



Cairo University

Cairo University

Cairo University
Faculty of Engineering
Computer Engineering Department

Natural Language Processing

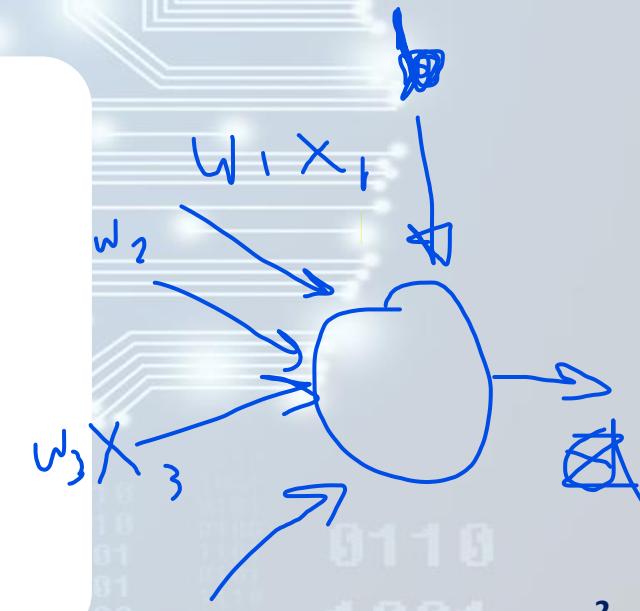
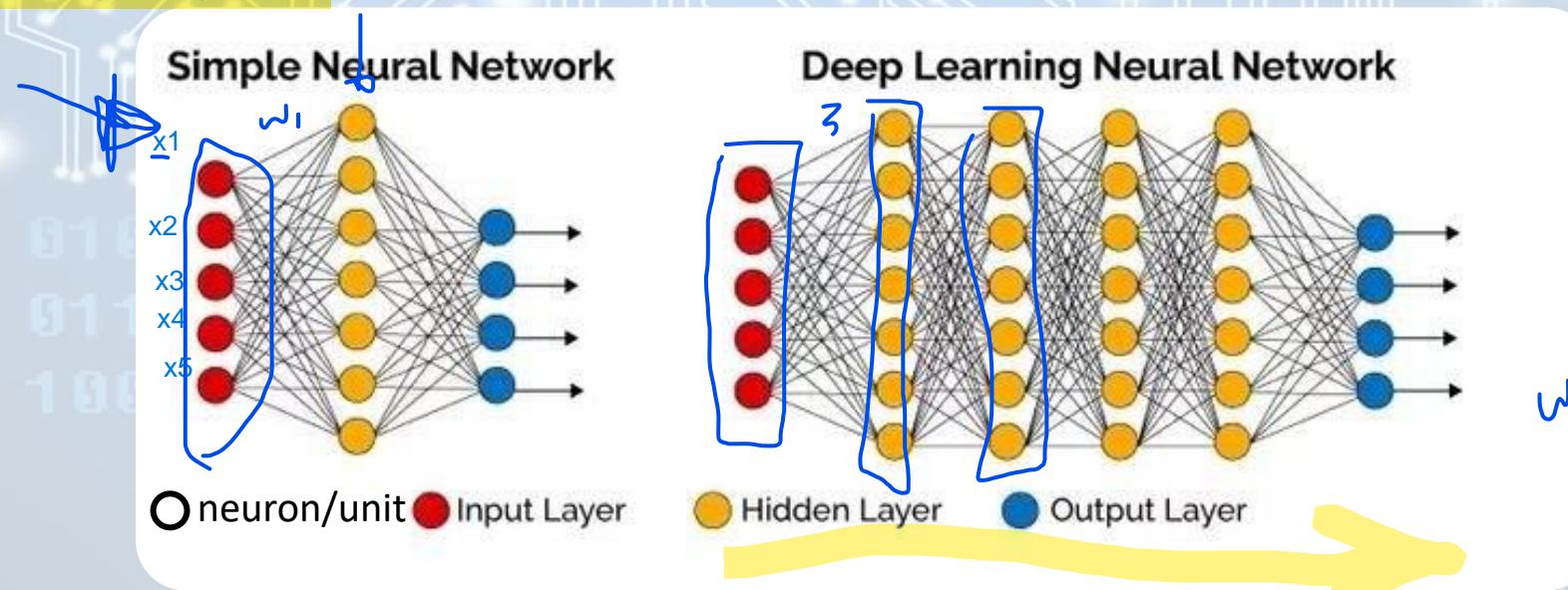
0100 0100
0110 0110
1001 1001
0100110
0100001
1101
1101

Dr. Sandra Wahid

learning processing
natural language NLP
text linguistics interaction
automatic programming technologies
understanding data science machine learning
linguistics conference automated networks
interaction data evolution evaluation systems
interaction communication artificial intelligence
interaction media machine learning
interaction communication artificial intelligence
interaction communication artificial intelligence

Overview

- Neural networks are a fundamental computational tool for language processing.
- Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.
- **Feedforward Network:** the computation proceeds iteratively from one layer of *units* to the next.
- **Deep Learning:** involves the use of modern neural nets that are often *deep* → have many layers.



Units/Neurons

- The building block of a neural network is a single computational unit.
 - A unit takes a set of real valued numbers as **input**, performs some computation on them, and produces an **output**.
- A neural unit takes a **weighted sum of its inputs**, with one additional term in the sum called a **bias term**.
 - Given a set of inputs x_1, x_2, \dots, x_n , a unit has a set of corresponding **weights** w_1, w_2, \dots, w_n and a **bias b** , so the weighted sum z can be represented as:
 - Using vectors:
$$z = \mathbf{w} \cdot \mathbf{x} + b$$
 where \cdot is the dot product
- Finally, instead of using z (a linear function of x) as the **output**, neural units apply a **non-linear function** f to z .
 - The output of this function is known as the **activation** value for the unit, a .
- The final **output** of the network is referred to as y .

$$a = f(z)$$

Example Activation Functions

- **Sigmoid function:**

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

- It maps the output into the range [0,1].
- It is useful in **squashing** outliers toward 0 or 1.
- It is **differentiable**.

- **tanh function:**

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

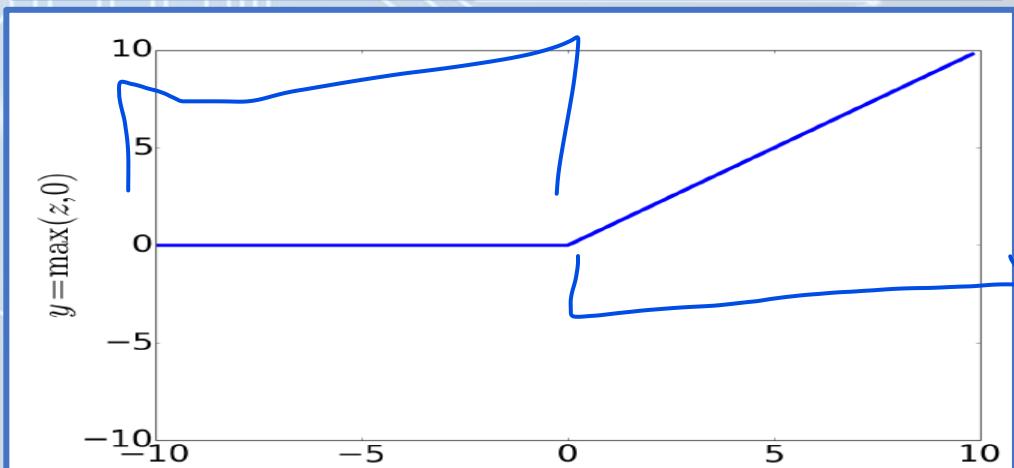
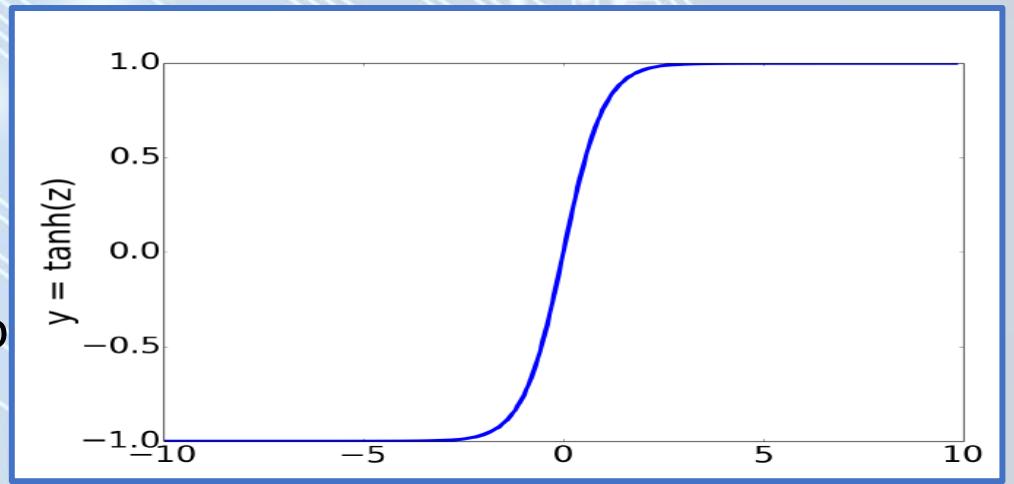
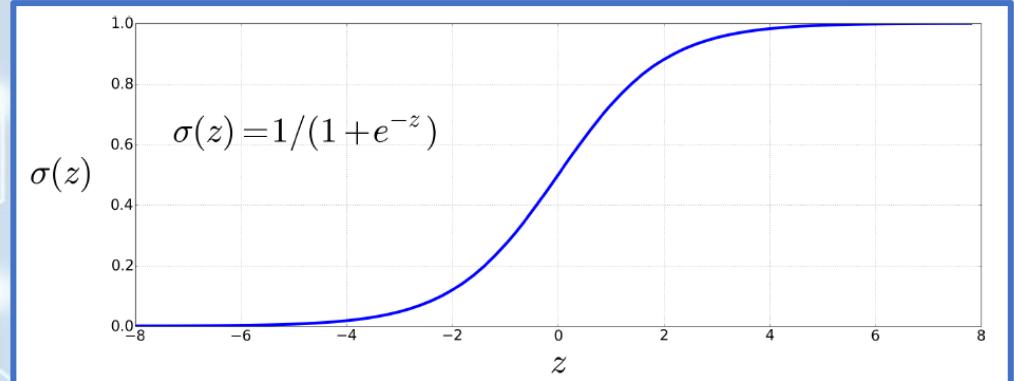
0 1 3 5

- It is a variant of the sigmoid that maps the output into the range [-1,1].
- It is **smoothly differentiable**.

- **ReLU function:**

$$\text{ReLU}(z) = \max(z, 0)$$

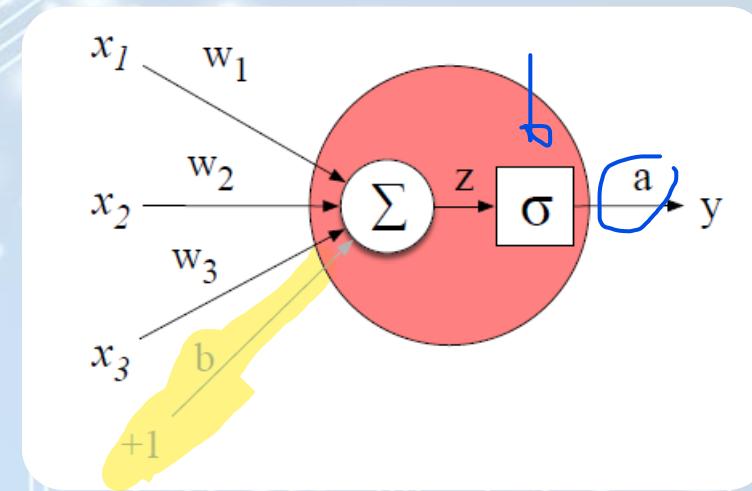
- **Rectified Linear Unit.**
- It is just the same as z when z is positive, and 0 otherwise.
- It gives a result that is **very close to linear**.



Basic Neural Unit Example

- The unit takes 3 input values x_1 , x_2 , and x_3 , and computes a weighted sum, multiplying each value by a weight (w_1 , w_2 , and w_3 , respectively), adds them to a bias term b , and then passes the resulting sum through a sigmoid function to result in a number between 0 and 1.

- 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.



- Using the following weight and bias values, what would this unit do with the following input vector x ?

$$\mathbf{w} = [0.2, 0.3, 0.9]$$

$$b = 0.5$$

$$\mathbf{x} = [0.5, 0.6, 0.1]$$

$$z = (0.5) + (0.2 \cdot 0.5 + 0.3 \cdot 0.6 + 0.9 \cdot 0.1) = 0.87$$

$$y = 1 / (1 + e^{-z}) = 0.7$$

$$y = \sigma(\mathbf{w} \cdot \mathbf{x} + b) = \frac{1}{1 + e^{-(\mathbf{w} \cdot \mathbf{x} + b)}} = \frac{1}{1 + e^{-(0.5 \cdot 2 + 0.6 \cdot 3 + 0.9 \cdot 0.1 + 0.5)}} = \frac{1}{1 + e^{-0.87}} = 0.70$$

0110
0100 001
1101

Feedforward Neural Networks

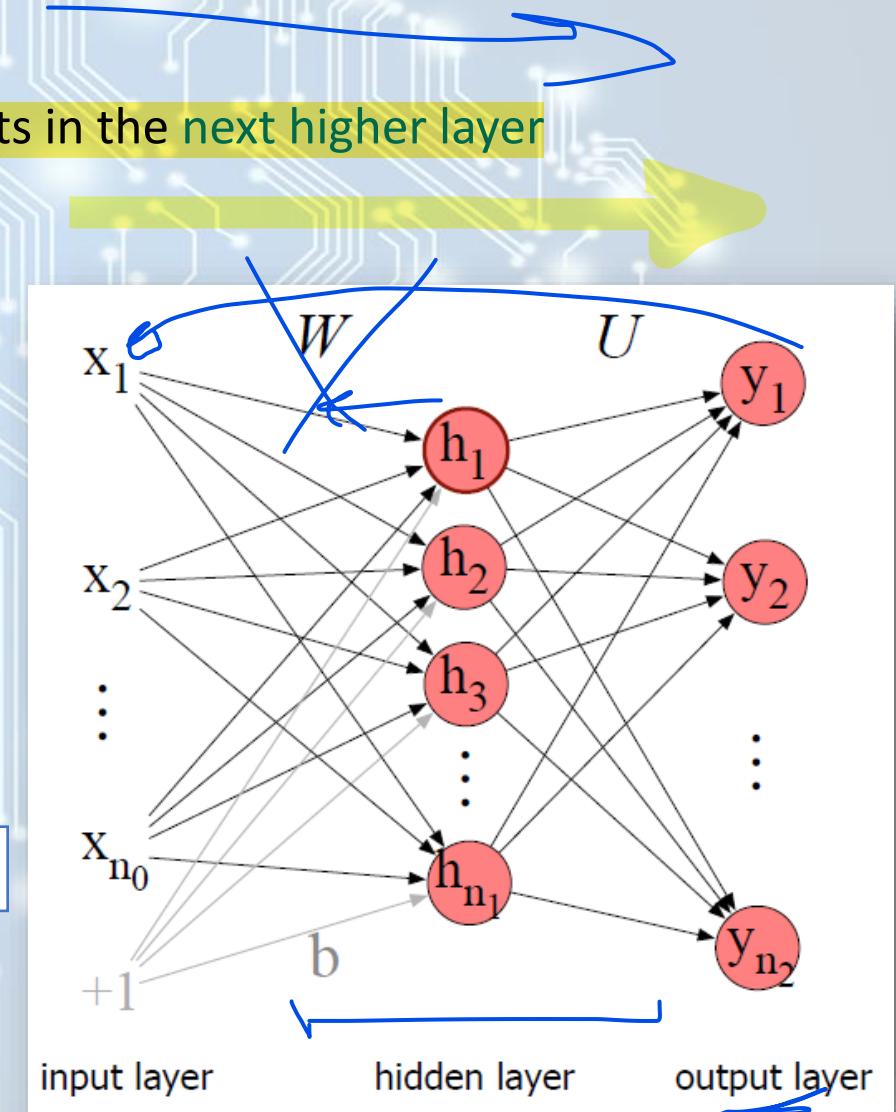
- A feedforward network:

- is a **multilayer** network
- the units are **connected** with **no cycles**
 - the outputs from units in each layer are **passed to** units in the **next higher layer**
 - **no** outputs are **passed back to** **lower layers**
- has three kinds of nodes:
 - **input** units, **hidden** units, and **output** units.

- A simple **2-layer feedforward network**:

- one hidden layer
- one output layer
- one input layer

*the input layer is usually **not** counted when enumerating layers*



Feedforward Neural Networks

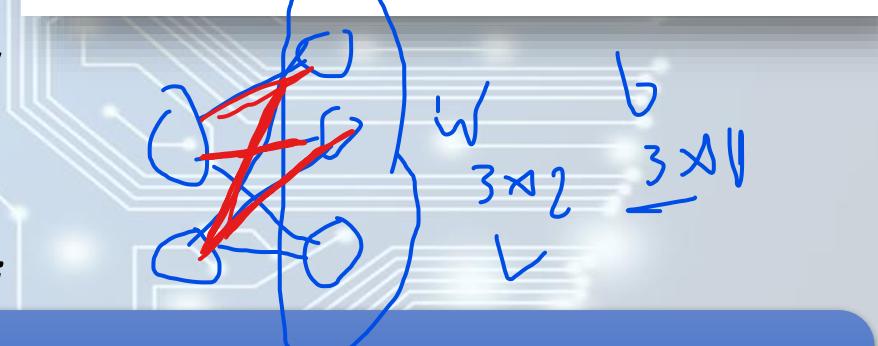
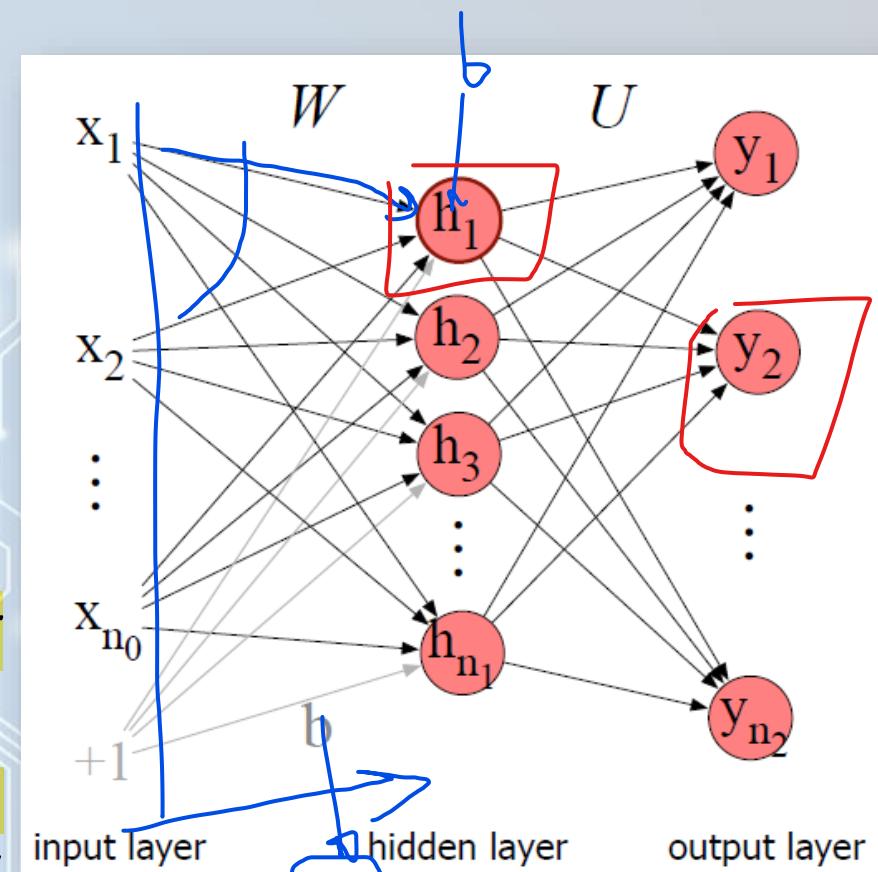
- In the standard architecture, each layer is **fully-connected**:
 - each unit in each layer takes as input the outputs from all the units in the previous layer.
 - there is a link between every pair of units from two adjacent layers.
- A single hidden unit has as parameters a **weight vector** and a **bias**.
 - We represent the parameters for the entire hidden layer as a **single weight matrix W** : with dimensionality $[n_1, n_0]$ and a **single bias vector \mathbf{b}** : with dimensionality $[n_1, 1]$
 - Each element W_{ji} of the weight matrix W represents the weight of the connection from the i th input unit x_i to the j th hidden unit h_j .

$$W_{3 \times 2} \quad \left(\begin{array}{cc} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{array} \right)$$

Why??

the hidden layer computation for a feedforward network can be done very efficiently with simple matrix operations.

$$\mathbf{h} = \sigma(\mathbf{Wx} + \mathbf{b})$$



Feedforward Neural Networks

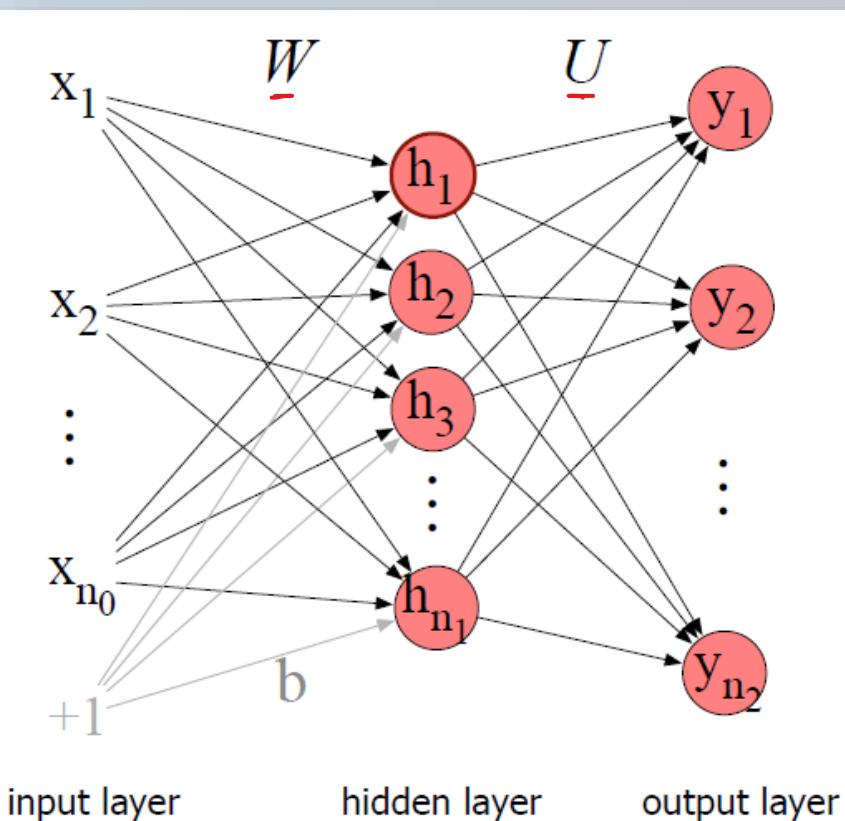
- Like the hidden layer, the output layer has **a weight matrix U** : with dimensionality $[n_2, n_1]$ but some models don't include a bias vector **b** in the output.
- The intermediate output z is given by:
$$z = Uh$$
- However, z can't be the output of a classifier, since it's a vector of real-valued numbers → while what we need for classification is **a vector of probabilities** → we need to perform **normalization**

- A popular function used is **softmax**

$$\text{softmax}(z_i) = \frac{\exp(z_i)}{\sum_{j=1}^d \exp(z_j)} \quad 1 \leq i \leq d$$

- Summing Up:

$$\begin{aligned} h &= \sigma(Wx + b) \\ z &= Uh \\ y &= \text{softmax}(z) \end{aligned}$$



$$z = [0.6, 1.1, -1.5, 1.2, 3.2, -1.1],$$

$$\text{softmax}(z) = [0.055, 0.090, 0.0067, 0.10, 0.74, 0.010]$$

$$\sum = 1$$

General Terminology

- Superscripts in square brackets to mean layer numbers, starting at 0 for the input layer.

- $\mathbf{W}^{[1]}$: weight matrix for the first hidden layer.

- $\mathbf{b}^{[1]}$: bias vector for the first hidden layer.

- n_j : number of units at layer j .

- $g(\cdot)$: activation function.

- (ReLU or tanh for intermediate layers and sigmoid or softmax for output layers).

- $\mathbf{a}^{[i]}$: output from layer i .

- $\mathbf{a}^{[0]}$ refers to the inputs \mathbf{x} .

- $\mathbf{z}^{[i]}$: combination of weights and biases

$$\mathbf{W}^{[i]} \mathbf{a}^{[i-1]} + \mathbf{b}^{[i]}$$



w_{ji} -> weight of link going to jth node in the target layer, coming from the ith node of the first layer

- For a 2-layer net:

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]} \mathbf{a}^{[0]} + \mathbf{b}^{[1]}$$

$$\mathbf{a}^{[1]} = g^{[1]}(\mathbf{z}^{[1]})$$

$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]} \mathbf{a}^{[1]} + \mathbf{b}^{[2]}$$

$$\mathbf{a}^{[2]} = g^{[2]}(\mathbf{z}^{[2]})$$

$$\hat{\mathbf{y}} = \mathbf{a}^{[2]}$$

- The algorithm for computing the forward step in an n-layer feedforward network:

```
for i in 1,...,n  
     $\mathbf{z}^{[i]} = \mathbf{W}^{[i]} \mathbf{a}^{[i-1]} + \mathbf{b}^{[i]}$   
     $\mathbf{a}^{[i]} = g^{[i]}(\mathbf{z}^{[i]})$   
 $\hat{\mathbf{y}} = \mathbf{a}^{[n]}$ 
```

Why non-linear activation functions?

- If we did not use them, the resulting network is exactly equivalent to a single-layer network.

- How??

- Consider the first two layers of such a network of purely linear layers:

$$z^{[1]} = \underline{W}^{[1]}x + b^{[1]}$$

$$z^{[2]} = W^{[2]}z^{[1]} + b^{[2]}$$

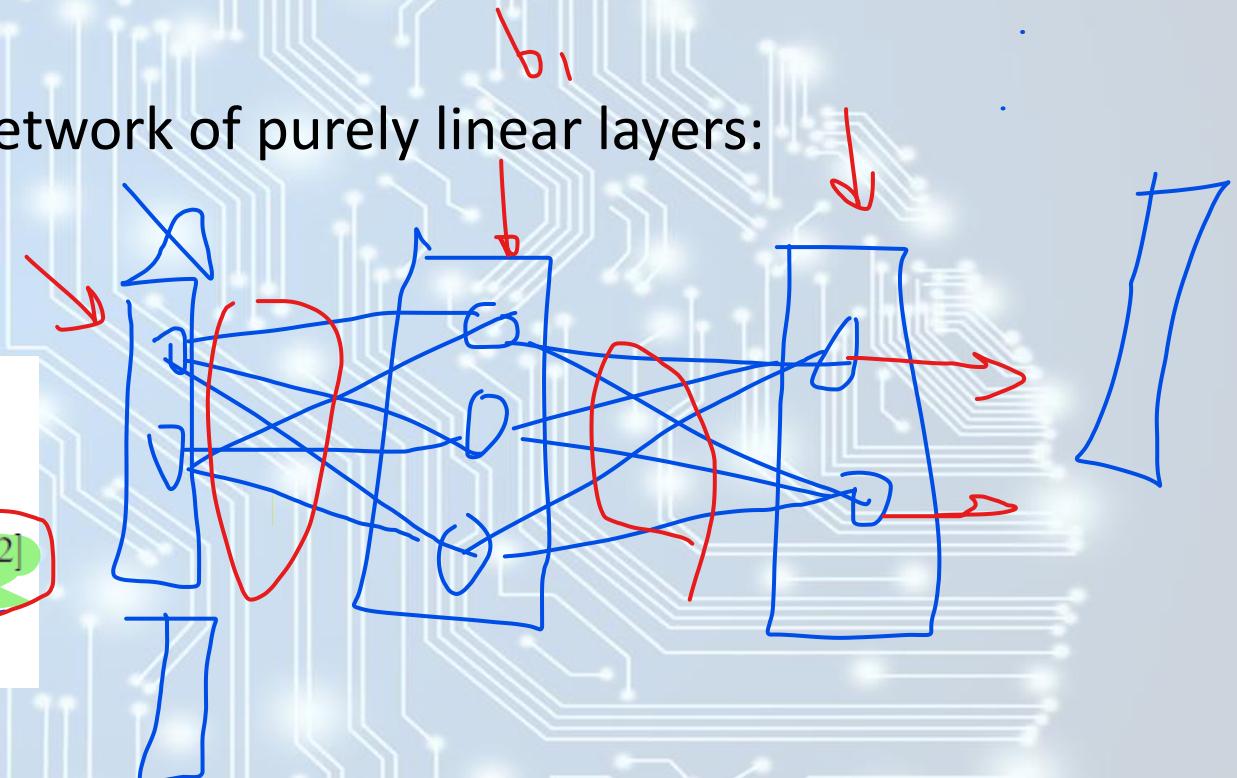
- Rewriting:

$$z^{[2]} = W^{[2]}z^{[1]} + b^{[2]}$$

$$= W^{[2]}(W^{[1]}x + b^{[1]}) + b^{[2]}$$

$$= \underbrace{W^{[2]}W^{[1]}x}_{W'x} + \underbrace{W^{[2]}b^{[1]} + b^{[2]}}_{b'}$$

$$= \underline{W'}x + b'$$

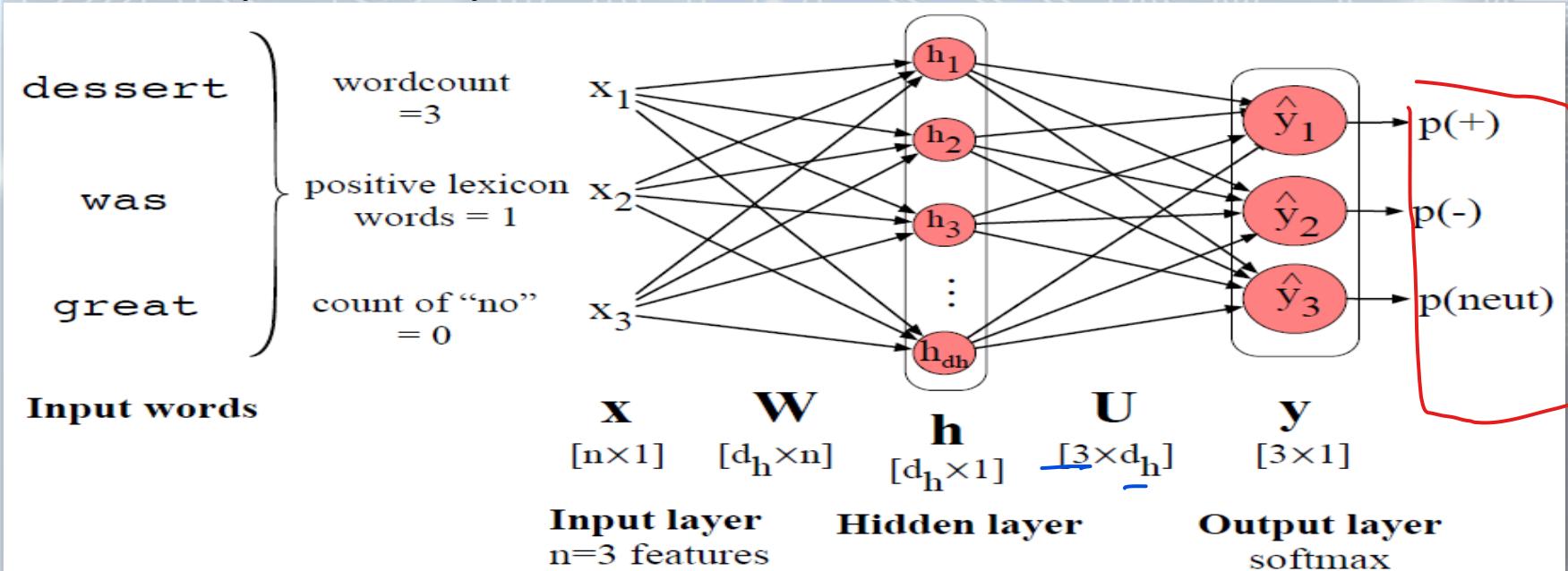


- This generalizes to any number of layers

without non-linear activation functions, a multilayer network is just a **single layer network with a different set of weights**, and we lose all the **representational power** of multilayer networks.

NLP Example

- A 2-layer sentiment classifier.
- The inputs are scalar features
 - e.g.: $x_1 = \text{count}(\text{words in doc})$, $x_2 = \text{count}(\text{positive lexicon words in doc})$, $x_3 = 1$ if “no” in doc, and so on.
- The output layer \hat{y} has 3 nodes (positive, negative, neutral):
 - \hat{y}_1 the estimated probability of positive sentiment
 - \hat{y}_2 the estimated probability of negative sentiment
 - \hat{y}_3 the estimated probability of neutral sentiment.



Training Neural Nets

- A feedforward neural net is an instance of **supervised machine learning** in which we know the **correct output y** for each **observation x** .
- The goal of the **training procedure** is to learn parameters $\mathbf{W}^{[i]}$ and $\mathbf{b}^{[i]}$ for each **layer i** that make \hat{y} (network output) for each **training observation** as close as possible to the **true y** .
- Components:
 1. **Loss function:** to model the distance between the network output and the gold output.
 2. **Gradient descent optimization:** to find the parameters that minimize this loss function.

$$w_{ji(\text{new})} = w_{ji(\text{old})} - \eta \frac{\partial L}{\partial w_{ji}}$$

where η is the learning rate

- 3. **Error backpropagation:** gradient descent requires knowing the gradient of the loss function, *the vector that contains the partial derivative of the loss function with respect to each of the parameters*.
 - For neural networks, with millions of parameters in many layers, it's much harder to see how to compute the partial derivative of some weight in layer 1 when the loss is attached to some much later layer → The answer is the algorithm called error backpropagation or backward differentiation.
 - The gradients are calculated starting from the final layer and then through use of the **chain rule**, the gradients can be passed backwards to calculate the gradients in the previous layers.
 - Chain rule: given $y=g(u)$, and $u=f(x)$

$$\frac{dy}{dx} = \frac{dy}{du} \times \frac{du}{dx}$$

Revise: Check computational graphs



Thank You

0100 0100
0110 0110
1001 1001 0100
0100 0110
0100 0011
0100 0001
1101

1010
0110
1001
0101
0100
1101
1101
0101
1110

0110
1001
0101
0110
0100
0011
1101
0101
1110
0100
0011
1101
1010

0110
1001