Lecture 8:
Feed-forward Neural Nets

Instructor: Swabha Swayamdipta
USC CSCI 544 Applied NLP
Sep 19, Fall 2024

Some slides adapted from Dan Jurafsky and Chris Manning

Announcements + Logistics

Announcements + Logistics

- HW and late days: DO NOT email, your submission time will determine how many late days you used.
    - NO partial late days - 1 day = 24 hours!
- Projects Proposal - what to expect (also see the class website)
    - Student teams should submit a ~1-page proposal (follow *CL format)
    - The proposal should:
        - state and motivate the problem by providing a problem or task definition (preferably with example inputs and expected outputs),
        - situate the problem within related work (this might help you find sources of data for training a model for your task),
            - Related work: publications, start by looking in the ACL anthology (https://aclanthology.org/)
            - References do not count towards page limit, but please follow the correct format
        - state a hypothesis to be verified and how to verify it (evaluation framework), and
        - a brief description of the approach to be followed to verify the hypothesis (such as proposed models and baselines, and metrics).
    - We will need you to submit all your code as a final deliverable. PLAGIARISM will be strictly penalized

Example Project

USC Viterbi

Example Project

Project Proposal: Code to Pseudocode Using LLM

Wenda Gu
wendagu@usc.edu

Egor Cherkashin
cherkash@usc.edu

Sarah Chen
snchen@usc.edu

Example Project

Project Proposal: Code to Pseudocode Using LLM

Wenda Gu wendagu@usc.edu
Egor Cherkashin cherkash@usc.edu
Sarah Chen snchen@usc.edu

Abstract

This document is a project proposal for a Large Language Model trained to convert Python code to pseudocode and an explanation of the code function. It goes over the problem and its importance, related research, and our methodology.

USC Viterbi

Example Project

Project Proposal: Code to Pseudocode Using LLM

Wenda Gu
wendagu@usc.edu

Egor Cherkashin
cherkash@usc.edu

Sarah Chen
snchen@usc.edu

Abstract

This document is a project proposal for a Large Language Model trained to convert Python code to pseudocode and an explanation of the code function. It goes over the problem and its importance, related research, and our methodology.

Input	Output

`int main() {`
`  int n;`
`  cin >> n;`
`  vector<int> A;`
`  A.resize(n);`
`  for(int i=0; i<A.size(); i++) cin >> A[i]`
`  for(int i=0; i<A.size(); i++) {`
`      int min_i = i;`
`      for(int j=i+1; j<A.size(); j++) {`
`          if(A[min_i] > A[j]) { min_i = j; }`
`      swap(A[i], A[min_i]);`
`  for(int i=0; i<A.size(); i++) cout<<A[i]<<" ";`
	`in function main`
	` let n be integer`
	`  read n`
	` let A be a vector of integers`
	`   set size of A = n`
	`  read n elements into A`
	`    for all elements in A`
	`     set min_i to i`
	`      for j=i+1 to size of A exclusive`
	`         set min_i to j if A[min_i]>A[j]`
	`      swap A[i], A[min_i]`
	` print all elements in A`
	
	`Explanation: An implementation of a sorting algorithm that sorts user-inputted values in a vector in ascending order, then prints the sorted vector.`

Table 1: Example input and output for our LLM.

Example Project

Project Proposal: Code to Pseudocode Using LLM

Wenda Gu
wendagu@usc.edu

Egor Cherkashin
cherkash@usc.edu

Sarah Chen
snchen@usc.edu

Abstract

This document is a project proposal for a Large Language Model trained to convert Python code to pseudocode and an explanation of the code function. It goes over the problem and its importance, related research, and our methodology.

3 Related Work

Input	Output

<pre>int main() {
  int n;
  cin >> n;
  vector<int> A;
  A.resize(n);
  for(int i=0; i<A.size(); i++) cin >> A[i]
  for(int i=0; i<A.size(); i++) {
      int min_i = i;
      for(int j=i+1; j<A.size(); j++) {
          if(A[min_i] > A[j]) { min_i = j; }
      swap(A[i], A[min_i]);
  for(int i=0; i<A.size(); i++) cout<<A[i]<<" ";</pre>
	<pre>in function main
	 let n be integer
	  read n
	 let A be a vector of integers
	   set size of A = n
	  read n elements into A
	    for all elements in A
	     set min_i to i
	      for j=i+1 to size of A exclusive
	         set min_i to j if A[min_i]>A[j]
	      swap A[i], A[min_i]
	 print all elements in A
	
	Explanation: An implementation of a sorting algorithm
	that sorts user-inputted values in a vector in
	ascending order, then prints the sorted vector.</pre>

Table 1: Example input and output for our LLM.

Example Project

Project Proposal: Code to Pseudocode Using LLM

Wenda Gu wendagu@usc.edu
Egor Cherkashin cherkash@usc.edu
Sarah Chen snchen@usc.edu

Abstract

This document is a project proposal for a Large Language Model trained to convert Python code to pseudocode and an explanation of the code function. It goes over the problem and its importance, related research, and our methodology.

3 Related Work

4 Hypothesis

A specially trained language model for code to pseudocode translation and explanation will perform better than general purpose language models.

Input	Output

<pre>int main() {
  int n;
  cin >> n;
  vector<int> A;
  A.resize(n);
  for(int i=0; i<A.size(); i++) cin >> A[i]
  for(int i=0; i<A.size(); i++) {
      int min_i = i;
      for(int j=i+1; j<A.size(); j++) {
          if(A[min_i] > A[j]) { min_i = j; }
      swap(A[i], A[min_i]);
  for(int i=0; i<A.size(); i++) cout<<A[i]<<" ";</pre>
	<pre>in function main
	 let n be integer
	  read n
	 let A be a vector of integers
	   set size of A = n
	  read n elements into A
	    for all elements in A
	     set min_i to i
	      for j=i+1 to size of A exclusive
	         set min_i to j if A[min_i]>A[j]
	      swap A[i], A[min_i]
	 print all elements in A
	
	Explanation: An implementation of a sorting algorithm
	that sorts user-inputted values in a vector in
	ascending order, then prints the sorted vector.</pre>

Table 1: Example input and output for our LLM.

Example Project

Project Proposal: Code to Pseudocode Using LLM

Wenda Gu wendagu@usc.edu
Egor Cherkashin cherkash@usc.edu
Sarah Chen snchen@usc.edu

Abstract

This document is a project proposal for a Large Language Model trained to convert Python code to pseudocode and an explanation of the code function. It goes over the problem and its importance, related research, and our methodology.

3 Related Work

4 Hypothesis

A specially trained language model for code to pseudocode translation and explanation will perform better than general purpose language models.

5 Methodology

5.1 Baseline Models

To evaluate the effectiveness of our model, we will compare its performance to that of GPT-3.5 Turbo and Mixtral-8x7b.

5.2 Datasets

We will focus on two main datasets. The first dataset is the Python Code Instruction dataset from Kaggle, which has the columns Instruction, Input, Output, and Prompt. This will be used to train our model to understand the purpose of given Python code, which will enable it to generate an explanation for provided code. The second dataset is the Django Dataset, which consists of 16000 training, 1000 development, and 1805 test annotations, where each data point is a line of Python code and its corresponding English pseudocode (Oda et al., 2015).

Input	Output

int main() {
  int n;
  cin >> n;
  vector<int> A;
  A.resize(n);
  for(int i=0; i<A.size(); i++) cin >> A[i]
  for(int i=0; i<A.size(); i++) {
    int min_i = i;
    for(int j=i+1; j<A.size(); j++) {
      if(A[min_i] > A[j]) { min_i = j; }
    swap(A[i], A[min_i]);
  for(int i=0; i<A.size(); i++) cout<<A[i]<<" ";
	in function main
	 let n be integer
	  read n
	 let A be a vector of integers
	   set size of A = n
	  read n elements into A
	   for all elements in A
	    set min_i to i
	     for j=i+1 to size of A exclusive
	      set min_i to j if A[min_i]>A[j]
	     swap A[i], A[min_i]
	 print all elements in A
	
	Explanation: An implementation of a sorting algorithm that sorts user-inputted values in a vector in ascending order, then prints the sorted vector.

Table 1: Example input and output for our LLM.

USC Viterbi

Example Project

Project Proposal: Code to Pseudocode Using LLM

Wenda Gu
wendagu@usc.edu

Egor Cherkashin
cherkash@usc.edu

Sarah Chen
snchen@usc.edu

Abstract

This document is a project proposal for a Large Language Model trained to convert Python code to pseudocode and an explanation of the code function. It goes over the problem and its importance, related research, and our methodology.

3 Related Work

4 Hypothesis

A specially trained language model for code to pseudocode translation and explanation will perform better than general purpose language models.

Input	Output

`int main() {`
`  int n;`
`  cin >> n;`
`  vector<int> A;`
`  A.resize(n);`
`  for(int i=0; i<A.size(); i++) cin >> A[i]`
`  for(int i=0; i<A.size(); i++) {`
`      int min_i = i;`
`      for(int j=i+1; j<A.size(); j++) {`
`          if(A[min_i] > A[j]) { min_i = j; }`
`      swap(A[i], A[min_i]);`
`  for(int i=0; i<A.size(); i++) cout<<A[i]<<" ";`
	`in function main`
	` let n be integer`
	`  read n`
	` let A be a vector of integers`
	`   set size of A = n`
	`  read n elements into A`
	`    for all elements in A`
	`     set min_i to i`
	`      for j=i+1 to size of A exclusive`
	`         set min_i to j if A[min_i]>A[j]`
	`      swap A[i], A[min_i]`
	` print all elements in A`
	
	`Explanation: An implementation of a sorting algorithm that sorts user-inputted values in a vector in ascending order, then prints the sorted vector.`

Table 1: Example input and output for our LLM.

5 Methodology

5.1 Baseline Models

To evaluate the effectiveness of our model, we will compare its performance to that of GPT-3.5 Turbo and Mixtral-8x7b.

5.2 Datasets

We will focus on two main datasets. The first dataset is the Python Code Instruction dataset from Kaggle, which has the columns Instruction, Input, Output, and Prompt. This will be used to train our model to understand the purpose of given Python code, which will enable it to generate an explanation for provided code. The second dataset is the Django Dataset, which consists of 16000 training, 1000 development, and 1805 test annotations, where each data point is a line of Python code and its corresponding English pseudocode (Oda et al., 2015).

References

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. 2015. Learning to generate pseudo-code from source code using statistical machine translation. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 574–584, Lincoln, Nebraska, USA. IEEE Computer Society.

Shuo Ren, Daya Guo, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. 2020. Codebleu: a method for automatic evaluation of code synthesis. *CoRR*, abs/2009.10297.

Tianyi Zhang, Varsha Kishore, Felix Wu, Kilian Q. Weinberger, and Yoav Artzi. 2019. Bertscore: Evaluating text generation with BERT. *CoRR*, abs/1904.09675.

Qinkai Zheng, Xiao Xia, Xu Zou, Yuxiao Dong, Shan Wang, Yufei Xue, Zihan Wang, Lei Shen, Andi Wang, Yang Li, Teng Su, Zhilin Yang, and Jie Tang. 2023. Codegeex: A pre-trained model for code generation with multilingual evaluations on humaneval-x.

Lecture Outline

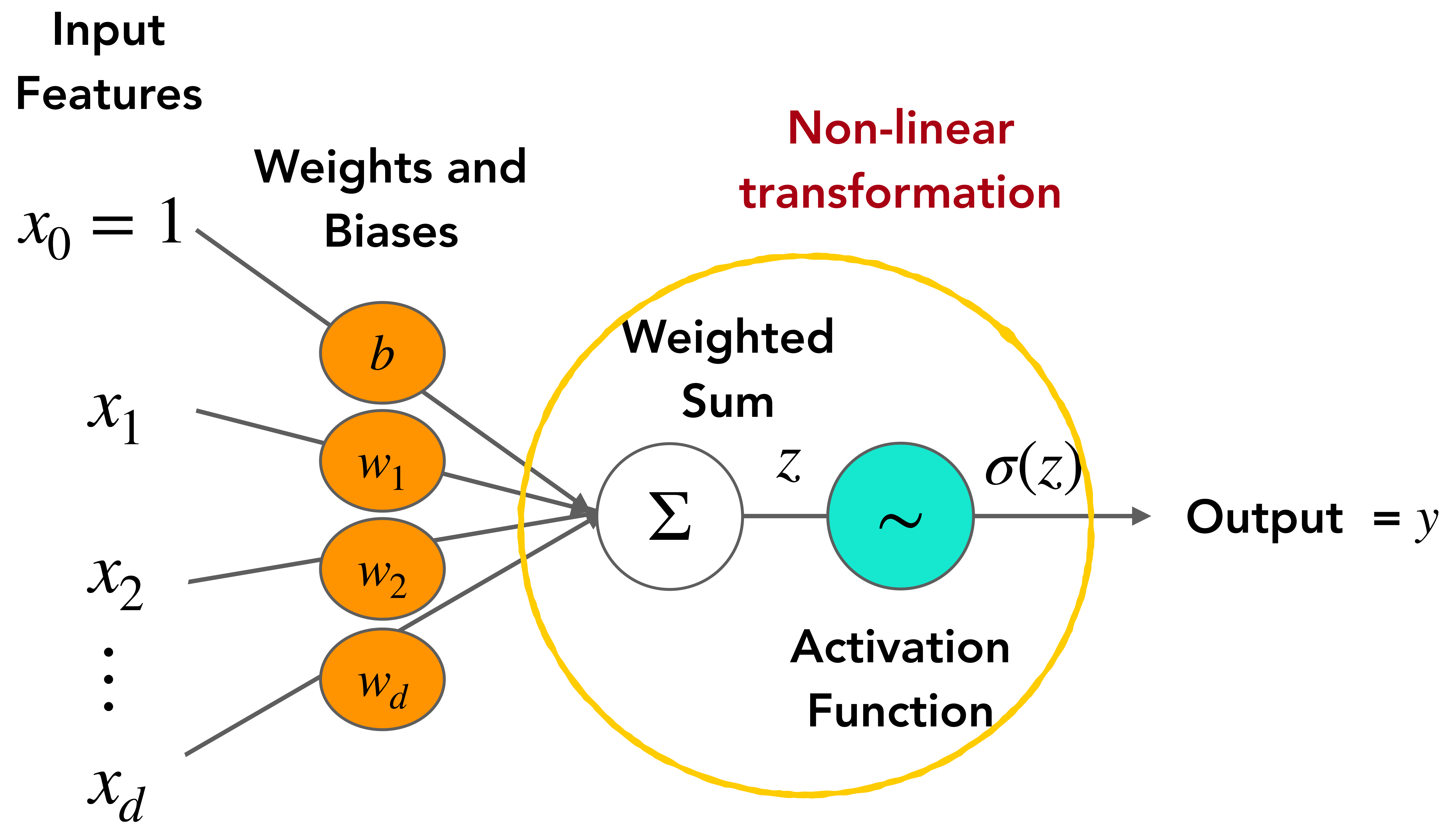
- Announcements
- Recap: FFNN
- Feedforward Neural Net Language Models
- FFNN for Classification
- Training FFNNs
- Computation Graph and Backdrop

Recap:
Feedforward Neural Nets

USC Viterbi

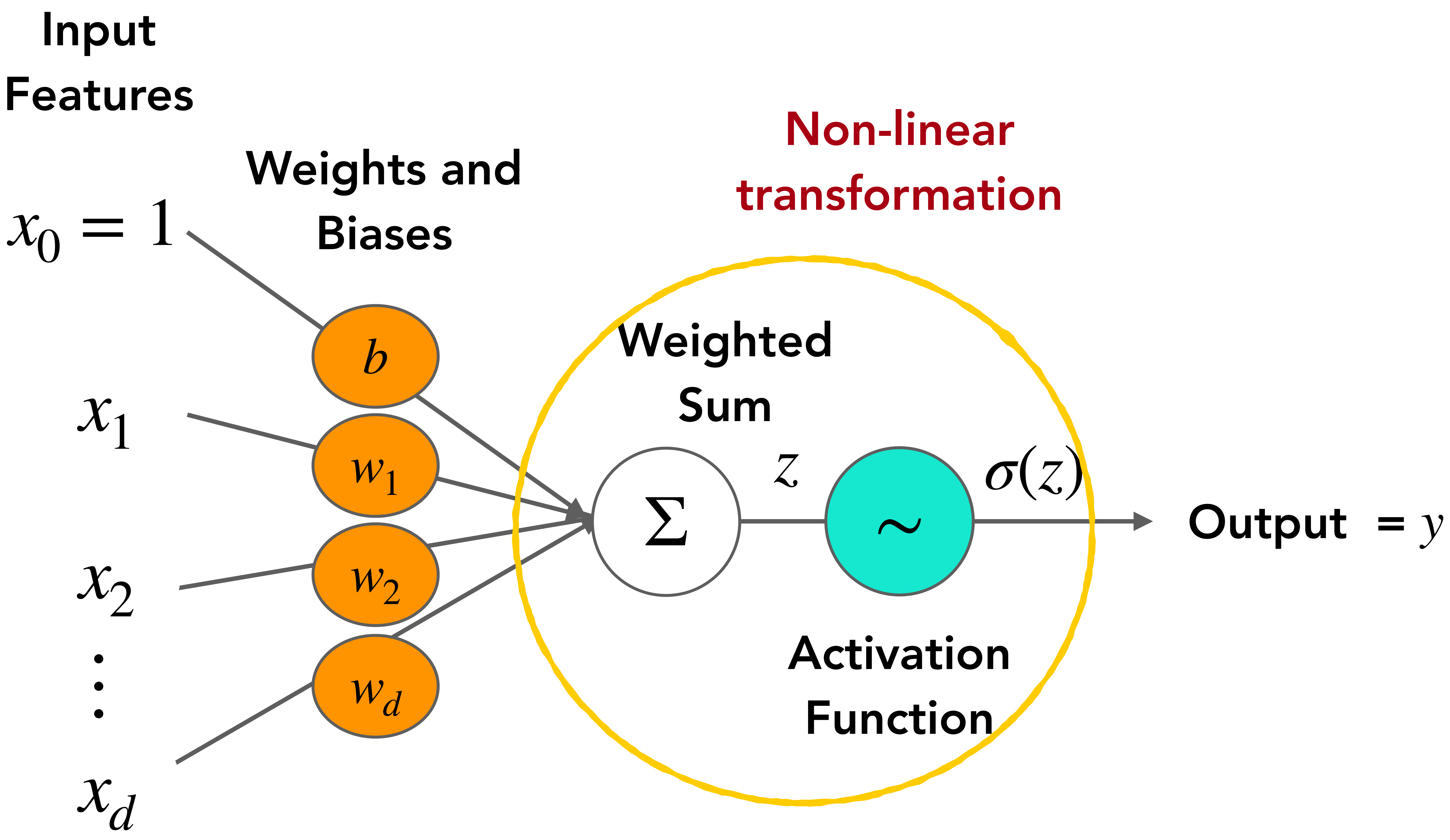
Neural Network Unit

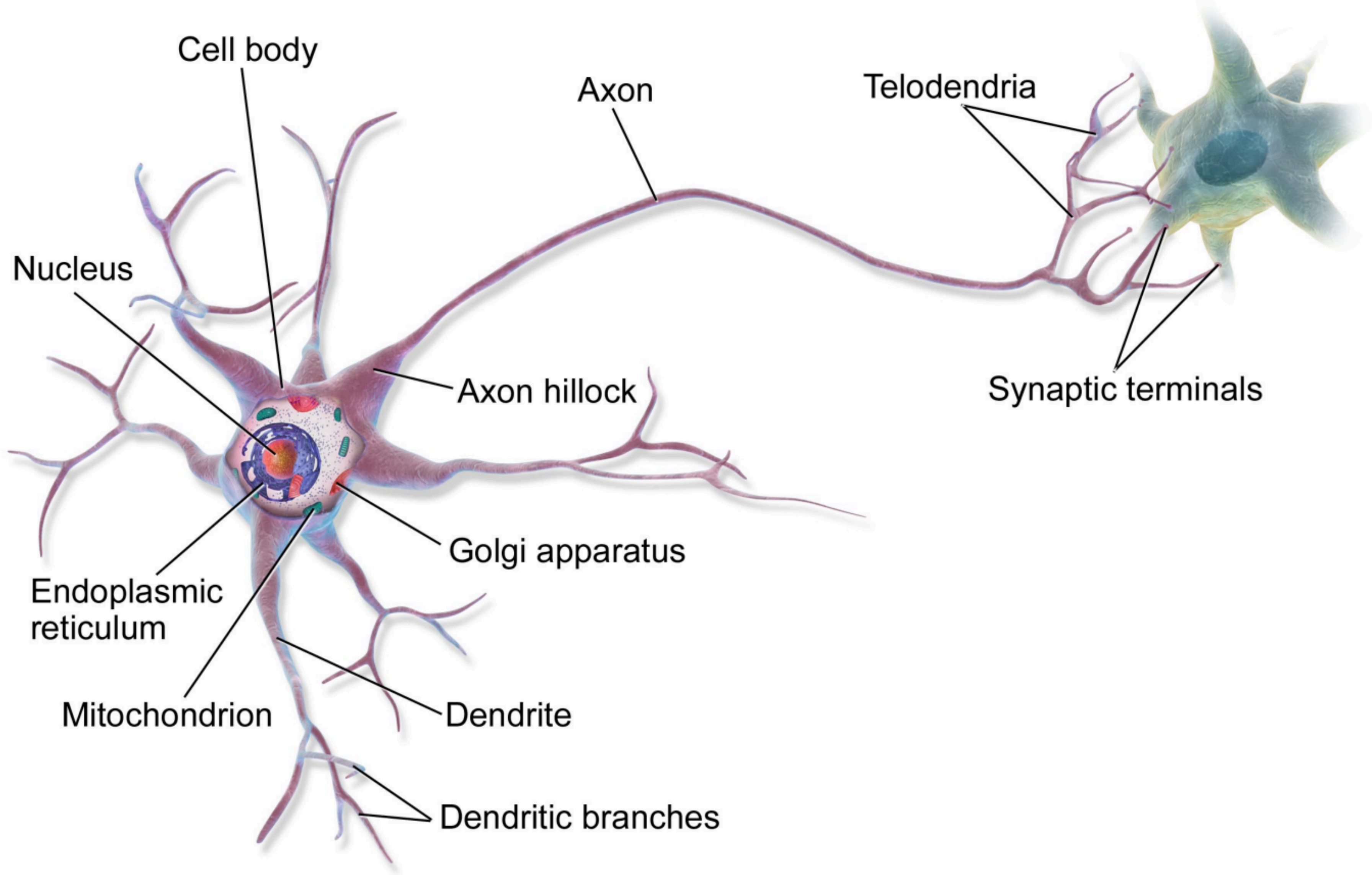
Logistic Regression is a very simple neural network



Neural Network Unit

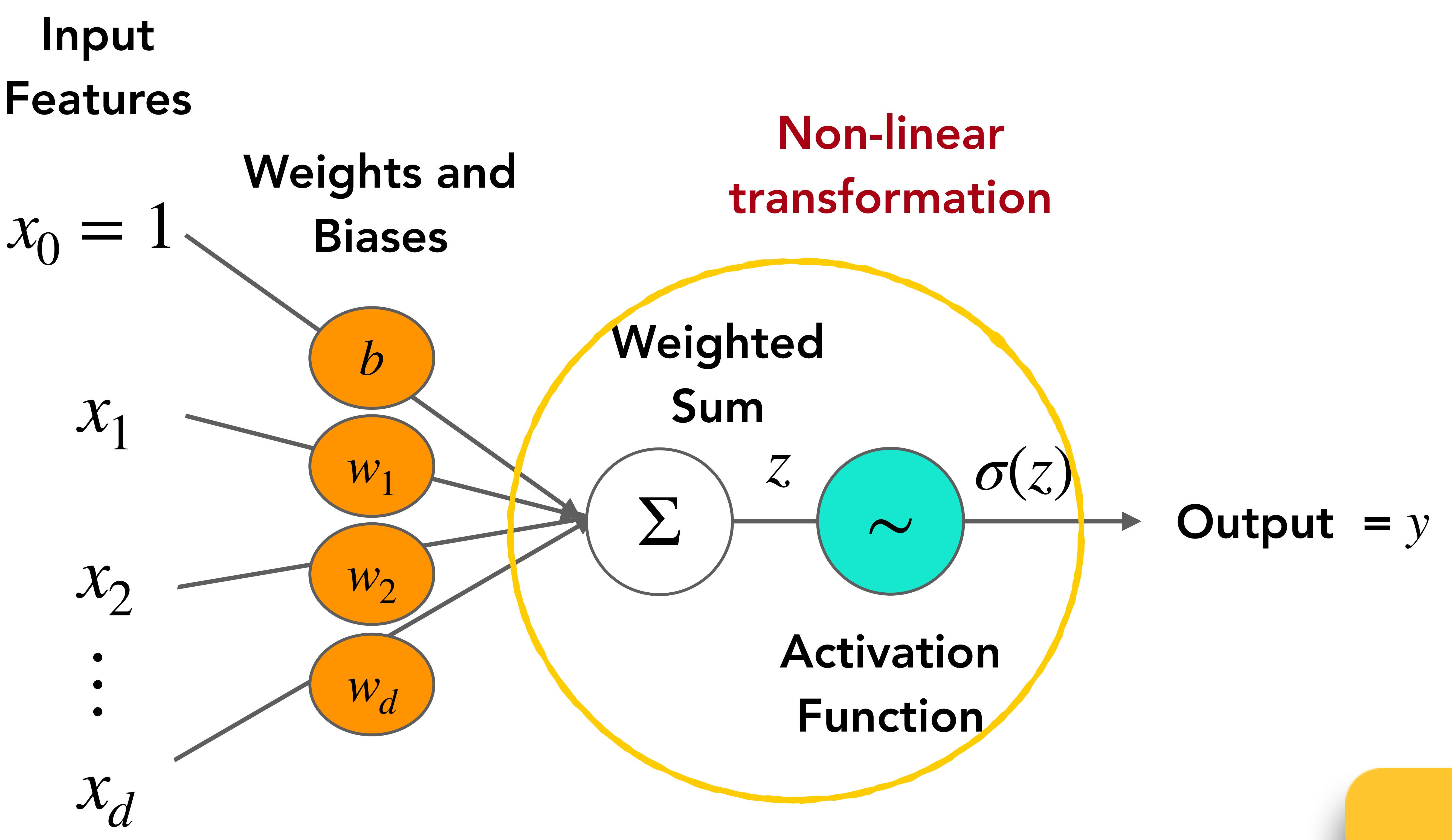
Logistic Regression is a very simple neural network

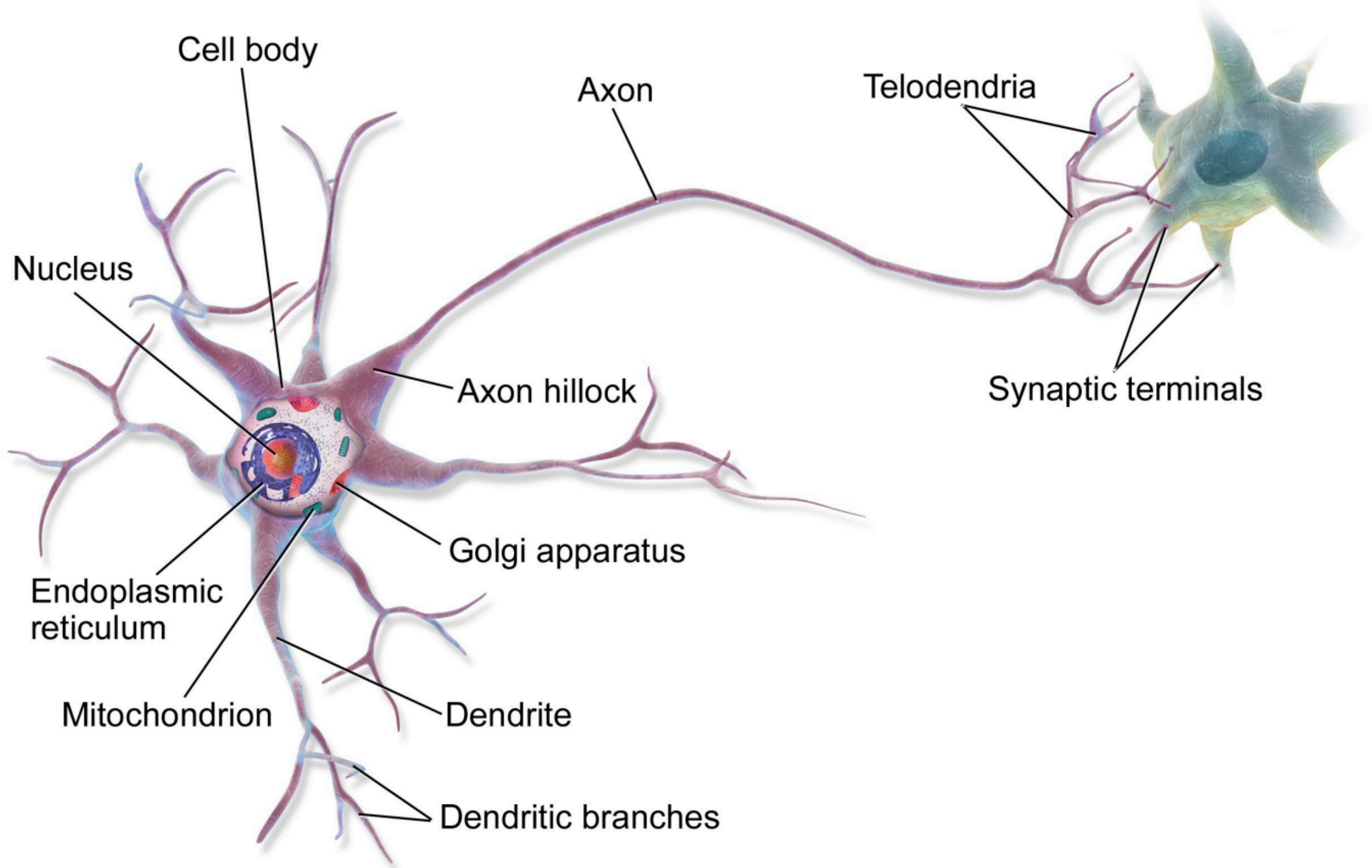




Neural Network Unit

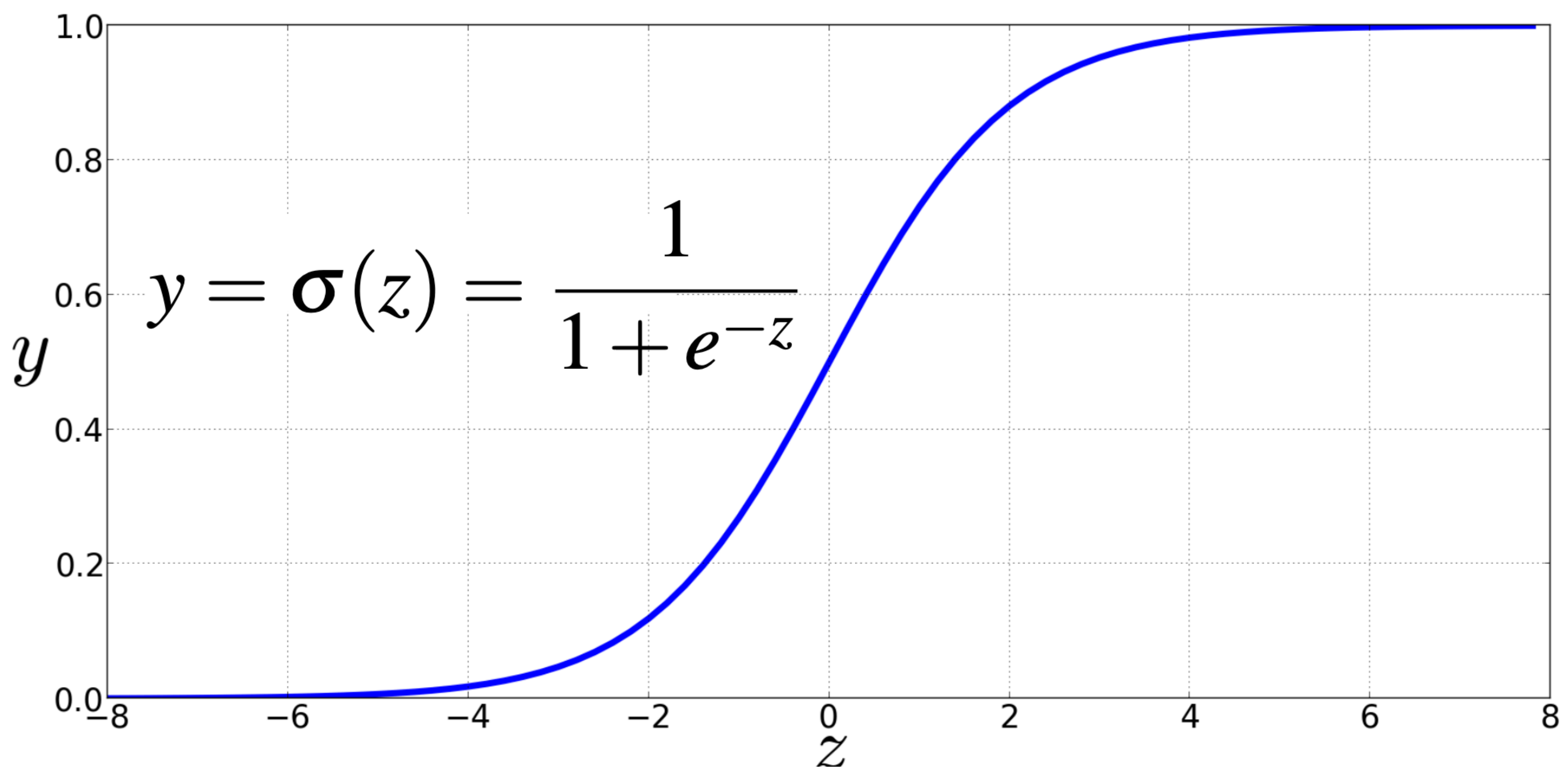
Logistic Regression is a very simple neural network





Resembles a neuron in the brain!

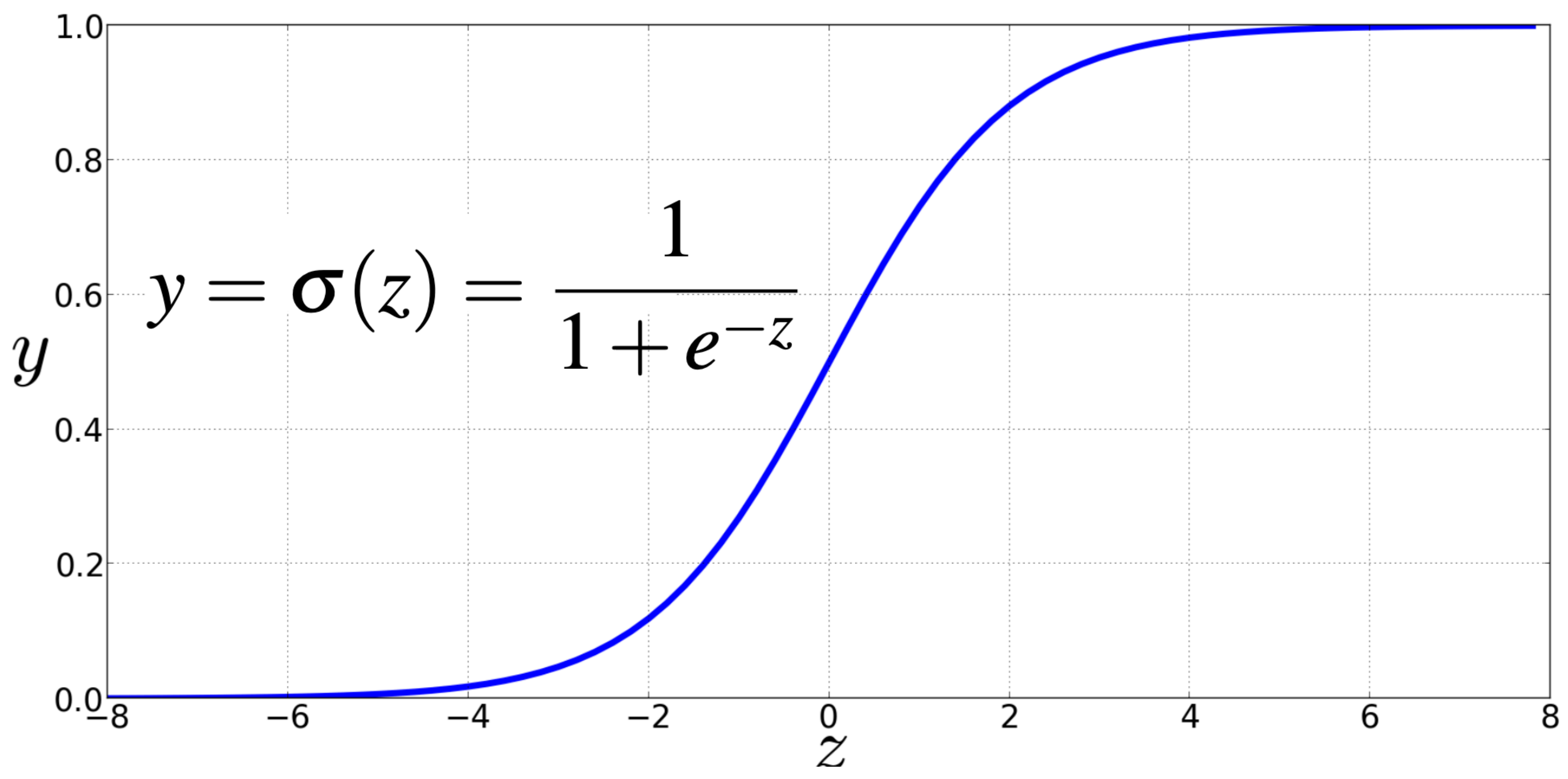
Non-Linear Activation Functions

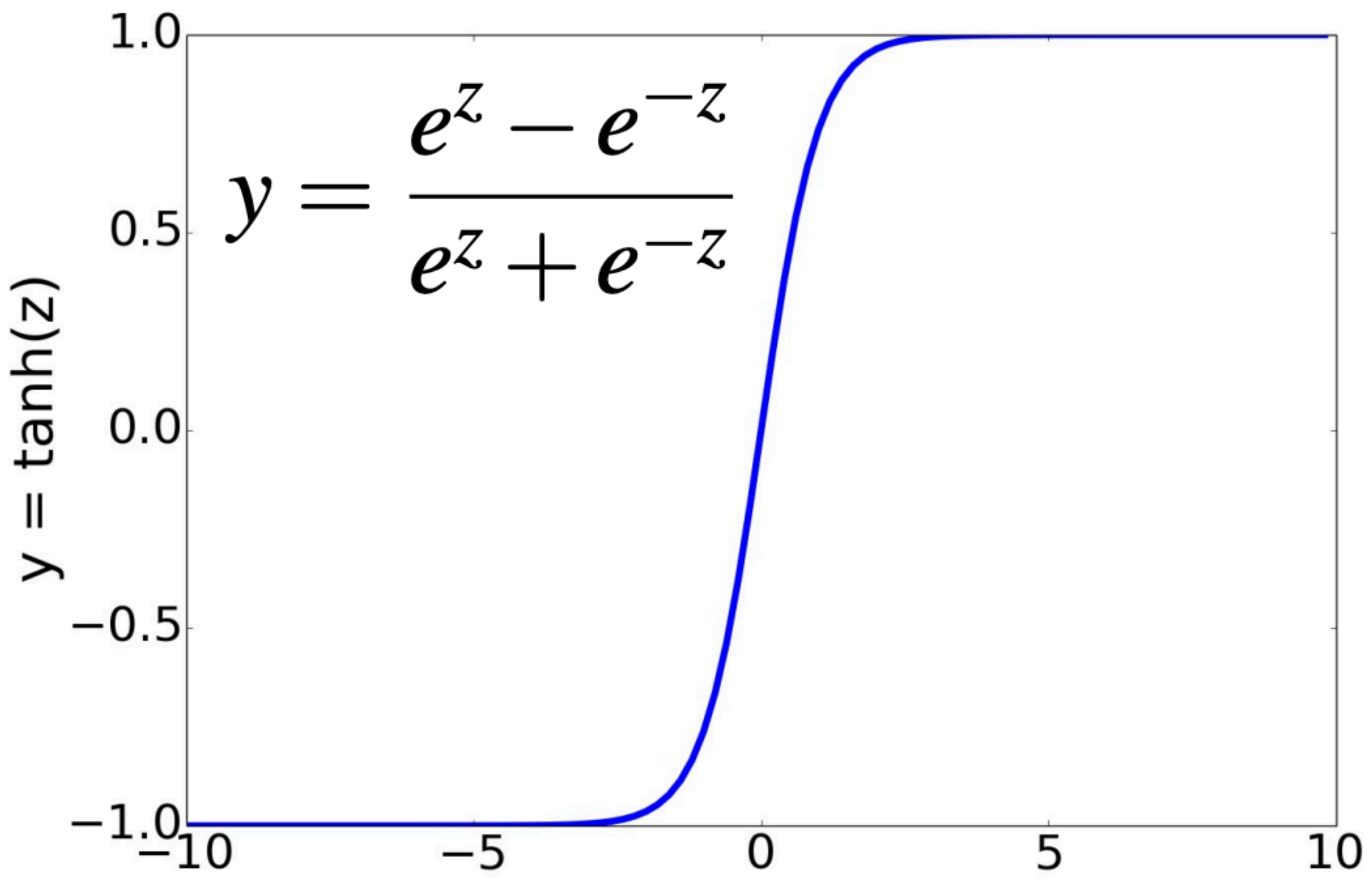


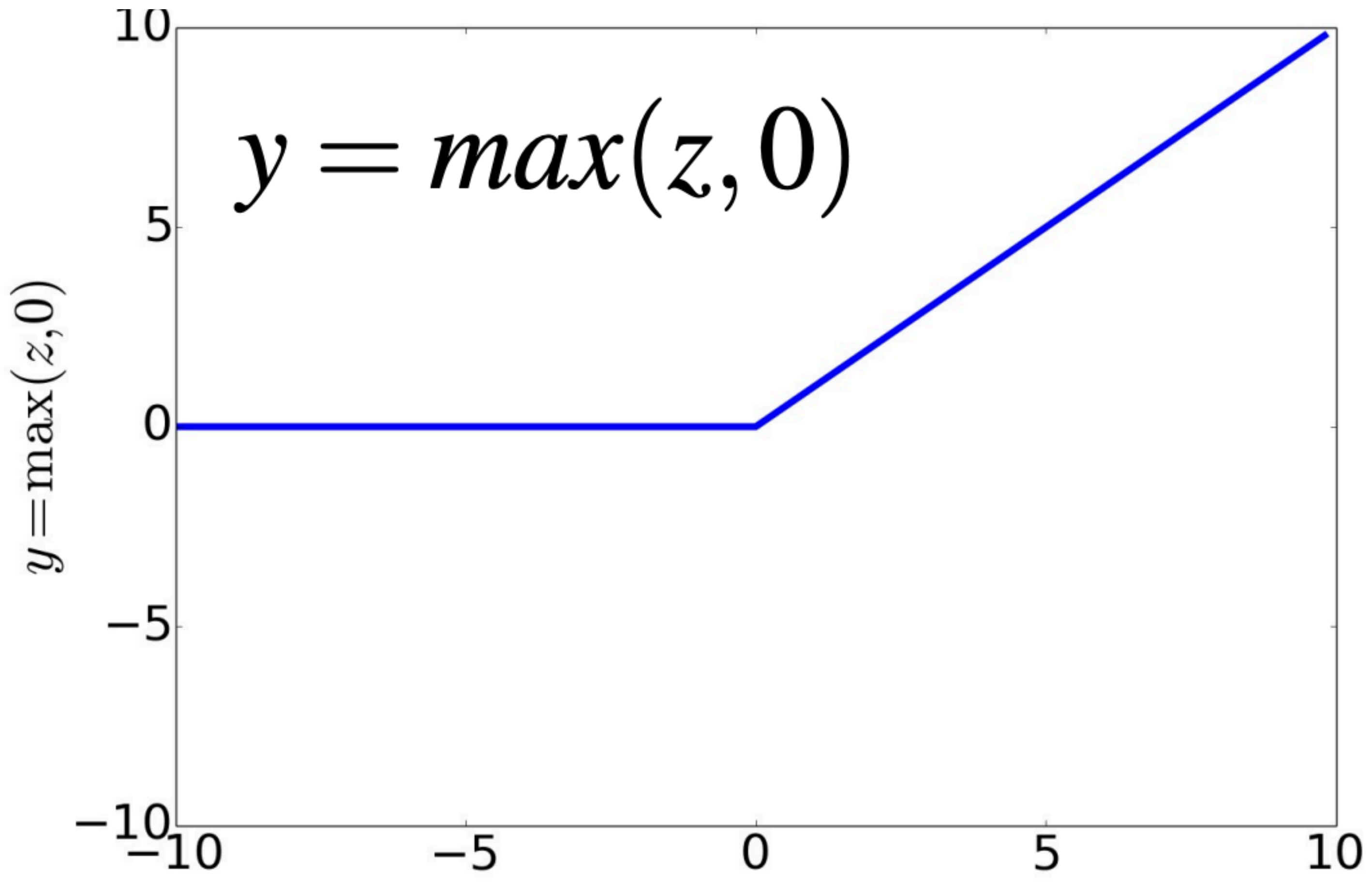
sigmoid

The key ingredient of a neural network is the non-linear activation function

Non-Linear Activation Functions







sigmoid

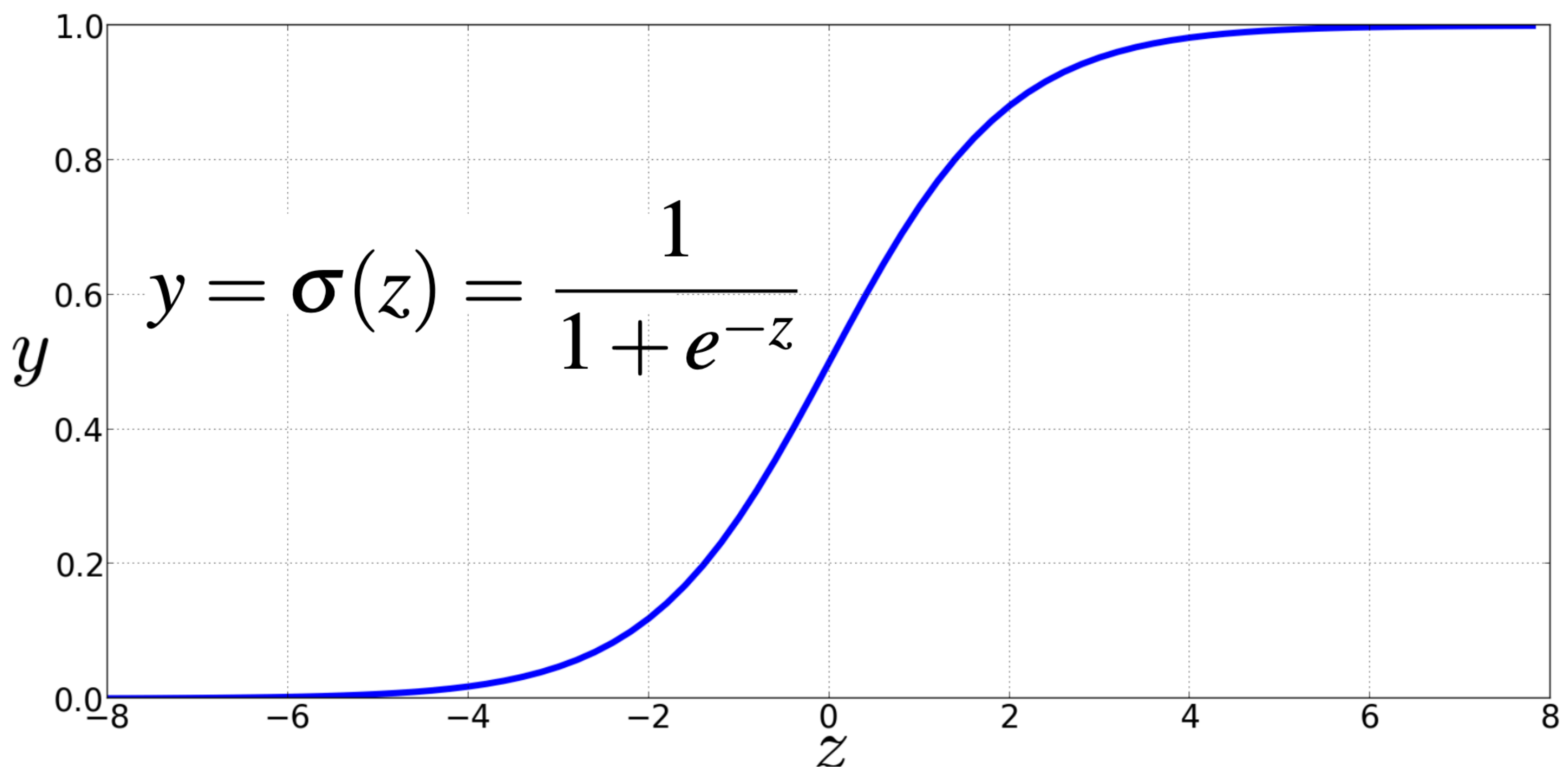
tanh

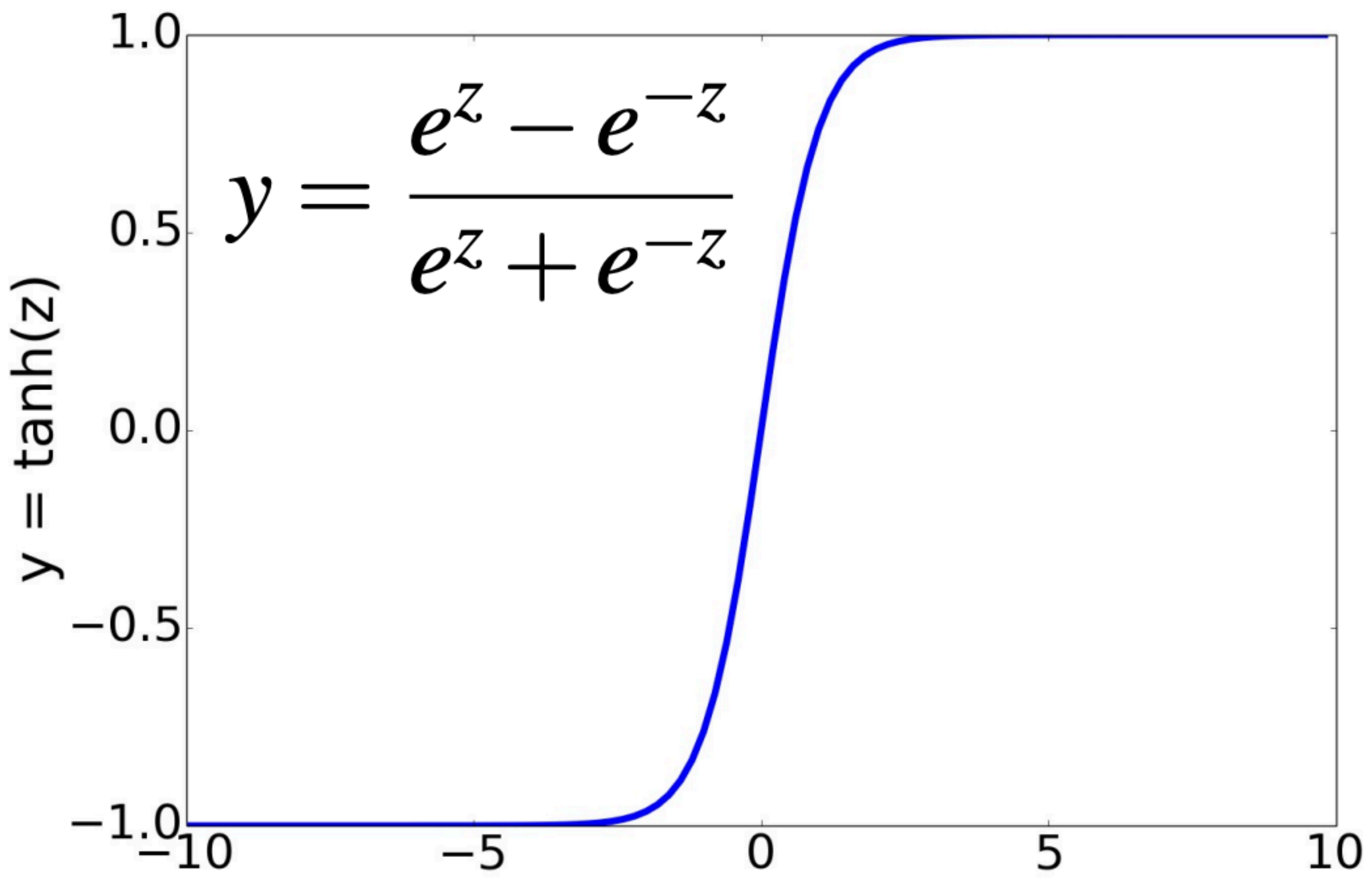
relu (Rectified Linear Unit)

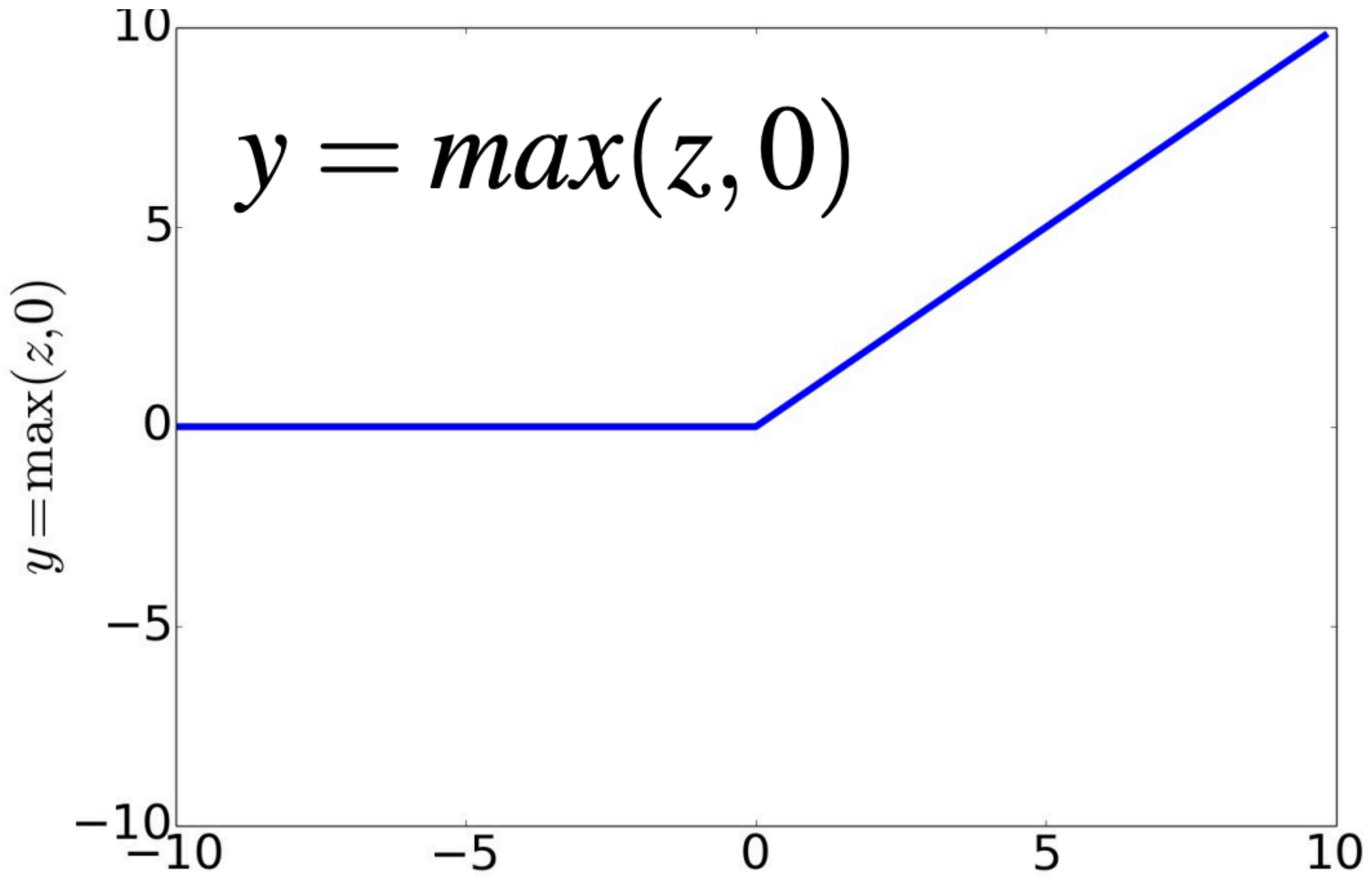
The key ingredient of a neural network is the non-linear activation function

Non-Linear Activation Functions

Most common!







sigmoid

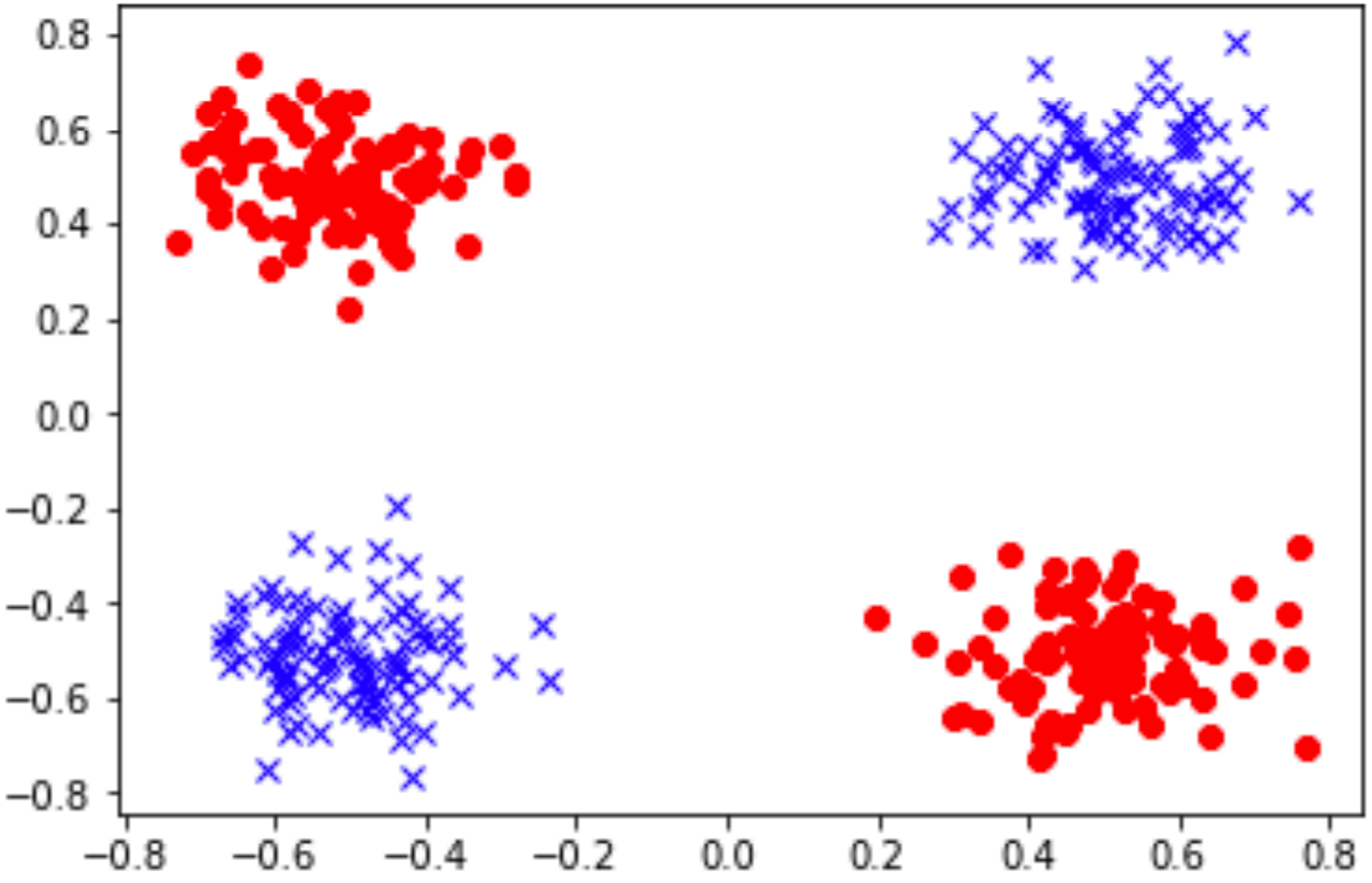
tanh

relu (Rectified Linear Unit)

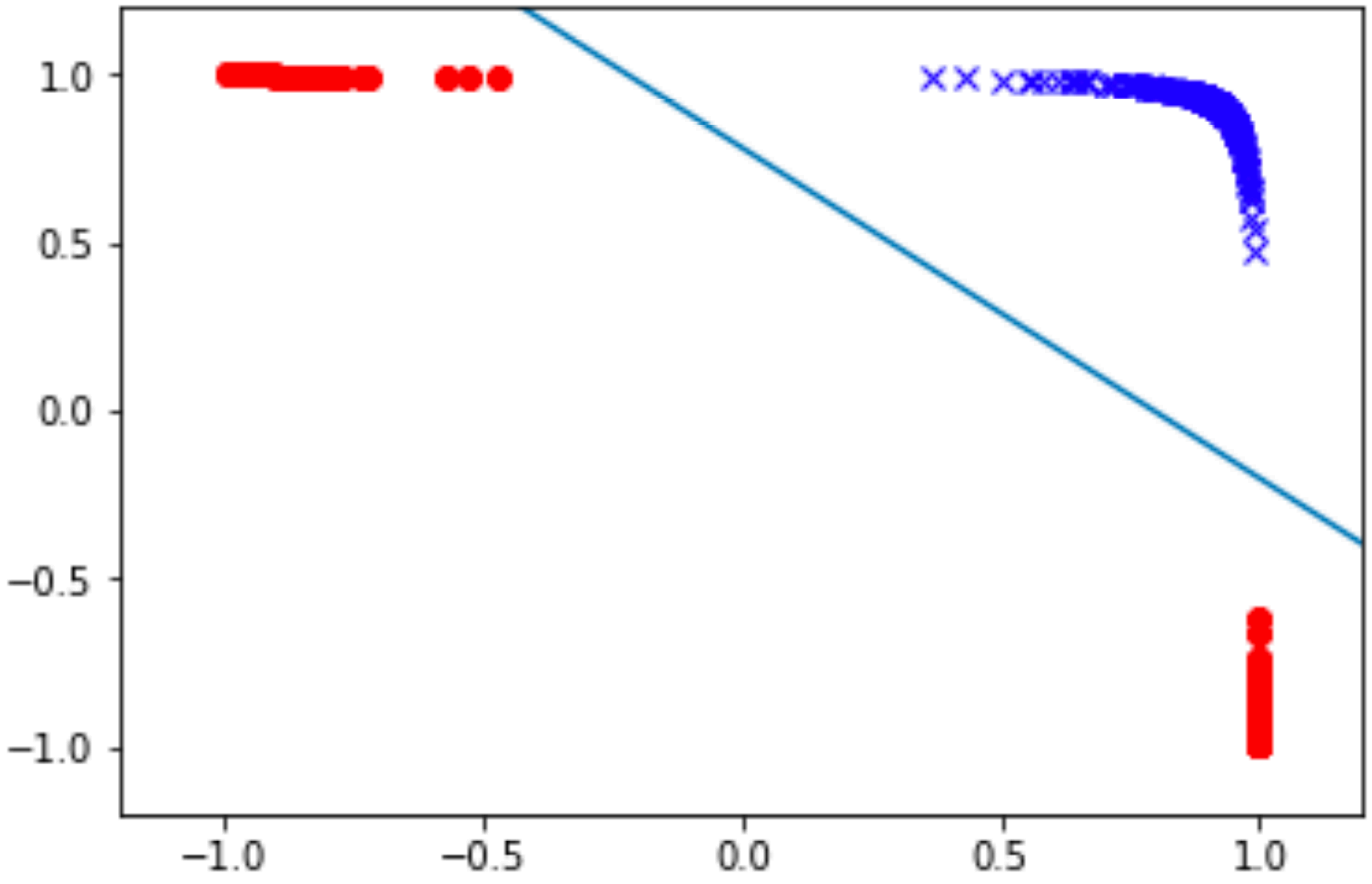
The key ingredient of a neural network is the non-linear activation function

Power of non-linearity

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



After a tanh(·) transformation:



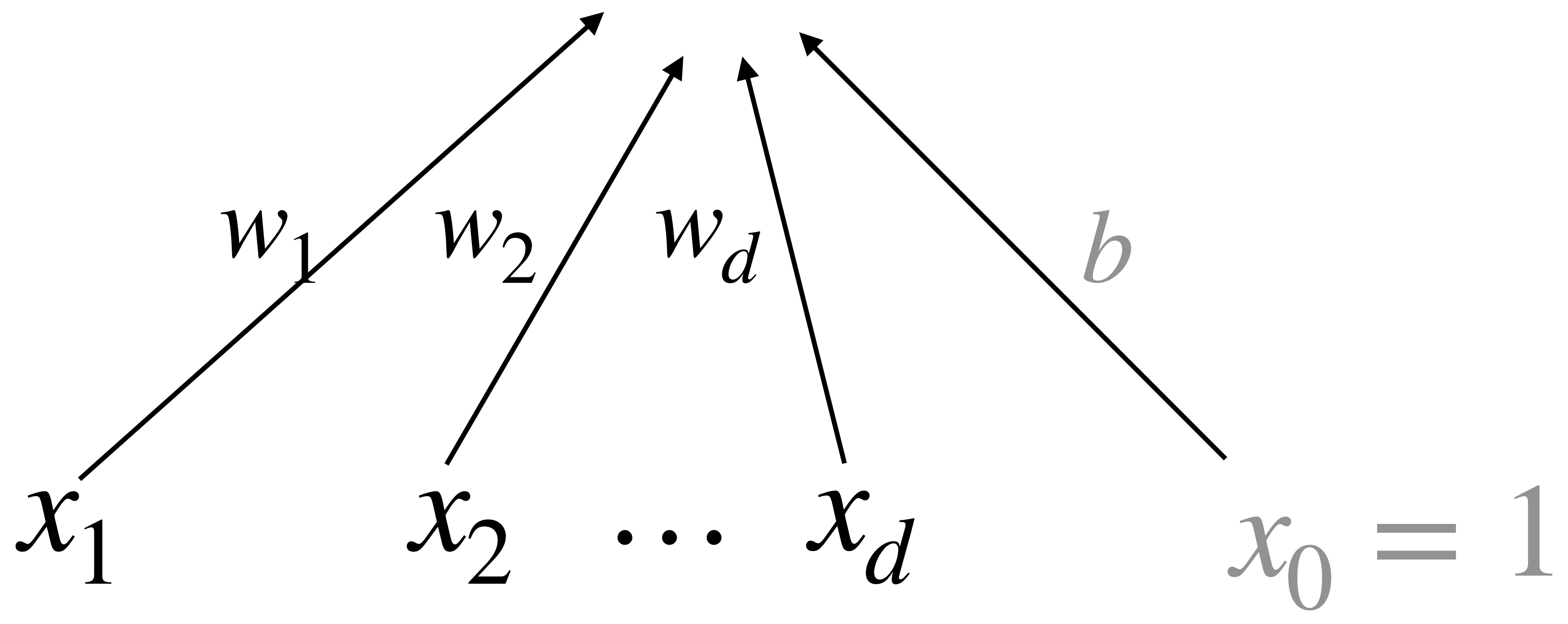
Binary Logistic Regression

Binary Logistic Regression

Input layer: vector $\mathbf{x}$ x_1 x_2 $\cdots$ x_d $x_0 = 1$

Binary Logistic Regression

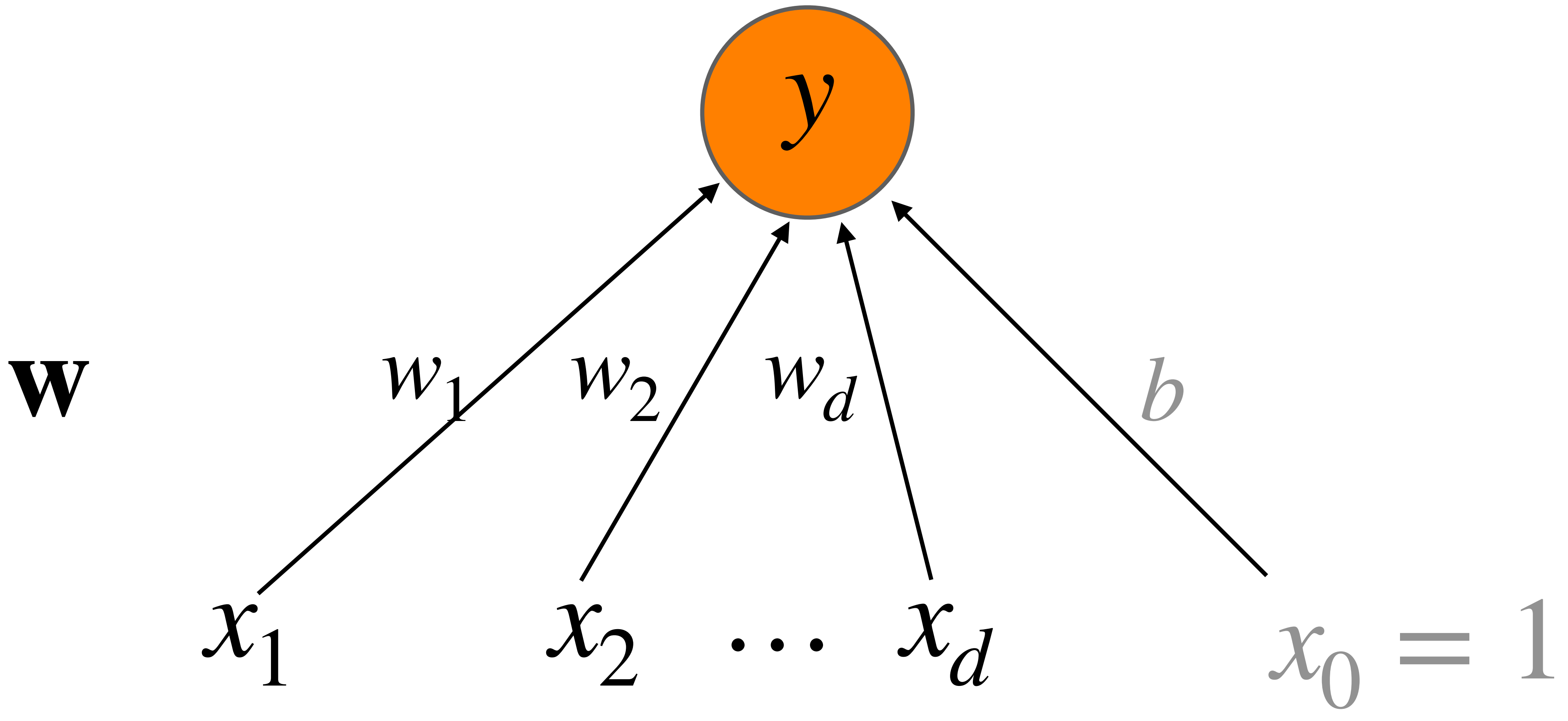
$\mathbf{w}$



Input layer: vector $\mathbf{x}$

Binary Logistic Regression

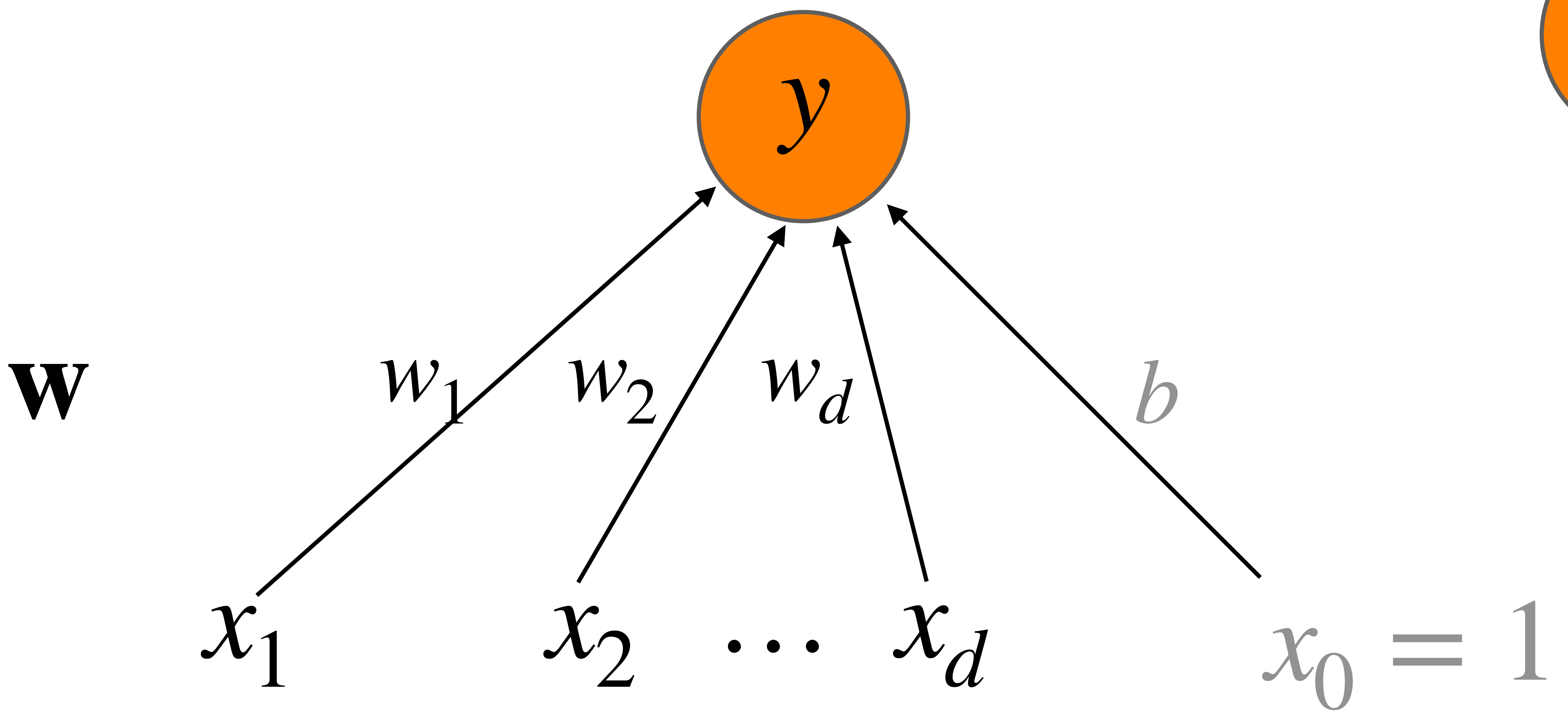
Output layer: $y = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$



Input layer: vector $\mathbf{x}$

Binary Logistic Regression

Output layer: $y = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$

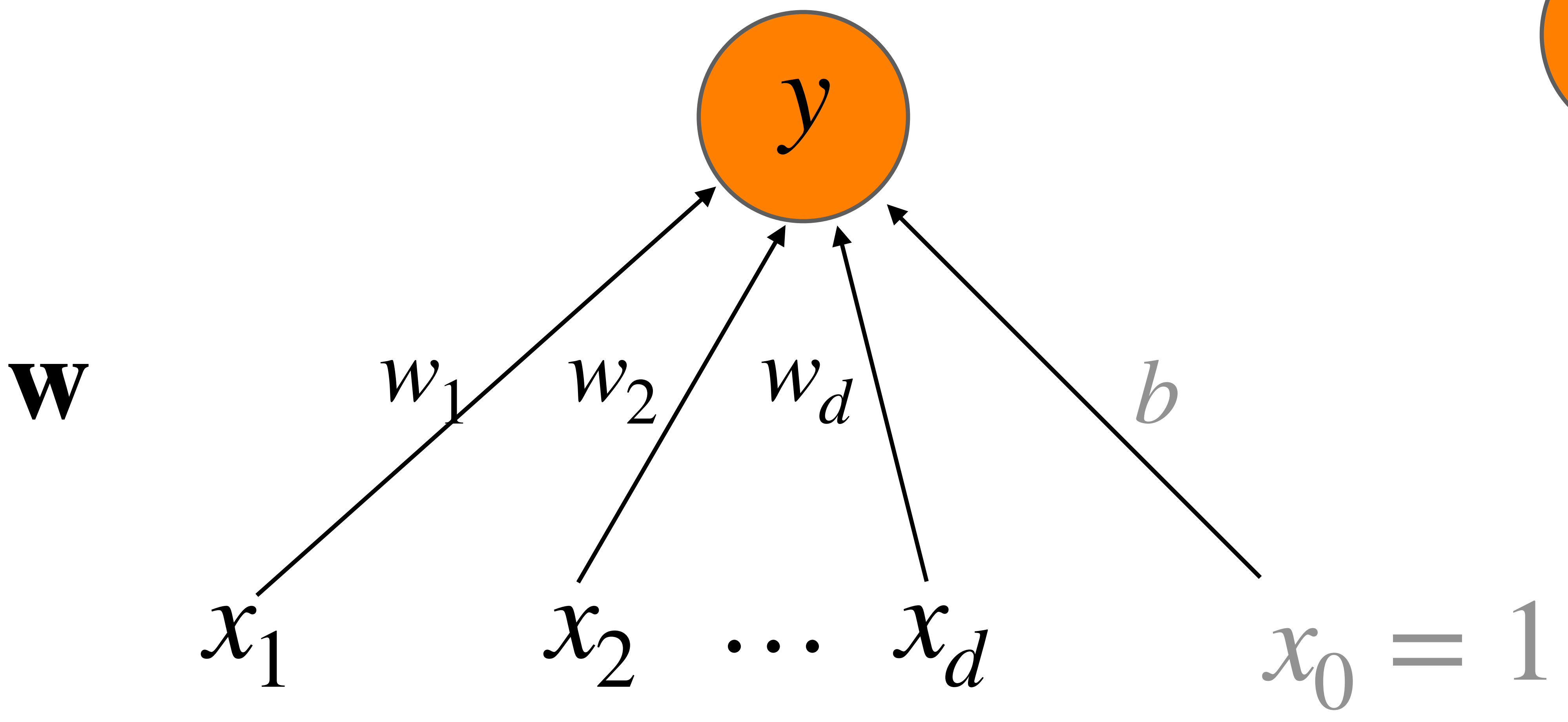


Input layer: vector $\mathbf{x}$

Weighted sum of all incoming, followed by a non-linear activation

Binary Logistic Regression

Output layer: $y = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$



Input layer: vector $\mathbf{x}$

Weighted sum of all incoming, followed by a non-linear activation

1-layer Network

Don't count the input layer in counting layers!

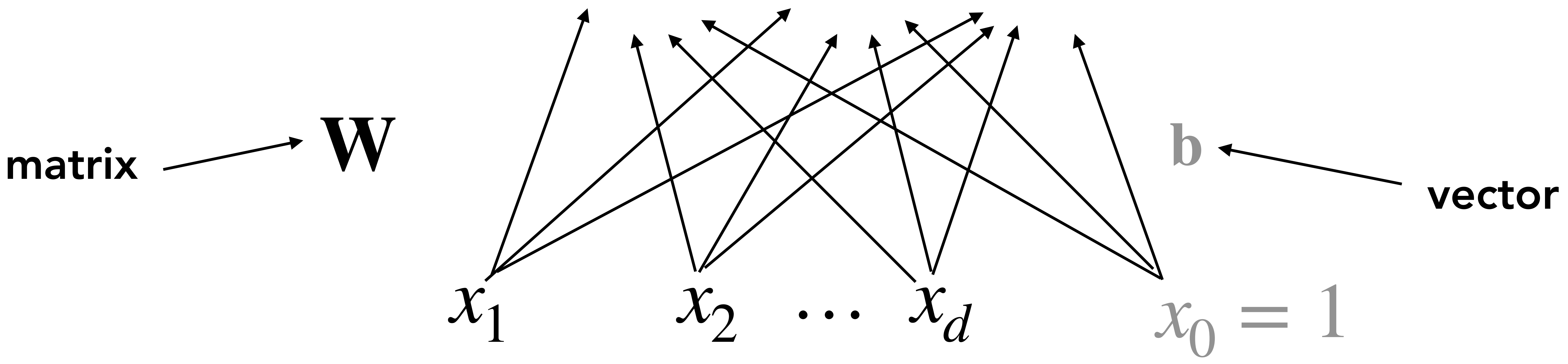
Multinomial Logistic Regression

Multinomial Logistic Regression

Input layer: vector $\mathbf{x}$

$x_1 \quad x_2 \quad \cdots \quad x_d \qquad x_0 = 1$

Multinomial Logistic Regression

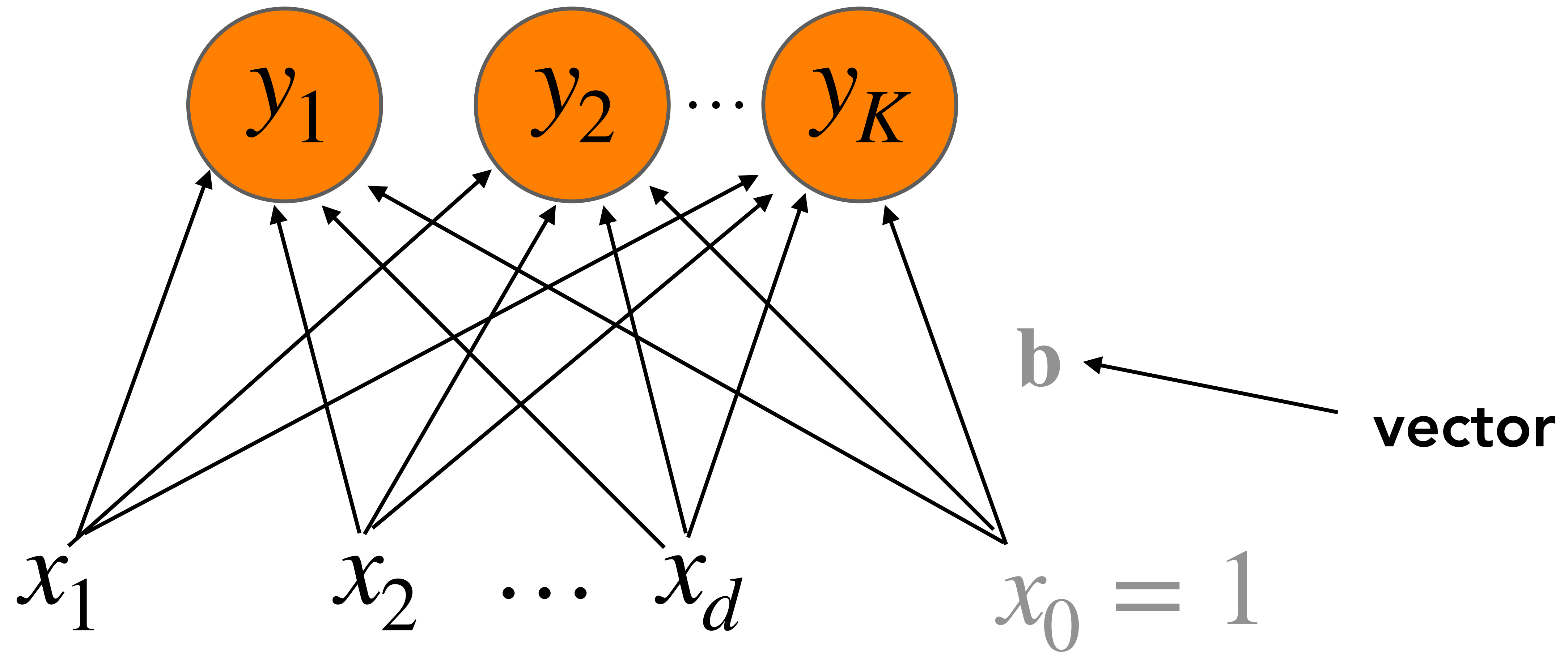


Input layer: vector $\mathbf{x}$

Multinomial Logistic Regression

Output layer: $\mathbf{y} = \text{softmax}(\mathbf{w} \cdot \mathbf{x} + b)$

matrix $\mathbf{W}$

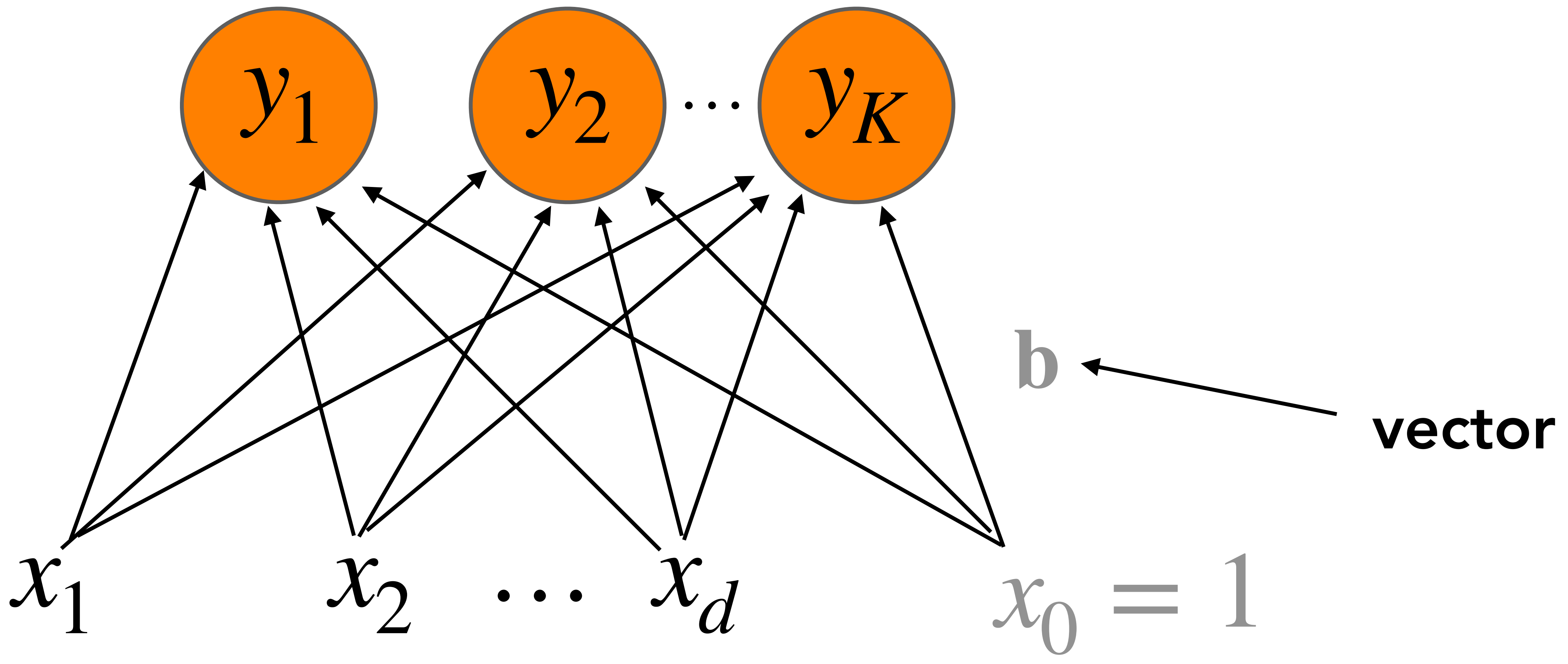


Input layer: vector $\mathbf{x}$

Multinomial Logistic Regression

Output layer: $\mathbf{y} = \text{softmax}(\mathbf{w} \cdot \mathbf{x} + b)$

matrix → $\mathbf{W}$



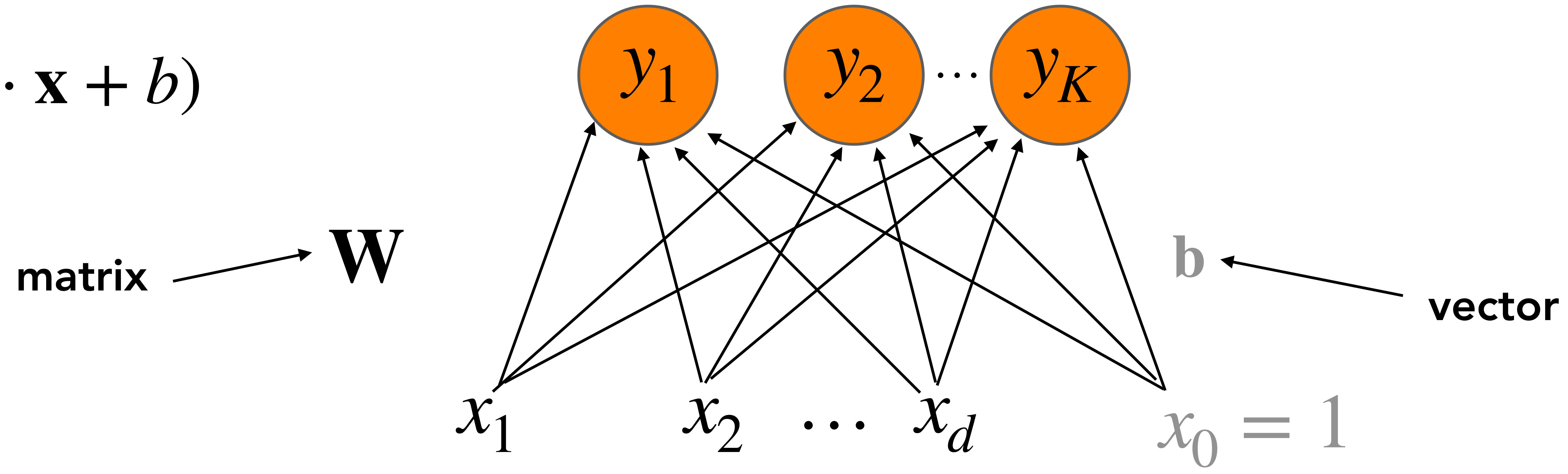
Input layer: vector $\mathbf{x}$

1-layer Network

Multinomial Logistic Regression

Output layer: $\mathbf{y} = \text{softmax}(\mathbf{w} \cdot \mathbf{x} + b)$

Input layer: vector $\mathbf{x}$



1-layer Network

Fully connected single layer network

Two-layer Feedforward Network

Two-layer Feedforward Network

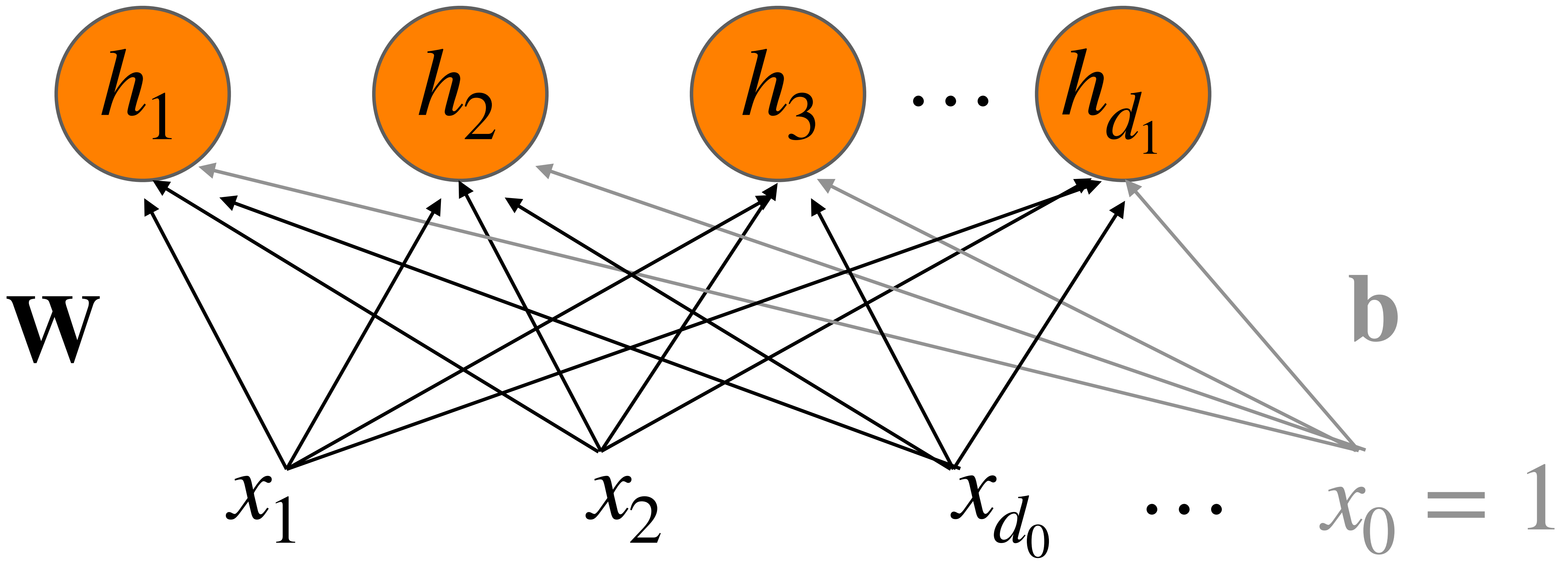
Input layer: vector $\mathbf{x}$

x_1 x_2 x_{d_0} $\cdots$ $x_0 = 1$

Two-layer Feedforward Network

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$

Input layer: vector $\mathbf{x}$

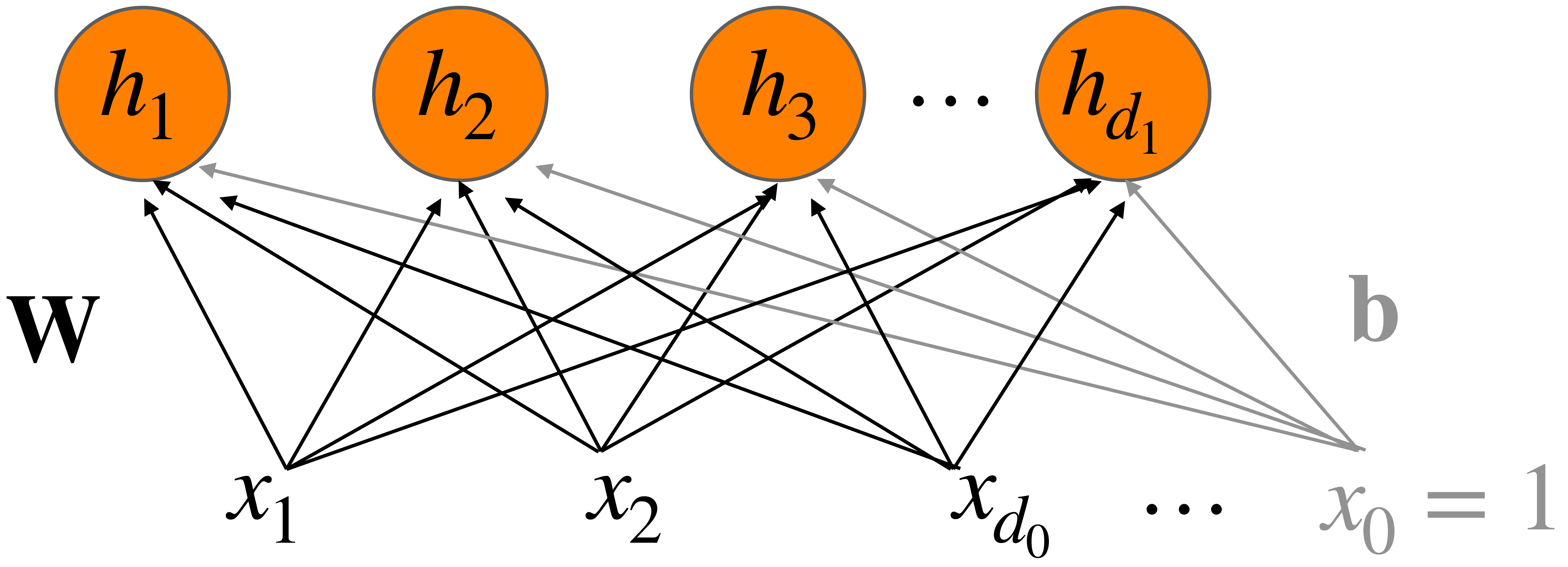


Two-layer Feedforward Network

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$

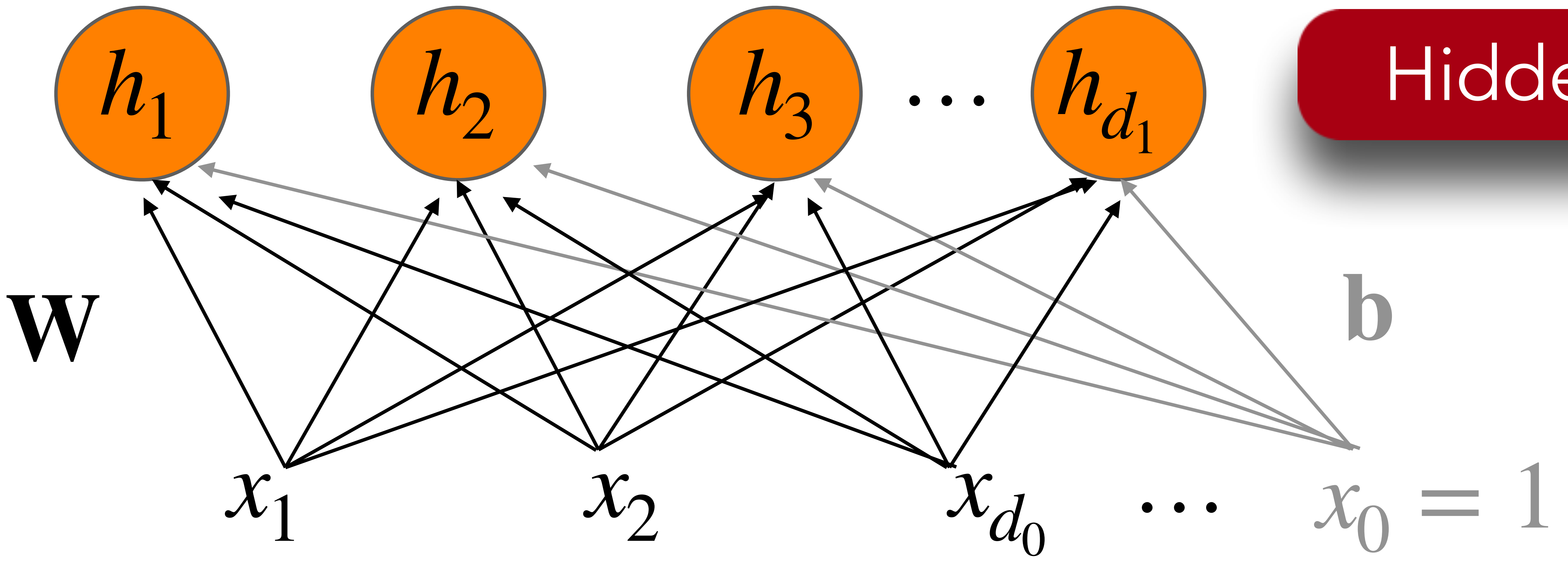


Two-layer Feedforward Network

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$



Hidden Unit, h_i

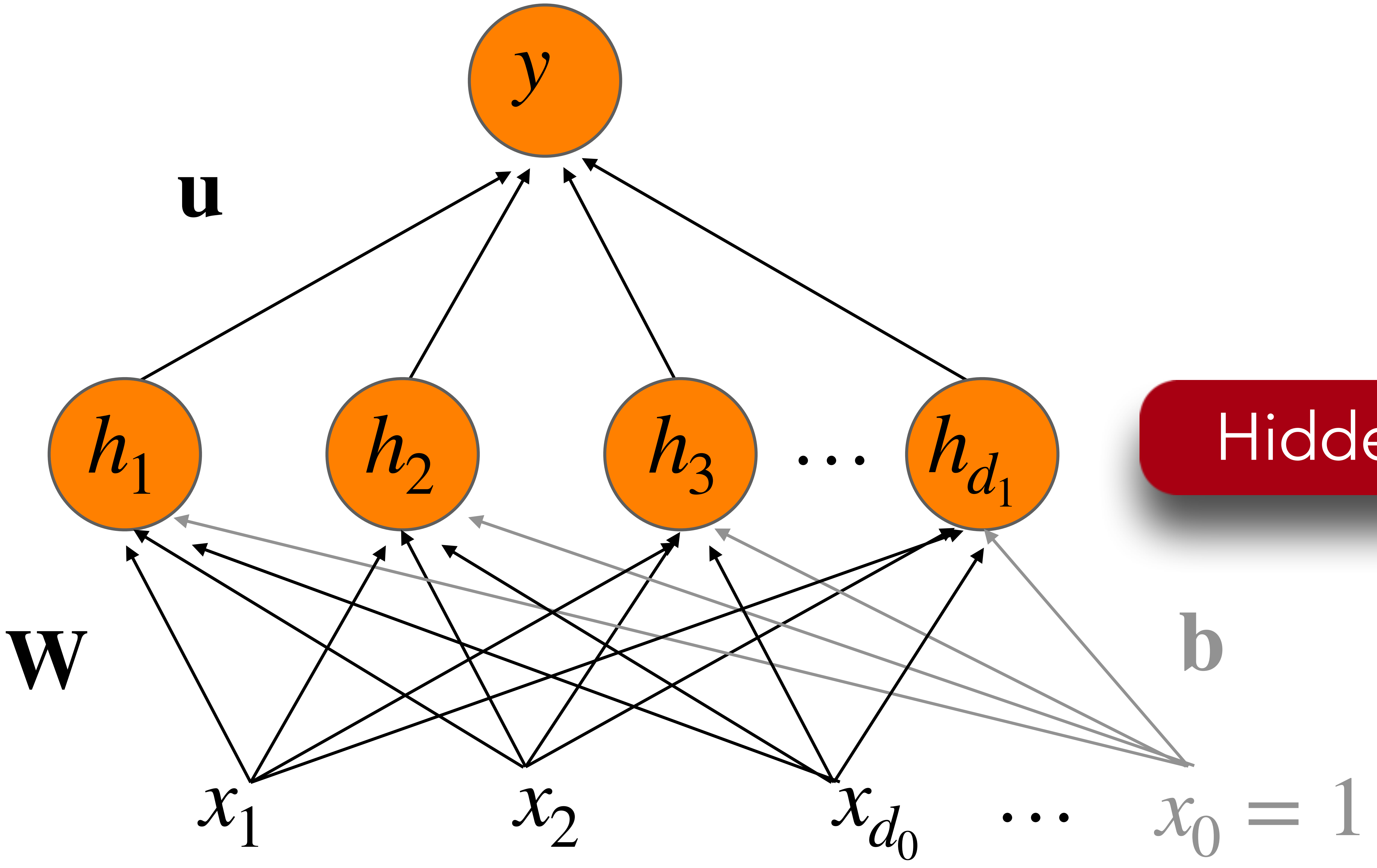
Two-layer Feedforward Network

Output layer: $y = \sigma(\mathbf{uh})$

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$



Hidden Unit, h_i

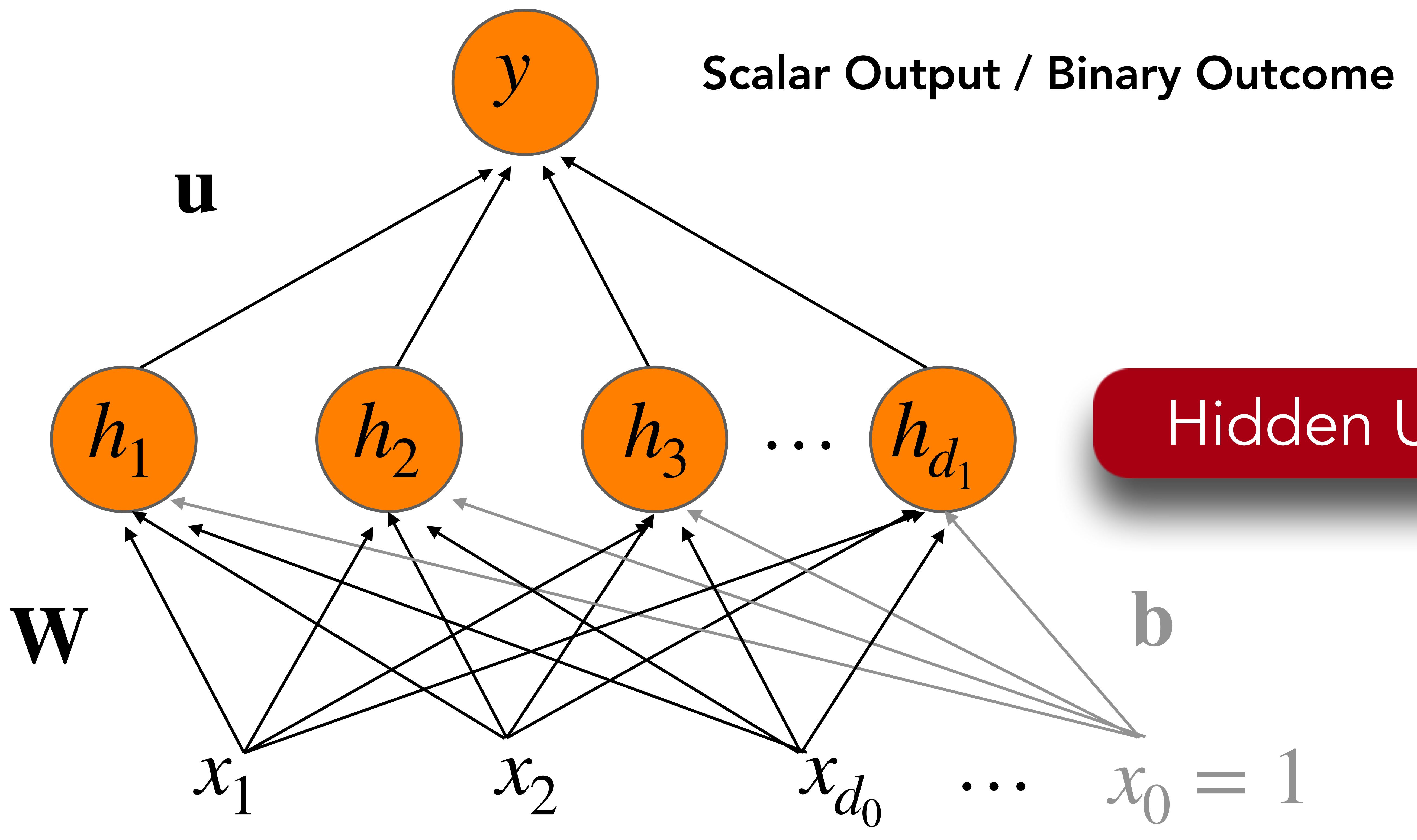
Two-layer Feedforward Network

Output layer: $y = \sigma(\mathbf{uh})$

Hidden layer: $\mathbf{h} = g(\mathbf{Wx} + \mathbf{b})$

Usually ReLU or $\tanh$

Input layer: vector $\mathbf{x}$



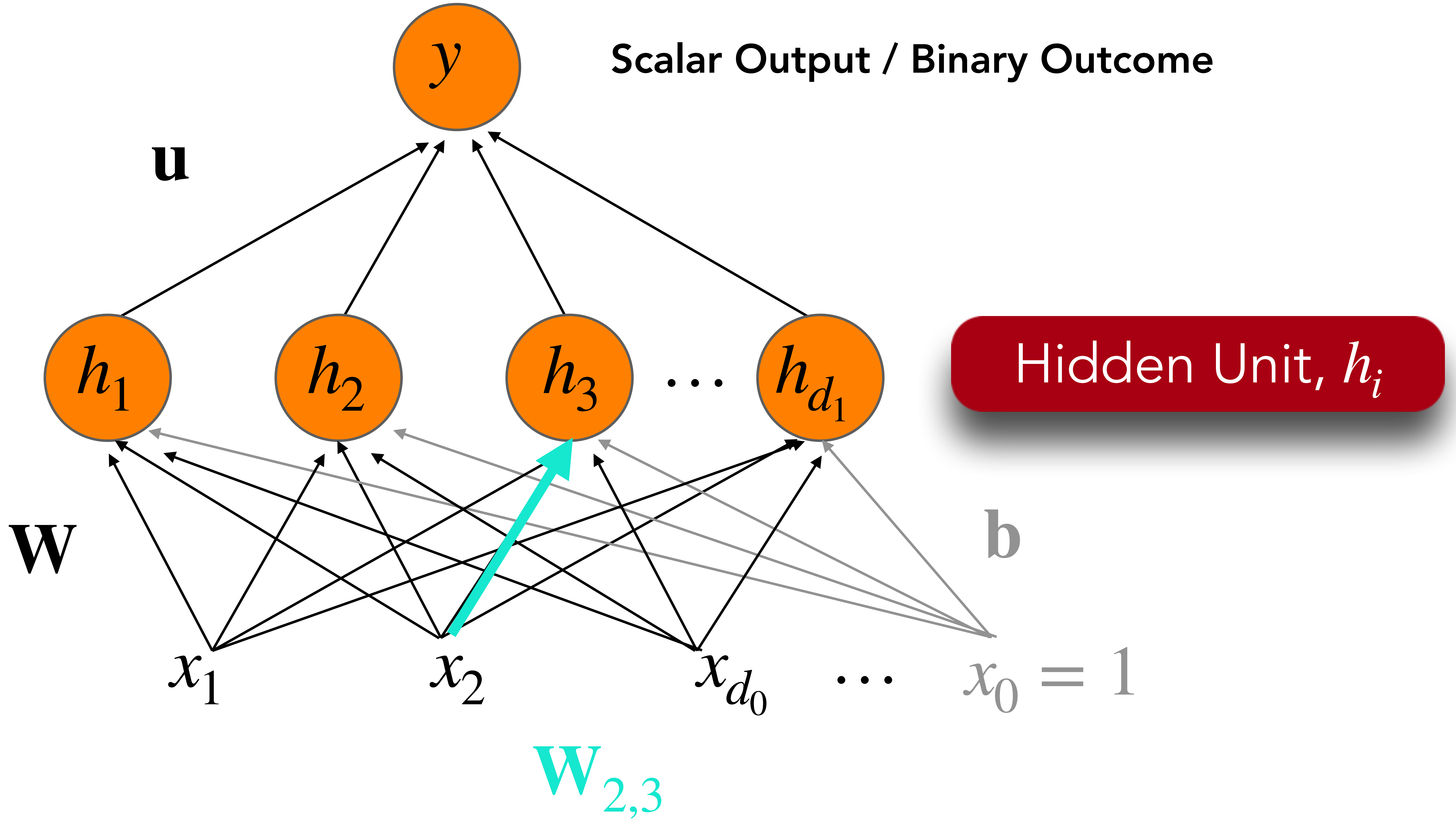
Two-layer Feedforward Network

Output layer: $y = \sigma(\mathbf{uh})$

Hidden layer: $\mathbf{h} = g(\mathbf{Wx} + \mathbf{b})$

Usually ReLU or tanh

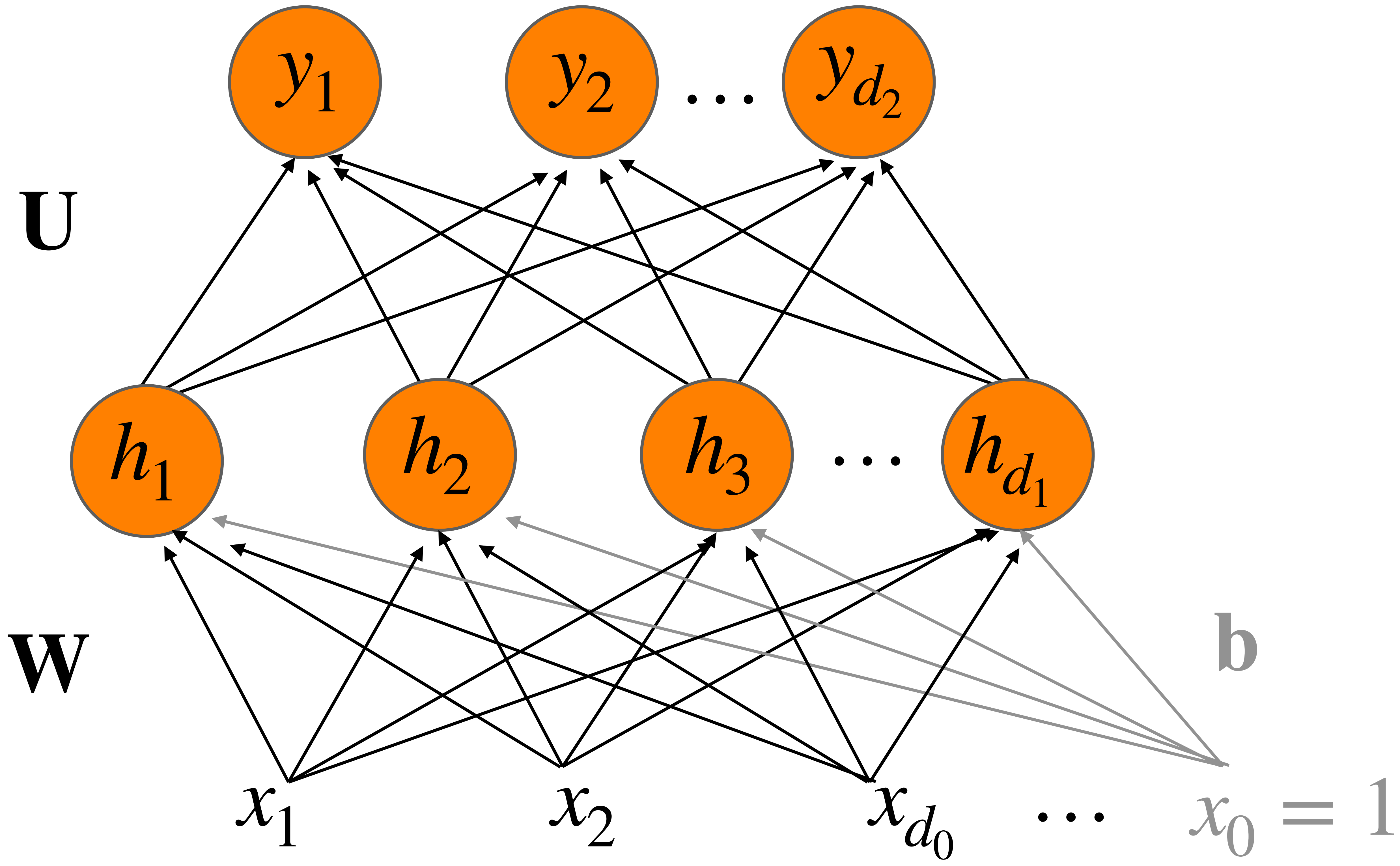
Input layer: vector $\mathbf{x}$



Two-layer Feedforward Network with Softmax Output

Output layer: $\mathbf{y} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$

Input layer: vector $\mathbf{x}$



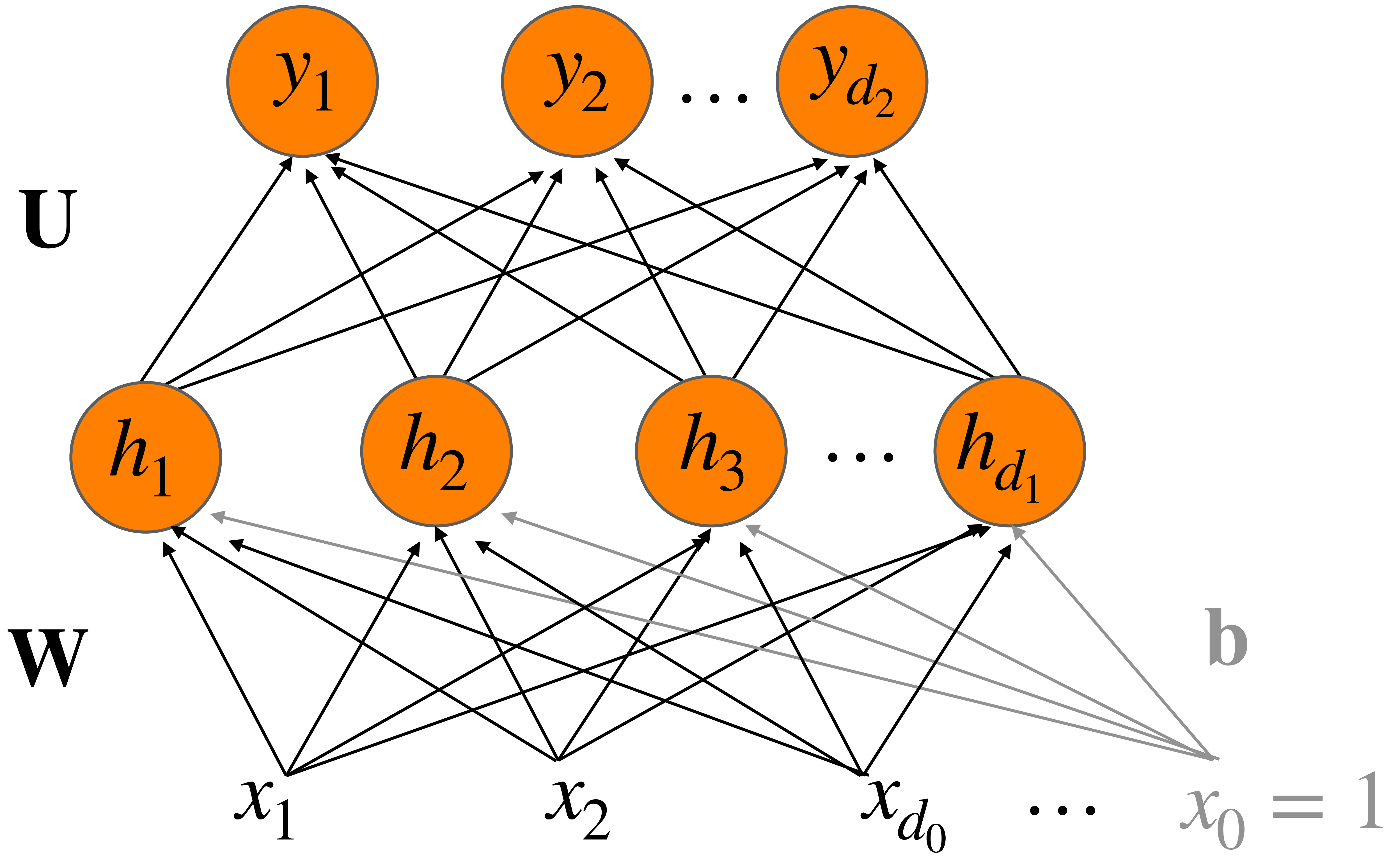
Two-layer Feedforward Network with Softmax Output

Output layer: $\mathbf{y} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$



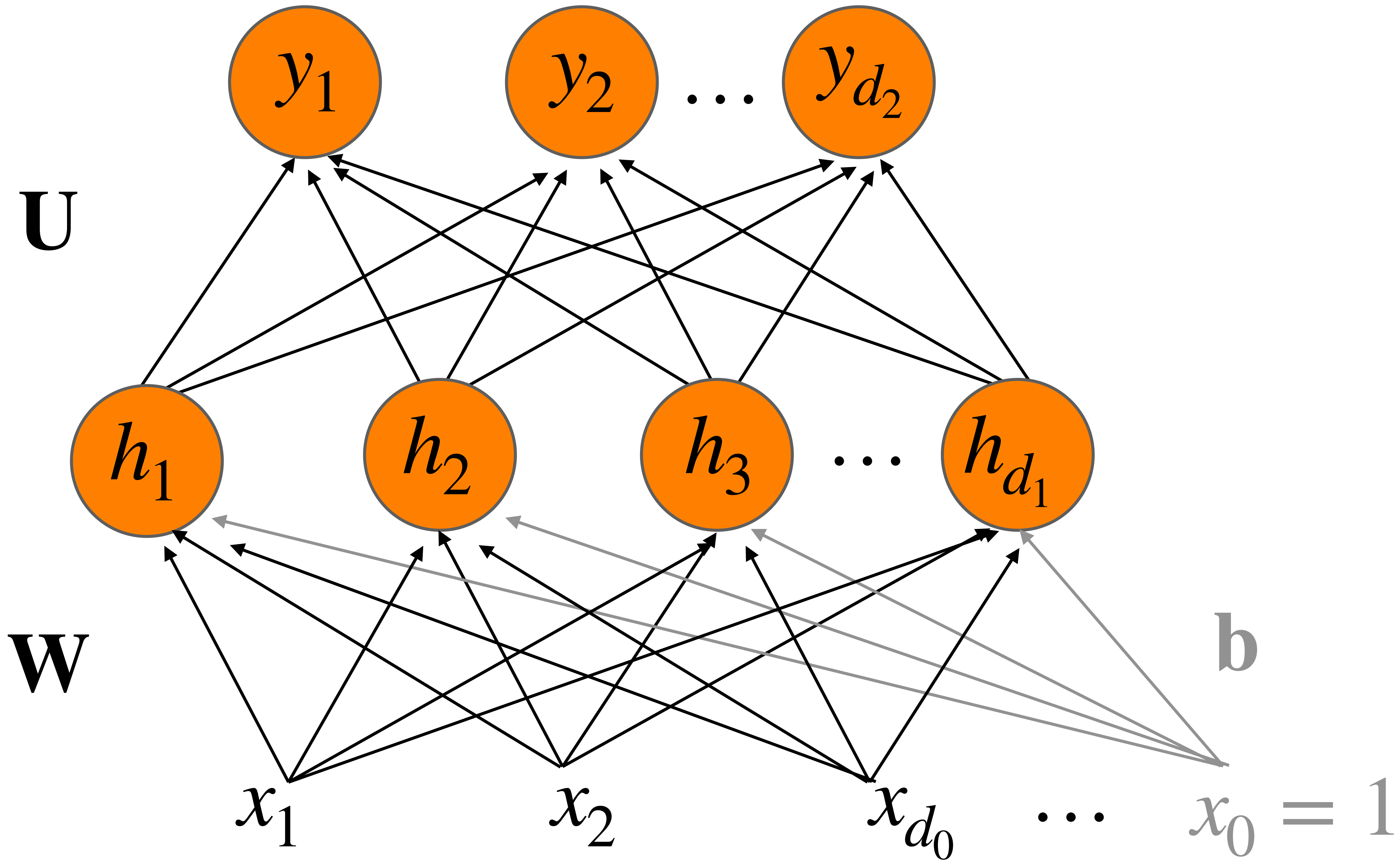
Two-layer Feedforward Network with Softmax Output

Output layer: $\mathbf{y} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x} + \mathbf{b})$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$



What is $\mathbf{y}$?

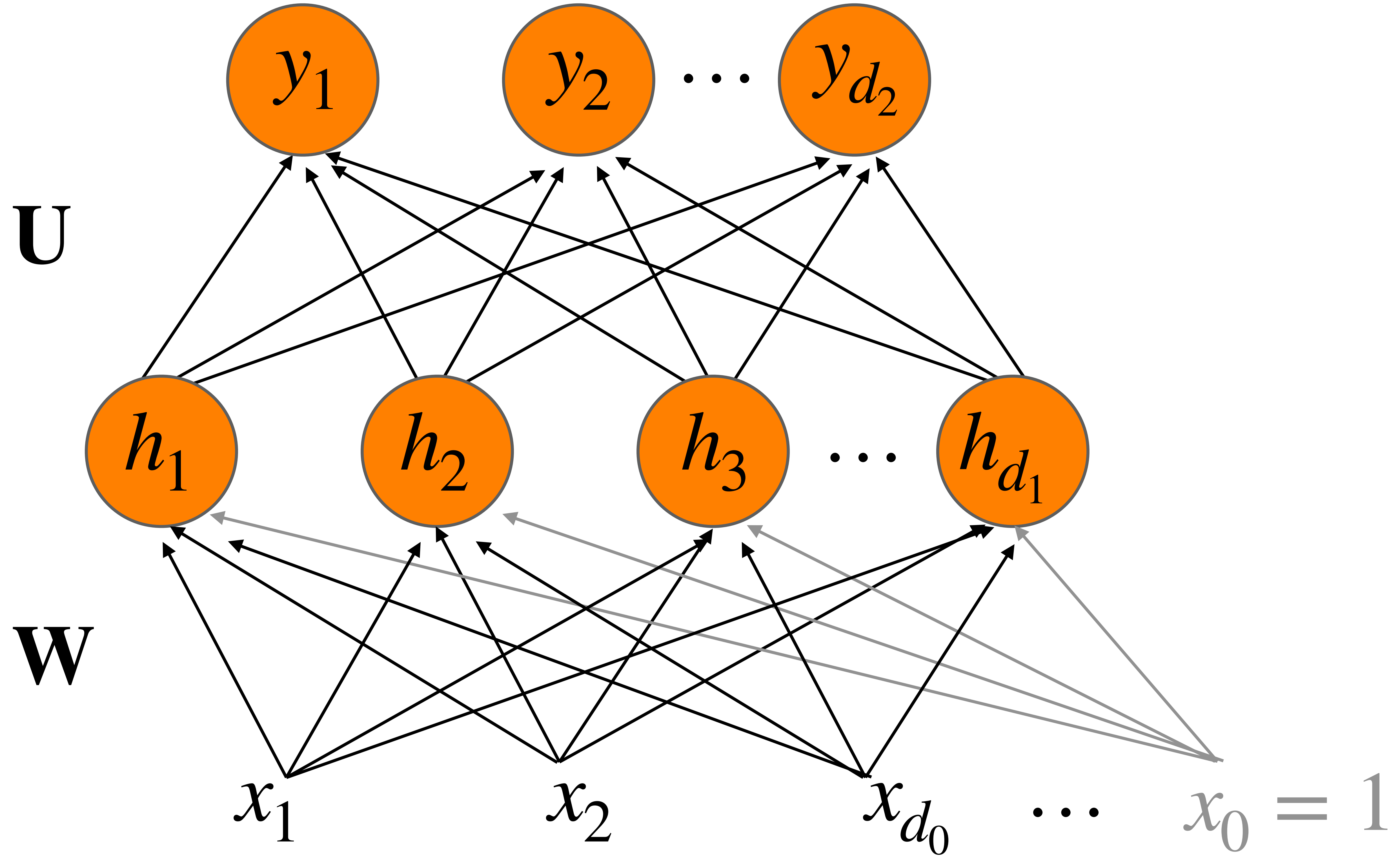
Two-layer FFNN: Notation

Output layer: $\mathbf{y} = \mathrm{softmax}(\mathbf{U} \cdot \mathbf{h})$

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x}) = g\left(\sum_{i=0}^{d_0} \mathbf{W}_{ji}\mathbf{x}_i\right)$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$



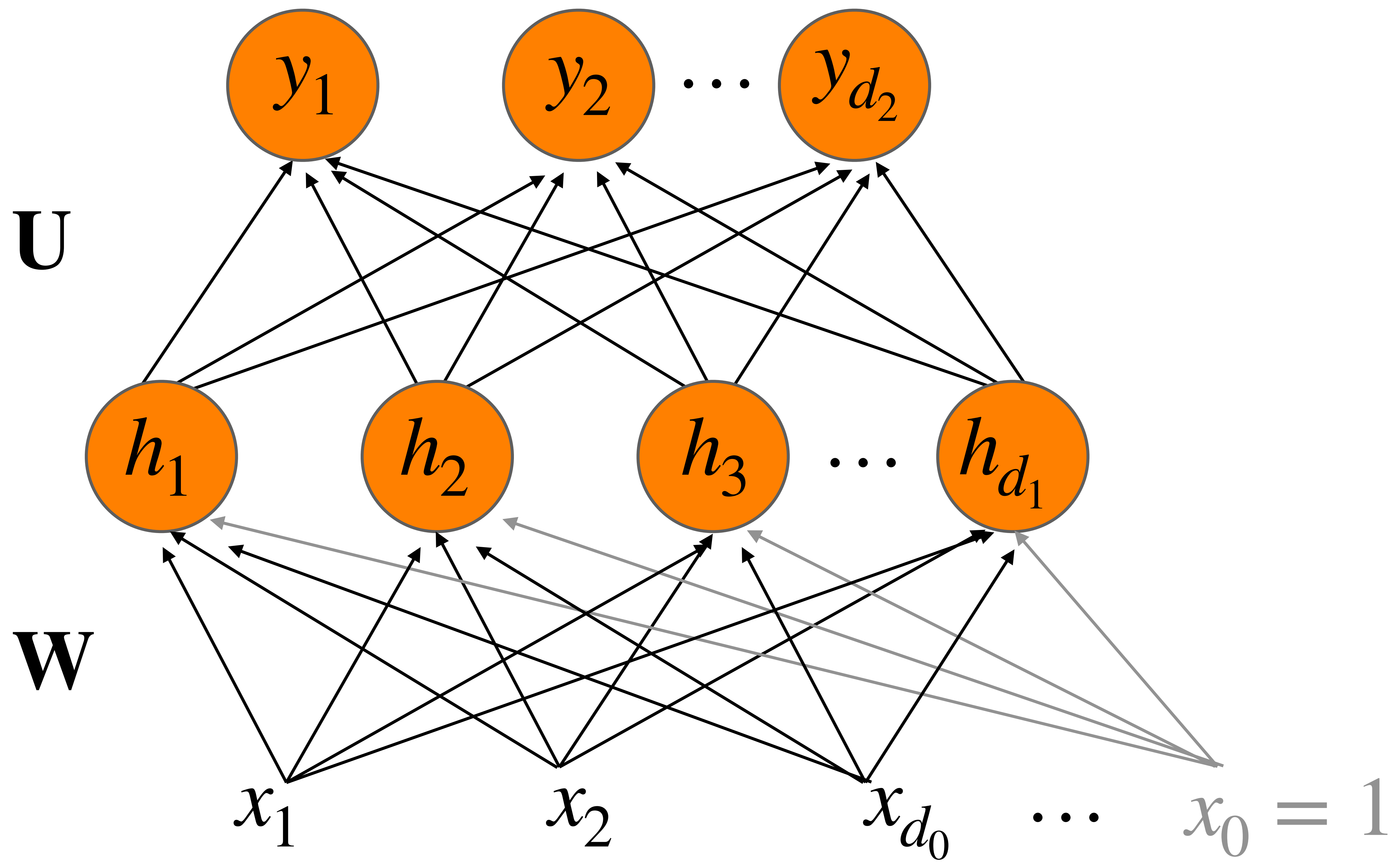
Two-layer FFNN: Notation

Output layer: $\mathbf{y} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$

Hidden layer: $\mathbf{h} = g(\mathbf{W}\mathbf{x}) = g\left(\sum_{i=0}^{d_0} \mathbf{W}_{ji}\mathbf{x}_i\right)$

Usually ReLU or tanh

Input layer: vector $\mathbf{x}$



We usually drop the $\mathbf{b}$ and add one dimension to the $\mathbf{W}$ matrix

FFNN Language Models

Feedforward Neural Language Models

Feedforward Neural Language Models

- Language Modeling: Calculating the probability of the next word in a sequence given some history.

Feedforward Neural Language Models

- Language Modeling: Calculating the probability of the next word in a sequence given some history.
- Compared to n-gram language models, neural network LMs achieve much higher performance

Why?

Feedforward Neural Language Models

- Language Modeling: Calculating the probability of the next word in a sequence given some history.
- Compared to n-gram language models, neural network LMs achieve much higher performance

Why?

Can neural LMs overcome the overfitting problem in n-gram LMs?

Feedforward Neural Language Models

- Language Modeling: Calculating the probability of the next word in a sequence given some history.
- Compared to n-gram language models, neural network LMs achieve much higher performance
  - In general, count-based methods can never do as well as optimization-based ones

Why?

Can neural LMs overcome the overfitting problem in n-gram LMs?

Feedforward Neural Language Models

- Language Modeling: Calculating the probability of the next word in a sequence given some history.
- Compared to n-gram language models, neural network LMs achieve much higher performance
  - In general, count-based methods can never do as well as optimization-based ones
- State-of-the-art neural LMs are based on more powerful neural network technology like Transformers

Why?

Can neural LMs overcome the overfitting problem in n-gram LMs?

Feedforward Neural Language Models

- Language Modeling: Calculating the probability of the next word in a sequence given some history.
- Compared to n-gram language models, neural network LMs achieve much higher performance
  - In general, count-based methods can never do as well as optimization-based ones
- State-of-the-art neural LMs are based on more powerful neural network technology like Transformers
- But **simple feedforward LMs** can do almost as well!

Why?

Can neural LMs overcome the overfitting problem in n-gram LMs?

Simple Feedforward Neural LMs

First introduced by Yoshua Bengio and colleagues in 2003

Simple Feedforward Neural LMs

Task: predict next word w_t given prior words $w_{t-1}, w_{t-2}, w_{t-3}, \ldots$

First introduced by Yoshua Bengio and colleagues in 2003

Simple Feedforward Neural LMs

Task: predict next word w_t given prior words $w_{t-1}, w_{t-2}, w_{t-3}, \ldots$

Problem: Now we are dealing with sequences of arbitrary length….

First introduced by Yoshua Bengio and colleagues in 2003

Simple Feedforward Neural LMs

Task: predict next word w_t given prior words $w_{t-1}, w_{t-2}, w_{t-3}, \ldots$

Problem: Now we are dealing with sequences of arbitrary length….

Solution: Sliding windows (of fixed length)

First introduced by Yoshua Bengio and colleagues in 2003

Simple Feedforward Neural LMs

Task: predict next word w_t given prior words $w_{t-1}, w_{t-2}, w_{t-3}, \ldots$

Problem: Now we are dealing with sequences of arbitrary length….

Solution: Sliding windows (of fixed length)

$$P(w_t | w_{t-1}) \approx P(w_t | w_{t-1:t-M+1})$$

First introduced by Yoshua Bengio and colleagues in 2003

Simple Feedforward Neural LMs

Task: predict next word w_t given prior words $w_{t-1}, w_{t-2}, w_{t-3}, \ldots$

Problem: Now we are dealing with sequences of arbitrary length….

Solution: Sliding windows (of fixed length)

Basis of word embedding models!

$$P(w_t \mid w_{t-1}) \approx P(w_t \mid w_{t-1:t-M+1})$$

First introduced by Yoshua Bengio and colleagues in 2003

Data: Feedforward Language Model

Data: Feedforward Language Model

- Self-supervised

Data: Feedforward Language Model

- Self-supervised
- Computation is divided into time steps t, where different sliding windows are considered

Data: Feedforward Language Model

- Self-supervised
- Computation is divided into time steps t, where different sliding windows are considered
- $x_t = (w_{t-1}, \ldots, w_{t-M+1})$ for the context

 USC Viterbi

Data: Feedforward Language Model

- Self-supervised
- Computation is divided into time steps t, where different sliding windows are considered
- $x_t = (w_{t-1}, \ldots, w_{t-M+1})$ for the context
    - represent words in this prior context by their embeddings, rather than just by their word identity as in n-gram LMs

Data: Feedforward Language Model

- Self-supervised
- Computation is divided into time steps t, where different sliding windows are considered
- $x_t = (w_{t-1}, \ldots, w_{t-M+1})$ for the context
    - represent words in this prior context by their embeddings, rather than just by their word identity as in n-gram LMs
    - allows neural LMs to generalize better to unseen data / similar data

Data: Feedforward Language Model

- Self-supervised
- Computation is divided into time steps t, where different sliding windows are considered
- $x_t = (w_{t-1}, \ldots, w_{t-M+1})$ for the context
    - represent words in this prior context by their embeddings, rather than just by their word identity as in n-gram LMs
    - allows neural LMs to generalize better to unseen data / similar data
    - All embeddings in the context are concatenated

Data: Feedforward Language Model

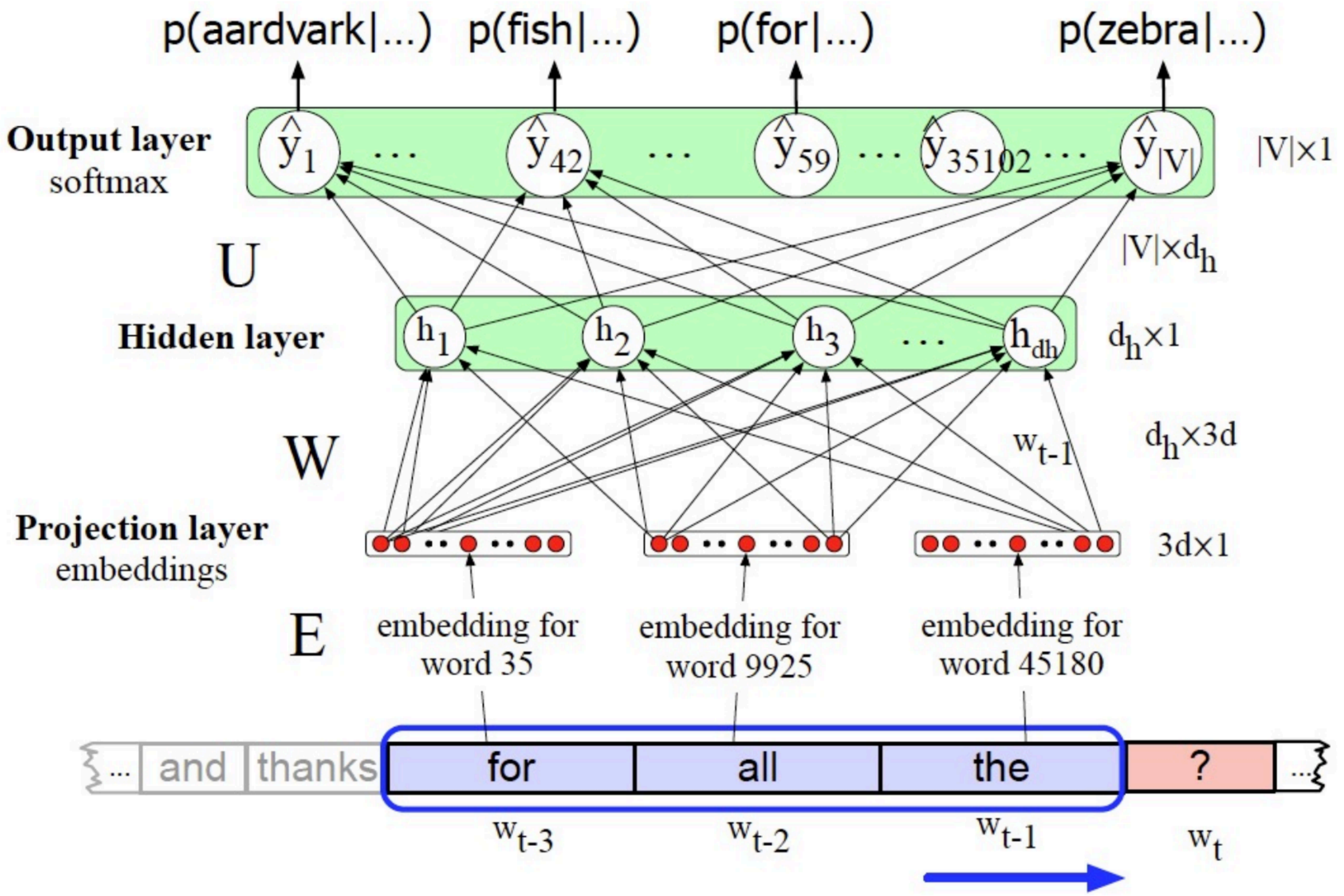
- Self-supervised
- Computation is divided into time steps t, where different sliding windows are considered
- $x_t = (w_{t-1}, \ldots, w_{t-M+1})$ for the context
  - represent words in this prior context by their embeddings, rather than just by their word identity as in n-gram LMs
  - allows neural LMs to generalize better to unseen data / similar data
  - All embeddings in the context are concatenated
- $y_t = w_t$ for the next word

Data: Feedforward Language Model

- Self-supervised
- Computation is divided into time steps t, where different sliding windows are considered
- $x_t = (w_{t-1}, \ldots, w_{t-M+1})$ for the context
  - represent words in this prior context by their embeddings, rather than just by their word identity as in n-gram LMs
  - allows neural LMs to generalize better to unseen data / similar data
  - All embeddings in the context are concatenated
- $y_t = w_t$ for the next word
  - Represented as a one hot vector of vocabulary size where only the ground truth gets a value of 1 and every other element is a 0

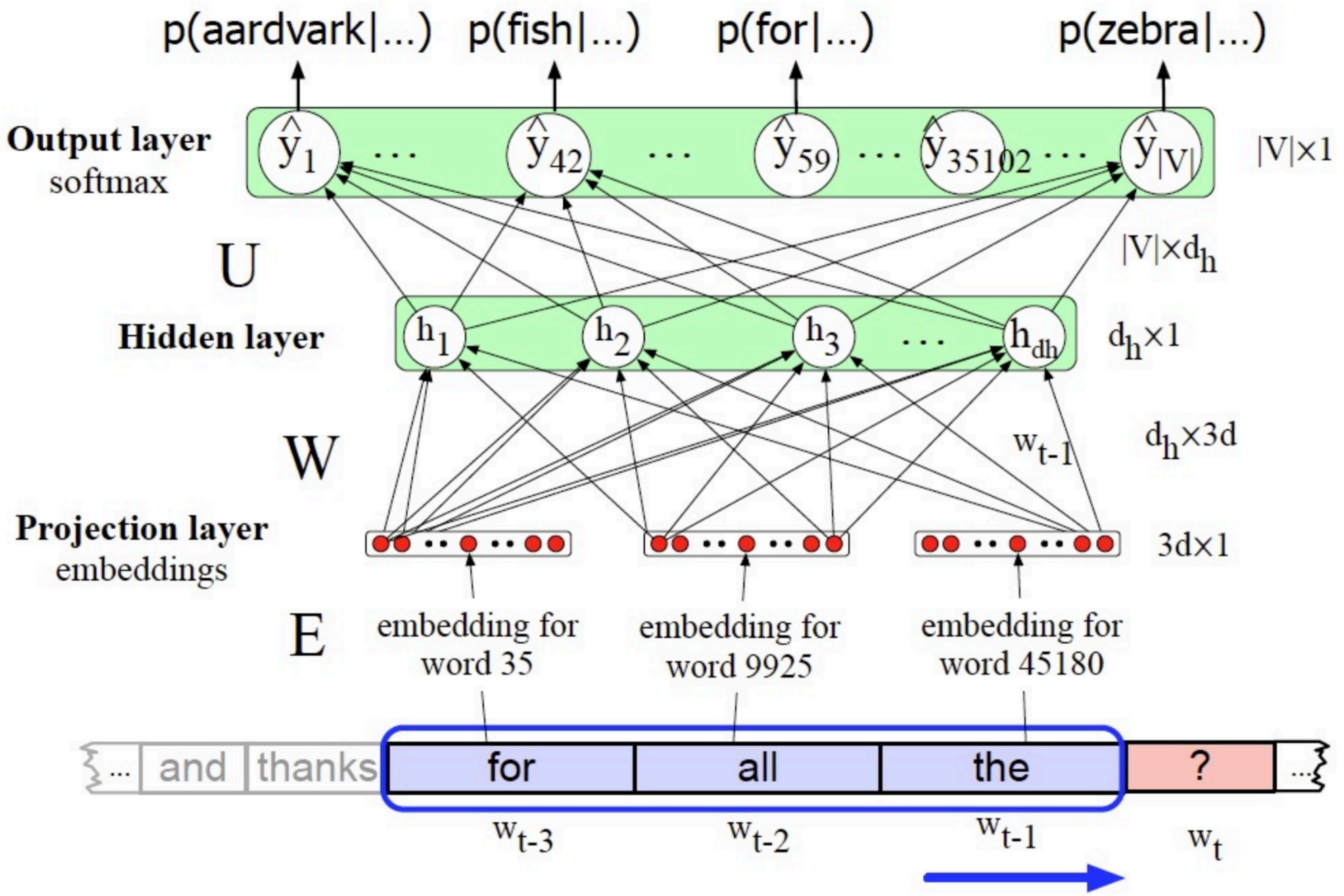
One-hot vector

Feedforward Neural LM



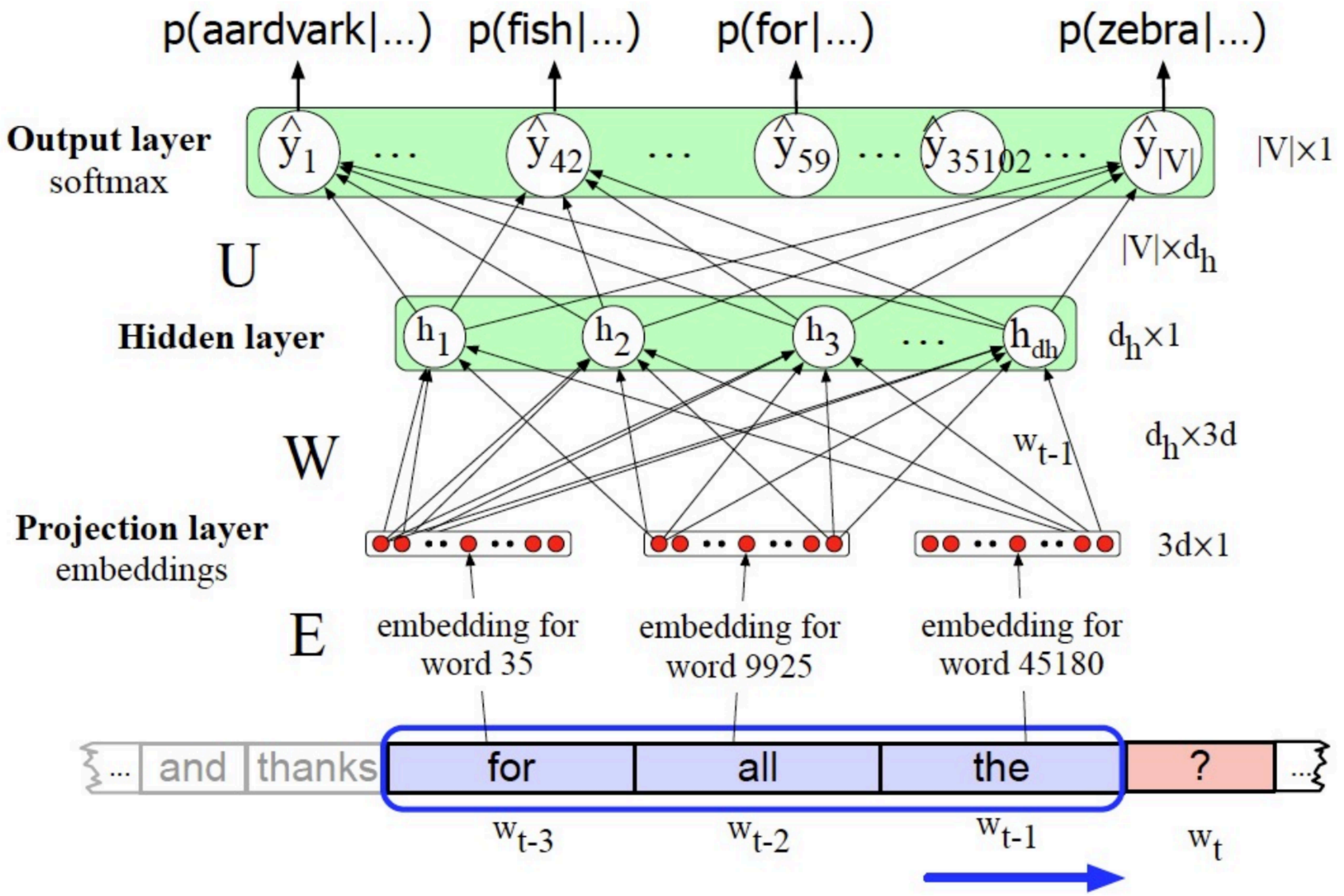
Feedforward Neural LM

- Sliding window of size 4 (including the target word)



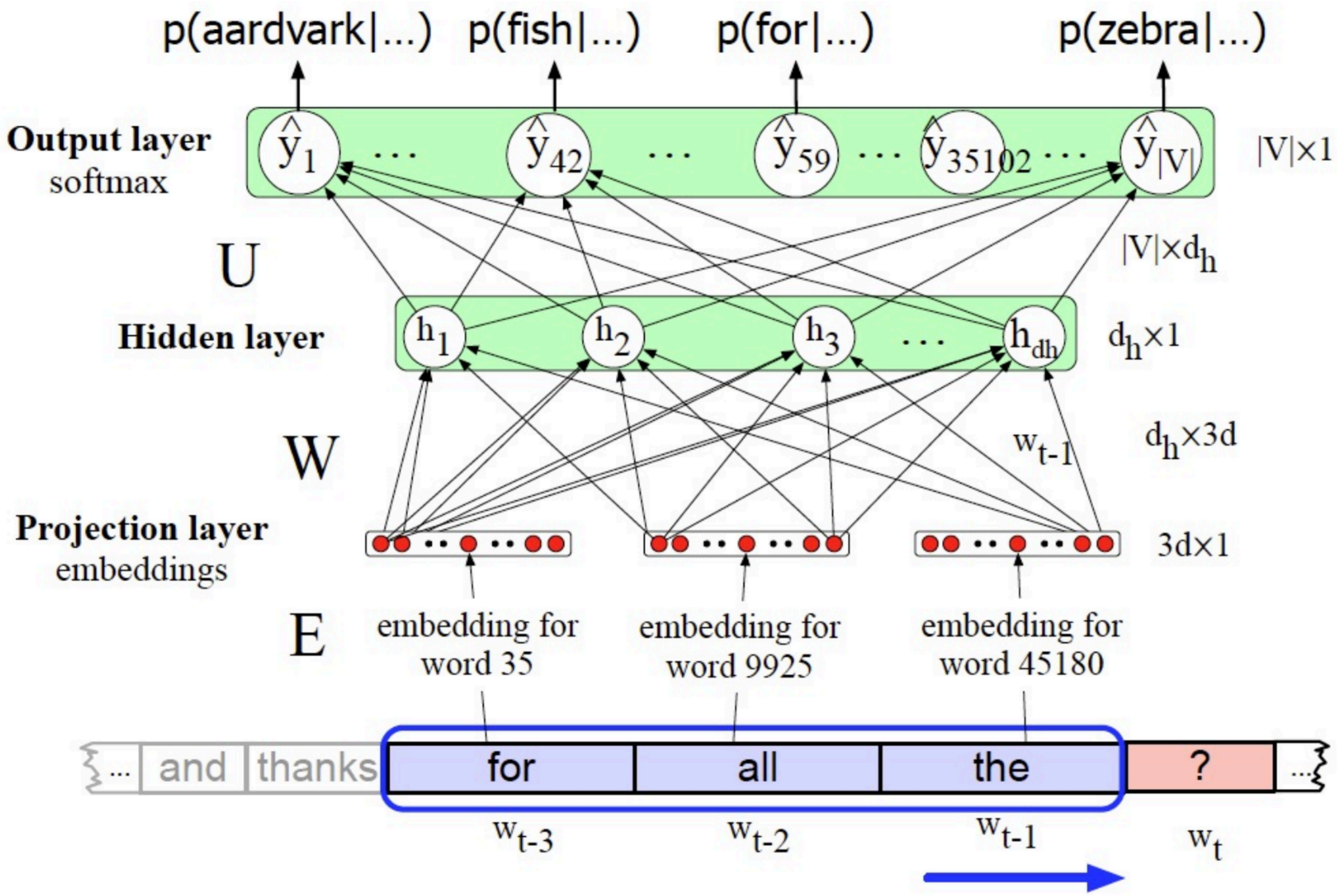
Feedforward Neural LM

- Sliding window of size 4 (including the target word)
- Every feature in the embedding vector connected to every single hidden unit



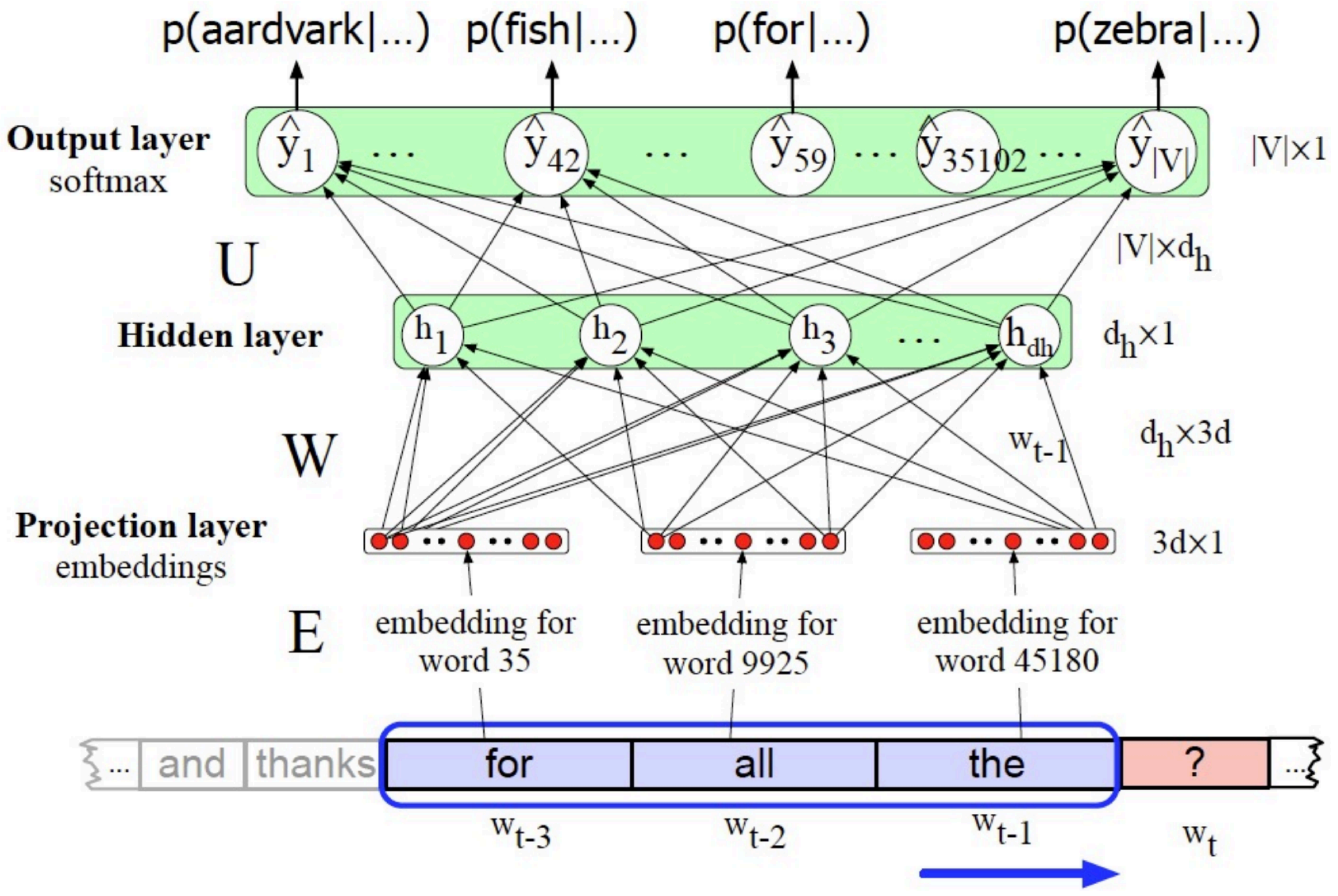
Feedforward Neural LM

- Sliding window of size 4 (including the target word)
- Every feature in the embedding vector connected to every single hidden unit
- Projection / embedding layer is a kind of input layer



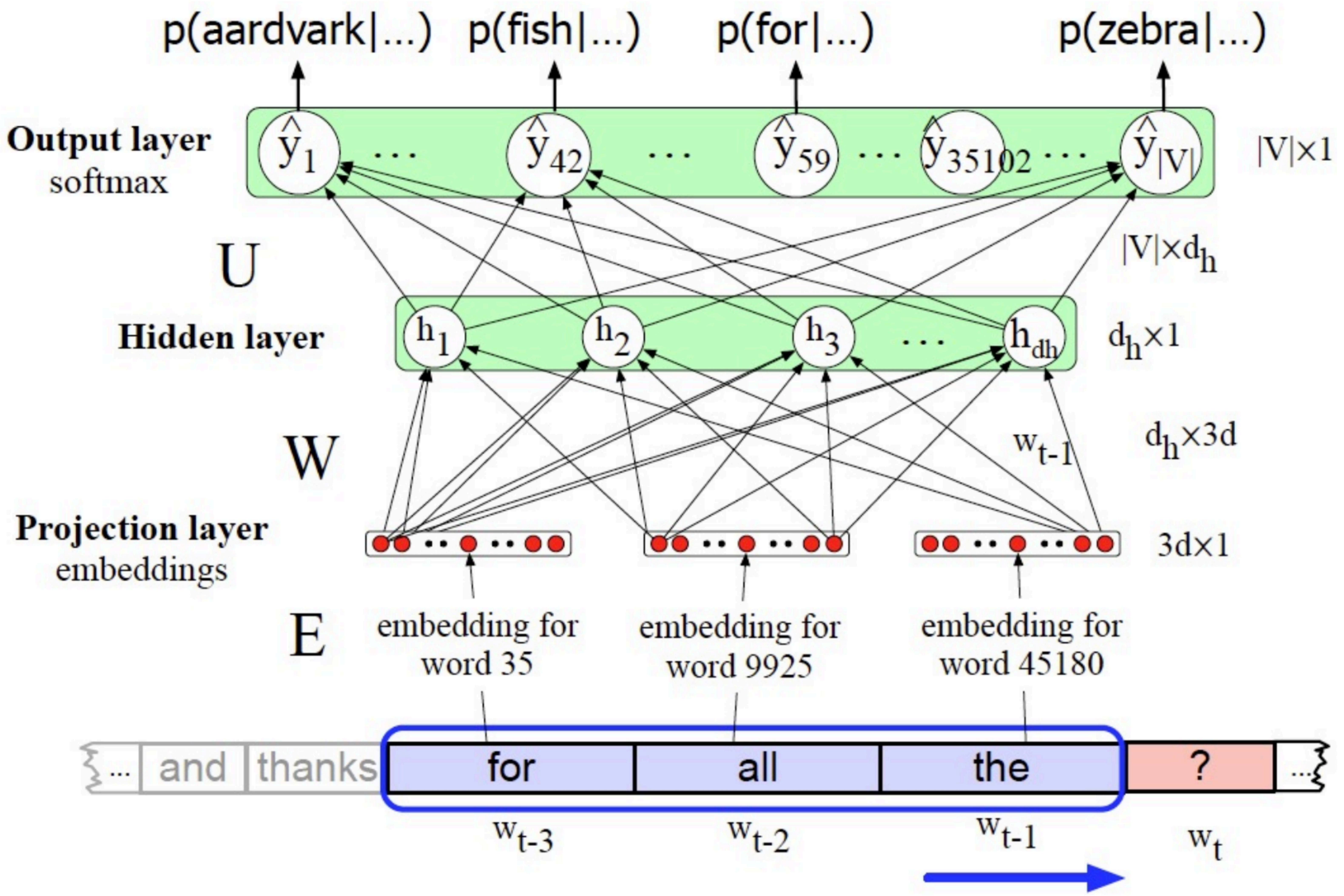
Feedforward Neural LM

- Sliding window of size 4 (including the target word)
- Every feature in the embedding vector connected to every single hidden unit
- Projection / embedding layer is a kind of input layer
    - This is where we plug in our word2vec embeddings

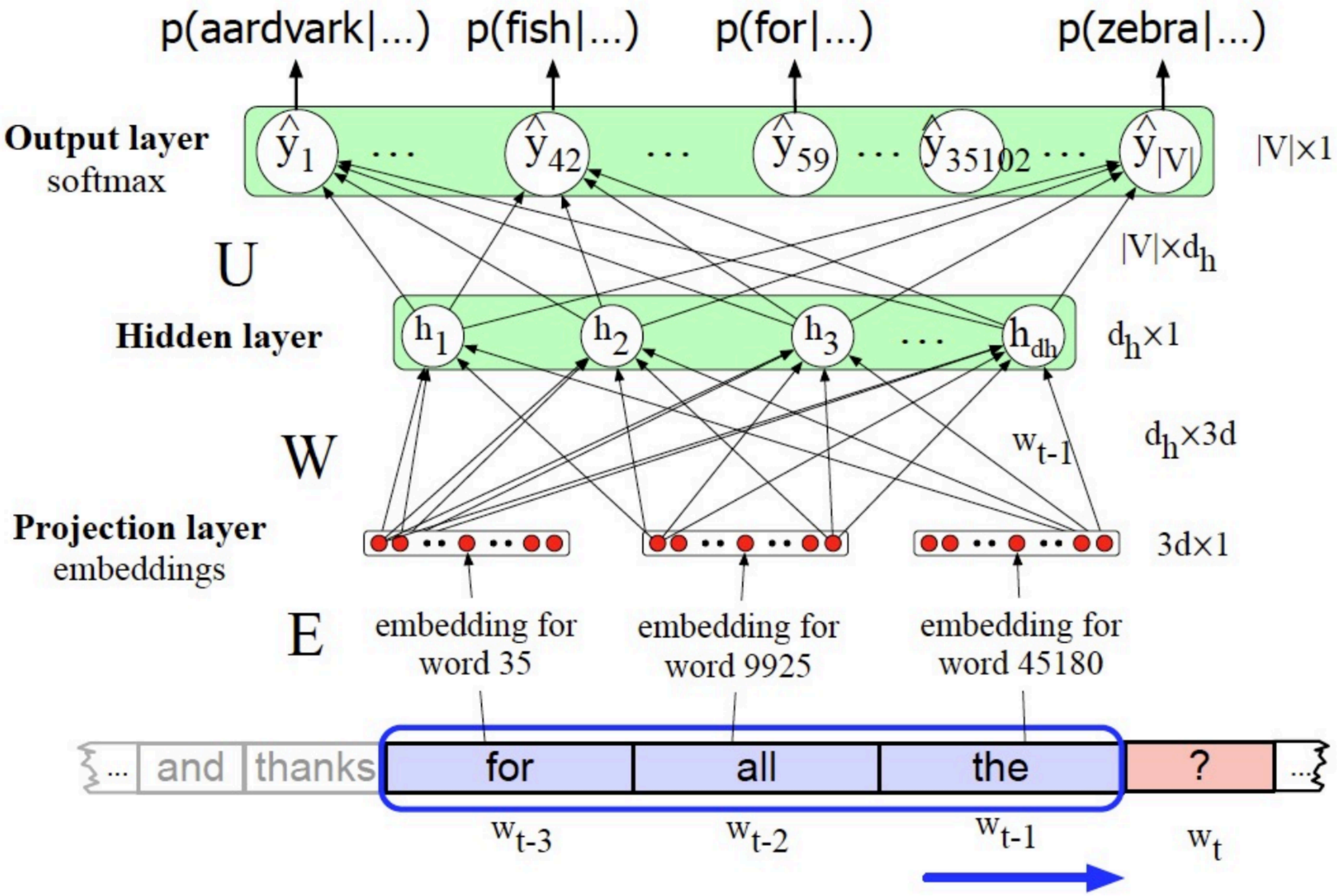


Feedforward Neural LM

- Sliding window of size 4 (including the target word)
- Every feature in the embedding vector connected to every single hidden unit
- Projection / embedding layer is a kind of input layer
  - This is where we plug in our word2vec embeddings
  - May or may not update embedding weights



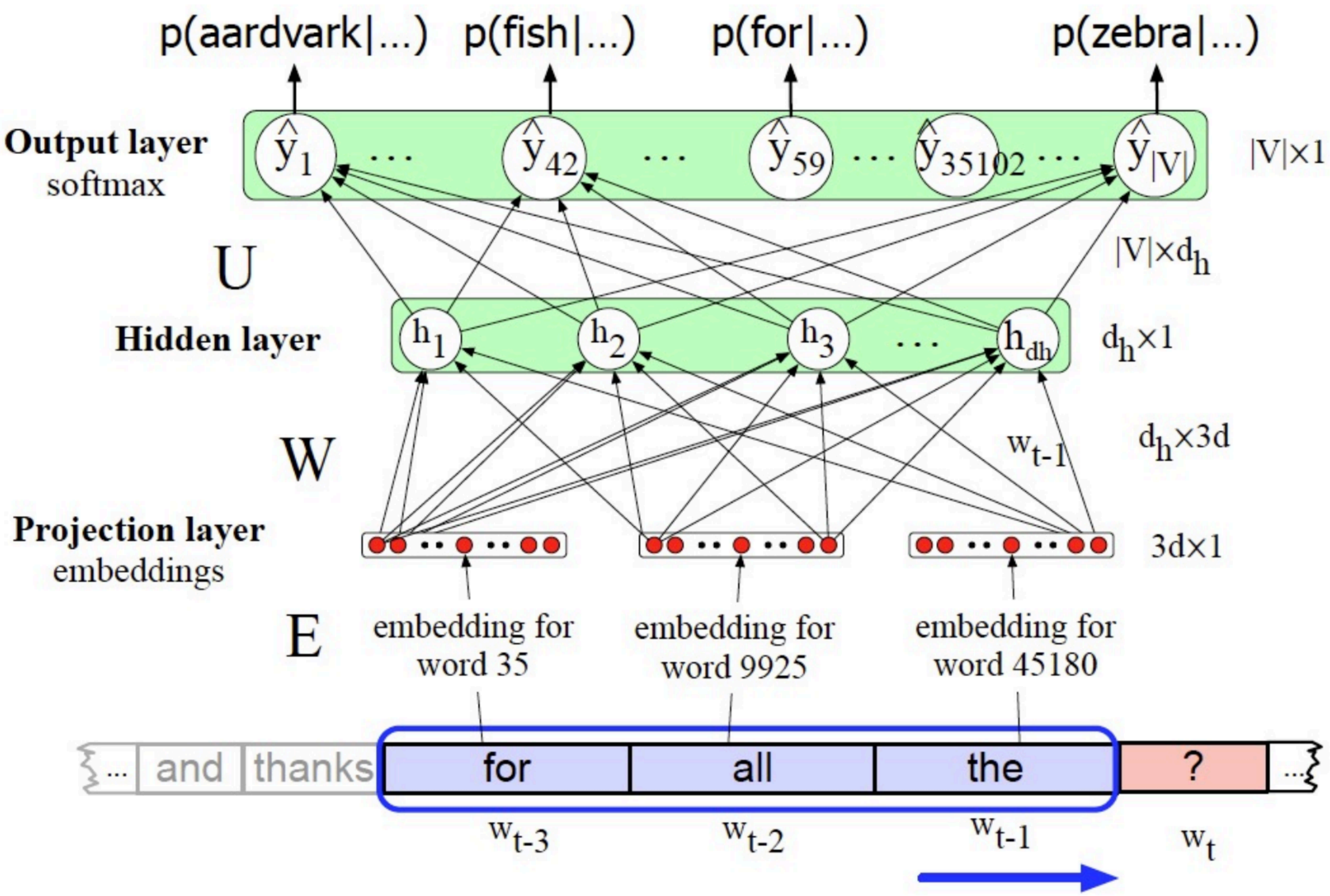
Simplified Representation

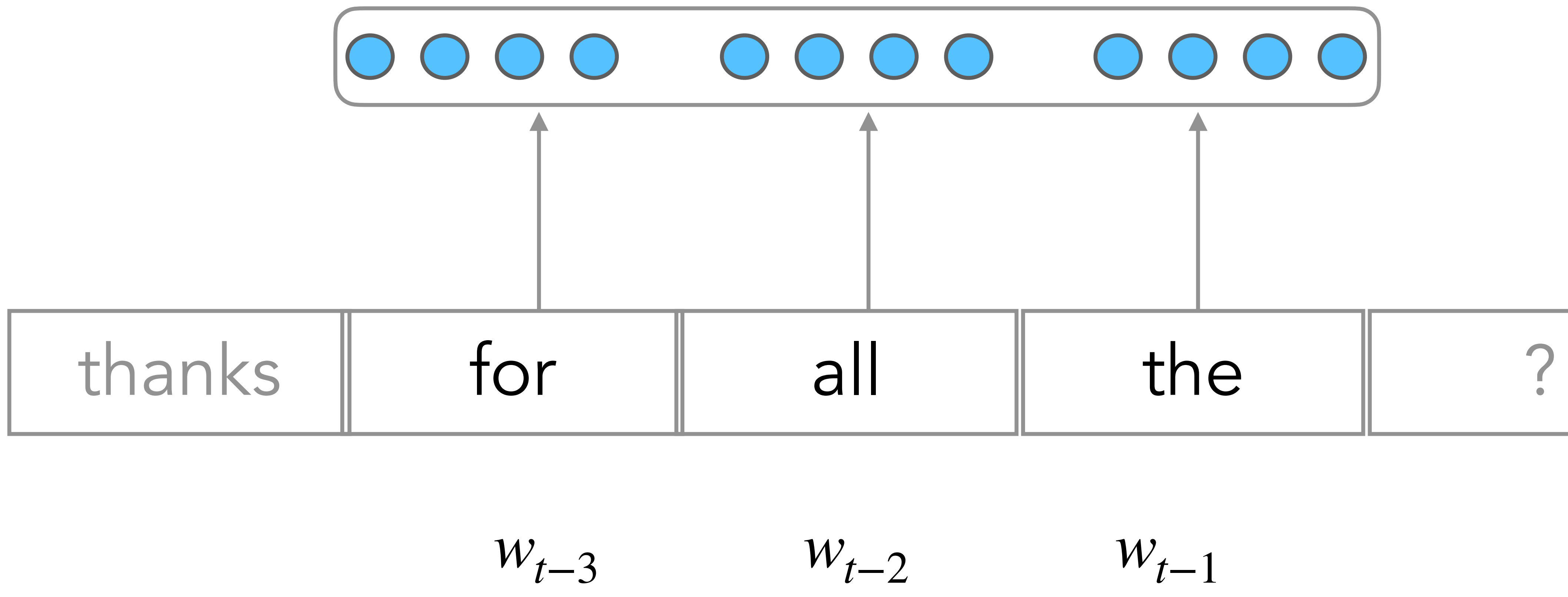


thanks	for	all	the	?
	w_{t-3}	w_{t-2}	w_{t-1}	

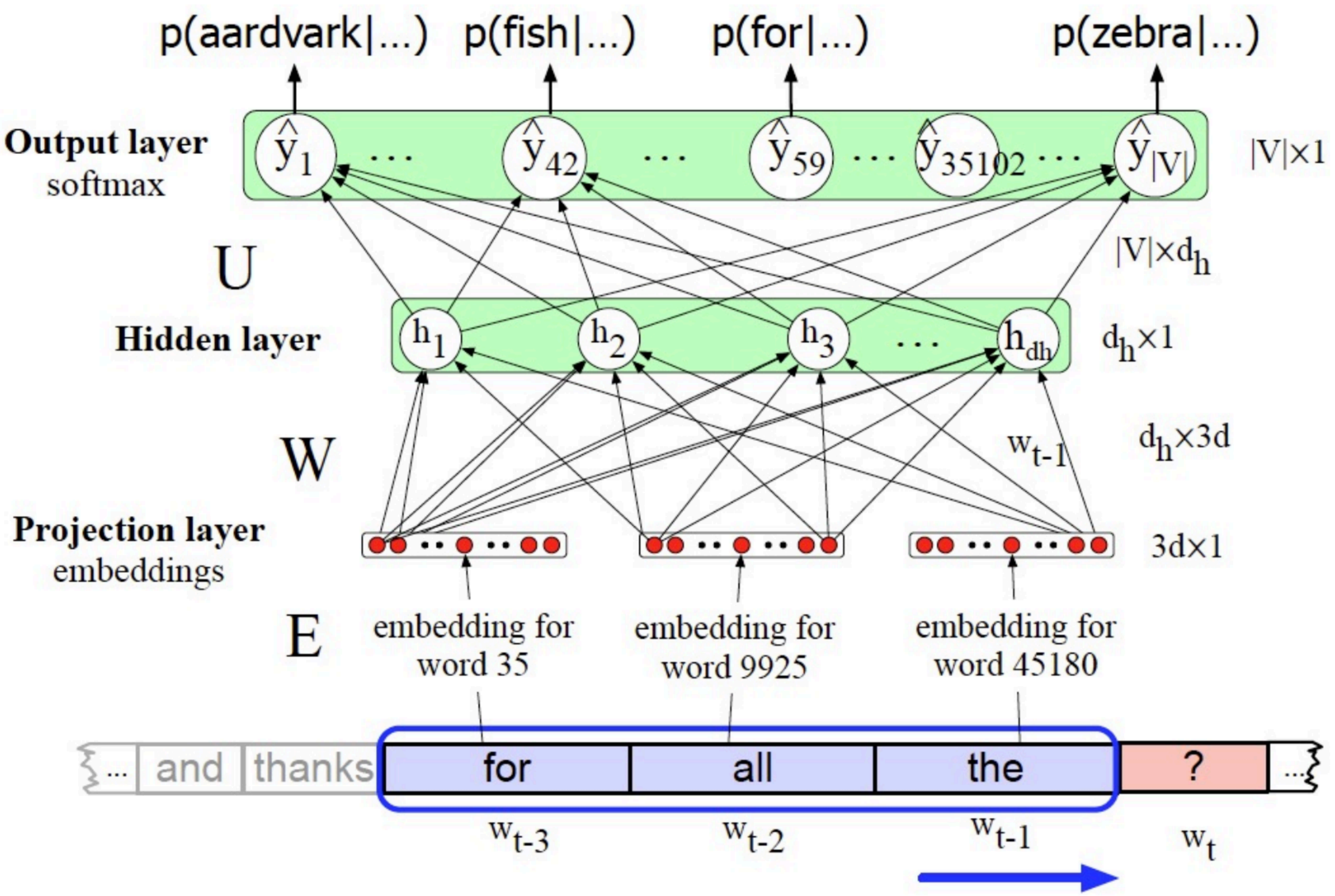
USC Viterbi

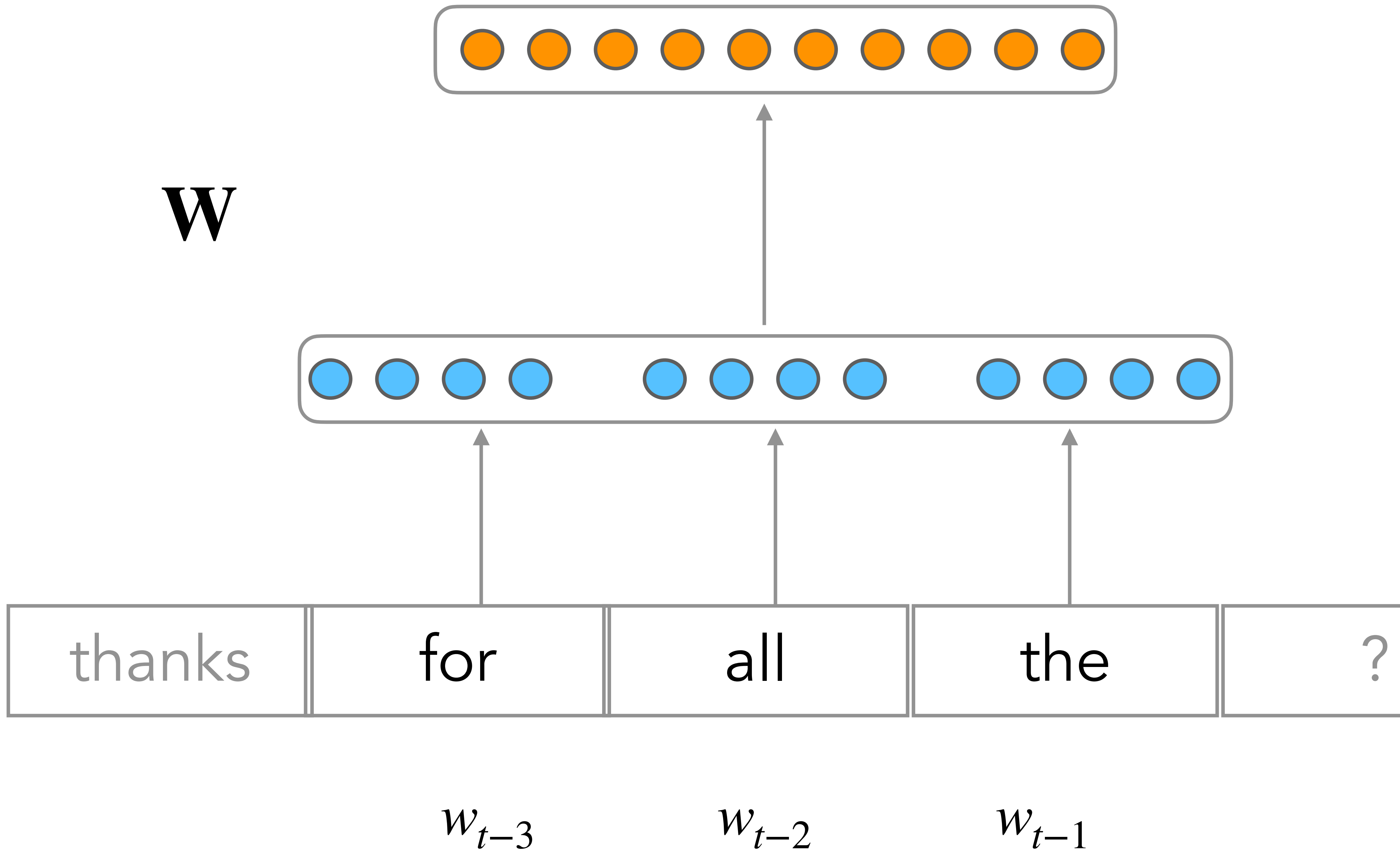
Simplified Representation



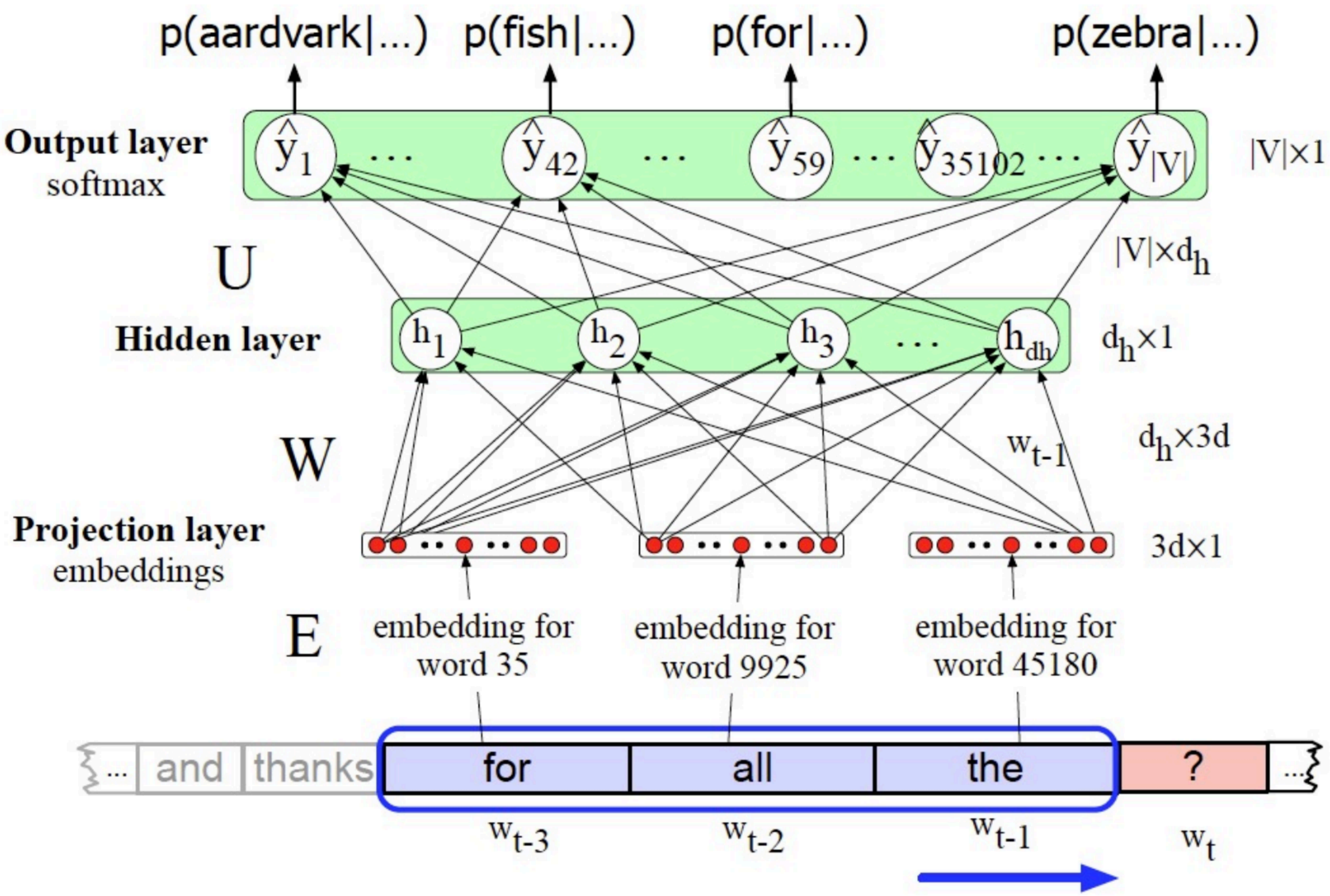


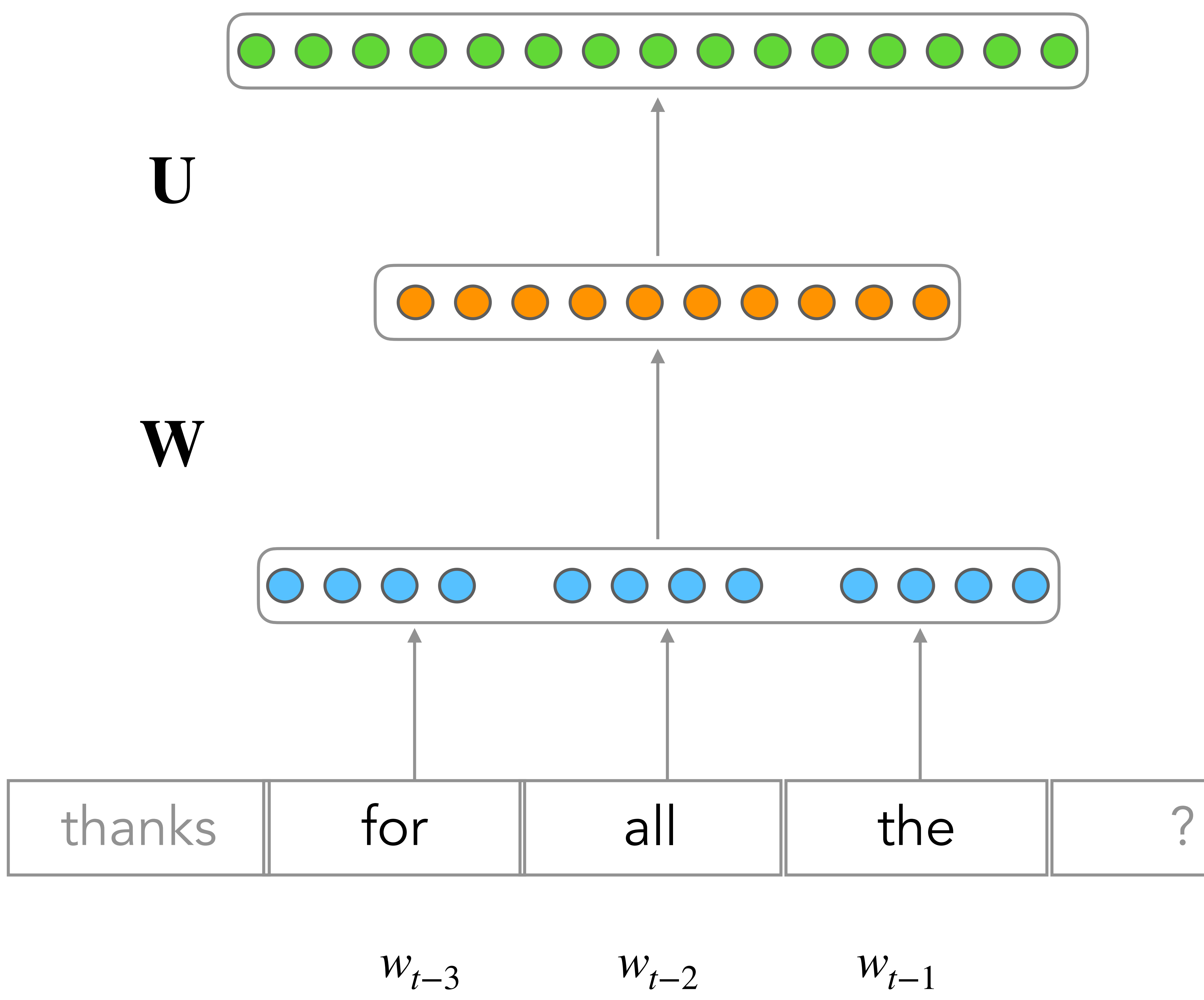
Simplified Representation



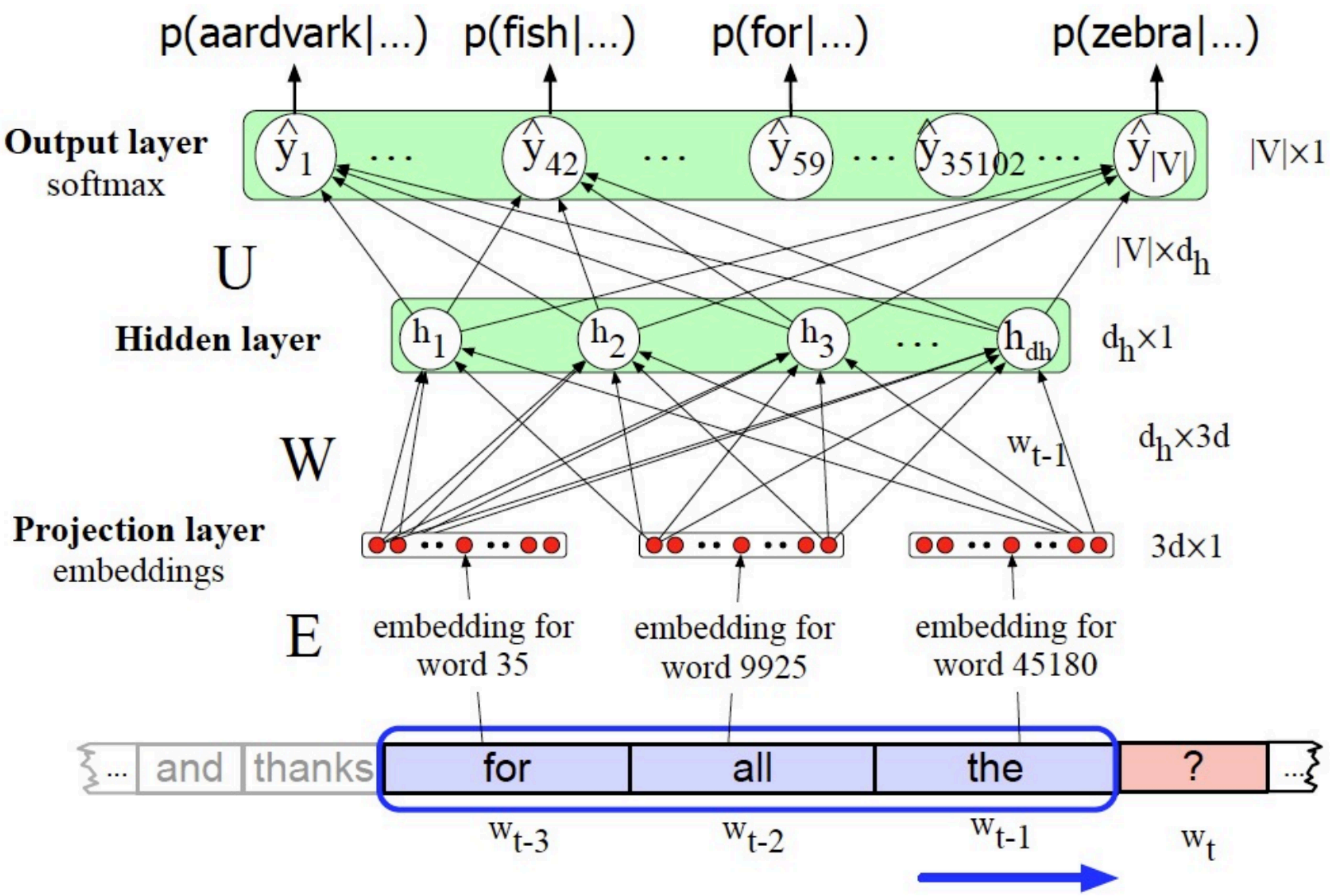


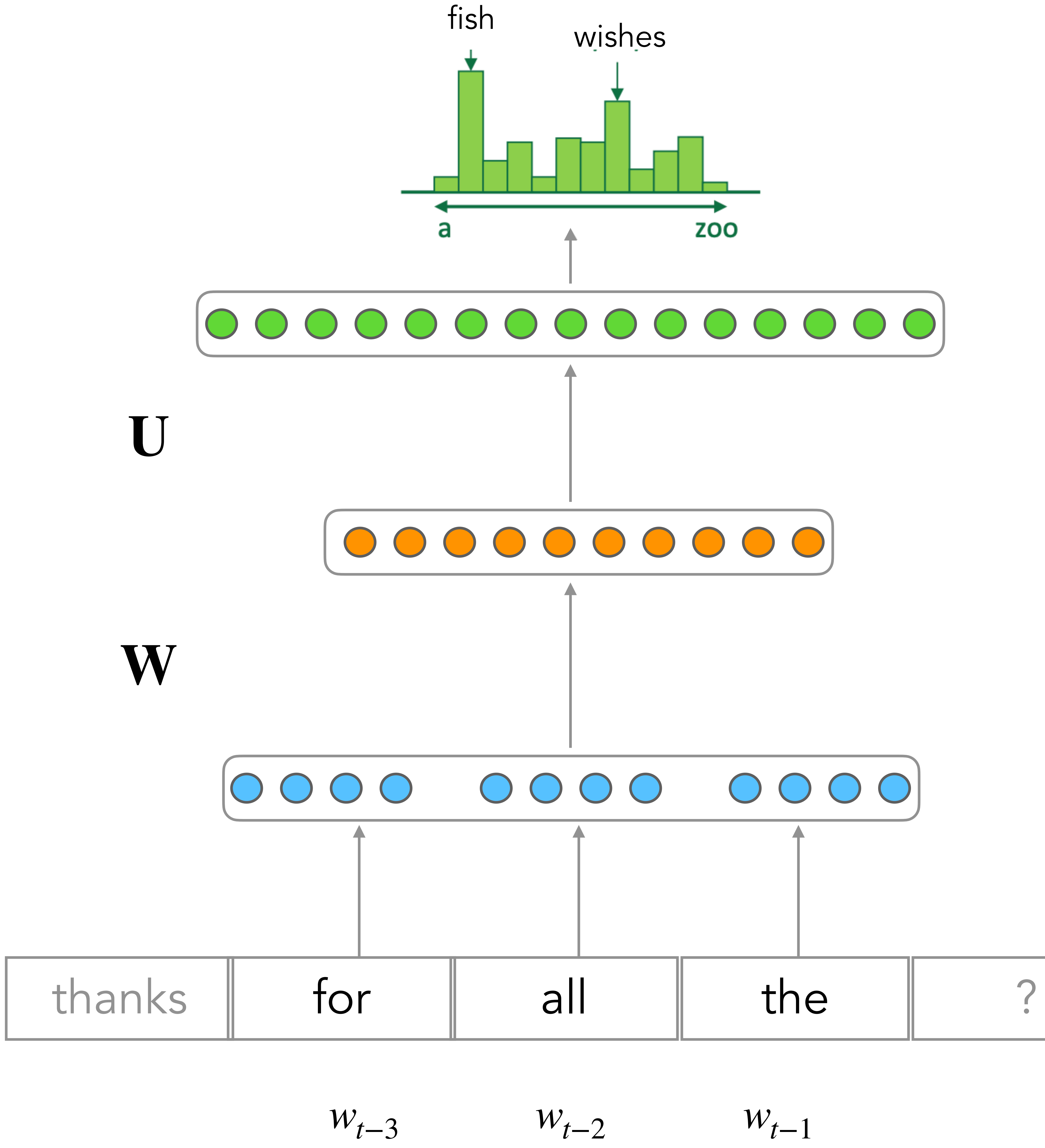
Simplified Representation



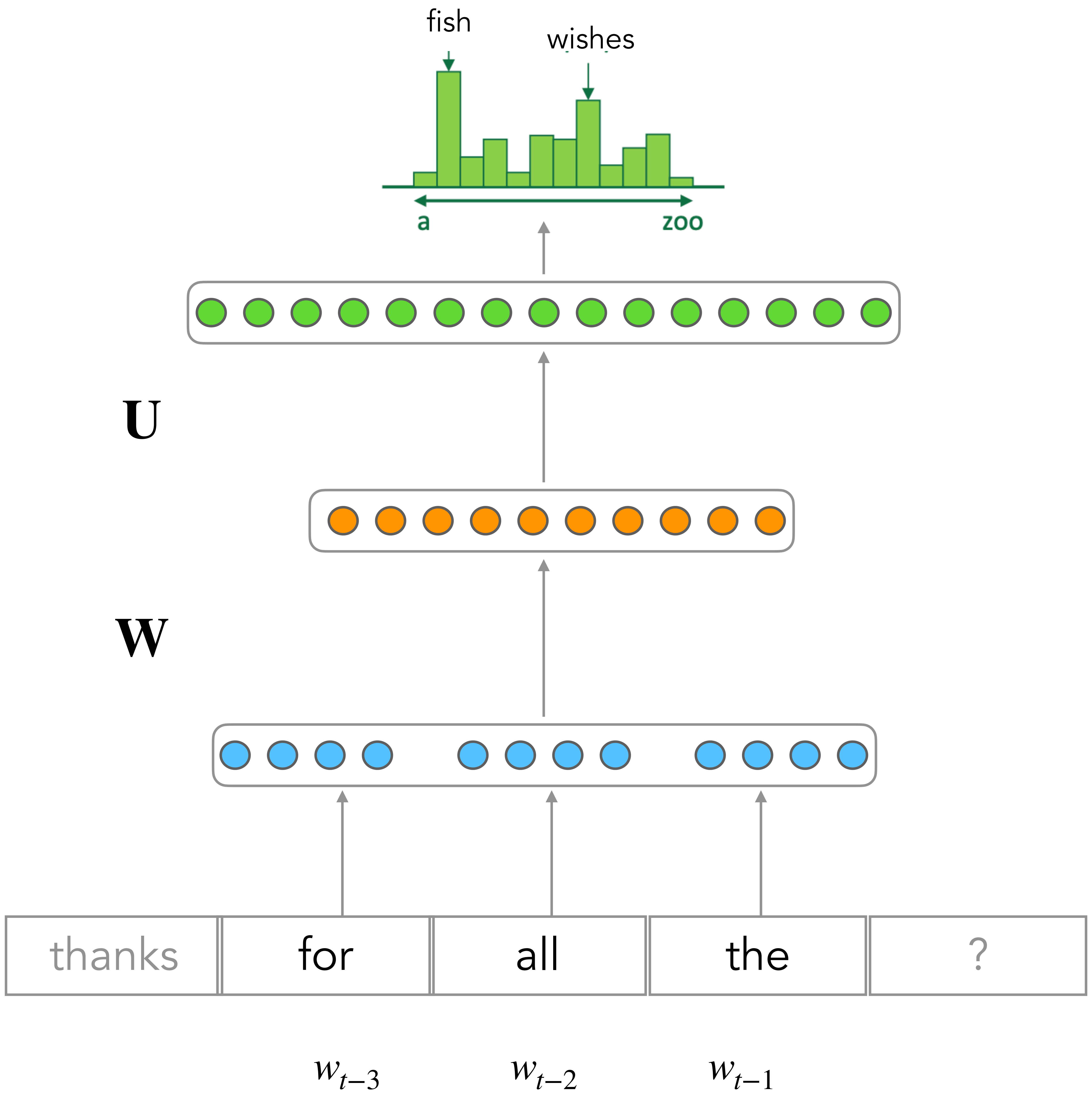


Simplified Representation



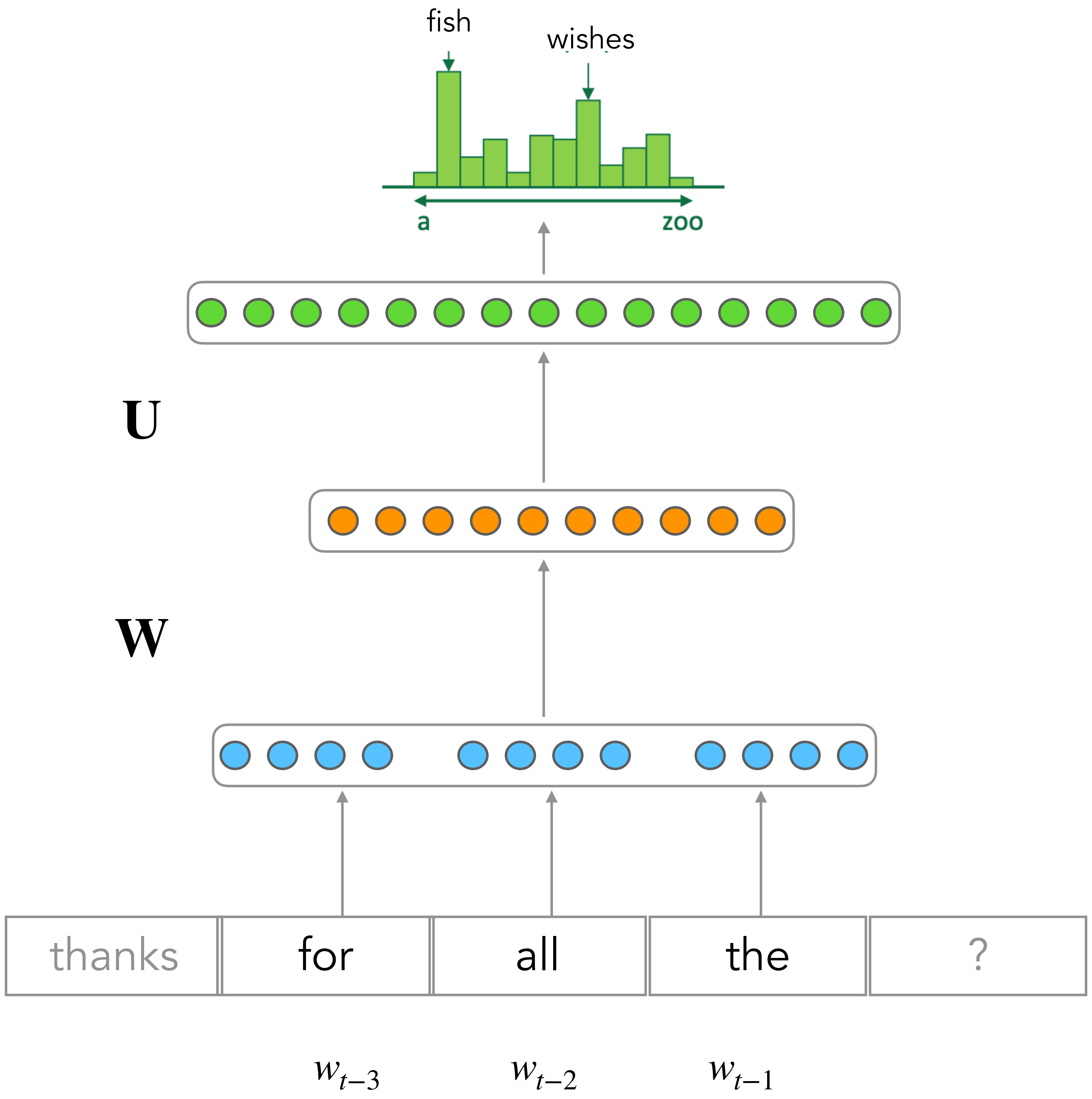


Feedforward LMs: Windows



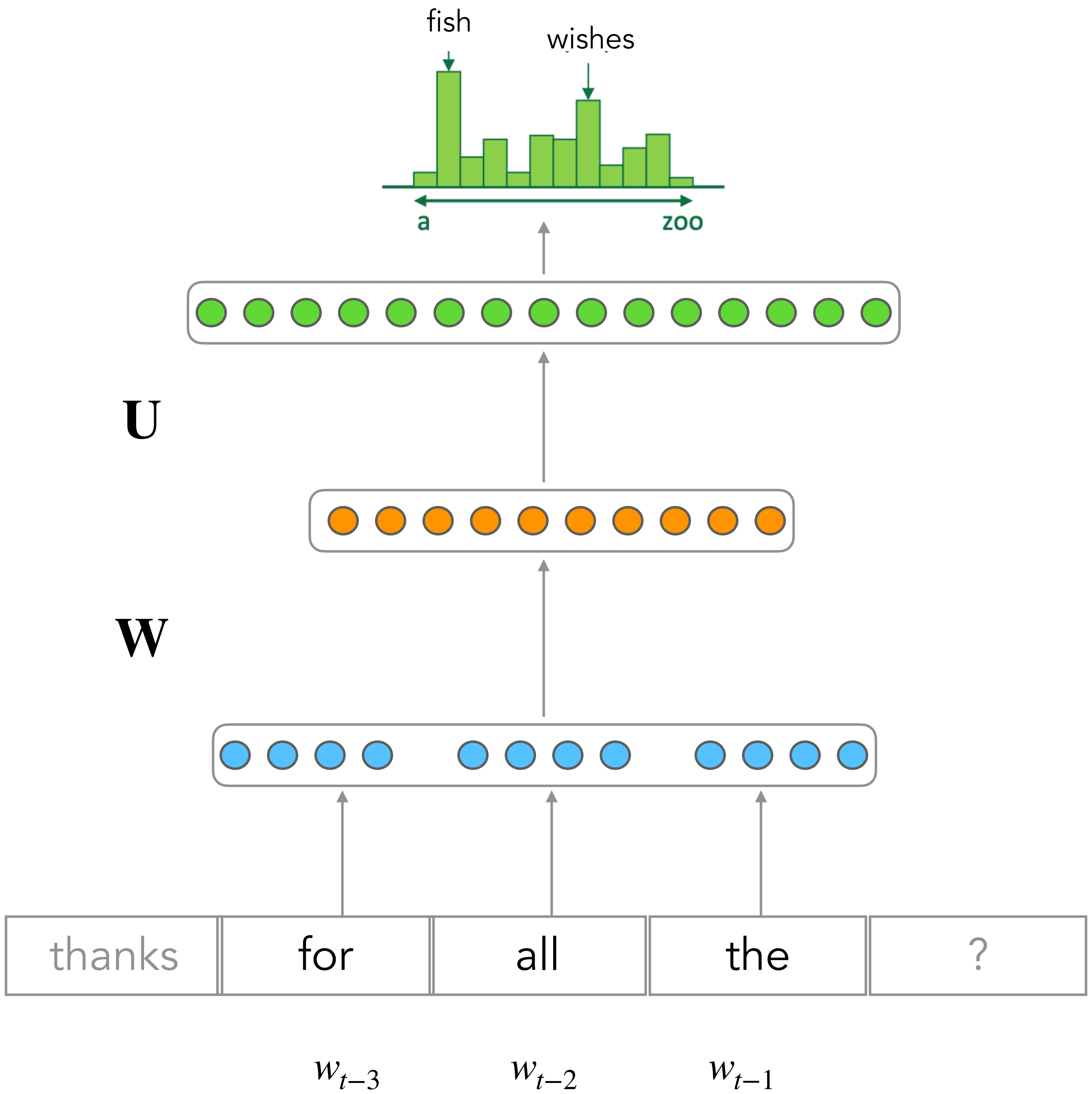
Feedforward LMs: Windows

- The goodness of the language model depends on the size of the sliding window!



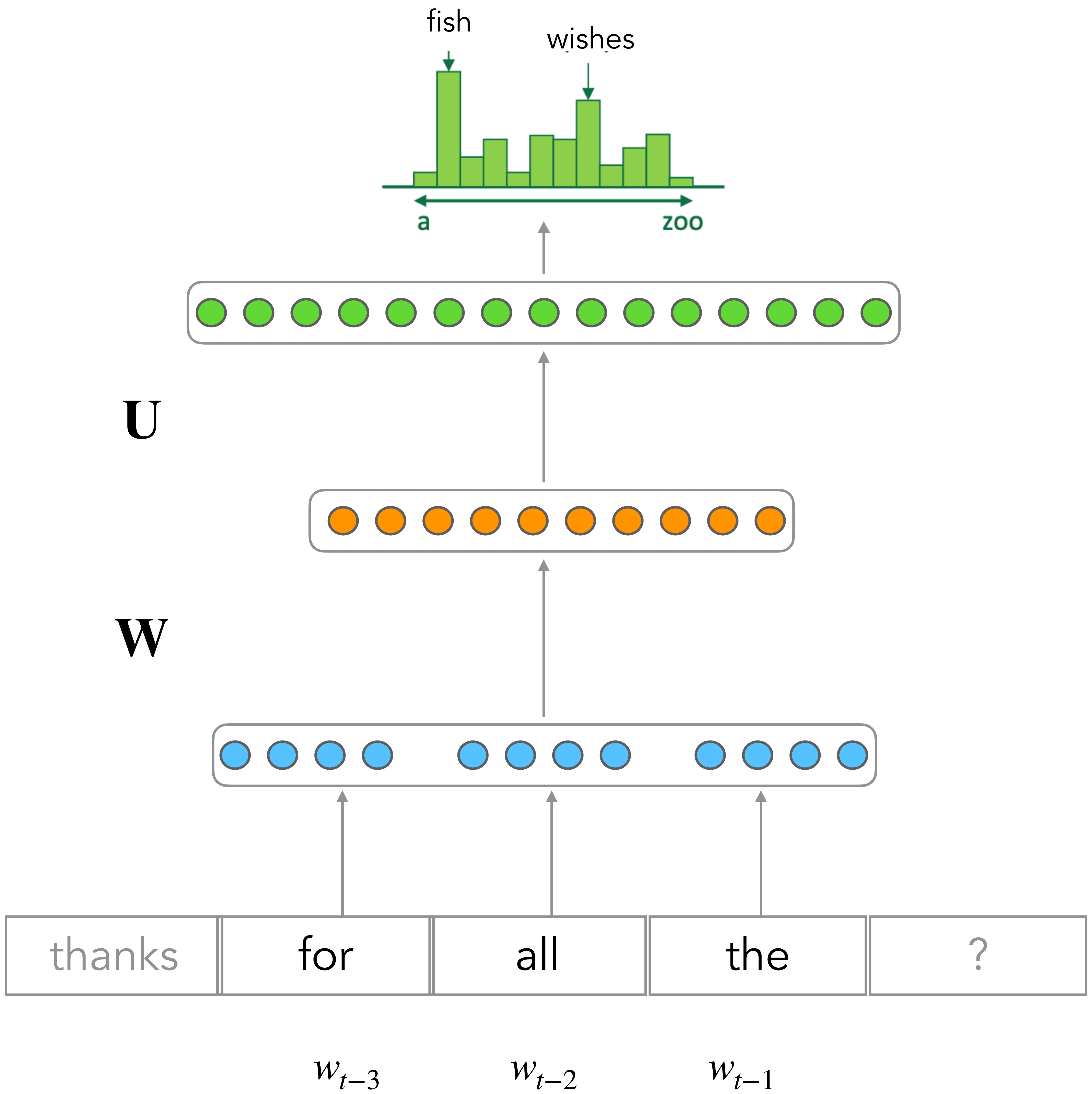
Feedforward LMs: Windows

- The goodness of the language model depends on the size of the sliding window!
- Fixed window can be too small



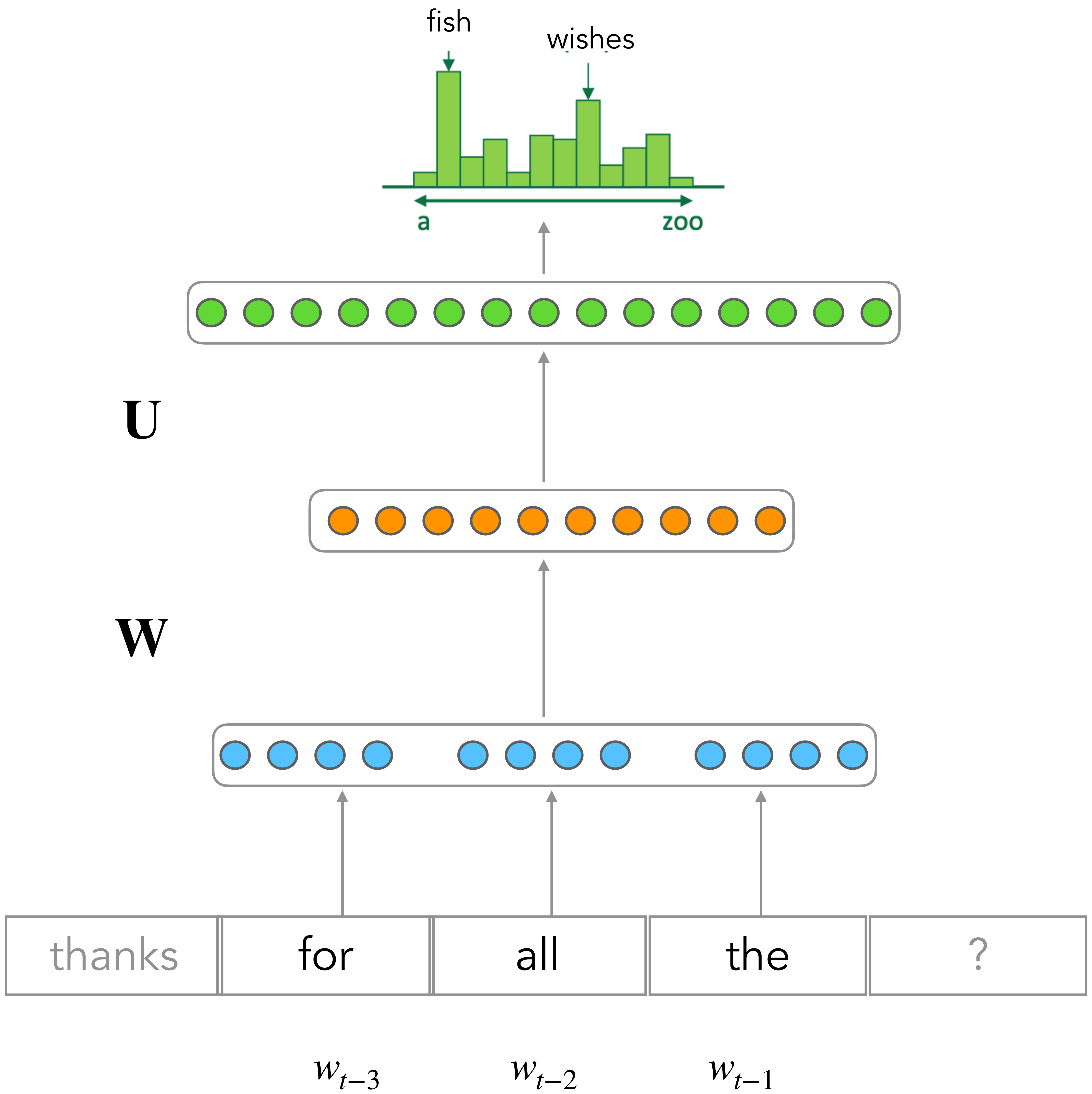
Feedforward LMs: Windows

- The goodness of the language model depends on the size of the sliding window!
- Fixed window can be too small
- Enlarging window enlarges $\mathbf{W}$



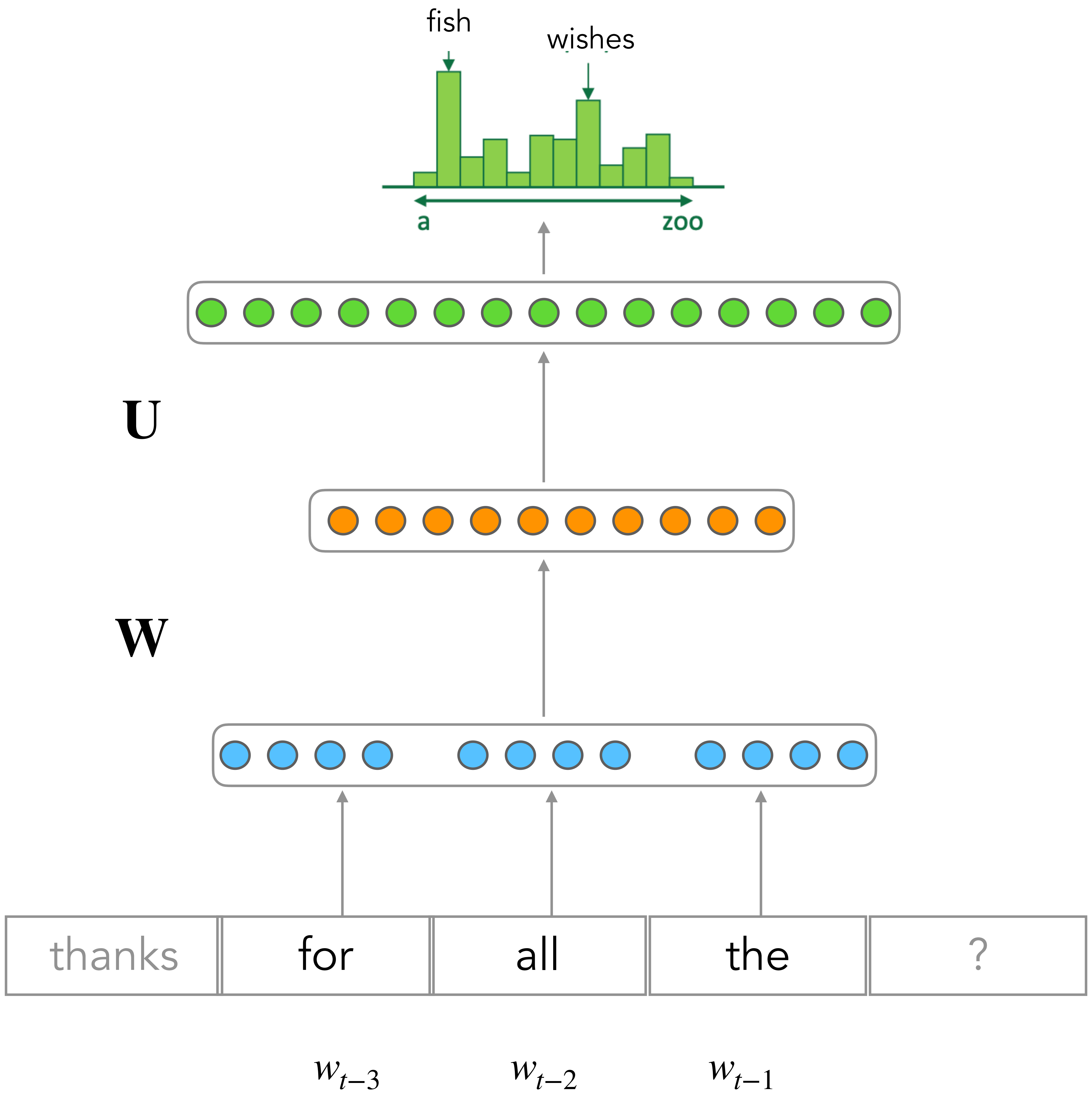
Feedforward LMs: Windows

- The goodness of the language model depends on the size of the sliding window!
- Fixed window can be too small
- Enlarging window enlarges $\mathbf{W}$
- Each word uses different rows of $\mathbf{W}$. We don't share weights across the window.



Feedforward LMs: Windows

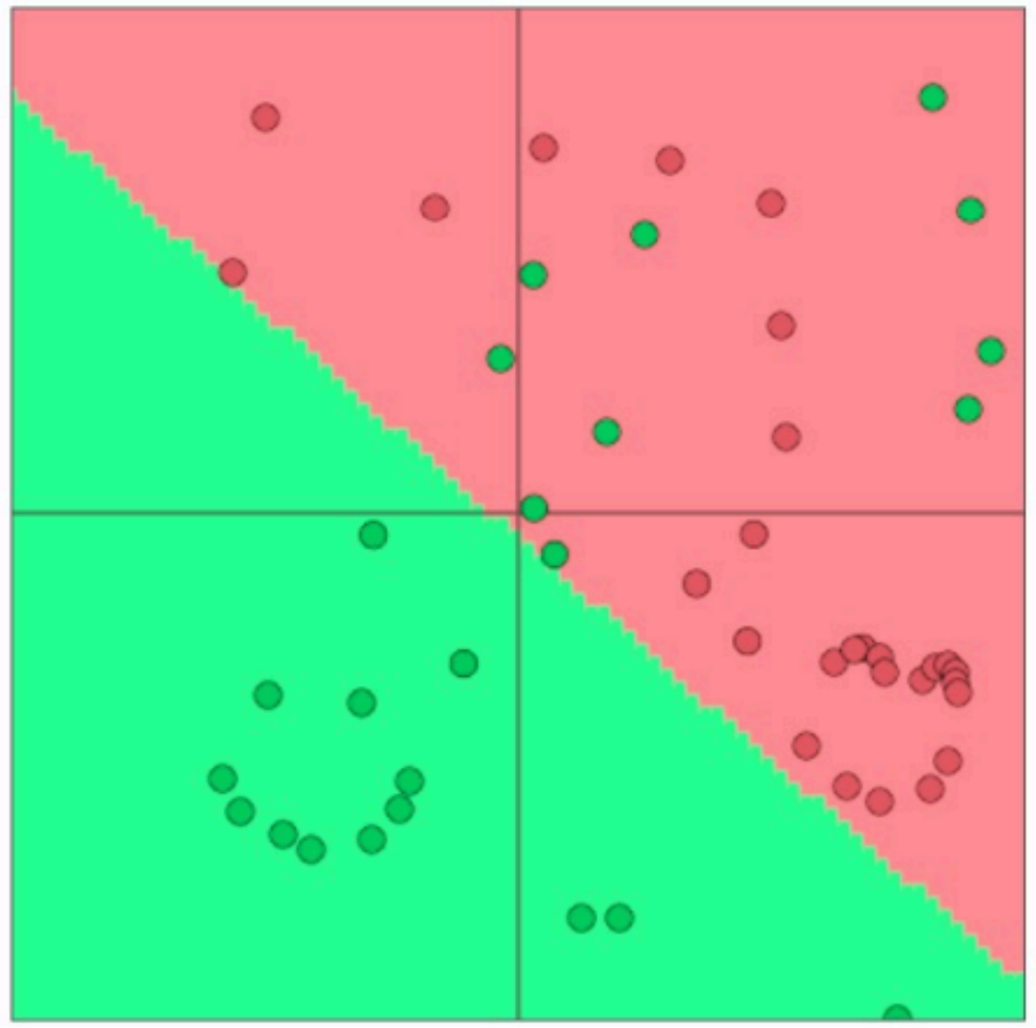
- The goodness of the language model depends on the size of the sliding window!
- Fixed window can be too small
- Enlarging window enlarges $\mathbf{W}$
- Each word uses different rows of $\mathbf{W}$. We don't share weights across the window.
- Window can never be large enough!

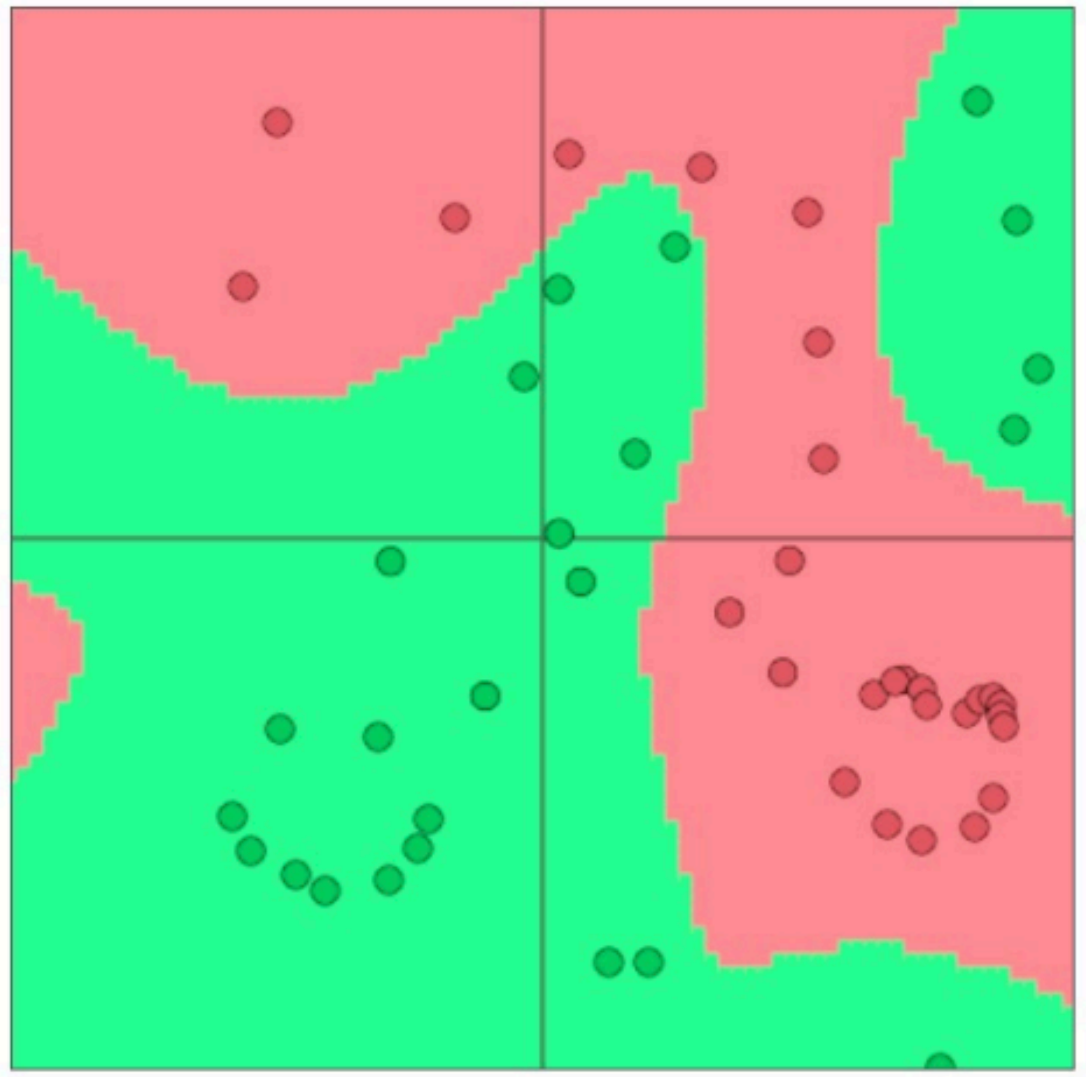


FFNN for Classification

FFNN and Classification

- Learn both $\mathbf{w}$ and (distributed!) representations for words
- The word vectors $\mathbf{x}$ re-represent one-hot vectors, moving them around in an intermediate layer vector space, for easy classification with a (linear) softmax classifier
- Conceptually, we have an embedding layer: $\mathbf{x}$
- We use deep networks—more layers—that let us re-represent and compose our data multiple times, giving a non-linear classifier



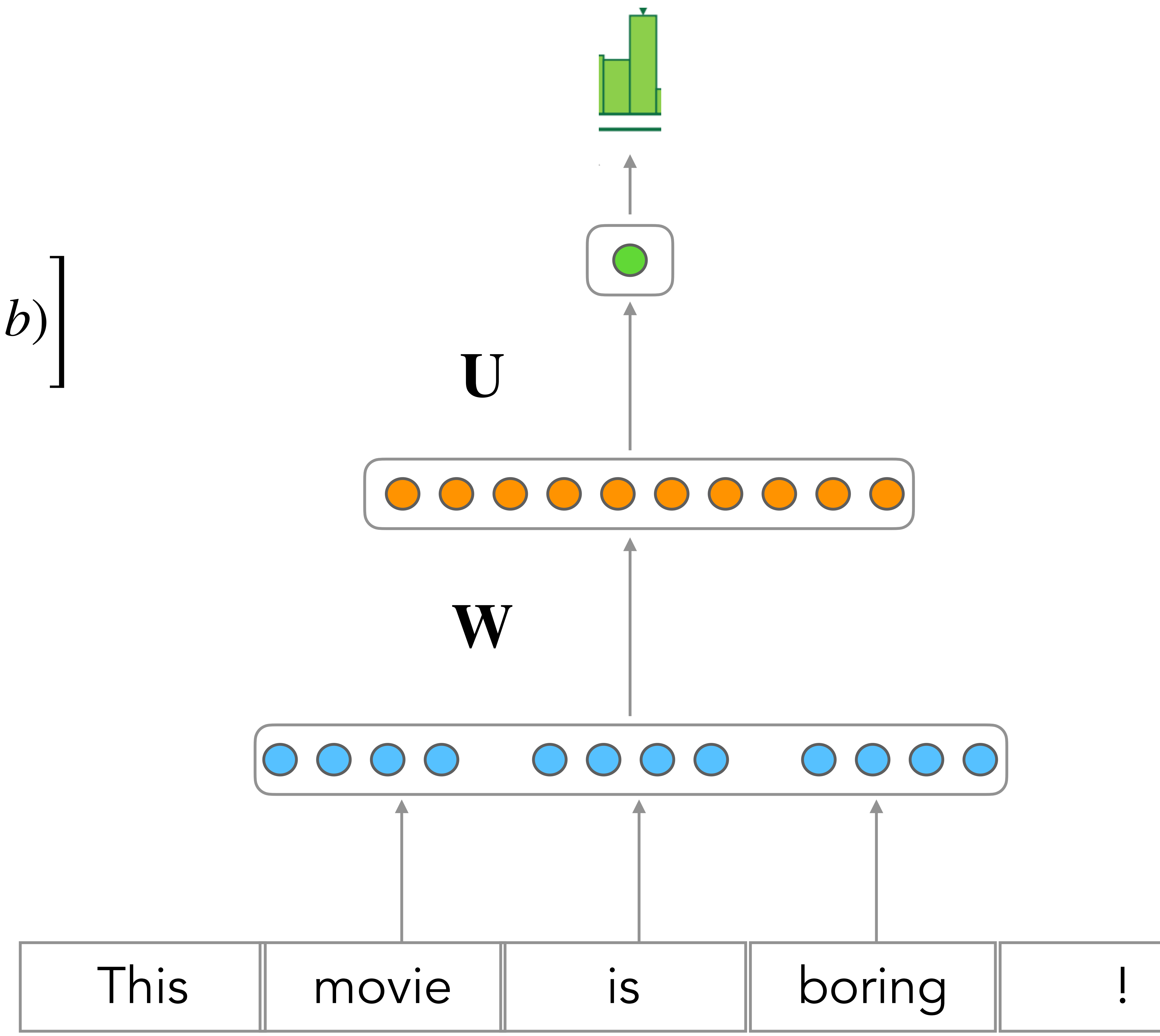


FNN and Classification

- Training Objective: For each training example $(\mathbf{x}, y)$, our objective is to maximize the probability of the correct class y or we can minimize the negative log probability of that class:

$$L_{CE} = -\log P(y = c \mid \mathbf{x}; \theta) = -(\mathbf{w}_c \cdot \mathbf{x} + b) + \log\left[\sum_{j=1}^{K} \exp(\mathbf{w}_j \cdot \mathbf{x} + b)\right]$$

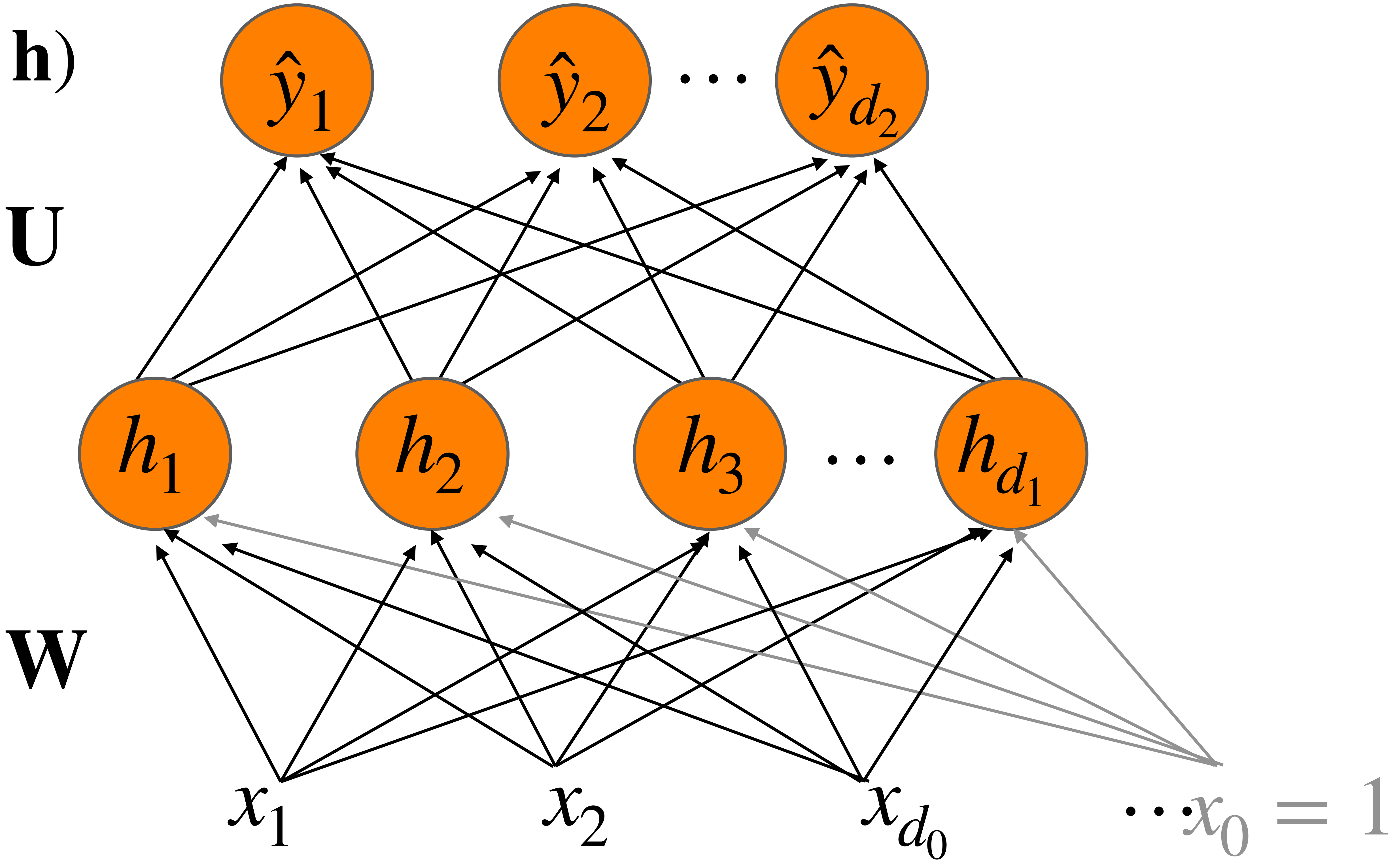
- Loss as Cross entropy: $H(p, q) = -\sum_{i=1}^{C} p_i \log q_i$
  - ground truth (or true or gold or target) is a 1-hot vector,$p = [0,\ldots,0,1,0,\ldots,0]$, then:
  - hence, the only term left is the negative log probability of the true class y_i^*: $-logp(y_i \mid x_i)$
  - True for both language modeling and classification



Training FFNNs

Intuition: Training a 2-layer Network

Model Output $\hat{\mathbf{y}} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$



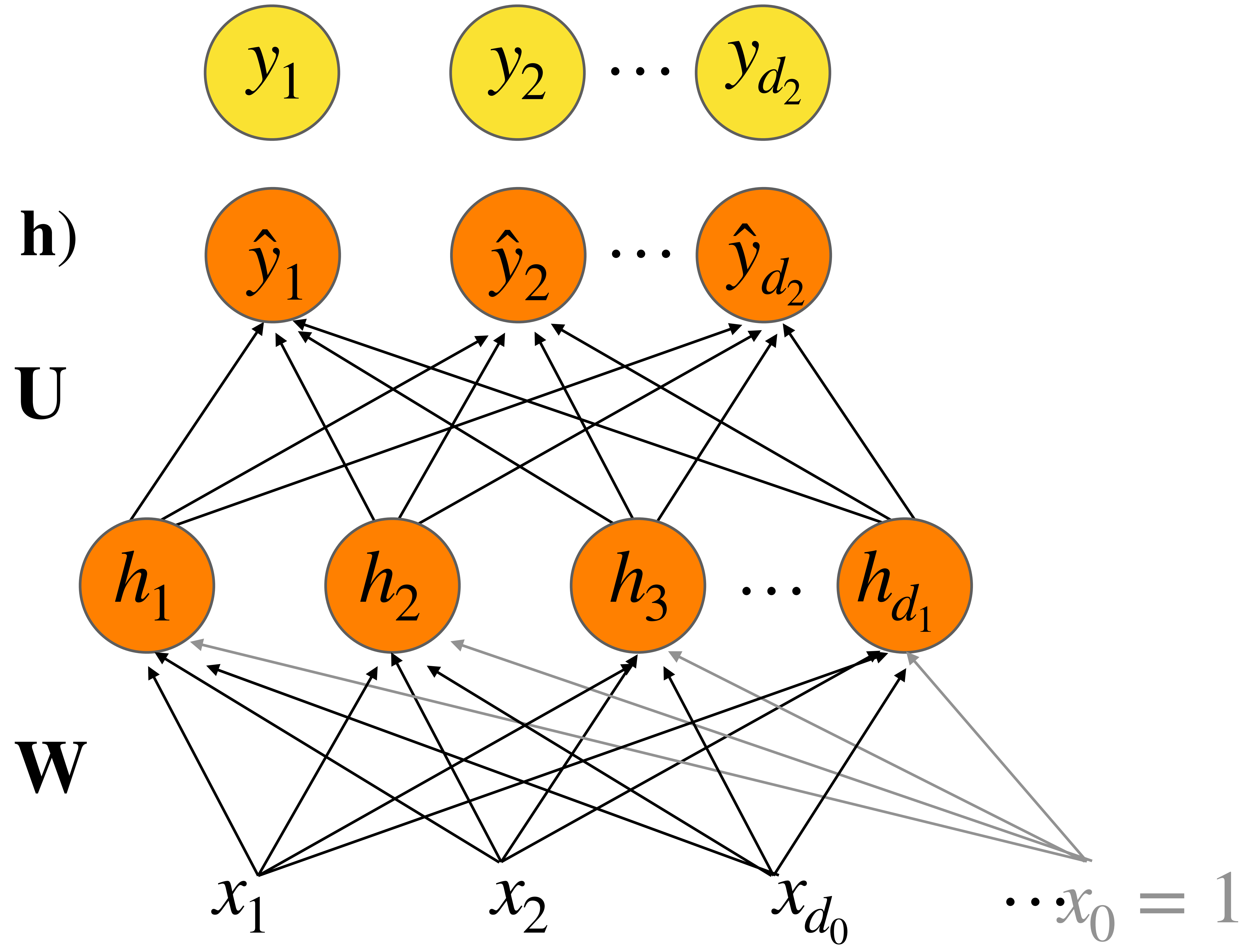
Training instance $\mathbf{x}$

Intuition: Training a 2-layer Network

Training instance $\mathbf{y}$

Model Output $\hat{\mathbf{y}} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$

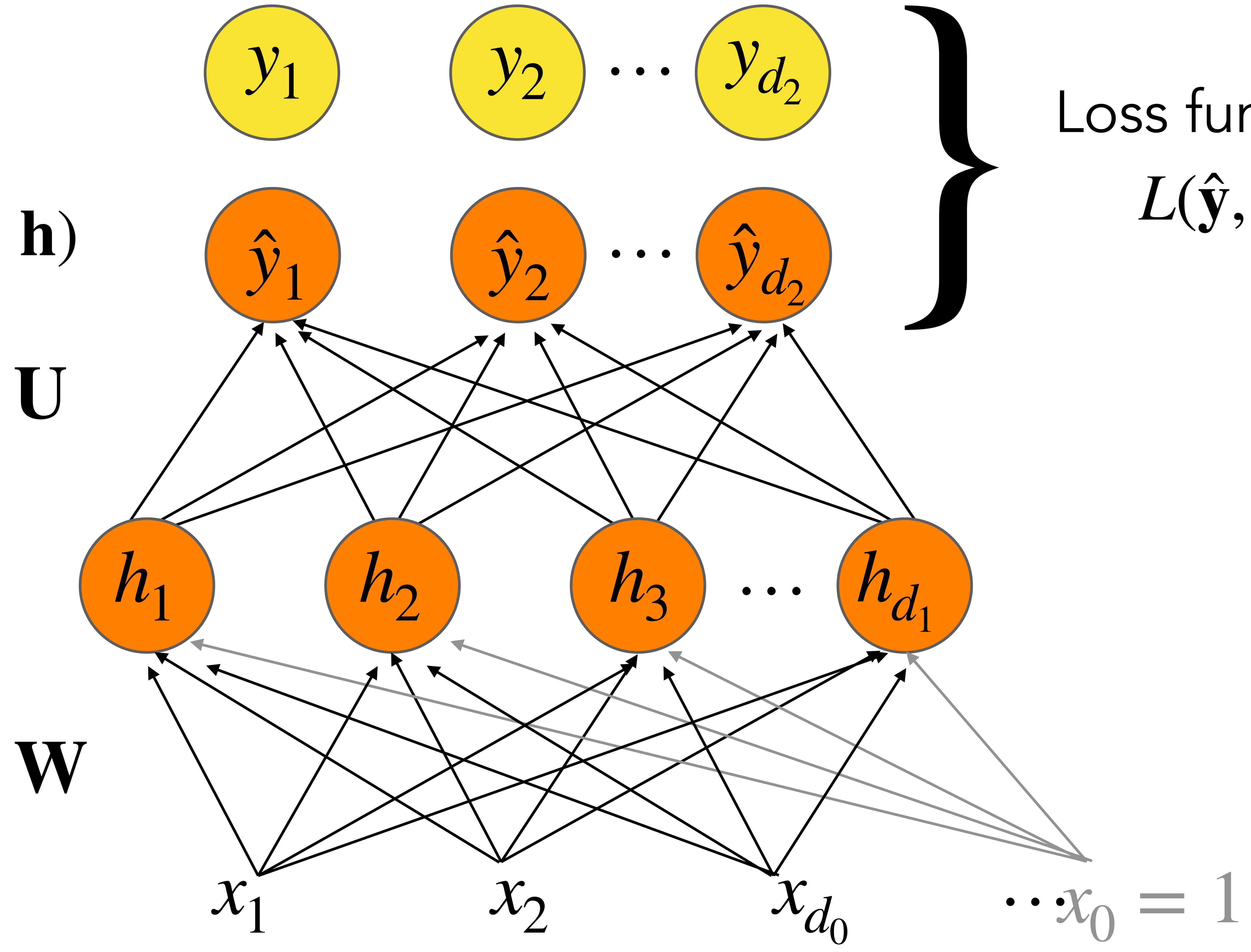
Training instance $\mathbf{x}$



Intuition: Training a 2-layer Network

Training instance $\mathbf{y}$

Model Output $\hat{\mathbf{y}} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$



Loss function

$L(\hat{\mathbf{y}}, \mathbf{y})$

Training instance $\mathbf{x}$

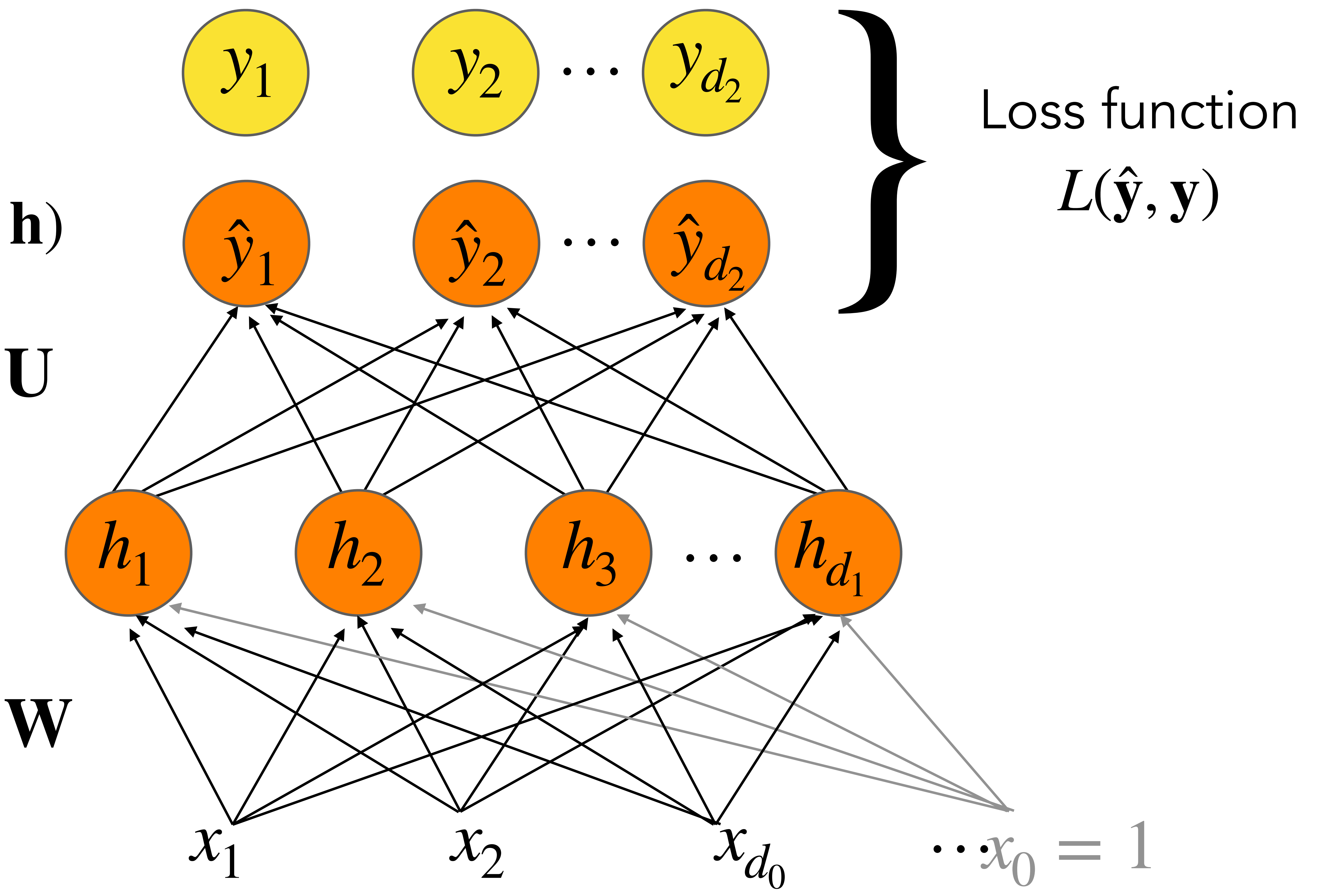
Intuition: Training a 2-layer Network

Training instance $\mathbf{y}$

Model Output $\hat{\mathbf{y}} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$

Forward Pass

Training instance $\mathbf{x}$



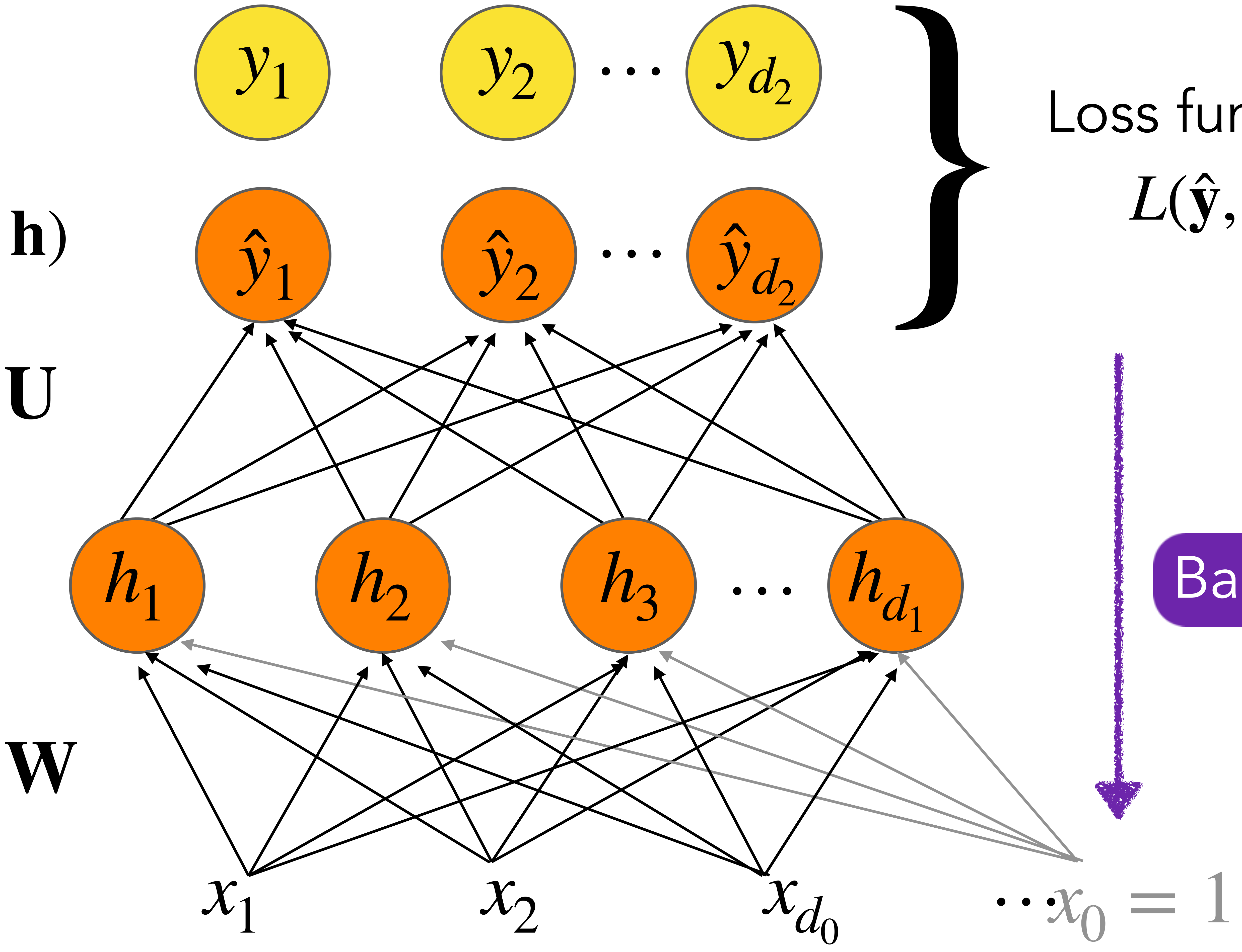
Intuition: Training a 2-layer Network

Training instance $\mathbf{y}$

Model Output $\hat{\mathbf{y}} = \text{softmax}(\mathbf{U} \cdot \mathbf{h})$

Forward Pass

Training instance $\mathbf{x}$



Backward Pass

Intuition: Training a 2-layer network

Intuition: Training a 2-layer network

For every training tuple (x, y)

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
    - For every output node

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
    - For every output node
        - Compute loss L between true y and the estimated $\hat{y}$
        - For every weight w from hidden layer to the output layer
            - Update the weight

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer
      - Update the weight
    - For every hidden node

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer
      - Update the weight
    - For every hidden node
      - Assess how much blame it deserves for the current answer

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer
      - Update the weight
    - For every hidden node
      - Assess how much blame it deserves for the current answer
      - For every weight w from input layer to the hidden layer

Intuition: Training a 2-layer network

For every training tuple (x, y)

- Run forward computation to find our estimate $\hat{y}$
- Run backward computation to update weights:
  - For every output node
    - Compute loss L between true y and the estimated $\hat{y}$
    - For every weight w from hidden layer to the output layer
      - Update the weight
    - For every hidden node
      - Assess how much blame it deserves for the current answer
      - For every weight w from input layer to the hidden layer
        - Update the weight

LR and FFNN: Similarities and Differences

Cross Entropy Loss again!

LR and FFNN: Similarities and Differences

Cross Entropy Loss again!

$$L_{CE}(y, \hat{y}) = -\log p(y \mid x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$
$$= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y)\log(\sigma(-\mathbf{w} \cdot \mathbf{x} + b))]$$

LR and FFNN: Similarities and Differences

Cross Entropy Loss again!

$$L_{CE}(y, \hat{y}) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$
$$= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y)\log(\sigma(-\mathbf{w} \cdot \mathbf{x} + b))]$$

Gradient Update

$$\frac{\partial L_{CE}(\hat{y}, y)}{\partial w_j} = [\sigma(\mathbf{w} \cdot \mathbf{x} + b) - y]x_j$$

LR and FFNN: Similarities and Differences

Cross Entropy Loss again!

$$L_{CE}(y, \hat{y}) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$
$$= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y)\log(\sigma(-\mathbf{w} \cdot \mathbf{x} + b))]$$

Gradient Update

$$\frac{\partial L_{CE}(\hat{y}, y)}{\boxed{\partial w_j}} = [\sigma(\mathbf{w} \cdot \mathbf{x} + b) - y]x_j$$

Only one parameter!

LR and FFNN: Similarities and Differences

Cross Entropy Loss again!

$$L_{CE}(y, \hat{y}) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$
$$= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y)\log(\sigma(-\mathbf{w} \cdot \mathbf{x} + b))]$$

Gradient Update
$$\frac{\partial L_{CE}(\hat{y}, y)}{\boxed{\partial w_j}} = [\sigma(\mathbf{w} \cdot \mathbf{x} + b) - y]x_j$$

Only one parameter!

As (multiple) hidden layers are introduced, there will be many more parameters to consider, not to mention activation functions!

LR and FFNN: Similarities and Differences

Cross Entropy Loss again!

$$L_{CE}(y, \hat{y}) = -\log p(y|x) = -[y \log \hat{y} + (1-y)\log(1-\hat{y})]$$
$$= -[y \log \sigma(\mathbf{w} \cdot \mathbf{x} + b) + (1-y)\log(\sigma(-\mathbf{w} \cdot \mathbf{x} + b))]$$

Gradient Update

$$\frac{\partial L_{CE}(\hat{y}, y)}{\boxed{\partial w_j}} = [\sigma(\mathbf{w} \cdot \mathbf{x} + b) - y]x_j$$

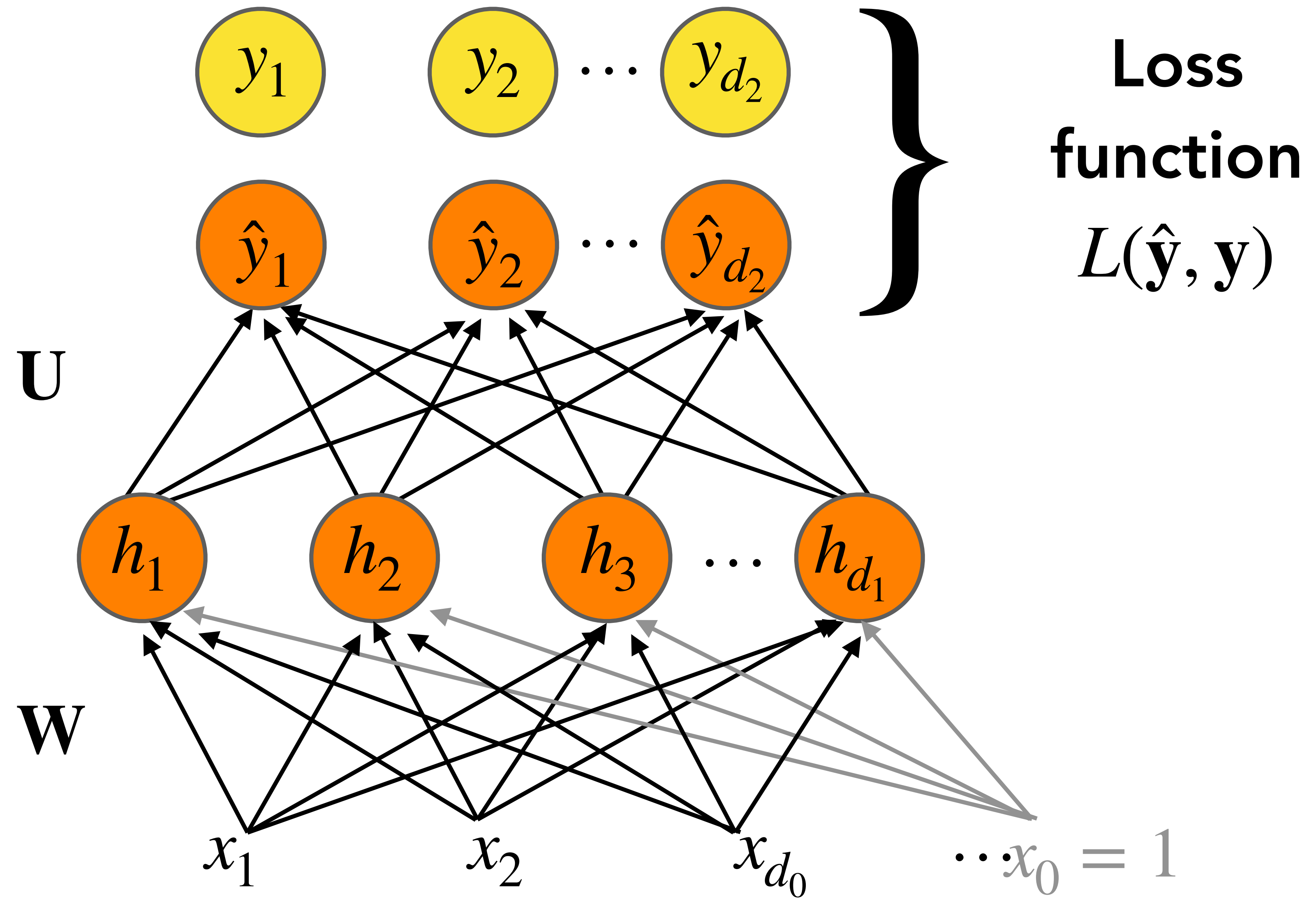
Only one parameter!

Computation Graphs

As (multiple) hidden layers are introduced, there will be many more parameters to consider, not to mention activation functions!

Computation Graphs and Backprop

Why Computation Graphs?

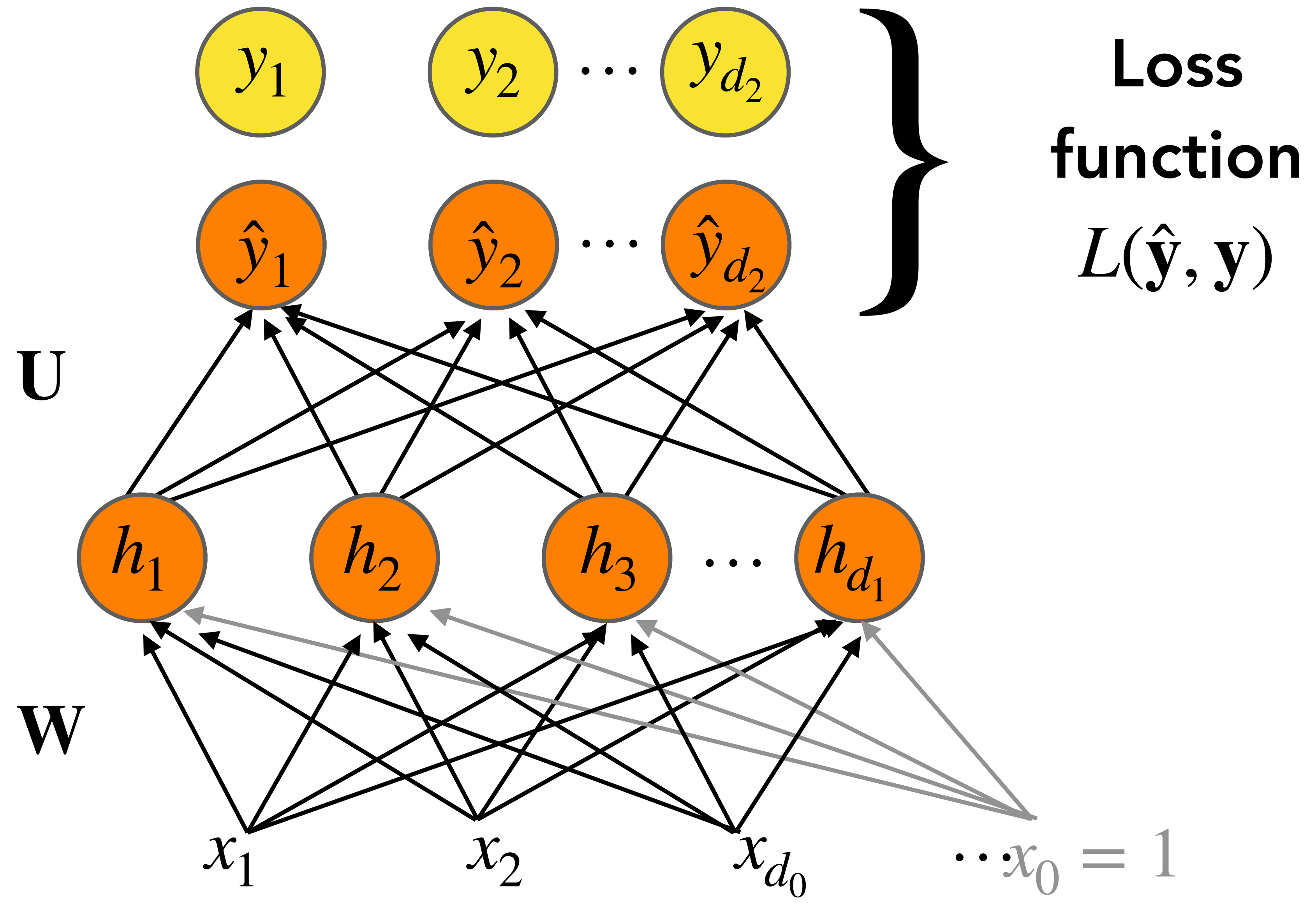


Graph representing the process of computing a mathematical expression

Rumelhart, Hinton, Williams, 1986

Why Computation Graphs?

- For training, we need the derivative of the loss with respect to each weight in every layer of the network

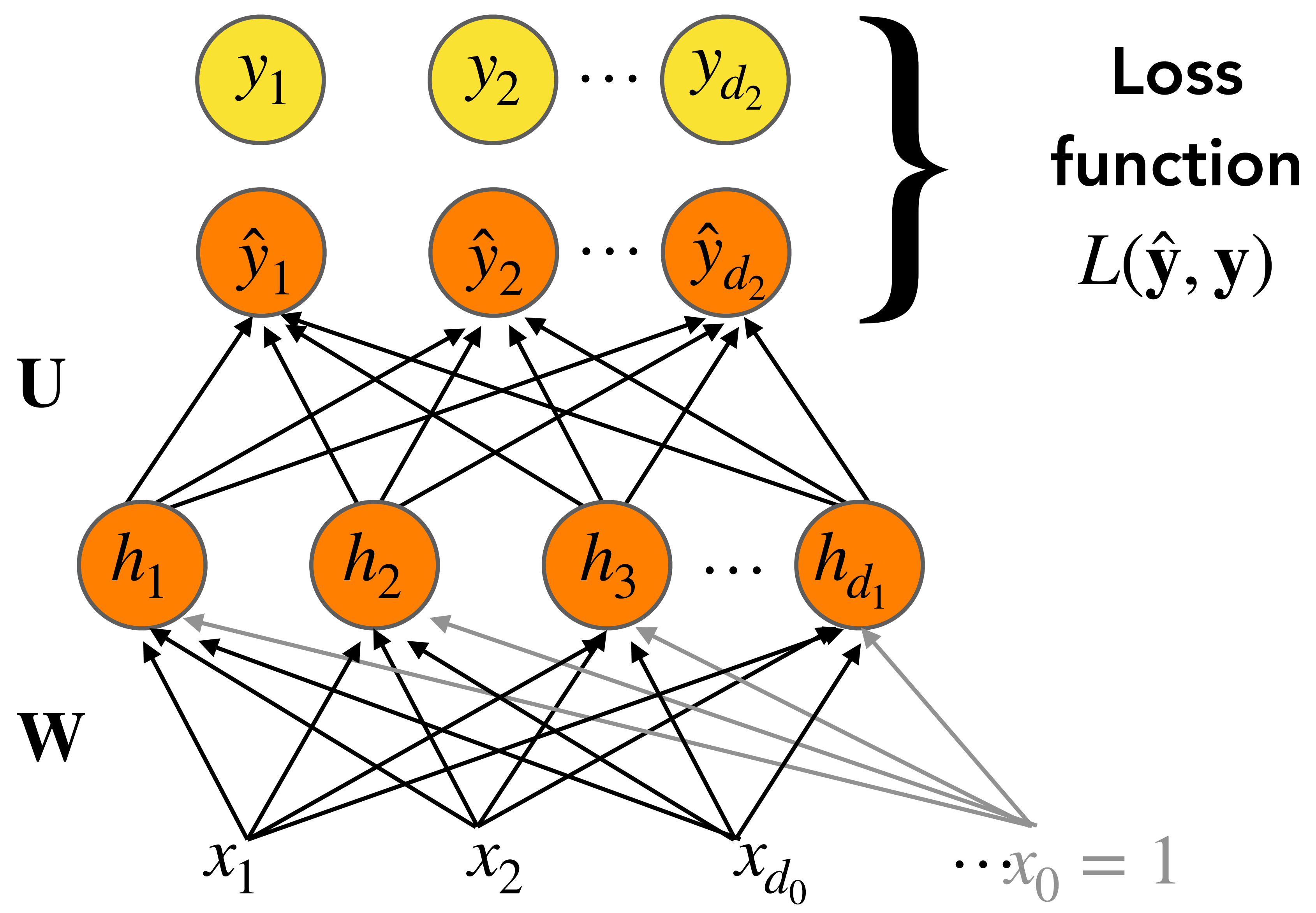


Graph representing the process of computing a mathematical expression

Rumelhart, Hinton, Williams, 1986

Why Computation Graphs?

- For training, we need the derivative of the loss with respect to each weight in every layer of the network
  - But the loss is computed only at the very end of the network!

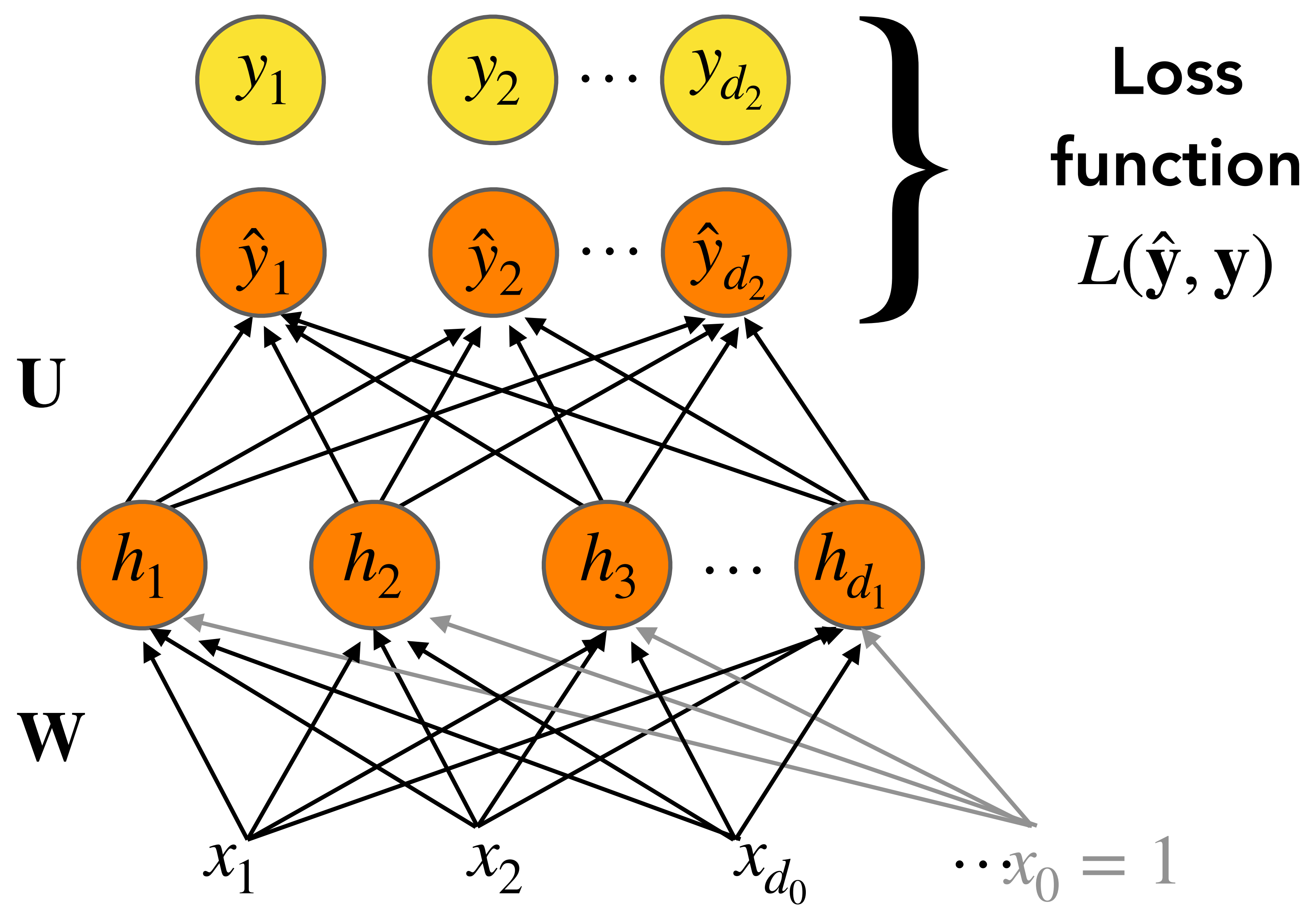


Graph representing the process of computing a mathematical expression

Rumelhart, Hinton, Williams, 1986

Why Computation Graphs?

- For training, we need the derivative of the loss with respect to each weight in every layer of the network
  - But the loss is computed only at the very end of the network!
- Solution: error backpropagation or backward differentiation

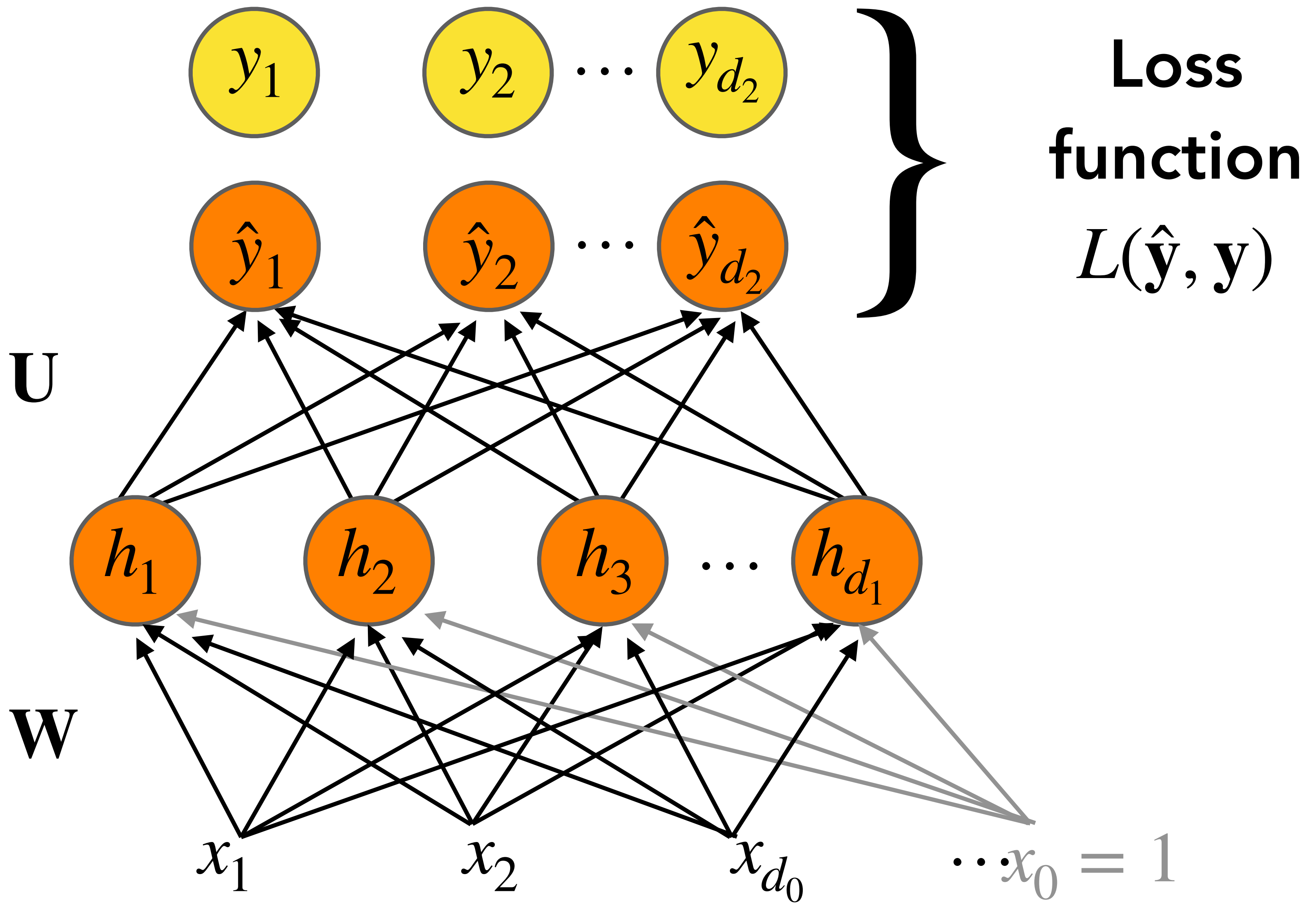


Graph representing the process of computing a mathematical expression

Rumelhart, Hinton, Williams, 1986

Why Computation Graphs?

- For training, we need the derivative of the loss with respect to each weight in every layer of the network
  - But the loss is computed only at the very end of the network!
- Solution: error backpropagation or backward differentiation
  - Backprop is a special case of backward differentiation



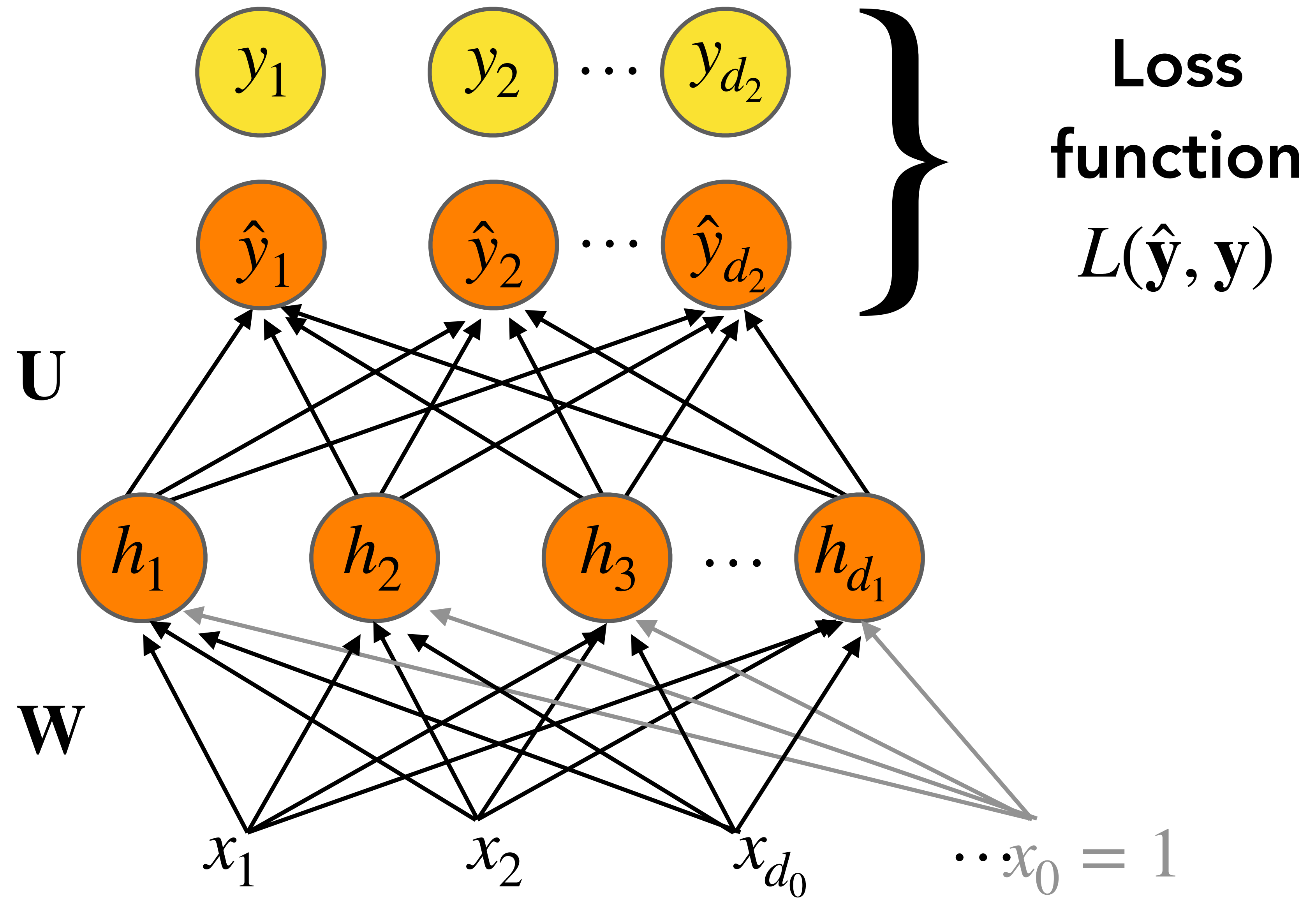
Backprop

Graph representing the process of computing a mathematical expression

Rumelhart, Hinton, Williams, 1986

Why Computation Graphs?

- For training, we need the derivative of the loss with respect to each weight in every layer of the network
  - But the loss is computed only at the very end of the network!
- Solution: error backpropagation or backward differentiation
  - Backprop is a special case of backward differentiation
    - Which relies on computation graphs



Backprop

Graph representing the process of computing a mathematical expression

Rumelhart, Hinton, Williams, 1986

Example: Computation Graph

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

Example: Computation Graph

$$d = 2 * b$$

$$e = a + d$$

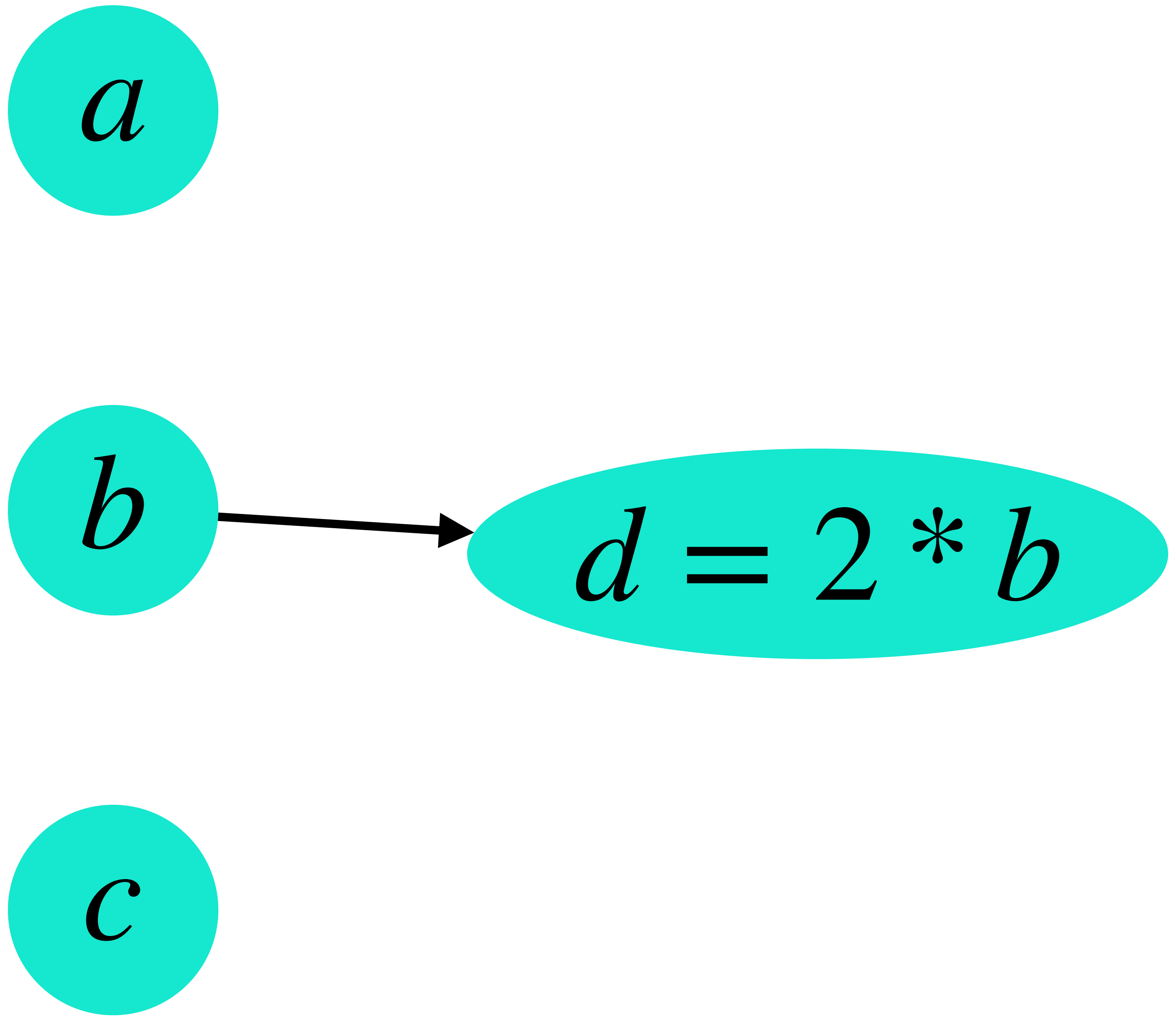
$$L = c * e$$

Example: Computation Graph

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

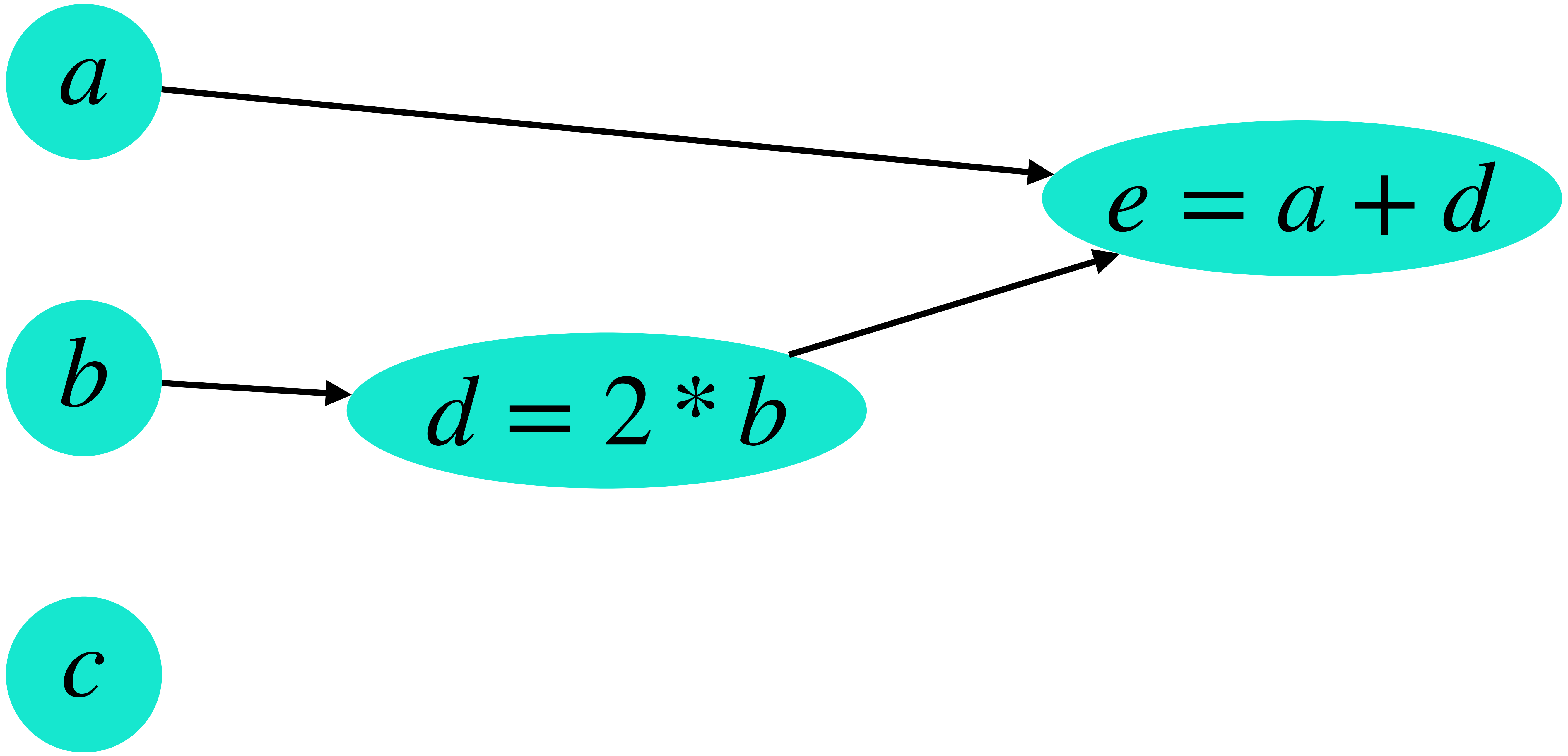


Example: Computation Graph

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

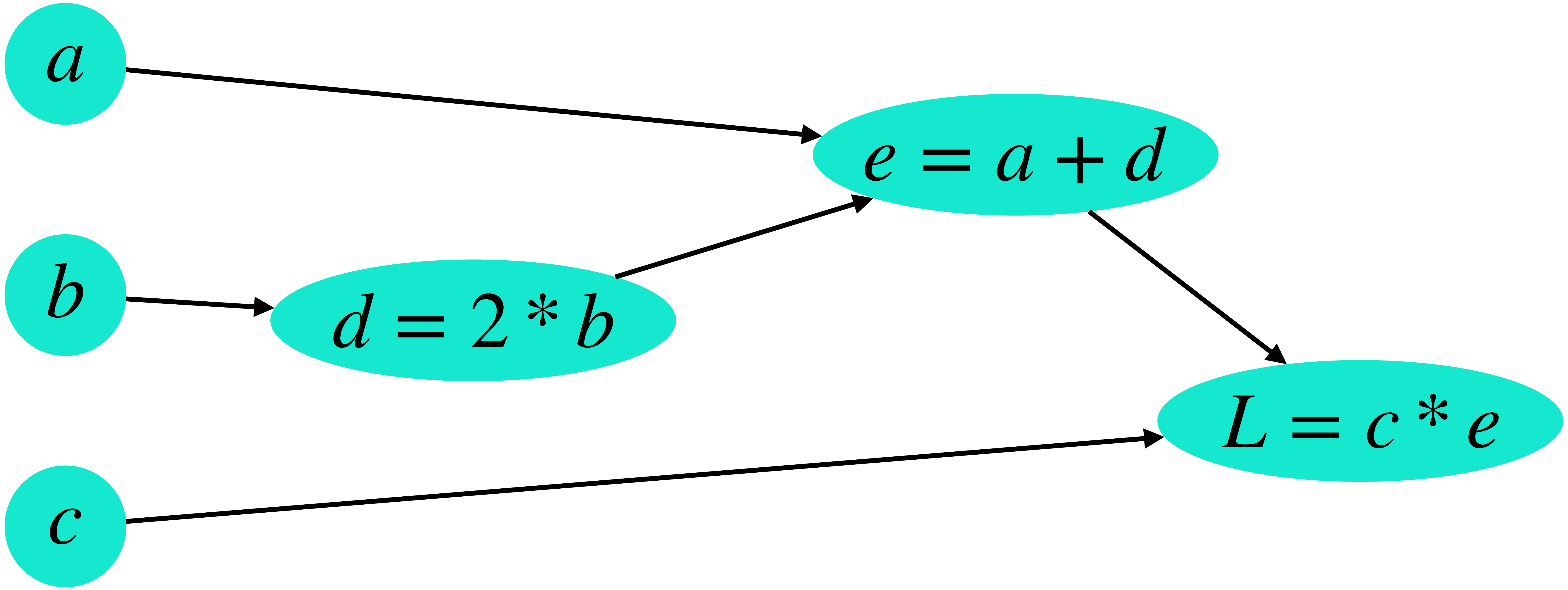


Example: Computation Graph

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

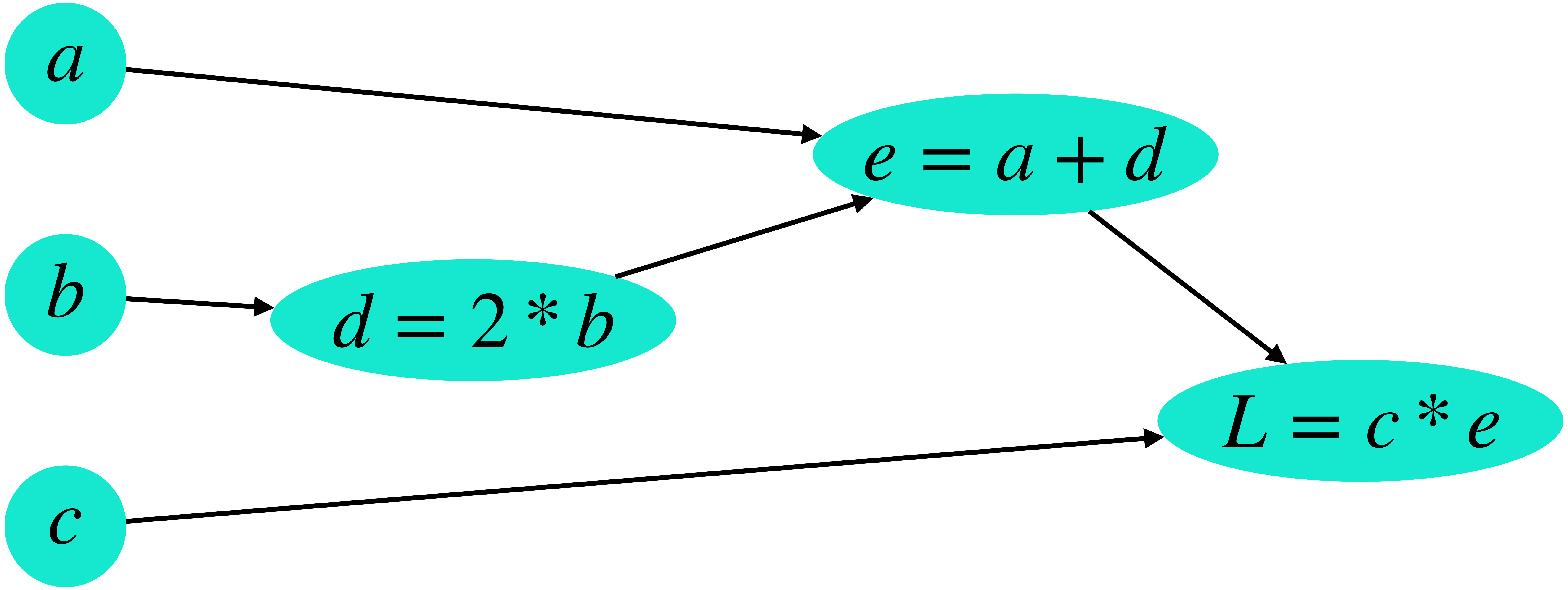


Example: Forward Pass

$d = 2 * b$

$e = a + d$

$L = c * e$

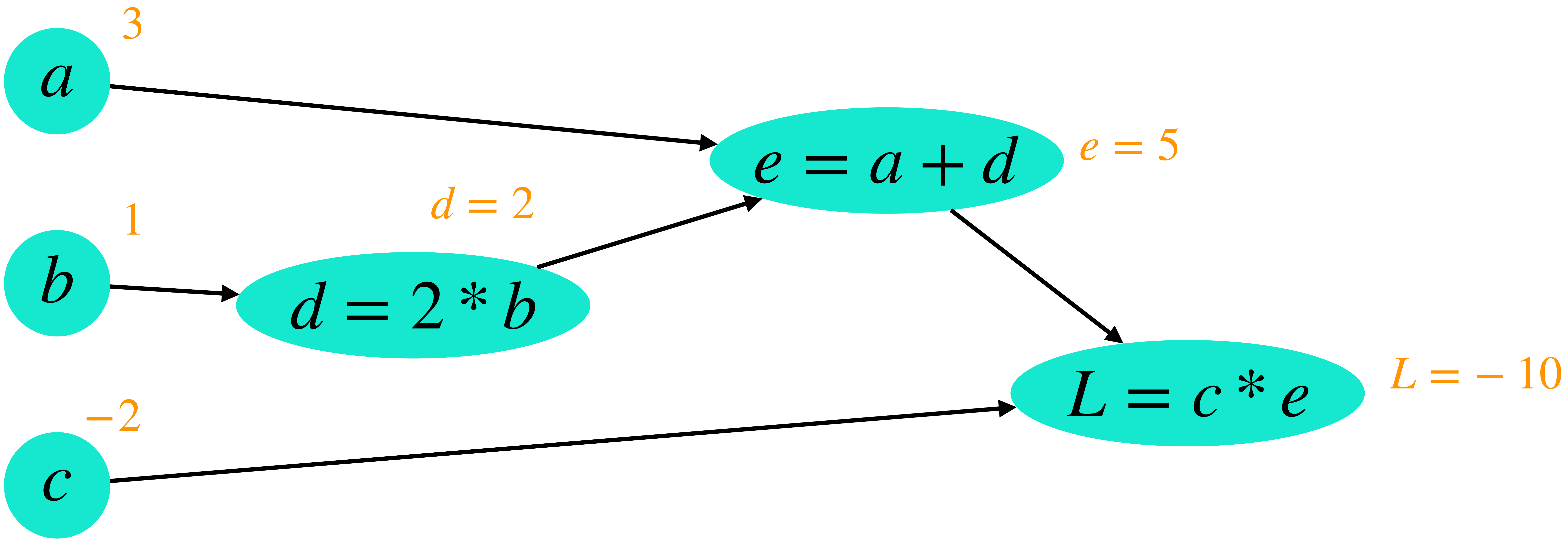


Example: Forward Pass

$d = 2 * b$

$e = a + d$

$L = c * e$

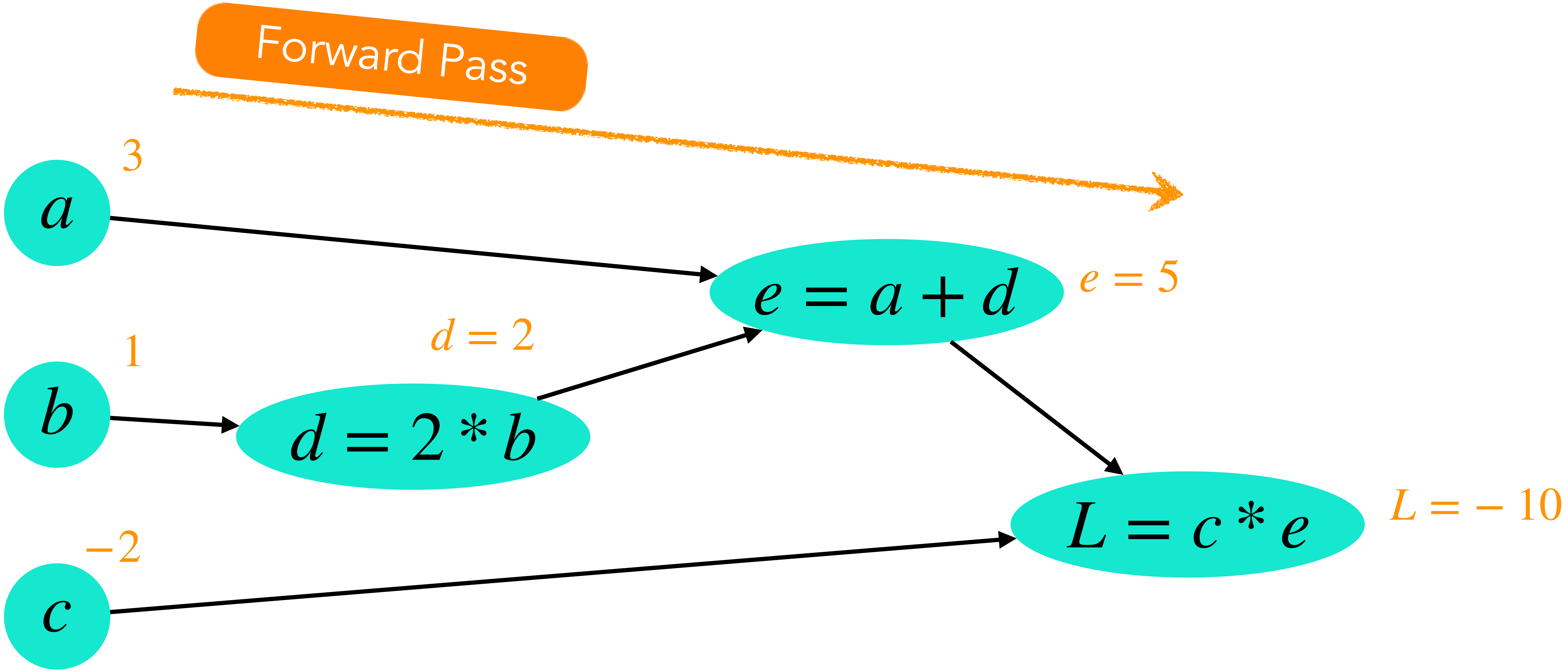


Example: Forward Pass

$d = 2 * b$

$e = a + d$

$L = c * e$

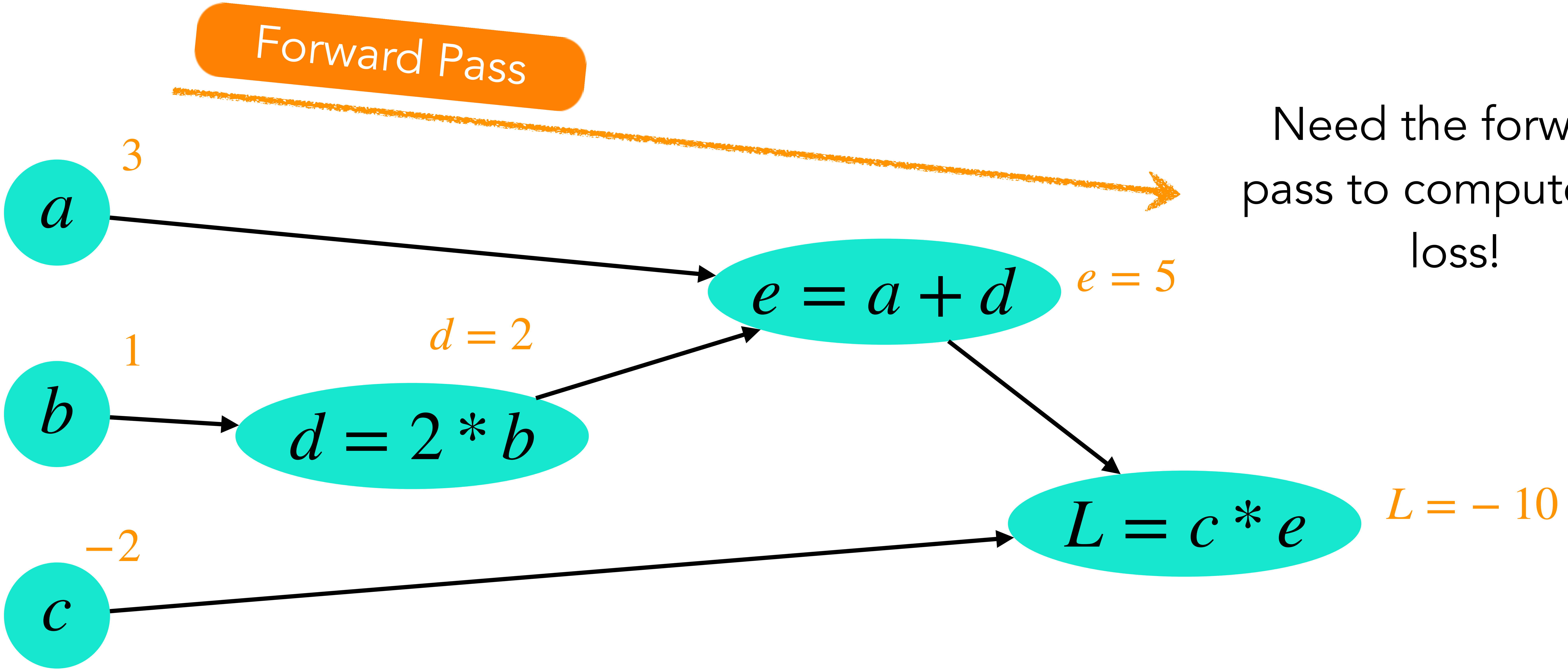


Example: Forward Pass

$d = 2 * b$

$e = a + d$

$L = c * e$



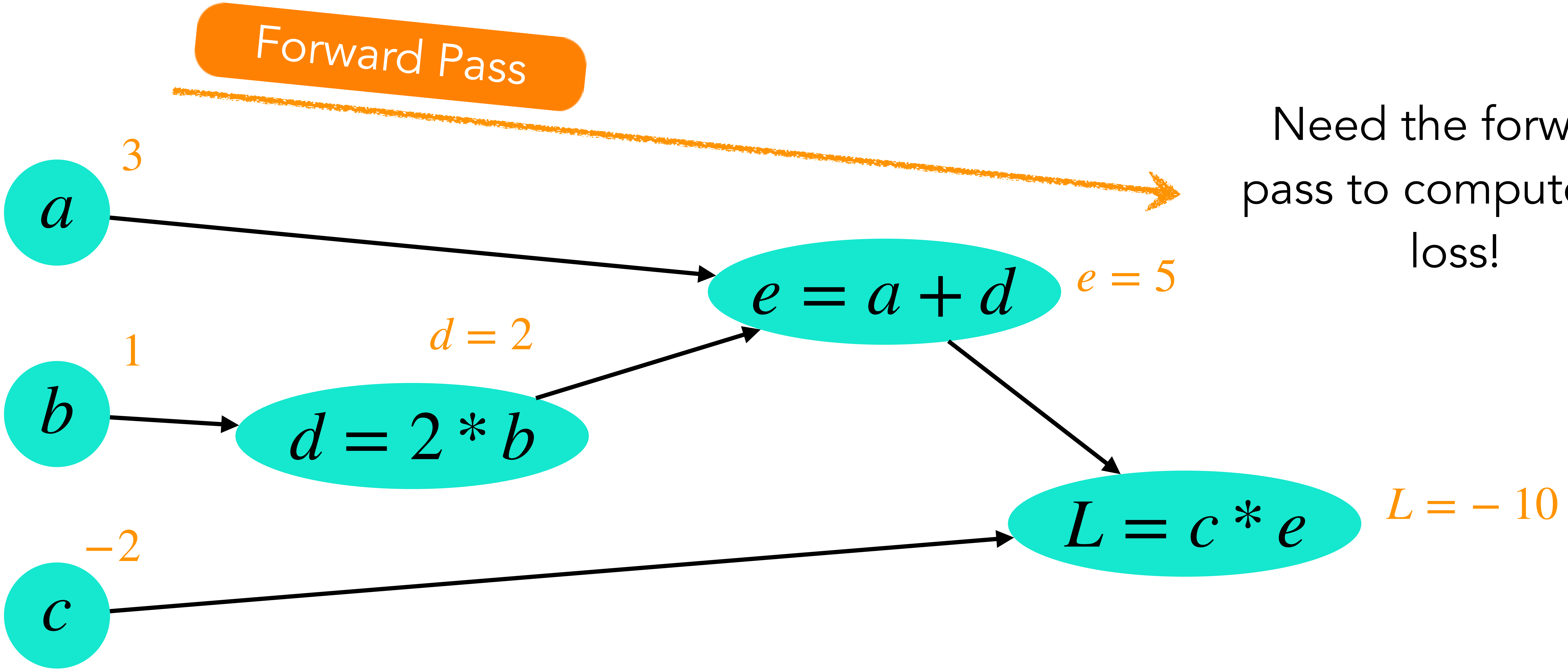
Need the forward pass to compute the loss!

Example: Forward Pass

$d = 2 * b$

$e = a + d$

$L = c * e$



Need the forward pass to compute the loss!

But how to compute parameter updates?

Example: Backward Pass Intuition

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

Example: Backward Pass Intuition

- The importance of the computation graph comes from the **backward pass**

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

Example: Backward Pass Intuition

- The importance of the computation graph comes from the **backward pass**
- Used to compute the derivatives needed for the weight updates

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

Example: Backward Pass Intuition

- The importance of the computation graph comes from the **backward pass**
- Used to compute the derivatives needed for the weight updates

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

$$\frac{\partial L}{\partial a} = ? \qquad \frac{\partial L}{\partial b} = ? \qquad \frac{\partial L}{\partial c} = ?$$

Example: Backward Pass Intuition

- The importance of the computation graph comes from the **backward pass**
- Used to compute the derivatives needed for the weight updates

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

$$\left.\frac{\partial L}{\partial a} = ? \qquad \frac{\partial L}{\partial b} = ? \qquad \frac{\partial L}{\partial c} = ? \right\}$$

Input Layer Gradients

Example: Backward Pass Intuition

- The importance of the computation graph comes from the **backward pass**
- Used to compute the derivatives needed for the weight updates

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

$$\left.\frac{\partial L}{\partial a} = ? \qquad \frac{\partial L}{\partial b} = ? \qquad \frac{\partial L}{\partial c} = ?\right\}$$

Input Layer Gradients

$$\frac{\partial L}{\partial d} = ? \qquad \frac{\partial L}{\partial e} = ?$$

Example: Backward Pass Intuition

- The importance of the computation graph comes from the **backward pass**
- Used to compute the derivatives needed for the weight updates

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

Input Layer Gradients

$$\frac{\partial L}{\partial a} = ? \quad \frac{\partial L}{\partial b} = ? \quad \frac{\partial L}{\partial c} = ?$$

Hidden Layer Gradients

$$\frac{\partial L}{\partial d} = ? \quad \frac{\partial L}{\partial e} = ?$$

Example: Backward Pass Intuition

- The importance of the computation graph comes from the **backward pass**
- Used to compute the derivatives needed for the weight updates

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

$$\left.\frac{\partial L}{\partial a} = ? \quad \frac{\partial L}{\partial b} = ? \quad \frac{\partial L}{\partial c} = ?\right\}$$

Input Layer Gradients

Hidden Layer Gradients

$$\left\{\frac{\partial L}{\partial d} = ? \quad \frac{\partial L}{\partial e} = ?\right.$$

Chain Rule of Differentiation!

Example: Applying the chain rule

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

$$\frac{\partial L}{\partial c} = e$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

$$\frac{\partial L}{\partial e} = c$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}$$

Example: Applying the chain rule

$$d = 2 * b$$

$$e = a + d$$

$$L = c * e$$

$$\frac{\partial L}{\partial c} = e$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$

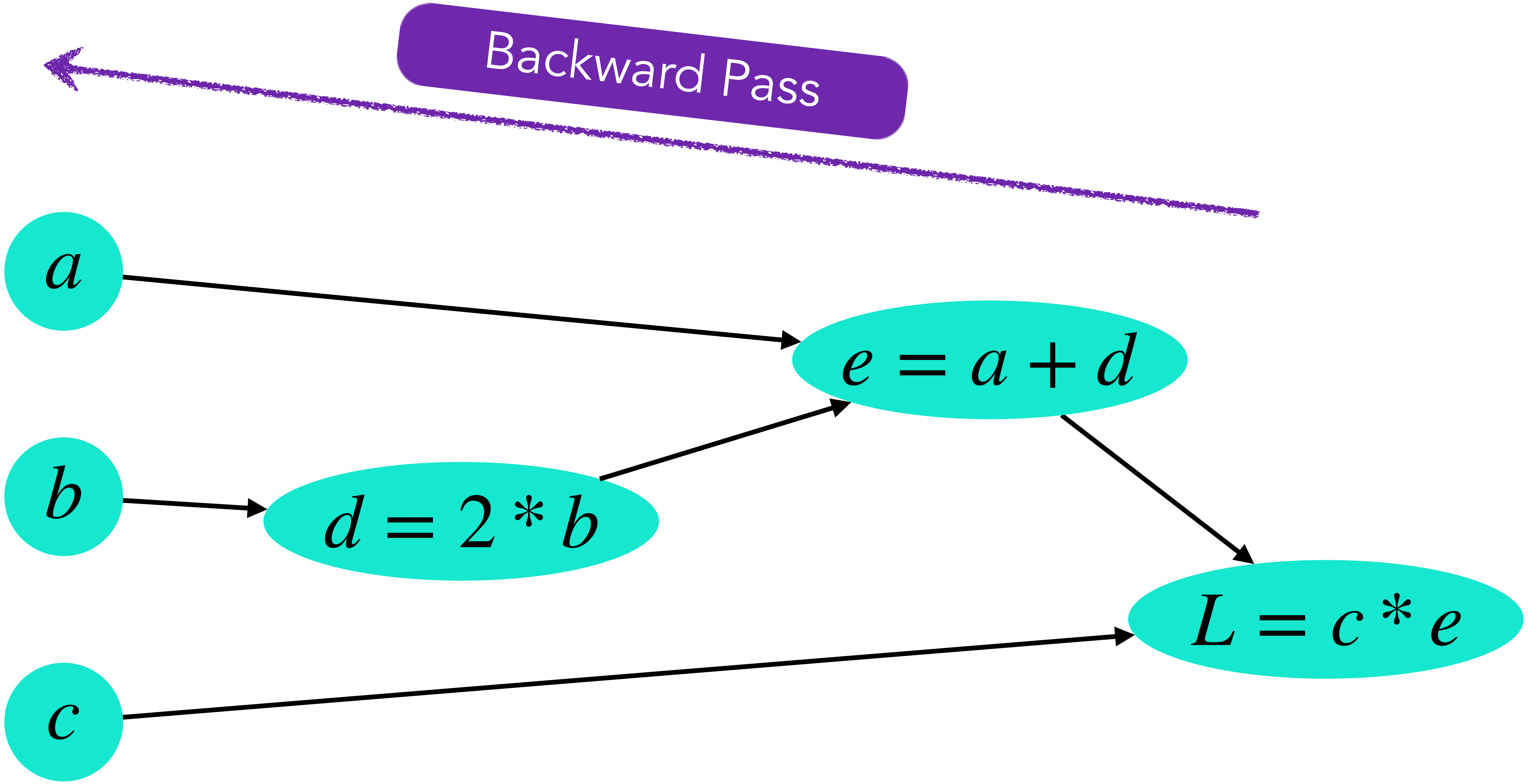
$$\frac{\partial L}{\partial e} = c$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}$$

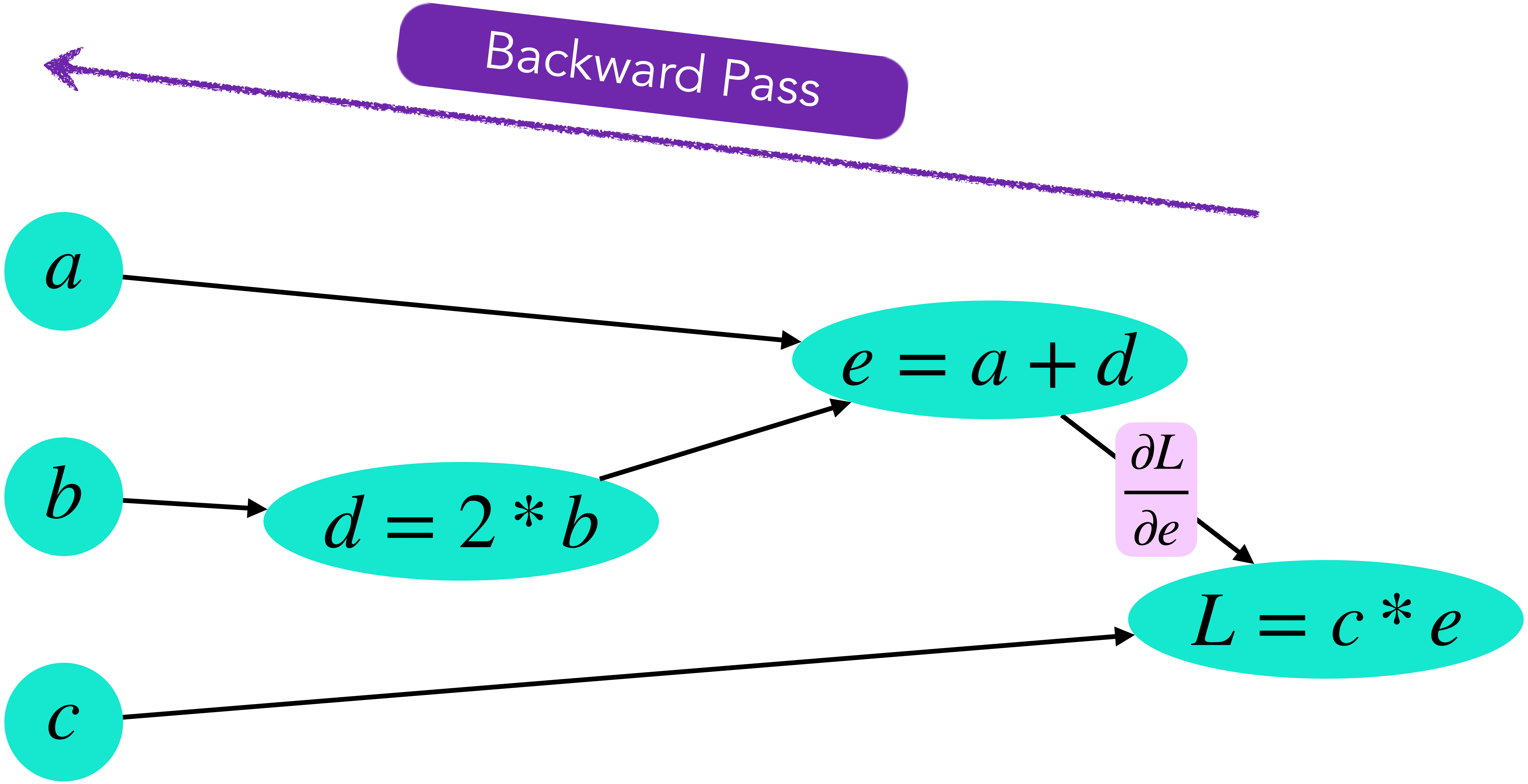
Cannot do all at once, need to follow an order…

USC Viterbi

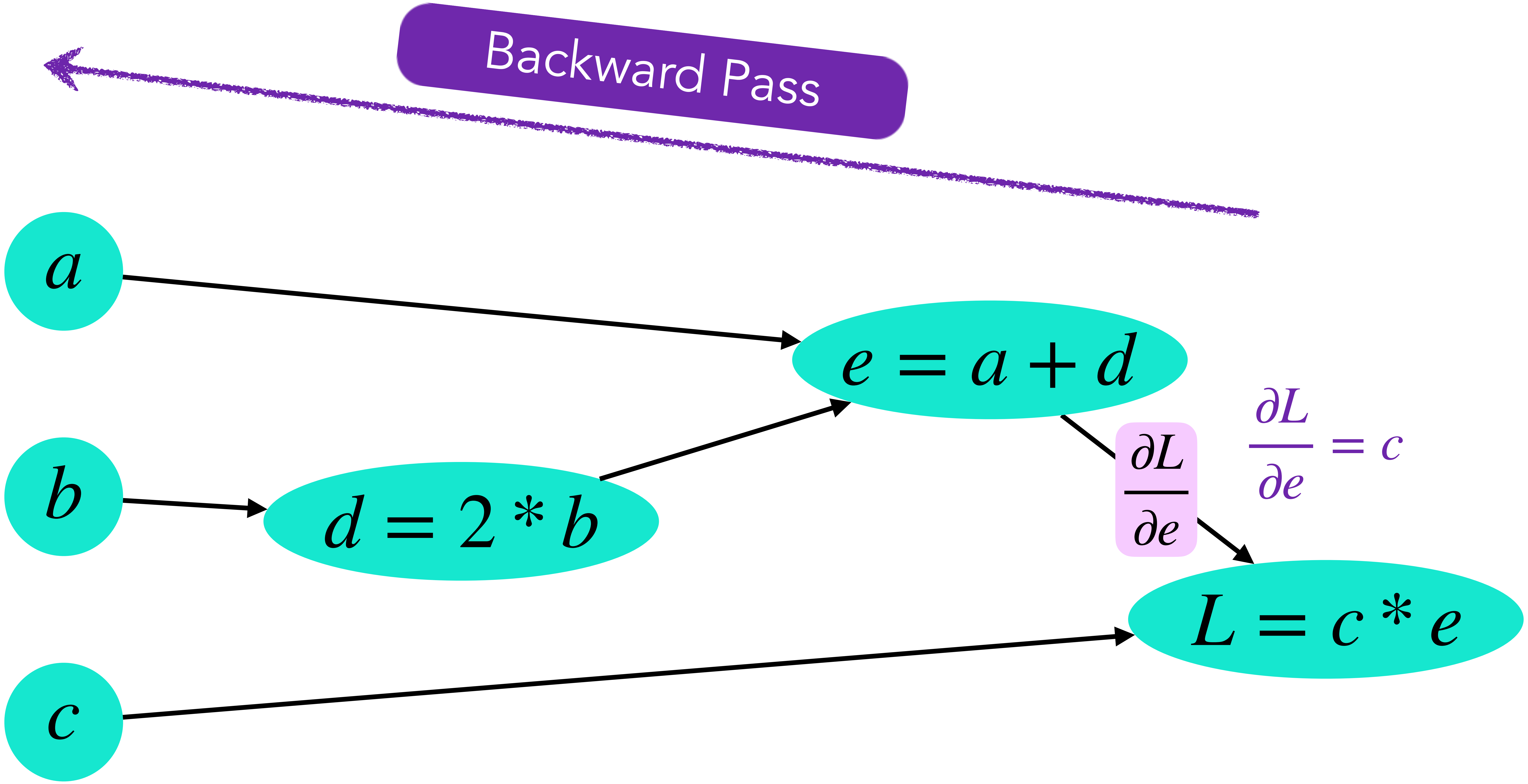
Example: Backward Pass



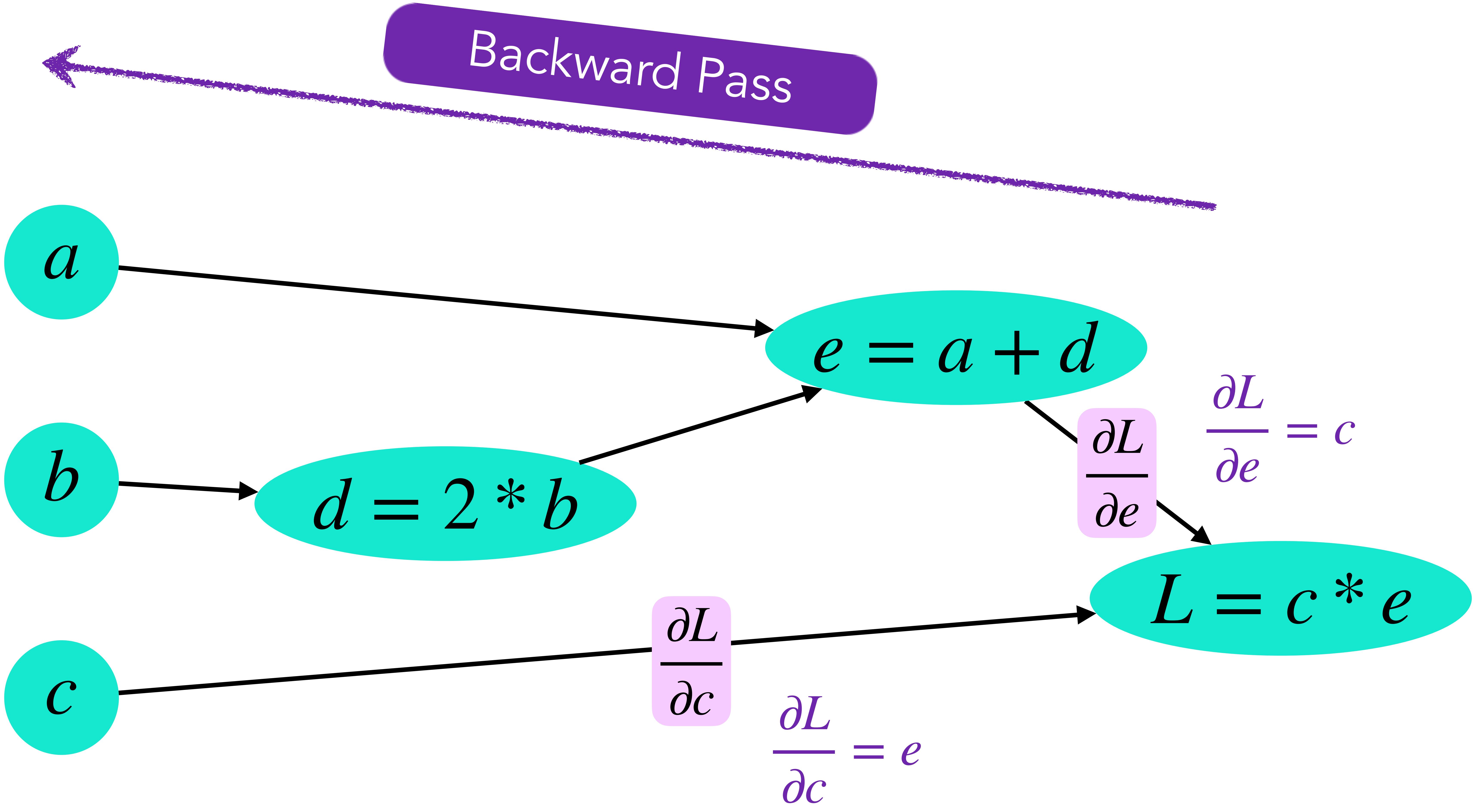
Example: Backward Pass



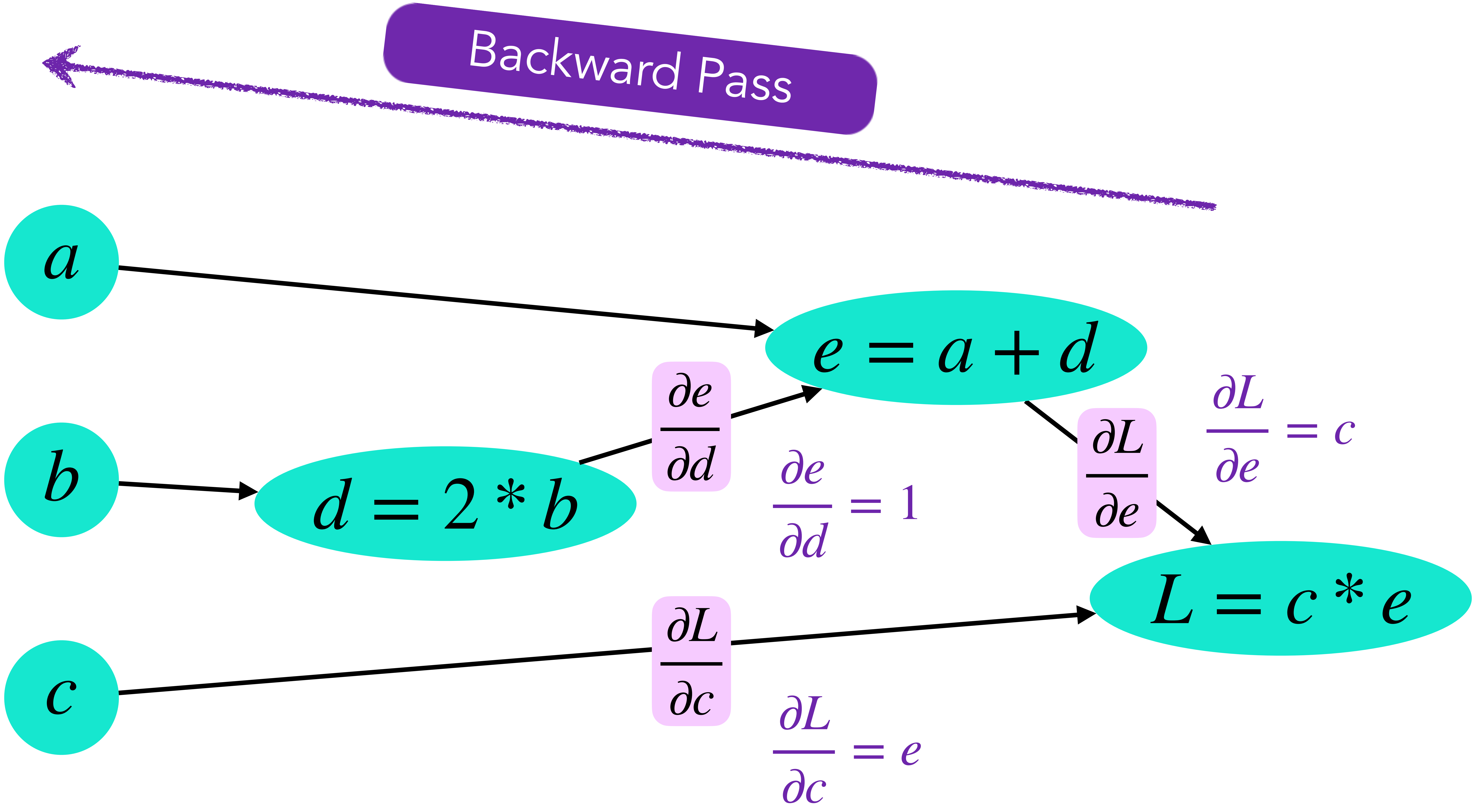
Example: Backward Pass



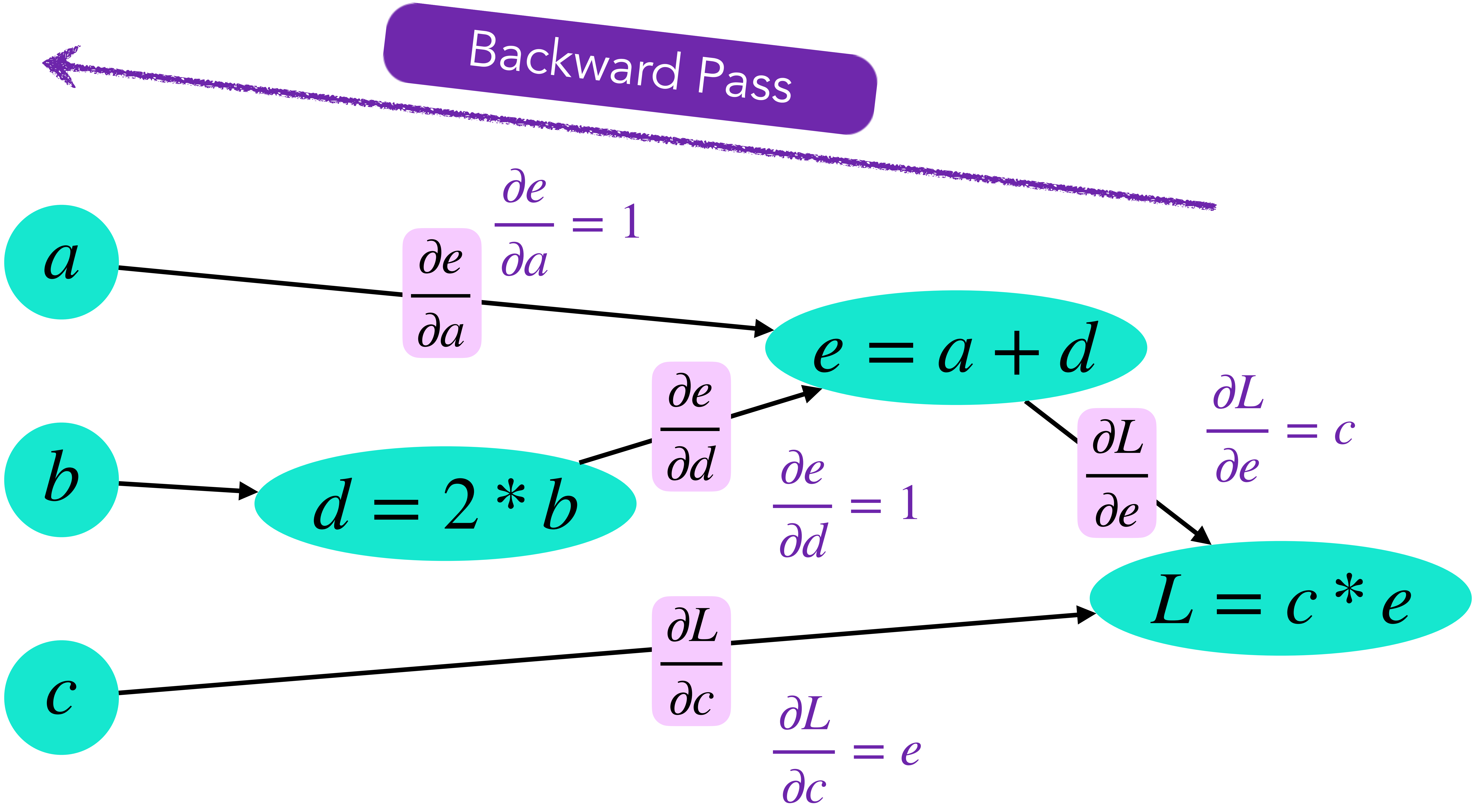
Example: Backward Pass



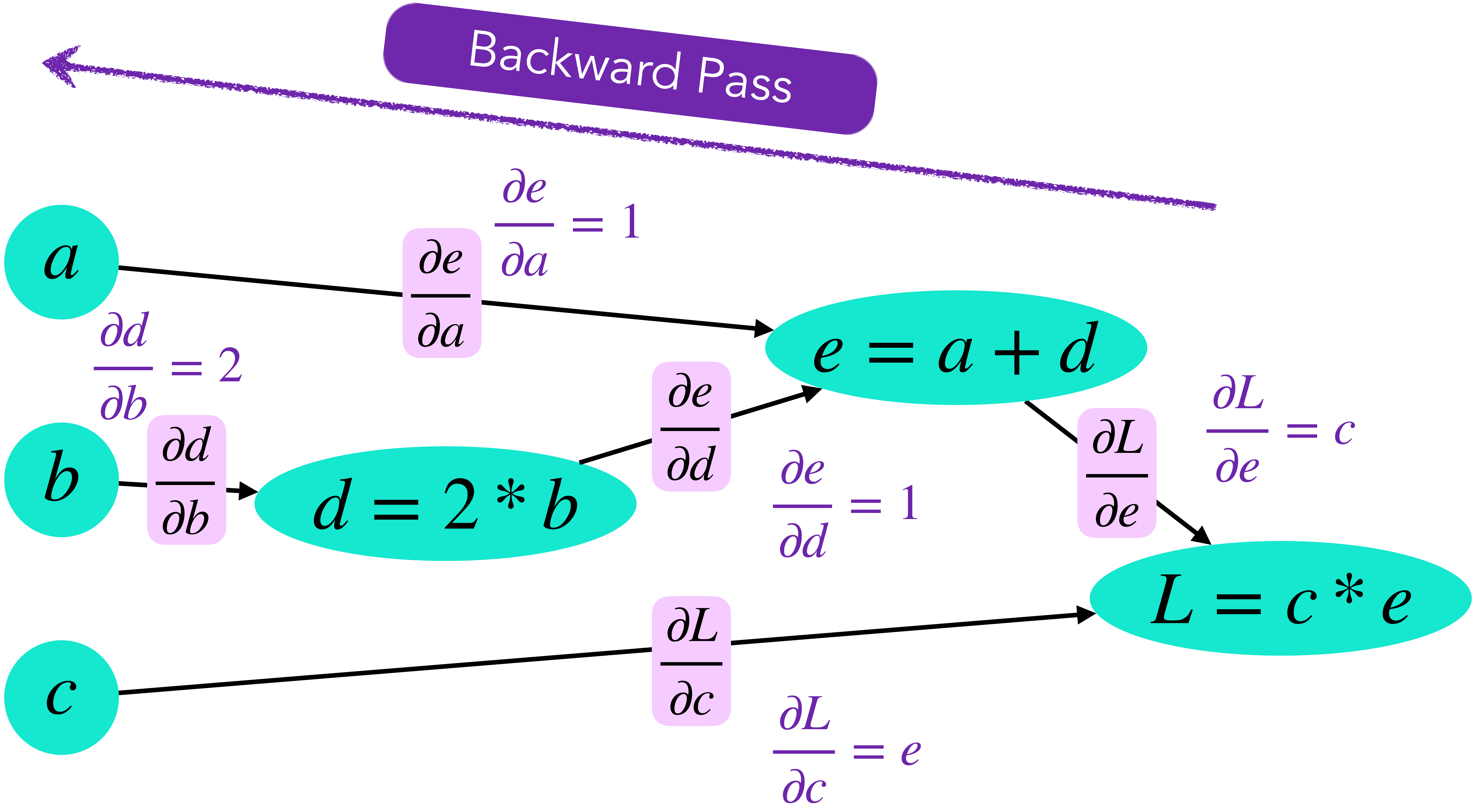
Example: Backward Pass



Example: Backward Pass

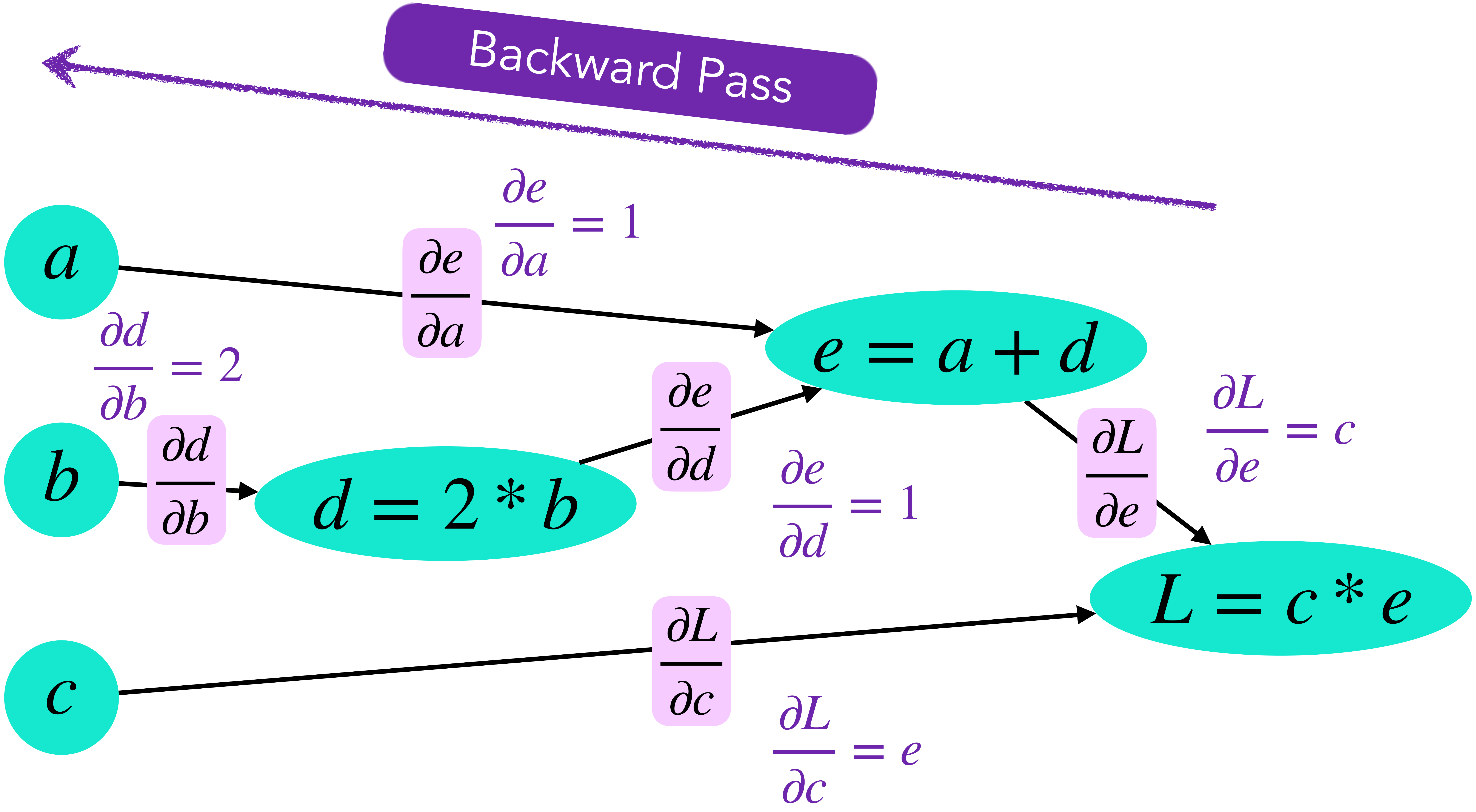


Example: Backward Pass



Example: Backward Pass

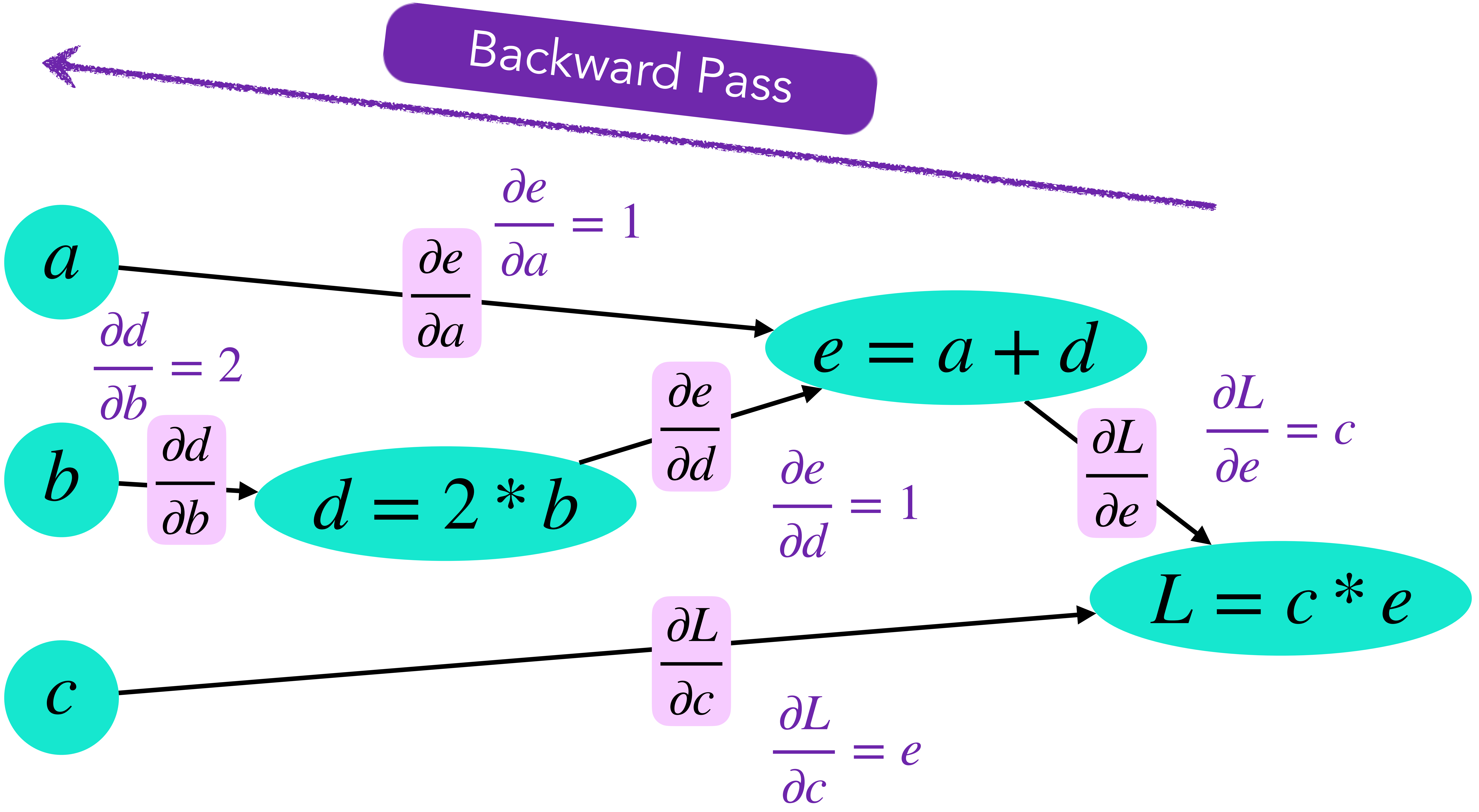
But we need the gradients of the loss with respect to parameters…



Example: Backward Pass

But we need the gradients of the loss with respect to parameters...

$$\frac{\partial L}{\partial c} = e \qquad \frac{\partial L}{\partial e} = c$$

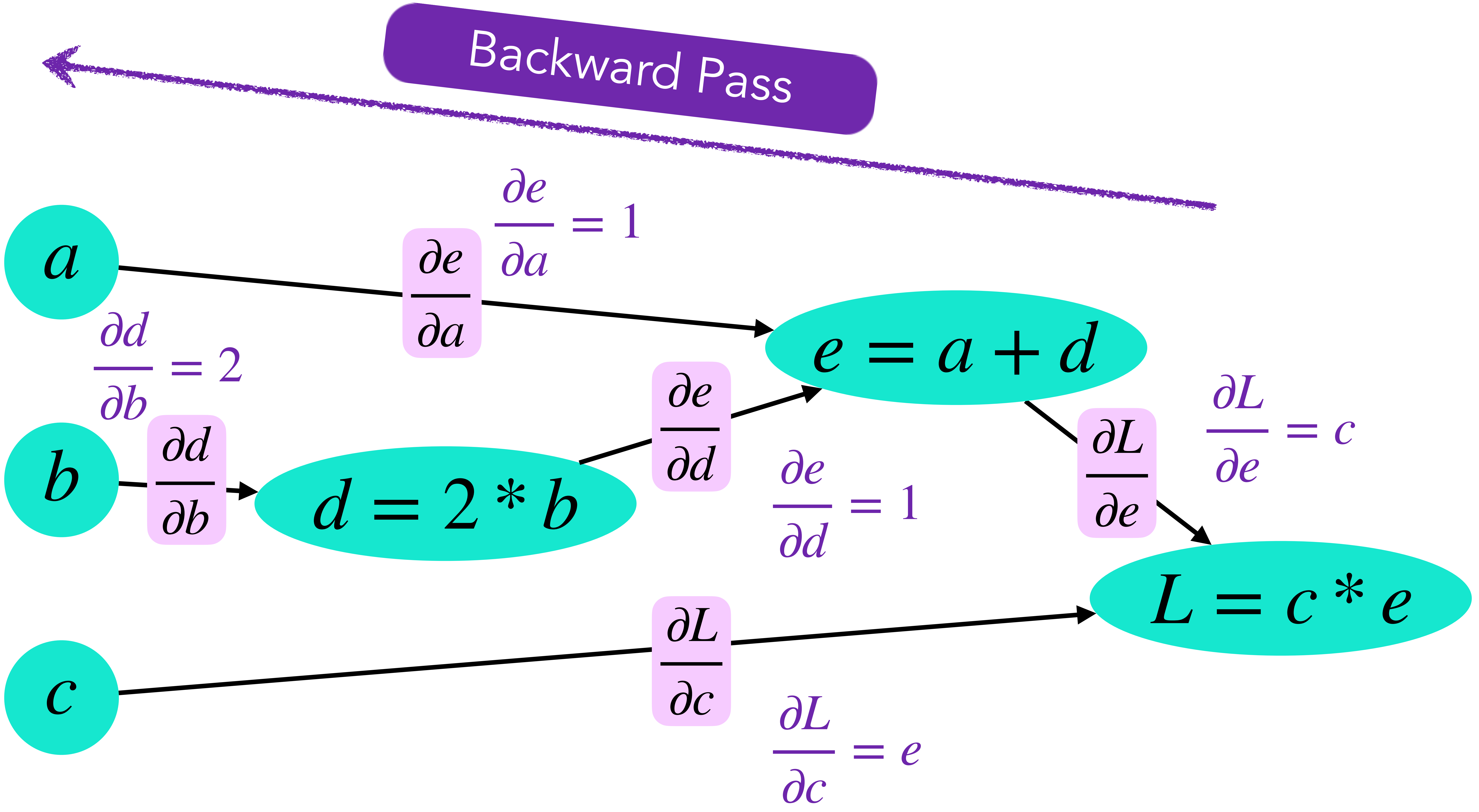


Example: Backward Pass

But we need the gradients of the loss with respect to parameters…

$$\frac{\partial L}{\partial c} = e \qquad \frac{\partial L}{\partial e} = c$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$



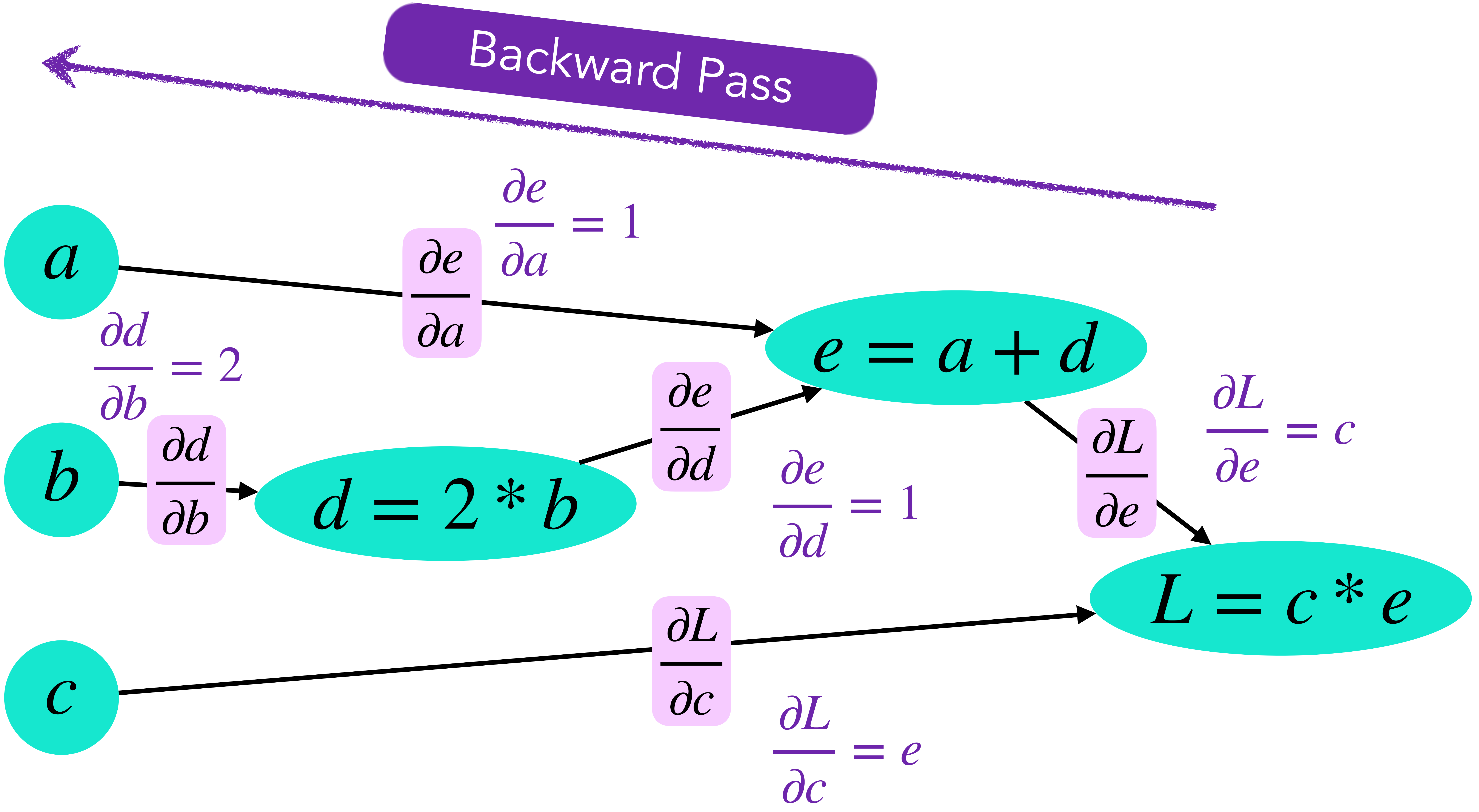
Example: Backward Pass

But we need the gradients of the loss with respect to parameters…

$$\frac{\partial L}{\partial c} = e \qquad \frac{\partial L}{\partial e} = c$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}$$



Example: Backward Pass

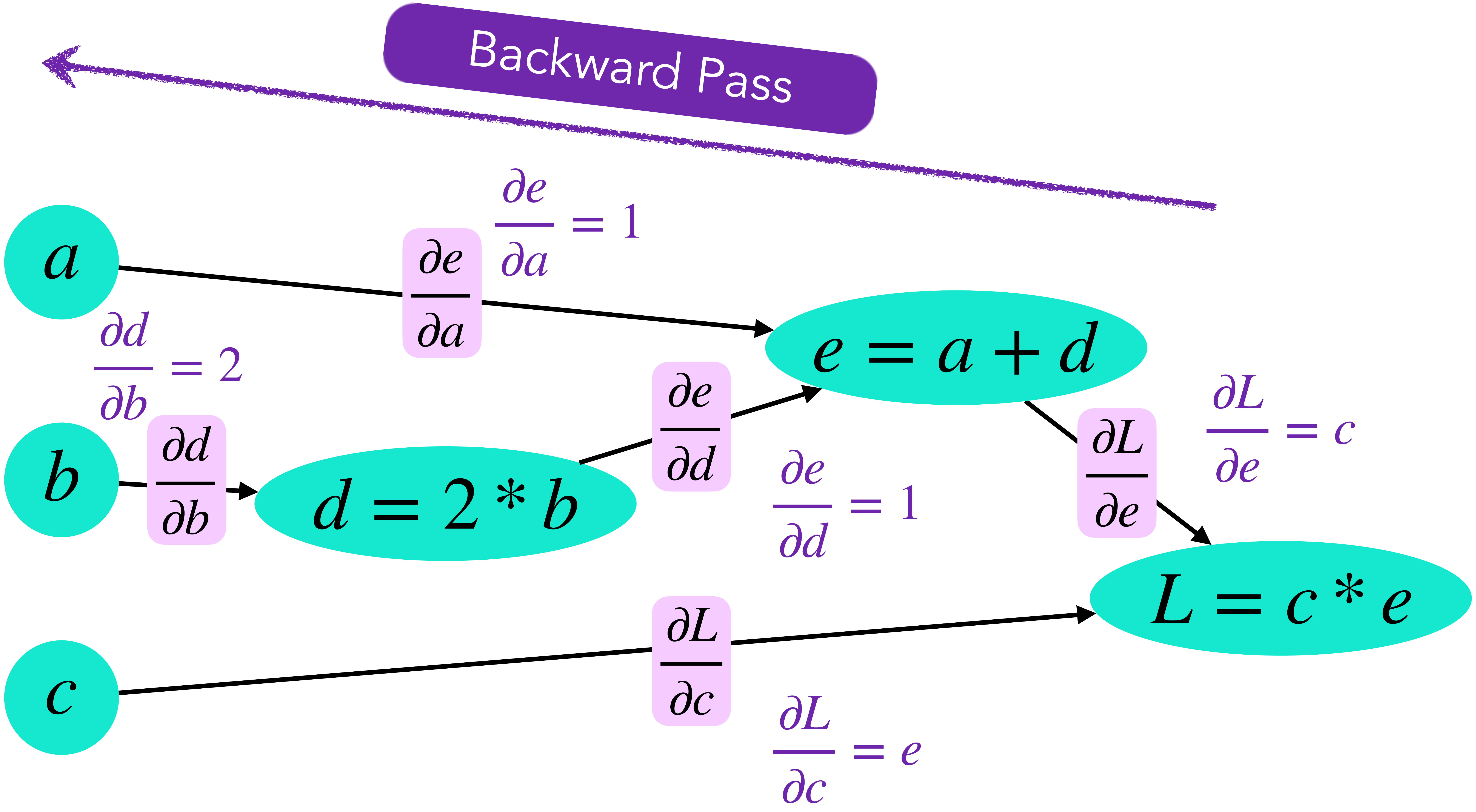
But we need the gradients of the loss with respect to parameters…

$$\frac{\partial L}{\partial c} = e \qquad \frac{\partial L}{\partial e} = c$$

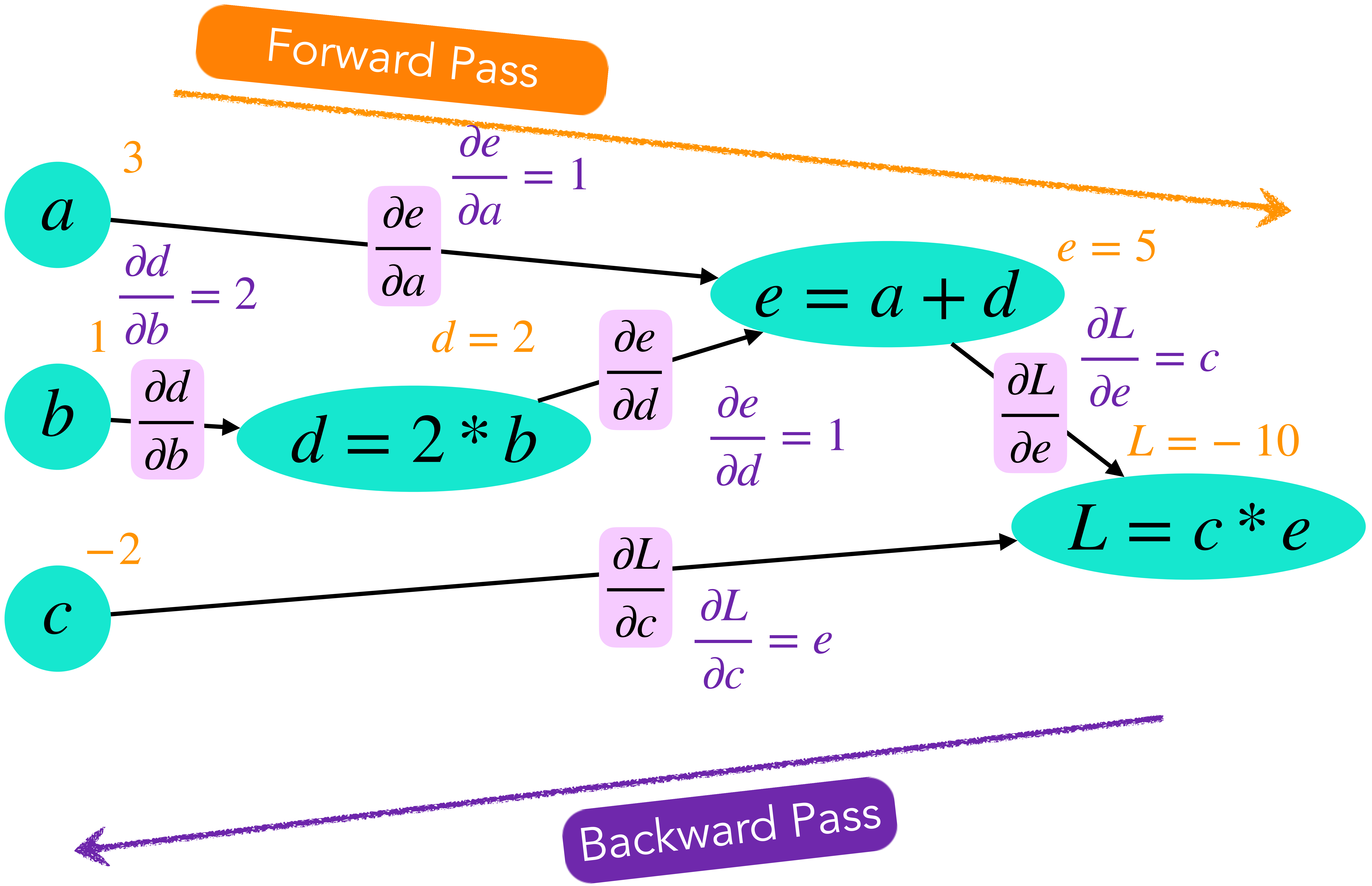
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b}$$



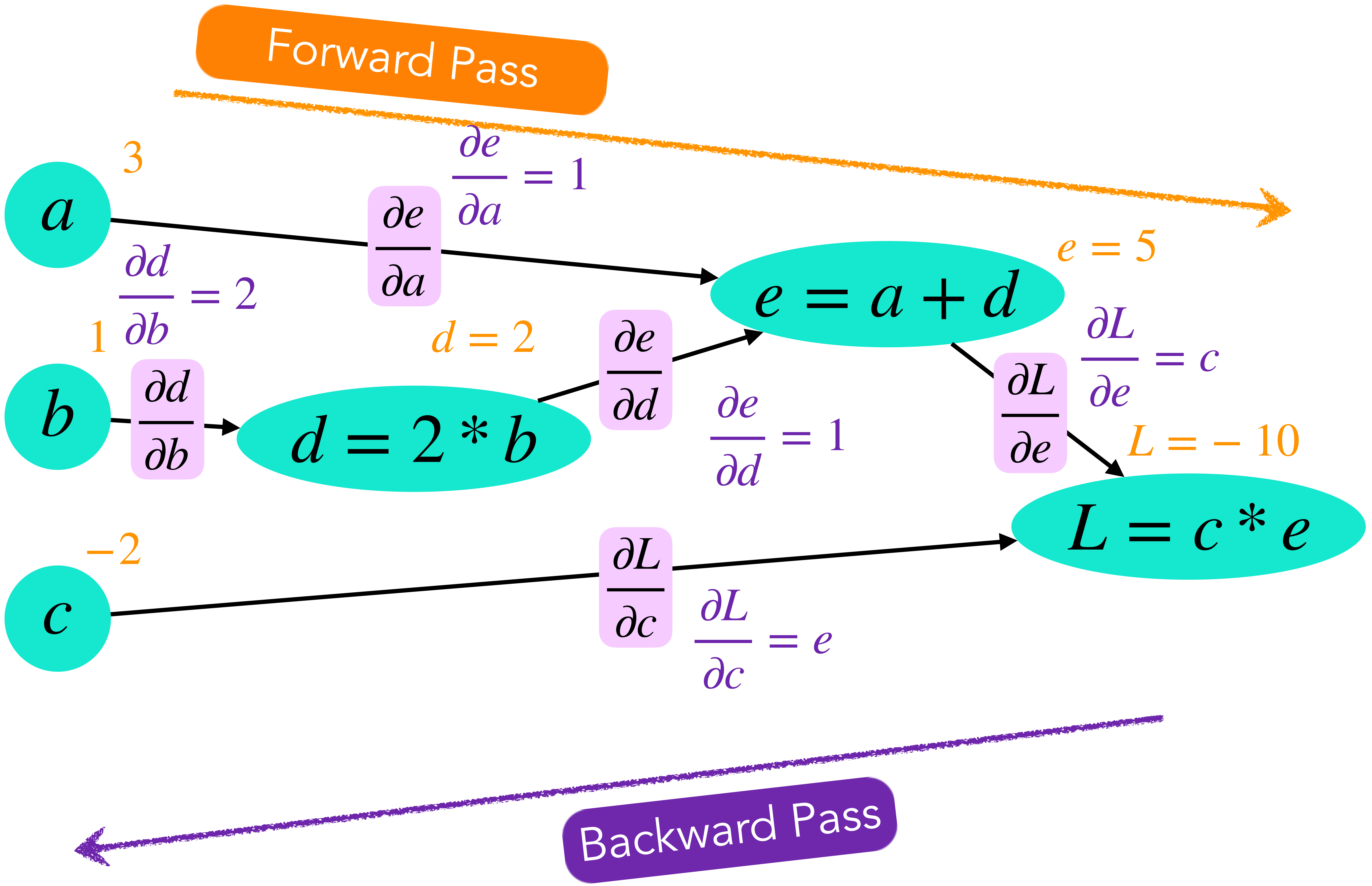
Example



Forward Pass
3
a
-2
c
1
b
$\frac{\partial e}{\partial a} = 1$
$\frac{\partial e}{\partial a}$
$\frac{\partial d}{\partial b} = 2$
$\frac{\partial d}{\partial b}$
$d = 2$
$d = 2 * b$
$\frac{\partial e}{\partial d}$
$\frac{\partial e}{\partial d} = 1$
$e = 5$
$e = a + d$
$\frac{\partial L}{\partial e} = c$
$\frac{\partial L}{\partial e}$
$L = -10$
$L = c * e$
$\frac{\partial L}{\partial c}$
$\frac{\partial L}{\partial c} = e$
Backward Pass

Example

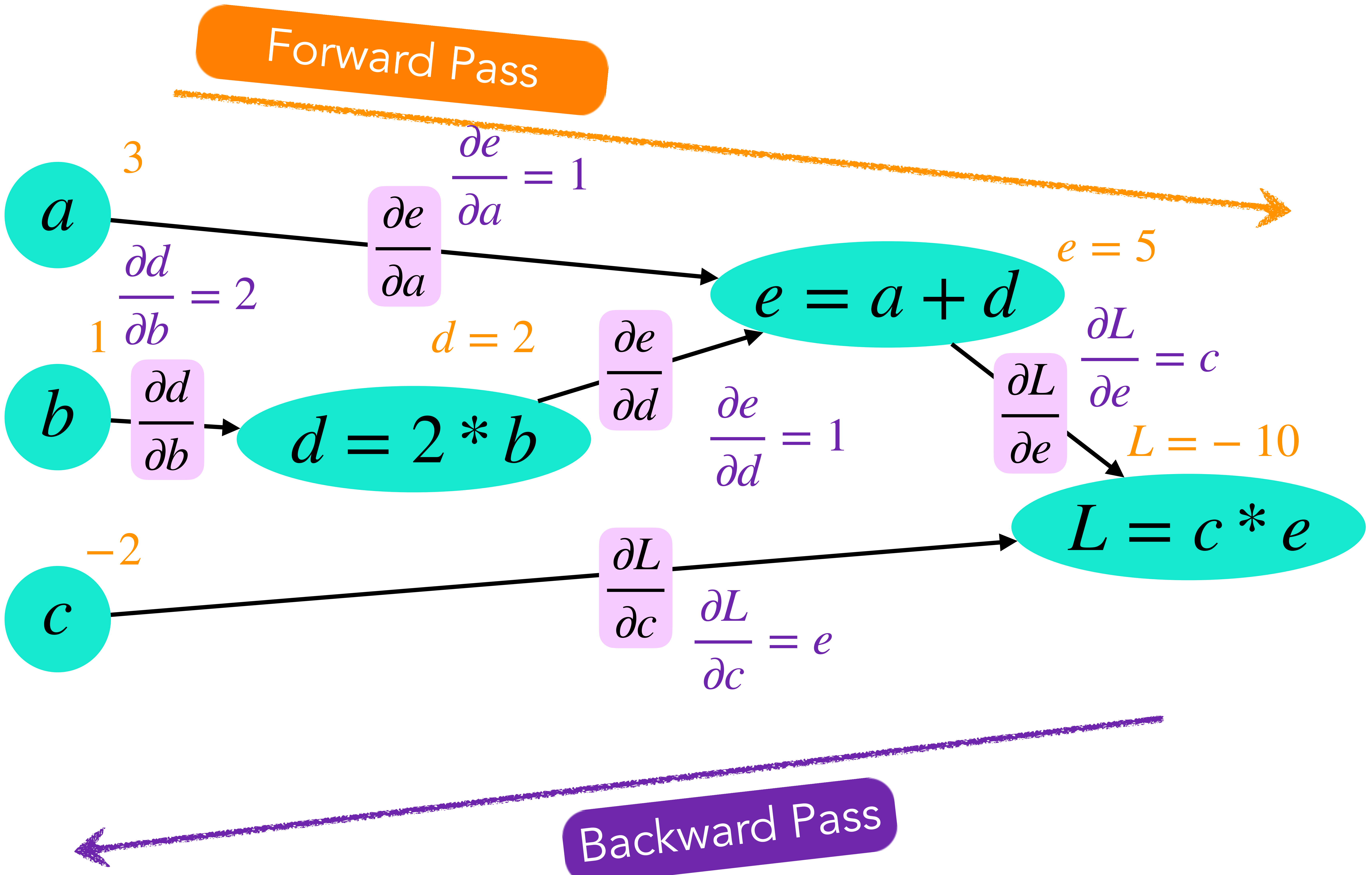
$$\frac{\partial L}{\partial e} = c = -2$$



Example

$$\frac{\partial L}{\partial e} = c = -2$$

$$\frac{\partial L}{\partial c} = e = 5$$

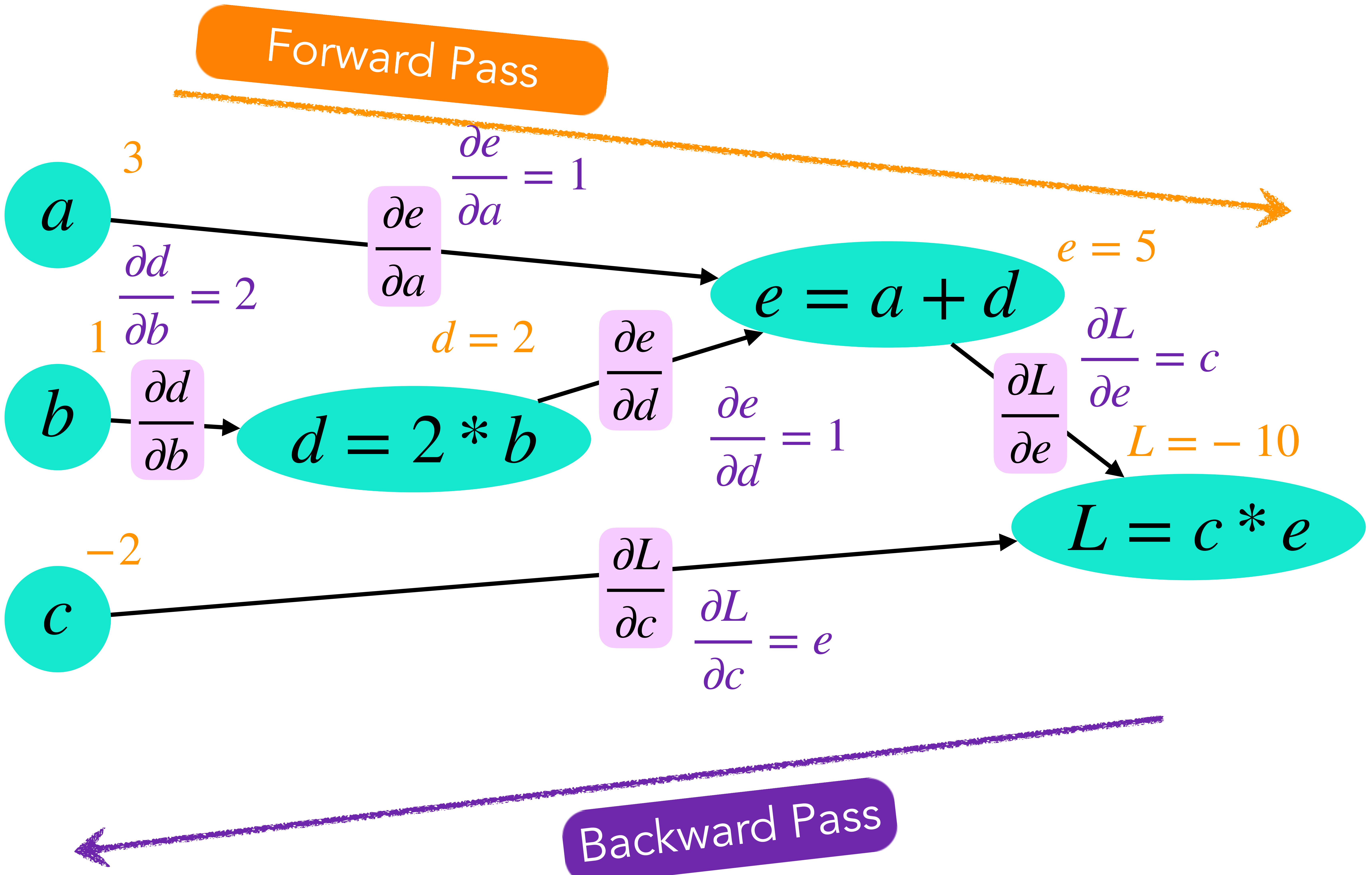


Example

$$\frac{\partial L}{\partial e} = c = -2$$

$$\frac{\partial L}{\partial c} = e = 5$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} = -2$$



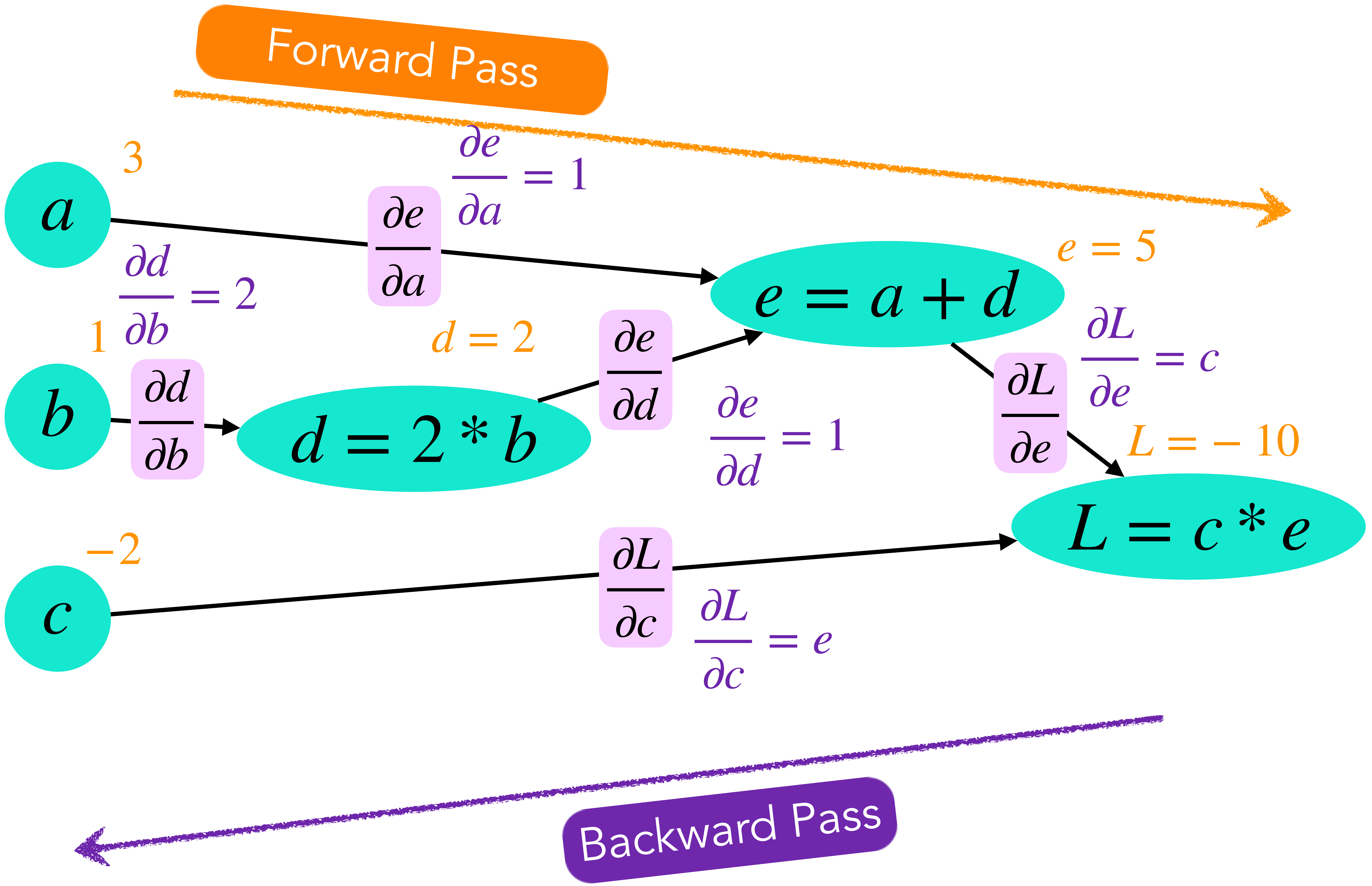
Example

$$\frac{\partial L}{\partial e} = c = -2$$

$$\frac{\partial L}{\partial c} = e = 5$$

$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} = -2$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d} = -2$$



Example

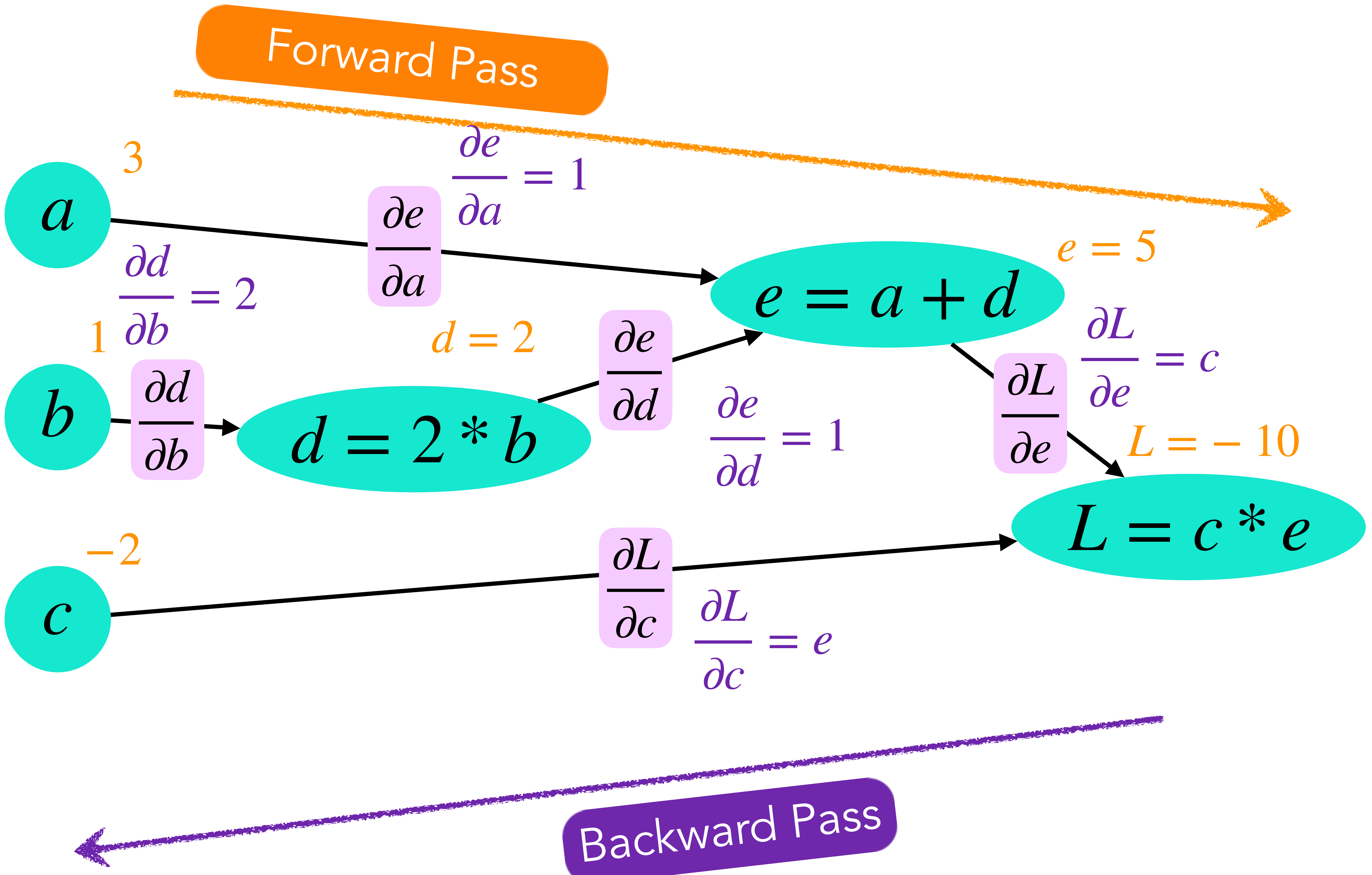
$$\frac{\partial L}{\partial e} = c = -2$$

$$\frac{\partial L}{\partial c} = e = 5$$

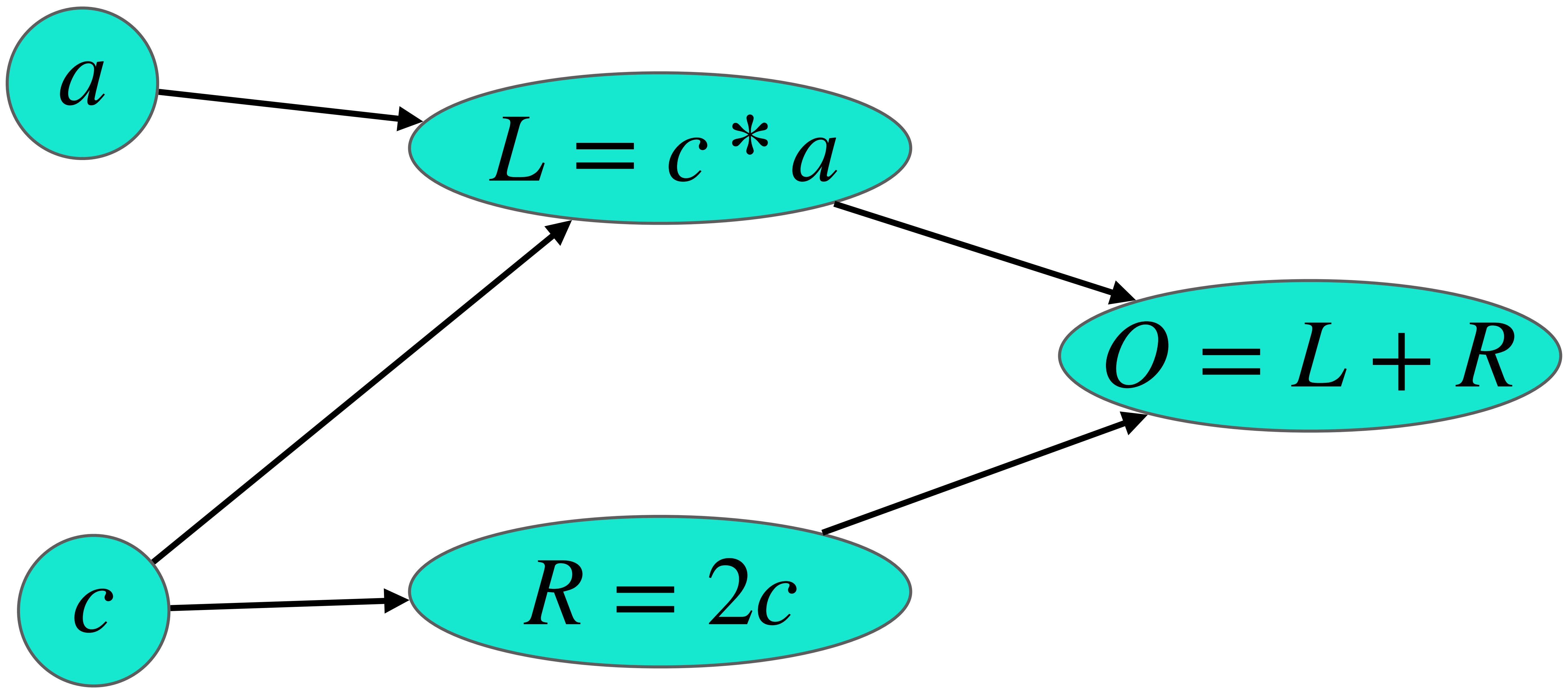
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} = -2$$

$$\frac{\partial L}{\partial d} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d} = -2$$

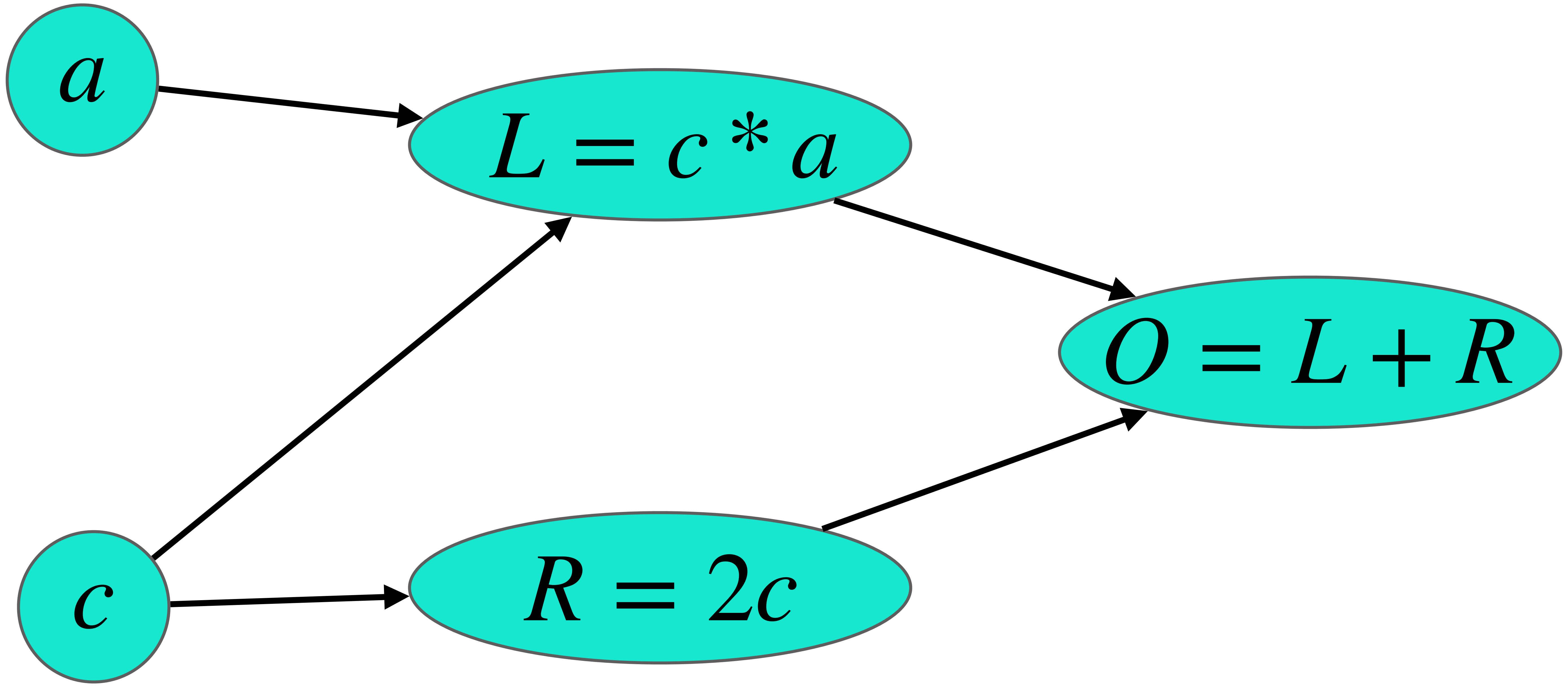
$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} = -4$$



Example: Two Paths

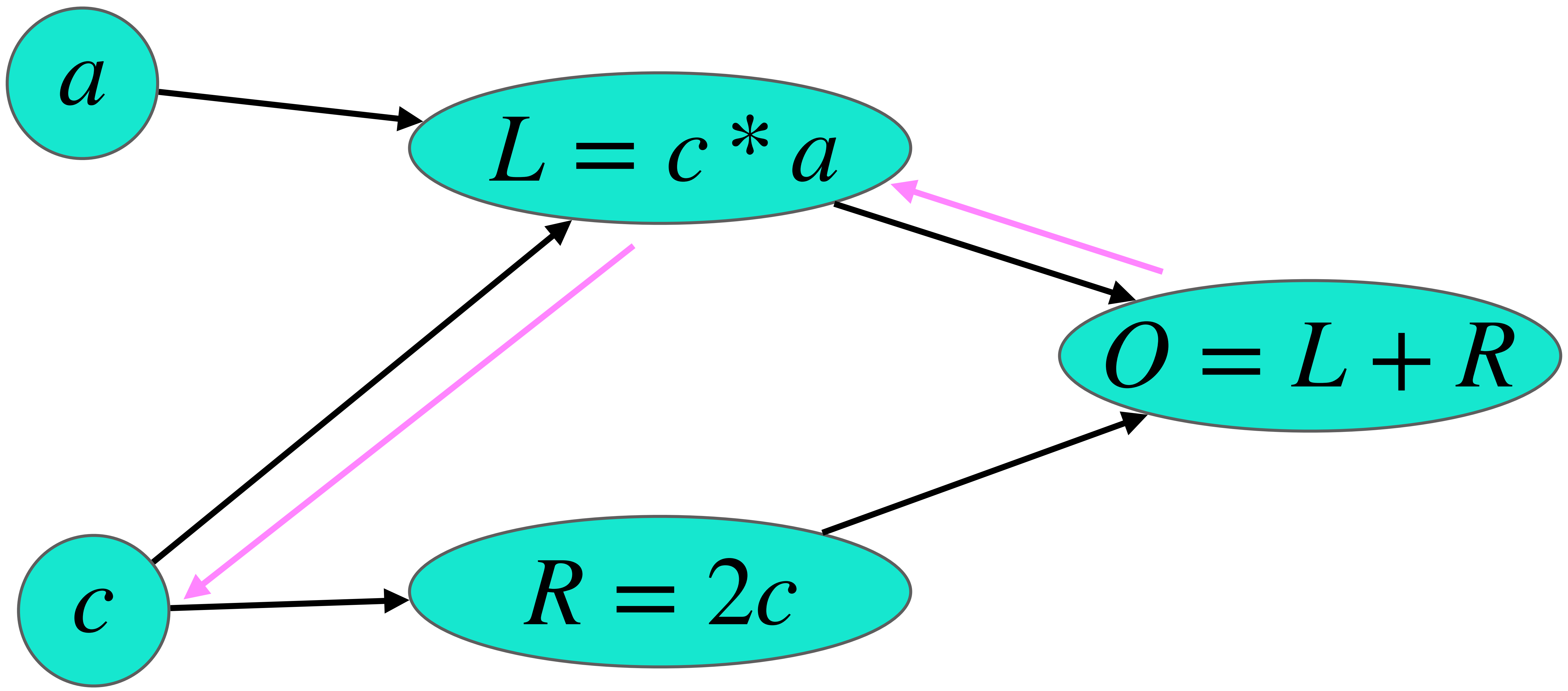


Example: Two Paths



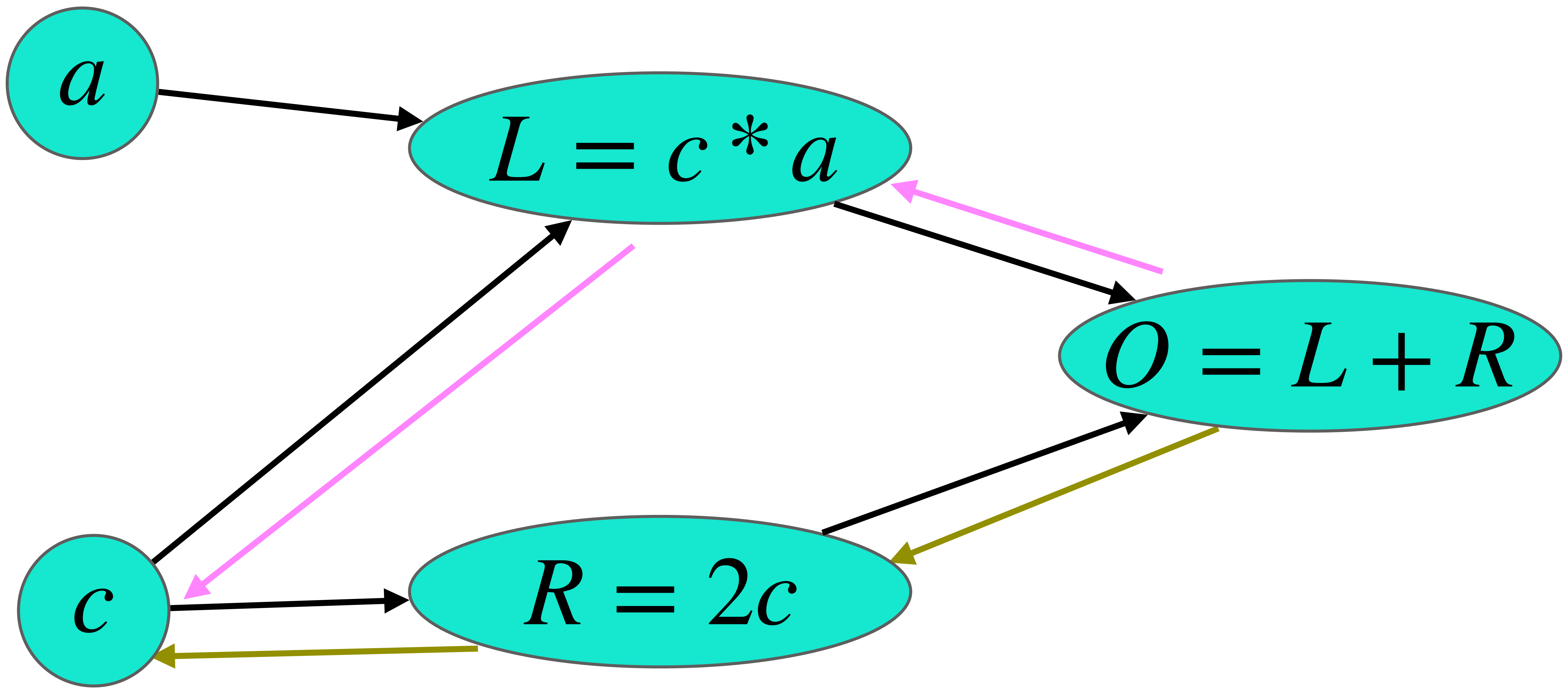
$$\frac{\partial O}{\partial c} =$$

Example: Two Paths



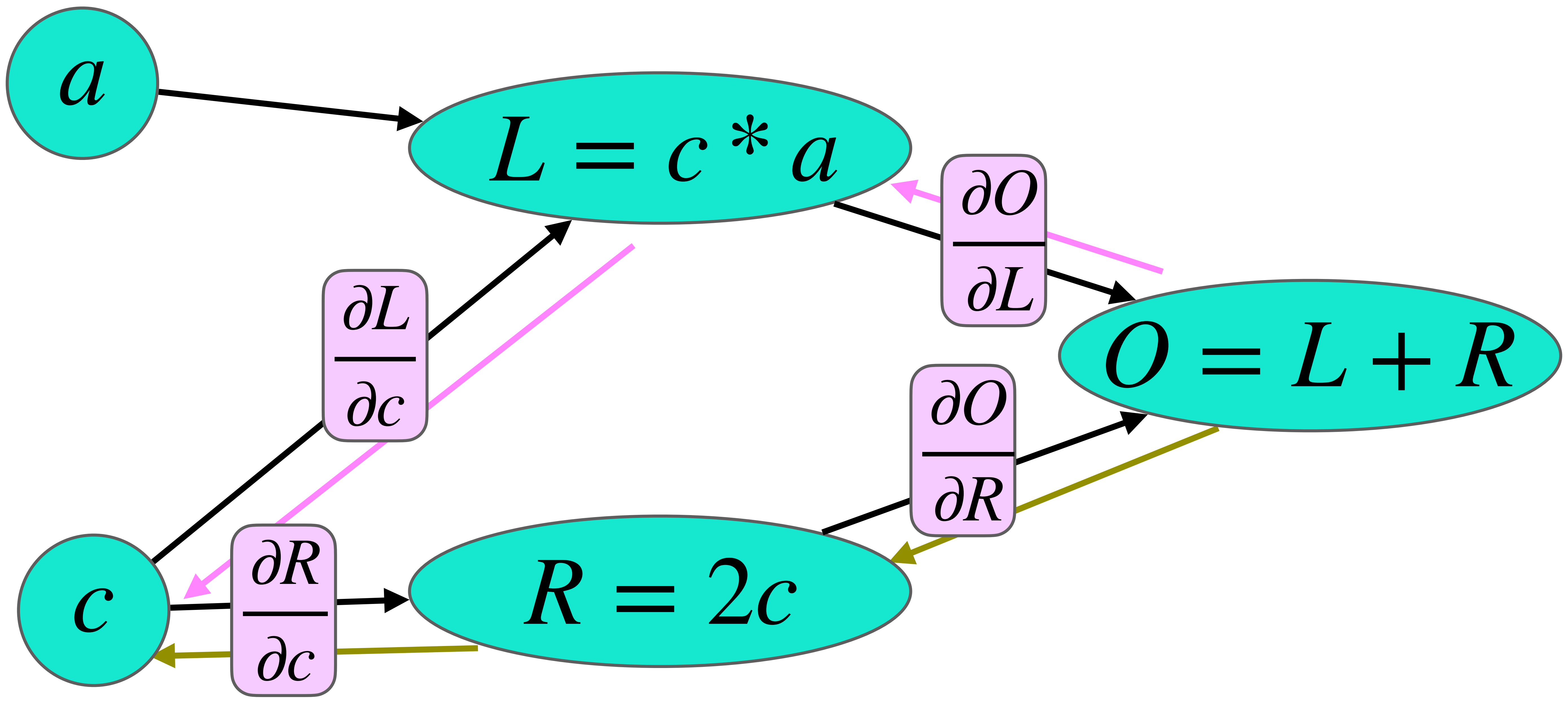
$$\frac{\partial O}{\partial c} =$$

Example: Two Paths



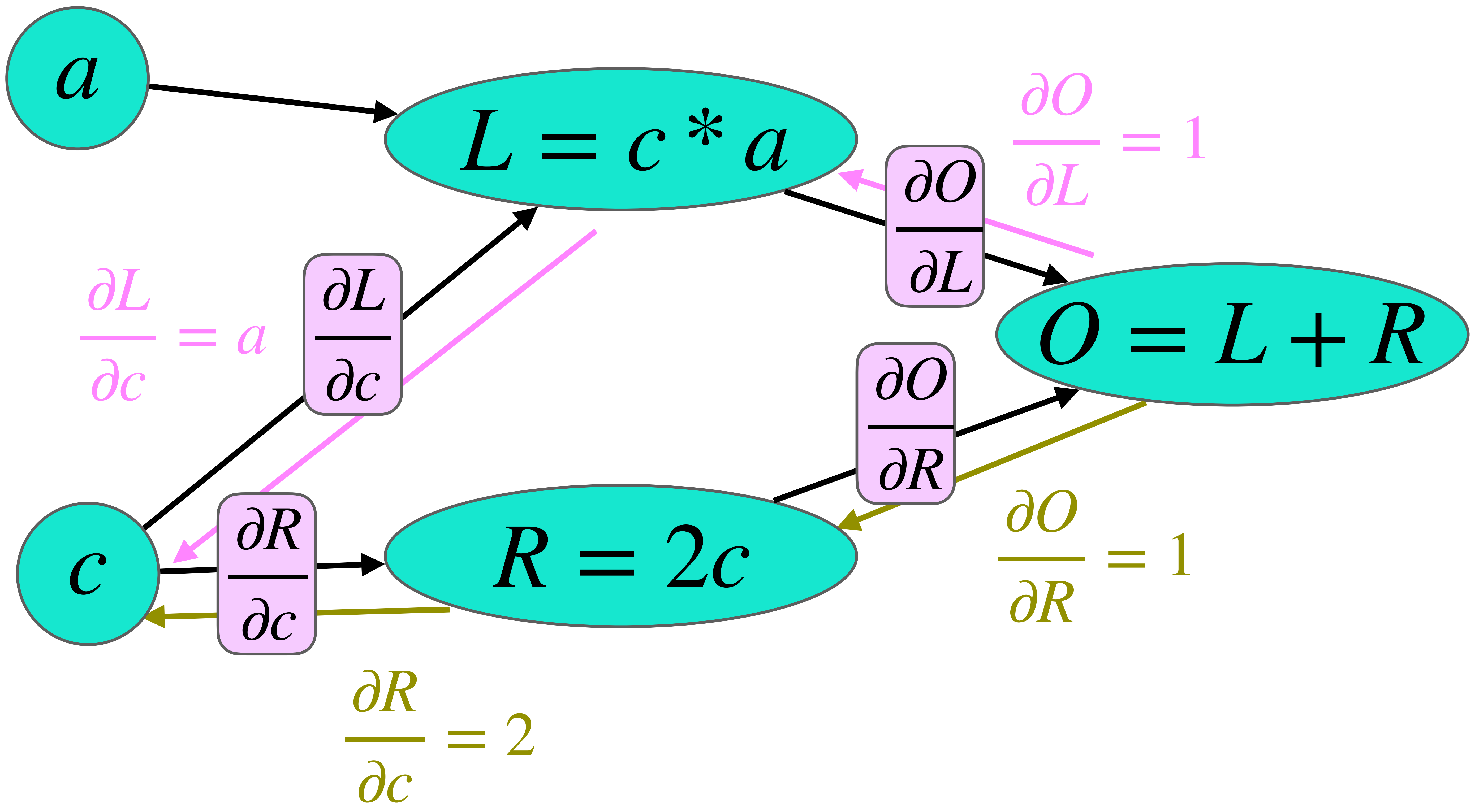
$$\frac{\partial O}{\partial c} =$$

Example: Two Paths



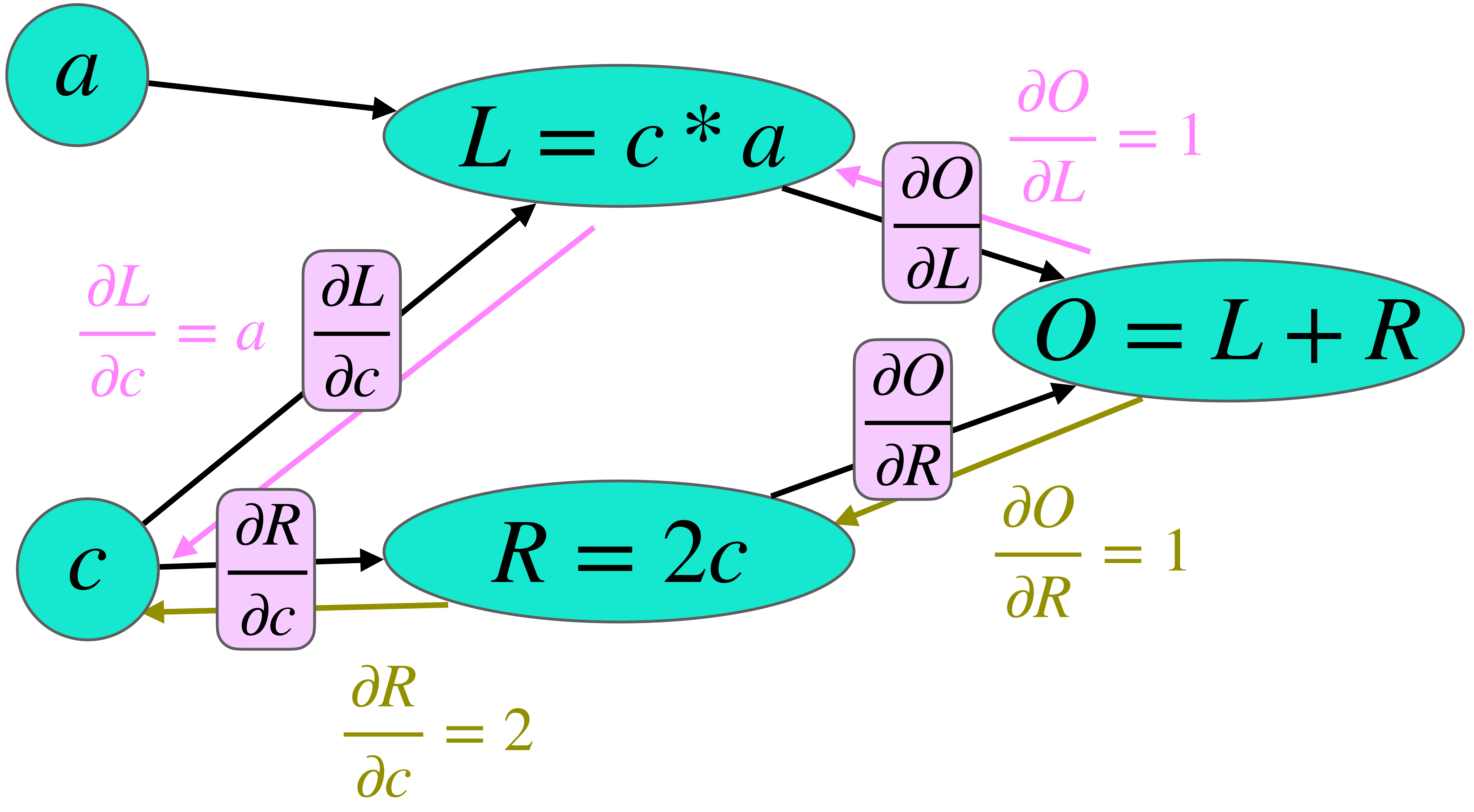
$$\frac{\partial O}{\partial c} =$$

Example: Two Paths



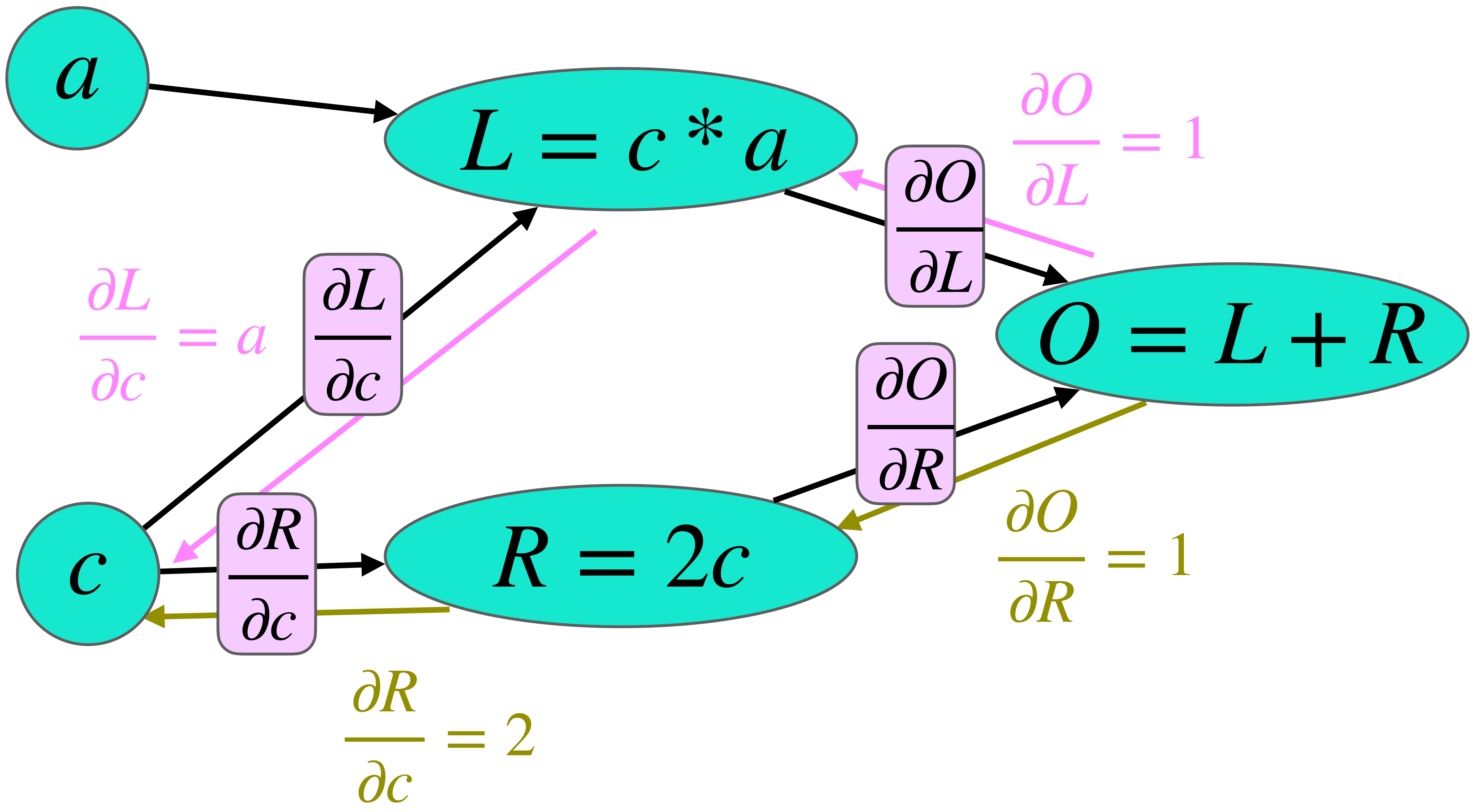
$$\frac{\partial O}{\partial c} =$$

Example: Two Paths



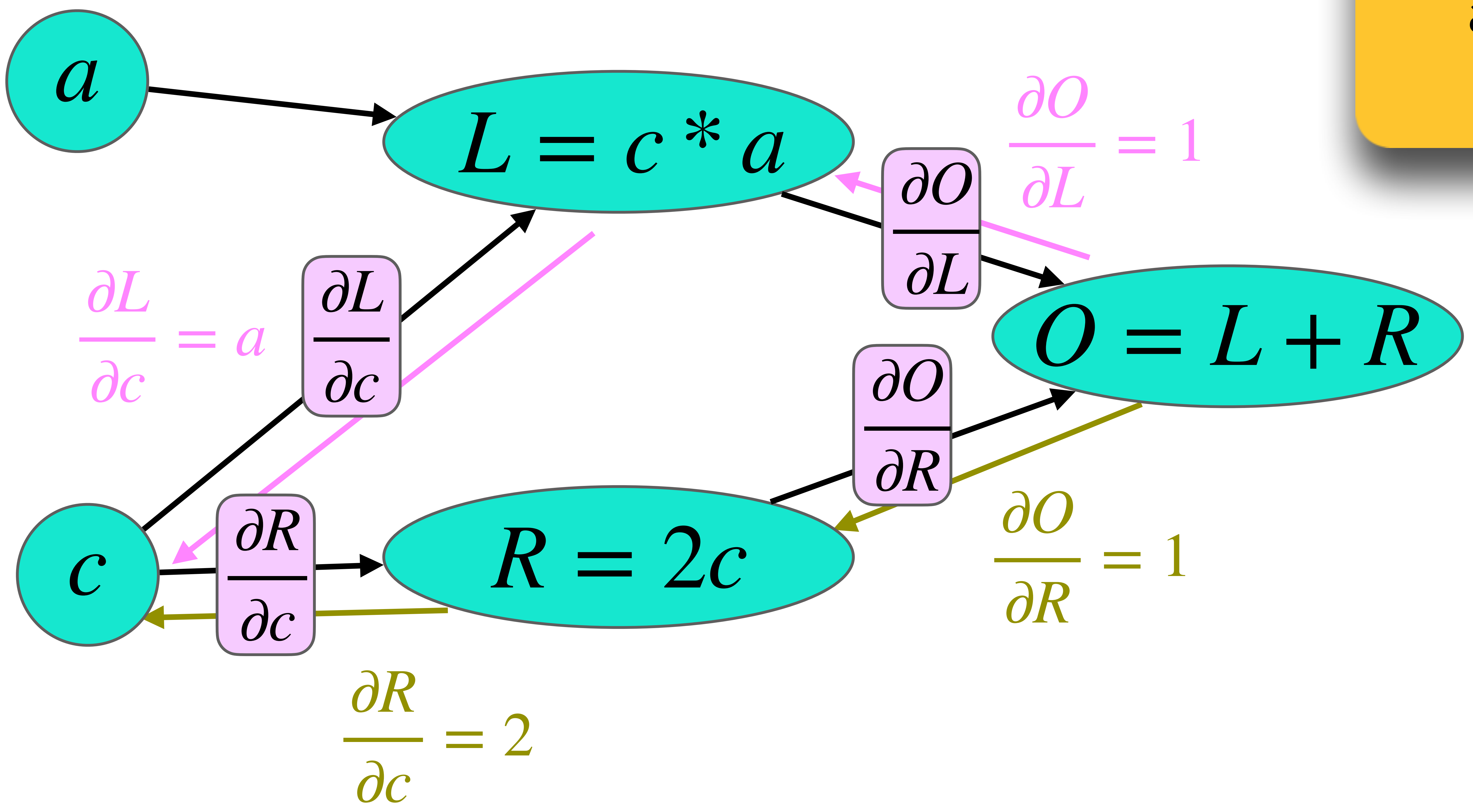
$$\frac{\partial O}{\partial c} = \frac{\partial O}{\partial L}\frac{\partial L}{\partial c}$$

Example: Two Paths



$$\frac{\partial O}{\partial c} = \frac{\partial O}{\partial L}\frac{\partial L}{\partial c} + \frac{\partial O}{\partial R}\frac{\partial R}{\partial c}$$

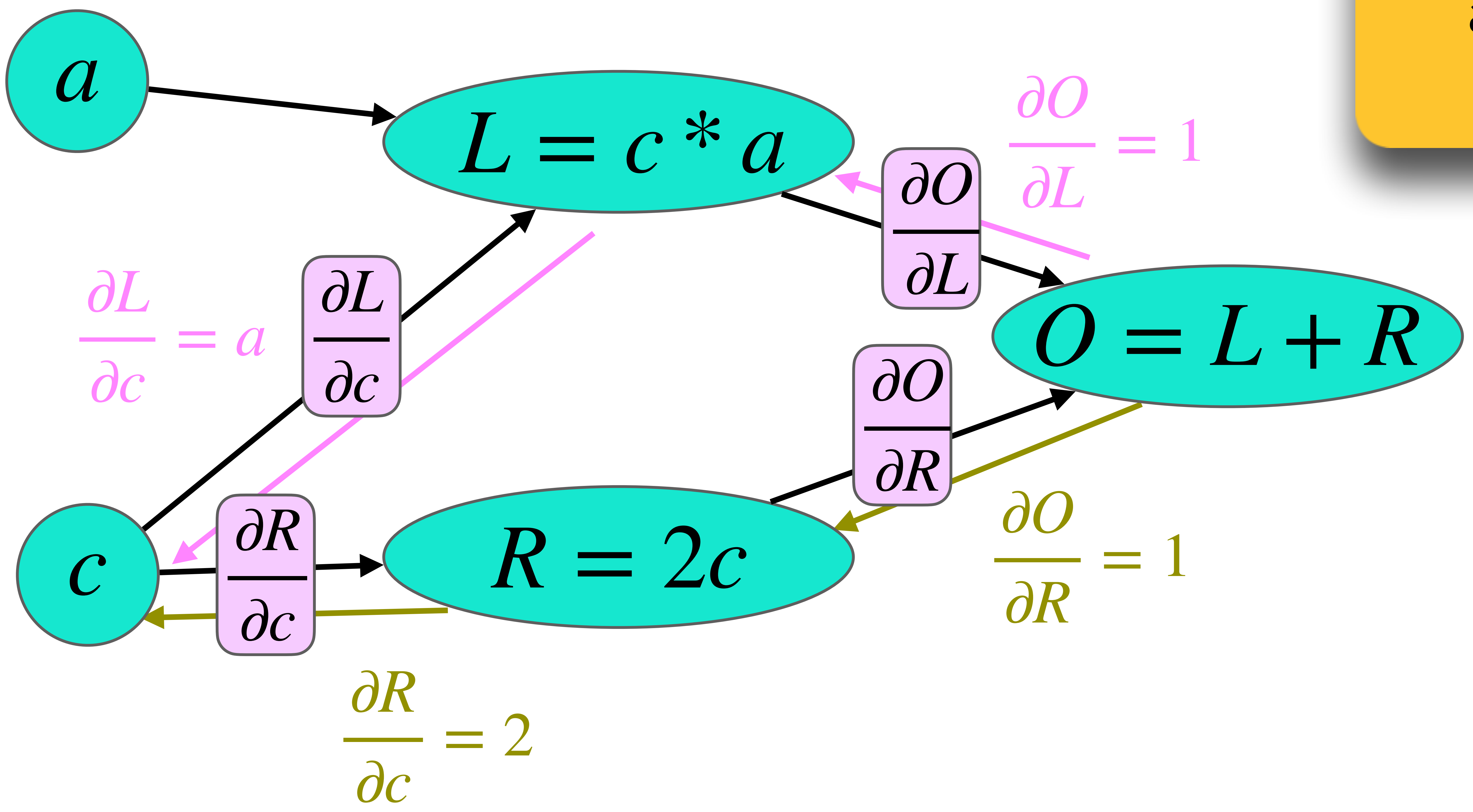
Example: Two Paths



When multiple branches converge on a single node we will add these branches

$$\frac{\partial O}{\partial c} = \frac{\partial O}{\partial L}\frac{\partial L}{\partial c} + \frac{\partial O}{\partial R}\frac{\partial R}{\partial c}$$

Example: Two Paths

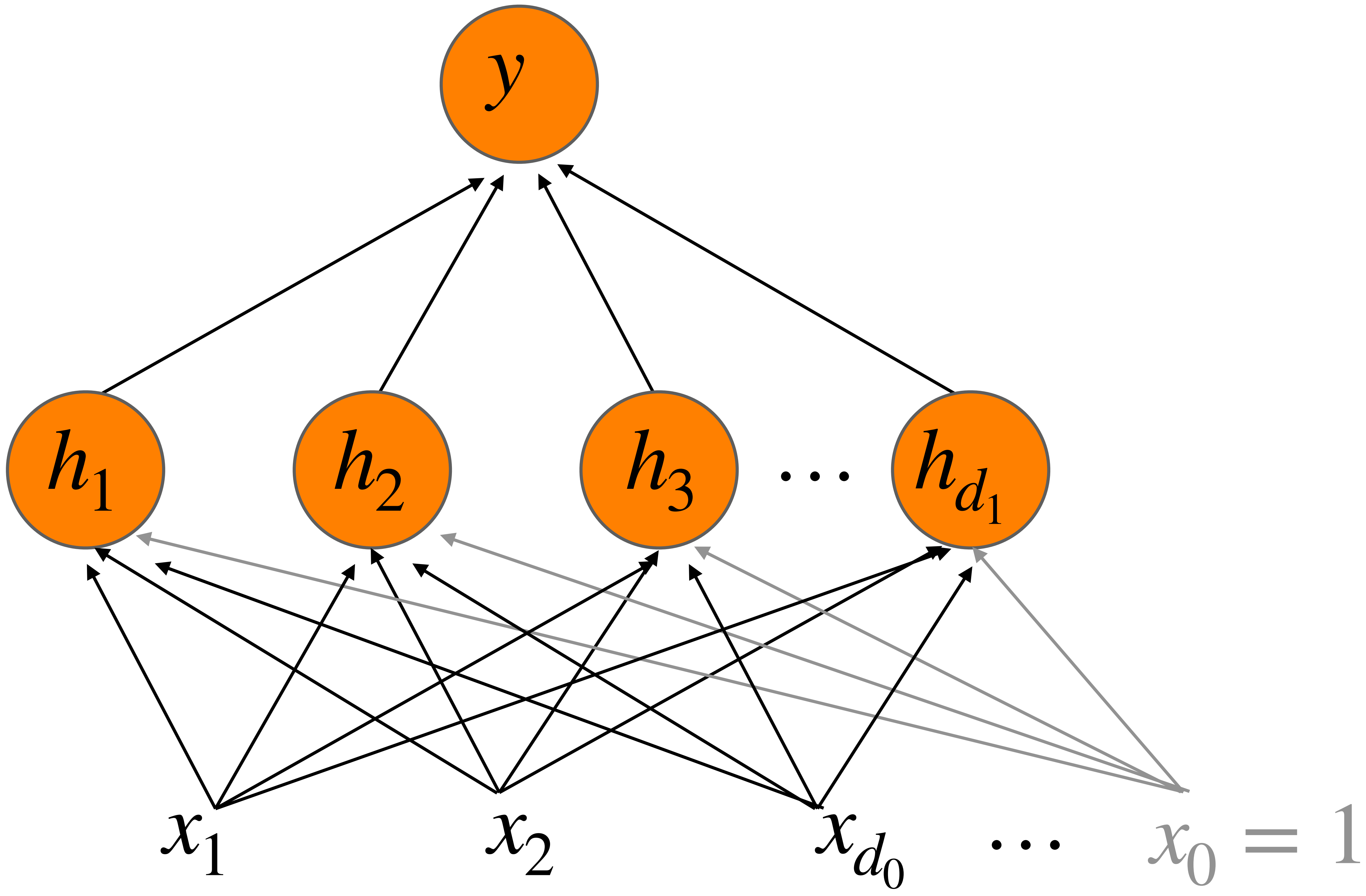


When multiple branches converge on a single node we will add these branches

$$\frac{\partial O}{\partial c} = \frac{\partial O}{\partial L}\frac{\partial L}{\partial c} + \frac{\partial O}{\partial R}\frac{\partial R}{\partial c}$$

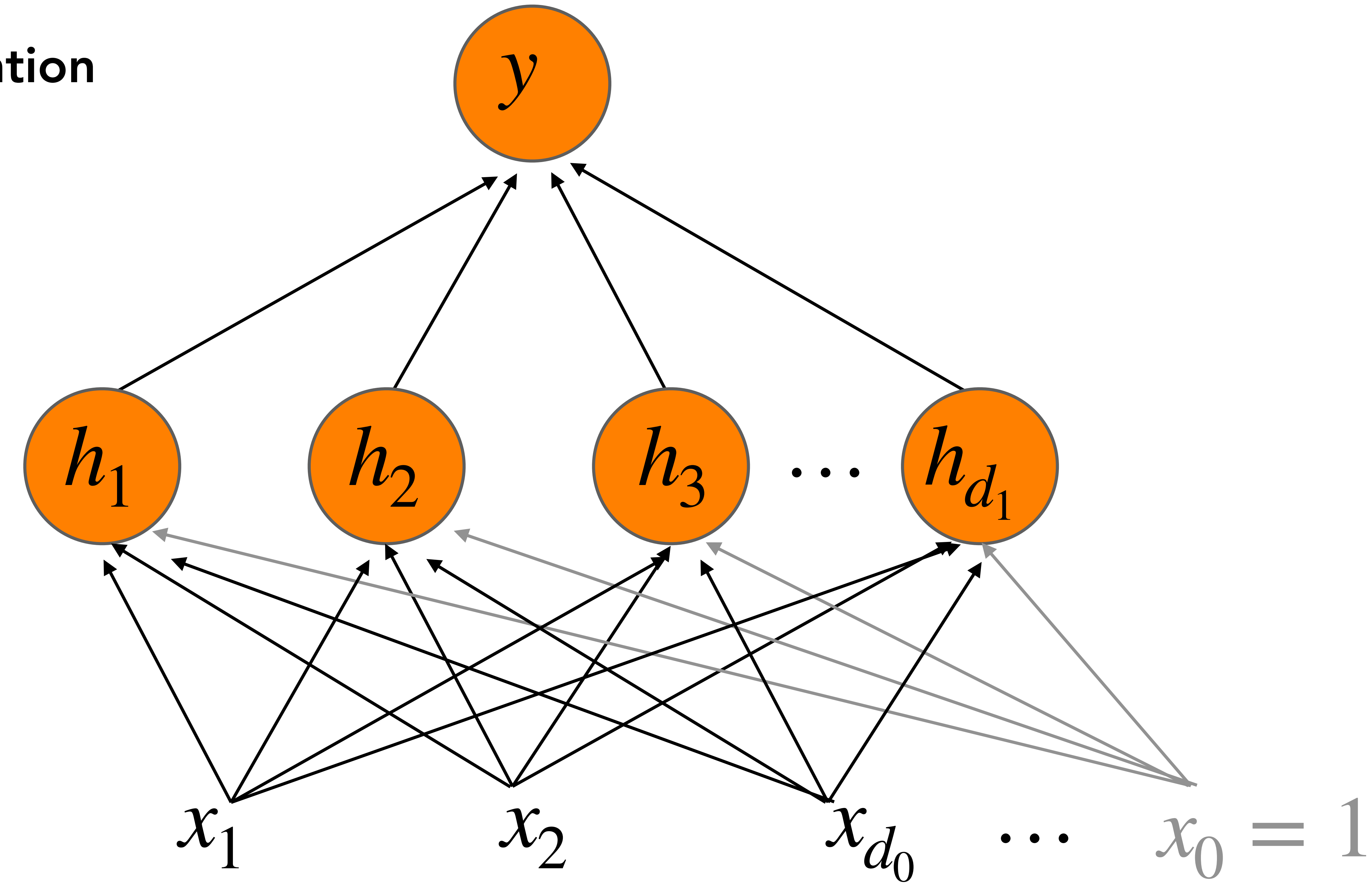
Such cases arise when considering regularized loss functions

Backward Differentiation on a 2-layer MLP

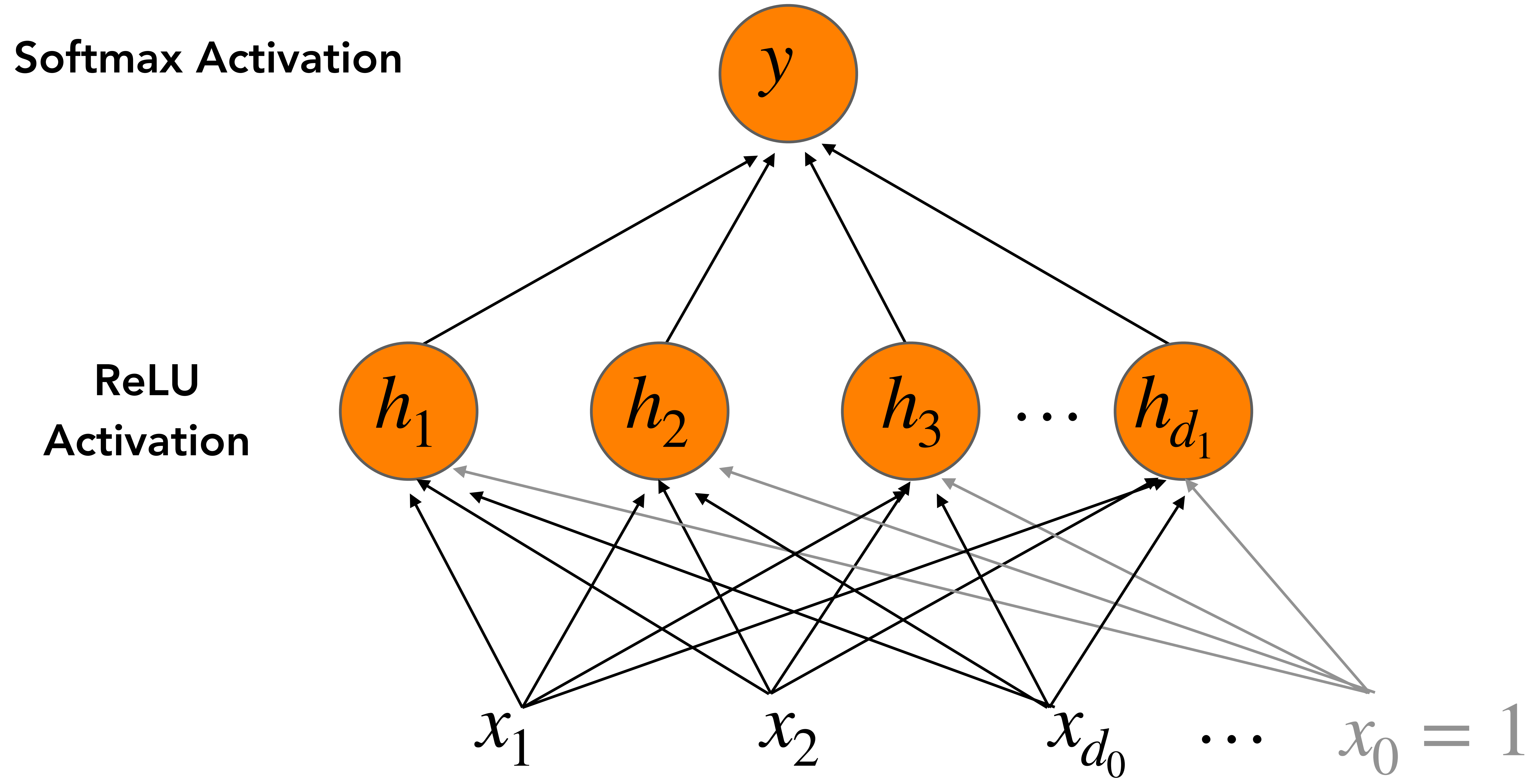


Backward Differentiation on a 2-layer MLP

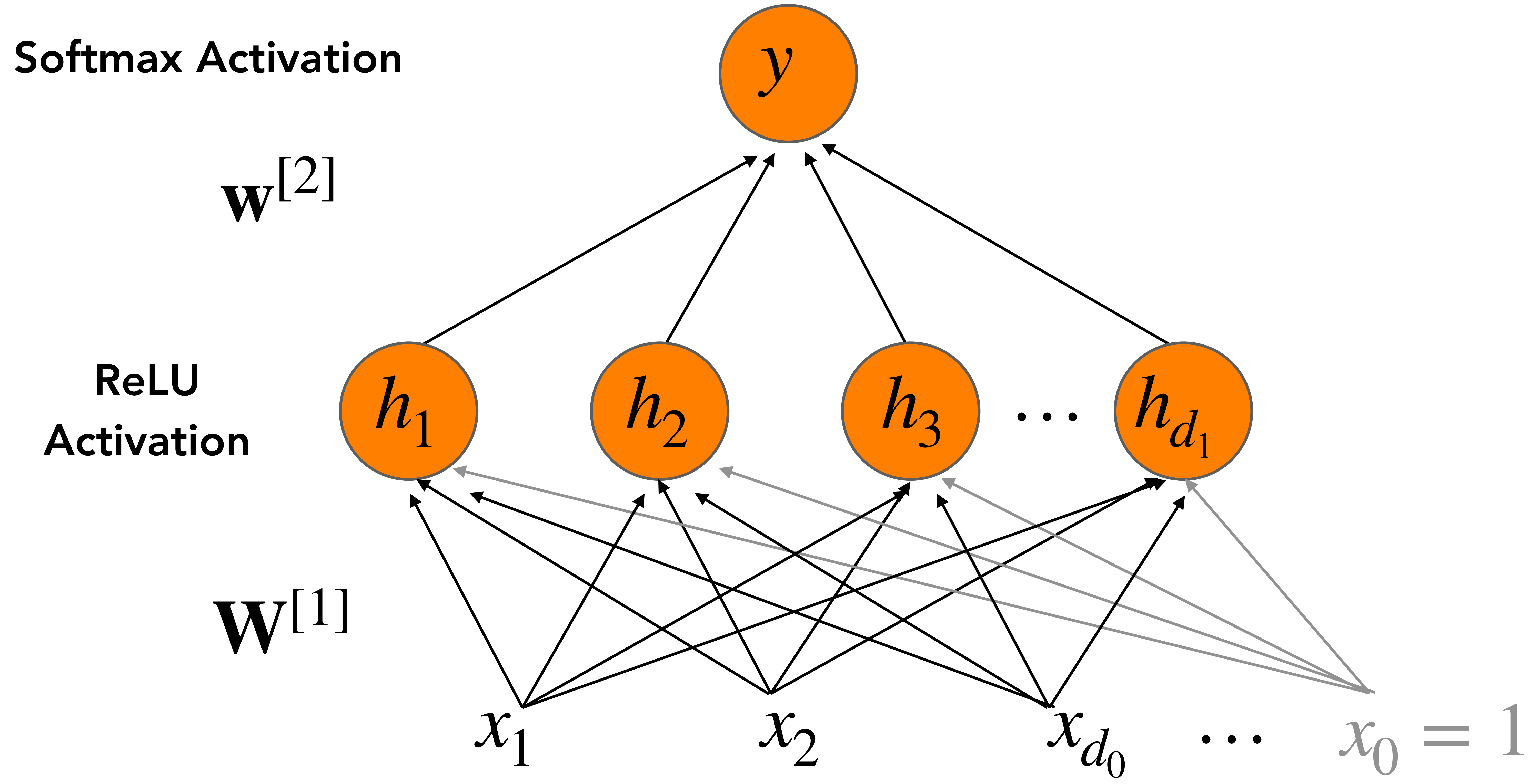
Softmax Activation



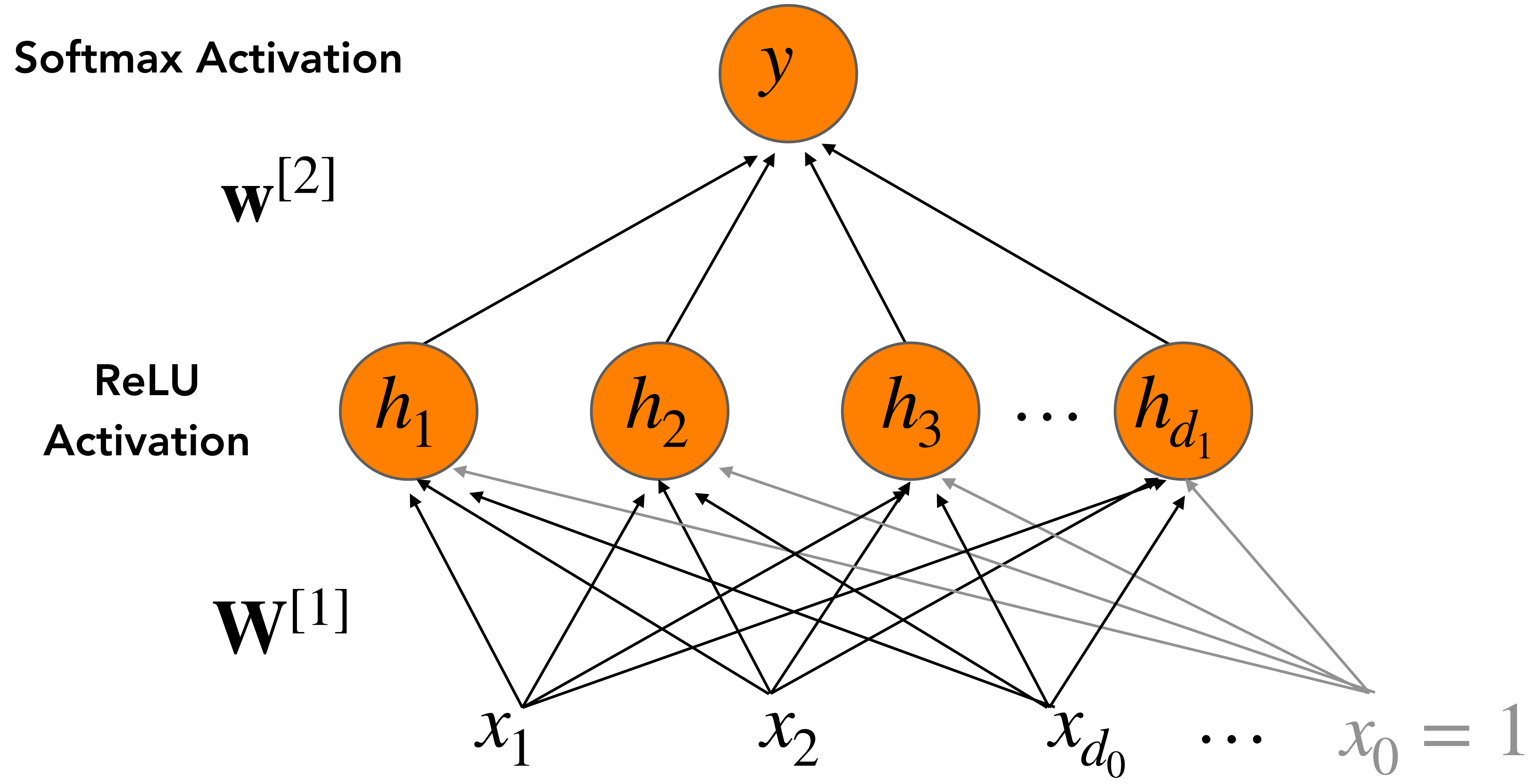
Backward Differentiation on a 2-layer MLP



Backward Differentiation on a 2-layer MLP

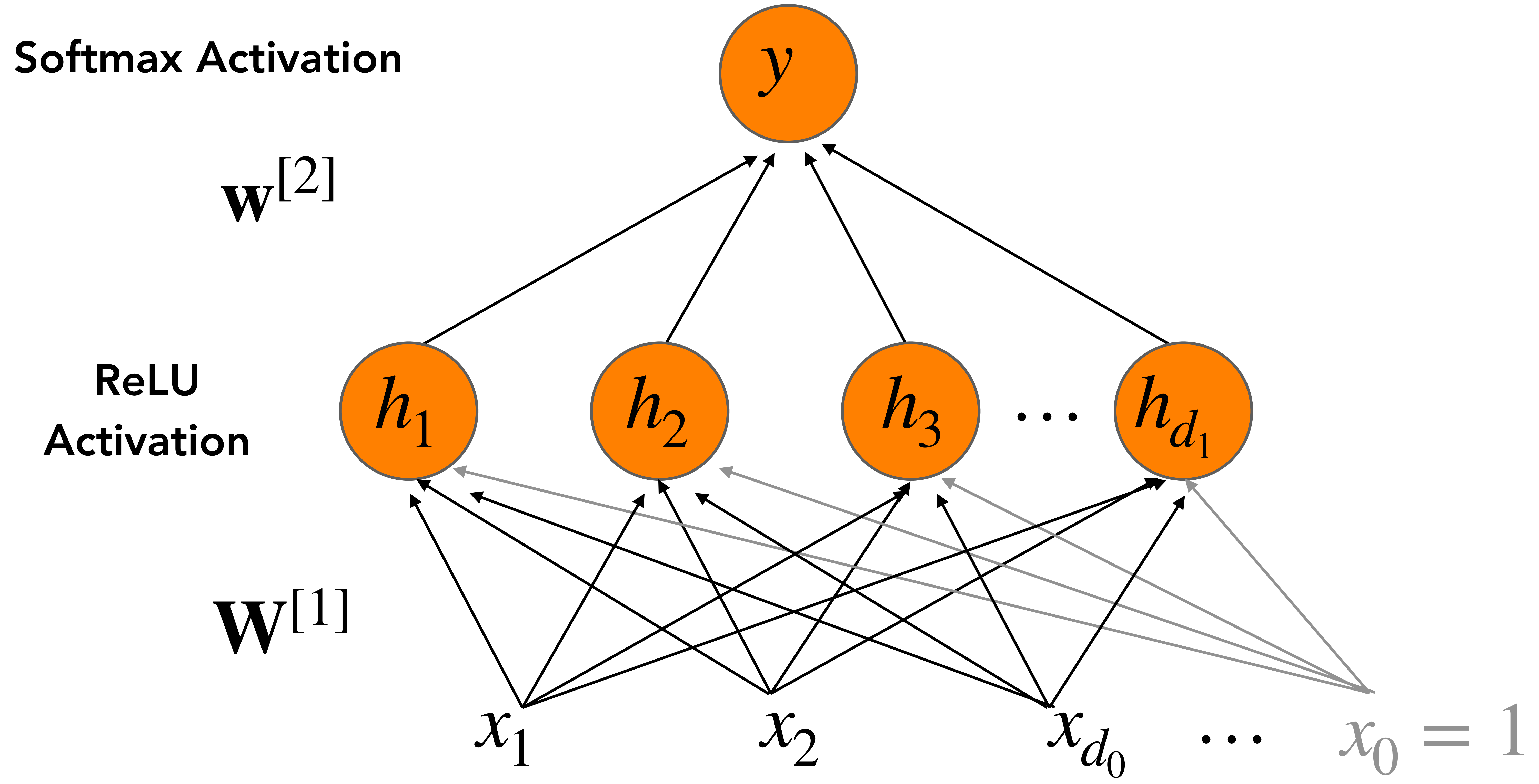


Backward Differentiation on a 2-layer MLP



$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}$$

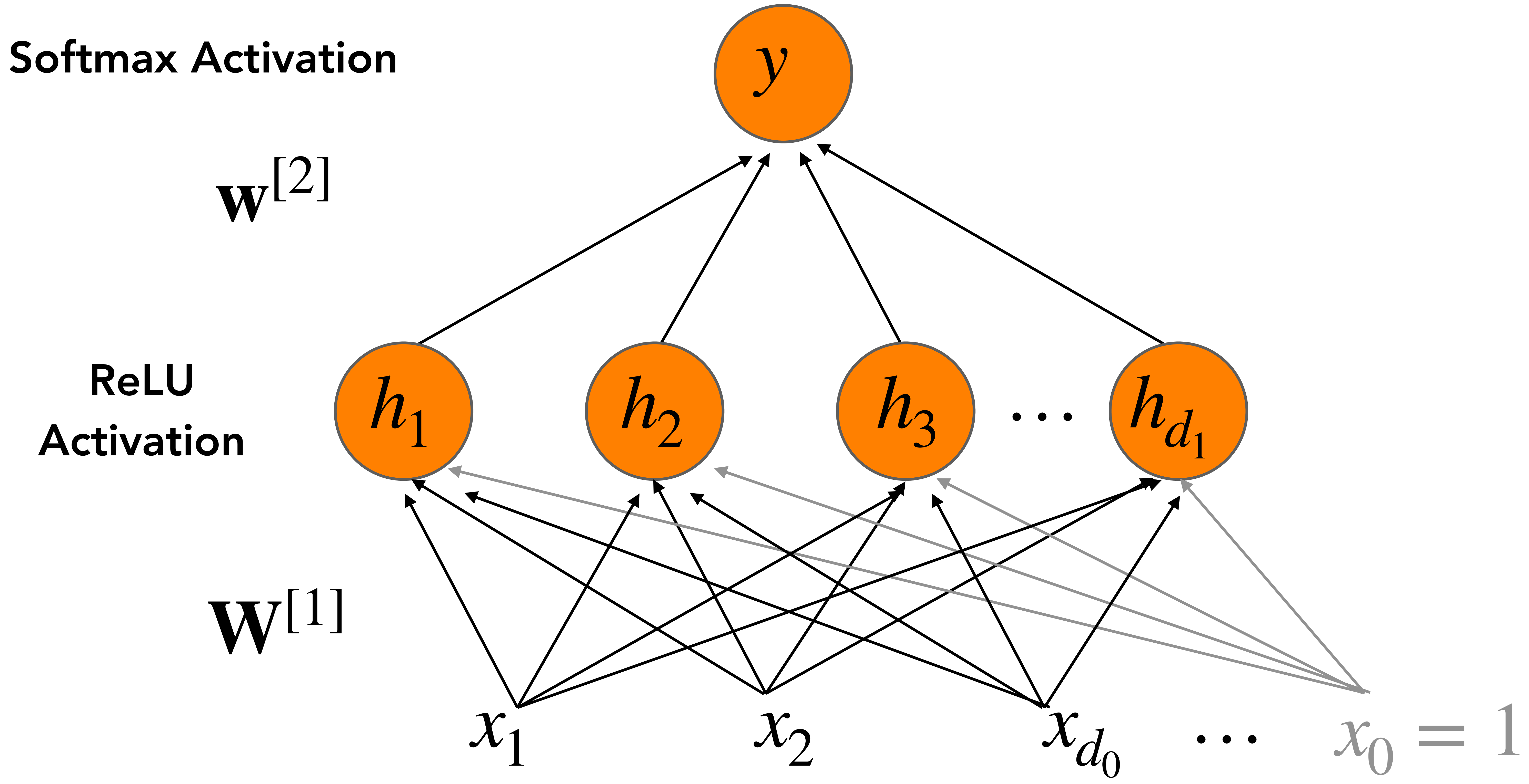
Backward Differentiation on a 2-layer MLP



$$\mathbf{h}^{[1]} = \textbf{ReLU}(\mathbf{z}^{[1]})$$ Element-wise

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}$$

Backward Differentiation on a 2-layer MLP

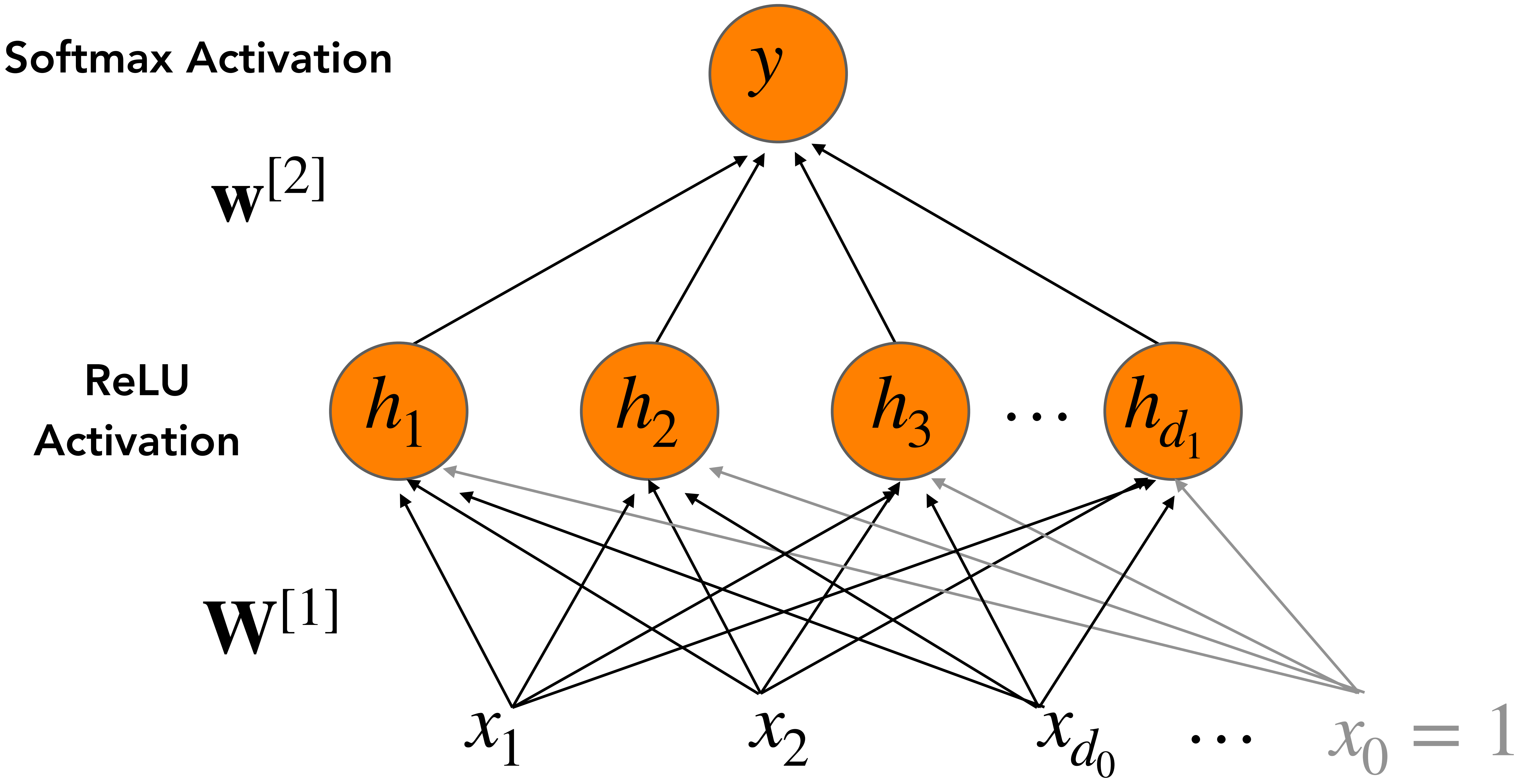


$$z^{[2]} = \mathbf{w}^{[2]} \cdot \mathbf{h}^{[1]}$$

$$\mathbf{h}^{[1]} = \text{ReLU}(\mathbf{z}^{[1]})$$ **Element-wise**

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}$$

Backward Differentiation on a 2-layer MLP



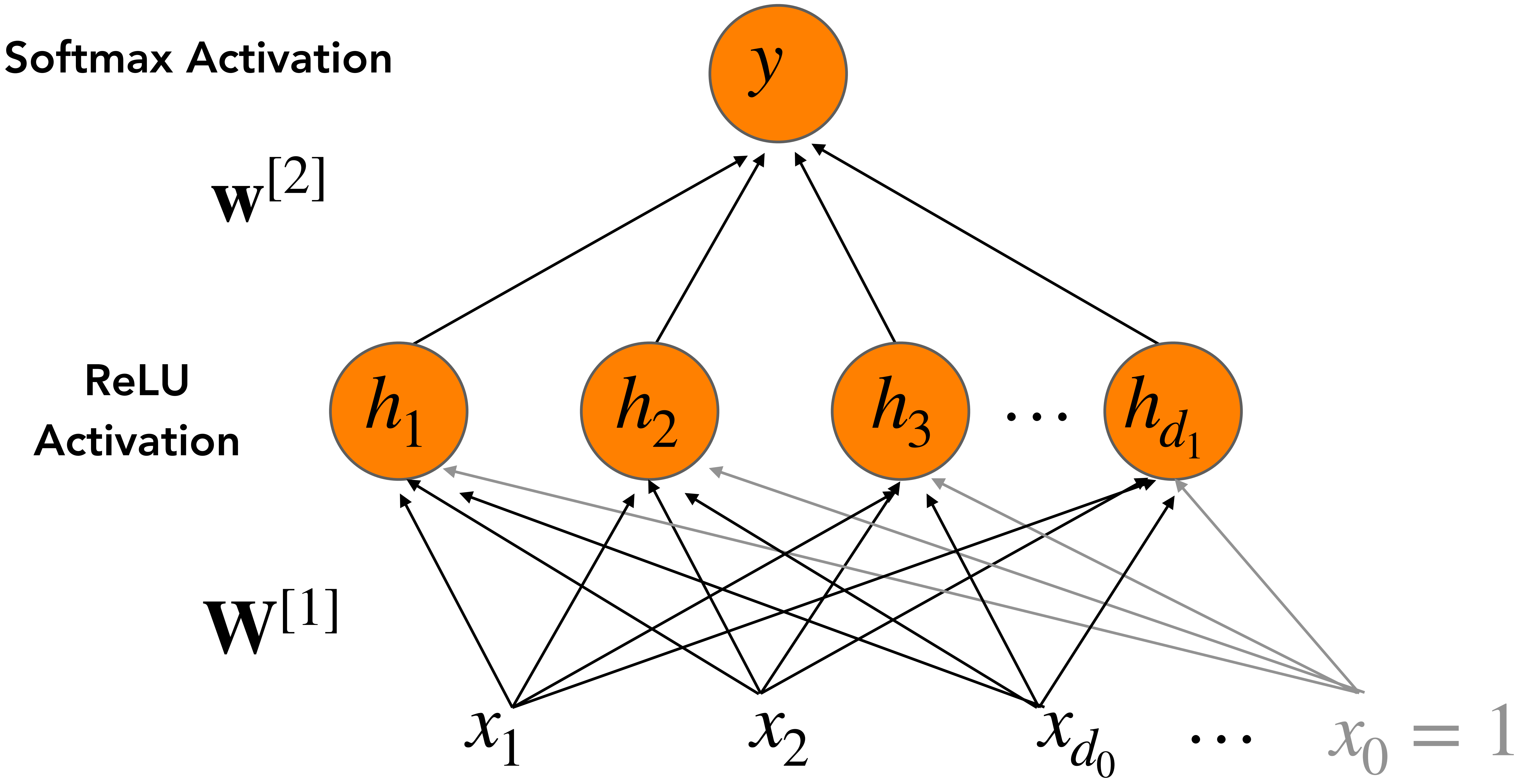
$$\hat{y} = \sigma(z^{[2]})$$

$$z^{[2]} = \mathbf{w}^{[2]} \cdot \mathbf{h}^{[1]}$$

$$\mathbf{h}^{[1]} = \textbf{ReLU}(\mathbf{z}^{[1]})$$ **Element-wise**

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}$$

Backward Differentiation on a 2-layer MLP



$$\hat{y} = \sigma(z^{[2]})$$

$$z^{[2]} = \mathbf{w}^{[2]} \cdot \mathbf{h}^{[1]}$$

$$\mathbf{h}^{[1]} = \mathbf{ReLU}(\mathbf{z}^{[1]})$$ **Element-wise**

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}$$

$$\boxed{\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}}$$

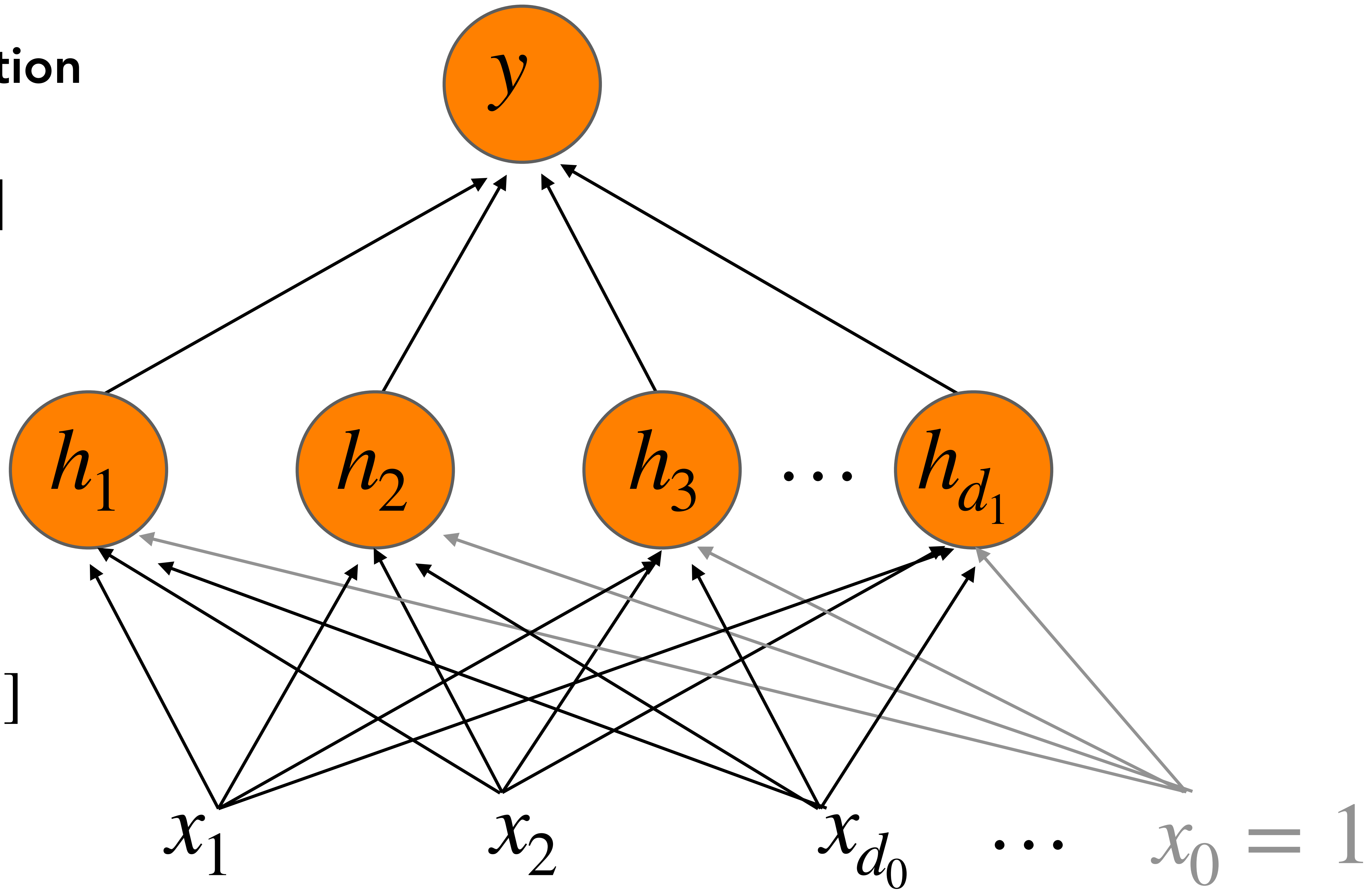
Backward Differentiation on a 2-layer MLP

Softmax Activation

$\mathbf{w}^{[2]}$

ReLU Activation

$\mathbf{W}^{[1]}$



$$\hat{y} = \sigma(z^{[2]})$$

$$z^{[2]} = \mathbf{w}^{[2]} \cdot \mathbf{h}^{[1]}$$

$$\mathbf{h}^{[1]} = \mathbf{ReLU}(\mathbf{z}^{[1]})$$ **Element-wise**

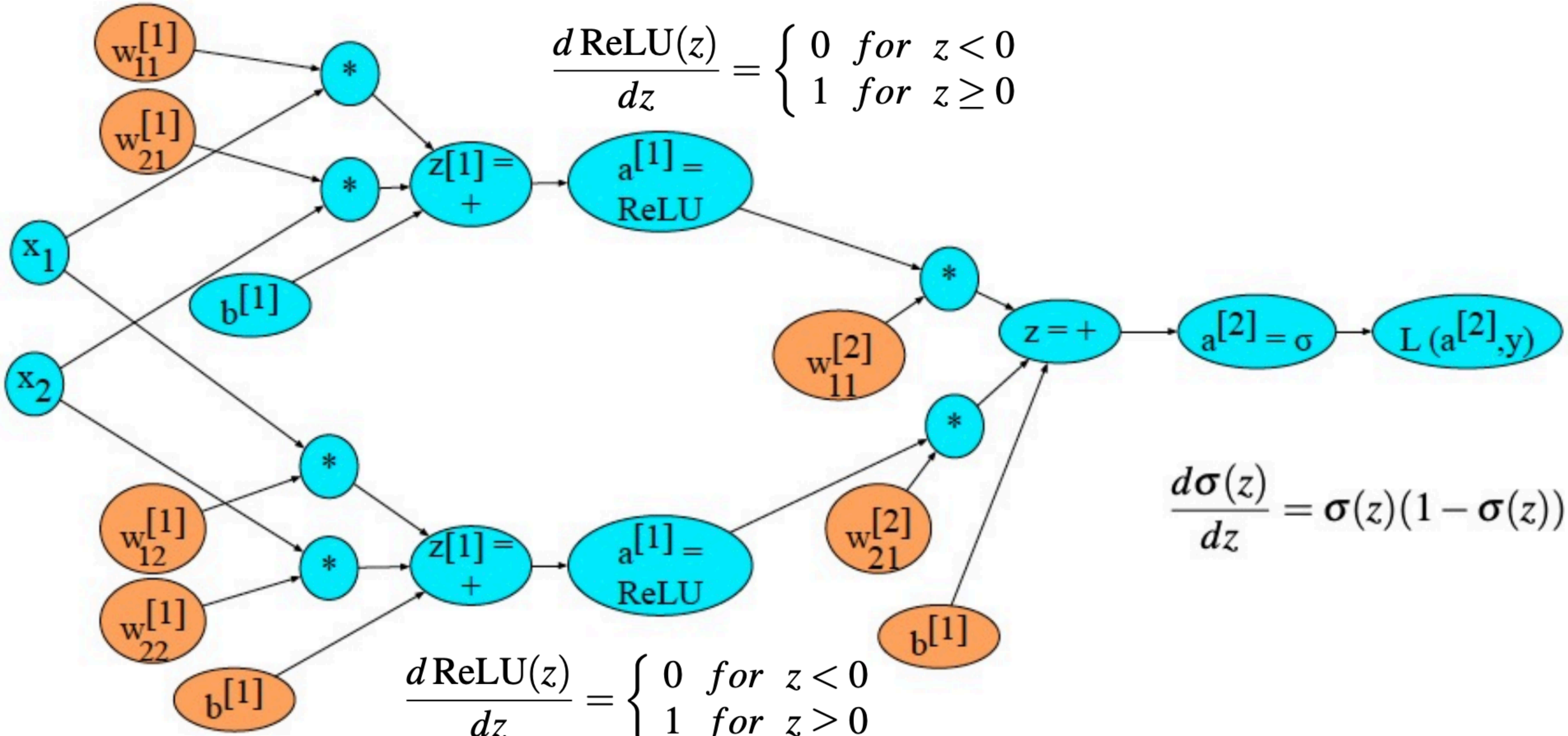
$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x}$$

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)\sigma(-z) = \sigma(z)(1 - \sigma(z))$$

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}$$

USC Viterbi

2 layer MLP with 2 input features



$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}$$

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1 - \sigma(z))$$

$$\frac{d\,\mathrm{ReLU}(z)}{dz} = \begin{cases} 0 & for \quad z < 0 \\ 1 & for \quad z \geq 0 \end{cases}$$

Starting off the backward pass: $\frac{\partial L}{\partial z}$

(I'll write a for $a^{[2]}$ and z for $z^{[2]}$)

$$
\begin{aligned}
z^{[1]} &= W^{[1]}\mathbf{x} + b^{[1]} \\
a^{[1]} &= \mathrm{ReLU}(z^{[1]}) \\
z^{[2]} &= W^{[2]}a^{[1]} + b^{[2]} \\
a^{[2]} &= \sigma(z^{[2]}) \\
\hat{y} &= a^{[2]}
\end{aligned}
$$

$$L(\hat{y}, y) = -(\mathrm{y}\log(\hat{y}) + (1-y)\log(1-\hat{y}))$$

$$L(a, y) = -(\mathrm{y}\log a + (1-y)\log(1-a))$$

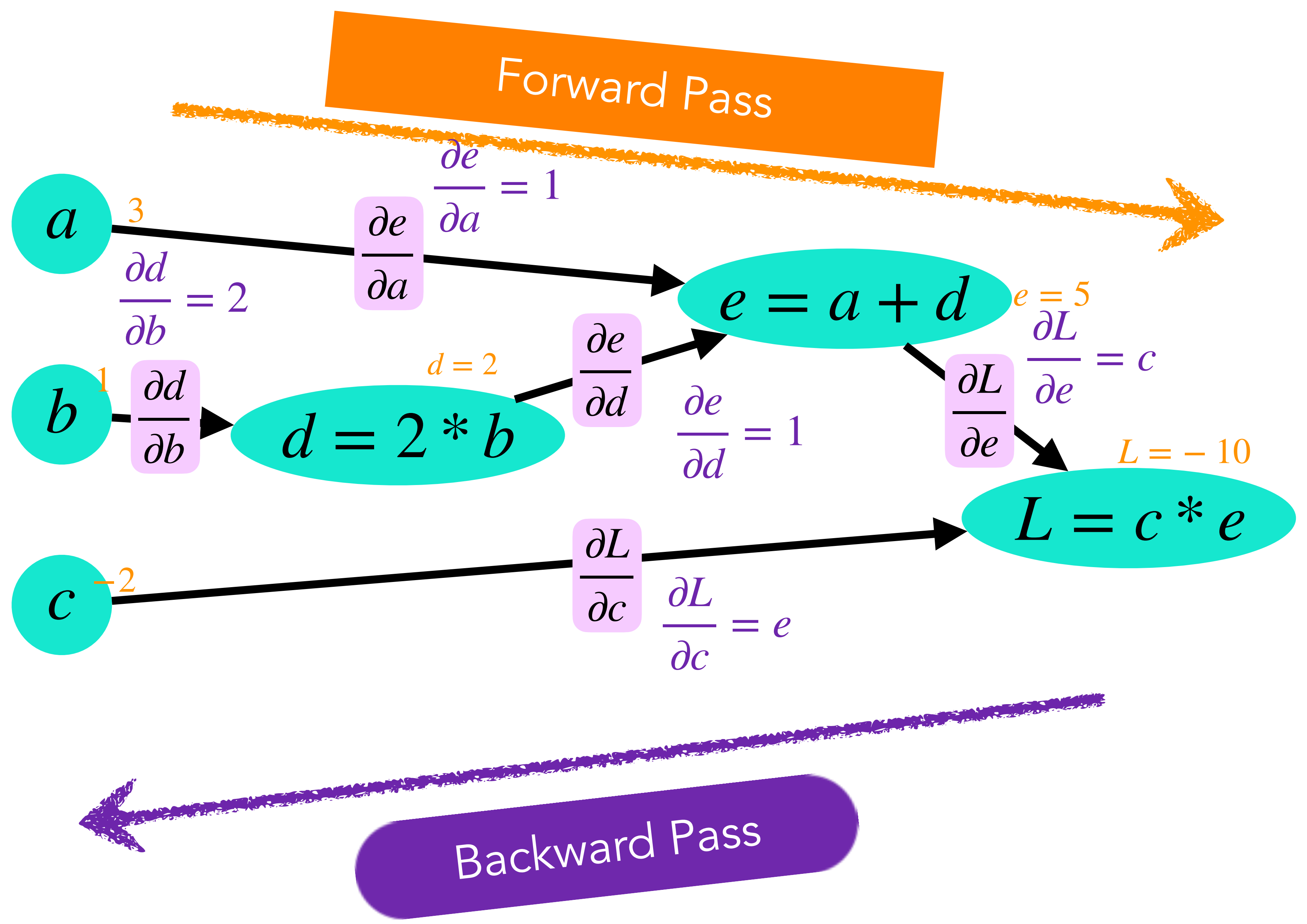
$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}$$

$$\frac{\partial L}{\partial a} = -\left(\left(y\frac{\partial \log(a)}{\partial a}\right) + (1-\mathrm{y})\,\frac{\partial \log(1-a)}{\partial a}\right)$$

$$= -\left(\left(y\frac{1}{a}\right) + (1-\mathrm{y})\,\frac{1}{1-a}(-1)\right) = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)$$

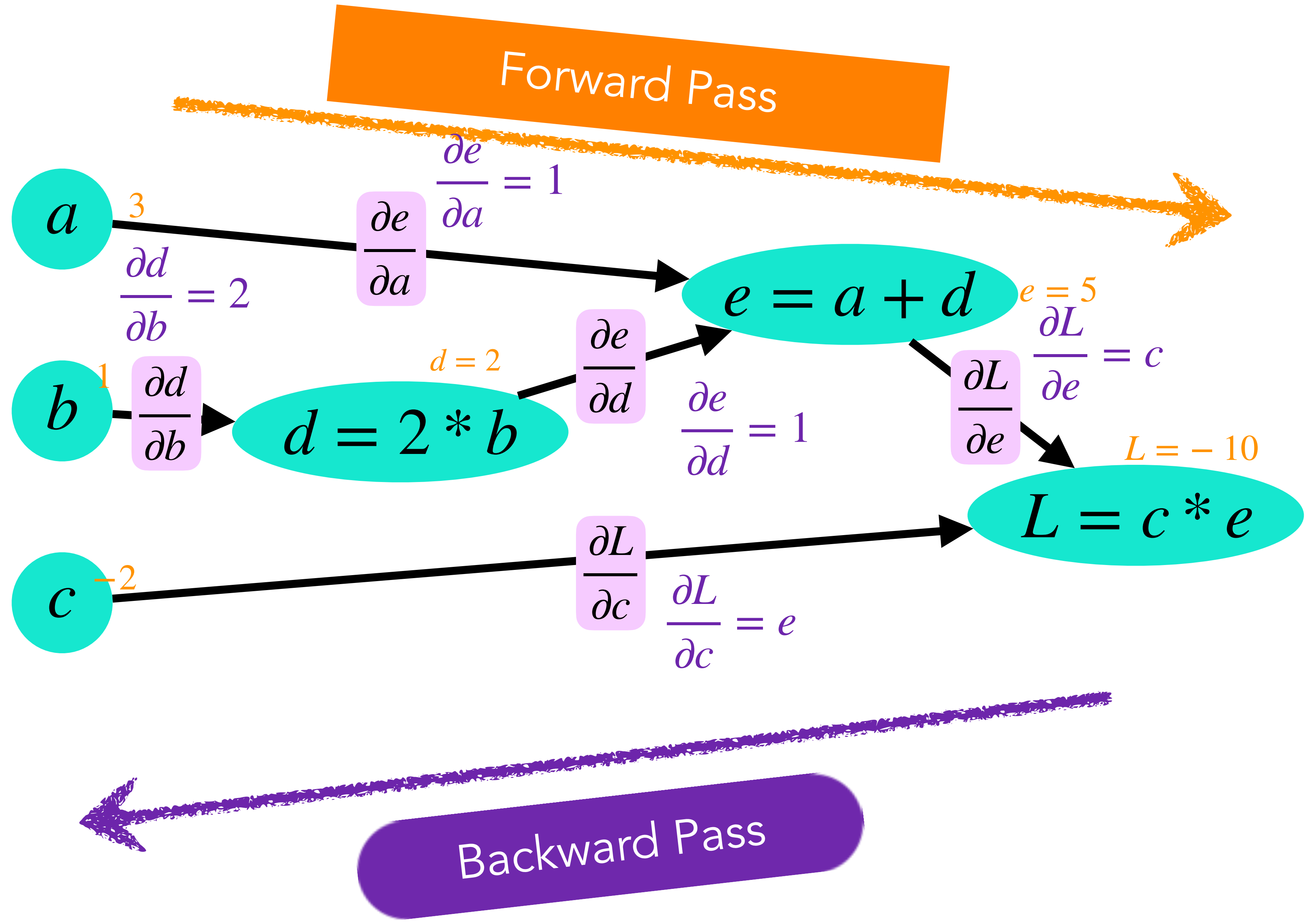
$$\frac{\partial a}{\partial z} = a(1-a) \qquad \frac{\partial L}{\partial z} = -\left(\frac{y}{a} + \frac{y-1}{1-a}\right)a(1-a) = \mathrm{a} - \mathrm{y}$$

Summary: Backprop / Backward Differentiation



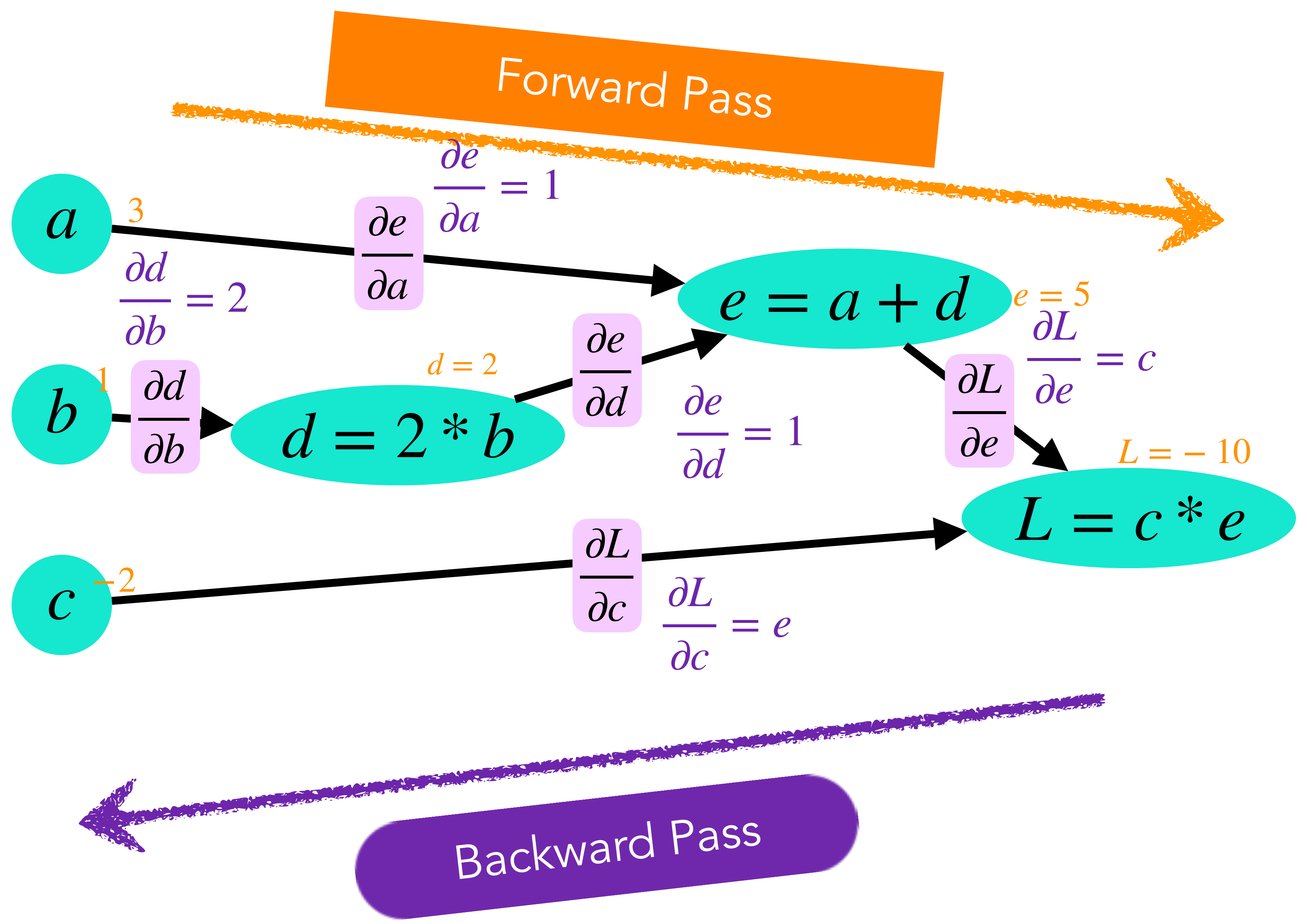
Summary: Backprop / Backward Differentiation

- For training, we need the derivative of the loss with respect to weights in early layers of the network
  - But loss is computed only at the very end of the network!
- Solution: **backward differentiation**



Summary: Backprop / Backward Differentiation

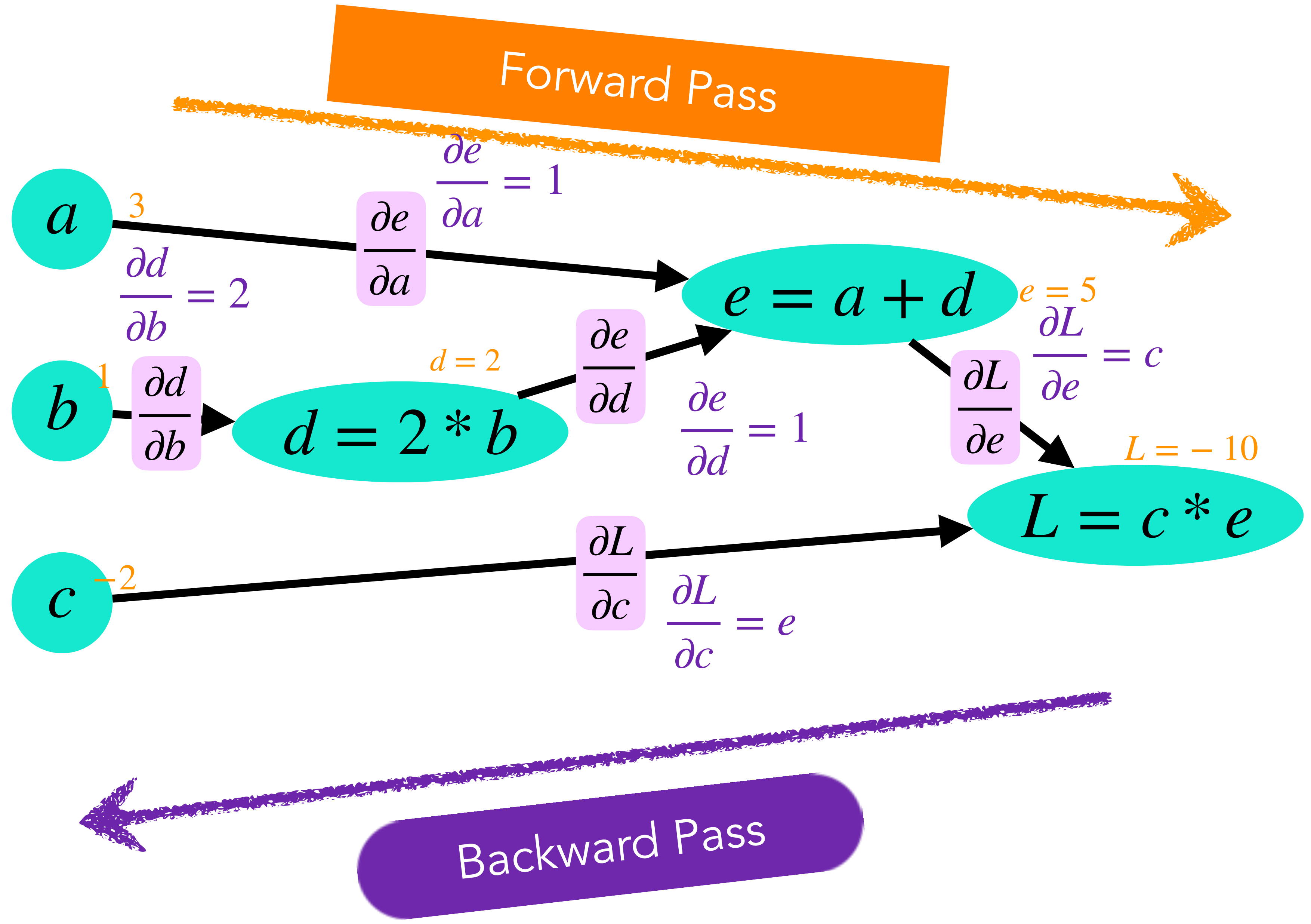
- For training, we need the derivative of the loss with respect to weights in early layers of the network
  - But loss is computed only at the very end of the network!
- Solution: **backward differentiation**



Given a computation graph and the derivatives of all the functions in it we can automatically compute the derivative of the loss with respect to these early weights.

Summary: Backprop / Backward Differentiation

- For training, we need the derivative of the loss with respect to weights in early layers of the network
  - But loss is computed only at the very end of the network!
- Solution: **backward differentiation**



Given a computation graph and the derivatives of all the functions in it we can automatically compute the derivative of the loss with respect to these early weights.

Libraries such as PyTorch do this for you in a single line: `model.backward()`