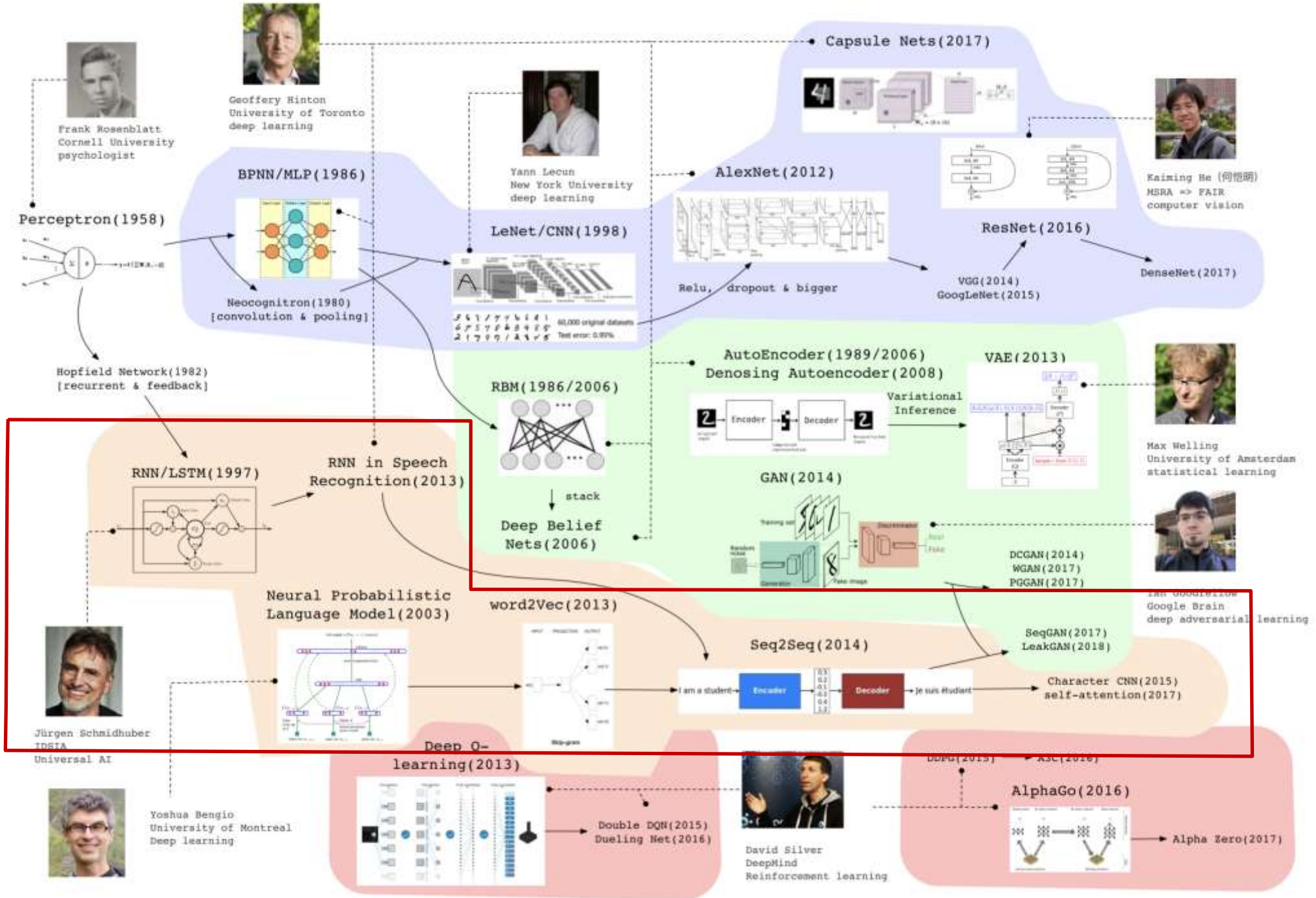
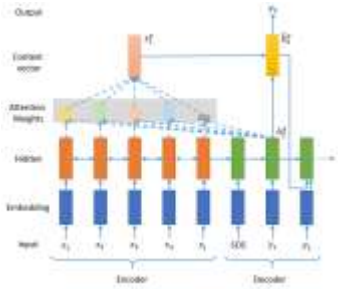
Transformer Basics

Jie Tang
Department of Computer Science & Technology
Tsinghua University


Capsule Nets(2017)
Frank Rosenblatt
Cornell University
psychologist
Geoffery Hinton
University of Toronto
deep learning
Yann Lecun
New York University
deep learning
Kaiming He (何恺明)
MSRA => FAIR
computer vision
BPNN/MLP(1986)
AlexNet(2012)
Perceptron(1958)
LeNet/CNN(1998)
ResNet(2016)
Neocognitron(1980)
[convolution & pooling]
Relu, dropout & bigger
VGG(2014)
GoogLeNet(2015)
DenseNet(2017)
Hopfield Network(1982)
[recurrent & feedback]
AutoEncoder(1989/2006)
Denosing Autoencoder(2008)
VAE(2013)
Variational
Inference
Encoder
Decoder
RBM(1986/2006)
Max Welling
University of Amsterdam
statistical learning
RNN/LSTM(1997)
RNN in Speech
Recognition(2013)
GAN(2014)
stack
Deep Belief
Nets(2006)
DCGAN(2014)
WGAN(2017)
PGGAN(2017)
Google Brain
deep adversarial learning
Neural Probabilistic
Language Model(2003)
word2Vec(2013)
SeqGAN(2017)
LeakGAN(2018)
Seq2Seq(2014)
I am a student
Encoder
Decoder
Je suis étudiant
Character CNN(2015)
self-attention(2017)
Jürgen Schmidhuber
IDSIA
Universal AI
Deep Q-
learning(2013)
Yoshua Bengio
University of Montreal
Deep learning
Double DQN(2015)
Dueling Net(2016)
David Silver
DeepMind
Reinforcement learning
AlphaGo(2016)
Alpha Zero(2017)

Transformer History



2014

Attention Seq2Seq

OpenAI

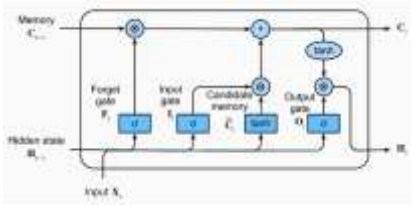
2018

GPT / Pretraining

Jürgen Schmidhuber
IDSIA
Universal AI

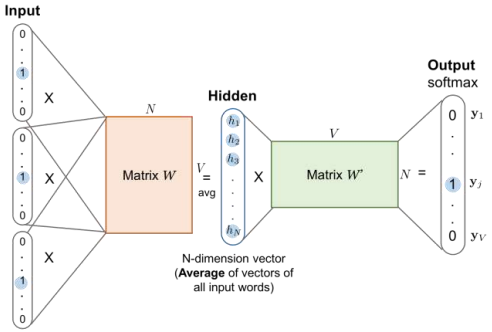
1997

LSTM



2013

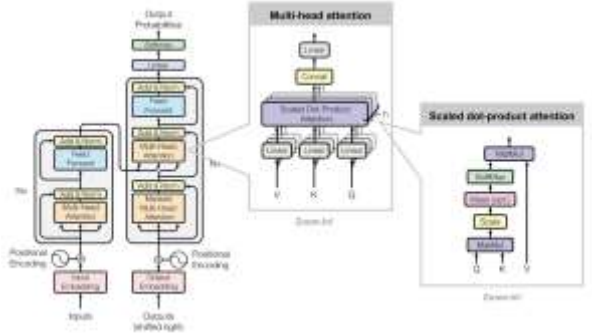
Word2Vec



Google

2017

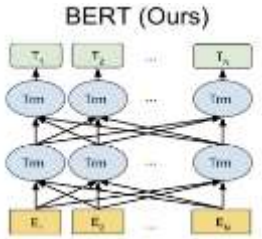
Transformer



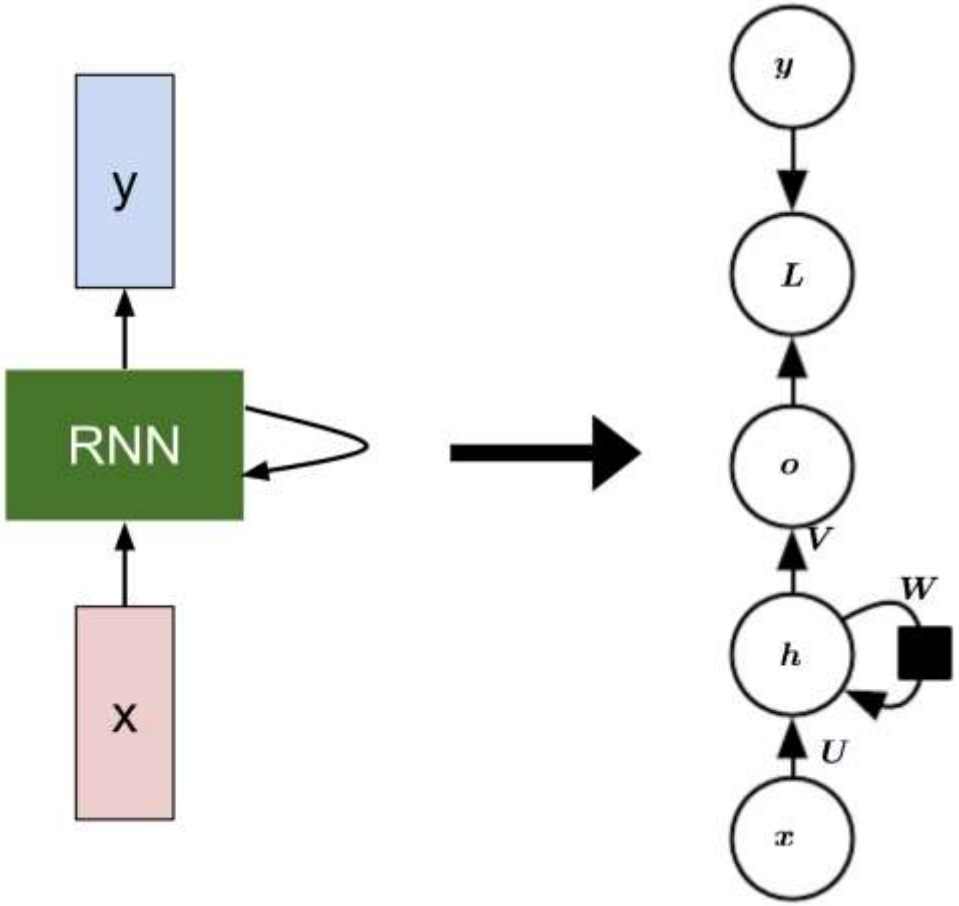
Jacob Devlin

2018

BERT



Recurrent Neural Network (RNN)



- Input x and output y are from the training data
- o is the estimated output value
- L is to evaluate the difference (loss) between the estimated output $\hat{y}$ = softmax(o) and the true output y
- U, W, V are three weigh matrices

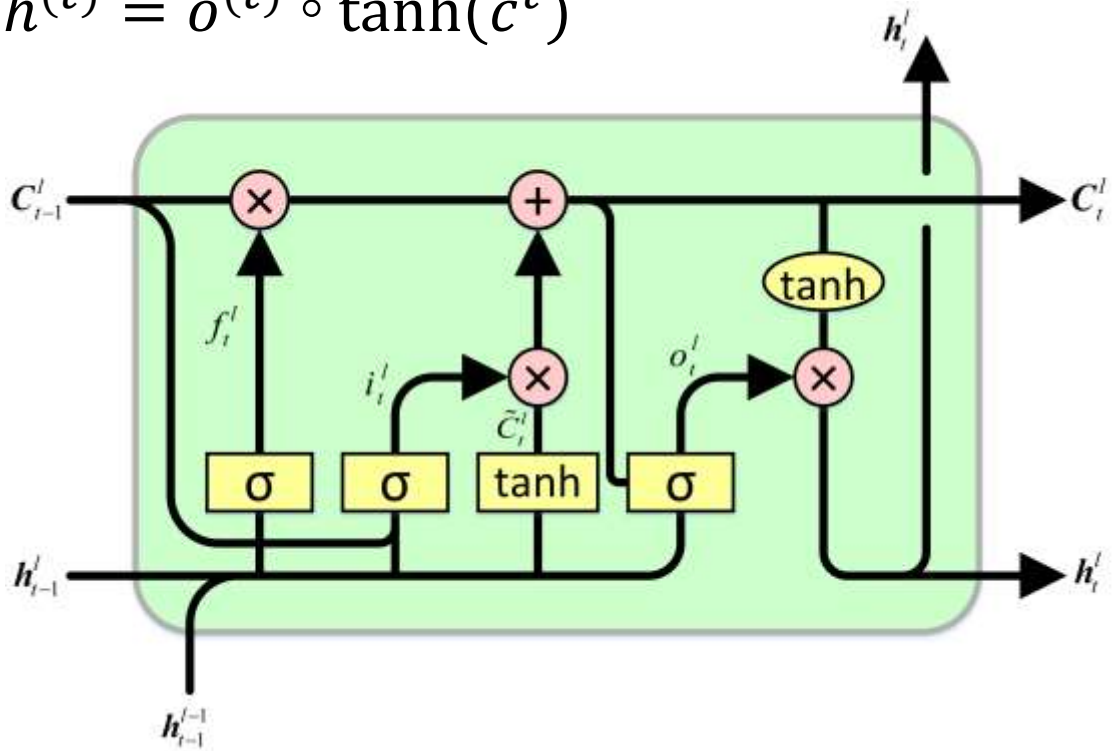
$P(y_i \mid x_i) \rightarrow P(\bar{y} \mid \bar{x})$, where $y_i \in \bar{y}$ and $x_i \in \bar{x}$

Summary

	Long-Term Dependency	Parallelization	Computation	Flexibility for Length	Gradient Flow	Complexity
RNN	Poor, due to recurrent structure	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for input, fixed for output	Problems with vanishing/exploding gradients	Simple, few hyperparameters

Long Short-Term Memory (LSTM)

- Forget Gate: $f^{(t)} = sigmoid(W^f h^{(t-1)} + U^f x^{(t)} + b^f)$
- Input Gate: $i^{(t)} = sigmoid(W^i h^{(t-1)} + U^i x^{(t)} + b^i)$
- Output Gate: $o^{(t)} = sigmoid(W^o h^{(t-1)} + U^o x^{(t)} + b^o)$
- Cell State Vector: $c^{(t)} = f^{(t)} \circ c^{(t-1)} + i^{(t)} \circ \tanh(W^c h^{(t-1)} + U^c x^{(t)} + b^c)$
- Final Hidden State: $h^{(t)} = o^{(t)} \circ \tanh(c^t)$



Gated Recurrent Units (GRU)

- Update Gate: $z^{(t)} = sigmoid(W^z h^{(t-1)} + U^z x^{(t)} + b^z)$
- Reset Gate: $r^{(t)} = sigmoid(W^r h^{(t-1)} + U^r x^{(t)} + b^r)$
- New Memory Content:

$$\hat{h}^{(t)} = \tanh(W(r^{(t)} \circ h^{(t-1)}) + Ux^{(t)} + b)$$

- Final Memory: $h^{(t)} = z^{(t)} \circ h^{(t-1)} + (1 - z^{(t)}) \circ \hat{h}^{(t)}$

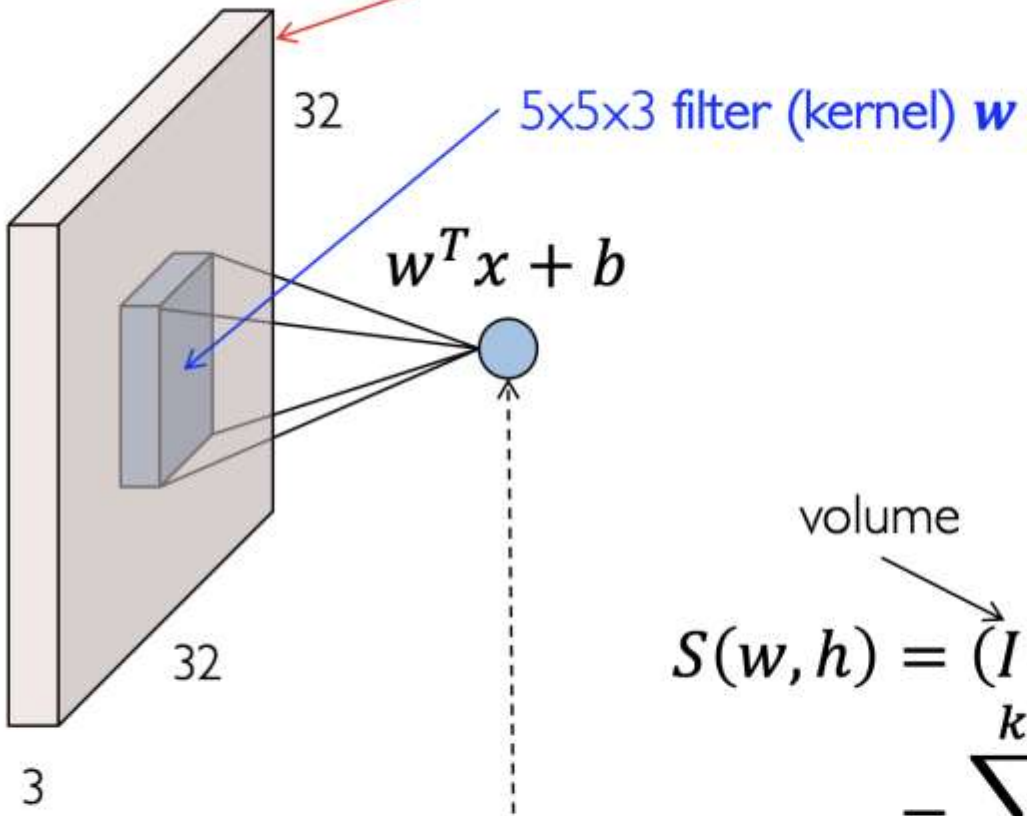
LSTM vs GRU

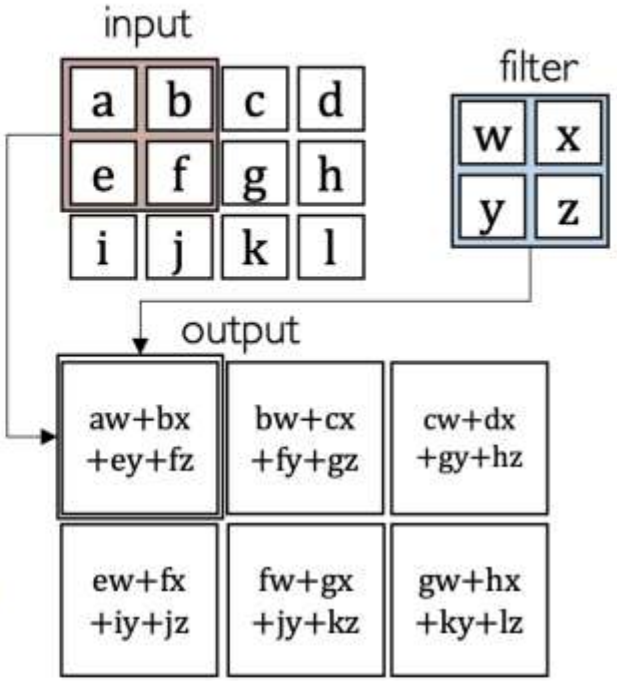
- GRUs achieve similar performance in a variety of tasks compared to LSTM.
- GRUs train faster than LSTMs on less training data.
- LSTM outperform GRUs in tasks requiring modeling long-distance relations.

Summary

	Long-Term Dependency	Parallelization	Computation	Flexibility for Length	Gradient Flow	Complexity
RNN	Poor, due to recurrent structure	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for input, fixed for output	Problems with vanishing/exploding gradients	Simpler, fewer hyperparameters
LSTM	**Better**, still rooms to improve	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for input, fixed for output	**Better gradient flow with less vanishing**	Complex, many hyperparameters

Convolution Neural Network (CNN)





volume filter

$$S(w,h) = (I \star K)(w,h)$$

$$= \sum_{i=1}^{k} \sum_{j=1}^{k} I(w+i, h+j) K(i,j)$$

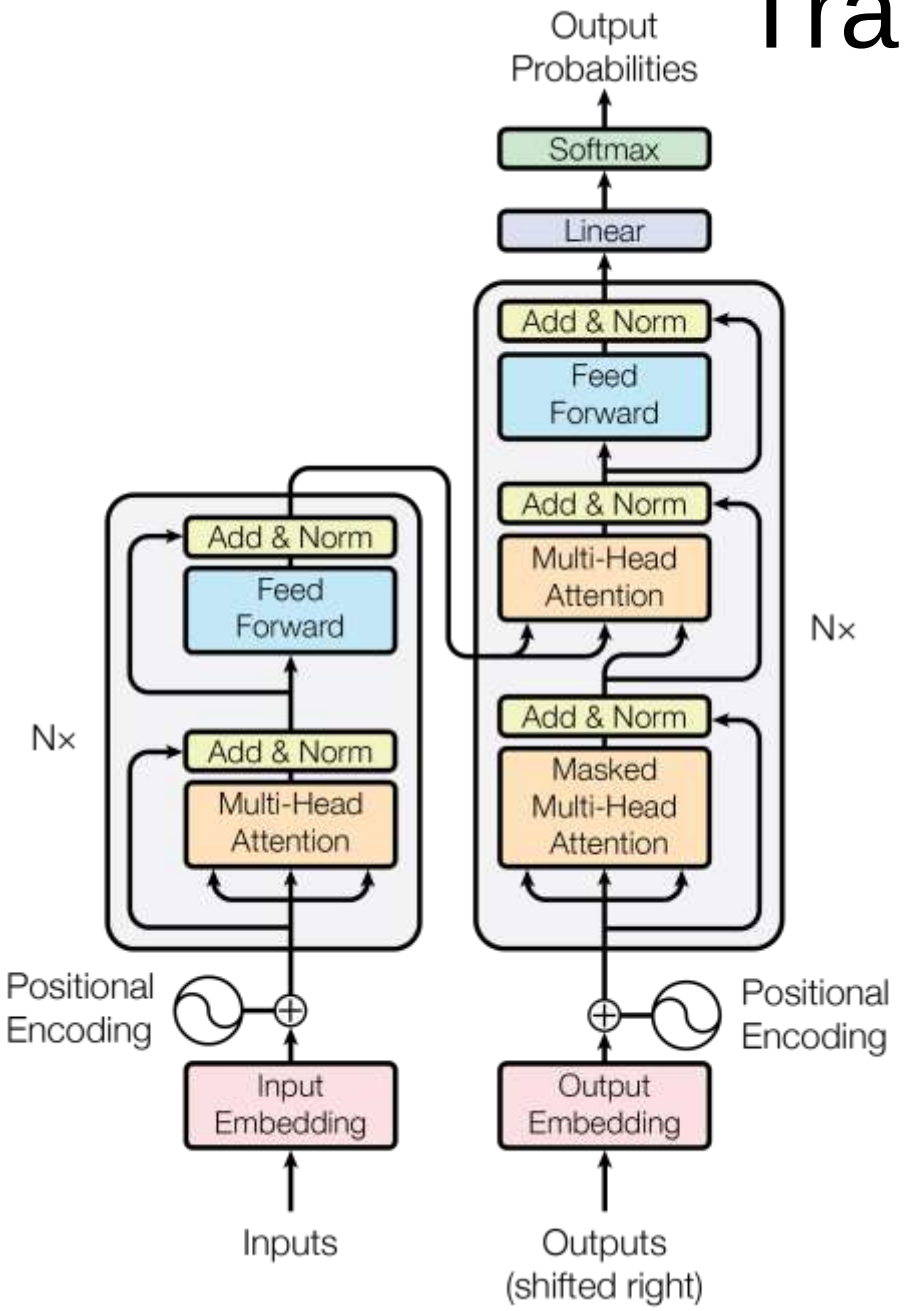
- The output of each neuron:
  - Dot product between filter and a small 5x5x3 chunk of the volume

Summary

	Long-Term Dependency	Parallelization	Computation	Flexibility for Length	Gradient Flow	Complexity
RNN	Poor, due to recurrent structure	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for input, fixed for output	Problems with vanishing/exploding gradients	Simpler, fewer hyperparameters
LSTM	Better, still rooms to improve	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for input, fixed for output	Better gradient flow with less vanishing	Complex, many hyperparameters
CNN	**Not inherent**	**Highly parallelizable**	**Fast** due to parallelizable operations	**Less flexible**, typically requires fixed-size inputs	**Stable gradient flow**	Complex, many hyperparameters

Table of Contents

Transformer w/ NLP



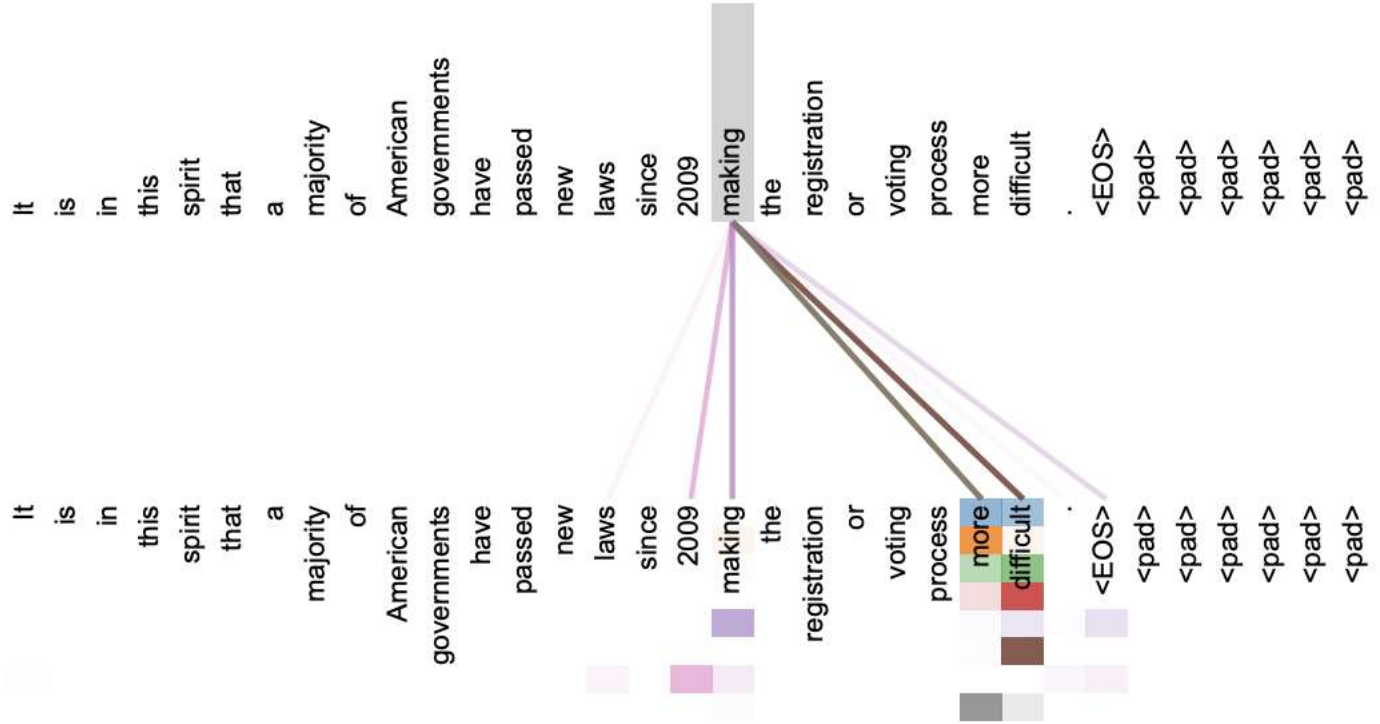


Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb 'making', completing the phrase 'making...more difficult'. Attentions here shown only for the word 'making'. Different colors represent different heads. Best viewed in color.

Tokenization

- A language model is a probability distribution over a **sequence of tokens** where each token comes from some vocabulary:

  the mouse, ate, the, cheese

- However, natural language doesn't come as a sequence of tokens, but as just a string (concretely, sequence of Unicode characters):

  the mouse ate the cheese

- A **tokenizer** converts any string into a sequence of tokens.

  the mouse ate the cheese⇒[the, mouse, ate, the, cheese]

Simplest Solution: Split by Spaces

- The simplest solution is to split by spaces:

  text.split(‘’)

- Cons:

  1. This doesn’t work for languages such as Chinese, where sentences are written without spaces between words:

     我今天去了商店。 [gloss: *I went to the store.*]

  2. Then there are languages like German that have long compound words (e.g., *Abwasserbehandlungsanlange*).

Simplest Solution: Split by Spaces

- The simplest solution is to split by spaces:

  text.split(‘’)

- Cons:

  3. Even in English, there are hyphenated words (e.g., *father-in-law*) and contractions (e.g., *don’t*), which should get split up. For example, the Penn Treebank splits *don’t* into *do* and *n’t*, a linguistically-informed but not obvious choice.

- Therefore, splitting by spaces by spaces to identify words is quite problematic.

What makes a good tokenization?

- We don't want too **many** tokens (extreme: characters or bytes)
  - or else the sequence becomes difficult to model.
- We don't want too **few** tokens
  - or else there won't be parameter sharing between words (e.g., should *mother-in-law* and *father-in-law* be completely different)?
  - This is especially problematic for morphologically rich languages (e.g., Arabic, Turkish, etc.).
- Each token should be a linguistically or statistically meaningful unit.

Byte Pair Encoding (BPE) Algorithm

- Sennrich et al, 2015 applied the **byte pair encoding (BPE)** algorithm, originally developed for data compression, to produce one of the most commonly used tokenizers.
- **Learning the tokenizer**. Intuition: start with each character as its own token and combine tokens that co-occur a lot.
  - Input: a training corpus (sequence of characters).
  - Initialize the vocabulary be the set of characters.
  - While we want to still grow:
    - Find the pair of elements $x, x' \in V$ that co-occur the most number of times.
    - Replace all occurrences of x, x' with a new symbol $x\ x'$.
    - Add $x\ x'$ to V.

Byte Pair Encoding (BPE) Algorithm

- Example:
  1. [t, h, e, ␣, c, a, r], [t, h, e, ␣, c, a, t], [t, h, e, ␣, r, a, t]
  2. [th, e, ␣, c, a, r], [th, e, ␣, c, a, t], [th, e, ␣, r, a, t] (*th* occurs 3x)
  3. [the, ␣, c, a, r], [the, ␣, c, a, t], [the, ␣, r, a, t] (*the* occurs 3x)
  4. [the, ␣, ca, r], [the, ␣, ca, t], [the, ␣, r, a, t] (*ca* occurs 2x)
- The output of learning is:
  - **Updated vocabulary**: [a, c, e, h, t, r, ca, th, the]
  - **The merges that we made** (hint: important for applying the tokenizer):
    - *t*, *h* ⇒ *th*
    - *th*, *e* ⇒ *the*
    - *c*, *a* ⇒ *ca*

Byte Pair Encoding (BPE) Algorithm

- (Results on the last slide)
  - **Updated vocabulary**: [a, c, e, h, t, r, ca, th, the]
  - **The merges that we made**
    - *t, h ⇒ th*
    - *th, e ⇒ the*
    - *c, a ⇒ ca*

Applying the tokenizer:

- To tokenize a new string, apply the merges in the same order:

[t, h, e, ␣, o, x]

[th, e, ␣, o, x]

[the, ␣, o, x]

Byte Pair Encoding (BPE) Algorithm

- One problem is that (especially in the multilingual setting), there are **a lot (144,697) of Unicode characters**.
- We certainly will not see all characters in the training data.
- In order to **reduce data sparsity even further**, we can run BPE on **bytes** instead of Unicode characters (Wang et al. 2019).
- Example in Chinese:

今天 [gloss: *today*]
[x62, x11, 4e, ca]

Unigram model (SentencePiece)

- Rather than just splitting by frequency, a more “principled” approach is to **define an objective function** that captures what a good tokenization looks like.
- Given a sequence $x_{1:L}$, tokenization T is a set of

$$p(x_{1:L}) = \prod_{(i,j)\in T} p(x_{i:j}).$$

- Example:
  - Training data (string): ababc
  - Tokenization T= { (1, 2) , (3, 4) , (5, 5) } (V= {ab, c})
  - Likelihood: $p(x_{1:L}) = \frac{2}{3} \cdot \frac{2}{3} \cdot \frac{1}{3} = \frac{4}{9}.$

Unigram model (SentencePiece)

- Unigram model algorithm:
- Start with a “reasonably big” seed vocabulary V.
- Repeat:
  - Given V, optimize p(x) and T using the EM algorithm.
  - Compute loss(x) for each token x ∈ V capturing how much the likelihood would be reduced if x were removed from V.
  - Sort by loss and keep the top 80% tokens in V.

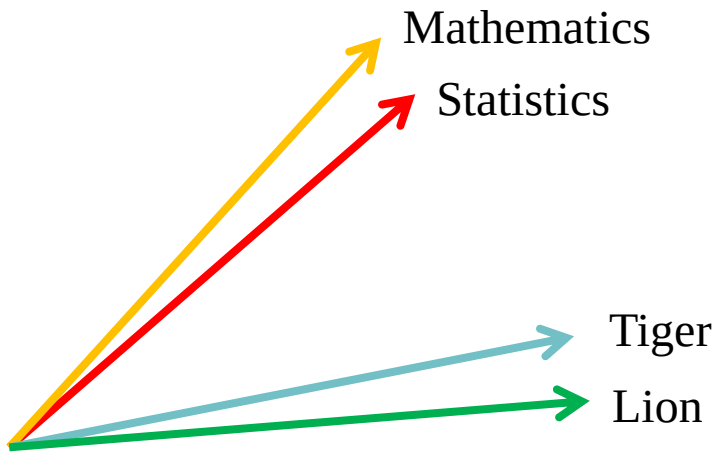
Comparing tokenizers

- Comparing tokenizers
  - GPT-2 and GPT-3 used BPE, vocabulary size of 50K
  - Jurassic used SentencePiece with vocabulary size of 256K
- Impact:
  - Given the same string, Jurassic requires 28% fewer tokens than GPT-3, so it is 1.4x faster
  - Both Jurassic and GPT-3 use the same context size (2048), so one can feed in 39% more text into the prompt.
- Examples of tokenizations for both GPT-3 and Jurassic:
  - GPT-3: [Ab, raham, ␣Lincoln, ␣lived, ␣at, ␣the, ␣White, ␣House, .]
  - Jurassic: [Abraham␣Lincoln, ␣lived, ␣at␣the␣White␣House, .]

Word2Vec

- After transforming texts into tokens, the next challenge is **how do we transform the tokens into vector representations?**
- One-hot encoding: Each dimension of the vector represents a word in the vocabulary. The dimension corresponding to the word is 1 and other dimensions are 0.

Words with similar meanings should have a similar representation



star [0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, ⋯]

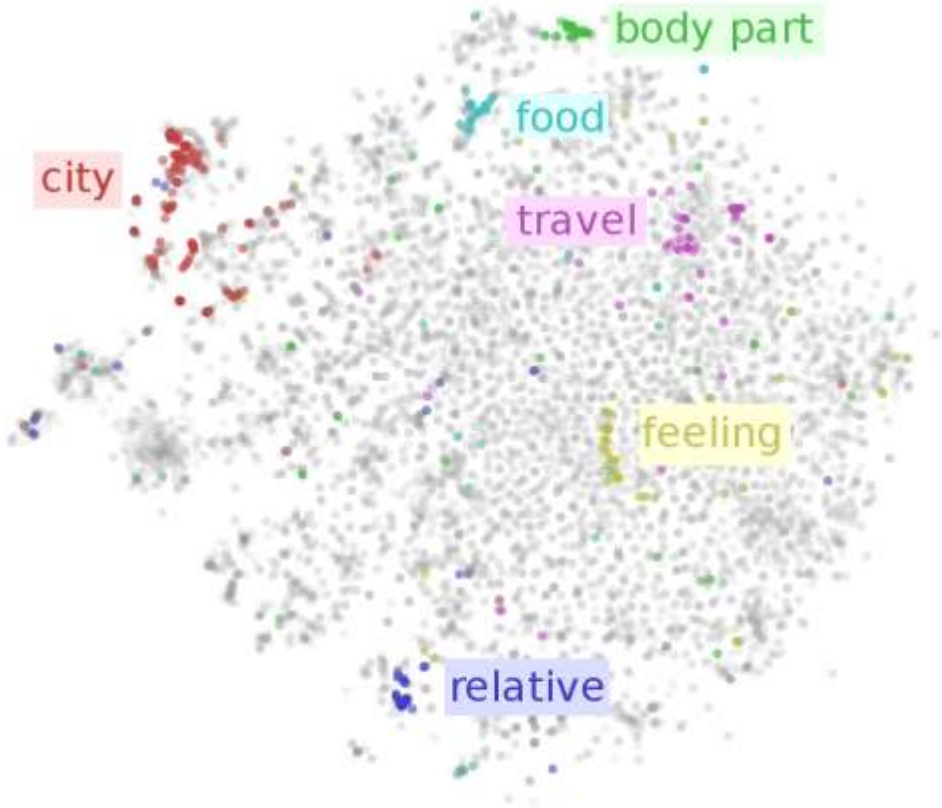
sun [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, ⋯]

sim(star, sun) = 0

Tokens / One-hot encoding DON'T!

Distributed Representation

- In deep learning, we prefer distributed representation, which means many-to-many relationship between concepts and neurons:
  - One concept is represented by more than one neuron activation
  - One neuron represents more than one concept.
- Each word is represented as a dense, real-valued, and low-dimensional vector.



Word2Vec

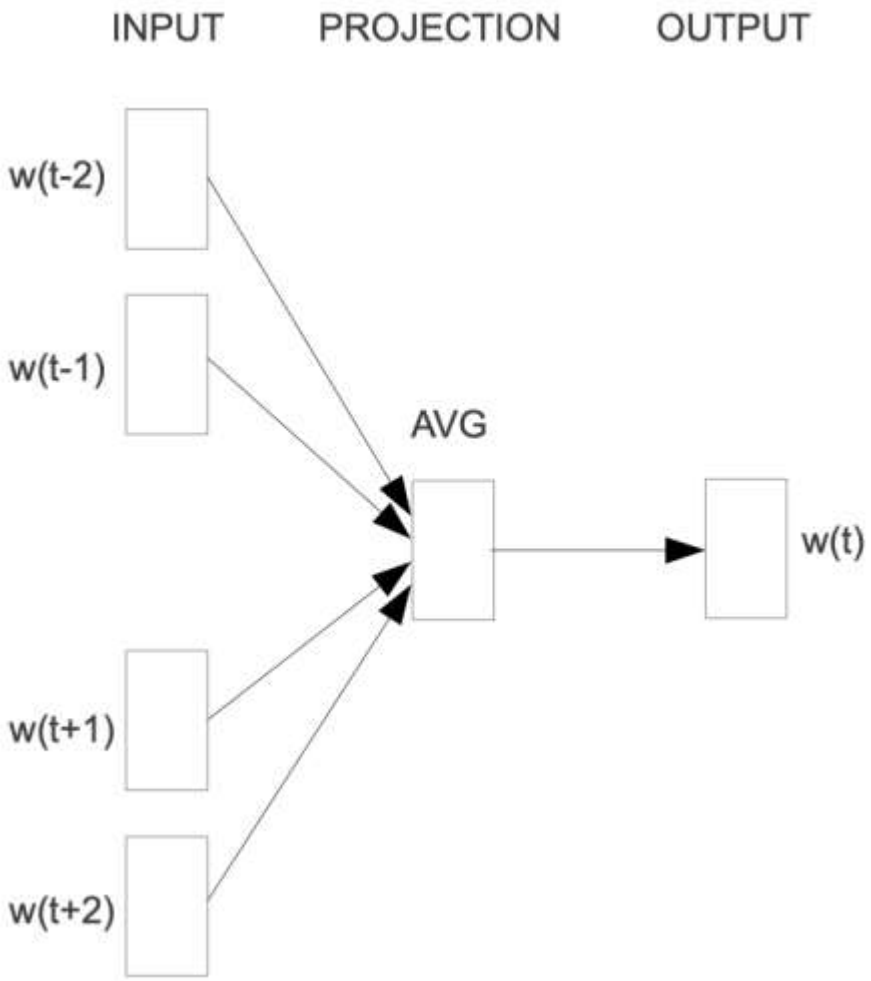
- We want to learn distributed representations for words in the vocabulary. The training data is usually unlabeled text, such as Wikipedia.
- There are two models
  - Continuous Bag-of-Words model (CBOW)
  - Skip-gram model
- For each model, there are two learning methods:
  - Hierarchical Softmax
  - Negative Sampling

Continuous Bag-of-Words Model

- The CBOW model predicts the current word based on the context.
- The context C_t is defined as the words before and after w_t. The context representation is computed by averaging the word vectors of the context words.

$$\boldsymbol{C_t} = \frac{1}{|C_t|}\prod_{w_i \in C_t} u_{w_i}$$

- The model is called bag-of-words model as the order of words in the context does not influence projection of the context representation.



Continuous Bag-of-Words Model

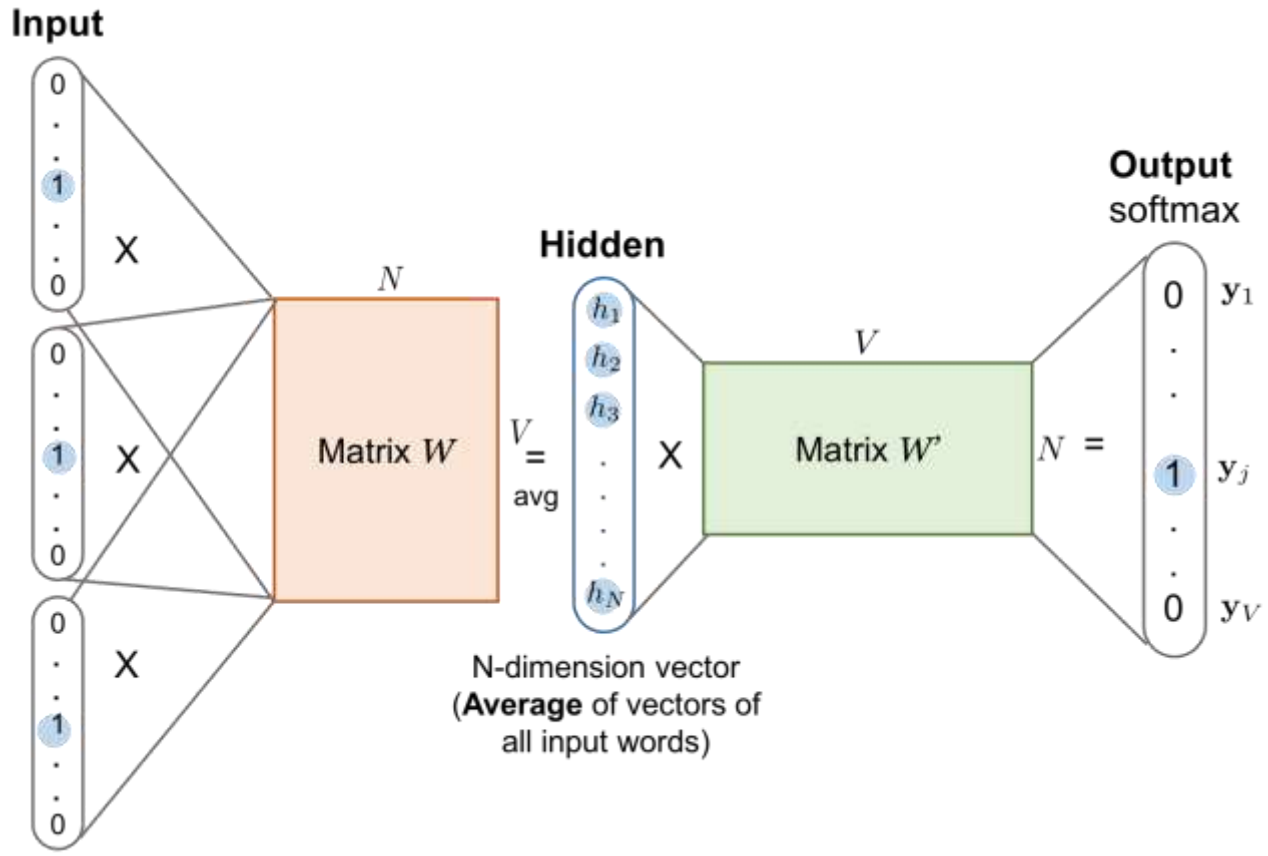
- The goal is to maximize:

$$\prod_{t=1}^{T} p(w_t|\boldsymbol{C}_t)$$

- If we use exponential energy function (Softmax):

$$p(w_t|C_t) = \frac{\exp(v_{w_t} \cdot \boldsymbol{C}_t)}{\sum_{i=1}^{|V|} \exp(v_{w_i} \cdot \boldsymbol{C}_t)}$$

- It needs to scan all the words in the vocabulary to get the probability.



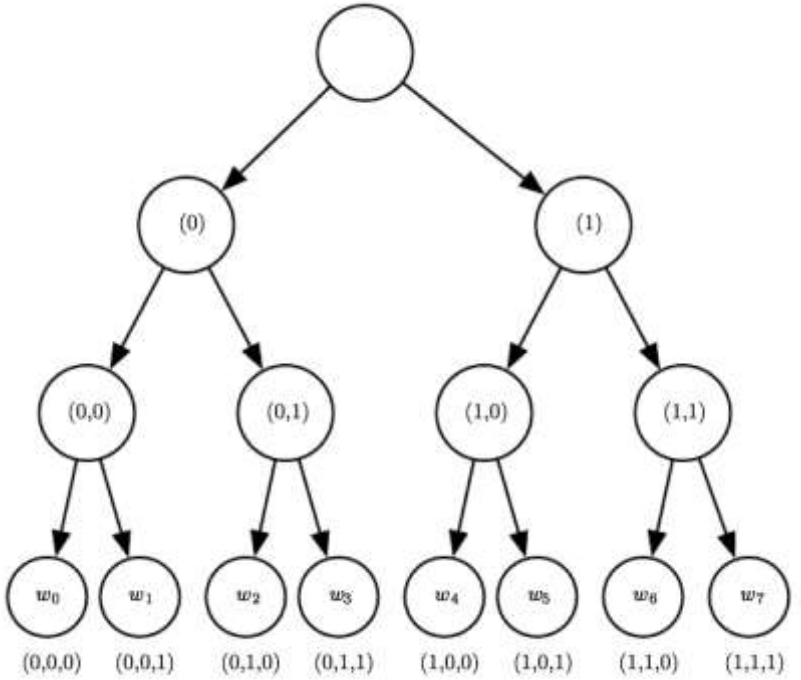
Hierarchical Softmax

- We can build a binary tree for all words in the vocabulary, with words as leaves, the left child encoded as 1 and the right child encoded as 0.
- Each word can be assigned a unique code from the root to the leaf.
- The probability is

$$p(w_t|C_t) = \prod_{k=1}^{K} p(d_k|q_k, C_t)$$

$$= \prod_{k=1}^{K} (\sigma(q_k \cdot \boldsymbol{C}_t)^{1-d_k} \cdot \left(1 - \sigma(q_k \cdot \boldsymbol{C}_t)\right)^{d_k})$$

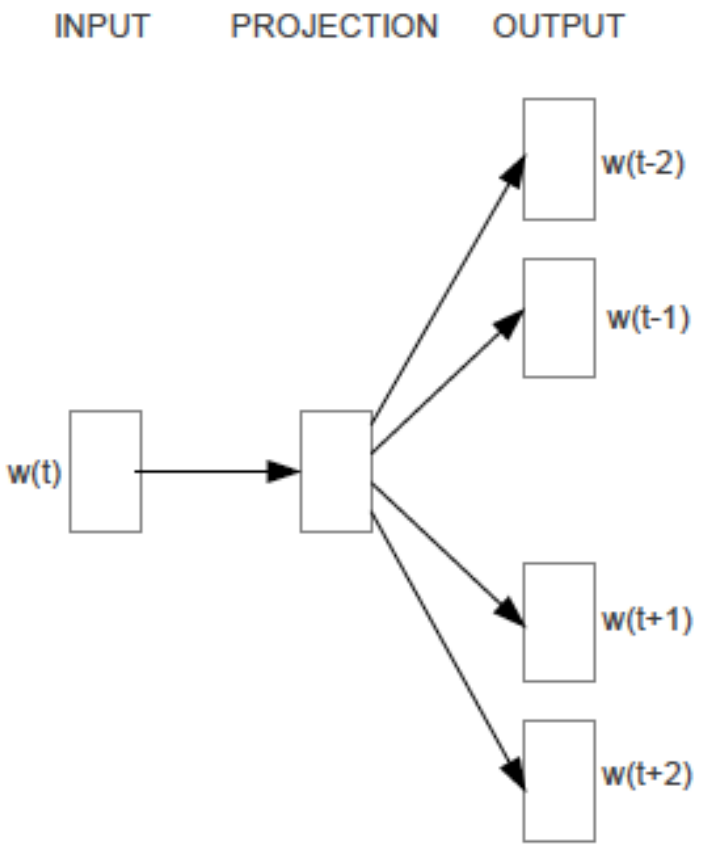
where σ is the sigmoid function, q_k is the vector of k-th node on the path from root to word leaf, d_k is the corresponding code of q_k.



Skip-gram Model

- The skip-gram model predicts the context based on the current word.
- The context is defined as the words within a certain range before and after the current word.
- The goal is to maximize:

$$\prod_{t=1}^{T} \sum_{w_o \in C_t} p(w_o|w_t)$$



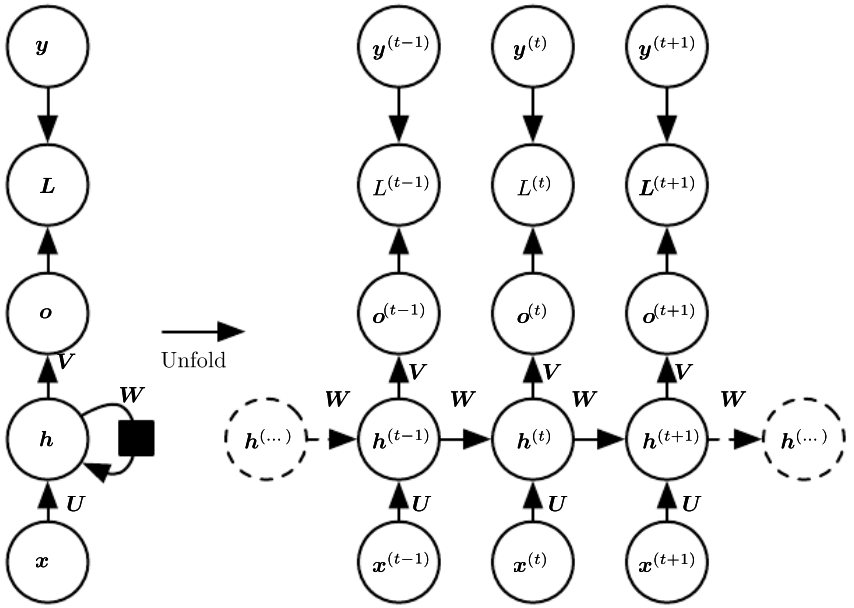
CBOW vs Skip-gram: Which is better?

- It is a common belief in the NLP community that CBOW word embeddings tend to underperform skip-gram embeddings.
- Recently, researchers find that this belief can be attributed to incorrect CBOW implementations in the original paper and standard software libraries.
- The correct implementation of CBOW yields word embeddings that are fully competitive with skip-gram on various tasks while being more than three times as fast to train.

Irsoy et al. A Corrected CBOW Implementation. 2020.

Sequence Modeling

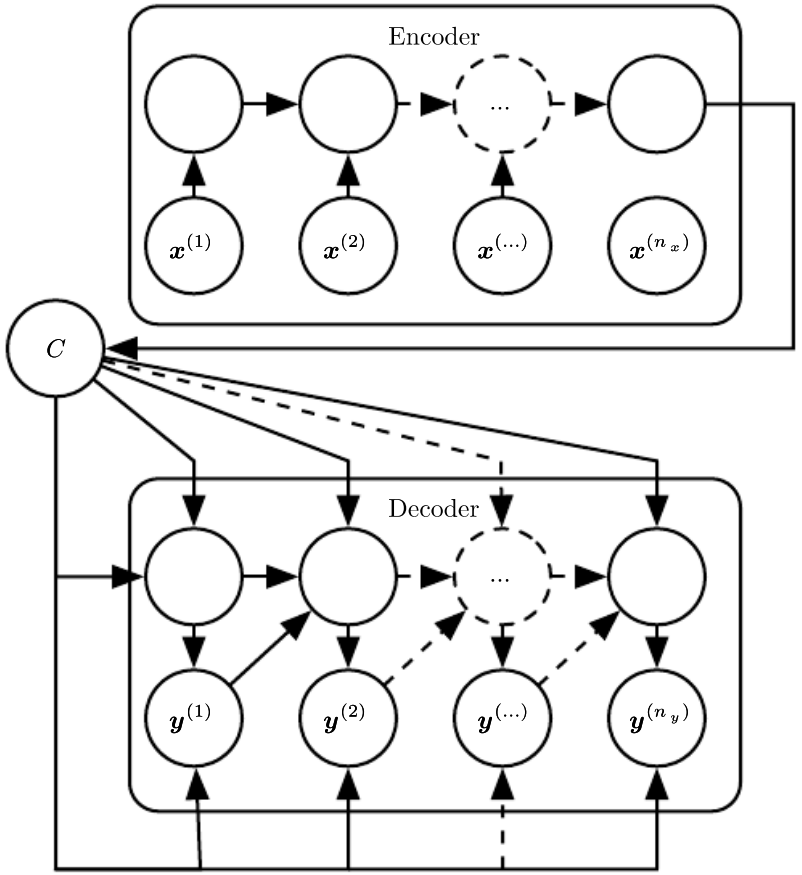
- With word2vec, we can transform text from sequence of words into sequence of vectors.
- To model sequences, previous SOTA models are recurrent neural network (RNN) and its variants



Problems of RNN

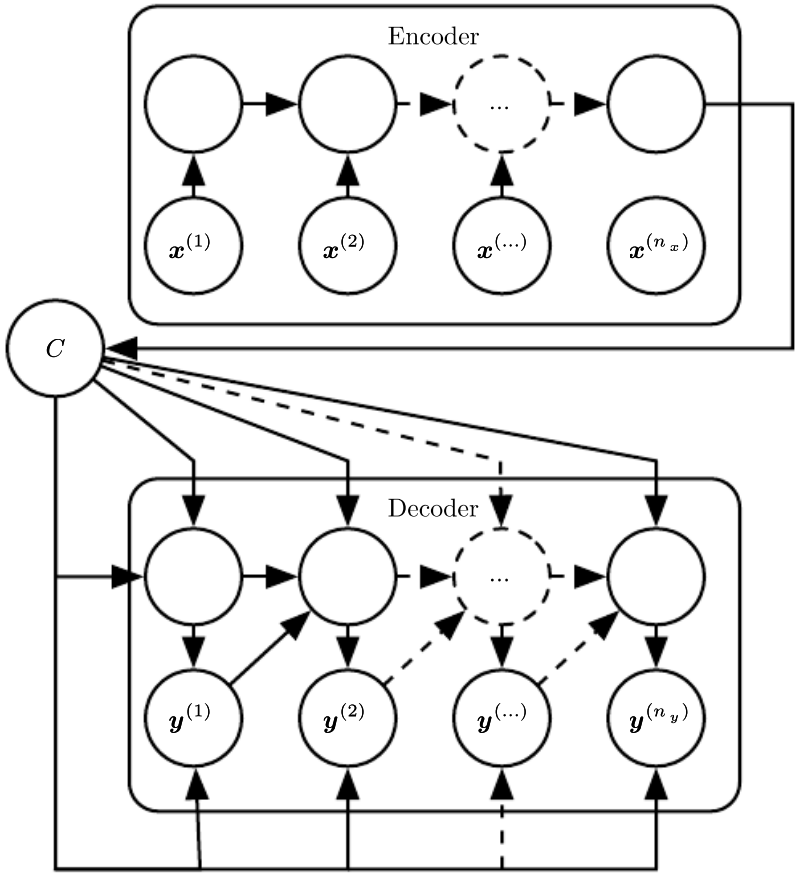
- Can we train a RNN model to map an input sequence to an output sequence which is not necessarily of the same length?
- This is important in many applications: speech recognition, machine translation, or question answering.

Challenge: Transmit local and global information through one bottleneck (in seq2seq models)



Encoder-Decoder Seq-to-Seq Architectures

- When C is a fixed-size vector, this architecture could be reviewed as a combination of a sequence to fixed-size vector RNN and a fixed-size vector to sequence RNN.
- Limitation: C may be too small to summarize a long sequence (Bahdanau et al. (2015)).



Bahdanau Attention

- *C* may be too small to summarize a long sequence (Bahdanau et al. (2015)). They proposed to make C a variable-length sequence. Additionally, they introduced the first attention mechanism -- Bahdanau Attention (also called Additive Attention).

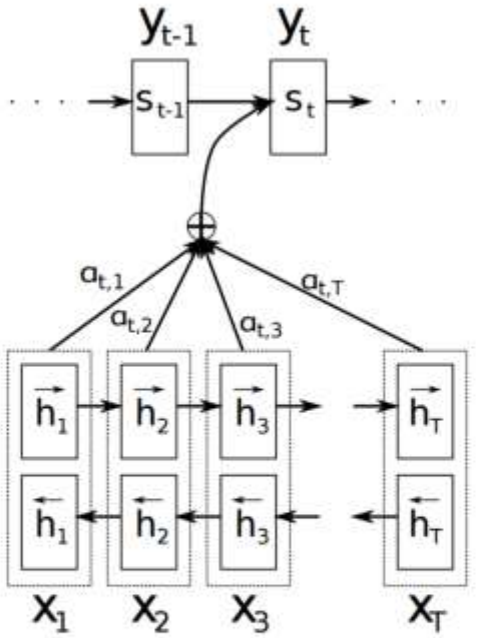
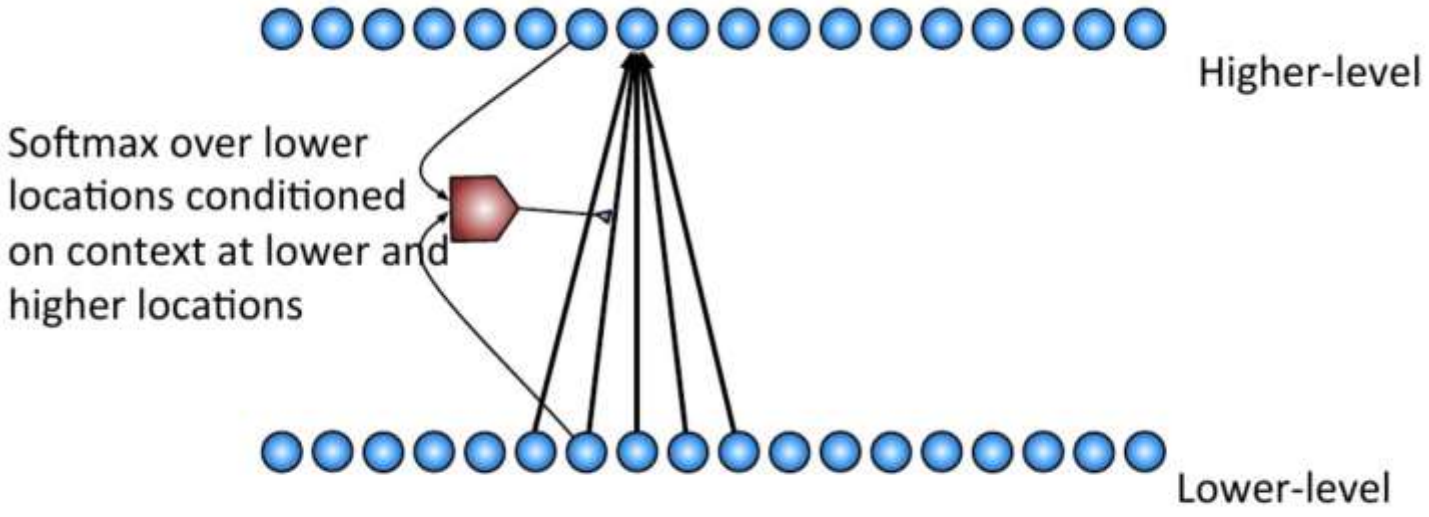


Figure 6: Formulatin: $a(\cdot)$ is an alignment model which scores how well the inputs around position j and the output at position i match, which is parameterized as a feedforward neural network

Inspiration for Attention

- Consider an input (or intermediate) sequence
- Consider an upper level representation, which can choose "where to look" when producing some task (e.g., machine translation) at each position



Summary

	Long-Term Dependency	Parallelization	Computation	Flexibility for Length	Gradient Flow	Complexity
RNN	Poor, due to recurrent structure	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for input, fixed for output	Problems with vanishing/exploding gradients	Simpler, fewer hyperparameters
LSTM	Better, still rooms to improve	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for input, fixed for output	Better gradient flow with less vanishing	Complex, many hyperparameters
CNN	Not inherent	Highly parallelizable	Fast due to parallelizable operations	Less flexible, typically requires fixed-size inputs	Stable gradient flow	Complex, many hyperparameters
Bahdanau Attention	Good with attention mechanism	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for both input and output	Better gradient flow with less vanishing	Simpler, fewer hyperparameters

Still Not Solved!

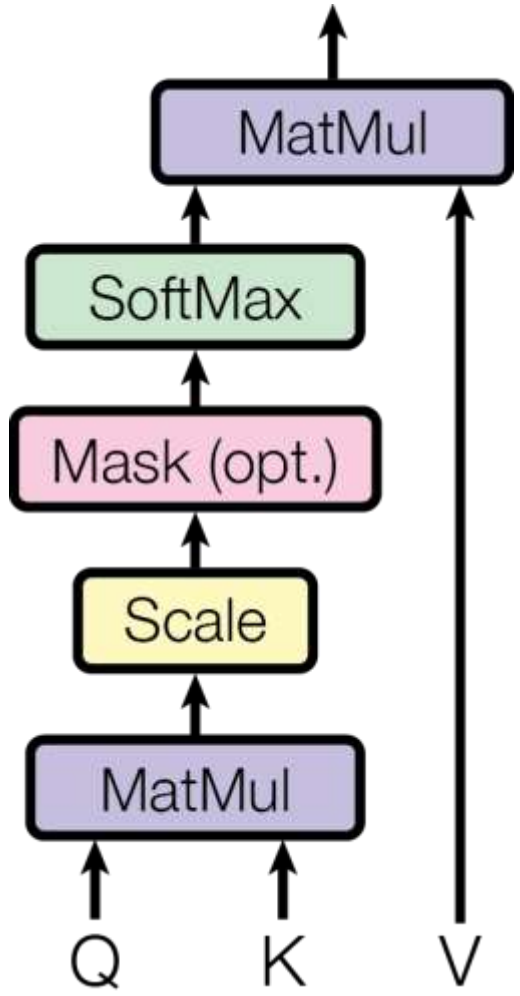
Scaled Dot-Product Attention

- Attention: a weighted sum of the values, where the weight assigned to each value is computed by a function of the query and the key:

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

where $Q, K \in R^{L \times d_K}$, $V \in R^{L \times d_V}$.

- Scale of dot product increases as dimensions get larger, so the attention scores are scaled by the size of the vector.



Scaled Dot-Product Attention

- Suppose we have two words: thinking and machines
- Each word already has a *d*-dimension vector representation
- *q*, *k*, *v* can be obtained by multiplying the three matrices W^Q, W^K, W^V

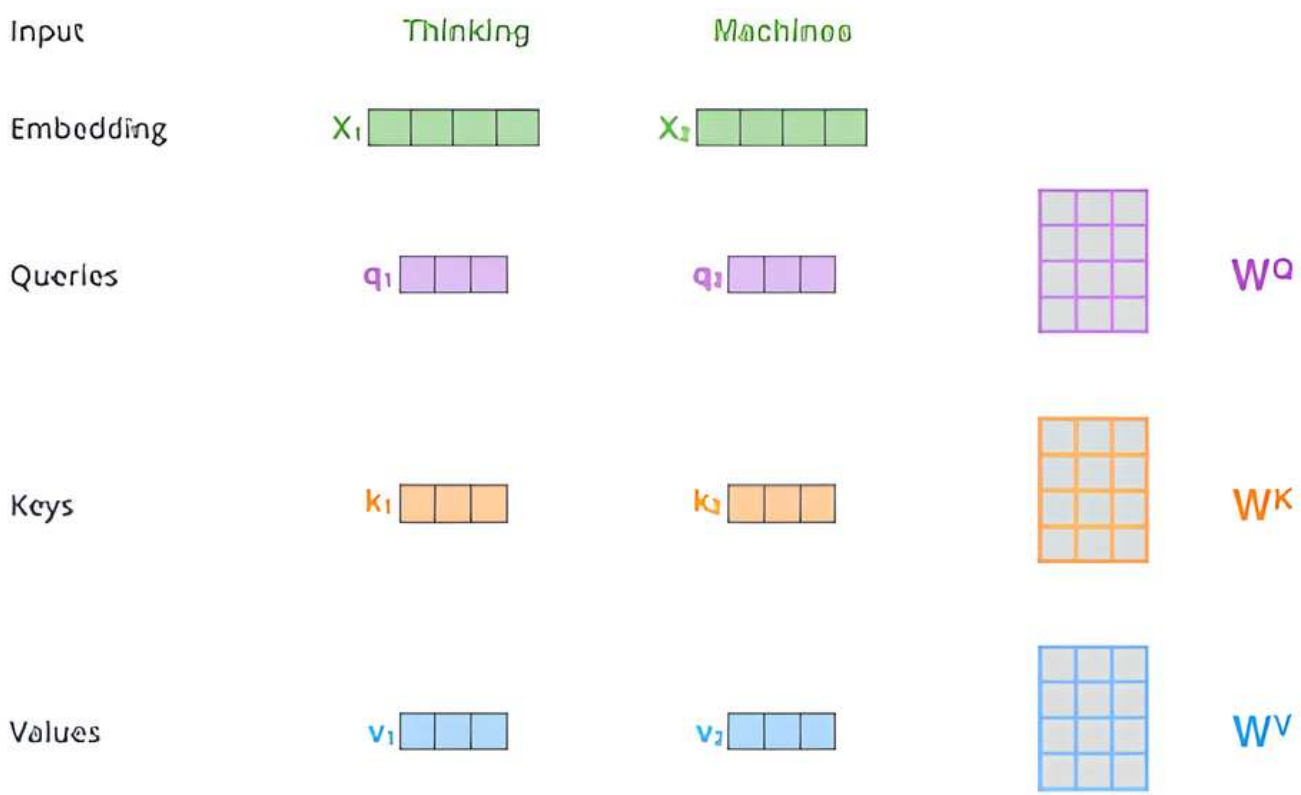


figure credit: https://www.cnblogs.com/baobaotql/p/11662720.html

Scaled Dot-Product Attention

- Dot product for (q_1, k_1) and (q_1, k_2)

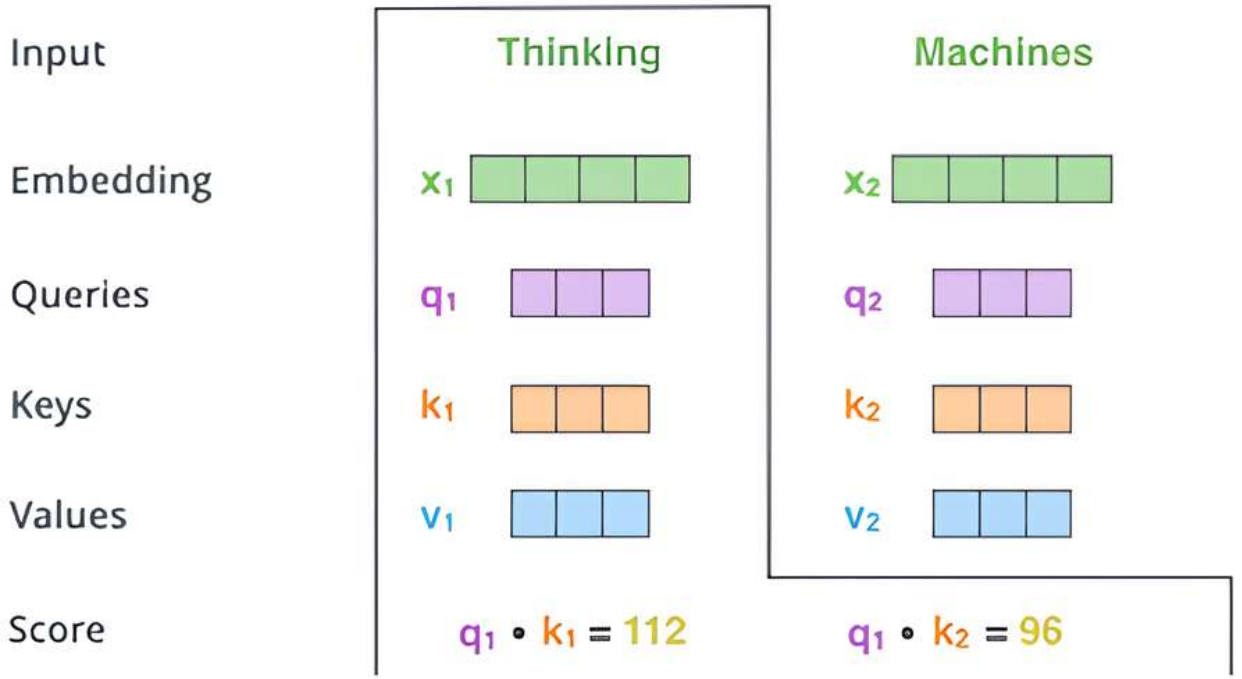


figure credit: https://www.cnblogs.com/baobaotql/p/11662720.html

Scaled Dot-Product Attention

- After normalization, and then apply softmax
- Then multiply *v*, then we obtain *z*

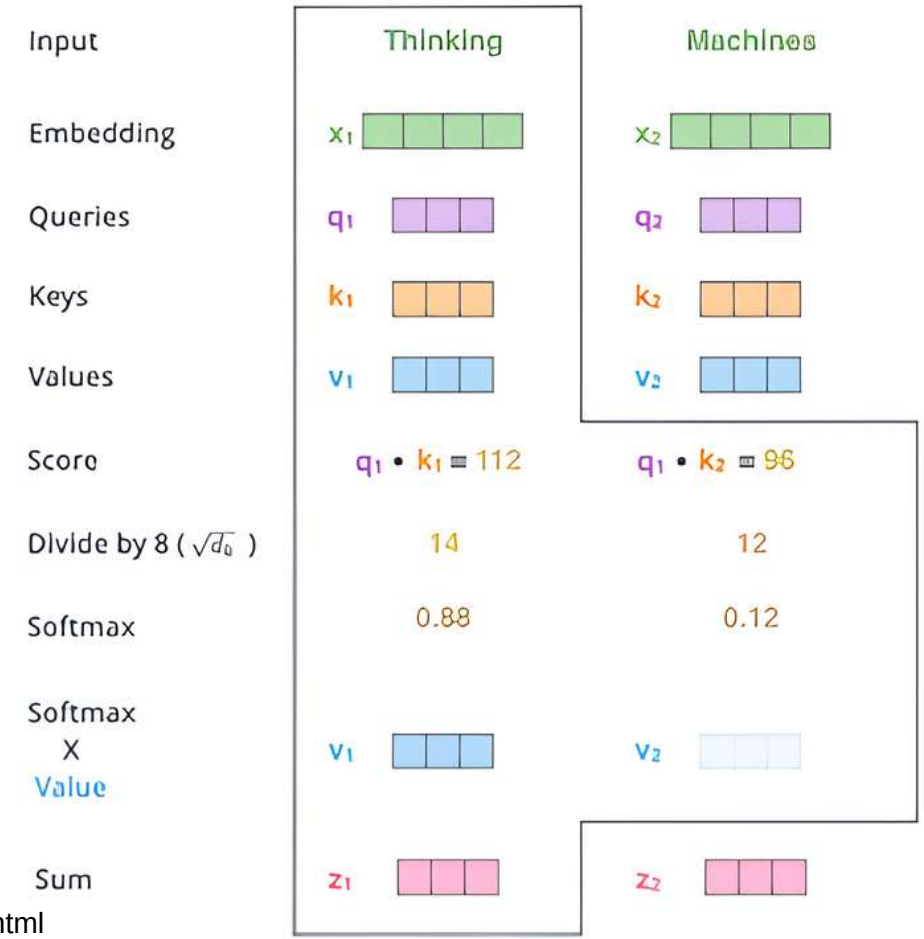
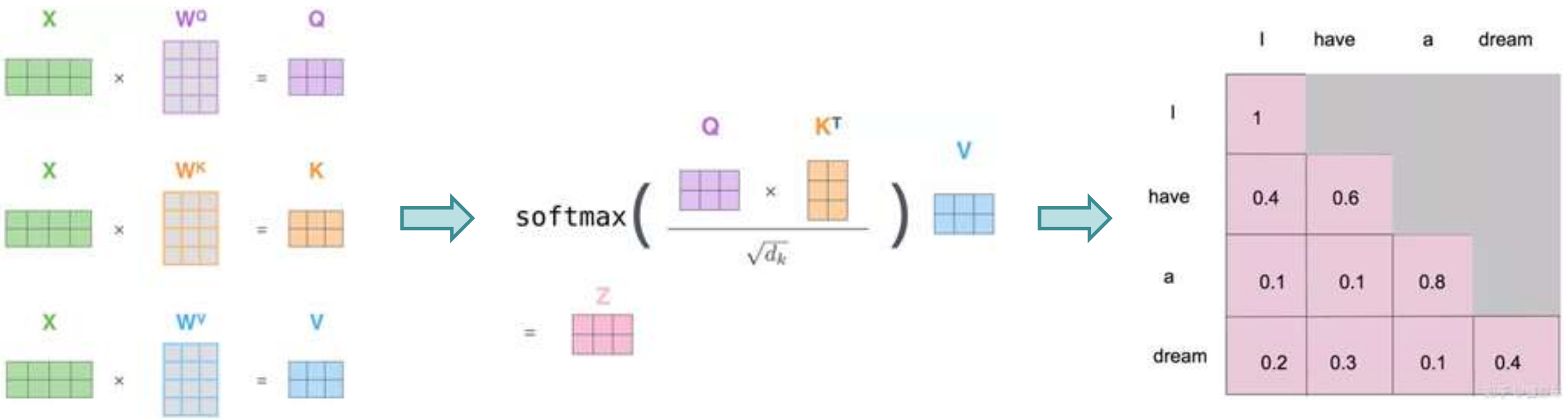


figure credit: https://www.cnblogs.com/baobaotql/p/11662720.html

To summarize

- Step 1: Calculate three matrices Q, K, V
- Step 2: Scaled dot product
- Step 3: Output attention map



Why $/\sqrt{\mathrm{d}_k}$?

- SDPA:

$$\boldsymbol{o}_i = \sum_{j=1}^{n} a_{i,j}\boldsymbol{v}_j, \quad a_{i,j} = \frac{e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j}}{\sum_{j=1}^{n} e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j}}$$

- Invariance of entropy: H_i should be insensitive to n

$$\mathcal{H}_i = -\sum_{j=1}^{n} a_{i,j} \log a_{i,j}$$

- Why? After adding new tokens, existing tokens can still be focused

https://spaces.ac.cn/archives/8823/comment-page-1

Why $/\sqrt{\mathrm{d}_k}$?

$$\mathcal{H}_i = -\sum_{j=1}^{n} a_{i,j} \log a_{i,j} = \log \sum_{j=1}^{n} e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j} - \frac{\sum_{j=1}^{n} e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j} (\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j)}{\sum_{j=1}^{n} e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j}}$$

We need to determine the appropriate λ to offset the effect of length

$$\sum_{j=1}^{n} e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j} = n \times \frac{1}{n} \sum_{j=1}^{n} e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j} \approx n \, \mathbb{E}_j[e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j}]$$

$$\mathcal{H}_i \approx \log n + \log \mathbb{E}_j[e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j}] - \frac{\lambda \, \mathbb{E}_j[e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j} (\boldsymbol{q}_i \cdot \boldsymbol{k}_j)]}{\mathbb{E}_j[e^{\lambda \boldsymbol{q}_i \cdot \boldsymbol{k}_j}]}$$

https://spaces.ac.cn/archives/8823/comment-page-1

Why $/\sqrt{\mathrm{d}_k}$?

With Laplace approximation:

$$\mathcal{H}_i \approx \log n - 0.24\lambda d + \mathcal{O}(1)$$

To counteract the effect of length n:

$$\log n - 0.24\lambda d = 0$$

$$\lambda = \log n/(0.24d)$$

Introducing hyperparameter k for approximation:

$$\lambda = \frac{\kappa \log n}{d}$$

https://spaces.ac.cn/archives/8823/comment-page-1

Why $/\sqrt{\mathrm{d}_k}$?

$$Attention(Q,K,V) = softmax\left(\frac{\kappa \log n}{d} QK^\top\right) V$$

Suppose n = 512, we set $\kappa = \frac{\sqrt{d}}{\log 512}$

$$Attention(Q,K,V) = softmax\left(\frac{\log_{512} n}{\sqrt{d}} QK^\top\right) V$$

Not necessarily $1/\sqrt{\mathrm{d}_k}$! In ”Overcoming a Theoretical Limitation of Self-Attention” (ACL 2022)

$$Attention(Q,K,V) = softmax\left(\frac{\log n}{\sqrt{d}} QK^\top\right) V$$

https://spaces.ac.cn/archives/8823/comment-page-1

Attention Visualization

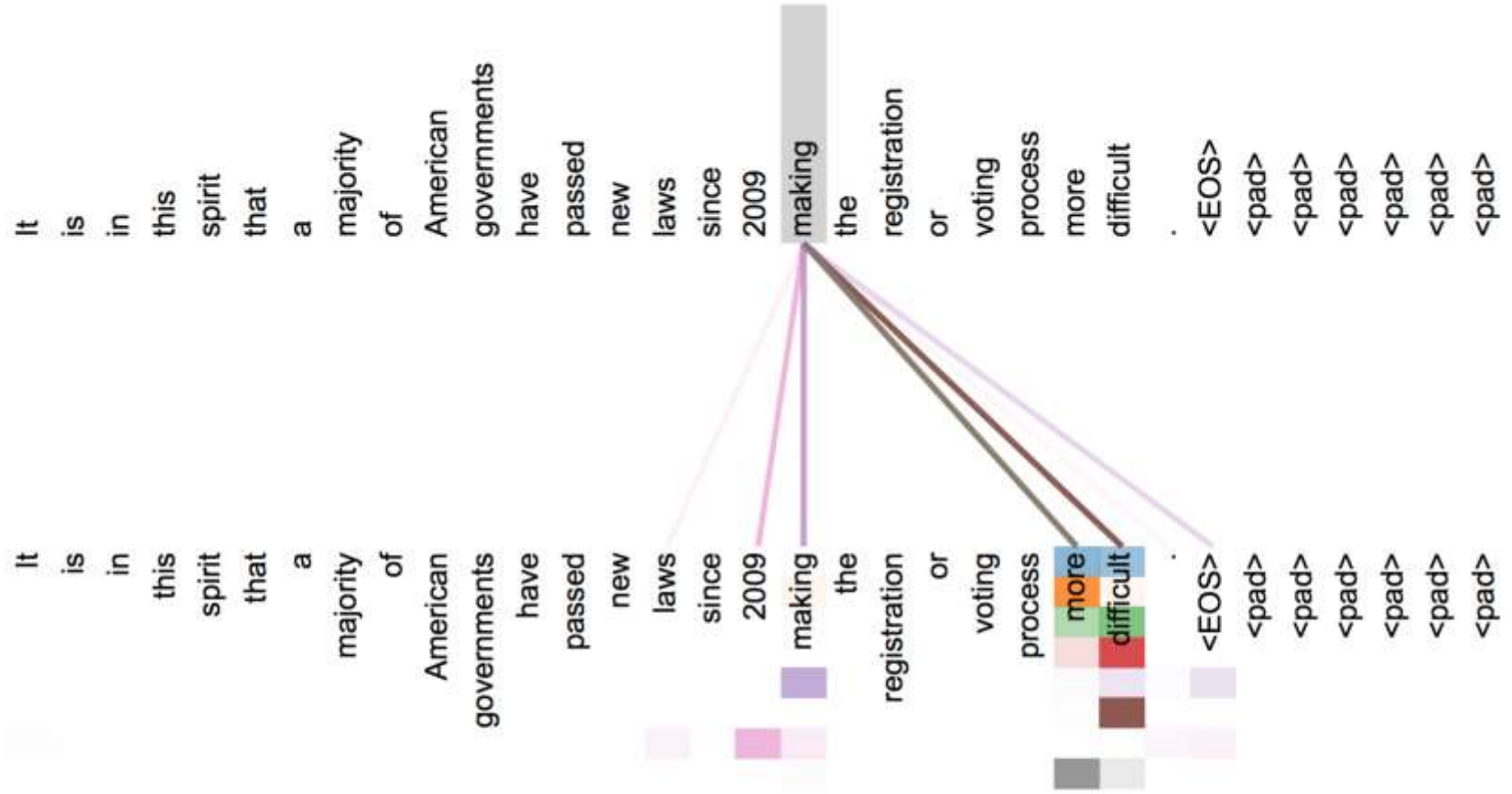


Figure 3: An example of the attention mechanism following long-distance dependencies in the encoder self-attention in layer 5 of 6. Many of the attention heads attend to a distant dependency of the verb 'making', completing the phrase 'making...more difficult'. Attentions here shown only for the word 'making'. Different colors represent different heads. Best viewed in color.

Why Self-Attention

- Constant path length between any two positions
- Trivial to parallelize per layer

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)	$O(r \cdot n \cdot d)$	$O(1)$	$O(n/r)$

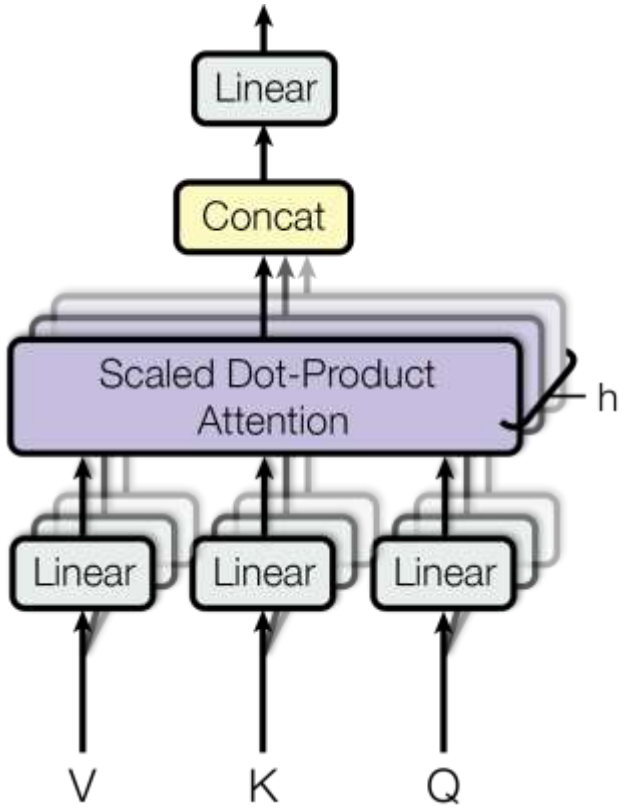
What's missing from Scaled Dot-Product Attention?

- Convolution: a different linear transformation for each relative position.
- Scaled Dot-Product: a weighted average :(

The Fix: Multi-Head Attention

- Multiple attention layers (heads) in parallel (shown by different colors).
- Each head uses different linear transformations. Different heads can learn different relationships.

Multi-head Attention



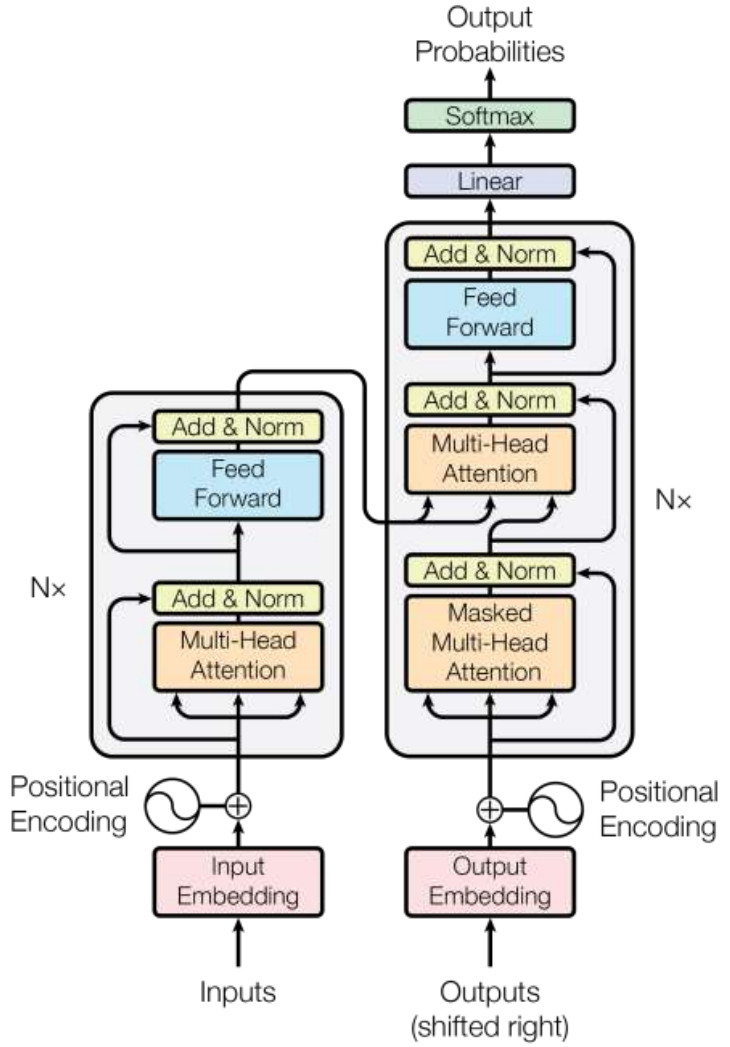
$$\text{Multihead}(Q, K, V) = \text{Concat}(\text{head}_1, \cdots, \text{head}_n)W^O$$

$$\text{head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V\right),$$

where $W_i^Q, W_i^K, W_i^V \in R^{L \times d_k/n}$

Transformer

- The standard transformer has an encoder and a decoder, each of which consists of several transformer layers.
- Each transformer layer has two sub-layers, a multi-head self-attention mechanism and a position-wise fully connected feed-forward network.
- We employ a residual connection around each of the two sub-layers, followed by a layer normalization. The output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$

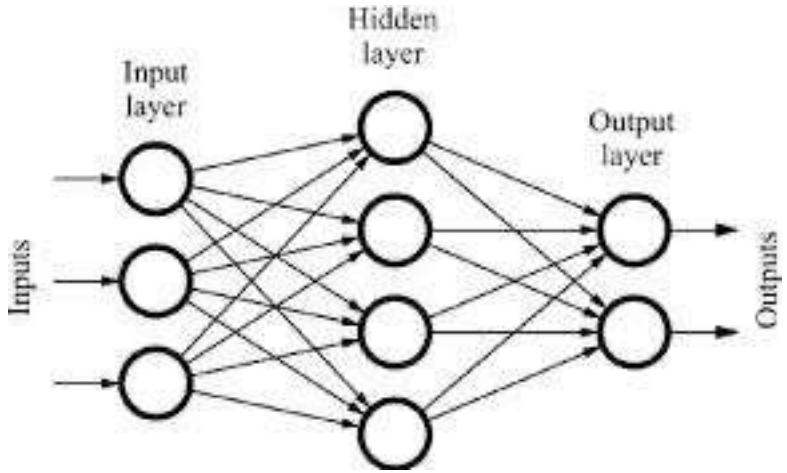


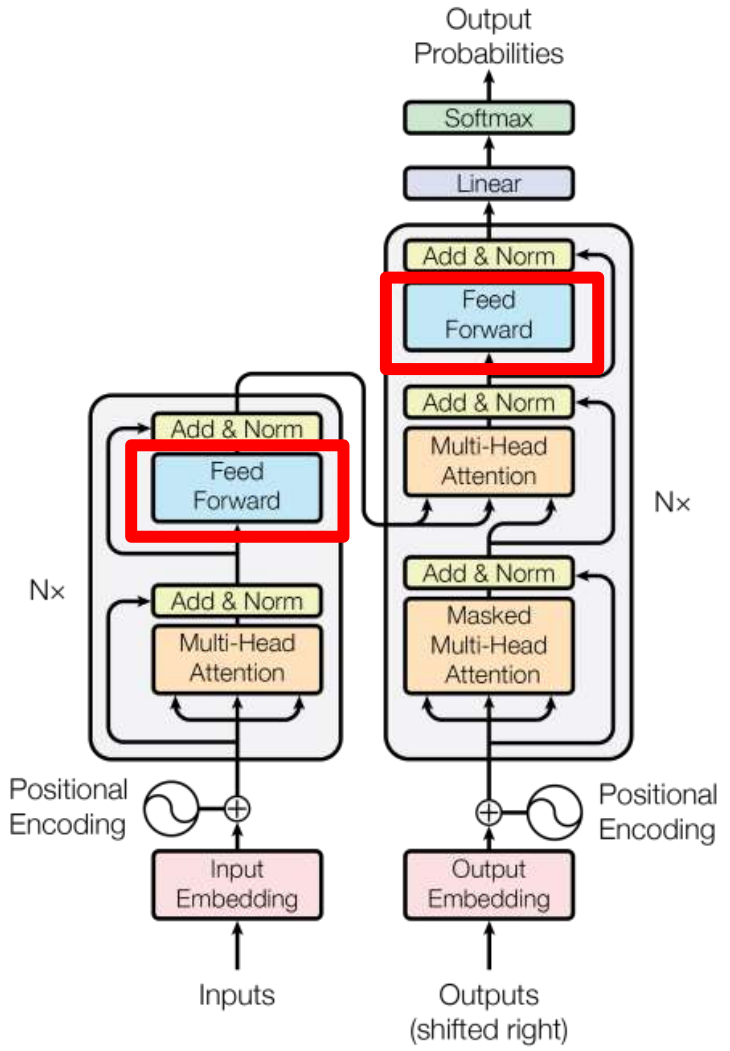
Feed-Forward Networks

- The full connected feed-forward network is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between.

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

where $W_1 \in R^{d_{\mathrm{model}} \times d_{\mathrm{ff}}}$, $W_2 \in R^{d_{\mathrm{ff}} \times d_{\mathrm{model}}}$, $b_1 \in R^{d_{\mathrm{ff}}}$, $b_2 \in R^{d_{\mathrm{model}}}$.

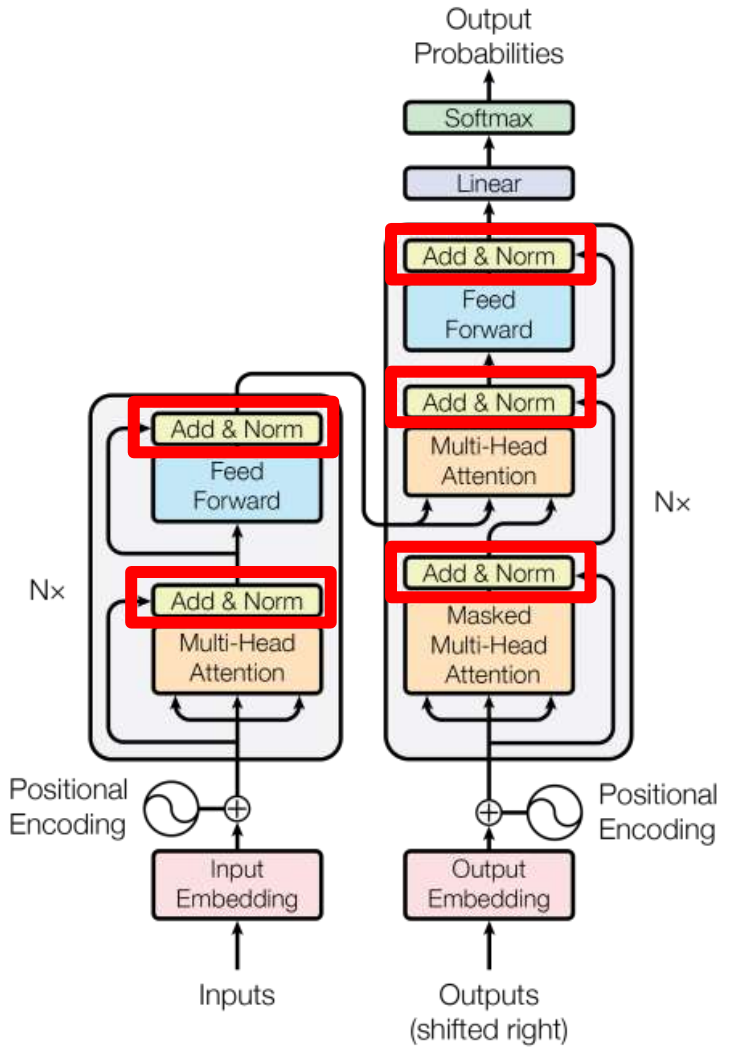




Layernorm

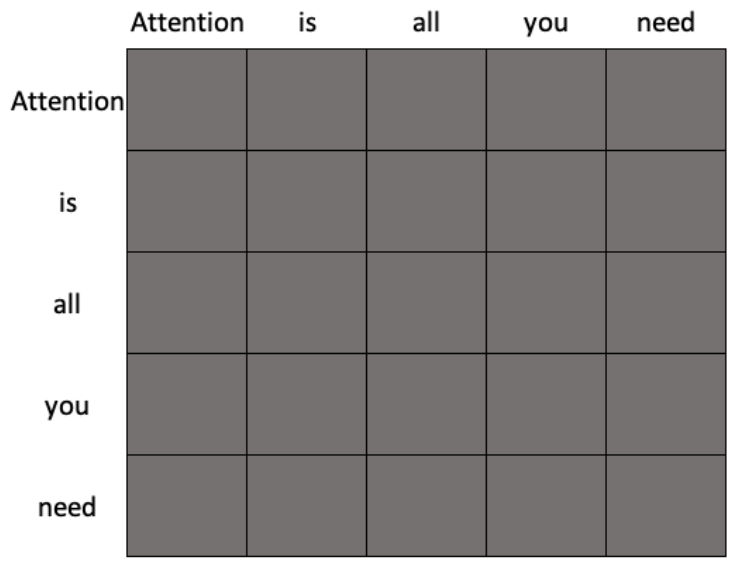
- Batch normalization significantly reduces the training time in feed-forward neural networks. However, the effect of batch normalization is dependent on the mini-batch size and it is not obvious how to apply it to recurretn neural networks.
- Layer normalization computes the statistics over all the hidden units in the same layers:

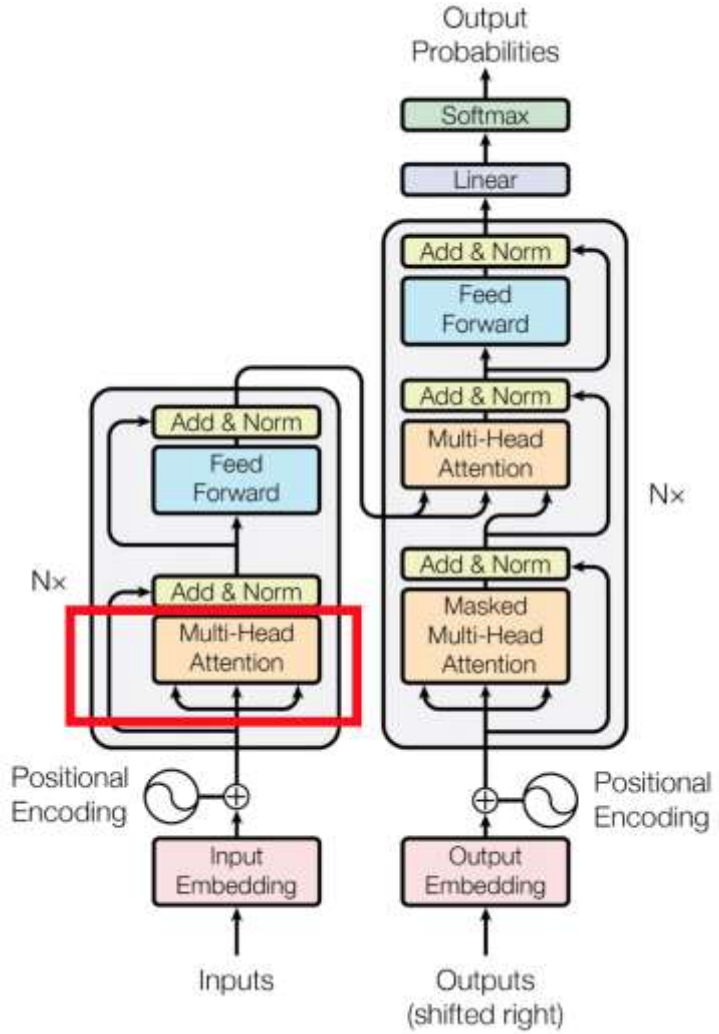
$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} \cdot \gamma + \beta$$



Encoder Self-Attention

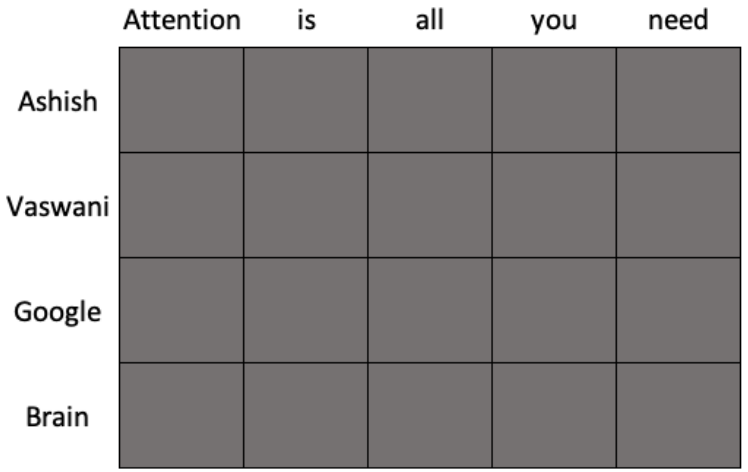
- All of the queries, keys, and values come from the same place, in this case, the output of the previous layer.
- Bidirectional attention: each position can attend to all the positions.

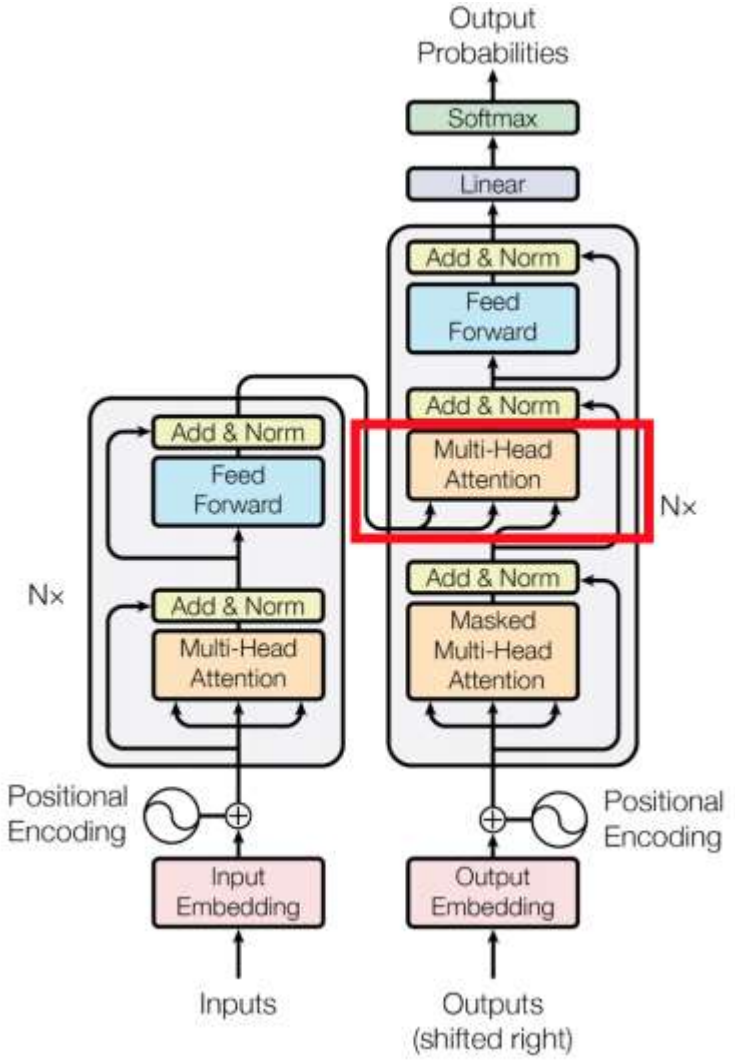




Encoder-Decoder Attention

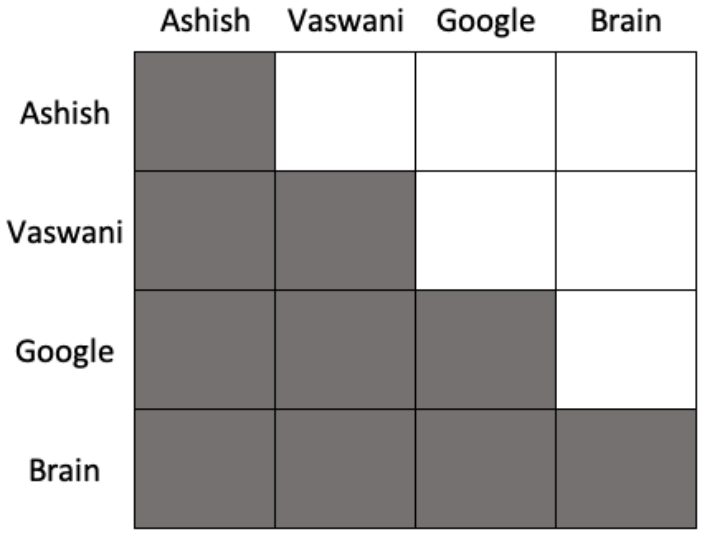
- The queries come from the previous decoder layer, and the memory keys and values come from the output of the encoder.
- Each position in the decoder can attend to all the positions in the encoder.

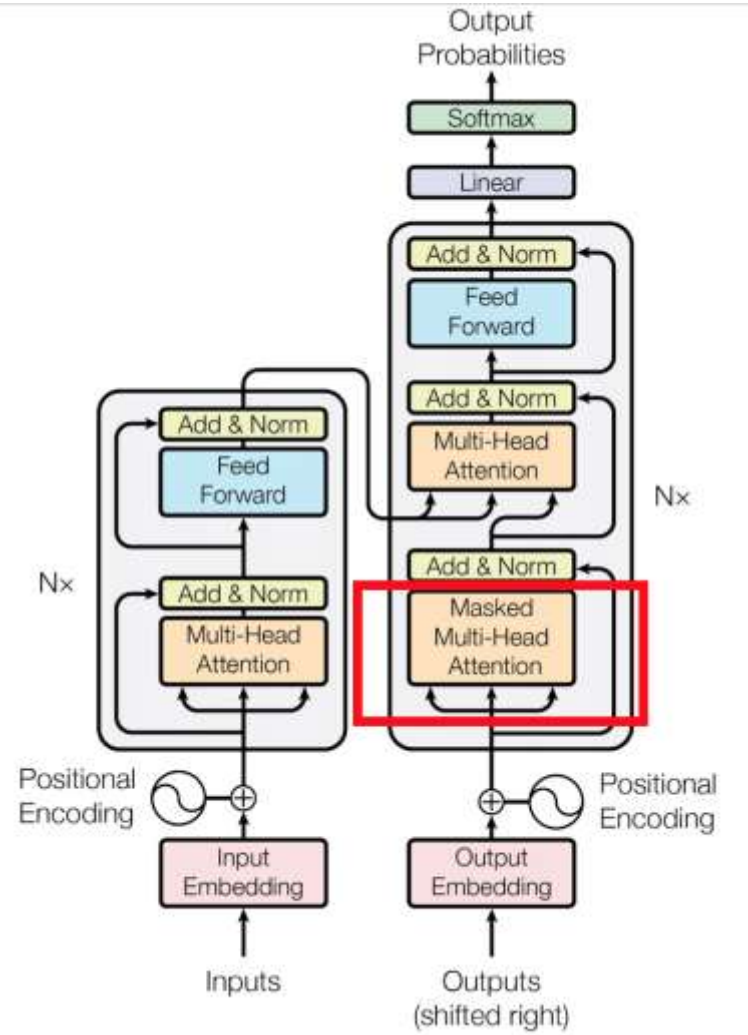




Masked Decoder Self-Attention

- All of the queries, keys, and values come from the output of the previous layer.
- Causal mask: each position in the decoder can only attend to positions in the decoder up to and including that position.





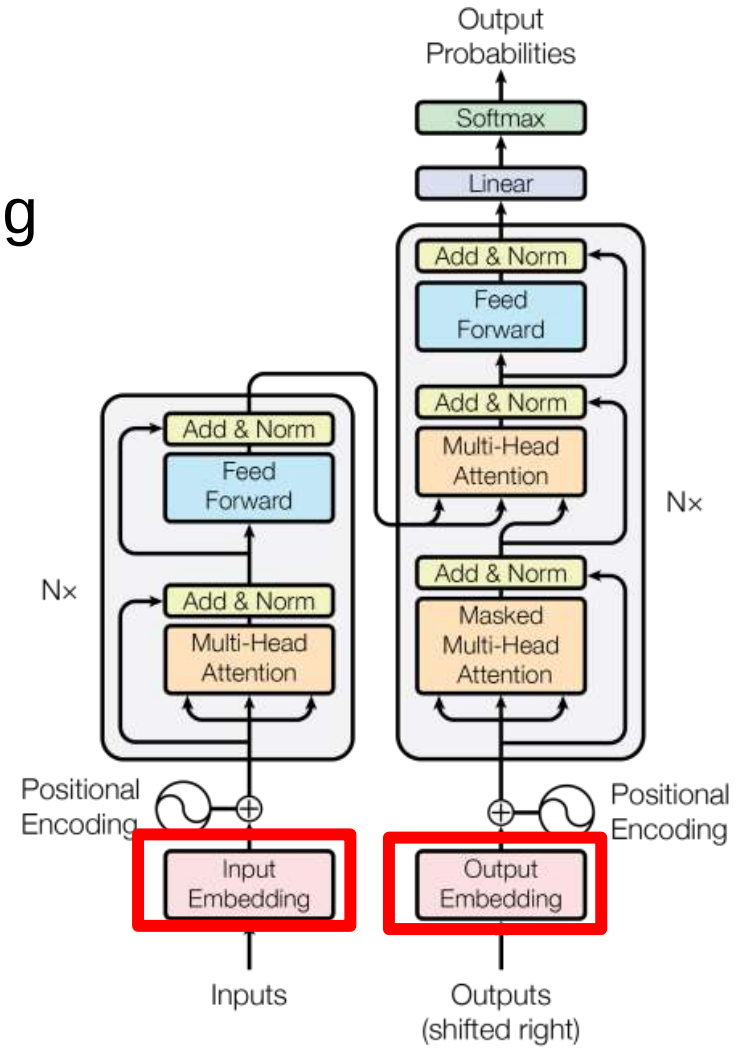
Positional Encoding

- The attention mechanism is independent of positions (except causal mask). To make use of the order of the sequence, we inject absolute position of tokens by adding "positional encoding" to the input embeddings at the bottom of the encoder and the decoder:

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

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

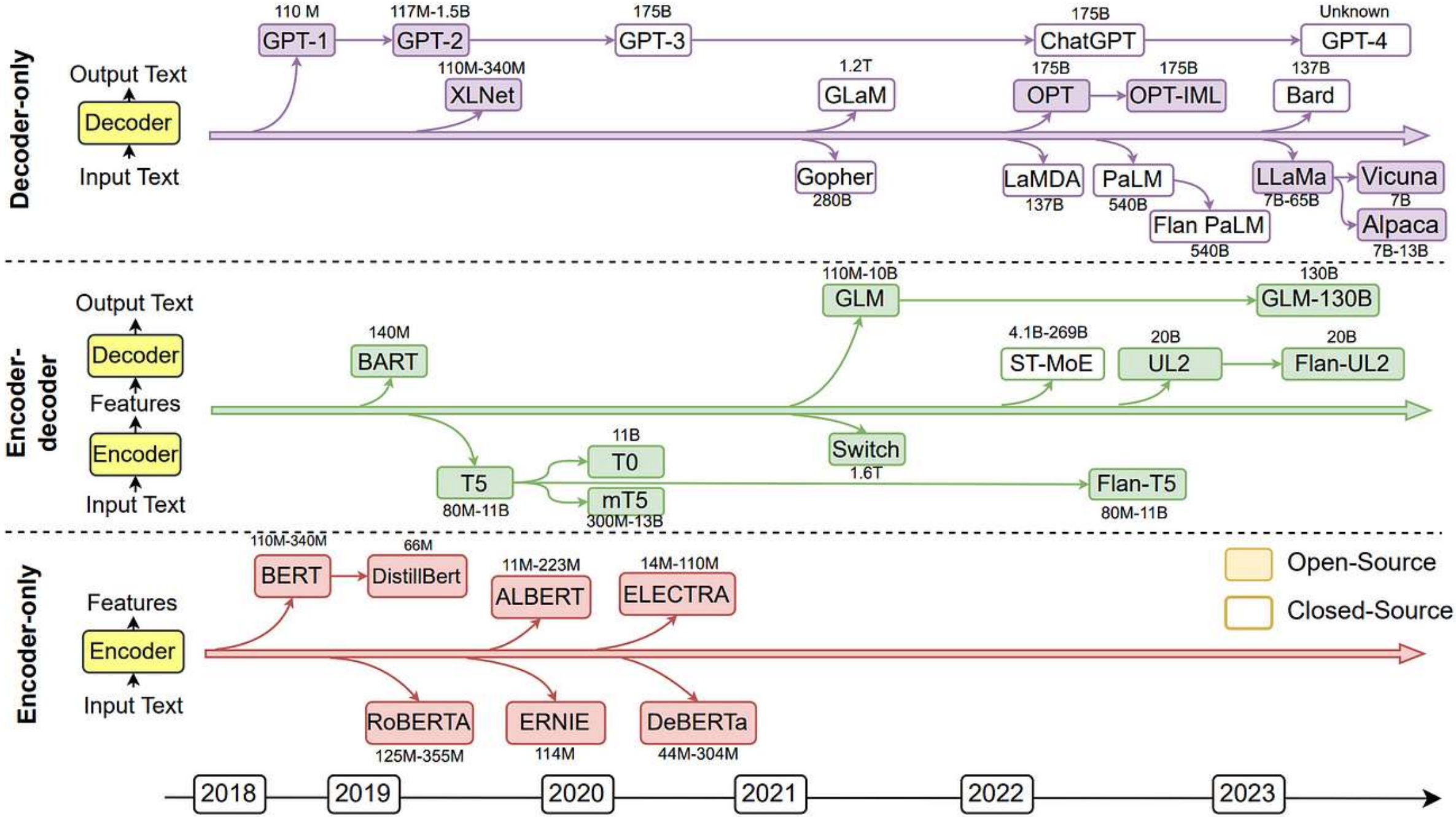
- For any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.
- Alternatively, we can learn positional embeddings along with model parameters



Summary

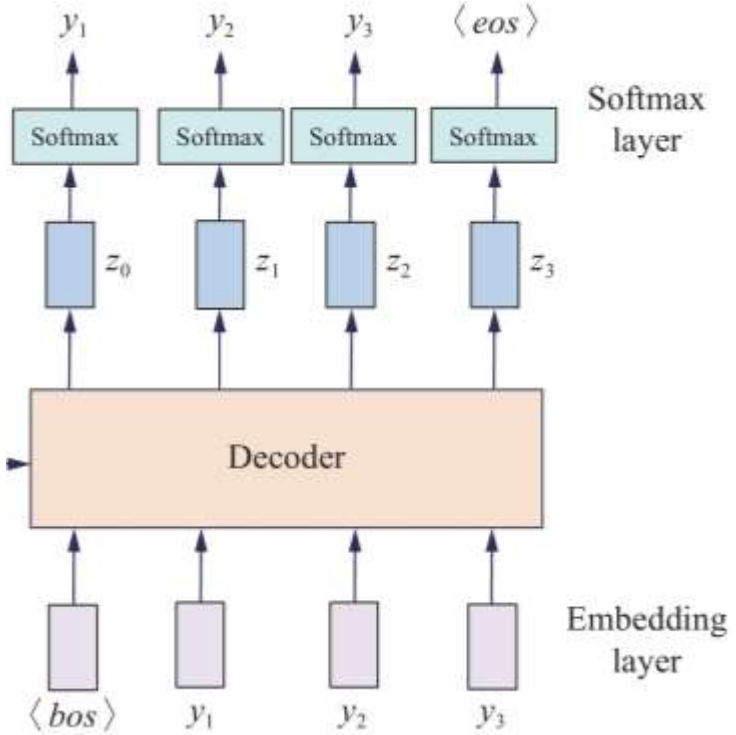
	Long-Term Dependency	Parallelization	Computation	Flexibility for Length	Gradient Flow	Complexity
RNN	Poor, due to recurrent structure	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for input, fixed for output	Problems with vanishing/exploding gradients	Simpler, fewer hyperparameters
LSTM	Better, still rooms to improve	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for input, fixed for output	Better gradient flow with less vanishing	Complex, many hyperparameters
CNN	Not inherent	Highly parallelizable	Fast due to parallelizable operations	Less flexible, typically requires fixed-size inputs	Stable gradient flow	Complex, many hyperparameters
Bahdanau Attention	Good with attention mechanism	Limited, due to sequential processing	Slow, especially for long sequences	Flexible for both input and output	Better gradient flow with less vanishing	Simpler, fewer hyperparameters
Transformer	**Good with attention mechanism**	**Highly parallelizable**	**Fast due to parallelizable operations**	**Flexible for both input and output**	**Stable gradient flow**	**Simpler, fewer hyperparameters**

Table of Contents


Decoder-only
Output Text
Decoder
Input Text
110 M
GPT-1
117M-1.5B
GPT-2
175B
GPT-3
175B
ChatGPT
Unknown
GPT-4
110M-340M
XLNet
1.2T
GLaM
175B
OPT
175B
OPT-IML
137B
Bard
Gopher
280B
LaMDA
137B
PaLM
540B
Flan PaLM
540B
LLaMa
7B-65B
Vicuna
7B
Alpaca
7B-13B
Encoder-decoder
Output Text
Decoder
Features
Encoder
Input Text
110M-10B
GLM
130B
GLM-130B
140M
BART
4.1B-269B
ST-MoE
20B
UL2
20B
Flan-UL2
11B
T0
T5
80M-11B
mT5
300M-13B
Switch
1.6T
Flan-T5
80M-11B
Encoder-only
Features
Encoder
Input Text
110M-340M
BERT
66M
DistillBert
11M-223M
ALBERT
14M-110M
ELECTRA
RoBERTA
125M-355M
ERNIE
114M
DeBERTa
44M-304M
Open-Source
Closed-Source
2018
2019
2020
2021
2022
2023

Language Model: Auto-regressive Models

- Auto-regressive Models (Decoder-only)
  - $x_{1:i} \Rightarrow \phi(x_{1:i}), p(x_{i+1} \mid x_{1:i})$.
  - Example: text autocomplete
  - [[CLS],the,movie,was]⇒great



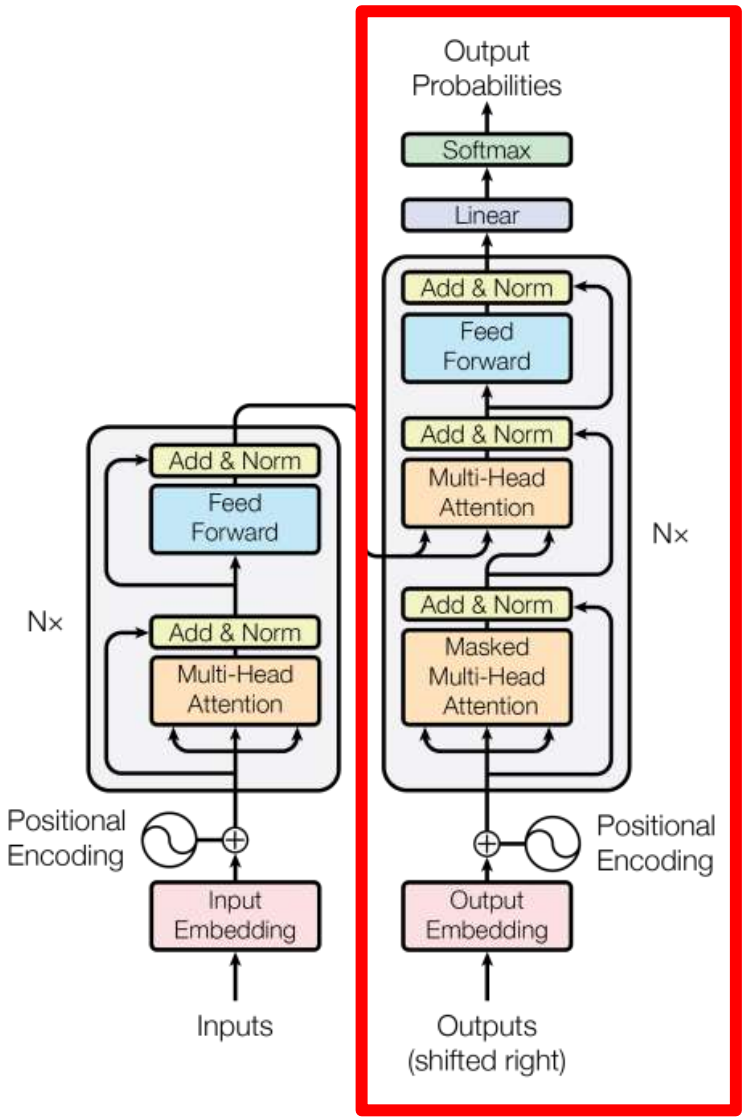
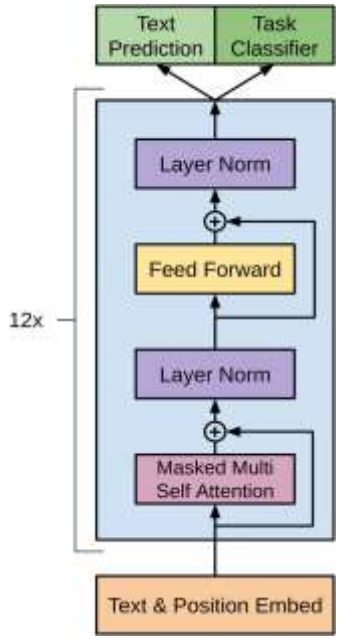


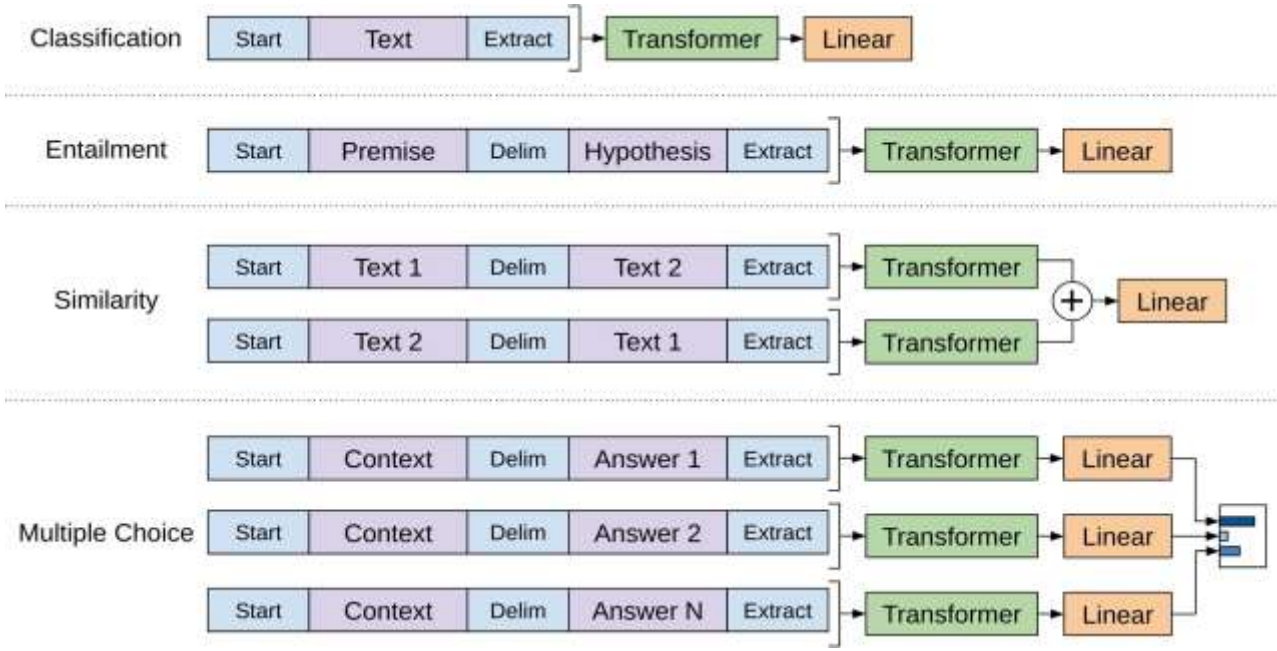
Image (left) source: Transformer: A General Framework from Machine Translation to Others

GPT

- Generative pre-training of a (left-to-right) language model on a diverse corpus of unlabeled text, followed by discriminative fine-tuning on each specific task.



GPT structure (Decoder)



Finetuning tasks

GPT - Pretraining

- Pretraining objective

$$\max_{\theta} \sum_{t} \log P(x^t | x^{t-k}, \cdots, x^{t-1})$$

- A multi-layer Transformer decoder is used for the language model

$$h_0 = UW_e + W_p$$
$$h_l = \text{TransformerBlock}(h_{l-1}) \forall i \in [1, n]$$
$$P(X) = \text{softmax}(h_n W_e^T)$$

where $X = (x^1, \cdots, x^k)$ is the sequence of tokens, n is the number of layers, W_e is the token embedding matrix, and W_p is the position embedding matrix.

GPT - Finetuning

- Given a labeled dataset D, where each instance consists of a sequence of input tokens, $(x^1, \cdots, x^m)$ along with a label y.
- The inputs are passed through the pre-trained model to obtain the final transformer block's activation of the final token h_l^m, which is then then fed into an added linear output layer to predict the class.

$$P(y|x^1, \cdots, x^m) = \text{softmax}(h_l^m W_y)$$

- The objective is to maximize the log probability of the correct class

$$\max_{\theta} \sum_{(x,y)\in D} \log P(y|x^1, \cdots, x^m)$$

GPT-2

- Language models can learn natural language processing tasks without any explicit supervision when trained on webpages.

"I'm not the cleverest man in the world, but like they say in French: **Je ne suis pas un imbecile [I'm not a fool].**

In a now-deleted post from Aug. 16, Soheil Eid, Tory candidate in the riding of Joliette, wrote in French: "**Mentez mentez, il en restera toujours quelque chose**," which translates as, "**Lie lie and something will always remain.**"

"I hate the word '**perfume**,"' Burr says. 'It's somewhat better in French: '**parfum**.'

If listened carefully at 29:55, a conversation can be heard between two guys in French: "**-Comment on fait pour aller de l'autre coté? -Quel autre coté?**", which means "**- How do you get to the other side? - What side?**".

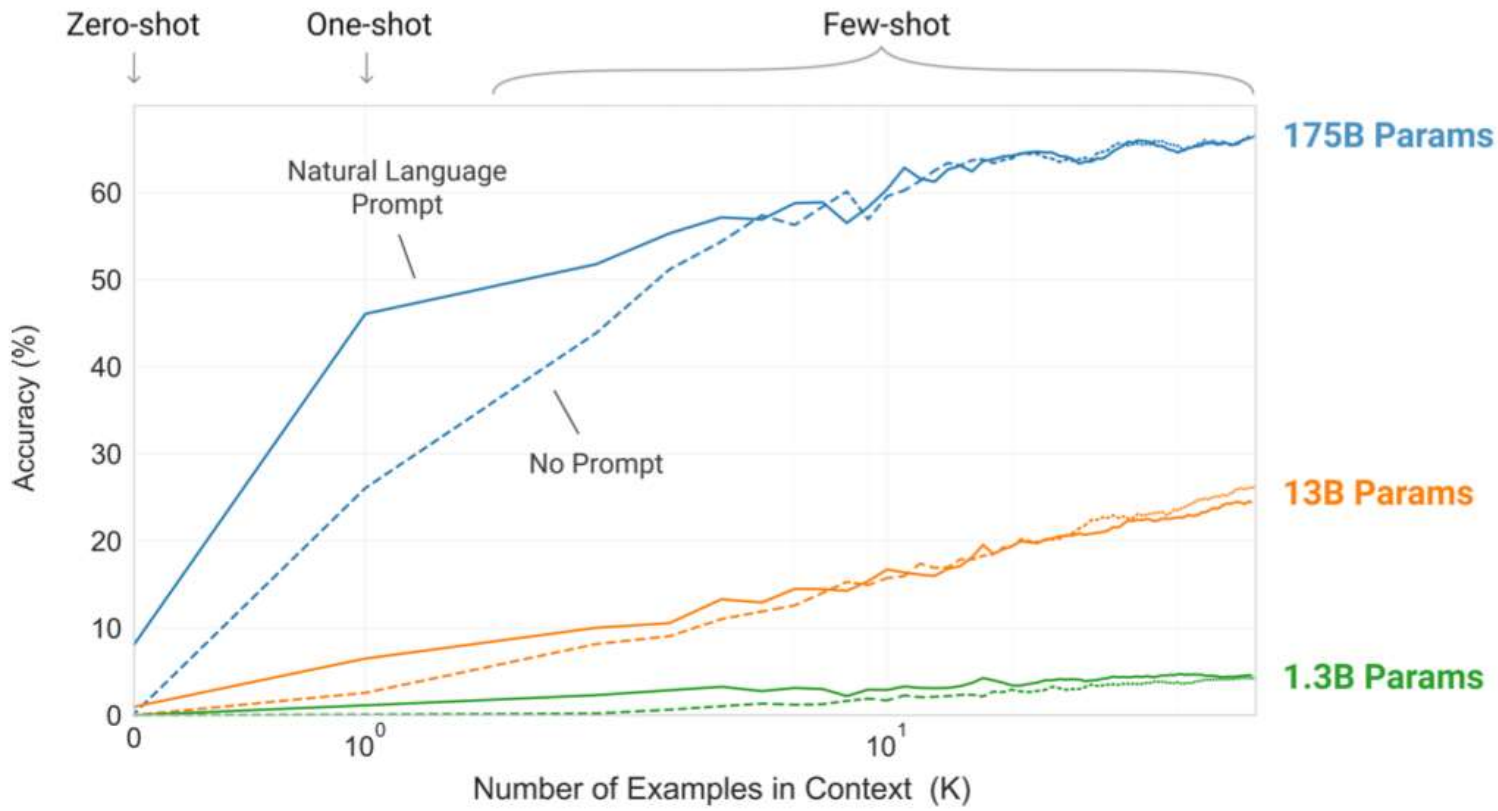
If this sounds like a bit of a stretch, consider this question in French: **As-tu aller au cinéma?**, or **Did you go to the movies?**, which literally translates as Have-you to go to movies/theater?

"**Brevet Sans Garantie Du Gouvernement**", translated to English: "**Patented without government warranty**".

Language Models are Unsupervised Multitask Learners. (OpenAI, Radford et al.)

GPT-3

- GPT-3 scales up language models to 175B parameters, and greatly improves few-shot performance.

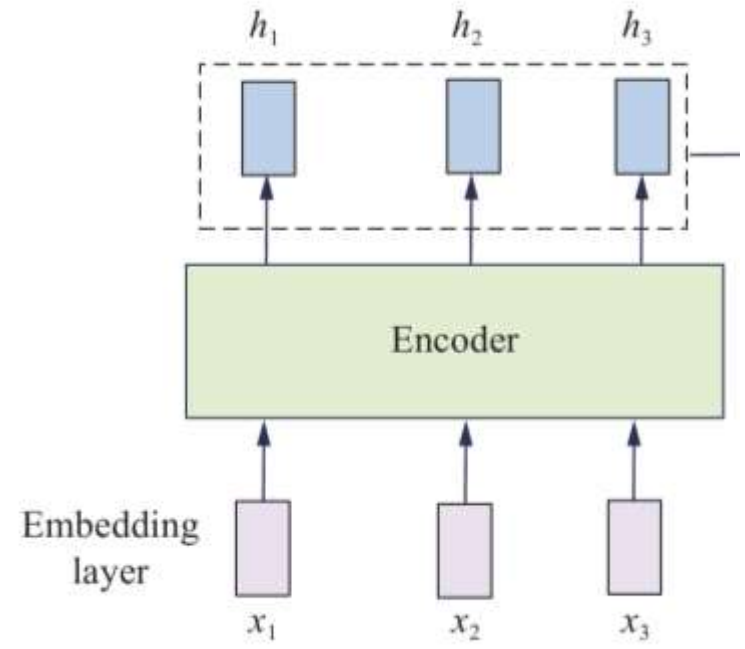


Language Models are Few-Shot Learners. (NeurIPS 2020, Brown et al.)

Language Model: Auto-encoding Models

- Auto-encoding Models (Encoder-only)
  - $x_{1:L} \Rightarrow \phi(x_{1:L})$.
  - Example: sentiment classification

    [[CLS],the,movie,was,great]⇒positive



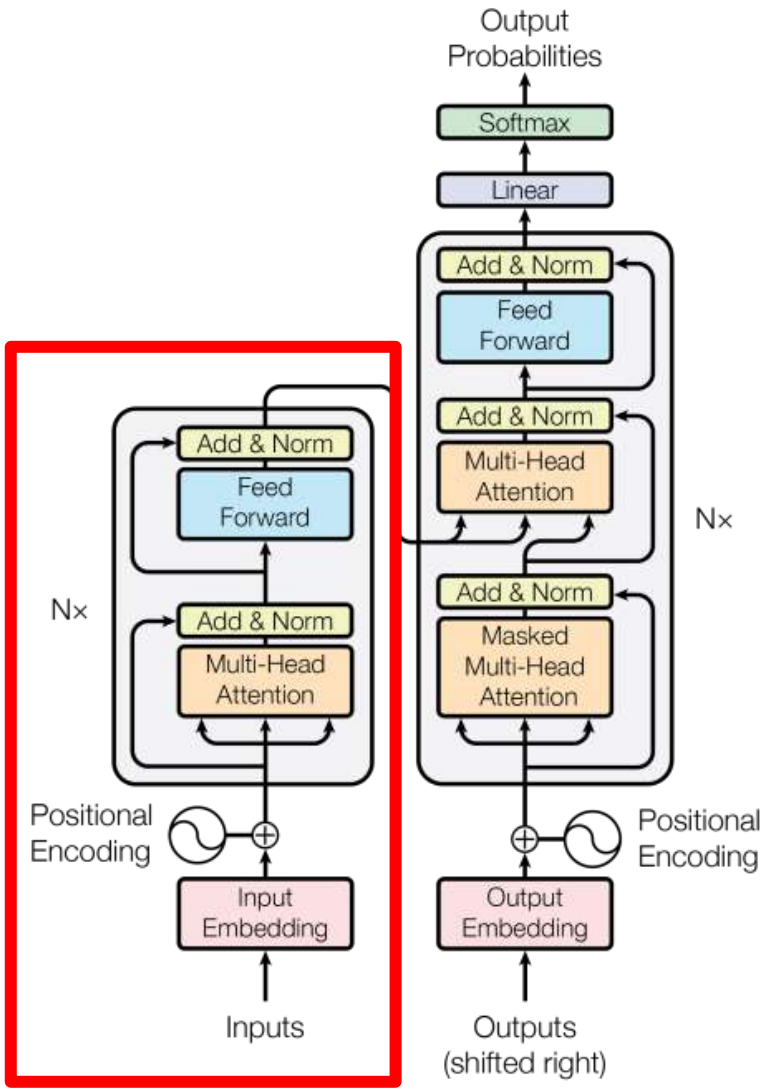


Image (left) source: Transformer: A General Framework from Machine Translation to Others

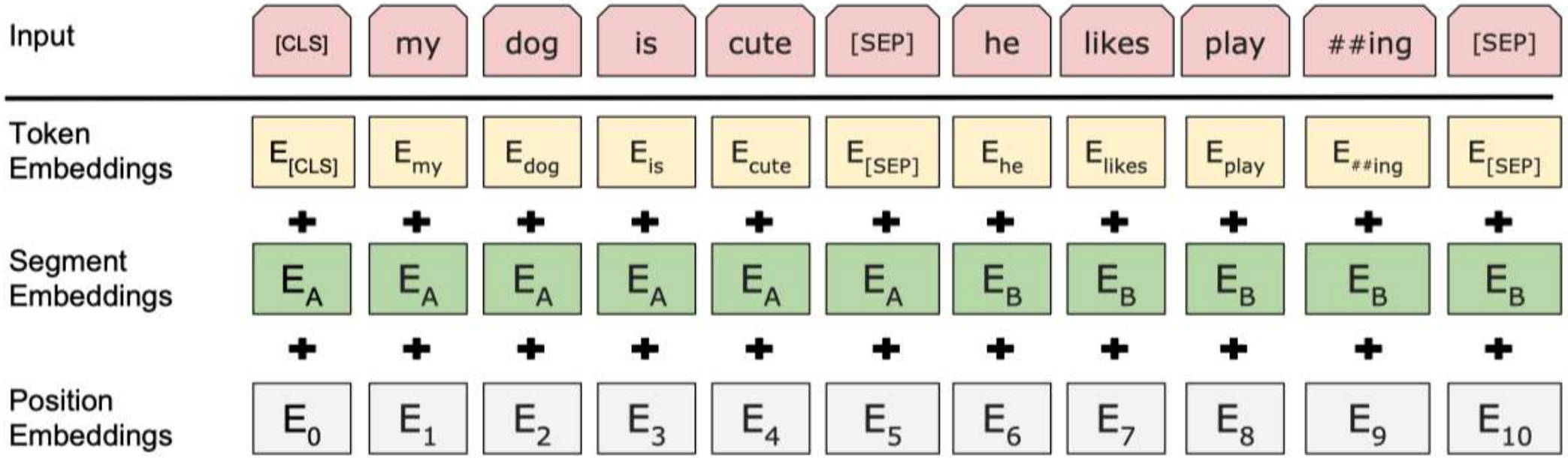
BERT

- Language models are unidirectional, in which every token can only attend to previous tokens in the self-attention layers of the Transformer.
- For token-level tasks such as question answering, it is crucial to incorporate context from both directions.
- Instead, BERT uses a “masked language model” (MLM) pre-training objective. The masked language model randomly masks some of the tokens from the input, and the objective is to predict the original vocabulary id of the masked word based only on its context.

BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. (NAACL 2019, Devlin et al.)

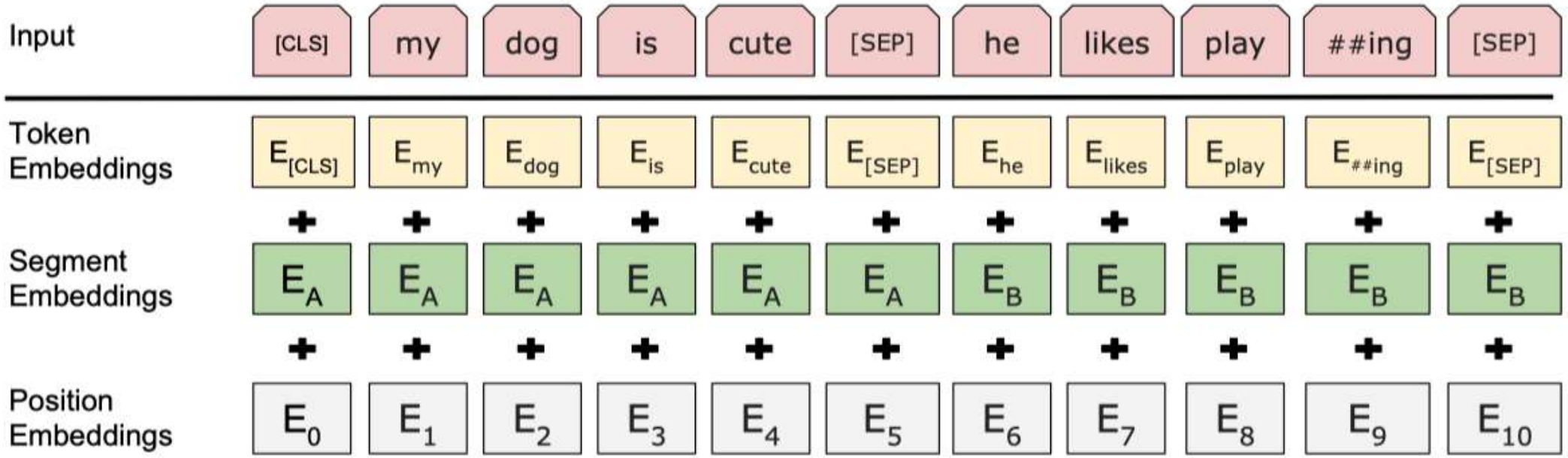
BERT

The input sequence can be a single sentence or two sentences packed together.
The first token of every sequence is always a special classification token ([CLS]). The final hidden state corresponding to this token is used as the aggregate sequence representation for classification tasks.

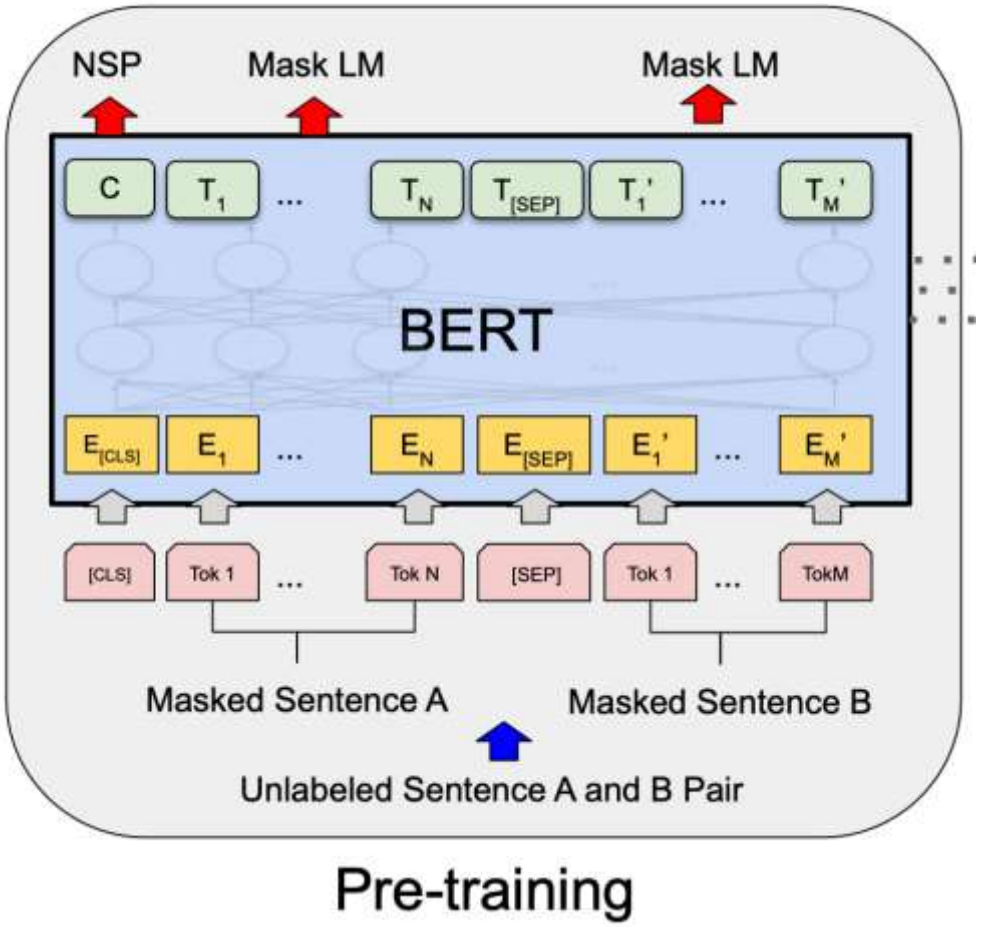


BERT

Sentence pairs are packed together into a single sequence, separated with a special token ([SEP]).
BERT adds a learned embedding to every token indicating whether it belongs to sentence A or sentence B.

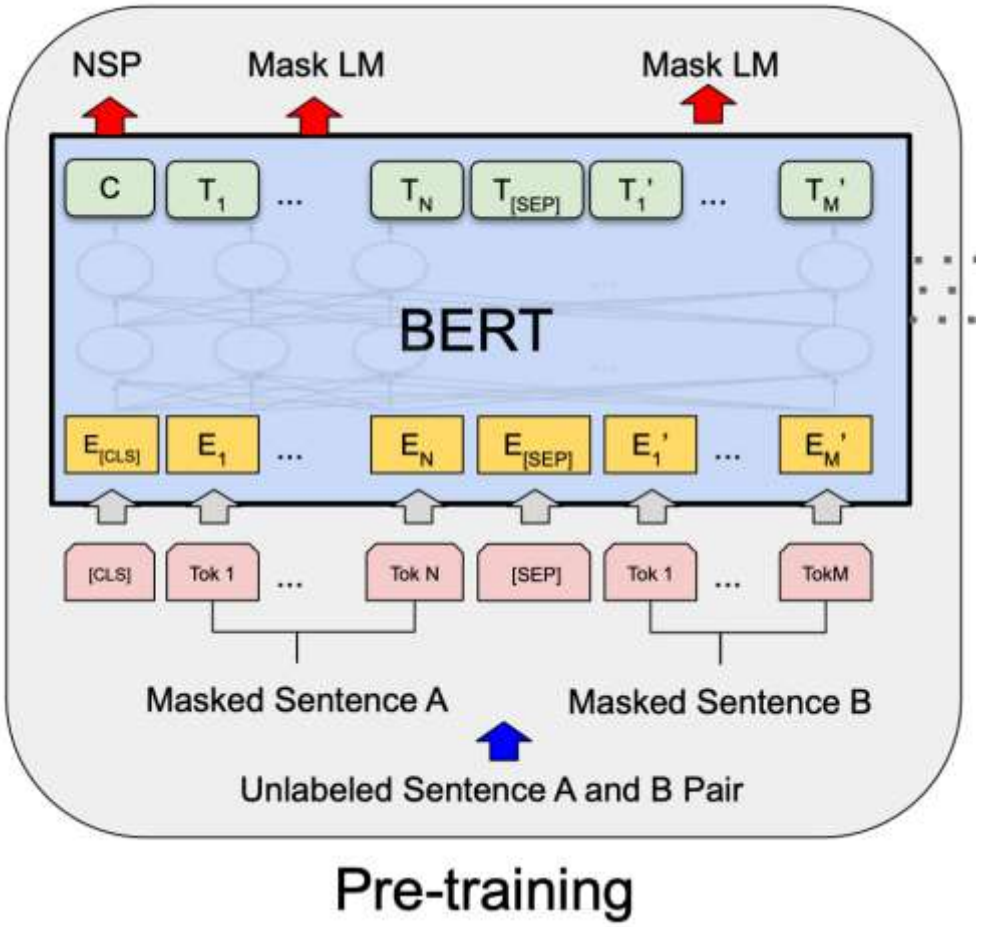


Task1: Masked Language Model



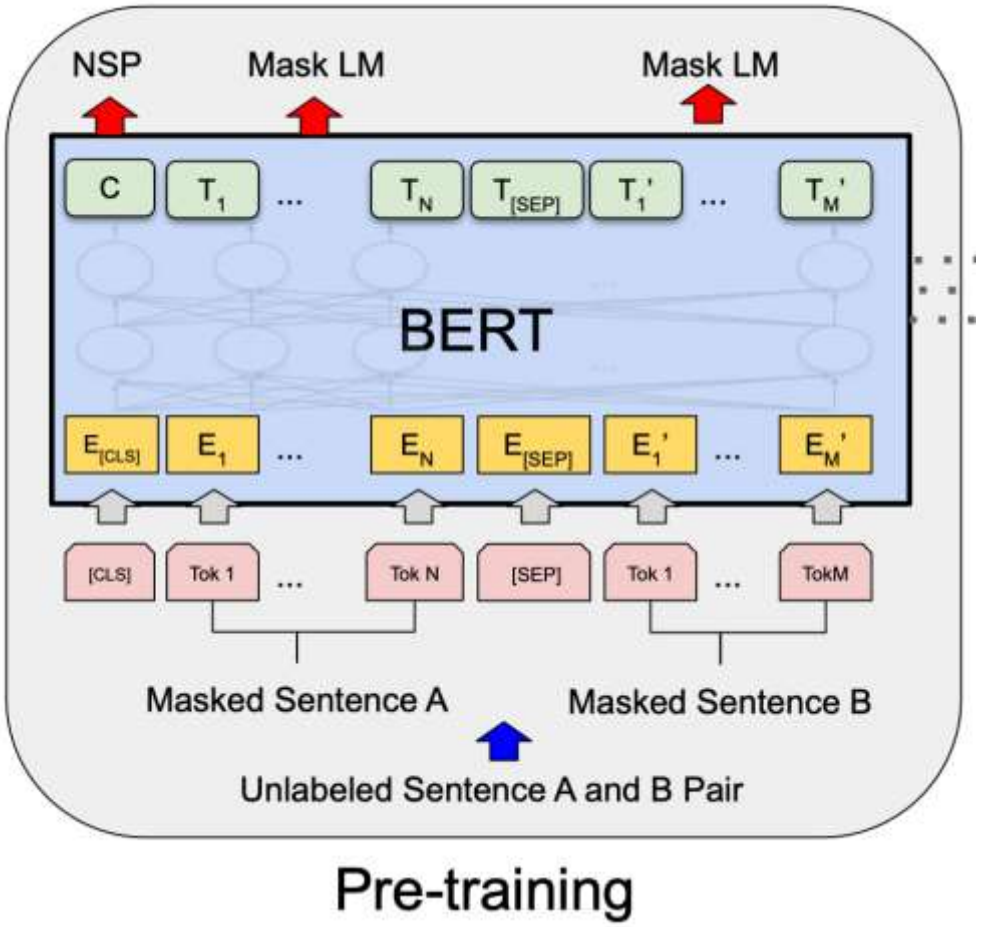
- Replace 15% of the input tokens at random with [MASK] tokens and then predict the original tokens with final hidden states of [MASK] tokens.

Task2: Next Sentence Prediction



- Many NLP tasks such as question answering and natural language inference are based on understanding the relationship between two sentences.
- In order to train a model that understands sentence relationships, BERT adds a binarized next sentence prediction (NSP) task.

Task2: Next Sentence Prediction



- When choosing the sentences A and B for each pretraining example, 50% of the time B is the actual next sentence that follows A (labeled as *IsNext*), and 50% of the time it is a random sentence from the corpus (labeled as *NotNext*).
- The final hidden state of [CLS] token is used for next sentence prediction.

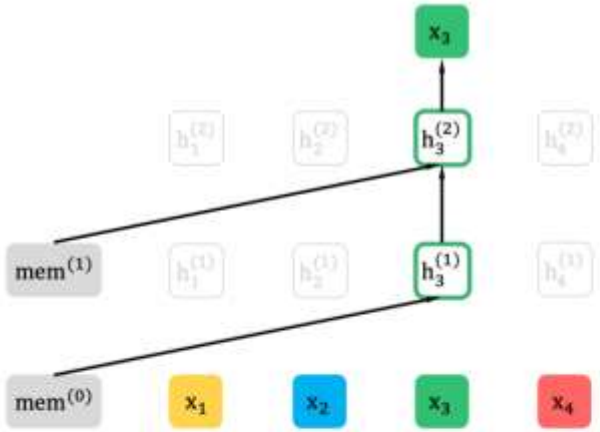
XLNet

- BERT overcomes autoregressive language models with the capability of modeling bidirectional contexts.
- However, relying on corrupting the input with masks, BERT ignores dependency between the masked positions.
- XLNet proposes the permutation language modeling objective that retains both the benefits of autoregressive models and bidirectional models

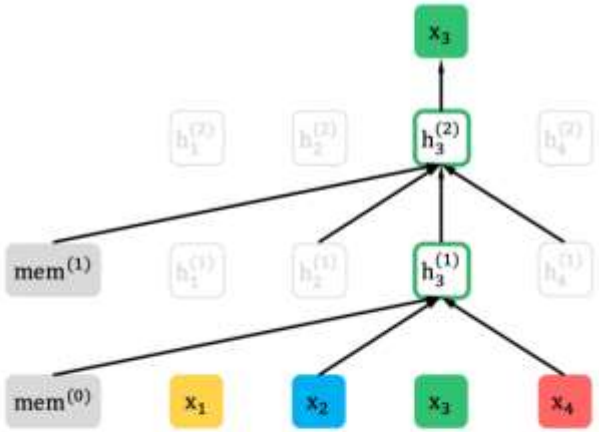
$$\max_{\theta} \mathbb{E}_{\boldsymbol{z}\sim\mathcal{Z}_m}[\sum_{t=1}^{m} \log P_\theta(x_{z_t}|\boldsymbol{x}_{\boldsymbol{z}<t})]$$

where $\mathcal{Z}_t$ is the set of all possible permutations of the length-T index sequence $[1,\cdots,T]$.

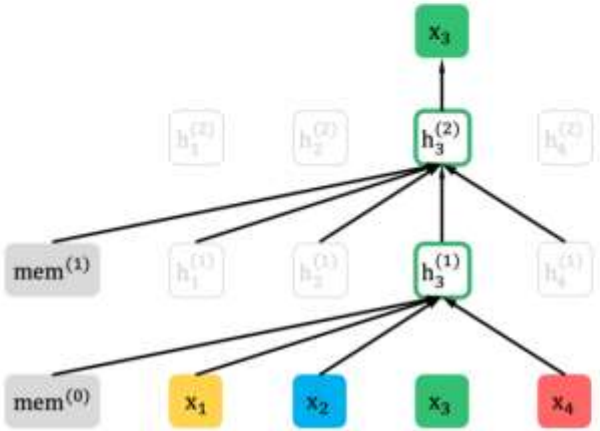
XLNet



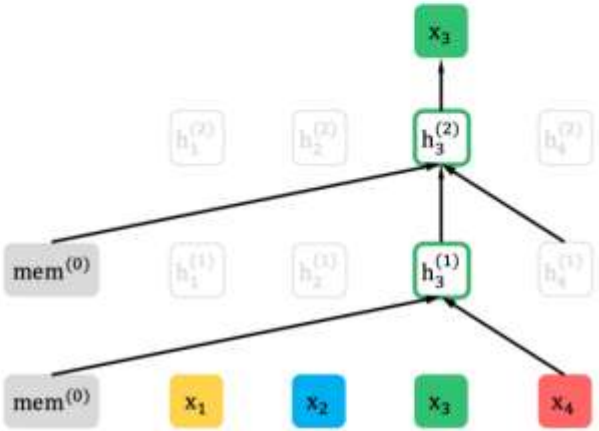
Factorization order: 3 → 2 → 4 → 1



Factorization order: 2 → 4 → 3 → 1



Factorization order: 1 → 4 → 2 → 3



Factorization order: 4 → 3 → 1 → 2

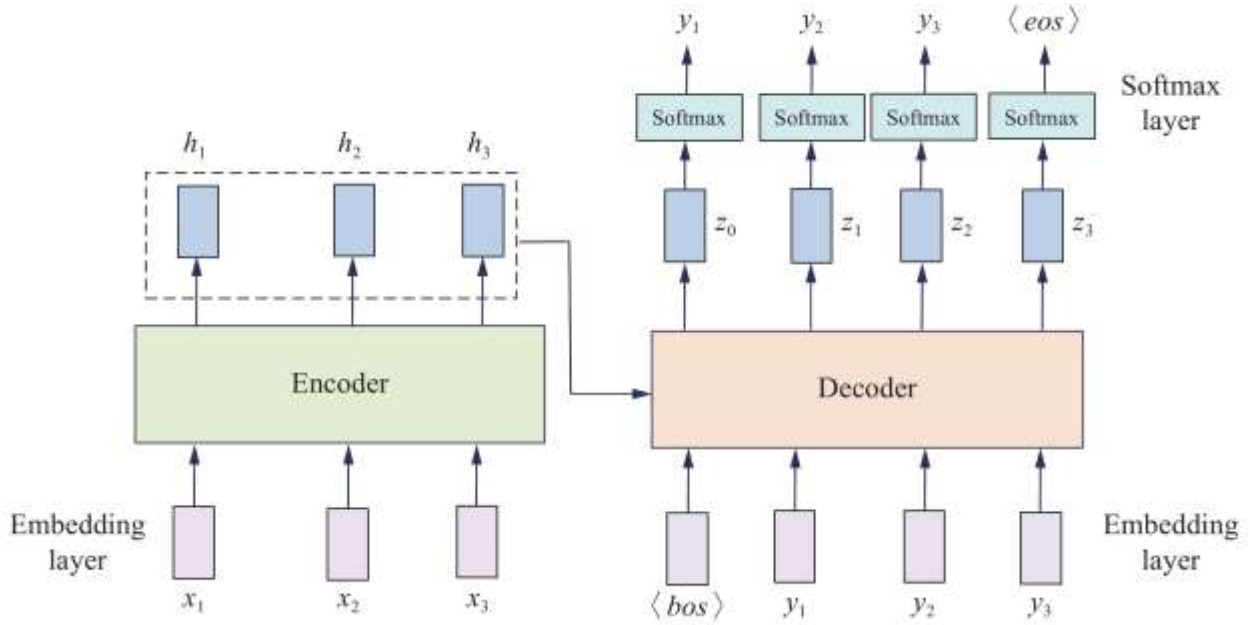
Intuitively, if model parameters are shared across all factorization orders, in expectation, the model will learn to gather information from all positions on both sides.

Language Model: Encoder-Decoder Models

- Encoder-Decoder Models
  - $x_{1:L} \Rightarrow \phi(x_{1:L}), p(y_{1:L} \mid \phi(x_{1:L}))$.
  - Example: table-to-text generation

    [name,:,Clowns,|,eatType,:,coffee,shop]

    ⇒[Clowns,is,a,coffee,shop].



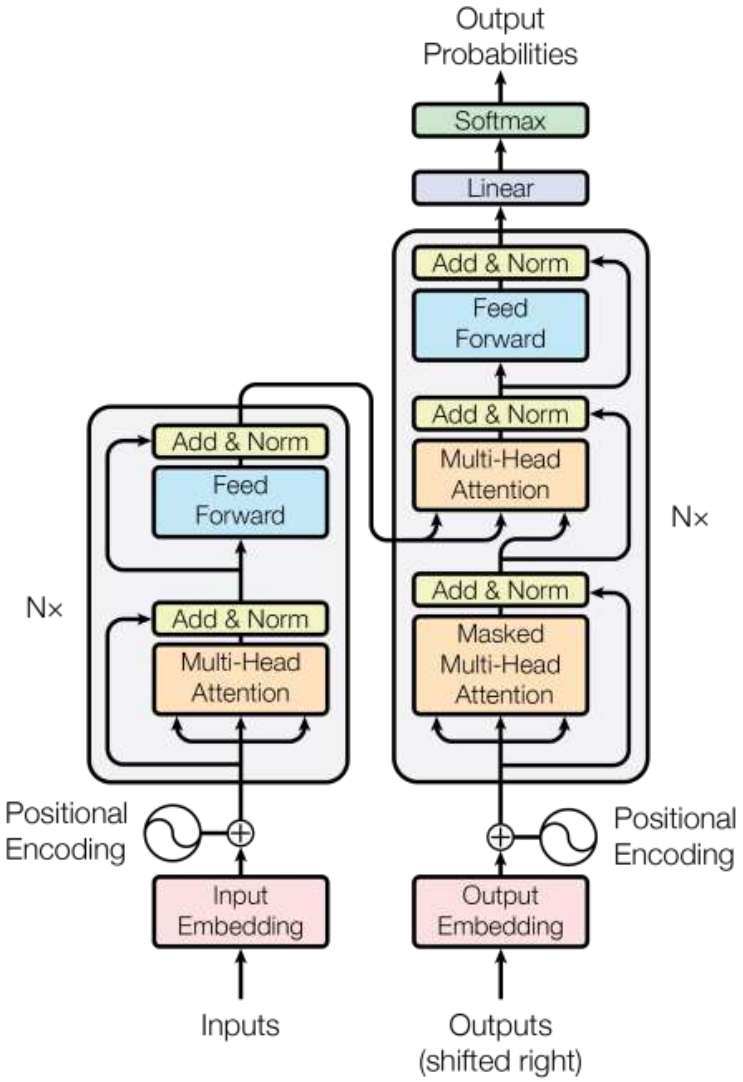
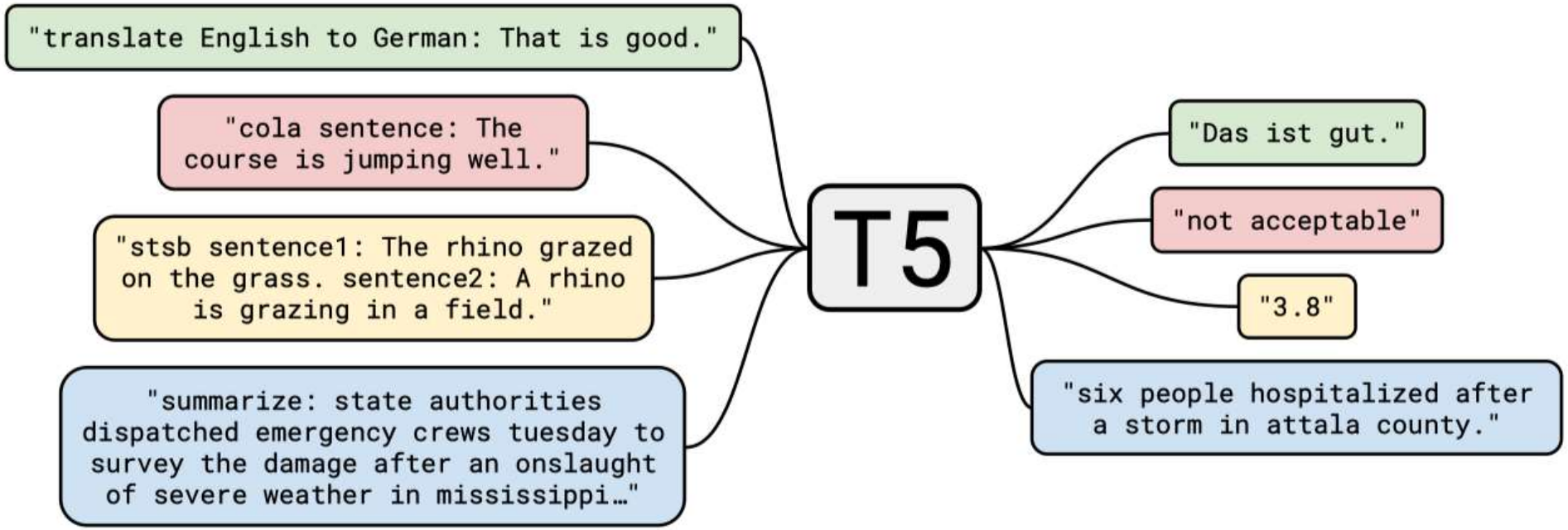


Image (left) source: Transformer: A General Framework from Machine Translation to Others

T5

- An encoder-decoder Transformer is pre-trained with the span-corruption objective.
- Different NLP tasks are converted into a unified text-to-text format.



Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. (JMLR, Raffel et al.)

Summary

- **Auto-regressive Models (Decoder-only)**

  Given a prompt $x_{1:i}$, produces both contextual embeddings and a distribution over next tokens x_{i+1}(and recursively, over the entire completion $x_{i+1:L}$).

  - $x_{1:i} \Rightarrow \phi(x_{1:i}), p(x_{i+1} \mid x_{1:i})$.
  - Example: text autocomplete
    [[CLS],the,movie,was]⇒great[[CLS],the,movie,was]⇒great

- Con: contextual embedding for xi can only depend **unidirectionally** on both the left context ($x_{1:i-1}$).
- Pro: can naturally generate completions.
- Pro: simple training objective (maximum likelihood).

Summary

- **Auto-encoding Models (Encoder-only)**

  These language models produce contextual embeddings but cannot be used directly to generate text.

  - $x_{1:L} \Rightarrow \phi(x_{1:L})$.
  - Example: sentiment classification

    [[CLS],the,movie,was,great]⇒positive.[[CLS],the,movie,was,great]⇒positive.

- Pro: contextual embedding for x_i can depend **bidirectionally** on both the left context ($x_{1:i-1}$) and the right context ($x_{i+1:L}$).
- Con: cannot naturally **generate** completions.
- Con: requires more **ad-hoc training** objectives (masked language modeling).

Summary

- **Encoder-Decoder Models**

  They can use bidirectional contextual embeddings for the input $x_{1:L}$ and can generate the output $y_{1:L}$.

  - $x_{1:L} \Rightarrow \phi(x_{1:L}), p(y_{1:L} \mid \phi(x_{1:L}))$.
  - Example: table-to-text generation

    [name,:,Clowns,|,eatType,:,coffee,shop]⇒[Clowns,is,a,coffee,shop].

- Pro: contextual embedding for x_i can depend **bidirectionally** on both the left context ($x_{1:i-1}$) and the right context ($x_{i+1:L}$).
- Pro: can naturally **generate** outputs.
- Con: requires more **ad-hoc training** objectives.

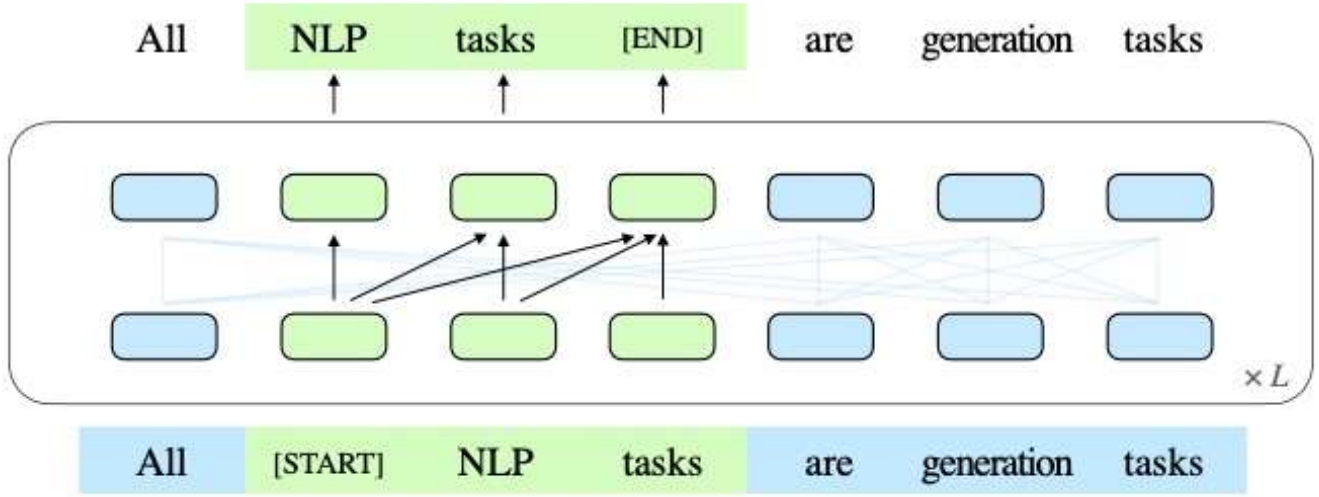
Motivation

- None of the pretraining frameworks performs the best for all tasks.

Architecture	NLU (Sentiment Analysis, Extractive QA, …)	Seq2Seq (Summarization,…)	Unconditional Generation (Language Modeling, …)
Autoencoding	✓	×	×
Autoregressive	—	—	✓
Encoder-Decoder	—	✓	—

- As the scale of the pretrained LM increases, the high computational cost of pretraining makes it infeasible to develop different models for different types of downstream tasks.

General Language Model (GLM)



- We propose General Language Model (GLM), unifying different pre-training architectures with autoregressive blank filling.
- Experimental results show that GLM can be adapted to different types of downstream tasks.

GLM: General Langauge Model Pretraining with Autoregressive Blank Infilling. (ACL 2022, Du et al.)

Autoregressive Blank Filling

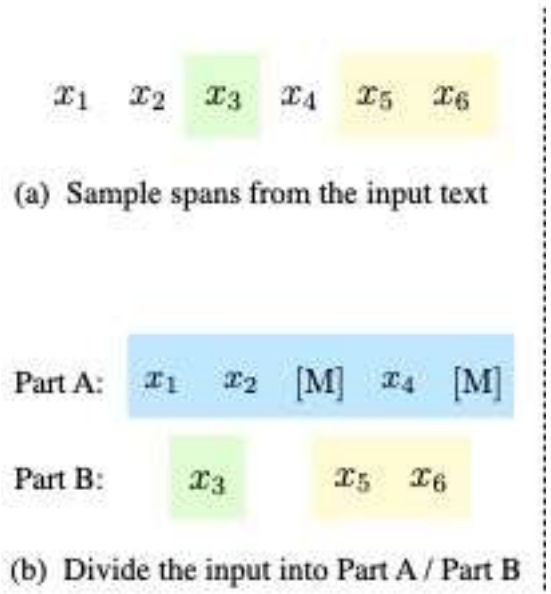
- Given an input text $\boldsymbol{x} = [x_1, \cdots, x_m]$, multiple text spans $\{s_1, \cdots, s_n\}$ are sampled. Each span is replaced with a single [MASK] token, forming a corrupted text $\boldsymbol{x}_{\text{corrupt}}$.
- Formally, let Z_n be the set of all possible permuations of the length-n index sequence, the objective is

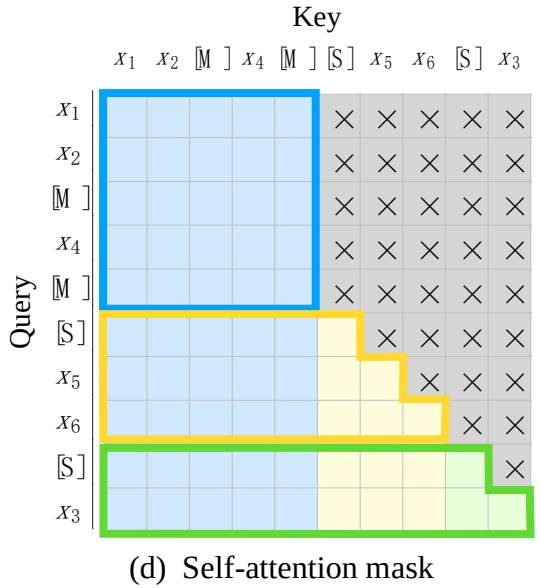
$$\max_{\theta} \mathbb{E}_{z \sim Z_n} [\sum_{i=1}^{n} \log p_\theta(\boldsymbol{s}_{z_i} | x_{\text{corrupt}}, \boldsymbol{s}_{\boldsymbol{z}_{<i}})]$$

- In a single span, GLM predicts the missing tokens in an autoregressive manner following the left-to-right order.

$$p_\theta(\boldsymbol{s}_{z_i} | \boldsymbol{x}_{\text{corrupt}}, s_{z_{<i}}) = \prod_{j=1}^{l_i} p(s_{z_i,j} | \boldsymbol{x}_{\text{corrupt}}, \boldsymbol{s}_{z_{<i}}, \boldsymbol{s}_{i,<j})$$

Autoregressive Blank Filling



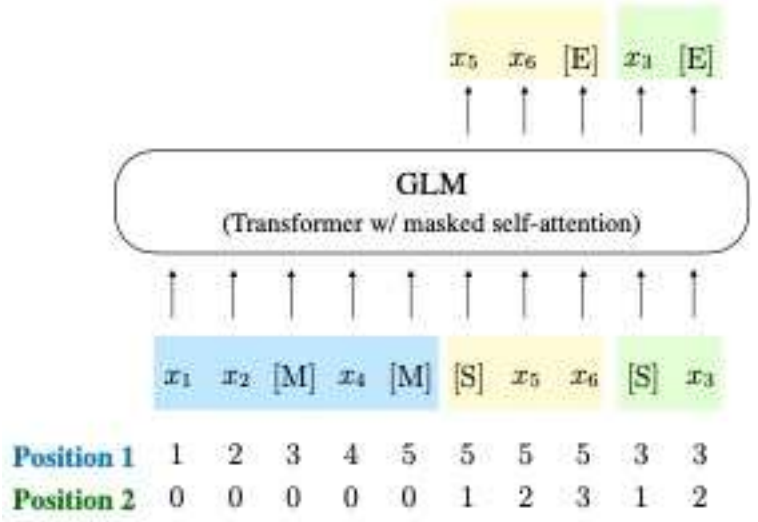


- The input x is divided into two parts: Part A consists of the corrupted text $x_{corrupt}$, and Part B consists of the masked spans.
- Tokens in Part A can attend all the tokens in A, but cannot attend to any tokens in B. Tokens in Part B can attend to tokens in A and its antecedents in B, but cannot attend to any subsequent tokens in B.

Unified Pre-training Objective

- We design three pre-training objectives corresponding to three categories of downstream tasks:
  - Token-level objective: we repeatedly sample the length of each span from the Poission distribution with $\lambda = 3$ until at least 15% of the original tokens are corrupted. (NLU)
  - Sentence-level objective: The masked spans must be full sentences. Multiple sentences are sampled to cover 15% of the original tokens. (Seq2Seq)
  - Document-level objective: we sample a single span whose length is sampled from a uniform distribution over 50%-100% of the original length. (Language Modeling)

2D Positional Encoding



- One of the challenges of the autoregressive blank filling task is how to encode the positional information. We propose 2D positional encodings to address the challenge.
- Each token is encoded with two position ids. The first positional id represents the position in the corrupted text. The second positional id represents the intra-span position.

Thank you!

Jie Tang, KEG, Tsinghua University

http://keg.cs.tsinghua.edu.cn/jietang
https://github.com/THUDM