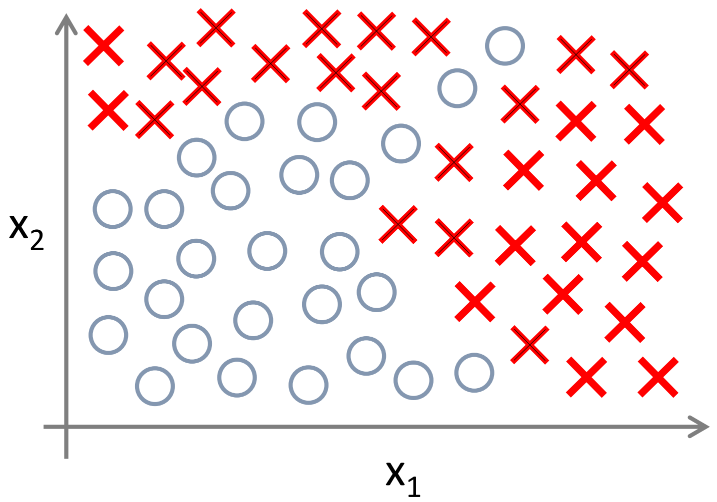
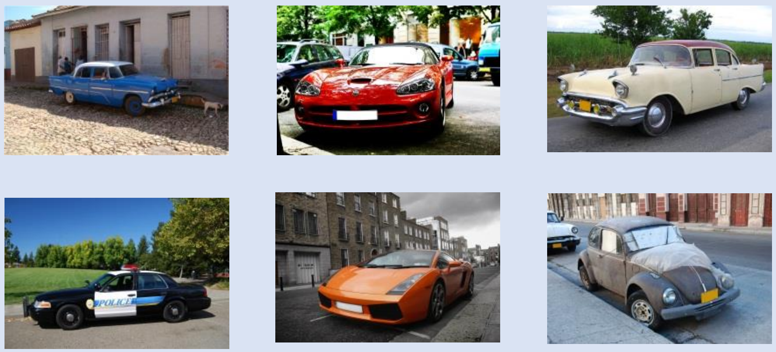
Lecture 7
Non-Linear Decision Surfaces

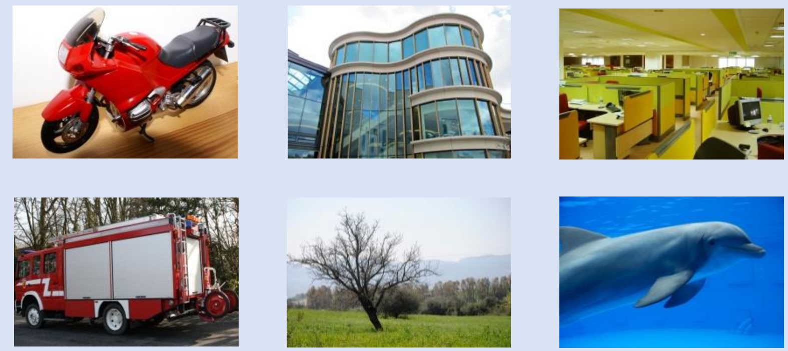


- There is no linear decision boundary

Car Classification



Cars

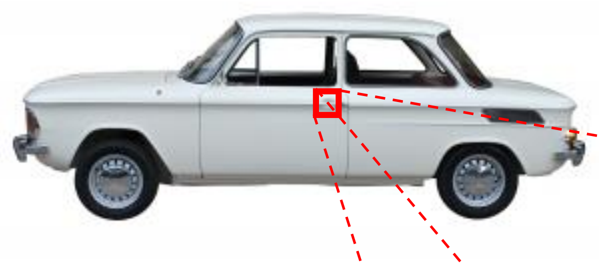


Not a car

Testing: 

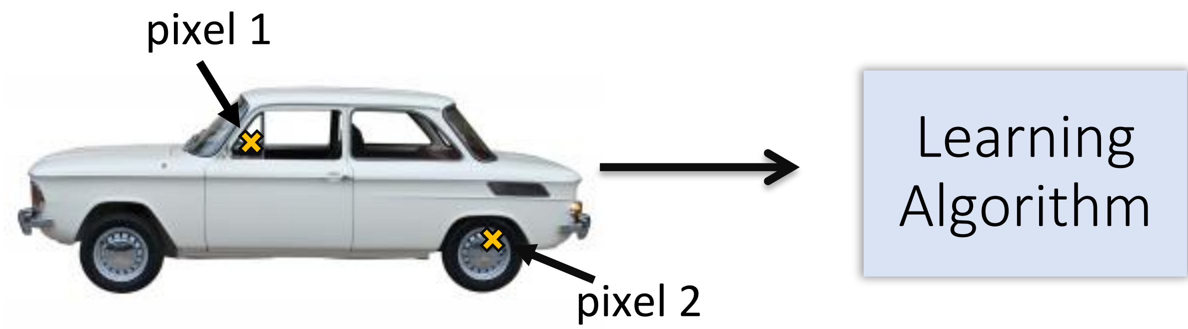
What is this?

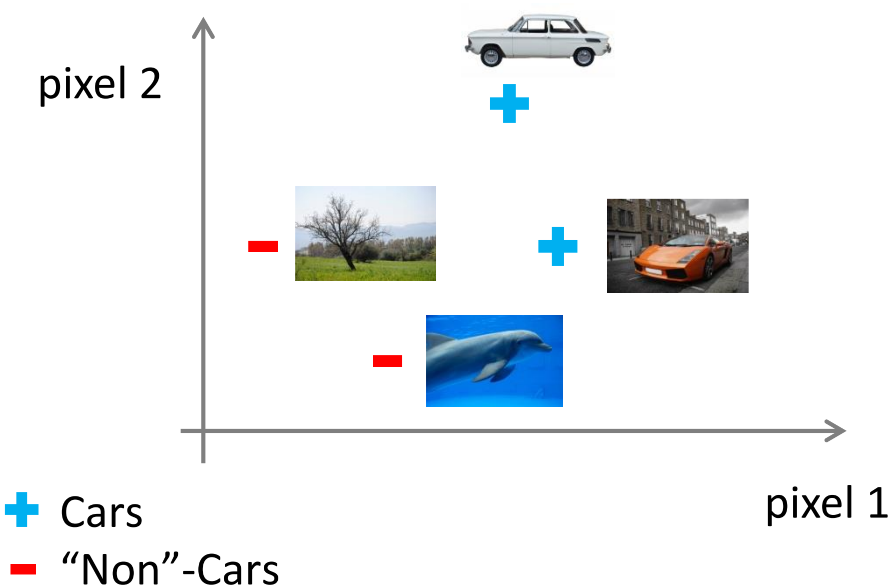
You see this:

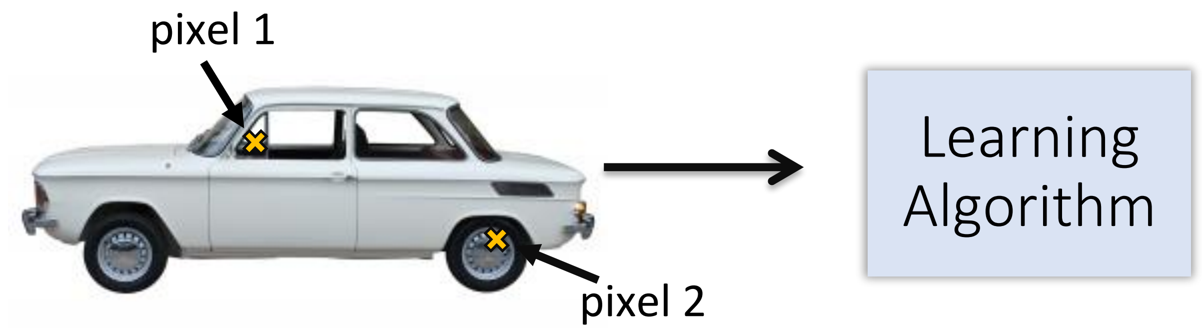


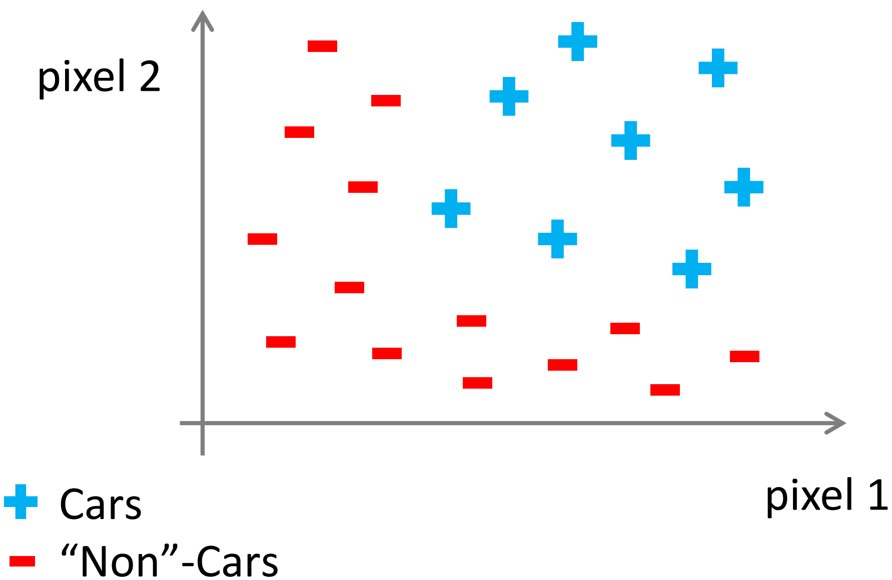
But the camera sees this:

194	210	201	212	199	213	215	195	178	158	182	209
180	189	190	221	209	205	191	167	147	115	129	163
114	126	140	188	176	165	152	140	170	106	78	88
87	103	115	154	143	142	149	153	173	101	57	57
102	112	106	131	122	138	152	147	128	84	58	66
94	95	79	104	105	124	129	113	107	87	69	67
68	71	69	98	89	92	98	95	89	88	76	67
41	56	68	99	63	45	60	82	58	76	75	65
20	43	69	75	56	41	51	73	55	70	63	44
50	50	57	69	75	75	73	74	53	68	59	37
72	59	53	66	84	92	84	74	57	72	63	42
67	61	58	65	75	78	76	73	59	75	69	50


pixel 1
pixel 2
Learning Algorithm


pixel 2
pixel 1
Cars
"Non"-Cars





50 x 50 pixel images → 2500 pixels

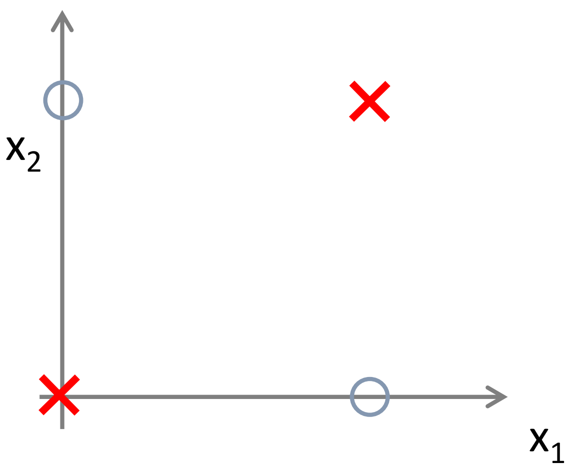
$n = 2500$ (7500 if RGB)

$$x = \begin{bmatrix} \text{pixel 1 intensity} \\ \text{pixel 2 intensity} \\ \vdots \\ \text{pixel 2500 intensity} \end{bmatrix}$$

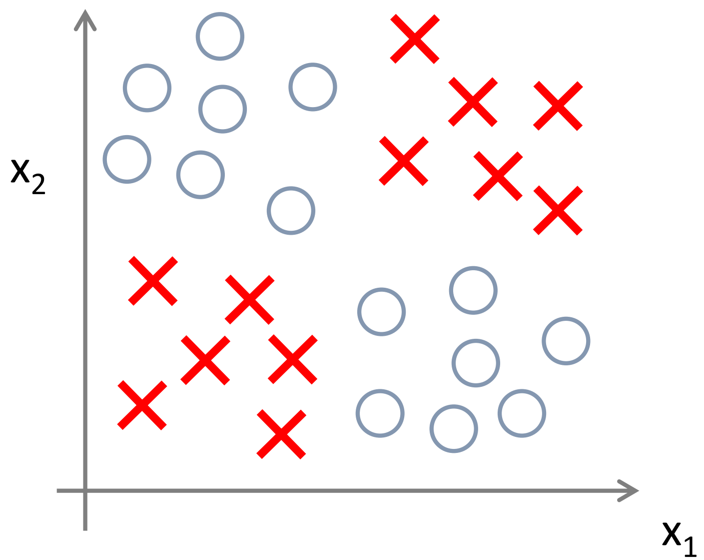
Quadratic features ($x_i \times x_j$): ≈3 million features

Simple Non-Linear Classification Example

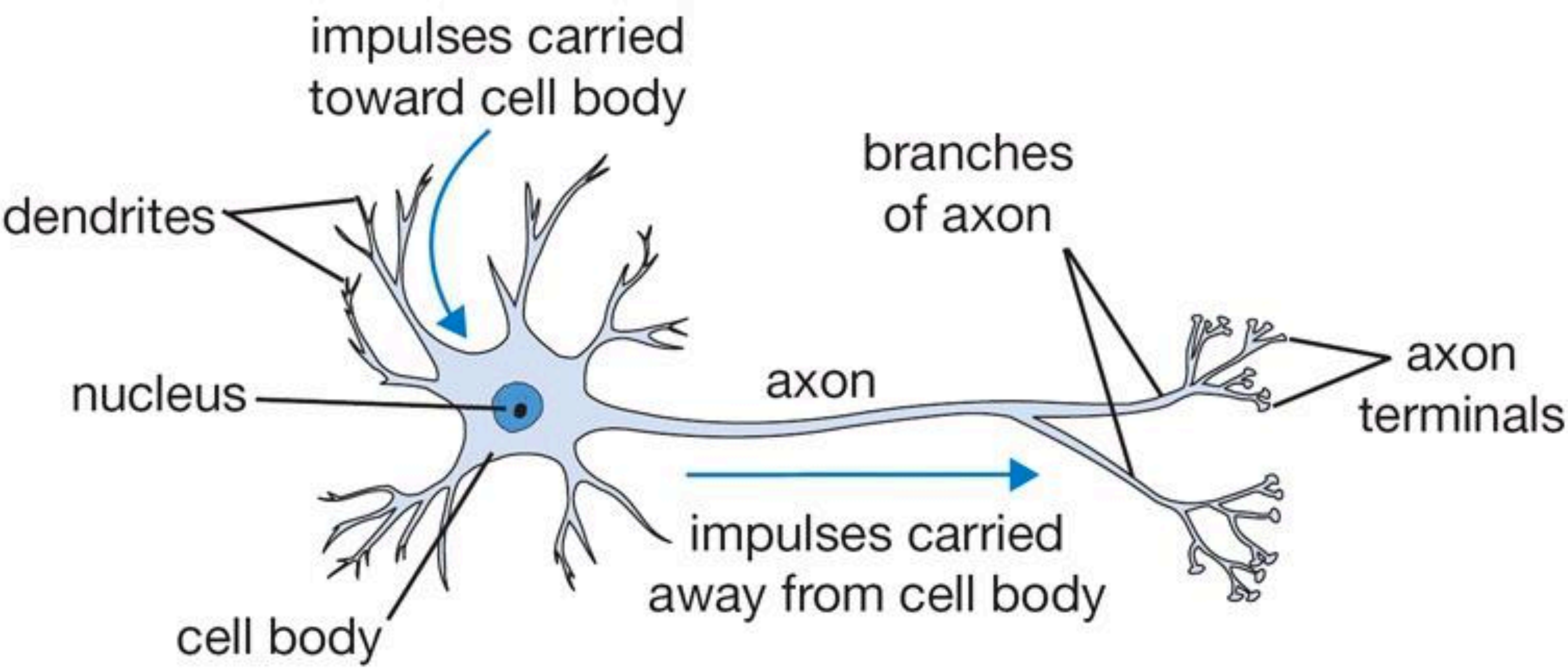
$x_1 \quad x_2$



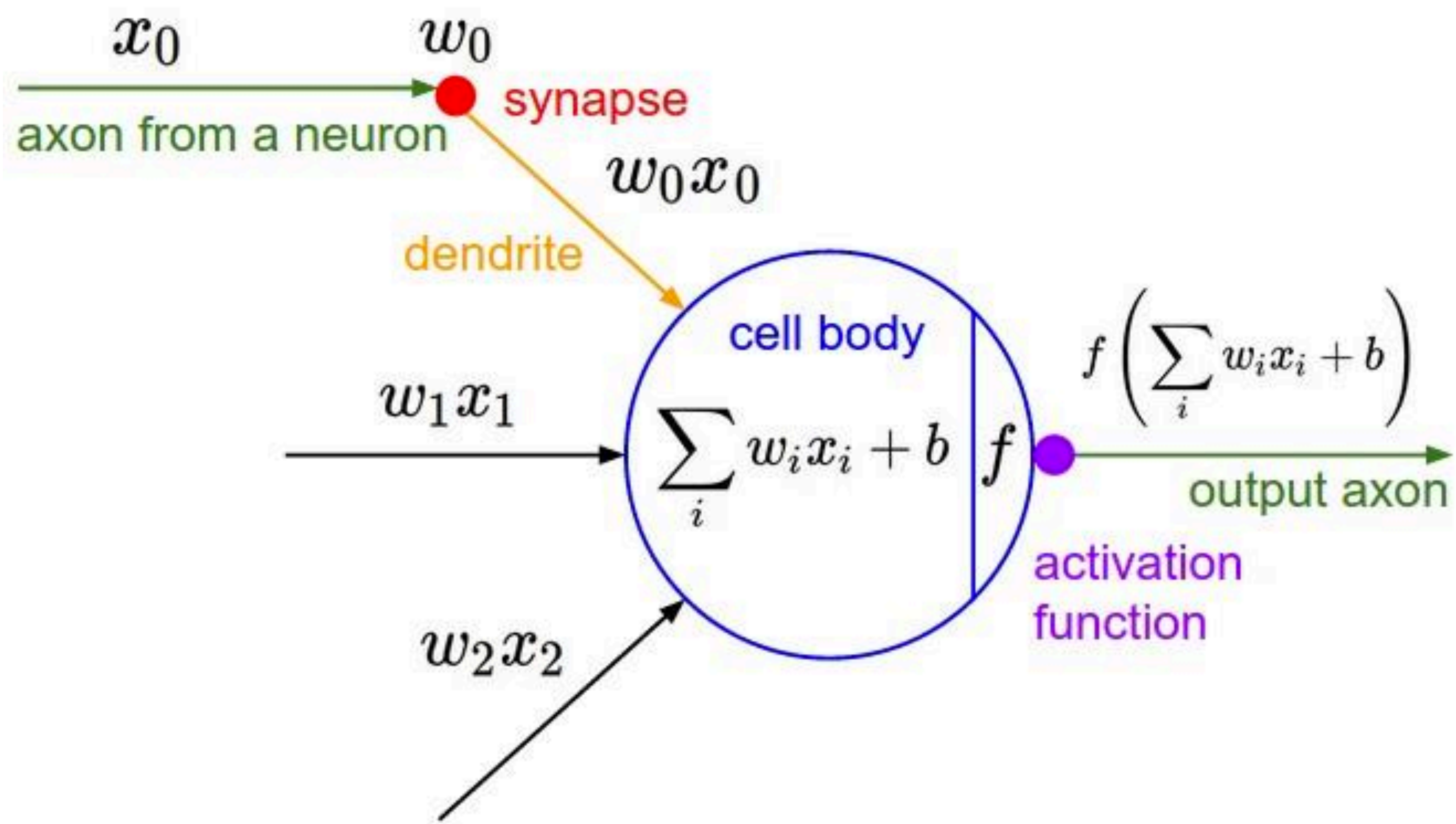
$$y = x_1 \text{ XOR } x_2$$



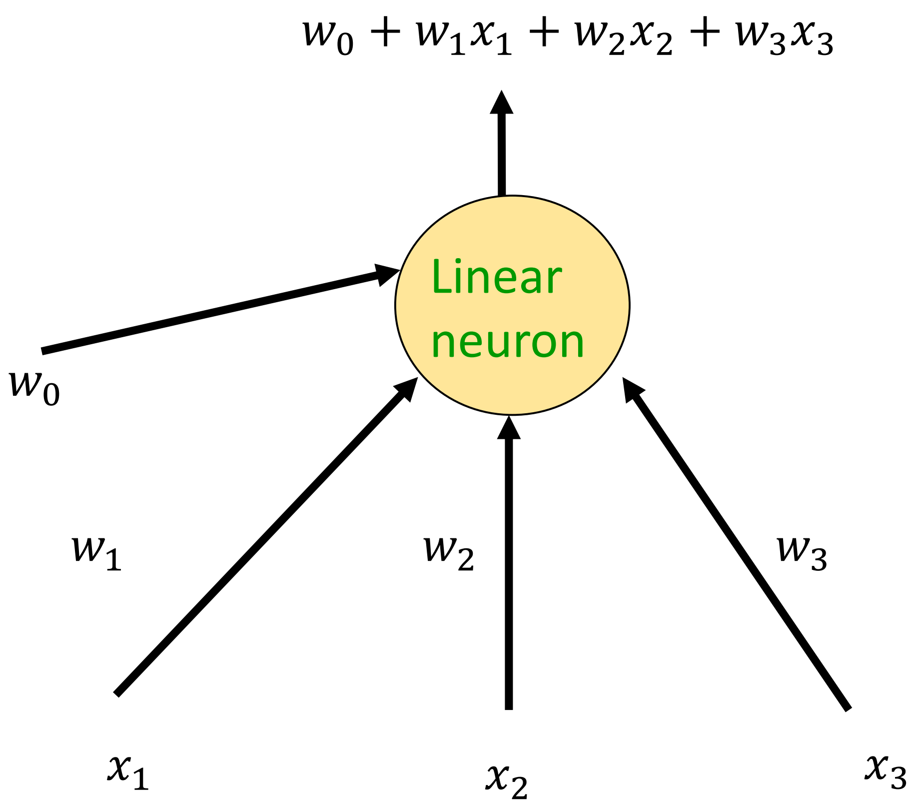
Inspiration: The Brain



Inspiration: The Brain



Linear Neuron



Linear Neuron: Cost Function

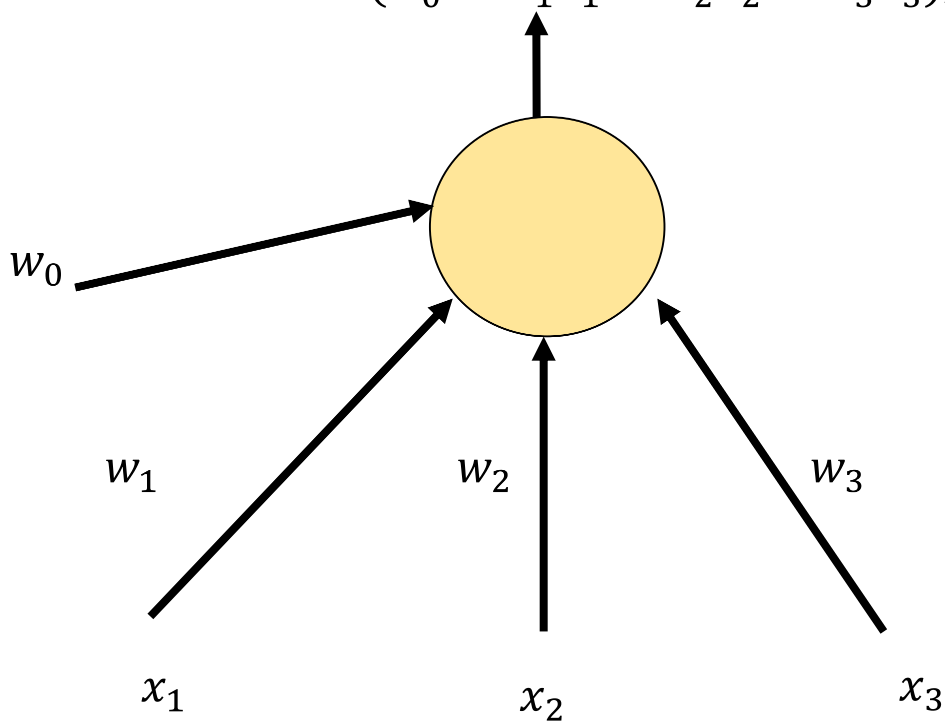
- Any number of choices. The one made for linear regression is

$$\sum_{i=1}^{m}\left(y^{(i)} - w^{T}x^{(i)}\right)^{2}$$

- Can minimize this using gradient descent to obtain the best weights *w* for the training set

Logistic Neuron

$$\sigma(w_0 + w_1 x_1 + w_2 x_2 + w_3 x_3), \sigma(t) = \frac{1}{1 + \exp(-t)}$$



Logistic Neuron: Cost Function

- Could use the quadratic cost function again
- Could use the “log-loss” function to make the neuron perform logistic regression

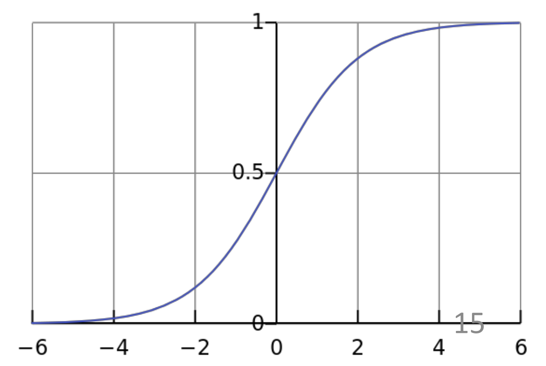
$$-\left(\sum_{i=1}^{m} y^{(i)} \log\left(\frac{1}{1+\exp(-w^T x^{(i)})}\right) + (1-y^{(i)}) \log\left(\frac{\exp(-w^T x^{(i)})}{1+\exp(-w^T x^{(i)})}\right)\right)$$

(Note: we derived this cost function by saying we want to maximize the likelihood of the data under a certain model, but there’s nothing stopping us from just making up a loss function)

Logistic Regression Cost Function: Another Look

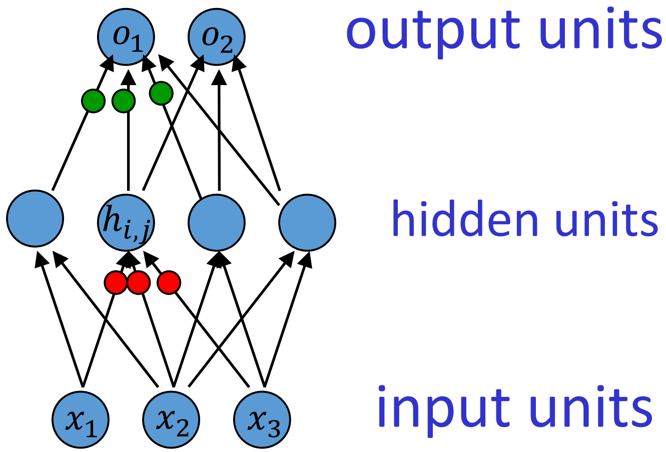
- $Cost(h_w(x), y) = \begin{cases} -\log\left(h_w(x)\right), y = 1 \\ -\log\left(1 - h_w(x)\right), y = 0 \end{cases}$
- If y = 1, want the cost to be small if $h_w(x)$ is close to 1 and large if $h_w(x)$ is close to 0
  - -log(t) is 0 for t=1 and infinity for t = 0
- If y = 0, want the cost to be small if $h_w(x)$ is close to 0 and large if $h_w(x)$ is close to 1
- Note:$0 < \sigma(t) < 1$

$$\sigma(t) = \frac{1}{1 + \exp(-t)}$$

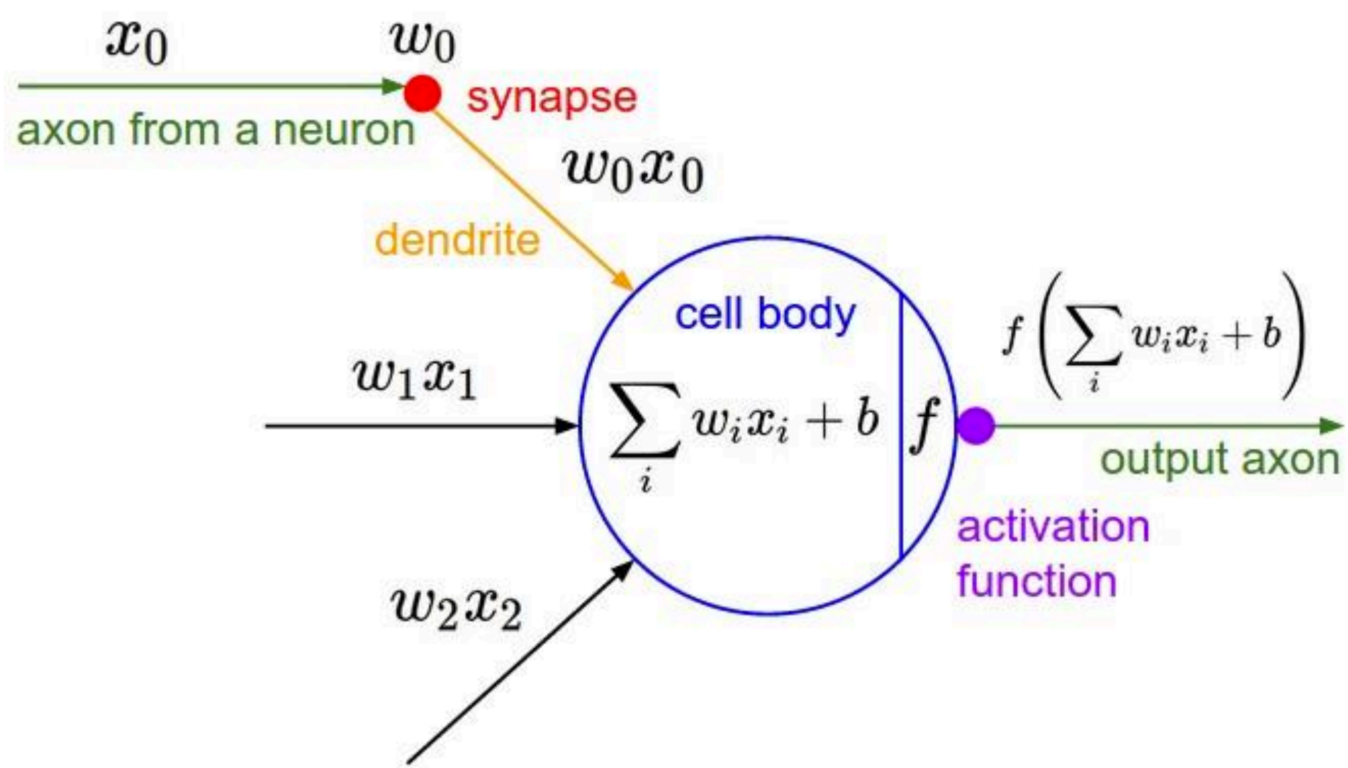


Multilayer Neural Networks

- $h_{i,j} = \mathrm{g}(\mathrm{W}_{i,j}x)$
  $= g(\sum_k W_{i,j,k} x_k)$
- $x_0 = 1$ always
- $W_{i,j,0}$ is the "bias"
- g is the **activation function**
  - Could be $g(t) = t$
  - Could be $g(t) = \sigma(t)$
    - Nobody uses those anymore…



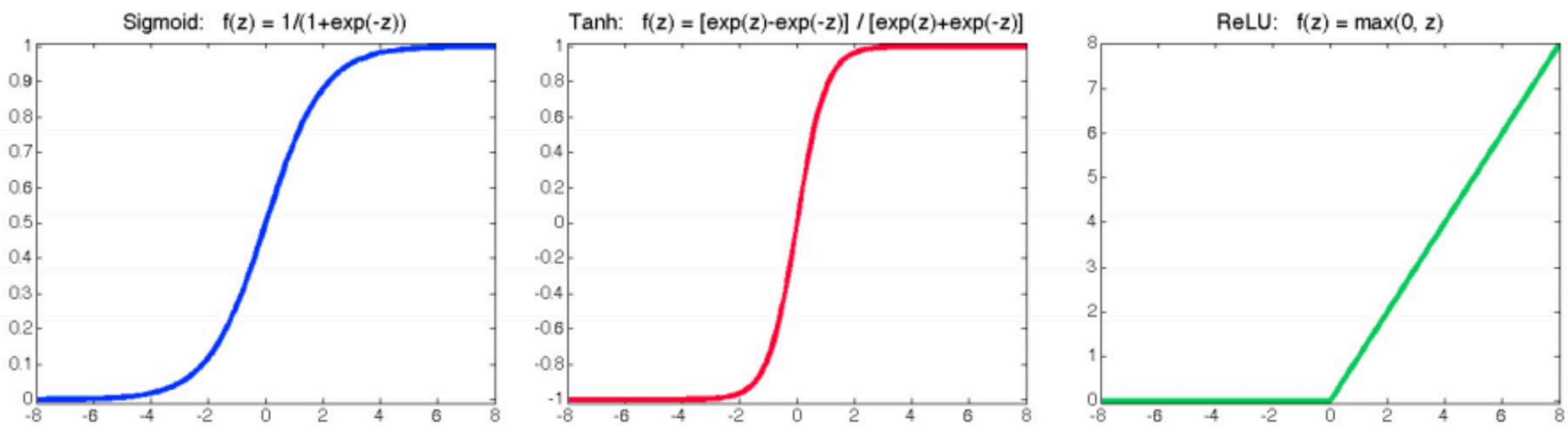
Why do we need activation functions?



Activation functions?

Most commonly used activation functions:

- Sigmoid: $\sigma(z) = \frac{1}{1+\exp(-z)}$
- Tanh: $\tanh(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$
- ReLU (Rectified Linear Unit): $\mathrm{ReLU}(z) = \max(0, z)$

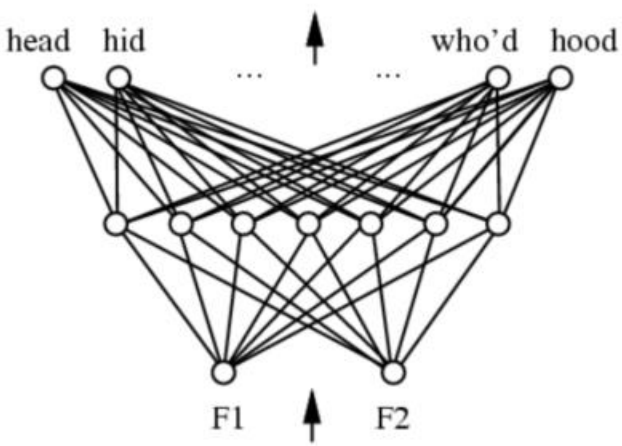


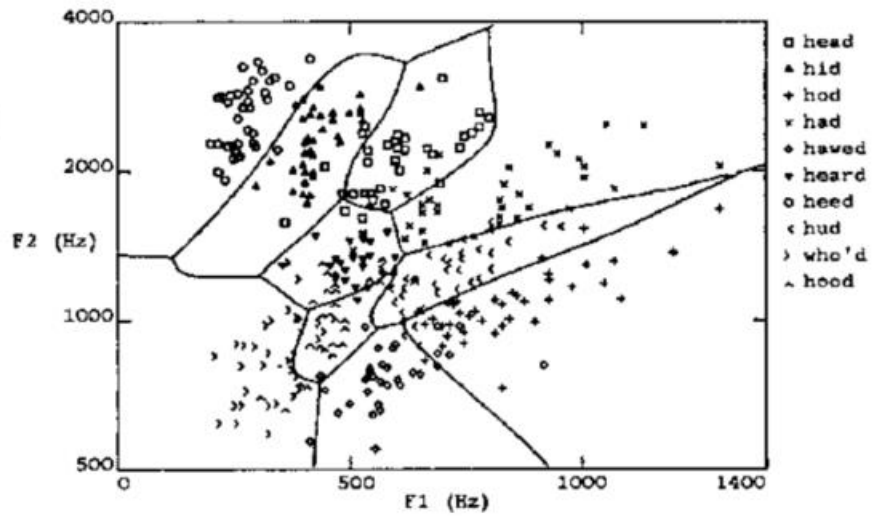
Exercise:

- Hand code a neural network to compute:
  - AND
  - XOR

- Use *only* sigmoid
- Use *only* ReLU activations

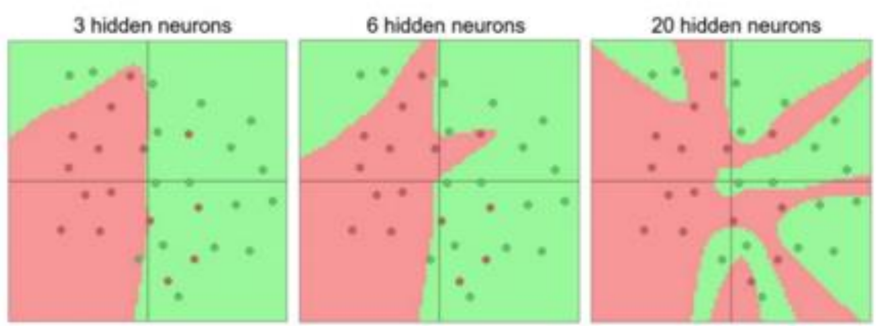
Multilayer Neural Network: Speech Recognition Example (multi-class classification)





Universal Approximator

- Neural networks with at least one hidden layer (and enough neurons) are *universal approximators*
  - Can represent any (smooth) function
- The *capacity* (ability to represent different functions) increases with more hidden layers and more neurons
- Why go deeper? One hidden layer might need *a lot* of neurons. Deeper and narrower networks are more compact

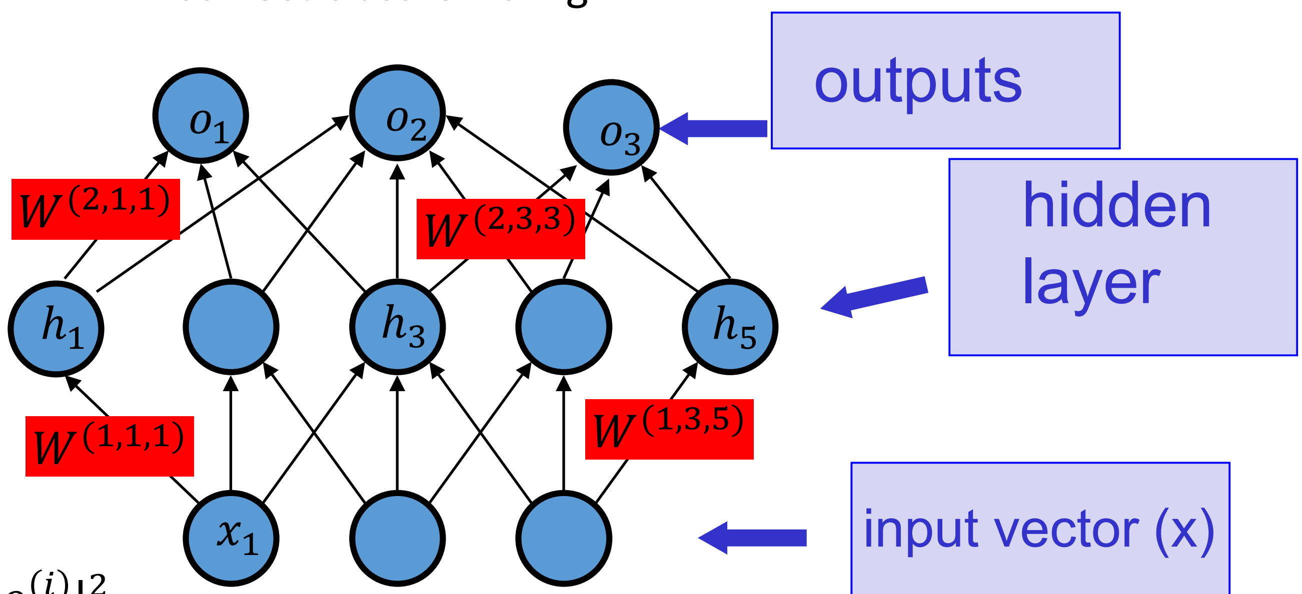


Computation in Neural Networks

- Forward pass
  - Making predictions
  - Plug in the input x, get the output y
- Backward pass
  - Compute the gradient of the cost function with respect to the weights

Multilayer Neural Network for Classification:

o_i is large if the probability that the correct class is *i* is high



A possible cost function:

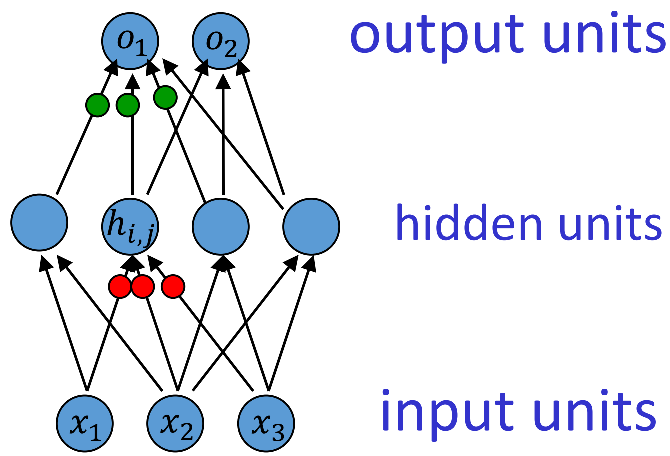
$$C(o,y) = \sum_{i=1}^{m} |y^{(i)} - o^{(i)}|^2$$

$y^{(i)}$'s and $o^{(i)}$'s encoded using one-hot encoding

Forward Pass (vectorized)

$$\mathbf{o} = g\left(\left(W^{(2)}\right)^T \boldsymbol{h} + b^{(2)}\right)$$
$$\mathbf{h} = g\left(\left(W^{(1)}\right)^T \boldsymbol{x} + b^{(1)}\right)$$

…etc… if there are more layers



Backwards Pass (training)

Need to find

$$W = \underset{W}{\operatorname{argmin}} \sum_{i=1}^{m} loss(\boldsymbol{o}^{(i)}, \boldsymbol{y}^{(i)})$$

Where:

- $\boldsymbol{o}^{(i)}$ is the output of the neural network
- $\boldsymbol{y}^{(i)}$ is the ground truth
- W is all the weights in the neural network
- $loss$ is a continuous loss function.

Use **gradient descent** to find a good W

But how to compute gradient?

- To optimize the weights / parameters o the neural network, we need to compute gradient of the cost function:
$C(\mathbf{o}, \mathbf{y}) = \sum_{i=1}^{m} loss(\boldsymbol{o}^{(i)}, \boldsymbol{y}^{(i)})$
with respect to every weight in the neural network.
- Need to compute, for every layer and weight l, j, i :
$$\frac{\partial C}{\partial W^{(l,j,i)}}$$
- How to do this? How to do this **efficiently**?

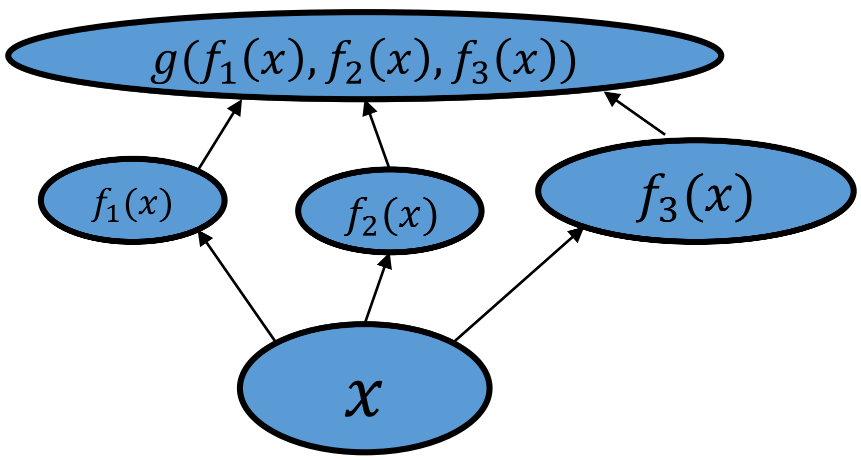
Review: Chain Rule

- Univariate Chain Rule

$$\frac{d}{dt} g\big(f(t)\big) = \frac{dg}{df} \cdot \frac{df}{dt}$$

- Multivariate Chain Rule

$$\frac{\partial g}{\partial x} = \sum \frac{\partial g}{\partial f_i} \frac{\partial f_i}{\partial x}$$



Gradient of Single Weight (last layer)

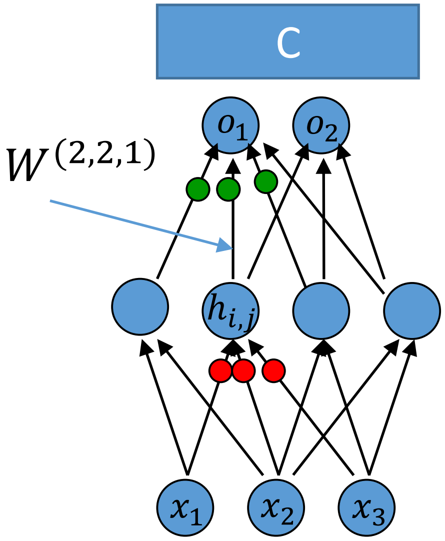
- We need the partial derivatives of the cost function $C(o, y)$ w.r.t all the W and b .
- $o_i = g\left(\sum_j W^{(2,j,i)} h_j + b^{(2,j)}\right)$
- Let $z_i = \sum_j W^{(2,j,i)} h_j + b^{(2,j)}$ so that $o_i = g(z_i)$
- Partial derivative of $C(o, y)$ with respect to $W^{(2,j,i)}$ all evaluated at (x, y, W, b, h, o)

$$\frac{\partial C}{\partial W^{(2,j,i)}} = \frac{\partial o_i}{\partial W^{(2,j,i)}} \frac{\partial C}{\partial o_i}$$

$$= \frac{\partial z_i}{\partial W^{(2,j,i)}} \frac{\partial g}{\partial z_i} \frac{\partial C}{\partial o_i}$$

$$= h_j \frac{\partial g}{\partial z_i} \frac{\partial C}{\partial o_i}$$

$$= h_j g'(z_i) \frac{\partial}{\partial o_i} C(o, y)$$



Gradient of Single Weight (last layer)

- For example, if we use:
  - sigmoid activation $g = \sigma, \sigma'(t) = \sigma(t)\big(1 - \sigma(t)\big)$
  - MSE loss: $C(\boldsymbol{o}, \boldsymbol{y}) = \sum_{i=1}^{N} (o_i - y_i)^2$
- Then

$$\frac{\partial C}{\partial W^{(2,j,i)}} = h_j \boxed{g'(z_i)} \boxed{\frac{\partial C}{\partial o_i}}$$
$$= h_j \boxed{g(z_i)\big(1 - g(z_i)\big)} \boxed{2(o_i - y_i)}$$
$$= h_j \boxed{o_i(1 - o_i)} \boxed{2(o_i - y_i)}$$

Vectorization

- For a single weight, we had:

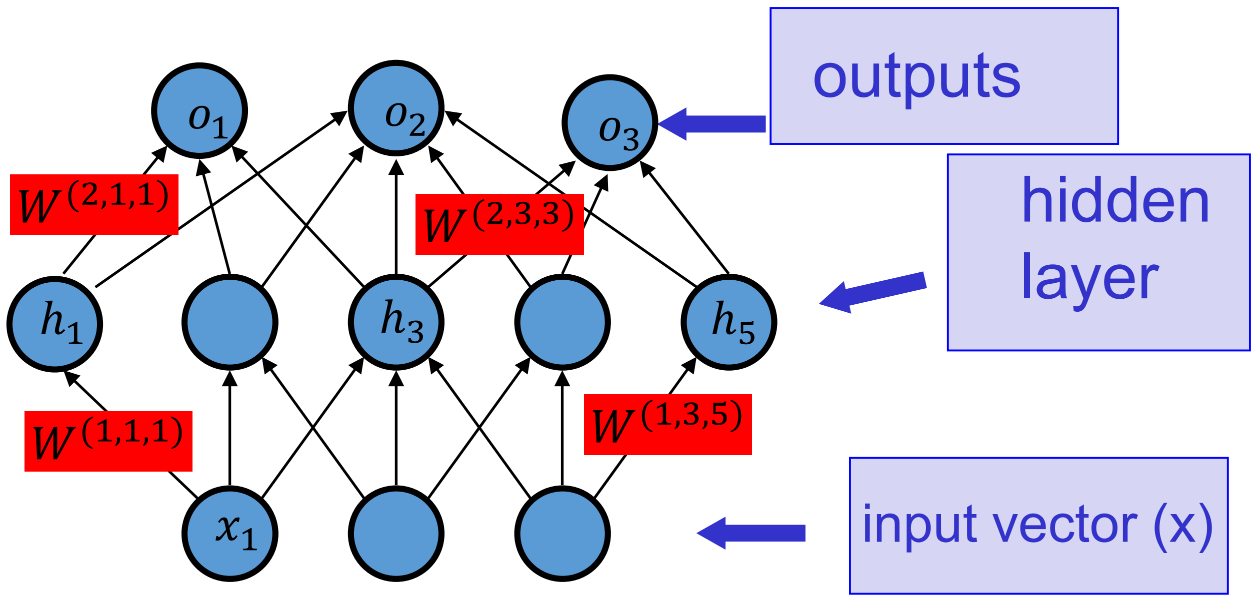
$$\frac{\partial C}{\partial W^{(2,j,i)}} = h_j o_i (1 - o_i) 2 (o_i - y_i)$$

- Vectorizing, we get

$$\frac{\partial C}{\partial \boldsymbol{W}^{(2)}} = 2\boldsymbol{h} \cdot \left(\boldsymbol{o}.(1 - \boldsymbol{o}).(\boldsymbol{o} - \boldsymbol{y})\right)^T$$

- *Note this is for sigmoid activation, square loss*

What about earlier layers?

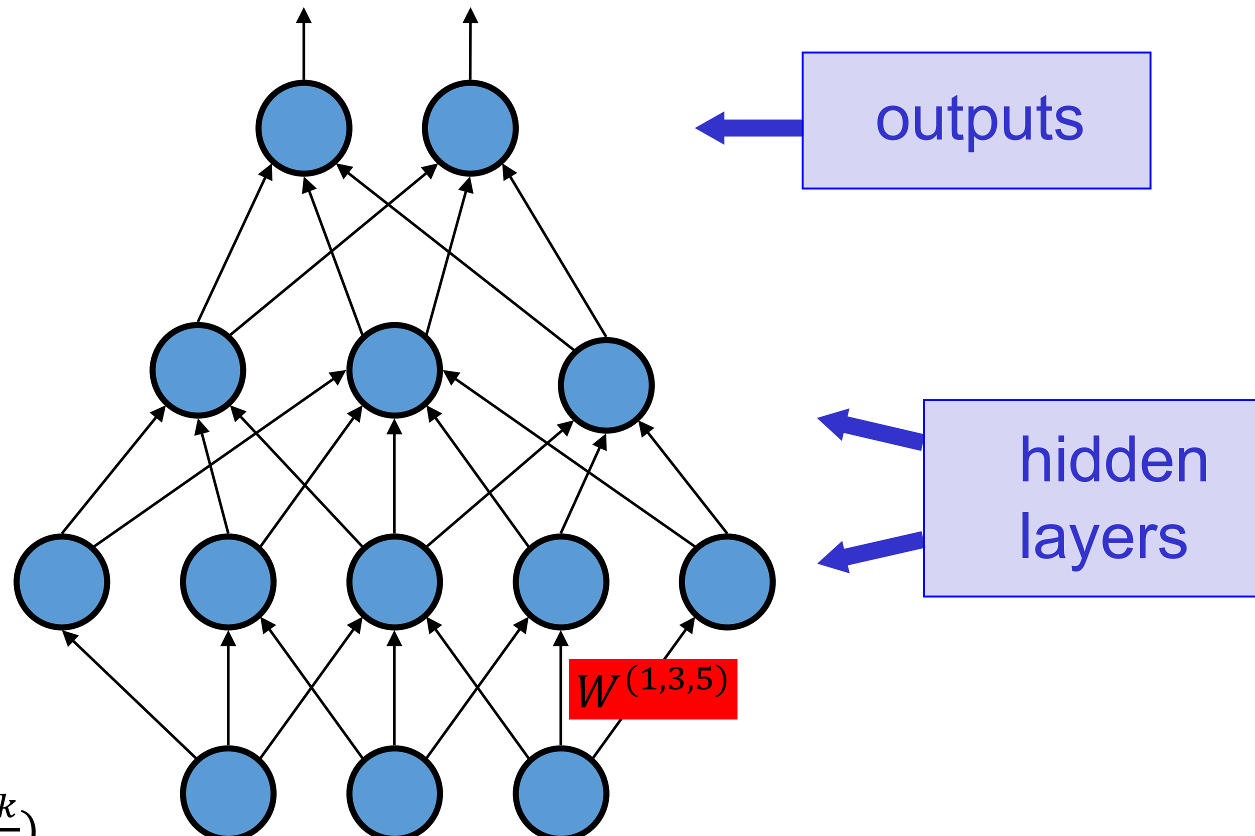


Use multivariate chain rule:

$$\frac{\partial C}{\partial h_i} = \sum_k \left(\frac{\partial C}{\partial o_k} \frac{\partial o_k}{h_i}\right)$$

$$\frac{\partial C}{\partial W^{(1,j,i)}} = \frac{\partial C}{\partial h_i} \frac{\partial h_i}{\partial W^{(1,j,i)}}$$

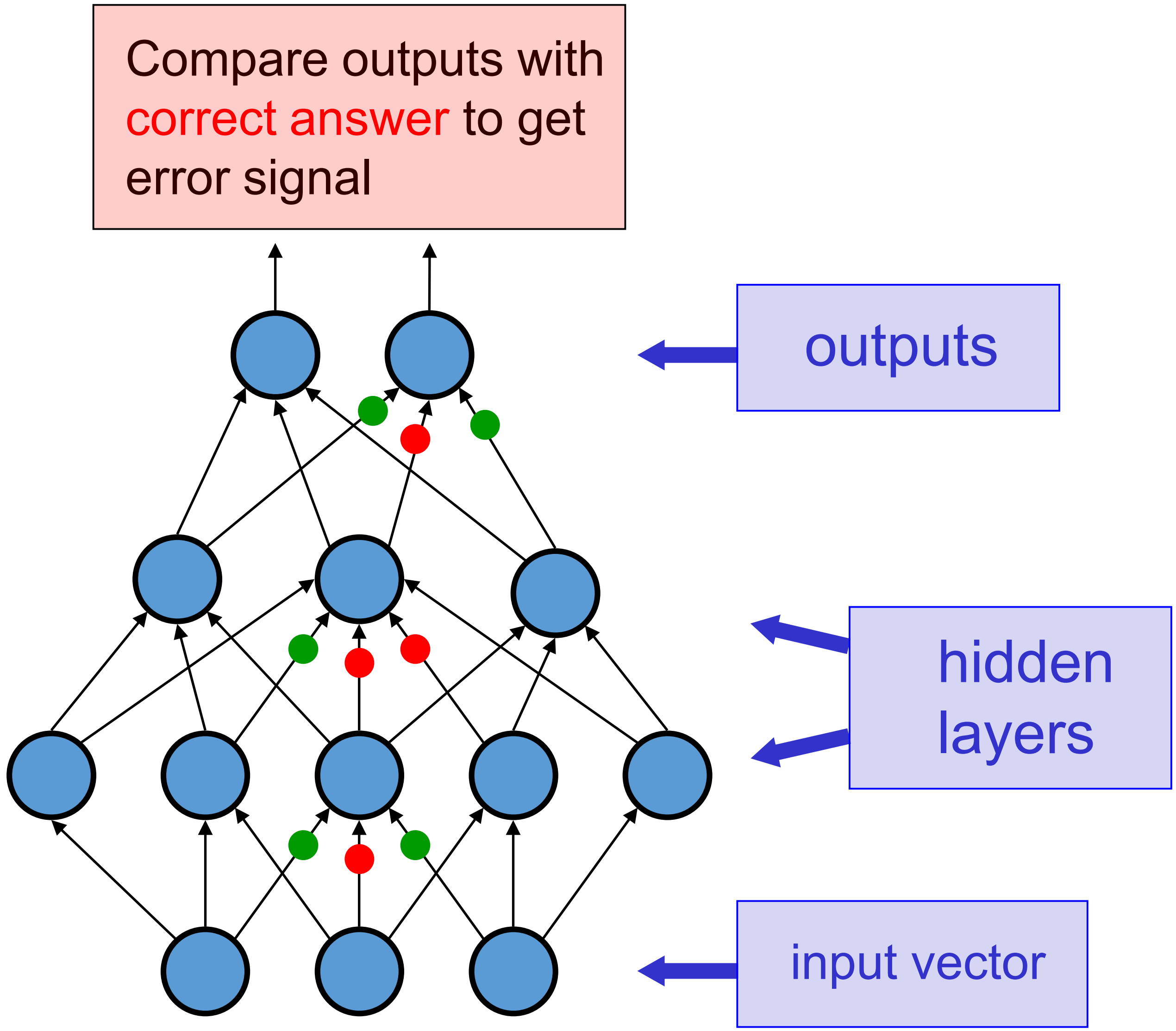
Backpropagation



$$\frac{\partial C}{\partial hidB_i} = \sum_k \left(\frac{\partial C}{\partial hidA_k}\frac{\partial hidA_k}{\partial hidB_i}\right)$$

$$\frac{\partial C}{\partial W^{(1,j,i)}} = \frac{\partial C}{\partial hidB_i}\frac{\partial hidB_i}{\partial W^{(1,j,i)}}$$

Back-propagate error signal to get derivatives for learning



Training Summary

Training neural nets:

Loop until convergence:

- for each example n
  1. Given input $\mathbf{x}^{(n)}$, propagate activity forward ($\mathbf{x}^{(n)} \rightarrow \mathbf{h}^{(n)} \rightarrow o^{(n)}$) (**forward pass**)
  2. Propagate gradients backward (**backward pass**)
  3. Update each weight (via gradient descent)

Why is training neural networks so hard?

- Hard to optimize:
  - Not convex
  - Local minima, saddle points, etc…
  - Can take a long time to train
- Architecture choices:
  - How many layers? How many units in each layer?
  - What activation function to use?
- Choice of optimizer:
  - We talked about gradient descent, but there are techniques that improves upon gradient descent

Demo

http://playground.tensorflow.org/