ITCS 5356 Intro to Machine Learning

Feed-Forward Neural Networks

Backpropagation

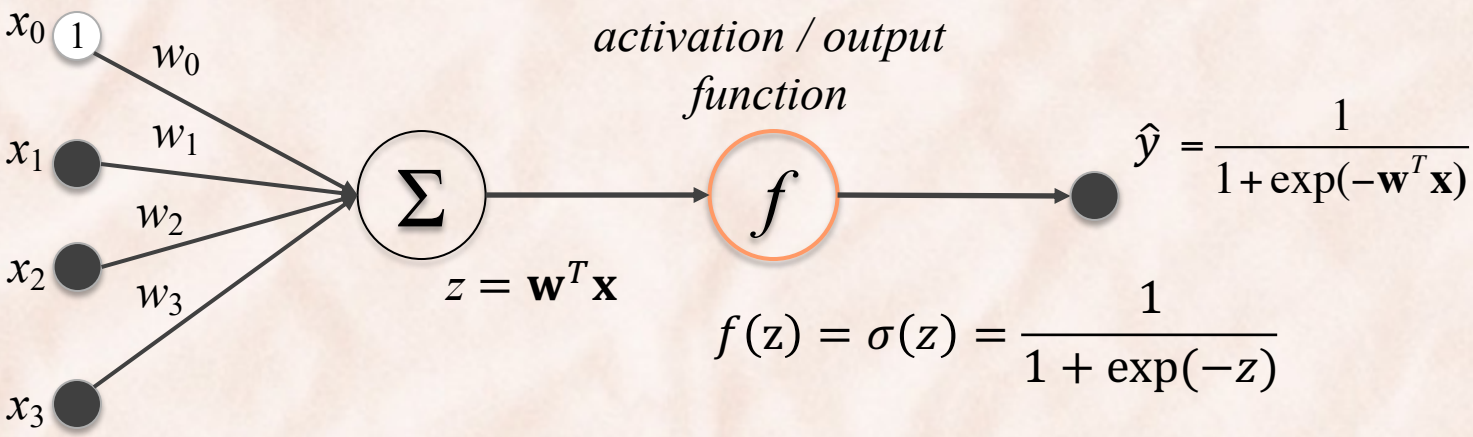
Deep Learning

Razvan C. Bunescu

Computer Science @ CCI

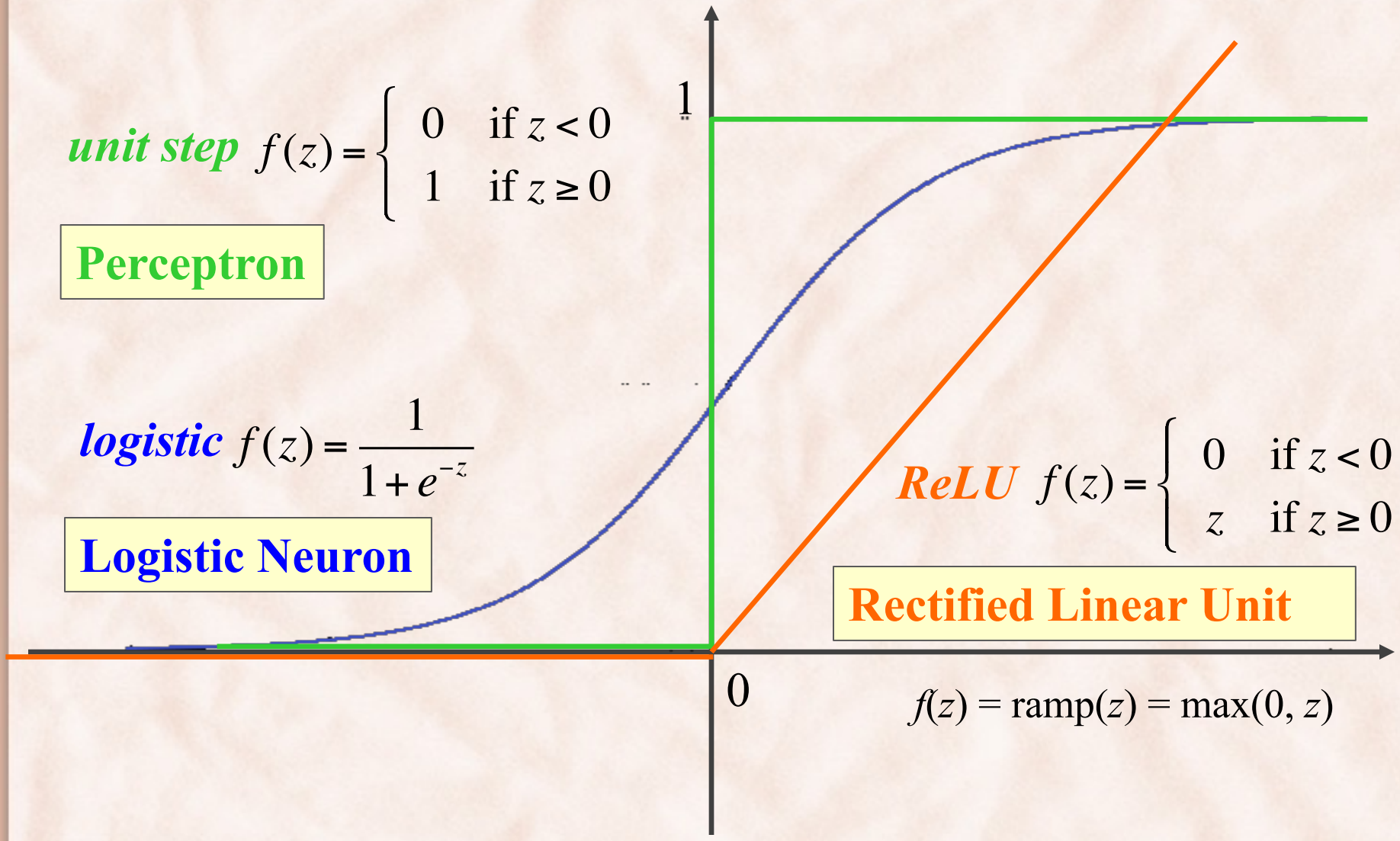
razvan.bunescu@charlotte.edu

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



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 σ 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 $i = 1, ..., N$, such that:

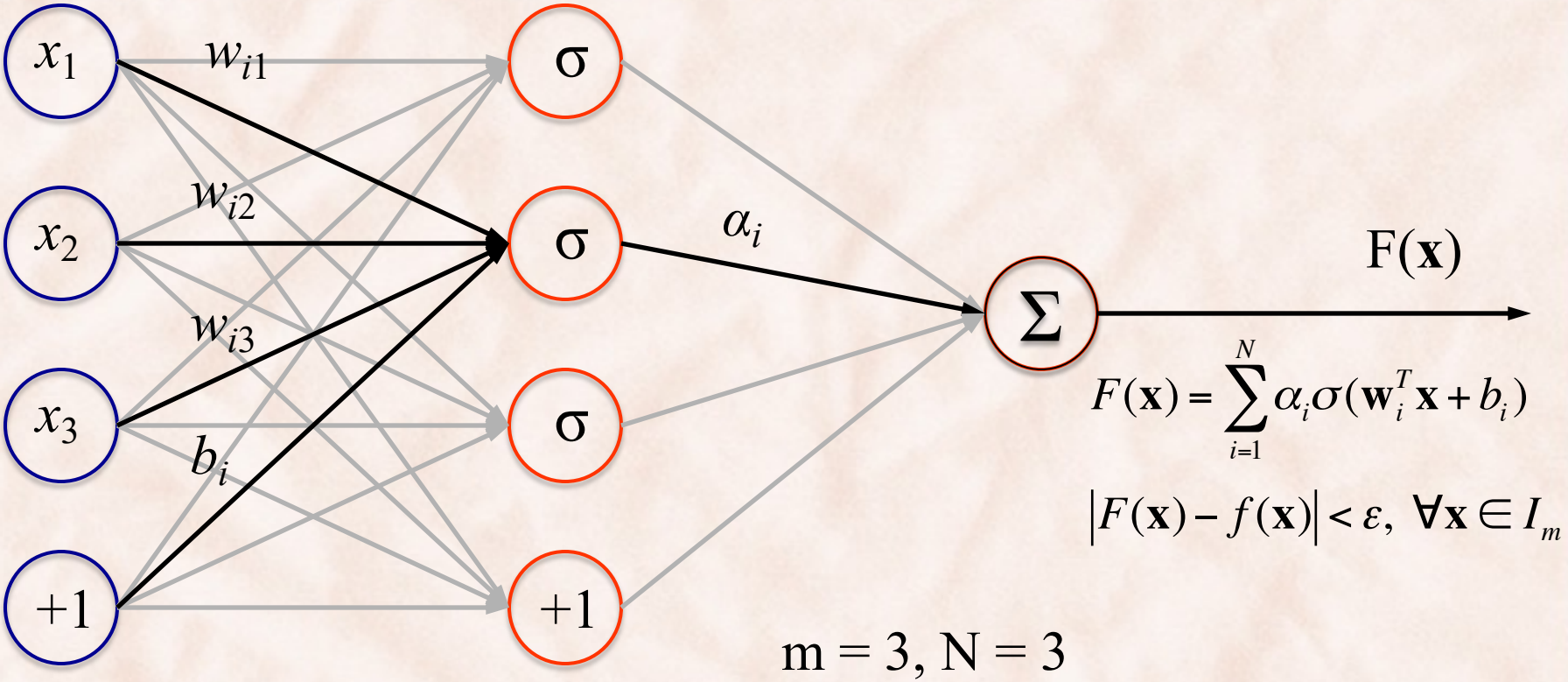
$$\left|F(\mathbf{x}) - f(\mathbf{x})\right| < \varepsilon, \quad \forall \mathbf{x} \in I_m$$

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)



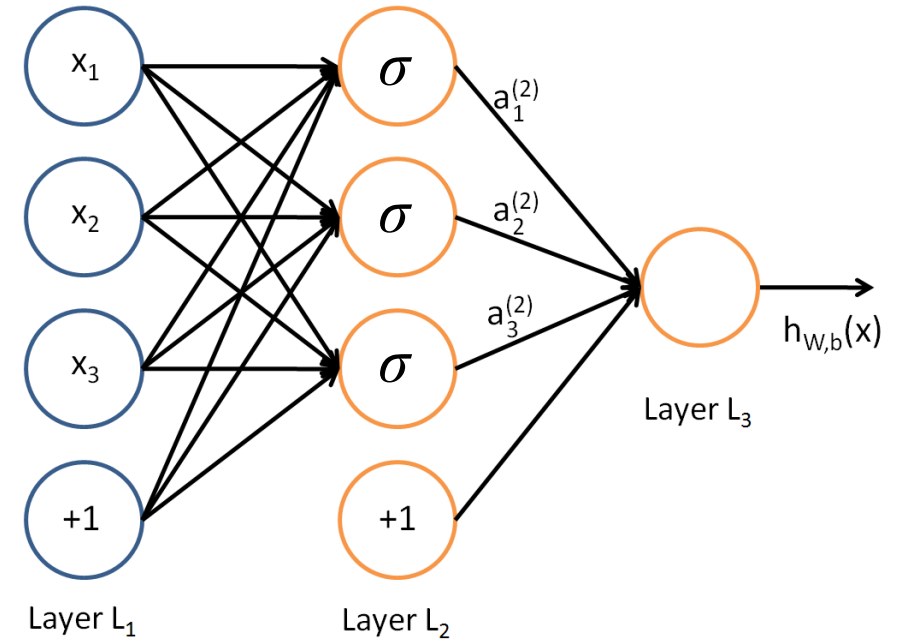
m = 3, N = 3

$\mathbf{x} = [x_1, x_2, x_3]$

$\mathbf{w}_i = [w_{i1}, w_{i2}, w_{i3}]$

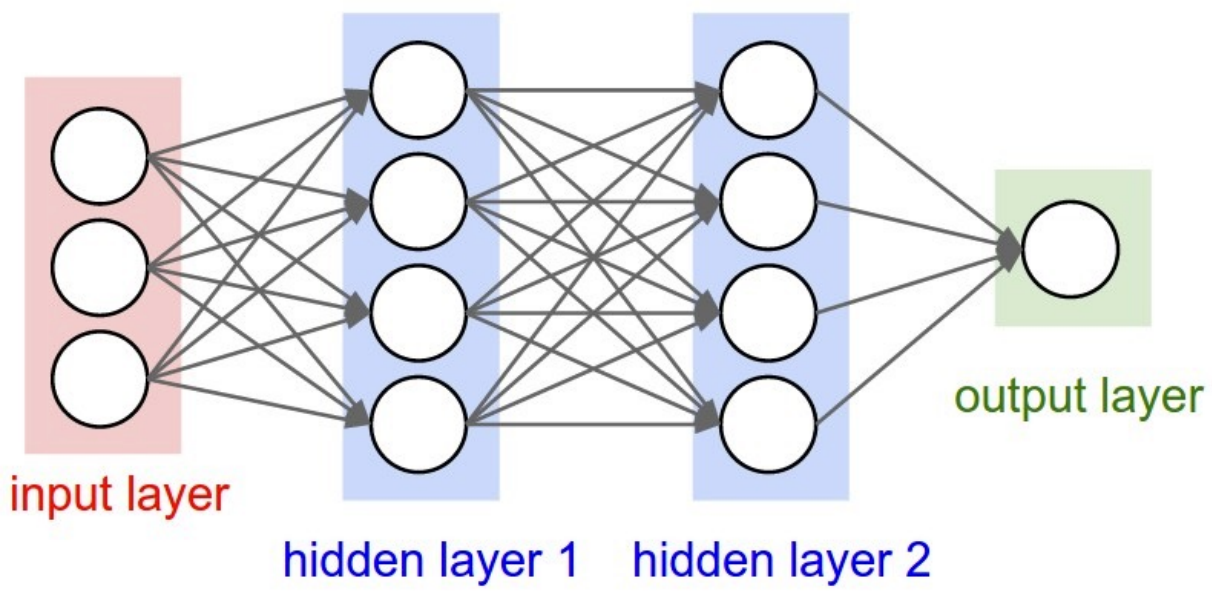
Neural Network Model

- 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

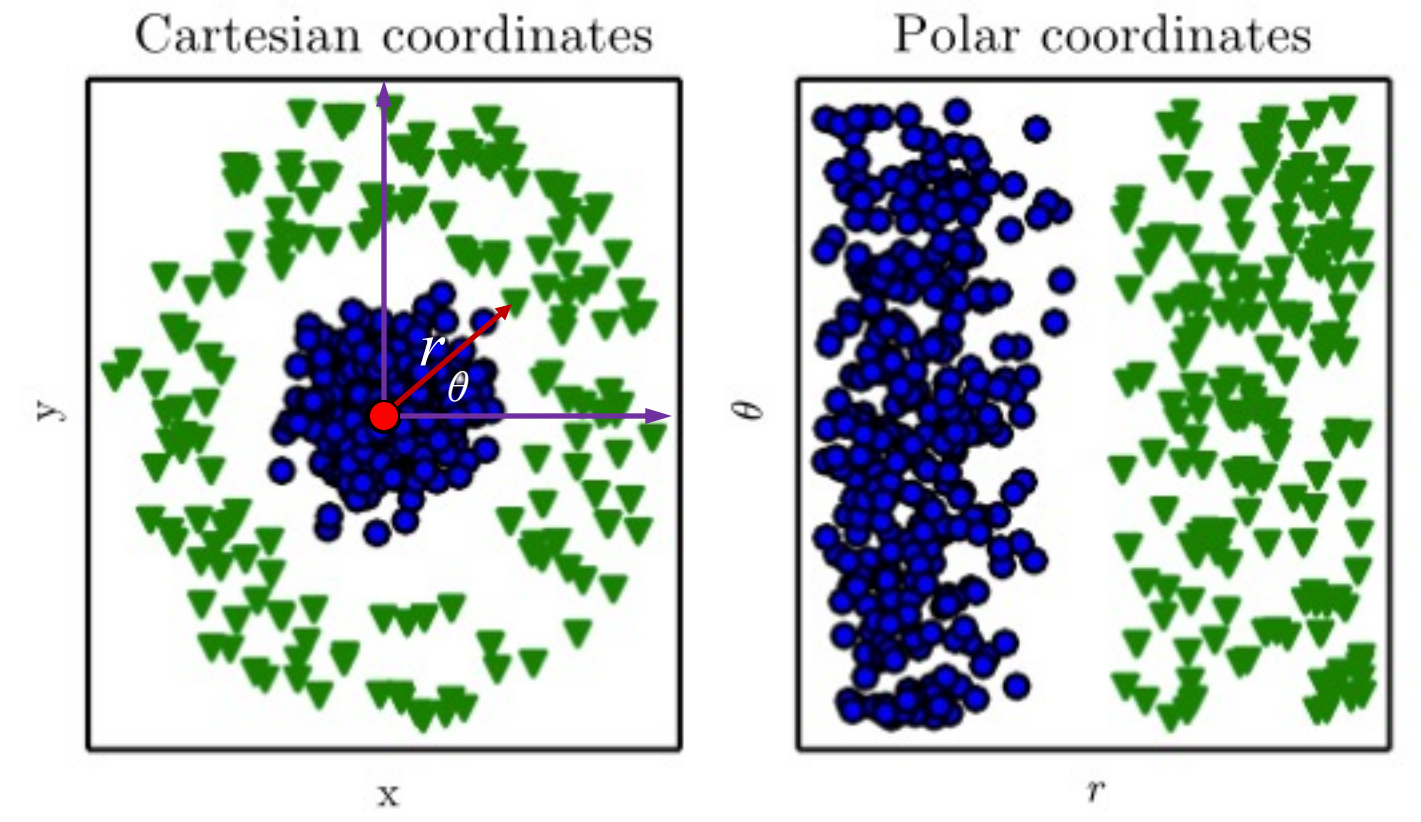


10 10 5 1

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 = **110 params**.
2. For the 5 neurons on hidden layer 2, we need 5 * 11 = **55 params**.
3. For the output neurons, we need 5 + 1 = **6 params**.

The Importance of Representation

http://www.deeplearningbook.org



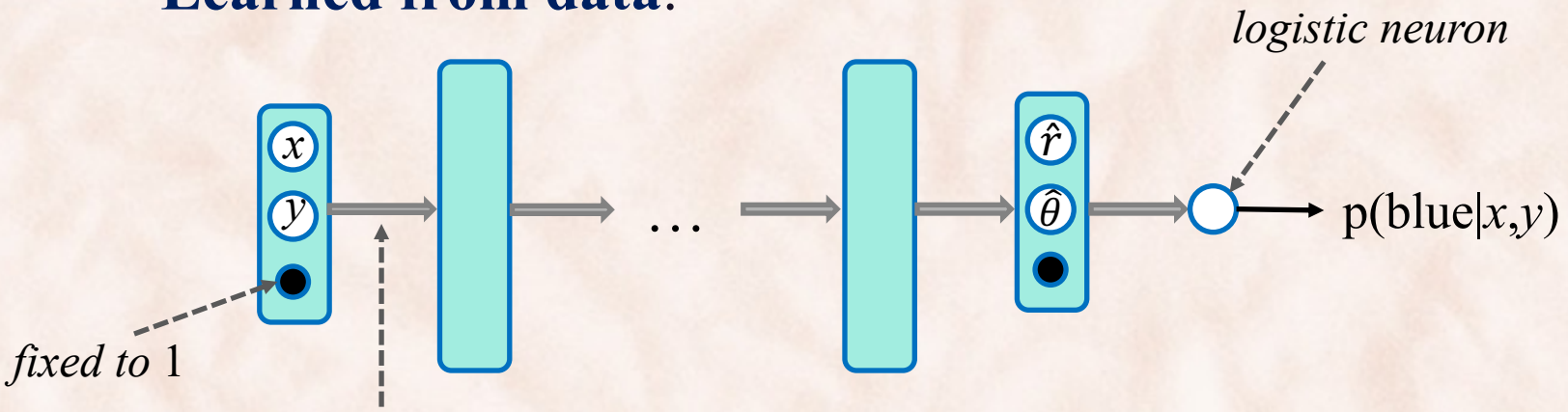
From Cartesian to Polar Coordinates

- **Manually engineered**:

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

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

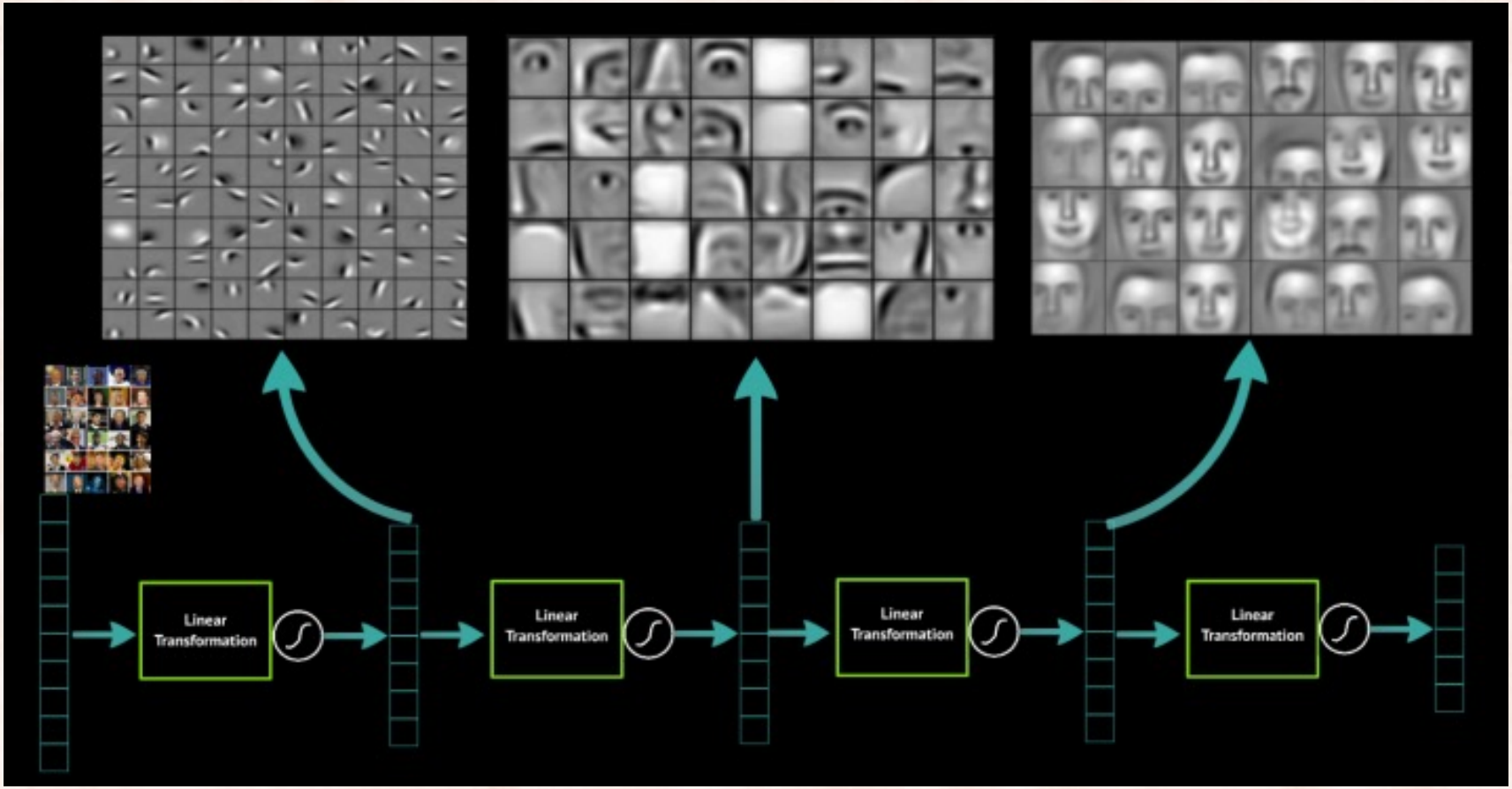
- **Learned from data**:



Fully connected layers: linear transformation W + *element-wise nonlinearity f* => *f*(W**x**)

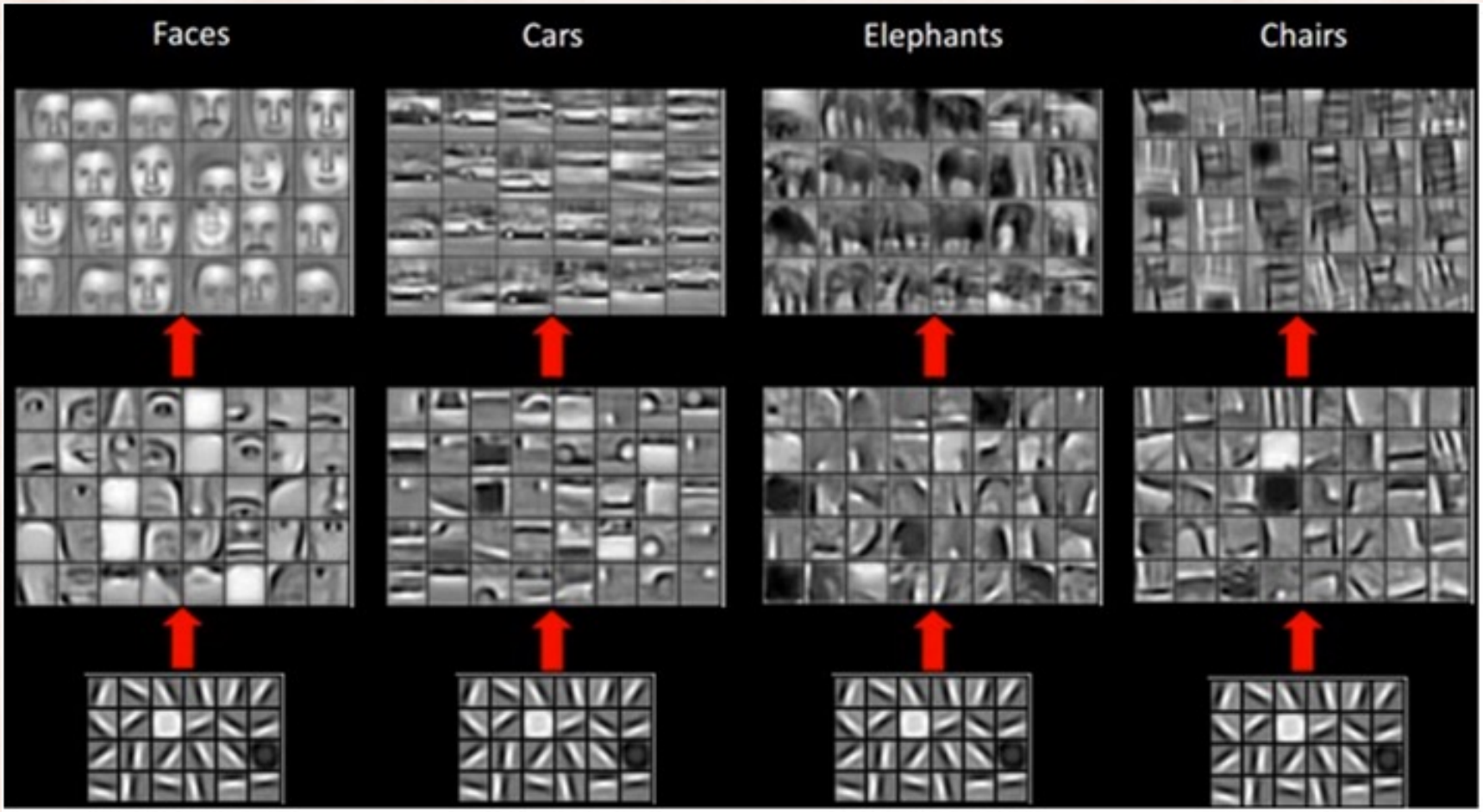
Representation Learning: Images

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



Representation Learning: Images

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

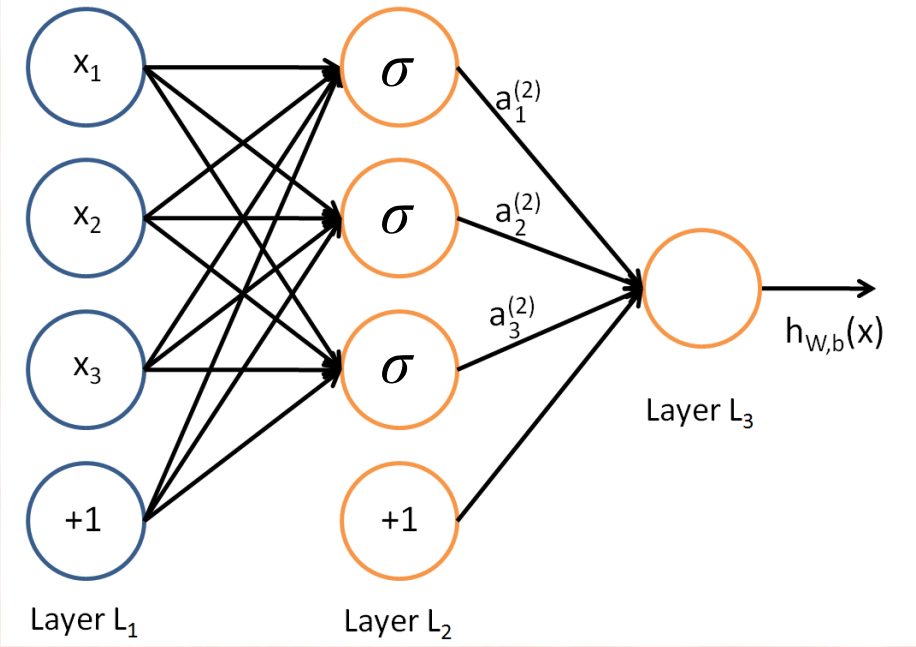


A Rapidly Evolving Field

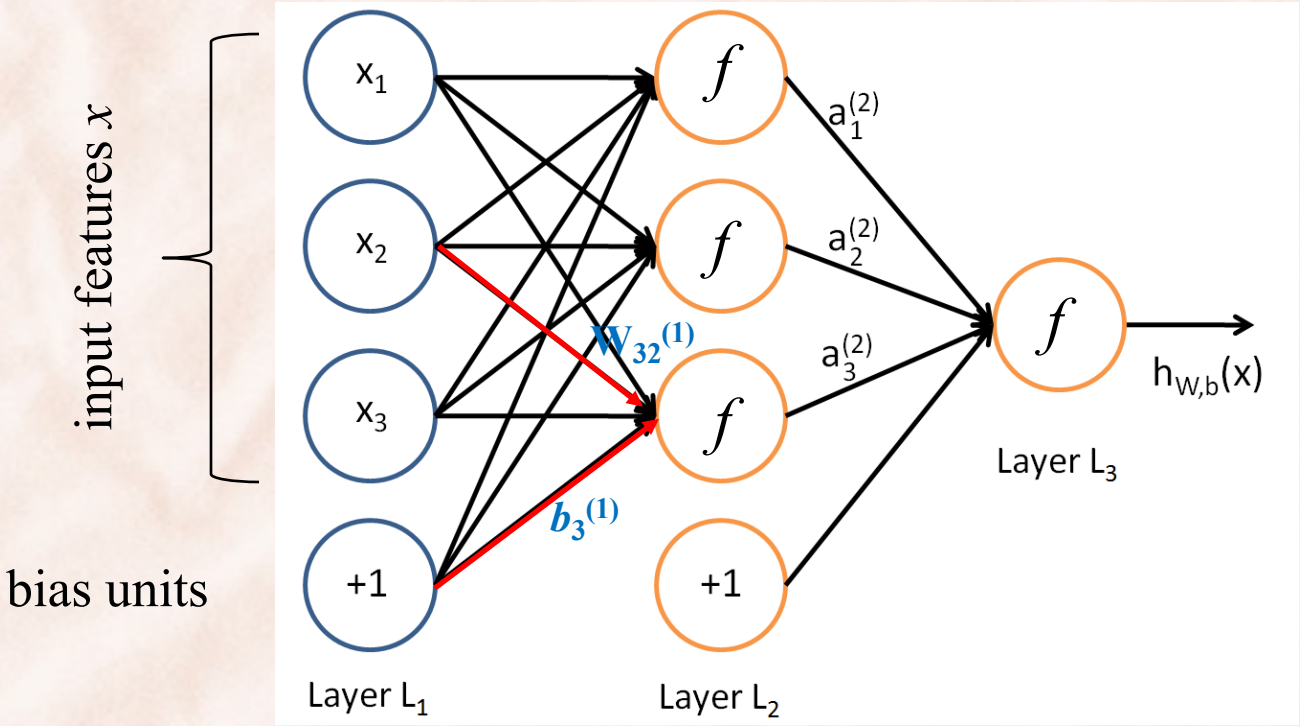
- Used to think that training deep networks requires **greedy layer-wise pretraining**:
  - Unsupervised learning of representations with **auto-encoders** (2012).
- Better random **weight initialization** schemes now allow training deep networks from scratch.
- **Batch normalization** allows for training even deeper models (2014).
  - Sometimes replaced 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).

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_1 is the input layer, L_3 is the output layer
- (**W**, **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:

$a_i^{(l)} = f(z_i^{(l)})$ where $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j + b_i^{(1)}$

Forward Propagation

- Forward propagation (unrolled):

$$
\begin{aligned}
a_1^{(2)} &= f(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}) \\
h_{W,b}(x) &= a_1^{(3)} = f(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)})
\end{aligned}
$$

- Forward propagation (compressed):

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

- Element-wise application:

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

Forward Propagation

- Forward propagation (compressed):

$$
\begin{aligned}
z^{(2)} &= W^{(1)}x + b^{(1)} \\
a^{(2)} &= f(z^{(2)}) \\
z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{aligned}
$$

- Composed of two ***forward propagation steps***:

$$
\begin{aligned}
z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\
a^{(l+1)} &= f(z^{(l+1)})
\end{aligned}
$$

Forward Propagation for FCNs: Regression

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

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

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

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

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

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

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

Backpropagation for FCNs for Regression: 1 example

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

1. For output layer, compute:

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

true label

2. For $l = n_l, n_l-2, n_l-3, \ldots, 2$ 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 $J(W, b, x, y)$

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

Backpropagation for FCNs for Regression: *m* examples

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

1. For output layer, compute:

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

$\mathbf{y}$ ..... *true label vector*

2. For $l = n_l, n_l-2, n_l-3, \ldots, 2$ 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 $J(W, b, x, y)$

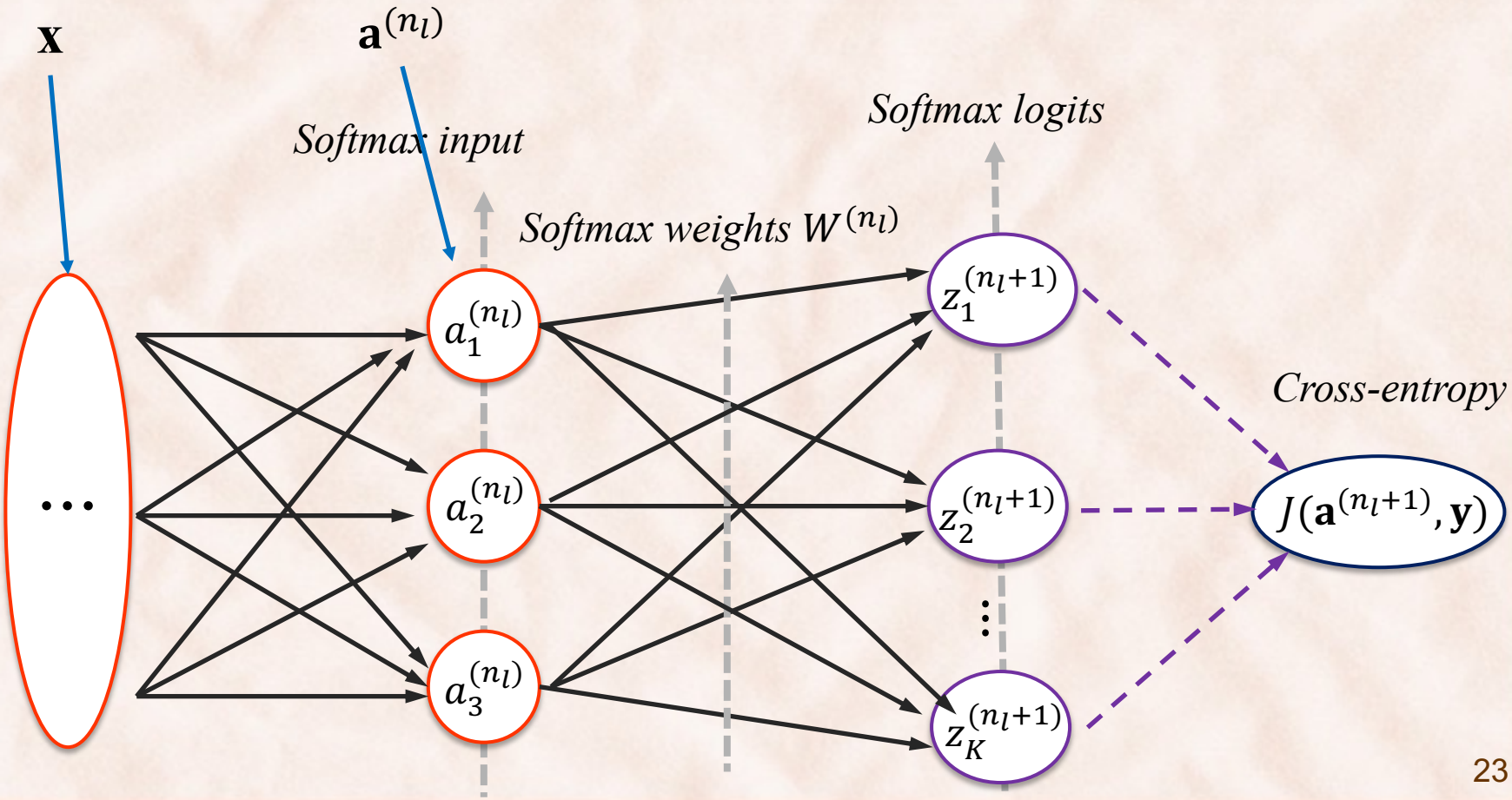
$$\nabla_{W^{(l)}} J = \delta^{(l+1)}\left(a^{(l)}\right)^T / m \qquad \nabla_{b^{(l)}} J = \delta^{(l+1)}.\text{col\_avg()}$$

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

Multinomial Softmax

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



Forward Propagation for FCNs: Classification

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

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

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

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

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

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

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

Backpropagation Algorithm: Softmax (1)

1. Feedforward pass on $\mathbf{x}$ to compute activations $\mathbf{a}^{(l)}$ for layers $l = 1, 2, \ldots, n_l$.
2. Compute softmax outputs $\mathbf{a}^{(n_l+1)}$ and objective $J(\mathbf{a}^{(n_l+1)}, \mathbf{y})$.
3. Let $\mathbf{y} = [\delta_1(y), \delta_2(y), \ldots, \delta_K(y)]^{\mathrm{T}}$ 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 $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ compute:

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

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

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

Backpropagation Algorithm: Softmax for 1 Example

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

1. For softmax layer, compute:

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

one-hot label vector

2. For $l = n_l, n_l-1, n_l-2, \ldots, 2$ 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 $J(W, b, x, y)$

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

Backpropagation Algorithm: Softmax for Dataset of *m* Examples

- Feedforward to compute activations $a^{(l)} = f(z^{(l)})$ 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 $l = n_l, n_l-1, n_l-2, \ldots, 2$ 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 $J(W, b, x, y)$

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

if using L_2 regularization

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

Backpropagation Algorithm: Softmax for Dataset of *m* Examples

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

1. For softmax layer, compute:

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

K x m, where K is the # of classes

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

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

s_l x m, where s_l is the # neurons on layer l

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

$$\nabla_{W^{(l)}} J = \underset{s_{l+1} \text{x } m}{\delta^{(l+1)}} \left(\underset{s_l \text{ x } m}{a^{(l)}}\right)^T / m + \alpha W^{(l)}$$

s_{l+1} x s_l, where s_l is the # neurons on layer l

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

np.mean(axis = 1)

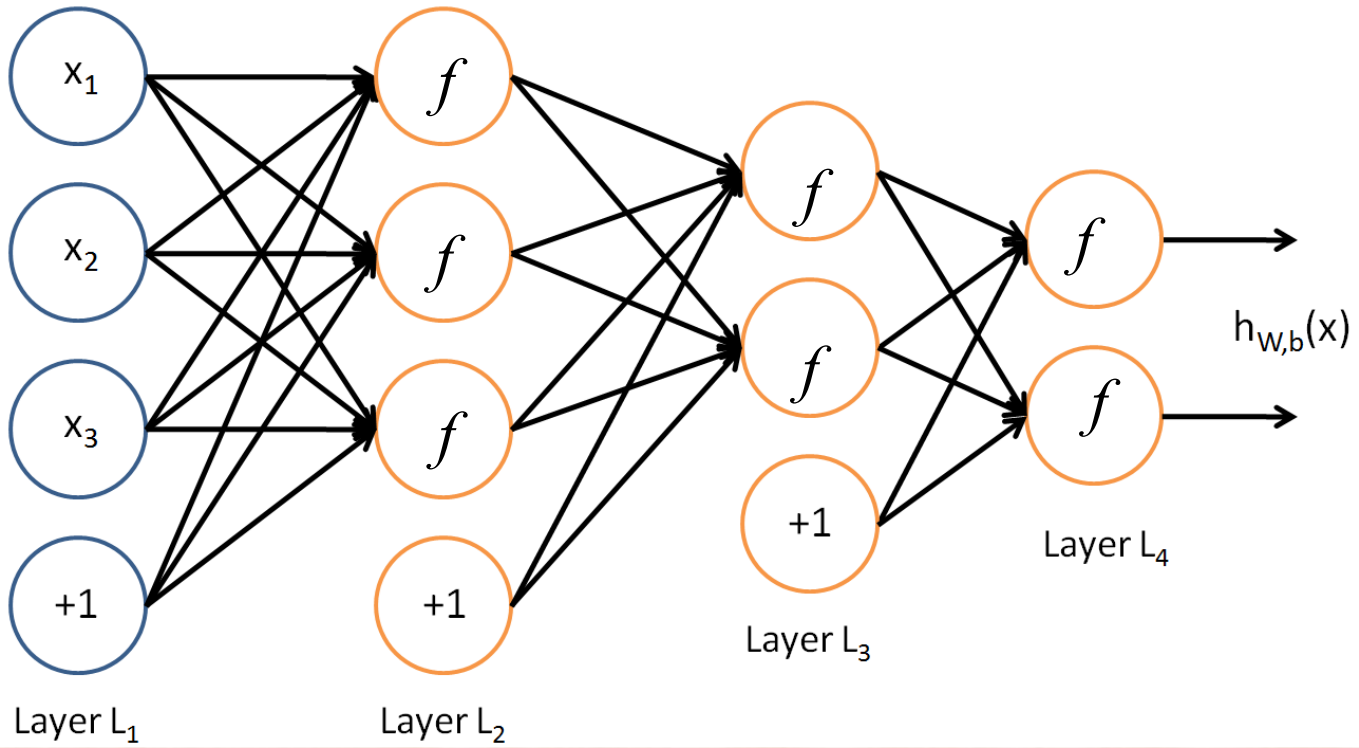
s_{l+1} x 1, where s_{l+1} is the # neurons on layer l+1

Softmax Regression Cost: From 1 to *m* examples

- Ground truth vector **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:
  - $J(\mathrm{W}, b, x, y) = -\ln p(y|\mathrm{W}, b, x) = -\sum_{k=1}^{K} \delta_k(y) \ln p(C_k|x)$
- 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 **y** and the log of $\mathbf{a}^{(n_l+1)}$
    - $J(\mathrm{W}, b, x, y) = J\left(\mathbf{a}^{(n_l+1)}, \mathbf{y}\right) = -\mathbf{y}^T \ln \mathbf{a}^{(n_l+1)} = -sum(\mathbf{y} \circ \ln \mathbf{a}^{(n_l+1)})$
- When vectorized for *m* examples + regularization, when **y** is the ground-truth matrix and **a** is the matrix of softmax probabilities of all *m* examples:
  - $J(\mathrm{W}, b) = J\left(\mathbf{a}^{(n_l+1)}, \mathbf{y}\right) = -\frac{1}{m} sum(\mathbf{y} \circ \ln \mathbf{a}^{(n_l+1)}) + \frac{\alpha}{2} \|W\|^2$

Multiple Hidden Units, Multiple Outputs

- Write down the forward propagation steps for:



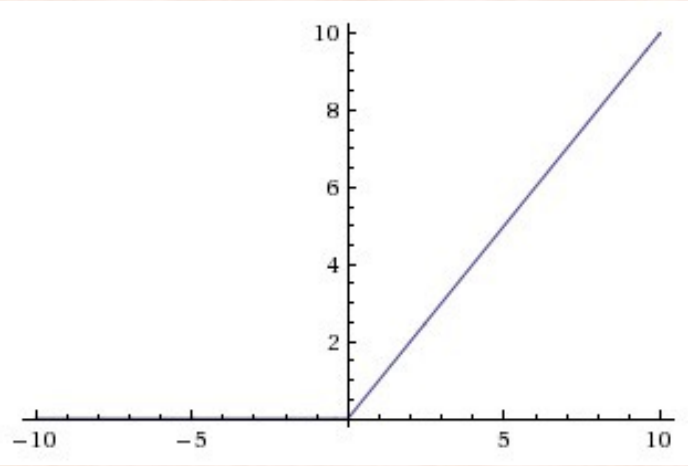
ReLU and Generalizations

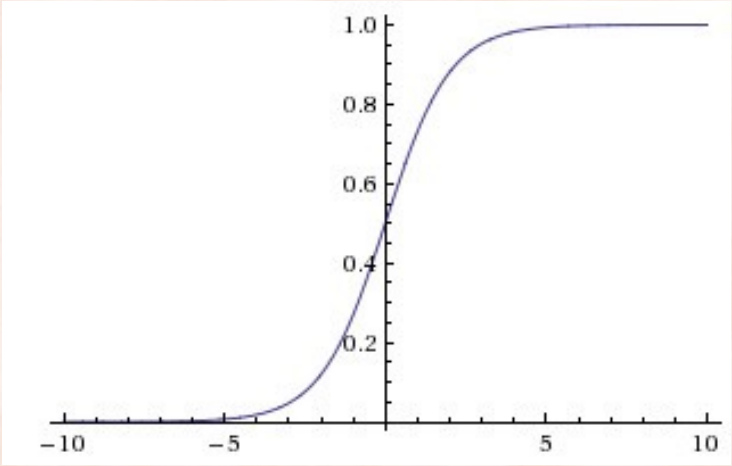
- It has become more common to use piecewise linear activation functions for hidden units:
  - **ReLU**: the rectifier activation $g(z) = \max\{0, z\}$.
  - **Absolute value ReLU**: $g(z) = |z|$.
  - **Maxout**: $g(a_1, ..., a_k) = \max\{a_1, ..., a_k\}$.
    - needs k weight vectors instead of 1.
  - **Leaky ReLU**: $g(a) = \max\{0, a\} + \alpha \min(0, a)$.

$\Rightarrow$ the network computes a *piecewise linear function* (up to the output activation function).

ReLU vs. Sigmoid and Tanh

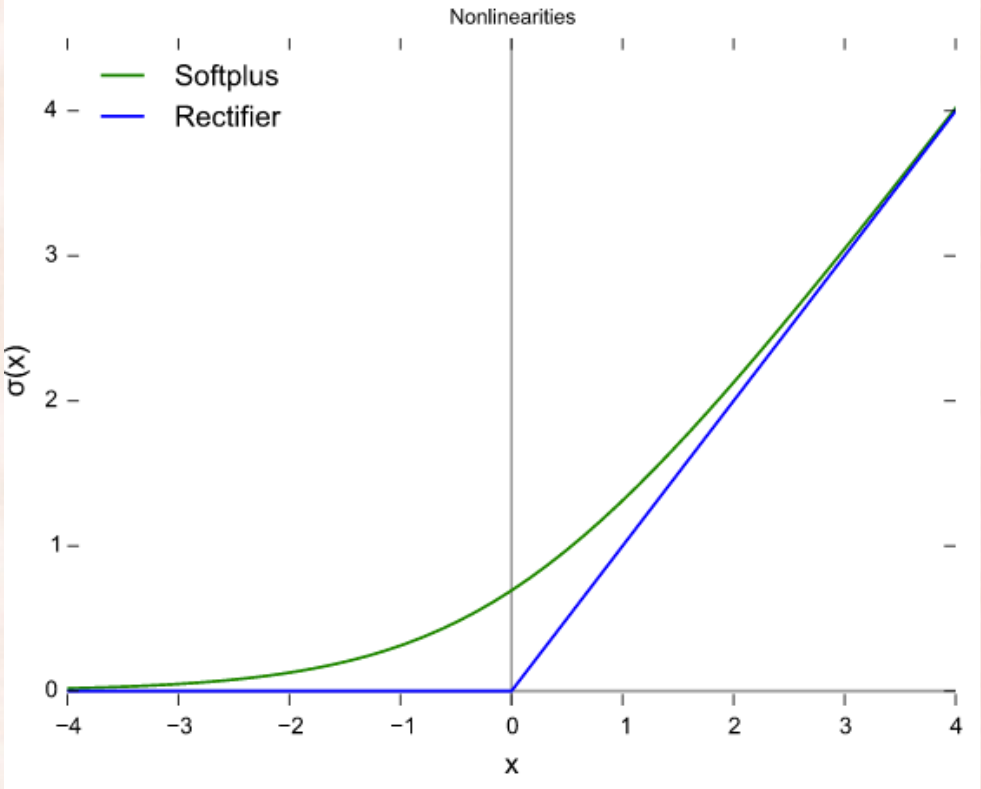
- Sigmoid and Tanh saturate for values not close to 0:
  - "kill" gradients, bad behavior for gradient-based learning.
- ReLU does not saturate for values > 0:
  - greatly accelerates learning, fast implementation.
  - fragile during training and can "die", due to 0 gradient:
    - initialize all *b*'s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





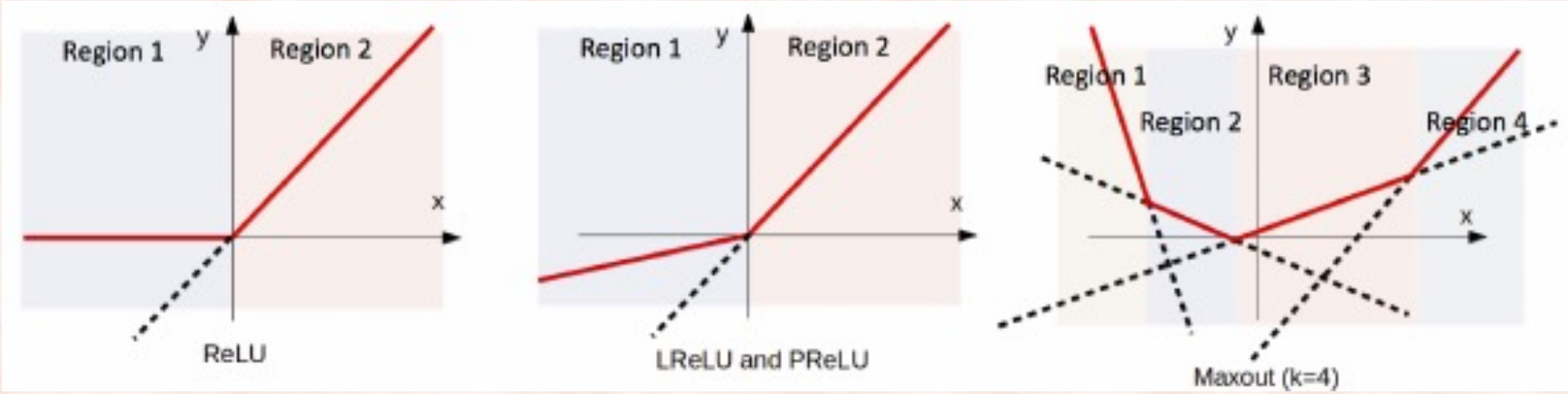
ReLU vs. Softplus

- 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.

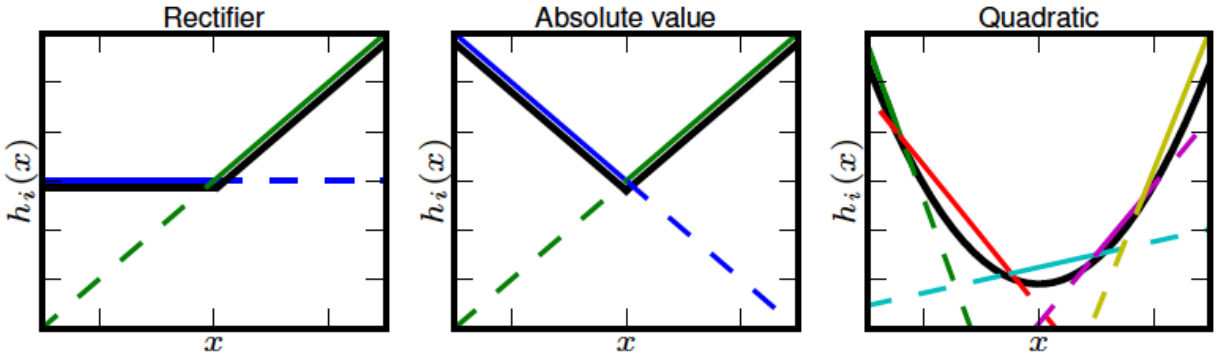


Figure 1. Graphical depiction of how the maxout activation function can implement the rectified linear, absolute value rectifier, and approximate the quadratic activation function. This diagram is 2D and only shows how maxout behaves with a 1D input, but in multiple dimensions a maxout unit can approximate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

Start Supplemental Material

Derivation of Backpropagation

Backpropagation for FCNs for Regression: 1 example

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

1. For softmax layer, compute:

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

true label

2. For $l = n_l, n_l-1, n_l-2, \ldots, 2$ 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 $J(W, b, x, y)$

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

Learning: Regression vs. Classification

- **Regression** => *loss* = squared error:

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

- **Classification** => *loss* = negative log-likelihood:

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

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

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

Learning: Backpropagation for Regresion

- Regularized sum of squares error:

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

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

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

- 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):

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

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

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:

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

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

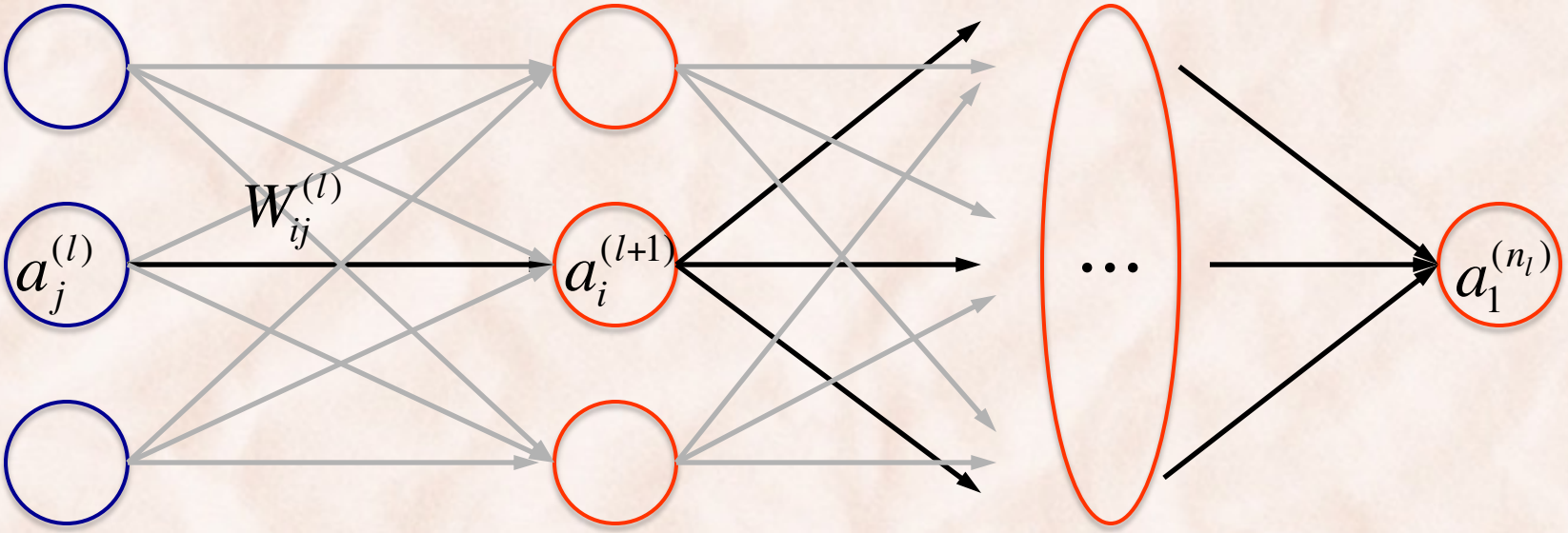
- Example:

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

$$g_1(x) = 3x$$

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

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



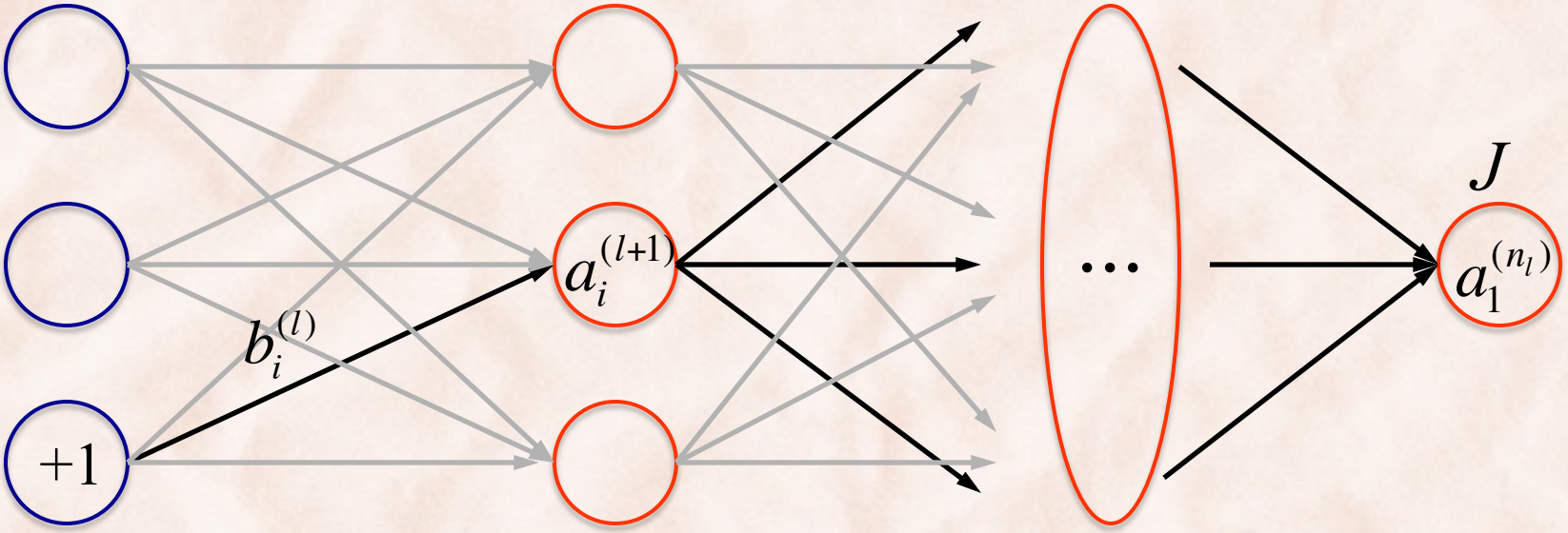
- J depends on $W_{ij}^{(l)}$ only through $a_i^{(l+1)}$, which depends on $W_{ij}^{(l)}$ only through $z_i^{(l+1)}$.

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

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

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

Backpropagation: $b_i^{(l)}$



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

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

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

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

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 ?

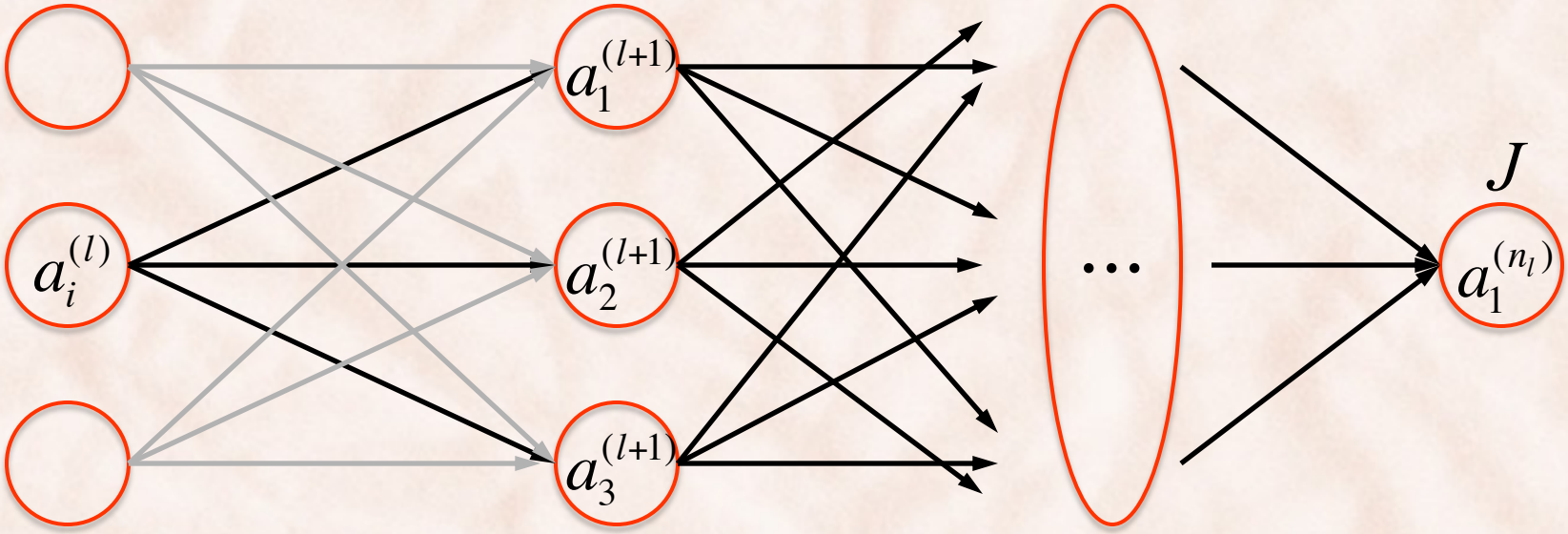
$$\frac{\partial J}{\partial b_i^{(l)}} = \underbrace{\frac{\partial J}{\partial a_i^{(l+1)}} \times \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}}}_{\delta_i^{(l+1)}} \times \underbrace{\frac{\partial z_i^{(l+1)}}{\partial b_i^{(l)}}}_{+1} = \delta_i^{(l+1)}$$

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 $a_1^{(l+1)}$, $a_2^{(l+1)}$, ...



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

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

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

- 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 δ's for the output layer:

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

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

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

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 $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ 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 $J(W, b, x, y)$

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

Backpropagation Algorithm: Vectorization for 1 Example

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

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

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

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

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

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

Backpropagation Algorithm: Vectorization for Dataset of *m* Examples

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

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

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

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

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

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

End Supplemental Material

Derivation of Backpropagation

Backpropagation: Logistic Regression

Bonus points

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

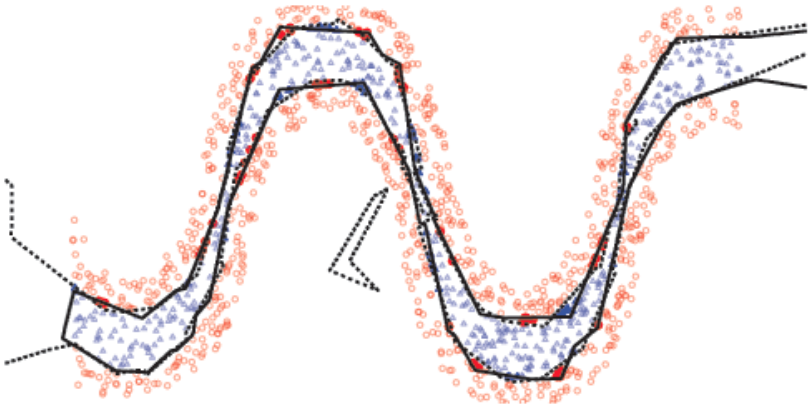
A deep network has significantly greater representational power than a shallow one.

Number of Linear Regions of Shallow vs. Deep Networks

[Montufar et a., NIPS'14]

Conjecture

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



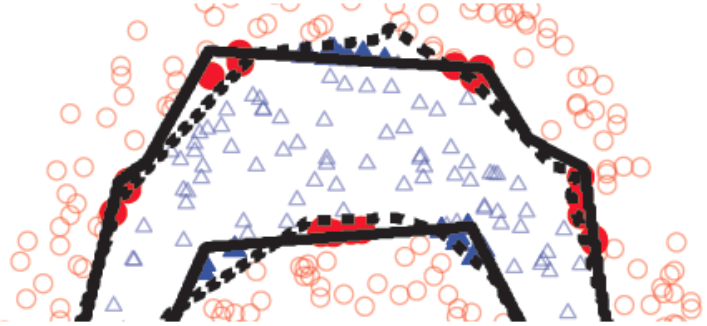


Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

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.

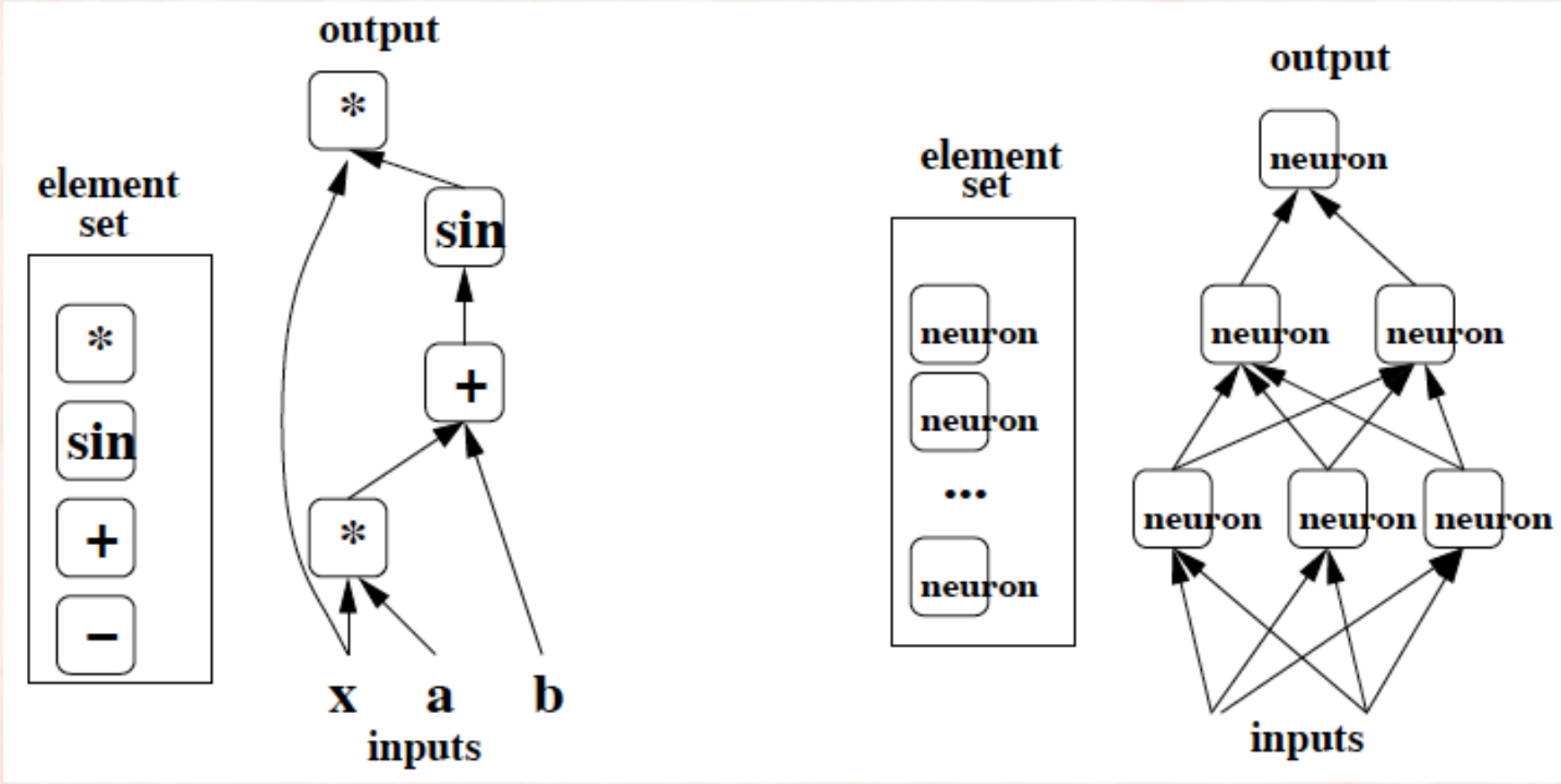
Graphs of Computations

- 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.

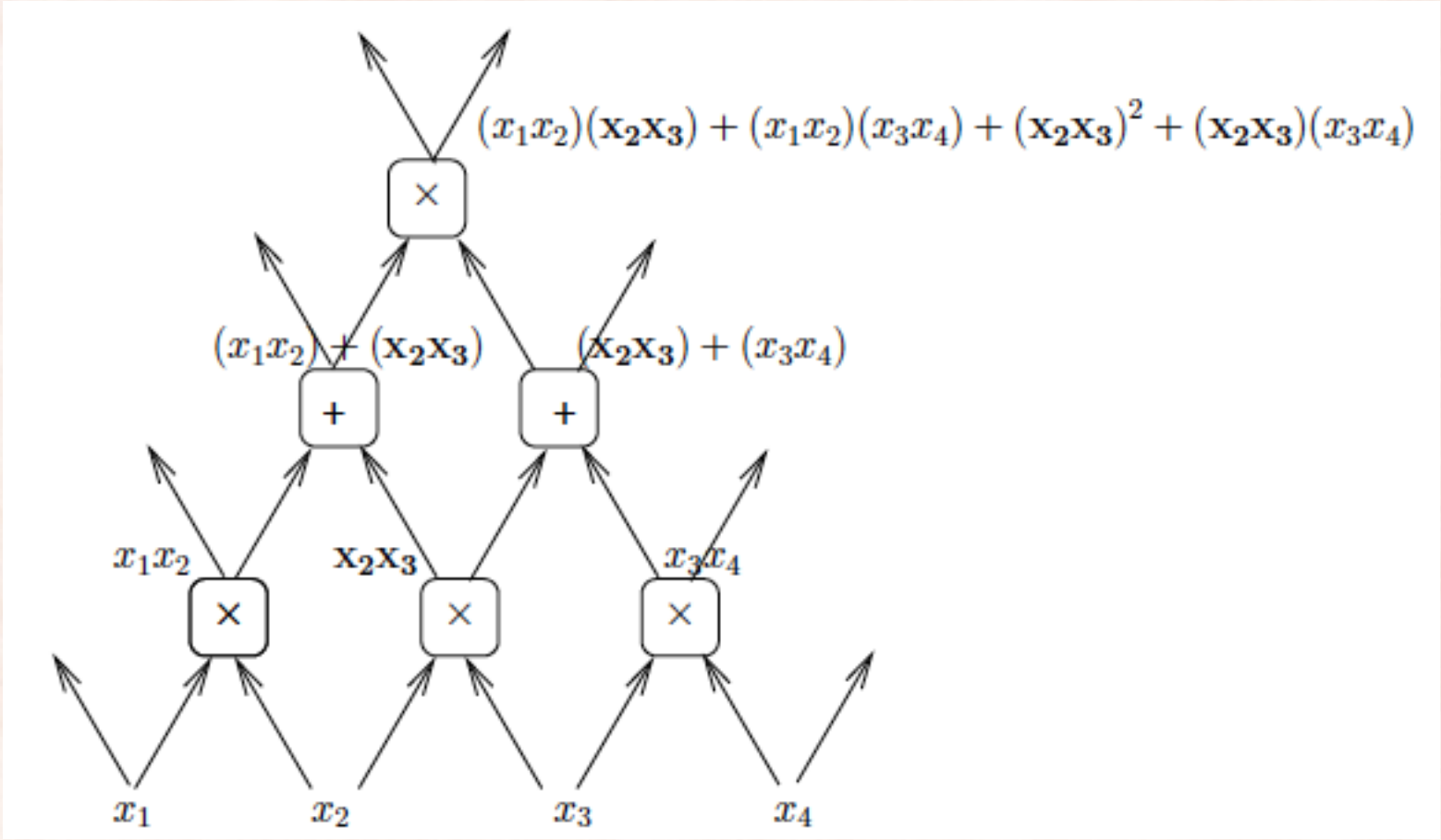
Functions as Graphs of Computations

[Bengio, FTML'09]



Polynomials as Graphs of Computations

[Bengio, FTML'09]



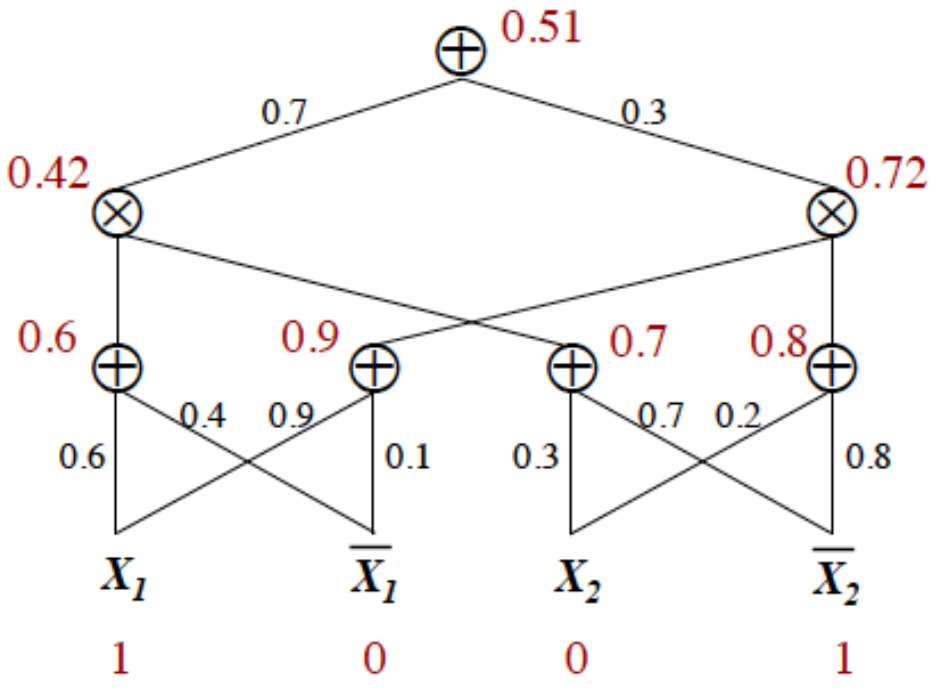
Sum-Product Networks (SPNs)

[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
$\overline{X}_1$	0
X_2	0
$\overline{X}_2$	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 $K(\mathbf{u}, \mathbf{v})$ 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 $K(\mathbf{x}, \mathbf{x}_i)$ for each prototype $\mathbf{x}_i$ (a selected representative training example) and matches the input vector $\mathbf{x}$ with the prototypes $\mathbf{x}_i$. The second level performs an affine combination $b + \sum_i \alpha_i K(\mathbf{x}, \mathbf{x}_i)$ to associate the matching prototypes $\mathbf{x}_i$ 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

[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:

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

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-Cullogh & 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.

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]

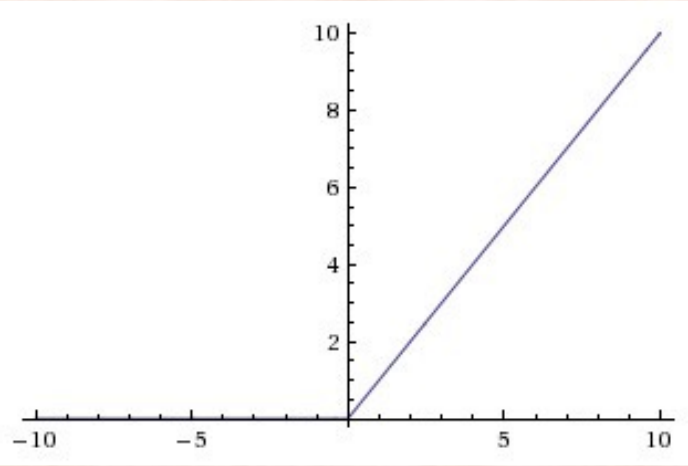
ReLU and Generalizations

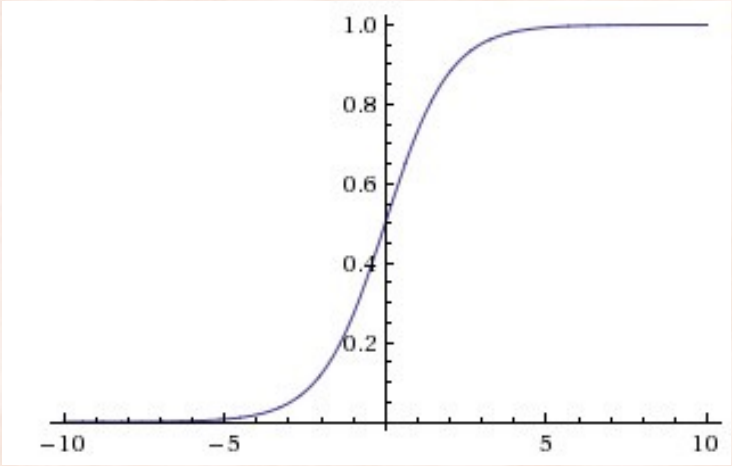
- It has become more common to use piecewise linear activation functions for hidden units:
  - **ReLU**: the rectifier activation $g(a) = \max\{0, a\}$.
  - **Absolute value ReLU**: $g(a) = |a|$.
  - **Maxout**: $g(a_1, ..., a_k) = \max\{a_1, ..., a_k\}$.
    - needs k weight vectors instead of 1.
  - **Leaky ReLU**: $g(a) = \max\{0, a\} + \alpha \min(0, a)$.

$\Rightarrow$ the network computes a *piecewise linear function* (up to the output activation function).

ReLU vs. Sigmoid and Tanh

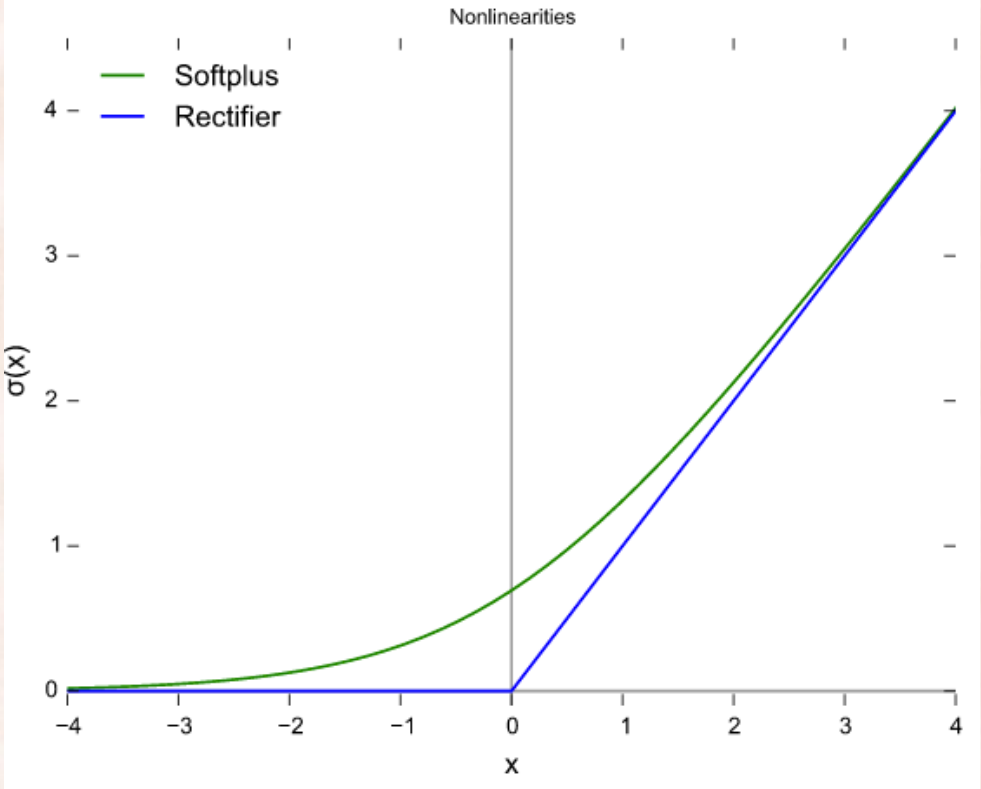
- Sigmoid and Tanh saturate for values not close to 0:
  - "kill" gradients, bad behavior for gradient-based learning.
- ReLU does not saturate for values > 0:
  - greatly accelerates learning, fast implementation.
  - fragile during training and can "die", due to 0 gradient:
    - initialize all *b*'s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





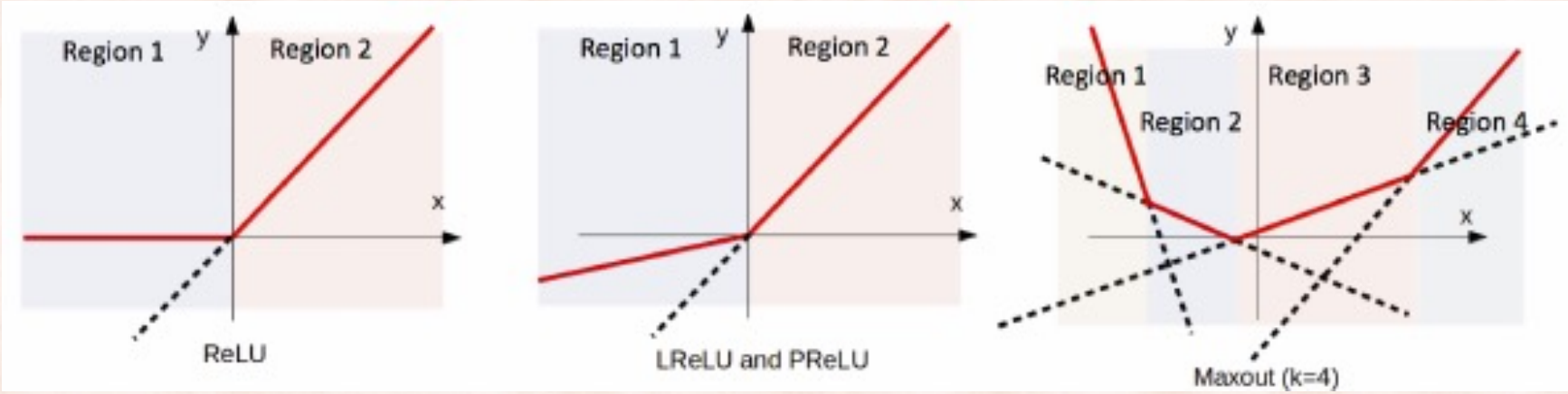
ReLU vs. Softplus

- Softplus $g(a) = \ln(1+e^a)$ 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.

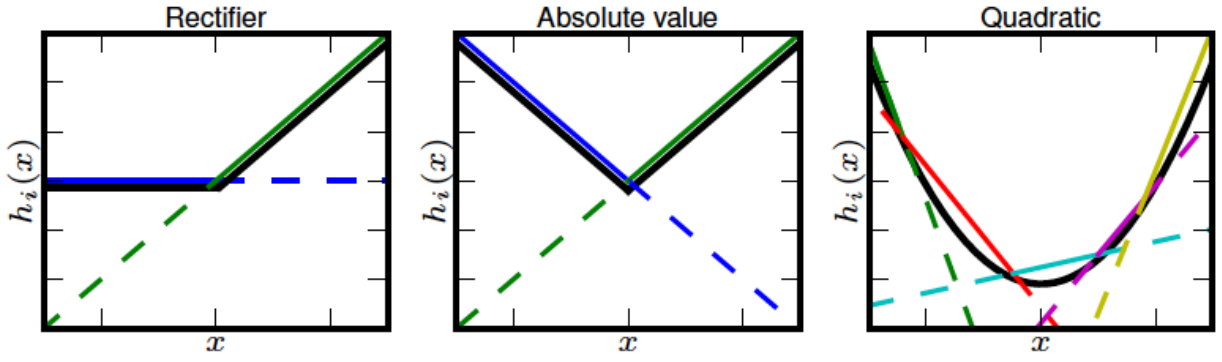
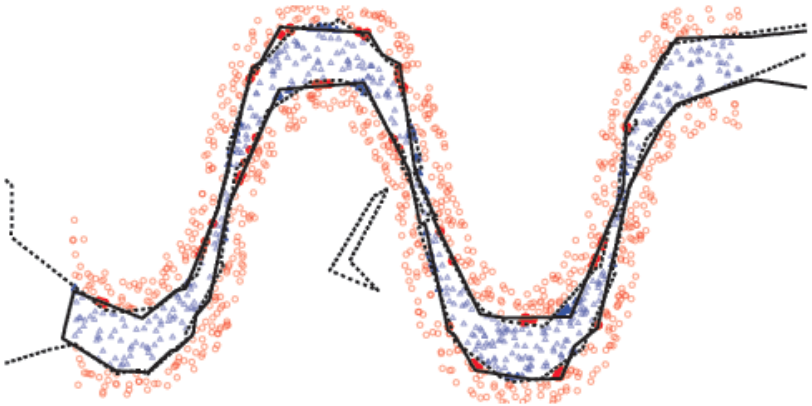


Figure 1. Graphical depiction of how the maxout activation function can implement the rectified linear, absolute value rectifier, and approximate the quadratic activation function. This diagram is 2D and only shows how maxout behaves with a 1D input, but in multiple dimensions a maxout unit can approximate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

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.



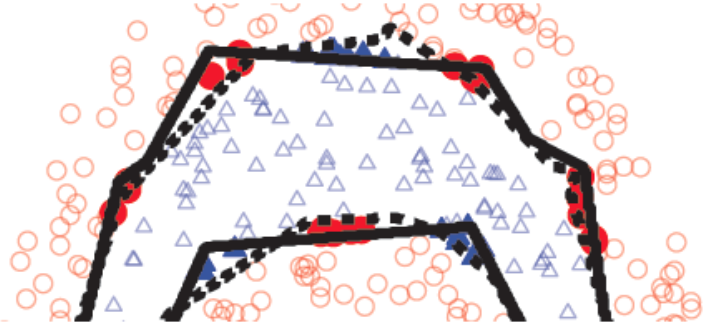
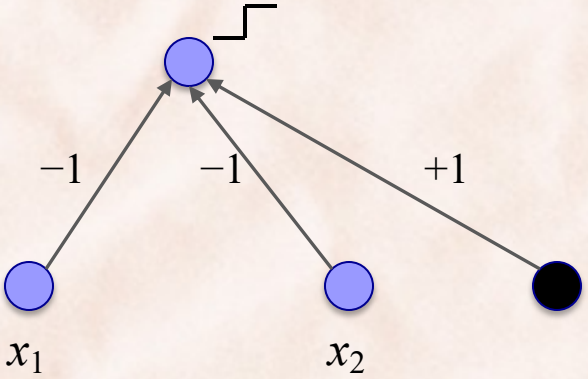
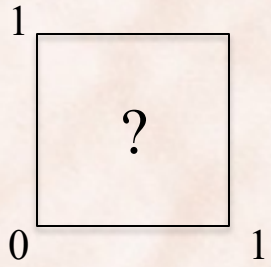


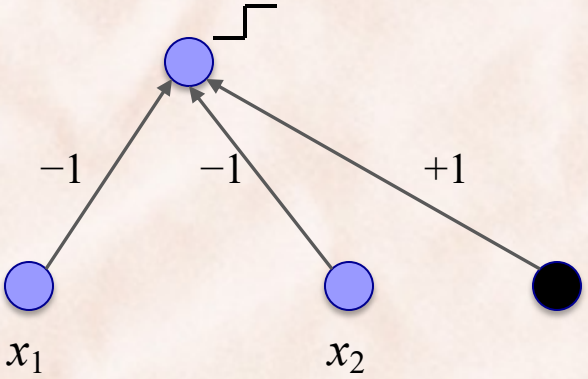
Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

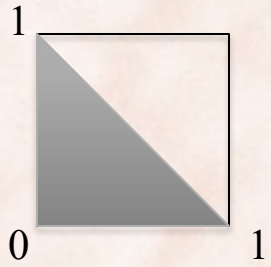
Folding Example



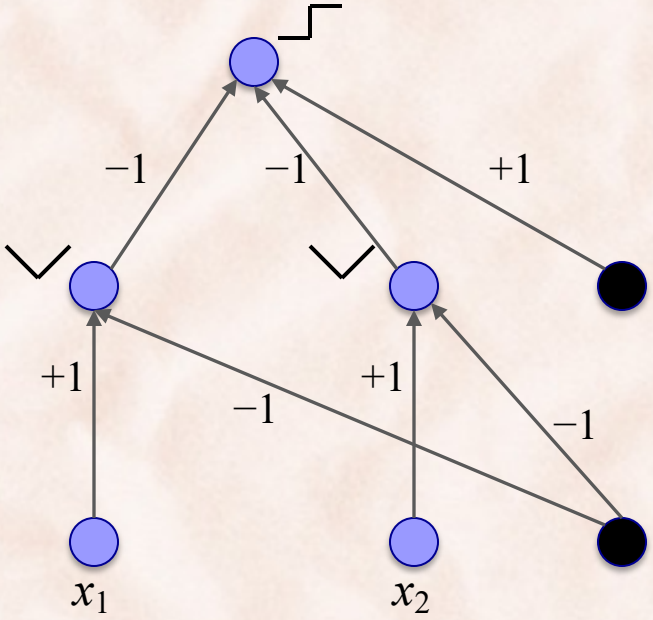


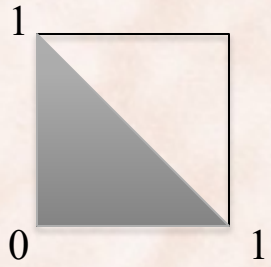
Folding Example

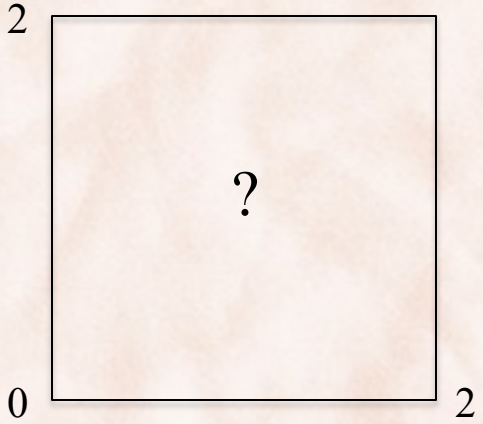




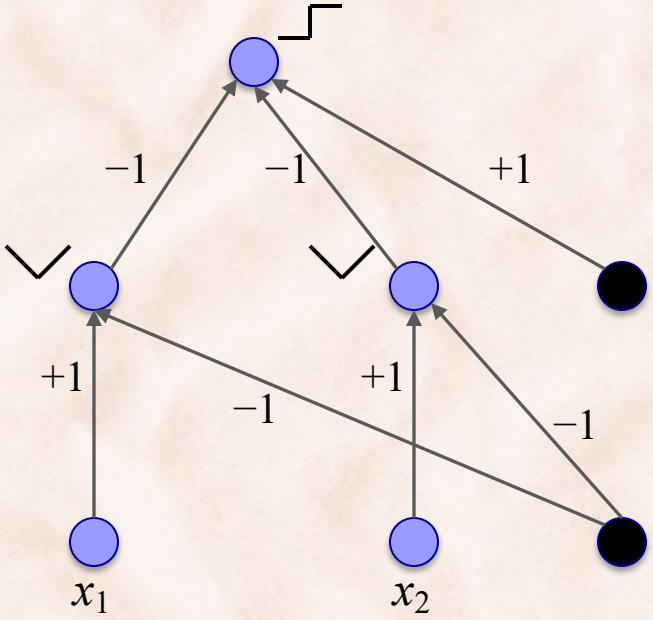
Folding Example

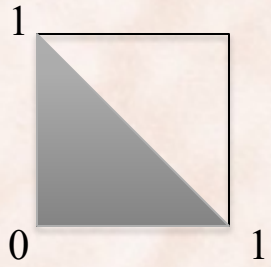


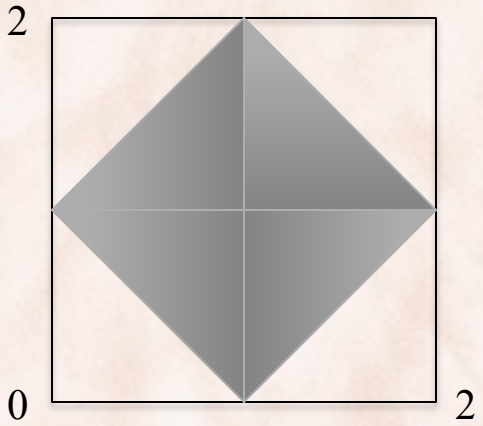




Folding Example

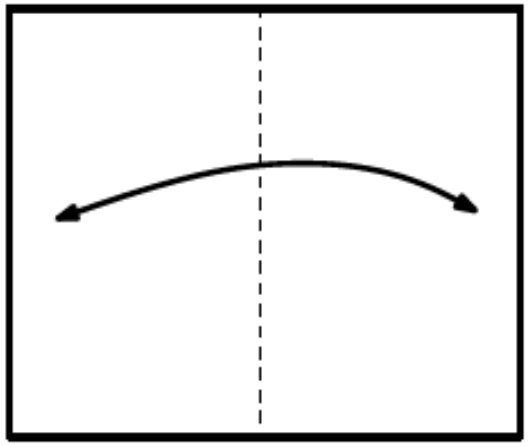




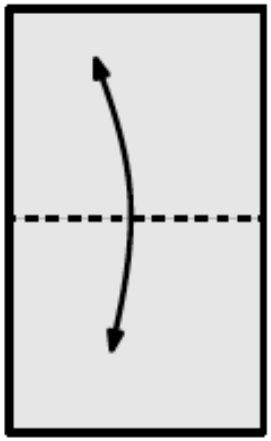


Space Foldings

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



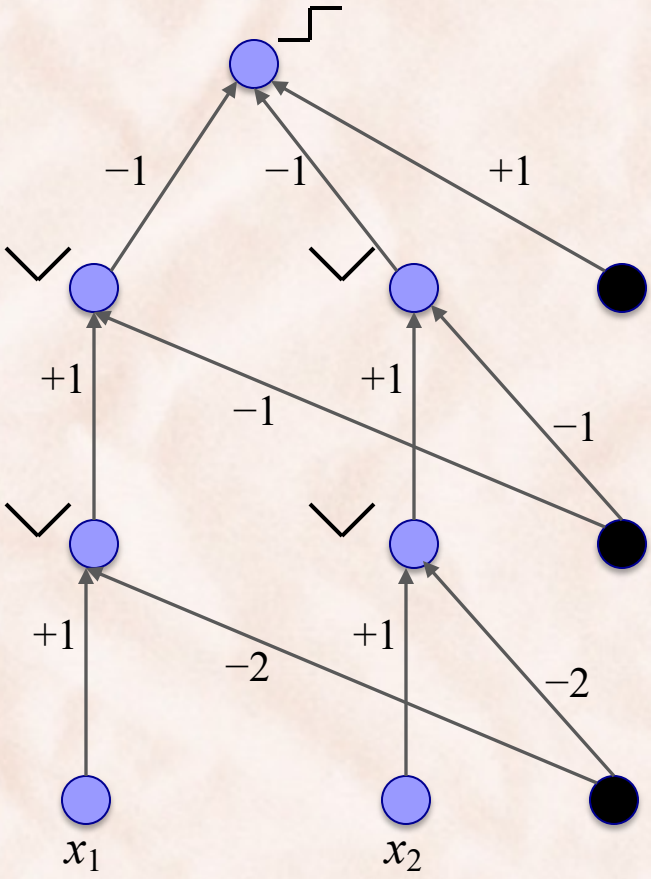
1. Fold along the vertical axis

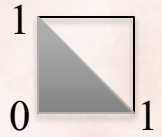


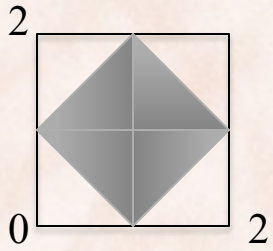
2. Fold along the horizontal axis

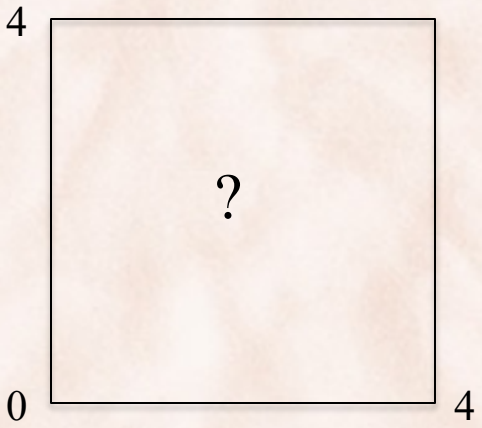
3.

Folding Example

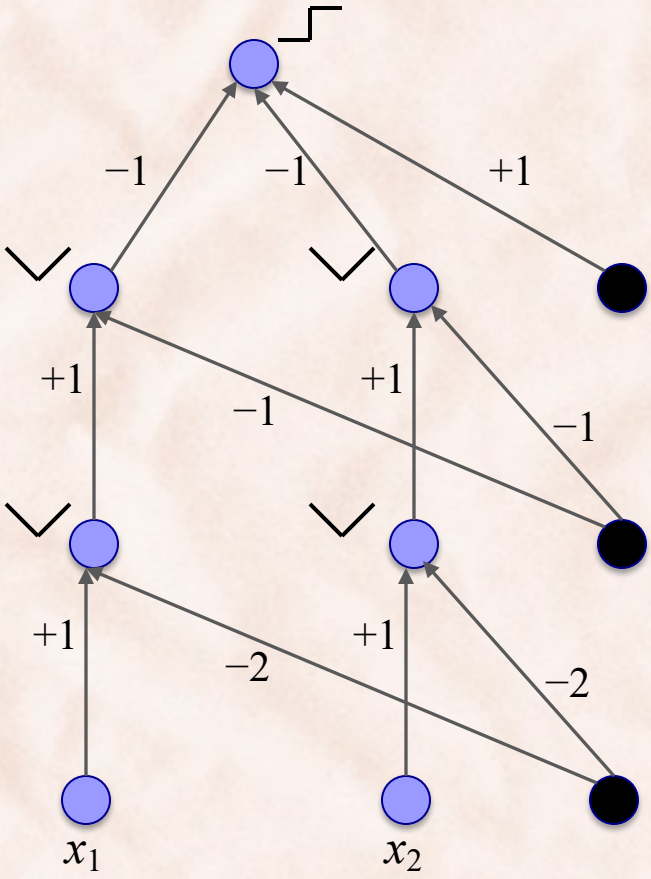


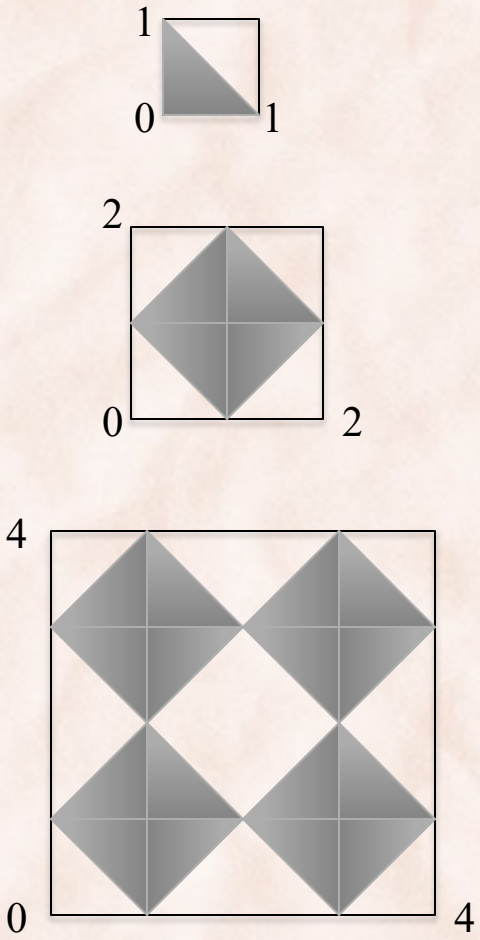






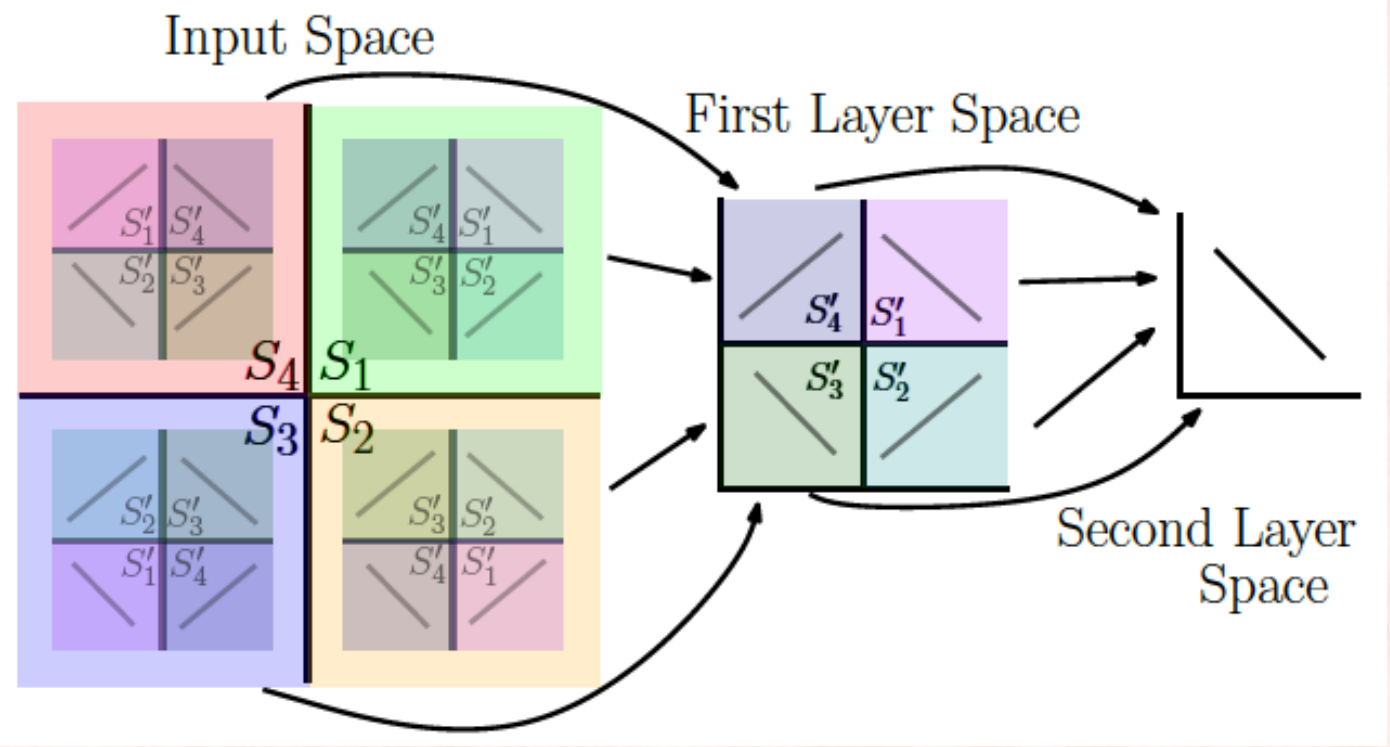
Folding Example



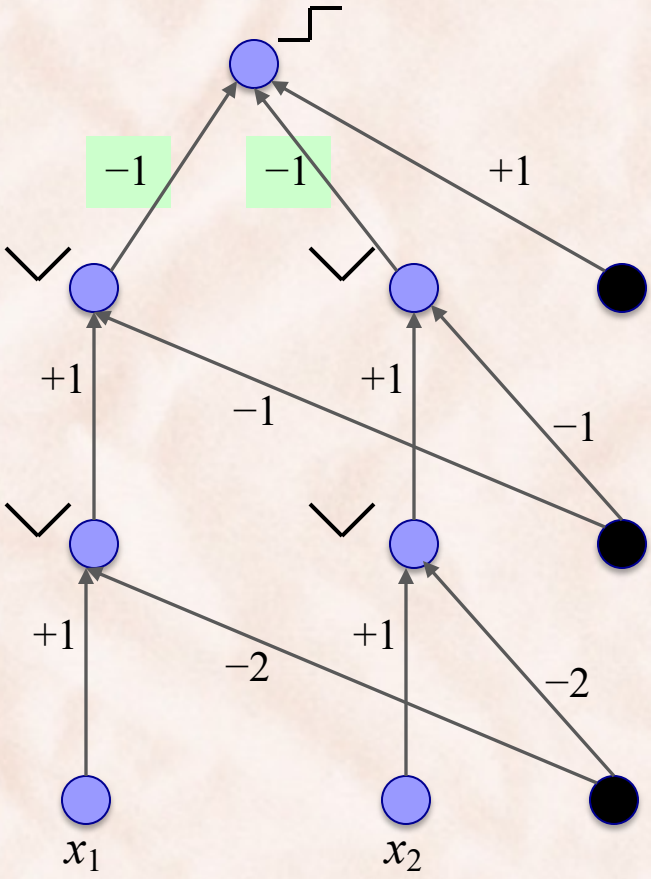


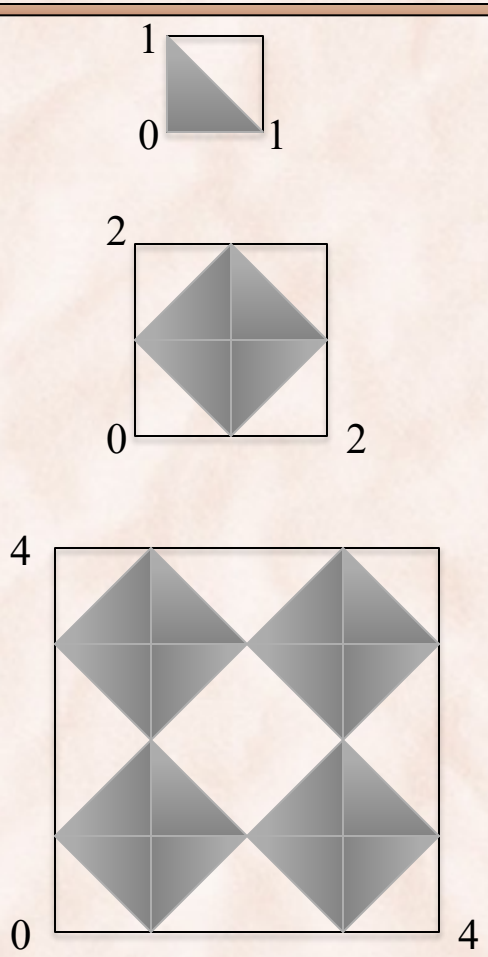
Space foldings

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

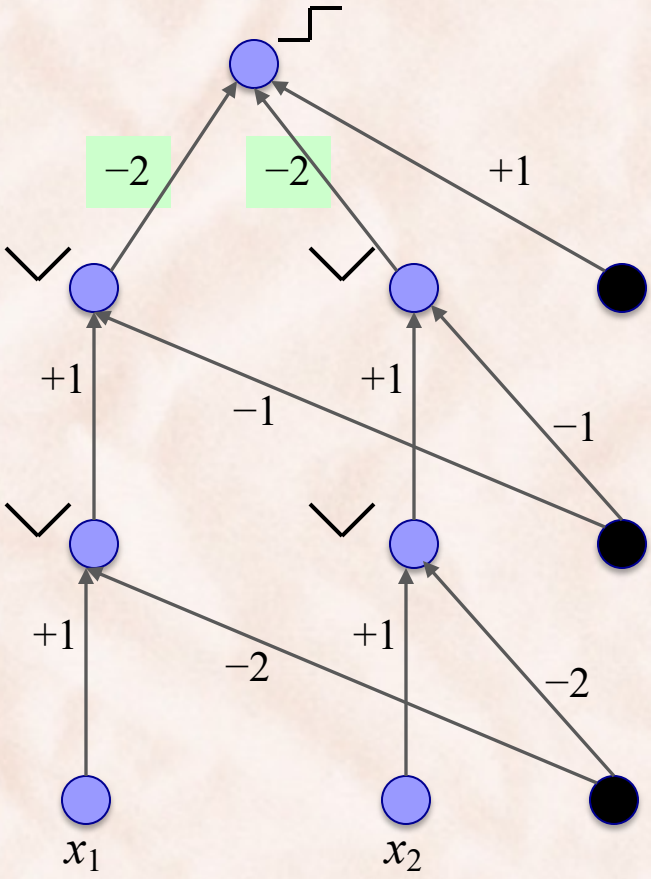


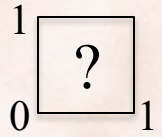
Folding Example

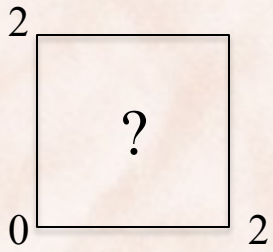


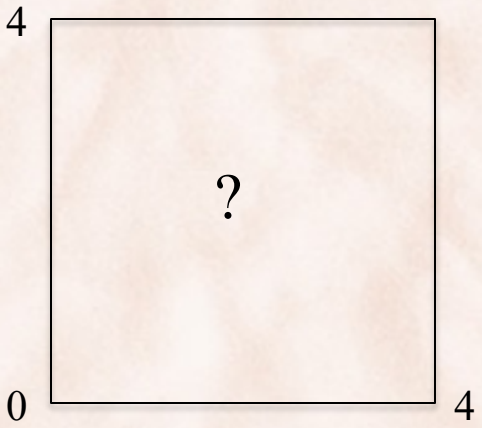


Folding Example

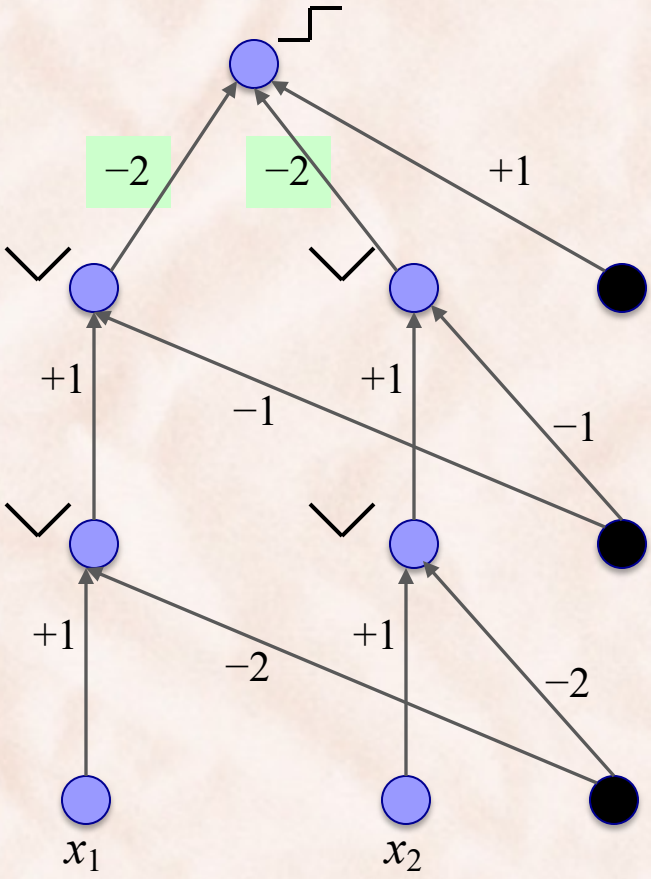


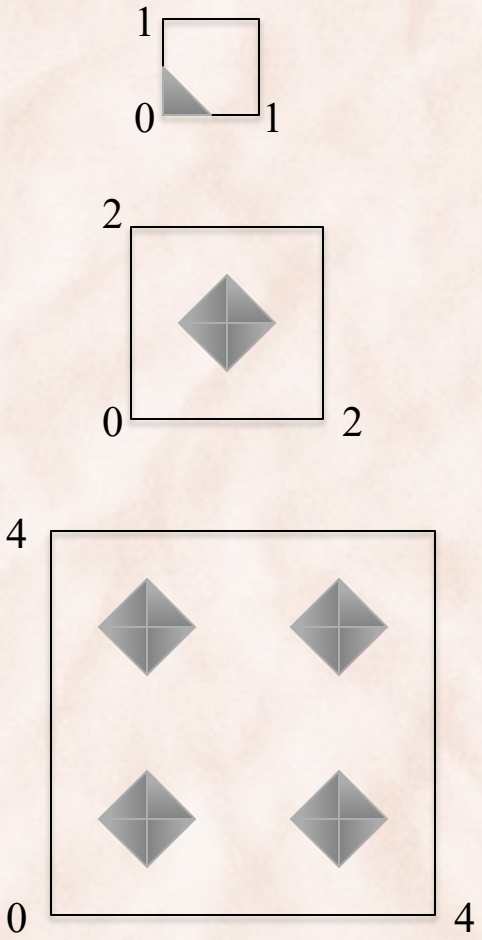






Folding Example





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.
- This means that 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.

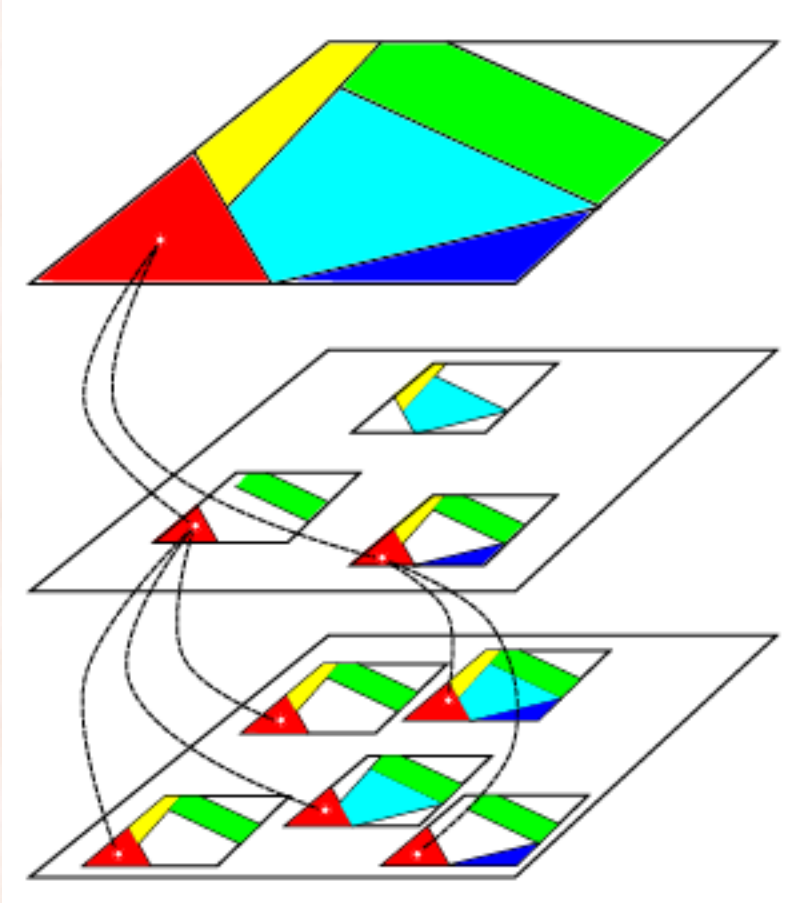
Space Foldings

[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 **W** and biases **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

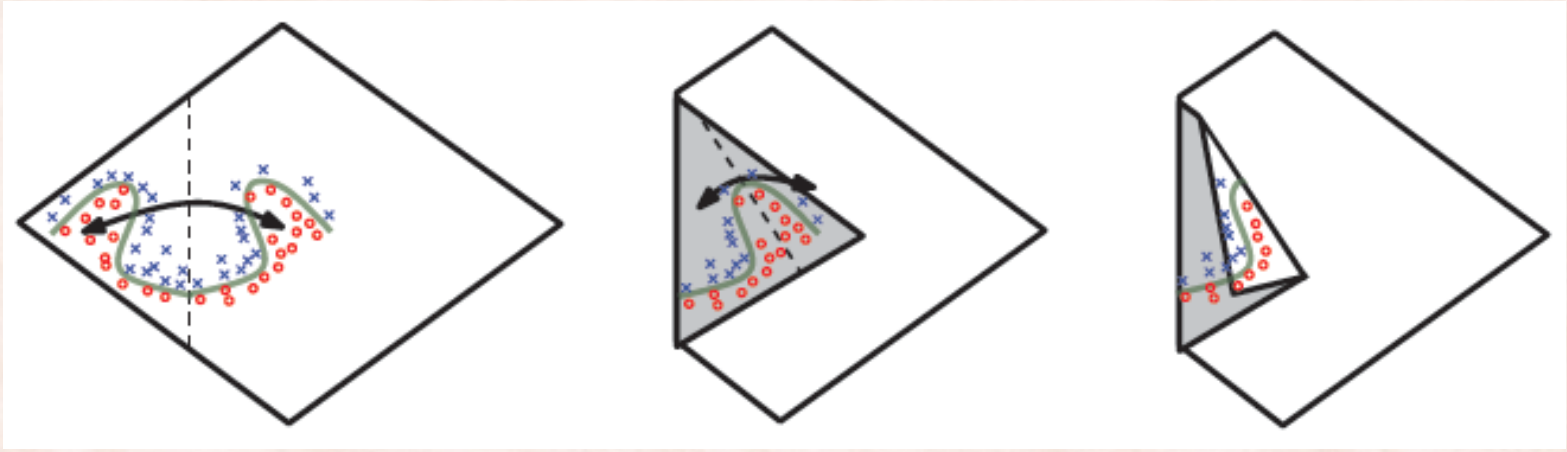
[Montufar et al., NIPS'14]



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.



Deep vs. Shallow Rectifier Networks

[Montufar et al., NIPS'14]

- A *linear region* of a piecewise linear function F: $R^d \rightarrow R^m$ is a maximal connected subset of the input-space 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:

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

Start Supplemental Material

Why are Deep Architectures Good for AI?

Why are Deep Architectures Good for AI?

- There is no guarantee that the kinds of functions we want to learn share this "folding" property.
- Choosing a deep model encodes a very general belief that:
  - The function we want to learn should involve composition of several simpler functions, OR
  - The learning problem consists of discovering a set of underlying factors of variation that can in turn be described in terms of other, simpler underlying factors of variation, OR
  - The function we want to learn is a computer program consisting of multiple steps, where each step uses of the previous step's output.
- Empirically, greater depth does seem to result in better generalization for a wide variety of tasks.

Digit Recognition Accuracy vs. Depth

[DL book, 2016]

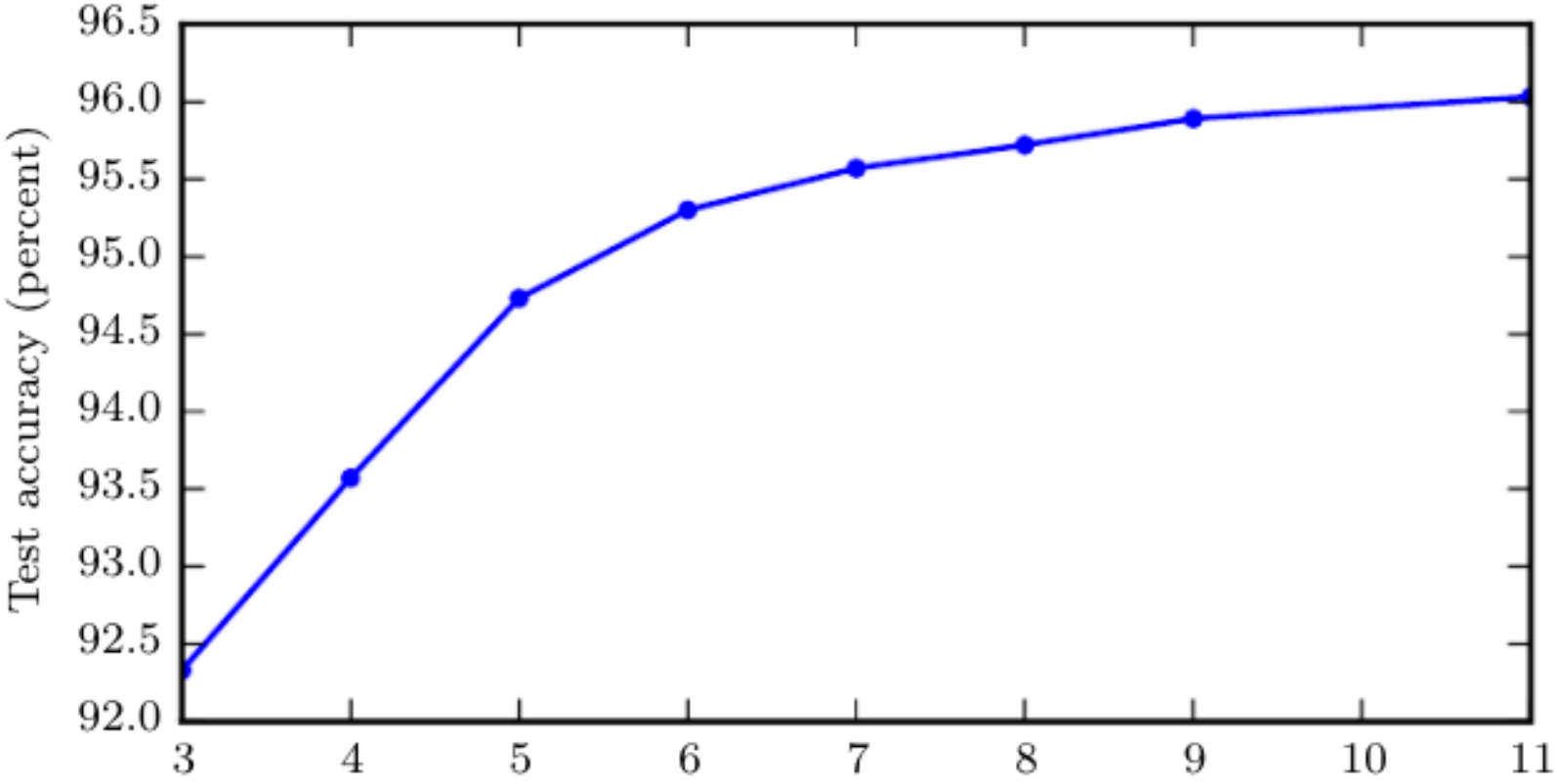


Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

Accuracy vs. Depth vs. Capacity

[DL book, 2016]

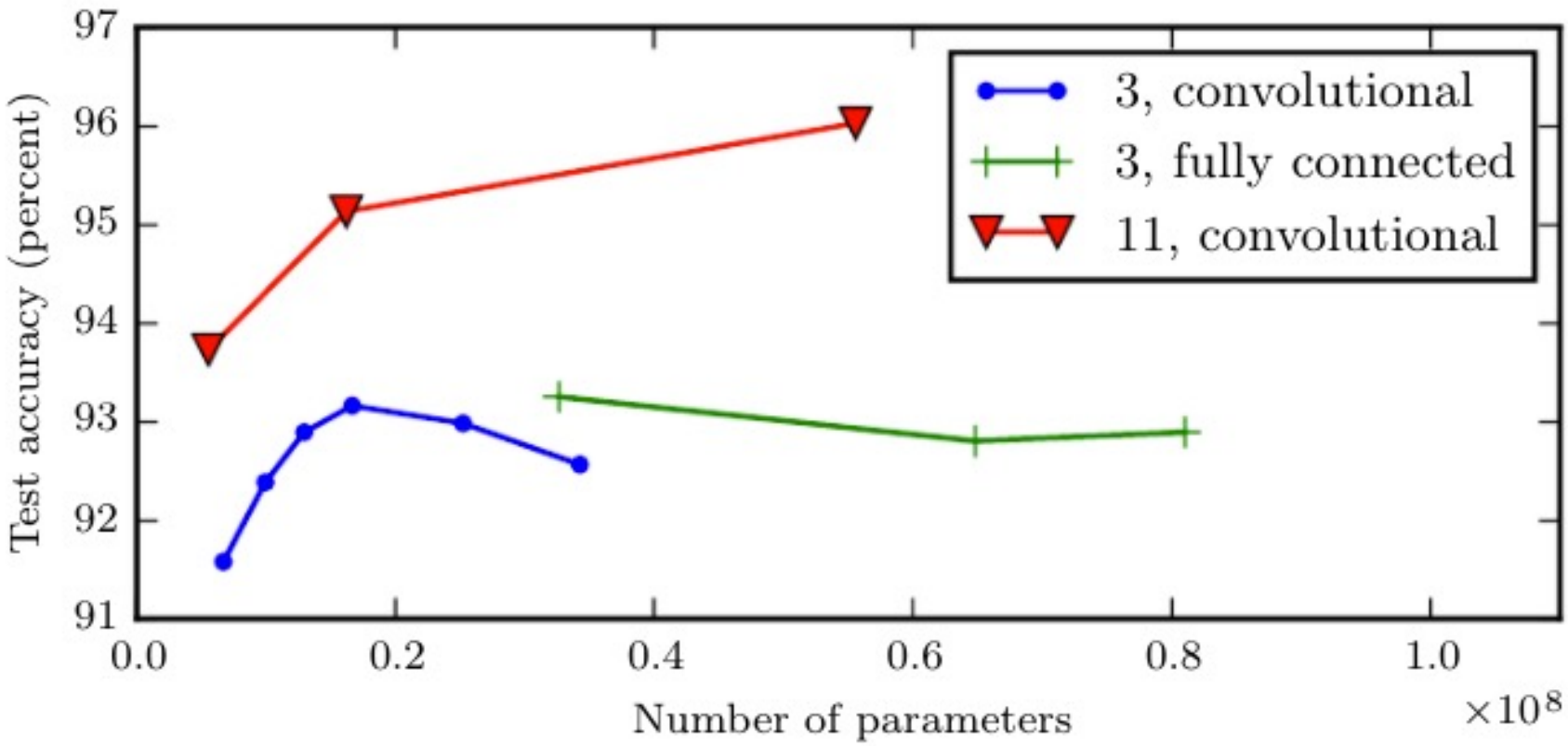


Figure 6.7: Deeper models tend to perform better. This is not merely because the model is larger. This experiment from Goodfellow *et al.* (2014d) shows that increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance. The legend indicates the depth of network used to make each curve and whether the curve represents variation in the size of

Why do Cheap & Deep Architectures Work for AI?

[Lin & Tegmark, 2016]

- **Paradox**: How can neural networks approximate functions well in practice, when the set of possible functions is exponentially larger than the set of practically possible networks?
  - **Example**: classify megapixel greyscale images into two categories, e.g., cats or dogs.
  - If each pixel can take one of 256 values, then there are $256^{1000000}$ possible images, and for each one, we wish to compute the probability that it depicts a cat.
  - This means that an arbitrary function is defined by a list of $256^{1000000}$ probabilities, i.e., way more numbers than there are atoms in our universe (about 10^{78}).

Why do Cheap & Deep Architectures Work for AI?

[Lin & Tegmark, 2016]

- **Paradox**: How can neural networks approximate functions well in practice, when the set of possible functions is exponentially larger than the set of practically possible networks?

- **Conjecture**: The data sets and functions we care about form a minuscule minority, and it is plausible that they can also be efficiently implemented by neural networks reflecting their generative process.

Why do Cheap & Deep Architectures Work for AI?

[Lin & Tegmark, 2016]

1. **Cheap**: The exceptional simplicity of physics-based functions hinges on properties such as **symmetry**, **locality**, **compositionality** and **polynomial log-probability**.
   - These properties translate into exceptionally simple neural networks approximating both natural phenomena such as images and abstract representations thereof such as drawings.
2. **Deep**: The statistical process generating the data is of a certain **hierarchical** form prevalent in physics and machine learning:
   - Therefore, a deep neural network can be more efficient than a shallow one.

Why do Cheap Architectures Work for AI?

[Lin & Tegmark, 2016]

- **Low polynomial order**: For reasons that are still not fully understood, our universe can be accurately described by polynomial Hamiltonians of low order d.
  - **Standard model**: At a fundamental level, the Hamiltonian of the standard model of particle physics has $d = 4$.
  - **Central Limit Theorem**: many probability distributions in machine-learning and statistics can be accurately approximated by multivariate Gaussians => Hamiltonian $H = -\ln p$ has $d = 2$.
- **Translation and rotation invariance**.
- **Locality**.
- **Symmetry**.

Why do Cheap Architectures Work for AI?

[Lin & Tegmark, 2016]

- **Low polynomial order**: For reasons that are still not fully understood, our universe can be accurately described by polynomial Hamiltonians of low order d.

- Neural networks can efficiently approximate multiplication!

Polynomials as Simple NNs

[Lin & Tegmark, 2016]

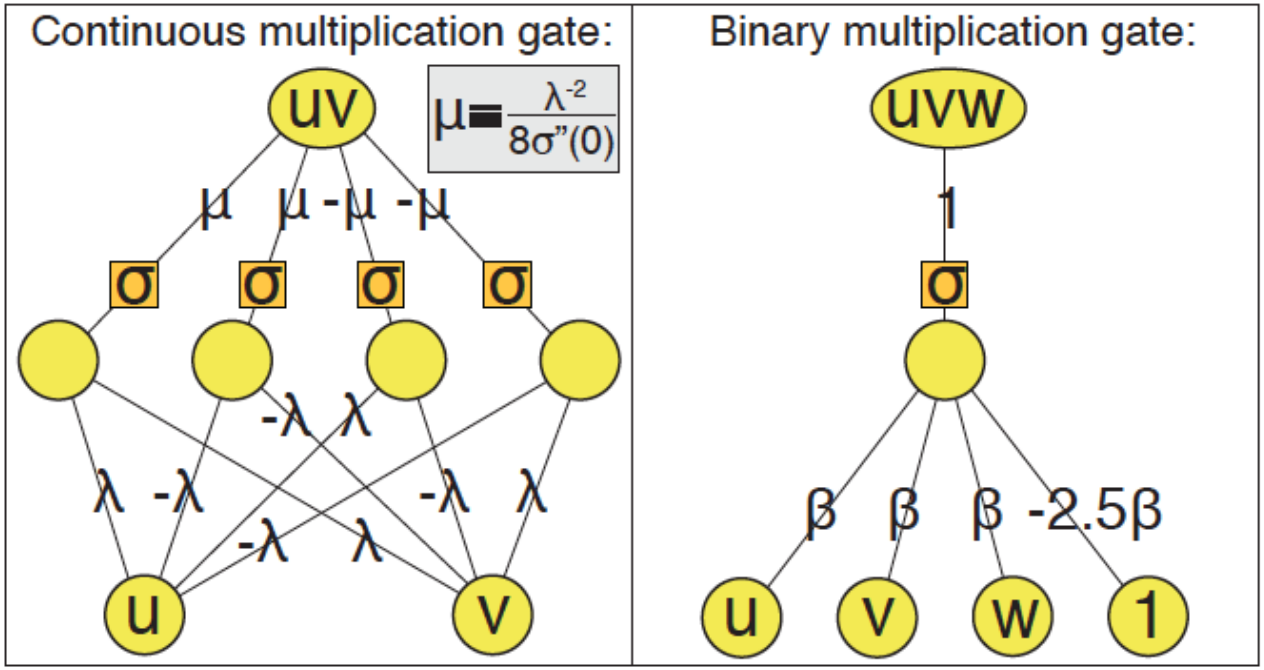
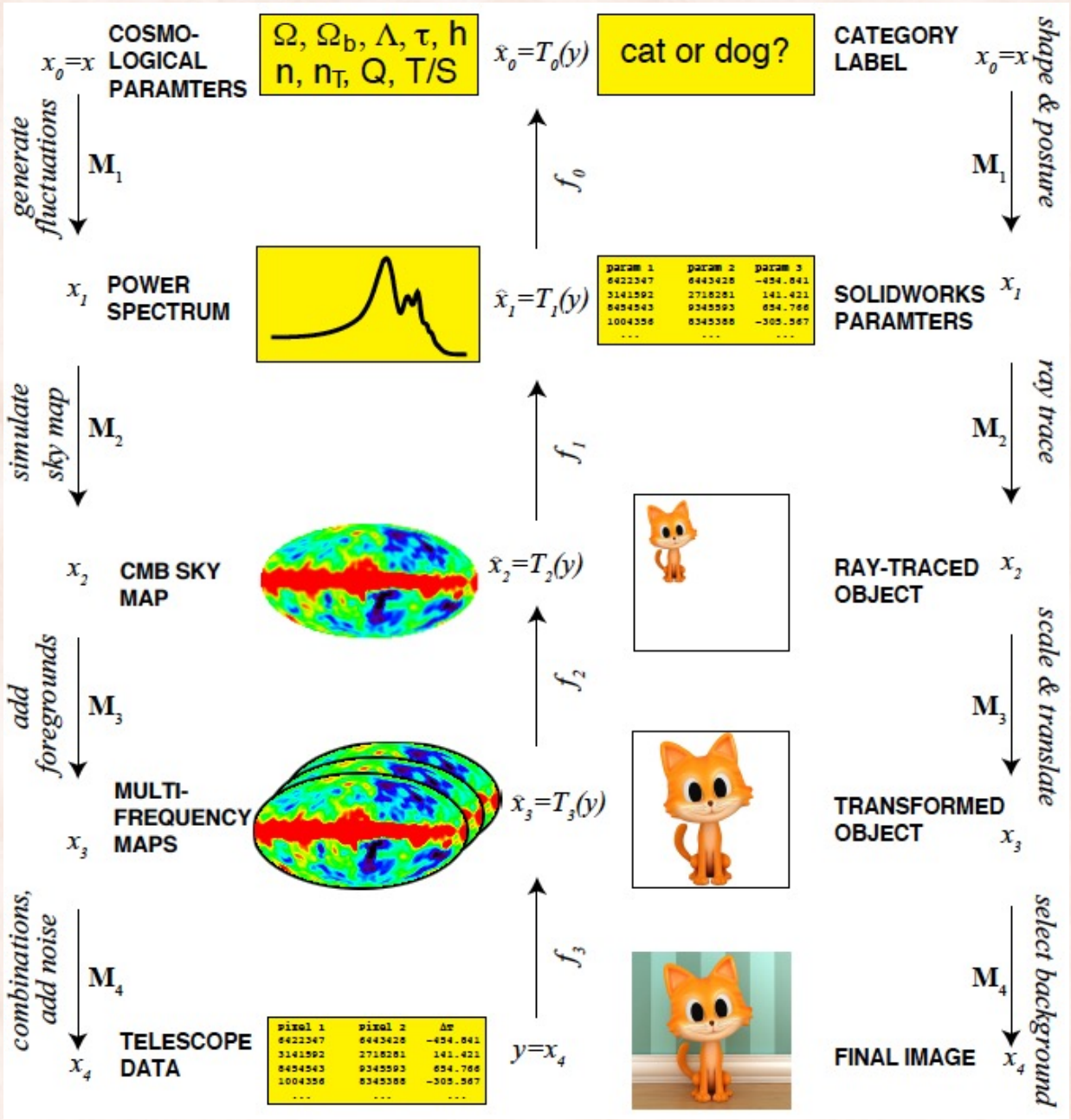


FIG. 2: Multiplication can be efficiently implemented by simple neural nets, becoming arbitrarily accurate as $\lambda \to 0$ (left) and $\beta \to \infty$ (right). Squares apply the function σ, circles perform summation, and lines multiply by the constants labeling them. The "1" input implements the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

Why do Deep Architectures Work for AI?

[Lin & Tegmark, 2016]

- **Hierarchical Structure**: One of the most striking features of the physical world is its hierarchical structure.
  - **Spatially**, it is an object hierarchy: elementary particles form atoms which in turn form molecules, cells, organisms, planets, solar systems, galaxies, etc.
  - **Causally**, complex structures are frequently created through a distinct sequence of simpler steps.


$x_0=x$ COSMO-LOGICAL PARAMTERS
$\Omega, \Omega_b, \Lambda, \tau, h$ $n, n_T, Q, T/S$
$\hat{x}_0=T_0(y)$
cat or dog?
CATEGORY LABEL $x_0=x$
generate fluctuations
M_1
f_0
shape & posture
x_1 POWER SPECTRUM
$\hat{x}_1=T_1(y)$
SOLIDWORKS PARAMTERS x_1
simulate sky map
M_2
f_1
ray trace
x_2 CMB SKY MAP
$\hat{x}_2=T_2(y)$
RAY-TRACED OBJECT x_2
add foregrounds
M_3
f_2
scale & translate
MULTI-FREQUENCY MAPS x_3
$\hat{x}_3=T_3(y)$
TRANSFORMED OBJECT x_3
combinations, add noise
M_4
f_3
select background
TELESCOPE DATA x_4
$y=x_4$
FINAL IMAGE x_4

Why do Cheap & Deep Architectures Work for AI?

[Lin & Tegmark, 2016]

- **Cheap & Deep**:
  - **Paradox**: The number of parameters required to describe an arbitrary function of the input data y is beyond astronomical.
  - **Solution**: The generative process can be specified by a more modest number of parameters, because each of its steps can.
    - For a megapixel image of a galaxy, its entire prob. distribution is defined by the standard model of particle physics with its 32 parameters, which together specify the process transforming primordial hydrogen gas into galaxies.
    - Giving the simple low-information content instruction “draw a cute kitten" to a random sample of artists will produce a wide variety of images y with a complicated probability distribution over colors, postures, etc. But the pre-stored information about cat probabilities in these artists' brains is modest in size.

End Supplemental Material

Why are Deep Architectures Good for AI?

Readings

- Chapter 7 on Neural Netwoks in the NLP textbook.