Chapter 2:
Large Language Models

Ernest K. Ryu

University of California, Los Angeles

Natural language processing (NLP)

Natural language processing (NLP) is concerned with computationally processing natural (human) languages. The goal is to design and/or train a system that can understand, and process information written in documents.

A *natural language* or *ordinary language* is any language that has evolved naturally in humans through use and repetition without conscious planning or premeditation such as English or Korean. They are distinguished from formal and constructed languages such as C, Python, Lojban, and Esperanto.

NLP was once a field that relied on insight into linguistics, but modern NLP is dominated by data-driven deep-learning based approaches.

Task: Sentiment analysis

Given a review $X \in \mathcal{X}$ on a reviewing website, decide whether its label $Y \in \mathcal{Y} = \{-1, 0, +1\}$ is negative (-1), neutral (0), or positive $(+1)$.

Eg.

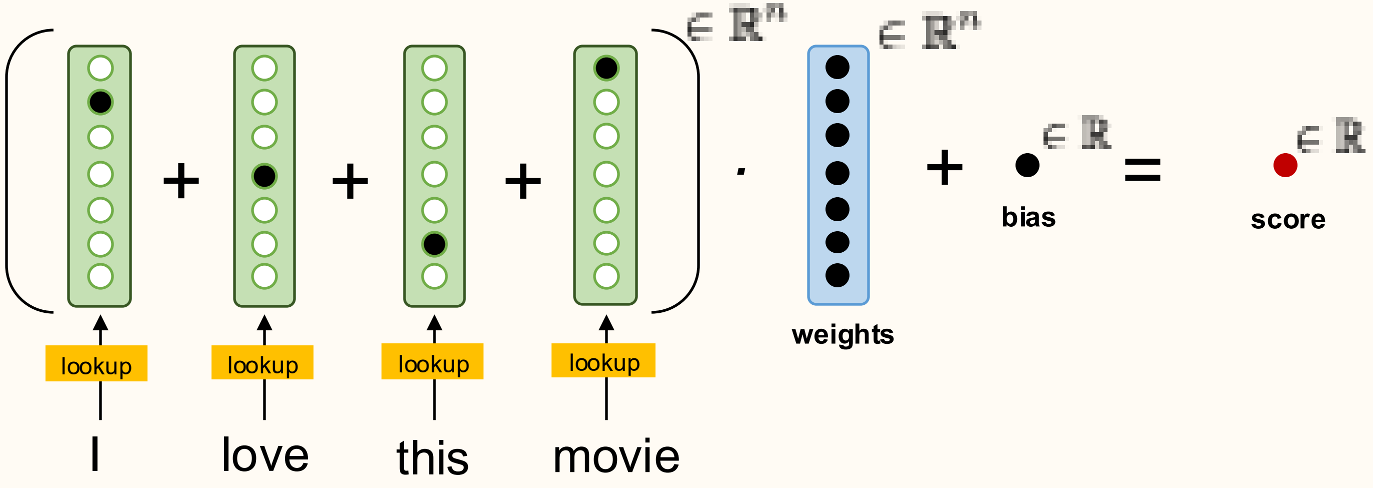
Review: I do not like this movie
Sentiment: Negative

Review: I love this movie
Sentiment: Positive

Input is variable-length. Output is fixed-size.

Sentiment analysis with BOW

A bag of words (BOW) model makes the prediction with a linear combination of tokenized word. This is a simple baseline.



More generally “bag of words” refers to models that view a sentence as an unordered collection (bag) of words. Completely disregarding word order is a significant drawback of the method.

Sequence (seq) notation

Let $\mathcal{U}$ be any set. Define k-tuples of $\mathcal{U}$ as

$$\mathcal{U}^k = \{(u_1, \ldots, u_k) | u_1, \ldots, u_k \in \mathcal{U}\}$$

The *Kleene star* notation

$$\mathcal{U}^* = \bigcup_{k \geq 0} \mathcal{U}^k = \{(u_1, \ldots, u_k) | u_1, \ldots, u_k \in \mathcal{U}, k \geq 0\}$$

denotes sequences of $\mathcal{U}$ of arbitrary finite length.

Although unimportant in most practical setups, we define the empty sequence $()$ is a valid sequence of length 0 and write $() \in \mathcal{U}^*$.

Characters

Let $\mathcal{C}$ be a set of “characters”.

- $\mathcal{C}$ can be the set of English characters, space, and some punctuation.
- $\mathcal{C}$ can be the set of all unicode characters.

Let $\mathcal{X} = \mathcal{C}^*$ be the set of finite-length sequence of characters, i.e., $X \in \mathcal{X}$ is raw text.

(The definition and scope of “characters” is not obvious, and different choices present different trade-offs. The modern approach is the view each “byte” of the unicode encoding to be the indivisible basic unit. More on this when we cover byte pair encodings.)

Tokenization

Neural networks perform arithmetic on vectors and numbers, so tokenizers convert text into a sequence of vectors.

Given $X = (c_1, \ldots, c_T) \in \mathcal{C}^*$, a *tokenizer* is a function $\tau : \mathcal{C}^* \to (\mathbb{R}^n)^*$ such that

$$\tau(c_1, c_2, \ldots, c_T) = (u_1, u_2, \ldots, u_L)$$

where $u_1, u_2, \ldots, u_L \in \mathbb{R}^n$. T and L are often not the same. (Usually $T \geq L$.)Sometimes τ is fixed, and sometimes it is trainable (e.g. word2vec).

For text generation, we want the tokenizer to be invertible.

Character-level tokenizer v.0

Example: $\mathcal{C} = \{a, b, \ldots, z, \_, ., ?, !\}$ and

$$\tau(X) = \tau(c_1, \ldots, c_L) = \left(\tau(c_1), \ldots, \tau(c_L)\right)$$

$$\tau(a) = 1, \quad \tau(b) = 2, \quad \ldots \quad \tau(z) = 26, \quad \ldots$$

So $n = 1$ and $L = T$.

This doesn’t work very well.

We want distinct tokens to be vectors of distinct directions. Neural networks are better at distinguishing directions than magnitudes.

Character-level tokenizer v.1

Example: $\mathcal{C} = \{a, b, \dots, z, \_, ., ?, !\}$

$$\tau(X) = \tau(c_1, \dots, c_L) = \big(\tau(c_1), \dots, \tau(c_L)\big)$$

$$\tau(a) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \tau(b) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \dots \quad \tau(!) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

So $n = 30$ and $T = L$. The output vectors are called *one-hot-encodings* as only one element of the encoded vector is nonzero (hot).

Word-level tokenizer

Examples: $\mathcal{C} = \{a, b, \dots, z, \_\}$ (so English letters and space) and $\mathcal{W} = \{\text{English words}\}$

$$\tau(X) = \tau(c_1, \dots, c_T) = \tau(w_1, \dots, w_L) = \big(\tau(w_1), \dots, \tau(w_L)\big)$$

$$\tau(\text{‘aardvark’}) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \tau(\text{‘ability’}) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \quad \dots, \quad \tau(\text{‘Zyzzyva’}) = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}, \quad \dots$$

where $w_1, \dots, w_L \in \mathcal{W}$. So $n = |\mathcal{W}| = (\text{size of dictionary})$ and $L \leq T$.

I.e., this is a one-hot encoding of words.

End-of-string (EOS) token

Given $X \in \mathcal{X}$ and its length $0 \leq T < \infty$, we equivalently consider a special "end-of-string" token <EOS> to be the final $(T+1)$-th element. In other words,

$$X = (c_1, c_2, \dots, c_T) = (c_1, c_2, \dots, c_T, \text{<EOS>})$$

for any $X \in \mathcal{X}$, where $c_1, \dots, c_T \in \mathcal{C}$.

We use the same notation for elements in $\mathcal{U}^*$, i.e.,

$$(u_1, u_2, \dots, u_L) = (u_1, u_2, \dots, u_L, \text{<EOS>}) \in \mathcal{U}^*$$

Discussion on tokenizers

Q) Advantage of word-level tokenizer over character-level tokernizer?

A) Shorter tokenized sequence. Uses dictionary. (Model need not learn words from scratch.)

Q) Advantage of character-level tokenizer over word-level tokernizer?

A) Can learn to handle misspellings (‘learning’ ≈ ‘lerning’) and inflections (‘running’ = ‘run’ + ‘ing’). Better for multi-language models. (Dictionaries of multiple languages is too large.)

Q) Are there other tokenizers?

A) Word2Vec and subword tokenization (byte-pair encoding) are trained tokenizers.
More on these later.

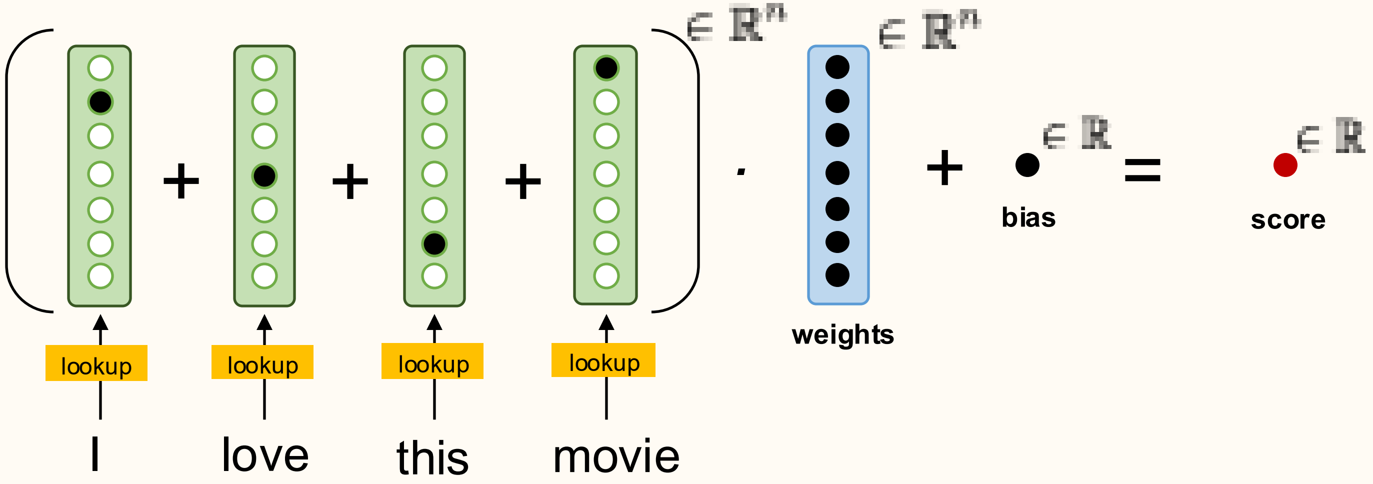
Basic BOW implementation

Let τ be a word-level tokenizer with dictionary $\mathcal{W}$.

For $X = (w_1, \ldots, w_L)$ the bag-of-word (BOW) model f_θ is

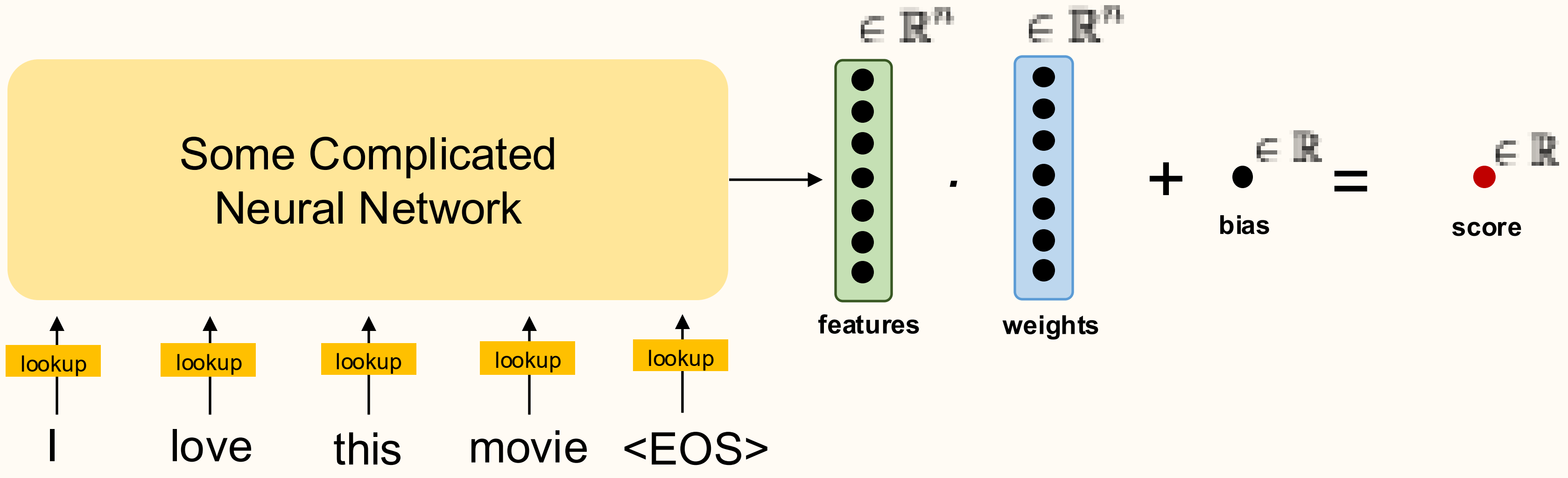
$$f_\theta(X) = b + a \cdot \sum_{\ell=1}^{L} \tau(w_\ell) = b + a \cdot \sum_{\ell=1}^{L} \big(\tau(X)\big)_\ell$$

where $\theta = (a, b) \in \mathbb{R}^{n+1}$ is the trainable parameter.



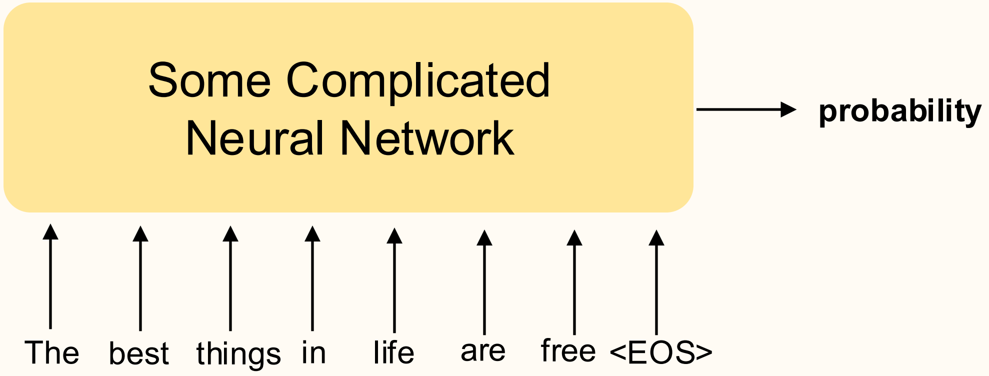
Sentiment analysis with DNN

Modern state-of-the-art NLP methods are based on deep neural networks (DNN).



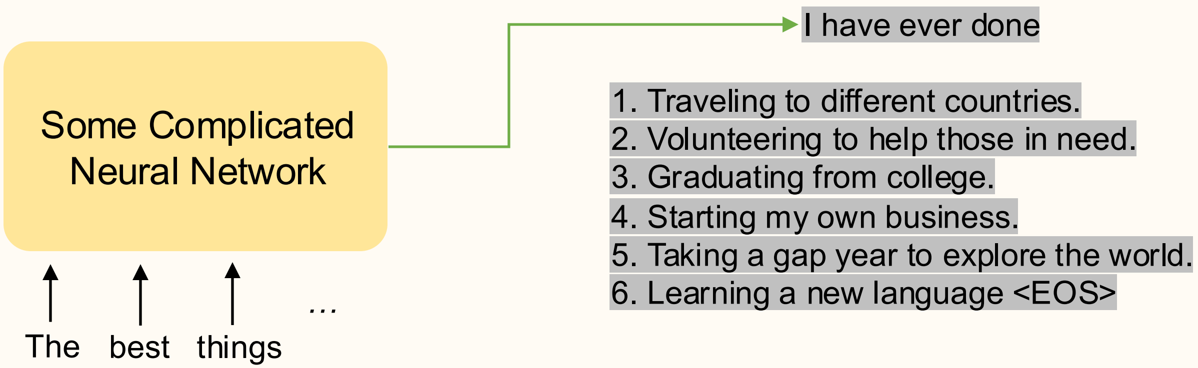
Task: Language model (LM)

A *language model* (LM) achieves one or two of the following goals.



Goal 1: Assign probabilities/likelihoods to sentences.

Goal 2: Generate likely (coherent) sentences.



Why is an LM useful? Prior to ChatGPT, this was not at all an obvious question.

(This definition excludes encoder-only transformer models such as BERT, but we will not be overly concerned with these definitions.)

Applications of LM: Voice-to-text

In a voice-to-text system, two interpretations can be auditorily ambiguous but semantically not ambiguous. An LM can determine which interpretation is more likely.

“The parcel was secured by grey tape.” (✓)

“The parcel was secured by great ape.”

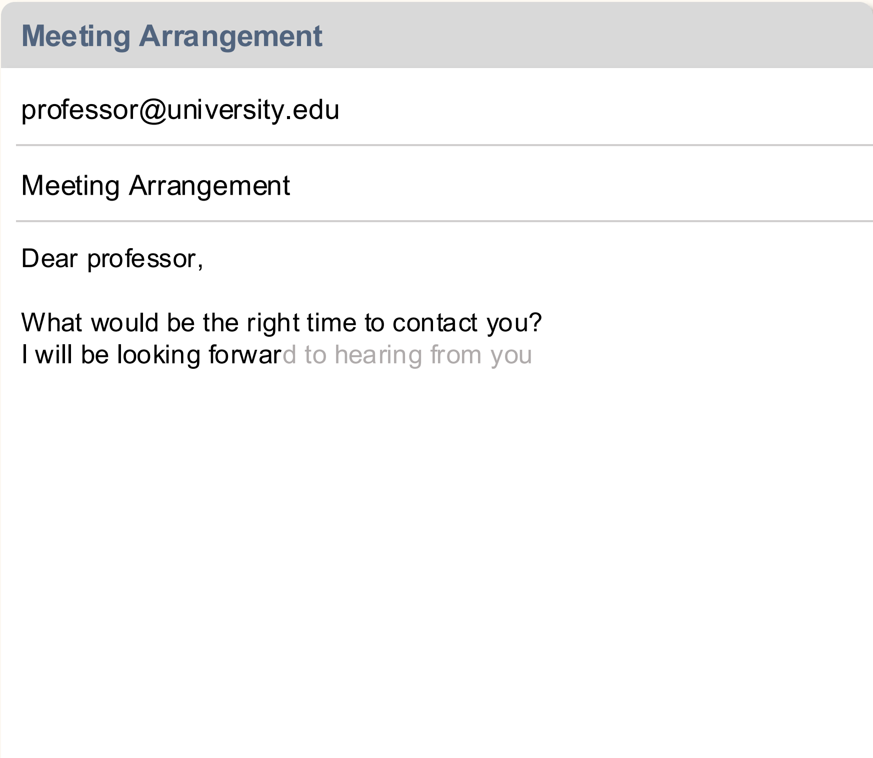
“he was a lighthouse keeper” (✓)

“he was a light housekeeper”

A similar application with spelling correction.

Applications of LM: Autocomplete

An autocomplete system can assist writing by suggesting likely completions of a sentence.



Applications of LM: SSL pre-training and universal interface

Training a NN to be a language model is a useful *pretext task* for *transfer learning* in the sense of self-supervised learning (SSL). Pre-trained language models serve as *foundation models* that can be *fine-tuned* for other downstream tasks.

- More on this when we talk about ELMo, BERT, and GPT 1.

A sufficiently powerful LM can serve as a universal language-based interface to the capabilities that the language model has learned.

- More on this when we talk about T5 and GPT3.

Probabilities with sequences

Assume a sequence

$$(u_1, u_2, \dots, u_L) = (u_1, u_2, \dots, u_L, \text{<EOS>}) \in \mathcal{U}^*$$

is generated randomly, i.e., we can assign a probability

$$\mathbb{P}\big((u_1, u_2, \dots, u_L, \text{<EOS>})\big) \in [0,1]$$

The sequence length L is also a random variable. Imagine $u_1, u_2, \dots$ being generated sequentially. Given $u_1, u_2, \dots, u_\ell$, the next token may be $u_{\ell+1}=\text{<EOS>}$ and the sequence terminates. Otherwise, $u_{\ell+1} \neq \text{<EOS>}$ and the generation continues to $u_{\ell+2}$.

Probability notation with <EOS>

Clarification) Given $0 \leq L < \infty$ and $u_1, u_2, \ldots, u_L \in \mathcal{U}$,

$$\mathbb{P}((u_1, \ldots u_L)) = \mathbb{P}((u_1, \ldots u_L, \texttt{<EOS>}))$$

is the probability that a random sequence in $\mathcal{U}^*$ has values $u_1, u_2, \ldots, u_L$ for the first L elements and then terminates, i.e., u_{L+1}=<EOS>.

On the other hand, if $u_1, u_2, \ldots, u_L \in \mathcal{U}$,

$$\mathbb{P}(u_1, \ldots u_L)$$

is the probability that a random sequence in $\mathcal{U}^*$ has values $u_1, u_2, \ldots, u_L$ for the first L elements (and none of them are <EOS>) but u_{L+1} but may or may not be <EOS>. In particular,

$$\begin{aligned}
\mathbb{P}(u_1, \ldots u_L) &= \mathbb{P}(u_1, \ldots u_L, u_{L+1} = \texttt{<EOS>}) + \mathbb{P}(u_1, \ldots u_L, u_{L+1} \neq \texttt{<EOS>}) \\
&= \mathbb{P}((u_1, \ldots u_L)) + \mathbb{P}(u_1, \ldots u_L, u_{L+1} \neq \texttt{<EOS>})
\end{aligned}$$

Conditional probabilities with sequences

With the chain rule (conditional probability), we have

$$
\begin{aligned}
\mathbb{P}((u_1, \ldots u_L)) &= \mathbb{P}((u_1, \ldots u_L, \texttt{<EOS>})) \\
&= \mathbb{P}(u_{L+1} = \texttt{<EOS>} \mid u_1, \ldots, u_L)\mathbb{P}(u_1, \ldots, u_L) \\
&= \mathbb{P}(u_{L+1} = \texttt{<EOS>} \mid u_1, \ldots, u_L)\mathbb{P}(u_L \mid u_1, \ldots, u_{L-1})\mathbb{P}(u_1, \ldots u_{L-1}) \\
&= \mathbb{P}(u_{L+1} = \texttt{<EOS>} \mid u_1, \ldots, u_L) \prod_{\ell=1}^{L} \mathbb{P}(u_\ell \mid u_1, \ldots, u_{\ell-1})
\end{aligned}
$$

where $\mathbb{P}(u_\ell | u_1, \ldots, u_{\ell-1})$ is the probability of u_ℓ conditioned on the past. (For $\ell = 1$, we mean $\mathbb{P}(u_1 | u_1, \ldots, u_0) = \mathbb{P}(u_1)$.) So, the probability of the entire sequence $(u_1, u_2, \ldots, u_L) = (u_1, u_2, \ldots, u_L, \text{<EOS>})$ is the product of the conditional probabilities.

To clarify, we have made no assumptions on the sequence probabilities. (We have not assumed that anything is Markov or that anything is independent.)

Cond. prob. with continuous sequences

If sequence elements u_t are continuous random variables, then we need density functions instead of discrete probability mass functions. However, calculations are essentially the same, so we do not repeat it.

In NLP, vocabulary is finite, so consider seqs with discrete elements.

Some RL problems have continuous states and rewards.

For image patches (vision transformers), seq elements are (essentially) continuous.

Autoregressive (AR) modelling

An *autoregressive model* of a sequence learns to predict u_ℓ given the past ovservations $u_1, \dots, u_{\ell-1}$. Goal is to learn a model f_θ that approximates the full conditional distribution

$$f_\theta(u_\ell; u_1, \dots, u_{\ell-1}) \approx \mathbb{P}(u_\ell | u_1, \dots, u_{\ell-1})$$

(Etymology is ‘auto’ ≈ ’self’ and ‘regress’ ≈ ’fit’.)

Sequence likelihood with AR model

Given a trained autoregressive model $f_\theta(u_\ell; u_1, \ldots, u_{\ell-1}) \approx \mathbb{P}(u_\ell | u_1, \ldots, u_{\ell-1})$, we can (approximately) compute the likelihood of a sequence $(u_1, \ldots, u_L)$ with

$$\begin{aligned}
\mathbb{P}((u_1, \ldots u_L)) &= \mathbb{P}(u_{L+1} = \texttt{<EOS>} \mid u_1, \ldots, u_L) \prod_{\ell=1}^{L} \mathbb{P}(u_\ell \mid u_1, \ldots, u_{\ell-1}) \\
&\approx f_\theta(u_{L+1} = \texttt{<EOS>}; u_1, \ldots, u_L) \prod_{\ell=1}^{L} f_\theta(u_\ell; u_1, \ldots, u_{\ell-1})
\end{aligned}$$

Sequence generation with AR model

Given a trained autoregressive model $f_\theta(u_t; u_1, \ldots, u_{\ell-1}) \approx \mathbb{P}(u_\ell | u_1, \ldots, u_{\ell-1})$, and an un-terminated sequence $u_1, \ldots, u_{\ell-1}$ (if $\ell = 1$, then start generation from nothing) we can generate $(u_1, \ldots, u_{\ell-1}, u_\ell, \ldots, u_L) \sim \mathbb{P}(u_t, \ldots, u_L, u_{L+1}=\texttt{<EOS>} | u_1, \ldots, u_{\ell-1})$ by sampling

$$u_{\ell'} \sim f_\theta(\cdot; u_1, \ldots, u_{\ell'-1}), \qquad \ell' = \ell, \ldots \text{ until } u_{\ell'} = \texttt{<EOS>}$$

which is justified by

$$\begin{aligned}\mathbb{P}(u_\ell, \ldots u_L, u_{L+1} = \texttt{<EOS>} \mid u_1, \ldots, u_{\ell-1}) &= \mathbb{P}(u_{L+1} = \texttt{<EOS>} \mid u_1, \ldots, u_L) \prod_{\ell'=\ell}^{L} \mathbb{P}(u_{\ell'} \mid u_1, \ldots, u_{\ell'-1}) \\ &\approx f_\theta(u_{L+1} = \texttt{<EOS>}; u_1, \ldots, u_L) \prod_{\ell'=\ell}^{L} f_\theta(u_{\ell'}; u_1, \ldots, u_{\ell'-1})\end{aligned}$$

Modern NLP and sequence processing

Modern NLP solves various tasks, especially language modelling, with deep neural networks.

We need a general approach to process sequences (variable-length data) as inputs and outputs. We start with RNNs and then move on to transformers.

Why still learn RNNs? Although transformers have been replacing RNNs and CNNs in recent years, RNNs and CNNs are not yet obsolete. Also, much of the architecture design of transformers are inspired by practices inherited from the RNN era. One still needs to know RNNs to fully understand modern NLP.

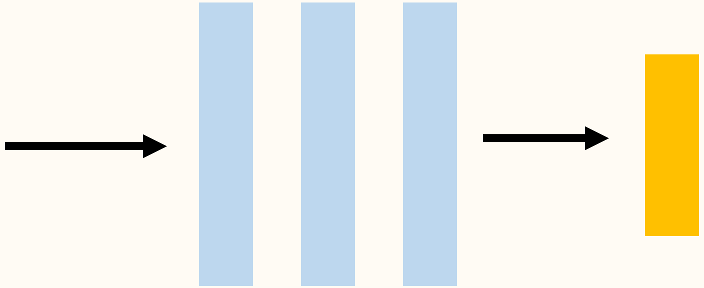
Learning with variable-sized inputs

In image classification, the input $X \in \mathbb{R}^{3\times n\times m}$ is of fixed size and processed by a deep CNN.
We now want to process variable-sized input $X \in \mathcal{C}^*$ with a neural network.

Simple idea: Zero-pad up to length of longest sequence.

$$\tau(X_1) = (u_{1,1}, u_{1,2}, u_{1,3}, u_{1,4})$$
$$\tau(X_2) = (u_{2,1}, u_{2,2}, u_{2,3})$$
$$\tau(X_3) = (u_{3,1}, u_{3,2}, u_{3,3}, u_{3,4}, u_{3,5})$$

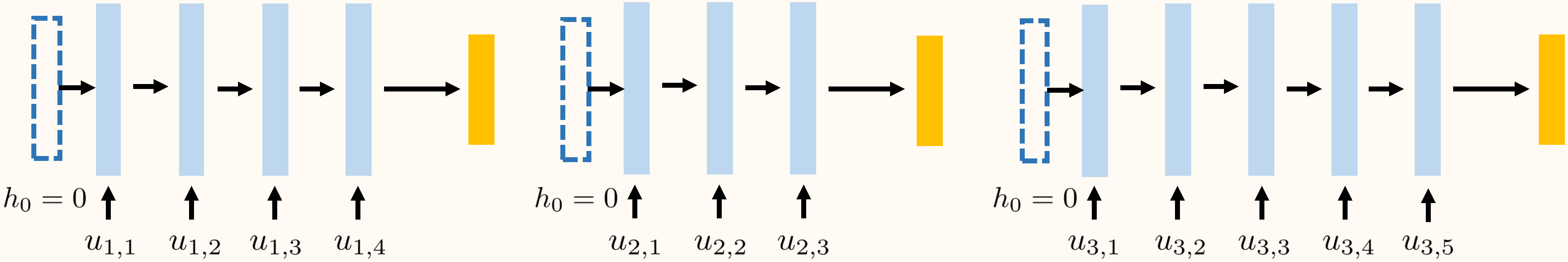
$$(u_{i,1}, u_{i,2}, u_{i,3}, 0, 0)$$



- (+) This can work as a quick and temporary solution.
- (–) Does not scale well for long sequences if fully-connected layer is used.
- (–) Maximum length must be specified.

Process one input per layer

$$\tau(X_1) = (u_{1,1}, u_{1,2}, u_{1,3}, u_{1,4})$$
$$\tau(X_2) = (u_{2,1}, u_{2,2}, u_{2,3})$$
$$\tau(X_3) = (u_{3,1}, u_{3,2}, u_{3,3}, u_{3,4}, u_{3,5})$$



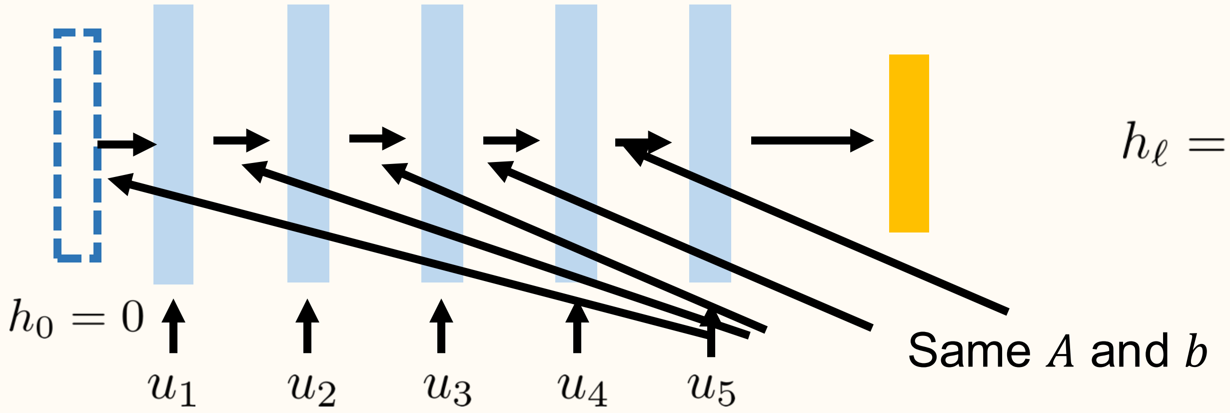
Idea: Process one input per layer

$$h_\ell = \sigma\left(A_\ell \begin{bmatrix} h_{\ell-1} \\ u_\ell \end{bmatrix} + b_\ell\right)$$

- (+) Shorter sequences require fewer layers to evaluate.
- (+) Each layer is much smaller than a giant layer one would need to process the whole sequence at once.
- (–) Total number of weights and biases increase with maximum sequence length.
- (–) Exploding/vanishing gradients.

Weight sharing

Idea: What if the parameters are the same (use weight sharing) for all layers?



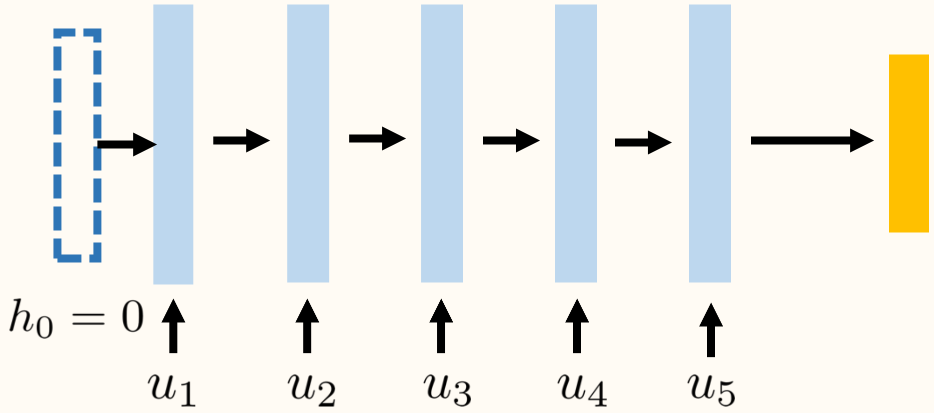
$$h_\ell = \sigma\left(A \begin{bmatrix} h_{\ell-1} \\ u_\ell \end{bmatrix} + b \right)$$

- (+) Can process an arbitrary number of inputs.
- (–) Exploding/vanishing gradients.

This is called a *recurrent neural network* (RNN).

Recurrent neural networks (RNN)

More generally, an RNN has the form



$$
\begin{aligned}
h_0 &= 0 \\
h_\ell &= q_{\tilde{\theta}}(h_{\ell-1}, u_\ell), \qquad \ell = 1, \ldots, L \\
f_\theta(X) &= A h_L + b \\
\theta &= (A, b, \tilde{\theta})
\end{aligned}
$$

where $\tilde{\theta}$, A, and b are the trainable parameters.
The $q_{\tilde{\theta}}$ is called the *recurrent function*.

The exploding/vanishing gradient problem still remains.
RNNs work only if $q_{\tilde{\theta}}$ is chosen to mitigate this problem.

J. L. Elman, Finding structure in time, *Cognitive Science*, 1990.

RNNs are extremely deep networks

Seq. length of 100s or 1000s is common.

$$\frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial \tilde{\theta}} = \sum_{\ell=1}^{L} \frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial h_\ell} \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{\ell-1}, u_\ell)$$

$$\frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial h_\ell} = \frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial h_L} \frac{\partial h_L}{\partial h_{L-1}} \frac{\partial h_{L-1}}{\partial h_{L-2}} \cdots \frac{\partial h_{\ell+1}}{\partial h_\ell}$$

Multiplying many numbers is unstable:

- If most of the numbers > 1, we get ∞ ("Exploding gradients". Can fix with gradient clipping.)
- If most of the numbers < 1, we get 0 ("Vanishing gradients". Bigger problem.)

Reasonably-sized product if numbers are all close to 1.

For matrices, a similar reasoning holds with eigenvalues or singular values.

Y. Bengio, P. Simard, and P. Frasconi, Learning long-term dependencies with gradient descent is difficult, *IEEE Transactions on Neural Networks*, 1994.

Exploding gradients and gradient clipping

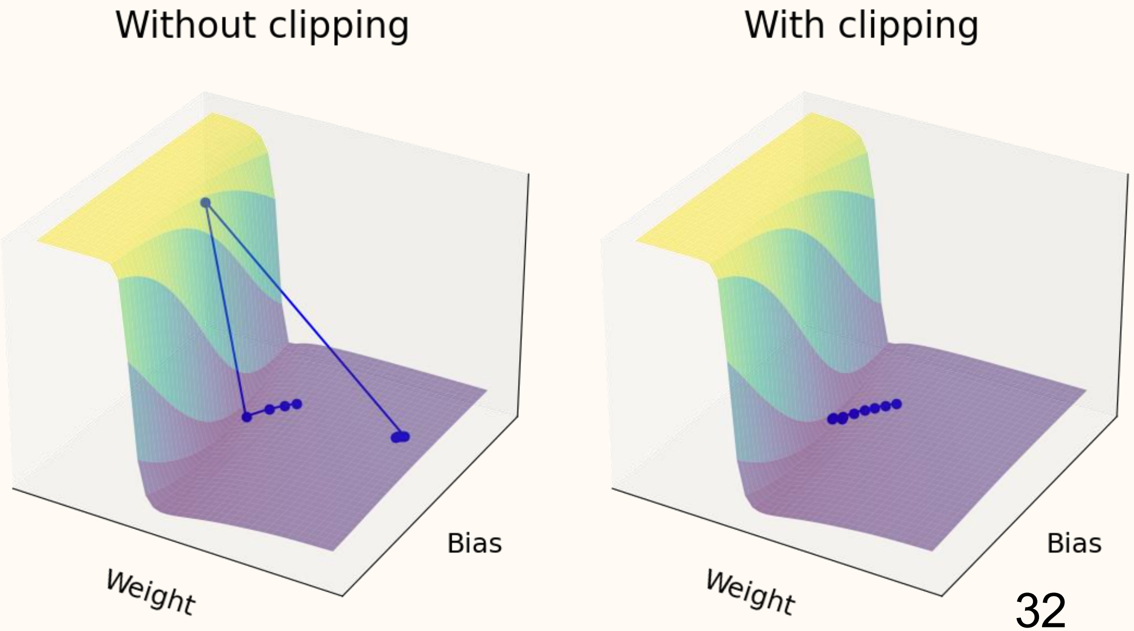
The *exploding gradient* problem occurs when the gradient magnitude is very large.

Exploding gradients imply the output is very sensitive to small changes of the parameters in a certain direction. Sometimes, such gradients are unworkable and the neural network architecture must be changed.

Sometimes, however, the *direction* of the gradient is fine. If so, one can use *gradient clipping* and use the clipped gradient in the optimization.

Gradient clipping with threshold value v:

$$g \leftarrow \max\left(1, \frac{v}{\|g\|}\right) g = \begin{cases} g & \text{if } \|g\| \leq v \\ \frac{v}{\|g\|} g & \text{otherwise} \end{cases}$$



Vanishing gradients

The *vanishing gradient* problem occurs when the magnitude of a gradient is very small.

Intuitively, vanishing gradients means the gradient signal does not reach the earlier layers. In an RNN, for example, $\partial \mathcal{L}/\partial h_L$ may not be small but

$$\frac{\partial \mathcal{L}}{\partial h_\ell} = \frac{\partial \mathcal{L}}{\partial h_L} \frac{\partial h_L}{\partial h_{L-1}} \frac{\partial h_{L-1}}{\partial h_{L-2}} \cdots \frac{\partial h_{\ell+1}}{\partial h_\ell}$$

can be small.

This means changes in h_ℓ do not affect the output $\mathcal{L}$. Since $\frac{\partial \mathcal{L}}{\partial u_\ell} = \frac{\partial \mathcal{L}}{\partial h_\ell} \frac{\partial h_\ell}{\partial u_\ell}$, this further implies that (small) changes in u_ℓ do not affect $\mathcal{L}$. We can intuitively understand this as the RNN not utilizing information of u_ℓ, i.e., RNN does not remember[#] u_ℓ at step L. Moreover, gradient signal $\partial \mathcal{L}/\partial h_L$ is lost by the time backprop reaches time ℓ, and the model can't learn to how to utilize u_ℓ. (The model forgets u_ℓ by step L, and gradient updates can't fix this.)

[#]This argument is not precisely correct since large changes in u_ℓ may affect $\mathcal{L}$.

Promoting better gradient flow

Consider

$$\frac{\partial \mathcal{L}}{\partial h_\ell} = \frac{\partial \mathcal{L}}{\partial h_{\ell+1}} \frac{\partial h_{\ell+1}}{\partial h_\ell}$$

If the Jacobian is close to identity, i.e., $\frac{\partial h_{\ell+1}}{\partial h_\ell} = \frac{\partial q_{\tilde{\theta}}}{\partial h}(h_\ell, u_{\ell+1}) \approx I \in \mathbb{R}^{n \times n}$ then we say the gradient *flows* through the layer $h_{\ell+1}$ well.

If $\frac{\partial h_{\ell+1}}{\partial h_\ell} \approx 0$, then $\frac{\partial \mathcal{L}}{\partial h_\ell} \approx 0$ and we say the gradient does not flow well through the layer $h_{\ell+1}$ well since the information contained in $\frac{\partial \mathcal{L}}{\partial h_{\ell+1}}$ is lost.

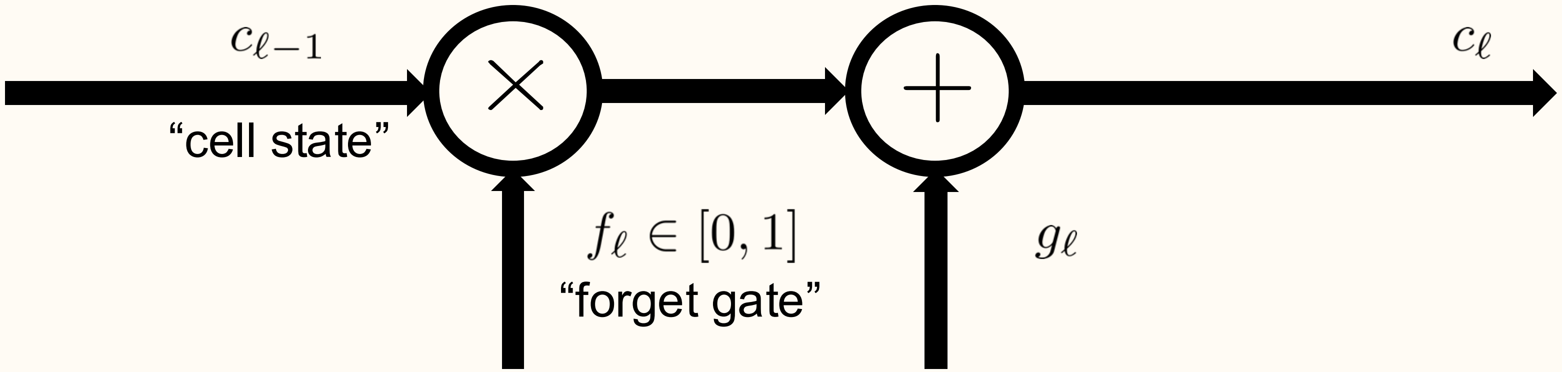
Promoting better gradient flow

So then, do we always want good gradient flow? Do we always want $\frac{\partial h_{\ell+1}}{\partial h_\ell} \approx I$?

No. We want $\frac{\partial h_{\ell+1}}{\partial h_\ell} \approx I$ when we want to *remember* information.

We want $\frac{\partial h_{\ell+1}}{\partial h_\ell} \approx 0$ when we want to *forget*.

Solution) Design a "neural circuit" that explicitly controls when to remember information and when to forget information.



$$c_\ell = c_{\ell-1} \odot f_\ell + g_\ell$$
$$\frac{dc_{\ell,i}}{dc_{\ell-1,i}} = f_{\ell,i} \in [0, 1]$$

LSTM cells

Long short-term memory (LSTM) cells has an intricate and somewhat arbitrary structure. (Cf. GRU cell[#] which simplifies the LSTM structure.)

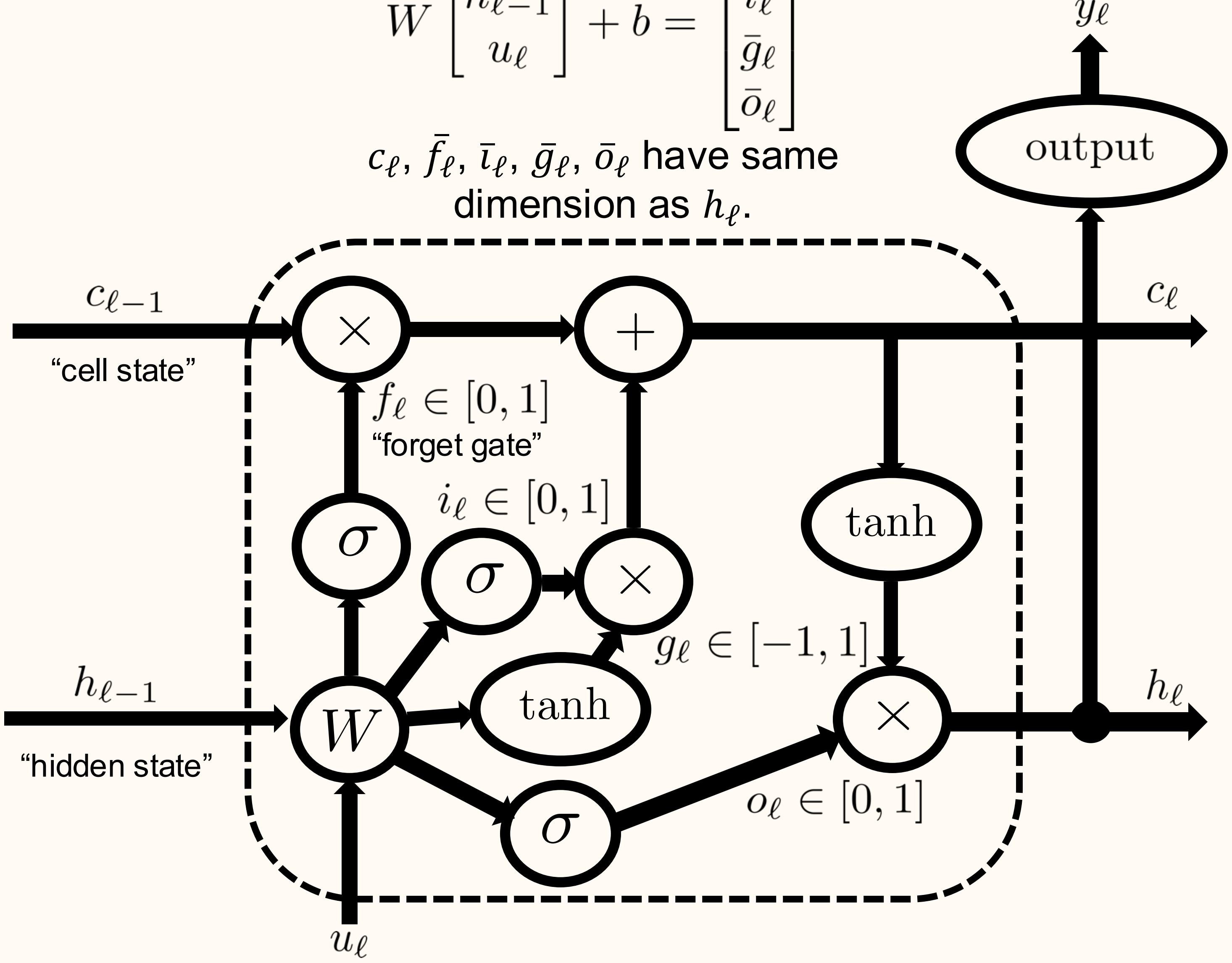
Works much better than a naïve RNN!

Cell state c_ℓ serves as memory.

(In retrospect, the cell state should be called the hidden state, as it is more similar to the hidden states of RNNs or hidden Markov models. However, this notation is now standard.)

$$W \begin{bmatrix} h_{\ell-1} \\ u_\ell \end{bmatrix} + b = \begin{bmatrix} \bar{f}_\ell \\ \bar{i}_\ell \\ \bar{g}_\ell \\ \bar{o}_\ell \end{bmatrix}$$

c_ℓ, $\bar{f}_\ell$, $\bar{\imath}_\ell$, $\bar{g}_\ell$, $\bar{o}_\ell$ have same dimension as h_ℓ.



S. Hochreiter and J. Schmidhuber, Long short-term memory, *Neural Computation*, 1997.
F. A. Gers, J. Schmidhuber, and F. Cummins, Learning to forget: continual prediction with LSTM, *Neural Computation*, 2000.
[#]J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, Empirical evaluation of gated recurrent neural networks on sequence modeling, *NeurIPS Workshop on Deep Learning and Representation Learning*, 2014.

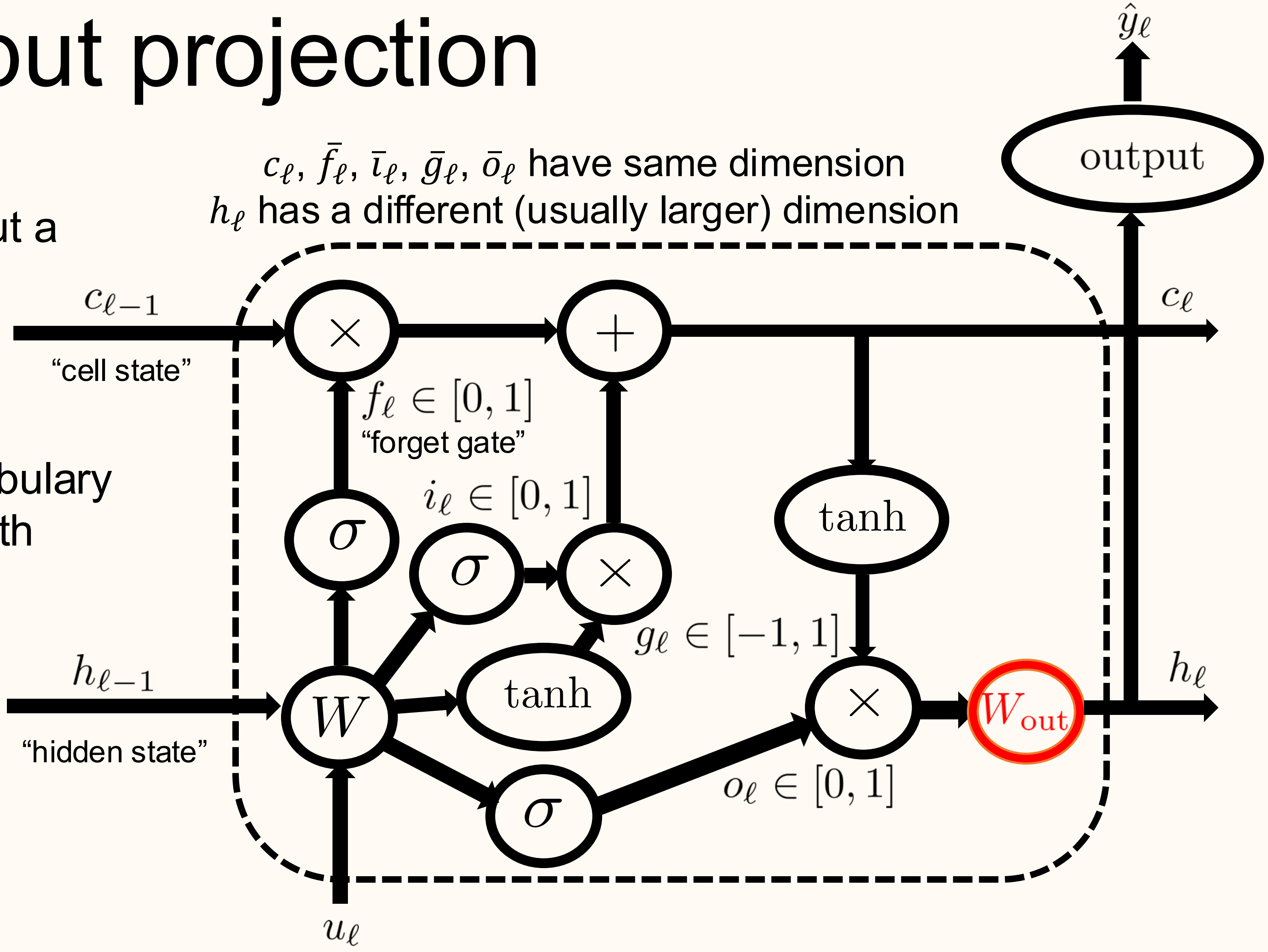
LSTM with output projection

$c_\ell, \bar{f}_\ell, \bar{\imath}_\ell, \bar{g}_\ell, \bar{o}_\ell$ have same dimension
h_ℓ has a different (usually larger) dimension

Sometimes, you want the LSTM to output a large hidden state while maintaining a reasonably-sized internal computation.

(In LMs, the output size can be the vocabulary size or the number of possible tokens with byte pair encoding, both are large.)

Solution) Output projection.



H. Sak, A. W. Senior, F. Beaufays, Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition, *arXiv*, 2014.

LSTM name meaning

To clarify, “long short-term memory” does not mean long-term & short-term memory.

Rather, it means that the cell state serves as a longer short-term memory. In contrast, a naïve RNN (that uses an MLP rather an LSTM cell as the recurrent function) would have a much shorter short-term memory.

A true long-term memory would correspond to some external storage, which an LSTM RNN doesn’t have.

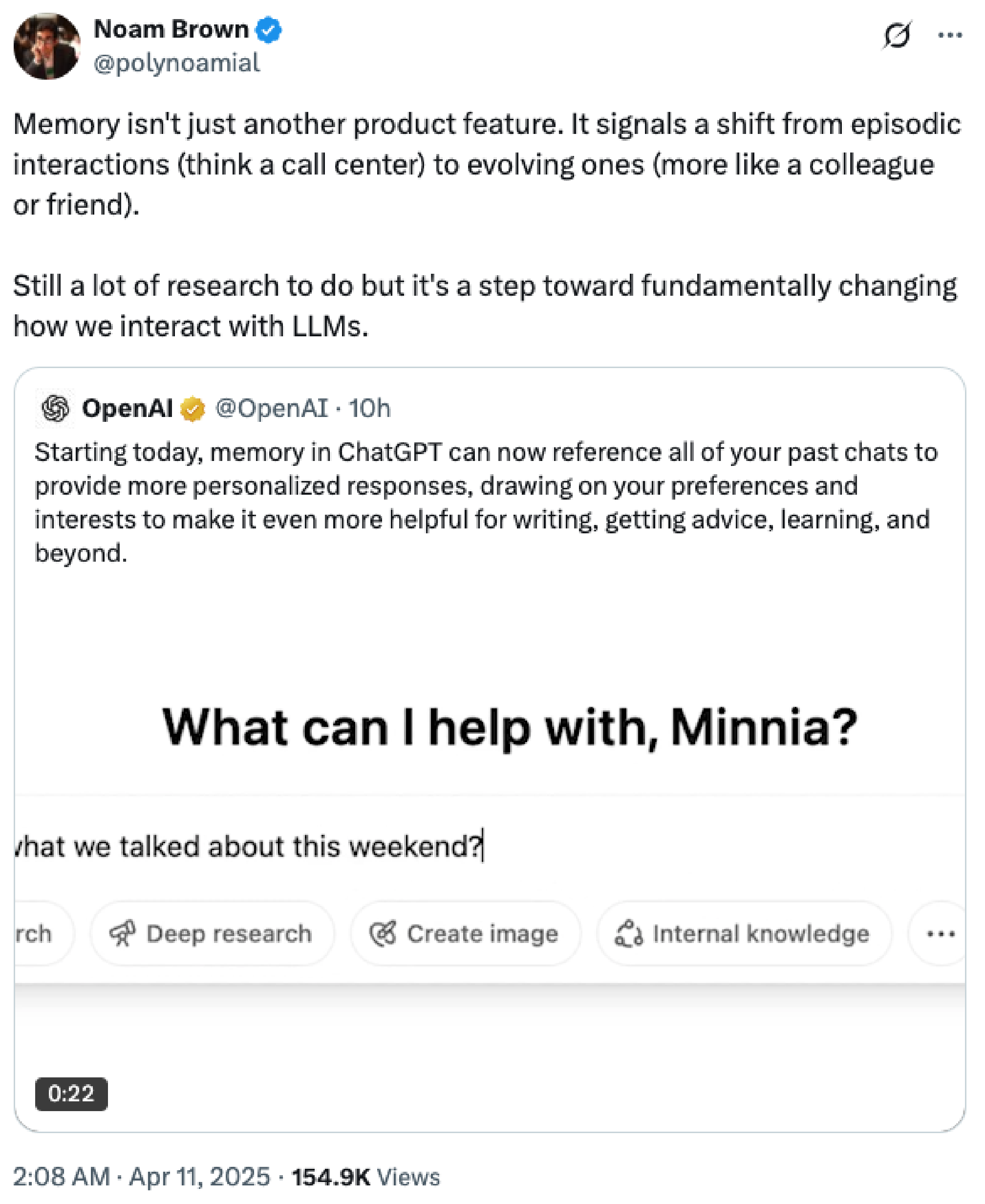
Proprietary LLMs now have long-term memory

Recently, OpenAI announced its new memory features in ChatGPT.

Other chatbot services will likely follow-up with similar features.

LLMs now have long-term memory, and this will profoundly affect how we interact with LLMs.

https://x.com/polynoamial/status/1910379351759347860



Aside: Exploding/vanishing gradient problem

The exploding/vanishing gradient problem arises in for many deep neural networks, not just RNNs.

The ResNet architectue, and more generally the use of residual connections is one approach to mitigate the exploding/vanishing gradient problem.

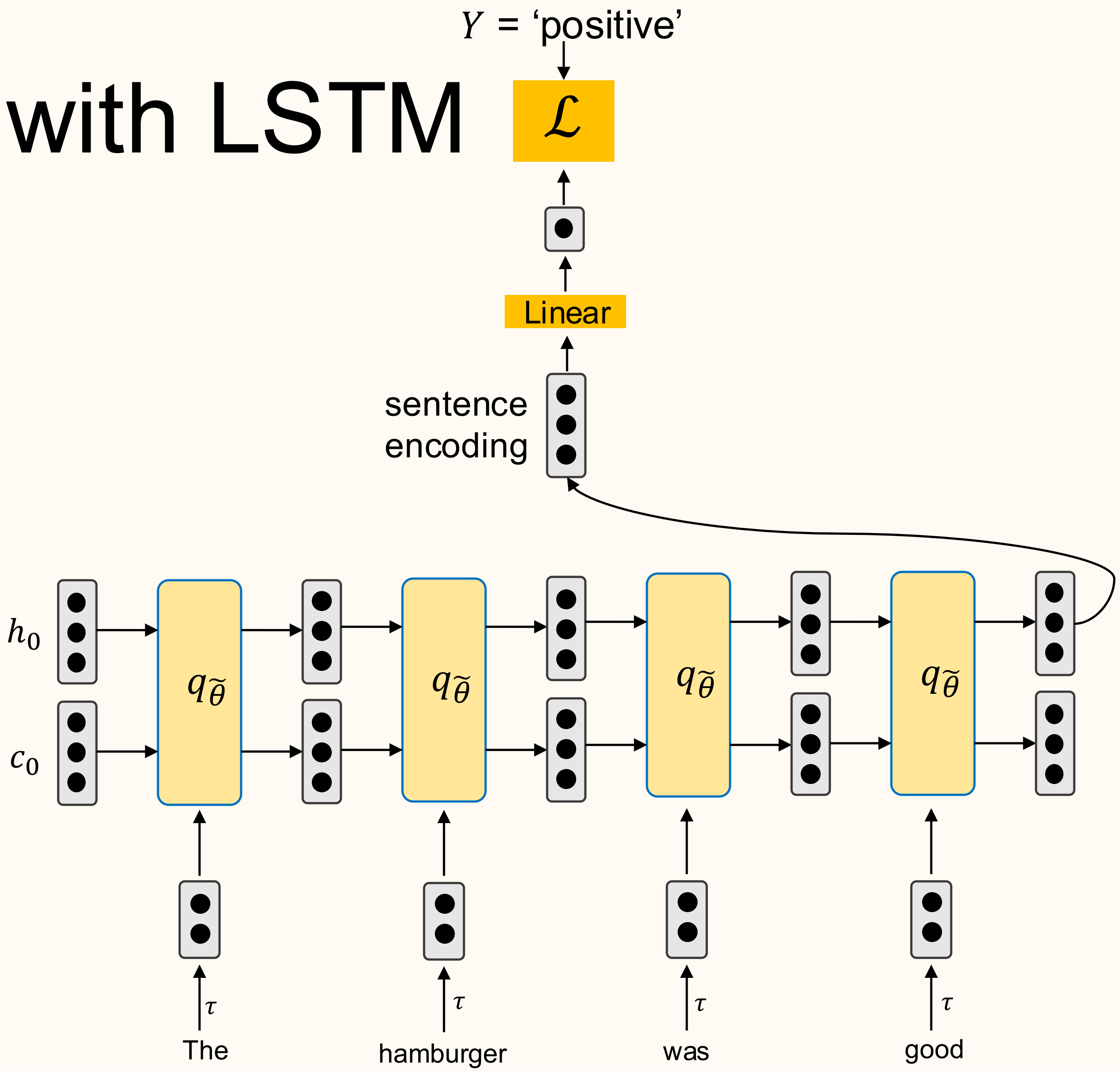
Another technique is the use of normalization layers such as batch norm.

RNNs can use batch norm[#], but it is not common.

[#]T. Cooijmans, N. Ballas, C. Laurent, C. Gülçehre, and A. Courville, Recurrent batch normalization, *ICLR*, 2017.

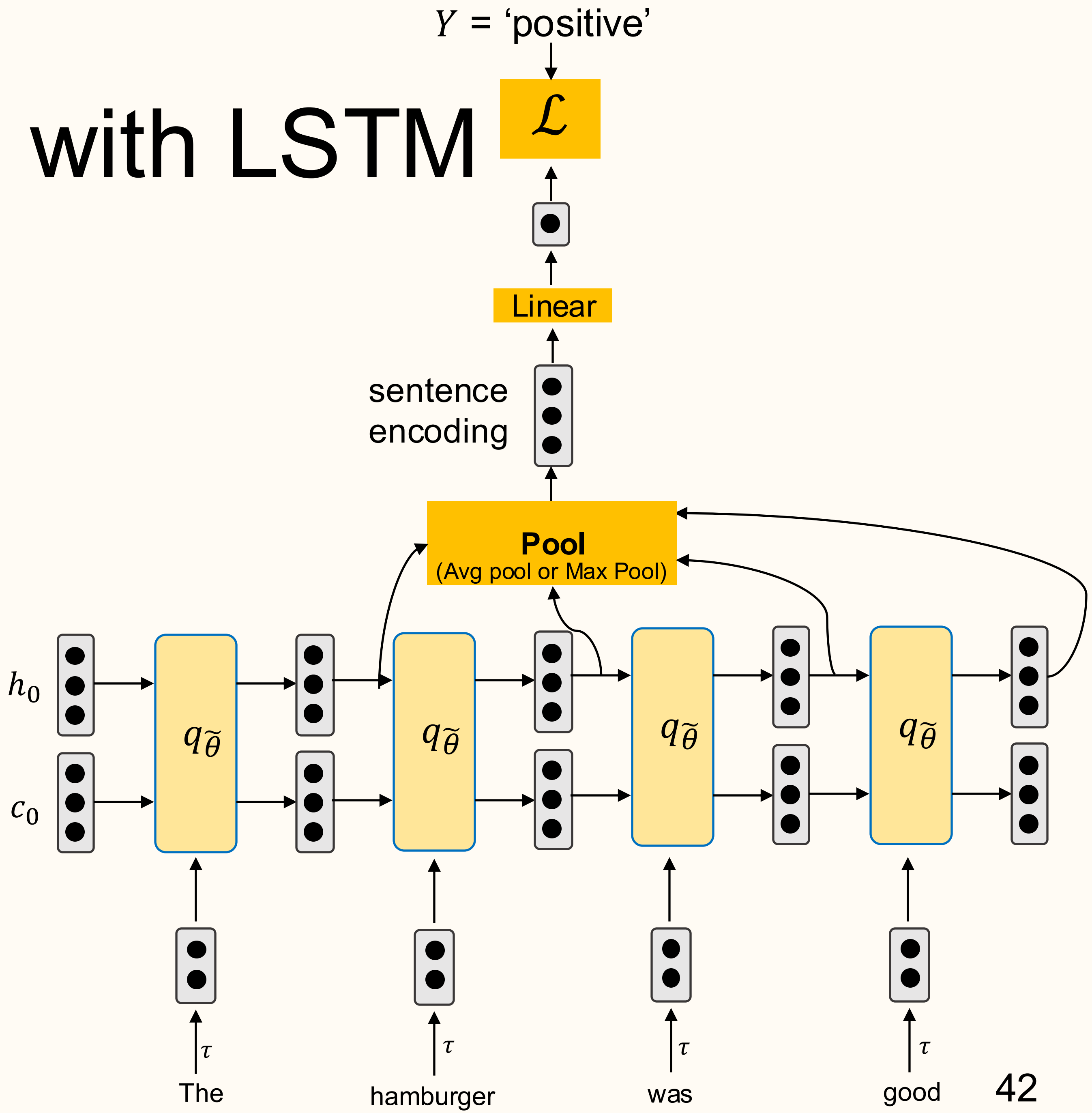
Sentiment analysis with LSTM

The output hidden state can be used for the single (non-sequence) output.



Sentiment analysis with LSTM

Pooling all of the hidden states often performs better than then using only the last one for learning a single (non-sequence) output.



Stacked RNN

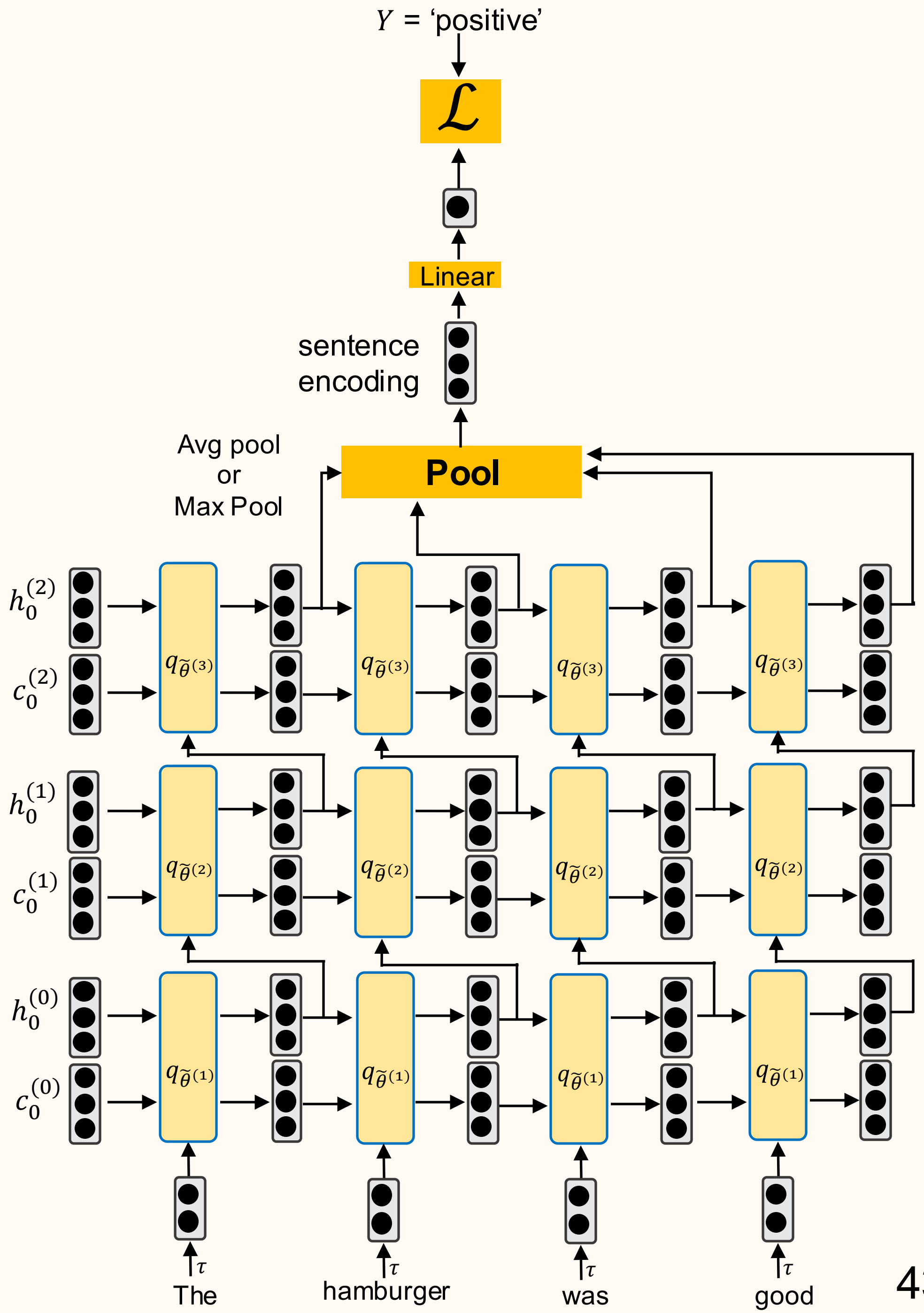
Stacked RNNs use more depth and can learn more complex representations.

Rule of thumb is to use 2–8 stacked LSTM layers.

- 2 layers is almost always better than 1.
- 3 layers is not always better than 2.

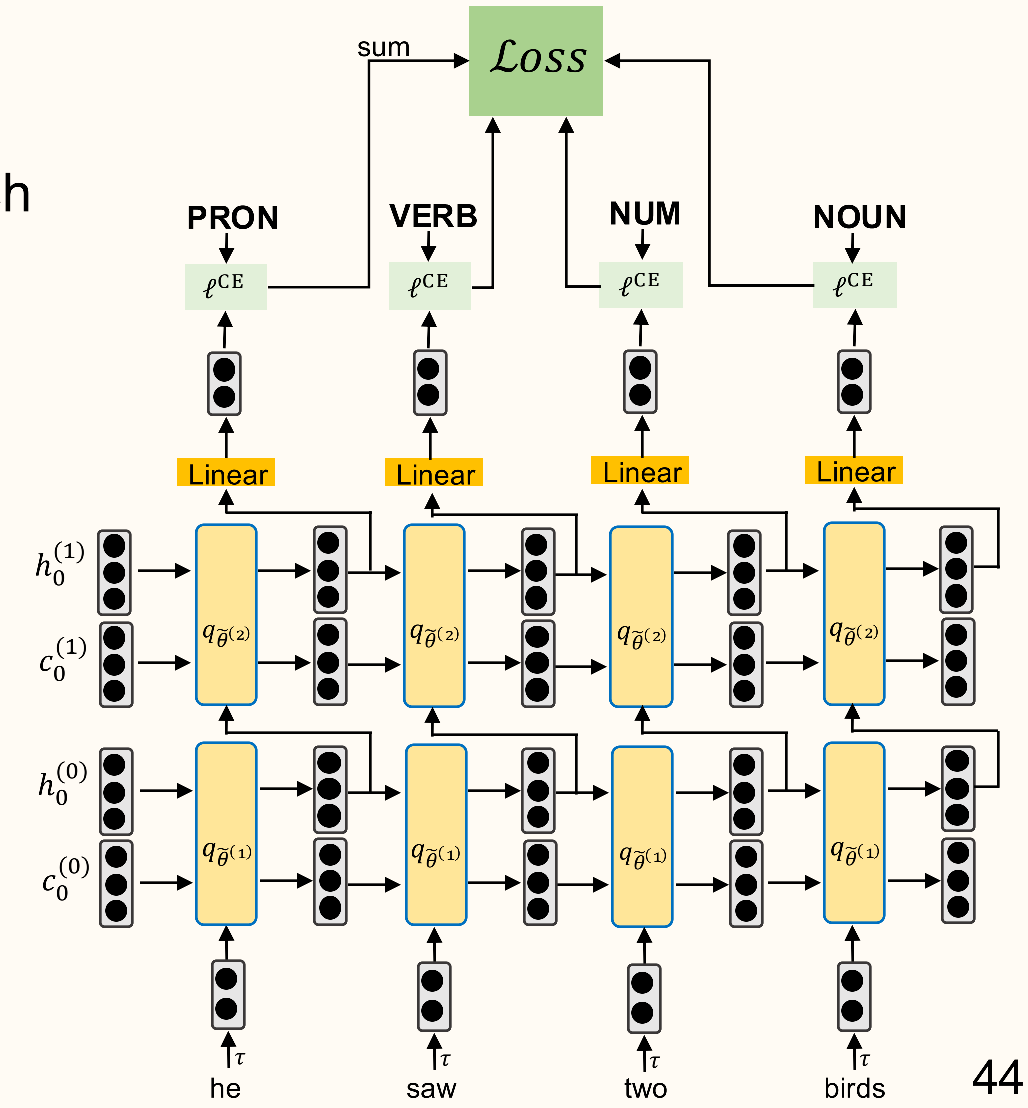
Each layer of an RNN transforms a sequence to a sequence:

$$\{u_\ell\}_{\ell=1}^{L} \mapsto \{h_\ell^{(1)}\}_{\ell=1}^{L} \mapsto \{h_\ell^{(2)}\}_{\ell=1}^{L} \mapsto \{h_\ell^{(3)}\}_{\ell=1}^{L}$$



Example task: Parts of speech tagging

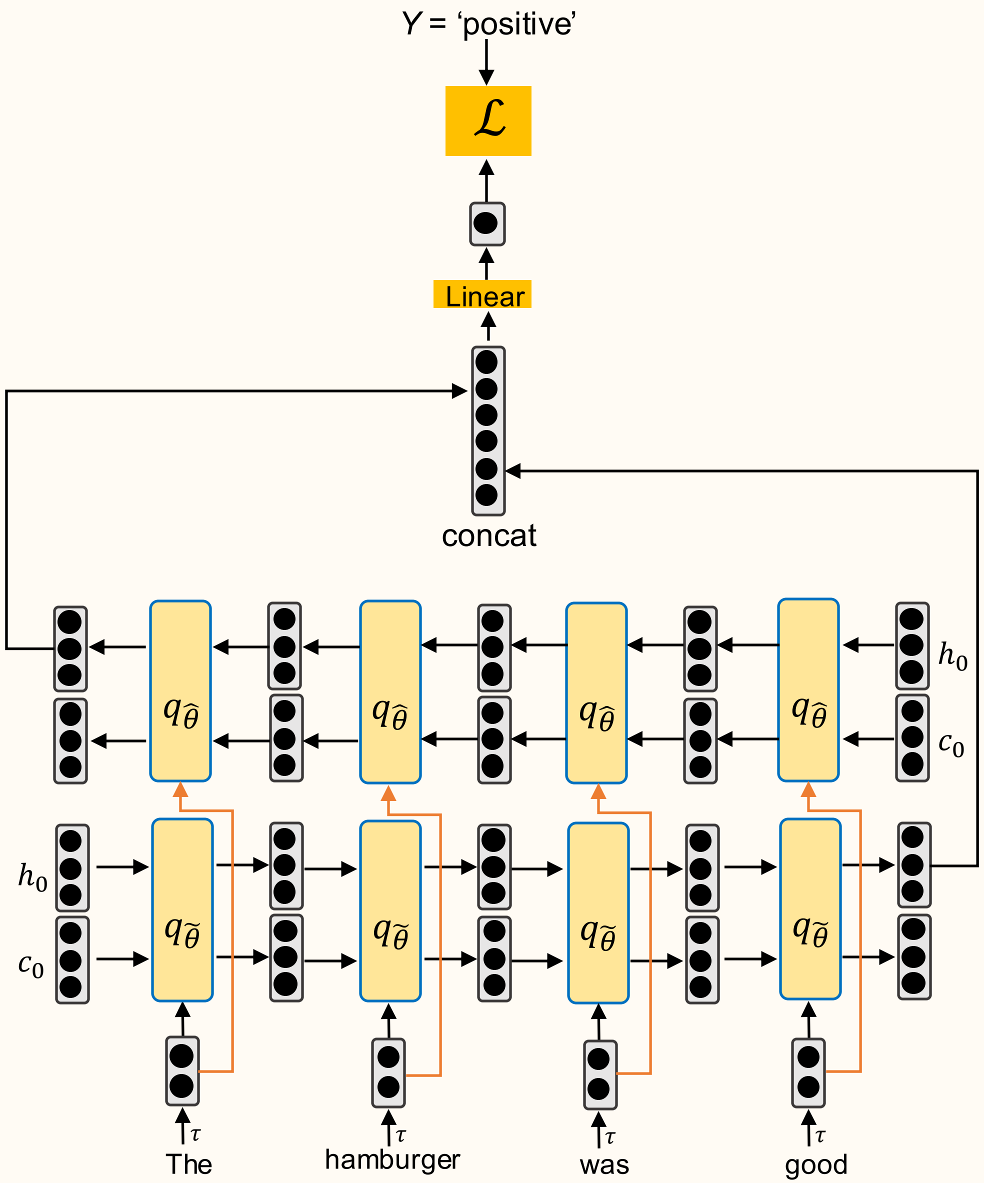
For some RNN tasks, the output is a sequence, and the total loss is the sum of the losses incurred at each sequence term.



Bidirectional RNN

(Unidirectional) RNNs process information forward in time. In language, however, it is common for later words to provide necessary context for understanding a previous word.

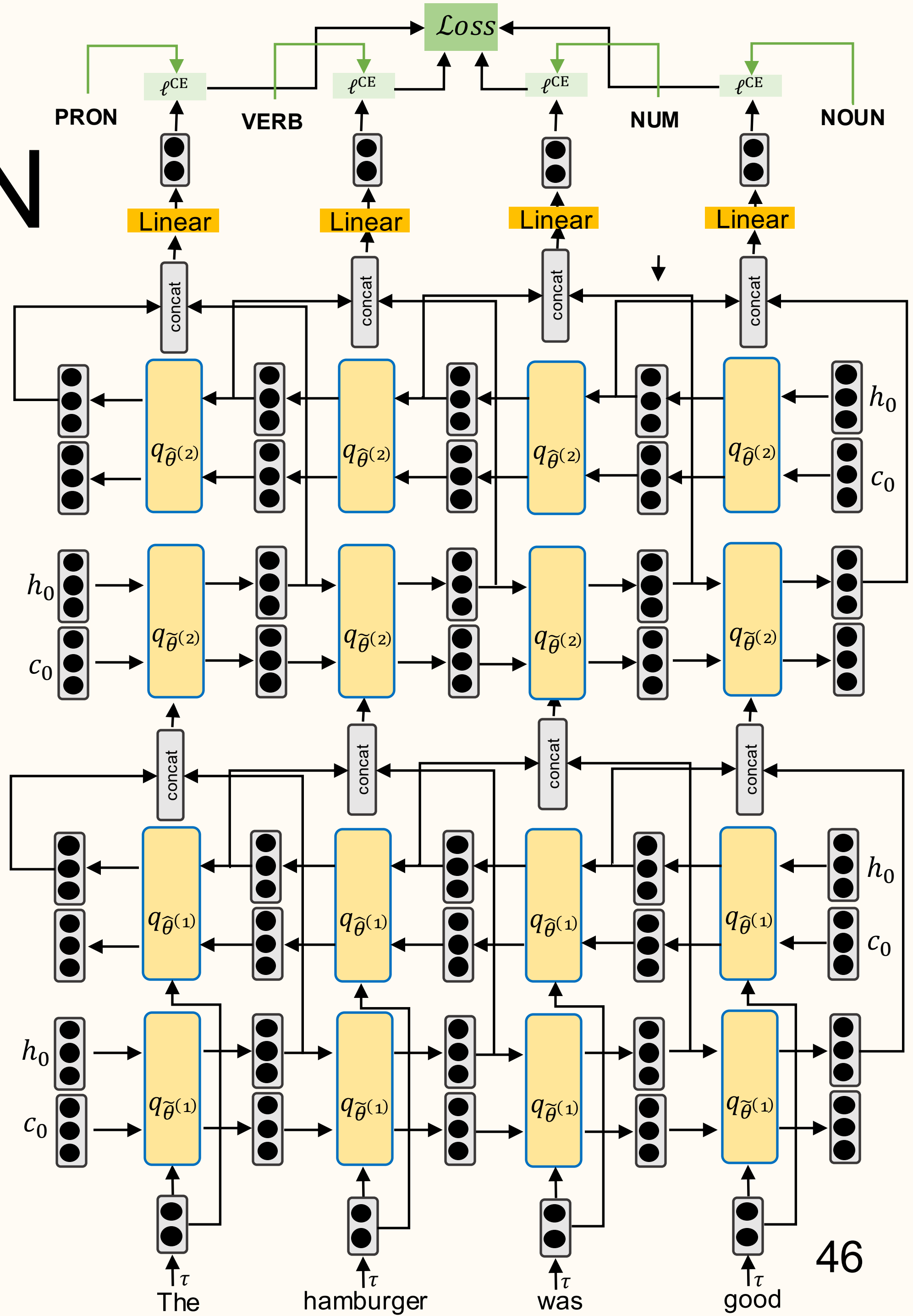
A *bidirectional RNN* combines forward and reverse directional RNNs to process a sequence without a single sense of direction.



M. Schuster and K. K. Paliwal , Bidirectional recurrent neural networks, *IEEE TSP*, 1997.

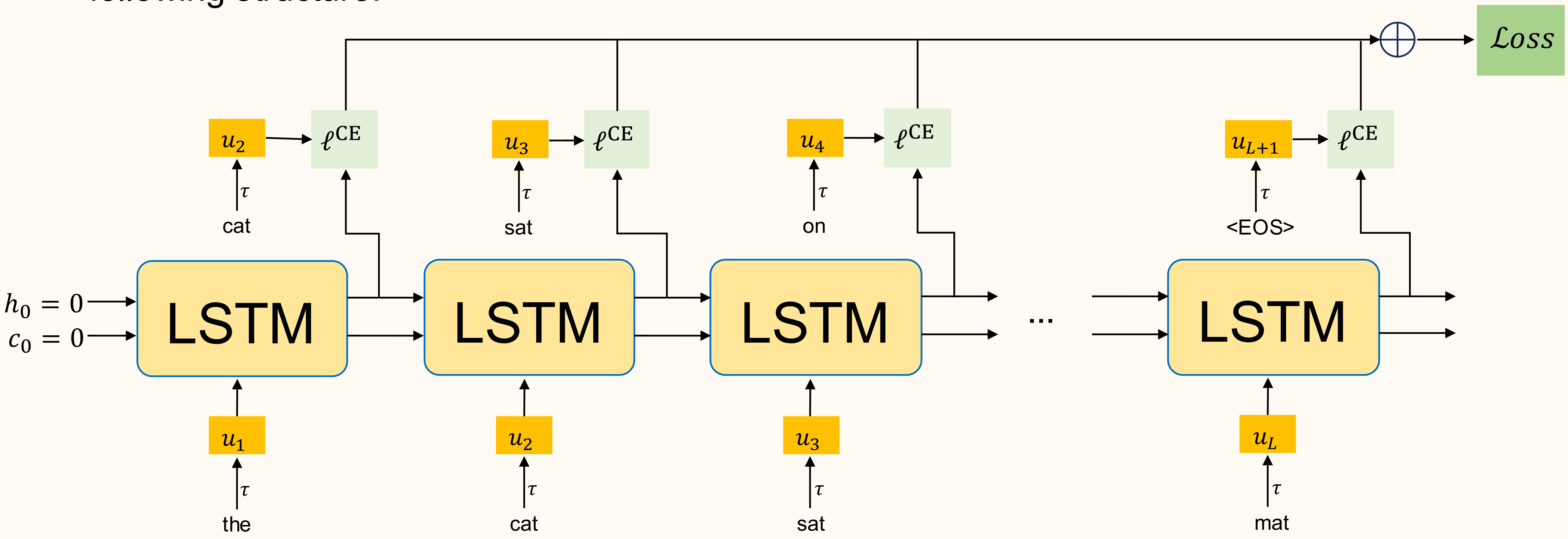
Stacked bidirectional RNN

Stacking and bidirectionality can be combined.



RNN-LM

The RNN language model (RNN-LM) is trained as an autoregressive model with the following structure.



T. Mikolov, S. Kombrink, L. Burget, J. H. Černocký, and S. Khudanpur, Extensions of recurrent neural network language model, *ICASSP*, 2011.

LM loss: Next token prediction

Let us interpret the loss

$$\mathcal{L} = \sum_{\ell=1}^{L} \ell^{\mathrm{CE}}(h_\ell, u_{\ell+1})$$

We are given a sequence of tokens $(u_1, \dots, u_L)$, which are one-hot vectors indicating the specific token. RNN predicts the next token

$$u_{\ell+1} \approx h_\ell = f_\theta(u_1, \dots, u_\ell)$$

for $\ell = 1, \dots, L$. (h_ℓ are logits, so $\approx$ means $\mathrm{softmax}(h_\ell) \approx u_{\ell+1}$, i.e. assigns high probability to the token $u_{\ell+1}$.) The (in)accuracy of the prediction is measured by the cross entropy loss

$$\ell^{\mathrm{CE}}(h_\ell, u_{\ell+1})$$

for $\ell = 1, \dots, L$. The sequence $(u_1, \dots, u_L)$ creates L next-token-prediction problems. Namely, given $u_1, \dots, u_\ell$, predict $u_{\ell+1}$ for $\ell = 1, \dots, L$. Our loss $\mathcal{L}$ is the sum of the losses on these L problems.

Sentence prob. vs. language generation

The training and inference of language models consider two different tasks, and it is instructive to view the distinction from an RL perspective.

Language models are trained on:

- Assigning probabilities/likelihoods to a given sentence via next-token-prediction.
- Next-token-prediction is analogous to imitation learning.
- Training done on real sentences (in distribution).

Language models (LM) are often used to:

- Generate likely (coherent) sentences.
- Generation is analogous to policies acting in an environment.
- Unless the LM is very good, partial generations will be unnatural, and the generation conditioned on the unnatural generation (out of distribution) will be worse. This is a consistent problem with acting on a policy trained with imitation learning.

Teacher forcing

Teacher forcing[#] (TF) is the practice of feeding ground-truth sequence values back into the RNN after each step, even though the RNN would not generate such a sequence, exactly like imitation learning (IL). This forces the RNN to stay in the training distribution.

However, once the model makes a mistake (generate one bad token) it does not know how to recover, since the model was trained only on good text through TF/IL.

Resolution 1: Start training with TF and transition into something else.[%] What this something else should be is not obvious. Can't use DAgger due to data constraints. (Why?)

Resolution 2: Just make the LM large and train on lots of data, including both good text and poorly written messy internet text. This is the solution of modern LLMs.

[#] R. J. Williams and D. Zipser, A learning algorithm for continually running fully recurrent neural networks, *Neural Computation*, 1989.
[%] S. Bengio, O. Vinyals, N. Jaitly, and N. Shazeer, Scheduled sampling for sequence prediction with recurrent neural networks, *NeurIPS*, 2015.

Trainable tokenizer

The tokenizer is the first contact between language and our algorithm.

Instead of using one-hot encodings with a fixed dictionary, it is better to have some trainable component in the tokenizer.

Currently, byte-pair encoding has become the standard choice, but we shall consider the historical context.

What does a word mean?

Denotational semantics: A word is the collection of the objects it describes.

This is the intuitive and straightforward view of the meaning of words, but it is not a very actionable definition.

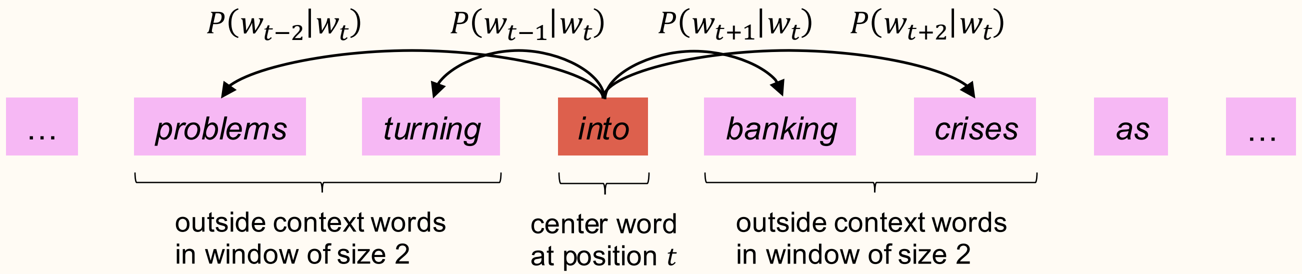
Distributional semantics: A word is characterized by the company it keeps.

A word's meaning is given by the words that frequently appear close-by. This is much more actionable in the realm of computational linguistics.

Word2vec

Train a model such that a word predicts a neighboring word.

“Context word” defined by a window of size m.
m is a hyperparameter that is tuned



We use two vectors per word:

- v_w when w is a center word
- u_w when w is a context word

Then, for a center word w_t and context word w_{t+k}:

$$\mathbb{P}(w_{t+k}|w_t) = \frac{\exp\left(u_{w_{t+k}} \cdot v_{w_t}\right)}{\sum_{w \in V} \exp\left(u_w \cdot v_{w_t}\right)}$$

T. Mikolov, K. Chen, G. Corrado, and J. Dean, Efficient estimation of word representations in vector space, *ICLR Workshop*, 2013.
T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, Distributed representations of words and phrases and their compositionality, *NeurIPS*, 2013.
NeurIPS 2023 Test of Time Award

Word2vec

Specifically, we minimize the loss function

$$\mathcal{L}(\theta) = \sum_{\ell=1}^{L} \sum_{\substack{-m \le k \le m \\ k \ne 0}} -\log p_\theta(w_{\ell+k} | w_k)$$

where $\theta = (v_1, \dots, v_n, u_1, \dots, u_n)$ is the trainable parameter.

Insight: Words w and $\tilde{w}$ are “similar” if $\mathbb{P}(\text{neighboring words}|w) \approx \mathbb{P}(\text{neighboring words}|\tilde{w})$ and this happens when $v_w \approx v_{\tilde{w}}$.

Word2vec: Trained word-level tokenizer

Goal of word2vec is a trained word-level tokenizer:
Given input $X = (w_1, \dots, w_L)$ chunked into words $w_1, \dots, w_L \in \mathcal{W}$, the trained tokenizer τ_θ outputs the corresponding v-vectors. This is a data-driven way to map similar words to similar vectors.

Using word2vec as inputs to RNNs significantly improves performance, compared to simple one-hot tokenizers.

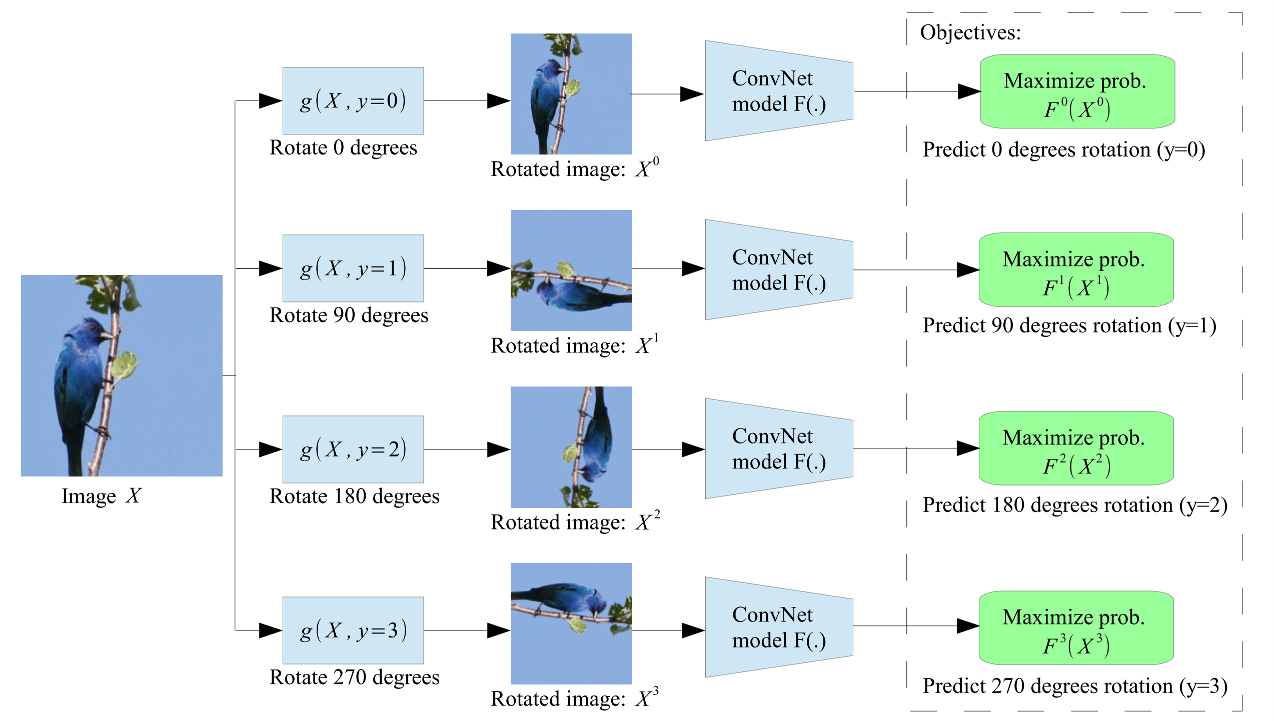
Downside: The word-level tokenizer $\tau_\theta(w_\ell)$ does not consider the context in which the word w_ℓ is used in (cf. polysemy).

T. Mikolov, K. Chen, G. Corrado, and J. Dean, Efficient estimation of word representations in vector space, *ICLR Workshop*, 2013.
T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, Distributed representations of words and phrases and their compositionality, *NeurIPS*, 2013.

Self-supervised learning

In self-supervised learning (SSL), models learn useful representations from unlabeled data by solving *pretext tasks* that automatically generate supervisory signals. The pretext task is not by itself useful, but it is related to the downstream task of interest. The pre-trained model is subsequently trained on downstream tasks, usually with a smaller labeled dataset.

The success of SSL is most dramatic in NLP, and Word2Vec was the first notable success case.



In computer vision, predicting random rotations of an image is one example of a pre-text task.

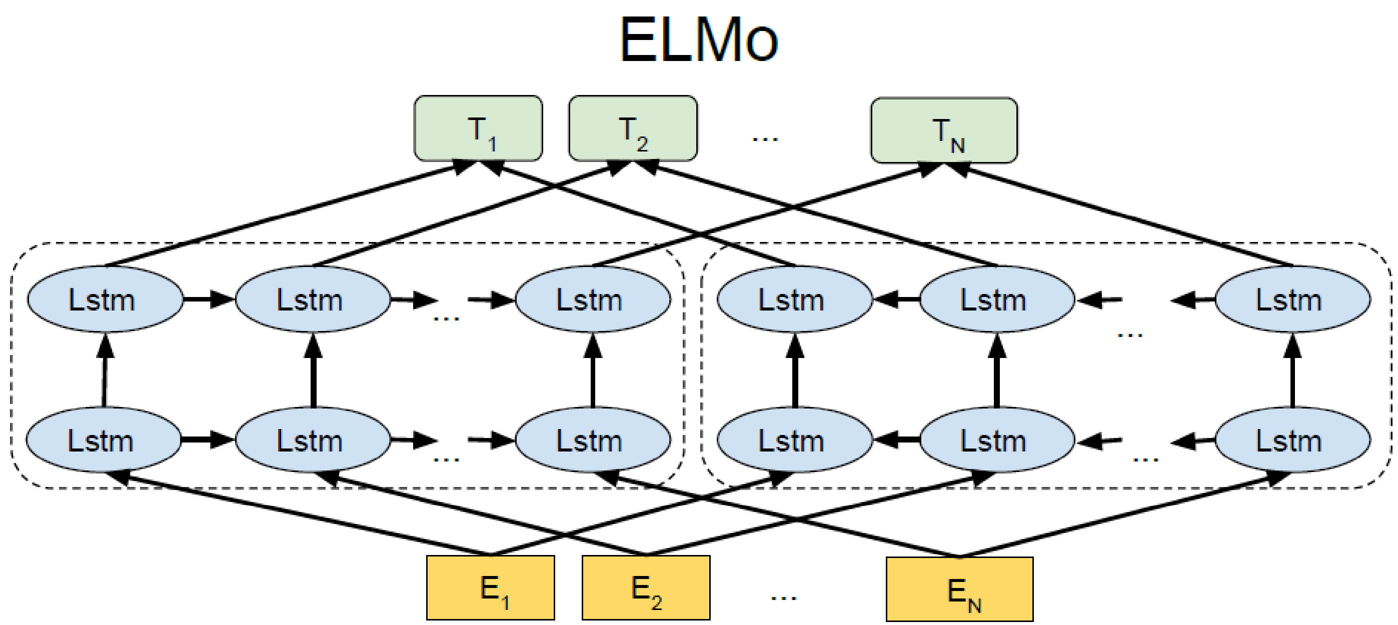
S. Gidaris, P. Singh, and N. Komodakis, Unsupervised representation learning by predicting image rotations, *ICLR*, 2018.

ELMo

Embeddings from Language Models (ELMo) is an in-context tokenizer. Produces word representations in the context of the entire sentence.

Uses bidirectional LSTM structure. The states of RNNs are hidden states, but they can also be considered tokernized values of the given words.

ELMo has its own tokenizer layer with trainable parameters (like a transformer's embedding layer), but we won't pay attention to it.



M. E. Peters, W. Ammar, C. Bhagavatula, and R. Power, Semi-supervised sequence tagging with bidirectional language models, *ACL*, 2017.
M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, Deep contextualized word representations, *NAACL*, 2018.

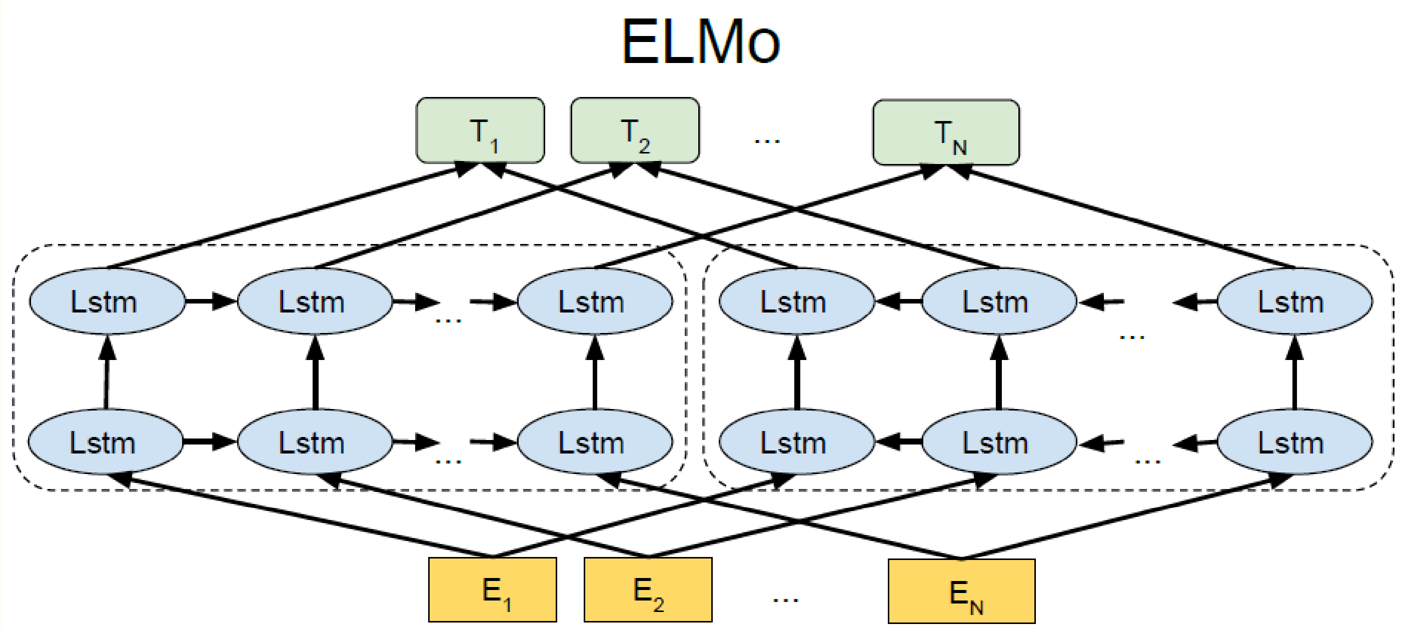
Bidirectional LM loss for pre-training

Pre-training uses the loss

$$\mathcal{L}(\overrightarrow{\theta_{\text{LSTM}}}, \overleftarrow{\theta_{\text{LSTM}}}, \theta_{\text{other}}) = \sum_{\ell=1}^{L} -\log f_{\overrightarrow{\theta_{\text{LSTM}}},\theta_{\text{other}}}(u_\ell \mid u_1, \ldots, u_{\ell-1}) + \sum_{\ell=1}^{L} -\log f_{\overleftarrow{\theta_{\text{LSTM}}},\theta_{\text{other}}}(u_\ell \mid u_{\ell+1}, \ldots, u_L)$$

where $\overrightarrow{\theta_{\text{LSTM}}}$ and $\overleftarrow{\theta_{\text{LSTM}}}$ are the parameters of the forward and backward LSTM cells and θ_{other} are the shared parameters for the input (tokenizer) and output (softmax) stages.

Training data is unlabeled text.
(Labels are the ground truth words.)



Non-causal language model

Causal language models learn

$$f_\theta(u_\ell; u_1, \ldots, u_{\ell-1}) \approx \mathbb{P}(u_\ell | u_1, \ldots, u_{\ell-1})$$

i.e., the causal LM learns to predict the next token left-to-right.

ELMo and BERT are non-causal language models. (Half of ELMo is a causal language model, but that is not the point.) ELMo and BERT can understand language and solve many NLP tasks, but it cannot generate text.

GPT is a causal language model, and it can generate text.

ELMo fine-tuning

Given a prior NLP method (which can be very specialized and tailored to the specific task) that takes in $\{x_\ell\}_{\ell=1}^{L}$, replace the input $\{x_\ell\}_{\ell=1}^{L}$ with $\{\tilde{x}_\ell\}_{\ell=1}^{L}$, where

$$\tilde{x}_\ell = \left[x_\ell; \sum_{k=0}^{K} s_k^{\text{task}} \overrightarrow{h_{k,\ell}}; \sum_{k=0}^{K} s_k^{\text{task}} \overleftarrow{h_{k,\ell}}\right]$$

where K is the depth of the LSTM RNN, $k = 0$ corresponds to the tokenization layer, and s_k^{task} are the task-specific trainable parameters. (The sum is over the LSTM depth.)

Then, train the entire pipeline, including the ELMo weights, $\{s_k^{\text{task}}\}_{k=0}^{K}$, and the weights of the NLP method on labeled task-specific data.

Results

ELMo achieved state-of-the-art performance on a wide range of NLP tasks.

- Question answering
- Textual entailment (determining whether a “hypothesis” is true, given a “premise”)
- Semantic role labeling (Answers “Who did what to whom”)
- Coreference resolution (clustering mentions in text that refer to the same underlying real-world entities)
- Named entity extraction (finding four types of named entities (PER, LOC, ORG, MISC) in news articles)
- Sentiment analysis (whether paragraph is positive or negative)

Discussion of ELMo

Although the idea of self-supervised learning through large-scale pre-training and fine-tuning was not new (Word2Vec is a predecessor and [Dai and Le 2015] proposed an idea like ELMo) ELMo executed it very well and advanced the state of the art substantially. ELMo make it clear that self-supervised pre-training is an indispensable paradigm.

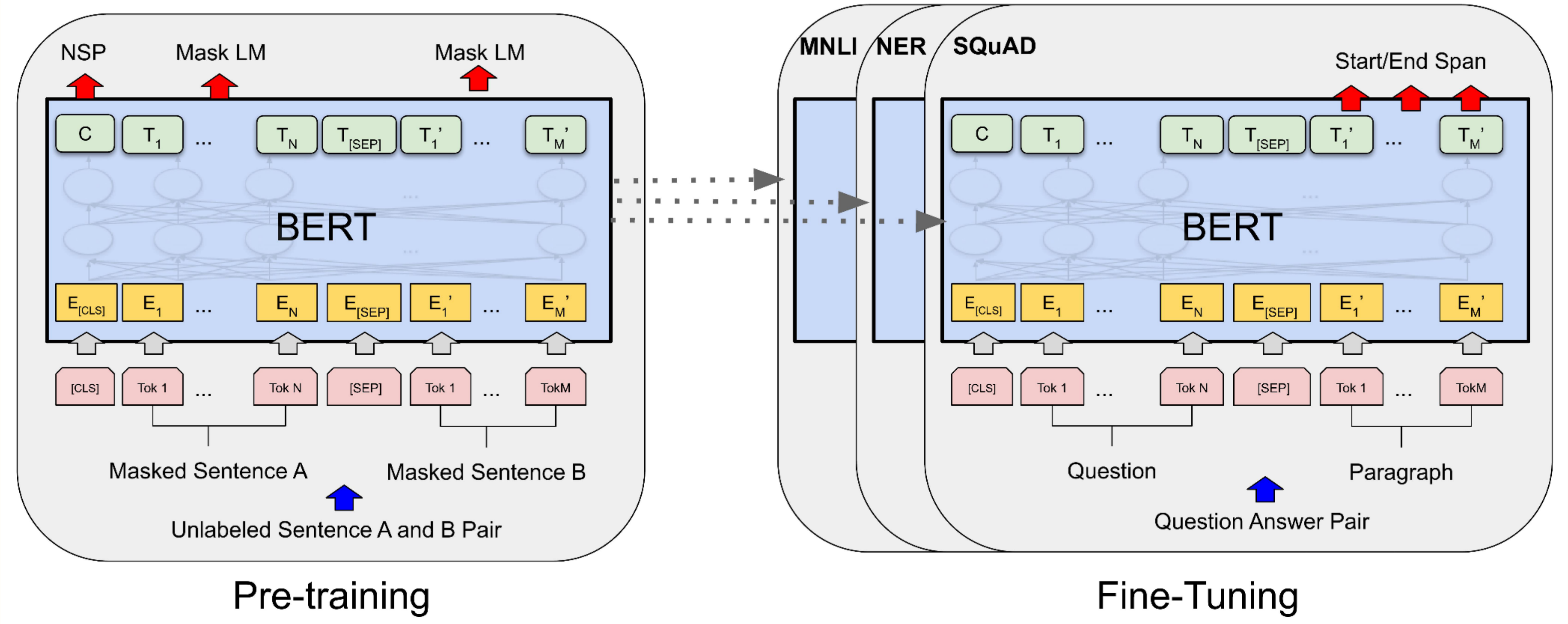
However, LSTM RNN is not the best architecture. The left- and right-directional RNNs only process information unidirectionally. What if the model needs to examine the entire sentence to make inference? Also, RNNs are fundamentally computationally inefficient.

The overall approach is still not universal; each task needs a tailored method and ELMo only served to provide better tokenization.

A. M. Dai and Q. V. Le, Semi-supervised sequence learning, *NeurIPS*, 2015.

BERT

Bidirectional Encoder Representations from Transformers (BERT) (i) replaces the LSTMs of ELMo with (encoder-only) transformer layers and (ii) proposed a more universal approach. BERT set a new state-of-the-art on almost every benchmark.



J. Devlin, M. Chang, K. Lee, and K. Toutanova, BERT: Pre-training of deep bidirectional transformers for language understanding, *NAACL*, 2019.

Transformers

Transformer architectures are sequence-to-sequence models. They “transform” a sequence to another sequence in each layer.

There are 3 types of transformers, listed in order of complexity.

- Encoder-only (BERT)
- Decoder-only (GPT)
- Encoder-decoder (Original transformer of Vaswani et al. 2017)

We first discuss the encoder-only transformer.

Encoder-only transformer

The transformer architecture relies on the following components

- Token embeddings
- Multi-head self-attention
- Residual connections
- Layer normalization
- Position-wise FFN
- Positional encodings

Token embeddings

Let τ be a tokenizer such that

$$\tau(X) = (u_1, u_2, \dots, u_L)$$

where we can think of $u_i = k \in \{1, \dots, n\}$ or, equivalently, $u_i = e_k \in \mathbb{R}^n$. Here, e_k denotes the k-th unit (one-hot) vector, and n denotes the number of distinct tokens. A word-level tokenizer is one of the simplest instances of this.

Then, the token embedding layer maps $(u_1, u_2, \dots, u_L) \mapsto (v_1, v_2, \dots, v_L)$, where

$$v_\ell = a_{u_\ell} \quad \text{or} \quad v_\ell = Au_\ell$$

depending on whether we view $u_\ell = k$ or $u_\ell = e_k$. Here, $A = [a_1\ a_2 \cdots a_n] \in \mathbb{R}^{d \times n}$ is a trainable parameter (defined as an "embedding layer" in PyTorch).

The token embedding layer is applied at the beginning of the transformer.

Single-head self attention

$$Q = \begin{bmatrix} — q_1^\intercal — \\ — q_2^\intercal — \\ \vdots \\ — q_L^\intercal — \end{bmatrix}, \qquad K = \begin{bmatrix} — k_1^\intercal — \\ — k_2^\intercal — \\ \vdots \\ — k_L^\intercal — \end{bmatrix}, \qquad V = \begin{bmatrix} — v_1^\intercal — \\ — v_2^\intercal — \\ \vdots \\ — v_L^\intercal — \end{bmatrix}$$

$$X = \begin{bmatrix} — x_1^\intercal — \\ — x_2^\intercal — \\ \vdots \\ — x_L^\intercal — \end{bmatrix}, \qquad Y = \begin{bmatrix} — y_1^\intercal — \\ — y_2^\intercal — \\ \vdots \\ — y_L^\intercal — \end{bmatrix}$$

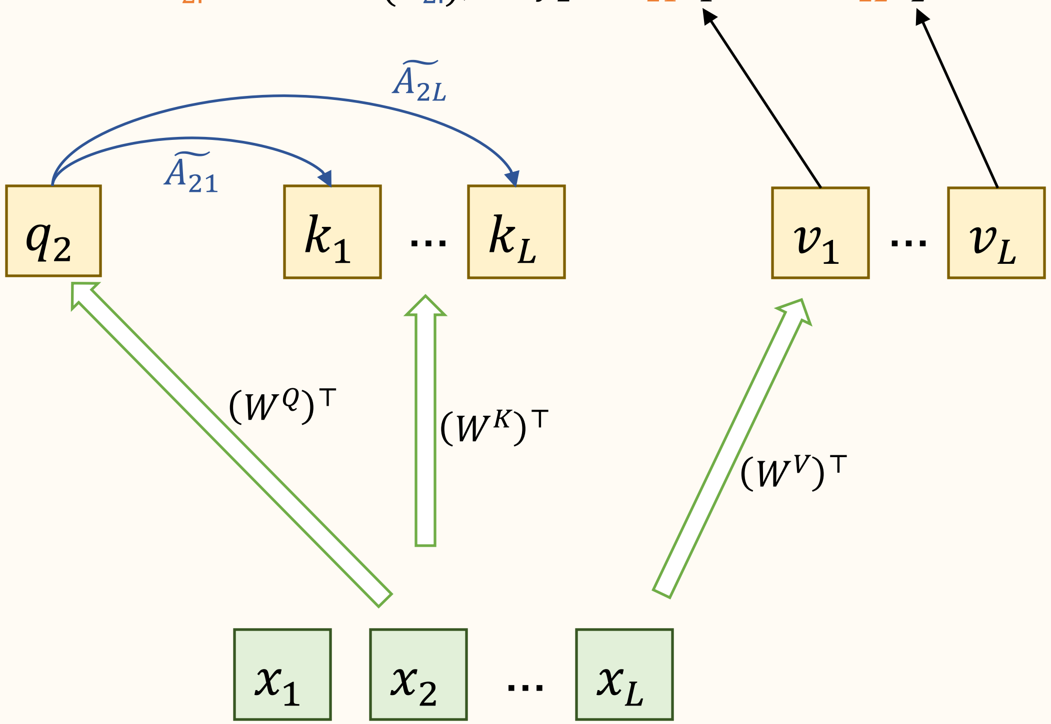
$$x_1, \dots, x_L \in \mathbb{R}^{d_X}, \qquad \{x_\ell\}_{\ell=1}^L = X \in \mathbb{R}^{L \times d_X}$$

$$Q = XW^Q \in \mathbb{R}^{L \times d_K}, \qquad K = XW^K \in \mathbb{R}^{L \times d_K}, \qquad V = XW^V \in \mathbb{R}^{L \times d_V}$$

$$q_\ell = (W^Q)^\intercal x_\ell, \qquad k_\ell = (W^K)^\intercal x_\ell, \qquad v_\ell = (W^V)^\intercal x_\ell$$

$$Y = \text{Attention}(Q, K, V) = \underbrace{\text{softmax}\left(\frac{QK^\intercal}{\sqrt{d_K}}\right)}_{L \times L} \underbrace{V}_{L \times d_V} \in \mathbb{R}^{L \times d_V}$$

$$A_{2:} = \text{softmax}(\widetilde{A_{2:}}), \qquad y_2 = A_{21} v_1 + \cdots + A_{2L} v_L$$



$$A_{ij} = \frac{e^{q_i^\intercal k_j / \sqrt{d_K}}}{\sum_{j'=1}^L e^{q_i^\intercal k_{j'} / \sqrt{d_K}}}, \qquad \text{for } i, j \in \{1, \dots, L\}$$

$$y_\ell = \sum_{r=1}^L A_{\ell r} v_r, \qquad \text{for } \ell = 1, \dots, L$$

$$y_1, \dots, y_L \in \mathbb{R}^{d_V}, \qquad \{y_\ell\}_{\ell=1}^L = Y \in \mathbb{R}^{L \times d_V}$$

Attention is a pseudo-linear operation

Functions $f : \mathbb{R}^n \to \mathbb{R}^m$ of the form

$$f(x) = A(x)x$$

are said to be “pseudo-linear”. (It is not linear because they the matrix $A(x) \in \mathbb{R}^{n\times m}$.)

Attention is a pseudo-linear mapping from $V \in \mathbb{R}^{L\times d_V}$ to $Y \in \mathbb{R}^{L\times d_\mathrm{V}}$.

Pseudo-linear operations are common in signal processing and kernel methods.

(I quickly point this out as it is a nice and simple observation.)

Multi-head self attention (MHA)

Just as one uses multiple CNN channels, we use multiple attention heads.

$$x_1, \ldots, x_L \in \mathbb{R}^{d_X}, \qquad \{x_\ell\}_{\ell=1}^{L} = X \in \mathbb{R}^{L \times d_X}$$

$$\text{for } h = 1, \ldots, H$$

$$Y_h = \text{Attention}(XW_h^Q, XW_h^K, XW_h^V) \in \mathbb{R}^{L \times d_V}$$

$$Z = \text{MHA}(X) = \underbrace{\text{concat}(Y_1, \ldots, Y_H)}_{L \times Hd_V} \; \underbrace{W^O}_{Hd_V \times d_Z}$$

$$z_1, \ldots, z_L \in \mathbb{R}^{d_Z}, \qquad \{z_\ell\}_{\ell=1}^{L} = Z \in \mathbb{R}^{L \times d_Z}$$

Seq-to-seq transformation $\{x_\ell\}_{\ell=1}^{L} \mapsto \{z_\ell\}_{\ell=1}^{L}$. Often $d_X = d_Z$ required by residual connection.

Encoder-only transformer

One transformer layer consists of:

View one layer of TF as a sequence-to-sequence transformation $\left\{x_\ell^{(k)}\right\}_{\ell=1}^{L} \mapsto \left\{x_\ell^{(k+1)}\right\}_{\ell=1}^{L}$. TF stacks many such layers.

The "addition" block is a residual connection, which helps with optimization.

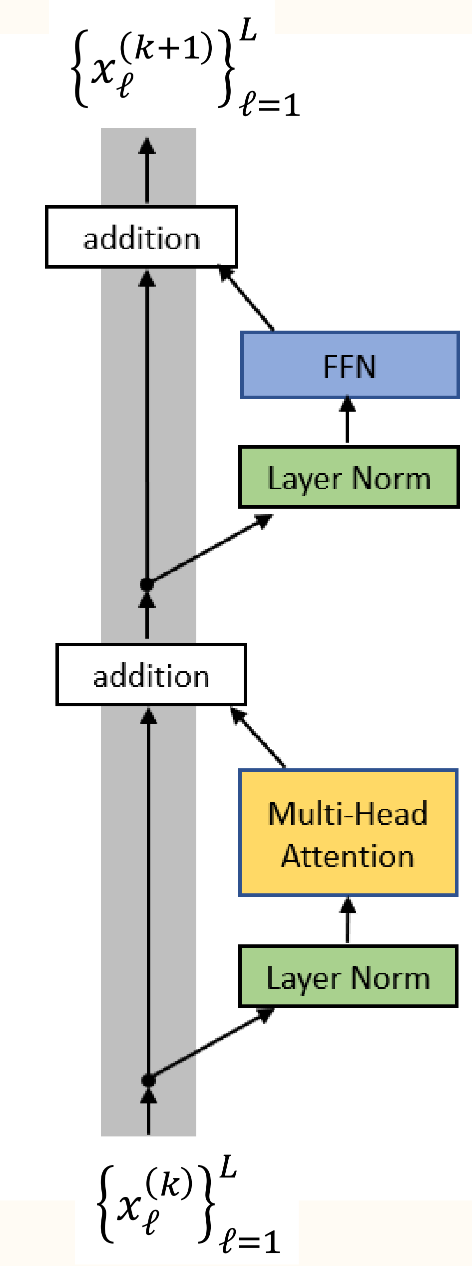


Figure due to:
R. Xiong, Y. Yang, D. He, K. Zheng, X. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T.-Y. Liu, On layer normalization in the transformer architecture, *ICML*, 2020.

Layer normalization

Layer normalization (LN) also stabilizes training by normalizing the features and thereby avoiding exploding and vanishing gradients.

Normalization across the features. Does not normalize over sequence lengths or batch elements. Assume X has dimension (batch × sequence length × channel/feature)

$$\hat{\mu}[:,:] = \frac{1}{C}\sum_{c=1}^{C} X[:,:,c]$$

$$\hat{\sigma}^2[:,:] = \frac{1}{C}\sum_{c=1}^{C} (X[:,:,c] - \hat{\mu}[:,:])^2$$

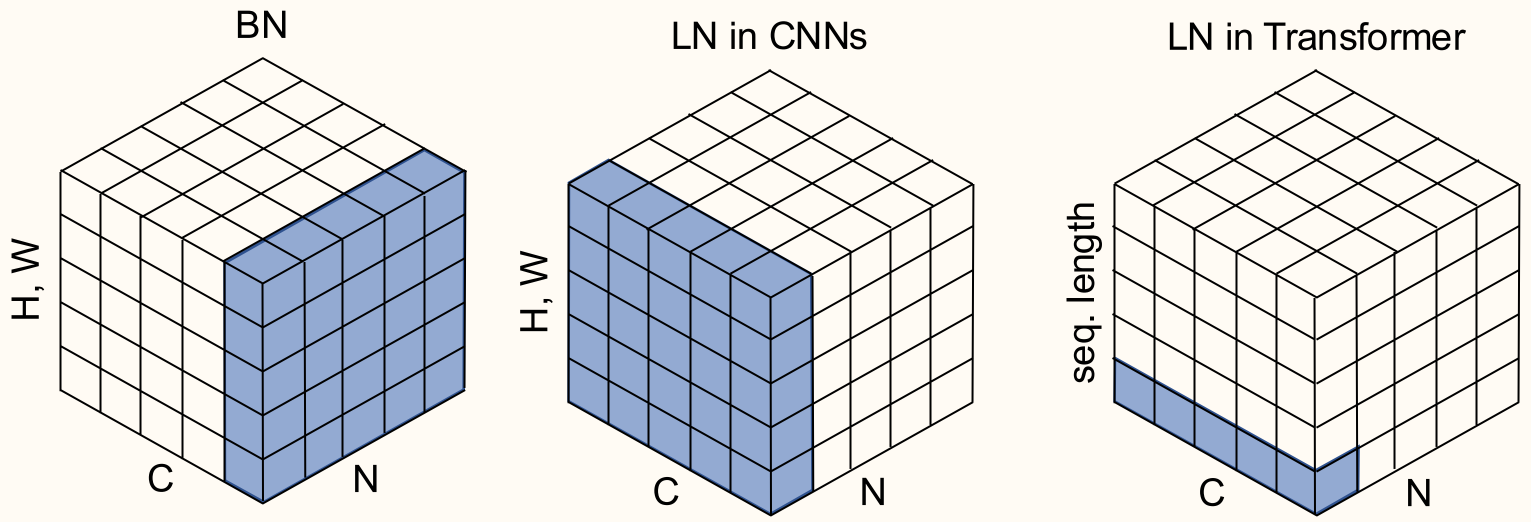
$$\mathrm{LN}_{\gamma,\beta}(X)[:,:,c] = \gamma[c]\frac{X[:,:,c] - \hat{\mu}[:,:]}{\sqrt{\hat{\sigma}^2[:,:] + \varepsilon}} + \beta[c] \qquad c = 1, \dots, C$$

J. L. Ba, J. R. Kiros, and G. E. Hinton, Layer normalization, *arXiv*, 2016.

CNN LN ≠ TF LN

How LN is used in CNNs is different from how it's used in Transformers (including ViT).

For CNNs, LN normalize over channels and spatial dimensions. For transformers, LN normalizes over channels and not over spatial dimensions.



BatchNorm takes in a 4d tensor and normalizes across 3 dimensions: spatial and batch dimensions.

LayerNorm takes in a 3d tensor and normalizes across 1 dimension: vector (channel) dimension.

Position-wise FFN

Position-wise FFN is a 2-layer MLP with ReLU, GELU, or SiLU activation functions:

$$\mathrm{MLP}(x) = W_2\sigma(W_1 x + b_1) + b_2$$

Let d be the token size, i.e., $x \in \mathbb{R}^d$.
Often, $W_1 \in \mathbb{R}^{4d\times d}$, $W_2 \in \mathbb{R}^{d\times 4d}$ (expansion factor of 4).

Applies independently on each sequence element, i.e.,

$$\{x_\ell\}_{\ell=1}^{L} \mapsto \{\mathrm{MLP}(x_\ell)\}_{\ell=1}^{L}$$

GELU, SiLU, Swish activations

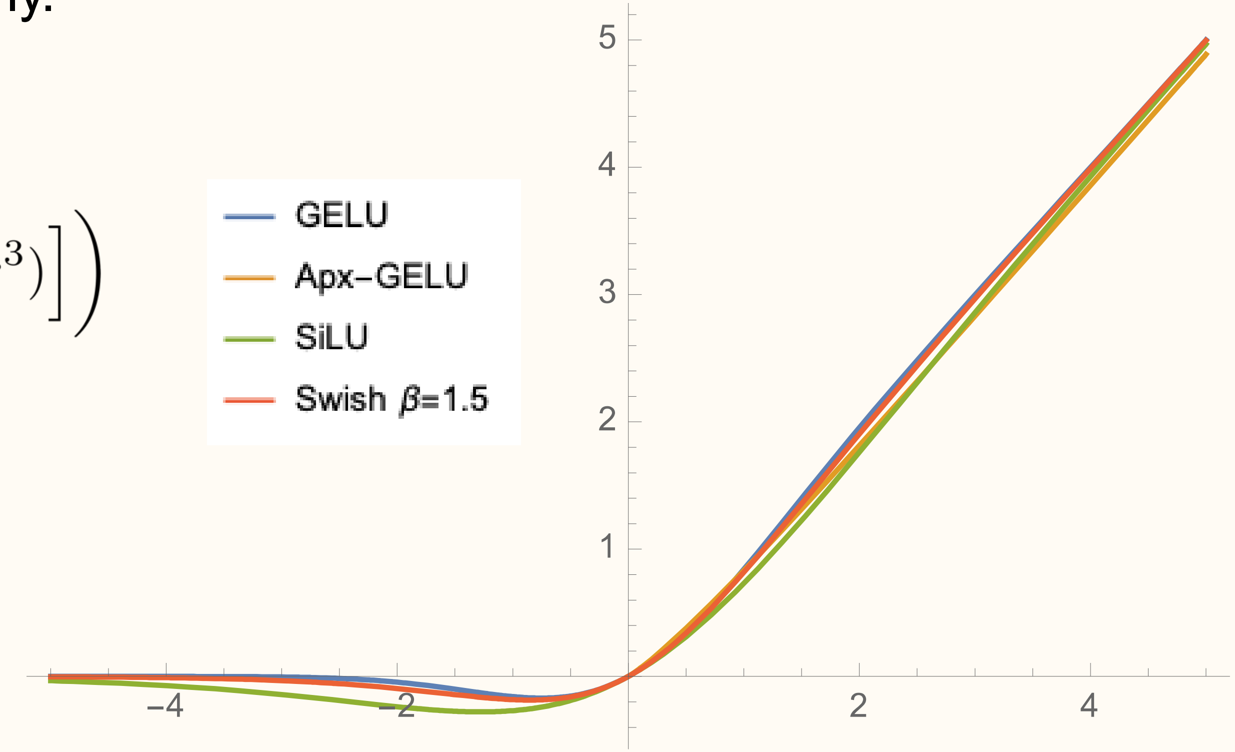
Gaussian error linear unit (GELU), Sigmoid-weighted linear unit (SiLU), and Swish are smooth non-monotone activation functions. The three are qualitatively similar: they decrease near 0 and then increase nearly linearly.

$$\mathrm{GELU}(x) = x\Phi(x) = x\mathbb{P}(Z \le x, Z \sim \mathcal{N}(0,1))$$

$$\approx 0.5\,x\left(1 + \tanh\left[\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)\right]\right)$$

$$\mathrm{SiLU}(x) = x\sigma(x) = \frac{x}{1+e^{-x}}$$

$$\mathrm{Swish}_\beta(x) = x\sigma(\beta x) = \frac{x}{1+e^{-\beta x}}$$



D. Hendrycks and K. Gimpel, Gaussian error linear units (GELUs), *arXiv*, 2016.
P. Ramachandran, B. Zoph, and Q. V. Le, Searching for activation functions, *arXiv*, 2017.
S. Elfwing, E. Uchibe, and K. Doya, Sigmoid-weighted linear units for neural network function approximation in reinforcement learning, *Neural Networks*, 2018.

Positional encoding/embedding

Problem: Encoder-only transformer architecture is permutation equivariant and it does not know positional information of tokens. Relative positions of tokens (word order or patch location) obviously carries important meaning in language.

Solution: After token embedding layer $X \mapsto \{u_\ell\}_{\ell=1}^L \mapsto \{v_\ell\}_{\ell=1}^L$, add positional embedding vectors $\{p_\ell\}_{\ell=1}^L$ and then pass

$$\{v_\ell + p_\ell\}_{\ell=1}^L$$

as input to the transformer layers.

S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, End-to-end memory networks, *NeurIPS*, 2015.
J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, Convolutional sequence to sequence learning, *ICML*, 2017.
A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, Attention is all you need, *NeurIPS*, 2017.

Positional encoding/embedding

NLP transformers often use the sinuisoidal positional encoding $p_1, \dots, p_L \in \mathbb{R}^d$

$$p_\ell = \begin{bmatrix} \sin(\ell/10000^{2\cdot 1/d}) \\ \cos(\ell/10000^{2\cdot 1/d}) \\ \sin(\ell/10000^{2\cdot 2/d}) \\ \cos(\ell/10000^{2\cdot 2/d}) \\ \vdots \\ \sin(\ell/10000^{2\cdot \frac{d}{2}/d}) \\ \cos(\ell/10000^{2\cdot \frac{d}{2}/d}) \end{bmatrix}$$

(Feels like a very arbitrary design, but this work well and is hard to beat.) Since NLP transformers must accommodate arbitrary sequence length L, using a positional encoding with an analytical formula makes sense.

On the other hand, vision transformers let $\{p_\ell\}_{\ell=1}^{L}$ be trainable. Possible since image resolution and hence sequence length L is fixed.

Positional encoding/embedding

Idea is often attributed to Vaswani et al. 2017,

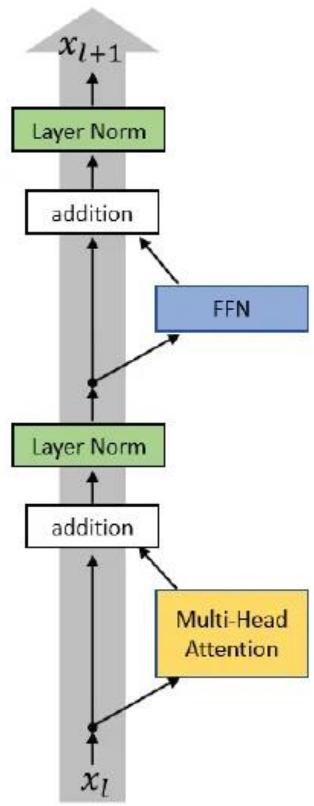
However, Sukhbaatar et al. 2015 and Gehring et al. 2017 did publish the positional encoding technique earlier. The sinusoidal encoding is due to Vaswani et al. 2017.

S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, End-to-end memory networks, *NeurIPS*, 2015.
J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, Convolutional sequence to sequence learning, *ICML*, 2017.
A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, Attention is all you need, *NeurIPS*, 2017.

Post-LN vs Pre-LN TF architectutres

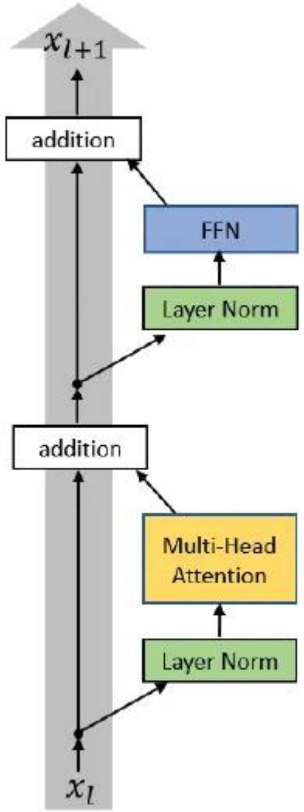
There are 2 variants of the transformer architecture based on the position of LN.

The original (Vaswani et al. 2017) paper illustrates postLN in its figure. However, their updated official codebase uses pre-LN. It is later reported that Pre-LN is more stable.



Post-LN Transformer

$$
\begin{aligned}
x_{l,i}^{post,1} &= \mathrm{MultiHeadAtt}(x_{l,i}^{post}, [x_{l,1}^{post}, \cdots, x_{l,n}^{post}]) \\
x_{l,i}^{post,2} &= x_{l,i}^{post} + x_{l,i}^{post,1} \\
x_{l,i}^{post,3} &= \mathrm{LayerNorm}(x_{l,i}^{post,2}) \\
x_{l,i}^{post,4} &= \mathrm{ReLU}(x_{l,i}^{post,3} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l} \\
x_{l,i}^{post,5} &= x_{l,i}^{post,3} + x_{l,i}^{post,4} \\
x_{l+1,i}^{post} &= \mathrm{LayerNorm}(x_{l,i}^{post,5})
\end{aligned}
$$



Pre-LN Transformer

$$
\begin{aligned}
x_{l,i}^{pre,1} &= \mathrm{LayerNorm}(x_{l,i}^{pre}) \\
x_{l,i}^{pre,2} &= \mathrm{MultiHeadAtt}(x_{l,i}^{pre,1}, [x_{l,1}^{pre,1}, \cdots, x_{l,n}^{pre,1}]) \\
x_{l,i}^{pre,3} &= x_{l,i}^{pre} + x_{l,i}^{pre,2} \\
x_{l,i}^{pre,4} &= \mathrm{LayerNorm}(x_{l,i}^{pre,3}) \\
x_{l,i}^{pre,5} &= \mathrm{ReLU}(x_{l,i}^{pre,4} W^{1,l} + b^{1,l}) W^{2,l} + b^{2,l} \\
x_{l+1,i}^{pre} &= x_{l,i}^{pre,5} + x_{l,i}^{pre,3}
\end{aligned}
$$

Final LayerNorm: $x_{Final,i}^{pre} \leftarrow \mathrm{LayerNorm}(x_{L+1,i}^{pre})$

R. Xiong, Y. Yang, D. He, K. Zheng, X. Zheng, C. Xing, H. Zhang, Y. Lan, L. Wang, and T.-Y. Liu, On layer normalization in the transformer architecture, *ICML*, 2020.

Transformer depth

Thanks to the residual connections and layer norm, transformers can often be much deeper than stacked RNNs. (ELMo has 2 layers, while BERT has 24 layers.)

To clarify, the layer norm and the residual connection are used to mitigate the exploding/vanishing gradient problem across the transformer depth.

The transformer does not have exploding/vanishing gradient problem along the sequence length L due to its use of attention mechanism.

Why transformers over RNNs?

Handling long sequence length:

RNNs struggle with long input sequences due to a fixed memory size and vanishing or exploding gradients. LSTMs are designed to mitigate this problem, but transformers really solve this problem. Transformers allows the full input sequence to be considered without having to compress the information into a hidden state vector.

Efficient parallel computation:

RNNs are inherently sequential (inefficient) during training. (RNNs are efficient during inference.) In contrast, transformers are completely parallelizable in training, and we can better leverage efficient large-scale GPU computation.

BERT pre-training

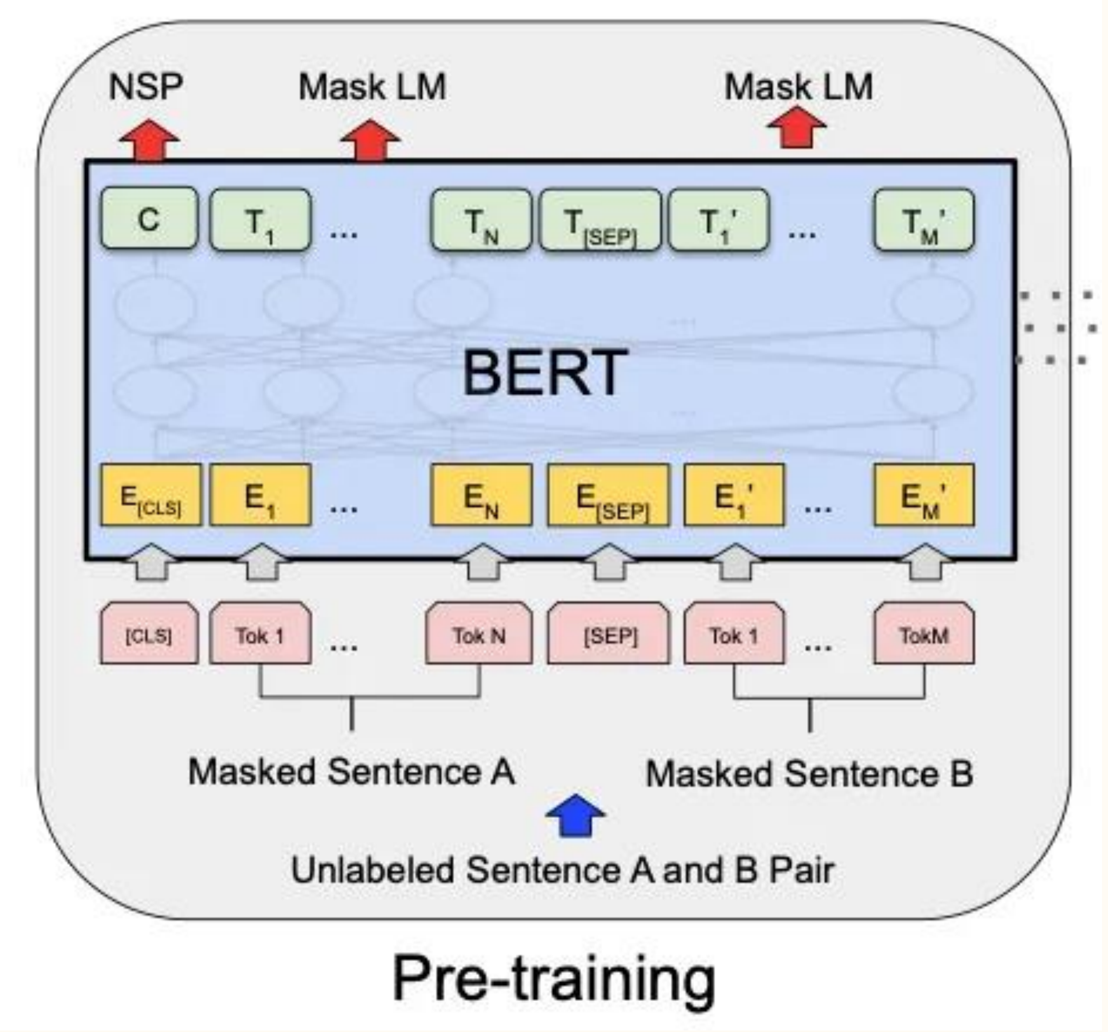
BERT pre-training uses two losses.

1. Masked LM (MLM)

Randomly mask out 15% of the words and let BERT predict it. Output tokens corresponding to masked words are fed into softmax and CE loss.

2. Next sentence prediction (NSP)

Provide two sentences A and B separated with [SEP] token with 50% probability of B following A and 50% probability of B unrelated to A, and make binary prediction. Attach classification head to the output corresponding to [CLS] token.

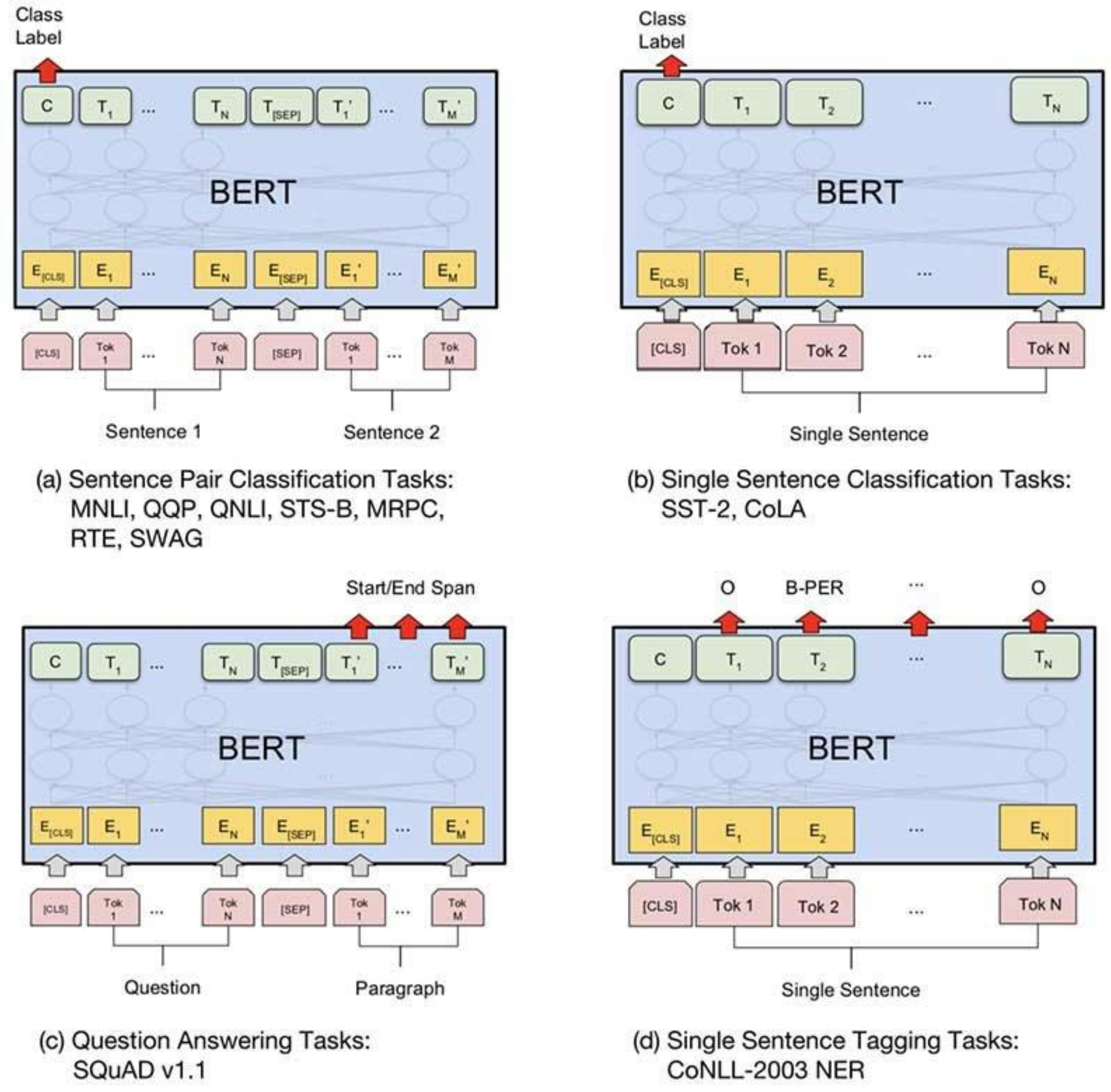


BERT fine-tuning

Many NLP tasks roughly fit the MLM and NSP shape.

For fine-tuning, make minimal modifications to the BERT baseline and fine-tune the whole model.

Fine-tuning is computationally very cheap (<1 hour on a single Google TPU).



(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

(b) Single Sentence Classification Tasks: SST-2, CoLA

(c) Question Answering Tasks: SQuAD v1.1

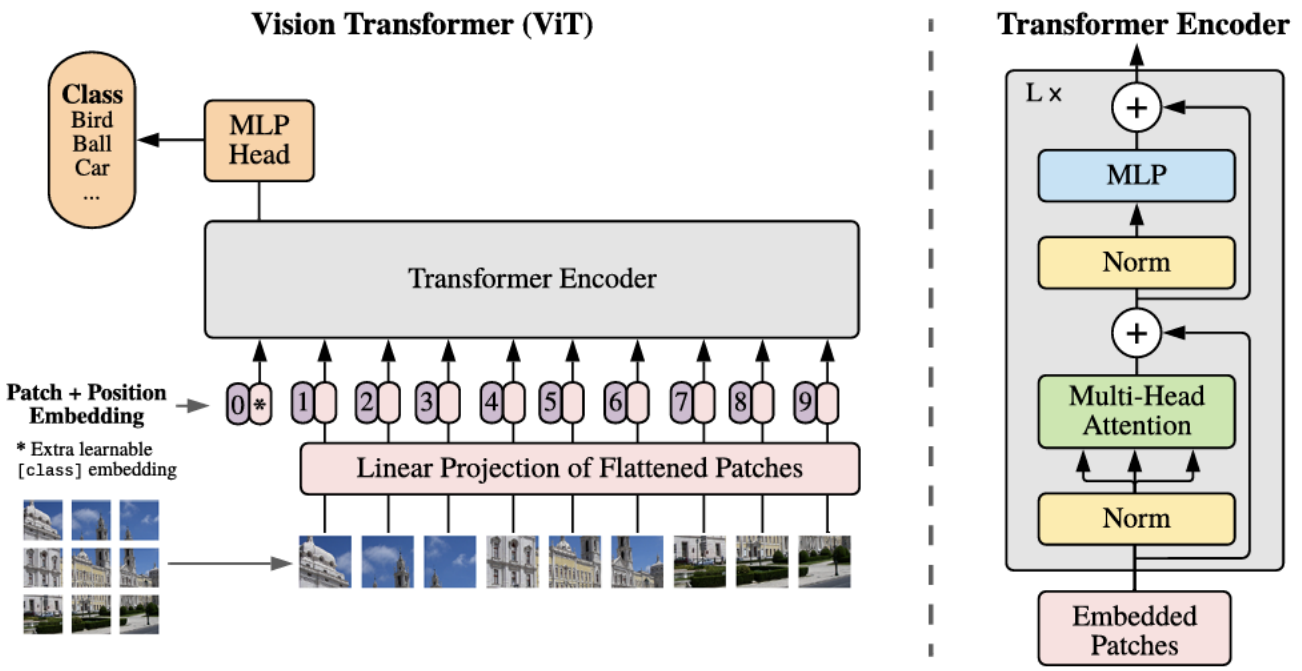
(d) Single Sentence Tagging Tasks: CoNLL-2003 NER

Vision transformer

Vision transformer is an encoder-only transformer architecture.

Given an image, each 16×16 patch is a token, and the patches are placed into a linear sequence.

Output corresponding to [CLS] token is used for classification.



Supervised pre-training on image classification had better performance compared to self-supervised pre-training with masked patch prediction.

A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, An image is worth 16x16 words: Transformers for image recognition at scale, *ICLR*, 2021.

GPT-1

GPT (generative pre-training) uses a causal language model loss

$$\mathcal{L}(\theta) = \sum_{\ell=1}^{L-1} -\log p_\theta(u_{\ell+1} \,|\, u_1, \ldots, u_\ell)$$

Initially, GPT was trained to be an self-supervised pre-trained model in the vein of BERT, and the its text generation ability was not that strong.

(However, the focus of GPT-2 shifted to text generation.)

Compared to BERT's encoder only transformer, GPT-1's decoder-only transformer additionally utilizes (i) masked attention and (ii) output projection layers.

A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, Improving language understanding by generative pre-training, 2018.

Masked attention

In RNNs, information is naturally processed sequentially.

However, there is a problem with using an encoder-only (BERT-style) transformer for a causal language model: The model can see the entire sequence, the past and the future.

Therefore, GPT uses a masked attention that allows the current sequence element to only query earlier sequence elements.

Masked single-head self attention

$$x_1, \ldots, x_L \in \mathbb{R}^{d_X}, \qquad \{x_\ell\}_{\ell=1}^L = X \in \mathbb{R}^{L \times d_X}$$

$$Q = XW^Q \in \mathbb{R}^{L \times d_K}, \qquad K = XW^K \in \mathbb{R}^{L \times d_K}, \qquad V = XW^V \in \mathbb{R}^{L \times d_V}$$

$$\tilde{A}_{ij} = \begin{cases} q_i^\intercal k_j / \sqrt{d_K} & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases} \qquad \text{for } i, j \in \{1, \ldots, L\}$$

$$Y = \mathrm{CausalAttention}(Q, K, V) = \underbrace{\mathrm{softmax}(\tilde{A})}_{L \times L} \underbrace{V}_{L \times d_V} \in \mathbb{R}^{L \times d_V}$$

Only lower-triangular components of $\tilde{A}$ are finite.

Only lower-triangular components of A are nonzero.

Crucially, q_i is allowed to query only $k_1, \ldots, k_i$.

$$A_{ij} = \frac{e^{\tilde{A}_{ij}}}{\sum_{j'=1}^{L} e^{\tilde{A}_{ij'}}}, \qquad \text{for } i, j \in \{1, \ldots, L\}$$

$$y_\ell = \sum_{r=1}^{\ell} A_{\ell r} v_r, \qquad \text{for } \ell = 1, \ldots, L$$

y_ℓ is a linear combination of $v_1, \ldots, v_\ell$.
$\big(\exp(-\infty) = 0\big)$

$$y_1, \ldots, y_L \in \mathbb{R}^{d_V}, \qquad \{y_\ell\}_{\ell=1}^L = Y \in \mathbb{R}^{L \times d_V}$$

y_ℓ only depends on $x_1, \ldots, x_\ell$.
($\{x_\ell\}_{\ell=1}^L \mapsto \{y_\ell\}_{\ell=1}^L$ has causal dependency)

Masked multi-head self attention

$$x_1, \ldots, x_L \in \mathbb{R}^{d_X}, \qquad \{x_\ell\}_{\ell=1}^{L} = X \in \mathbb{R}^{L \times d_X}$$

$$\text{for } h = 1, \ldots, H$$

$$Y_h = \text{CausalAttention}(XW_h^Q, XW_h^K, XW_h^V) \in \mathbb{R}^{L \times d_V}$$

$$Z = \text{CausalMHA}(X) = \underbrace{\text{concat}(Y_1, \ldots, Y_H)}_{L \times Hd_V} \underbrace{W^O}_{Hd_V \times d_Z}$$

$$z_1, \ldots, z_L \in \mathbb{R}^{d_Z}, \qquad \{z_\ell\}_{\ell=1}^{L} = Z \in \mathbb{R}^{L \times d_Z}$$

Seq-to-seq transformation $\{x_\ell\}_{\ell=1}^{L} \mapsto \{z_\ell\}_{\ell=1}^{L}$ with causal dependence:
z_ℓ only depends on $x_1, \ldots, x_\ell$.

Since other components of transformer all act positionwise, the transformer with causal MHA is a seq-to-seq transformation with causal dependence.

Output projection layer

The transformer layers produce embedding vectors $v_1, \ldots, v_L \in \mathbb{R}^d$. These must be enlarged to dimension n to produce a probability distribution on the n possible tokens. (Usually $d < n$.)

The output projection layer

$$w_\ell = B v_\ell$$

does this.

Decoder-only transformer

$\mu(w_1) \cdots \mu(w_L) \in \Delta^n$

set of probability mass vectors

softmax function μ

$w_1 \cdots w_L \in \mathbb{R}^n$

trainable

output projection $w_\ell = B v_\ell$

$v_1 \cdots v_L \in \mathbb{R}^d$

hidden dimension d doesn't change due to residual connection

$\times$ Depth TF layers with causal mask

$v_1 \cdots v_L \in \mathbb{R}^d$ $\quad v_\ell = A u_\ell + p_\ell$

token embedding + positional encoding

$u_1 \cdots u_L \in \mathbb{R}^n$

τ

X

Llama 3.1 405B	GPT3 175B
depth (layers) 126	depth (layers) 96
d (hidden) 16k	d (hidden) 12k
n (tokens) 128k	n (tokens) 50k

Output projection layer

It is customary to do weight tying, and set

$$B = A^\top$$

where $A \in \mathbb{R}^{d \times n}$ is the matrix for the token embedding layer. Empirically, this works well.#

Why might this be a good idea? In modern signal processing (dictionary learning and compressed sensing), the transpose of a matrix is often used as an approximate inverse. Indeed,

$$A : \text{token} \mapsto \text{emb. vector}$$
$$B : \text{emb. vector} \mapsto \text{token}$$

are, loosely speaking, inverse operations. Interestingly, at initialization, $A^\top$ does act like an inverse of A. More precisely, if $A \sim \mathcal{N}(0, \sigma^2)$ IID with $A \in \mathbb{R}^{d \times n}$, then

$$A^\top A \propto I$$

#O. Press and L. Wolf, Using the output embedding to improve language models, *EACL*, 2017.

Output projection layer

At initialization, if $A \sim \mathcal{N}(0, \sigma^2)$ IID with $A \in \mathbb{R}^{d \times n}$, then $\frac{1}{d\sigma^2} A^\top A \approx I$.

```
d, n, sigma = 100, 50, 2.0
A = np.random.normal(scale=sigma, size=(d, n))
print("Sample of A^T A:\n", (A.T @ A /(d*sigma**2))[:5, :5])  # print 5x5 block
```

```
Sample of A^T A:
[[ 1. -0.01 -0.06 0. 0. ]
[-0.01 0.9 -0.04 0.09 0.07]
[-0.06 -0.04 0.83 -0.07 -0.08]
[ 0. 0.09 -0.07 0.95 0.04]
[ 0. 0.07 -0.08 0.04 1.19]]
```

(There is no reason to expect $A^\top A \propto I$ to continue to hold once training begins.)

Main point: (i) B is conceptually performing the inverse operation of A (ii) $A^\top$ is an approximate inverse of A at initialization (iii) it is sensible to weight-tie $B = A^\top$.

Self-supervised pre-training

Let X be the input text tokenzed as $\tau(X) = (u_1, u_2, \dots, u_L)$.

Let f_θ be the transformer mapping $\{u_\ell\}_{\ell=1}^{L} \mapsto \{w_\ell\}_{\ell=1}^{L}$, where $w_\ell \in \mathbb{R}^n$. Then,

$$p_\theta(u_{\ell+1} \,|\, u_1, \dots, u_\ell) = \mathrm{softmax}(w_\ell)$$

and

$$\begin{aligned}\mathcal{L}(\theta) &= \sum_{\ell=1}^{L} -\log p_\theta(u_{\ell+1} \,|\, u_1, \dots, u_\ell) \\ &= \sum_{\ell=1}^{L} \ell^{\mathrm{CE}}(w_\ell, u_{\ell+1})\end{aligned}$$

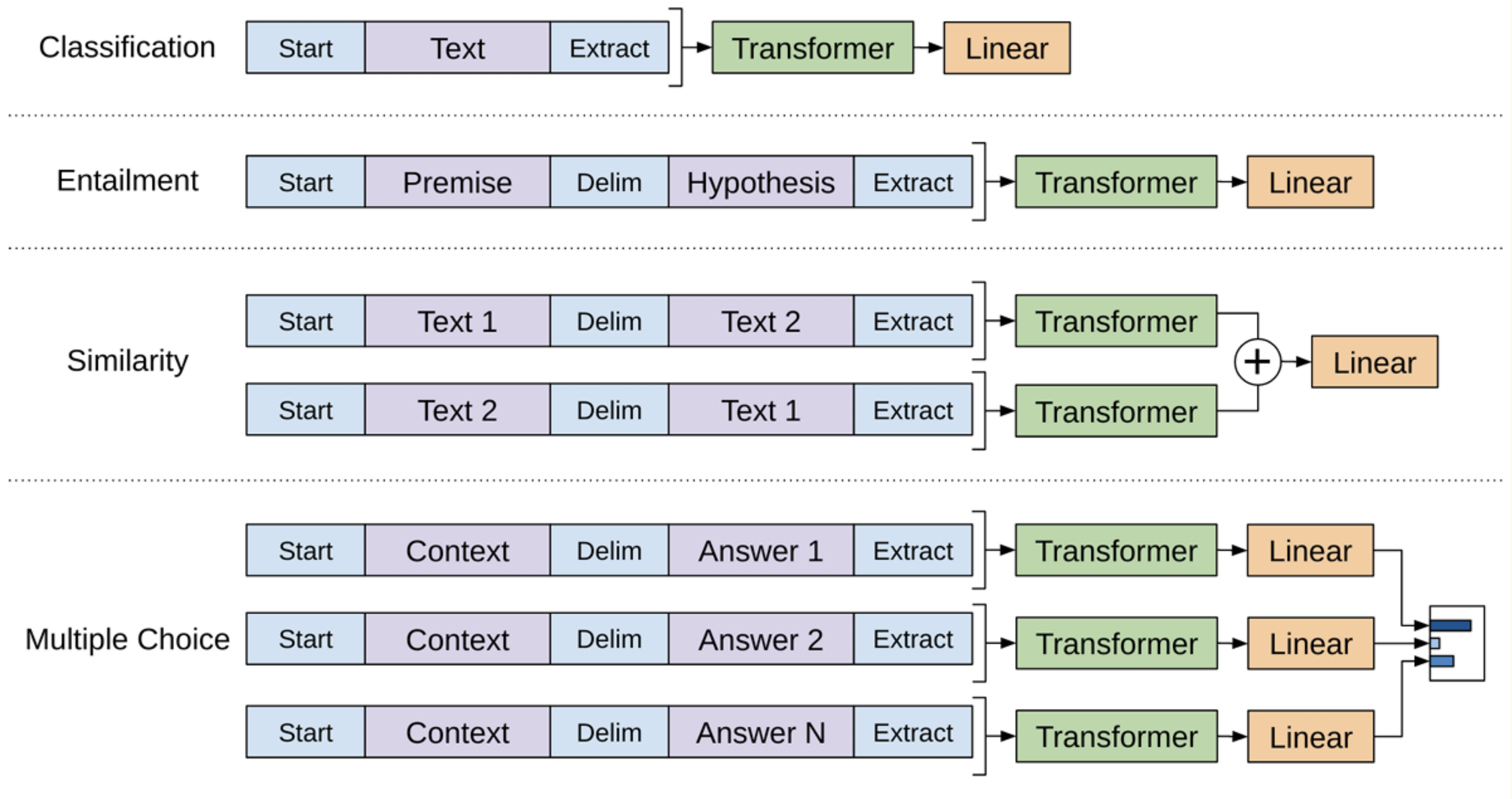
where ℓ^{CE} is the cross-entropy loss with $u_{\ell+1} \in \{1, \dots, n\}$ above viewed as an integer.

Supervised fine-tuning

First, transform the relevant text into sequence with appropriate delimiter tokens.

At the end of the transformer, the token corresponding to the “extract” token position is extracted fed into a linear layer.

The full GPT-1 model and the final linear layer is fine-tuned.



Supervised fine-tuning

For classification, given an input text X and a tokenizer τ, the transformer maps

$$(\text{<Start>}, \tau(X), \text{<Extract>}) = \{u_\ell\}_{\ell=1}^{L} \mapsto \{w_\ell\}_{\ell=1}^{L}$$

The final token w_L corresponding to the <Extract> token, is extracted. The loss is

$$\text{loss}(Aw_L + b, Y)$$

where A and b are the parameters of the linear layer and Y is the label corresponding to X. (Only w_L is used for the loss and $\{w_\ell\}_{\ell=1}^{L-1}$ is not used).

BERT had a <Cls> token at the start of the input, and it basically served the same role as the <Extract> token for GPT. Different from BERT, GPT is a causal language model, so the <Extract> token must be at the end if we want w_L to encode information about the full sentence.

The full GPT-1 model (the pre-trained TF), the final linear layer, and the vector embeddings corresponding to <Start>, <Extract>, and <Delim> are trained.

(LLM fine-tuning is no longer done this way.)

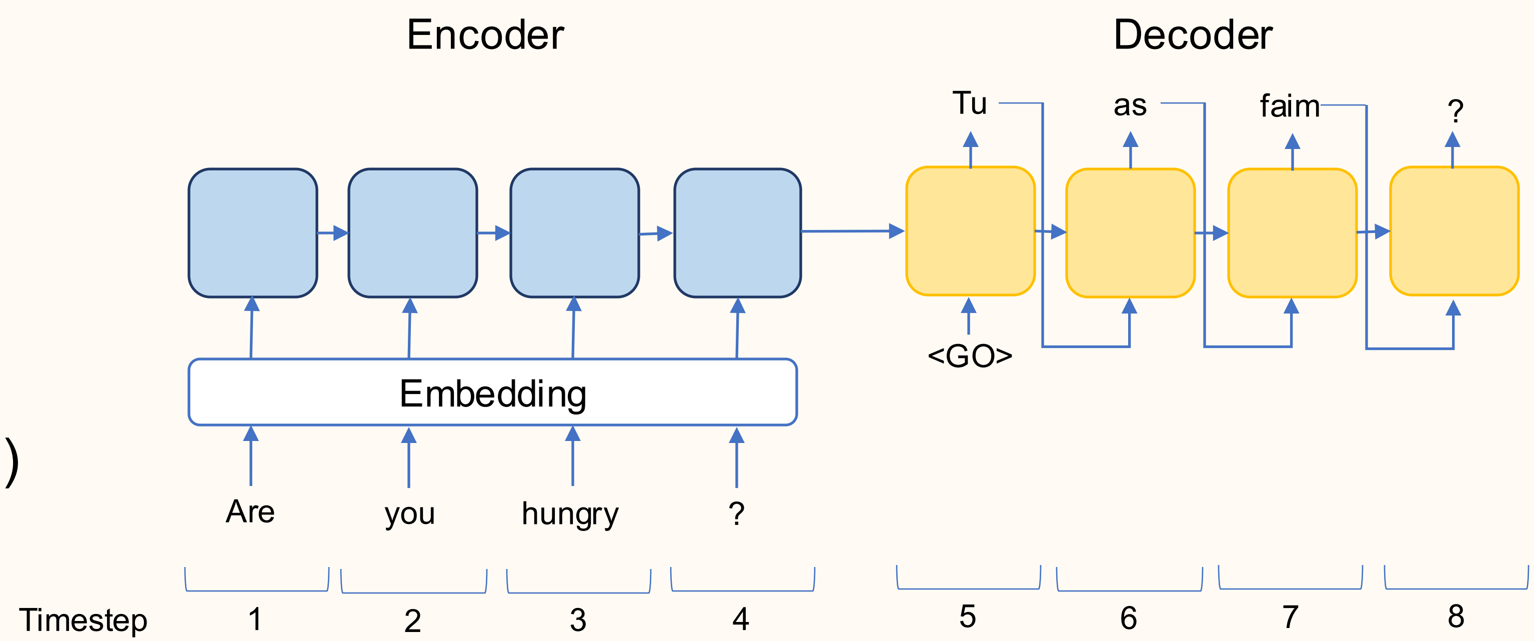
Example task: Machine translation

In machine translation, training data contains translation pairs between different languages.

Classically with an RNN, the encoding stage encodes (summarizes) the entire sentence into a latent vector, and the decode generates translation text autoregressively.

(For better performance, a stacked bidirectional RNN encoder and a stacked unidirectional RNN decoder should be used.)

(Interestingly, reversing the input sentence often improves performance.)

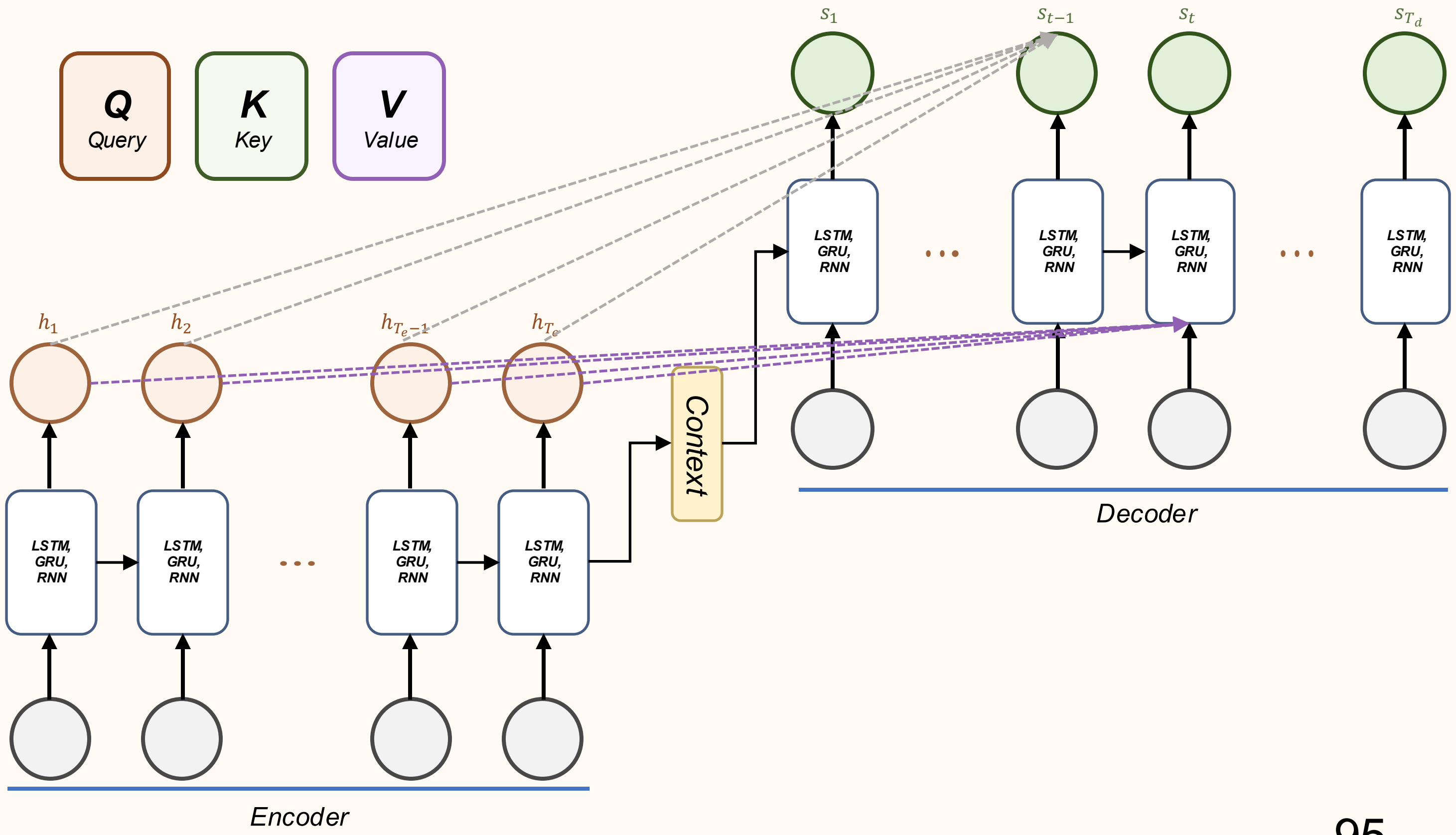


Bahdanau attention and cross attention

The problem with the previous approach is that the hidden state passed from the encoder RNN to the decoder RNN acts as a bottleneck, and the hidden state may not be able to retain all the necessary information.

Solution: Allow the decoder RNN cells to access the hidden states of the encoder RNNs.

This attention mechanism is now called *cross attention*, and it is now commonly used to attend across different modalities. E.g. text decoder attends to image patches.

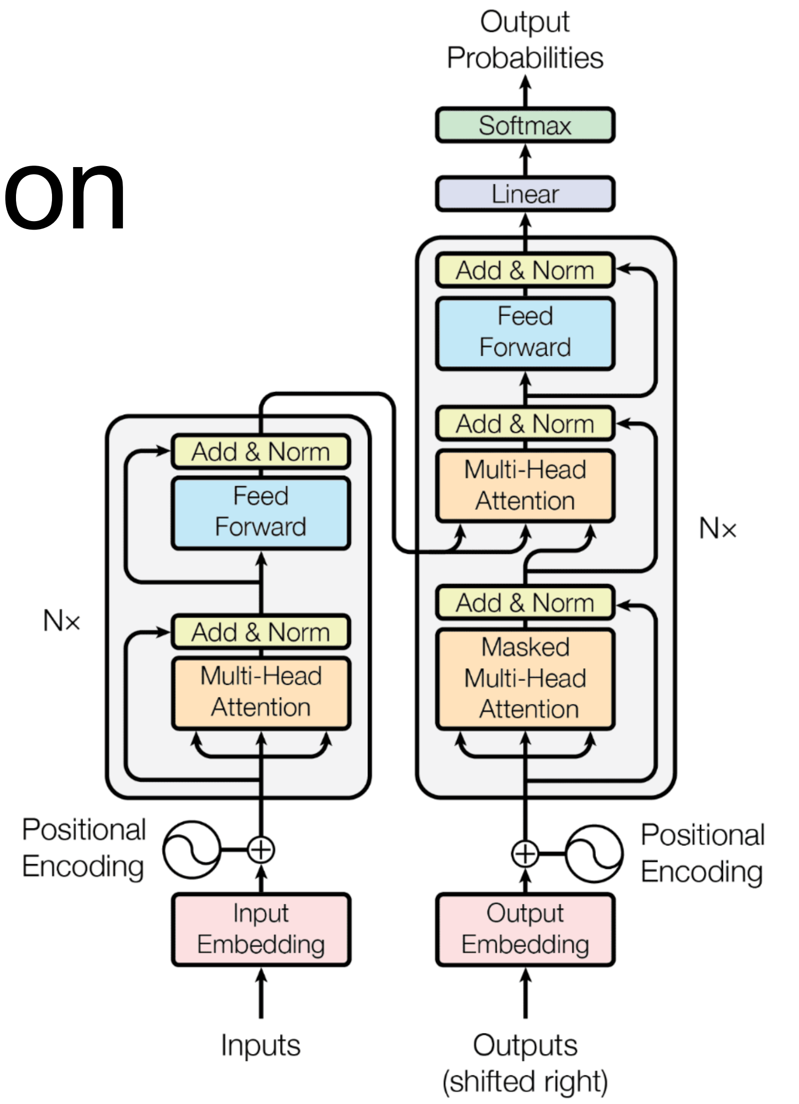


D. Bahdanau, K. Cho, Y. Bengio, Neural machine translation by jointly learning to align and translate, *ICLR*, 2015. (ICLR 2025 Test of Time award Runner Up.)

Transformer and cross attention

Vaswani et al. questioned whether the RNN mechanism was necessary. They concluded "Attention is all you need".

Cross attention layer derives $\{q_\ell\}_{\ell=1}^{L}$ from previous layer's $\{x_\ell^{\mathrm{dec}}\}_{\ell=1}^{L}$ but $\{k_\ell\}_{\ell=1}^{L'}$ and $\{v_\ell\}_{\ell=1}^{L'}$ are derived from the encoder layer's $\{x_\ell^{\mathrm{enc}}\}_{\ell=1}^{L}$. (In cross attention, number of queries need not match the number of keys and values.)



(Figure incorrectly depicts post-LN.)

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, Attention is all you need, *NeurIPS*, 2017.

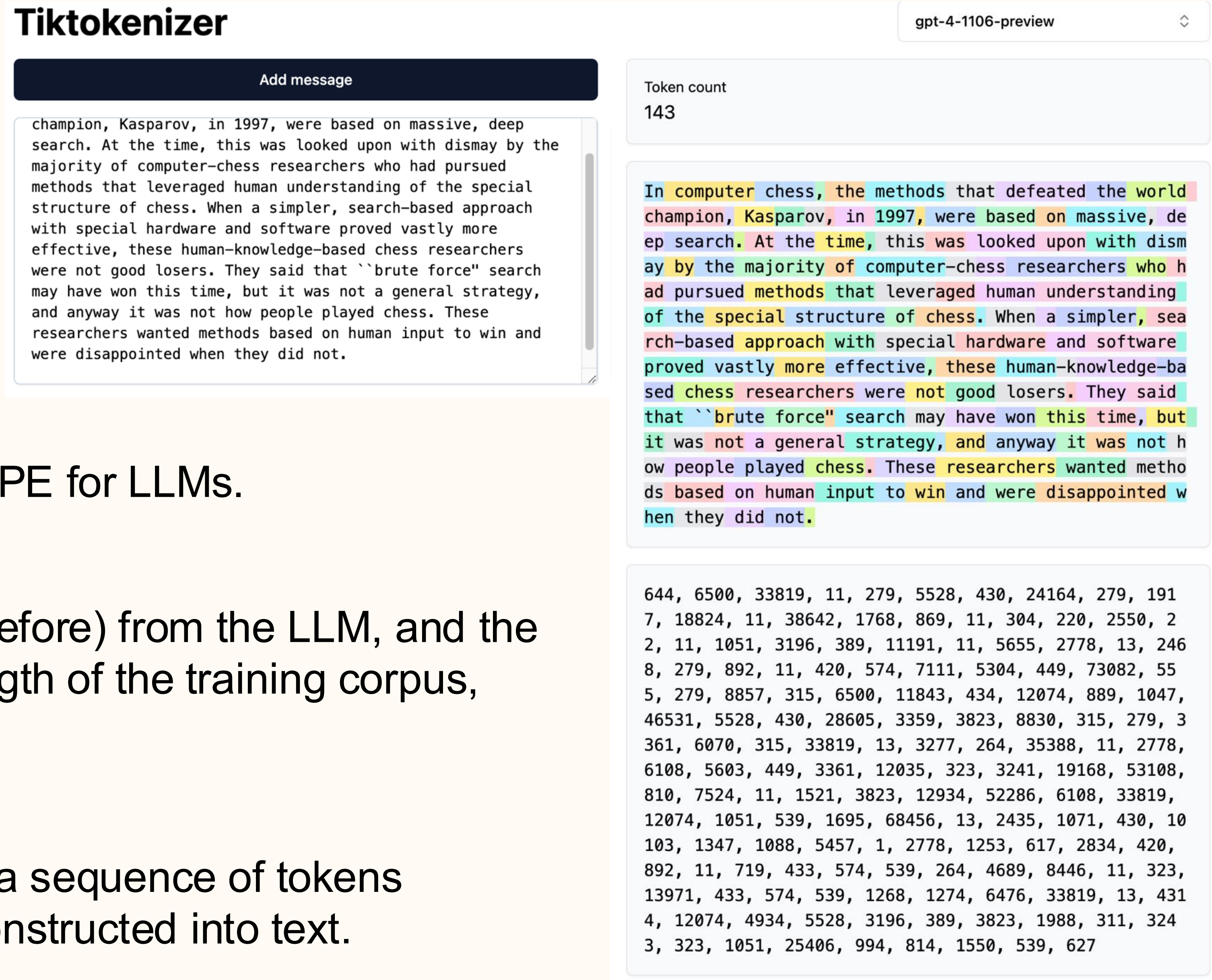
Understanding TF from historical context

The transformer architecture feels somewhat arbitrary, but we can understand the designers' intent through the historical context.

There is no mathematical or first-principles reason that things must be the way they are, and the standard architecture will likely change in the future.

The historical context does inform us of the intended purpose of the components, and it gives us a rough guideline of what things will certainty not work and what new components may work.

Byte-pair encoding



Byte-pair encoding (BPE) is a sub-word tokenizer.

The GPT-1 popularized the use of BPE for LLMs.

Tokenizers are trained separately (before) from the LLM, and the training minimizes the sequence length of the training corpus, given a fixed vocabulary size.

For text generation, LLMs generate a sequence of tokens (integers) and those tokens are reconstructed into text.

Tiktokenizer website: https://tiktokenizer.vercel.app/
P. Gage, A new algorithm for data compression, *C Users J.*, 1994.
R. Sennrich, B. Haddow, and A. Birch, Neural machine translation of rare words with subword units, *ACL*, 2016.
A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, Improving language understanding by generative pre-training, 2018.

Byte-pair encoding

The BPE tokenizer can gracefully deal with misspellings.

```
time is an oshean but it end's at the shoaer.

4580, 382, 448, 1994, 144309, 889, 480, 1268, 885, 54
0, 290, 641, 13750, 259, 13
```

Subtleties of the BPE tokenizers:

- Pre-pending space to a word changes the tokenization.
- Numbers are chunked up somewhat arbitrarily (making arithmetic quite difficult for LLMs.)

```
machine machine Machine learning learning.

57857 + 345834 = 403691

48082, 7342, 19121, 7524, 7524, 364, 42747, 5085, 659,
220, 22901, 49302, 314, 220, 23909, 46173
```

Andrej Karpathy's tutorial *Let's build the GPT Tokenizer* is an excellent resource for learning about further details of the BPE tokenizer: https://youtu.be/zduSFxRajkE?si=hJ3gTgpfSJ4PWsQd

Modern transformers: RMS Norm

Recall that LayerNorm have the form

$$\mathrm{LN}_{\gamma,\beta}(x) = \gamma \odot \frac{x - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \varepsilon}} + \beta$$

where $x = X[b, \ell, :]$ (i.e., LN normalizes across the channel/feature dimension). The γ and β are trainable parameters.

Root mean square layer normalization (RMSNorm) simplifies this to

$$\mathrm{RMSN}_{\gamma} = \gamma \odot \frac{x}{\sqrt{\|x\|^2 + \varepsilon}}$$

where γ is the only trainable parameter.

B. Zhang and R. Sennrich, Root mean square layer normalization, *NeurIPS*, 2019.

Modern transformers: RMS Norm

RMSNorm is clearly simpler than LN, but does it matter?

$$\mathrm{LN}_{\gamma,\beta}(x) = \gamma \odot \frac{x - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \varepsilon}} + \beta \qquad\qquad \mathrm{RMSN}_{\gamma} = \gamma \odot \frac{x}{\sqrt{\|x\|^2 + \varepsilon}}$$

RMSNorm performs (generalizes) similarly to LN.[#]

RMSNorm is meaningfully faster than LN (due to low-level memory movement issues) despite having essentially the same flop count.[%]

	% flop	% Runtime
Matrix multiplication	99.80	61.0
Normalization	0.17	25.5
Element-wise	0.03	13.5

[#] B. Zhang and R. Sennrich, Root mean square layer normalization, *NeurIPS*, 2019.
[%] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, Data movement is all you need: A Case study on optimizing transformers, *MLSys*, 2021.

Modern transformers: No bias

In fact, most modern transformers mostly avoid bias terms.

- Token embedding and output projection layers have no bias terms.
- Attention layers are designed without bias terms.

The original transformer used bias terms in the FFN layer:

$$\mathrm{FFN}(x) = W_2\sigma(W_1 x + b_1) + b_2$$

But modern implementations do not. If not gated, the have the form:

$$\mathrm{FFN}(x) = W_2\sigma(W_1 x)$$

Modern transformers: GLU

Gated Linear Units (GLU) slightly improve the FFN layers:

$$\mathrm{FFN}(x) = W_2\big(\sigma(W_1 x) \odot (Vx)\big)$$

where $\odot$ is elementwise multiplication. The trainable parameters are W_1, W_2, and V.

- If σ is ReLU, this FFN is called ReGLU.
- If σ is GeLU, this FFN is called GeGLU.
- If σ is swish, this FFN is called SwiGLU.

Model	Log-Perplexity
$\mathrm{FFN}_{\mathrm{ReLU}}$	1.677
$\mathrm{FFN}_{\mathrm{GELU}}$	1.679
$\mathrm{FFN}_{\mathrm{Swish}}$	1.683
$\mathrm{FFN}_{\mathrm{GEGLU}}$	**1.633**
$\mathrm{FFN}_{\mathrm{SwiGLU}}$	**1.636**
$\mathrm{FFN}_{\mathrm{ReGLU}}$	1.645

As the table from[#] shows, GLU variants work slightly bettter than non-gated FFNs.

Y. N. Dauphin, A. Fan, M. Auli, and D. Grangier, Language modeling with gated convolutional networks, *ICML*, 2017.
[#]N. Shazeer, GLU variants improve transformer, *arXiv*, 2020.

Modern transformers: GLU

But doesn't this increase the compute cost?

For non-gated FFNs, $\mathrm{FFN}(x) = W_2\sigma(W_1 x)$ the standard dimension parameters are $W_1 \in \mathbb{R}^{4d \times d}$ and $W_2 \in \mathbb{R}^{d \times 4d}$. (Many independent tuning attempts have lead to the hyperparameter value of 4)

For gated FFN, $\mathrm{FFN}(x) = W_2\big(\sigma(W_1 x) \odot (Vx)\big)$ the standard dimension parameters are $W_1, V \in \mathbb{R}^{(8/3)d \times d}$ and $W_2 \in \mathbb{R}^{d \times (8/3)d}$.

With the value $(8/3)$, the flop count is $\sim 16d^2$ for both architectures.

Classical positional embeddings

Recall classical positional embeddings: After token embedding layer $X \mapsto \{u_\ell\}_{\ell=1}^L \mapsto \{v_\ell\}_{\ell=1}^L$, add positional embedding vectors $\{p_\ell\}_{\ell=1}^L$ and then pass

$$\{v_\ell + p_\ell\}_{\ell=1}^L$$

as input to the transformer layers. Then,

$$q_\ell = W_Q^\top (v_\ell + p_\ell)$$
$$k_\ell = W_K^\top (v_\ell + p_\ell)$$
$$v_\ell = W_V^\top (v_\ell + p_\ell)$$

$$p_\ell = \begin{bmatrix} \sin(\ell\theta_1) \\ \cos(\ell\theta_1) \\ \sin(\ell\theta_2) \\ \cos(\ell\theta_2) \\ \vdots \\ \sin(\ell\theta_{d_K/2}) \\ \cos(\ell\theta_{d_K/2}) \end{bmatrix}$$

$$\theta_i = 10000^{-2i/d}, \quad i = 1, \ldots, d_K/2$$

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, Attention is all you need, *NeurIPS*, 2017.

Modern transformers: RoPE

Rotary Position Embedding (RoPE) instead identifies

$$W_Q^\top v_\ell, W_K^\top v_\ell \in \mathbb{C}^{d_K/2} \cong \mathbb{R}^{d_K}$$

and

$$k_\ell = \begin{bmatrix} e^{i\ell\theta_1} \\ e^{i\ell\theta_2} \\ \vdots \\ e^{i\ell\theta_{d_K/2}} \end{bmatrix} \odot (W_K^\top v_\ell) \qquad q_\ell = \begin{bmatrix} e^{i\ell\theta_1} \\ e^{i\ell\theta_2} \\ \vdots \\ e^{i\ell\theta_{d_K/2}} \end{bmatrix} \odot (W_Q^\top v_\ell)$$

$$\theta_i = 10000^{-2(i-1)/d}, \quad i = 1, \ldots, d_K/2$$

So,

$$k_1, \ldots, k_L, q_1, \ldots, q_L \in \mathbb{C}^{d_K/2} \cong \mathbb{R}^{d_K}$$

J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, RoFormer: Enhanced transformer with rotary position embedding, *Neurocomputing*, 2024.

Modern transformers: RoPE

Then, RoPE feeds the rotated q and k vectors into the attention function. (v vectors are not rotated.)

$$q_\ell = R_\ell W_Q^\top v_\ell$$
$$k_\ell = R_\ell W_K^\top v_\ell$$
$$v_\ell = W_V^\top v_\ell$$

$$\tilde{A}_{ij} = \frac{q_i^\intercal k_j}{\sqrt{d_K}}$$
$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\tilde{A})V$$

$$R_\ell = \begin{pmatrix} \cos \ell\theta_1 & -\sin \ell\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ \sin \ell\theta_1 & \cos \ell\theta_1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos \ell\theta_2 & -\sin \ell\theta_2 & \cdots & 0 & 0 \\ 0 & 0 & \sin \ell\theta_2 & \cos \ell\theta_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos \ell\theta_{d/2} & -\sin \ell\theta_{d/2} \\ 0 & 0 & 0 & 0 & \cdots & \sin \ell\theta_{d/2} & \cos \ell\theta_{d/2} \end{pmatrix}$$

J. Su, Y. Lu, S. Pan, A. Murtadha, B. Wen, and Y. Liu, RoFormer: Enhanced transformer with rotary position embedding, *Neurocomputing*, 2024.

Modern transformers: RoPE

Properties of RoPE:

- The rotation operation ensures that only relative positional information influences the attention mechanism, i.e.,

$$\tilde{A}_{ij} = \frac{q_i^\intercal k_j}{\sqrt{d_K}} = f\big(v_i, v_j, (i-j)\big)$$

  for some function f.

- Long-term decay in the sense of $\tilde{A}_{ij} \to 0$ as $(i-j) \to \pm\infty$ given fixed embedding vectors x_i and x_j. This is desirable since we expect the tokens' interactions to weaken as they get farther away.

- Can be implemented very efficiently by further simplifying the formulas and using

$$\begin{pmatrix} \cos \ell\theta_i & -\sin \ell\theta_i \\ \sin \ell\theta_i & \cos \ell\theta_i \end{pmatrix} \begin{pmatrix} \cos \ell'\theta_i & -\sin \ell'\theta_i \\ \sin \ell'\theta_i & \cos \ell'\theta_i \end{pmatrix}^\intercal = \begin{pmatrix} \cos(\ell-\ell')\theta_i & -\sin(\ell-\ell')\theta_i \\ \sin(\ell-\ell')\theta_i & \cos(\ell-\ell')\theta_i \end{pmatrix}$$

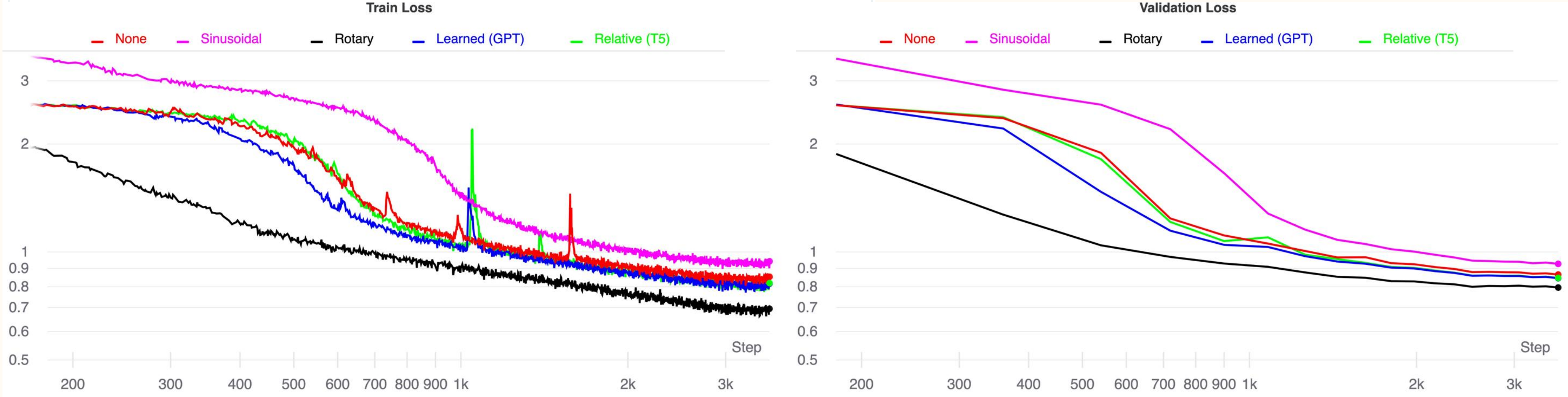
  (rotating forward by $\ell\theta_i$ and backward by $\ell'\theta_i$ = rotating forward by $(\ell-\ell')\theta_i$.)

RoPE works quite well.

Stella Biderman
@BlancheMinerva

You: Gee Stella, #EleutherAI sure hypes rotary embeddings a lot. Are you sure that they're that good?

Me:



9:36 AM · May 17, 2021

https://x.com/BlancheMinerva/status/1394089508723900422

Apply RoPE on all layers

Classical positional embeddings usually applied once, at the input layer.

- Applying positional encoding to all of the layers is arguably unnecessary, since the residual connections provide the positional encoding information to all of the layers.

In contrast, RoPE is usually applied many times, at all attention layers.

NoPE: No positional encodings

Actually, positional encodings aren't required for decoder-only attention layers. (The un-masked encoder only transformers like BERT or ViT do need positional encodings.)

Transformers without position encodings (NoPE) may work just fine.[#]

Llama 4[%] uses interleaved attention layers without positional embeddings. (Some layers are RoPE and some layers are NoPE.)

Decoder only transformers are not permutation-equivariant because of the causal mask. The layers can learn to count the number of tokens they can attend, which reveals the absolute position.

[#] A. Haviv, O. Ram, O. Press, P. Izsak, and O. Levy, Transformer language models without positional encodings still learn positional information, *EMNLP*, 2022.
[#] A. Kazemnejad, I. Padhi, K. Natesan, P. Das, and S. Reddy, The impact of positional encoding on length generalization in transformers, *NeurIPS*, 2023.
[%] https://ai.meta.com/blog/llama-4-multimodal-intelligence/

Modern transformers: KV caching

During inference, the LLM generates tokens $y_1, \dots, y_L$ sequentially.

To compute y_ℓ, we need $q_\ell, k_1, \dots, k_{\ell-1}, k_\ell, v_1, \dots, v_{\ell-1}, v_\ell$.

KV caching is an optimization that stores (caches) the previously computed $k_1, \dots, k_{\ell-1}, v_1, \dots, v_{\ell-1}$. Only k_ℓ and v_ℓ are newly computed at the ℓ-th step.

KV caching uses trades off memory usage with compute.

$$\tilde{A}_{ij} = \begin{cases} q_i^\intercal k_j / \sqrt{d_K} & \text{if } i \geq j \\ -\infty & \text{if } i < j \end{cases}$$

$$A_{ij} = \frac{e^{\tilde{A}_{ij}}}{\sum_{j'=1}^{L} e^{\tilde{A}_{ij'}}}$$

$$y_\ell = \sum_{r=1}^{\ell} A_{\ell r} v_r$$

Modern transformers: KV caching

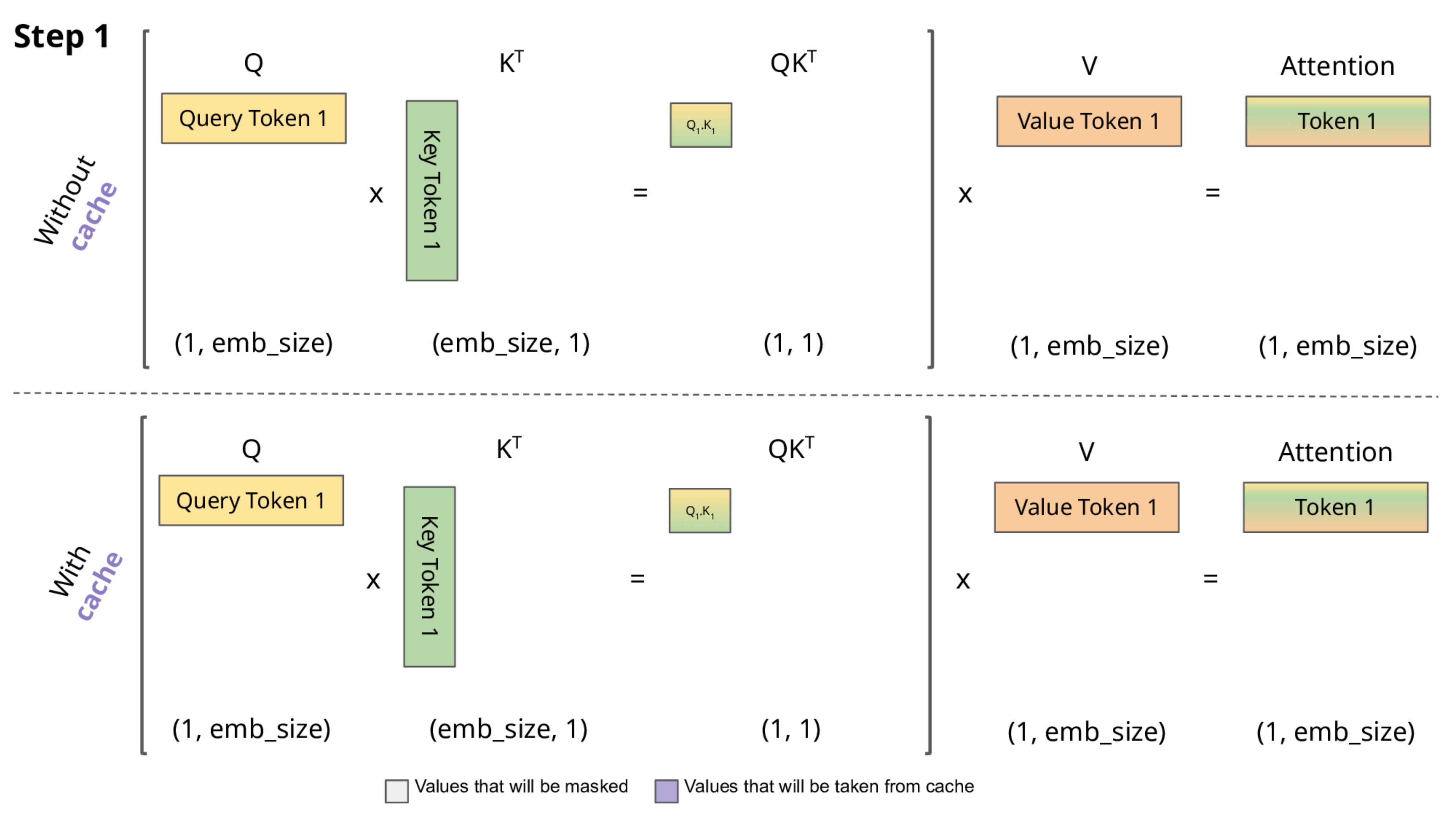


Image credit: João Lages, https://medium.com/@joaolages/kv-caching-explained-276520203249

Modern transformers: Multi-query attention (MQA)

Attention layers have H heads, so computing $y_\ell^{(1)}, \dots, y_\ell^{(H)}$ requires:

- Computing $q_\ell^{(h)}, k_\ell^{(h)}, v_\ell^{(h)}$ for $h = 1, \dots, H$. (Fast ✅)
- Loading from GPU memory $k_1^{(h)}, \dots, k_{\ell-1}^{(h)}, v_1^{(h)}, \dots, v_{\ell-1}^{(h)}$ for $h = 1, \dots, H$. (Slow ❌)

GPUs are very efficient with compute but are less efficient with memory IO.

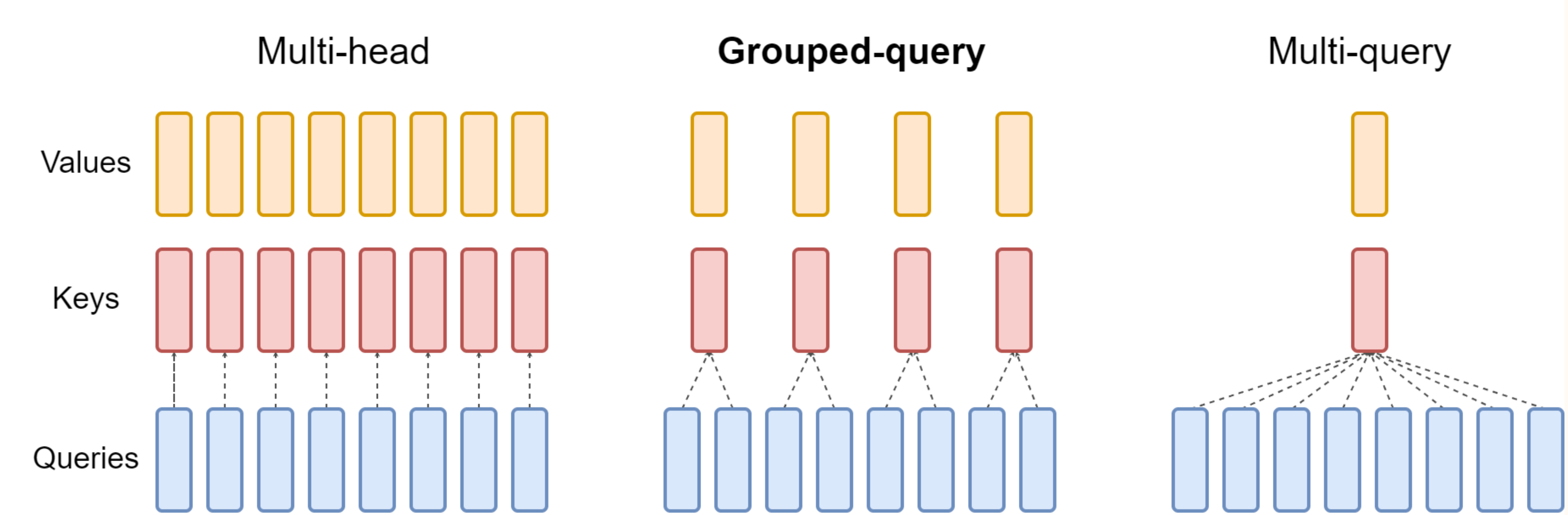
Multi-query attention (MQA) shares the keys and values across the heads. So computing $y_\ell^{(1)}, \dots, y_\ell^{(H)}$ requires:

- Computing $q_\ell^{(h)}, k_\ell, v_\ell$ for $h = 1, \dots, H$. (Fast ✅)
- Loading from GPU memory $k_1, \dots, k_{\ell-1}, v_1, \dots, v_{\ell-1}$. (IO is slow but volume is small ✅)

N. Shazeer, Fast transformer decoding: One write-head is all you need, *arXiv*, 2019.

Modern transformers: Grouped-query attention

MQA significantly reduces inference cost, albeit with a slight degradation in performance. Grouped-query attention (GQA) offers a compromise by sharing key and value vectors within groups. GQA nearly matches the performance of plain multi-head attention (MHA).



J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebron, and S. Sanghai, GQA: Training generalized multi-query transformer models from multi-head checkpoints, *EMLNP*, 2023.

How to decoding with causal LM

Assume a causal language model $p_\theta(u_\ell | u_1, \dots, u_{\ell-1})$ has been trained. If p_θ were perfect, then naïve sampling (as defined soon) should be sufficient for text generation (decoding). However, p_θ is imperfect, so effective sampling requires the following techniques:

- Naïve sampling (with temperature)
- Greedy sampling
- Beam search
- Top-k sampling
- Top-p (nucleus) sampling

Naïve sampling

Let $f_\theta(u_1, \dots, u_\ell) \in \mathbb{R}^n$ be the final ℓ-th output token of a decoder-only transformer architecture (n is the number of distinct tokens) such that

$$p_\theta(u_{\ell+1} = i | u_1, \dots, u_\ell) = \Big(\mathrm{softmax}\big(f_\theta(u_1, \dots, u_\ell)\big)\Big)_i$$

for $i = 1, \dots, n$. Let $u_1, \dots, u_\ell$ be given and $u_\ell \neq$ <EOS>.

Naïve sampling continues the text with

$$u_s \sim p_\theta(\cdot\, | u_1, \dots, u_{s-1})$$

for $s = \ell + 1, \ell + 2, \dots$ until $u_s =$ <EOS> (and sets $L = s - 1$).

Problem: Low-probability words are sampled with low but non-zero probabilities, and they tend to be bad. The imperfections of p_θ manifest in these low-probability words.

Naïve sampling with temperature β

Naïve sampling with temperature adjusts the “temperature” parameter of the softmax

$$p_\theta(u_{\ell+1} = i | u_1, \dots, u_\ell) = \left(\mathrm{softmax}\left(\frac{f_\theta(u_1, \dots, u_\ell)}{\beta} \right) \right)_i$$

for $i = 1, \dots, n$, where $\beta > 0$. Note, $\beta = 1$ is regular naïve sampling. With $\beta < 1$, we suppress the likelihood of the low-probability words. In modern LLMs, $\beta = 0.7$ or $\beta = 1$ are common default choices.

In the limit of $\beta \to 0$, we recover greedy sampling.

Problem: Even with $\beta < 1$, low-probability words are still sampled.

Greedy sampling

If the low probability words are problematic, then why not just sample the most likely word?

Greedy sampling continues the text with

$$u_s = \underset{u_s=1,\ldots,n}{\operatorname{argmax}}\, p_\theta(u_s | u_1, \ldots, u_{s-1})$$

for $s = \ell + 1, \ell + 2,$ ... until $u_s =$ <EOS>. (Ties in the argmax are broken arbitrarily.)

Problem 1) Greedy sampling does *not* generate the most likely sequence tokens.

Problem 2) Likely text is not always good. More on this later.

Exact MAP decoding

Exact maximum a posteriori (MAP) decoding is the globally optimal (most likely) generation

$$(u_{\ell+1}, \dots, u_L = \text{<EOS>}) = \underset{u_{\ell+1}, u_{\ell+2}, \dots}{\operatorname{argmax}} \sum_{s=\ell+1}^{L} \log p_\theta(u_s | u_1, \dots, u_{s-1})$$

Problem 1) Not implementable. Exponentially many candidate sequences. Computing exact MAP decoding is intractable (NP-Hard).[#]

Problem 2) Likely text is not always good. More on this later.

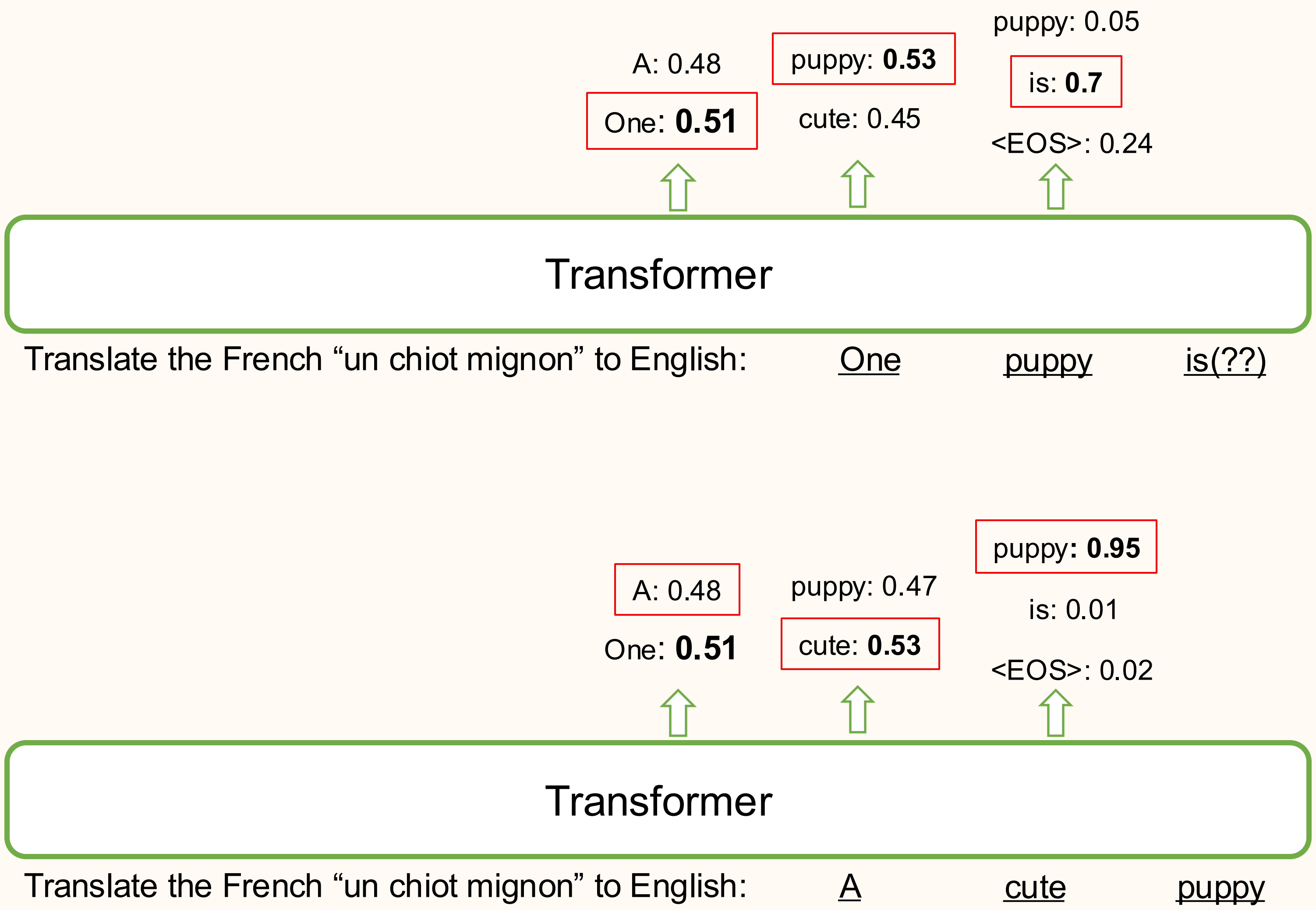
[#]K. Knight, Decoding complexity in word-replacement translation models, *Computational Linguistics*, 1999.

Why greedy ≠ MAP?

MAP maximizes the product of the probabilities (= sum of log-probabilities). Greedy commits to the most likely individual word one word at a time.

You want to see how the completion pans out before committing to a word.

Sometimes, by committing to the most likely token early on, you eliminate better completion paths.



Beam search decoding

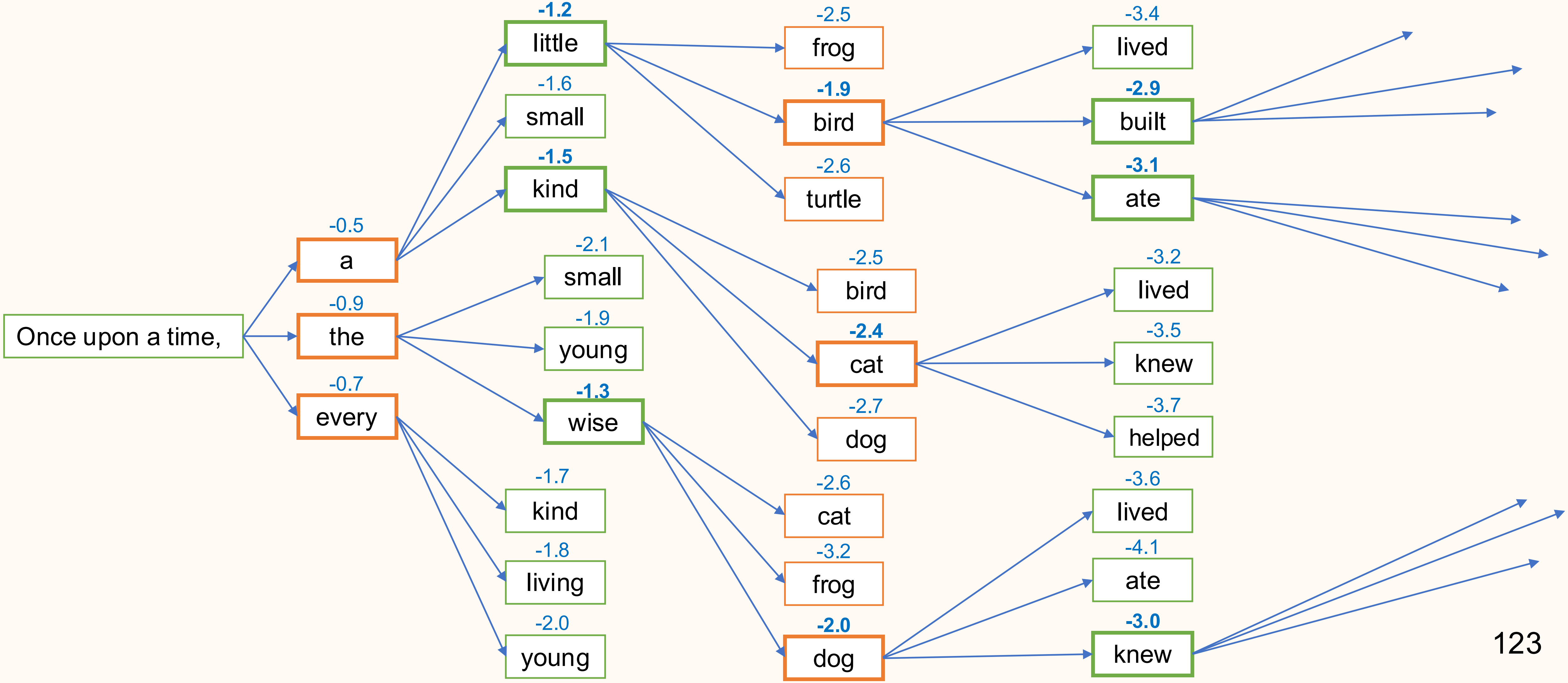
Beam search is a tractable, heuristic approximation to exact MAP decoding that produces sequences with higher likelihood compared to greedy search.

Intuition: While choosing highest probability word on the first step may not be optimal, choosing a very low probability word is very unlikely to lead to a good result. (We shouldn't be fully greedy, but we can be somewhat greedy.)

Idea: On each step of the decoder, keep track of the k most probable partial generations. k is also called the *beam size* and $k = 5$ or $k = 10$ are common values.

Beam search decoding: Example

Beam size k = 3. Blue numbers : $\text{score}(y_1, \dots, y_t) = \sum_i \log P_{LM}(y_i | y_1, \dots, y_{i-1}, x)$.



Beam search decoding: Pseudocode

$$\log p(u_{i,1}, \ldots, u_{i,\ell}) = \sum_{t=1}^{\ell} \log p(u_{i,t} \mid u_{i,1}, \ldots, u_{i,t-1}), \qquad \text{for } i = 1, \ldots, k$$

For each sequence step ℓ:

1. For each $\{u_{i,t}\}_{t=1}^{\ell}$ that we are tracking for $i = 1, \ldots, k$, find the top k tokens $u_{i,\ell+1}^{(1)}, \ldots, u_{i,\ell+1}^{(k)}$.
2. Sort the resulting k^2 sequences (of length $\ell + 1$) by their total log-probability.
3. End loop if a generation encounters <EOS> and return the completed generation.
4. Keep the top k.

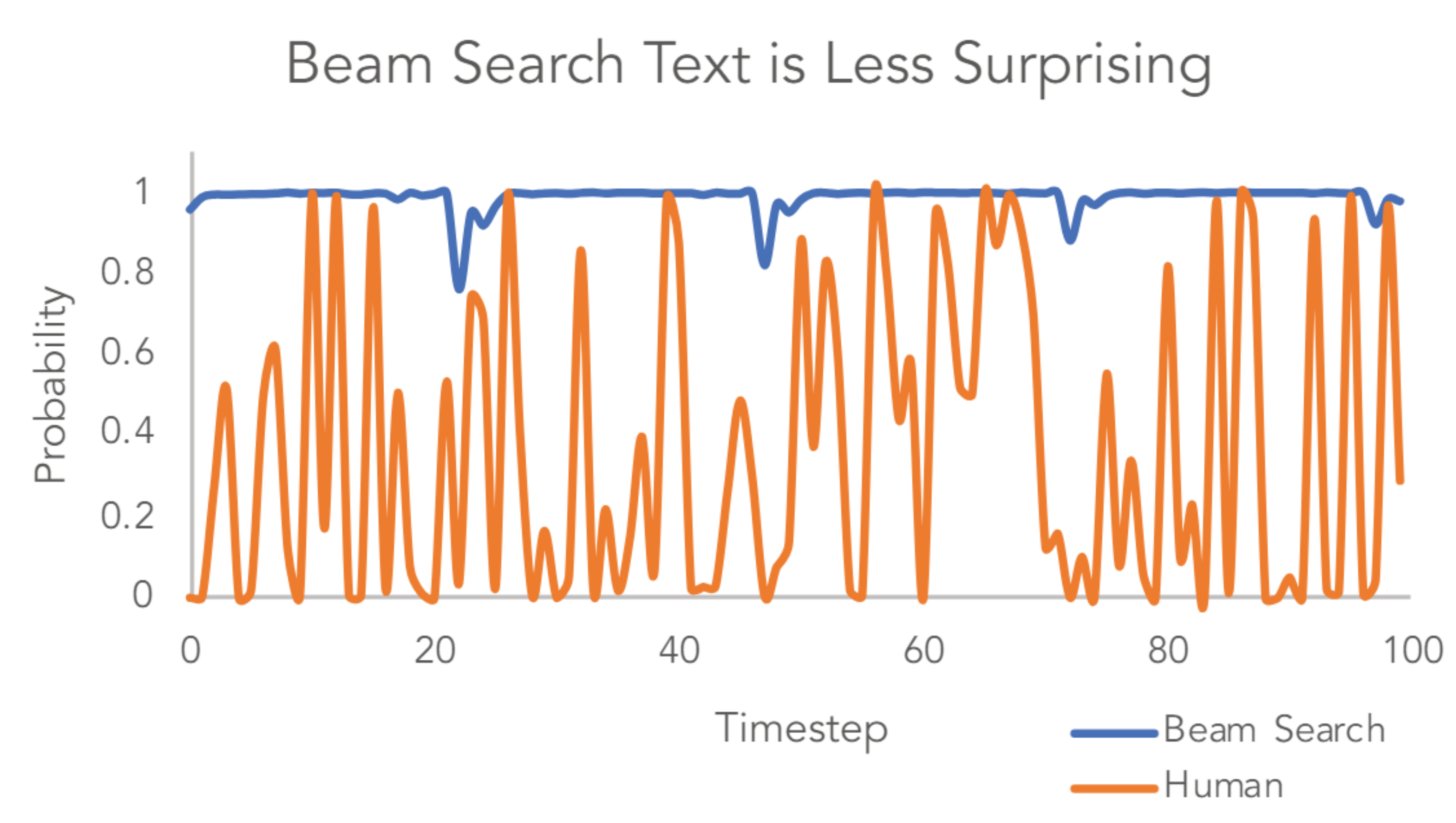
When $k = 1$, beam search reduces to greedy search.

Human text is not too predictable

High quality human text does not follow a distribution of high probability next words.

Beam search avoids generation with elements of surprise and suffers from repetitive and uninteresting content.

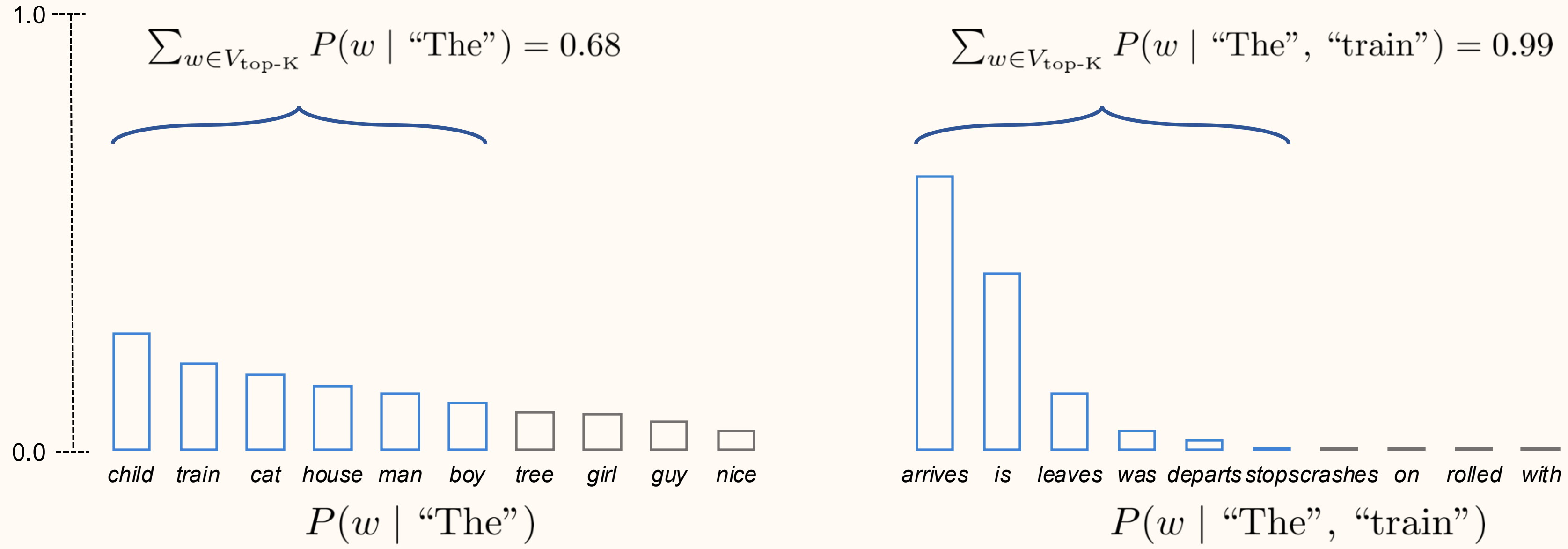
Beam search is no longer used in mainstream LLMs. As models became better, the lower probability tokens became better.



A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, The curious case of neural text degeneration, *ICLR*, 2020.

Top-K sampling

Top-K sampling samples among the K most likely word, with probabilities determined by softmax applied to the top K words with a temperature $\beta > 0$.



($K = 6$ in this example)

$K \sim 50$ is a is a common default choice. $K = \infty$ means top-K is not used.

Problem) Low-probability words can still be sampled.

A. Fan, M. Lewis, and Y. Dauphin, Hierarchical neural story generation, *ACL*, 2018.

Top-p (nucleus) sampling

Top-p sampling samples among the fewest most likely words such that their probabilities (with a temperature $\beta > 0$) exceeds probability p. Once the set top words $V^{(p)}$ are defined

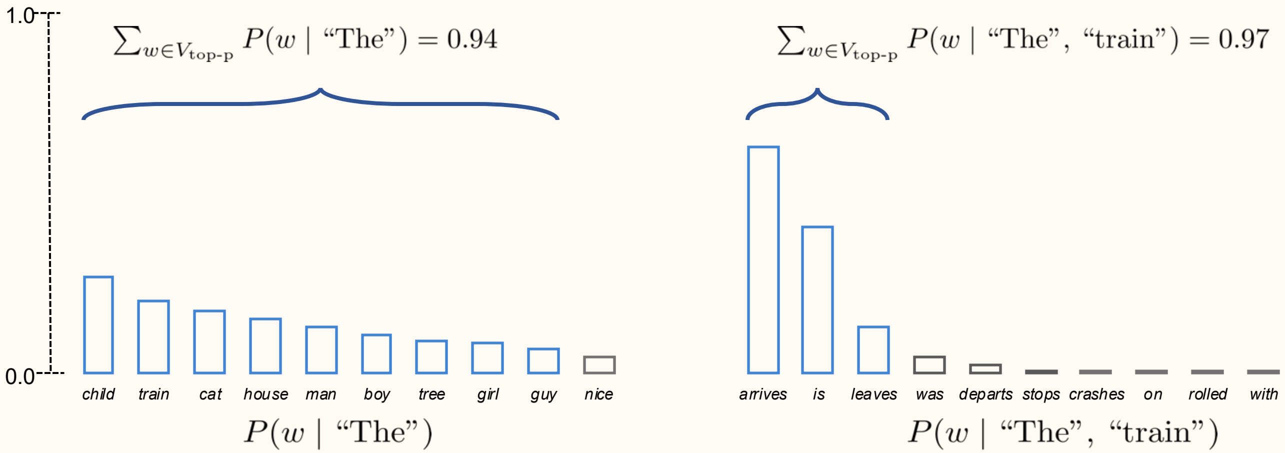
$$\tilde{p}_\theta(u_{\ell+1} = i | u_1, \dots, u_\ell) = \frac{p_\theta(u_{\ell+1} = i | u_1, \dots, u_\ell)}{\sum_{i \in V^{(p)}} p_\theta(u_{\ell+1} = i | u_1, \dots, u_\ell)}$$

sampling is done from $\tilde{p}_\theta$. (So $\sum_{i \in V^{(p)}} p_\theta(u_{\ell+1} = i | u_1, \dots, u_\ell) \geq p$.)

($p = 0.9$ in this example)

$p \sim 0.9$ is a common default choice.

$p = 1$ means top-p is not used.



A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, The curious case of neural text degeneration, *ICLR*, 2020.

Combining β, top-K, and top-p

Most modern LLMs combine the temperature parameter β, top-K, and top-p.

Precisely how these are combined slightly differ from model to model.

LLMs as a universal interface

As LLMs were scaled up, researchers started to notice a crucial capability of LLMs emerge: LLMs can just follow textual instructions.

Prior paradigm: Labeled data defines the task. This was the case prior to (sufficiently large) language models. Self-supervised pre-training on large unlabeled text would improve the efficiency, but labeled data was still needed to define the task.

New paradigm: Define the task with a natural language description. Researchers gradually crystalized this paradigm through GPT-2, GPT-3, T5, FLAN, and Flan-PaLM.

GPT-2 and GPT-3

The GPT-2 and GPT-3 papers had very similar titles and messages. GPT-3 simply scales up GPT-2 and achieves stronger results. (Architecture didn't change much from GPT-1.)

- GPT-(1,2,3) Model size: 117M→1.5B→175B
- GPT-(1,2,3) Data size: 4GB→40GB→600GB

Main message: GPT can solve many tasks with a unified task-agnostic architecture and without supervised fine-tuning. Task-specific training data is not used (zero-shot) or only a few is used during inference (few-shot in-context learning).

No task-specific training data is used for training or fine-tuning. (However, having diverse task-specific training data is helpful, as T5 and Flan-T5 shows.)

(ELMo was not at all task agnostic. BERT and GPT-1 was a little more task agnostic but had task-specific heads.)

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, Language models are unsupervised multitask learners, *Tech. Report*, Feb. 2019.
T. B. Brown, B. Mann, N. Ryder, M. Subbiah …, A. Radford, I. Sutskever, and D. Amodei, Language models are few-shot learners, *NeurIPS*, 2020. (*arXiv* May 2020)

Zero-shot and few-shot learning

Few-shot learning refers to a model learning a behavior with a few labeled samples or demonstrations. Classically, few-shot learning would have a pre-trained model fine-tuned (gradient updates) on the few labeled data points.

Zero-shot learning would mean performing a task with no demonstrations. How is this possible?

Zero-shot summarization with GPT-2:

“To induce summarization behavior, we add the text TL;DR: after the article and generate 100 tokens with Top-k random sampling ….”

GPT-2 can perform summarization without any explicitly paired summarization dataset.

A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, Language models are unsupervised multitask learners, *Tech. Report*, Feb. 2019.

In-context learning (ICL)

In In-context learning (ICL), first explicitly reported with GPT-3#, the model discerns the task implied by the context of demonstrations with or without explicit descriptions.

	Standard Next-word Prediction	In-Context Learning (ICL)	ICL with arbitrary rules
Prompt:	The quick fox jumps on the lazy ____.	(rose : red) (sunflower : yellow) (lavender : ____)	(xy, yz, @, xyyzyz) (ab, cd, @, abcdcd) (mn, op, @, _____).
Completion:	The quick fox jumps on the lazy dog.	(rose : red) (sunflower : yellow) (lavender : purple)	(xy, yz, @, xyyzyz) (ab, cd, @, abcdcd) (mn, op, @, mnopop).

#T. B. Brown, B. Mann, N. Ryder, M. Subbiah …, A. Radford, I. Sutskever, and D. Amodei, Language models are few-shot learners, *NeurIPS*, 2020. (*arXiv* May 2020)

ICL example

ChatGPT o1-preview

Banana -> Manzana
Car -> Bicicleta
Grass -> Árbol
Coffee ->

Based on the pattern you've provided:

- **Banana** (an English word for a fruit) corresponds to **Manzana** (Spanish for "apple").
- **Car** (an English word for a vehicle) corresponds to **Bicicleta** (Spanish for "bicycle").
- **Grass** (an English word for a plant) corresponds to **Árbol** (Spanish for "tree").

In each case, the English word is a type of item (fruit, vehicle, plant), and the Spanish word is a different item in the same category.

Following this pattern, **Coffee** (an English word for a beverage) should correspond to the Spanish word for a different beverage. A common alternative beverage is **tea**, which is **"Té"** in Spanish.

Answer: Té

In-context learning (ICL)

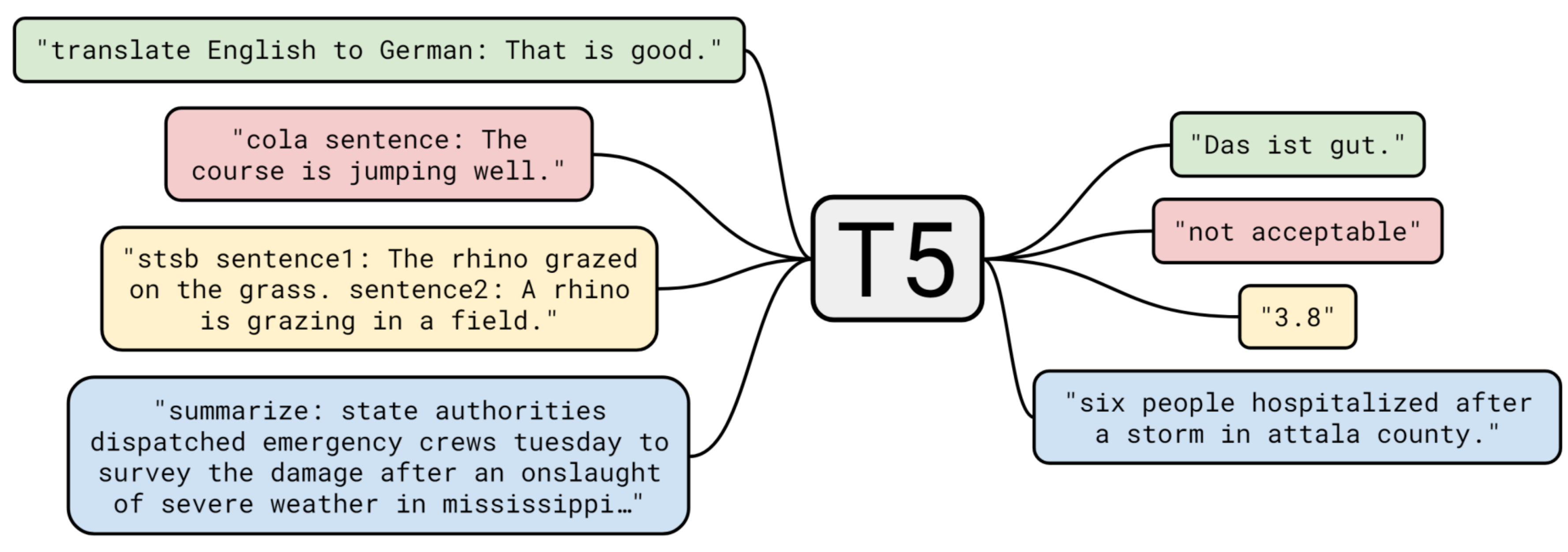
GPT-2# had hints of in-context learning capabilities: “Similar to translation, the context of the language model is seeded with example question answer pairs which helps the model infer the short answer style of the dataset.”

The term *in-context learning* refers to the LLM’s ability to learn the user’s intent through examples within the context of the text and <u>without any parameter updates</u>. You show a handful of demonstrations (labeled datapoints) and ask the model to “follow the examples”. This is few-shot learning with no gradient updates.

ICL is a crucial capability, since most natural language instructions do not specificy all details with complete precision. It is important that models understand what you mean through examples.

#A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, Language models are unsupervised multitask learners, *Tech. Report*, Feb. 2019.

T5 model



Text-to-text-transfer-transformer (T5) uses an encoder-decoder transformer and formats all pre-training and fine-tuning into a text-to-text format.

Unified task-agnostic architecture. The many tasks, which are not semantically related, are formatted into a text-to-text format. Same model, objective, training procedure and decoding process to every task that we consider.

C. Raffel, N. Shazeer, A. Roberts, K .Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, *JMLR*, 2020. (*arXiv* Oct. 2019)

T5 pre-training

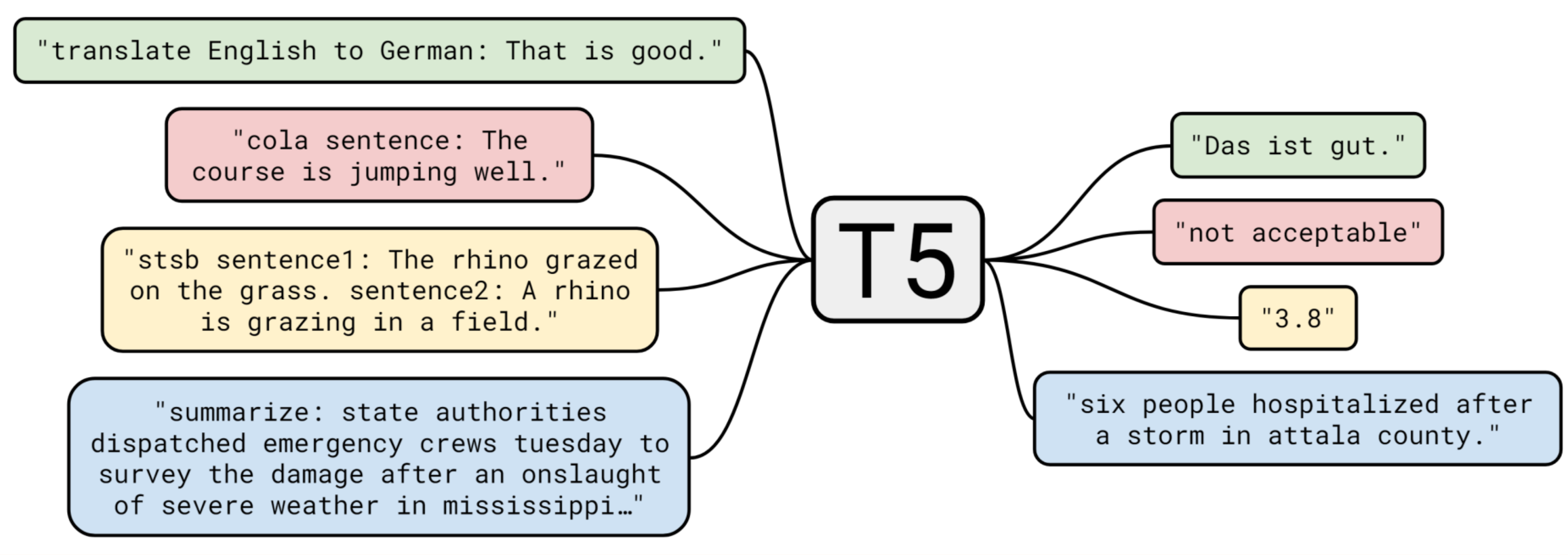
Pre-training on large unlabeled text with diverse objectives inspired by prior work.

Objective	Inputs	Targets
Prefix language modeling	Thank you for inviting	me to your party last week .
BERT-style Devlin et al. (2018)	Thank you <M> <M> me to your party apple week .	*(original text)*
Deshuffling	party me for your to . last fun you inviting week Thank	*(original text)*
MASS-style Song et al. (2019)	Thank you <M> <M> me to your party <M> week .	*(original text)*
I.i.d. noise, replace spans	Thank you <X> me to your party <Y> week .	<X> for inviting <Y> last <Z>
I.i.d. noise, drop tokens	Thank you me to your party week .	for inviting last
Random spans	Thank you <X> to <Y> week .	<X> for inviting me <Y> your party last <Z>

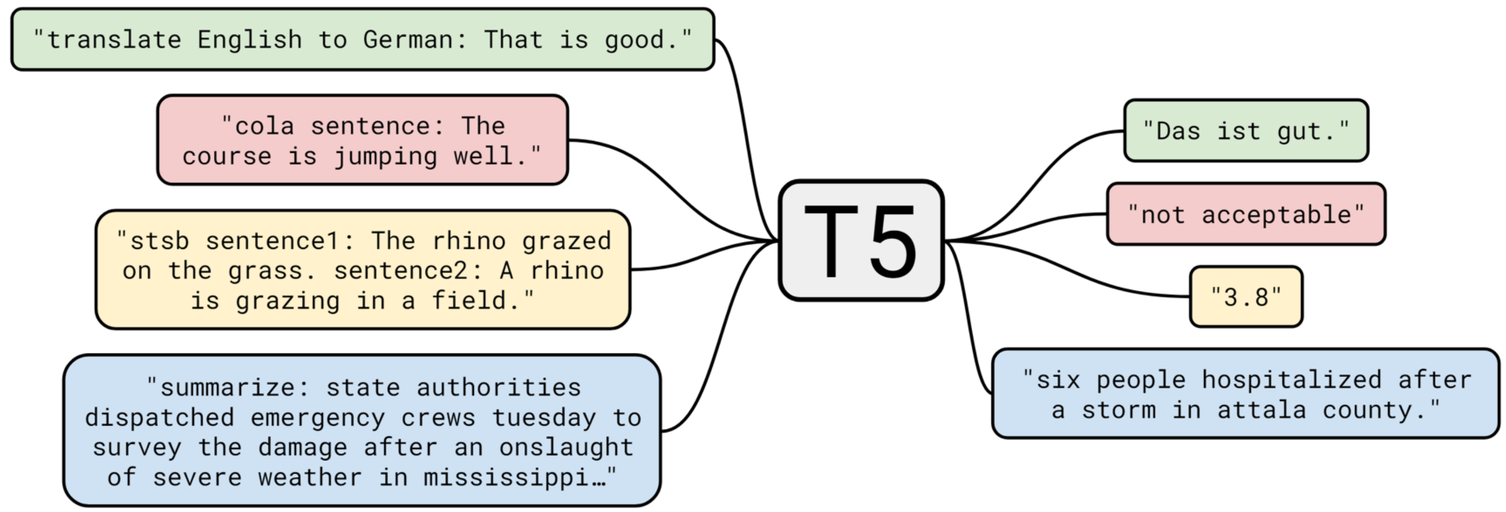
The “inputs” are fed into the encoder block while the “target” text is generated by the decoder one token at a time.

T5 fine-tuning

Simultaneously fine-tune on a wide range of tasks. Simply prompt the model differently for each task to inform T5 of the specific task to solve.



T5 contribution



Advanced state-of-the-art with the pre-train-than-fine-tune approach.

Further demonstrated the idea that language models can understand and respond to natural language instructions. We can simply tell a language model what we want and it will follow our instructions.

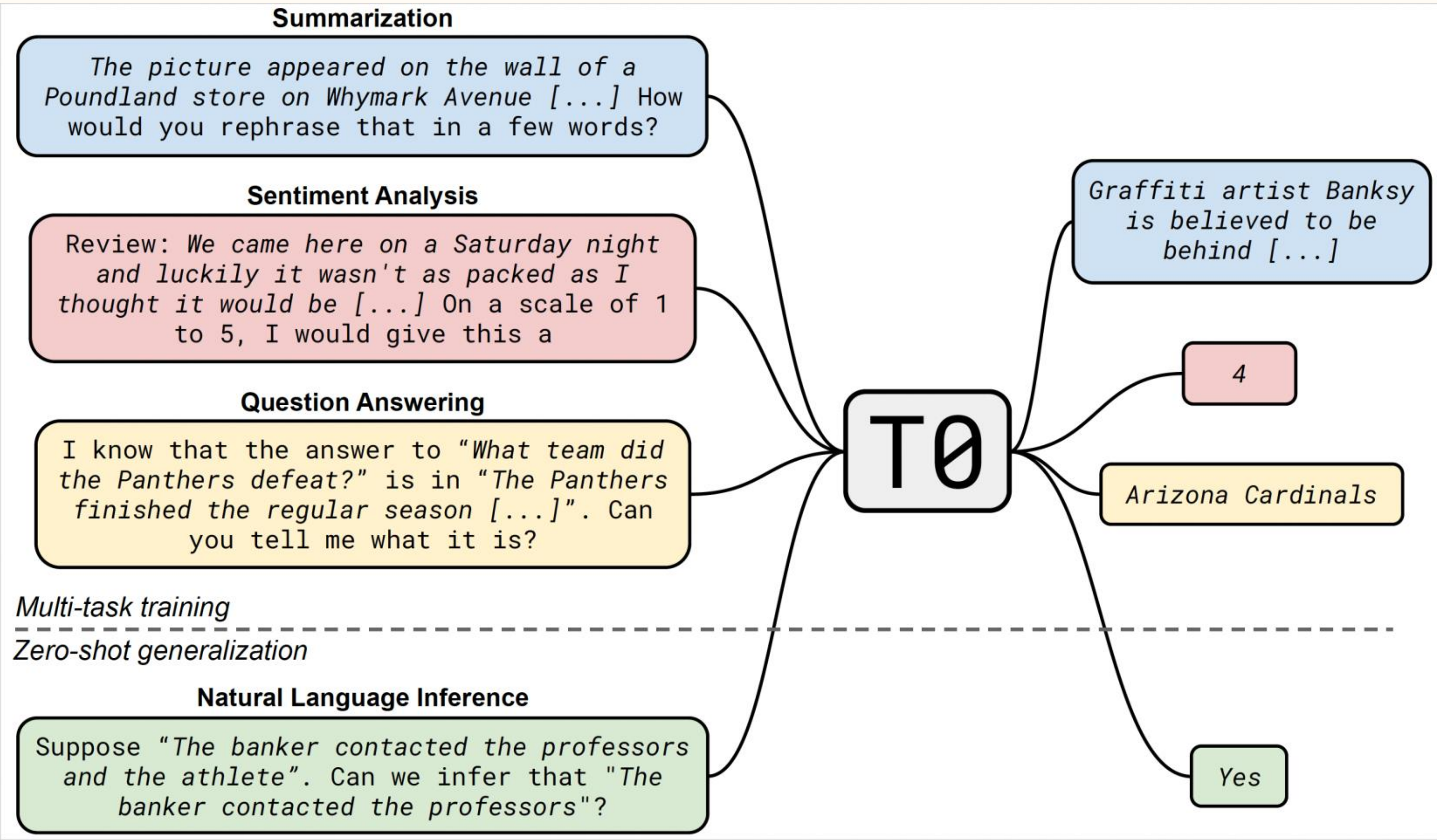
Problem: The prompts were unnatural as they did not fully describe the task at hand. (What does "stsb sentence 1" mean?) It is a half-way measure between a fully arbitrary label (like "task 3A") and a full natural-language description.

C. Raffel, N. Shazeer, A. Roberts, K .Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, Exploring the limits of transfer learning with a unified text-to-text transformer, *JMLR*, 2020. (*arXiv* Oct. 2019)

Instruction fine-tuning

Instruction fine-tuning, presented in the FLAN[#] and T0[*] papers, fine-tunes a pre-trained model on a collection of datasets described via natural-language instructions.

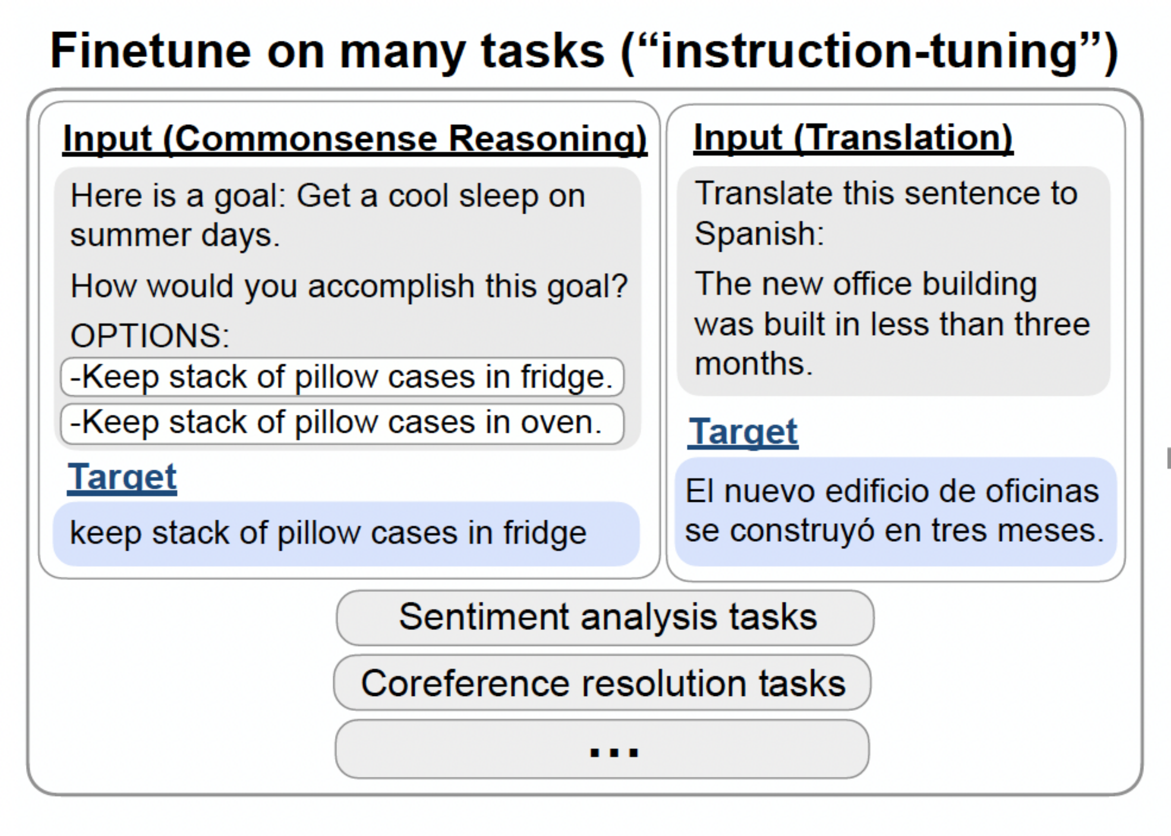
Unification: All tasks are sub-tasks of the meta task of following the natural language instruction.



[#] J. Wei, M. Bosma, V. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, Q. V. Le, Finetuned language models are zero-shot learners, *ICLR*, 2022. (*arXiv* Sept. 2021)
[*] V. Sanh, A. Webson, C. Raffel, S. H. Bach, …, Alexander M. Rush, Multitask prompted training enables zero-shot task generalization, *ICLR*, 2022. (*arXiv* Oct. 2021)

Instruction fine-tuning

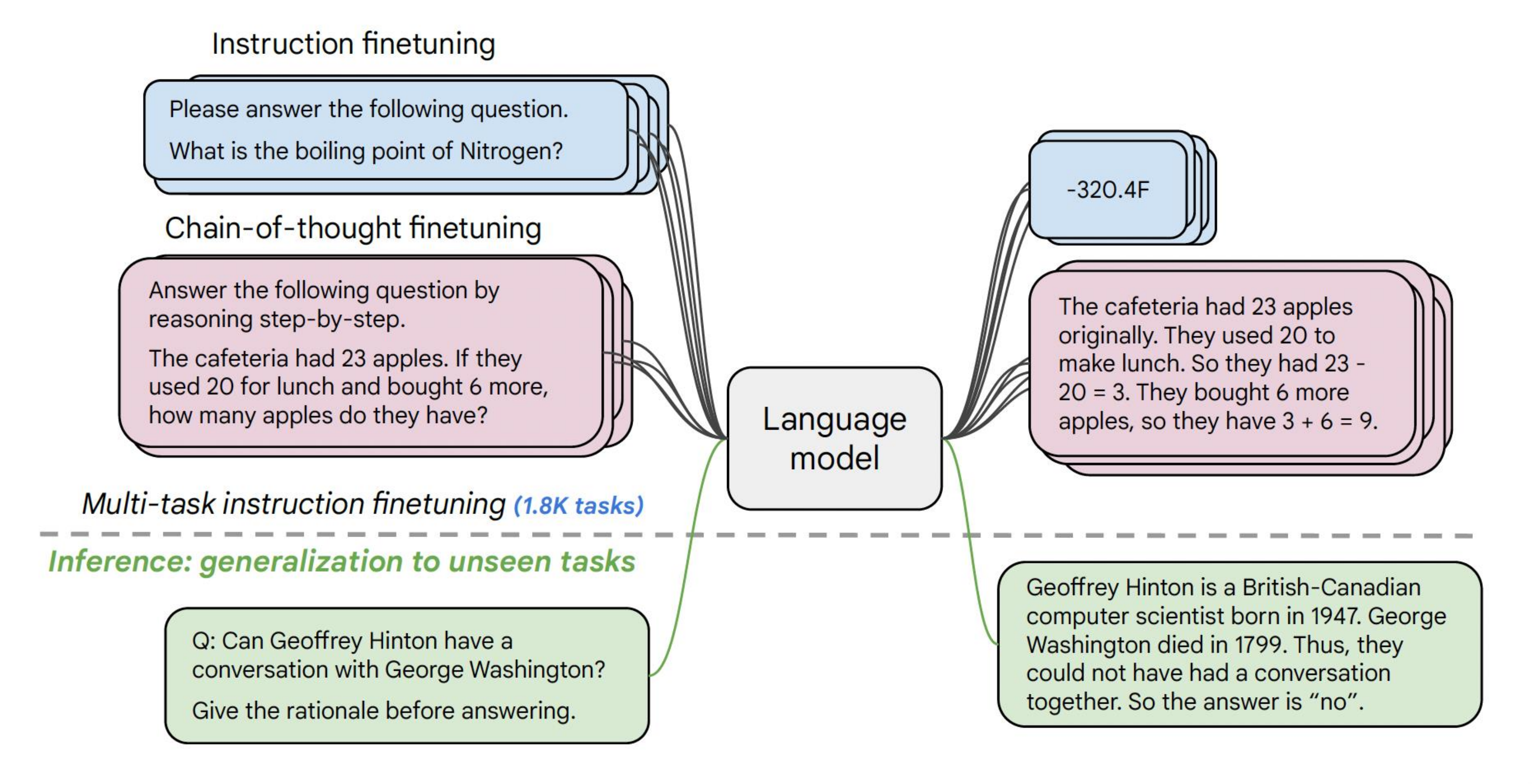
FLAN is a 137B parameter model instruction fine-tuned on over 60 NLP datasets with instructions verbalized via natural language instruction templates.



J. Wei, M. Bosma, V. Zhao, K. Guu, A. W. Yu, B. Lester, N. Du, A. M. Dai, Q. V. Le, Finetuned language models are zero-shot learners, *ICLR*, 2022. (*arXiv* Sept. 2021)

Scaling instruction fine-tuning

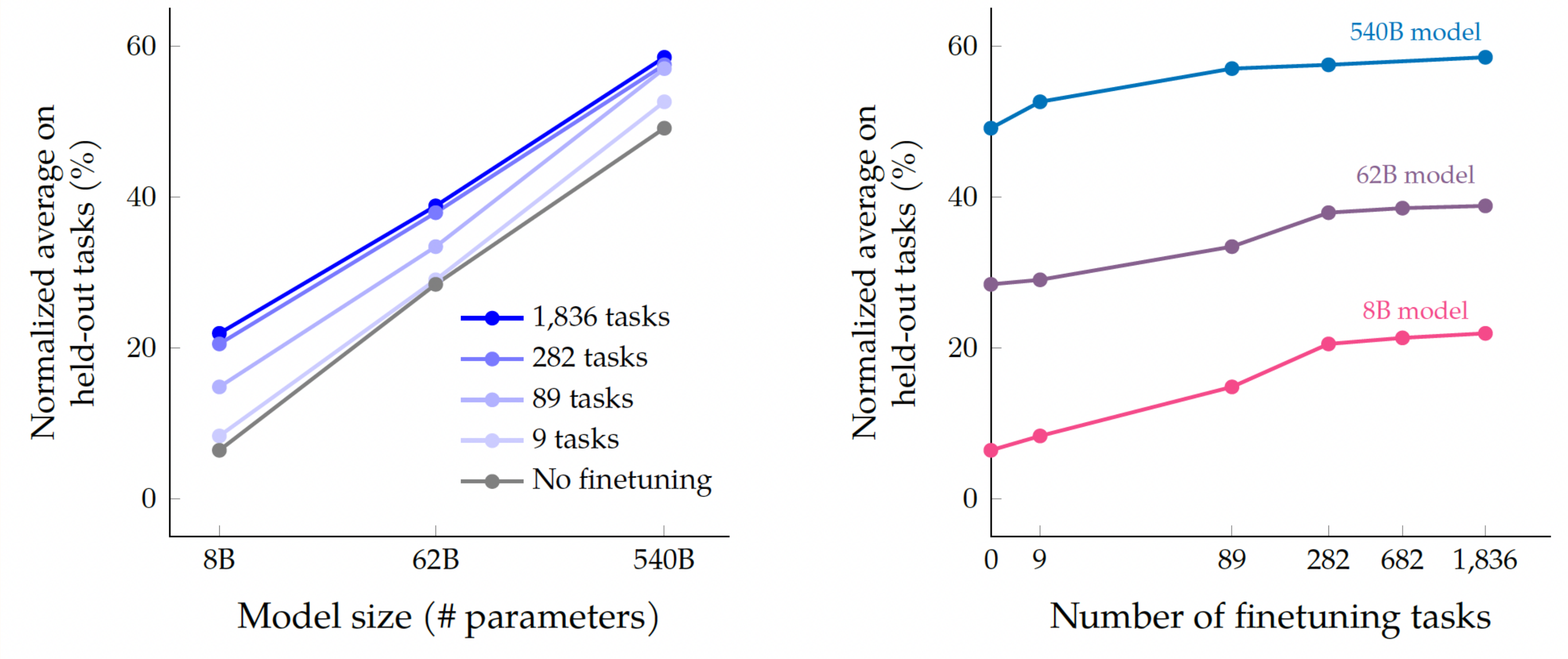
Flan-PaLM scales instruction fine-tuning up to a 540B model with 1836 instruction-finetuning tasks.



H. W. Chung, L. Hou, S. Longpre, … Jason Wei, Scaling instruction-finetuned language models, JMLR, 2024. (arXiv Oct. 2022)

Scaling instruction fine-tuning

Key finding: Task diversity is essential not just in having the model be multi-task, but also in benefiting the individual task performances. Training on tasks A, B, C, … improved performance on task A.



H. W. Chung, L. Hou, S. Longpre, … Jason Wei, Scaling instruction-finetuned language models, JMLR, 2024. (arXiv Oct. 2022)

Adding special tokens

During pre-training, LLMs are trained with some special tokens such as `<|endoftext|>`.

During instruction fine-tuning, it is important to add additional special tokens such as `<|im_start|>` and `<|im_end|>`, which stand for “instruction message start/end”.

This is done by increasing the vocabulary size and adding new columns to the token embedding matrix A, which also used in the output projection step when weight tying is used. The new columns are randomly initialized.

Although the data size and the number of updates during fine-tuning is much smaller than that of pre-training, LLMs can learn to use special tokens relatively quickly.

Chat template with special tokens

The special tokens `<|im_start|>` and `<|im_end|>` are used to format the conversation in a chat format. This trick significantly improve the LLMs ability to follow instructions and engage in turn-based conversations. (This is not optional.)

```
<|im_start|>system
You are a friendly chatbot who always responds in the style of a pirate<|im_end|>
<|im_start|>user
How many helicopters can a human eat in one sitting?<|im_end|>
<|im_start|>assistant
Oh just 6.<|im_end|>
<|im_start|>user
Are you sure about that?<|im_end|>
<|im_start|>assistant …
```

Before and after instruction fine-tuning

User Prompt:

Q: What is the capital of France?

Response without instruction fine-tuning:	Response after instruction fine-tuning:
Q: What is the currency of France?	A: The capital of France is Paris.
Q: What is the population of France?	
Q: What is the official language of France?	

Bitter Lesson II

“A counterintuitive implication of scale: trying to solve a more general version of the problem is an easier way to solve the original problem than directly tackling it.

Attempting a more general problem encourages you to come up with a more general and simpler approach. This often leads to a more scalable method. By leveraging increasingly cheaper compute, you solve the specific problem as a by-product of tackling a more general one.

Some examples:
- Directly solving NLU tasks (e.g. question answering) vs. learning a general language model and solving the task as a next token prediction.
- Instead of directly working on machine translation, work on a general problem of learning all languages (mT5 vs. translation-specific models).”

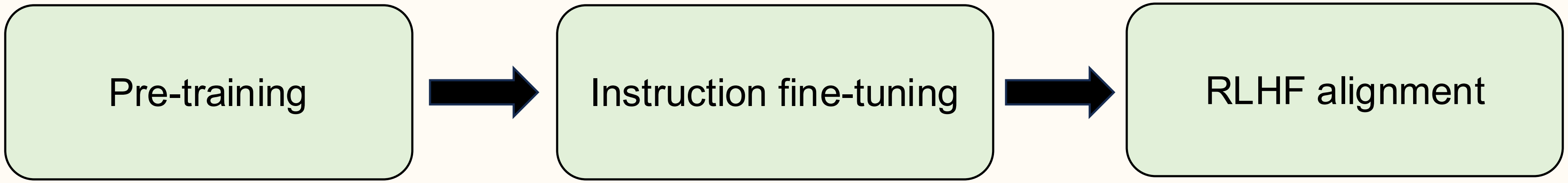
— Hyung Won Chung —

October 11, 2023

https://twitter.com/hwchung27/status/1712209280529727705

3-stage training of LLMs

Gradually, researchers have adopted a three-stage training process for LLMs.
(Although this paradigm is now evolving with the advent of mid-training.)



1. Pre-traigning produces model with base capabilties, but the model just tries to complete text and babble on. Model does not have the propensity to follow instructions or be helpful. (Pre-training is the most compute-heavy.)

2. (Part of post-training) Instruction fine-tuning induces the model to follow instructions and be "helpful." Model can engage in chat-bot-style dialogue after instruction fine-tuning.

3. (Part of post-training) RLHF further aligns LLM with human values and expectations.