# Neural Networks

## Neural Language Models

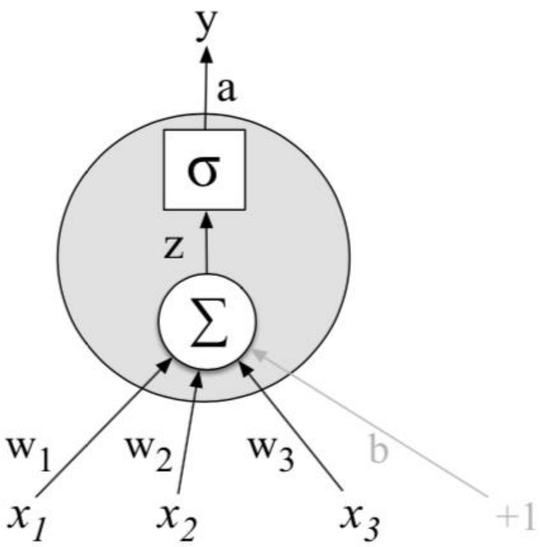
T-622-ARTI

Stefán Ólafsson
Spring 2023

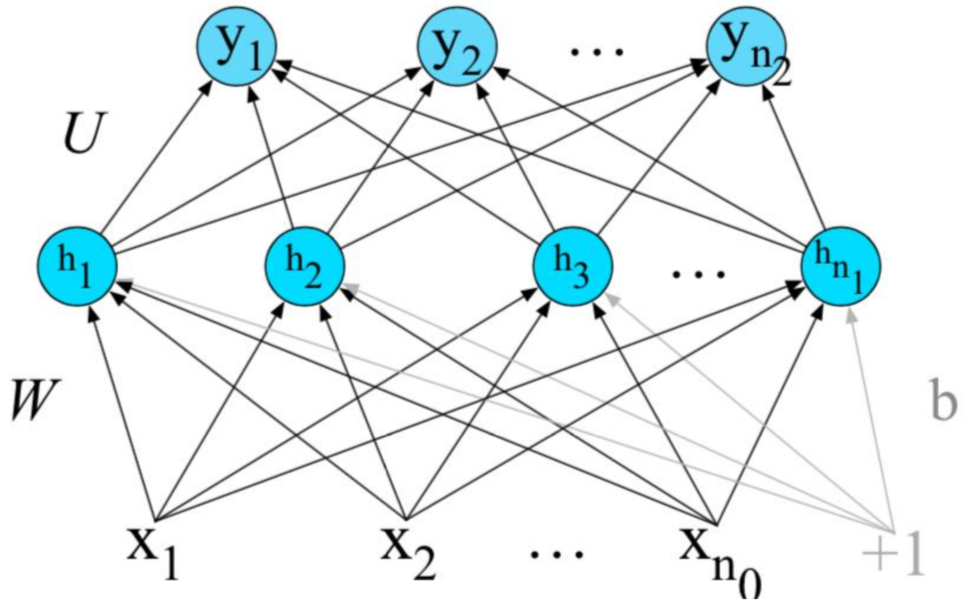
# Neural Network

- **A network of small computing units (nodes)**
- **Each unit takes a vector of input values**
- **Produces a single output value**

**Single unit**




**Network**



# Feed-forward network and deep learning

- **Feed-forward:**
  - Computation proceeds iteratively from one layer of units to the next
- **Deep learning:**
  - Networks that are deep (many layers)

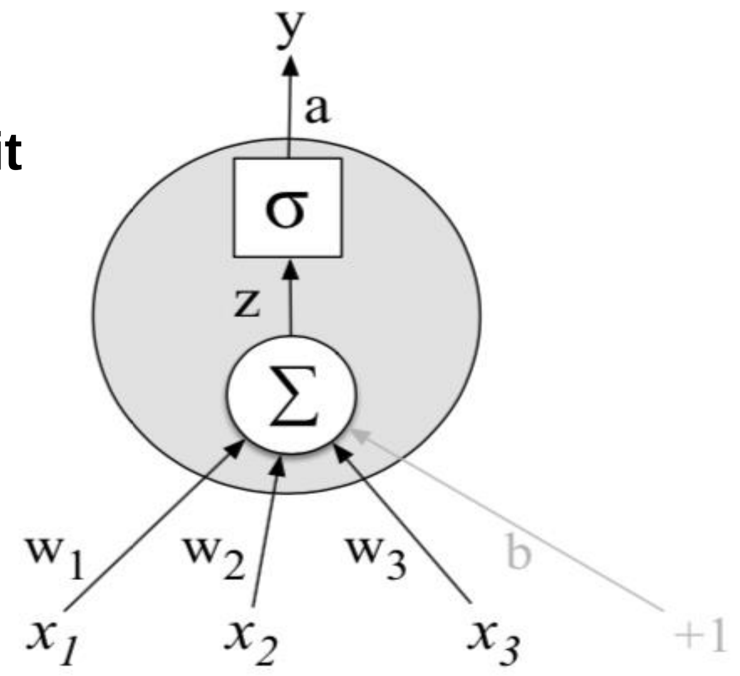
# Neural Network (NN) and logistic regression

- **NN share much of the same mathematics as logistic regression**
  - But a more powerful classifier than logistic regression
  - Because of these hidden layers
- **Logistic regression uses feature templates based on domain knowledge**
- **More common in NN to avoid the use of hand-derived features**
  - Take raw words as inputs
  - Learns to induce features as part of the learning process

# Units

- A neural unit takes a **weighted sum** of its input and adds a **bias term**
- Input: $x_1, x_2 \ldots x_n$
- Weights: $w_1, w_2 \ldots w_n$

**Single unit**



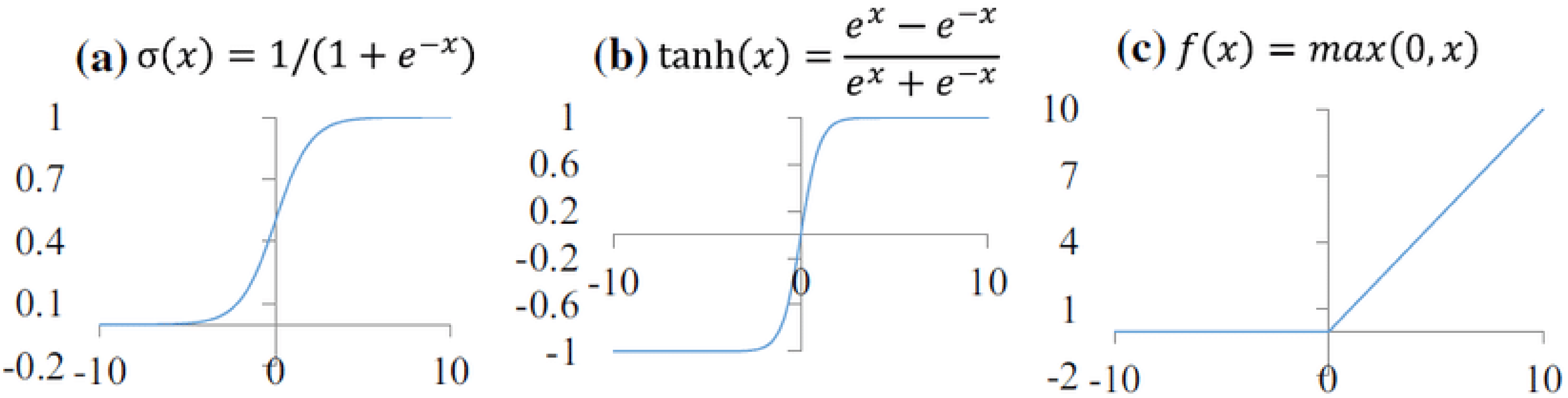

- $\mathrm{z} = \sum_{i=1}^{n} w_i x_i + \mathrm{b}$
- $\mathrm{z} = \mathrm{w} * \mathrm{x} + \mathrm{b}$

# Activation value

- **Neural units apply a non-linear function *f* to *z* to produce the output, the activation value, *a*.**

$$y = a = f(z)$$

- **Popular non-linear functions: <u>sigmoid</u>, <u>tanh</u>, <u>ReLU</u>**


(a) $\sigma(x) = 1/(1+e^{-x})$
1
0.7
0.4
0.1
-0.2
-10
0
10
(b) $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
1
0.6
0.2
-0.2
-0.6
-1
-10
0
10
(c) $f(x) = max(0, x)$
10
7
4
1
-2
-10
0
10

# Example

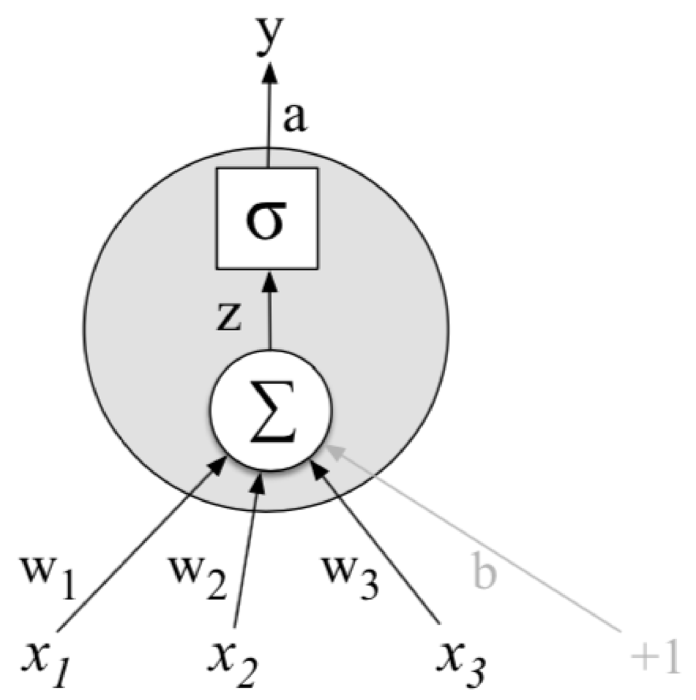



**Figure 7.2** A neural unit, taking 3 inputs $x_1$, $x_2$, and $x_3$ (and a bias $b$ that we represent as a weight for an input clamped at +1) and producing an output y. We include some convenient intermediate variables: the output of the summation, $z$, and the output of the sigmoid, $a$. In this case the output of the unit $y$ is the same as $a$, but in deeper networks we'll reserve $y$ to mean the final output of the entire network, leaving $a$ as the activation of an individual node.

# Why do we need non-linear activation functions?

- **to introduce *non-linearity* into the network**
- **this allows you to model output that varies non-linearly with its input variables**
- **if we only allow linear activation functions in a neural network, the output will just be a linear transformation of the input**
- **non-linear means that the output cannot be reproduced from a linear combination of the input**

# Why do we need multi-layer networks?

- **Because a single neural unit cannot compute some very simple functions of its input.**

# Logical AND and OR

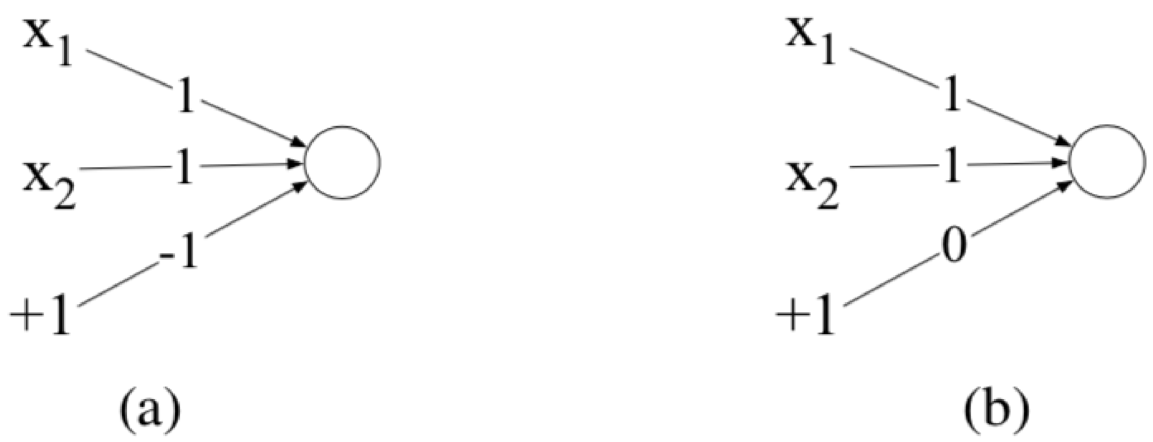



**Figure 7.4** The weights $w$ and bias $b$ for perceptrons for computing logical functions. The inputs are shown as $x_1$ and $x_2$ and the bias as a special node with value $+1$ which is multiplied with the bias weight $b$. (a) logical AND, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = -1$. (b) logical OR, showing weights $w_1 = 1$ and $w_2 = 1$ and bias weight $b = 0$. These weights/biases are just one from an infinite number of possible sets of weights and biases that would implement the functions.

$$y = \begin{cases} 0, if\ w * x + b \ \leq 0 \\ 1, if w * x + b \ > 0 \end{cases}$$

**Perceptron**: binary classifier; purely linear

# Logical XOR

- **Not possible to build a perceptron to compute logical XOR!**
- **A perceptron is a linear classifier**
- **XOR is not a linearly separable function**

# Linearly separable

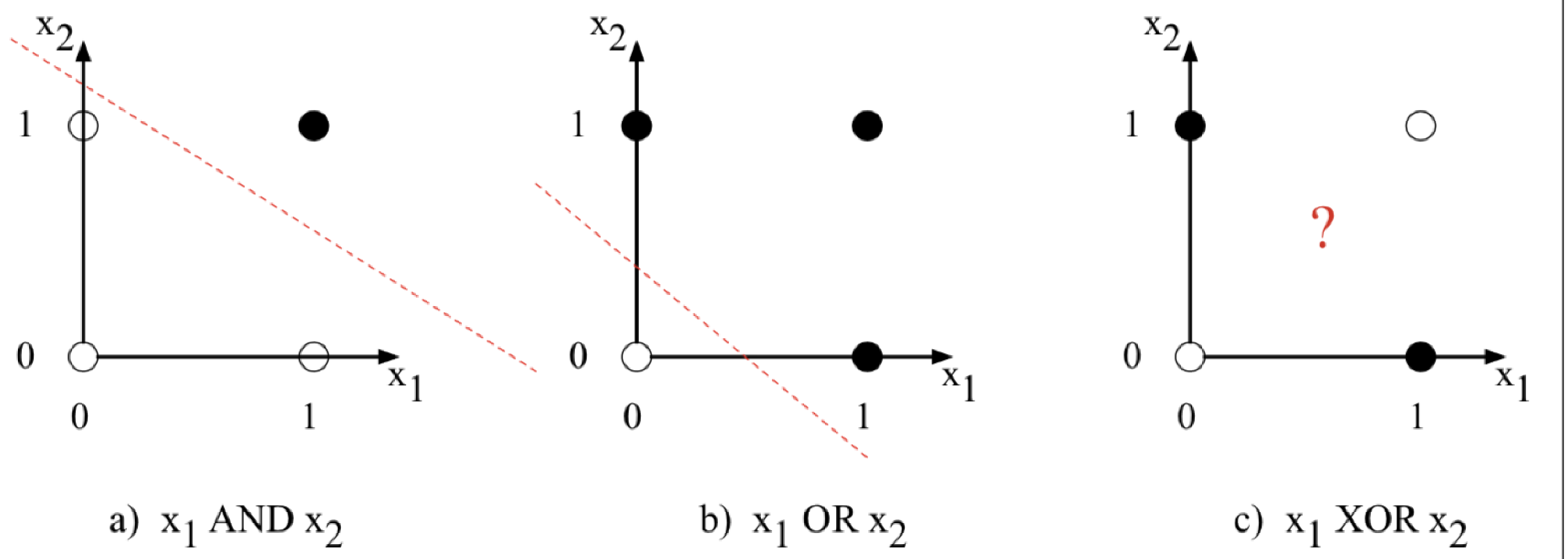



**Figure 7.5** The functions AND, OR, and XOR, represented with input $x_0$ on the x-axis and input $x_1$ on the y axis, Filled circles represent perceptron outputs of 1, and white circles perceptron outputs of 0. There is no way to draw a line that correctly separates the two categories for XOR. Figure styled after Russell and Norvig (2002).

# XOR solution

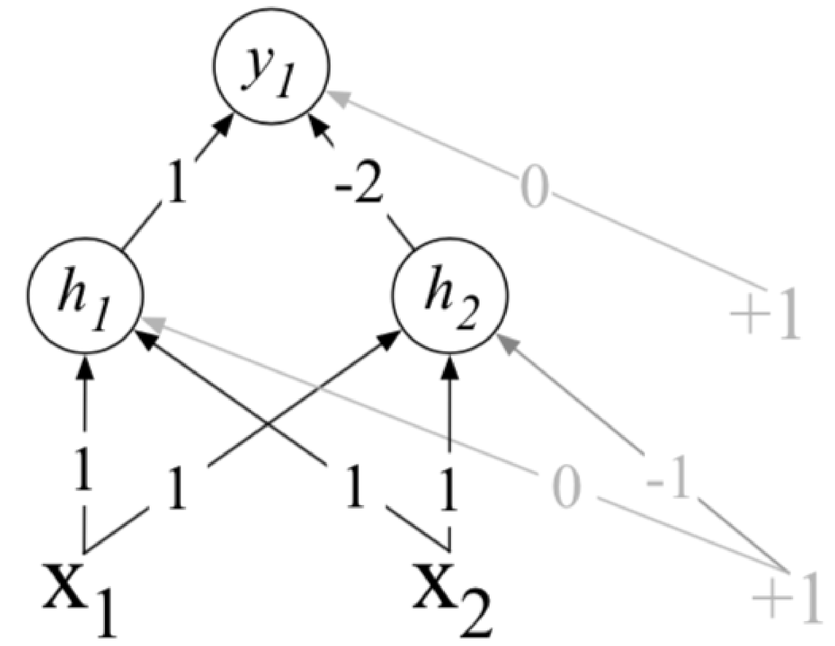



**Figure 7.6** XOR solution after Goodfellow et al. (2016). There are three ReLU units, in two layers; we've called them $h_1$, $h_2$ ($h$ for "hidden layer") and $y_1$. As before, the numbers on the arrows represent the weights $w$ for each unit, and we represent the bias $b$ as a weight on a unit clamped to +1, with the bias weights/units in gray.

**You should work out what happens for the inputs!**

# New representation

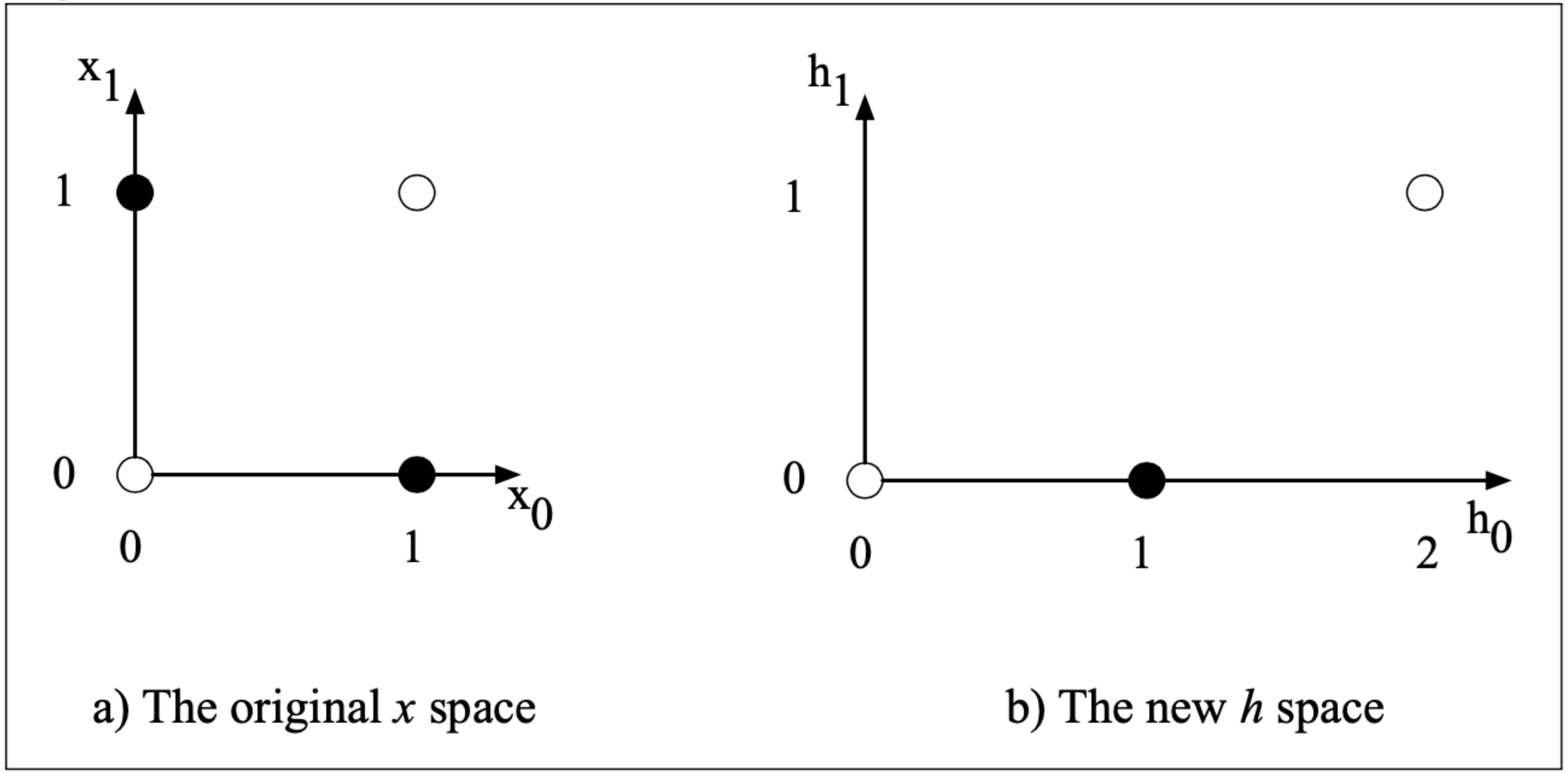



**Figure 7.7** The hidden layer forming a new representation of the input. Here is the representation of the hidden layer, $h$, compared to the original input representation $x$. Notice that the input point [0 1] has been collapsed with the input point [1 0], making it possible to linearly separate the positive and negative cases of XOR. After Goodfellow et al. (2016).

# Hidden layer

- **Key advantage of neural networks:**
  - Can automatically learn useful representations of the input

# Feed-forward Neural Networks

- **Simple feed-forward networks have three kinds of nodes:**
  - Input units
  - Hidden units
  - Output units

# Feed-forward Neural Networks

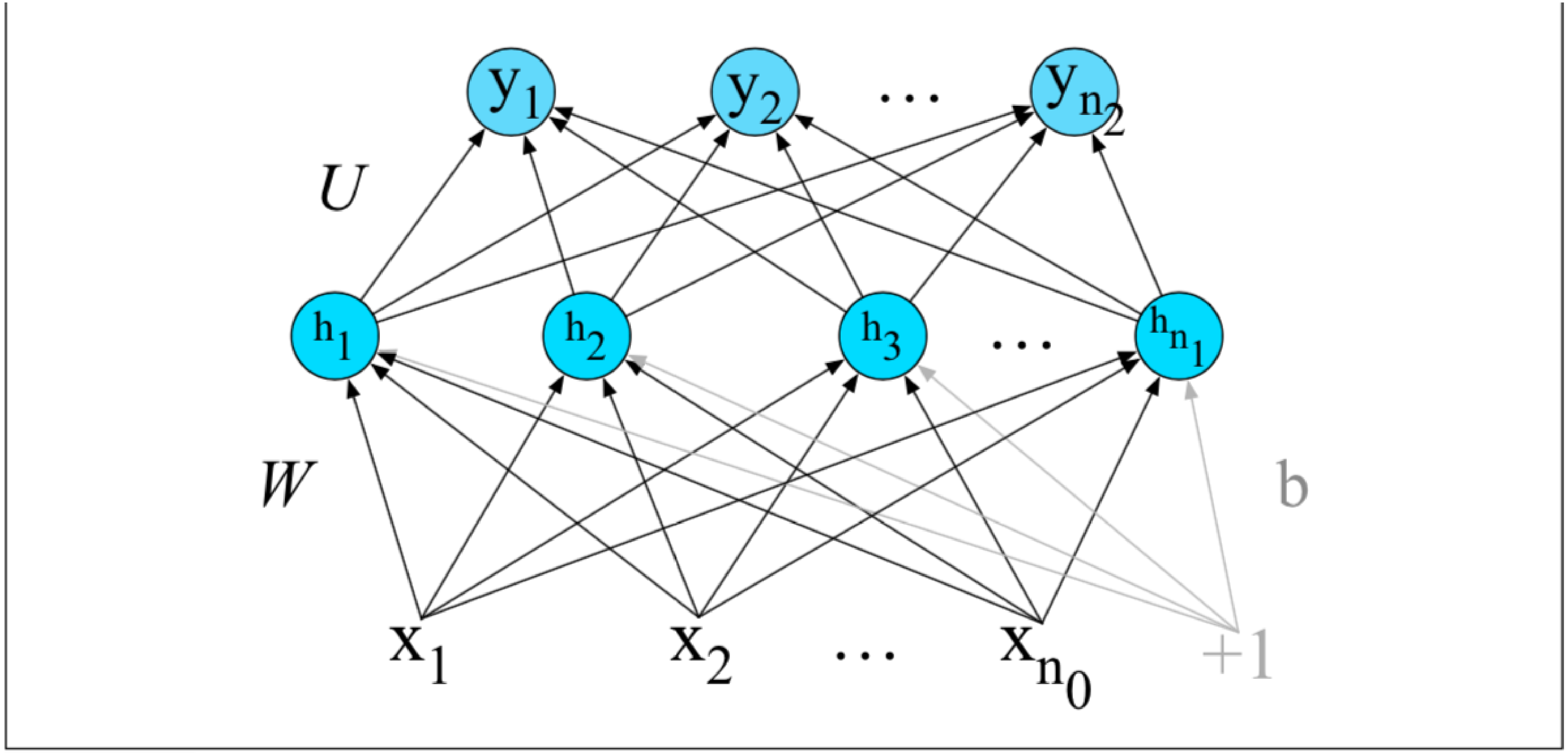
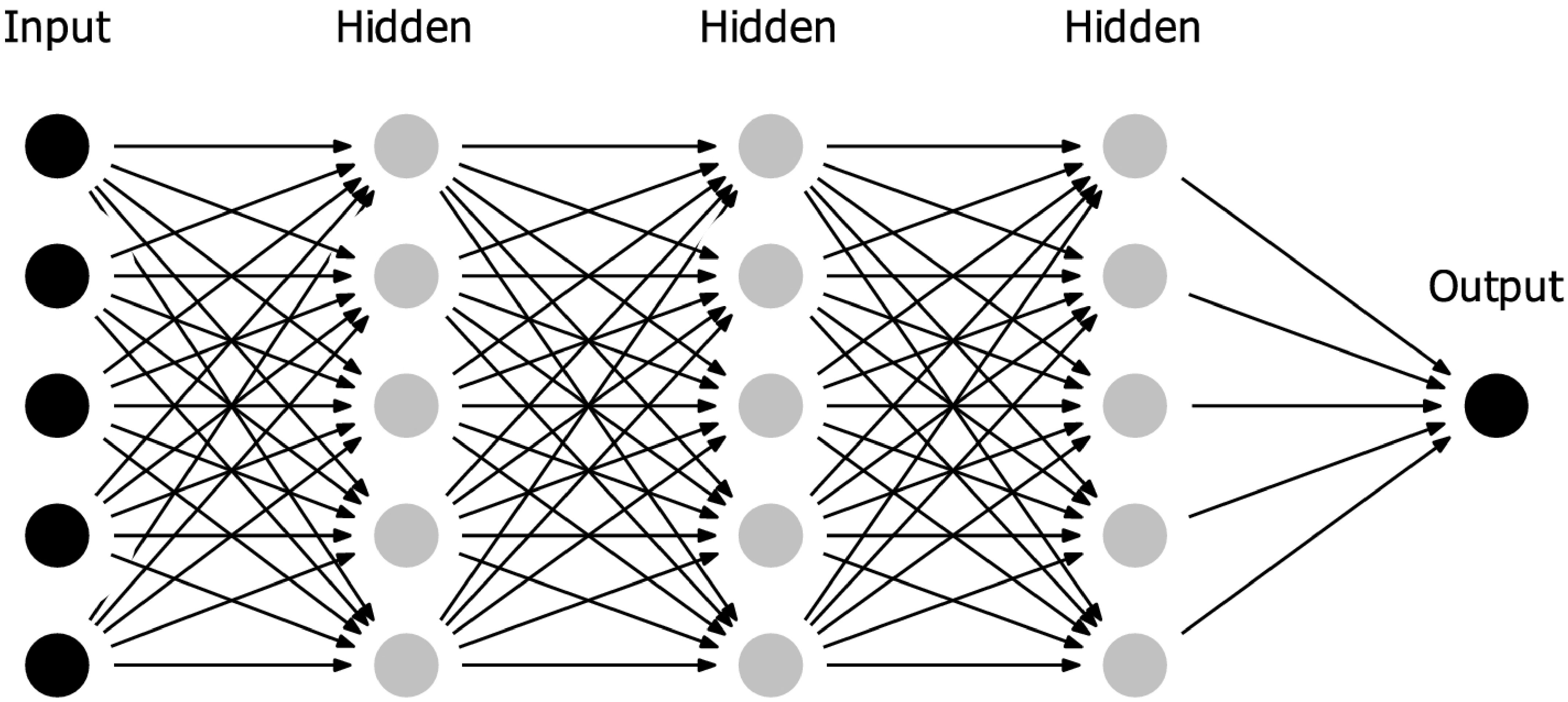



**Figure 7.8** A simple 2-layer feed-forward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

# Deep Feed-forward Neural Network



# Feed-forward Neural Networks

- Weight matrix **W** for a hidden layer
- $W_{ij}$: the weight of the connection from the i-th input $x_i$ to the j-th hidden unit $h_j$
- Allows for effective matrix operations
- $h = \sigma(W * x + b)$
- **W** is a matrix, **x, b** and **h** are vectors

# Output layer

- **h** forms a representation of the input
- The output layer takes this new representation **h** and computes a final output.
- The output could be a real value
- In many cases, however, the network makes a classification decision.

# Output layer

- If binary classification, then single output node
- y is then the probability of the positive class
- If multinomial classification (e.g. PoS tagging), then one output node for each possible PoS
- The output layer then gives a probability distribution over output nodes.

# Output layer

- The output layer has a weight matrix, **U**.
- z = U*h
- z is a vector of real-valued numbers
- For multiclass classification we need a vector of probabilities
- Can convert it to a probability distribution by using the **softmax** function (d is the dimension of z):
- $softmax(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{d} e^{z_j}} \quad 1 \leq i \leq d$

# Feed-forward Neural Networks

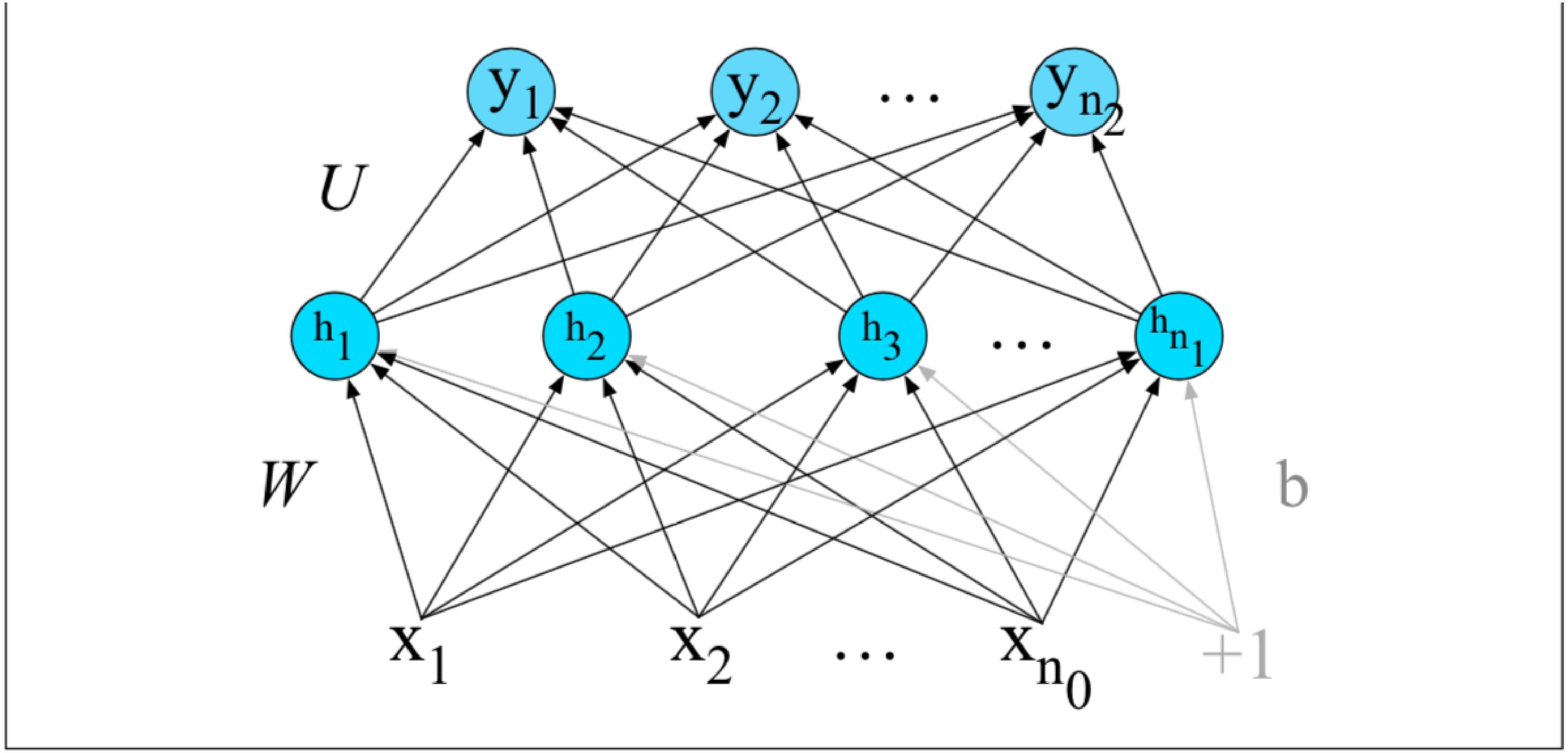



**Figure 7.8** A simple 2-layer feed-forward network, with one hidden layer, one output layer, and one input layer (the input layer is usually not counted when enumerating layers).

# Softmax example

- **Given a vector** $z$=[0.6, 1.1, -1.5, 1.2, 3.2, -1.1]
- **softmax(*z*) is** [0.055, 0.090, 0.0067, 0.10, 0.74, 0.010].

# Neural network classifier with one hidden layer

- Builds a vector **h**, a hidden layer representation of the input
- Then runs standard logistic regression on the features that the network develops in **h**.
- Deep neural network is like layer after layer of logistic regression classifiers:
  - Prior layers induce the feature representations themselves, as opposed to using hand-crafted features

# Neural network classifier with one hidden layer

- $h = \sigma(W * x + b)$
- z = U * h
- y = softmax(z)
- 2-layer network

# Training

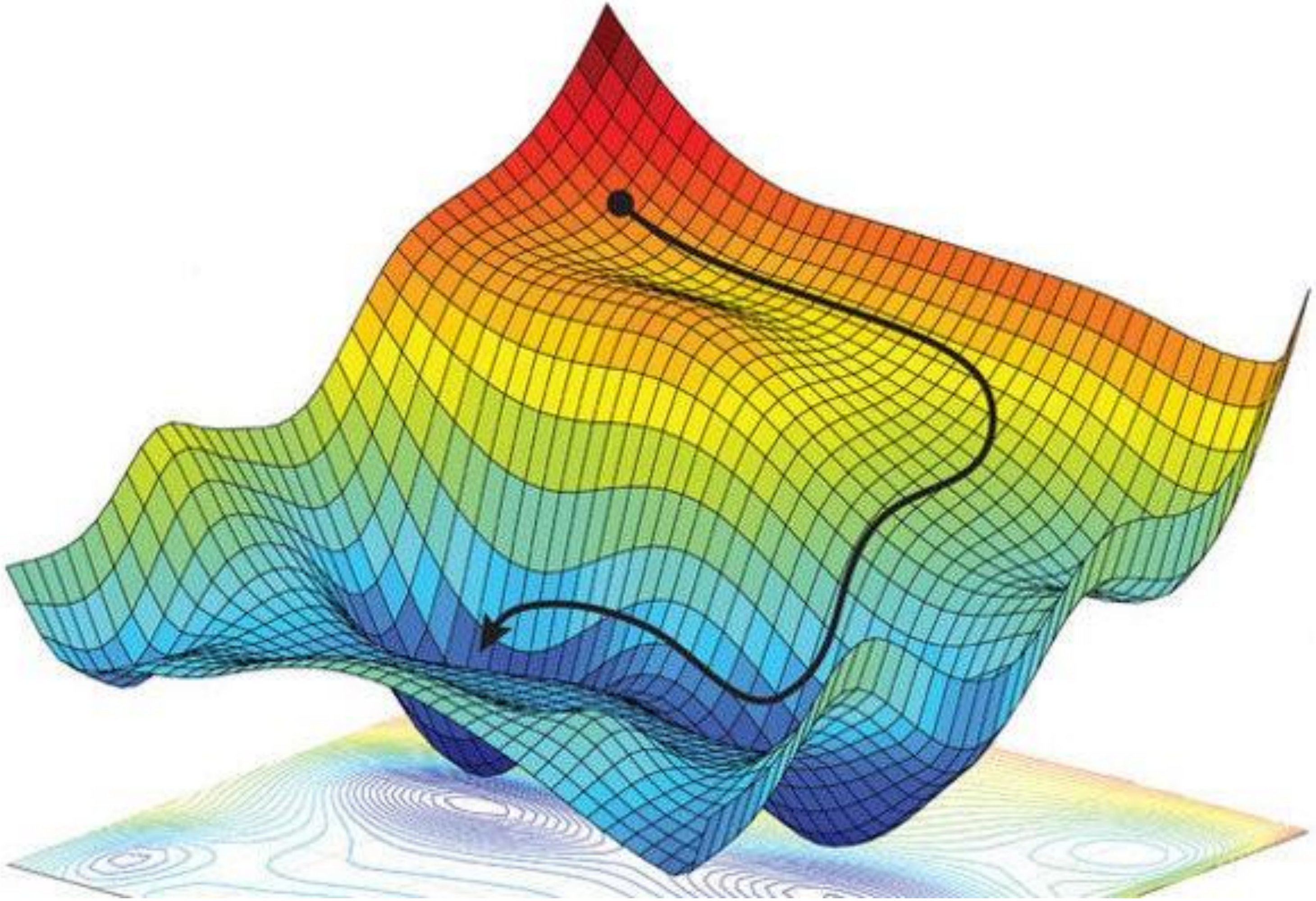
- Learn weights $\boldsymbol{W}^{[i]}$ and bias $\boldsymbol{b}^{[i]}$ that make $\hat{\boldsymbol{y}}$ for each training observation as close as possible to the true **y**.

1. loss (cost) function
2. Iteratively updating the weights => gradient descent

# Difference to logistic regression?

- In logistic regression, for each observation we directly compute the derivative of the loss function with respect to an individual **w** or **b**.
  - Cost function is convex
- For neural networks, with many layers, we need what is called **error back-propagation**
- See chapter 7.4 in the textbook

# More than one layer: non-convex



# Application: Neural Language Models (NLM)

- **Language modeling:**
  - Predicting upcoming words from prior word contexts
- **Advantages of NLMs compared to n-gram language models:**
  - Don't need smoothing
  - Can handle much longer histories
  - Can generalize over context of similar words
  - Usually give higher accuracy

# Neural Language Models (NLM)

- **Disadvantages of NLMs compared to n-gram language models:**
  - Strikingly slower to train

# Feed-forward neural language model

- Standard feedforward network
- Takes as input at time **t** a representation of some number of previous words ($w_{t-1}$, $w_{t-2}$ …)
- Output a probability distribution over possible next words
- $P(w_t|w_1^{t-1}) \approx P(w_t|w_{t-N+1}^{t-1})$
- N=4, 4-gram: P($w_t$ I $w_{t-1}$, $w_{t-2}$ , $w_{t-3}$)

# Word Embeddings

- Prior context is represented by embeddings of the previous words.
- Allows neural language models to generalize to unseen data, much better than n-gram LMs

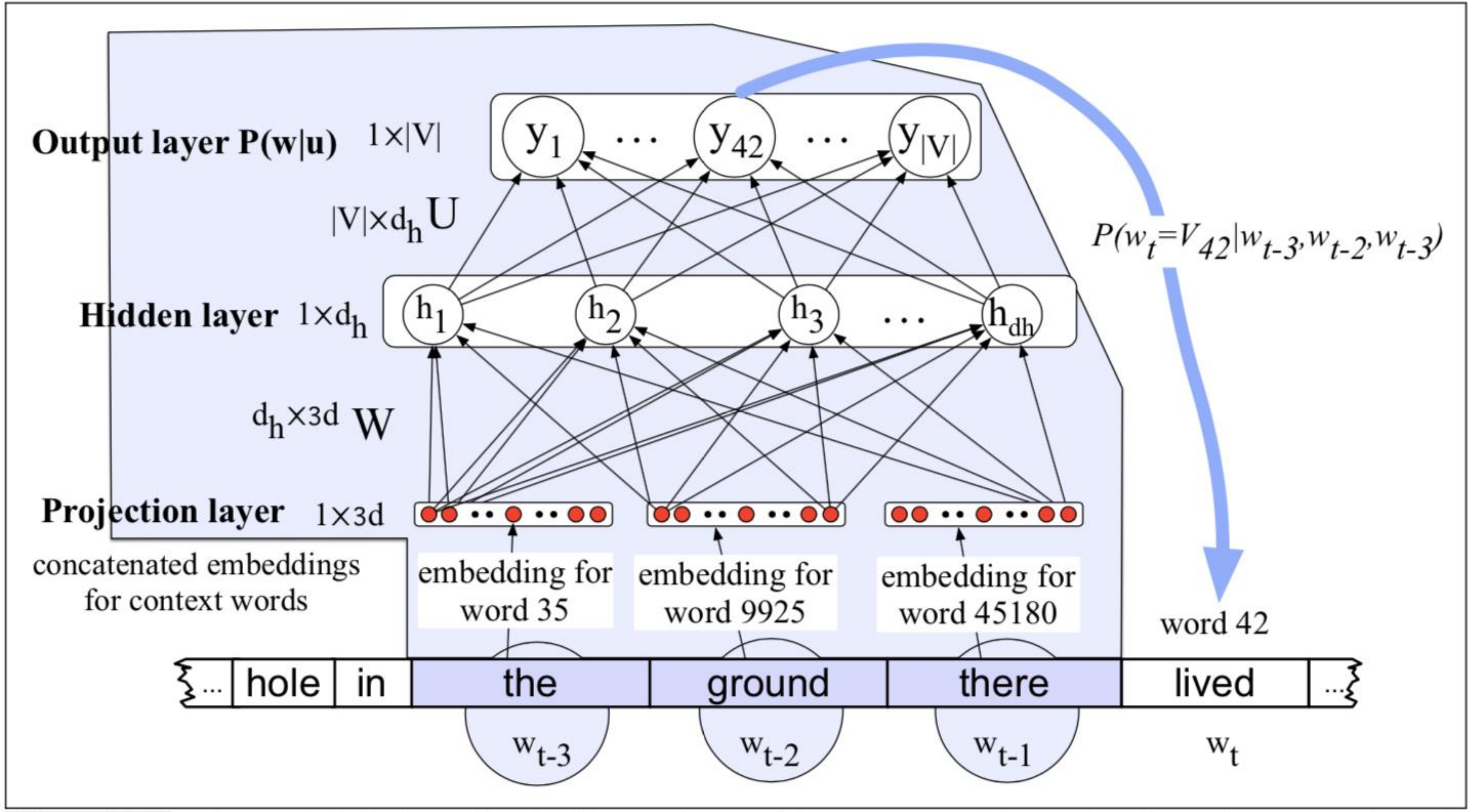



**Figure 7.12** A simplified view of a feedforward neural language model moving through a text. At each timestep $t$ the network takes the 3 context words, converts each to a $d$-dimensional embeddings, and concatenates the 3 embeddings together to get the $1 \times Nd$ unit input layer $x$ for the network. These units are multiplied by a weight matrix $W$ and bias vector $b$ and then an activation function to produce a hidden layer $h$, which is then multiplied by another weight matrix $U$. (For graphic simplicity we don't show $b$ in this and future pictures). Finally, a softmax output layer predicts at each node $i$ the probability that the next word $w_t$ will be vocabulary word $V_i$. (This picture is simplified because it assumes we just look up in an embedding dictionary $E$ the $d$-dimensional embedding vector for each word, precomputed by an algorithm like word2vec.)

# What if we do not have pre-trained embeddings?

- We then need to learn the embeddings during training of the network
- We represent each of the previous words as a one-hot vector
  - one dimension for each word in the vocabulary
  - [0 0 0 0 1 0 0 ... 0 0 0 0]

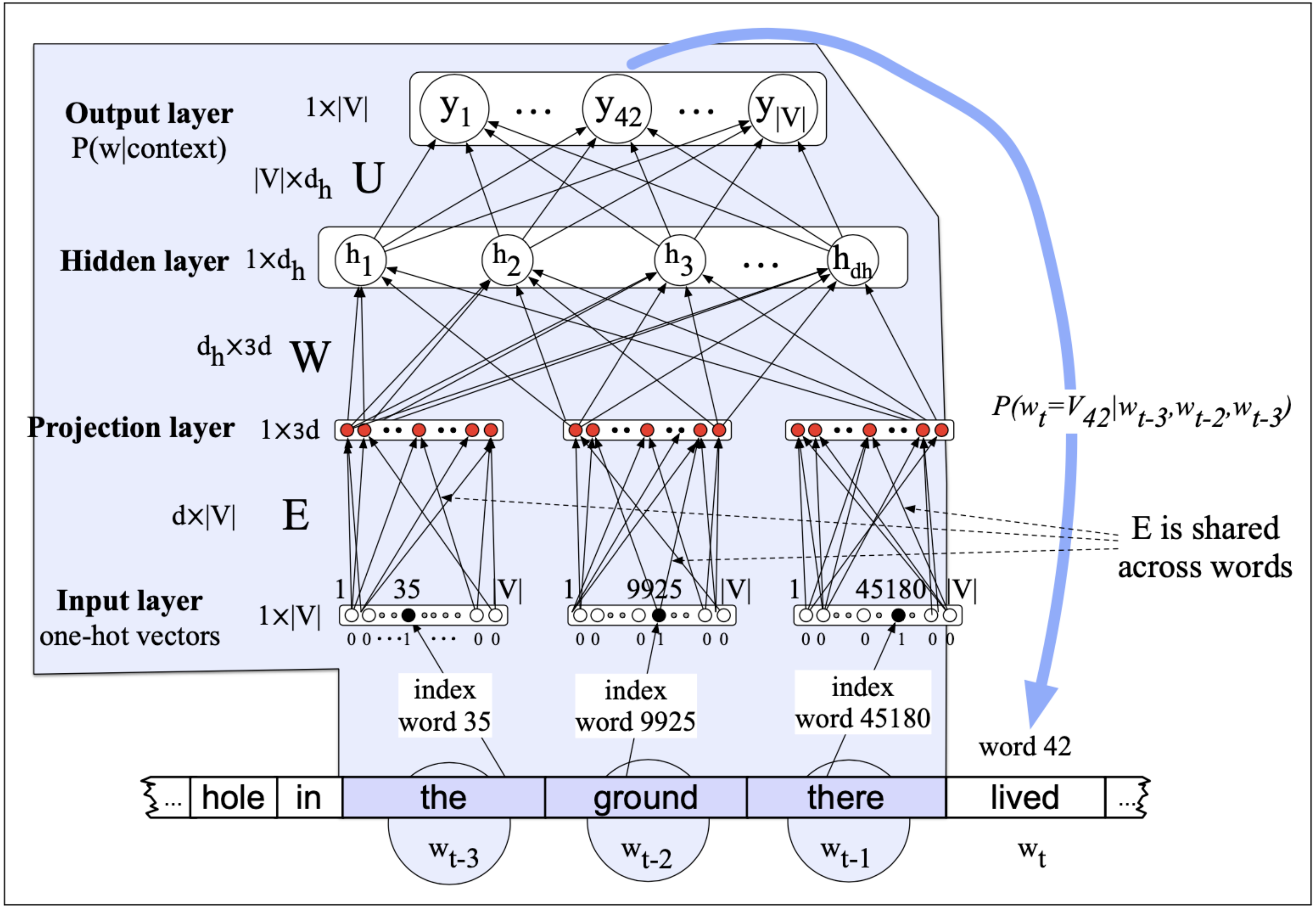



**Figure 7.13** Learning all the way back to embeddings. Notice that the embedding matrix $E$ is shared among the 3 context words.