Multi-layer Neural Networks

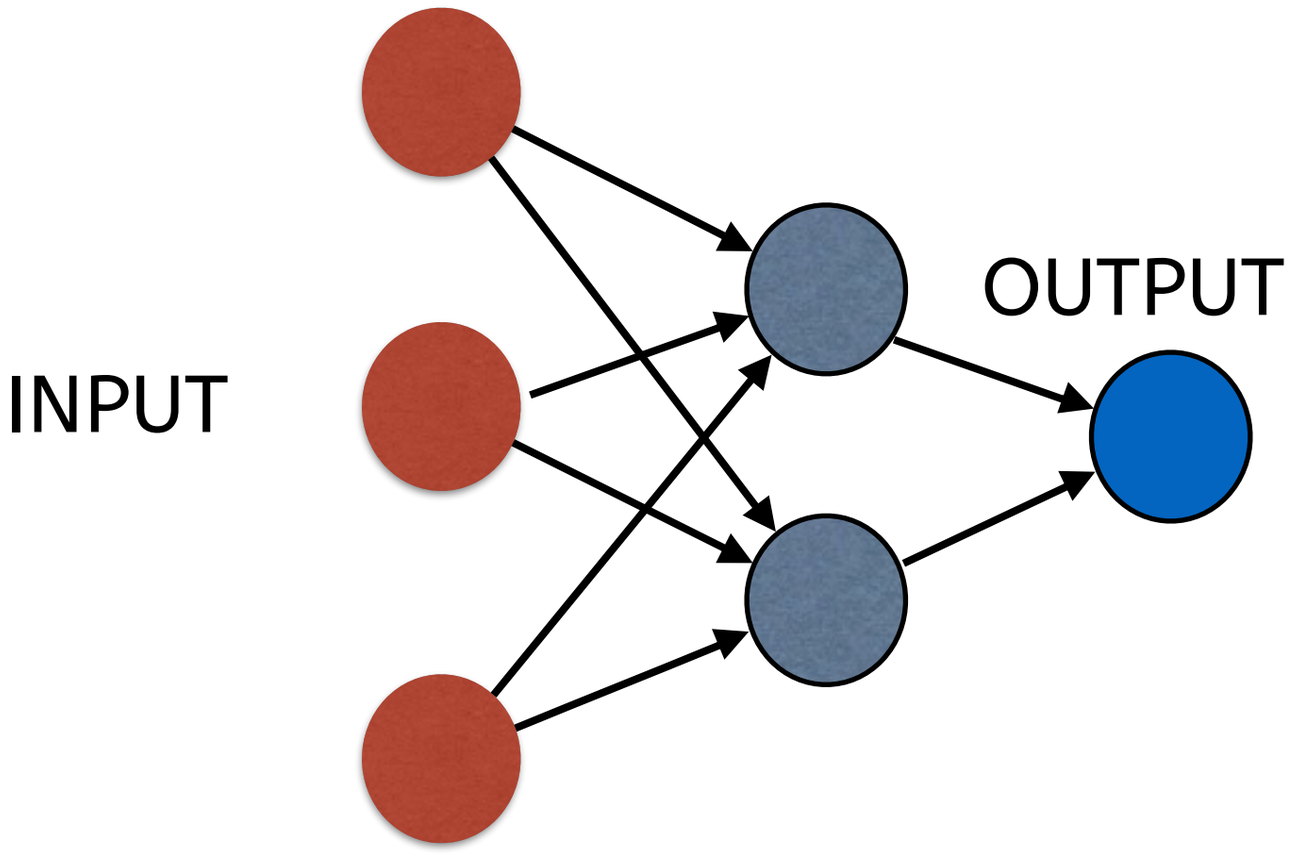
COMP 527
Danushka Bollegala

Neural Networks

- We have already learnt single layer neural networks
  - It was called the Perceptron
- Unfortunately, it could learn only linear relationships
  - Quiz: What is meant by *linear separability*?
- Can we stack multiple layers of neurons to learn more complex (from non-linearly separable data) functions?

Artificial Neural Networks

- There are different types of artificial neural networks
  - **Feed forward** neural networks
    - no backward links, only forward links



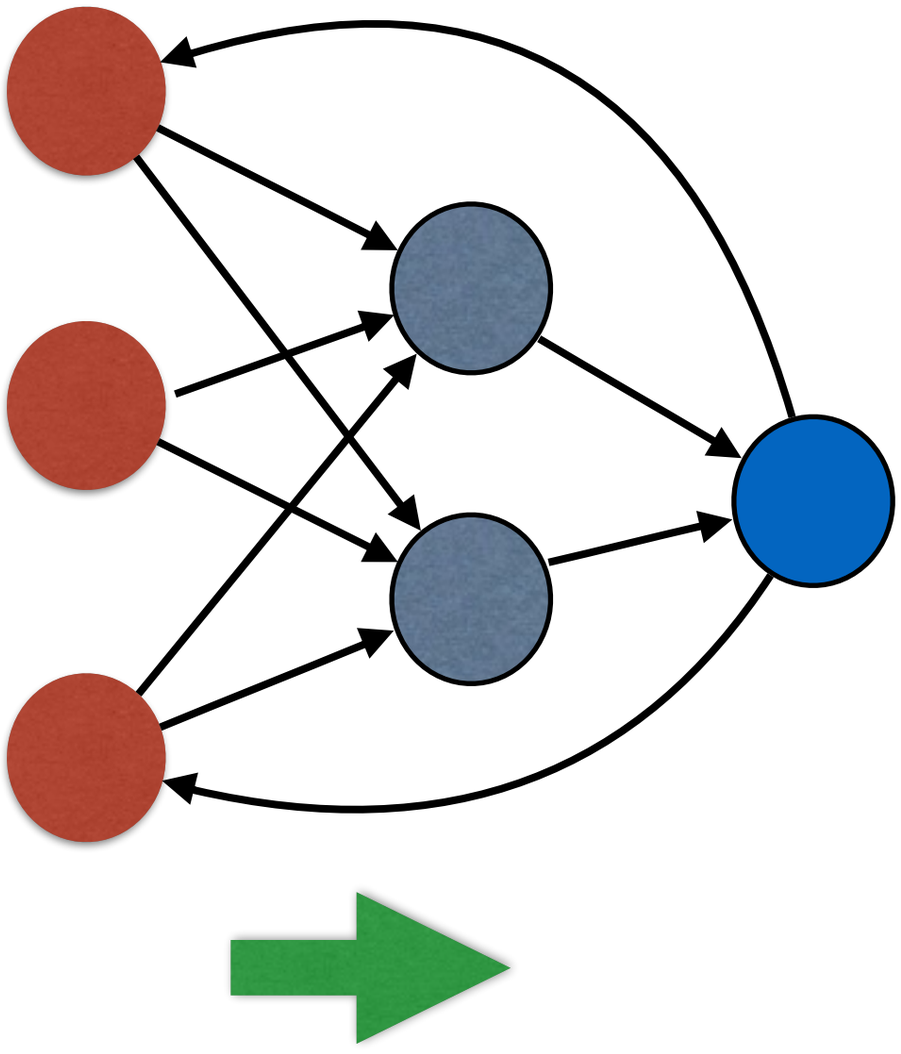
The most popular case.
Useful for learning classifiers/regression models.

Typically 3 layers (one hidden).

Recently there have been work that learn 7 or more hidden layers!

Artificial Neural Networks

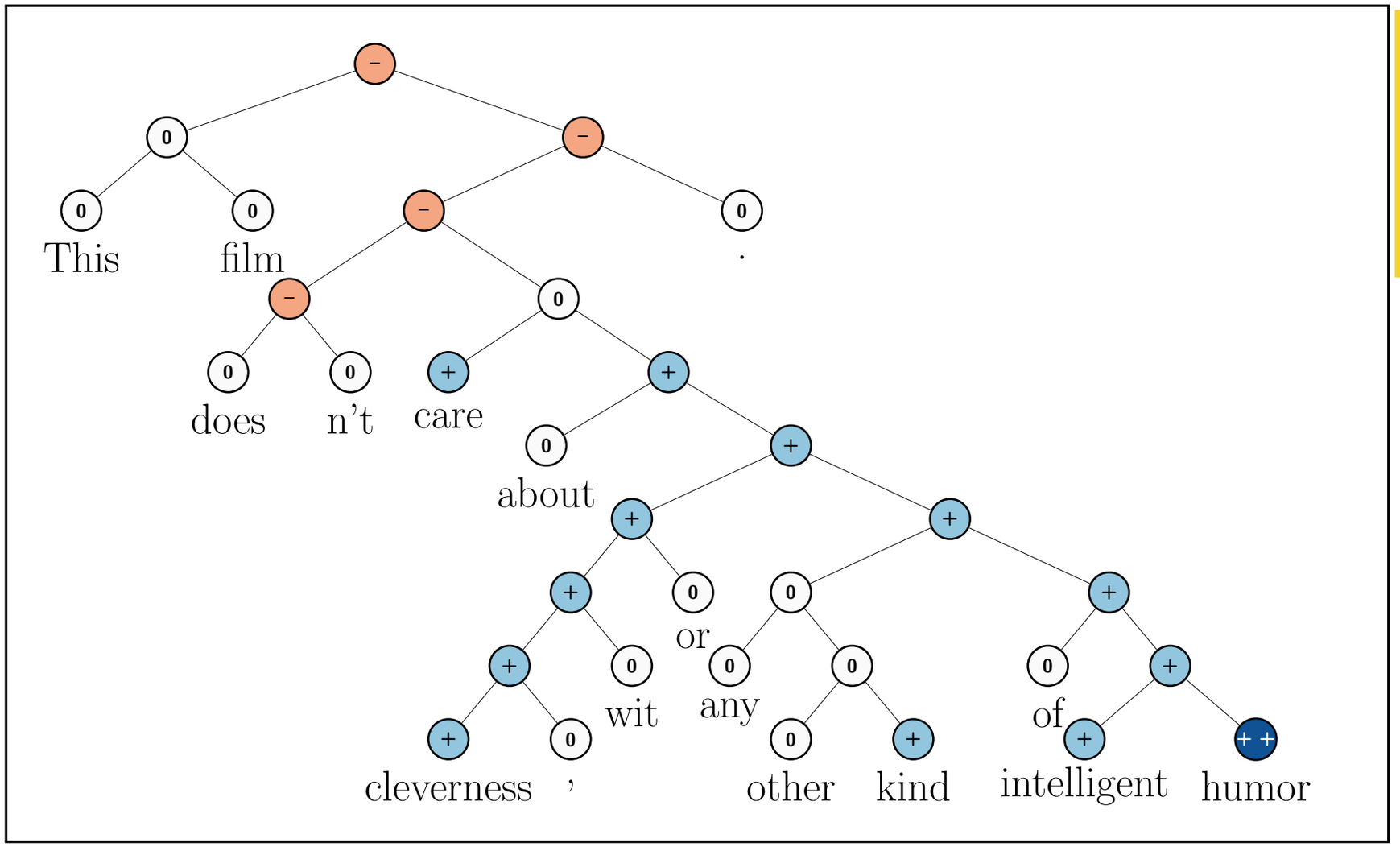
- There are different types of artificial neural networks
  - **Recurrent** neural networks
    - both backward and forward links exist



Useful for learning temporal relations.

Artificial Neural Networks

- There are different types of artificial neural networks
  - **Recursive** neural networks
    - Same neural network is applied recursively, following some structure

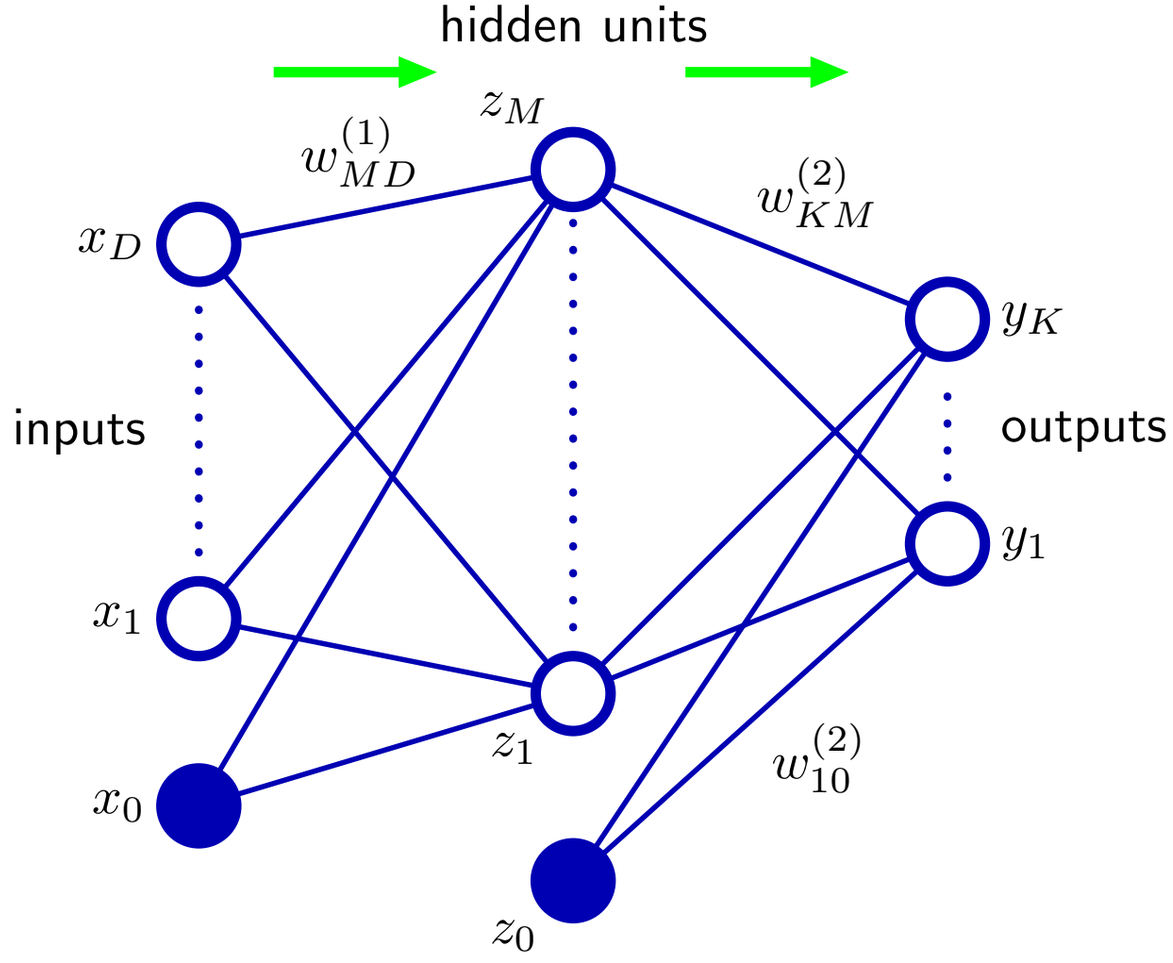


Used in NLP for dependency parsing, sentiment classification, etc.

[Socher+ 13]

Can we learn multiple classes?

- With 1 logistic output node we can only predict two classes (binary classification)
- With 2 such output nodes we can predict 4 classes [(0,0), (0,1), (1,0), (1,1)]
- With k such output nodes we can predict 2^k classes!

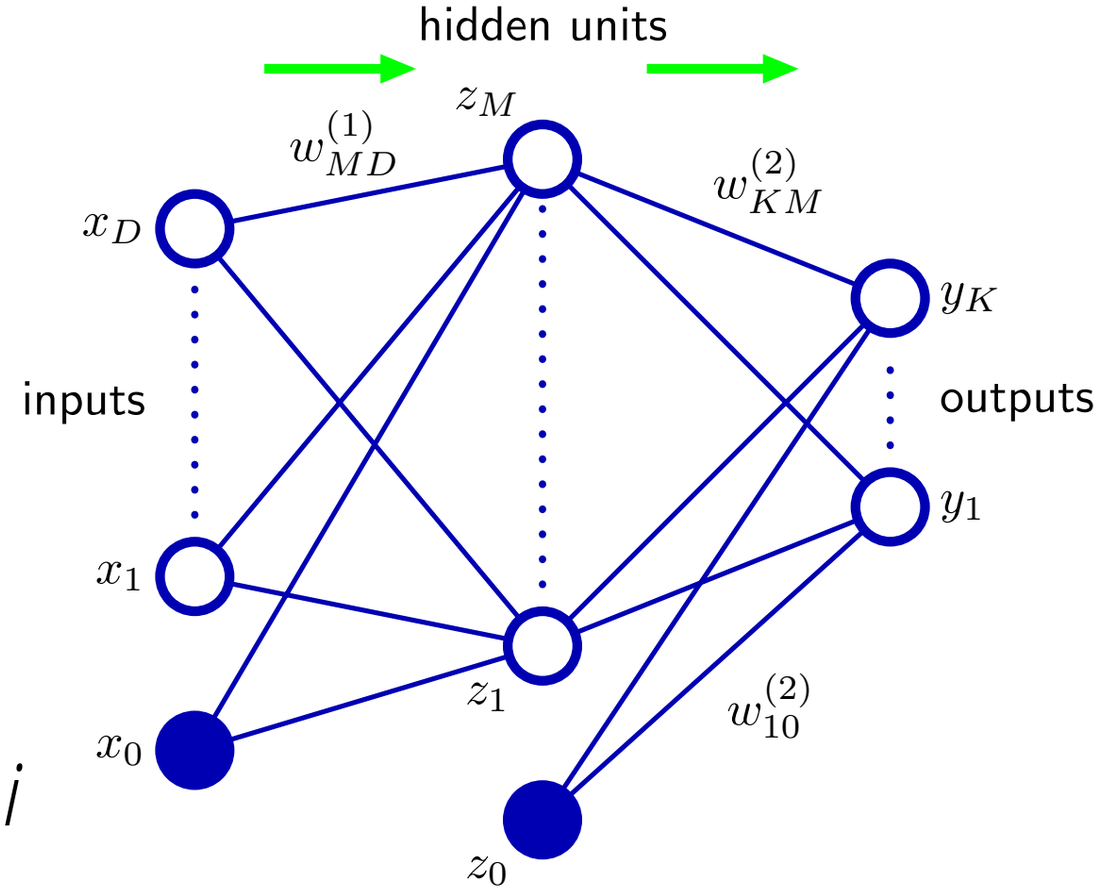


How can we learn NNs?

- We only have supervision for the output layer.
- How can we infer the weights for the edges (connections) in the inner (hidden) layers?
  - error backpropagation (aka *backprop*)

Derivation of Backprop

- Let us consider the following feed forward neural network.
  - D input nodes (features)
  - M hidden nodes
  - bias terms x_0 and z_0
  - k output nodes
- The weight connecting node *i* to node *j* is w_{ji}
- Note the reverse notation!



Derivation of Backprop

- Let us assume that the prediction error of the *n*-th training instance is measured by the *squared loss function*.

$$E_n = \frac{1}{2}\sum_k (y_{nk} - t_{nk})^2$$

- The derivative of loss w.r.t. a particular weight is

$$\frac{\partial E_n}{\partial y_{nk}} = y_{nk} - t_{nk}$$

- Activation z_j at node j is given by

$$a_j = \sum_i w_{ji} z_i$$

- Using some activation function h

$$z_j = h(a_j)$$

Derivation of Backprop

- We would like to update the weights of the neural network such that the loss is minimized.
- If we can somehow compute the derivative of the loss w.r.t. to a particular weight then we can use stochastic gradient descent for this purpose (recall logistic regression).

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}.$$

Recall Calculus — Chain Rule

- If y is a function of x (y=f(x)), and x is a function z (x = g(z)), then we can write

$$\frac{dy}{dx} = \frac{dy}{dz}\frac{dz}{dx}$$

You can remember this as the two dz *factors* canceling each other out. But remember these are not factors but differentials.

Derivation of Backprop

- A new notation

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j}$$

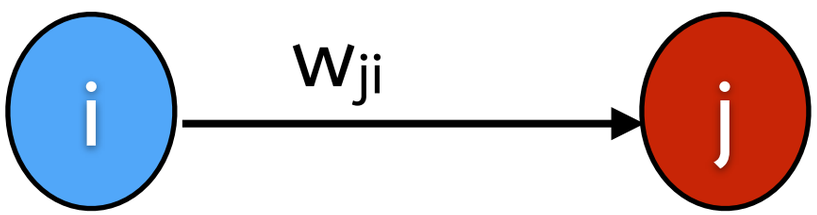
delta can be thought as the *error* associated with node j.

How much overall error does it cause when we vary the input arriving at node j?

Derivation of Backprop

$$a_j = \sum_i w_{ji} z_i$$

$$\frac{\partial a_j}{\partial w_{ji}} = z_i.$$



$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i.$$

error at output δ_j

activation at input z_i

Derivation of Backprop

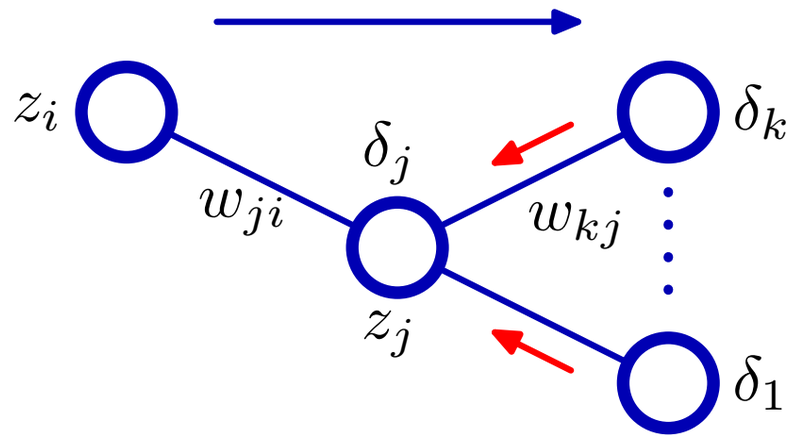
- We know all activations z for all nodes by forward propagation.
- We know the error δ for the output nodes. For squared error this was,

$$\delta_k = y_k - t_k$$

- If we can somehow compute the δs for the inner (hidden) nodes, then we are done.
  - backpropagation = recursive updating

Derivation of Backprop

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$



Q: why do we sum over k?

A: a_j depends on a_k as follows

$$a_k = \sum_j w_{kj} z_j = \sum_j w_{kj} h(a_j)$$

From this relationship we have,

$$\frac{\partial a_k}{\partial a_j} = w_{kj} h'(a_j)$$

By the definition of δ, $\frac{\partial E_n}{\partial a_k} = \delta_k$

Putting it altogether

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

Error at an inner node j

Derivative of the activation function at node j

The weight connecting node j to k

Error at an outer node k

Backprop Algorithm

1. Apply the input vector $\mathbf{x}_n$ to the network and forward propagate the network using the following equations to find the activations of all the hidden and output nodes.

$$a_j = \sum_i w_{ji} z_i \qquad\qquad z_j = h(a_j).$$

2. Evaluate δ_k for all the output nodes using

$$\delta_k = y_k - t_k$$

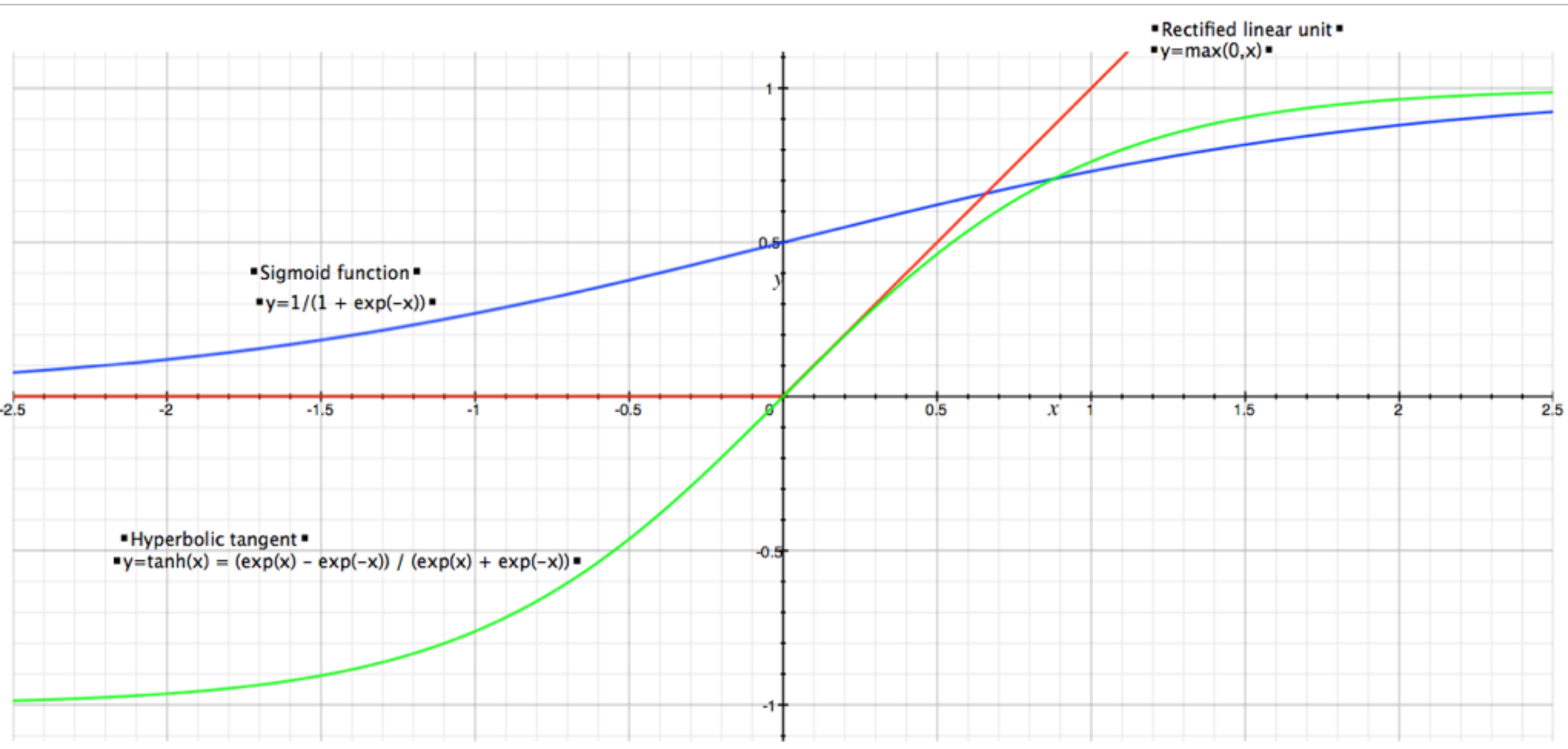
3. Backpropagate the δ's using the following equation to obtain δ_j for each hidden node in the network.

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k$$

4. Use SGD to update weights

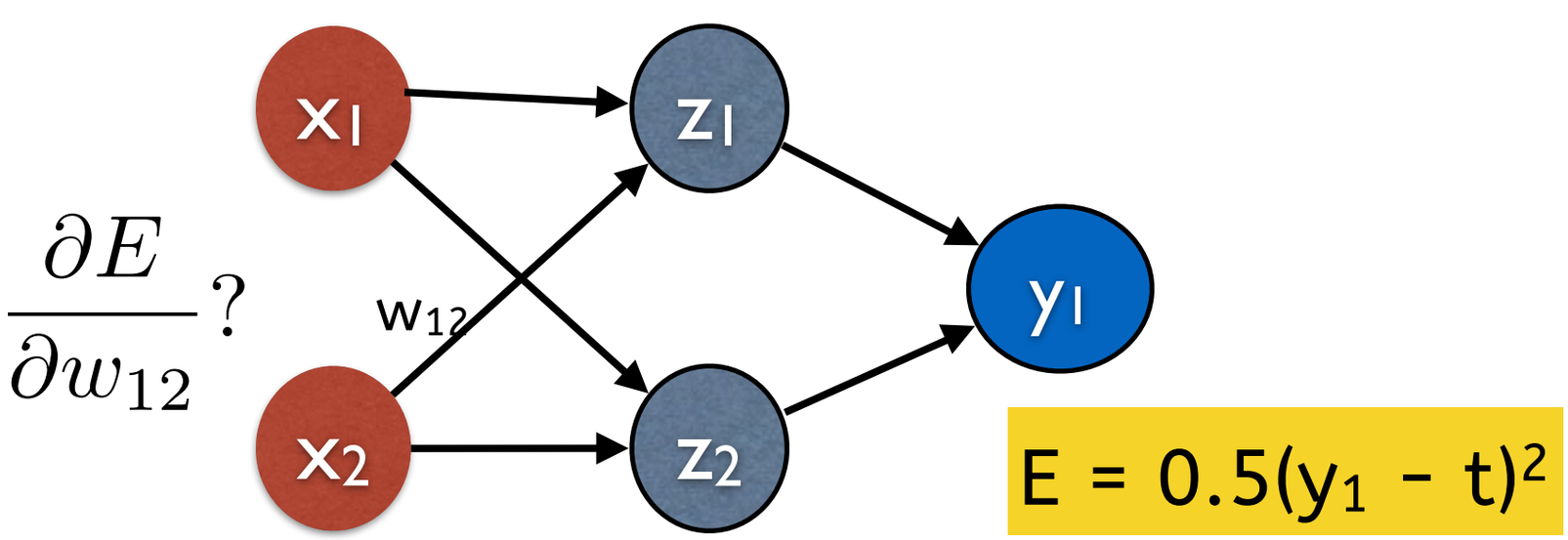
$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \qquad\qquad w_{ji}^{(t+1)} = w_{ji}^{(t)} - \eta \frac{\partial E_n}{\partial w_{ji}}$$

Activation Functions



Quiz

- Consider the following neural network with two input nodes, two hidden nodes, and a single output node. Assuming that we are using the squared loss function and the activation function at all nodes is tanh, compute the gradient of the error w.r.t. w_{12}



References

- Pattern Recognition and Machine Learning (PRML) Sec 5.3.