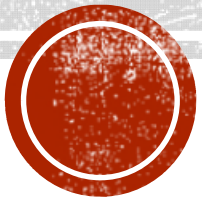
IE4424 Machine Learning Design and Application

Week 5: Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM) & Transformer



Dr Yap Kim Hui

Email: ekhyap@ntu.edu.sg

References

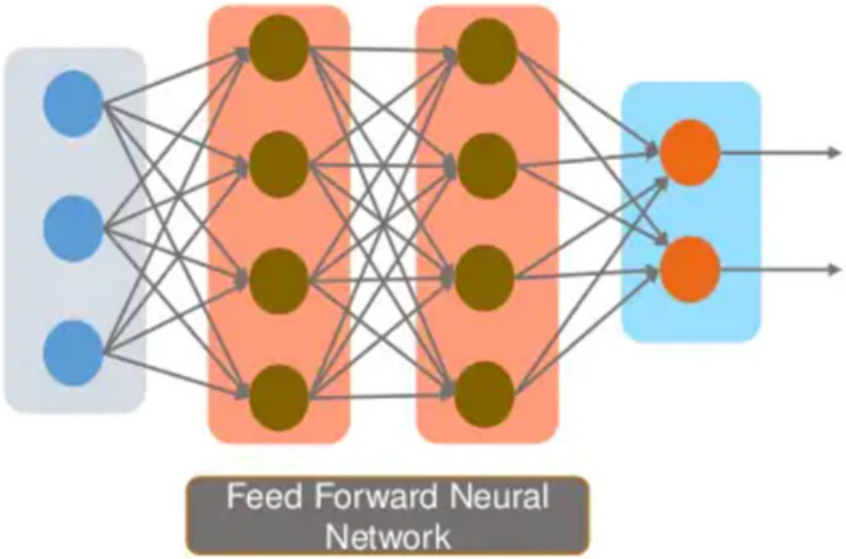
- Stanford Lecture Notes, CS231n: Deep Learning for Computer Vision.
- Ian Goodfellow, Yoshua Bengio and Aaron Courville, Deep Learning, MIT Press, http://www.deeplearningbook.org
- Pytorch Tutorial. https://pytorch.org/tutorials/
- University of Wisconsin Madison Lecture Notes, CS638. University of Michigan, EECS 498-007 / 598-005: Deep Learning for Computer Vision
- Simplilearn, Recurrent Neural Network Tutorial
- Shusen Wang, Transformer Model, Youtube Online Videos

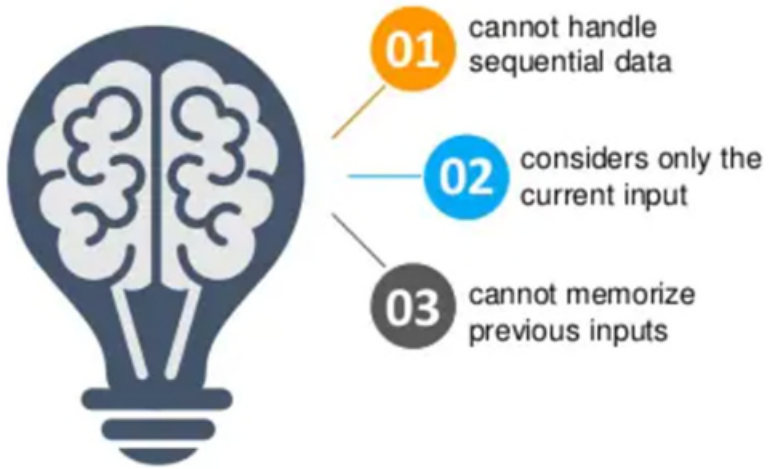
Overview

- Introduction
- Recurrent Neural Network (RNN)
- Long Short-Term Memory (LSTM)
- Transformer

Introduction

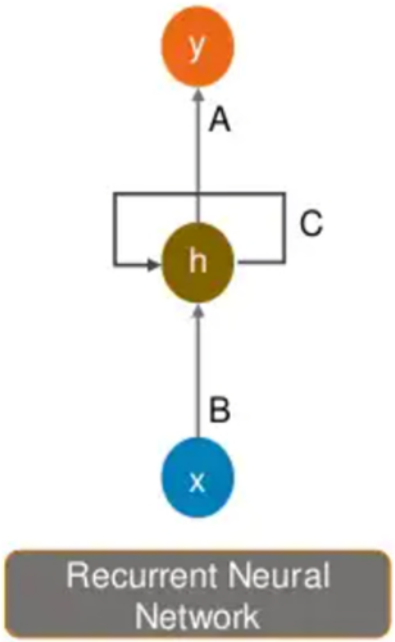
Limitations of Feedforward Network

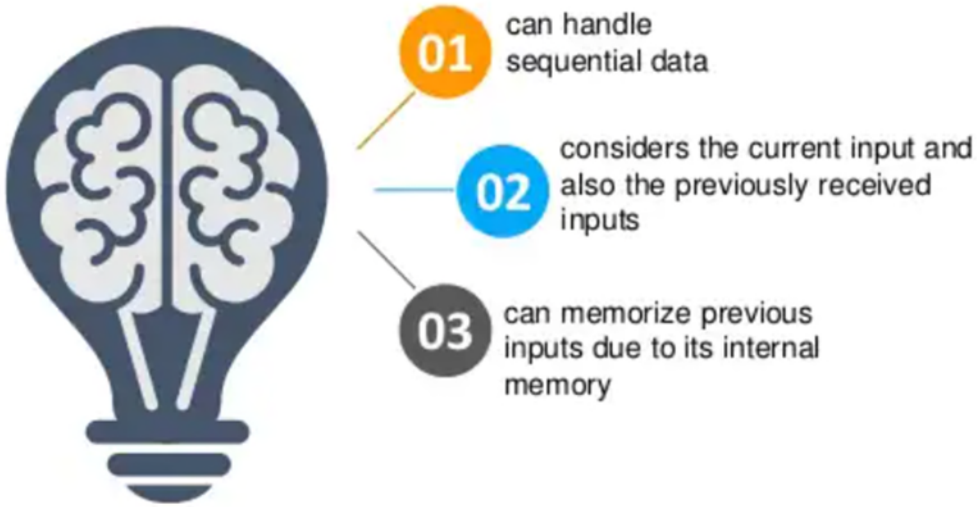




Source: Recurrent Neural Network (RNN) | RNN LSTM Tutorial | Deep Learning Course | Simplilearn

Why Recurrent Neural Network?

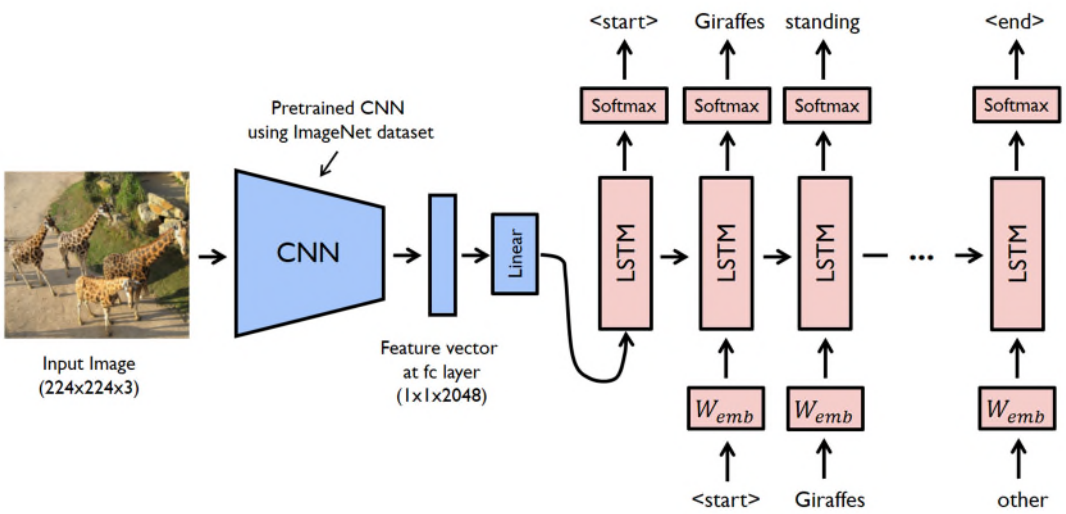


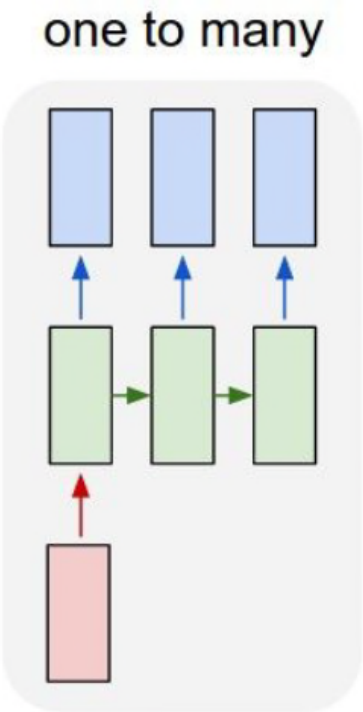


Sequence Modelling

Sequence Modelling (One-to-Many Mapping)

- One input, many outputs (sequence of outputs)
- E.g., image captioning
  - Image → sequence of words



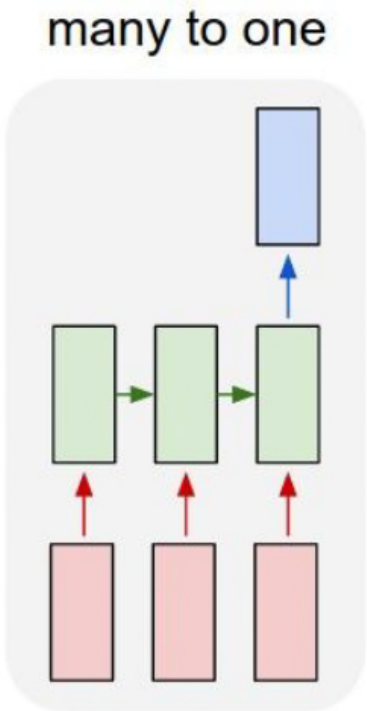


Source: UMich EECS 498-007 / 598-005: Deep Learning for Computer Vision

Source: https://www.analyticsvidhya.com/blog/2018/04/solving-an-image-captioning-task-using-deep-learning/

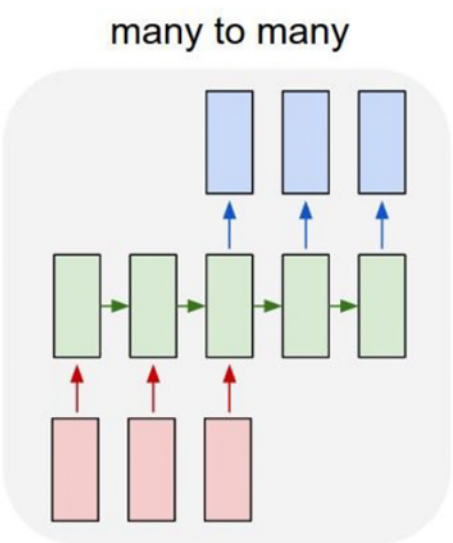
Sequence Modelling (Many-to-One Mapping)

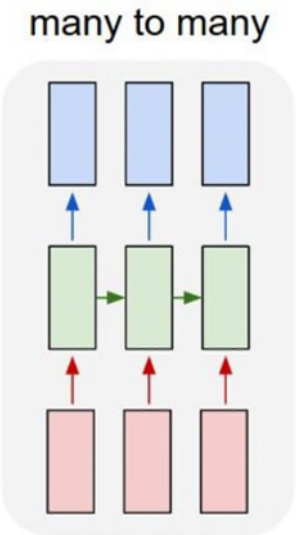
- Many inputs (sequence of inputs), one output
- E.g., Video classification
  - Sequence of images → Label
- Sentiment classification
  - Sequence of words → Label



Sequence Modelling (Many-to-Many Mapping)

- Many inputs (sequence of inputs), many outputs (sequence of outputs)
- E.g., machine translation
  - Sequence of words → sequence of words
- Per-frame video classification
  - Sequence of images → sequence of labels



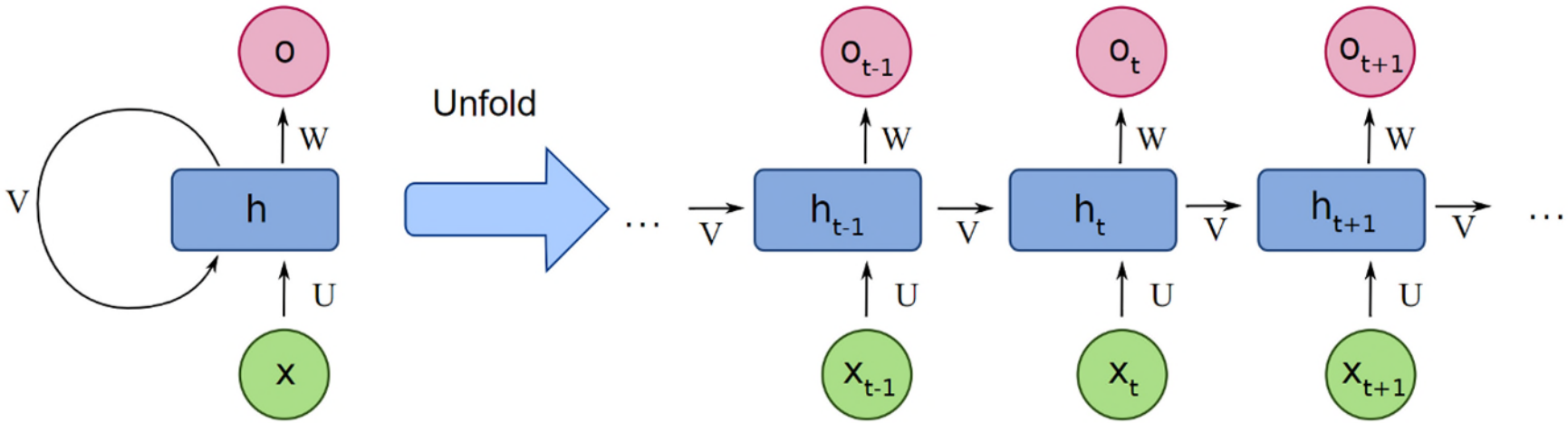


Recurrent Neural Networks (RNNs)

What are RNNs?

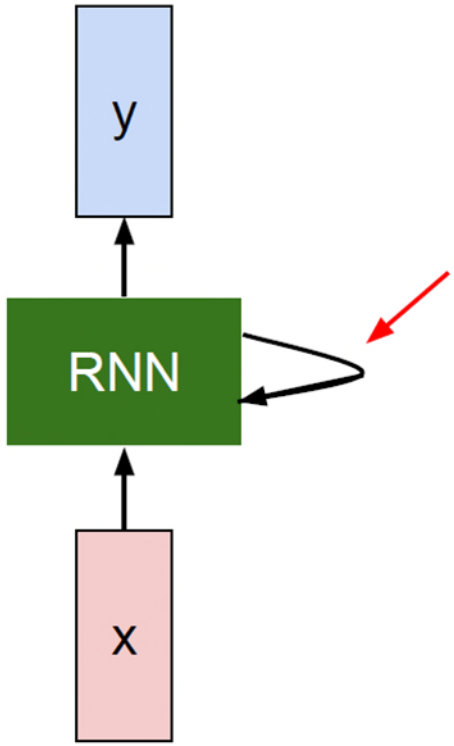
- RNNs are a class of neural networks which have inputs in the forms of sequential data, e.g. time series data.
- Commonly used in analysis of temporal / sequential data.
- Widely deployed in applications including Natural Language Processing (NLP), speech recognition, machine translation, image captioning, etc.

RNN Architecture



https://medium.com/deeplearningbrasilia/deep-learning-recurrent-neural-networks-f9482a24d010

RNN Model



Key idea: RNNs have an "internal state" that is updated as a sequence is processed

Source: Stanford Lecture Notes, CS231n

Hidden State Update

We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

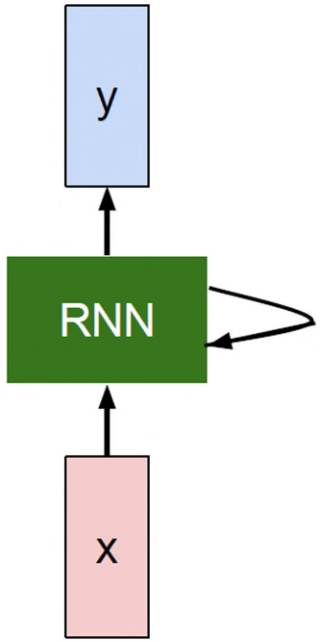
$$h_t = f_W(h_{t-1}, x_t)$$

new state

old state

input vector at some time step

some function with parameters W



Source: Stanford Lecture Notes, CS231n

Output Generation

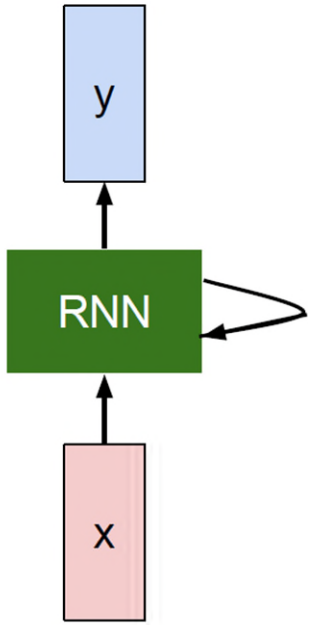
We can process a sequence of vectors **x** by applying a **recurrence formula** at every time step:

$$y_t = f_{W_{hy}}(h_t)$$

output

new state

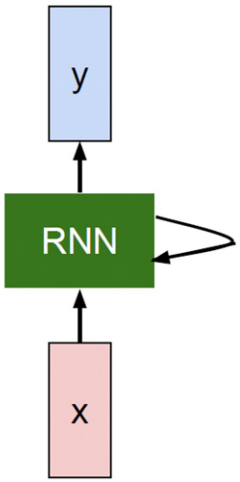
another function with parameters W_o



Source: Stanford Lecture Notes, CS231n

Vanilla RNN (Elman RNN)

The state consists of a single *"hidden"* vector **h**:

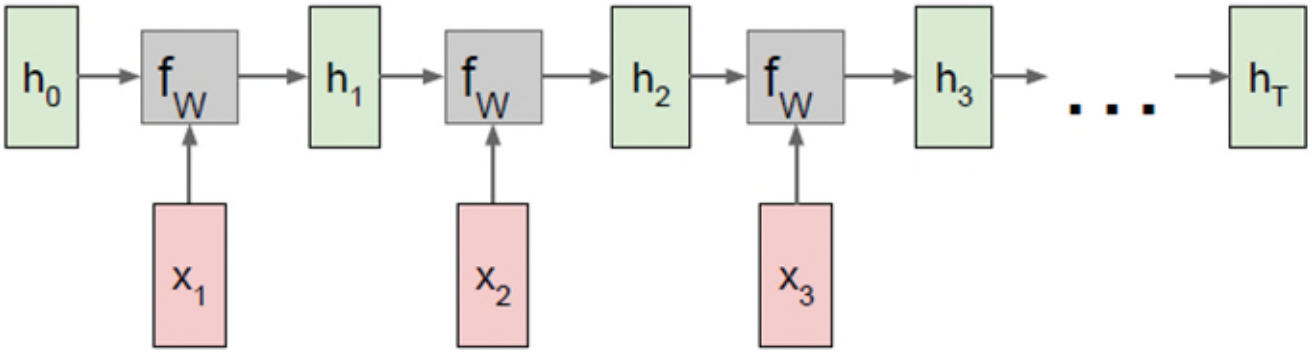


$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Sometimes called a "Vanilla RNN" or an "Elman RNN" after Prof. Jeffrey Elman



Source: Stanford Lecture Notes, CS231n

Exercise: RNN

(a) A Vanilla Recurrent Neural Network (RNN) has the following settings.

Initial hidden state, $\mathbf{h}_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$,

Hidden state weight matrix, $\mathbf{W}_{hh} = \begin{bmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \end{bmatrix}$,

Input weight matrix, $\mathbf{W}_{xh} = \begin{bmatrix} 0.5 & 0.2 \\ 0.2 & 0.1 \end{bmatrix}$,

Output weight matrix, $\mathbf{W}_{hy} = \begin{bmatrix} 0.1 & 0.4 \end{bmatrix}$.

Assume no bias is used in the computation of the RNN.

A 2-timestep input is given by $\mathbf{x} = \begin{bmatrix} \mathbf{x}_1 & \mathbf{x}_2 \end{bmatrix}$ where $\mathbf{x}_1 = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$ and $\mathbf{x}_2 = \begin{bmatrix} 1 \\ 6 \end{bmatrix}$.

(i) Find the hidden state $\mathbf{h}_1$ at timestep $t = 1$.

(ii) Find the output y_1 at timestep $t = 1$.

Solution

Pytorch RNN Implementation (1)

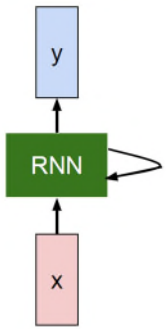
```
class Model(nn.Module):
    def __init__(self, input_size, output_size, hidden_dim, n_layers):
        super(Model, self).__init__()

        # Defining some parameters
        self.hidden_dim = hidden_dim
        self.n_layers = n_layers

        #Defining the layers
        # RNN Layer
        self.rnn = nn.RNN(input_size, hidden_dim, n_layers, batch_first=True)
        # Fully connected layer
        self.fc = nn.Linear(hidden_dim, output_size)
```

Layer Definition

The state consists of a single *"hidden"* vector **h**:



$$h_t = f_W(h_{t-1}, x_t)$$

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

$$y_t = W_{hy}h_t$$

Sometimes called a "Vanilla RNN" or an "Elman RNN" after Prof. Jeffrey Elman

Pytorch RNN Implementation (2)

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

Applies a multi-layer Elman RNN with $\tanh$ or ReLU non-linearity to an input sequence.

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

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

where h_t is the hidden state at time *t*, x_t is the input at time *t*, and $h_{(t-1)}$ is the hidden state of the previous layer at time *t-1* or the initial hidden state at time *0*. If `nonlinearity` is `'relu'`, then ReLU is used instead of $\tanh$.

```
>>> rnn = nn.RNN(10, 20, 2)
>>> input = torch.randn(5, 3, 10)
>>> h0 = torch.randn(2, 3, 20)
>>> output, hn = rnn(input, h0)
```

input_size=10, hidden_dim=20, n_layer=2 (batch_first: default is false)

seq_length=5, batch_size=3, input_size=10

n_layer=2, batch_size=3, hidden_dim=20

```
# RNN Layer
self.rnn = nn.RNN(input_size, hidden_dim, n_layers
```

- **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`

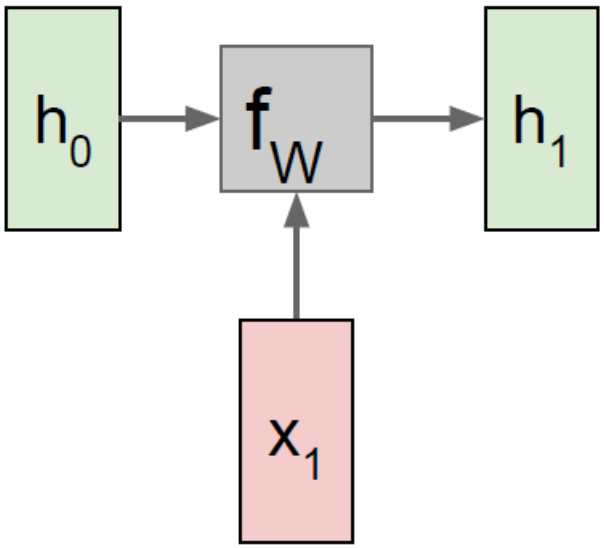
Source: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html

RNN Pros and Cons

- Pros:
  - Can process any length of input
  - In theory, computation for step *t* can use information from many steps back
  - Model size does not increase for longer input
  - Same weights applied for every timestep, so there is consistency in how inputs are processed.
- Cons:
  - Recurrent computation is slow
  - In practice, difficult to leverage information from many steps back

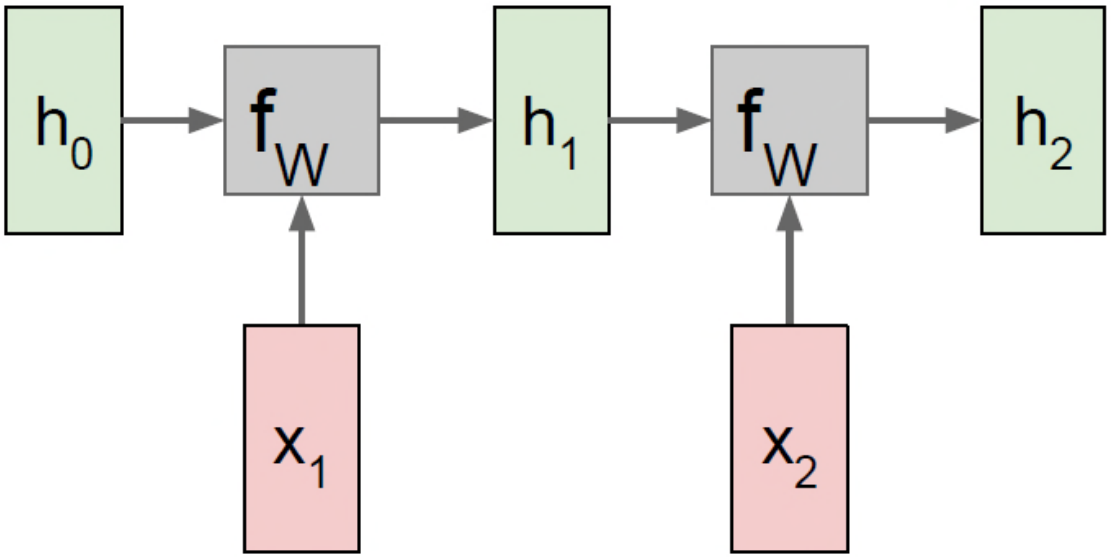
RNN Training & Optimization

Computational Graph (1)



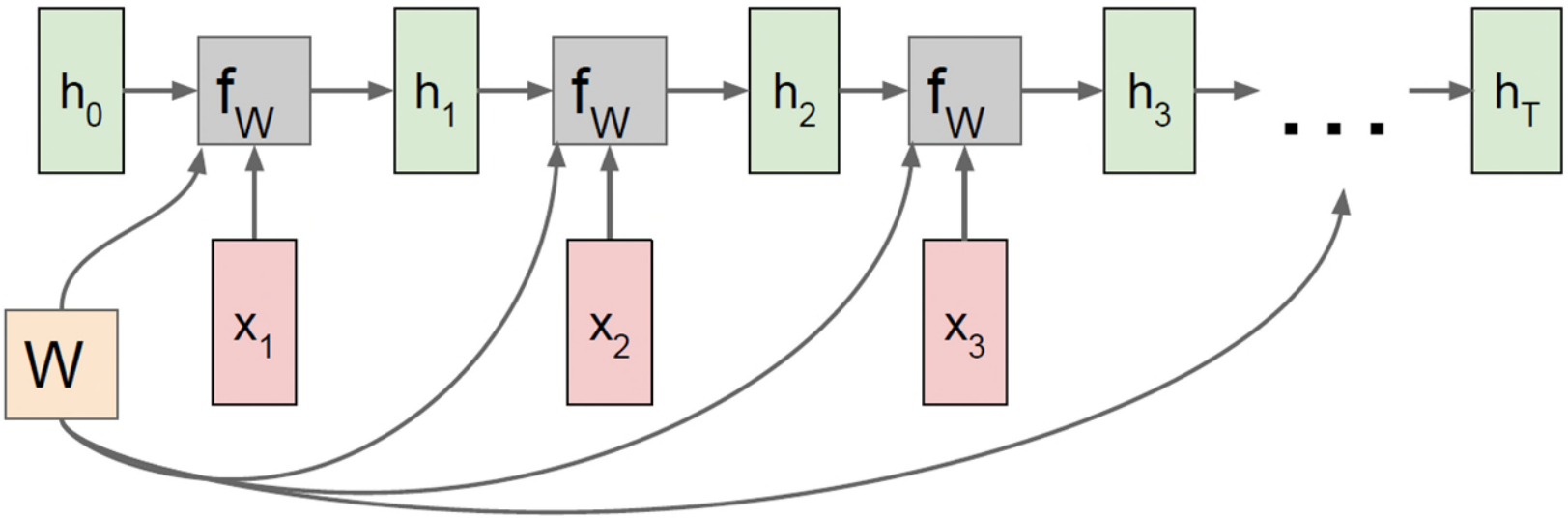
Source: Stanford Lecture Notes, CS231n

Computational Graph (2)

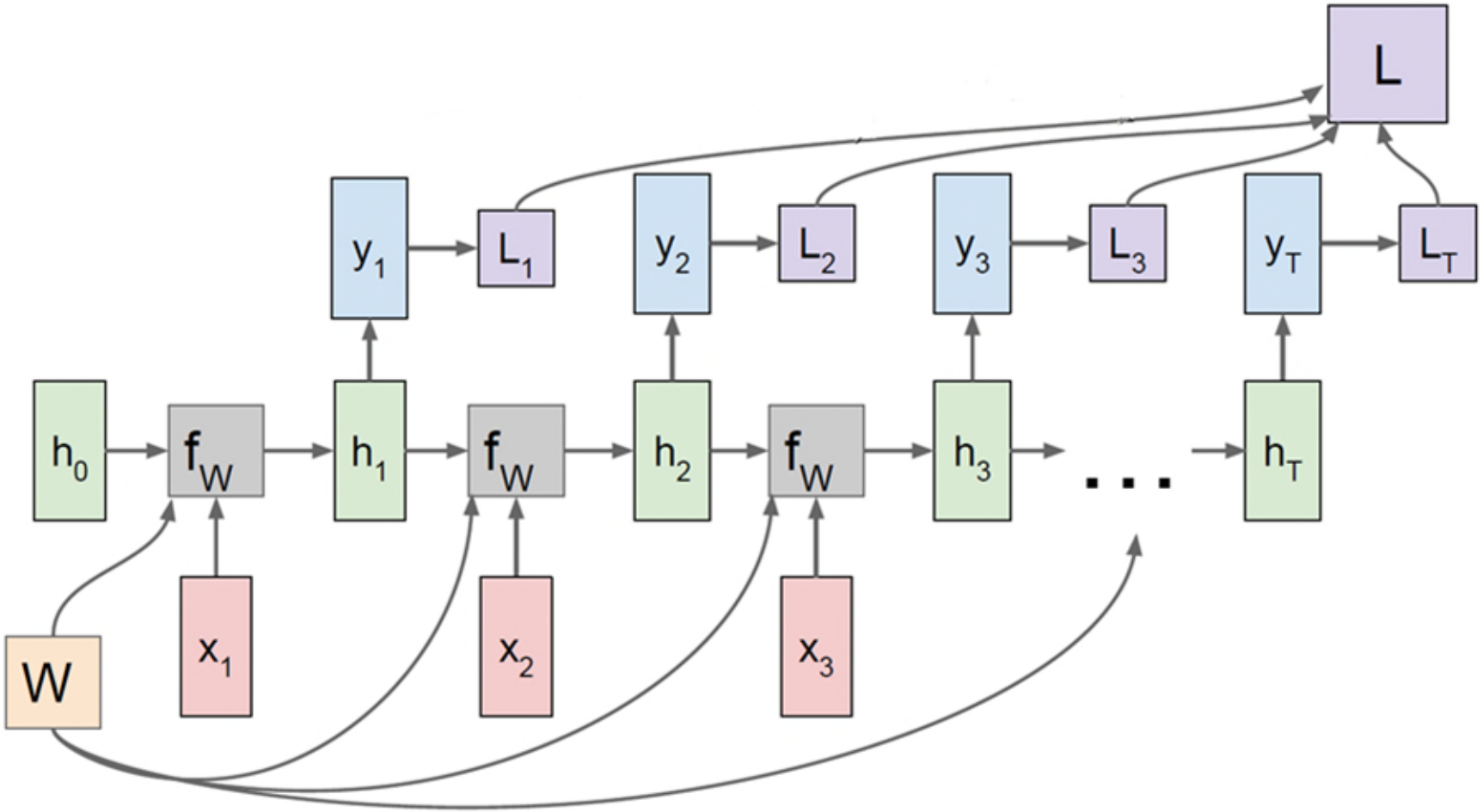


Computational Graph (3)

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

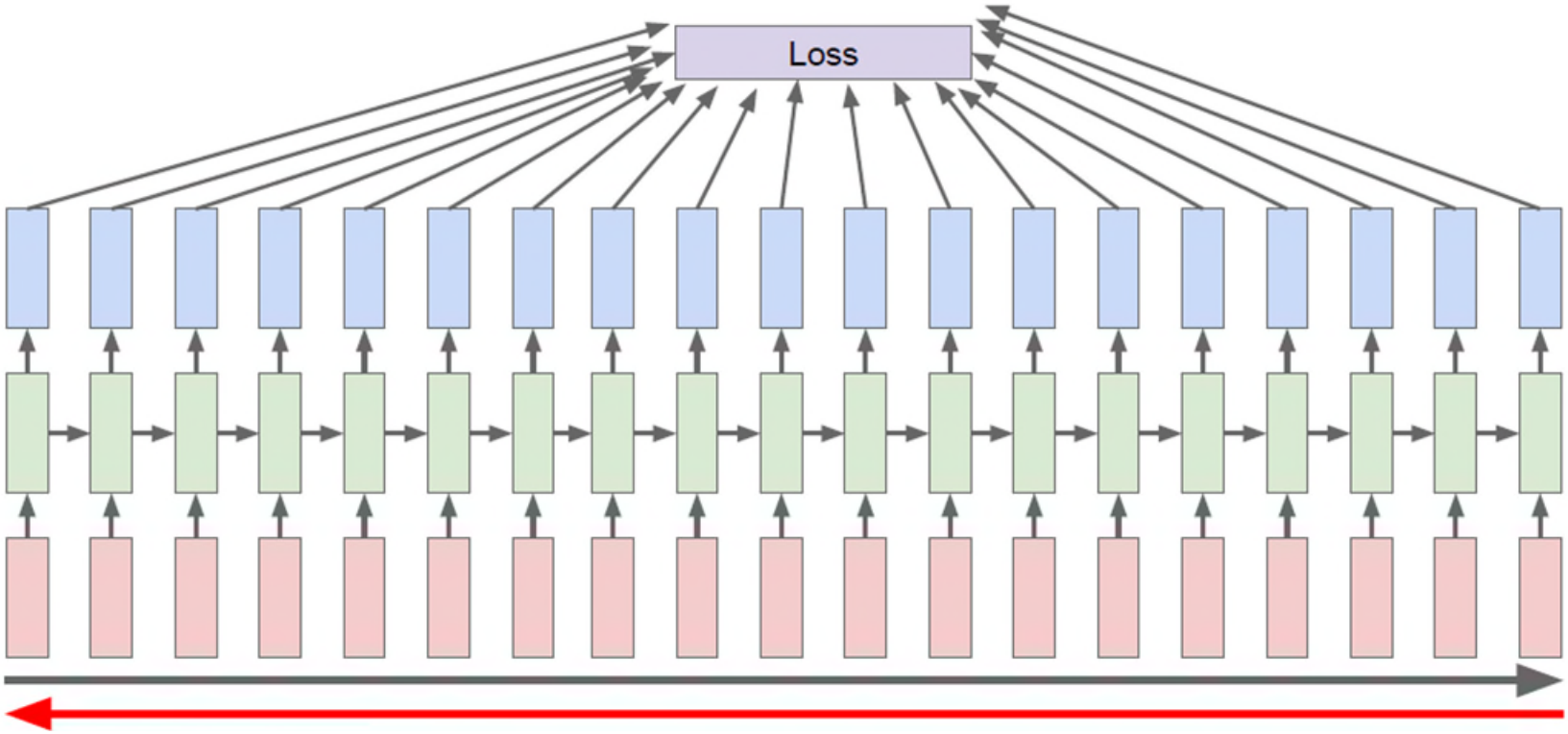


Computational Graph (4)

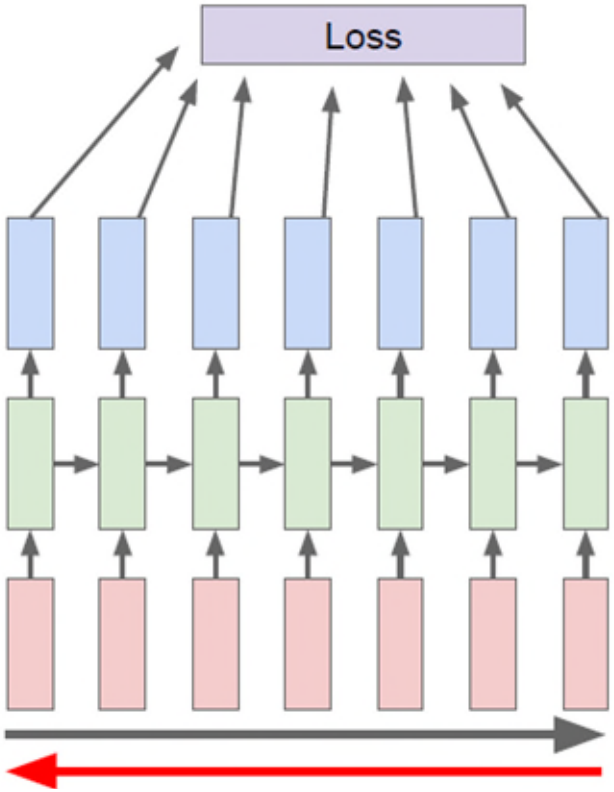


Backpropagation Through Time (BPTT)

Forward through entire sequence to compute loss, then backward through entire sequence to compute gradient

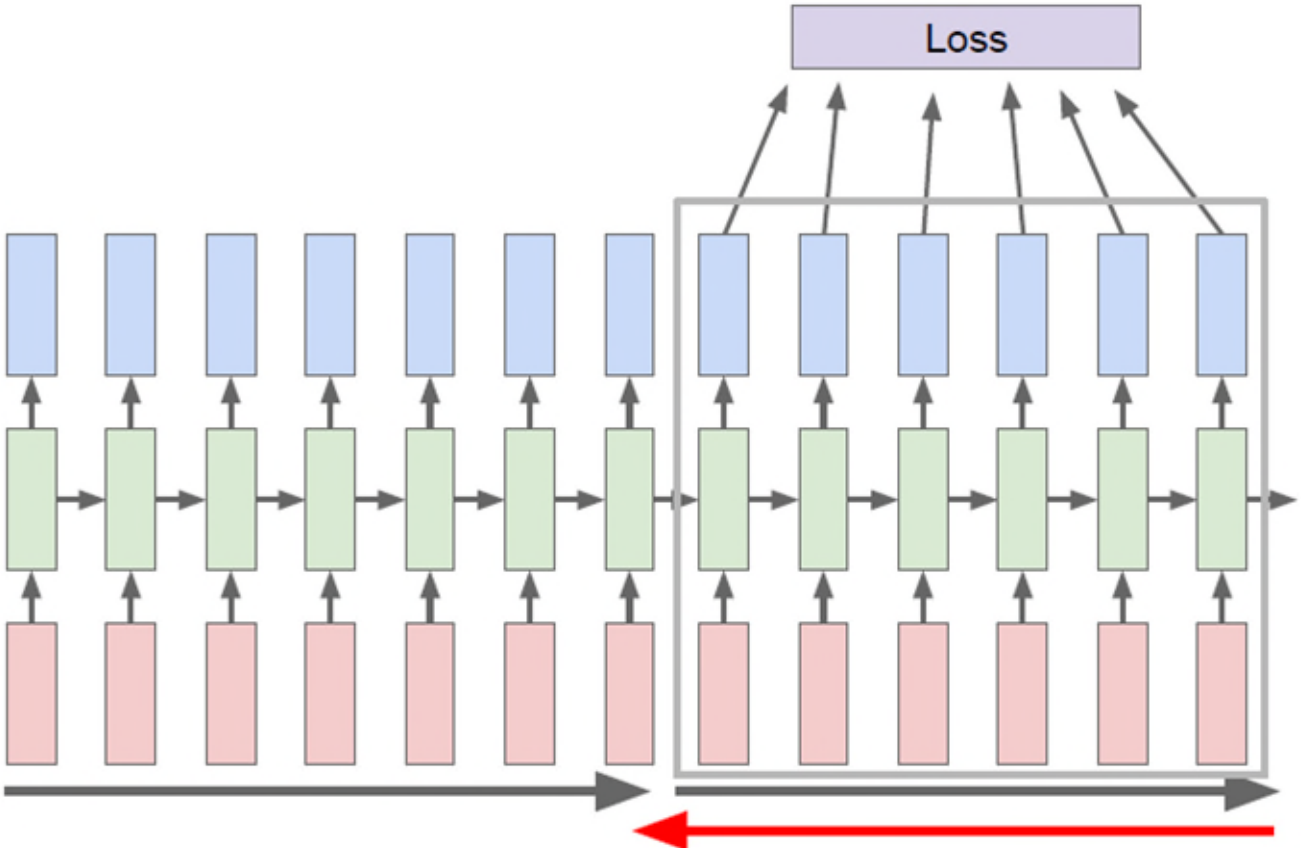


Truncated Backpropagation Through Time (1)



Run forward and backward through chunks of the sequence instead of whole sequence

Truncated Backpropagation Through Time (2)

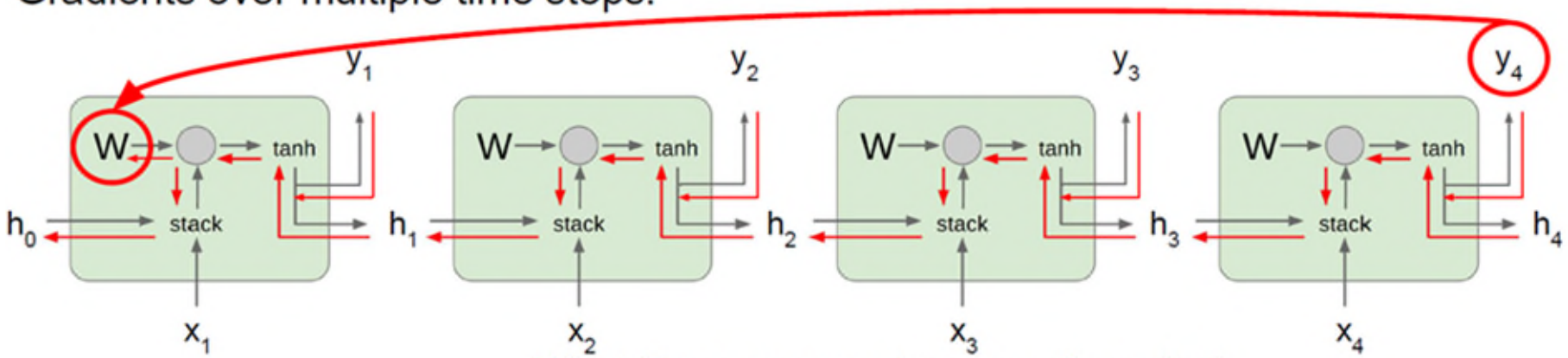


Carry hidden states forward in time forever, but only backpropagate for some smaller number of steps

Exploding / Vanishing Gradients

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

Gradients over multiple time steps:



What if we assumed no non-linearity?

$$\frac{\partial L}{\partial W} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial W}$$

Largest singular value > 1:
Exploding gradients

$$\frac{\partial L_T}{\partial W} = \frac{\partial L_T}{\partial h_T} W_{hh}^{T-1} \frac{\partial h_1}{\partial W}$$

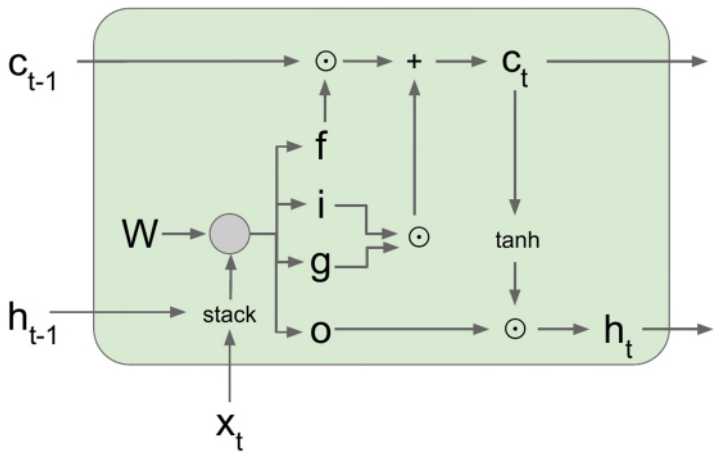
Largest singular value < 1:
Vanishing gradients → Change RNN architecture

Source: Stanford Lecture Notes, CS231n

Long Short-Term Memory (LSTM)

Key Ideas

- A memory cell which can maintain its state over time.
- Consist of cell state vector (c_t), gates (i, f, o, g) and hidden state (h_t)
- h_t : short term memory, c_t : long term memory.
- Cell state vector (c_t) undergoes changes via forgetting old memory (through forget (f) gate) and adding new memory (through input (i) gate & gate (g) gate).
- Gates control the flow of information to the memory.
- Gate are obtained through a sigmoid/tanh layer, and they update c_t and h_t cell states using pointwise multiplication operator.



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

RNNs vs LSTMs

Vanilla RNN

$$h_t = \tanh\left(W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

LSTM

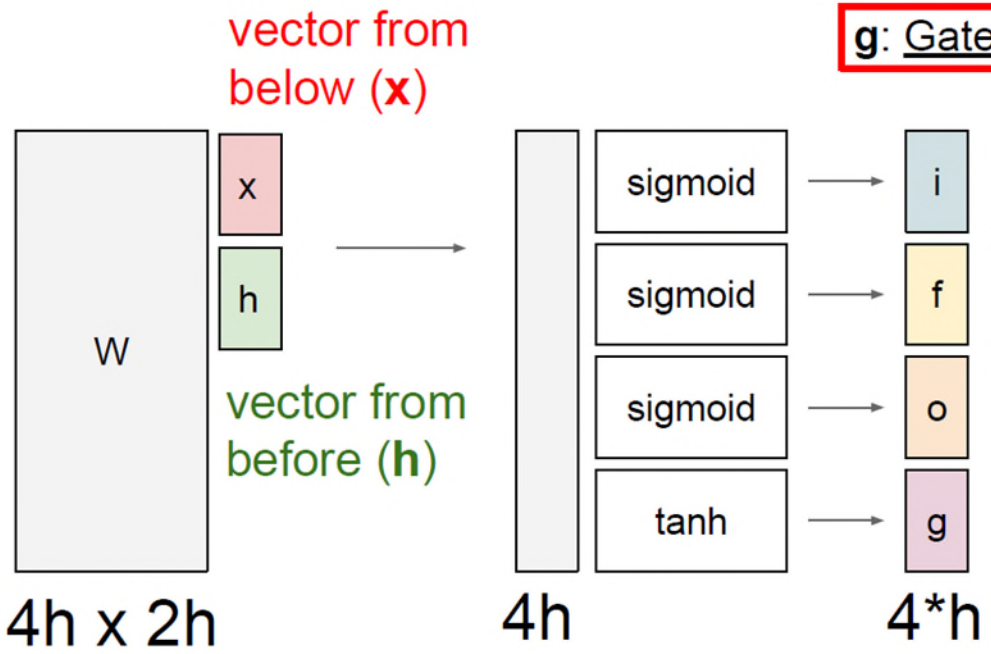
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Source: Stanford Lecture Notes, CS231n

LSTM Computation (1)



g: Gate gate (?), How much to write to cell

$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

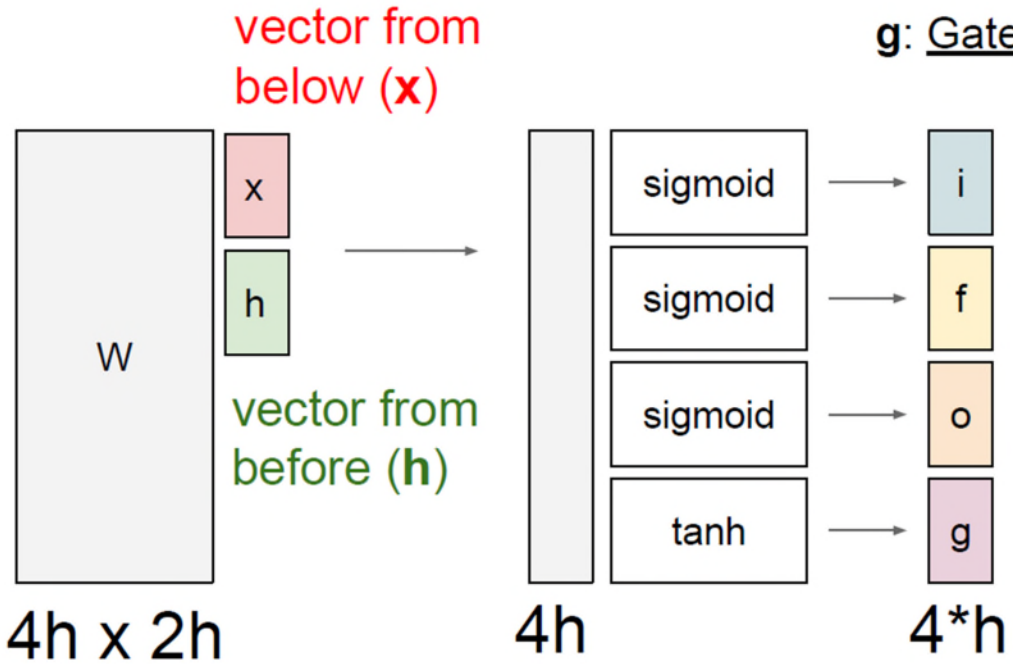
$$h_t = o \odot \tanh(c_t)$$

Source: Stanford Lecture Notes, CS231n

LSTM Computation (2)

i: Input gate, whether to write to cell

g: Gate gate (?), How much to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

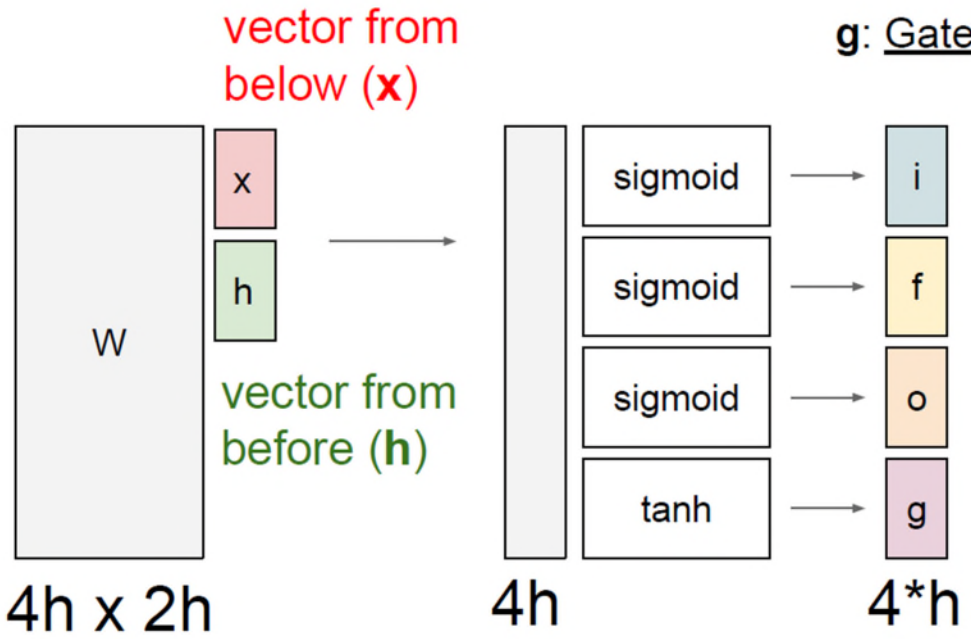
Source: Stanford Lecture Notes, CS231n

LSTM Computation (3)

i: Input gate, whether to write to cell

f: Forget gate, Whether to erase cell

g: Gate gate (?), How much to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

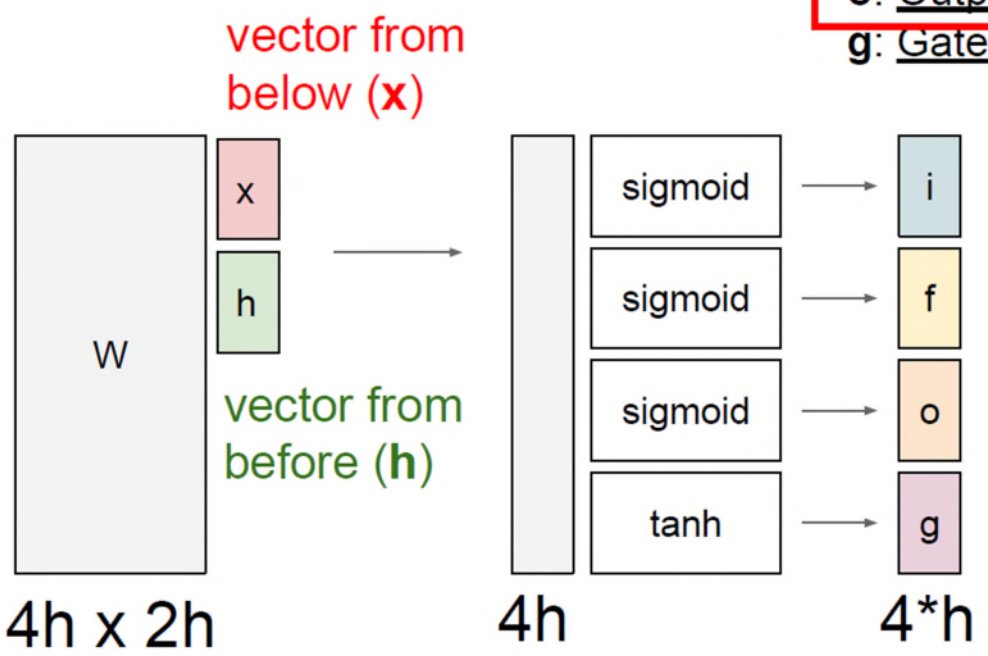
$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Source: Stanford Lecture Notes, CS231n

LSTM Computation (4)

i: Input gate, whether to write to cell
f: Forget gate, Whether to erase cell
o: Output gate, How much to reveal cell
g: Gate gate (?), How much to write to cell



$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$

$$c_t = f \odot c_{t-1} + i \odot g$$

$$h_t = o \odot \tanh(c_t)$$

Source: Stanford Lecture Notes, CS231n

Exercise: LSTM

Given an LSTM with the following settings:

Given input **x** at 2 time-steps: $\mathbf{x}_1 = \begin{bmatrix}1\\2\end{bmatrix}$, $\mathbf{x}_2 = \begin{bmatrix}3\\4\end{bmatrix}$; Initial hidden state, $\mathbf{h}_0 = \begin{bmatrix}0\\0\end{bmatrix}$

Output gate weight matrix $\boldsymbol{W_o} = [\mathbf{W}_{ho}\, \mathbf{W}_{xo}] = \begin{bmatrix}0.2 & 0.1 & 0.2 & 0.1\\0.1 & 0.2 & 0.1 & 0.2\end{bmatrix}$

Assume cell state $\mathbf{c}_1 = \begin{bmatrix}0.31\\0.31\end{bmatrix}$. Assume no bias is used in the computation.

(a) What is the output gate $\mathbf{o}_1$ at time-step t_1 (rounded to 2 decimal places)?

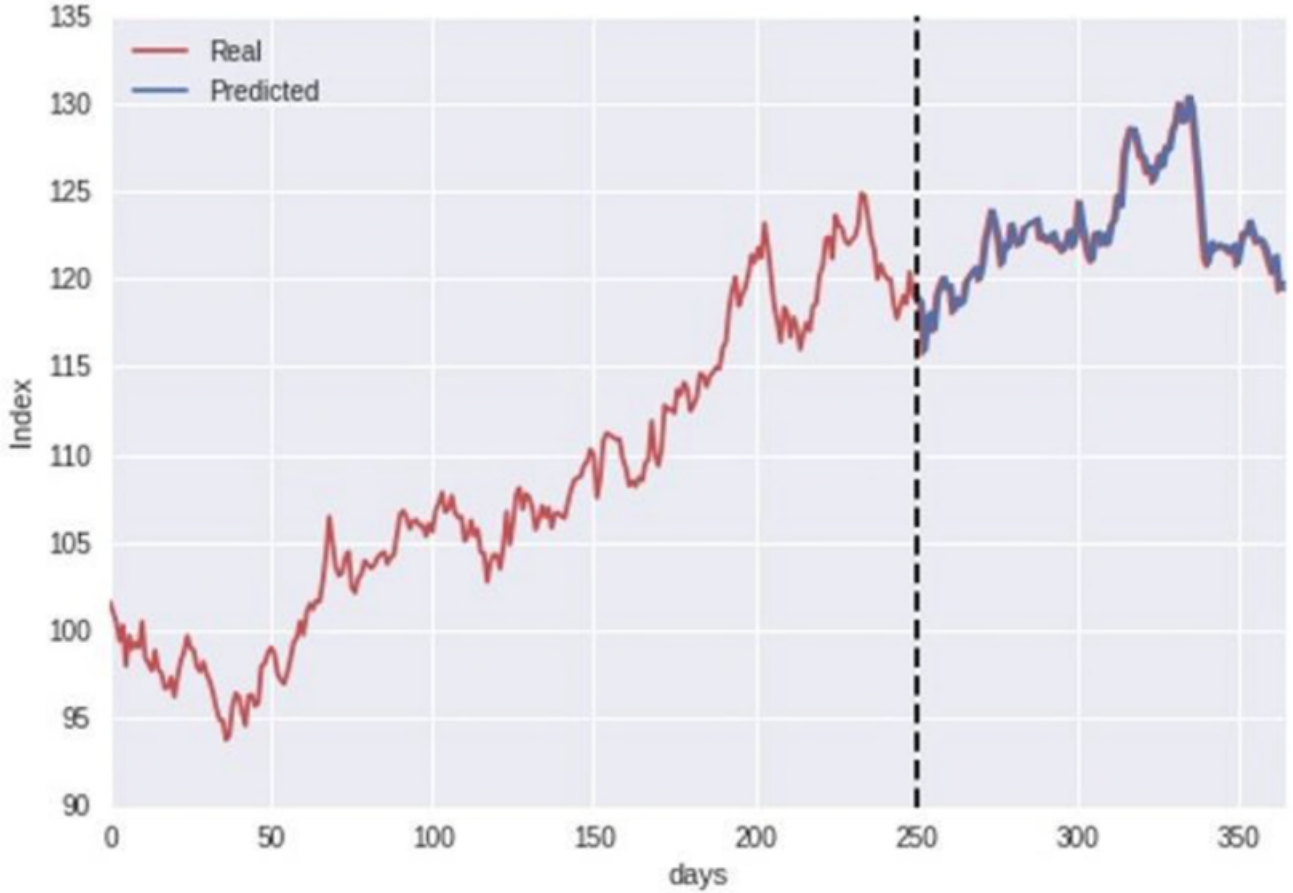
(b) What is the hidden state $\mathbf{h}_1$ at time-step t_1 (rounded to 2 decimal places)?

$$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

$$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$$

Solution

Application: Time Series Prediction

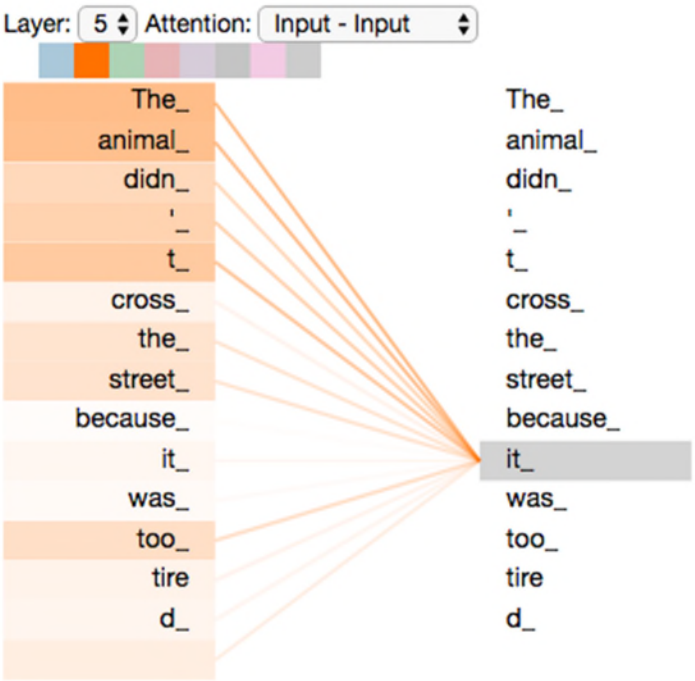


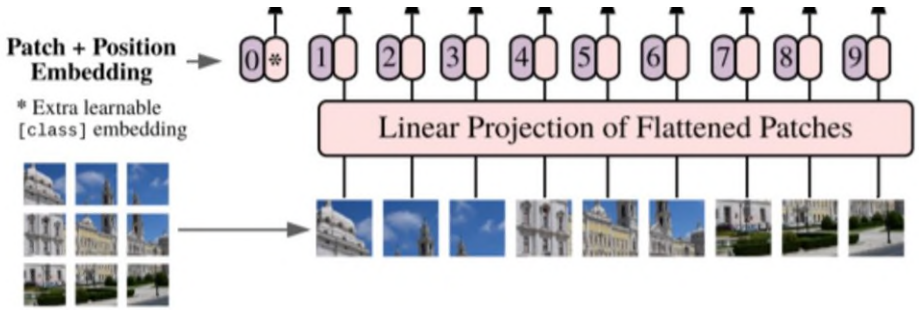
Source: https://www.kdnuggets.com/2019/05/machine-learning-time-series-forecasting.html

Transformer

What is Attention?

- Attention is used to determine which input tokens (e.g., words in NLP, image patches in CV) are relevant to the current input / token.
- Attention is computed through correlation (dot product) between 2 vectors.
- Correlation -> similarity / relatedness / importance -> attention



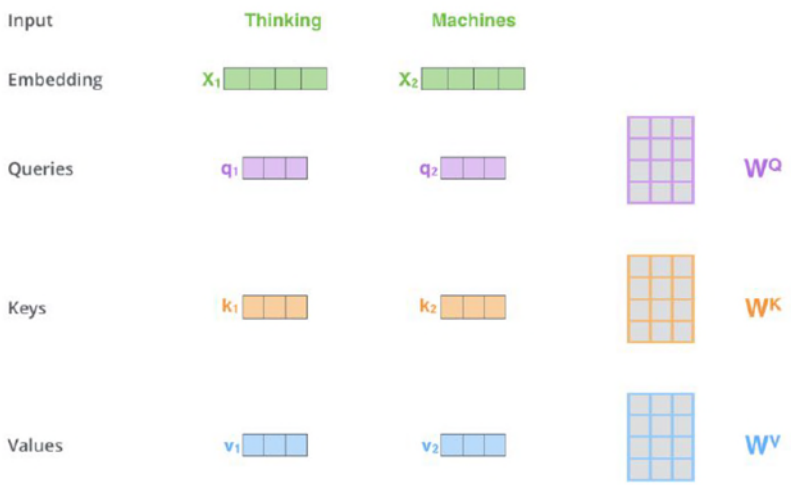


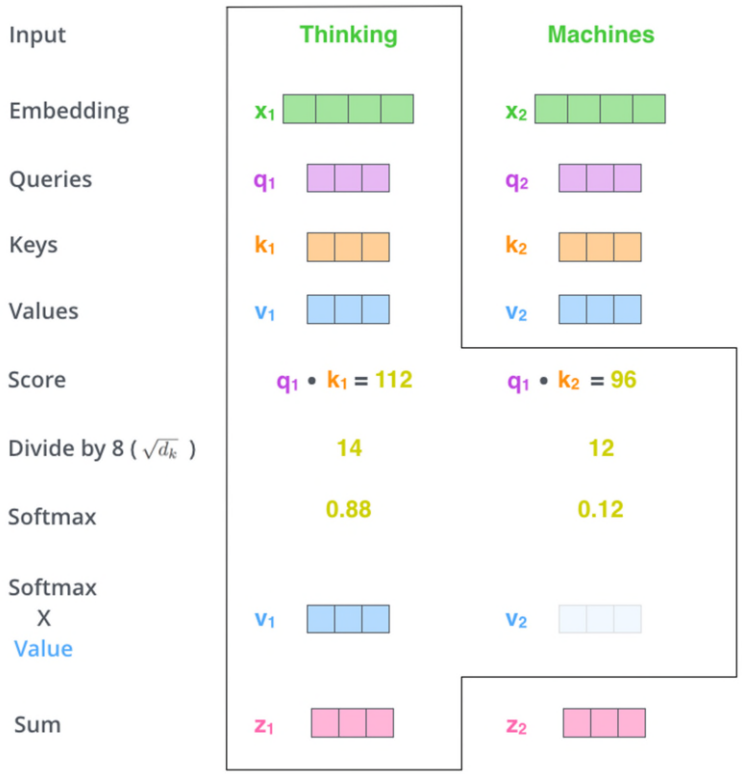
Source: https://jalammar.github.io/illustrated-transformer/
An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale (2020), Alexey Dosovitskiy et. al.

How is Attention Computed?

- Each input token generates 3 vectors: query (q), key (k) and value (v). These vectors provide more flexible representation through linear mapping (W_Q, W_K, W_v) to learn the underlying relationship / attention between input tokens.
- Attention is computed using:
  - Step 1: compute the correlation (dot product) between the query (q) and key (k) vectors.
  - Step 2: correlation values from Step 1 are scaled and normalized using Softmax function.
  - Step 3: multiplied output from Step 2 by corresponding value (v) vectors and sum them up.

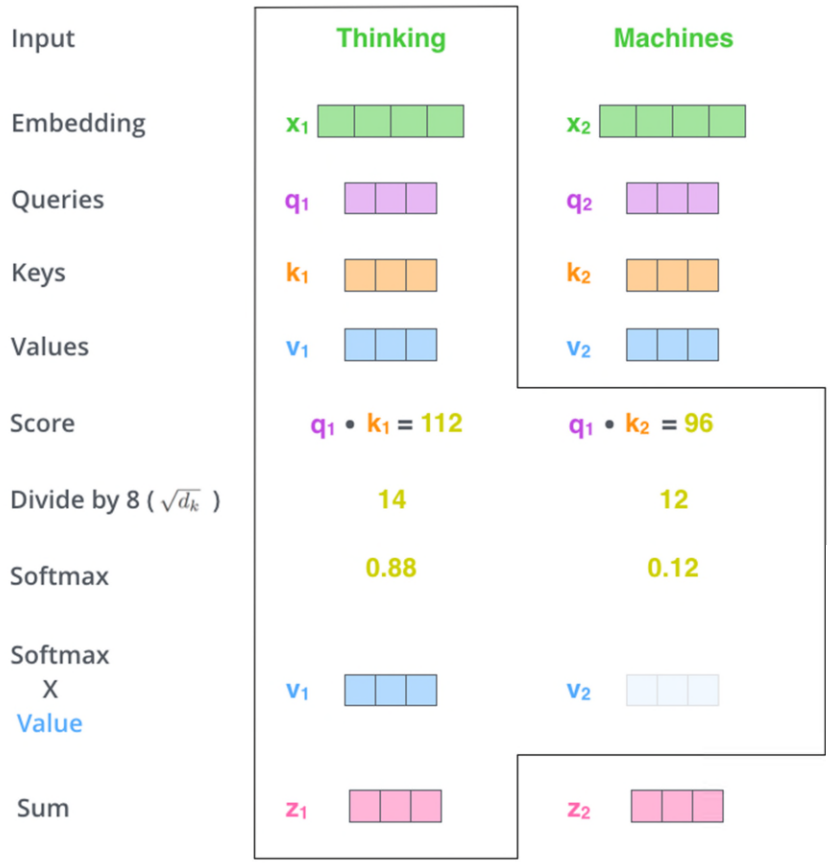
$$A(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



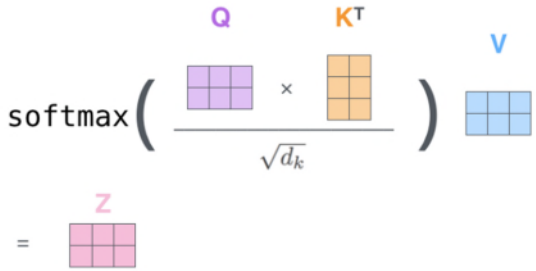


Scaled Dot-Product Attention

- Step 1: compute the correlation (dot product) between the query (q) and key (k) vectors.
- Step 2: correlation values from Step 1 are scaled and normalized using Softmax function.
- Step 3: multiplied output from Step 2 by corresponding value (v) vectors and sum them up.



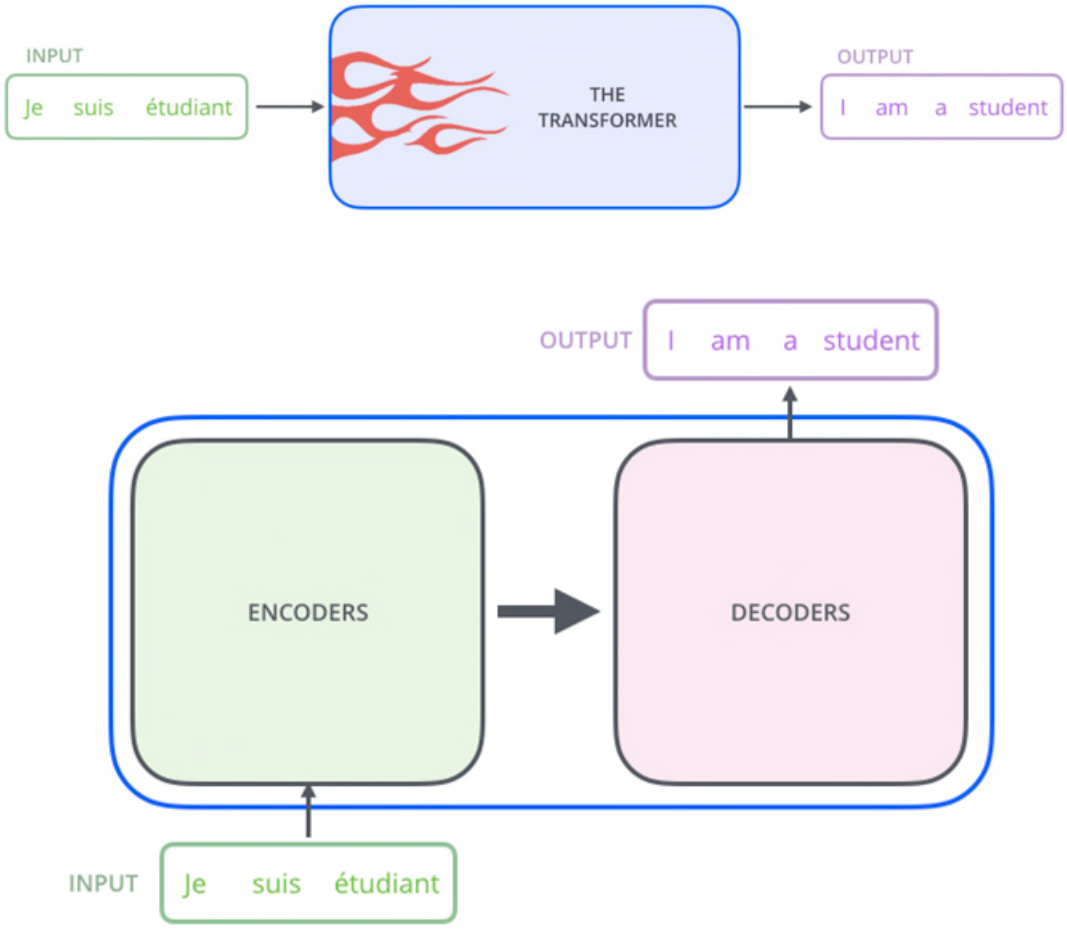
$$A(Q,K,V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



What is a Transformer?

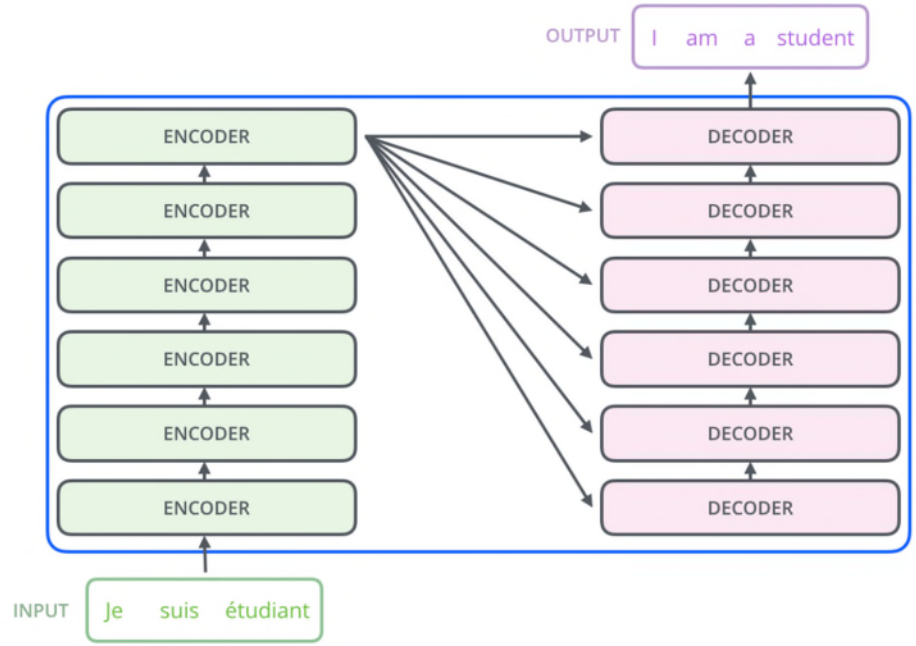
- Transformer uses attention mechanism to process input sequence in parallel.
- Use attention mechanism and dense/feedforward/MLP layers.
- Highly parallelizable.
- Can offer global attention.
- Good at modelling long-range dependency.
- Achieve state-of-the-art performance in many vision and NLP applications.
- Lead to other SOTA methods such as BERT: Pre-training of Deep Bidirectional Transformers for Language.

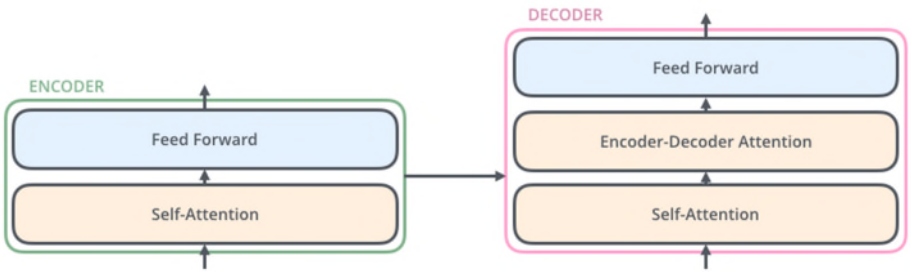
Transformer in Machine Translation (1)

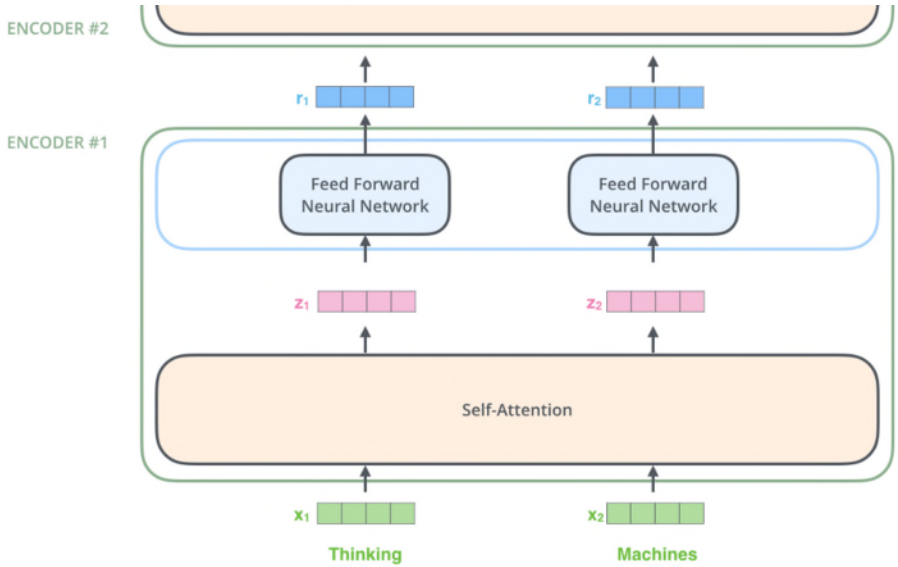


Source: https://jalammar.github.io/illustrated-transformer/

Transformer in Machine Translation (2)







Source: https://jalammar.github.io/illustrated-transformer/

Attention is All You Need

- Initially designed for neural machine translation.
- Later extended to visual tasks such as recognition, detection, etc, with great success.
- Leverage on attention mechanism to analyze the importance of a token (word/image patch) with respect to other tokens/image patches.
- Consist of transformer encoder and transformer decoder.

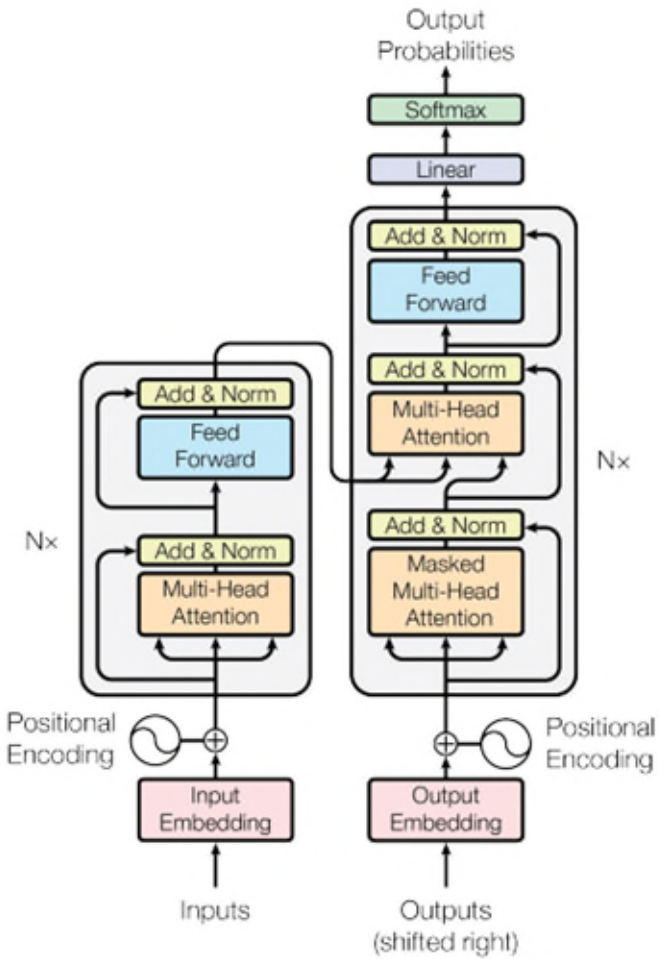
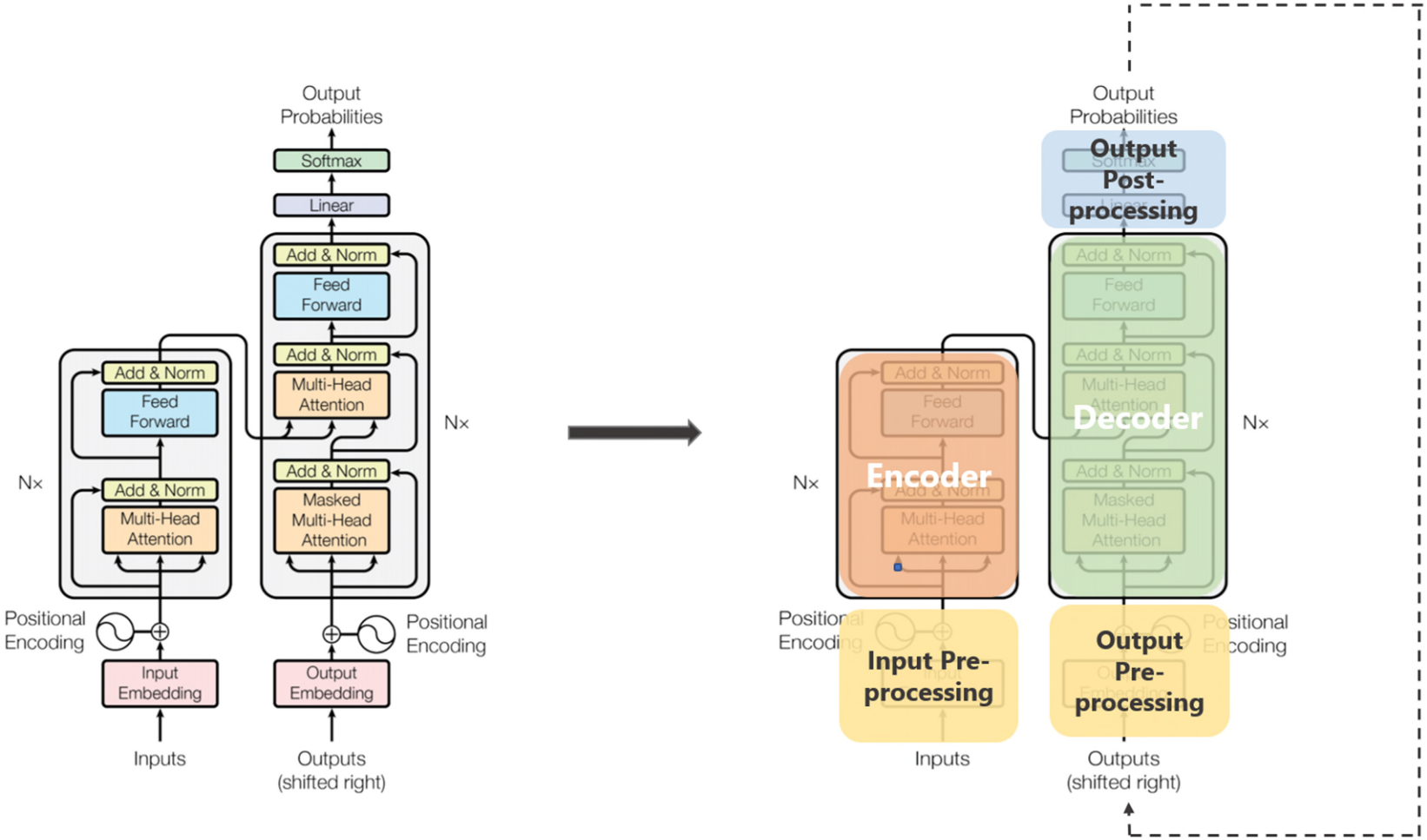


Figure 1: The Transformer - model architecture.

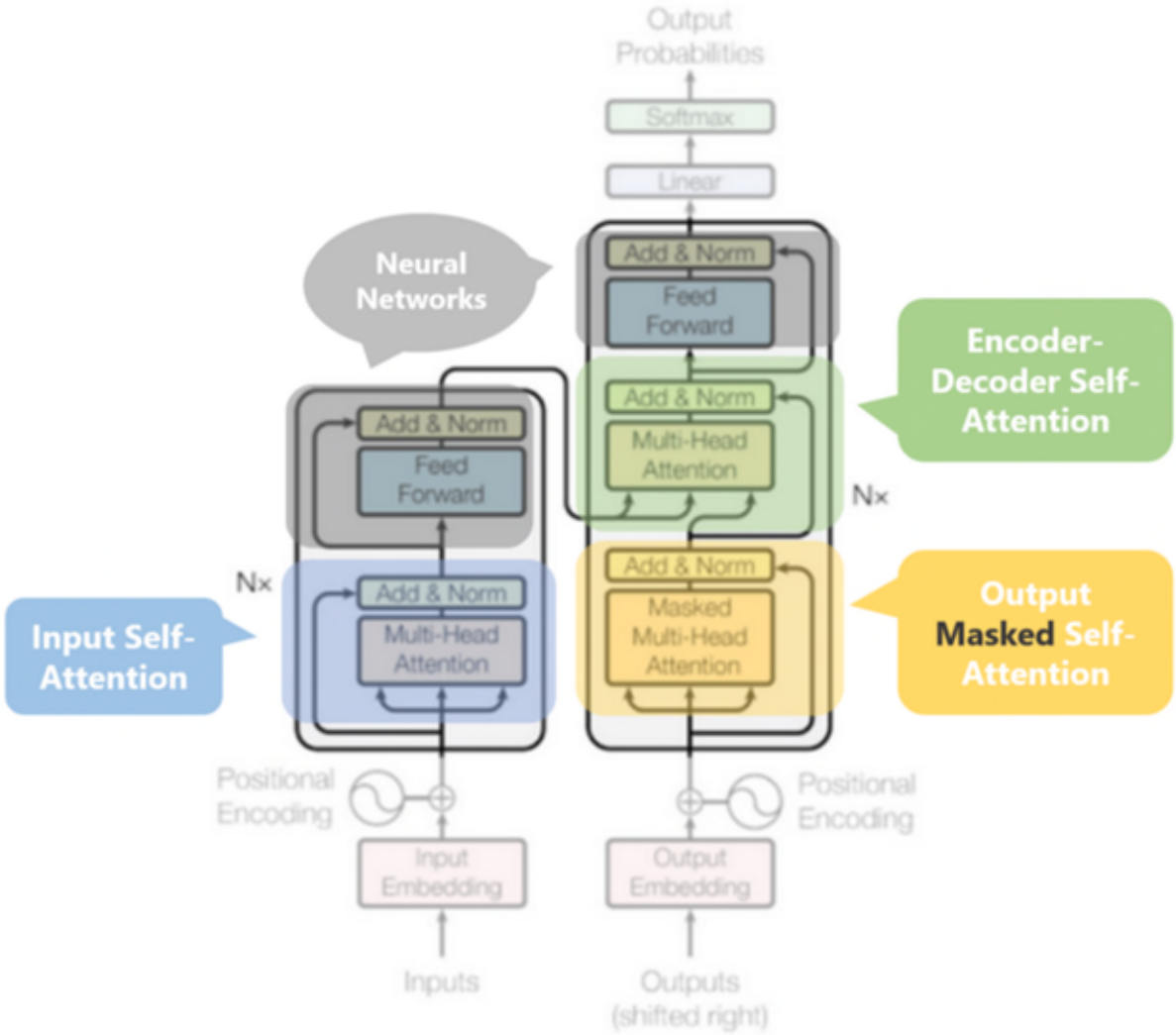
Attention Is All You Need (2017) https://arxiv.org/abs/1706.03762
Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, Illia Polosukhin

Transformer Architecture (1)



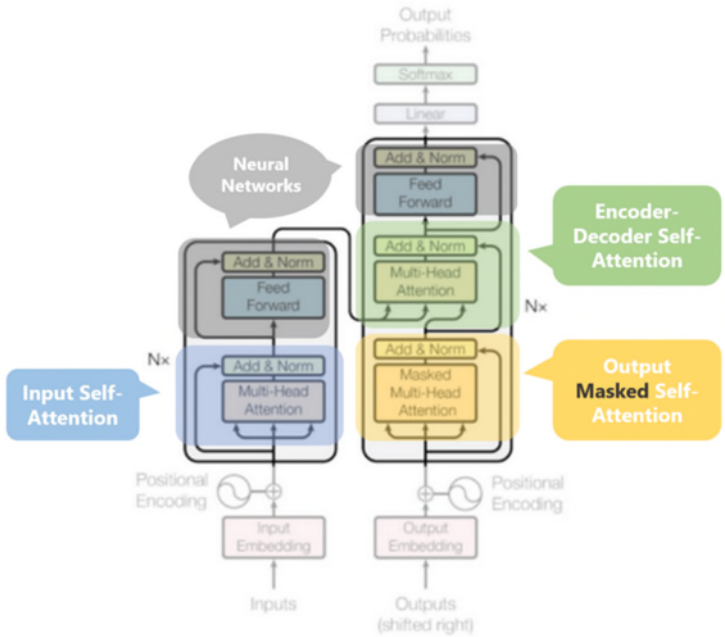
Source: Ria Kulshrestha, Transformers. https://towardsdatascience.com/transformers-89034557de14

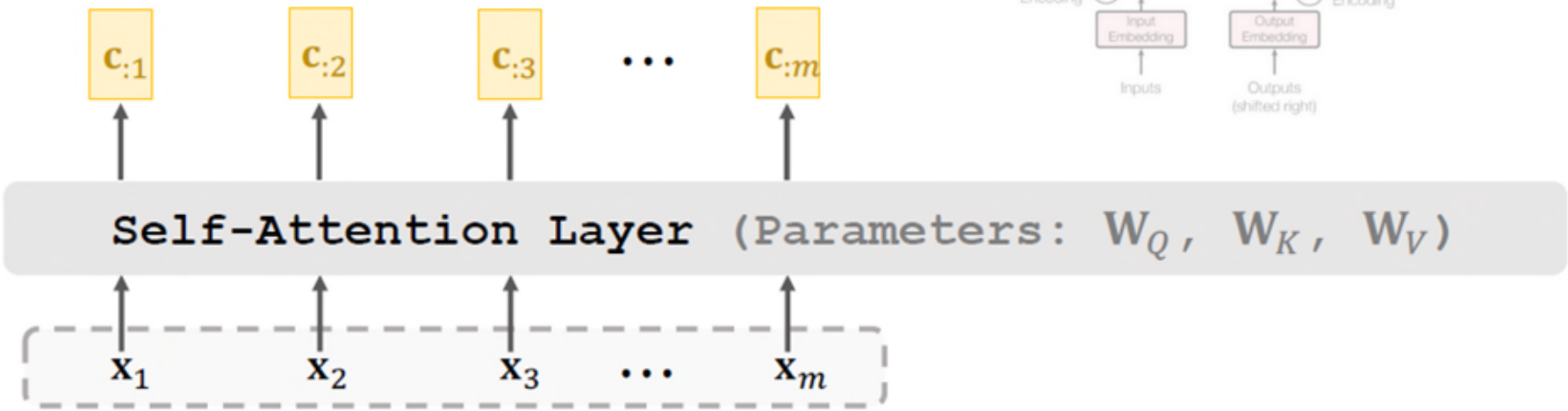
Transformer Architecture (2)



Self-Attention Layer Overview

- Self-attention layer: $\mathbf{C} = \mathrm{Attn}(\mathbf{X}, \mathbf{X})$.
  - Inputs: $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \cdots, \mathbf{x}_m]$.
  - Parameters: $\mathbf{W}_Q$, $\mathbf{W}_K$, $\mathbf{W}_V$.



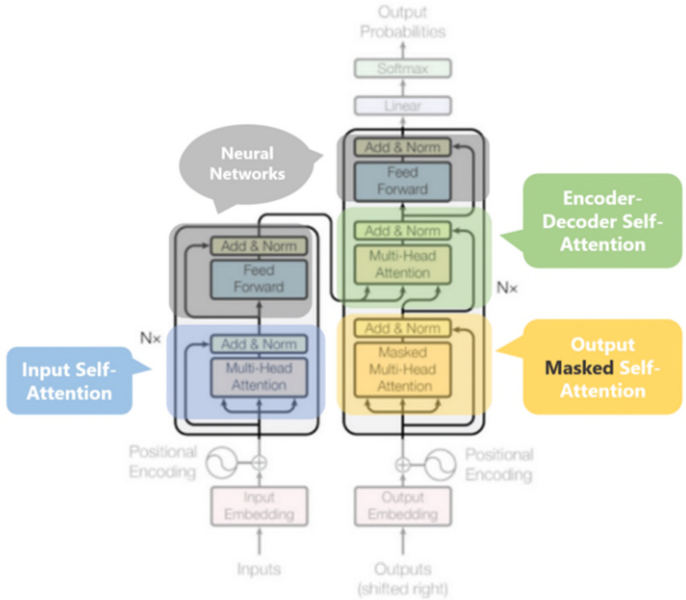


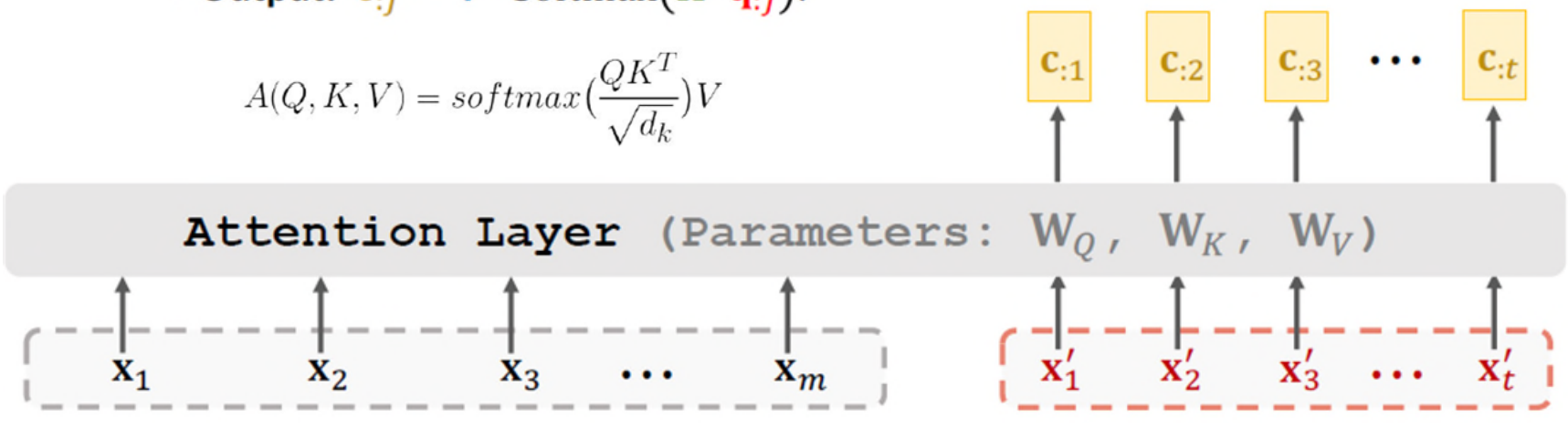
Source: Shusen Wang, Transformer Model

Cross-Attention / Encoder-Decoder Self-Attention

- Attention layer: $\mathbf{C} = \mathrm{Attn}(\mathbf{X}, \mathbf{X}')$.
  - **Query:** $\mathbf{q}_{:j} = \mathbf{W}_Q \mathbf{x}'_j$,
  - **Key:** $\mathbf{k}_{:i} = \mathbf{W}_K \mathbf{x}_i$,
  - **Value:** $\mathbf{v}_{:i} = \mathbf{W}_V \mathbf{x}_i$.
  - **Output:** $\mathbf{c}_{:j} = \mathbf{V} \cdot \mathrm{Softmax}(\mathbf{K}^T \mathbf{q}_{:j})$.

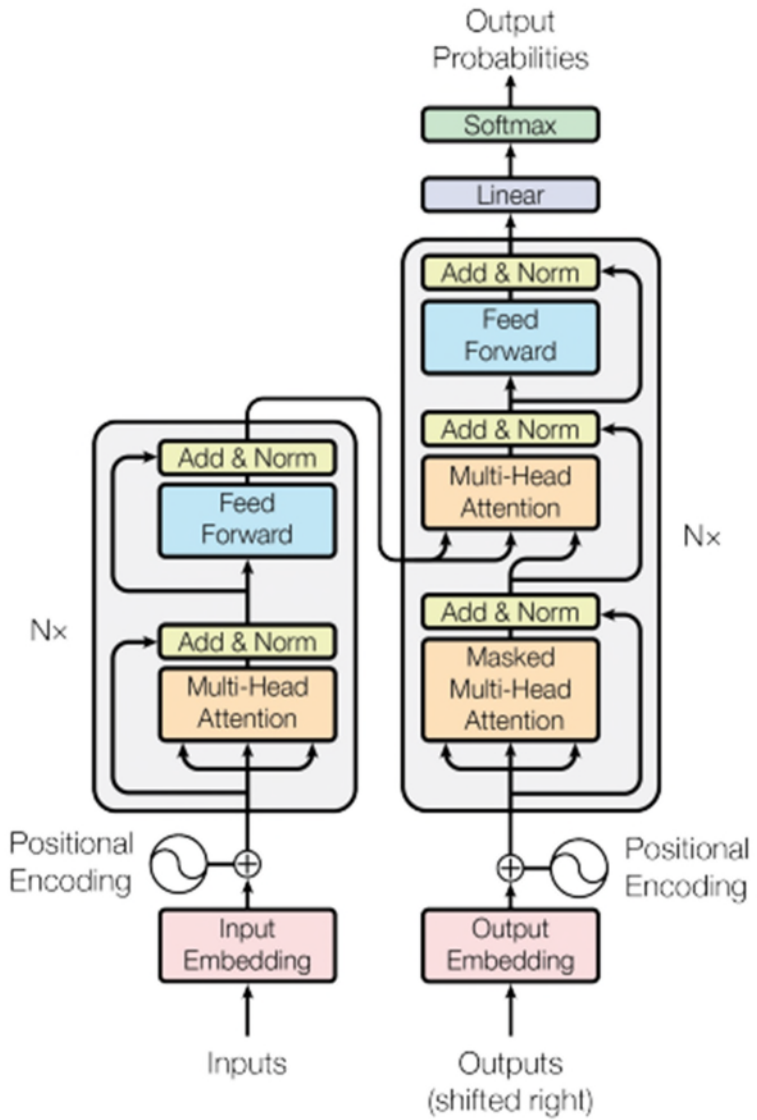
$$A(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$



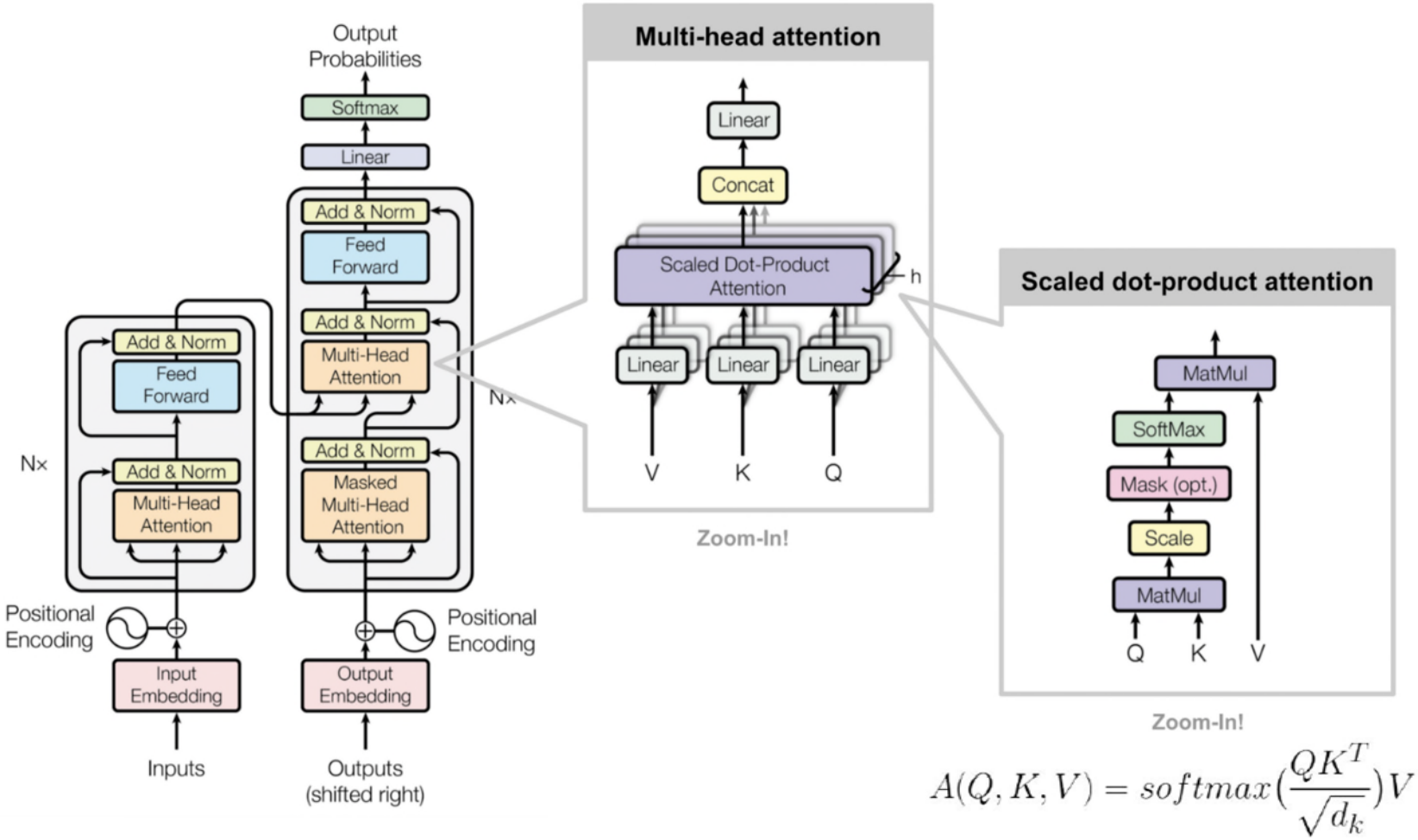


More Details

- Multi-Head Attention
- Feed Forward Network
- Positional Encoding
- Residual Connection
- Layer Normalization (Norm)
- Input/Output Embedding
- Masked Attention



Multi-Head Attention

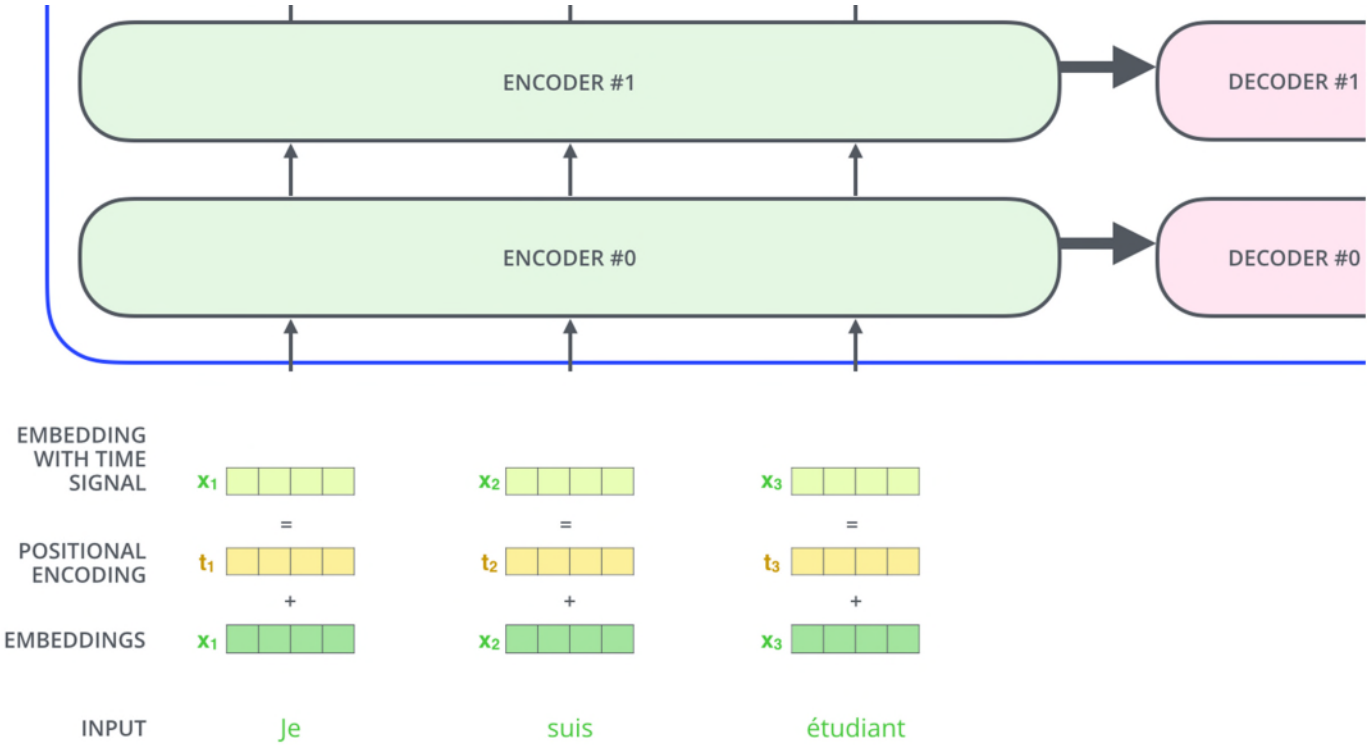


$$A(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

Source: https://deepfrench.gitlab.io/deep-learning-project/

Position Encoding

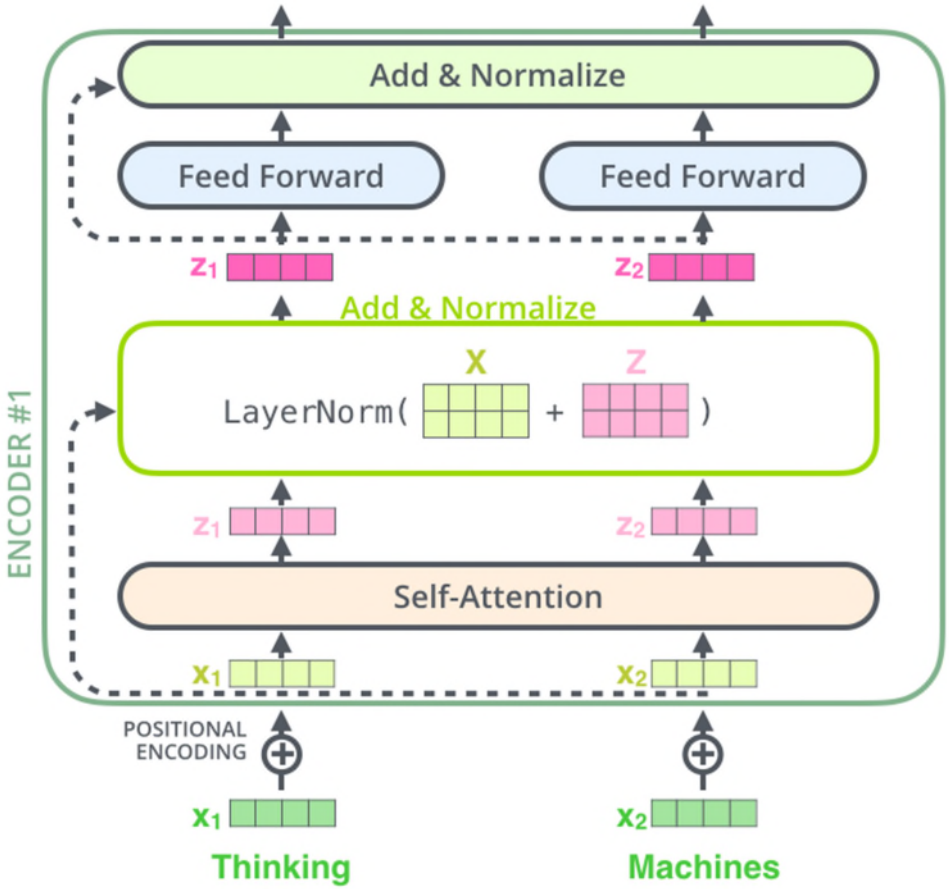
- Represent the position information of individual input tokens.
- Position encoding using sin / cos functions are often used.



Source: https://jalammar.github.io/illustrated-transformer/

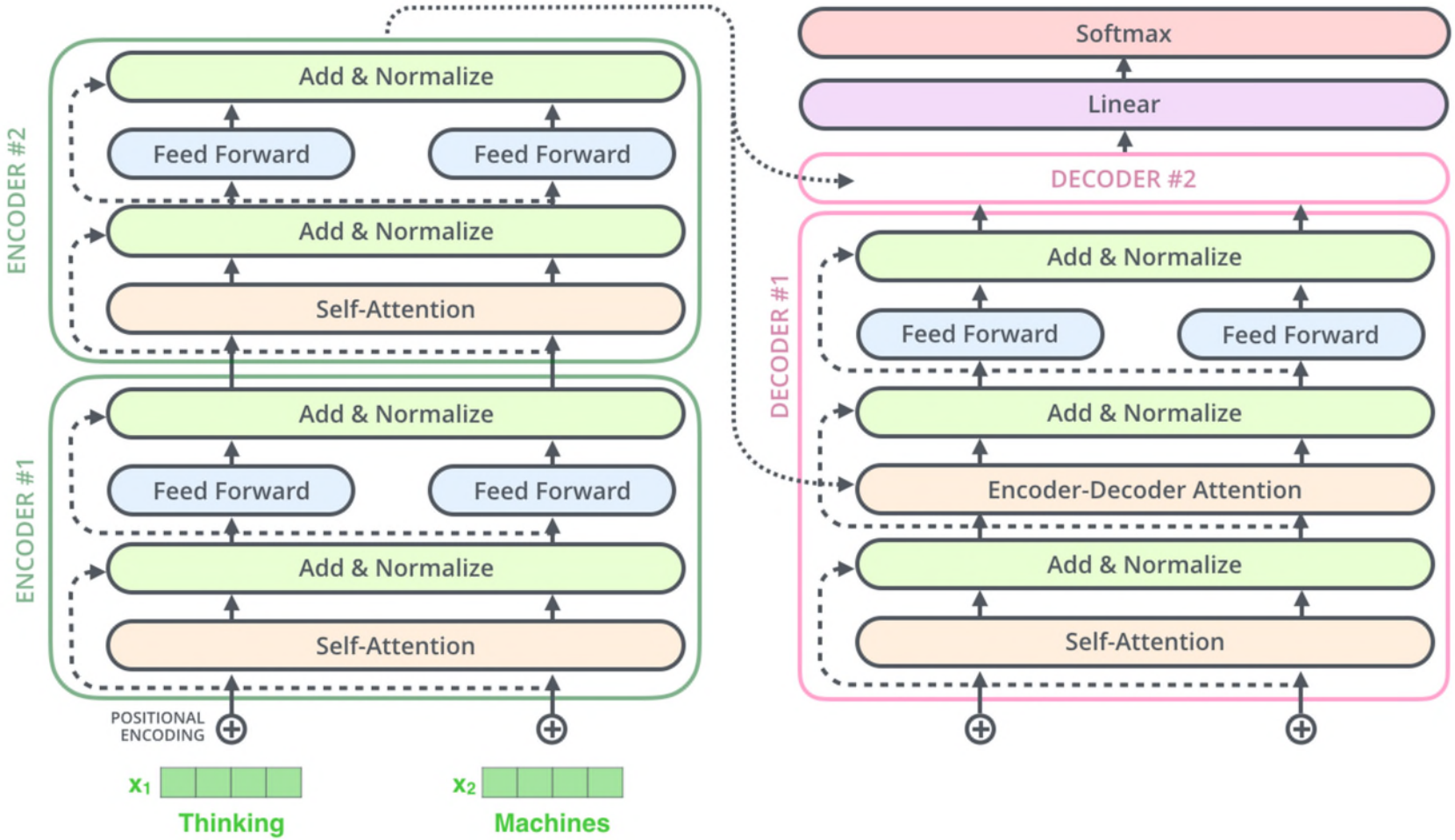
Residual Connection & Layer Norm

- Residual connection leverages the idea of residual learning in Resnet.
- Layer Norm is used to perform normalization.



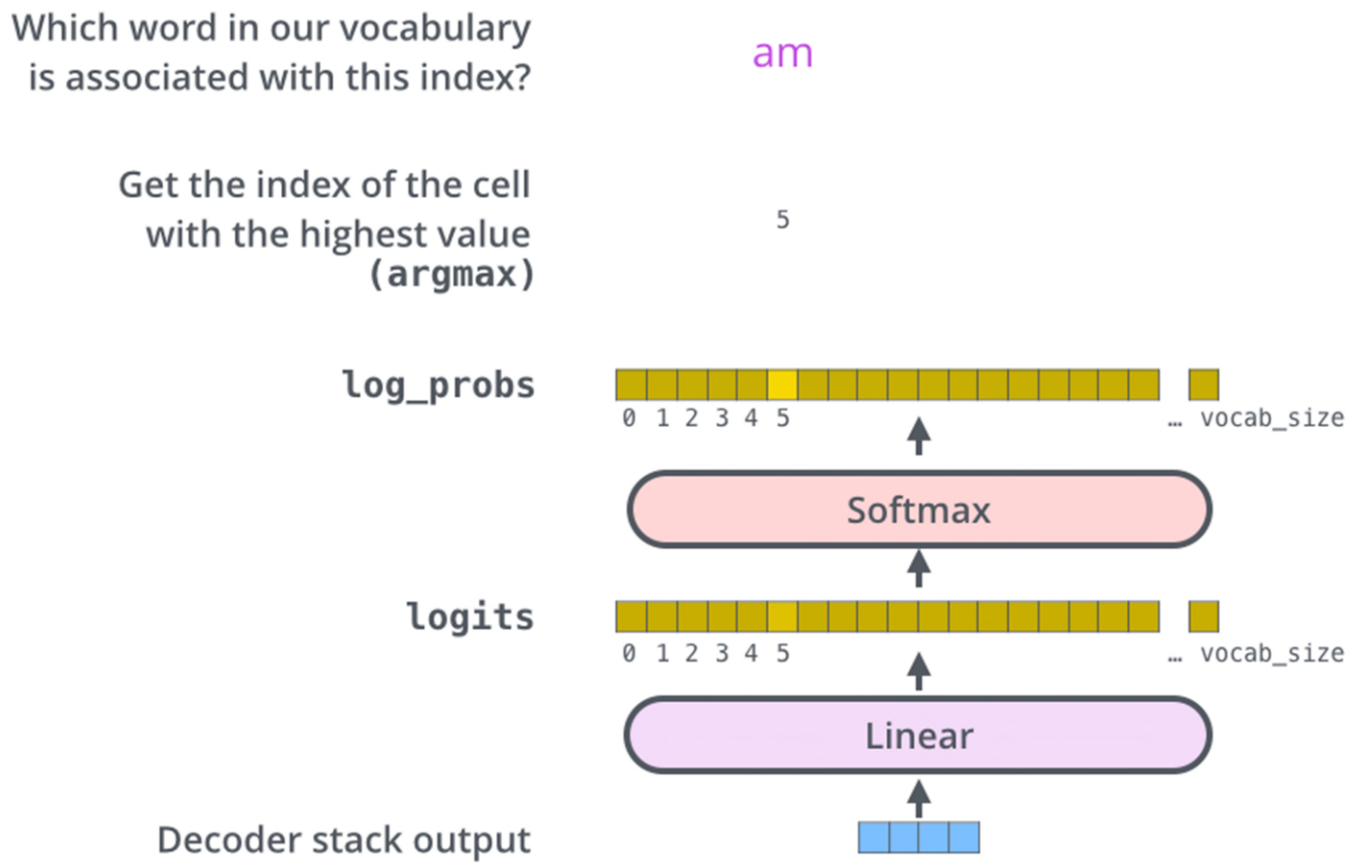
Source: https://jalammar.github.io/illustrated-transformer/

Transformer Encoder + Decoder Architecture



Source: https://jalammar.github.io/illustrated-transformer/

Final Linear and Softmax Layer



Source: https://jalammar.github.io/illustrated-transformer/

Summary

- The lecture covers the following topics:
  - Introduction
  - Recurrent Neural Network (RNN)
  - Long Short-Term Memory (LSTM)
  - Transformer