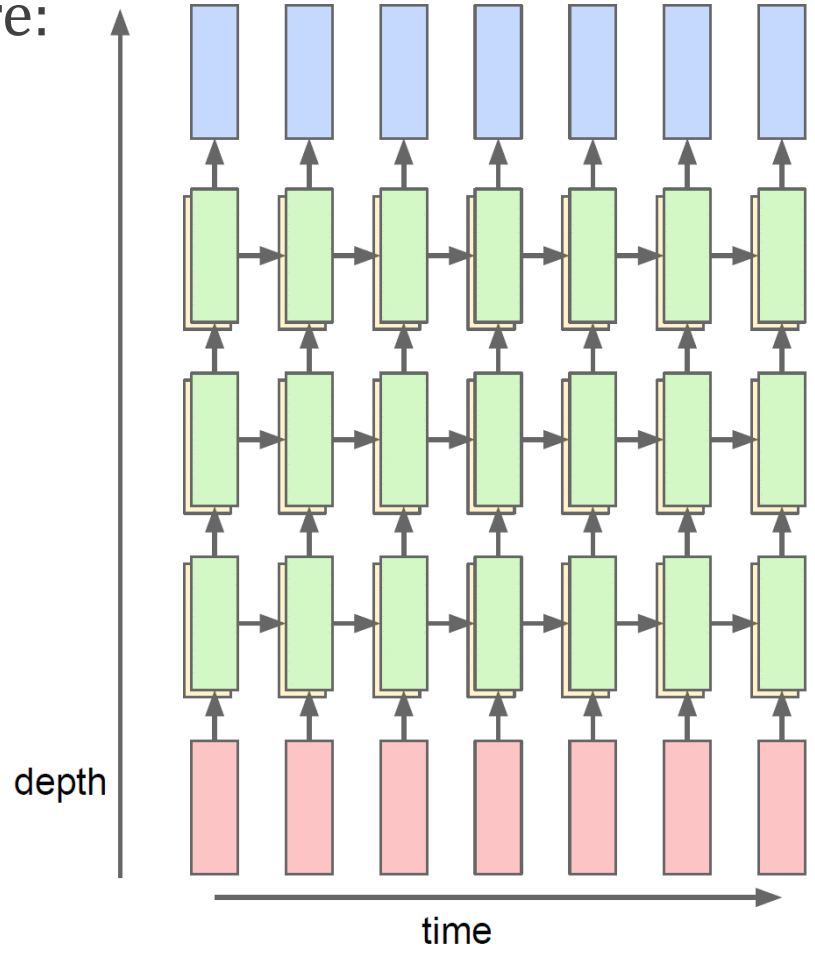


(a) RNN (b) DT-RNN (b*) DT(S)-RNN (c) DOT-RNN (d) Stacked RNN

Figure 2: Illustrations of four different recurrent neural networks (RNN). (a) A conventional RNN. (b) Deep Transition (DT) RNN. (b*) DT-RNN with shortcut connections (c) Deep Transition, Deep Output (DOT) RNN. (d) Stacked RNN

Deep (Multilayer) RNN

- Most used architecture:



RNN Advantages and Drawbacks

- Advantages:
  - Possibility of processing input of *any* length
  - Model size not increasing with size (time course) of input
  - Computation takes into account *historical* information
  - Weights are *shared* across time
- Drawbacks:
  - Computation being slow
  - Difficulty of accessing information from a *long time* ago
  - Cannot consider any *future* input for the current state

Long Short-Term Memory

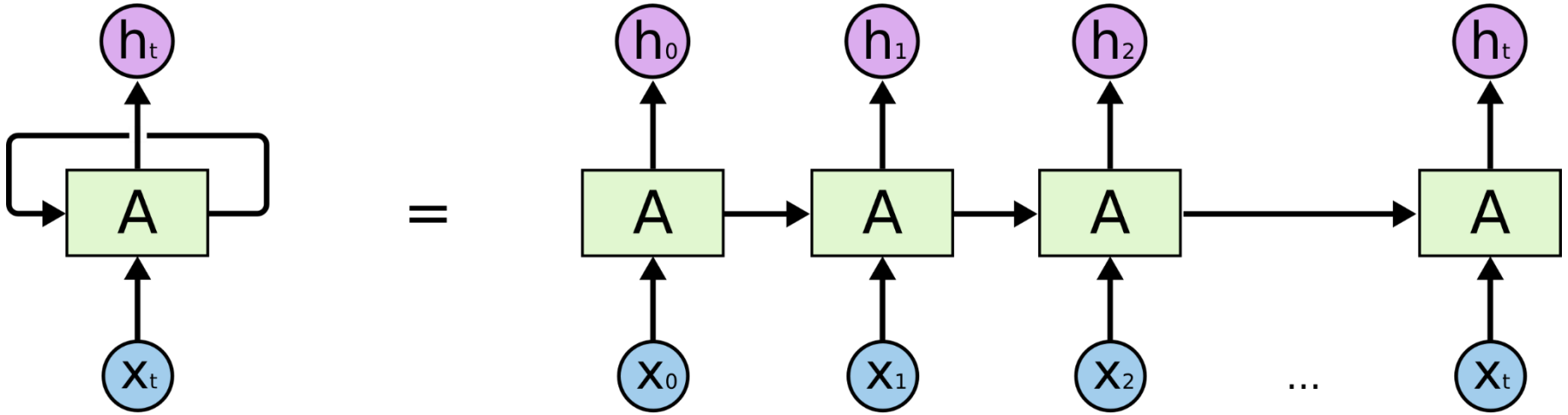
- First Generation: Long Short-Term Memory, *Neural Computation*, 1997.
- To be continued! Gated Recurrent Unit (GRU), 2014, and …
- Aims/Idea:

A memory cell that is not subject to matrix multiplication, thereby avoiding gradient decay

RNN – Remember

- Vanila RNN

$$h_t = f_{\boldsymbol{W}}(h_{t-1}, x_t)$$
$$h_t = f_{\boldsymbol{W}}(\boldsymbol{W}_{hh} h_{t-1} + \boldsymbol{W}_{xh} x_t)$$
$$y_t = \boldsymbol{W}_{hy} h_t$$



LSTM and GRU

DEEP LEARNING, FALL 2023

SHARIF UNIVERSITY OF TECHNOLOGY

LSTM

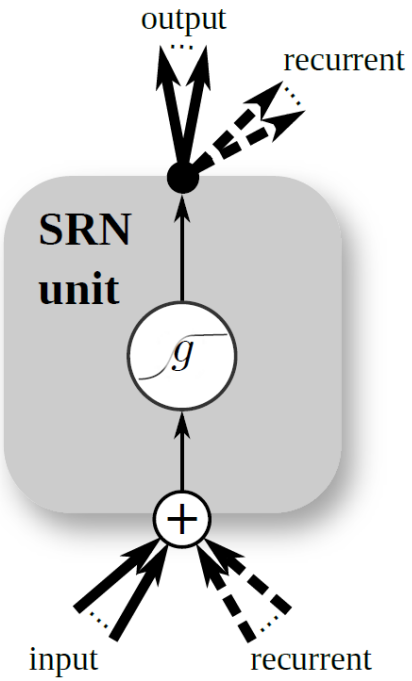
- Motivation: Memory Block Operation!
  - Read, How much?
  - Write, How much?
  - Reset , How much?
- "How much?" implementation
  - Read: input gate (if closed, overwrite by new input is not possible)
  - Write: output gate
  - Reset: clear saved memory

Simple Recurrent Network (SRN)

- Detailed schematic of the Simple Recurrent Network (SRN)

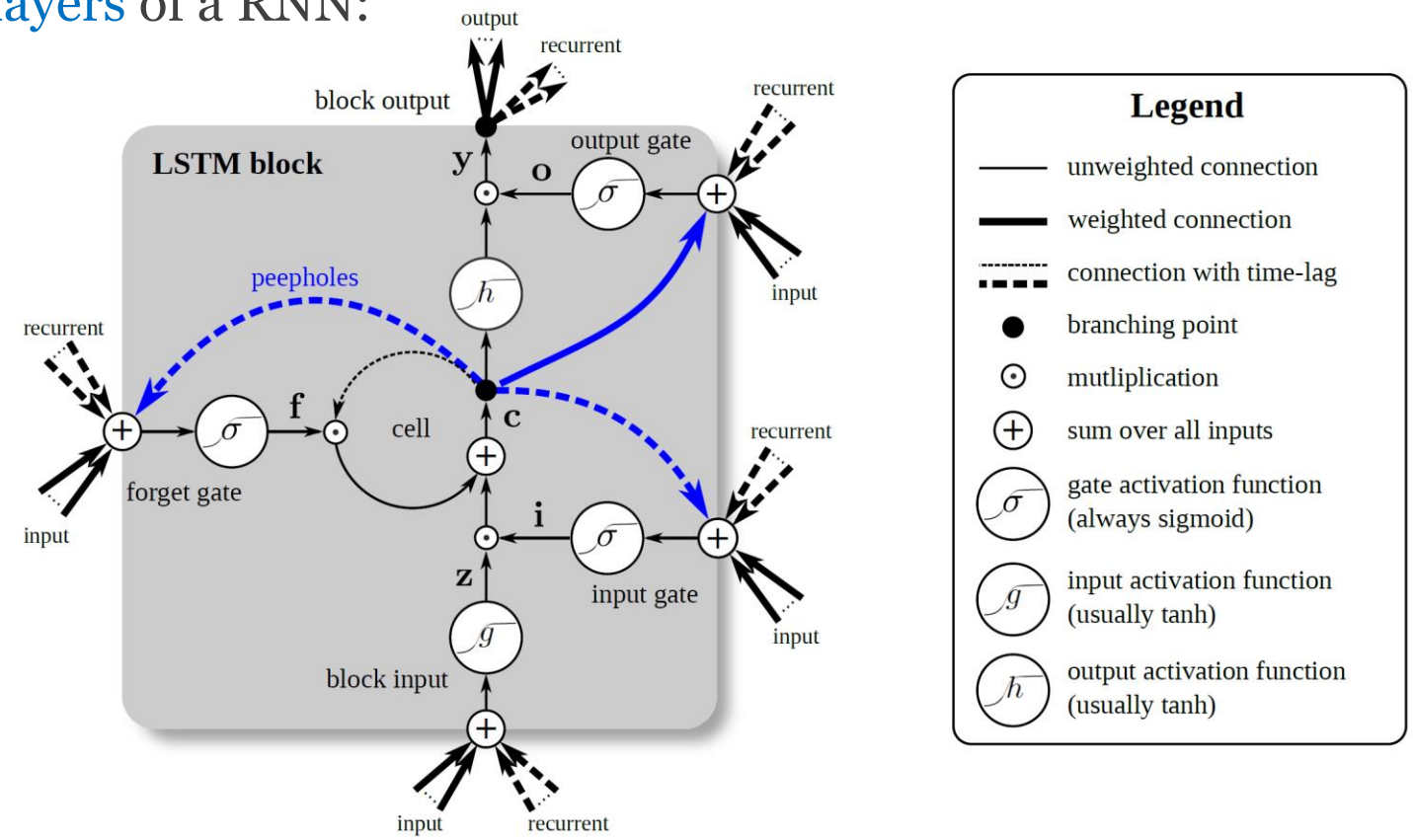
$$\mathbf{y}_t = tanh(\boldsymbol{W}_{hh}\mathbf{y}_{t-1} + \boldsymbol{W}_{xh}x_t)$$

- Here $\mathbf{y}_t$ is state.



LSTM Details

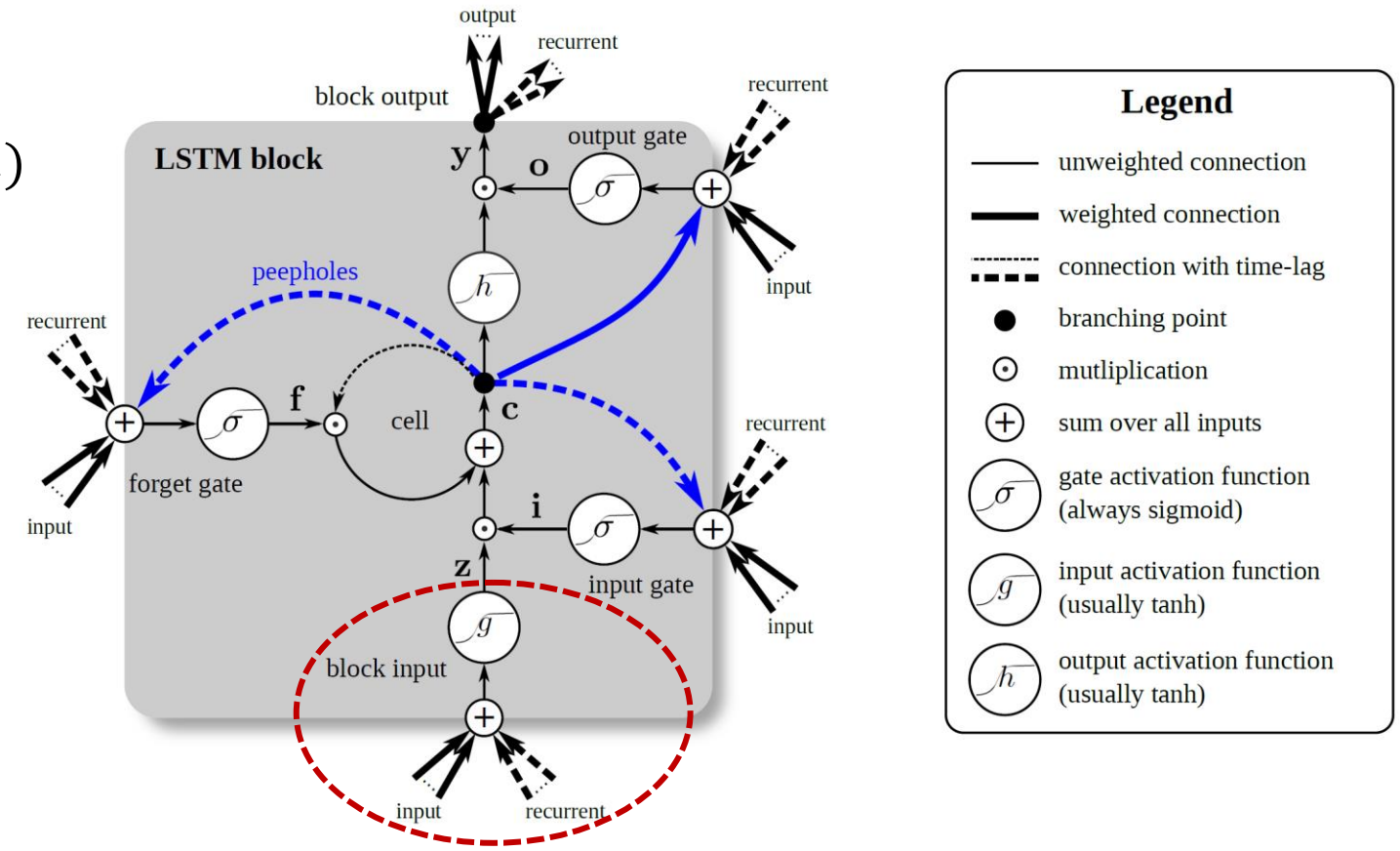
- LSTM as used in the hidden layers of a RNN:
- **x**: input
- **y**: output
- **i**: input gate
- **o**: output gate
- **c**: memory cell
- **f**: forget gate



Long Short-Term Memory (LSTM)

- Input Block:

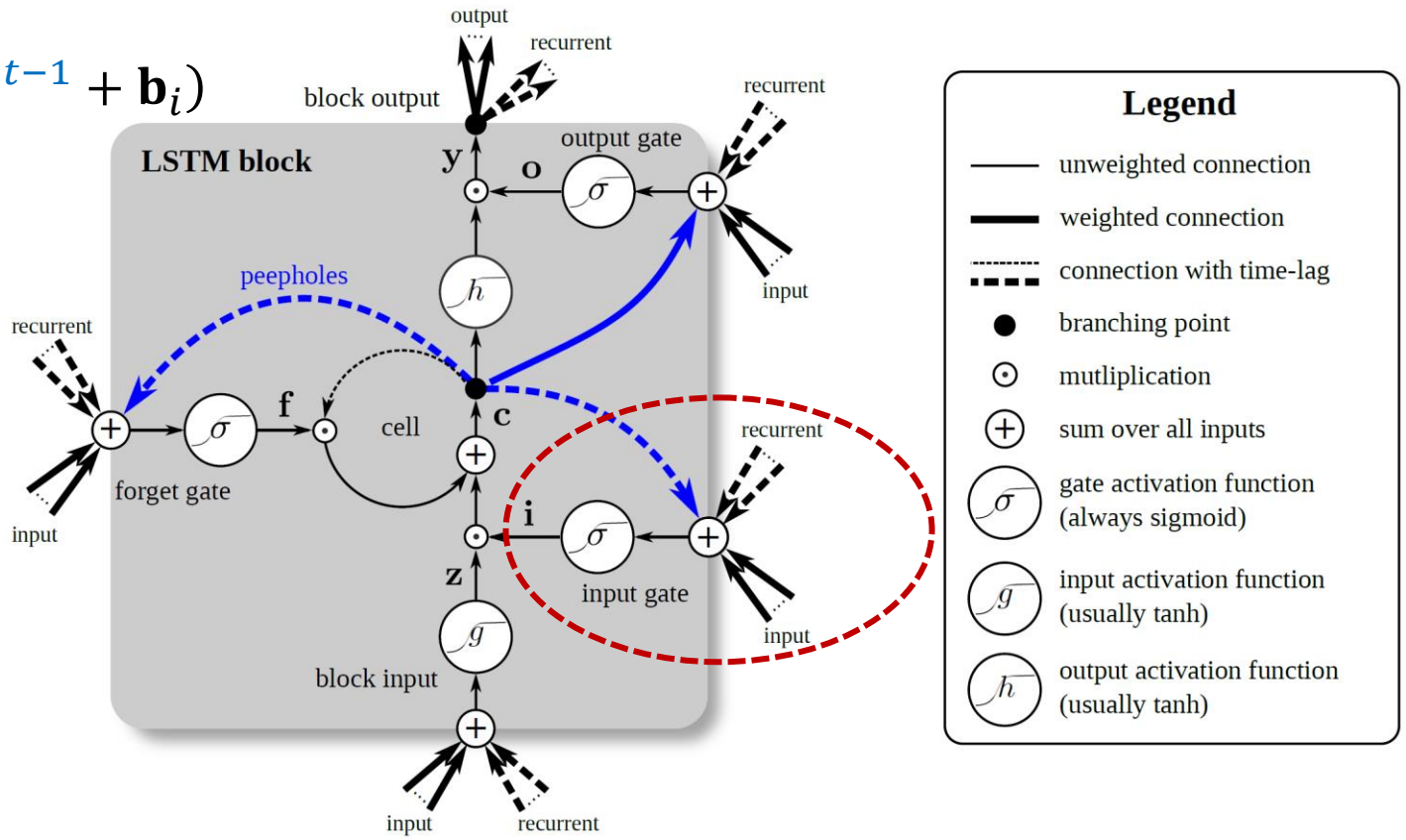
$$\mathbf{z}^t = g(\mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z)$$
$$\mathbf{z}^t = tanh(\mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z)$$



Long Short-Term Memory (LSTM)

- Input Gait:

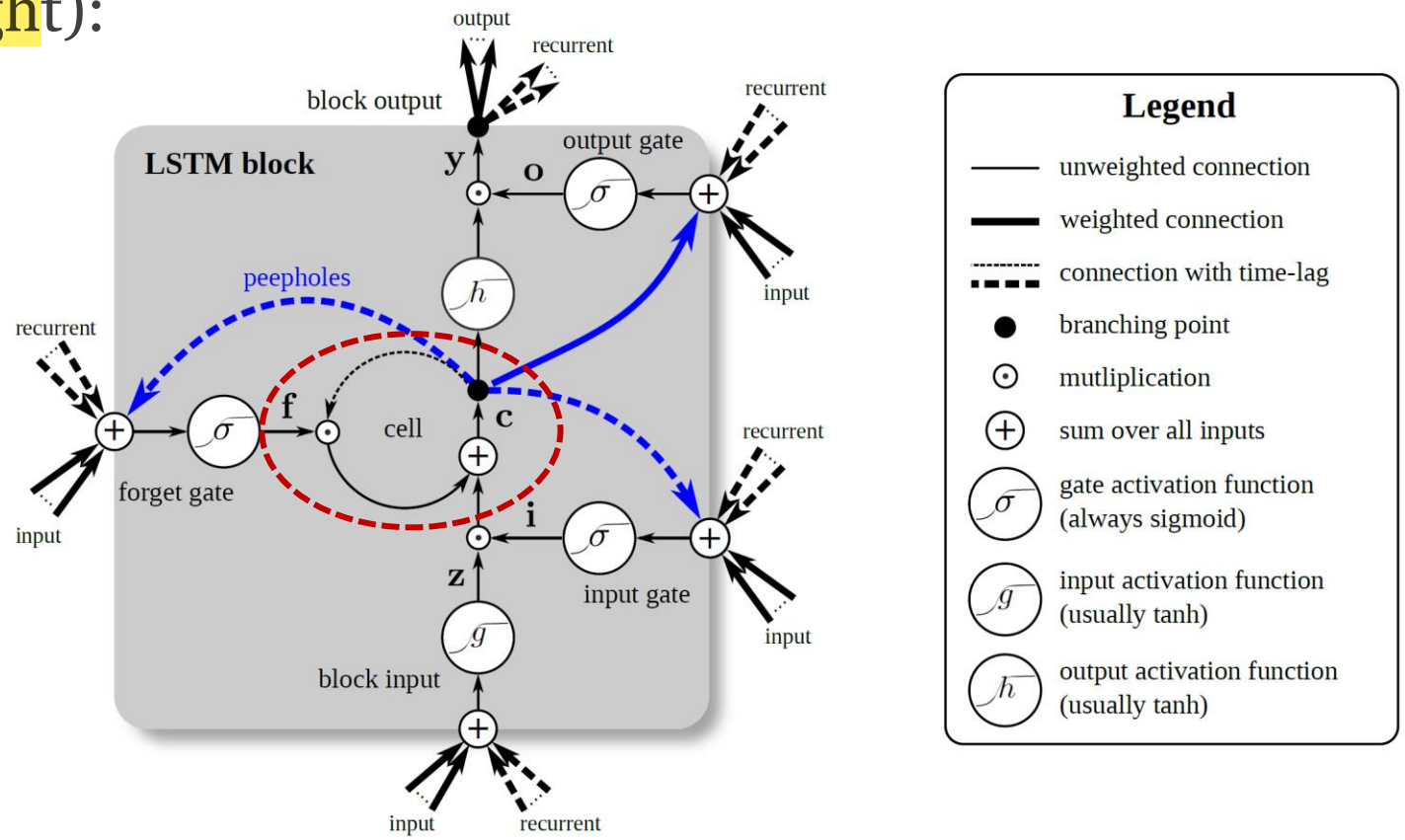
$$\mathbf{i}^t = \sigma(\mathbf{W}_i\mathbf{x}^t + \mathbf{R}_i\mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i)$$



Long Short-Term Memory (LSTM)

- Memory Cell (unity weight):

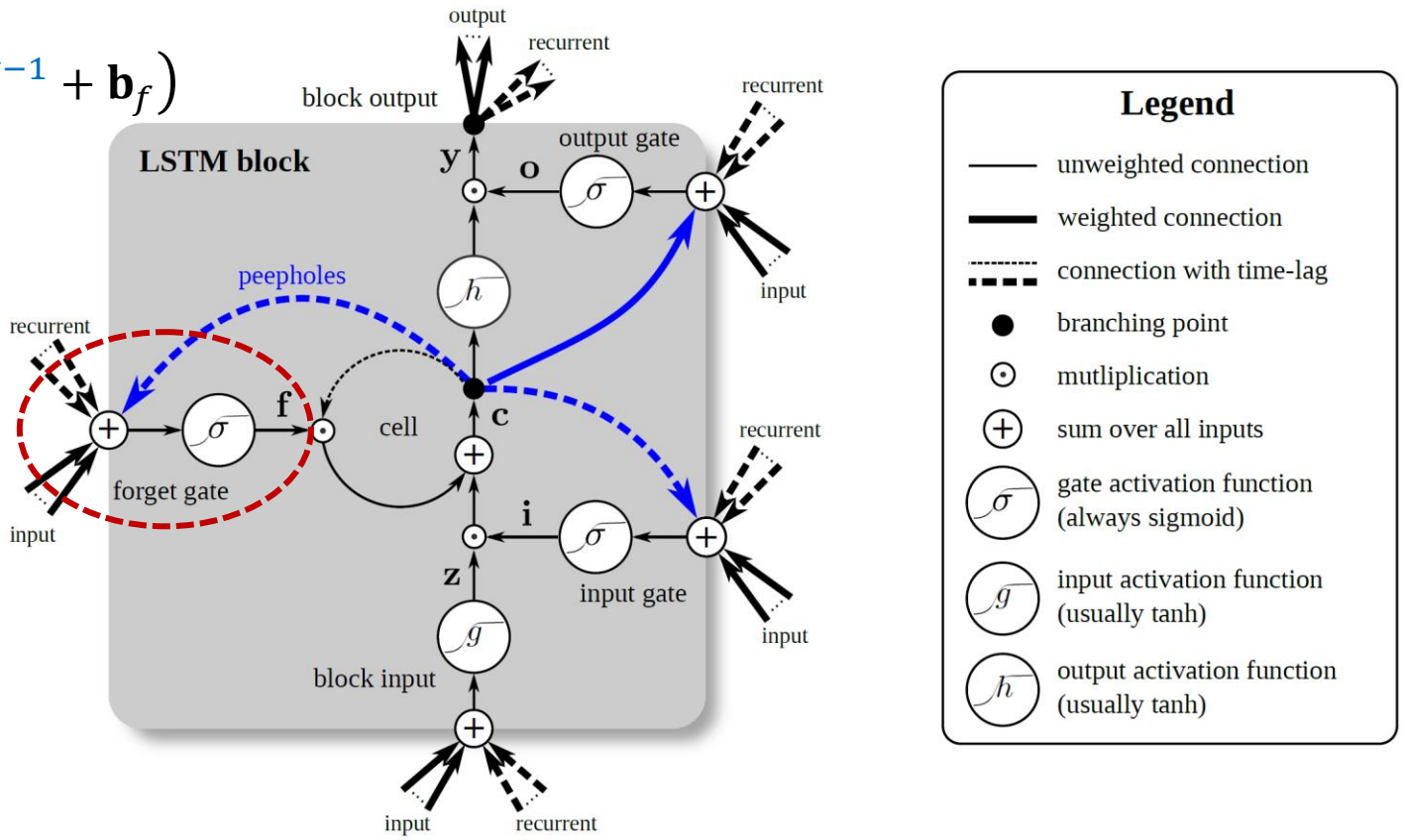
$$\mathbf{c}^t = \boldsymbol{z}^t \odot \boldsymbol{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t$$



Long Short-Term Memory (LSTM)

- Forget Gate:

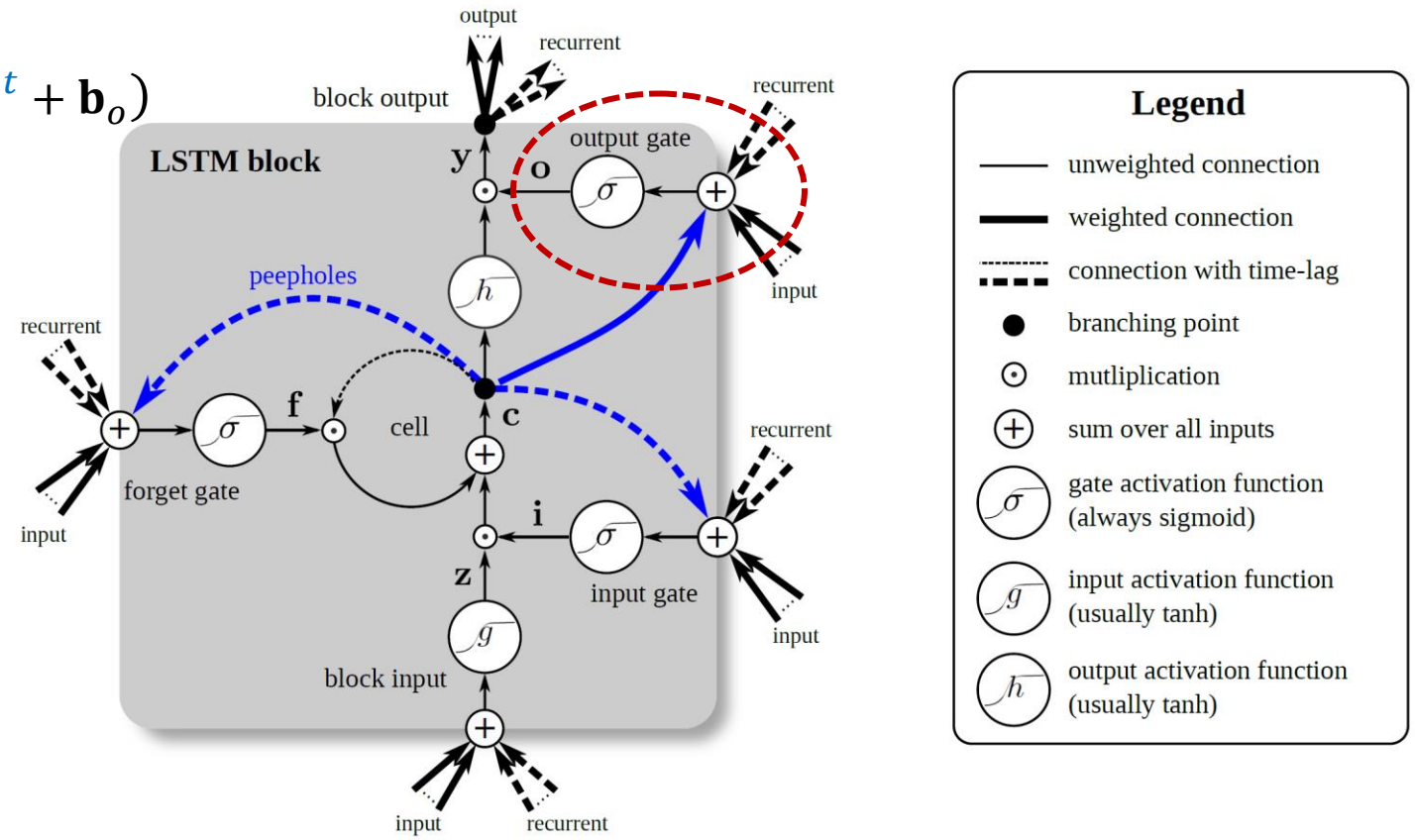
$$\mathbf{f}^t = \sigma\left(\mathbf{W}_f \mathbf{x}^t + \mathbf{R}_f \mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f\right)$$



Long Short-Term Memory (LSTM)

- Output Gait:

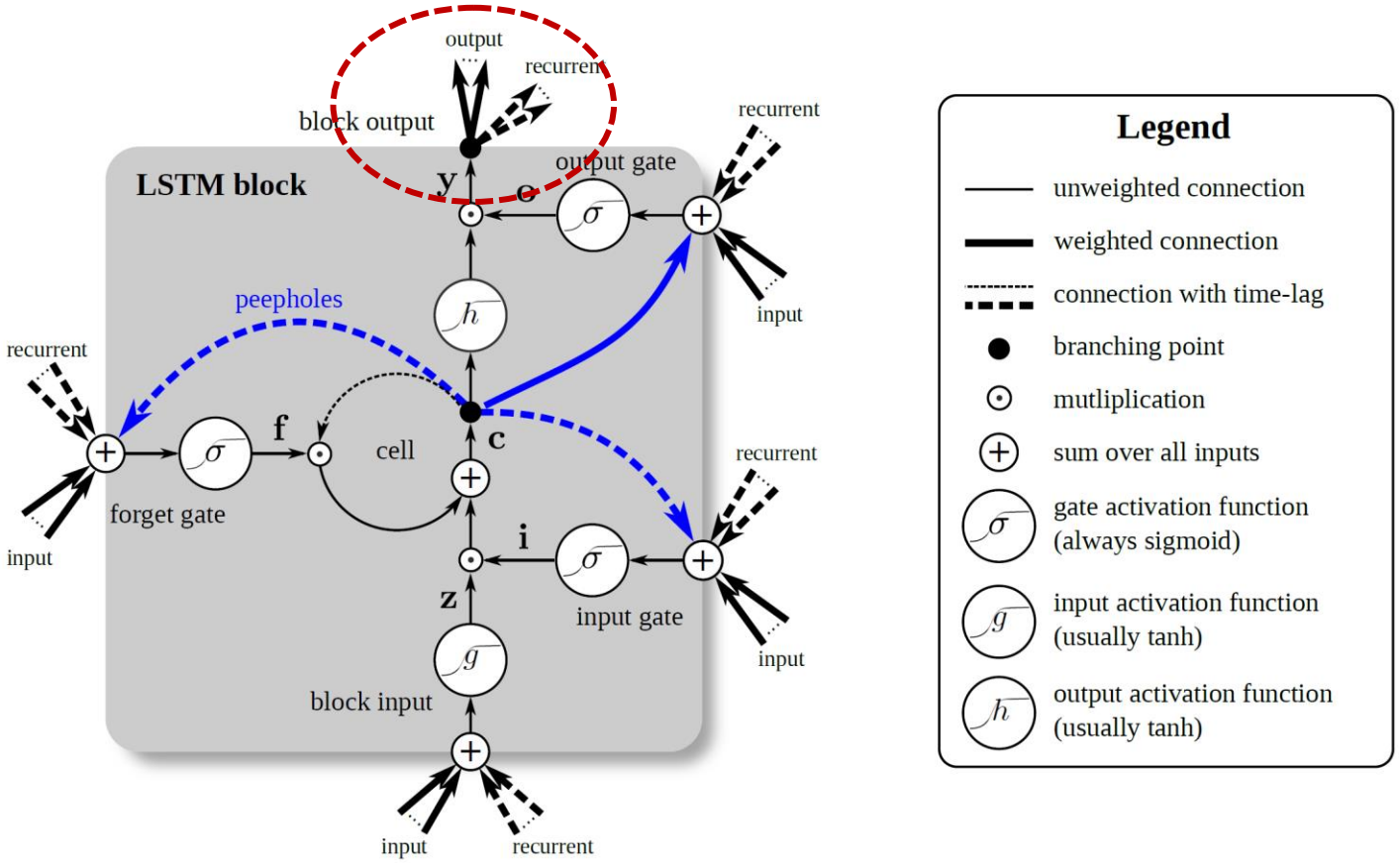
$$\mathbf{o}^t = \sigma(\mathbf{W}_o\mathbf{x}^t + \mathbf{R}_o\mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o)$$



Long Short-Term Memory (LSTM)

- Output Block:

$$\mathbf{y}^t = h(\mathbf{c}^t) \odot \mathbf{o}^t$$
$$\mathbf{y}^t = tanh(\mathbf{c}^t) \odot \mathbf{o}^t$$



Long Short-Term Memory (LSTM)

- Details:
  - $\mathbf{x}^t \in \mathbb{R}^M$: input vector at time t
  - N: Number of LSTM blocks
  - M: Number of inputs
- Learnable Parameters
  - Input weights: $\mathbf{W}_z, \mathbf{W}_i, \mathbf{W}_f, \mathbf{W}_o \in \mathbb{R}^{N \times M}$
  - Recurrent weights: $\mathbf{R}_z, \mathbf{R}_i, \mathbf{R}_f, \mathbf{R}_o \in \mathbb{R}^{N \times \mathrm{N}}$
  - Peephole weights: $\mathbf{p}_i, \mathbf{p}_f, \mathbf{p}_o \in \mathbb{R}^N$
  - Bias weights: $\mathbf{b}_z, \mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^N$

Long Short-Term Memory (LSTM)

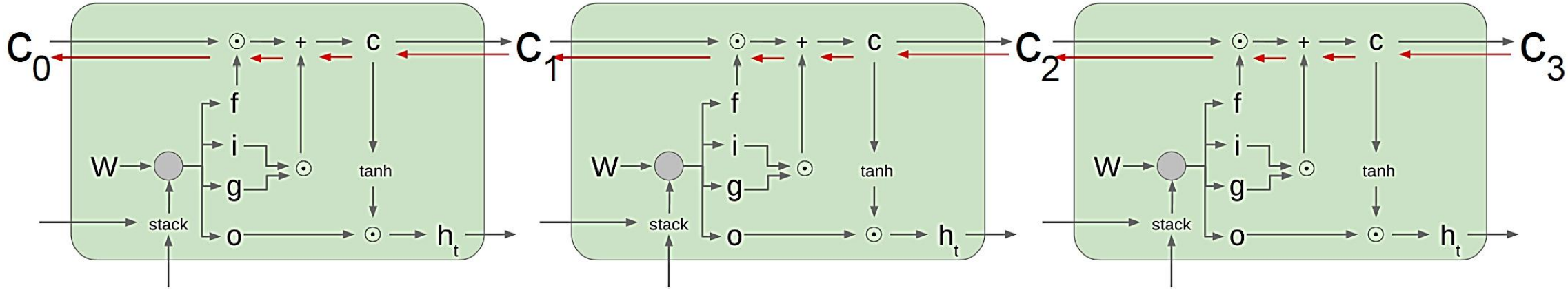
- Forward pass:
  - $\mathbf{z}^t = g(\mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z) = tanh(\mathbf{W}_z\mathbf{x}^t + \mathbf{R}_z\mathbf{y}^{t-1} + \mathbf{b}_z)$
  - $\mathbf{i}^t = \sigma(\mathbf{W}_i\mathbf{x}^t + \mathbf{R}_i\mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i)$
  - $\mathbf{f}^t = \sigma\left(\mathbf{W}_f\mathbf{x}^t + \mathbf{R}_f\mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f\right)$
  - $\mathbf{c}^t = \mathbf{z}^t \odot \mathbf{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t$
  - $\mathbf{o}^t = \sigma(\mathbf{W}_o\mathbf{x}^t + \mathbf{R}_o\mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^t + \mathbf{b}_o)$
  - $\mathbf{y}^t = h(\mathbf{c}^t) \odot \mathbf{o}^t = tanh(\mathbf{c}^t) \odot \mathbf{o}^t$

Vanilla RNN vs LSTM

- Forward pass
  - Vanilla RNN: $\mathbf{y}^t = tanh\left(\mathbf{W}\begin{bmatrix}\mathbf{y}^{t-1}\\ \mathbf{x}^t\end{bmatrix}\right)$
  - LSTM: $\begin{bmatrix}\mathbf{i}\\ \mathbf{f}\\ \mathbf{o}\\ \mathbf{z}\end{bmatrix} = \begin{bmatrix}\sigma\\ \sigma\\ \sigma\\ g\end{bmatrix}\mathbf{W}\begin{bmatrix}\mathbf{y}^{t-1}\\ \mathbf{x}^t\end{bmatrix}, \boldsymbol{c}^t = \boldsymbol{z}^t \odot \boldsymbol{i}^t + \mathbf{c}^{t-1} \odot \mathbf{f}^t, \mathbf{y}^t = h(\mathbf{c}^t) \odot \mathbf{o}^t$
  - Note $\mathbf{y}^t$ is hidden state

LSTM - Unfolding

- Uninterrupted gradient flow!

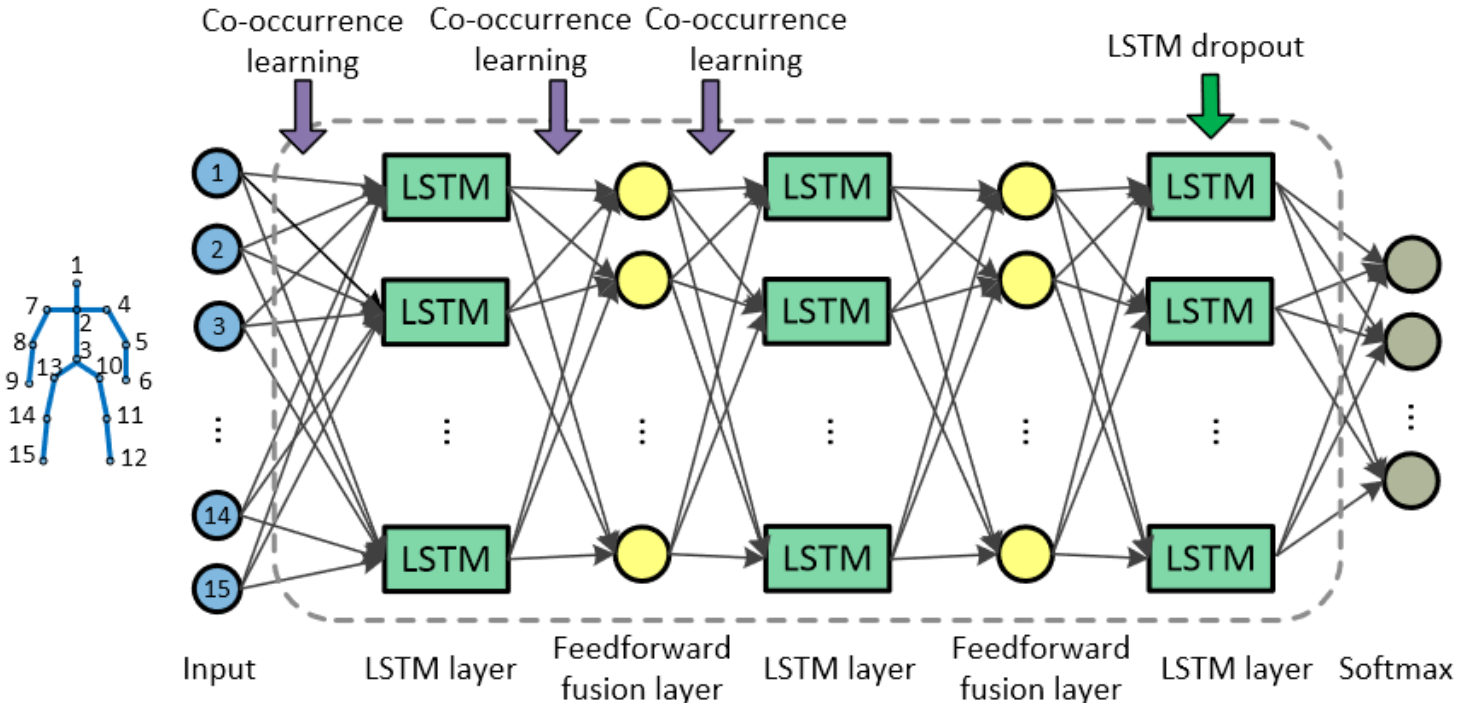


LSTM – Variants (See: LSTM: A Search Space Odyssey)

- NIG (LSTM-i): No Input Gate: $\mathbf{i}^t = 1$
- NFG (LSTM-f): No Forget Gate: $\mathbf{f}^t = 1$
- OFGB (LSTM-b): The forget gate bias is set to 1: $\mathbf{b}_f = 1$
- NOG (LSTM-o): No Output Gate: $\mathbf{o}^t = 1$
- NIAF: No Input Activation Function: $g(x) = x$
- NOAF: No Output Activation Function: $h(x) = x$
- CIFG: Coupled Input and Forget Gate: $\mathbf{f}^t = 1 - \mathbf{i}^t$
- NP: No Peepholes
- FGR (Full Gate Recurrence):
  - $\mathbf{i}^t = \sigma\left(\mathbf{W}_i\mathbf{x}^t + \mathbf{R}_i\mathbf{y}^{t-1} + \mathbf{p}_i \odot \mathbf{c}^{t-1} + \mathbf{b}_i + \mathbf{R}_{ii}\mathbf{i}^{t-1} + \mathbf{R}_{fi}\mathbf{f}^{t-1} + \mathbf{R}_{oi}\mathbf{o}^{t-1}\right)$
  - $\mathbf{f}^t = \sigma\left(\mathbf{W}_f\mathbf{x}^t + \mathbf{R}_f\mathbf{y}^{t-1} + \mathbf{p}_f \odot \mathbf{c}^{t-1} + \mathbf{b}_f + \mathbf{R}_{if}\mathbf{i}^{t-1} + \mathbf{R}_{ff}\mathbf{f}^{t-1} + \mathbf{R}_{of}\mathbf{o}^{t-1}\right)$
  - $\mathbf{o}^t = \sigma\left(\mathbf{W}_o\mathbf{x}^t + \mathbf{R}_o\mathbf{y}^{t-1} + \mathbf{p}_o \odot \mathbf{c}^{t} + \mathbf{b}_o + \mathbf{R}_{io}\mathbf{i}^{t-1} + \mathbf{R}_{fo}\mathbf{f}^{t-1} + \mathbf{R}_{oo}\mathbf{o}^{t-1}\right)$

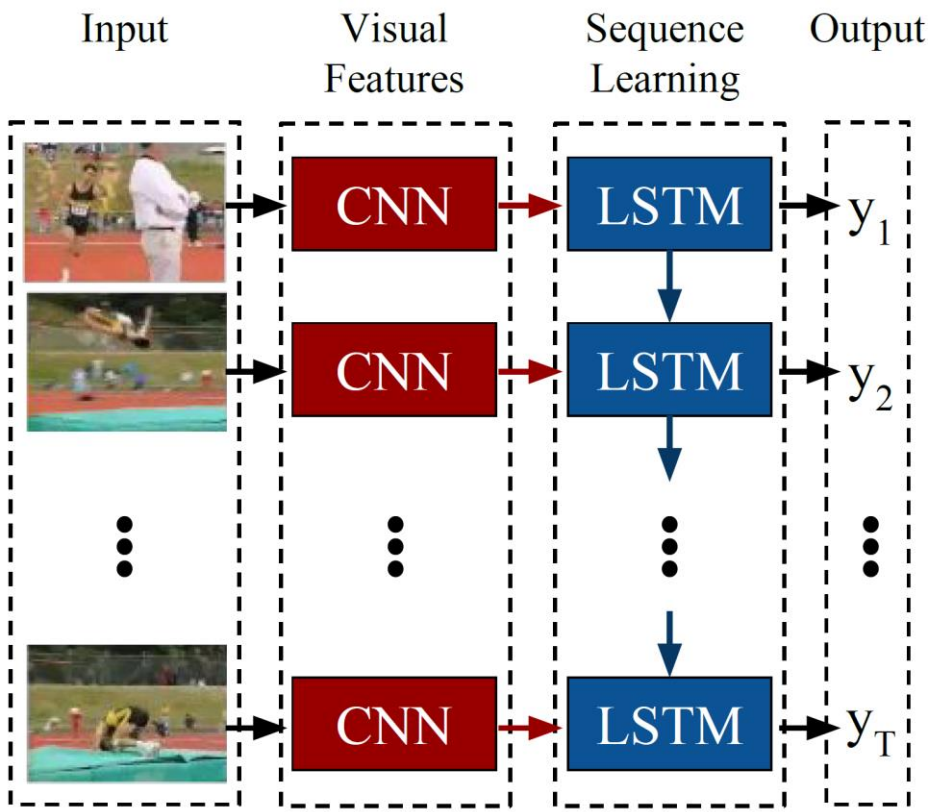
Deep LSTM

- A Complex Net



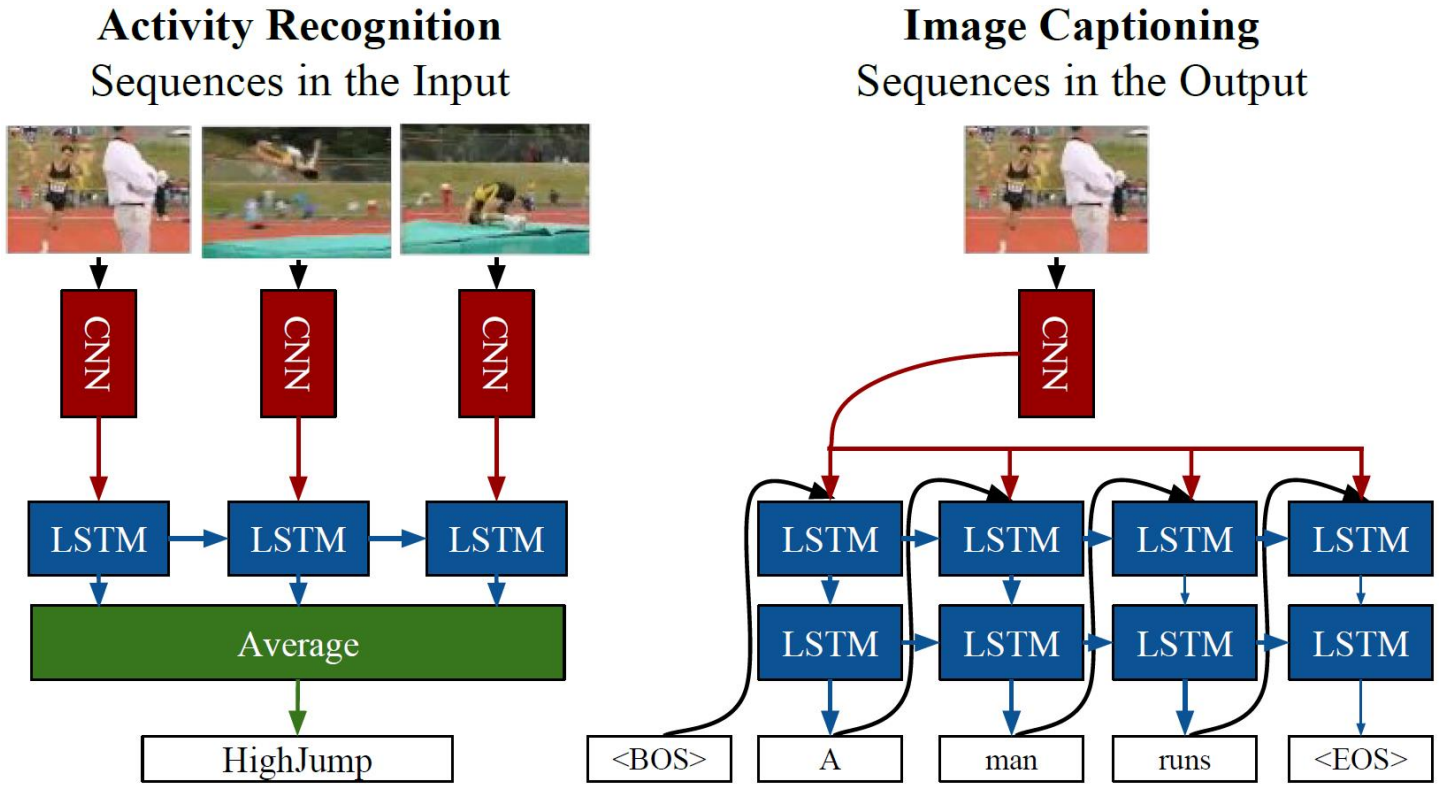
LSTM Application

- Long-term Recurrent Convolutional Networks for Visual Recognition and Description.
- Training: CNN+LSTM



LSTM Application

- Long-term Recurrent Convolutional Networks for Visual Recognition and Description.



LSTM Application

- Long-term Recurrent Convolutional Networks for Visual Recognition and Description.
- Example:

A large clock mounted to the side of a building.

A bunch of fruit that are sitting on a table.

A toothbrush holder sitting on top of a white sink.

Gate Recurrent Unit (GRU)

- The GRU is like a long short-term memory (LSTM) with a forget gate, but:
  - Has fewer parameters than LSTM, as it lacks an output gate.
  - GRU's performance on some popular tasks was found to be similar to that of LSTM.

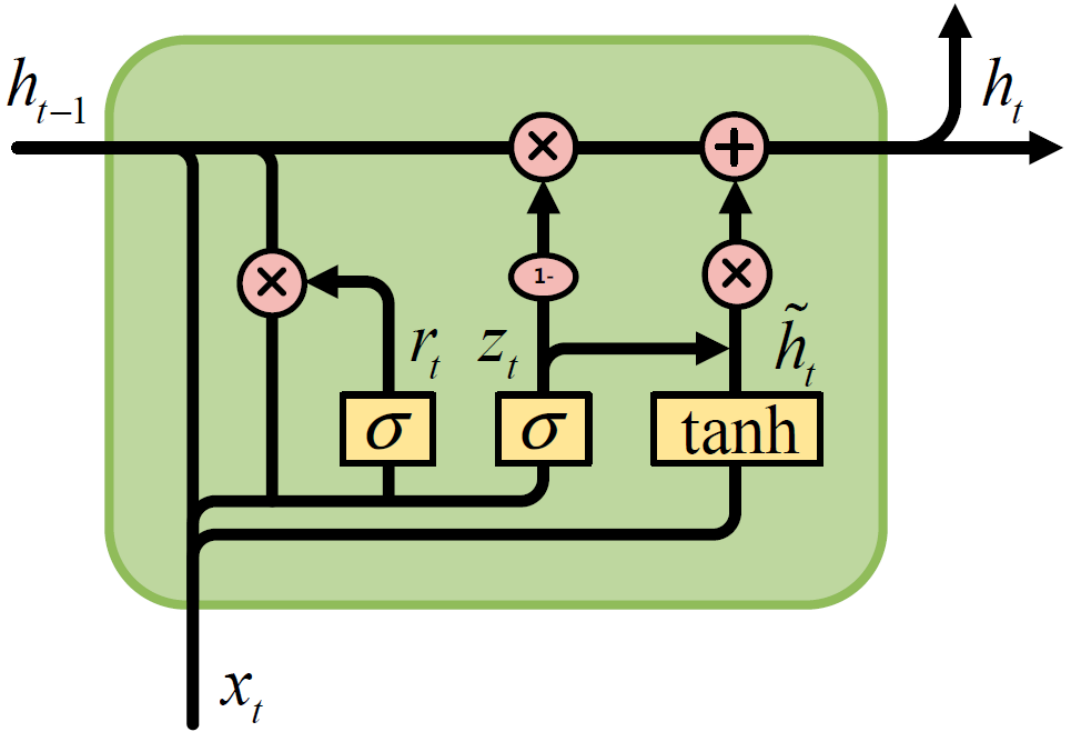
Fully Gated Unit:

- Variables:
  - $\mathbf{x}_t$: input vector
  - $\mathbf{h}_t$: output vector
  - $\hat{\mathbf{h}}_t$: candidate activation vector
  - $\boldsymbol{z}_t$: update gate vector
  - $\mathbf{r}_t$: reset gate vector
  - $\boldsymbol{W}, \mathbf{b}$: parameter matrices and vector

- Equation:
- $\mathbf{z}_t = \sigma_g(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z)$
- $\mathbf{r}_t = \sigma_g(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\boldsymbol{h}_{t-1} + \mathbf{b}_r)$
- $\hat{\mathbf{h}}_t = \phi_h(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hr}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h)$
- $\mathbf{h}_t = \boldsymbol{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \hat{\mathbf{h}}_t$
- σ_g: sigmoid function
- ϕ_h: tanh function

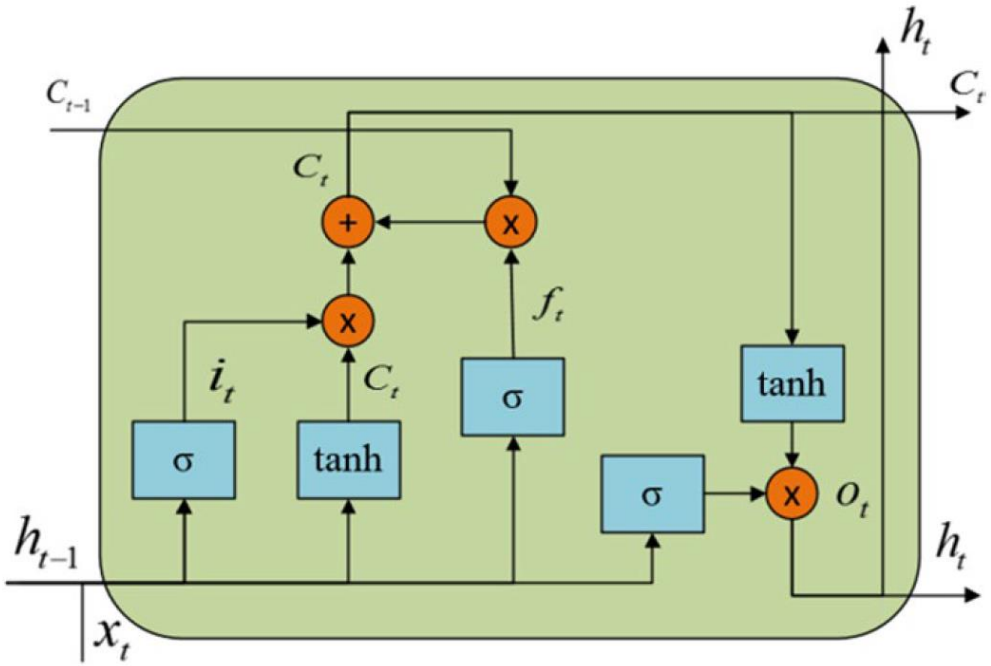
GRU - Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling

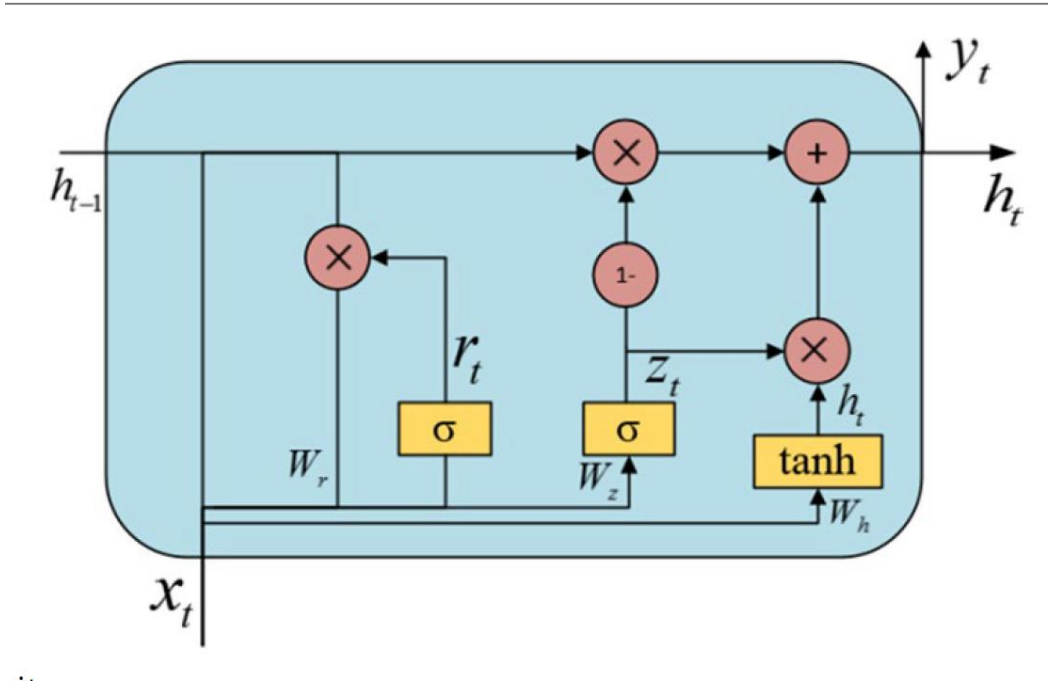
- Fully Gated Unit



- Equation:
- $\mathbf{z}_t = \sigma_g(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z)$
- $\mathbf{r}_t = \sigma_g(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\boldsymbol{h}_{t-1} + \mathbf{b}_r)$
- $\hat{\mathbf{h}}_t = \phi_h(\mathbf{W}_{xh}\mathbf{x}_t + \mathbf{W}_{hr}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h)$
- $\mathbf{h}_t = \boldsymbol{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \hat{\mathbf{h}}_t$
- σ_g: sigmoid function
- ϕ_h: tanh function

LSTM vs GRU





GRU – Type I, II, and III

- GRU Type I:
  - $\mathbf{z}_t = \sigma_g(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z) = \sigma_g(\mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z)$
  - $\mathbf{r}_t = \sigma_g(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r) = \sigma_g(\mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r)$
- GRU Type II:
  - $\mathbf{z}_t = \sigma_g(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z) = \sigma_g(\mathbf{W}_{hz}\mathbf{h}_{t-1})$
  - $\mathbf{r}_t = \sigma_g(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_{t-1} + \mathbf{b}_r) = \sigma_g(\mathbf{W}_{hr}\mathbf{h}_{t-1})$
- Gru Type III:
  - $\mathbf{z}_t = \sigma_g(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_{t-1} + \mathbf{b}_z) = \sigma_g(\mathbf{b}_z)$
  - $\mathbf{r}_t = \sigma_g(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\boldsymbol{h}_{t-1} + \mathbf{b}_r) = \sigma_g(\mathbf{b}_r)$

Minimal Gate Unit (MGU)

- Variables:
  - $\mathbf{x}_t$: input vector
  - $\mathbf{h}_t$: output vector
  - $\hat{\mathbf{h}}_t$: candidate activation vector
  - $\mathbf{f}_t$: update gate vector
  - $\boldsymbol{W}, \mathbf{b}$: parameter matrices and vector

- Equation:
- $\mathbf{f}_t = \sigma_g\left(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fh}\mathbf{h}_{t-1} + \mathbf{b}_f\right)$
- $\hat{\mathbf{h}}_t = \phi_h\left(\mathbf{W}_{hx}\mathbf{x}_t + \mathbf{W}_{hf}(\mathbf{f}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h\right)$
- $\mathbf{h}_t = \mathbf{f}_t \odot \hat{\mathbf{h}}_t + (1 - \mathbf{f}_t) \odot \mathbf{h}_{t-1}$
- σ_g: sigmoid function
- ϕ_h: tanh function

Light Gated Recurrent Unit (LiGRU)

- Variables:
  - $\mathbf{x}_t$: input vector
  - $\mathbf{h}_t$: output vector
  - $\hat{\mathbf{h}}_t$: candidate activation vector
  - $\mathbf{z}_t$: update gate vector
  - $\boldsymbol{W}, \mathbf{b}$: parameter matrices and vector

- Equation:
- $\mathbf{z}_t = \sigma_g(BN(\mathbf{W}_{xz}\mathbf{x}_t) + \mathbf{W}_{hz}\mathbf{h}_{t-1})$
- $\hat{\mathbf{h}}_t = ReLU(BN(\mathbf{W}_{xh}\mathbf{x}_t) + \mathbf{W}_{hh}\mathbf{h}_{t-1})$
- $\mathbf{h}_t = \boldsymbol{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \hat{\mathbf{h}}_t$
- σ_g: sigmoid function
- ϕ_h: tanh function

Architecture Found by Search

- MUT1:
  - $\mathbf{z}_t = \sigma\left(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{b}_z\right)$
  - $\mathbf{r}_t = \sigma\left(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_t + \mathbf{b}_r\right)$
  - $\mathbf{h}_{t+1} = \tanh(\mathbf{W}_{hh}(\mathbf{r}_t \odot \mathbf{h}_t) + \tanh(\mathbf{x}_t) + \mathbf{b}_{\mathrm{h}}) \odot \mathbf{z}_t + \mathbf{h}_t \odot (1 - \mathbf{z}_t)$
- MUT2:
  - $\mathbf{z}_t = \sigma(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}\mathbf{h}_t + \mathbf{b}_z)$
  - $\mathbf{r}_t = \sigma\left(\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_t + \mathbf{b}_r\right)$
  - $\mathbf{h}_{t+1} = \tanh(\mathbf{W}_{hh}(\mathbf{r}_t \odot \mathbf{h}_t) + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_{\mathrm{h}}) \odot \mathbf{z}_t + \mathbf{h}_t \odot (1 - \mathbf{z}_t)$
- MUT3:
  - $\mathbf{z}_t = \sigma(\mathbf{W}_{xz}\mathbf{x}_t + \mathbf{W}_{hz}tanh(\mathbf{h}_t) + \mathbf{b}_z)$
  - $\mathbf{r}_t = \sigma(\mathbf{W}_{xr}\mathbf{x}_t + \mathbf{W}_{hr}\mathbf{h}_t + \mathbf{b}_r)$
  - $\mathbf{h}_{t+1} = \tanh(\mathbf{W}_{hh}(\mathbf{r}_t \odot \mathbf{h}_t) + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_{\mathrm{h}}) \odot \mathbf{z}_t + \mathbf{h}_t \odot (1 - \mathbf{z}_t)$

GRU vs LSTM

- GRU: "update" and "reset" gates
- LSTM: "input", "output", and "forget" gates
- No internal memory state $\mathbf{c}$ in GRU
- No nonlinearity (sigmoid) before the output gate in GRU

Google Image Captioning

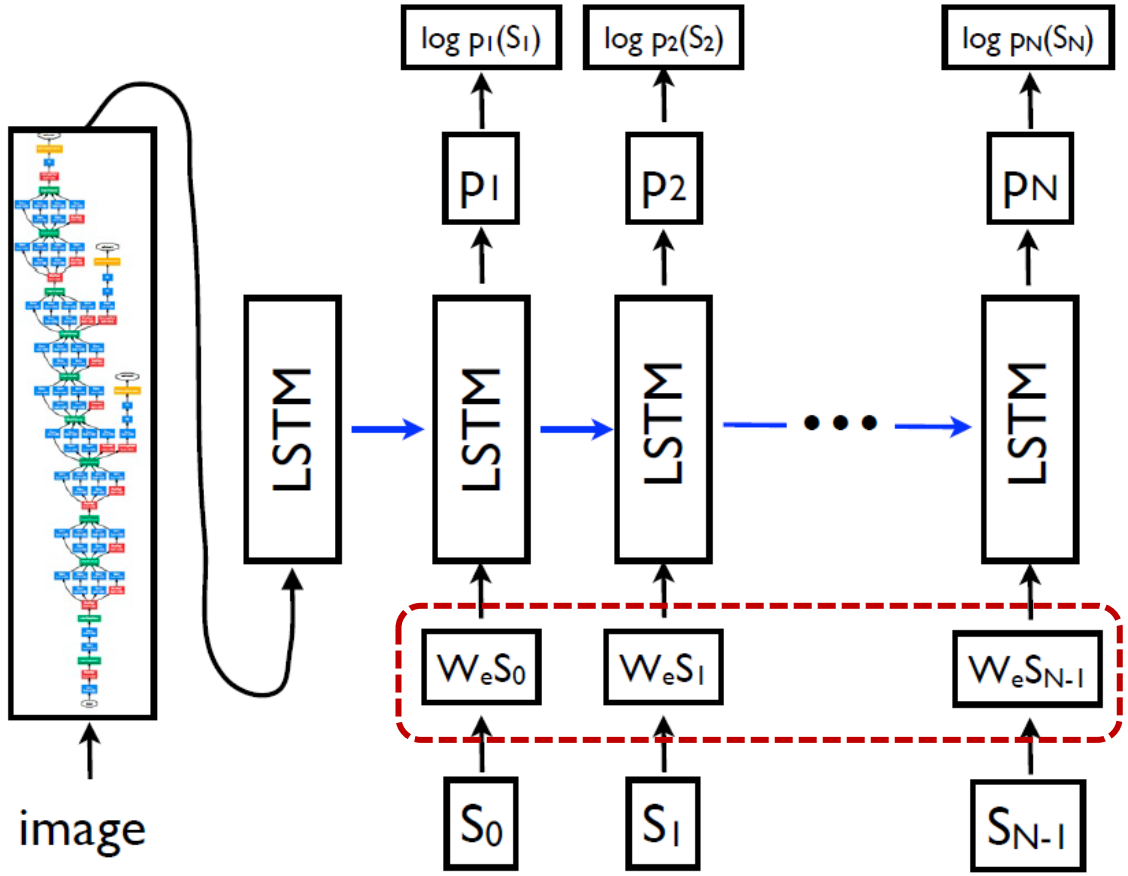
- Unfoldded Model:
- $x_{-1} = CNN(\text{Image})$

$x_t = \mathbf{W}_e S_t, \quad t$

$p_{t+1} = LSTM(x$

$S_0 =< START$

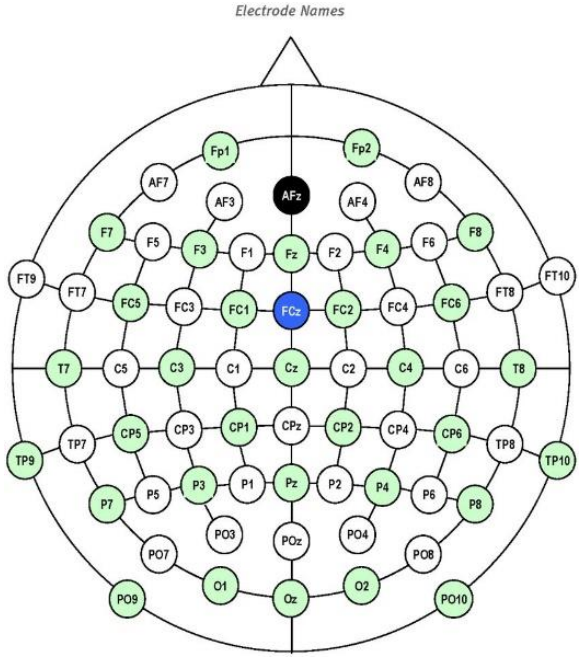
$S_N =< STOP >$



- Note: $p_{t+1} = softmax(\mathbf{y}_t)$
- $\mathbf{y}_t$: same as LSTM formulation
- $\mathbf{W}_e$: Embedding matrix (word to vector)

Medical Application – EEG Classification

- EEG – Brain Electrical Activity acquired by several sensors
  - Sensors → Signal Bank → Topographic Map



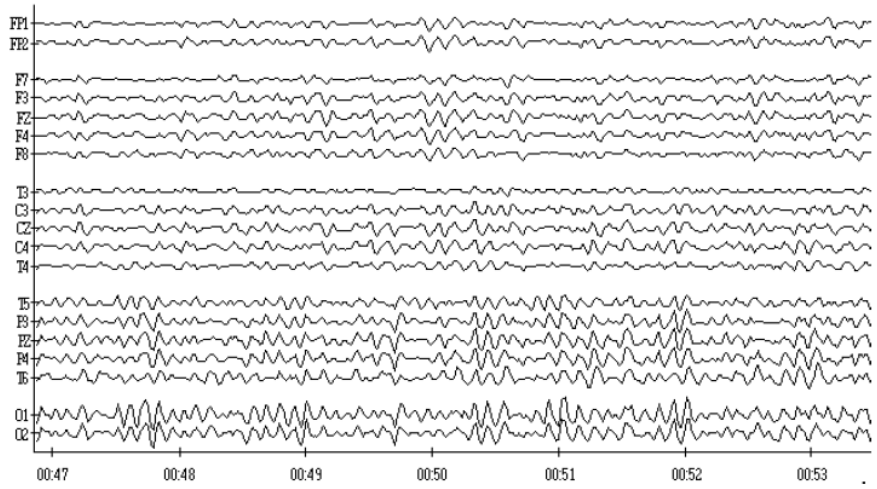
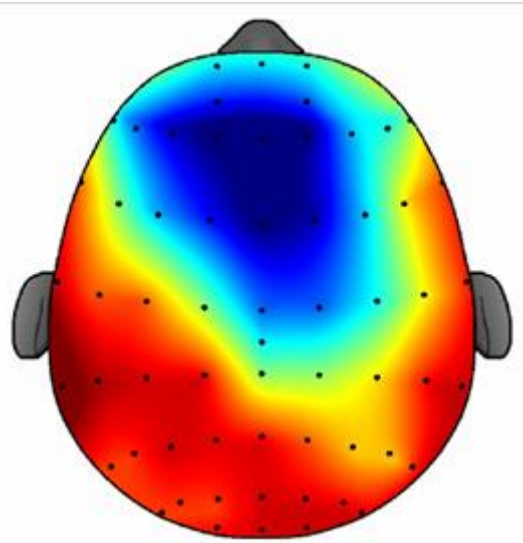


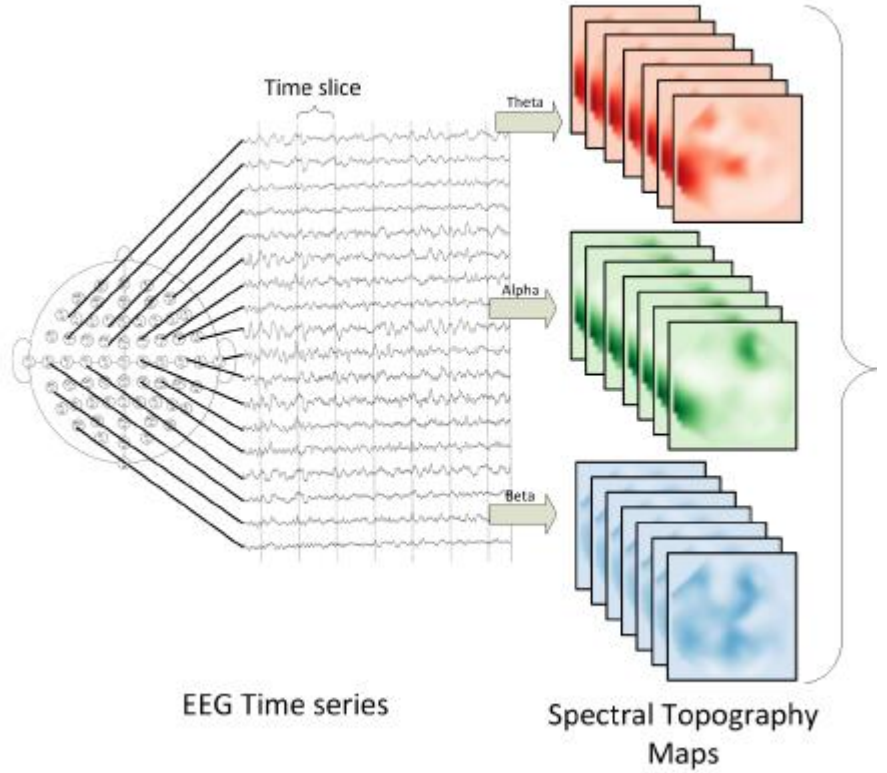
Figure 5. EEG data from 19 channels recorded from an adult.



Sharif University of Technology, School of Electrical Engineering

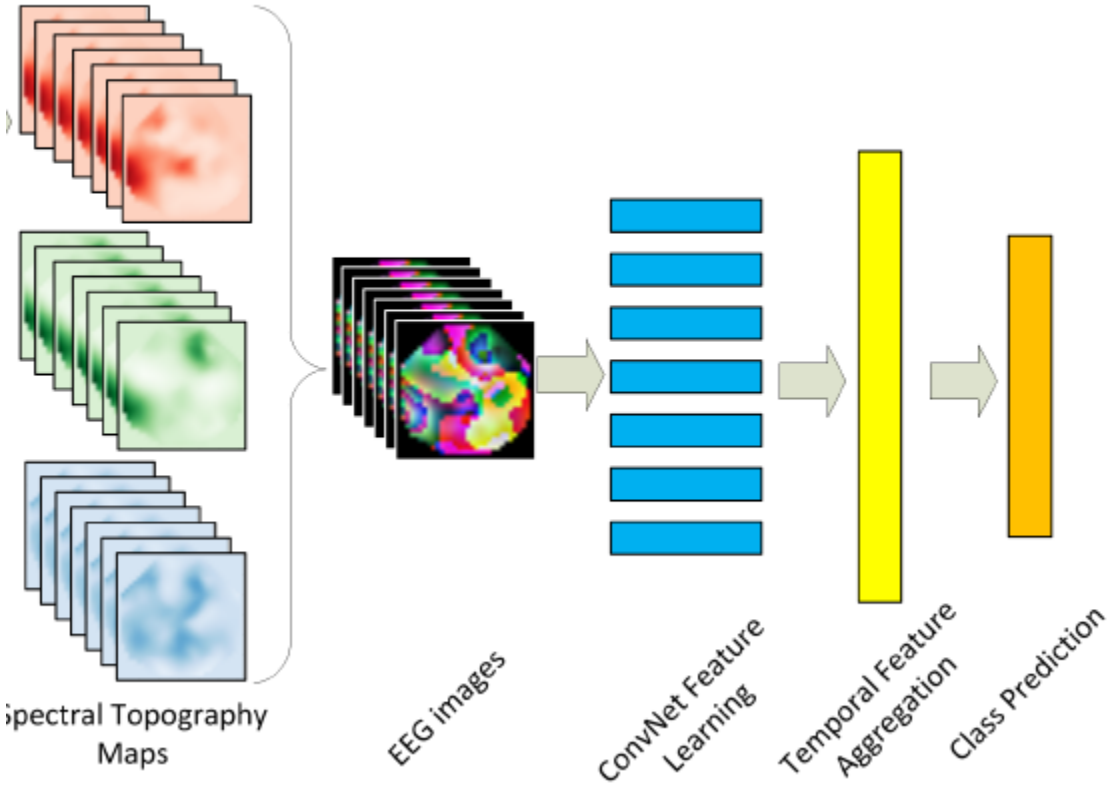
EEG Classification - Learning Representations From EEG With Deep Recurrent-Convolutional Neural Networks, ICLR2016

- Sliding Windows
- Three Frequency filtering
- Spectral power for each sensor
- Interpolate brain manifold surface to 2D images



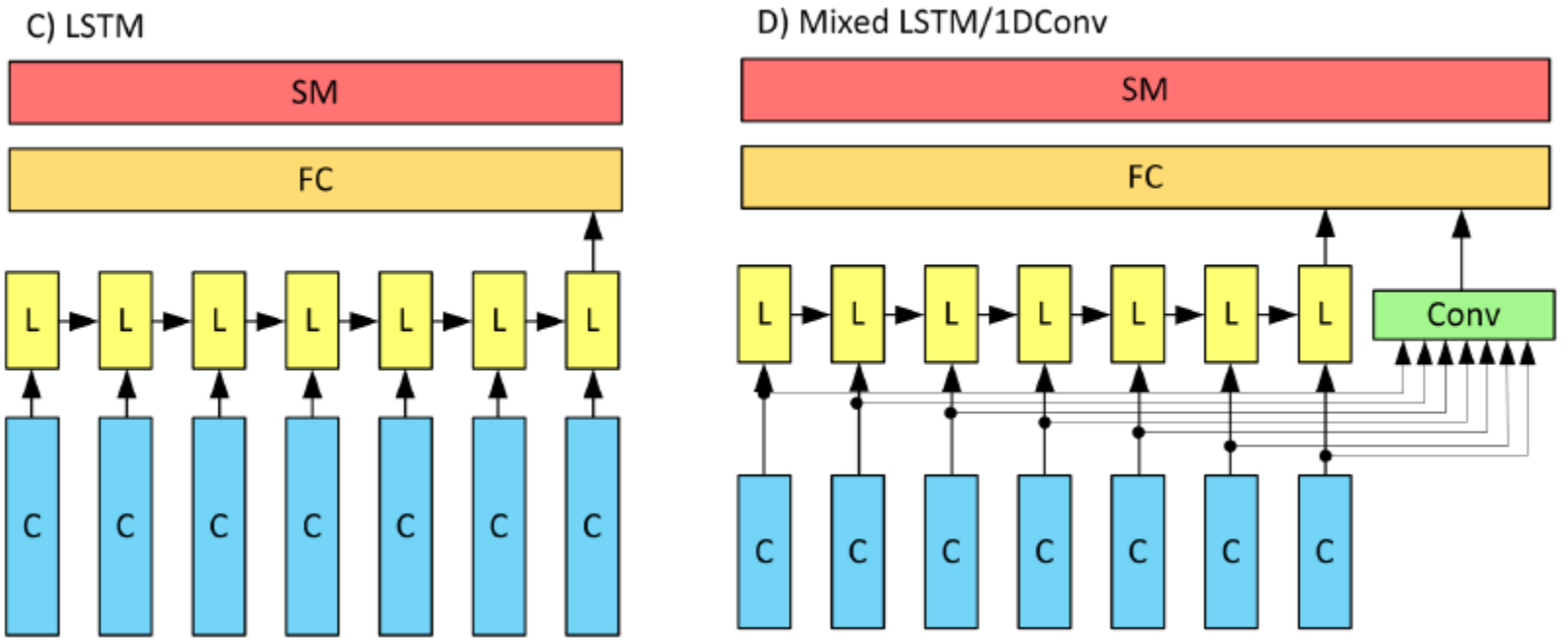
EEG Classification - Learning Representations From EEG With Deep Recurrent-Convolutional Neural Networks, ICLR2016

- Create a RGB image of brain
- VGG ConvNet
- 1D Temporal Conv. x16 and x32
- LSTM



EEG Classification - Learning Representations From EEG With Deep Recurrent-Convolutional Neural Networks, ICLR2016

- LSTM:



RNN-LSM - Advances in Optimizing Recurrent Networks, 2012

- Summary:
  - RNN family handle variable length input/output
  - (Single/Multi) (input/output) scenario are possible.
  - LSTM/GRU/… solve the curse of vanishing gradient
  - Mini-batch/Dropout/Momentum/Regularization/…
  - Gradient Clipping handle gradient exploding
  - ReLU (Sparse Gradient)

Natural Language Processing - NLP

DEEP LEARNING, FALL 2023

SHARIF UNIVERSITY OF TECHNOLOGY

NLP Applications

- Sequence Tagging
- Sentiment Classification
- Question Answering
- Machine Translation
- Text Generation (from any media)
- Predictive typing
- Speech recognition
- Handwriting recognition
- Spelling/grammar correction
- Authorship identification
- Machine translation
- Summarization
- Dialogue.
- Everything else in NLP has now been rebuilt upon Language Modeling: *GPT-n* is an LLM!

NLP Steps

- Tokenization
- Word Embedding
- Model Generation

Tokenization

- The process of converting a sequence of text into smaller parts, known as tokens. These tokens can be as small as characters or as long as words.
- Consider: *Chatbots based on deep learning are awesome, why not use?*
- Techniques:
  - Character level tokenization:

    ["C", "h", ..., "?"]
  - Word level tokenization:

    ["Chatbots", "based", ..., "awesome,", ..., "use?"]
  - Sub-word level tokenization:

    ["Chat", "bots", "based", ..., "learn", "ing", "awe ", "some", ".", ...]
  - Sentence level tokenization:

    ["Chatbots based on deep learning are awesome", "why not use?"]

Tokenization

- Tools:
  - NLTK (Natural Language Toolkit)
  - SpaCy
  - TextBlob
  - Gensim
  - BERT tokenizer
  - Byte-Pair Encoding (BPE)

Word Embedding

- Definition: A representation of a word (token).
- Typically, the representation is a real-valued vector that encodes the meaning of the word in such a way that words that are closer in the vector space are expected to be similar in meaning.
- Example:

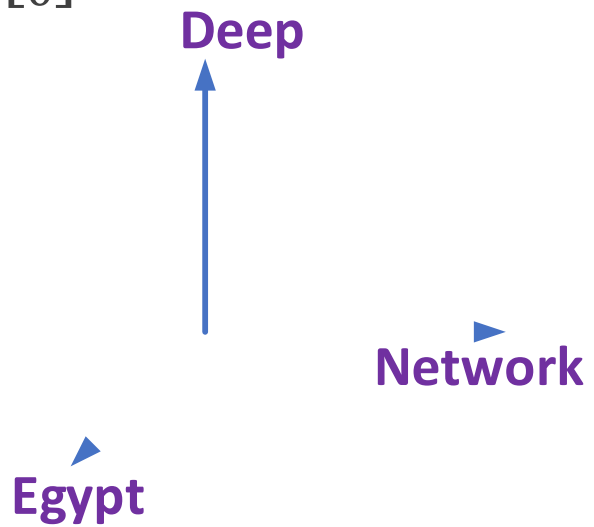
$$Deep \longrightarrow \begin{bmatrix} 1.1 \\ -2 \\ 15 \\ 9 \end{bmatrix}$$

Word Embedding

- Naïve approach: one-hot coding

$$Deep \longrightarrow \begin{bmatrix}0\\0\\1\\0\end{bmatrix}, Network \longrightarrow \begin{bmatrix}1\\0\\0\\0\end{bmatrix}, Egypt \longrightarrow \begin{bmatrix}0\\1\\0\\0\end{bmatrix}$$

- Problems:
  - Sparse and Low Capacity
  - High-dimensional
  - Hard-coded (new words new dimension)
  - No similarity information ($Deep \perp Network \perp Egypt$)

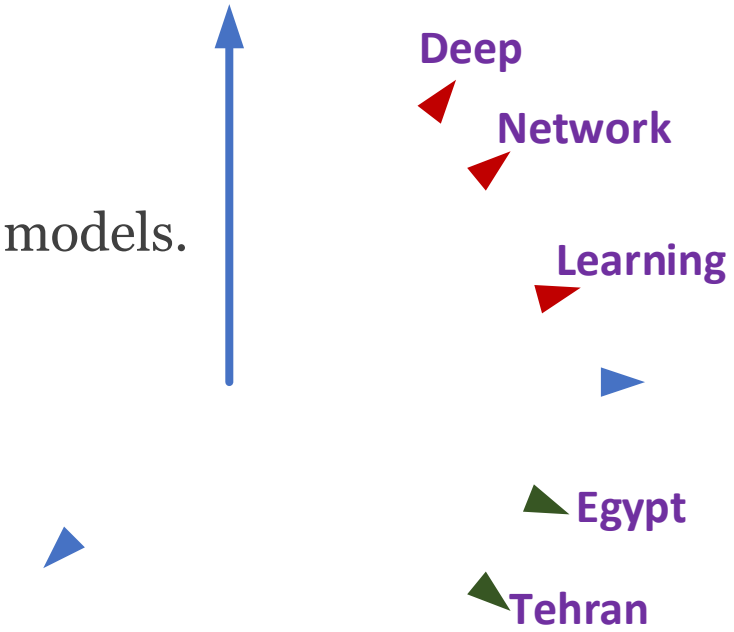


Word Embedding

- Word Embedding using embedding matrix

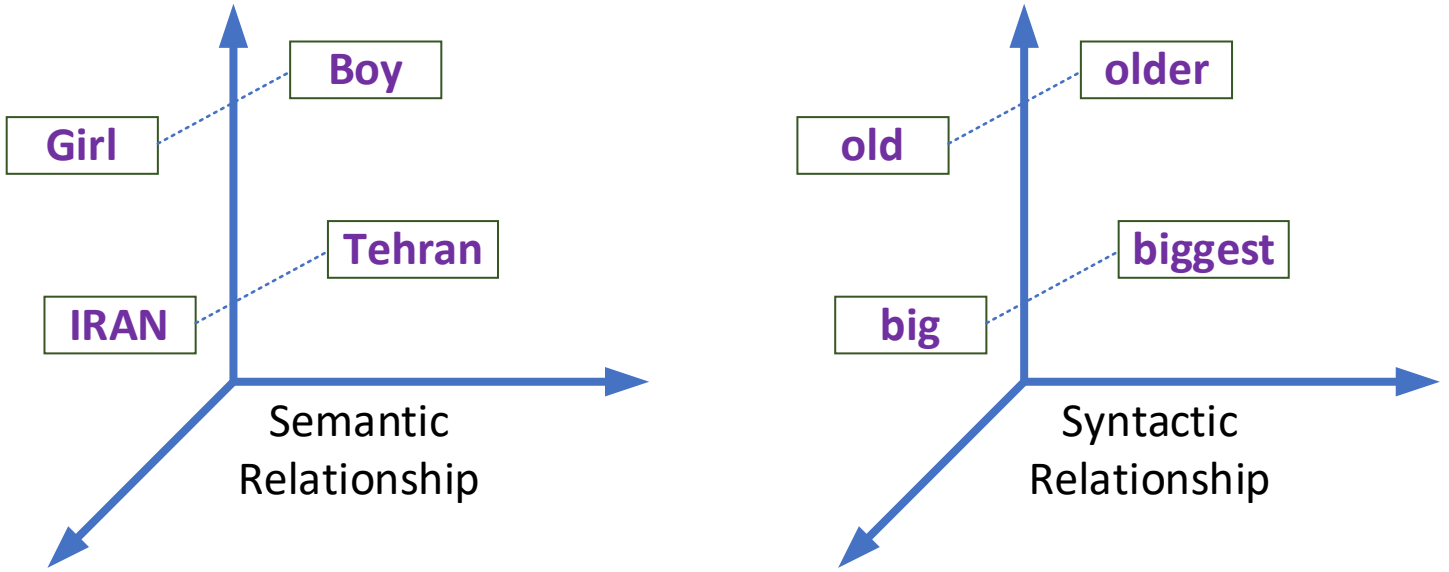
$$E_{word} = \mathbf{W} \times O_{word}$$

- $O_{word} \in \{0,1\}^V$ one-hot vectors
  - Typical value of V: 783M, 1.6B, 6B
- $E_{word} \in \mathbb{R}^N, N \ll V$
  - Typical value of E: 100, 300, 600, 1000
- $\mathbf{W} \in \mathcal{R}^{N \times V}$: can be learned using target/context likelihood models.



Word2vec - Introduction

- Word2Vec is one of the most popular technique to learn word embeddings using *shallow neural network*. It was developed by Google.
- The basic idea of Word2vec is that instead of representing words as one-hot encoding in high dimensional space, it represent words in dense low dimensional space in a way that similar words get similar word vectors, so they are mapped to nearby points.



Word2vec – How it works

- A large corpus (typically billions of words) is fed to the model.
- Corpus source: Google News, Wikipedia, Twitter (X), …
- A n-sized sliding window is used to capture the words that lie either side of each word in the corpus to determine each word's context.
- The context for each word is then used to determine the word's embedding vector.
- Example (word, context), $n = 5$:

The	quick	brown	fox	jumps	over	the	lazy	dog
The	quick	brown	fox	jumps	over	the	lazy	dog
The	quick	brown	fox	jumps	over	the	lazy	dog
The	quick	brown	fox	jumps	over	the	lazy	dog
The	quick	brown	fox	jumps	over	the	lazy	dog

Word2vec – How it works

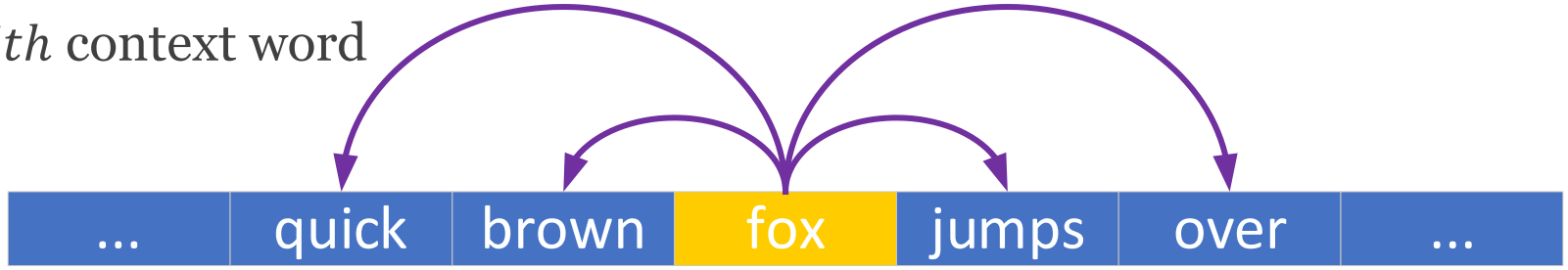
- Input⇔Output pairs

The	quick	brown	fox	jumps	over	the	lazy	dog
The	quick	brown	fox	jumps	over	the	lazy	dog
The	quick	brown	fox	jumps	over	the	lazy	dog
The	quick	brown	fox	jumps	over	the	lazy	dog
The	quick	brown	fox	jumps	over	the	lazy	dog

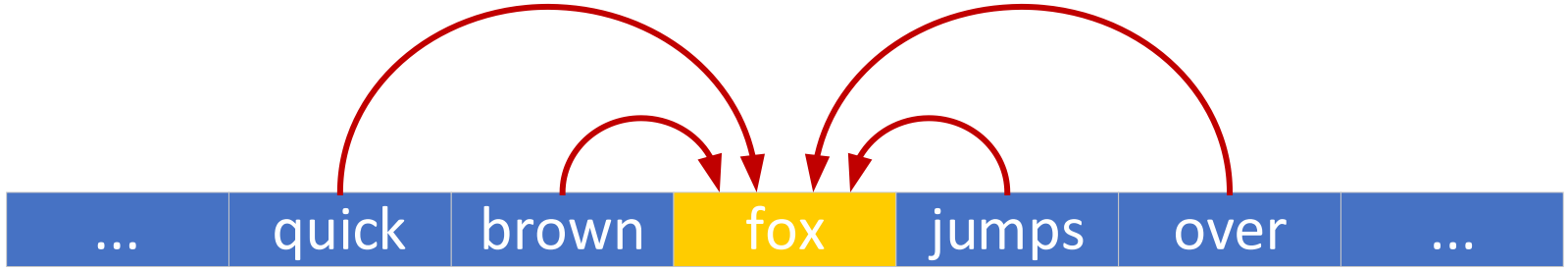
Input⇔Output pairs
{The, quick}, {The,brown}
{quick, The}, {quick, brown}, {quick, fox}
{brown, The}, {brown, quick}, {brown, fox}, {brown, jumps}
{fox, quick}, {fox, brown}, {fox, jumps}, {fox, over}

Word2vec – Two Policies

- Estimate $P\{w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m} | w_c\}$
  - w_c : center word
  - w_{c-j} : jth context word

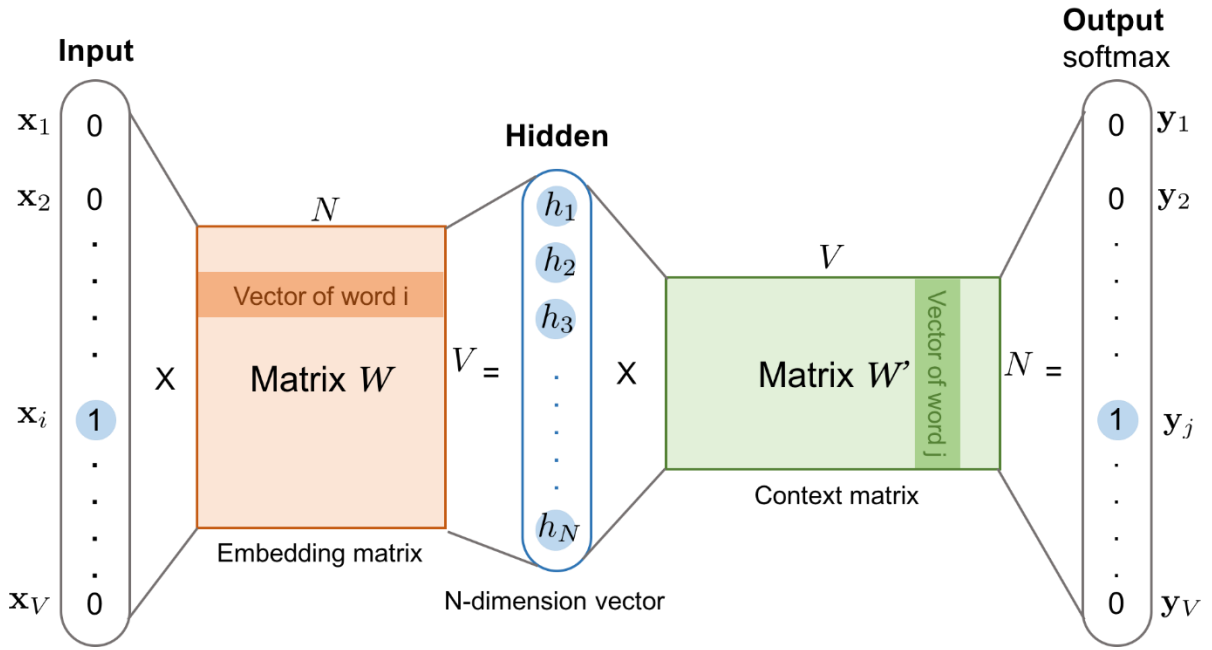


- Estimate $P\{w_c | w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m}\}$



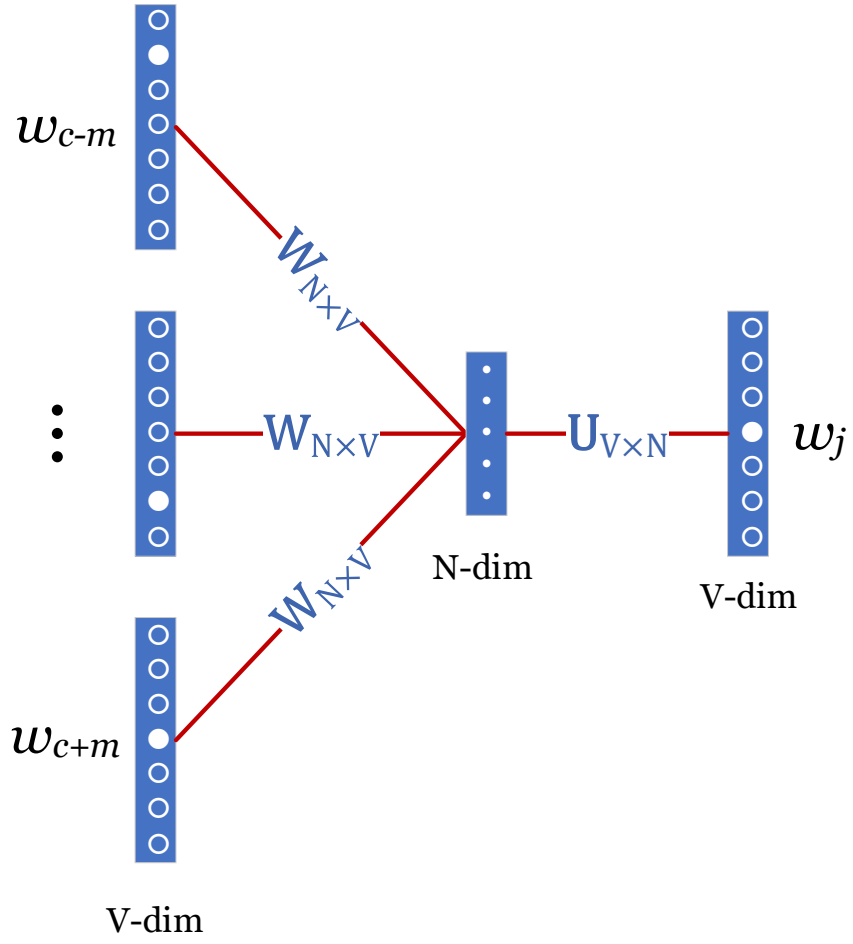
Word2vec – Network Architecture

- Architecture is too simple but HUGE:
  - Input layer (one-hot), $O_{word} \in \{0,1\}^V, V = \mathcal{O}(10^{6-8})$
  - Linear hidden layer, $E_{word} \in \mathbb{R}^N, N = \mathcal{O}(10^{2-3})$
  - Output Layer (SoftMax), $P_{word} \in [0,1]^V, V = \mathcal{O}(10^{6-8})$



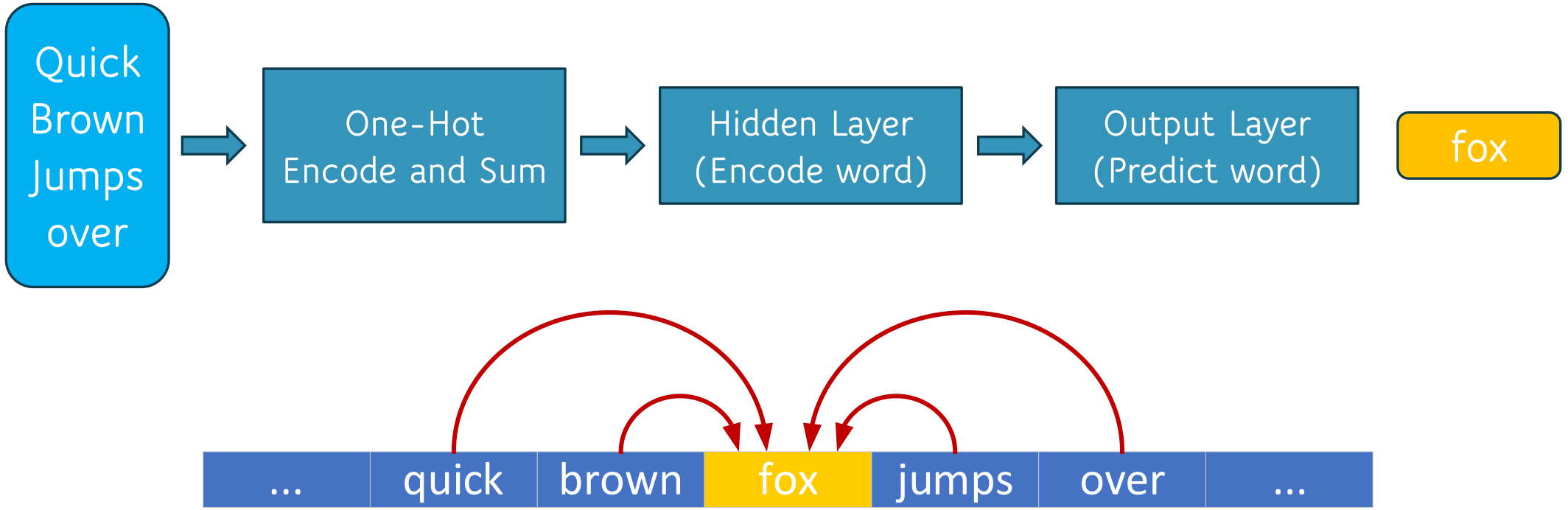
Word2vec – Learning Algorithms

- Continuous Bag-of-Words (CBoW)
- Estimate $P\{w_c|w_{c-m},\dots,w_{c-1},w_{c+1},\dots,w_{c+m}\}$
- Typical context $2m = \{4-16\}$



Word2vec – Learning Algorithms

- Continuous Bag-of-Words (CBoW)
- Estimate $P\{w_c|w_{c-m},\dots,w_{c-1},w_{c+1},\dots,w_{c+m}\}$



Continuous Bag of Words Model (CBoW)

- How to estimate $P\{w_c|w_{c-m},\dots,w_{c-1},w_{c+1},\dots,w_{c+m}\}$?
- Generate one-hot word vectors for the input context of size m

$$\left(x^{(c-m)},\dots,x^{(c-1)},x^{(c+1)},\dots,x^{(c+m)} \in \mathbb{R}^V\right)$$

- Compute embedded word vectors for the contexts (linear hidden layer):

$$\left(v^{(c-m)} = Wx^{(c-m)},\dots,v^{(c-1)} = Wx^{(c-1)},v^{(c+1)} = Wx^{(c+1)},\dots,v^{(c+m)} = Wx^{(c+m)} \in \mathbb{R}^N\right)$$

- Note: $v^j \in \mathbb{R}^N$ is j^{th} column of embedding matrix
- Average these vectors:

$$\bar{v} = \frac{1}{2m}\left(v^{(c-m)} + \cdots + v^{(c-1)} + v^{(c+1)} + \cdots + v^{(c+m)}\right) \in \mathbb{R}^N = W\bar{x}$$

- Generate a score vector $z = U\bar{v} \in \mathbb{R}^V$ (each element is dot product embedded word and rows of U)

Continuous Bag of Words Model (CBoW)

- Convert score to probability using SoftMax: $\hat{y} = softmax(z = U\bar{v}) \in \mathbb{R}^V$
- The generated probability $\hat{y} \in \mathbb{R}^V$ should match to the true probability $y \in \mathbb{R}^V$ which is one-hot coded.
- Loss function:

$$H(y, \hat{y}) = -\sum_{j=1}^{V} y_i log(\hat{y}_i) = -y_c log(\hat{y}_c) = -log(\hat{y}_c)$$

- where, c is the index where the correct word's (center word) one-hot vector is 1.

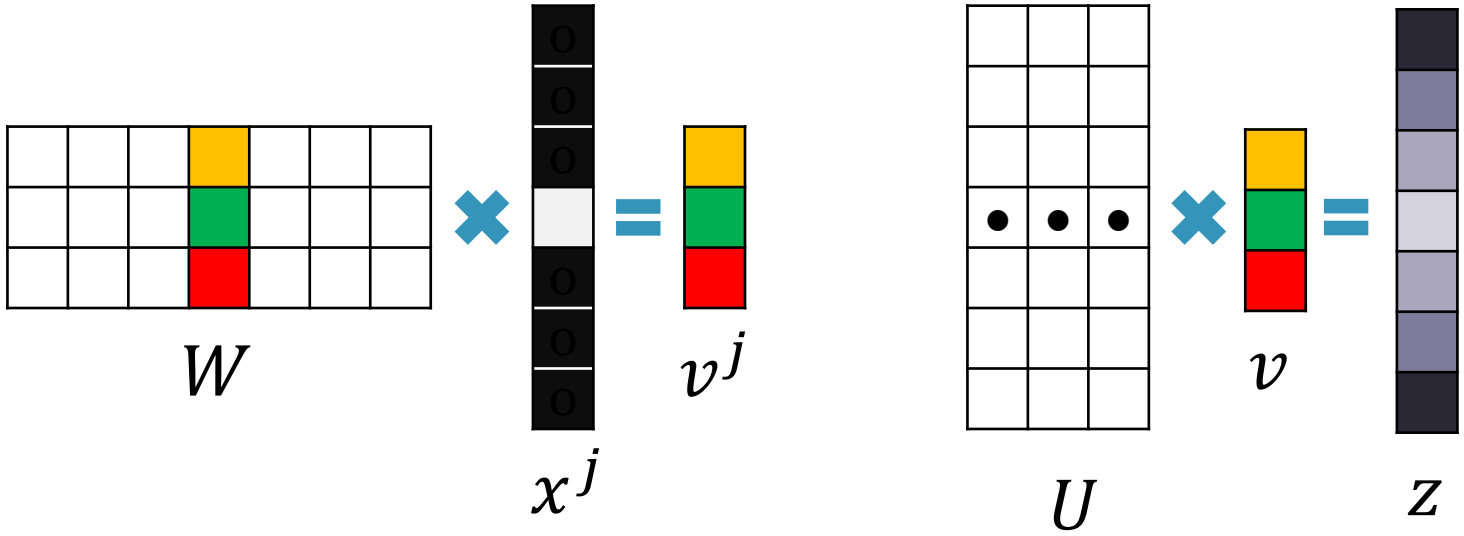
$$P\{w_c | w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m}\} = P\{u_c | \bar{v}\} = \frac{exp({u_c}^T \bar{v})}{\sum_{j=1}^{V} exp({u_j}^T \bar{v})}$$

$$Loss = -log\left(\frac{exp({u_c}^T \bar{v})}{\sum_{j=1}^{V} exp({u_j}^T \bar{v})}\right) = -{u_c}^T \bar{v} + log\left(\sum_{j=1}^{V} exp({u_j}^T \bar{v})\right)$$

- Use SGD to find word vectors u_j (rows of U) and v^j (columns of W)

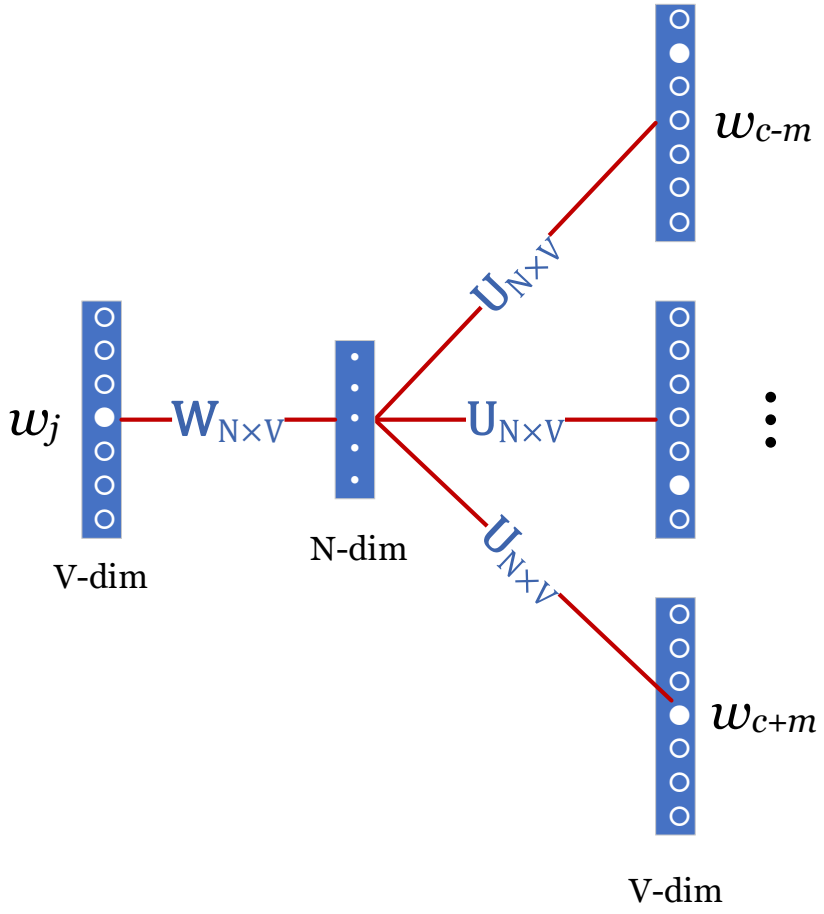
Continuous Bag of Words Model (CBoW)

- Matrix Multiplications



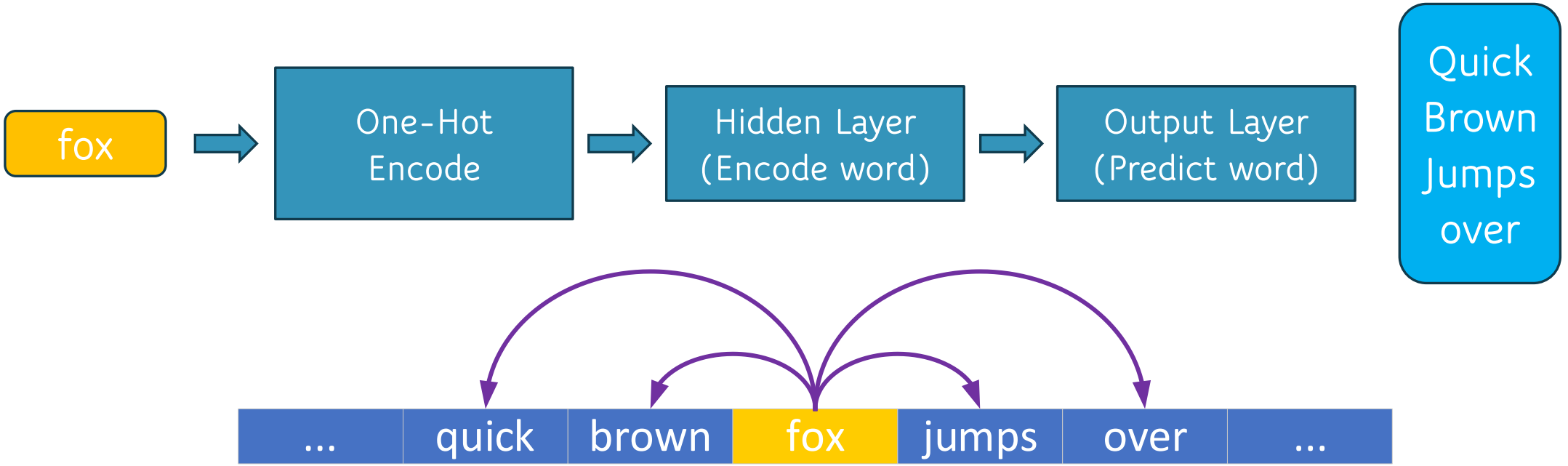
Word2vec – Learning Algorithms

- Skip-Gram (SG)
- Estimate $P\{w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m} | w_c\}$



Word2vec – Learning Algorithms

- Skip-Gram (SG)
- Estimate $P\{w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m} | w_c\}$



Skip-Gram (SG)

- How to estimate $P\{w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m} | w_c\}$?
- Generate one-hot word vectors for the input word: $x \in \mathbb{R}^V$
- Compute embedded word vectors for center word (linear hidden layer): $v_c = Wx \in \mathbb{R}^N$
- Note: v_c is c$^{\text{th}}$ column of embedding matrix
- Generate a score vector $z = Uv_c \in \mathbb{R}^V$ (each element is dot product embedded word and rows of U)
- Convert score to probability using softmax: $\hat{y} = softmax(z = Uv_c) \in \mathbb{R}^V$
- Note: $\left(\hat{y}^{(c-m)}, \dots, \hat{y}^{(c-1)}, \hat{y}^{(c+1)}, \dots, \hat{y}^{(c+m)}\right)$ are the probabilities of observing context words

Skip-Gram (SG)

- The generated probability $\left(\hat{y}^{(c-m)}, \dots, \hat{y}^{(c-1)}, \hat{y}^{(c+1)}, \dots, \hat{y}^{(c+m)}\right)$ should match to the true probability $\left(y^{(c-m)}, \dots, y^{(c-1)}, y^{(c+1)}, \dots, y^{(c+m)}\right)$ which are one-hot coded.
- Loss function (assume output words are independent):

$$Loss = -logP\{w_{c-m}, \dots, w_{c-1}, w_{c+1}, \dots, w_{c+m}|w_c\} = -log\left(\prod_{j=0, j\neq m}^{2m} P\left(w_{c-m+j}|w_c\right)\right)$$

$$Loss = -log\left(\prod_{j=0, j\neq m}^{2m} P\left(w_{c-m+j}|w_c\right)\right) = -log\left(\prod_{j=0, j\neq m}^{2m} P\left(u_{c-m+j}|v_c\right)\right)$$

- Remember that: vectors u_j (rows of U) and v^j (columns of W)

Skip-Gram (SG)

- The generated probability $\left(\hat{y}^{(c-m)}, \dots, \hat{y}^{(c-1)}, \hat{y}^{(c+1)}, \dots, \hat{y}^{(c+m)}\right)$ should match to the true probability $\left(y^{(c-m)}, \dots, y^{(c-1)}, y^{(c+1)}, \dots, y^{(c+m)}\right)$ which are one-hot coded.

$$Loss = -log\left(\prod_{j=0, j\neq m}^{2m} P\left(u_{c-m+j} \middle| v_c\right)\right) = -log\left(\prod_{j=0, j\neq m}^{2m} \frac{exp\left(u_{c-m+j}^T v_c\right)}{\sum_{k=1}^{V} exp(u_k{}^T v_c)}\right)$$

$$Loss = -\sum_{j=0, j\neq m}^{2m} u_{c-m+j}^T v_c + 2m log\left(\sum_{k=1}^{V} exp\left(u_k^T v_c\right)\right)$$

- Use SGD to find word vectors u_j (rows of U) and v^j (columns of W)
- This is a like as a minibatch learning consist of center word and all contexts

Word2vec last words

- In softmax computation, denominator has to be computed over the entire vocabulary!

$$\frac{exp(u_j^T v_c)}{\sum_{k=1}^{V} exp({u_k}^T v_c)}, V = \mathcal{O}(10^{6-8})$$

- In *word2vec* trick of negative sampling has been used (only a subset of randomly chosen negative samples has been used).
- Alternative methods for word embedding:
  - Stanford's GloVe (2014)
  - Facebook's FastText (2016)
- An interesting arithmetic properties:
  - $vector('Rome') \approx vector('Paris') - vector('France') + vector('Italy')$
  - $vector('Queen') \approx vector('King') - vector('Man') + vector('Woman')$

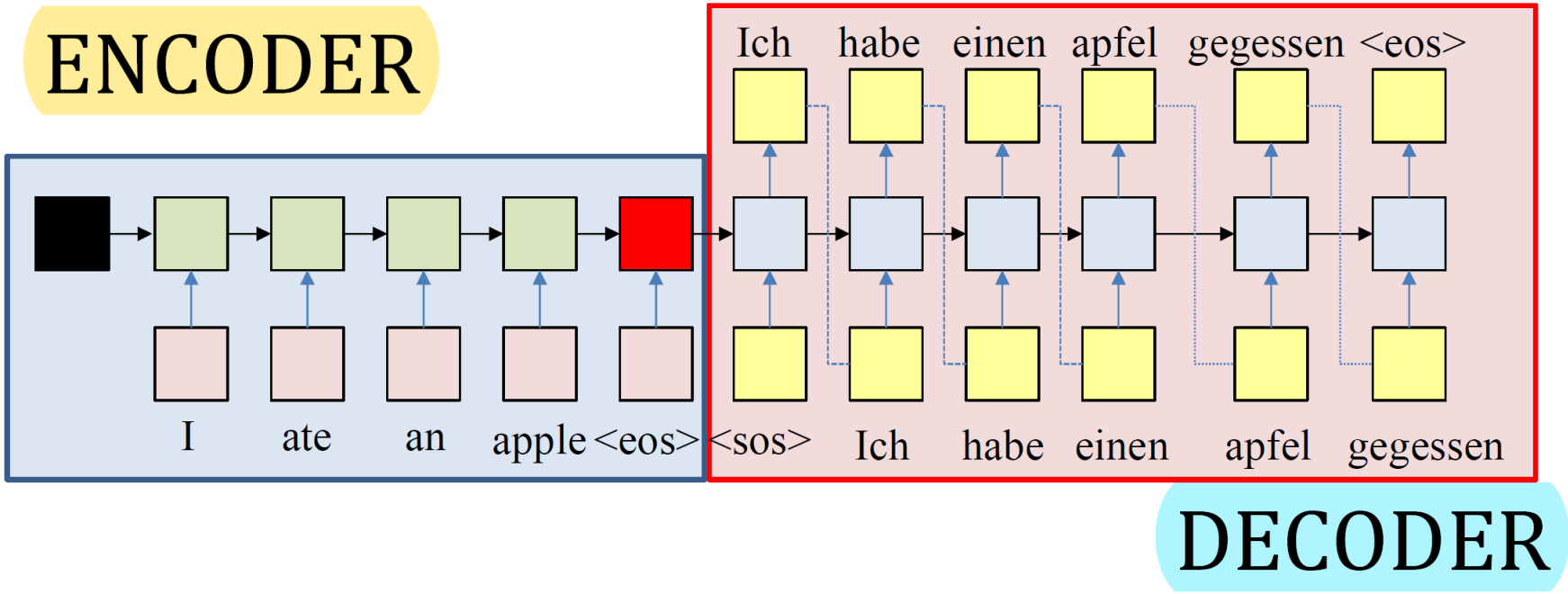
Attention and Self Attention

DEEP LEARNING, FALL 2023

SHARIF UNIVERSITY OF TECHNOLOGY

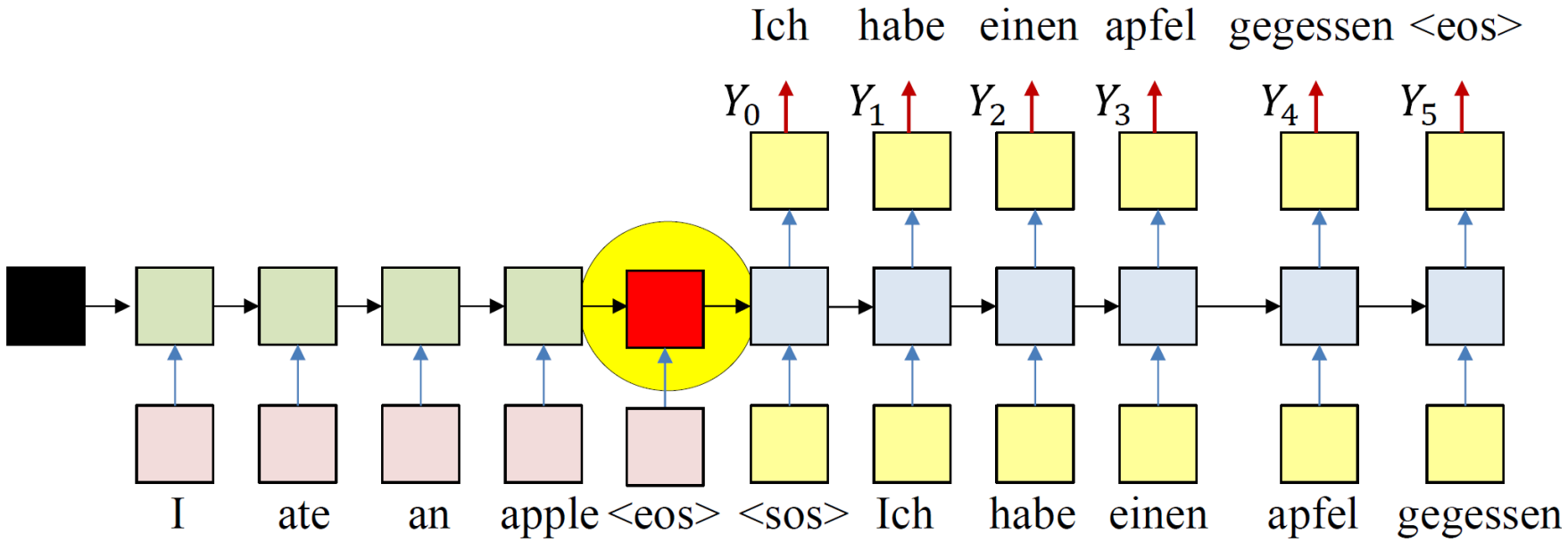
Seq2Seq Architecture

- Simple Seq2Seq (Translation) Model:



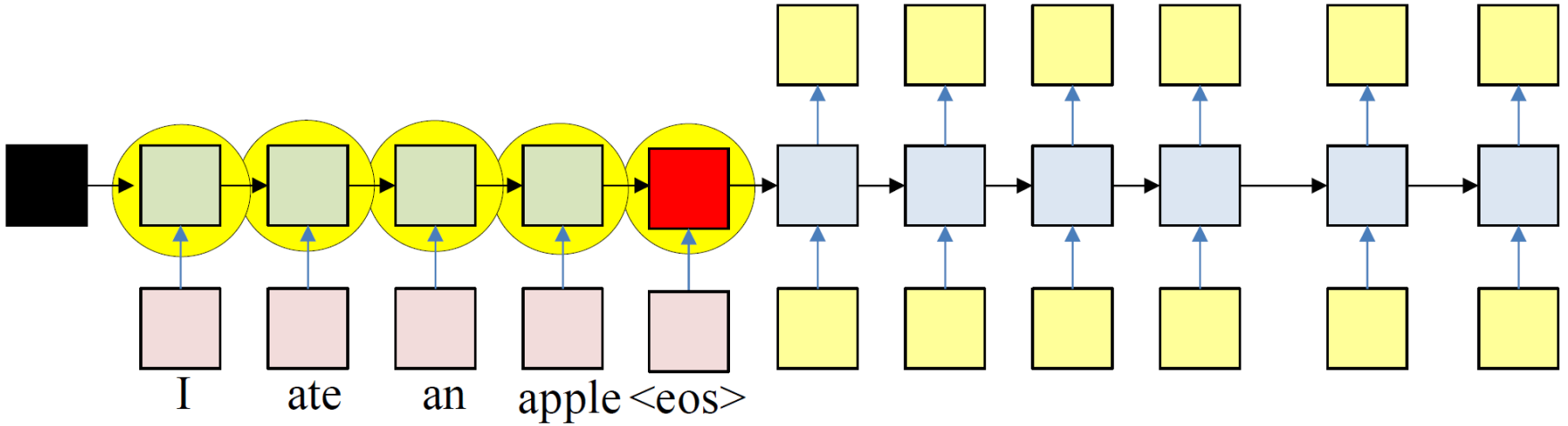
Simple Seq2Seq Architecture

- All the information about the input sequence is embedded into a single vector, the last hidden neuron "overloaded" with information (Particularly for the long sequences)



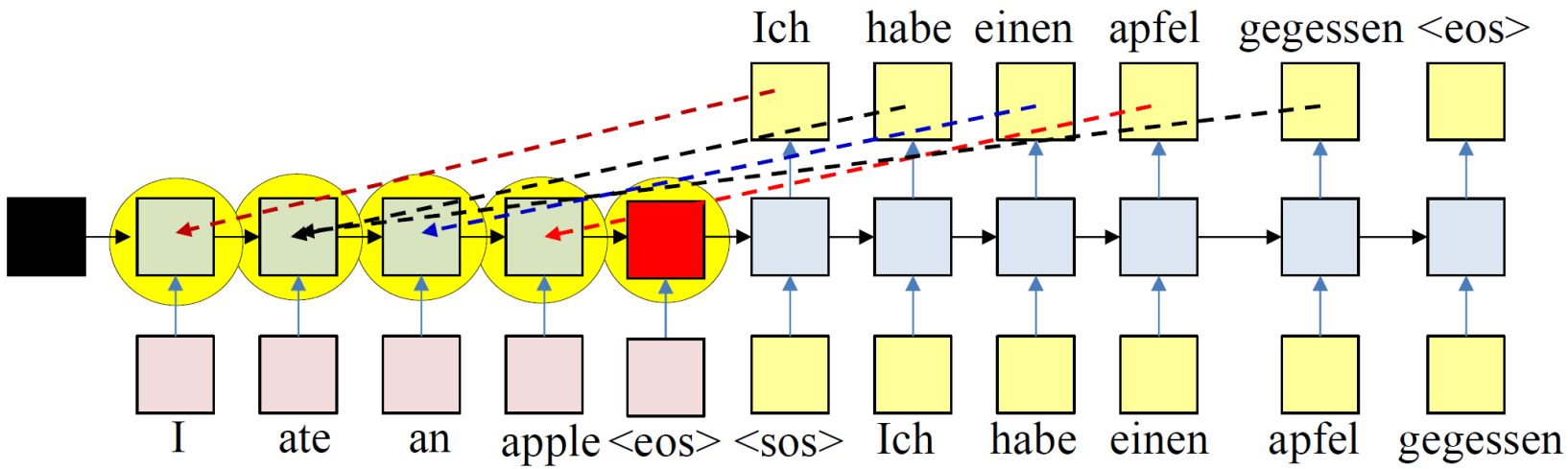
Simple Seq2Seq Architecture

- Note that ALL hidden values carry information, but some of which may be diluted by the time we get to the final state of the encoder.



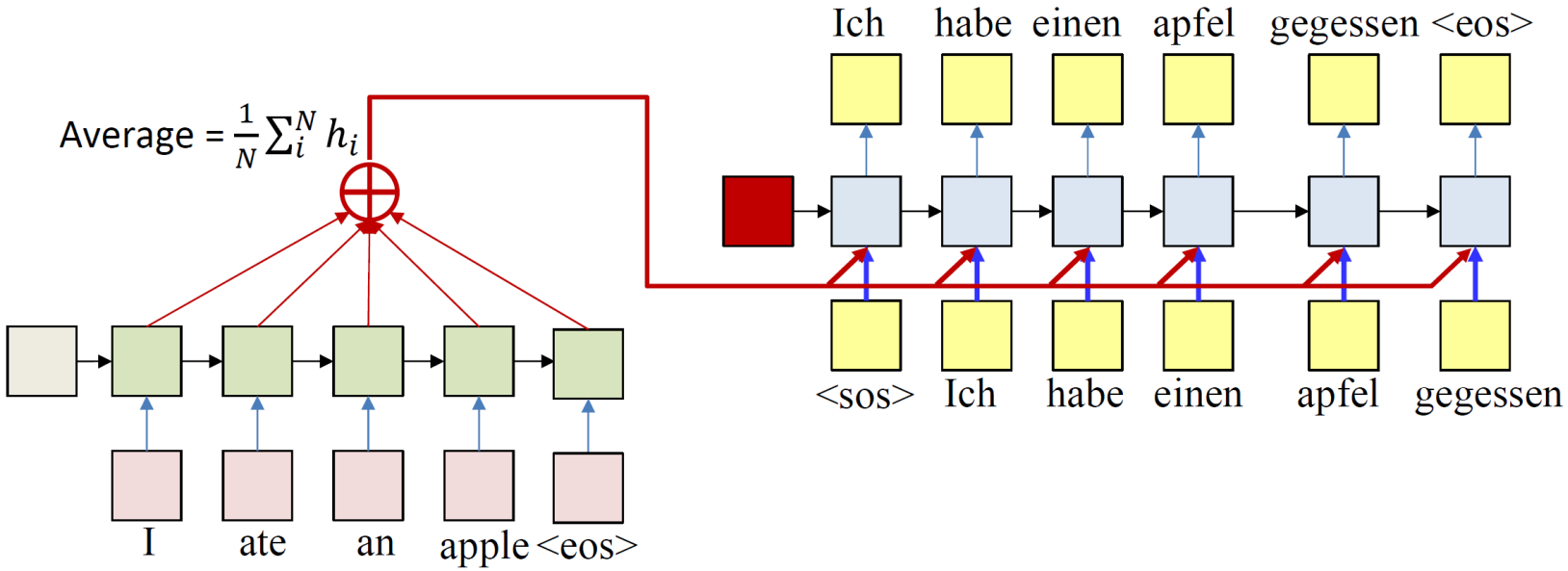
Simple Seq2Seq Architecture

- Every output is related to the input directly:
  - Not sufficient to have the encoder hidden state to only the initial state of the decoder
  - Misses the direct relation of the outputs to the inputs



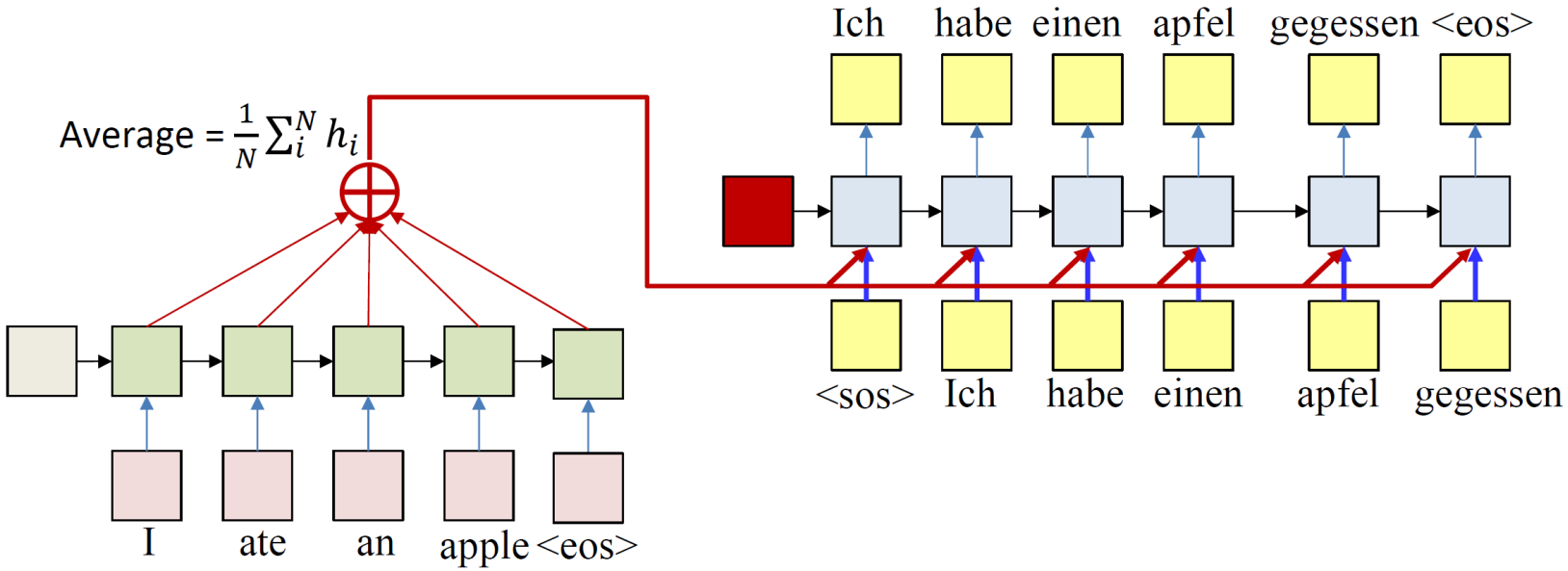
A Simple Solution: Using all input hidden states

- Compute the average of all encoder hidden states
  - Fed this average to every stage of the decoder
  - The initial decoder hidden state is now separate from the encoder, and may be a learnable parameter



A Simple Solution: Using all input hidden states

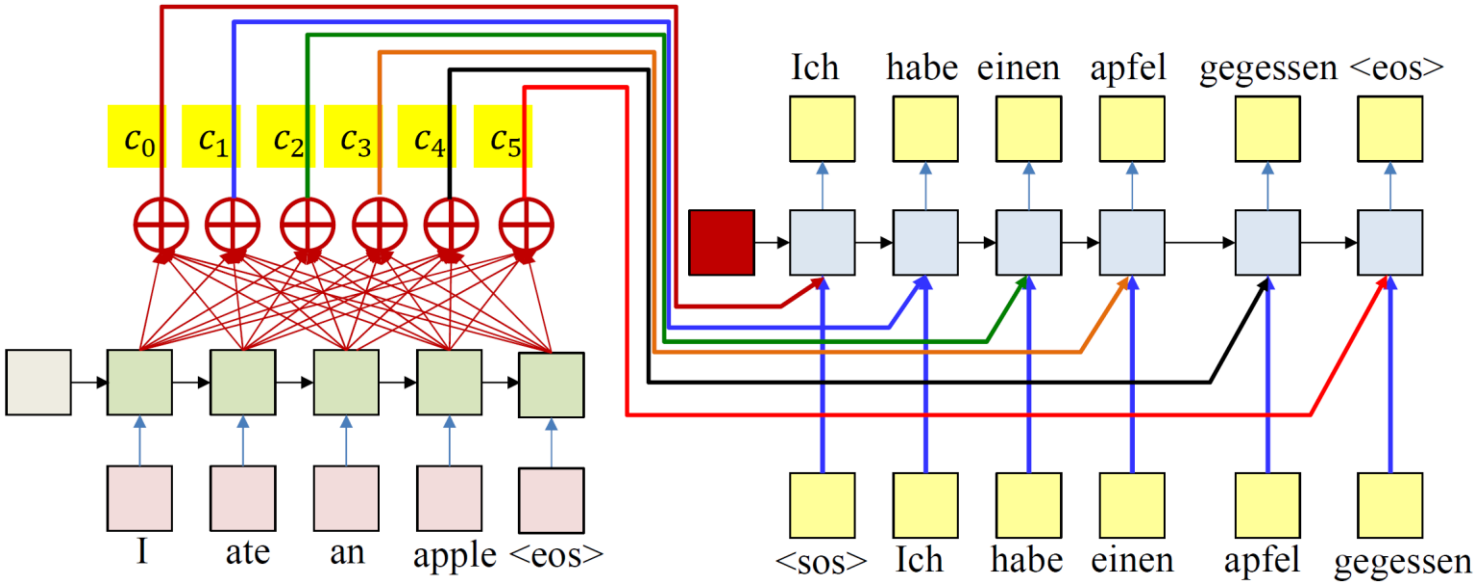
- Problem: The average applies the same weight to every input
  - It supplies the same average to every output word
  - In practice, different outputs may be related to different inputs ("Ich" is most related to "I", and "habe" and "gegessen" are both most related to "ate")



A Simple Solution: Using all input hidden states

- Solution: Use a different weighted average for each output word:

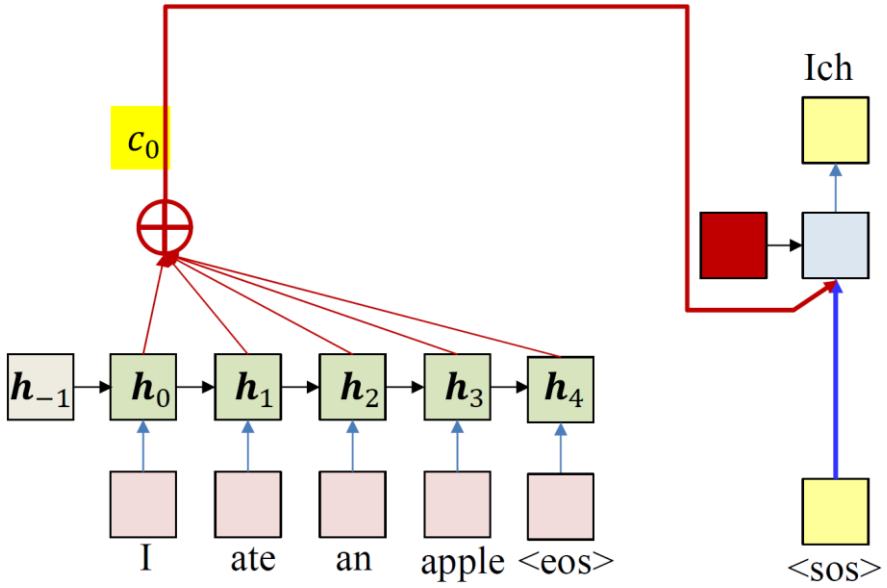
$$c_t = \frac{1}{N}\sum_{i}^{N} w_i(t) h_i$$



A Simple Solution: Using all input hidden states

- Use a different weighted average for each output word:

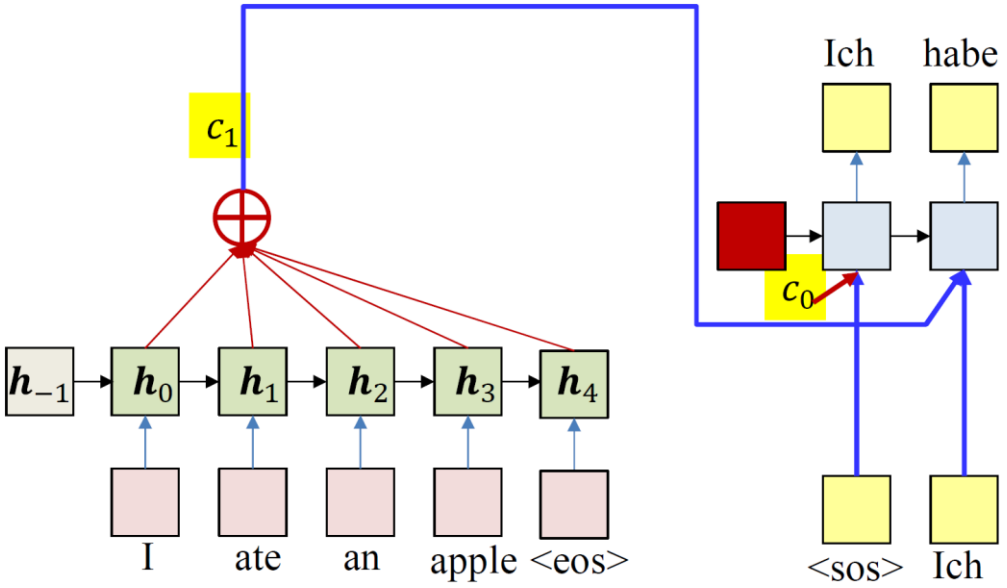
$$c_0 = \frac{1}{N}\sum_{i}^{N} w_i(0)h_i$$



A Simple Solution: Using all input hidden states

- Use a different weighted average for each output word:

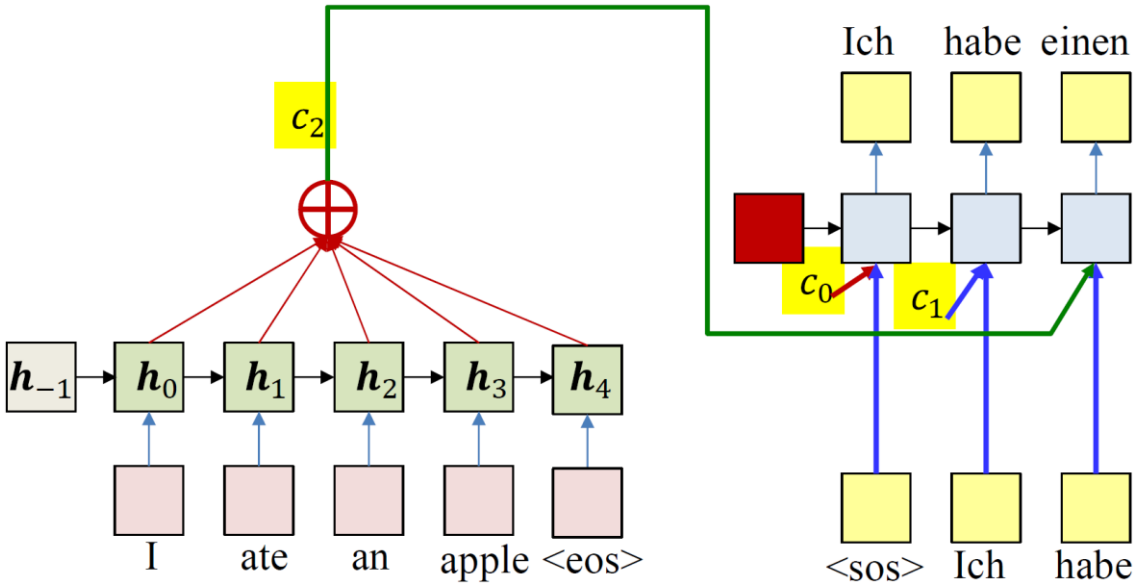
$$c_1 = \frac{1}{N}\sum_{i}^{N} w_i(1)h_i$$



A Simple Solution: Using all input hidden states

- Use a different weighted average for each output word:

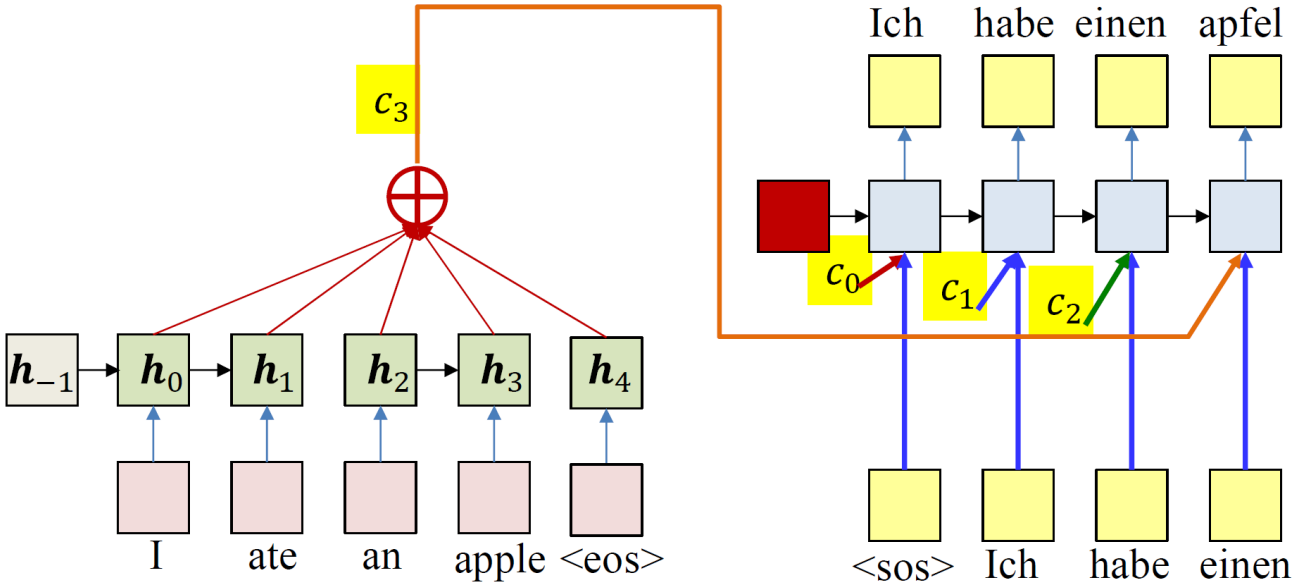
$$c_2 = \frac{1}{N}\sum_{i}^{N} w_i(2)h_i$$



A Simple Solution: Using all input hidden states

- Use a different weighted average for each output word:

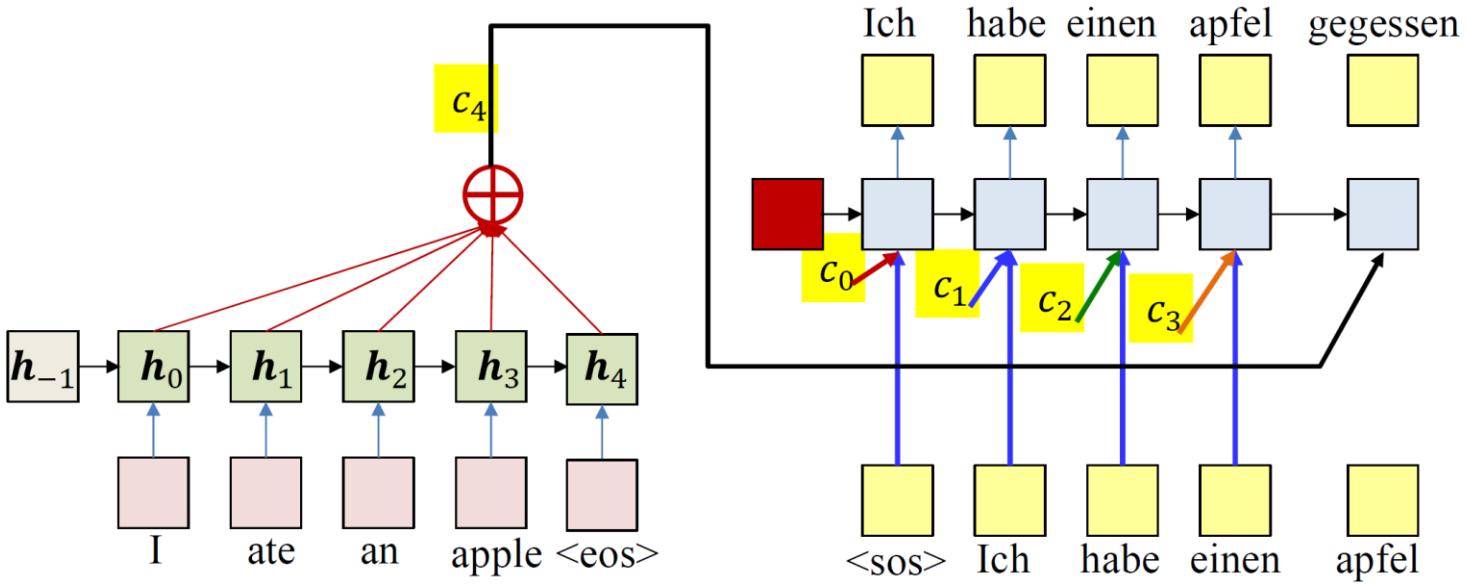
$$c_3 = \frac{1}{N}\sum_{i}^{N} w_i(3)h_i$$



A Simple Solution: Using all input hidden states

- Use a different weighted average for each output word:

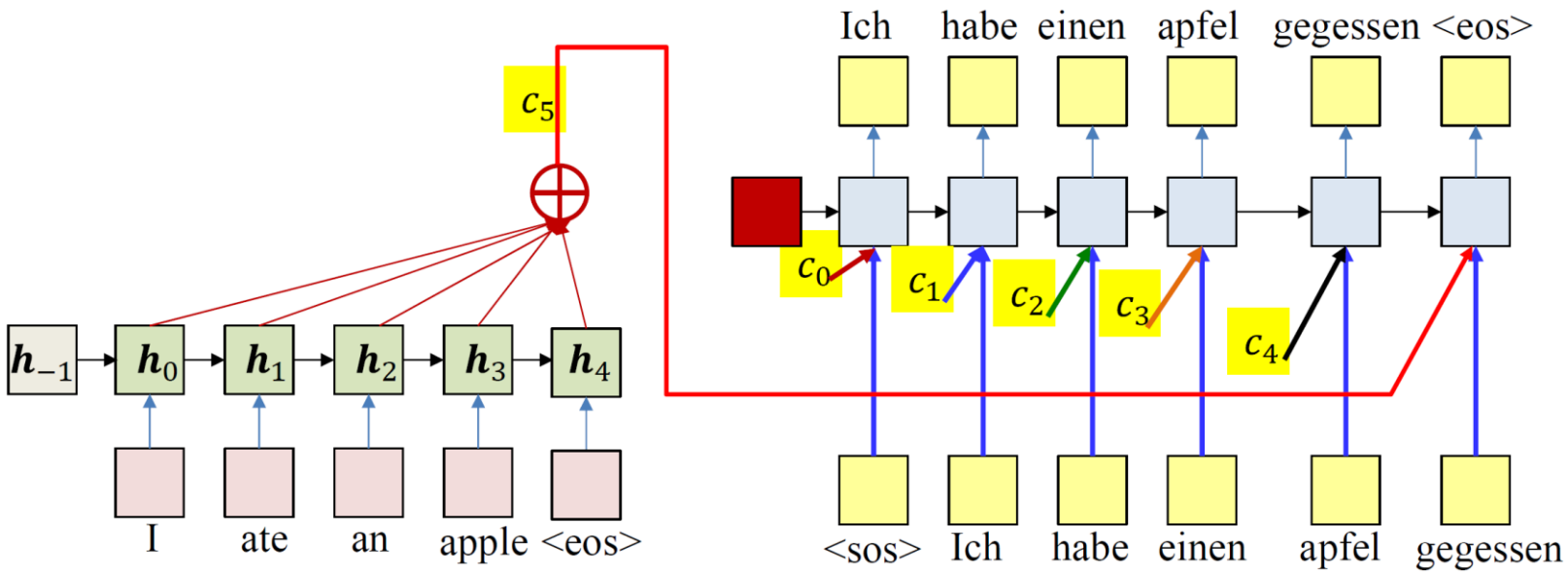
$$c_4 = \frac{1}{N}\sum_{i}^{N} w_i(4)h_i$$



A Simple Solution: Using all input hidden states

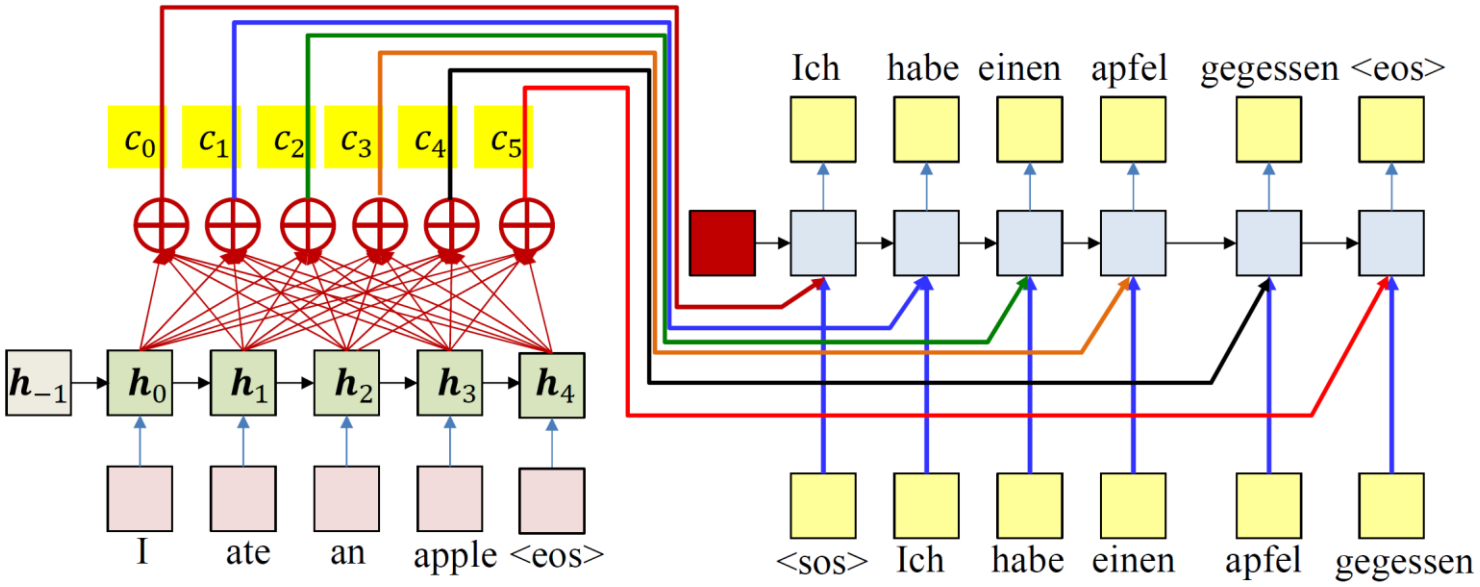
- Use a different weighted average for each output word:

$$c_5 = \frac{1}{N}\sum_{i}^{N} w_i(5)h_i$$



A Simple Solution: Using all input hidden states

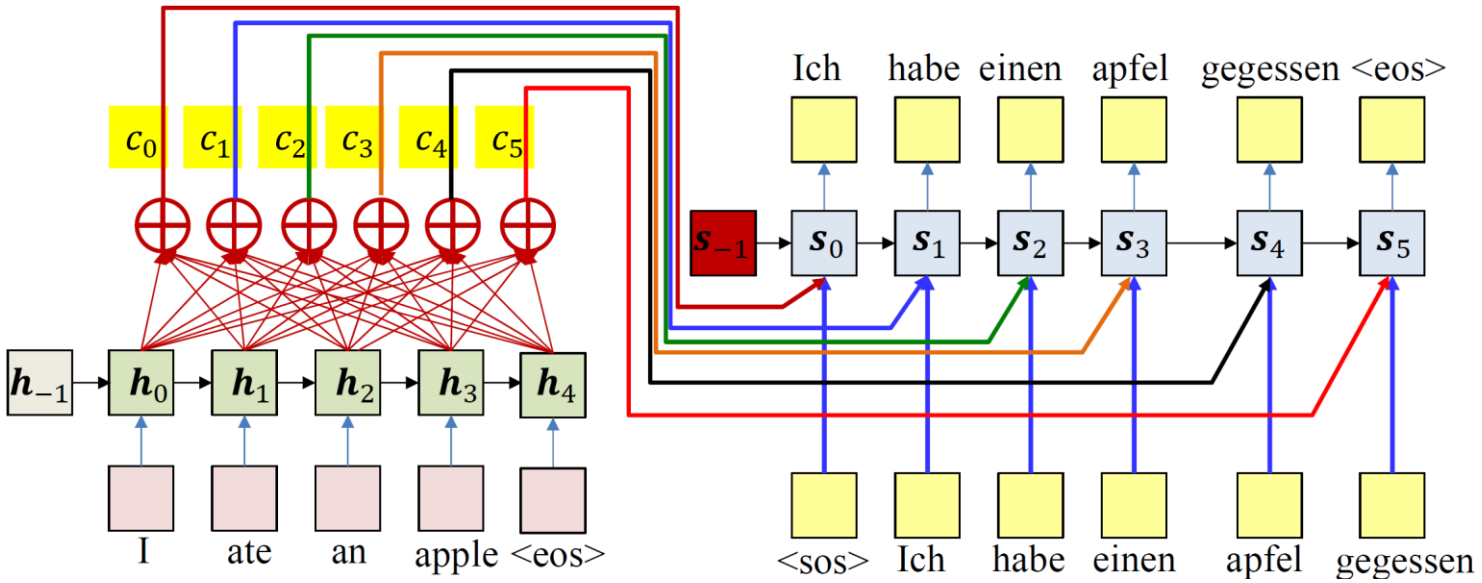
- This solution will work if the weights $w_i(k)$ can somehow be made to "focus" on the right input word, e.g. when predicting the word "apfel", $w_3(4)$, the weight for "apple" must be high while the rest must be low.



Attention Model!

- Attention weights, $w_i(t)$, are dynamically computed as functions of decoder state:

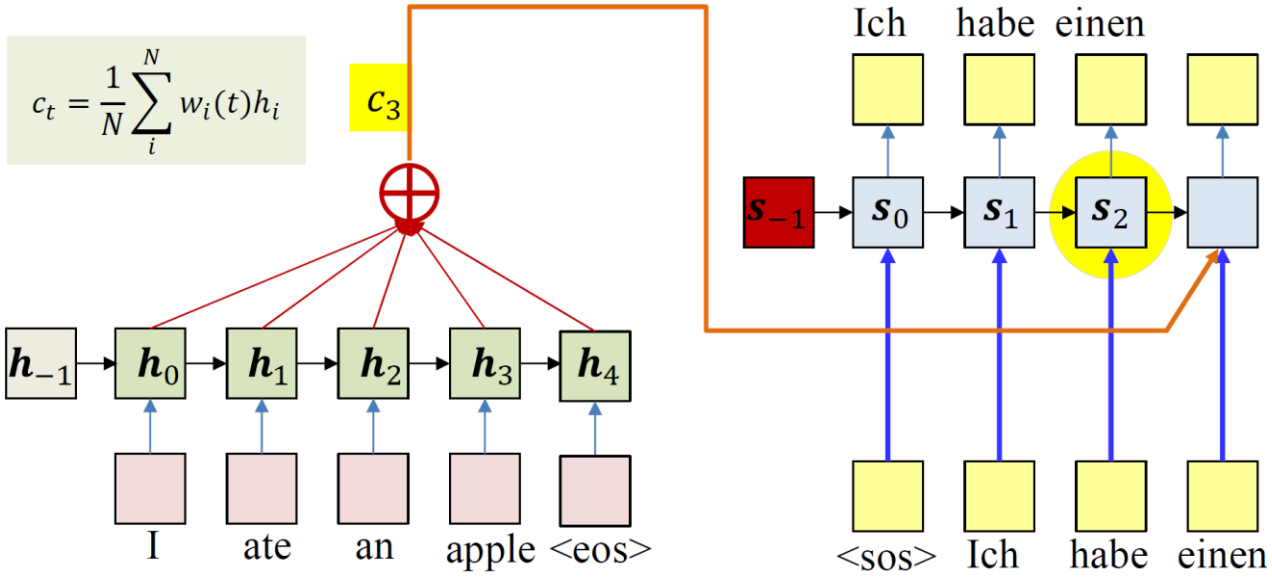
$$c_t = \frac{1}{N}\sum_{i}^{N} w_i(t) h_i$$



Attention Model!

- The "attention" weights $w_i(\mathrm{t})$ at time t must be computed from available information at time t. The primary information is $s(\mathrm{t}-1)$, the state at time $t-1$.

$$a_t = a(h_i, s_{t-1})$$

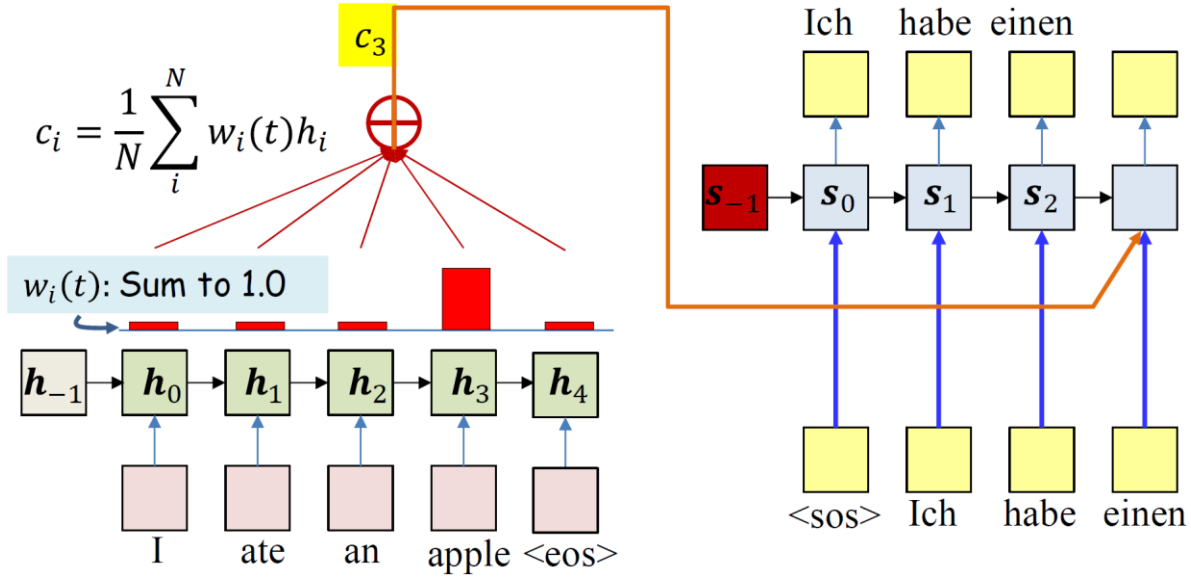


Requirement on attention weights

- The weights $w_i(\mathrm{t})$ must be positive and sum to 1.0:
  - Be a distribution
  - Ideally, they must be high for the most relevant inputs for the i^{th} output and low elsewhere

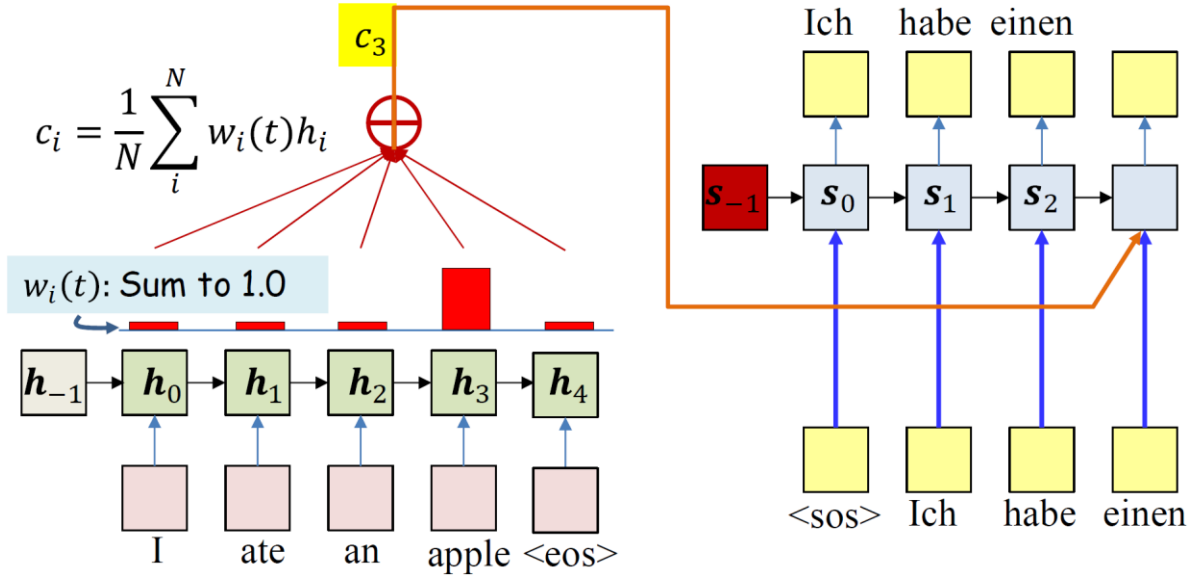
$$e_i(\mathrm{t}) = g(h_i, s_{t-1})$$

$$w_i(\mathrm{t}) = \frac{exp\left(e_i(\mathrm{t})\right)}{\sum_j exp\left(e_j(\mathrm{t})\right)} = softmax\left(e_i(\mathrm{t})\right)$$



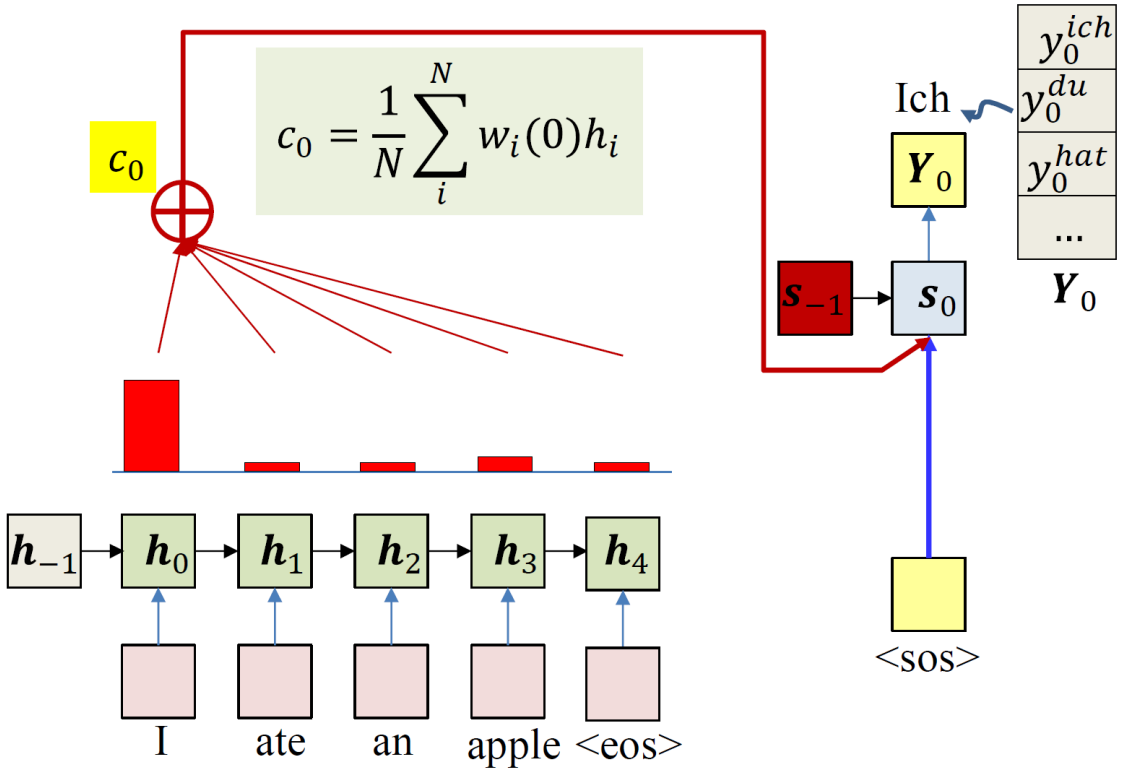
Requirement on attention weights

- The weights $w_i(\mathrm{t})$ must be positive and sum to 1.0:
  - $e_i(\mathrm{t}) = g(h_i, s_{t-1})$
  - $w_i(\mathrm{t}) = \frac{exp(e_i(\mathrm{t}))}{\sum_j exp(e_j(\mathrm{t}))} = softmax(e_i(\mathrm{t}))$
- Typical fixed and learnable $g(\cdot,\cdot)$:
  - $g(h_i, s_{t-1}) = h_i^T s_{t-1}$ $(*)$
  - $g(h_i, s_{t-1}) = h_i^T W_g s_{t-1}$
  - $g(h_i, s_{t-1}) = v_g^T tanh\left(W_g \begin{bmatrix} h_i \\ s_{t-1} \end{bmatrix}\right)$
  - $g(h_i, s_{t-1}) = MLP([h_i \quad s_{t-1}])$



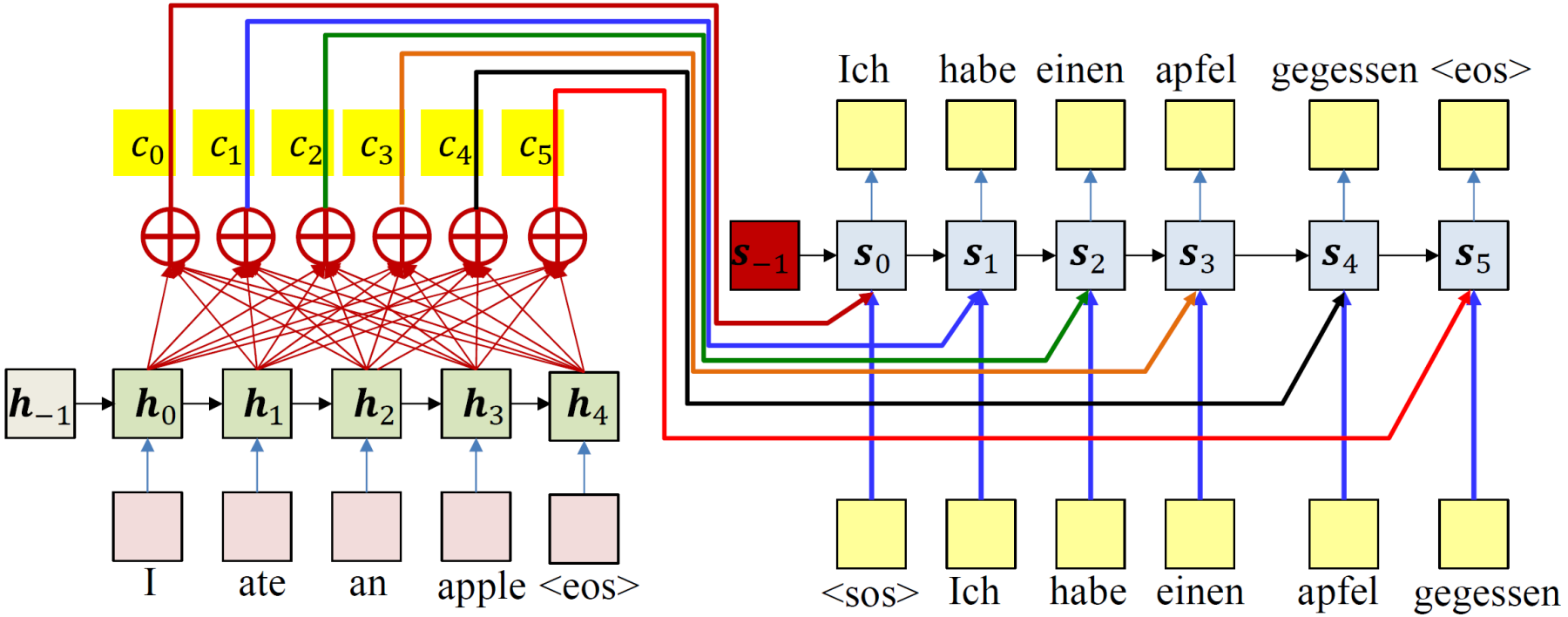
Inference (Translating)

- Pass the input through the encoder to produce hidden representations $\{h_i\}'s$
- Before decoding we need s_{-1}
  - A learned parameter
  - Set to some fixed value (0/1 vector)
  - The average of all the encoder $mean\{h_i\}$
- Compute $w_i(0)$ based on $\{h_i\}'s$ and s_{-1}
- Fed c_0 and $\langle sos \rangle$ and get s_0 and Y_0
- Compute $w_i(1)$ based on $\{h_i\}'s$ and s_0
- Fed c_1 and Y_0 and get s_1 and Y_1
- Until get $\langle eos \rangle$



Attention-Based Decoding

- General Architecture:

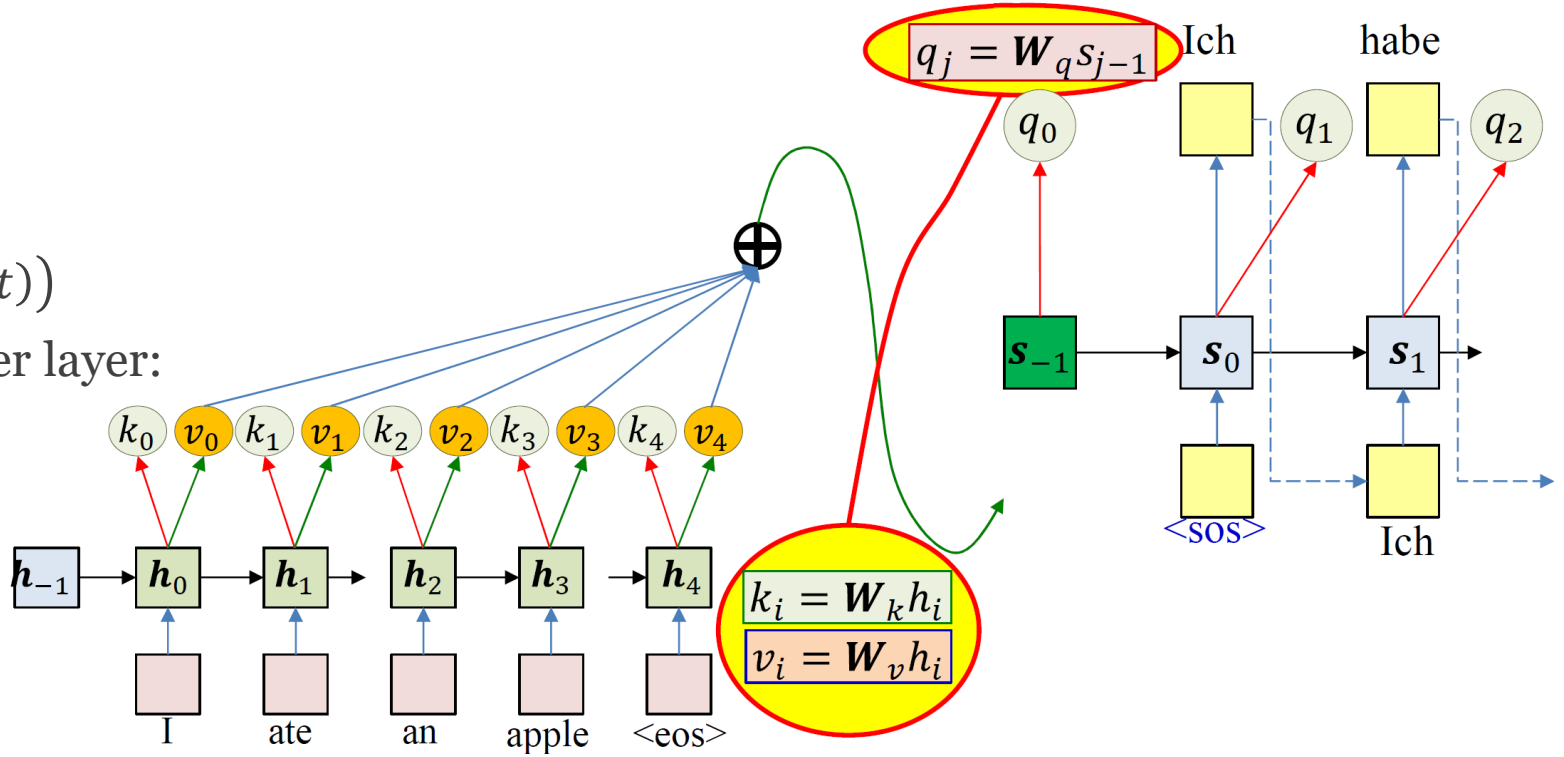


Modification: Query-Key-Value

- What is Q-K-V:
- By Example:
  - Youtube, the search engine will map your Query (text in the search bar) against a set of Keys (video title, description, etc.) associated with candidate videos in their database, then present you the best matched videos (Values).
- You enter a word in search box (Query)
- Video title or description (Keys)
- Best match video (Values)

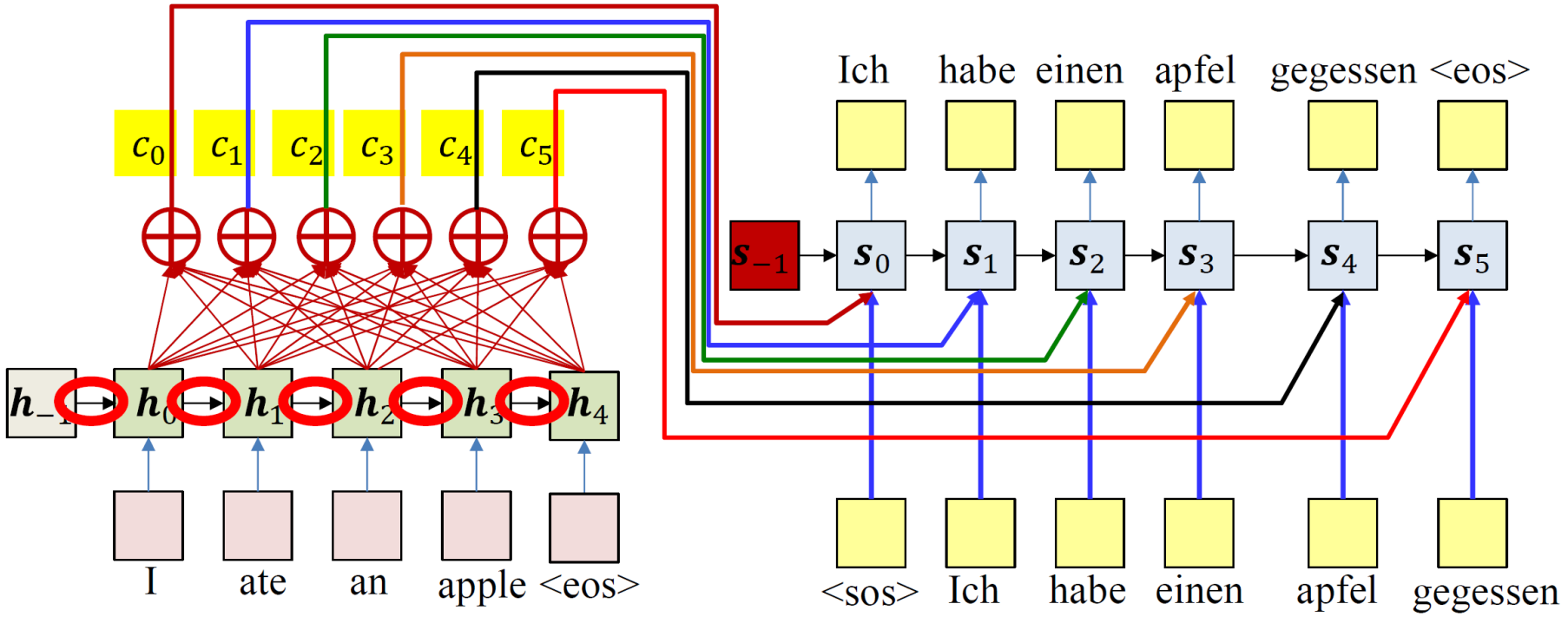
Modification: Query-Key-Value

- There are three new learnable matrices: $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v$
- $\boldsymbol{k}_i = \boldsymbol{W}_k h_i$
- $\boldsymbol{v}_i = \boldsymbol{W}_v h_i$
- $\boldsymbol{q}_j = \boldsymbol{W}_q s_{j-1}$
- $e_i(t) = g(\boldsymbol{k}_i, \boldsymbol{q}_t)$
- $w_i(t) = softmax\big(e_i(t)\big)$
- Fed to hidden decoder layer:
- $\sum_i w_i(t)\, \boldsymbol{v}_i$



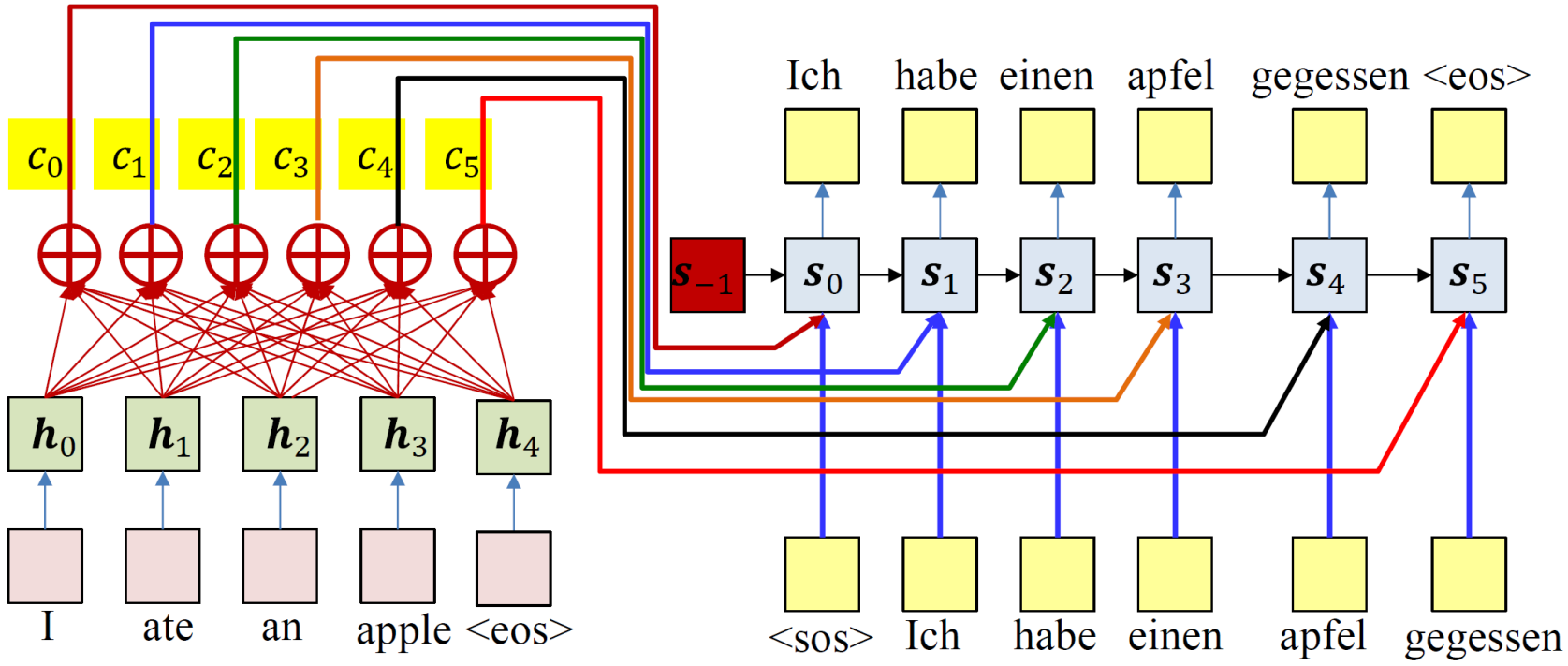
Attention-Based Decoding

- Problem: If the decoder is automatically figuring out which words of the input to attend to at each time, is recurrence in the input even necessary?



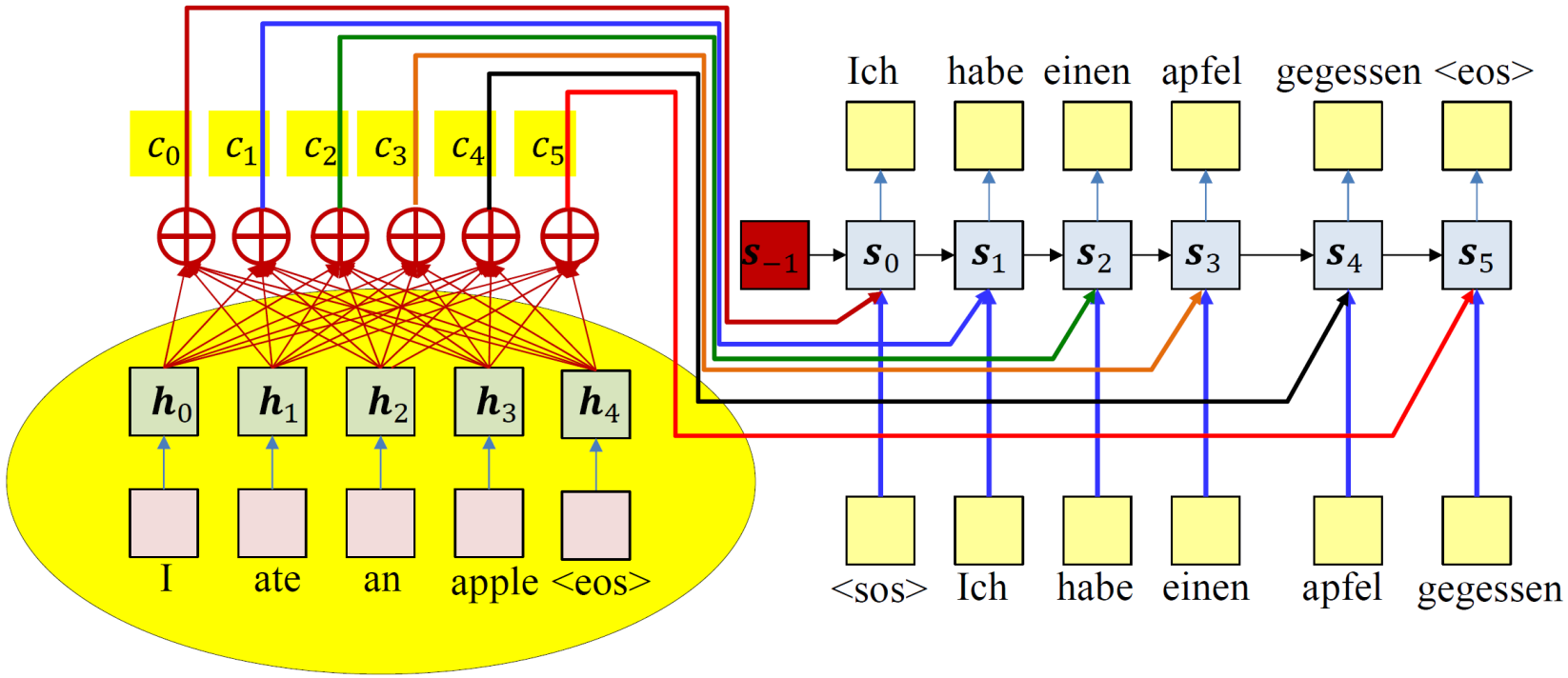
Non-Recurrent Encoder

- Modification: Let us eliminate the recurrence in the encoder



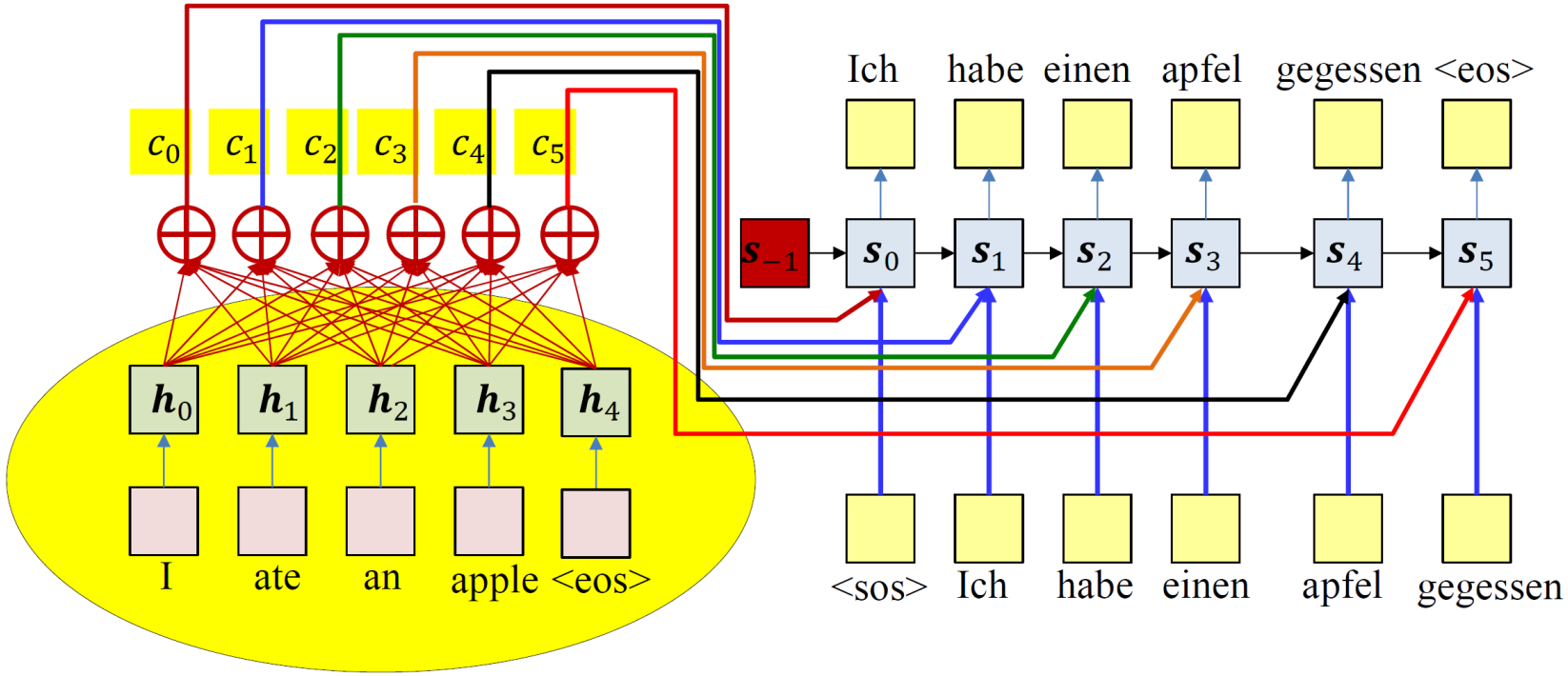
Non-Recurrent Encoder

- Problem: This will eliminate context-specificity in the encoder embeddings.



Non-Recurrent Encoder

- Solution: Use the attention framework itself to introduce context specificity in embeddings: Self Attention



Self Attention

- The updated representation for the word is the attention-weighted sum of the values for all words including itself:
  - $\boldsymbol{k}_i = \boldsymbol{W}_k h_i$
  - $\boldsymbol{v}_i = \boldsymbol{W}_v h_i$
  - $\boldsymbol{q}_i = \boldsymbol{W}_q h_i$
  - $\boldsymbol{e}_{ij} = \boldsymbol{q}_i^T \boldsymbol{k}_j$
  - $w_{ij} = softmax(\boldsymbol{e}_{ij})$
  - $w_{ij} = softmax(\boldsymbol{q}_i^T \boldsymbol{k}_j)$
  - $w_{ij} = attn(\boldsymbol{q}_i, \boldsymbol{k}_j)$
  - $o_i = \sum_j w_{ij} \boldsymbol{v}_j$

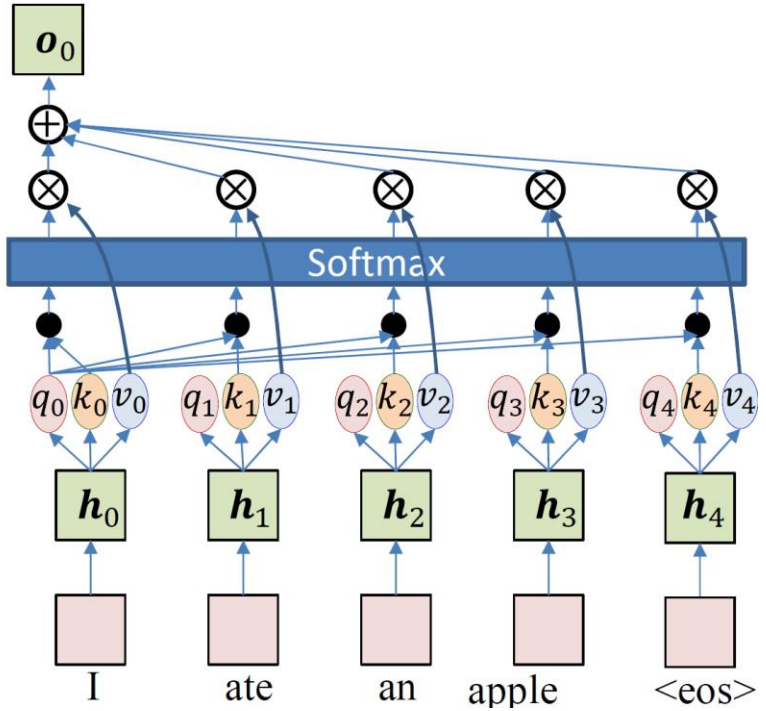
$q_i = \boldsymbol{W}_q h_i$

$k_i = \boldsymbol{W}_k h_i$

$v_i = \boldsymbol{W}_v h_i$

$w_{ij} = attn(q_i, k_{0:N})$

$o_i = \sum_j w_{ij} v_j$



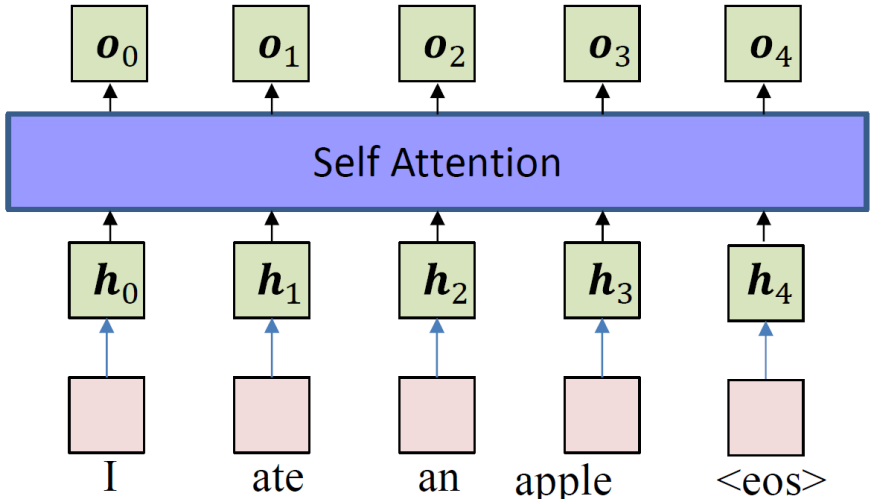
Self Attention

- Compute query-key-value sets for every word.
- This is *single-head self-attention* block

$$q_i = \boldsymbol{W}_q h_i$$
$$k_i = \boldsymbol{W}_k h_i$$
$$v_i = \boldsymbol{W}_v h_i$$

$$w_{ij} = attn(q_i, k_{0:N})$$

$$o_i = \sum_j w_{ij} v_j$$



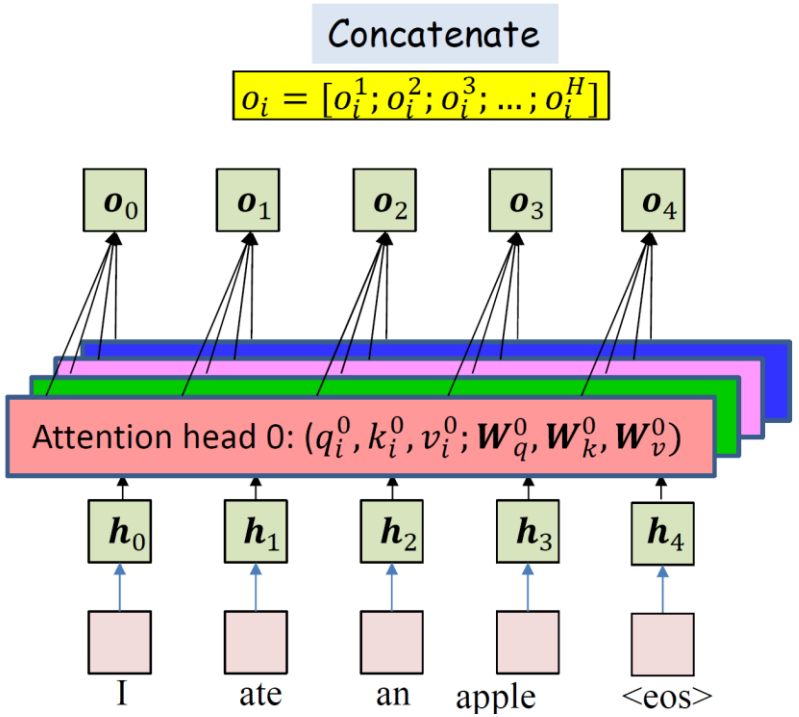
Multi-head Self Attention

- We can have multiple such attention "heads"
- Each will have an independent set of Qs-Ks-Vs (and independent attention weights)

$$q_i^a = \boldsymbol{W}_q^a h_i$$
$$k_i^a = \boldsymbol{W}_k^a h_i$$
$$v_i^a = \boldsymbol{W}_v^a h_i$$

$$w_{ij}^a = attn(q_i^a, k_{0:N}^a)$$

$$o_i^a = \sum_j w_{ij}^a v_j^a$$



Multi-head Self Attention

- Multiple self-attention modules in parallel

$$q_i^a = \boldsymbol{W}_q^a h_i$$

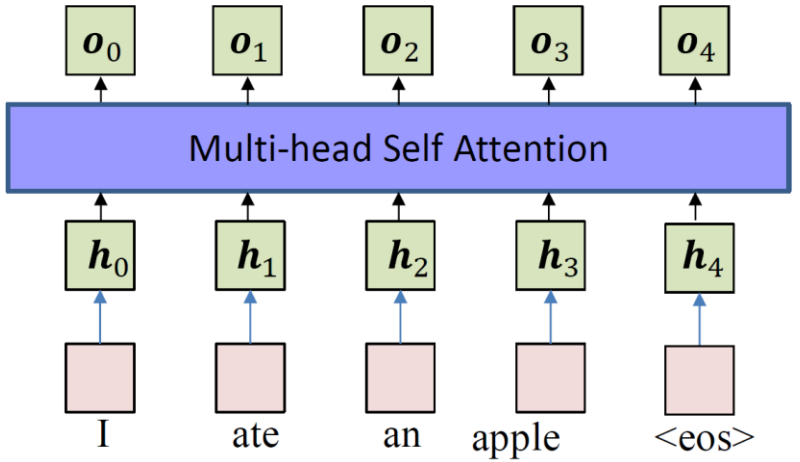
$$k_i^a = \boldsymbol{W}_k^a h_i$$

$$v_i^a = \boldsymbol{W}_v^a h_i$$

$$w_{ij}^a = attn(q_i^a, k_{0:N}^a)$$

$$o_i^a = \sum_j w_{ij}^a v_j^a$$

$$o_i = [o_i^1; o_i^2; o_i^3; \dots; o_i^H]$$



Multi-head Self Attention

- Typically, the output of the multi-head self attention is passed through one or more regular feed-forward layers.

$$q_i^a = \boldsymbol{W}_q^a h_i$$

$$k_i^a = \boldsymbol{W}_k^a h_i$$

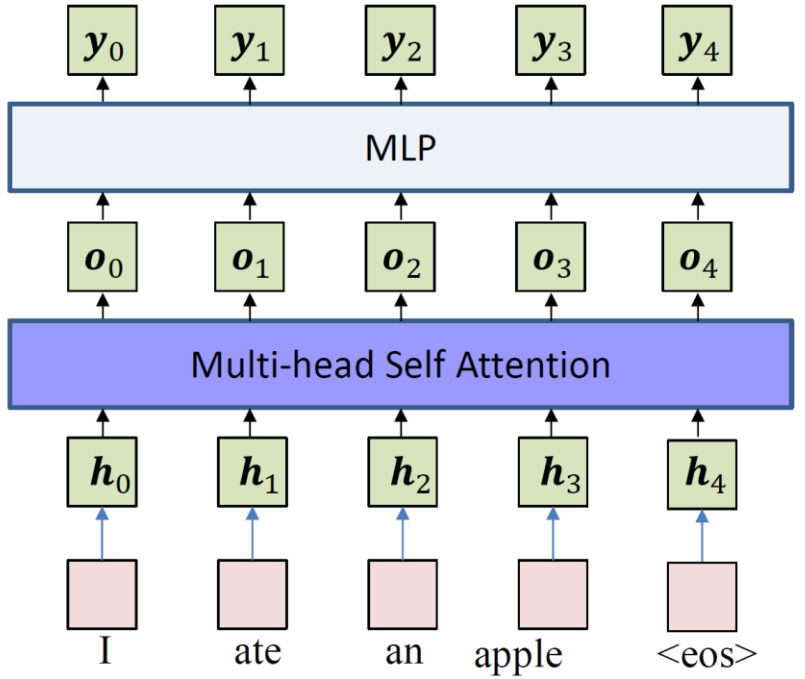
$$v_i^a = \boldsymbol{W}_v^a h_i$$

$$w_{ij}^a = attn(q_i^a, k_{0:N}^a)$$

$$o_i^a = \sum_j w_{ij}^a v_j^a$$

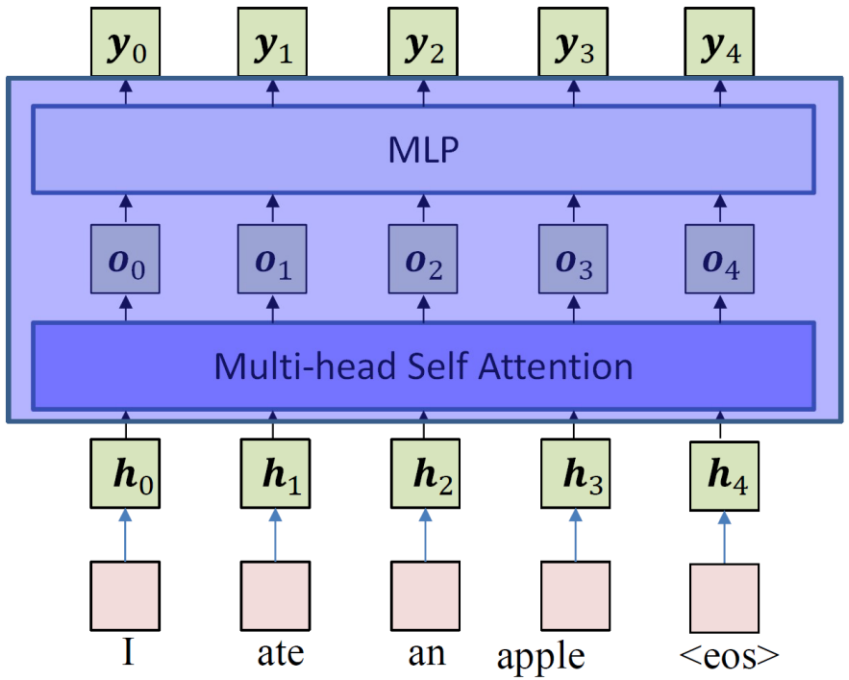
$$o_i = [o_i^1; o_i^2; o_i^3; \ldots; o_i^H]$$

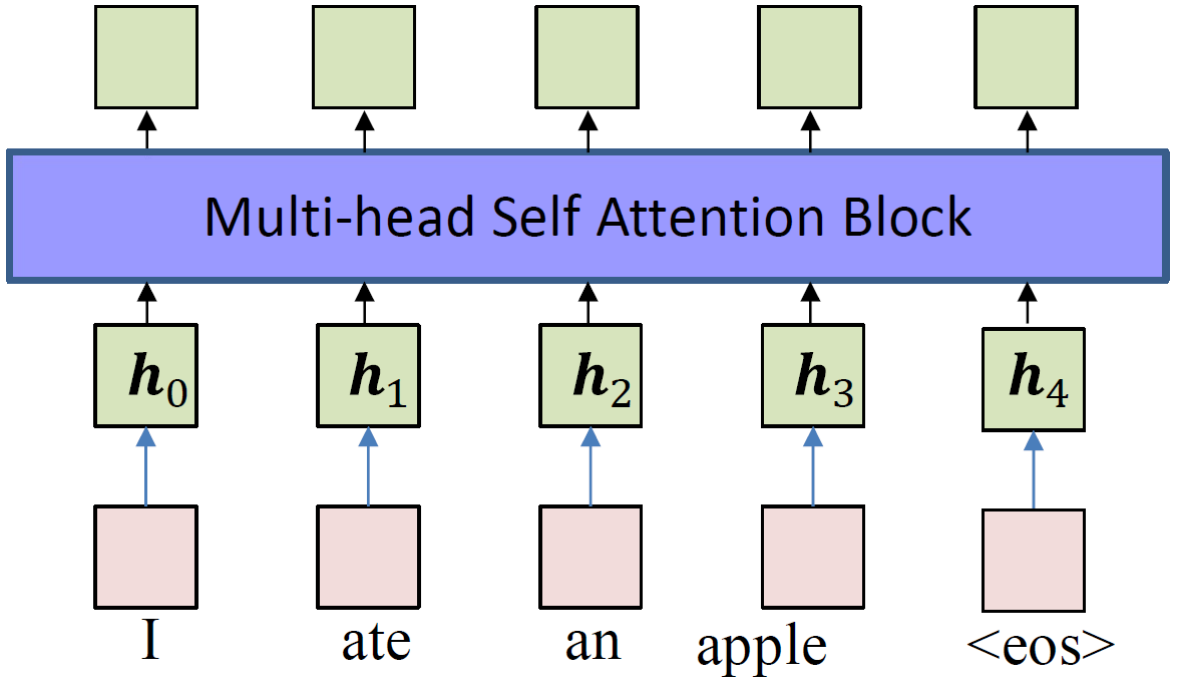
$$y_i = MLP(o_i)$$



Multi-head Self Attention Block

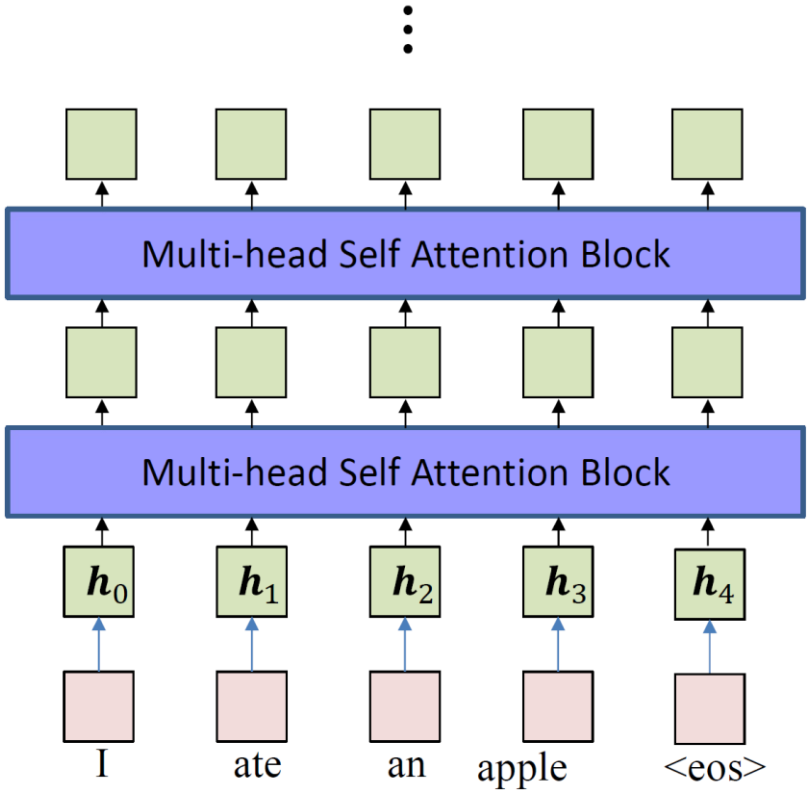
- The entire unit, including multi-head self-attention module followed by MLP is a *multi-head self-attention block*





Multi-head Self Attention Block

- The encoder can include many layers of such blocks
- No need for recurrence

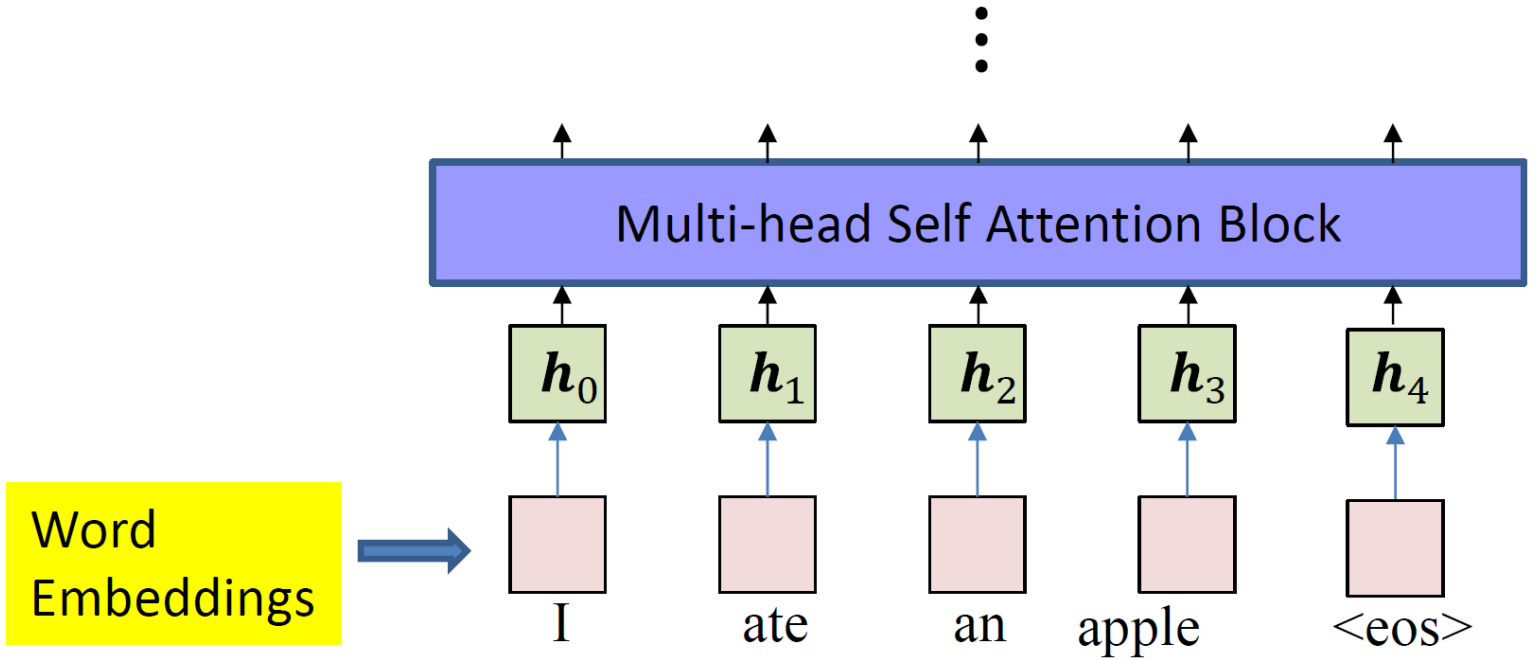


Positional Encoding

- The encoder in a sequence-to-sequence model can replace recurrence through a series of "multi-head self attention" blocks.
- But this still ignores relative position
- A context word ONE word away is different from one TEN words away
- The attention framework does not take distance into consideration

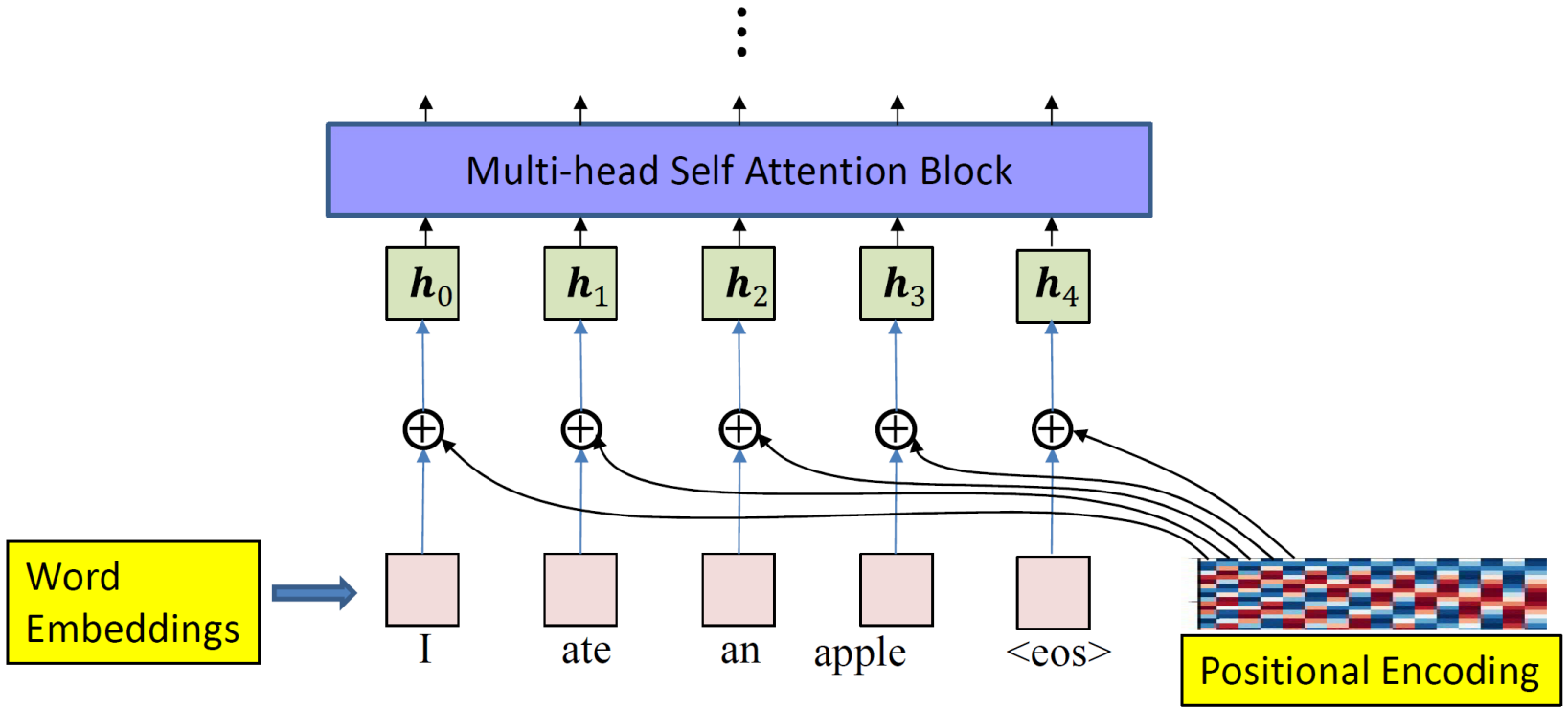
Positional Encoding

- Note that the inputs are actually word embeddings



Positional Encoding

- Solution: Add a "positional" encoding to them to capture the relative distance from one another

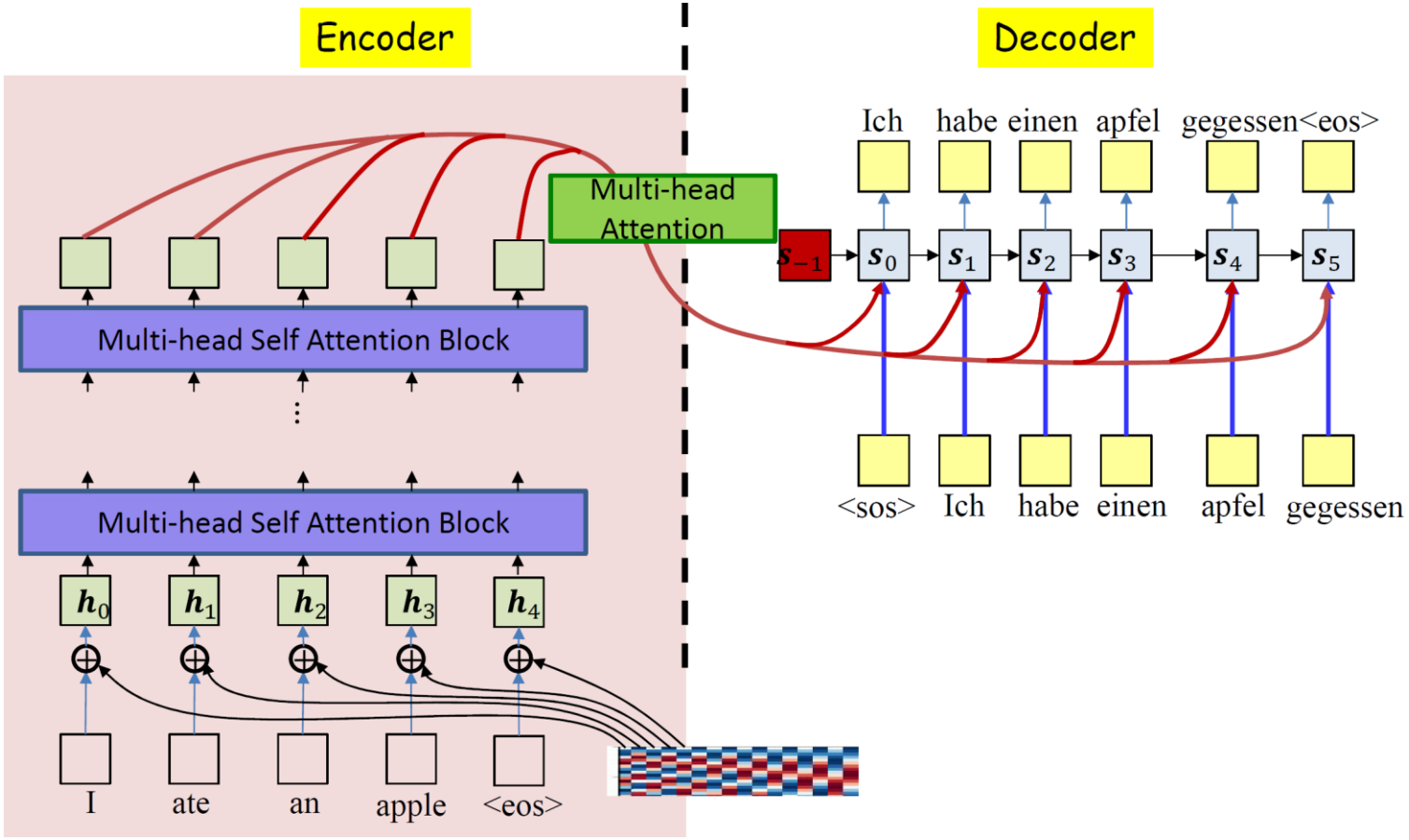


Positional Encoding

- A sequence of vectors $P_0, P_1, \dots, P_N$, to encode position:
  - Every vector is unique (and uniquely represents time)
  - Relationship between P_t and $P_{t+\tau}$ only depends on the distance between them: $P_{t+\tau} = M_\tau P_{t+\tau}$
  - The linear relationship between P_t and $P_{t+\tau}$ enables the net to learn shift invariant "gap" dependent relationships.

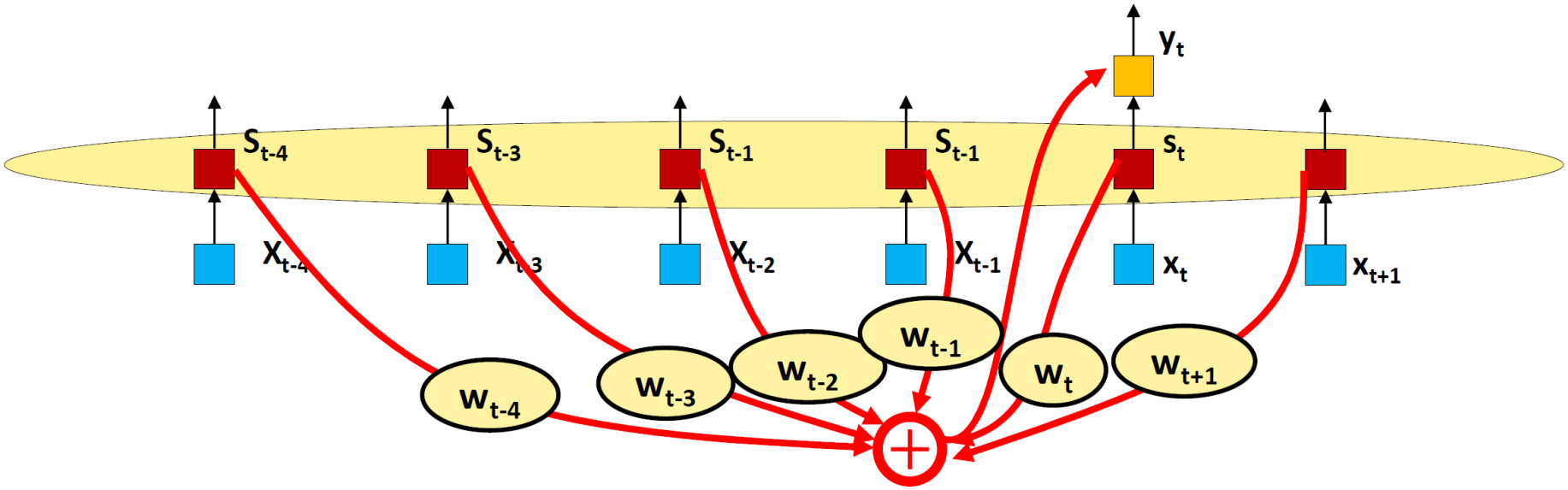
$$P_t = \begin{bmatrix} sin\omega_1 t \\ cos\omega_1 t \\ sin\omega_2 t \\ cos\omega_2 t \\ \vdots \\ sin\omega_{d/2} t \\ cos\omega_{d/2} t \end{bmatrix} \in \mathbb{R}^d, \omega_d = \frac{1}{10000^{\frac{2l}{d}}}$$

The Self-Attending Encoder



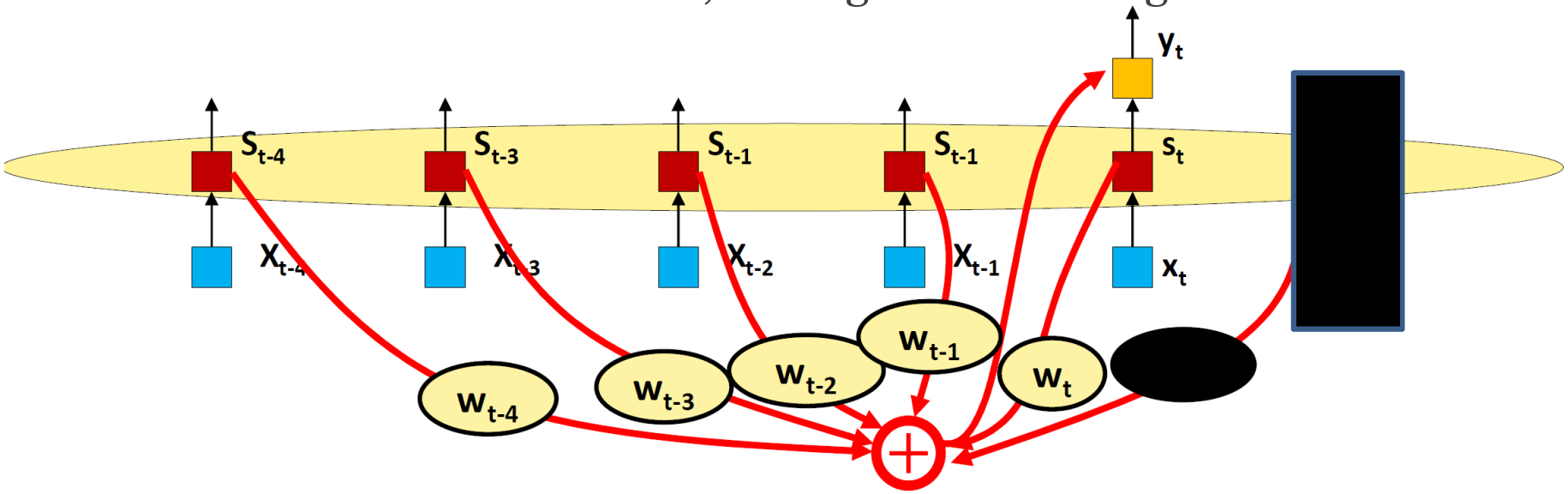
The Self-Attending Encoder

- Self attention in encoder: Can use input embedding at time $t + 1$ and further to compute output at time t, *because all inputs are available.*



The Self Attention in Decoder

- Decoder is sequential:
  - Each word is produced using the *previous* word as input
  - Only embeddings until time t are available to compute the output at time t
- The attention will have to be "masked", forcing attention weights for $t + 1$ and later to 0



Masked Self-Attention Block

- The "*masked self attention block*" includes an MLP after the masked self attention, like as encoder

$$q_i = \boldsymbol{W}_q h_i$$

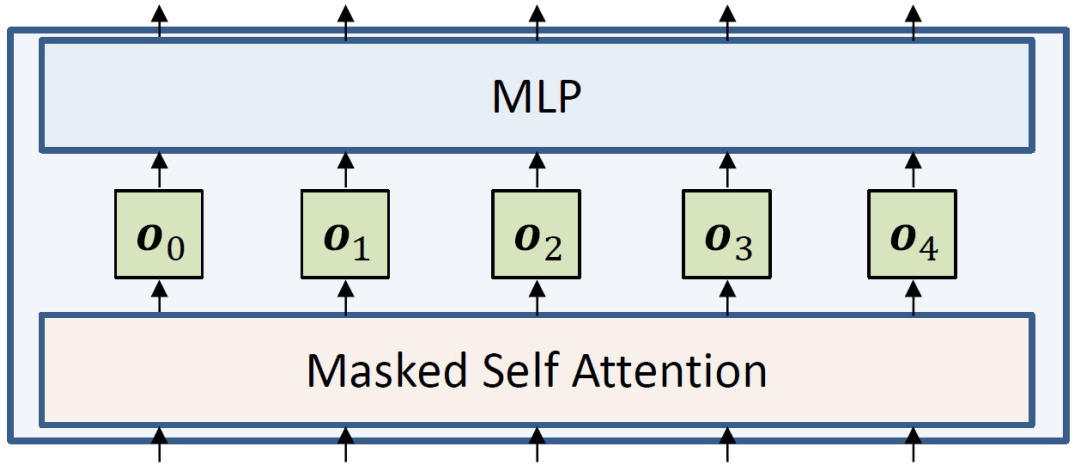
$$k_i = \boldsymbol{W}_k h_i$$

$$v_i = \boldsymbol{W}_v h_i$$

$$e_{ij} = q_i^T k_j$$

$$w_{i0}, \dots, w_{ii} = softmax(e_{i0}, \dots, e_{ii})$$

$$o_i = \sum_{j=0}^{i-1} w_{ij} v_j$$



Masked Self-Attention Block

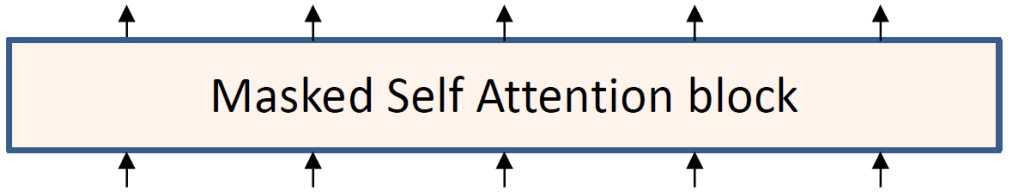
- The "*masked self attention block*" sequentially computes outputs begin to end
- Sequential nature of decoding prevents outputs from being computed in parallel, unlike in an encoder.

$$q_i = \boldsymbol{W}_q h_i$$
$$k_i = \boldsymbol{W}_k h_i$$
$$v_i = \boldsymbol{W}_v h_i$$

$$e_{ij} = q_i^T k_j$$

$$w_{i0}, \ldots, w_{ii} = softmax(e_{i0}, \ldots, e_{ii})$$

$$o_i = \sum_{j=0}^{i-1} w_{ij} v_j$$



Masked Multi-Head Self-Attention

- The "*masked multi-head self attention block*" includes multiple (independent) masked attention heads, like in the encoder.

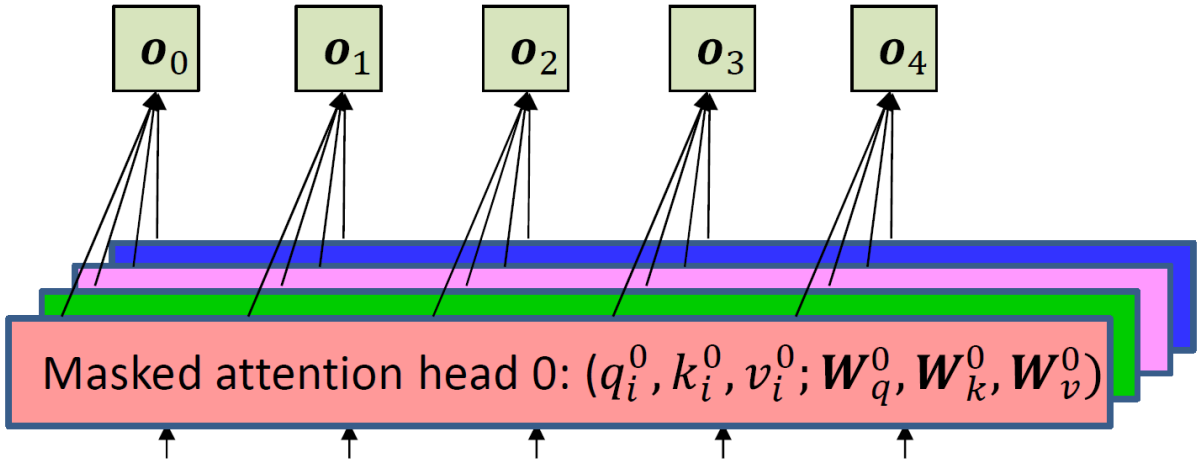
$$q_i^a = \boldsymbol{W}_q^a h_i$$

$$k_i^a = \boldsymbol{W}_k^a h_i$$

$$v_i^a = \boldsymbol{W}_v^a h_i$$

$$w_{ij}^a = attn(q_i^a, k_{0:i-1}^a)$$

$$o_i^a = \sum_j w_{ij}^a v_j^a$$



Masked Multi-Head Self-Attention Block

- The "*masked multi-head self attention block*" includes multiple masked attention heads followed by an MLP , like in the encoder.

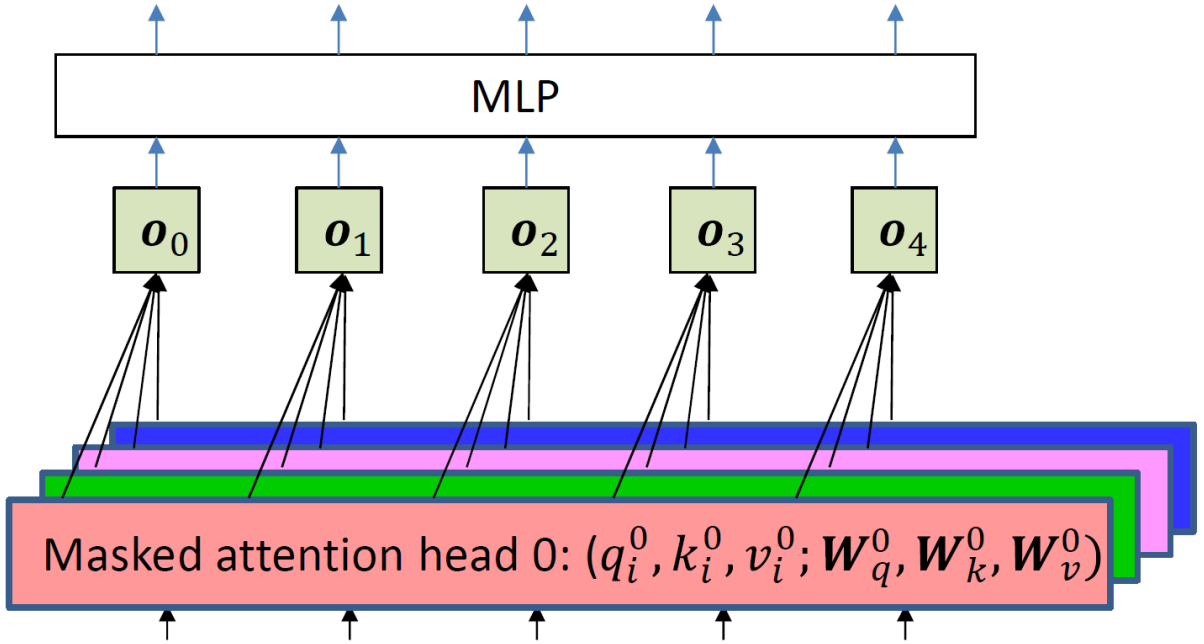
$$q_i^a = \boldsymbol{W}_q^a h_i$$

$$k_i^a = \boldsymbol{W}_k^a h_i$$

$$v_i^a = \boldsymbol{W}_v^a h_i$$

$$w_{ij}^a = attn(q_i^a, k_{0:i-1}^a)$$

$$o_i^a = \sum_j w_{ij}^a v_j^a$$



Masked Multi-Head Self-Attention Block

- The "*masked multi-head self attention block*" includes multiple masked attention heads followed by an MLP , like in the encoder.

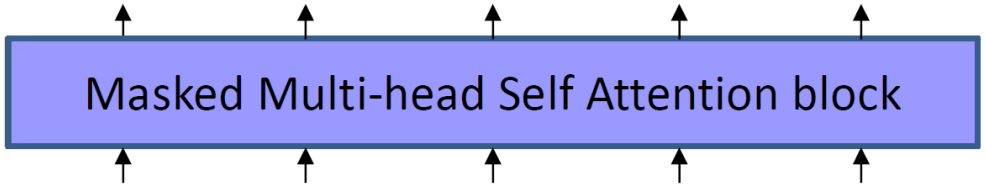
$$q_i^a = \boldsymbol{W}_q^a h_i$$

$$k_i^a = \boldsymbol{W}_k^a h_i$$

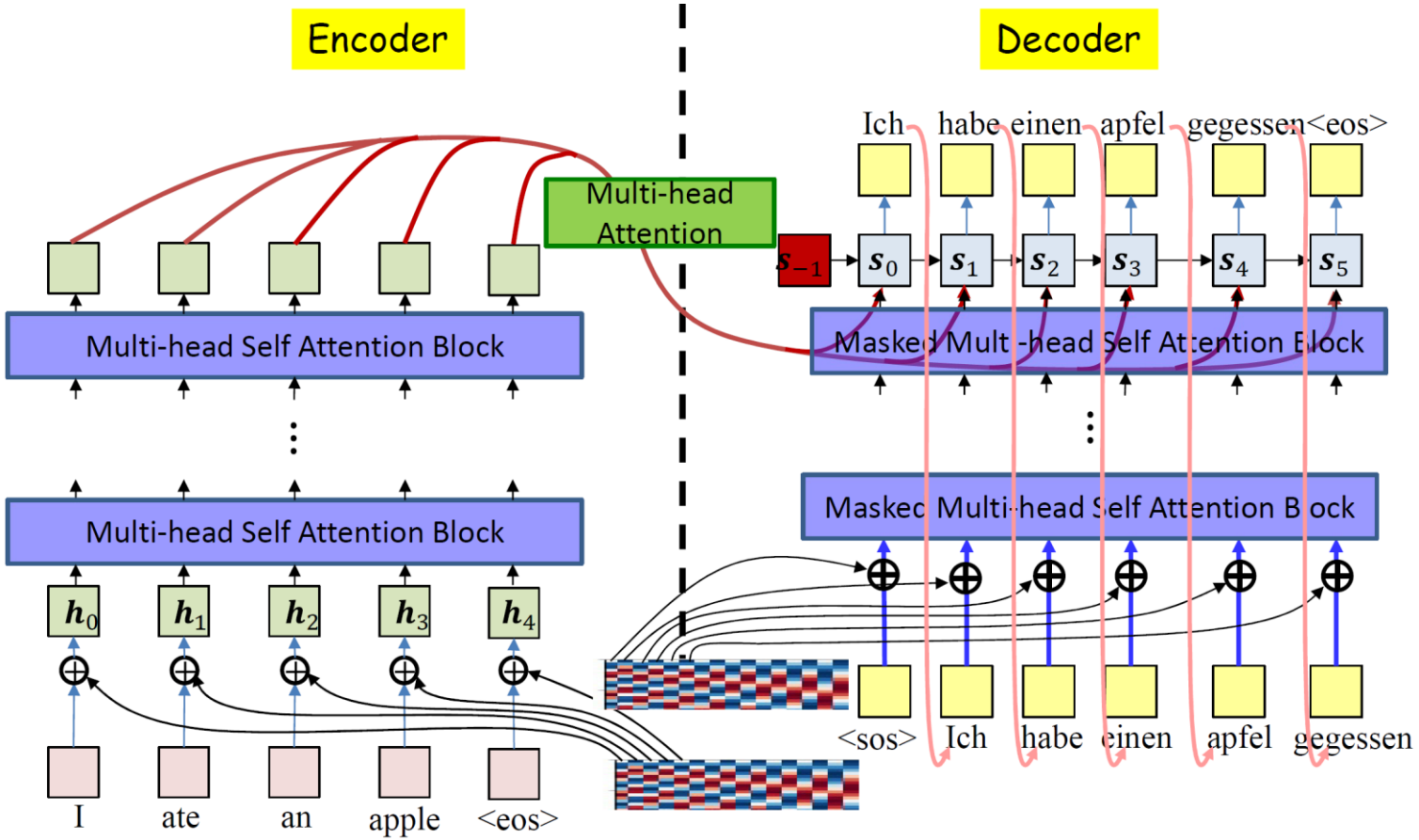
$$v_i^a = \boldsymbol{W}_v^a h_i$$

$$w_{ij}^a = attn(q_i^a, k_{0:i-1}^a)$$

$$o_i^a = \sum_j w_{ij}^a v_j^a$$



Self Attention Encoder-Decoder

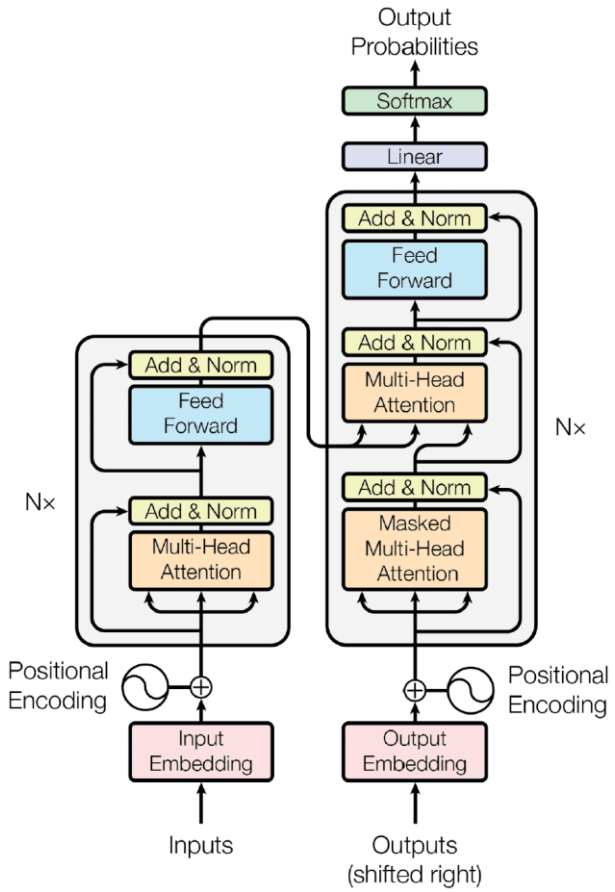


Transformers

VASWANI, ASHISH, ET AL. "*ATTENTION IS ALL YOU NEED*." ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS. 2017.

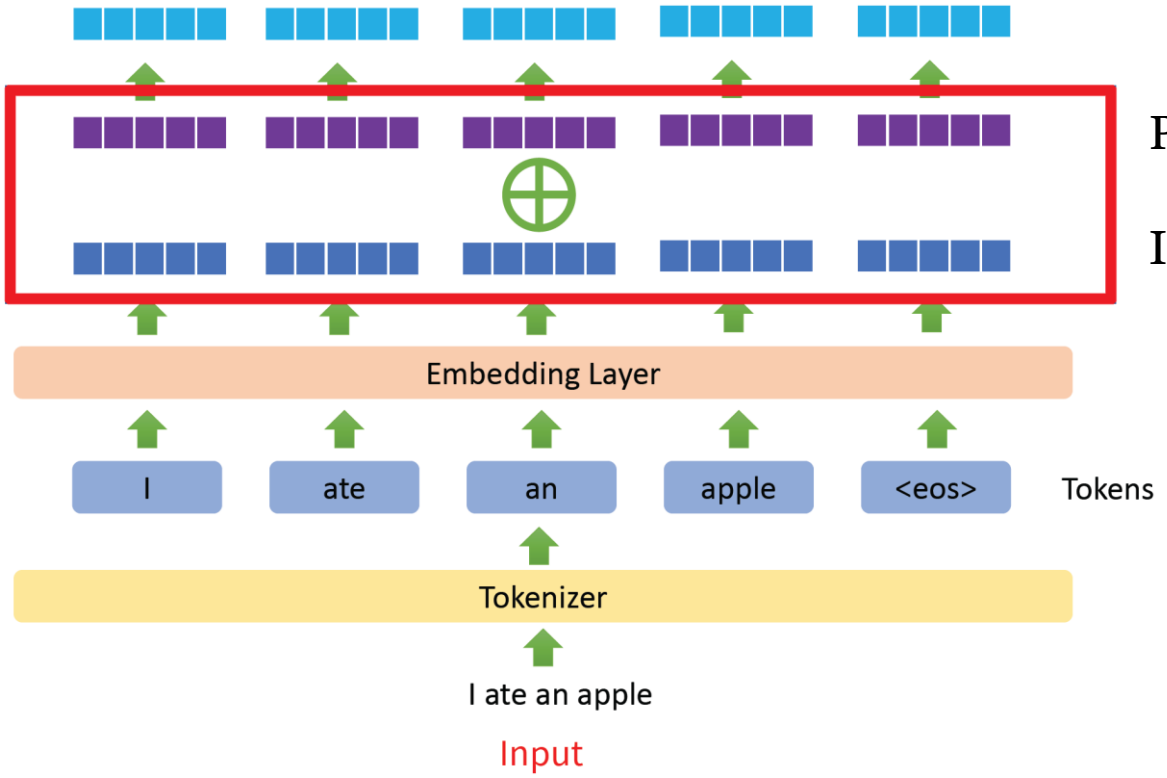
Original Transformer

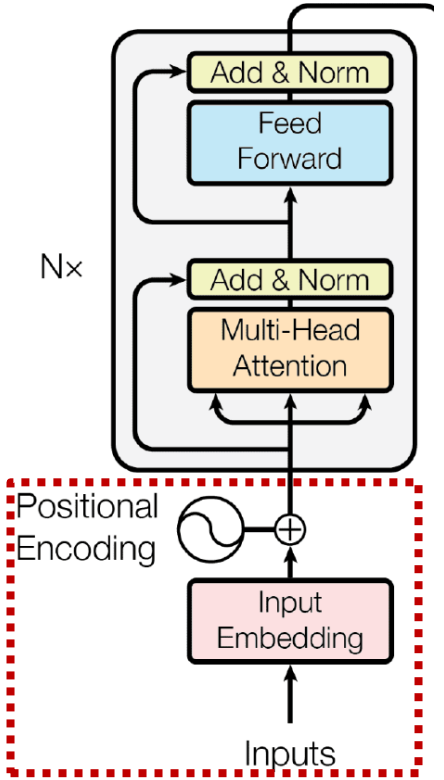
- Components:
  - Tokenizaton
  - Input/Position Embeddings
  - Residuals
  - Query/Key/Value
  - Add & Norm
  - Encoder/Decoder
  - Attention/Self Attention
  - Multi Head Attention/Masked Attention
  - Encoder Decoder Attention
  - Output Probabilities/Logits/Softmax



Inputs Embeddings

- Generate Input Embeddings:





Inputs Embeddings Algorithms

- Algorithms:

Algorithm 1: Token embedding.

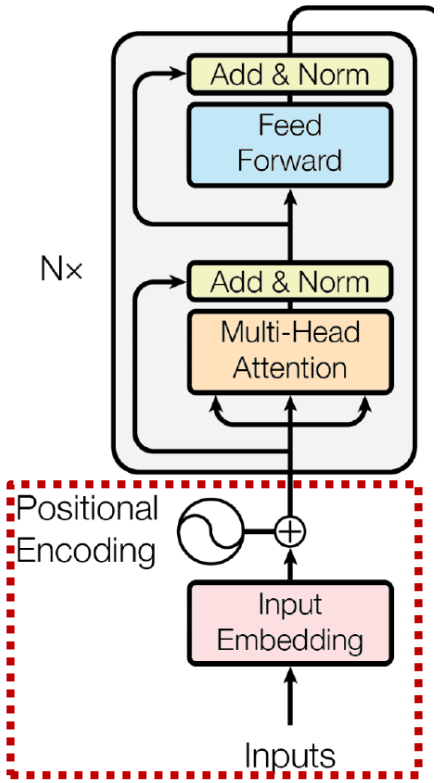
Input: $v \in V \cong [N_V]$, a token ID.
Output: $\boldsymbol{e} \in \mathbb{R}^{d_e}$, the vector representation of the token.
Parameters: $\boldsymbol{W_e} \in \mathbb{R}^{d_e \times N_V}$, the token embedding matrix.

1 **return** $\boldsymbol{e} = \boldsymbol{W_e}[:, v]$

Algorithm 2: Positional embedding.

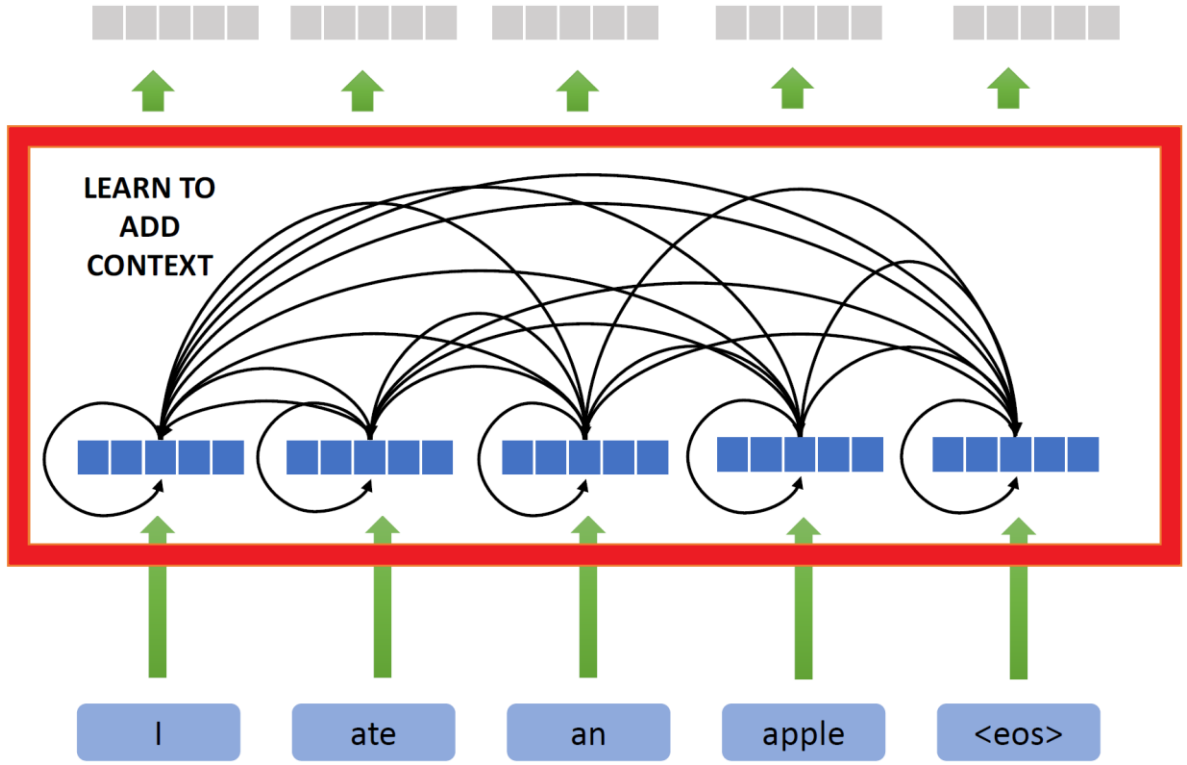
Input: $\ell \in [\ell_{max}]$, position of a token in the sequence.
Output: $\boldsymbol{e_p} \in \mathbb{R}^{d_e}$, the vector representation of the position.
Parameters: $\boldsymbol{W_p} \in \mathbb{R}^{d_e \times \ell_{max}}$, the positional embedding matrix.

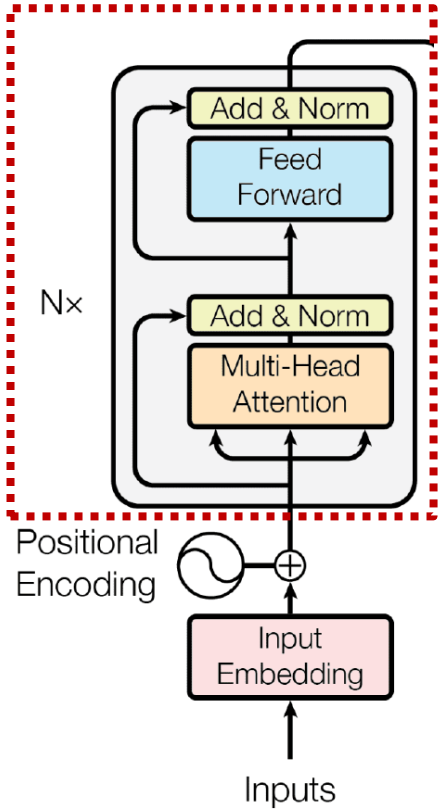
1 **return** $\boldsymbol{e_p} = \boldsymbol{W_p}[:, \ell]$



Encoder

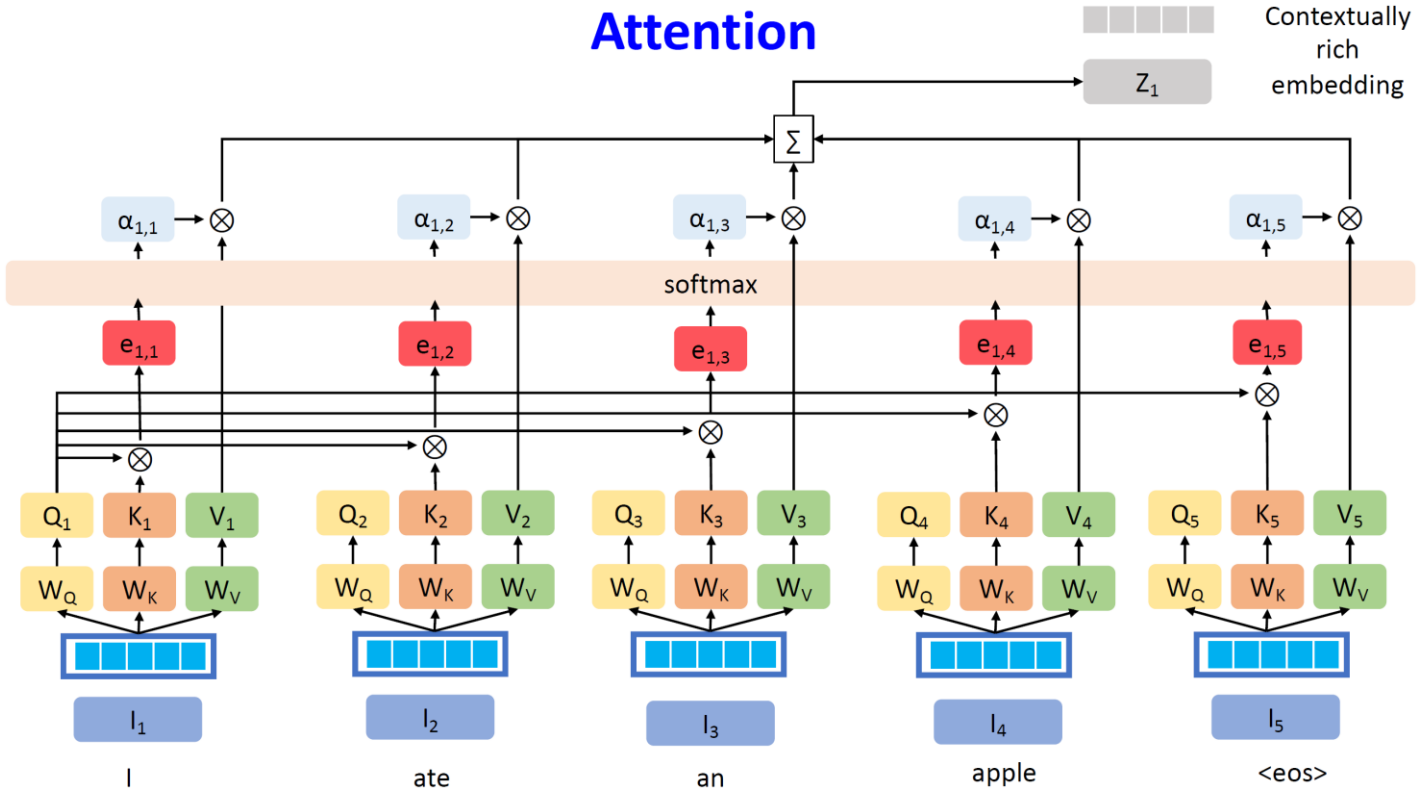
- Contextually Rich Embeddings (Nx=6, cascaded)

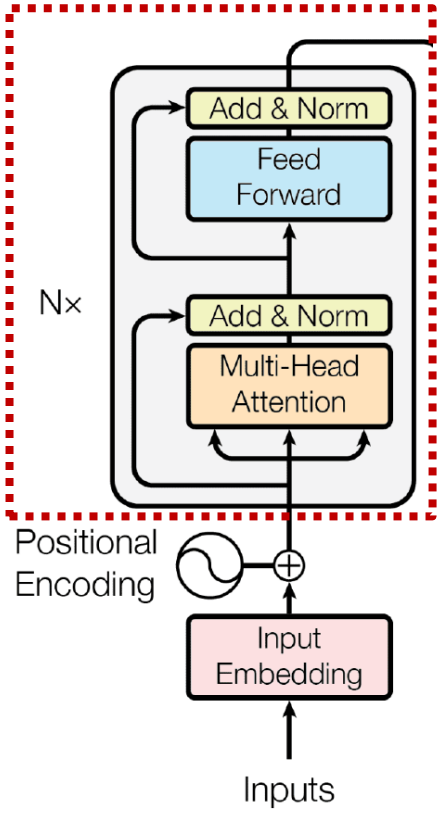




Encoder

- Attention Layers has no recurrence and thus it is parallel

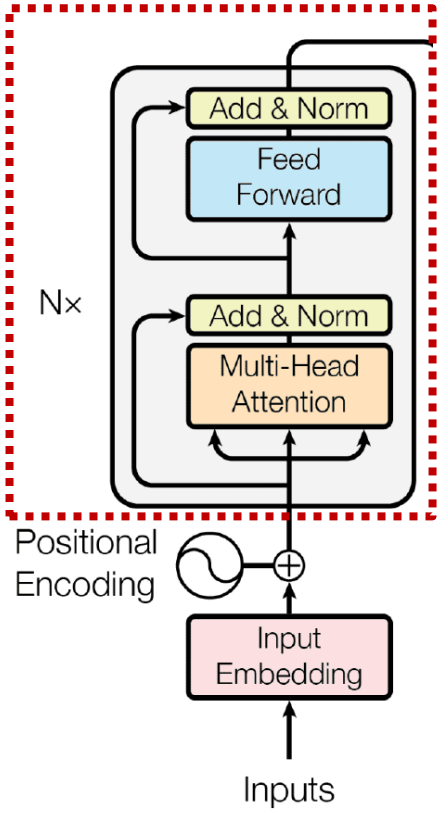




Encoder - Self Attention

- Attention Layers has no recurrence and thus it is parallel.
- In Self-Attention Query=Key=Value
  - $\boldsymbol{k}_i = \boldsymbol{W}_k h_i,\ \boldsymbol{v}_i = \boldsymbol{W}_v h_i,\ \ \boldsymbol{q}_i = \boldsymbol{W}_q h_i,\ \ \boldsymbol{e}_{ij} = \boldsymbol{q}_i^T \boldsymbol{k}_j$
  - $w_{ij} = softmax(\boldsymbol{q}_i^T \boldsymbol{k}_j)$
  - $o_i = \sum_j w_{ij}\, \boldsymbol{v}_j$
- Put all in matrix form

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_{att}}}\right)V$$



- $\sqrt{d_{att}}$ is normalization factor:
- length of vector $\mathbf{1} \in \mathbb{R}^{d_{att}}$ is $\sqrt{d_{att}}$

Encoder - Self Attention

- Algorithm:

Algorithm 3: Basic single-query attention.

Input: $\boldsymbol{e} \in \mathbb{R}^{d_{\text{in}}}$, vector representation of the current token

Input: $\boldsymbol{e}_t \in \mathbb{R}^{d_{\text{in}}}$, vector representations of context tokens $t \in [T]$.

Output: $\tilde{\boldsymbol{v}} \in \mathbb{R}^{d_{\text{out}}}$, vector representation of the token and context combined.

Parameters: $\boldsymbol{W}_q, \boldsymbol{W}_k \in \mathbb{R}^{d_{\text{attn}} \times d_{\text{in}}}$, $\boldsymbol{b}_q, \boldsymbol{b}_k \in \mathbb{R}^{d_{\text{attn}}}$, the query and key linear projections.

Parameters: $\boldsymbol{W}_v \in \mathbb{R}^{d_{\text{out}} \times d_{\text{in}}}$, $\boldsymbol{b}_v \in \mathbb{R}^{d_{\text{out}}}$, the value linear projection.

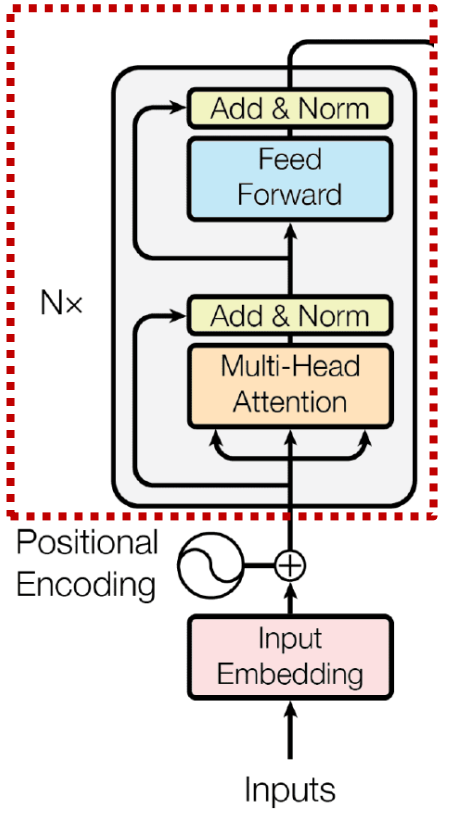
$\boldsymbol{q} \leftarrow \boldsymbol{W}_q \boldsymbol{e} + \boldsymbol{b}_q$

$\forall t: \ \boldsymbol{k}_t \leftarrow \boldsymbol{W}_k \boldsymbol{e}_t + \boldsymbol{b}_k$

$\forall t: \ \boldsymbol{v}_t \leftarrow \boldsymbol{W}_v \boldsymbol{e}_t + \boldsymbol{b}_v$

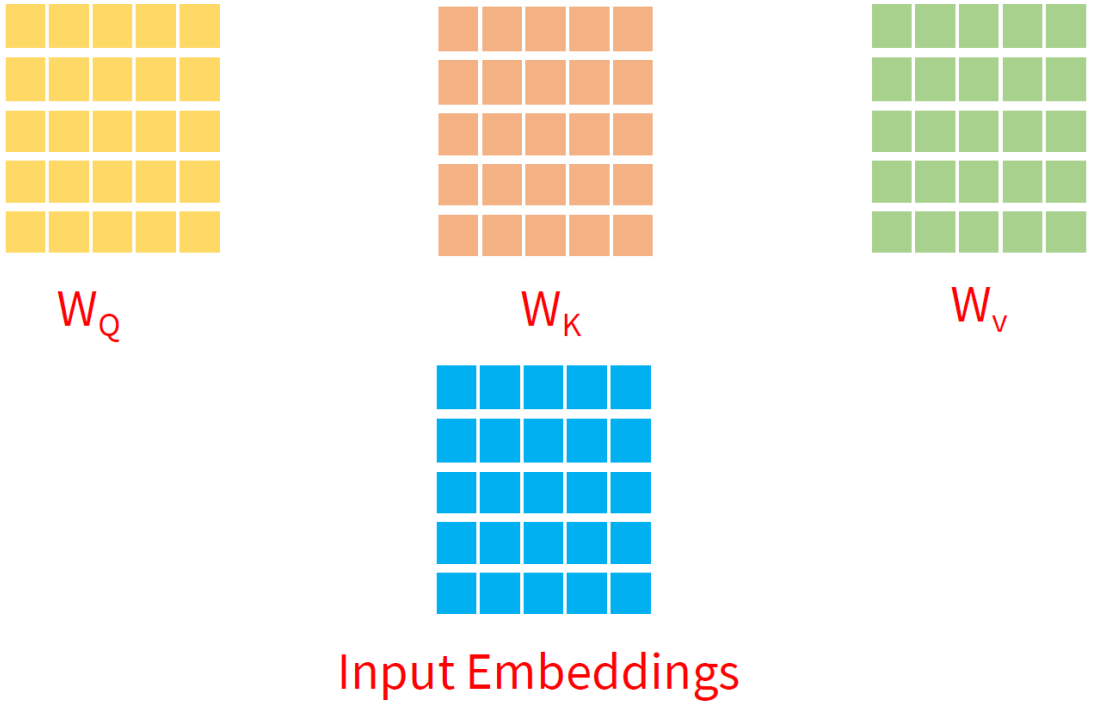
$\forall t: \ \alpha_t = \frac{\exp\left(\boldsymbol{q}^\top \boldsymbol{k}_t / \sqrt{d_{\text{attn}}}\right)}{\sum_u \exp\left(\boldsymbol{q}^\top \boldsymbol{k}_u / \sqrt{d_{\text{attn}}}\right)}$

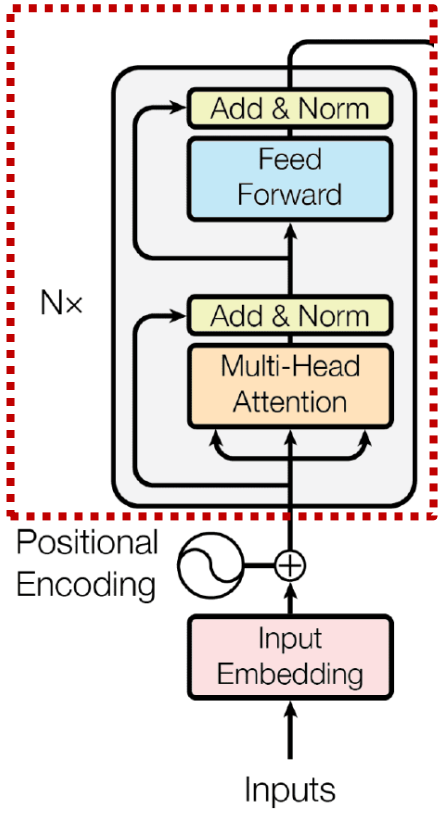
return $\tilde{\boldsymbol{v}} = \sum_{t=1}^{T} \alpha_t \boldsymbol{v}_t$



Encoder - Self Attention

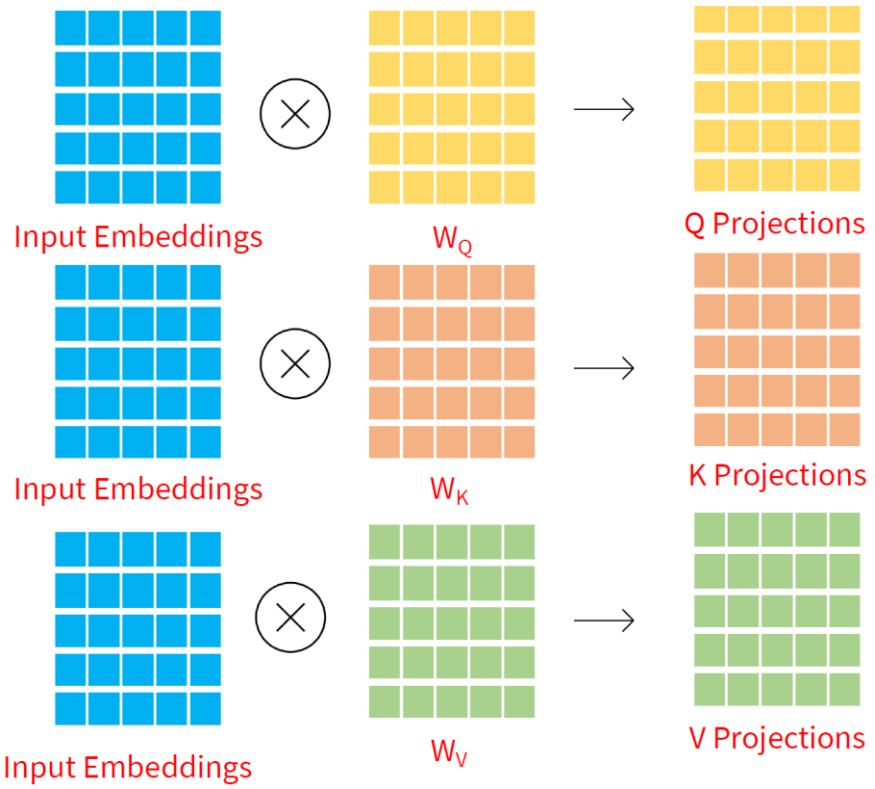
- Matrix Representation:

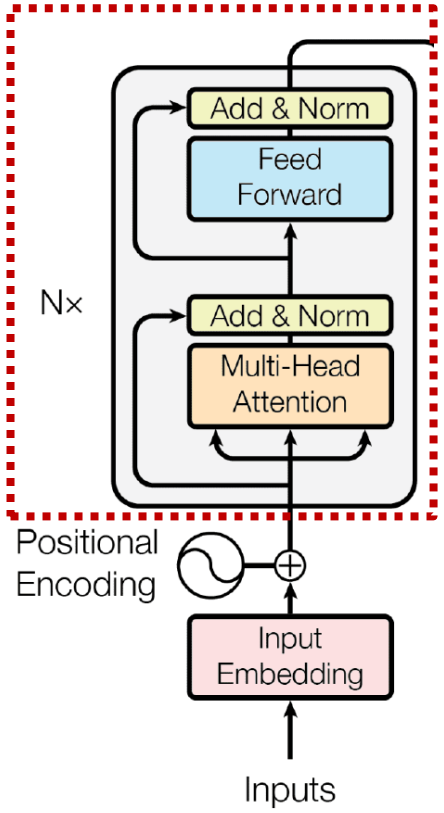




Encoder - Self Attention

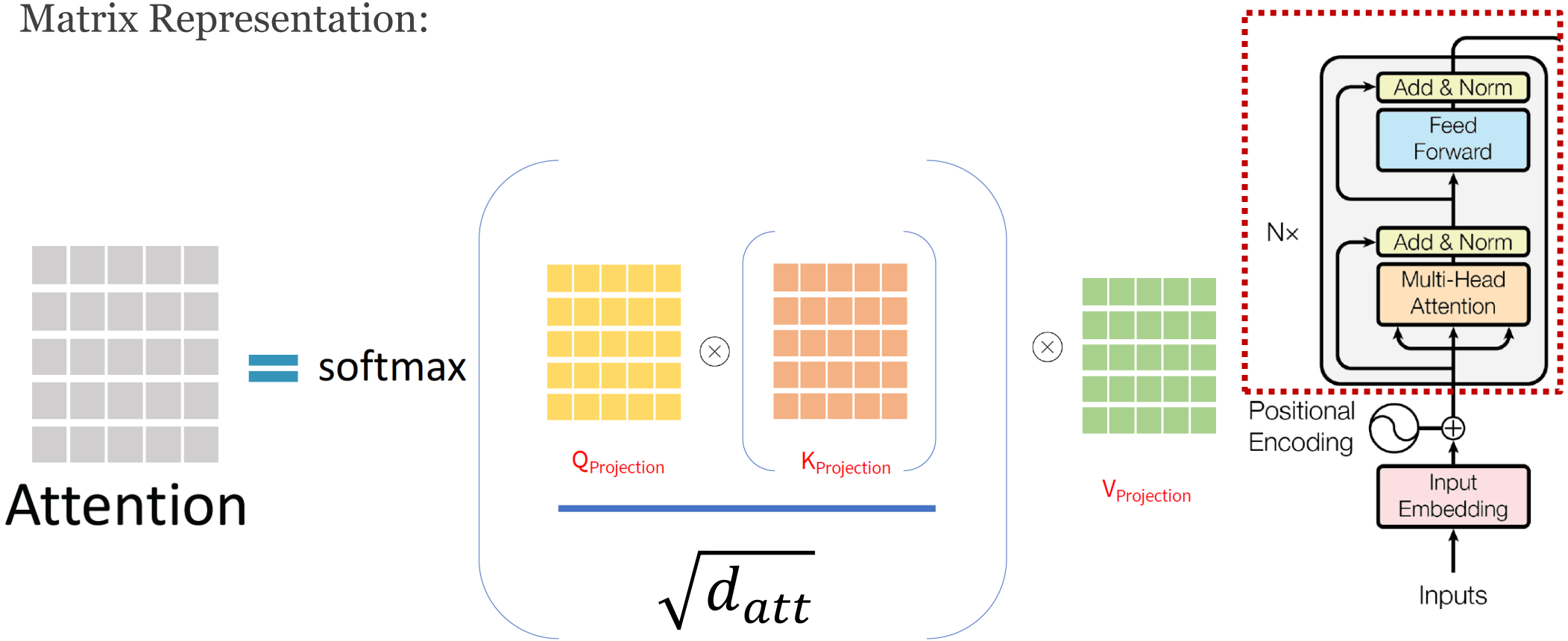
- Matrix Representation:





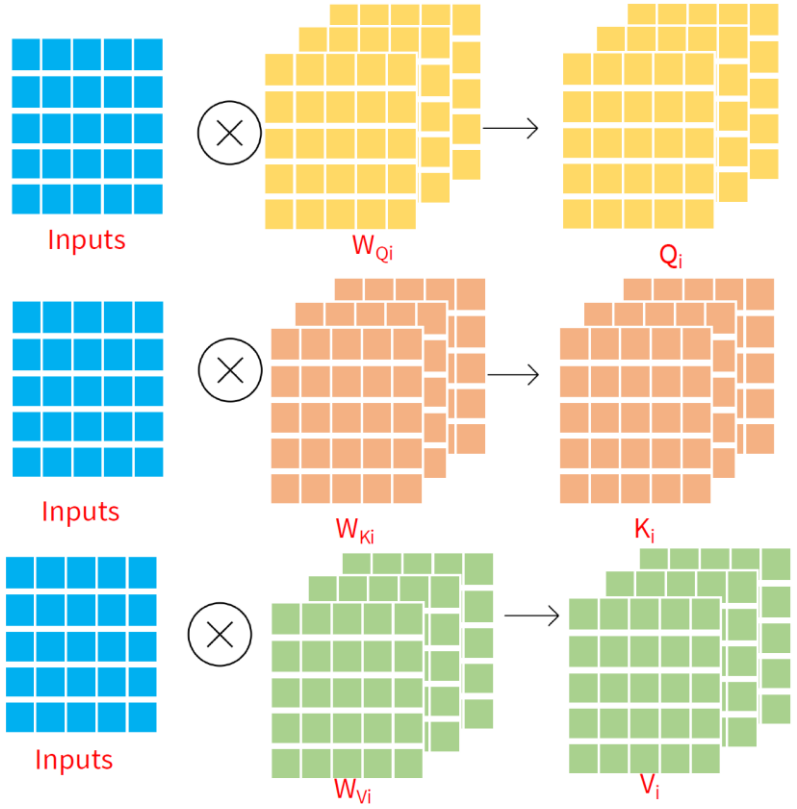
Encoder – Multi-Head Self Attention

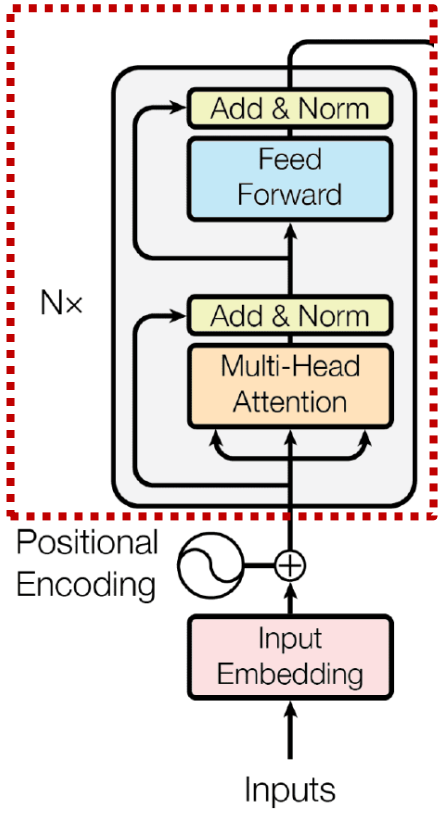
- Matrix Representation:



Encoder - Multi-Head Attention

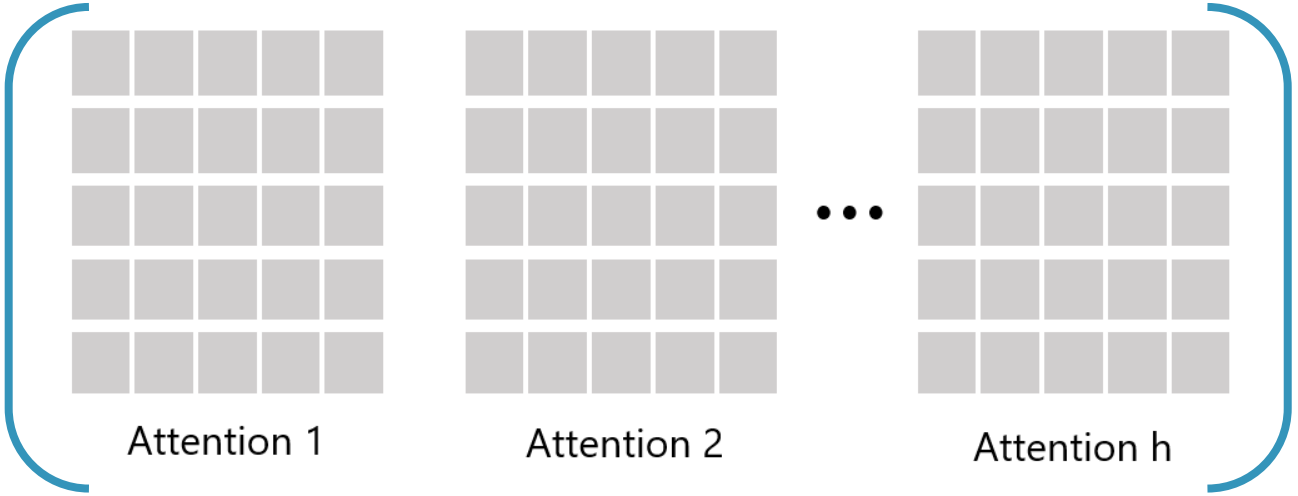
- Matrix Representation:

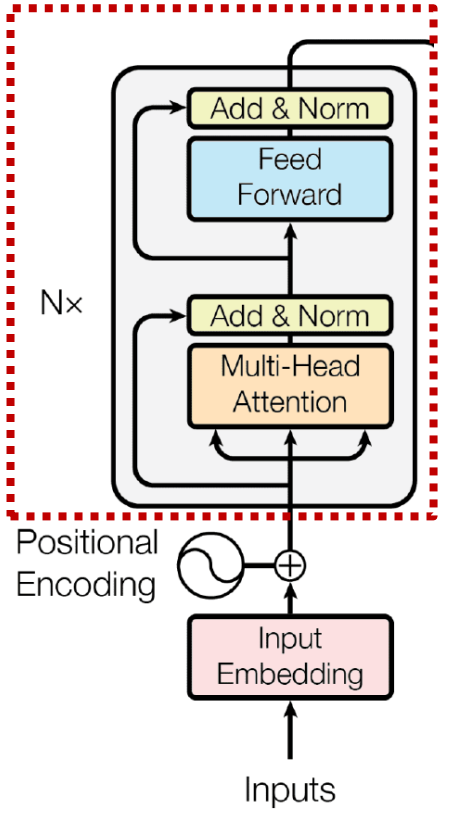




Encoder - Multi-Head Attention

- Matrix Representation:



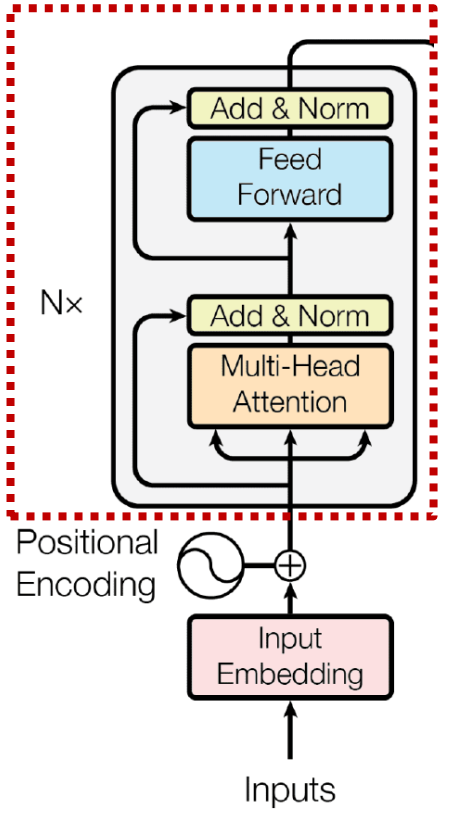


Encoder - Multi-Head Attention

- Residual connection:
  - Input
  - Layer Normalization (fixed or learned, mean and std)
- Feedforward:
  - Two FC layers with Relu/GELU activation functions

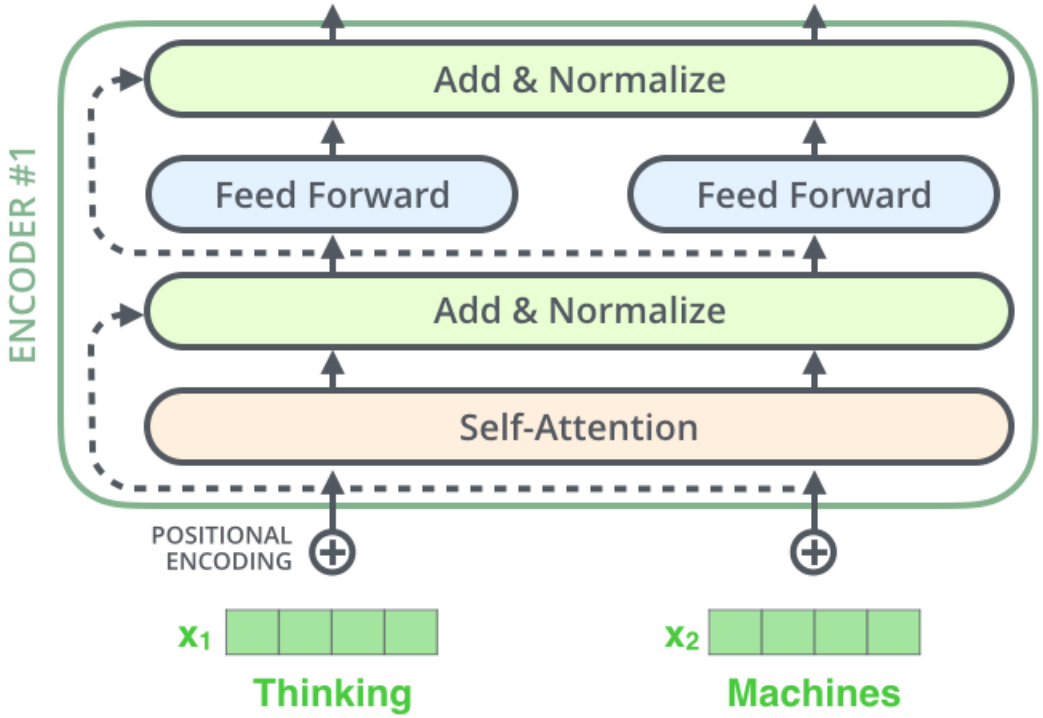
$$FFN(x) = ReLu(W_1 x + b_1)W_2 + b_2$$

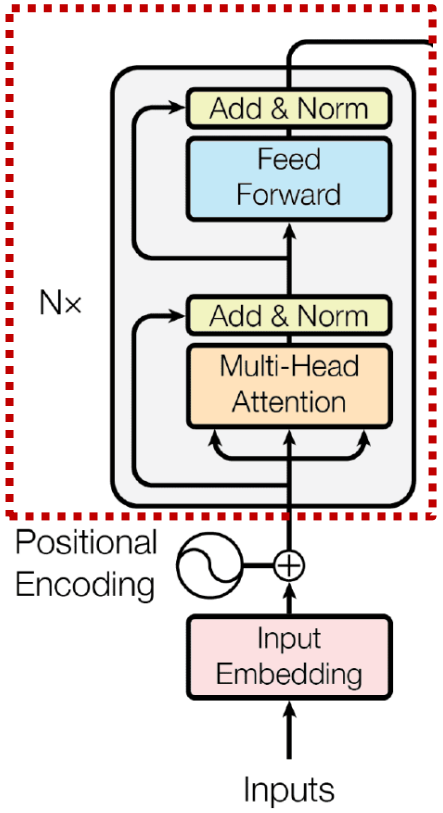
- While the linear transformations are the same across different positions, they use different parameters from layer to layer.



Encoder - Multi-Head Attention

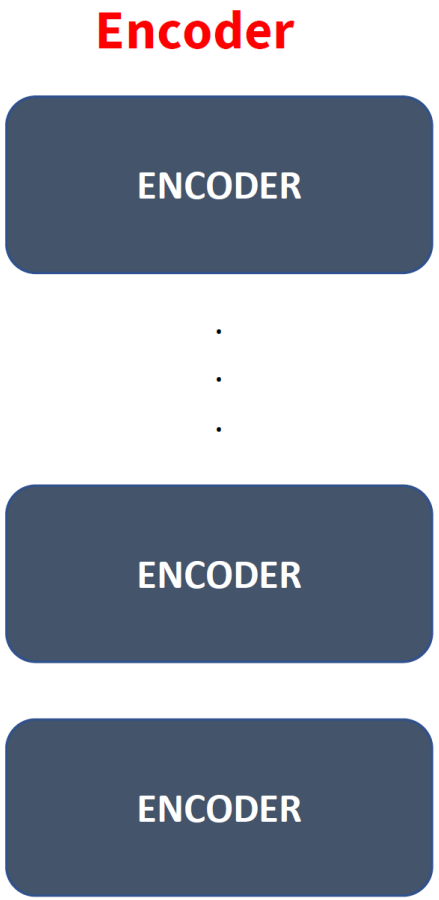
- Feedforward:

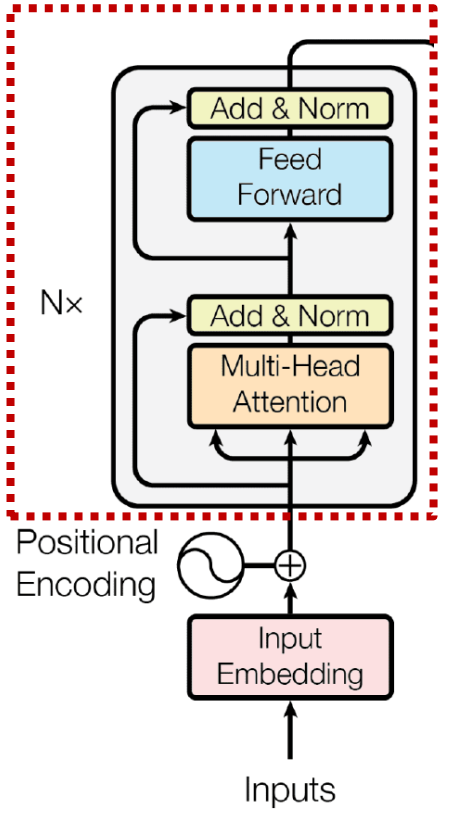




Encoder - Cascading

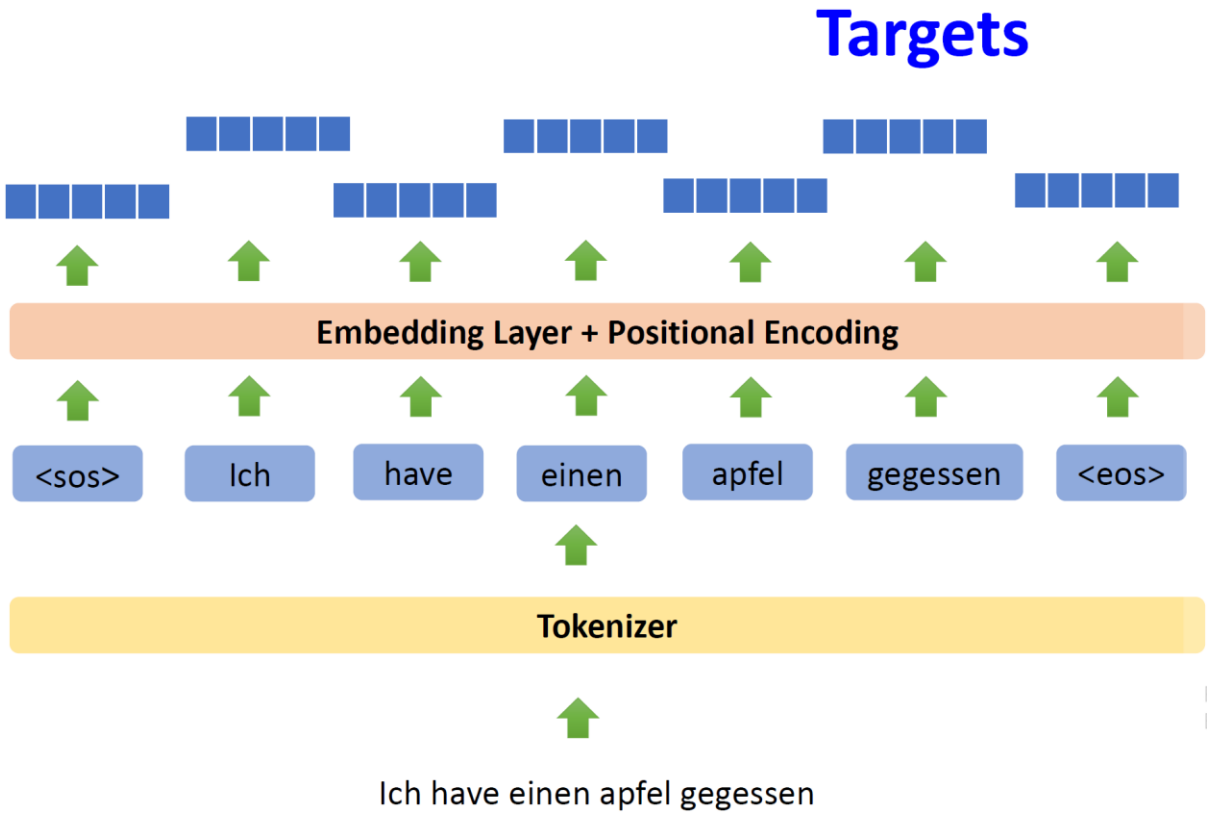
- There are Nx (=6) encoder in cascade

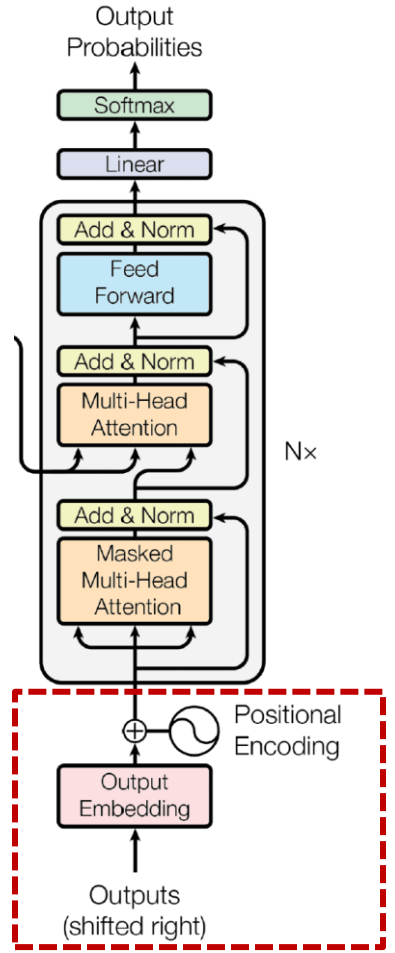




Decoder – Target Embedding

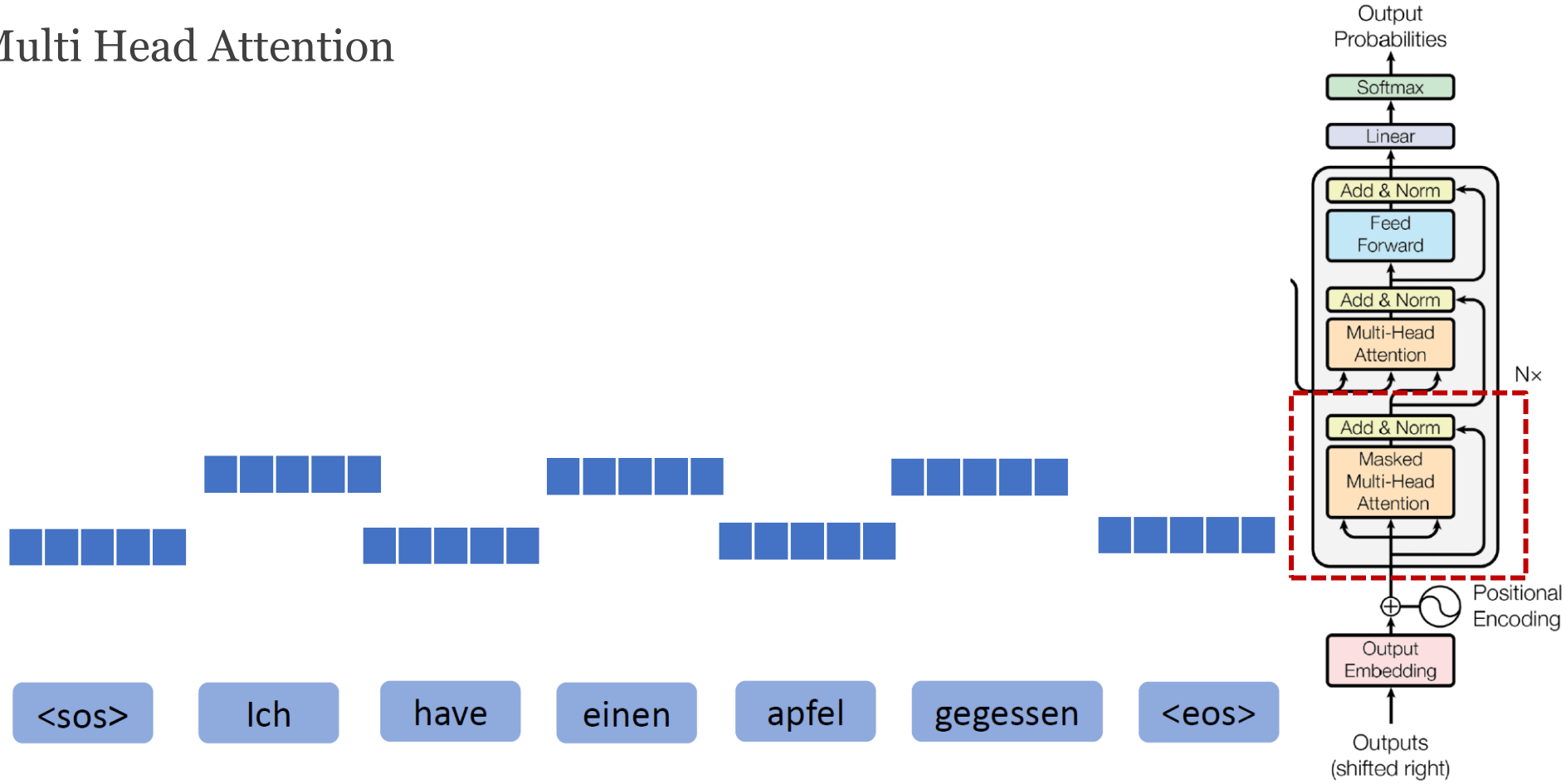
- Generate Target Embeddings





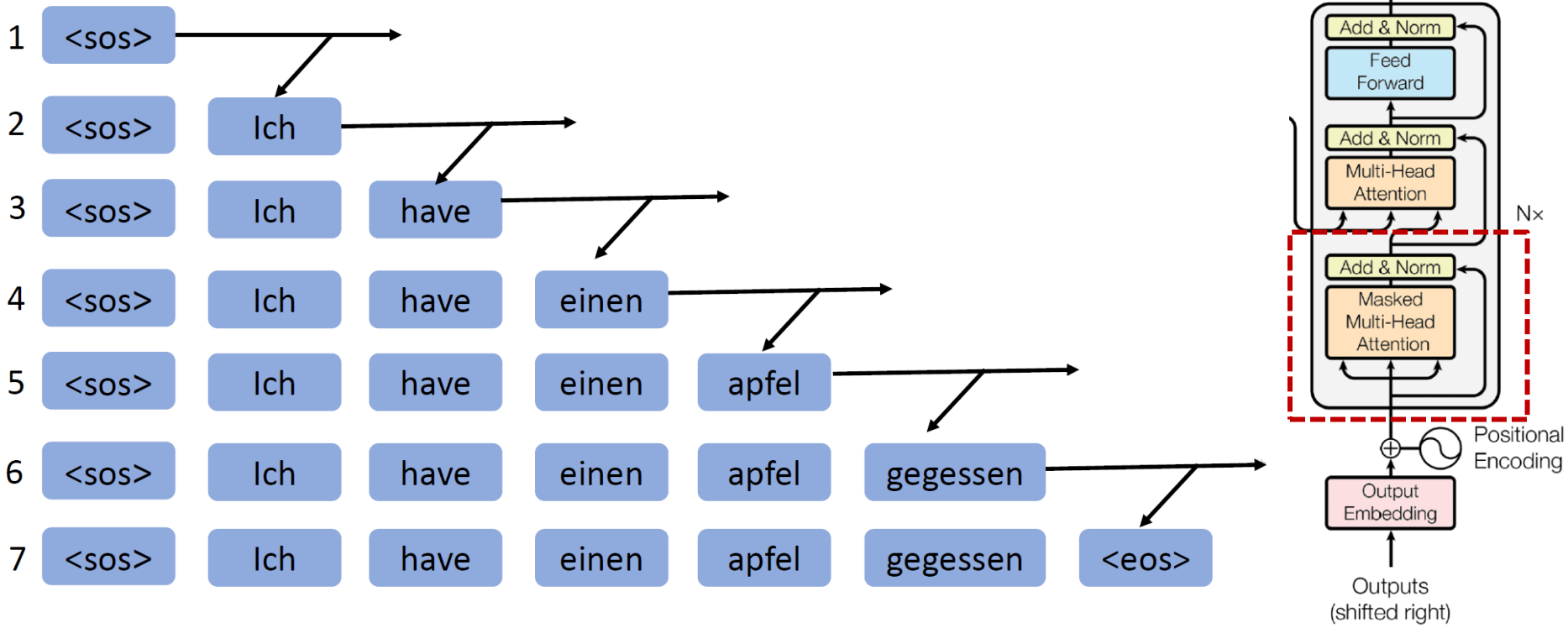
Decoder – Masked Multi Head Attention

- Masked Multi Head Attention



Decoder – Masked Multi Head Attention

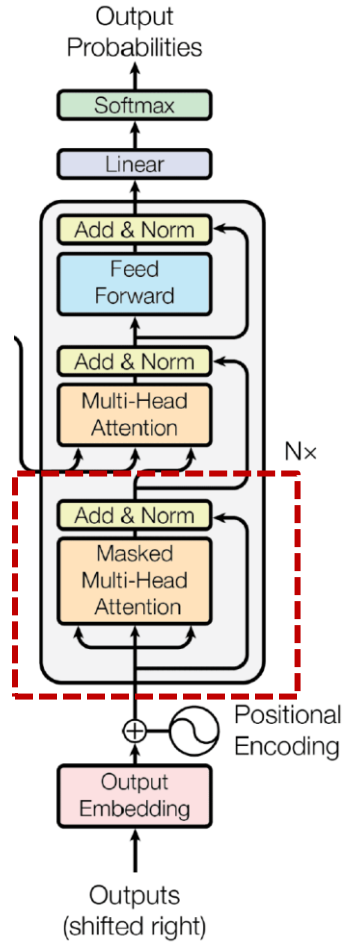
- Decoding step by step (Inference)



Decoder – Masked Multi Head Attention

- Outputs at time T should only pay attention to outputs until time $T-1$

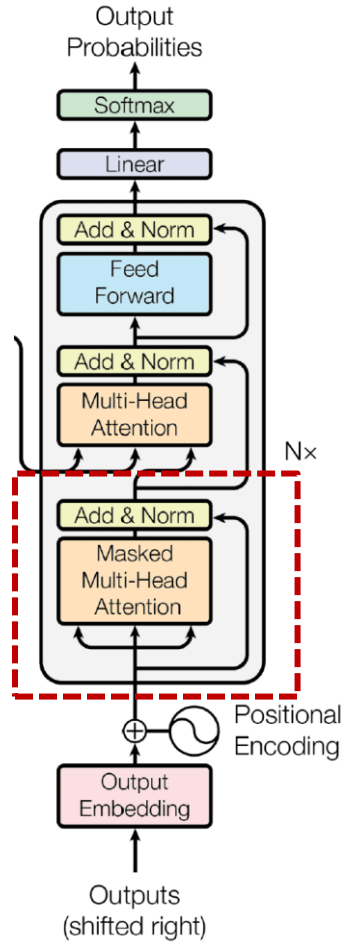
1	<sos>	Ich	have	einen	apfel	gegessen	<eos>
2	<sos>	Ich	have	einen	apfel	gegessen	<eos>
3	<sos>	Ich	have	einen	apfel	gegessen	<eos>
4	<sos>	Ich	have	einen	apfel	gegessen	<eos>
5	<sos>	Ich	have	einen	apfel	gegessen	<eos>
6	<sos>	Ich	have	einen	apfel	gegessen	<eos>
7	<sos>	Ich	have	einen	apfel	gegessen	<eos>



Decoder – Masked Multi Head Attention

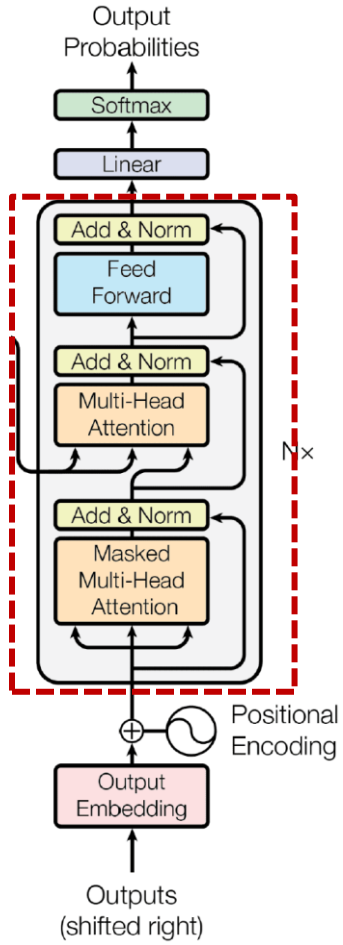
- Outputs at time T should only pay attention to outputs until time $T-1$
- $Softmax(-\infty) = 0$

1	<sos>	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
2	<sos>	Ich	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$
3	<sos>	Ich	have	$-\infty$	$-\infty$	$-\infty$	$-\infty$
4	<sos>	Ich	have	einen	$-\infty$	$-\infty$	$-\infty$
5	<sos>	Ich	have	einen	apfel	$-\infty$	$-\infty$
6	<sos>	Ich	have	einen	apfel	gegessen	$-\infty$
7	<sos>	Ich	have	einen	apfel	gegessen	<eos>



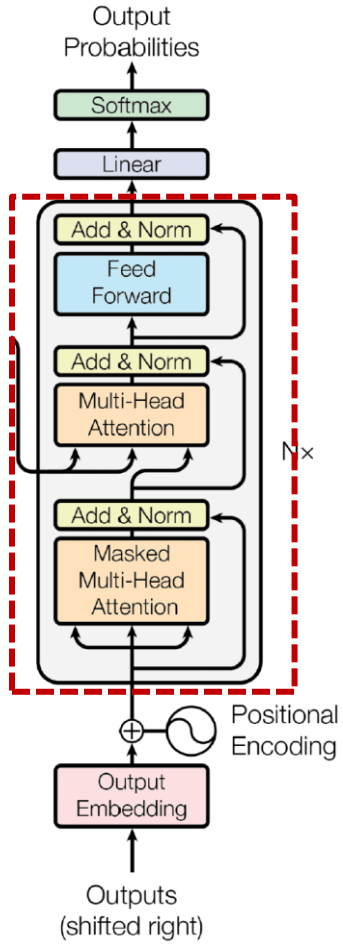
Decoder – Encoder and Decoder Attention

- Encoder Self Attention:
  - Queries from Encoder Inputs
  - Keys from Encoder Inputs
  - Values from Encoder Inputs
- Decoder Masked Self Attention:
  - Queries from Decoder Inputs
  - Keys from Decoder Inputs
  - Values from Decoder Inputs



Decoder – Encoder and Decoder Attention

- Encoder-Decoder Self Attention:
  - Keys from Encoder Outputs (Final output)
  - Values from Encoder Outputs (Final output)
  - Queries from Decoder Inputs (Final output)

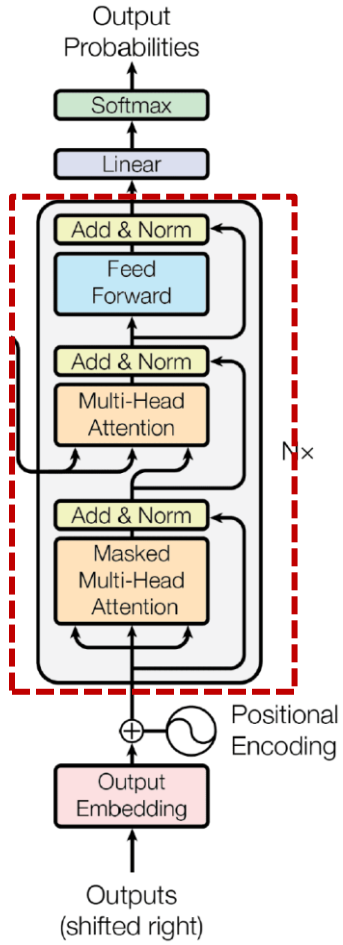


Decoder - Multi-Head Attention

- Residual connection:
  - Input
  - Layer Normalization (fixed or learned, mean and std)
- Feedforward:
  - Two FC layers with Relu/GELU activation functions

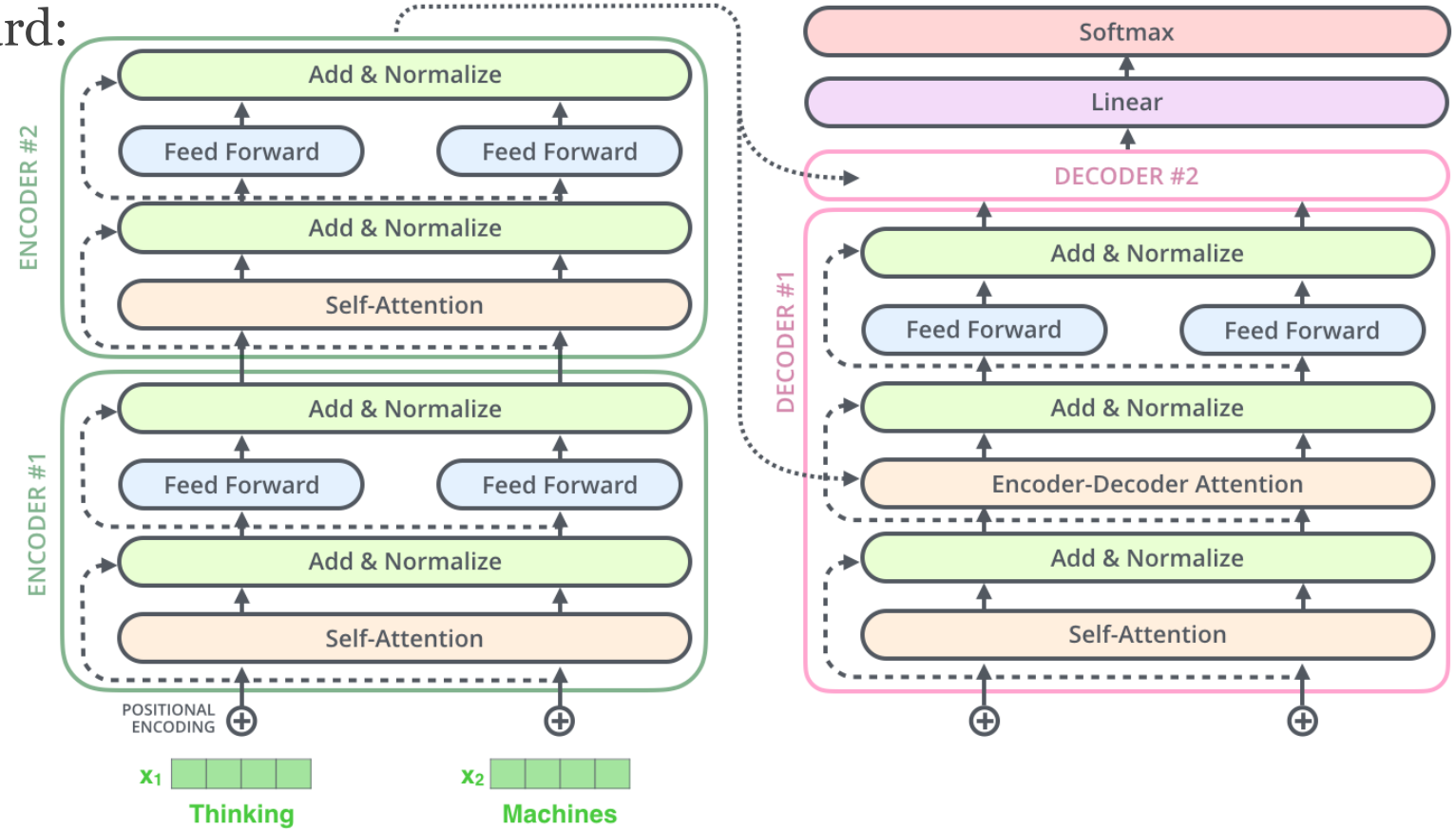
$$FFN(x) = ReLu(W_1 x + b_1)W_2 + b_2$$

- While the linear transformations are the same across different positions, they use different parameters from layer to layer.



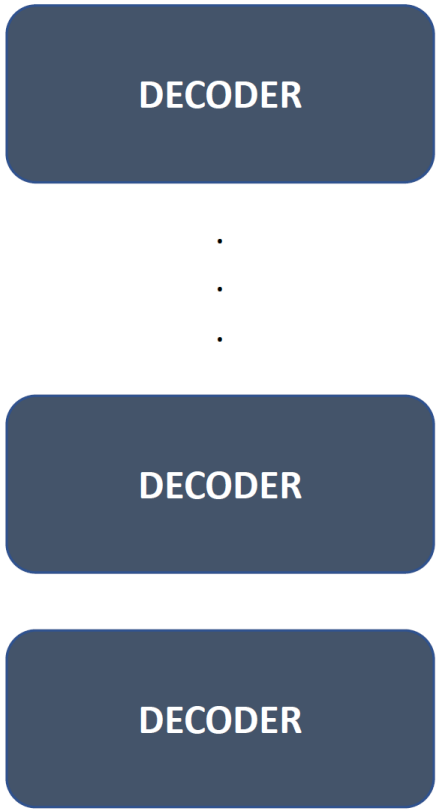
Decoder - Multi-Head Attention

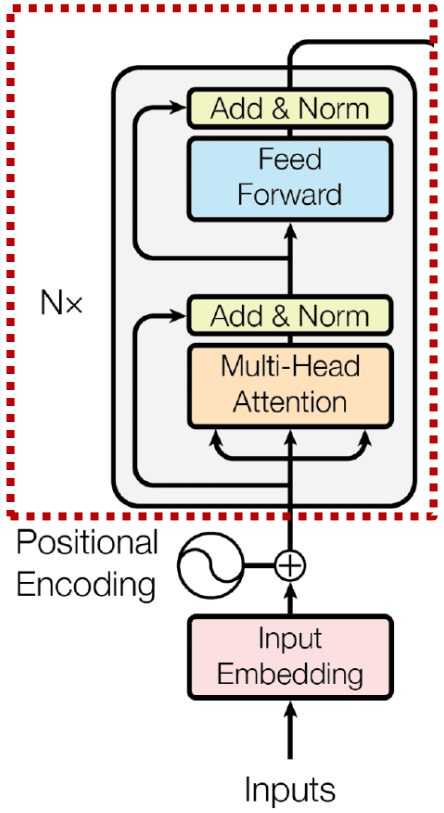
- Feedforward:



Decoder - Cascading

- There are Nx (=6) decoder in cascade





Transformer Illustration

- The Illustrated Transformer, Jay Alammar
- Transformers from scratch, Peter Bloem

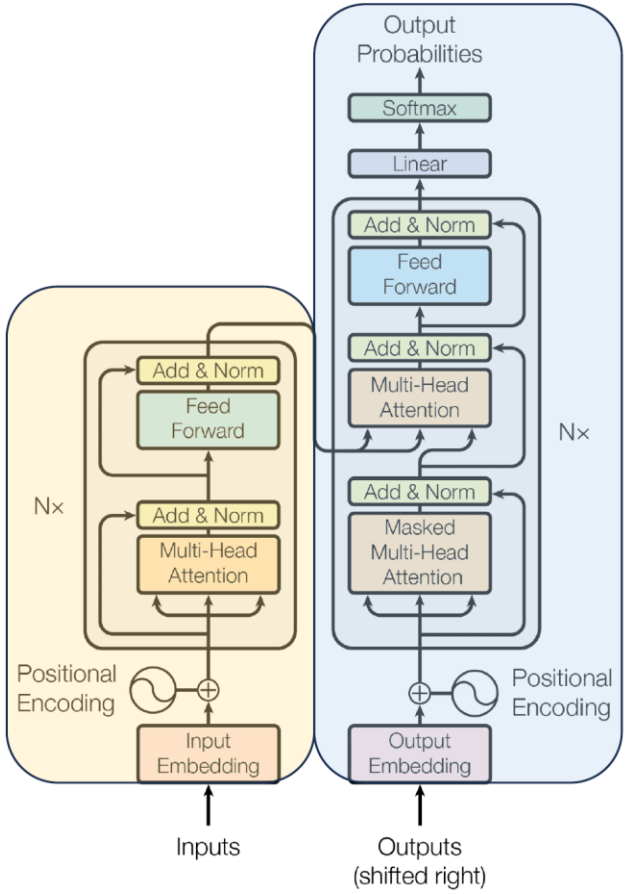
Transformer and LLM

Input – input tokens
Output – hidden states

Model can see all timesteps

Does not usually output tokens, so no inherent auto-regressivity

Representation



Input – output tokens and hidden states*
Output – output tokens

Model can only see previous timesteps

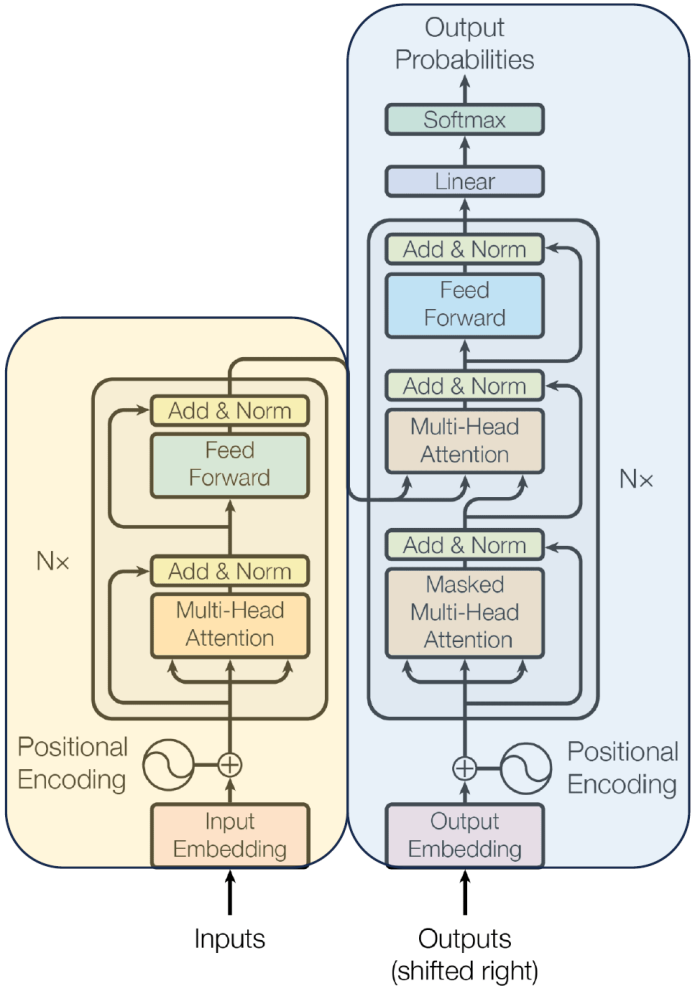
Model is auto-regressive with previous timesteps' outputs

Generation

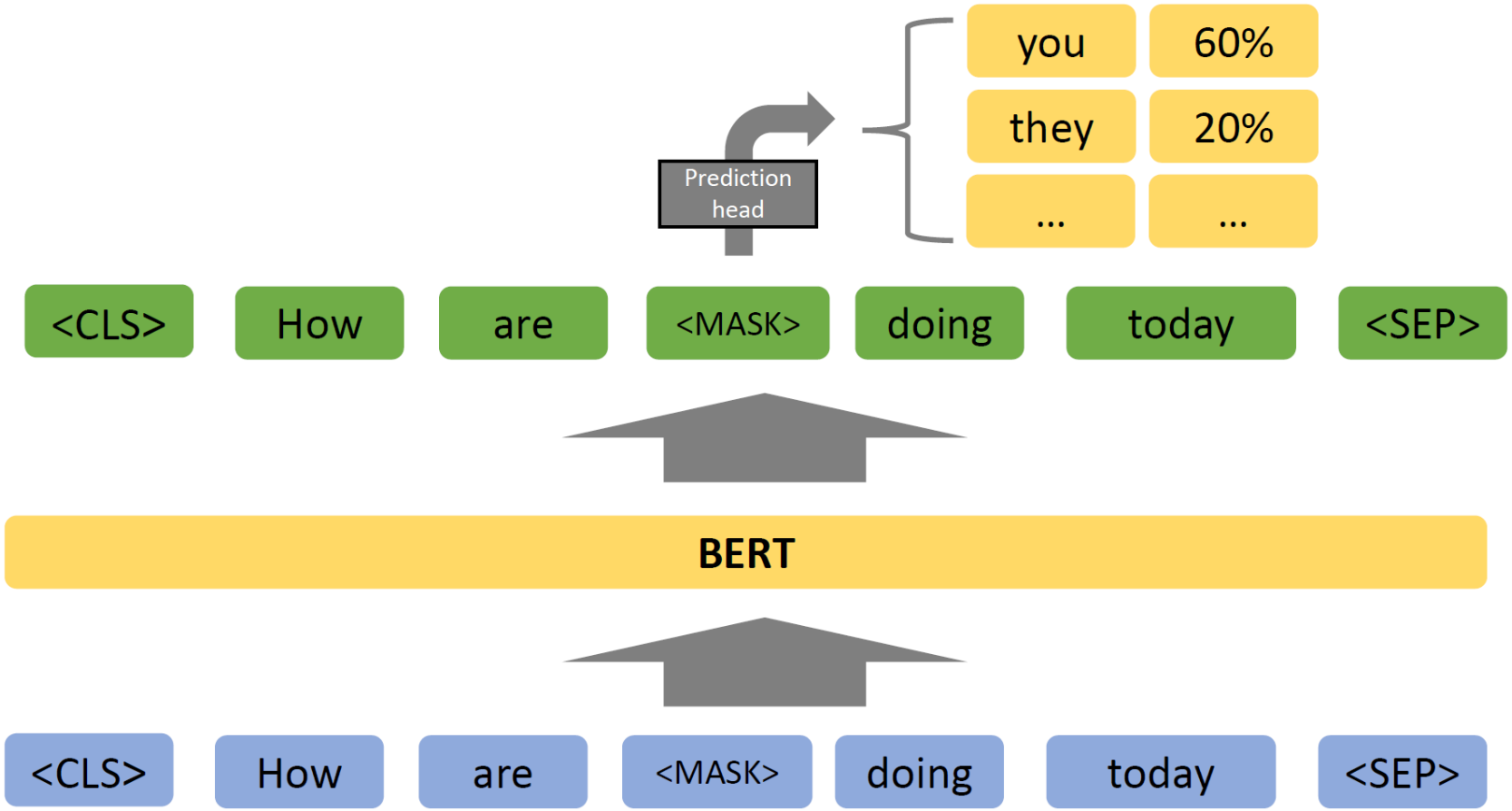
BERT (Bidirectional Encoder Representations)

- **BERT Pre-Training Corpus:**
  - English Wikipedia - 2,500 million words
  - Book Corpus - 800 million words
- **BERT Pre-Training Tasks:**
  - MLM* (Masked Language Modeling)
  - NSP (Next Sentence Prediction)
- **BERT Pre-Training Results:**
  - BERT-Base – 110M Params
  - BERT-Large – 340M Params

*MLMs are trained to predict masked words or tokens in a given input sequence, given the context provided by the surrounding words.

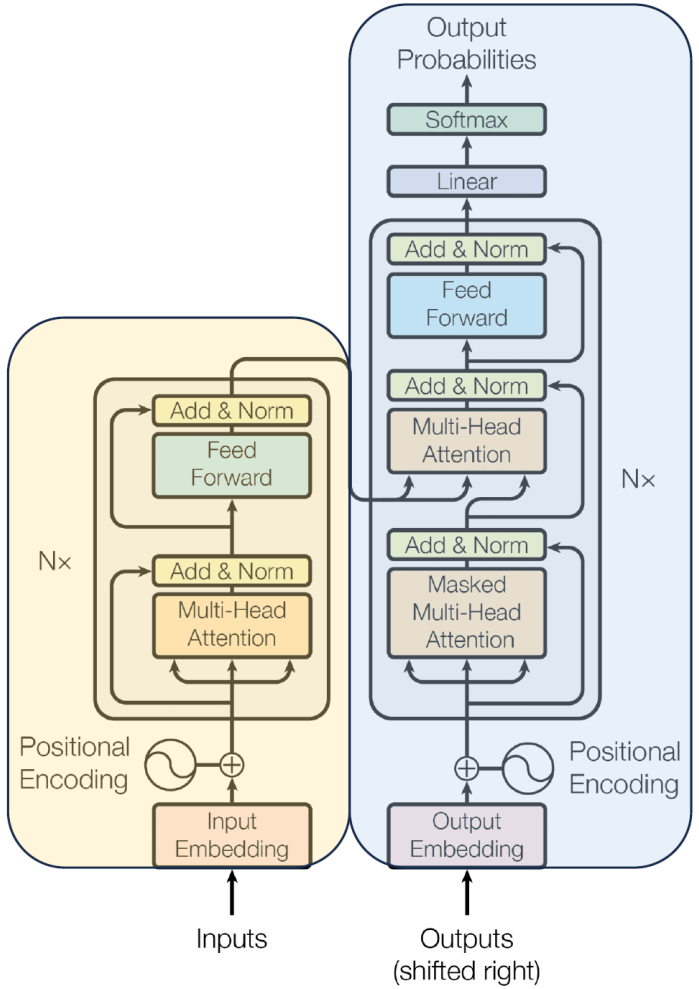


BERT - MLM (Masked Language Modeling)



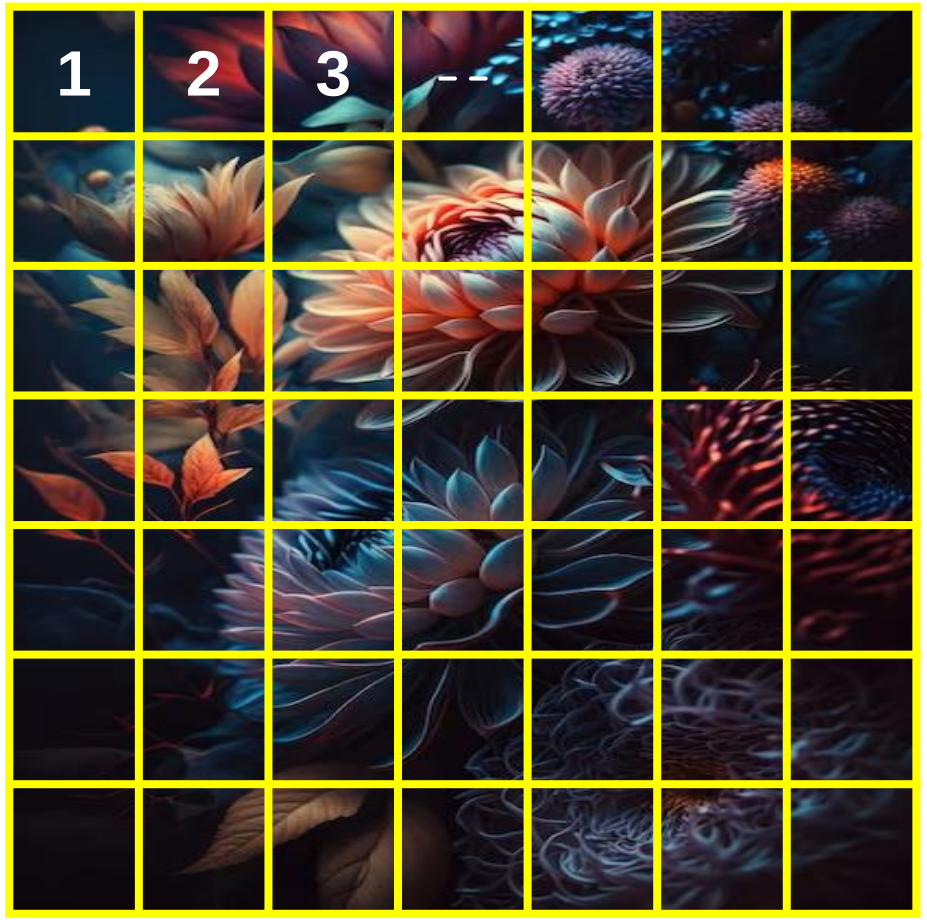
GPT (Generative Pretrained Transformer)

- **Different Design**
- **GPT Pre-Training Corpus:**
  - Similarly, BooksCorpus and English Wikipedia
- **GPT Pre-Training Tasks:**
  - Predict the next token, given the previous tokens
  - More learning signals than MLM
- **GPT Pre-Training Results:**
  - GPT – 117M Params



Vision Transform (ViT)

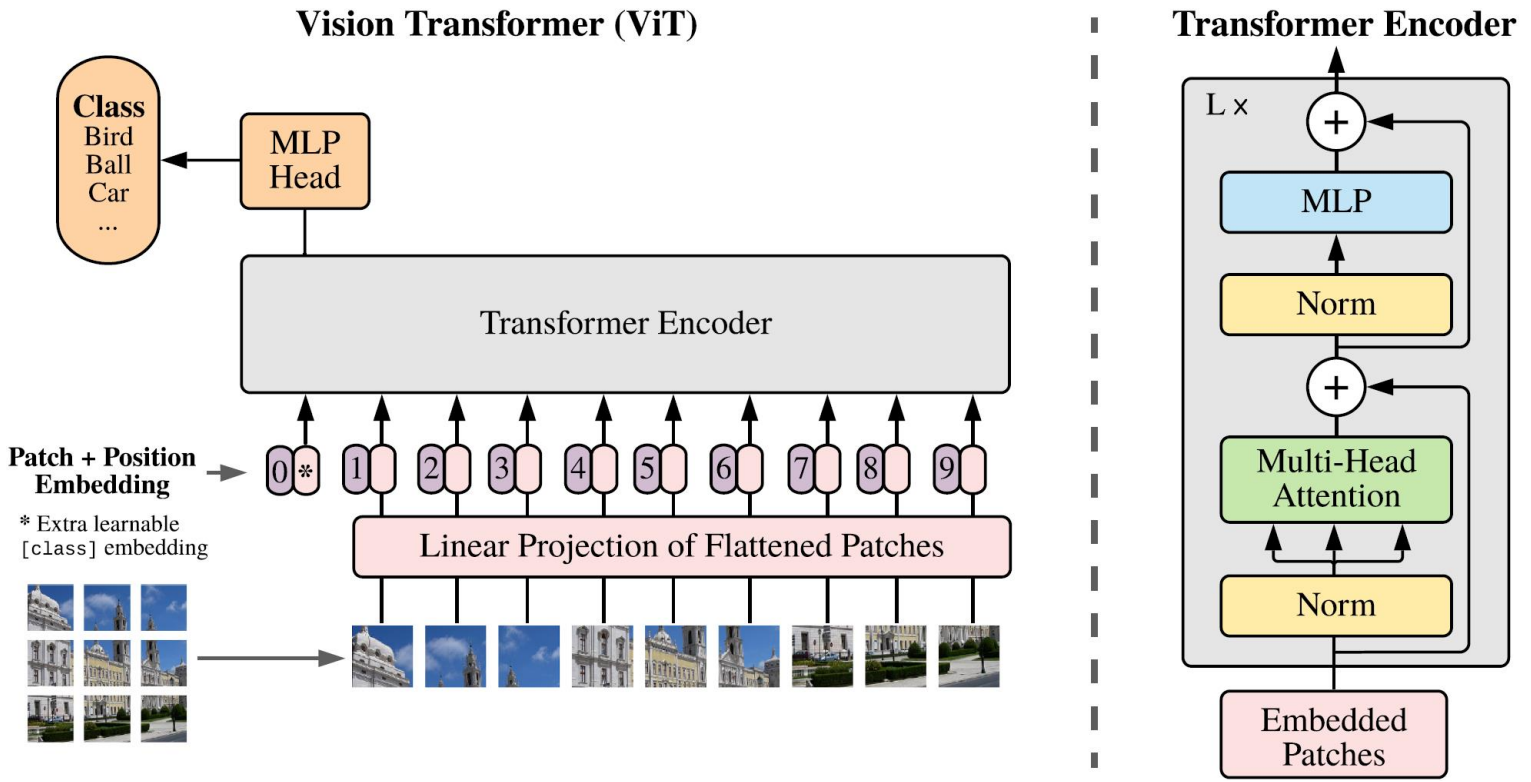
- Image and Sequence?
- Token~ Image Patches
- Scan the patch and build sequence
  - Note no need to recurrence (Self Attention)



ViT Steps

- ViT steps:
  - Split an image into patches ($\mathbb{R}^{D\times D}$)
  - Flatten the patches ($\mathbb{R}^{D^2}$)
  - Produce a linear embeddings from the flattened patches (learnable), known as patch embedding.
  - Add *positional* embeddings
  - Feed the sequence as an input to a *standard* transformer encoder
  - Pretrain the model with image labels (fully supervised on a huge dataset)
  - Finetune on the downstream dataset for image classification

ViT Configuration



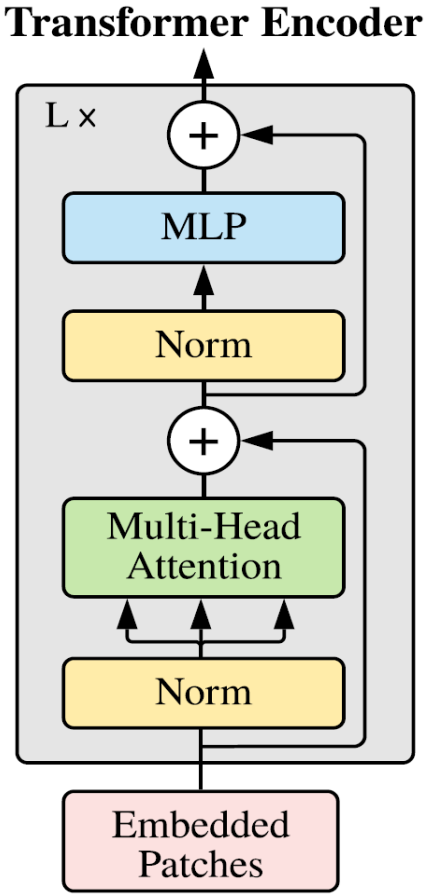
ViT Inference - Animation

ViT Notation

- ViT Notations:
  - Image: $x \in \mathbb{R}^{H \times W \times C}$
  - Patch resolution: $P \times P$, number of patch $N = \frac{HW}{P^2}$, N is input sequence length
  - Sequence of flattened patch: $x_p \in \mathbb{R}^{N \times (P^2 C)}$
  - Latent vector size: D, input embedding
  - Final MLP Head: fed by Class Token output.

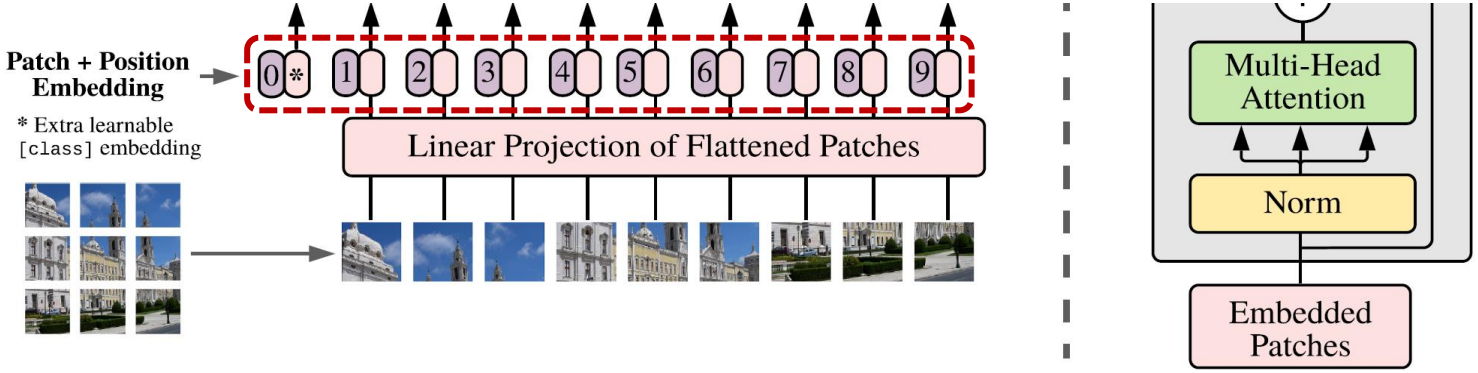
ViT Mathematics

- $Z_0 = [X_{Class}; X_p^1 E; X_p^2 E; \dots; X_p^N E] + E_{pos}, E \in \mathbb{R}^{(P^2 C) \times D}, E_{pos} \in \mathbb{R}^{(N+1) \times D}$
- $Z'_l = MSA(LN(Z_{l-1})) + Z_{l-1}, l = 1, \dots, L$
- $Z_l = MLP(LN(Z'_l)) + Z'_l, l = 1, \dots, L$
- $y = LN(Z_L^0)$
- $Xclass$: Token class, a learnable embedding (concatenating to patch embedding)
- E_{pos}: Learnable position embedding
- L: Number of blocks
- MLP: Two Layer with GELU activation function
- MSA: Multi-head Self Attention
- LN: Layer Normalization



ViT Three Embedding

- (Patch+Class)/Position Embedding:



- $0 - 1 - 2 - \cdots - N$: $(N + 1)$ Position Encoding (0 : class token position embedding)
- $* - E_1 - E_2 - \cdots - E_N$: $(N + 1)$ Patch Embedding ($*$: learnable class token)
- The output of [class] token is transformed into a class prediction via a small MLP with tanh as non-linearity in the single hidden layer.

ViT Original Configuration

- From original paper: An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale.
  - Patch size: 16×16
  - D: (patch embedding size):

Model	Layers	Hidden size D	MLP size	Heads	Params
ViT-Base	12	768	3072	12	86M
ViT-Large	24	1024	4096	16	307M
ViT-Huge	32	1280	5120	16	632M

ViT Original Comparison

- Image classification benchmark:
  - Dataset: JFT-300M: Google dataset, One billion labels for the 300M images.

	Ours-JFT (ViT-H/14)	Ours-JFT (ViT-L/16)	Ours-I21k (ViT-L/16)	BiT-L (ResNet152x4)	Noisy Student (EfficientNet-L2)
ImageNet	**88.55** ± 0.04	87.76 ± 0.03	85.30 ± 0.02	87.54 ± 0.02	88.4/88.5*
ImageNet ReaL	**90.72** ± 0.05	90.54 ± 0.03	88.62 ± 0.05	90.54	90.55
CIFAR-10	**99.50** ± 0.06	99.42 ± 0.03	99.15 ± 0.03	99.37 ± 0.06	–
CIFAR-100	**94.55** ± 0.04	93.90 ± 0.05	93.25 ± 0.05	93.51 ± 0.08	–
Oxford-IIIT Pets	**97.56** ± 0.03	97.32 ± 0.11	94.67 ± 0.15	96.62 ± 0.23	–
Oxford Flowers-102	99.68 ± 0.02	**99.74** ± 0.00	99.61 ± 0.02	99.63 ± 0.03	–
VTAB (19 tasks)	**77.63** ± 0.23	76.28 ± 0.46	72.72 ± 0.21	76.29 ± 1.70	–

Application

- Image Classification
- Image Captioning
- Image Segmentation
- Anomaly detection
- Action Recognition
- Autonomous Driving

Any Question