Neural Network Basics

Shi Yu, Chaoqun He, Jing Yi

THUNLP

Content

- Neural Network Components
  - Simple Neuron; Multilayer; Feedforward; Non-linear; …
- How to Train
  - Objective; Gradients; Backpropogation
- Word Representation: Word2Vec
- Common Neural Networks
  - RNN
    - Sequential Memory; Language Model
    - Gradient Problem for RNN
    - Variants: GRU; LSTM; Bidirectional;
  - CNN
- NLP Pipeline Tutorial (PyTorch)

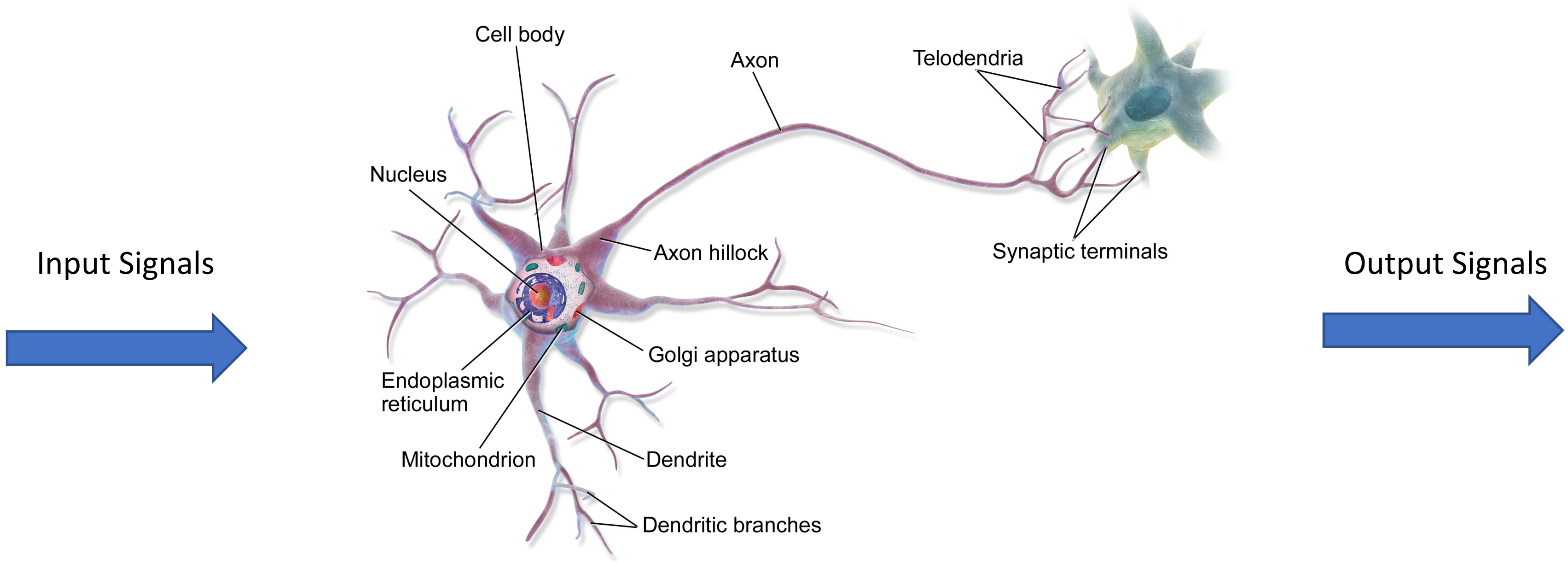
Neural Network Components

Shi Yu

THUNLP

Neural Network

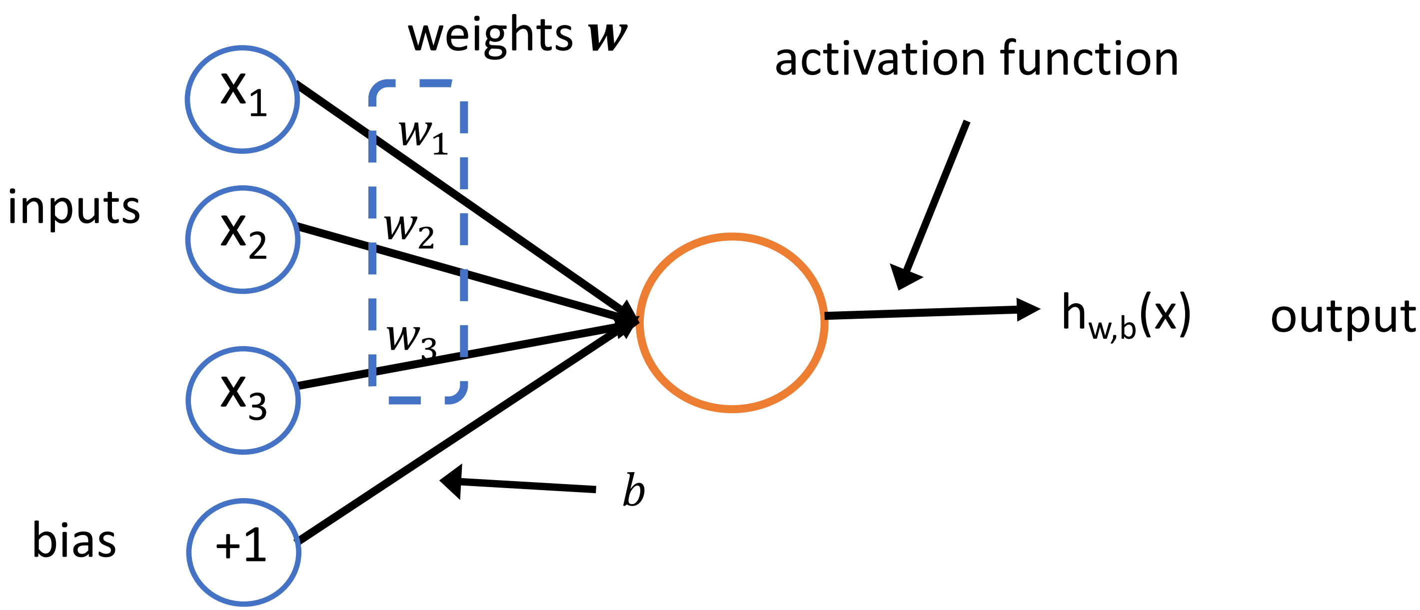
- (Artificial) Neural Network
- Inspired by the biological neural networks in brains



Source: Wikipedia

(Artificial) Neuron

- A neuron is a computational unit with n inputs and 1 output and parameters $\boldsymbol{w}, b$

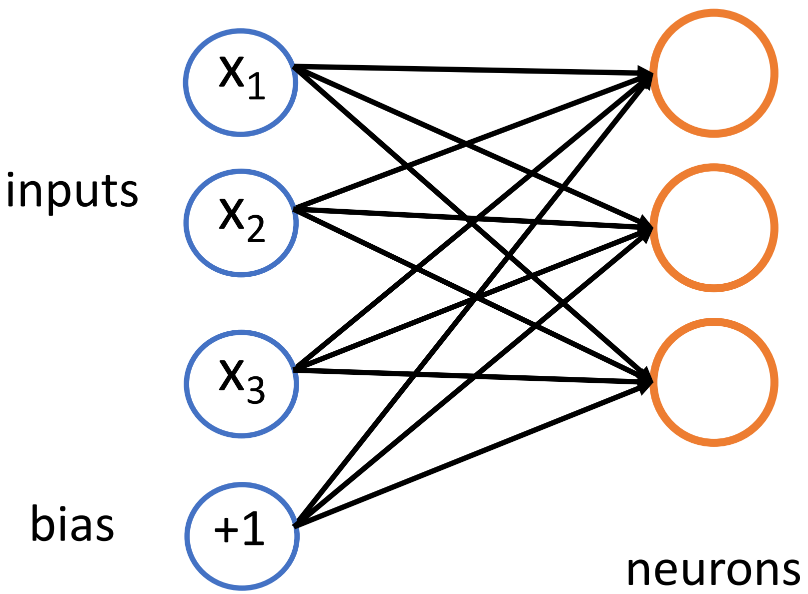


$$h_{\boldsymbol{w},b}(\boldsymbol{x}) = f(\boldsymbol{w}^T\boldsymbol{x} + b)$$

$\boldsymbol{w}$, b are the parameters of this neuron

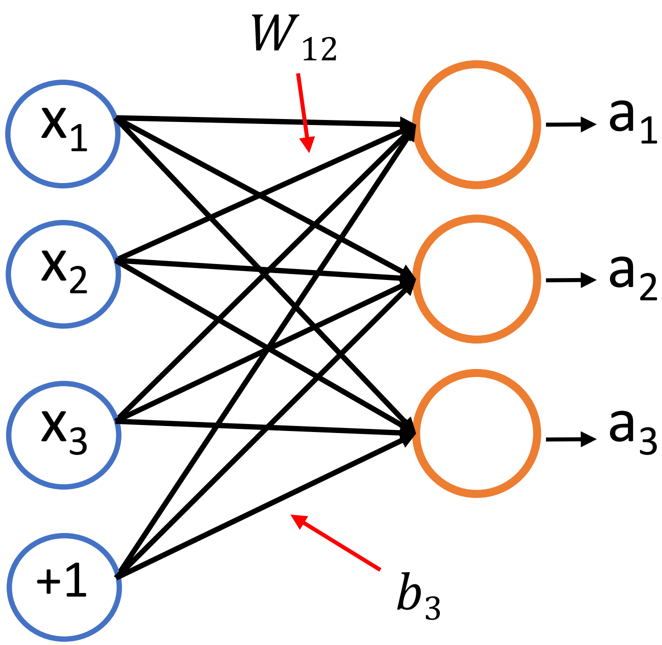
Single Layer Neural Network

- A single layer neural network: Hooking together many simple neurons



Matrix Notation

- A single layer neural network: Hooking together many simple neurons



$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3 + b_1)$$

$$a_2 = f(W_{21}x_1 + W_{22}x_2 + W_{23}x_3 + b_2)$$

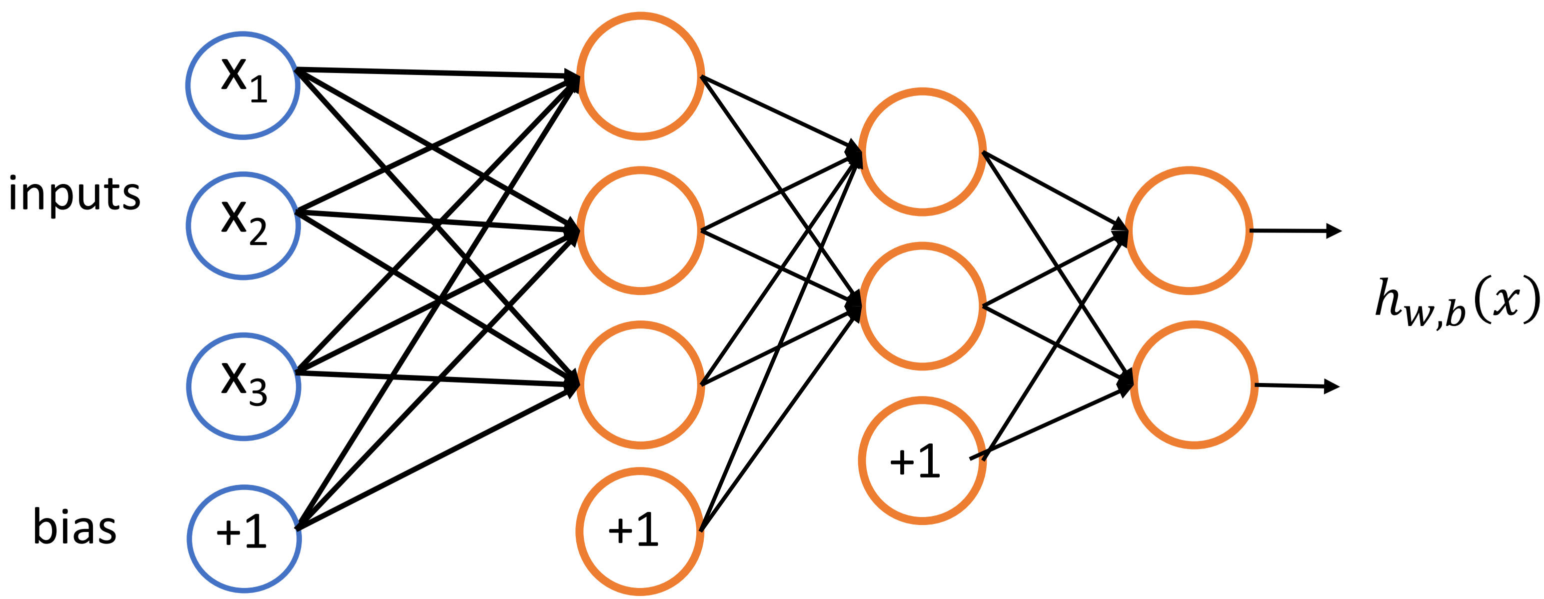
$$a_3 = f(W_{31}x_1 + W_{32}x_2 + W_{33}x_3 + b_3)$$

In matrix form:

$$\boldsymbol{a} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

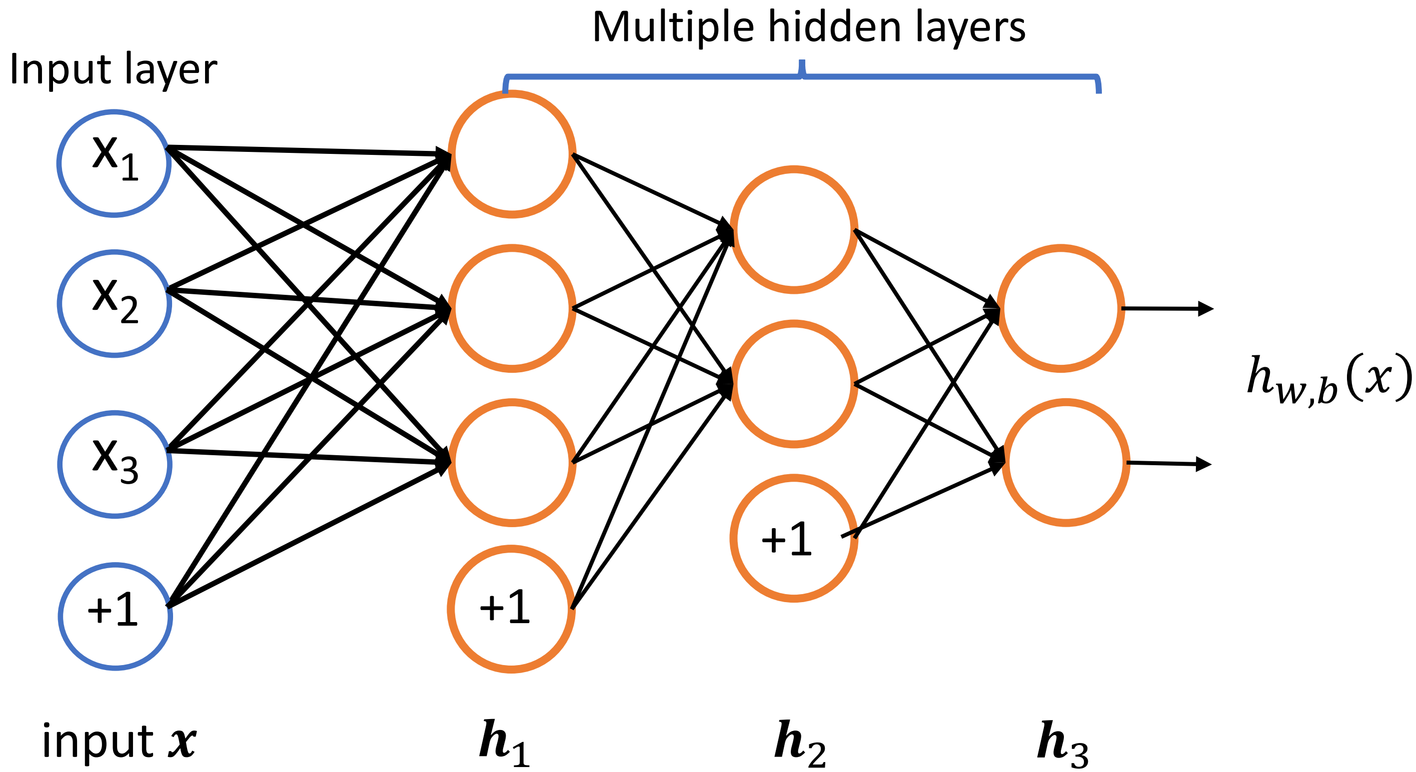
Multilayer Neural Network

- Stacking multiple layers of neural networks



Feedforward Computation

Feedforward Computation



$$\boldsymbol{h}_1 = f(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1)$$

$$\boldsymbol{h}_2 = f(\boldsymbol{W}_2\boldsymbol{h}_1 + \boldsymbol{b}_2)$$

$$\boldsymbol{h}_3 = f(\boldsymbol{W}_3\boldsymbol{h}_2 + \boldsymbol{b}_3)$$

Why use non-linearities (f)?

- Without non-linearities, deep neural networks cannot do anything more than a linear transform
  - Extra layers could just be compiled down into a single linear transform

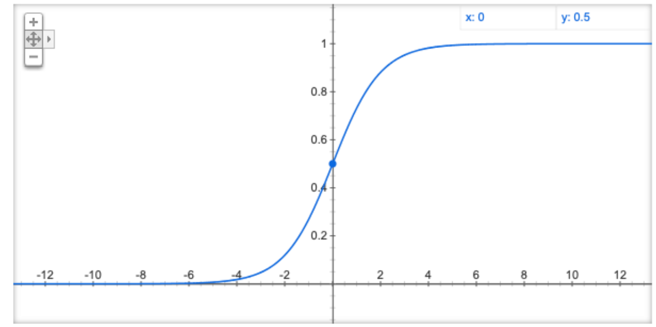
$$\boldsymbol{h}_1 = \boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1 \quad \boldsymbol{h}_2 = \boldsymbol{W}_2\boldsymbol{h}_1 + \boldsymbol{b}_2 \quad \Rightarrow \quad \boldsymbol{h}_2 = \boldsymbol{W}_2\boldsymbol{W}_1 x + \boldsymbol{W}_2\boldsymbol{b}_1 + \boldsymbol{b}_2$$

- With non-linearities, neural networks can approximate more complex functions with more layers!

Choices of non-linearities

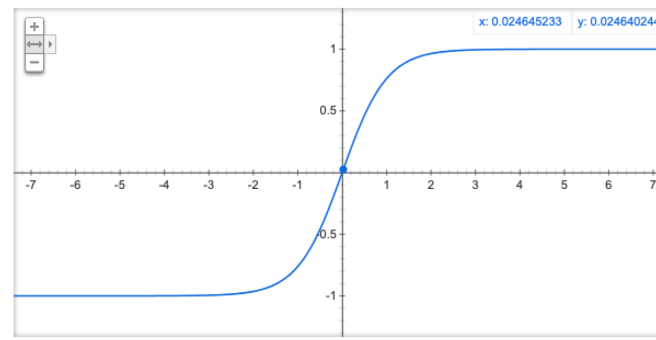
- Sigmoid

$$f(z) = \frac{1}{1 + e^{-z}}$$



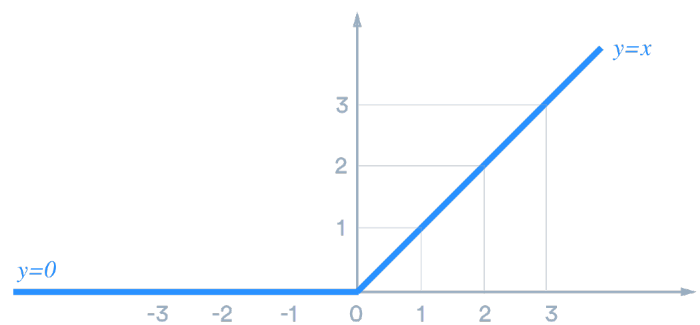
- Tanh

$$f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$



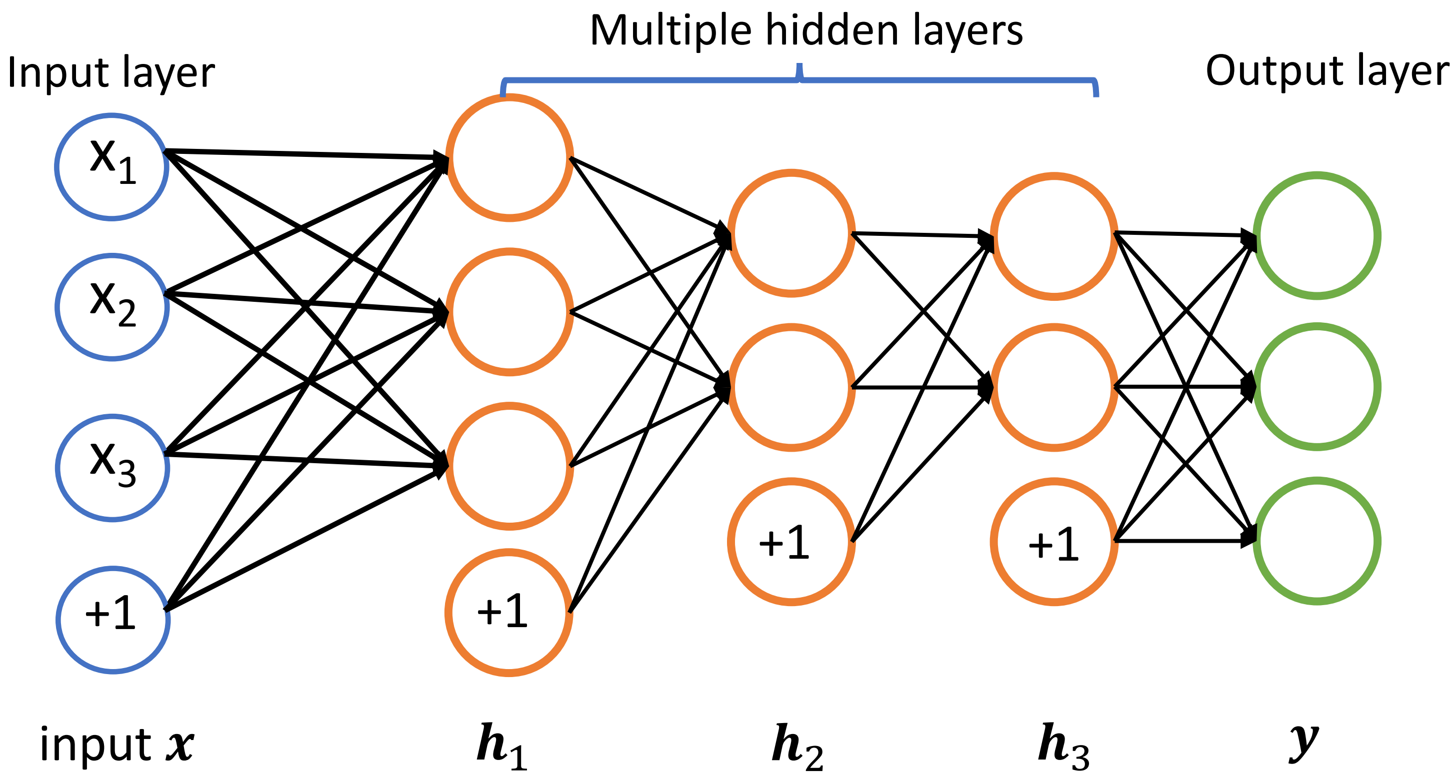
- ReLU

$$f(z) = \max(z, 0)$$



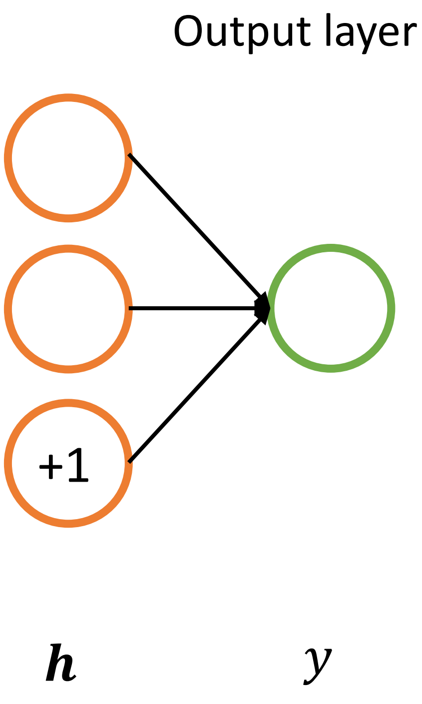
- …

Output Layer



Output Layer

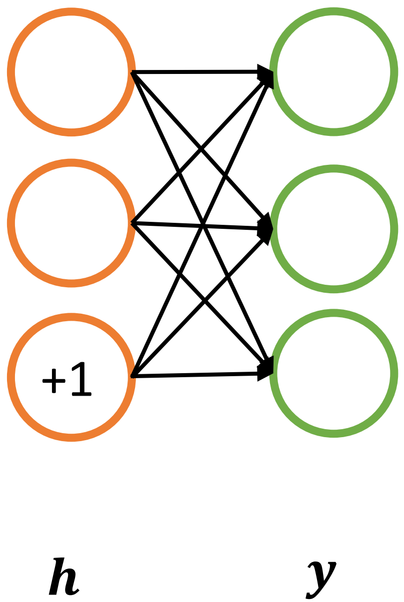
- Linear output
  - $y = \boldsymbol{w}^T \boldsymbol{h} + b$
- Sigmoid
  - $y = \sigma(\boldsymbol{w}^T \boldsymbol{h} + b)$
  - For binary classification
    - y for one class
    - $1 - y$ for another



Output Layer

- Softmax
  - $y_i = \text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$
  - $\boldsymbol{z} = \boldsymbol{W}\boldsymbol{h} + \boldsymbol{b}$
  - For multi-class classification

Output layer



Summary

- Simple neuron
- Single layer neural network
- Multilayer neural network
  - Stack multiple layers of neural networks
- Non-linearity activation function
  - Enable neural nets to represent more complicated features
- Output layer
  - For desired output

How to Train a Neural Network

Shi Yu

THUNLP

Training Objective

- Mean Squared Error
  - Given N training examples $\{(x_i, y_i)\}_{i=1}^{N}$ where x_i and y_i are the attributes and price of a computer. We want to train a neural network $F_\theta(\cdot)$ which takes the attributes x as input and predicts its price y. A reasonable training objective is Mean Squared Error:

$$\min_\theta J(\theta) = \min_\theta \frac{1}{N} \sum_{i=1}^{N} (y_i - F_\theta(x_i))^2 \,,$$

where θ is the parameters in neural network $F_\theta(\cdot)$.

Training Objective

- Cross-entropy
  - Given N training examples $\{(x_i, y_i)\}_{i=1}^{N}$ where x_i and y_i are the sentence and its sentiment label. We want to train a neural network $F_\theta(\cdot)$ which takes the sentence x as input and predicts its sentiment y. A reasonable training objective is Cross-entropy:

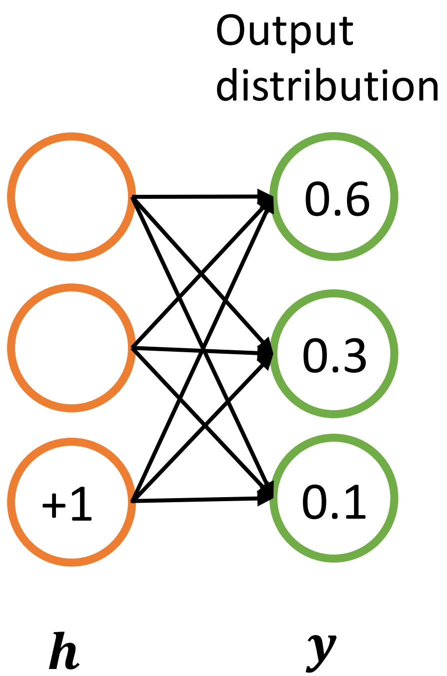
$$\min_\theta J(\theta) = \min_\theta -\frac{1}{N}\sum_{i=1}^{N} \log P_{\text{model}}(F_\theta(x_i) = y_i),$$

where θ is the parameters in neural network $F_\theta(\cdot)$.

Training Objective

- Cross-entropy

$$\min_{\theta} J(\theta) = \min_{\theta} -\frac{1}{N}\sum_{i=1}^{N} \log P_{\text{model}}(F_{\theta}(x_i) = y_i)\,,$$



If ground truth is y=1 (first class), then the loss for this instance is

$-\log P_{\text{model}}(F_{\theta}(x) = 1) = -\log(0.6) = 0.74.$

If y=2 …

$-\log P_{\text{model}}(F_{\theta}(x) = 2) = -\log(0.3) = 1.74.$

If y=3 …

$-\log P_{\text{model}}(F_{\theta}(x) = 3) = -\log(0.1) = 3.32.$

Stochastic Gradient Descent

- Update rule:

$$\theta^{new} = \theta^{old} - \alpha \nabla_{\theta} \mathrm{J}(\theta)$$

α is step size or learning rate

- Just like climbing a mountain
  - find the steepest direction
  - take a step

Gradients

- Given a function with 1 output and n inputs:

$$\mathrm{F}(\boldsymbol{x}) = F(x_1, x_2 \dots x_n)$$

- Its gradient is a vector of partial derivatives:

$$\frac{\partial \mathrm{F}}{\partial \boldsymbol{x}} = [\frac{\partial \mathrm{F}}{\partial x_1}, \frac{\partial \mathrm{F}}{\partial x_2} \dots \frac{\partial \mathrm{F}}{\partial x_n}]$$

Jacobian Matrix: Generalization of the Gradient

- Given a function with m outputs and n inputs:

$$\mathrm{F}(\boldsymbol{x}) = [F_1(x_1, x_2 \ldots x_n), F_2(x_1, x_2 \ldots x_n) \ldots F_m(x_1, x_2 \ldots x_n)]$$

- Its Jacobian matrix is an $m \times n$ matrix of partial derivatives:

$$\frac{\partial \mathrm{F}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial \mathrm{F_1}}{\partial x_1} & \cdots & \frac{\partial \mathrm{F_1}}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial \mathrm{F_m}}{\partial x_1} & \cdots & \frac{\partial \mathrm{F_m}}{\partial x_n} \end{bmatrix} \qquad \left(\frac{\partial \mathrm{F}}{\partial \boldsymbol{x}}\right)_{ij} = \frac{\partial \mathrm{F_i}}{\partial x_j}$$

Chain Rule for Jacobians

- For one-variable functions: multiply derivatives

$$z = 3y$$

$$y = x^2$$

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} = 3 \times 2x = 6x$$

- For multiple variables: multiply Jacobians

$$\boldsymbol{h} = f(\boldsymbol{z})$$

$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$

$$\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{x}} = \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{x}} = \cdots$$

Back to Neural Network

- Given $s = \boldsymbol{u}^T\boldsymbol{h}, \boldsymbol{h} = f(\boldsymbol{z}), \boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, what is $\frac{\partial s}{\partial \boldsymbol{b}}$?

Back to Neural Network

- Given $s = \boldsymbol{u}^T\boldsymbol{h}, \boldsymbol{h} = f(\boldsymbol{z}), \boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$, what is $\frac{\partial s}{\partial \boldsymbol{b}}$?
  - Apply the chain rule:

$$\frac{\partial s}{\partial \boldsymbol{b}} = \underbrace{\boxed{\frac{\partial s}{\partial \boldsymbol{h}}}}_{\boldsymbol{u}^T}\underbrace{\boxed{\frac{\partial \boldsymbol{h}}{\partial \boldsymbol{z}}}}_{\mathrm{diag}(\mathrm{f}'(z))}\underbrace{\boxed{\frac{\partial \boldsymbol{z}}{\partial \boldsymbol{b}}}}_{\mathbf{I}}$$

Backpropagation

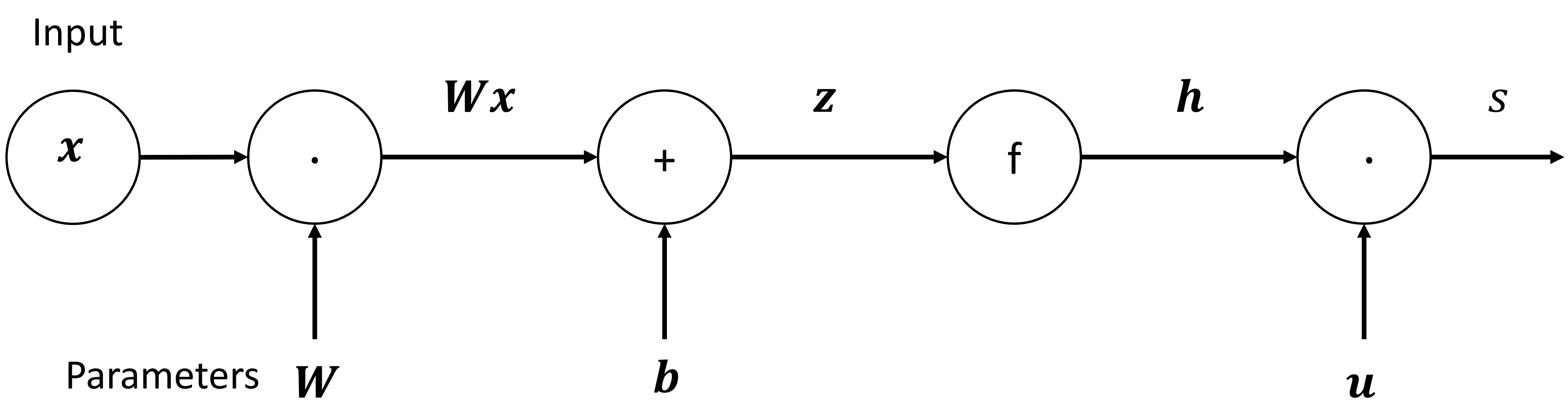
- Compute gradients algorithmically

- Used by deep learning frameworks (TensorFlow, PyTorch, etc.)

Computational Graphs

- Representing our neural net equations as a graph
  - Source node: inputs
  - Interior nodes: operations
  - Edges pass along result of the operation

$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$\boldsymbol{x}$ input

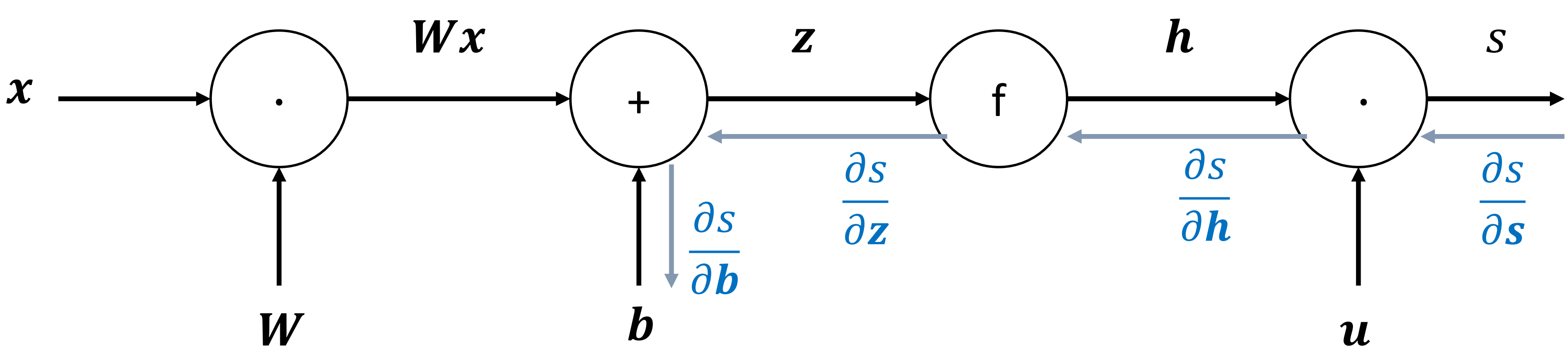


"Forward Propagation"

Backpropagation

- Go backwards along edges
  - Pass along gradients

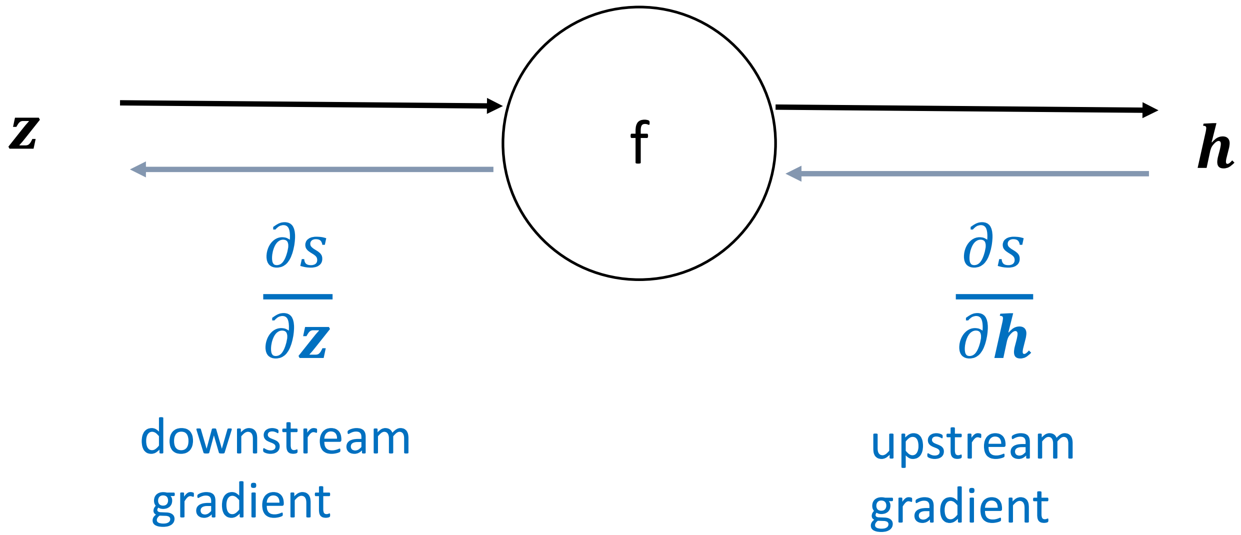
$$s = \boldsymbol{u}^T \boldsymbol{h}$$
$$\boldsymbol{h} = f(\boldsymbol{z})$$
$$\boldsymbol{z} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$$
$$\boldsymbol{x} \text{ input}$$



Backpropagation: Single Node

- Node receives an “upstream gradient”
- Goal is to pass on the correct “downstream gradient”

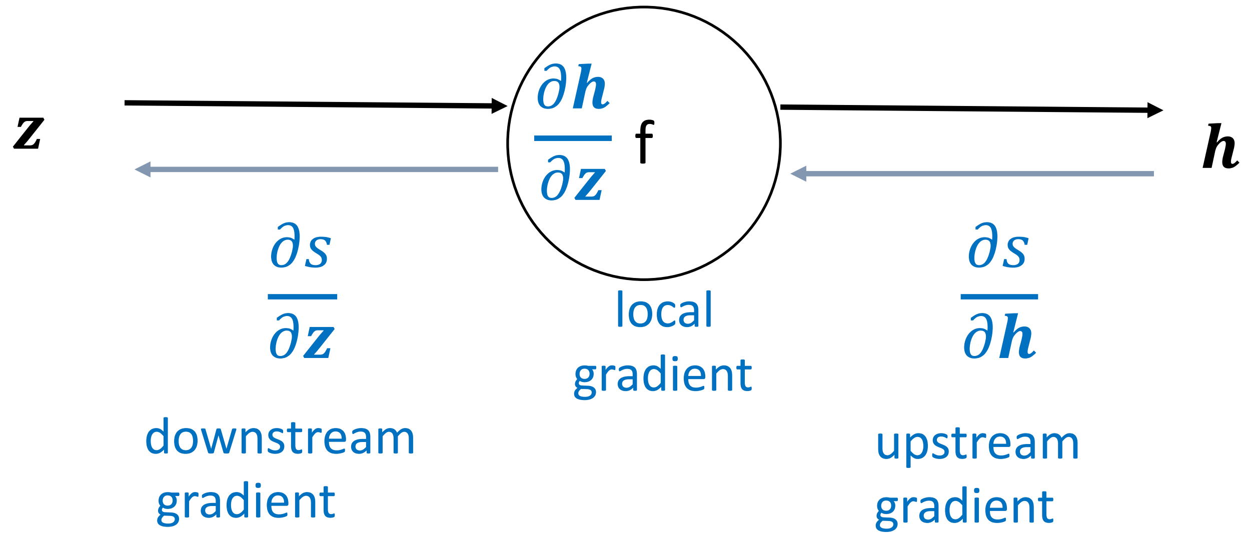
$$\boldsymbol{h} = f(\boldsymbol{z})$$



Backpropagation: Single Node

- Each node has a local gradient
  - The gradient of its output with respect to its input

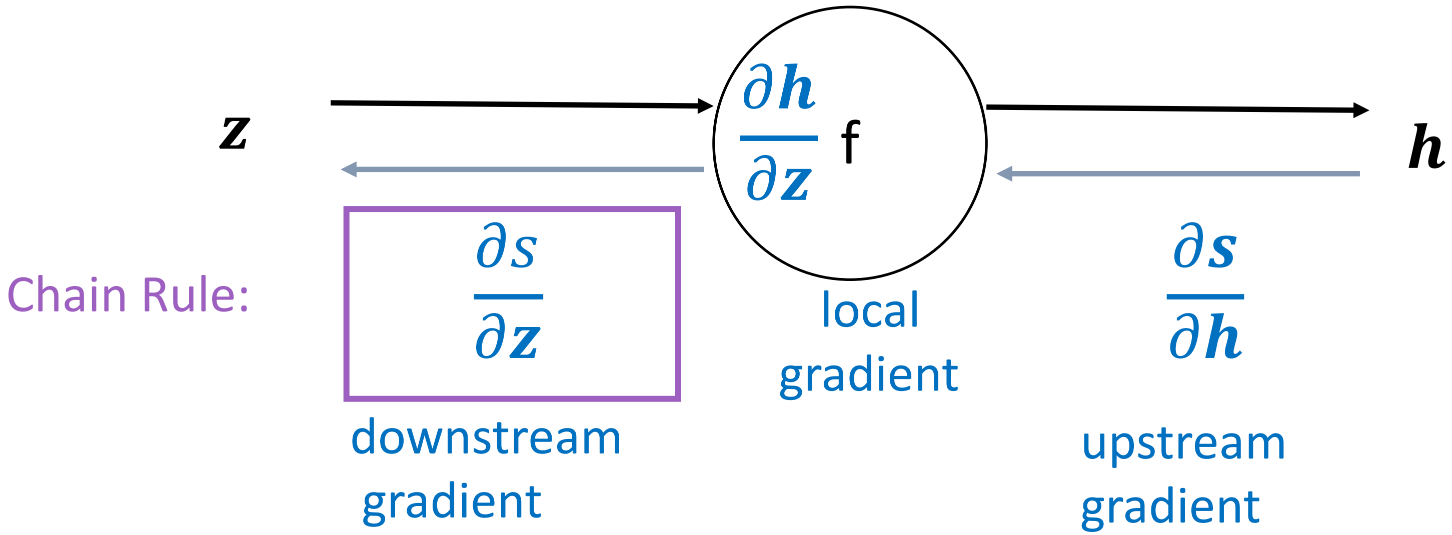
$$\boldsymbol{h} = f(\boldsymbol{z})$$



Backpropagation: Single Node

- Each node has a local gradient
  - The gradient of its output with respect to its input
- [downstream gradient] = [upstream gradient] x [local gradient]

$$\boldsymbol{h} = f(\boldsymbol{z})$$



An Example

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps:

$$a = x + y = 3$$

$$\mathrm{b} = \max(y, z) = 2$$

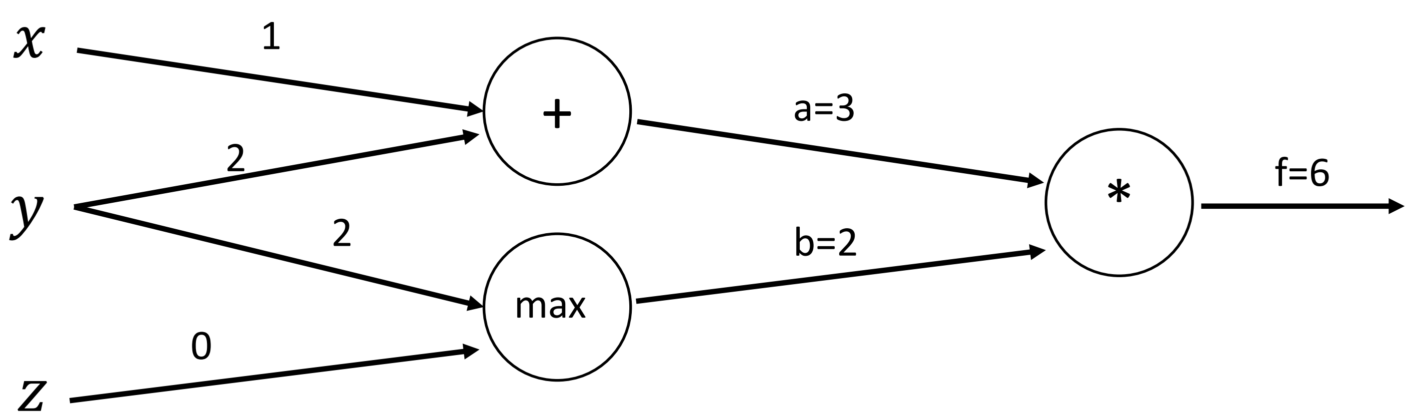
$$f = ab = 6$$

Local gradients:

$$\frac{\partial a}{\partial x} = 1, \frac{\partial a}{\partial y} = 1$$

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1, \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

$$\frac{\partial f}{\partial a} = b = 2, \frac{\partial f}{\partial b} = a = 3$$



An Example

$$f(x, y, z) = (x + y)\max(y, z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps:

$$a = x + y = 3$$

$$\mathrm{b} = \max(y, z) = 2$$

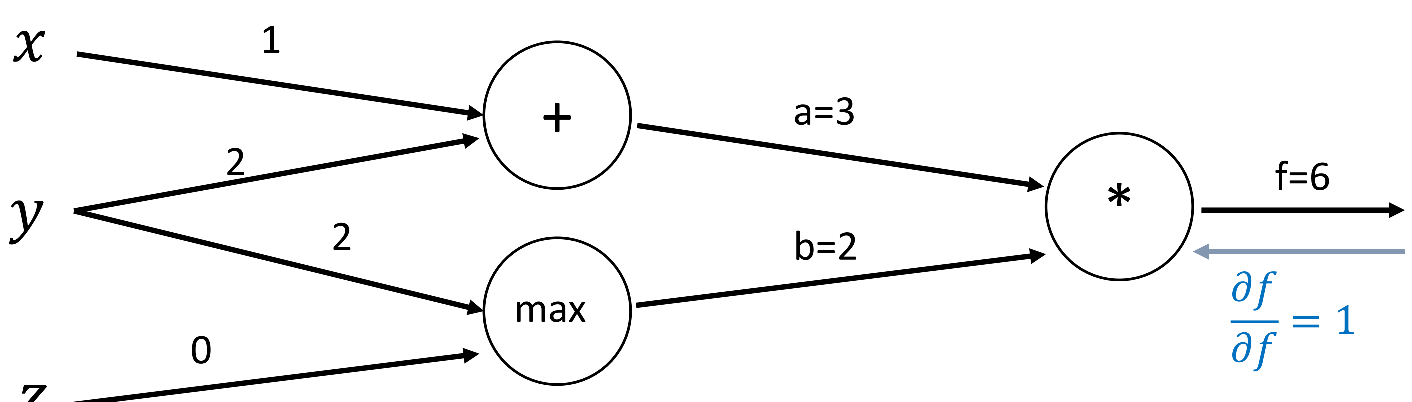
$$f = ab = 6$$

Local gradients:

$$\frac{\partial a}{\partial x} = 1, \frac{\partial a}{\partial y} = 1$$

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1, \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

$$\frac{\partial f}{\partial a} = b = 2, \frac{\partial f}{\partial b} = a = 3$$



upstream * local = downstream

An Example

$$\boxed{\begin{gathered} f(x, y, z) = (x + y)\max(y, z) \\ x = 1, y = 2, z = 0 \end{gathered}}$$

Forward prop steps:

$$a = x + y = 3$$

$$\mathrm{b} = \max(y, z) = 2$$

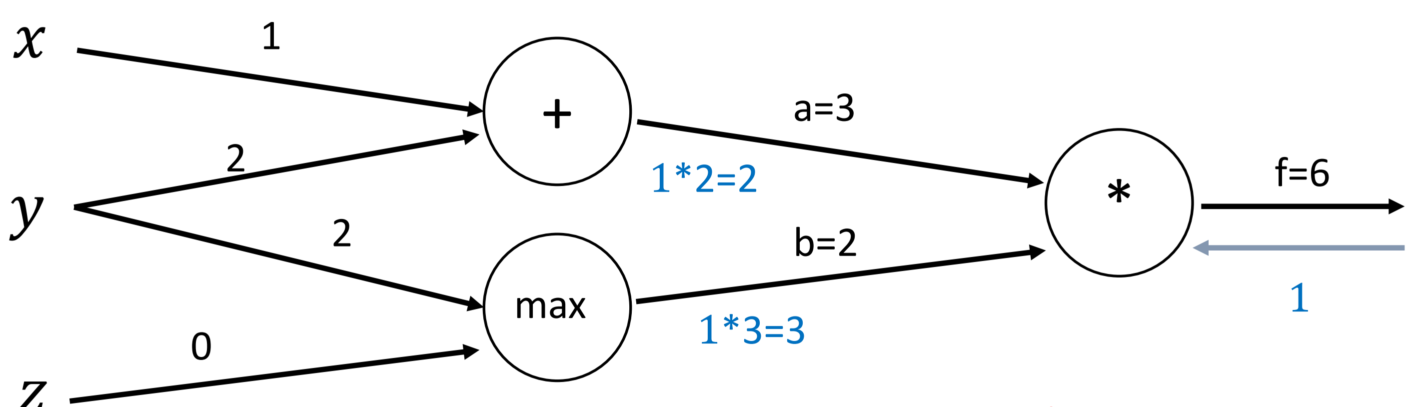
$$f = ab = 6$$

Local gradients:

$$\frac{\partial a}{\partial x} = 1, \frac{\partial a}{\partial y} = 1$$

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1, \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

$$\boxed{\frac{\partial f}{\partial a} = b = 2, \frac{\partial f}{\partial b} = a = 3}$$



upstream * local = downstream

An Example

$$\boxed{\begin{gathered} f(x, y, z) = (x + y)\max(y, z) \\ x = 1, y = 2, z = 0 \end{gathered}}$$

Forward prop steps:

$$a = x + y = 3$$

$$\mathrm{b} = \max(y, z) = 2$$

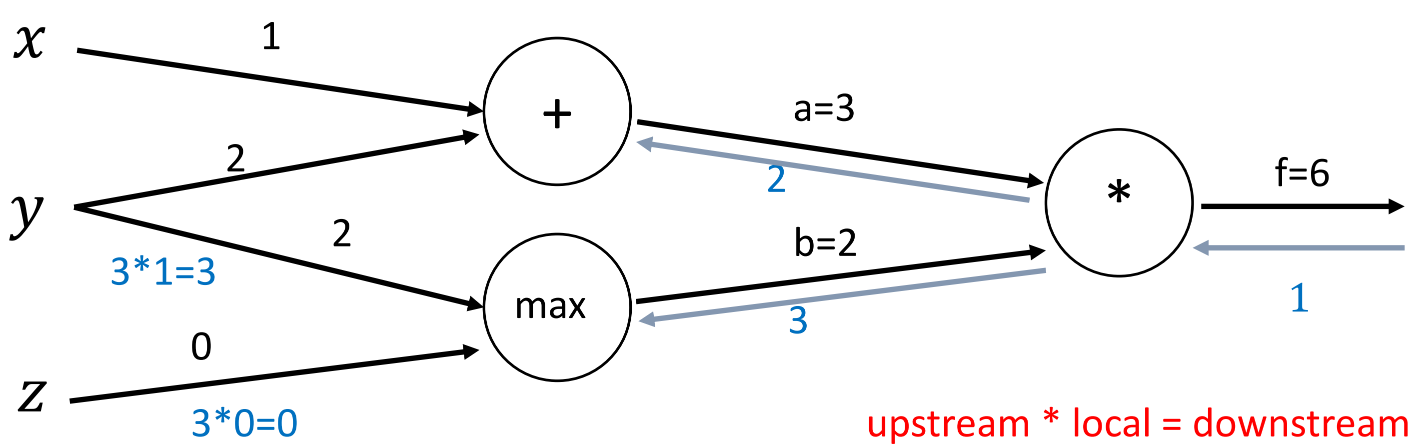
$$f = ab = 6$$

Local gradients:

$$\frac{\partial a}{\partial x} = 1, \frac{\partial a}{\partial y} = 1$$

$$\boxed{\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1, \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0}$$

$$\frac{\partial f}{\partial a} = b = 2, \frac{\partial f}{\partial b} = a = 3$$



upstream * local = downstream

An Example

$$f(x,y,z) = (x+y)\max(y,z)$$
$$x=1, y=2, z=0$$

Forward prop steps:

$$a = x + y = 3$$

$$\mathrm{b} = \max(y,z) = 2$$

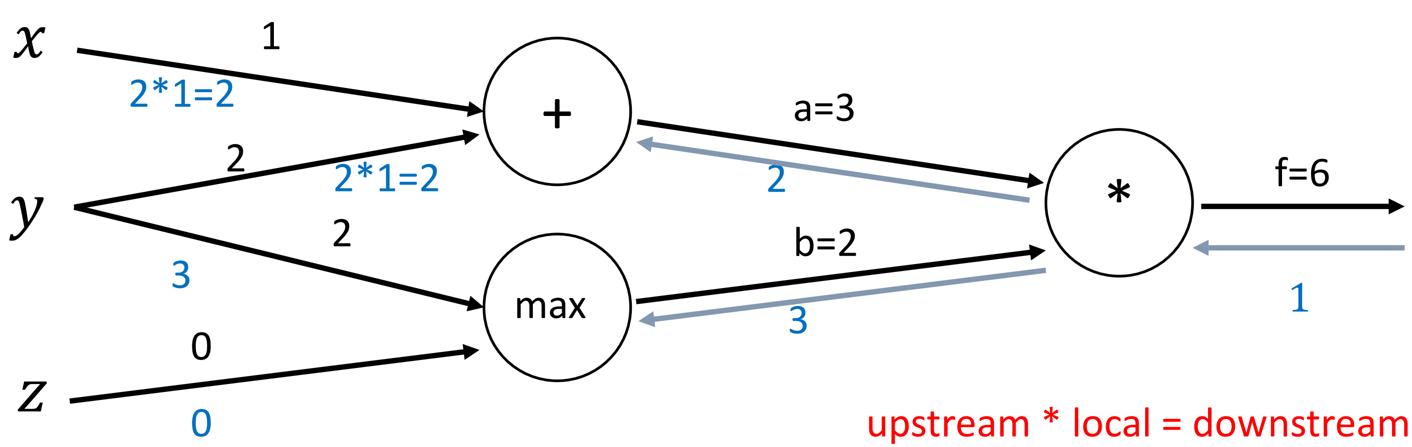
$$f = ab = 6$$

Local gradients:

$$\frac{\partial a}{\partial x} = 1, \frac{\partial a}{\partial y} = 1$$

$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1, \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

$$\frac{\partial f}{\partial a} = b = 2, \frac{\partial f}{\partial b} = a = 3$$



upstream * local = downstream

An Example

$$f(x,y,z) = (x+y)\max(y,z)$$
$$x = 1, y = 2, z = 0$$

Forward prop steps:

$$a = x + y = 3$$

$$\mathrm{b} = \max(y,z) = 2$$

$$f = ab = 6$$

Local gradients:

$$\frac{\partial a}{\partial x} = 1, \frac{\partial a}{\partial y} = 1$$

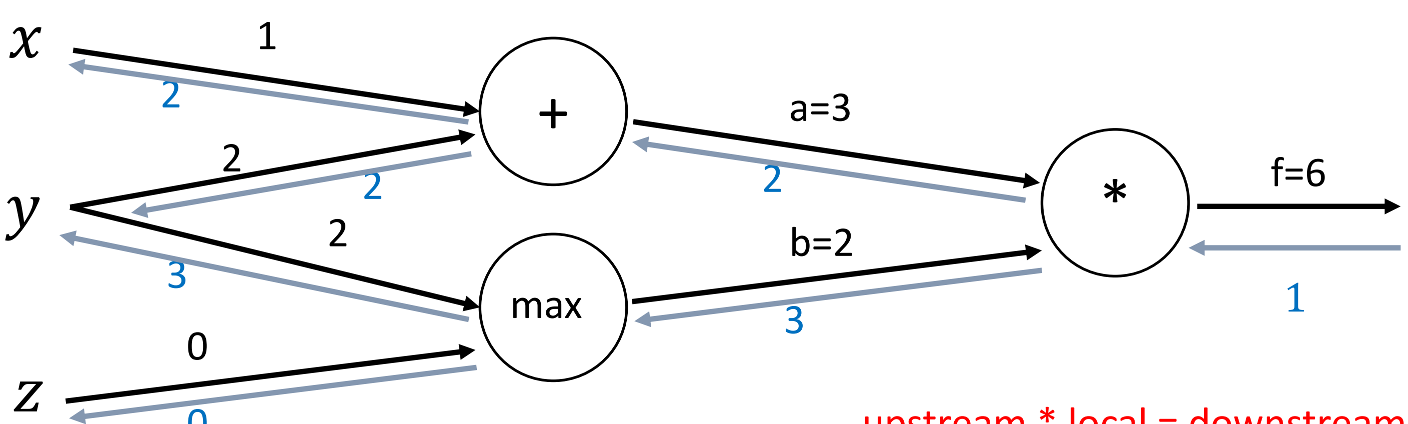
$$\frac{\partial b}{\partial y} = \mathbf{1}(y > z) = 1, \frac{\partial b}{\partial z} = \mathbf{1}(z > y) = 0$$

$$\frac{\partial f}{\partial a} = b = 2, \frac{\partial f}{\partial b} = a = 3$$

$$\frac{\partial f}{\partial x} = 2$$

$$\frac{\partial f}{\partial y} = 3 + 2 = 5$$

$$\frac{\partial f}{\partial z} = 0$$



upstream * local = downstream

Summary

- Forward pass: compute results of operation and save intermediate values

- Backpropagation: recursively apply the chain rule along computational graph to compute gradients
  - [downstream gradient] = [upstream gradient] x [local gradient]

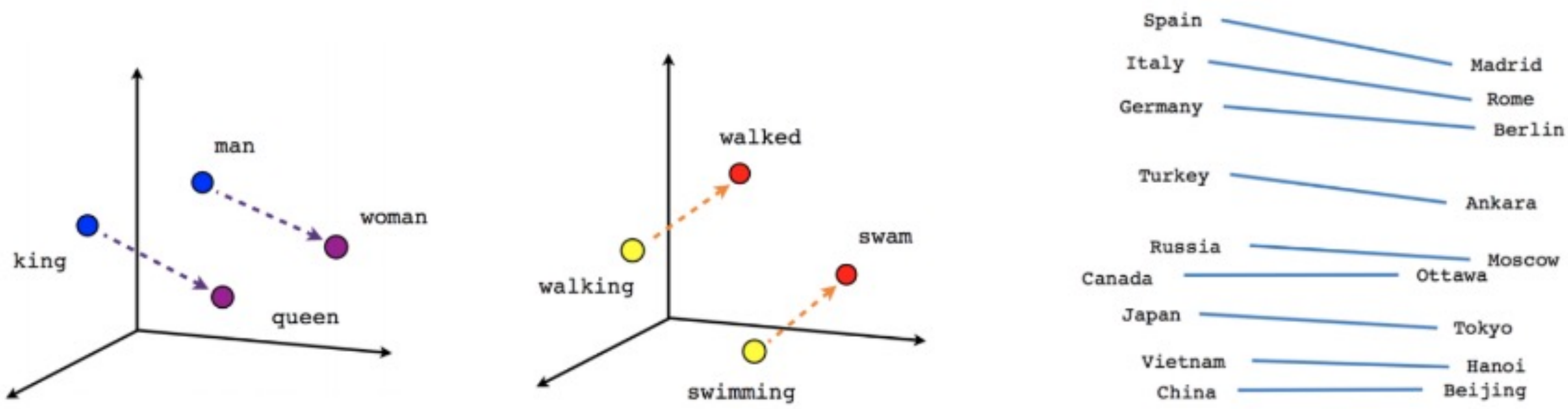
Word Representation: Word2Vec

Shi Yu

THUNLP

Word2Vec

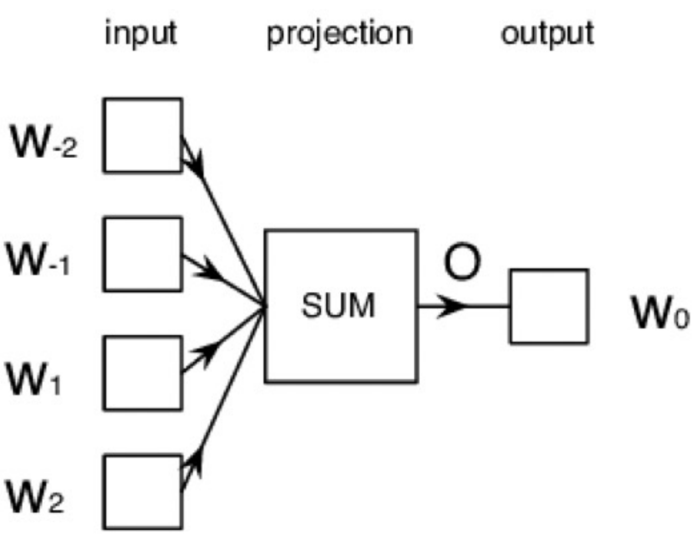
- Word2vec uses shallow neural networks that associate words to distributed representations
- It can capture many linguistic regularities, such as:



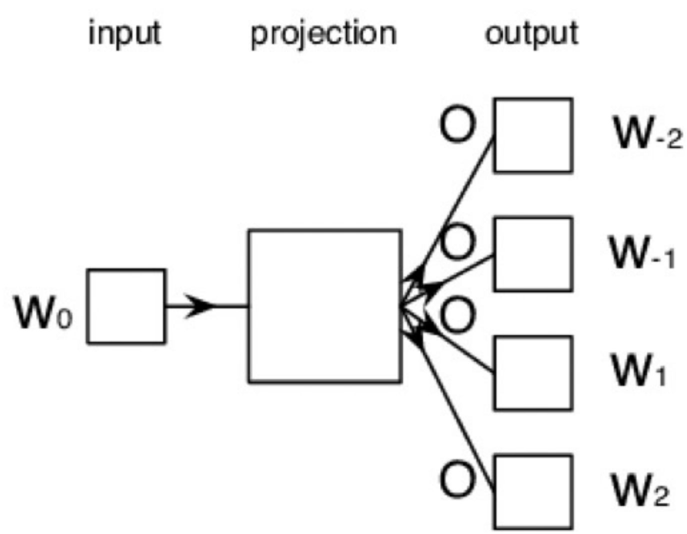
Male-Female   Verb tense   Country-Capital

Typical Models

- Word2vec can utilize two architectures to produce distributed representations of words:
  - Continuous bag-of-words (CBOW)
  - Continuous skip-gram



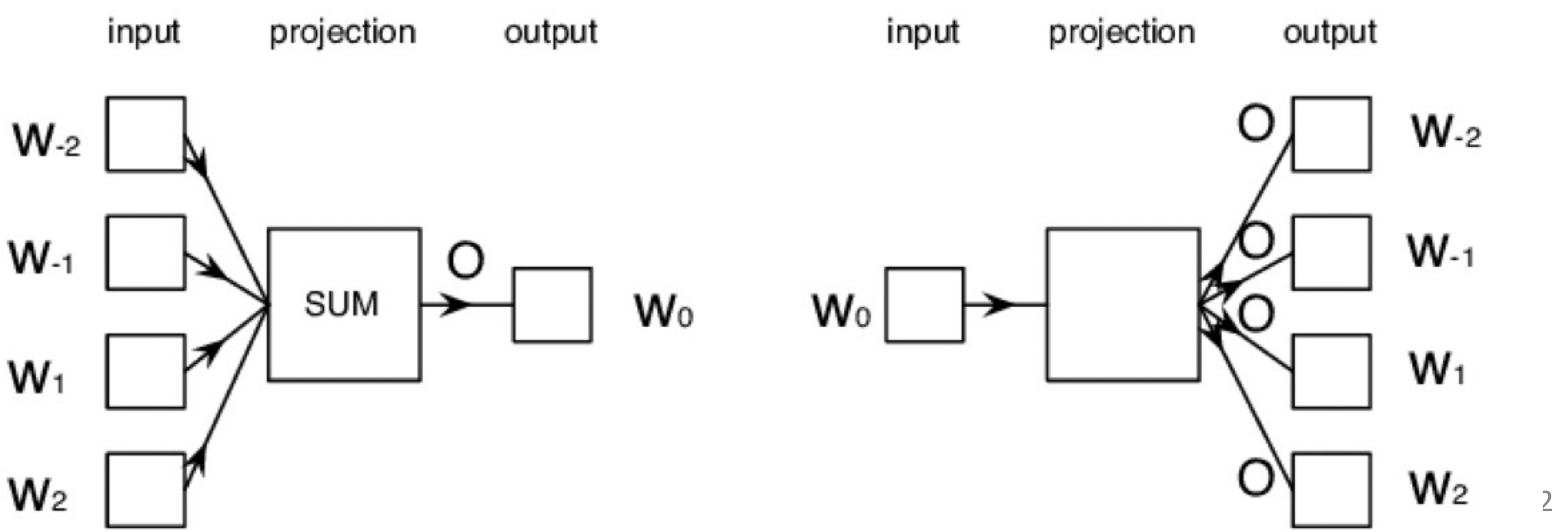
CBOW



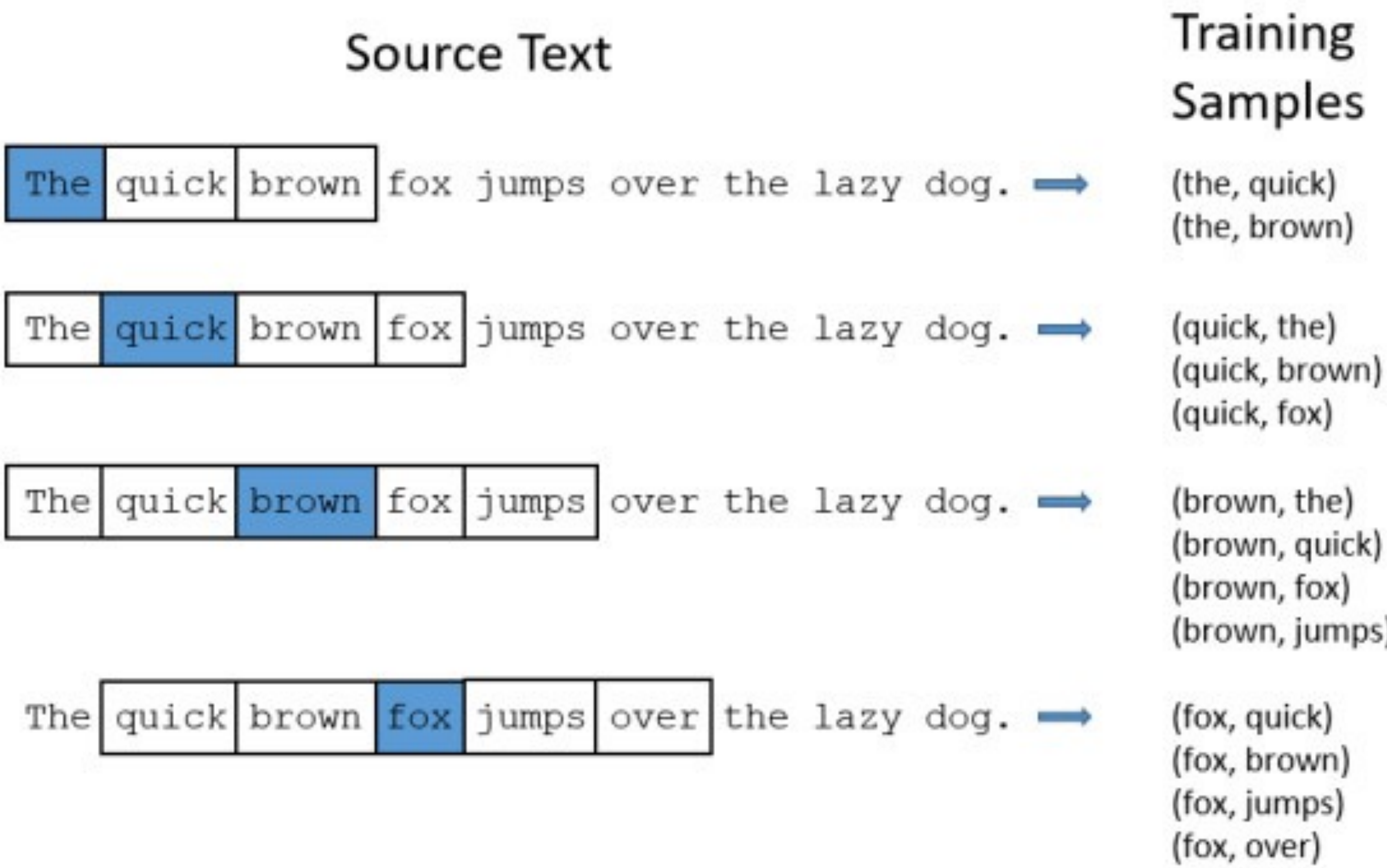
Skip-Gram

Sliding Window

- Word2vec uses a sliding window of a fixed size moving along a sentence
- In each window, the middle word is the target word, other words are the context words
  - Given the context words, CBOW predicts the probabilities of the target word
  - While given a target word, skip-gram predicts the probabilities of the context words



An Example of the Sliding Window



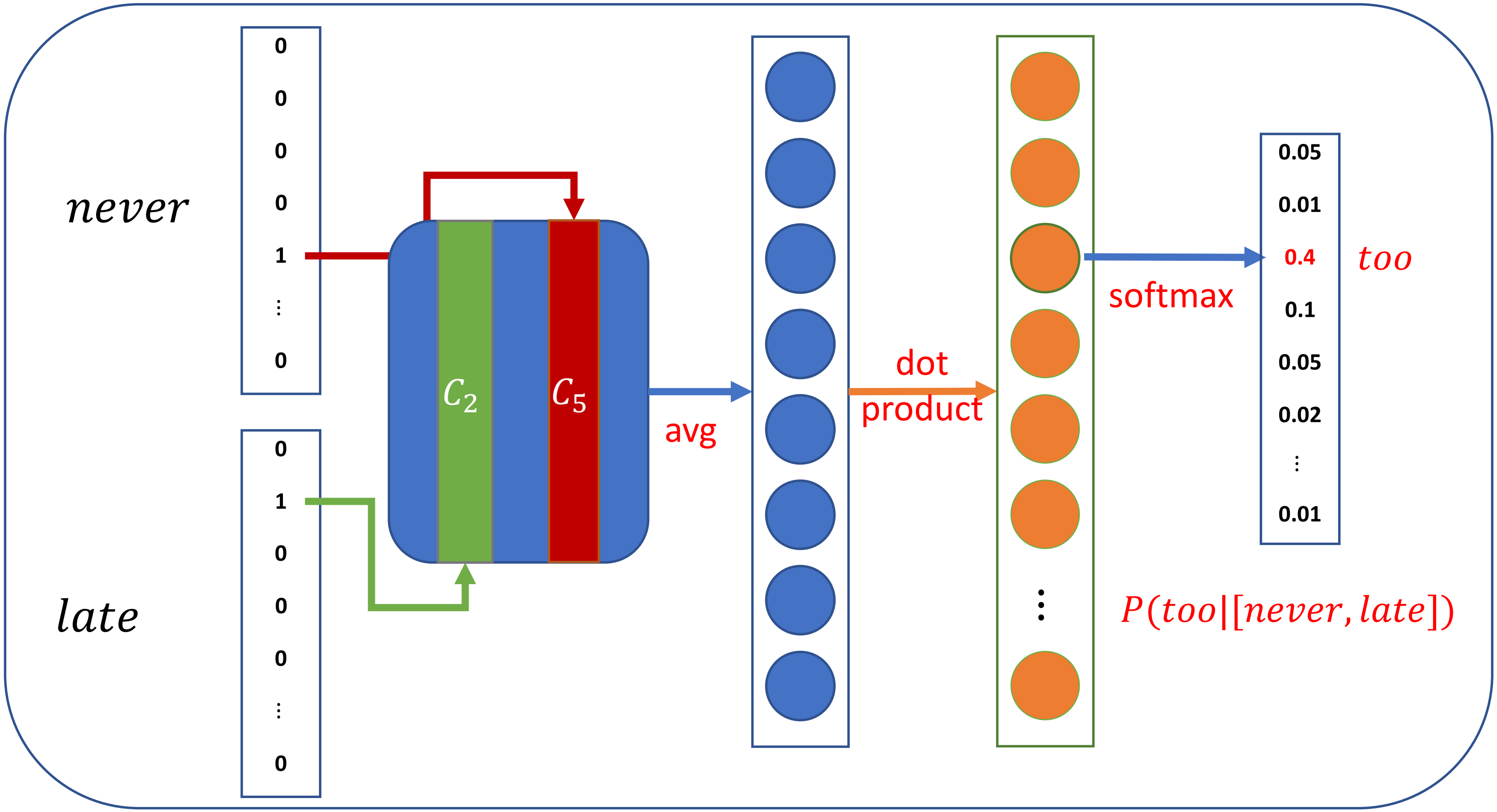
Sliding window size = 5

Continuous Bag-of-Words

- In CBOW architecture, the model predicts the target word given a window of surrounding context words
- According to the bag-of-word assumption: The order of context words does not influence the prediction
  - Suppose the window size is 5
    - $Never\ too\ late\ to\ learn$
    - $P(late|[never, too, to, learn])$ …

Continuous Bag-of-Words

- *Never too late to learn*

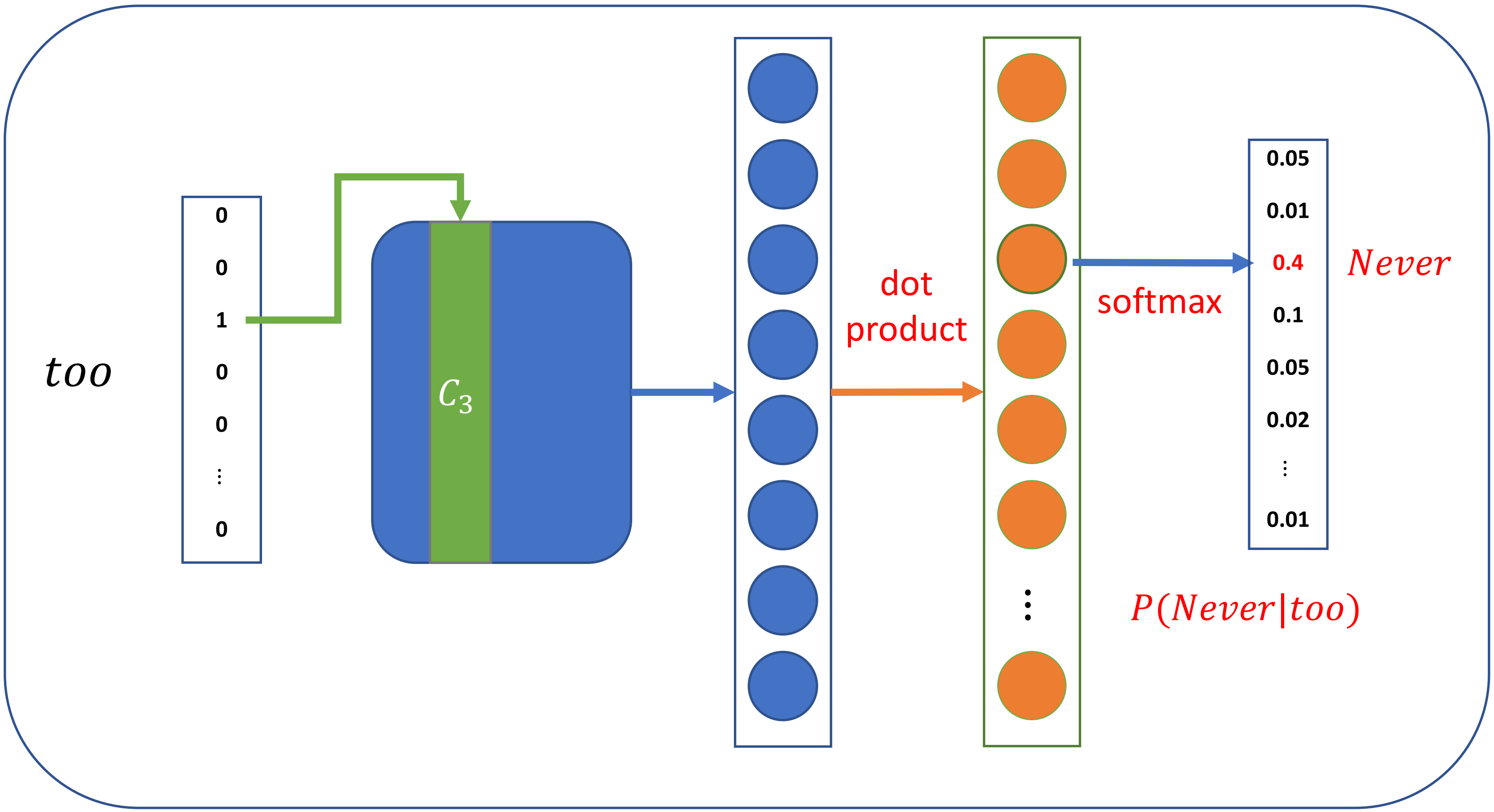


Continuous Skip-Gram

- In skip-gram architecture, the model predicts the context words from the target word
  - Suppose the window size is 5
    - $Never\ too\ late\ to\ learn$
    - $P([too, late]|Never)$, $P([Never, late, to]|too)$, …
  - Skip-gram predict one context word each step, and the training samples are:
    - $P(too|Never)$, $P(late|Never)$, $P(Never|too)$, $P(late|too)$, $P(to|too)$, …

Continuous Skip-Gram

- $Never\ too\ late\ to\ learn$



Problems of Full Softmax

- When the vocabulary size is very large
  - Softmax for all the words every step depends on a huge number of model parameters, which is computationally impractical
  - We need to improve the computation efficiency

Improving Computational Efficiency

- In fact, we do not need a full probabilistic model in word2vec
- There are two main improvement methods for word2vec:
  - Negative sampling
  - Hierarchical softmax

Negative Sampling

- As we discussed before, the vocabulary is very large, which means our model has a tremendous number of weights need to be updated every step
- The idea of negative sampling is, to only update a small percentage of the weights every step

Negative Sampling

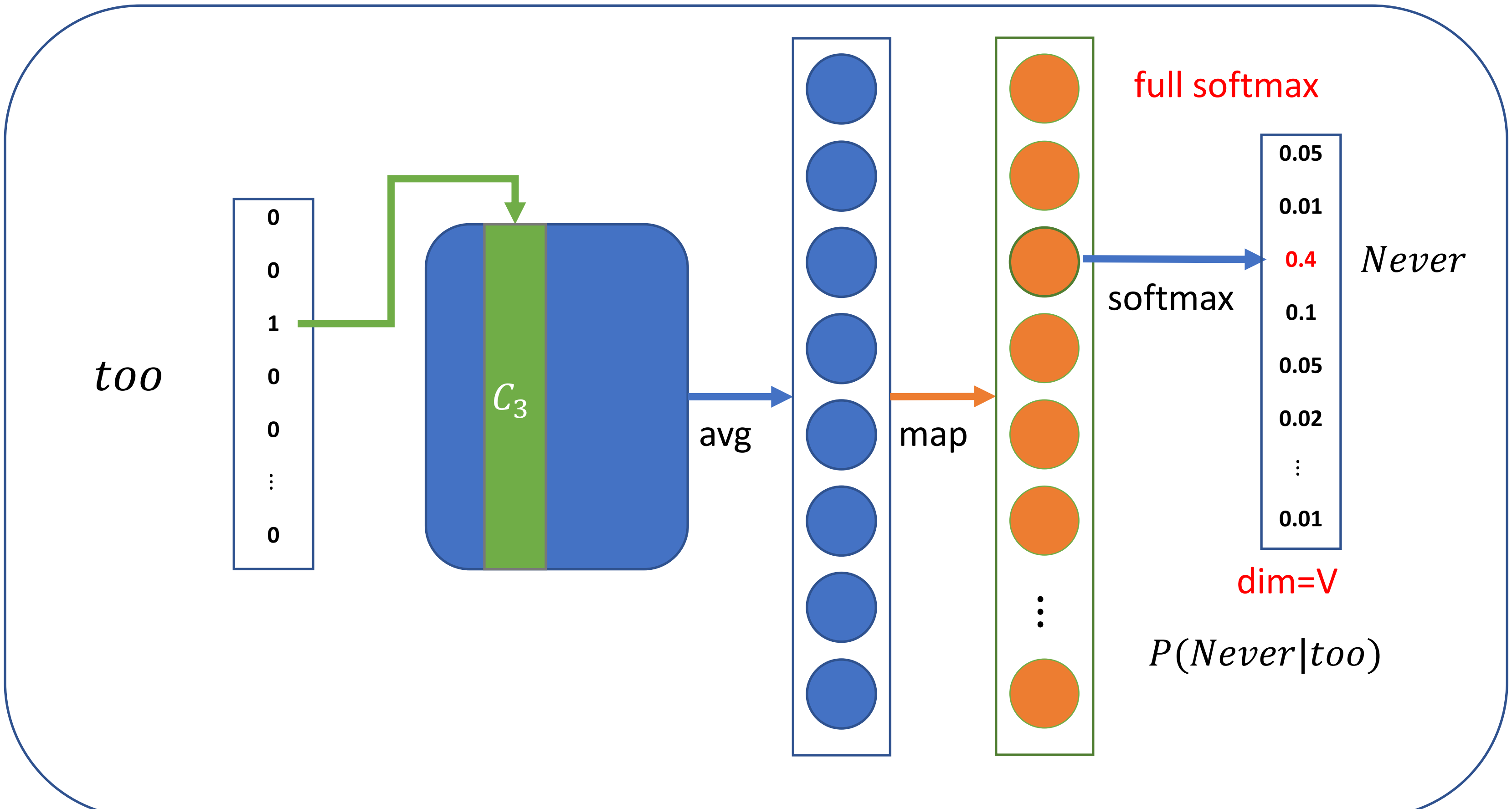
- Since we have the vocabulary and know the context words, we can select a couple of words not in the context word list by probability:

$$P(w_i) = \frac{f(w_i)^{3/4}}{\sum_{j=1}^{V} f(w_j)^{3/4}}$$

$f(w_i)$ is the frequency of w_i, compared to $\frac{f(w_i)}{\sum_{j=1}^{V} f(w_j)}$, this can increase the probability of low-frequency words.

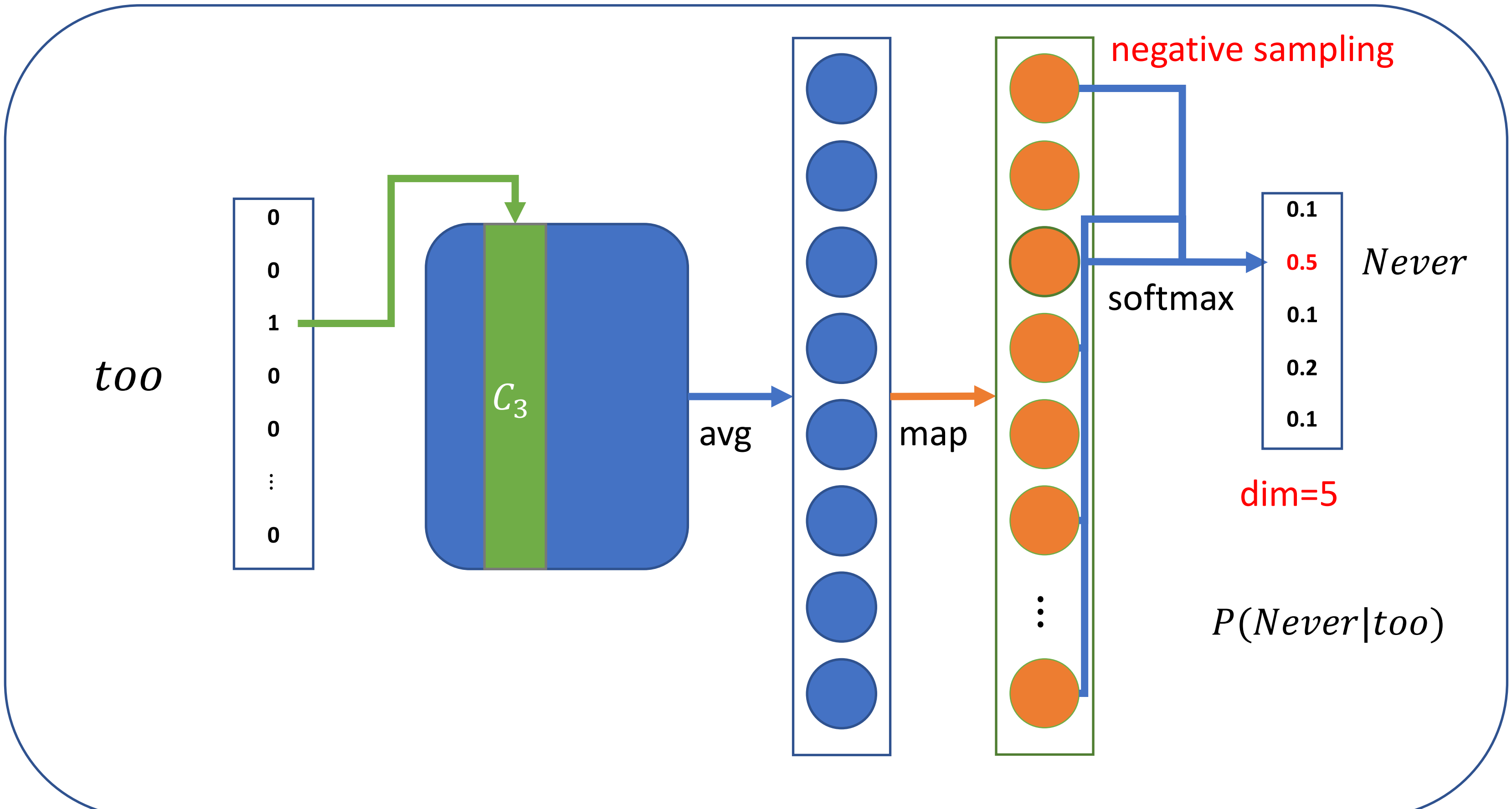
Negative Sampling

- Here is one training step of skip-gram model which computes all probabilities at the output layer



Negative Sampling

- Suppose we only sample 4 negative words:



Negative Sampling

- Then we can compute the loss, and optimize the weights (not all of the weights) every step
- Suppose we have a weight matrix of size $300 \times 10{,}000$, the output size is 5
- We only need to update 300×5 weights, that is only 0.05% of all the weights

Other Tips for Learning Word Embeddings

- **Sub-Sampling.** Rare words can be more likely to carry distinct information, according to which, sub-sampling discards words w with probability:

$$1 - \sqrt{t/f(w)}$$

$f(w)$ is the word frequency and t is an adjustable threshold.

Other Tips for Learning Word Embeddings

- **Soft sliding window.** Sliding window should assign less weight to more distant words
- Define the max size of the sliding window as S_{max}, the actual window size is randomly sampled between 1 and S_{max} for every training sample
- Thus, those words near the target word are more likely to be in the window

Recurrent Neural Networks (RNNs)

Chaoqun He

THUNLP

Sequential Memory

- Key concept for RNNs: Sequential memory during processing sequence data
- Sequential memory of human:
  - Say the alphabet in your head

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

  - Pretty easy

Sequential Memory

- Key concept for RNNs: Sequential memory during processing sequence data
- Sequential memory of human:
  - Say the alphabet backward

Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

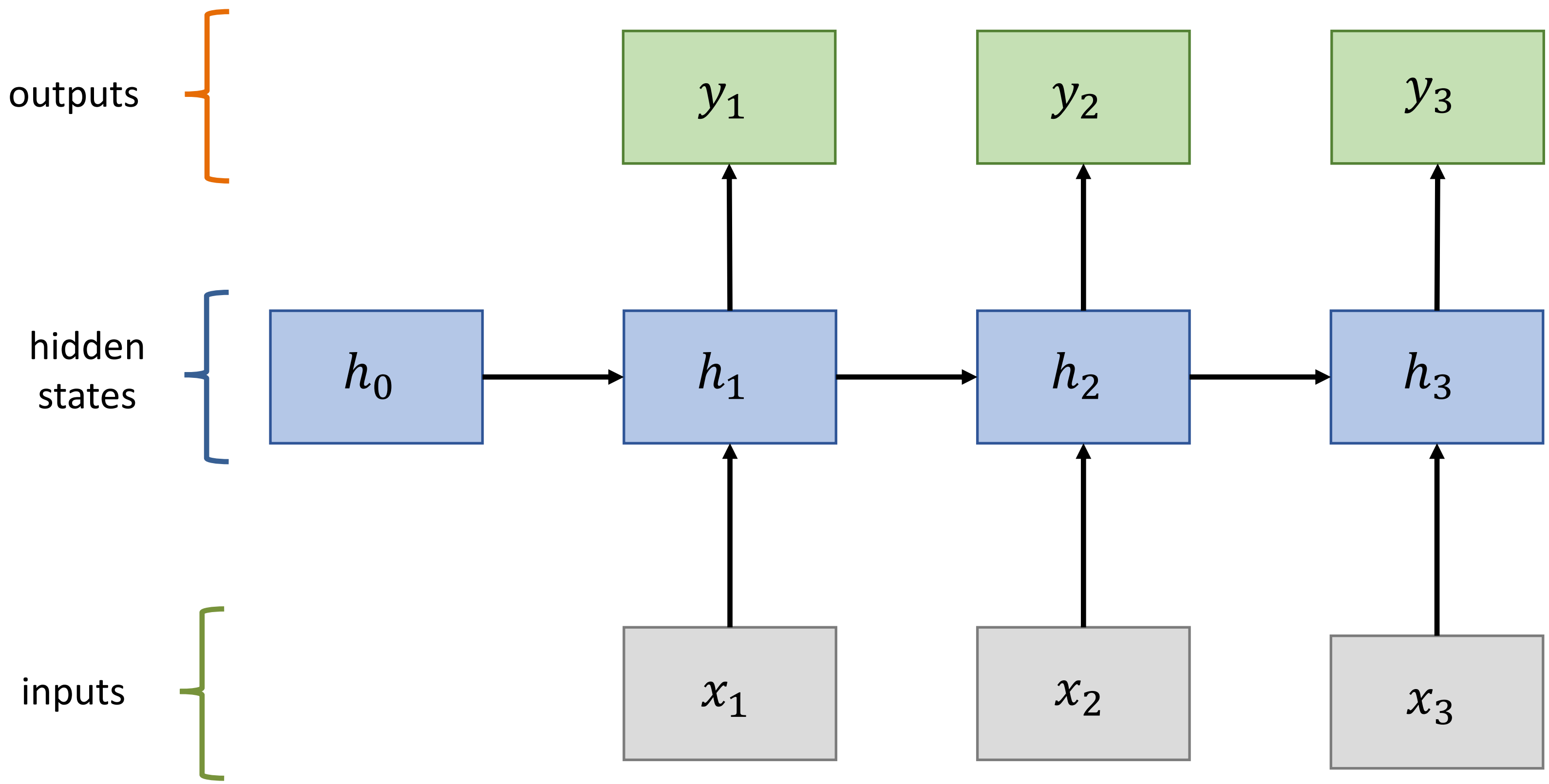
  - Much harder

Sequential Memory

- Definition: a mechanism that makes it easier for your brain to recognize sequence patterns

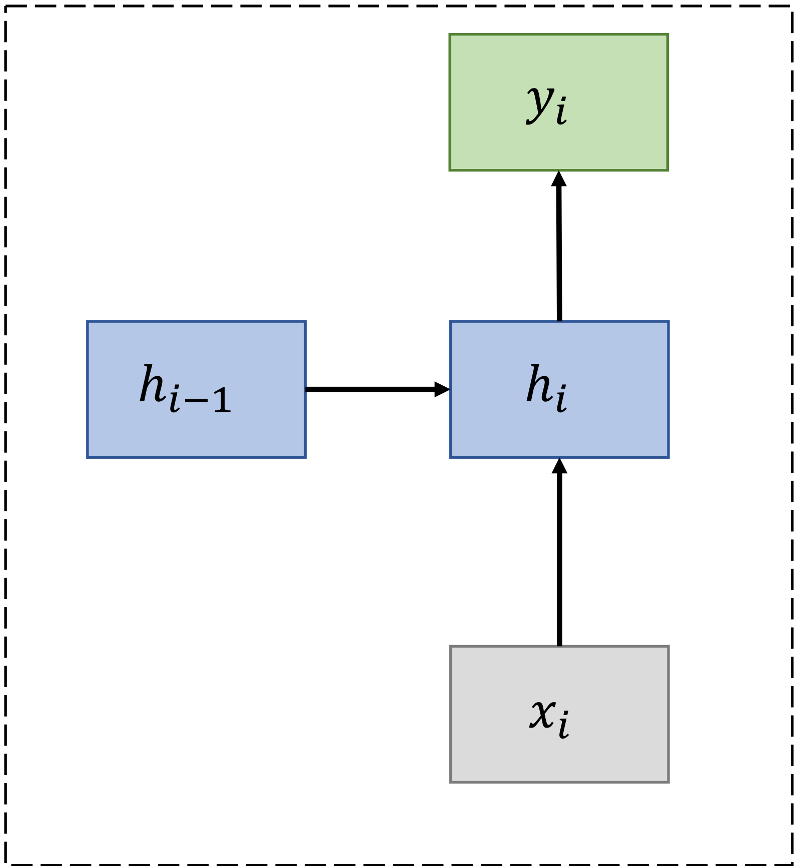
- RNNs update the sequential memory recursively for modeling sequence data

Recurrent Neural Networks



Recurrent Neural Networks

- RNN Cell



$$h_i = \tanh(W_x x_i + W_h h_{i-1} + b)$$

$$y_i = F(h_i)$$

RNN Language Model

hidden states

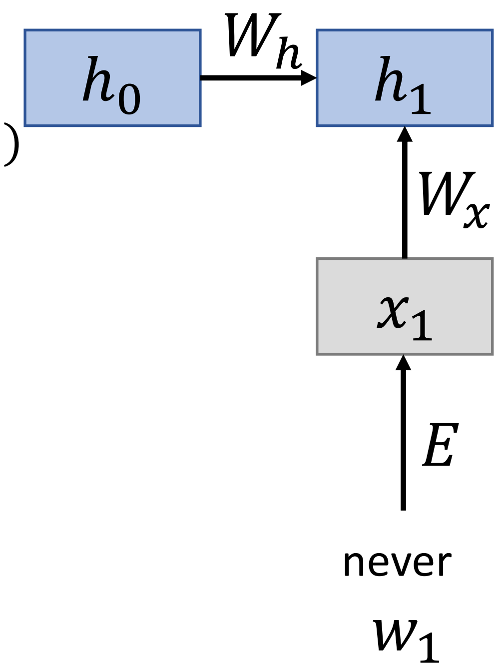
$$h_i = \tanh(W_x x_i + W_h h_{i-1} + b_1)$$

word embeddings

$$x_i = E w_i$$

one-hot vectors

$$w_i \in \mathbb{R}^{|V|}$$



RNN Language Model

hidden states

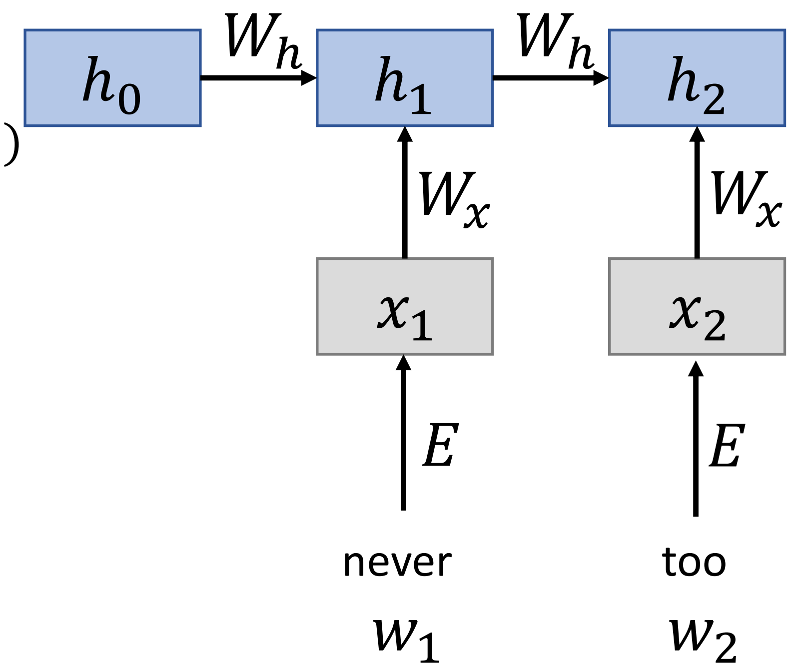
$h_i = \tanh(W_x x_i + W_h h_{i-1} + b_1)$

word embeddings

$x_i = E w_i$

one-hot vectors

$w_i \in \mathbb{R}^{|V|}$



RNN Language Model

hidden states

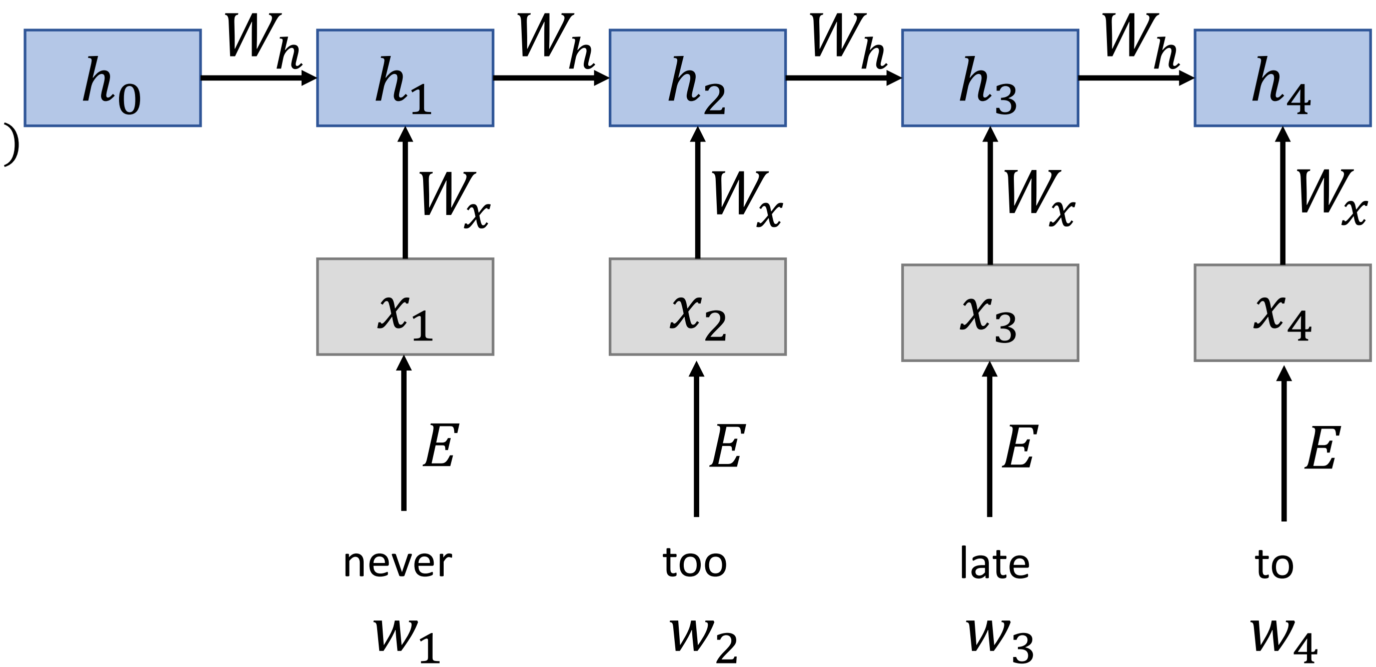
$h_i = \tanh(W_x x_i + W_h h_{i-1} + b_1)$

word embeddings

$x_i = E w_i$

one-hot vectors

$w_i \in \mathbb{R}^{|V|}$



RNN Language Model

output distribution

$y_4 = softmax(Uh_4 + b_2) \in \mathbb{R}^{|V|}$

hidden states

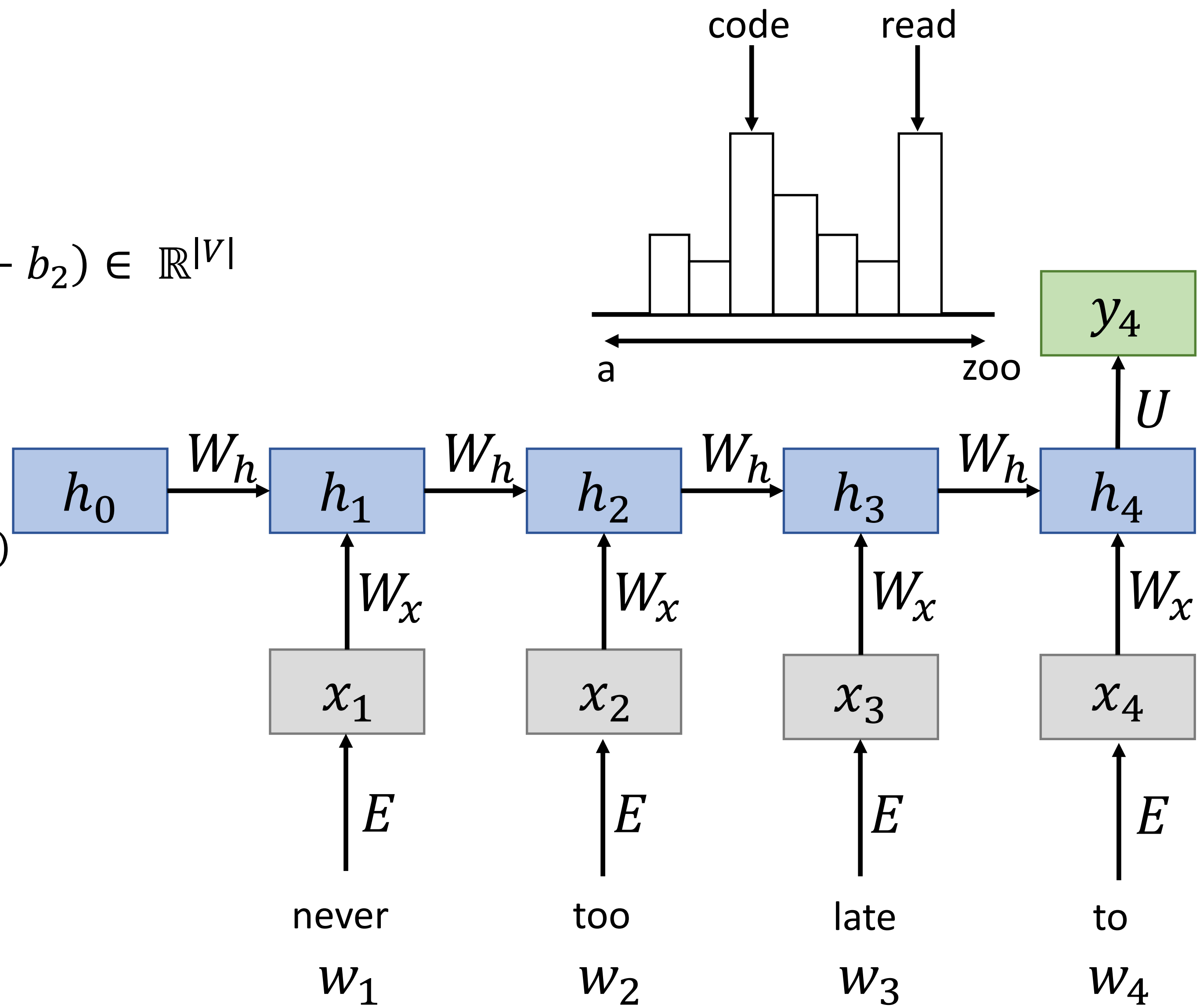
$h_i = \tanh(W_x x_i + W_h h_{i-1} + b_1)$

word embeddings

$x_i = E w_i$

one-hot vectors

$w_i \in \mathbb{R}^{|V|}$



Application Scenarios

- Sequence Labeling
  - Given a sentence, the lexical properties of each word are required
- Sequence Prediction
  - Given the temperature for seven days a week, predict the weather conditions for each day
- Photograph Description
  - Given a photograph, create a sentence that describes the photograph
- Text Classification
  - Given a sentence, distinguish whether the sentence has a positive or negative emotion

Recurrent Neural Networks

- Advantages:
  - Can process any length input
  - Model size does not increase for longer input
  - Weights are shared across timesteps
  - Computation for step i can (in theory) use information from many steps back

- Disadvantages:
  - Recurrent computation is slow
  - In practice, it's difficult to access information from many steps back

Gradient Problem for RNN

- Gradient vanish or explode

$$h_i = \tanh(W_x x_i + W_h h_{i-1} + b)$$

$$\Delta w_1 = \frac{\partial Loss}{\partial w_2} = \frac{\partial Loss}{\partial h_n} \frac{\partial h_n}{\partial h_{n-1}} \frac{\partial h_{n-1}}{\partial h_{n-2}} \cdots \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial w_2}$$

Too many chain derivations

$$\frac{\partial h_n}{\partial h_{n-1}}$$

>1. As the number of layers increases, the gradient update will increase exponentially, i.e., a gradient explosion occurs

<1. As the number of layers increases, the gradient update will decay exponentially, i.e., gradient disappearance occurs.

RNN Variants

Chaoqun He

THUNLP

Solution for Better RNNs

- Better Units!
- The main solution to the Vanishing Gradient Problem is to use a more complex hidden unit computation in recurrence
  - GRU
  - LSTM
- Main ideas:
  - Keep around memories to capture long distance dependencies

Gated Recurrent Unit (GRU)

Chaoqun He

THUNLP

Gated Recurrent Unit (GRU)

- Vanilla RNN computes hidden layer at next time step directly:

$$h_i = \tanh(W_x x_i + W_h h_{i-1} + b)$$

- Introduce gating mechanism into RNN
- Update gate

$$z_i = \sigma(W_x^{(z)} x_i + W_h^{(z)} h_{i-1} + b^{(z)})$$

- Reset gate

$$r_i = \sigma(W_x^{(r)} x_i + W_h^{(r)} h_{i-1} + b^{(r)})$$

- Gates are used to balance the influence of the past and the input

Gated Recurrent Unit (GRU)

- Update gate

$$z_i = \sigma(W_x^{(z)} x_i + W_h^{(z)} h_{i-1} + b^{(z)})$$

- Reset gate

$$r_i = \sigma(W_x^{(r)} x_i + W_h^{(r)} h_{i-1} + b^{(r)})$$

- New activation $\tilde{h}_i$

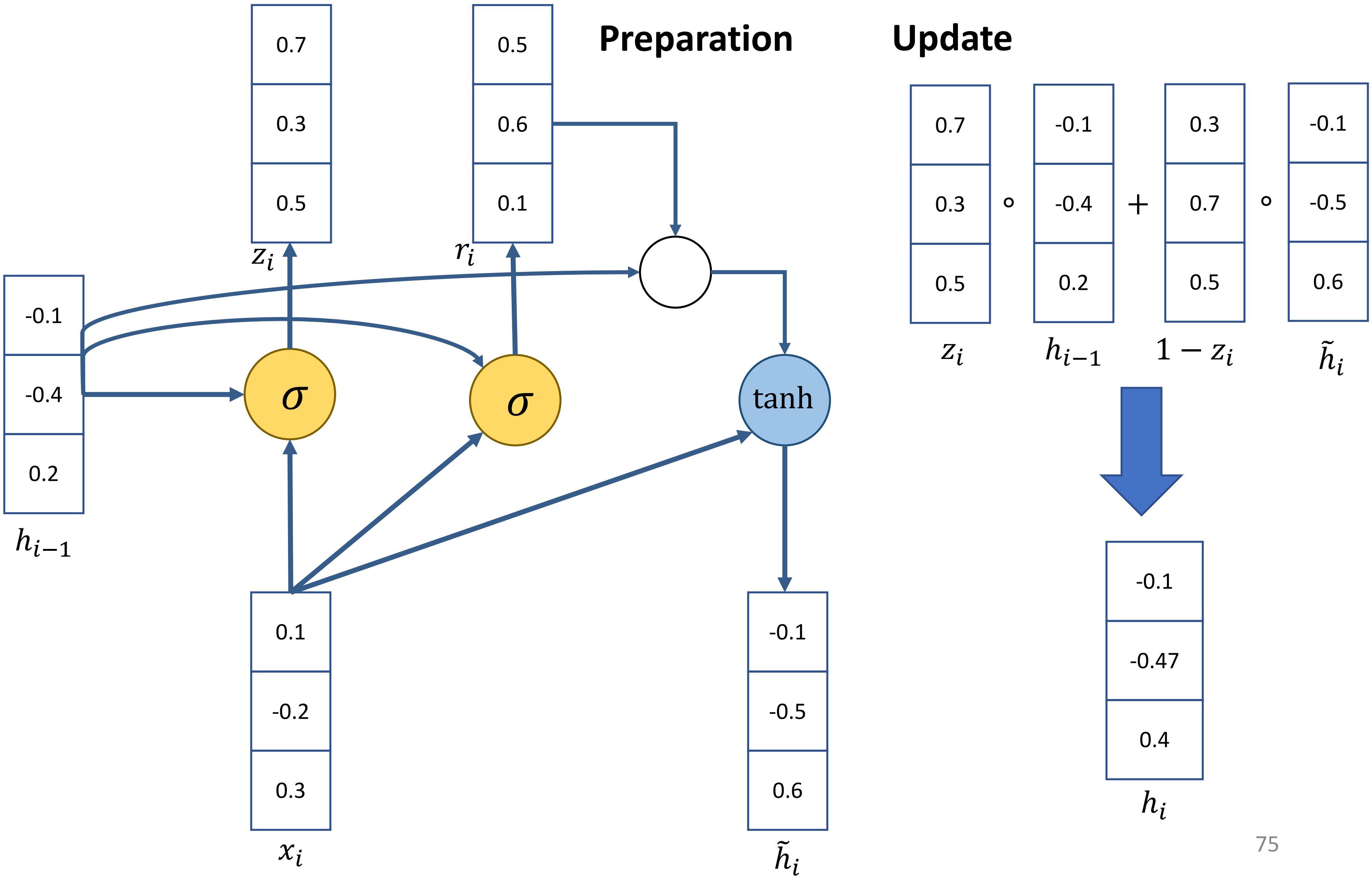
$$\tilde{h}_i = \tanh(W_x x_i + r_i * W_h h_{i-1} + b)$$

- Final hidden state h_i

$$h_i = z_i * h_{i-1} + (1 - z_i) * \tilde{h}_i$$

- Where $*$ refers to element-wise product

Gated Recurrent Unit (GRU)



Gated Recurrent Unit (GRU)

- If reset r_i is close to 0

$$\tilde{h}_i \approx \tanh(W_x x_i + 0 * W_h h_{i-1} + b)$$

$$\tilde{h}_i \approx \tanh(W_x x_i + b)$$

- Ignore previous hidden state, which indicates the current activation is irrelevant to the past.

- E.g., at the beginning of a new article, the past information is useless for the current activation.

Gated Recurrent Unit (GRU)

- Update gate z_i controls how much of past state should matter compared to the current activation.

- If z_i close to 1, then we can copy information in that unit through many time steps!

$$h_i = 1 * h_{i-1} + (1 - 1) * \tilde{h}_i = h_{i-1}$$

- If z_i close to 0, then we drop information in that unit and fully take the current activation.

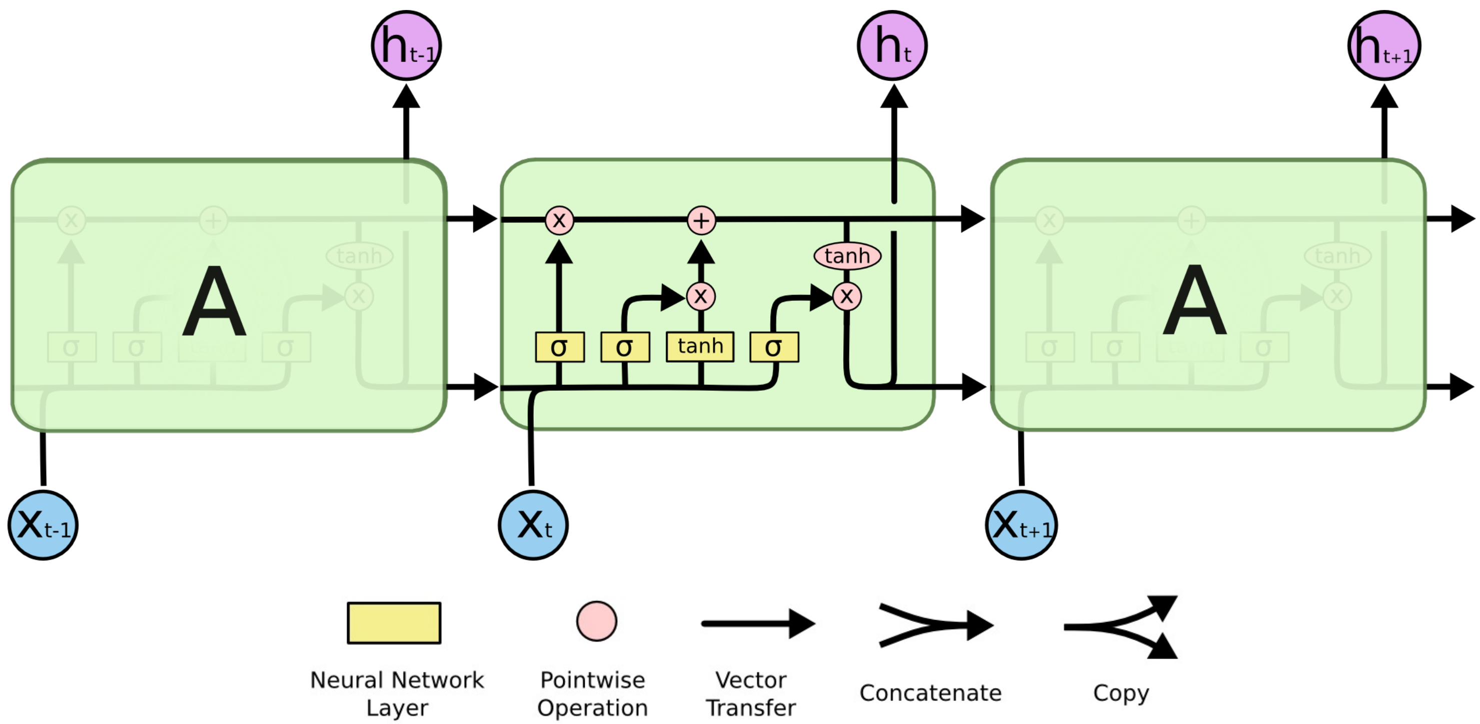
Long Short-Term Memory Network (LSTM)

Chaoqun He

THUNLP

Long Short-Term Memory Network

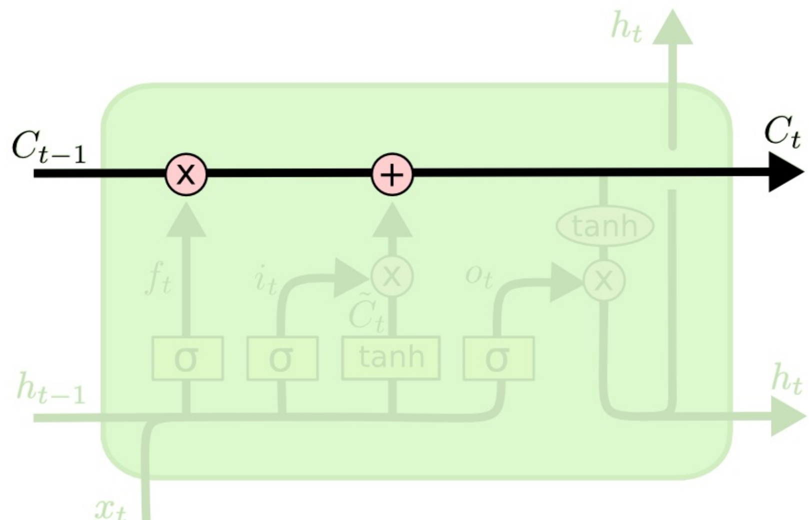
- Long Short-Term Memory network (LSTM)
- LSTM are a special kind of RNN, capable of learning long-term dependencies like GRU



By Chris Olah: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Long Short-Term Memory Network

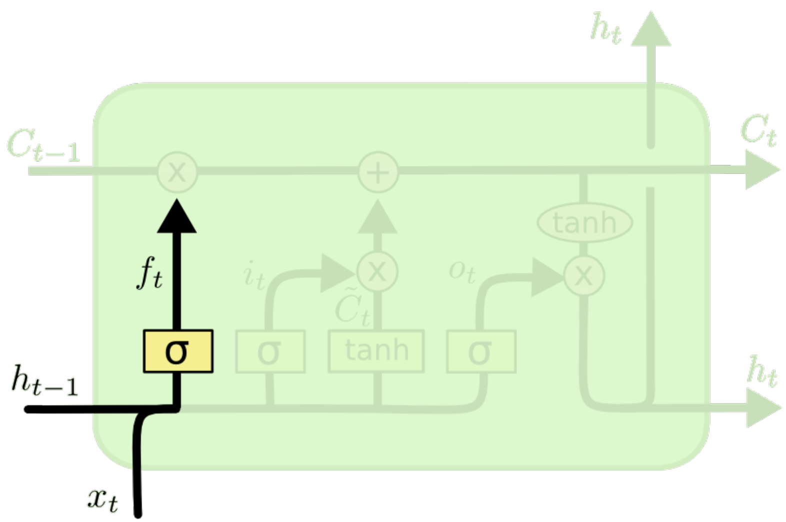
- The key to LSTMs is the cell state C_t



  - Extra vector for capturing long-term dependency
  - Runs straight through the entire chain, with only some minor linear interactions
  - Easy to remove or add information to the cell state

Long Short-Term Memory Network

- The first step is to decide what information to throw away from the cell state
- Forget gate f_t

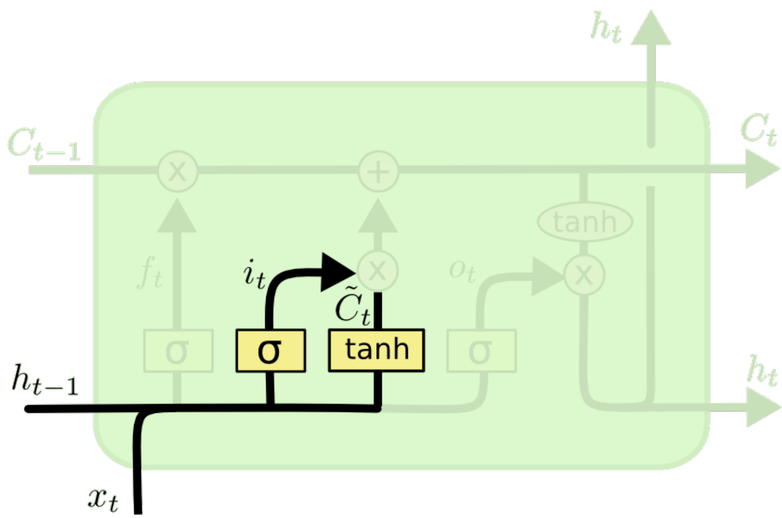


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

- Where $[h_{t-1}, x_t]$ is the concatenation of vectors
- Forget past if $f_t = 0$

Long Short-Term Memory Network

- The next step is to decide what information to store in the cell state
- Input gate i_t and new candidate cell state $\tilde{C}_t$



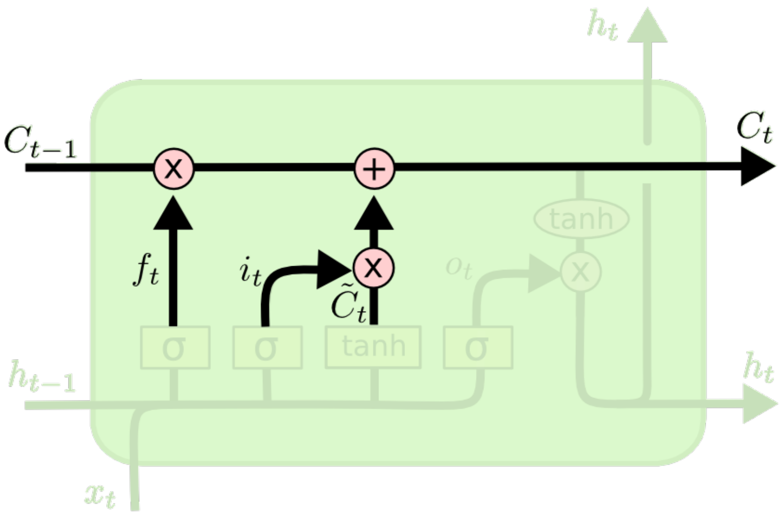
$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

- Recall z_t and $\tilde{h}_t$ in GRUs

Long Short-Term Memory Network

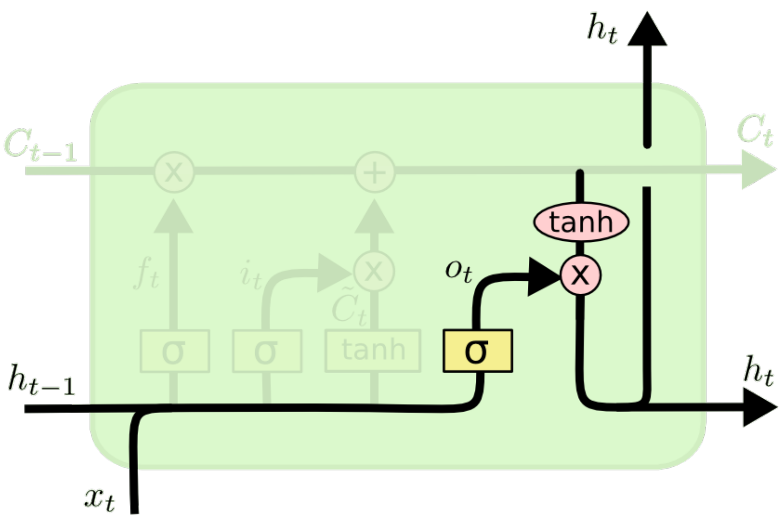
- Update the old cell state C_{t-1}
- Combine the results from the previous two steps



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Long Short-Term Memory Network

- The final step is to decide what information to output
- Adjust the sentence information for a specific word representation



$$o_t = \sigma\left(W_o \left[h_{t-1}, x_t\right] + b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Long Short-Term Memory Network

- Powerful especially when stacked and made even deeper (each hidden layer is already computed by a deep internal network)

- Very useful if you have plenty of data

Bidirectional RNNs

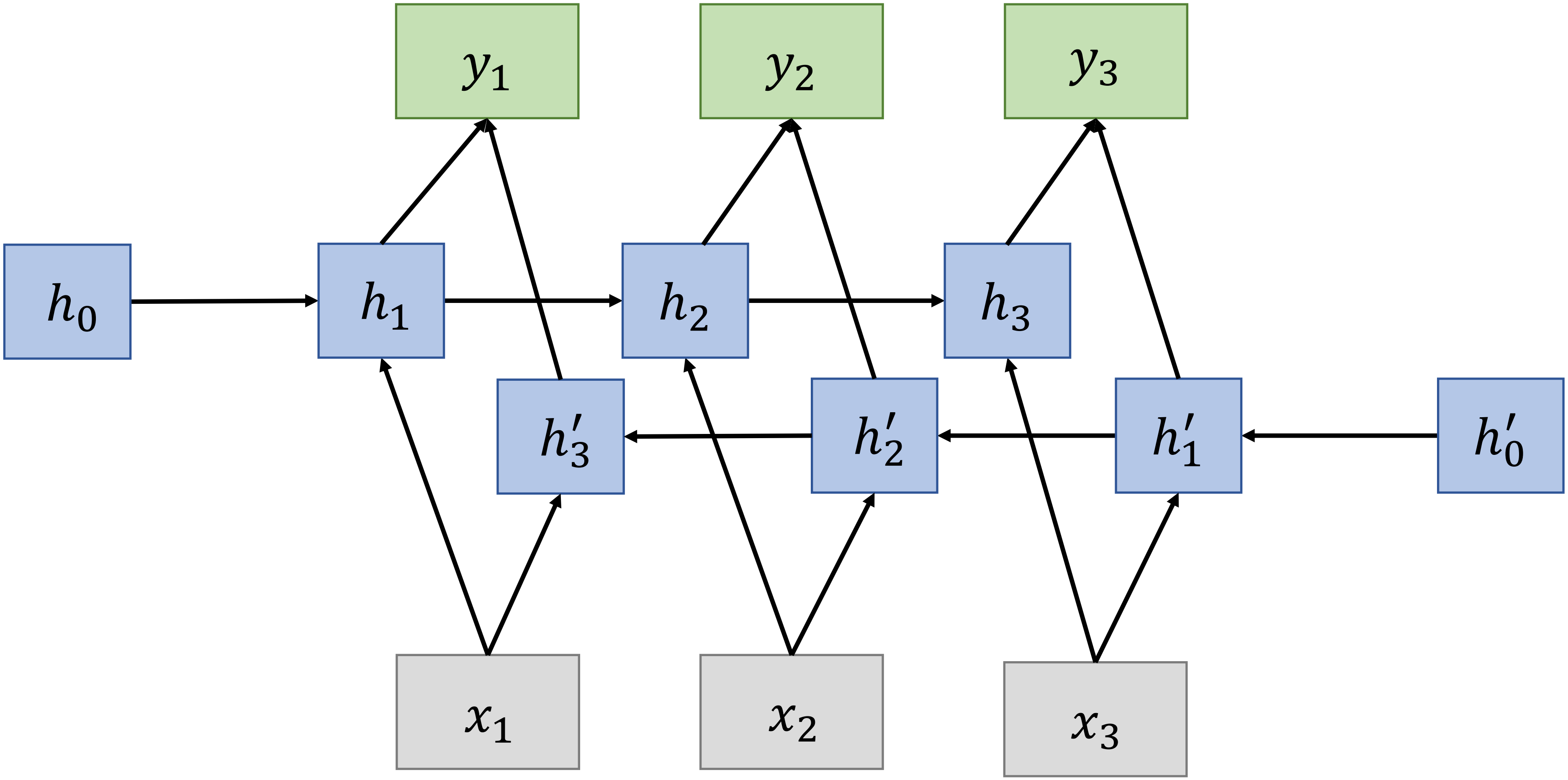
Chaoqun He

THUNLP

Bidirectional RNNs

- In traditional RNNs, the state at time t only captures information from the past
$$h_t = f(x_{t-1}, \dots, x_2, x_1)$$
- Problem: in many applications, we want to have an output y_t depending on the whole input sequence
- For example
  - Handwriting recognition
  - Speech recognition

Bidirectional RNNs



Summary

- Recurrent Neural Network
  - Sequential Memory
  - Gradient Problem for RNN
- RNN Variants
  - Gated Recurrent Unit (GRU)
  - Long Short-Term Memory Network (LSTM)
  - Bidirectional Recurrent Neural Network

Convolutional Neural Networks (CNNs)

Chaoqun He

THUNLP

CNN for Sentence Representation

- Convolutional Neural Networks (CNNs)
  - Generally used in Computer Vision (CV)
  - Achieve promising results in a variety of NLP tasks:
    - Sentiment classification
    - Relation classification
    - …
- CNNs are good at extracting local and position-invariant patterns
  - In CV, colors, edges, textures, etc.
  - In NLP, phrases and other local grammar structures

CNN for Sentence Representation

- CNNs extract patterns by:
  - Computing representations for all possible n-gram phrases in a sentence.
  - Without relying on external linguistic tools (e.g., dependency parser)

possible n-gram phrases

The plane is taking off

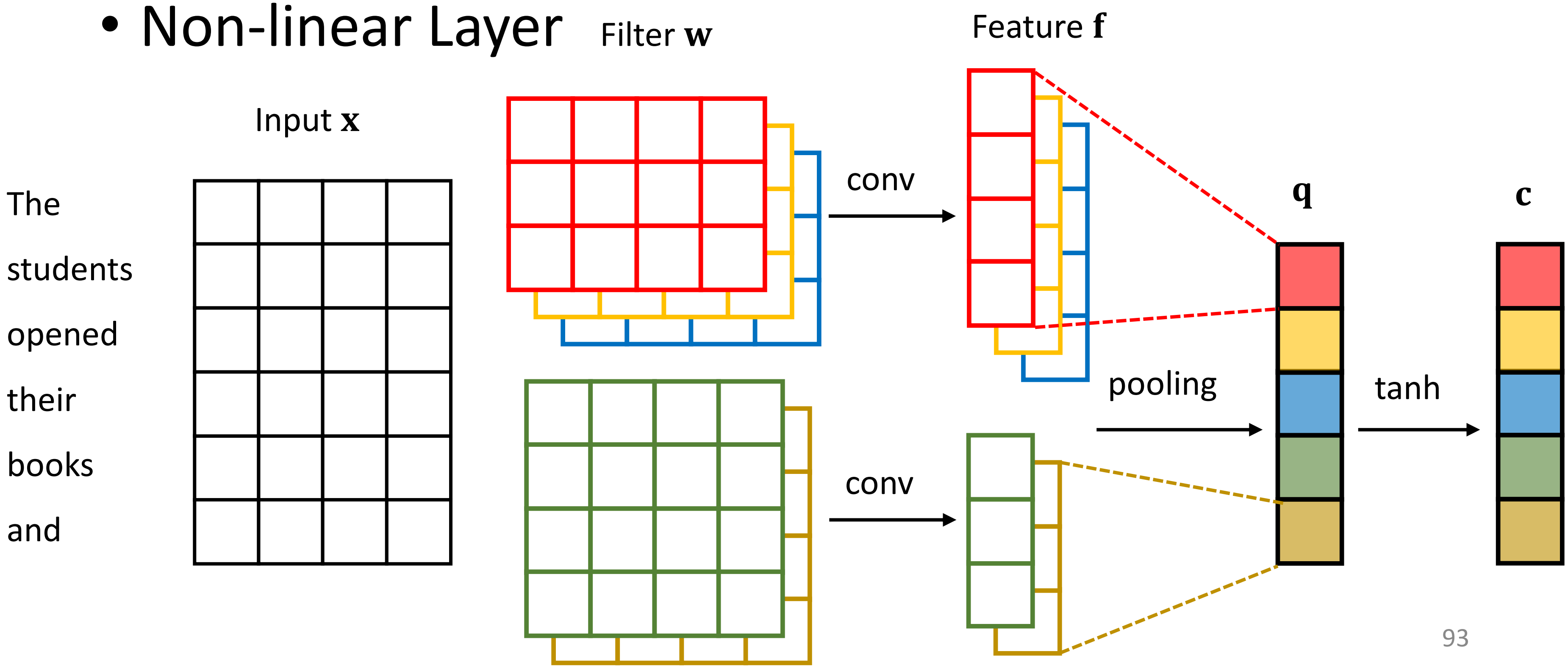
Bigram: *The plane, plane is, is taking, taking off*

Trigram: *The plane is, plane is taking, is taking off*

n-gram: …

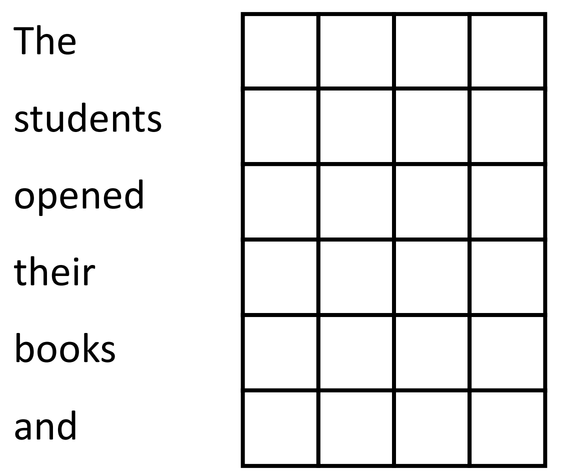
Architecture

- Input Layer
- Convolutional Layer
- Max-pooling Layer
- Non-linear Layer



Input Layer

- Transform words into input representations $\mathbf{x}$ via word embeddings
- $\mathbf{x} \in \mathrm{R}^{m \times d}$: input representation
  - m is the length of sentence
  - d is the dimension of word embeddings



Convolution Layer

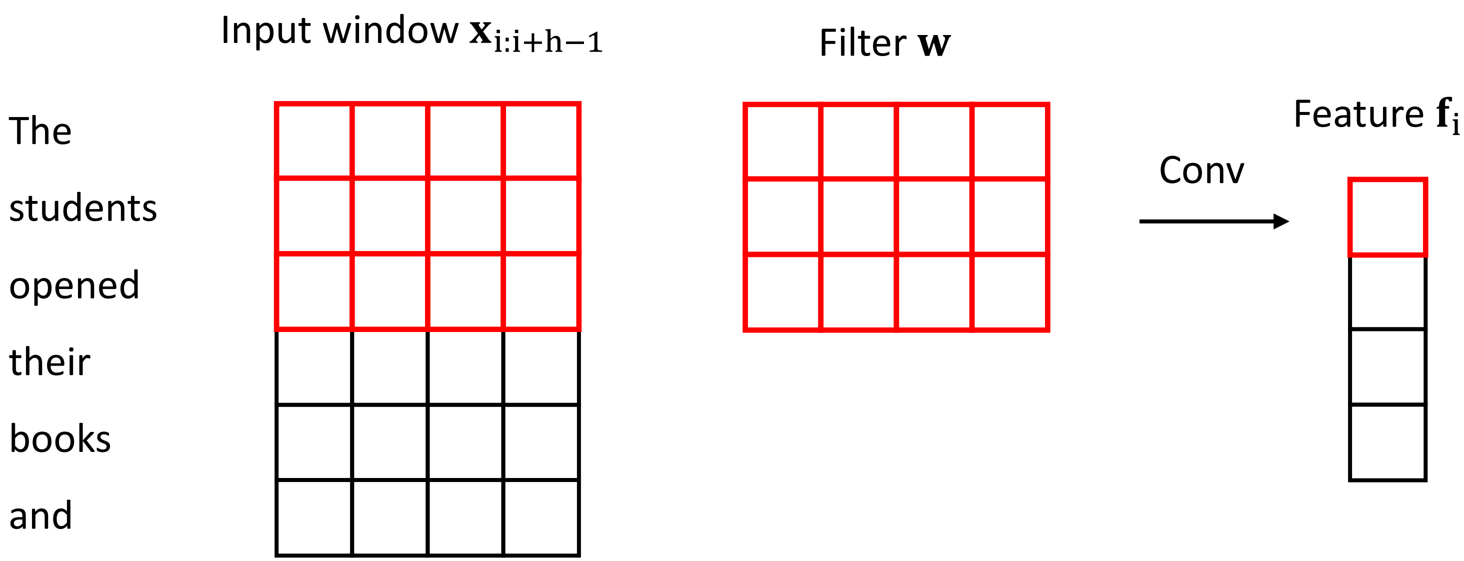
- Extract feature representation from input representation via a sliding convolving filter
  - $\mathbf{x} \in \mathrm{R}^{m \times d}$: input representation
  - $\mathbf{x}_{i:i+j} \in \mathrm{R}^{(j+1)d}$: (j+1)-gram representation, concatenation of $\mathbf{x}_i, \mathbf{x}_{i+1}, \dots, \mathbf{x}_{i+j}$
  - $\mathbf{w} \in \mathrm{R}^{h \times d}$: convolving filter, b is a bias term (h is window size)
  - $\mathbf{f} \in \mathrm{R}^{n-h+1}$: convolved feature representation

  - $\cdot$ is dot product

Convolution Layer

- Extract feature representation from input representation via a sliding convolving filter

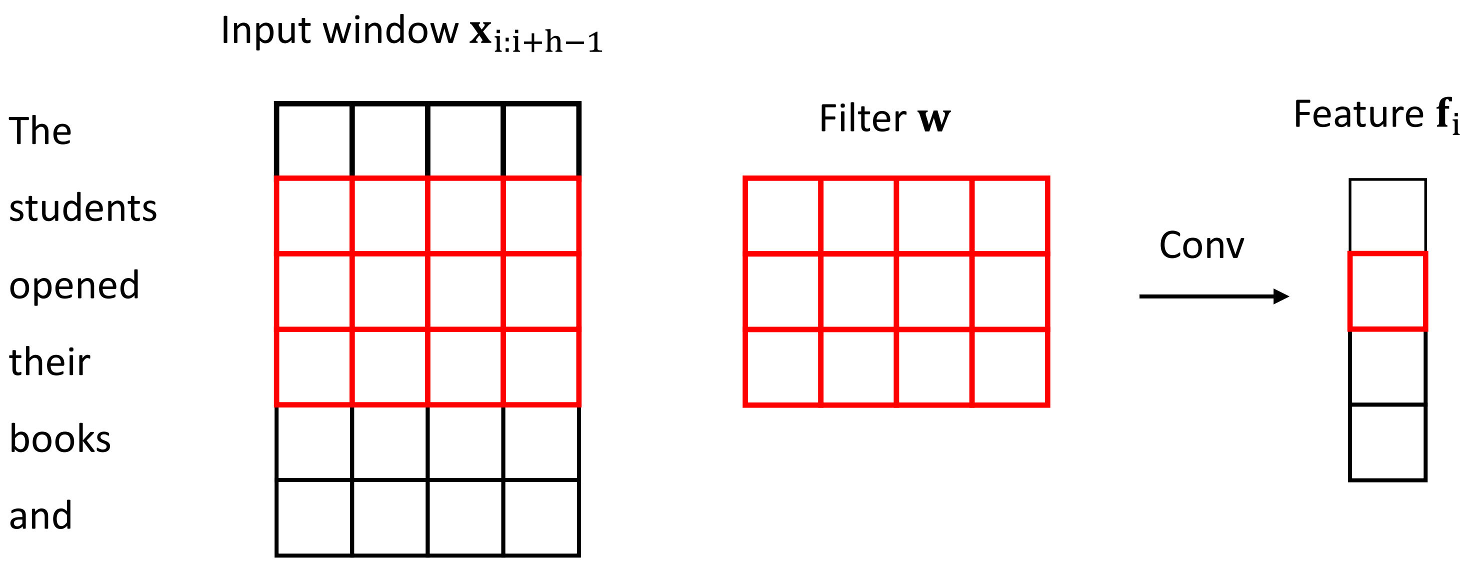
$$\mathbf{f}_{\mathrm{i}} = \mathbf{w} \cdot \mathbf{x}_{\mathrm{i:i+h-1}} + \mathrm{b}, \qquad \mathrm{i} = 1,2,\ldots,\mathrm{n}-\mathrm{h}+1$$



Convolution Layer

- Extract feature representation from input representation via a sliding convolving filter

$$\mathbf{f}_\mathrm{i} = \mathbf{w} \cdot \mathbf{x}_{\mathrm{i:i+h-1}} + \mathrm{b}, \qquad \mathrm{i} = 1,2, \dots, \mathrm{n-h+1}$$



Application Scenarios

- Object Detection
  - You Only Look Once: Unified, Real-Time Object Detection
- Video Classification
  - Large-scale Video Classification with Convolutional Neural Networks
- Speech Recognition
  - Convolutional, Long Short-Term Memory, fully connected Deep Neural Networks
- Text Classification
  - Convolutional Neural Networks for Sentence Classification

Compare CNN with RNN

- CNN vs. RNN

	CNNs	**RNNs**
Advantages	Extracting local and position-invariant features	Modeling long-range context dependency
Parameters	Less parameters	More parameters
Parallelization	Better parallelization within sentences	Cannot be parallelized within sentences

Summary

- Convolutional Neural Network
  - Architecture
    - Input layer
    - Convolution layer
    - Max-pooling layer
    - Non-linear layer
  - Extract local features
  - Capture different n-gram patterns

NLP Pipeline Tutorial（PyTorch）

Jing Yi

THUNLP

Pipeline for Deep Learning

- prepare data
- build model

- train model
- evaluate model
- test model

word language model

- target: to predict next word
  - input: never too old to learn
  - output: too old to learn English
- model: LSTM
- loss: cross_entropy

Exercise

task: sentiment analysis

- dataset: glue-sst2
- model: RNN or any other you interested in.

Thanks

THUNLP