Deep Learning

Paul Rad, Ph.D.

Associate Professor
Cyber Analytics and AI
Information Systems and Cyber Security
College of Business School
210.872.7259

Outline

1. History Neural Network
2. Single Layer Neural Networks
3. Deep Learning Model
4. Learning Algorithm - Gradient Descent
5. Dropout
6. Regularization

Neural Network History

- Algorithms that try to mimic the brain
- It was very widely used in 80s and early 90s.
- Recently very popular again due to State-of-the-art technique for many applications.
- Human brain: Massively parallel network
- The basic computational unit of the brain is a **neuron**. Approximately **86 billion neurons** can be found in the human nervous system and they are connected with approximately 10^14 - 10^15 synapses

What is deep learning useful for?

Like all machine learning: making predictions

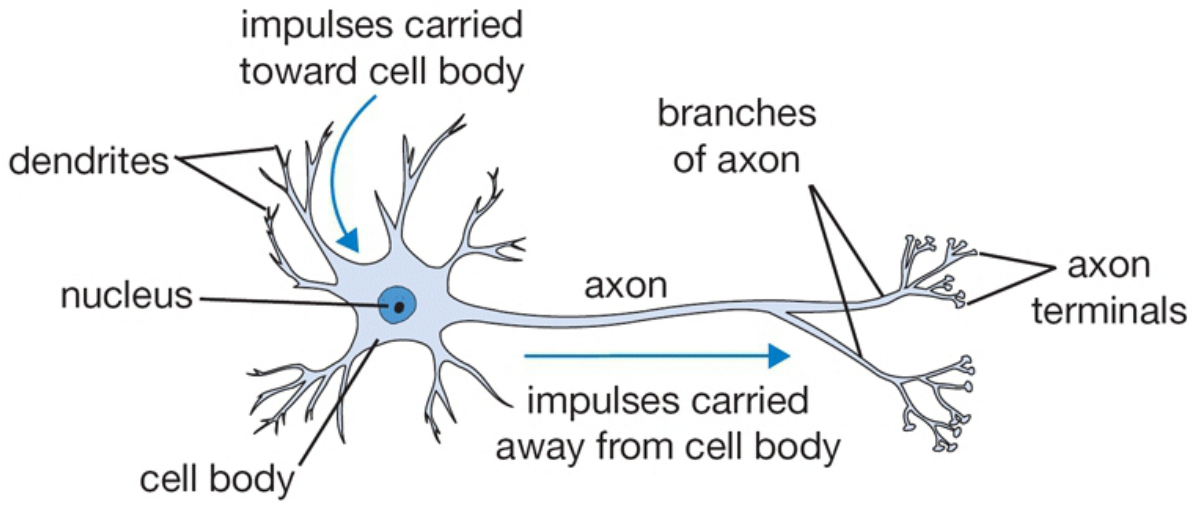
Deep learning does it with higher accuracy for some problems

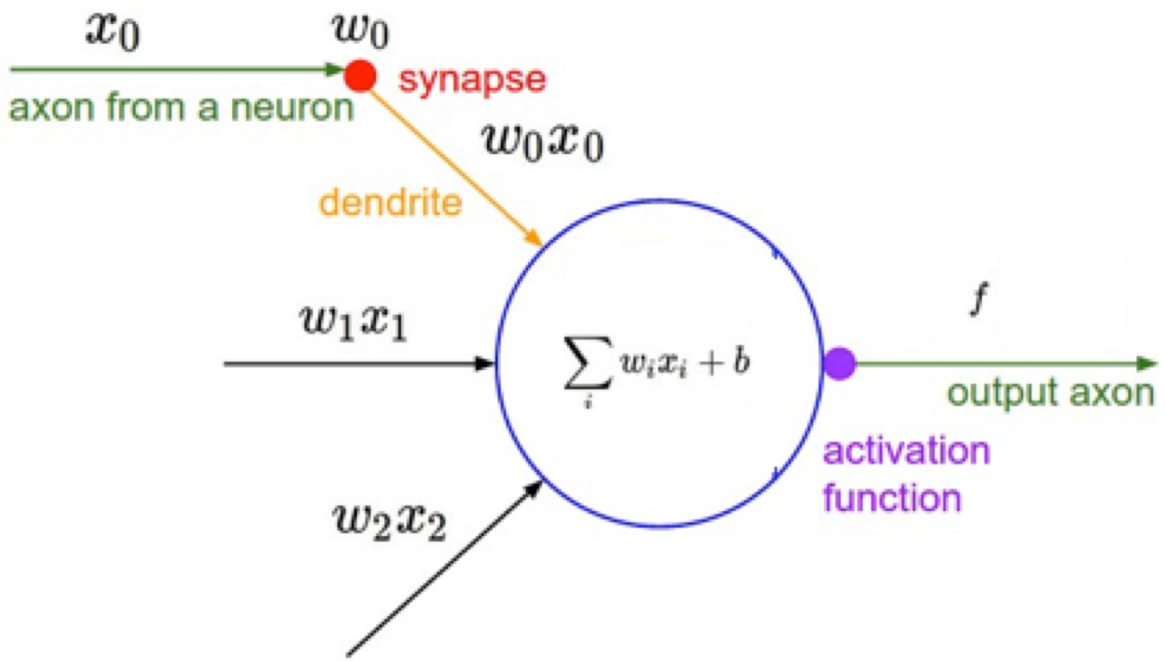
For example?

- Self-driving cars
- Stock market prediction
- Recognizing faces in an image

Under active development and research

Biological neuron and mathematical model

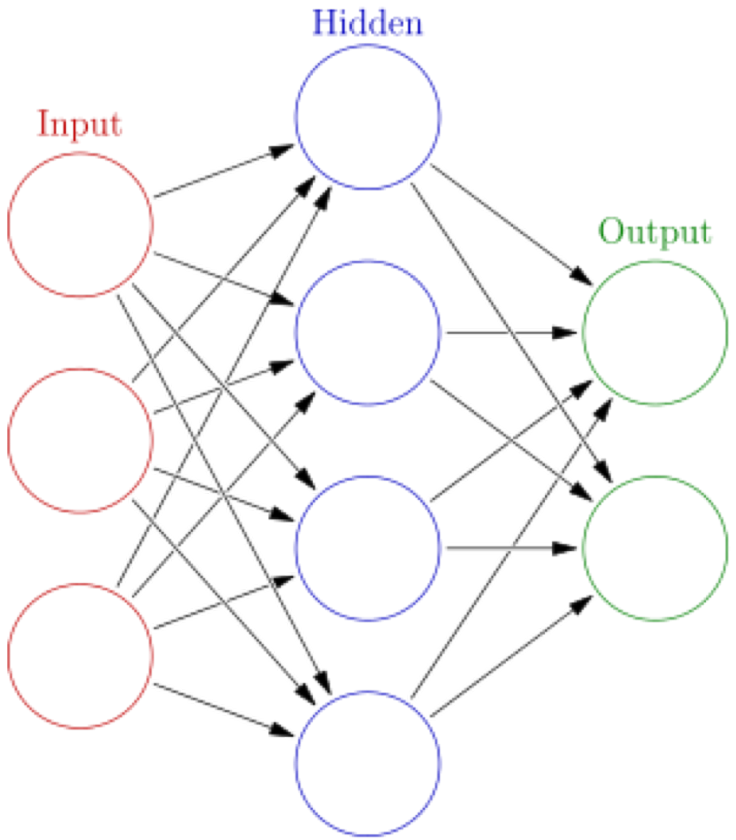




What Is A Neural Network?

The simplest definition of a neural network, more properly referred to as an 'artificial' neural network (ANN), is provided by the inventor of one of the first neurocomputers, Dr. Robert Hecht-Nielsen. He defines a neural network as:

"...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs.



Every node in one layer is connected to every node in the next layer.

Signals get transmitted from the input, to the hidden layer, to the output

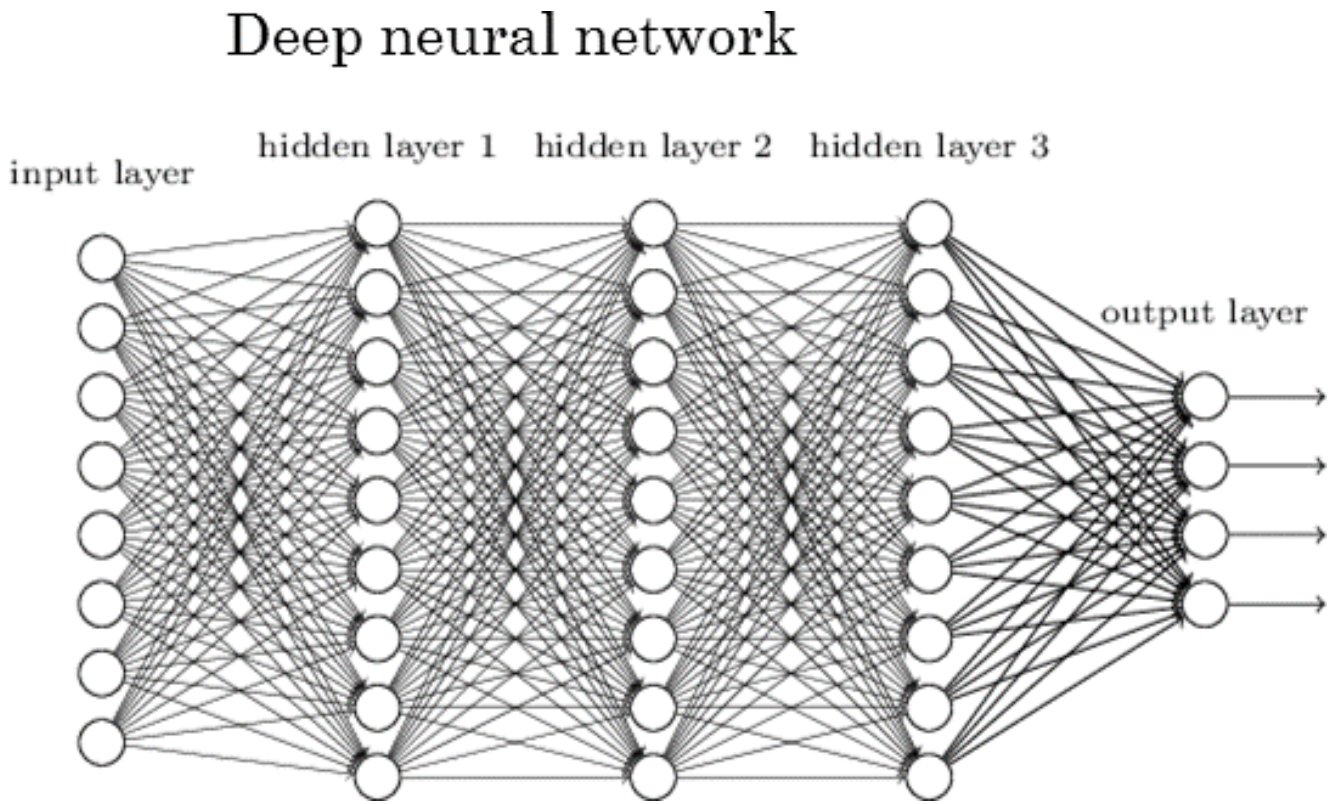
Deep Learning Architecture

An Artificial Neural Network with one or more hidden layers = Deep Learning

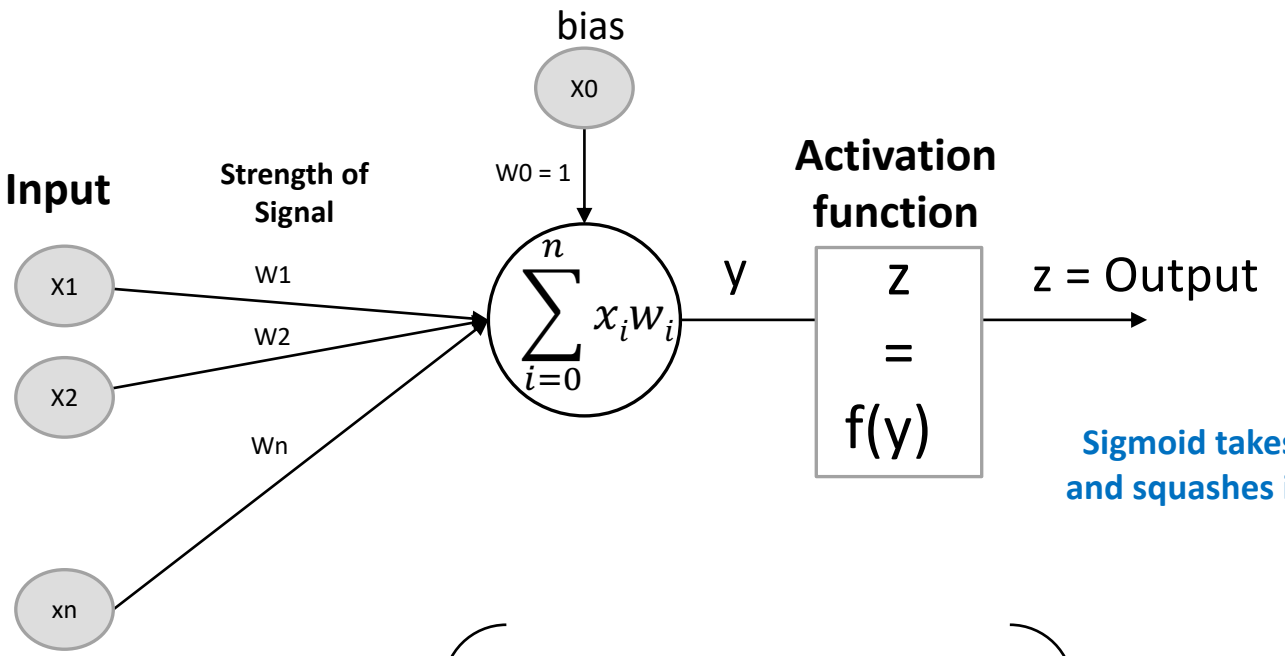
They typically consist of many hundreds of simple processing units which are wired together in a complex communication network.

Each unit or node is a simplified model of a real neuron which fires (sends off a new signal) if it receives a sufficiently strong input signal from the other nodes to which it is connected.

The output is aiming for a target.



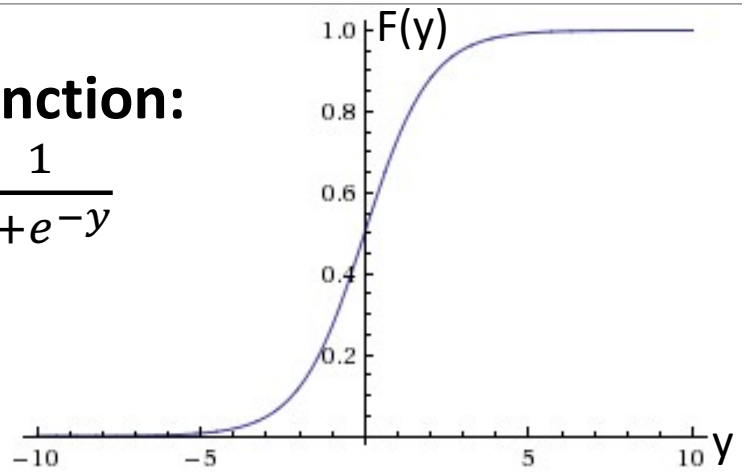
Single Layer Neural Networks with Nonlinear Math Model



Sigmoid function:

$$f(y) = \frac{1}{1+e^{-y}}$$

(0 , 1)



Sigmoid takes a real-valued input and squashes it to range between 0 and 1

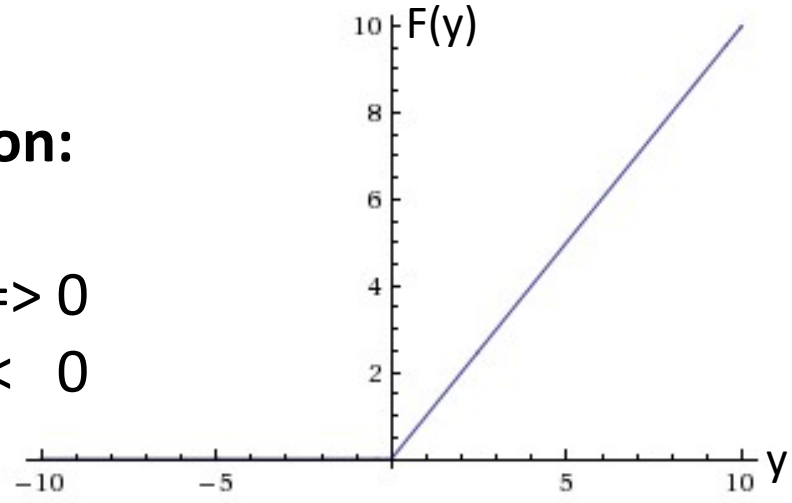
Output = f ([x0 x1 x2 … xn] x [1, w1, w2, …, wn])

ReLU function:

f(y) =

y y => 0

0 y < 0

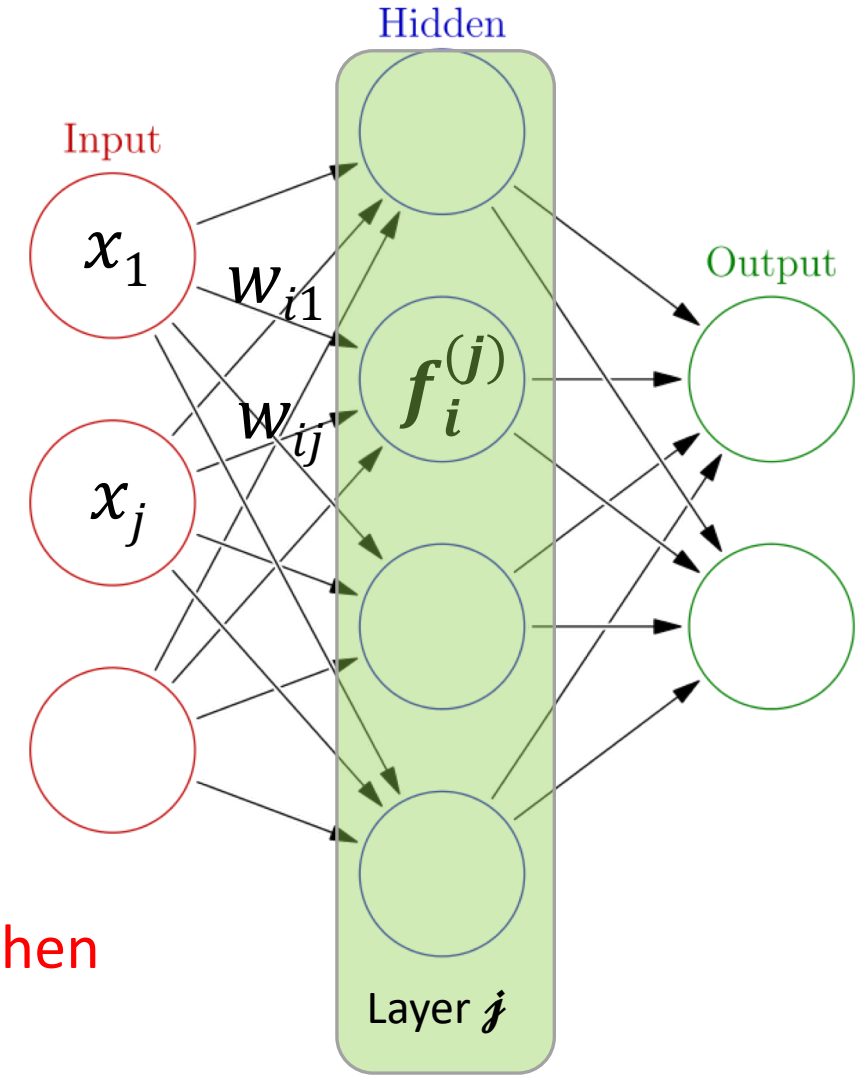


Multi Layer NN Nonlinear Math Model

$f_i^{(j)}$ = "activation function" of unit i in layer j

W $^{(j)}$ = matrix of weights controlling function mapping from layer j to layer j+1

$$y_i = fi(\sum_{j=0}^{n} w_{ij}\, x_j\,)$$



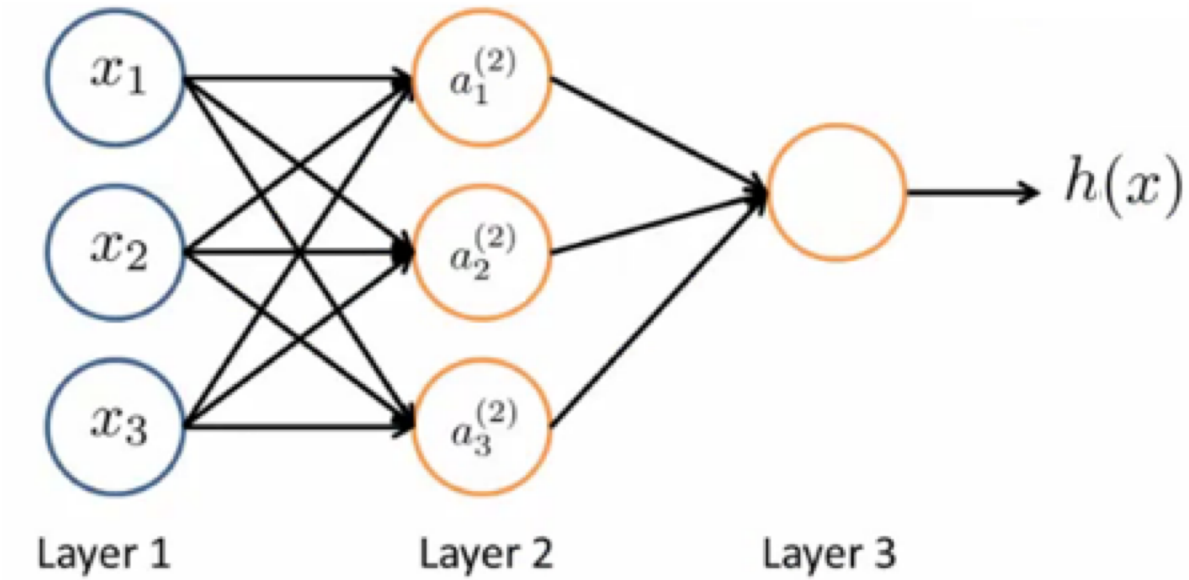
If network has N units in layer j, and M units in layer (j-1), then W $^{(j)}$ will be of dimension of (M+1) x N

Forward Propagation to Calculate h(x)

$$a_1^{(2)} = f\left(\sum_{j=0}^{n} w_{1j}\, x_j\right) =$$

$$a_2^{(2)} = f\left(\sum_{j=0}^{n} w_{2j}\, x_j\right) =$$

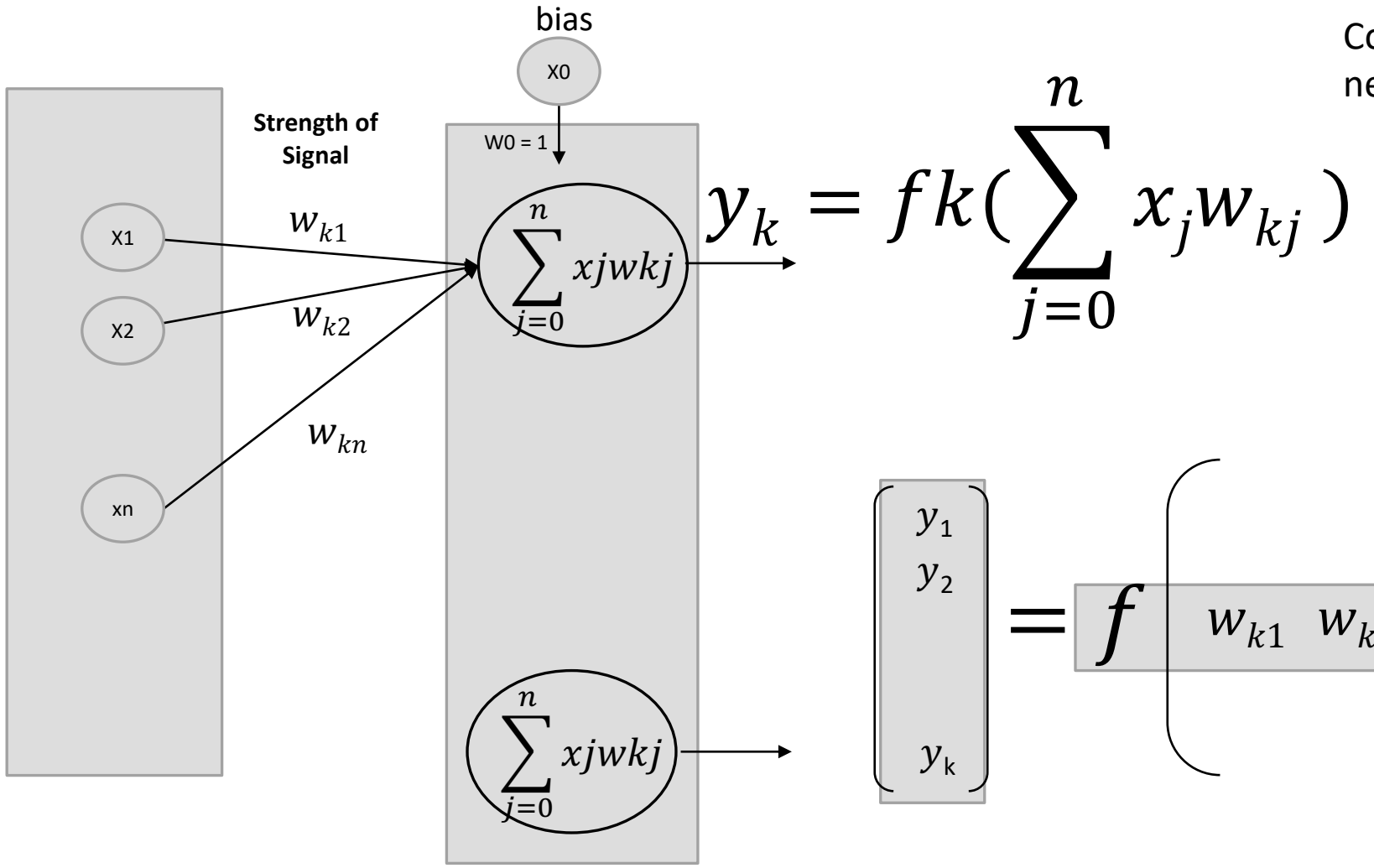
$$a_3^{(2)} = f\left(\sum_{j=0}^{n} w_{3j}\, x_j\right) =$$



$$a_i^{(2)} = f\left(\sum_{j=0}^{n} w_{ij}\, x_j\right)$$

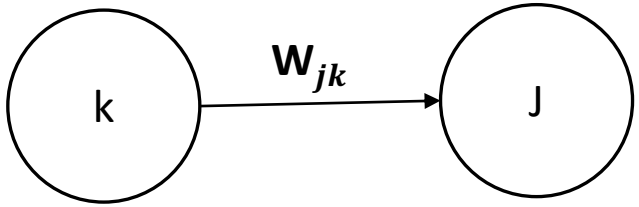
$$\boldsymbol{h\,(x) = \; f\,(w_3 a_3^{(2)} \; + w_2 a_2^{(2)} + w_1 a_1^{(2)} + a_0)}$$

Multi Layer NN Nonlinear Math Model



$$y_k = fk\left(\sum_{j=0}^{n} x_j w_{kj}\right)$$

Connection weight from neuron in layer k to neuron in layer j = $\mathbf{W_{jk}}$



$$\begin{pmatrix} y_1 \\ y_2 \\ \\ \\ y_k \end{pmatrix} = f \begin{pmatrix} w_{k1} & w_{k2} & & w_{kj} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \\ \\ xj \end{pmatrix}$$

Error

$$x_j^{(l)} = \theta\left(\sum_{i=0}^{d} w_{ij}^{(l-1,l)} x_i^{(l-1)}\right) \qquad h(x) = x_j^{(l)} \text{ when } l = L$$

If we have all the weights $w=\left\{w_{ij}^{(l-1,l)}\right\}$ for each layer then we can determine $h(x)$.

Error on example (X_n, Y_n) is $e(h(X_n), Y_n) = e(w)$

We need to calculate Gradient:

$$\nabla \mathbf{e}(\mathbf{w}): \quad \frac{\partial\, \mathbf{e}(\mathbf{w})}{\partial\, w_{ij}^{(l)}} \quad \text{for all} \quad i, j, l$$

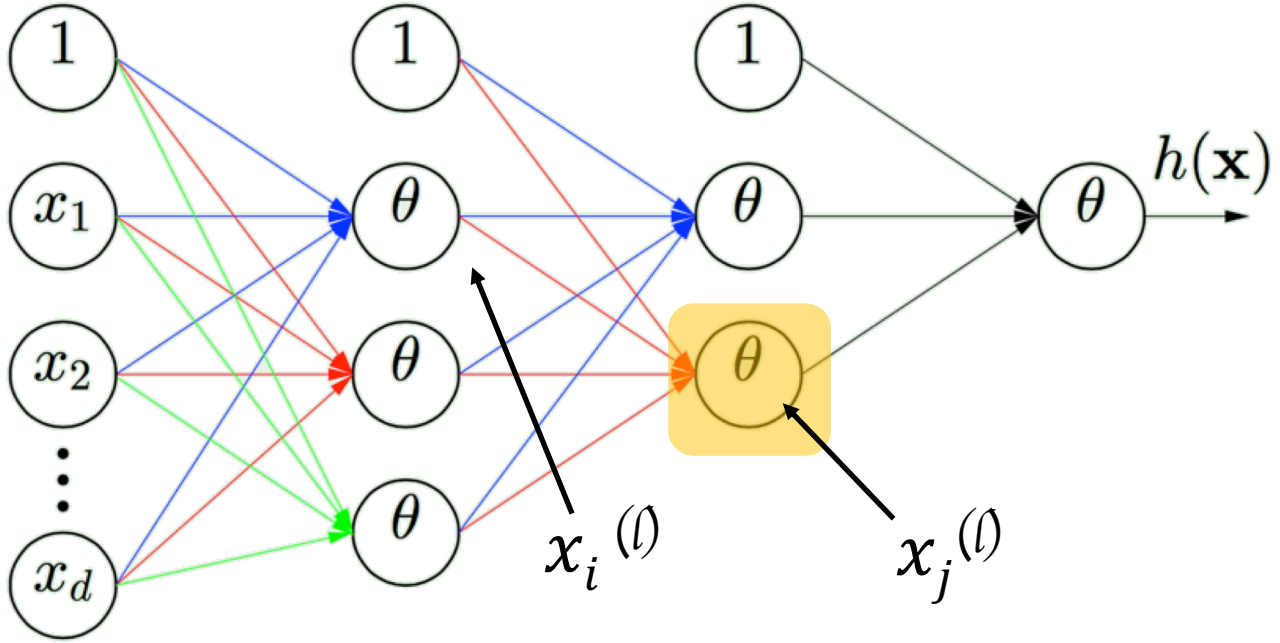
Calculating partial derivatives of $e(W)$

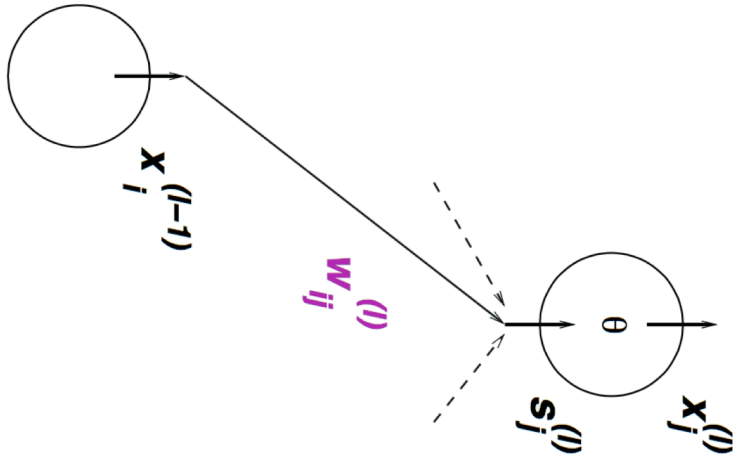
$$\nabla \mathbf{e}(\mathbf{w}): \quad \frac{\partial\, \mathbf{e}(\mathbf{w})}{\partial\, w_{ij}^{(l)}} \quad \text{for all} \quad i, j, l$$

$$\frac{\partial\, \mathbf{e}(\mathbf{w})}{\partial\, w_{ij}^{(l)}} = \frac{\partial\, \mathbf{e}(\mathbf{w})}{\partial\, s_j^{(l)}} \times \frac{\partial\, s_j^{(l)}}{\partial\, w_{ij}^{(l)}}$$

$$\frac{\partial\, \mathbf{e}(\mathbf{w})}{\partial\, s_j^{(l)}} = \delta_j^{(l)}$$

$$x_i^{(l-1)}$$





For the final layer $\delta_j^{(l)}$

$$\frac{\partial\, e(w)}{\partial\, s_j^{(l)}} = \delta_j^{(l)}$$

For the final layer $l = \mathcal{L}$ and $j=1$

$$e(w) = (x_1^{(\mathcal{L})} - y)^2 \qquad x_1^{(\mathcal{L})} = \Theta(s_1^{(\mathcal{L})}) \qquad \Theta'(s) = \Theta(x)\,(1 - \Theta(x))$$

$$\frac{\partial\, e(w)}{\partial\, s_j^{(l)}} = \delta_j^{(l)}$$

$$\frac{\partial\, (\Theta(s_1^{(\mathcal{L})}) - y)^2}{\partial\, s_j^{(l)}}$$

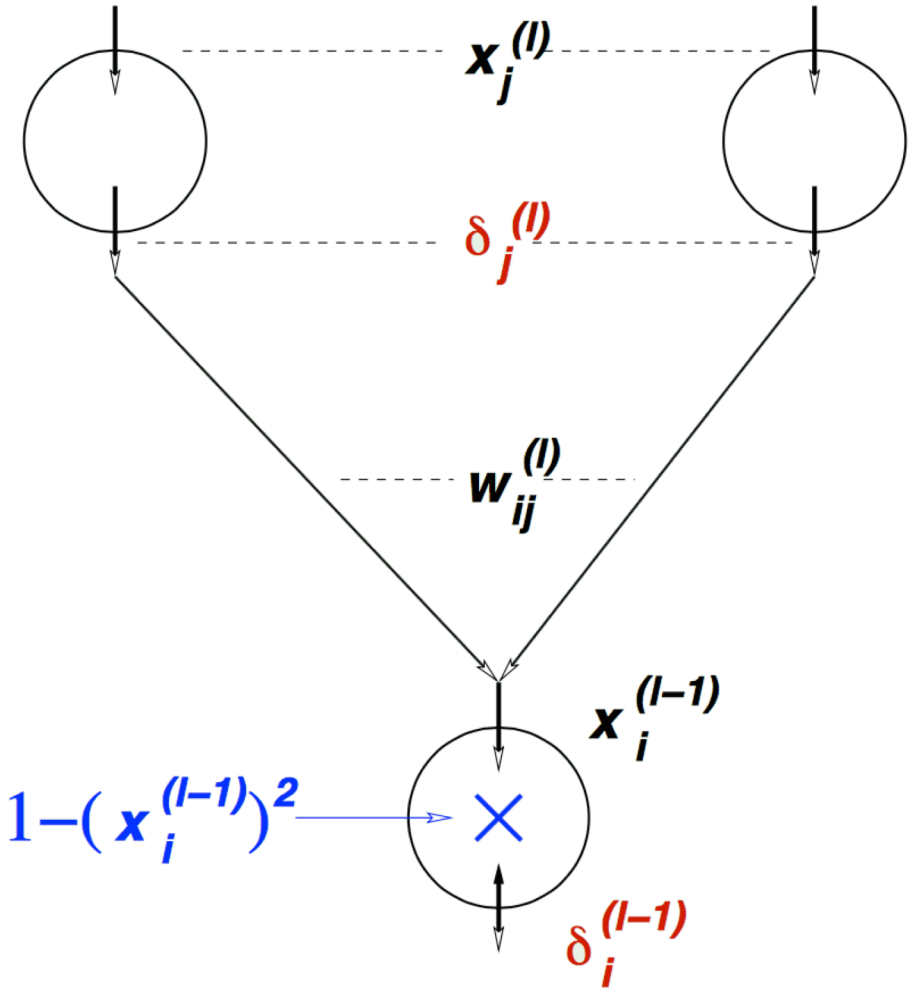
Back propagation of $\delta_j^{(l)}$

$$\delta_i^{(l-1)} = \frac{\partial\, \mathbf{e}(\mathbf{w})}{\partial\, s_i^{(l-1)}}$$

$$= \sum_{j=1}^{d^{(l)}} \frac{\partial\, \mathbf{e}(\mathbf{w})}{\partial\, s_j^{(l)}} \times \frac{\partial\, s_j^{(l)}}{\partial\, x_i^{(l-1)}} \times \frac{\partial\, x_i^{(l-1)}}{\partial\, s_i^{(l-1)}}$$

$$= \sum_{j=1}^{d^{(l)}} \delta_j^{(l)} \times w_{ij}^{(l)} \times \theta'(s_i^{(l-1)})$$

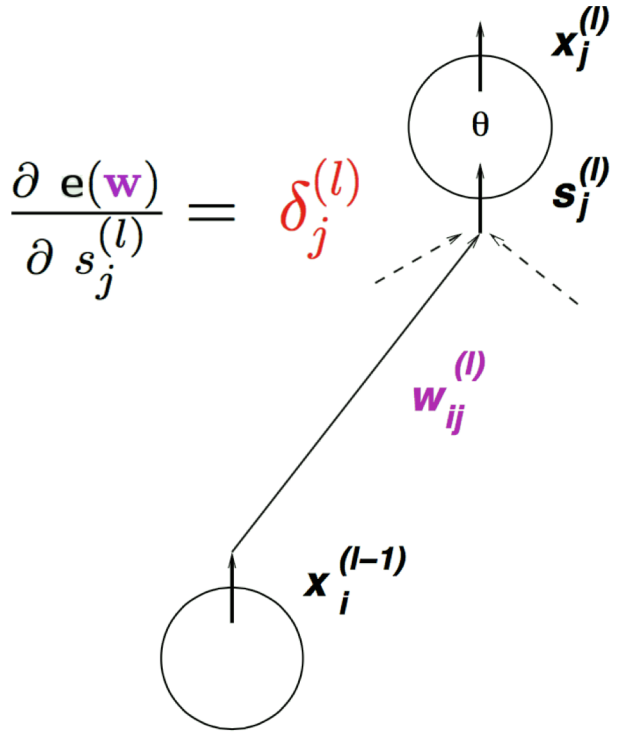
$$\delta_i^{(l-1)} = x_i^{(l-1)} \left(1 - x_i^{(l-1)}\right) \sum_{j=1}^{d^{(l)}} w_{ij}^{(l)}\, \delta_j^{(l)}$$



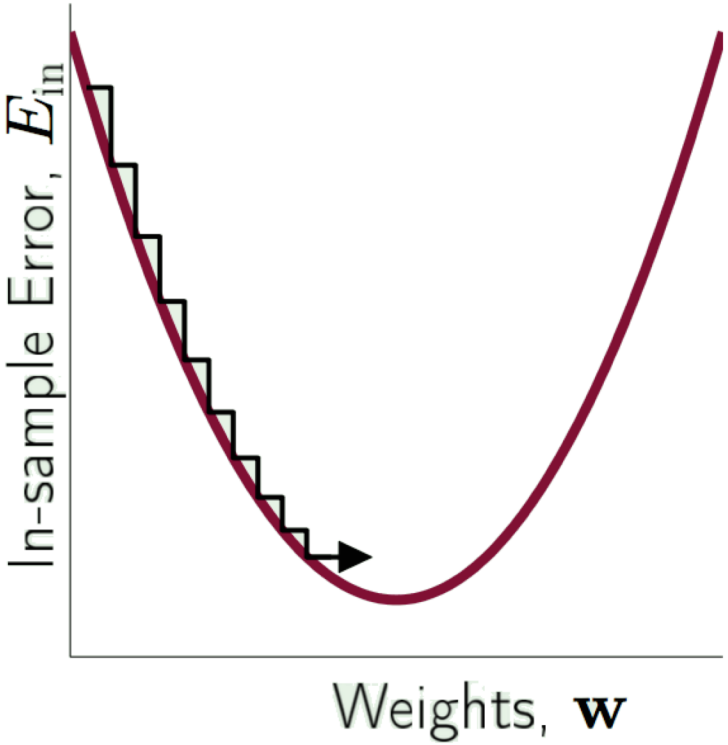
Gradient Descent - Backpropagation Algorithm

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

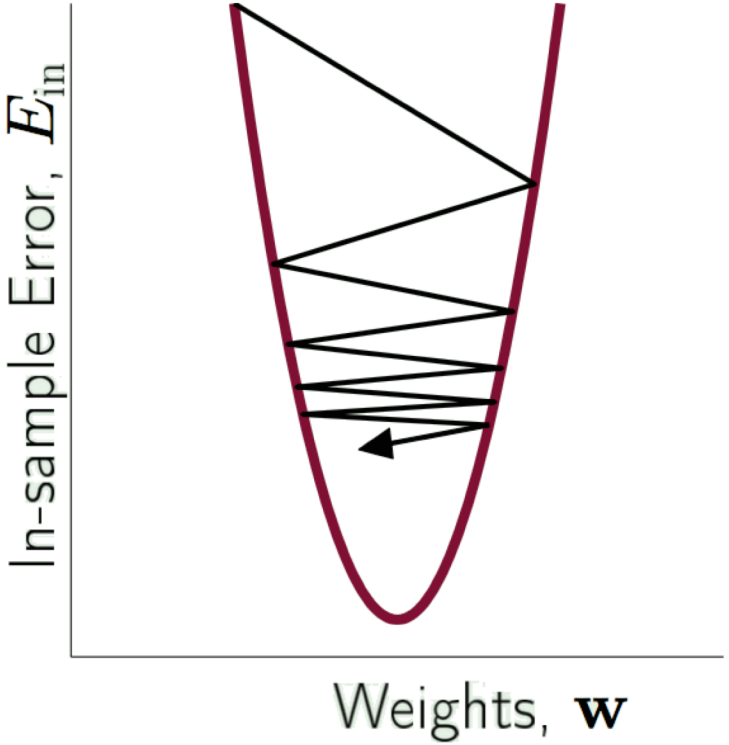
$$w_j = w_j - \eta \frac{d}{dw_j} loss(w)$$



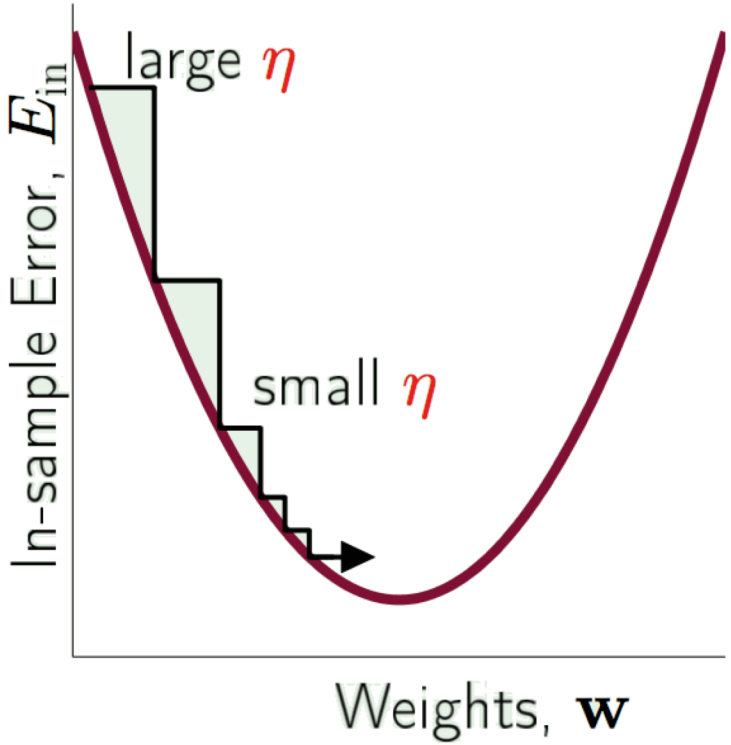
Learning Rate: Steps



η too small



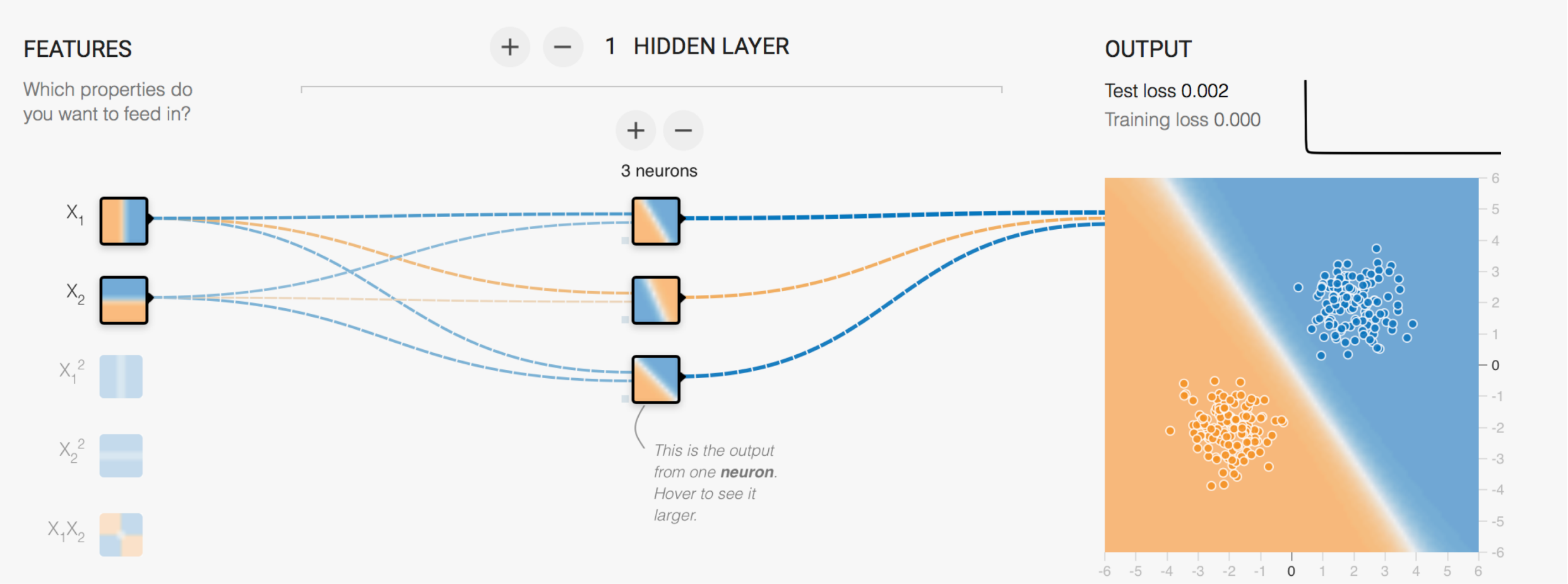
η too large



variable η – just right

Learning Rate should increase with the slope

Neural Network Playground

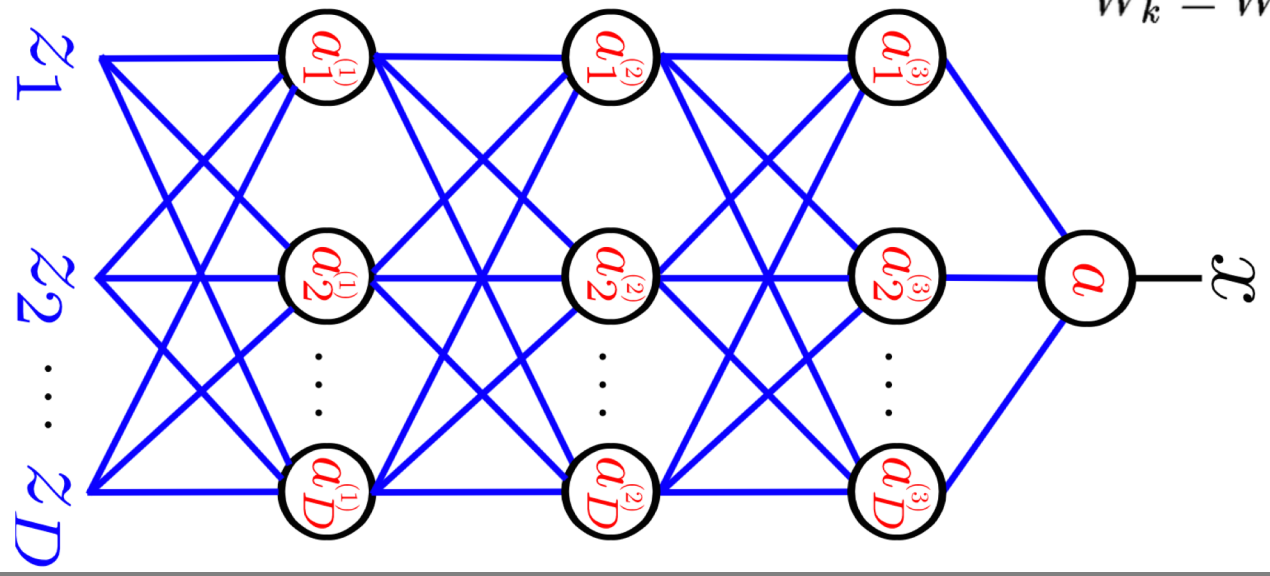


Stochastic Gradient Descent (SGD)

Loop:

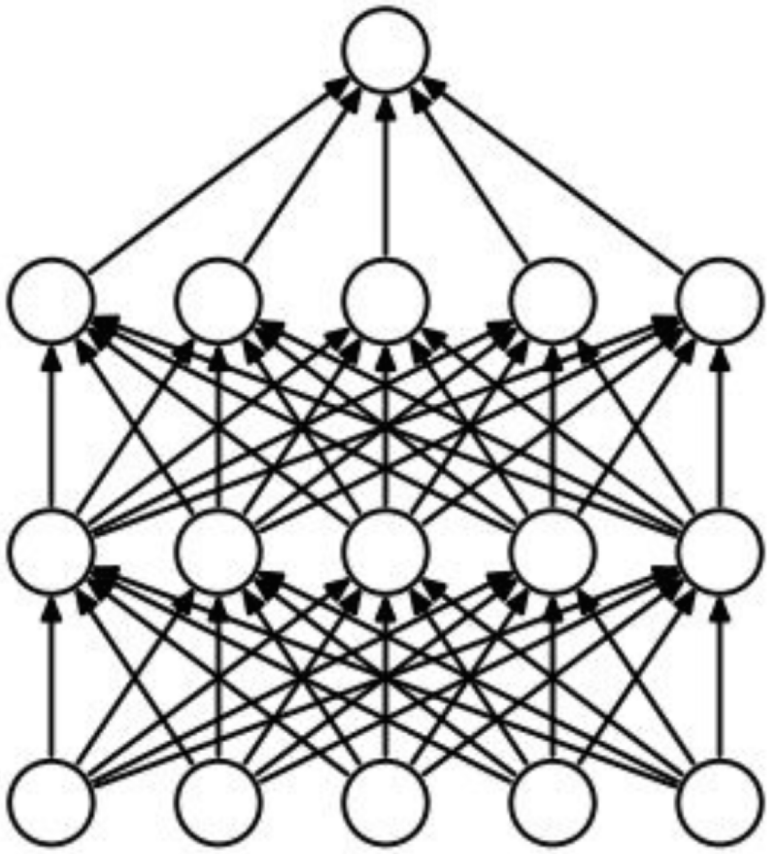
1. **Sample a batch of data with their labels**
2. **Forward Propagate it through the graph, calculate the error**
3. **Backpropagate to calculate the gradients**
4. **Update the parameters (weights) using the gradient**

$$W_k = W_{k-1} - \epsilon \frac{\partial E(W)}{\partial W}$$

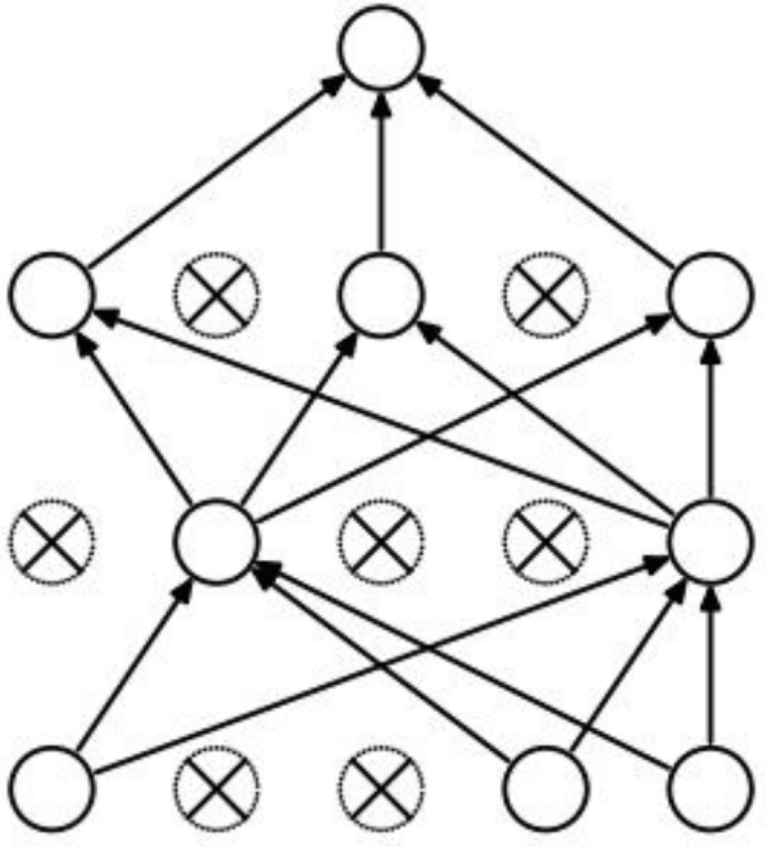


Dropout

Randomly set some neurons to zero in the forward pass

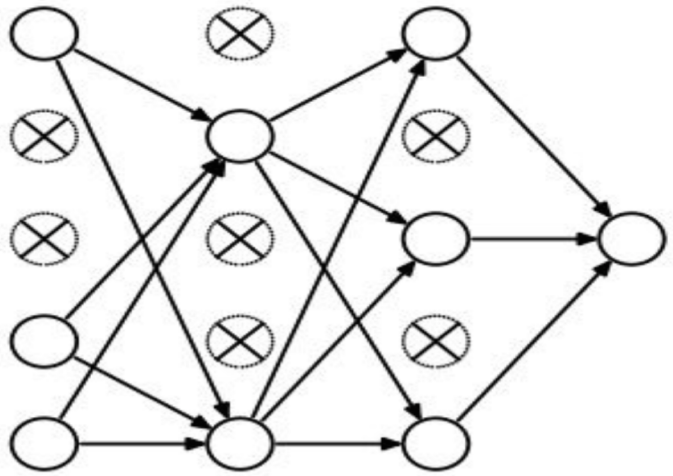


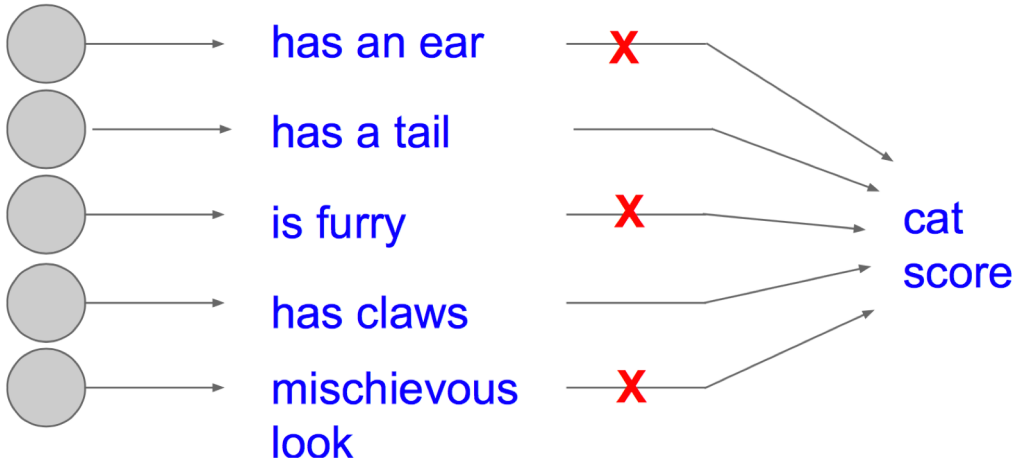
After Applying Dropout



Srivastava et al., 2014, https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

Dropout





Forces the network to have a redundant representation

What's wrong with standard neural networks?

Hard to Train

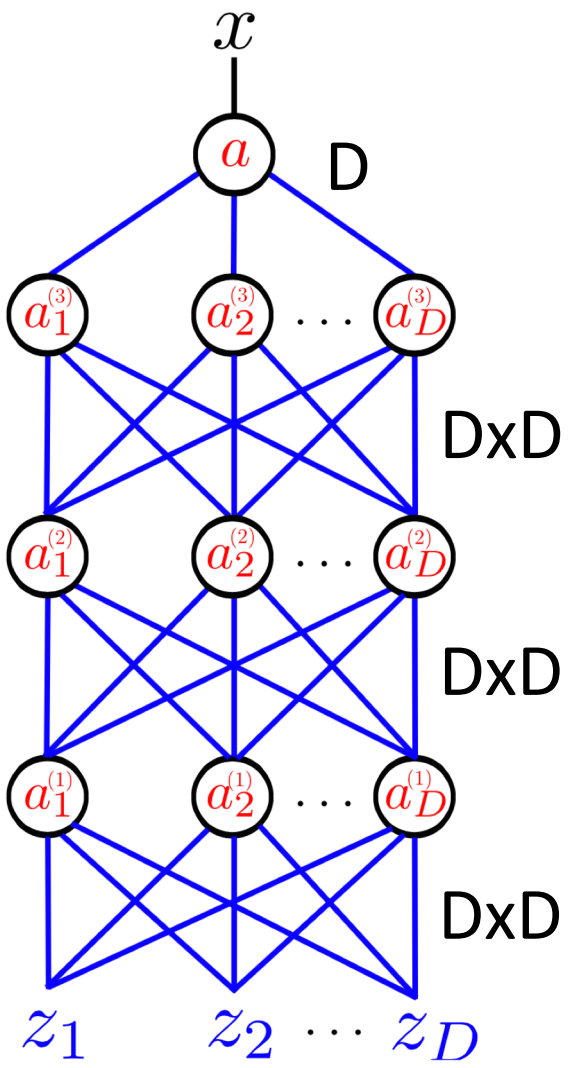
How many parameters does this network have?

Number of Parameters = 3 x (DxD) + D

For a small D = 32 x 32 = 1024 MNIST image:

Number of Parameters = 3 x (1024x1024) + 1024

~ 3×10^6



Gradient descent

- pick a starting point (w)
- repeat until loss doesn't decrease in all dimensions:
  - pick a dimension
  - move a small amount in that dimension towards decreasing loss (using the derivative)

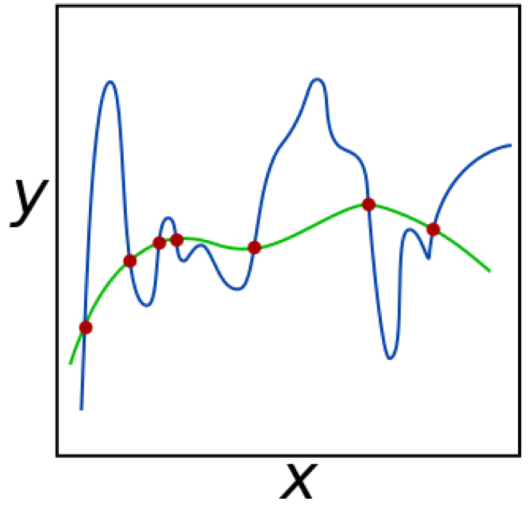
$$w_j = w_j - \eta \frac{d}{dw_j} loss(w)$$

Overfitting revisited: regularization

A regularizer is an additional criteria to the loss function to make sure that we don't overfit

It's called a regularizer since it tries to keep the parameters more normal/regular

It is a bias on the model forces the learning to prefer certain types of weights over others



$$\text{argmin}_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda\ regularizer(w,b)$$

Regularizers

Generally, we don't want huge weights

If weights are large, a small change in a feature can result in a large change in the prediction

Also gives too much weight to any one feature

Might also prefer weights of 0 for features that aren't useful

How do we encourage small weights? or penalize large weights?

Regularizers

How do we encourage small weights? or penalize large weights?

$$\text{argmin}_{w,b} \sum_{i=1}^{n} loss(yy') + \lambda \; regularizer(w,b)$$

Common regularizers

sum of the weights $r(w,b) = \sum_{w_j} |w_j|$

sum of the squared weights $r(w,b) = \sqrt{\sum_{w_j} |w_j|^2}$

Squared weights penalizes large values more
Sum of weights will penalize small values more