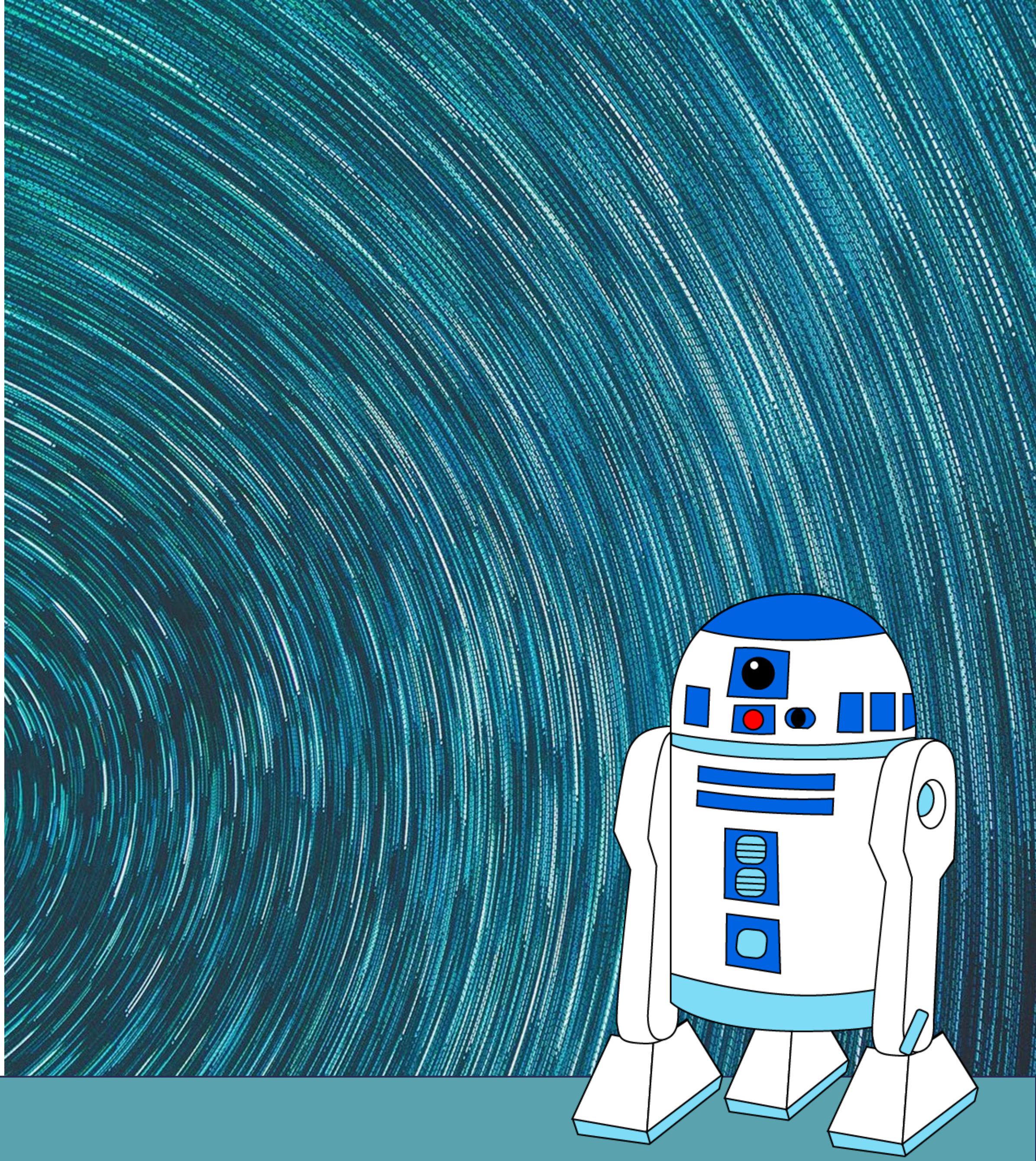
CIS 421/521:
ARTIFICIAL INTELLIGENCE

Neural Networks

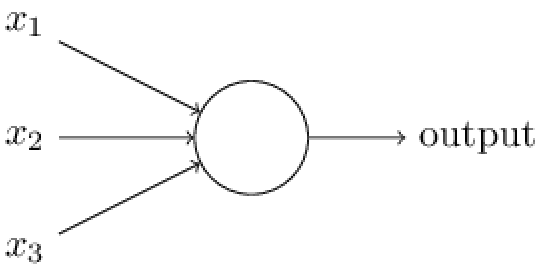
Jurafsky and Martin Chapter 7

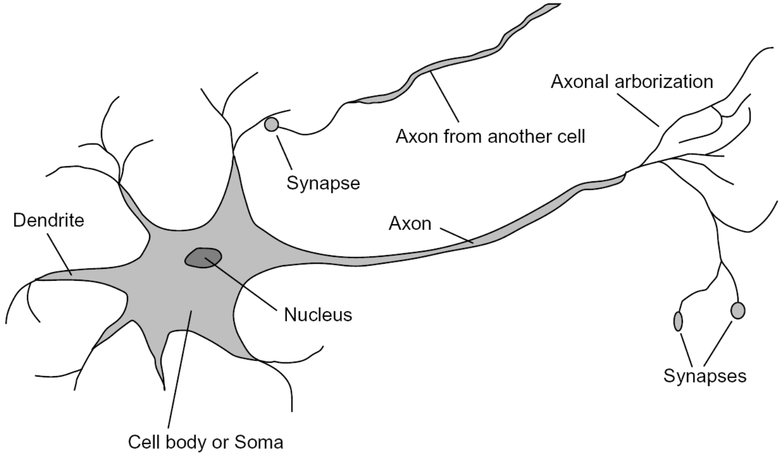


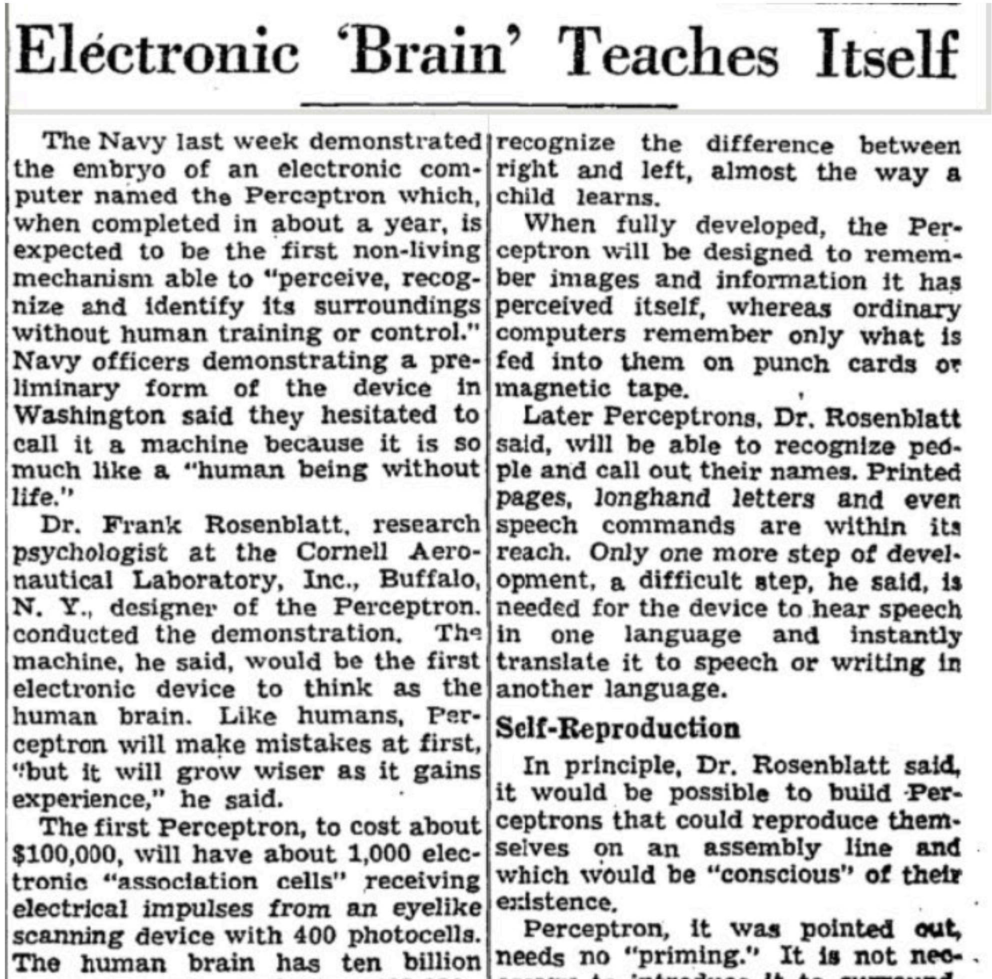
Penn Engineering
UNIVERSITY of PENNSYLVANIA

Review: Perceptron

- Perceptrons were developed in the 1950s and 1960s loosely inspired by the neuron.







Eléctronic 'Brain' Teaches Itself

The Navy last week demonstrated the embryo of an electronic computer named the Perceptron which, when completed in about a year, is expected to be the first non-living mechanism able to "perceive, recognize and identify its surroundings without human training or control." Navy officers demonstrating a preliminary form of the device in Washington said they hesitated to call it a machine because it is so much like a "human being without life."

Dr. Frank Rosenblatt, research psychologist at the Cornell Aeronautical Laboratory, Inc., Buffalo, N. Y., designer of the Perceptron, conducted the demonstration. The machine, he said, would be the first electronic device to think as the human brain. Like humans, Perceptron will make mistakes at first, "but it will grow wiser as it gains experience," he said.

The first Perceptron, to cost about $100,000, will have about 1,000 electronic "association cells" receiving electrical impulses from an eyelike scanning device with 400 photocells. The human brain has ten billion recognize the difference between right and left, almost the way a child learns.

When fully developed, the Perceptron will be designed to remember images and information it has perceived itself, whereas ordinary computers remember only what is fed into them on punch cards or magnetic tape.

Later Perceptrons, Dr. Rosenblatt said, will be able to recognize people and call out their names. Printed pages, longhand letters and even speech commands are within its reach. Only one more step of development, a difficult step, he said, is needed for the device to hear speech in one language and instantly translate it to speech or writing in another language.

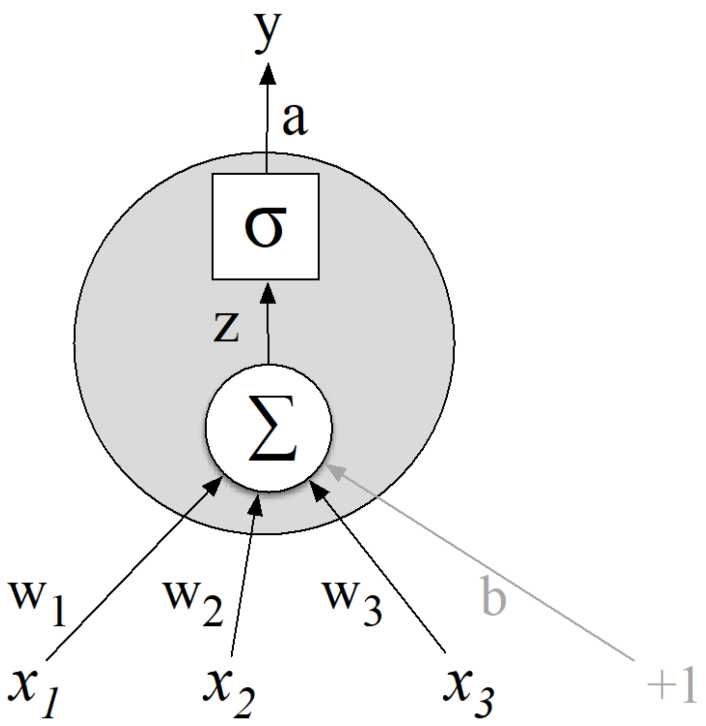
Self-Reproduction

In principle, Dr. Rosenblatt said, it would be possible to build Perceptrons that could reproduce themselves on an assembly line and which would be "conscious" of their existence.

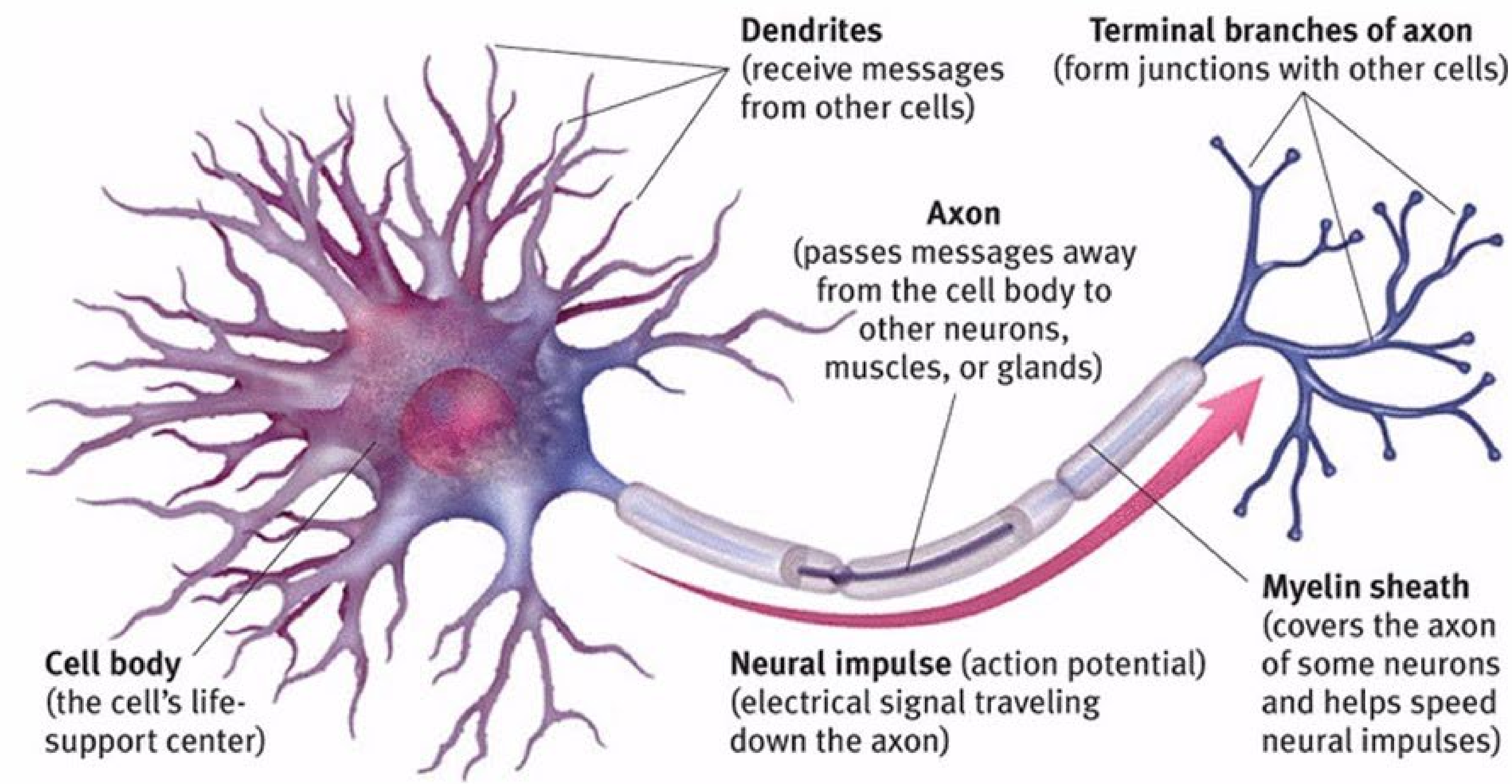
Perceptron, it was pointed out, needs no "priming." It is not nec-

Neural Networks

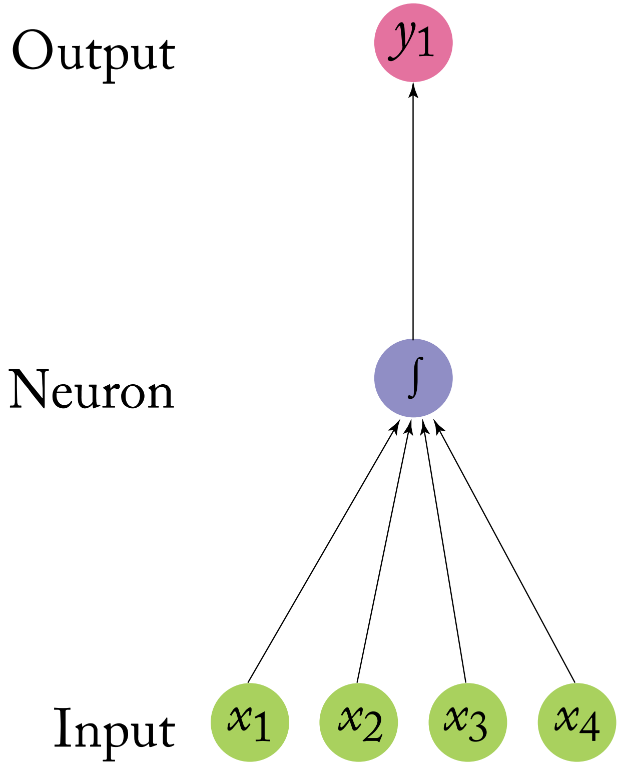
- The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation.

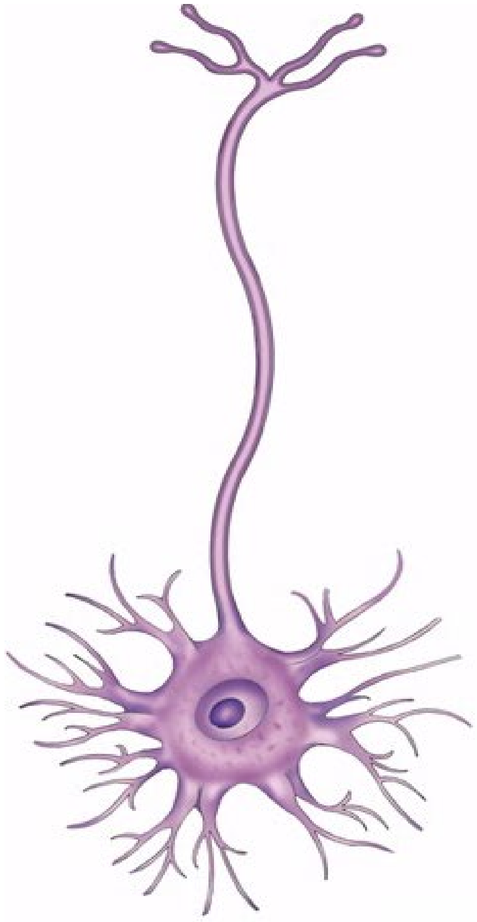


Neural Networks: A brain-inspired metaphor

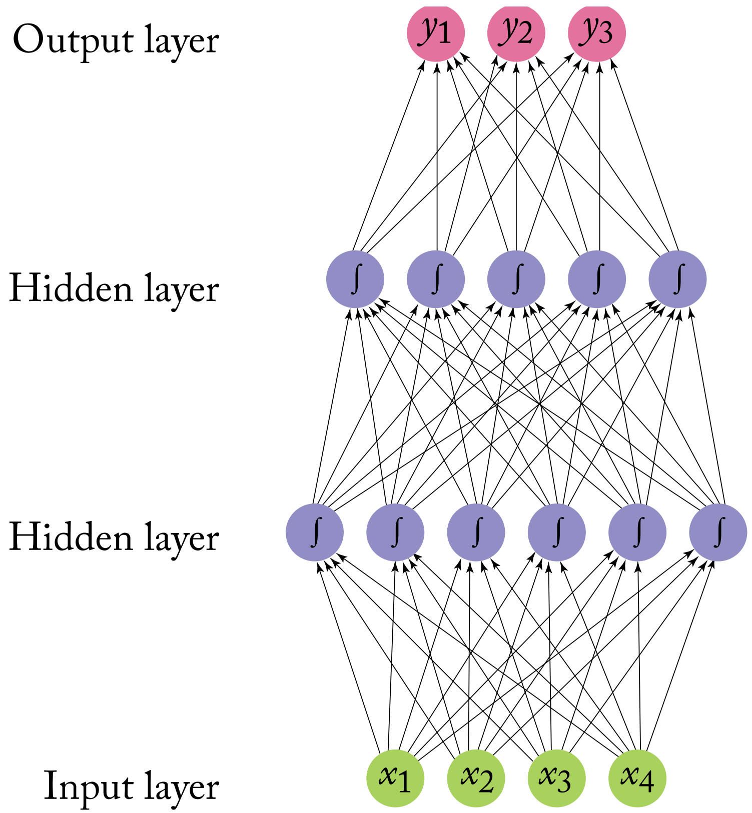


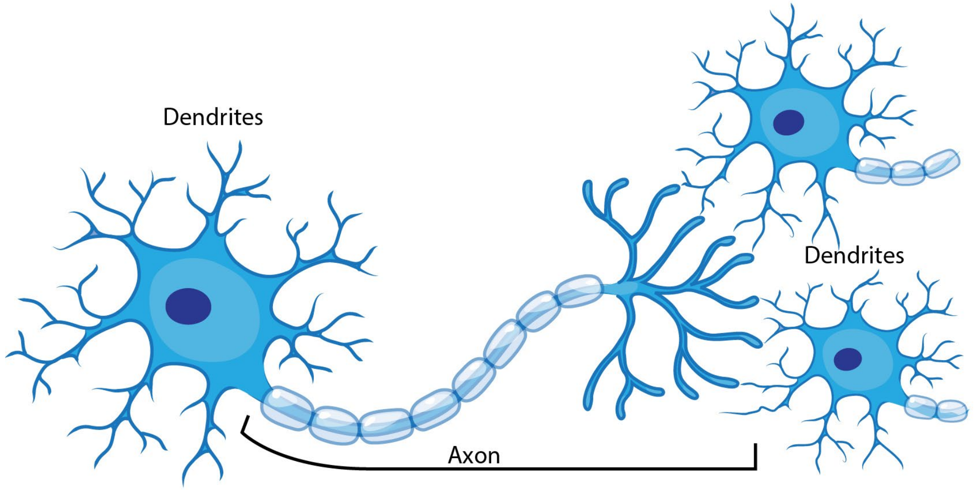
A single neuron





Neural networks



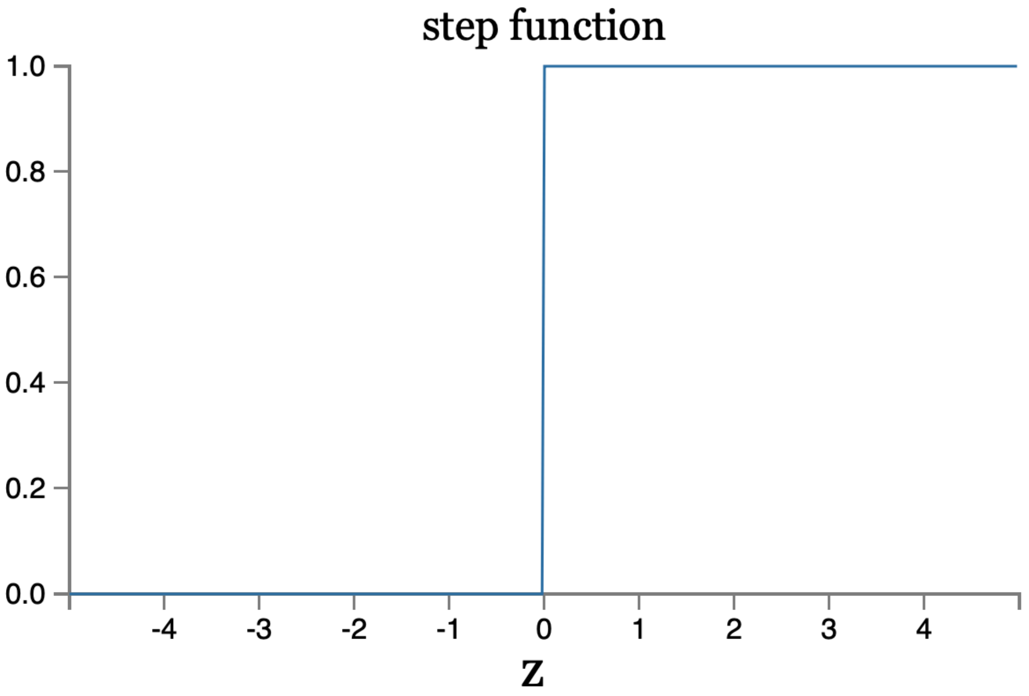


Perceptron -> Logistic Regression

- Like the Perceptron, logistic regression uses a vector of **weights** and a **bias term**.
- $z = \sum_i w_i x_i + b$
- This can also be written as a dot product:
- $z = w \cdot x + b$
- Instead of outputting z directly, logistic regression transforms it with the sigmoid function σ(z).

Perceptron

$$z \equiv w \cdot x + b$$



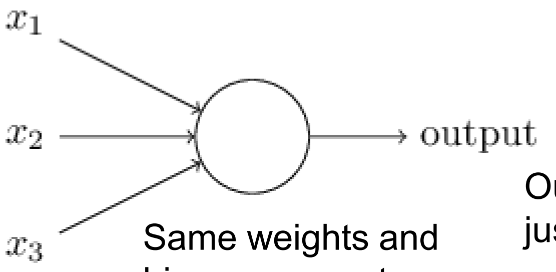
$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Sigmoid neurons

- Problem: a small change in the weights or bias of any single perceptron in the network can causes the output to completely flip from 0 to 1.
- Solution: sigmoid neuron

Perceptron

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$



Same weights and bias as perceptron

Output is no longer just 1 or 0.

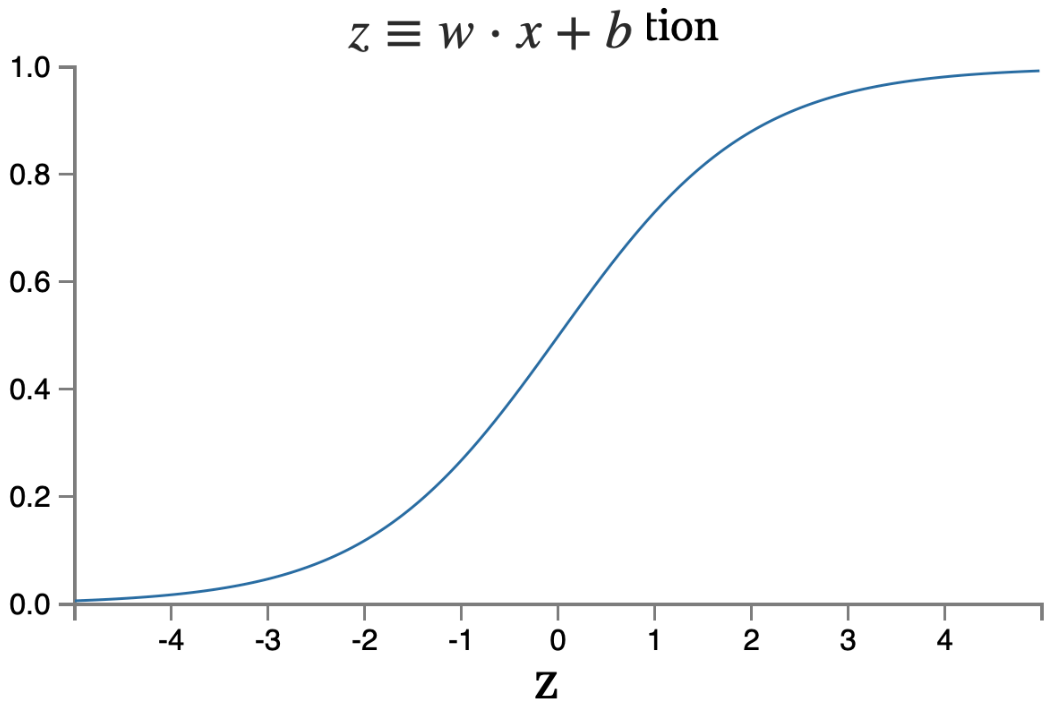
Inputs: any real-valued number

Sigmoid neuron output = $\sigma(w \cdot x + b)$

Sigmoid function:

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

Sigmoid neuron



$z \equiv w \cdot x + b$ is a large positive number. Then $e^{-z} \approx 0$

$z = w \cdot x + b$ is very negative. Then $e^{-z} \rightarrow \infty$, and $\sigma(z) \approx 0$

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

Smoothness is crucial

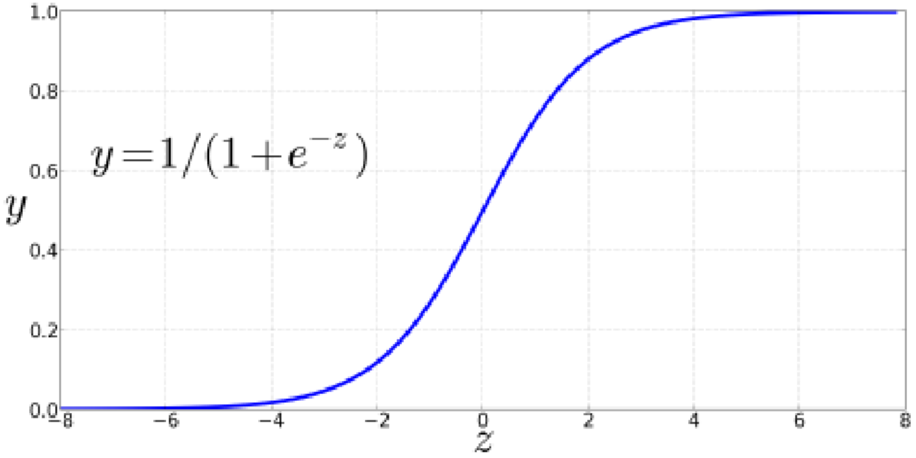
- Smoothness of σ means that small changes in the weights w_j and in the bias b will produce a small change the output from the neuron

$$\Delta\text{output} \approx \sum_j \frac{\partial\, \text{output}}{\partial w_j}\Delta w_j + \frac{\partial\, \text{output}}{\partial b}\Delta b$$

- Δoutput is a *linear function* of the changes Δwj and Δb
- This makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output

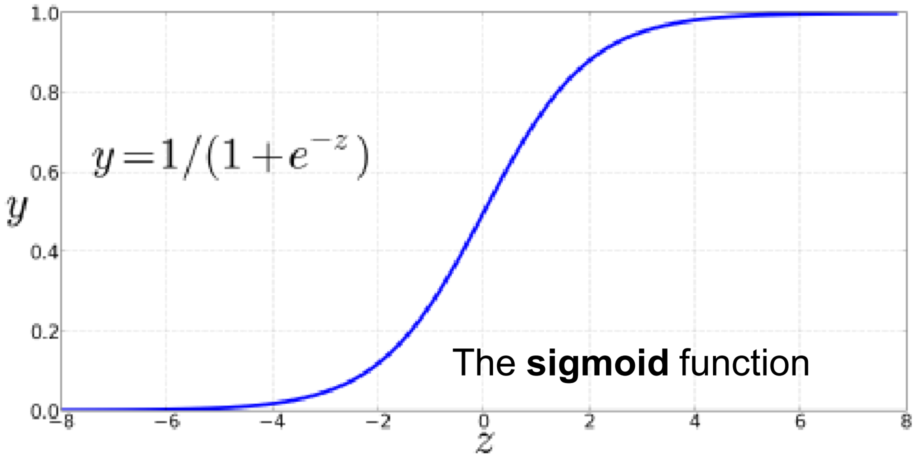
Activation Functions

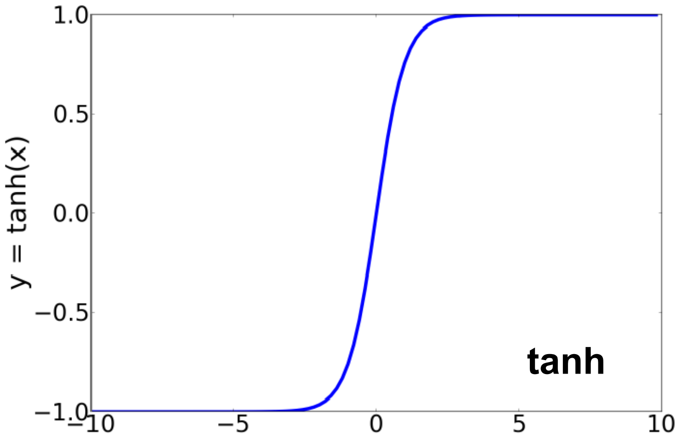
- Instead of directly outputting $z = w \cdot x + b$, which is a linear function of x, neuron units apply a non-linear function f to z.
- The output of this function is called the **activation value** for the unit, represented by the variable **a**. The output of a neural network is called **y,** so if the activation of a node is the final output of a network then
  - $y = a = f(z)$
- There are 3 commonly used non-linear functions used for f:
  The **sigmoid** function
  The **tanh** function
  The **rectified linear unit ReLU**

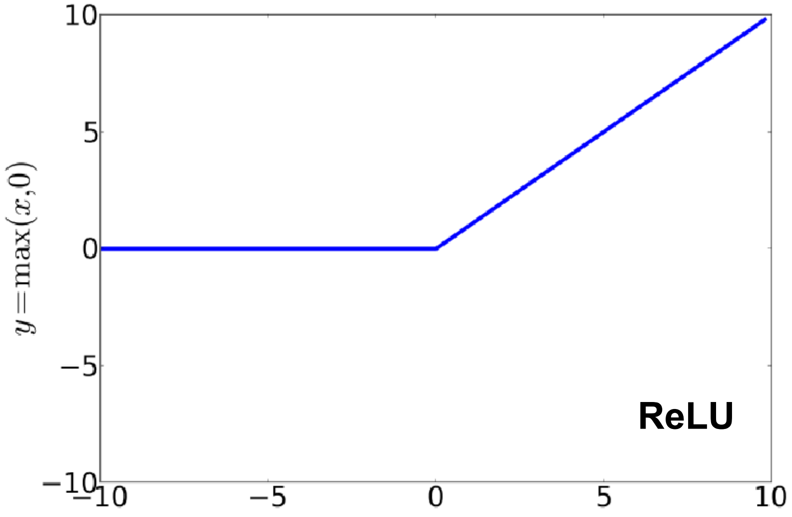


The **sigmoid** function

Activation Functions

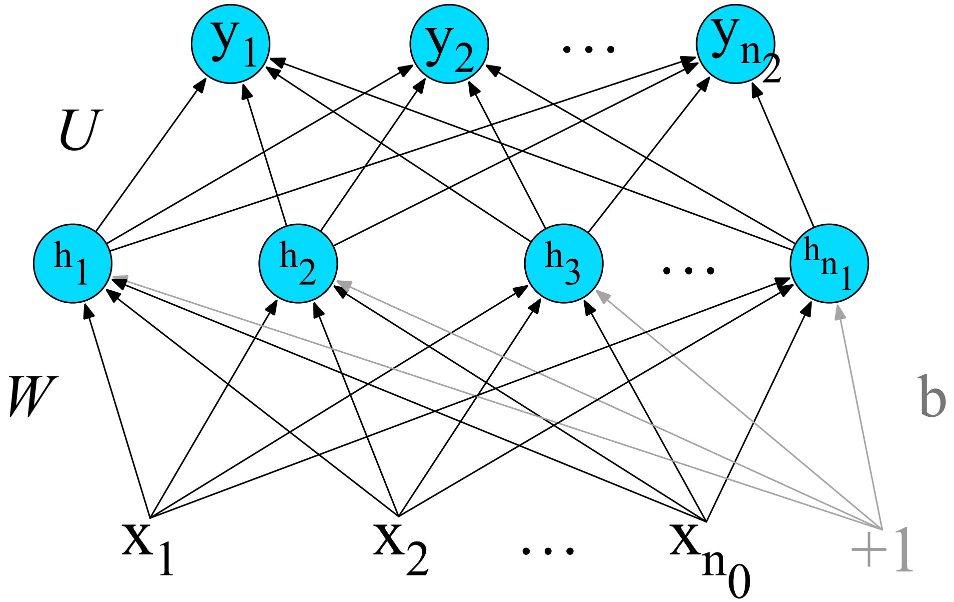






Feed-Forward Neural Network

- The simplest kind of NN is the **Feed-Forward Neural Network**
- **Multilayer** network, all units are usually **fully-connected**, and **no cycles**.
- The outputs from each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.



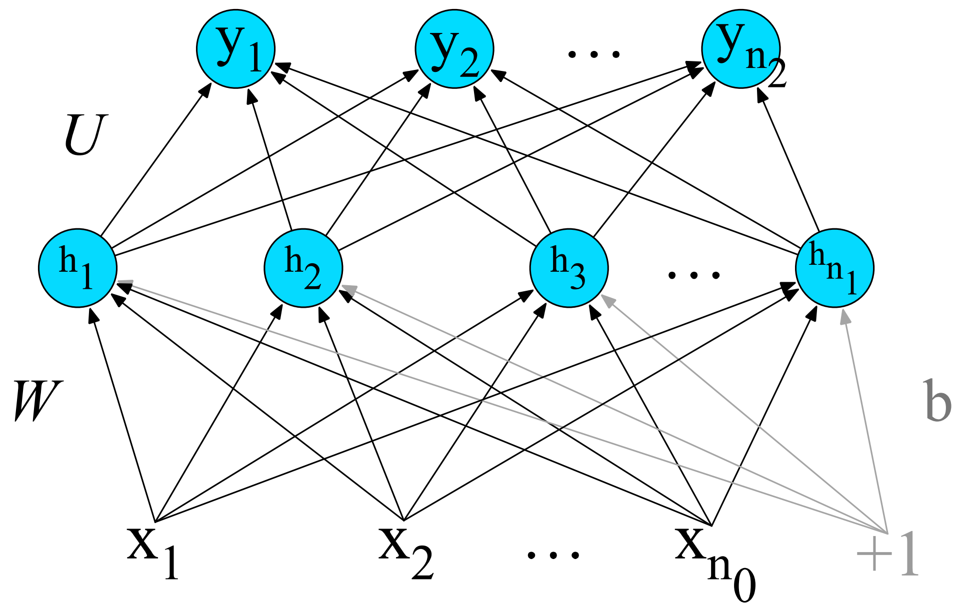
Layer 2 (output layer)

Layer 1 (hidden layer)

Layer 0 (input layer)

Equations for a feedforward network

- A single hidden unit has parameters ***w*** (the weight vector) and ***b*** (the bias scalar).
- We represent the parameters for the **entire hidden layer** by combining the weight vector $\mathbf{w}_i$ and bias $\boldsymbol{b}_i$ for each unit ***i*** into a single weight matrix ***W*** and a single bias vector **b** for the whole layer.



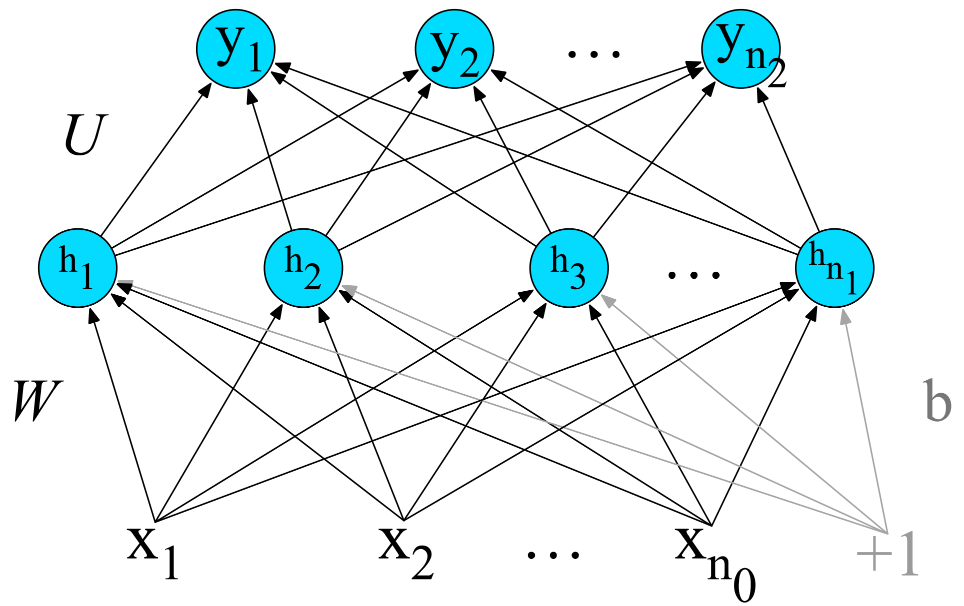
Equations for a feedforward network

- The advantage of using a single matrix ***W*** for the weights of the entire layer is the hidden layer computation can be done efficiently with simple matrix operations.
- The computation has three steps:
- 1. multiplying the weight matrix by the input vector x,
  2. adding the bias vector b, and
  3. applying the activation function g (such as Sigmoid)
- The output of the hidden layer, the vector h, is thus the following, using the sigmoid function σ:

$$h = \sigma(Wx+b)$$

Equations for a feedforward network

- Like the hidden layer, the output layer has a weight matrix ***U***.
  Its weight matrix is multiplied by its input vector (***h***) to produce the intermediate output ***z***.
  - $z = Uh$



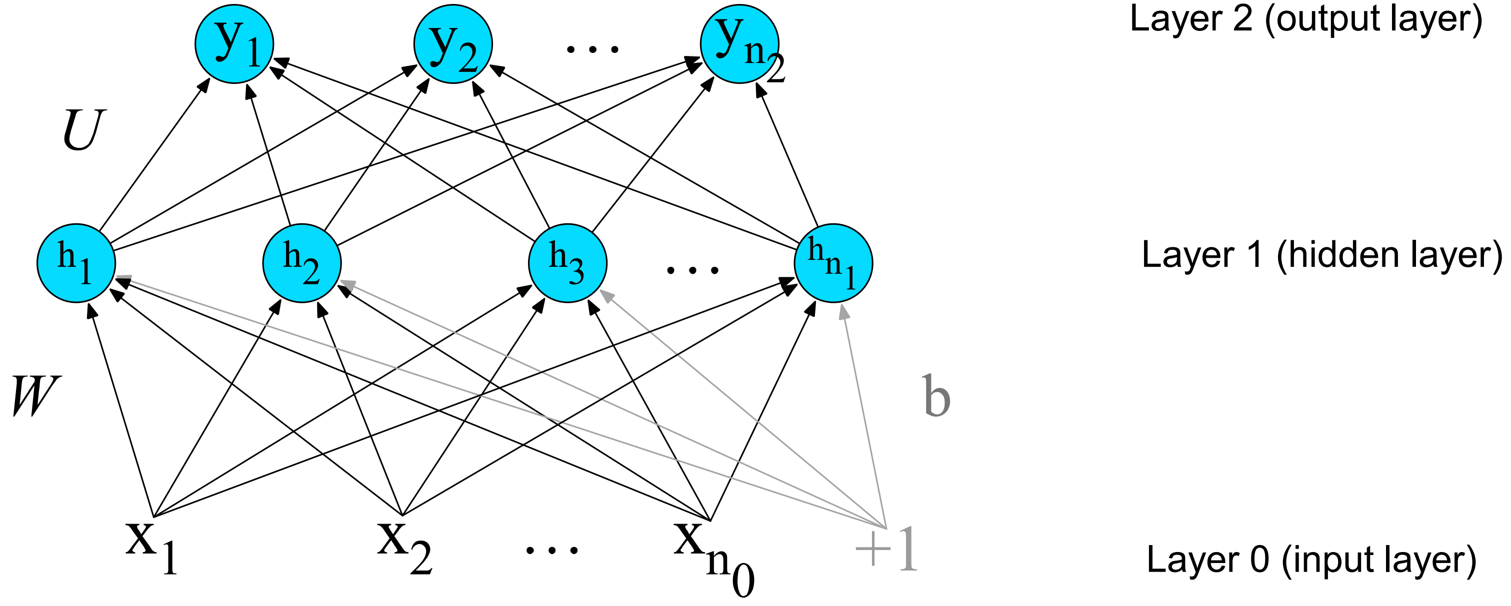
Equations for a feedforward network

- Here are the final equations for a feedforward network with a single hidden layer, which takes an input vector x, outputs a probability distribution y, and is parameterized by weight matrices W and U and a bias vector b:
- $h = \sigma(Wx+b)$
  $z = Uh$
  $y = \text{softmax}(z)$

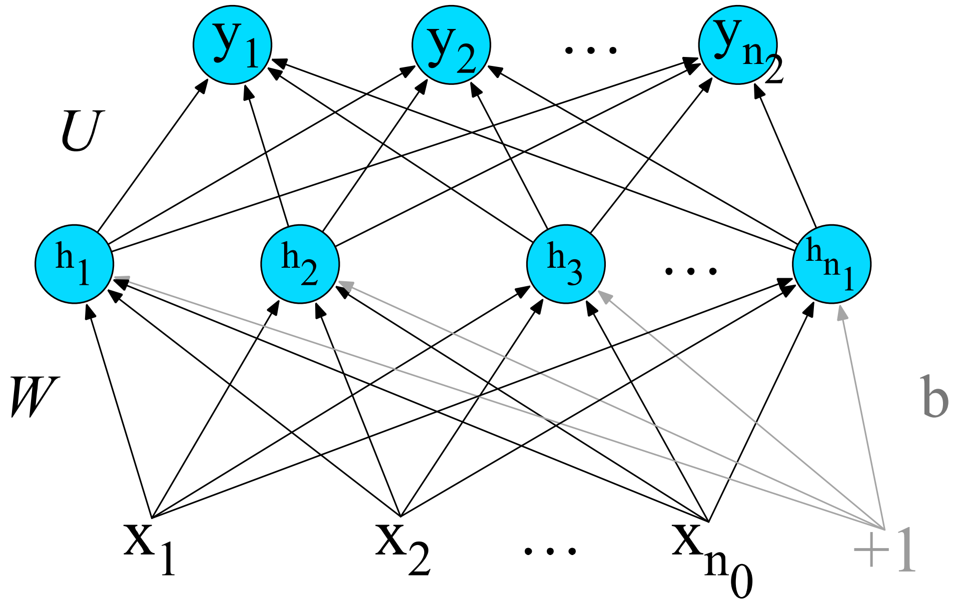
- Like with logistic regression, softmax normalizes the output and turns it into a probability distribution.

Review: Feed-Forward Neural Network

- The simplest kind of is the **Feed-Forward Neural Network**
- **Multilayer** network, all units are usually **fully-connected**, and **no cycles**.
- The outputs from each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers.



Review: Feed-Forward Neural Network

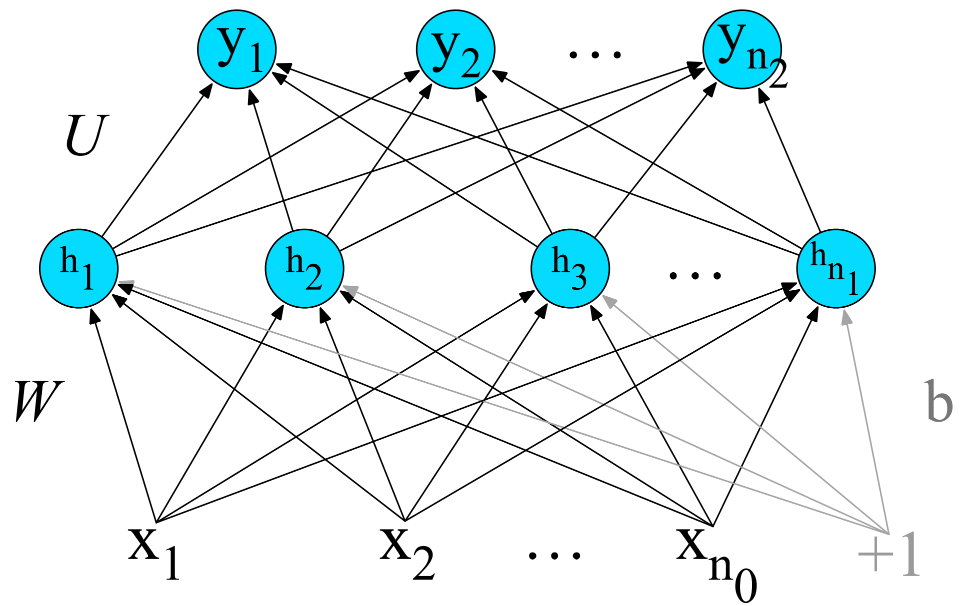


- A single hidden unit has parameters ***w*** (the weight vector) and ***b*** (the bias scalar).
- We represent the parameters for the **entire hidden layer** by combining the weight vector $\mathbf{w}_i$ and bias $\boldsymbol{b}_i$ for each unit ***i*** into a single weight matrix ***W*** and a single bias vector **b** for the whole layer.

Review: Feed-Forward Neural Network

- The advantage of using a single matrix ***W*** for the weights of the entire layer is the hidden layer computation can be done efficiently with simple matrix operations.
- The computation has three steps:
- 1. multiplying the weight matrix by the input vector x,
  2. adding the bias vector b, and
  3. applying the activation function g (such as Sigmoid)
- The output of the hidden layer, the vector h, is thus the following, using the sigmoid function σ:
  - $h = \sigma(Wx+b)$

Review: Feed-Forward Neural Network



- Like the hidden layer, the output layer has a weight matrix ***U***.
  Its weight matrix is multiplied by its input vector (***h***) to produce the intermediate output ***z***.
  - *z=Uh*

Review: Feed-Forward Neural Network

- Here are the final equations for a feedforward network with a single hidden layer, which takes an input vector x, outputs a probability distribution y, and is parameterized by weight matrices W and U and a bias vector b:
- $h = \sigma(Wx+b)$
  $z = Uh$
  $y = \text{softmax}(z)$
- Like with logistic regression, softmax normalizes the output and turns it into a probability distribution.

Training Neural Nets

- Like logistic regression, we want to learn the best parameters for the neural net to make its predictions $\hat{y}$ as close to possible as the gold standard labels in our training data *y*.
- What do we need?
- **A loss function** – cross-entropy loss
- **An optimization algorithm** – gradient descent
- **A way of computing the gradient of the loss function** – error propagation

Cross-Entropy Loss

- If the neural network is a binary classifier with a sigmoid at the final layer, the loss function is exactly the same as we saw in logistic regression:

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

Cross-Entropy Loss

- If the neural network is a binary classifier with a sigmoid at the final layer, the loss function is exactly the same as we saw in logistic regression:

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

- For multinomial classification
  - $L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$

- If there is only one correct answer, where the truth is $y_i=1$, then this simplifies to be
- $L_{CE}(\hat{y}, y) = -\log \hat{y}_i$

- Plugging into softmax:
- $L_{CE}(\hat{y}, y) = -\log \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$

Computing the gradient

- Logistic regression can be thought of as a network with just one weight layer and a sigmoid output. In that case the gradient is:

- $\frac{\partial LCE(w,b)}{\partial wj} = (\hat{y} - y)\, xj$
- $= (\sigma(w \cdot x + b) - y)x_j$

- But these derivatives only give correct updates for the last weight layer! For deeper networks, computing the gradients requires looking back through all the earlier layers in the network, even though the loss is only computed with respect to the output of the network.

Solution: **error backpropagation algorithm**

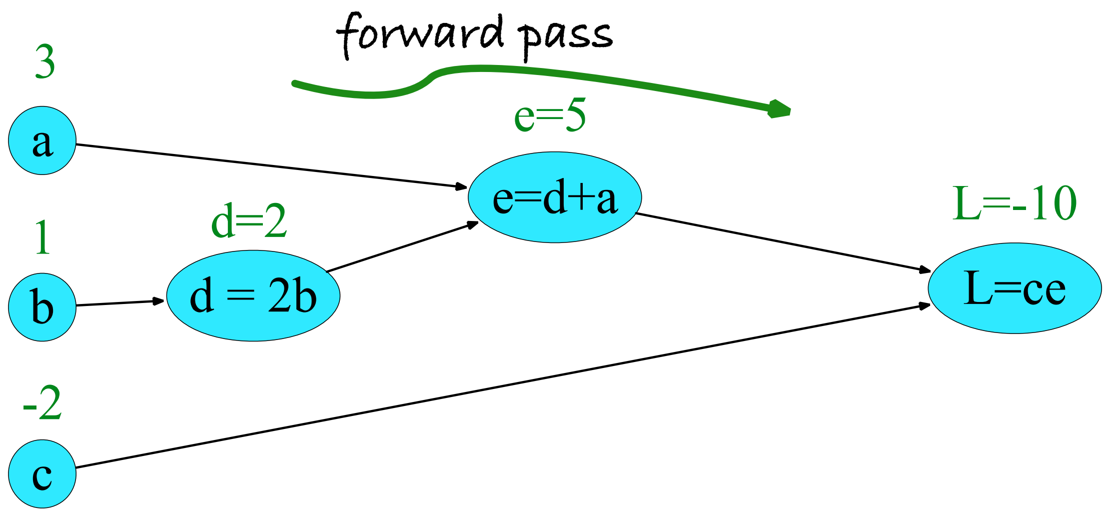
Computation Graphs

- Although backpropagation was invented for neural nets, it is related to general procedure called **backward differentiation**, which depends on the notion of **computation graphs**.
- A computation graph represents the process of computing a mathematical expression. The computation is broken down into separate operations. Each operation is a node in a graph.

$L(a, b, c) = c(a + 2b)$

$d = 2*b$
$e = a+d$
$L = c*e$

Forward pass



$L(a, b, c) = c(a + 2b)$

$d = 2*b$
$e = a+d$
$L = c*e$

inputs $a = 3$, $b = 1$, $c = -2$,

Backward differentiation

- The importance of the computation graph comes from the backward pass, which is used to compute the derivatives that we'll need for the weight update.
- How do we compute the derivative of our output function L with respect to the input variables a, b, and c?

$$\frac{\partial L}{\partial a}, \frac{\partial L}{\partial b}, \text{ and } \frac{\partial L}{\partial c}$$

- Backwards differentiation uses the **chain rule** from calculus.

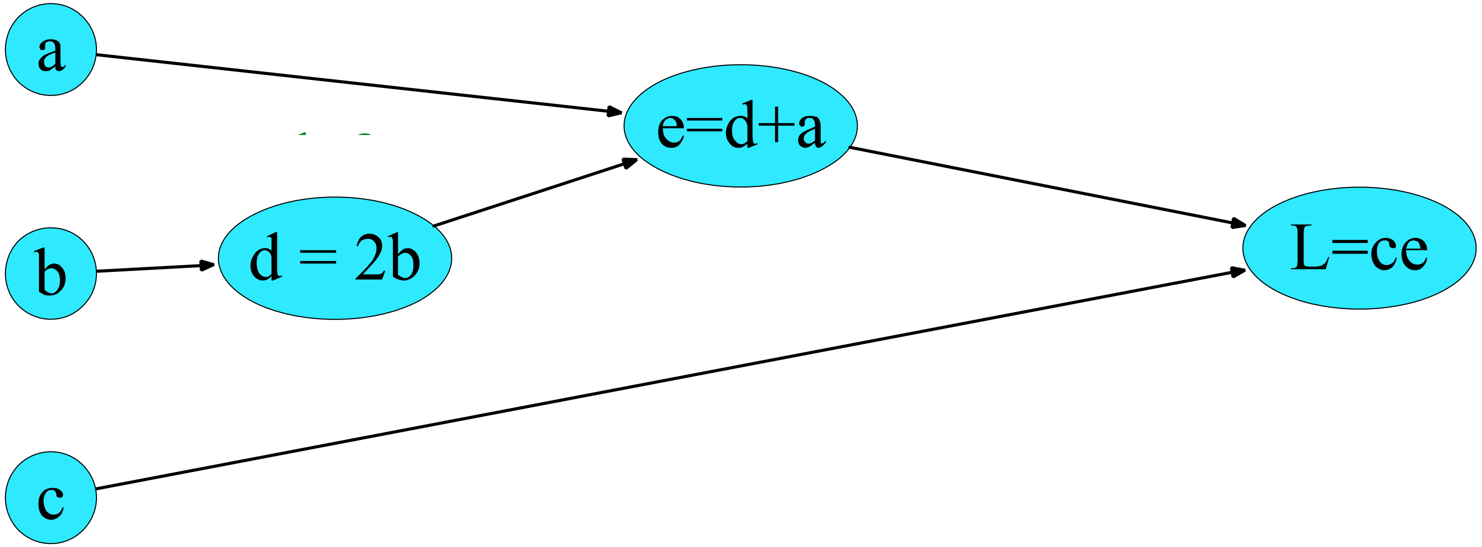
Chain rule

- For a composite function $f(x) = u(v(x))$, the derivative of $f(x)$ is:

$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dx}$$

- Similarly for, $f(x) = u(v(w(x)))$, the derivative of $f(x)$ is:

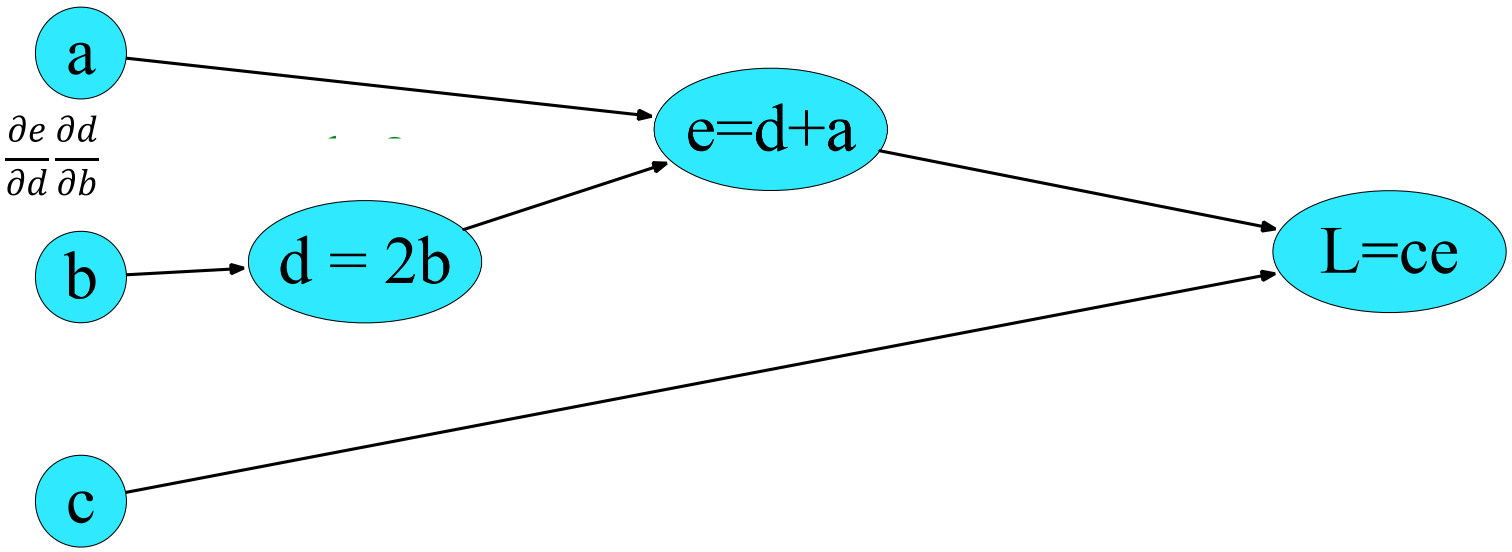
$$\frac{df}{dx} = \frac{du}{dv} \cdot \frac{dv}{dw} \cdot \frac{dw}{dx}$$



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

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

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

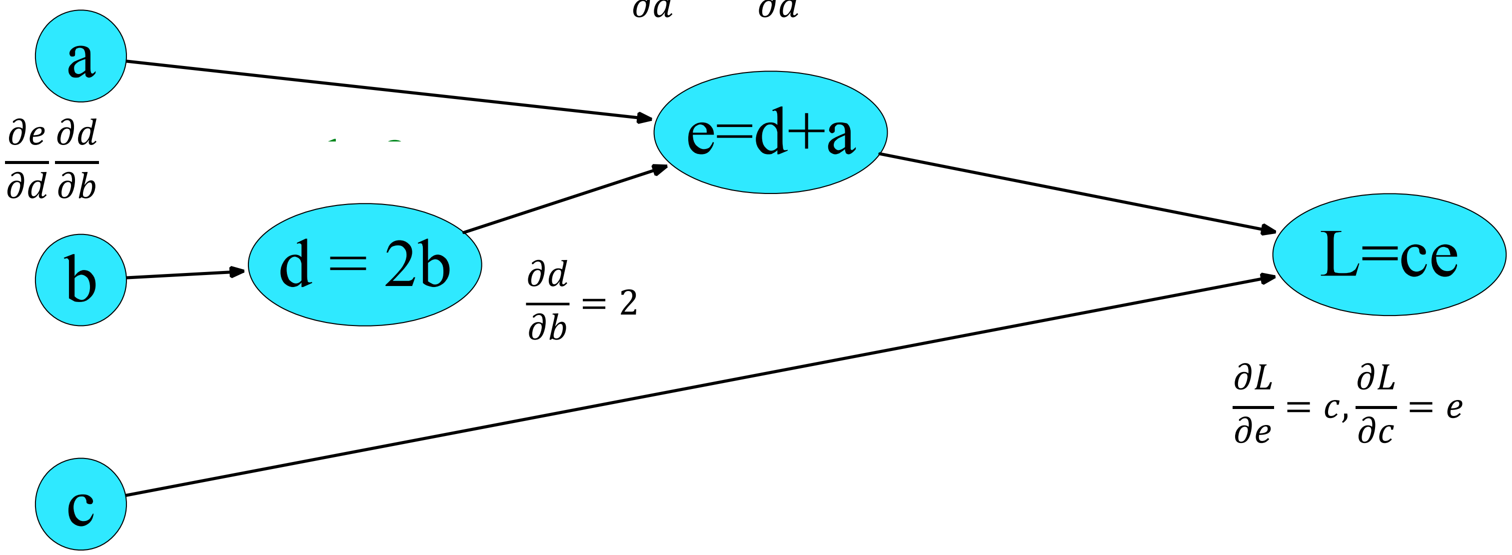


$$\frac{\partial L}{\partial c} = e$$

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

$$\frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$$

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

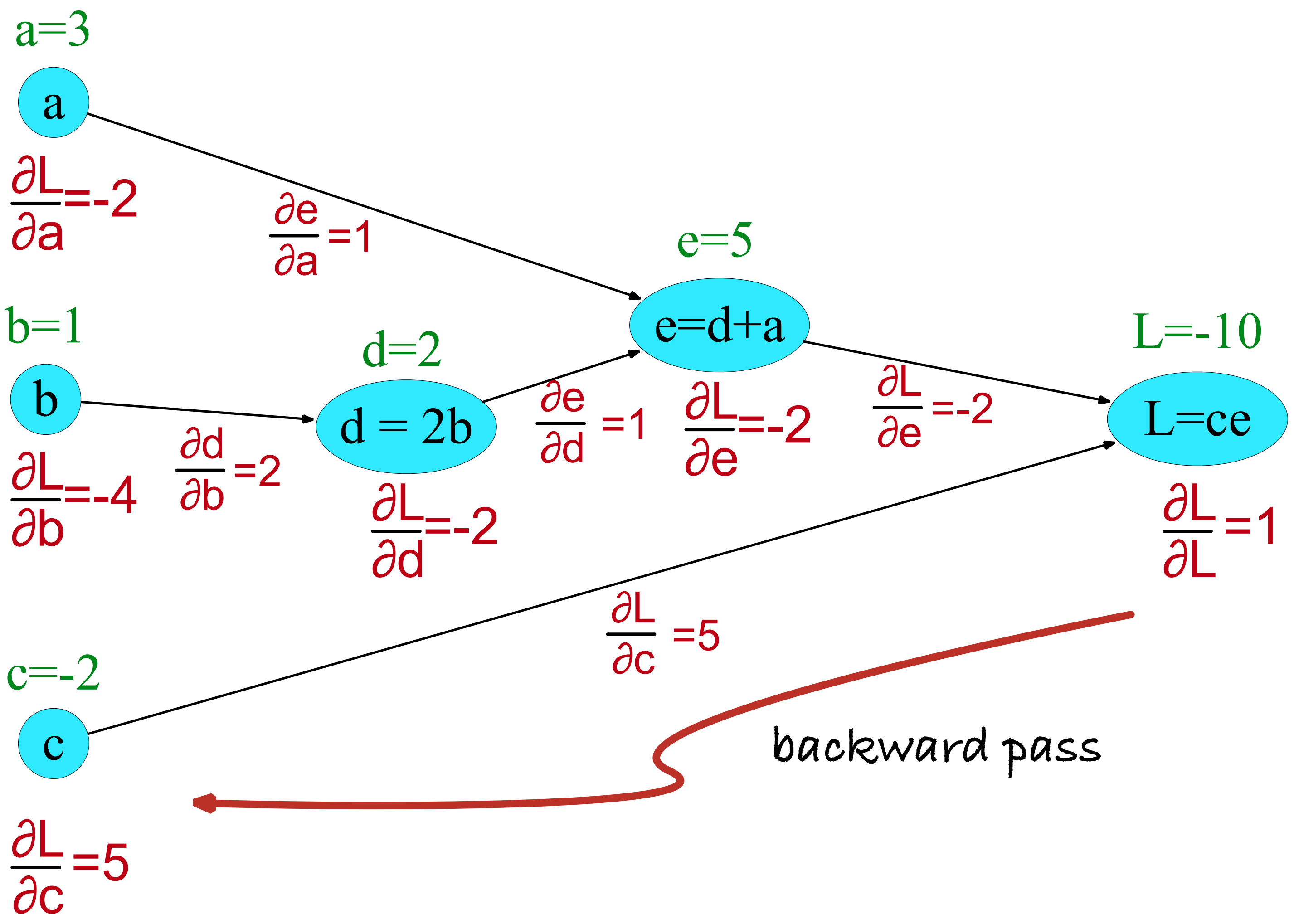


$$\frac{\partial d}{\partial b} = 2$$

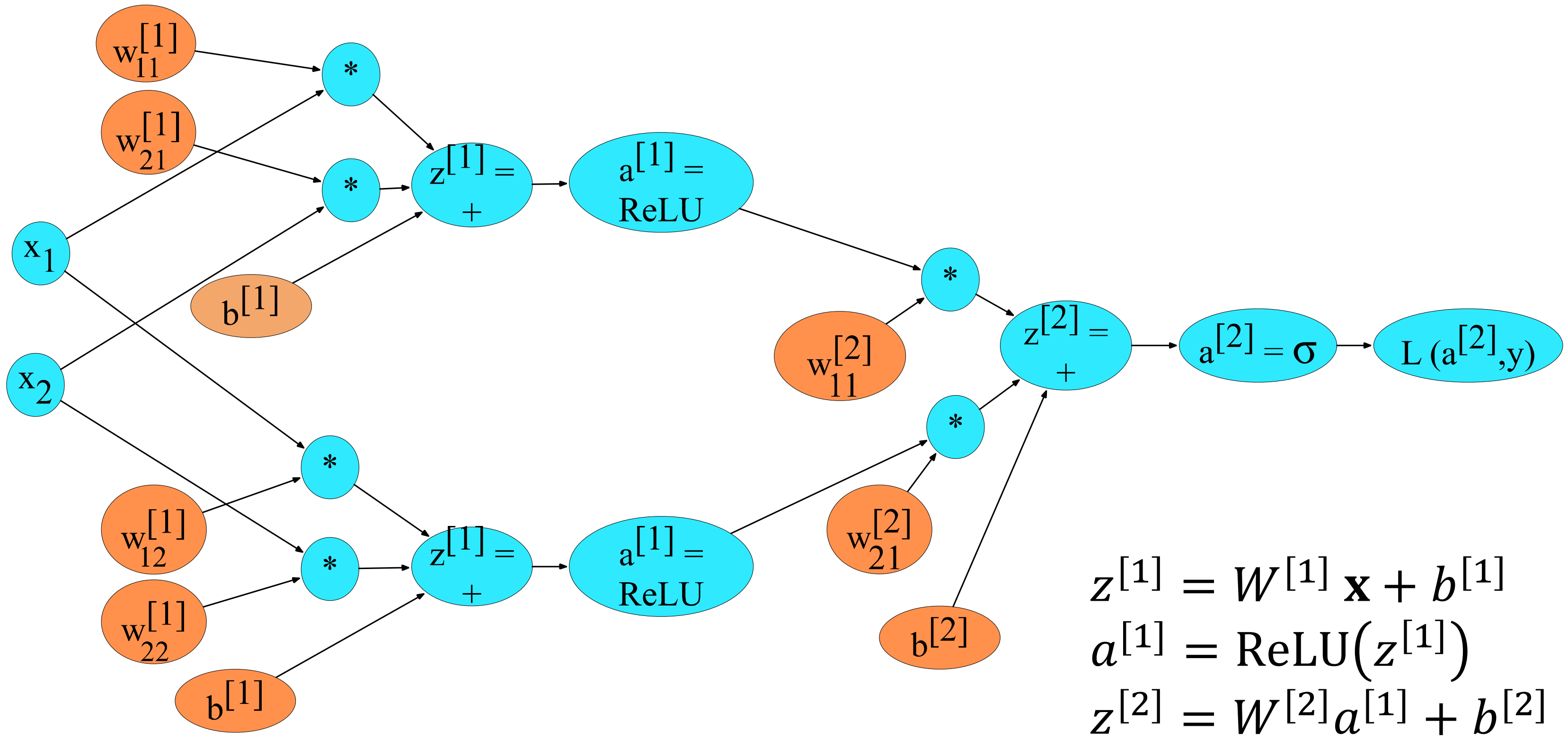
$$\frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

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

Backward pass



Computation Graph for a NN



$$z^{[1]} = W^{[1]}\,\mathbf{x} + b^{[1]}$$
$$a^{[1]} = \mathrm{ReLU}\left(z^{[1]}\right)$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]}$$
$$a^{[2]} = \sigma\left(z^{[2]}\right)$$
$$\hat{y} = a^{[2]}$$