Natural Language Processing

Generative AI and Foundation Models

Spring 2024

Department of Mathematical Sciences

Ernest K. Ryu

Seoul National University

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: Review 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.

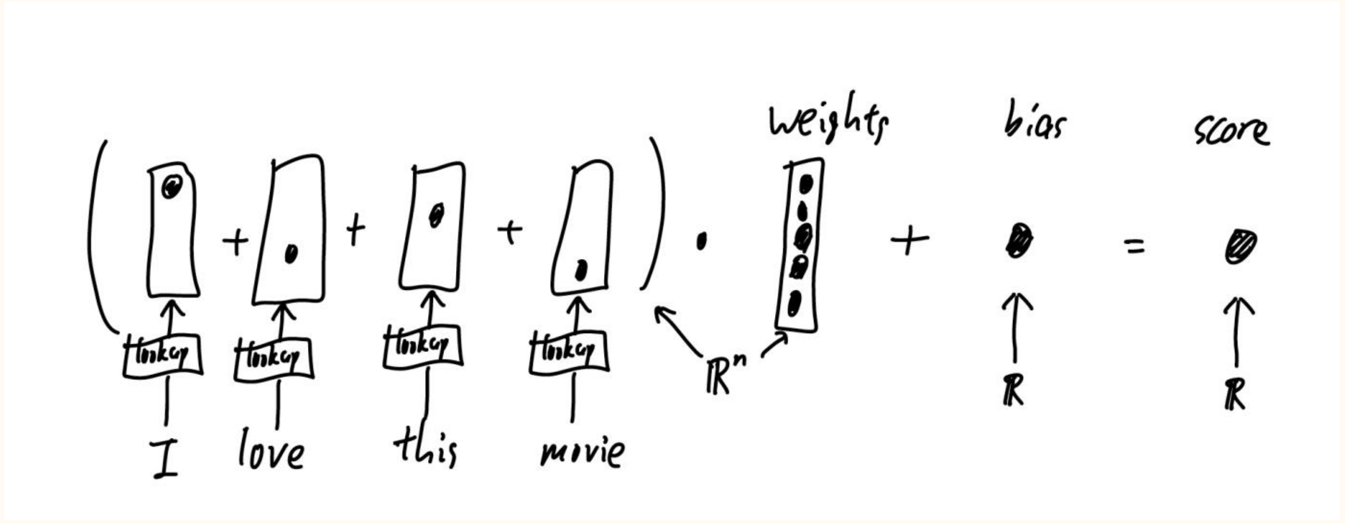
Review: I hate this movie
Sentiment: Negative

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.

Tokenization

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

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

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

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

This doesn’t work very well.

We want distinct tokens to be vectors of distinct directions.

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) Why tokenizers?

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

Q) Why can’t we map characters or words to integers? Why map to one-hot-vectors?

A) We want different words or things to map to different vectors. Vectors can represent differences through different directions and magnitudes, while scalars are far more restrictive as they can only use magnitudes to represent differences.

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’ $\approx$ ‘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.

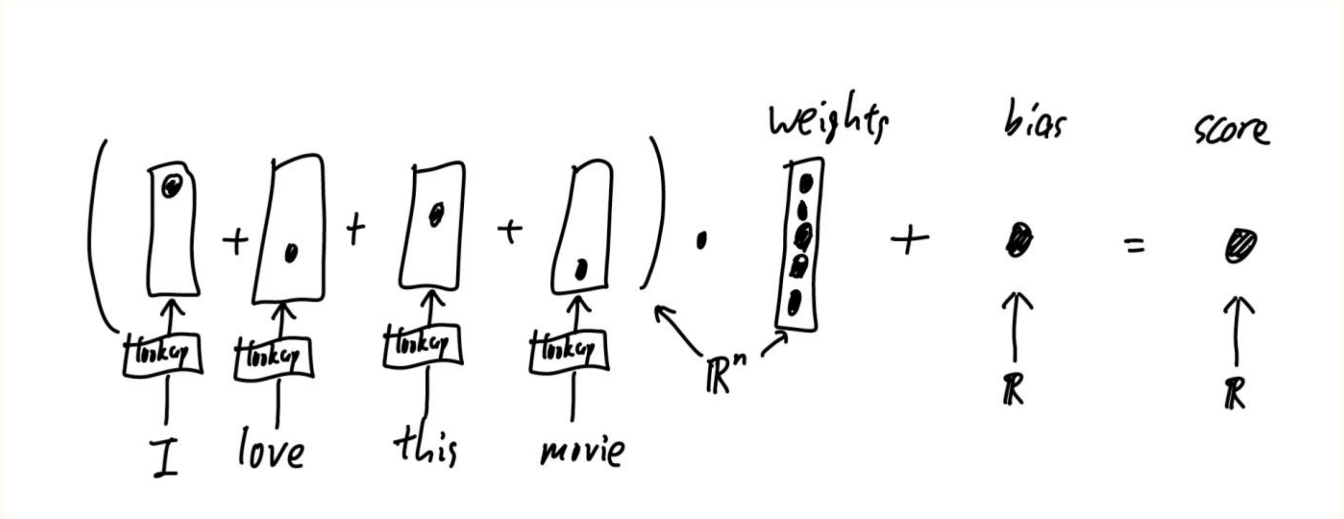
Basic BOW implementation

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

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

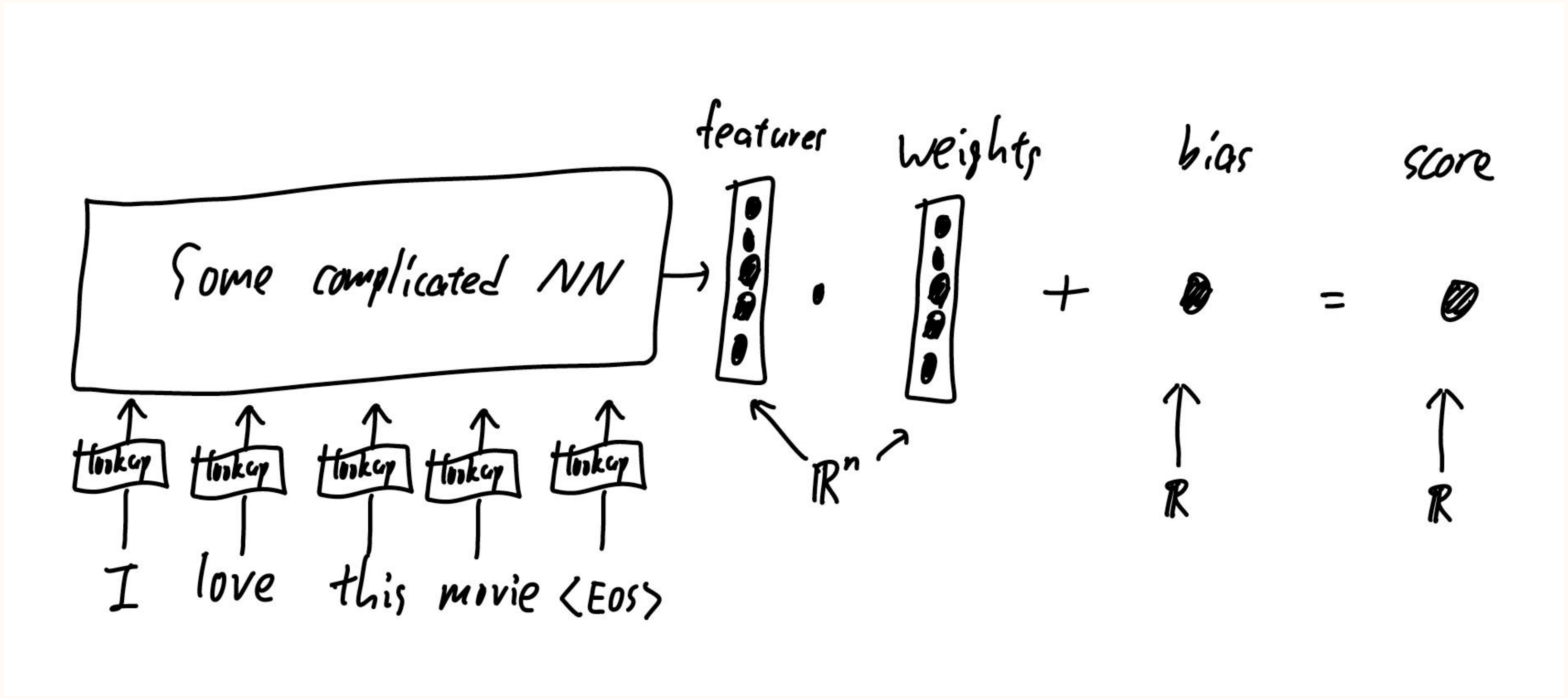
$$f_\theta(X) = b + a \cdot \sum_{\ell=1}^{L} \tau(w_\ell) = b + a \cdot \sum_{\ell=1}^{L} \big(\tau(X)\big)_\ell$$

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



Sentiment analysis with DNN

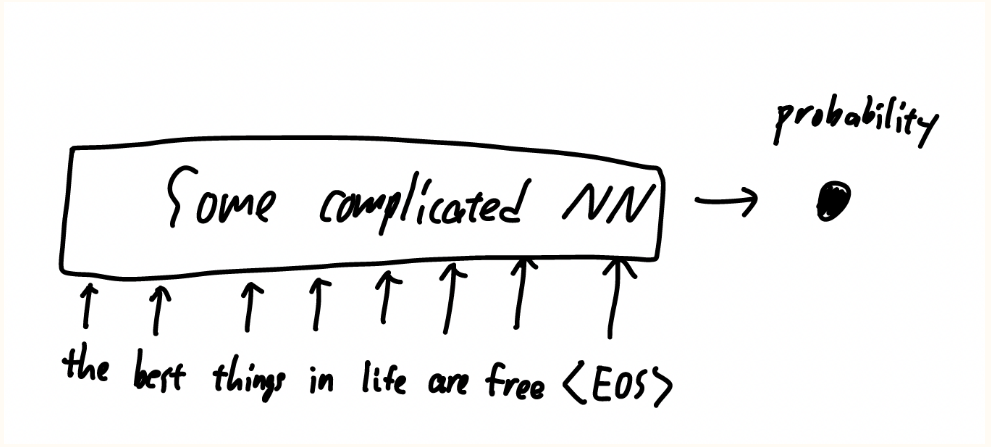
Modern state-of-the-art NLP methods are based on deep neural networks (DNN).



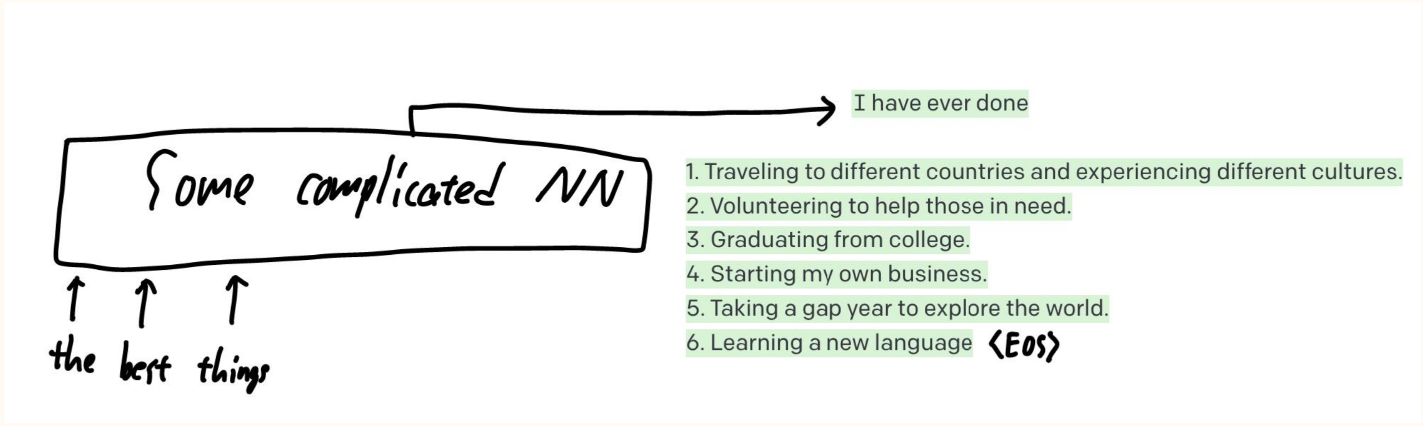
Task: Language model (LM)

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

Goal 1: Assign probabilities/likelihoods to sentences.



Goal 2: Generate coherent (likely) sentences.


Some complicated NN
the best things
I have ever done
1. Traveling to different countries and experiencing different cultures.
2. Volunteering to help those in need.
3. Graduating from college.
4. Starting my own business.
5. Taking a gap year to explore the world.
6. Learning a new language <EOS>

(This definition excluded encoder-only transformer models such as BERT from language models, 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” (✓)

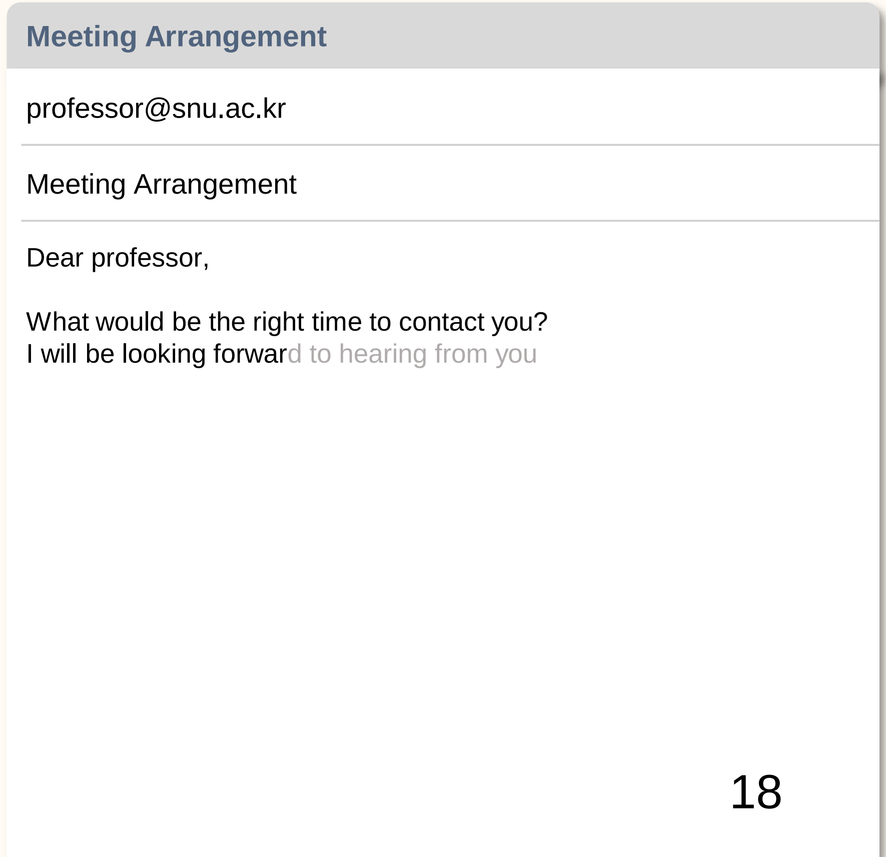
“he was a light housekeeper”

A similar application with spelling correction.

Applications of LM: Autocomplete

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

(Note to self:
Replace image with
our own to avoid
copyright issues.)



Meeting Arrangement

professor@snu.ac.kr

Meeting Arrangement

Dear professor,

What would be the right time to contact you?
I will be looking forward to hearing from you

Applications of LM: SSL pre-training and universal interface

Training an NN to be a language model is a useful *pretext task* in the sense of *self-supervised* training and *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

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. There are two equivalent ways to think of generation of L.

- Given $u_1, u_2, \dots, u_\ell$, the sequence may end here and $u_{\ell+1} = \text{<EOS>}$. Otherwise, the next token $u_{\ell+1} \neq \text{<EOS>}$ is generated.
- 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 $u_1, u_2, \dots, u_L \in \mathcal{U}$,

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

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

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

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

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

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

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 $(u_1, u_2, \ldots, u_L) = (u_1, u_2, \ldots, u_L, \texttt{<EOS>})$ 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. (Measure-theoretic probability theory unifies analysis.)

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

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

which is justified by

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

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.

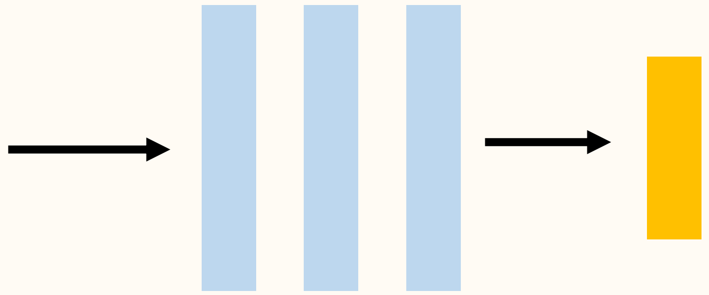
Learning with variable-size inputs

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-size input $X \in \mathcal{C}^*$ with a neural network.

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

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

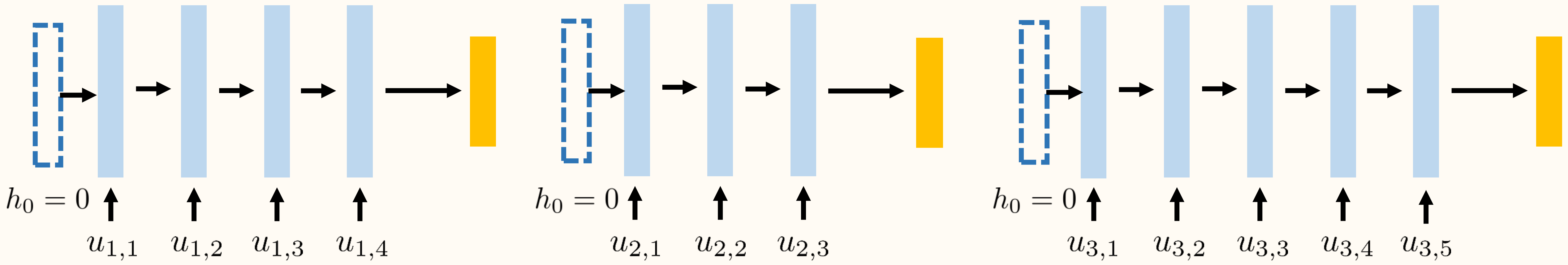
$$(u_{i,1}, u_{i,2}, u_{i,3}, 0, 0)$$



- (+) 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

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



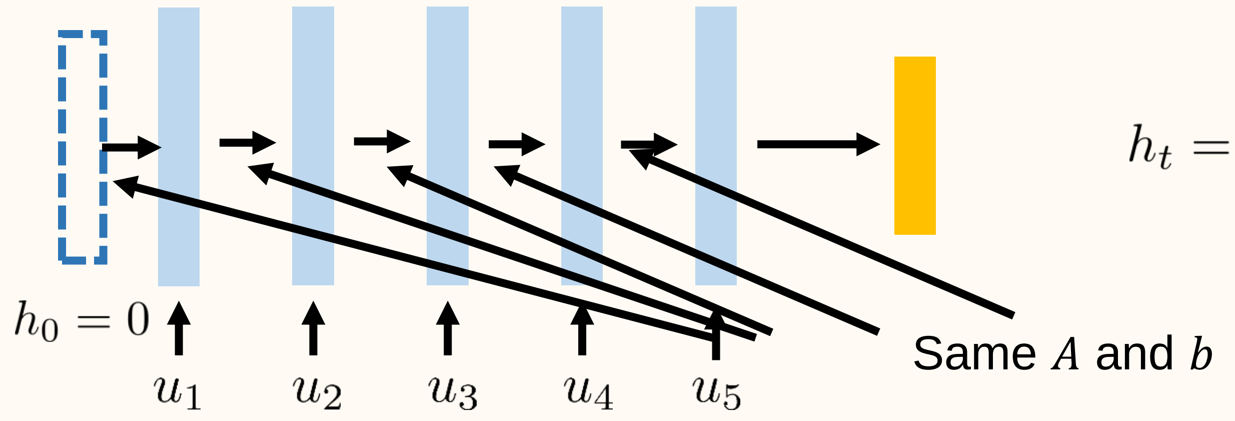
Idea: Process one input per layer

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

- (+) 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?



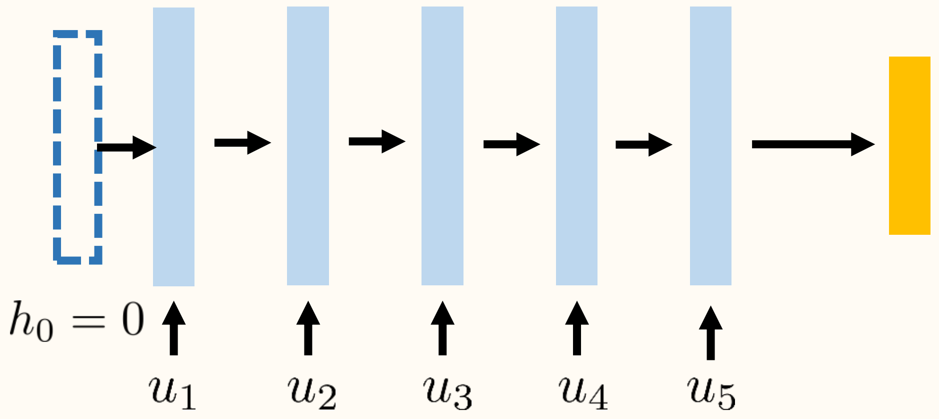
$$h_t = \sigma\left(A \begin{bmatrix} h_{\ell-1} \\ u_\ell \end{bmatrix} + b \right)$$

- (+) 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



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

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.

Backprop for RNN

Let $\tau(X) = (u_1, \dots, u_L)$ and $f_\theta(X) \in \mathbb{R}$. So $A \in \mathbb{R}^{1\times n}$, $b \in \mathbb{R}$, $\tilde{\theta} \in \mathbb{R}^p$, and $h_0, \dots, h_L \in \mathbb{R}^n$.

Let $\theta = (A, b, \tilde{\theta})$. Then,

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

Consider backpropagation on $\mathcal{L}(f_\theta(X), Y)$:

$$\frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial \theta} = \left(\frac{\partial \mathcal{L}}{\partial f}(f_\theta(X), Y) \right) \frac{\partial f_\theta(X)}{\partial \theta}$$

$$\frac{\partial f_\theta(X)}{\partial A} = h_L, \qquad \frac{\partial f_\theta(X)}{\partial b} = 1,$$

$$\frac{\partial f_\theta(X)}{\partial \tilde{\theta}} = A \frac{\partial h_L}{\partial \tilde{\theta}}.$$

Backprop for RNN

$$h_0 = 0$$

$$h_\ell = q_{\tilde{\theta}}(h_{\ell-1}, u_\ell), \qquad \ell = 1, \dots, L$$

Next, compute $\partial h_T / \partial \tilde{\theta}$:

$$\begin{aligned}
\frac{\partial h_L}{\partial \tilde{\theta}} &= \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{L-1}, u_L) + \frac{\partial q_{\tilde{\theta}}}{\partial h}(h_{L-1}, u_L) \frac{\partial h_{L-1}}{\partial \tilde{\theta}} + \left(\frac{\partial q_{\tilde{\theta}}(h,u)}{\partial u}(h_{L-1}, u_L) \right) \cancelto{0}{\frac{\partial u_L}{\partial \tilde{\theta}}} \\
&= \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{L-1}, u_L) + \frac{\partial q_{\tilde{\theta}}}{\partial h}(h_{L-1}, u_L) \left(\frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{L-2}, u_{L-1}) + \frac{\partial q_{\tilde{\theta}}}{\partial h}(h_{L-2}, u_{L-1}) \frac{\partial h_{L-2}}{\partial \tilde{\theta}} \right) \\
&= \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{L-1}, u_L) + \frac{\partial q_{\tilde{\theta}}}{\partial h}(h_{L-1}, u_L) \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{L-2}, u_{L-1}) + \frac{\partial q_{\tilde{\theta}}}{\partial h}(h_{L-1}, u_L) \frac{\partial q_{\tilde{\theta}}}{\partial h}(h_{L-2}, u_{L-1}) \frac{\partial h_{L-2}}{\partial \tilde{\theta}} \\
&= \sum_{\ell=1}^{L} \left(\left(\prod_{s=\ell+1}^{L} \frac{\partial q_{\tilde{\theta}}}{\partial h}(h_{s-1}, u_s) \right) \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{\ell-1}, u_\ell) \right) \\
&= \sum_{\ell=1}^{L} \left(\left(\prod_{s=\ell+1}^{L} \frac{\partial h_s}{\partial h_{s-1}} \right) \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{\ell-1}, u_\ell) \right) \qquad \left(\text{define } \frac{\partial h_s}{\partial h_{s-1}} = \frac{\partial q_{\tilde{\theta}}}{\partial h}(h_{s-1}, u_s) \right) \\
&= \sum_{\ell=1}^{L} \frac{\partial h_L}{\partial h_\ell} \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{\ell-1}, u_\ell) \qquad \left(\text{define } \frac{\partial h_L}{\partial h_\ell} = \prod_{s=\ell+1}^{L} \frac{\partial h_s}{\partial h_{s-1}} \right) \\
&= \sum_{\ell=1}^{L} \frac{\partial h_L}{\partial h_\ell} \frac{\partial h_\ell}{\partial \tilde{\theta}} \qquad \left(\text{define } \frac{\partial h_\ell}{\partial \tilde{\theta}} = \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{\ell-1}, u_\ell) \right)
\end{aligned}$$

Backprop for RNN

Translate calculation to $\partial \ell / \partial \tilde{\theta}$:

$$\frac{\partial f_\theta(X)}{\partial \tilde{\theta}} = \sum_{\ell=1}^{L} \frac{\partial f_\theta(X)}{\partial h_\ell} \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{\ell-1}, u_\ell) \qquad \left(\text{define } \frac{\partial f_\theta(X)}{\partial h_\ell} = \frac{\partial f_\theta(X)}{\partial h_L} \frac{\partial h_L}{\partial h_\ell} = A \frac{\partial h_L}{\partial h_\ell}\right)$$

$$\begin{aligned} \frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial \tilde{\theta}} &= \sum_{\ell=1}^{L} \frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial f} \frac{\partial f_\theta(X)}{\partial h_\ell} \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{\ell-1}, u_\ell) \\ &= \sum_{\ell=1}^{L} \frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial h_\ell} \frac{\partial q_{\tilde{\theta}}}{\partial \tilde{\theta}}(h_{\ell-1}, u_\ell) \qquad \left(\text{define } \frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial h_\ell} = \frac{\partial \mathcal{L}(f_\theta(X), Y)}{\partial f} \frac{\partial f_\theta(X)}{\partial h_\ell}\right) \end{aligned}$$

This is called *backpropagation through time* (BPTT).

P. J. Werbos, Generalization of backpropagation with application to a recurrent gas market model, *Neural Networks*, 1988.

Backprop code for RNN

```
# Forward pass given $\tau(X)$=(u[1],...u[L])
h[0] = 0
for l = 1,2,...,L:
  h[l] = q(th,h[l-1],u[l])                # $h_\ell = q_{\tilde\theta}(h_{\ell-1}, u_\ell)$
fX = A @ h[L] + b                         # $f_\theta(X) = Ah_L + b$
ell = loss(fX,Y)                          # $\mathcal{L}(f_\theta(X), Y)$

# Backward pass
dldf = loss.df(fX,Y)                      # $\partial\mathcal{L}/\partial f$
dldA = dldf @ h[L]                        # $\partial\mathcal{L}/\partial A$ ✓
dldb = dldf                               # $\partial\mathcal{L}/\partial b$ ✓

dldh = dldf @ A                           # $\partial\mathcal{L}/\partial h_L$
dldth = 0                                 # .zero_grad()
for l = L,L-1,...,2,1 :
  dldth += dldh @ q.dth(h[l-1],u[l]) # $\partial\mathcal{L}/\partial\tilde\theta$ (partial sum)
  dldh = dldh @ q.dh(h[l-1],u[l])         # $\partial\mathcal{L}/\partial h_{\ell-1}$

# $\partial\mathcal{L}/\partial\tilde\theta$ ✓ (all terms in sum accounted for)
```

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

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.

Exploding gradients and gradient clipping

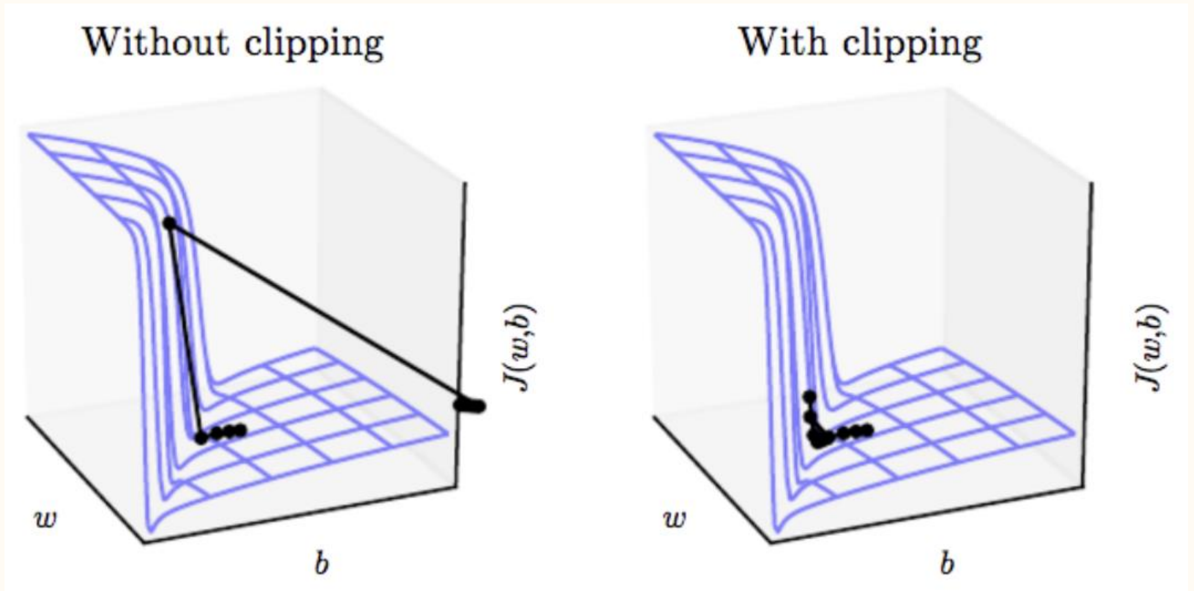
The *exploding gradient* problem occurs when the gradient magnitude is very large.

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

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

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. (Although this argument is not precisely correct since large changes in u_ℓ may affect $\mathcal{L}$.) In any case, the gradient signal from far away at time L is lost and the model can't learn what information to preserve at time ℓ.

Promoting better gradient flow

As an example, 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; any information contained in $\frac{\partial \mathcal{L}}{\partial h_{\ell+1}}$ is lost.

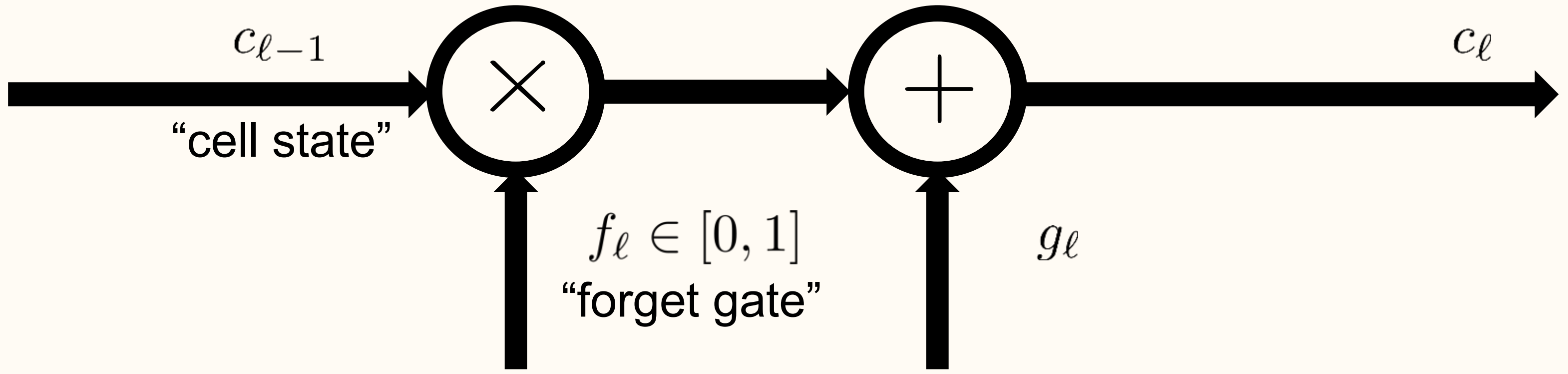
Promoting better gradient flow

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.

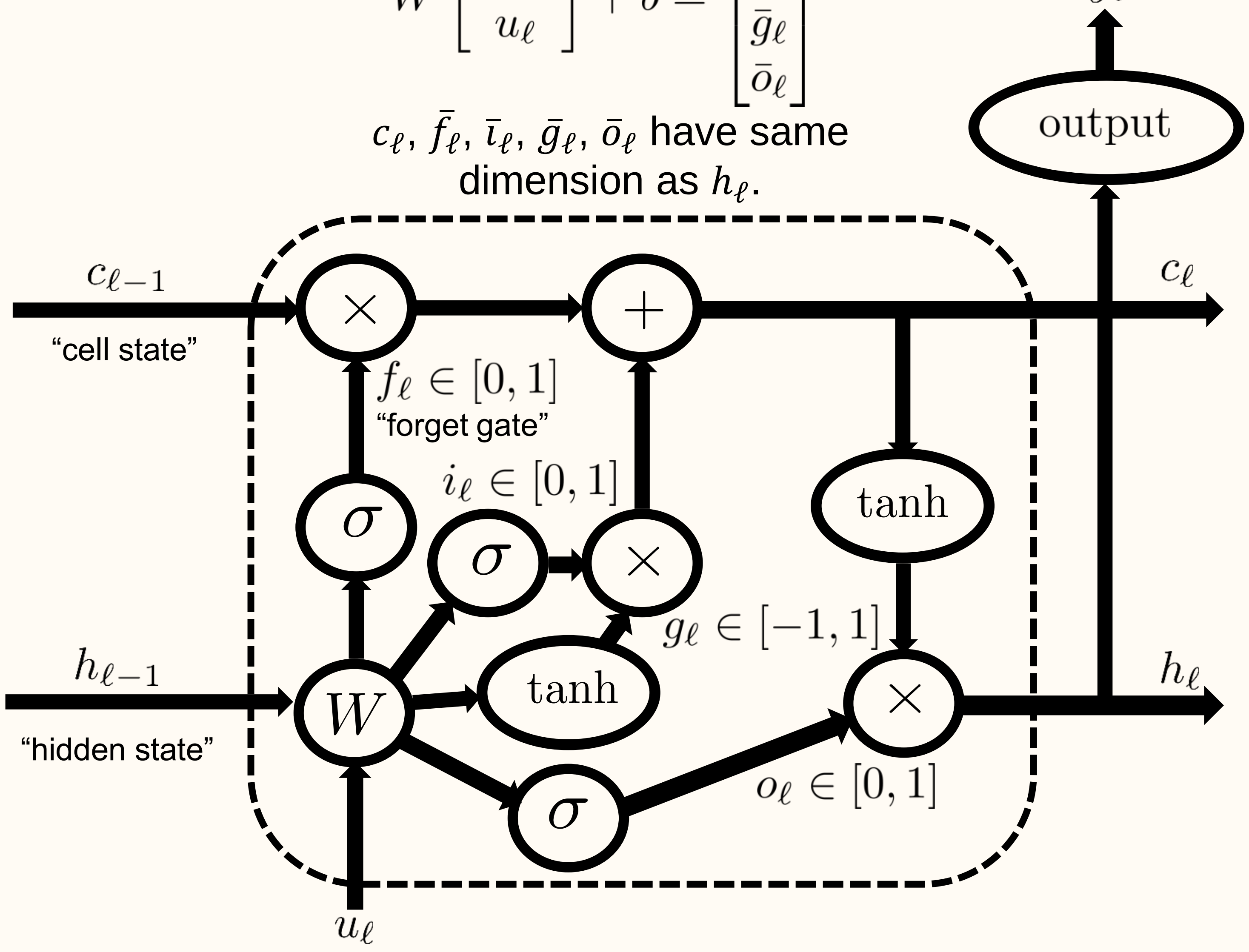
Works much better than a naïve RNN!

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.

LSTM implementation

```
class LSTMScratch(nn.Module):
  def __init__(self, u_dim, h_dim):
    super().__init__()
    self.W = nn.Linear(u_dim + h_dim, 4 * h_dim)  # (W initialization omitted)
    self.u_dim, self.h_dim = u_dim, h_dim

  # input shape (length,batch,u_dim)
  # for now, assume all seq. in batch have same length
  def forward(self, inputs, h, c) :
    # h, c are usually zero
    output = zeros(inputs.shape[0], inputs.shape[1], num_hiddens)
    for ind,u in enumerate(inputs) :
      (f,i,g,o) = self.W(torch.cat(h,u,dim=2)).chunk(4)
      f, i, g, o = torch.sigmoid(f), torch.sigmoid(i), torch.tanh(g), torch.sigmoid(o)
      c = c * f + i * g
      h = o * torch.tanh(c)
      output[ind,:,:] = h
    #output shape (length,batch,h_dim)
    return outputs, (h, c)
```

$$W \begin{bmatrix} h_{t-1} \\ u_t \end{bmatrix} + b = \begin{bmatrix} \bar{f}_t \\ \bar{i}_t \\ \bar{g}_t \\ \bar{o}_t \end{bmatrix}$$

LSTM implementation

Example usage of LSTM model:

```
rnn = nn.LSTMScratch(10, 20)    # (u_dim, h_dim)
input = torch.randn(5, 3, 10)   # (length, batch, u_dim)
h0 = torch.zeros(3, 20)         # (batch, h_dim)
c0 = torch.zeros(3, 20)         # (batch, h_dim)
output, (hn, cn) = rnn(input, (h0, c0))
```

(Recurrent application of LSTM done within class method.)

LSTM is somewhat complicated. However, programming abstraction makes it easy to use once implemented.

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. (In fact no mainstream NLP system currently uses long term memory.)

Aside: Exploding/vanishing gradient problem

The exploding/vanishing gradient problem is a problem not just for RNNs. It can be a problem for all deep neural networks.

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

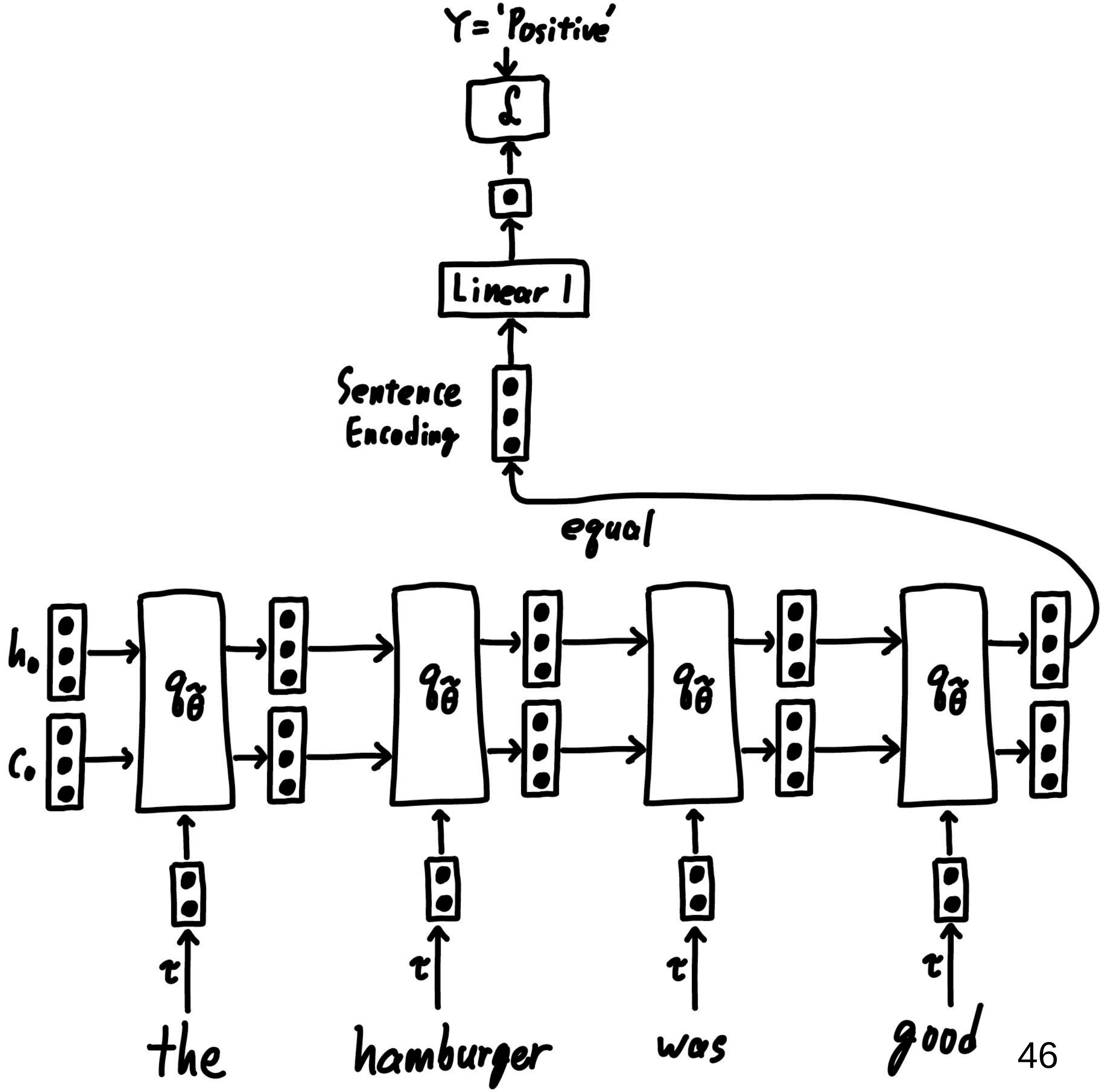
Another technique is the use of normalization layers such as batch norm.

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.

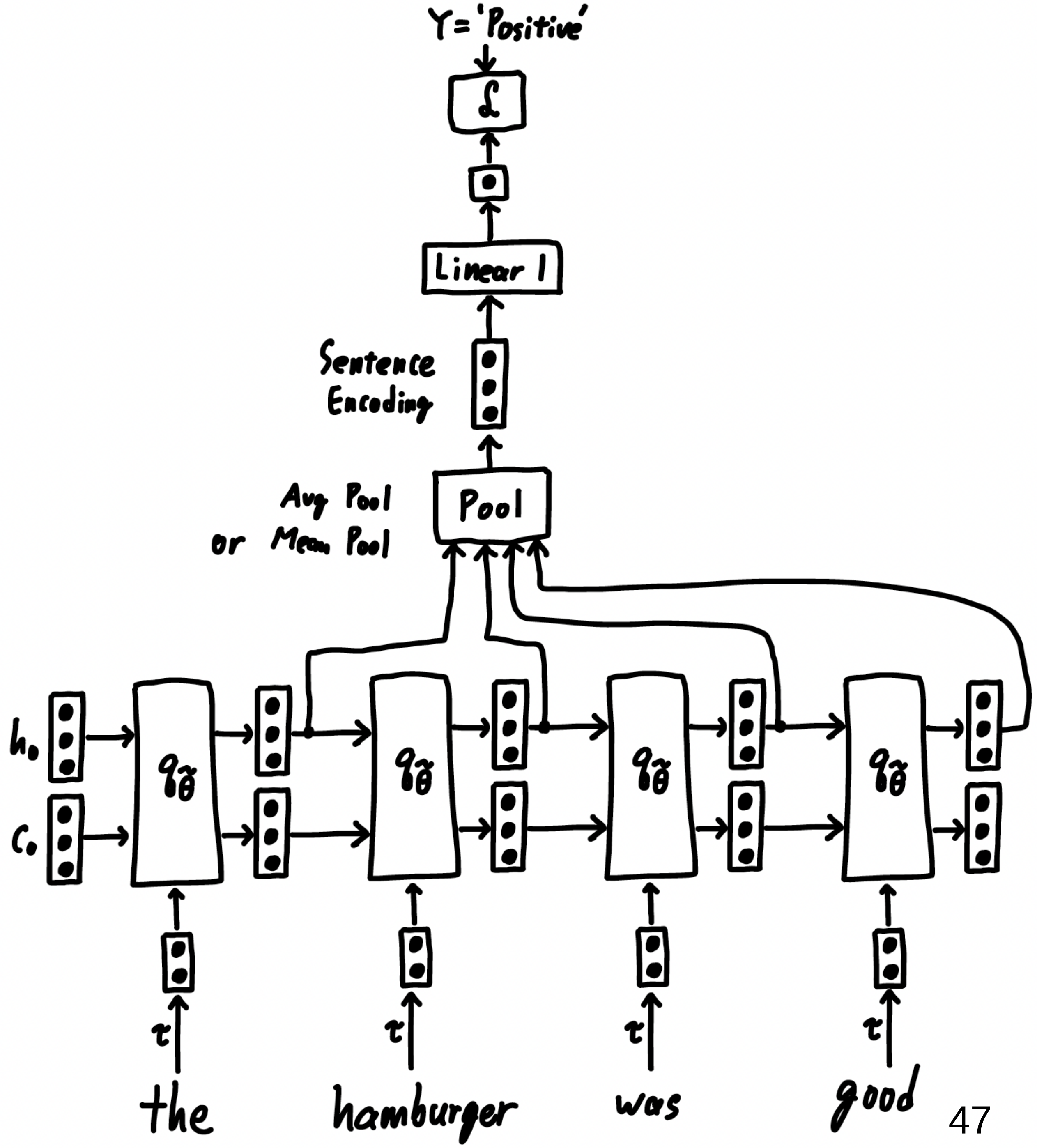
Sentiment analysis with LSTM

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.



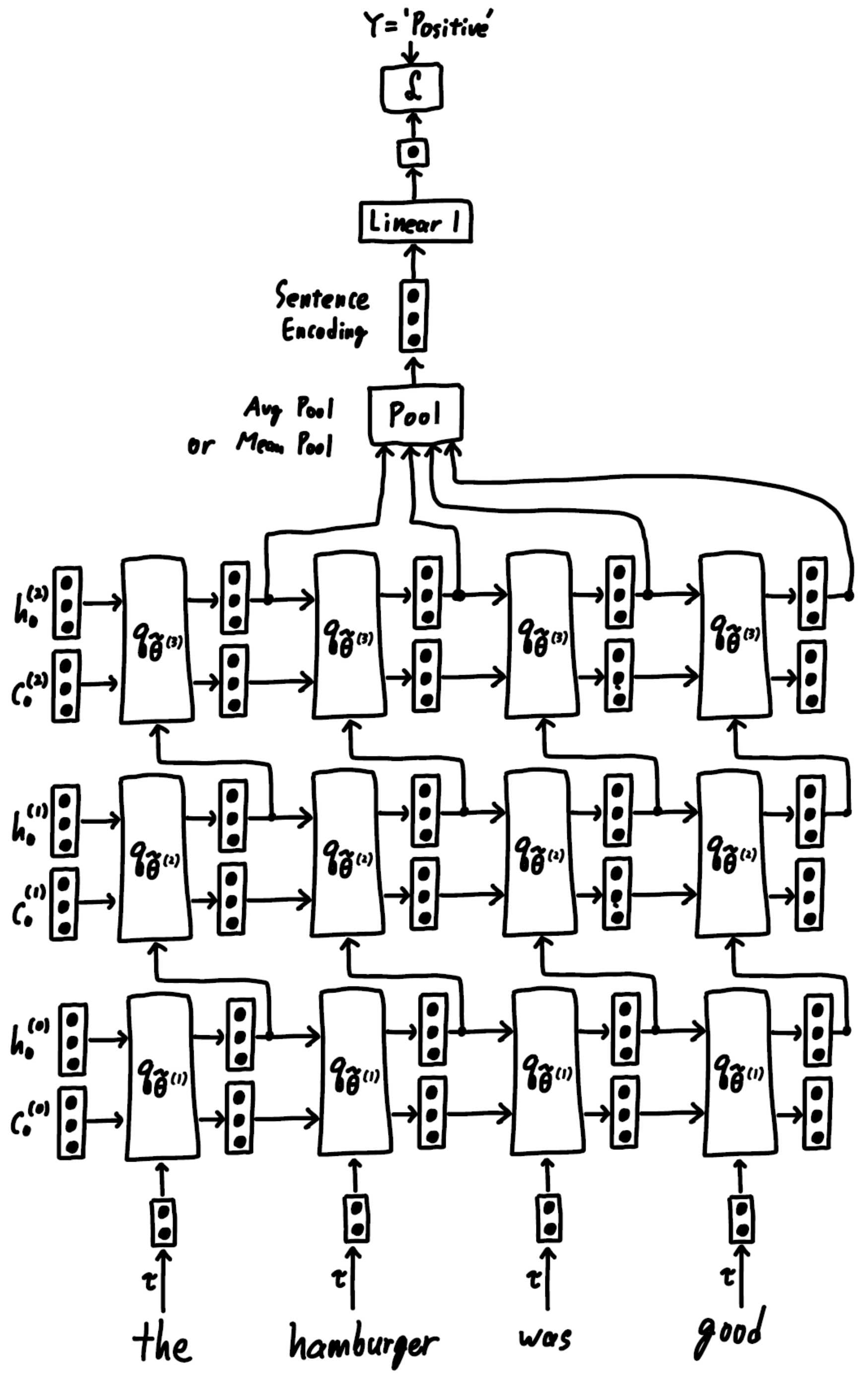
Stacked RNN

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

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

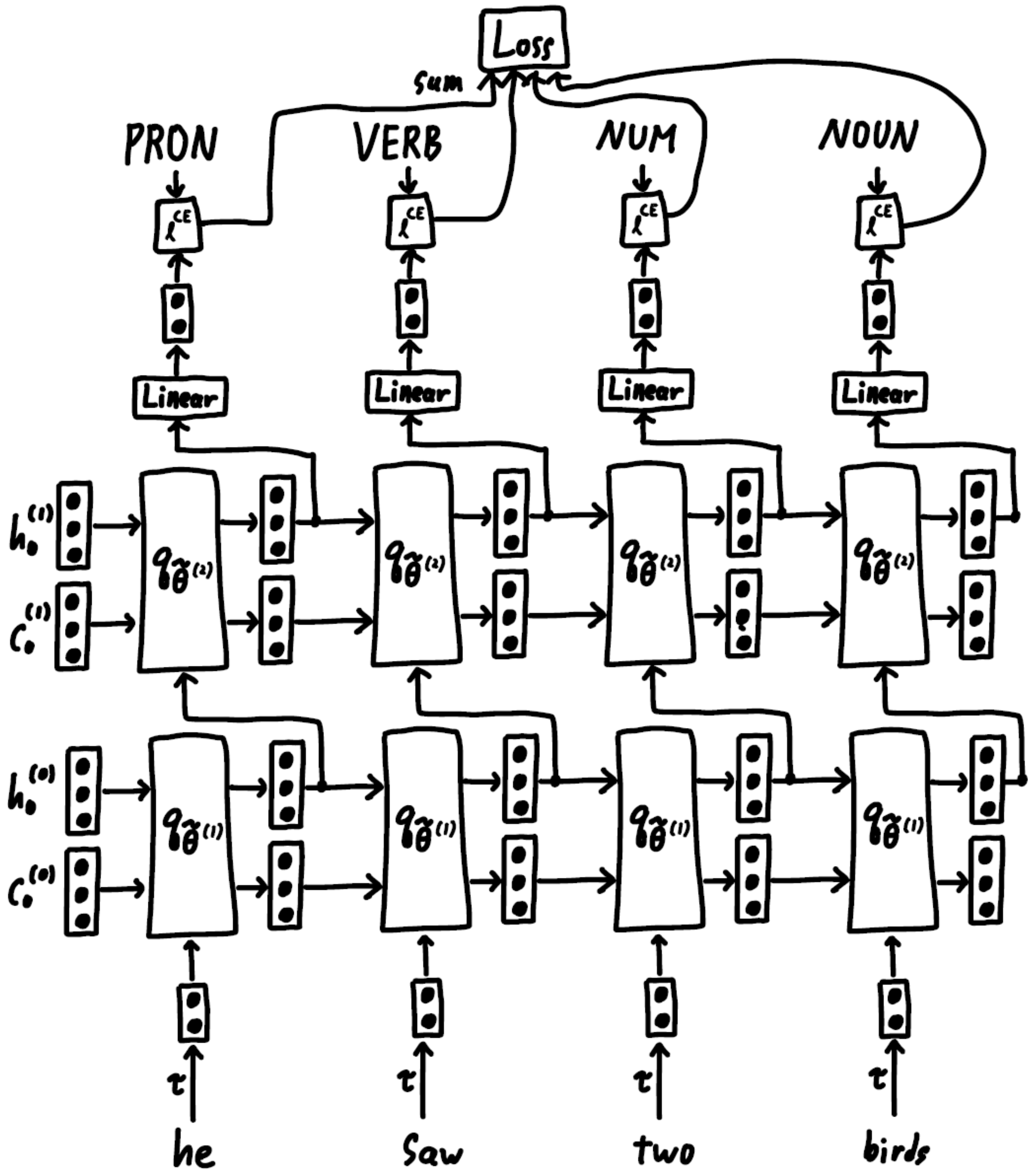
- 2 layers is almost always better than 1.
- 3 layers is not always better than 2.

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



Example task: Parts of speech tagging

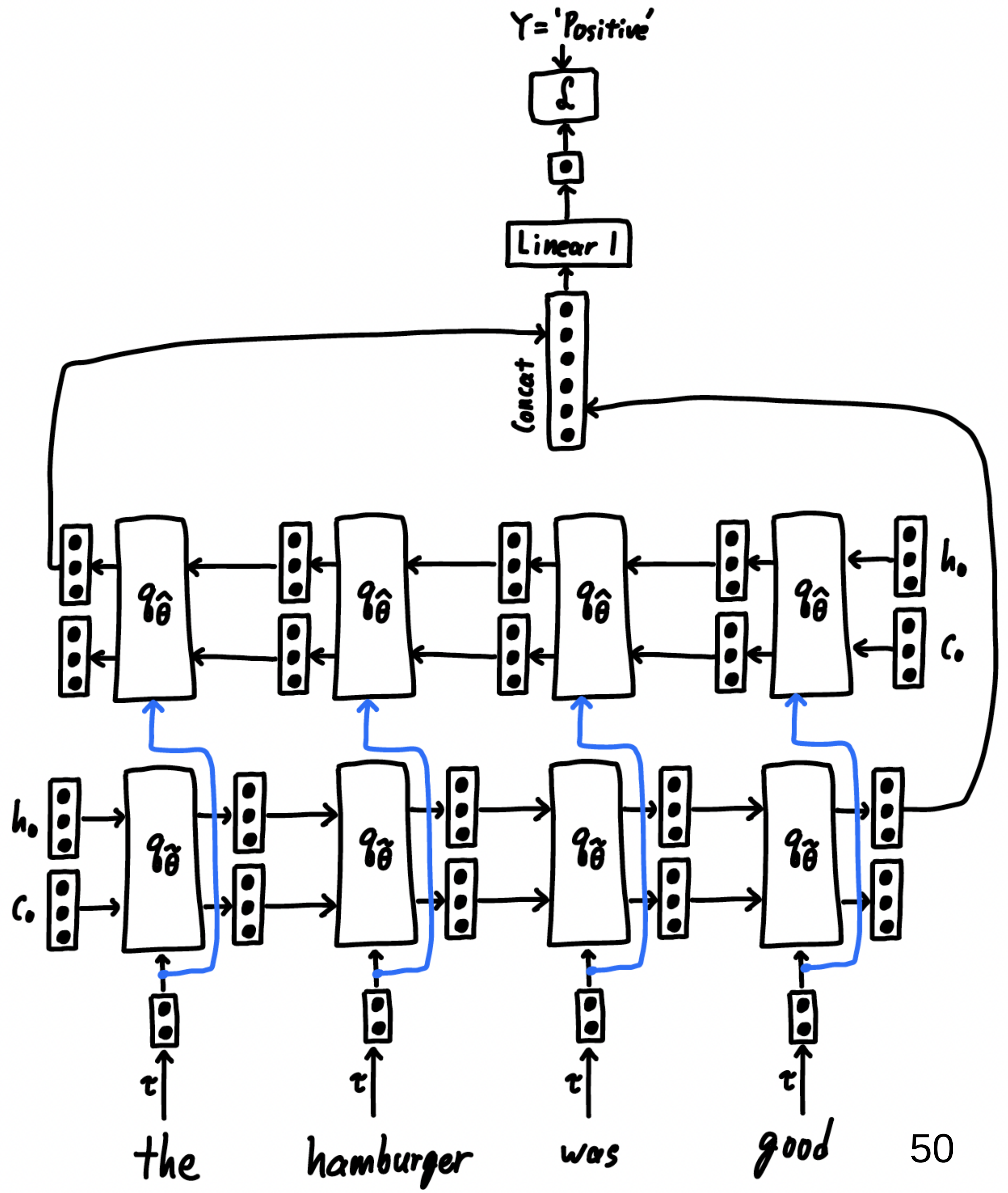
For some RNN tasks, the output is a sequence, and the total loss is the sum of the losses incurred at each sequence term.



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.

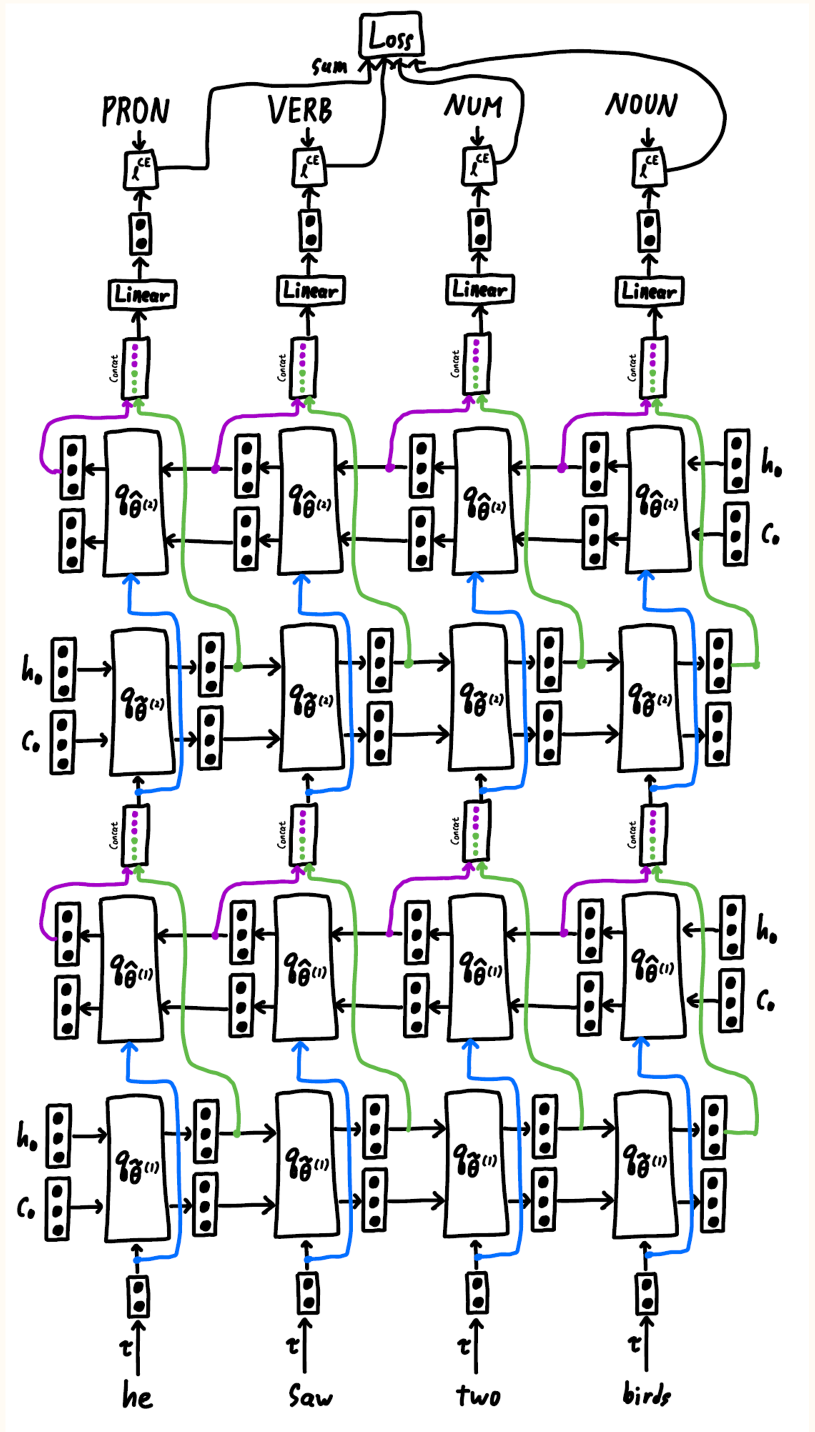
A *bidirectional RNN* combines forward and reverse directional RNNs to process a sequence without a single sense of direction.



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

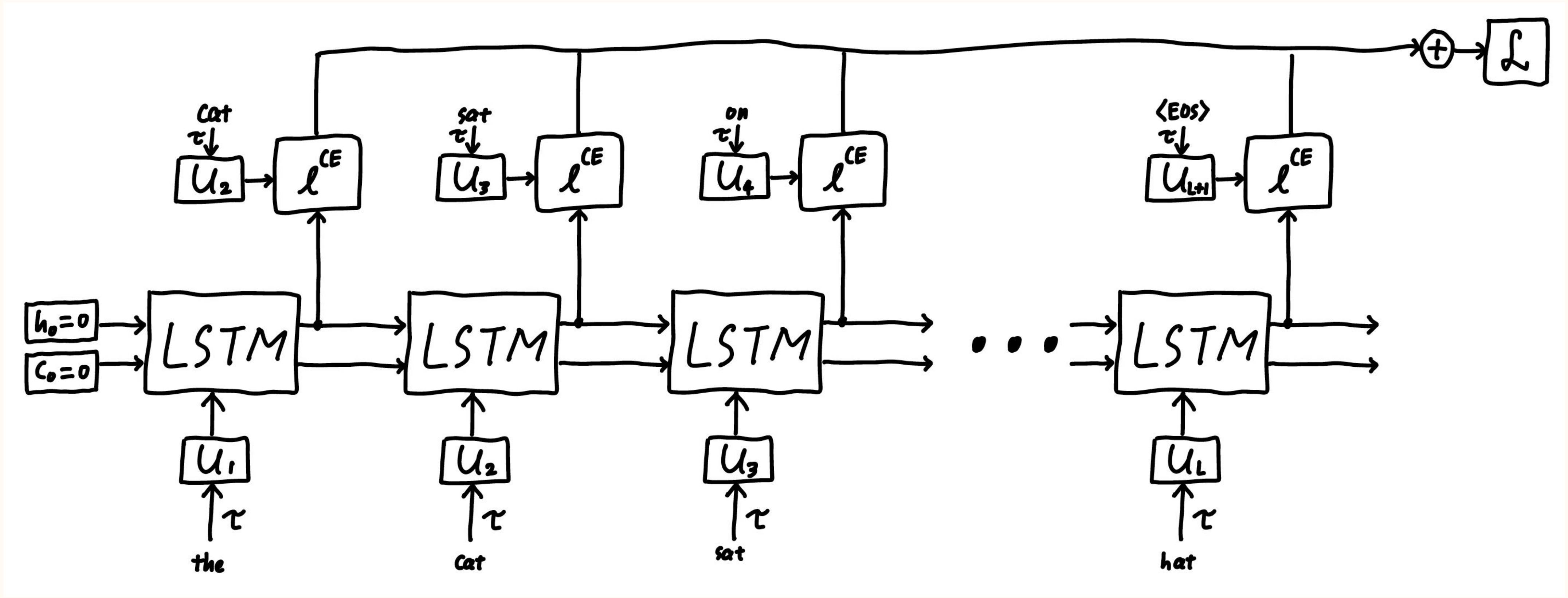
Stacked bidirectional RNN

Stacking and bidirectionality can be combined.



RNN-LM

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



LM loss

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 $(u_1, \dots, u_L)$. RNN predicts the next token

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

for each ℓ. The (in)accuracy of the prediction is measured by the cross entropy loss

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

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

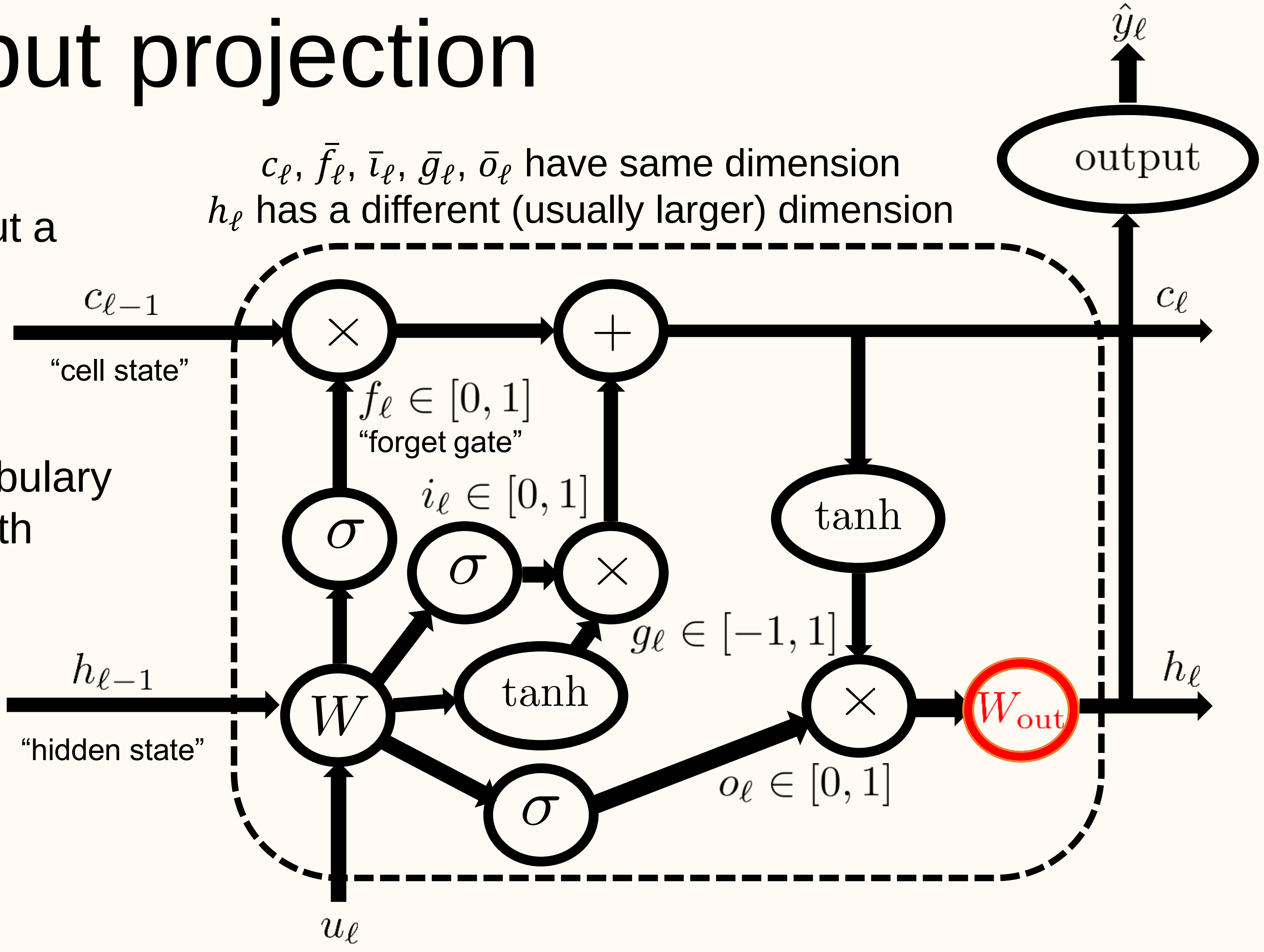
LSTM with output projection

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.

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

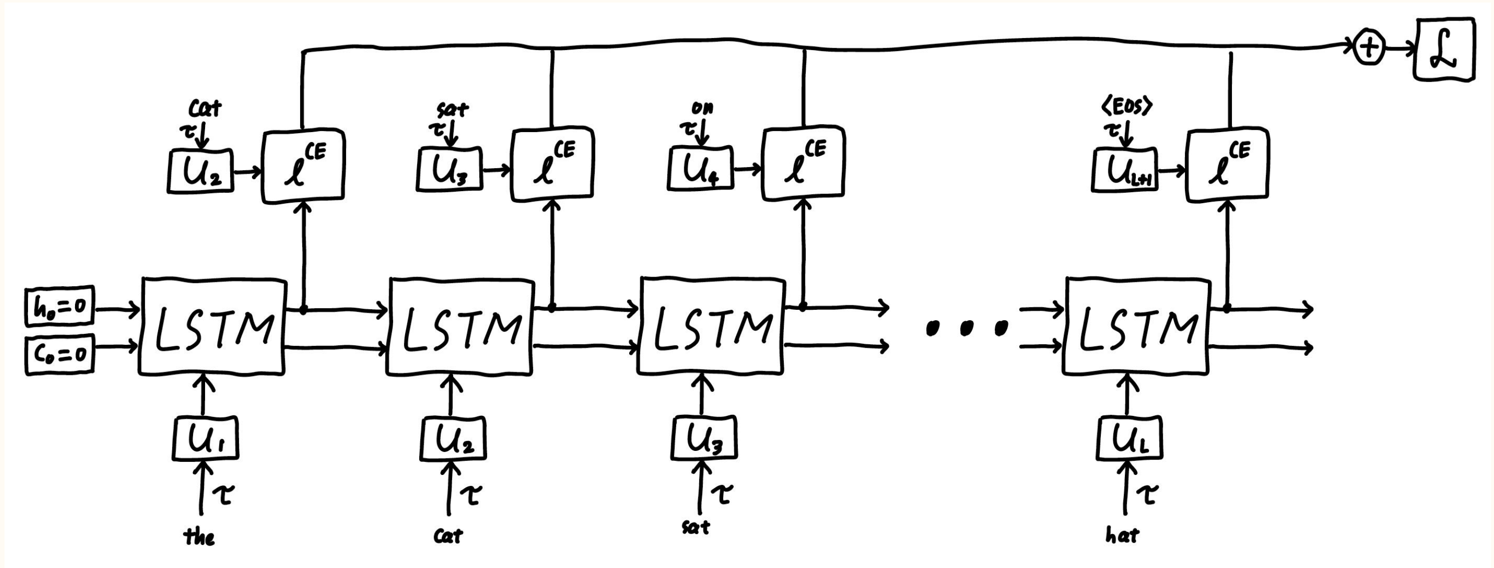


Backprop with RNN?

In this RNN-LM, the output of LSTM goes into two blocks, so the backprop computation should be the sum of the two contributions.

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

This leads to the general graph-form backprop.



Trainable tokenizer

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

Instead of using one-hot encodings, which is fixed (given a 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.

word2vec

Given an input $X = (w_1, \dots, w_L)$ chunked into words $w_1, \dots, w_L \in \mathcal{W}$, train a tokenizer τ_θ such that $\tau_\theta(w_\ell) \in \mathbb{R}^d$ is determined by word co-occurrence. Intuition is that two words are similar if the distribution of nearby words are similar.

Train τ_θ with a large corpus of unlabeled text.

Using such a trained τ_θ with RNNs significantly improves performance, compared to simple one-hot tokenizers.

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

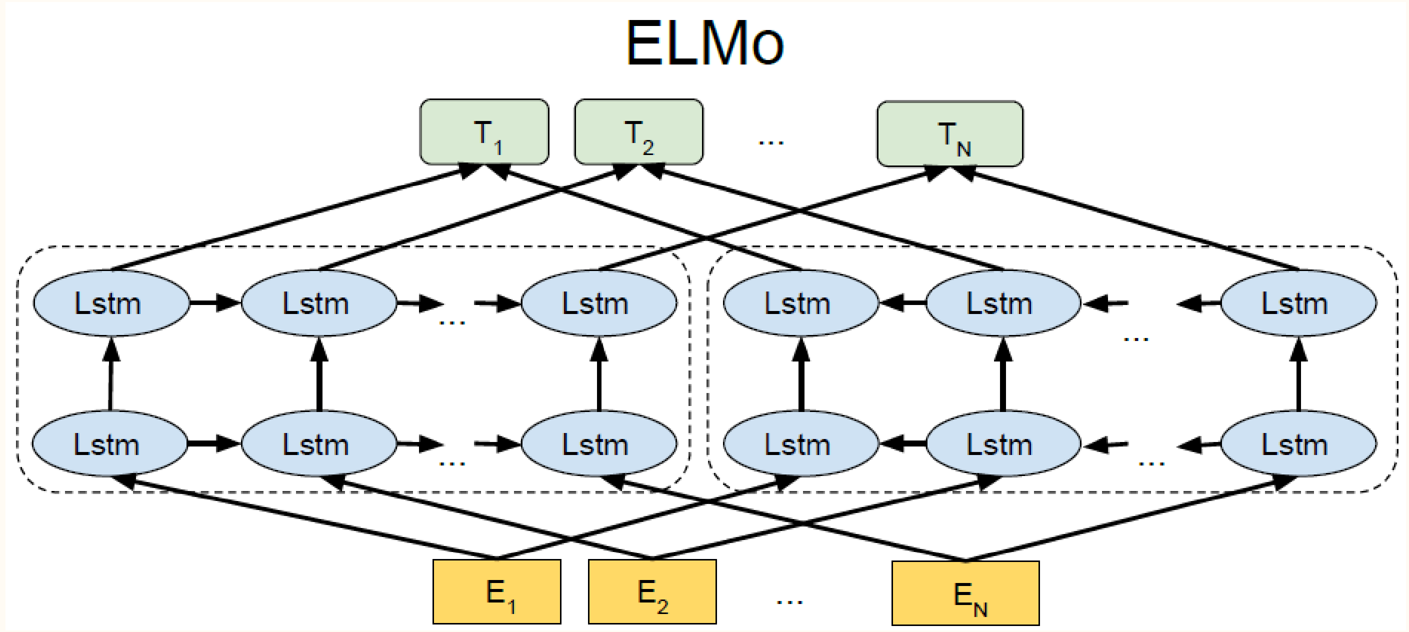
T. Mikolov, K. Chen, G. Corrado, and J. Dean, Efficient estimation of word representations in vector space, *ICLR Workshop*, 2013.
T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, Distributed representations of words and phrases and their compositionality, *NeurIPS*, 2013.

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, but we won’t pay attention to it.)



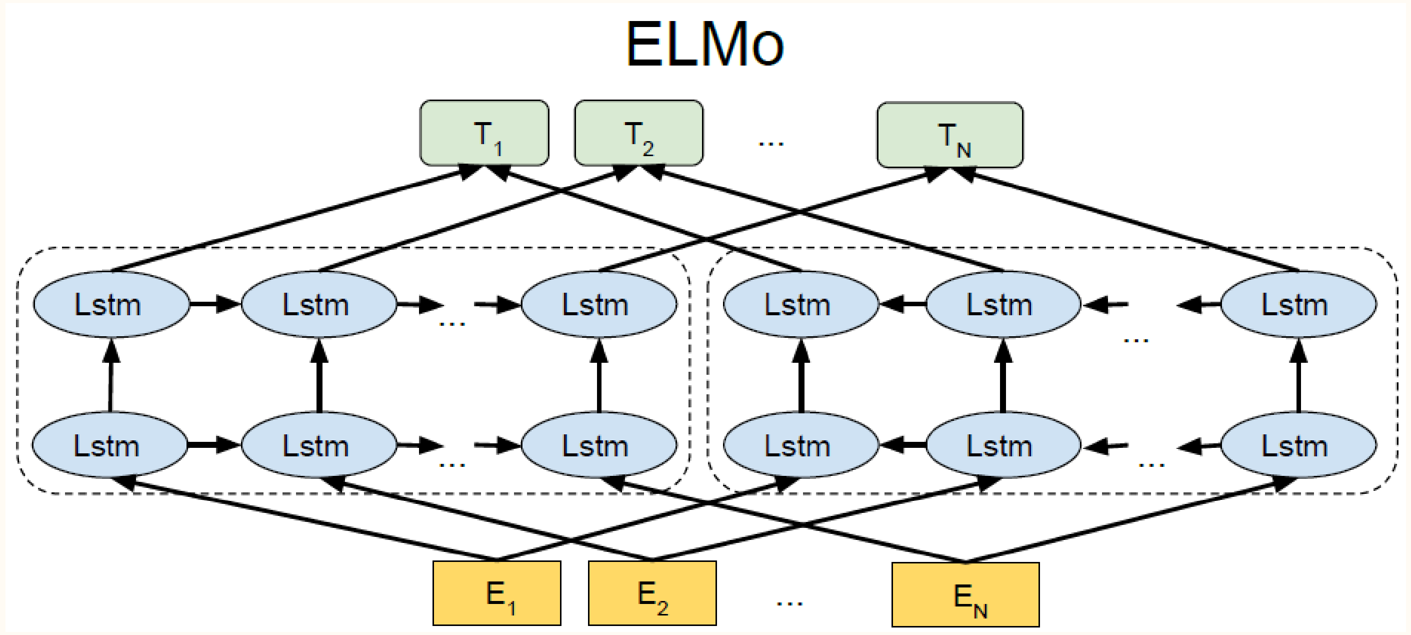
M. E. Peters, W. Ammar, C. Bhagavatula, and R. Power, Semi-supervised sequence tagging with bidirectional language models, *ACL*, 2017.
M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, Deep contextualized word representations, *NAACL*, 2018.

Bidirectional LM loss for pre-training

Pre-training uses the loss

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

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.



Non-causal language model

Causal language models learn

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

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

ELMo and BERT are not 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, $\left\{s_k^{\text{task}}\right\}_{k=0}^{K}$, and the weights of the NLP method on labeled fine-tuning data.

Results

ELMo achieves 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 semi-supervised learning through large-scale pre-training and fine-tuning was not new (Dai and Le 2015) ELMo executed it very well and advanced the state of the art substantially.

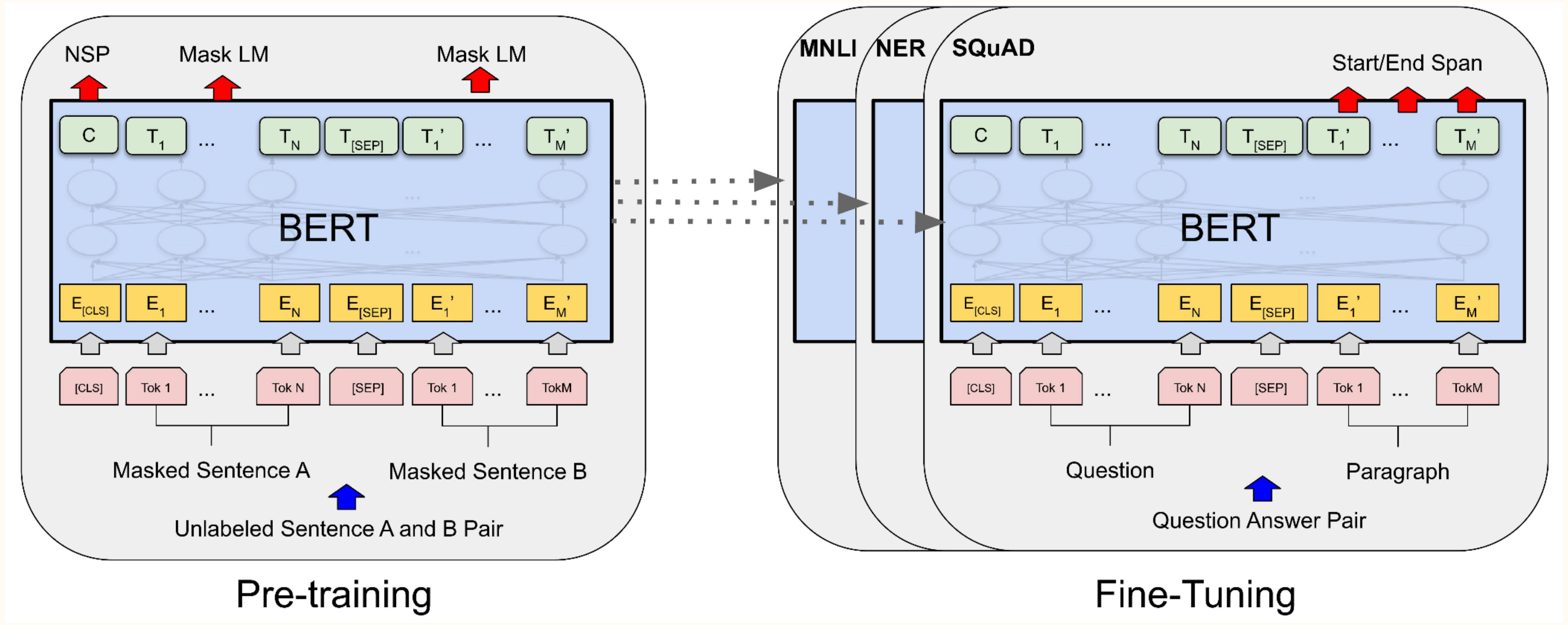
However, LSTM RNN is not the best architecture. The left- and right-directional RNNs only process information unidirectionally. What is the model needs to examine the entire sentence to make inference? Also, RNNs are fundamentally computationally inefficient.

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

- Multi-head self-attention
- Residual connections
- Layer Normalization
- Position-wise FFN
- Positional encodings

Single-head self attention

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

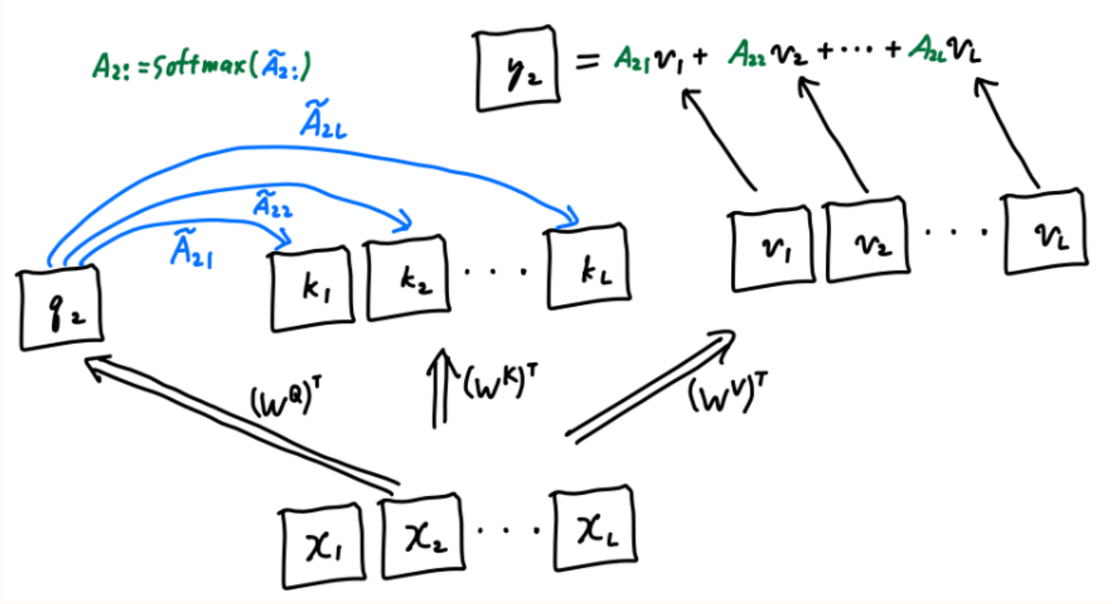
$$Q = XW^Q \in \mathbb{R}^{L \times d_K}, \qquad K = XW^K \in \mathbb{R}^{L \times d_K}, \qquad V = XW^V \in \mathbb{R}^{L \times d_V}$$

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

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

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

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



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 $V \in \mathbb{R}^{L\times d_{\text{in}}}$ to $\text{Output} \in \mathbb{R}^{L\times d_{\text{out}}}$.

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.

$$
\begin{aligned}
&x_1, \ldots, x_L \in \mathbb{R}^{d_X}, \qquad \{x_\ell\}_{\ell=1}^L = X \in \mathbb{R}^{L \times d_X} \\
&\text{for } h = 1, \ldots, H \\
&\qquad Y_h = \text{Attention}(XW_h^Q, XW_h^K, XW_h^V) \in \mathbb{R}^{L \times d_V} \\
&Z = \text{MHA}(X) = \underbrace{\text{concat}(Y_1, \ldots, Y_H)}_{L \times H d_V} \; \underbrace{W^O}_{H d_V \times d_Z} \\
&z_1, \ldots, z_L \in \mathbb{R}^{d_Z}, \qquad \{z_\ell\}_{\ell=1}^L = Z \in \mathbb{R}^{L \times d_Z}
\end{aligned}
$$

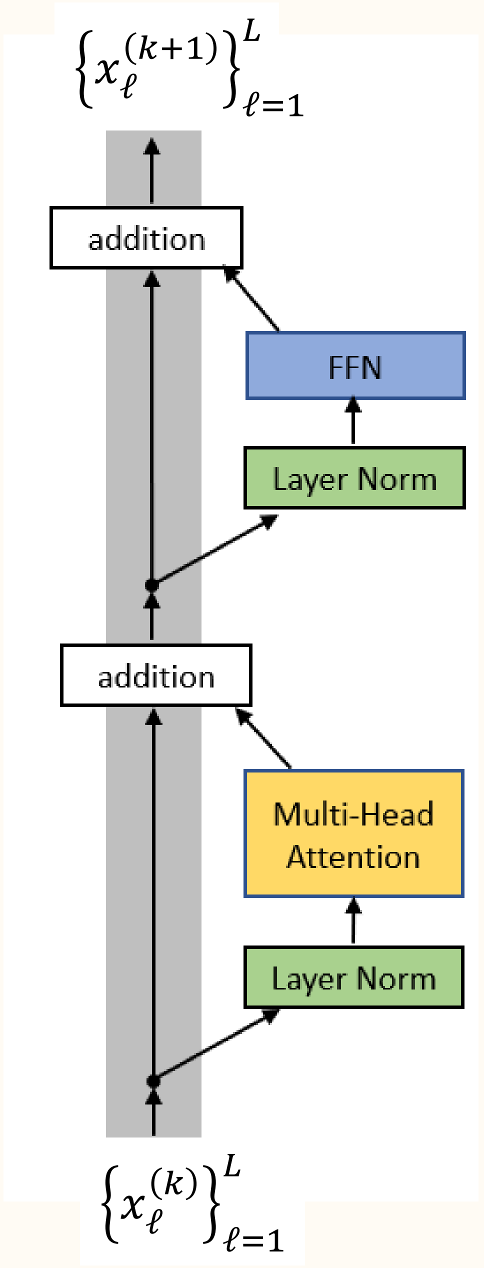
Seq-to-seq transformation $\{x_\ell\}_{\ell=1}^L \mapsto \{z_\ell\}_{\ell=1}^L$. Often $d_X = d_Z$ required by residual connection.

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.



$\left\{x_\ell^{(k+1)}\right\}_{\ell=1}^{L}$
addition
FFN
Layer Norm
addition
Multi-Head Attention
Layer Norm
$\left\{x_\ell^{(k)}\right\}_{\ell=1}^{L}$

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)

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

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

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

TF LN ≠ CNN 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.

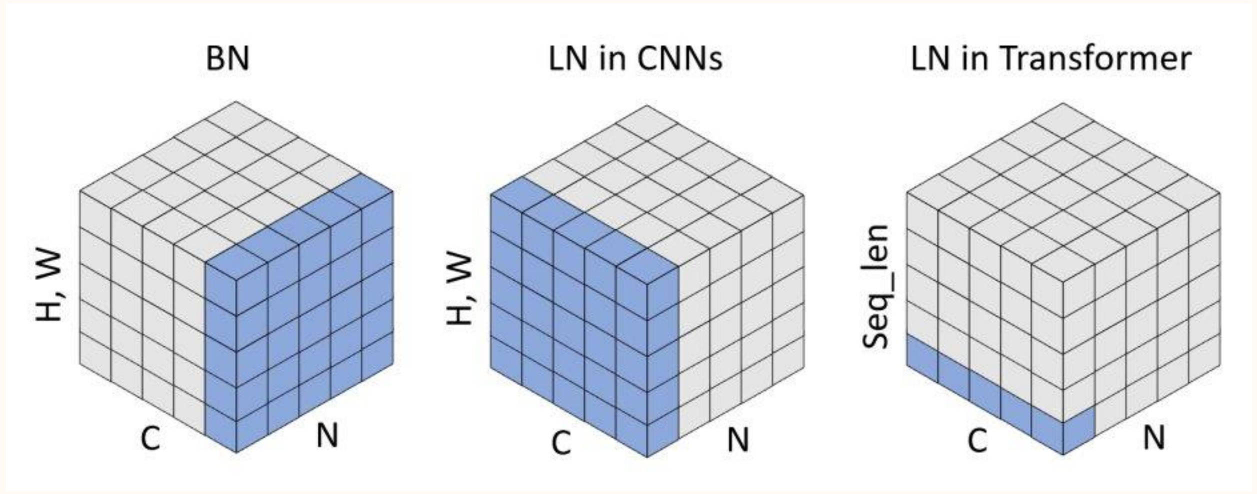


Figure credit to:
Z. Yao, Y. Cao, Y. Lin, Z. Liu, Z. Zhang, and H. Hu, Leveraging batch normalization for vision transformers, *ICCVW*, 2021.

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

Applies independently on each embedding, i.e.,

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

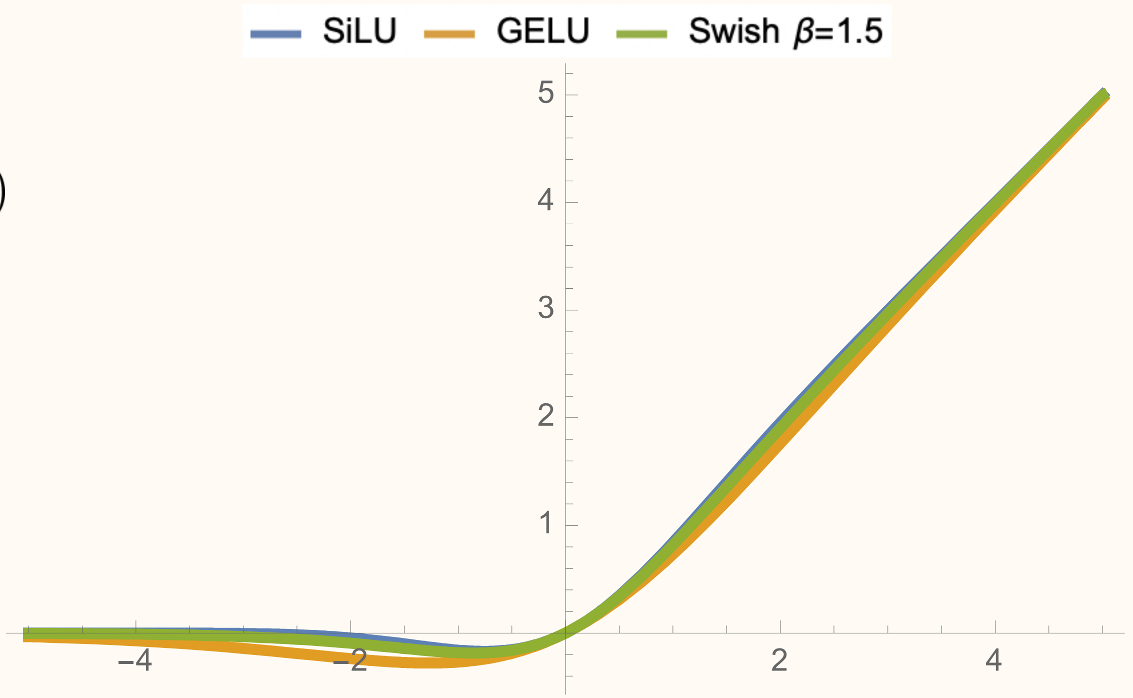
GELU, SiLU, Swish activations

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.

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

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

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



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: 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.

Solution: After tokenization $\{u_\ell\}_{\ell=1}^{L} = \tau(X)$, add positional embedding vectors and then pass

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

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 $p_1, \ldots, p_L \in \mathbb{R}^d$

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

(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,

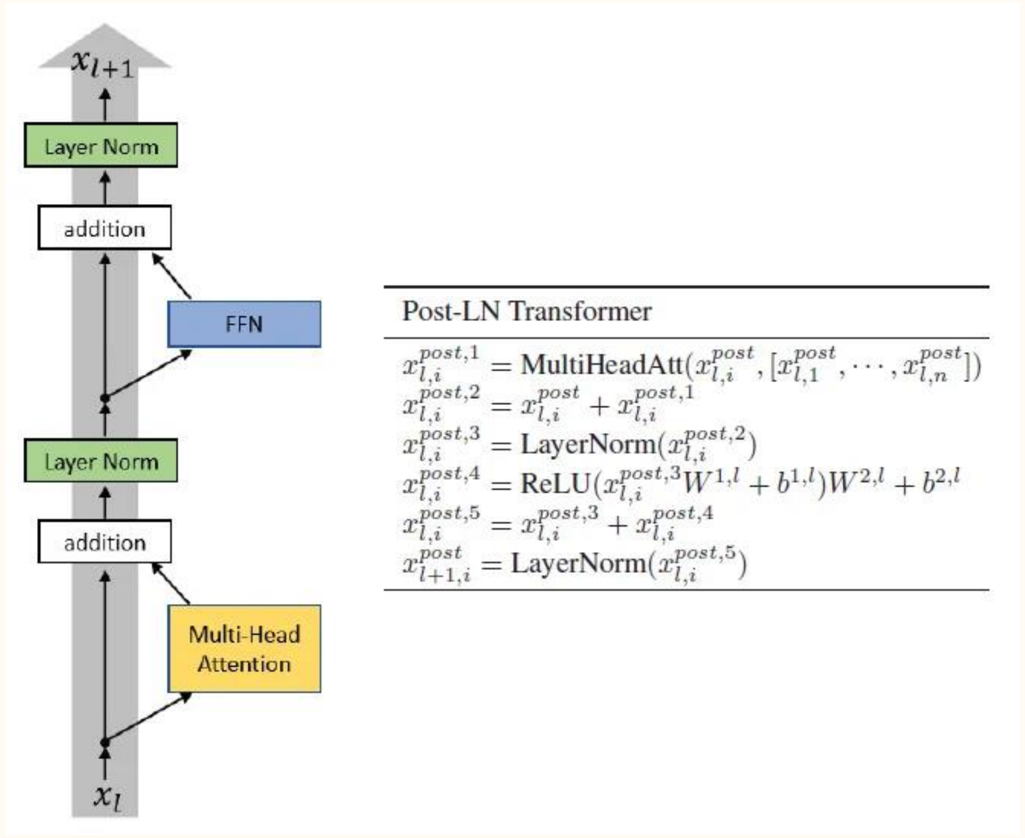
However, Sukhbaatar et al. 2015 and Gehring et al. 2017 did publish the positional encoding technique earlier. The sinusoidal encoding is due to Vaswani et a. 2017.

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

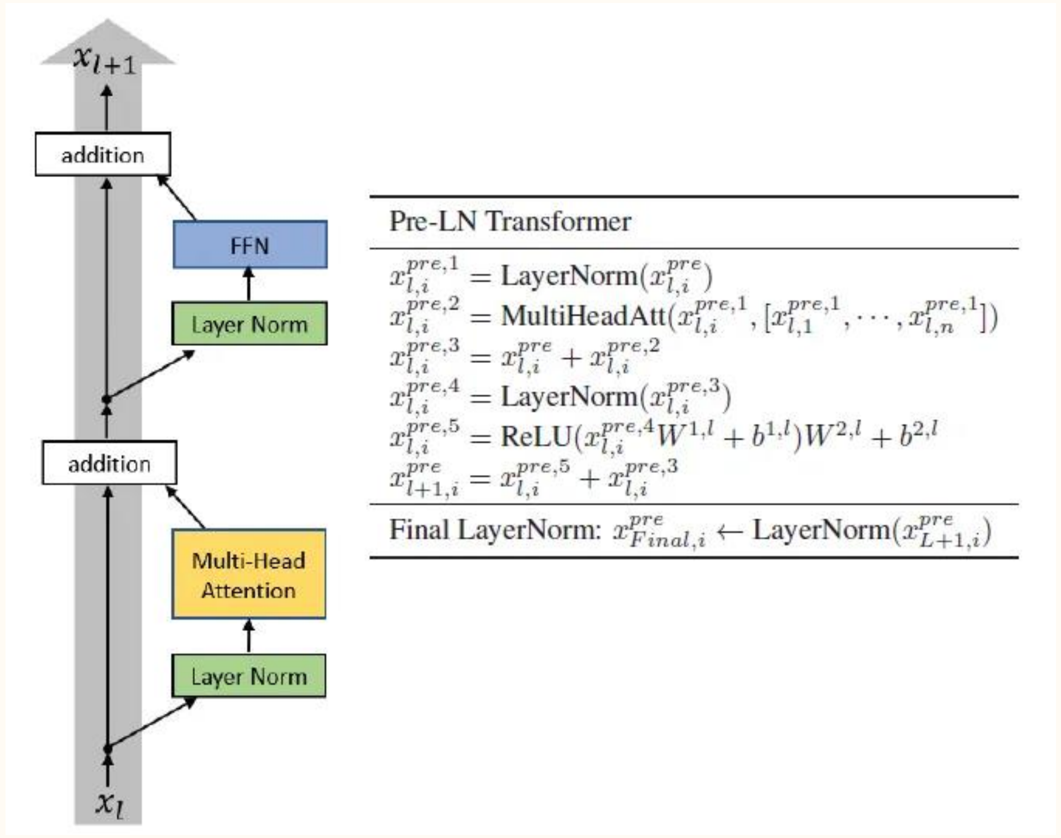
There are 2 variants of the transformer architecture based on the position of LN.

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

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



Pre-LN Transformer

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

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

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 can’t handle 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 when computing the representation of each token.

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

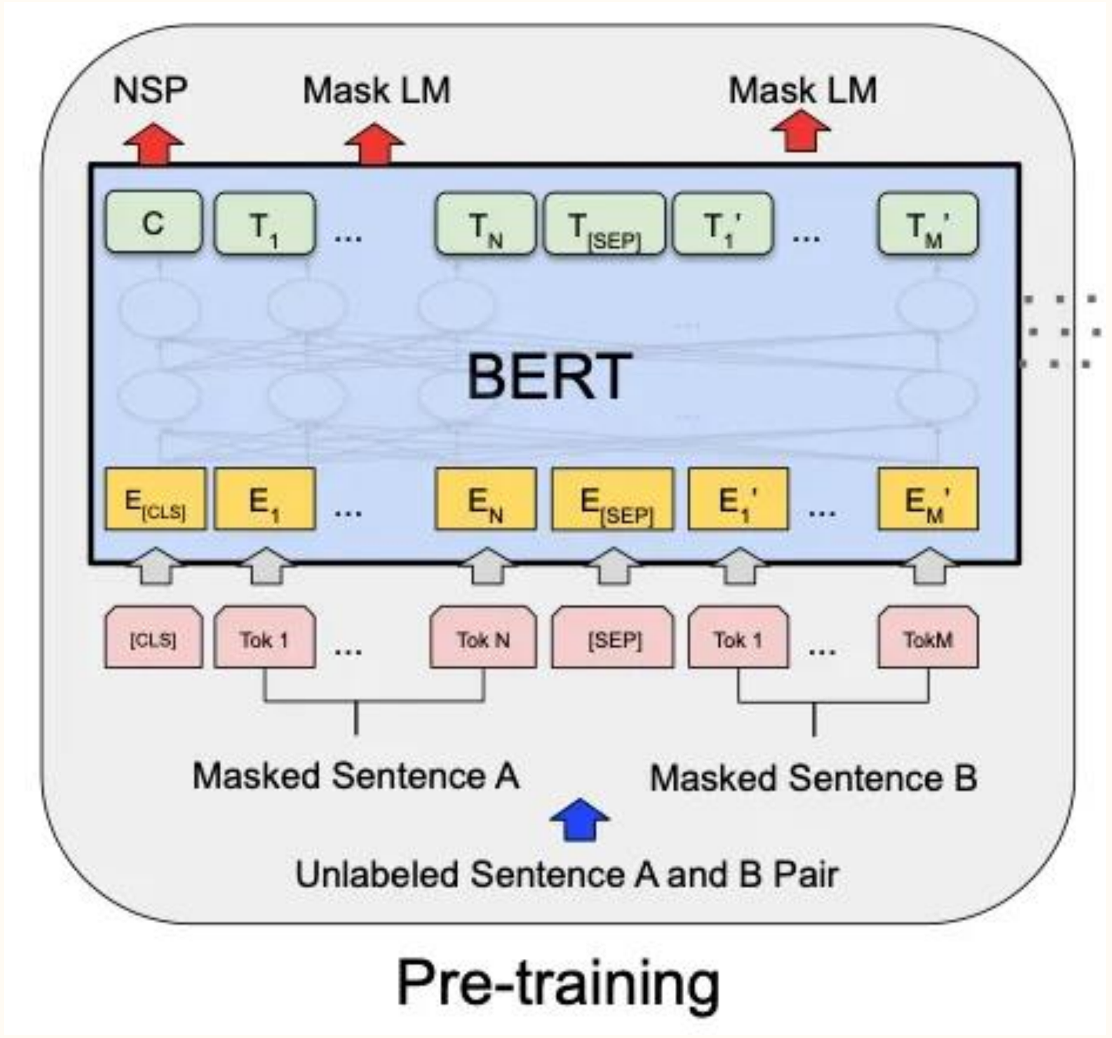
BERT pre-training uses two losses.

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.

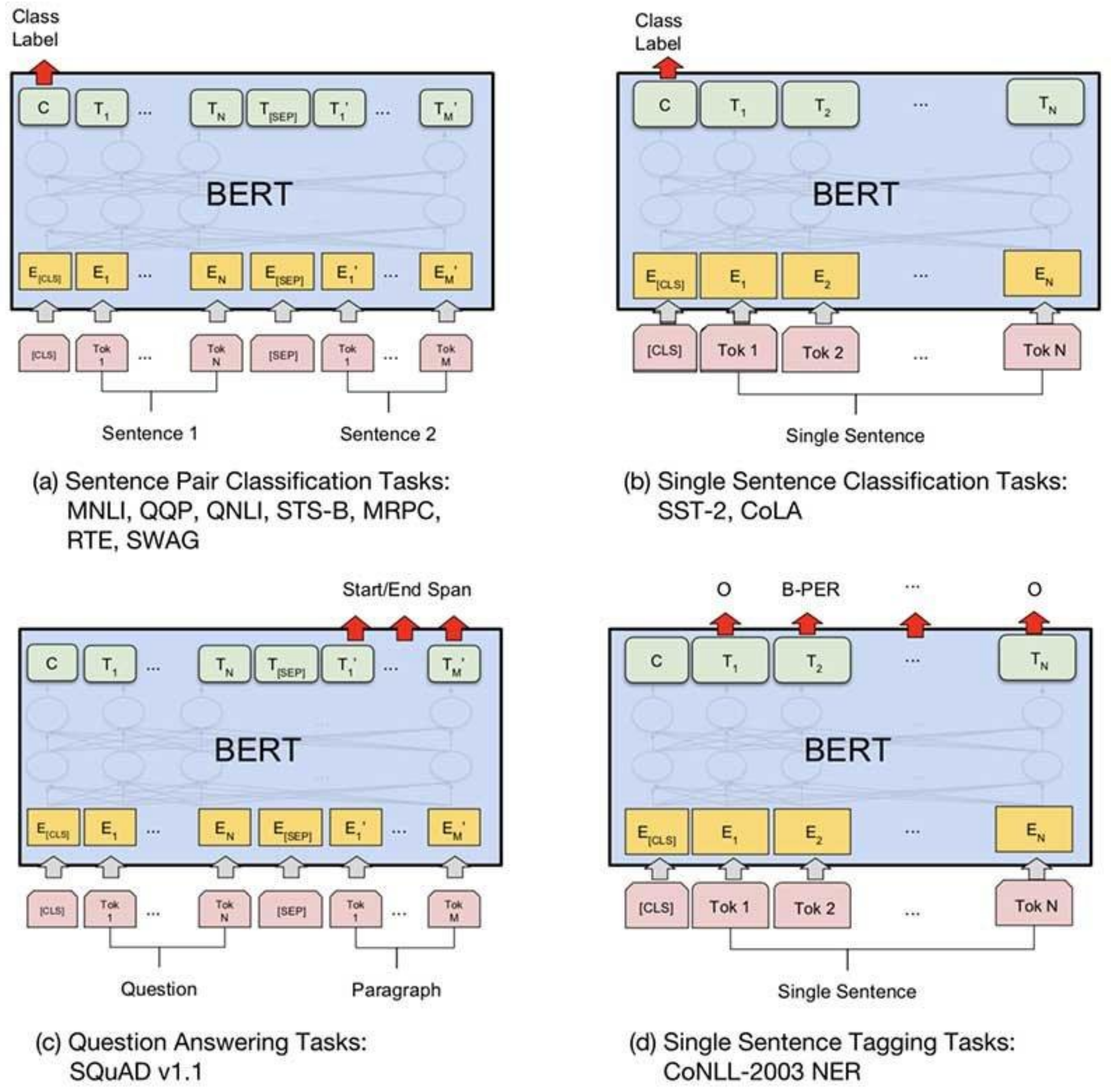


BERT fine-tuning

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).



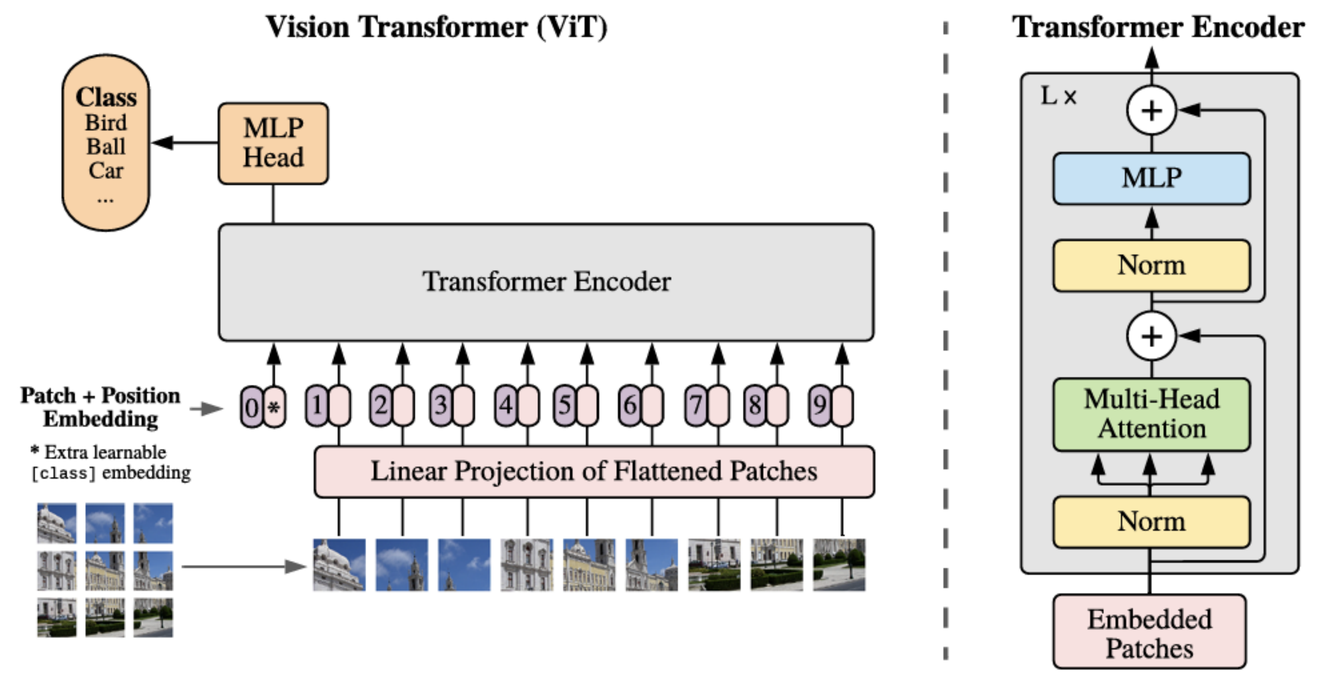
(a) Sentence Pair Classification Tasks: MNLI, QQP, QNLI, STS-B, MRPC, RTE, SWAG

(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 unsupervised 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.)

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, so prediciitng the current word is trivial.

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

Masked single-head self attention

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

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

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

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

Only lower-triangular components of A are nonzero.

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

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_ℓ is a linear combination of $v_1, \ldots, v_\ell$.

$\big(\exp(-\infty) = 0\big)$

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

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

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

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

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

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

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

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

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.

Self-supervised pre-training

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

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,

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

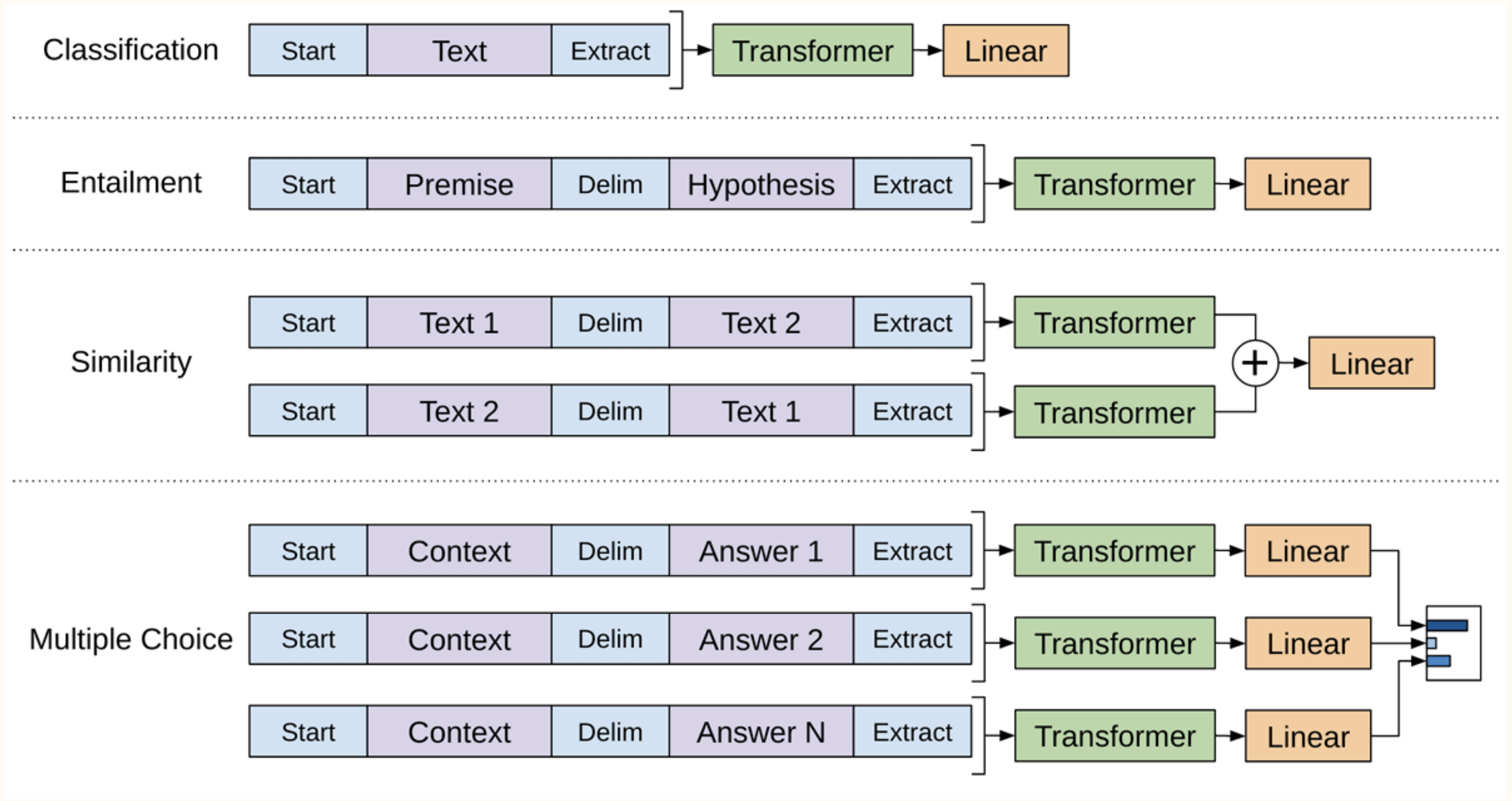
where ℓ^{CE} is the cross entropy loss.

Supervised fine-tuning

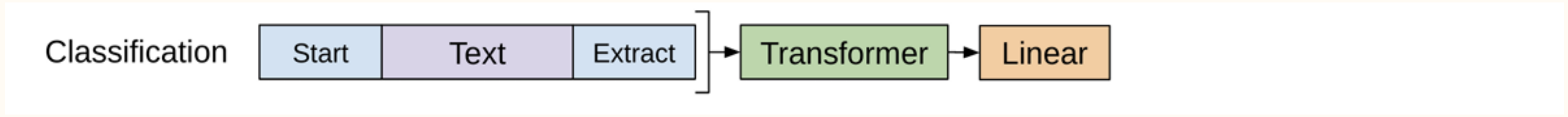
First, transform the relevant text into sequence with appropriate delimiter tokens.

At the end of the transformer, the token corresponding to the “extract” toke 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

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

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

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

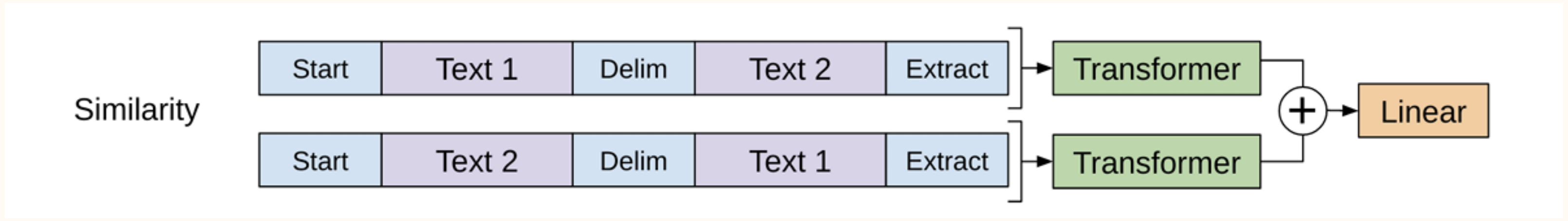
where A and b are the parameters of the linear layer and Y is the label corresponding to X, is used.

In all cases, only w_L is extracted to form the supervised fine-tuning loss.

(Note that 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.)

Supervised fine-tuning

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.



For similarity tasks, there is no inherent ordering of the two sentences being compared. So the transformer is given both orderings.

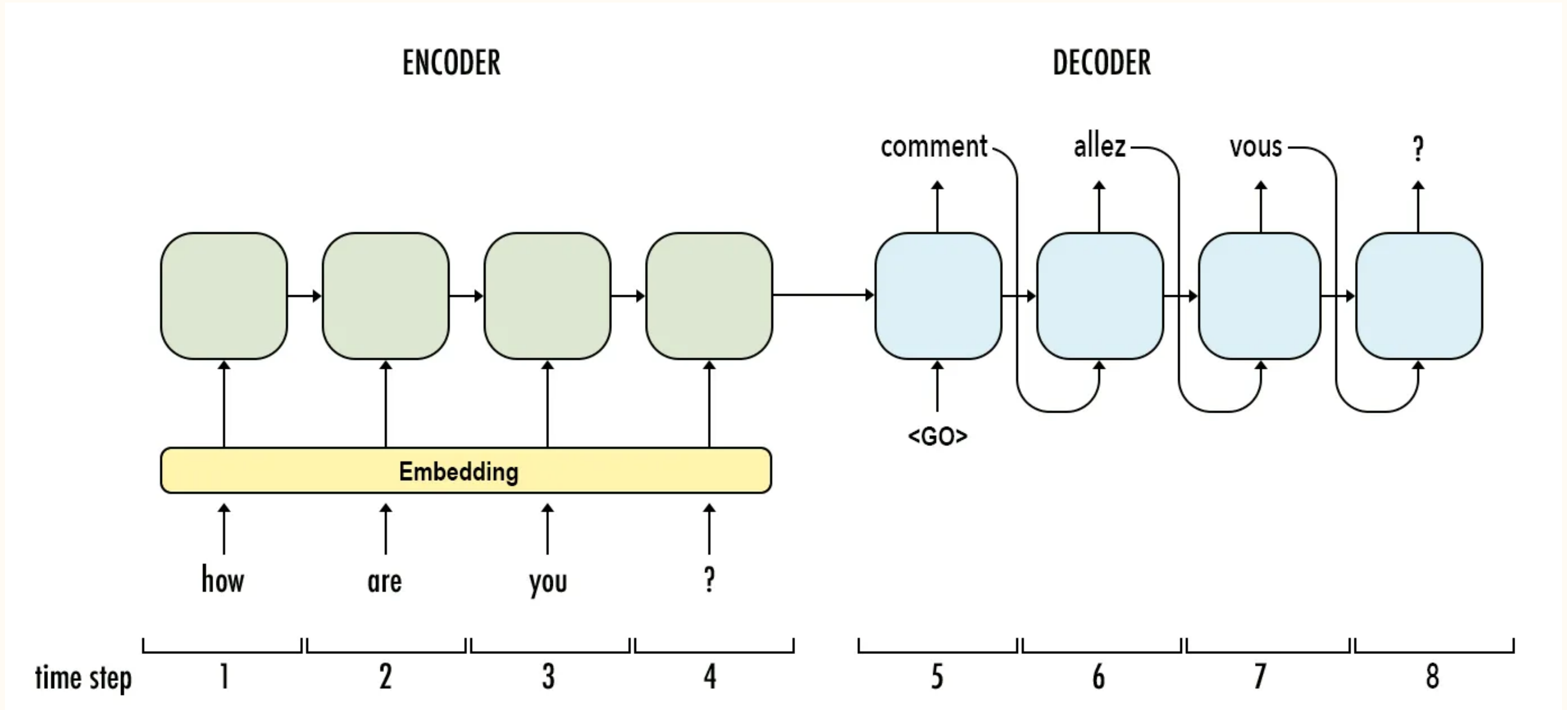
Example task: Machine translation

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.)

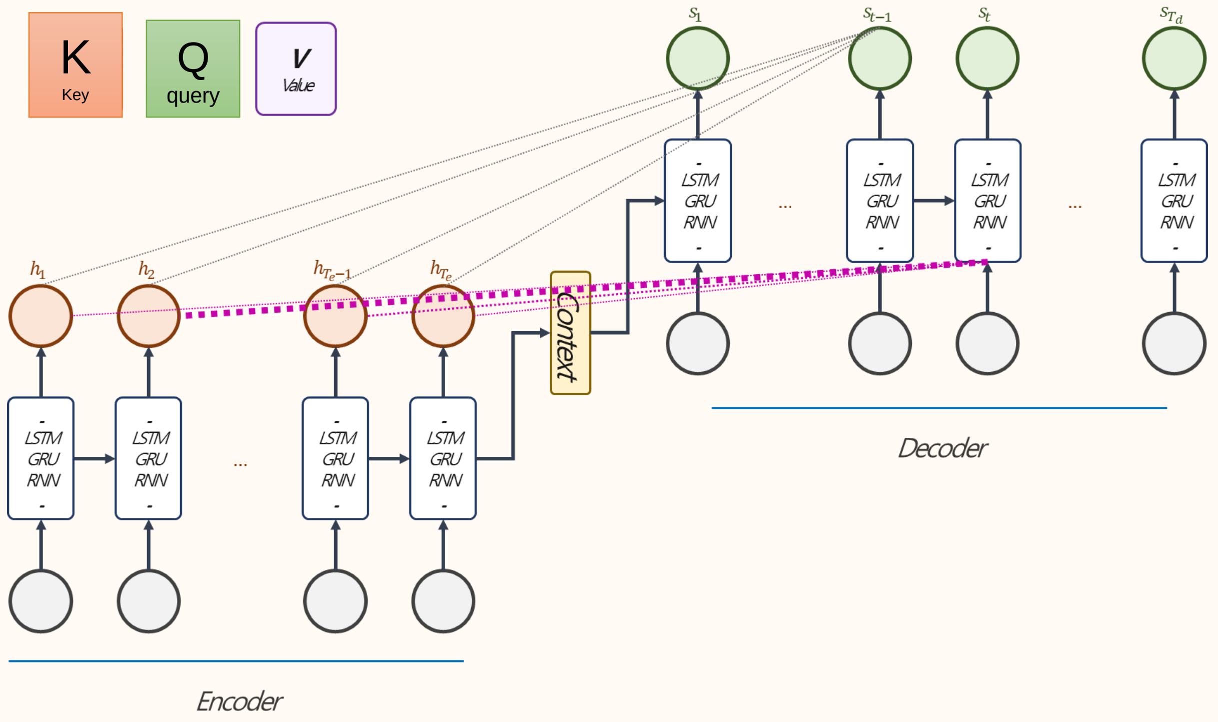


Bahdanau attention and cross attention

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.

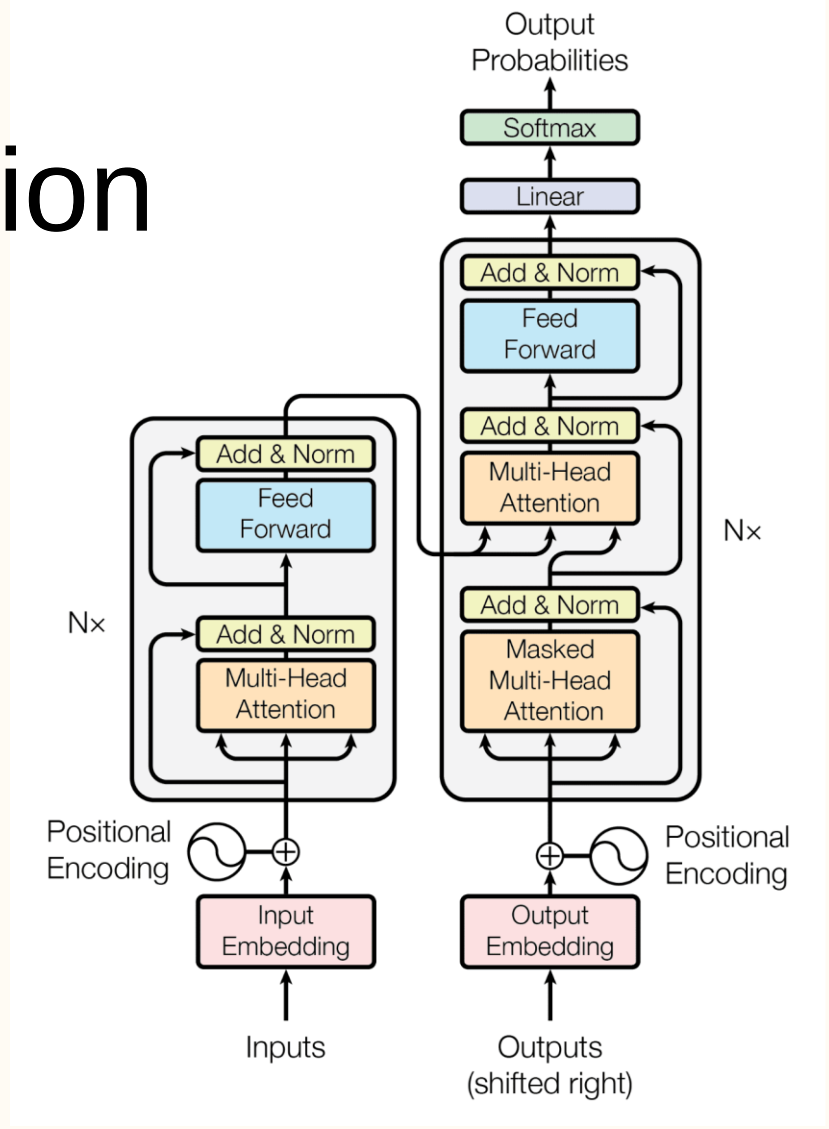
This attention mechanism is now called *cross attention*, and it is now commonly used to attend across different modalities. E.g. text decoder attends to image patches.



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 $\left\{x_\ell^{\mathrm{dec}}\right\}_{\ell=1}^{L}$ 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.)

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 reason that thing 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.