# Lecture 4:
# Briefly on Neural Networks and Basic Neural Language Models

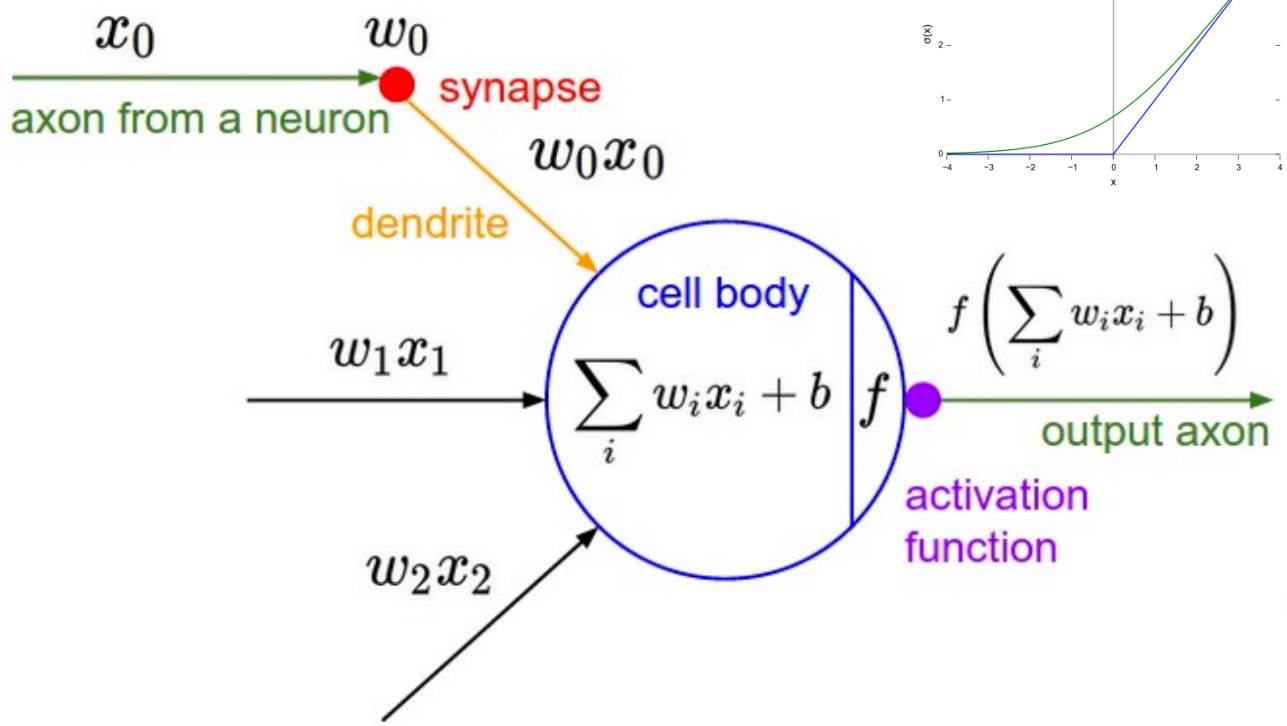
This presentation was built in assistance with Stanford's NLP group course material.

# Briefly on Neural Networks

- Neural network algorithms date from the 70's
- Originally inspired by early neuroscience
- Historically slow, complex, and unwieldy
- Now: term is abstract enough to encompass a wide variety of models
- Dramatic shift in NLP (since ~2015) away from log-linear models (linear, convex) to "neural net" (non-linear, non-convex architecture)

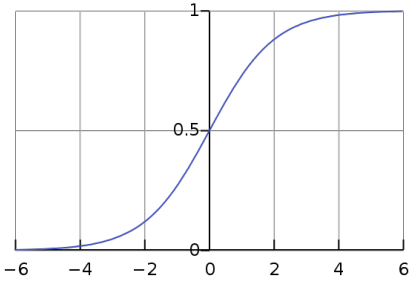
# Nodes in Neural Networks

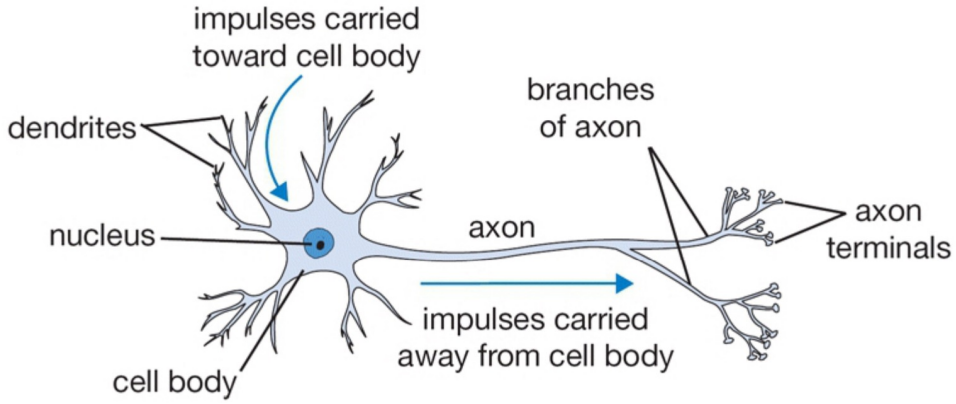
Parameters: $w_i$ and $b$







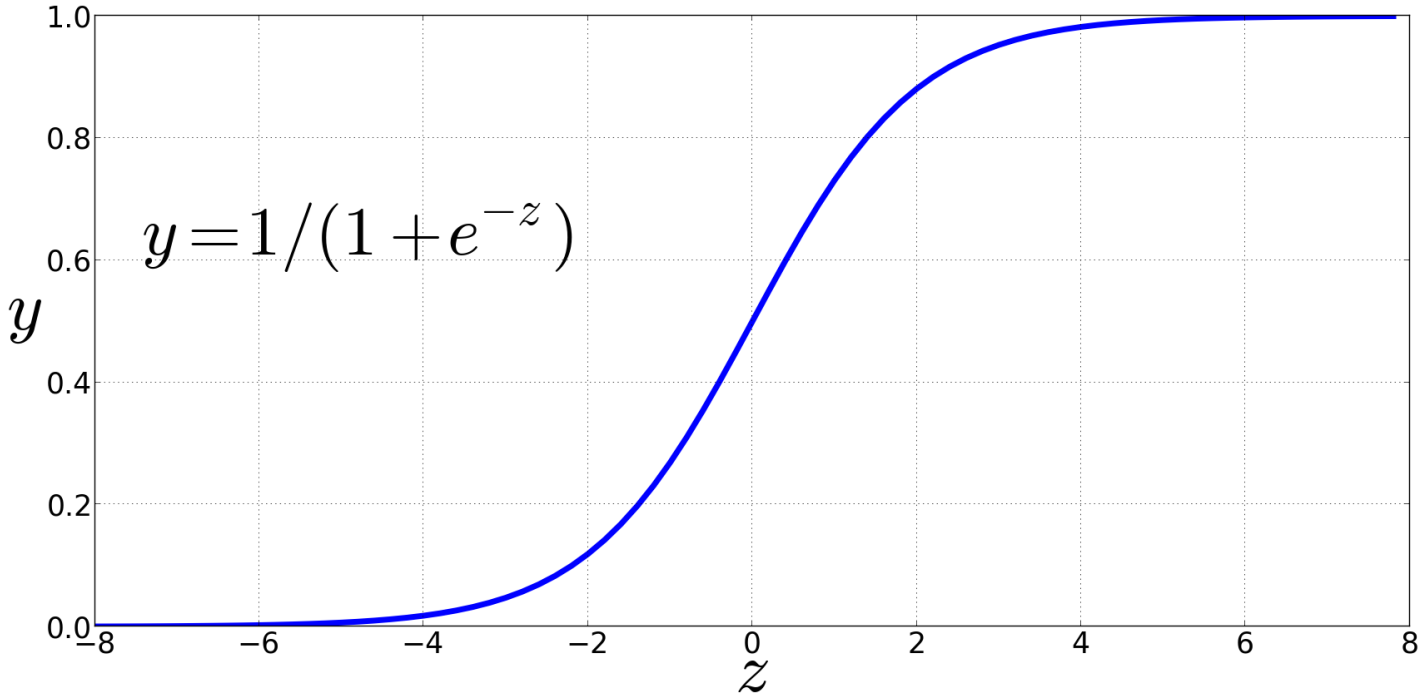


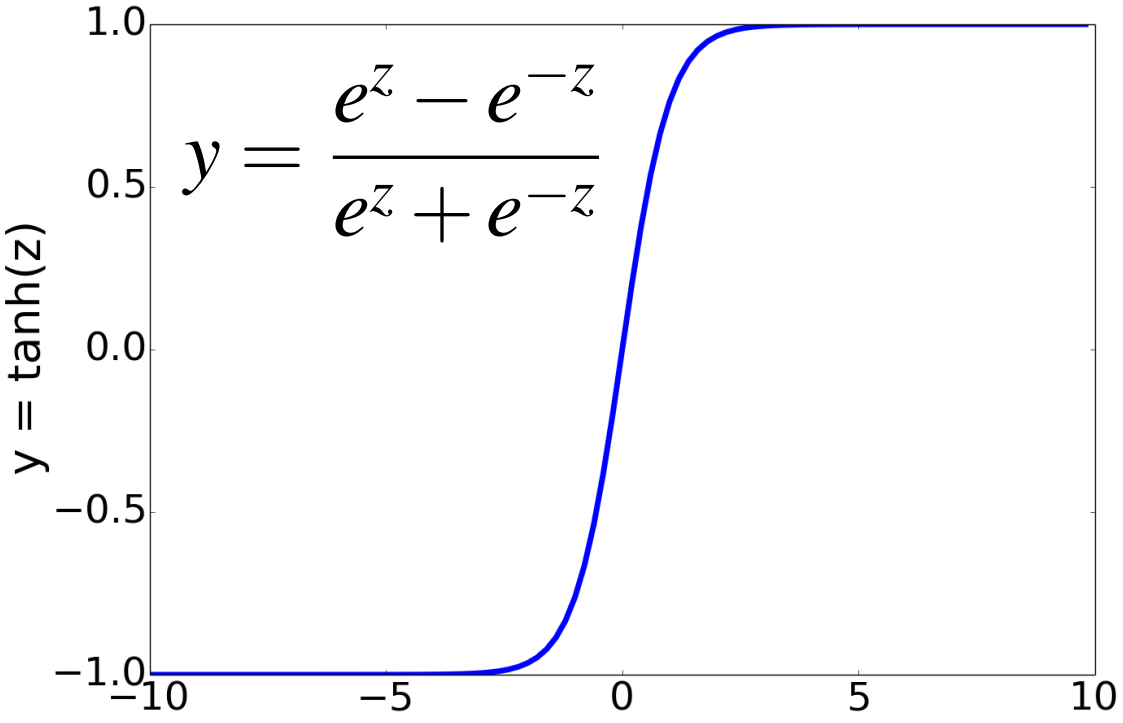
# Non-Linear Activation Functions

## Sigmoid

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

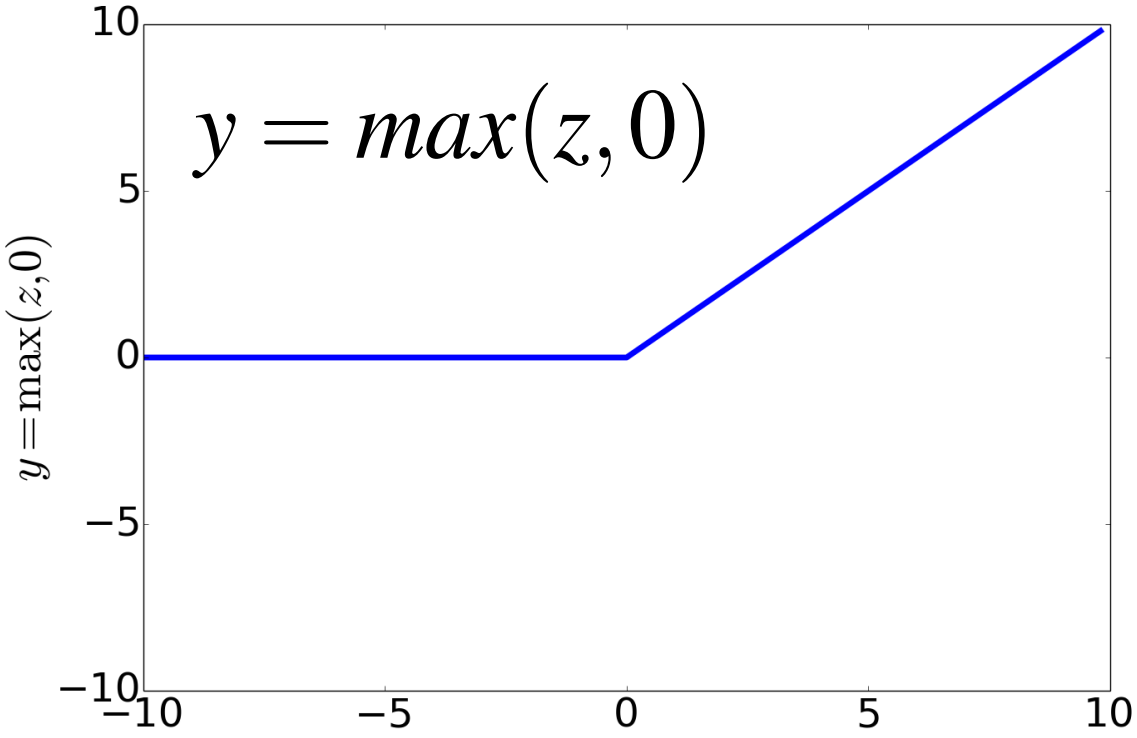


# Non-Linear Activation Functions




tanh

Most Common:




ReLU

Rectified Linear Unit

# Final unit again

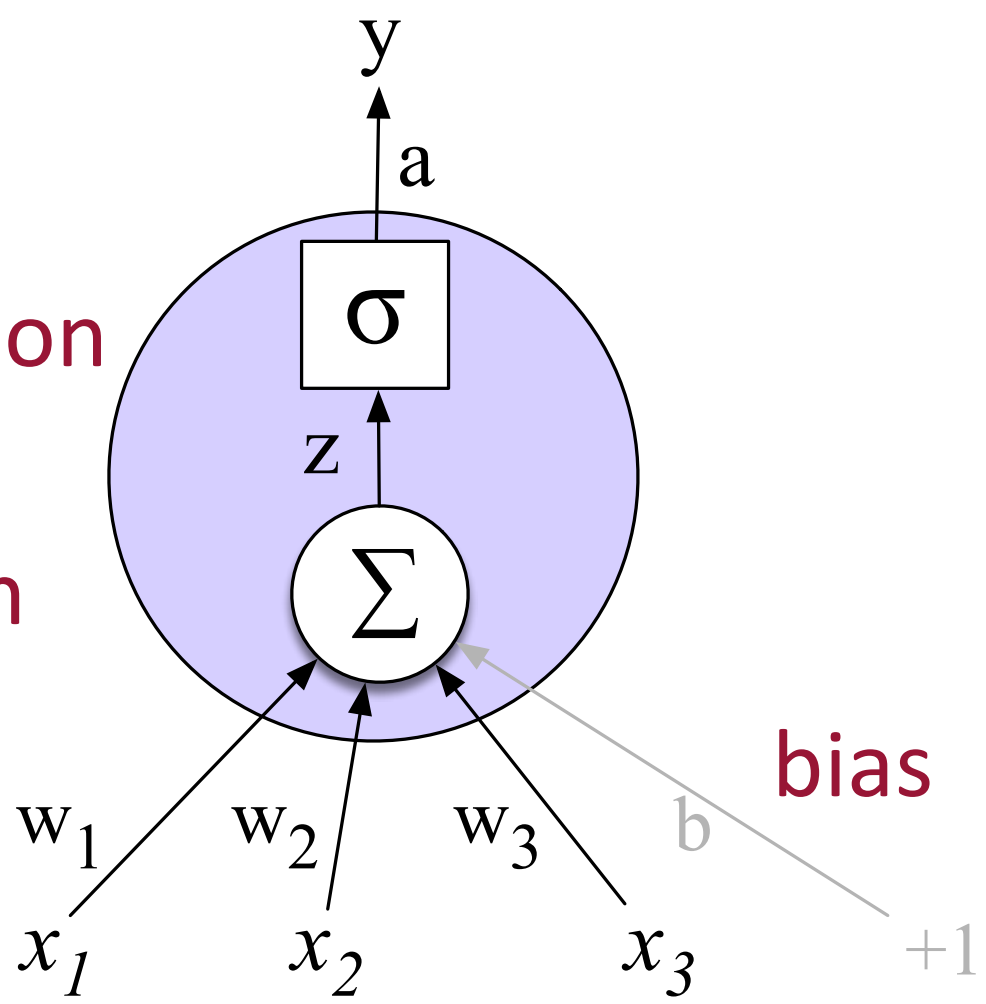
Output value

Non-linear activation function

Weighted sum

Weights

Input layer



# An example

- *Suppose a unit has:*
  - `w = [0.2,0.3,0.9]`
  - `b = 0.5`
- What happens with input x:
  - `x = [0.5,0.6,0.1]`

$$y = \sigma(w \cdot x + b) =$$

# An example

- *Suppose a unit has:*
  - `w = [0.2,0.3,0.9]`
  - `b = 0.5`
- What happens with input x:
  - `x = [0.5,0.6,0.1]`

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

$$\frac{1}{1 + e^{-(.5*.2 + .6*.3 + .1*.9 + .5)}} =$$

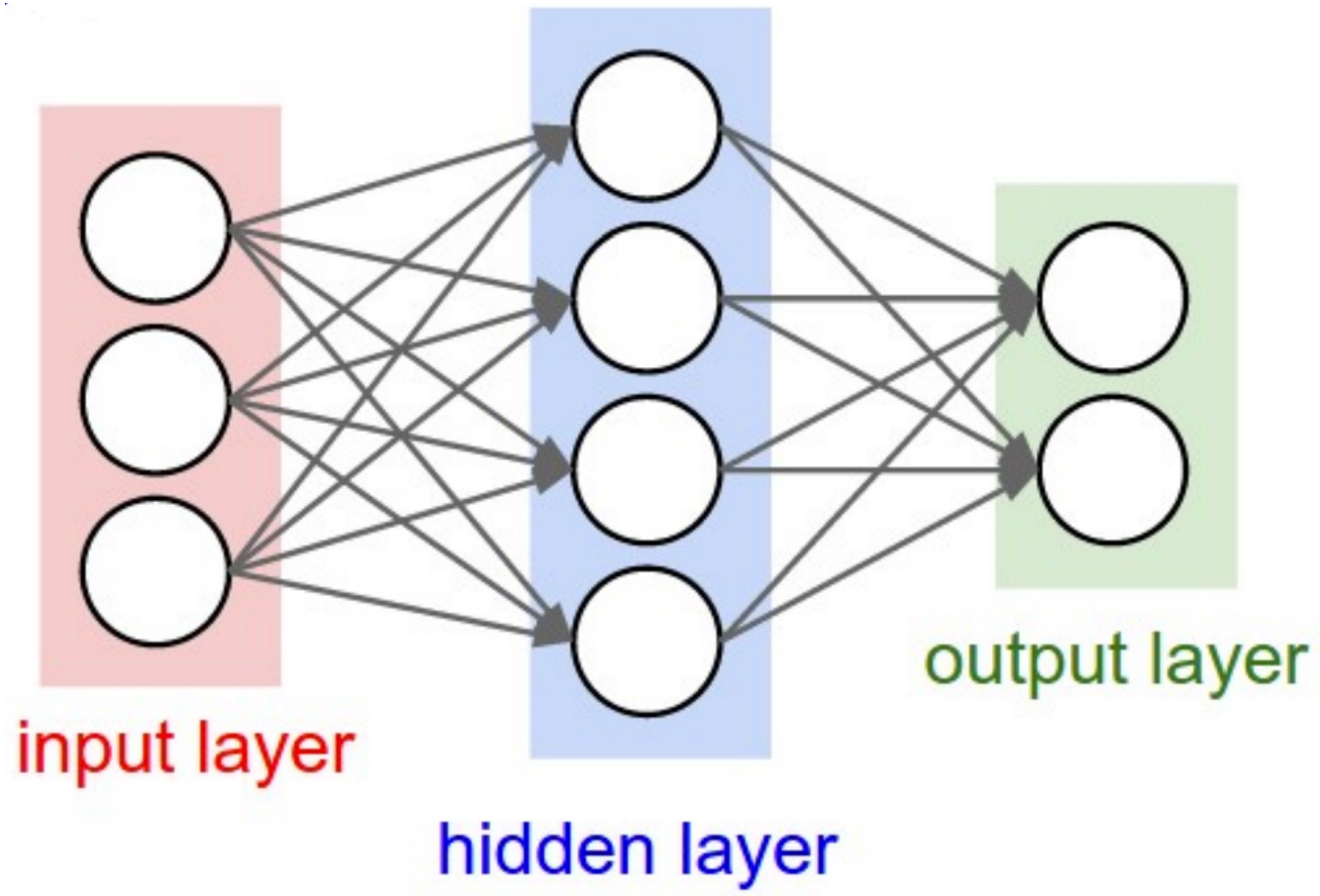
# An example

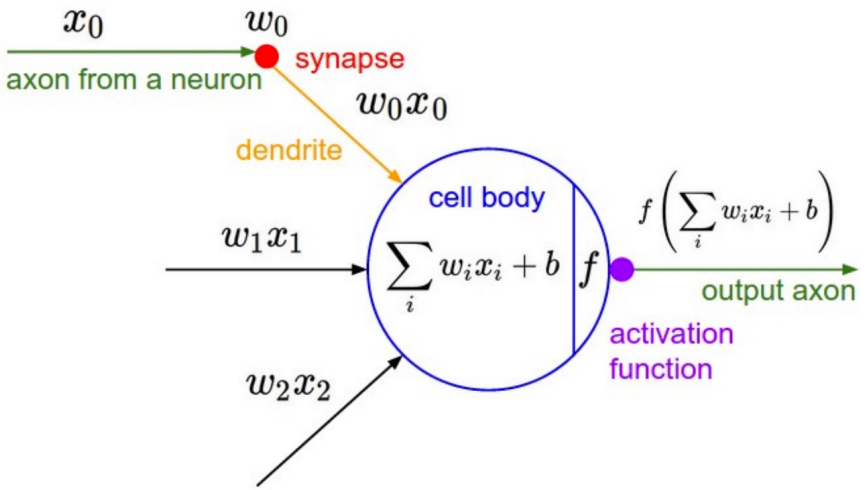
- *Suppose a unit has:*
  - `w = [0.2,0.3,0.9]`
  - `b = 0.5`
- What happens with input x:
  - `x = [0.5,0.6,0.1]`

$$y = \sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}} =$$

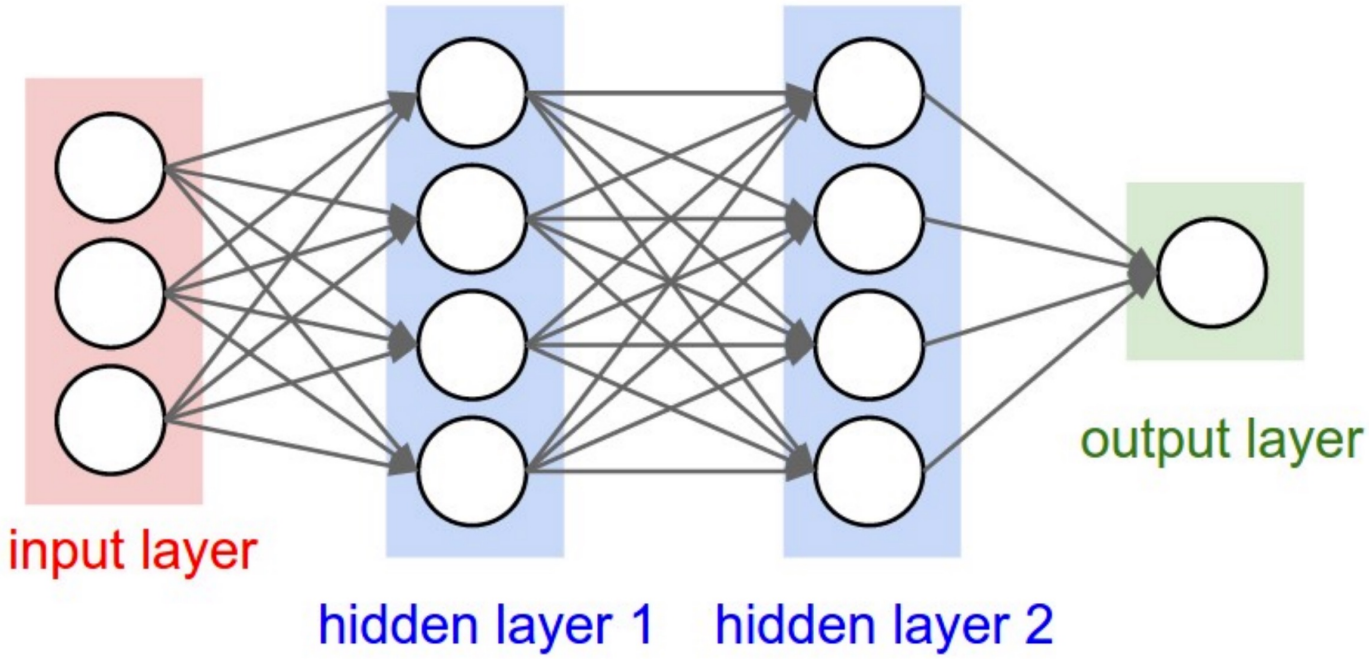
$$\frac{1}{1 + e^{-(.5*.2+.6*.3+.1*.9+.5)}} = \frac{1}{1 + e^{-0.87}} = .70$$

# Feed-forward Neural Network



# Feed-forward Neural Network




$$o = f(W_3 \cdot \vec{h_2} + b_3)$$

$$\vec{h_1} = f(W_1 \cdot \vec{x} + b_1) \qquad \vec{h_2} = f(W_2 \cdot \vec{h_1} + b_2)$$

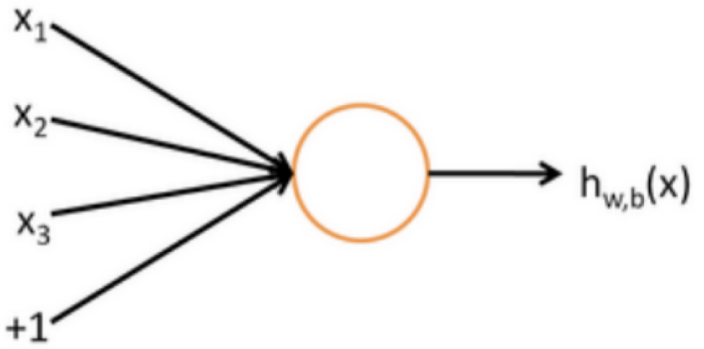
# Log-linear Models and Neural Nets

- A single-layer NN with a sigmoid activation, and  function is just a binary log-linear model
- If it's binary, we can assume: $\phi(x, CLASS_1) = -\phi(x, CLASS_2) := \phi(x)$

$$P(CLASS_1|x; w) = \frac{e^{w^T \phi(x,y)}}{e^{w^T \phi(x)} + e^{-w^T \phi(x)}} = \frac{1}{1 + e^{-2 \cdot w^T \phi(x)}}$$

$$h_{w,b}(z) = f(w^\top z + b)$$

$$f(u) = \frac{1}{1 + e^{-u}}$$



# One-hot vectors

- A vector of length |V|
- 1 for the target word and 0 for other words
- So if “popsicle” is vocabulary word #5, the **one-hot vector** is

  [0,0,0,0,1,0,0,0,0…….0]
- Often the vocabulary is truncated at some frequency threshold

# Softmax Layers

- Softmax layers turn vector outputs into a probability distribution

$$SOFTMAX : \mathcal{R}^n \rightarrow \mathcal{R}^n$$

$$SOFTMAX(\vec{x})_i = \frac{e^{x_i}}{\sum_i e^{x_i}}$$

- Log-linear models (with *n* labels) is equivalent to a FF network with no hidden layers, and a softmax layer at the end
  - Simple exercise: show the formulations express the same set of distributions

# (Multi-class) Log-linear Models and Soft-max

- A soft-max layer at the end is standardly used to define networks for multi-class classification

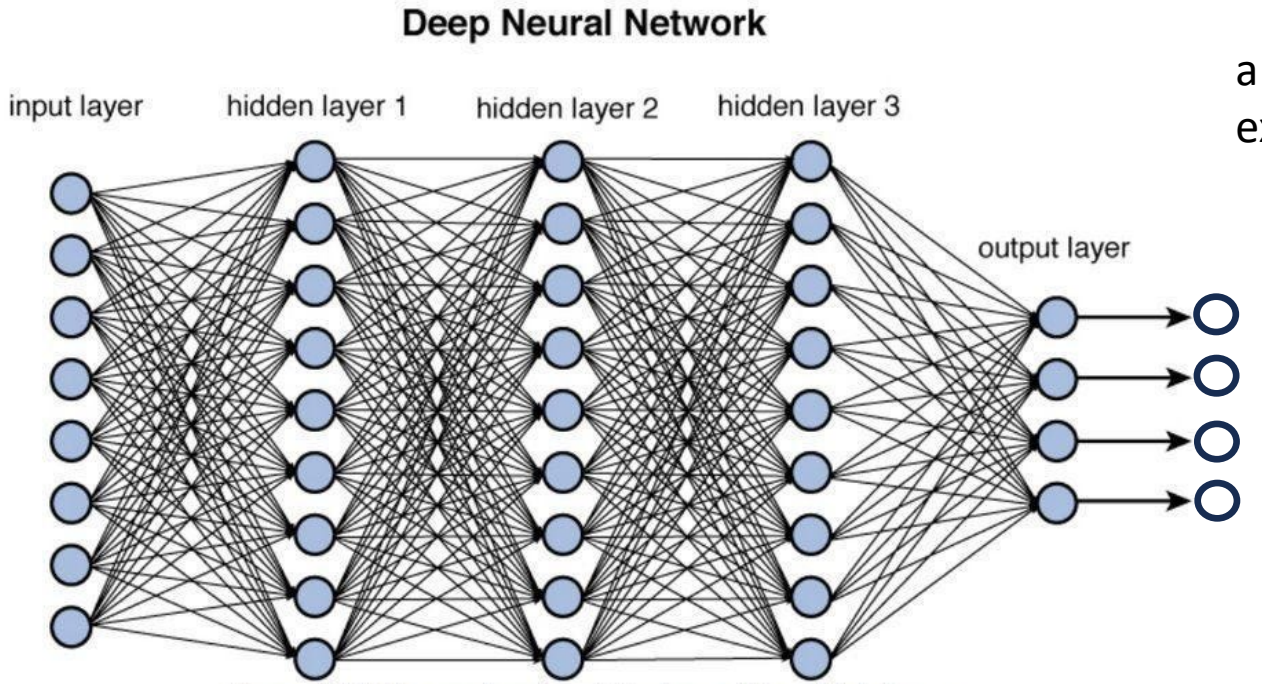


Figure 12.2 Deep network architecture with multiple layers.

a soft-max layer, which exponentiates and normalizes

$$\frac{e^a}{e^a + e^b + e^c + e^d}$$

$$\frac{e^d}{e^a + e^b + e^c + e^d}$$

# Log-linear Models and Softmax

- Log-linear models (with *n* labels) is equivalent to a FF network with no hidden layers, and a softmax layer at the end
  - Simple exercise: show the formulations express the same set of distributions

# Loss Functions

- In order to train neural networks, we need to define a **loss function**, which determines the degree to which a prediction error matters

- A loss function is a function that maps the predicted output and the ground truth output to a real number

# Loss Functions

- The standard training minimizes the expected loss over (generally: with additional terms, such as regularization):

$$L_S(\theta) = \frac{1}{m}\sum_{i=1}^{m} \ell(\theta; (x_i, y_i))$$

θ are the parameters of the network

- For classification, it is standard to use the cross-entropy loss. For a case where the ground truth is deterministic:

$$L_S(\theta) = -\frac{1}{m}\sum_{i=1}^{m} log(P_\theta(y_i|x_i))$$

the log probability the network assigns to the true label

# Gradient Descent with Backpropogation

- The standard way of training neural networks is through some form of stochastic gradient descent
  - That is, we compute the gradient of the loss function for each sample,
  - Often the updates are performed in batches, i.e., (relatively small) sets of examples, instead of summing over all examples

$$\sum_{i \in B} \nabla \ log(P(y_i|x_i))$$

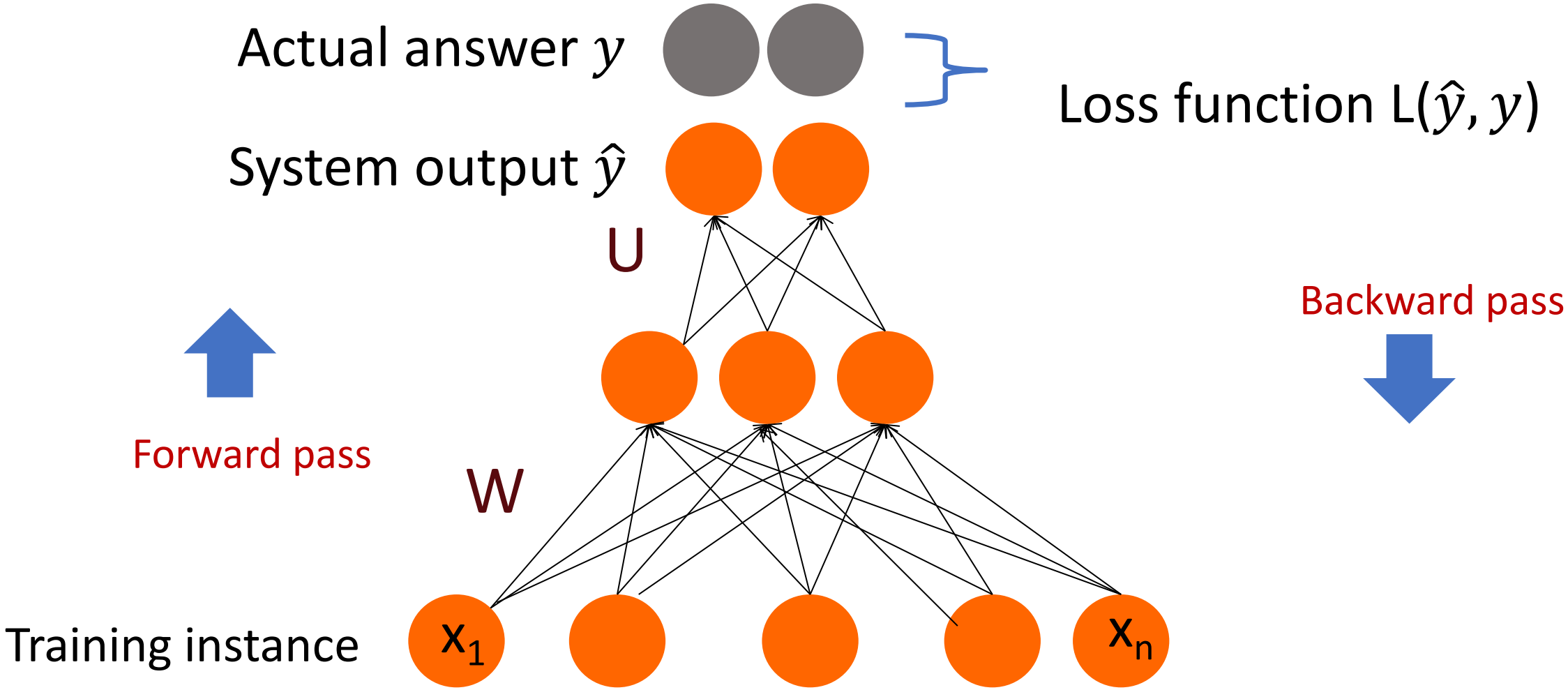
B is a batch

# Gradient Descent with Backpropogation

- The computation of the gradident can be done efficiently, using the chain rule, through an algorithm called *backpropogation*

- There are many subtleties here, but we will not pursue them in this course for lack of time
  - Instead, we will take a more hands-on perspective, leaving further discussion to courses on deep learning and optimization

# Intuition: Training a 2-layer Network



# Intuition: Training a 2-layer network

- For every training tuple $(x, y)$
  - Run *forward* computation to find $\hat{y}$
  - 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 loss
      - For every weight $w$ from input layer to the hidden layer
        - Update the weight

# Reminder: gradient descent for weight updates

- Use the partial derivative vector (gradient) of the loss function with respect to weights $\frac{d}{dw} L(f(x; w), y)$

- Move the weights in the opposite direction of the gradient

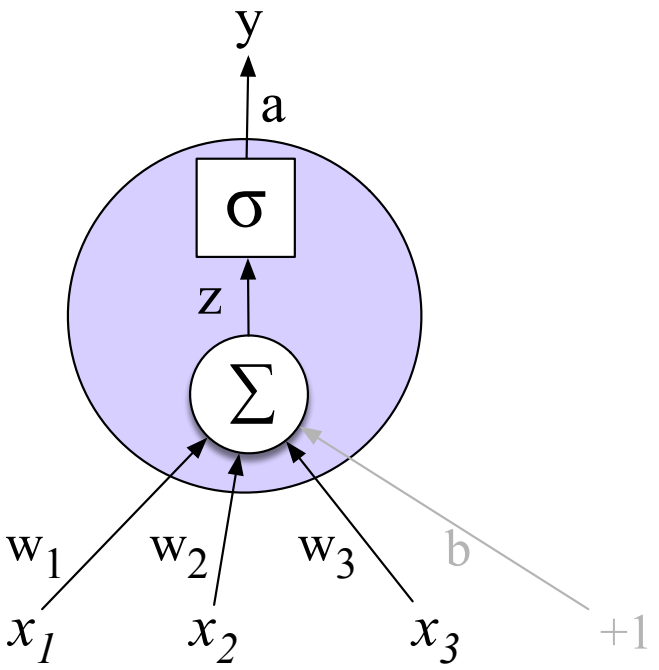
$$\mathrm{w} \leftarrow w - \eta \frac{d}{dw} L(f(x; w), y)$$

$\eta$ is the *learning rate* (“step size”)

# Where did that derivative come from?

- Using the chain rule $f(x) = u(v(x))$
- Intuition (see the text for details)

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




Derivative of the Activation

Derivative of the Loss

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \frac{\partial y}{\partial z} \frac{\partial z}{\partial w_i}$$

Derivative of the weighted sum

# How can I find that gradient for every weight in the network?

- These derivatives on the prior slide only give the updates for one weight layer: the last one

- What about deeper networks?
  - Lots of layers, different activation functions?
  - Use of the chain rule again!
  - Computation graphs and backward differentiation!

# 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*** (Rumelhart, Hinton, Williams, 1986)
  - which relies on ***computation graphs***

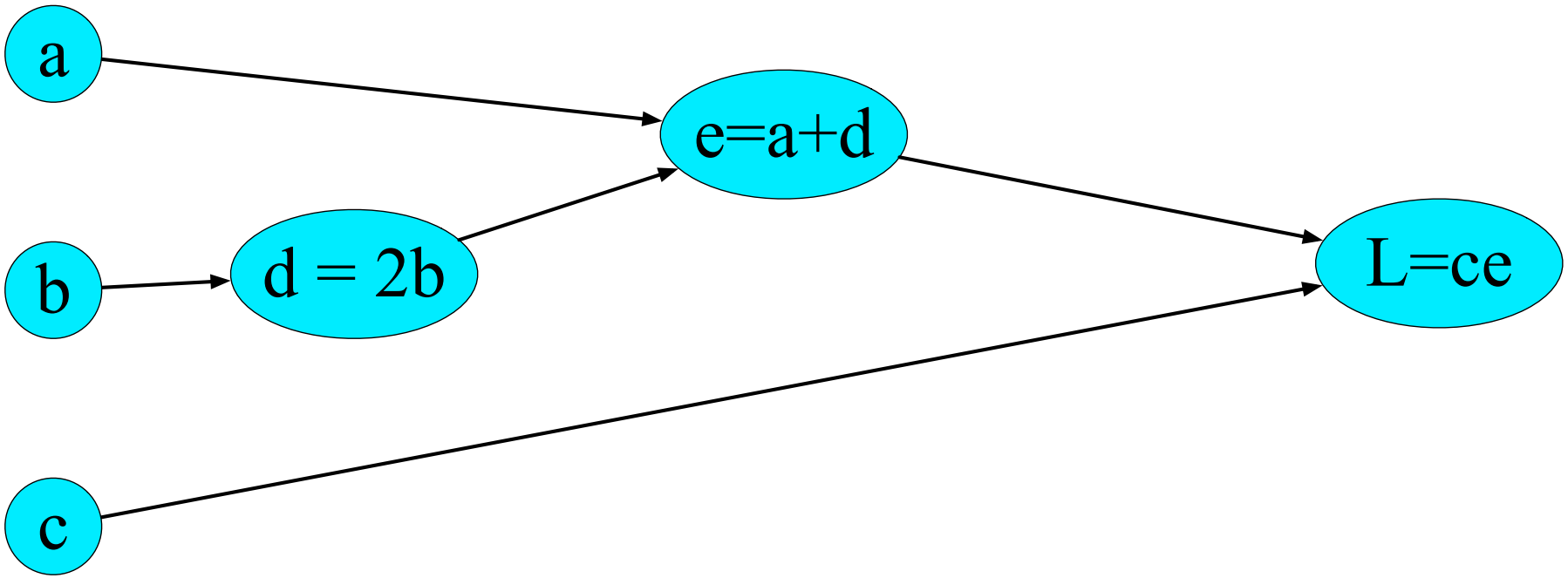
# Computation Graphs

A computation graph represents the process of computing a mathematical expression

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

Computations:

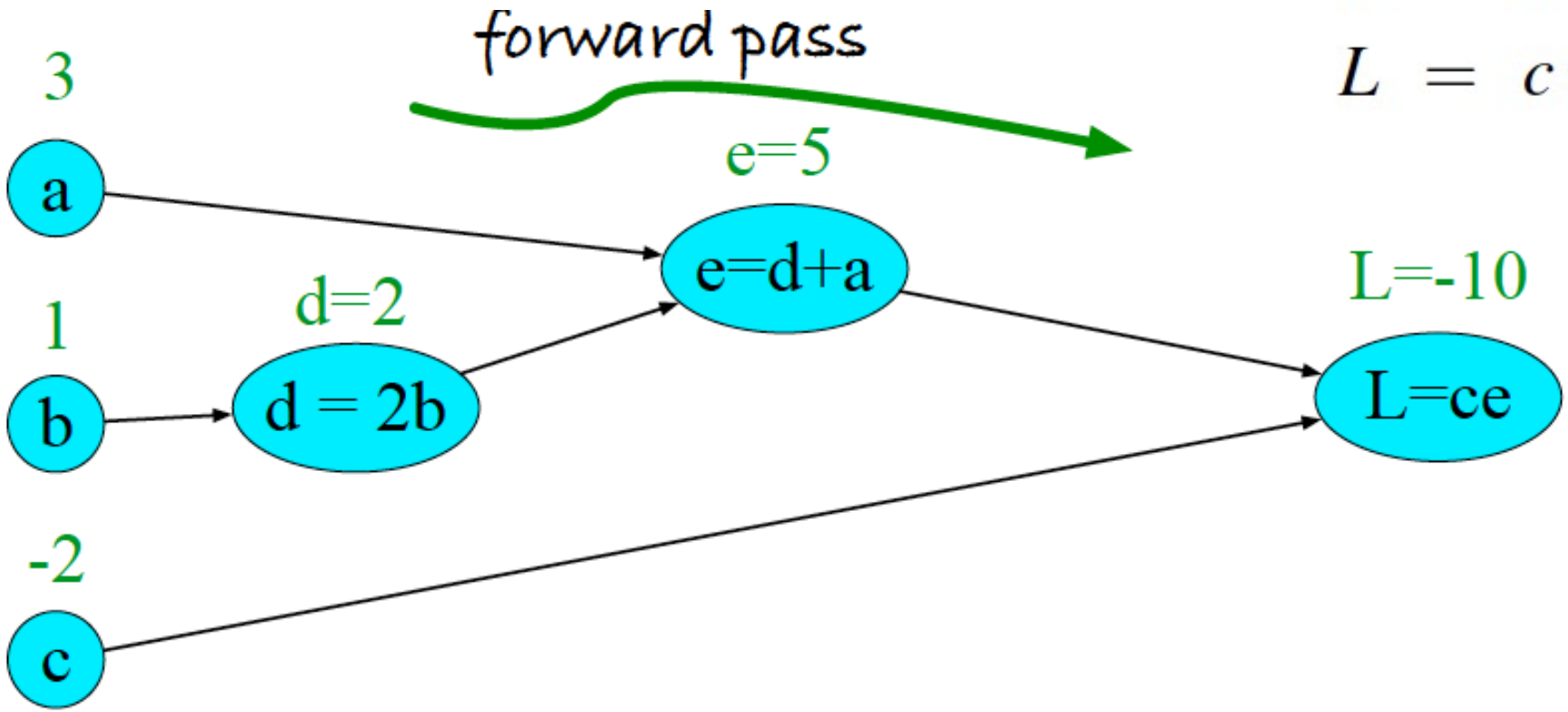
$$d = 2*b$$
$$e = a+d$$
$$L = c*e$$



# Example: $L(a,b,c) = c(a+2b)$

Computations:

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



# Backwards Differentiation in Computation Graphs

- The importance of the computation graph comes from the backward pass
- This is used to compute the derivatives needed for weight updates

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

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$

We want: $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$ *and* $\frac{\partial L}{\partial c}$

The derivative $\frac{\partial L}{\partial a}$, tells us how much a small change in *a* affects *L*

# The chain rule

- Computing the derivative of a composite function:

$$f(x) = u(v(x))$$

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

$$f(x) = u(v(w(x)))$$

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

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

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$

$$\begin{aligned} \frac{\partial L}{\partial c} &= e \\ \frac{\partial L}{\partial a} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial a} \\ \frac{\partial L}{\partial b} &= \frac{\partial L}{\partial e}\frac{\partial e}{\partial d}\frac{\partial d}{\partial b} \end{aligned}$$

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

$$\begin{aligned} d &= 2*b \\ e &= a+d \\ L &= c*e \end{aligned}$$

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

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

$$\begin{aligned} L = ce &: \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e \\ e = a+d &: \quad \frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1 \\ d = 2b &: \quad \frac{\partial d}{\partial b} = 2 \end{aligned}$$

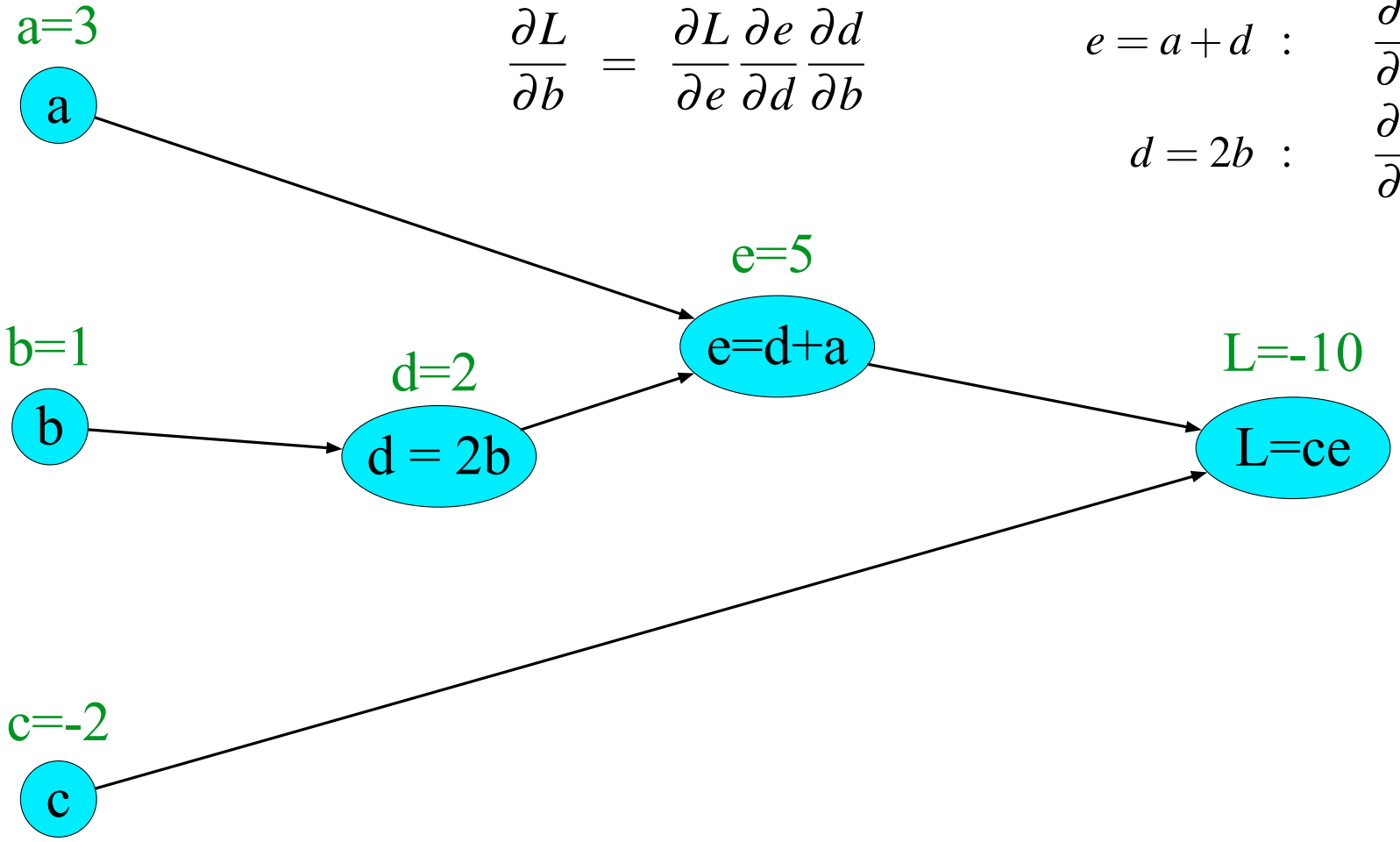
$$\frac{\partial L}{\partial a} = \frac{\partial L}{\partial e}\frac{\partial e}{\partial a}$$

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

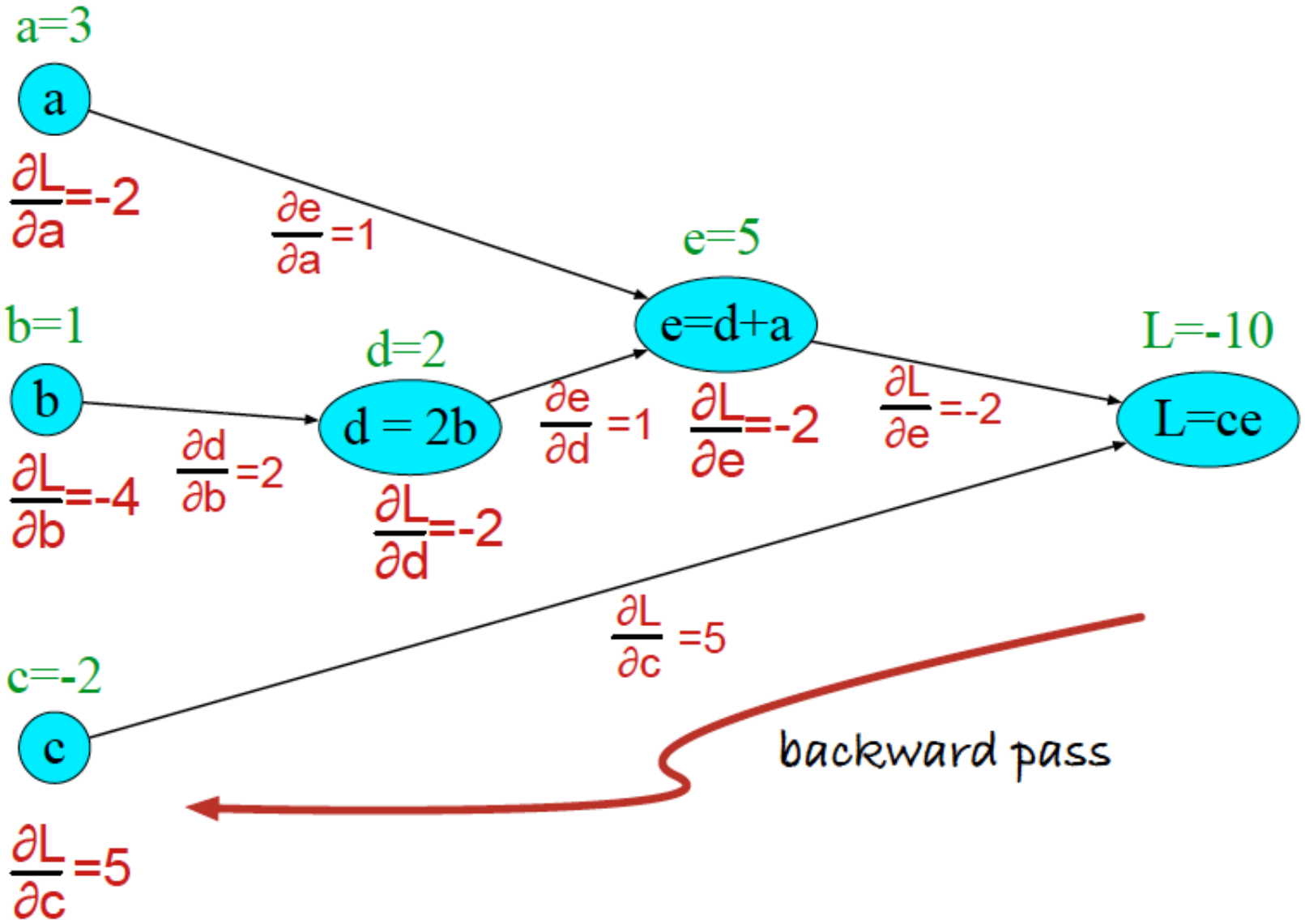
$$L = ce : \quad \frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$$

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

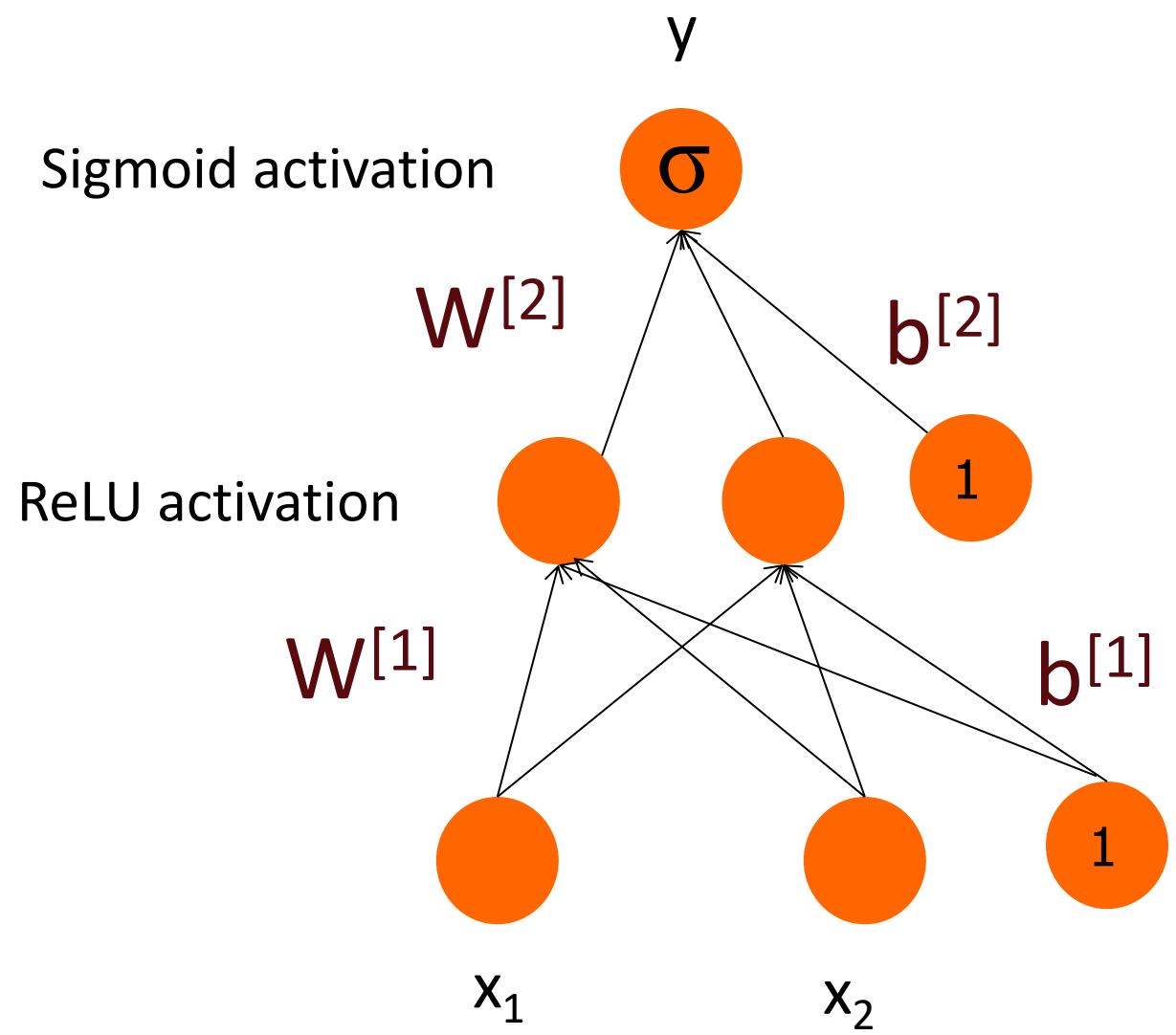
$$d = 2b : \quad \frac{\partial d}{\partial b} = 2$$



# Example


a=3
a
∂L/∂a=-2
∂e/∂a=1
e=5
e=d+a
b=1
b
∂L/∂b=-4
∂d/∂b=2
d=2
d = 2b
∂e/∂d=1
∂L/∂e=-2
∂L/∂e=-2
L=-10
L=ce
∂L/∂d=-2
∂L/∂c=5
c=-2
c
backward pass
∂L/∂c=5

# Backward differentiation on a two-layer network




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

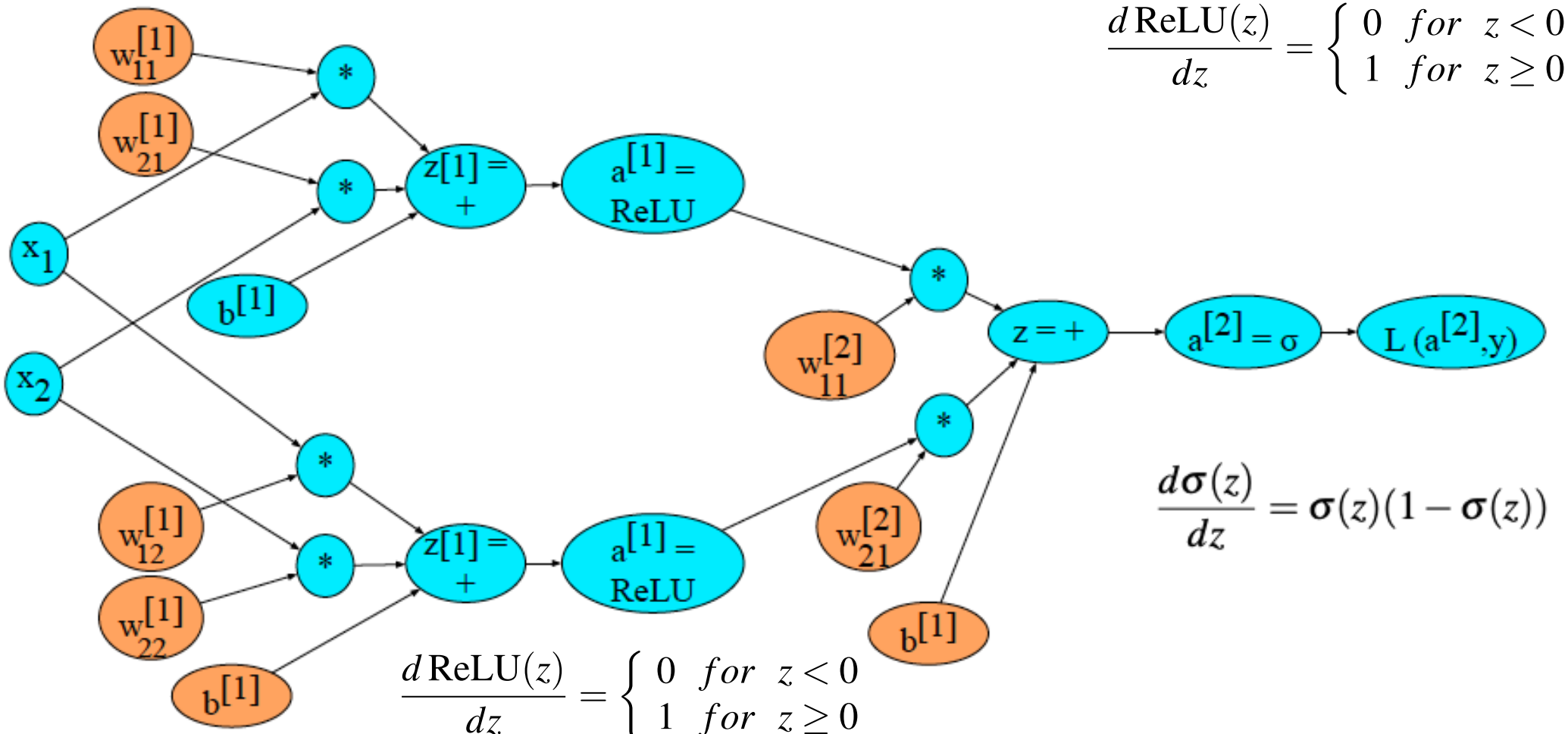
# Backward differentiation on a two-layer network

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

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

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

# Backward differentiation on a 2-layer network




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

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

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

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

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

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

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

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

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

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

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

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

# Simple Exercise

- Decide on an assignment for each of the inputs, and for each of the weights
- Compute the forward pass
- Compute the backward pass
- Write down the gradient for the current input and weights

# Summary

- 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

# Computation Graphs In PyTorch

- Computing gradients can be tiresome
- Thankfully, many packages can do all the work for us
- Here we will demonstrate backpropagation in a computation graph in *PyTorch*

# Tensors

Tensors are the basic element of computation in PyTorch. They can be seen as a multidimensional array. Tensors can be created by values, from numpy, or using special functions.

```
[58] import torch
     import numpy as np
     # Create a tensor
     t1 = torch.tensor([0., 1., 2., 3.])
     a = np.array([0., 1., 2., 3.])
     t2 = torch.tensor(a)
     t3 = torch.arange(4, dtype=float)

     # Print
     print(t1)
     print(t2)
     print(t3)
```

# Tensors

Tensors are the basic element of computation in PyTorch. They can be seen as a multidimensional array. Tensors can be created by values, from numpy, or using special functions.

```
[58] import torch
     import numpy as np
     # Create a tensor
     t1 = torch.tensor([0., 1., 2., 3.])
     a = np.array([0., 1., 2., 3.])
     t2 = torch.tensor(a)
     t3 = torch.arange(4, dtype=float)

     # Print
     print(t1)
     print(t2)
     print(t3)
```

```
tensor([0., 1., 2., 3.])
tensor([0., 1., 2., 3.], dtype=torch.float64)
tensor([0., 1., 2., 3.], dtype=torch.float64)
```

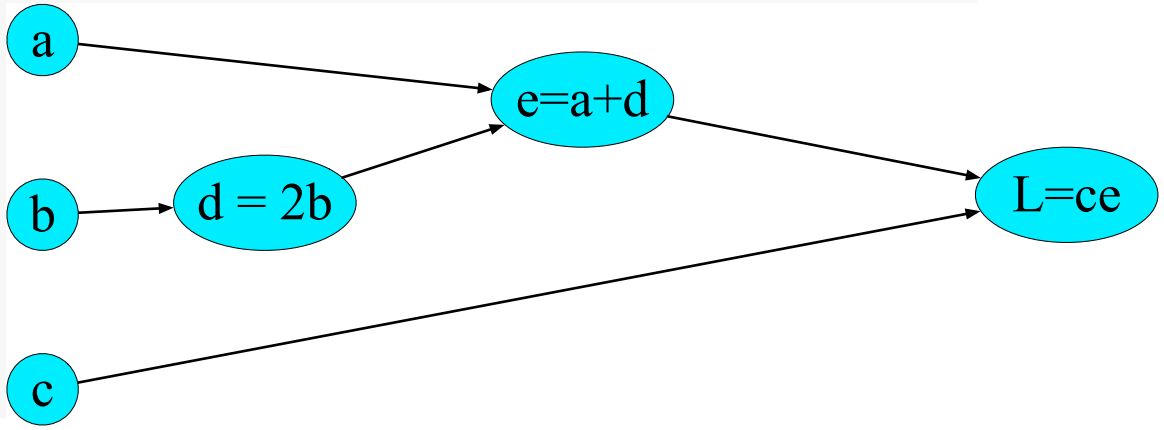
# Computational Graphs (1)

Computational graphs are used to represent computations in PyTorch. Nodes in the graph represent tensors, and edges represent the operations between tensors.

Let's take an example of a simple computation.

```
[14] # Create tensors.
     a = torch.tensor(3., requires_grad=True)
     b = torch.tensor(1., requires_grad=True)
     c = torch.tensor(-2., requires_grad=True)

     # Build a computational graph
     d = 2 * b
     e = d + a
     L = c * e
     print(L)
```



# Computational Graphs (1)

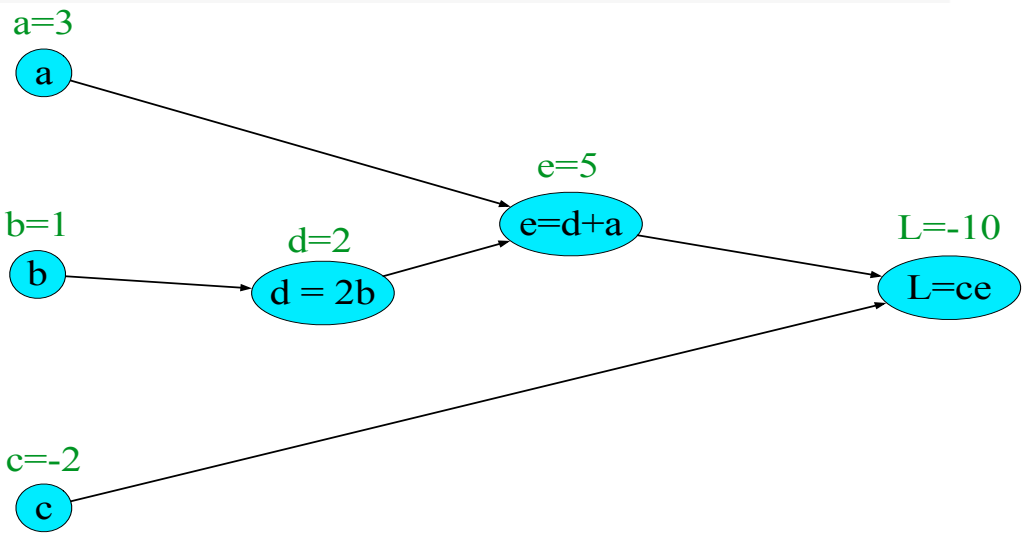
Computational graphs are used to represent computations in PyTorch. Nodes in the graph represent tensors, and edges represent the operations between tensors.

Let's take an example of a simple computation.

```
[14] # Create tensors.
     a = torch.tensor(3., requires_grad=True)
     b = torch.tensor(1., requires_grad=True)
     c = torch.tensor(-2., requires_grad=True)

     # Build a computational graph
     d = 2 * b
     e = d + a
     L = c * e
     print(L)
```

```
tensor(-10., grad_fn=<MulBackward0>)
```



# Computational Graphs (2)

```
[20] # Print out the gradients.
     print("L grads:")
     print(L.grad_fn)
     print(L.grad_fn.next_functions)
     print(L.grad_fn.next_functions[0][0].variable)
     print("e grads:")
     print(e.grad_fn)
     print(e.grad_fn.next_functions)
     print(e.grad_fn.next_functions[1][0].variable)
     print("d grads:")
     print(d.grad_fn)
     print(d.grad_fn.next_functions)
     print(d.grad_fn.next_functions[0][0].variable)
     print("a grads:")
     print(a.grad_fn)
```

# Computational Graphs (2)

```
[20] # Print out the gradients.
     print("L grads:")
     print(L.grad_fn)
     print(L.grad_fn.next_functions)
     print(L.grad_fn.next_functions[0][0].variable)
     print("e grads:")
     print(e.grad_fn)
     print(e.grad_fn.next_functions)
     print(e.grad_fn.next_functions[1][0].variable)
     print("d grads:")
     print(d.grad_fn)
     print(d.grad_fn.next_functions)
     print(d.grad_fn.next_functions[0][0].variable)
     print("a grads:")
     print(a.grad_fn)
```

```
L grads:
<MulBackward0 object at 0x7cd545263d60>
((<AccumulateGrad object at 0x7cd5452627d0>, 0), (<AddBackward0
tensor(-2., requires_grad=True)
e grads:
<AddBackward0 object at 0x7cd545261ea0>
((<MulBackward0 object at 0x7cd545262920>, 0), (<AccumulateGrad
tensor(3., requires_grad=True)
d grads:
<MulBackward0 object at 0x7cd545262920>
((<AccumulateGrad object at 0x7cd545260220>, 0), (None, 0))
tensor(1., requires_grad=True)
a grads:
None
```

c

a

b

# Computational Graphs (3)

```
# Compute gradients.
L.backward()

# Print out the gradients.
print(a.grad)
print(b.grad)
print(c.grad)
```

```
tensor(-2.)
tensor(-4.)
tensor(5.)
```

Notice that in PyTorch the graph is revealed when iterating over the nodes, which are the python variables. Every variable that was defined as a calculation from variables that require gradients adds a node to the graph. This method of creating the graph is called **dynamic computational graphs**. This provides a lot of flexibility and makes PyTorch more intuitive and (relatively) easy for debugging.

Other libraries, like TensorFlow use **static computational graphs** which are defined once and then run many times. In this method the graph is optimizable and possibly giving a performance boost during training large models. The price comes in flexibility.

# Modules (1)

Modules in PyTorch help us design complex neural network architectures in a cleaner and more accessible way. PyTorch's torch.nn package helps us in designing complex architectures easily.

```
[56] import torch.nn as nn

     class TwoLayerNetwork(nn.Module):
         def __init__(self):
             super().__init__()
             self.linear1 = nn.Linear(2, 2, bias=True)
             self.linear2 = nn.Linear(2, 2, bias=True)

         def forward(self, x):
             l1 = self.linear1(x)
             r1 = nn.ReLU()(l1)
             l2 = self.linear1(r1)
             r2 = nn.ReLU()(l2)
             return nn.Sigmoid()(r2)
```

# Modules (2)

```
# create an instance of the model
model = TwoLayerNetwork()
print("Parameters in the network")
print(list(model.named_parameters()))

print("Example output")
with torch.no_grad():
  print(model(torch.tensor([[1., 1.], [2., 2.]])))
```

```
Parameters in the network
('linear1.weight', Parameter containing:
tensor([[ 0.3091,  0.0702],
        [ 0.1436, -0.5881]], requires_grad=True))
('linear1.bias', Parameter containing:
tensor([0.2081, 0.0784], requires_grad=True))
('linear2.weight', Parameter containing:
tensor([[ 0.3993, -0.4095],
        [ 0.4800, -0.6442]], requires_grad=True))
('linear2.bias', Parameter containing:
tensor([ 0.2295, -0.2573], requires_grad=True))
Example output
tensor([[0.5962, 0.5406],
        [0.6241, 0.5541]])
```

# Optimizers (1)

Optimizers are used to optimize the model parameters. PyTorch provides a package called torch.optim that contains many common optimizers, such as SGD and Adam.

```
[57] import torch.optim as optim

     # create an optimizer
     optimizer = optim.SGD(model.parameters(), lr = 0.01)

     # input
     x = torch.tensor([[1.0, 1.0]])
     # forward pass, which is the default call
     output = model(x)

     # compute loss
     loss = (output - torch.tensor([[2., 2.]]))**2
     loss = loss.mean()
     # zero gradients
     optimizer.zero_grad()
     # backpropagation
     loss.backward()
     # update weights
     optimizer.step()
```

# Optimizers (2)

```
print("Parameters in the network after one step")
print(list(model.named_parameters()))
```

```
Parameters in the network after one step
('linear1.weight', Parameter containing:
tensor([[ 0.3127,  0.0717],
        [ 0.1457, -0.5881]], requires_grad=True))
('linear1.bias', Parameter containing:
tensor([0.2130, 0.0821], requires_grad=True))
('linear2.weight', Parameter containing:
tensor([[ 0.3993, -0.4095],
        [ 0.4800, -0.6442]], requires_grad=True))
('linear2.bias', Parameter containing:
tensor([ 0.2295, -0.2573], requires_grad=True))
```

# Notes

- All code examples are in a notebook (in Moodle)
  - Code can really be learned only by experience, so experiment with it!
  - See notebook for additional practical advice