Natural Language Processing

Lecture 11:
Machine Learning: Feed-forward Neural Networks

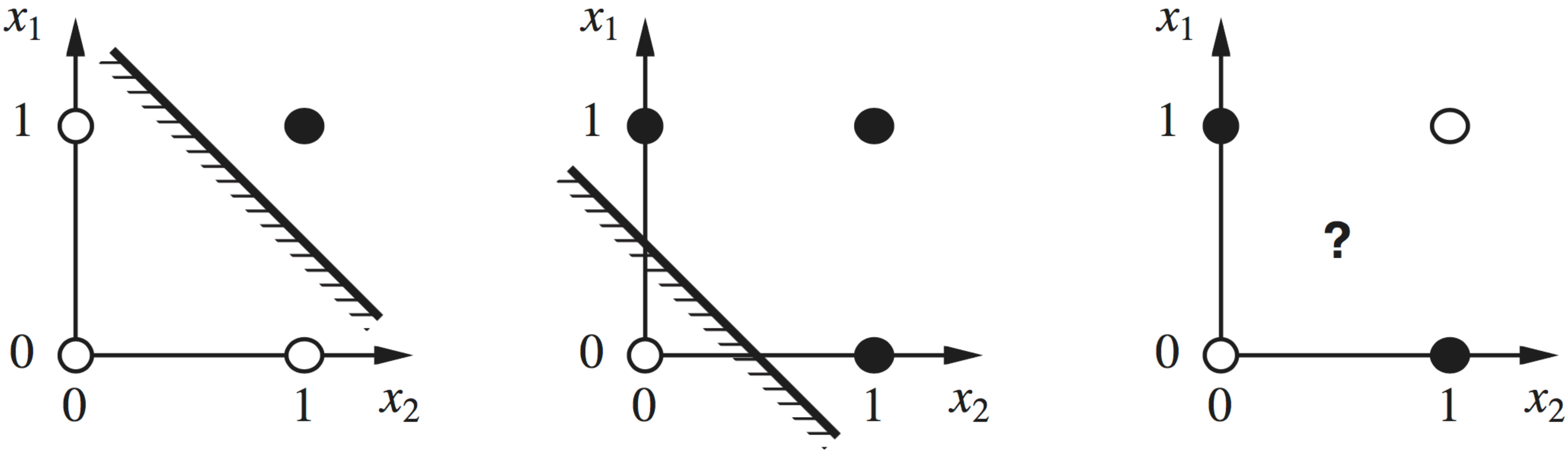
10/23/2018 & 10/25/2018

COMS W4705
Daniel Bauer

Perceptron Expressiveness

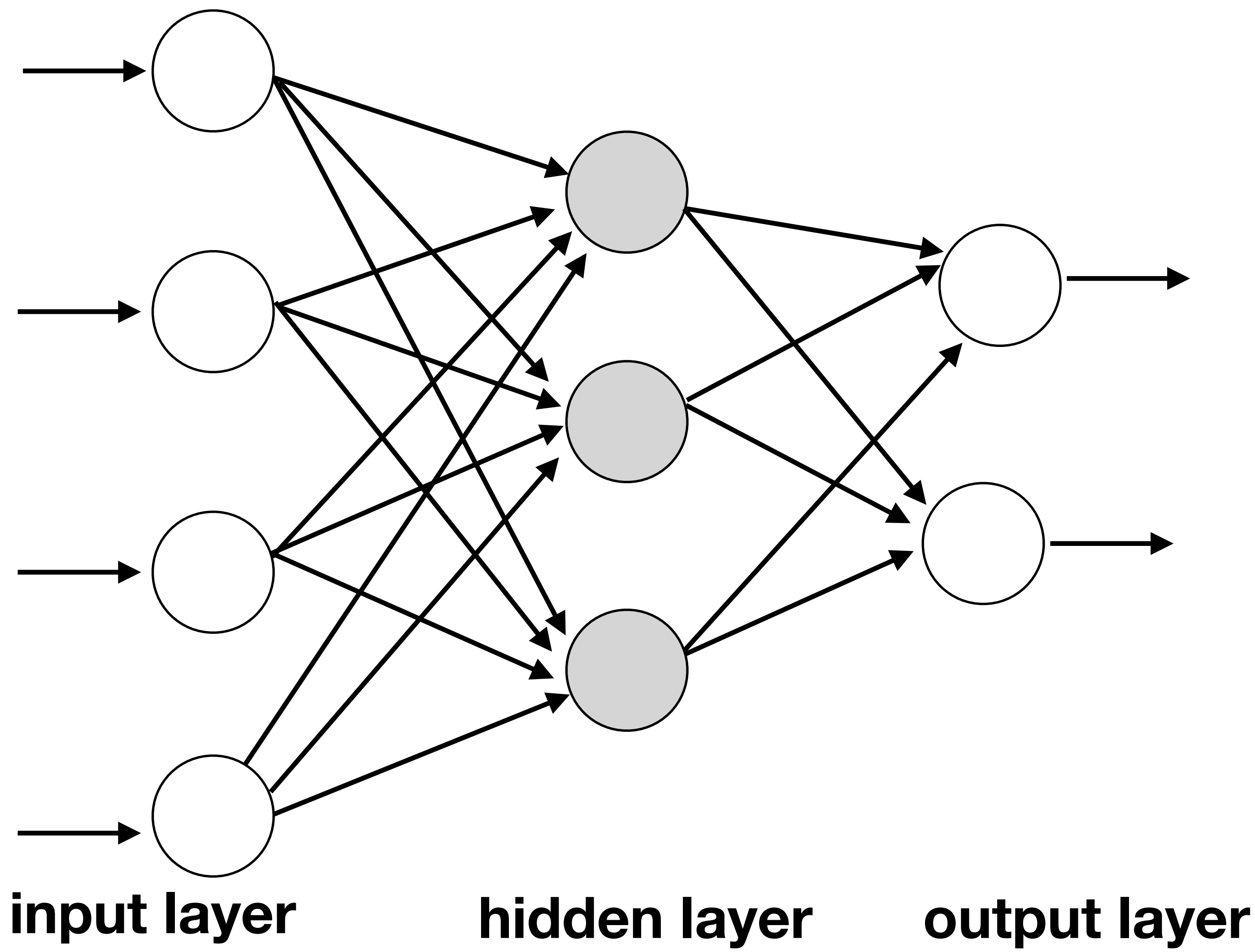
- Simple perceptron learning algorithm, starts with an arbitrary hyperplane and adjusts it using the training data.
    - Step function is not differentiable, so no closed-form solution.
- Perceptron produces a linear separator.
    - Can only learn linearly separable patterns.
- Can represent boolean functions like **and, or, not** but not the **xor** function.

The problem with *xor*



(a) x_1 **and** x_2 (b) x_1 **or** x_2 (c) x_1 **xor** x_2

Multi-Layer Neural Networks

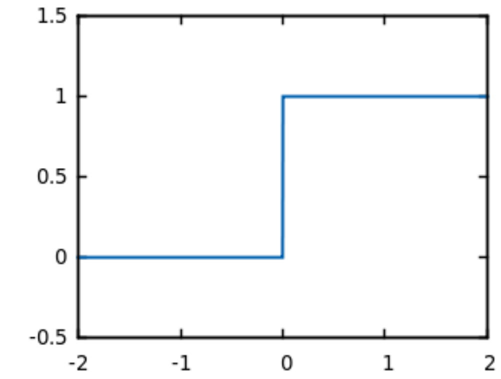


- Basic idea: represent any (non-linear) function as a composition of soft-threshold functions. This is a form of non-linear regression.

- Lippmann 1987: Two hidden layers suffice to represent any arbitrary region (provided enough neurons), even discontinuous functions!

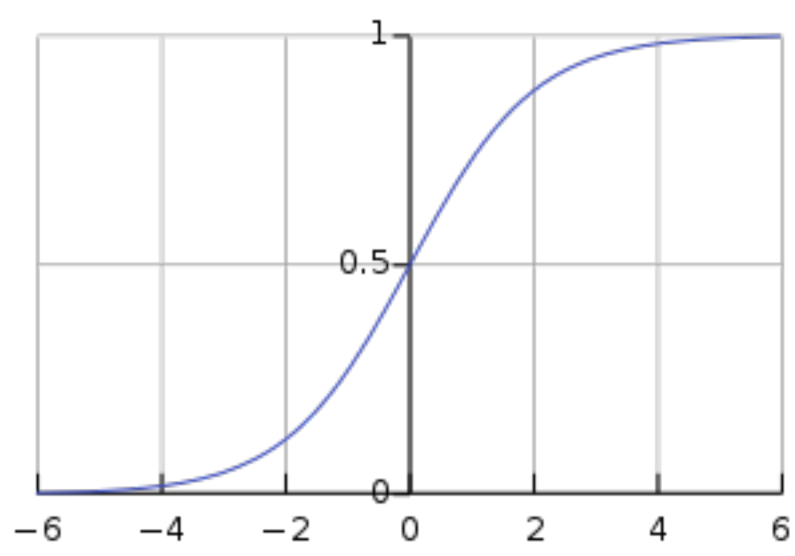
Activation Functions

- One problem with perceptrons is that the **threshold function (step function)** is undifferentiable.



- It is therefore unsuitable for gradient descent.

- One alternative is the **sigmoid (logistic) function:**

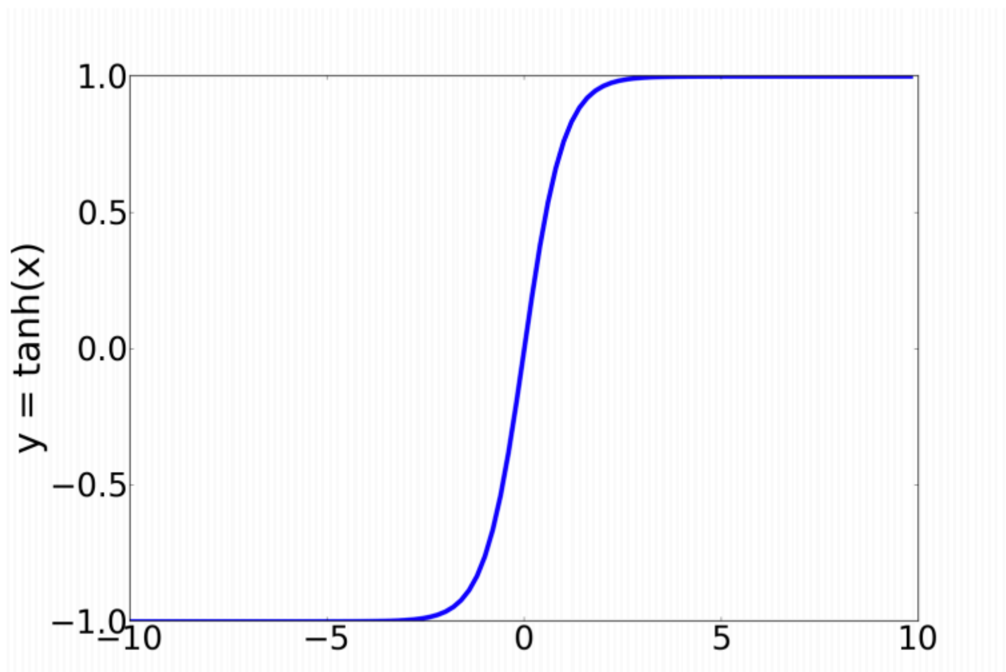


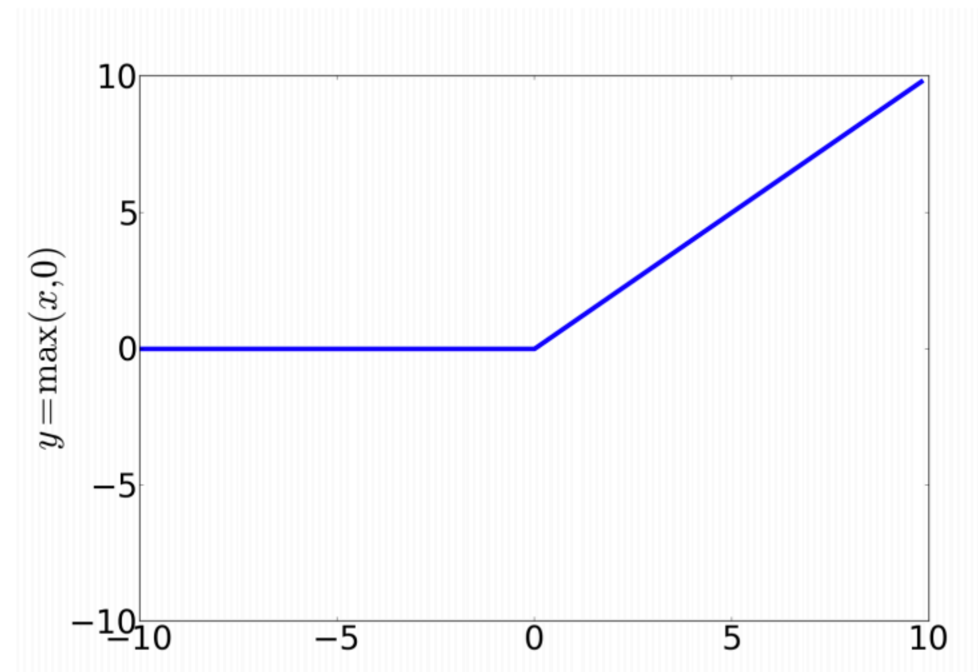
$$g(z) = \frac{1}{1 + e^{-z}}$$

$g(z) = 0$ if $z \rightarrow -\infty$
$g(z) = 1$ if $z \rightarrow \infty$

Activation Functions

- Two other popular activation functions:

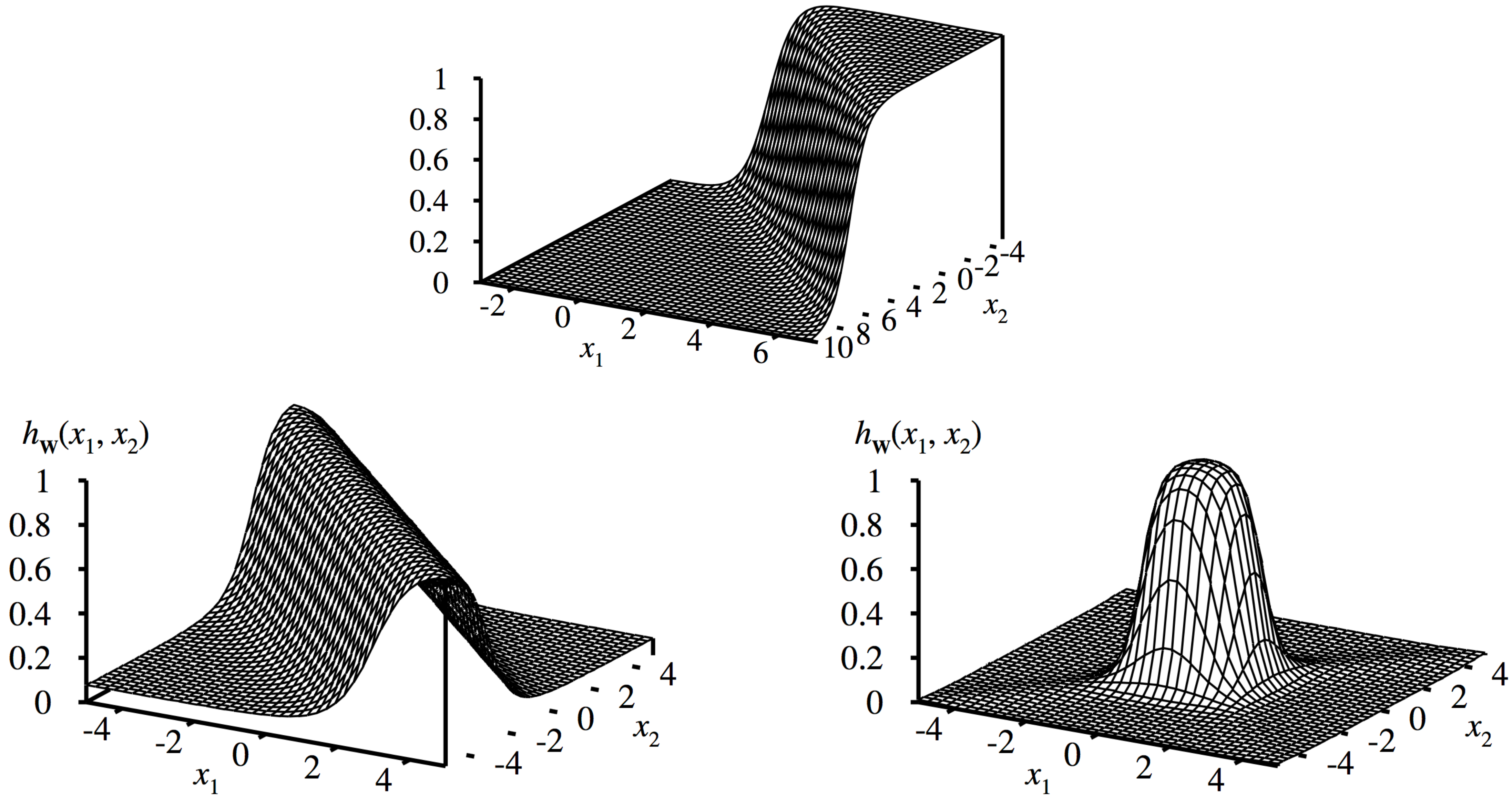




$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$relu(z) = max(z, 0)$$

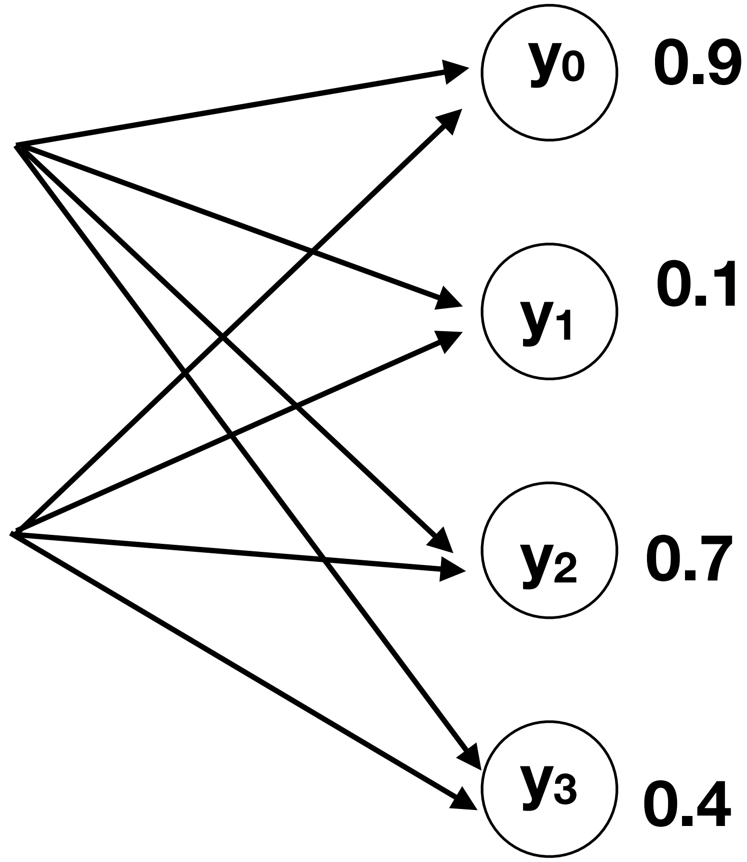
How Do Neural Networks Represent Functions?



Output Representation

- Many NLP Problems are multi-class classification problems.

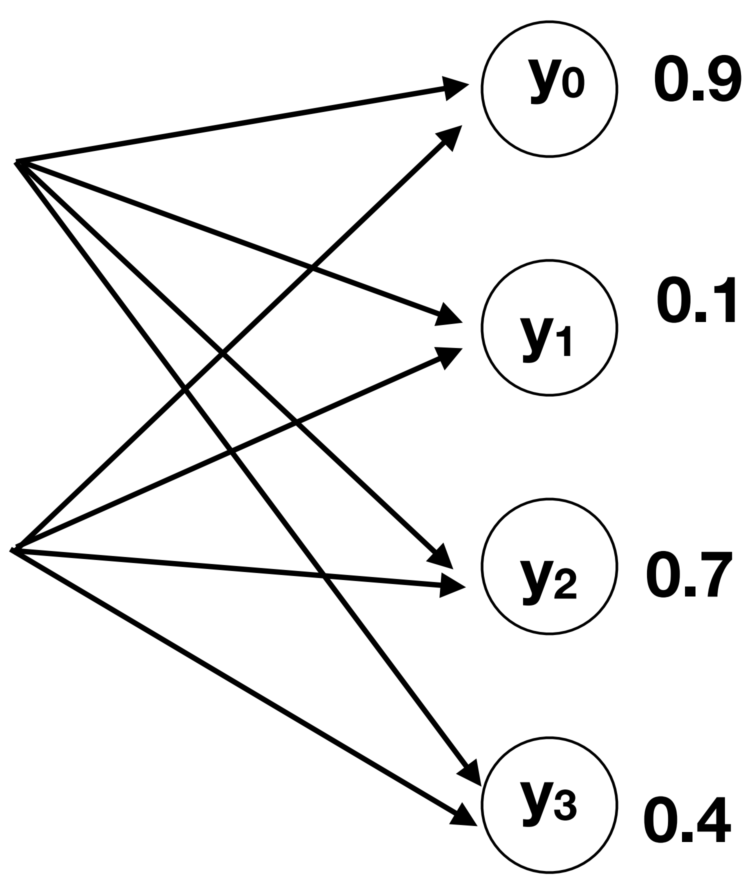
- Each output neuron represents one class. Predict the class with the highest activation.



Softmax

- We often want the activation at the output layer to represent probabilities.

- Normalize activation of each output unit by the sum of all output activations (as in log-linear models).



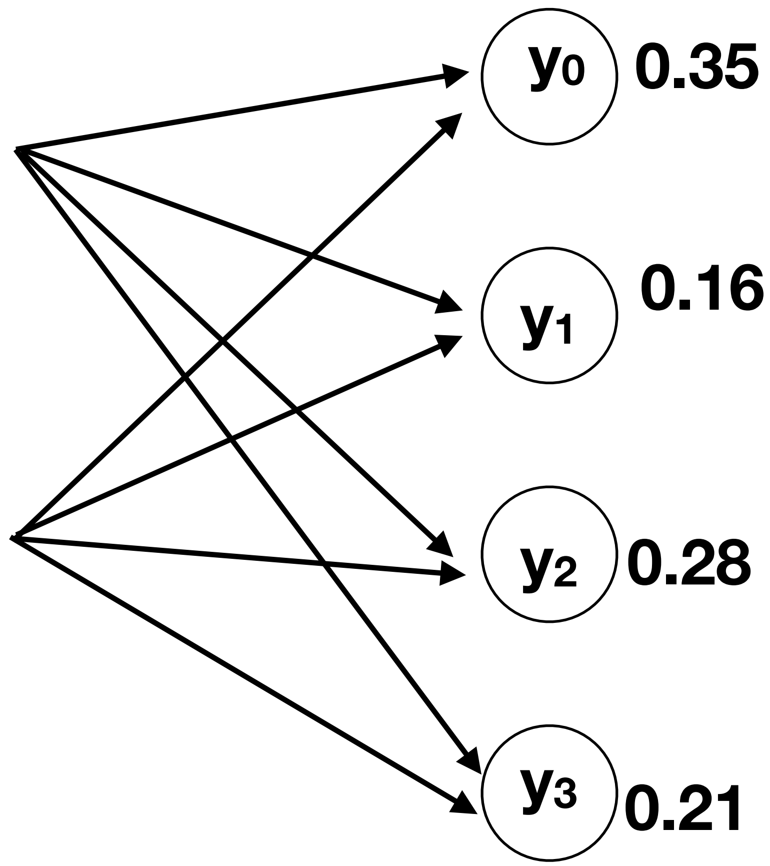
$$softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{k} exp(z_j)}$$

The network computes a probability

$$P(c_i | \mathbf{x}; \mathbf{w})$$

Softmax

- We often want the activation at the output layer to represent probabilities.
- Normalize activation of each output unit by the sum of all output activations (as in log-linear models).



$$softmax(z_i) = \frac{exp(z_i)}{\sum_{j=1}^{k} exp(z_j)}$$

The network computes a probability

$$P(c_i | \mathbf{x}; \mathbf{w})$$

Learning in Multi-Layer Neural Networks

- Network structure is fixed, but we want to train the weights. Assume **feed-forward** neural networks: no connections that are loops.

- **Backpropagation Algorithm:**

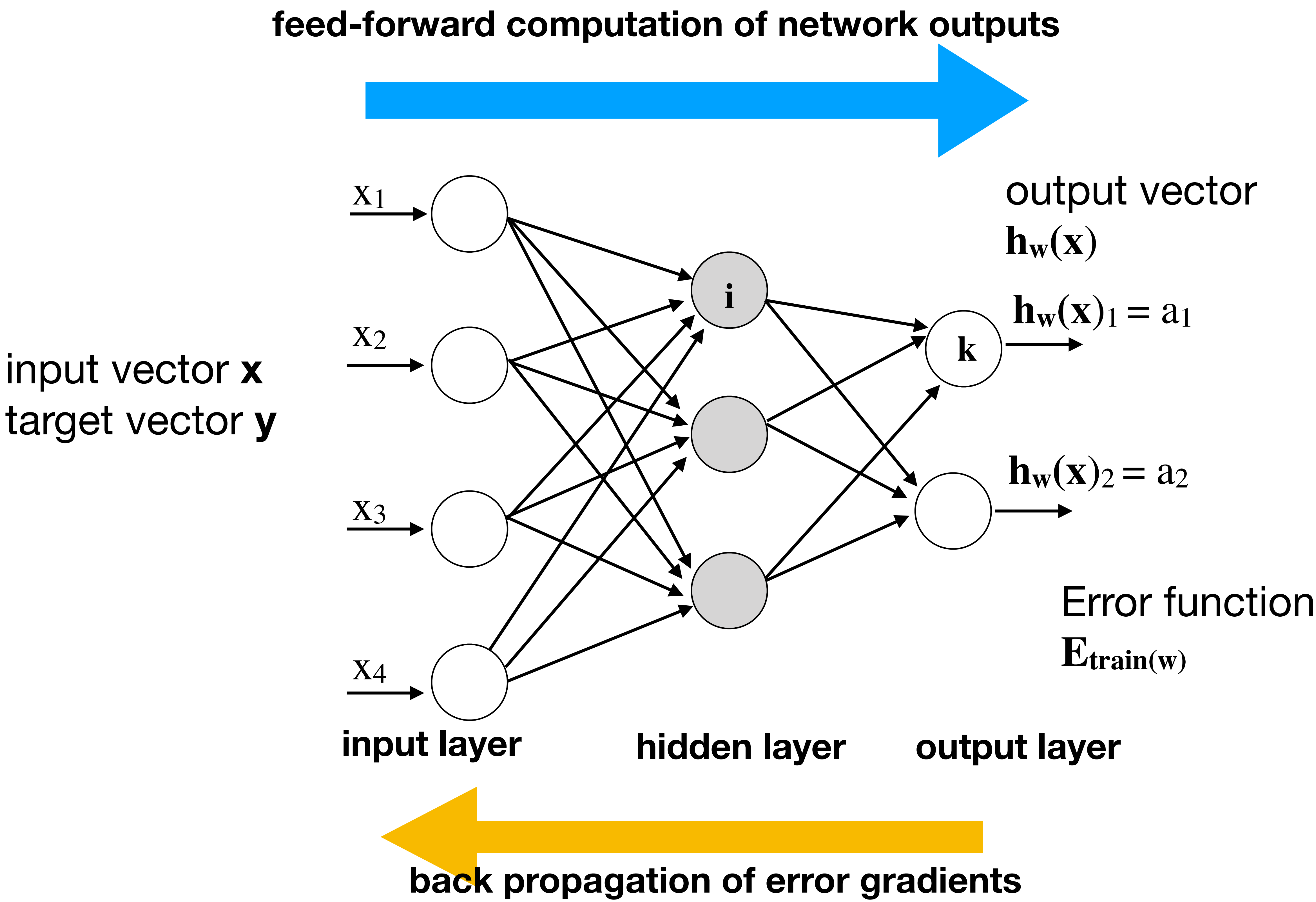
  - Given current weights, get network output and compute loss function (assume multiple outputs / a vector of outputs).

  - Can use gradient descent to update weights and minimize loss.

  - Problem: We only know how to do this for the last layer!

  - Idea: Propagate error backwards through the network.

Backpropagation



Negative Log-Likelihood

(also known as cross-entropy)

- Assume target output is a one-hot vector and $c(y)$ is the target class for target **y**.

- Compute the negative log-likehood for a single example

$$Loss(\mathbf{y}, h_{\mathbf{w}}(x)) = -logP(c(\mathbf{y})|\mathbf{x}; \mathbf{w})$$

- Empirical error for the entire training data:

$$E_{train}(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} -logP(c(\mathbf{y}^{(i)})|\mathbf{x}^{i}; \mathbf{w})$$

Stochastic Gradient Descent

- Goal: Learn parameters that minimize the empirical error.

Randomly initialize w

for a set number of iterations T:

  shuffle training data $\mathcal{D} = (x^{(j)}, y^{(j)})|_{j=1}^{n}$

  for $j = 1 ... N$:

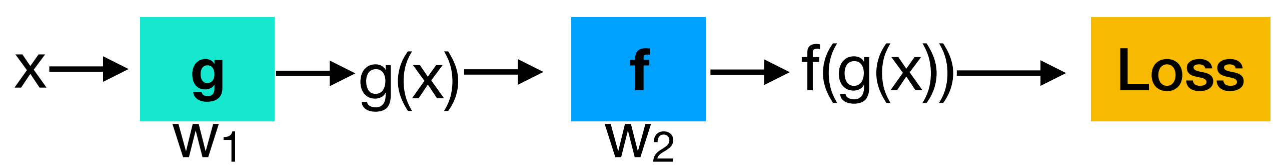
    for each w_i (all weights in the network):

$$w_i \leftarrow w_i - \eta \frac{\partial}{\partial w_i} Loss(y^{(j)}, h_{\mathbf{w}}(x^{(j)}))$$

- η is the learning rate.
- It often makes sense to compute the gradient over batches of examples, instead of just one ("mini-batch").

Backpropgation

- Simplified multi-layer case (a single unit per layer):



- Stochastic Gradient Descent should perform the following update:

$$w_2 \leftarrow w_2 - \eta \frac{\partial Loss(y, f(g(x))}{\partial w_2}$$

$$w_1 \leftarrow w_1 - \eta \frac{\partial Loss(y, f(g(x))}{\partial w_1}$$

- Problem: How do we compute the gradient for parameters w_1 and w_2?

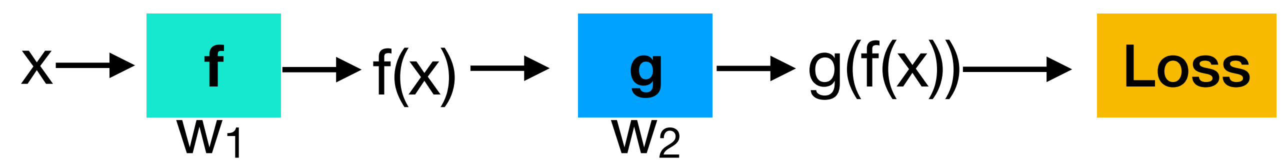
Chain Rule of Calculus

- To compute gradients for hidden units, we need to use apply the chain rule of calculus:

The derivative of $f(g(x))$ is

$$\frac{df(g(x))}{dx} = \frac{df(g(x))}{dg(x)} \cdot \frac{dg(x)}{dx}$$

Backpropagation

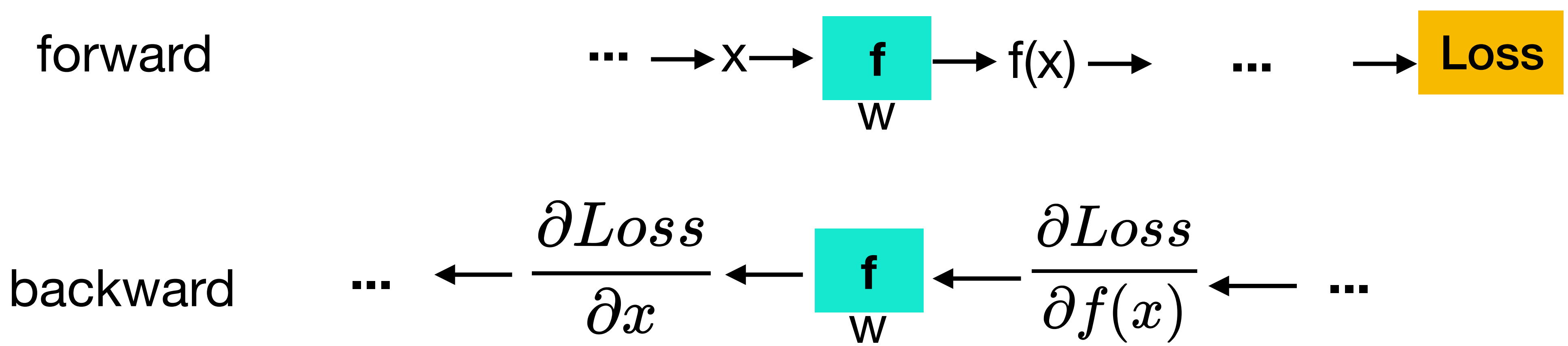


$$\frac{\partial Loss}{w_2} = \left(\frac{\partial Loss}{\partial f(g(x))}\right)\left(\frac{\partial f(g(x)}{\partial w2}\right)$$

$$\frac{\partial Loss}{w_1} = \left(\frac{\partial Loss}{\partial g(x)}\right)\left(\frac{\partial g(x)}{\partial w_1}\right)$$

$$= \left(\frac{\partial Loss}{\partial f(g(x))}\right)\left(\frac{\partial f(g(x))}{\partial g(x)}\right)\left(\frac{\partial g(x)}{\partial w_1}\right)$$

Backpropagation



Assume we know $\frac{\partial Loss}{\partial f(x)}$

We want to compute $\frac{\partial Loss}{\partial x}$ to propagate it back.

and $\frac{\partial Loss}{\partial w}$ (for the weight update)

Backpropagation

forward $\cdots \longrightarrow x \longrightarrow \boxed{f}_{\,w} \longrightarrow f(x) \longrightarrow \cdots \longrightarrow \boxed{\text{Loss}}$

backward $\cdots \longleftarrow \dfrac{\partial Loss}{\partial x} \longleftarrow \boxed{f}_{\,w} \longleftarrow \dfrac{\partial Loss}{\partial f(x)} \longleftarrow \cdots$

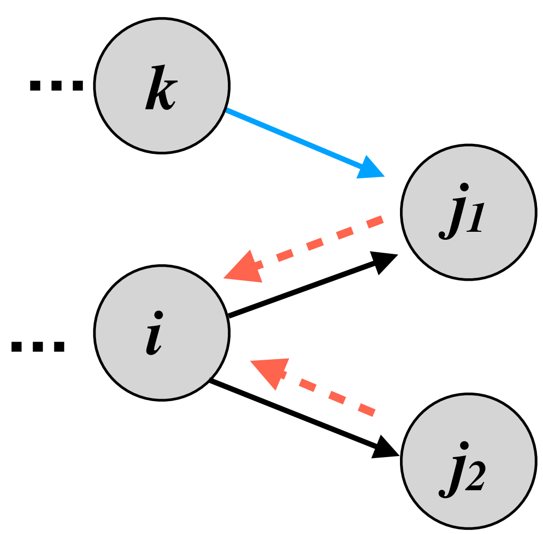
$$\frac{\partial Loss}{\partial x} = \left(\frac{\partial Loss}{\partial f(x)}\right)\left(\frac{\partial f(x)}{\partial x}\right)$$

$$\frac{\partial Loss}{\partial w} = \left(\frac{\partial Loss}{\partial f(x)}\right)\left(\frac{\partial f(x)}{\partial w}\right)$$

these depend on the function f.

Backpropagation with Multiple Neurons

- Let $\Delta_j = \frac{\partial Loss}{\partial j}$ be the derivative of the loss w.r.t to the output of unit j.



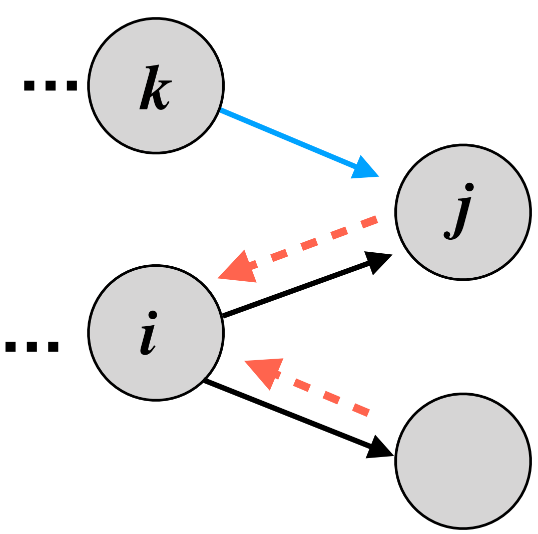
$$\Delta_i = \frac{\partial Loss}{\partial i} = \sum_j \left(\frac{\partial Loss}{\partial j}\right)\left(\frac{\partial j}{\partial i}\right)$$

$$= \sum_j \Delta_j \left(\frac{\partial j}{\partial i}\right)$$

- The output of j is computed during the forward pass.

Backpropagation with Multiple Neurons

- Once the Δ_j have been computed, we can compute the gradients w.r.t to the weights.



$$\frac{\partial Loss}{\partial w_{ij}} = \Delta_j \frac{\partial j}{\partial w_{ij}}$$

Some Neural Network Tricks

- When implementing the model, try to fit to 100% accuracy on 1 or 2 data points.
- Decrease learning rate after each epoch or when loss stops decreasing.
- Find good initial learning rate before optimizing other hyperparameters.
- Try other optimizers:
  - SGD with momentum, rmsprop, adagrad, adadelta, adam

Acknowledgments

- Some slides by Chris Kedzie