03 | Introduction to Deep Neural Networks

Stephen F Elston, Principle Consultant, Quantia Analytics, LLC

Building Blocks of Deep Learning

- Forward propagation and linear networks
- The perceptron
- Better representations
- Deep architectures; depth vs. breath
- Nonlinearity and activation functions
- Learning with backpropagation
- Loss function
- Computing gradients with the chain rule
- Performance metrics

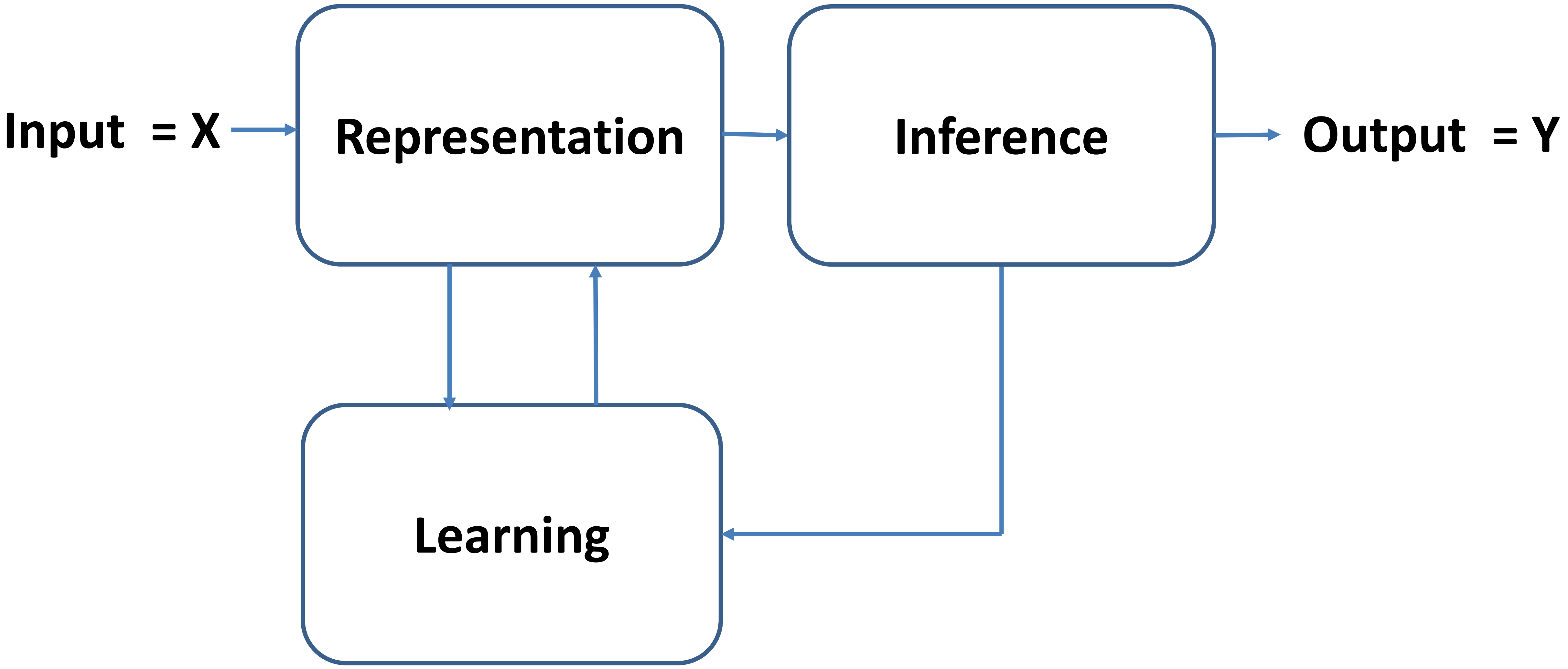
Function Approximation with Deep Neural Networks

- Deep neural networks are powerful **function approximators**

$$y = f(x)$$

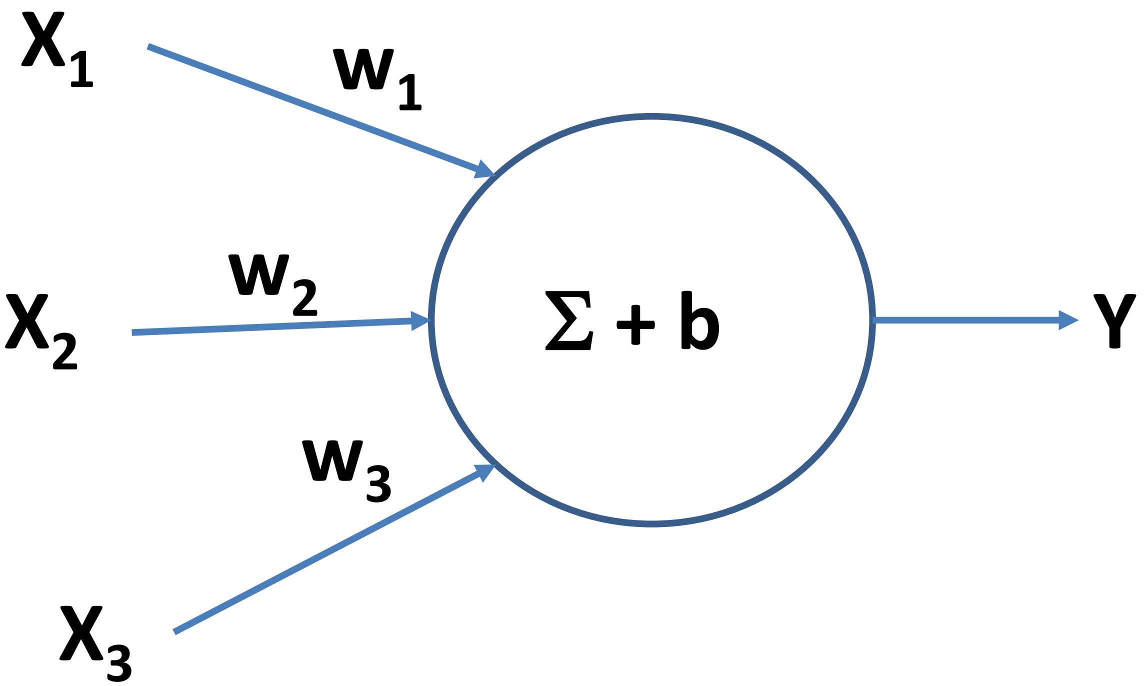
- Most deep neural networks use **supervised learning**
  - Labelled cases used to learn f(x)
  - f(x) is nonlinear and can be quite complex
  - Complexity leads to problems with generalization

Essential Elements of Deep Learning



Representation: Linear Neural Network

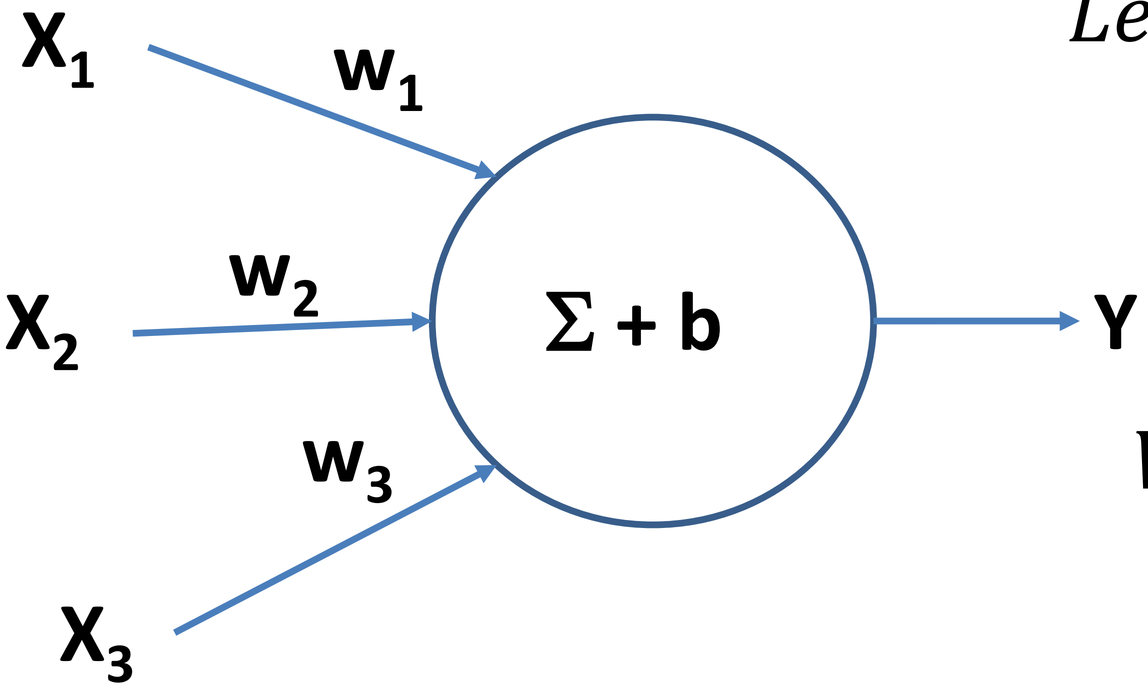
Proposed by McCulloch and Pitts (1943)



$$y = f(x) = \sum_i w_i \cdot x_i + b$$

Representation: Linear Neural Network

Early learning model for a neural network - Heeb (1949)



Learning model for the weights

$$w_{ij} = x_i x_j$$

$$\Delta w_{ij} = \eta x_i x_j$$

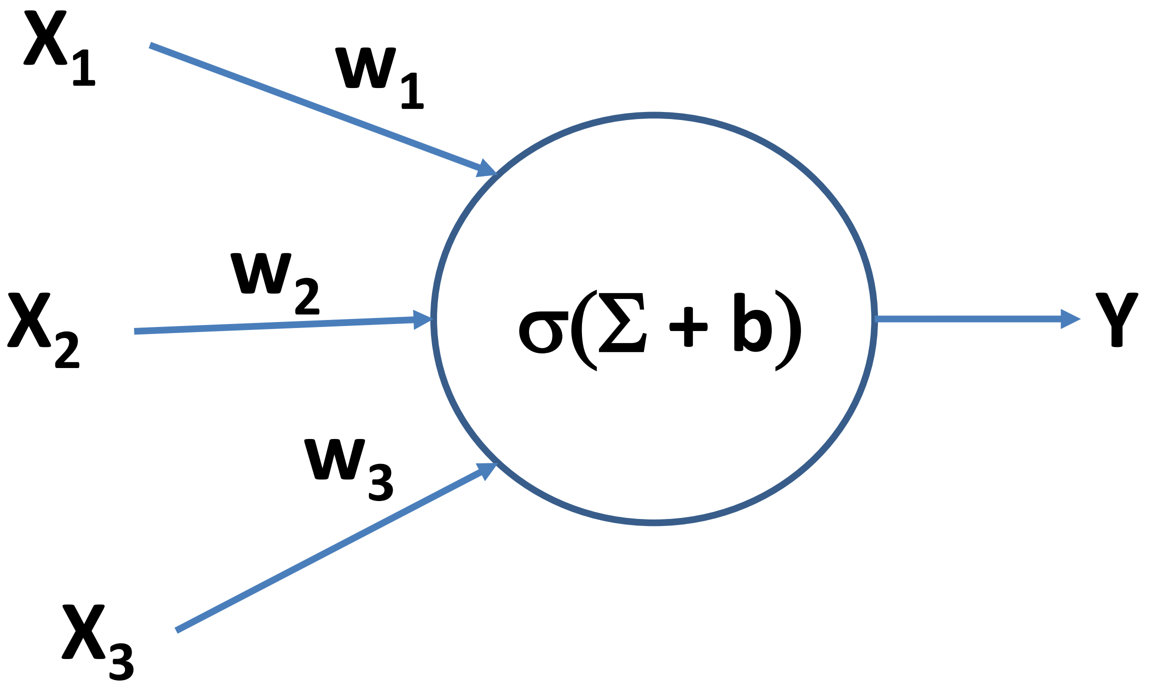
Where, η is the learning rate

$$y = f(x) = \sum_i w_i \cdot x_i + b$$

But, this is just **linear regression!**

Representation: Perceptron

Use of **nonlinear activation** proposed by Rosenblatt (1962)



$$y = f(x) = \sigma\left(\sum_i w_i \cdot x_i + b\right)$$

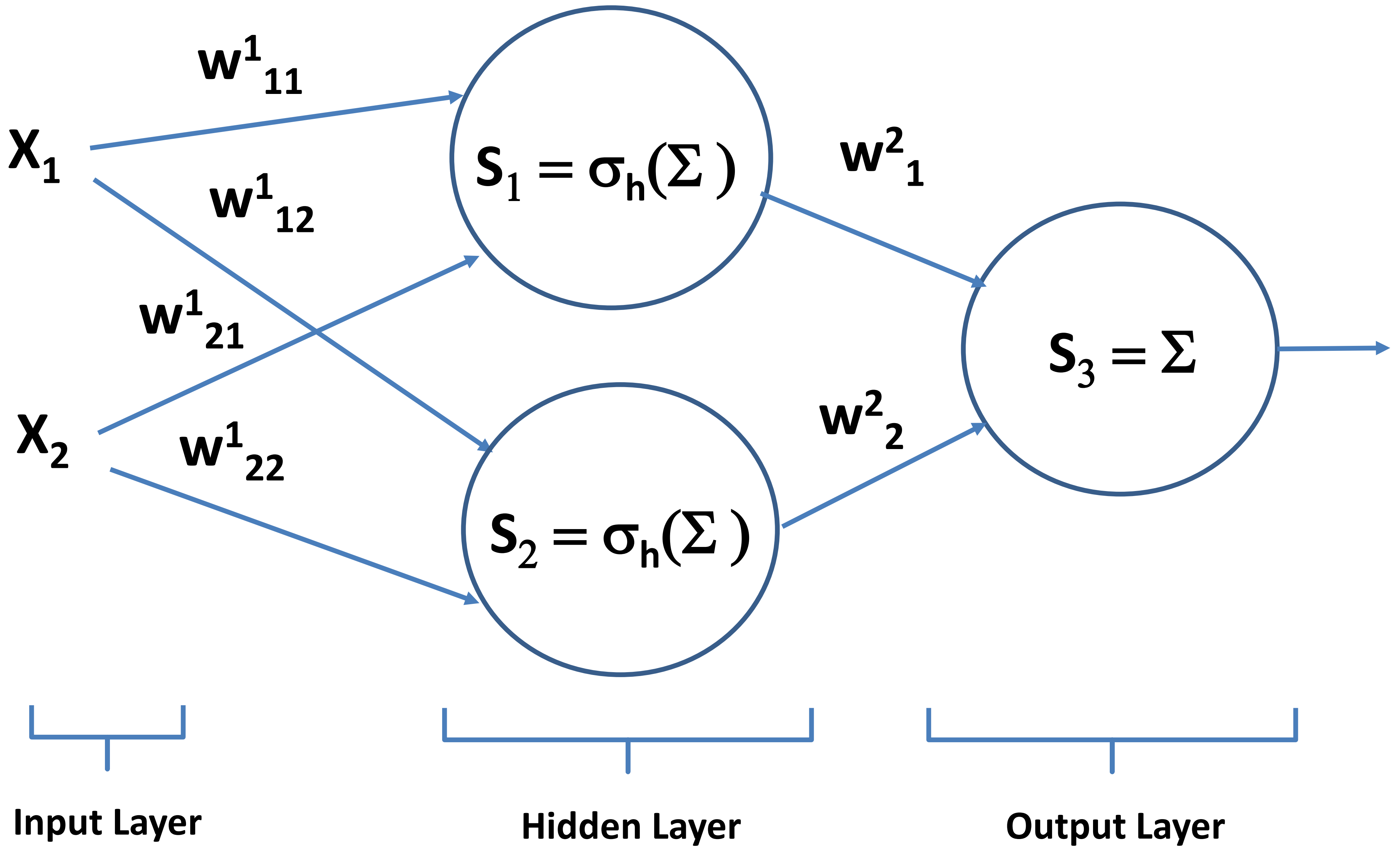
This is just **logistic regression**!

Minsky and Papert (1969) showed the perceptron cannot represent an **exclusive or (XOR)**

We Need a Better Deep Representation

- By mid-1980s need for architecture with **hidden layers** for **greater model capacity** was recognized
  - **Input layer**
  - Multiple **hidden layers**
  - **Output layer**
- Apply **nonlinear activations** in **hidden units**
- Can **fully connect** between layers
- **Learn weights** for complex function approximation
- Can solve XOR problem and much more

We Need a Better Deep Representation



We Need a Better Deep Representation

- What is the output of the simple network?
- Start with the output of the hidden layer:

$$S_1 = \sigma(\Sigma_i \, x_i * W^1_{1i})$$

$$S_2 = \sigma(\Sigma_i \, x_i * W^1_{2i})$$

- Next, compute the output of the output layer

$$S_3 = \Sigma_j \, W^2_j * \sigma(\Sigma_i \, x_i * W^1_{ji})$$

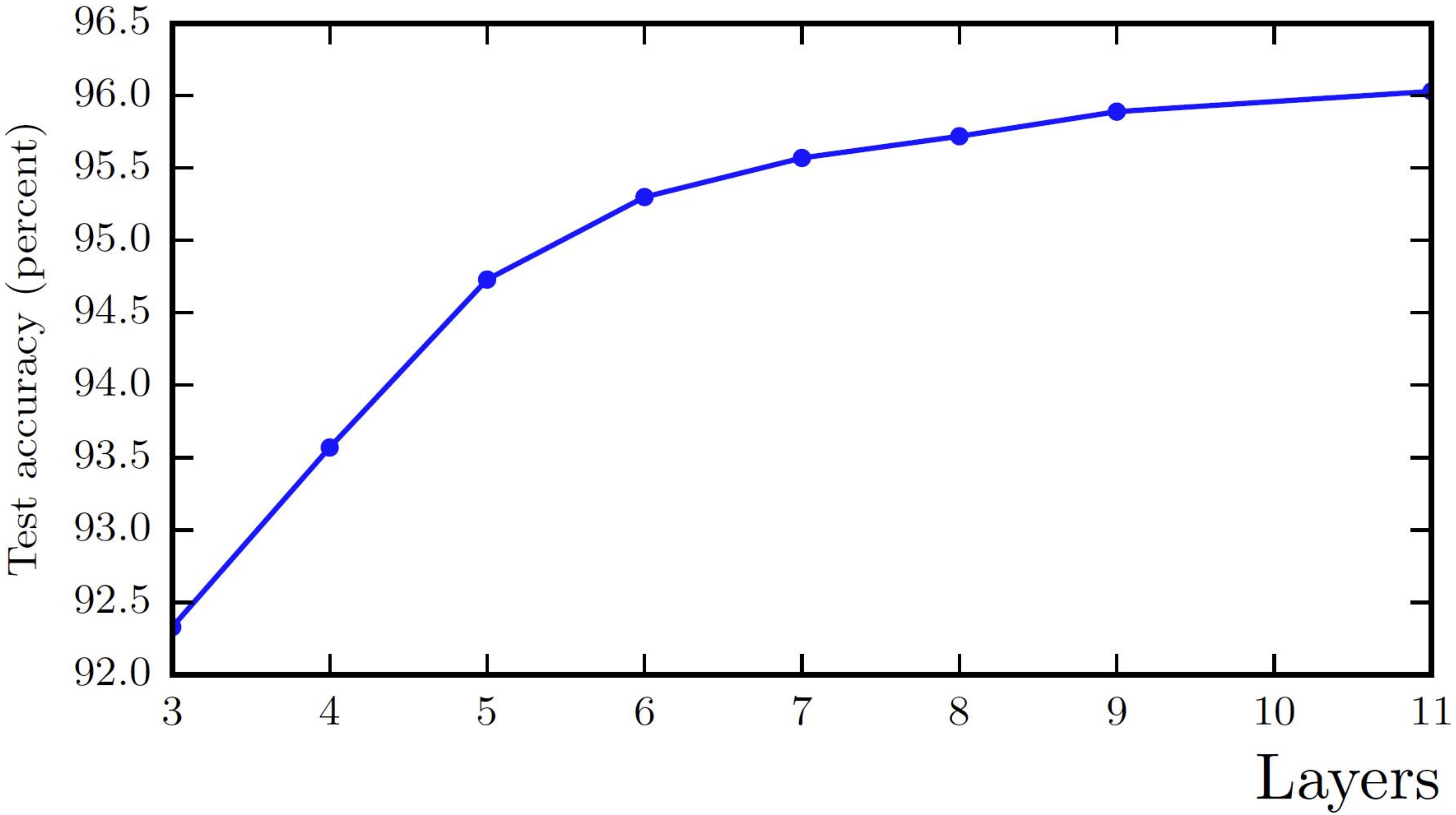
Model Capacity

- The **universal approximation theorem**, Hornik (1991), tells us that an **infinitely wide hidden layer can represent any function**
- Usefulness limited:
  - It’s nice to know we can represent complex functions
  - But, completely **infeasible** in practice
- What can we do?
  - Trade depth for breath

Model Capacity

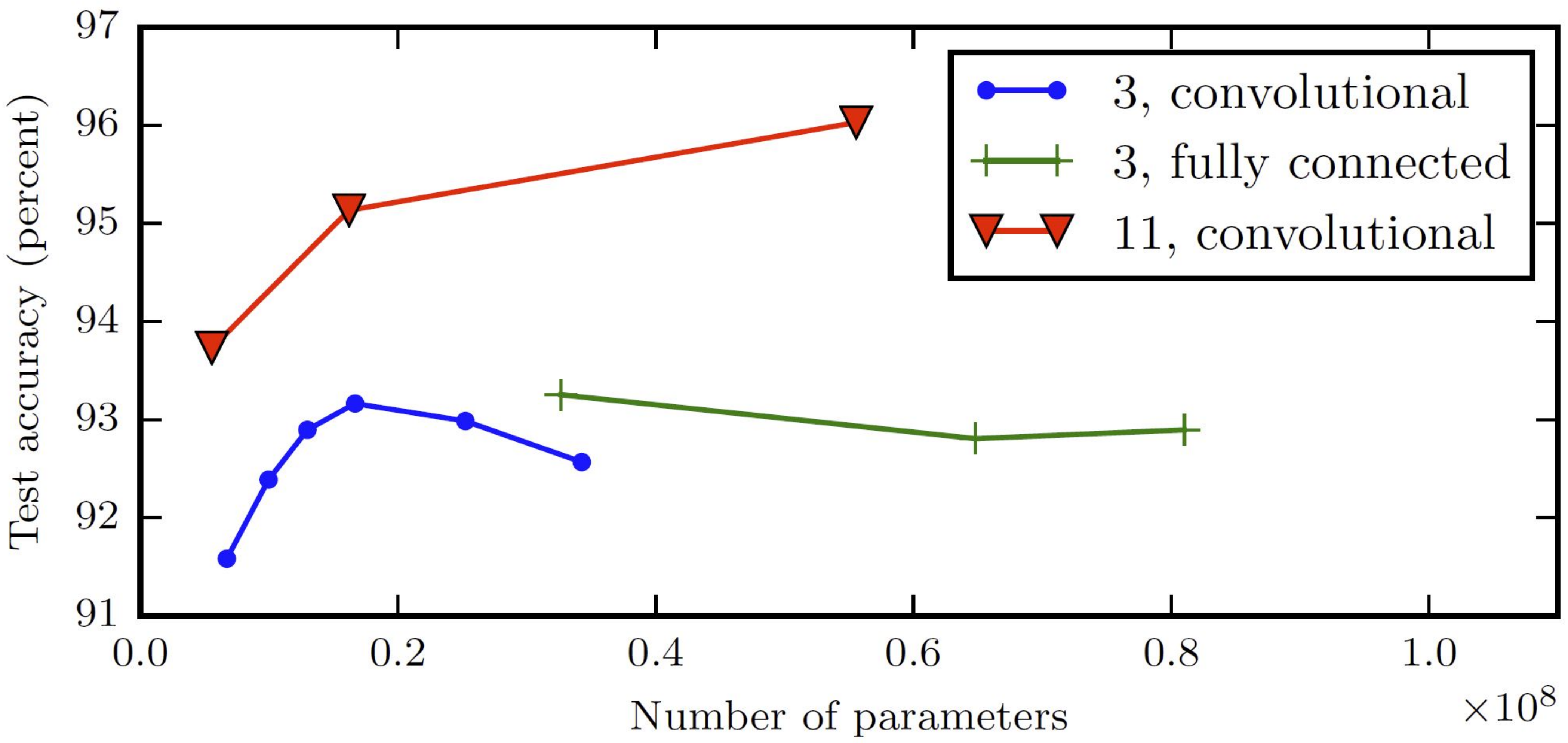
- Model capacity is fundamentally related to the **bias-variance trade-off** of machine learning
  - **Low capacity** models have **high bias but low variance**
  - **High cap**acity models have **low bias but high variance**
- High capacity models have a tendency to be overfit
- We have more to say about this problem in another lesson

Model Capacity



Model capacity with increasing depth. From Goodfellow et. al. 2014.

Model Capacity



Model capacity vs. number of parameters. From Goodfellow et. al. 2014.

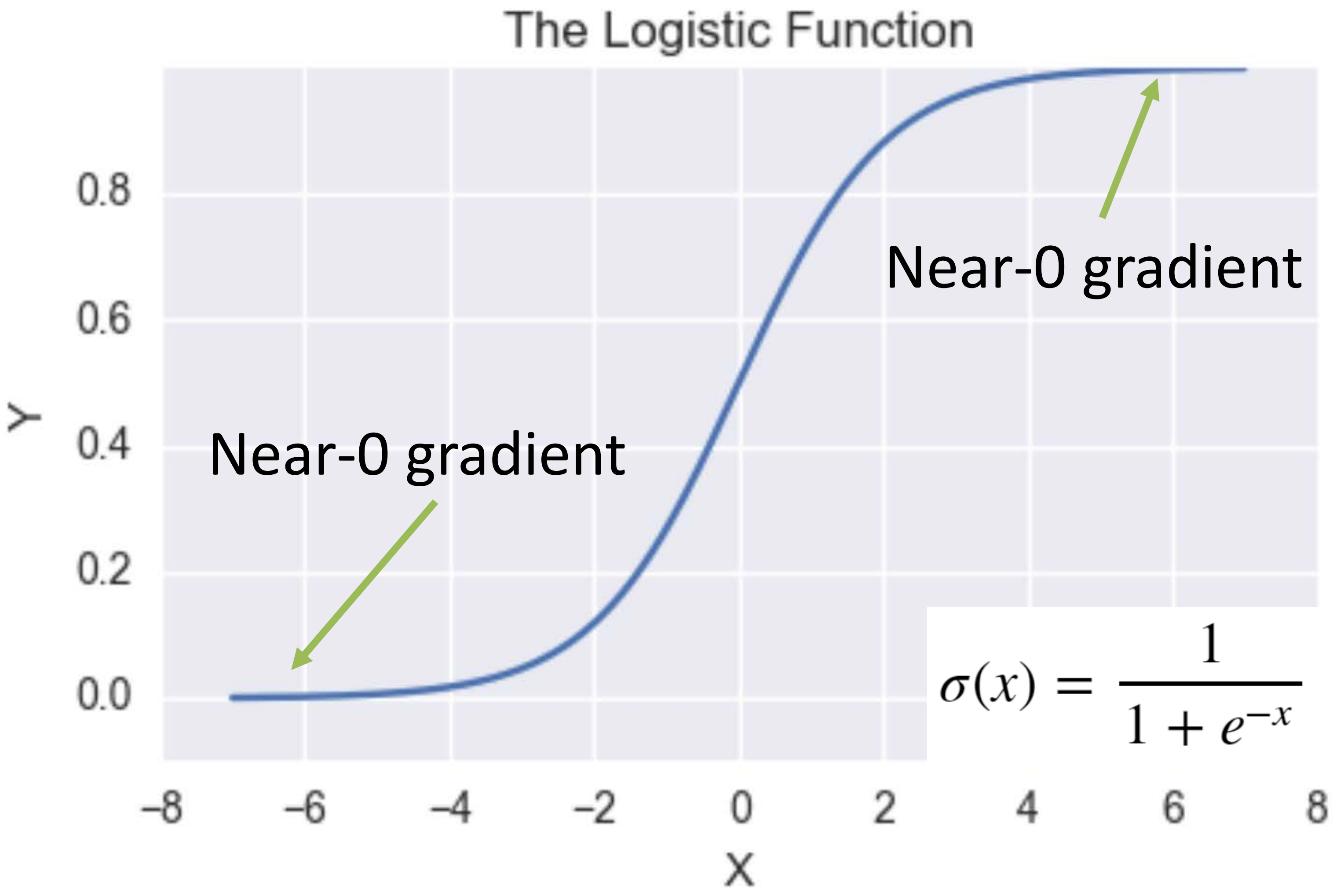
Activation functions

- Nonlinear activation is key to achieving good function approximation.
- Many activation functions have been tried, here are a few:

Function	How Used?	Comments
Sigmoid	Binary classifier output layer	Historically the most used

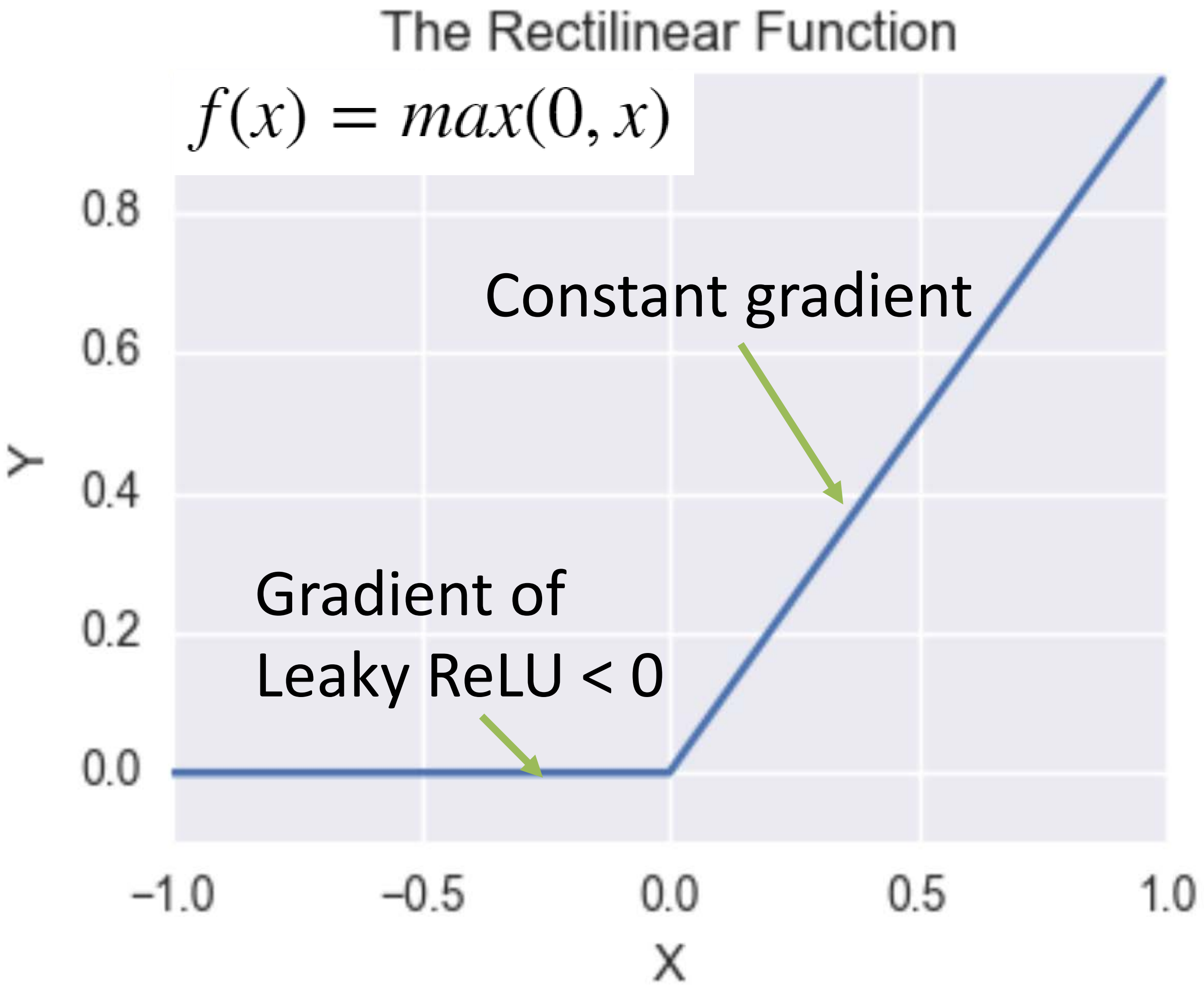
Activation functions

Sigmoid has vanishing gradients



Activation functions

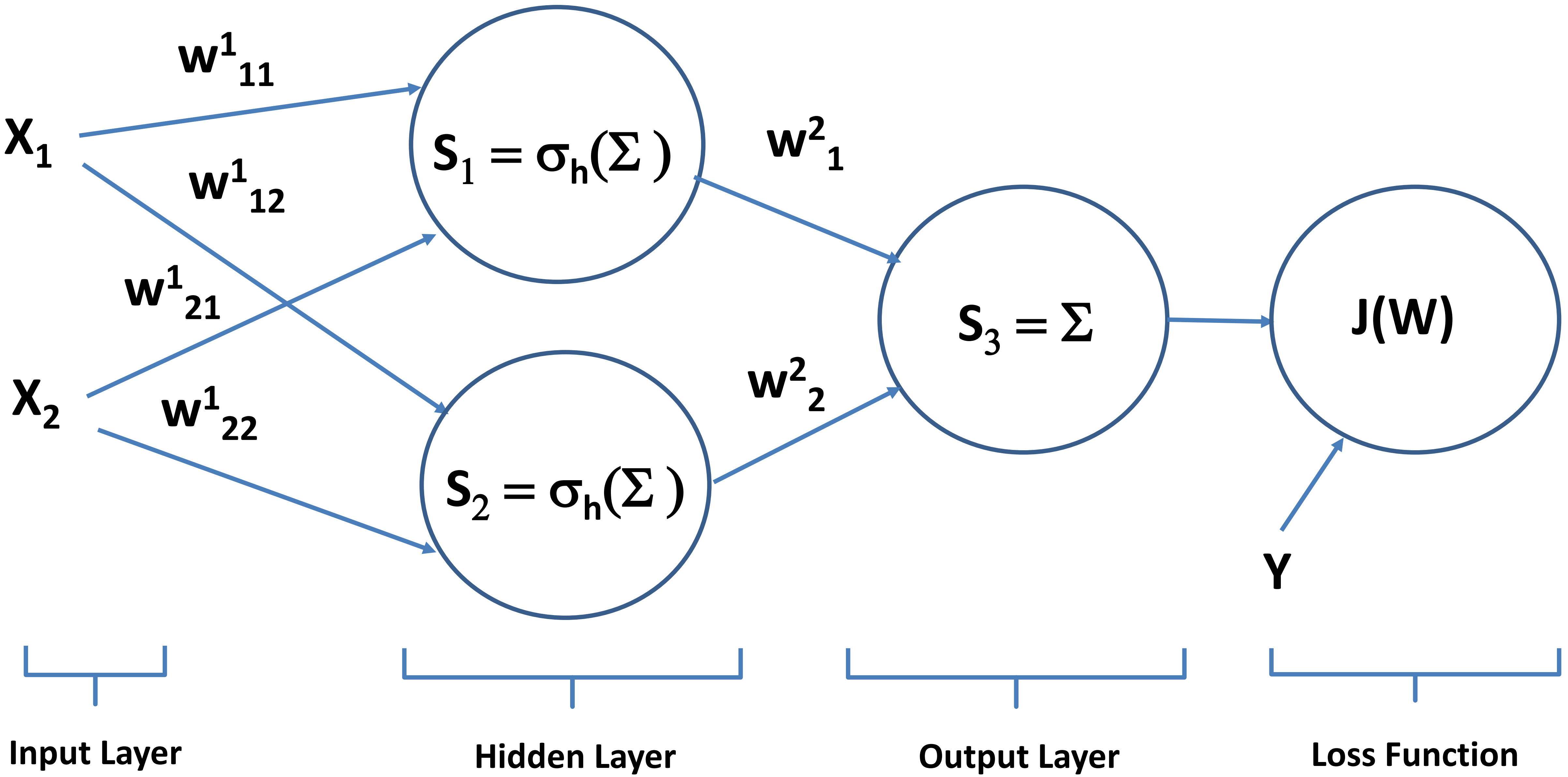
Rectilinear function has constant gradient for positive values



The Backpropagation Algorithm

- To find function approximation, f(x), we need to **learn model weights**
- The primary algorithm we use to learn model weights is known as **backpropagation**
  - Backpropagation was applied to learning (system identification) for control problems as early as 1960 by Henry Kelly and 1961 by Arthur Bryson for dynamic programming
  - First applied to neural networks by Paul Werbos in 1974
  - In 1986 by Rumelhart, Hinton and Williams showed that backpropagation was effective for learning the weights of hidden layers

The Backpropagation Algorithm



The Backpropagation Algorithm

To **learn model weight tensor** we must **minimize the loss function** using the **gradient:**

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t)$$

Where**:**

W_t = the tensor of weights or model parameters at step t

$J(W)$ = loss function given the weights

$\nabla_W J(W)$ = gradient of J with respect to the weights W

α = step size or learning rate

The Back Propagation Algorithm

- Backpropagation is a **gradient decent algorithm**
- Weight updates are taken as small steps in the direction of the gradient of the loss function

$$\alpha \nabla_W J(W_t)$$

- Backpropagation converges when the gradient is approximately 0

Loss Functions for Training Neural Networks

- What are some choices for a loss function, $J(W)$, given the weight tensor?
- For regression problems use MSE
- Which loss function should we use for classification problems?
  - **Cross entropy** is a good choice, but is a bit abstract

Loss Functions for Training Neural Networks

What is **Shannon Entropy**?

$$\mathbb{H}(I) = E[I(X)]$$

Where: $E[X] =$ the expectation of X.
$I(X) =$ the information content of X.

But, we work with probability distributions, so:

$$\mathbb{H}(I) = E[-ln_b(P(X))] = -\sum_{i=1}^{n} P(x_i)ln_b(P(x_i)$$

Where: $P(X) =$ probability of X.
$b =$ base of the logarithm.

Loss Functions for Training Neural Networks

- We need to measure the difference between the distribution of our function approximation and the distribution of the data
- The **Kullback-Leibler divergence** between **two distributions P(X) and Q(X)** is such a measure:

$$\mathbb{D}_{KL}(P \parallel Q) = -\sum_{i=1}^{n} p(x_i)\; ln_b \frac{p(x_i)}{q(x_i)}$$

Loss Functions for Training Neural Networks

- How do we compute KL divergence?
- If we knew P(X) we would not need to compute KL divergence
- We can expand KL divergence as:

$$\mathbb{D}_{KL}(P \parallel Q) = \sum_{i=1}^{n} p(x_i)\, ln_b p(x_i) - \sum_{i=1}^{n} p(x_i)\, ln_b q(x_i)$$

$$\mathbb{D}_{KL}(P \parallel Q) = \mathbb{H}(P) + \mathbb{H}(P, Q)$$

$$\mathbb{D}_{KL}(P \parallel Q) = Entropy(P) + Cross\ Entropy(P, Q)$$

Loss Functions for Training Neural Networks

Given: $\mathbb{D}_{KL}(P \parallel Q) = \mathbb{H}(P) + \mathbb{H}(P, Q)$

The term $\mathbb{H}(P)$ is constant

So, we only need the **cross entropy** term:

$$\mathbb{H}(P, Q) = -\sum_{i=1}^{n} p(x_i)\ ln_b q(x_i)$$

Loss Functions for Training Neural Networks

How can we compute cross entropy when we don’t know P(X):

$$\mathbb{H}(P, Q) = -\sum_{i=1}^{n} p(x_i)\; ln_b q(x_i)$$

Since we don’t know P(X), use the approximation:

$$\mathbb{H}(P, Q) = -\frac{1}{N}\sum_{i=1}^{n} ln_b q(x_i)$$

Loss Functions for Training Neural Networks

Consider the special case of **Gaussian likelihood**:

$$p(data|model) = p(data|f(\theta)) = p(x_i|f(\hat{\mu}, \sigma)) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x_i-\hat{\mu})^2}{2\sigma^2}}$$

Taking the negative logarithm:

$$-log\Big(p(data|model)\Big) = -\frac{1}{2}\Big(log(2\pi\sigma^2) + \frac{(x_i - \hat{\mu})^2}{2\sigma^2}\Big)$$

Ignoring the constant terms, the minimum of cross entropy is:

$$min\Big(\mathbb{H}(P, Q)\Big) \propto argmin_{\mu}\Big(- \sum_{i=1}^{n}(x_i - \hat{\mu})^2\Big)$$

The Chain Rule of Calculus

- In order to compute the gradients of the loss function though the layers of a deep neural network we need to apply the **chain rule of calculus**
- To consider a function z = f(y), where y = g(x); then z = f(g(x)). Then the derivative of z with respect to x is:

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

The Chain Rule of Calculus

- We need the gradient of real-valued loss function, J, given a **M** dimensional weight tensor, W
- This leads to the general form of the chain rule:

$$\frac{\partial z}{\partial x} = \sum_{j \in M} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}$$

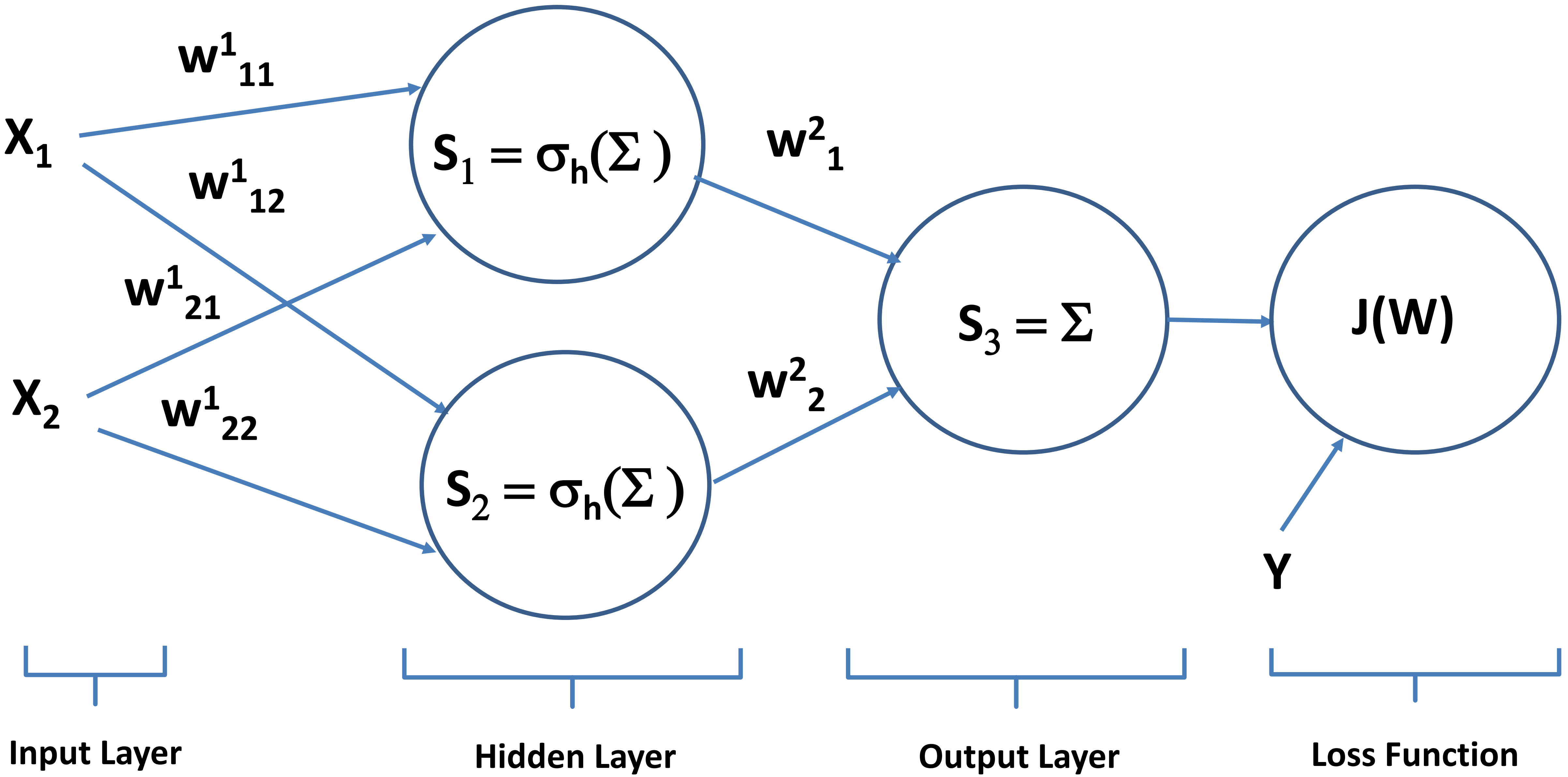
Or,

$$\nabla_x z = \left(\frac{\partial x}{\partial y} \right)^T \nabla_y z$$

Where, $\frac{\partial x}{\partial y}$ = is the nxm **Jacobian matrix** of partial derivatives

$\nabla_y z$ = the gradient of z with respect to y

Example: Computing a Gradient



Example: Computing a Gradient

Start with forward propagation relationships

- The output of the hidden units are computed as:

$$S_{\{1,2\}} = \sigma_h\left(W^1 \cdot X_{\{1,2\}}\right) = \sigma\Big(\sum_j W^1_{i,j} x_j\Big)$$

- The output unit relation is:

$$S_3 = W^2 \cdot S_{\{1,2\}} = \sum_i W^2_i \sigma\Big(\sum_j W^1_{i,j} x_j\Big)$$

Example: Computing a Gradient

Goal is to compute the gradient:

The loss function is:

$$J(W) = -\frac{1}{2}\sum_{l=1}^{n}(y_l - S_{3,l})^2$$

$$\frac{\partial J(W)}{\partial W} = \begin{bmatrix} \frac{\partial J(W)}{\partial W_{11}^2} \\ \frac{\partial J(W)}{\partial W_{12}^2} \\ \frac{\partial J(W)}{\partial W_{21}^2} \\ \frac{\partial J(W)}{\partial W_{22}^2} \\ \frac{\partial J(W)}{\partial W_1^1} \\ \frac{\partial J(W)}{\partial W_2^1} \end{bmatrix}$$

Example: Computing a Gradient

Start with the easier case of the gradient with respect to the output tensor.

- Applying the chain rule yields:

$$\frac{\partial J(W)}{\partial W_k^2} = \frac{\partial J(W)}{\partial S_{3,k}} \frac{\partial S_{3,k}}{\partial W_k^2}$$

Example: Computing a Gradient

The first partial derivative is:

$$\frac{\partial J(W)}{\partial S_{3,k}} = \frac{\partial - \frac{1}{2}(y_k - S_{3,k})^2}{\partial S_{3,k}} = y_k - S_{3,k}$$

The second partial derivative is:

$$\frac{\partial S_{3,k}}{\partial W_k^2} = \frac{\partial W_k^2 S_{j,k}}{\partial W_k^2} = S_{j,l},\ j \in \{1, 2\}$$

And the gradient with respect to the output tensor is then:

$$\frac{\partial J(W)}{\partial W_k^2} = S_{j,k}(y_k - S_{3,k}),\ j \in \{1, 2\}$$

Example: Computing a Gradient

The gradient with respect to the input tensor is a bit more complicated

- Apply the chain rule twice to get:

$$\frac{\partial J(W)}{\partial W^1_{i,j}} = \frac{\partial J(W)}{\partial S_3}\frac{\partial S_3}{\partial S_j}\frac{\partial S_j}{\partial W^1_{i,j}}$$

Example: Computing a Gradient

The output layer has linear activation so the left most partial derivative is just 1.

The middle partial derivative :

$$\frac{\partial S_3}{\partial S_j} = W_j^2$$

The right most partial derivative, given ReLU activation :

$$\frac{\partial S_j}{\partial W_{i,j}^1} = \begin{cases} \frac{\partial W_{i,j}^1 x_{i,k}}{\partial W_{i,j}^1} \\ 0, \end{cases} = \begin{cases} 1, & \text{if } S_j > 0 \\ 0, & \text{otherwise} \end{cases}$$

Example: Computing a Gradient

The gradient with respect to the input weights is then:

$$\frac{\partial J(W)}{\partial W^1_{i,j}} = \frac{\partial J(W)}{\partial S_3}\frac{\partial S_3}{\partial S_j}\frac{\partial S_j}{\partial W^1_{i,j}} = \begin{cases} (y_k - S_{3,k})W^2_j, & \text{if } S_j > 0 \\ 0, & \text{otherwise} \end{cases}$$

Performance Metrics for Deep NNs

- How can we measure the performance of deep neural networks?
  - Use the same metrics used for other machine learning algorithms
  - RMSE, R^2, etc for regression
  - Accuracy, precision, recall, etc. for classification

Microsoft

©2019 Microsoft Corporation. All rights reserved. Microsoft, Windows, Office, Azure, System Center, Dynamics and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries. The information herein is for informational purposes only and represents the current view of Microsoft Corporation as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation. MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.