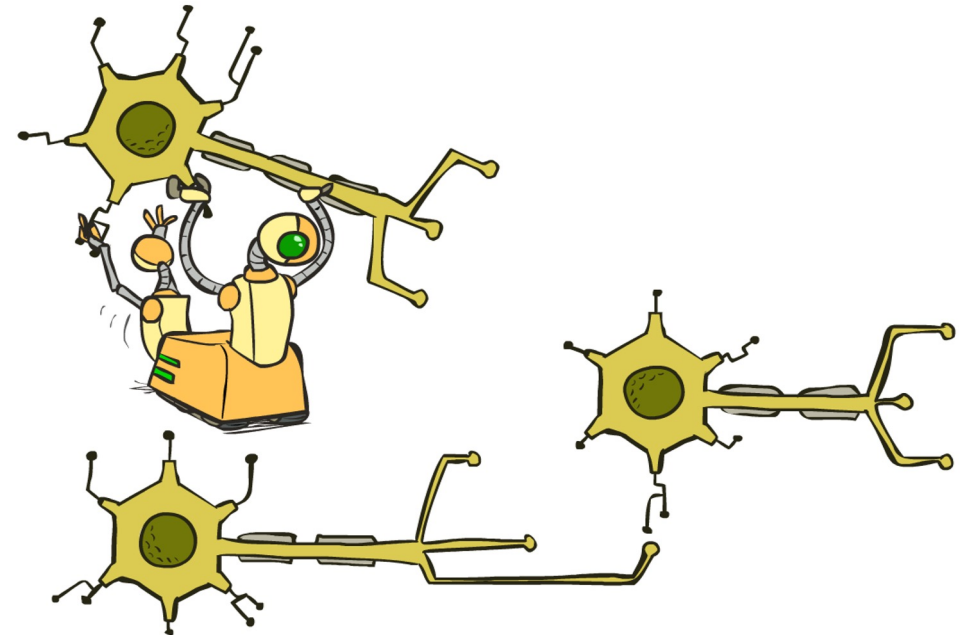# Artificial Intelligence - INFOF311

Neural networks and conclusion

Instructor : Tom Lenaerts
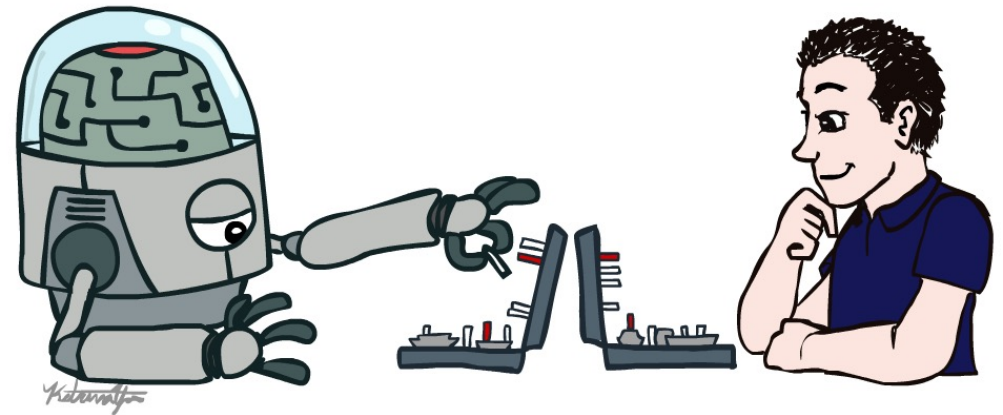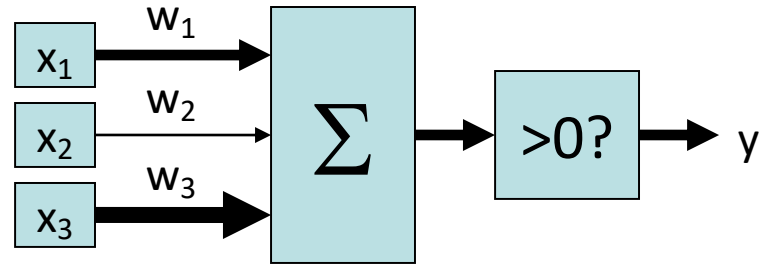
Acknowledgement

Center for
Human-Compatible
Artificial
Intelligence

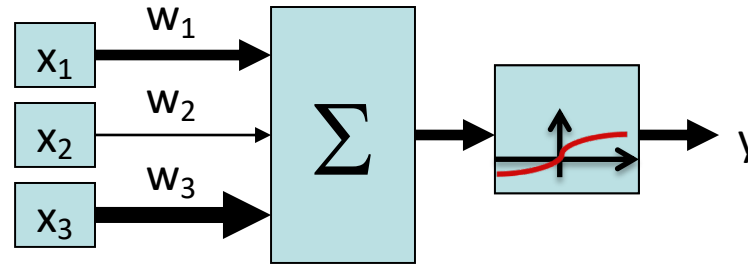The slides for INFOF311 are slightly modified versions of the slides of the spring and summer CS188 sessions in 2021 and 2022

Review: Perceptron

$$y = \begin{cases} 1 & w_1 x_1 + w_2 x_2 + w_3 x_3 > 0 \\ 0 & \text{otherwise} \end{cases}$$

Review: Perceptron with Sigmoid Activation

$$y = \phi(w_1 x_1 + w_2 x_2 + w_3 x_3)$$

$$= \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + w_3 x_3)}}$$

2-Layer, 2-Neuron Neural Network
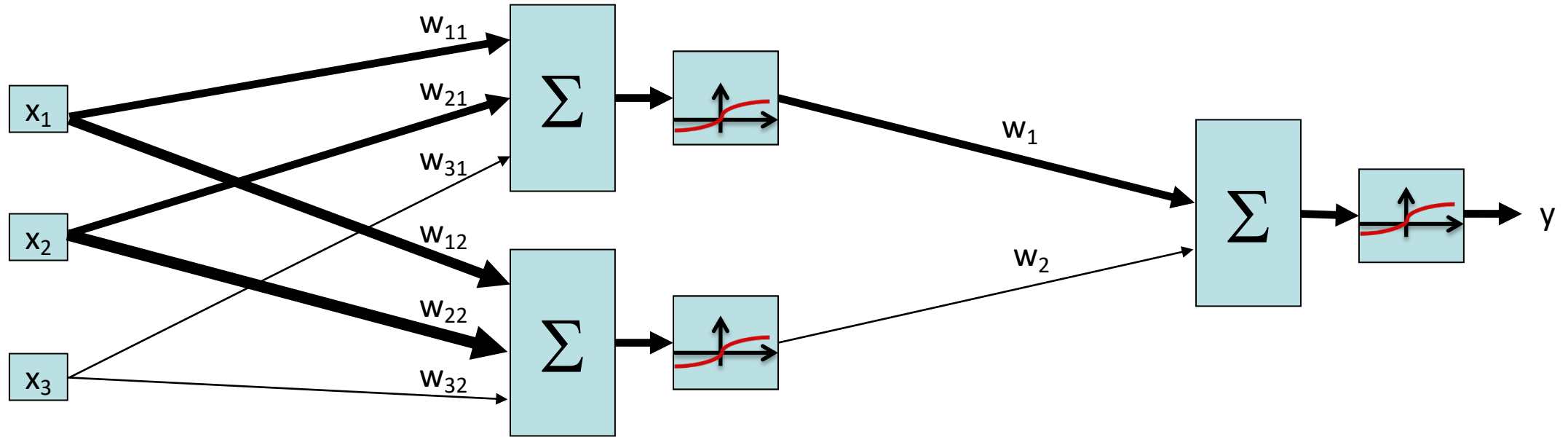
2-Layer, 2-Neuron Neural Network
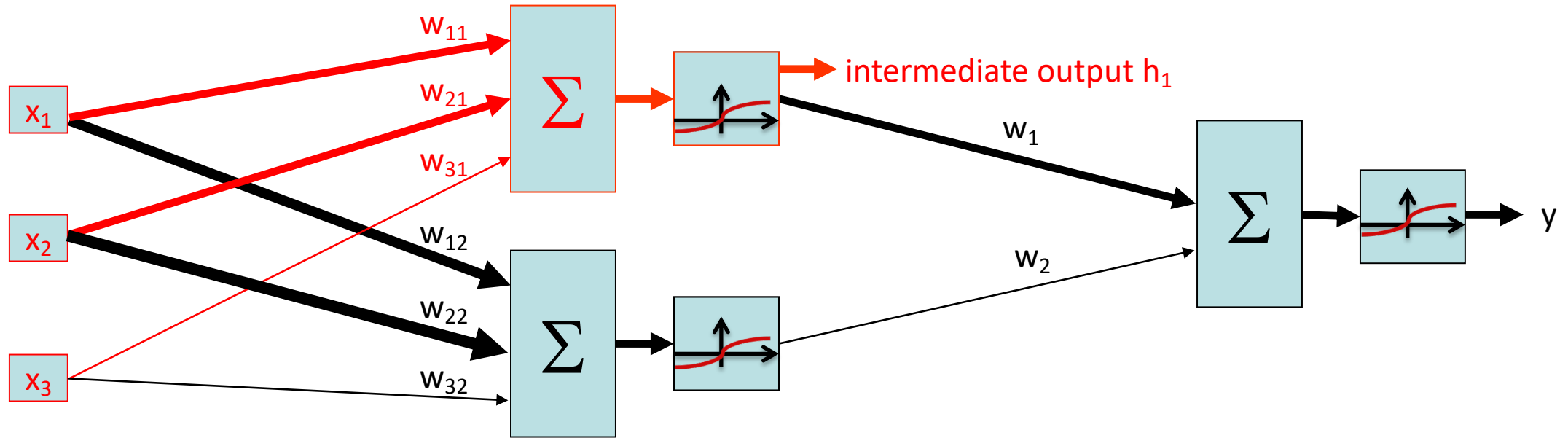
$$\text{intermediate output } h_1 = \phi(w_{11}x_1 + w_{21}x_2 + w_{31}x_3)$$

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

2-Layer, 2-Neuron Neural Network

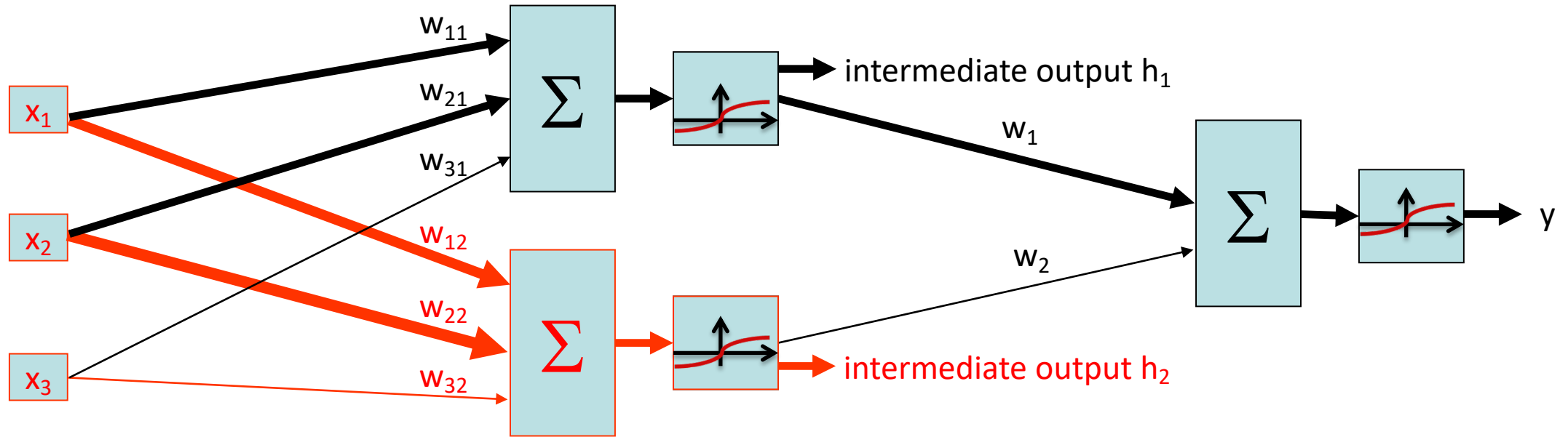$$\text{intermediate output } h_2 = \phi(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)$$

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

2-Layer, 2-Neuron Neural Network
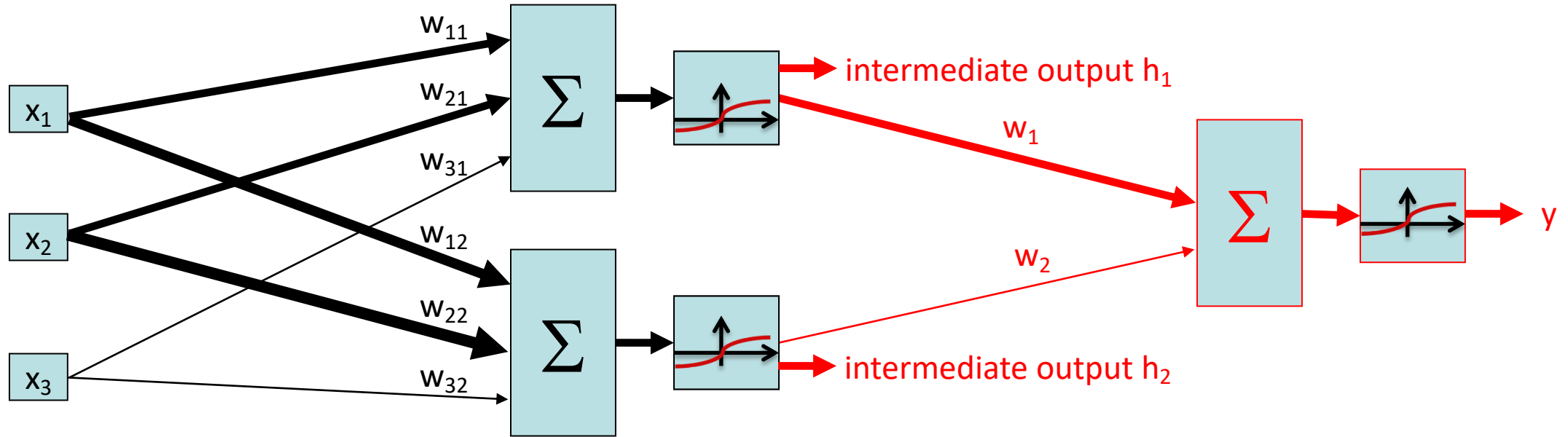
$$y = \phi(w_1 h_1 + w_2 h_2)$$
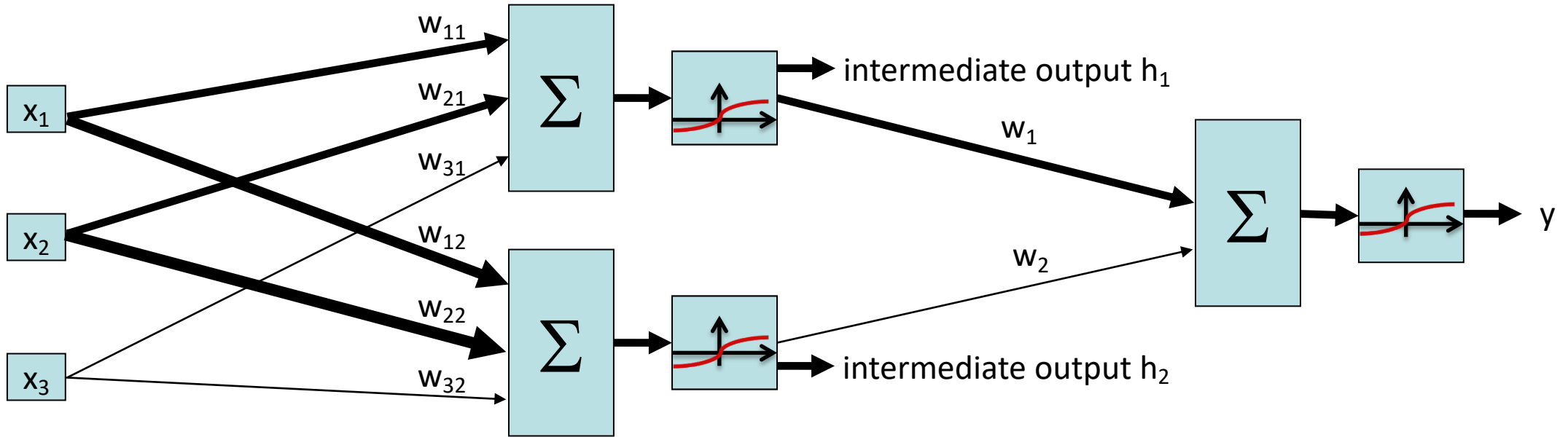
$$= \frac{1}{1 + e^{-(w_1 h_1 + w_2 h_2)}}$$

2-Layer, 2-Neuron Neural Network

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

2-Layer, 2-Neuron Neural Network

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

The same equation, formatted with matrices:

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

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

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

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

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.

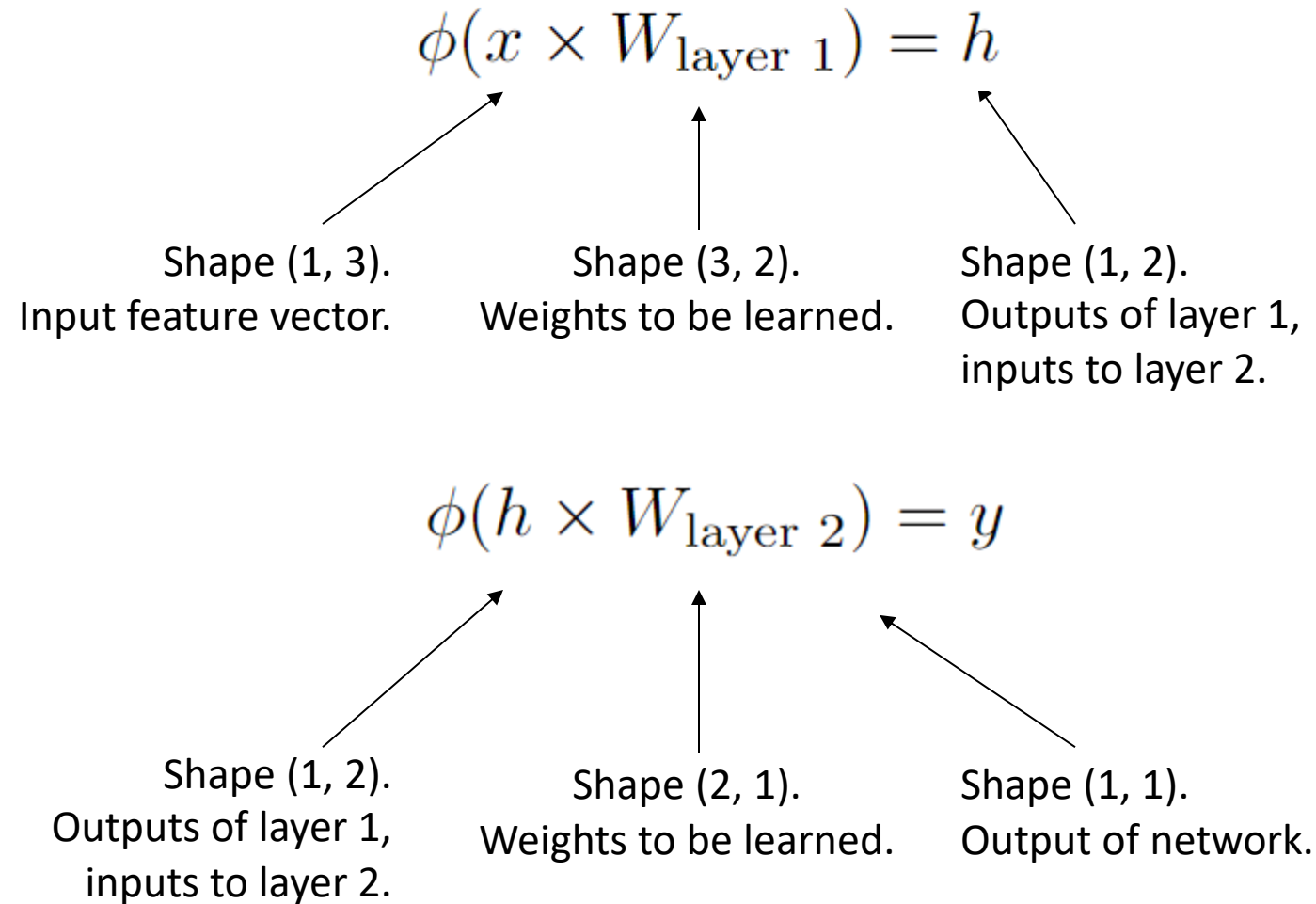# 2-Layer, 3-Neuron Neural Network

2-Layer, 3-Neuron Neural Network

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

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

$$= \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix}$$

$$\phi \left(\begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \right) = \phi \left(w_1 h_1 + w_2 h_2 + w_3 h_3 \right) = y$$
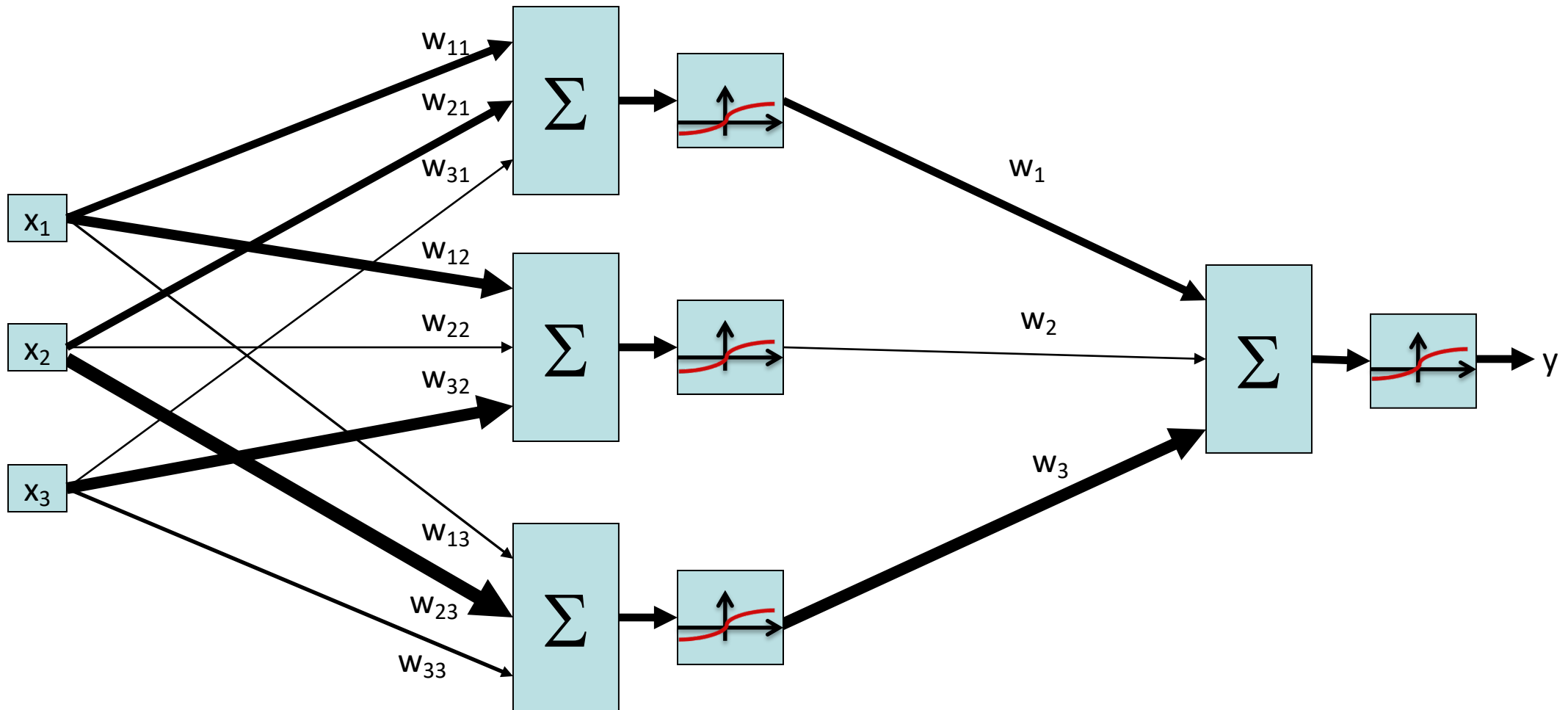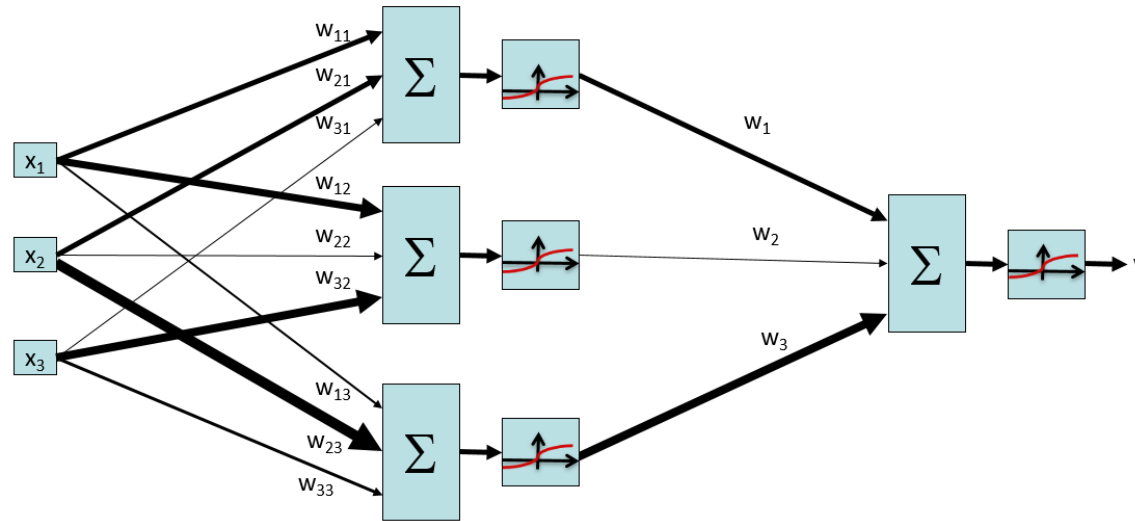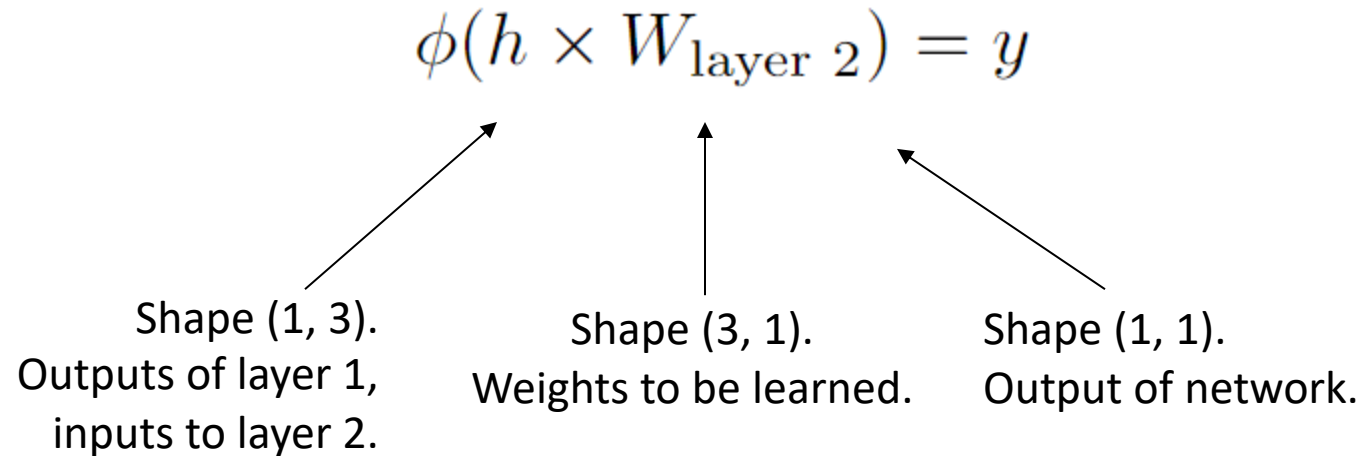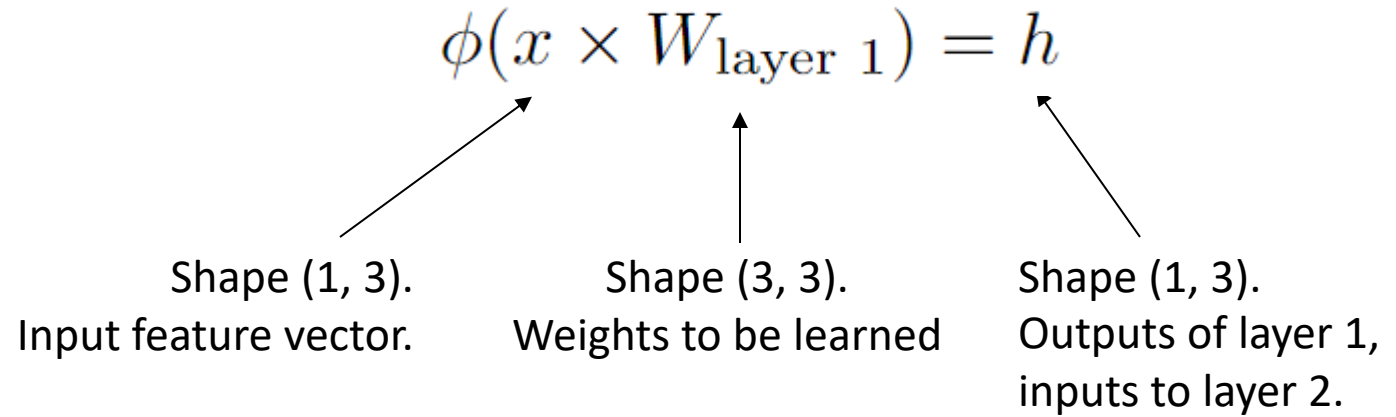
2-Layer, 3-Neuron Neural Network

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

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.
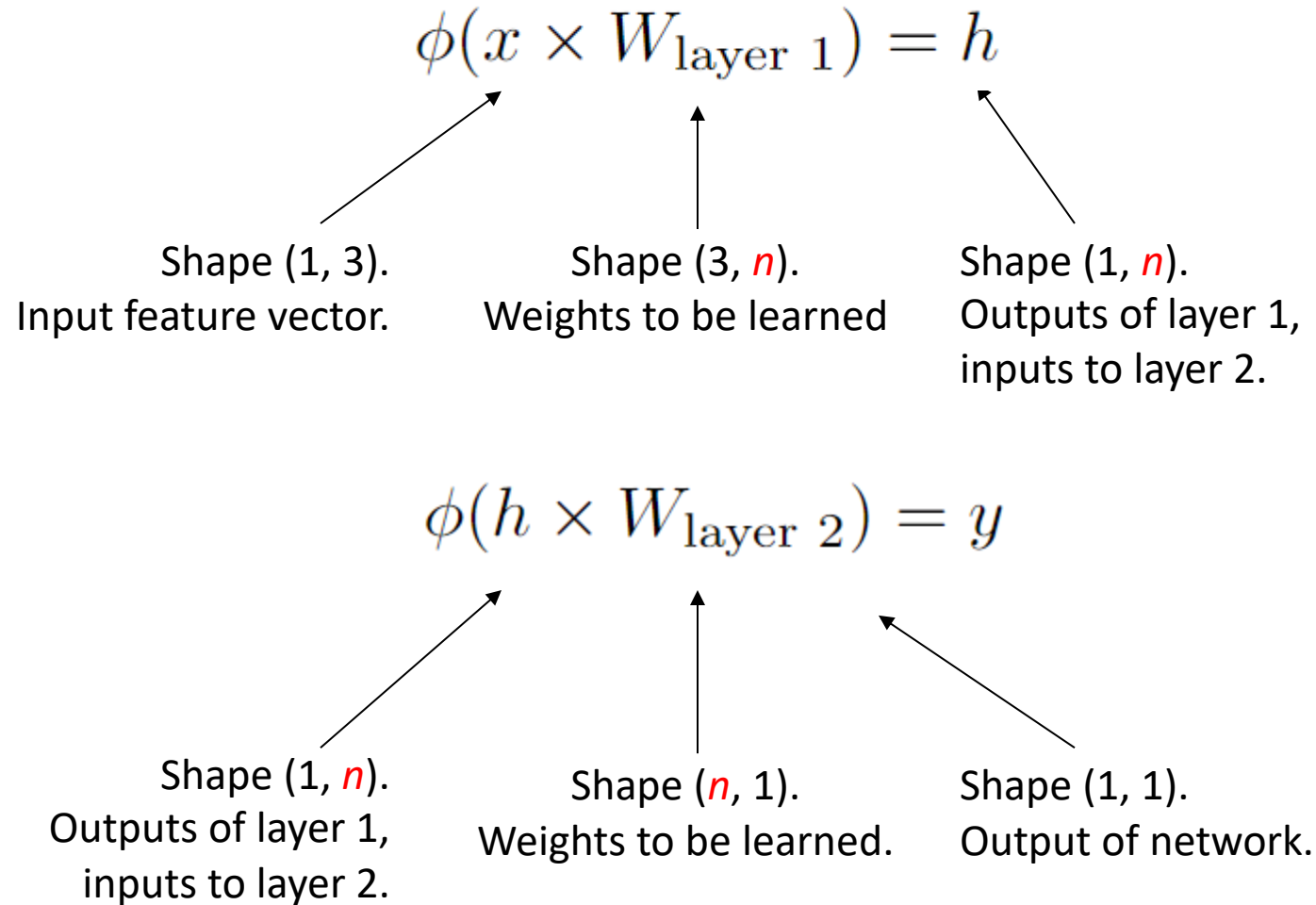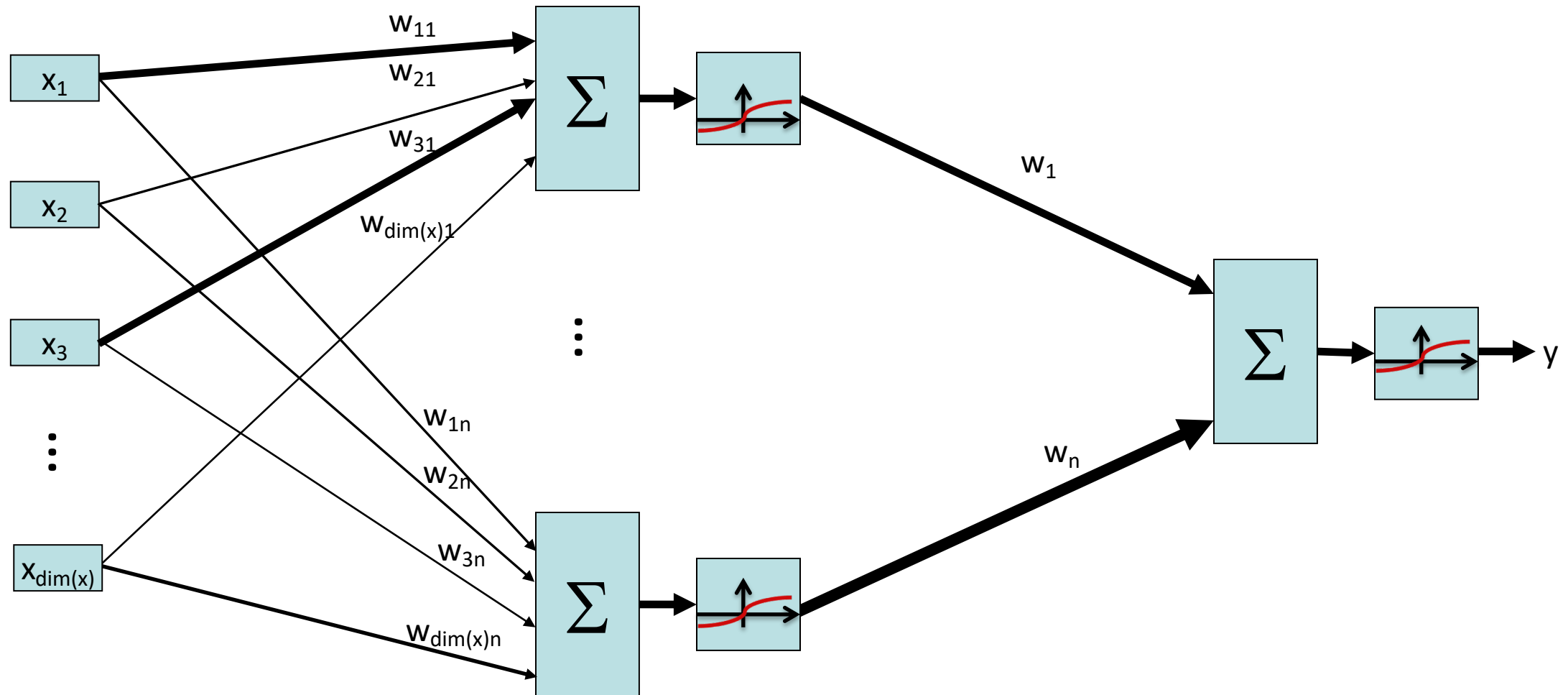
Shape (1, 1).
Output of network.
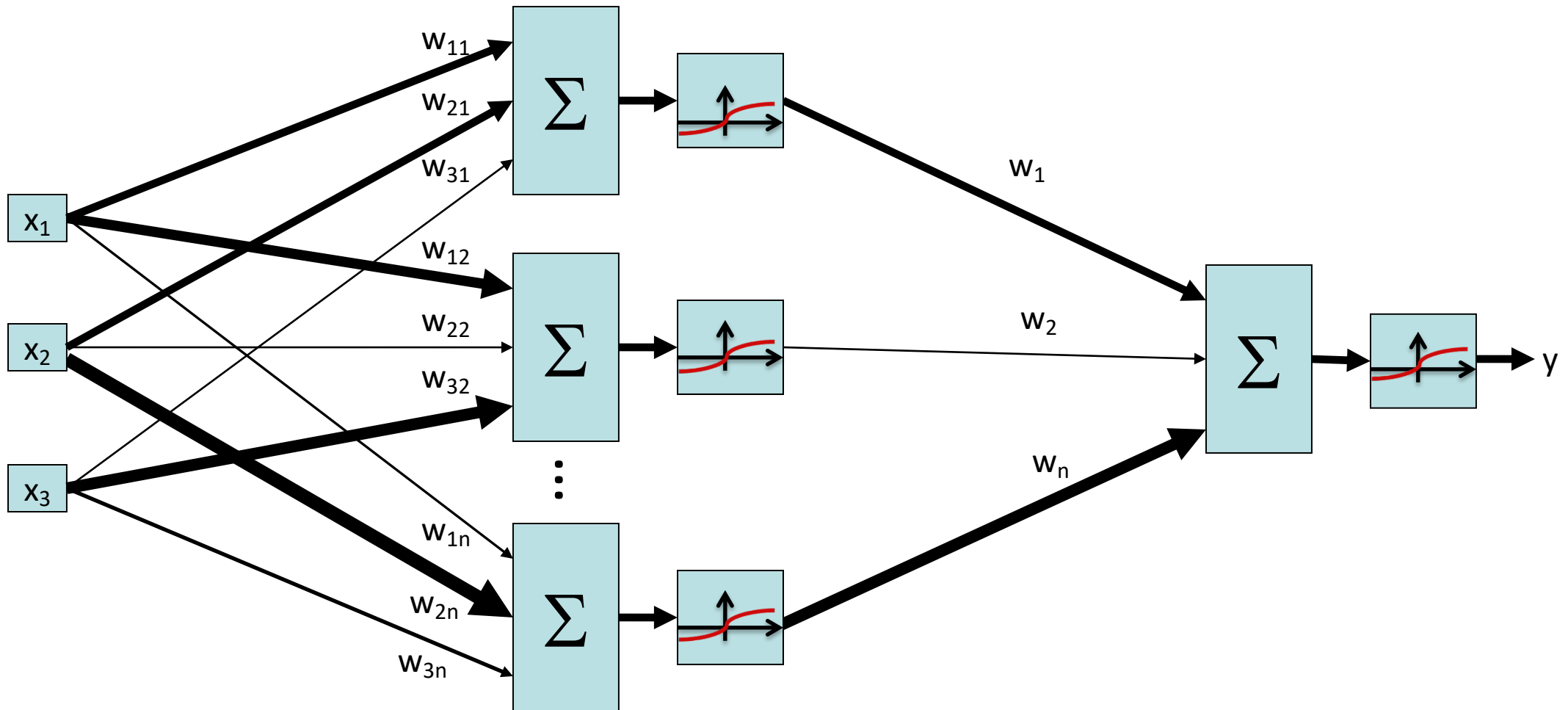
Generalize: Number of hidden neurons

The hidden layer doesn't necessarily need to have 3 neurons; it 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.

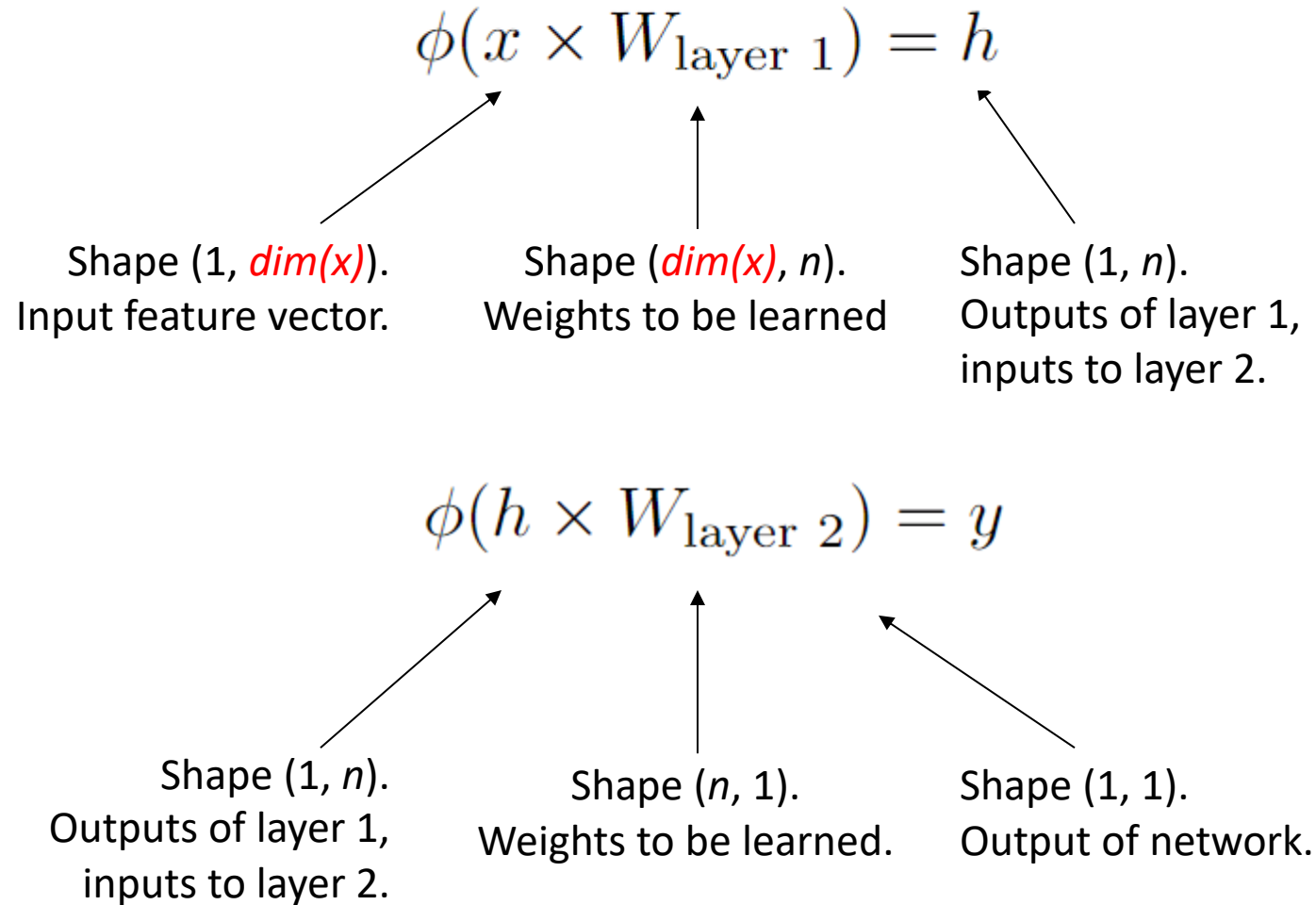The hidden layer doesn't necessarily need to have 3 neurons; it could have any arbitrary number n neurons.
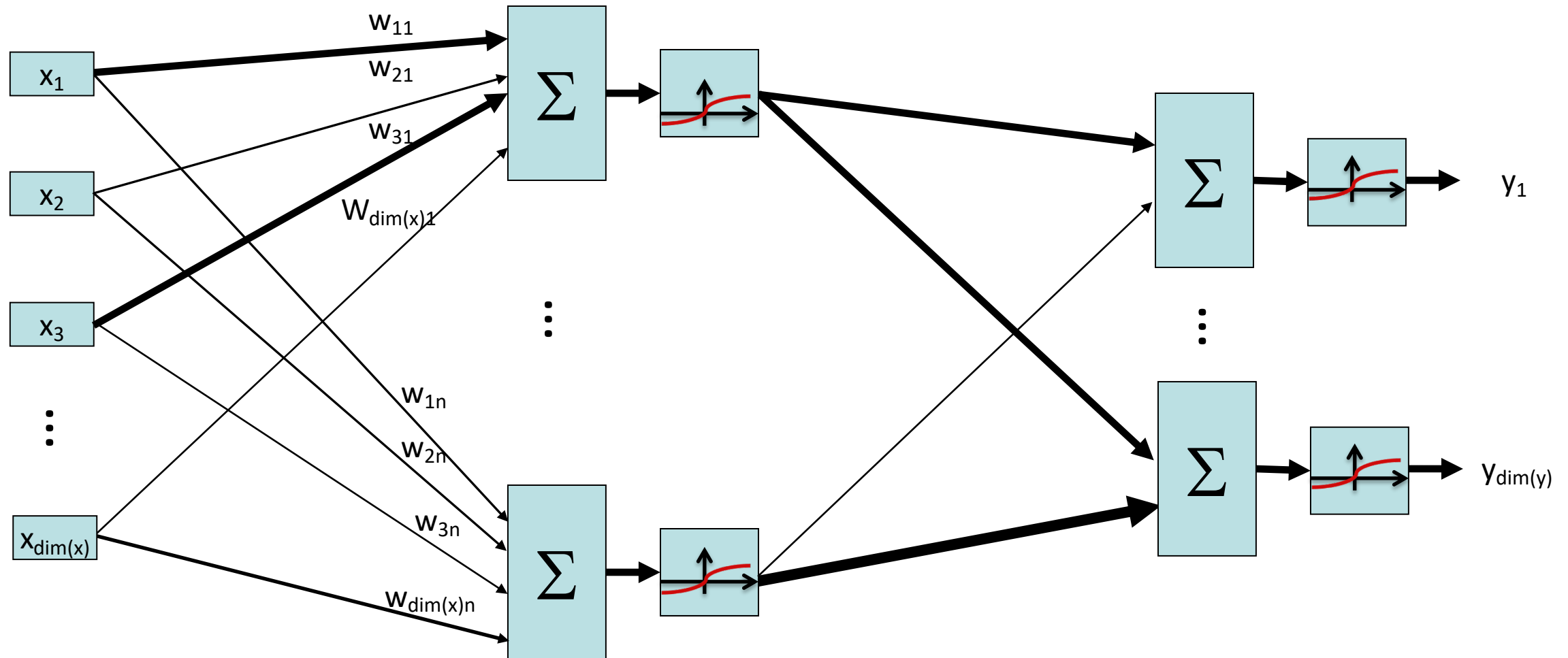
Generalize: Number of input features

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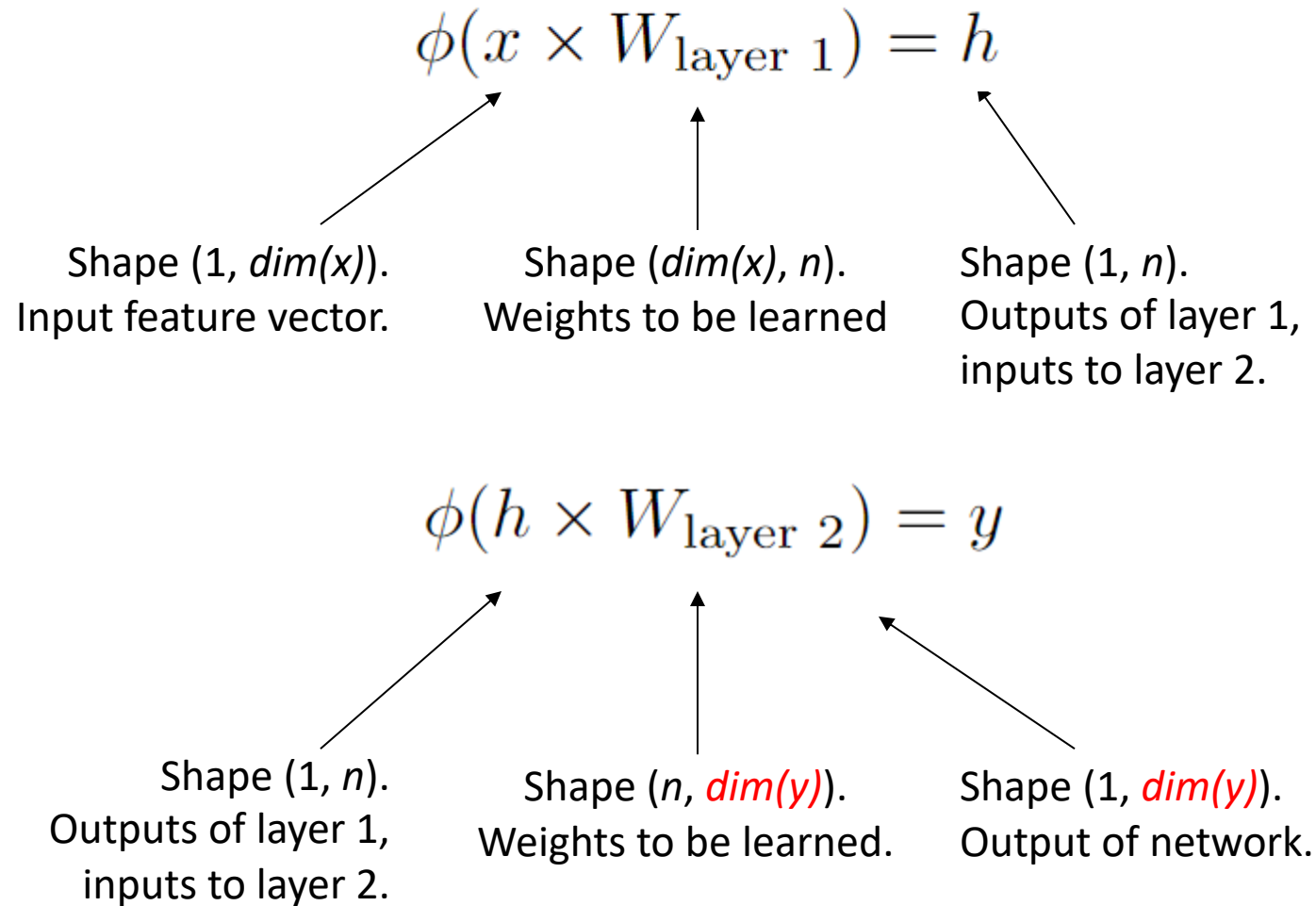
Generalize: Number of input features

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

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*, 1).
Weights to be learned.

Shape (1, 1).
Output of network.

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 outputs

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

Generalize: Number of input features

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

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.

Big idea: 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
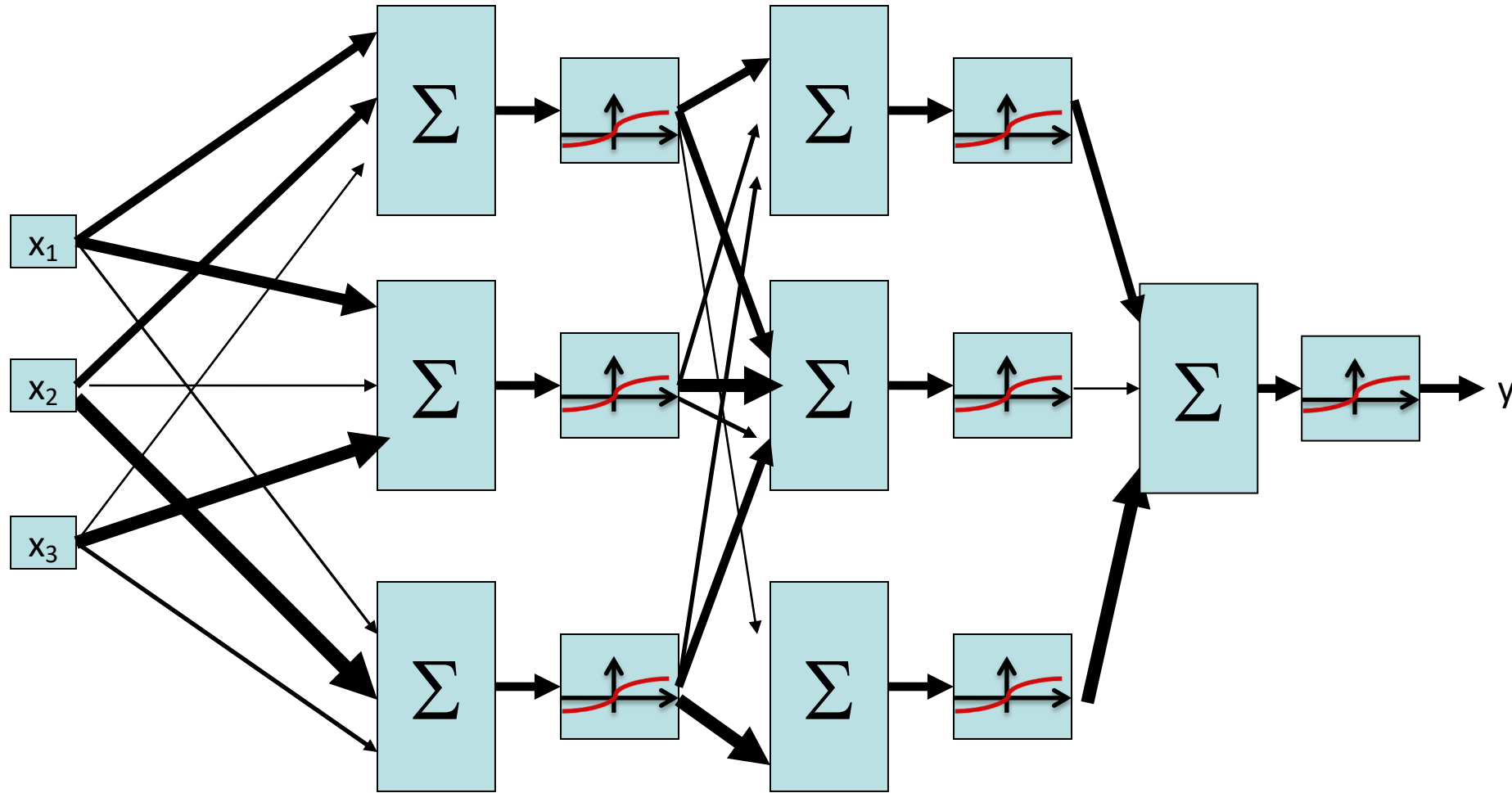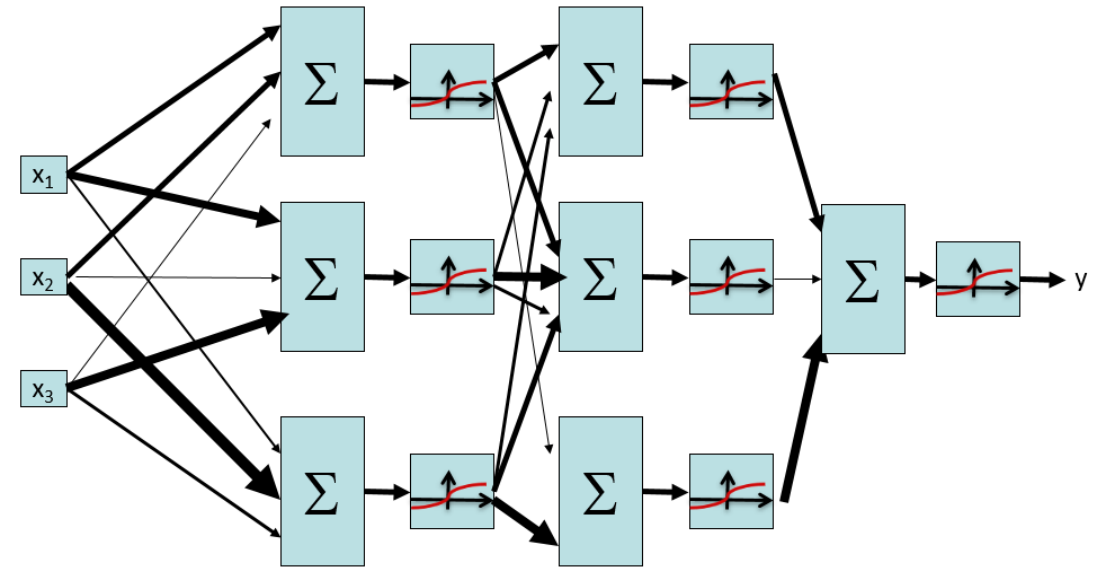
3-Layer, 3-Neuron Neural Network

- 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.
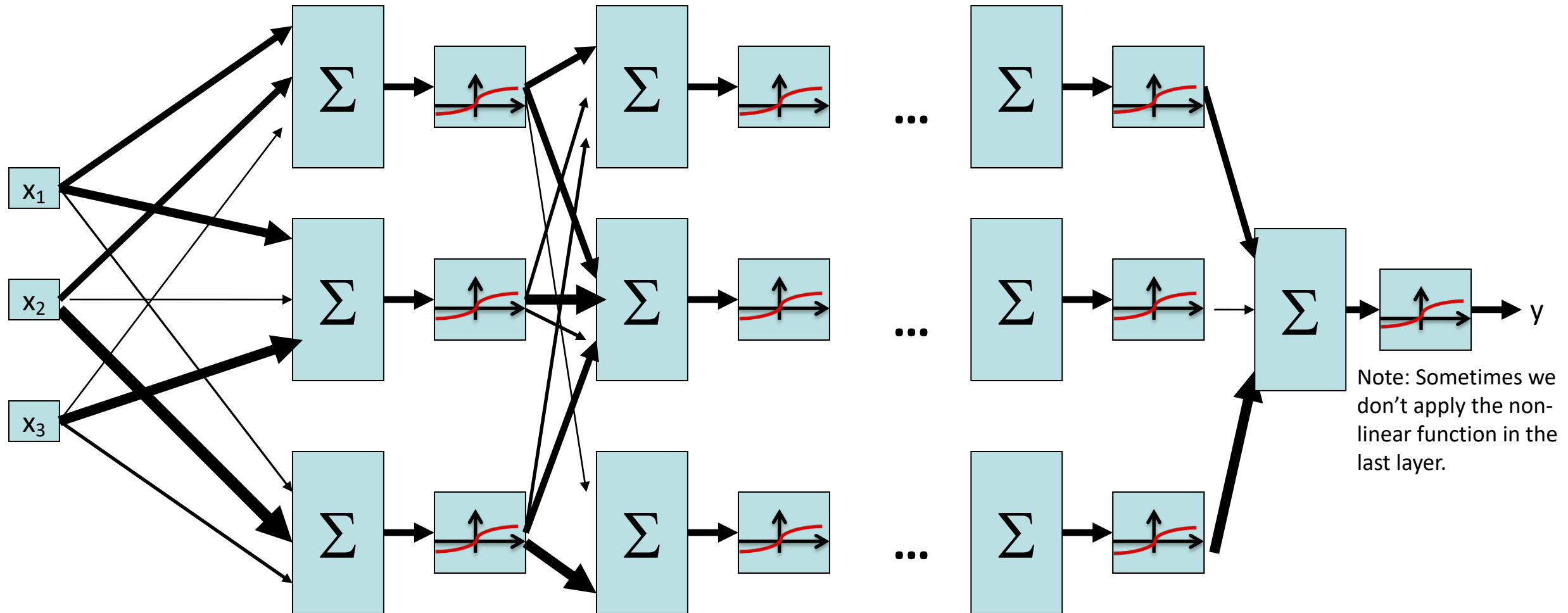 - y has shape (1, 1). Output of the final neuron.

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

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

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

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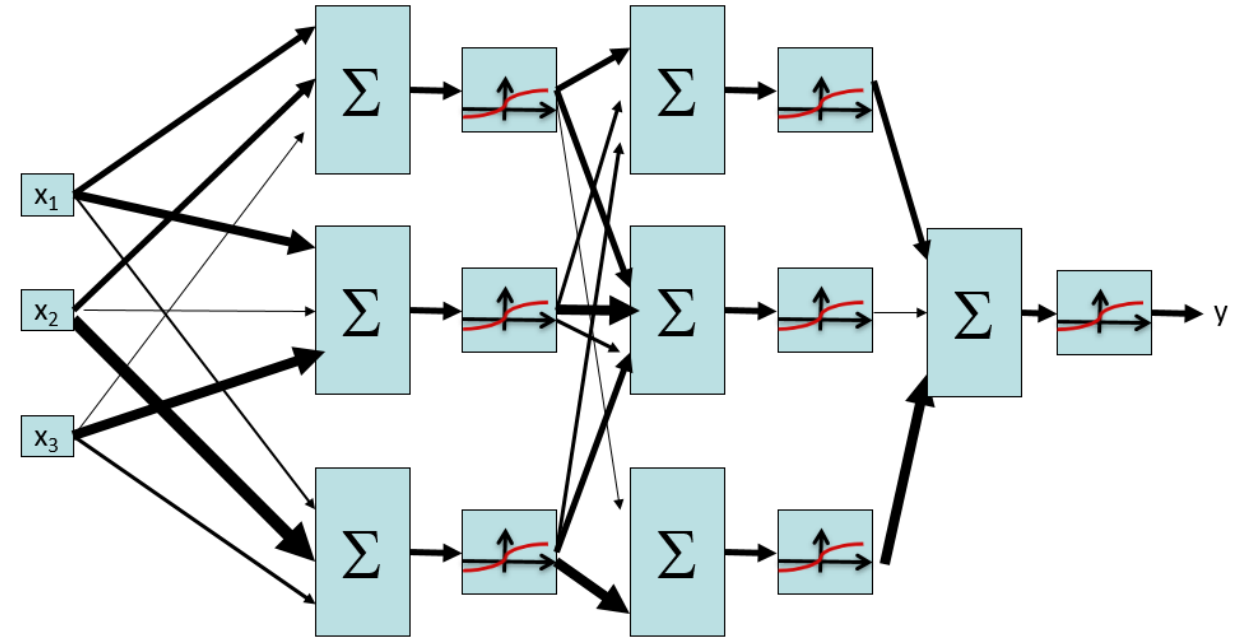
$$\phi(x \times W_{\text{layer 1}}) = h_{\text{layer 1}}$$

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

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

Multi-Layer Neural Network

Note: Sometimes we don't apply the non-linear function in the last layer.

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

Common Activation Functions

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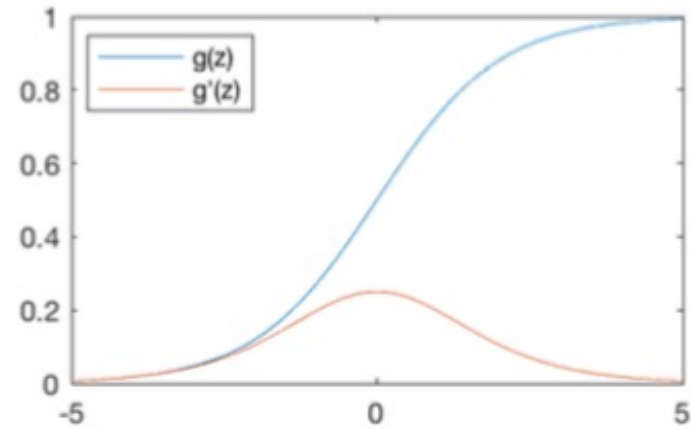
$$g(z) = \frac{1}{1 + e^{-z}}$$

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

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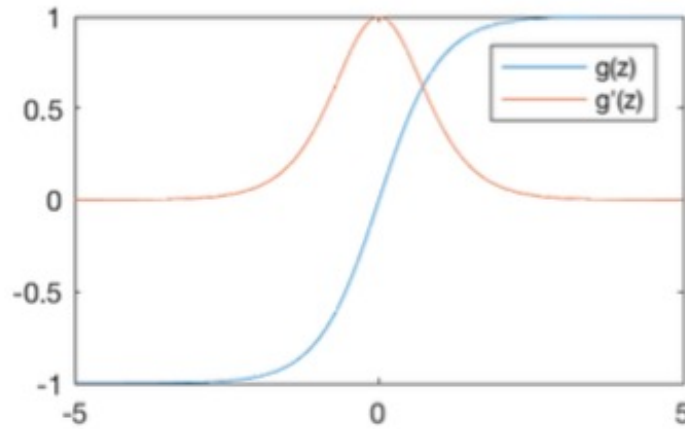
$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

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

Rectified Linear Unit (ReLU)
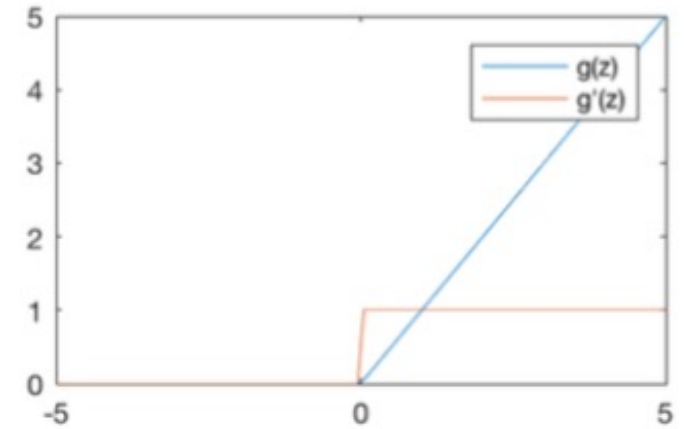
$$g(z) = \max(0, z)$$

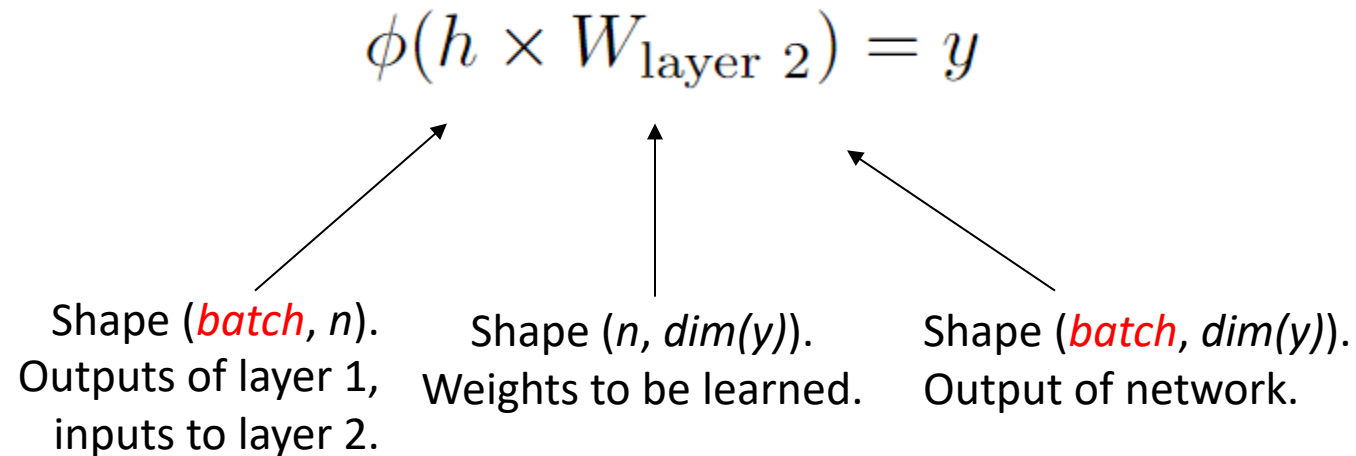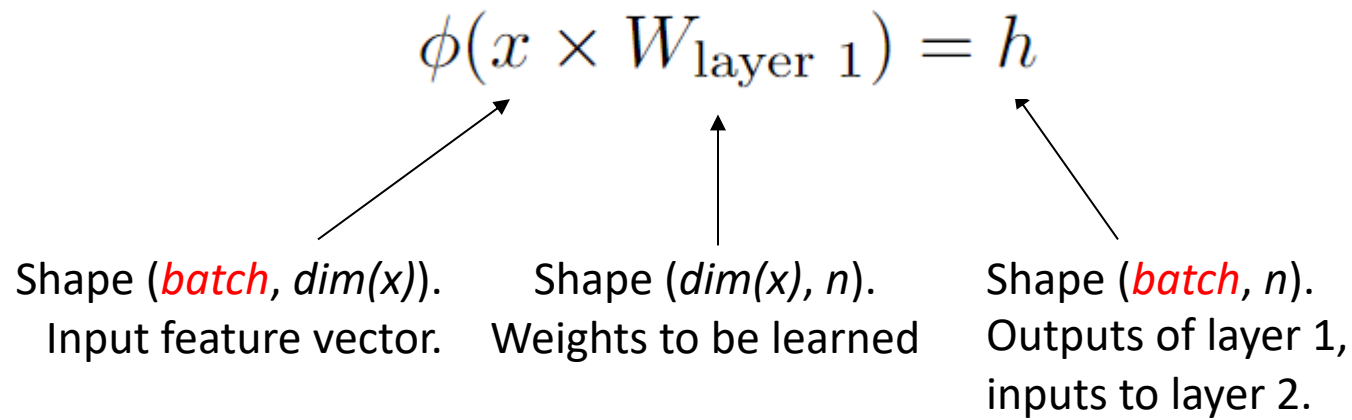$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

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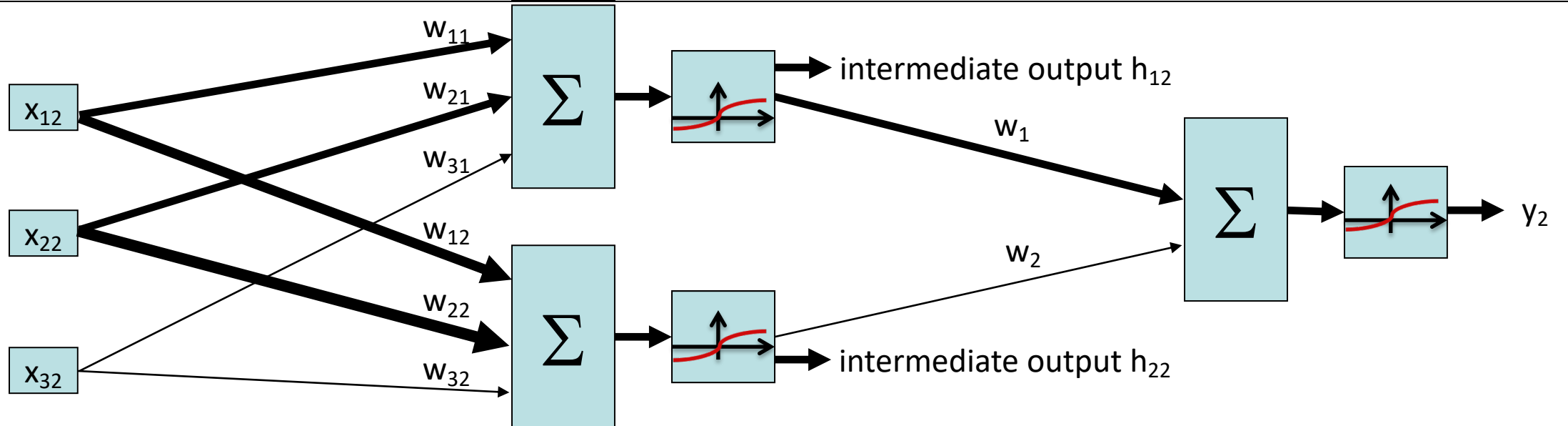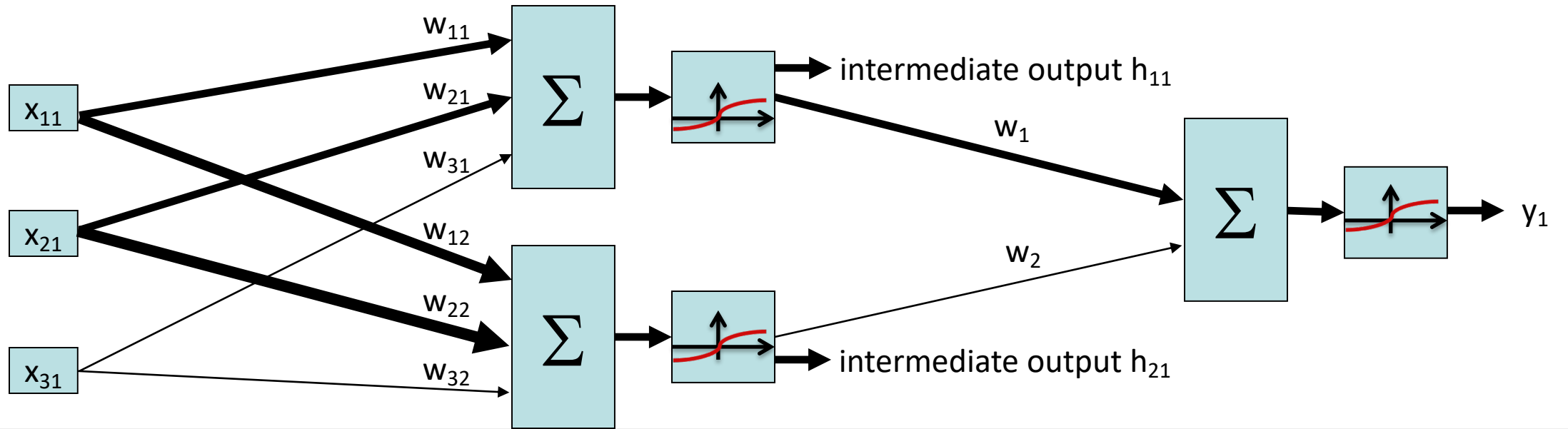
Batch Sizes
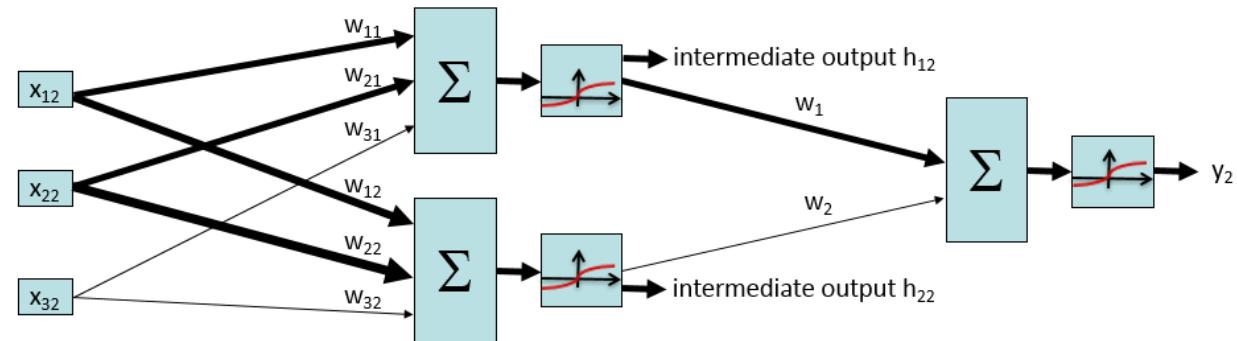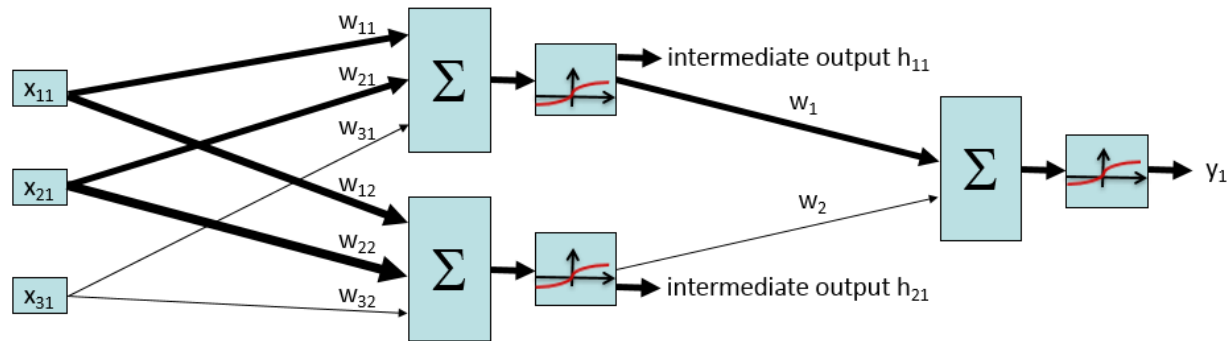
$$y_1 = \phi(w_1 h_{11} + w_2 h_{12})$$
$$= \phi(w_1 \phi(w_{11} x_{11} + w_{21} x_{12} + w_{31} x_{13}) + w_2 \phi(w_{12} x_{11} + w_{22} x_{12} + w_{32} x_{13}))$$
$$y_2 = \phi(w_1 h_{21} + w_2 h_{22})$$
$$= \phi(w_1 \phi(w_{11} x_{21} + w_{21} x_{22} + w_{31} x_{23}) + w_2 \phi(w_{12} x_{21} + w_{22}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

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

Batch Sizes

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

Rewriting in matrix form:

$$\phi\left(\begin{bmatrix} x_{11} & x_{21} & x_{31} \\ x_{12} & x_{22} & x_{32} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \right)$$

$$= \phi\left(\begin{bmatrix} w_{11} x_{11} + w_{21} x_{21} + w_{31} x_{31} & w_{12} x_{11} + w_{22} x_{21} + w_{32} x_{31} \\ w_{11} x_{12} + w_{21} x_{22} + w_{31} x_{32} & w_{12} x_{12} + w_{22} x_{22} + w_{32} x_{32} \end{bmatrix} \right)$$

$$= \begin{bmatrix} h_{11} & h_{21} \\ h_{12} & h_{22} \end{bmatrix}$$

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

Batch Sizes

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

Shape (*batch*, *dim(x)*).
Input feature vector.

Shape (*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*, *dim(y)*).
Weights to be learned.

Shape (*batch*, *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.

Training Neural Networks
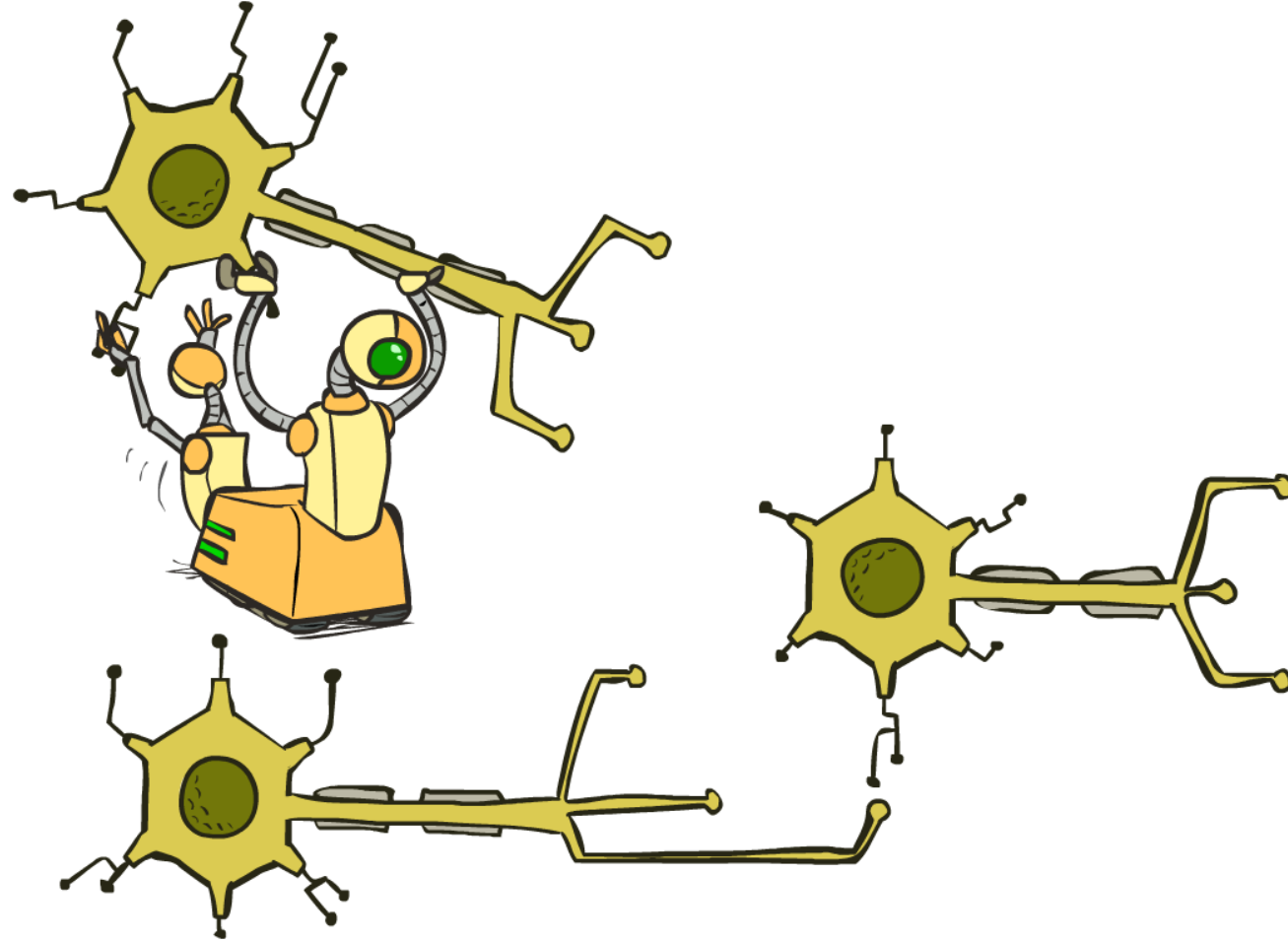
Training Neural Networks

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

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

- 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 ith data point in the training dataset
  - 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 → high loss value
  - If classifier gives p near 1:
    The model was confident in guessing y=1
    Good model → low loss value
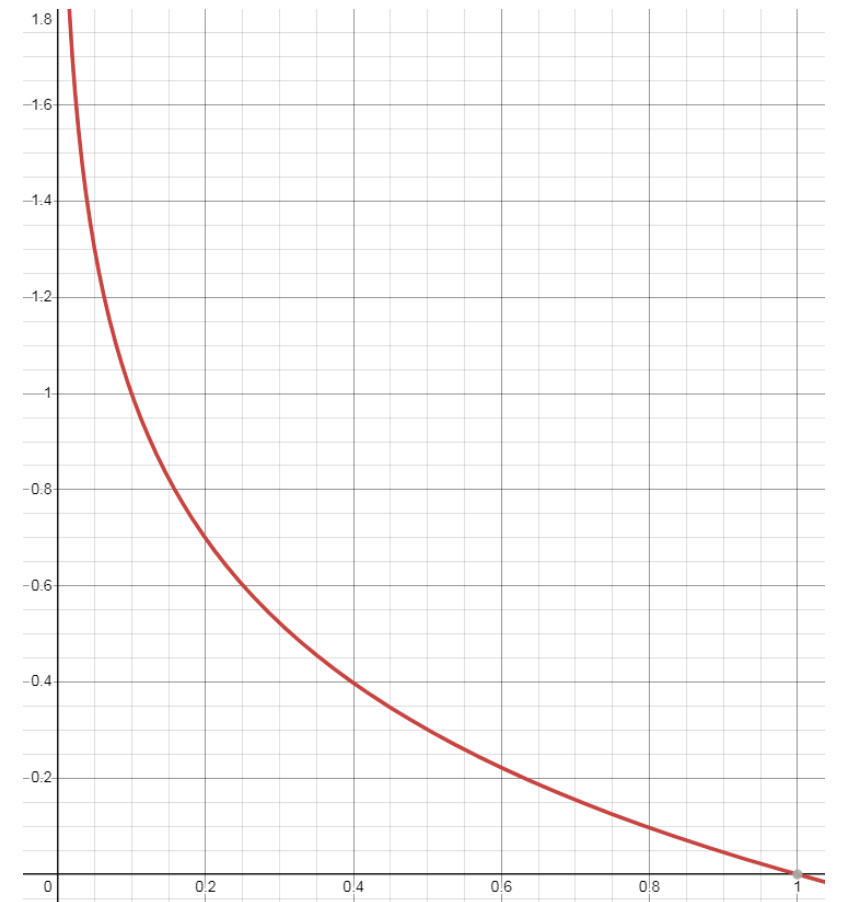
Log Loss Function

- 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
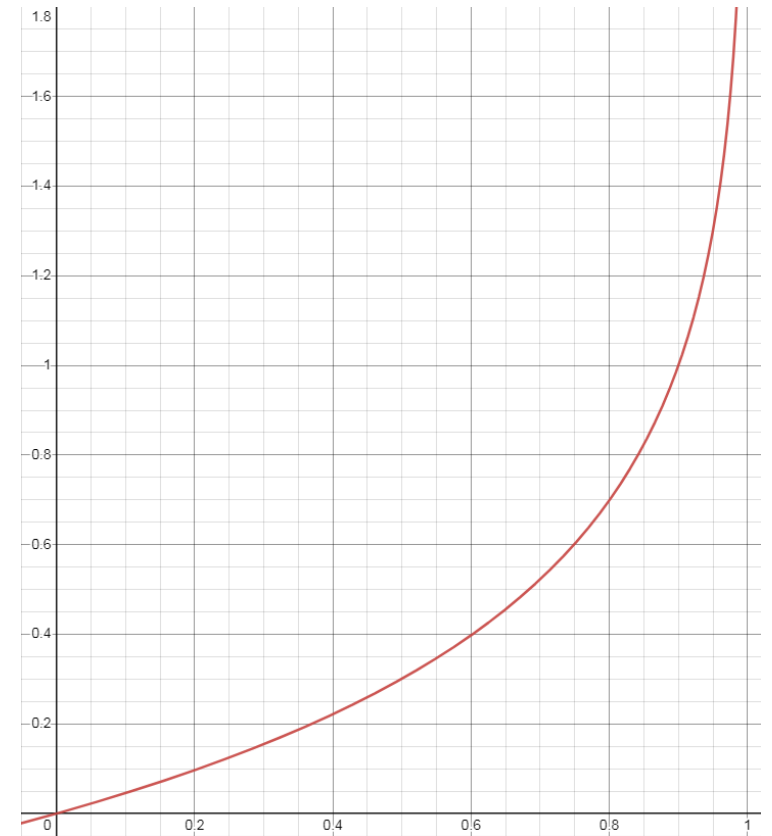
Log Loss Example

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

- 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 INFOF311

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 × w$_2$

$$\frac{\partial g}{\partial a} = \frac{\partial g}{\partial b}\frac{\partial b}{\partial a} = 1 \cdot w_2 = 3 \qquad \frac{\partial g}{\partial w_2} = \frac{\partial g}{\partial b}\frac{\partial b}{\partial w_2} = 1 \cdot a = 16$$

- 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$_1$

$$\frac{\partial g}{\partial w_3} = \frac{\partial g}{\partial c}\frac{\partial c}{\partial w_3} = 1 \cdot 5 = 5$$
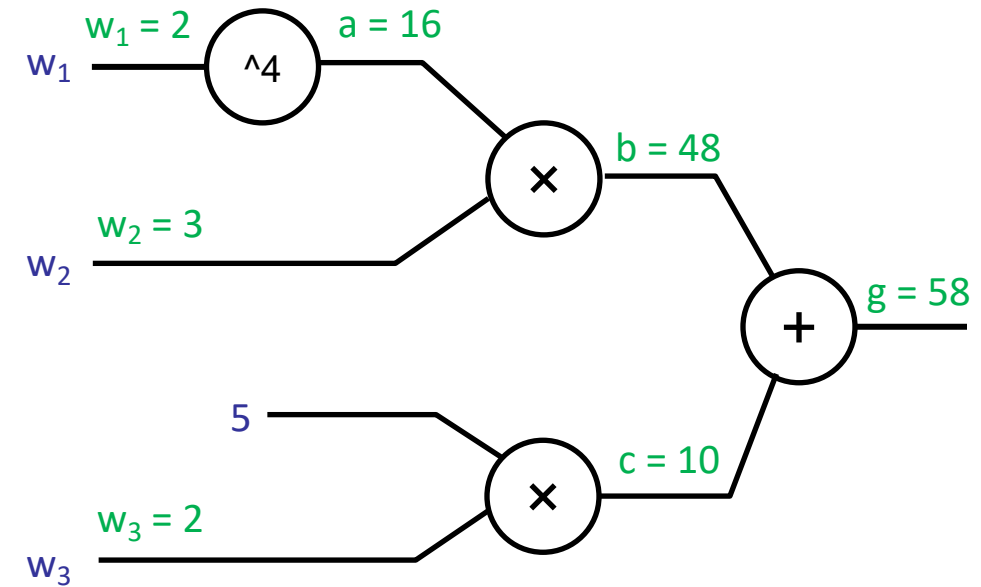
Properties of Neural Networks
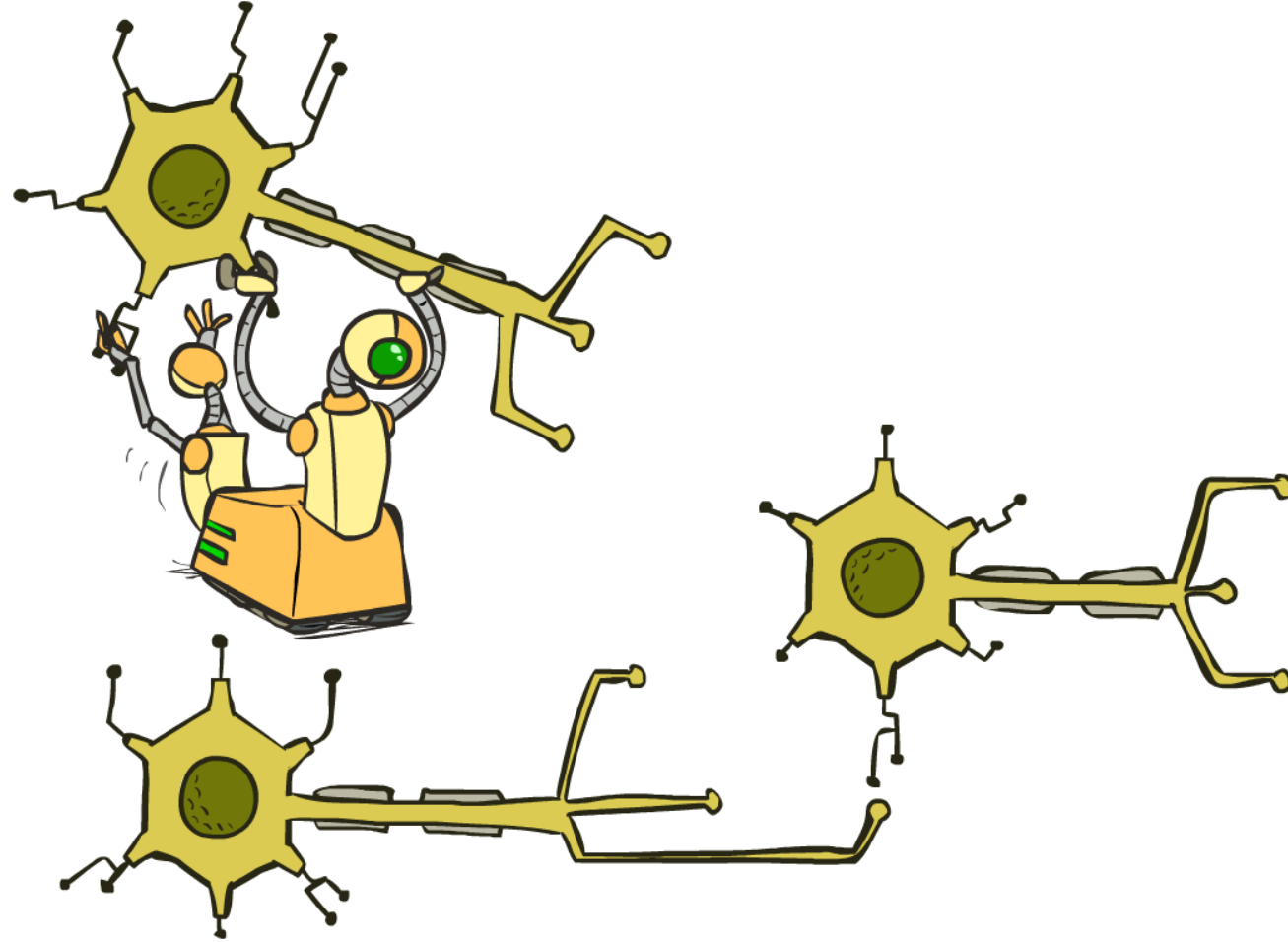
Neural Networks Properties

- 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 nonconstant*, 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.

- <u>In words:</u> 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*

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 $\qquad \max_{w} \; ll(w) = \max_{w} \; \sum_{i} \log P(y^{(i)} | x^{(i)}; w)$

- 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 INFOF311)

Where to Go Next?
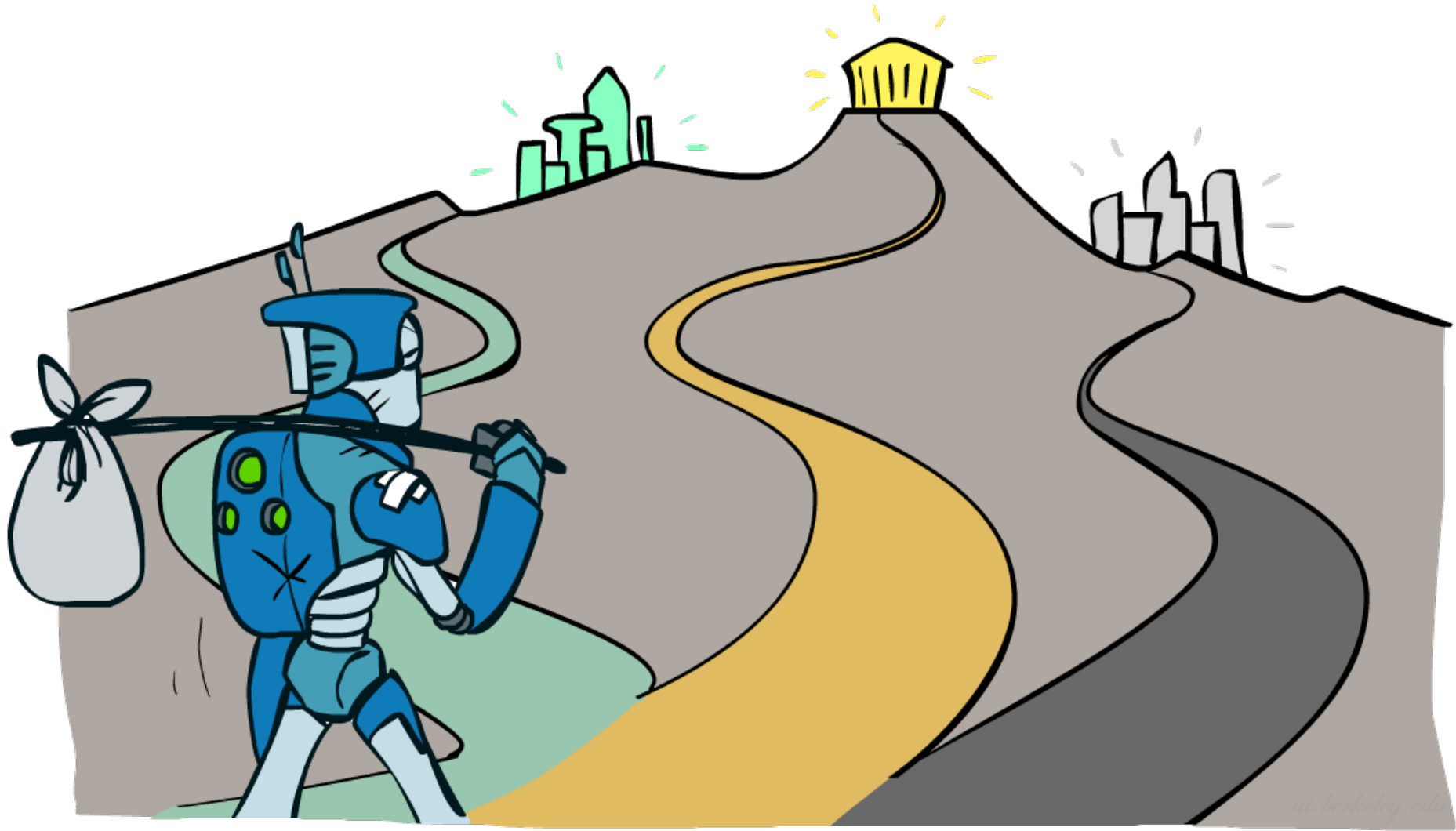
Where to go next?

- Congratulations, you've seen the basics of modern AI
    - … and done some amazing work putting it to use!

- How to continue:
    - Machine learning: INFOF422
    - Multi-agent learning: INFOF409
    - Swam intelligence: INFOH414
    - Data Science: INFOF515, INFOF440
    - Optimization: INFOF413, INFOF424
    - Bioinformatics: INFOF439
    - NLP: INFOY004
    - … and more; ask if you're interested
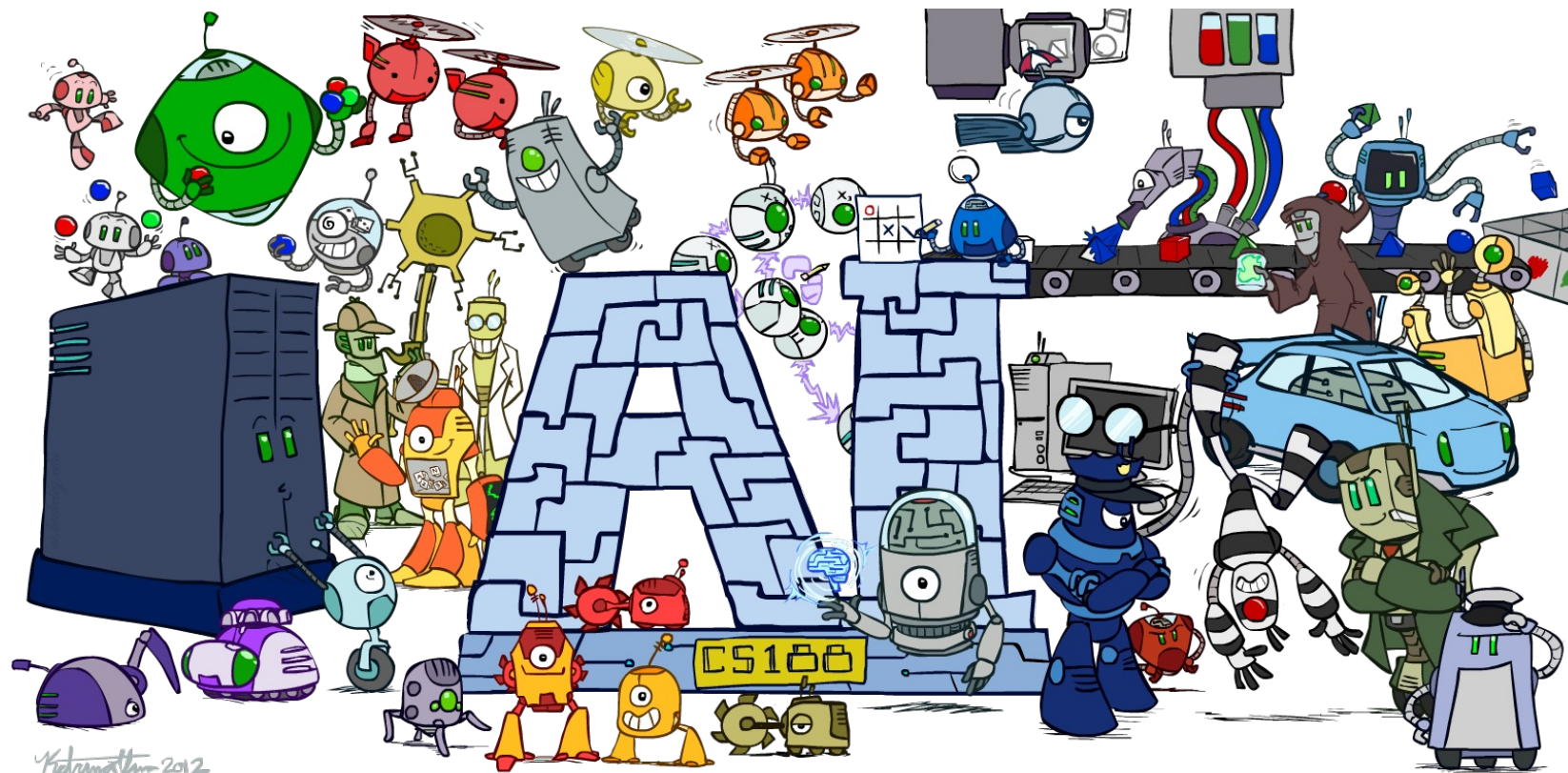
Ketrina Yim

CS188 Artist

That's It!