CENG501 – Deep Learning

Week 6

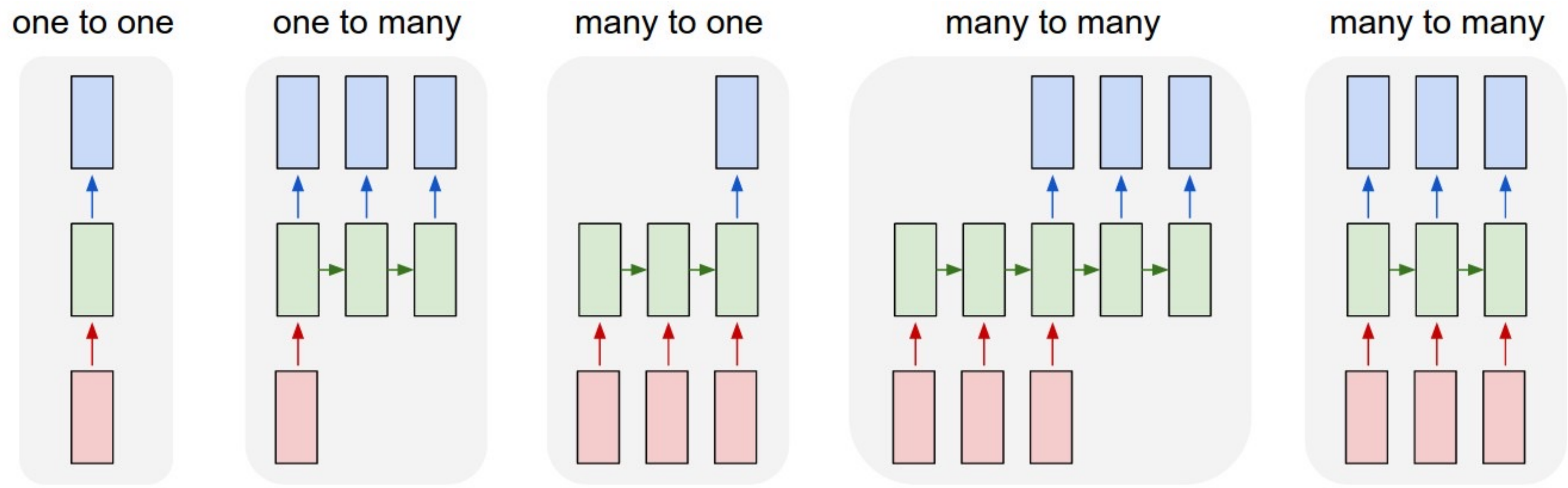
Spring 2026

Sinan Kalkan

Dept. of Computer Engineering, METU

Different types of sequence learning / recognition problems

Previously on CENG501

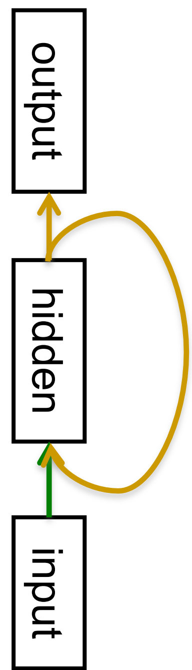


http://karpathy.github.io/2015/05/21/rnn-effectiveness/

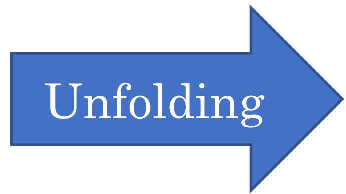
Previously on CENG501

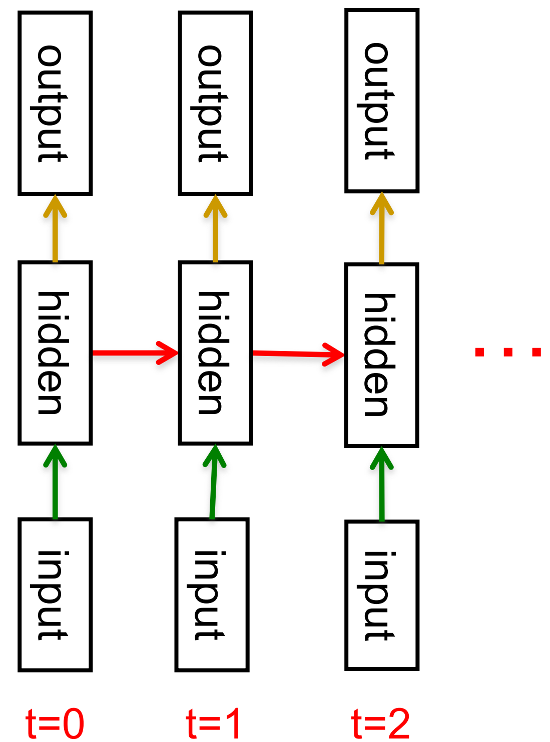
Unfolding

Feed-forward networks



Recurrent networks





time →

Feedforward through Vanilla RNN

Previously on CENG501

The Vanilla RNN Model

First time-step ($t = 1$):

$$\mathbf{h}_1 = tanh\left(W^{xh} \cdot \mathbf{x}_1 + W^{hh} \cdot \mathbf{h}_0\right)$$

$$\hat{\mathbf{y}}_1 = softmax\left(W^{hy} \cdot \mathbf{h}_1\right)$$

$$\mathcal{L}_1 = CE(\hat{\mathbf{y}}_1, \mathbf{y}_1)$$

In general:

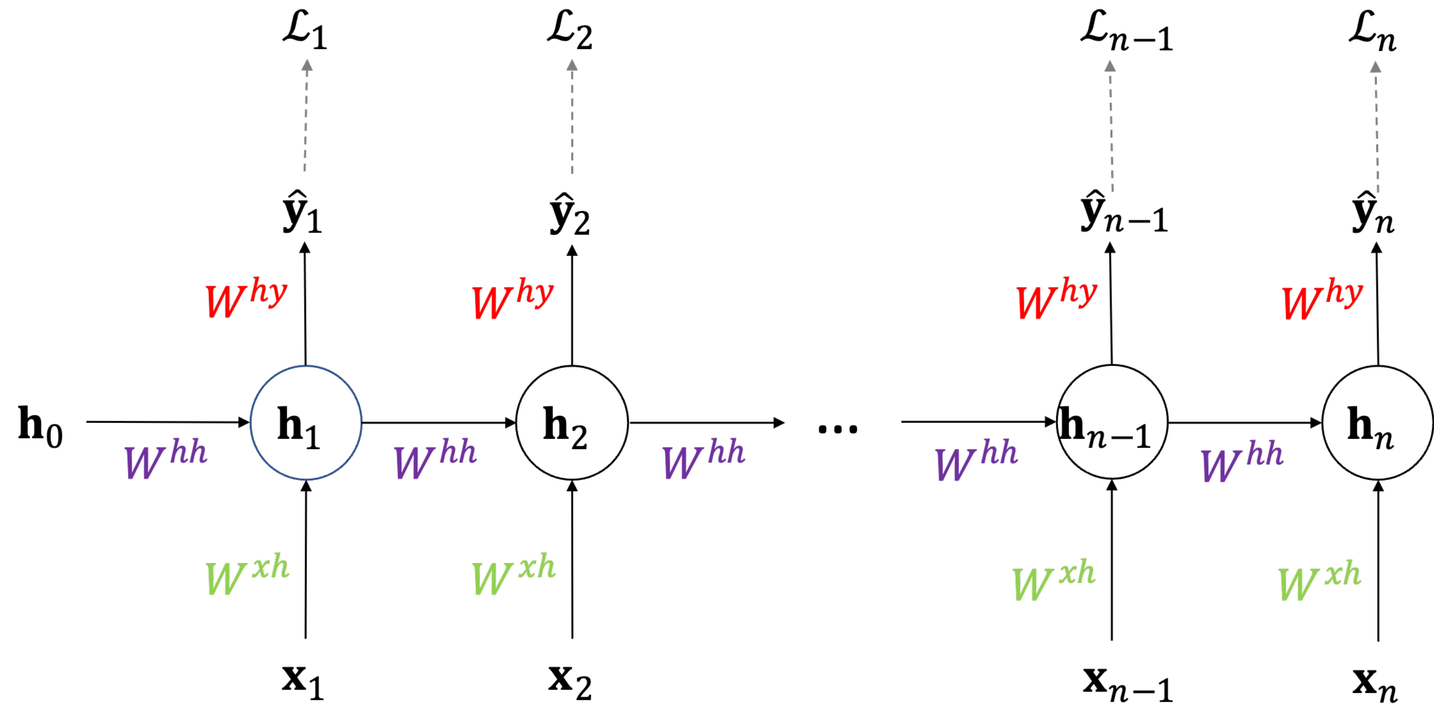
$$\mathbf{h}_t = tanh\left(W^{xh} \cdot \mathbf{x}_t + W^{hh} \cdot \mathbf{h}_{t-1}\right)$$

$$\hat{\mathbf{y}}_t = softmax\left(W^{hy} \cdot \mathbf{h}_t\right)$$

$$\mathcal{L}_t = CE(\hat{\mathbf{y}}_t, \mathbf{y}_t)$$

In total:

$$\mathcal{L} = \sum_t \mathcal{L}_t$$



Previously on CENG501

The problem of exploding or vanishing gradients

- What happens to the magnitude of the gradients as we backpropagate through many layers?
  - If the weights are small, the gradients shrink exponentially.
  - If the weights are big the gradients grow exponentially.
- Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.

- In an RNN trained on long sequences (*e.g.* 100 time steps) the gradients can easily explode or vanish.
  - We can avoid this by initializing the weights very carefully.
- Even with good initial weights, its very hard to detect that the current target output depends on an input from many time-steps ago.
  - So RNNs have difficulty dealing with long-range dependencies.

Slide: Hinton

LSTM in detail

Previously on CENG501

- We first compute an activation vector, a:
$$a = W_x x_t + W_h h_{t-1} + b$$
- Split this into four vectors of the same size:
$$a_i, a_f, a_o, a_g \leftarrow a$$
- We then compute the values of the gates:
$$i = \sigma(a_i) \quad f = \sigma(a_f) \quad o = \sigma(a_o) \quad g = \tanh(a_g)$$
where σ is the sigmoid.
- The next cell state c_t and the hidden state h_t:
$$c_t = f \odot c_{t-1} + i \odot g$$
$$h_t = o \odot \tanh(c_t)$$
where $\odot$ is the element-wise product of vectors

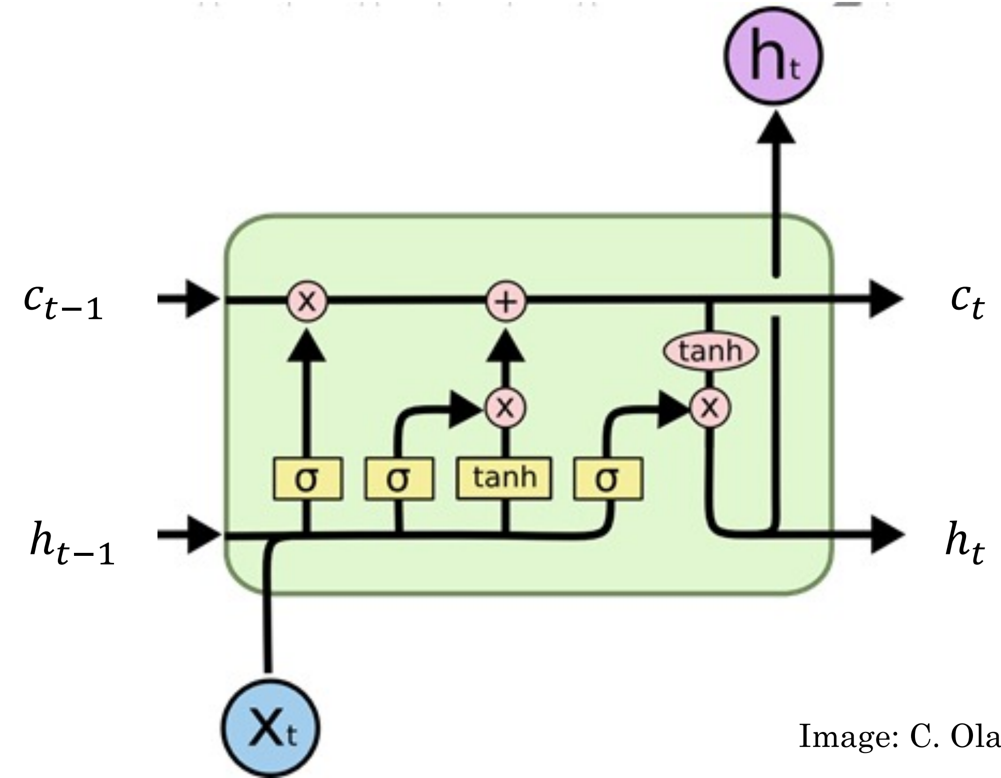


Image: C. Olah

Alternative formulation:

$$i_t = g(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$
$$f_t = g(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$
$$o_t = g(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

Eqs: Karpathy

Character-level Text Modeling

Previously on CENG501

- Problem definition: Find c_{n+1} given $c_1, c_2, \ldots, c_n$.
- Modelling:

$$p(c_{n+1} \mid c_n, \ldots, c_1)$$

- In general, we just take the last N characters:

$$p(c_{n+1} \mid c_n, \ldots, c_{n-(N-1)})$$

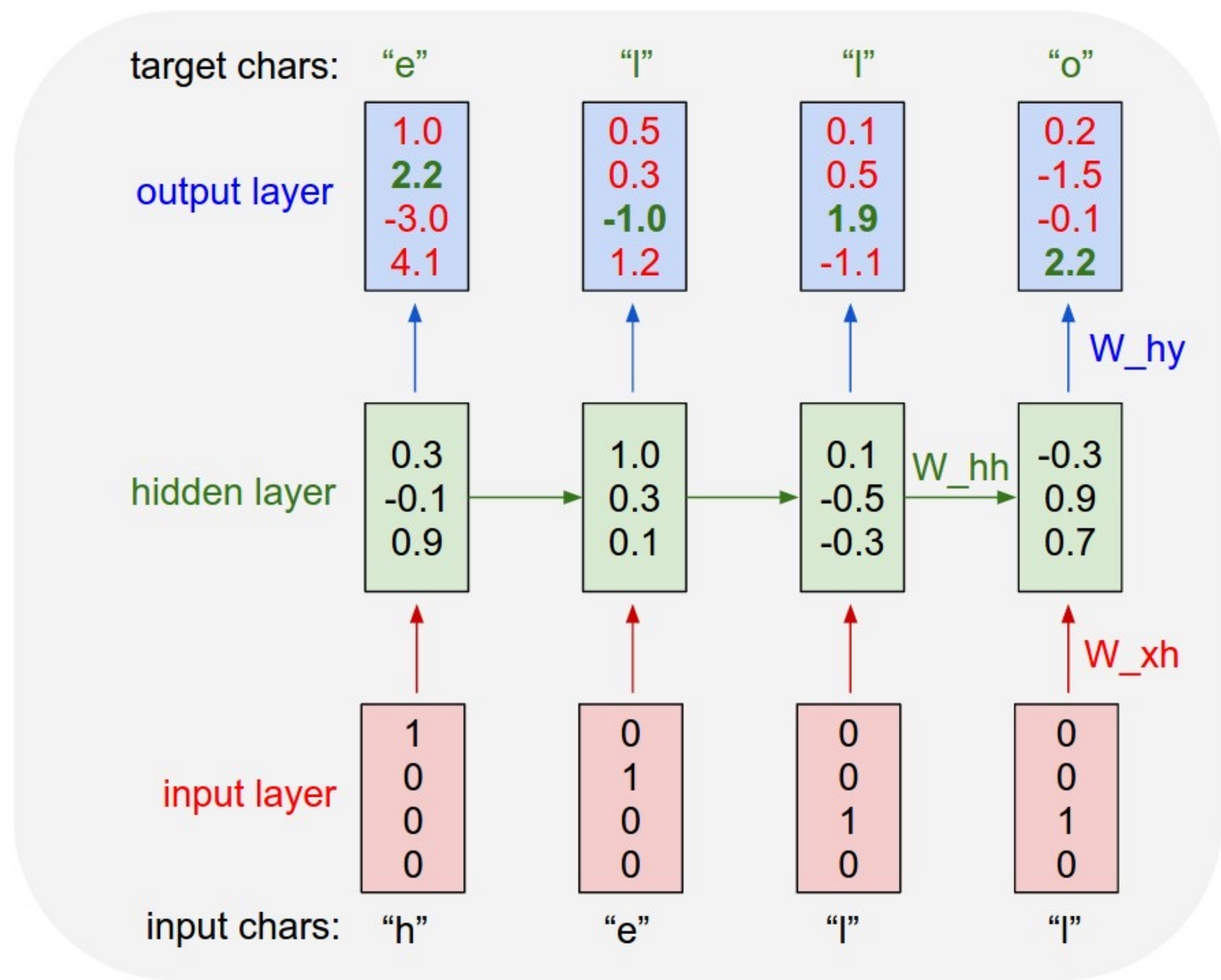
- Learn $p(c_{n+1} = 'a' \mid 'Ankar')$ from data such that

$$p(c_{n+1} = 'a' \mid 'Ankar') > p(c_{n+1} = 'o' \mid 'Ankar')$$

A simple scenario

Previously on CENG501

- Alphabet: h, e, l, o
- Text to train to predict: "hello"



http://karpathy.github.io/2015/05/21/rnn-effectiveness/

Previously on CENG501

Word-level Text Modeling

- Problem definition: Find ω_{n+1} given $\omega_1, \omega_2, \ldots, \omega_n$.
- Modelling:

$$p(\omega_{n+1} \mid \omega_n, \ldots, \omega_1)$$

- In general, we just take the last N words:

$$p(\omega_{n+1} \mid \omega_n, \ldots, \omega_{n-(N-1)})$$

- Learn $p(\omega_{n+1} = 'Turkey' \mid 'Ankara\ is\ the\ capital\ of\ ')$ from data such that:

$$p(\omega_{n+1} = 'Turkey' \mid 'Ankara\ is\ the\ capital\ of\ ') \quad > \quad p(\omega_{n+1} = 'UK' \mid 'Ankara\ is\ the\ capital\ of\ ')$$

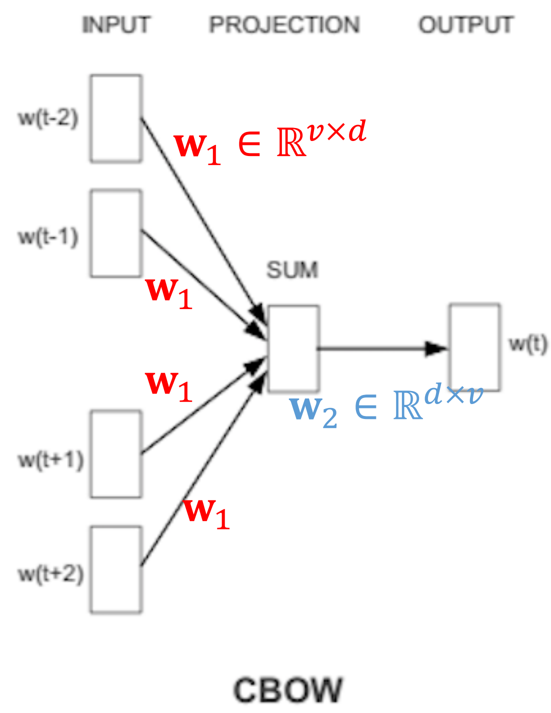
Previously on CENG501

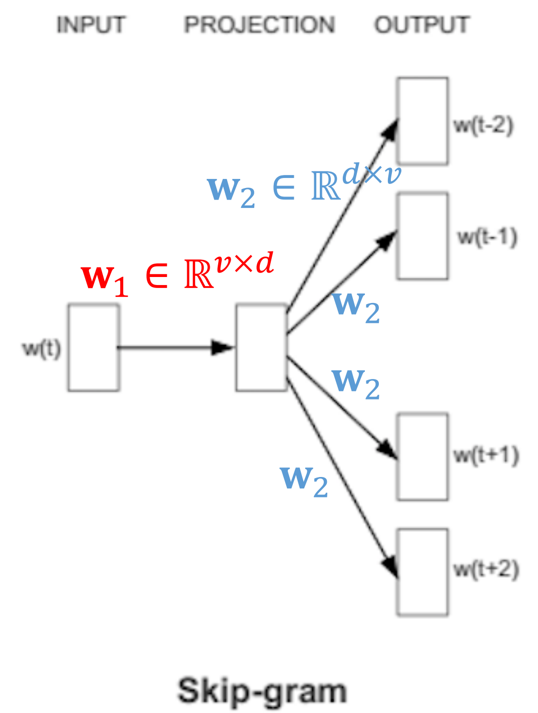
Two different ways to train

1. Using context to predict a target word (~ continuous bag-of-words)
2. Using word to predict a target context (skip-gram)

- If the vector for a word cannot predict the context, the mapping to the vector space is adjusted
- Since similar words should predict the same or similar contexts, their vector representations should end up being similar

v: vocabulary size
d: hidden dimension





http://deeplearning4j.org/word2vec

Note that the weight matrix is a look-up table

Previously on CENG501

- In both approaches, the weight matrix is used as follows:

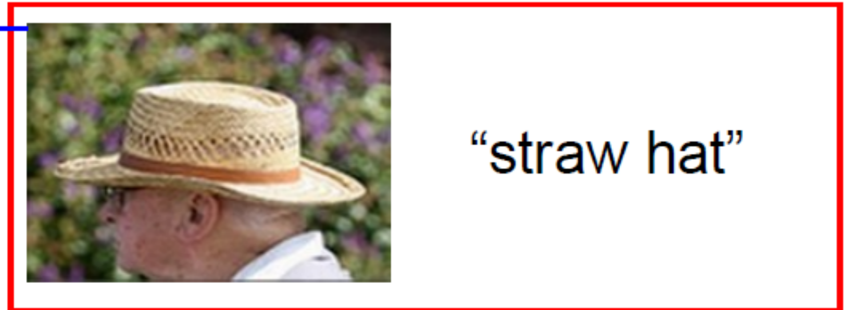
$$\underbrace{\begin{bmatrix} 0 & 1 & 0 \end{bmatrix}}_{\text{input } 1\times V} \underbrace{\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \end{bmatrix}}_{W_1\ V\times N} = \underbrace{\begin{bmatrix} e & f & g & h \end{bmatrix}}_{\text{hidden } 1\times N}$$

W_1

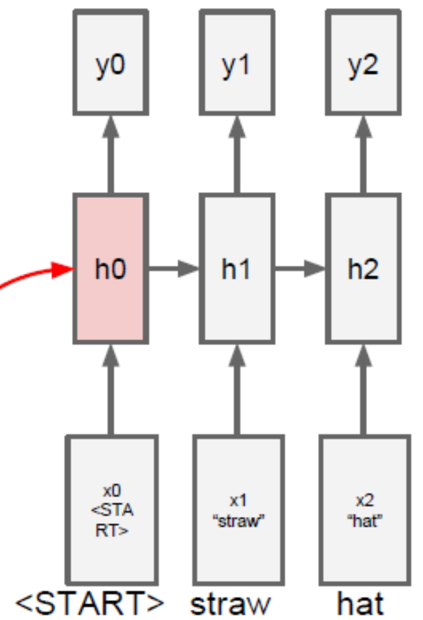
https://medium.com/@zafaralibagh6/a-simple-word2vec-tutorial-61e64e38a6a1

Training

Previously on CENG501



training example



before:

h0 = max(0, Wxh * x0)

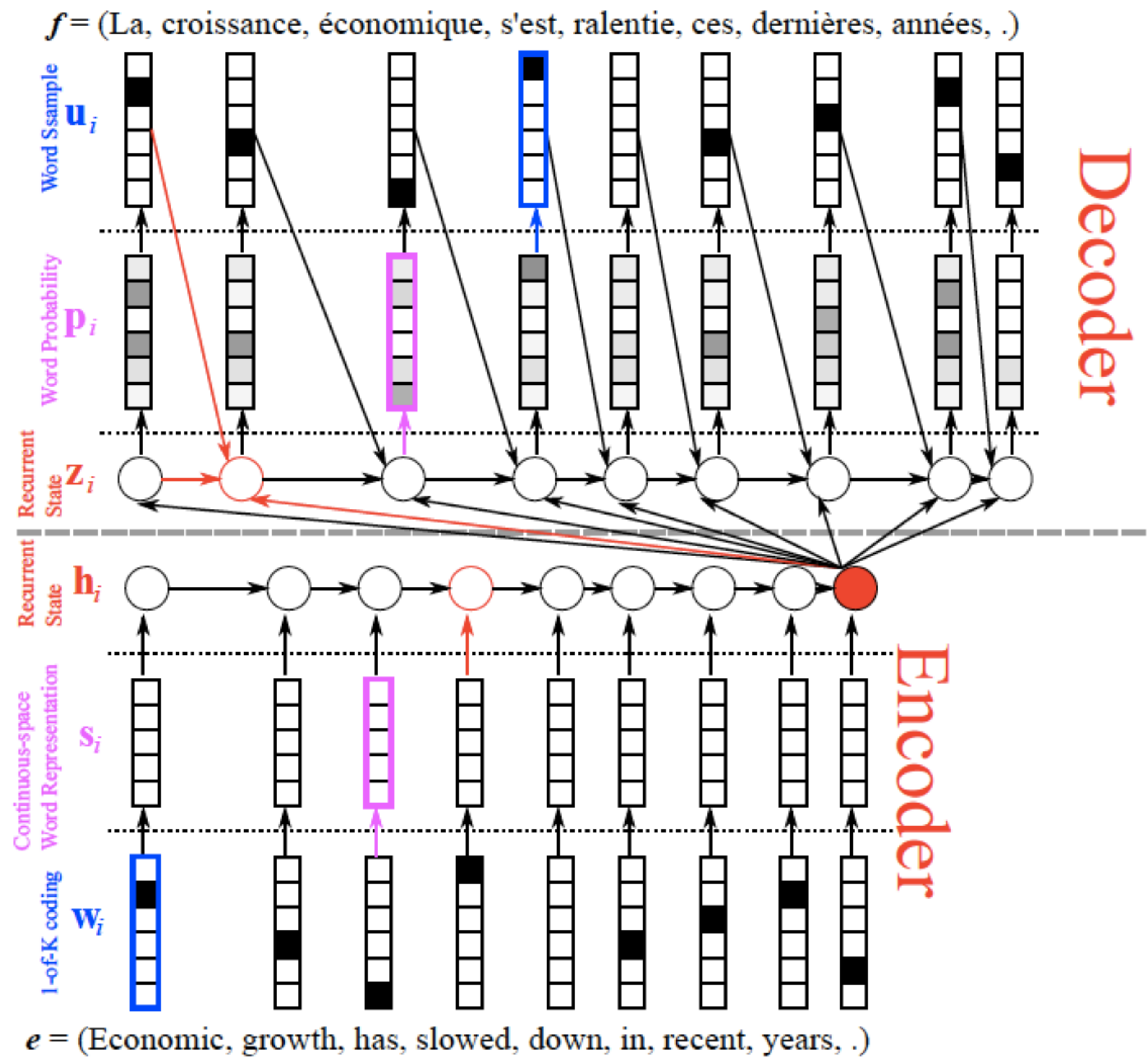
now:

h0 = max(0, Wxh * x0 + Wih * v)

Slide: Karpathy

Neural Machine Translation

Previously on CENG501

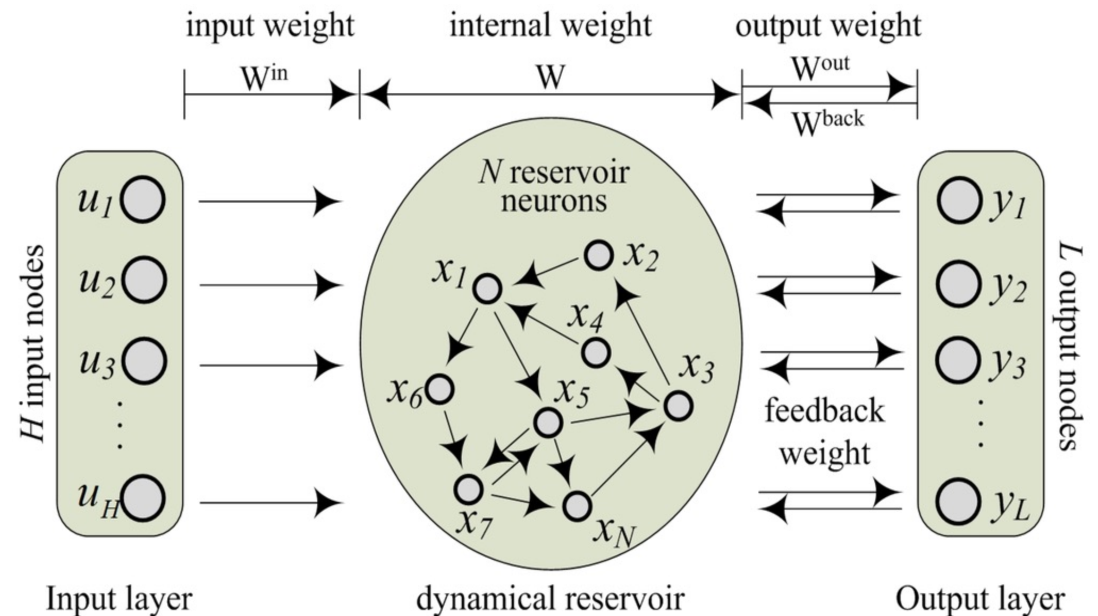


Cho: From Sequence Modeling to Translation

Previously on CENG501

Echo State Networks (ESN)

- Reservoir of a set of neurons
  - Randomly initialized and fixed
  - Run input sequence through the network and keep the activations of the reservoir neurons
  - Calculate the "readout" weights using linear regression.
- Has the benefits of recurrent connections/networks
- No problem of vanishing gradient



Li et al., 2015.

Today

- Attention
- Self-attention
- Transformer
- State-space Models
- Mamba

Administrative Notes

- Paper Selection
- Project next steps:
  - GitHub repository shared. **Accept the invitations as soon as possible.**
  - Milestones:
    1. Milestone (April 10, midnight):
       - Read & understand the paper
       - Download the datasets
       - Prepare the Readme file excluding the results & conclusion
    2. Milestone (May 4, midnight)
       - The results of the first experiment
    3. Milestone (June 1, midnight)
       - Final report (Readme file)
       - Repo with all code & trained models

Attention

BLEU: Bilingual Evaluation Understudy

https://cloud.google.com/translate/automl/docs/evaluate#bleu

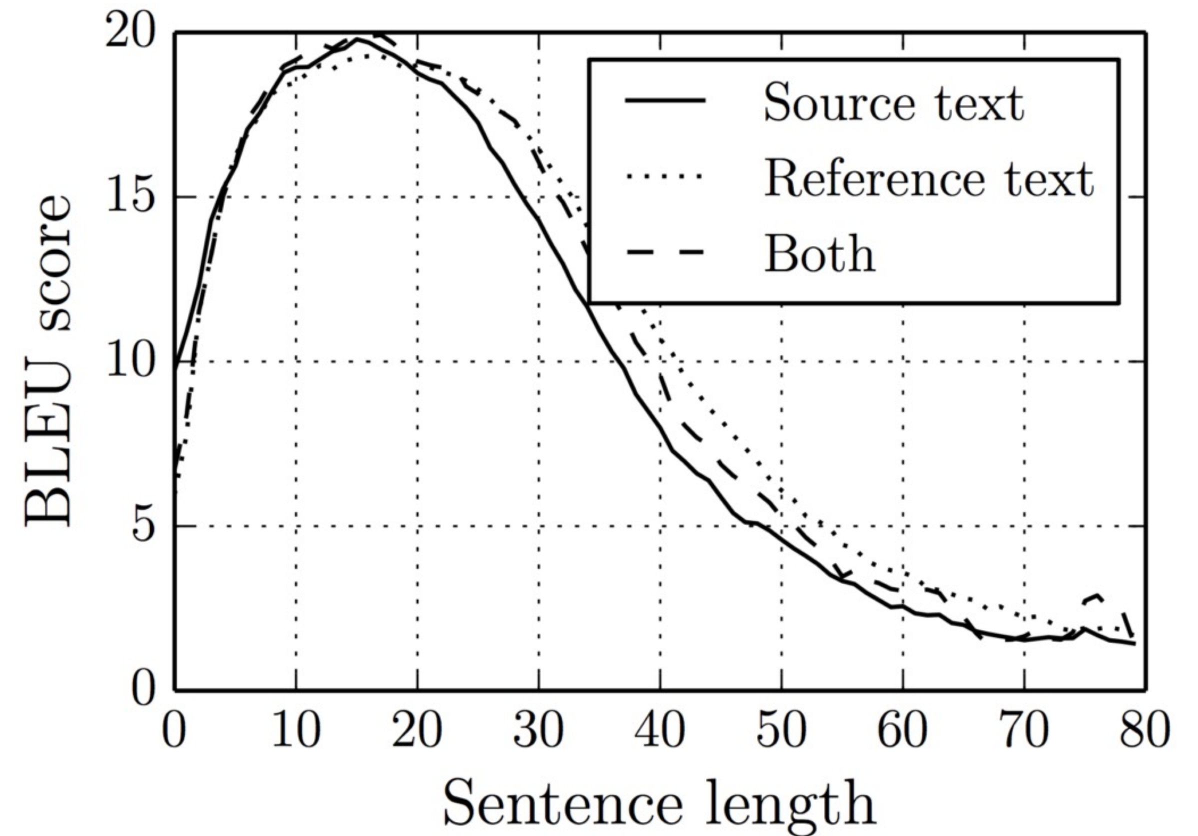
Attention

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal



f = (La, croissance, économique, s'est, ralentie, ces, dernières, années, .)

Word Ssample $\mathbf{u}_i$

Recurrent State $\mathbf{z}_i$

Attention Mechanism a_j

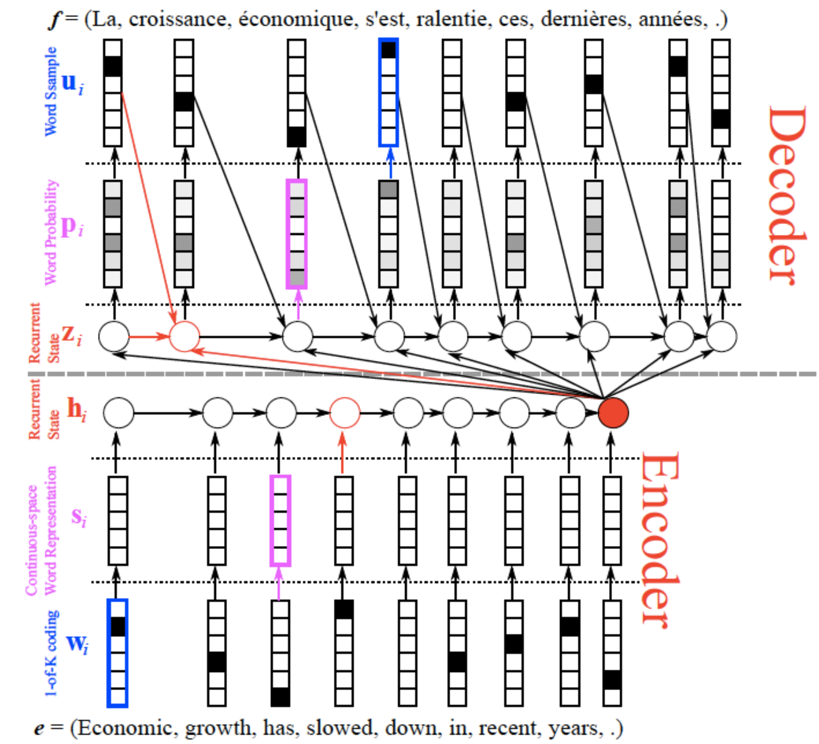
Attention weight

$\Sigma a_j = 1$

(2)

Annotation Vectors $\mathbf{h}_j$

e = (Economic, growth, has, slowed, down, in, recent, years, .)



Attention

Published as a conference paper at ICLR 2015

NEURAL MACHINE TRANSLATION
BY JOINTLY LEARNING TO ALIGN AND TRANSLATE

Dzmitry Bahdanau
Jacobs University Bremen, Germany

KyungHyun Cho **Yoshua Bengio***
Université de Montréal

In a new model architecture, we define each conditional probability in Eq. (2) as:

$$p(y_i|y_1, \ldots, y_{i-1}, \mathbf{x}) = g(y_{i-1}, s_i, c_i), \quad (4)$$

where s_i is an RNN hidden state for time i, computed by

$$s_i = f(s_{i-1}, y_{i-1}, c_i).$$

It should be noted that unlike the existing encoder–decoder approach (see Eq. (2)), here the probability is conditioned on a distinct context vector c_i for each target word y_i.

The context vector c_i depends on a sequence of *annotations* $(h_1, \cdots, h_{T_x})$ to which an encoder maps the input sentence. Each annotation h_i contains information about the whole input sequence with a strong focus on the parts surrounding the i-th word of the input sequence. We explain in detail how the annotations are computed in the next section.

The context vector c_i is, then, computed as a weighted sum of these annotations h_i:

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j. \quad (5)$$

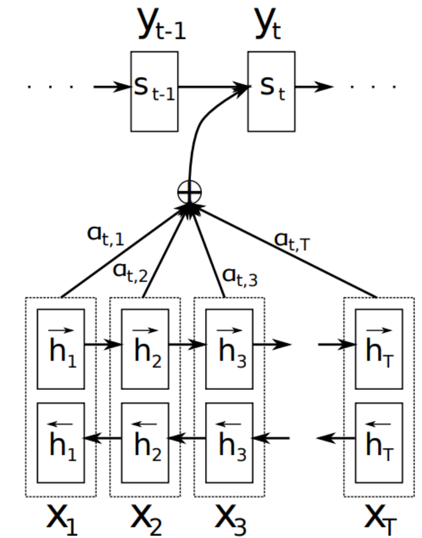


Figure 1: The graphical illustration of the proposed model trying to generate the t-th target word y_t given a source sentence $(x_1, x_2, \ldots, x_T)$.

The weight α_{ij} of each annotation h_j is computed by

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{T_x} \exp(e_{ik})}, \quad (6)$$

where

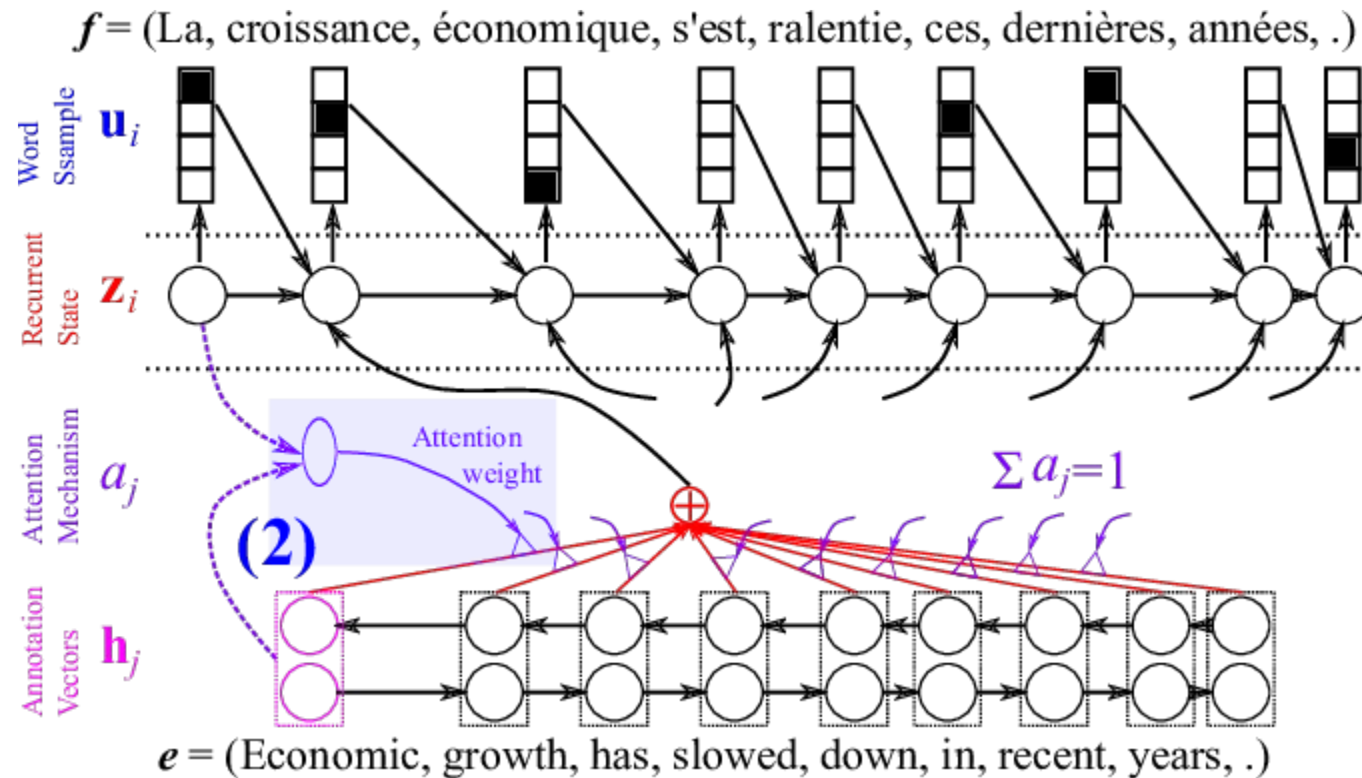
$$e_{ij} = a(s_{i-1}, h_j)$$

is an *alignment model* which scores how well the inputs around position j and the output at position i match. The score is based on the RNN hidden state s_{i-1} (just before emitting y_i, Eq. (4)) and the j-th annotation h_j of the input sentence.

We parametrize the alignment model a as a feedforward neural network which is jointly trained with all the other components of the proposed system. Note that unlike in traditional machine translation,

CENG502

Attention



Attention mechanism: A two-layer neural network.

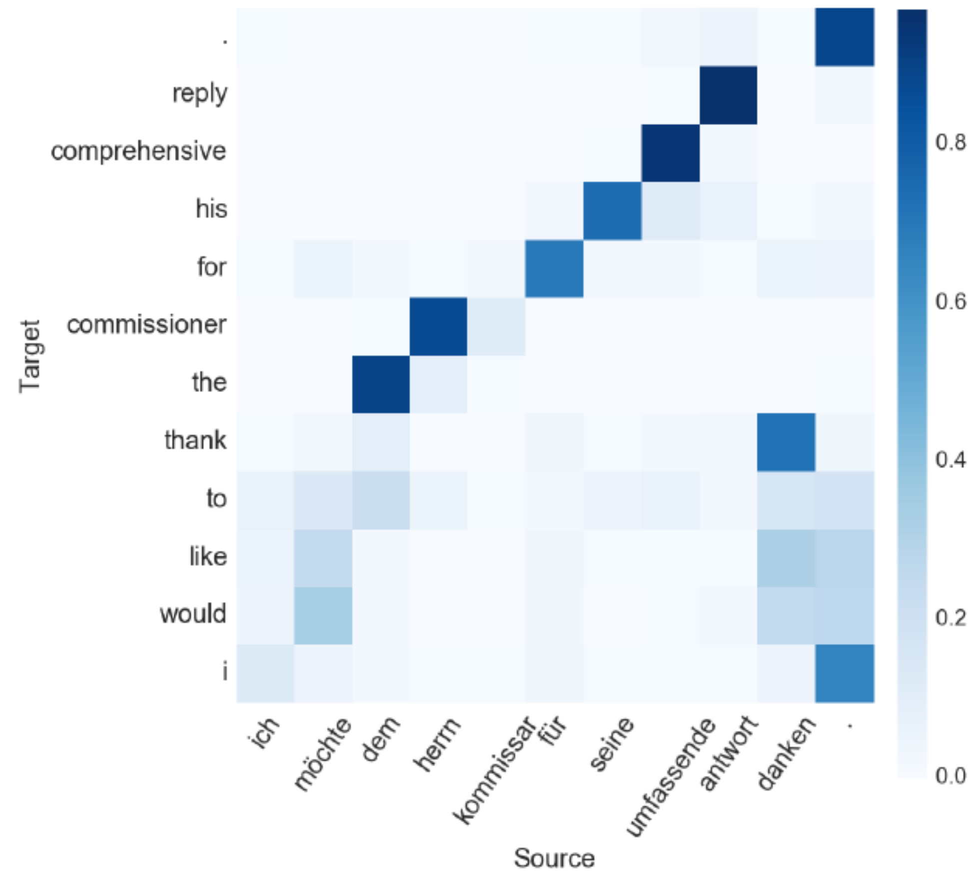
Input: z_i and h_j

Output: e_j, a scalar for the importance of word j.

The scores of words are normalized: $a_j = \text{softmax}(\text{e}_\text{j})$

https://devblogs.nvidia.com/introduction-neural-machine-translation-gpus-part-3/

Attention



What does Attention in Neural Machine Translation Pay Attention to?

Hamidreza Ghader and **Christof Monz**
Informatics Institute, University of Amsterdam, The Netherlands
h.ghader, c.monz@uva.nl

2017

Attention Types

- Let's rewrite Bahdanau et al.'s attention model:

$$\mathbf{c}_t = \sum_{i=1}^{n} \alpha_{t,i} \boldsymbol{h}_i \qquad \text{; Context vector for output } y_t$$

$$\alpha_{t,i} = \text{align}(y_t, x_i) \qquad \text{; How well two words } y_t \text{ and } x_i \text{ are aligned.}$$

$$= \frac{\exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_i))}{\sum_{i'=1}^{n} \exp(\text{score}(\boldsymbol{s}_{t-1}, \boldsymbol{h}_{i'}))} \qquad \text{; Softmax of some predefined alignment score..}$$

$$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$$

where both $\mathbf{v}_a$ and $\mathbf{W}_a$ are weight matrices to be learned in the alignment model.

Eqs: https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Attention Types

Name	Alignment score function	Citation
Content-base attention	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \text{cosine}[\boldsymbol{s}_t, \boldsymbol{h}_i]$	Graves2014
Additive(*)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \mathbf{v}_a^\top \tanh(\mathbf{W}_a[\boldsymbol{s}_t; \boldsymbol{h}_i])$	Bahdanau2015
Location-Base	$\alpha_{t,i} = \text{softmax}(\mathbf{W}_a \boldsymbol{s}_t)$ Note: This simplifies the softmax alignment to only depend on the target position.	Luong2015
General	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \mathbf{W}_a \boldsymbol{h}_i$ where $\mathbf{W}_a$ is a trainable weight matrix in the attention layer.	Luong2015
Dot-Product	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \boldsymbol{s}_t^\top \boldsymbol{h}_i$	Luong2015
Scaled Dot-Product(^)	$\text{score}(\boldsymbol{s}_t, \boldsymbol{h}_i) = \frac{\boldsymbol{s}_t^\top \boldsymbol{h}_i}{\sqrt{n}}$ Note: very similar to the dot-product attention except for a scaling factor; where n is the dimension of the source hidden state.	Vaswani2017

(*) Referred to as "concat" in Luong, et al., 2015 and as "additive attention" in Vaswani, et al., 2017.
(^) It adds a scaling factor $1/\sqrt{n}$, motivated by the concern when the input is large, the softmax function may have an extremely small gradient, hard for efficient learning.

Table: https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Vanilla Self-attention

$$e_i' = \sum_j \frac{\exp\left(e_j^T e_i\right)}{\sum_m \exp(e_m^T e_i)} e_j$$

Attention: Transformer

- Vanilla self attention:

$$e_i{}' = \sum_j \frac{\exp\left(e_j^T e_i\right)}{\sum_m \exp\left(e_m^T e_i\right)} e_j$$

- Scaled-dot product attention:

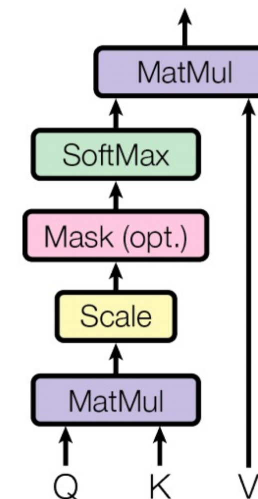
$$e_i{}' = \sum_j \frac{\exp\left(k\left(e_j^T\right) q\left(e_i\right)\right)}{\sum_m \exp\left(k\left(e_m^T\right) q\left(e_i\right)\right)} v\left(e_j\right)$$

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

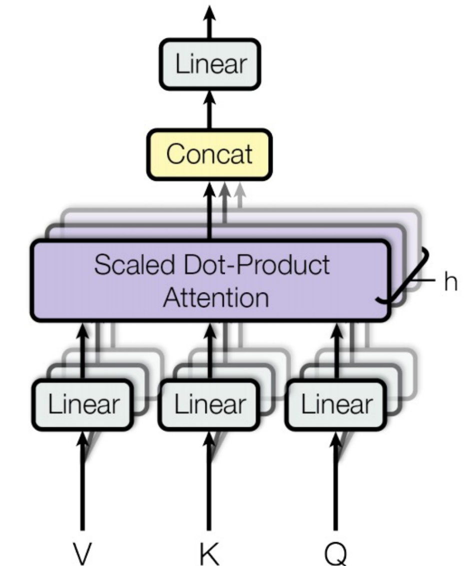
Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com
Noam Shazeer* Google Brain noam@google.com
Niki Parmar* Google Research nikip@google.com
Jakob Uszkoreit* Google Research usz@google.com
Llion Jones* Google Research llion@google.com
Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu
Łukasz Kaiser* Google Brain lukaszkaiser@google.com
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Scaled Dot-Product Attention



Multi-Head Attention



Multi-head Attention

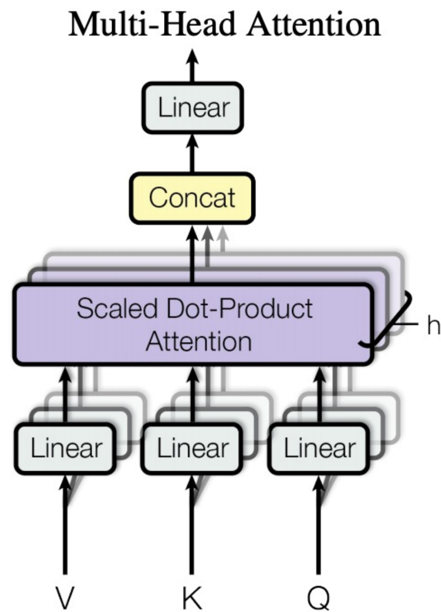


Fig: Attention is all you need.

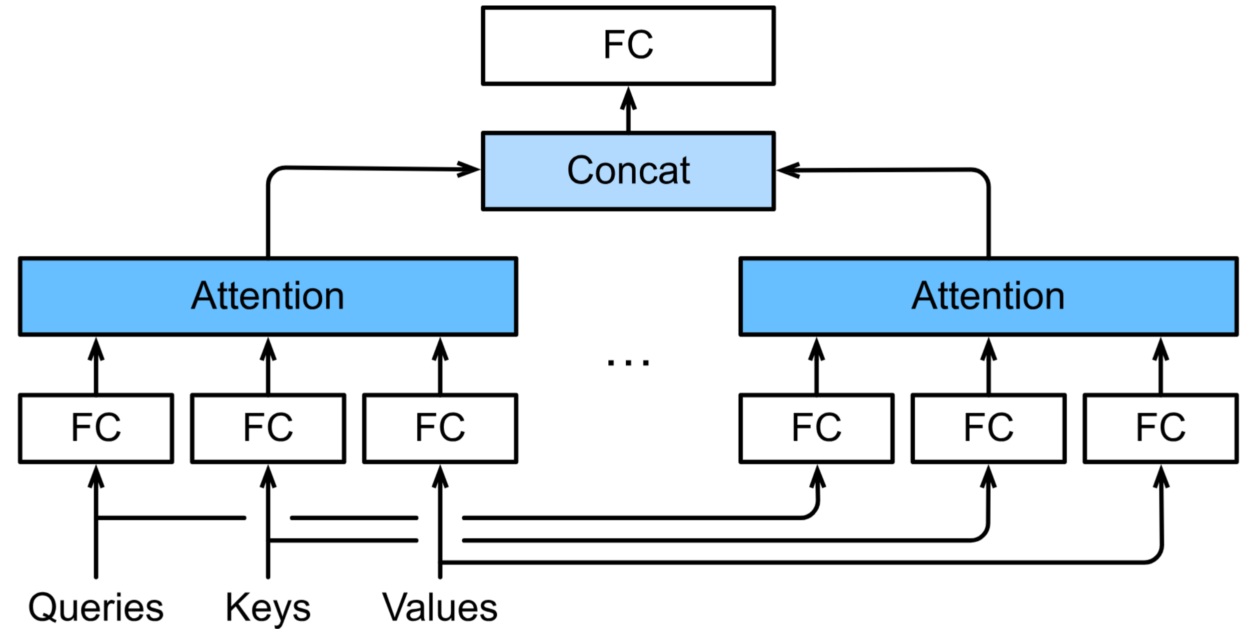


Fig: https://d2l.ai/chapter_attention-mechanisms-and-transformers/multihead-attention.html

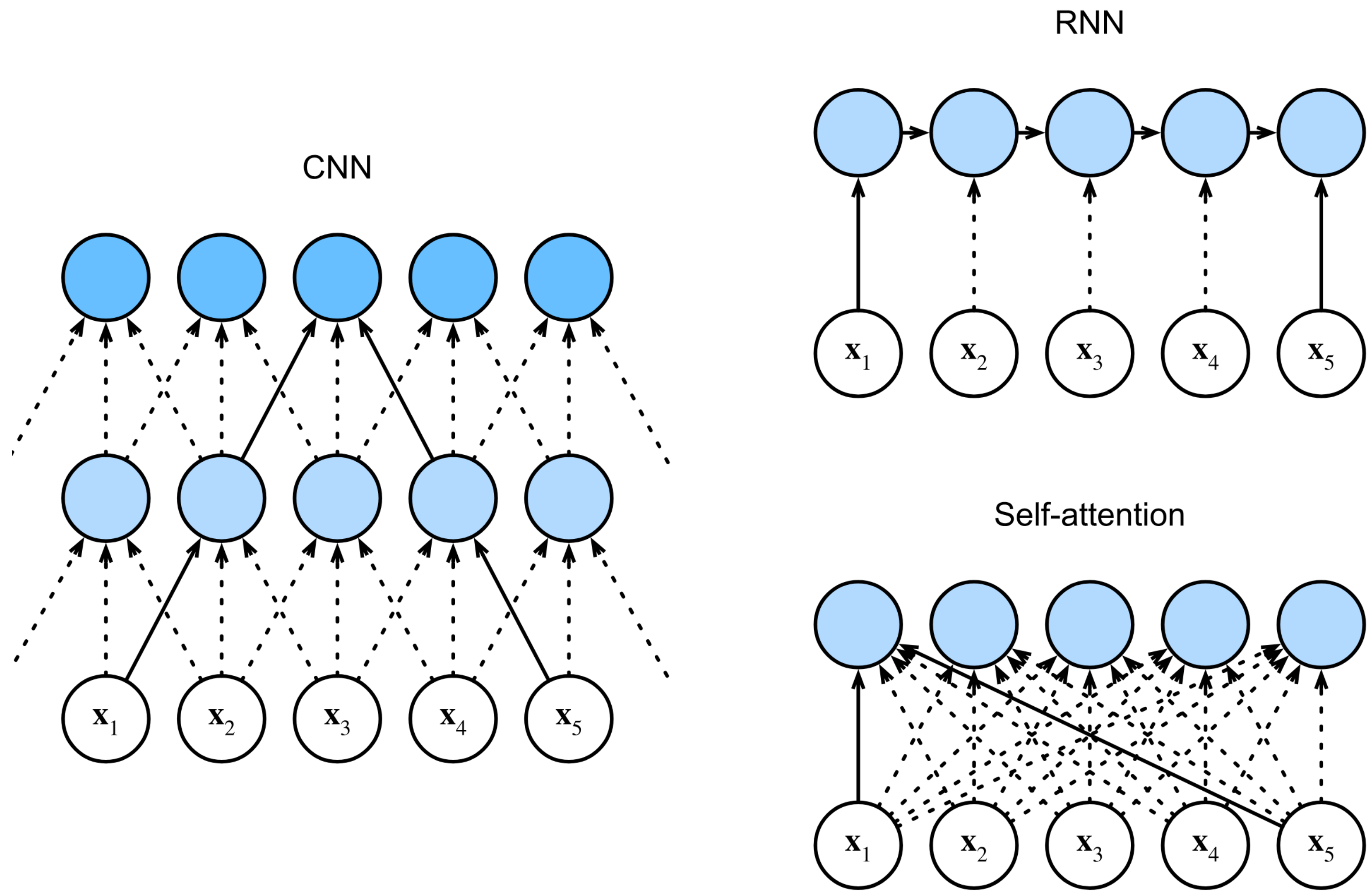
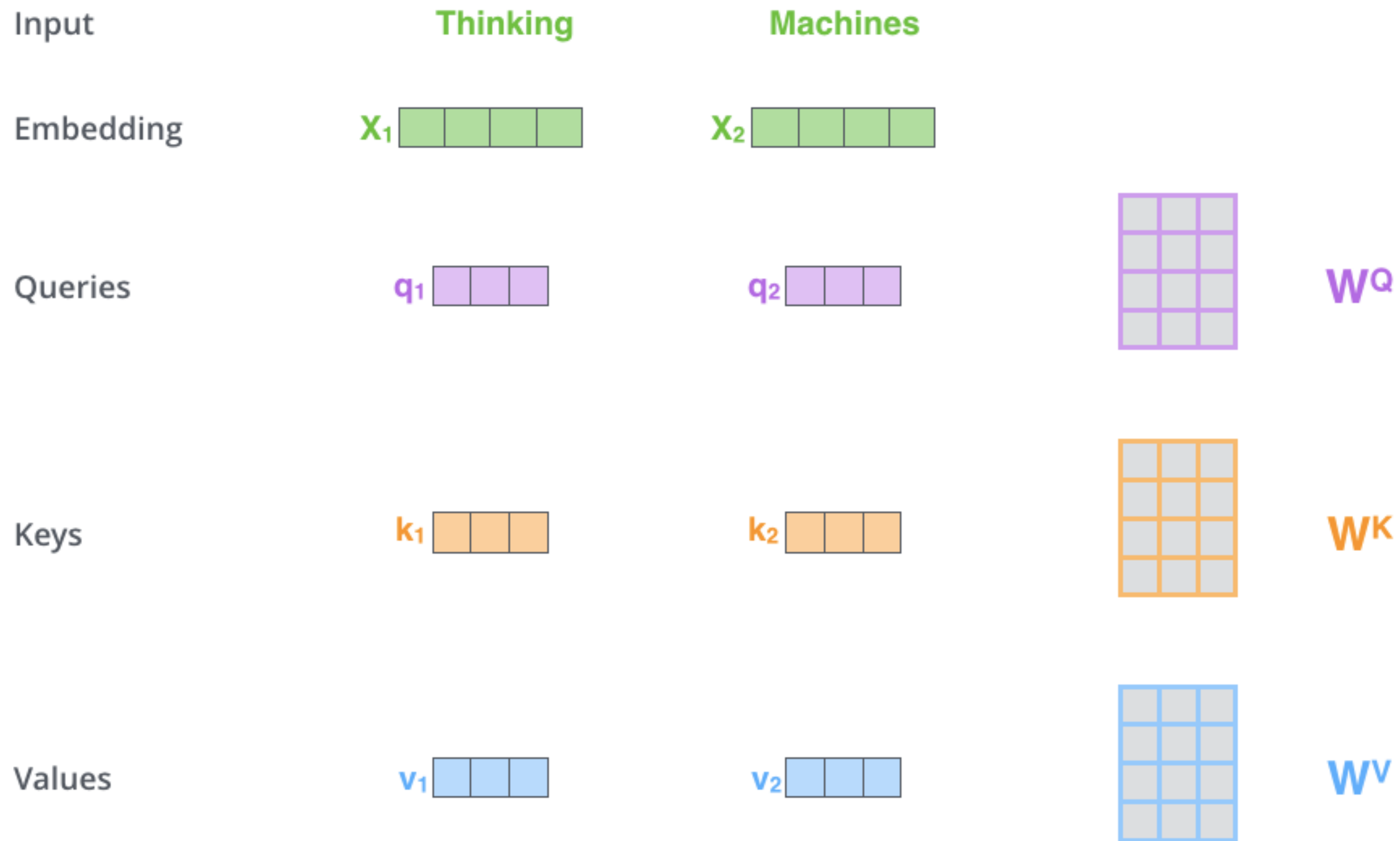
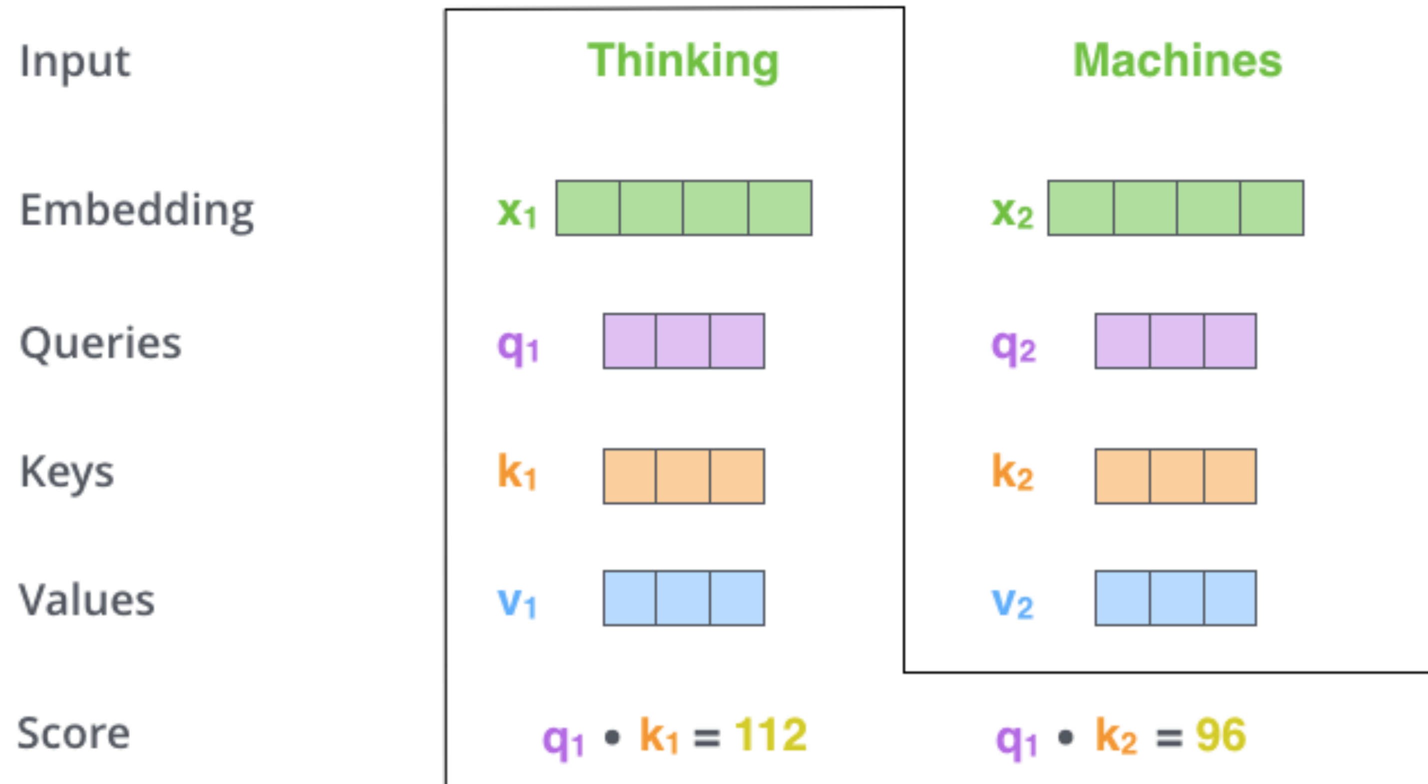


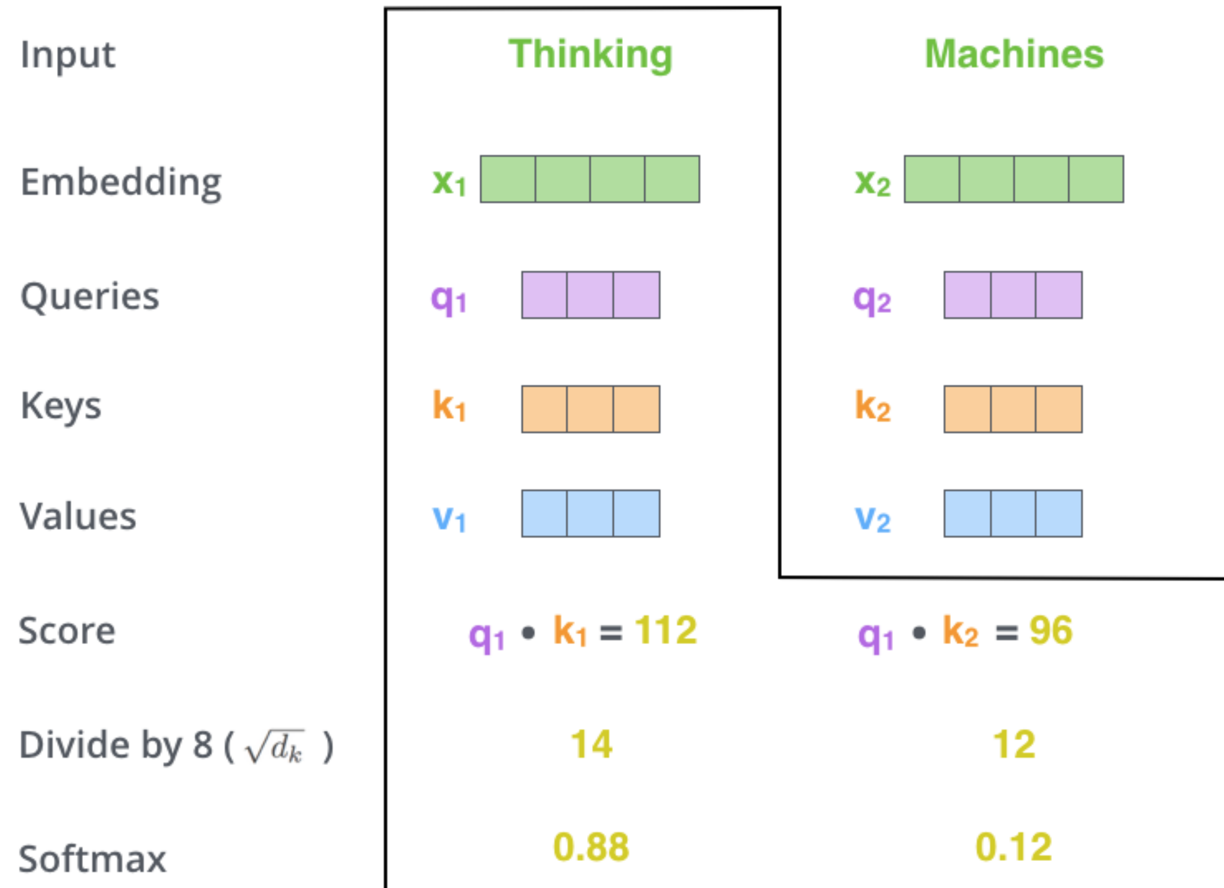
Fig: https://d2l.ai/chapter_attention-mechanisms-and-transformers/self-attention-and-positional-encoding.html



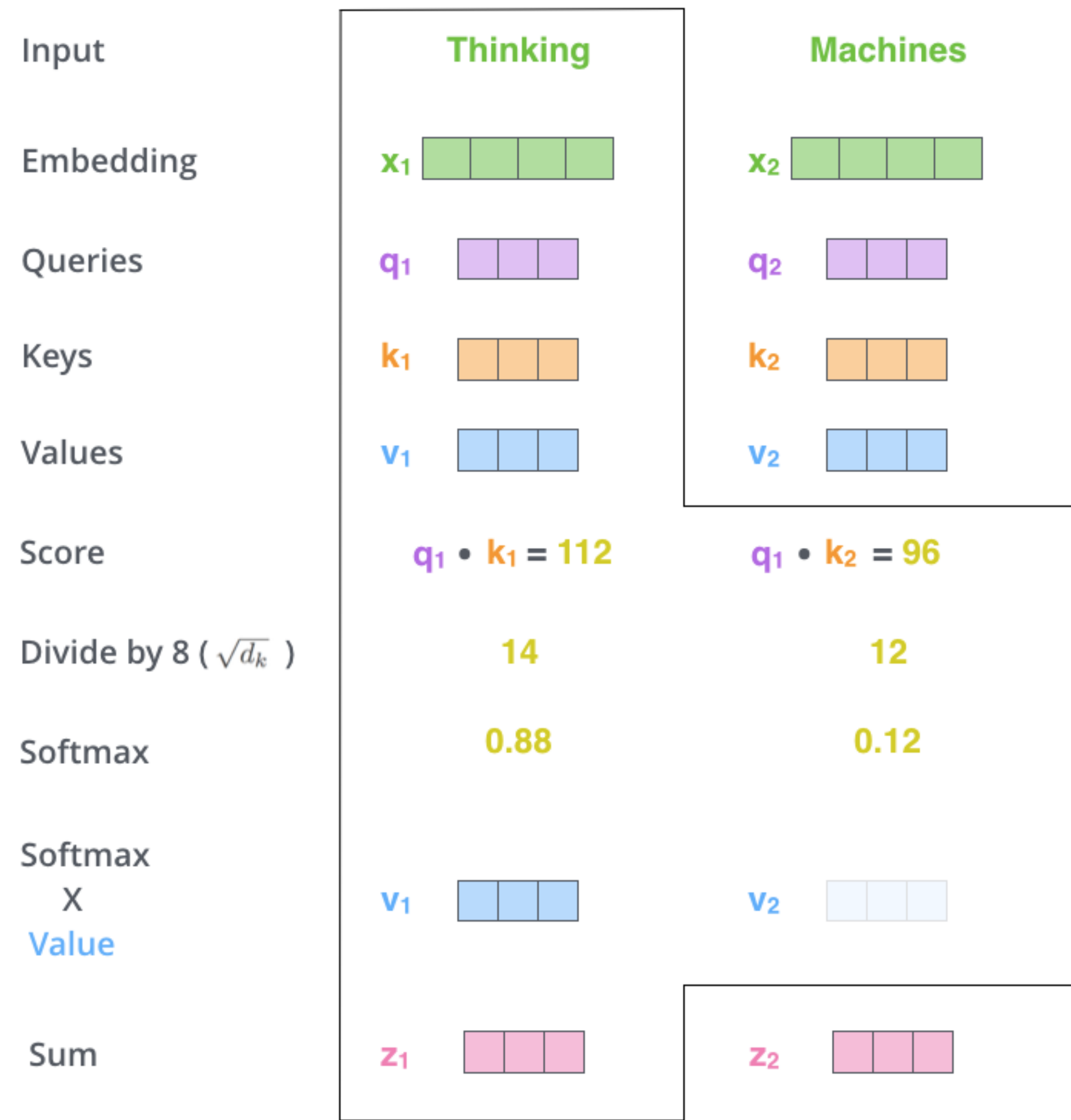
https://jalammar.github.io/illustrated-transformer/



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

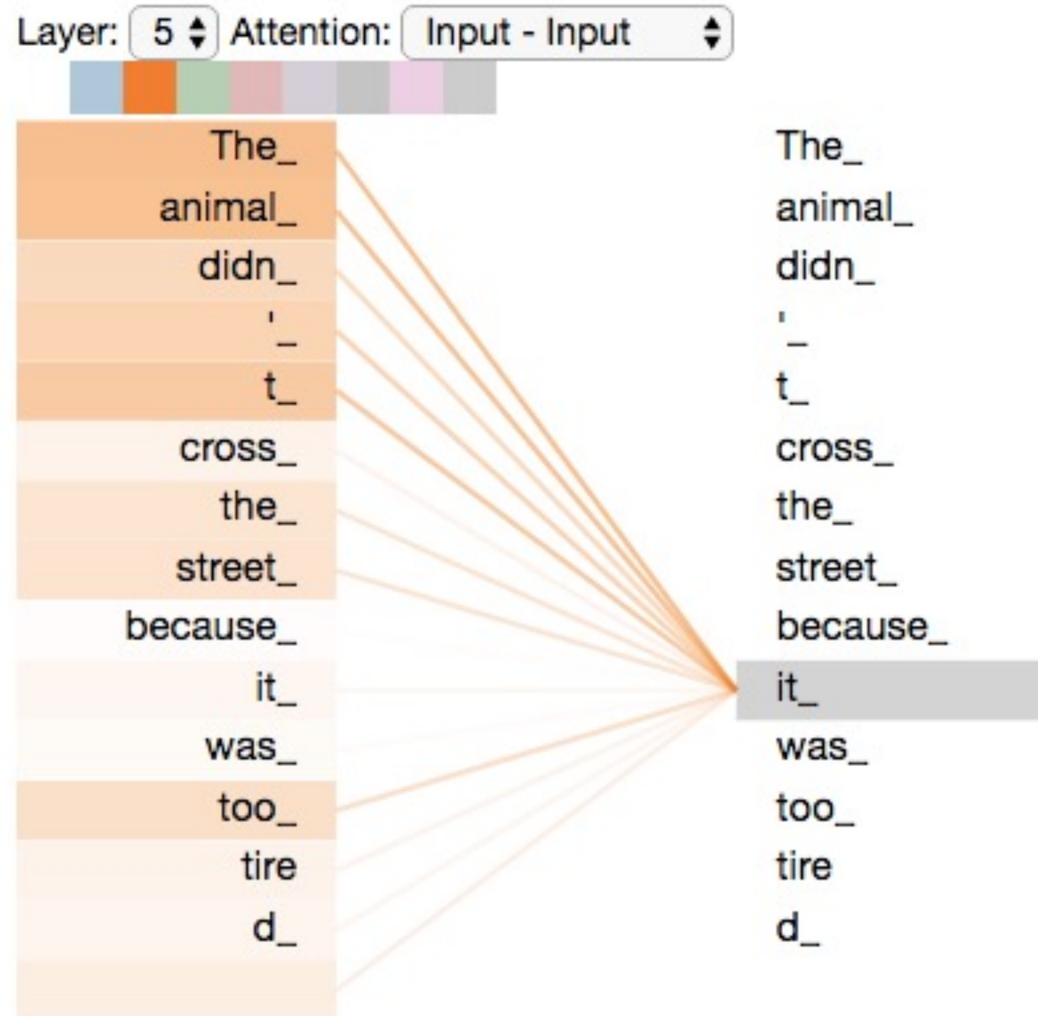


https://jalammar.github.io/illustrated-transformer/



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

```python
import numpy as np

# 1. Our "Sentence" (3 words, each has a 2-number meaning)
# Word 1: [1, 0], Word 2: [0, 1], Word 3: [1, 1]
x = np.array([
    [1, 0], # Word 1
    [0, 1], # Word 2
    [1, 1]  # Word 3
])

# 2. Randomly initialize our weights
# In a real model, these are learned over time.
W_q = np.array([[1, 0], [0, 1]])
W_k = np.array([[1, 1], [0, 1]])
W_v = np.array([[1, 2], [3, 0]])

# 3. Create our Q, K, and V
Q = x @ W_q
K = x @ W_k
V = x @ W_v

# 4. Calculate Scores (How much does Word A like Word B?)
scores = Q @ K.T

# 5. Softmax (Turn scores into percentages that sum to 100%)
def softmax(x):
    return np.exp(x) / np.sum(np.exp(x), axis=-1, keepdims=True)

weights = softmax(scores)

# 6. Final Result: The "Context-Aware" version of our words
output = weights @ V
```



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

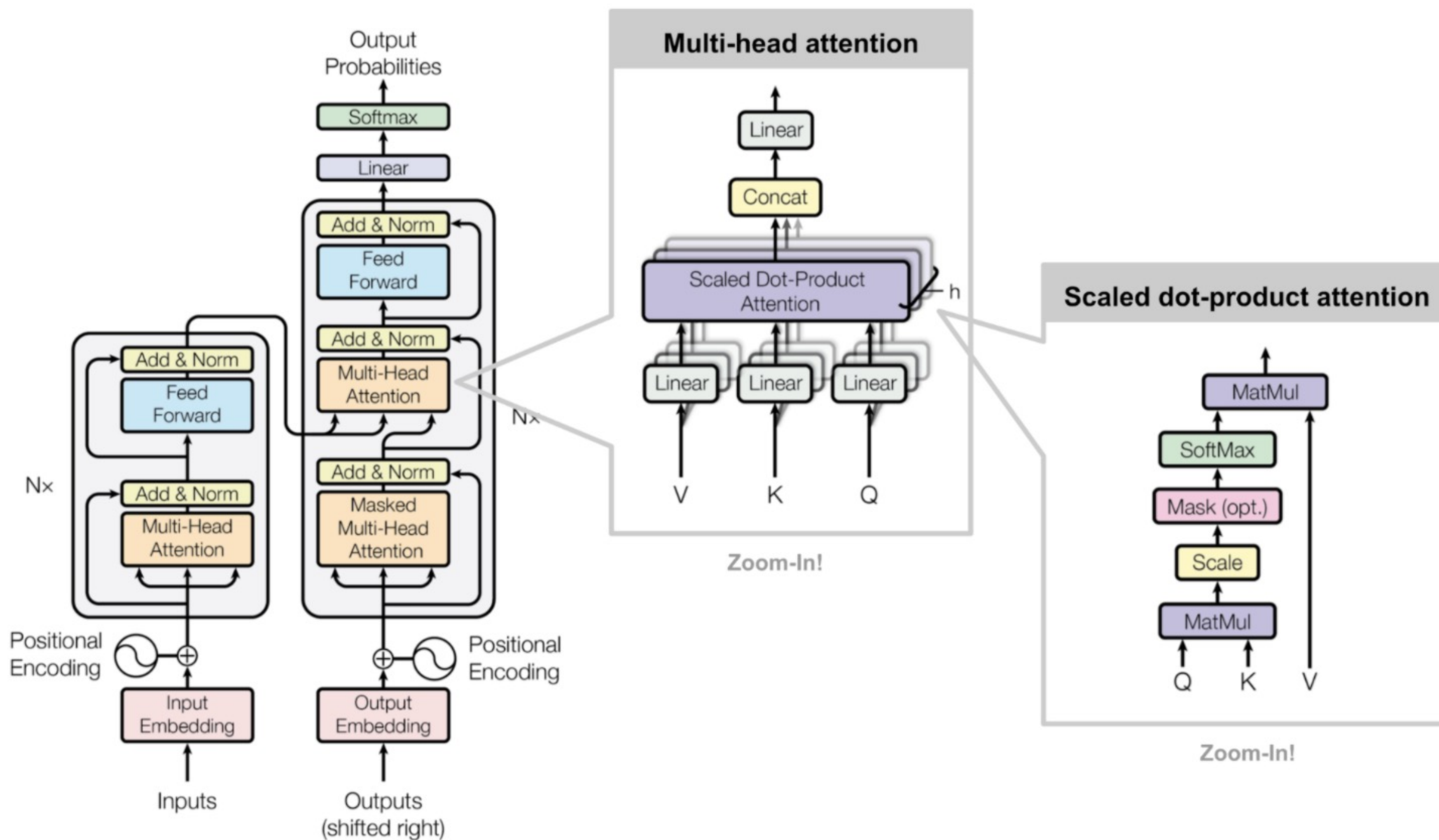
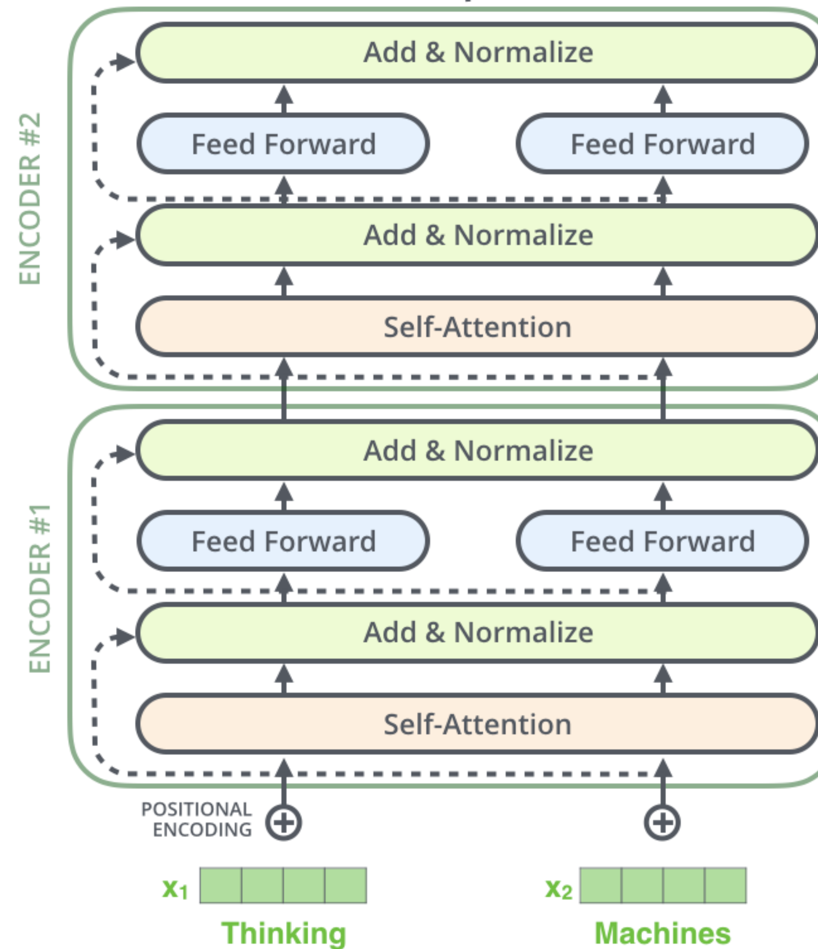


Fig. 17. The full model architecture of the transformer. (Image source: Fig 1 & 2 in Vaswani, et al., 2017.)

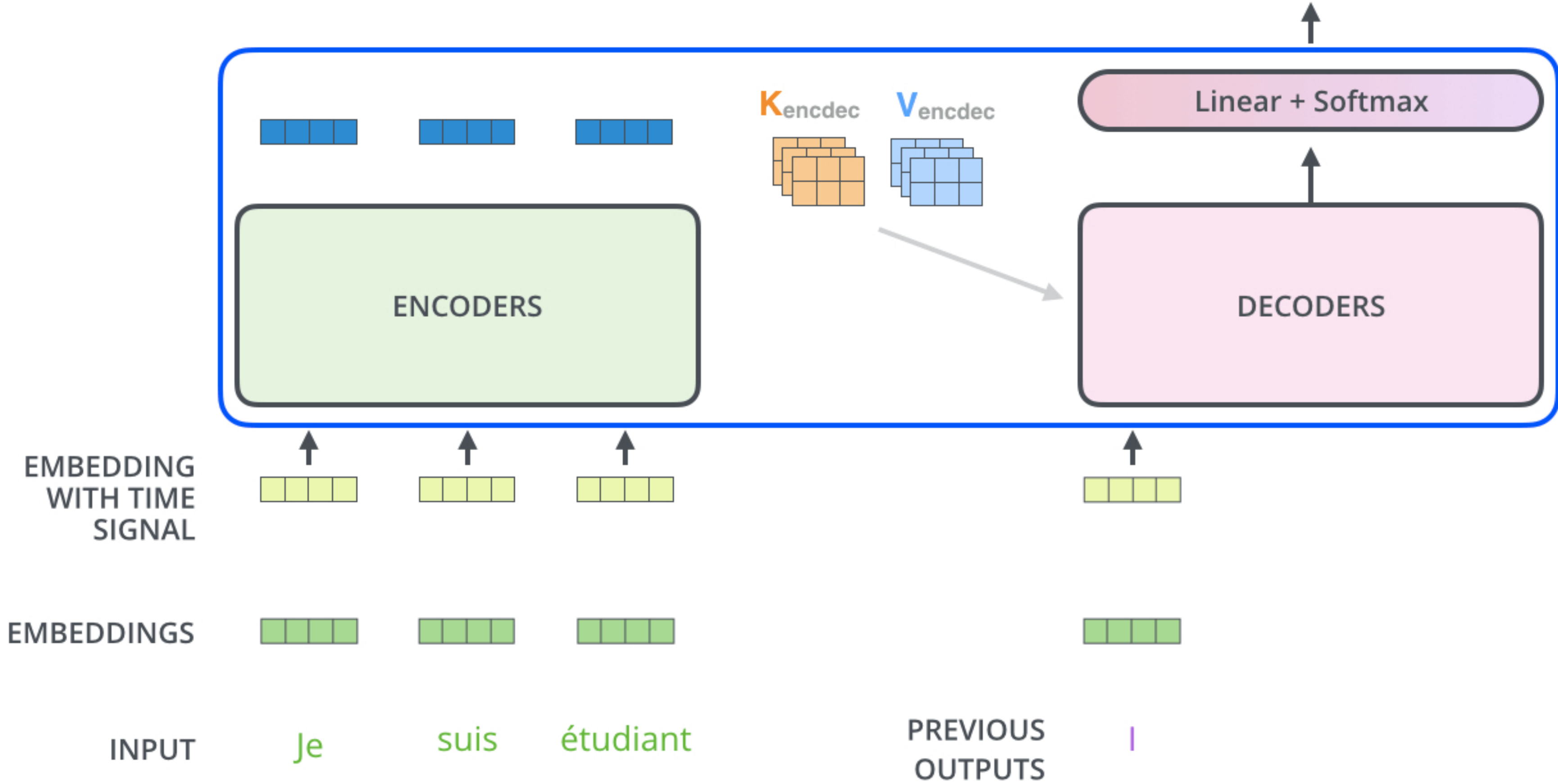
https://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

Encoder



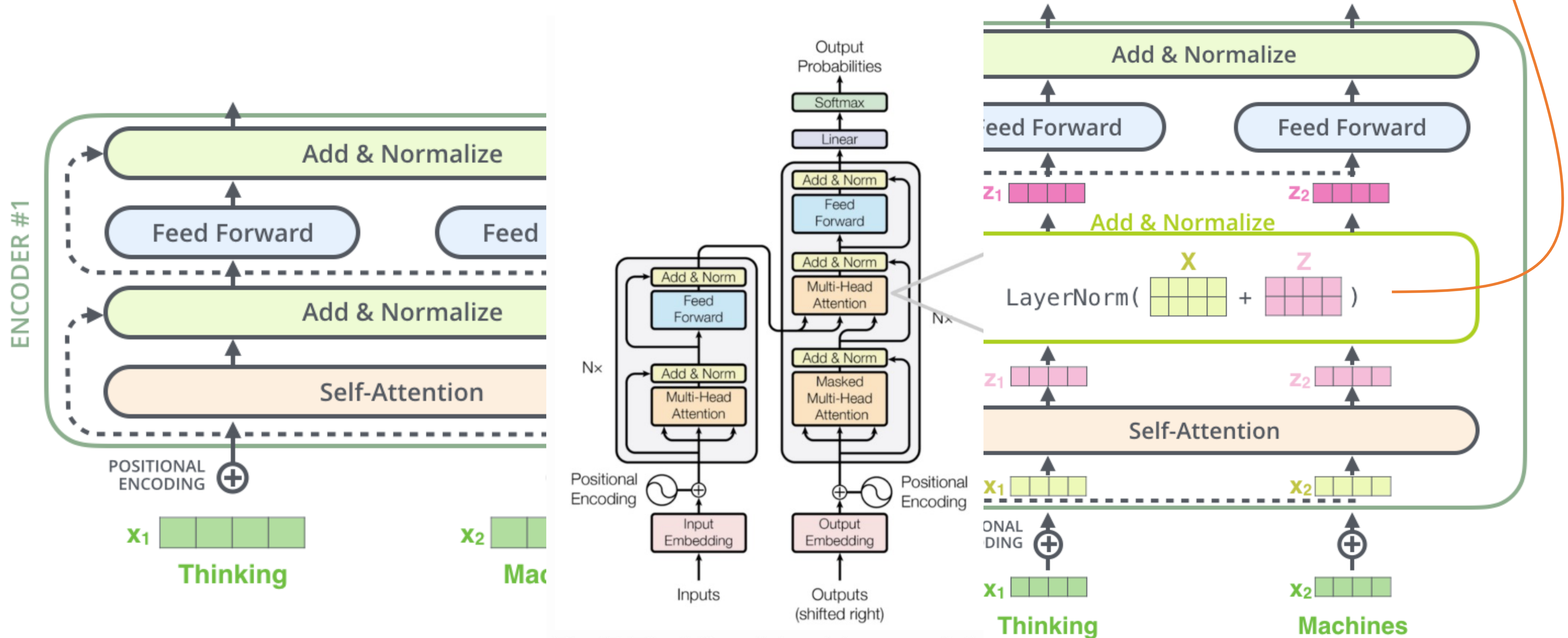
https://jalammar.github.io/illustrated-transformer/

Decoder



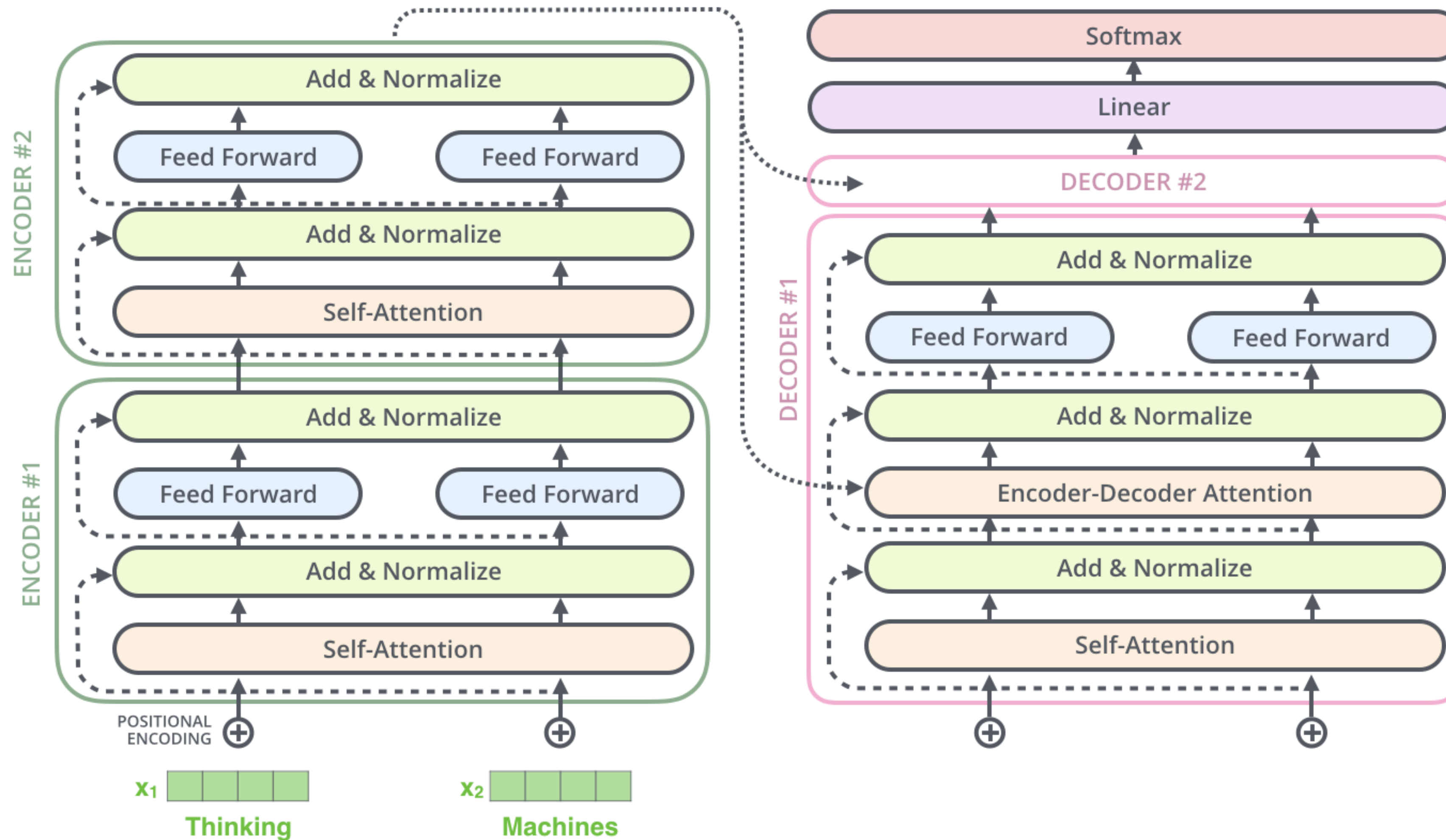
Skip Connections & Normalization

LayerNorm applied to each embedding independently



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

Skip Connections & Normalization



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

Reference Material

- Chapter on self-attention and Transformer with code:
  - https://d2l.ai/chapter_attention-mechanisms-and-transformers/index.html
- Implementations from Scratch in Python
  - https://huggingface.co/datasets/bird-of-paradise/transformer-from-scratch-tutorial/blob/main/Transformer_Implementation_Tutorial.ipynb
  - https://github.com/shunzh/llm.ipynb/blob/main/llm.ipynb
- Tutorials
  - https://e2eml.school/transformers.html
  - https://jalammar.github.io/illustrated-transformer/

Providing the Input Embeddings

- For image data
  - Divide an image into patches
  - Pass each patch of pixels through a linear layer
- For text data
  - Character-level
  - Word-level
  - Sub-word level
- Also add positional information

Tokenization and Token Embeddings for Textual Data

- Character tokenization with one-hot encodings
- Word tokenization with e.g. word2vec embeddings
- Sub-word tokenization with byte-pair embeddings/encoding

Byte-pair Encoding (BPE)

Example from: https://huggingface.co/learn/llm-course/en/chapter6/5

- Represent frequent byte-pairs as tokens
- E.g., given the corpus:

  Corpus (“word”, frequency): ("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)

  our vocabulary would be:

  ("h" "u" "g", 10), ("p" "u" "g", 5), ("p" "u" "n", 12), ("b" "u" "n", 4), ("h" "u" "g" "s", 5)
- “ug” and “un” can be recognized to be very frequent. So, combine them:

  Vocabulary: ["b", "g", "h", "n", "p", "s", "u", "ug", "un", "hug"]

  Corpus: ("hug", 10), ("p" "ug", 5), ("p" "un", 12), ("b" "un", 4), ("hug" "s", 5)

WordPiece

Example from: https://huggingface.co/learn/llm-course/chapter6/6

- First, split each word into the first letter and non-first letters:
  - E.g., given:

    ("hug", 10), ("pug", 5), ("pun", 12), ("bun", 4), ("hugs", 5)
  - Calculate (“##” marks a “WordPiece”):

  ("h" "##u" "##g", 10), ("p" "##u" "##g", 5), ("p" "##u" "##n", 12), ("b" "##u" "##n", 4), ("h" "##u" "##g" "##s", 5)

- Merge pairs using:

$$\text{score}=\frac{\mathrm{frequency}(AB)}{\mathrm{frequency}(A)\times\mathrm{frequency}(B)}$$

  - i.e., if merging two pairs is sufficiently frequent compared to their frequencies multiplied together, merge them.

A Comparison among Embeddings

Comparison Criteria:

- Out-of-vocabulary (OOV) words
  - Word embeddings struggle with out-of-vocabulary (OOV) words
  - Char embeddings are better with OOV words as they use chars. However, char embeddings fail at capturing semantically meaningful entities larger than chars
- Vocabulary size
  - Word embeddings have large vocabulary size
  - Char embeddings are better in this regard
  - BPE provides a good balance
- Sub-word (prefix, suffix, root/stem) semantics
  - BPE handles sub-words better
- Language specificity
  - Word embeddings are language specific
- Sequence Length
  - Word-embeddings are shorter than character embeddings

Comparison among Embeddings

Method	Out-of-Vocabulary Generalization	Vocabulary Size	Sub-word Semantics	Language Specificity	Sequence Length
Character-level	Excellent	Very small	Weak	Language agnostic	Very long
Word-level	Poor	Very large	None	Language specific	Short (one token per word)
Byte-pair	Very good	Medium/flexible (user-defined)	Very good	Medium	Longer than word-level
WordPiece	Good (slightly worse than BPE)	Medium/flexible (user-defined)	Very good	Medium	Longer than word-level

Positional Encoding: Motivation

Important to distinguish:

“man bites dog”

vs

“dog bites man”

Self – attention is unaware about positions:

$$e_i' = \sum_j \frac{\exp\left(k\left(e_j^T\right)q(e_i)\right)}{\sum_m \exp(k(e_m^T)q(e_i))} v(e_j)$$

	Index 2	Index 1	Index 0
Embedding 0	0	0	0
Embedding 1	0	0	1
Embedding 2	0	1	0
Embedding 3	0	1	1
Embedding 4	1	0	0
Embedding 5	1	0	1
Embedding 6	1	1	0
Embedding 7	1	1	1

Positional Encoding

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

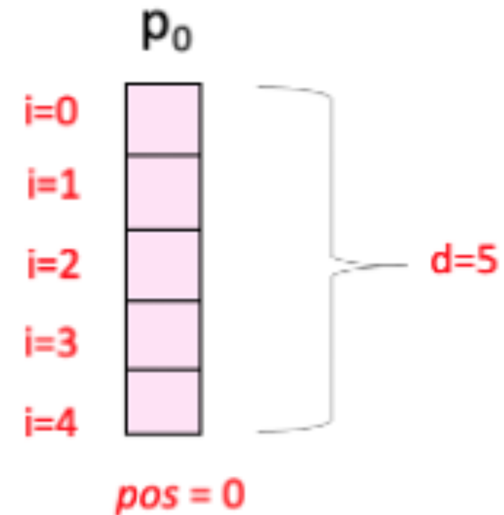


Fig from: https://www.youtube.com/watch?v=dichIcUZfOw

Positional Encoding

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

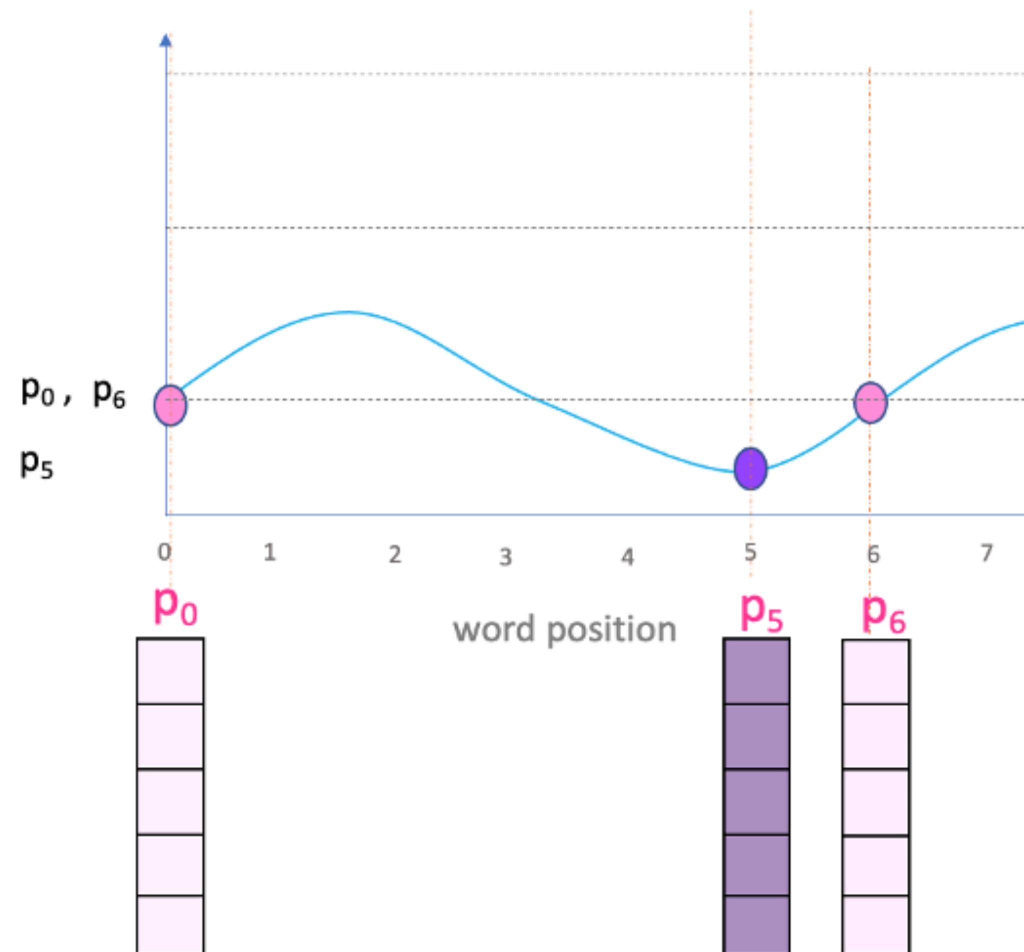


Fig from: https://www.youtube.com/watch?v=dichIcUZfOw

Positional Encoding

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d}}}\right)$$

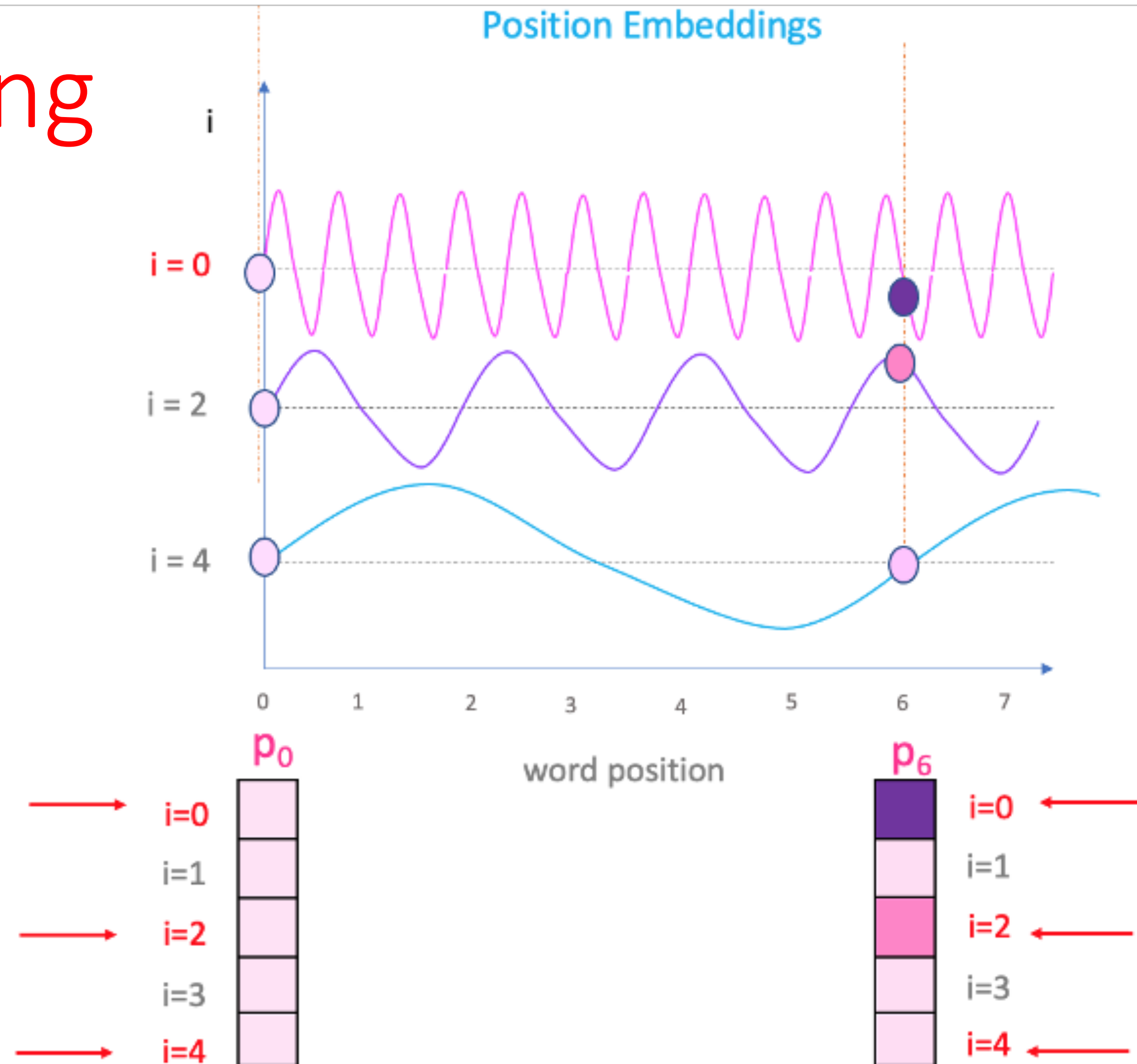


Fig from: https://www.youtube.com/watch?v=dichIcUZfOw

Positional Encoding: Alternatives

Important to distinguish: “man bites dog” vs “dog bites man”

- Hand-crafted position embeddings (using the sin function)
- Learnable position embeddings
- Relative position embeddings
- Rotary positional embedding (ROPE)

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Positional Encoding: Alternatives

- Hand-crafted position embeddings (using the sin function)
- **Learnable position embeddings**
- Relative position embeddings
- Rotary positional embedding (ROPE)

Define position embedding matrix as a learnable tensor (each e_{ij} is a learnable parameter):

	Index 2	Index 1	Index 0
Embedding 0	e_{02}	e_{01}	e_{00}
Embedding 1	e_{12}	e_{11}	e_{10}
Embedding 2	e_{22}	e_{21}	e_{20}
Embedding 3	e_{32}	e_{31}	e_{30}
Embedding 4	e_{42}	e_{41}	e_{40}
Embedding 5	e_{52}	e_{51}	e_{50}
Embedding 6	e_{62}	e_{61}	e_{50}
Embedding 7	e_{72}	e_{71}	e_{70}

Positional Encoding: Alternatives

- Hand-crafted position embeddings (using the sin function)
- Learnable position embeddings
- **Relative position embeddings**
- Rotary positional embedding (ROPE)

Self-Attention with Standard Position Embeddings

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V) \qquad \alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}} \qquad e_{ij} = \frac{(x_i W^Q)(x_j W^K)^T}{\sqrt{d_z}}$$

Self-Attention with Relative Position Embeddings

$$z_i = \sum_{j=1}^{n} \alpha_{ij}(x_j W^V + a_{ij}^V) \qquad \alpha_{ij} = \frac{\exp e_{ij}}{\sum_{k=1}^{n} \exp e_{ik}} \qquad e_{ij} = \frac{x_i W^Q (x_j W^K + a_{ij}^K)^T}{\sqrt{d_z}}$$

a_{ij} encodes relative positions between embeddings x_i and x_j.
w: learnable parameter.

$$a_{ij}^K = w_{\text{clip}(j-i,k)}^K$$
$$a_{ij}^V = w_{\text{clip}(j-i,k)}^V$$
$$\text{clip}(x, k) = \max(-k, \min(k, x))$$

Shaw et al., Self-Attention with Relative Position Representations, 2018.

Positional Encoding: Alternatives

- Hand-crafted position embeddings (using the sin function)
- Learnable position embeddings
- Relative position embeddings
- Rotary positional embedding (ROPE)

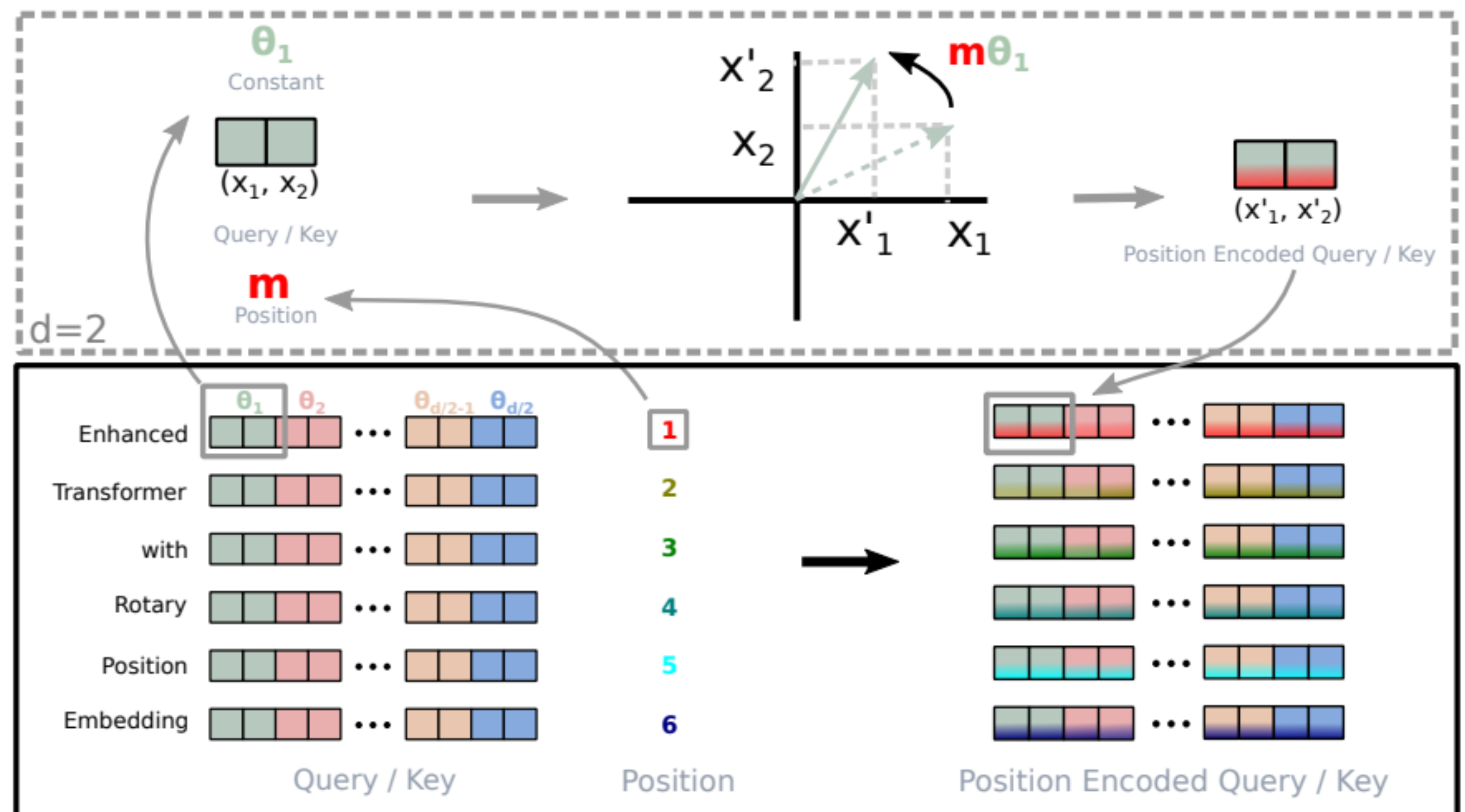


Fig: Su et al., RoFormer: Enhanced Transformer with Rotary Position Embedding, 2021.

$$\theta_i = 10000^{-2(i-1)/d}, i \in [1, 2, ..., d/2]$$

A Significant Issue with Self-Attention: Complexity

$$e_i' = \sum_j \frac{\exp\left(k(e_j^T)q(e_i)\right)}{\sum_m \exp(k(e_m^T)q(e_i))} v(e_j)$$

- If there are n tokens/embeddings,
  - Updating a single token requires $O(n)$ operations.
  - Overall: $O(n^2)$
- What is the complexity of an RNN layer with n time steps?

Linear Attention

ICML 2020

Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention

Angelos Katharopoulos [1,2] Apoorv Vyas [1,2] Nikolaos Pappas [3] François Fleuret [2,4,*]

Self-attention:

$$Q = xW_Q,$$
$$K = xW_K,$$
$$V = xW_V, \quad (2)$$
$$A_l(x) = V' = \operatorname{softmax}\left(\frac{QK^T}{\sqrt{D}}\right)V.$$

Rewrite Eq 2 for one row of the matrix:

$$V_i' = \frac{\sum_{j=1}^{N} \operatorname{sim}(Q_i, K_j) V_j}{\sum_{j=1}^{N} \operatorname{sim}(Q_i, K_j)}. \quad (3)$$

Equation 3 is equivalent to equation 2 if we substitute the similarity function with $\operatorname{sim}(q, k) = \exp\left(\frac{q^T k}{\sqrt{D}}\right)$.

Constraint for sim(): It should be non-negative.
Then, we can choose any other kernel/function:

Given such a kernel with a feature representation $\phi(x)$ we can rewrite equation 2 as follows,

$\phi(x) = \operatorname{elu}(x) + 1$

$$V_i' = \frac{\sum_{j=1}^{N} \phi(Q_i)^T \phi(K_j) V_j}{\sum_{j=1}^{N} \phi(Q_i)^T \phi(K_j)}, \quad (4)$$

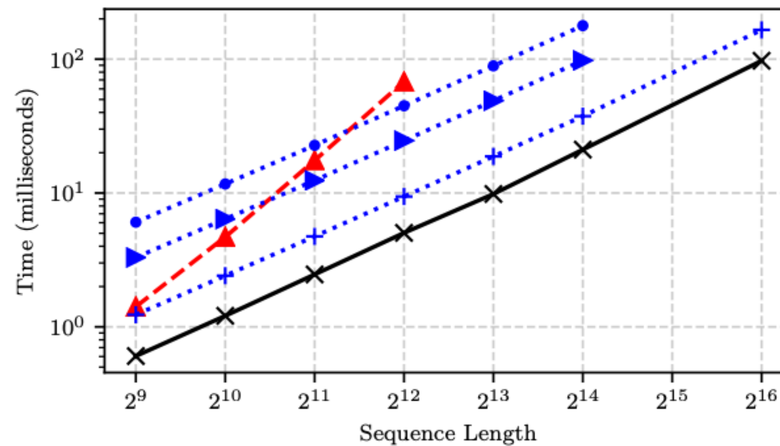
and then further simplify it by making use of the associative property of matrix multiplication to

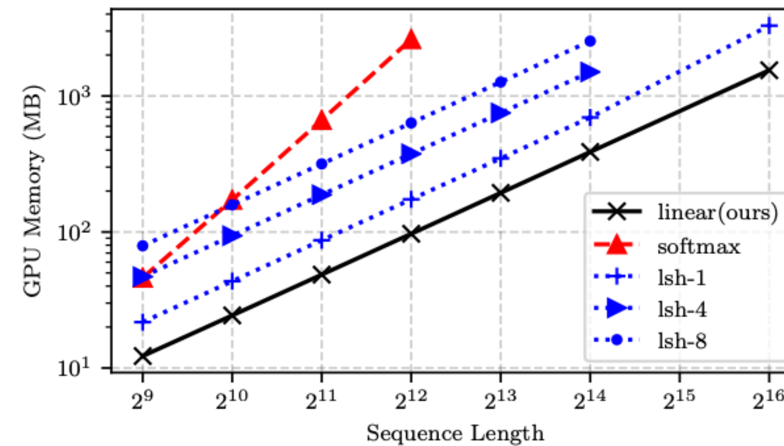
$$V_i' = \frac{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j) V_j^T}{\phi(Q_i)^T \sum_{j=1}^{N} \phi(K_j)}. \quad (5)$$

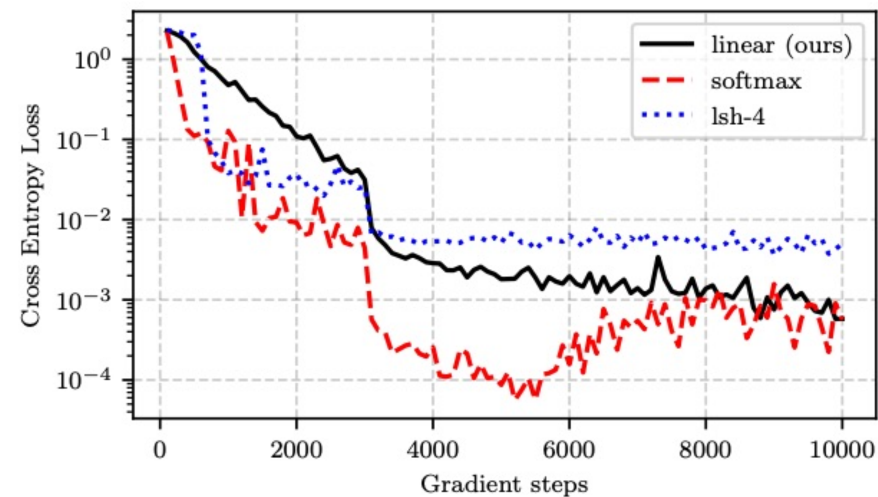
Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention

Angelos Katharopoulos [1 2] Apoorv Vyas [1 2] Nikolaos Pappas [3] François Fleuret [2 4 *]

Linear Attention







Method	Validation PER	Time/epoch (s)
Bi-LSTM	10.94	1047
Softmax	5.12	2711
LSH-4	9.33	2250
Linear (ours)	8.08	**824**

Table 3: Performance comparison in automatic speech recognition on the WSJ dataset. The results are given in the form of phoneme error rate (PER) and training time per epoch. Our model outperforms the LSTM and Reformer while being faster to train and evaluate. Details of the exper-