CS 6770 Natural Language Processing

Recurrent Neural Networks Language Models

Yangfeng Ji

Information and Language Processing Lab
Department of Computer Science
University of Virginia

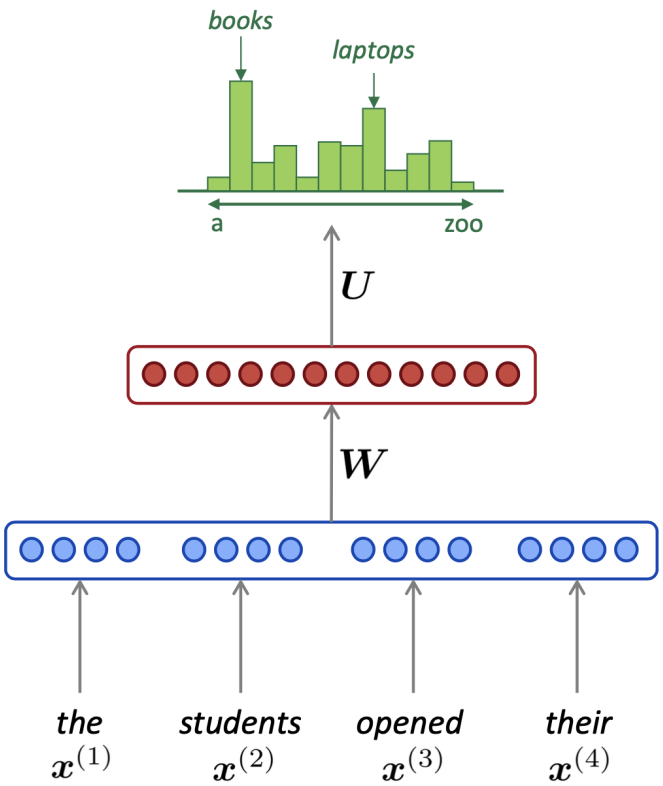
UVA ILP
UNIVERSITY OF VIRGINIA
INFORMATION AND LANGUAGE PROCESSING LAB

Overview

1. Neural Network Language Models
2. Recurrent Neural Networks
3. Computation Graph
4. RNN Language Modeling
5. Challenge of Training RNNs

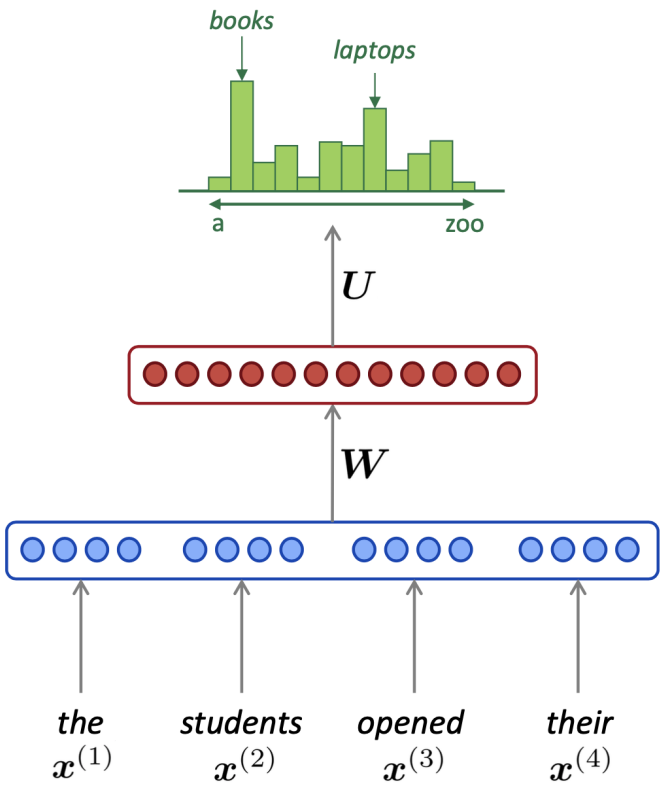
Neural Network Language Models

A Neural Language Model with Fixed Window Size



- Word indices: x_1, x_2, x_3, x_4

A Neural Language Model with Fixed Window Size

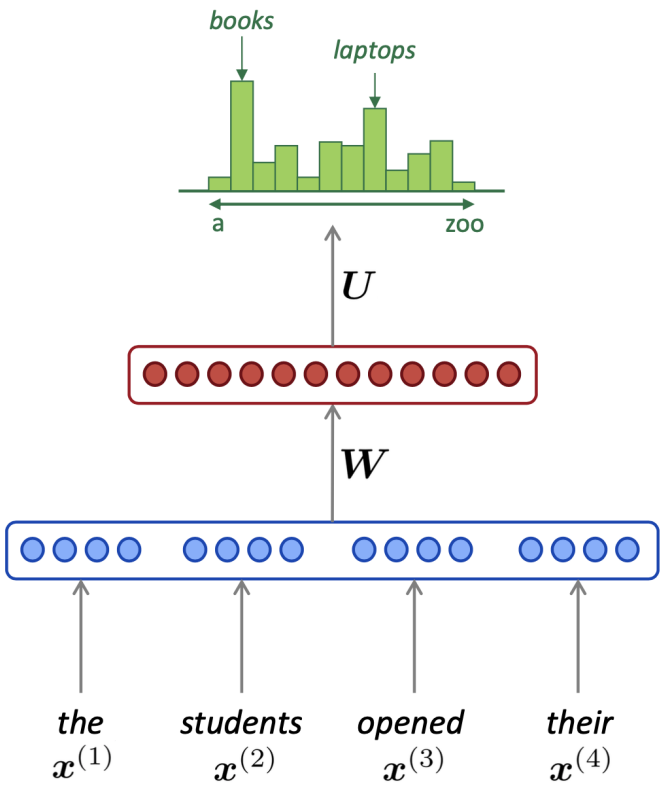


- Concatenated word embeddings

$$v = [v_{x_1}, v_{x_2}, v_{x_3}, v_{x_4}] \qquad (3)$$

- Word indices: x_1, x_2, x_3, x_4

A Neural Language Model with Fixed Window Size



- Hidden layer: $f(\cdot)$ could be any nonlinear activation function

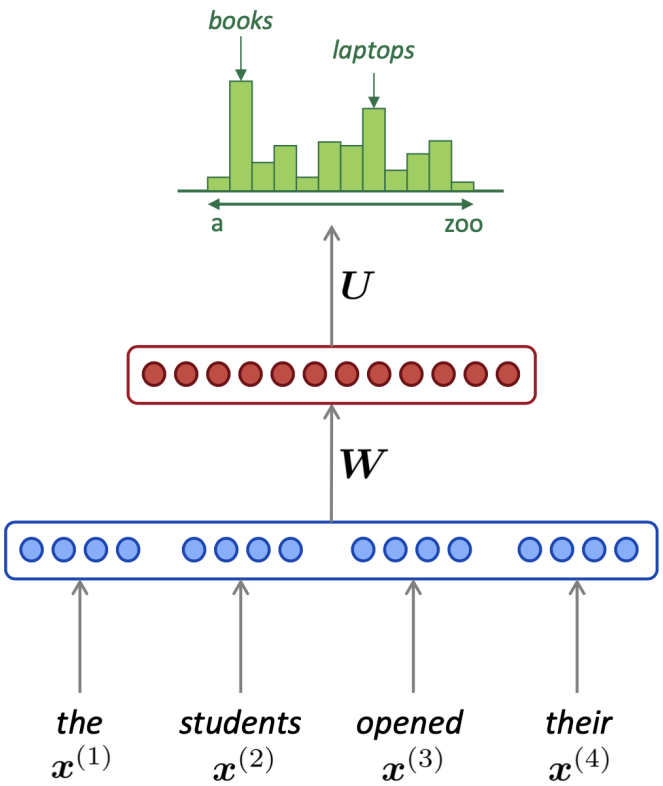
$$h = f(Wv + b_1) \tag{2}$$

- Concatenated word embeddings

$$v = [v_{x_1}, v_{x_2}, v_{x_3}, v_{x_4}] \tag{3}$$

- Word indices: x_1, x_2, x_3, x_4

A Neural Language Model with Fixed Window Size



- Output distribution

$$P(X_5 \mid \boldsymbol{X}_{1:4}) = \text{softmax}(\boldsymbol{U}\boldsymbol{h}+\boldsymbol{b}_2) \in \mathbb{R}^{|\mathcal{V}|} \quad (1)$$

- Hidden layer: $f(\cdot)$ could be any nonlinear activation function

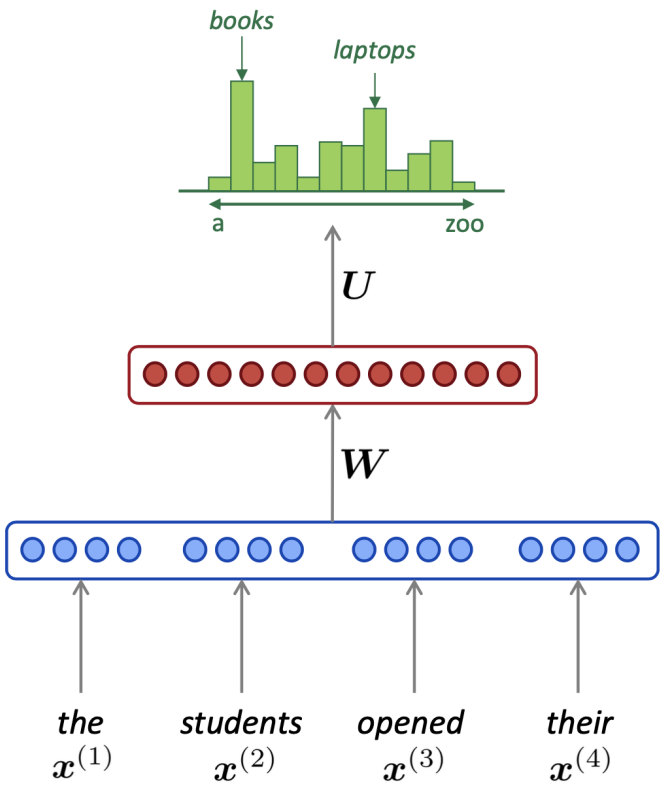
$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{b}_1) \quad (2)$$

- Concatenated word embeddings

$$\boldsymbol{v} = [\boldsymbol{v}_{x_1}, \boldsymbol{v}_{x_2}, \boldsymbol{v}_{x_3}, \boldsymbol{v}_{x_4}] \quad (3)$$

- Word indices: x_1, x_2, x_3, x_4

A Neural Language Model with Fixed Window Size



- Output distribution

$$P(X_5 \mid \boldsymbol{X}_{1:4}) = \text{softmax}(\boldsymbol{U}\boldsymbol{h}+\boldsymbol{b}_2) \in \mathbb{R}^{|\mathcal{V}|} \quad (1)$$

- Hidden layer: $f(\cdot)$ could be any nonlinear activation function

$$\boldsymbol{h} = f(\boldsymbol{W}\boldsymbol{v} + \boldsymbol{b}_1) \quad (2)$$

- Concatenated word embeddings

$$\boldsymbol{v} = [\boldsymbol{v}_{x_1}, \boldsymbol{v}_{x_2}, \boldsymbol{v}_{x_3}, \boldsymbol{v}_{x_4}] \quad (3)$$

- Word indices: x_1, x_2, x_3, x_4

This is the very first neural neural language model [Bengio et al., 2001], which has a similar network architecture as the one discussed text classification.

A Neural Probabilistic Language Model

The first paragraph of the paper *A Neural Probabilistic Language Model* [Bengio et al., 2001]

1 Introduction

A fundamental problem that makes language modeling and other learning problems difficult is the *curse of dimensionality*. It is particularly obvious in the case when one wants to model the joint distribution between many discrete random variables (such as words in a sentence, or discrete attributes in a data-mining task). For example, if one wants to model the joint distribution of 10 consecutive words in a natural language with a vocabulary V of size 100,000, there are potentially $100\,000^{10} - 1 = 10^{50} - 1$ free parameters.

[1] For more precise description, please refer to [Shalev-Shwartz and Ben-David, 2014]

A Neural Probabilistic Language Model

The first paragraph of the paper *A Neural Probabilistic Language Model* [Bengio et al., 2001]

1 Introduction

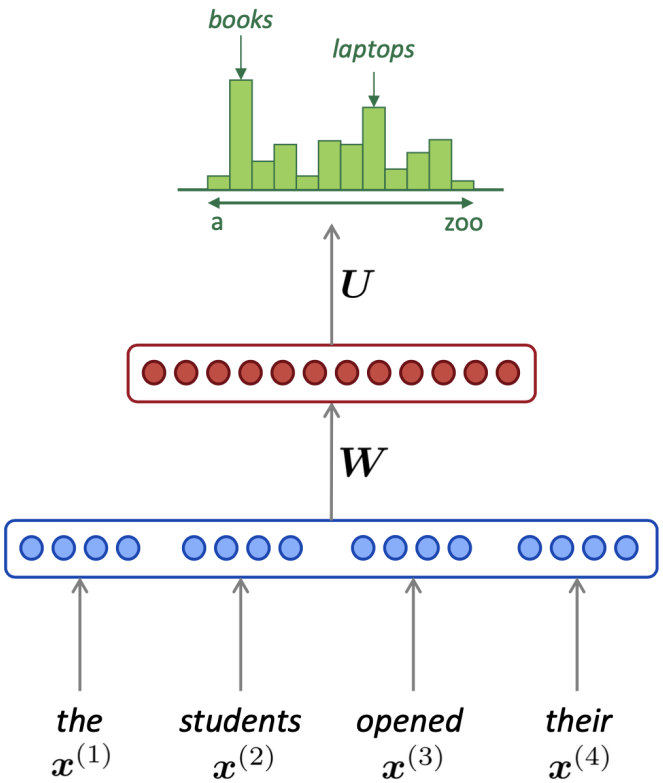
A fundamental problem that makes language modeling and other learning problems difficult is the *curse of dimensionality*. It is particularly obvious in the case when one wants to model the joint distribution between many discrete random variables (such as words in a sentence, or discrete attributes in a data-mining task). For example, if one wants to model the joint distribution of 10 consecutive words in a natural language with a vocabulary V of size 100,000, there are potentially $100\,000^{10} - 1 = 10^{50} - 1$ free parameters.

Curse of dimensionality

The sample complexity is an exponential function of the dimensionality of data[1]

[1] For more precise description, please refer to [Shalev-Shwartz and Ben-David, 2014]

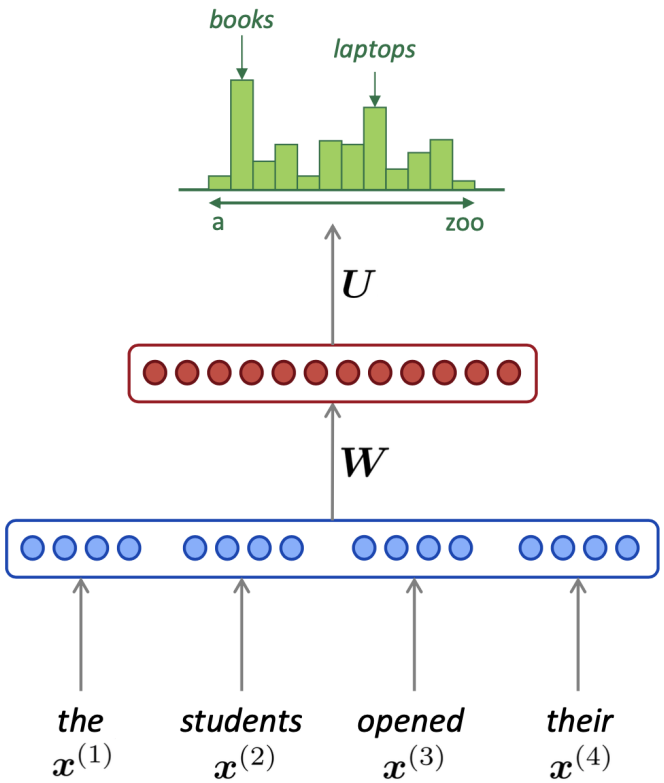
A Neural Language Model with Fixed Window Size



Improvement over n-gram language models

- Less parameters (with large n's)
- No sparsity problem
- No smoothing is needed

A Neural Language Model with Fixed Window Size



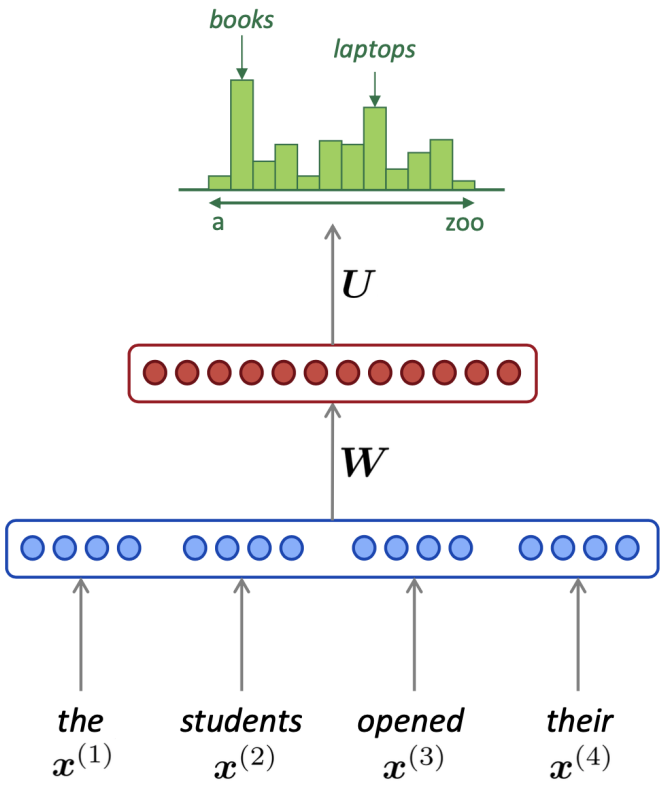
Improvement over n-gram language models

- Less parameters (with large n's)
- No sparsity problem
- No smoothing is needed

Remaining issues

- Fixed window size – similar to n-gram models
- No explicit modeling of word order beyond the window size
- Same word will be computed k times along the sliding window, where k is the window size

A Neural Language Model with Fixed Window Size



Improvement over n-gram language models

- Less parameters (with large n's)
- No sparsity problem
- No smoothing is needed

Remaining issues

- Fixed window size – similar to n-gram models
- No explicit modeling of word order beyond the window size
- Same word will be computed k times along the sliding window, where k is the window size

We need a new neural network architecture that can read words continuously along predictions

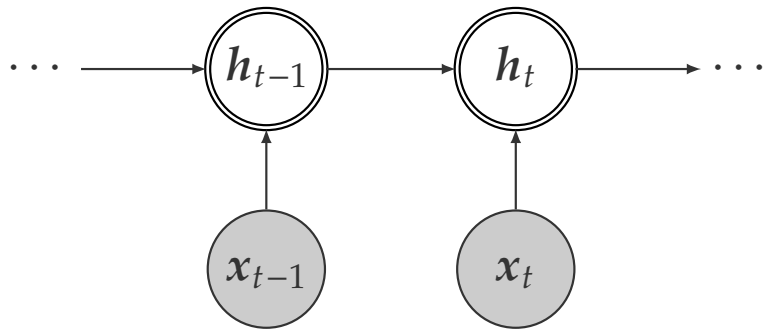
Recurrent Neural Networks

Recurrent Neural Networks (RNNs)

A simple RNN is defined by the following recursive function

$$h_t = f(x_t, h_{t-1}) \tag{4}$$

and depicted as



where

- h_{t-1}: hidden state at time step $t-1$
- x_t: input at time step t
- h_t: hidden state at time step t

A Simple Transition Function

In the simplest case, the transition function f is defined with an element-wise Sigmoid function and a linear transformation of $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = \sigma(\mathbf{W}_h \boldsymbol{h}_{t-1} + \mathbf{W}_i \boldsymbol{x}_t + \boldsymbol{b}) \tag{5}$$

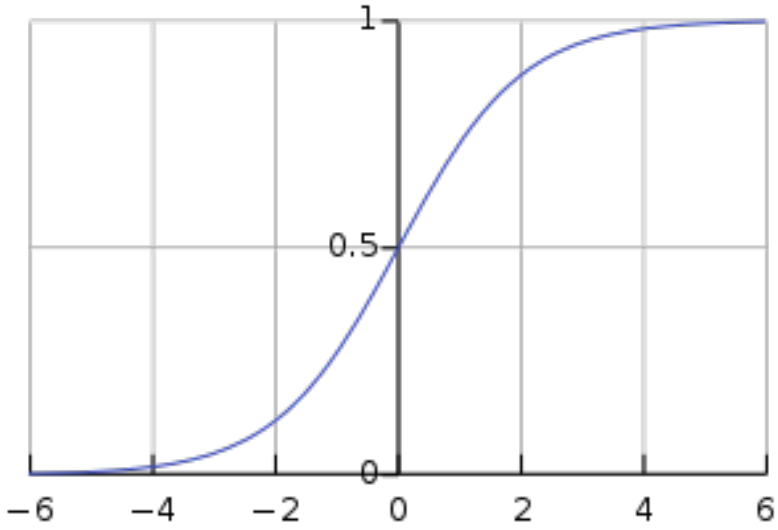
where

- $\boldsymbol{x}_t$: input word embedding
- $\boldsymbol{h}_{t-1}$: hidden statement from previous time step
- $\mathbf{W}_h$: parameter matrix for hidden states
- $\mathbf{W}_i$: parameter matrix for inputs
- $\boldsymbol{b}$: bias term (also a parameter)

Sigmoid Function

A Sigmoid function with one-dimensional input $x \in (-\infty, \infty)$

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



The potential numeric issue caused by the Sigmoid function

- $\sigma(x) \to 1$ with $x \gg 6$
- $\sigma(x) \to 0, x \ll -6$

The output of the Sigmoid function will approximate a constant, when the input value is beyond certain ranges

Unfolding RNNs

We can unfold this recursive definition of a RNN

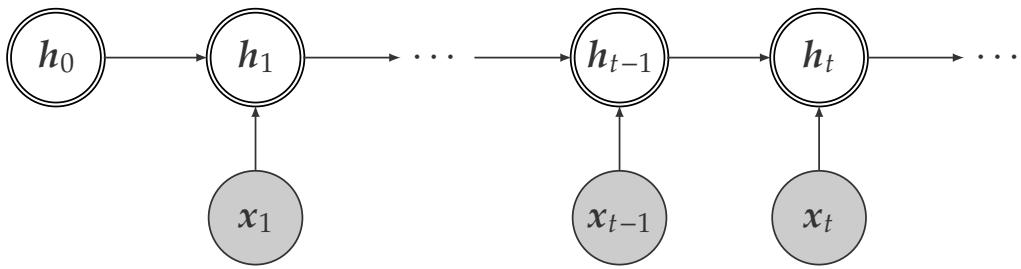
$$\boldsymbol{h}_t = \boldsymbol{f}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) \tag{6}$$

Unfolding RNNs

We can unfold this recursive definition of a RNN

$$\boldsymbol{h}_t = f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) \tag{6}$$

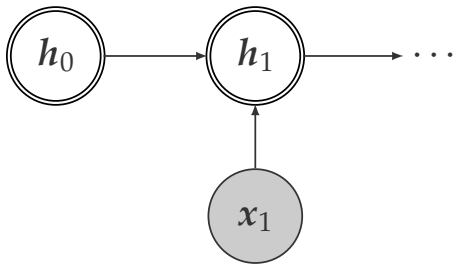
as



$$\begin{aligned}
\boldsymbol{h}_t &= f(\boldsymbol{x}_t, f(\boldsymbol{x}_{t-1}, \boldsymbol{h}_{t-2})) \\
&= f(\boldsymbol{x}_t, f(\boldsymbol{x}_{t-1}, f(\boldsymbol{x}_{t-2}, \boldsymbol{h}_{t-3}))) \\
&= \cdots \\
&= f(\boldsymbol{x}_t, f(\boldsymbol{x}_{t-1}, f(\boldsymbol{x}_{t-2}, \cdots f(\boldsymbol{x}_1, \boldsymbol{h}_0) \cdots))) \qquad (7)
\end{aligned}$$

Base Condition

Base condition defines the starting point of the recursive computation



$$h_t = f(x_t, f(x_{t-1}, f(x_{t-2}, \cdots f(x_1, h_0) \cdots))) \tag{8}$$

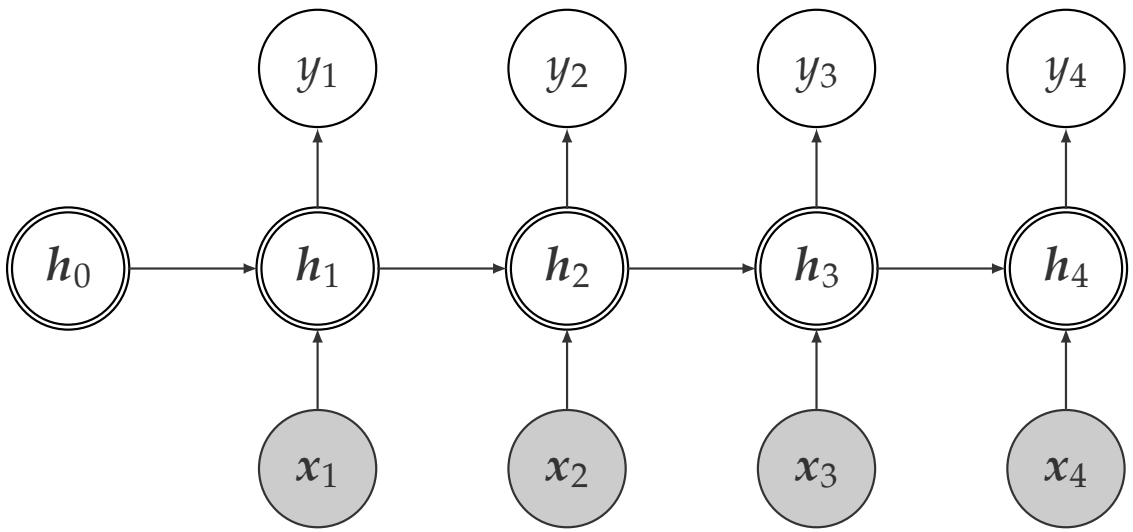
- h_0: zero vector or parameter
- x_1: input at time $t = 1$

RNN for Sequential Prediction

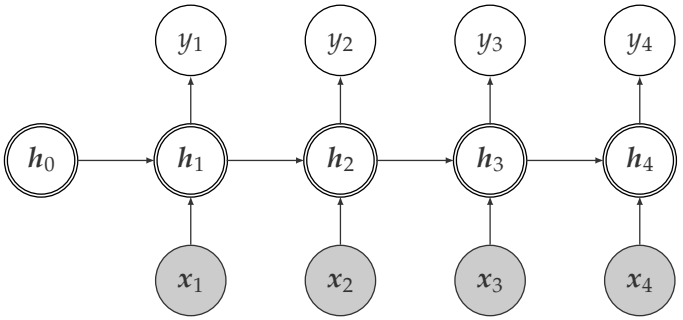
In general, RNNs can be used for any sequential modeling tasks

$$\boldsymbol{h}_t = \boldsymbol{f}(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) \tag{9}$$

$$P(Y_t; \boldsymbol{h}_t) = \text{softmax}(\boldsymbol{W}_o \boldsymbol{h}_t + \boldsymbol{b}_o) \tag{10}$$



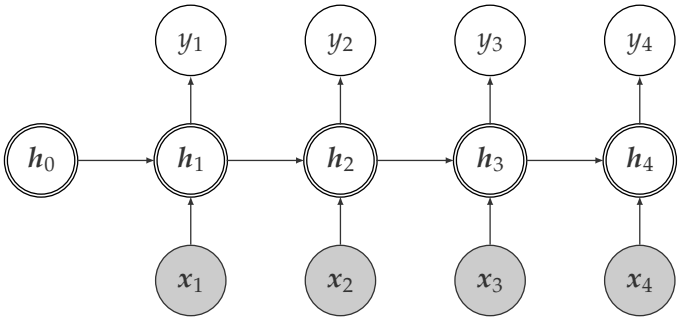
Sequential Modeling as Classification



- Prediction at each time step t

$$\hat{y}_t = \underset{y}{\operatorname{argmax}}\, P(Y_t = y; \boldsymbol{h}_t) \tag{11}$$

Sequential Modeling as Classification



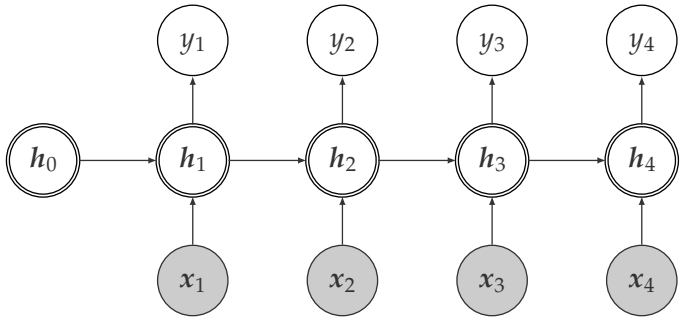
- Prediction at each time step t

$$\hat{y}_t = \underset{y}{\operatorname{argmax}}\, P(Y_t = y; \boldsymbol{h}_t) \tag{11}$$

- Loss at single time step t

$$L_t(y_t, \hat{y}_t) = -\log P(y_t; \boldsymbol{h}_t) \tag{12}$$

Sequential Modeling as Classification



- Prediction at each time step t

$$\hat{y}_t = \underset{y}{\operatorname{argmax}}\, P(Y_t = y; \boldsymbol{h}_t) \tag{11}$$

- Loss at single time step t

$$L_t(y_t, \hat{y}_t) = -\log P(y_t; \boldsymbol{h}_t) \tag{12}$$

- The total loss

$$\ell = \sum_{t=1}^{T} L_t(y_t, \hat{y}_t) \tag{13}$$

Computation Graph

Forward Operations

For simplicity, consider the example of a two-layer neural network

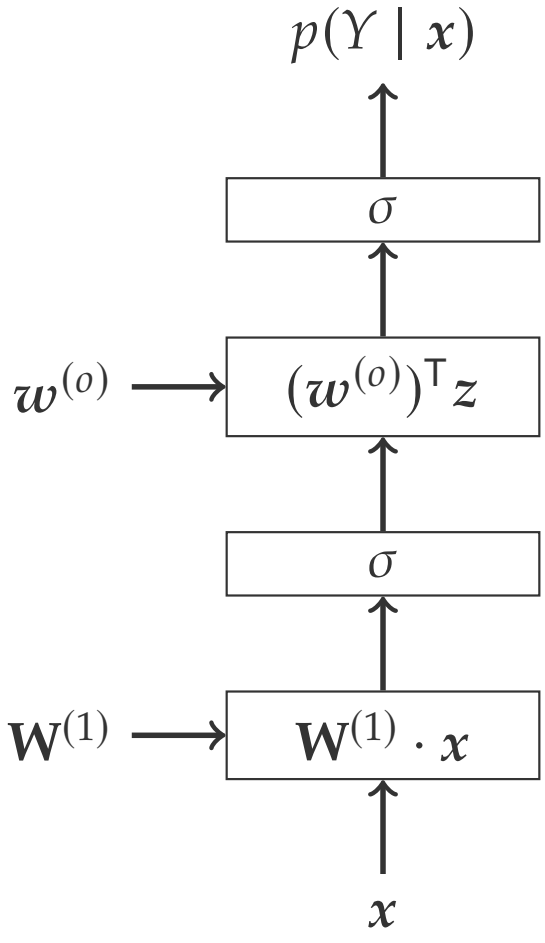
$$P(Y = +1 \mid \boldsymbol{x}) = \sigma\left((\boldsymbol{w}^{(o)})^{\mathsf{T}}\sigma(\mathbf{W}^{(1)}\boldsymbol{x})\right) \tag{14}$$

A neural network is a composition of some basic functions and operations. For example

- the Sigmoid function $\sigma(\cdot)$
- matrix transpose $(\boldsymbol{w}^{(o)})^{\mathsf{T}}$
- matrix-vector multiplication $\mathbf{W}^{(1)}\boldsymbol{x}$

Forward Graph

The computation graph of the two-layer neural network[2]



[2]For simplicity, the *transpose* operation blocks are ignored from the graph

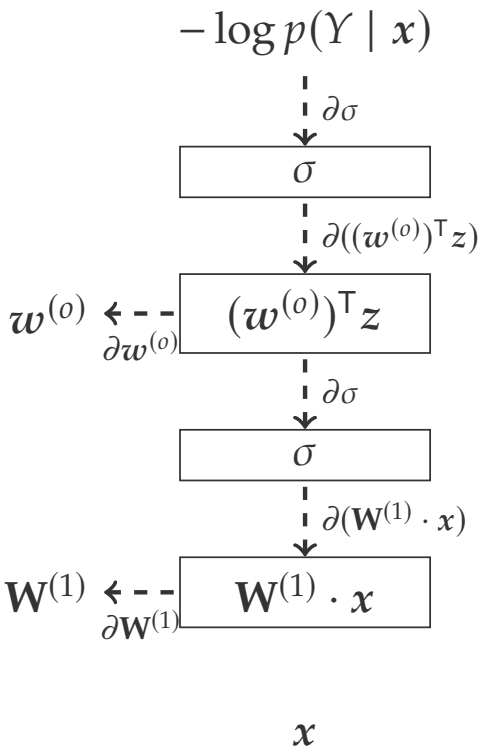
Backward Operations

Similarly, the gradient of neural network parameters are computed with a series of backward operations associated with the derivative of some basic function. For example

- $\frac{\mathrm{d}bx}{\mathrm{d}x} = b$
- $\frac{\mathrm{d}\log(x)}{\mathrm{d}x} = \frac{1}{x}$
- $\frac{\mathrm{d}\sigma(x)}{\mathrm{d}x} = \sigma(x)(1-\sigma(x))$
- $\frac{\mathrm{d}\boldsymbol{a}^{\mathsf{T}}\boldsymbol{x}}{\mathrm{d}\boldsymbol{x}} = \boldsymbol{a}$
- $\frac{\mathrm{d}\mathbf{W}\boldsymbol{x}}{\mathrm{d}\mathbf{W}} = \begin{bmatrix} \boldsymbol{x}^{\mathsf{T}} \\ \vdots \\ \boldsymbol{x}^{\mathsf{T}} \end{bmatrix}$

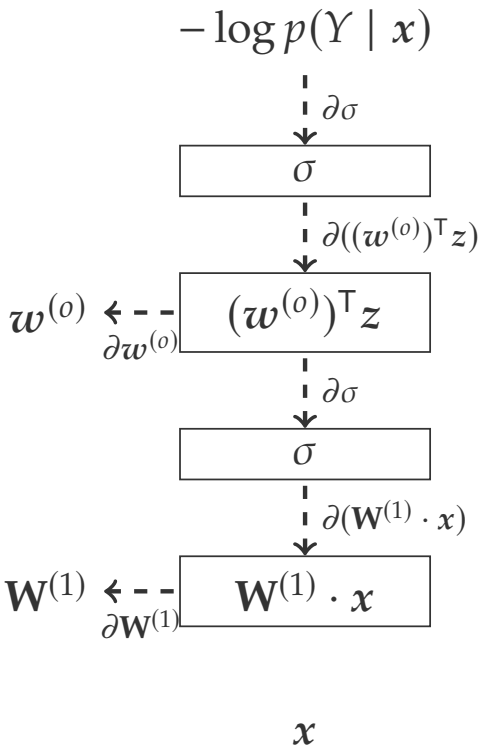
Backward Graph

- The gradient of each operations can be computed individually based on the input and output
- With the chain rule, gradient of the loss function with respect to any parameter is a sequence of multiplications of individual gradients



Backward Graph

- The gradient of each operations can be computed individually based on the input and output
- With the chain rule, gradient of the loss function with respect to any parameter is a sequence of multiplications of individual gradients



For example,

$$\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{w}^{(o)}} = -\frac{\partial \log \sigma\left(\cdot\right)}{\partial \sigma\left(\cdot\right)} \cdot \frac{\partial \sigma\left((\boldsymbol{w}^{(o)})^\top \sigma(\mathbf{W}^{(1)}\boldsymbol{x})\right)}{\partial (\boldsymbol{w}^{(o)})^\top \sigma(\mathbf{W}^{(1)}\boldsymbol{x})} \cdot \frac{\partial (\boldsymbol{w}^{(o)})^\top \sigma(\mathbf{W}^{(1)}\boldsymbol{x})}{\partial \boldsymbol{w}^{(o)}}$$

Example: MiniTorch

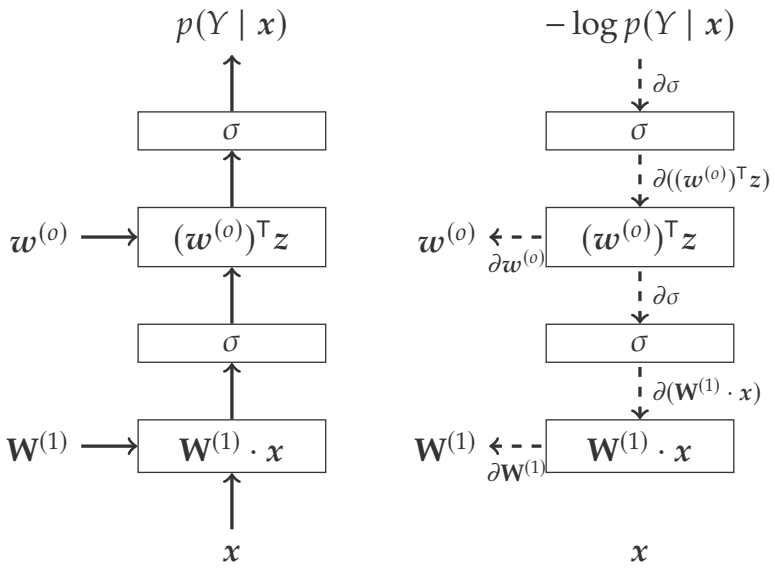
A screenshot from the MiniTorch library[3]

```python
class Add(Function):
    @staticmethod
    def forward(ctx, t1, t2):
        return add_zip(t1, t2)

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output, grad_output

class Mul(Function):
    @staticmethod
    def forward(ctx, a, b):
        ctx.save_for_backward(a, b)
        return mul_zip(a, b)

    @staticmethod
    def backward(ctx, grad_output):
        a, b = ctx.saved_values
        return mul_zip(b, grad_output), mul_zip(a, grad_output)

class Sigmoid(Function):
    @staticmethod
    def forward(ctx, a):
        out = sigmoid_map(a)
        ctx.save_for_backward(out)
        return out

    @staticmethod
    def backward(ctx, grad_output):
        sigma = ctx.saved_values
        return sigma * (-sigma + 1.0) * grad_output
```

[3] A teaching library of PyTorch developed by Sasha Rush and Ge Gao

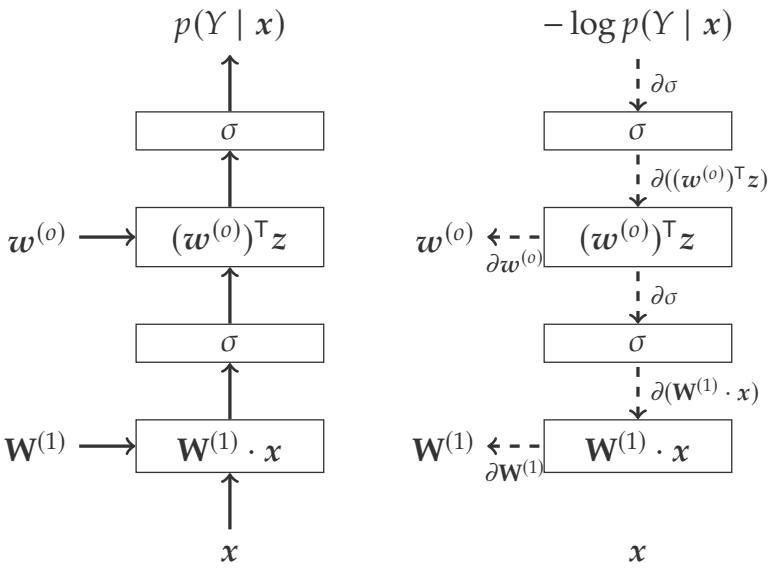
Computation Graph

Perform the forward/backward step with a graph of basic operations (e.g., PyTorch, Tensorflow)



Computation Graph

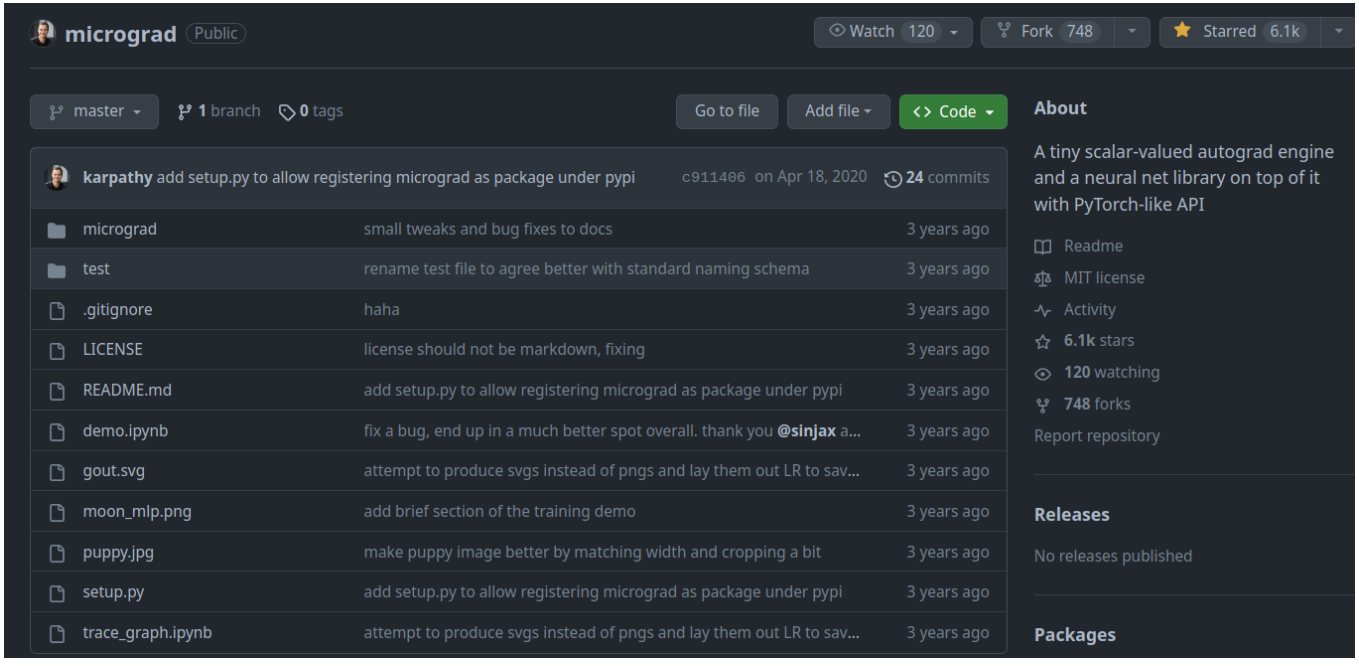
Perform the forward/backward step with a graph of basic operations (e.g., PyTorch, Tensorflow)



- Modular implementation: implement each module with its forward/backward operations together
- Automatic differentiation: automatically run with the backward step

Example: Micrograd

MICROGRAD is an extremely simple example to illustrate the idea of using autograd to implementing the backpropagation algorithm.



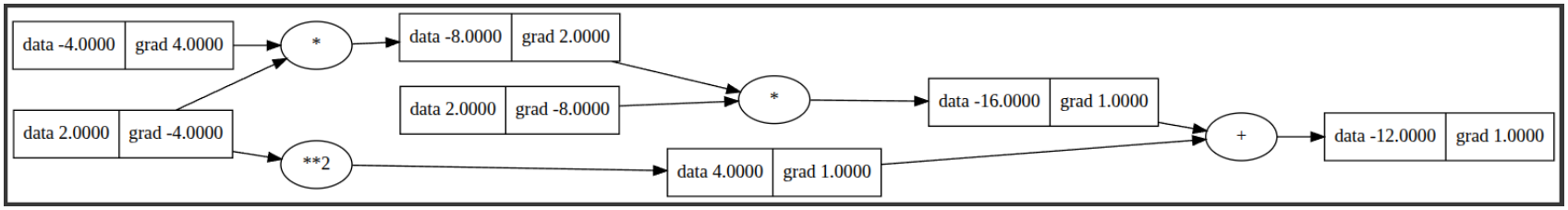
Demo

Example: Gradient Computation

Function

$$f(a,b) = 2ab + b^2$$

The computation graph on forward and backward operations



RNN Language Modeling

Language Models

A language model defines the probability of x_t given $\boldsymbol{x} = (x_1, x_2, \ldots, x_{t-1})$ as

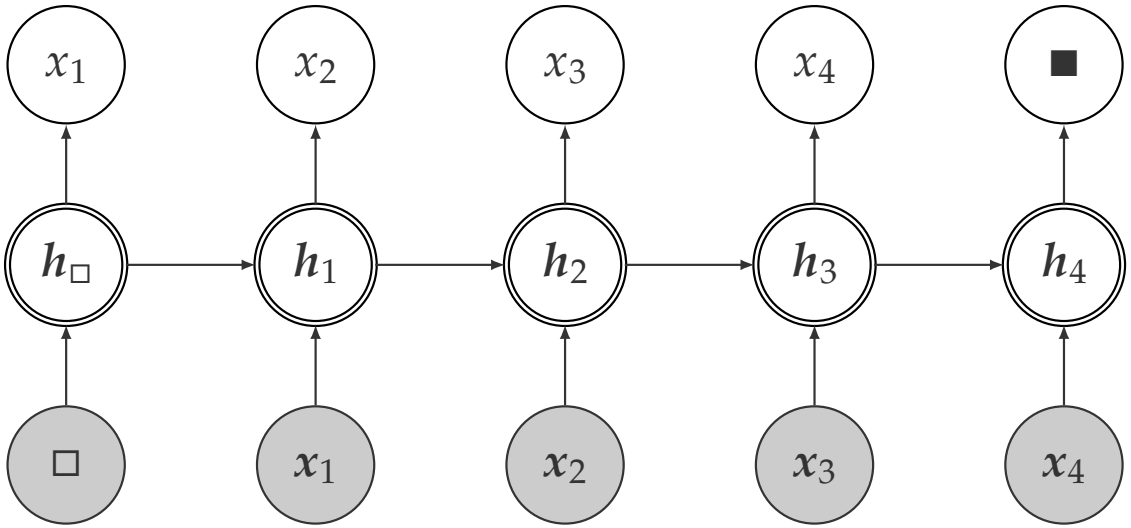
$$P(x_t \mid x_1, \ldots, x_{t-1}) \tag{15}$$

and the joint probability as

$$\begin{aligned} P(\boldsymbol{x}_{1:T}) \quad = \quad & P(x_1) \cdot P(x_2 \mid x_1) \\ & \ldots \\ & \cdot P(x_T \mid x_1, x_2, \ldots, x_{T-1}) \end{aligned}$$

Language Modeling with RNNs

Using RNNs for language modeling

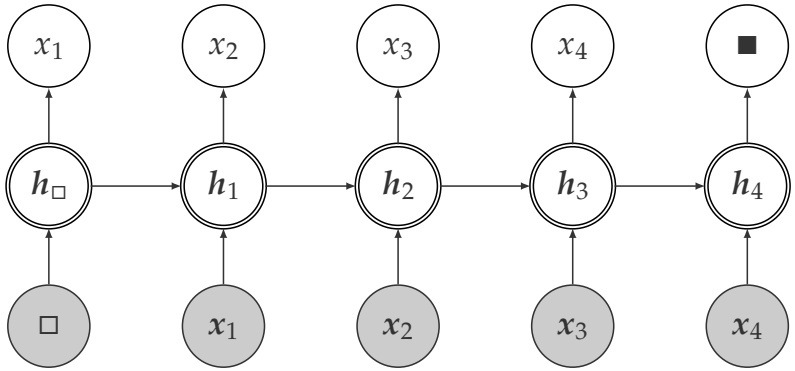


with two special tokens

$$\{\square, x_1, \ldots, x_T, \blacksquare\}$$

RNN Language Models

For a given sentence $\{x_1, \ldots, x_t\}$, the input at time t is word embedding $\boldsymbol{x}_t$



The probability distribution of next word X_t

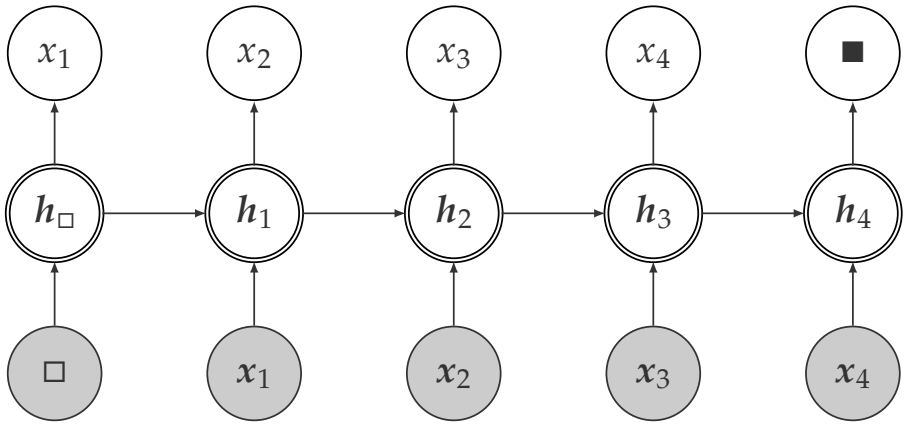
$$P(X_t = x \mid \boldsymbol{x}_{1:t-1}) = \frac{\exp(\boldsymbol{w}_{o,x}^\top \boldsymbol{h}_{t-1})}{\sum_{x' \in \mathcal{V}} \exp(\boldsymbol{w}_{o,x'}^\top \boldsymbol{h}_{t-1})} \tag{16}$$

where

- $\boldsymbol{w}_{o,x}$ is the output weight vector (parameter) associated with word x
- $\mathcal{V}$ is the word vocabulary

Special Cases

Similar to statistical language modeling, there are also two special cases that we need to consider



$$\{\square, x_1, \ldots, x_T, \blacksquare\}$$

The corresponding prediction functions are defined as

- At time $t = 1$

$$P(X_1 = x) \propto \exp(\boldsymbol{w}_{o,x}^\top \boldsymbol{h}_\square) \tag{17}$$

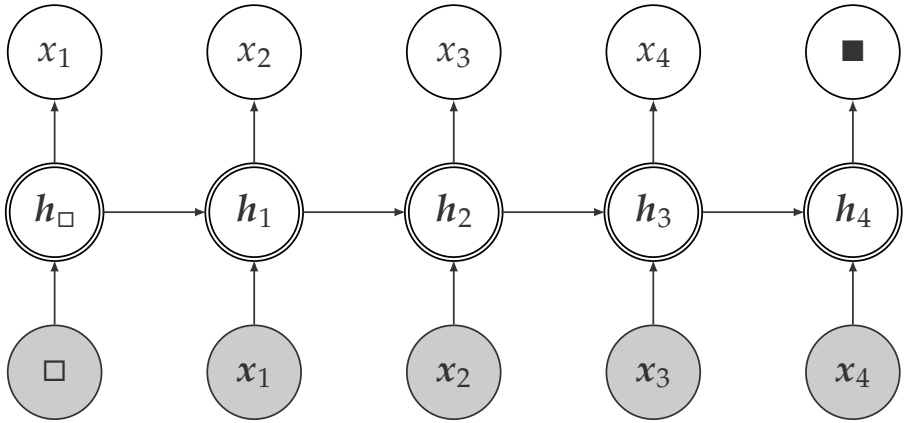
- At time $t = T$

$$P(X_T = \blacksquare \mid \boldsymbol{x}_{1:T-1}) \propto \exp(\boldsymbol{w}_{o,x}^\top \boldsymbol{h}_{T-1}) \tag{18}$$

Challenge of Training RNNs

Objective

The training objective for each timestep is to predict the next token in the text

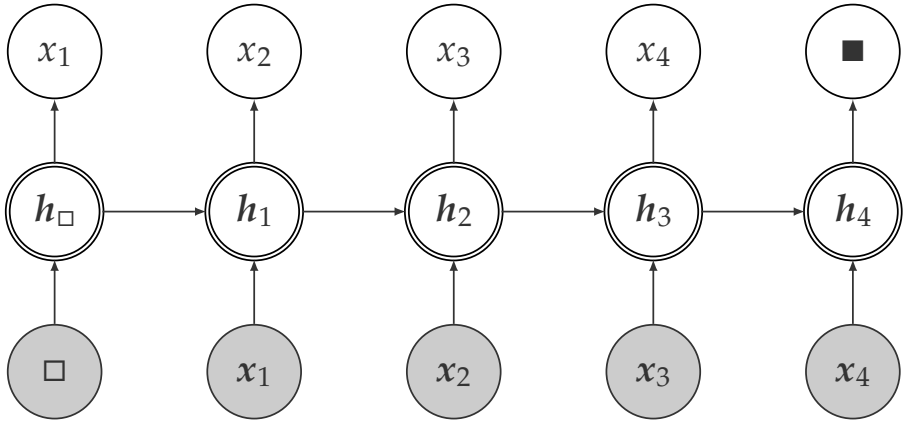


- Prediction at step t, $P(X_t = x \mid \boldsymbol{x}_{1:t-1}) = \frac{\exp(\boldsymbol{w}_{o,x}^\top \boldsymbol{h}_{t-1})}{\sum_{x' \in \mathcal{V}} \exp(\boldsymbol{w}_{o,x'}^\top \boldsymbol{h}_{t-1})}$
- Loss at step t, $L_t = -\log P(X_t = x \mid \boldsymbol{x}_{1:t-1})$

Gradients

Let $\boldsymbol{\theta}$ denote all model parameters

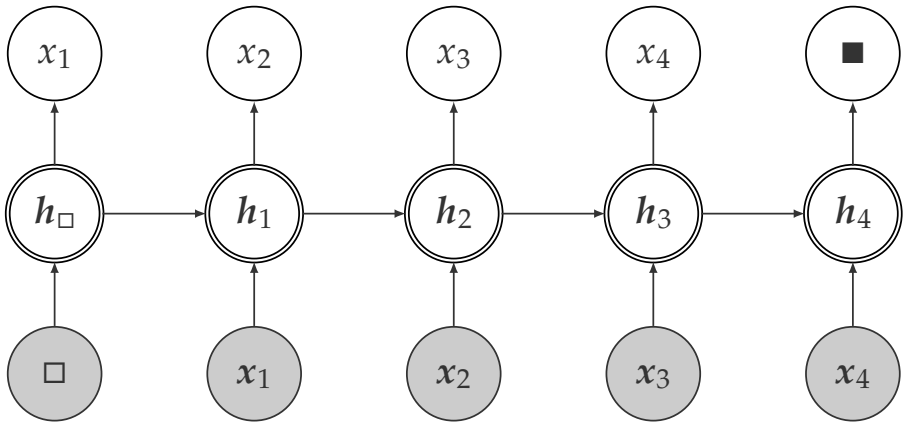
$$\frac{\partial \ell}{\partial \boldsymbol{\theta}} = \sum_{t=1}^{T} \frac{\partial L_t}{\partial \boldsymbol{\theta}} \tag{19}$$



Backpropagation Through Time [Rumelhart et al., 1985, BPTT]

Model Parameters

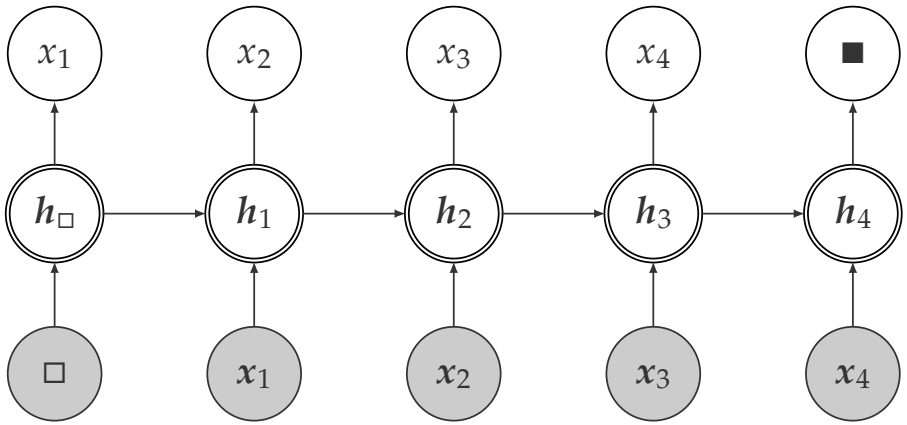
Before computing the gradient of each L_t with respect to model parameters, let us count how many parameters that we need consider



- Output parameter matrix $\boldsymbol{W}_o = (\boldsymbol{w}_{o,1}, \ldots, \boldsymbol{w}_{o,V})$

Model Parameters

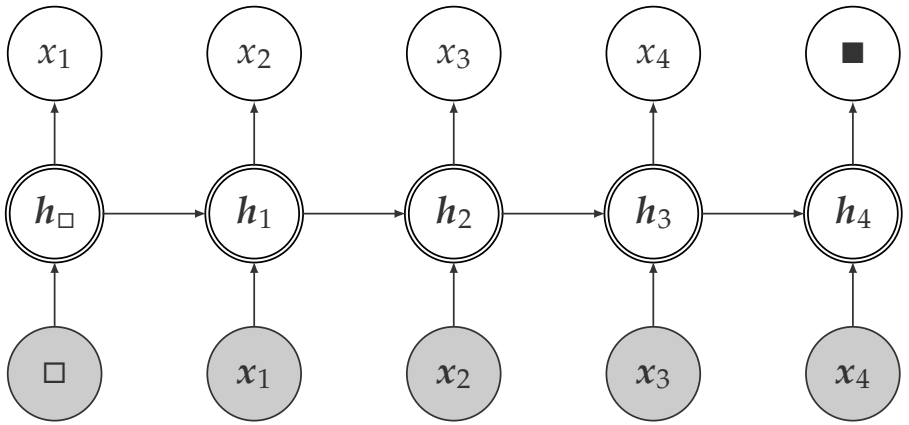
Before computing the gradient of each L_t with respect to model parameters, let us count how many parameters that we need consider



- Output parameter matrix $\boldsymbol{W}_o = (\boldsymbol{w}_{o,1}, \ldots, \boldsymbol{w}_{o,V})$
- Input word embedding matrix $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_V)$

Model Parameters

Before computing the gradient of each L_t with respect to model parameters, let us count how many parameters that we need consider



- Output parameter matrix $\boldsymbol{W}_o = (\boldsymbol{w}_{o,1}, \ldots, \boldsymbol{w}_{o,V})$
- Input word embedding matrix $\boldsymbol{X} = (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_V)$
- Neural network parameters $\boldsymbol{W}_h, \boldsymbol{W}_i, \boldsymbol{b}$

Backpropagation Through Time

Take time step t as an example, we can take a look the gradient computation of some specific parameters

- Output model parameter $\frac{\partial L_t}{\partial \boldsymbol{w}_{o,\cdot}}$

Backpropagation Through Time

Take time step t as an example, we can take a look the gradient computation of some specific parameters

- Output model parameter $\frac{\partial L_t}{\partial \boldsymbol{w}_{o,\cdot}}$
- Neural network parameters, for example $\boldsymbol{W}_h$

$$\frac{\partial L_t}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \left\{ \frac{\partial L_t}{\partial \boldsymbol{h}_t} \cdot \left(\prod_{j=i}^{t-1} \frac{\partial \boldsymbol{h}_{j+1}}{\partial \boldsymbol{h}_j} \right) \cdot \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{W}_h} \right\} \tag{20}$$

Similar patterns for the other two neural network parameters $\boldsymbol{W}_i$ and $\boldsymbol{b}$

Backpropagation Through Time

Take time step t as an example, we can take a look the gradient computation of some specific parameters

- Output model parameter $\frac{\partial L_t}{\partial \boldsymbol{w}_{o,\cdot}}$
- Neural network parameters, for example $\boldsymbol{W}_h$

$$\frac{\partial L_t}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \left\{ \frac{\partial L_t}{\partial \boldsymbol{h}_t} \cdot \left(\prod_{j=i}^{t-1} \frac{\partial \boldsymbol{h}_{j+1}}{\partial \boldsymbol{h}_j} \right) \cdot \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{W}_h} \right\} \tag{20}$$

  Similar patterns for the other two neural network parameters $\boldsymbol{W}_i$ and $\boldsymbol{b}$

- Word embedding $\frac{\partial L_t}{\partial \boldsymbol{x}_{t'}}$
  - E.g., word embedding $\boldsymbol{x}_{t'}$ is the input of $\boldsymbol{h}_t$ if $t' \leq t$, so ...

Challenges

For each timestep, we need to compute the gradient using the chain rule:

$$\frac{\partial L_t}{\partial \boldsymbol{W}_h} = \sum_{i=1}^{t} \{ \frac{\partial L_t}{\partial \boldsymbol{h}_t} \cdot (\prod_{j=i}^{t-1} \frac{\partial \boldsymbol{h}_{j+1}}{\partial \boldsymbol{h}_j}) \cdot \frac{\partial \boldsymbol{h}_i}{\partial \boldsymbol{W}_h} \} \tag{21}$$

The chain rule of gradient will cause two potential problems in training RNNs

- vanishing gradients: $\frac{\partial L_t}{\partial \boldsymbol{\theta}} \to 0$
- exploding gradients: $\frac{\partial L_t}{\partial \boldsymbol{\theta}} \geq M$

[Pascanu et al., 2013]

Exploding Gradients

Solution: **norm clipping** [Pascanu et al., 2013].

Consider the gradient $g = \frac{\partial \ell}{\partial \theta}$,

$$\hat{g} \leftarrow \tau \cdot \frac{g}{\|g\|} \tag{22}$$

when $\|g\| > \tau$.

- Usually, $\tau = 3$ or 5 in practice.
- Smaller gradient will cause slower learning progress

Vanishing Gradients

Solution:

- initialize parameters carefully
- replace hidden state transition function $\sigma(\cdot)$ with other options

$$f(\boldsymbol{x}_t, \boldsymbol{h}_{t-1}) = \sigma(\mathbf{W}_h \boldsymbol{h}_{t-1} + \mathbf{W}_i \boldsymbol{x}_t + \boldsymbol{b}) \tag{23}$$

  - LSTM [Hochreiter and Schmidhuber, 1997]
  - GRU [Cho et al., 2014]

Long Short-Term Memory

From the first page of the original paper proposing LSTM [Hochreiter and Schmidhuber, 1997]

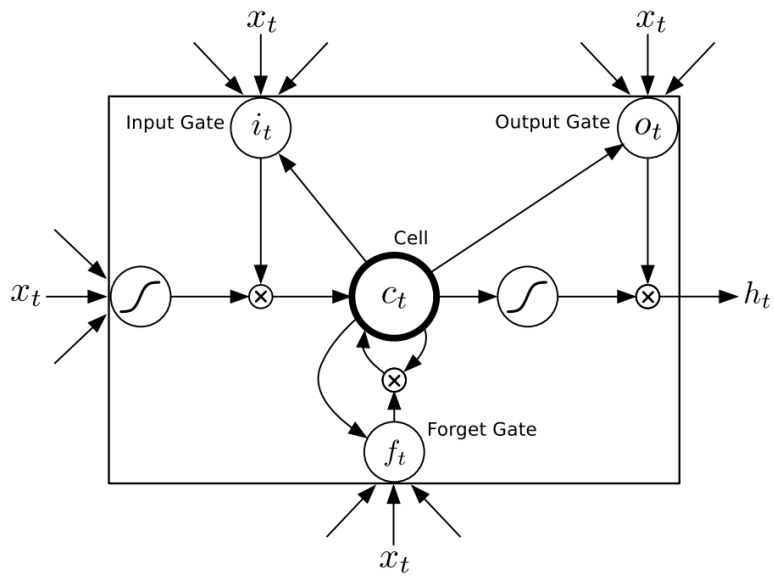
The problem. With conventional "Back-Propagation Through Time" (BPTT, e.g., Williams and Zipser 1992, Werbos 1988) or "Real-Time Recurrent Learning" (RTRL, e.g., Robinson and Fallside 1987), error signals "flowing backwards in time" tend to either (1) blow up or (2) vanish: the temporal evolution of the backpropagated error exponentially depends on the size of the weights (Hochreiter 1991). Case (1) may lead to oscillating weights, while in case (2) learning to bridge long time lags takes a prohibitive amount of time, or does not work at all (see section 3).

The remedy. This paper presents *"Long Short-Term Memory"* (LSTM), a novel recurrent network architecture in conjunction with an appropriate gradient-based learning algorithm. LSTM is designed to overcome these error back-flow problems. It can learn to bridge time intervals in excess of 1000 steps even in case of noisy, incompressible input sequences, without loss of short time lag capabilities. This is achieved by an efficient, gradient-based algorithm for an architecture

Long Short-Term Memory

Rather than directly taking input and hidden state as simple transition function, LSTM relies on three cates to control *how much* information it should take from input and hidden state before combining them together

$$
\begin{aligned}
\boldsymbol{i}_t &= \sigma(\mathbf{W}_{xi}\boldsymbol{x}_t + \mathbf{W}_{hi}\boldsymbol{h}_{t-1} + \mathbf{W}_{ci}\boldsymbol{c}_{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{f}_t &= \sigma(\mathbf{W}_{xf}\boldsymbol{x}_t + \mathbf{W}_{hf}\boldsymbol{h}_{t-1} + \mathbf{W}_{cf}\boldsymbol{c}_{t-1} + \boldsymbol{b}_f) \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \circ \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \circ \tanh(\mathbf{W}_{xc}\boldsymbol{x}_t + \mathbf{W}_{hc}\boldsymbol{h}_{t-1} + \boldsymbol{b}_c) \\
\boldsymbol{o}_t &= \sigma(\mathbf{W}_{xo}\boldsymbol{x}_t + \mathbf{W}_{ho}\boldsymbol{h}_{t-1} + \mathbf{W}_{co}\boldsymbol{c}_t + \boldsymbol{b}_o) \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \circ \tanh(\boldsymbol{c}_t)
\end{aligned}
$$



where $\circ$ is the element-wise multiplication, $\{\boldsymbol{W}.\}$ and $\{\boldsymbol{b}.\}$ are parameters. [Graves, 2013]

Reference

Bengio, Y., Ducharme, R., and Vincent, P. (2001).

A neural probabilistic language model.

In *NIPS*.

Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014).

On the properties of neural machine translation: Encoder-decoder approaches.

arXiv preprint arXiv:1409.1259.

Graves, A. (2013).

Generating sequences with recurrent neural networks.

arXiv preprint arXiv:1308.0850.

Hochreiter, S. and Schmidhuber, J. (1997).

Long short-term memory.

Neural computation, 9(8):1735–1780.

Pascanu, R., Mikolov, T., and Bengio, Y. (2013).

On the difficulty of training recurrent neural networks.

In *International Conference on Machine Learning*, pages 1310–1318.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985).

Learning internal representations by error propagation.

Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Shalev-Shwartz, S. and Ben-David, S. (2014).

Understanding machine learning: From theory to algorithms.

Cambridge university press.