Neural Sequence Modeling

He He

NEW YORK UNIVERSITY

September 18, 2024

Logistics

- HW1 due this Friday at 12pm.
- HW2 will be released this Friday.

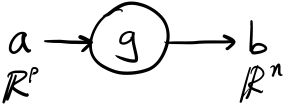
Table of Contents

Computation graphs

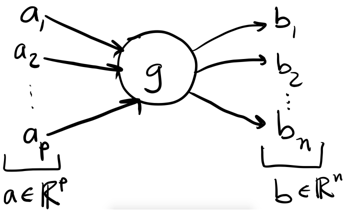
(adpated from David Rosenberg's slides)

Function as a *node* that takes in *inputs* and produces *outputs*.

- Typical computation graph:



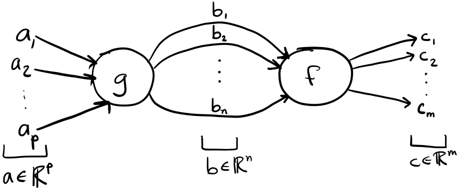
- Broken out into components:



Compose multiple functions

(adpated from David Rosenberg's slides)

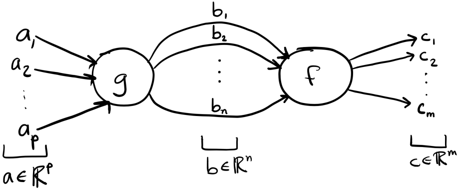
Compose two functions $g : \mathbb{R}^p \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^m$: $c = f(g(a))$



Compose multiple functions

(adpated from David Rosenberg's slides)

Compose two functions $g : \mathbb{R}^p \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^m$: $c = f(g(a))$

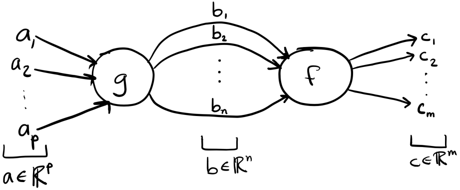


- Derivative: How does change in a_j affect c_i?

Compose multiple functions

(adpated from David Rosenberg's slides)

Compose two functions $g : \mathbb{R}^p \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^m$: $c = f(g(a))$



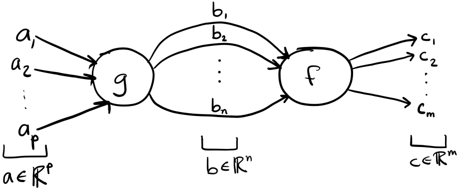
- Derivative: How does change in a_j affect c_i?

$$\frac{\partial c_i}{\partial a_j} = \sum_{k=1}^{n} \frac{\partial c_i}{\partial b_k} \frac{\partial b_k}{\partial a_j}.$$

Compose multiple functions

(adpated from David Rosenberg's slides)

Compose two functions $g : \mathbb{R}^p \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}^m$: $c = f(g(a))$



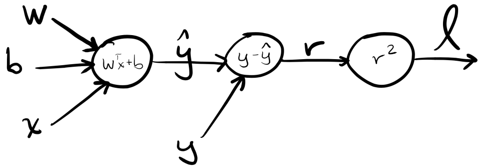
- Derivative: How does change in a_j affect c_i?

$$\frac{\partial c_i}{\partial a_j} = \sum_{k=1}^{n} \frac{\partial c_i}{\partial b_k} \frac{\partial b_k}{\partial a_j}.$$

- Visualize the multivariable **chain rule**:
  - Sum changes induced on all paths from a_j to c_i.
  - Changes on one path is the product of changes across each node.

Computation graph example

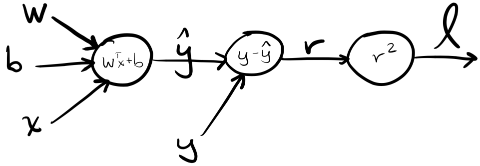
(adpated from David Rosenberg's slides)



(What is this graph computing?)

Computation graph example

(adpated from David Rosenberg's slides)



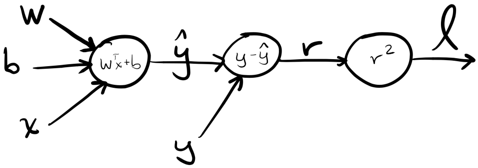
(What is this graph computing?)

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r$$

$$\frac{\partial \ell}{\partial w_j} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_j} = (-2r)\, x_j = -2r x_j$$

Computation graph example

(adpated from David Rosenberg's slides)



(What is this graph computing?)

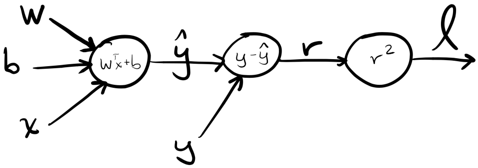
$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r$$

$$\frac{\partial \ell}{\partial w_j} = \frac{\partial \ell}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_j} = (-2r)\, x_j = -2rx_j$$

Computing the derivatives in certain order allows us to save compute!

Computation graph example

(adpated from David Rosenberg's slides)



(What is this graph computing?)

$$\frac{\partial \ell}{\partial r} = 2r$$

$$\frac{\partial \ell}{\partial \hat{y}} = \frac{\partial \ell}{\partial r}\frac{\partial r}{\partial \hat{y}} = (2r)(-1) = -2r$$

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r$$

$$\frac{\partial \ell}{\partial w_j} = \frac{\partial \ell}{\partial \hat{y}}\frac{\partial \hat{y}}{\partial w_j} = (-2r)x_j = -2rx_j$$

Computing the derivatives in certain order allows us to save compute!

Backpropogation

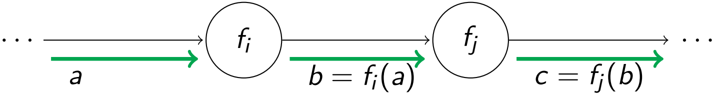
Backpropogation = chain rule + dynamic programming on a computation graph

Backpropogation

Backpropogation = chain rule + dynamic programming on a computation graph

Forward pass

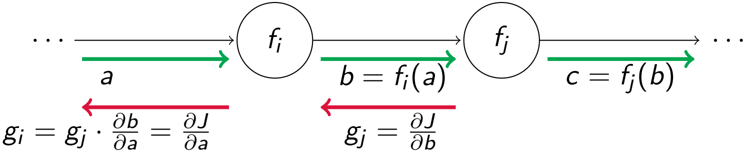
- **Topological order**: every node appears before its children
- For each node, compute the output given the input (from its parents).



Backpropogation

Backward pass

- **Reverse topological order**: every node appear after its children
- For each node, compute the partial derivative of its output w.r.t. its input, multiplied by the partial derivative from its children (chain rule).



Summary

Key idea in neural nets: feature/representation learning

Building blocks:

- Input layer: raw features (no learnable parameters)
- Hidden layer: perceptron + nonlinear activation function
- Output layer: linear (+ transformation, e.g. softmax)

Optimization:

- Optimize by SGD (implemented by back-propogation)
- Objective is non-convex, may not reach a global minimum

Table of Contents

Overview

Problem setup: given an input sequence, come up with a (neural network) model that outputs a representation of the sequence for downstream tasks (e.g., classification)

Overview

Problem setup: given an input sequence, come up with a (neural network) model that outputs a representation of the sequence for downstream tasks (e.g., classification)

Key challenge: how to model interaction among words?

Overview

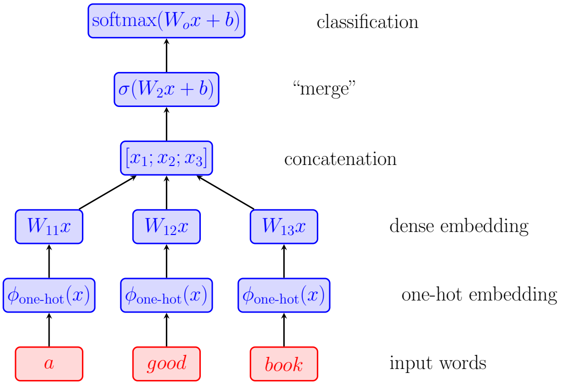
Problem setup: given an input sequence, come up with a (neural network) model that outputs a representation of the sequence for downstream tasks (e.g., classification)

Key challenge: how to model interaction among words?

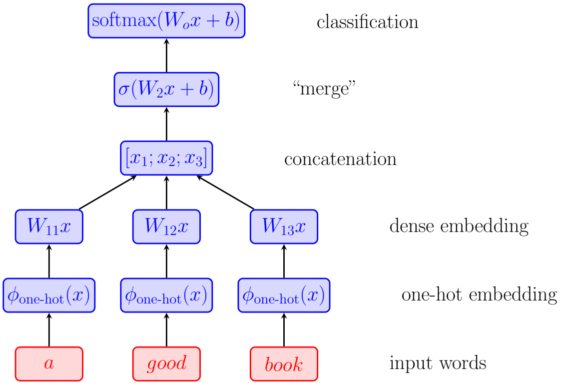
Approach:

- Aggregation (pooling word embeddings)
- Recurrence
- Self-attention

Feed-forward neural network for text classification



Feed-forward neural network for text classification



🤔 Where is the interaction between words modeled?
How to adapt the network to handle sequences with arbitrary length?

Recurrent neural networks

- **Goal**: compute representation of sequence $x_{1:T}$ of varying lengths
- **Idea**: combine new symbols with previous symbols recurrently

Recurrent neural networks

- **Goal**: compute representation of sequence $x_{1:T}$ of varying lengths
- **Idea**: combine new symbols with previous symbols recurrently
  - Update the representation, i.e. **hidden states** h_t, recurrently

$$h_t = f(h_{t-1}, x_t)$$

    - Output from previous time step is the input to the current time step
    - Apply the same transformation f at each time step

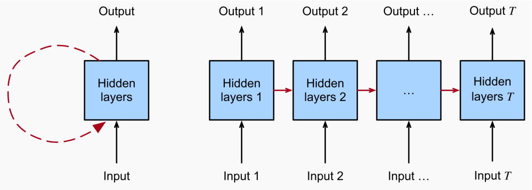
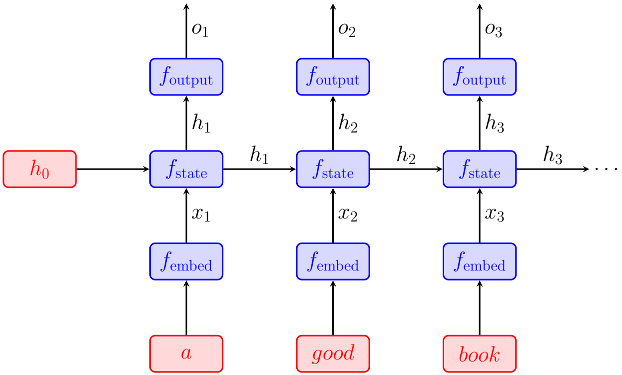


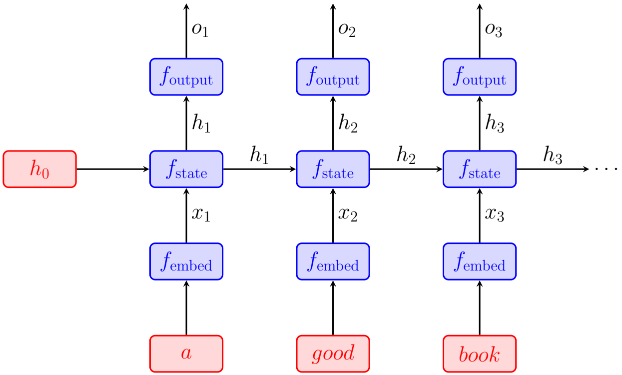
Figure: 9.1 from d2l.ai

Forward pass



A deep neural network with shared weights in each layer

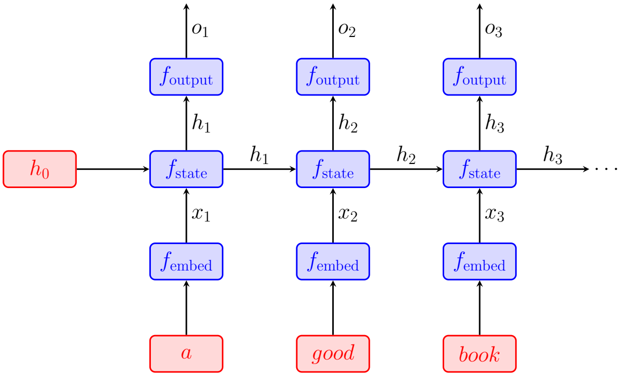
Forward pass



$$\begin{aligned} x_t &= f_{\text{embed}}(s_t) \\ &= W_e \phi_{\text{one-hot}}(s_t) \end{aligned}$$

A deep neural network with shared weights in each layer

Forward pass

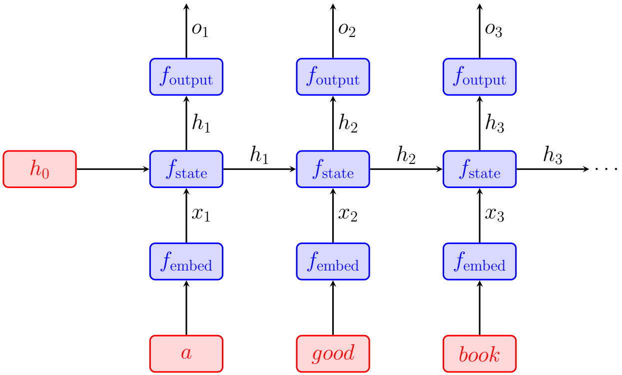


$$
\begin{aligned}
x_t &= f_{\text{embed}}(s_t) \\
&= W_e \phi_{\text{one-hot}}(s_t)
\end{aligned}
$$

$$
\begin{aligned}
h_t &= f_{\text{state}}(x_t, h_{t-1}) \\
&= \sigma(W_{hh} h_{t-1} + W_{ih} x_t + b_h)
\end{aligned}
$$

A deep neural network with shared weights in each layer

Forward pass



$$
\begin{aligned}
x_t &= f_{\text{embed}}(s_t) \\
&= W_e \phi_{\text{one-hot}}(s_t)
\end{aligned}
$$

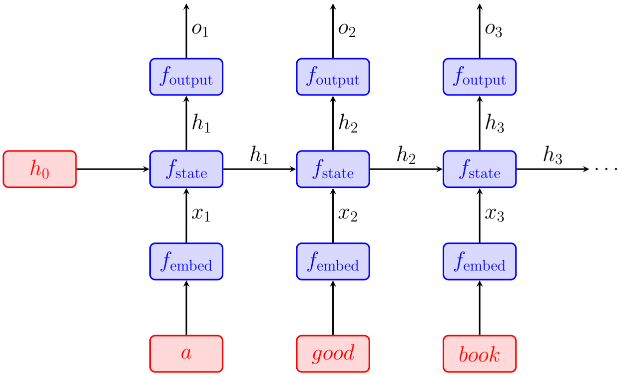
$$
\begin{aligned}
h_t &= f_{\text{state}}(x_t, h_{t-1}) \\
&= \sigma(W_{hh} h_{t-1} + W_{ih} x_t + b_h)
\end{aligned}
$$

$$
\begin{aligned}
o_t &= f_{\text{output}}(h_t) \\
&= \mathrm{softmax}(W_{ho} h_t + b_o)
\end{aligned}
$$

(a distribution over classes)

A deep neural network with shared weights in each layer

Forward pass



A deep neural network with shared weights in each layer

$$x_t = f_{\text{embed}}(s_t)$$
$$= W_e \phi_{\text{one-hot}}(s_t)$$

$$h_t = f_{\text{state}}(x_t, h_{t-1})$$
$$= \sigma(W_{hh} h_{t-1} + W_{ih} x_t + b_h)$$

$$o_t = f_{\text{output}}(h_t)$$
$$= \text{softmax}(W_{ho} h_t + b_o)$$

(a distribution over classes)

🤔 Which computation can be parallelized?

Loss functions on RNNs

Sequence labeling and language modeling:

- Input: $x_1, \ldots, x_T$ (a sequence of tokens)
- Output: $y_1, \ldots, y_T$ (e.g., POS tags, next words)
- Loss function: $\sum_{i=1}^{T} \ell(y_t, o_t)$
    - NLL loss: $\sum_{i=1}^{T} - \log o_t[y_t]$

Loss functions on RNNs

Sequence labeling and language modeling:

- Input: $x_1, \ldots, x_T$ (a sequence of tokens)
- Output: $y_1, \ldots, y_T$ (e.g., POS tags, next words)
- Loss function: $\sum_{i=1}^{T} \ell(y_t, o_t)$
  - NLL loss: $\sum_{i=1}^{T} - \log o_t[y_t]$

Text classification:

- Input: $x_1, \ldots, x_T$
- Output: $y \in \{1, \ldots, K\}$ (K classes)
- Loss function: $\ell(y, f_{\text{output}}(\mathrm{pool}(h_1, \ldots, h_T)))$
  - Can use last hidden state or mean of all hidden states

Backward pass

Given the loss $\ell(y_t, o_t)$, compute the gradient with respect to W_{hh}.

$$\frac{\partial \ell_t}{\partial W_{hh}} =$$

Backward pass

Given the loss $\ell(y_t, o_t)$, compute the gradient with respect to W_{hh}.

$$\frac{\partial \ell_t}{\partial W_{hh}} = \frac{\partial \ell_t}{\partial o_t} \frac{\partial o_t}{\partial h_t} \frac{\partial h_t}{\partial W_{hh}}$$

Backward pass

Given the loss $\ell(y_t, o_t)$, compute the gradient with respect to W_{hh}.

$$\frac{\partial \ell_t}{\partial W_{hh}} = \frac{\partial \ell_t}{\partial o_t} \frac{\partial o_t}{\partial h_t} \frac{\partial h_t}{\partial W_{hh}}$$

Computation graph of h_t: $h_t = \sigma(W_{hh} h_{t-1} + W_{hi} x_t + b)$

Backpropagation through time

Problem with standard backpropagation:

- Gradient involves repeated multiplication of W_{hh}
- Gradient will vanish / explode (depending on the eigenvalues of W_{hh})

Backpropagation through time

Problem with standard backpropagation:

- Gradient involves repeated multiplication of W_{hh}
- Gradient will vanish / explode (depending on the eigenvalues of W_{hh})

Quick fixes:

- Reduce the number of repeated multiplication: truncate after k steps (h_{t-k} has no influence on h_t)
- Limit the norm (or value) of the gradient in each step: gradient clipping (can only mitigate explosion)

Long-short term memory (LSTM)

Vanilla RNN: always update the hidden state

- Cannot handle long range dependency due to gradient vanishing

Long-short term memory (LSTM)

Vanilla RNN: always update the hidden state

- Cannot handle long range dependency due to gradient vanishing

LSTM: learn when to update the hidden state

- First successful solution to the gradient vanishing and explosion problem

Long-short term memory (LSTM)

Vanilla RNN: always update the hidden state

- Cannot handle long range dependency due to gradient vanishing

LSTM: learn when to update the hidden state

- First successful solution to the gradient vanishing and explosion problem

Key idea is to use a **gating mechanism**: multiplicative weights that modulate another variable

- How much should the new input affect the state?
- When to ignore new inputs?
- How much should the state affect the output?

Long-short term memory (LSTM) parametrization

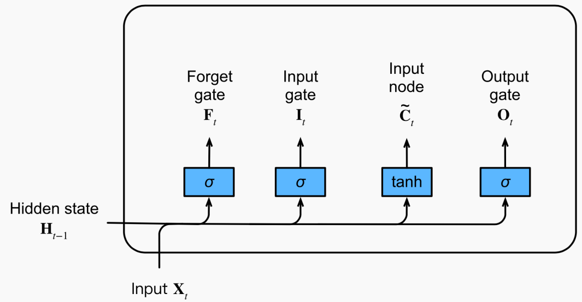


Figure: 10.1.2 from d2l.ai

Update with the new input x_t (same as in vanilla RNN)

$$\tilde{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \quad \text{new cell content}$$

Long-short term memory (LSTM) parametrization

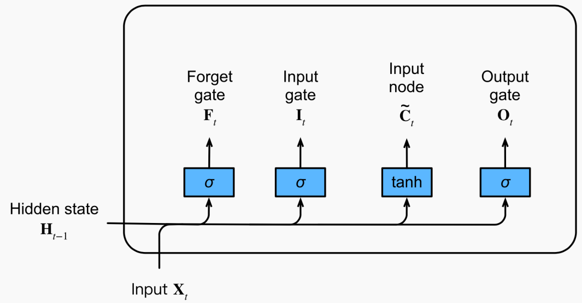


Figure: 10.1.2 from d2l.ai

Update with the new input x_t (same as in vanilla RNN)

$$\tilde{c}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c) \quad \text{new cell content}$$

Should we update with the new input x_t?

Long-short term memory (LSTM) parametrization

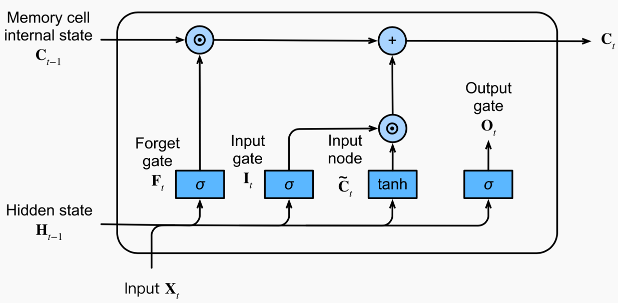


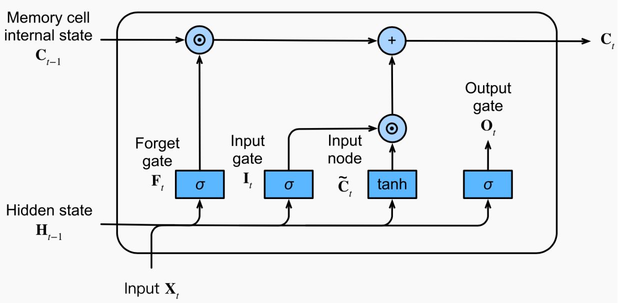
Figure: 10.1.3 from d2l.ai

Choose between $\tilde{c}_t$ (update) and c_{t-1} (no update): ($\odot$: elementwise product)

$$\textbf{memory cell} \quad c_t = i_t \odot \tilde{c}_t + f_t \odot c_{t-1}$$

- f_t: proportion of the old state (preserve $\uparrow$ or erase $\downarrow$ the old memory)
- i_t: proportion of the new state (write $\uparrow$ or ignore $\downarrow$ the new input)
- What is c_t if $f_t = 1$ and $i_t = 0$?

Long-short term memory (LSTM) parametrization



Input gate and forget gate depends on data:

$$i_t = \text{sigmoid}(W_{xi} x_t + W_{hi} h_{t-1} + b_i) \ ,$$
$$f_t = \text{sigmoid}(W_{xf} x_t + W_{hf} h_{t-1} + b_f) \ .$$

Each coordinate is between 0 and 1.

Long-short term memory (LSTM) parametrization

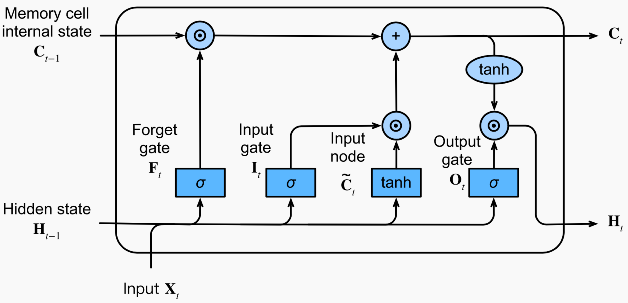


Figure: 10.1.4 from d2l.ai

How much should the memory cell state influence the rest of the network:

$$h_t = o_t \odot c_t$$

$$o_t = \text{sigmoid}(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

c_t may accumulate information without impact the network if o_t is close to 0

How does LSTM solve gradient vanishing / explosion?

Intuition: gating allows the network to learn to control how much gradient should vanish.

- Vanilla RNN: gradient depends on repeated multiplication of the same weight matrix
- LSTM: gradient depends on repeated multiplication of some quantity that depends on the data (values of input and forget gates)
- So the network can learn to reset or update the gradient depending on whether there is long-range dependencies in the data.

Table of Contents

Improve the efficiency of RNN

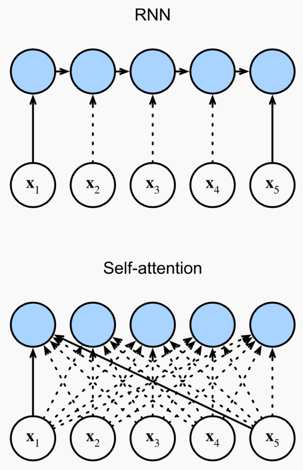


Figure: 11.6.1 from d2l.ai

Recall that our goal is to come up with a good respresentation of a sequence of words.

RNN:

- Past words influence the sentence representation through recurrent update
- Sequential computation $O(\text{sequence length})$, hard to scale

Improve the efficiency of RNN

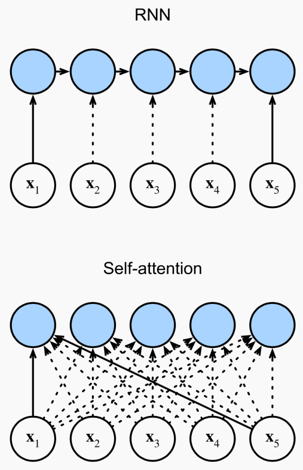


Figure: 11.6.1 from d2l.ai

Recall that our goal is to come up with a good respresentation of a sequence of words.

RNN:

- Past words influence the sentence representation through recurrent update
- Sequential computation O(sequence length), hard to scale

Can we handle dependency more efficiently?

- Direct interaction between any pair of words in the sequence
- Parallelizable computation

Model interaction between two variables

Given a **query** and some context, how do we find relevant information from the context?

Context representation:

- **values**: content in the context
- **keys**: matching agaisnt the query to compute relevance

Model interaction between two variables

Given a **query** and some context, how do we find relevant information from the context?

Context representation:

- **values**: content in the context
- **keys**: matching agaisnt the query to compute relevance

Example:

- Reading comprehension: query=question, context=document
- Machine translation: query=text in French, context=text in English

Model interaction between words

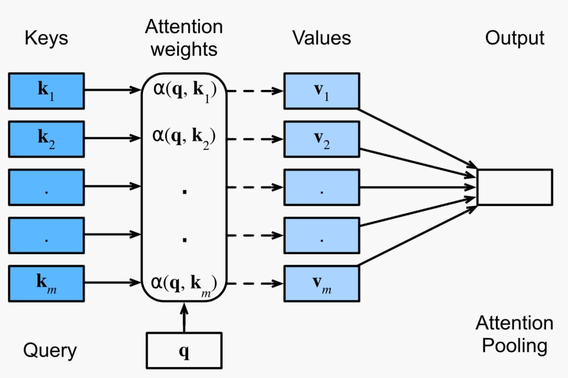


Figure: 11.1.1 from d2l.ai

- **Attention weights** $\alpha(q, k_i)$: how strong is q matched to k_i
- **Attention pooling**: combine v_i's according to their "relatedness" to the query

Model interaction between words using a "soft" database

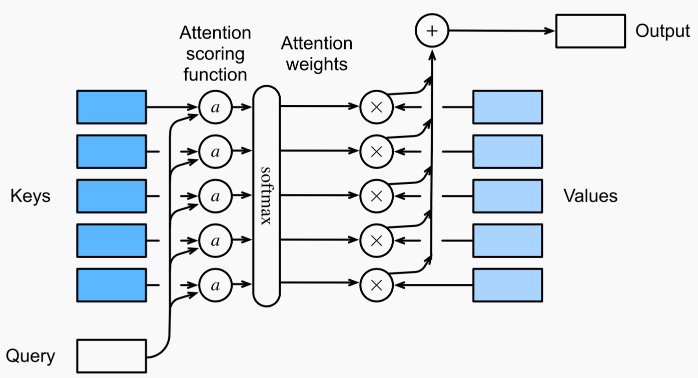


Figure: 11.3.1 from d2l.ai

- Model attention weights as a distribution: $\alpha = \mathrm{softmax}(a(q, k_1), \ldots, a(q, k_m))$
- Output a weighted combination of values: $o_i = \sum_{i=1}^{m} \alpha(q, k_i) v_i$

Self-attention

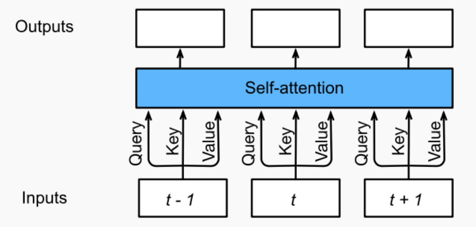
Given two sets of objects (queries and context), attention allows us to model interactions between them.

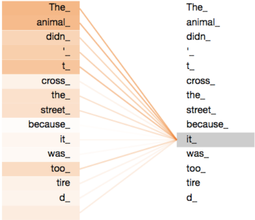
We can use it to model the interaction between each pair of words in a sentence.

Self-attention

Given two sets of objects (queries and context), attention allows us to model interactions between them.

We can use it to model the interaction between each pair of words in a sentence.



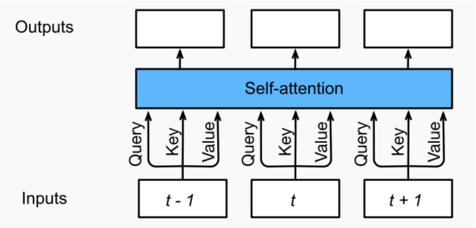


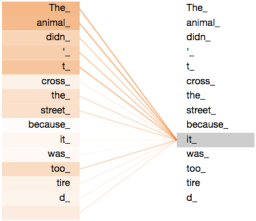
- Input: map each symbol to a query, a key, and a value (embeddings)

Self-attention

Given two sets of objects (queries and context), attention allows us to model interactions between them.

We can use it to model the interaction between each pair of words in a sentence.



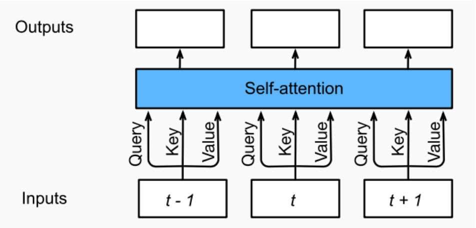


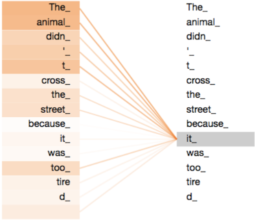
- Input: map each symbol to a query, a key, and a value (embeddings)
- Attend: each word (as a query) interacts with all words (keys)

Self-attention

Given two sets of objects (queries and context), attention allows us to model interactions between them.

We can use it to model the interaction between each pair of words in a sentence.





- Input: map each symbol to a query, a key, and a value (embeddings)
- Attend: each word (as a query) interacts with all words (keys)
- Output: *contextualized* representation of each word (weighted sum of values)

Attention scoring functions

Design the function that measures relatedness between queries and keys:

$\alpha = \mathrm{softmax}(a(q, k_1), \ldots, a(q, k_m)) \qquad a \colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$

Attention scoring functions

Design the function that measures relatedness between queries and keys:

$\alpha = \mathrm{softmax}(a(q, k_1), \ldots, a(q, k_m)) \qquad a \colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$

Dot-product attention

$$a(q, k) = q \cdot k$$

Attention scoring functions

Design the function that measures relatedness between queries and keys:

$\alpha = \mathrm{softmax}(a(q, k_1), \ldots, a(q, k_m)) \qquad a : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$

Dot-product attention

$$a(q, k) = q \cdot k$$

Scaled dot-product attention

$$a(q, k) = q \cdot k / \sqrt{d}$$

- $\sqrt{d}$: dimension of the key vector
- Avoids large attention weights that push the softmax function into regions of small gradients

Attention scoring functions

Design the function that measures relatedness between queries and keys:

$\alpha = \mathrm{softmax}(a(q, k_1), \ldots, a(q, k_m)) \qquad a \colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$

Dot-product attention

$$a(q, k) = q \cdot k$$

Scaled dot-product attention

$$a(q, k) = q \cdot k / \sqrt{d}$$

- $\sqrt{d}$: dimension of the key vector
- Avoids large attention weights that push the softmax function into regions of small gradients

MLP attention

$$a(q, k) = u^T \tanh(W[q; k])$$

Multi-head attention: motivation

Time flies like an arrow

- Each word attends to all other words in the sentence
- Which words should “like” attend to?

Multi-head attention: motivation

Time flies like an arrow

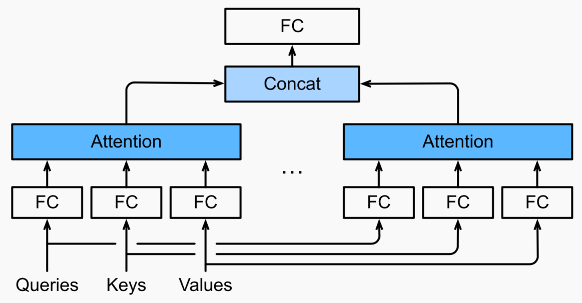
- Each word attends to all other words in the sentence
- Which words should “like” attend to?
  - Syntax: “flies”, “arrow” (a preposition)
  - Semantics: “time”, “arrow” (a metaphor)

Multi-head attention: motivation

Time flies like an arrow

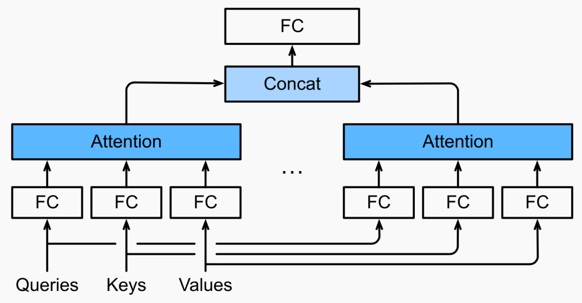
- Each word attends to all other words in the sentence
- Which words should "like" attend to?
  - Syntax: "flies", "arrow" (a preposition)
  - Semantics: "time", "arrow" (a metaphor)
- We want to represent different roles of a word in the sentence: need more than a single embedding
- Instantiation: multiple self-attention modules

Multi-head attention



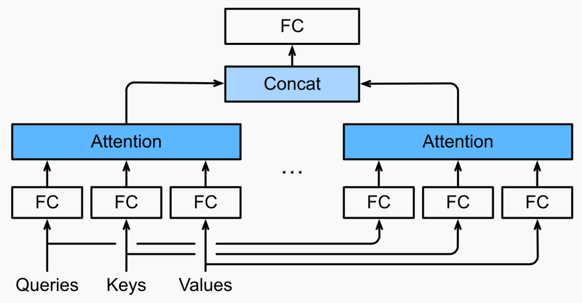
- Multiple attention modules: same architecture, different parameters

Multi-head attention



- Multiple attention modules: same architecture, different parameters
- A **head**: one set of attention outputs

Multi-head attention



- Multiple attention modules: same architecture, different parameters
- A **head**: one set of attention outputs
- Concatenate all heads (increased output dimension)
- Linear projection to produce the final output

Matrix representation: input mapping

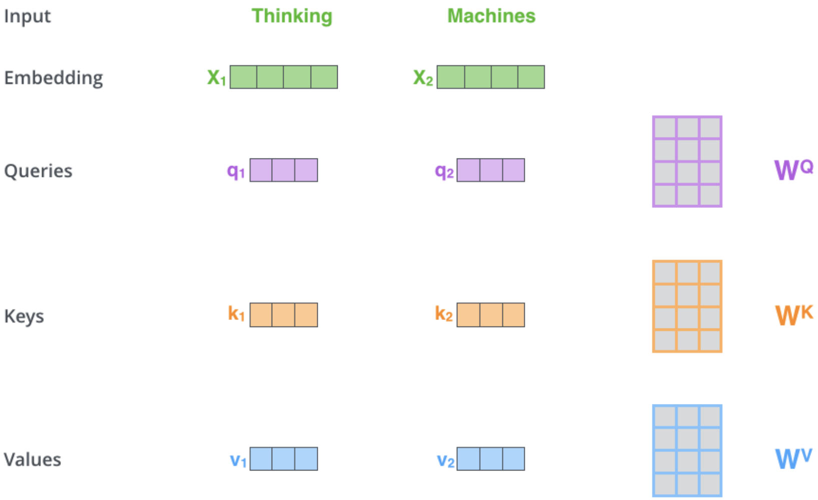


Figure: From The Illustrated Transformer

Matrix representation: attention weights

Scaled dot product attention

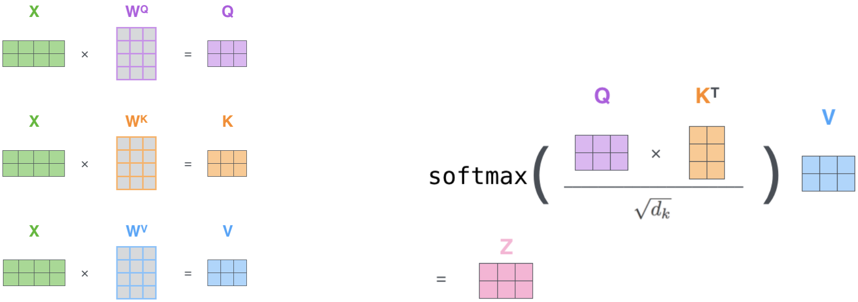


Figure: From The Illustrated Transformer

Multi-head attention

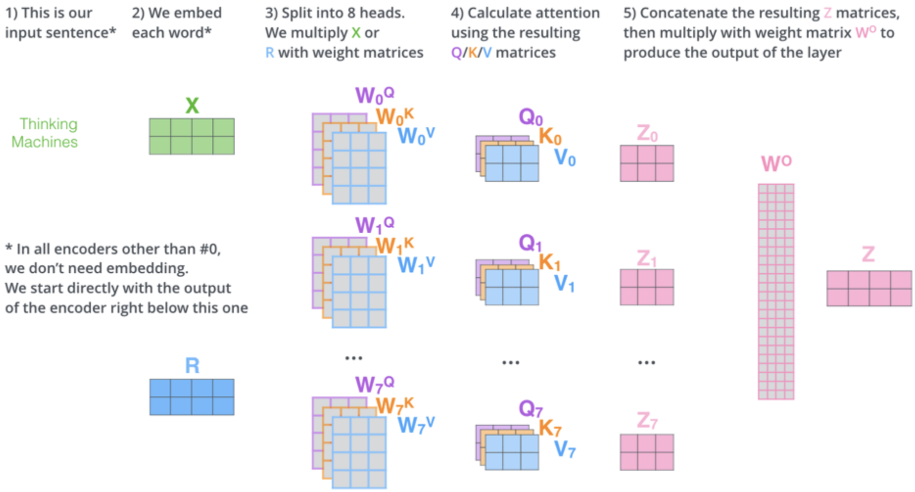


Figure: From The Illustrated Transformer

Summary so far

- Sequence modeling
  - Input: a sequence of words
  - Output: a sequence of contextualized embeddings for each word
  - Models interaction among words

Summary so far

- Sequence modeling
  - Input: a sequence of words
  - Output: a sequence of contextualized embeddings for each word
  - Models interaction among words
- Building blocks
  - Feed-forward / fully-connected neural network
  - Recurrent neural network
  - Self-attention

Summary so far

- Sequence modeling
  - Input: a sequence of words
  - Output: a sequence of contextualized embeddings for each word
  - Models interaction among words
- Building blocks
  - Feed-forward / fully-connected neural network
  - Recurrent neural network
  - Self-attention

🤔 Which of these can handle sequences of arbitrary length?

Table of Contents

Overview

- Use self-attention as the core building block
- Vastly increased scalability (model and data size) compared to recurrence-based models
- Initially designed for machine translation (next week)
  - *Attention is all you need*. Vaswani et al., 2017.
- The backbone of today's large-scale models
- Extended to non-sequential data (e.g., images and molecules)

Transformer block

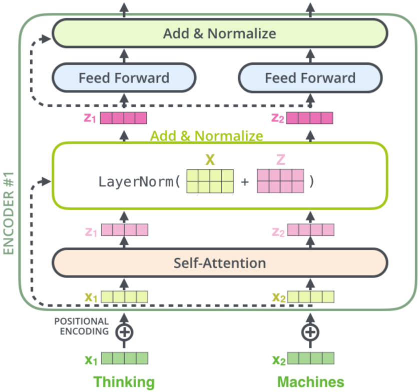


Figure: From The Illustrated Transformer

- Multi-head self-attention

Transformer block

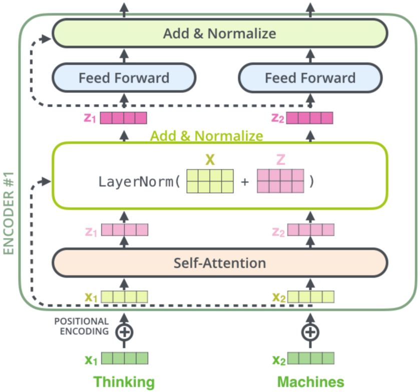


Figure: From The Illustrated Transformer

- Multi-head self-attention
  - Capture dependence among input symbols

Transformer block

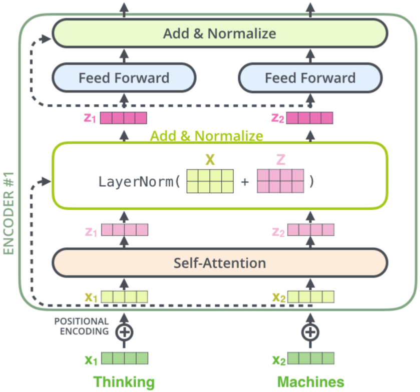


Figure: From The Illustrated Transformer

- Multi-head self-attention
  - Capture dependence among input symbols
- Positional encoding

Transformer block

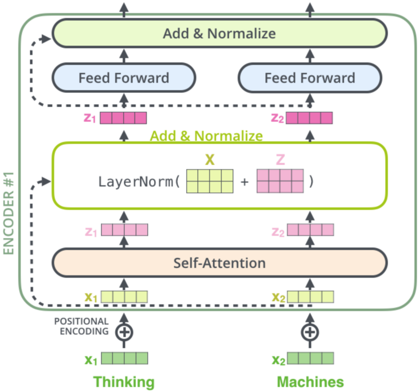


Figure: From The Illustrated Transformer

- Multi-head self-attention
  - Capture dependence among input symbols
- Positional encoding
  - Capture the order of symbols

Transformer block

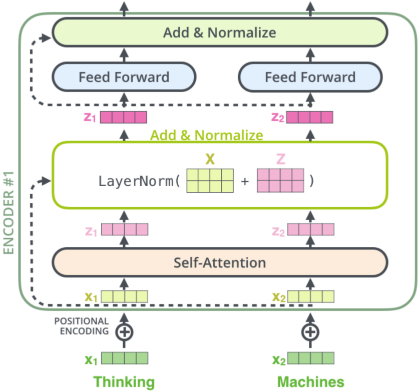


Figure: From The Illustrated Transformer

- Multi-head self-attention
  - Capture dependence among input symbols
- Positional encoding
  - Capture the order of symbols
- Residual connection and layer normalization

Transformer block

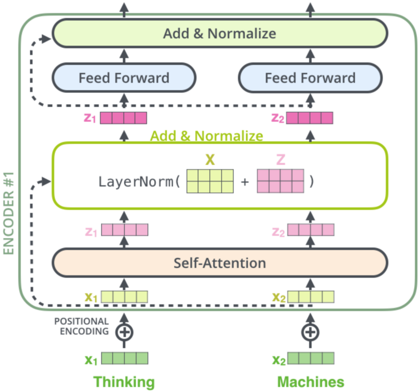
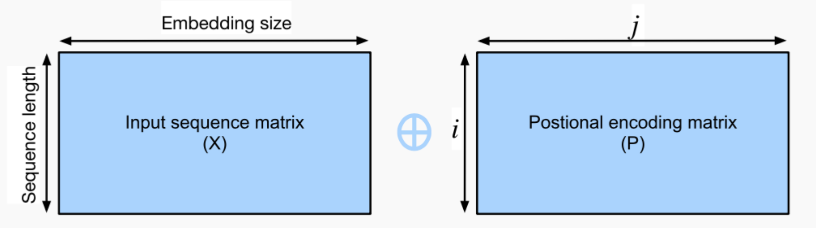


Figure: From The Illustrated Transformer

- Multi-head self-attention
  - Capture dependence among input symbols
- Positional encoding
  - Capture the order of symbols
- Residual connection and layer normalization
  - More efficient and stable optimization

Position embedding

Motivation: model word order in the input sequence
Solution: add a position embedding to each word



Position embedding:

- Encode absolute and relative positions of a word
- Same dimension as word embeddings
- Learned or deterministic

Sinusoidal position embedding

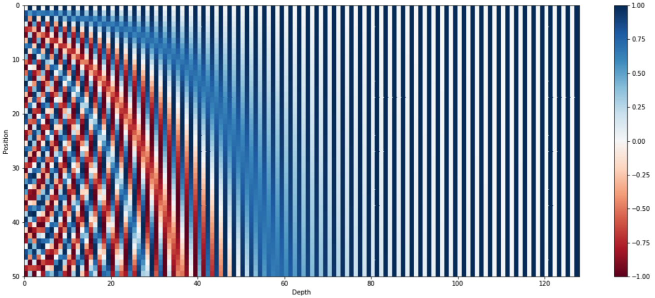
Intuition: continuous approximation of binary encoding of positions (integers)

0 :	0	0	0	0
1 :	0	0	0	1
2 :	0	0	1	0
3 :	0	0	1	1
4 :	0	1	0	0
5 :	0	1	0	1
6 :	0	1	1	0
7 :	0	1	1	1

Sinusoidal position embedding

Intuition: continuous approximation of binary encoding of positions (integers)

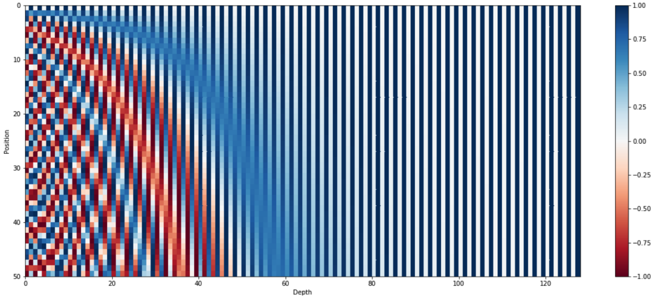
0 :	0	0	0	0
1 :	0	0	0	1
2 :	0	0	1	0
3 :	0	0	1	1
4 :	0	1	0	0
5 :	0	1	0	1
6 :	0	1	1	0
7 :	0	1	1	1



Sinusoidal position embedding

Intuition: continuous approximation of binary encoding of positions (integers)

$0:$	0	0	0	0
$1:$	0	0	0	1
$2:$	0	0	1	0
$3:$	0	0	1	1
$4:$	0	1	0	0
$5:$	0	1	0	1
$6:$	0	1	1	0
$7:$	0	1	1	1



$$\vec{p_t} = \begin{bmatrix} \sin(\omega_1 . t) \\ \cos(\omega_1 . t) \\ \\ \sin(\omega_2 . t) \\ \cos(\omega_2 . t) \\ \\ \vdots \\ \\ \sin(\omega_{d/2} . t) \\ \cos(\omega_{d/2} . t) \end{bmatrix}_{d \times 1}$$

Figure: From Amirhossein Kazemnejad's Blog

$$\omega_k = 1/10000^{\frac{2k}{d}}$$

Learned position embeddings

Sinusoidal position embedding:

- Not learnable
- Can extrapolate to longer sequences but doesn't work well

Learned position embeddings

Sinusoidal position embedding:

- Not learnable
- Can extrapolate to longer sequences but doesn't work well

Learned absolute position embeddings (most common now):

- Consider each position as a word. Map positions to dense vectors: $W_{d \times n}\phi_{\text{one-hot}}(\text{pos})$
- Column i of W is the embedding of position i

Learned position embeddings

Sinusoidal position embedding:

- Not learnable
- Can extrapolate to longer sequences but doesn't work well

Learned absolute position embeddings (most common now):

- Consider each position as a word. Map positions to dense vectors: $W_{d \times n} \phi_{\text{one-hot}}(\text{pos})$
- Column i of W is the embedding of position i
- Need to fix maximum position/length beforehand
- Cannot extrapolate to longer sequences

Residual connection

Motivation:

- Gradient explosion/vanishing is not RNN-specific!
- It happens to all very deep networks (which are hard to optimize).

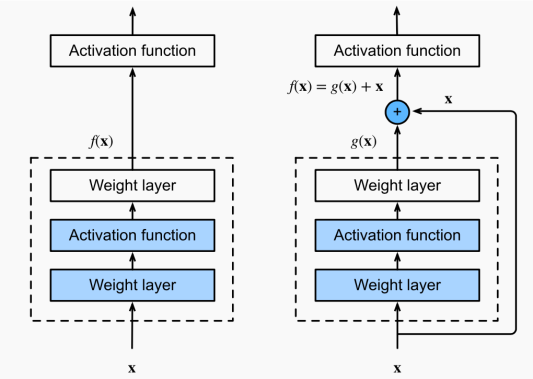
Residual connection

Motivation:

- Gradient explosion/vanishing is not RNN-specific!
- It happens to all very deep networks (which are hard to optimize).
- In principle, a deep network can always represent a shallow network (by setting higher layers to identity functions), thus it should be at least as good as the shallow network.
- For some reason, deep neural networks are bad at learning identity functions.
- How can we make it easier to recover the shallow solution?

Residual connection

Solution: Deep Residual Learning for Image Recognition [He et al., 2015]



Without residual connection: learn $f(x) = x$.

With residual connection: learn $g(x) = 0$ (easier).

Layer normalization

- **Problem**: inputs of a layer may shift during training
- **Solution**: normalize (zero mean, unit variance) across features [Ba et al., 2016]
- Let $x = (x_1, \ldots, x_d)$ be the input vector (e.g., word embedding, previous layer output)

$$\mathrm{LayerNorm}(x) = \frac{x - \hat{\mu}}{\hat{\sigma}},$$

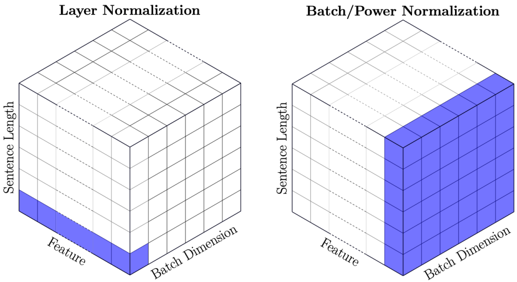
$$\text{where } \hat{\mu} = \frac{1}{d}\sum_{i=1}^{d} x_i, \quad \hat{\sigma}^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \hat{\mu})^2$$

Layer normalization

- **Problem**: inputs of a layer may shift during training
- **Solution**: normalize (zero mean, unit variance) across features [Ba et al., 2016]
- Let $x = (x_1, \ldots, x_d)$ be the input vector (e.g., word embedding, previous layer output)

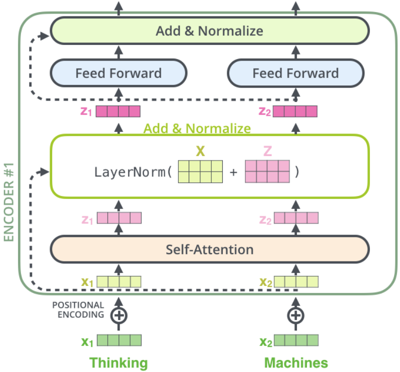
$$\mathrm{LayerNorm}(x) = \frac{x - \hat{\mu}}{\hat{\sigma}},$$

$$\text{where } \hat{\mu} = \frac{1}{d}\sum_{i=1}^{d} x_i, \quad \hat{\sigma}^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \hat{\mu})^2$$



- A deterministic transformation of the input
- Independent of train/inference and batch size

Residual connection and layer normalization in Transformer



- Add (residual connection) & Normalize (layer normalization) after each layer
- Position-wise feed-forward networks: same mapping for all positions

Summary

- We have seen two families of models for sequences modeling: **RNNs** and **Transformers**
- Both take a sequence of (discrete) symbols as input and output a sequence of embeddings
- They are often called **encoders** and are used to represent text
- Transformers are dominating today because of its scalability