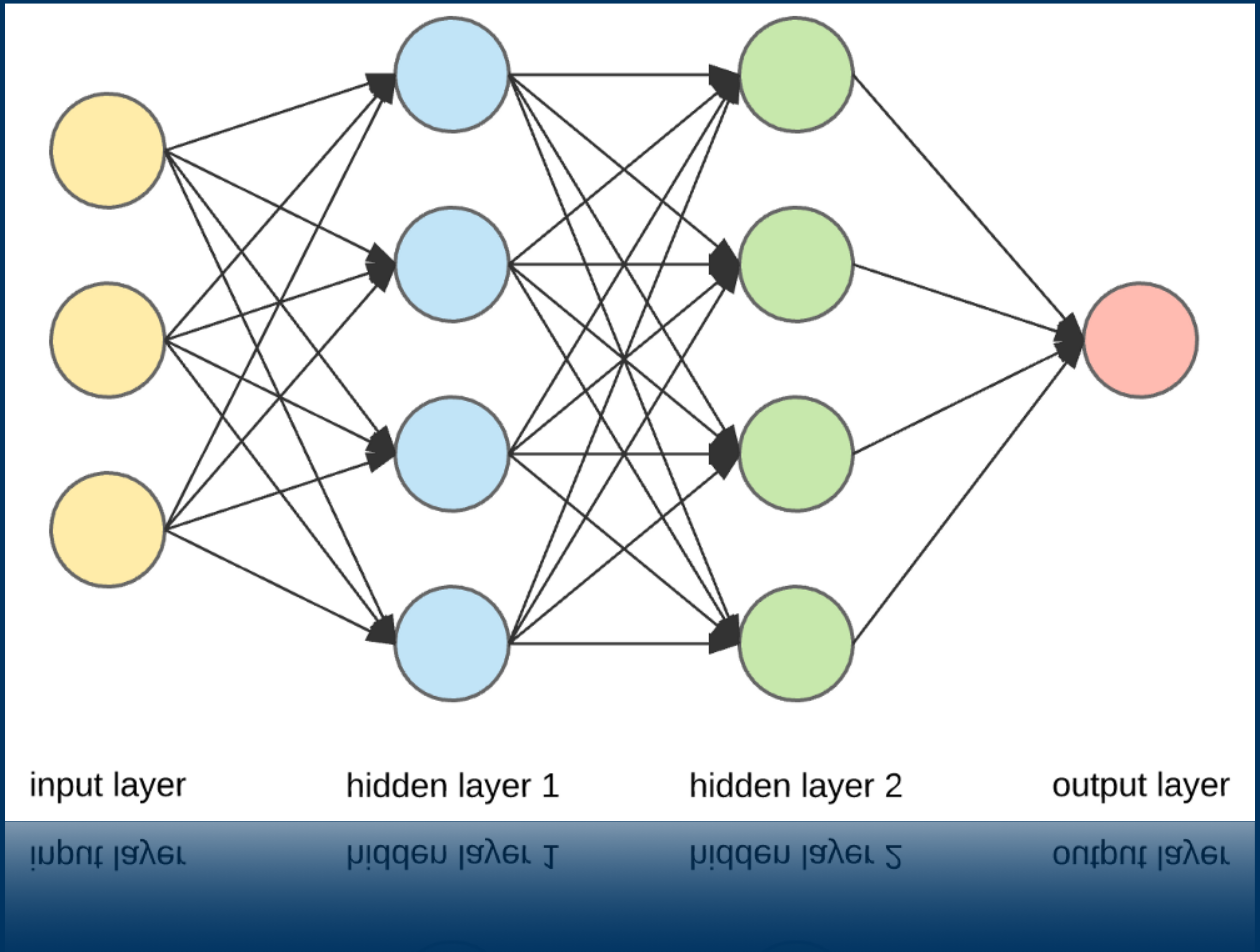


Neural Networks II

CSCI-P556 Applied Machine Learning
Lecture 14

D.S. Williamson

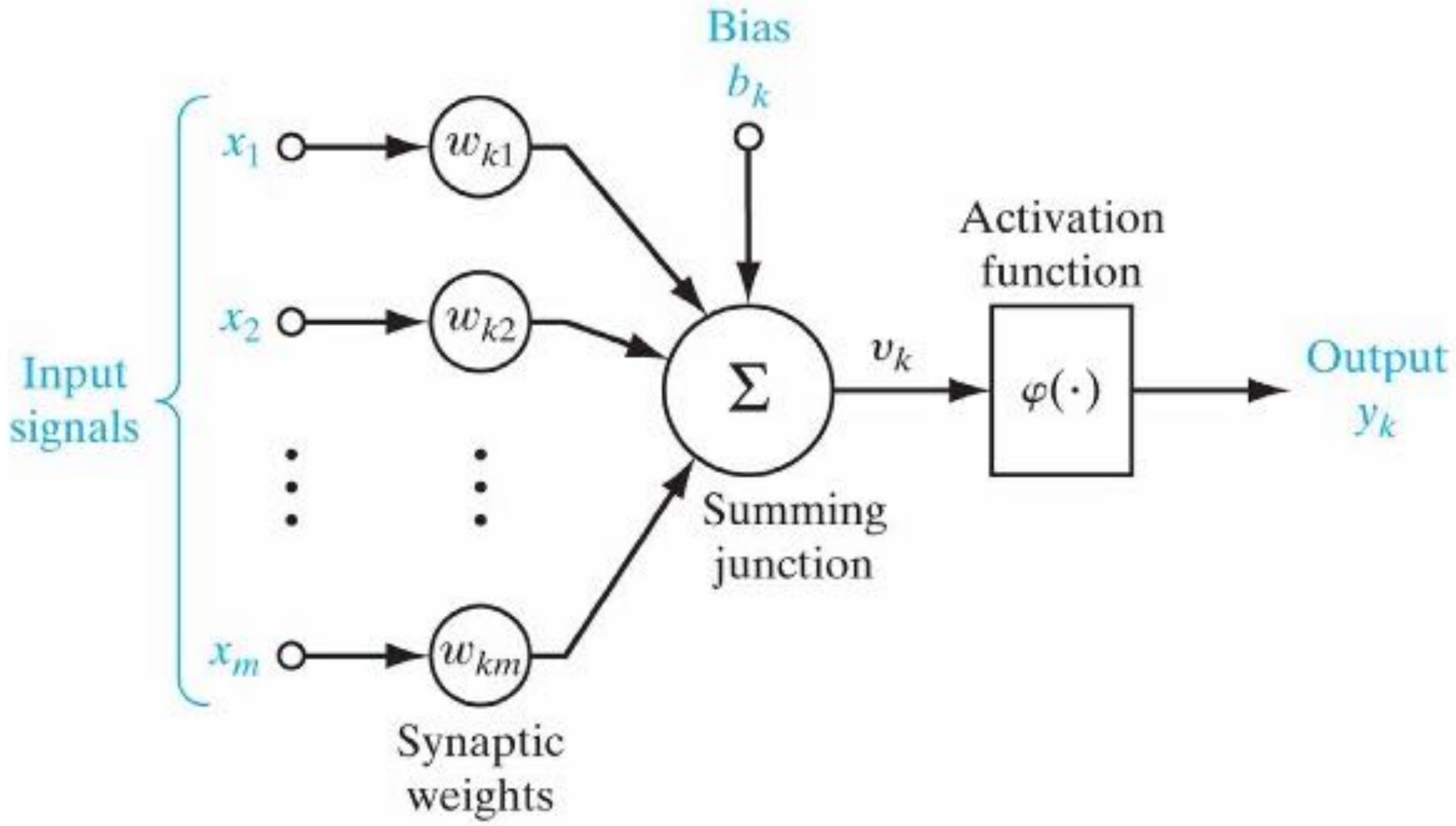
Agenda and Learning Outcomes

Today's Topic(s)

- **Topic(s): Neural Networks**
  - Multi-layer Perceptrons (MLP)
  - Deep Neural Networks
  - Backpropagation
- **Announcements**

Recall: Perceptron Neuron Model

Modified version of McCulloch-Pitts neuron



- **Perceptrons** use ***real-valued inputs***.
- Perceptrons are also used for learning.
- Invented by Rosenblatt in 1957
- Useful for binary classification

$$x_i \in \mathbb{R}$$ Real-valued inputs

$$v = \sum_{i=1}^{m} w_i x_i + b$$ Activation potential

$$y = \phi(v)$$ Output

$$\phi(v) = \begin{cases} 1, & \text{if } v \geq 0 \\ -1, & \text{if } v < 0 \end{cases}$$ Activation function

Recall: Decision Boundary

$$\phi(v) = \begin{cases} 1, & \text{if } v \geq 0 \\ -1, & \text{if } v < 0 \end{cases}$$

Perceptrons as classifiers

- The ***decision boundary*** of a perceptron is the line (or hyperplane) where the activation potential equals 0. This is based on the activation function of a perceptron
- For a 2-D input space with a bias: $$g(x_1, x_2) = w_1 x_1 + w_2 x_2 + b = 0$$

=> After re-writing the equation we can see that it follows the equation of a line

$$x_2 = -\frac{w_1}{w_2} x_1 - \frac{b}{w_2}$$

- The discriminant function is a line with slope $-w_1/w_2$ and intercept $-b/w_2$
- The distance of the function to the origin is $|b|/||\mathbf{w}||$, where

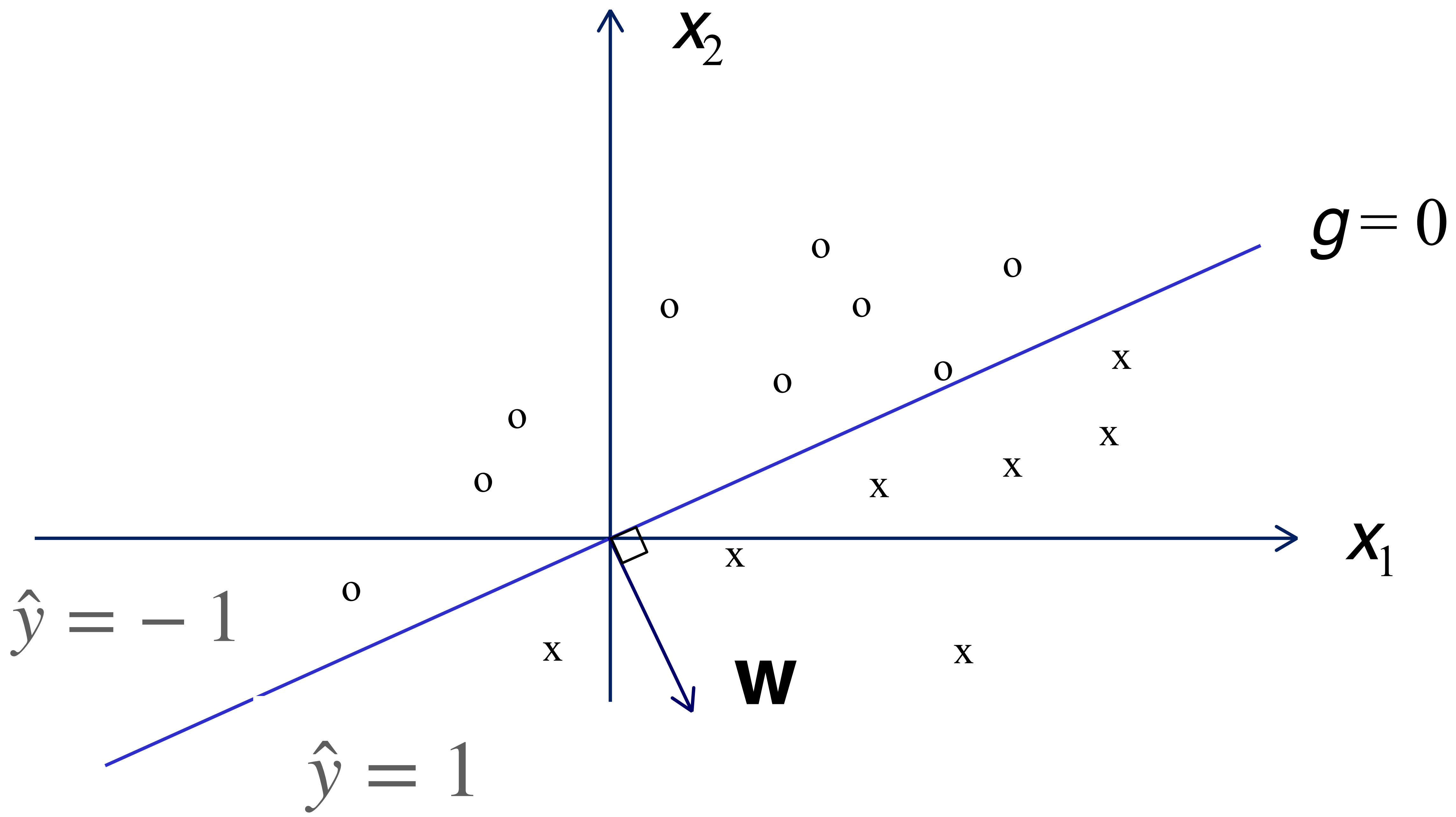
$$||\boldsymbol{w}|| = \sqrt{w_1^2 + w_2^2}$$

Recall: Decision Boundary

Perceptrons as classifiers

$$x_2 = -\frac{w_1}{w_2}x_1 - \frac{b}{w_2}$$

- For an m-dimensional input space, the decision boundary is an (m-1)-dimensional hyperplane perpendicular to **w**.
  - By definition, the weight vector is orthogonal to the decision boundary. **Why**? Recall: inner product
- The hyperplane separates the input space into two halves
  - One half having $\hat{y}$ = 1
  - The other half having $\hat{y}$ = -1
- When b=0, the hyperplane goes through the origin



Recall: Perceptron Learning Rule: Finding Weights

Single Perceptron, Multiple inputs

- How do we (iteratively) change the weights if they are too small or too large?
  - Weights must be initialized (more on this next week)

$$\mathbf{w}^{n+1} = \mathbf{w}^n + \nabla(\mathbf{w})$$
$$= \mathbf{w}^n + \eta(y - \hat{y})\mathbf{x}$$

$n:$ iteration number

$\eta:$ step size or learning rate

Actual output

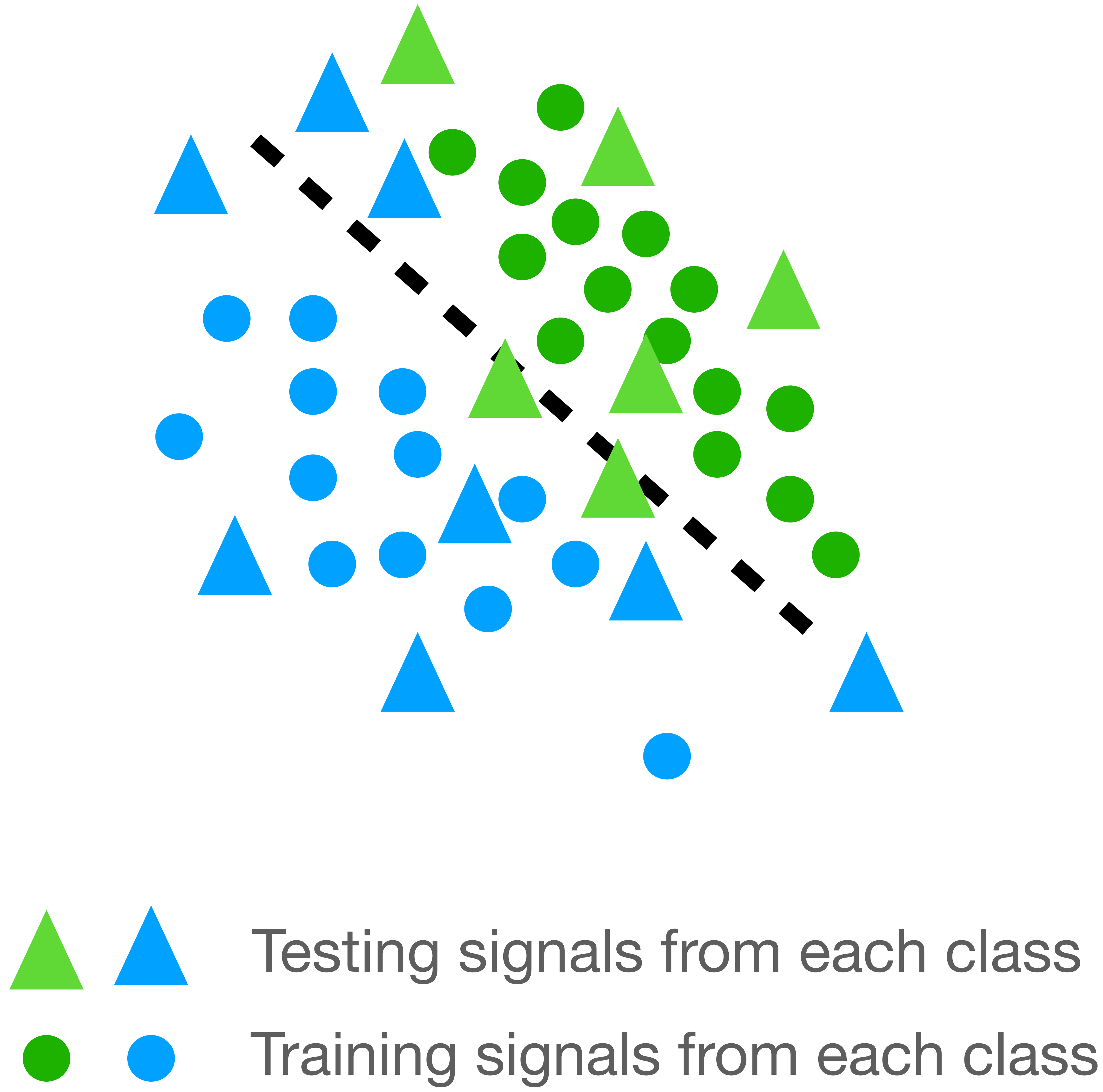
Desired output

- Note:
  - The outputs are bipolar {-1,1}
  - Thus, [y-$\hat{y}$] is either 0 (correct), -2 (too strong), or 2 (too weak)

Generalization Once Trained

Perceptrons during testing

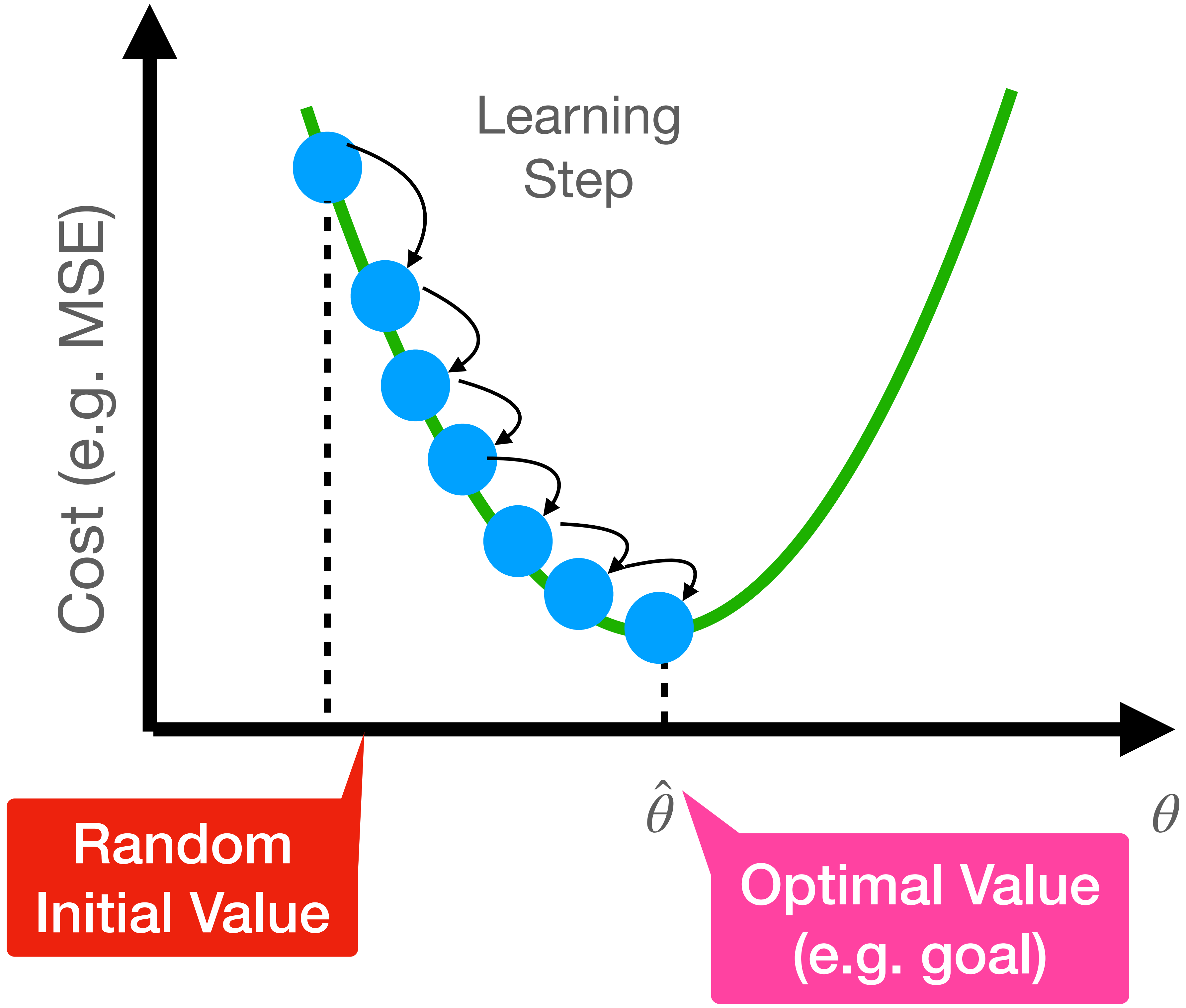
- Recall the definition of ***generalization***
  - **Definition**: Performance of a learning machine on test patterns not seen (or used) during training.
- Perceptrons generalize by deriving a decision boundary in the input space.
  - The selection of training patterns is thus important for generalization
  - The solution weight vector is not unique. There are infinite possible solutions and decision boundaries.



Gradient Descent vs. Perceptron Learning Rule

The math is the same

- ***Perceptron learning*** is for McCulloch-Pitts neurons (with real inputs), which are ***nonlinear***
  - Gradient descent (as discussed) is for linear neurons (e.g. linear regression)
  - Perceptron learning is for classification
  - Linear regression learning via gradient descent is for estimation (or regression)



Perceptron Learning

$$\phi(v) = \begin{cases} 1, & \text{if } v \geq 0 \\ -1, & \text{if } v < 0 \end{cases}$$

$$v = \sum_{i=1}^{m} w_i x_i + b \qquad y = \phi(v)$$

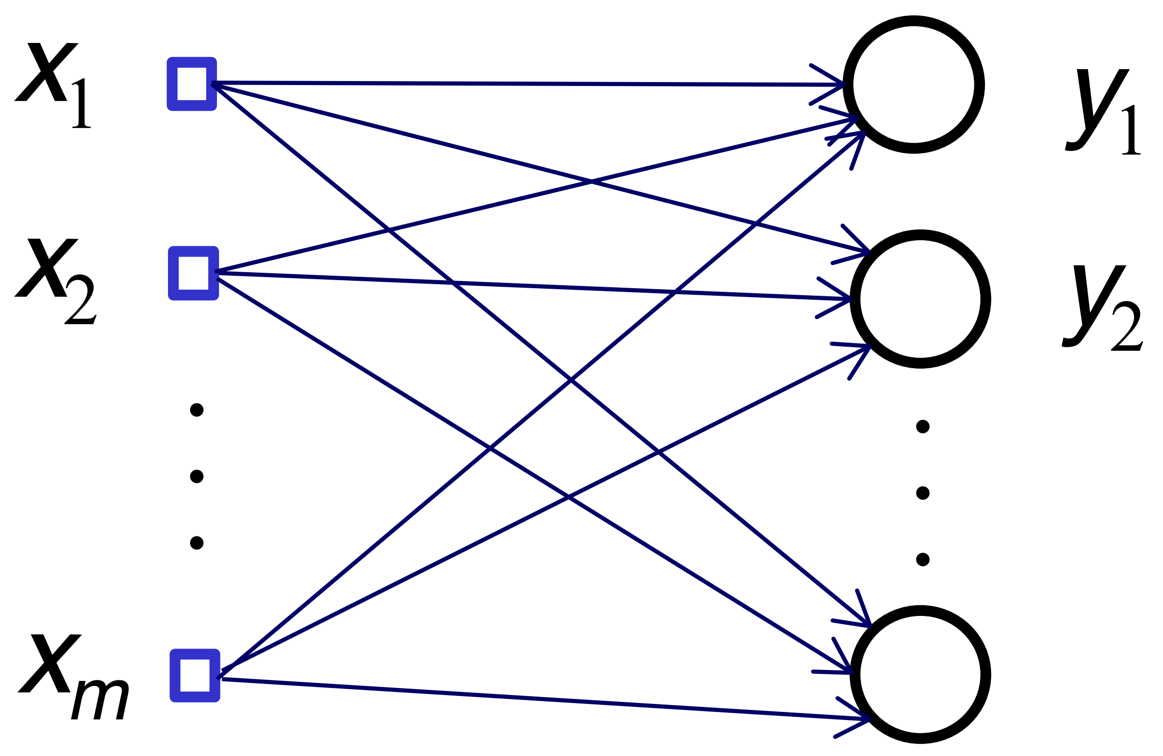
Linear Regression

$$y = \sum_{i=1}^{m} w_i x_i + b \qquad \phi(v) = v$$

Perceptrons for Multi-dimensional outputs

For classification or regression

- Multiple perceptrons can be used when performing multi-class classification (one for each class) or multi-dimensional estimation
  - **Classification**: Perceptron output is 1 when input is from the corresponding class. Its output is 0 otherwise. Each perceptron forms it's own decision boundary
  - **Regression:** Each perception corresponds to one of the output dimensions
- When these perceptrons have the same inputs (but with different weights), this stacking of perceptrons is called a ***layer***

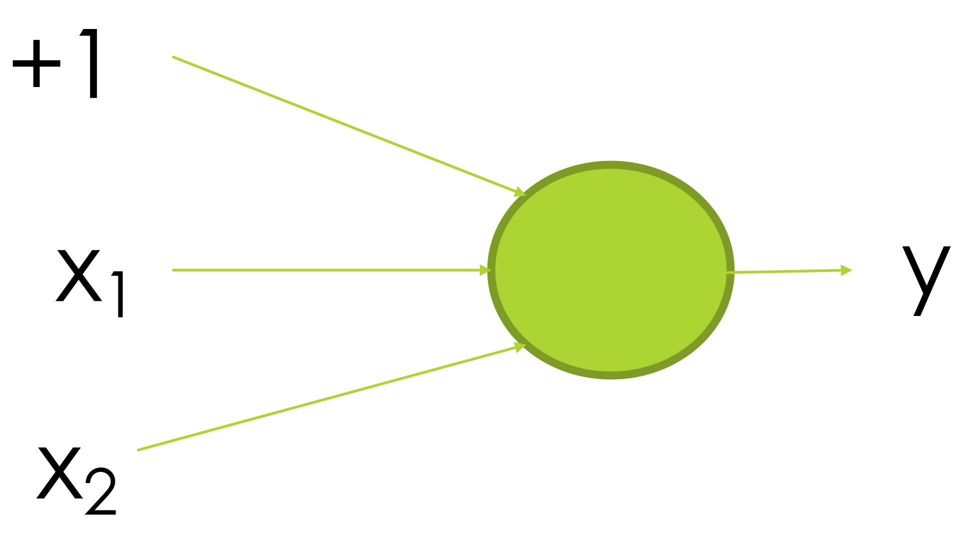


A limitation of single-layer perceptrons

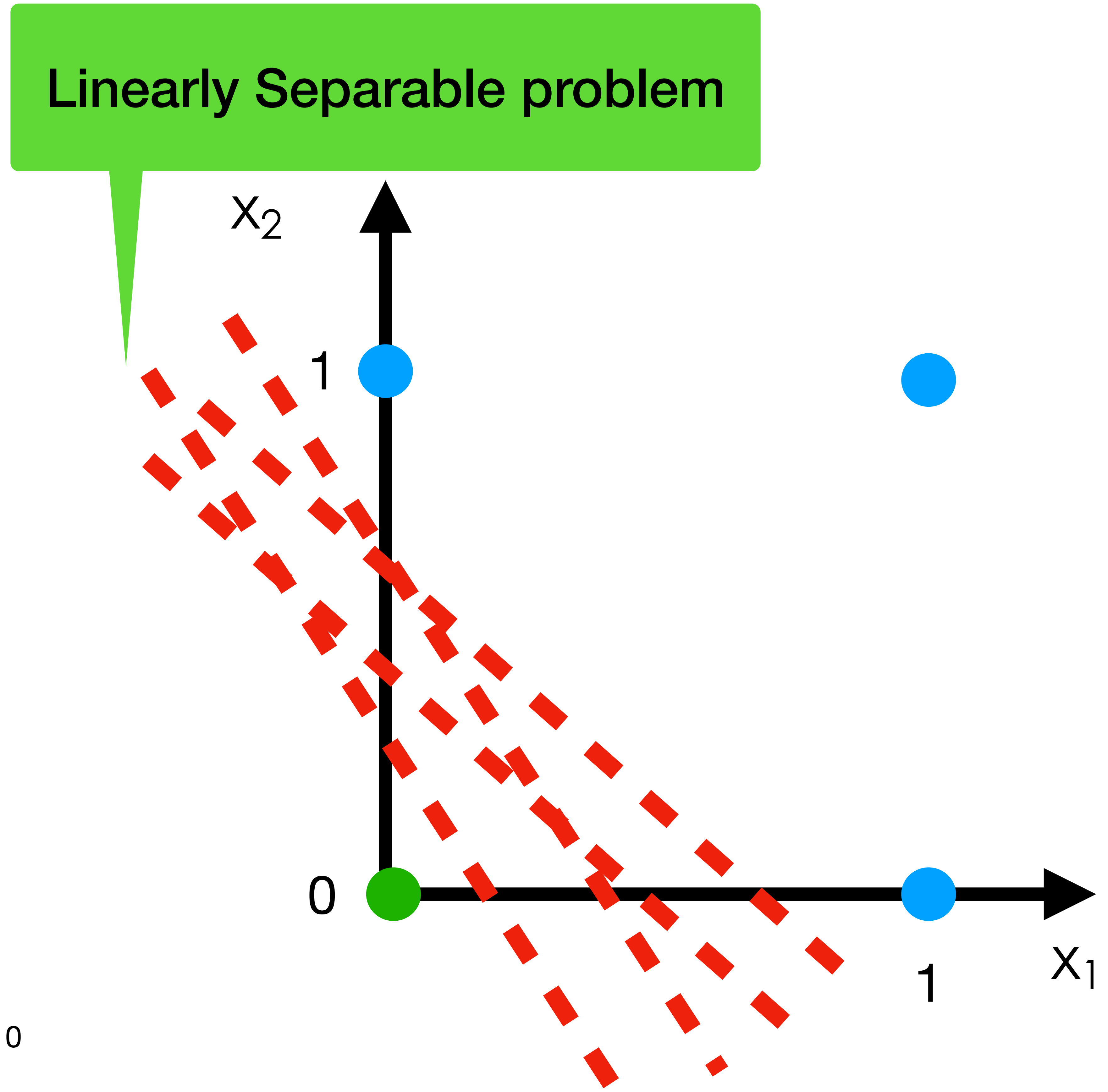
Logic Gate Example: OR Gate

- Single-layer perceptron networks can be used to solve multi-class classification problems, but they are not as useful for problems that are linearly inseparable or linearly-separable problems that require multiple boundaries per class

OR Function



x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1

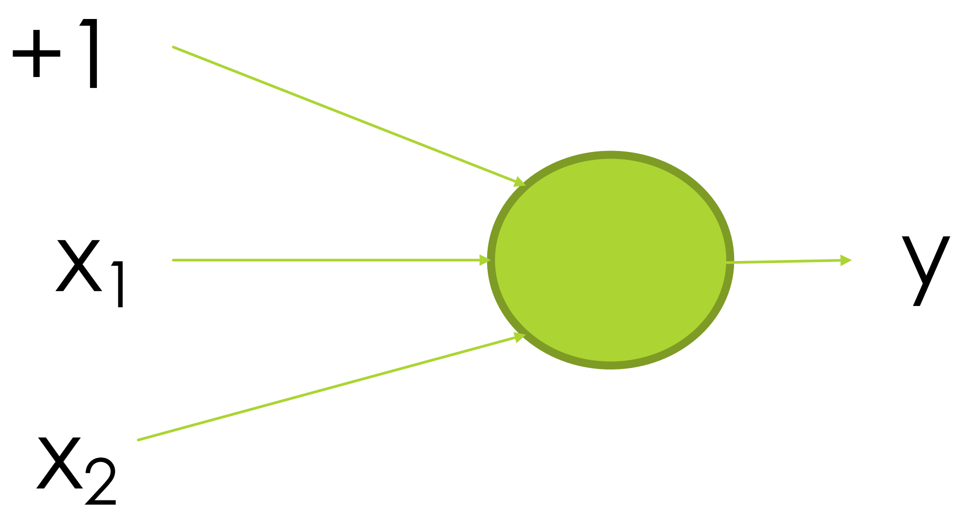


A limitation of single-layer perceptrons

Logic Gate Example: XOR Gate

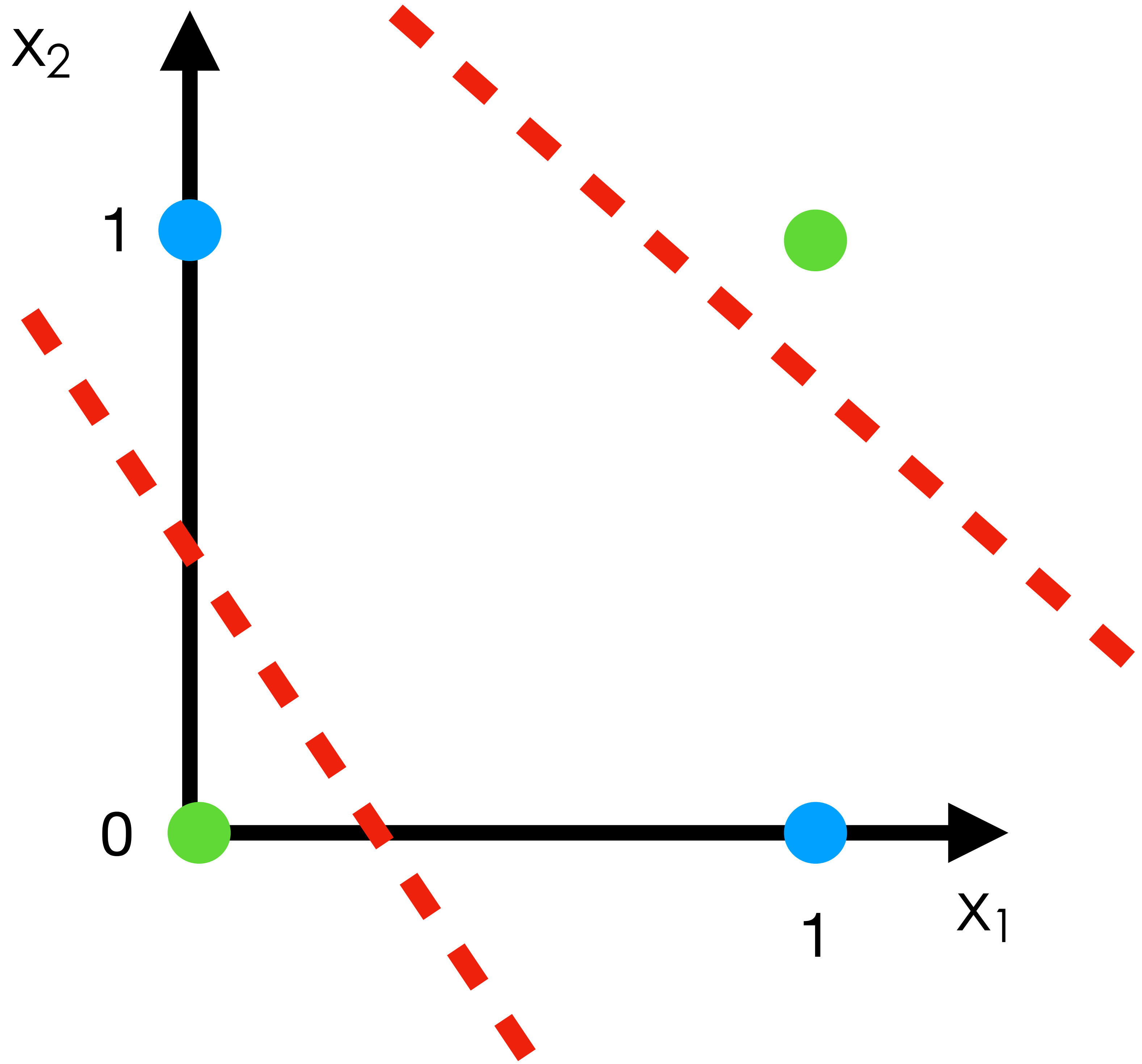
- Single-layer perceptron networks can be used to solve multi-class classification problems, but they are not as useful for problems that are linearly inseparable or linearly-separable problems that require multiple boundaries per class

XOR Function



x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0

A single (layer) perceptron cannot solve this problem



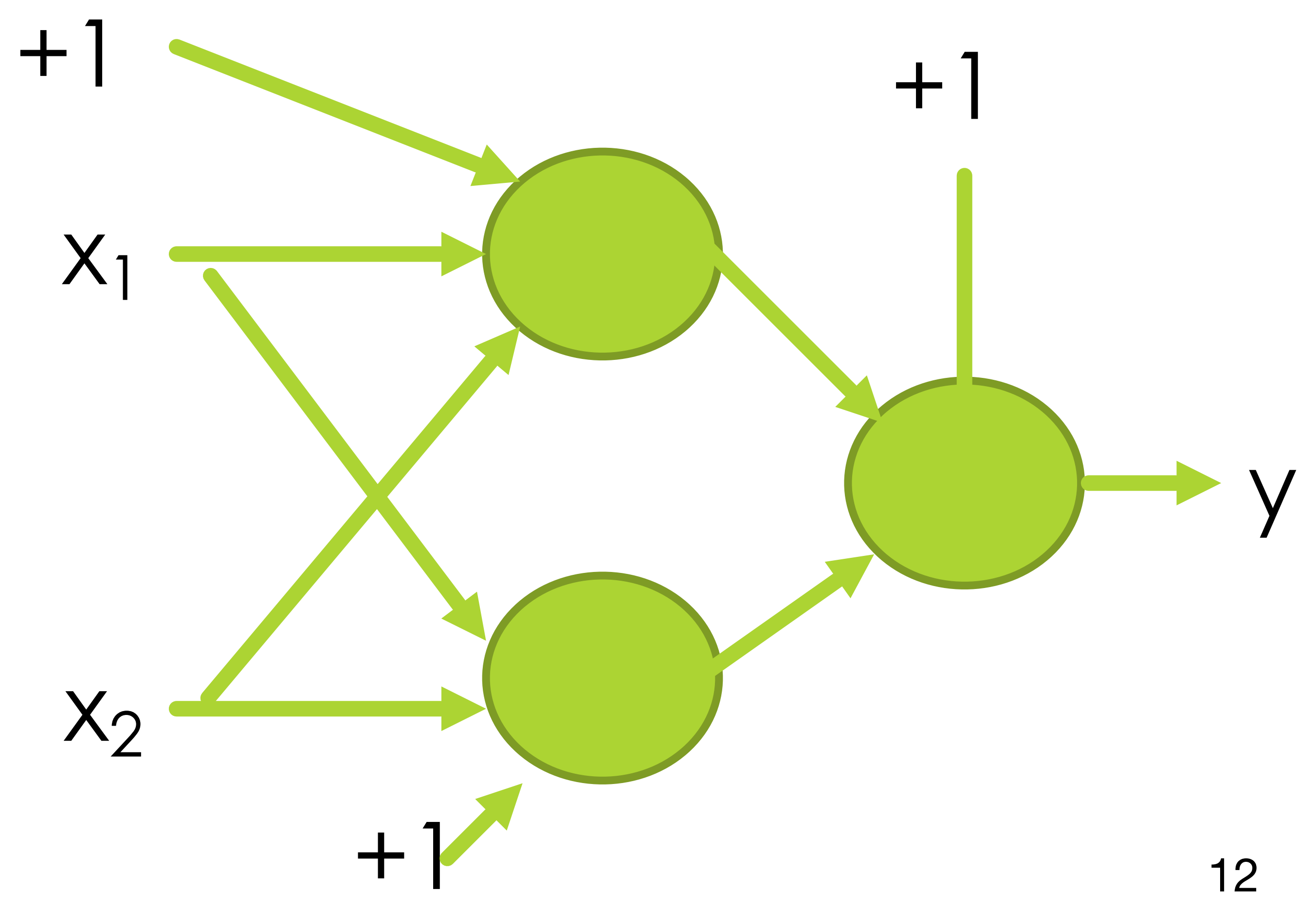
The Case for Multi-Layered Perceptrons (MLP)

Logic Gate Example

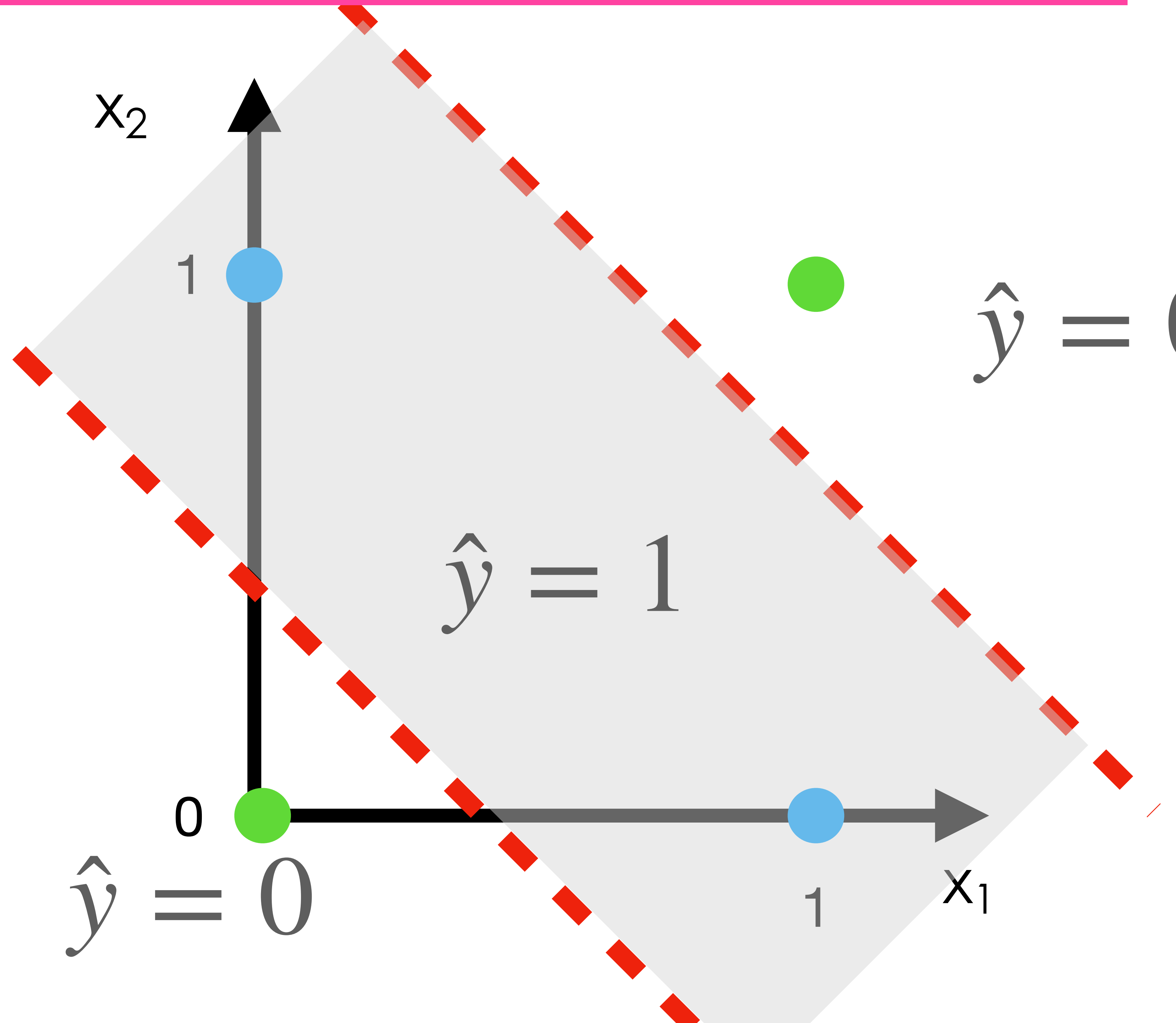
- Adding layers to a network can help solve more complicated problems. The resulting neural network is called a ***Multi-Layer Perceptron (MLP)***

XOR Function

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	0



The decision boundary becomes a decision region

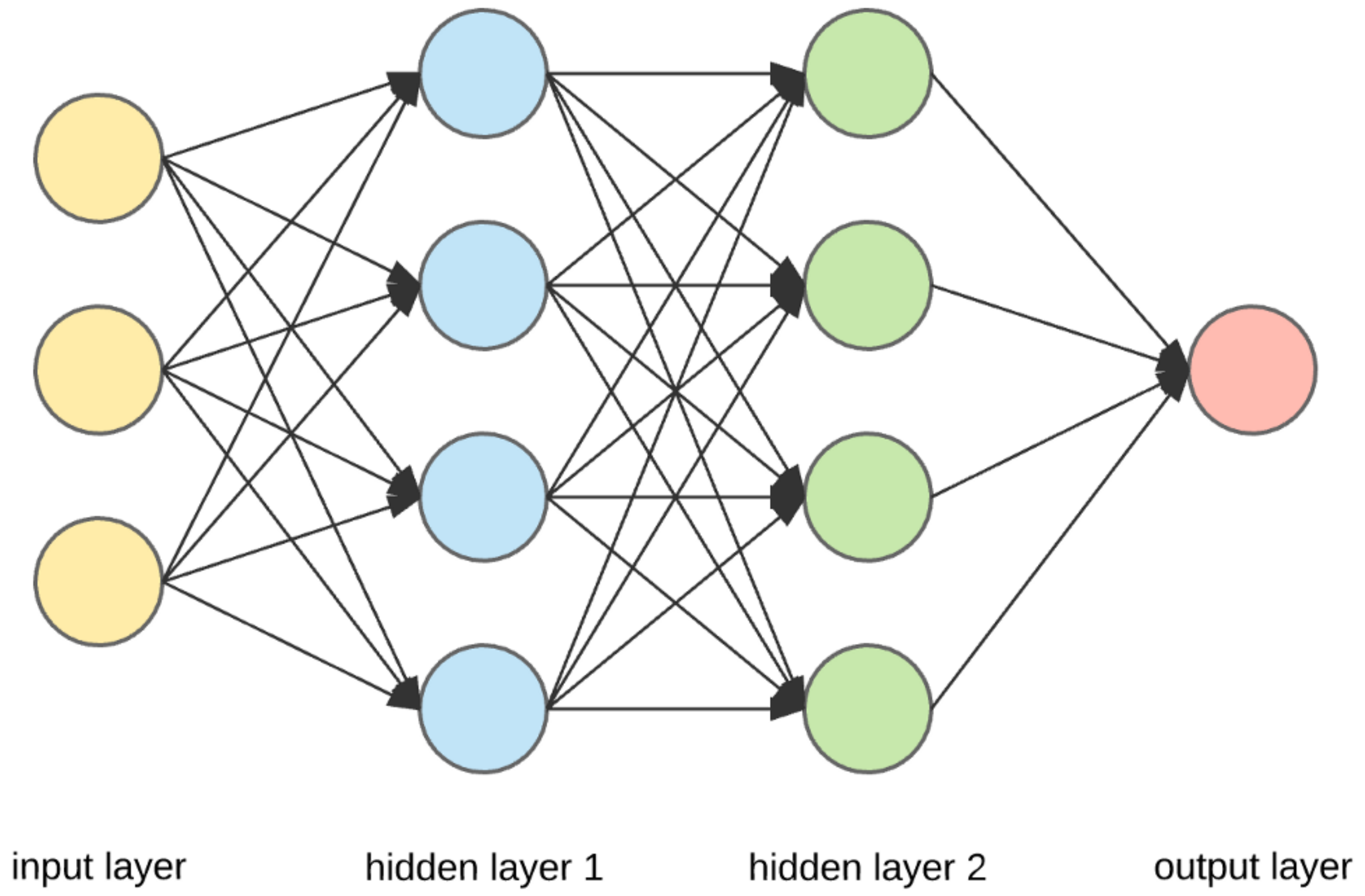


Multi-Layer Perceptron (MLP)

Components of the MLP

- **Input layer**: not really a layer, but serves to represent the inputs to a network
- **Hidden layers**: Layers where the output goes to another layer of neurons
- **Output layer**: Final layer of network, where output(s) are computed

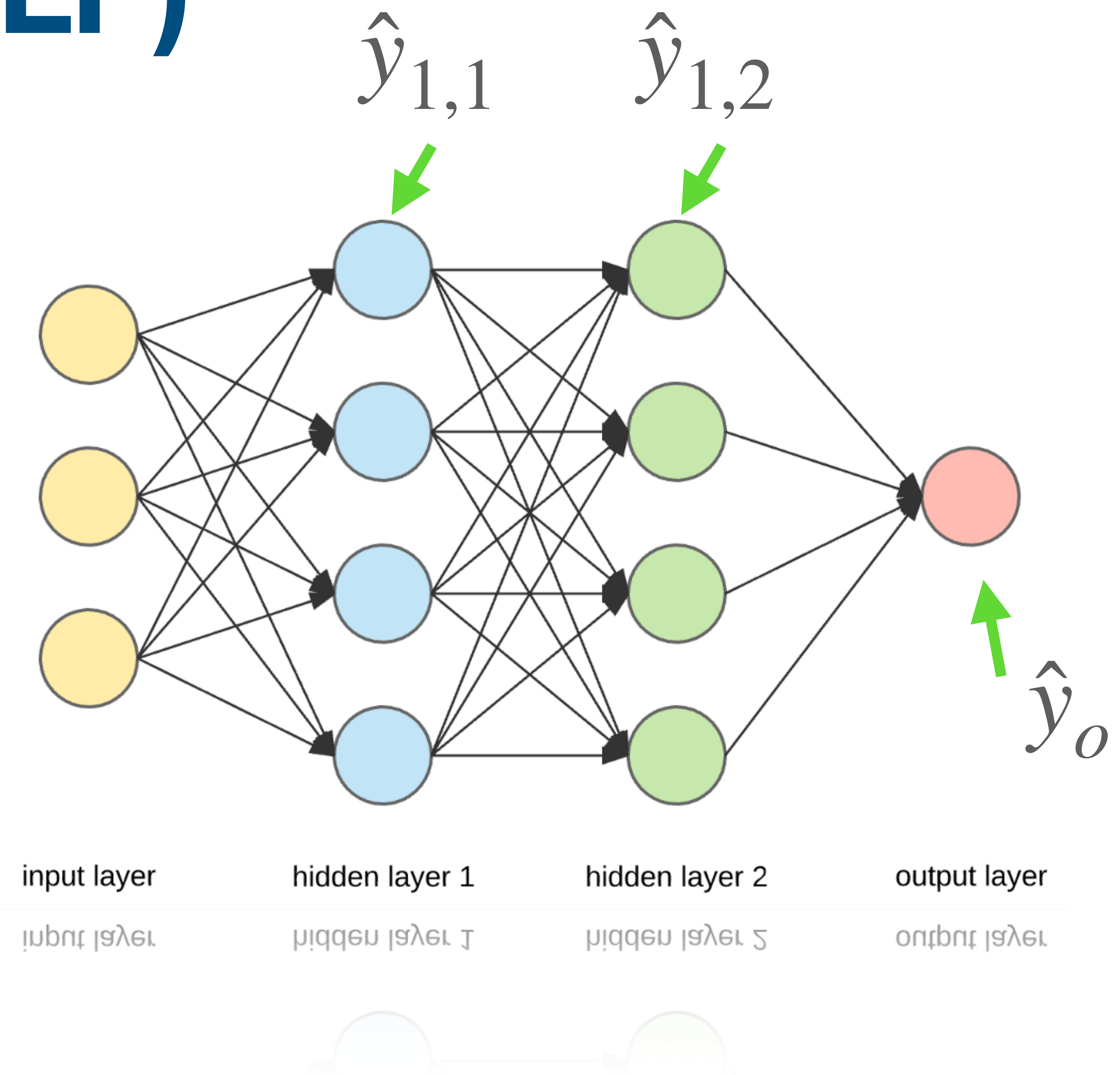
- ***Deep neural networks(DNN)*** have two or more hidden layers (or three or more layers, excluding the input layer)
  - This is an example of a three-layer DNN. The input layer is not counted.



Multi-Layer Perceptron (MLP)

Computing Output(s): Forward Pass

- The output(s) of the network is(are) computed in a layer-wise fashion. This is known as the ***forward pass***
    - The output from layer one is first computed and this becomes the input to the next layer
    - This continues until the output(s) is(are) computed



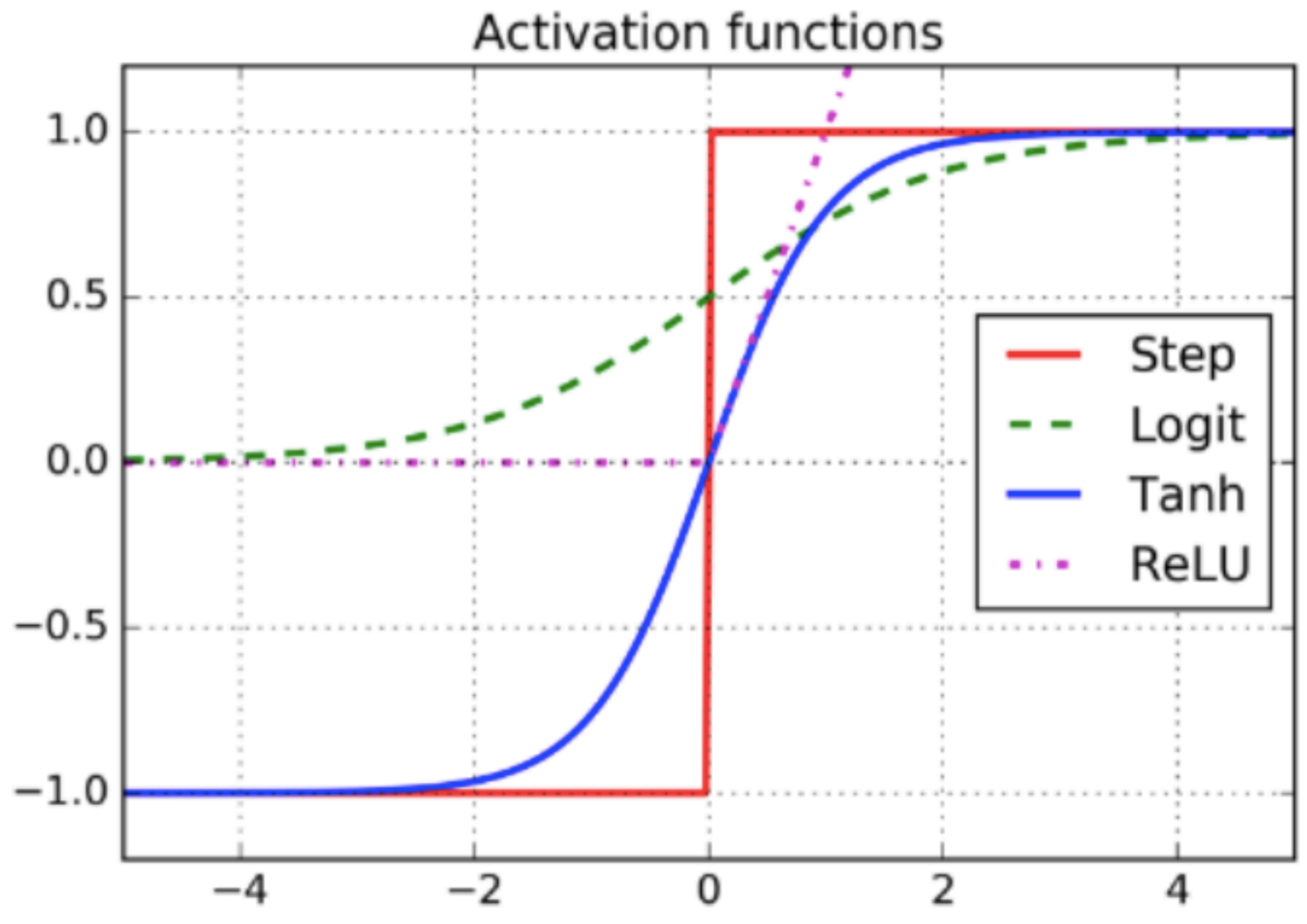
$$\hat{y}_{1,1} = \phi(\mathbf{w}_{1,1}^T \mathbf{x})$$

$$\hat{y}_{1,2} = \phi(\mathbf{w}_{1,2}^T \hat{\mathbf{y}}_1) \qquad \hat{y}_o = \phi(\mathbf{w}_o^T \mathbf{y}_2)$$

Deep Neural Networks

Distinction with MLPs: Activation Functions

- The sign (or step) function of MLPs is mostly useful for classification problems. It also is not as useful for more complicated problems.
- To address these problems, other activation functions are often used in DNNs. These activation functions all use the activation potential as originally defined, as their input
  - **Sigmoid (or logit)**: $\phi(v) = 1/(1 + e^{-v})$
  - **Hyperbolic Tangent (tanh)**: $\phi(v) = 2/(1 + e^{-2v}) - 1$
  - **Rectified Linear (ReLU)**: $\phi(v) = \max(0, v)$
  - **Linear**: $\phi(v) = v$
- Different activation functions may be used in different layers (not required).



DNN Weight Learning

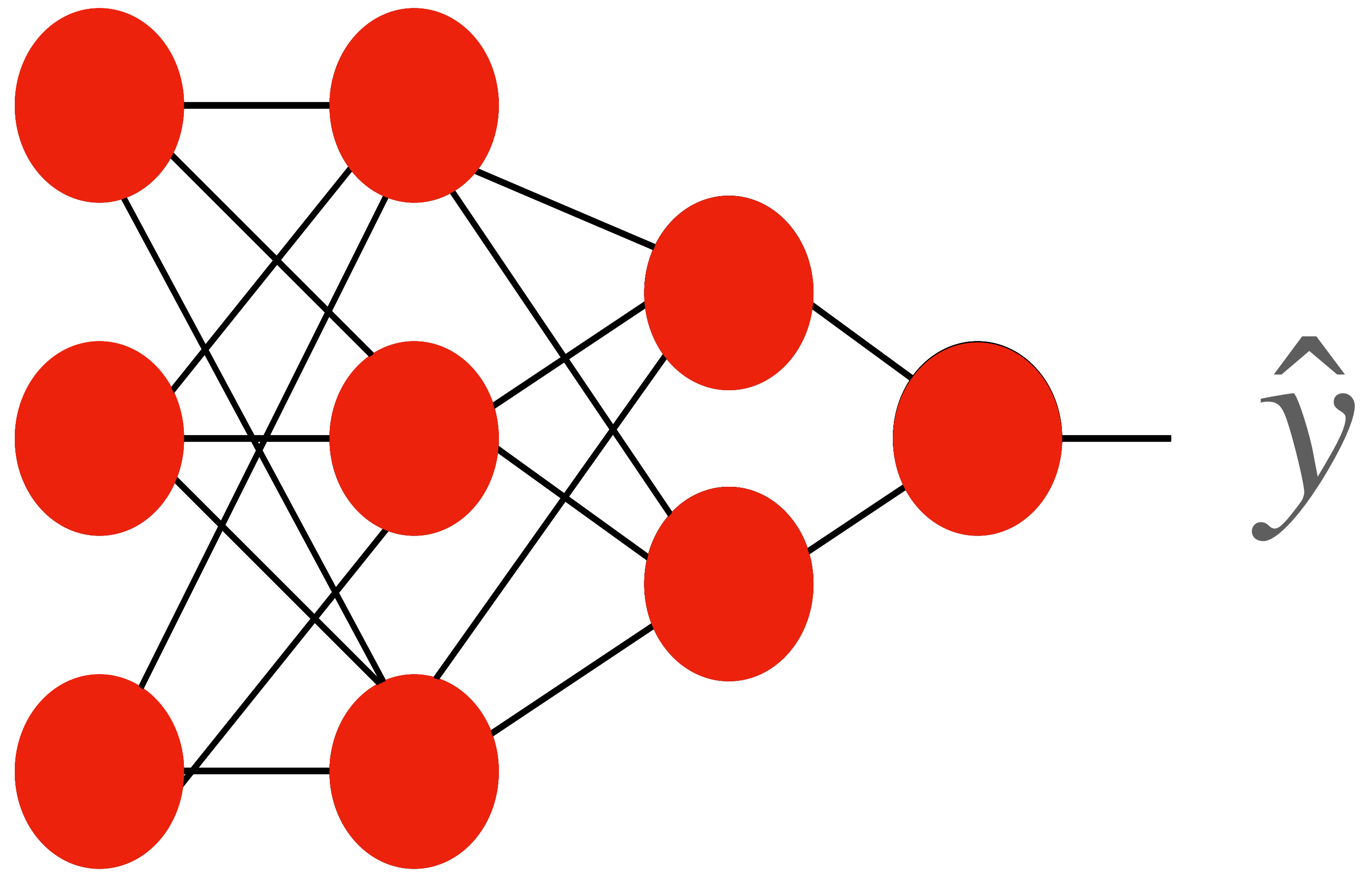
DNNs are supervised learning algorithms

- **Goal**: Find the weights that minimize the error (often MSE) between true and predicted values using a training set.
- During the training phase, the ***back propagation*** algorithm is used to find these weights
- **General idea of back propagation**:
    - For each training sample (e.g. input), compute the output using the forward pass
    - Compute the estimation/prediction error
    - Sequentially go through each layer (in reverse order) to measure the error contribution from each connection
    - Update the weights based on the error contribution to reduce the error (e.g. gradient descent)

DNN Weight Learning

Graphical depiction of backpropagation

- <u>**General idea of back propagation**</u>:
  - For each training sample (e.g. input), compute the output using the forward pass
  - Compute the estimation/prediction error
  - Sequentially go through each layer (in reverse order) to measure the error contribution from each connection
  - Update the weights based on the error contribution to reduce the error (e.g. gradient descent)

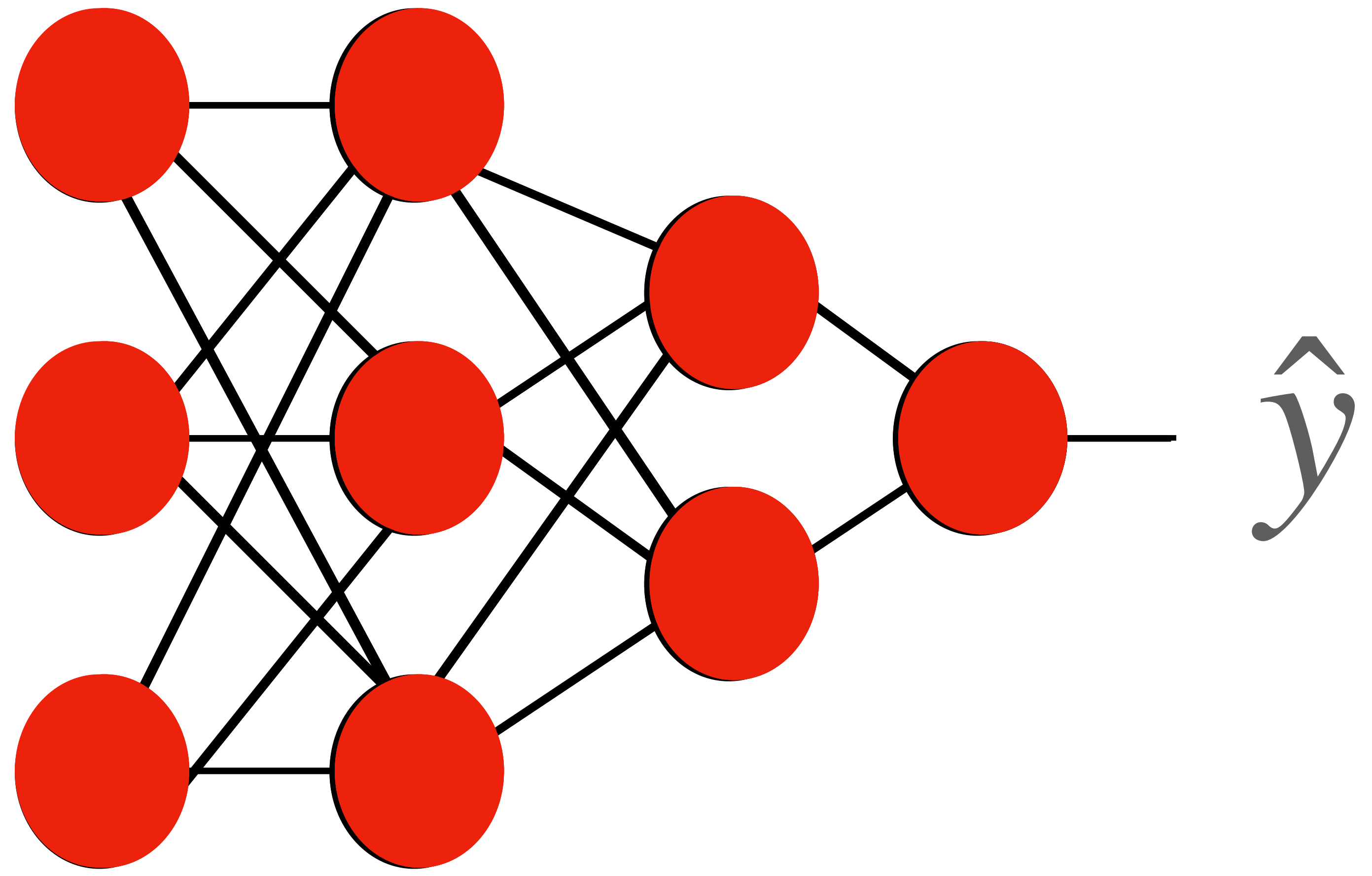


Compute Forward Pass of DNN to get output from input

DNN Weight Learning

Graphical depiction of backpropagation

- **<u>General idea of back propagation</u>**:
    - For each training sample (e.g. input), compute the output using the forward pass
    - Compute the estimation/prediction error
    - Sequentially go through each layer (in reverse order) to measure the error contribution from each connection
    - Update the weights based on the error contribution to reduce the error (e.g. gradient descent)



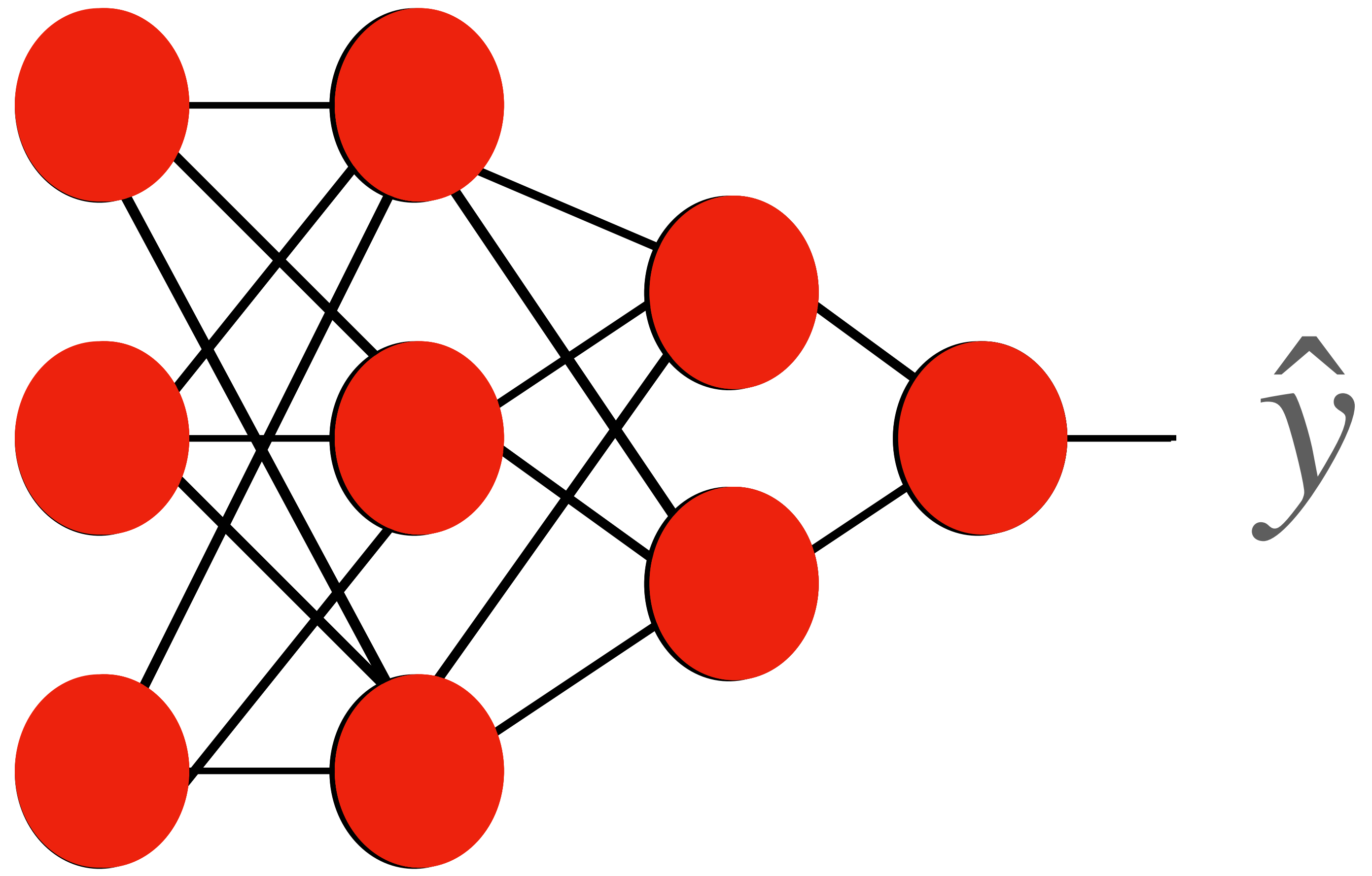
Compute output error

$$y - \hat{y}$$

DNN Weight Learning

Graphical depiction of backpropagation

- <u>**General idea of back propagation**</u>:
  - For each training sample (e.g. input), compute the output using the forward pass
  - Compute the estimation/prediction error
  - Sequentially go through each layer (in reverse order) to measure the error contribution from each connection
  - Update the weights based on the error contribution to reduce the error (e.g. gradient descent)



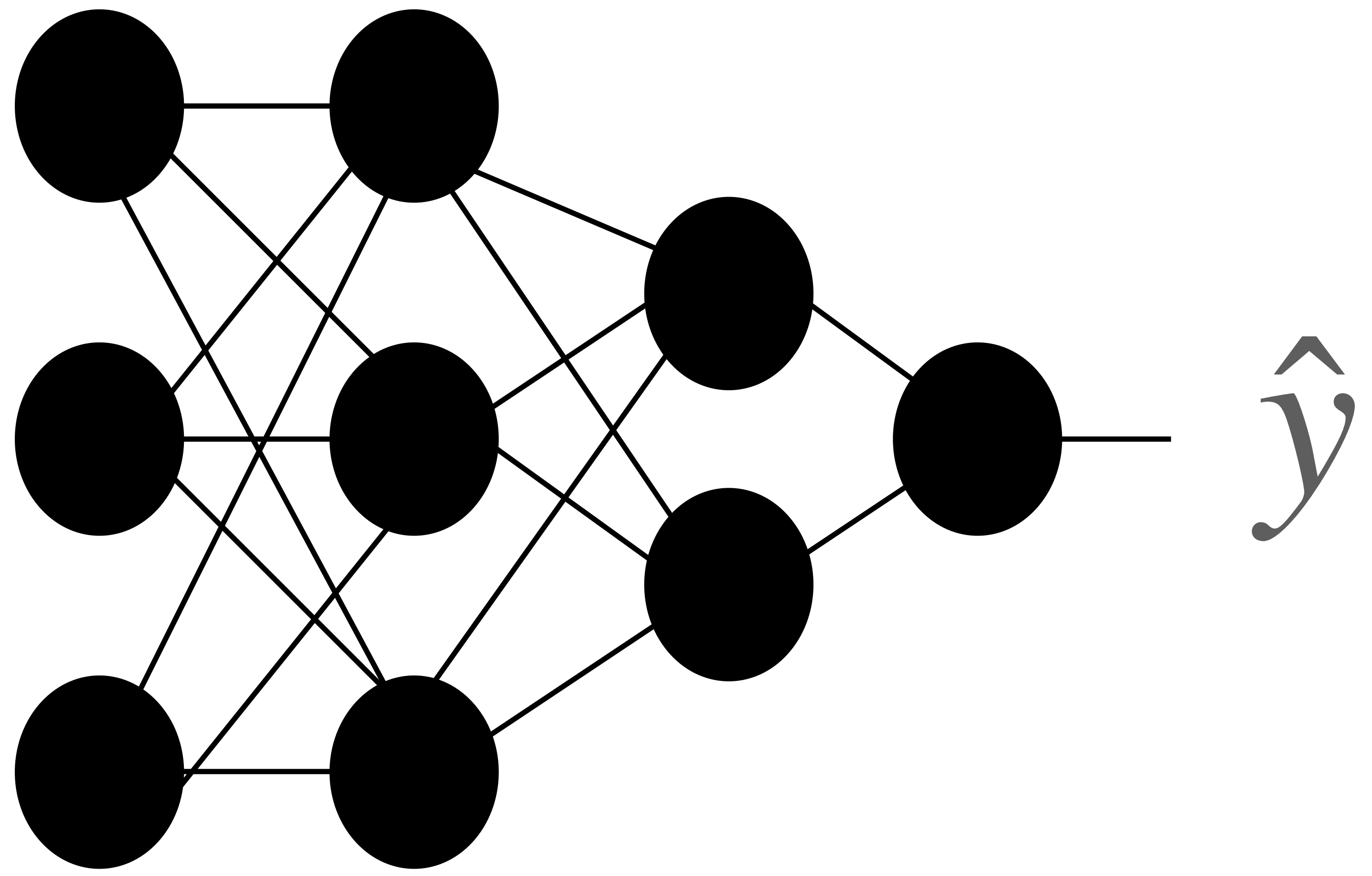
Compute contribution from each neuron in each layer

δ_k

DNN Weight Learning

Graphical depiction of backpropagation

- **<u>General idea of back propagation</u>**:
  - For each training sample (e.g. input), compute the output using the forward pass
  - Compute the estimation/prediction error
  - Sequentially go through each layer (in reverse order) to measure the error contribution from each connection
  - Update the weights based on the error contribution to reduce the error (e.g. gradient descent)

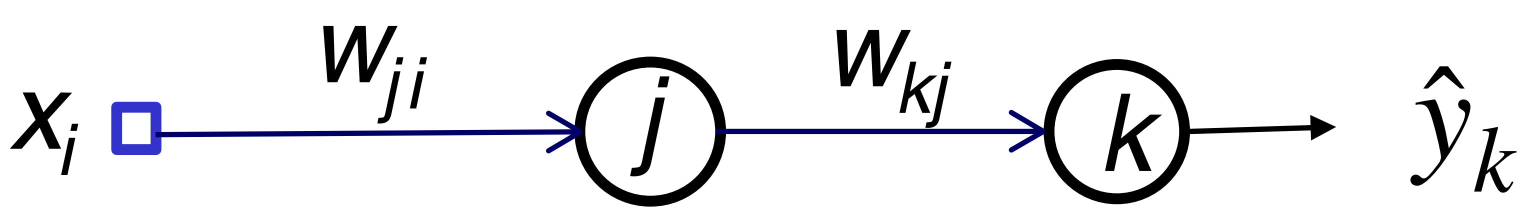


Update weights based on error and (weighted) contributions for each neuron and layer

$$\mathbf{w}_k = \mathbf{w}_k - \eta \nabla E(w)$$

Mathematics behind Backpropagation

Simplified DNN

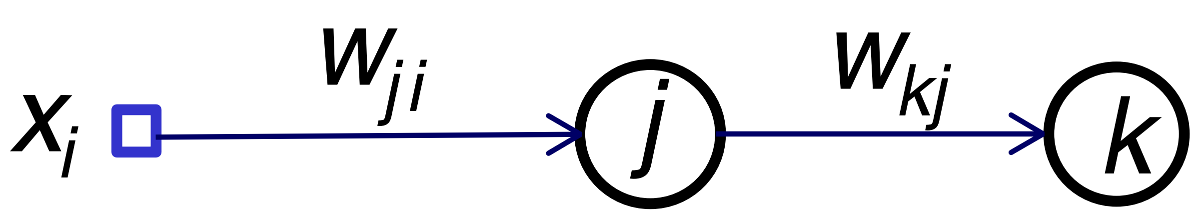


- Notation for one hidden layer
  - Weight(s) for the k-th neuron in the output layer are denoted as $\mathbf{w}_{kj}$
  - Weight(s) for the j-th neuron in the hidden layer are denoted as $\mathbf{w}_{ji}$
- **Iterative weight update equations based on MSE loss function each iteration performs both in an interlaced manner** (e.g. output update, hidden layer update, output update, hidden layer update, out…)
  - **Output layer (delta rule)**: $\mathbf{w}_{kj}(n+1) = \mathbf{w}_{kj}(n) + \eta e_k(n)\phi'(v_k)\hat{y}_j(n) = \mathbf{w}_{kj}(n) + \eta\delta_k(n)\hat{y}_j(n)$
  - ***Hidden layer (generalized delta rule)***:

$$\mathbf{w}_{ji}(n+1) = \mathbf{w}_{ji}(n) + \eta\phi'(v_j(n))\left[\sum_k \delta_k(n) w_{kj}(n)\right]\mathbf{x}_i(n) = \mathbf{w}_{ji}(n) + \eta\delta_j(n)\mathbf{x}_i(n)$$

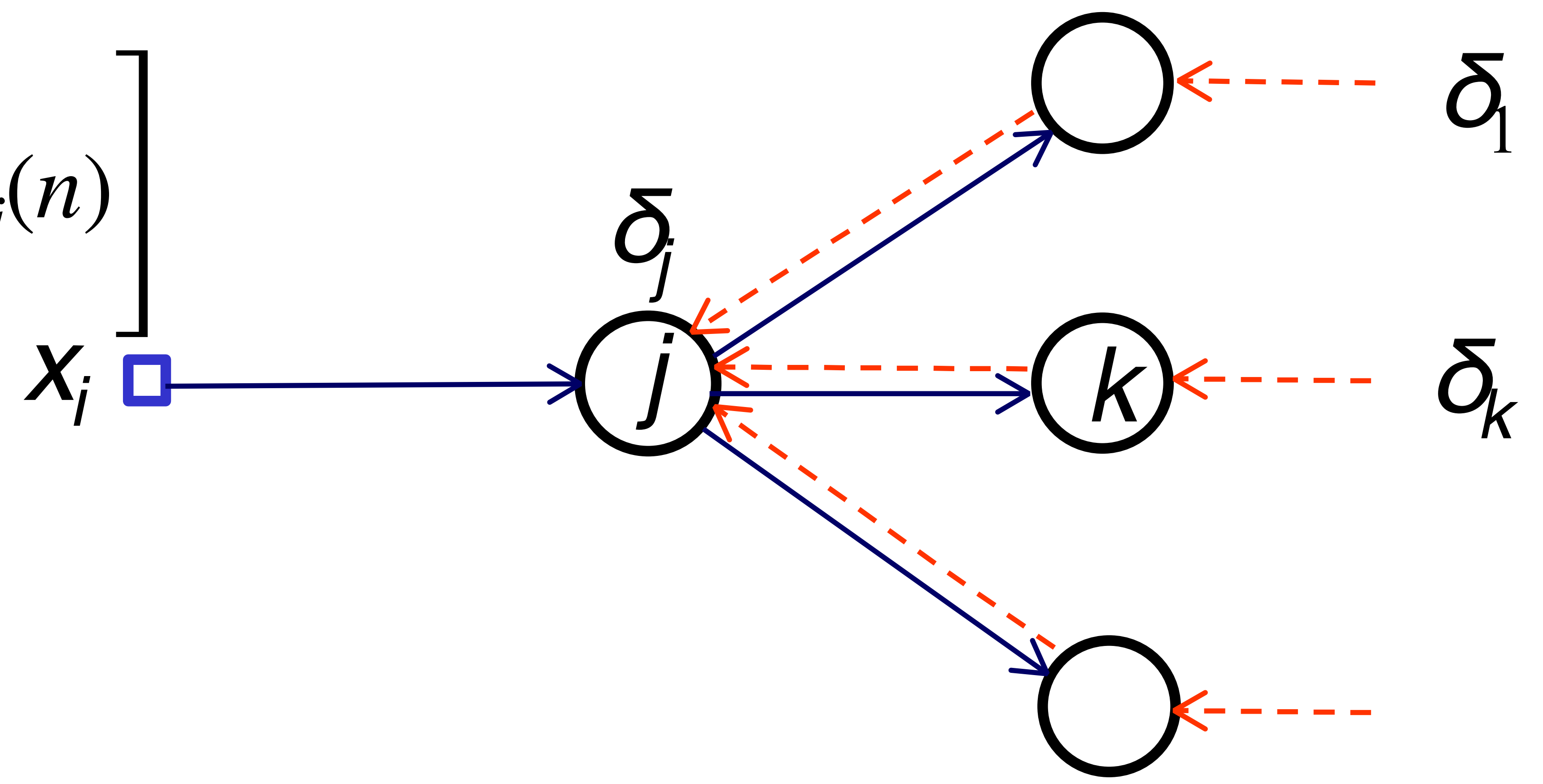
Backpropagation

Illustration of delta rules



- Illustration of the generalized δ rule,

$$\delta_j(n) = \phi'(v_j(n))\left[\sum_k \delta_k(n) w_{kj}(n)\right]$$



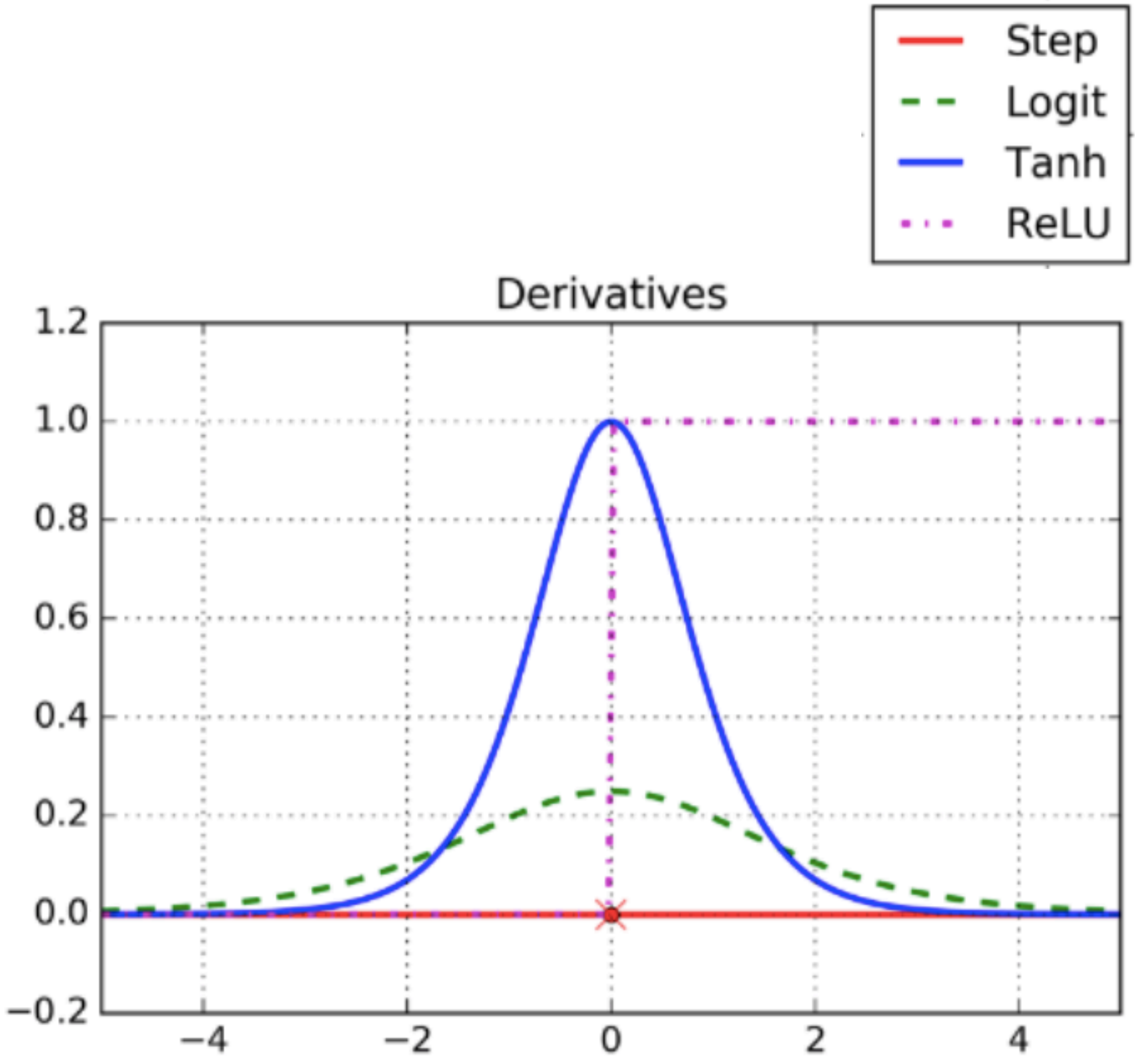
$$\delta_k(n) = (y_k - \hat{y}_k)\phi'(v_k) = e_k(n)\phi'(v_k)$$

- The ***generalized*** δ ***rule*** gives a solution to the credit (blame) assignment problem (e.g. which neuron is most responsible for the error).
- The updates depend on the partial derivative of the activation function $\phi'(v)$
- Updates can be done using Stochastic or Mini-batch gradient descent

Derivatives of activation functions

Needed for weight update

- Plot of partial derivatives for different activation functions



- For a logistic sigmoid for ϕ

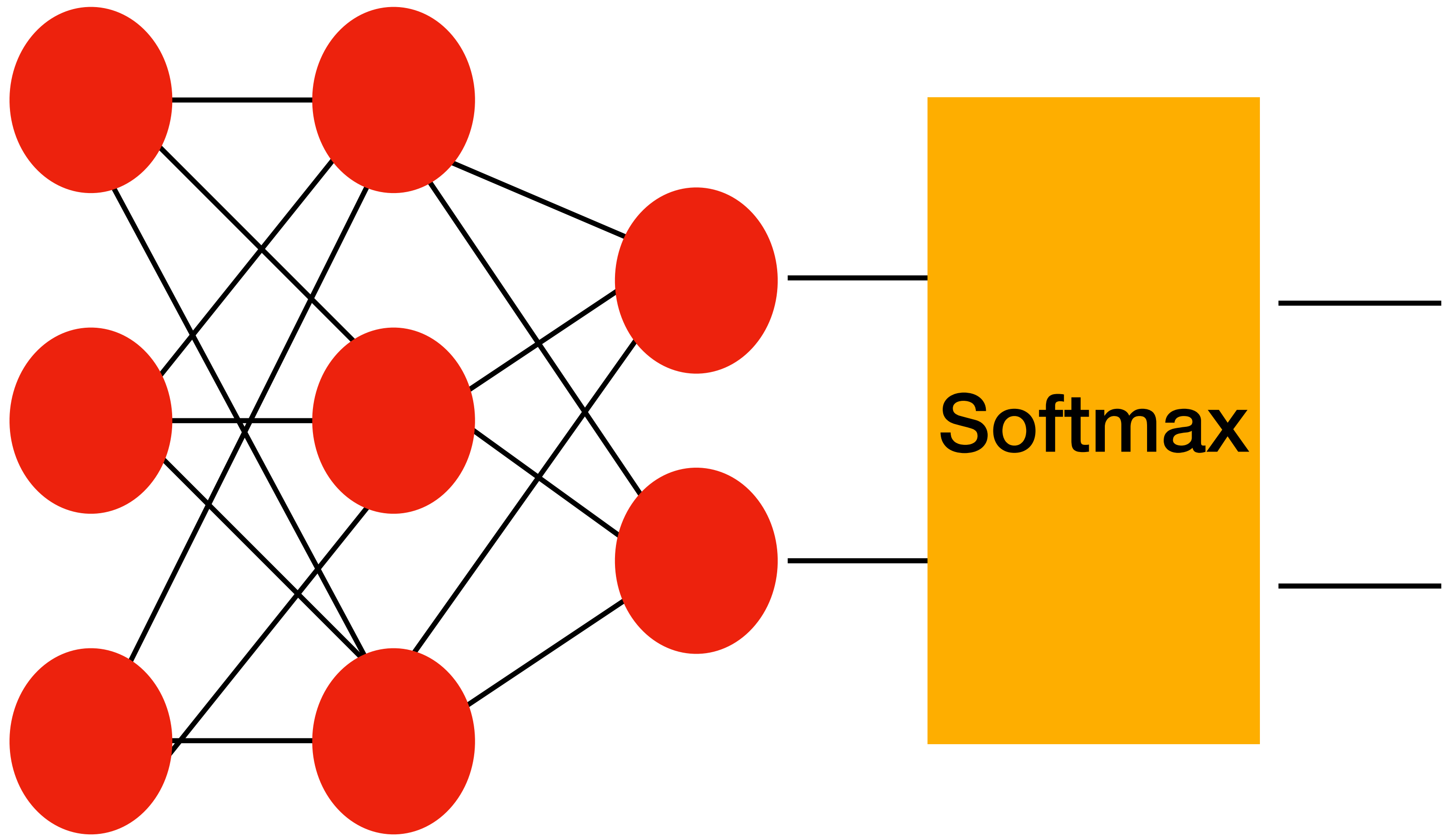
$$\phi(v) = \frac{1}{1 + e^{-v}} \quad => \quad \phi'(v) = \phi(v)[1 - \phi(v)]$$

DNNs as Multi-Class Classifiers

Softmax layer

- The prior approach assumes that the DNN is being trained to solve a regression problem. What happens if we need to solve a classification problem?
- For binary classification, we can merely threshold the output neuron, as was shown previously for Linear Regression
- For multi-class classification (when the classes are exclusive), each neuron in the output layer corresponds to a single class.
  - The output layer is then modified to replace the individual activation functions with a shared ***soft-max function***
  - Outputs represent estimated probability of the corresponding class

$$\hat{y}_i = p_{y_i} = \phi_k(v) = \frac{e^{v_k}}{\sum_{j=1}^{K} e^{v_j}}$$



Cross Entropy Loss to train network

$$L_{ce} = -\sum_{i=1}^{m} y_i \log(p_{y_i})$$

Next Class: Neural Networks III

Training a DNN

A Python Example using TensorFlow

- MNIST Digit Recognition Example
- Other considerations during training