

# ITCS 6101/8101: Natural Language Processing

---

Feed-Forward Neural Networks

Backpropagation

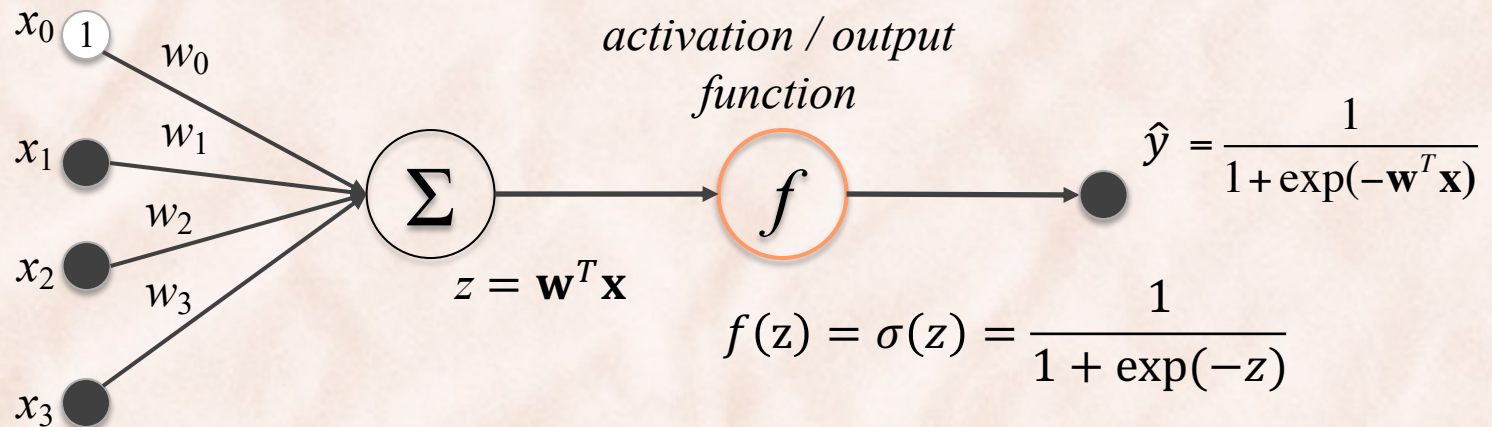
Deep Learning

Razvan C. Bunescu

Computer Science @ CCI

[razvan.bunescu@charlotte.edu](mailto:razvan.bunescu@charlotte.edu)

# Logistic Neuron = Logistic Regression



- Algebraic interpretation:
  - The output of the neuron is a **linear combination** of inputs from other neurons, **rescaled by** the synaptic **weights**.
    - weights  $w_i$  correspond to the synaptic weights (activating or inhibiting).
    - summation corresponds to combination of signals in the soma.
  - It is often transformed through a monotonic **activation function**.

# Activation Functions

*unit step*  $f(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$

**Perceptron**

*logistic*  $f(z) = \frac{1}{1 + e^{-z}}$

**Logistic Neuron**

*ReLU*  $f(z) = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{if } z \geq 0 \end{cases}$

**Rectified Linear Unit**

$f(z) = \text{ramp}(z) = \max(0, z)$

**GELU, SwiGLU, Swish / SiLU, ...**

More activation functions in [GLU Variants Improve Transformer, by Noam Shazeer, Google 2020.](#)

# Perceptron vs. Logistic Neuron

---

- **Logistic neuron = Logistic regression:**

- At inference time, same decision function as **perceptron**, for binary classification with equal misclassification costs (prove it):

$$\hat{t}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w}^T \mathbf{x} > 0 \\ 0 & \text{otherwise} \end{cases}$$

- **Perceptron** cannot represent the XOR function:
  - **Logistic neuron, ReLU, Tanh** have the same limitation.

- How can we use (**logistic**) **neurons** to achieve better representational power?

# Universal Approximation Theorem

[Hornik \(1991\), Cybenko \(1989\)](#)

---

- Let  $\sigma$  be a nonconstant, bounded, and monotonically-increasing continuous function;
  - Let  $I_m$  denote the  $m$ -dimensional unit hypercube  $[0,1]^m$ ;
  - Let  $C(I_m)$  denote the space of continuous functions on  $I_m$ ;
- **Theorem:** Given any function  $f \in C(I_m)$  and  $\varepsilon > 0$ , there exist an integer  $N$  and real constants  $\alpha_i, b_i \in \mathbb{R}$ ,  $\mathbf{w}_i \in \mathbb{R}^m$ , where  $i = 1, \dots, N$ , such that:

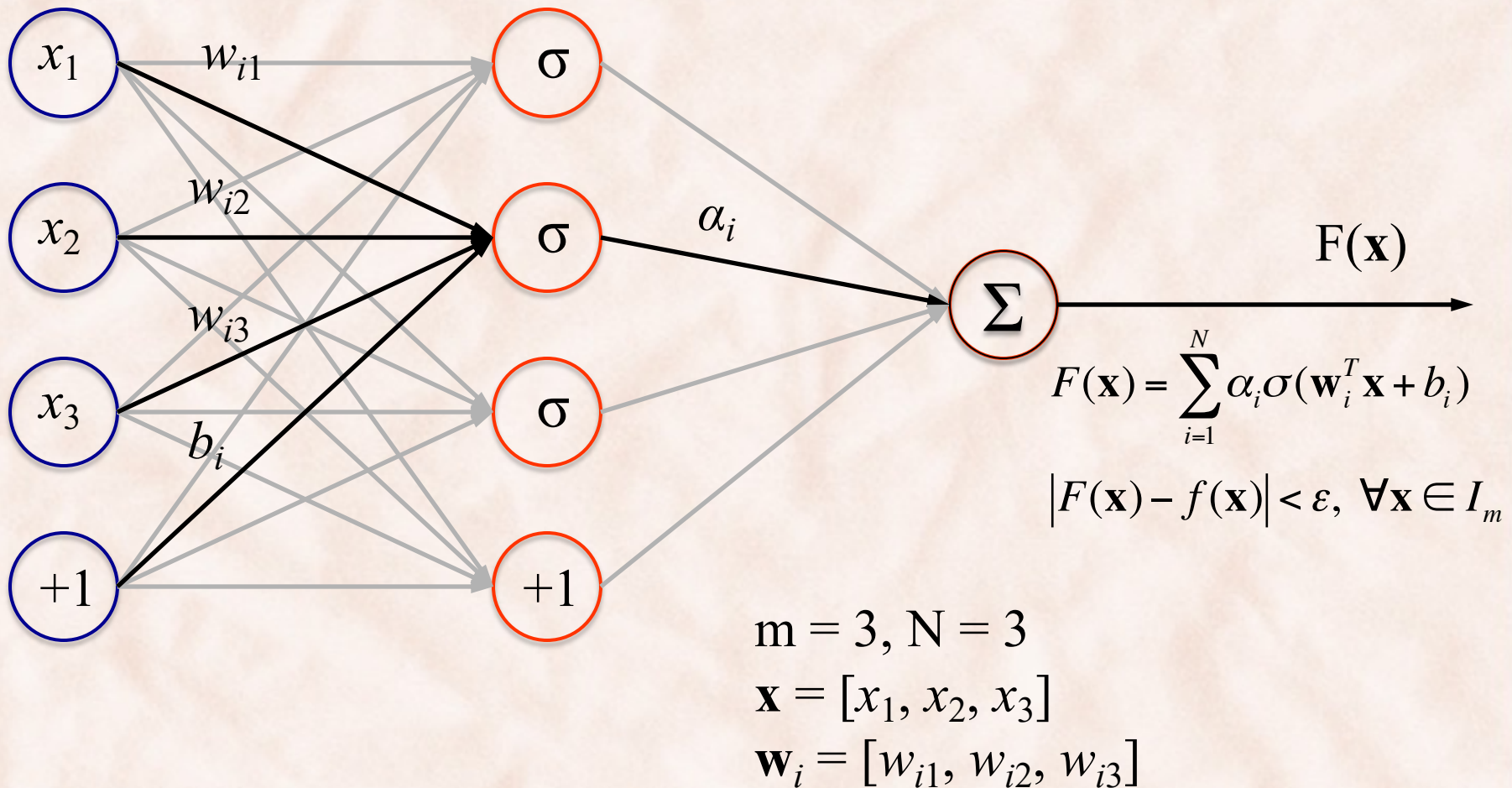
$$|F(\mathbf{x}) - f(\mathbf{x})| < \varepsilon, \quad \forall \mathbf{x} \in I_m$$

where

$$F(\mathbf{x}) = \sum_{i=1}^N \alpha_i \sigma(\mathbf{w}_i^T \mathbf{x} + b_i)$$

# Universal Approximation Theorem

Hornik (1991), Cybenko (1989)



# Polynomials as Simple NNs

[Lin & Tegmark, 2016]

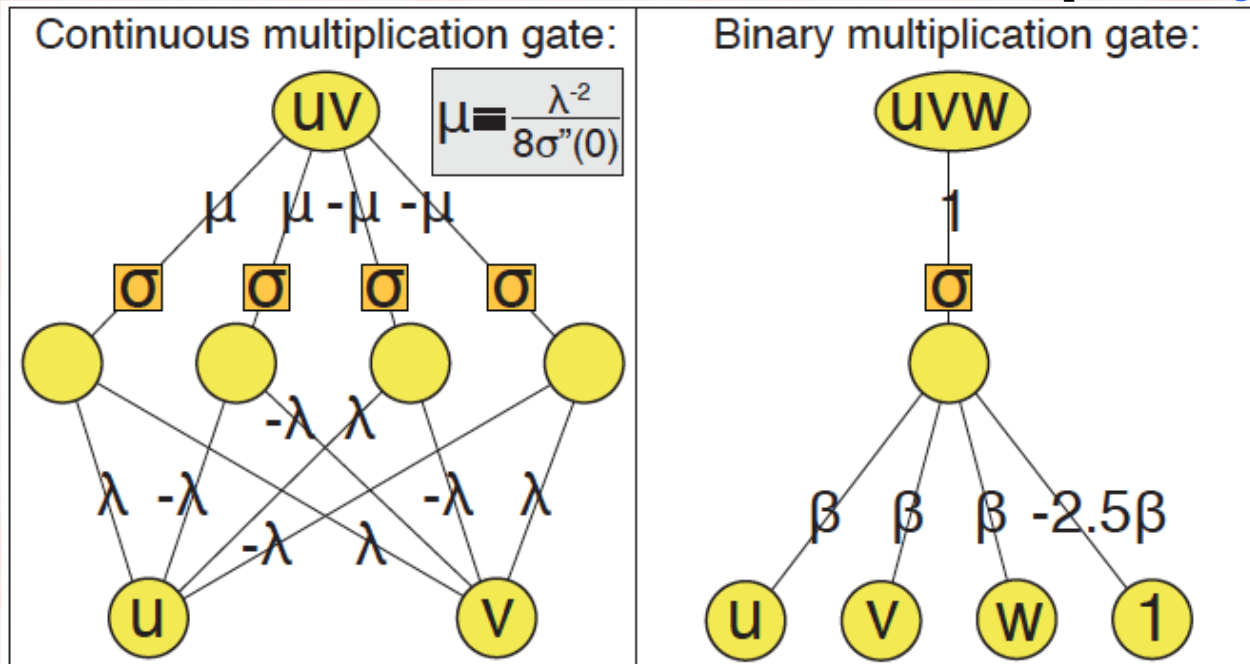
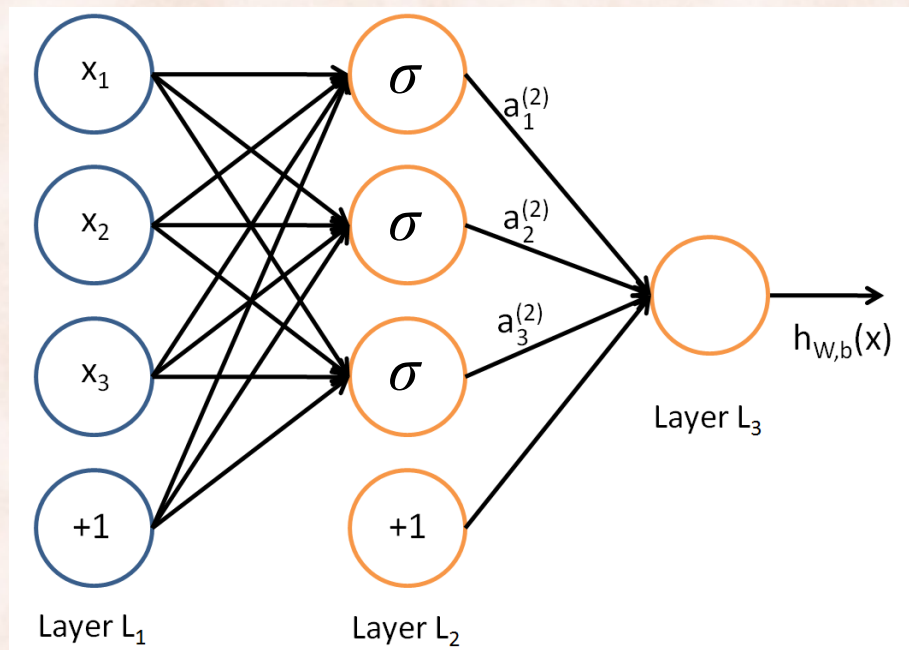


FIG. 2: Multiplication can be efficiently implemented by simple neural nets, becoming arbitrarily accurate as  $\lambda \rightarrow 0$  (left) and  $\beta \rightarrow \infty$  (right). Squares apply the function  $\sigma$ , circles perform summation, and lines multiply by the constants labeling them. The “1” input implements the bias term. The left gate requires  $\sigma''(0) \neq 0$ , which can always be arranged by biasing the input to  $\sigma$ . The right gate requires the sigmoidal behavior  $\sigma(x) \rightarrow 0$  and  $\sigma(x) \rightarrow 1$  as  $x \rightarrow -\infty$  and  $x \rightarrow \infty$ ,

# Neural Network Model

- Put together many neurons in layers, such that the output of a neuron on layer  $l$  can be the input of another neuron on layer  $l + 1$ :

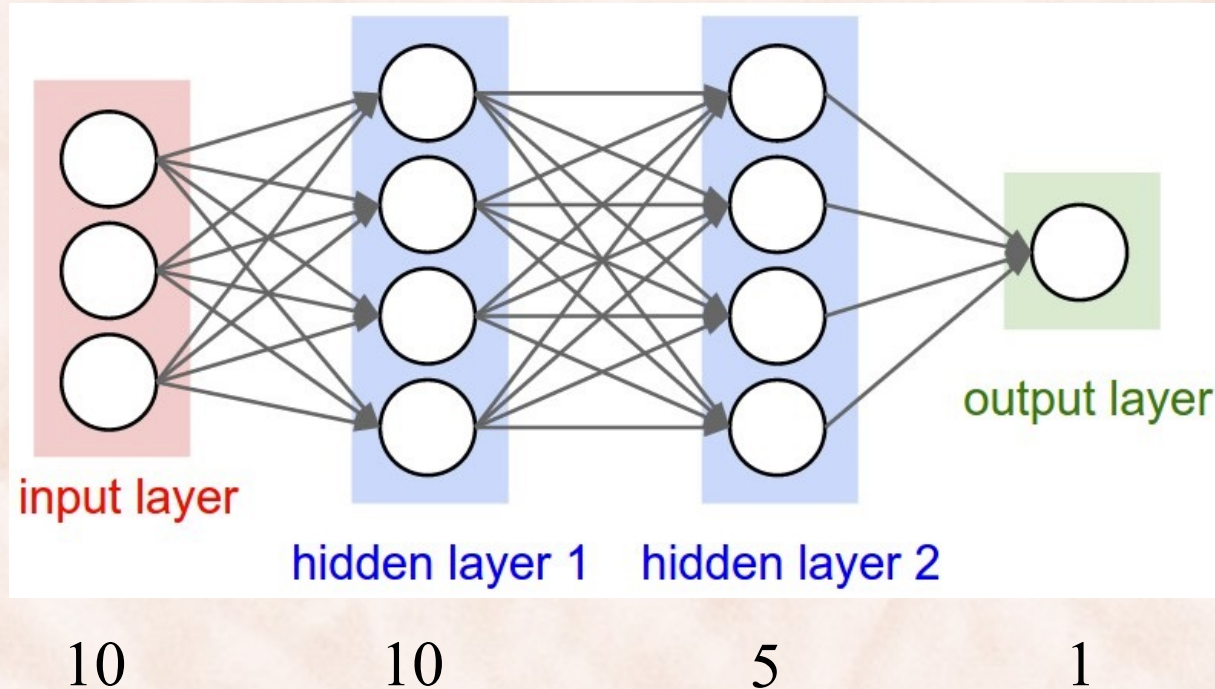


*input layer*

*hidden layer*

*output layer*

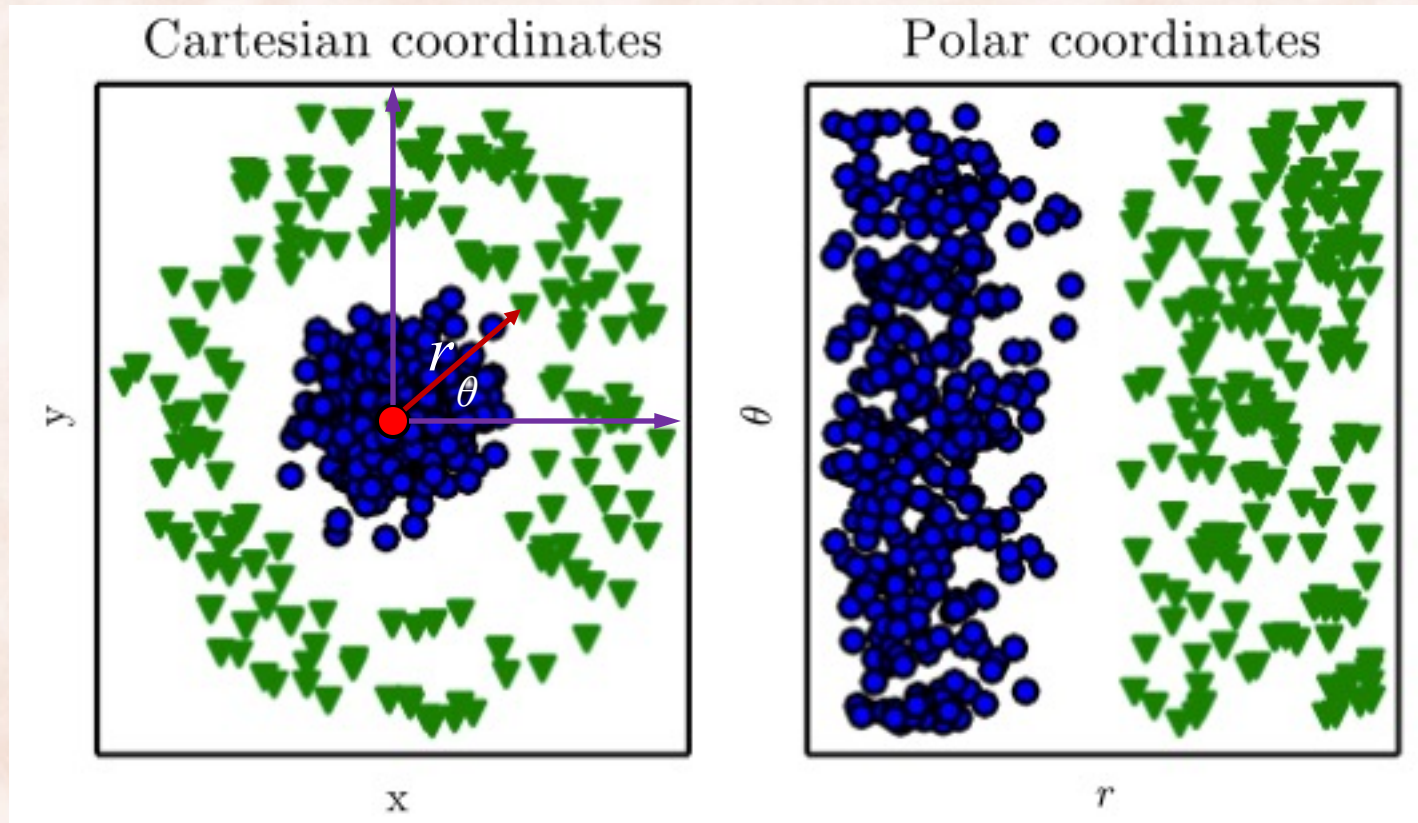
# Feed-Forward Neural Networks



1. For each neuron in hidden layer 1, we need  $10 + 1 = 11$  params. For the 10 neurons on hidden layer 1, we need in total  $10 * 11 = \mathbf{110}$  params.
2. For the 5 neurons on hidden layer 2, we need  $5 * 11 = \mathbf{55}$  params.
3. For the output neurons, we need  $5 + 1 = \mathbf{6}$  params.

# The Importance of Representation

<http://www.deeplearningbook.org>



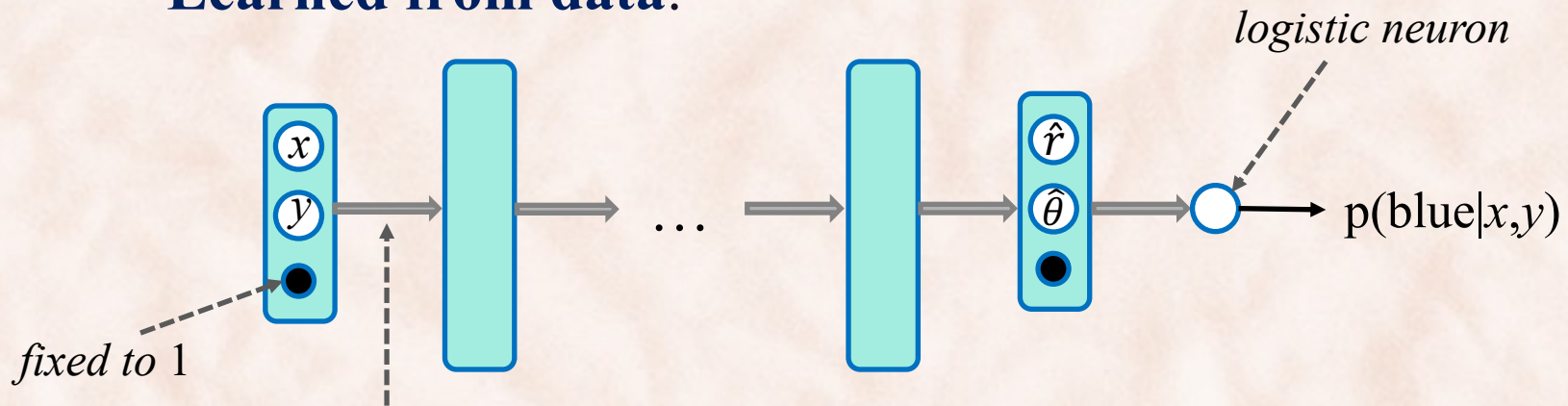
# From Cartesian to Polar Coordinates

- **Manually engineered:**

$$r = \sqrt{x^2 + y^2}$$

$$\theta = \tan^{-1} \left| \frac{y}{x} \right| \text{ (first quadrant)}$$

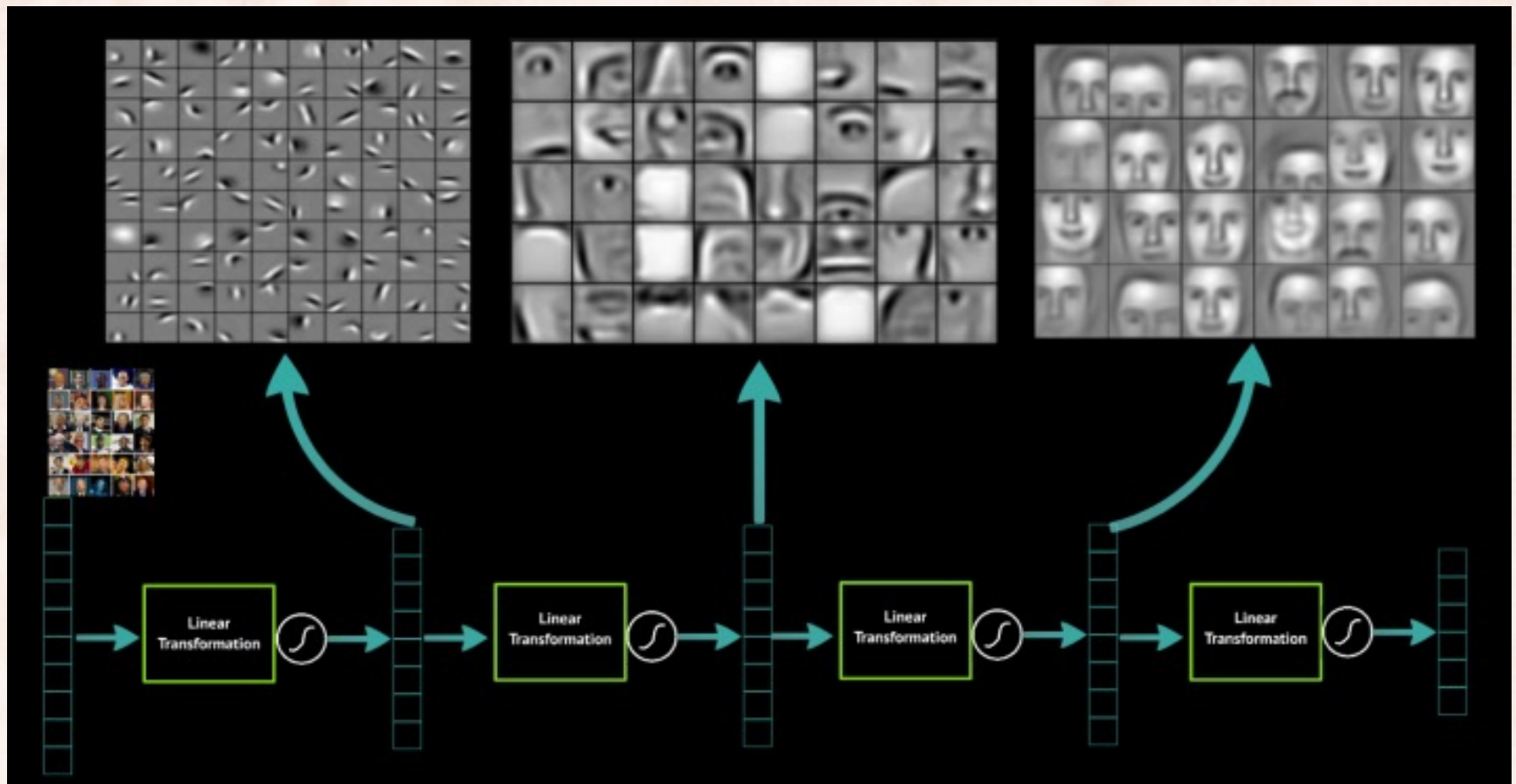
- **Learned from data:**



*Fully connected layers: linear transformation  $W$  + element-wise nonlinearity  $f \Rightarrow f(W\mathbf{x})$*

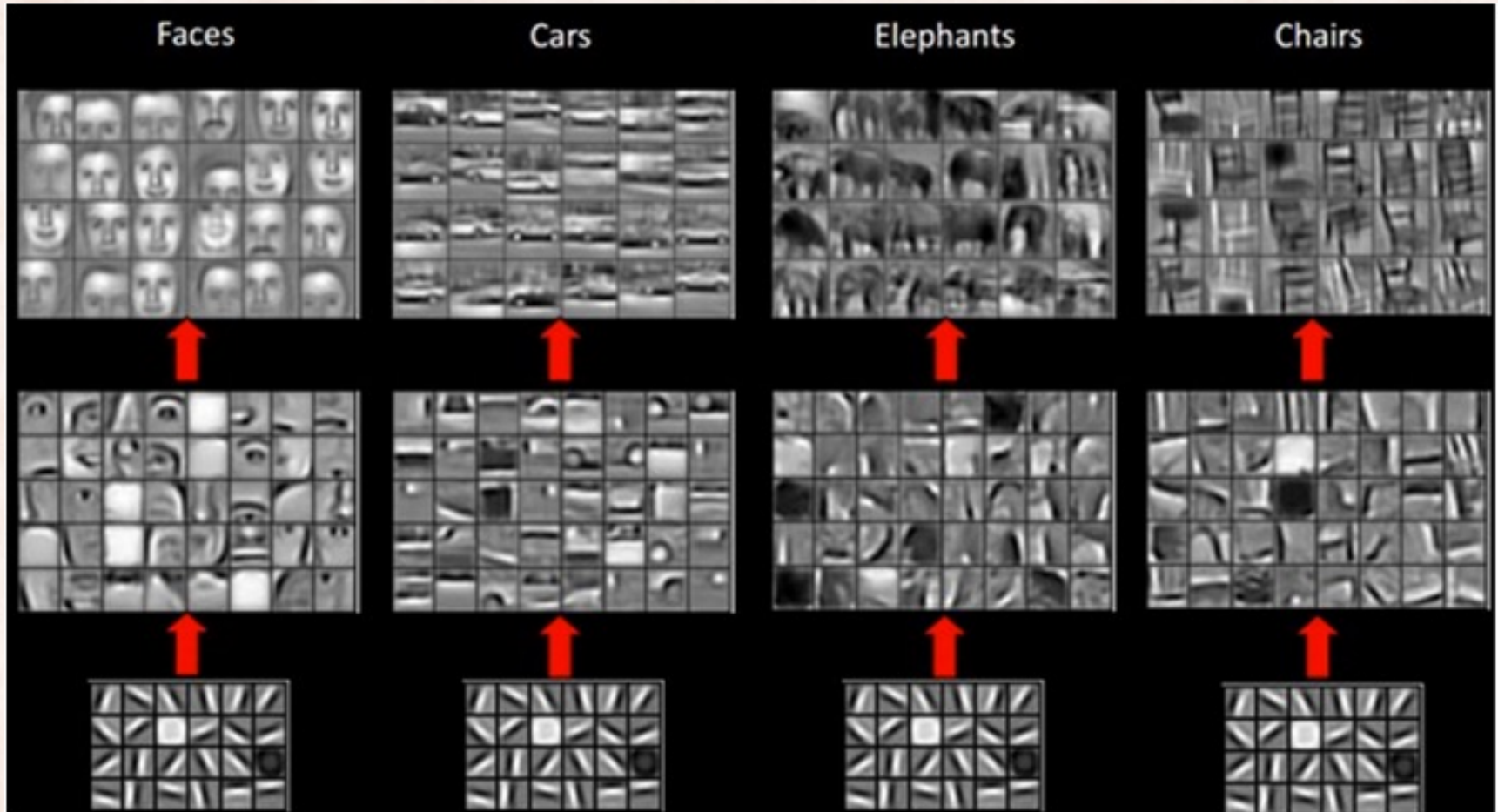
# Representation Learning: Images

<https://www.datarobot.com/blog/a-primer-on-deep-learning/>



# Representation Learning: Images

<https://www.datarobot.com/blog/a-primer-on-deep-learning/>



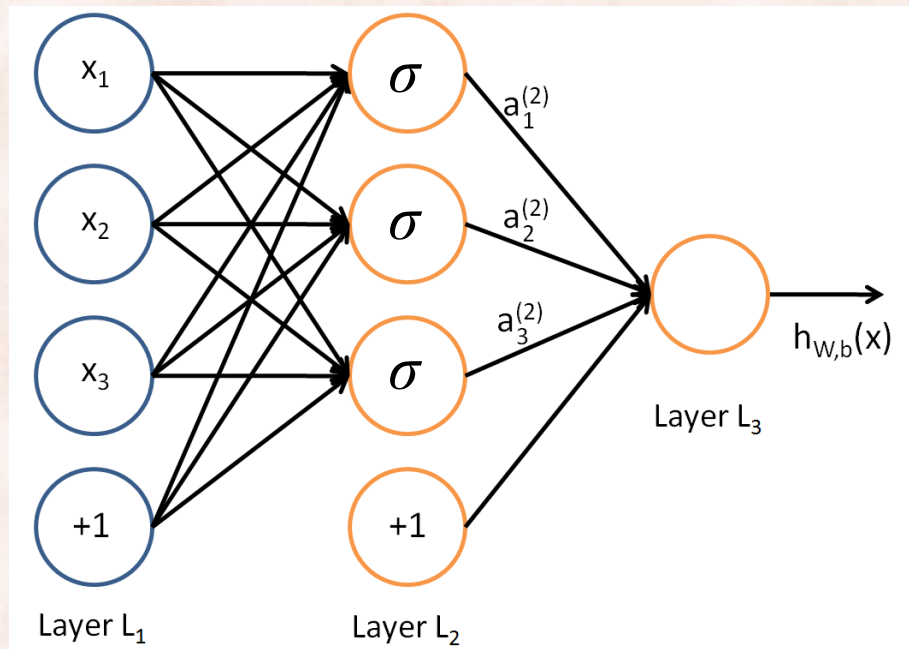
# A Rapidly Evolving Field

---

- Training deep networks used to require **greedy layer-wise pretraining**:
  - Unsupervised learning of representations with **auto-encoders** (2012).
- Better random **weight initialization** schemes now allow training deep networks from scratch.
- **Batch normalization** allows for training even deeper models (2014).
  - Replaced in Transformer by the simpler **Layer Normalization** (2016).
- **Residual learning** allows training arbitrarily deep networks (2015).
- Attention-based **Transformers** replace RNNs and CNNs in NLP (2018):
  - **BERT**: Pre-training of Deep Bidirectional Transformers for Language Understanding (2019).
  - RL with [verifiable rewards](#) lead to huge leaps in coding and math performance:
    - **Gemini Deep Think** obtains gold medal in IMO (2025).
    - **Claude 4.5 Opus, Gemini 3 Pro, GPT-5.2** performance in [software engineering tasks](#).

# Neural Network Model

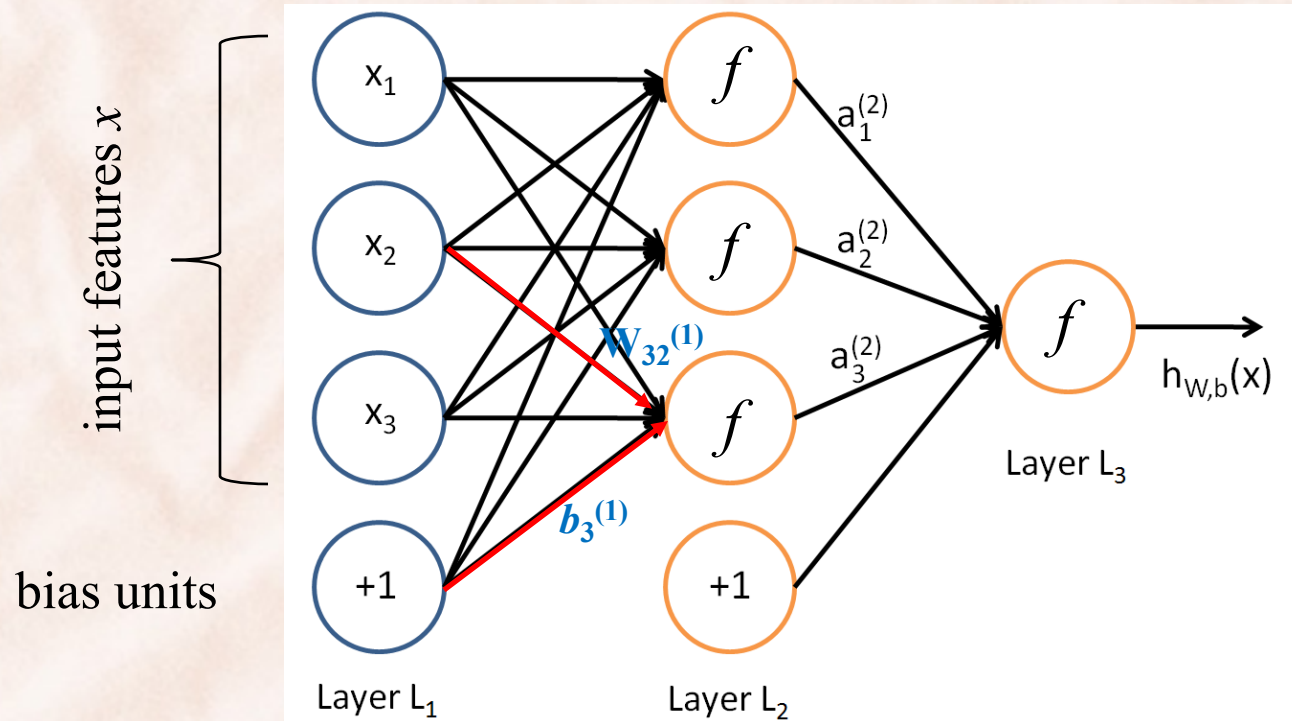
- Put together many neurons in layers, such that the output of a neuron can be the input of another:



*input layer*

*hidden layer*

*output layer*



- $n_l = 3$  is the number of **layers**.
  - L<sub>1</sub> is the input layer, L<sub>3</sub> is the output layer
- $(\mathbf{W}, \mathbf{b}) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$  are the parameters:
  - $W^{(l)}_{ij}$  is the **weight** of the connection between unit  $j$  in layer  $l$  and unit  $i$  in layer  $l + 1$ .
  - $b^{(l)}_i$  is the **bias** associated unit unit  $i$  in layer  $l + 1$ .
- $a^{(l)}_i$  is the **activation** of unit  $i$  in layer  $l$ , e.g.  $a^{(1)}_i = x_i$  and  $a^{(3)}_1 = h_{W,b}(x)$ .

# Inference: Forward Propagation

---

- The activations in the hidden layer are:

$$a_1^{(2)} = f(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + W_{13}^{(1)} x_3 + b_1^{(1)})$$

$$a_2^{(2)} = f(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{23}^{(1)} x_3 + b_2^{(1)})$$

$$a_3^{(2)} = f(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_3^{(1)})$$

- The activations in the output layer are:

$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)})$$

- Compressed notation:

$$a_i^{(l)} = f(z_i^{(l)}) \text{ where } z_i^{(l)} = \sum_{j=1}^n W_{ij}^{(l)} x_j + b_i^{(l)}$$

# Forward Propagation

---

- Forward propagation (unrolled):

$$a_1^{(2)} = f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)})$$

$$a_2^{(2)} = f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)})$$

$$a_3^{(2)} = f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)})$$

$$h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})$$

- Forward propagation (compressed):

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$h_{W,b}(x) = a^{(3)} = f(z^{(3)})$$

- Element-wise application:

$$f(\mathbf{z}) = [f(z_1), f(z_2), f(z_3)]$$

# Forward Propagation

---

- Forward propagation (compressed):

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$

$$h_{W,b}(x) = a^{(3)} = f(z^{(3)})$$

- Composed of two *forward propagation steps*:

$$z^{(l+1)} = W^{(l)}a^{(l)} + b^{(l)}$$

$$a^{(l+1)} = f(z^{(l+1)})$$

# Forward Propagation for FCNs: Regression

---

1. Input activations are  $\mathbf{a}^{(1)} = \mathbf{x}$

2. For each layer  $l = 1, 2, \dots, n_l - 1$  compute  $\mathbf{a}^{(l+1)}$

$$\mathbf{z}^{(l+1)} = W^{(l)} \mathbf{a}^{(l)} + \mathbf{b}^{(l)} \quad \textit{matrix multiply and add}$$

$$\mathbf{a}^{(l+1)} = f(\mathbf{z}^{(l+1)}) \quad \textit{apply element-wise non-linear function } f$$

3. For last layer  $n_l + 1$  compute regression output  $\mathbf{a}^{(n_l+1)}$

$$\mathbf{z}^{(n_l+1)} = W^{(n_l)} \mathbf{a}^{(n_l)} + \mathbf{b}^{(n_l)}$$

$$\mathbf{a}^{(n_l+1)} = \mathbf{z}^{(n_l+1)} \quad \textit{output (regression)}$$

# Backpropagation for FCNs for Regression: 1 example

- Feedforward to compute activations  $a^{(l)} = f(\mathbf{z}^{(l)})$  at layers  $l$

1. For output layer, compute:

$$\delta^{(n_l+1)} = (a^{(n_l+1)} - y) \quad \text{true label}$$

2. For  $l = n_l, n_l-2, n_l-3, \dots, 2$  compute:

$$\delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \cdot f'(z^{(l)})$$

3. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} (a^{(l)})^T \quad \nabla_{b^{(l)}} J = \delta^{(l+1)}$$

# Backpropagation for FCNs for Regression: *m* examples

- Feedforward to compute activations  $\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$  at layers  $l$

1. For output layer, compute:

$$\delta^{(n_l+1)} = (\mathbf{a}^{(n_l+1)} - \mathbf{y}) \quad \text{true label vector}$$

2. For  $l = n_l, n_l-2, n_l-3, \dots, 2$  compute:

$$\delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \cdot f'(z^{(l)})$$

3. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} (a^{(l)})^T / m \quad \nabla_{b^{(l)}} J = \delta^{(l+1)}.col\_avg()$$

# Multinomial Softmax

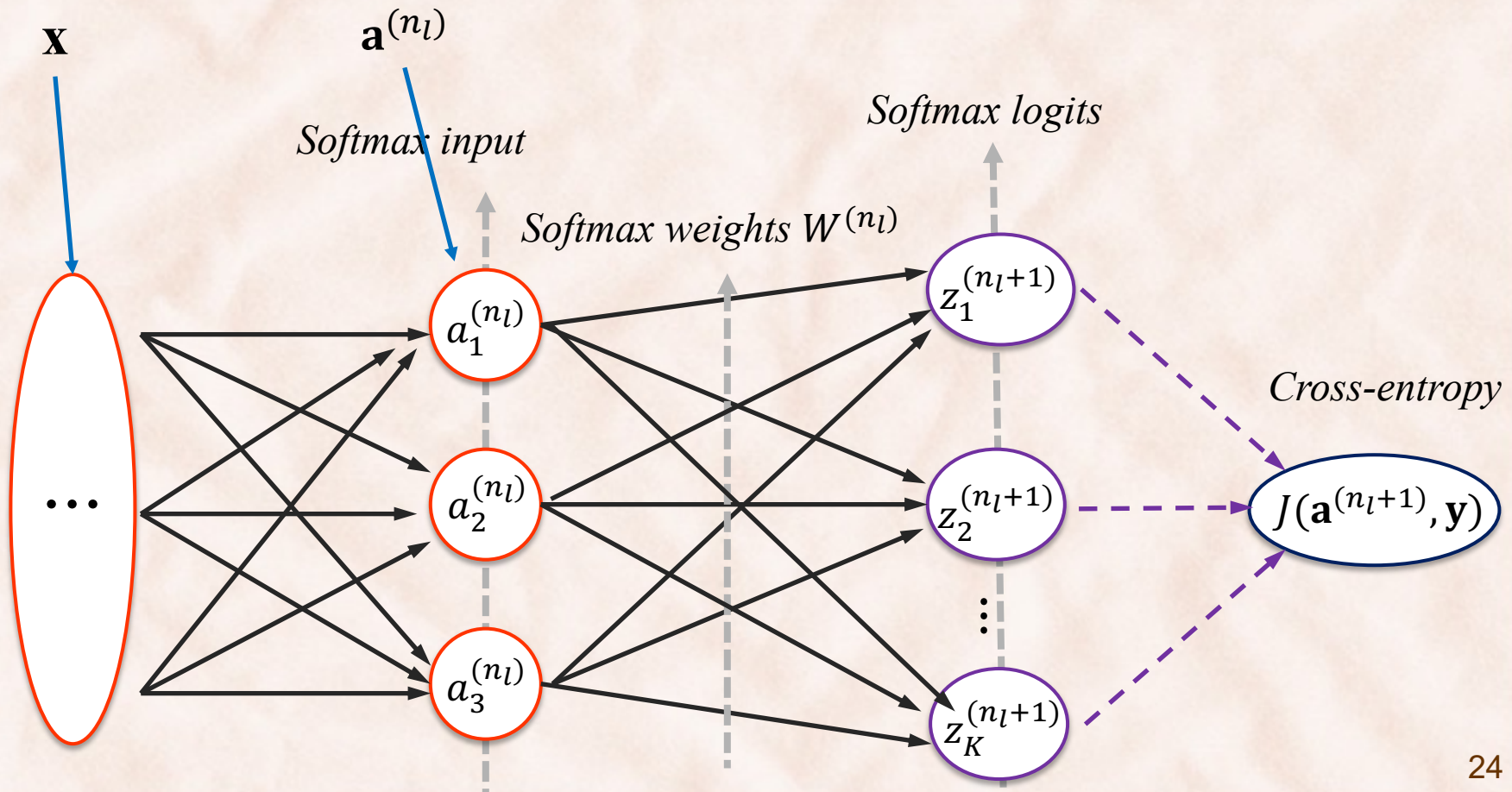
---

- Consider layer  $n_l$  to be the input to the softmax layer i.e. softmax output layer is  $n_l+1$ .
- Softmax weights stored in matrix  $W^{(n_l)}$ .

- K classes  $\Rightarrow W^{(n_l)} = \begin{bmatrix} -\mathbf{w}_1^T & - \\ -\mathbf{w}_2^T & - \\ \vdots & \\ -\mathbf{w}_K^T & - \end{bmatrix}$

# Multinomial Softmax

- Softmax output is  $\mathbf{a}^{(n_{l+1})} = \text{softmax}(\mathbf{z}^{(n_{l+1})})$



# Forward Propagation for FCNs: Classification

---

1. Input activations are  $\mathbf{a}^{(1)} = \mathbf{x}$

2. For each layer  $l = 1, 2, \dots, n_l - 1$  compute  $\mathbf{a}^{(l+1)}$

$$\mathbf{z}^{(l+1)} = W^{(l)} \mathbf{a}^{(l)} + \mathbf{b}^{(l)} \quad \textit{matrix multiply and add}$$

$$\mathbf{a}^{(l+1)} = f(\mathbf{z}^{(l+1)}) \quad \textit{apply element-wise non-linear function } f$$

3. For last layer  $n_l + 1$  compute probability output  $\mathbf{a}^{(n_l+1)}$

$$\mathbf{z}^{(n_l+1)} = W^{(n_l)} \mathbf{a}^{(n_l)} + \mathbf{b}^{(n_l)}$$

$$\mathbf{a}^{(n_l+1)} = \text{softmax}(\mathbf{z}^{(n_l+1)}) \quad \textit{softmax output (classification)}$$

# Backpropagation Algorithm: Softmax (1)

---

1. Feedforward pass on  $\mathbf{x}$  to compute activations  $\mathbf{a}^{(l)}$  for layers  $l = 1, 2, \dots, n_l$ .
2. Compute softmax outputs  $\mathbf{a}^{(n_l+1)}$  and objective  $J(\mathbf{a}^{(n_l+1)}, \mathbf{y})$ .
3. Let  $\mathbf{y} = [\delta_1(y), \delta_2(y), \dots, \delta_K(y)]^T$  be the one-hot vector representation for label  $y$ .
4. Compute gradient with respect to softmax weights:

$$\frac{\partial J}{\partial W^{(n_l)}} = (\mathbf{a}^{(n_l+1)} - \mathbf{y})\mathbf{a}^{(n_l)T}$$

## Backpropagation Algorithm: Softmax (2)

---

5. Compute gradient with respect to softmax inputs:

$$\delta^{(n_l)} = \underbrace{\left(W^{(n_l)}\right)^T \left(\mathbf{a}^{(n_{l+1})} - \mathbf{y}\right)}_{\frac{\partial J}{\partial \mathbf{a}^{(n_l)}}} \circ f'(\mathbf{z}^{(n_l)})$$

6. For  $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$  compute:

$$\delta^{(l)} = \left( \left(W^{(l)}\right)^T \delta^{(l+1)} \right) \bullet f'(\mathbf{z}^{(l)})$$

7. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left(a^{(l)}\right)^T \quad \nabla_{b^{(l)}} J = \delta^{(l+1)}$$

# Backpropagation Algorithm: Softmax for 1 Example

- Feedforward to compute activations  $\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$  at all layers

1. For softmax layer, compute:

$$\delta^{(n_l+1)} = (\mathbf{a}^{(n_l+1)} - \mathbf{y}) \quad \text{one-hot label vector}$$

2. For  $l = n_l, n_l-1, n_l-2, \dots, 2$  compute:

$$\delta^{(l)} = \left( \left( W^{(l)} \right)^T \delta^{(l+1)} \right) \cdot f'(z^{(l)})$$

3. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left( \mathbf{a}^{(l)} \right)^T \quad \nabla_{b^{(l)}} J = \delta^{(l+1)}$$

# Backpropagation Algorithm: Softmax for Dataset of $m$ Examples

- Feedforward to compute activations  $\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$  at all layers

1. For softmax layer, compute:

$$\delta^{(n_l+1)} = (\mathbf{a}^{(n_l+1)} - \mathbf{y})$$

*ground-truth label matrix*

2. For  $l = n_l, n_l-1, n_l-2, \dots, 2$  compute:

$$\delta^{(l)} = \left( \left( W^{(l)} \right)^T \delta^{(l+1)} \right) \bullet f'(z^{(l)})$$

3. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left( a^{(l)} \right)^T / m$$

$$+ \alpha W^{(l)}$$

$$\nabla_{b^{(l)}} J = \delta^{(l+1)}.col\_avg()$$

*if using  $L_2$  regularization*

# Backpropagation Algorithm: Softmax for Dataset of $m$ Examples

- Feedforward to compute activations  $\mathbf{a}^{(l)} = f(\mathbf{z}^{(l)})$  at all layers

1. For softmax layer, compute:

$$\delta^{(n_l+1)} = (\mathbf{a}^{(n_l+1)} - \mathbf{y})$$

$K \times m$ , where  $K$  is the # of classes

2. For  $l = n_l, n_l - 1, n_l - 2, \dots, 2$  compute:

$$\delta^{(l)} = \left( \left( W^{(l)} \right)^T \delta^{(l+1)} \right) \cdot f'(z^{(l)})$$

$s_l \times m$ , where  $s_l$  is the # neurons on layer  $l$

3. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left( \mathbf{a}^{(l)} \right)^T / m + \alpha W^{(l)}$$

$s_{l+1} \times s_l$ , where  $s_l$  is the # neurons on layer  $l$

$$\nabla_{b^{(l)}} J = \delta^{(l+1)} \cdot \text{col\_avg}()$$

`np.mean(axis = 1)`

$s_{l+1} \times 1$ , where  $s_{l+1}$  is the # neurons on layer  $l+1$

# Softmax Regression Cost: From 1 to $m$ examples

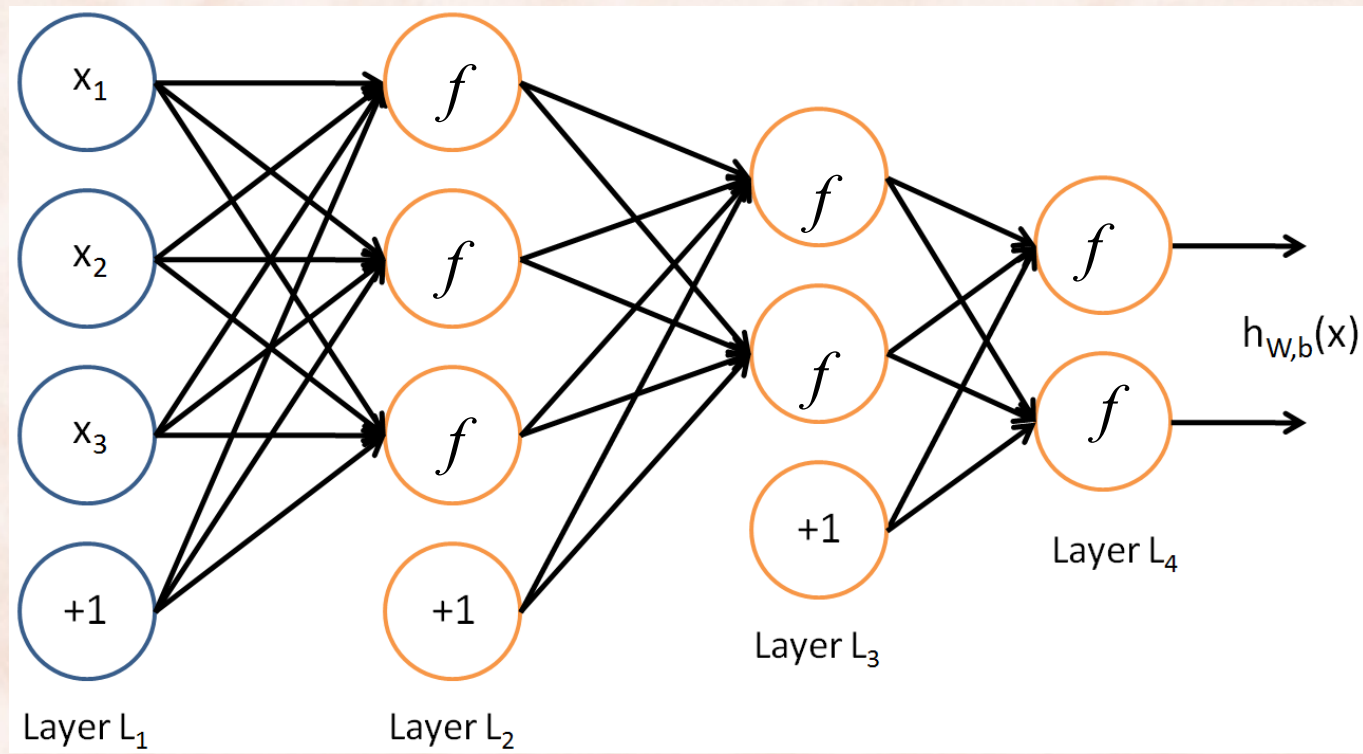
---

- Ground truth vector  $\mathbf{y}$  is a one-hot vector where:
  - $y_k = 1$  if the true class label  $y$  is  $k$ , otherwise  $y_k = 0$ .
- The negative log-likelihood (NLL) part of the cost is:
  - $J(W, b, x, y) = -\ln p(y|W, b, x) = -\sum_{k=1}^K \delta_k(y) \ln p(C_k|x)$
- Using our NN notation,  $y_k = \delta_k(y)$  and  $a_k^{(n_l+1)} = p(C_k|x)$ 
  - Therefore, we can write the NLL part of the cost as a dot-product between the one-hot ground truth vector  $\mathbf{y}$  and the log of  $\mathbf{a}^{(n_l+1)}$ 
    - $J(W, b, x, y) = J(\mathbf{a}^{(n_l+1)}, \mathbf{y}) = -\mathbf{y}^T \ln \mathbf{a}^{(n_l+1)} = -\text{sum}(\mathbf{y} \circ \ln \mathbf{a}^{(n_l+1)})$
- When vectorized for  $m$  examples + regularization, when  $\mathbf{y}$  is the ground-truth matrix and  $\mathbf{a}$  is the matrix of softmax probabilities of all  $m$  examples:

$$\text{▪ } J(W, b) = J(\mathbf{a}^{(n_l+1)}, \mathbf{y}) = -\frac{1}{m} \text{sum}(\mathbf{y} \circ \ln \mathbf{a}^{(n_l+1)}) + \frac{\alpha}{2} \|W\|^2$$

# Multiple Hidden Units, Multiple Outputs

- Write down the forward propagation steps for:



# Start Supplemental Material

---

## **Derivation of Backpropagation**

# Backpropagation for FCNs for Regression: 1 example

- Feedforward to compute activations  $a^{(l)} = f(\mathbf{z}^{(l)})$  at layers  $l$

1. For softmax layer, compute:

$$\delta^{(n_l+1)} = (a^{(n_l+1)} - y) \quad \text{true label}$$

2. For  $l = n_l, n_l-1, n_l-2, \dots, 2$  compute:

$$\delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \cdot f'(z^{(l)})$$

3. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} (a^{(l)})^T \quad \nabla_{b^{(l)}} J = \delta^{(l+1)}$$

# Learning: Regression vs. Classification

---

- **Regression**  $\Rightarrow$   $loss$  = squared error:

$$J(W, b, x, y) = \frac{1}{2} \|h_{W,b}(x) - y\|^2 + \frac{\lambda}{2} \|W\|^2$$

- **Classification**  $\Rightarrow$   $loss$  = negative log-likelihood:

$$J(W, b, x, y) = -\ln p(y|W, b, x) + \frac{\lambda}{2} \|W\|^2$$

- Need to compute the gradient of the loss with respect to parameters  $W, b$ :

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = ?$$

$$\frac{\partial J}{\partial b_i^{(l)}} = ?$$

# Learning: Backpropagation for Regression

- Regularized sum of squares error:

$$J(W, b, x, y) = \frac{1}{2} \|h_{W,b}(x) - y\|^2$$

$$J(W, b) = \frac{1}{m} \sum_{k=1}^m J(W, b, x^{(k)}, y^{(k)}) + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_{l+1}} \sum_{j=1}^{s_l} (W_{ij}^{(l)})^2$$

Squared Frobenius norm of  $W^{(l)}$

- Gradient: ?

$$\frac{\partial J(W, b)}{\partial W_{ij}^{(l)}} = \frac{1}{m} \sum_{k=1}^m \frac{\partial J(W, b, x^{(k)}, y^{(k)})}{\partial W_{ij}^{(l)}} + \lambda W_{ij}^{(l)}$$

$$\frac{\partial J(W, b)}{\partial b_i^{(l)}} = \frac{1}{m} \sum_{k=1}^m \frac{\partial J(W, b, x^{(k)}, y^{(k)})}{\partial b_i^{(l)}}$$

# Backpropagation for Regression

---

- Need to compute the gradient of the squared error with respect to a single training example  $(x, y)$ :

$$J(W, b, x, y) = \frac{1}{2} \|h_{W,b}(x) - y\|^2 = \frac{1}{2} \|a^{(n_l)} - y\|^2$$

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = ?$$

$$\frac{\partial J}{\partial b_i^{(l)}} = ?$$

# Univariate Chain Rule for Differentiation

---

- Univariate Chain Rule:

$$f = f \circ g \circ h = f(g(h(x)))$$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial h} \frac{\partial h}{\partial x}$$

- Example:

$$f(g(x)) = 2g(x)^2 - 3g(x) + 1$$

$$g(x) = x^3 + 2x$$

# Multivariate Chain Rule for Differentiation

---

- Multivariate Chain Rule:

$$f = f(g_1(x), g_2(x), \dots, g_n(x))$$

$$\frac{\partial f}{\partial x} = \sum_{i=1}^n \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x}$$

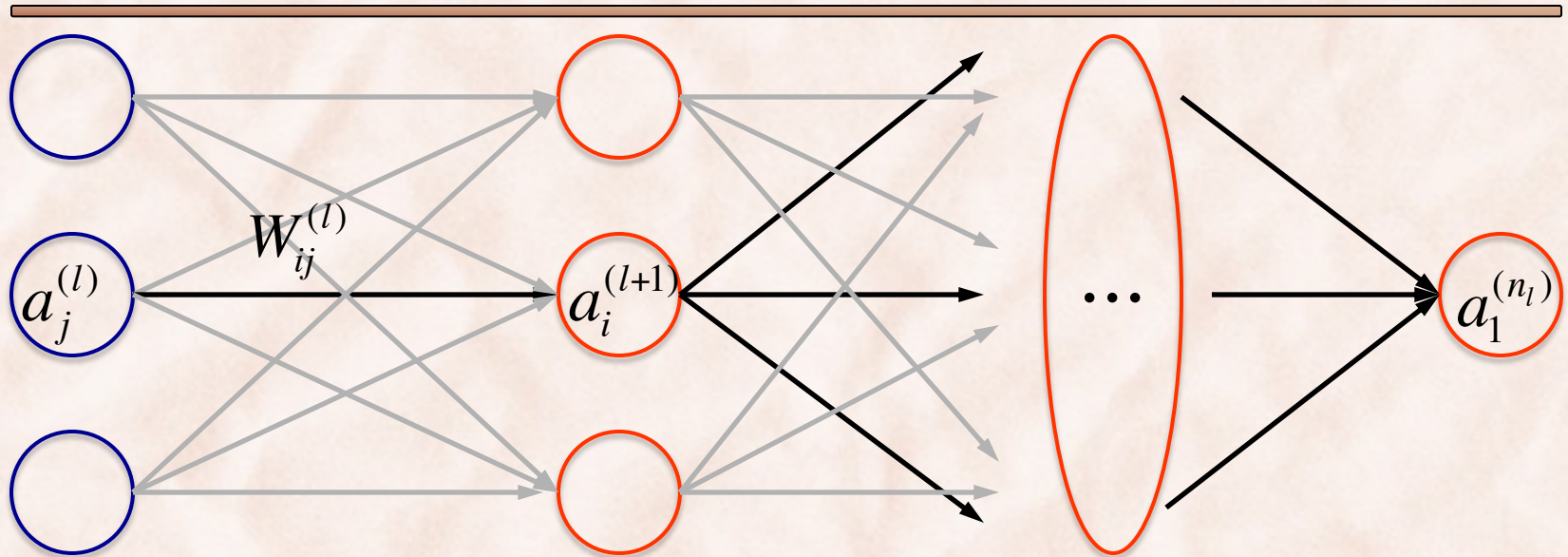
- Example:

$$f(g_1(x), g_2(x)) = 2g_1(x)^2 - 3g_1(x)g_2(x) + 1$$

$$g_1(x) = 3x$$

$$g_2(x) = x^2 + 2x$$

# Backpropagation: $W_{ij}^{(l)}$

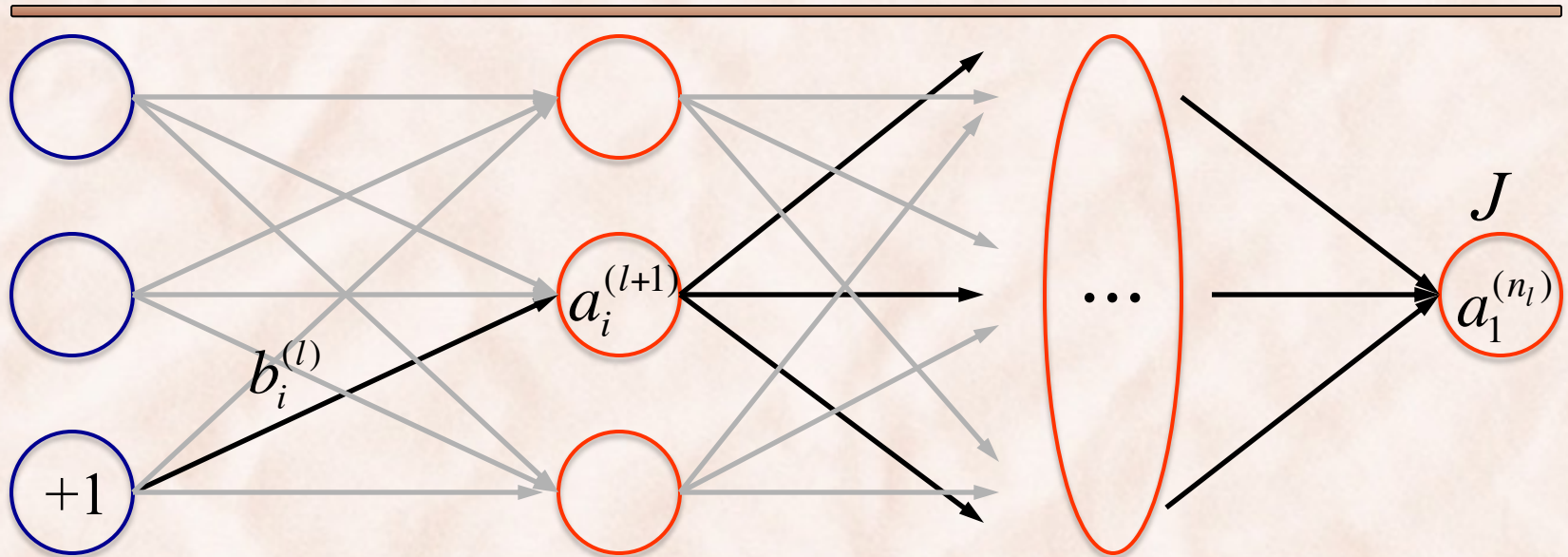


- $J$  depends on  $W_{ij}^{(l)}$  only through  $a_i^{(l+1)}$ , which depends on  $W_{ij}^{(l)}$  only through  $z_i^{(l+1)}$ .

$$J(W, b, x, y) = \frac{1}{2} \|a^{(n_l)} - y\|^2$$

$$a_i^{(l+1)} = f(z_i^{(l+1)})$$
$$z_i^{(l+1)} = \sum_{j=1}^{s_l} W_{ij}^{(l)} a_j^{(l)} + b_i^{(l)}$$

# Backpropagation: $b_i^{(l)}$



- $J$  depends on  $b_i^{(l)}$  only through  $a_i^{(l+1)}$ , which depends on  $b_i^{(l)}$  only through  $z_i^{(l+1)}$ .

$$J(W, b, x, y) = \frac{1}{2} \|a^{(n_l)} - y\|^2$$

$$a_i^{(l+1)} = f(z_i^{(l+1)})$$

$$z_i^{(l+1)} = \sum_{j=1}^{s_l} W_{ij}^{(l)} a_j^{(l)} + b_i^{(l)}$$

## Backpropagation: $W_{ij}^{(l)}$ and $b_i^{(l)}$

---

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = \underbrace{\frac{\partial J}{\partial a_i^{(l+1)}} \times \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}}}_{\delta_i^{(l+1)}} \times \underbrace{\frac{\partial z_i^{(l+1)}}{\partial W_{ij}^{(l)}}}_{a_j^{(l)}} = a_j^{(l)} \delta_i^{(l+1)}$$

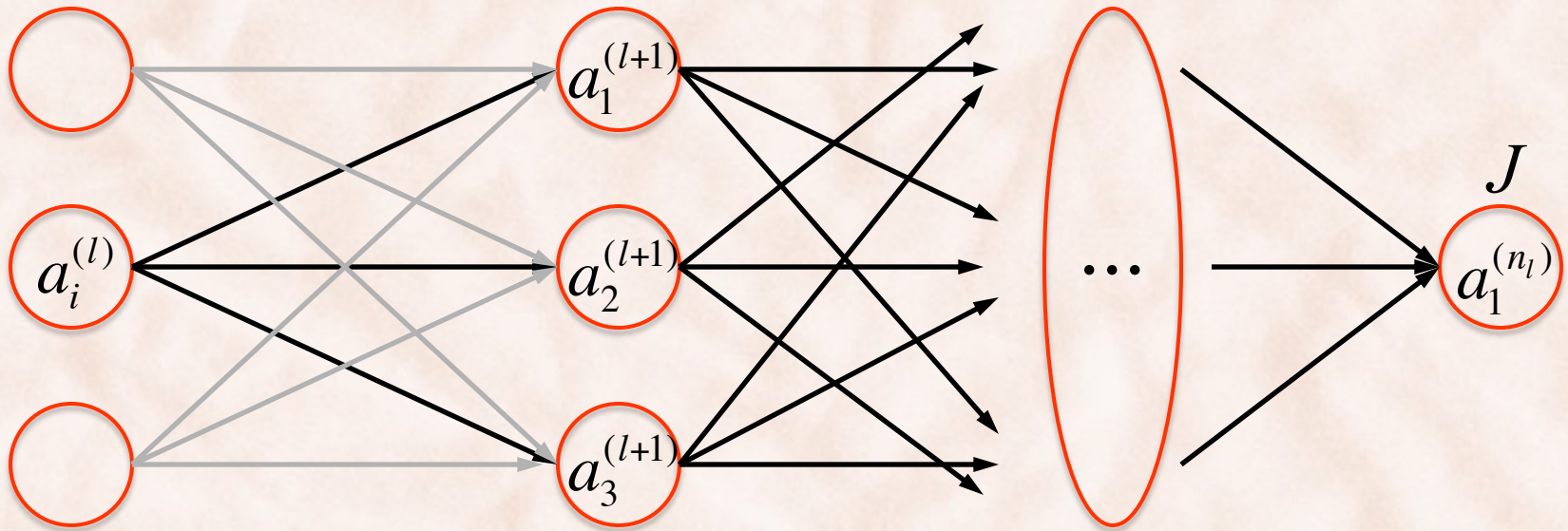
*How to compute  $\delta_i^{(l)}$   
for all layers  $l$ ?*

$$\frac{\partial J}{\partial b_i^{(l)}} = \underbrace{\frac{\partial J}{\partial a_i^{(l+1)}} \times \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}}}_{\delta_i^{(l+1)}} \times \underbrace{\frac{\partial z_i^{(l+1)}}{\partial b_i^{(l)}}}_{+1} = \delta_i^{(l+1)}$$

# Backpropagation: $\delta_i^{(l)}$

$$\delta_i^{(l)} = \frac{\partial J}{\partial a_i^{(l)}} \times \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = \frac{\partial J}{\partial a_i^{(l)}} \times f'(z_i^{(l)})$$

- $J$  depends on  $a_i^{(l)}$  only through  $a_1^{(l+1)}, a_2^{(l+1)}, \dots$



# Backpropagation: $\delta_i^{(l)}$

---

- $J$  depends on  $a_i^{(l)}$  only through  $a_1^{(l+1)}, a_2^{(l+1)}, \dots$

$$\frac{\partial J}{\partial a_i^{(l)}} = \sum_{j=1}^{s_{l+1}} \frac{\partial J}{\partial a_j^{(l+1)}} \times \frac{\partial a_j^{(l+1)}}{\partial a_i^{(l)}} = \sum_{j=1}^{s_{l+1}} \underbrace{\frac{\partial J}{\partial a_j^{(l+1)}} \times \frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}}}_{\delta_j^{(l+1)}} \times \underbrace{\frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}}}_{W_{ji}^{(l)}}$$

- Therefore,  $\delta_i^{(l)}$  can be computed as:

$$\delta_i^{(l)} = \frac{\partial J}{\partial a_i^{(l)}} \times f'(z_i^{(l)}) = \left( \sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) \times f'(z_i^{(l)})$$

# Backpropagation: $\delta_i^{(l)}$

---

- Start computing  $\delta$ 's for the output layer:

$$\delta_i^{(n_l)} = \frac{\partial J}{\partial a_i^{(n_l)}} \times \frac{\partial a_i^{(n_l)}}{\partial z_i^{(n_l)}} = \frac{\partial J}{\partial a_i^{(n_l)}} \times f'(z_i^{(n_l)})$$

$$J = \frac{1}{2} \|a^{(n_l)} - y\|^2 \Rightarrow \frac{\partial J}{\partial a_i^{(n_l)}} = (a_i^{(n_l)} - y_i)$$

---

$$\delta_i^{(n_l)} = (a_i^{(n_l)} - y_i) \times f'(z_i^{(n_l)})$$

# Backpropagation Algorithm

---

1. Feedforward pass on  $x$  to compute activations  $a_i^{(l)}$
2. For each output unit  $i$  compute:

$$\delta_i^{(n_l)} = \left( a_i^{(n_l)} - y_i \right) \times f'(z_i^{(n_l)})$$

3. For  $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$  compute:

$$\delta_i^{(l)} = \left( \sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) \times f'(z_i^{(l)})$$

4. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\frac{\partial J}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \quad \frac{\partial J}{\partial b_i^{(l)}} = \delta_i^{(l+1)}$$

# Backpropagation Algorithm: Vectorization for 1 Example

---

1. Feedforward pass on  $x$  to compute activations  $a_i^{(l)}$
2. For last layer compute:

$$\delta^{(n_l)} = \left( a^{(n_l)} - y \right) \cdot f'(z^{(n_l)})$$

3. For  $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$  compute:

$$\delta^{(l)} = \left( \left( W^{(l)} \right)^T \delta^{(l+1)} \right) \cdot f'(z^{(l)})$$

4. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left( a^{(l)} \right)^T \quad \nabla_{b^{(l)}} J = \delta^{(l+1)}$$

# Backpropagation Algorithm: Vectorization for Dataset of $m$ Examples

---

1. Feedforward pass on  $X$  to compute activations  $a_i^{(l)}$
2. For last layer compute:

$$\delta^{(n_l)} = (a^{(n_l)} - y) \cdot f'(z^{(n_l)})$$

3. For  $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$  compute:

$$\delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \cdot f'(z^{(l)})$$

4. Compute the partial derivatives of the cost  $J(W, b, x, y)$

$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left( a^{(l)} \right)^T / m \qquad \nabla_{b^{(l)}} J = \delta^{(l+1)}.col\_avg()$$

# End Supplemental Material

---

## **Derivation of Backpropagation**

# Important Architectural Techniques for NNs, e.g., Transformer

---

- **Activation** functions.
  - ReLU, SiLU, Swish, **GELU**, GLU, **SwiGLU**.
- **Normalization** of logits / activations:
  - Batch normalization.
  - **Layer normalization**.
- **Residual connections**.
  - Deep **residual networks**.
- **Deep architectures**.
  - **Compact** representation of **highly-varying** functions.

# Piecewise Linear Activation Functions

---

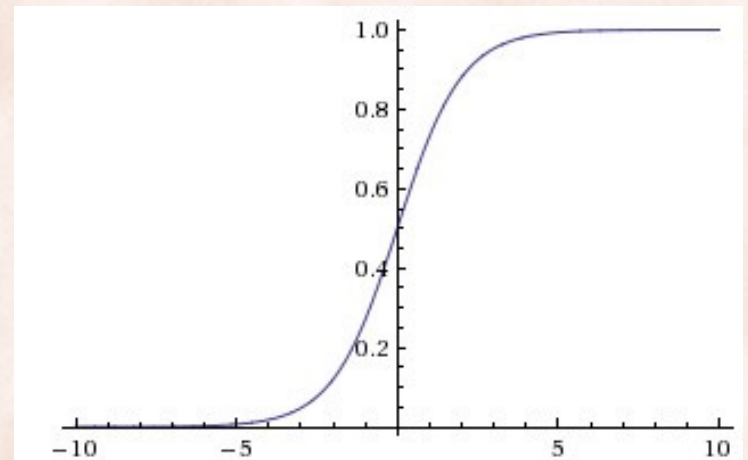
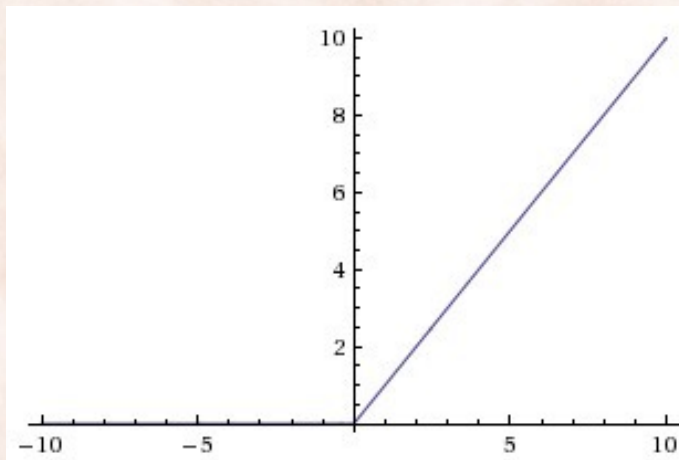
- **ReLU**: the rectifier activation  $g(z) = \max\{0, z\}$ .
- **Absolute value ReLU**:  $g(z) = |z|$ .
- **Leaky ReLU**:  $g(a) = \max\{0, a\} + \alpha \min(0, a)$ .
- **Maxout**:  $g(a_1, \dots, a_k) = \max\{a_1, \dots, a_k\}$ .
  - needs  $k$  weight vectors instead of 1.

⇒ the network computes a *piecewise linear function*.

# ReLU vs. Sigmoid and Tanh

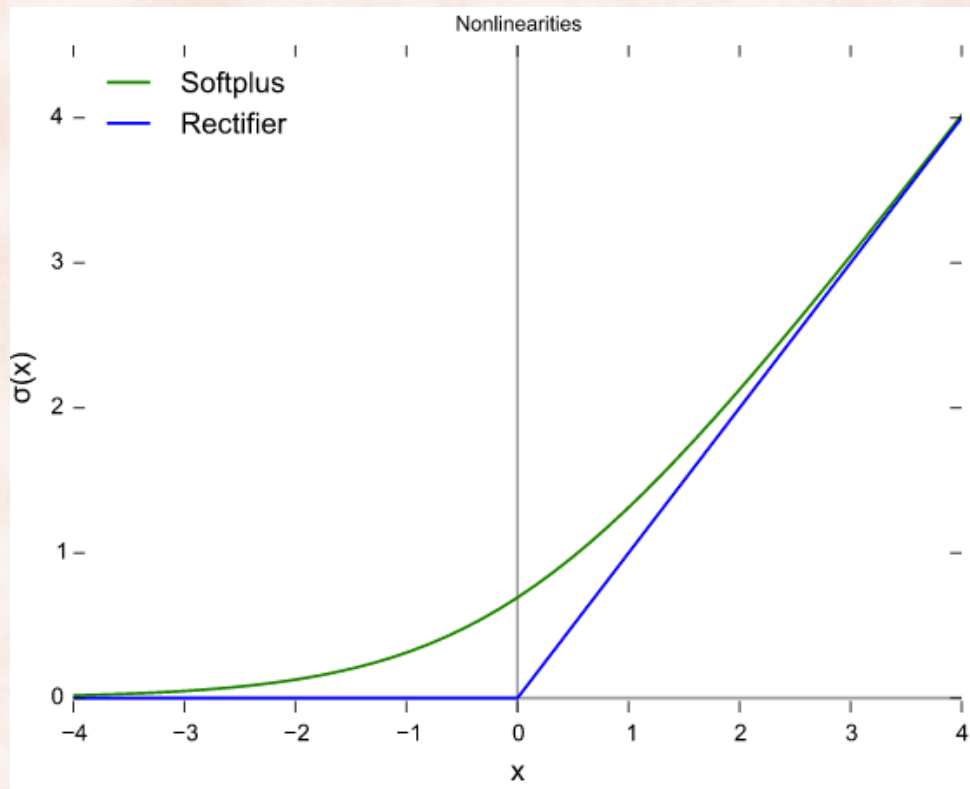
---

- Sigmoid and Tanh saturate for values not close to 0:
  - “kill” gradients, bad behavior for gradient-based learning.
- ReLU does not saturate for values  $> 0$ :
  - greatly accelerates learning, fast implementation.
  - fragile during training and can “die”, due to 0 gradient:
    - initialize all  $b$ 's to a small, positive value, e.g. 0.1.



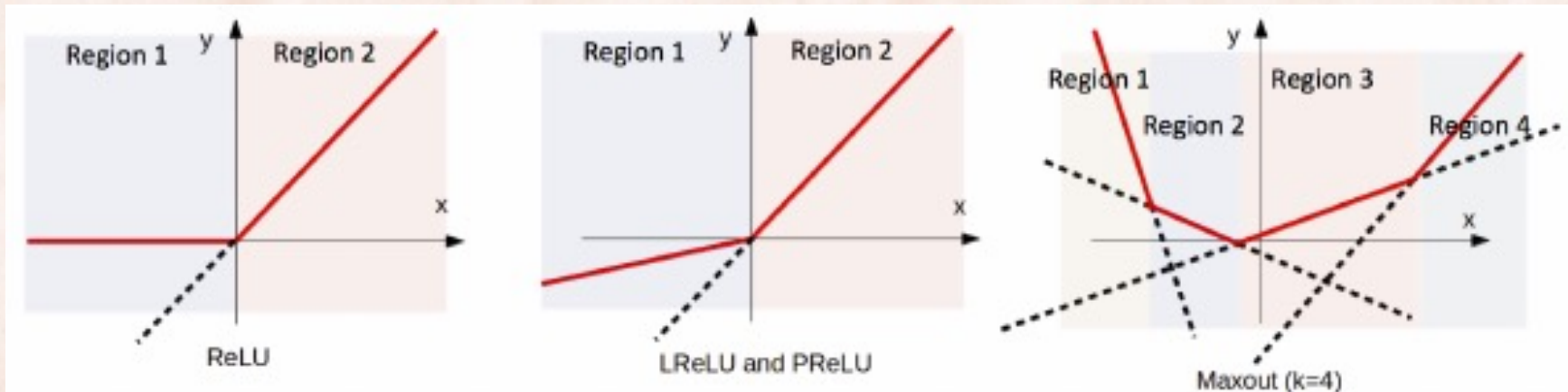
# ReLU vs. Softplus

- Softplus  $g(z) = \ln(1+e^z)$  is a smooth version of the rectifier.
  - Saturates less than ReLU, yet ReLU still does better [Glorot, 2011].



# ReLU and Generalizations

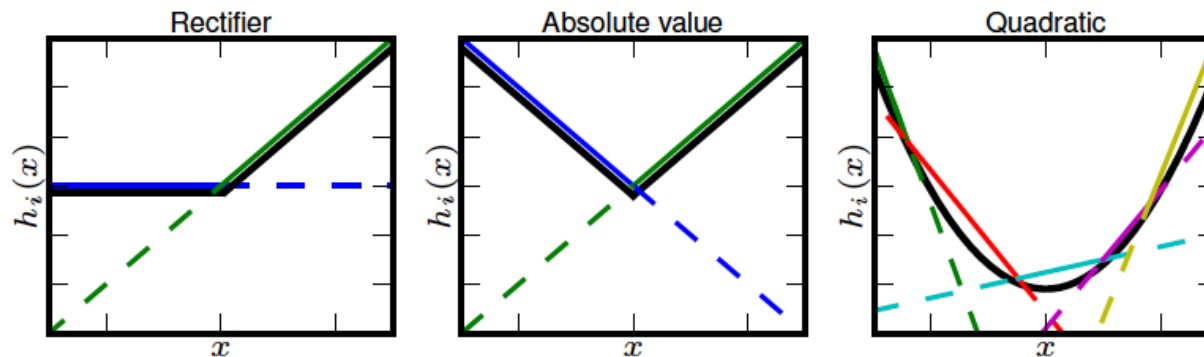
- Leaky ReLU attempts to fix the “dying” ReLU problem.
- Maxout subsumes (leaky) ReLU, but needs more params.



# Maxout Networks

[Goodfellow et al., ICML'13]

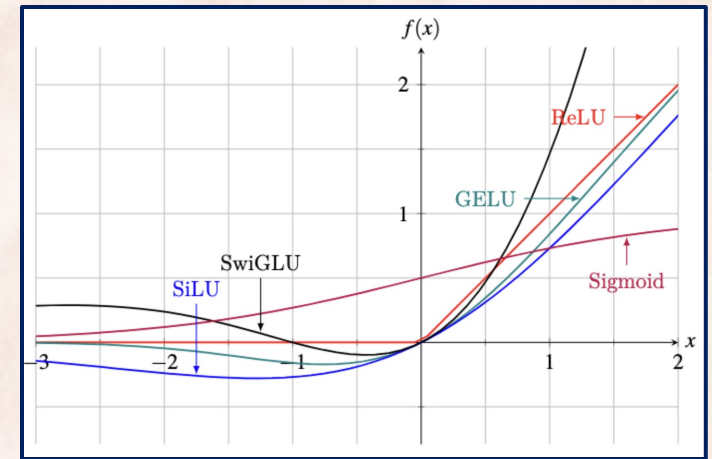
- Maxout units can learn the activation function.



*Figure 1.* Graphical depiction of how the maxout activation function can implement the rectified linear, absolute value rectifier, and approximate the quadratic activation function. This diagram is 2D and only shows how maxout behaves with a 1D input, but in multiple dimensions a maxout unit can approximate arbitrary convex functions.

# More Activation Functions

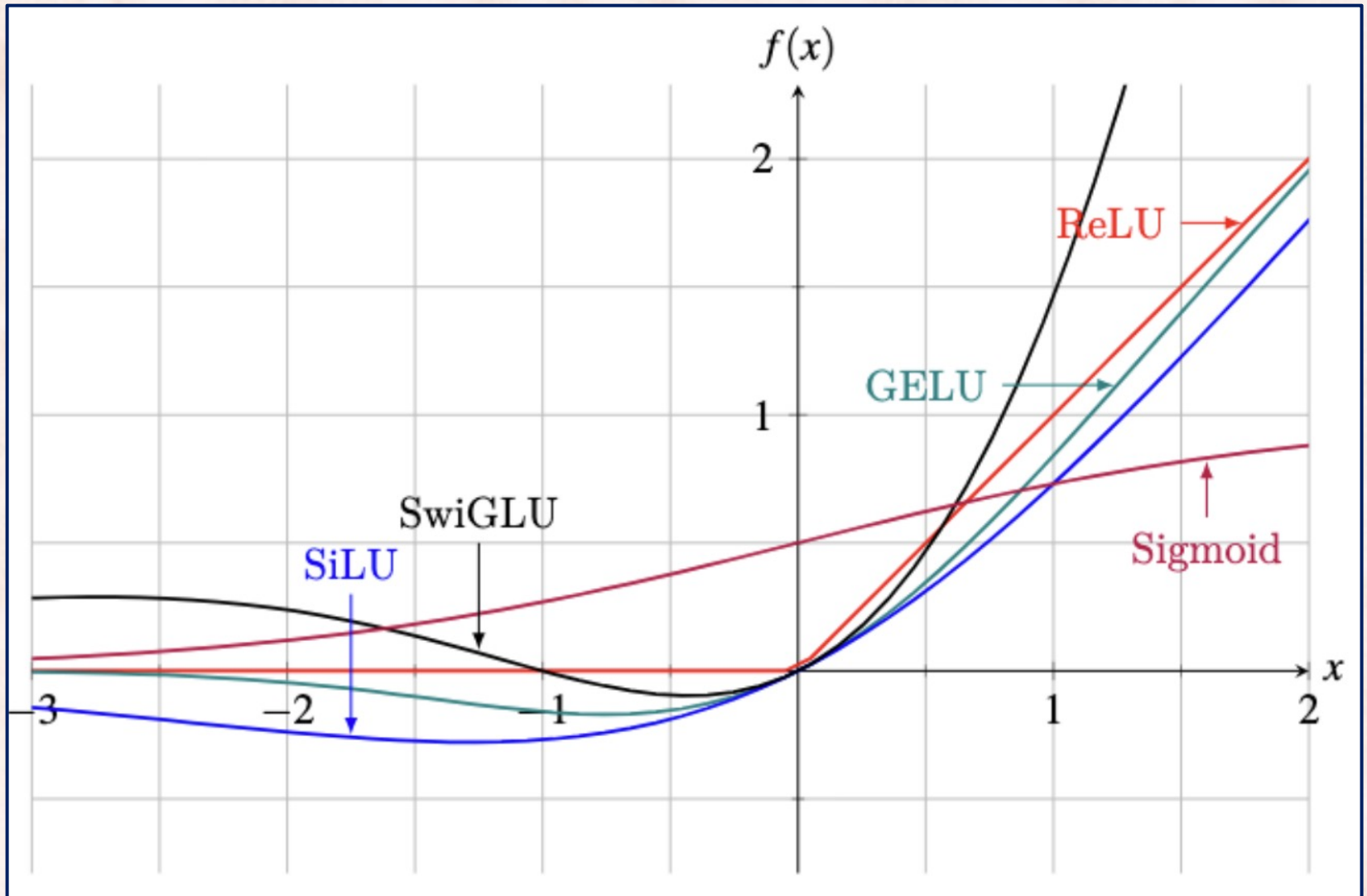
- **ReLU**: the rectifier activation  $g(z) = \max\{0, z\}$ .
- **Absolute value ReLU**:  $g(z) = |z|$ .
- **Leaky ReLU**:  $g(a) = \max\{0, a\} + \alpha \min(0, a)$ .  
⇒ the network computes a *piecewise linear function*.



- **Sigmoid Linear Unit**:  $\text{SiLU}(x) = x\sigma(x)$
- **Swish**:  $\text{Swish}_\beta(x) = x\sigma(\beta x)$
- **Gaussian Error Linear Units**:  $\text{GELU}(x) = xP(z \leq x)$ , where  $z \sim N(0,1)$ 
  - $0.5x(1 + \tanh[\sqrt{2/\pi}(x + 0.044715x^3)]) \cong x\sigma(1.702x)$ 
    - Widely used in Transformer models.
- **Gated Linear Unit**:  $\text{GLU}(x) = (xV + v) \otimes \sigma(xW + b)$
- **Swish-gated Linear Unit**:  $\text{SwiGLU}(x) = (xV + v) \otimes \text{Swish}_\beta(xW + b)$ 
  - Frequently used, e.g. Llama 3, Qwen 3, ...

# More Activation Functions

*GLU Variants Improve Transformer, by Noam Shazeer, 2020*



# Important Architectural Techniques for NNs, e.g., Transformer

- **Activation** functions.
  - ReLU, SiLU, Swish, **GELU**, GLU, **SwiGLU**.
- **Normalization** of logits / activations:
  - Batch normalization.
  - **Layer normalization**.
- **Residual connections**.
  - Deep **residual networks**.
- **Deep architectures**.
  - **Compact** representation of **highly-varying** functions.

# Batch Normalization: Reducing Internal Covariate Shift

---

[Ioffe & Szegedy, ICML'15]

- **Internal Covariate Shift:**

- The distribution of each layer's inputs changes during training:
  - because the parameters of the previous layers change.
- This slows down the training:
  - requires lower learning rates and careful param initialization.
  - notoriously hard to train models with saturating nonlinearities.

- **Batch Normalization:**

- Normalize the input for each layer, for each training minibatch:
  - Allows for much higher learning rates, init. less important.
  - Acts as a regularizer, eliminating the need for Dropout.

# Batch Normalization

[Ioffe & Szegedy, ICML'15]

---

1. Normalize each logit  $x$  using minibatch  $\mu$  and  $\sigma$ .

$$\hat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

2. Train parameters that scale ( $\gamma$ ) and shift ( $\beta$ ) the normalized value:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

- Thus allow the overall transformation to represent the identity transform.

# Batch Normalization: Inference @ Training

[Ioffe & Szegedy, ICML'15]

**Input:** Values of  $x$  over a mini-batch:  $\mathcal{B} = \{x_{1\dots m}\}$ ;

Parameters to be learned:  $\gamma, \beta$

**Output:**  $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

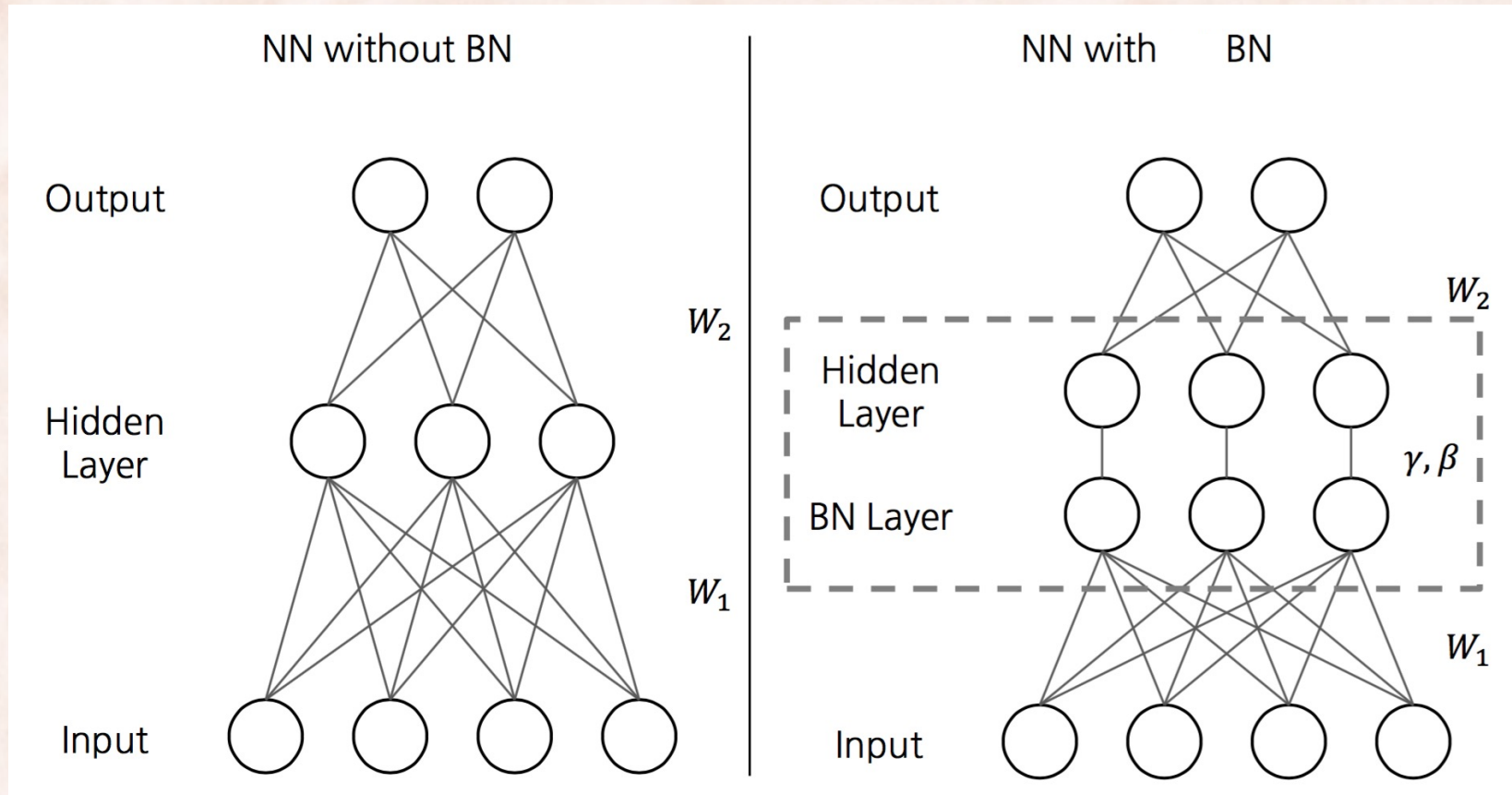
$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i) \quad // \text{ scale and shift}$$

**Algorithm 1:** Batch Normalizing Transform, applied to activation  $x$  over a mini-batch.

# Batch Normalization: Computation Graph

[Ioffe & Szegedy, ICML'15]



# Batch Normalization: Gradients @ Training

[Ioffe & Szegedy, ICML'15]

*Apply the chain rule to compute gradient of loss  $l$  w.r.t. new parameters  $\gamma$  and  $\beta$  via gradients of loss  $l$  w.r.t. the new logit value  $y$ .*

$$\frac{\partial l}{\partial \hat{x}_i} = \frac{\partial l}{\partial y_i} \cdot \gamma$$

$$\frac{\partial l}{\partial \sigma_B^2} = \sum_{i=1}^m \frac{\partial l}{\partial \hat{x}_i} \cdot (x_i - \mu_B) \cdot \frac{-1}{2} (\sigma_B^2 + \epsilon)^{-3/2}$$

$$\frac{\partial l}{\partial \mu_B} = \sum_{i=1}^m \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{-1}{\sqrt{\sigma_B^2 + \epsilon}}$$

$$\frac{\partial l}{\partial x_i} = \frac{\partial l}{\partial \hat{x}_i} \cdot \frac{1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial l}{\partial \sigma_B^2} \cdot \frac{2(x_i - \mu_B)}{m} + \frac{\partial l}{\partial \mu_B} \cdot \frac{1}{m}$$

$$\frac{\partial l}{\partial \gamma} = \sum_{i=1}^m \frac{\partial l}{\partial y_i} \cdot \hat{x}_i$$

$$\frac{\partial l}{\partial \beta} = \sum_{i=1}^m \frac{\partial l}{\partial y_i}$$

# Batch Normalization: Inference @ Test

[Ioffe & Szegedy, ICML'15]

At test time, use the trained  $\gamma$  and  $\beta$  to scale the logits scores, but instead of minibatch mean and variance **use the population statistics**:

- **Moving averages for the mean**  $E[x]$ , over all training minibatches.
- **Unbiased estimate for the variance**  $\text{Var}[x]$ , over all training minibatches.

(Duchi et al., 2011). The normalization of activations that depends on the mini-batch allows efficient training, but is neither necessary nor desirable during inference; we want the output to depend only on the input, deterministically. For this, once the network has been trained, we use the normalization

$$\hat{x} = \frac{x - E[x]}{\sqrt{\text{Var}[x] + \epsilon}}$$

using the population, rather than mini-batch, statistics. Neglecting  $\epsilon$ , these normalized activations have the same mean 0 and variance 1 as during training. We use the unbiased variance estimate  $\text{Var}[x] = \frac{m}{m-1} \cdot E_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$ , where the expectation is over training mini-batches of size  $m$  and  $\sigma_{\mathcal{B}}^2$  are their sample variances. Using moving averages instead, we can track the accuracy of a model as it trains.

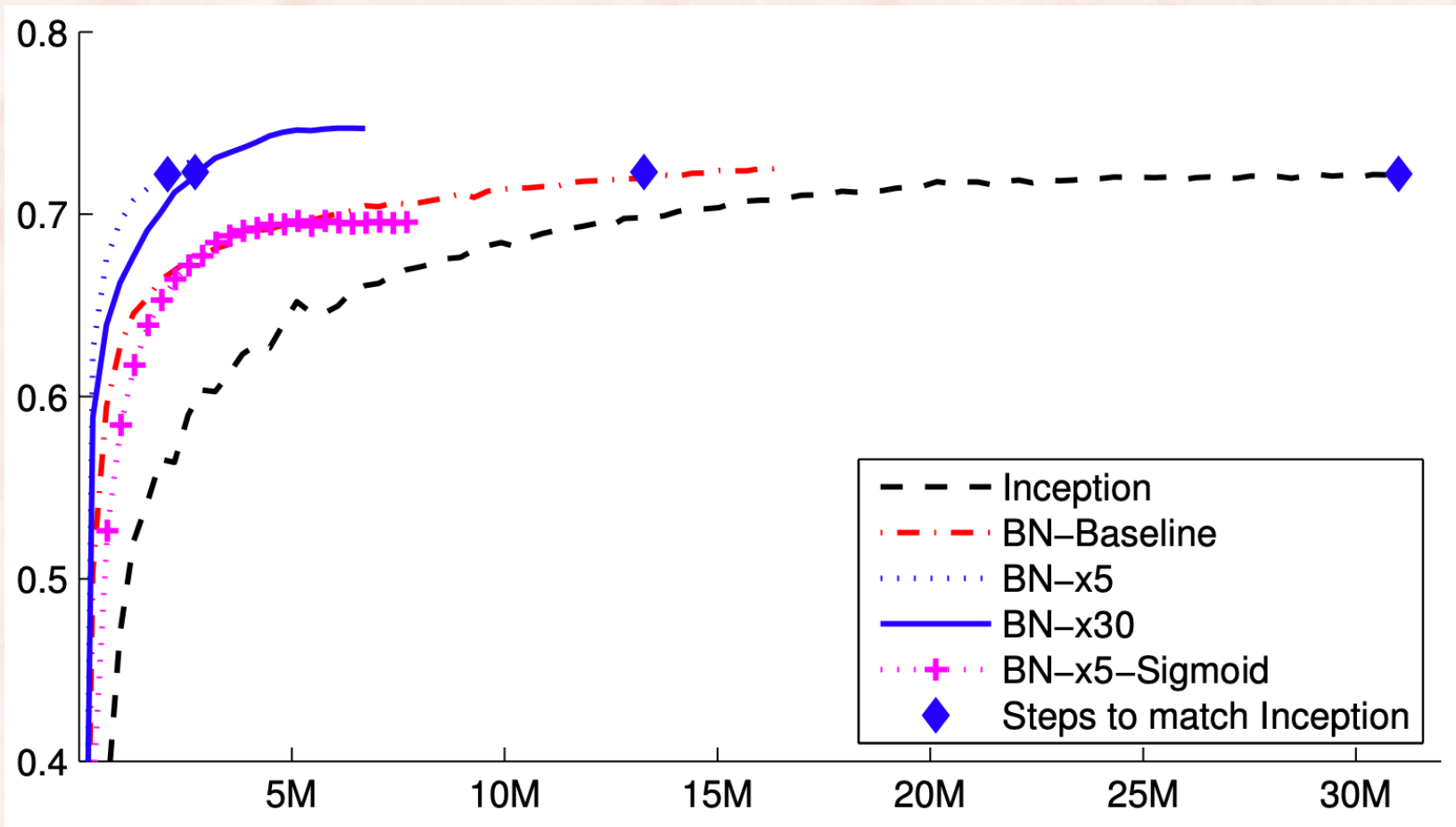
- ```
8: for  $k = 1 \dots K$  do
9:   // For clarity,  $x \equiv x^{(k)}$ ,  $\gamma \equiv \gamma^{(k)}$ ,  $\mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$ , etc.
10:  Process multiple training mini-batches  $\mathcal{B}$ , each of
    size  $m$ , and average over them:
     $E[x] \leftarrow E_{\mathcal{B}}[\mu_{\mathcal{B}}]$ 
     $\text{Var}[x] \leftarrow \frac{m}{m-1} E_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$ 
11:  In  $N_{\text{BN}}^{\text{inf}}$ , replace the transform  $y = \text{BN}_{\gamma, \beta}(x)$  with
     $y = \frac{\gamma}{\sqrt{\text{Var}[x] + \epsilon}} \cdot x + \left( \beta - \frac{\gamma E[x]}{\sqrt{\text{Var}[x] + \epsilon}} \right)$ 
12: end for
```

# Batch Normalization:

Remove Dropout, Reduce L2 weight regularization,  
Allow **higher learning rates**, Accelerates learning rate decay

[Ioffe & Szegedy, ICML'15]

- $-xK$  means  $K$  times original learning rate.



# Layer Normalization

[Ba, Kiros & Hinton, 2016]

---

- **Batch Normalization:**
  - The effect is dependent on the mini-batch size.
  - Not obvious how to apply it to recurrent neural networks.
- **Layer Normalization:** Fix the mean and the variance of the summed inputs within each layer:
  - Compute the mean and variance used for normalization from all of the summed inputs to the neurons in a layer on a single training example.
  - [Like BN] Give each neuron its own adaptive bias and gain which are applied after the normalization but before the non-linearity.
  - [Unlike BN] Layer normalization performs exactly the same computation at training and test times.

# Layer Normalization

[Ba, Kiros & Hinton, 2016]

---

1. Compute layer normalization statistics over all hidden units in the same layer:

$$\mu^l = \frac{1}{H} \sum_{i=1}^H a_i^l \quad \sigma^l = \sqrt{\frac{1}{H} \sum_{i=1}^H (a_i^l - \mu^l)^2}$$

2. Learn an adaptive bias  $b$  and gain  $g$  for each neuron after the normalization:

$$h_i = f\left(\frac{g_i}{\sigma_i} (a_i - \mu_i) + b_i\right)$$

3. BN better than LN for CNNs, LN works well for RNNs.

# Important Architectural Techniques for NNs, e.g., Transformer

- **Activation** functions.
  - ReLU, SiLU, Swish, **GELU**, GLU, **SwiGLU**.
- **Normalization** of logits / activations:
  - Batch normalization.
  - **Layer normalization**.
- **Residual connections**.
  - Deep **residual networks**.
- **Deep architectures**.
  - **Compact** representation of **highly-varying** functions.

# Training/Testing Error Increases with Very Large Depth

[[He et al, CVPR 2016](#)]

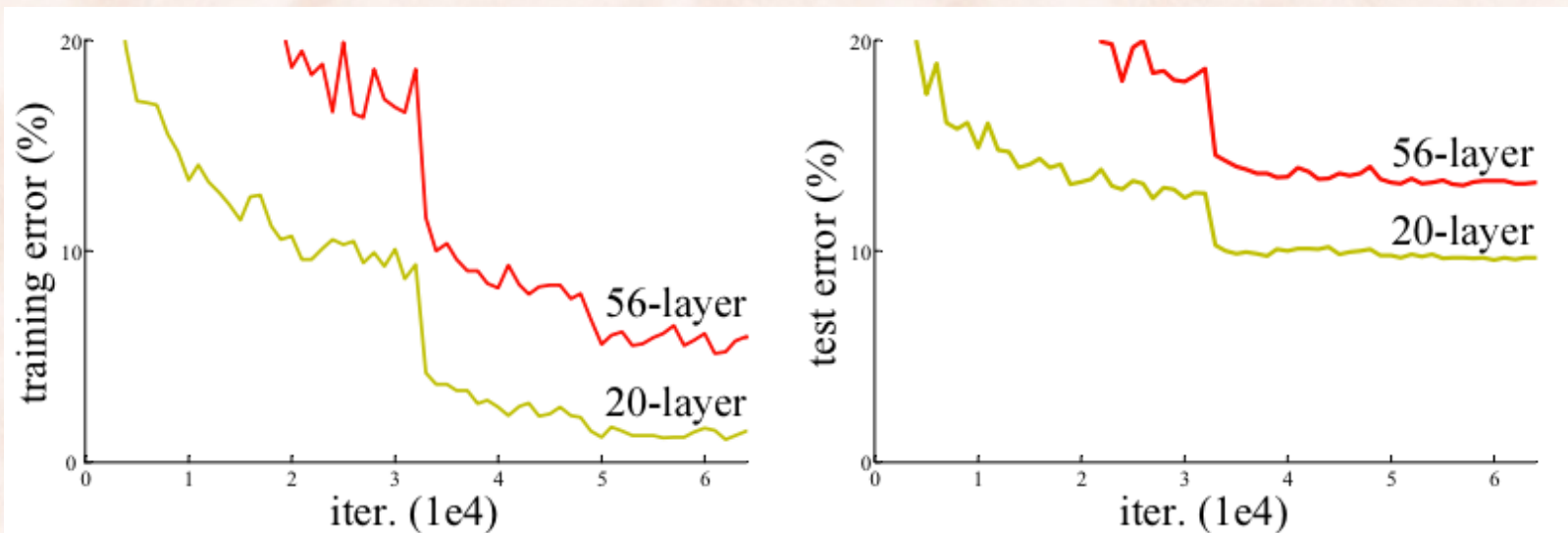


Figure 1. Training error (left) and test error (right) on CIFAR-10 with 20-layer and 56-layer “plain” networks. The deeper network has higher training error, and thus test error. Similar phenomena

# Residual Connections

[[He et al, CVPR 2016](#)]

- Make it easier for the network to learn identity mappings.
- **Shortcut connections:**
  - Make identity mappings trivial (all weights 0).
  - Addition operation distributes the gradient.

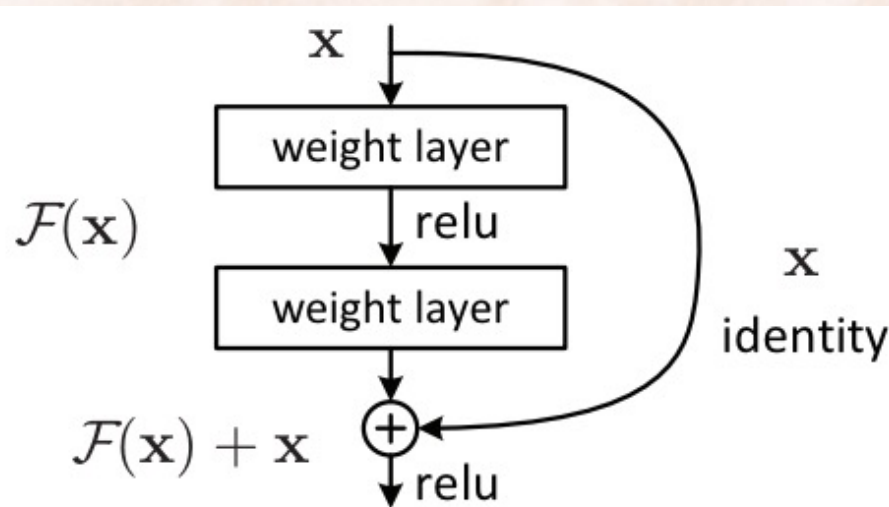
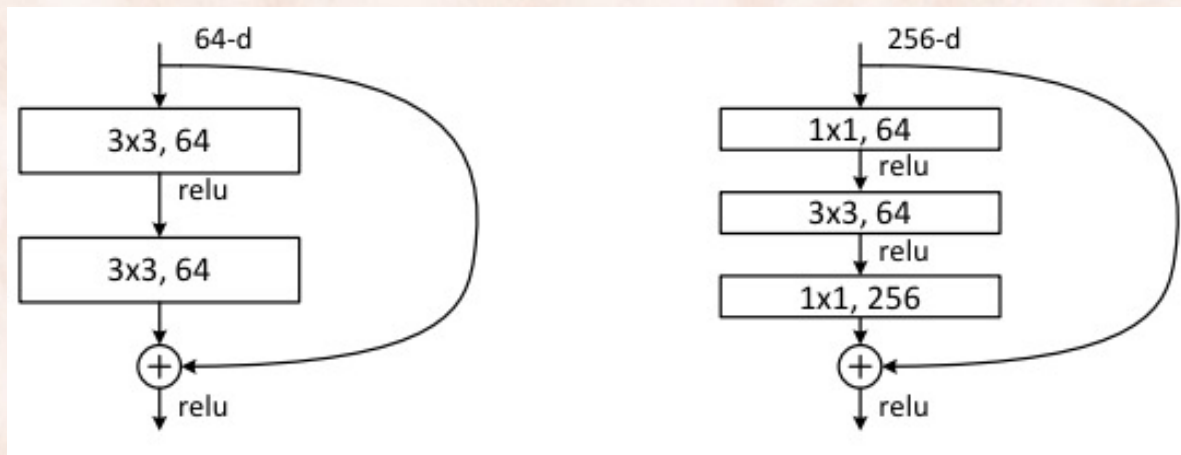


Figure 2. Residual learning: a building block.

# Residual Connections: Microsoft ResNet

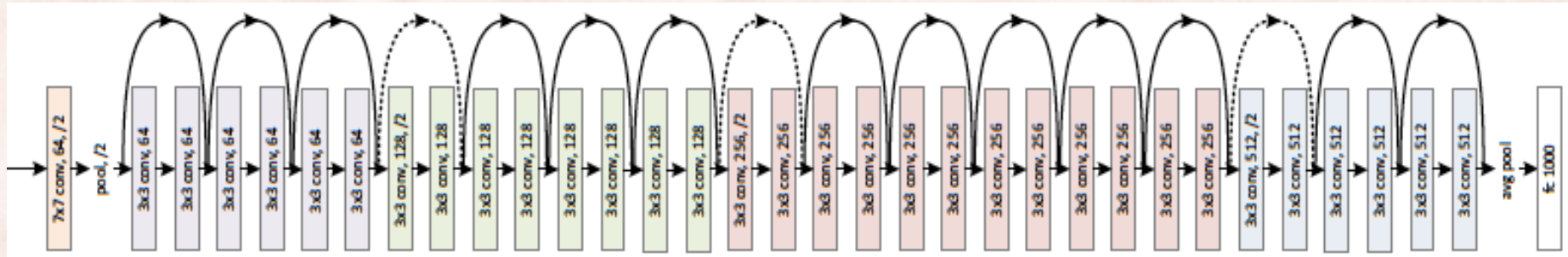
[[He et al, CVPR 2016](#)]

- The winner in ImageNet 2015:
  - **Ultra-deep**: from 34 to 152 layers.
  - **Batch normalization** after each convolution, before activation.
  - First layer is (7x7 conv, 64 kernels, S=2), and (3x3 pool, S=2).
  - A **shortcut connection** is added for every block of:
    - Two (3x3,64;relu) layers (34 deep).
    - Three (1x1,64;relu, 3x3,64;relu, 1x1,256;relu) layers (152 deep).



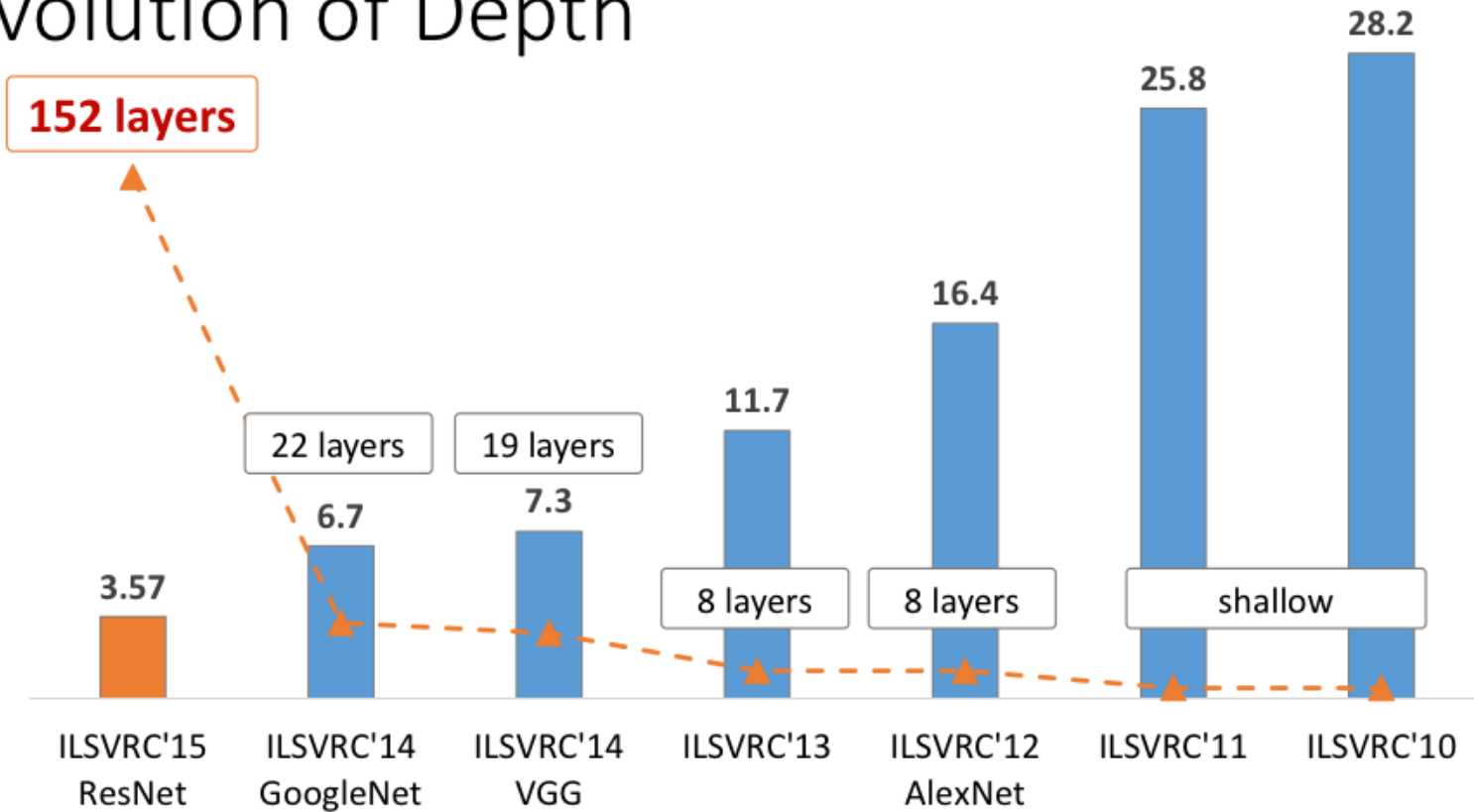
# Microsoft ResNet

[He et al, CVPR 2016]



- SGD with minibatch of 256, momentum of 0.9, decay of 0.0001.
- Learning rate = 0.1 divided by 10 when error plateaus,  $6 \times 10^5$  epochs.
- Depth vs. error rate on validation:
  - 34 depth: top5 = 7.4%, top1 = 24.2%.
  - 152 depth: top5 = **5.7%**, top1 = **21.4%**

# Revolution of Depth



ImageNet Classification top-5 error (%)

# Important Architectural Techniques for NNs, e.g., Transformer

- **Activation** functions.
  - ReLU, SiLU, Swish, **GELU**, GLU, **SwiGLU**.
- **Normalization** of logits / activations:
  - Batch normalization.
  - **Layer normalization**.
- **Residual connections**.
  - Deep **residual networks**.
- **Deep architectures**.
  - **Compact** representation of **highly-varying** functions.

# Shallow vs. Deep Networks

---

- A 1-hidden layer network is a fairly **shallow network**.
  - Effective for MNIST, but limited by simplicity of features.
- A **deep network** is a  $k$ -layer network,  $k > 1$ .
  - Computes more complex features of the input, as  $k$  gets larger.
  - Each hidden layer computes a non-linear transformation of the previous layer.

## *Conjecture*

A deep network has significantly greater representational power than a shallow one.

# Number of Linear Regions of Shallow vs. Deep Networks

[Montufar et al., NIPS'14]

## *Conjecture*

A deep network has significantly greater representational power than a shallow one.

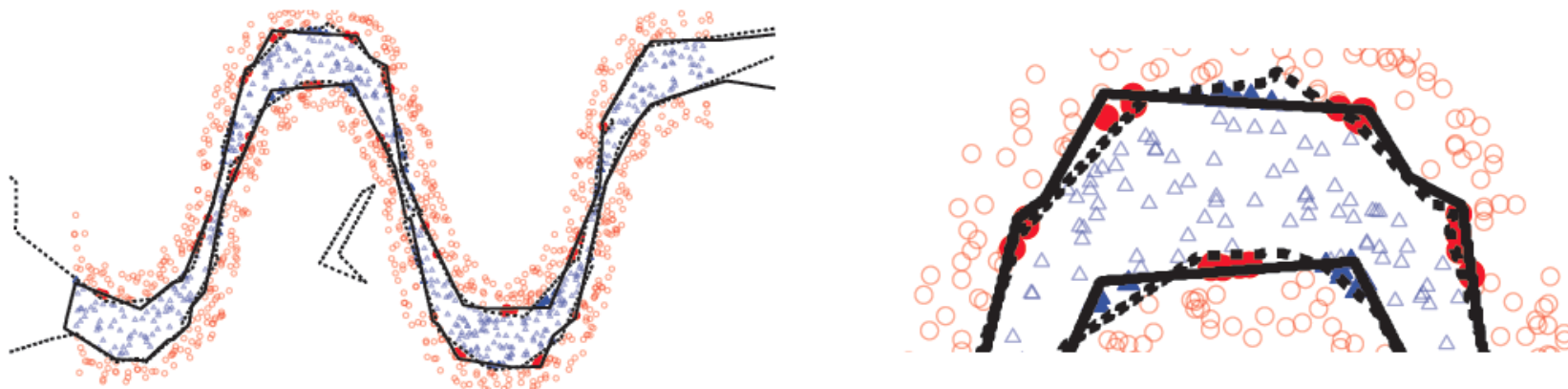


Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

# Deep vs. Shallow Architectures

---

- A function is **highly varying** when a piecewise (linear) approximation would require a large number of pieces.
- **Depth** of an architecture refers to the number of levels of composition of non-linear operations in the function computed by the architecture.
- *Conjecture:* **Deep architectures** can **compactly** represent **highly-varying functions**:
  - The expression of a function is **compact** when it has few computational elements.
  - Same highly-varying functions would require very large shallow networks.

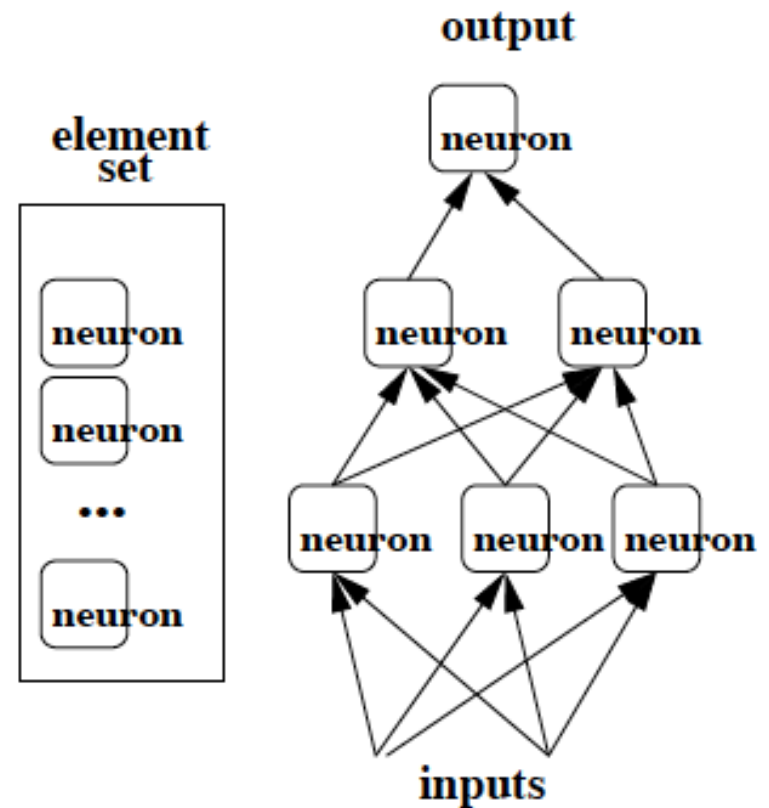
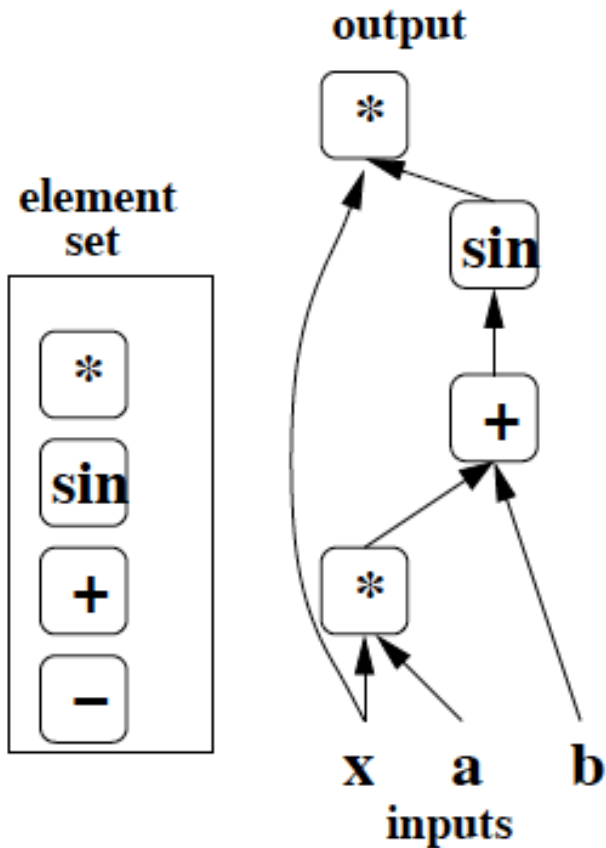
# Graphs of Computations

---

- A function can be expressed by the composition of **computational elements** from a given set:
  - logic operators.
  - logistic operators.
  - multiplication and additions.
- The function is defined by a **graph of computations**:
  - A directed acyclic graph, with one node per computational element.
  - Depth of architecture = depth of the graph = longest path from an input node to an output node.

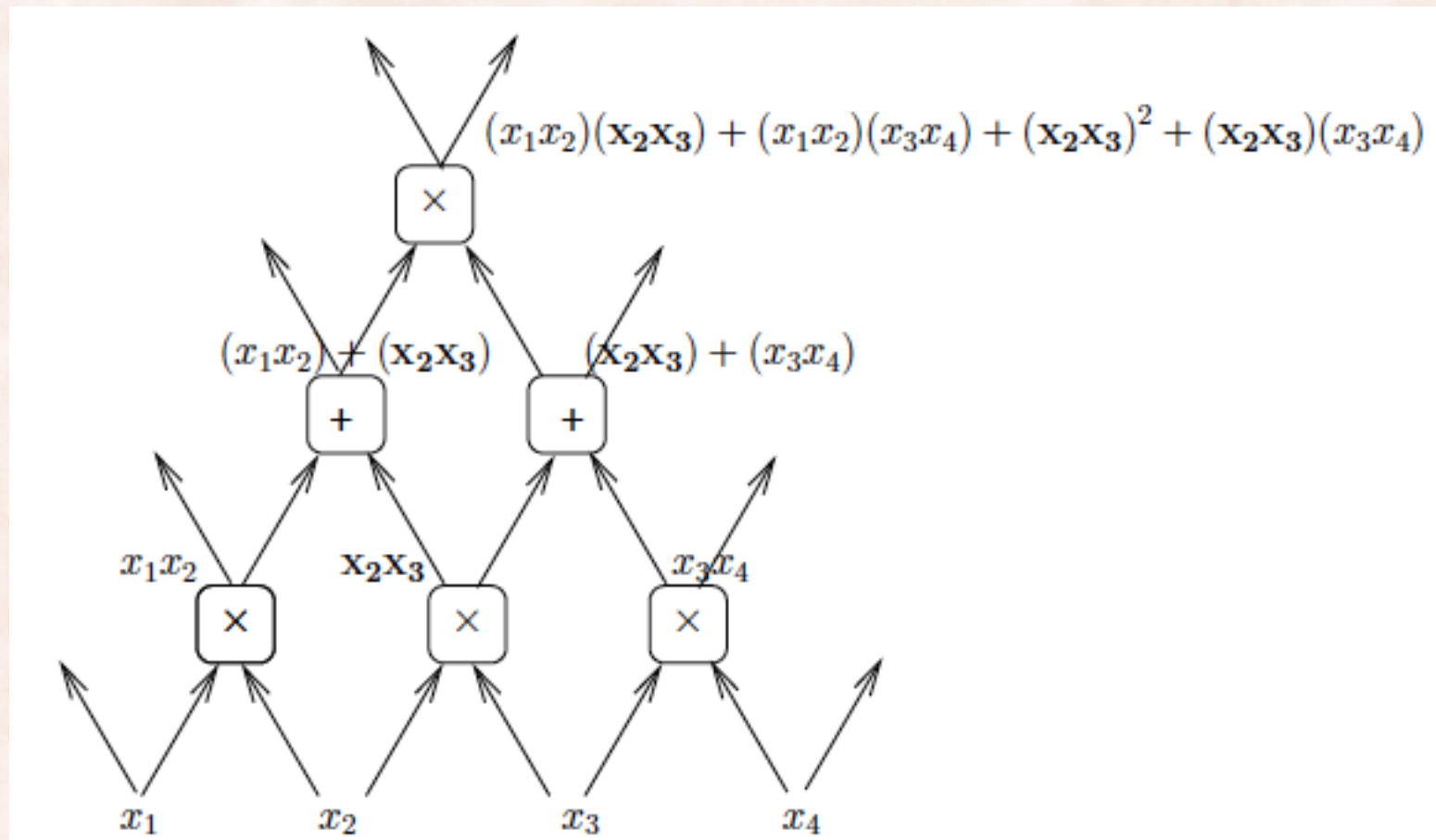
# Functions as Graphs of Computations

[Bengio, FTML'09]



# Polynomials as Graphs of Computations

[Bengio, FTML'09]



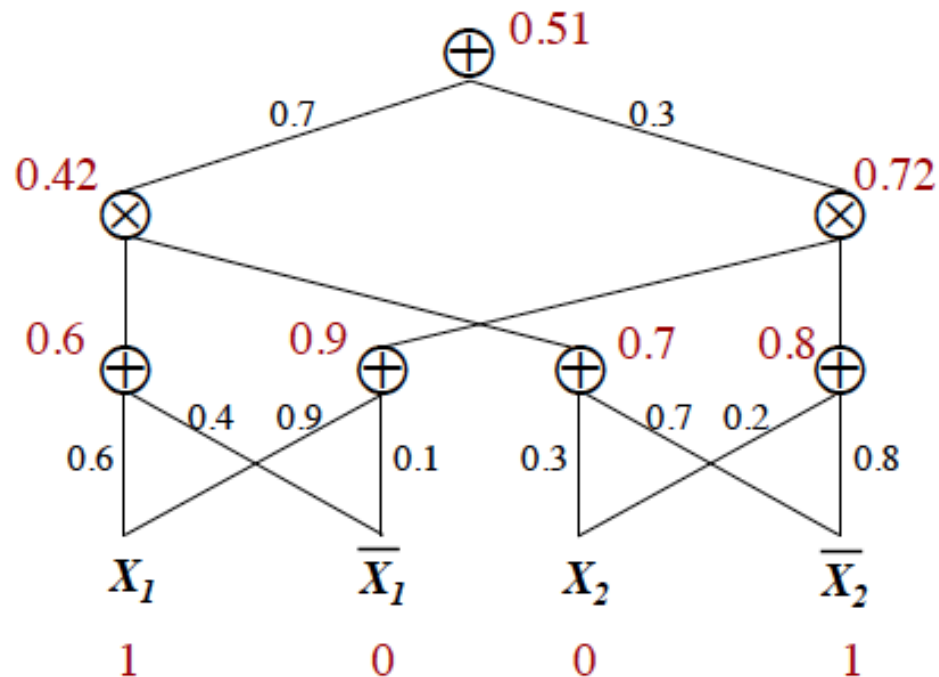
# Sum-Product Networks (SPNs)

[Poon & Domingos, UAI'11]

- Rooted, weighted DAG.
- **Nodes:** Sum, Product, (Input) Indicators.
- **Weights** on **edges** from sums to children.

$X: X_1 = 1, X_2 = 0$

|             |   |
|-------------|---|
| $X_1$       | 1 |
| $\bar{X}_1$ | 0 |
| $X_2$       | 0 |
| $\bar{X}_2$ | 1 |



# ML Models as Graphs of Computations

[Bengio, FTML'09]

---

- If we include affine operations and their possible composition with sigmoids in the set of computational elements, **linear regression and logistic regression** have depth 1, i.e., have a single level.
- When we put a fixed kernel computation  $K(u, v)$  in the set of allowed operations, along with affine operations, **kernel machines** (Schölkopf, Burges, & Smola, 1999a) with a fixed kernel can be considered to have two levels. The first level has one element computing  $K(x, x_i)$  for each prototype  $x_i$  (a selected representative training example) and matches the input vector  $x$  with the prototypes  $x_i$ . The second level performs an affine combination  $b + \sum_i \alpha_i K(x, x_i)$  to associate the matching prototypes  $x_i$  with the expected response.
- When we put artificial neurons (affine transformation followed by a non-linearity) in our set of elements, we obtain ordinary **multi-layer neural networks** (Rumelhart et al., 1986b). With the most common choice of one hidden layer, they also have depth two (the hidden layer and the output layer).
- **Boosting** (Freund & Schapire, 1996) usually adds one level to its base learners: that level computes a vote or linear combination of the outputs of the base learners.
- **Stacking** (Wolpert, 1992) is another meta-learning algorithm that adds one level.
- Based on current knowledge of brain anatomy (Serre et al., 2007), it appears that **the cortex** can be seen as a deep architecture, with 5 to 10 levels just for the visual system.

# Deep vs. Shallow Architectures

---

- Deep architectures were shown to be more compact for:
  - *Boolean circuits* [Hastad, 1986].
  - *Monotone weighted threshold circuits* [Hastad and Goldman, 1993].
- Same holds for *networks with continuous-valued activations* [Maass, 1992].
- Many modern neural networks use rectified linear units:
  1. *ReLU networks* are universal approximators [Leshno et al., 1993].
  2. Are deep ReLU networks more compact than shallow ones?
    - **YES!** [Montufar et al., NIPS'14]

# Number of Linear Regions of Shallow vs. Deep Networks

[Montufar et al., NIPS'14]

## *Theorem*

A deep network has significantly greater representational power than a shallow one.

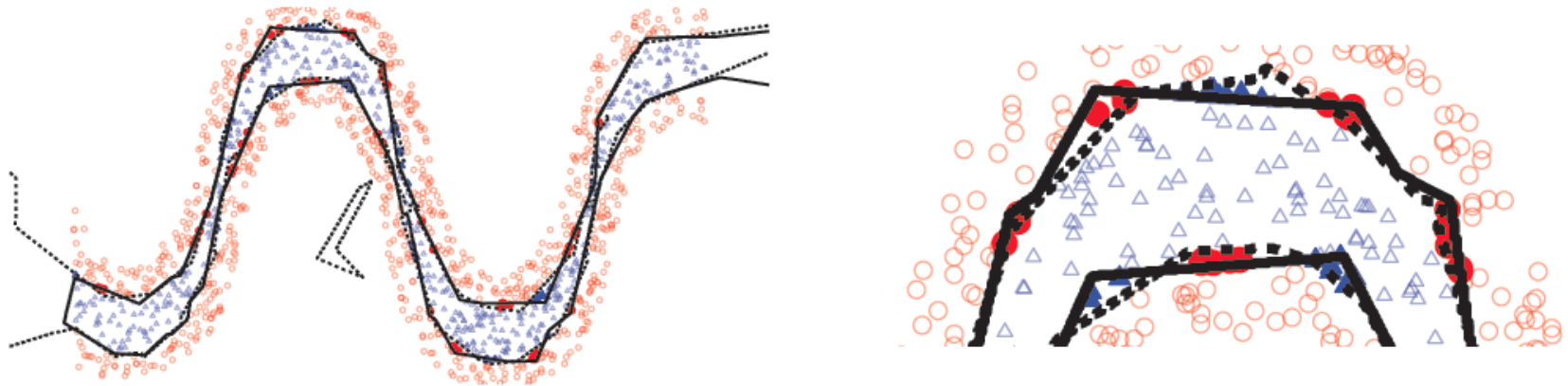


Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

# Deep vs. Shallow Rectifier Networks

[Montufar et al., NIPS'14]

---

- A *linear region* of a piecewise linear function  $F: \mathbb{R}^d \rightarrow \mathbb{R}^m$  is a maximal connected subset of the input-space  $\mathbb{R}^d$ , on which  $F$  is linear.
  - The number of linear regions carved out by a *deep rectifier network* with  $d$  inputs, depth  $l$ , and  $n$  units per hidden layer, is:

$$O \left( \binom{n}{d}^{d(l-1)} n^d \right)$$

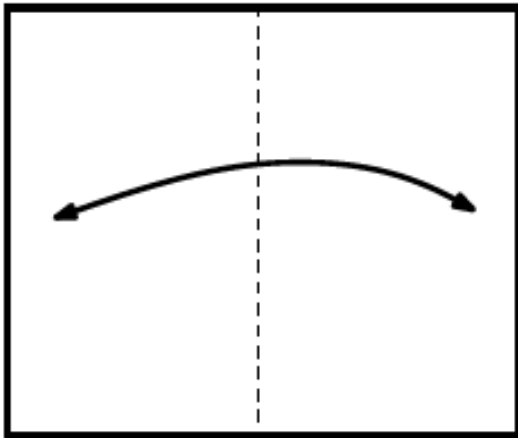
- In the case of *maxout networks* with  $k$  filters per unit, the number of linear regions is:

$$O \left( k^{(l-1)+d} \right)$$

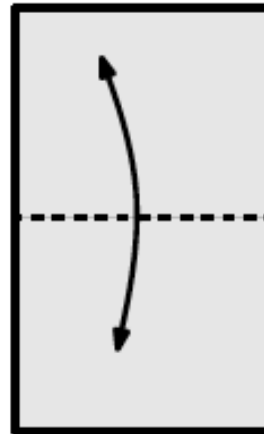
# Space Foldings

---

- Each hidden layer of a deep neural network can be associated with a folding operator.



1. Fold along the vertical axis



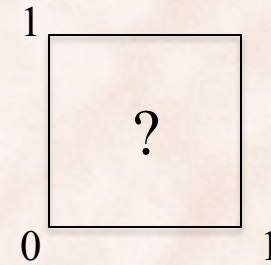
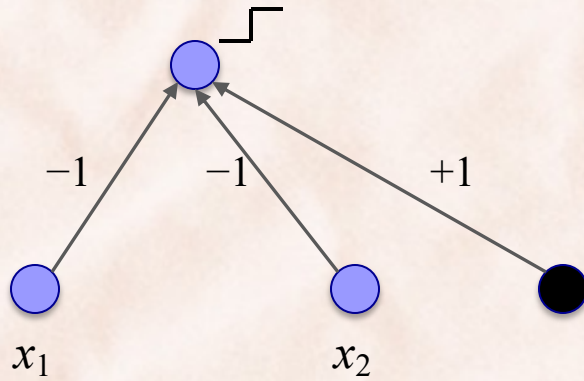
2. Fold along the horizontal axis



3.

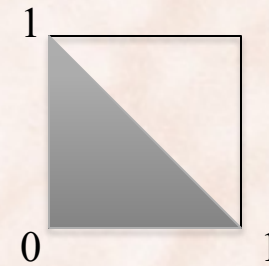
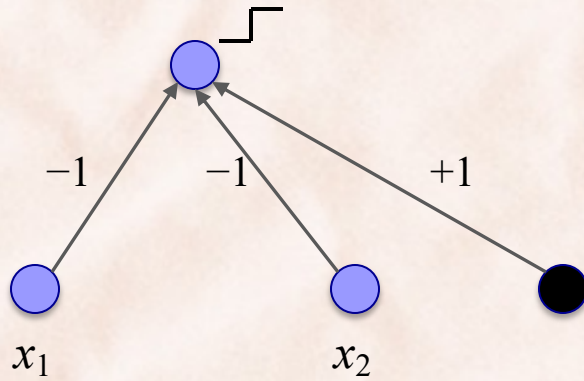
# Folding Example

---

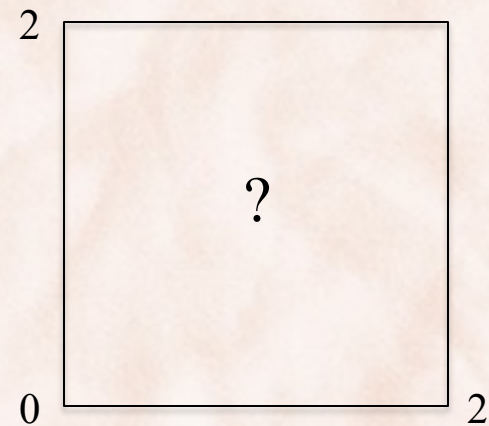
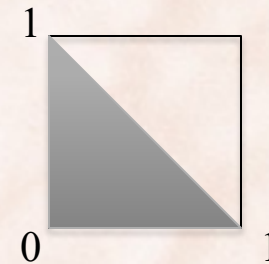
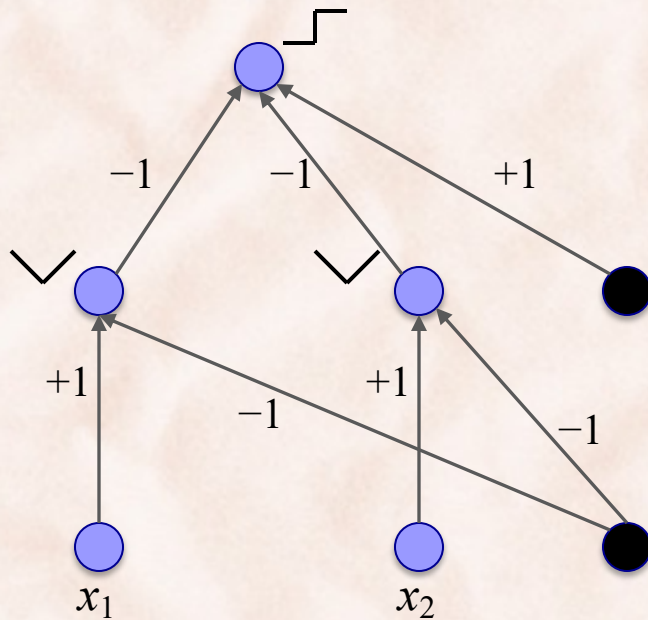


# Folding Example

---

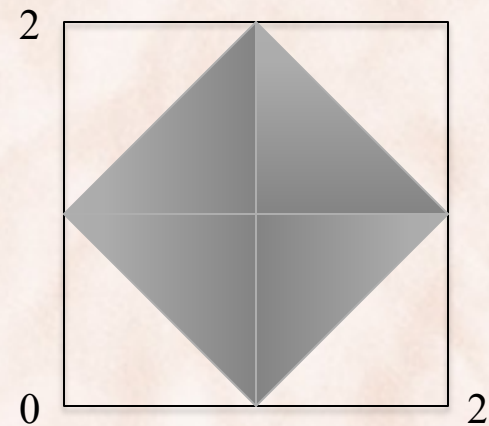
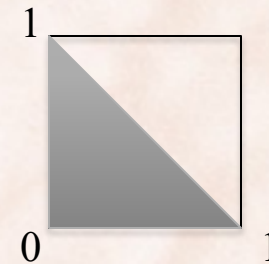
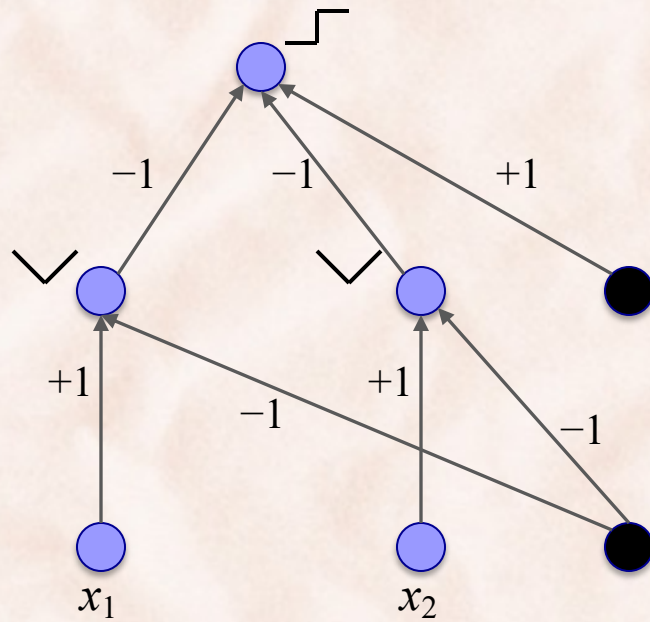


# Folding Example

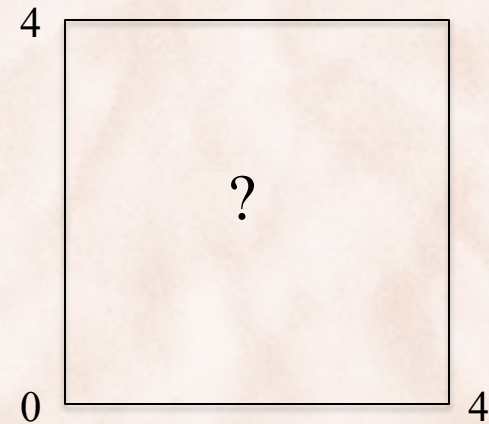
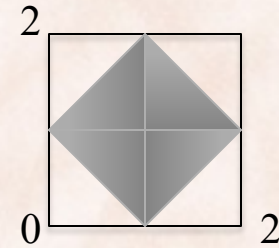
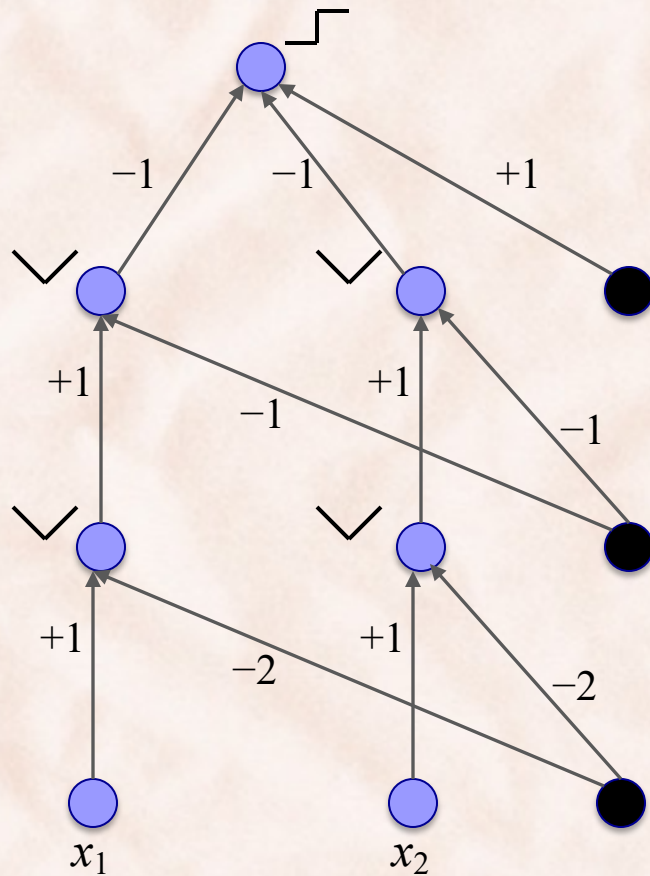


# Folding Example

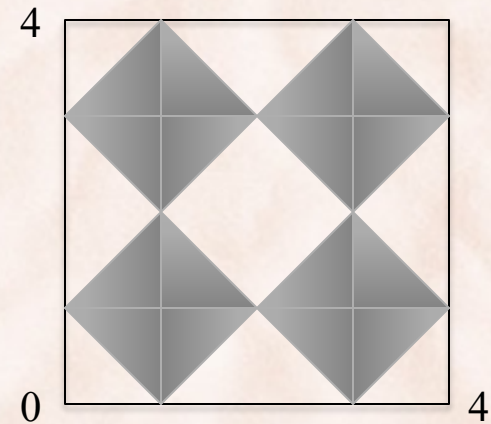
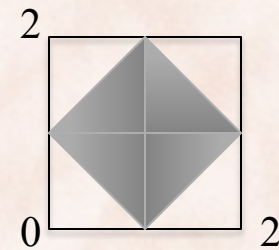
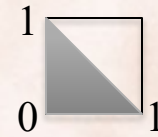
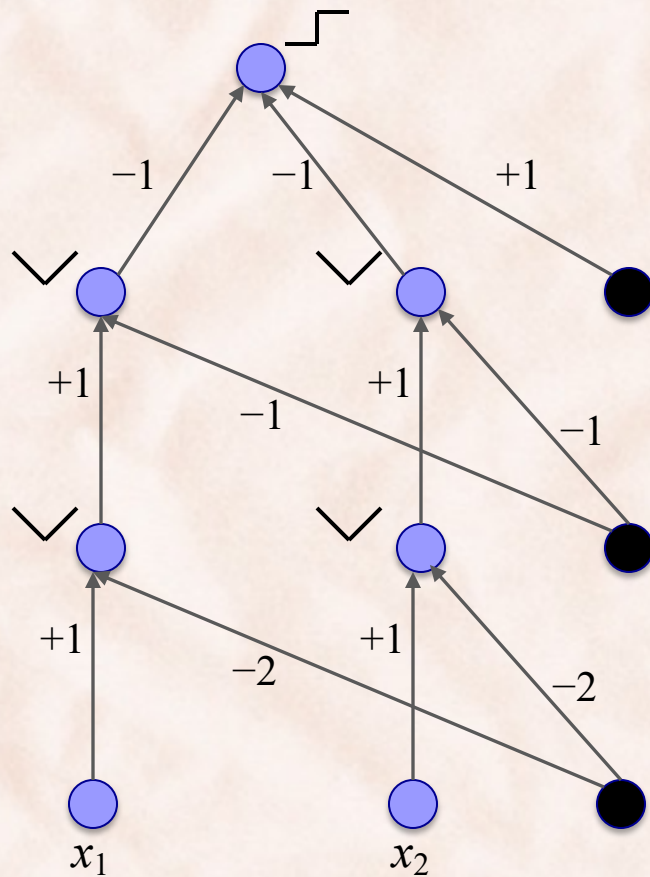
---



# Folding Example

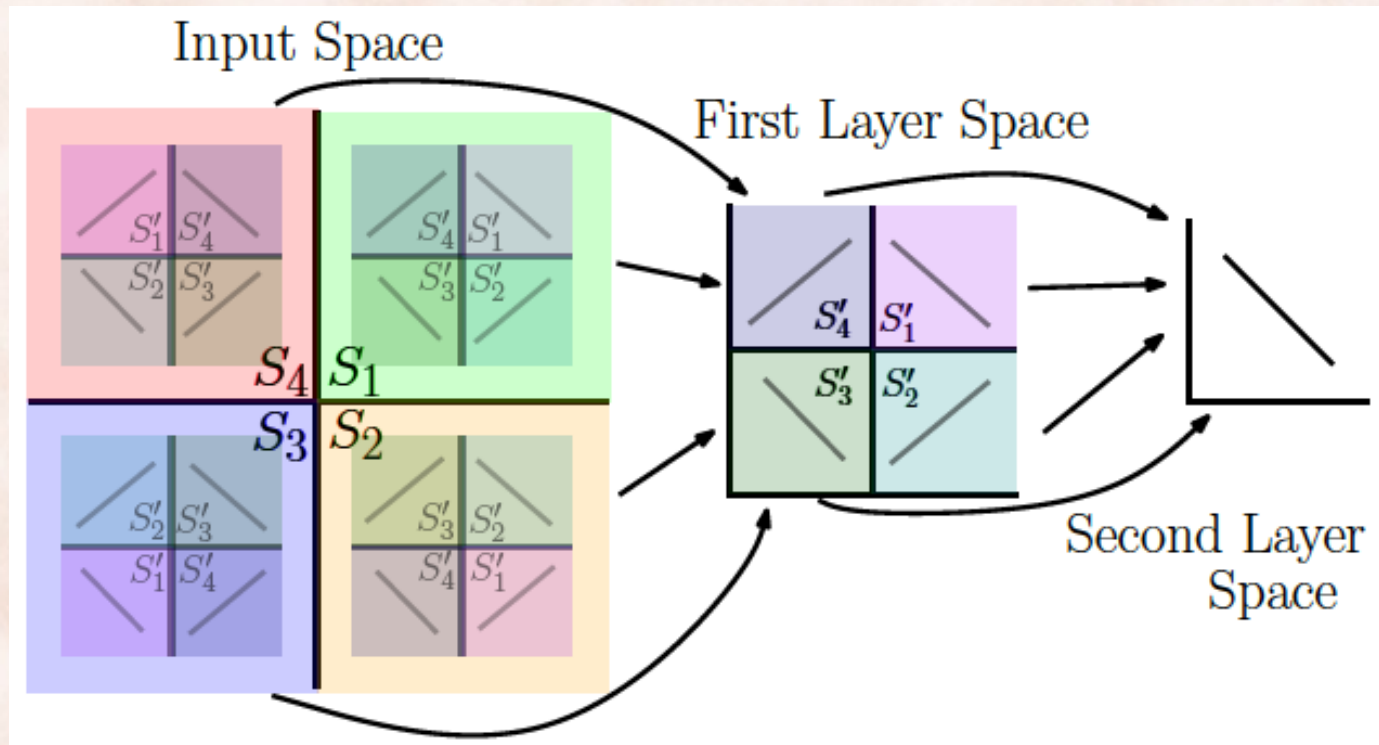


# Folding Example

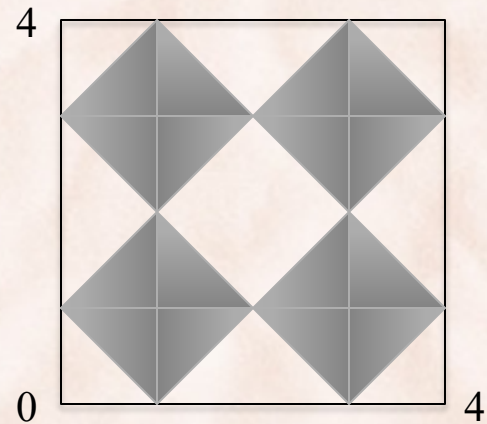
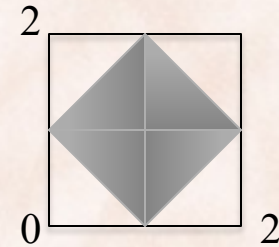
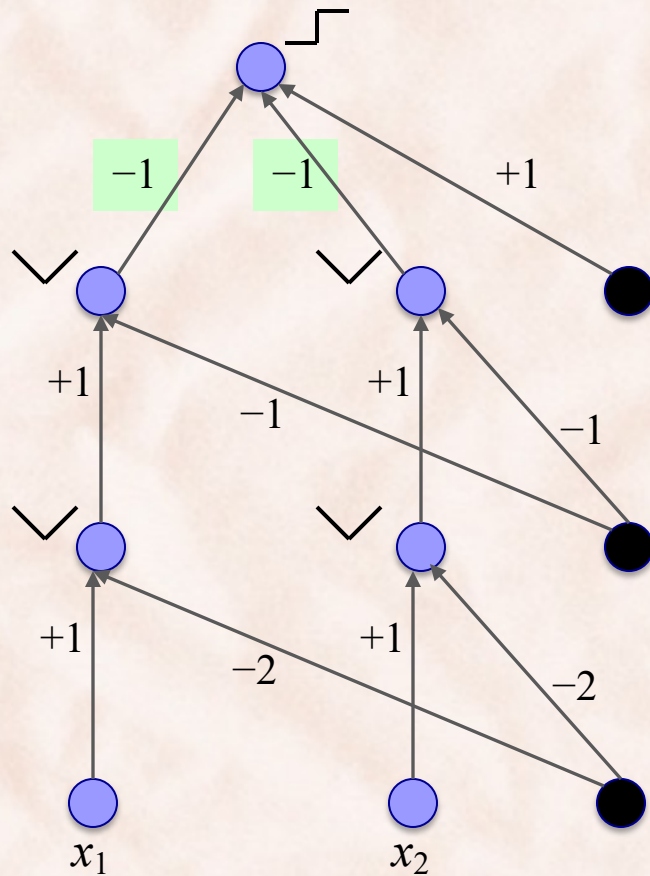


# Space foldings

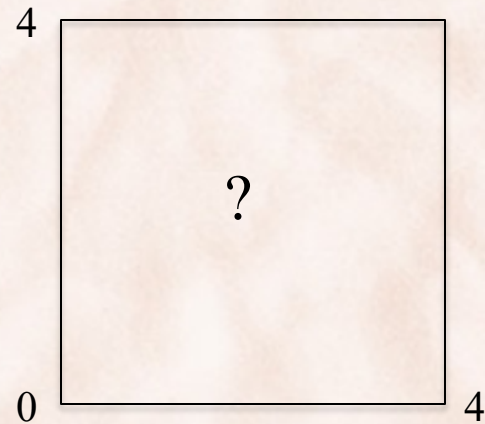
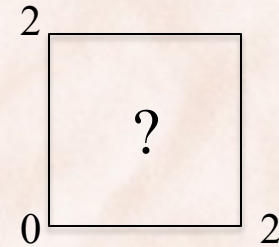
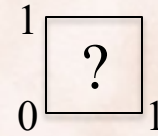
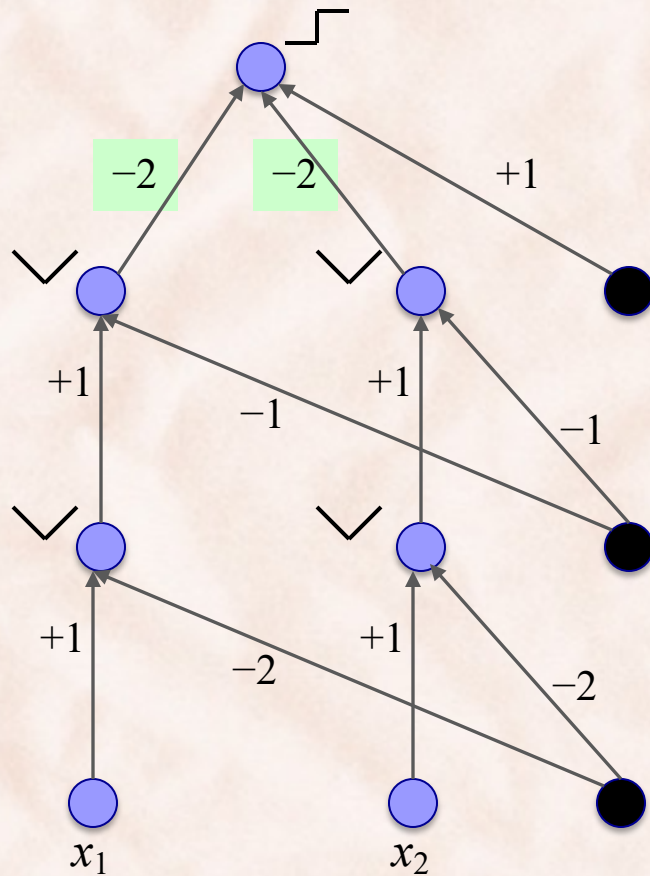
- Each hidden layer of a deep neural network can be associated with a folding operator.



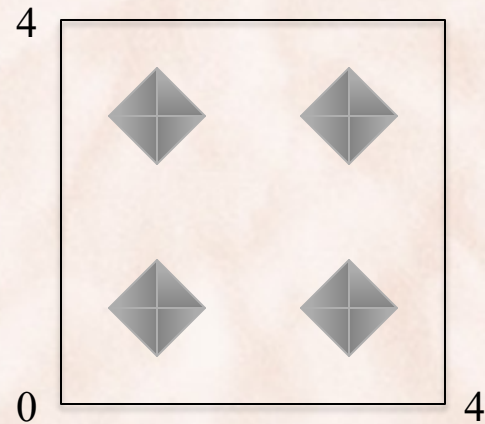
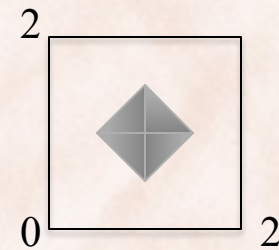
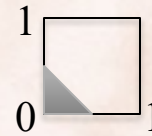
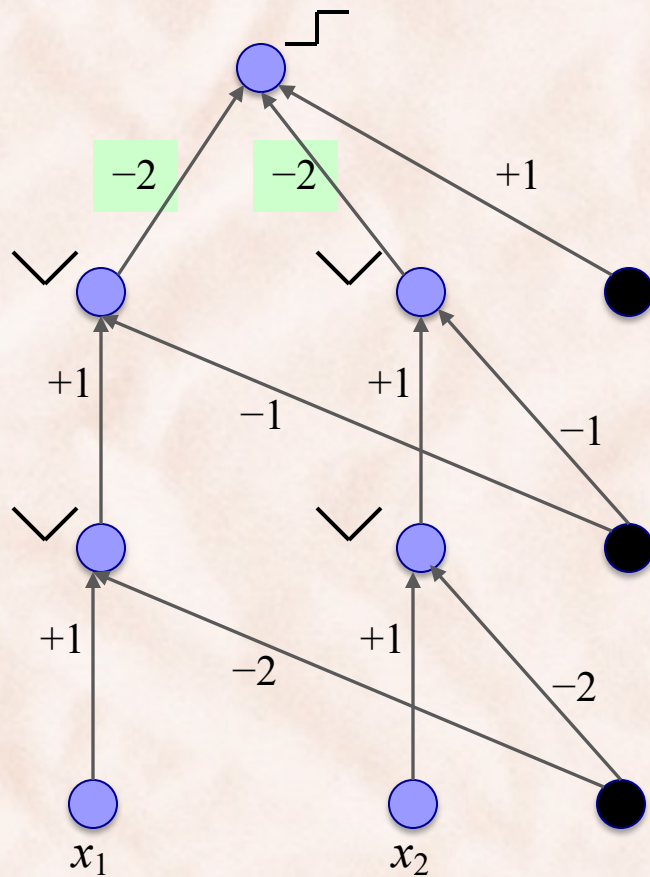
# Folding Example



# Folding Example



# Folding Example



# Space Foldings

[Montufar et al., NIPS'14]

---

- **Each hidden layer** of a deep neural network can be associated with a **folding operator**:
  - Each hidden layer folds the space of activations of the previous layer.
  - In turn, a deep neural network effectively folds its input-space recursively, starting with the first layer.
- Any function computed on the final folded space will apply to all the collapsed subsets identified by the map corresponding to the succession of foldings.
- In a deep model, any partitioning of the last layer's image-space is replicated in all input-space regions which are *identified* by the succession of foldings.

# Space Foldings

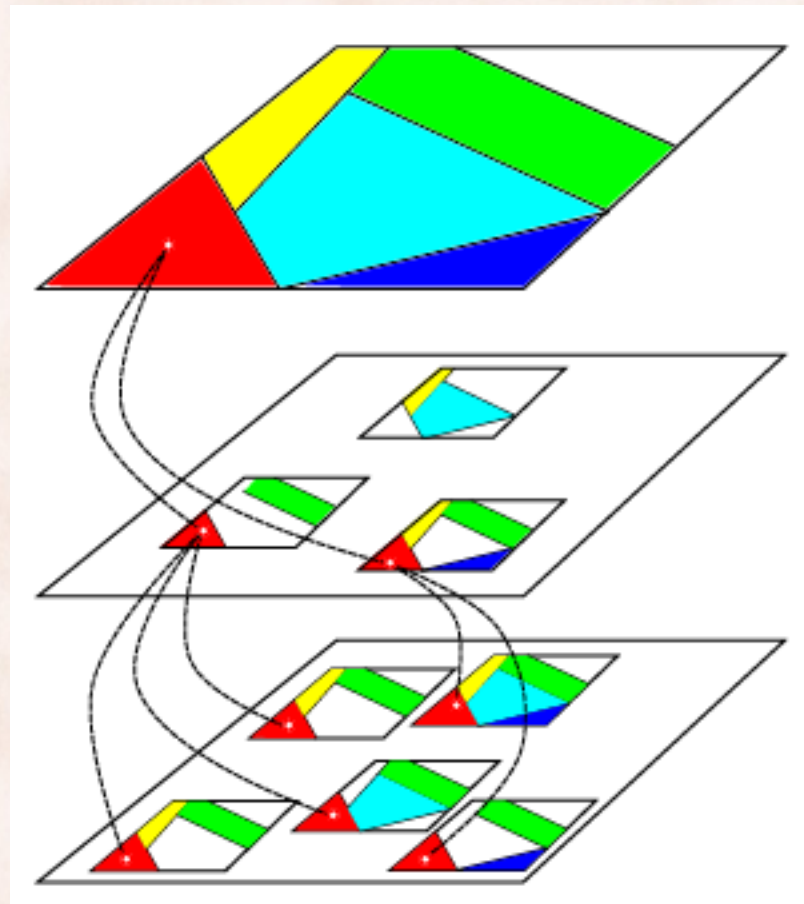
[Montufar et al., NIPS'14]

---

- Space foldings are not restricted to foldings along coordinate axes and they do not have to preserve lengths:
  - The space is folded depending on the orientations and shifts encoded in:
    - The input weights  $\mathbf{W}$  and biases  $\mathbf{b}$ .
    - The nonlinear activation function used at each hidden layer.
  - The sizes and orientations of *identified* input-space regions may differ from each other.
  - For activation functions which are not piece-wise linear, the folding operations may be even more complex.

# Space Foldings

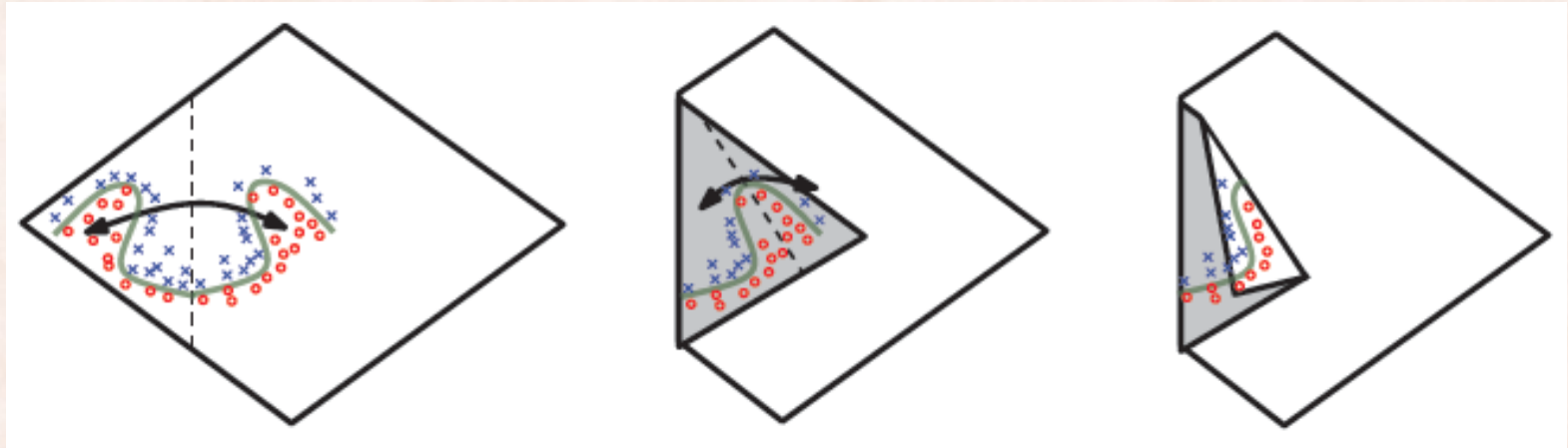
[Montufar et al., NIPS'14]



# Space Foldings

[Montufar et al., NIPS'14]

- Space folding of 2-D space in a non-trivial way:
  - The folding can potentially identify symmetries in the boundary that it needs to learn.



# Readings

---

- [Chapter 6](#) on Neural Networks in the NLP textbook.

# Deep vs. Shallow Architectures

[Bengio, FTML'09]

- When a function can be compactly represented by a deep architecture, it might need a very large architecture to be represented by an insufficiently deep one.

A two-layer circuit of logic gates can represent any Boolean function (Mendelson, 1997). Any Boolean function can be written as a sum of products (disjunctive normal form: AND gates on the first layer with optional negation of inputs, and OR gate on the second layer) or a product of sums (conjunctive normal form: OR gates on the first layer with optional negation of inputs, and AND gate on the second layer). To understand the limitations of shallow architectures, the first result to consider is that with depth-two logical circuits, most Boolean functions require an *exponential* (with respect to input size) number of logic gates (Wegener, 1987) to be represented.

More interestingly, there are functions computable with a polynomial-size logic gates circuit of depth  $k$  that require exponential size when restricted to depth  $k - 1$  (Håstad, 1986). The proof of this theorem relies on earlier results (Yao, 1985) showing that  *$d$ -bit parity circuits of depth 2 have exponential size*. The  *$d$ -bit parity function* is defined as usual:

$$\text{parity} : (b_1, \dots, b_d) \in \{0, 1\}^d \mapsto \begin{cases} 1 & \text{if } \sum_{i=1}^d b_i \text{ is even} \\ 0 & \text{otherwise.} \end{cases}$$

# Deep vs. Shallow Architectures

[Bengio, FTML'09]

- Many of the results for Boolean circuits can be generalized to architectures whose computational elements are *linear threshold* units i.e. Mc-Culloch & Pitts neurons:

$$f(\mathbf{x}) = \mathbf{1}[\mathbf{w}^T \mathbf{x} + b \geq 0]$$

- *Monotone weighted threshold circuits* = multi-layer neural networks with linear threshold units and positive weights.

**Theorem 2.1.** *A monotone weighted threshold circuit of depth  $k - 1$  computing a function  $f_k \in \mathcal{F}_{k,N}$  has size at least  $2^{cN}$  for some constant  $c > 0$  and  $N > N_0$  (Håstad & Goldmann, 1991).*

The class of functions  $\mathcal{F}_{k,N}$  is defined as follows. It contains functions with  $N^{2k-2}$  inputs, defined by a depth  $k$  circuit that is a tree. At the leaves of the tree there are unnegated input variables, and the function value is at the root. The  $i$ -th level from the bottom consists of AND gates when  $i$  is even and OR gates when  $i$  is odd. The fan-in at the top and bottom level is  $N$  and at all other levels it is  $N^2$ .