CS60050: Machine Learning

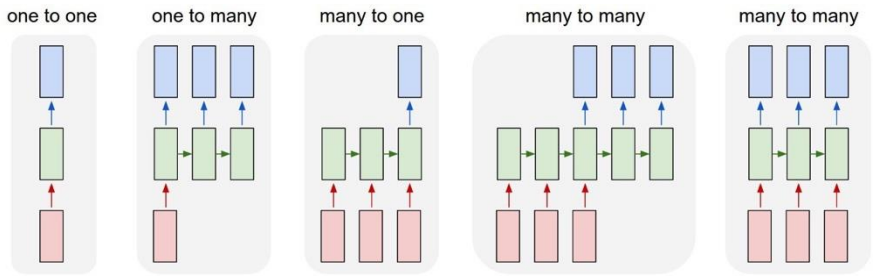
RNN, Attention and Transformers

Sourangshu Bhattacharya

Recurrent neural networks

Recurrent neural networks

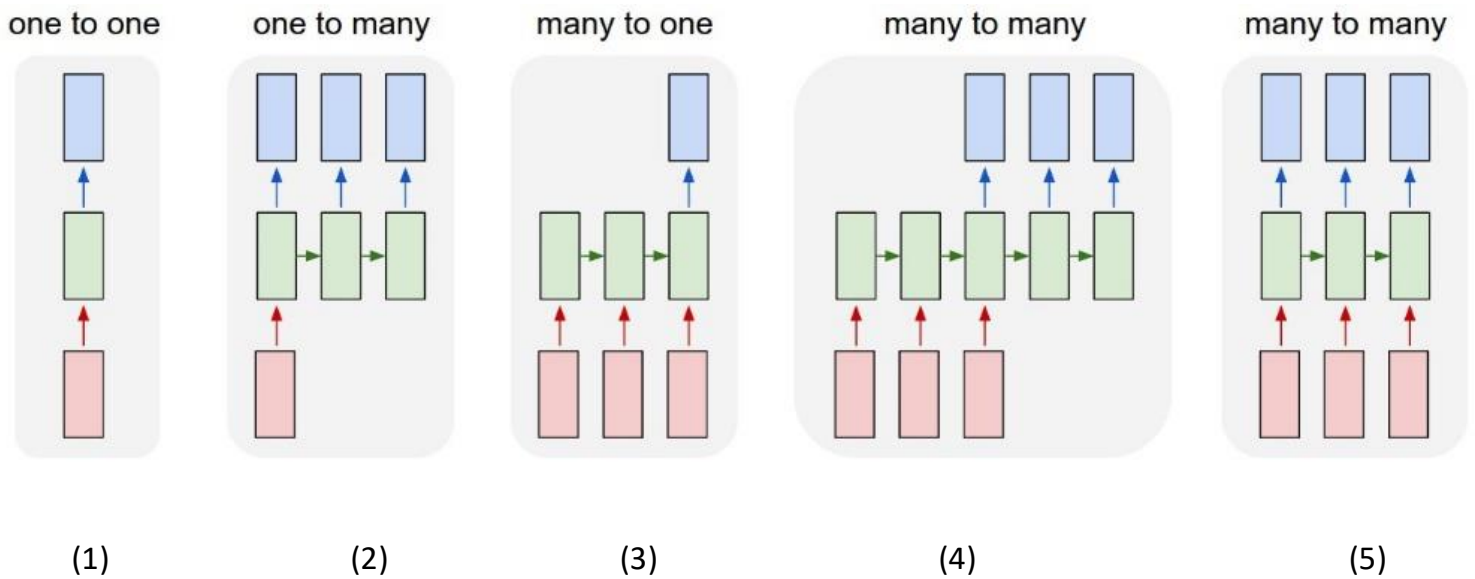
- Lots of information is sequential and requires a memory for successful processing
- Sequences as input, sequences as output



- Recurrent neural networks(RNNs) are called recurrent because they perform same task for every element of sequence, with output dependent on previous computations
- RNNs have memory that captures information about what has been computed so far
- RNNs can make use of information in arbitrarily long sequences – in practice they limited to looking back only few steps

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

Topologies of Recurrent Neural Network



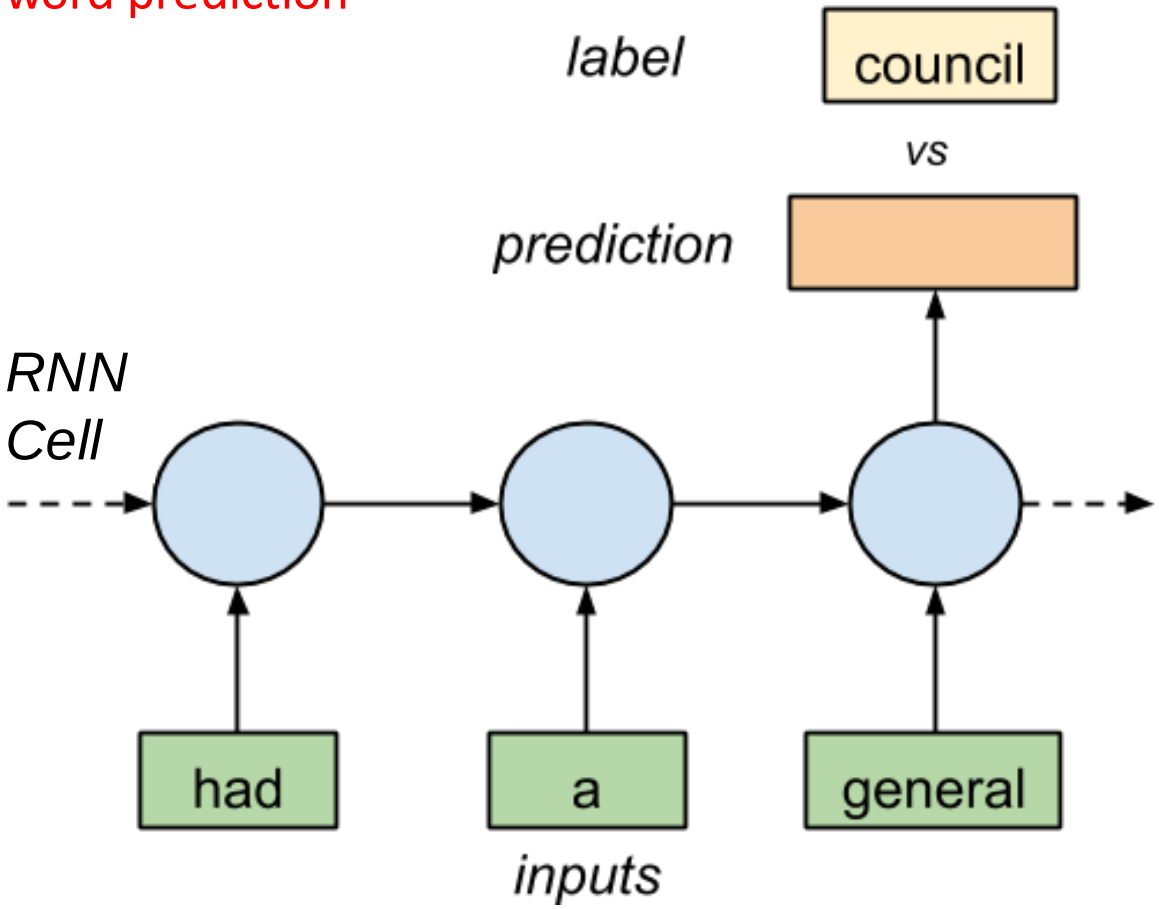
1) Common Neural Network (e.g. feed forward network)
2) Prediction of future states base on single observation
3) Sentiment classification
4) Machine translation
5) Simultaneous interpretation

Language Model

- Compute the probability of a sentence
- Useful in machine translation
  - Word ordering: p(the cat is small) > p(small the cat is)
  - Word choice: p(walking home after school) > p(walking house after school)

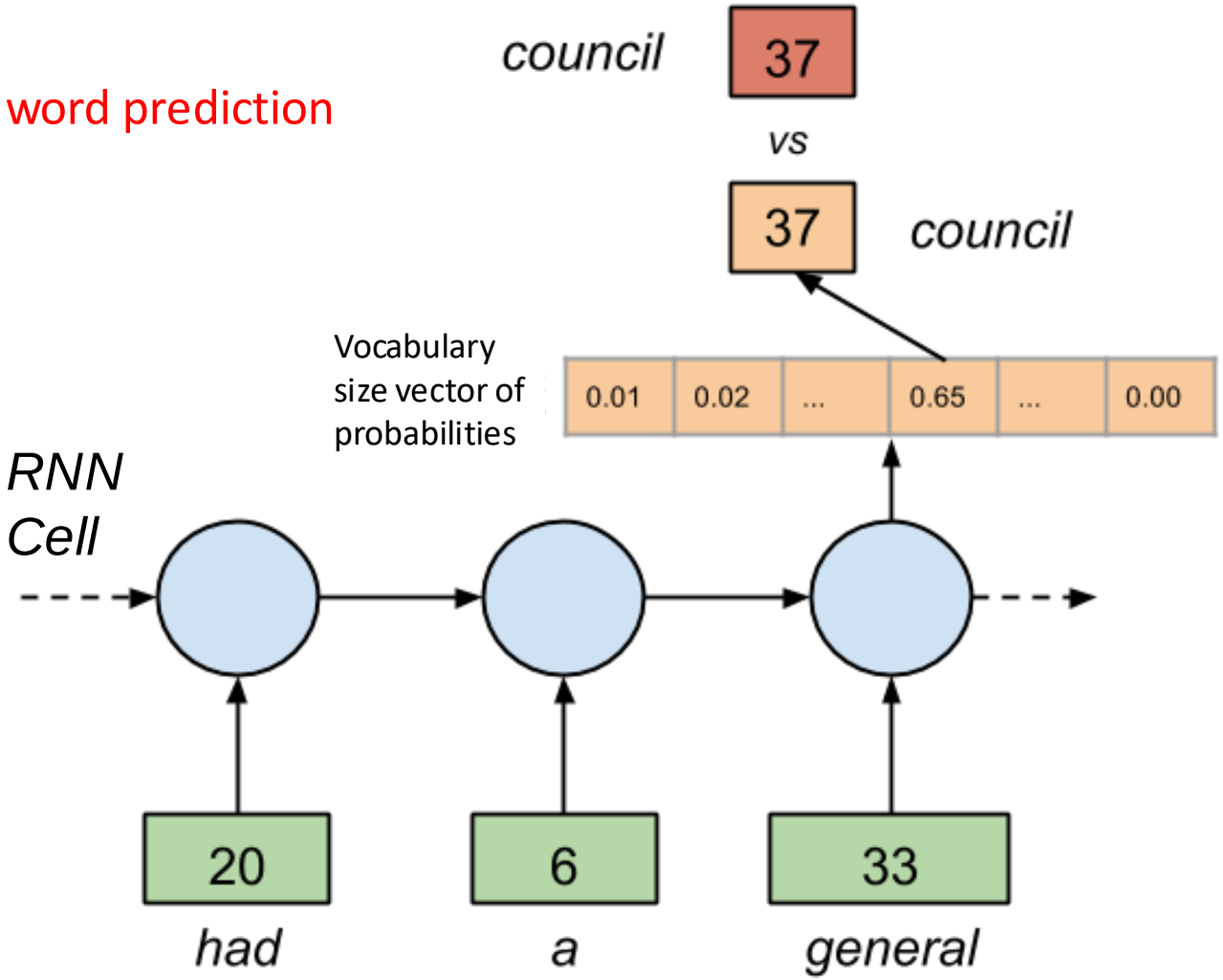
Recurrent Neural Network

Next word prediction

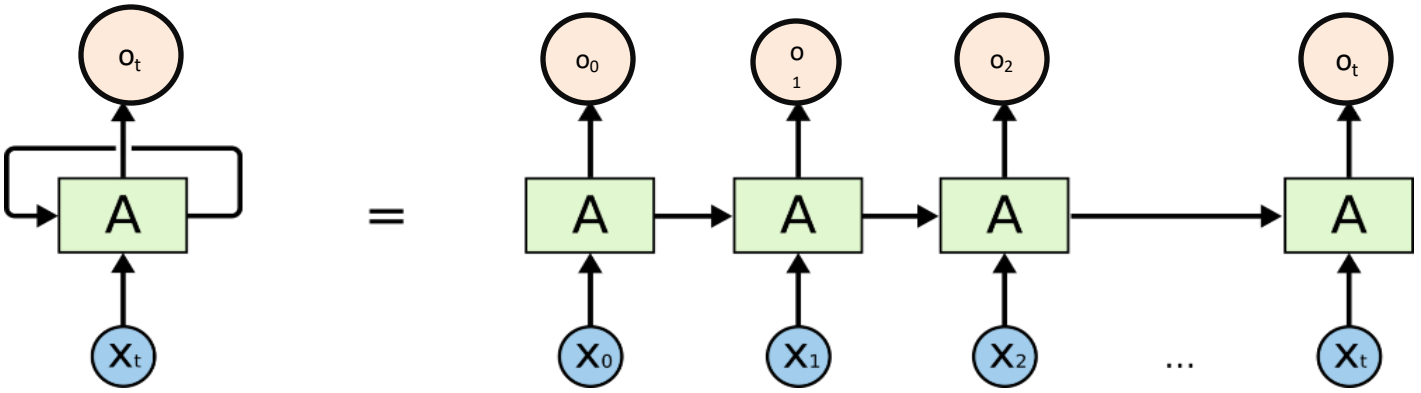


Recurrent Neural Network

Next word prediction



Recurrent Neural Network

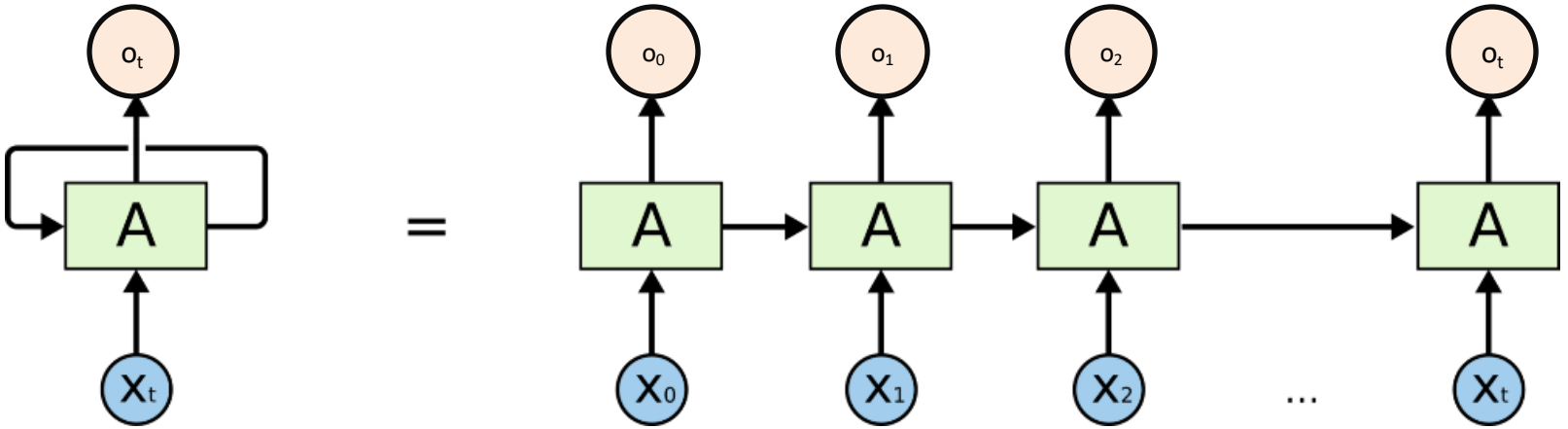


- Recurrent Neural Network have an internal state
- State is passed from input x_t to x_{t+1}

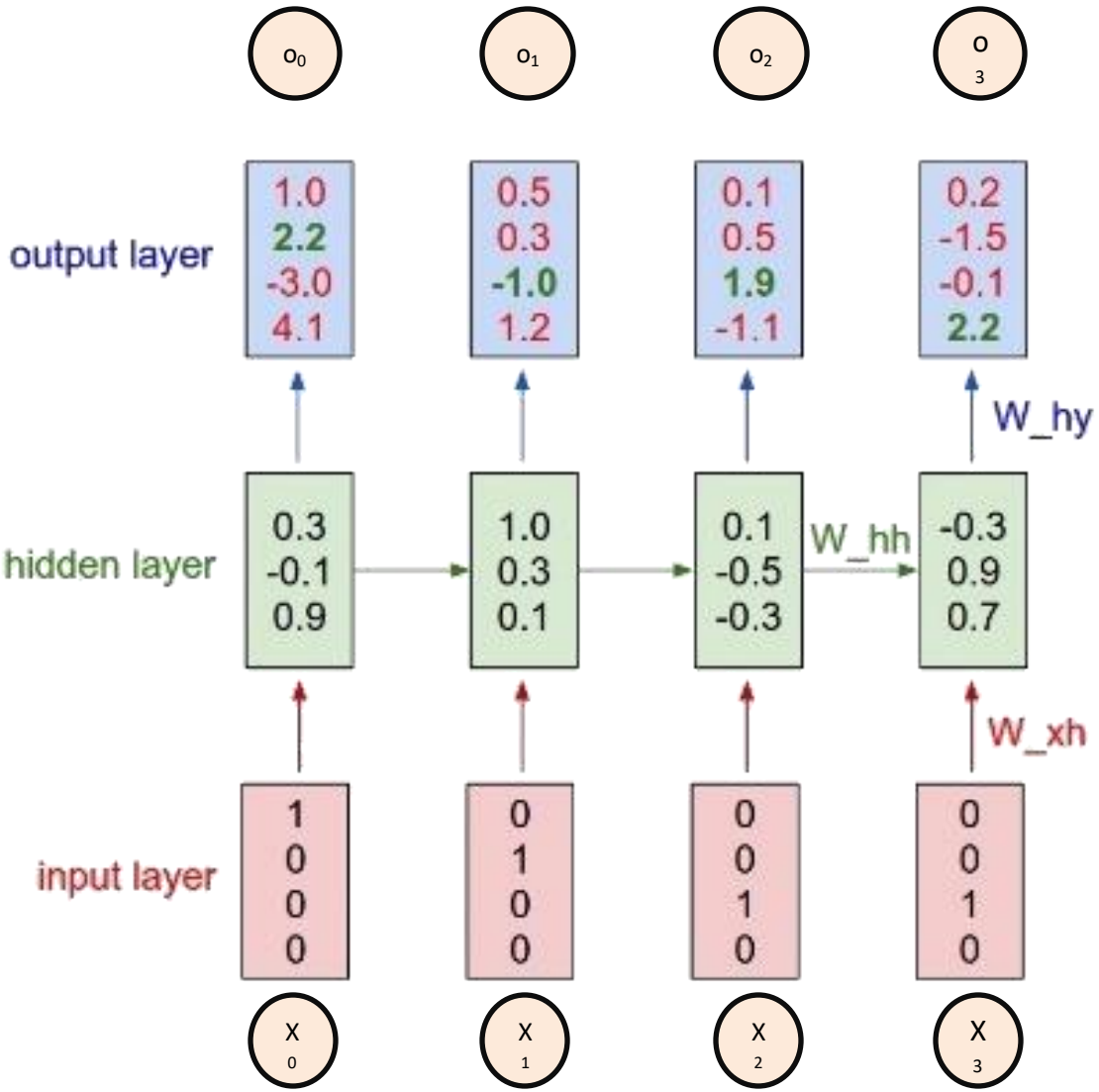
Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Language Models with RNN

- Let $x_0, x_1, x_2 \ldots$ denote words (input)
- Let $o_0, o_1, o_2 \ldots$ denote the probability of the sentence(output)
- Memory requirement scales nicely (linear with the number of word embeddings / number of character)

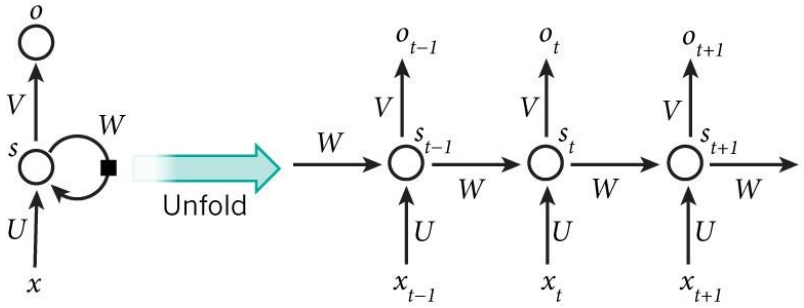


Recurrent Neural Network



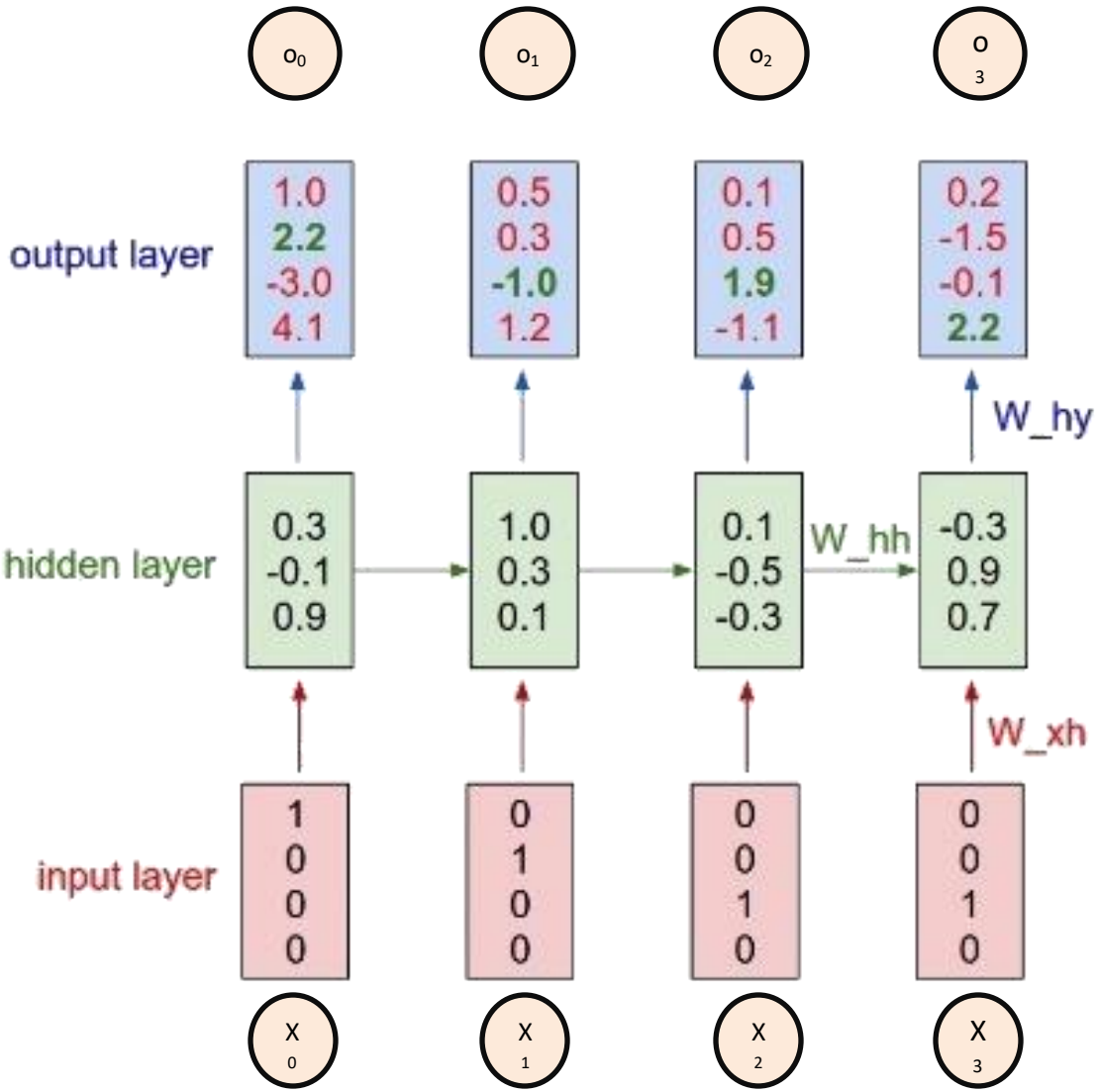
Recurrent neural networks

- RNN being unrolled (or unfolded) into full network
- Unrolling: write out network for complete sequence



- Image credits: Nature

RNN: How to learn?



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

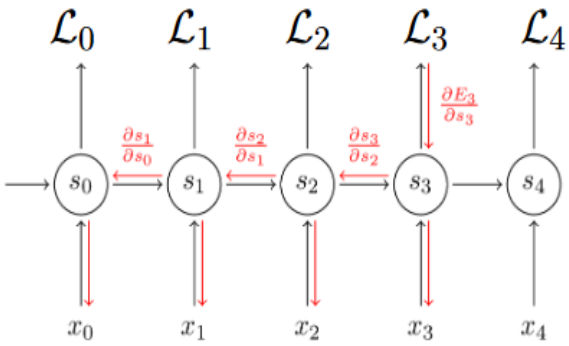
No Magic Involved (in Theory)

- You unroll your data in time
- You compute the gradients
- You use back propagation to train your network
- Karpathy presents a Python implementation for Char-RNN with 112 lines

- Training RNNs is hard:
  - Inputs from many time steps ago can modify output
  - Vanishing / Exploding Gradient Problem

- Vanishing gradients can be solved by Gated-RNNs like Long-Short-Term-Memory (LSTM) Models
  - LSTM became popular in NLP in 2015

Vanishing and exploding gradients



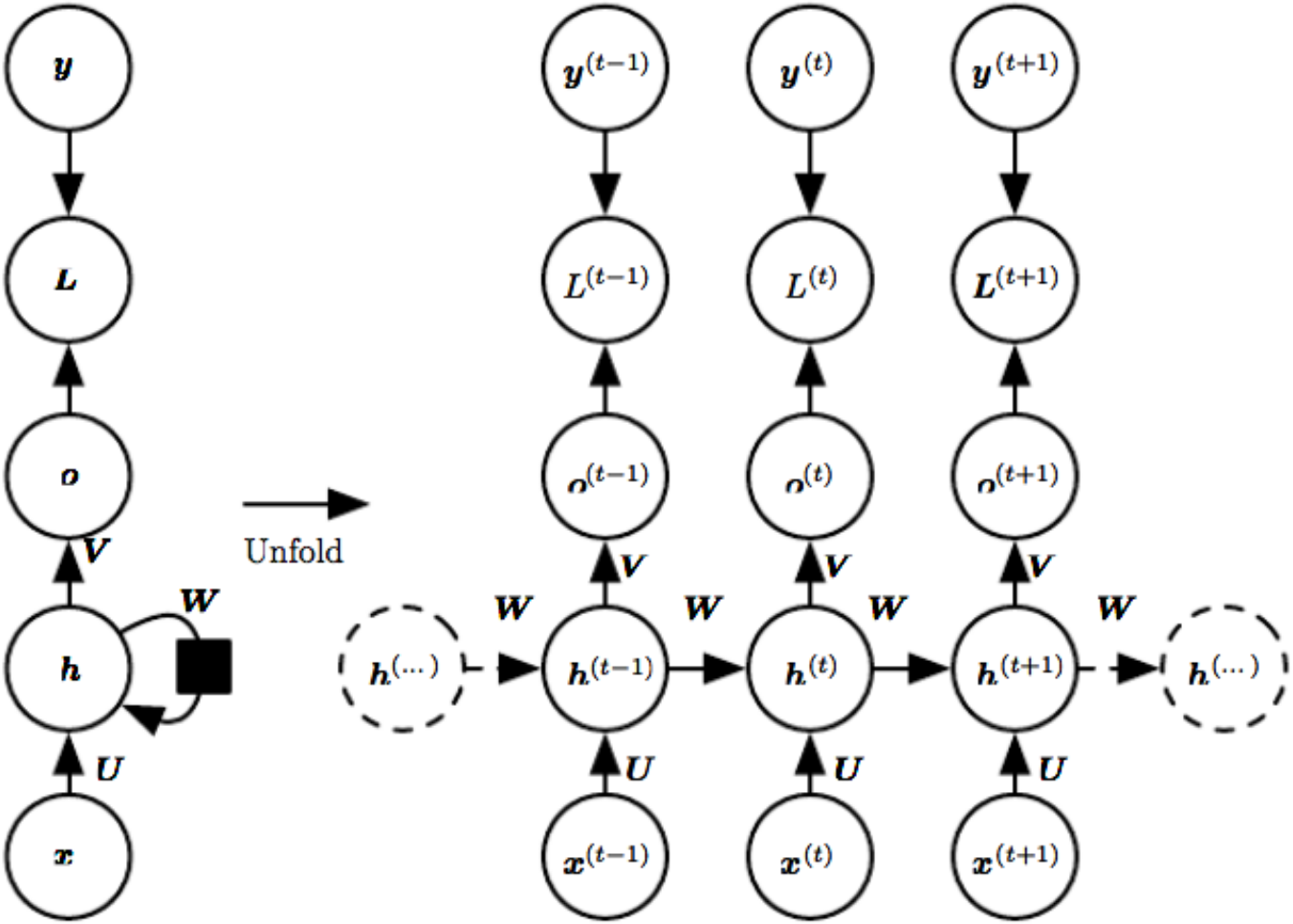
- For training RNNs, calculate gradients for U, V, W – ok for V but for W and U ...
- Gradients for W:

$$\frac{\partial \mathcal{L}_3}{\partial W} = \frac{\partial \mathcal{L}_3}{\partial o_3}\frac{\partial o_3}{\partial s_3}\frac{\partial s_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial \mathcal{L}_3}{\partial o_3}\frac{\partial o_3}{\partial s_3}\frac{\partial s_3}{\partial s_k}\frac{\partial s_k}{\partial W}$$

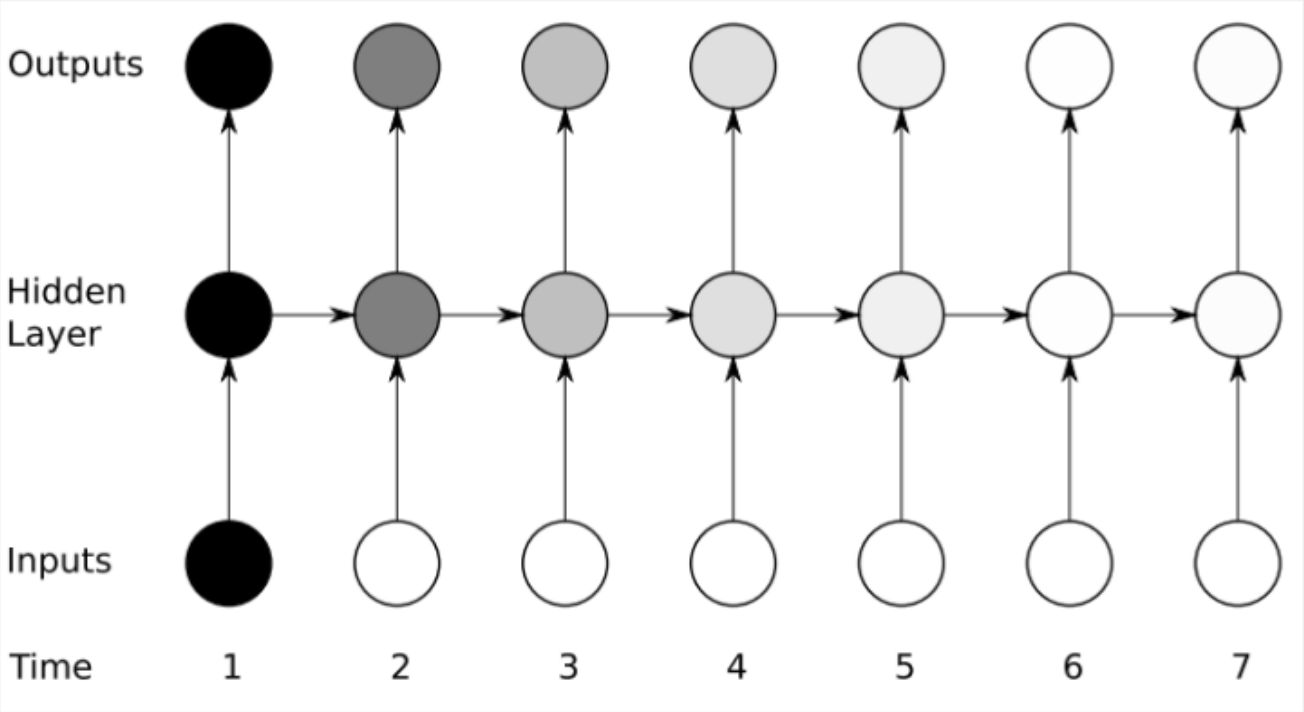
- More generally: $\frac{\partial \mathcal{L}}{\partial s_t} = \frac{\partial \mathcal{L}}{\partial s_m} \cdot \underbrace{\frac{\partial s_m}{\partial s_{m-1}}}_{<1} \cdot \underbrace{\frac{\partial s_{m-1}}{\partial s_{m-2}}}_{<1} \cdot \ldots \cdot \underbrace{\frac{\partial s_{t+1}}{\partial s_t}}_{<1} \Rightarrow \ll 1$
- Gradient contributions from far away steps become zero: state at those steps doesn't contribute to what you are learning

L_i – Loss, U, V, W – Parameters, S_i - states

Vanishing and exploding gradients



Vanishing and exploding gradients



Heatmap

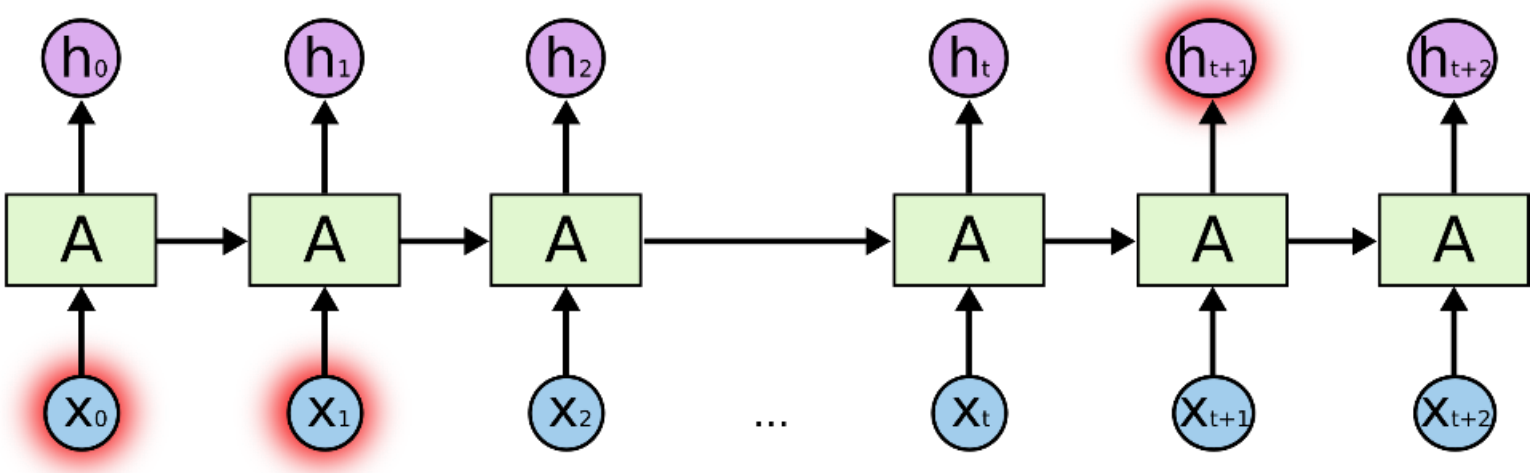
Long Short Term Memory [Hochreiter and Schmidhuber, 1997]

LSTMs designed to combat vanishing gradients through gating mechanism

How LSTM calculates hidden state s_t

$$
\begin{aligned}
i &= \sigma(x_t U^i + s_{t-1} W^i) \\
f &= \sigma(x_t U^f + s_{t-1} W^f) \\
o &= \sigma(x_t U^o + s_{t-1} W^o) \\
g &= \tanh(x_t U^g + s_{t-1} W^g) \\
c_t &= c_{t-1} \circ f + g \circ i \\
s_t &= \tanh(c_t) \circ o
\end{aligned}
$$

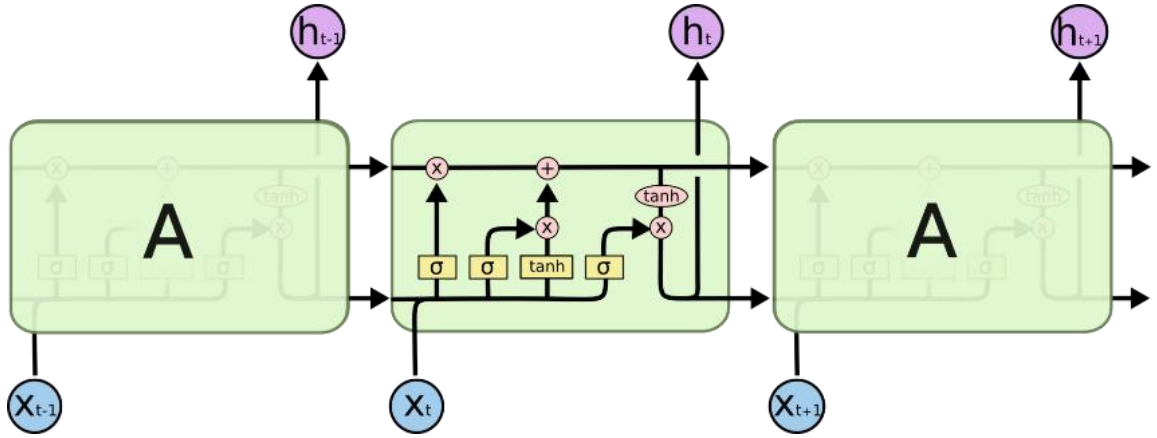
Long-Short-Term Memory (LSTM)



- Long-term dependencies:
  I grew up in France and lived there until I was 18. Therefore I speak fluent ???
- Presented (vanilla) RNN is unable to learn long term dependencies
  - Issue: More recent input data has higher influence on the output
- Long-Short-Term Memory (LSTM) models solves this problem

Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

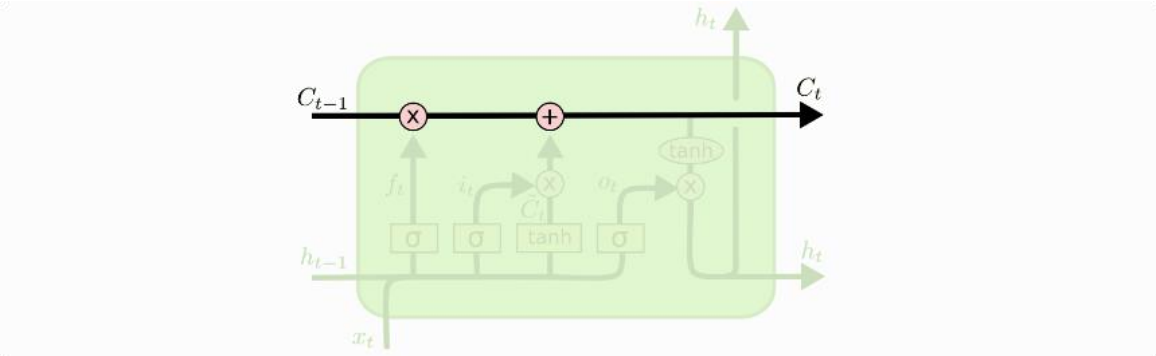
LSTM Model



- The LSTM model implements a *forget-gate* and an *add-gate*
- The models learns when to forget something and when to update internal storage

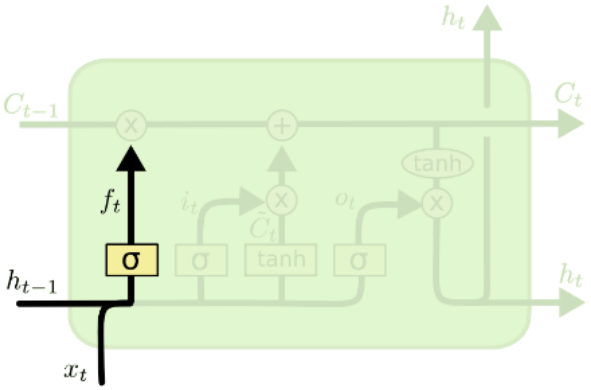
Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

LSTM Model



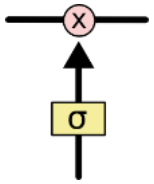
- Core: Cell-state *C* (a vector of certain size)
- The model has the ability to remove or add information using Gates

Forget-Gate



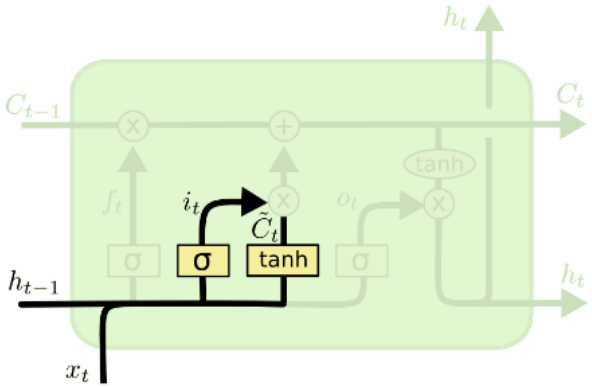
$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

- Sigmoid function σ output a value between 0 and 1
- The output is point-wise multiplied with the cell state C_{t-1}
- Interpretation:
  - 0: *Let nothing through*
  - 1: *Let everything through*



- Example: When we see a new subject, forget gender of old subject

Set-Gate

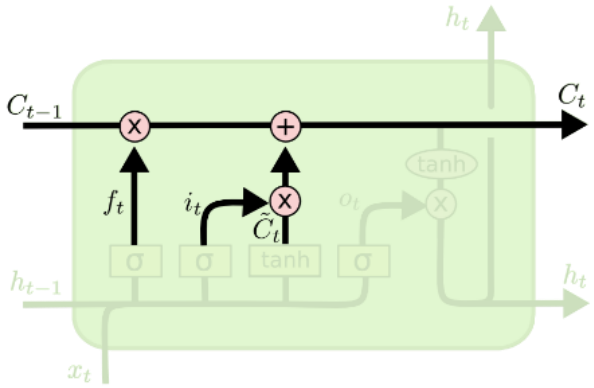


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

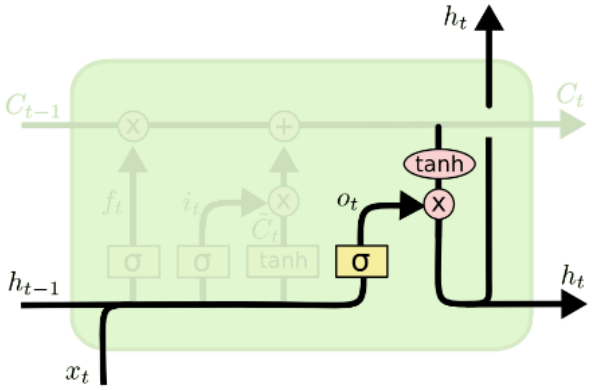
- Compute i_t which cells we want to update and to which degree (σ: 0 … 1)
- Compute the new cell value using the *tanh* function

Update Internal Cell State



$$C_t = \underbrace{f_t * C_{t-1}}_{\text{Forget state cells}} + \underbrace{i_t * \tilde{C}_t}_{\text{Update state cells}}$$

Compute Output h_t



$$o_t = \sigma\left(W_o\ [h_{t-1}, x_t]\ +\ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

- We use the updated cell state C_t to compute the output
- We might not need the complete cell state as output
  - Compute o_t, defining how relevant each cell is for the output
  - Pointwise multiply o_t with *tanh(C_t)*

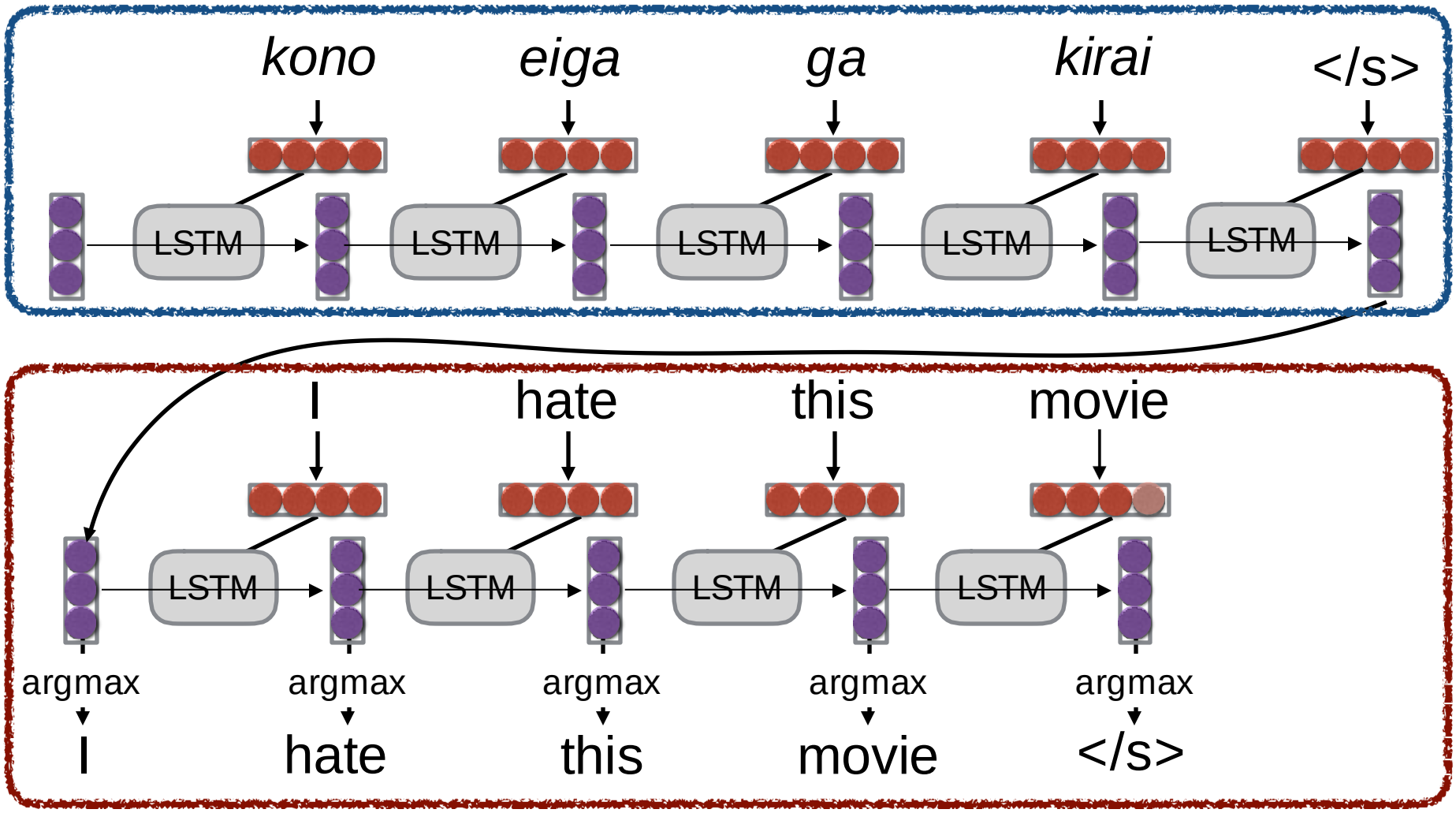
- Cell state C_t and output h_t is passed to the next time step

Img Source: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Machine translation

Encoder-decoder Models

(Sutskever et al. 2014)

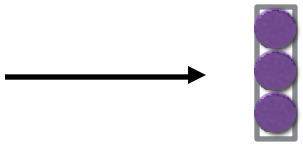


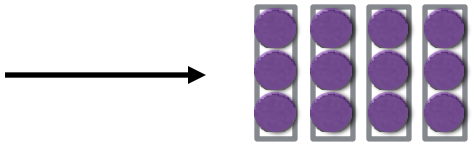
Sentence Representations

Problem!

"You can't cram the meaning of a whole %&!$ing sentence into a single $&!*ing vector!"
— Ray Mooney

- But what if we could use multiple vectors, based on the length of the sentence.

this is an example 

this is an example 

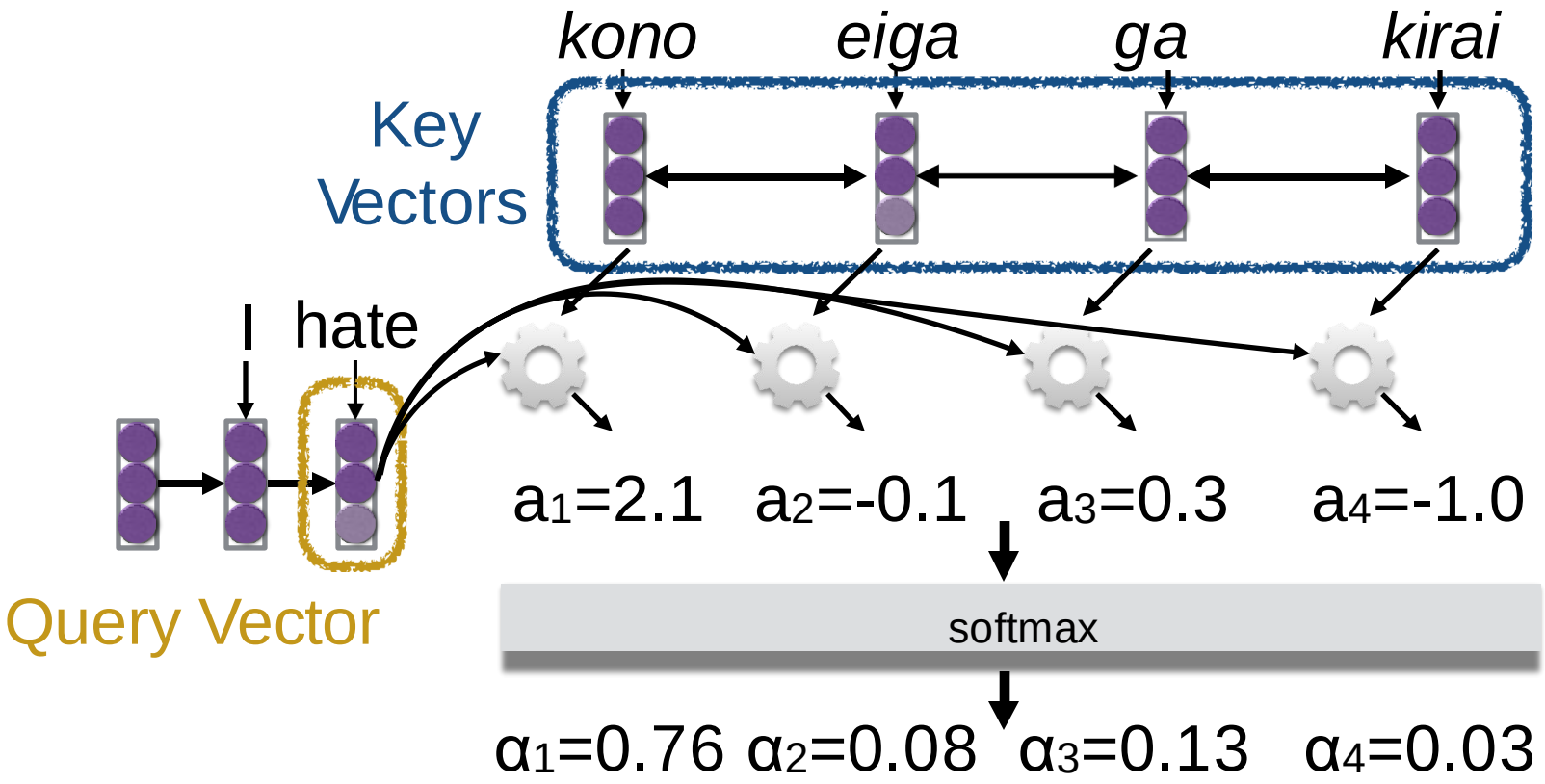
Attention - Basic Idea

(Bahdanau et al. 2015)

- Encode each word in the sentence into a vector
- When decoding, perform a linear combination of these vectors, weighted by “attention weights”
- Use this combination in picking the next word

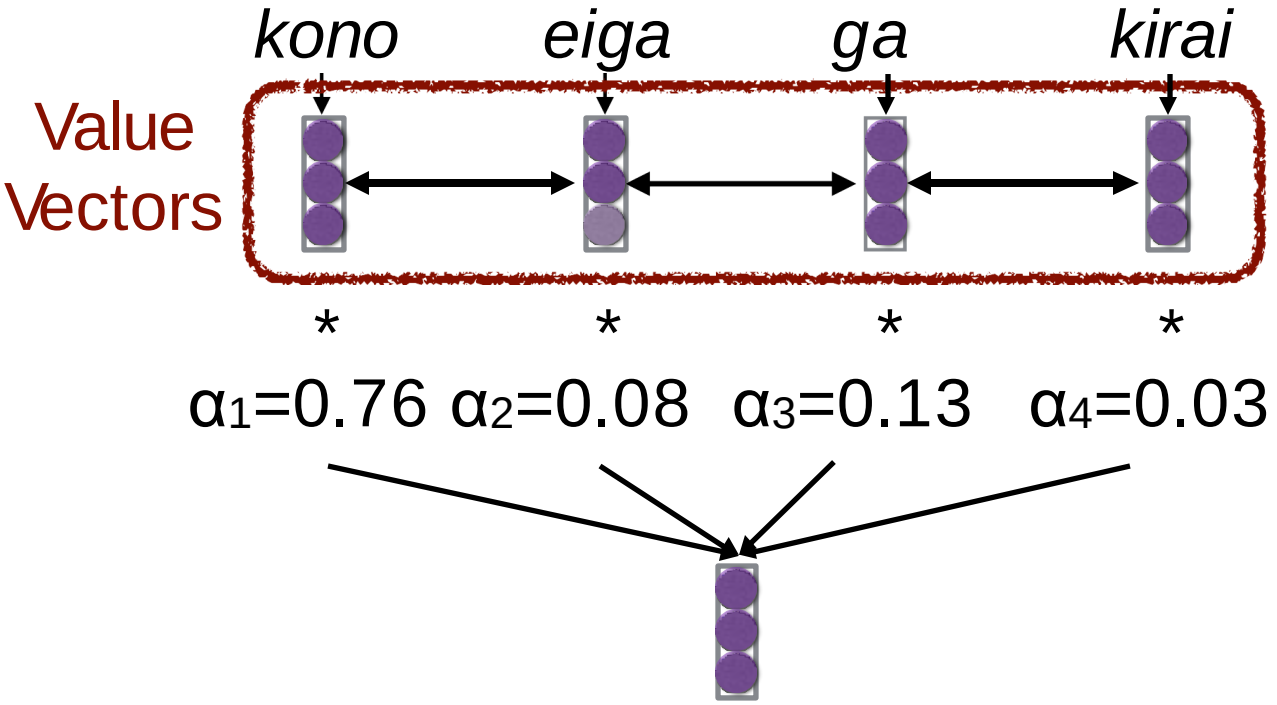
Calculating Attention (1)

- Use “query” vector (decoder state) and “key” vectors (all encoder states)
- For each query-key pair, calculate weight
- Normalize to add to one using softmax



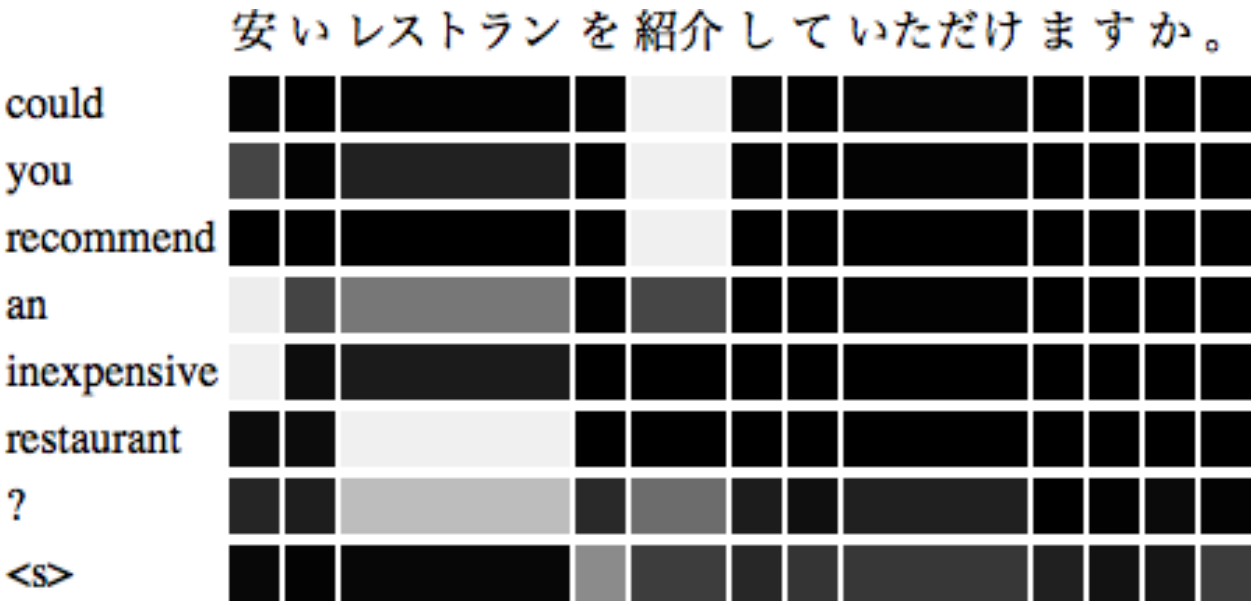
Calculating Attention (2)

- Combine together value vectors (usually encoder states, like key vectors) by taking the weighted sum



- Use this in any part of the model you like

A Graphical Example



Attention Score Functions (1)

- $\boldsymbol{q}$ is the query and $\boldsymbol{k}$ is the key
- **Multi-layer Perceptron** (Bahdanau et al. 2015)

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{w}_2^\intercal \tanh(W_1[\boldsymbol{q}; \boldsymbol{k}])$$

  - Flexible, often very good with large data

- **Bilinear** (Luong et al. 2015)

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^\intercal W \boldsymbol{k}$$

Attention Score Functions (2)

- **Dot Product** (Luong et al. 2015)

$$a(\boldsymbol{q}, \boldsymbol{k}) = \boldsymbol{q}^\intercal \boldsymbol{k}$$

  - No parameters! But requires sizes to be the same.

- **Scaled Dot Product** (Vaswani et al. 2017)

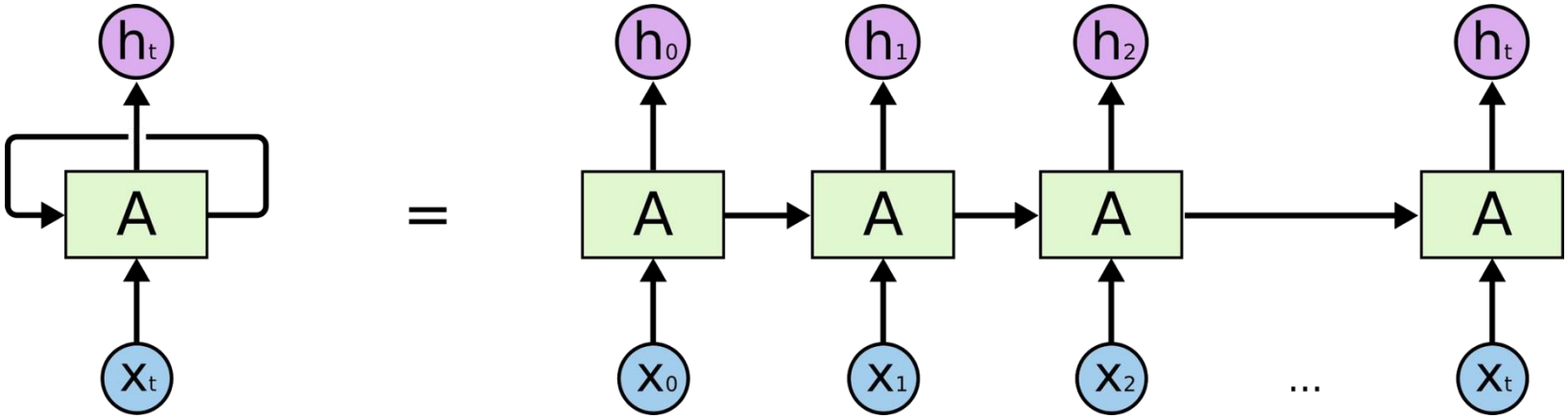
  - Problem: scale of dot product increases as dimensions get larger

  - Fix: scale by size of the vector

$$a(\boldsymbol{q}, \boldsymbol{k}) = \frac{\boldsymbol{q}^\intercal \boldsymbol{k}}{\sqrt{|\boldsymbol{k}|}}$$

Transformer: “Attention is All You Need” (Vaswani et al. 2017)

Problem: RNN constrained by previous timestep computation



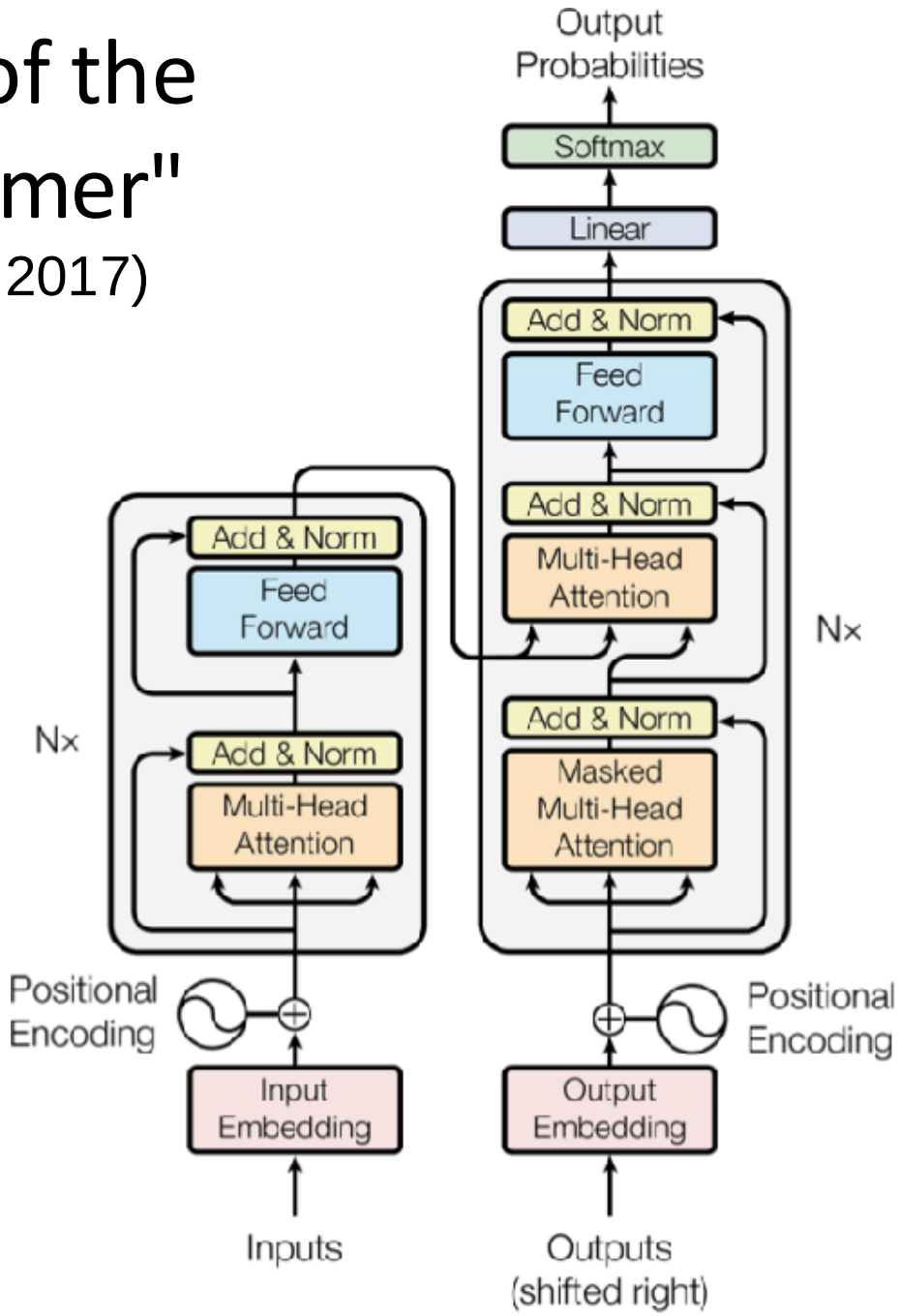
http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Target is to Improve the perfomrance and get rid of sequential computation

Summary of the "Transformer"

(Vaswani et al. 2017)

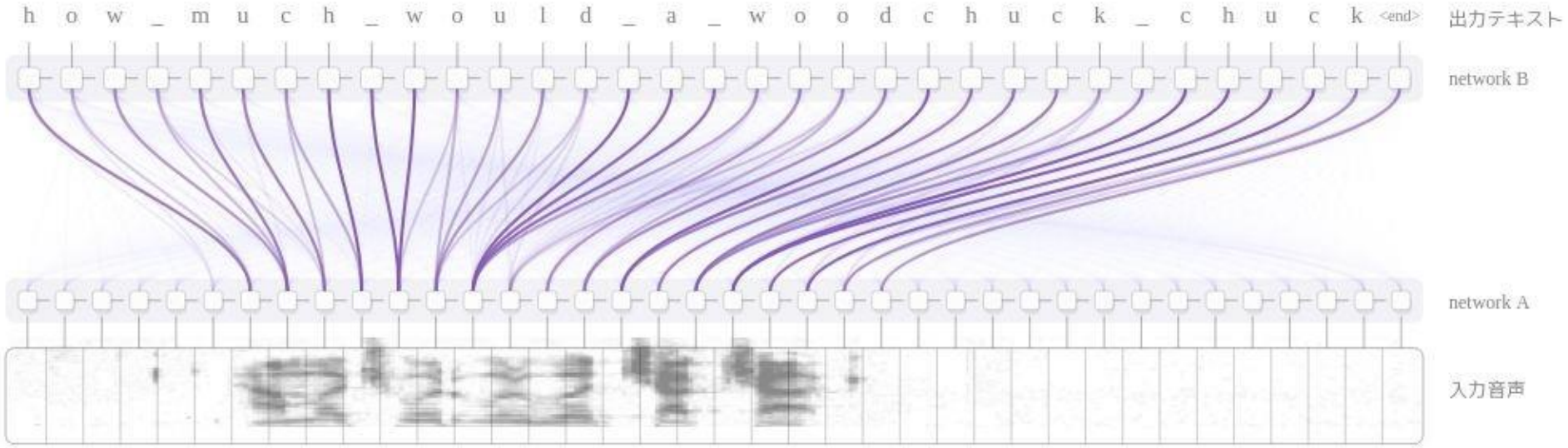
- A sequence-to-sequence model based entirely on attention
- Strong results on standard WMT datasets
- Fast: only matrix multiplications



Attention Tricks

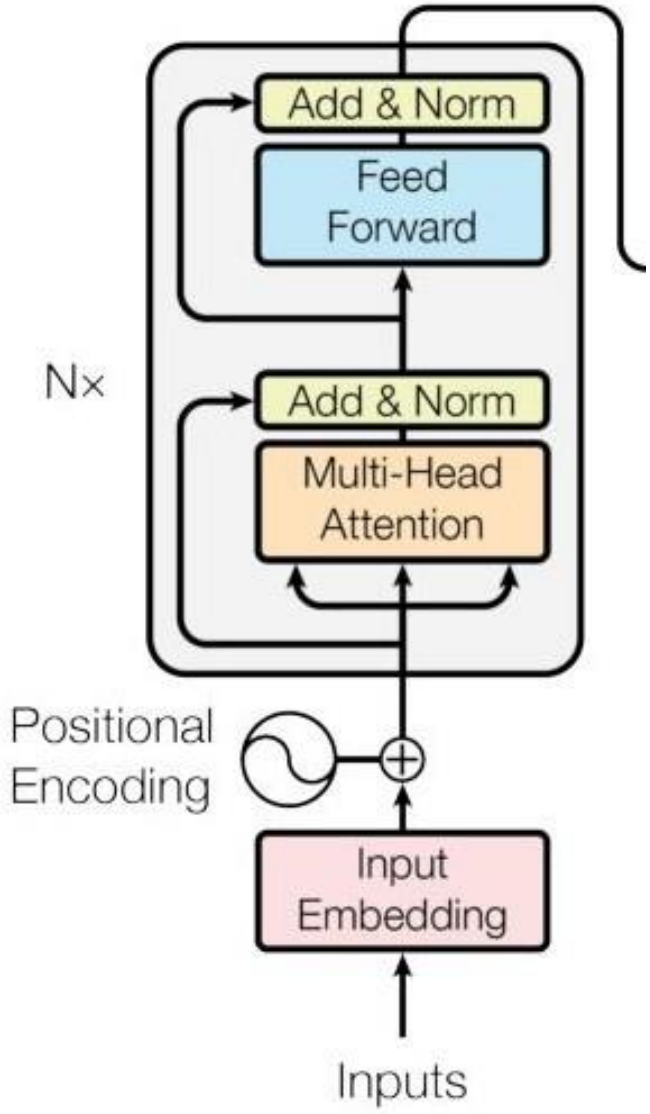
- **Self Attention:** Each layer combines words with others
- **Multi-headed Attention:** 8 attention heads learned independently
- **Normalized Dot-product Attention:** Remove bias in dot product when using large networks
- **Positional Encodings:** Make sure that even if we don’t have RNN, can still distinguish positions

Self-Attention: focus on the important parts.



Model: Encoder

- N=6
- All Layers output size 512
- Embedding
- Positional Encoding
- Notice the Residual connection
- Multi-head Attention
- LayerNorm(x +Sublayer(x))
- Position wise feed forward



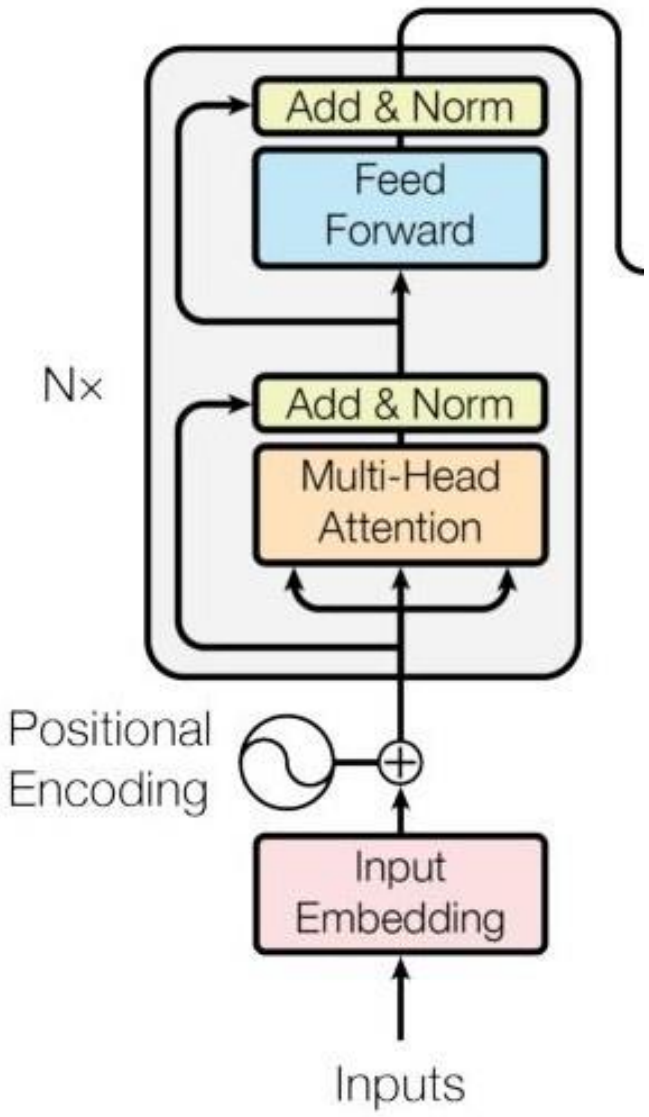
Model: Encoder

- N=6
- All Layers output size 512
- Embedding
- Positional Encoding

- Multi-head Attention
- LayerNorm(x +Sublayer(x))
- Position wise feed forward

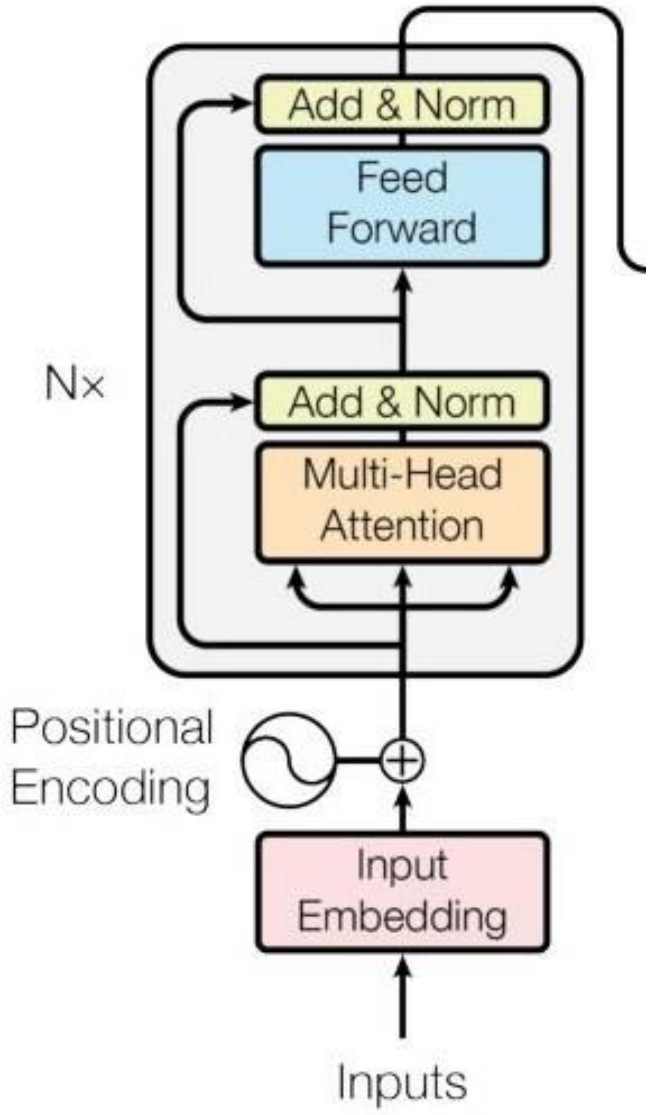
$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

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



Model: Encoder

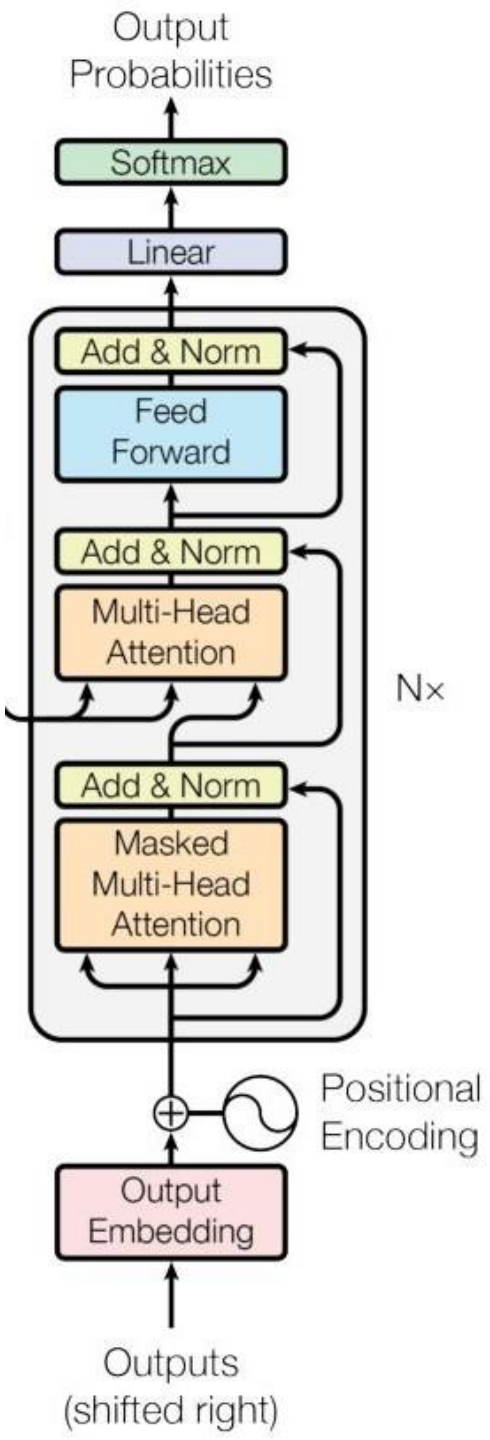
- N=6
- All Layers output size 512
- Embedding
- Positional Encoding
- Notice the Residual connection
- Multi-head Attention
- LayerNorm(x +Sublayer(x))
- Position wise feed forward



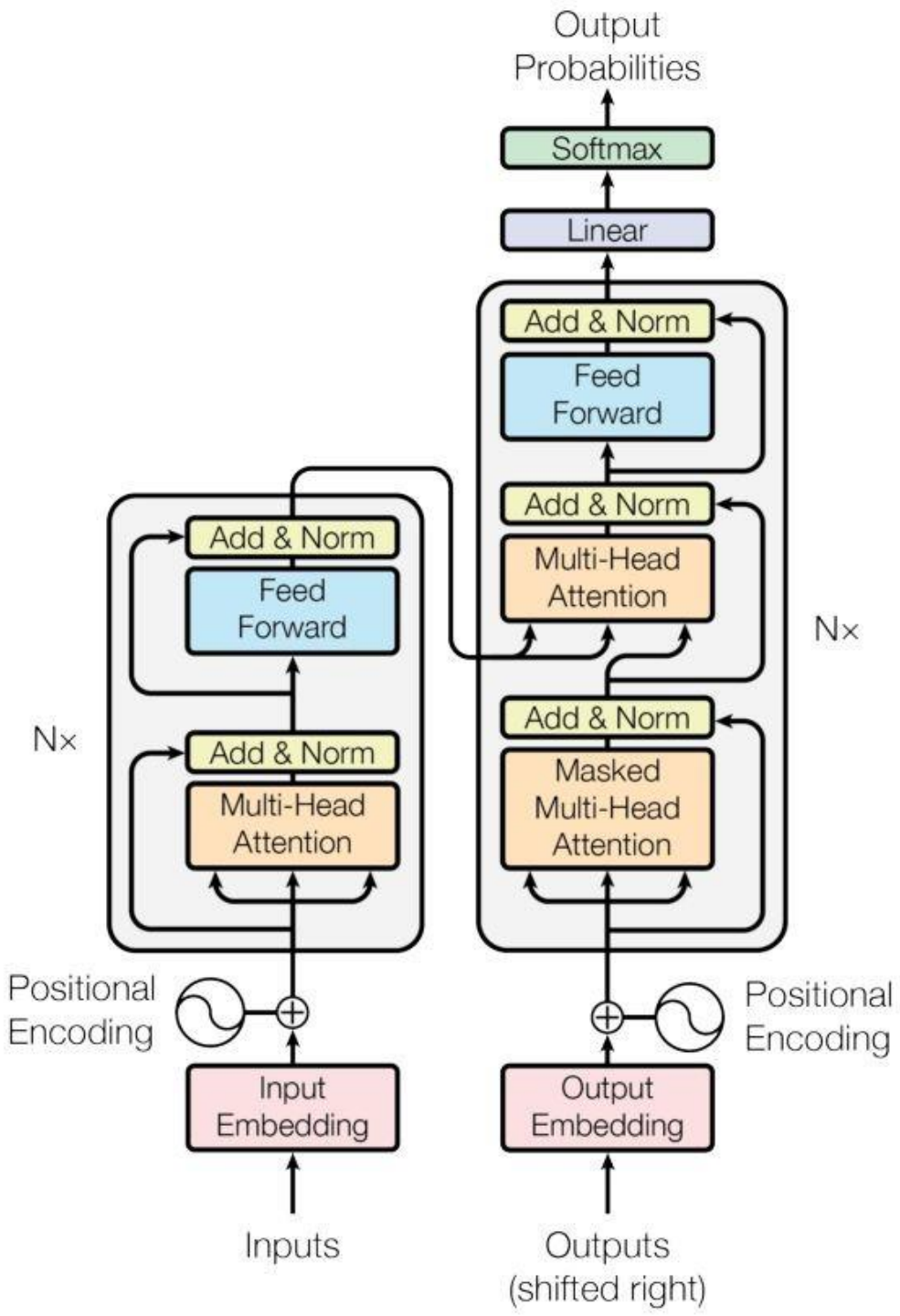
Model: Decoder

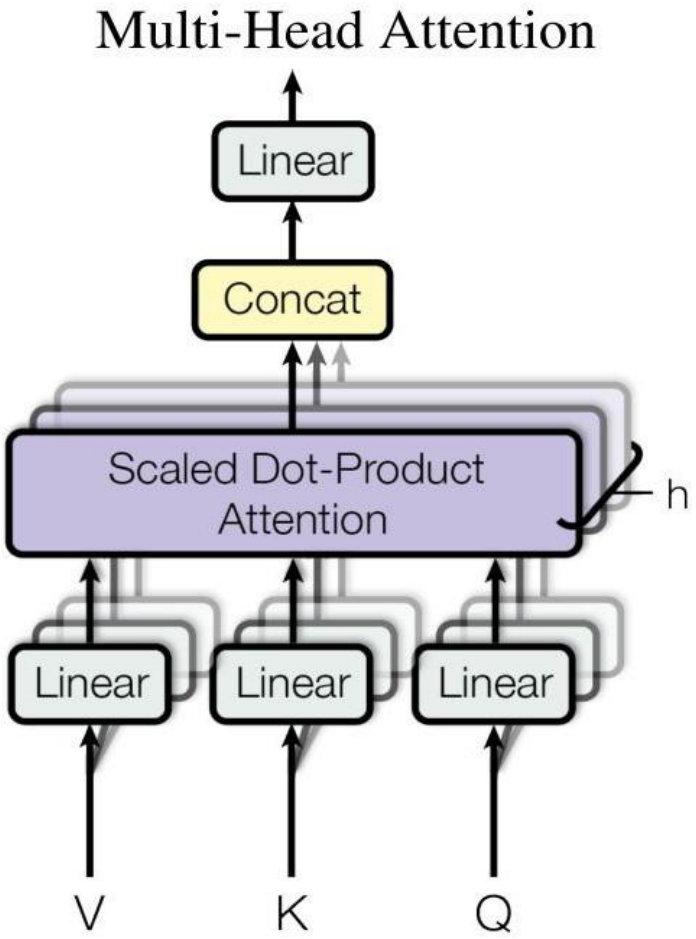
- N=6
- All Layers output size 512
- Embedding
- Positional Encoding
- Notice the Residual connection
- Multi-head Attention
- LayerNorm(x +Sublayer(x))
- Position wise feed forward

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$



Model: Complete

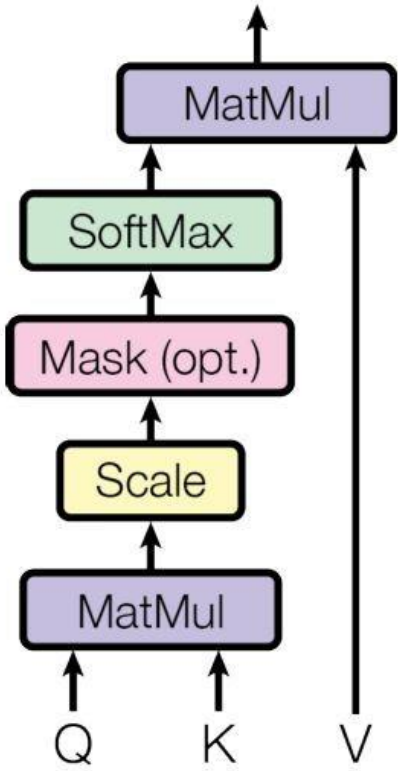




$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$

$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

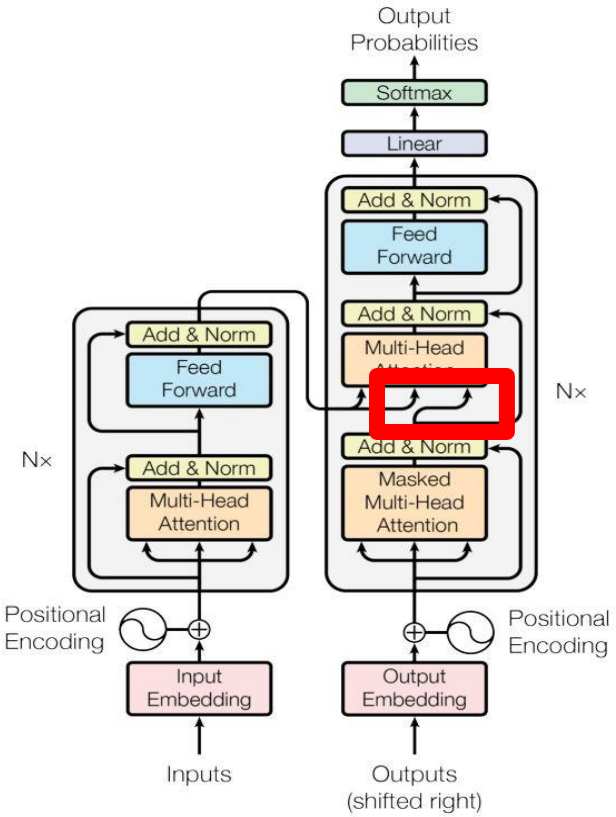
Scaled Dot-Product Attention



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

Q,K,V

- "encoder-decoder attention" layers, the queries (Q) come from the previous decoder layer, and the memory keys (K) and values (V) come from the output of the encoder.”
- Otherwise: all three come from previous layer (Hidden state)



https://www.reddit.com/r/MachineLearning/comments/6kc7py/d_where_does_the_query_keys_and_values_come_from/

Transformer Block

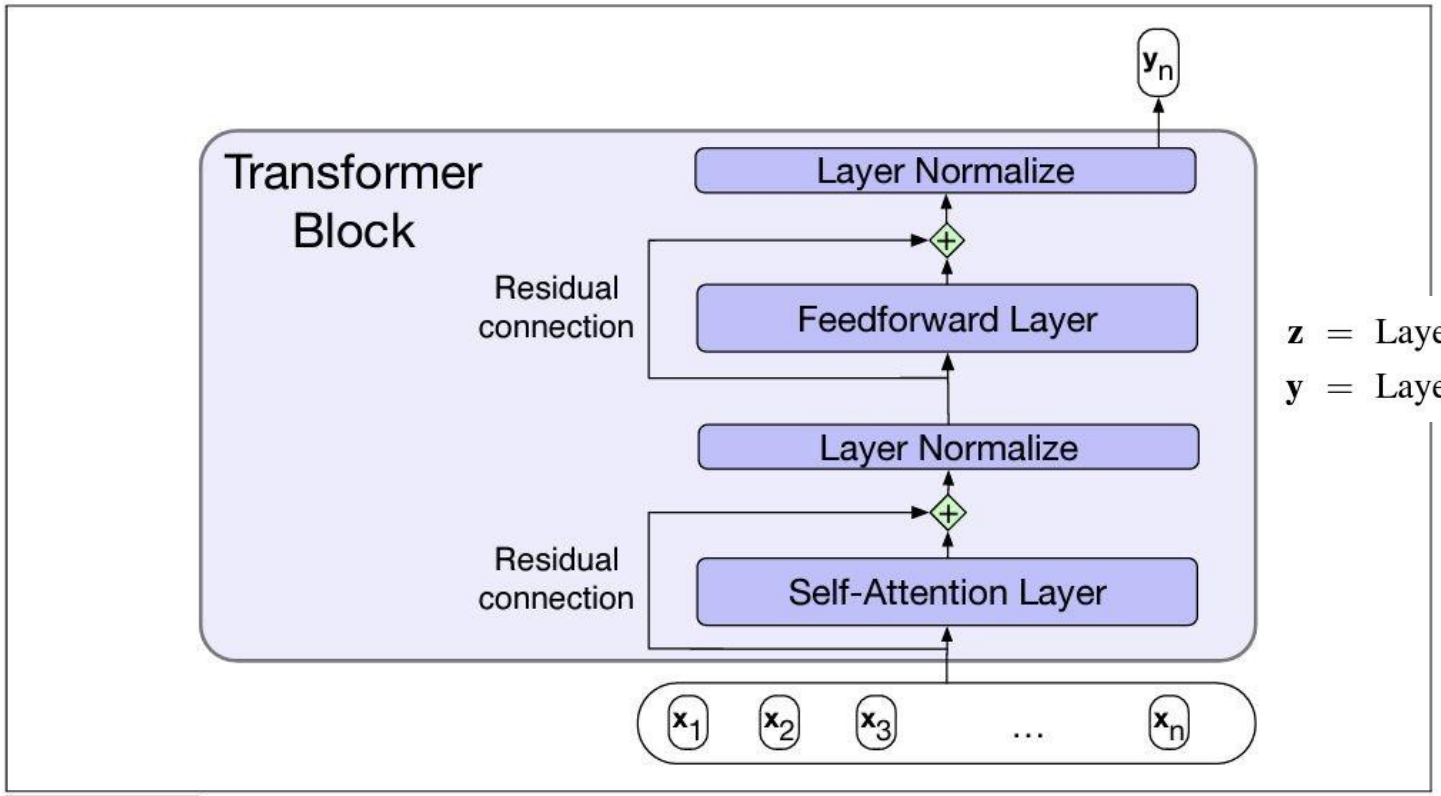


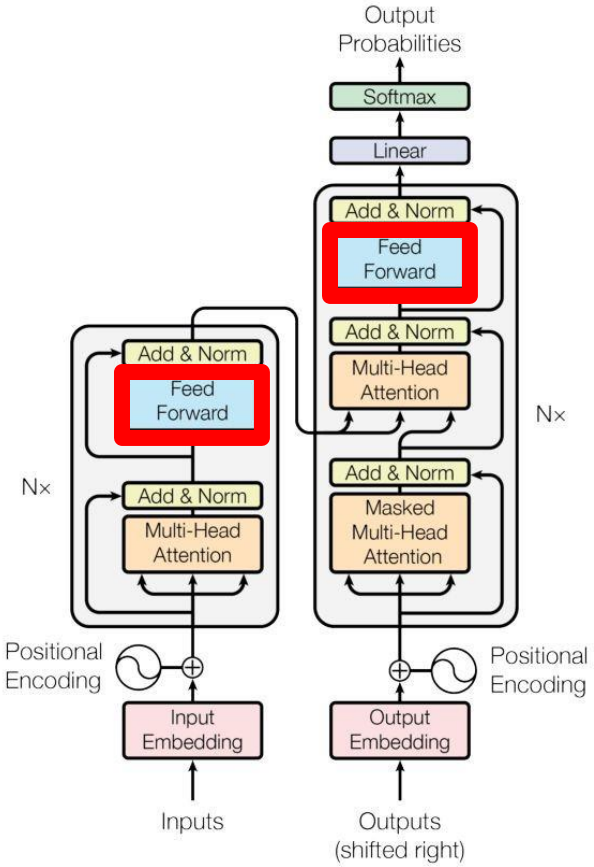
Figure 10.4 A transformer block showing all the layers.

Complexity

Table 1: Maximum path lengths, per-layer complexity and minimum number of sequential operations for different layer types. n is the sequence length, d is the representation dimension, k is the kernel size of convolutions and r the size of the neighborhood in restricted self-attention.

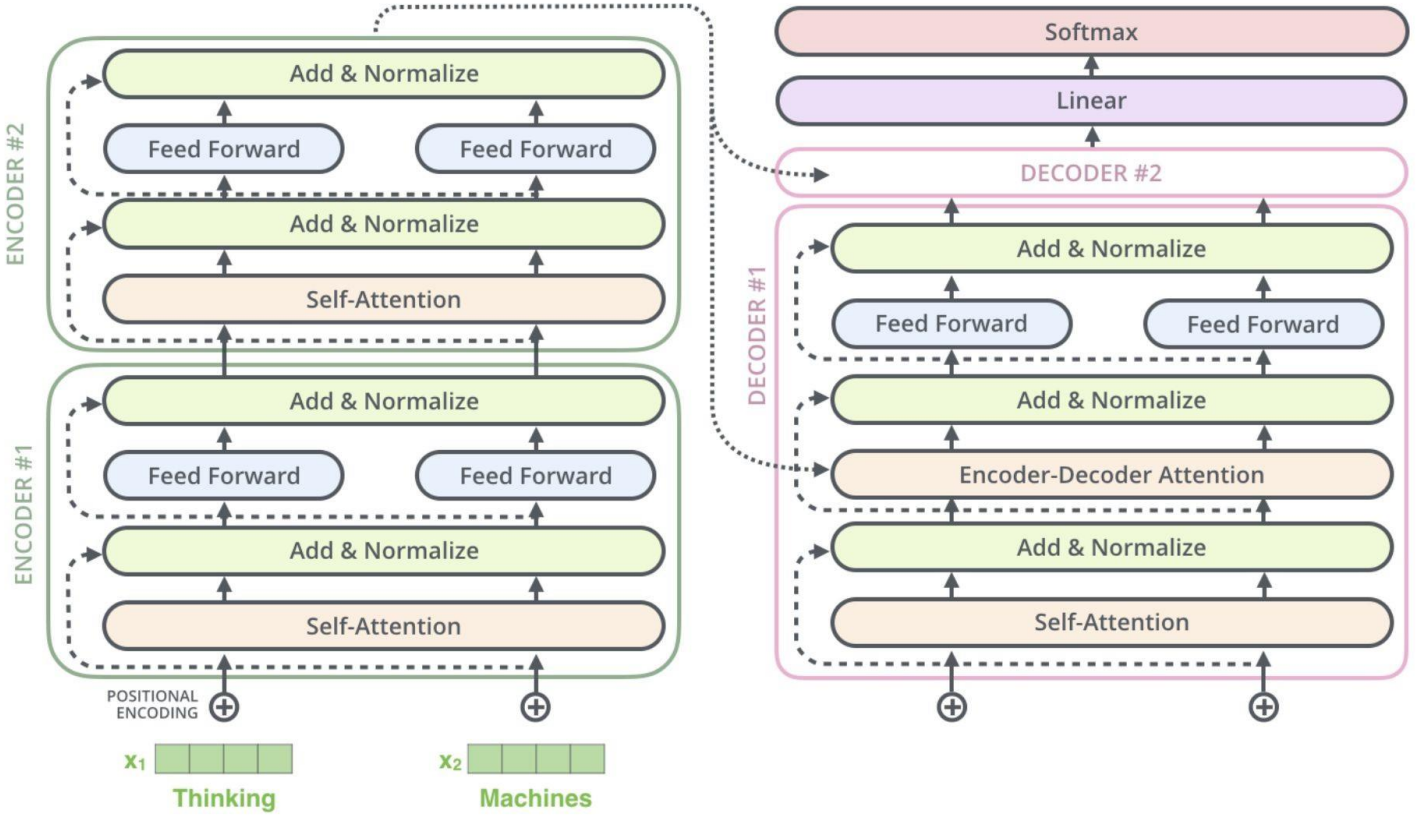
Layer Type	Complexity per Layer	Sequential Operations	Maximum Path Length
Self-Attention	$O(n^2 \cdot d)$	$O(1)$	$O(1)$
Recurrent	$O(n \cdot d^2)$	$O(n)$	$O(n)$
Convolutional	$O(k \cdot n \cdot d^2)$	$O(1)$	$O(log_k(n))$
Self-Attention (restricted)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

Position-wise Feed-Forward network

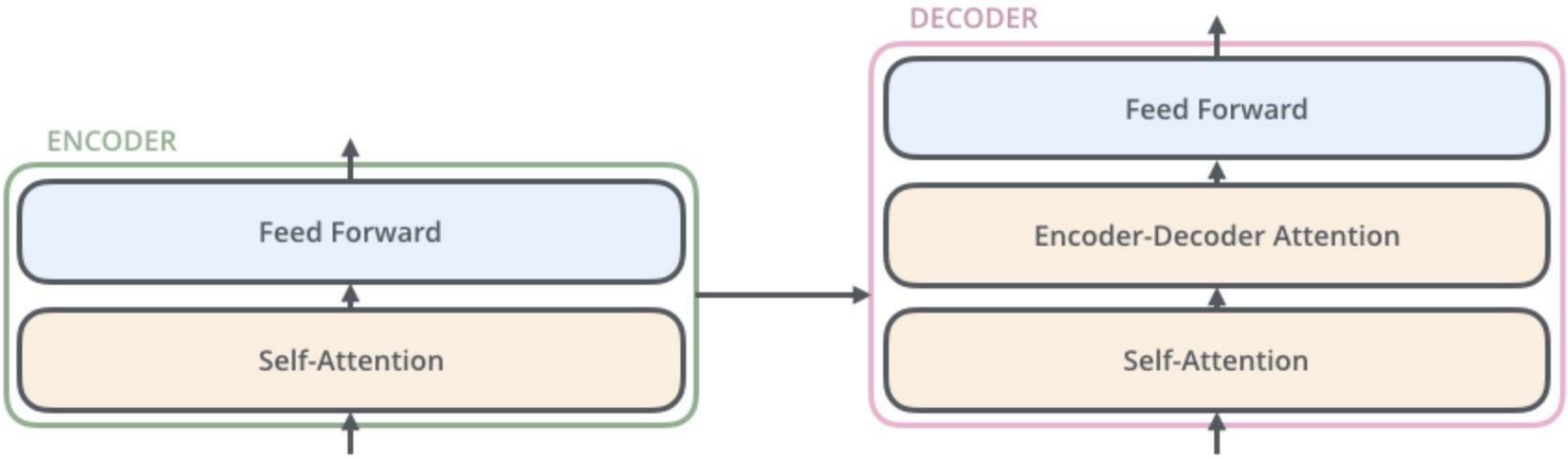


$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

Transformer with 2 stacked encoders and decoders



How is the decoder different?



Masked Self-attention for decoder (to avoid seeing the future tokens)

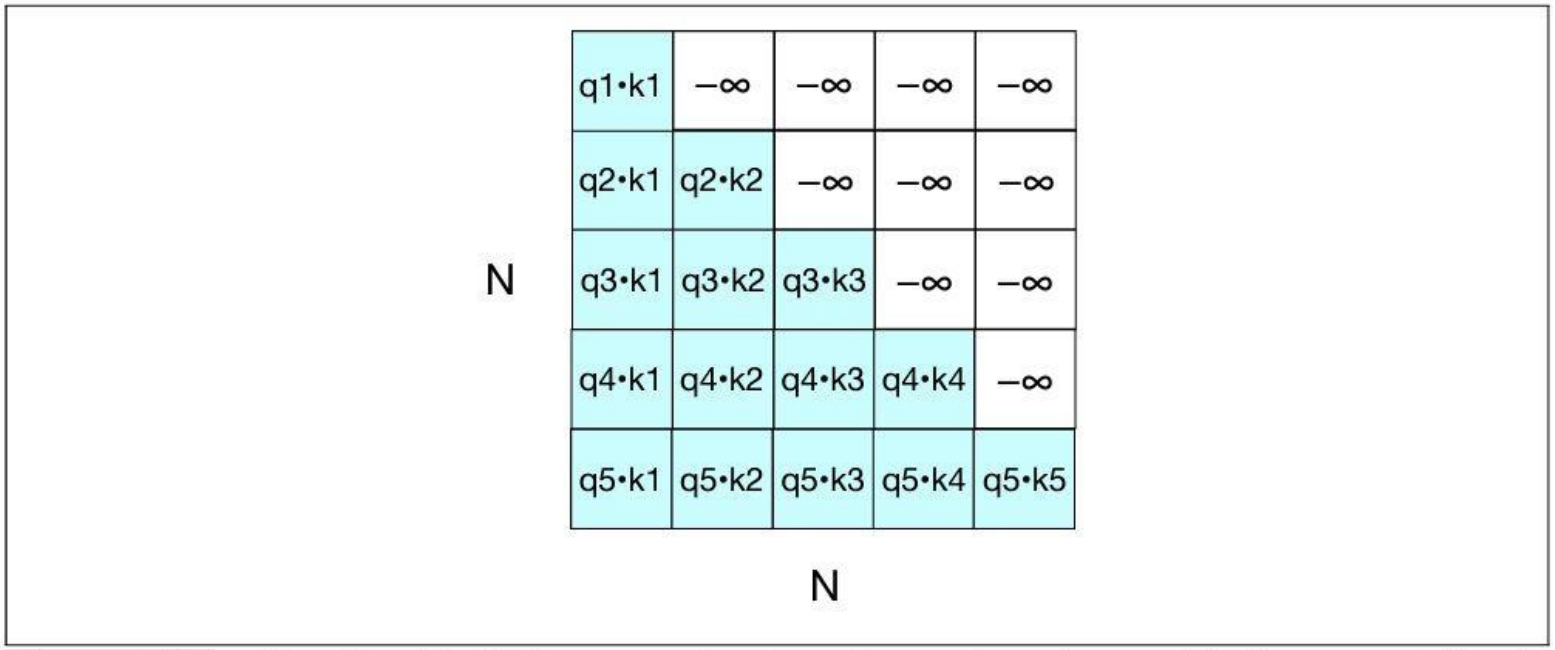
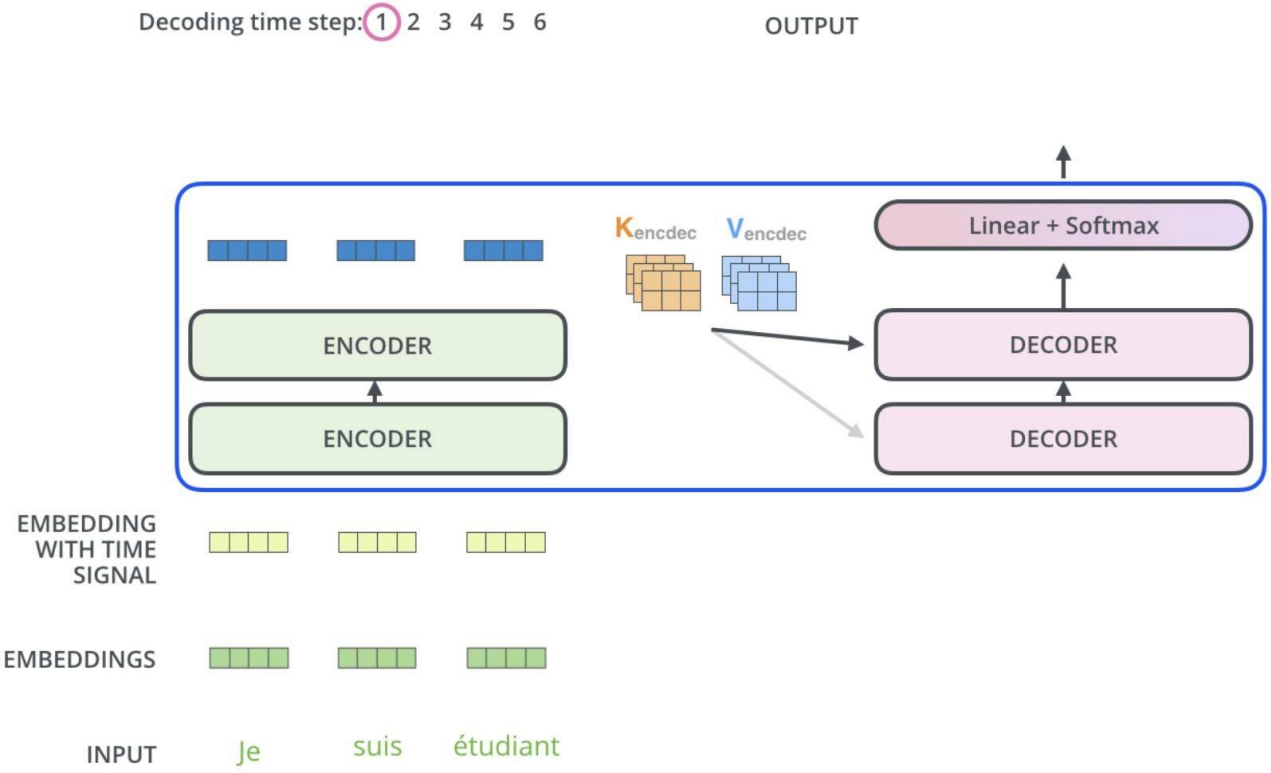


Figure 10.3 The $N \times N$ $\mathbf{QK}^\intercal$ matrix showing the $q_i \cdot k_j$ values, with the upper-triangle portion of the comparisons matrix zeroed out (set to $-\infty$, which the softmax will turn to zero).

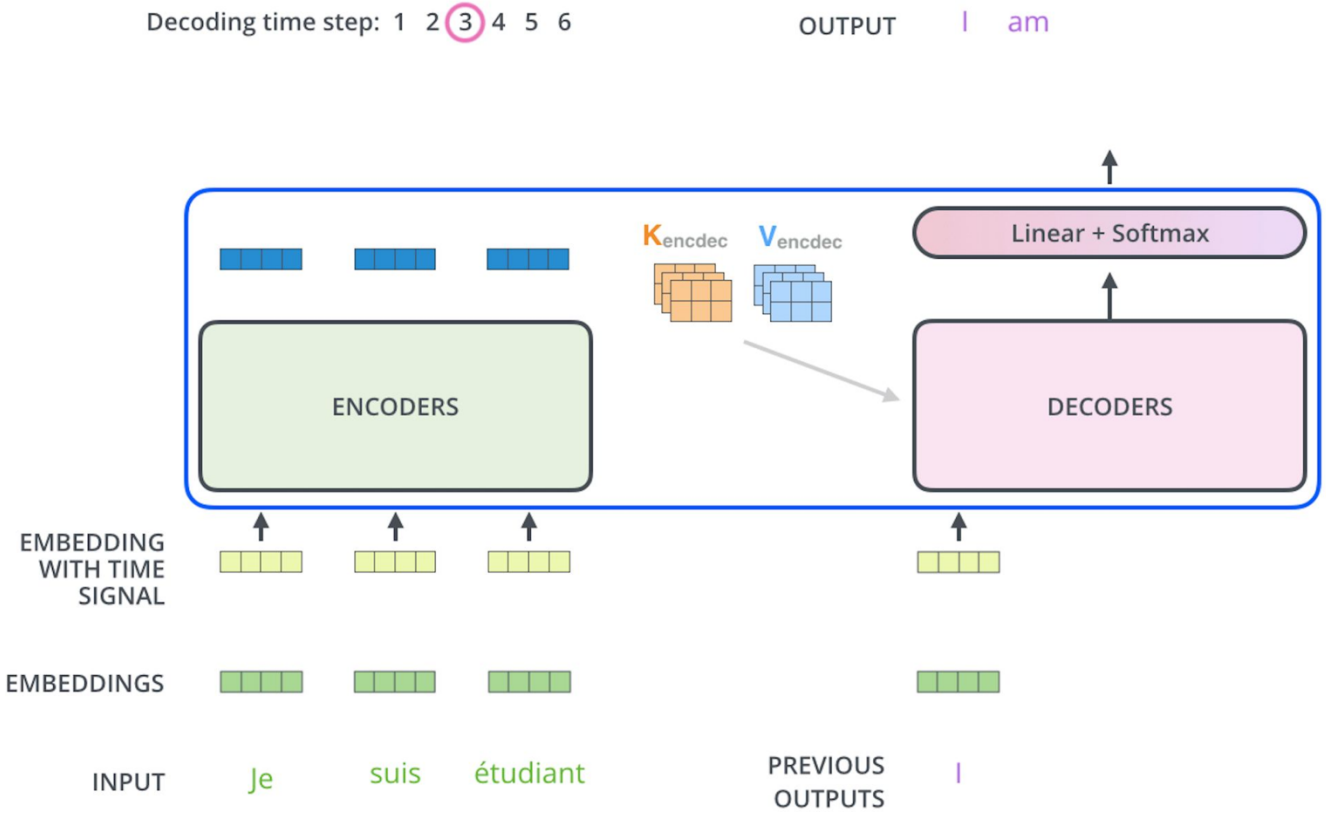
Encoder-Decoder Attention



The output of the top encoder is transformed into a set of attention vectors K and V.

These are to be used by each decoder in its "***encoder-decoder attention***" layer which helps the decoder focus on appropriate places in the input sequence:
It creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the output of the encoder stack.

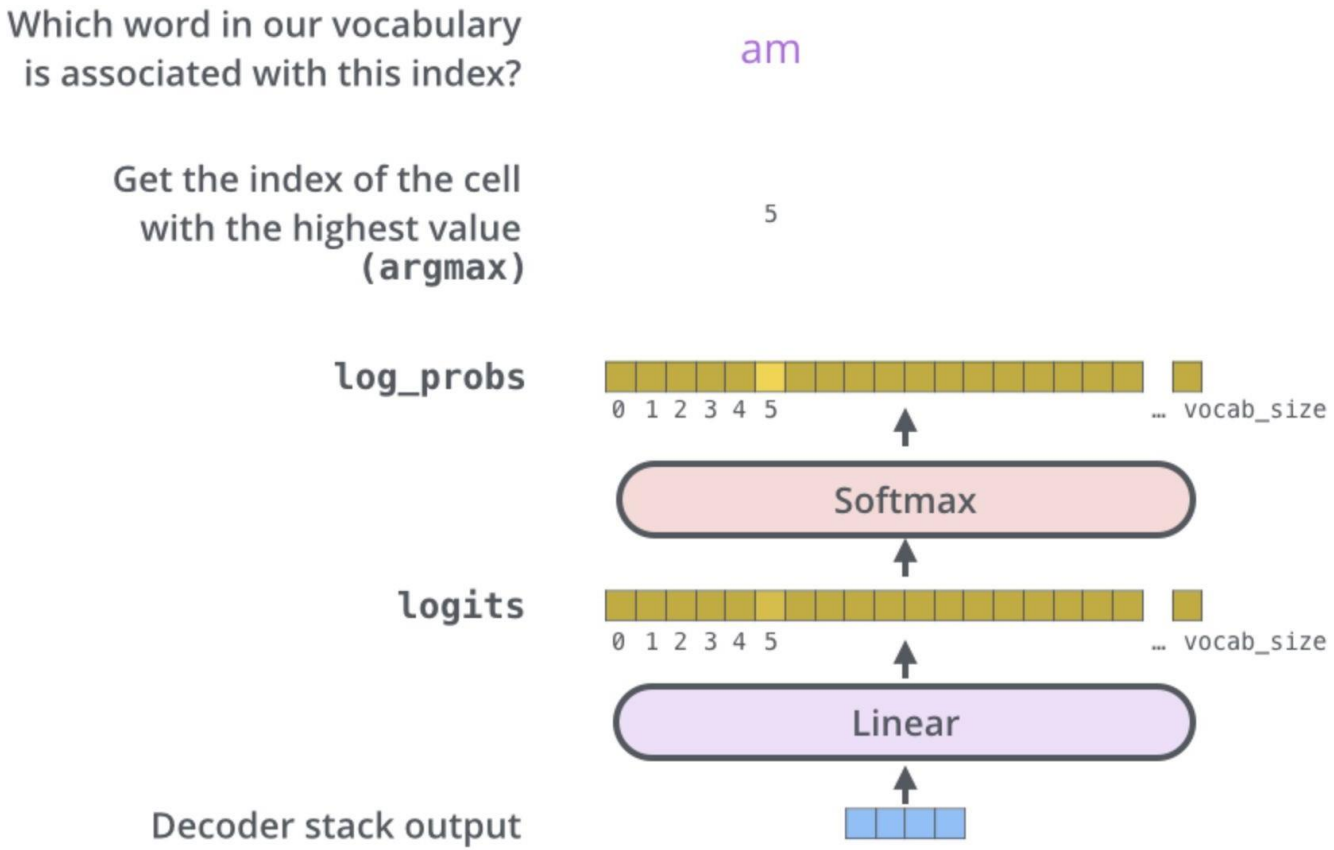
Decoding



The output of each step is fed to the bottom decoder in the next time step, and the decoders bubble up their decoding results just like the encoders did.

And just like we did with the encoder inputs, we embed and add positional encoding to those decoder inputs to indicate the position of each word.

Converting decoder stack output to words



That's the job of the final ***Linear layer*** which is followed by a ***Softmax Layer***.

The Linear layer is a simple fully connected neural network that projects the vector produced by the stack of decoders, into a much, much larger vector called a logits vector.

The softmax layer then turns those scores into probabilities. The cell with the highest probability is chosen.

Training

- Data sets:
  - WMT 2014 English-German: 4.5 million sentences pairs with 37K tokens.
  - WMT 2014 English-French: 36M sentences, 32K tokens.
- Hardware:
  - 8 Nvidia P100 GPus (Base model 12 hours, big model 3.5 days)

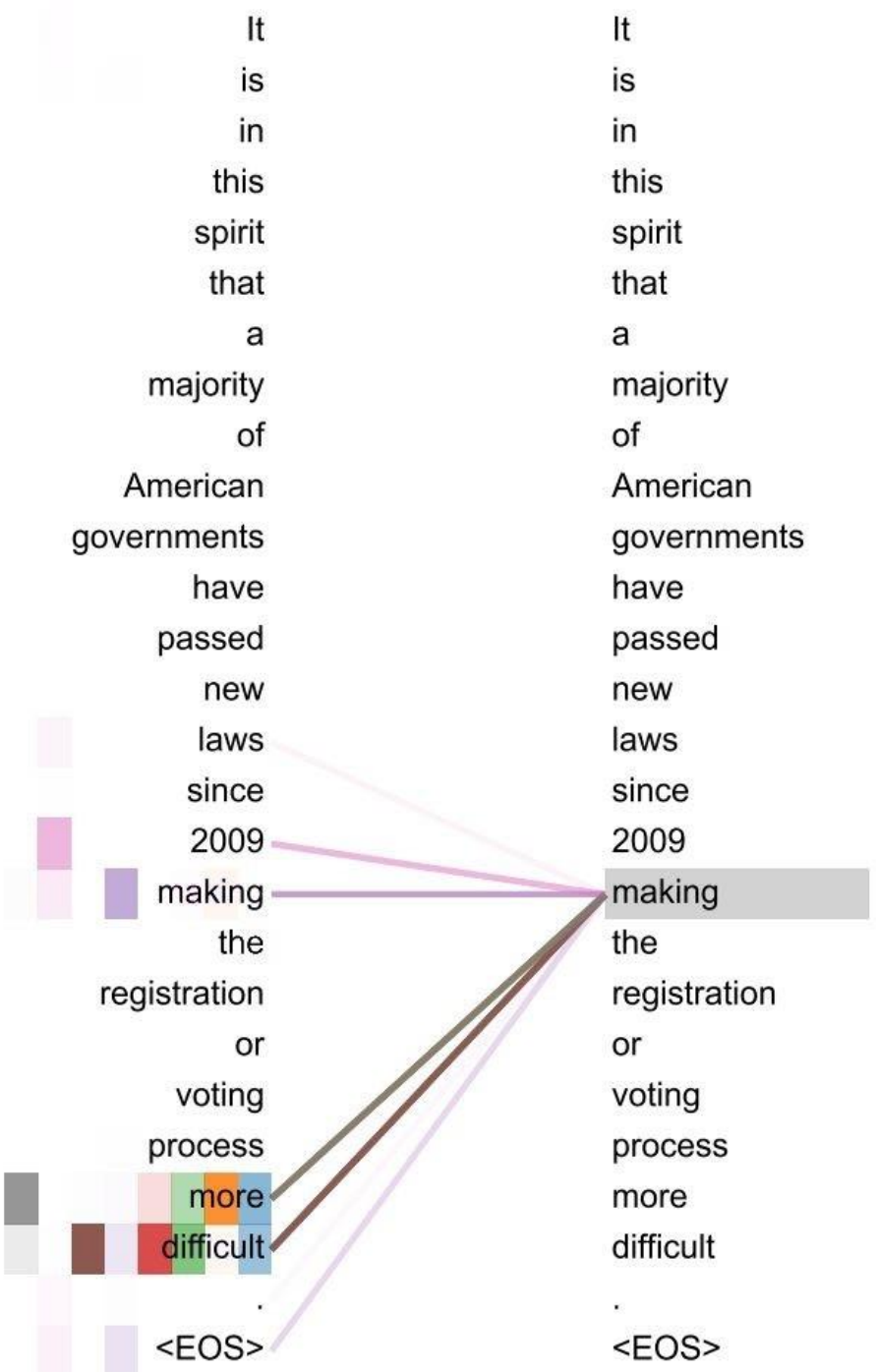
http://www.prioritiesusa.org/recommendations-for-sports-training/

Results

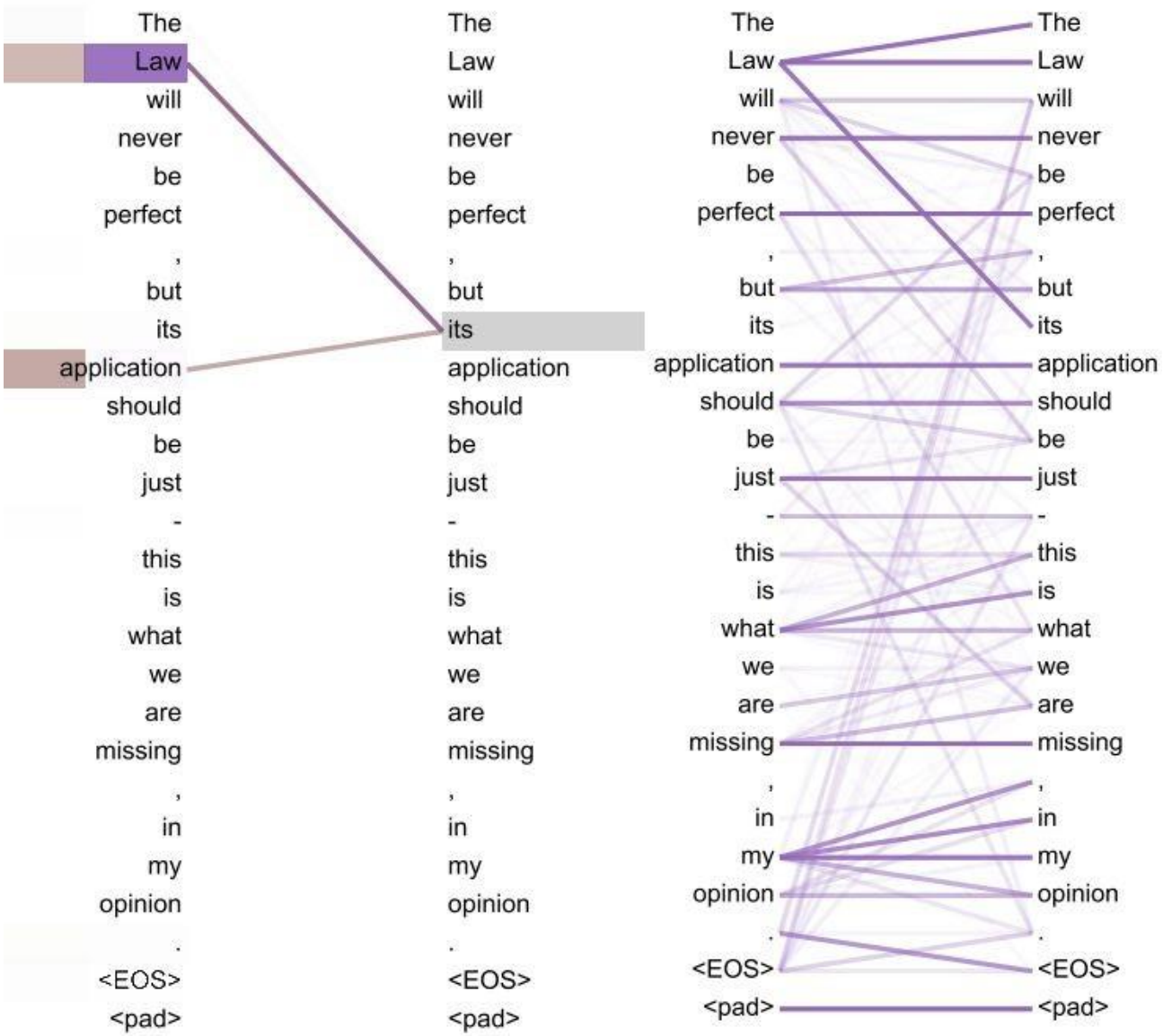
Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [17]	23.75			
Deep-Att + PosUnk [37]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [36]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [31]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [37]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [36]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.0**	$2.3 \cdot 10^{19}$	

Results


It
is
in
this
spirit
that
a
majority
of
American
governments
have
passed
new
laws
since
2009
making
the
registration
or
voting
process
more
difficult
.
<EOS>
It
is
in
this
spirit
that
a
majority
of
American
governments
have
passed
new
laws
since
2009
making
the
registration
or
voting
process
more
difficult
.
<EOS>

Results


The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>
The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>
The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>
The
Law
will
never
be
perfect
,
but
its
application
should
be
just
-
this
is
what
we
are
missing
,
in
my
opinion
.
<EOS>
<pad>

Conclusion

- Deep learning approaches – Powerful mechanisms for introducing non-linearity in learning
- Learning using backpropagation
- Embeddings for word representations
- Sequence Labelling using RNNs
- LSTMs, GRUs are special kind of RNNs
- CNNs for text and Image recognition.

References

- **Deep Learning for NLP - Nils Reimers.** https://github.com/UKPLab/deeplearning4nlp-tutorial/tree/master/2017-07_Seminar
- CS231n: Convolutional Neural Networks for Visual Recognition. Andrej Karpathy http://cs231n.github.io/convolutional-networks/
- http://karpathy.github.io/2015/05/21/rnn-effectiveness/
- **Neural Networks for Information Retrieval**. SIGIR 2017 Tutorial http://nn4ir.com/
- **CSE 446 - Machine Learning - Spring 2015,** University of Washington. Pedro Domingos. https://courses.cs.washington.edu/courses/cse446/15sp/