

Review: Logistic Regression

- Maximum likelihood estimation:

$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

with:

$$P(y^{(i)} = +1 | x^{(i)}; w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$
$$P(y^{(i)} = -1 | x^{(i)}; w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}$$

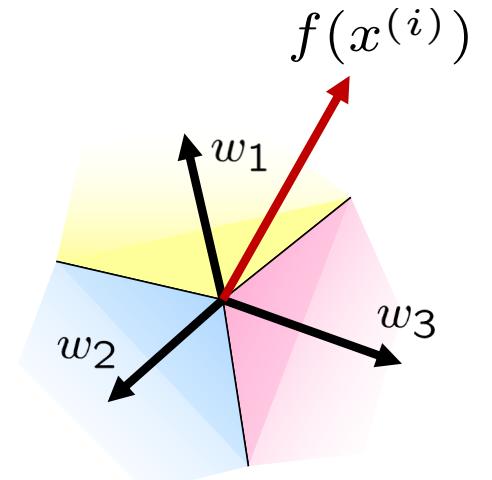
Review: Multi-Class Logistic Regression

- Maximum likelihood estimation:

$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

with: $P(y^{(i)} | x^{(i)}; w) = \frac{e^{w_{y^{(i)}} \cdot f(x^{(i)})}}{\sum_y e^{w_y \cdot f(x^{(i)})}}$

Suppose $y^{(i)} = 3 \dots$



What's still needed

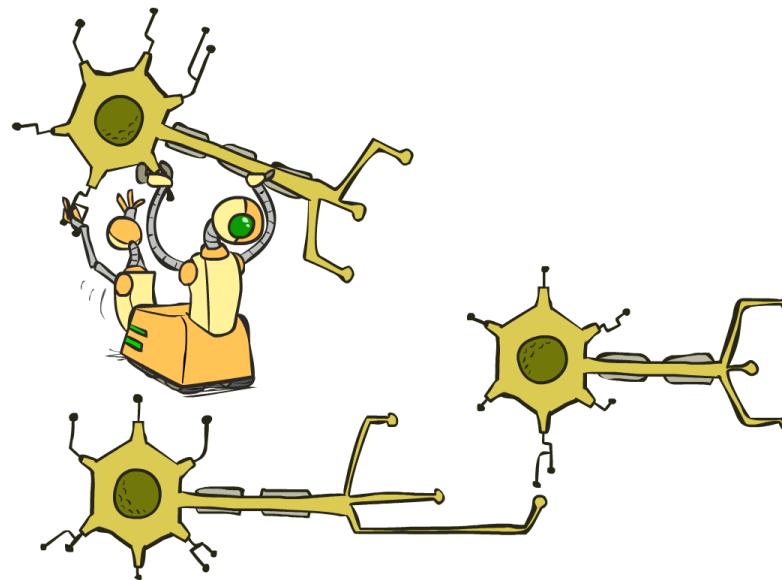
- Optimization

- i.e., how do we solve:

$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

CS 188: Artificial Intelligence

Optimization and Neural Nets



Spring 2024 --- University of California, Berkeley

[These slides were created by Dan Klein and Pieter Abbeel for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at <http://ai.berkeley.edu>.]

Review: Derivatives and Gradients

- What is the derivative of the function $g(x) = x^2 + 3$?

$$\frac{dg}{dx} = 2x$$

- What is the derivative of $g(x)$ at $x=5$?

$$\frac{dg}{dx}|_{x=5} = 10$$

Review: Derivatives and Gradients

- What is the gradient of the function $g(x, y) = x^2y$?
 - Recall: Gradient is a vector of partial derivatives with respect to each variable

$$\nabla g = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 2xy \\ x^2 \end{bmatrix}$$

- What is the derivative of $g(x, y)$ at $x=0.5, y=0.5$?

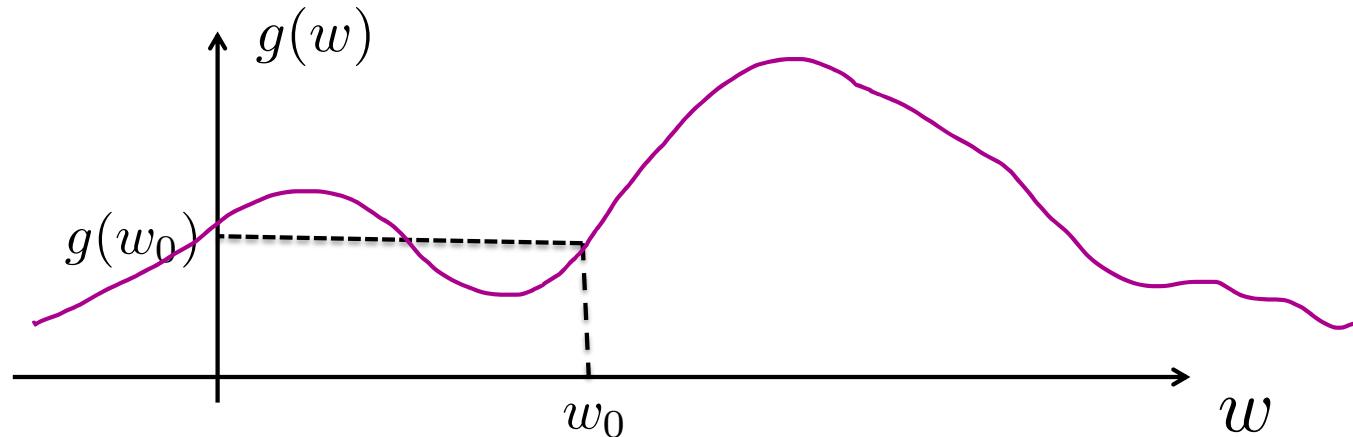
$$\nabla g|_{x=0.5, y=0.5} = \begin{bmatrix} 2(0.5)(0.5) \\ (0.5^2) \end{bmatrix} = \begin{bmatrix} 0.5 \\ 0.25 \end{bmatrix}$$

Hill Climbing

- Recall from local search: simple, general idea
 - Start wherever
 - Repeat: move to the best neighboring state
 - If no neighbors better than current, quit
- What's particularly tricky when hill-climbing for multiclass logistic regression?
 - Optimization over a continuous space
 - Infinitely many neighbors!
 - How to do this efficiently?

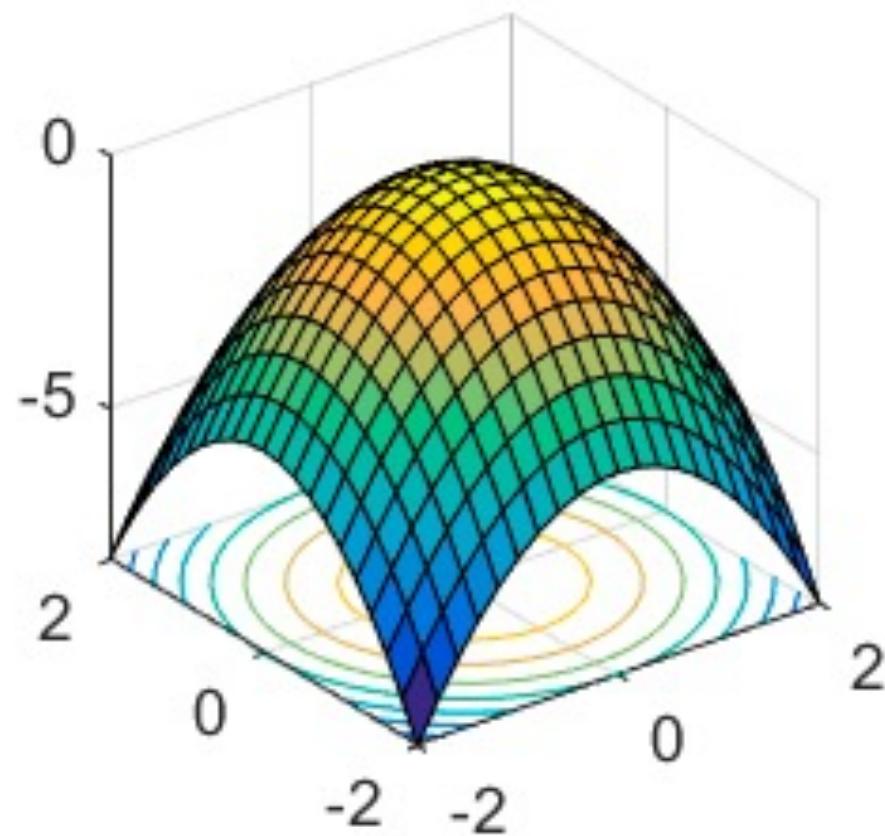


1-D Optimization



- Could evaluate $g(w_0 + h)$ and $g(w_0 - h)$
 - Then step in best direction
- Or, evaluate derivative: $\frac{\partial g(w_0)}{\partial w} = \lim_{h \rightarrow 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}$
 - Tells which direction to step into

2-D Optimization



Source: offconvex.org

Gradient Ascent

- Perform update in uphill direction for each coordinate
- The steeper the slope (i.e. the higher the derivative) the bigger the step for that coordinate
- E.g., consider: $g(w_1, w_2)$

- Updates:

$$w_1 \leftarrow w_1 + \alpha * \frac{\partial g}{\partial w_1}(w_1, w_2)$$

$$w_2 \leftarrow w_2 + \alpha * \frac{\partial g}{\partial w_2}(w_1, w_2)$$

- Updates in vector notation:

$$w \leftarrow w + \alpha * \nabla_w g(w)$$

with: $\nabla_w g(w) = \begin{bmatrix} \frac{\partial g}{\partial w_1}(w) \\ \frac{\partial g}{\partial w_2}(w) \end{bmatrix}$ = **gradient**

Gradient Ascent

- Idea:
 - Start somewhere
 - Repeat: Take a step in the gradient direction

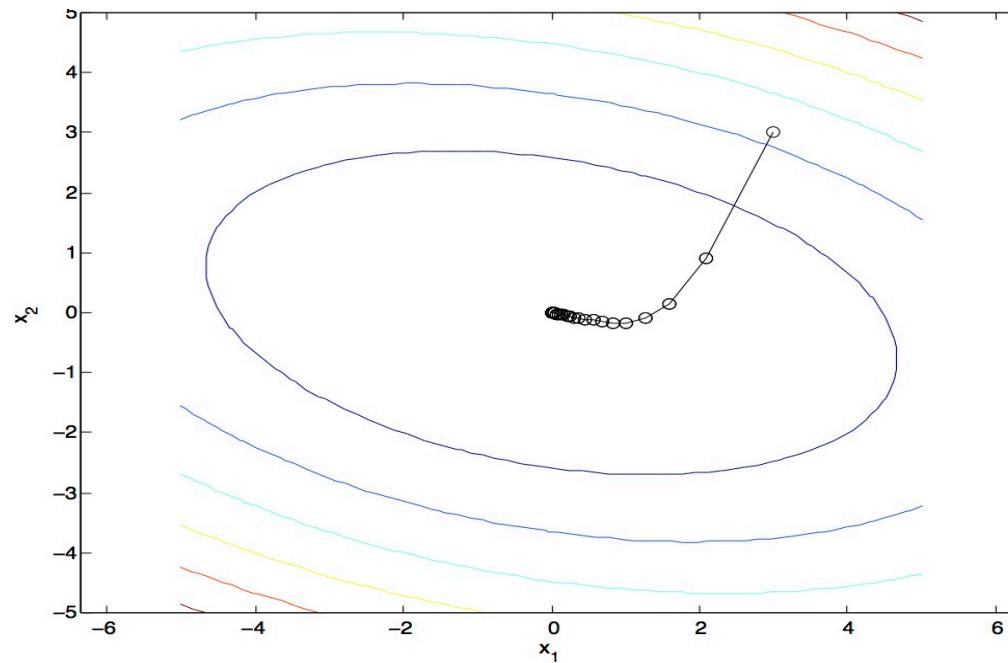


Figure source: Mathworks

Gradient in n dimensions

$$\nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\ \frac{\partial g}{\partial w_2} \\ \vdots \\ \frac{\partial g}{\partial w_n} \end{bmatrix}$$

Optimization Procedure: Gradient Ascent

- init w
- for iter = 1, 2, ...

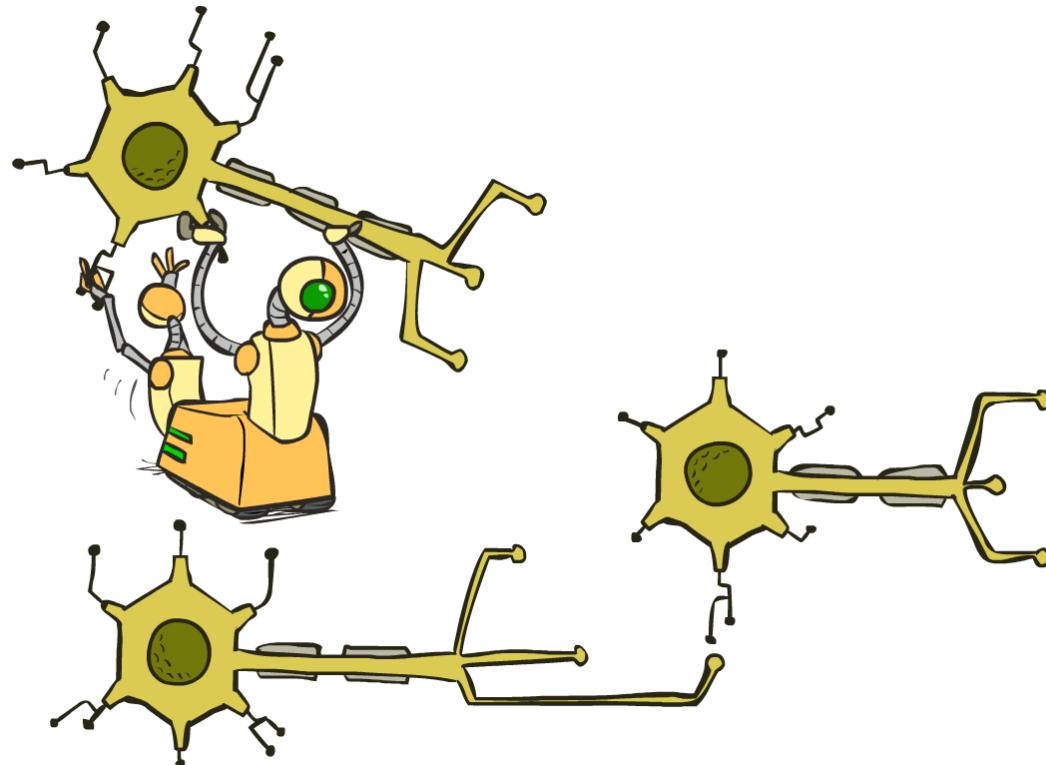
$$w \leftarrow w + \alpha * \nabla g(w)$$

- α : learning rate --- tweaking parameter that needs to be chosen carefully
- How? Try multiple choices
 - Crude rule of thumb: update changes w about 0.1 – 1 %

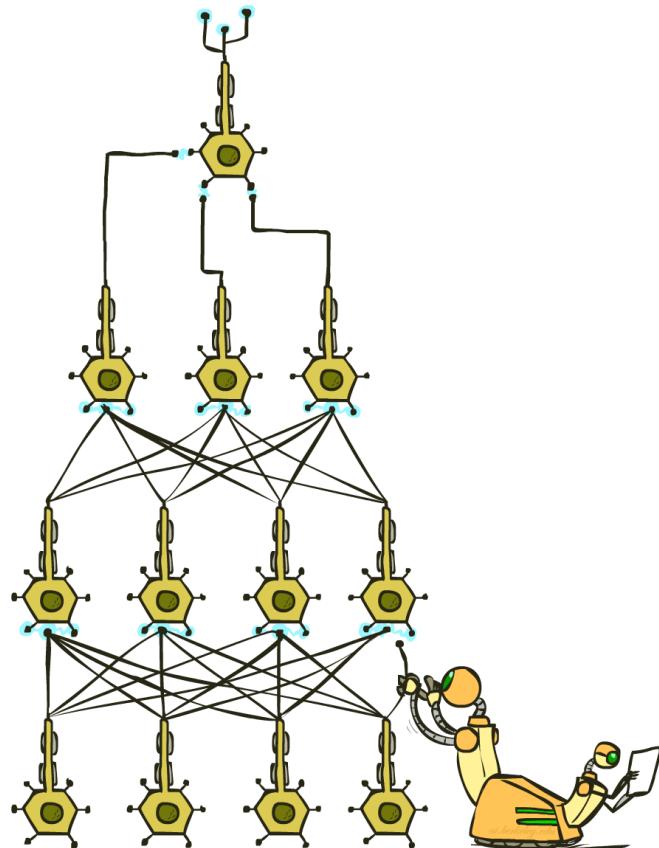
How about computing all the derivatives?

- We'll talk about that once we covered neural networks, which are a generalization of logistic regression

Neural Network Architectures

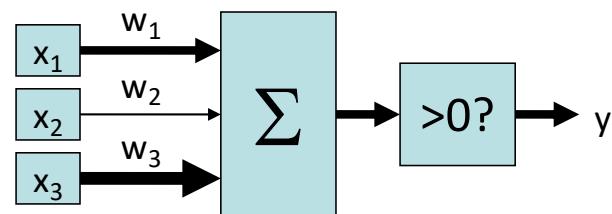


Manual Feature Design vs. Deep Learning



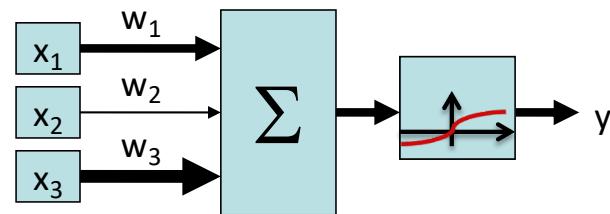
- Manual feature design requires:
 - Domain-specific expertise
 - Domain-specific effort
- What if we could learn the features, too?
 - **Deep Learning**

Review: Perceptron



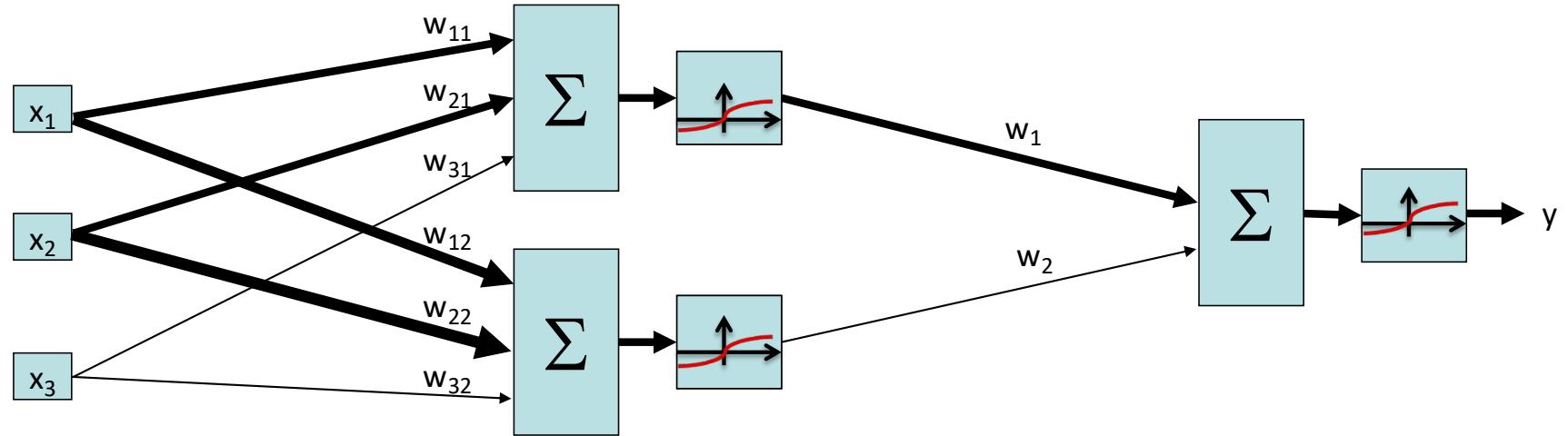
$$y = \begin{cases} 1 & w_1x_1 + w_2x_2 + w_3x_3 > 0 \\ 0 & \text{otherwise} \end{cases}$$

Review: Perceptron with Sigmoid Activation

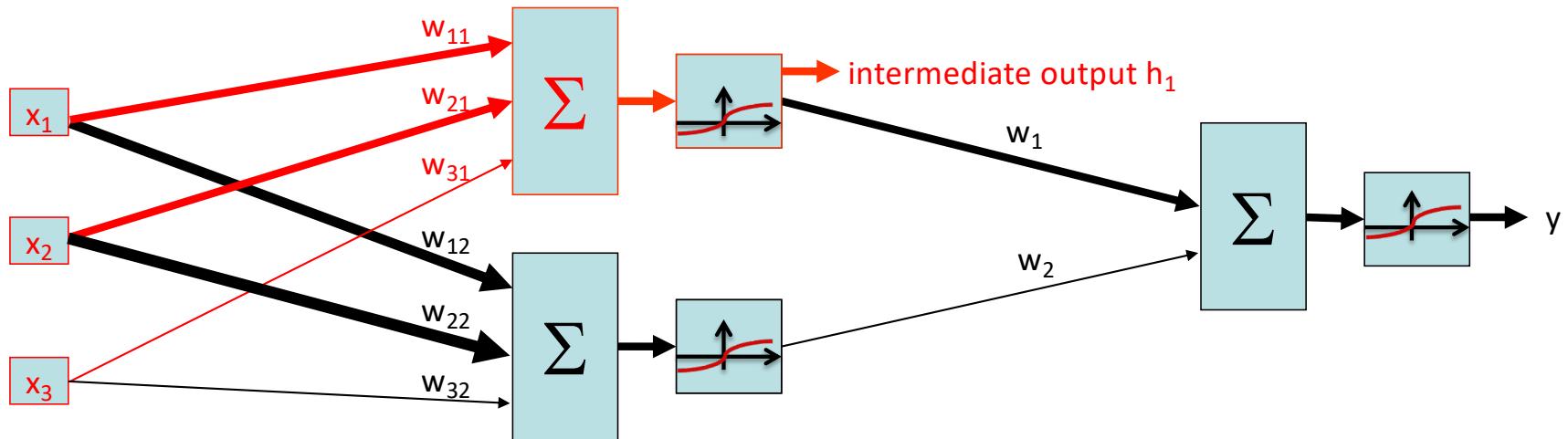


$$\begin{aligned}y &= \phi(w_1x_1 + w_2x_2 + w_3x_3) \\&= \frac{1}{1 + e^{-(w_1x_1 + w_2x_2 + w_3x_3)}}\end{aligned}$$

2-Layer, 2-Neuron Neural Network



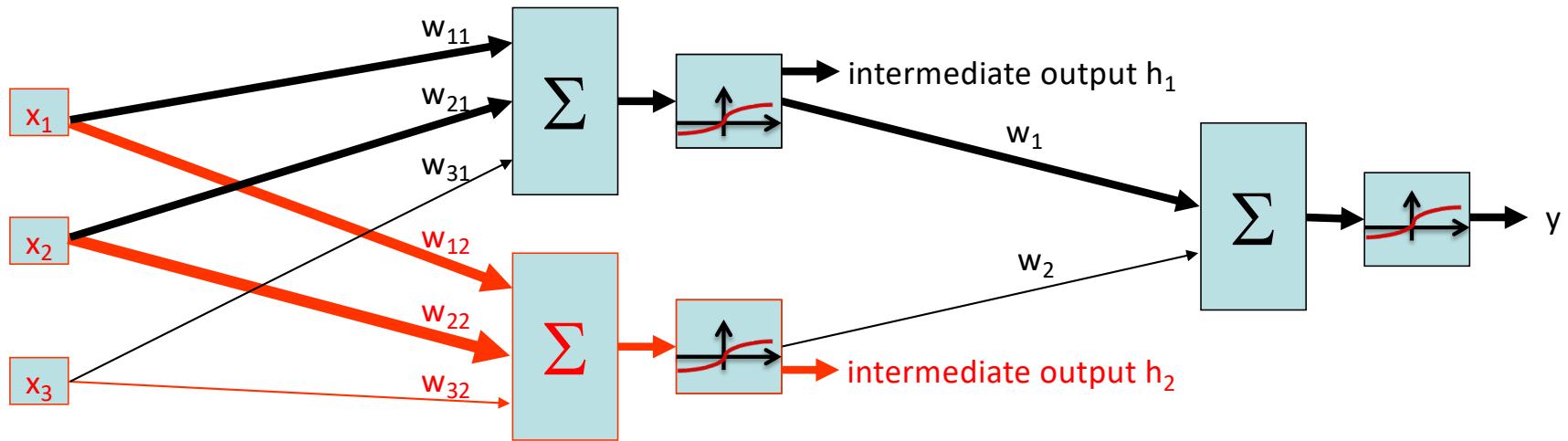
2-Layer, 2-Neuron Neural Network



$$\text{intermediate output } h_1 = \phi(w_{11}x_1 + w_{21}x_2 + w_{31}x_3)$$

$$= \frac{1}{1 + e^{-(w_{11}x_1 + w_{21}x_2 + w_{31}x_3)}}$$

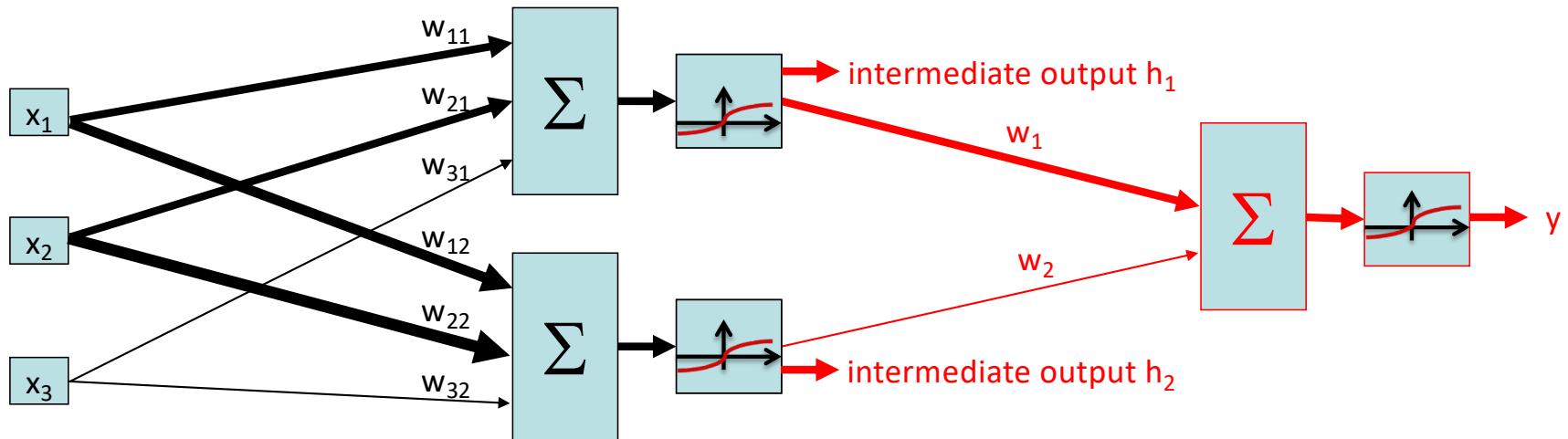
2-Layer, 2-Neuron Neural Network



$$\text{intermediate output } h_2 = \phi(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)$$

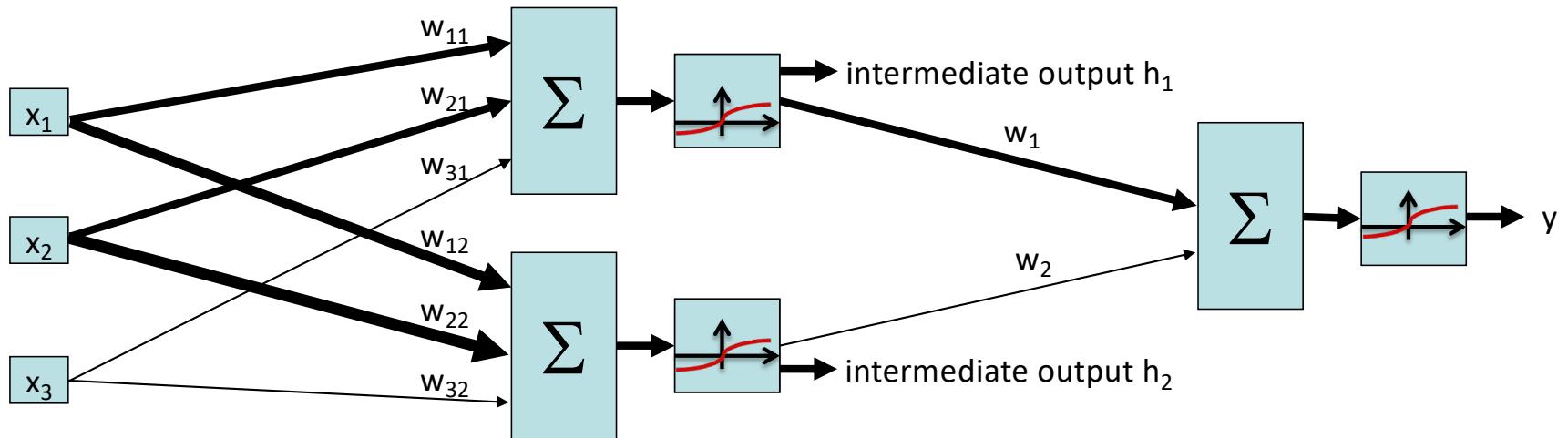
$$= \frac{1}{1 + e^{-(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)}}$$

2-Layer, 2-Neuron Neural Network



$$\begin{aligned}y &= \phi(w_1 h_1 + w_2 h_2) \\&= \frac{1}{1 + e^{-(w_1 h_1 + w_2 h_2)}}\end{aligned}$$

2-Layer, 2-Neuron Neural Network



$$\begin{aligned}y &= \phi(w_1 h_1 + w_2 h_2) \\&= \phi(w_1 \phi(w_{11}x_1 + w_{21}x_2 + w_{31}x_3) + w_2 \phi(w_{12}x_1 + w_{22}x_2 + w_{32}x_3))\end{aligned}$$

2-Layer, 2-Neuron Neural Network

$$\begin{aligned}y &= \phi(w_1 h_1 + w_2 h_2) \\&= \phi(w_1 \phi(w_{11}x_1 + w_{21}x_2 + w_{31}x_3) + w_2 \phi(w_{12}x_1 + w_{22}x_2 + w_{32}x_3))\end{aligned}$$

The same equation, formatted with matrices:

$$\begin{aligned}&\phi \left(\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \right) \\&= \phi \left(\begin{bmatrix} w_{11}x_1 + w_{21}x_2 + w_{31}x_3 & w_{12}x_1 + w_{22}x_2 + w_{32}x_3 \end{bmatrix} \right) \\&= \begin{bmatrix} h_1 & h_2 \end{bmatrix}\end{aligned}$$

$$\phi \left(\begin{bmatrix} h_1 & h_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \right) = \phi(w_1 h_1 + w_2 h_2) = y$$

The same equation, formatted more compactly by introducing variables representing each matrix:

$$\phi(x \times W_{\text{layer 1}}) = h \quad \phi(h \times W_{\text{layer 2}}) = y$$

2-Layer, 2-Neuron Neural Network

$$\phi(x \times W_{\text{layer 1}}) = h$$

Shape (1, 3).
Input feature vector.

Shape (3, 2).
Weights to be learned.

Shape (1, 2).
Outputs of layer 1,
inputs to layer 2.

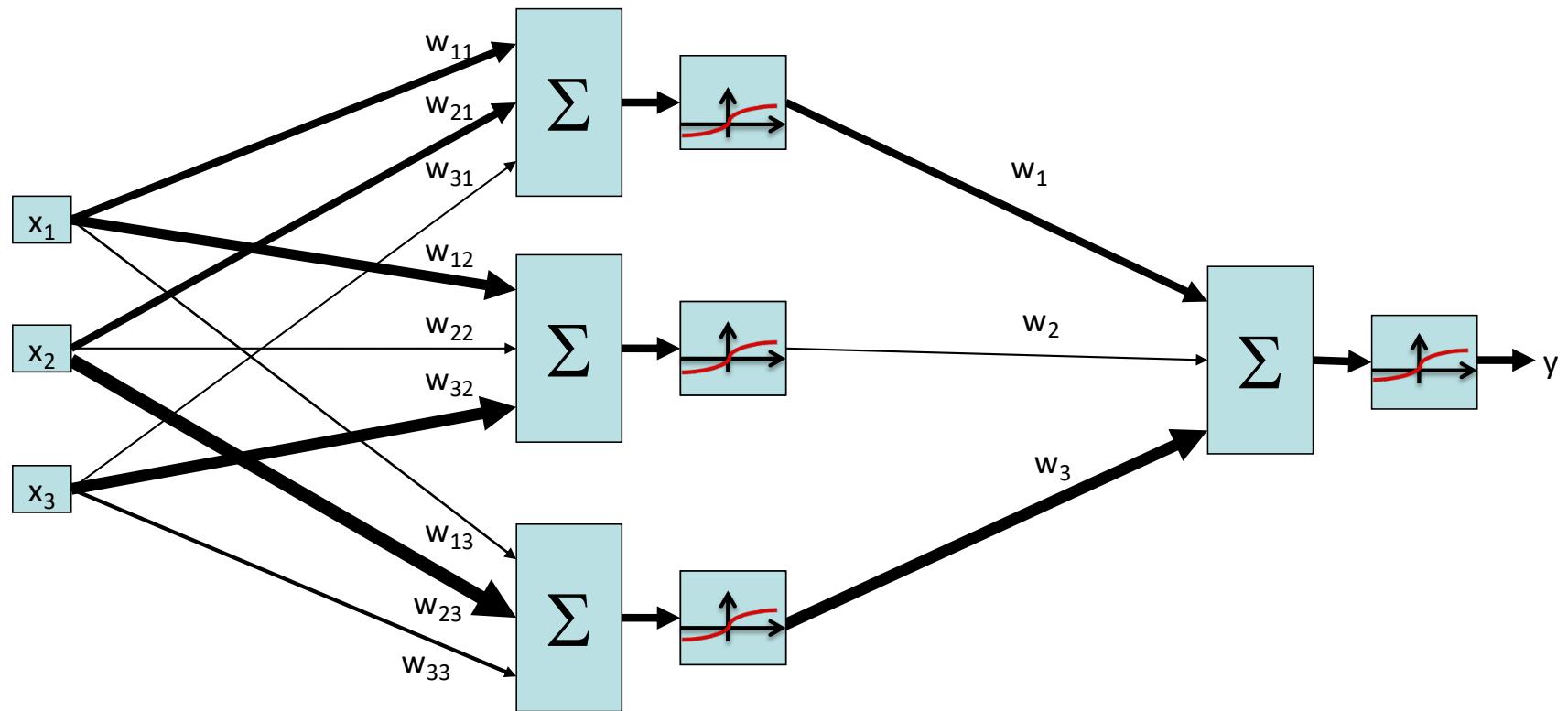
$$\phi(h \times W_{\text{layer 2}}) = y$$

Shape (1, 2).
Outputs of layer 1,
inputs to layer 2.

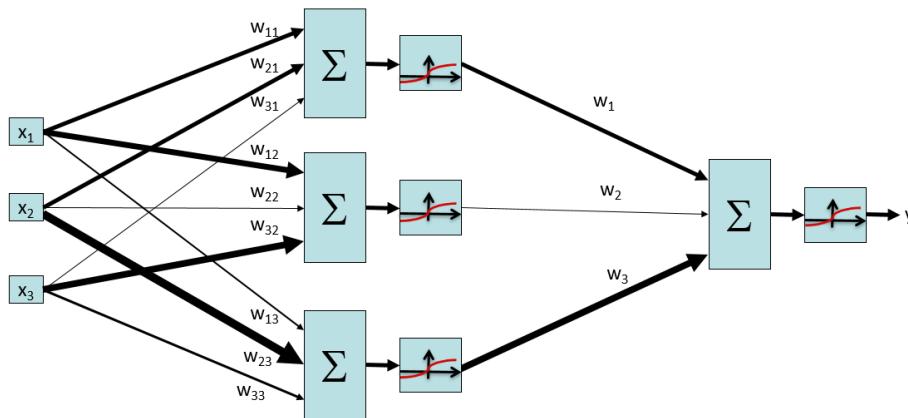
Shape (2, 1).
Weights to be learned.

Shape (1, 1).
Output of network.

2-Layer, 3-Neuron Neural Network



2-Layer, 3-Neuron Neural Network



$$\begin{aligned}
 & \phi \left(\begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \right) \\
 &= \phi \left(\begin{bmatrix} w_{11}x_1 + w_{21}x_2 + w_{31}x_3 & w_{12}x_1 + w_{22}x_2 + w_{32}x_3 & w_{13}x_1 + w_{23}x_2 + w_{33}x_3 \end{bmatrix} \right) \\
 &= \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix}
 \end{aligned}$$

$$\phi \left(\begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \right) = \phi(w_1h_1 + w_2h_2 + w_3h_3) = y$$

2-Layer, 3-Neuron Neural Network

$$\phi(x \times W_{\text{layer 1}}) = h$$

Shape (1, 3).
Input feature vector.

Shape (3, 3).
Weights to be learned

Shape (1, 3).
Outputs of layer 1,
inputs to layer 2.

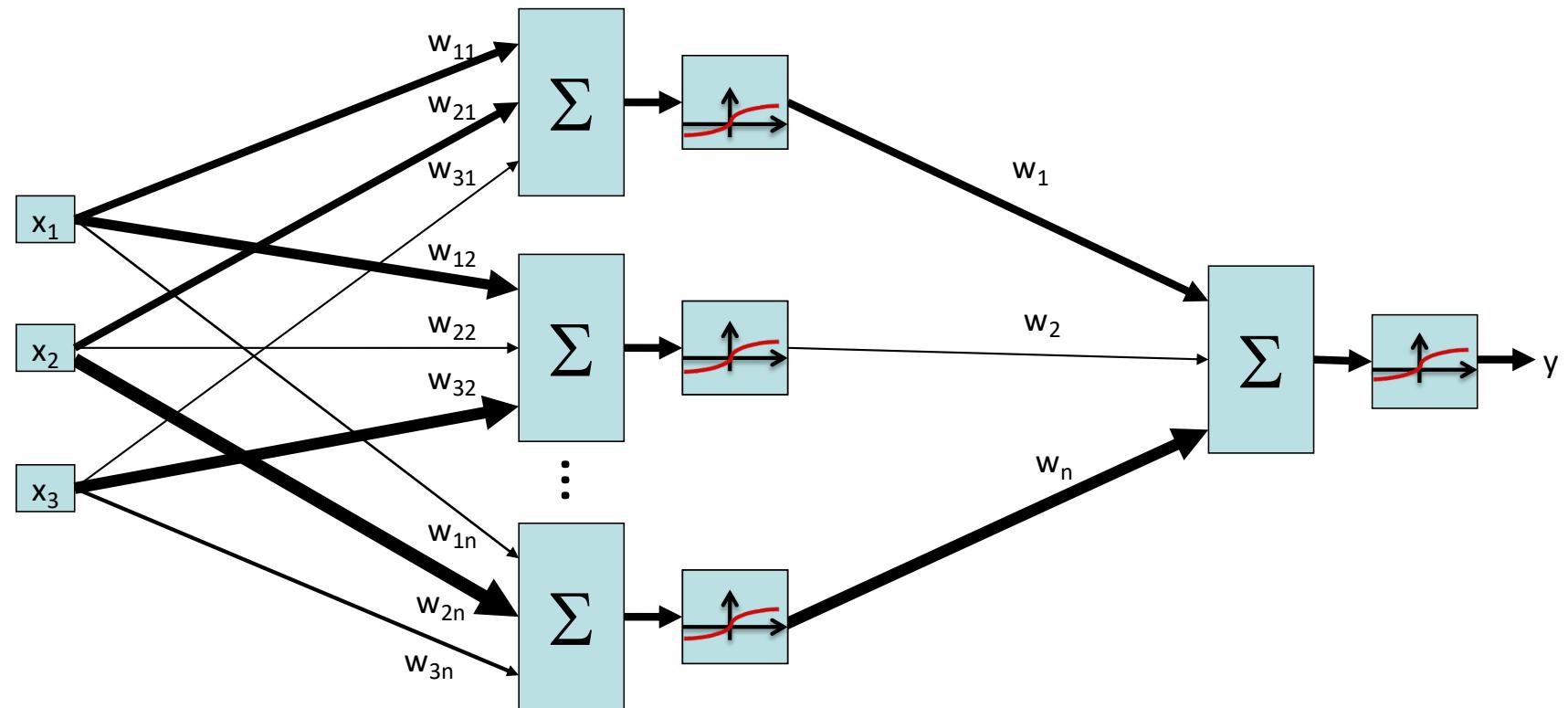
$$\phi(h \times W_{\text{layer 2}}) = y$$

Shape (1, 3).
Outputs of layer 1,
inputs to layer 2.

Shape (3, 1).
Weights to be learned.

Shape (1, 1).
Output of network.

Generalize: Number of hidden neurons



The hidden layer could have any arbitrary number n neurons.

Generalize: Number of hidden neurons

$$\phi(x \times W_{\text{layer 1}}) = h$$

Shape (1, 3).
Input feature vector.

Shape (3, n).
Weights to be learned

Shape (1, n).
Outputs of layer 1,
inputs to layer 2.

$$\phi(h \times W_{\text{layer 2}}) = y$$

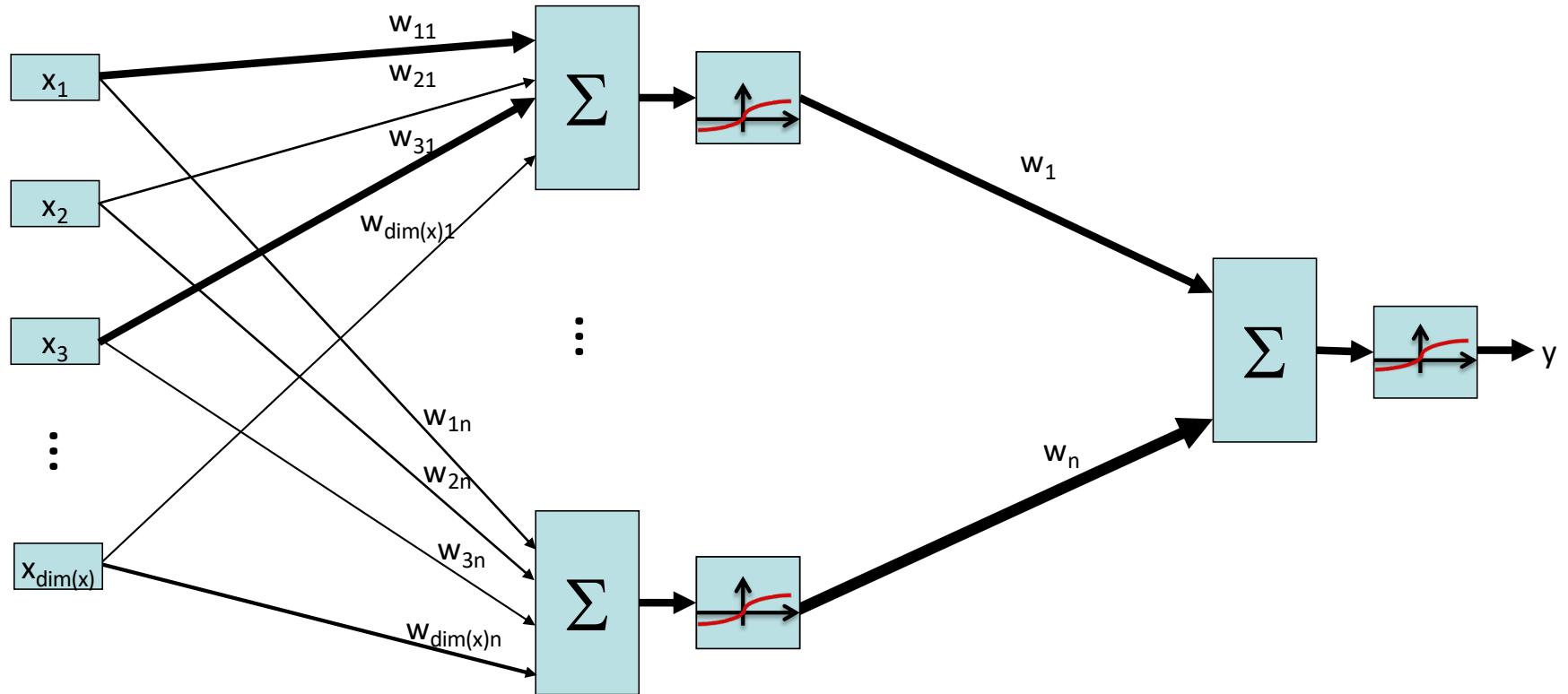
Shape (1, n).
Outputs of layer 1,
inputs to layer 2.

Shape (n , 1).
Weights to be learned.

Shape (1, 1).
Output of network.

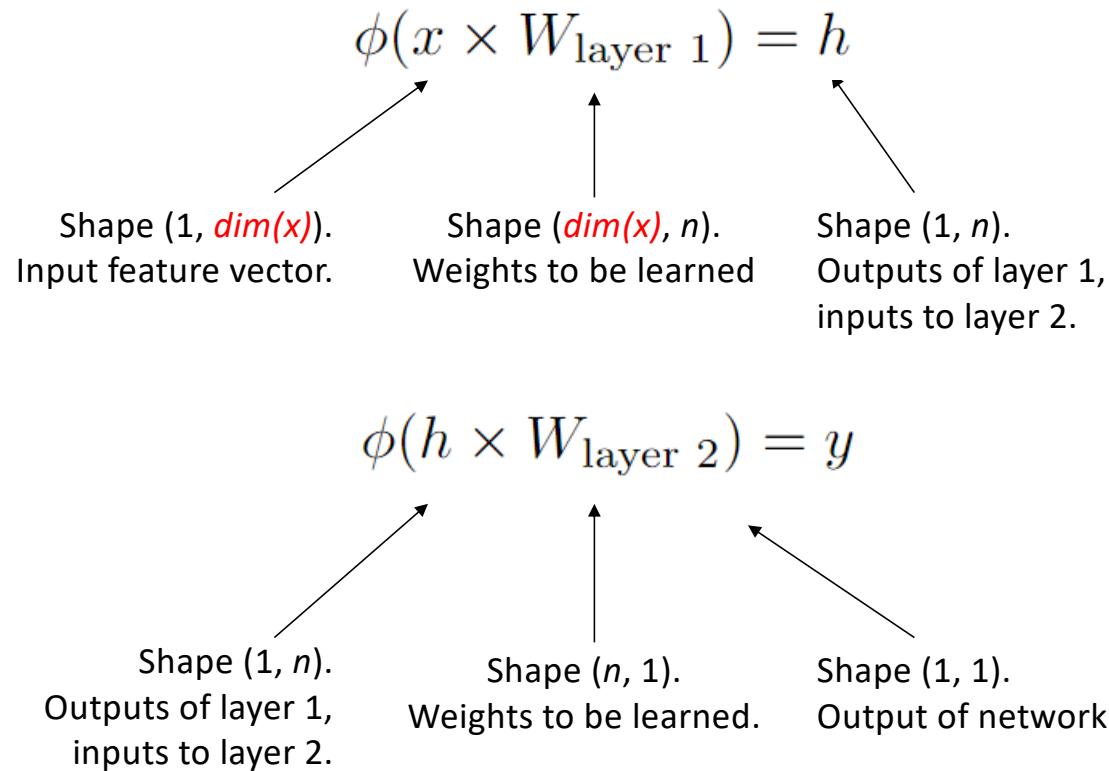
The hidden layer could have any arbitrary number n neurons.

Generalize: Number of input features



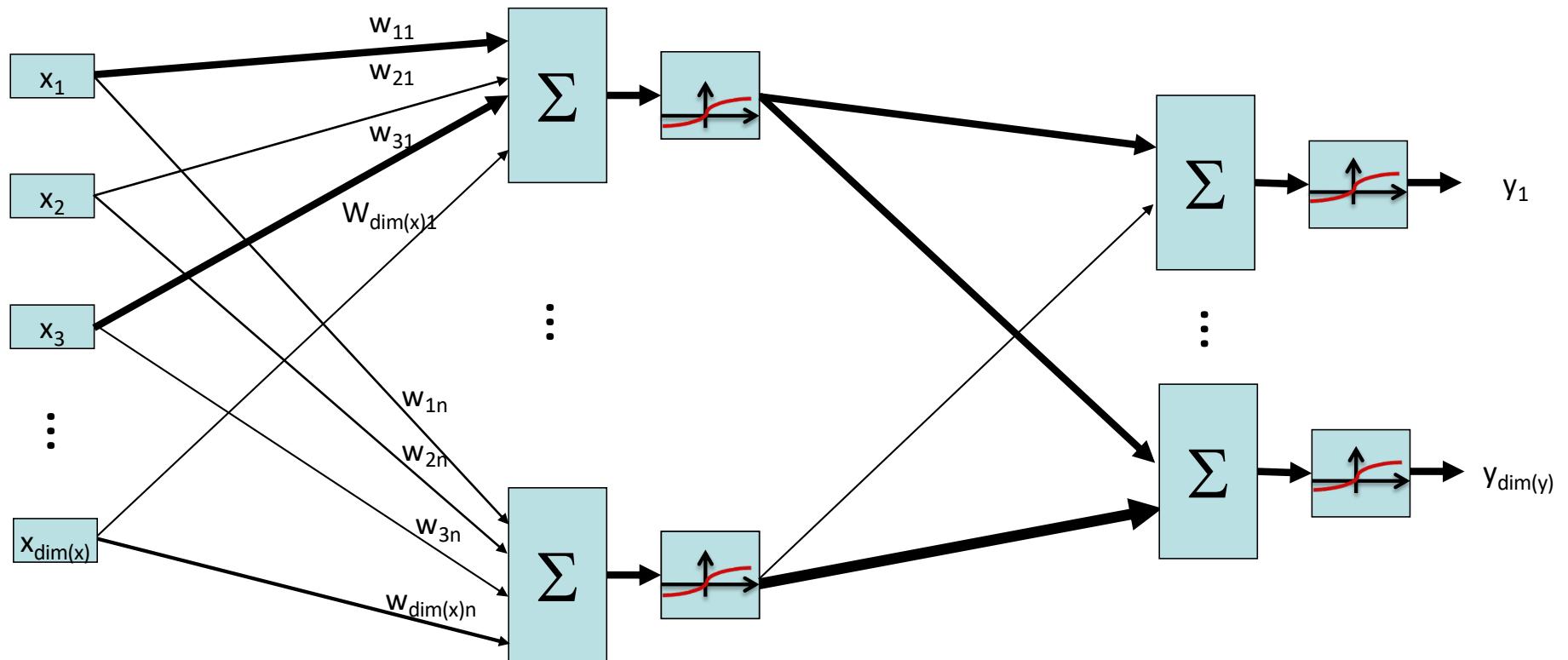
The input feature vector doesn't necessarily need to have 3 features; it could have some arbitrary number $\dim(x)$ of features.

Generalize: Number of input features



The input feature vector doesn't necessarily need to have 3 features; it could have some arbitrary number $\text{dim}(x)$ of features.

Generalize: Number of outputs



The output doesn't necessarily need to be just one number; it could be some arbitrary $\text{dim}(y)$ length vector.

Generalize: Number of outputs

$$\phi(x \times W_{\text{layer 1}}) = h$$

Shape $(1, \dim(x))$.
Input feature vector.

Shape $(\dim(x), n)$.
Weights to be learned

Shape $(1, n)$.
Outputs of layer 1,
inputs to layer 2.

$$\phi(h \times W_{\text{layer 2}}) = y$$

Shape $(1, n)$.
Outputs of layer 1,
inputs to layer 2.

Shape $(n, \dim(y))$.
Weights to be learned.

Shape $(1, \dim(y))$.
Output of network.

The output doesn't necessarily need to be just one number; it could be some arbitrary $\dim(y)$ length vector.

Generalized 2-Layer Neural Network

$$\phi(x \times W_{\text{layer 1}}) = h$$

Shape (1, $\dim(x)$).
Input feature vector.

Shape ($\dim(x), n$).
Weights to be learned

Shape (1, n).
Outputs of layer 1,
inputs to layer 2.

Layer 1 has weight matrix with shape ($\dim(x), n$).
These are the weights for n neurons, each taking
 $\dim(x)$ features as input.

This transforms a $\dim(x)$ -dimensional input
vector into an n -dimensional output vector.

$$\phi(h \times W_{\text{layer 2}}) = y$$

Shape (1, n).
Outputs of layer 1,
inputs to layer 2.

Shape ($n, \dim(y)$).
Weights to be learned.

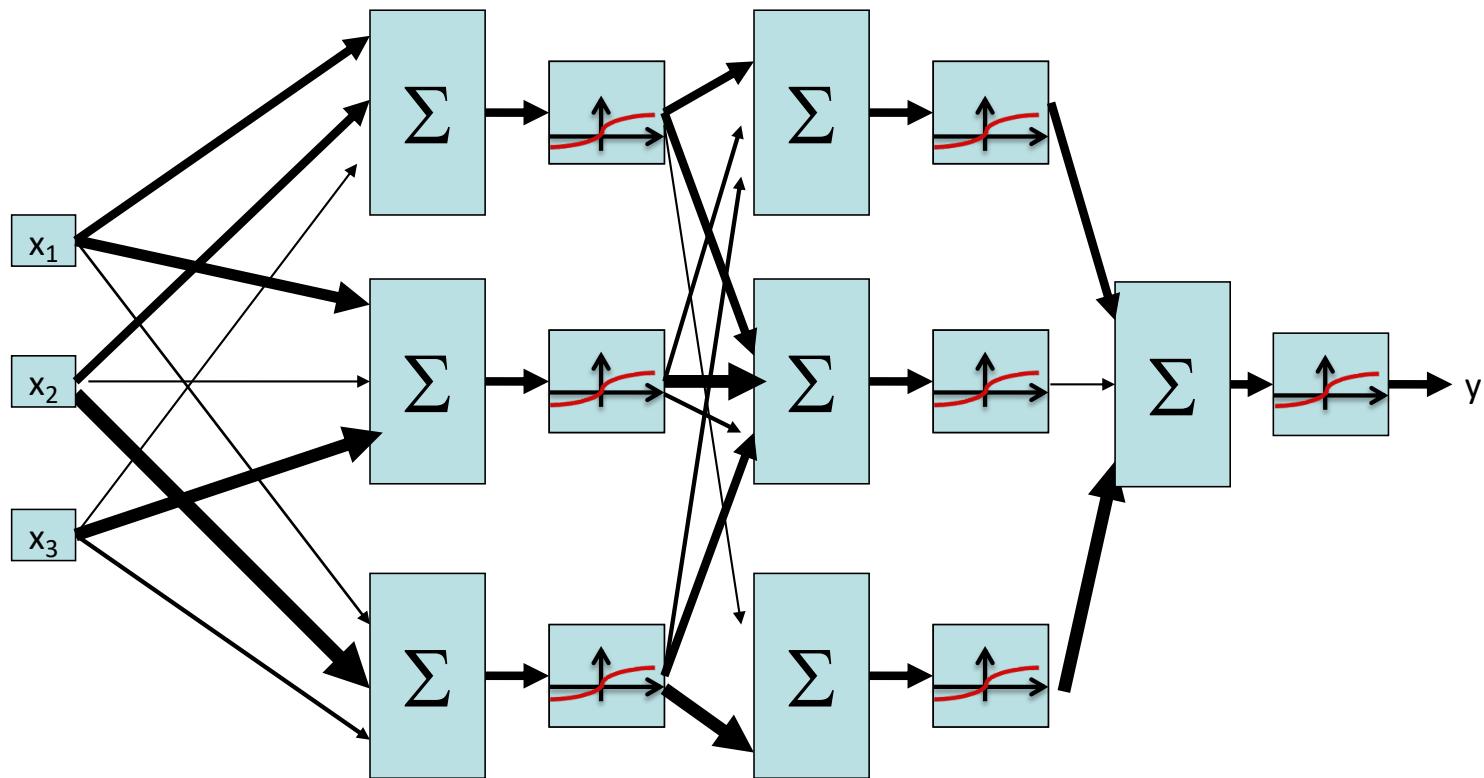
Shape (1, $\dim(y)$).
Output of network.

Layer 2 has weight matrix with shape ($n, \dim(y)$).
These are the weights for $\dim(y)$ neurons, each taking
 n features as input.

This transforms an n -dimensional input vector
into a $\dim(y)$ -dimensional output vector.

The shape of a weight matrix is determined by the dimensions of the input and output of that layer.

3-Layer, 3-Neuron Neural Network



3-Layer, 3-Neuron Neural Network

- Layer 1:

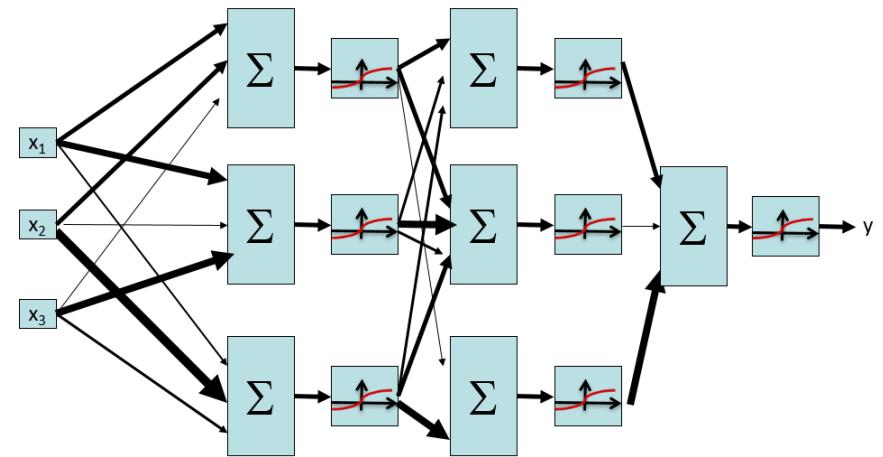
- x has shape $(1, 3)$. Input vector, 3-dimensional.
- $W_{\text{layer } 1}$ has shape $(3, 3)$. Weights for 3 neurons, each taking in a 3-dimensional input vector.
- $h_{\text{layer } 1}$ has shape $(1, 3)$. Outputs of the 3 neurons at this layer.

- Layer 2:

- $h_{\text{layer } 1}$ has shape $(1, 3)$. Outputs of the 3 neurons from the previous layer.
- $W_{\text{layer } 2}$ has shape $(3, 3)$. Weights for 3 new neurons, each taking in the 3 previous perceptron outputs.
- $h_{\text{layer } 2}$ has shape $(1, 3)$. Outputs of the 3 new neurons at this layer.

- Layer 3:

- $h_{\text{layer } 2}$ has shape $(1, 3)$. Outputs from the previous layer.
- $W_{\text{layer } 3}$ has shape $(3, 1)$. Weights for 1 final neuron, taking in the 3 previous perceptron outputs.
- y has shape $(1, 1)$. Output of the final neuron.



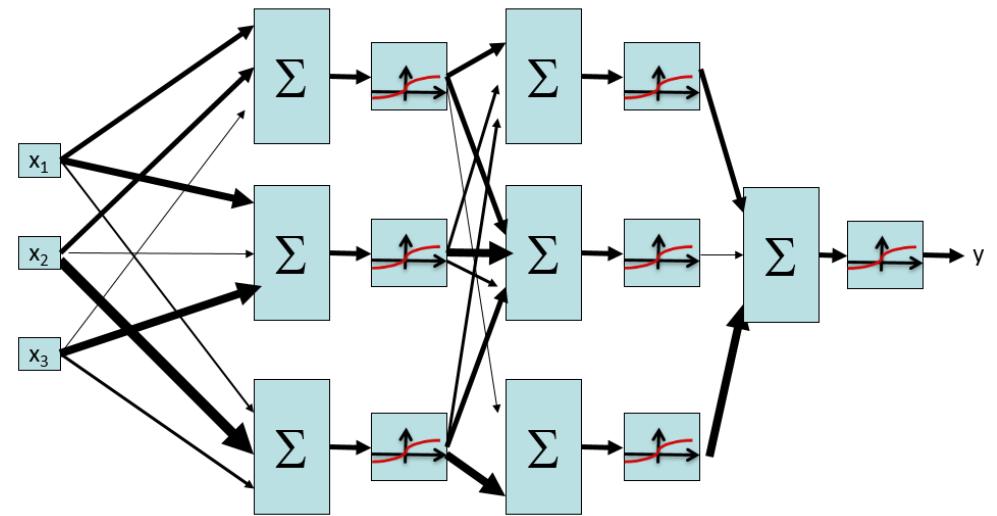
$$\phi(x \times W_{\text{layer } 1}) = h_{\text{layer } 1}$$

$$\phi(h_{\text{layer } 1} \times W_{\text{layer } 2}) = h_{\text{layer } 2}$$

$$\phi(h_{\text{layer } 2} \times W_{\text{layer } 3}) = y$$

Generalized 3-Layer Neural Network

- Layer 1:
 - x has shape $(1, \dim(x))$
 - $W_{\text{layer } 1}$ has shape $(\dim(x), \dim(L1))$
 - $h_{\text{layer } 1}$ has shape $(1, \dim(L1))$
- Layer 2:
 - $h_{\text{layer } 1}$ has shape $(1, \dim(L1))$
 - $W_{\text{layer } 2}$ has shape $(\dim(L1), \dim(L2))$
 - $h_{\text{layer } 2}$ has shape $(1, \dim(L2))$
- Layer 3:
 - $h_{\text{layer } 2}$ has shape $(1, \dim(L2))$
 - $W_{\text{layer } 3}$ has shape $(\dim(L2), \dim(y))$
 - y has shape $(1, \dim(y))$

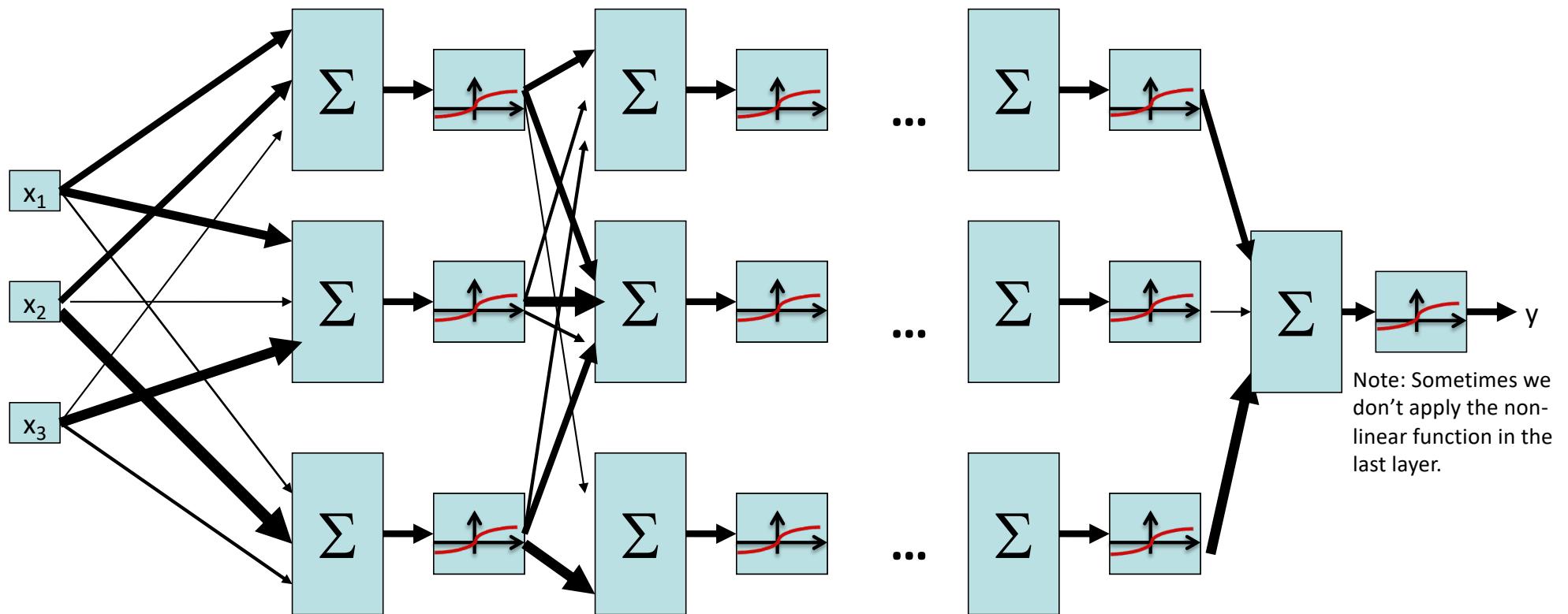


$$\phi(x \times W_{\text{layer } 1}) = h_{\text{layer } 1}$$

$$\phi(h_{\text{layer } 1} \times W_{\text{layer } 2}) = h_{\text{layer } 2}$$

$$\phi(h_{\text{layer } 2} \times W_{\text{layer } 3}) = y$$

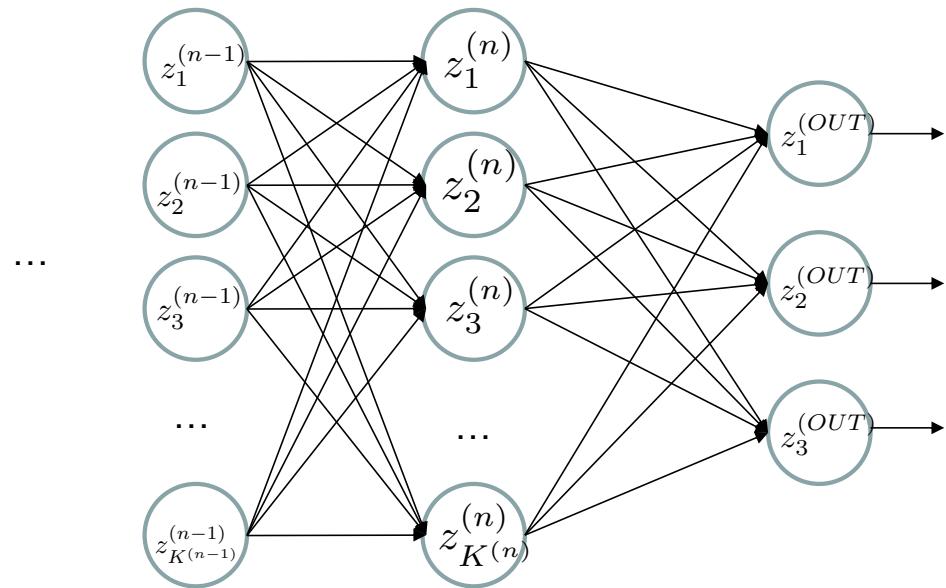
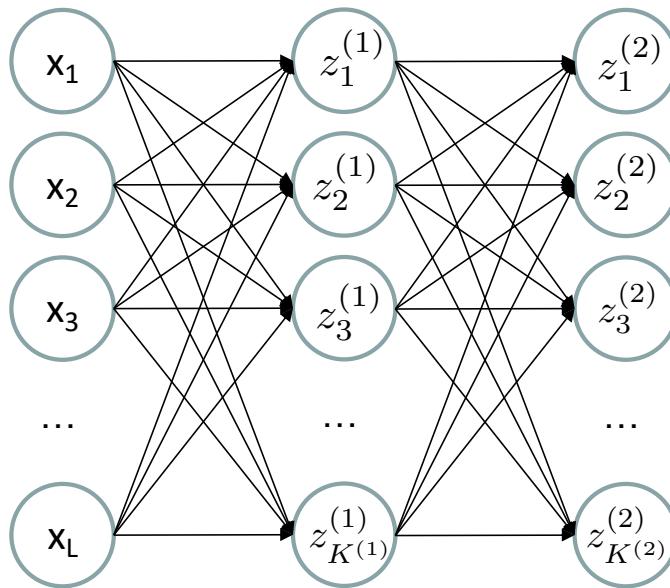
Multi-Layer Neural Network



Multi-Layer Neural Network

- Input to a layer: some $\dim(x)$ -dimensional input vector
- Output of a layer: some $\dim(y)$ -dimensional output vector
 - $\dim(y)$ is the number of neurons in the layer (1 output per neuron)
- Process of converting input to output:
 - Multiply the $(1, \dim(x))$ input vector with a $(\dim(x), \dim(y))$ weight vector.
The result has shape $(1, \dim(y))$.
 - Apply some non-linear function (e.g. sigmoid) to the result.
The result still has shape $(1, \dim(y))$.
- Big idea: Chain layers together
 - The input could come from a previous layer's output
 - The output could be used as the input to the next layer

Deep Neural Network

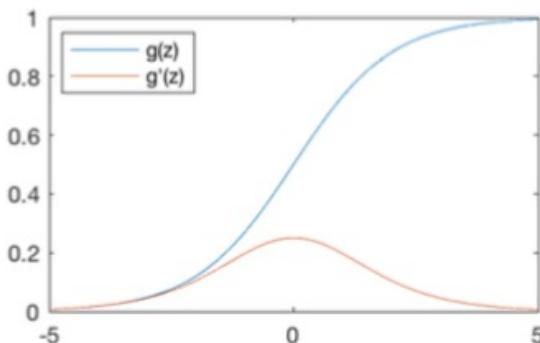


$$z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)$$

g = nonlinear activation function

Common Activation Functions

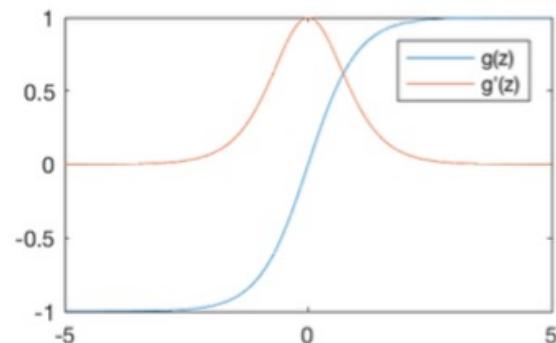
Sigmoid Function



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

$$g'(z) = g(z)(1 - g(z))$$

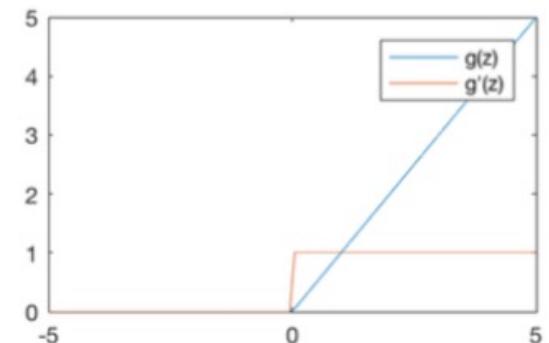
Hyperbolic Tangent



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

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



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

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

Same basic idea

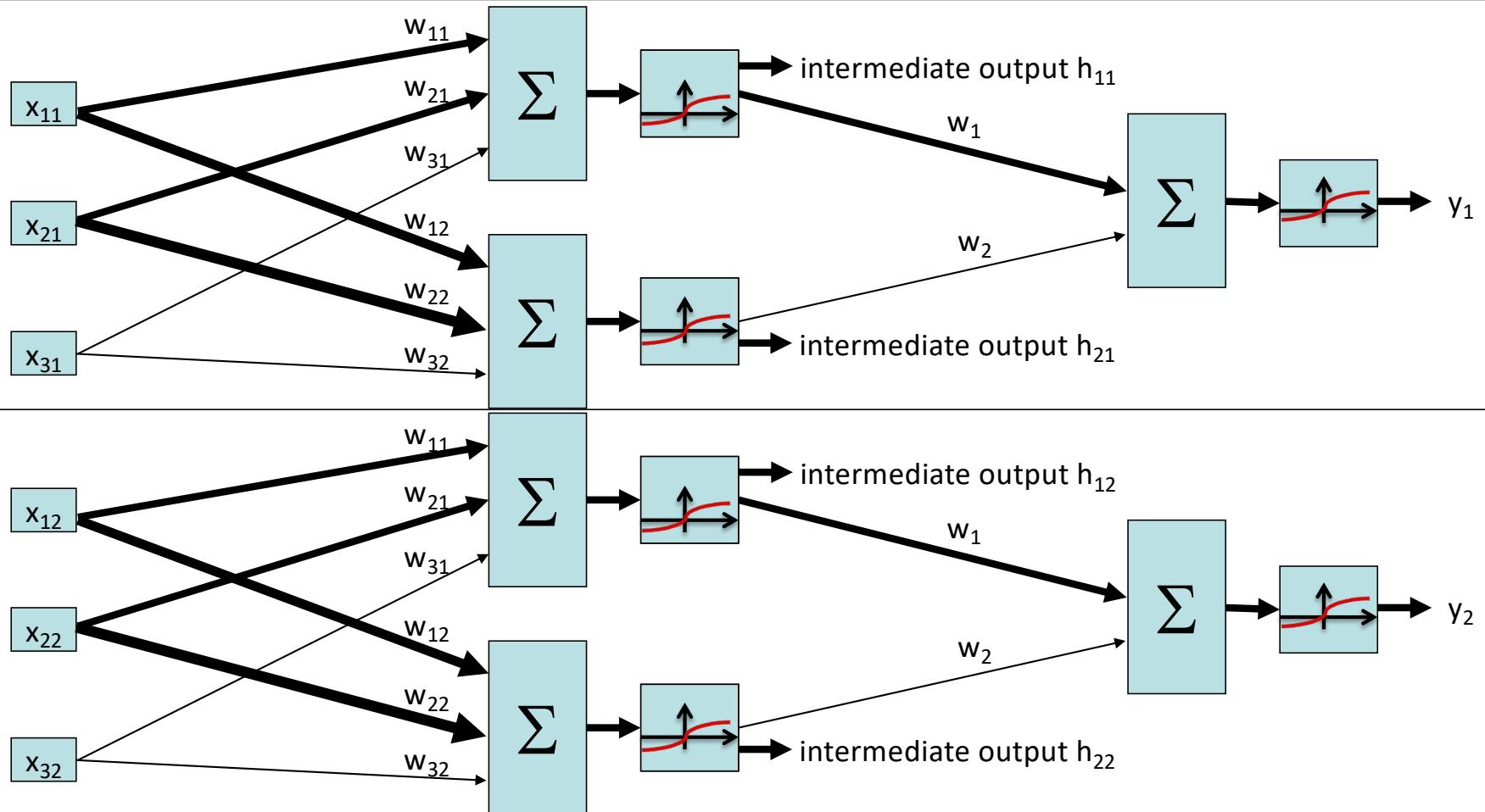
$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Dependance of P on w is much more elaborate

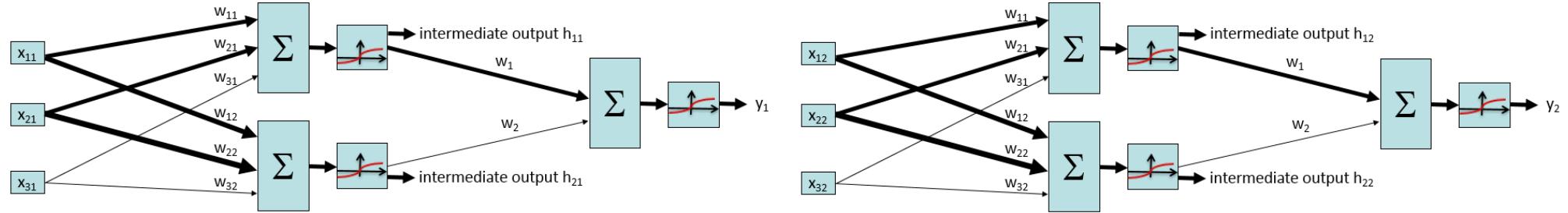
(set of all parameters often written as θ not w)

many arrays of parameters, not just a vector

“Batched” Computation



“Batched” Computation



$$\begin{aligned}y_1 &= \phi(w_1 h_{11} + w_2 h_{12}) \\&= \phi(w_1 \phi(w_{11} x_{11} + w_{21} x_{12} + w_{31} x_{13}) + w_2 \phi(w_{12} x_{11} + w_{22} x_{12} + w_{32} x_{13})) \\y_2 &= \phi(w_1 h_{21} + w_2 h_{22}) \\&= \phi(w_1 \phi(w_{11} x_{21} + w_{21} x_{22} + w_{31} x_{23}) + w_2 \phi(w_{12} x_{21} + w_{22} x_{22} + w_{32} x_{23}))\end{aligned}$$

We're not changing the architecture; we're just running the 2-neuron, 2-layer network twice to classify 2 inputs.

“Batched” Computation

$$\begin{aligned}y_1 &= \phi(w_1 h_{11} + w_2 h_{12}) \\&= \phi(w_1 \phi(w_{11}x_{11} + w_{21}x_{12} + w_{31}x_{13}) + w_2 \phi(w_{12}x_{11} + w_{22}x_{12} + w_{32}x_{13})) \\y_2 &= \phi(w_1 h_{21} + w_2 h_{22}) \\&= \phi(w_1 \phi(w_{11}x_{21} + w_{21}x_{22} + w_{31}x_{23}) + w_2 \phi(w_{12}x_{21} + w_{22}x_{22} + w_{32}x_{23}))\end{aligned}$$

Rewriting in matrix form:

$$\begin{aligned}&\phi \left(\begin{bmatrix} x_{11} & x_{21} & x_{31} \\ x_{12} & x_{22} & x_{32} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \right) \\&= \phi \left(\begin{bmatrix} w_{11}x_{11} + w_{21}x_{21} + w_{31}x_{31} & w_{12}x_{11} + w_{22}x_{21} + w_{32}x_{31} \\ w_{11}x_{12} + w_{21}x_{22} + w_{31}x_{32} & w_{12}x_{12} + w_{22}x_{22} + w_{32}x_{32} \end{bmatrix} \right) \\&= \begin{bmatrix} h_{11} & h_{21} \\ h_{12} & h_{22} \end{bmatrix} \\&\phi \left(\begin{bmatrix} h_{11} & h_{21} \\ h_{12} & h_{22} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \right) = \phi \left(\begin{bmatrix} w_1 h_{11} + w_2 h_{21} \\ w_1 h_{12} + w_2 h_{22} \end{bmatrix} \right) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}\end{aligned}$$

“Batched” Computation

$$\phi(x \times W_{\text{layer 1}}) = h$$

Shape (*batch*, $\text{dim}(x)$).
Input feature vector.

Shape ($\text{dim}(x)$, n).
Weights to be learned

Shape (*batch*, n).
Outputs of layer 1,
inputs to layer 2.

$$\phi(h \times W_{\text{layer 2}}) = y$$

Shape (*batch*, n).
Outputs of layer 1,
inputs to layer 2.

Shape (n , $\text{dim}(y)$).
Weights to be learned.

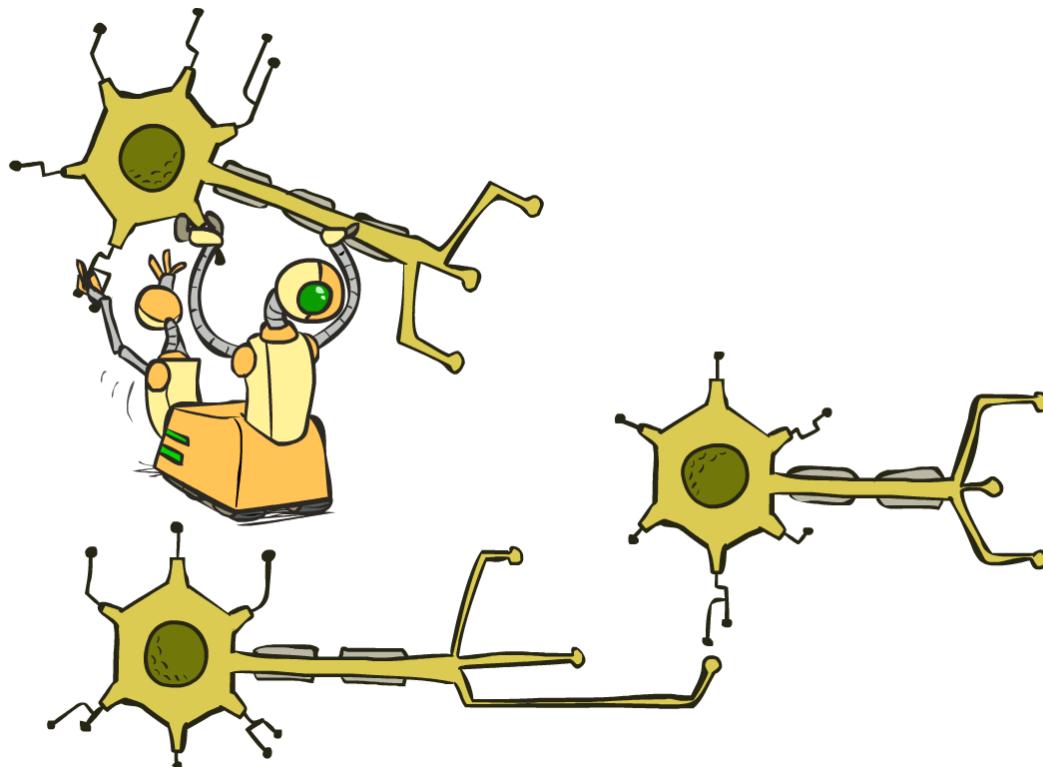
Shape (*batch*, $\text{dim}(y)$).
Output of network.

Big idea: We can “stack” inputs together to classify multiple inputs at once. The result is multiple outputs “stacked” together.

Multi-Layer Network, with Batches

- Input to a layer: *batch* different $\dim(x)$ -dimensional input vectors
- Output of a layer: *batch* different $\dim(y)$ -dimensional output vectors
 - $\dim(y)$ is the number of neurons in the layer (1 output per neuron)
- Process of converting input to output:
 - Multiply the (*batch*, $\dim(x)$) input matrix with a ($\dim(x)$, $\dim(y)$) weight vector.
The result has shape (*batch*, $\dim(y)$).
 - Apply some non-linear function (e.g. sigmoid) to the result.
The result still has shape (*batch*, $\dim(y)$).
- Big idea: Stack inputs/outputs to batch them
 - The multiplication by weights and non-linear function will be applied to each row (data point in the batch) separately.
 - *Efficiently parallelized on a GPU*

Training Neural Networks



Training Neural Networks

- Step 1: For each input in the training (sub)set x , predict a classification y using the current weights

$$\phi(x \times W_{\text{layer 1}}) = h \quad \phi(h \times W_{\text{layer 2}}) = y$$

- Step 2: Compare predictions with the true y values, using a **loss function**
 - Higher value of loss function = bad model
 - Lower value of loss function = good model
 - Example: **zero-one loss**: count the number of misclassified inputs
 - Example: **log loss** (derived from maximum likelihood; more on this soon)
 - Example: **sum of squared errors** (if you're solving a regression problem)
- Step 3: Use numerical method (e.g. gradient descent) to minimize loss
 - Loss is a function of the weights. Optimization goal: find weights that minimize loss

Log Loss Function

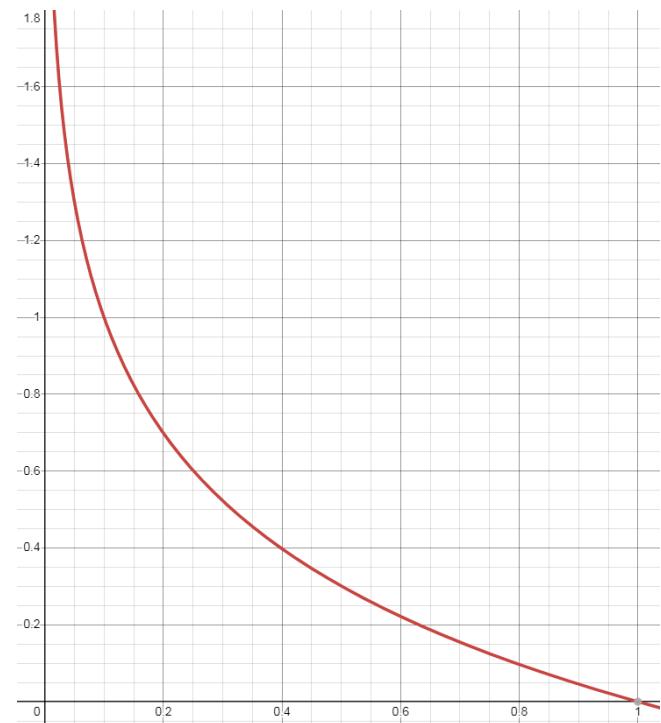
- Recall: loss function is a measure of how far off our model is
 - Higher value of loss function = bad model
 - Lower value of loss function = good model
- Log loss function for binary classification:

$$\text{Log Loss} = - \sum_i y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

- y_i = The true class of the i th data point in the training dataset (either 0 or 1)
- p_i = The probability of positive class, predicted by our classifier
- Each data point contributes some loss. The total loss is the sum over all data points.
- Note: Log loss is a function of the weights
 - Changing the weights changes the predictions p_i . The y_i do not change.

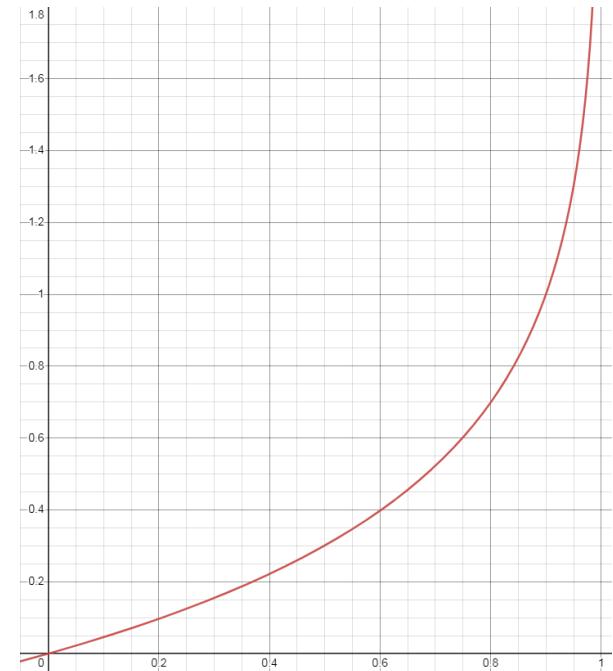
Log Loss Function

- Log loss function for a single data point: $-y \log(p) - (1 - y) \log(1 - p)$
- Case I: True class is $y = 1$
 - Log loss function becomes just $-\log(p)$
 - If classifier gives p near 0:
The model was confident in guessing $y=0$
Bad model \rightarrow high loss value
 - If classifier gives p near 1:
The model was confident in guessing $y=1$
Good model \rightarrow low loss value



Log Loss Function

- Log loss function for a single data point: $-y \log(p) - (1 - y) \log(1 - p)$
- Case II: True class is $y = 0$
 - Log loss function becomes just $-\log(1-p)$
 - If classifier gives p near 0:
The model was confident in guessing $y=0$
Good model → low loss value
 - If classifier gives p near 1:
The model was confident in guessing $y=1$
Bad model → high loss value
 - Note: This equation uses $y=0$, not $y=-1$,
for the negative class



Log Loss Example

$$\text{Log Loss} = - \sum_i y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

- Data point 1: True class $y=1$. We predicted $p=0.1$ probability it's the positive class.
 - Intuitively: Our guess is bad. We should have guessed closer to $p=1$.
 - This contributes $-\log(0.1) = 1$ to the loss function.
- Data point 2: True class $y=0$. We predicted $p=0$ probability it's the positive class.
 - Intuitively: Our guess was perfect. We were certain it was $y=0$.
 - This contributes $-\log(1.0) = 0$ to our loss function.
- Data point 3: True class $y=1$. We predicted 0.8 probability it's the positive class.
 - Intuitively: Our guess was pretty good, but not perfect.
 - This contributes $-\log(0.8) = 0.1$ to the loss function.
- Total loss: $1 + 0 + 0.1 = 1.1$
 - Goal: Find the weights that lead to the probabilities that minimize this loss function

Optimization Procedure: Gradient Descent

- init w
- for iter = 1, 2, ...
 $w \leftarrow w - \alpha \nabla \text{Log Loss}(w)$
where
$$\text{Log Loss} = - \sum_i y_i \log(p_i) + (1 - y_i) \log(1 - p_i)$$

and p_i is computed by running the network on input x_i and weights w
- α : learning rate --- tweaking parameter that needs to be chosen carefully

Computing Gradients

- How do we compute gradients of these loss functions?
 - Repeated application of the chain rule:

If $f(x) = g(h(x))$

Then $f'(x) = g'(h(x))h'(x)$

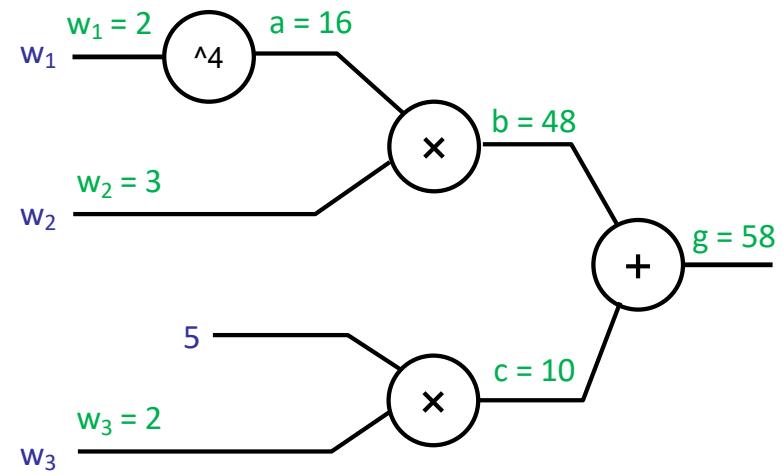
→ Derivatives can be computed by following well-defined procedures

Automatic Differentiation

- Automatic differentiation software
 - e.g. Theano, TensorFlow, PyTorch, Chainer
 - Only need to program the function $g(x,y,w)$
 - Can automatically compute all derivatives w.r.t. all entries in w
 - This is typically done by caching info during forward computation pass of f , and then doing a backward pass = “backpropagation”
 - Autodiff / Backpropagation can often be done at computational cost comparable to the forward pass
- Need to know this exists
- How this is done? -- outside of scope of CS188

Backpropagation*

- Gradient of $g(w_1, w_2, w_3) = w_1^4 w_2 + 5w_3$ at $w_1 = 2, w_2 = 3, w_3 = 2$
- Think of g as a composition of many functions
 - Then, we can use the chain rule to compute the gradient
- $g = b + c$
- $\frac{\partial g}{\partial b} = 1, \frac{\partial g}{\partial c} = 1$
- $b = a \times w_2$
- $\frac{\partial g}{\partial a} = \frac{\partial g}{\partial b} \frac{\partial b}{\partial a} = 1 \cdot w_2 = 3 \quad \frac{\partial g}{\partial w_2} = \frac{\partial g}{\partial b} \frac{\partial b}{\partial w_2} = 1 \cdot a = 16$
- $a = w_1^4$
- $\frac{\partial g}{\partial w_1} = \frac{\partial g}{\partial a} \frac{\partial a}{\partial w_1} = 3 \cdot 4w_1^3 = 96$
- $c = 5w_3$
- $\frac{\partial g}{\partial w_3} = \frac{\partial g}{\partial c} \frac{\partial c}{\partial w_3} = 1 \cdot 5 = 5$



PyTorch is Amazing

```
✓  [1]  1 import torch  
0s  
✓  [2]  1 # Create tensors  
      2 x = torch.tensor(2.0, requires_grad=True)  
      3 y = torch.tensor(3.0, requires_grad=True)  
      4  
      5 # Perform operations  
      6 z = x * y + x**2  
      7  
      8 # Compute gradients  
      9 z.backward()  
     10  
     11 # Print gradients  
     12 print("Gradient of z with respect to x:", x.grad)  
     13 print("Gradient of z with respect to y:", y.grad)  
  
Gradient of z with respect to x: tensor(7.)  
Gradient of z with respect to y: tensor(2.)
```

PyTorch is Amazing

✓
0s

```
1 x = torch.tensor([1., 2., 3., 4.], requires_grad=True)
2 y = torch.tensor([4., 3., 2., 1.], requires_grad=True)
3 M = torch.tensor([[2., 0., 0., 1.],
4                 [1., 2., 0., 0.],
5                 [0., 0., 2., 0.],
6                 [1., 0., 0., 2.]], requires_grad=True)
7 M_inv = torch.linalg.inv(M)
8 z = y @ (M_inv @ x) # ``@`` means matrix multiplication
9
10 z.backward()
11 print("Gradient of z with respect to x:", x.grad)
12 print("Gradient of z with respect to y:", y.grad)
13 print("Gradient of z with respect to M:", M.grad)
```

→ Gradient of z with respect to x: tensor([1.3333, 1.5000, 1.0000, -0.1667])
Gradient of z with respect to y: tensor([-0.6667, 1.3333, 1.5000, 2.3333])
Gradient of z with respect to M: tensor([[0.8889, -1.7778, -2.0000, -3.1111],
[1.0000, -2.0000, -2.2500, -3.5000],
[0.6667, -1.3333, -1.5000, -2.3333],
[-0.1111, 0.2222, 0.2500, 0.3889]])

Gradient Ascent on the Log Likelihood Objective

$$\max_w \text{ ll}(w) = \max_w \underbrace{\sum_i \log P(y^{(i)}|x^{(i)}; w)}_{g(w)}$$

- `init w`
- `for iter = 1, 2, ...`

$$w \leftarrow w + \alpha * \nabla \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

Gradient Ascent on the Log Likelihood Objective

$$\max_w \text{ll}(w) = \max_w \underbrace{\sum_i \log P(y^{(i)}|x^{(i)}; w)}_{g(w)}$$

$$f(x) = g(x) + h(x)$$

$$df/dx = d(g + h)/dx = dg/dx + dh/dx$$

- `init w`
- `for iter = 1, 2, ...`

$$w \leftarrow w + \alpha * \sum_i \nabla \log P(y^{(i)}|x^{(i)}; w)$$

Stochastic Gradient Ascent on the Log Likelihood Objective

$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Observation: once gradient on one training example has been computed, might as well incorporate before computing next one

- `init w`
- `for iter = 1, 2, ...`
 - `pick random j`

$$w \leftarrow w + \alpha * \nabla \log P(y^{(j)} | x^{(j)}; w)$$

Mini-Batch Gradient Ascent on the Log Likelihood Objective

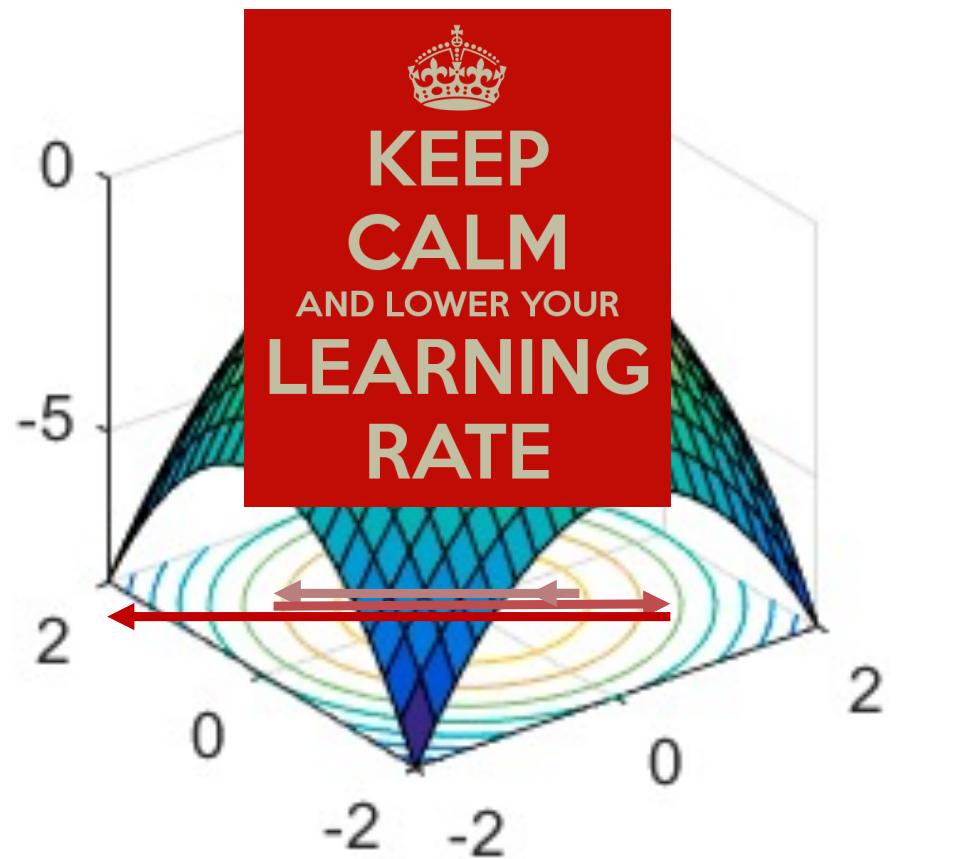
$$\max_w \text{ll}(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)$$

Observation: gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

- `init w`
- `for iter = 1, 2, ...`
 - pick random subset of training examples J

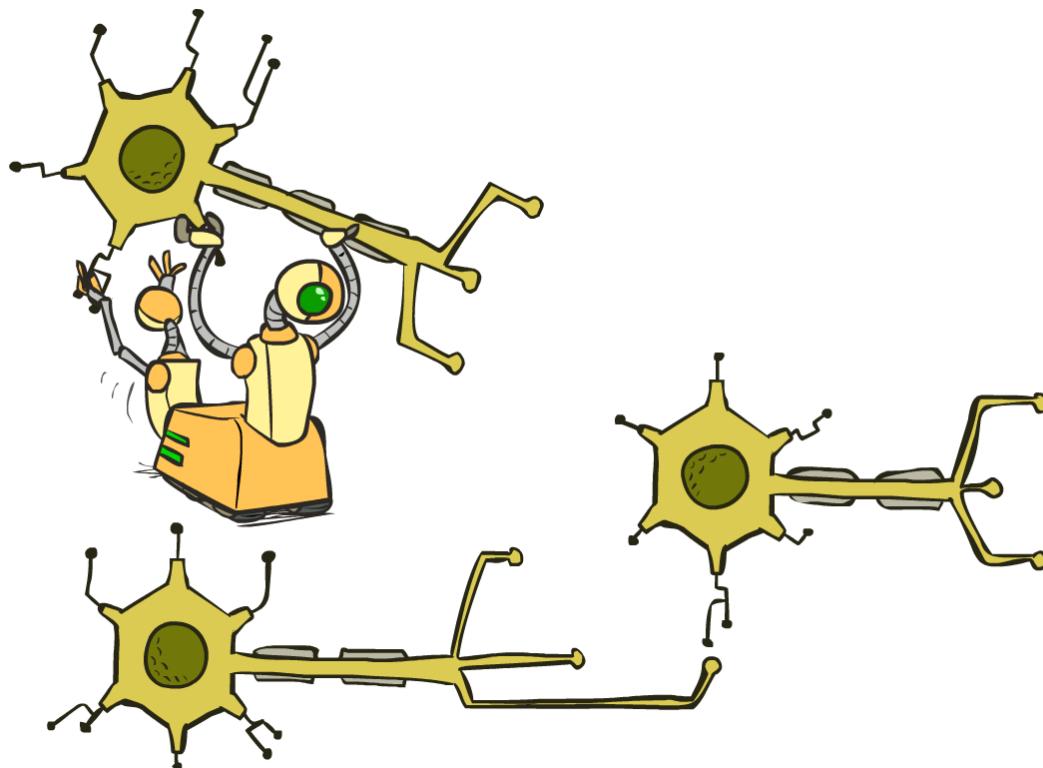
$$w \leftarrow w + \alpha * \sum_{j \in J} \nabla \log P(y^{(j)} | x^{(j)}; w)$$

Problem with High Learning Rate



Source: offconvex.org

Properties of Neural Networks



Neural Networks Properties

- Theorem (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.
- Practical considerations
 - Can be seen as learning the features
 - Large number of neurons
 - Danger for overfitting
 - (hence early stopping!)

Universal Function Approximation Theorem*

Hornik theorem 1: Whenever the activation function is *bounded and nonconstant*, then, for any finite measure μ , standard multilayer feedforward networks can approximate any function in $L^p(\mu)$ (the space of all functions on R^k such that $\int_{R^k} |f(x)|^p d\mu(x) < \infty$) arbitrarily well, provided that sufficiently many hidden units are available.

Hornik theorem 2: Whenever the activation function is *continuous, bounded and non-constant*, then, for arbitrary compact subsets $X \subseteq R^k$, standard multilayer feedforward networks can approximate any continuous function on X arbitrarily well with respect to uniform distance, provided that sufficiently many hidden units are available.

- In words: Given any continuous function $f(x)$, if a 2-layer neural network has enough hidden units, then there is a choice of weights that allow it to closely approximate $f(x)$.

Cybenko (1989) "Approximations by superpositions of sigmoidal functions"

Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks"

Leshno and Schocken (1991) "Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function"

Universal Function Approximation Theorem*

Math. Control Signals Systems (1989) 2: 303–314

Mathematics of Control,
Signals, and Systems
© 1989 Springer-Verlag New York Inc.

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results include an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Keywords. Neural networks, Approximation, Completeness.

1. Introduction

A number of diverse application areas are concerned with the representation of general functions of an n -dimensional real variable, $x \in \mathbb{R}^n$, by finite linear combinations of the form

$$\sum_{j=1}^k a_j \sigma(y_j^\top x + \theta_j), \quad (1)$$

where $y_j \in \mathbb{R}^n$ and $a_j, \theta_j \in \mathbb{R}$ are fixed, (y^\top is the transpose of y so that $y^\top x$ is the inner product of y and x .) Here the univariate function σ depends heavily on the context of the application. Our major concern is with so-called sigmoidal σ 's:

$$\sigma(t) \rightarrow \begin{cases} 1 & \text{as } t \rightarrow +\infty, \\ 0 & \text{as } t \rightarrow -\infty. \end{cases}$$

Such functions arise naturally in neural network theory as the activation function of a neural node (or *unit* as is becoming the preferred term [L1], [RH]). The main result of this paper is a demonstration of the fact that sums of the form (1) are dense in the space of continuous functions on the unit cube if σ is any continuous sigmoidal

* Date received: October 21, 1988. Date revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619103, ONR Contract N00036-G-0202 and DOE Grant DE-FG02-85ER25001.

† Center for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, U.S.A.

Neural Networks, Vol. 4, pp. 251–257, 1991
Printed in the USA. All rights reserved.
0893-6080/91 \$3.00 + .00
Copyright © 1991 Pergamon Press plc

ORIGINAL CONTRIBUTION

Approximation Capabilities of Multilayer Feedforward Networks

KURT HORNIK

Technische Universität Wien, Vienna, Austria

(Received 30 January 1990; revised and accepted 25 October 1990)

Abstract—We show that standard multilayer feedforward networks with as few as a single hidden layer and arbitrary bounded and nonconstant activation function are universal approximators with respect to $L(\mu)$ performance criteria, for arbitrary smooth input environment measures μ , provided only that sufficiently many hidden units are available. If the activation function is continuous, bounded and nonconstant, then continuous mappings can be learned uniformly over compact input sets. We also give very general conditions ensuring that networks with sufficiently smooth activation functions are capable of arbitrarily accurate approximation to a function and its derivatives.

Keywords—Multilayer feedforward networks, Activation function, Universal approximation capabilities, Input environment measure, $L(\mu)$ approximation, Uniform approximation, Sobolev spaces, Smooth approximation.

1. INTRODUCTION

The approximation capabilities of neural network architectures have recently been investigated by many authors, including Carroll and Dickinson (1989), Cybenko (1989), Funahashi (1989), Gallant and White (1988), Hecht-Nielsen (1989), Hornik, Stinchcombe, and White (1988, 1990), Ito and Miyake (1988), Lapedes and Farber (1988), Stinchcombe and White (1989, 1990). (This list is by no means complete.)

If we think of the network architecture as a rule for computing values of f output units given values at k input units, one implementation of a class of mappings from \mathbb{R}^k to \mathbb{R}^m we can ask how well arbitrary mappings from \mathbb{R}^k to \mathbb{R}^m can be approximated by the network, in particular, if as many hidden units as required for internal representation and computation may be employed.

How to measure the accuracy of approximation depends on how we measure closeness between functions, which in turn varies significantly with the specific problem to be dealt with. In many applications, it is necessary to have the network perform simultaneously well on all input units taken from some compact input set X in \mathbb{R}^k . In this case, closeness is

measured by the uniform distance between functions on X , that is,

$$\rho_{\text{unif}}(f, g) = \sup_{x \in X} |f(x) - g(x)|.$$

In other applications, we think of the inputs as random variables and are interested in the *average performance* where the average is taken with respect to the input environment measure μ , where $\mu(R^k) < \infty$. In this case, closeness is measured by the $L^p(\mu)$ distance

$$\rho_{L^p}(f, g) = \left[\int_{\mathbb{R}^k} |f(x) - g(x)|^p d\mu(x) \right]^{1/p}.$$

1 $\leq p < \infty$, the most popular choice being $p = 2$, corresponding to mean square error.

Of course, there are many other ways of measuring closeness of functions. In particular, in many applications, it is also necessary that the derivatives of the approximating function implemented by the network closely resemble those of the function to be approximated, up to some order. This issue was first taken up in Hornik et al. (1990), who discuss the sources of need of smooth functional approximation in more detail. Typical examples arise in robotics (kinematics and dynamics), control theory, signal processing (analysis of chaotic time series), for a recent application to problems of nonparametric inference in statistics and econometrics, see Gallant and White (1989).

All papers establishing certain approximation ca-

MULTILAYER FEEDFORWARD NETWORKS WITH NON-POLYNOMIAL ACTIVATION FUNCTIONS CAN APPROXIMATE ANY FUNCTION

by

Moshe Leshno
Faculty of Management
Tel Aviv University
Tel Aviv, Israel 69978

and

Shimon Schocken
Leonard N. Stern School of Business
New York University
New York, NY 10003

September 1991

Center for Research on Information Systems
Information Systems Department
Leonard N. Stern School of Business
New York University

Working Paper Series

STERN IS-91-26

Appeared previously as *Working Paper No. 21/91* at The Israel Institute Of Business Research

Cybenko (1989) "Approximations by superpositions of sigmoidal functions"

Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks"

Leshno and Schocken (1991) "Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function"

Summary of Key Ideas

- Optimize probability of label given input $\max_w ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)$
- Continuous optimization
 - Gradient ascent:
 - Compute steepest uphill direction = gradient (= just vector of partial derivatives)
 - Take step in the gradient direction
 - Repeat (until held-out data accuracy starts to drop = “early stopping”)
- Deep neural nets
 - Last layer = still logistic regression
 - Now also many more layers before this last layer
 - = computing the features
 - → the features are learned rather than hand-designed
 - Universal function approximation theorem
 - If neural net is large enough
 - Then neural net can represent any continuous mapping from input to output with arbitrary accuracy
 - But remember: need to avoid overfitting / memorizing the training data → early stopping!
 - Automatic differentiation gives the derivatives efficiently (how? = outside of scope of 188)

Application: Large Language Models

A

Can you please tell my Intro to AI class how a single attention head works, and how that building block allows you to be a proficient English speaker?

Hello class, I'm happy to explain how a single attention head works in AI systems like myself. An attention head is a key building block in the neural network architectures, like the transformers I'm built on, that allow AI to process and understand language.

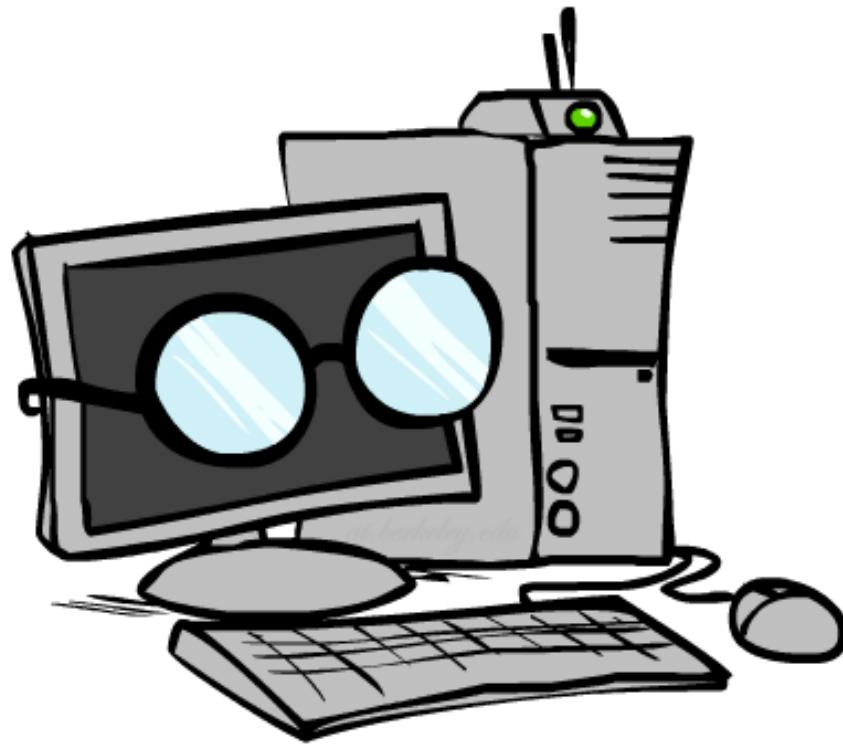
At its core, an attention head lets the AI system focus on and weight the importance of different parts of the input as it builds up an understanding. When I process a sentence, the attention head looks at each word and assigns a weight to how relevant that word is to understanding other words in the sentence.

For example, take the sentence "The quick brown fox jumps over the lazy dog." The attention head would highly weight the connections between "fox" and "jumps" since those are closely related. But it would assign a lower weight between "fox" and "lazy" since those words are less relevant to each other.

By building up these weighted connections between the different elements of the input, the attention head creates a rich representation that captures the meaning and context. Importantly, this happens bidirectionally - each word attends to all the other words, rather than just the previous words like in older language models.

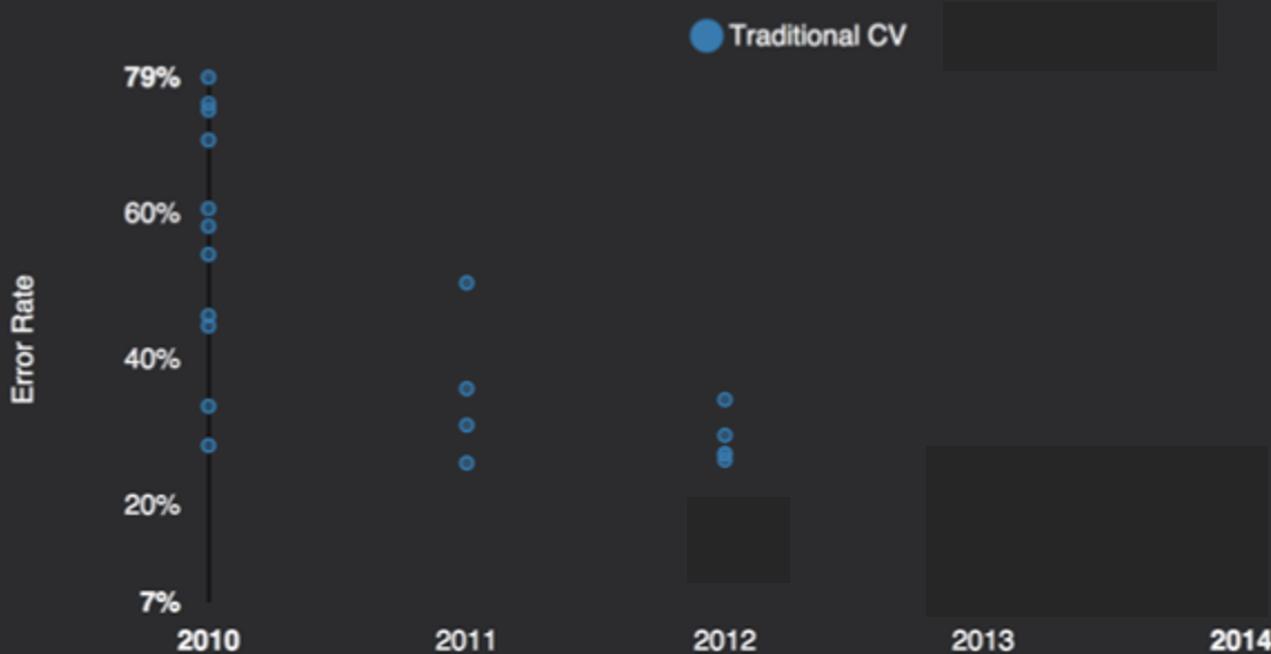
Multiple attention heads are then stacked together, allowing the model to build up increasingly sophisticated representations of the input text. Different heads can focus on

Application: Computer Vision



Performance

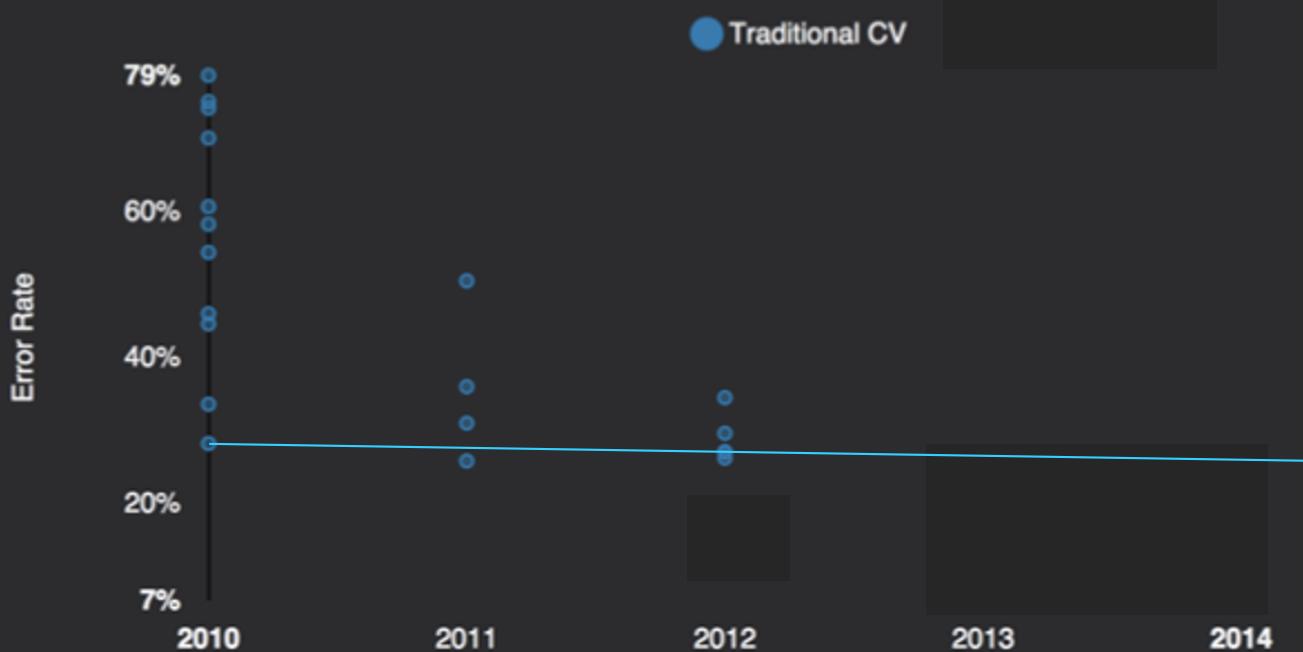
ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Performance

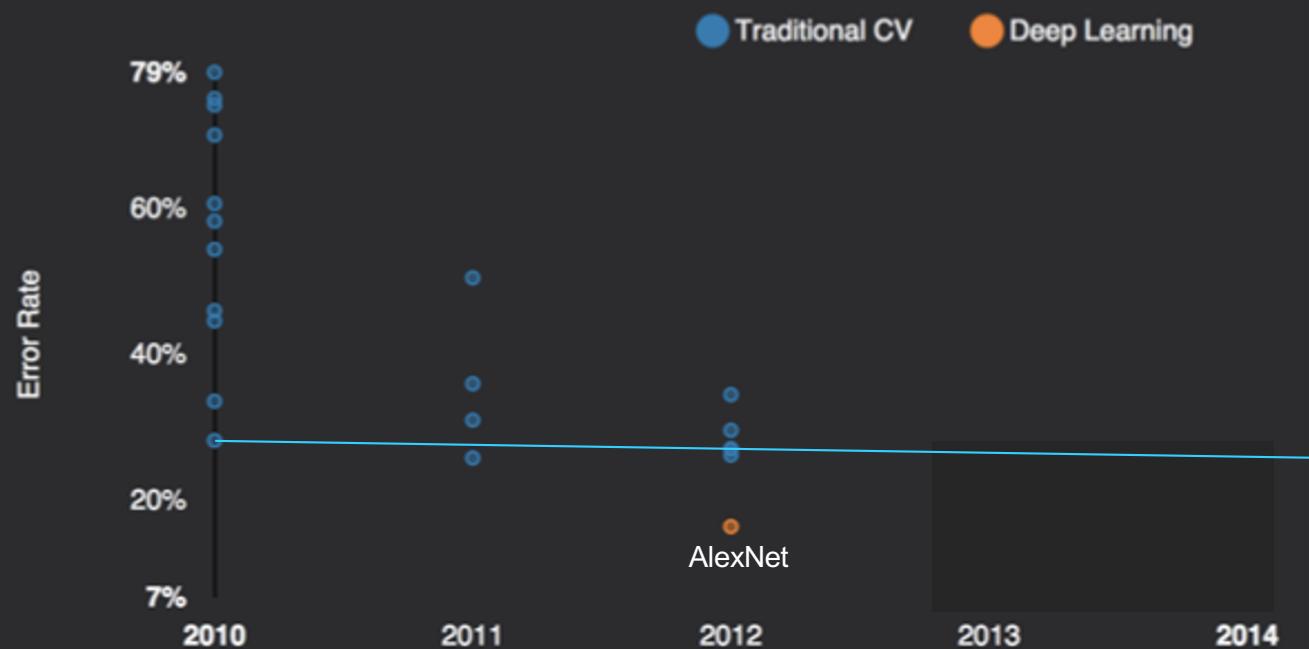
ImageNet Error Rate 2010-2014



*graph credit Matt
Zeiler, Clarifai*

Performance

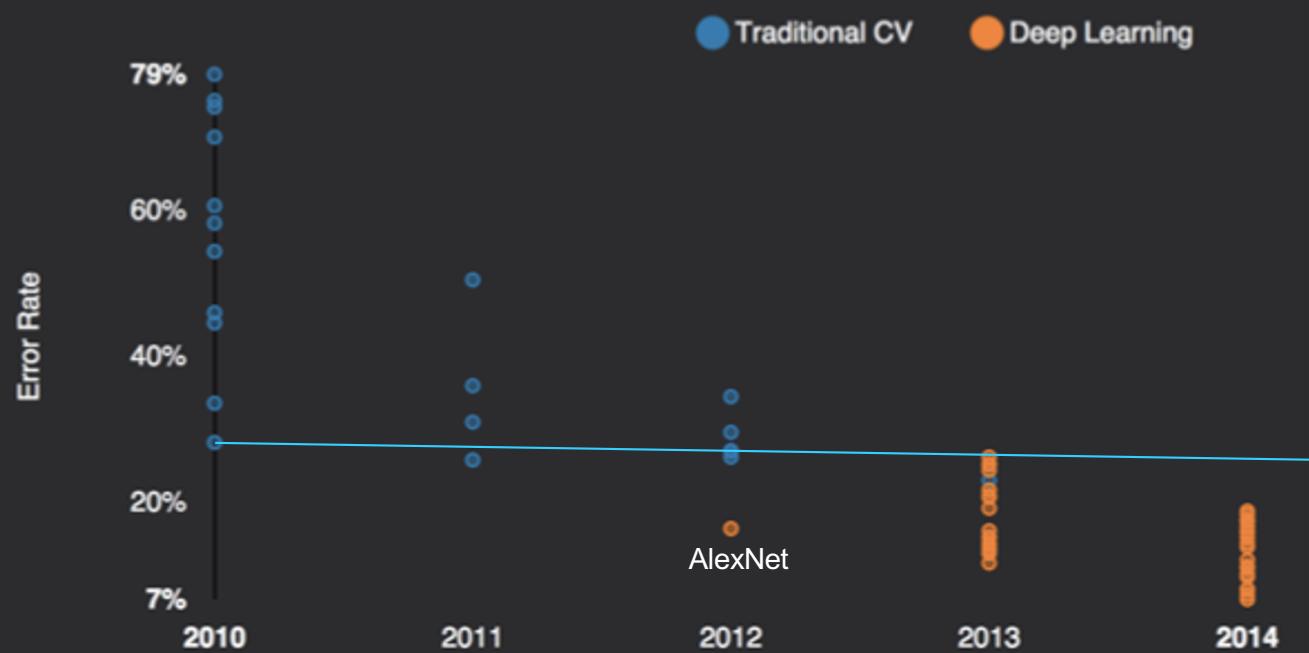
ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Performance

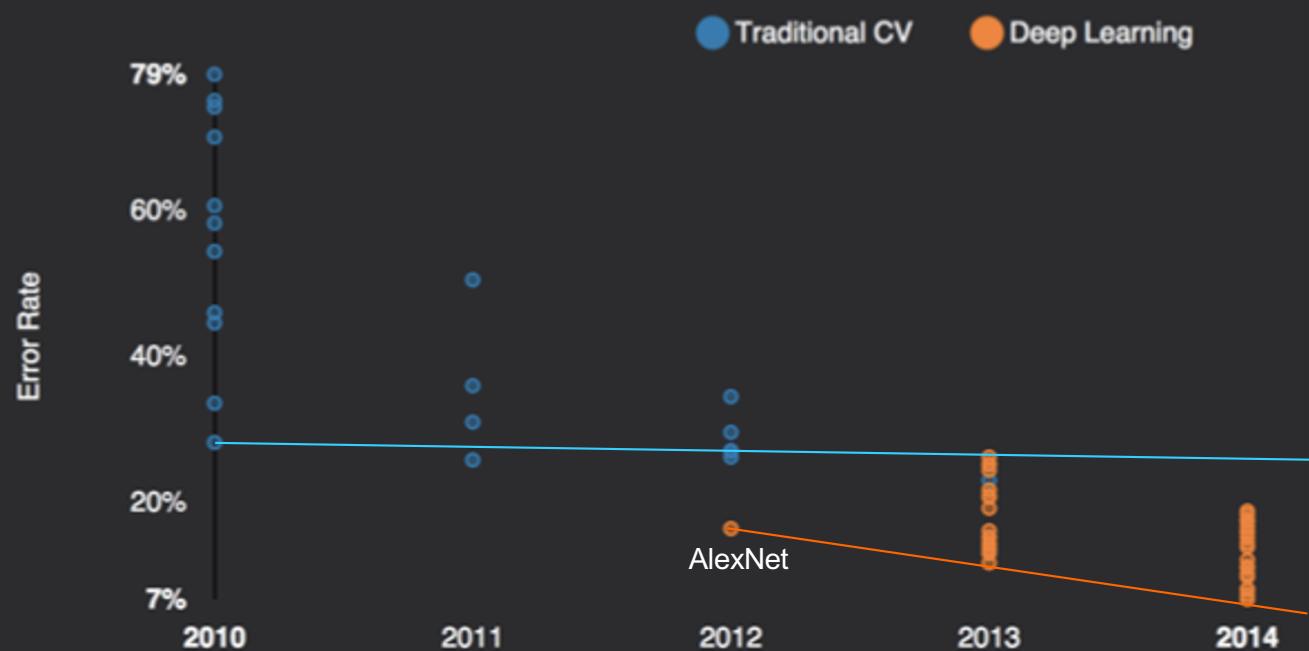
ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Performance

ImageNet Error Rate 2010-2014



graph credit Matt
Zeiler, Clarifai

Fun Neural Net Demo Site

- Demo-site:
 - <http://playground.tensorflow.org/>