

CS4501: Introduction to Computer Vision

Neural Networks (NNs)

Multi-layer Perceptrons (MLPs)

By Fuwen Tan (ft3ex@virginia.edu)



Previous

- **Softmax Classifier**
 - Inference vs Training
 - Gradient Descent (GD)
 - Stochastic Gradient Descent (SGD)
 - mini-batch Stochastic Gradient Descent (SGD)
- **Max-Margin Classifier**
- **Regression vs Classification**
- **Issues with Generalization / Overfitting**
 - Regularization / momentum

Today's Class

Neural Networks

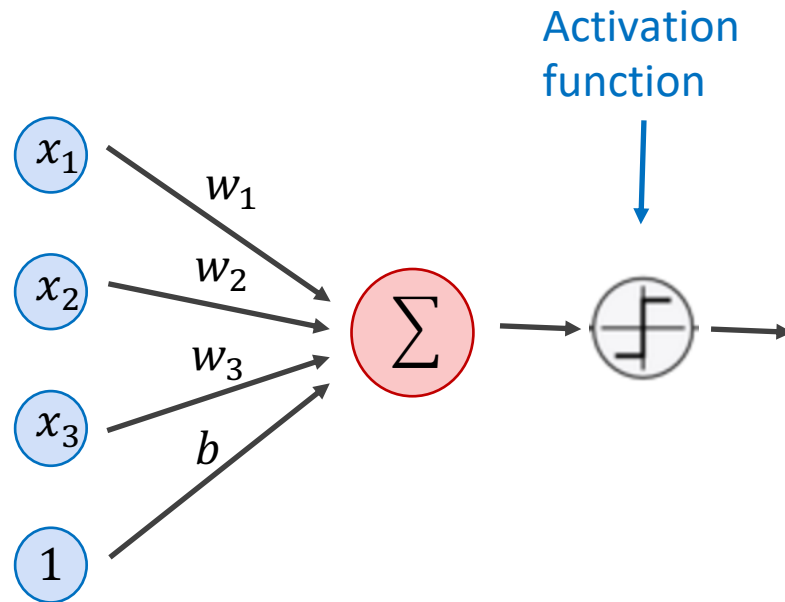
- The Perceptron Model
- The Multi-layer Perceptron (MLP)
- Forward-pass in an MLP (Inference)
- Backward-pass in an MLP (Backpropagation)

Perceptron Model

Frank Rosenblatt (1957) - Cornell University

A Linear function with
a binary output

$$f(x) = \begin{cases} 1, & \text{if } \sum_{i=0}^n w_i x_i + b > 0 \\ 0, & \text{otherwise} \end{cases}$$



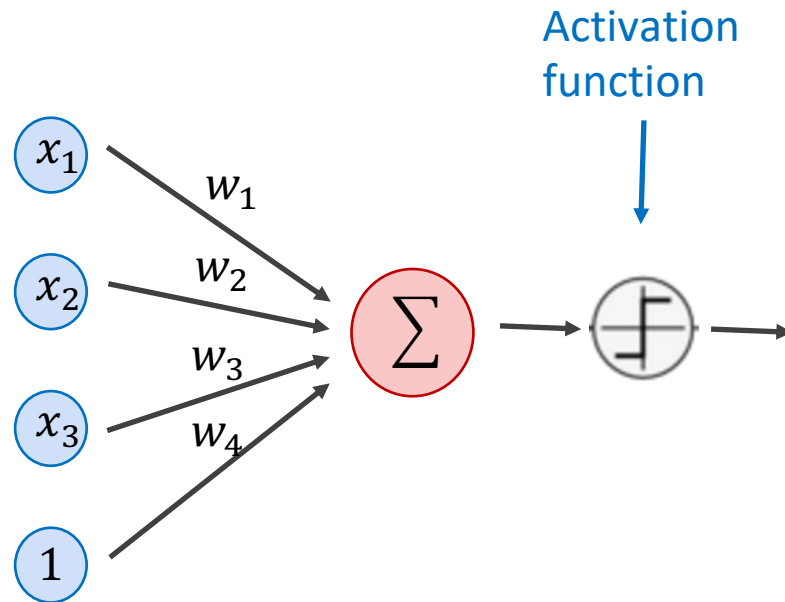
More: <https://en.wikipedia.org/wiki/Perceptron>

Perceptron Model

Frank Rosenblatt (1957) - Cornell University

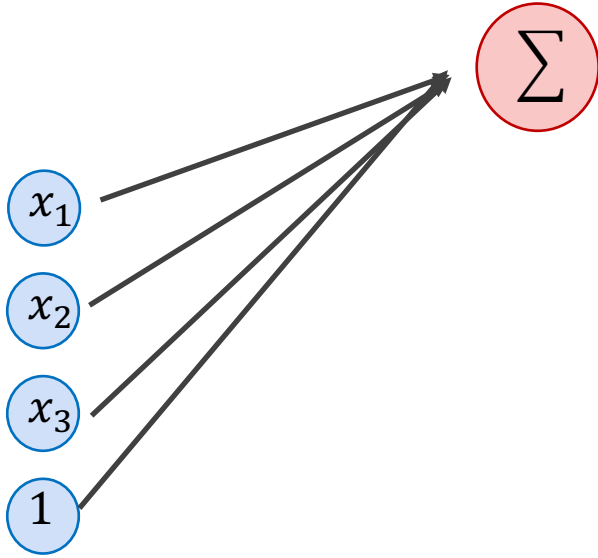
$$X = [x_1, x_2, x_3, 1]$$
$$W = [w_1, w_2, w_3, w_4]$$

$$f(x) = \begin{cases} 1, & \text{if } W \cdot X^T > 0 \\ 0, & \text{otherwise} \end{cases}$$

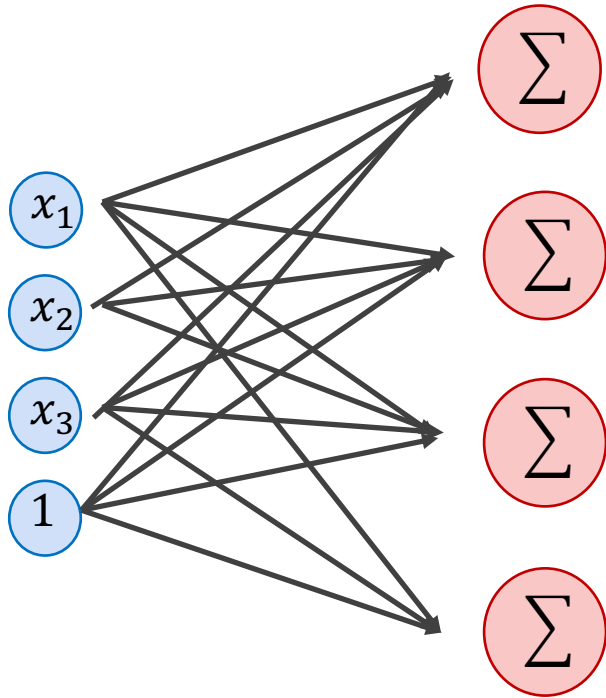


More: <https://en.wikipedia.org/wiki/Perceptron>

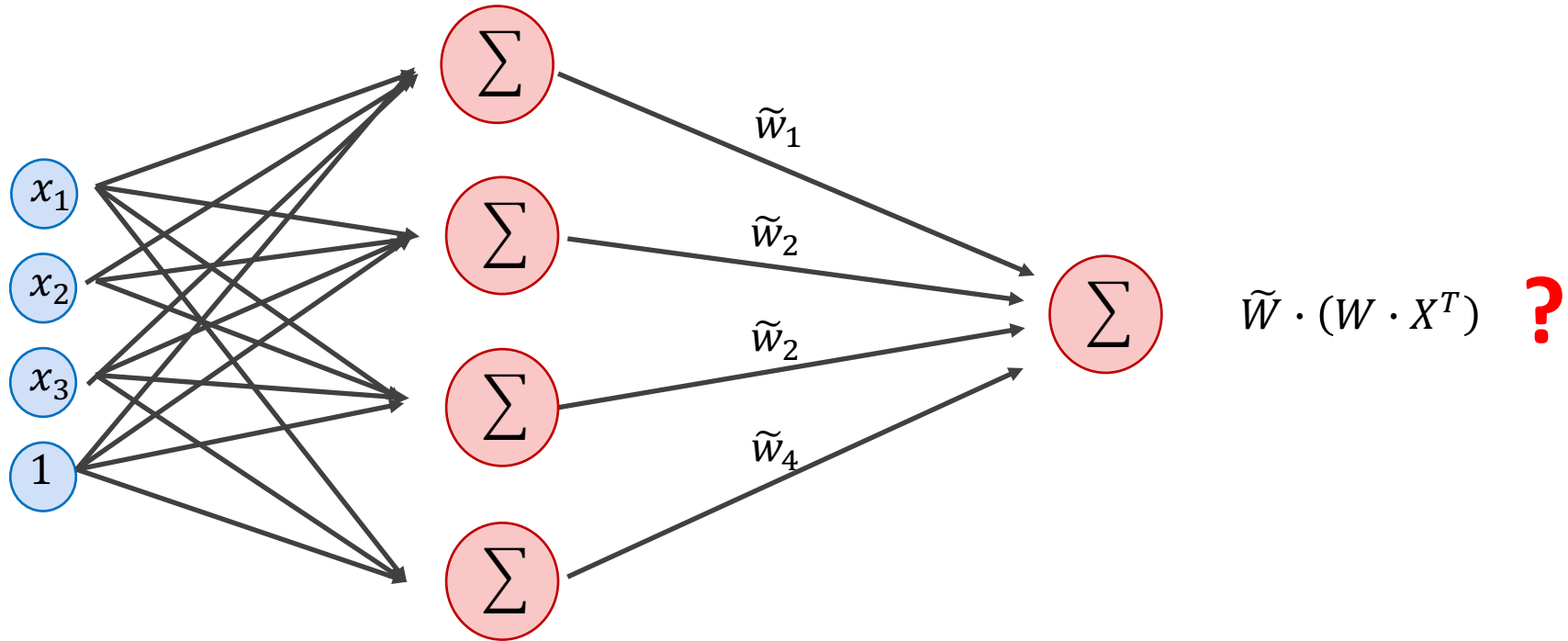
How Do We Get Two-Layer Perceptron



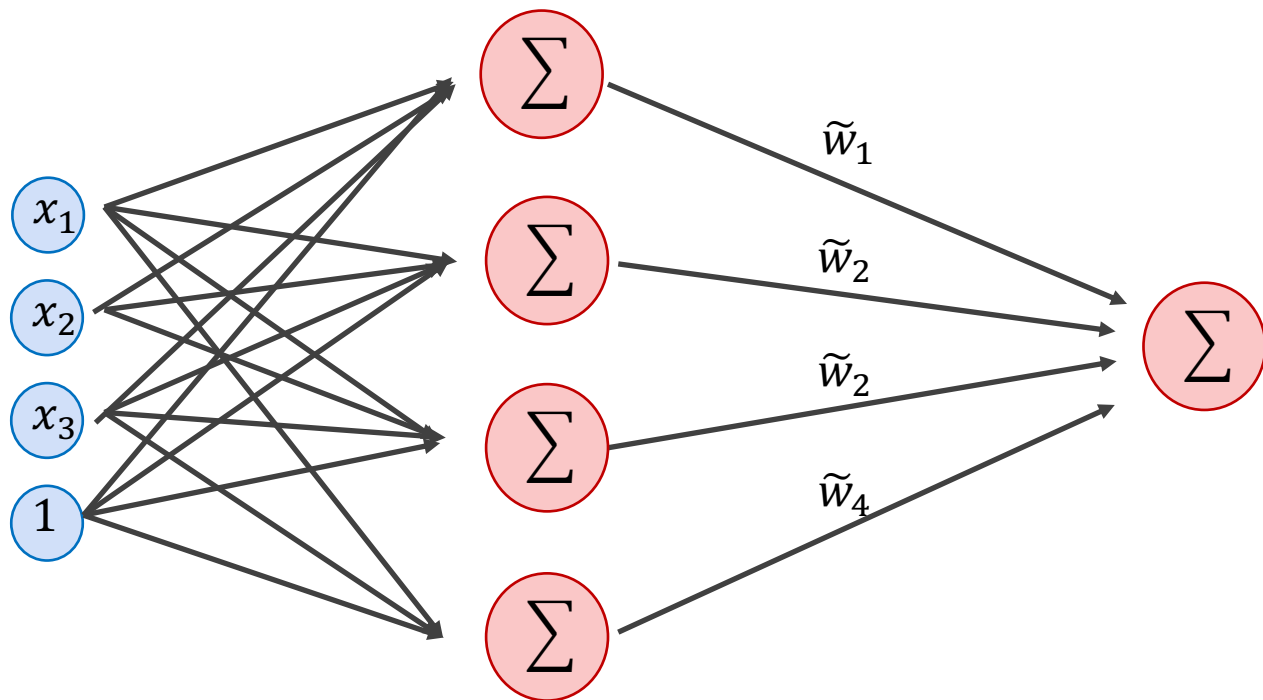
How Do We Get Two-Layer Perceptron



How Do We Get Two-Layer Perceptron



How Do We Get Two-Layer Perceptron



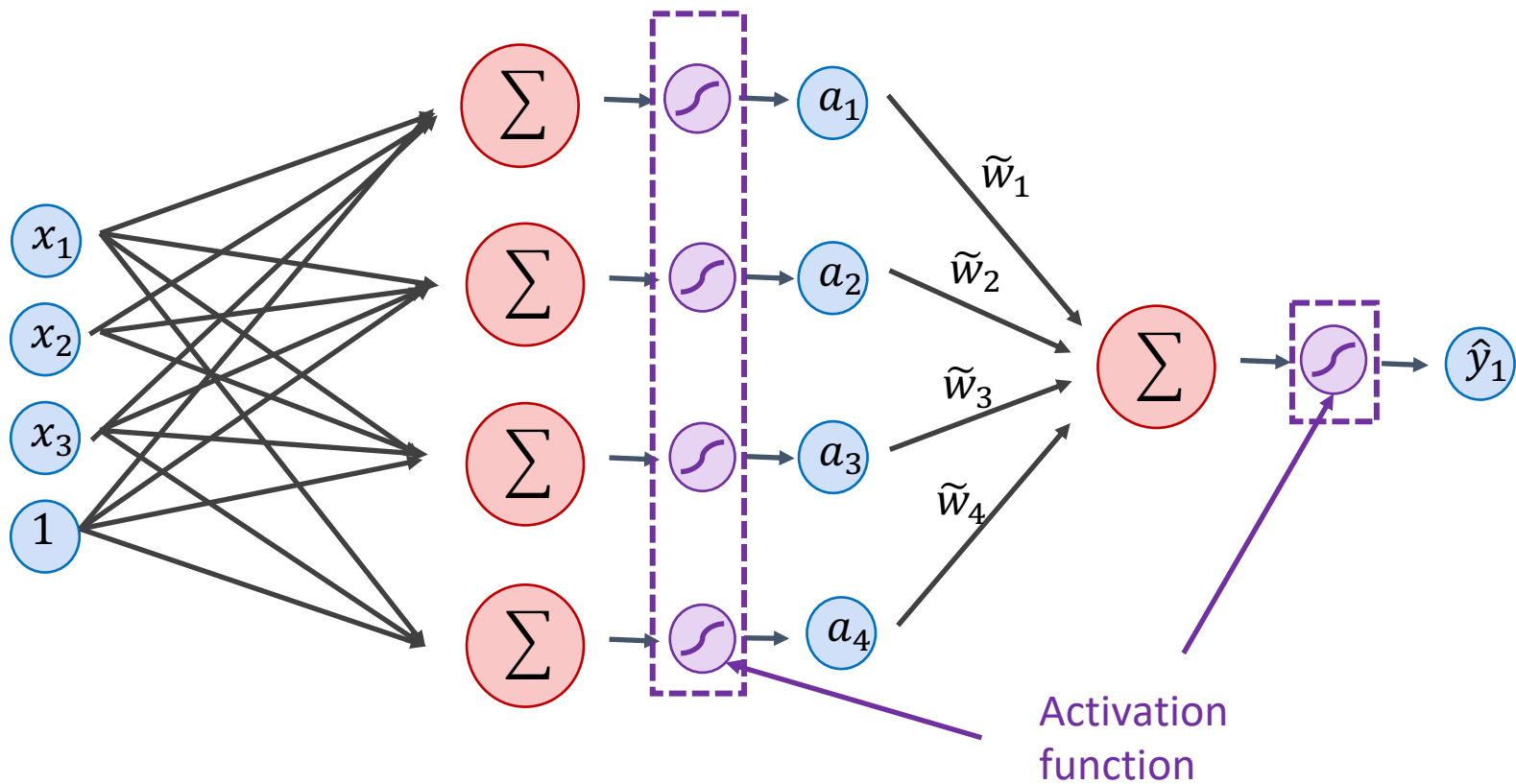
$$\begin{aligned} & \tilde{W} \cdot (W \cdot X^T) \\ &= (\tilde{W} \cdot W) \cdot X^T \\ &= \hat{W} \cdot X^T \end{aligned}$$

X

Where

$$\hat{W} = \tilde{W} \cdot W$$

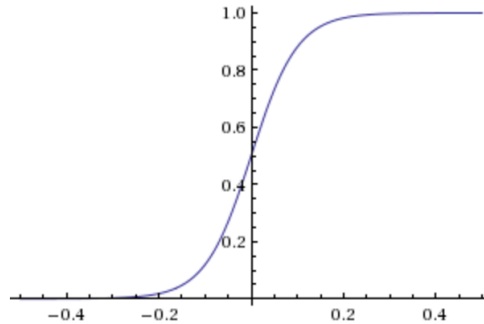
Two-Layer Perceptron



Activation Functions



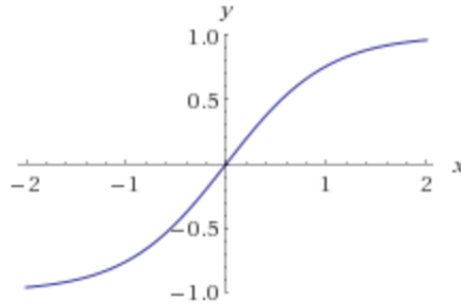
$$\textit{sigmoid}(x) = \frac{1}{1 + e^{-x}} \quad \in [0,1]$$



Activation Functions



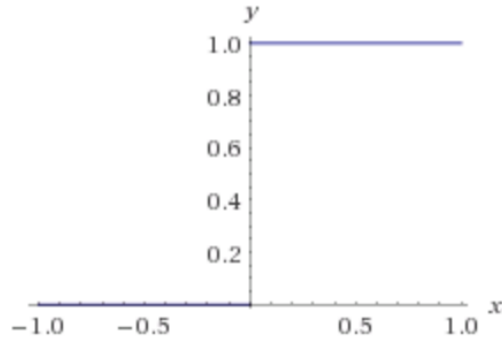
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \in [-1, 1]$$



Activation Functions



$$\text{step}(x) = \begin{cases} 1, & x > 0 \\ 0, & \text{otherwise} \end{cases}$$

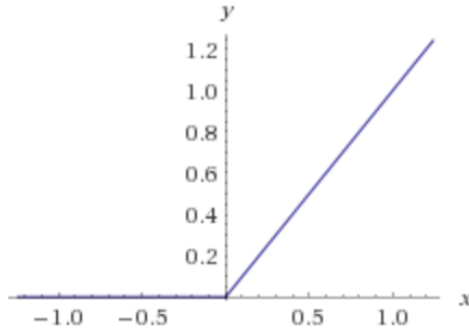


Activation Functions

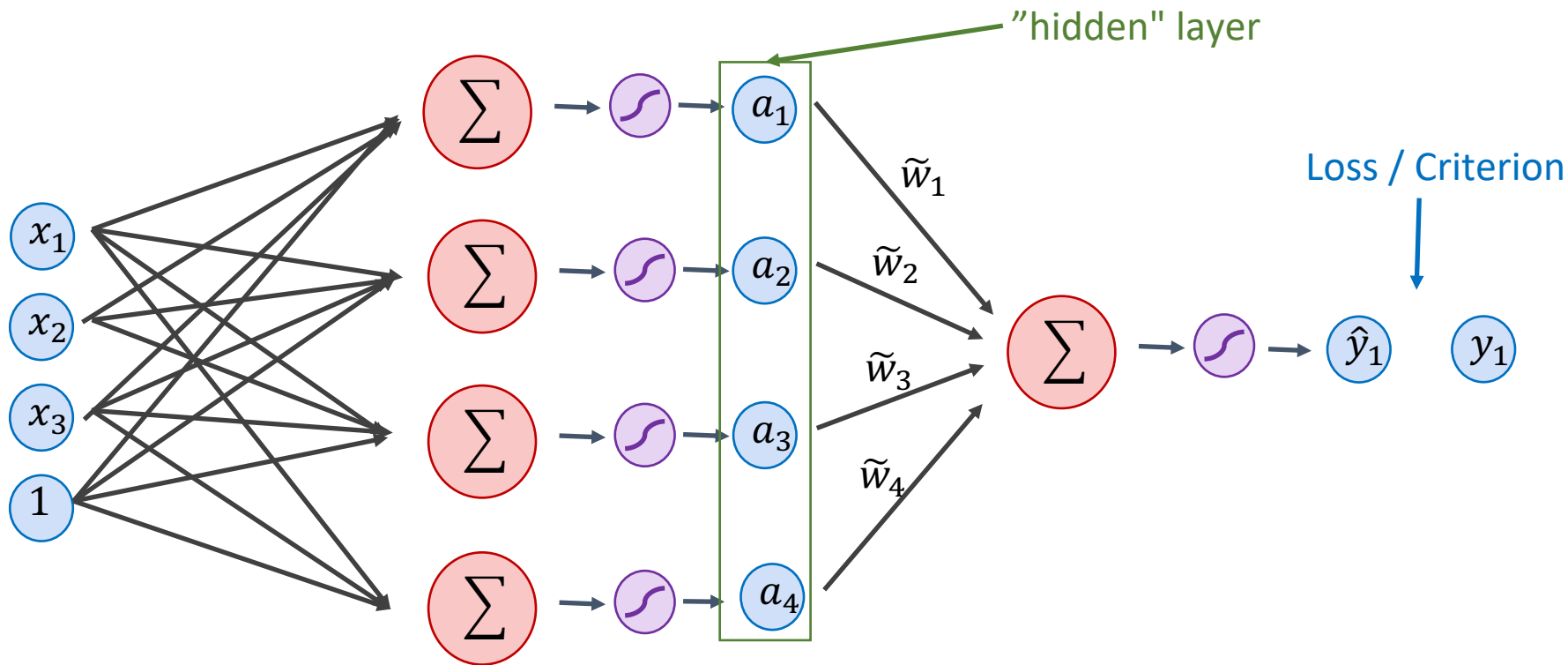
Rectified Linear Unit



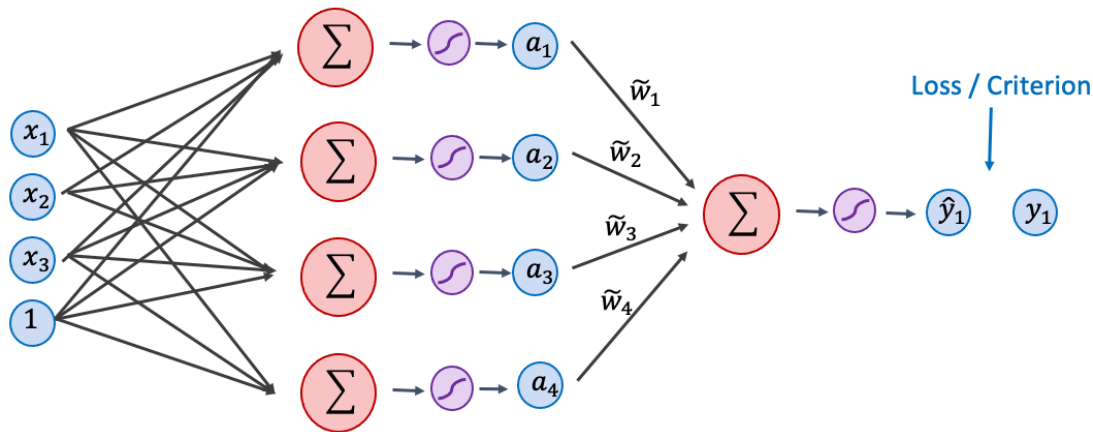
$$\text{ReLU}(x) = \begin{cases} x, & x > 0 \\ 0, & \text{otherwise} \end{cases}$$



Two-Layer Perceptron



Forward Pass: Short Version



$$\tilde{y} = \tilde{W} \cdot \text{activation}(W \cdot X^T)$$

$$\text{loss} = \text{some_loss_function}(y, \tilde{y})$$

Concrete Version: Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$g_c = w_{c1}x_{i1} + w_{c2}x_{i2} + w_{c3}x_{i3} + w_{c4}x_{i4} + b_c$$

$$g_d = w_{d1}x_{i1} + w_{d2}x_{i2} + w_{d3}x_{i3} + w_{d4}x_{i4} + b_d$$

$$g_b = w_{b1}x_{i1} + w_{b2}x_{i2} + w_{b3}x_{i3} + w_{b4}x_{i4} + b_b$$

$$f_c = e^{g_c} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_d = e^{g_d} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_b = e^{g_b} / (e^{g_c} + e^{g_d} + e^{g_b})$$

Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$g_c = w_{c1}x_{i1} + w_{c2}x_{i2} + w_{c3}x_{i3} + w_{c4}x_{i4} + b_c$$

$$g_d = w_{d1}x_{i1} + w_{d2}x_{i2} + w_{d3}x_{i3} + w_{d4}x_{i4} + b_d$$

$$g_b = w_{b1}x_{i1} + w_{b2}x_{i2} + w_{b3}x_{i3} + w_{b4}x_{i4} + b_b$$

$$W = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{d1} & w_{d2} & w_{d3} & w_{d4} \\ w_{b1} & w_{b2} & w_{b3} & w_{b4} \end{bmatrix}$$

$$b = [b_c \ b_d \ b_b]$$

$$f_c = e^{g_c} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_d = e^{g_d} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_b = e^{g_b} / (e^{g_c} + e^{g_d} + e^{g_b})$$

Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$g = wx^T + b^T$$

$$w = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{d1} & w_{d2} & w_{d3} & w_{d4} \\ w_{b1} & w_{b2} & w_{b3} & w_{b4} \end{bmatrix}$$

$$b = [b_c \ b_d \ b_b]$$

$$f_c = e^{g_c} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_d = e^{g_d} / (e^{g_c} + e^{g_d} + e^{g_b})$$

$$f_b = e^{g_b} / (e^{g_c} + e^{g_d} + e^{g_b})$$

Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_d \ f_b]$$

$$g = wx^T + b^T$$

$$w = \begin{bmatrix} w_{c1} & w_{c2} & w_{c3} & w_{c4} \\ w_{d1} & w_{d2} & w_{d3} & w_{d4} \\ w_{b1} & w_{b2} & w_{b3} & w_{b4} \end{bmatrix}$$

$$b = [b_c \ b_d \ b_b]$$

$$f = \text{softmax}(g)$$

Linear Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_a \ f_b]$$

$$f = \text{softmax}(wx^T + b^T)$$

Two-Layer Perceptron + Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_a \ f_b]$$

$$\mathbf{a}_1 = \text{sigmoid}(w_{[1]}x^T + b_{[1]}^T)$$

$$f = \text{softmax}(w_{[2]}\mathbf{a}_1^T + b_{[2]}^T)$$

N-layer Perceptron (MLP) + Softmax

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_a \ f_b]$$

$$\mathbf{a}_1 = \text{sigmoid}(w_{[1]}x^T + b_{[1]}^T)$$

$$a_2 = \text{sigmoid}(w_{[2]}\mathbf{a}_1^T + b_{[2]}^T)$$

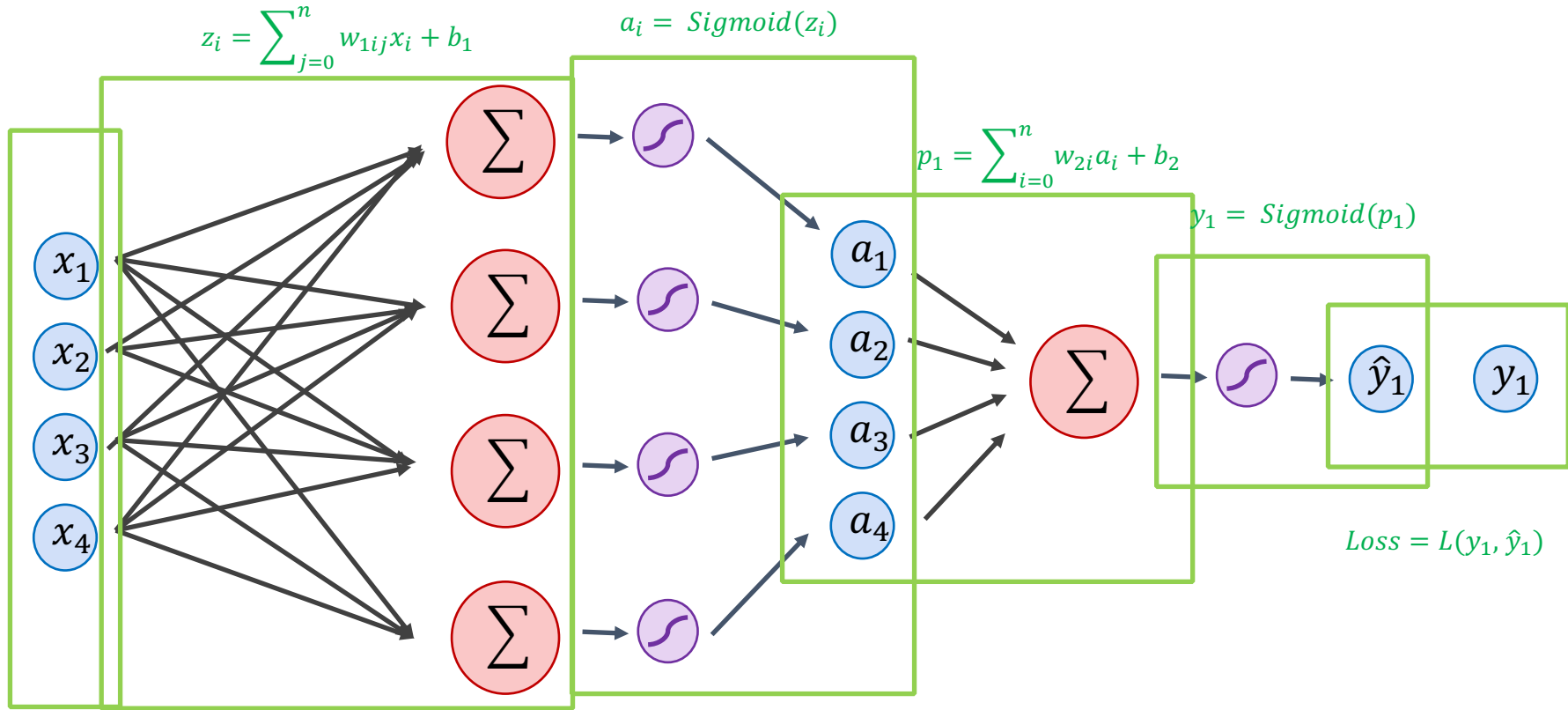
...

$$a_k = \text{sigmoid}(w_{[k]}a_{k-1}^T + b_{[k]}^T)$$

...

$$f = \text{softmax}(w_{[n]}a_{n-1}^T + b_{[n]}^T)$$

Forward pass (Forward-propagation)



How to train the parameters?

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_a \ f_b]$$

$$a_1 = \text{sigmoid}(\mathbf{w}_{[1]}x^T + \mathbf{b}_{[1]}^T)$$

$$a_2 = \text{sigmoid}(\mathbf{w}_{[2]}a_1^T + \mathbf{b}_{[2]}^T)$$

...

$$a_k = \text{sigmoid}(\mathbf{w}_{[k]}a_{k-1}^T + \mathbf{b}_{[k]}^T)$$

...

$$f = \text{softmax}(\mathbf{w}_{[n]}a_{n-1}^T + \mathbf{b}_{[n]}^T)$$

$$l = \text{loss}(f, y)$$

How to find
the optimal \mathbf{W} , \mathbf{b}

How to train the parameters?

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_a \ f_b]$$

$$a_1 = \text{sigmoid}(\mathbf{w}_{[1]}x^T + \mathbf{b}_{[1]}^T)$$

$$a_2 = \text{sigmoid}(\mathbf{w}_{[2]}a_1^T + \mathbf{b}_{[2]}^T)$$

...

$$a_k = \text{sigmoid}(\mathbf{w}_{[k]}a_{k-1}^T + \mathbf{b}_{[k]}^T)$$

...

$$f = \text{softmax}(\mathbf{w}_{[n]}a_{n-1}^T + \mathbf{b}_{[n]}^T)$$

$$l = \text{loss}(f, y)$$

SGD, but we need the
gradients of **W, b**
(millions of parameters)

$$\frac{\partial l}{\partial w_{[k]ij}} \quad \frac{\partial l}{\partial b_{[k]i}}$$

How to train the parameters?

$$x_i = [x_{i1} \ x_{i2} \ x_{i3} \ x_{i4}]$$

$$y_i = [1 \ 0 \ 0]$$

$$\hat{y}_i = [f_c \ f_a \ f_b]$$

$$a_1 = \text{sigmoid}(\mathbf{w}_{[1]}x^T + \mathbf{b}_{[1]}^T)$$

$$a_2 = \text{sigmoid}(\mathbf{w}_{[2]}a_1^T + \mathbf{b}_{[2]}^T)$$

...

$$a_k = \text{sigmoid}(\mathbf{w}_{[k]}a_{k-1}^T + \mathbf{b}_{[k]}^T)$$

...

$$f = \text{softmax}(\mathbf{w}_{[n]}a_{n-1}^T + \mathbf{b}_{[n]}^T)$$

$$l = \text{loss}(f, y)$$

Chain rule +
Reusing the Gradients

$$\frac{\partial l}{\partial w_{[k]ij}} = \frac{\partial l}{\partial a_{n-1}} \frac{\partial a_{n-1}}{\partial a_{n-2}} \cdots \frac{\partial a_k}{\partial a_{k-1}} \frac{\partial a_{k-1}}{\partial w_{[k]ij}}$$

Back-propagation: Application of Chain Rule



Geoffrey Hinton

[FOLLOW](#)

Emeritus Prof. Comp Sci, U.Toronto & Engineering Fellow, Google

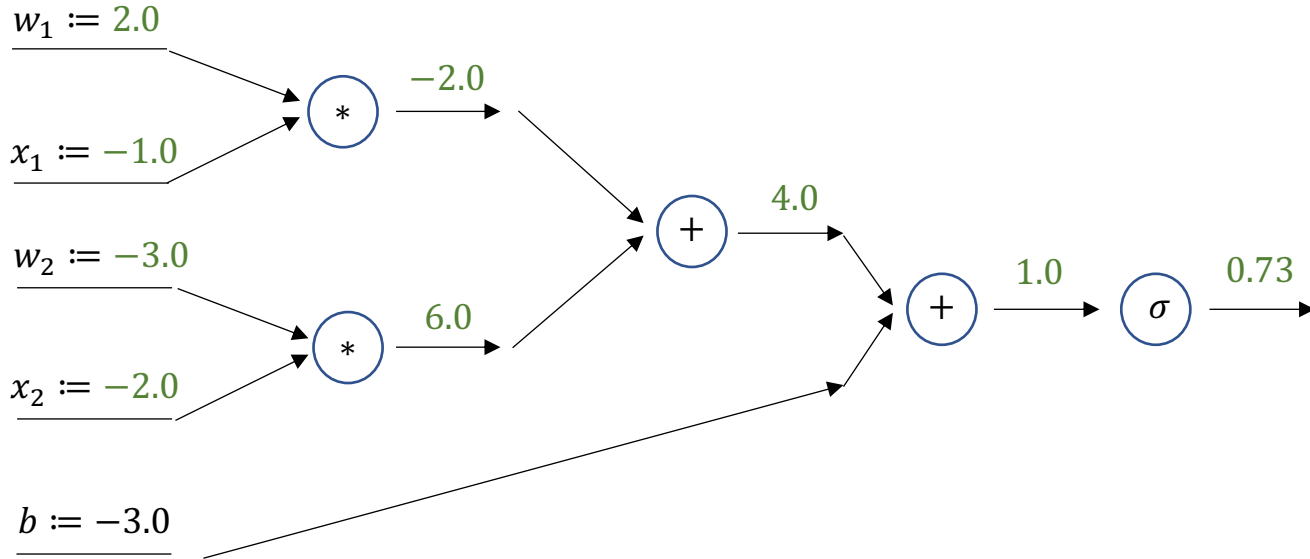
Verified email at cs.toronto.edu - [Homepage](#)

[machine learning](#) [psychology](#) [artificial intelligence](#) [cognitive science](#) [computer science](#)

TITLE	CITED BY	YEAR
Imagenet classification with deep convolutional neural networks A Krizhevsky, I Sutskever, GE Hinton Advances in neural information processing systems 25, 1097-1105	83557	2012
Deep learning Y LeCun, Y Bengio, G Hinton Nature 521 (7553), 436-444	37441	2015
Learning internal representations by error-propagation DE Rumelhart, GE Hinton, RJ Williams Parallel Distributed Processing: Explorations in the Microstructure of ...	27681	1986
Learning internal representations by error propagation DE Rumelhart, GE Hinton, RJ Williams Learning internal representations by error propagation	27616	1986
Learning internal representations by error propagation DE Rumelhart, GE Hinton, RJ Williams MIT Press, Cambridge, MA 1 (318)	27419	1986
Dropout: a simple way to prevent neural networks from overfitting N Srivastava, G Hinton, A Krizhevsky, I Sutskever, R Salakhutdinov The journal of machine learning research 15 (1), 1929-1958	27063	2014
Learning representations by back-propagating errors DE Rumelhart, GE Hinton, RJ Williams Nature 323 (6088), 533-536	24695	1986

Back-propagation: Example [*]

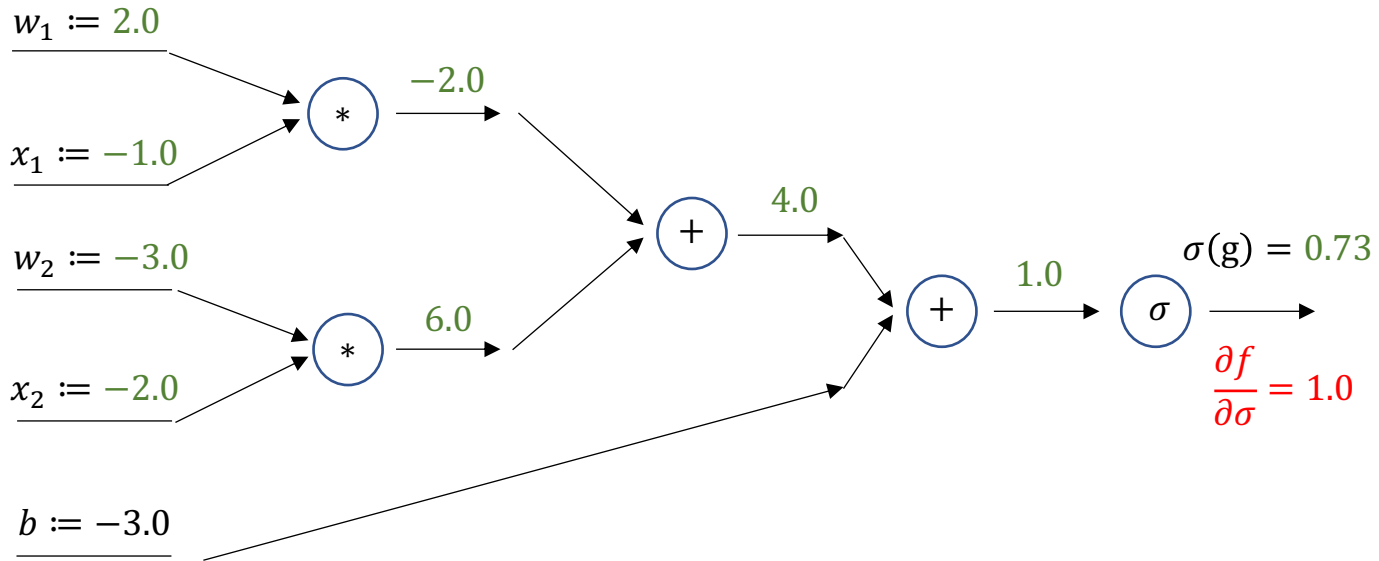
$$f(x) = \text{sigmoid}(wx^T + b) = \frac{1}{1 + e^{-(w_1x_1 + w_2x_2 + b)}}$$



[*] borrowed from http://cs231n.stanford.edu/slides/2016/winter1516_lecture4.pdf

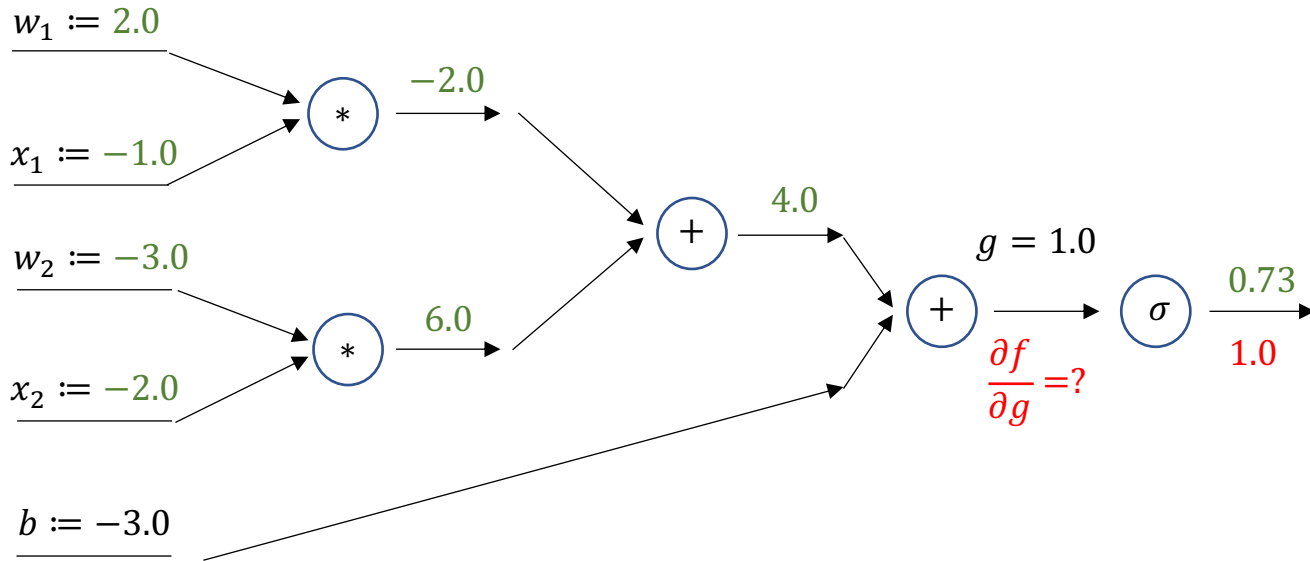
Back-propagation: Example [*]

$$f(x) = \sigma(g) = \frac{1}{1 + e^{-g}}, \quad g = z + b, \quad z = wx^T = w_1x_1 + w_2x_2$$



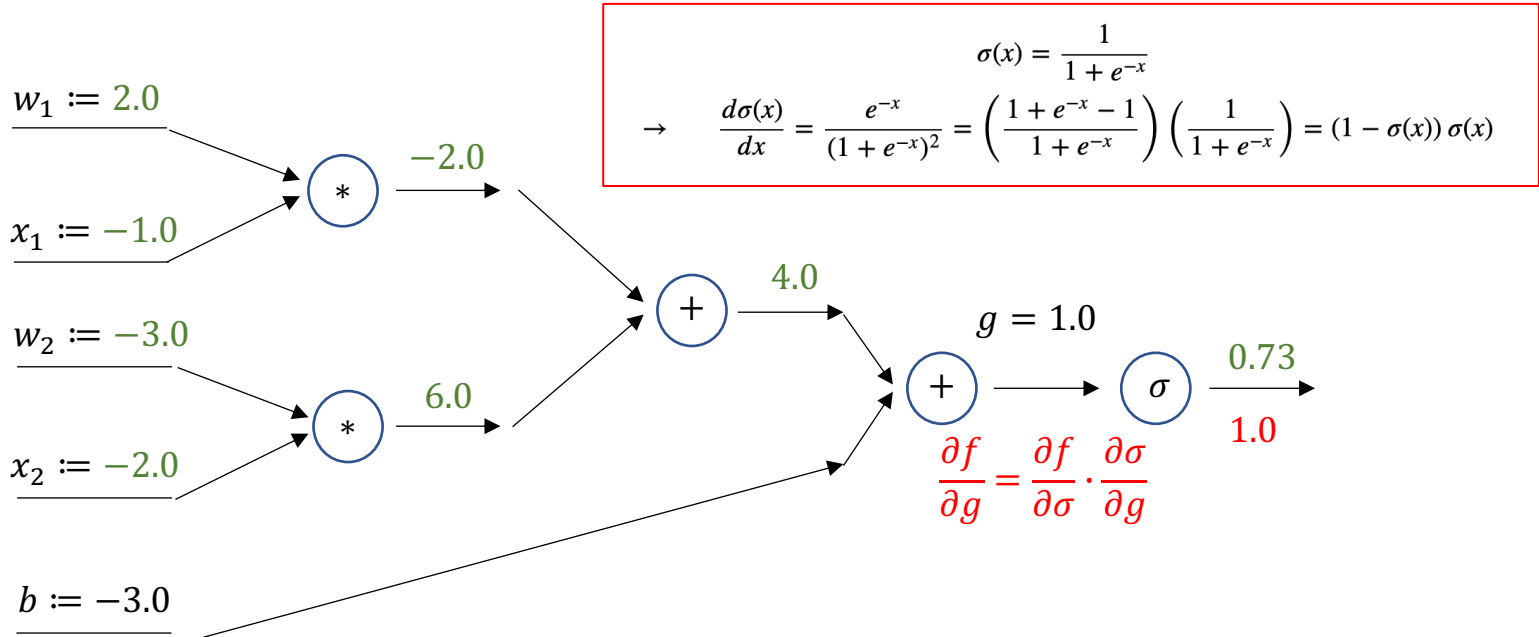
Back-propagation: Example [*]

$$f(x) = \sigma(g) = \frac{1}{1 + e^{-g}}, \quad g = z + b, \quad z = wx^T = w_1x_1 + w_2x_2$$



Back-propagation: Example [*]

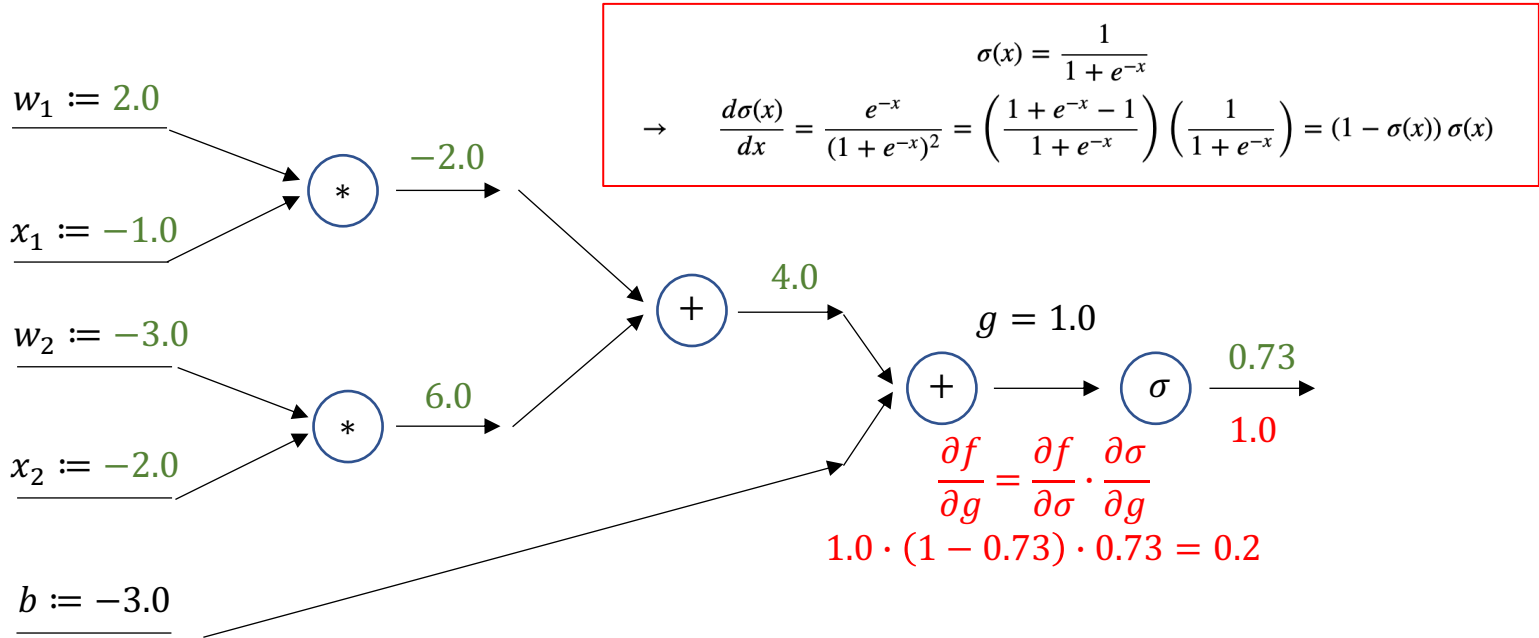
$$f(x) = \sigma(g) = \frac{1}{1 + e^{-g}}, \quad g = z + b, \quad z = wx^T = w_1x_1 + w_2x_2$$



[*] borrowed from http://cs231n.stanford.edu/slides/2016/winter1516_lecture4.pdf

Back-propagation: Example [*]

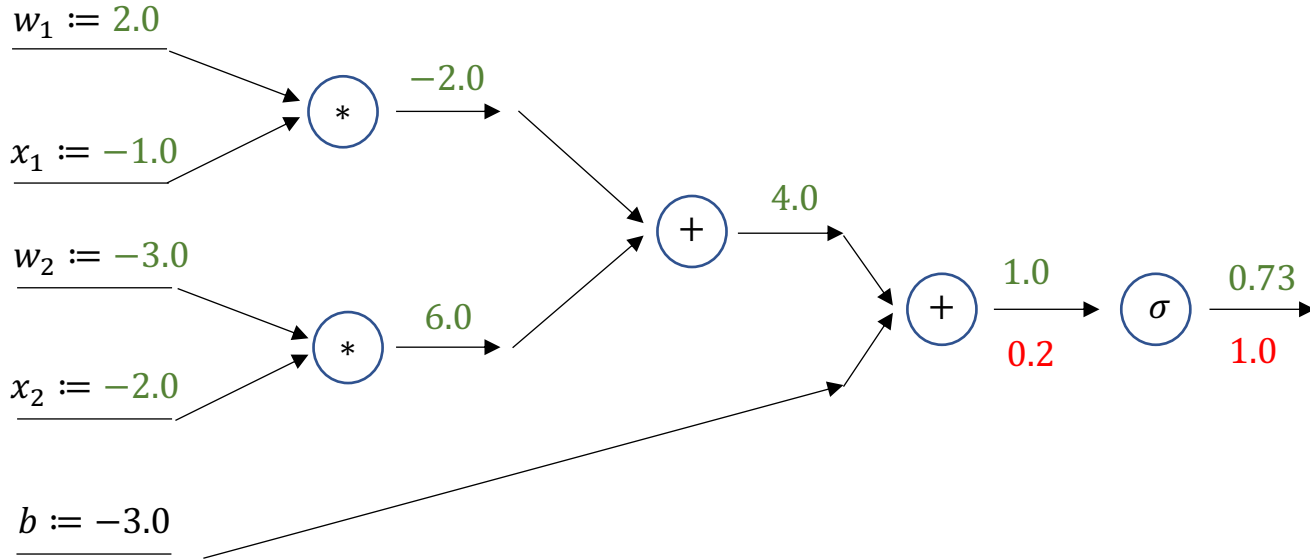
$$f(x) = \sigma(g) = \frac{1}{1 + e^{-g}}, \quad g = z + b, \quad z = wx^T = w_1x_1 + w_2x_2$$



[*] borrowed from http://cs231n.stanford.edu/slides/2016/winter1516_lecture4.pdf

Back-propagation: Example [*]

$$f(x) = \sigma(g) = \frac{1}{1 + e^{-g}}, \quad g = z + b, \quad z = wx^T = w_1x_1 + w_2x_2$$

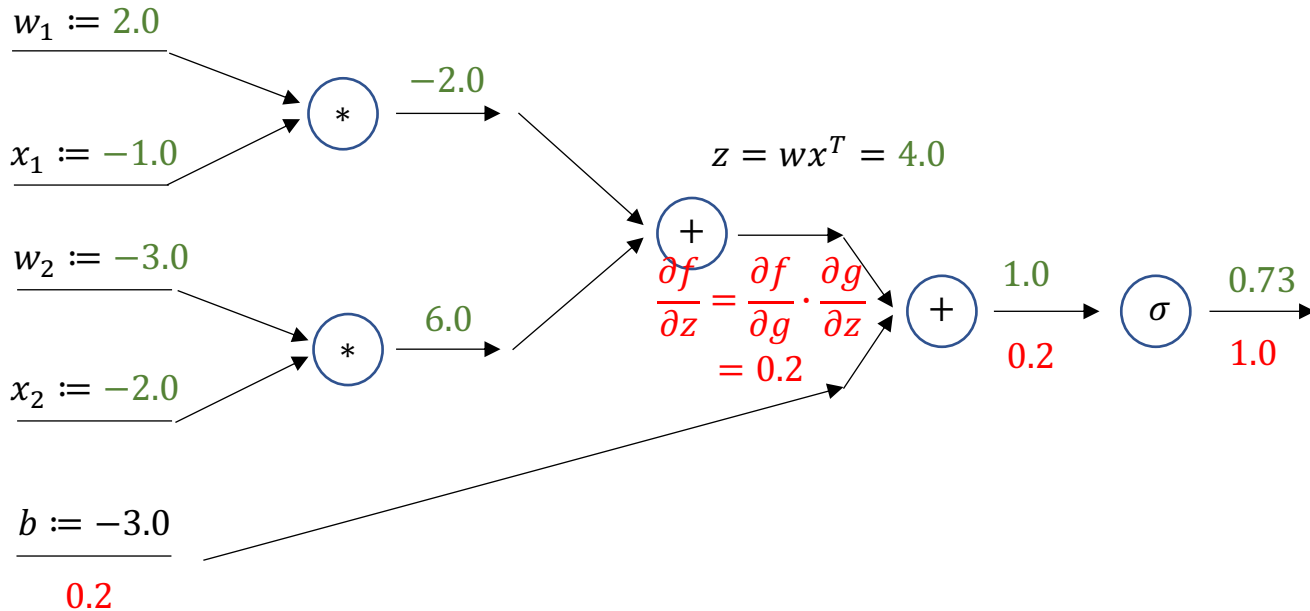


$$\frac{\partial f}{\partial b} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial b} = 0.2 \cdot 1.0$$

[*] borrowed from http://cs231n.stanford.edu/slides/2016/winter1516_lecture4.pdf

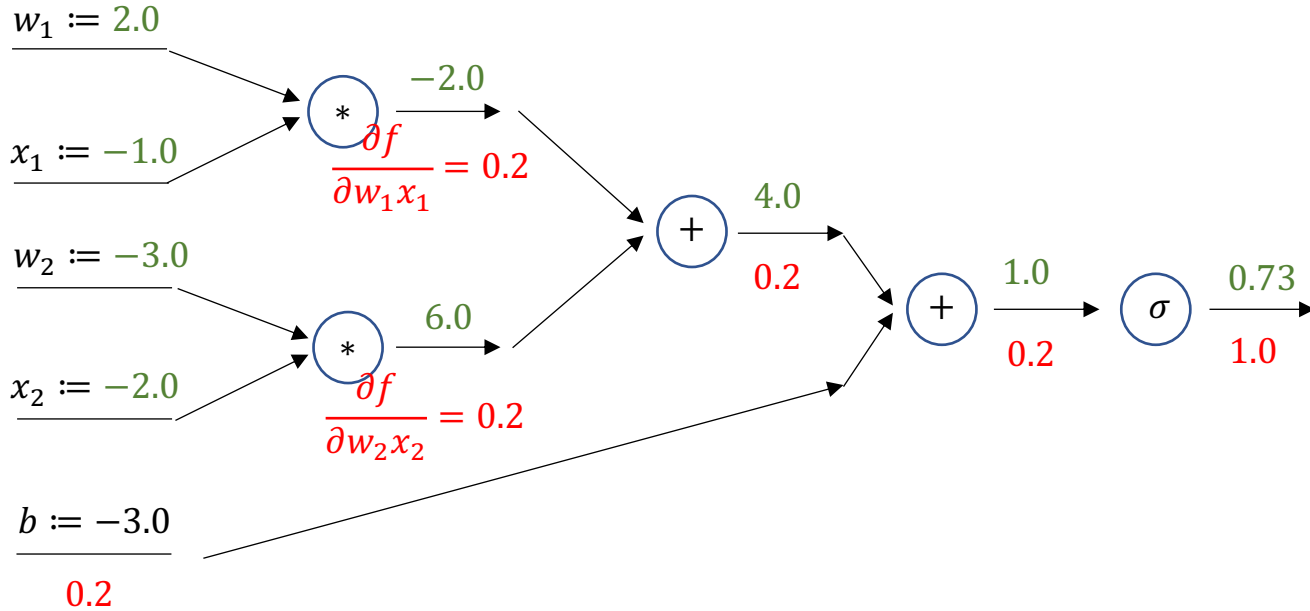
Back-propagation: Example [*]

$$f(x) = \sigma(g) = \frac{1}{1 + e^{-g}}, \quad g = z + b, \quad z = wx^T = w_1x_1 + w_2x_2$$



Back-propagation: Example [*]

$$f(x) = \sigma(g) = \frac{1}{1 + e^{-g}}, \quad g = z + b, \quad z = wx^T = w_1x_1 + w_2x_2$$

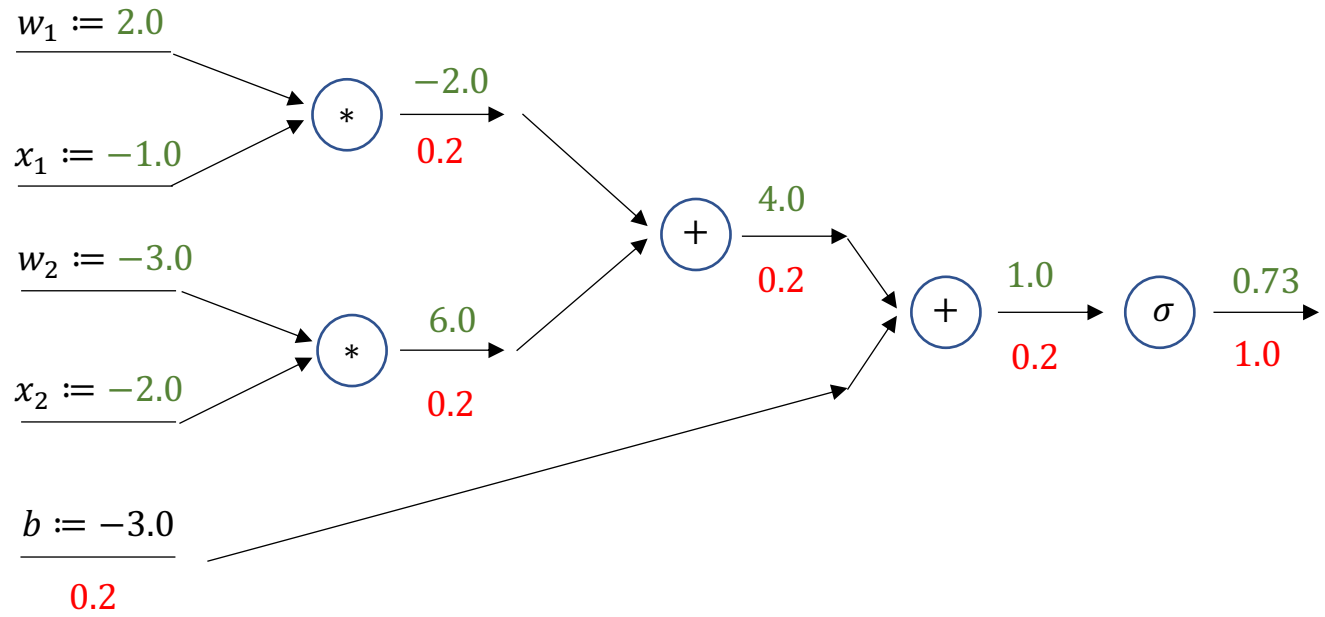


[*] borrowed from http://cs231n.stanford.edu/slides/2016/winter1516_lecture4.pdf

Back-propagation: Example [*]

$$f(x) = \sigma(g) = \frac{1}{1 + e^{-g}}, \quad g = z + b, \quad z = wx^T = w_1x_1 + w_2x_2$$

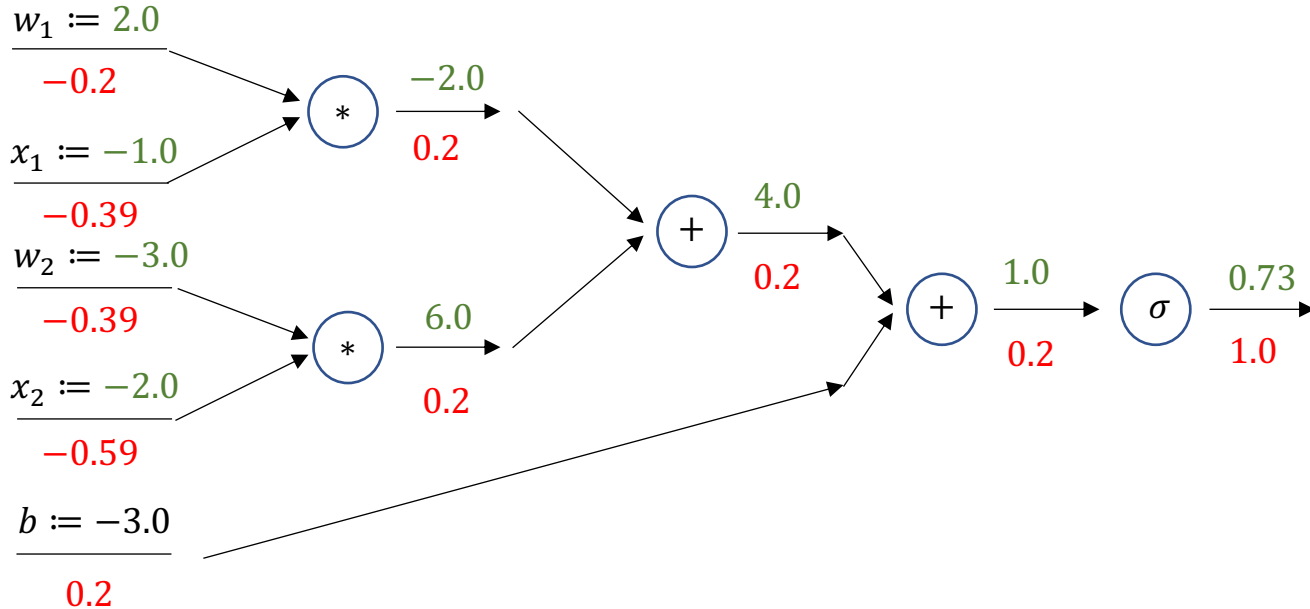
$$\frac{\partial f}{\partial w_1} = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial w_1} = 0.2 \cdot x_1 = -0.2$$



[*] borrowed from http://cs231n.stanford.edu/slides/2016/winter1516_lecture4.pdf

Back-propagation: Example [*]

$$f(x) = \sigma(g) = \frac{1}{1 + e^{-g}}, \quad g = z + b, \quad z = wx^T = w_1x_1 + w_2x_2$$



[*] borrowed from http://cs231n.stanford.edu/slides/2016/winter1516_lecture4.pdf

(mini-batch) Stochastic Gradient Descent (SGD)

$\lambda = 0.01$

Initialize w and b randomly

for $e = 0, \text{num_epochs}$ **do**

for $b = 0, \text{num_batches}$ **do**

Compute: $dl(w, b)/dw$ and $dl(w, b)/db$

Update w : $w = w - \lambda dl(w, b)/dw$

Update b : $b = b - \lambda dl(w, b)/db$

Print: $l(w, b)$ // Useful to see if this is becoming smaller or not.

end

end

$$l(w, b) = \sum_{i \in B} -\log f_{i, \text{label}}(w, b)$$

For Softmax Classifier

Automatic Differentiation

You only need to write code for the forward pass,
backward pass is computed automatically.

Frameworks such as Pytorch will “record” the operations performed on
tensors and compute gradients through the “recorded”
operations when requested.

Pytorch (Facebook -- mostly):

<https://pytorch.org/>

Tensorflow (Google -- mostly):

<https://www.tensorflow.org/>

DyNet (team includes UVA Prof. Yangfeng Ji):

<http://dynet.io/>

Example

- Provided in Assignment 4.

Defining a Linear Softmax classifier

```
import torch
from torch import nn

# This is the softmax classifier as studied in class.
class Classifier(nn.Module):

    def __init__(self):
        super(Classifier, self).__init__()
        # Linear transformation layer.
        # This computes  $a = wx + b$  where:
        # a is a vector of size 10
        # x: is a vector of size 3 x 32 x 32
        # b: is a vector of size 10
        # w: is a matrix of size 10 x (3 * 32 * 32)
        self.linear = nn.Linear(3 * 32 * 32, 10)

        # Softmax operator.
        # This is  $\log(\exp(a_i) / \sum(a))$ 
        self.log_softmax = nn.LogSoftmax(dim = 1)

    def forward(self, x):
        gx = self.linear(x)
        yhat = self.log_softmax(gx)
        return yhat
```

Defining a Linear Softmax classifier

```
import torch
from torch import nn

# This is the softmax classifier as studied in class.
class Classifier(nn.Module):
```

Parent class

```
    def __init__(self):
        super(Classifier, self).__init__()
        # Linear transformation layer.
        # This computes  $a = wx + b$  where:
        # a is a vector of size 10
        # x: is a vector of size 3 x 32 x 32
        # b: is a vector of size 10
        # w: is a matrix of size 10 x (3 * 32 * 32)
        self.linear = nn.Linear(3 * 32 * 32, 10)

        # Softmax operator.
        # This is  $\log(\exp(a_i) / \sum(a))$ 
        self.log_softmax = nn.LogSoftmax(dim = 1)

    def forward(self, x):
        gx = self.linear(x)
        yhat = self.log_softmax(gx)
        return yhat
```

Defining a Linear Softmax classifier

```
import torch
from torch import nn

# This is the softmax classifier as studied in class.
class Classifier(nn.Module):

    def __init__(self):
        super(Classifier, self).__init__()
        # Linear transformation layer.
        # This computes  $a = wx + b$  where:
        # a is a vector of size 10
        # x: is a vector of size 3 x 32 x 32
        # b: is a vector of size 10
        # w: is a matrix of size 10 x (3 * 32 * 32)
        self.linear = nn.Linear(3 * 32 * 32, 10)

        # Softmax operator.
        # This is  $\log(\exp(a_i) / \sum(a))$ 
        self.log_softmax = nn.LogSoftmax(dim = 1)

    def forward(self, x):
        gx = self.linear(x)
        yhat = self.log_softmax(gx)
        return yhat
```

Linear layer



Defining a Linear Softmax classifier

```
import torch
from torch import nn

# This is the softmax classifier as studied in class.
class Classifier(nn.Module):

    def __init__(self):
        super(Classifier, self).__init__()
        # Linear transformation layer.
        # This computes  $a = wx + b$  where:
        # a is a vector of size 10
        # x: is a vector of size 3 x 32 x 32
        # b: is a vector of size 10
        # w: is a matrix of size 10 x (3 * 32 * 32)
        self.linear = nn.Linear(3 * 32 * 32, 10)

        # Softmax operator.
        # This is  $\log(\exp(a_i) / \sum(a))$ 
        self.log_softmax = nn.LogSoftmax(dim = 1)

    def forward(self, x):
        gx = self.linear(x)
        yhat = self.log_softmax(gx)
        return yhat
```

Activation function



Defining a Linear Softmax classifier

```
import torch
from torch import nn

# This is the softmax classifier as studied in class.
class Classifier(nn.Module):

    def __init__(self):
        super(Classifier, self).__init__()
        # Linear transformation layer.
        # This computes  $a = wx + b$  where:
        # a is a vector of size 10
        # x: is a vector of size 3 x 32 x 32
        # b: is a vector of size 10
        # w: is a matrix of size 10 x (3 * 32 * 32)
        self.linear = nn.Linear(3 * 32 * 32, 10)

        # Softmax operator.
        # This is  $\log(\exp(a_i) / \sum(a))$ 
        self.log_softmax = nn.LogSoftmax(dim = 1)

    def forward(self, x):
        gx = self.linear(x)
        yhat = self.log_softmax(gx)
        return yhat
```

Forward pass



Using a Linear Softmax classifier

```
# Now let's try using it on a batch 5 of images represented  
# as 5 vectors of size 3 * 32 * 32.  
dummy_input = torch.rand(5, 3 * 32 * 32)  
out = classifier(dummy_input)  
print(out.exp())
```

Training a Linear Softmax classifier

```
classifier = Classifier()
criterion = nn.NLLLoss()

num_epochs = 30
learningRate = 0.001

classifier.train()

weight = classifier.linear.weight;
bias = classifier.linear.bias;

for epoch in range(0, num_epochs):
    for (i, (x, y)) in enumerate(trainLoader):

        x = x.view(x.shape[0], 3 * 32 * 32)
        yhat = classifier(x)
        loss = criterion(yhat, y)

        loss.backward()

        weight.data.add_(-learningRate * weight.grad.data)
        bias.data.add_(-learningRate * bias.grad.data)

    print(loss.item())
```

What is trainLoader?

```
batch_size = 128

# It additionally has utilities for threaded and multi-parallel data loading.
trainLoader = DataLoader(train_data, batch_size = batch_size,
                          shuffle = True, num_workers = 0)
valLoader = DataLoader(validation_data, batch_size = batch_size,
                       shuffle = False, num_workers = 0)

# Look-up python iterators if you need:
# https://anandology.com/python-practice-book/iterators.html
x, y = iter(trainLoader).next()
print('batch-of-images: ', x.shape)
print('batch-of-labels: ', y.shape)
```

Training a Linear Softmax classifier (improved)

```
classifier = Classifier()
criterion = nn.NLLLoss()

num_epochs = 30
optimizer = torch.optim.SGD(classifier.parameters(), lr = 0.001,
                             momentum = 0.9)

classifier.train()

weight = classifier.linear.weight;
bias = classifier.linear.bias;

for epoch in range(0, num_epochs):
    for (i, (x, y)) in enumerate(trainLoader):

        x = x.view(x.shape[0], 3 * 32 * 32)
        yhat = classifier(x)
        loss = criterion(yhat, y)

        loss.backward()

        optimizer.zero_grad()
        optimizer.step()

    print(loss.item())
```

This depends on the model
but we don't need it
anymore

Defining a Two-layer Neural Network

```
import torch
from torch import nn

# This is the softmax classifier as studied in class.
class Classifier(nn.Module):

    def __init__(self):
        super(Classifier, self).__init__()
        self.linear1 = nn.Linear(3 * 32 * 32, 512)
        self.sigmoid = nn.Sigmoid()
        self.linear2 = nn.Linear(512, 10)
        self.log_softmax = nn.LogSoftmax(dim = 1)

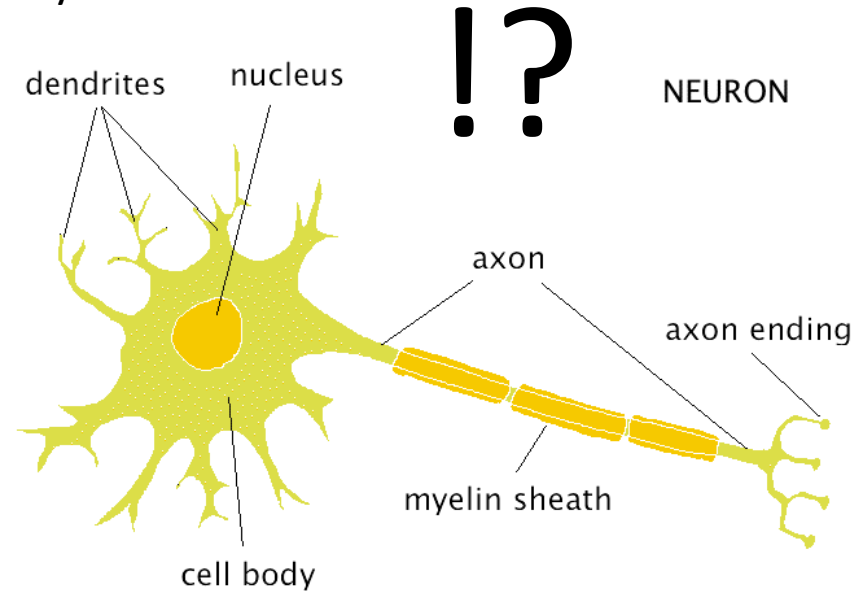
    def forward(self, x):
        gx = self.linear1(x)
        gxs = self.sigmoid(gx)
        mx = self.linear2(gxs)
        yhat = self.log_softmax(mx)
        return yhat
```

Questions?

Perceptron Model

Frank Rosenblatt (1957) - Cornell University

$$f(x) = \begin{cases} 1, & \text{if } \sum_{i=0}^n w_i x_i + b > 0 \\ 0, & \text{otherwise} \end{cases}$$



More: <https://en.wikipedia.org/wiki/Perceptron>

Backward pass (Back-propagation)

