CS310 Natural Language Processing
自然语言处理
Lecture 01 - Neural Networks for Text Classification

Instructor: Yang Xu
主讲人：徐炀
xuyang@sustech.edu.cn

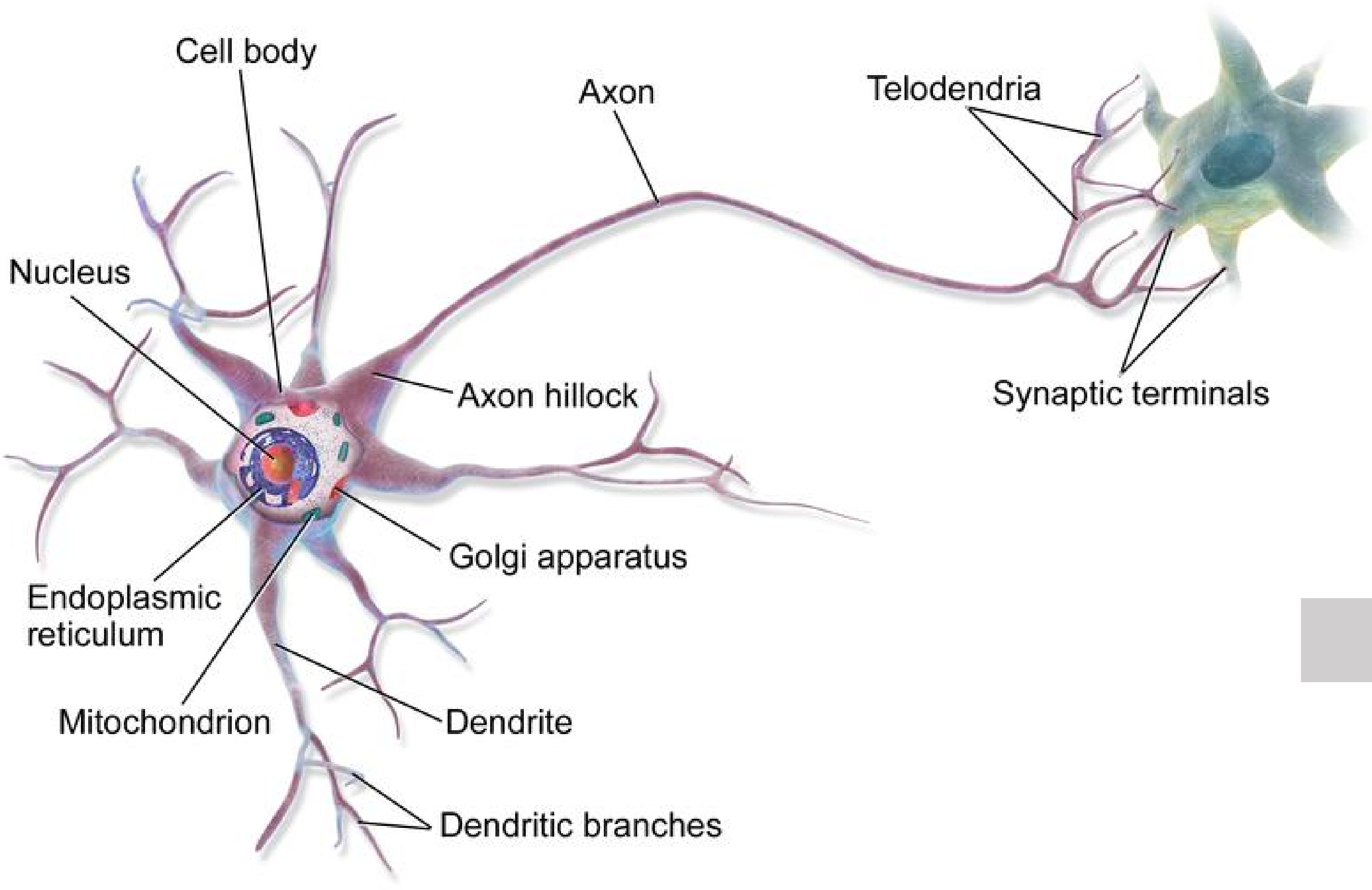
Content

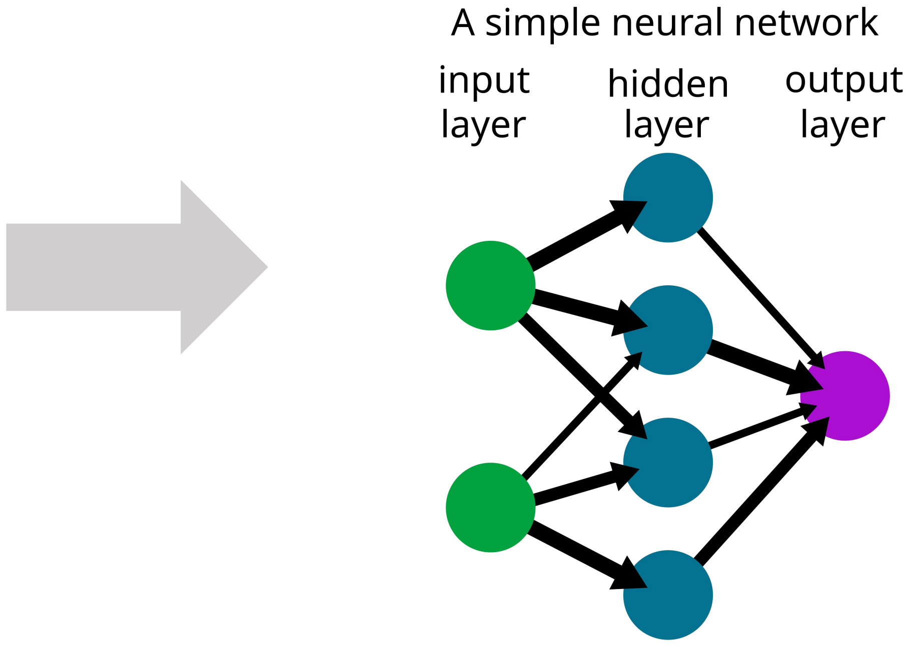
- Neural Networks
- Word Vectors
- Neural Text Classification

Content

- **Neural Networks**
  - **Logistic regression and gradient descent**
  - **Neural networks and back-propagation**
  - **PyTorch Implementation**
- Word Vectors
- Neural Text Classification

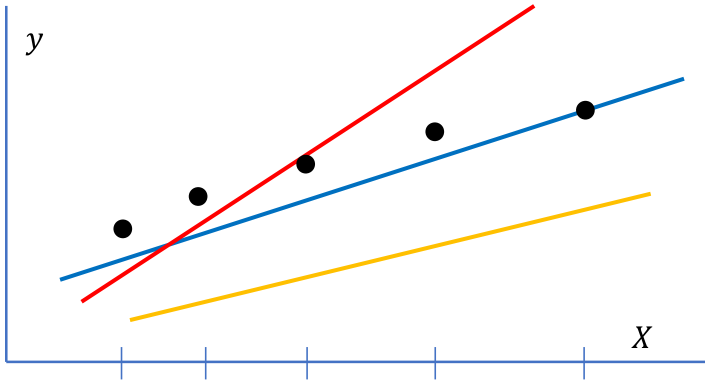
Neural Networks



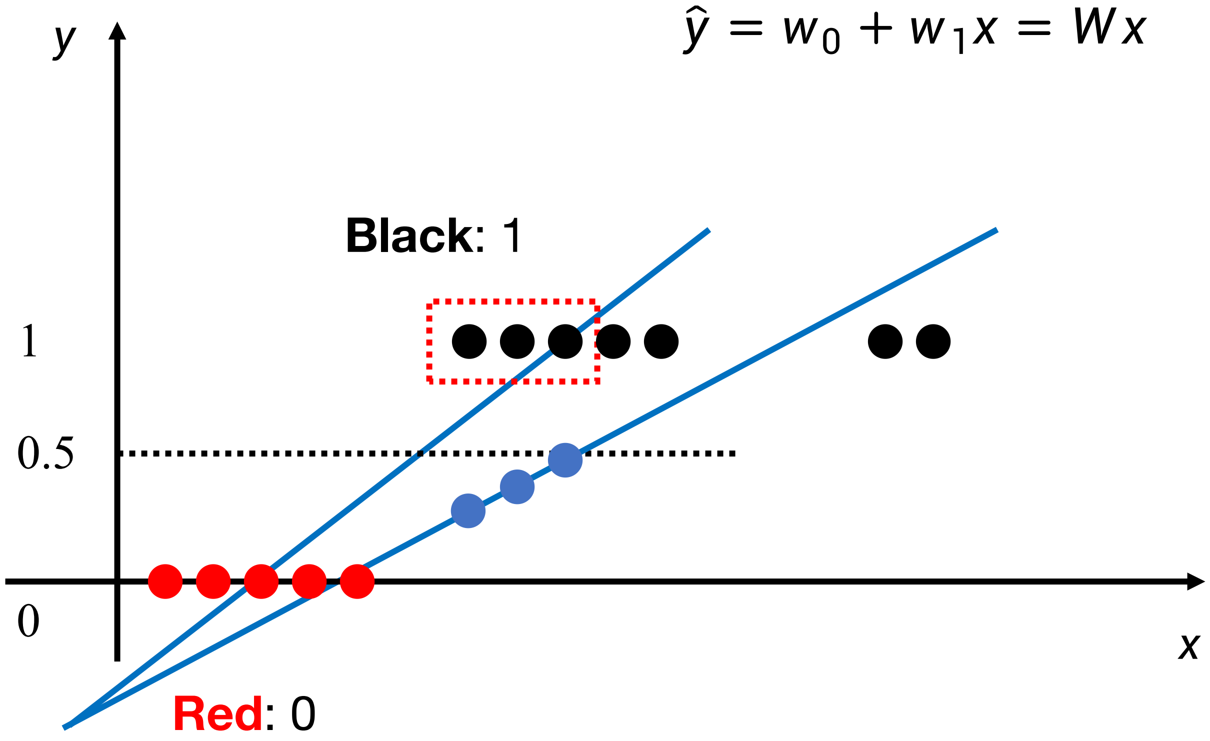


Logistic regression (binary classification)

Linear regregression $X \Rightarrow$ continuous y



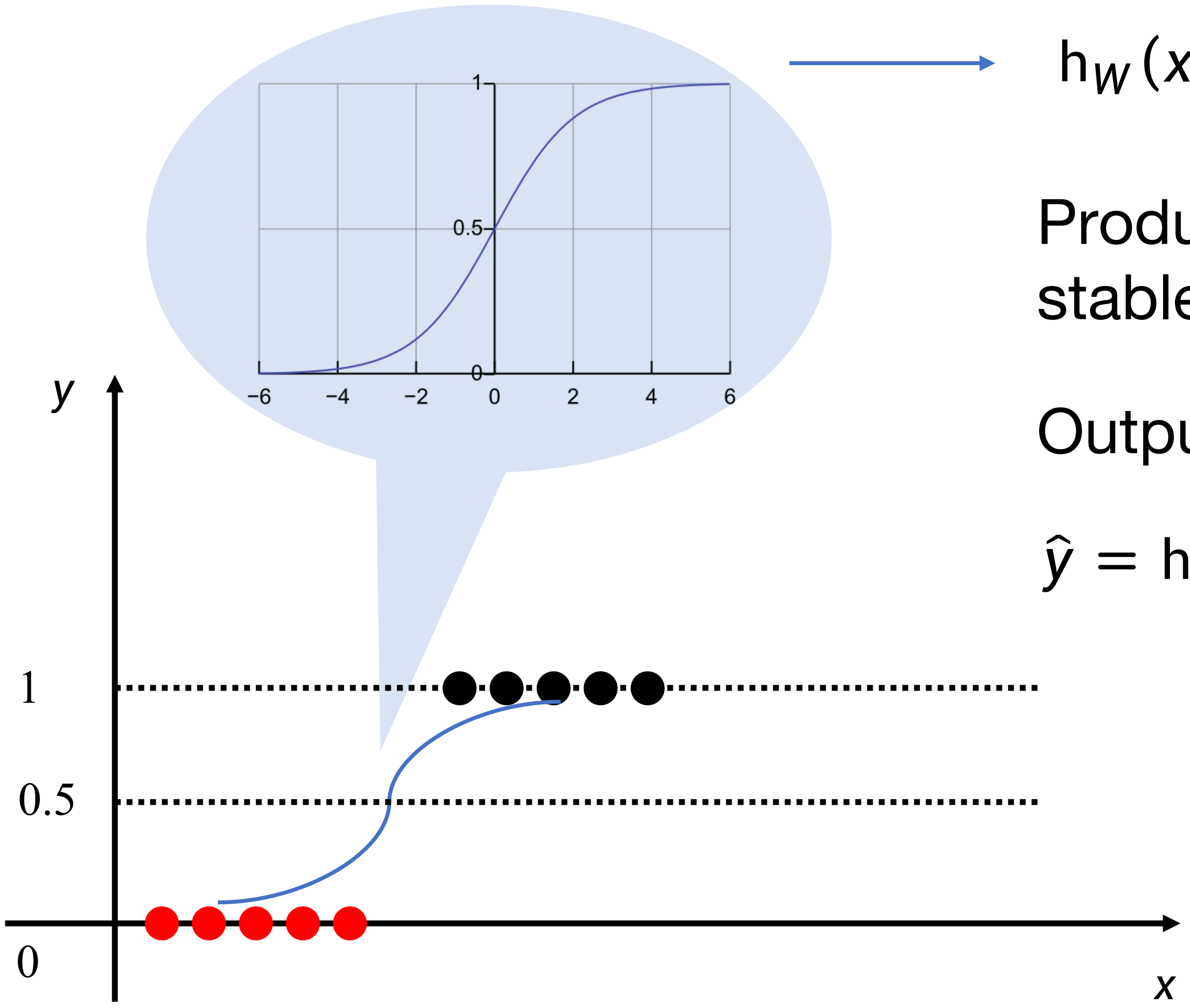
Linear function fails when y is binary, i.e., $y \in \{0,1\}$



Solution: Sigmoid function

$$\hat{y} = h_W(x) = \frac{1}{1 + e^{-Wx}}$$

Sigmoid function



$$h_W(x) = \frac{1}{1 + e^{-(wx+b)}}$$

Produces a smoother curve;
stable against outliers

Output $\hat{y}$ is bounded to [0, 1]

$\hat{y} = h_W(x)$ estimates the **probability** of x is class

$$\hat{y} = P(y = 1|x)$$

$$1 - \hat{y} = P(y = 0|x)$$

Question: How to learn the parameters w, b?

- Need to have an objective function to optimize
- **Likelihood**: $P(y=1|x) = \mathrm{h}_{w,b}(x)$, $P(y=0|x) = 1 - \mathrm{h}_{w,b}(x)$
- A compact way: $P(y|x) = (\mathrm{h}_{w,b}(x))^{y}(1-\mathrm{h}_{w,b}(x))^{1-y}$
- **Goal**: Learn the parameter W to maximize the likelihood of the given data, $\{(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \cdots, (x^{(m)}, y^{(m)})\}$

$$\mathbb{l}(w,b) = \prod_i (\mathrm{h}_{w,b}(x^{(i)}))^{y^{(i)}}(1-\mathrm{h}_{w,b}(x^{(i)}))^{1-y^{(i)}}$$

Hence, log-likelihood $\log(\mathbb{l}(w,b)) = \sum_i [y^{(i)} \log \mathrm{h}_{w,b}(x^{(i)}) + (1-y^{(i)}) \log(1-\mathrm{h}_{w,b}(x^{(i)}))]$

Log-Likelihood

Question: How to?

$\Rightarrow$ stochastic gradient descent

- To maximize log-likelihood

$$\mathtt{ll}(w,b) = \sum_i [y^{(i)} \log \mathrm{h}_{w,b}(x^{(i)}) + (1 - y^{(i)}) \log(1 - \mathrm{h}_{w,b}(x^{(i)}))]$$

- Equivalent to minimize **negative** log-likelihood

$$\mathcal{N}\mathtt{ll}(w,b) = -\sum_i [y^{(i)} \log \mathrm{h}_{w,b}(x^{(i)}) + (1 - y^{(i)}) \log(1 - \mathrm{h}_{w,b}(x^{(i)}))]$$

$$= -\sum_i [y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})]$$

only exists when $y^{(i)} = 1$, where $\hat{y}^{(i)} = P(y^{(i)} = 1|x^{(i)})$

only exists when $y^{(i)} = 0$, where $\hat{y}^{(i)} = P(y^{(i)} = 0|x^{(i)})$

Gradient descent (GD) algorithm

- Gradient points in the direction of the *fastest increase* of function
- ***Opposite*** of gradient points to the direction of ***fastest decrease***

- Goal: Minimize objective function $J(\theta)$

θ are parameters just like w, b

i. Start with $\theta^{(0)}$.

ii. Change $\theta^{(0)}$ a bit to $\theta^{(1)}$, so that $J(\theta^{(1)})$ decreases a bit from $J(\theta^{(0)})$.

$$\theta^{(1)} \leftarrow \theta^{(0)} - \alpha \frac{\partial J(\theta^{(0)})}{\partial \theta}$$

iii. Repeat until $J(\theta)$ no longer decreases significantly.

Gradients

- A function with 1 output and 1 input

$$f(x) = x^3$$

- Its gradient = its derivative

$$\frac{df}{dx} = 3x^2$$

- How much will the output change if we change the input a bit?
- $\frac{d}{dx} f(1) = 3$, output changes 3 times as much as input: $1.01^3 \approx 1.03$
- $\frac{d}{dx} f(2) = 12$, output changes 12 times as much as input: $2.01^3 \approx 8.12$

Gradients of multivariate functions

- A function of 1 output and n inputs

$$f(\boldsymbol{x}) = f(x_1, x_2, \ldots, x_n)$$

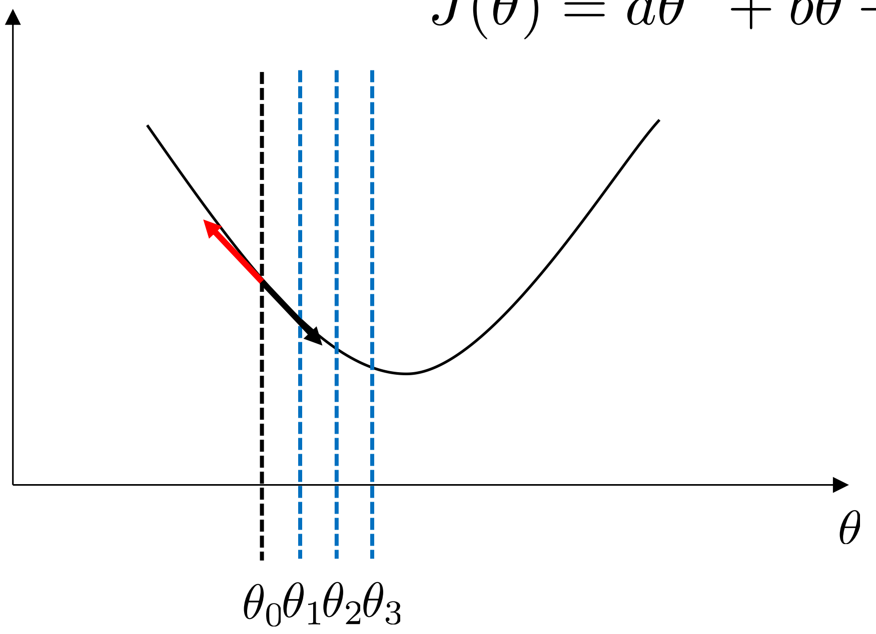
- The gradients is a vector of partial derivatives with respect to each input

$$\frac{\partial f}{\partial \boldsymbol{x}} = [\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n}]$$

1-D demo of GD

- How do we change θ to minimize $J(\theta)$

$$J(\theta) = a\theta^2 + b\theta + c$$



$$\frac{\partial J}{\partial \theta_0} < 0$$

Move towards the opposite direction of gradient by a step (determined by *learning rate* α):

$$\theta_1 = \theta_0 - \alpha \frac{\partial J}{\partial \theta_0}$$

Therefore, $\theta_1 > \theta_0$

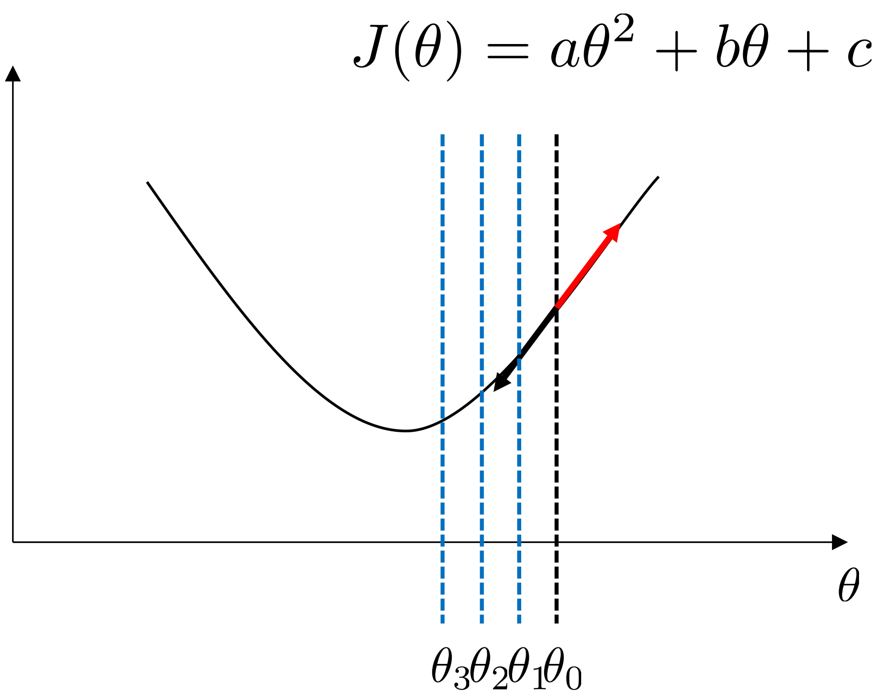
If α is small enough, θ_1 will be closer to minimum than θ_0

$$J(\theta_1) < J(\theta_0)$$

Repeat until a satisfactory θ_n is reached

1-D demo of GD (cont.)

- θ initialized to a different value



$$\frac{\partial J}{\partial \theta_0} > 0$$

Move towards the opposite direction of gradient:

$$\theta_1 = \theta_0 - \alpha \frac{\partial J}{\partial \theta_0}$$

Therefore $\theta_1 < \theta_0$

If α is small $J(\theta_1) < J(\theta_0)$

Repeat until satisfied.

The principle holds: **move along the opposite direction of gradient**

SOUTHERN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Gradient descent for logistic regression

- **Goal**: Minimize the **cost function** $J(W)$, i.e., the negative log-likelihood

$$J(W) = -\sum_i [y^{(i)} \log h_W(x^{(i)}) + (1 - y^{(i)}) \log(1 - h_W(x^{(i)}))]$$

- Need to compute the gradient of $J(W)$

$$\frac{\partial J}{\partial W} = \frac{\partial[-y^{(i)} \log h_W(x^{(i)}) - (1 - y^{(i)}) \log(1 - h_W(x^{(i)}))]}{\partial W}$$

(summation omitted)

$$= -\frac{y^{(i)}}{h_W(x^{(i)})} \frac{\partial h_W(x^{(i)})}{\partial W} - \frac{1 - y^{(i)}}{1 - h_W(x^{(i)})} \frac{-\partial h_W(x^{(i)})}{\partial W}$$

$$= -\frac{y^{(i)}}{\hat{y}^{(i)}} \frac{\partial h_W(x^{(i)})}{\partial W} - \frac{1 - y^{(i)}}{1 - \hat{y}^{(i)}} \frac{-\partial h_W(x^{(i)})}{\partial W}$$

Key: $\frac{\partial h_W(x^{(i)})}{\partial W}$

Gradient descent for logistic regression

$$h_W(x) = \frac{1}{1+e^{-Wx}}$$

let $z = Wx$, $g(z) = \frac{1}{1+e^{-z}}$, and then apply the chain rule of calculus

$$\frac{\partial h_W(x)}{\partial W} = \frac{\partial g}{\partial z}\frac{\partial z}{\partial W} = \frac{e^{-z}}{(1+e^{-z})^2}x = \frac{1}{1+e^{-z}}(1-\frac{1}{1+e^{-z}})x$$

$$\frac{\partial g}{\partial z} = \frac{\partial}{\partial z}(\frac{1}{1+e^{-z}}) = \frac{e^{-z}}{(1+e^{-z})^2}$$

$$\frac{\partial z}{\partial W} = x$$

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

$$= h_W(x)(1-h_W(x))x$$

$$= \hat{y}\,(1-\hat{y})x$$

Gradient descent for logistic regression

Plugin $\frac{\partial h_W(x^{(i)})}{\partial W} = \hat{y}^{(i)}\,(1-\hat{y}^{(i)})x^{(i)}$ to $\frac{\partial J(W)}{\partial W} = -\frac{y^{(i)}}{\hat{y}^{(i)}}\frac{\partial h_W(x^{(i)})}{\partial W} - \frac{1-y^{(i)}}{1-\hat{y}^{(i)}}\frac{-\partial h_W(x^{(i)})}{\partial W}$

$$\frac{\partial J(W)}{\partial W} = -\frac{y^{(i)}}{\hat{y}^{(i)}}\hat{y}^{(i)}\,(1-\hat{y}^{(i)})x^{(i)} - \frac{1-y^{(i)}}{1-\hat{y}^{(i)}}\hat{y}^{(i)}\,(1-\hat{y}^{(i)})x^{(i)} \quad = (\hat{y}^{(i)} - y^{(i)})x^{(i)}$$

Place back the summation and average over all samples:

$$\frac{\partial J(W)}{\partial W} = \frac{1}{m}\sum_i (\hat{y}^{(i)} - y^{(i)})x^{(i)}$$

Plugin to

Gradient descent

Initialize W

Repeat until satisfied {

Compute $\frac{\partial J(W)}{\partial W_t}$ using

Update $W_{t+1} = W_t - \alpha\frac{\partial J(W)}{\partial W_t}$

}

Computational graph

$z = w^T x + b$

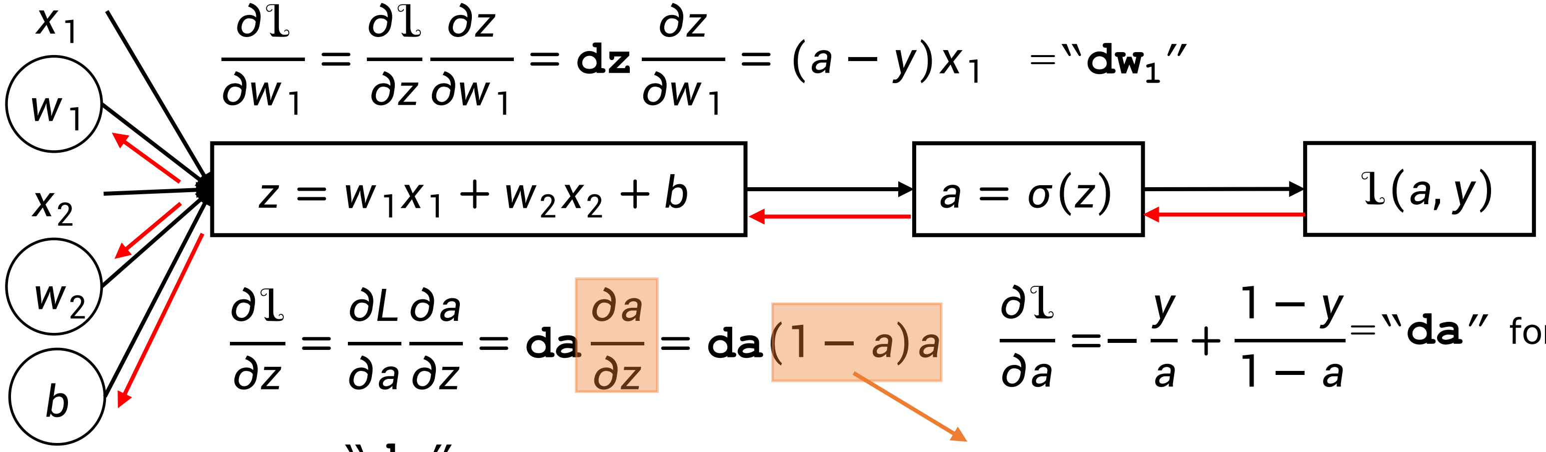
$\hat{y} = \underline{a} = \sigma(z) = \dfrac{1}{1+e^{-z}}$ $\quad a$ is for "activation"

$\underline{\mathfrak{l}(a,y)} = -\left(y\log(a) + (1-y)\log(1-a)\right)$

$\mathfrak{l}$ is for "loss", similar "cost"

Chain rule:

$$\frac{\partial \mathfrak{l}}{\partial c} = \frac{\partial \mathfrak{l}}{\partial a}\frac{\partial a}{\partial b}\frac{\partial b}{\partial c}$$



$$\frac{\partial \mathfrak{l}}{\partial w_1} = \frac{\partial \mathfrak{l}}{\partial z}\frac{\partial z}{\partial w_1} = \mathbf{dz}\frac{\partial z}{\partial w_1} = (a-y)x_1 \quad = \text{“}\mathbf{dw_1}\text{”}$$

$$\frac{\partial \mathfrak{l}}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z} = \mathbf{da}\boxed{\frac{\partial a}{\partial z}} = \mathbf{da}\boxed{(1-a)a}$$

$$= \text{“}\mathbf{dz}\text{”} \quad = a - y$$

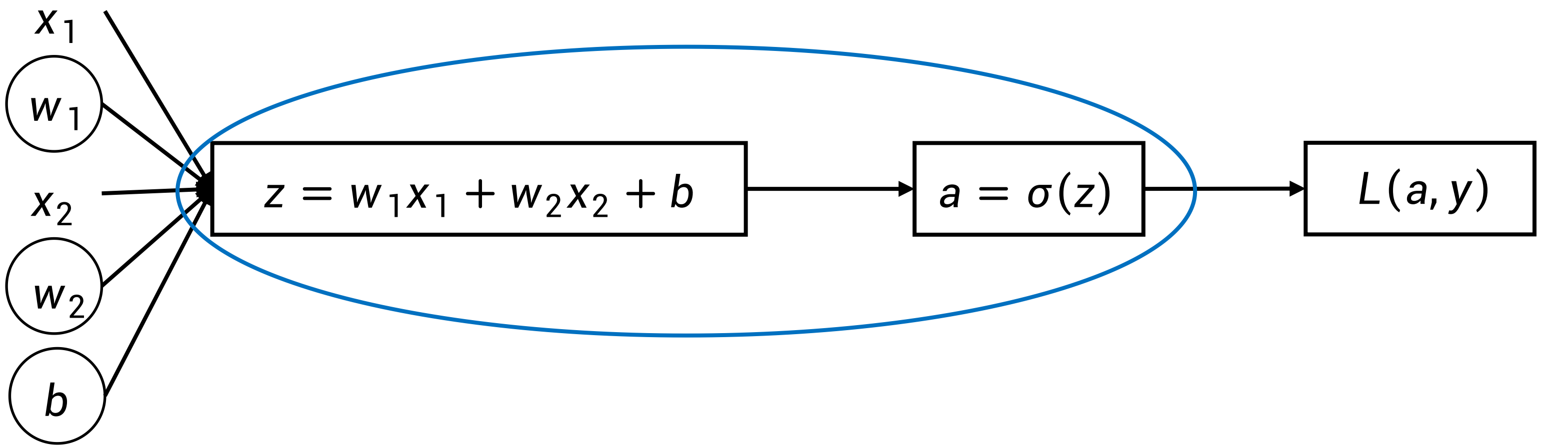
$$\frac{\partial \mathfrak{l}}{\partial a} = -\frac{y}{a} + \frac{1-y}{1-a} = \text{“}\mathbf{da}\text{” for short}$$

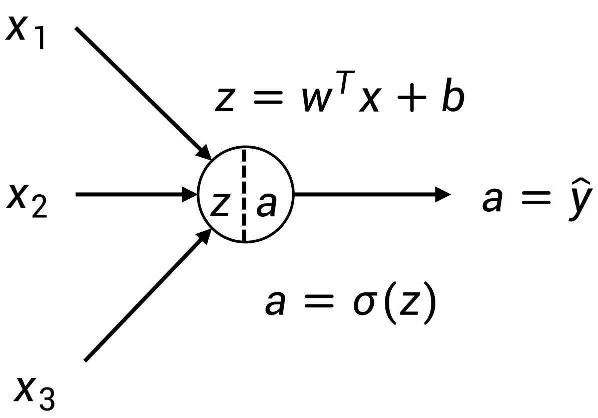
The derivative of sigmoid function

$\mathbf{dw_2} = (a-y)x_2$

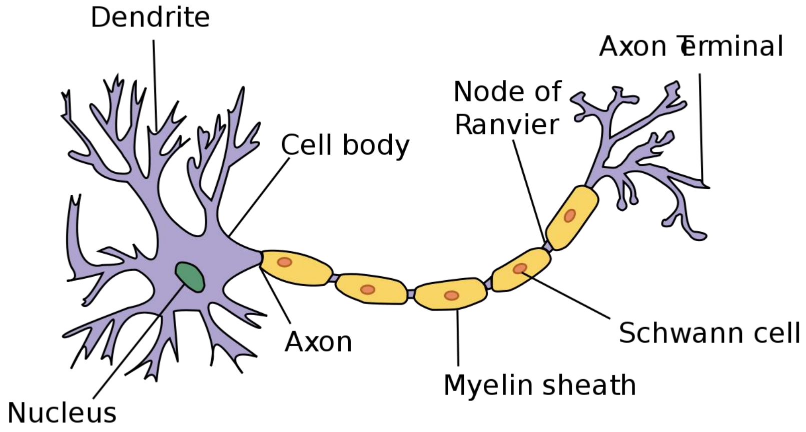
$\mathbf{db} = (a-y)$

Wrap logistic regression into a single unit



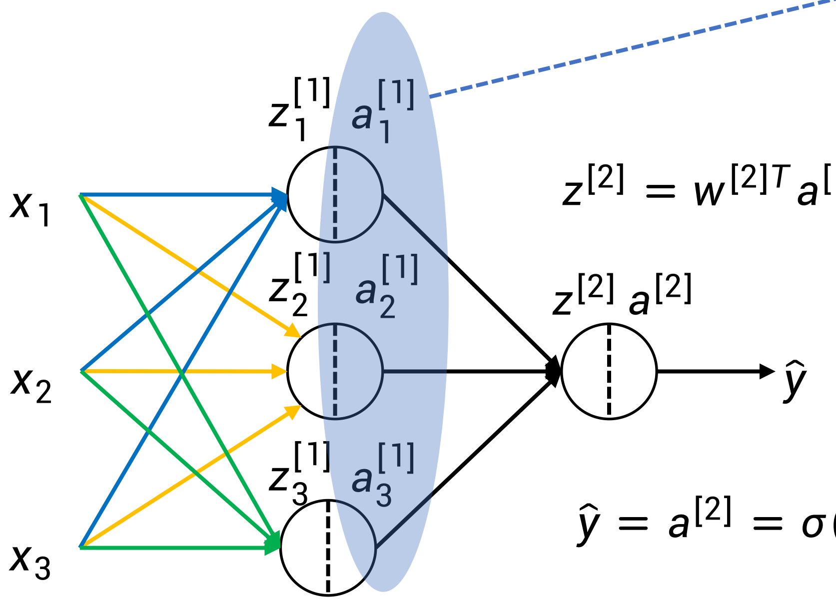


Neuron



Neural Networks

- Combine multiple neurons in one **layer**
- Put multiple layers in one network



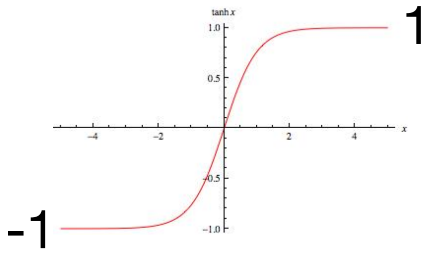
$$z^{[2]} = w^{[2]T} a^{[1]} + b^{[2]}$$

$$\hat{y} = a^{[2]} = \sigma(z^{[2]})$$

$$z_1^{[1]} = w_1^{[1]T} x + b_1^{[1]} \qquad a_1^{[1]} = \sigma(z_1^{[1]})$$

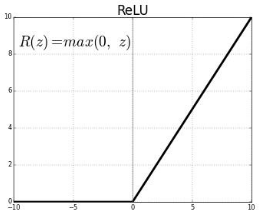
Other options for activation functions

- **Hyperbolic tangent**: Almost always works better than sigmoid.



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

- **Rectified linear unit (ReLU)**

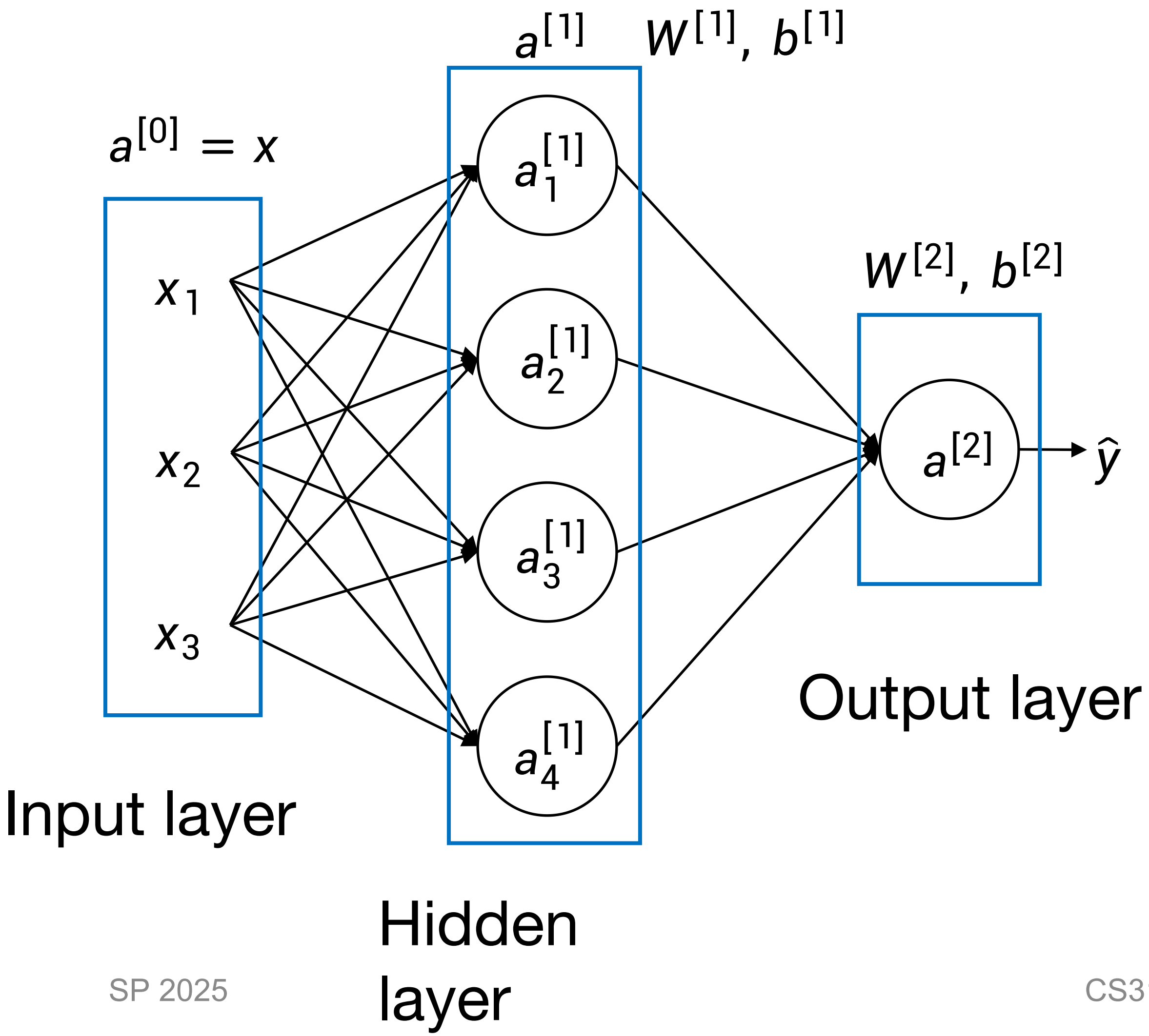


$$g(z) = \max(0, z)$$

$$g'(z) \in \{0, 1\}$$

Neural Networks - Matrix & vector repr.

A two-layer neural network



Parameters of layer 1 (input to hidden)

$$W^{[1]} = \begin{bmatrix} w^{[1]}_{11} w^{[1]}_{12} w^{[1]}_{13} \\ w^{[1]}_{21} w^{[1]}_{22} w^{[1]}_{23} \\ w^{[1]}_{31} w^{[1]}_{32} w^{[1]}_{33} \\ w^{[1]}_{41} w^{[1]}_{42} w^{[1]}_{43} \end{bmatrix} \in \mathbb{R}^{4\times3} \qquad b^{[1]} = \begin{bmatrix} b^{[1]}_1 \\ b^{[1]}_2 \\ b^{[1]}_3 \\ b^{[1]}_4 \end{bmatrix} \in \mathbb{R}^{4\times1}$$

Parameters of layer 2 (hidden to output)

$$W^{[2]} = [w^{[2]}_{11}, w^{[2]}_{12}, w^{[2]}_{13}, w^{[2]}_{14}] \in \mathbb{R}^{1\times4} \qquad b^{[2]} \in \mathbb{R}^{1\times1}$$

General principle

$$\underbrace{z^{[l]}}_{n^{[l]}\times1} = \underbrace{W^{[l]}}_{n^{[l]}\times n^{[l-1]}} \underbrace{a^{[l-1]}}_{n^{[l-1]}\times1} + \underbrace{b^{[l]}}_{n^{[l]}\times1}$$

$$a^{[l]} = g(z^{[l]})$$

$n^{[l]}$: number of units in layer l

g is the activation function

Backpropagation of neural networks

$$\mathbf{d}\boldsymbol{W}^{[1]} = \mathbf{dz}^{[1]} \frac{\partial z^{[1]}}{\partial W^{[1]}} = \mathbf{dz}^{[1]} \boldsymbol{x}^{T}$$

$$\mathbf{db}^{[1]} = \mathbf{dz}^{[1]} \frac{\partial z^{[1]}}{\partial b^{[1]}} = \mathbf{dz}^{[1]}$$

$$\mathbf{da}^{[1]} = \mathbf{dz}^{[2]} \frac{\partial z^{[2]}}{\partial a^{[1]}}$$

$$= W^{[2]T} \mathbf{dz}^{[2]}$$

$W^{[2]}$

$b^{[2]}$

$\mathbf{d}\boldsymbol{W}^{[2]} = ?$

$$\mathbf{d}\boldsymbol{W}^{[2]} = \mathbf{dz}^{[2]} \mathbf{a}^{[1]T}$$

$$\mathbf{db}^{[2]} = \mathbf{dz}^{[2]}$$

X

$W^{[1]}$

$b^{[1]}$

$$\boxed{z^{[1]} = W^{[1]} x + b^{[1]}} \rightarrow \boxed{a^{[1]} = g(z^{[1]})} \rightarrow \boxed{z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}} \rightarrow \boxed{a^{[2]} = \sigma(z^{[2]})} \rightarrow L(a^{[2]}, y)$$

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

$$\mathbf{da}^{[2]} = \frac{\partial L}{\partial a^{[2]}} = -\frac{y}{a^{[2]}} + \frac{1-y}{1-a^{[2]}}$$

$$\mathbf{dz}^{[1]} = \mathbf{da}^{[1]} * \frac{\partial a^{[1]}}{\partial z^{[1]}}$$

$$= W^{[2]T} \mathbf{dz}^{[2]} * g'(z^{[1]})$$

element-wise multiplication

$$\frac{\partial z^{[2]}}{\partial a^{[1]}} = W^{[2]T}$$

$\mathbf{da}^{[1]}$ is the same dimension as $a^{[1]}$

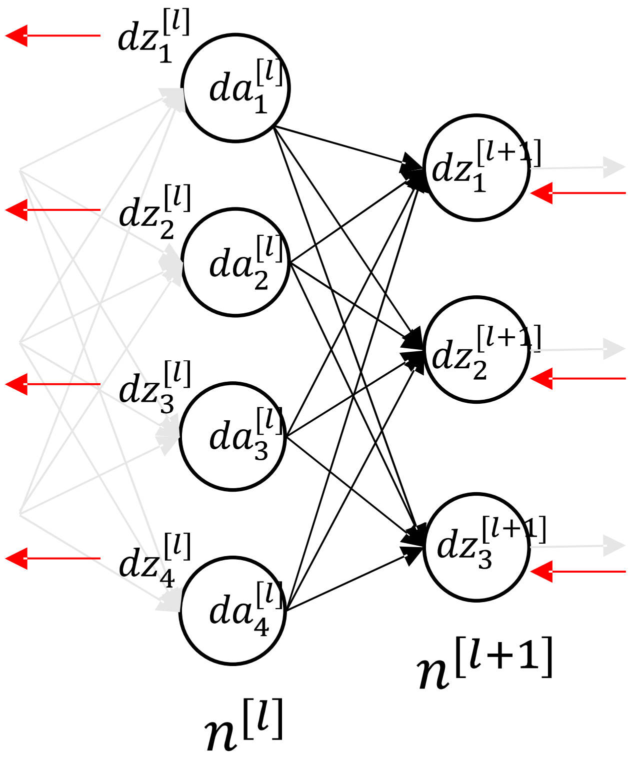
$$\frac{\partial z^{[2]}}{\partial W^{[2]}} = \frac{\partial z^{[2]}}{\partial w^{[2]}} = a^{[1]T}$$

$\mathbf{d}\boldsymbol{W}^{[2]}$ is the same dimension as $\boldsymbol{W}^{[2]}$

$\frac{\partial a^{[1]}}{\partial z^{[1]}} = g'(z^{[1]})$, depending on the activation function

Backprop in general

$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}$ $\quad a^{[l+1]} = g(a^{[l+1]})$ **Forward propagation**



$W^{[l+1]}, b^{[l+1]}$

$dW^{[l+1]} = dz^{[l+1]} a^{[l]T}$

$db^{[l+1]} = dz^{[l+1]}$

Backward propagation

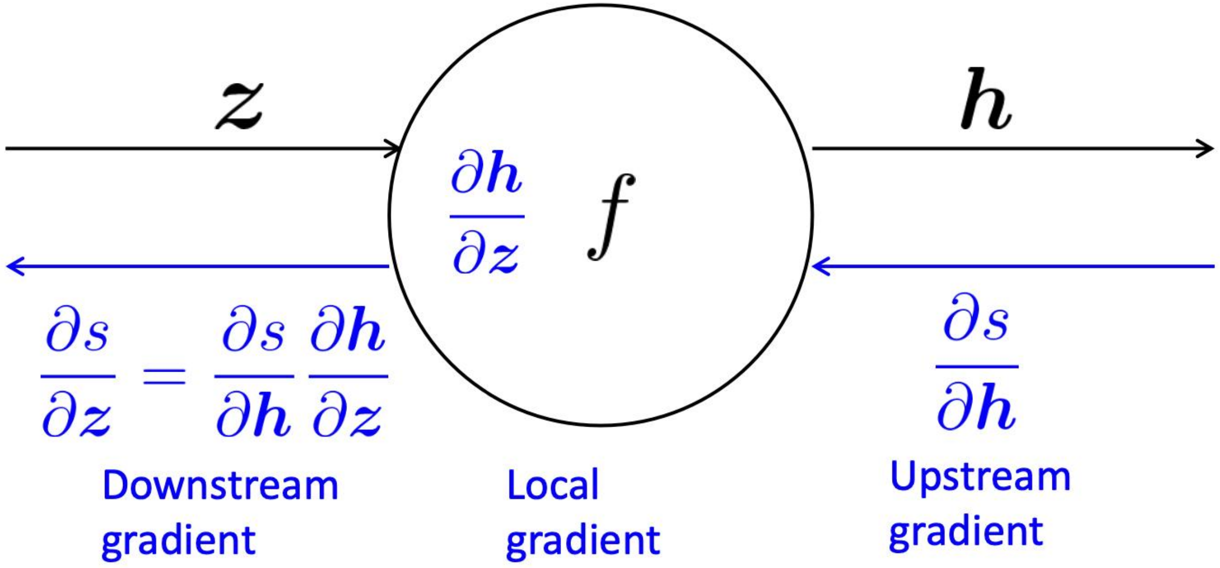
$da^{[l]} = W^{[l+1]T} dz^{[l+1]}$

$dz^{[l]} = da^{[l]} * g'(z^{[l]}) = W^{[l+1]T} dz^{[l+1]} * g'(z^{[l]})$

$dz^{[l]}$ is then passed to layer $l-1$ to compute $dW^{[l]}$ and $db^{[l]}$

Backprop: F-Prop + B-Prop

- F-Prop: Compute results and save intermediate values
- B-Prop: Apply chain rule to compute gradients



Each node

- Receives an “upstream gradient”
- Goal is to pass on the correct “downstream gradient”

[downstream gradient] = [upstream gradient] x [local gradient]

Slide credit to: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.12

Content

- **Neural Networks**
  - **Logistic regression**
  - **Gradient descent**
  - **Neural networks and back-propagation**
  - **PyTorch Implementation**
- Word Vectors
- Neural Text Classification

Basic neural network in PyTorch

- torch.nn.Linear

1. Use the torch.nn module

```
import torch
import torch.nn as nn
import numpy as np
torch.manual_seed(0)
```

2. Define a Linear layer

```
m = nn.Linear(4, 3)
# Equivalent to nn.Linear(in_features = 4, out_features = 3, bias = True
```

3. Let's look at the initial weight and bias:

```
print(m.weight)
print(m.bias)
```

```
Parameter containing:
tensor([[ 0.3742, -0.0806,  0.0529,  0.4527],
        [-0.4638, -0.3148, -0.1266, -0.1949],
        [ 0.4320, -0.3241, -0.2302, -0.3493]], requires_grad=True)
Parameter containing:
tensor([-0.4683, -0.2919,  0.4298], requires_grad=True)
```

Basic neural network in PyTorch

4. Set some values we like

```
# Manually set weight and bias
m.weight.data = torch.tensor(np.arange(1,13).reshape((3,4))).float()
print(m.weight)
m.bias.data = torch.ones_like(m.bias.data).float()
print(m.bias)
```

```
Parameter containing:
tensor([[ 1.,  2.,  3.,  4.],
        [ 5.,  6.,  7.,  8.],
        [ 9., 10., 11., 12.]], requires_grad=True)
Parameter containing:
tensor([1., 1., 1.], requires_grad=True)
```

Basic neural network in PyTorch

5. Prepare a single input data example

```
x = torch.randn(1, 4)
print(x.shape)
```

```
torch.Size([1, 4])
```

Get the output by ***calling*** the model

```
out = m(x)
print(out.shape)
```

```
torch.Size([1, 3])
```

x is 1×4:
By default, the first dimension of a tensor stands for batch size

out is 1×3:
As the result of m.weight (4×3) times x (1×4)

Basic neural network in PyTorch

6. Prepare a batch of data examples

```
x1 = torch.randn(100, 4)
out1 = m(x1)
print(out1.shape)
```

```
torch.Size([100, 3])
```

x is 100×4:
By default, the first dimension of a tensor stands for batch size

out is 100×3:
As the result of m.weight (4×3) times x (100×4)

Basic neural network in PyTorch

7. Let’s manually check what computation is done

Input vector: [1,1,1,1]

```
x2 = torch.ones(1,4)
print(x2)
out2 = m(x2)
print(out2)
```

```
tensor([[1., 1., 1., 1.]])
tensor([[11., 27., 43.]], grad_fn=<AddmmBackward>)
```

Manual computation:

Weight x Bias

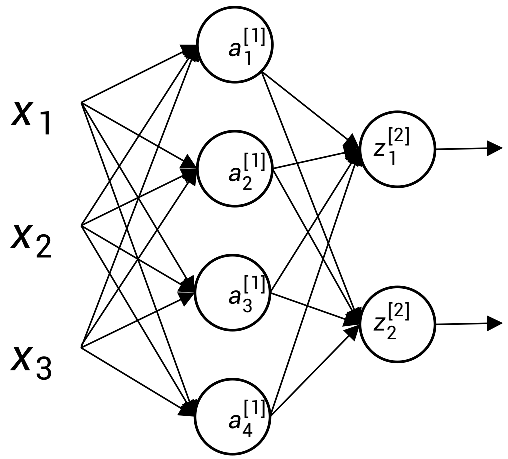
$$out2 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Backprop in PyTorch

Necessary imports

```python
import torch
import torch.nn as nn
import numpy as np
torch.manual_seed(0)
```

```
<torch._C.Generator at 0x111b7ddb0>
```



Define linear and activation layers

```python
linear1 = nn.Linear(3, 4)
act1 = nn.ReLU()
linear2 = nn.Linear(4, 2)
act2 = nn.Sigmoid()
```

Initialize to values easier to read:

```python
linear1.weight.data = torch.tensor(np.arange(1,13).reshape((4,3))).float()
print(linear1.weight)
linear1.bias.data = torch.zeros_like(linear1.bias.data).float()
print(linear1.bias)
```

```
Parameter containing:
tensor([[ 1.,  2.,  3.],
        [ 4.,  5.,  6.],
        [ 7.,  8.,  9.],
        [10., 11., 12.]], requires_grad=True)
Parameter containing:
tensor([0., 0., 0., 0.], requires_grad=True)
```

Backprop in PyTorch (layer 1)

Prepare some toy data, and watch the output from layer 1

```
x = torch.tensor([[1.,2.,3.]], requires_grad=True)
print('x: ', x)
z1 = linear1(x)
a1 = act1(z1)

print(z1)
print(a1)
print(a1.shape)
```

```
x:  tensor([[1., 2., 3.]], requires_grad=True)
tensor([[14., 32., 50., 68.]], grad_fn=<AddmmBackward>)
tensor([[14., 32., 50., 68.]], grad_fn=<ReluBackward0>)
torch.Size([1, 4])
```

Forward:

$$z1 = \underbrace{\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}}_{\text{Weight}} \underbrace{\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}}_{\text{x}} + \underbrace{\begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}}_{\text{Bias}}$$

Backprop in PyTorch (layer 1)

Create some pseudo gradients that are in the same shape as a1

```
external_grad = torch.ones_like(a1) * 0.5
a1.backward(gradient=external_grad)
```

$$dz = da = \begin{pmatrix} .5 \\ .5 \\ .5 \\ .5 \end{pmatrix} \qquad W = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix}$$

Gradients computed:

```
print(x.grad)
print(linear1.weight.grad)
print(linear1.bias.grad)
```

```
tensor(
tensor(


tensor(                                       )
```

$$dx = W^T dz = \begin{bmatrix} 1 & 4 & 7 & 10 \\ 2 & 5 & 8 & 11 \\ 3 & 6 & 9 & 12 \end{bmatrix} \begin{pmatrix} .5 \\ .5 \\ .5 \\ .5 \end{pmatrix} = \begin{pmatrix} 11 \\ 13 \\ 15 \end{pmatrix}$$

$$dW = dz \cdot x^T = \begin{pmatrix} .5 \\ .5 \\ .5 \\ .5 \end{pmatrix} (1., 2., 3.) =$$

Backprop in PyTorch (layer 2)

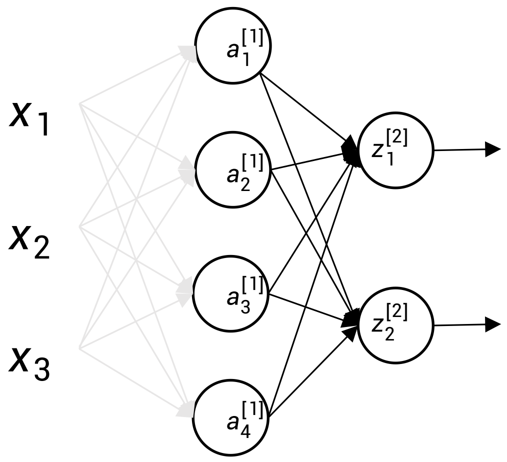
Initialize the weight of linear2 to zeros (0), because sigmoid function saturates very fast

```
linear2.weight.data = torch.zeros_like(linear2.weight.data).float()
linear2.bias.data = torch.zeros_like(linear2.bias.data).float()
print(linear2.weight)
print(linear2.bias)
```

```
Parameter containing:
tensor([[0., 0., 0., 0.],
        [0., 0., 0., 0.]], requires_grad=True)
Parameter containing:
tensor([0., 0.], requires_grad=True)
```

Forward pass:

```
x = torch.tensor([[1.,2.,3.]])
z1 = linear1(x)
a1 = act1(z1)
z2 = linear2(a1)
a2 = act2(z2)
```



Output:

```
z2:  tensor([[0., 0.]], grad_fn=<AddmmBackward>)
a2:  tensor([[0.5000, 0.5000]], grad_fn=<SigmoidBackward>)
```

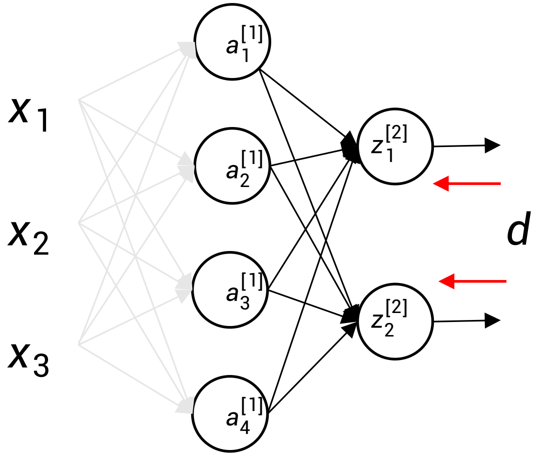
$$\sigma(0) = \frac{1}{1+e^0} = 0.5$$

Backprop in PyTorch (layer 2)

Backward pass:

Tell PyTorch to save the gradients for intermediate variables

```
z2.retain_grad()
a1.retain_grad()
external_grad = torch.ones_like(a2) * 0.5
a2.backward(gradient=external_grad)
```



$$da^{[2]} = \begin{pmatrix}0.5\\0.5\end{pmatrix}$$

Manually check dz_2

```
# dz = a*(1 - a) * da
a2_np = a2.data.numpy()
dz2_manual = a2_np * (1 - a2_np) * external_grad.numpy()
dz2 = z2.grad.numpy()
print(np.equal(dz2_manual, dz2))
print('dz2: ', dz2)
```

$$dz^{[2]} = da^2 * \frac{\partial a^{[2]}}{\partial z^{[2]}}$$

$$= da^2 * g'(z^{[2]})$$

$$= da^2 * \sigma'(z^{[2]})$$

Derivatives of sigmoid function:

$$\sigma'(z^{[2]})$$

$$= \sigma(z^{[2]}) \cdot (1 - \sigma(z^{[2]}))$$

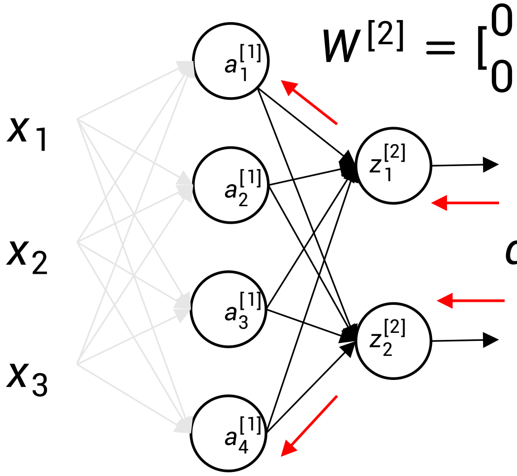
$$= a^{[2]} \cdot (1 - a^{[2]}))$$

$$= \begin{pmatrix}0.5\\0.5\end{pmatrix} \cdot (1 - \begin{pmatrix}0.5\\0.5\end{pmatrix}))$$

Output:

```
[[ True  True]]
dz2:  [[0.125 0.125]]
```

Backprop in PyTorch (layer 2)



$$W^{[2]} = \begin{bmatrix} 0000 \\ 0000 \end{bmatrix}$$

$$da^{[2]} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

$$da^{[1]} = ?$$

We know: $dz^{[2]} = \begin{pmatrix} 0.125 \\ 0.125 \end{pmatrix}$ so $db^{[2]} = da^{[2]} = \begin{pmatrix} 0.125 \\ 0.125 \end{pmatrix}$

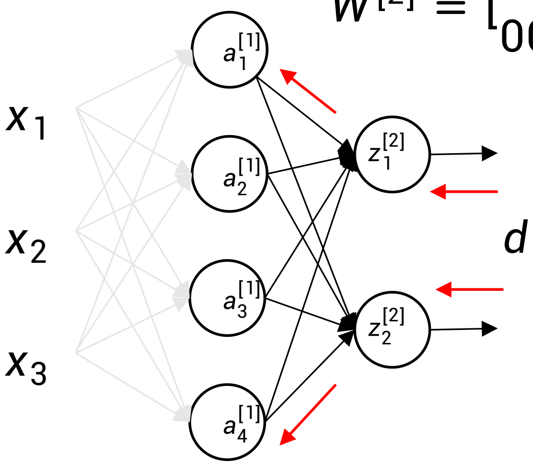
$$da^{[1]} = W^T dz^{[2]} = \begin{bmatrix} 0, 0 \\ 0, 0 \\ 0, 0 \\ 0, 0 \end{bmatrix} \begin{pmatrix} 0.125 \\ 0.125 \end{pmatrix} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$

```
print('dW2: ', linear2.weight.grad)
print('db2: ', linear2.bias.grad)
print('dz2: ', z2.grad)
print('da1: ', a1.grad)
```

```
dW2:  tensor([[1.7500, 4.0000, 6.2500, 8.5000],
        [1.7500, 4.0000, 6.2500, 8.5000]])
db2:  tensor([0.1250, 0.1250])
dz2:  tensor([[0.1250, 0.1250]])
da1:  tensor([[0., 0., 0., 0.]])
```

Backprop in PyTorch (layer 2)

$$W^{[2]} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$



$$da^{[2]} = \begin{pmatrix} 0.5 \\ 0.5 \end{pmatrix}$$

$$a^{[1]} = \begin{pmatrix} 14 \\ 32 \\ 52 \\ 68 \end{pmatrix}$$

$$dW^{[2]} = ?$$

$$dW_2 = dz_2 a_1^T = \begin{pmatrix} 0.125 \\ 0.125 \end{pmatrix} (14,\ 32,\ 50,\ 68)$$

Manually check dW_2

```
a1_np = a1.data.numpy()
print('a1: ', a1_np)
dW2 = np.dot(dz2.T, a1_np)
print('dW2: ', dW2)
```

Output:

```
dW2:  [[1.75 4.   6.25 8.5 ]
 [1.75 4.   6.25 8.5 ]]
```

Backprop in PyTorch - nn.module

```
class Net(nn.Module):
    def __init__(self):
        super(Net,
    self).__init__()
        ...
    def forward(self, x):
        …
        return xx
```

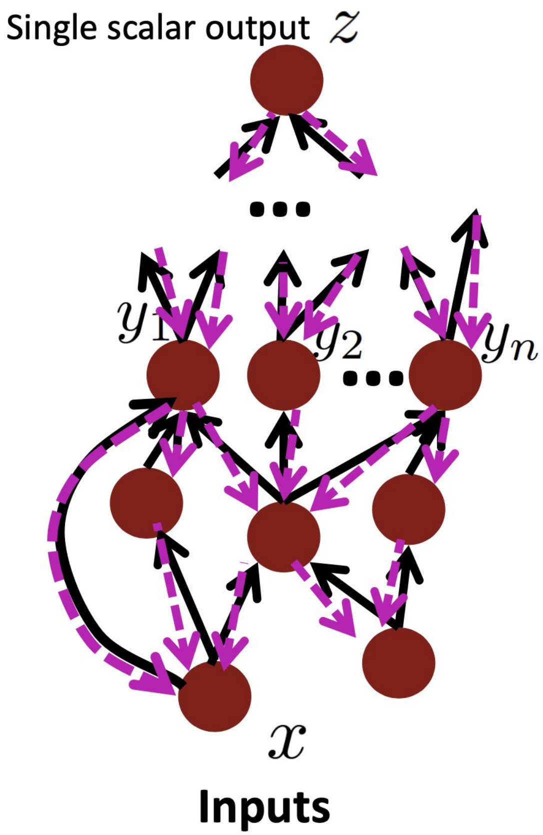
Define model architecture in __init__()

Define forward propagation in forward()

General procedure of gradient descent:

```
# Build the model
net = Net()
# Execute F-prop
output = net(input)
# Compute loss
loss = criterion(output, targe
# B-prop
loss.backward()
optimizer.step()
```

Backprop in General



1. F-prop: visit nodes in topological sort order
- Compute value of node given predecessors

2. B-prop:
- initialize output gradient = 1
- visit nodes in *reverse* order: Compute gradient w.r.t. each node using gradient w.r.t. successors

$\{y_1, y_2, \ldots, y_n\}$ = successors of x

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}$$

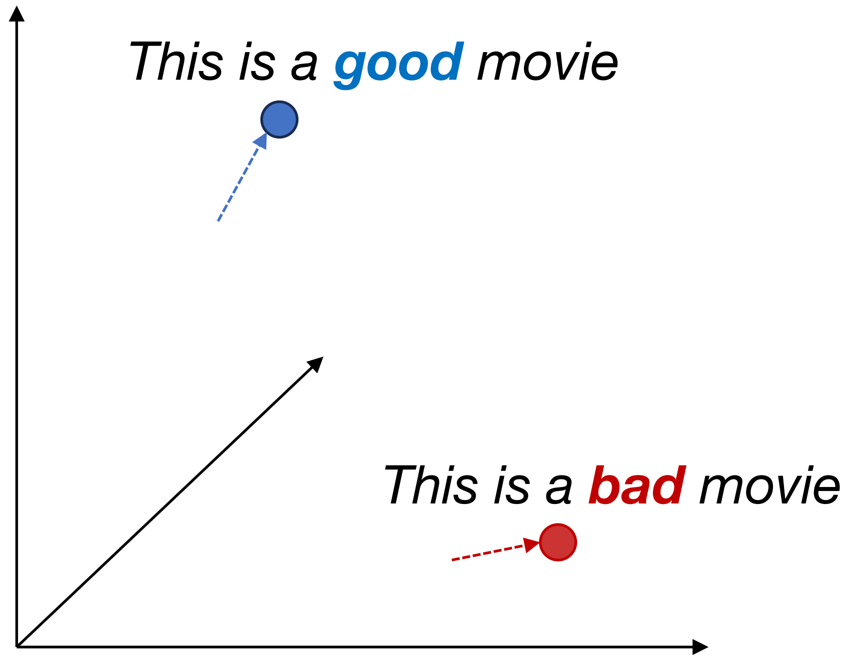
Slide credit to: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.12

Content

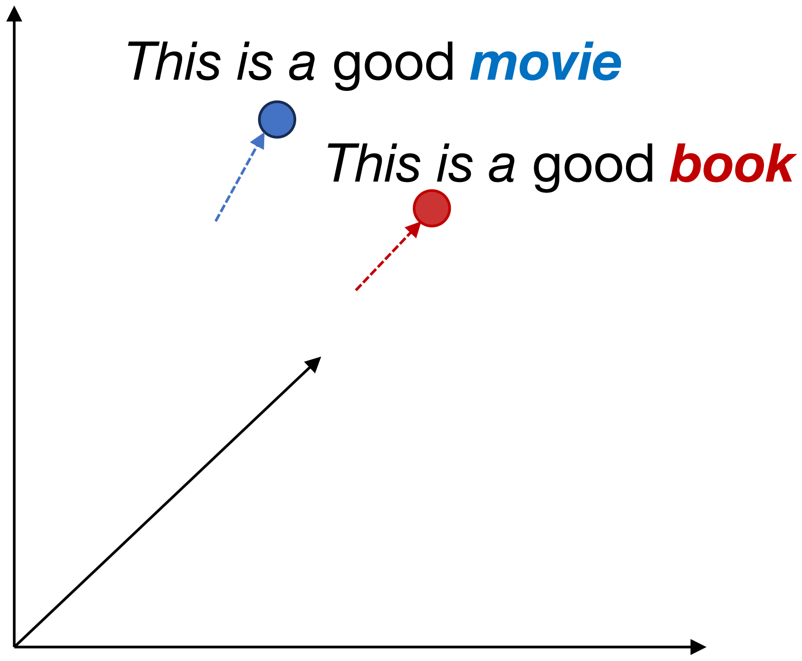
- Neural Networks
- **Word Vectors**
  - **Why do we need word vectors?**
  - **How? Discrete vs. continuous**
- Neural Text Classification

Why using vectors to represent words?

- Vectors are good representations for meanings



How ?



Motivation

- How do we represent word **meanings**?
- by *Webster* dictionary - the **idea** that is represented by a word, phrase, etc.
- by 《现代汉语词典》- 意义：语言文字或其他信号所表示的**内容**
- by 《说文解字》- 意：志也。从心，察言而知意也。

- From dictionary, we can get a sequence of *symbols*.
- $\text{Sym}_A \longrightarrow \text{Sym}_B \longrightarrow \text{Sym}_C \longrightarrow \text{Sym}_D \ldots$

Symbolic meanings

- Common linguistic way: use symbols as signifier

signifier (symbol) ⇔ signified (idea or thing)

= denotational semantics

tree ⇔ {, , , ...}

slide credit to: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.12

Computable meaning

- Example, **WordNet**, a thesaurus containing lists of synonym sets and hypernyms

```
>>> from nltk.corpus import wordnet as wn
```

```
>>> wn.synset('dog.n.01')
Synset('dog.n.01')
>>> print(wn.synset('dog.n.01').definition())
a member of the genus Canis (probably descended from the
```

```
>>> dog = wn.synset('dog.n.01')
>>> dog.hypernyms()
[Synset('canine.n.02'), Synset('domestic_animal.n.01')]
```

source: https://www.nltk.org/howto/wordnet.html

Words are nodes in a tree/graph structure

```
>>> dog = wn.synset('dog.n.01')
>>> cat = wn.synset('cat.n.01')
```

```
>>> dog.path_similarity(cat)
0.2...
```

How similar two word senses are, based on the shortest path

Limits of WordNet (-like) resources

- Missing nuance
  - E.g., “proficient” is listed as a synonym for “good”
  - This is only correct in some contexts
- Missing new meanings of words
  - E.g., wicked, badass, nifty, wizard, genius, ninja, bombast
  - Impossible to keep up-to-date
- Subjective
- Requires human labor to create and adapt
- Cannot be used to accurately compute word sense similarity

Slide credit to: https://web.stanford.edu/class/archive/cs/cs224n/cs224n.12

Words as one-hot vectors

- Words as discrete symbols ⇔ (equivalent to) localist representations
- **One-hot** vectors

Vocabulary (10k) = $\begin{bmatrix} a \\ about \\ all \\ \vdots \\ zoo \end{bmatrix}$

Apple [0 0 0 0 0 0 0 0 **1** 0 0 0 0 0 0 0 0 ... 0]

Orange [0 0 0 0 0 0 0 0 0 0 0 0 0 **1** 0 0 0 ... 0]

I would like some apple juice

I would like some orange ____

One-hot vector is not helpful

Distance between any pair of words is **constant:**

Euclidean distance = $\sqrt[2]{(1-0)^2+(1-0)^2}$

Cosine distance = 0

Words as real-valued vectors

	Man	Woman	King	Queen	Apple	Orange
Gender	-1	1	-0.98	0.97	0.00	-0.01
Royal	0.01	0.02	0.93	0.98	-0.01	0.00
Age	0.03	0.02	0.72	0.68	0.03	0.02
Food	0.00	0.00	0.01	0.02	0.95	0.97

main difference: gender

main difference: gender

$$e_{Man} = \begin{bmatrix} -1 \\ 0.01 \\ 0.03 \\ 0.0 \end{bmatrix} \quad e_{Woman} = \begin{bmatrix} 1 \\ 0.02 \\ 0.02 \\ 0.0 \end{bmatrix} \quad e_{Man} - e_{Woman} = \begin{bmatrix} -2 \\ -0.01 \\ 0.01 \\ 0.00 \end{bmatrix} \quad e_{King} - e_{Queen} = \begin{bmatrix} -1.95 \\ -0.05 \\ 0.04 \\ -0.01 \end{bmatrix}$$

With real-valued dense vectors, word similarity can be computed more accurately

What use of word vectors?

- More accurate semantic representation than WordNet-like method
- Better performance in almost all areas of supervised learning tasks:
  - Text classification
  - Named entity recognition
  - Sentiment analysis
  - Information retrieval
  - …
- One of the most useful task: **Neural Networks**-based text classification

Content

- Neural Networks
- Word Vectors
- **Neural Text Classification**
  - **Bag-of-words; evaluate**

Text Classification

Email : spam or not?

Sentiment analysis ♥: positive or negative?

Natural language inference (NLI) :
entailment, contraction, or neutral?

Traditional way of text classification

- Traditional = feature engineering. Question: What features?

Running example:

$\boxed{\boldsymbol{x} = \text{“The vodka was great, but don't touch the hamburgers.”}}$

A different representation of the text sequences: features.

- Often, these are term (word or word sequence) frequencies. E.g., $\phi^{\textit{freq.}}_{\text{hamburgers}}(\boldsymbol{x}) = 1$, $\phi^{\textit{freq.}}_{\text{the}}(\boldsymbol{x}) = 2$, $\phi^{\textit{freq.}}_{\text{delicious}}(\boldsymbol{x}) = 0$, $\phi^{\textit{freq.}}_{\text{don't touch}}(\boldsymbol{x}) = 1$.
- Can also be binary word "presence" features. E.g., $\phi^{\textit{presence}}_{\text{hamburgers}}(\boldsymbol{x}) = 1$, $\phi^{\textit{presence}}_{\text{the}}(\boldsymbol{x}) = 1$, $\phi^{\textit{presence}}_{\text{delicious}}(\boldsymbol{x}) = 0$, $\phi^{\textit{presence}}_{\text{don't touch}}(\boldsymbol{x}) = 1$.
- Transformations on word frequencies: logarithm, idf weighting

$$\forall v \in \mathcal{V}, \mathrm{idf}(v) = \log \frac{n}{|i : \mathrm{count}_{\boldsymbol{x}_i}(v) > 0|}$$

$$\phi^{\textit{tfidf}}_{v}(\boldsymbol{x}) = \phi^{\textit{freq.}}_{v}(\boldsymbol{x}) \cdot \mathrm{idf}(v)$$

Term frequency

Term presence

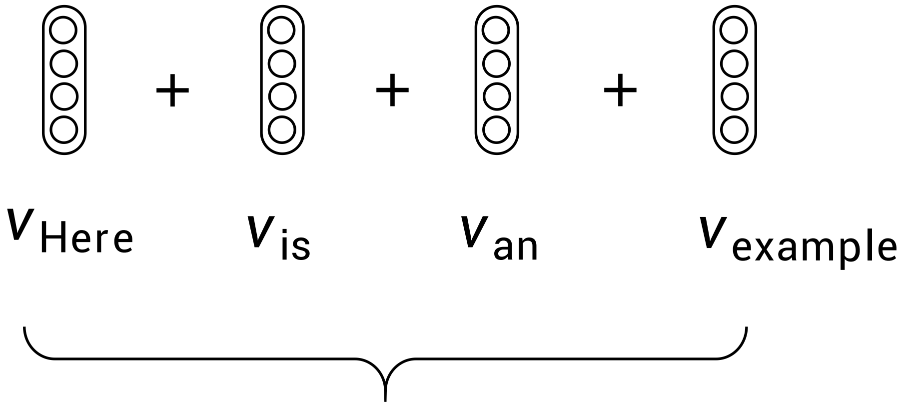
Term freq. * inverse document freq. (TFIDF)

Slide credit to: https://nasmith.github.io/NLP-winter23/calendar

Bag-of-Words Neural Networks

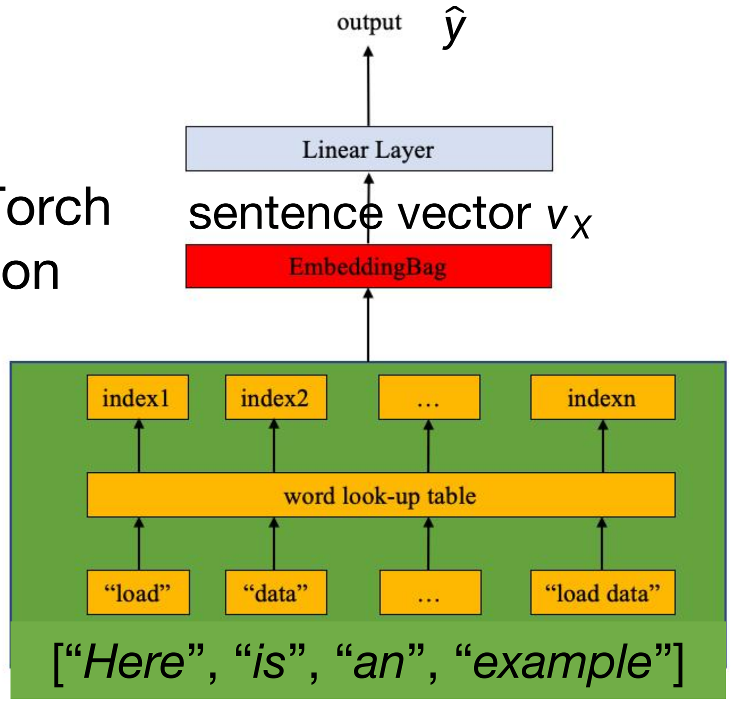
Task: News text classification

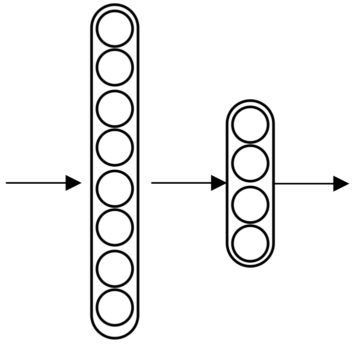
X: [“*Here*”, “*is*”, “*an*”, “*example*”]



sentence vector v_X

A typical PyTorch implementation

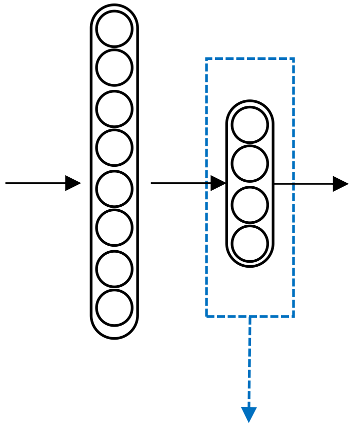




$$\hat{y} = \begin{bmatrix} P(\text{World}|X) \\ P(\text{Sports}|X) \\ P(\text{Business}|X) \\ P(\text{Sci/Tec}|X) \end{bmatrix}$$

Extended to multiple classes: Softmax layer

Size of output layer = number of classes = 4 (World, Sports, Business, Sci/Tec)



$$\hat{y} = \begin{bmatrix} P(\text{World}|X) \\ P(\text{Sports}|X) \\ P(\text{Business}|X) \\ P(\text{Sci/Tec}|X) \end{bmatrix}$$

Softmax activation function

Input: vector $z^{[L]}$

Output: vector $a^{[L]}$

Linear part: $z^{[L]} = W^{[L]} a^{[L-1]} + b^{[L]}$ $\quad 4 \times 1$

Activation part: $t = e^{z^{[L]}}$ $\quad 4 \times 1$

$$a^{[L]} = \frac{e^{z^{[L]}}}{\sum_{j=1}^{4} t_j} \quad 4 \times 1$$

In which, $a_i^{[L]} = \frac{t_i}{\sum_{j=1}^{4} t_j}$

$$z^{[L]} = \begin{bmatrix} 4 \\ 1 \\ -1 \\ 2 \end{bmatrix} \quad t = e^{z^{[L]}} = \begin{bmatrix} e^4 \\ e^1 \\ e^{-1} \\ e^2 \end{bmatrix} = \begin{bmatrix} 54.60 \\ 2.72 \\ 0.37 \\ 7.39 \end{bmatrix}$$

$$\sum_{j=1}^{4} t_j = 54.60 + 2.72 + 0.37 + 7.390 = 65.07$$

$$a^{[L]} = \begin{bmatrix} 54.60/65.07 \\ 2.72/65.07 \\ 0.37/65.07 \\ 7.39/65.07 \end{bmatrix} = \begin{bmatrix} 0.84 \\ 0.04 \\ 0.01 \\ 0.11 \end{bmatrix}$$

P(World |X) = 0.84
P(Sports |X) = 0.04
P(Business |X) = 0.01
P(Sci/Tec |X) = 0.11

Softmax layer: loss function

1□ $\hat{y} = \begin{bmatrix} 0.84 \\ 0.04 \\ 0.01 \\ 0.11 \end{bmatrix}, \; y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

2□ $\hat{y} = \begin{bmatrix} 0.84 \\ 0.04 \\ 0.01 \\ 0.11 \end{bmatrix}, \; y = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$

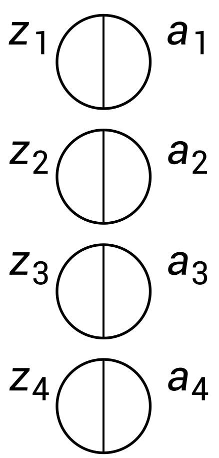
Cross entropy loss: $L = -\sum_i y_i \log(\hat{y}_i)$

Measures the distance between output distribution ($\hat{y}$) and the actual distribution (y)

In 1□, $L = -\log(0.84) = 0.174$

In 2□, $L = -\log(0.04) = 3.22$

Gradient in the same form as that of logistic regression



$L = -\sum_i y_i \log(a_i)$

$\frac{\partial L}{\partial a_i} = -\frac{y_i}{a_i}$

$\Rightarrow \quad \frac{\partial L}{\partial z_i} = \sum_k \frac{\partial L}{\partial a_k} \frac{\partial a_k}{\partial z_i} = -\sum_k \frac{y_k}{a_k} \boxed{\frac{\partial a_k}{\partial z_i}}$

$\boxed{\frac{\partial a_k}{\partial z_i}} = \begin{cases} k = i, & \frac{\partial}{\partial z_i}\left(\frac{e^{(z_i)}}{\sum_j e^{(z_j)}}\right) = a_i(1 - a_i) \\ k \neq i, & \frac{\partial}{\partial z_i}\left(\frac{e^{(z_k)}}{\sum_j e^{(z_j)}}\right) = -a_k a_i \end{cases}$

$\Rightarrow \quad \frac{\partial L}{\partial z_i} = -\frac{y_i}{a_i} a_i(1 - a_i) - \sum_{k \neq i} \frac{y_k}{a_k}(-a_k a_i) = -y_i + y_i a_i + \sum_{k \neq i} y_k a_i = \boxed{a_i - y_i}$

Evaluate a Classifier

Accuracy:

$$\begin{aligned}
\mathrm{A}(\mathrm{classify}) &= p(\mathrm{classify}(\boldsymbol{X}) = Y) \\
&= \sum_{\boldsymbol{x} \in \mathcal{V}^*, \ell \in \mathcal{L}} p(\boldsymbol{X} = \boldsymbol{x}, Y = \ell) \cdot \begin{cases} 1 & \text{if } \mathrm{classify}(\boldsymbol{x}) = \ell \\ 0 & \text{otherwise} \end{cases} \\
&= \sum_{\boldsymbol{x} \in \mathcal{V}^*, \ell \in \mathcal{L}} p(\boldsymbol{X} = \boldsymbol{x}, Y = \ell) \cdot \mathbf{1}\{\mathrm{classify}(\boldsymbol{x}) = \ell\}
\end{aligned}$$

Training accuracy

where p is the *true* distribution over data. Error is $1 - \mathrm{A}$.

This is *estimated* using a test dataset $\langle \bar{\boldsymbol{x}}_1, \bar{y}_1 \rangle, \ldots \langle \bar{\boldsymbol{x}}_m, \bar{y}_m \rangle$:

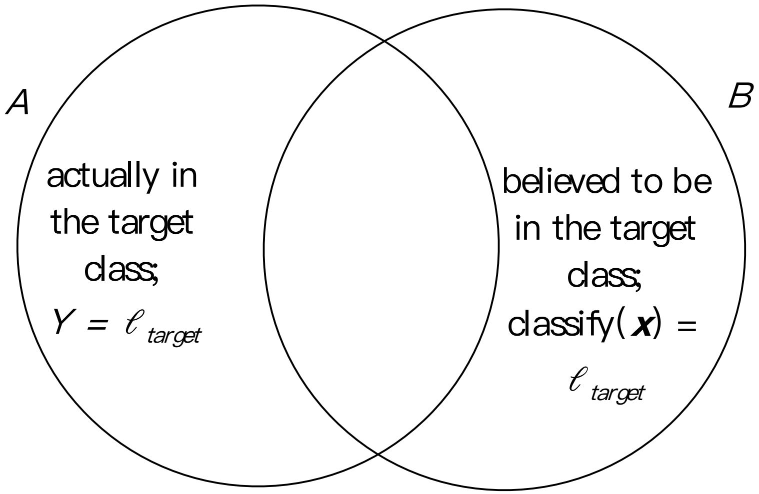
$$\hat{\mathrm{A}}(\mathrm{classify}) = \frac{1}{m} \sum_{i=1}^{m} \mathbf{1}\{\mathrm{classify}(\bar{\boldsymbol{x}}_i) = \bar{y}_i\}$$

Testing accuracy (unbiased)

Slide credit to: https://nasmith.github.io/NLP-winter23/calendar

Evaluation in the “Needle in a Haystack” Case

Suppose one label $\ell_{target} \in \mathcal{L}$ is a “target.”
Precision and recall encode the goals of returning a “pure” set of targeted instances and capturing *all* of them.



$$\hat{P}(\text{classify}) = \frac{|C|}{|B|} = \frac{|A \cap B|}{|B|}$$

$$\hat{R}(\text{classify}) = \frac{|C|}{|A|} = \frac{|A \cap B|}{|A|}$$

$$\hat{F}_1(\text{classify}) = 2 \cdot \frac{\hat{P} \cdot \hat{R}}{\hat{P} + \hat{R}}$$

Slide credit to: https://nasmith.github.io/NLP-winter23/calendar

To-Do List

- Start working on A1
- Read **Ch. 6 Vector Semantics and Embeddings** of SLP3
- Attend Lab2