

Neural Sequence Modeling

He He



NEW YORK UNIVERSITY

February 7, 2023

Logistics

- HW1 due this Friday.
- Lecture (65 min)
- Section on Pytorch and HPC (50 min)

Table of Contents

Neural networks basics

Recurrent neural networks

Self-attention

Tranformer

Feature learning

Linear predictor with **handcrafted features**: $f(x) = w \cdot \phi(x)$.

Can we learn **intermediate features**?

Feature learning

Linear predictor with **handcrafted features**: $f(x) = w \cdot \phi(x)$.

Can we learn **intermediate features**?

Example:

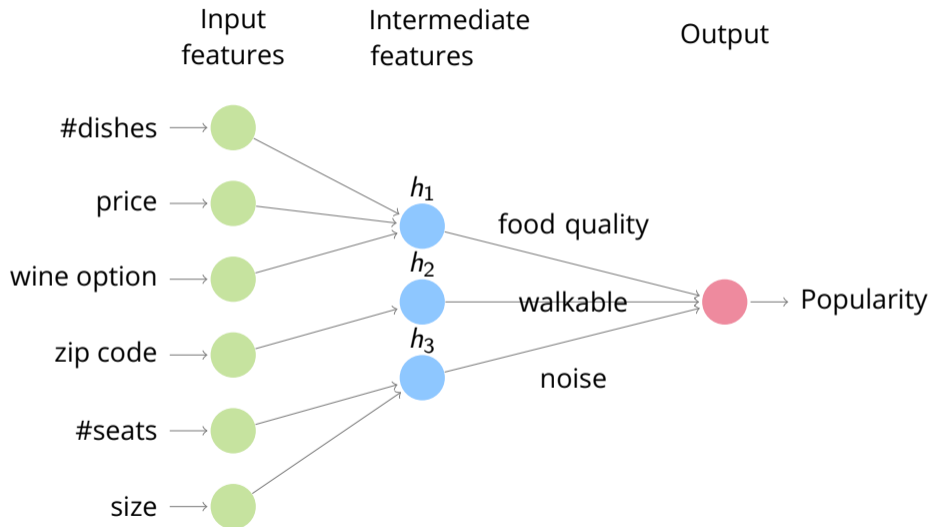
- Predict popularity of restaurants.
- Raw input: #dishes, price, wine option, zip code, #seats, size
- Decompose into subproblems:

$$h_1([\text{\#dishes, price, wine option}]) = \text{food quality}$$

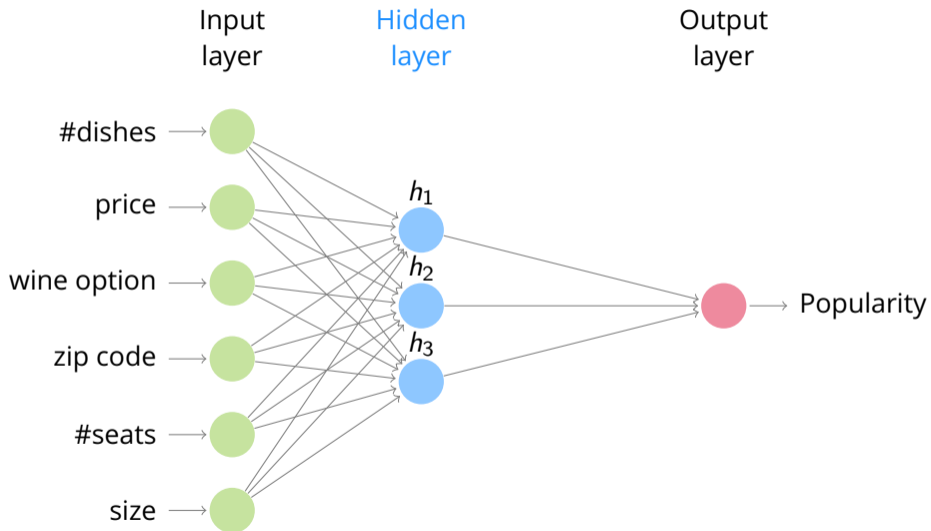
$$h_2([\text{zip code}]) = \text{walkable}$$

$$h_3([\text{\#seats, size}]) = \text{nosie}$$

Predefined subproblems



Learning intermediate features



Neural networks

Key idea: automatically learn the intermediate features.

Feature engineering: Manually specify $\phi(x)$ based on domain knowledge and learn the weights:

$$f(x) = w^T \phi(x).$$

Feature learning: Automatically learn both the features (K hidden units) and the weights:

$$h(x) = [h_1(x), \dots, h_K(x)], \quad f(x) = w^T h(x)$$

Activation function

- How should we parametrize h_i 's? Can it be linear?

Activation function

- How should we parametrize h_i 's? Can it be linear?

$$h_i(x) = \sigma(v_i^T x). \quad (1)$$

- σ is the *nonlinear* **activation function**.

Activation function

- How should we parametrize h_i 's? Can it be linear?

$$h_i(x) = \sigma(v_i^T x). \quad (1)$$

- σ is the *nonlinear* **activation function**.
- What might be some activation functions we want to use?

Activation function

- How should we parametrize h_i 's? Can it be linear?

$$h_i(x) = \sigma(v_i^T x). \quad (1)$$

- σ is the *nonlinear* **activation function**.
- What might be some activation functions we want to use?
 - sign function? **Non-differentiable**.

Activation function

- How should we parametrize h_i 's? Can it be linear?

$$h_i(x) = \sigma(v_i^T x). \quad (1)$$

- σ is the *nonlinear* **activation function**.
- What might be some activation functions we want to use?
 - sign function? **Non-differentiable**.
 - *Differentiable* approximations: sigmoid functions.
 - E.g., logistic function, hyperbolic tangent function, ReLU

Activation function

- How should we parametrize h_i 's? Can it be linear?

$$h_i(x) = \sigma(v_i^T x). \quad (1)$$

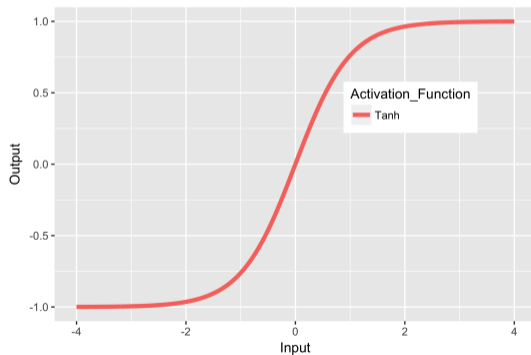
- σ is the *nonlinear* **activation function**.
- What might be some activation functions we want to use?
 - sign function? **Non-differentiable**.
 - *Differentiable* approximations: sigmoid functions.
 - E.g., logistic function, hyperbolic tangent function, ReLU
- Two-layer neural network (one **hidden layer** and one **output layer**) with K hidden units:

$$f(x) = \sum_{k=1}^K w_k h_k(x) = \sum_{k=1}^K w_k \sigma(v_k^T x) \quad (2)$$

Activation Functions

- The **hyperbolic tangent** is a common activation function:

$$\sigma(x) = \tanh(x).$$

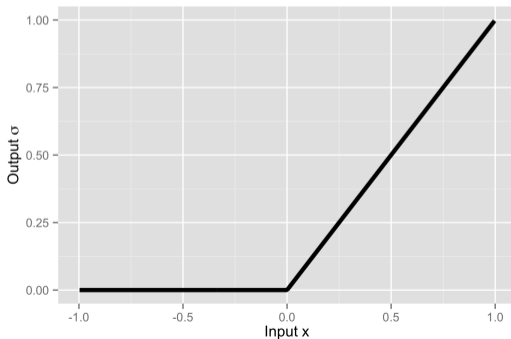


Activation Functions

- More recently, the **rectified linear (ReLU)** function has been very popular:

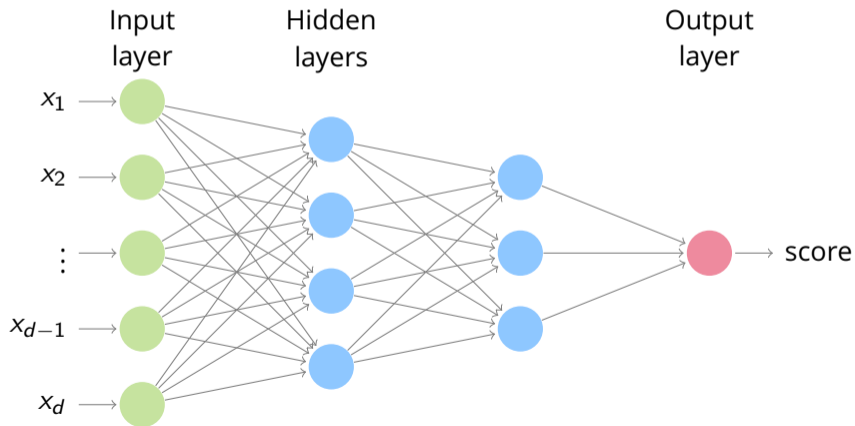
$$\sigma(x) = \max(0, x).$$

- Much **faster** to calculate, and to calculate its derivatives.
- Work well empirically.



Multilayer perceptron / Feed-forward neural networks

- Wider: more hidden units.
- Deeper: more hidden layers.



Multilayer Perceptron: Standard Recipe

- Each hidden layer takes the output $o \in \mathbb{R}^m$ of previous layer and produces

$$h^{(j)}(o^{(j-1)}) = \sigma \left(W^{(j)} o^{(j-1)} + b^{(j)} \right), \text{ for } j = 2, \dots, L$$

where $W^{(j)} \in \mathbb{R}^{m \times m}$, $b^{(j)} \in \mathbb{R}^m$.

Multilayer Perceptron: Standard Recipe

- Each hidden layer takes the output $o \in \mathbb{R}^m$ of previous layer and produces

$$h^{(j)}(o^{(j-1)}) = \sigma \left(W^{(j)} o^{(j-1)} + b^{(j)} \right), \text{ for } j = 2, \dots, L$$

where $W^{(j)} \in \mathbb{R}^{m \times m}$, $b^{(j)} \in \mathbb{R}^m$.

- The output layer is an *affine* mapping (no activation function):

$$a(o^{(L)}) = W^{(L+1)} o^{(L)} + b^{(L+1)},$$

where $W^{(L+1)} \in \mathbb{R}^{k \times m}$ and $b^{(L+1)} \in \mathbb{R}^k$.

Multilayer Perceptron: Standard Recipe

- Each hidden layer takes the **output** $o \in \mathbb{R}^m$ of previous layer and produces

$$h^{(j)}(o^{(j-1)}) = \sigma \left(W^{(j)} o^{(j-1)} + b^{(j)} \right), \text{ for } j = 2, \dots, L$$

where $W^{(j)} \in \mathbb{R}^{m \times m}$, $b^{(j)} \in \mathbb{R}^m$.

- The output layer is an *affine* mapping (no activation function):

$$a(o^{(L)}) = W^{(L+1)} o^{(L)} + b^{(L+1)},$$

where $W^{(L+1)} \in \mathbb{R}^{k \times m}$ and $b^{(L+1)} \in \mathbb{R}^k$.

- The full neural network function is given by the *composition* of layers:

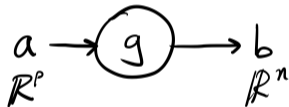
$$f(x) = \left(a \circ h^{(L)} \circ \dots \circ h^{(1)} \right) (x) \tag{3}$$

Computation graphs

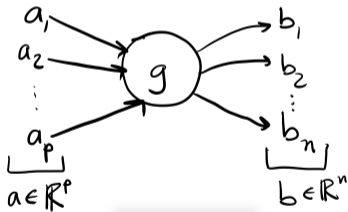
(adapted from David Rosenberg's slides)

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

- Typical computation graph:



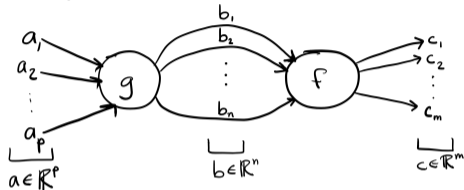
- Broken out into components:



Compose multiple functions

(adpated from David Rosenberg's slides)

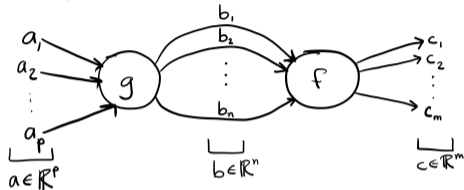
Compose two functions $g : \mathbb{R}^p \rightarrow \mathbb{R}^n$ and $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$: $c = f(g(a))$



Compose multiple functions

(adpated from David Rosenberg's slides)

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

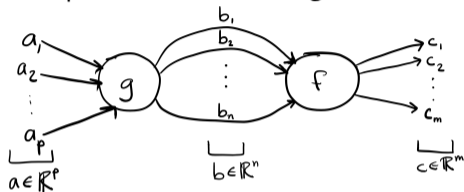


- Derivative: How does change in a_j affect c_i ?

Compose multiple functions

(adapted from David Rosenberg's slides)

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



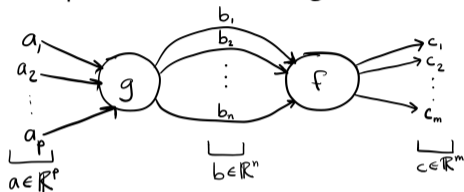
- Derivative: How does change in a_j affect c_i ?

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

Compose multiple functions

(adpated from David Rosenberg's slides)

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



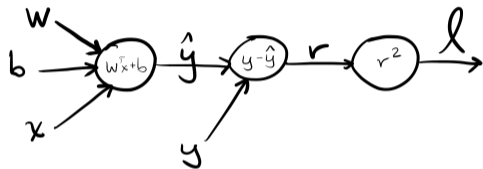
- Derivative: How does change in a_j affect c_i ?

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

- 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 on each edge.

Computation graph example

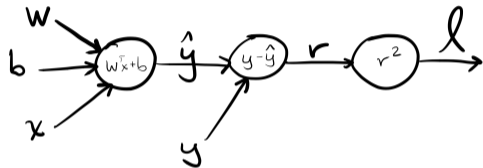
(adapted from David Rosenberg's slides)



(What is this graph computing?)

Computation graph example

(adapted from David Rosenberg's slides)

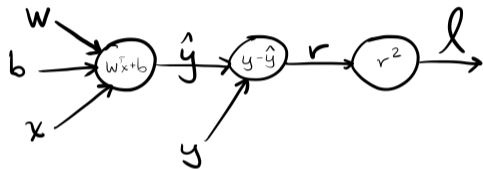


(What is this graph computing?)

$$\frac{\partial l}{\partial b} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r$$
$$\frac{\partial l}{\partial w_j} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_j} = (-2r)x_j = -2rx_j$$

Computation graph example

(adapted from David Rosenberg's slides)



(What is this graph computing?)

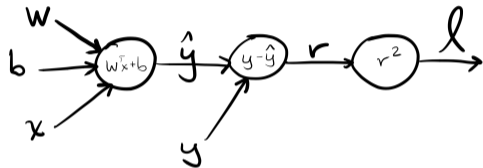
$$\frac{\partial l}{\partial b} = \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r$$

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

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

Computation graph example

(adapted from David Rosenberg's slides)



(What is this graph computing?)

$$\begin{aligned}\frac{\partial l}{\partial r} &= 2r \\ \frac{\partial l}{\partial \hat{y}} &= \frac{\partial l}{\partial r} \frac{\partial r}{\partial \hat{y}} = (2r)(-1) = -2r \\ \frac{\partial l}{\partial b} &= \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial b} = (-2r)(1) = -2r \\ \frac{\partial l}{\partial w_j} &= \frac{\partial l}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_j} = (-2r)x_j = -2rx_j\end{aligned}$$

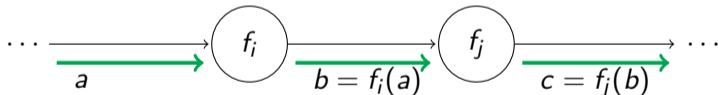
Computing the derivatives in certain order allows us to save compute!

Backpropogation

Backpropogation = chain rule + dynamic programming on a computation graph

Forward pass

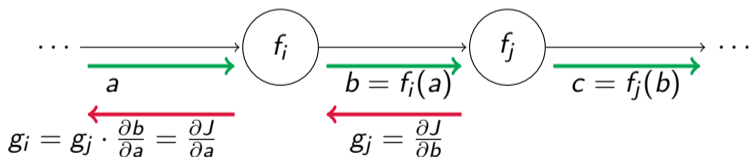
- **Topological order:** every node appears before its children
- For each node, compute the output given the input (from its parents).



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-propagation)
- Objective is non-convex, may not reach a global minimum

Table of Contents

Neural networks basics

Recurrent neural networks

Self-attention

Tranformer

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

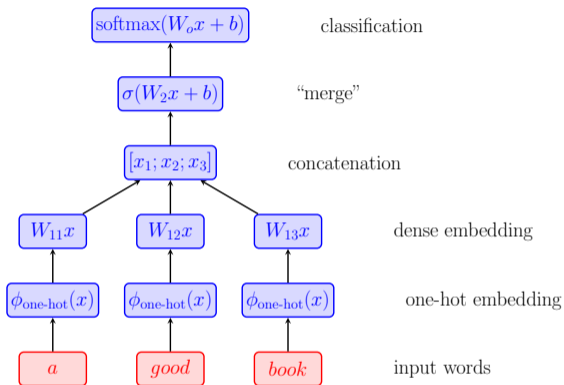
Problem setup: given an input sequence, come up with a (neural network) model that outputs a representation of the sequence for downstream tasks (e.g., classification)

Key challenge: how to model interaction among words?

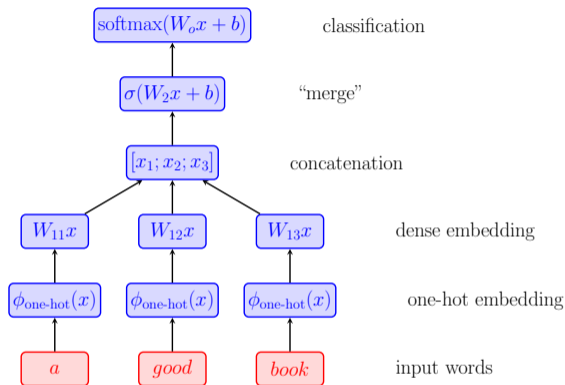
Approach:

- Aggregation / pooling word embeddings
- Recurrence
- Self-attention

Feed-forward neural network for text classification



Feed-forward neural network for text classification



What kind of features can be learned?

How to adapt the network to handle sequences with arbitrary length?

Recurrent neural networks

- **Goal:** represent a sequence of symbols of **varying lengths**
- **Idea:** combine new symbols with previous symbols recurrently by modeling the **temporal dynamics**

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

Recurrent neural networks

- **Goal:** represent a sequence of symbols of **varying lengths**
- **Idea:** combine new symbols with previous symbols recurrently by modeling the **temporal dynamics**

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

- Compute the **hidden states** h_t recurrently
 - Output from previous time step is the input to the current time step
 - Apply the same transformation f at each time step

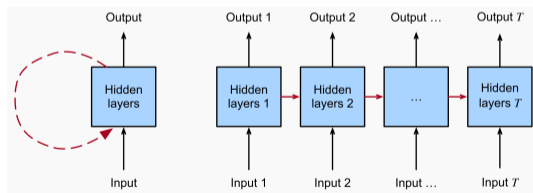
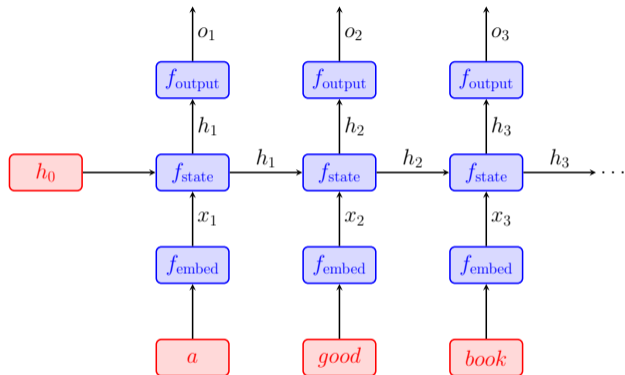


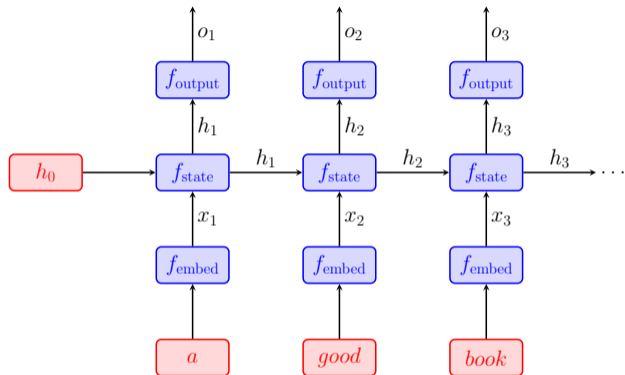
Figure: 9.1 from d2l.ai

Forward pass



A deep neural network with shared weights in each layer

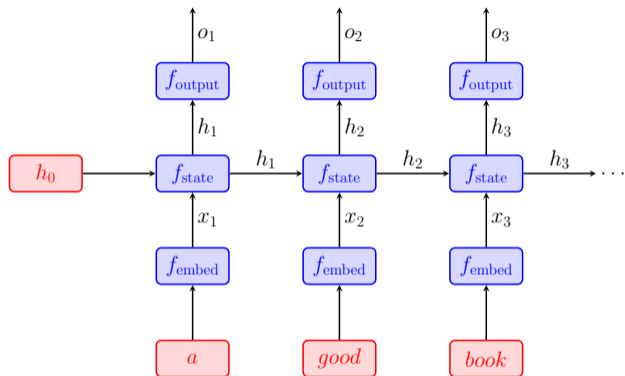
Forward pass



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

A deep neural network with shared weights in each layer

Forward pass

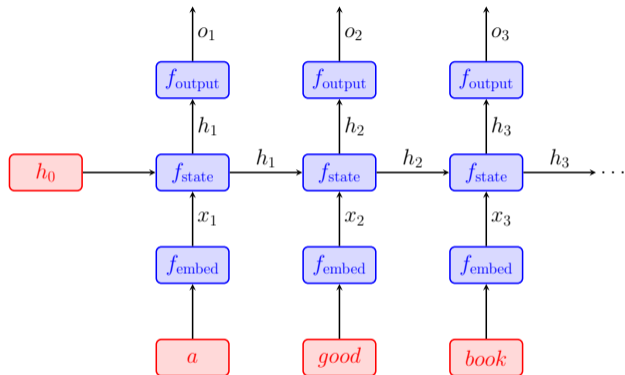


$$\begin{aligned}x_t &= f_{\text{embed}}(s_t) \\ &= W_e \phi_{\text{one-hot}}(s_t)\end{aligned}$$

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

A deep neural network with shared weights in each layer

Forward pass



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

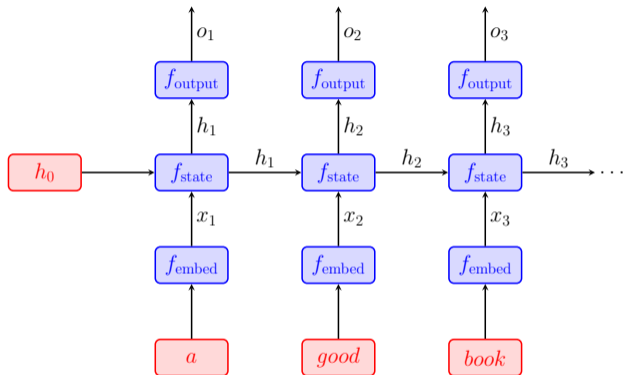
$$\begin{aligned}h_t &= f_{\text{state}}(x_t, h_{t-1}) \\ &= \sigma(W_{hh}h_{t-1} + W_{ih}x_t + b_h)\end{aligned}$$

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

A deep neural network with shared weights in each layer

Forward pass

Use o_t 's as features



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

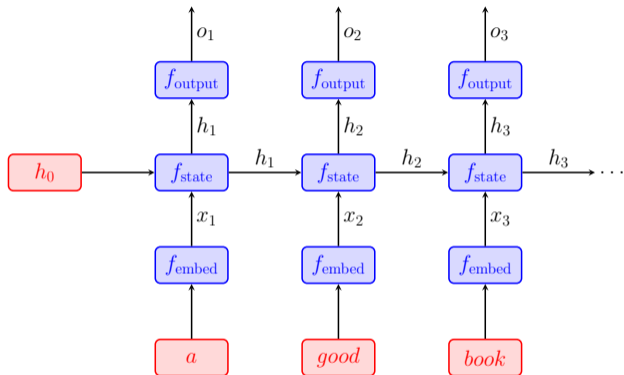
$$\begin{aligned}h_t &= f_{\text{state}}(x_t, h_{t-1}) \\ &= \sigma(W_{hh}h_{t-1} + W_{ih}x_t + b_h)\end{aligned}$$

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

A deep neural network with shared weights in each layer

Forward pass

Use o_t 's as features



A deep neural network with shared weights in each layer

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

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

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

Which computation can be parallelized?

Backward pass

Given the loss ℓ , compute the gradient with respect to W_{hh} .

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

Backward pass

Given the loss ℓ , compute the gradient with respect to W_{hh} .

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

Backward pass

Given the loss ℓ , compute the gradient with respect to W_{hh} .

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

Computation graph of h_t :

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 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

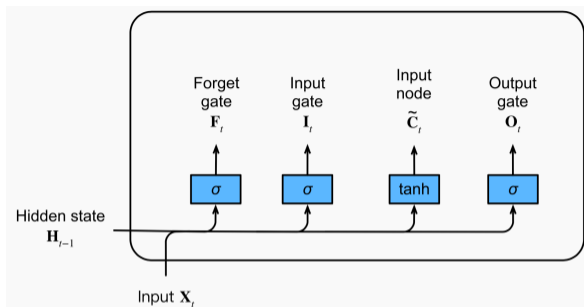


Figure: 10.1.2 from d2l.ai

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

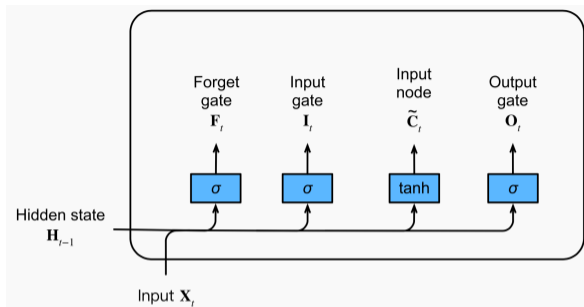


Figure: 10.1.2 from d2l.ai

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

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

Can we choose between $\tilde{\mathbf{c}}_t$ and another state that doesn't update with x_t ?

Long-short term memory (LSTM) parametrization

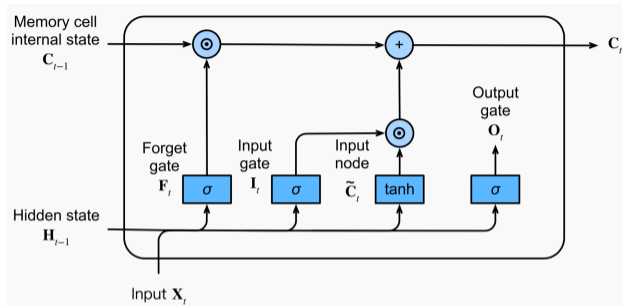


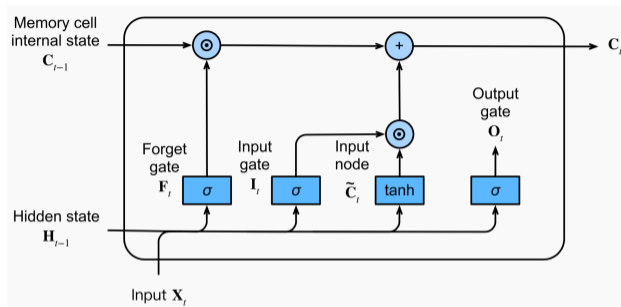
Figure: 10.1.3 from d2l.ai

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

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

- f_t : proportion of the old state (preserve or erase the old memory)
- i_t : proportion of the new state (write or ignore 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

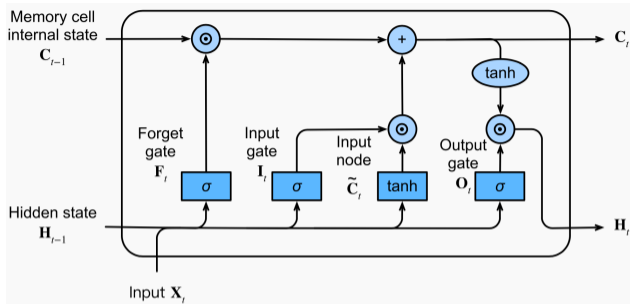


Figure: 10.1.4 from d2l.ai

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

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

$$h_t = o_t \odot c_t$$

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

Neural networks basics

Recurrent neural networks

Self-attention

Tranformer

Improve the efficiency of RNN

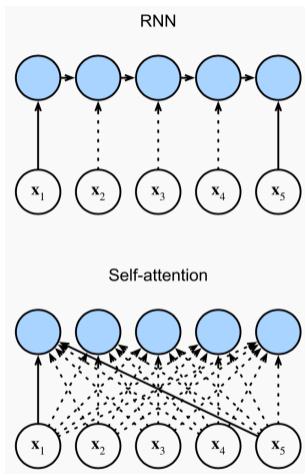


Figure: 11.6.1 from d2l.ai

Recall that our goal is to come up with a good representation 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

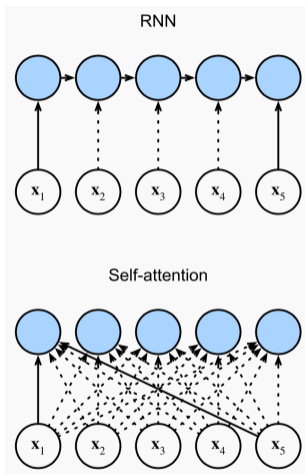


Figure: 11.6.1 from d2l.ai

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

RNN:

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

Can we handle dependency more **efficiently**?

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

Model interaction between words

Which word(s) is most related to “time”?

Model interaction between words

Which word(s) is most related to “time”?

A database approach:

query	keys	values
	arrow	time
	flies	flies
	like	like
	an	an
time	time	arrow

Model interaction between words

Which word(s) is most related to “time”?

A database approach:

query	keys	values
	arrow	time
	flies	flies
	like	like
	an	an
time	time	arrow

Output: arrow

Model interaction between words

Which word(s) is most related to “time”?

A database approach:

query	keys	values
	arrow	time
	flies	flies
	like	like
	an	an
time	time	arrow

Output: arrow

Limitations:

- Relatedness should not be hard-coded

Model interaction between words

Which word(s) is most related to “time”?

A database approach:

query	keys	values
	arrow	time
	flies	flies
	like	like
	an	an
time	time	arrow

Output: arrow

Limitations:

- Relatedness should not be hard-coded
- Need a function to measure relatedness between keys and values

Model interaction between words

Which word(s) is most related to “time”?

A database approach:

query	keys	values
	arrow	time
	flies	flies
	like	like
	an	an
time	time	arrow

Output: arrow

Limitations:

- Relatedness should not be hard-coded
Need a function to measure relatedness between keys and values
- A word is related to multiple words in a sentence

Model interaction between words

Which word(s) is most related to “time”?

A database approach:

query	keys	values
	arrow	time
	flies	flies
	like	like
	an	an
time	time	arrow

Output: arrow

Limitations:

- Relatedness should not be hard-coded
Need a function to measure relatedness between keys and values
- A word is related to multiple words in a sentence
Output should be an aggregation of the values

Model interaction between words using a "soft" database

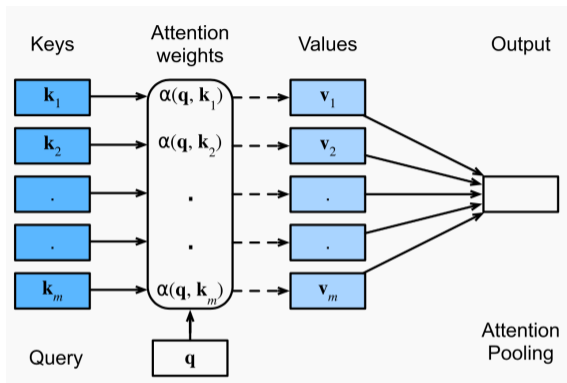


Figure: 11.1.1 from d2l.ai

- **Attention weights** $\alpha(q, k_i)$: how likely 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

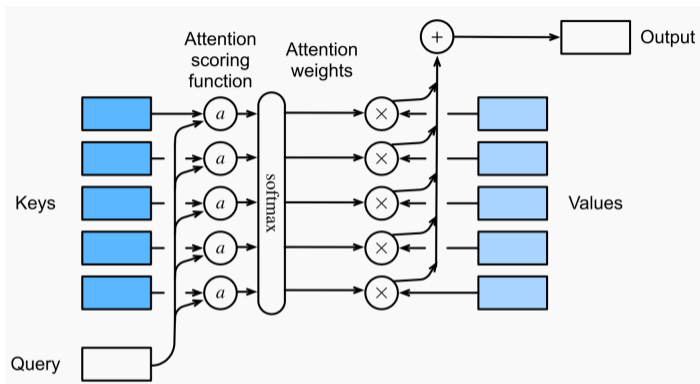


Figure: 11.3.1 from d2l.ai

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

Self-attention

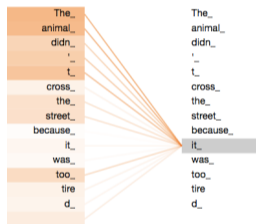
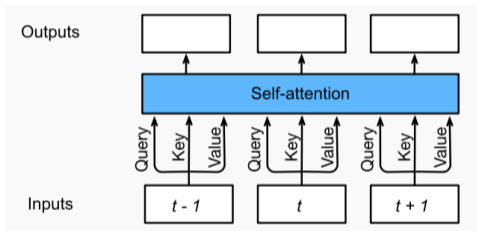
Goal: an efficient model of the interaction among symbols in a sequence

Idea: model the interaction between each pair of words (in parallel)

Self-attention

Goal: an efficient model of the interaction among symbols in a sequence

Idea: model the interaction between each pair of words (in parallel)

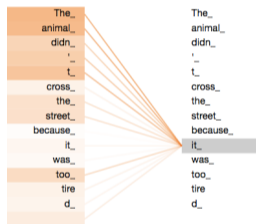
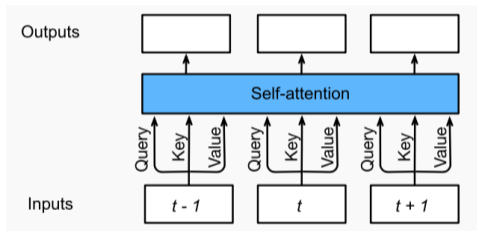


- **Input:** map each symbol to a query, a key, and a value (embeddings)

Self-attention

Goal: an efficient model of the interaction among symbols in a sequence

Idea: model the interaction between each pair of words (in parallel)

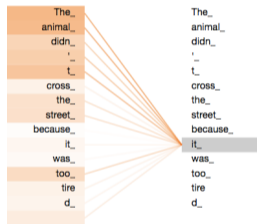
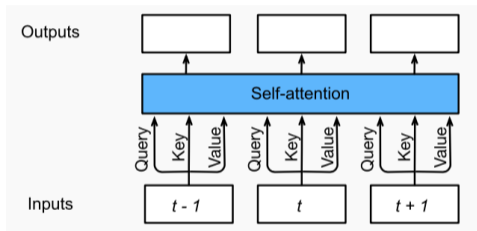


- **Input:** map each symbol to a query, a key, and a value (embeddings)
- **Attend:** each word (as a query) interacts with all words (keys)

Self-attention

Goal: an efficient model of the interaction among symbols in a sequence

Idea: model the interaction between each pair of words (in parallel)



- **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:

$$\alpha = \text{softmax}(a(q, k))$$

Attention scoring functions

Design the function that measures relatedness between queries and keys:

$$\alpha = \text{softmax}(a(q, k))$$

Dot-product attention

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

Attention scoring functions

Design the function that measures relatedness between queries and keys:

$$\alpha = \text{softmax}(a(q, k))$$

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:

$$\alpha = \text{softmax}(a(q, k))$$

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

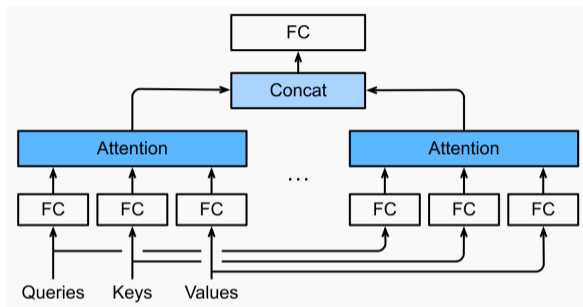
- Each word attends to all other words in the sentence
- Which words should “like” attend to?
 - Syntax: “flies”, “arrow” (a preposition)
 - Semantics: “time”, “arrow” (a metaphor)

Multi-head attention: motivation

Time flies like an arrow

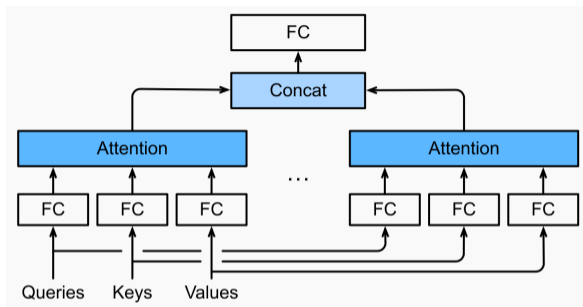
- Each word attends to all other words in the sentence
- Which words should “like” attend to?
 - Syntax: “flies”, “arrow” (a preposition)
 - Semantics: “time”, “arrow” (a metaphor)
- We want to represent different roles of a word in the sentence: need more than a single embedding
- Instantiation: multiple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

Multi-head attention



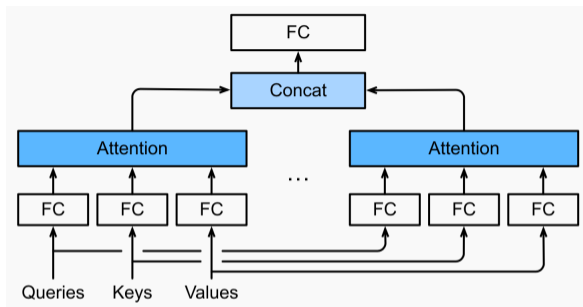
- Multiple attention modules: same architecture, different parameters

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

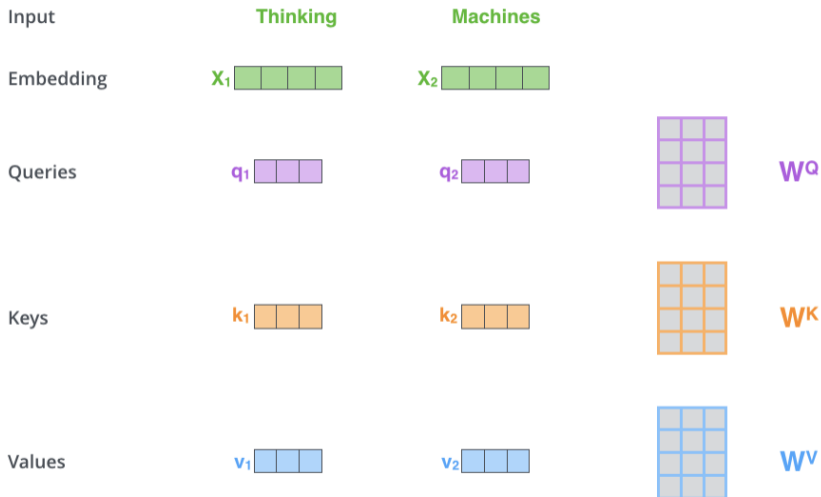
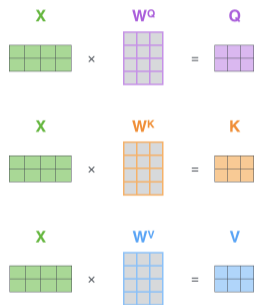


Figure: From [The Illustrated Transformer](#)

Matrix representation: attention weights

Scaled dot product attention



$$\text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) V$$

$= Z$

The diagram illustrates the calculation of the attention weights Z matrix from the Q and K^T matrices, and its multiplication with the V matrix.

- Q (purple 2x4 grid) multiplied by K^T (orange 4x2 grid) is divided by $\sqrt{d_k}$.
- The result is passed through a softmax function.
- The result is multiplied by V (blue 2x4 grid) to produce Z (pink 2x4 grid).

Figure: From [The Illustrated Transformer](#)

Multi-head attention

1) This is our input sentence*

Thinking
Machines



2) We embed each word*

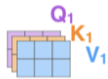


* In all encoders other than #0, we don't need embedding. We start directly with the output of the encoder right below this one

3) Split into 8 heads. We multiply X or R with weight matrices



4) Calculate attention using the resulting $Q/K/V$ matrices



5) Concatenate the resulting Z matrices, then multiply with weight matrix W^O to produce the output of the layer



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

Neural networks basics

Recurrent neural networks

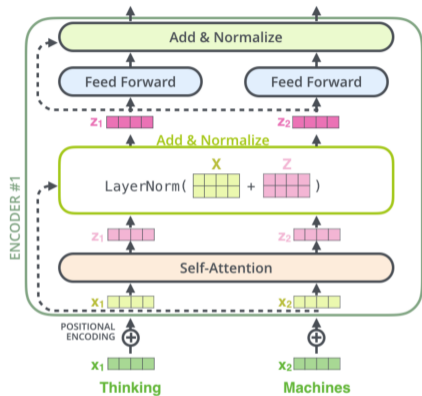
Self-attention

Tranformer

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)

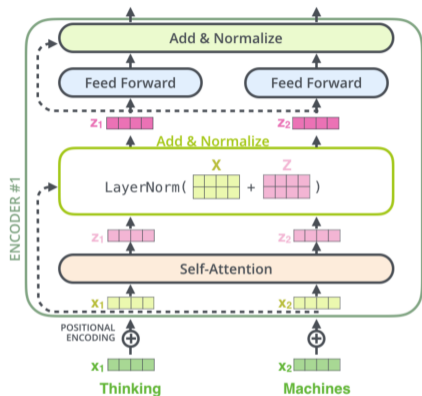
Transformer block



- Multi-head self-attention

Figure: From The Illustrated Transformer

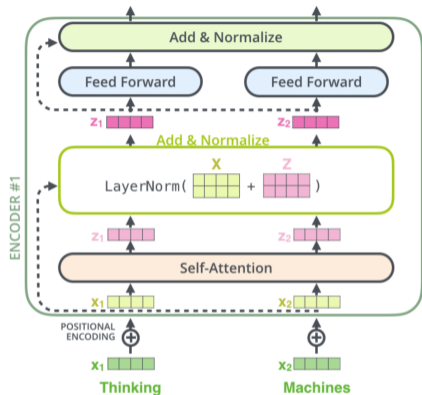
Transformer block



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

Figure: From **The Illustrated Transformer**

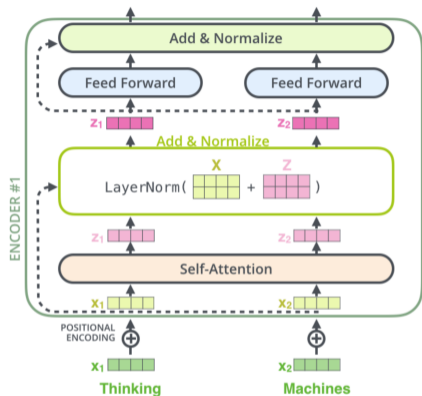
Transformer block



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

Figure: From [The Illustrated Transformer](#)

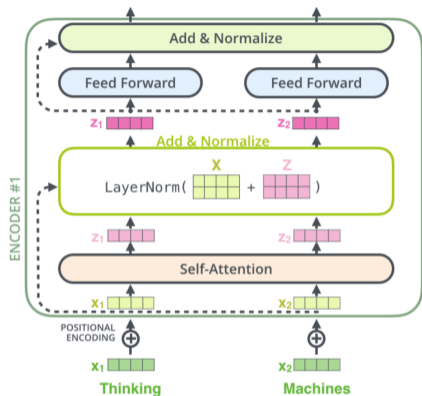
Transformer block



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

Figure: From **The Illustrated Transformer**

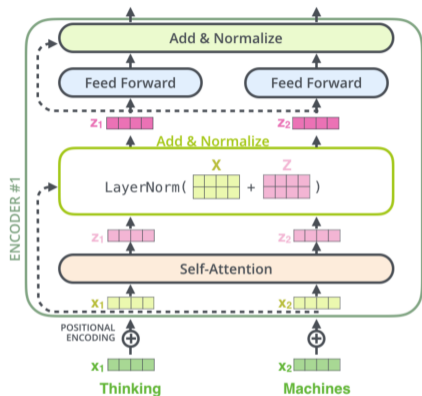
Transformer block



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

Figure: From [The Illustrated Transformer](#)

Transformer block



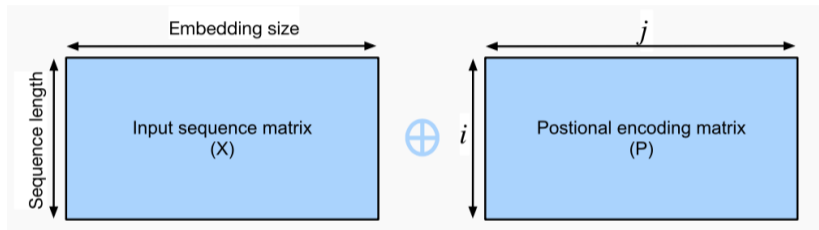
- Multi-head self-attention
 - Capture dependence among input symbols
- Positional encoding
 - Capture the order of symbols
- Residual connection and layer normalization
 - More efficient and better optimization

Figure: From [The Illustrated Transformer](#)

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

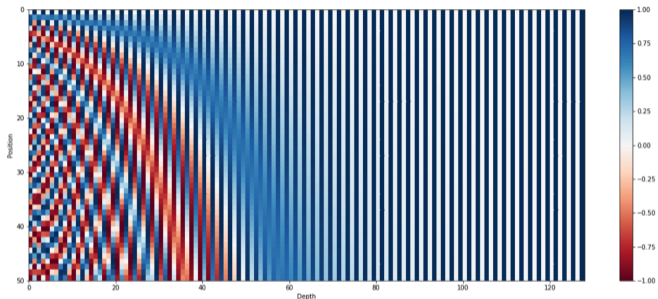
Intuition: continuous approximation of binary encoding of positions (integers)

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)

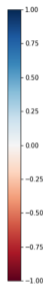
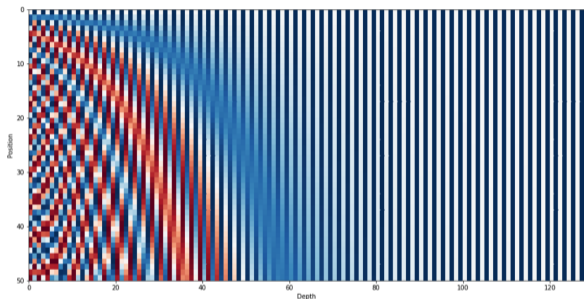
0 : 0 0 0 0
1 : 0 0 0 1
2 : 0 0 1 0
3 : 0 0 1 1
4 : 0 1 0 0
5 : 0 1 0 1
6 : 0 1 1 0
7 : 0 1 1 1



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



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

Figure: From Amirhossein Kazemnejad's Blog

$$\omega_{2i} = \omega_{2i+1} = 1/10000^{\frac{2i}{d}}$$

Learned position embeddings

Sinusoidal position embedding:

- Not learnable
- Extrapolating to longer sequences doesn't work

Learned position embeddings

Sinusoidal position embedding:

- Not learnable
- Extrapolating to longer sequences doesn't work

Learned absolute position embeddings (most used 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
- Extrapolating to longer sequences doesn't work

Learned absolute position embeddings (most used 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**).

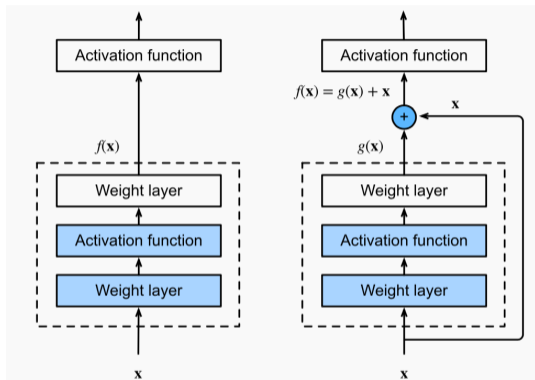
Residual connection

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.
- How can we make it easier to recover the shallow solution?

Residual connection

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



Learn the residual layer: $g(x) = f(x) - x$

If the shallow network is better, set $g(x) = 0$ (easier to learn).

Layer normalization

Layer Normalization [Ba et al., 2016]

- Normalize (zero mean, unit variance) across features
- Let $x = (x_1, \dots, x_d)$ be the input vector (e.g., word embedding, previous layer output)

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

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

Layer normalization

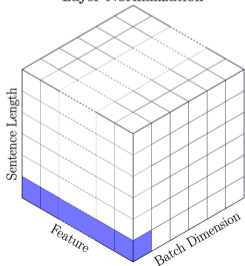
Layer Normalization [Ba et al., 2016]

- Normalize (zero mean, unit variance) across features
- Let $x = (x_1, \dots, x_d)$ be the input vector (e.g., word embedding, previous layer output)

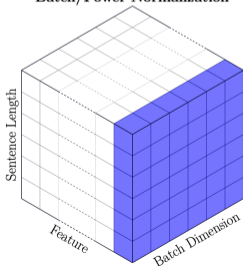
$$\text{LayerNorm}(x) = \frac{x - \hat{\mu}}{\hat{\sigma}},$$

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

Layer Normalization

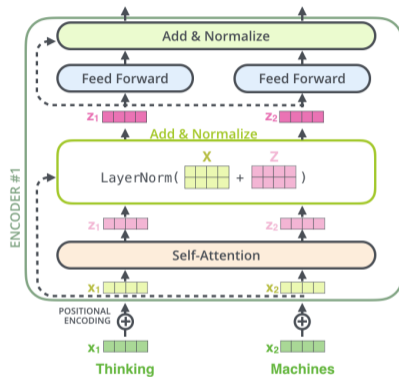


Batch/Power Normalization



- 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