



# Text Classification with Multi-Layer Perceptrons

2023–24

Ion Androutsopoulos

<http://www.aueb.gr/users/ion/>

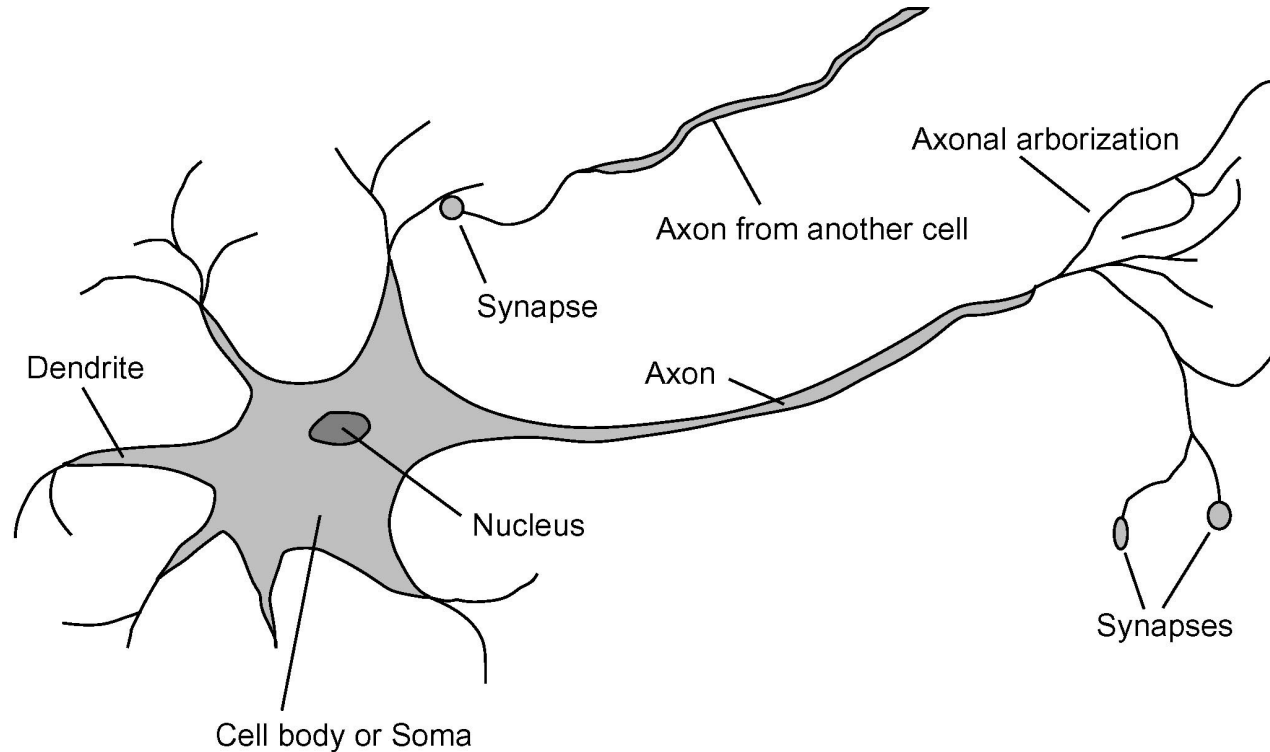
These slides are partly based on material from the books:

- *Artificial Intelligence – A Modern Approach* by S. Russel and P. Norvig, 2<sup>nd</sup> edition, Prentice Hall, 2003,
- *Artificial Intelligence* by I. Vlahavas et al., 3<sup>rd</sup> edition, University of Macedonia Press, 2006 (in Greek).
- *Machine Learning* by T. Mitchell, McGraw-Hill, 1997.

# Contents

- Natural and artificial neural networks (NNs).
- Perceptrons, training them with SGD, limitations.
- Multi-Layer Perceptrons (MLPs) and backpropagation.
- MLPs for text classification, regression, window-based token classification (e.g., for POS tagging, NER).
- Dropout, batch and layer normalization.
- Pre-training word embeddings with Word2Vec.
- Advice for training large neural networks.

# Natural neural networks

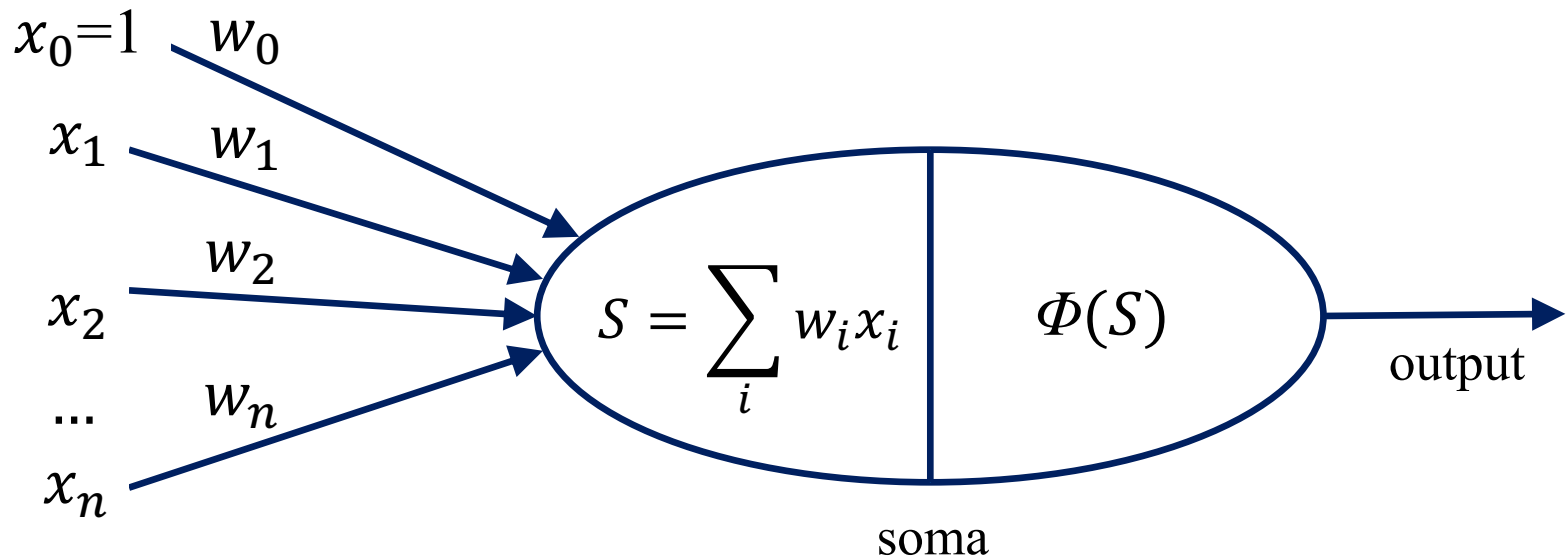


- **Neuron:** cell of the brain.
  - **Cell body or soma:** the main part, includes the **nucleus**.
  - **Dendrites** receive signals from other neurons.
  - **Axon:** transmits a single output to other neurons. Often much longer than the diameter of the soma.
  - **Synapses:** axon-dendrite interfaces, whose conductivities vary.
- **Neural network:** network of many neurons.

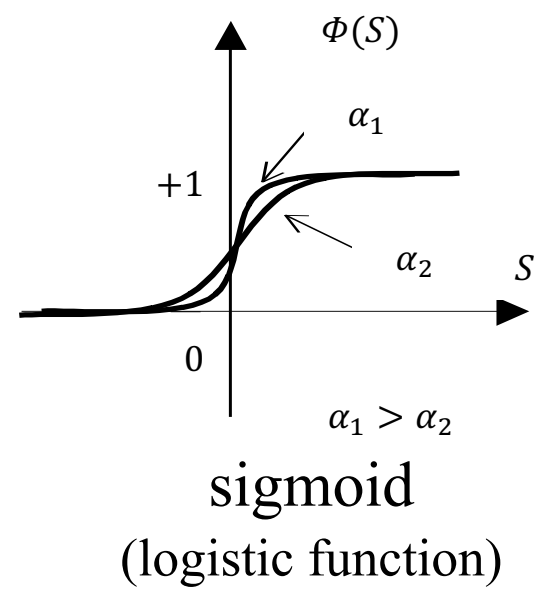
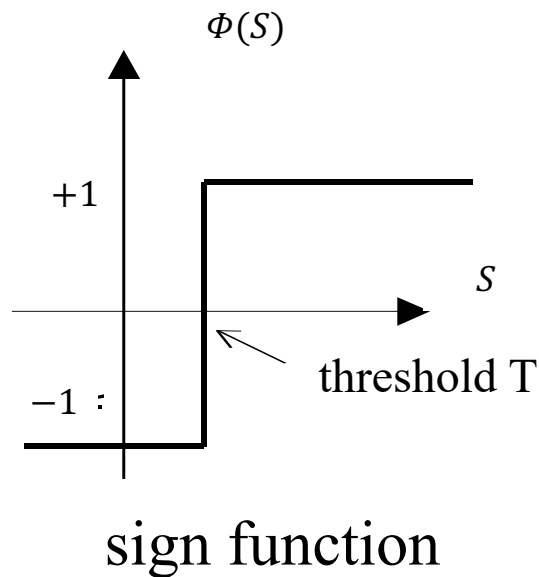
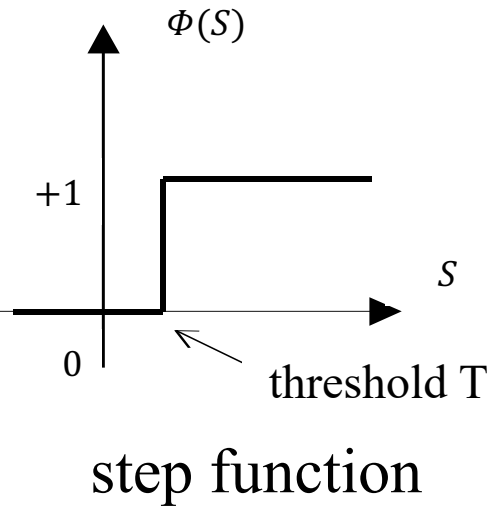
# Artificial neural networks

- **Artificial neuron:**

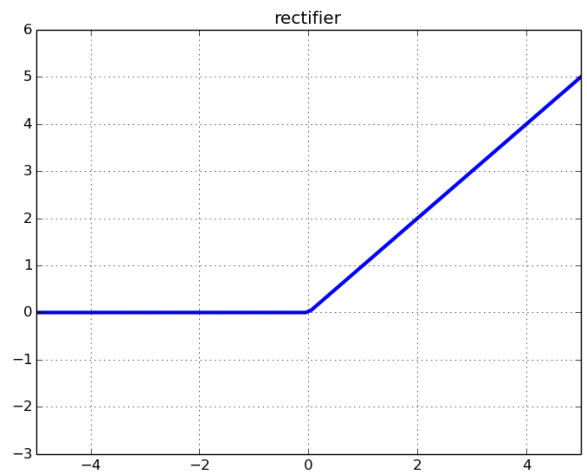
- **Input:** real variables.
- **Input weights:** real variables (roughly synapses).
- **Soma:** computes the **weighted sum** of the inputs, then applies an **activation function** to the sum.



# Activation functions



$$\Phi(S) = \frac{1}{1 + e^{-a \cdot S}}$$

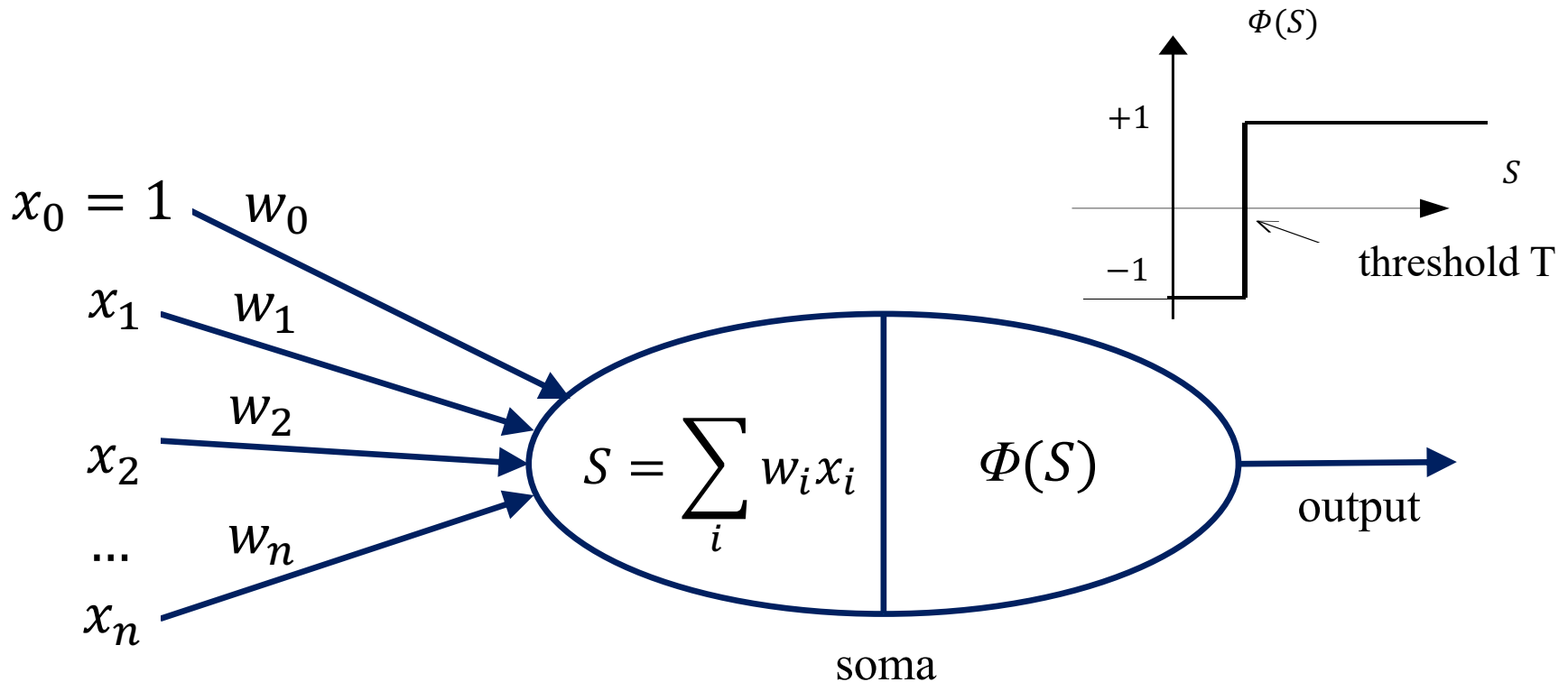


Rectified Linear Unit (ReLU)

The **sigmoid** is **differentiable everywhere**. The hyperbolic tangent (**tanh**) is very similar to the sigmoid, but with values from  $-1$  to  $+1$ ; usually better, unless we really want values in  $(0, 1)$ .

# Perceptron

- A **single neuron**, originally with **sign activation function**.
  - Equivalently, **step** activation function.
  - The Perceptron can be **generalized** (done later), to use a **sigmoid or other** activation function.
  - We can use **several Perceptrons** (e.g., to recognize a letter each).



We usually think of it as an algorithm, but it was a **machine** originally!

# The perceptron is a machine

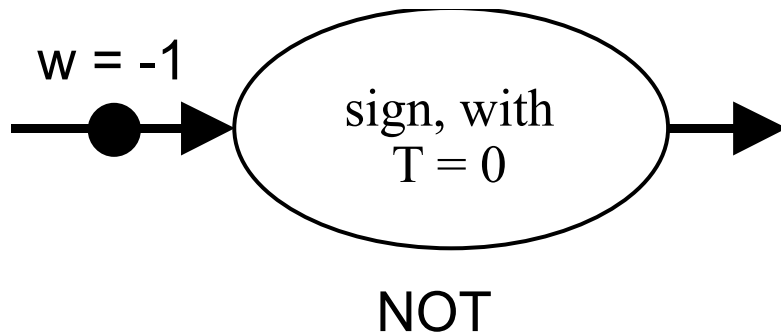
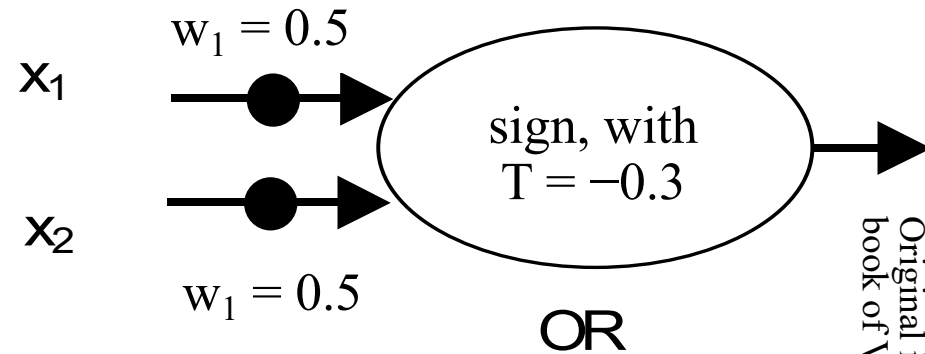
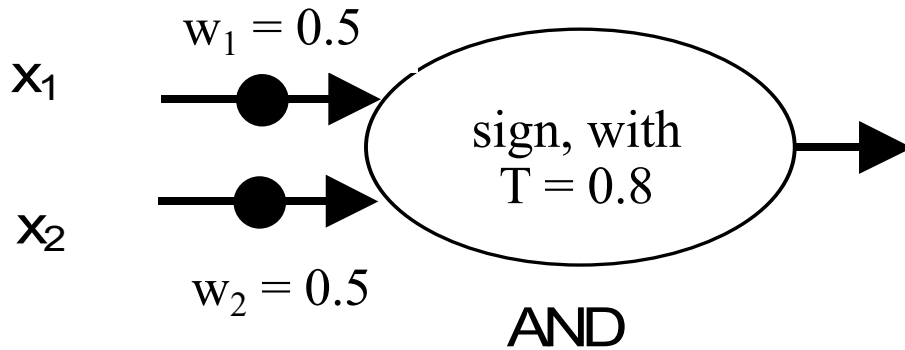


Slide from the presentation “Multilayer Neural Networks” of L. Bottou at the Deep Learning Summer School 2015 ([http://videlectures.net/deeplearning2015\\_montreal/](http://videlectures.net/deeplearning2015_montreal/)).



# Perceptrons as logical gates

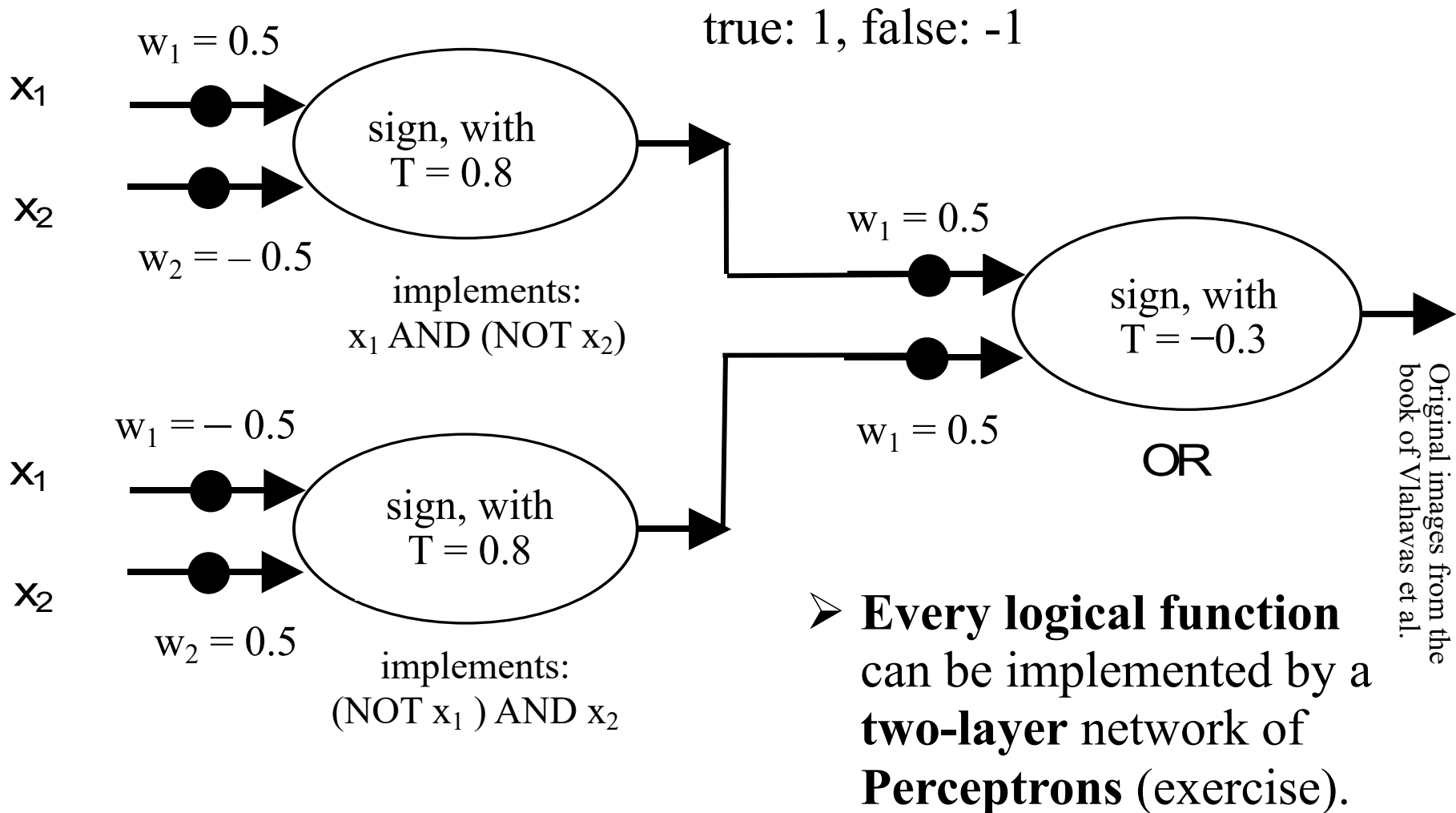
true: 1, false: -1



➤ We **cannot** implement every logical function with a single Perceptron.

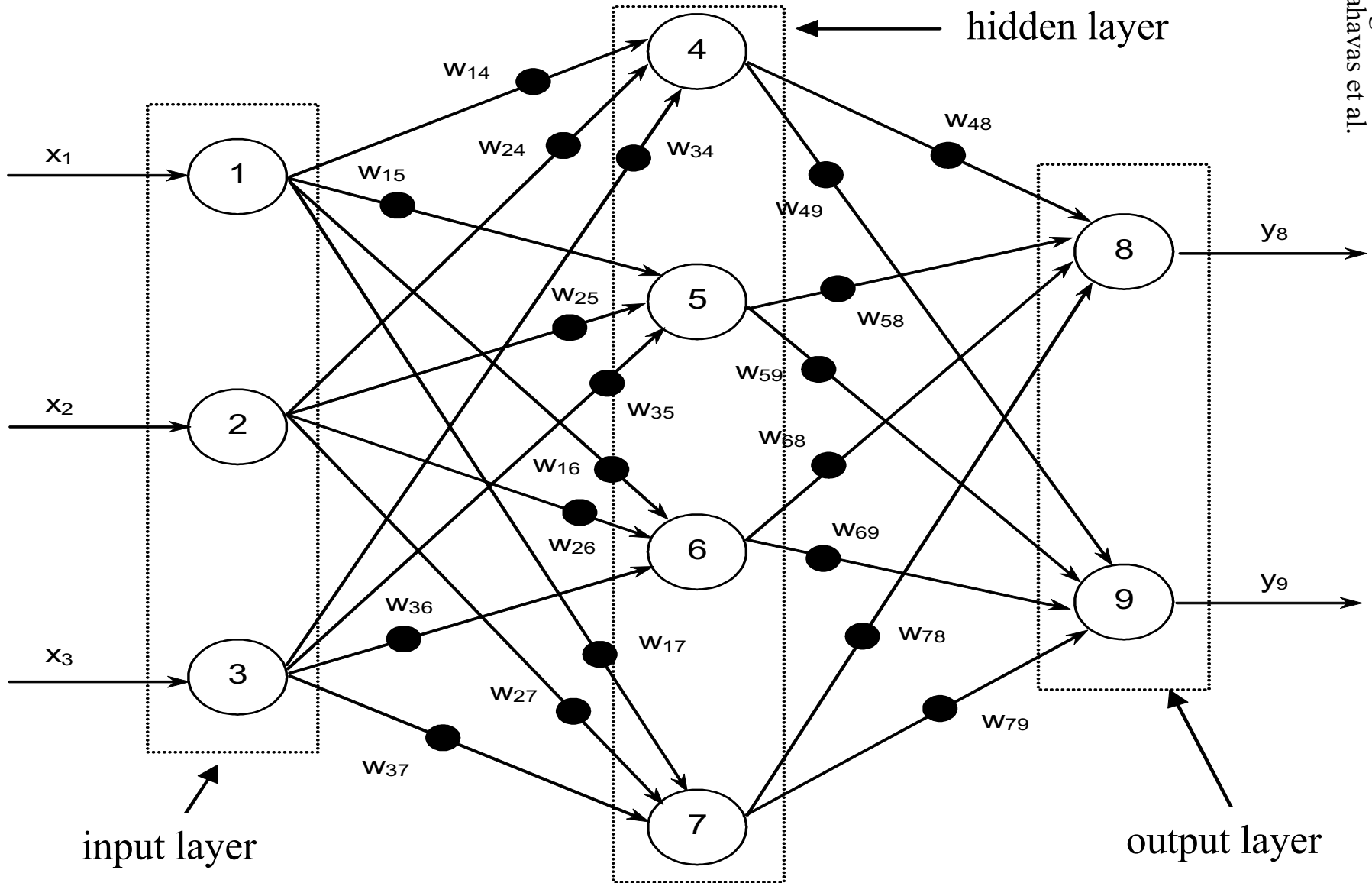
– E.g., we cannot implement an **XOR** gate.

# Two-level XOR implementation

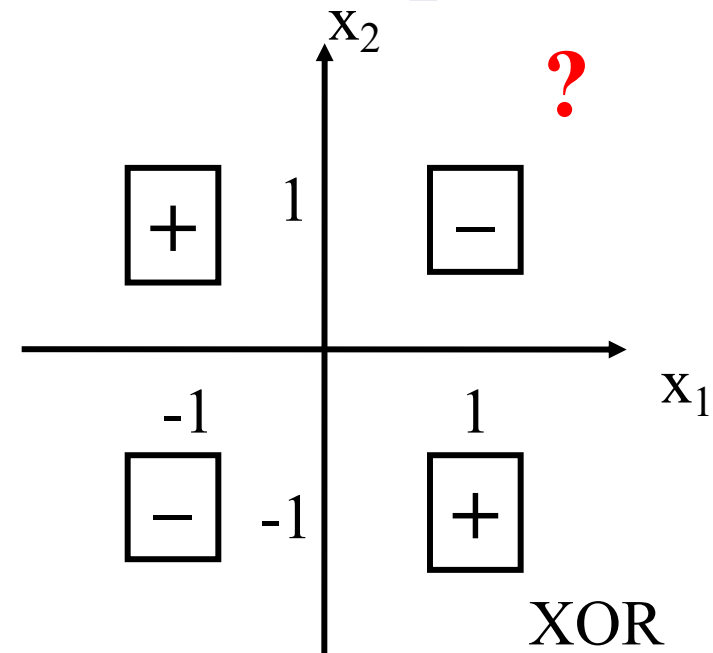
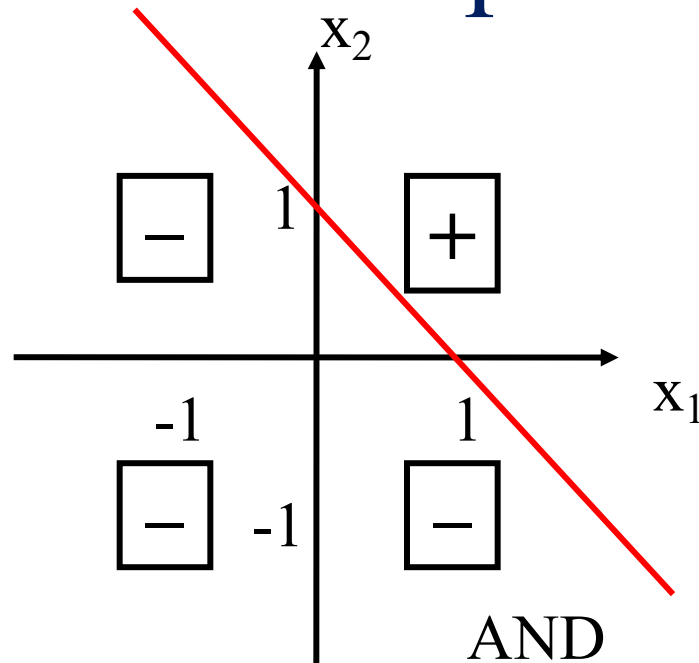


Original images from the  
book of Vlahavas et al.

# Multi-Layer Perceptron (MLP)



# The Perceptron is a linear separator



- Like logistic regression, a single **Perceptron** can learn only **linear separators** (exercise). For **non-linear separators**, we need **Multi-Layer Perceptrons**.
- An **MLP** with **1 hidden layer** can compute (in principle, also learn) **any mapping between discrete spaces** with finite dimensions.
- **For continuous spaces, very roughly speaking**: with **1 hidden layer** we can compute (in principle, also learn) almost any **bounded continuous function**; with **2 hidden layers** we can compute almost any **bounded function**.
- But we may need a very **large, unknown number of neurons** in the **hidden layers**, we may end up **memorizing** the training dataset (exercise), and we **may not** actually manage to **learn (find) the target function**.

# Perceptron's original learning algorithm

1. Start with random weights  $\vec{w}$ .
  2. Set  $i \leftarrow 1$  and  $s \leftarrow 0$ .
  3. Let  $t^{(i)}$  be the correct output for the  $i$ -th training instance and  $o^{(i)}$  the current output for that instance.
  4. Set  $s \leftarrow s + E_i(\vec{w})$ , with:  $E_i(\vec{w}) = 1/2 \cdot [t^{(i)} - o^{(i)}]^2$
  5. Update the weights:  $w_l \leftarrow w_l + \eta \cdot (t^{(i)} - o^{(i)}) \cdot x_l^{(i)}$
  6. If there is a next training instance, set  $i \leftarrow i + 1$  and go to step 3.
  7. If  $s$  had not converged and max number of scans (epochs) of training data not exceeded go to step 2.
- In the simplest case,  $\eta$  is a small positive constant.

# Perceptron with sigmoid or other $\Phi$

- More generally, when using a Perceptron with **activation function  $\Phi$  instead of step/sign**:

$$w_l \leftarrow w_l + \eta \cdot \Phi' \left( \sum_l w_l x_l^{(i)} \right) \cdot (t^{(i)} - \Phi \left( \sum_l w_l x_l^{(i)} \right)) \cdot x_l^{(i)}$$

- For sigmoid  $\Phi(S) = \frac{1}{1 + e^{-S}}$ ,  $\Phi'(S) = \Phi(S) \cdot (1 - \Phi(S))$ .

- And since  $o^{(i)} = \Phi \left( \sum_l w_l x_l^{(i)} \right)$ , the weights update rule becomes:

$$w_l \leftarrow w_l + \eta \cdot o^{(i)} \cdot (1 - o^{(i)}) \cdot (t^{(i)} - o^{(i)}) \cdot x_l^{(i)}$$

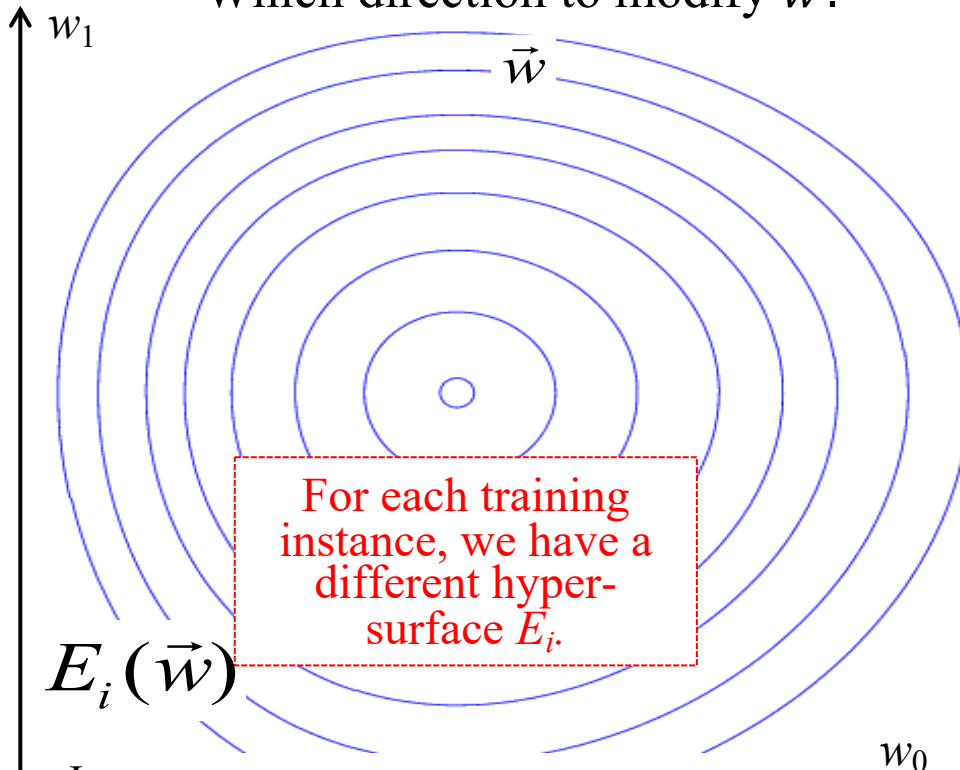
# Derivation of the update rule

- **Squared error loss on the current training instance:**

$$E_i(\vec{w}) = \frac{1}{2} (t^{(i)} - o^{(i)})^2 = \frac{1}{2} (t^{(i)} - \Phi \left( \sum_l w_l x_l^{(i)} \right))^2$$

Random initial weights. Loss on current training instance is  $E_i(\vec{w})$ .

Which direction to modify  $\vec{w}$ ?



For each training instance, we have a different hyper-surface  $E_i$ .

The **gradient**  $\nabla E_i(\vec{w})$  is a vector showing the **direction** we need to **modify**  $\vec{w}$  to obtain the **steepest increase** of  $E_i(\vec{w})$ .

At each iteration, take a **step** to the direction  $-\nabla E_i(\vec{w})$ :

$$\vec{w} \leftarrow \vec{w} - \eta \cdot \nabla E_i(\vec{w})$$

Simplest case:  $\eta$  is a **small positive constant**. **Stochastic gradient descent** to minimize the **total squared error loss**.

Image source:

[http://en.wikipedia.org/wiki/Gradient\\_descent](http://en.wikipedia.org/wiki/Gradient_descent) →

# Derivation of the update rule

$$E_i(\vec{w}) = \frac{1}{2} (t^{(i)} - o^{(i)})^2 = \frac{1}{2} (t^{(i)} - \Phi \left( \sum_{l=1}^n w_l x_l^{(i)} \right))^2$$

$$\nabla E_i(\vec{w}) = \left\langle \frac{\partial E_i(\vec{w})}{\partial w_0}, \dots, \frac{\partial E_i(\vec{w})}{\partial w_l}, \dots, \frac{\partial E_i(\vec{w})}{\partial w_n} \right\rangle$$

$$\frac{\partial E_i}{\partial w_l} = (t^{(i)} - \Phi \left( \sum_l w_l x_l^{(i)} \right)) \cdot \frac{\partial (t^{(i)} - \Phi \left( \sum_l w_l x_l^{(i)} \right))}{\partial w_l} = -(t^{(i)} - o^{(i)}) \cdot \Phi' \left( \sum_l w_l x_l^{(i)} \right) \cdot x_l^{(i)}$$

$$\begin{aligned} \text{Hence: } \nabla E_i(\vec{w}) &= -(t^{(i)} - o^{(i)}) \cdot \Phi' \left( \sum_l w_l x_l^{(i)} \right) \cdot \langle x_1^{(i)}, \dots, x_n^{(i)} \rangle \\ &= -(t^{(i)} - o^{(i)}) \cdot \Phi'(\vec{w} \cdot \vec{x}^{(i)}) \cdot \vec{x}^{(i)} \end{aligned}$$

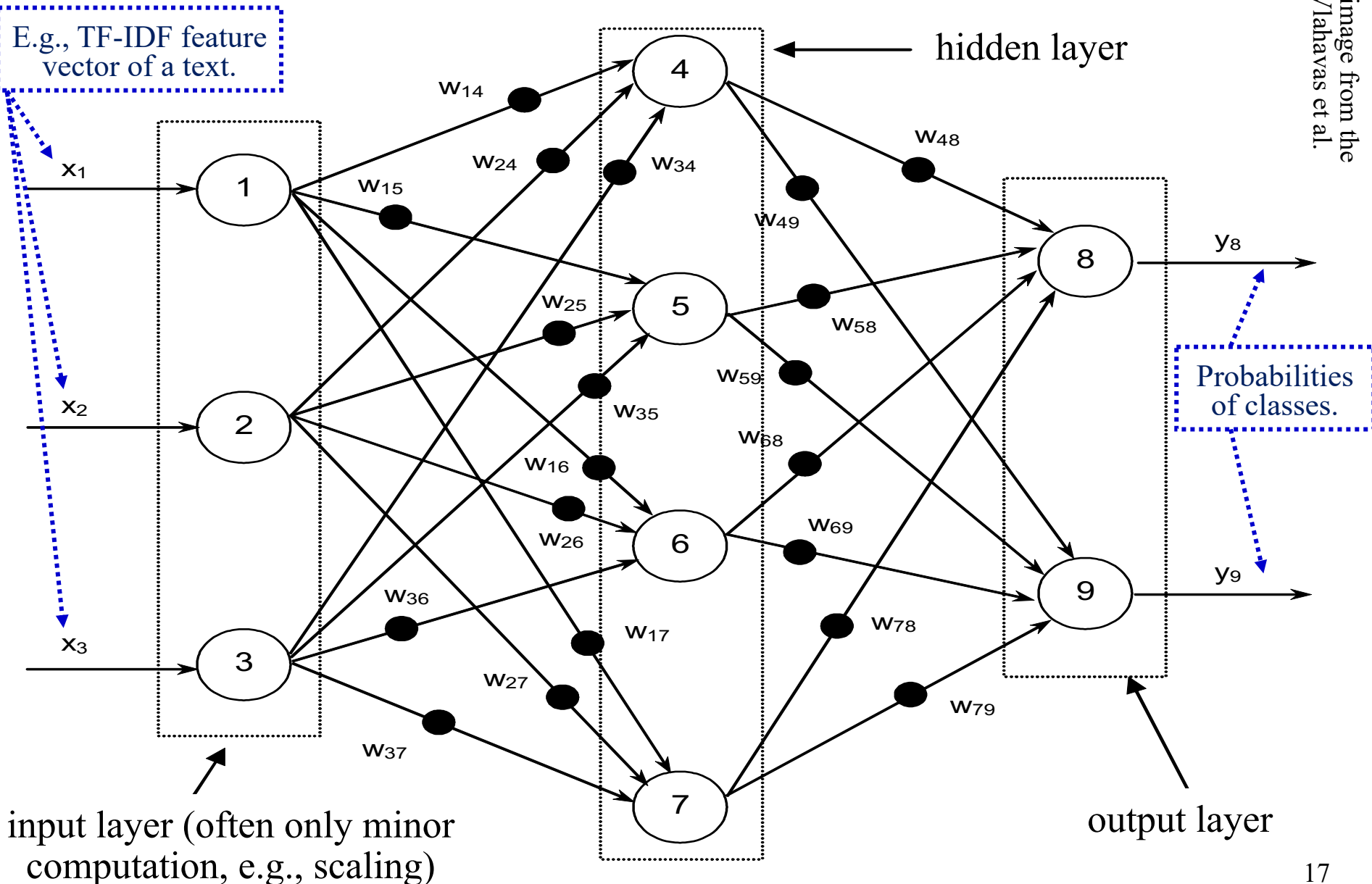
Weights update rule:

$$\vec{w} \leftarrow \vec{w} - \eta \cdot \nabla E_i(\vec{w}) = \vec{w} + \eta \cdot (t^{(i)} - o^{(i)}) \cdot \Phi'(\vec{w} \cdot \vec{x}^{(i)}) \cdot \vec{x}^{(i)}$$

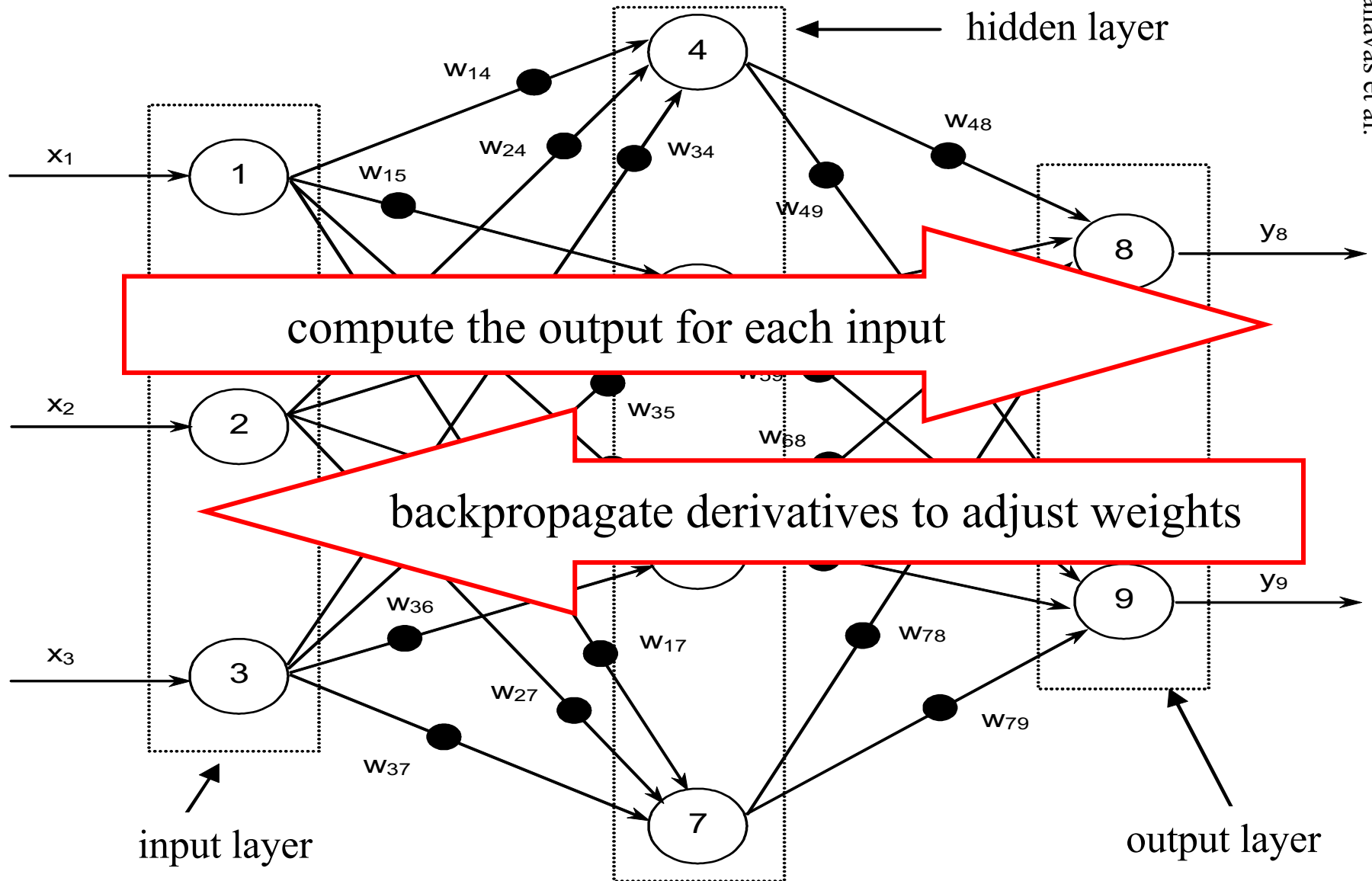
$$\text{For each weight: } w_l \leftarrow w_l + \eta \cdot (t^{(i)} - o^{(i)}) \cdot \Phi'(\vec{w} \cdot \vec{x}^{(i)}) \cdot x_l^{(i)}$$



# Multi-layer Perceptron (MLP)



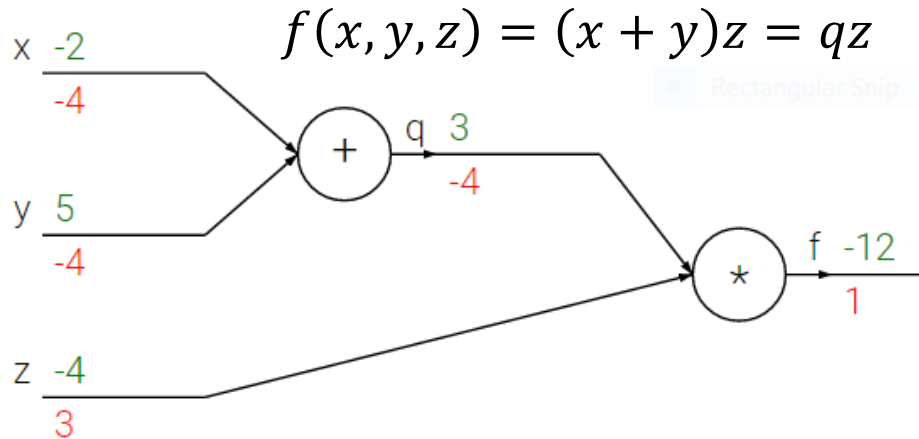
# Backpropagation



# Backpropagation

- **Initialize** all the **weights** to **small random values**.
  - E.g., sample from a **zero-centered Gaussian** with **small  $\sigma$** .
  - Better initializations exist (see DL course).
  - **Normalize** the **features** too (see “Important tricks” of Part 2).
- **In each epoch, for each training example** (or mini-batch):
  - Compute the **output**  $\langle o_1, o_2, \dots \rangle$  for the training example.
  - **For each weight**  $w_{ij}$ , **compute**  $\frac{\partial E}{\partial w_{ij}}$ , where  $E$  the loss on the training example. We **compute derivatives right to left**.
  - **Update each weight** as:  $w_{ij} \leftarrow w_{ij} - \eta \cdot \frac{\partial E}{\partial w_{ij}}$ , i.e., for all the weights together:  $W \leftarrow W - \eta \cdot \nabla_W E$
  - Hence, we use **SGD** (or variants). **No guarantee** SGD will find the **best solution**, but it (often) works in practice!

# Example of computation graph



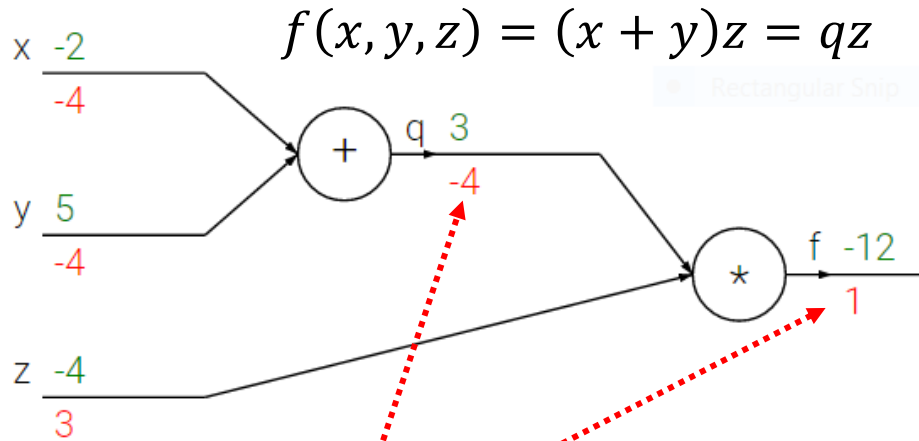
Example and figure from Stanford's  
 "CNNs for Visual Recognition" (2016,  
 F.-F. Li, A. Karpathy, J. Johnson)  
<http://cs231n.github.io/optimization-2/>

- **Forward pass:**  $\langle x, y, z \rangle = \langle -2, 5, -4 \rangle$ ,  $q = 3$ ,  $f = -12$
- Imagine we wish to **minimize  $f$**  using **SGD**.
  - In a more realistic scenario,  $f$  would be a **loss function**, and  $\langle x, y, z \rangle$  the **weights vector**.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \leftarrow \begin{bmatrix} x \\ y \\ z \end{bmatrix} - \eta \nabla f(x, y, z) = \begin{bmatrix} x \\ y \\ z \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \\ \frac{\partial f}{\partial z} \end{bmatrix}$$

We need:  
 $\frac{\partial f}{\partial x}$   $\frac{\partial f}{\partial y}$   $\frac{\partial f}{\partial z}$

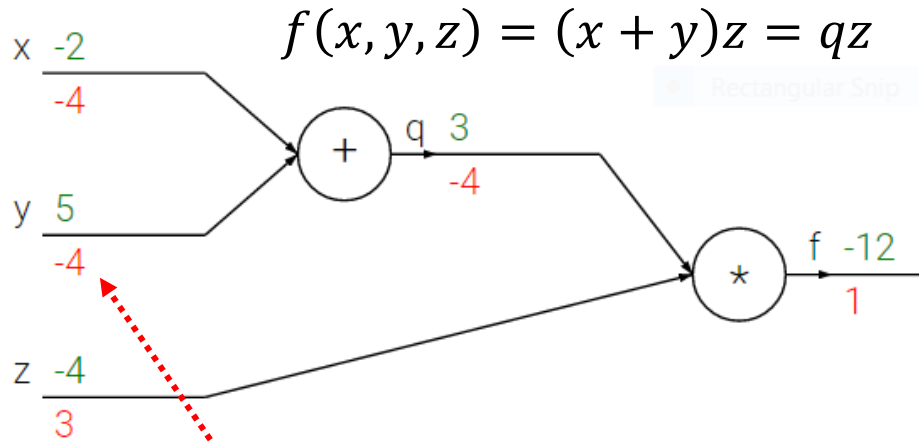
# Backpropagation in the graph



Example and figure from Stanford's  
"CNNs for Visual Recognition" (2016,  
F.-F. Li, A. Karpathy, J. Johnson)  
<http://cs231n.github.io/optimization-2/>

- **Backpropagation:** We compute derivatives right to left.
  - $\frac{\partial f}{\partial f} = 1$  by definition.
  - $\frac{\partial f}{\partial q} = z$ . And for this  $\langle x, y, z \rangle$  input,  $z = -4$ .
  - During the **forward pass**, we need to **save the outputs of all the nodes** (e.g., here we need the **value of z**).

# Backpropagation in the graph



Example and figure from Stanford's "CNNs for Visual Recognition" (2016, F.-F. Li, A. Karpathy, J. Johnson) <http://cs231n.github.io/optimization-2/>

• **Backpropagation:** We compute derivatives right to left.

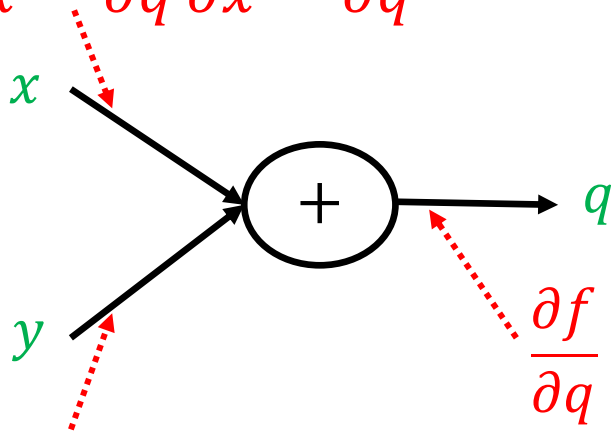
- $\frac{\partial f}{\partial f} = 1$  by definition.
- $\frac{\partial f}{\partial q} = z$ . And for this  $\langle x, y, z \rangle$  input,  $z = -4$ .
- $\frac{\partial f}{\partial z} = q$ . And for this  $\langle x, y, z \rangle$  input,  $q = 3$ .
- $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = \frac{\partial f}{\partial q} \cdot 1$ . And here  $\frac{\partial f}{\partial q}$  is  $-4$ .
- $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = \frac{\partial f}{\partial q} \cdot 1$ . And here  $\frac{\partial f}{\partial q}$  is  $-4$ .

incoming gradient

local gradient

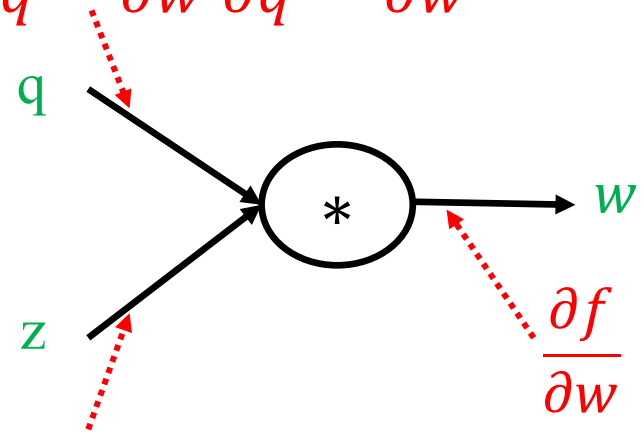
# Plug-and-play gates

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = \frac{\partial f}{\partial q} \cdot 1$$



$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = \frac{\partial f}{\partial q} \cdot 1$$

$$\frac{\partial f}{\partial q} = \frac{\partial f}{\partial w} \frac{\partial w}{\partial q} = \frac{\partial f}{\partial w} \cdot z$$



$$\frac{\partial f}{\partial z} = \frac{\partial f}{\partial w} \frac{\partial w}{\partial z} = \frac{\partial f}{\partial w} \cdot q$$

class PlusGate:

forward(x, y):

return x+y

backward( $\frac{\partial f}{\partial q}$ ):

return  $\langle \frac{\partial f}{\partial q}, \frac{\partial f}{\partial q} \rangle$

class StarGate:

forward(q, z):

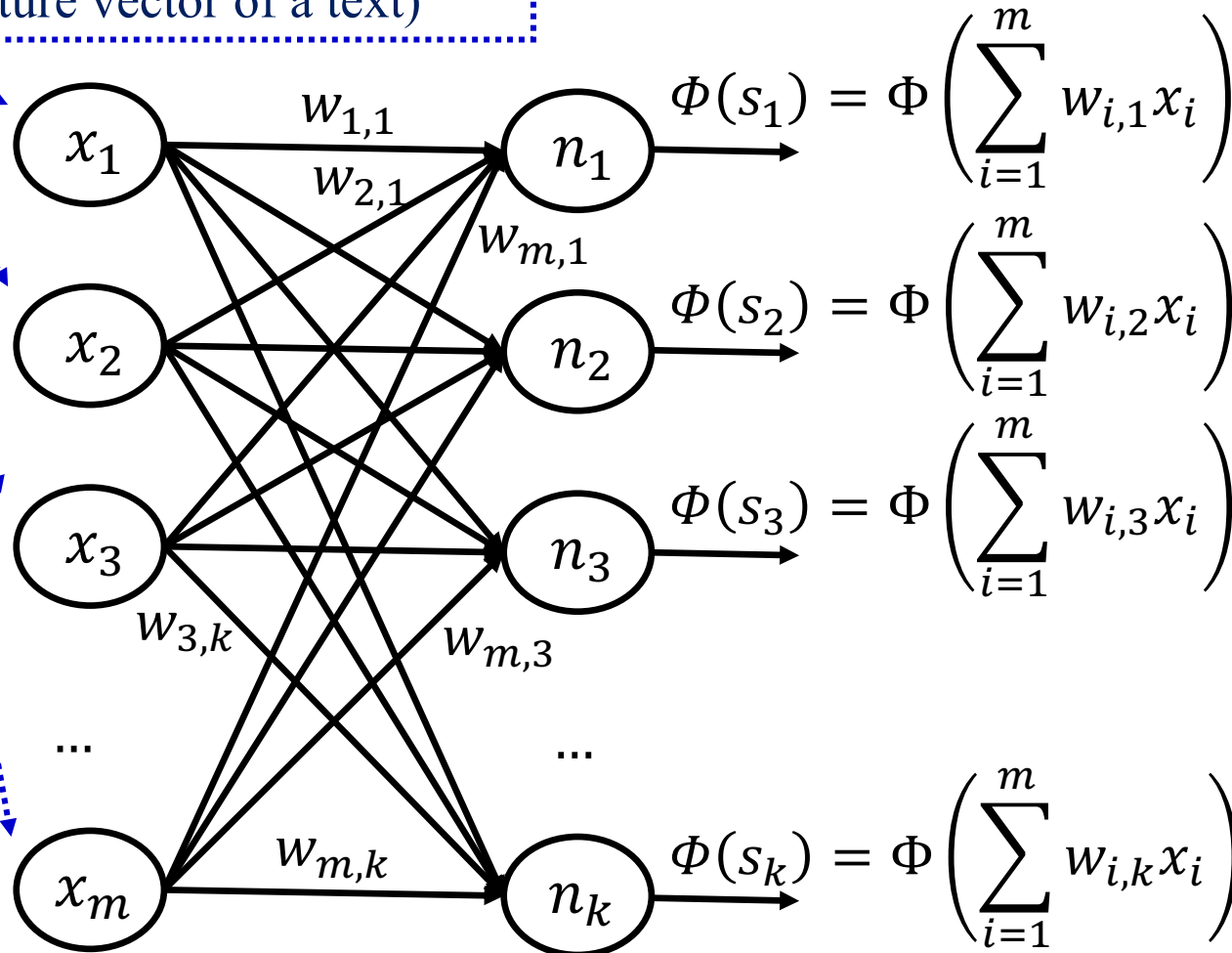
return q \* z

backward( $\frac{\partial f}{\partial w}$ ):

return  $\langle \frac{\partial f}{\partial w} \cdot z, \frac{\partial f}{\partial w} \cdot q \rangle$

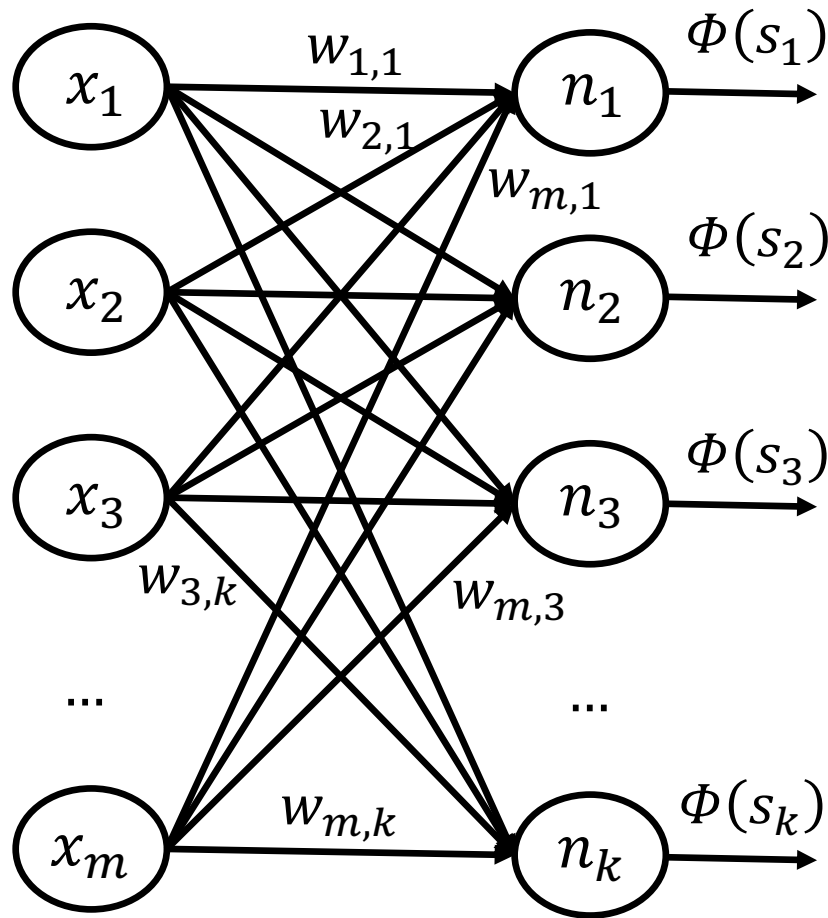
# More compact notation of NNs

Input instance (e.g., TF-IDF feature vector of a text)





# More compact notation of NNs



$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ \dots \\ s_k \end{bmatrix} = \begin{bmatrix} w_{1,1}x_1 + w_{2,1}x_2 + \dots + w_{m,1}x_m \\ w_{1,2}x_1 + w_{2,2}x_2 + \dots + w_{m,2}x_m \\ w_{1,3}x_1 + w_{2,3}x_2 + \dots + w_{m,3}x_m \\ \dots \\ w_{1,k}x_1 + w_{2,k}x_2 + \dots + w_{m,k}x_m \end{bmatrix}$$

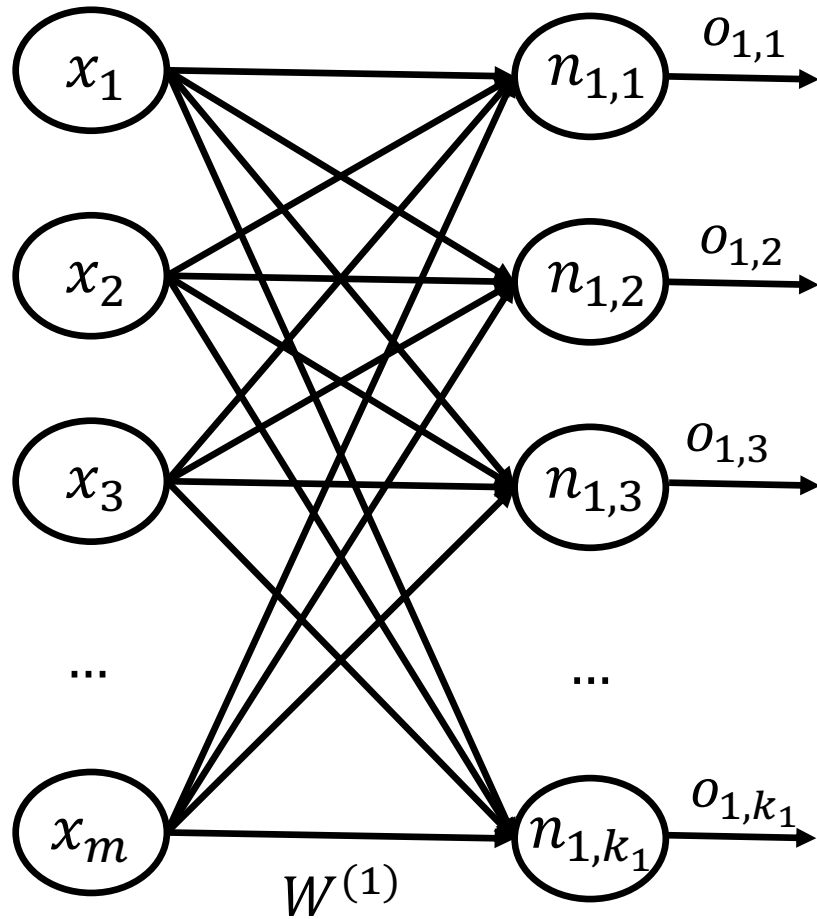
$$\begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ \dots \\ s_k \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{2,1} & \dots & w_{m,1} \\ w_{1,2} & w_{2,2} & \dots & w_{m,2} \\ w_{1,3} & w_{2,3} & \dots & w_{m,3} \\ \dots & \dots & \dots & \dots \\ w_{1,k} & w_{2,k} & \dots & w_{m,k} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \dots \\ x_m \end{bmatrix}$$

$$\vec{s} = W\vec{x}$$

We learn  $W$  with backpropagation.

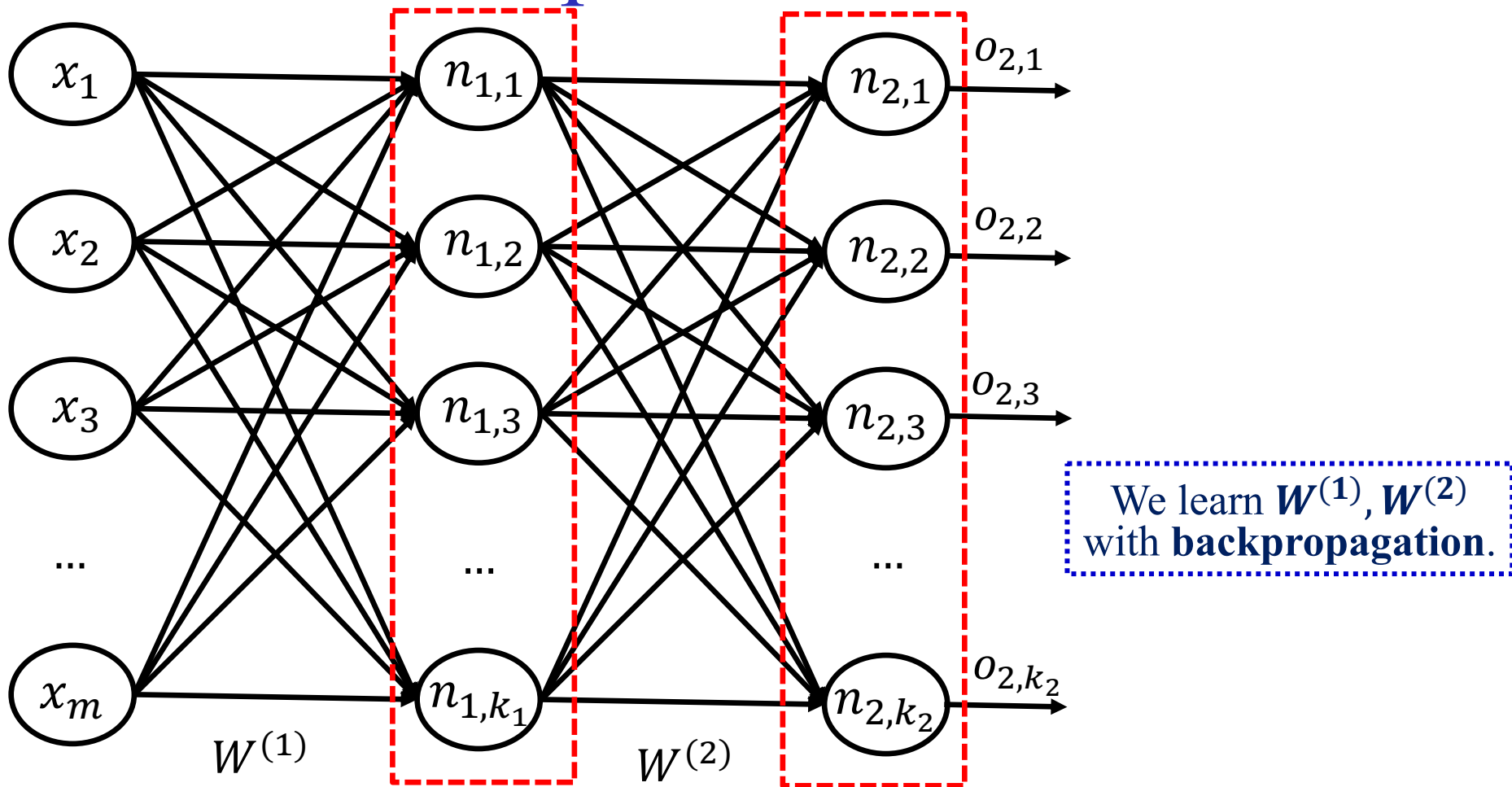
$$\vec{o} = \begin{bmatrix} o_1 \\ o_2 \\ o_3 \\ \dots \\ o_k \end{bmatrix} = \begin{bmatrix} \Phi(s_1) \\ \Phi(s_2) \\ \Phi(s_3) \\ \dots \\ \Phi(s_k) \end{bmatrix} = \Phi(\vec{s}) = \Phi(W\vec{x})$$

# More compact notation of NNs



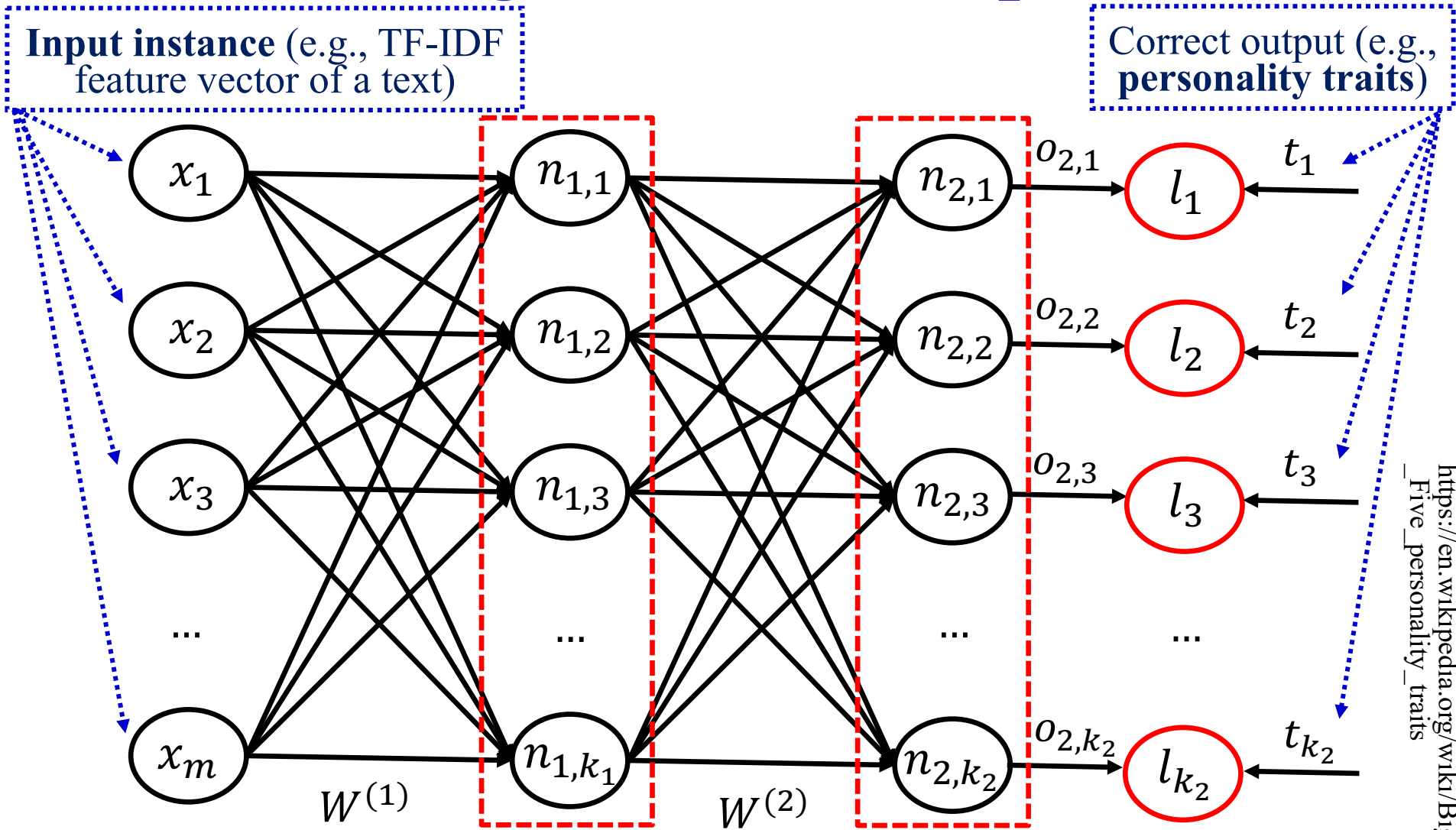
$$\vec{o}^{(1)} = \begin{bmatrix} o_{1,1} \\ o_{1,2} \\ \dots \\ o_{1,k_1} \end{bmatrix} = \Phi(\vec{s}^{(1)}) = \Phi(W^{(1)}\vec{x})$$

# More compact notation of NNs



$$\vec{o}^{(1)} = \begin{bmatrix} o_{1,1} \\ o_{1,2} \\ \dots \\ o_{1,k_1} \end{bmatrix} = \Phi(\vec{s}^{(1)}) = \Phi(W^{(1)}\vec{x}) \quad \vec{o}^{(2)} = \begin{bmatrix} o_{2,1} \\ o_{2,2} \\ \dots \\ o_{2,k_2} \end{bmatrix} = \Phi(\vec{s}^{(2)}) = \Phi(W^{(2)}\vec{o}^{(1)})$$

# Regression example



$$\vec{o}^{(1)} = \tanh(W^{(1)}\vec{x})$$

$$\vec{o}^{(2)} = W^{(2)}\vec{o}^{(1)}$$

**Squared error loss at current training example**

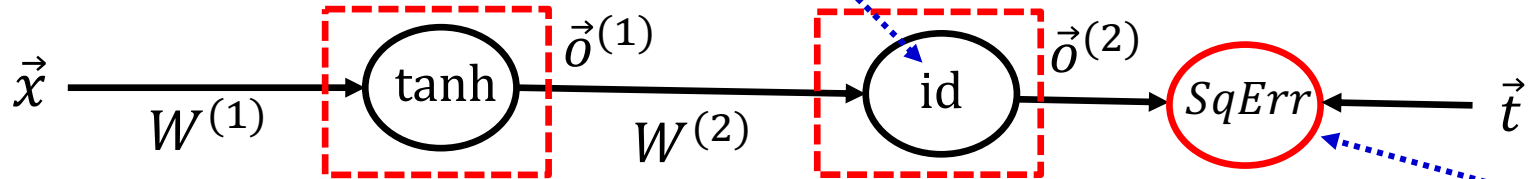
$$l = \frac{1}{2} \sum_{j=1}^{k_2} l_j^2 = \frac{1}{2} \sum_{j=1}^{k_2} (o_{2,j} - t_j)^2$$

# Regression example – more compact

Input instance (e.g., TF-IDF feature vector of a text)

$\Phi(s) = s$   
(identity)

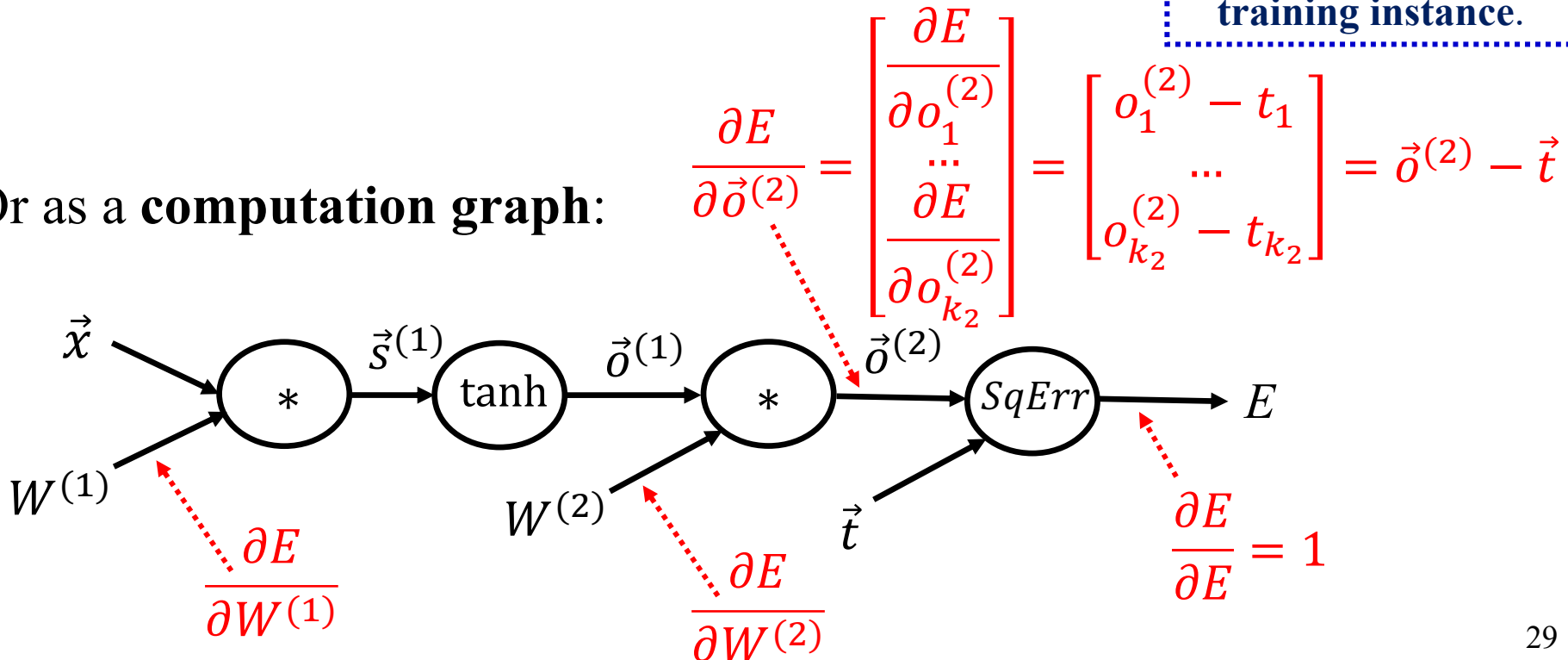
Correct output (e.g., personality traits)



$$\vec{o}^{(1)} = \tanh(W^{(1)}\vec{x}) \quad \vec{o}^{(2)} = W^{(2)}\vec{o}^{(1)}$$

Squared error loss for the current training instance.

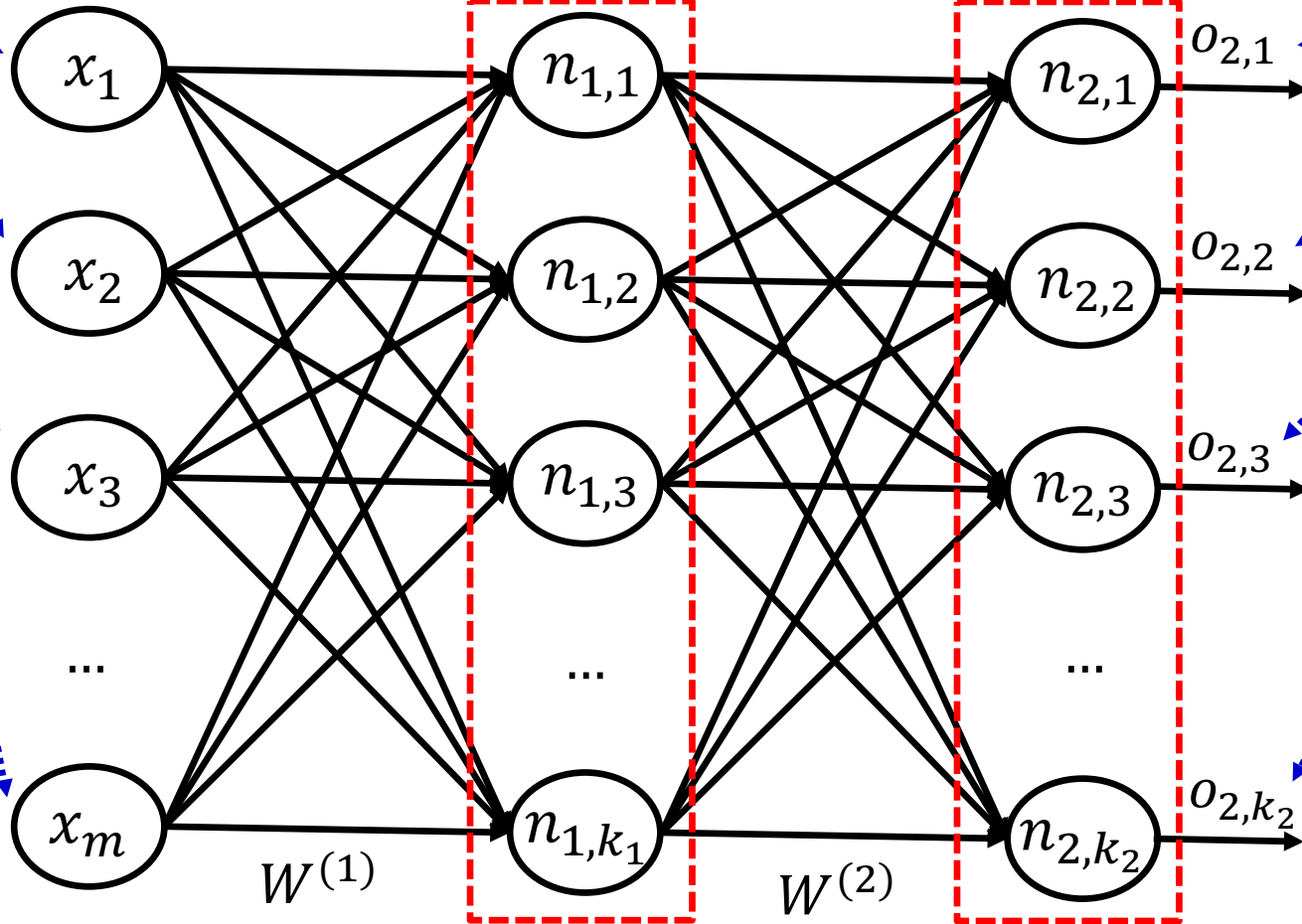
Or as a computation graph:



# Classification example

Input instance (e.g., TF-IDF feature vector of a text)

How probable the system believes it is that the input belongs in each one of  $k_2$  classes.



$$\vec{o}^{(1)} = \tanh(W^{(1)}\vec{x})$$

$$\vec{o}^{(2)} = \text{softmax}(W^{(2)}\vec{o}^{(1)})$$

# Softmax

$$W^{(2)}\vec{o}^{(1)} = \vec{s}^{(2)} = \begin{bmatrix} s_{2,1} \\ s_{2,2} \\ \dots \\ s_{2,k_2} \end{bmatrix}$$

Output of the last layer, without an activation function. **Confidence scores**, one for each class. We want to **convert them to probabilities summing to 1**.

$$\text{softmax}(W^{(2)}\vec{o}^{(1)}) = \text{softmax}(\vec{s}^{(2)}) = \text{softmax}\left(\begin{bmatrix} s_{2,1} \\ s_{2,2} \\ \dots \\ s_{2,k_2} \end{bmatrix}\right)$$

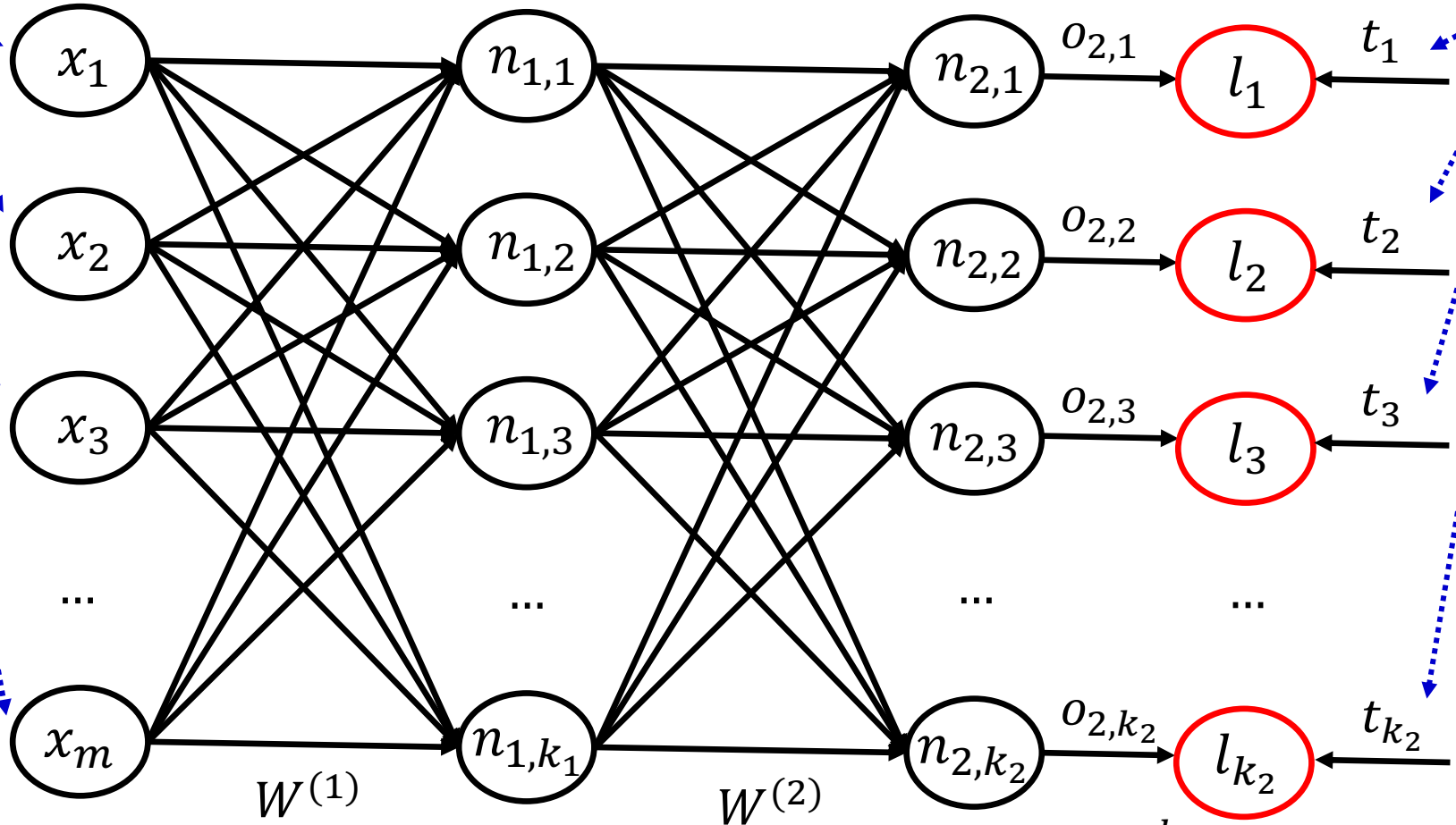
$$= \begin{bmatrix} \frac{\exp(s_{2,1})}{\sum_{j=1}^{k_2} \exp(s_{2,j})} \\ \frac{\exp(s_{2,2})}{\sum_{j=1}^{k_2} \exp(s_{2,j})} \\ \dots \\ \frac{\exp(s_{2,k_2})}{\sum_{j=1}^{k_2} \exp(s_{2,j})} \end{bmatrix}$$

Softmax also **moves the largest of its inputs towards 1** and the other inputs towards 0. Intuitively a **soft argmax!**

# Classification example

Input instance (e.g., TF-IDF feature vector of a text)

Correct output ( $t_j = 1$  means the single correct class is the  $j$ -th one)



$$\vec{o}^{(1)} = \tanh(W^{(1)}\vec{x})$$

$$\vec{o}^{(2)} = \text{softmax}(W^{(2)}\vec{o}^{(1)})$$

Cross entropy loss at the current training instance

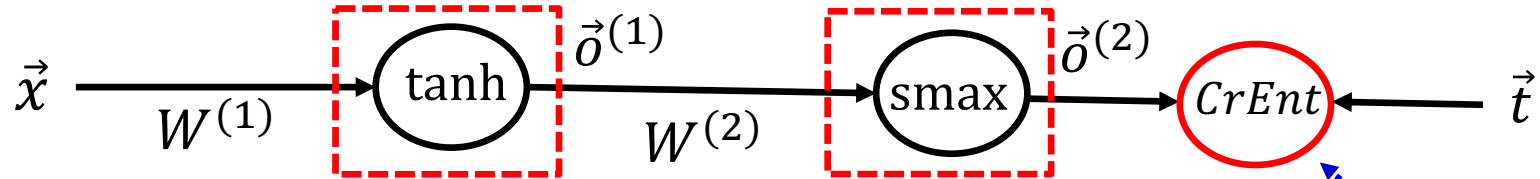
$$l = - \sum_{j=1}^{k_2} t_j \log(o_{2,j})$$



# Classification example – more compact

**Input instance** (e.g., TF-IDF feature vector of a text)

**Correct output** (correct class prediction, 1-hot vector)

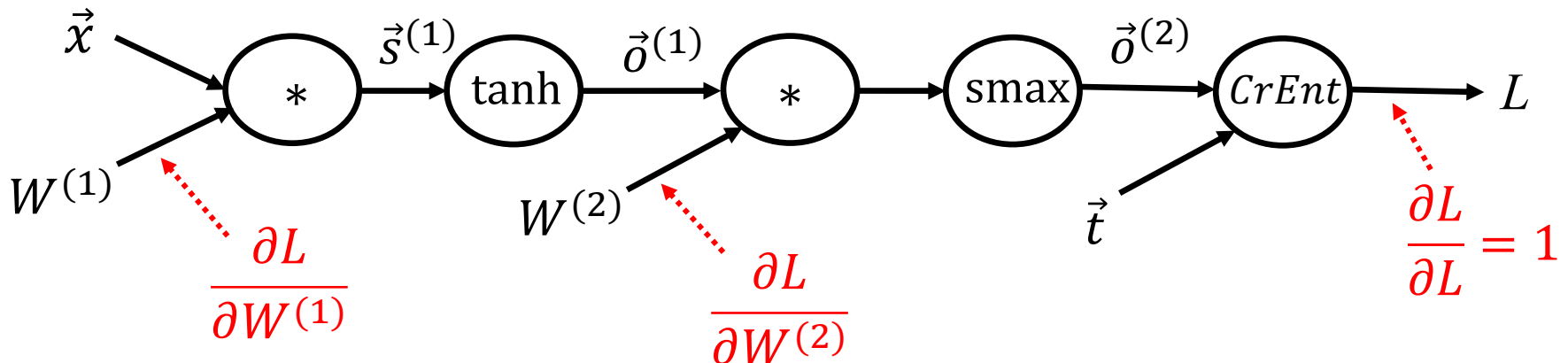


$$\vec{o}^{(1)} = \tanh(W^{(1)}\vec{x})$$

$$\vec{o}^{(2)} = \text{softmax}(W^{(2)}\vec{o}^{(1)})$$

**Cross-entropy loss** during training

Or as a **computation graph**:



# Extracting contract elements

THIS AGREEMENT is made the 15th day of October 2009  
(The “Effective Date”) BETWEEN:

- (1) Sugar 13 Inc., a corporation whose office is at James House, 42-50 Bond Street, London, EW2H TL (“Sugar”);
- (2) E2 UK Limited, a limited company whose registered office is at 260 Bathurst Road, Yorkshire, SL3 4SA (“Provider”).

## RECITALS:

- A. The Parties wish to enter into a framework agreement which will enable Sugar, from time to time, to [...]
- B. [...]

## NO THEREFORE IT IS AGREED AS FOLLOWS:

### ARTICLE I - DEFINITIONS

- “Sugar” shall mean: Sugar 13 Inc.  
“Provider” shall mean: E2 UK Limited  
“1933 Act” shall mean: Securities Act of 1933

### ARTICLE II - TERMINATION

The Service Period will be for five (5) years from the Effective Date (The “Initial Term”). The agreement is considered to be terminated in October 16, 2014.

### ARTICLE III - PAYMENT - FEES

During the service period monthly payments should occur. The estimated fees for the Initial Term are £154,800.

### ARTICLE IV - GOVERNING LAW

This agreement shall be governed and construed in accordance with the Laws of England & Wales. Each party hereby irrevocably submits to the exclusive jurisdiction of the courts sitting in Northern London.

IN WITNESS WHEREOF, the parties have caused their respective duly authorized officers to execute this Agreement.

BY: George Fake  
Authorized Officer  
Sugar 13 Inc.

BY: Olivier Giroux  
CEO  
E2 UK LIMITED

Identify start/end dates,  
duration, contractors, amount,  
legislations refs, jurisdiction  
etc. Similar to Named Entity  
Recognition (NER).

# Window-based NER example

$i$ -th word of the text being classified

3-word **window** (often larger)

yesterday language **tech** announced that...

**1-hot vectors** ( $|V| \times 1$ ) of the words in the **window**. ( $|V|$  is the **vocabulary size**).

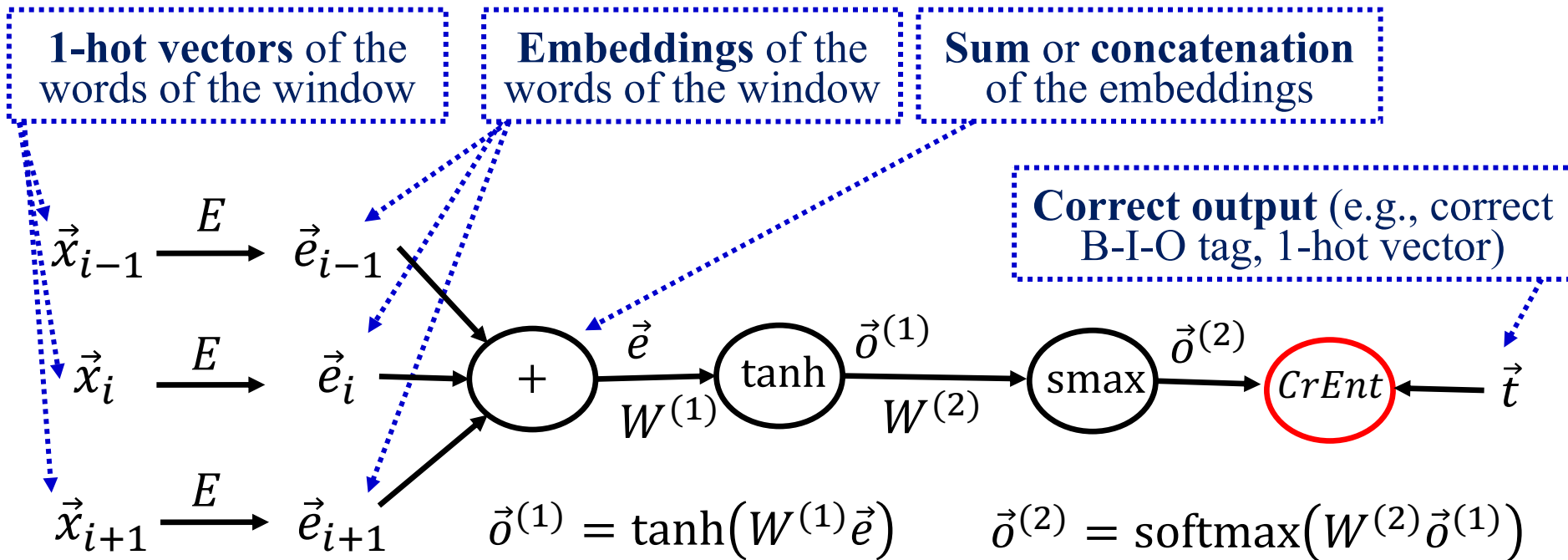
$$\vec{x}_{i-1} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} \quad \vec{x}_i = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \dots \\ 0 \end{bmatrix} \quad \vec{x}_{i+1} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ \dots \\ 1 \end{bmatrix}$$

**Embeddings** ( $d \times 1$ ) of the words in the **window**. ( $d$  is the **dimensionality** of the word embeddings).

$$\vec{e}_{i-1} = \begin{bmatrix} 1.8 \\ 2.3 \\ -1.4 \\ 3.7 \\ \dots \\ -1.1 \end{bmatrix} \quad \vec{e}_i = \begin{bmatrix} 2.4 \\ -3 \\ 9.3 \\ 5.1 \\ \dots \\ 3.9 \end{bmatrix} \quad \vec{e}_{i+1} = \begin{bmatrix} 2.2 \\ 3.8 \\ 1.2 \\ -6.4 \\ \dots \\ 7.1 \end{bmatrix}$$

Let  $E$  be a matrix ( $d \times |V|$ ) that **contains all the embeddings** of the **vocabulary as columns**. Then:  
 $\vec{e}_{i-1} = E\vec{x}_{i-1}$ ,  $\vec{e}_i = E\vec{x}_i$ , ...

# Window-based NER example



We learn  $W^{(1)}$ ,  $W^{(2)}$  with **backpropagation**. We can also learn (or modify) the **word embeddings  $E$**  during **backpropagation**! But when we don't have large training datasets (e.g., corpus manually annotated with B-I-O tags), it may be better to use **pre-trained embeddings**, which can be obtained from large non-annotated corpora (e.g., via Word2Vec, GloVe, to be discussed).

We can use the same window-based approach for **POS-tagging**, **chunking**, ...

# Cross-entropy loss

Word being classified.

3-word window (often larger).

yesterday language **tech** announced that...

$$\vec{o} = \begin{bmatrix} P_m(C = c_1) \\ P_m(C = c_2) \\ P_m(C = c_3) \\ \dots \\ P_m(C = c_k) \end{bmatrix} = \begin{bmatrix} 0.05 \\ 0.12 \\ 0.08 \\ \dots \\ 0.14 \end{bmatrix}$$

Probability estimates produced by the classifier for the class of the word “tech”.

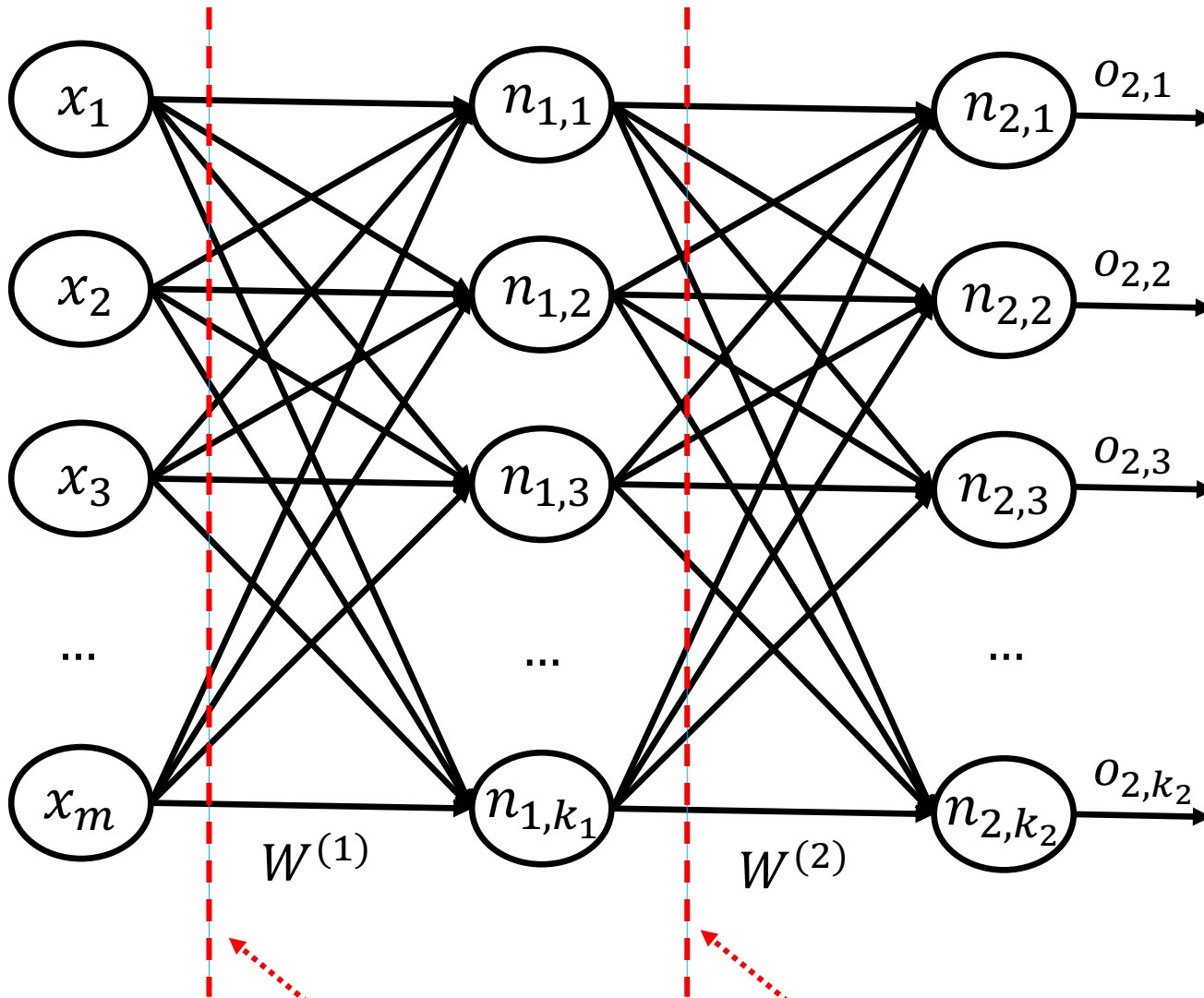
The correct “probabilities” for the class of “tech”. A 1-hot vector.

$$\vec{t} = \begin{bmatrix} P(C = c_1) \\ P(C = c_2) \\ P(C = c_3) \\ \dots \\ P(C = c_k) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \dots \\ 0 \end{bmatrix}$$

The log-likelihood of the correct class according to the classifier (with a minus sign).

$$H_{P_m}(C) = - \sum_{i=1}^k P(C = c_i) \log_2 P_m(C = c_i) = - \log_2 P_m(C = c_2)$$

# Dropout



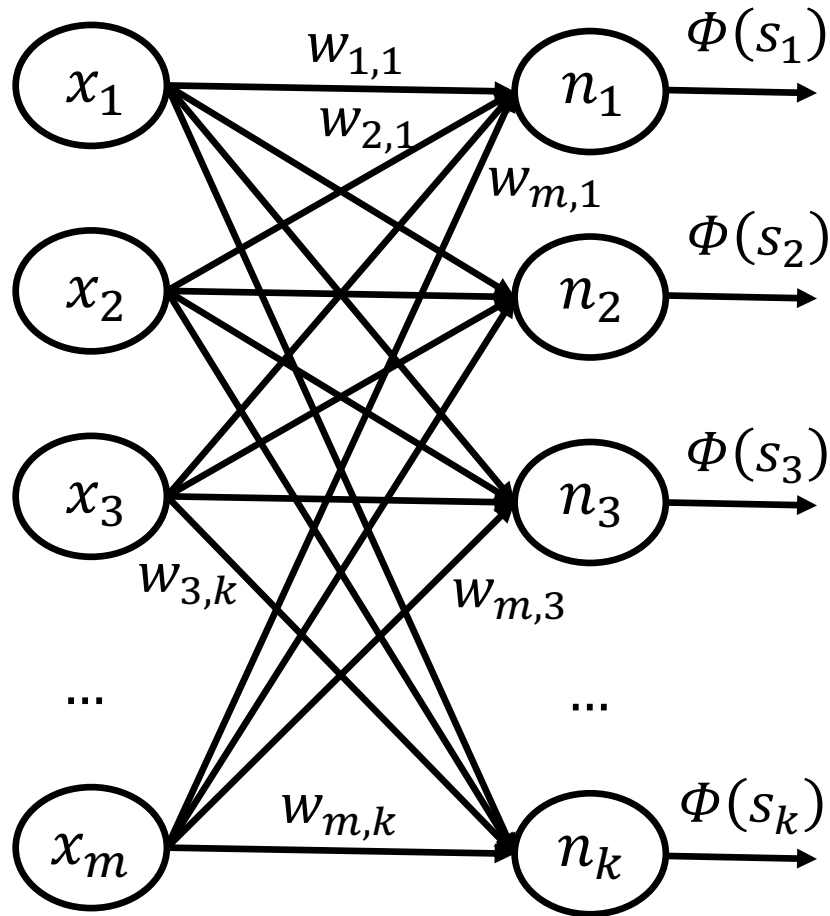
Dropout at the input layer.  
E.g.  $p_{drop} = 0.2$ .

Dropout at the output of a hidden  
layer. E.g.,  $p_{drop} = 0.5$ .

# Dropout

- **For each training instance** (or mini-batch), we **drop** (remove) **each neuron** of the layer where dropout is applied with **probability**  $p_{drop} = 1 - p_{keep}$ .
  - Helps the neural net **avoid relying too much on particular neurons** (or inputs). A form of **regularization**. Works well!
  - **Gradients** also **do not flow** through dropped neurons.
  - Alternative explanation: we train an **ensemble** of networks, containing **all the pruned networks** that dropout creates.
- **During testing**, we **multiply the output** of each neuron (of the layer where dropout was applied) by  $p_{keep}$ , so that the neuron's **expected output value** will be **as in training**.
  - **Or we divide** the output by  $p_{keep}$  **during training** instead.
  - **We don't drop neurons during testing** (only during training).

# Batch normalization



At each layer, instead of:

$$s_j = \sum_{i=1}^m w_{i,j} x_i$$

we use:

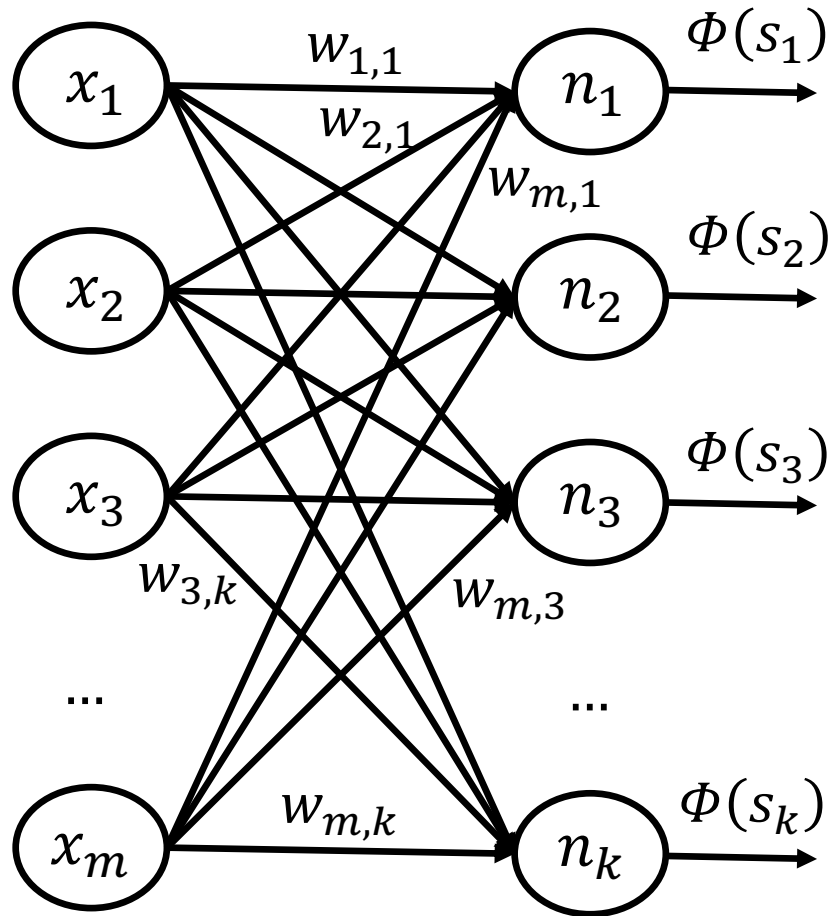
$$\bar{s}_j = \frac{g_j}{\sigma_j} (s_j - \mu_j) + b_j$$

- $\mu_j, \sigma_j$  are the **mean** and **std. dev. of  $s_j$**  in the **mini-batch**,
- $g_j, b_j$  are **learned** parameters (constant after training).
- $\Phi$  should be applied to  $\bar{s}_j$ .

See <https://arxiv.org/pdf/1607.06450.pdf> for **batch vs. layer normalization**.



# Layer normalization



At each layer, instead of:

$$s_j = \sum_{i=1}^m w_{i,j} x_i$$

we use:

$$\bar{s}_j = \frac{g_j}{\sigma} (s_j - \mu) + b_j$$

- $\mu, \sigma$  are the **mean** and **std. dev. of  $s_1, \dots, s_k$**  in the layer,
- $g_j, b_j$  are **learned** parameters (constant after training).

See <https://arxiv.org/pdf/1607.06450.pdf> for **batch vs. layer normalization**.  
The latter is better for RNNs (next part), where layers are time-steps.

# Pretraining word embeddings with Word2Vec (skip-gram version)

- Every word  $w$  of the vocabulary has two vectors:

$$\vec{w}^{(in)}, \vec{w}^{(out)}$$

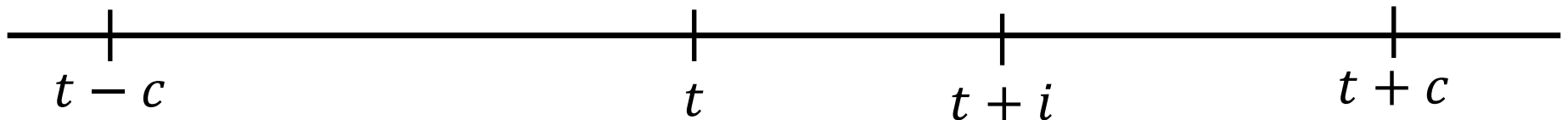
- The **vectors** are **randomly initialized**. We **learn** them.
- For every **token**  $w_t$  at **position**  $t$  of a **corpus** and **every position**  $t + i$  ( $i \neq 0$ ) within a **window**  $[t - c, t + c]$  around position  $t$ :

$$w_{t+i} = \dots?$$

$$w_{t+i} = \text{"starring"}?$$

$$w_{t+i} = \text{"directed"}?$$

$$w_t = \text{"film"} \quad w_{t+i} = \text{"famous"}?$$



- We want to be able to predict **which vocabulary word** occurs at **position**  $t + i$  by multiplying  $\vec{w}_t^{(in)}$  and  $\vec{w}_{t+i}^{(out)}$ .

# Word2Vec (skip-gram version)

In the **skip-gram** version of Word2Vec, the **central word** of each window “**predicts**” the **other words** of the **window**. In the **CBOW version**, the **context** (sum of the embeddings of the other words of the window) “**predicts**” the **central word**.

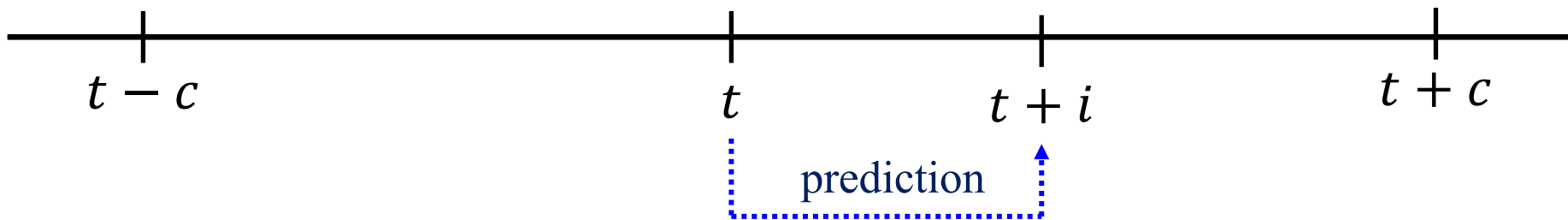
$$w_{t+i} = \dots?$$

$$w_{t+i} = \text{"starring"}$$

$$w_{t+i} = \text{"directed"}$$

$$w_{t+i} = \text{"famous"}$$

$$w_t = \text{"film"}$$



$$P(w_{t+i}|w_t) = \text{softmax} \left( \vec{w}_{t+i}^{(out)} \cdot \vec{w}_t^{(in)} \right)$$

$$= \frac{\exp \left( \vec{w}_{t+i}^{(out)} \cdot \vec{w}_t^{(in)} \right)}{\sum_{w \in V} \exp \left( \vec{w}^{(out)} \cdot \vec{w}_t^{(in)} \right)}$$

# Word2Vec (skip-gram version)

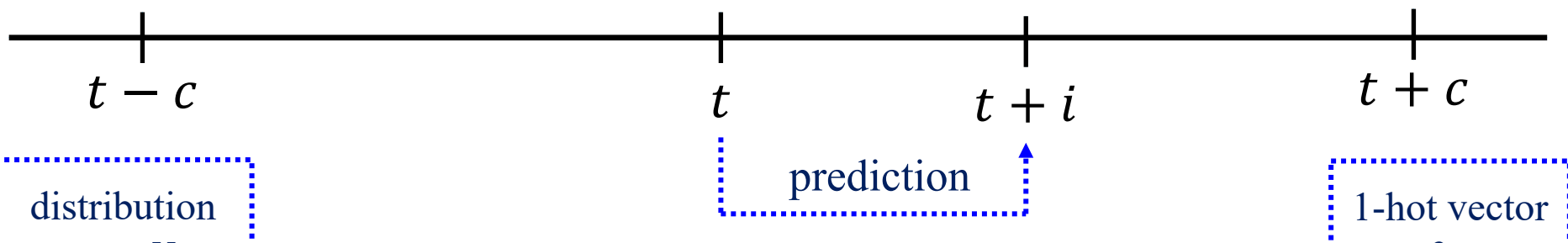
- We learn the  $\vec{w}^{(in)}, \vec{w}^{(out)}$  by maximizing the probability assigned to the  $w_{t+i}$  that actually occurs at each position  $t + i$ , i.e., we maximize the likelihood of the correct predictions:

$$\langle \hat{E}^{(in)}, \hat{E}^{(out)} \rangle = \operatorname{argmax}_{\langle E^{(in)}, E^{(out)} \rangle} \sum_{t=1+c}^{T-c} \sum_{-c \leq i \leq c, i \neq 0} \ln P(w_{t+i} | w_t) \quad L$$

- Matrices  $E^{(in)}, E^{(out)}$  contain in their columns all the *in* and *out* vectors (word embeddings) of all vocabulary words.
- $T$  is the corpus size,  $t$  is the center of the sliding window.
- For each  $t$  value, we get  $2c$  training examples.
- For batch gradient ascent, we would do steps:
$$\langle \hat{E}^{(in)}, \hat{E}^{(out)} \rangle \leftarrow \langle \hat{E}^{(in)}, \hat{E}^{(out)} \rangle + \eta \nabla L$$
- In practice, we use SGD (or variants), i.e., we use the likelihood  $L_i$  of a mini-batch of training examples (e.g., all  $2c$  of a window):
$$\langle \hat{E}^{(in)}, \hat{E}^{(out)} \rangle \leftarrow \langle \hat{E}^{(in)}, \hat{E}^{(out)} \rangle + \eta \nabla L_i$$

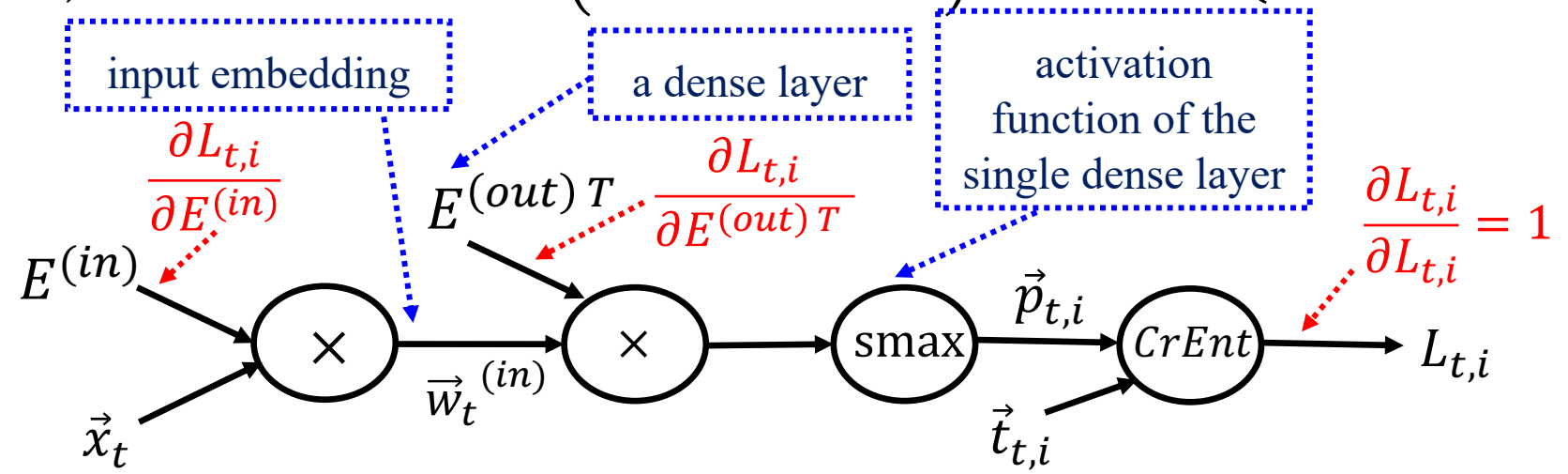
# Word2Vec (skip-gram) as a shallow NN

$w_{t+i} = \dots?$   
 $w_{t+i} = \text{"starring"}$ ?  
 $w_t = \text{"film"}$       $w_{t+i} = \text{"famous"}$ ?



$$P(w_{t+i}|w_t) = \text{softmax}(\vec{w}_{t+i}^{(out)} \cdot \vec{w}_t^{(in)})$$

$$\vec{p}_{t,i}(\vec{w}_t^{(in)}) = \text{softmax}(E^{(out)T} \vec{w}_t^{(in)}) = \text{softmax}(E^{(out)T} E^{(in)} \vec{x}_t)$$



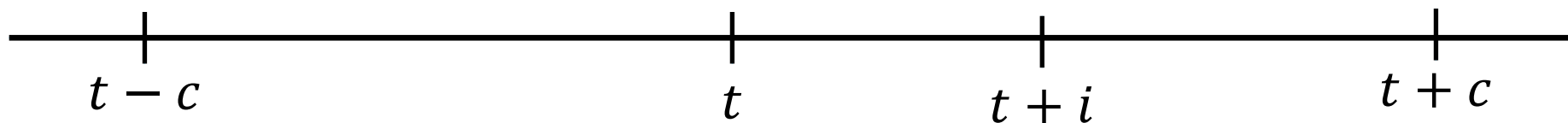
# Word2Vec (skip-gram with negative sampling)

When the **vocabulary  $V$**  is **large**, computing the **softmax** is very **time-consuming**. A more efficient alternative is **negative sampling**, a kind of **contrastive learning**.

We construct **positive (+)** and **negative (-)** examples, using the word  $w_{t+i}$  that **actually occurs** at position  $t + i$ , and **random words  $r_{t+i}$**  that **do not actually occur** at position  $t + i$ .

$r_{t+i} = \text{"medical"}$  (random, negative)

$w_t = \text{"film"}$      $w_{t+i} = \text{"famous"}$  (true, positive)



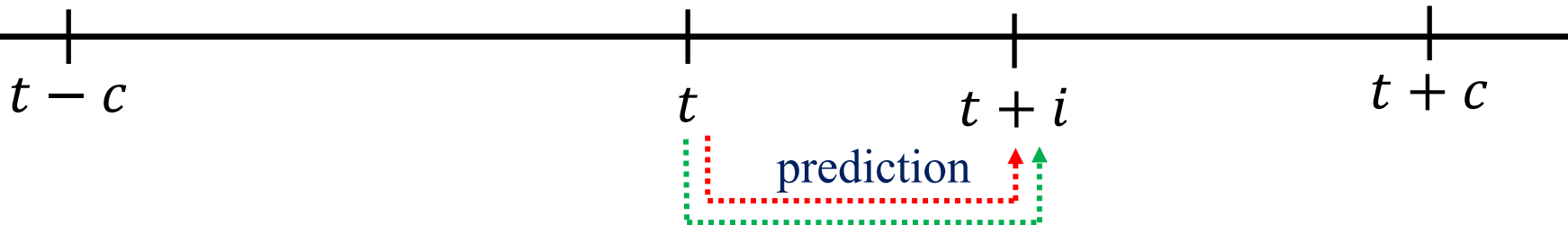
$$\max_{\langle E^{(in)}, E^{(out)} \rangle} \sum_{t=1+c}^{T-c} \sum_{-c \leq i \leq c, i \neq 0} \ln P(+ | w_{t+i}, w_t) + \ln P(- | r_{t+i}, w_t)$$

We try to learn to assign **high probabilities** to the **correct classes**. In practice, we use **multiple random words  $r_{t+i}$**  at each position  $t + i$ .

# Word2Vec (skip-gram with negative sampling)

$r_{t+i}$  = "medical" (random, negative)

$w_t$  = "film"     $w_{t+i}$  = "famous" (true, positive)



$$\max_{\langle E^{(in)}, E^{(out)} \rangle} \sum_{t=1+c}^{T-c} \sum_{-c \leq i \leq c, i \neq 0} \ln P(+|w_{t+i}, w_t) + \ln [1 - P(+|r_{t+i}, w_t)]$$

$$\sum_{t=1+c}^{T-c} \sum_{-c \leq i \leq c, i \neq 0} \ln \sigma(\vec{w}_{t+i}^{(out)} \cdot \vec{w}_t^{(in)}) + \ln [1 - \sigma(\vec{r}_{t+i}^{(out)} \cdot \vec{w}_t^{(in)})]$$

sigmoid as in logistic regression

We **no longer** try to produce a **probability distribution over the vocabulary** for the words  $w_{t+i}$  that may occur at  $t+i$ .

We are **given**  $w_t$  and a **particular**  $w_{t+i}$  or  $r_{t+i}$  and we need to decide if it is a **positive** or **negative** case.

# Loss as a function of epochs

Figure 5.1. Canonical overfitting behavior

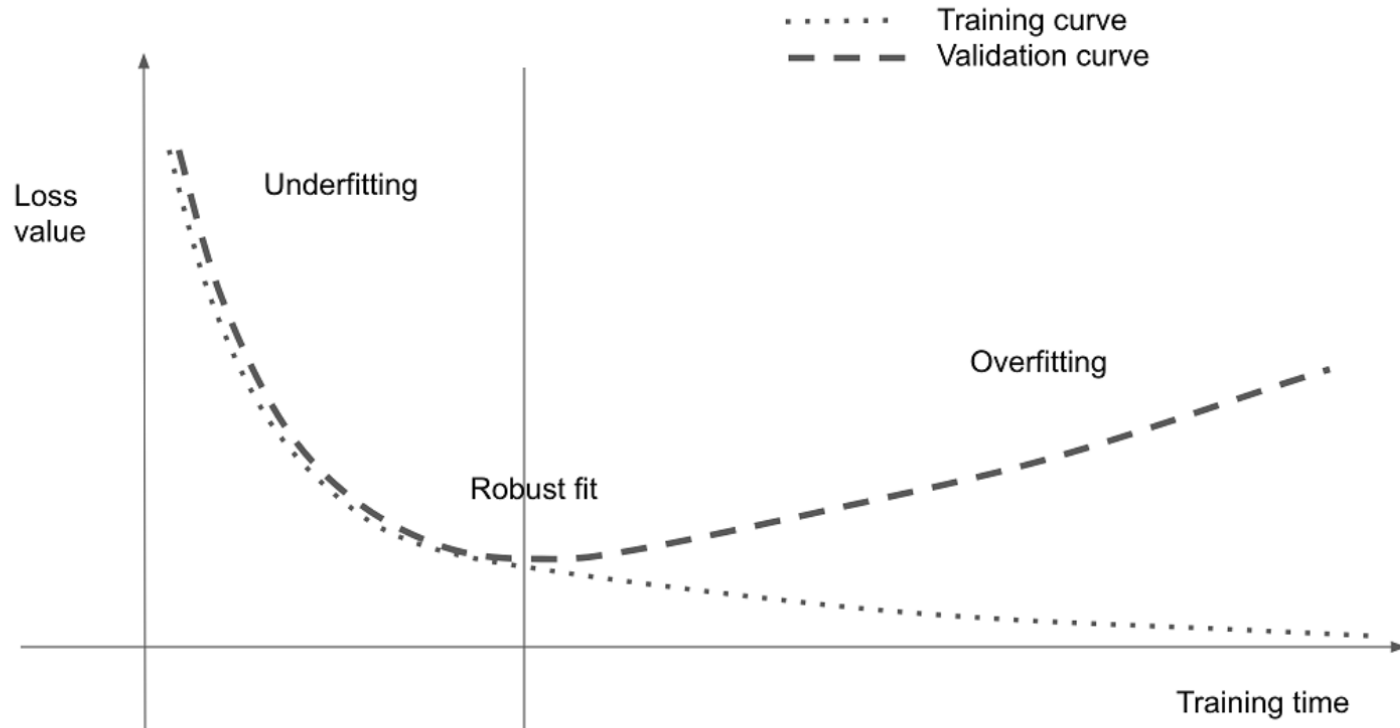


Figure from the **recommended book “Deep Learning with Python”** by F. Chollet, Manning Publications, 2nd edition. The 1<sup>st</sup> edition is freely available.

<https://www.manning.com/books/deep-learning-with-python>

<https://www.manning.com/books/deep-learning-with-python-second-edition>



# Practical advice for training deep NNs

- **Check simple baselines:** (e.g., majority, random, ...)
  - If you can't beat them, you may have **bugs, data problems, ...**
  - Look at how data are **tokenized, pre-processed, ...**
- **Get the training and validation loss to start falling:**
  - Tune the **learning rate** and the **mini-batch size**.
  - Use appropriate **models** (e.g., for sequences, images, ...).
  - Examine misclassification **errors** (e.g., extreme/frequent cases).
- **Reach the overfitting behavior** of the previous slide.
  - The **training and validation loss** (or metric) **both fall up to a point**, then the **training loss continues to improve** ideally reaching **near zero**, the **validation loss deteriorates**.
  - **Increase capacity** (e.g., **layers, neurons per layer**), ...
- **Then dropout, early stopping, batch/layer norm, ...**
- **Check Chollet's Chapter 5** for more advice...

# Regularizing a high-capacity model

Figure 5.21 Effect of dropout on validation loss

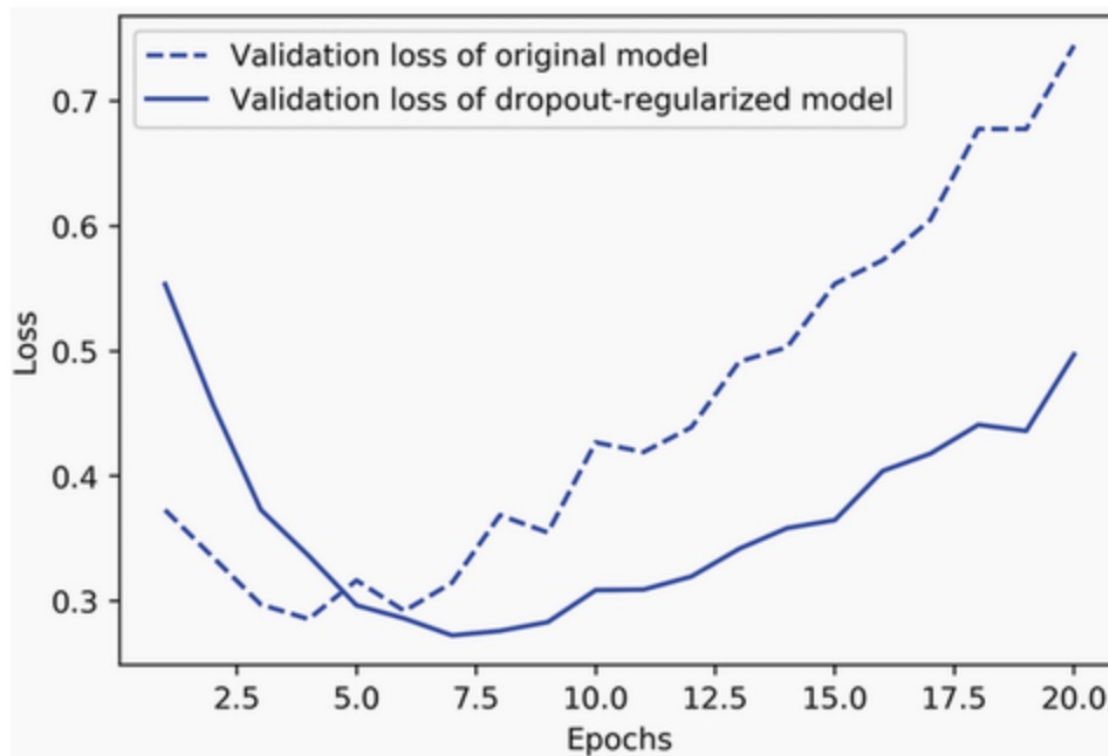


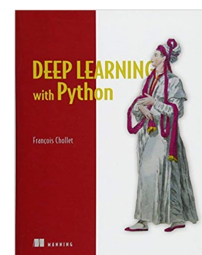
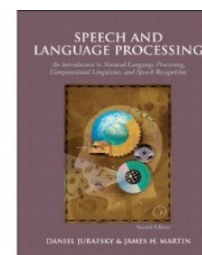
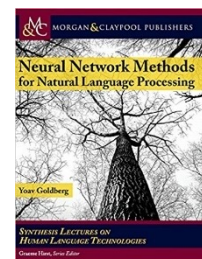
Figure from the **recommended book “Deep Learning with Python”** by F. Chollet, Manning Publications, 2nd edition. The 1<sup>st</sup> edition is freely available.

<https://www.manning.com/books/deep-learning-with-python>

<https://www.manning.com/books/deep-learning-with-python-second-edition>

# Recommended reading

- Y. Goldberg, *Neural Network Models for Natural Language Processing*, Morgan & Claypool Publishers, 2017.
  - Mostly chapters 3–5 and 10. Available at AUEB's library.
- Jurafsky & Martin's, *Speech and Language Processing* is being revised (3<sup>rd</sup> ed.) to include Deep Learning methods.
  - <http://web.stanford.edu/~jurafsky/slp3/>
- For an introduction to Keras (used in labs) and practical DL for NLP and vision, see F. Chollet's *Deep Learning in Python*, Manning Publications, 1st edition, 2017.
  - The 1<sup>st</sup> edition is freely available and sufficient for this course. <https://www.manning.com/books/deep-learning-with-python>
  - 2<sup>nd</sup> edition (2022) now available, requires payment. Highly recommended.



# Other recommended resources

- Useful maths background: T. Parr και J. Howard, *The Matrix Calculus You Need for Deep Learning*.
  - <https://explained.ai/matrix-calculus>
- PyTorch tutorials:
  - <https://pytorch.org/tutorials/>
- C. Manning's (Stanford) *NLP with Deep Learning* course.
  - Course material: <http://web.stanford.edu/class/cs224n/>
  - Videos: available on YouTube.
- See also the recommended books of Part 0 (Introduction).