

# Neural Network Introduction

LING 575K Deep Learning for NLP

Shane Steinert-Threlkeld

April 6 2022

# Announcements

- HW1 due tomorrow night, upload readme and hw1.tar.gz to Canvas
  - NB: separate files!
  - Do not put readme inside of tar.gz
- indices\_to\_tokens: no error handling
- You can/should use `Vocabulary.from\_text\_files` to build your vocab object
  - Factory design pattern allows for different initialization signatures in Python
  - E.g. from\_csv in pandas, from\_pretrained in huggingface (later this course)
- Note on \*args and \*\*kwargs
  - [https://book.pythontips.com/en/latest/args\\_and\\_kwargs.html](https://book.pythontips.com/en/latest/args_and_kwargs.html)

# \*args and \*\*kwargs

```
def add(a, b):
    return a + b

print(add(1, 2))  # 3
print(add(*(1, 2))) # 3

def add_any(*args):
    return sum(args)

print(add_any(1, 2, 3)) # 6
print(add_any(1, 2, 3, 4)) # 10
```

# \*args and \*\*kwargs

```
def keywords(name="Shane", course="575k"):
    return f"{name} is teaching {course}"

print(keywords(name="Agatha"))
print(keywords(**{"name": "Agatha"}))

def keywords_any(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

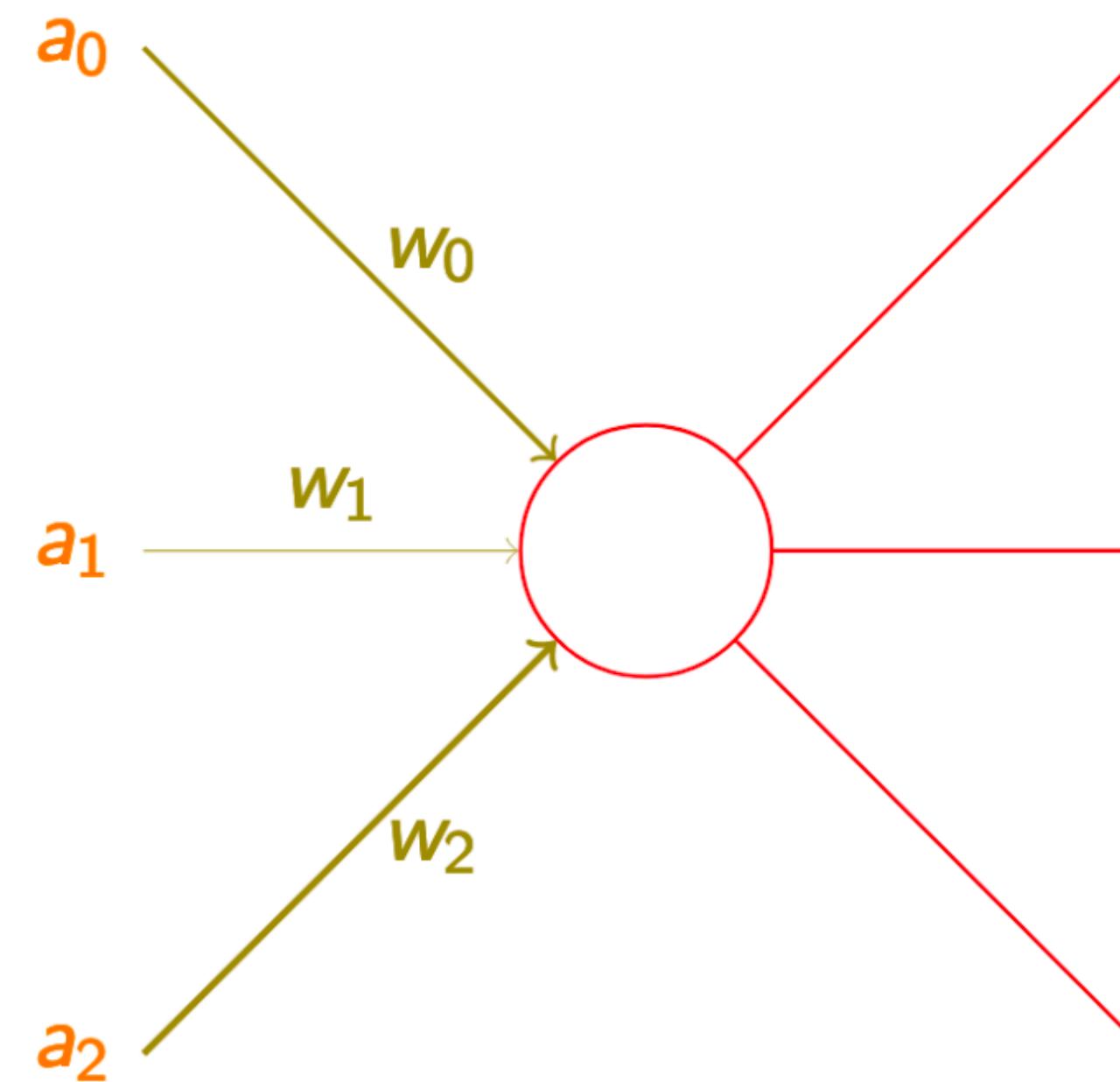
keywords_any(name="Shane", course="575k")
keywords_any(name="Shane", course="575k", foo="bar")
keywords_any(**{"name": "Shane", "course": "575k"})
```

# Plan for Today

- Last time:
  - Prediction-based word vectors
  - Skip-gram with negative sampling [model + loss]
- Today: intro to feed-forward neural networks
  - Basic computation + expressive power
  - Multilayer perceptrons
  - Mini-batches
  - Hyper-parameters and regularization

# Computation: Basic Example

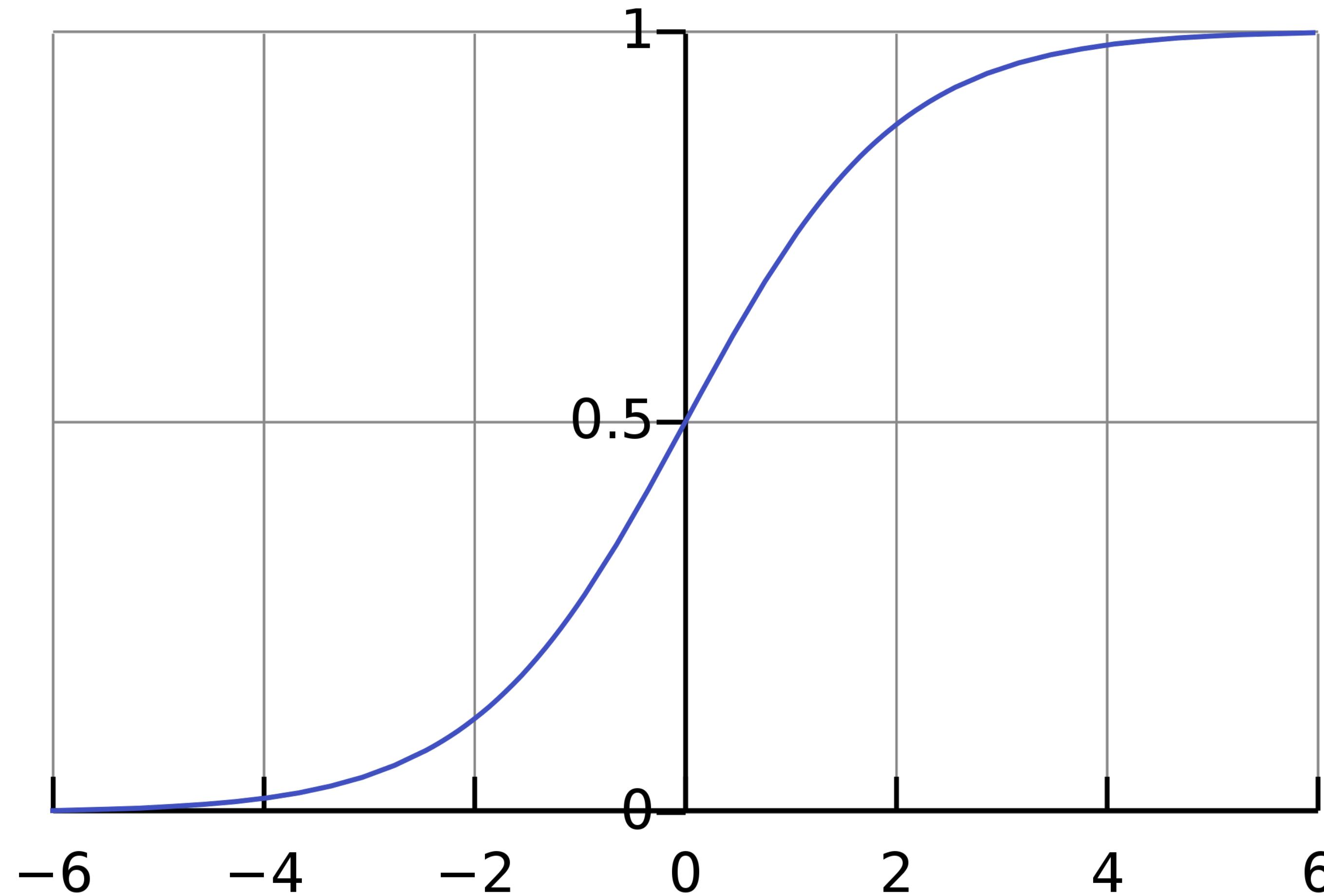
# Artificial Neuron



$$a = f(a_0 \cdot w_0 + a_1 \cdot w_1 + a_2 \cdot w_2)$$

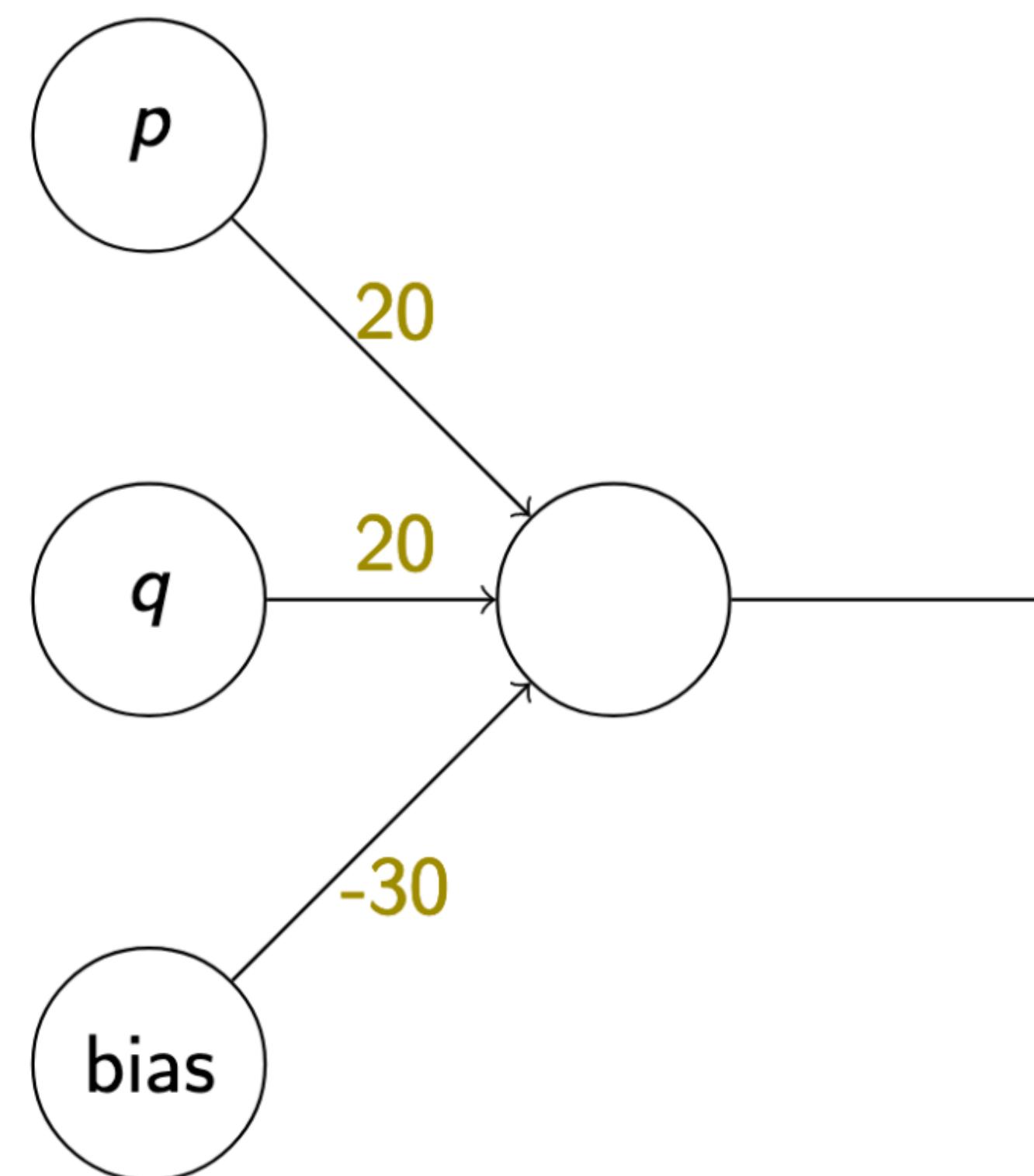
<https://github.com/shanest/nn-tutorial>

# Activation Function: Sigmoid



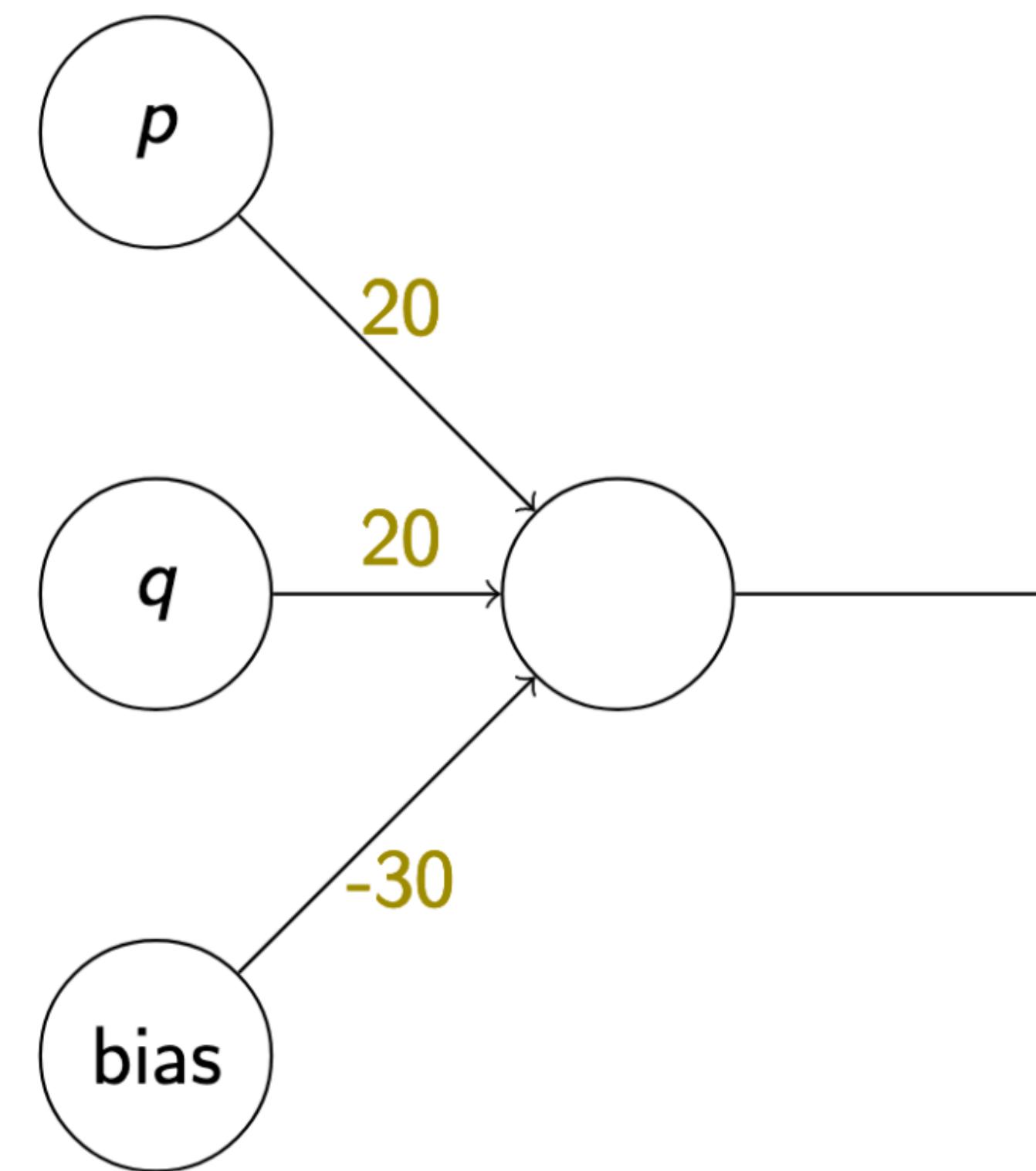
$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

# Computing a Boolean function



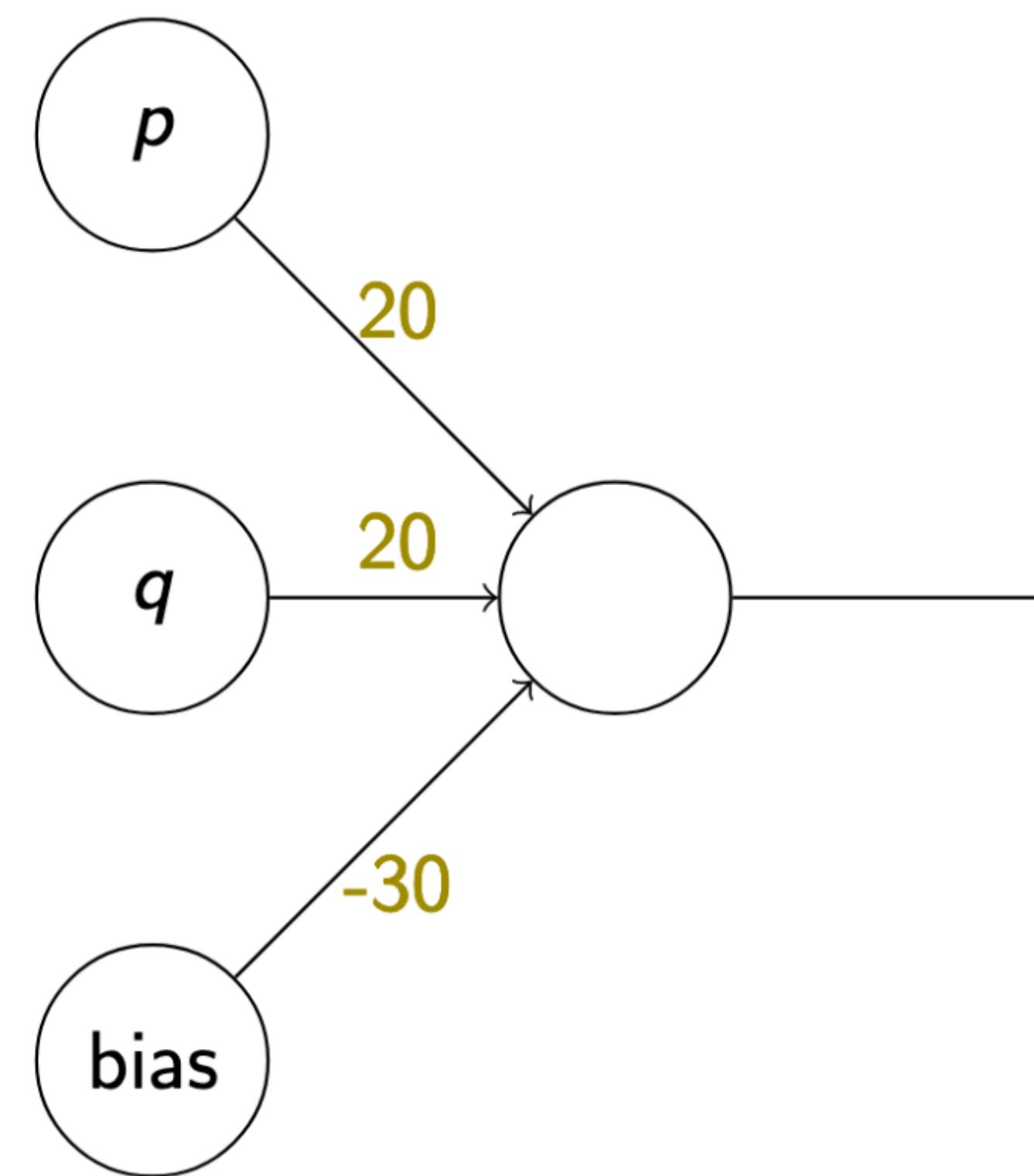
# Computing a Boolean function

p	q	a
---	---	---



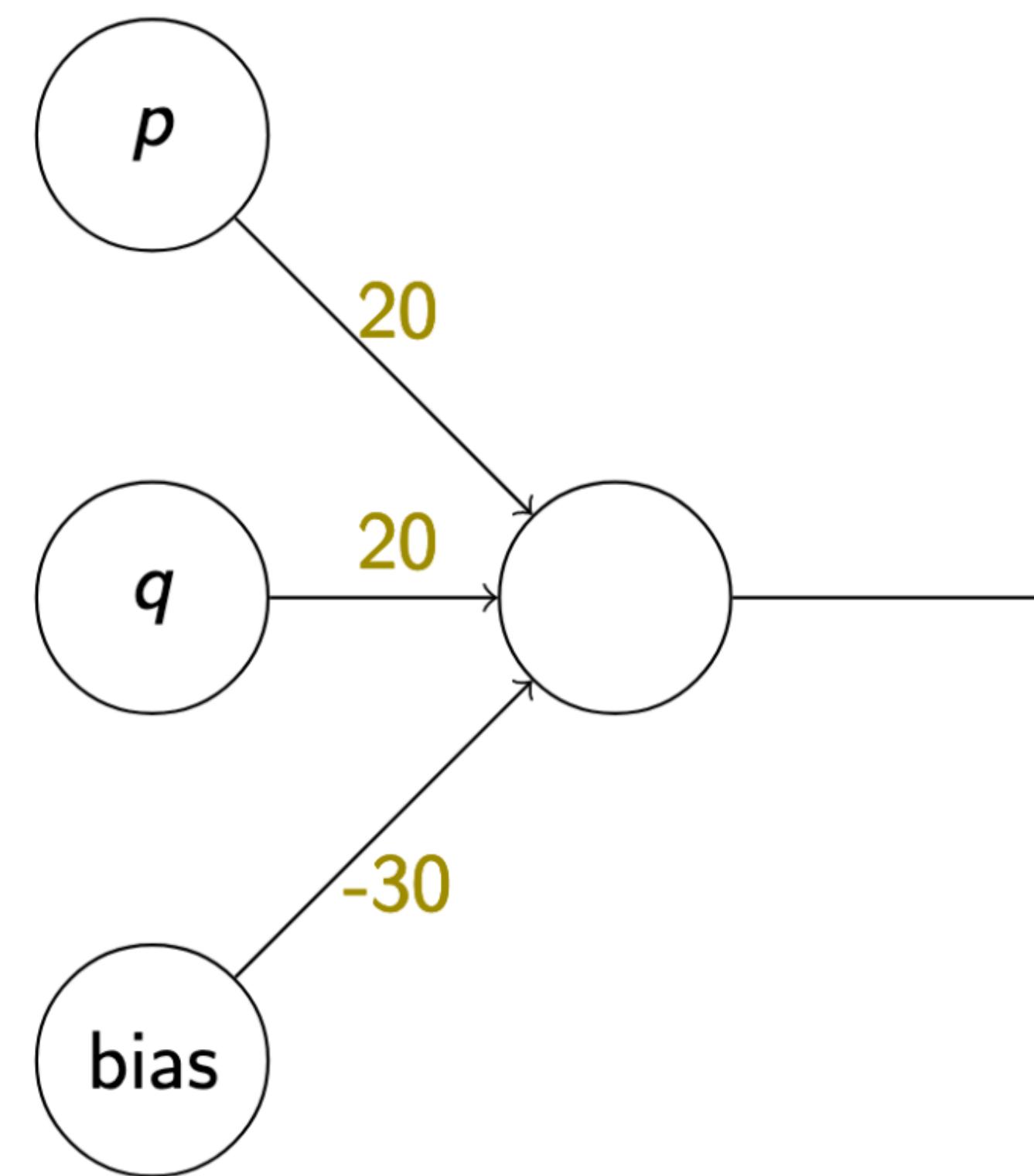
# Computing a Boolean function

p	q	a
1	1	1



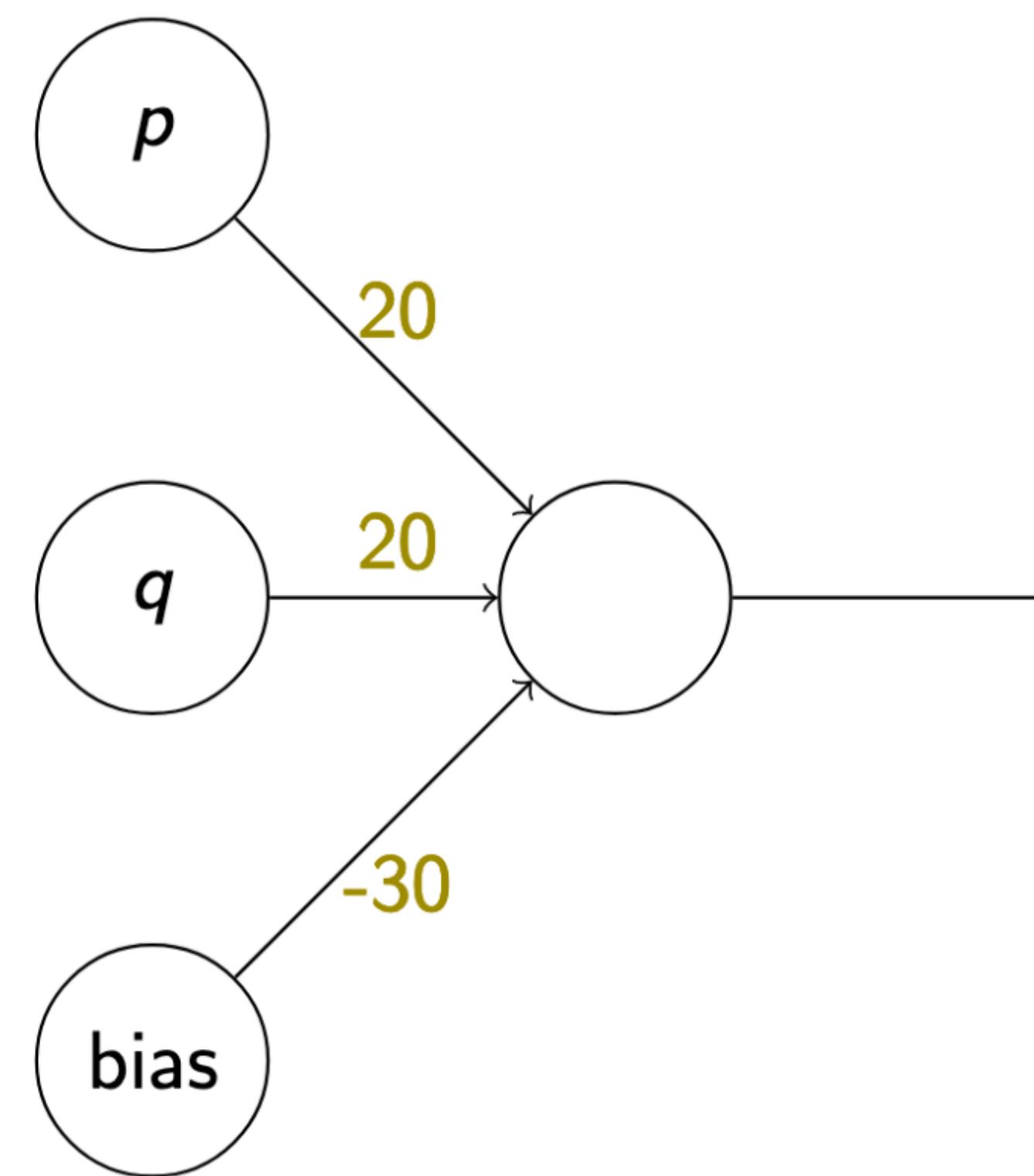
# Computing a Boolean function

p	q	a
1	1	1
1	0	0



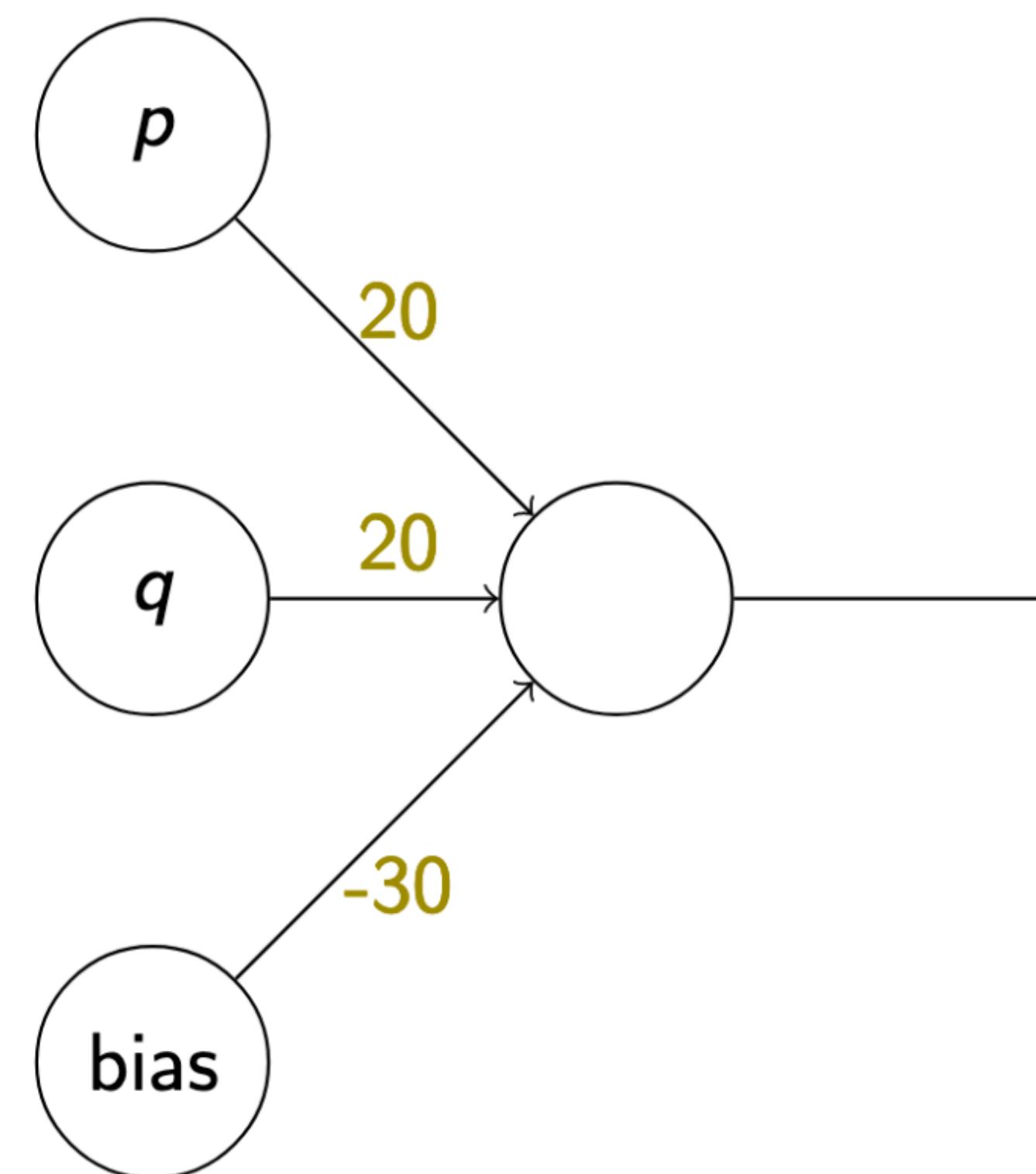
# Computing a Boolean function

p	q	a
1	1	1
1	0	0
0	1	0

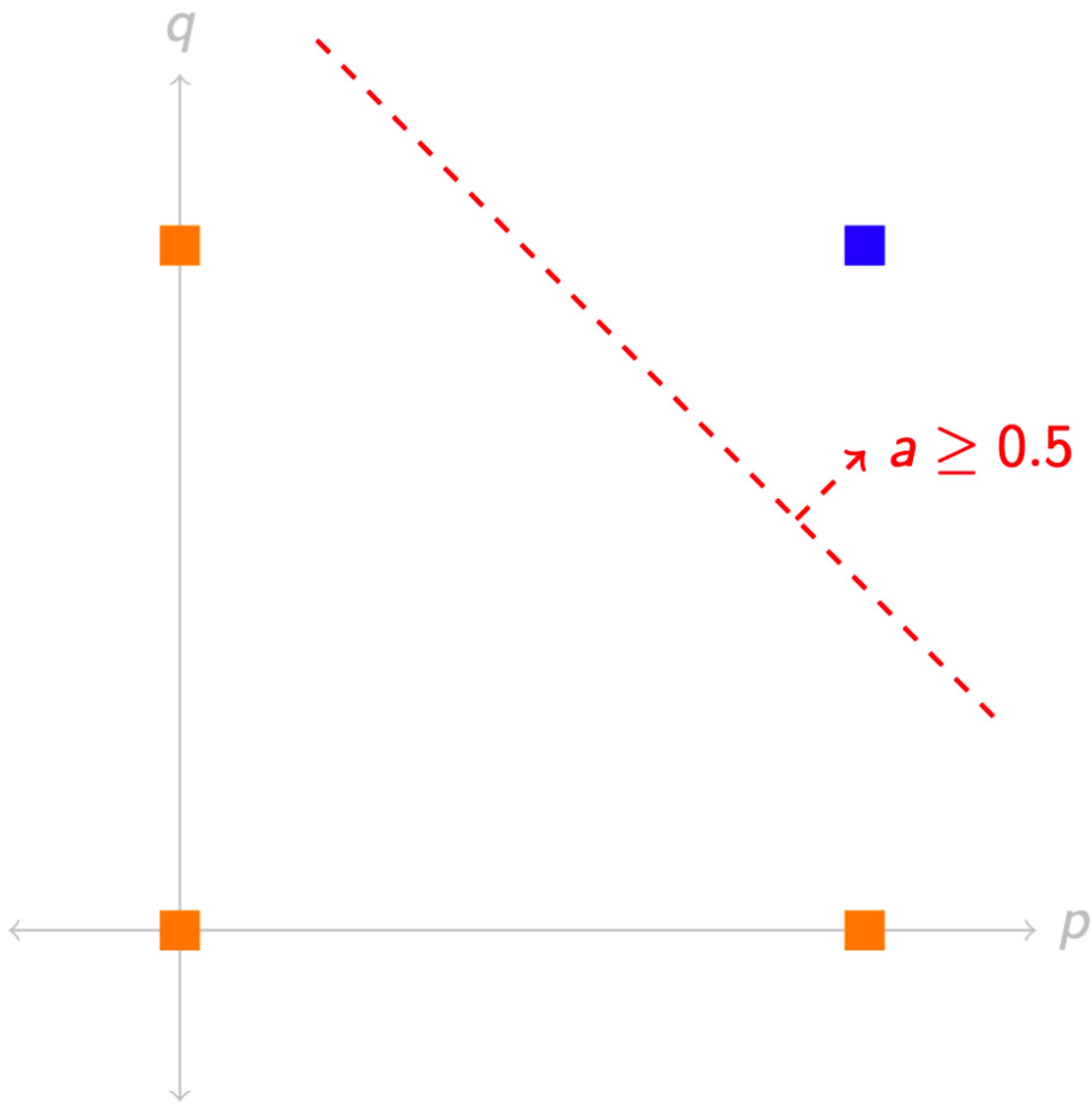


# Computing a Boolean function

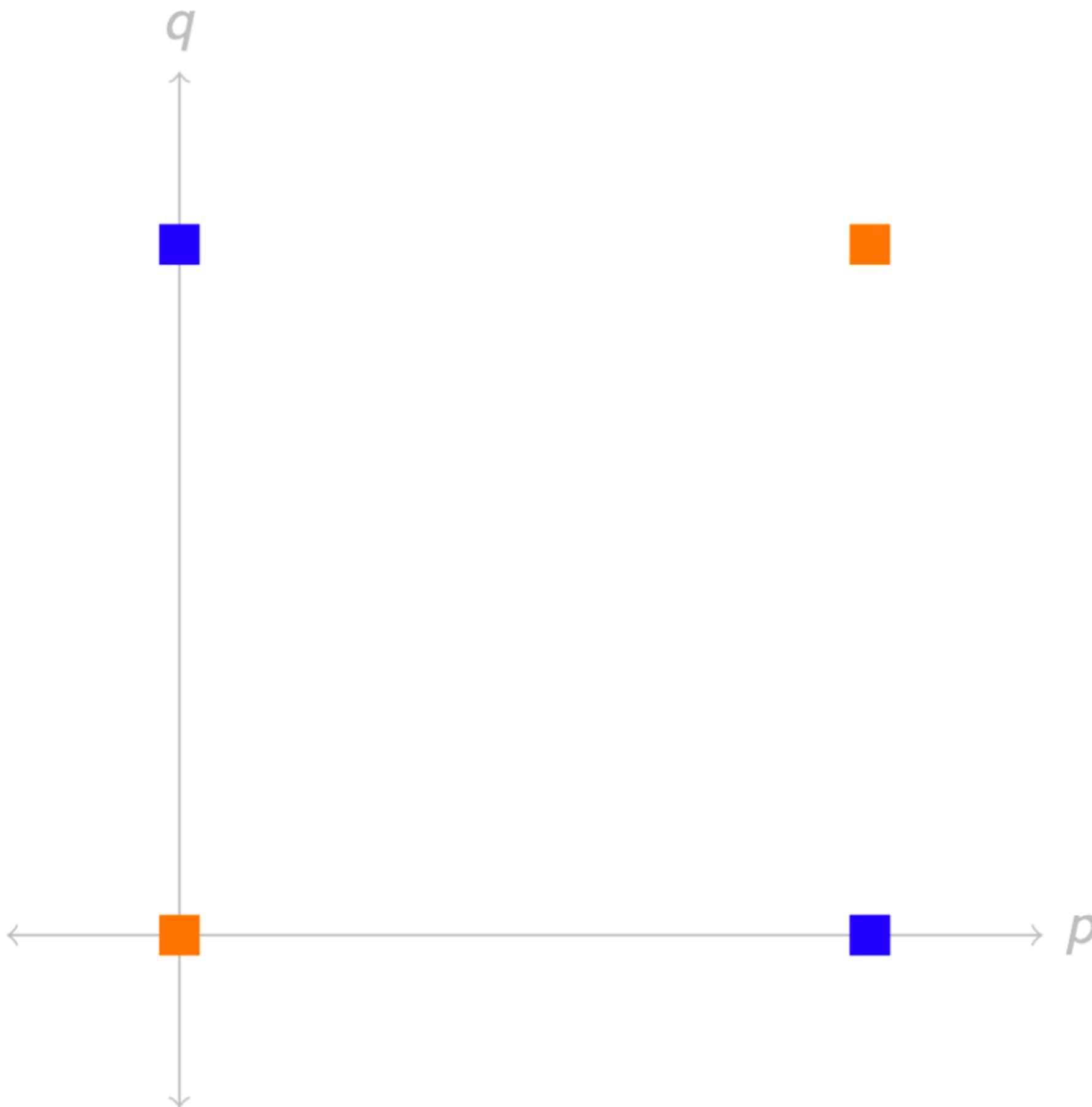
p	q	a
1	1	1
1	0	0
0	1	0
0	0	0



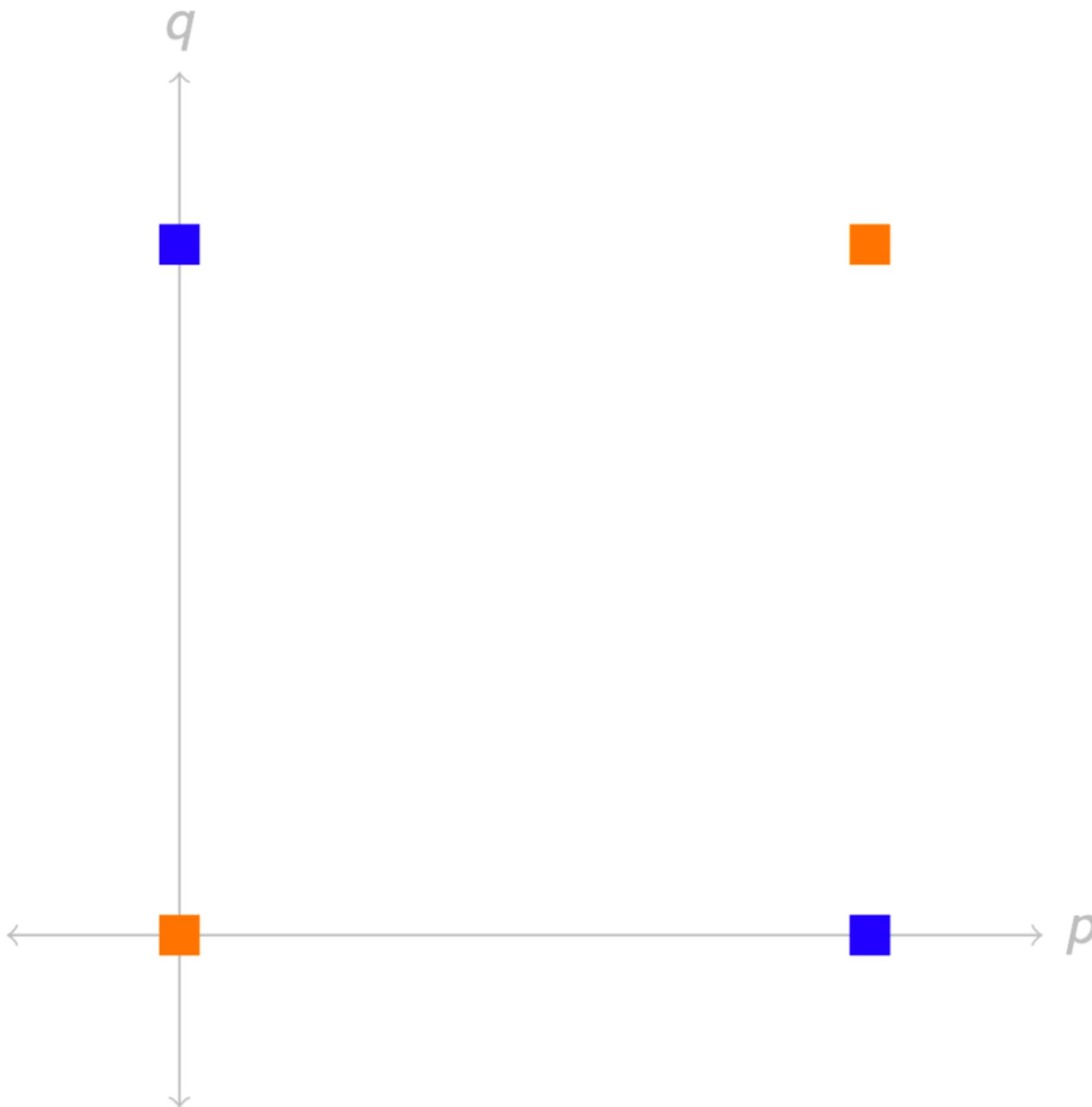
# Computing ‘and’



# The XOR problem

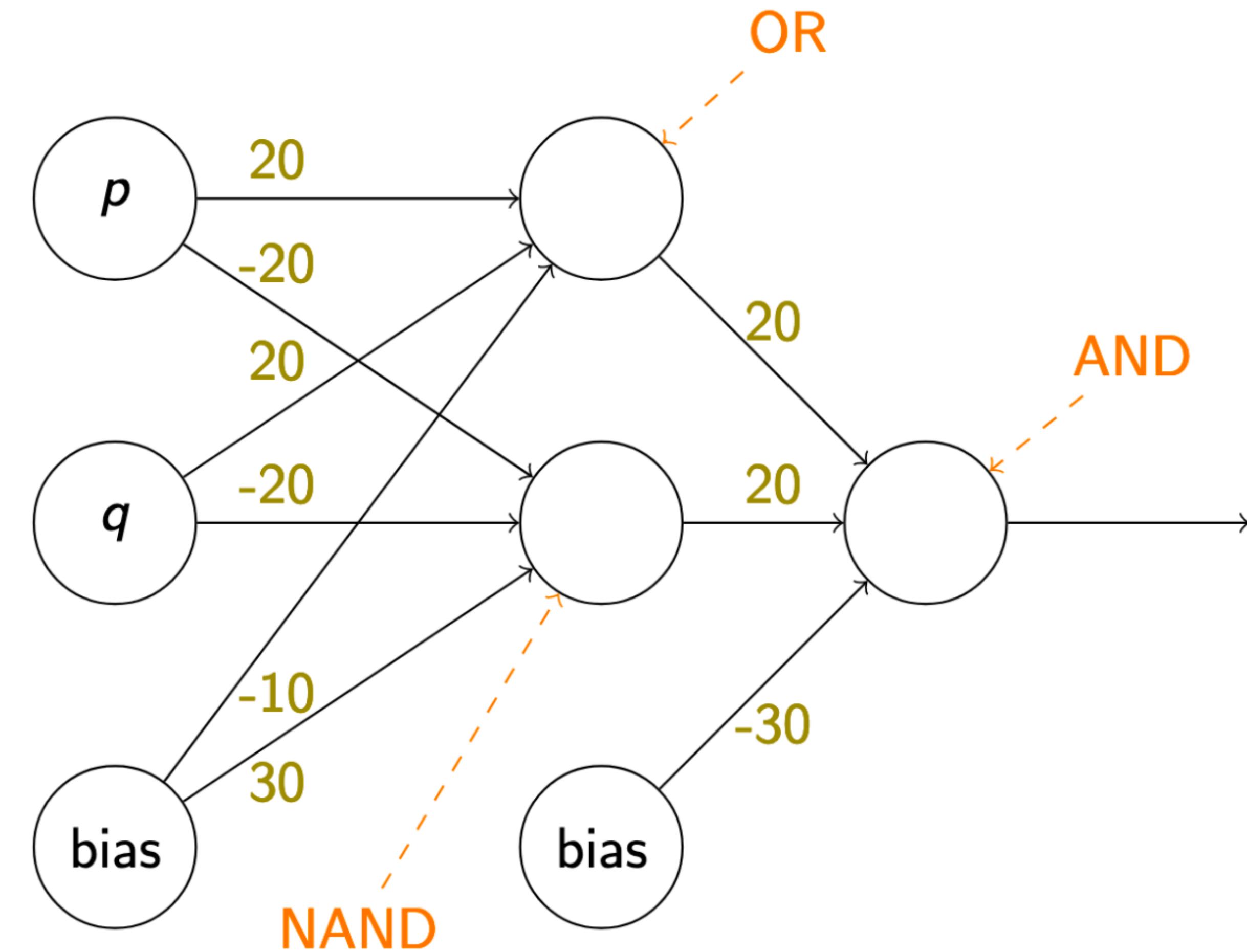


# The XOR problem

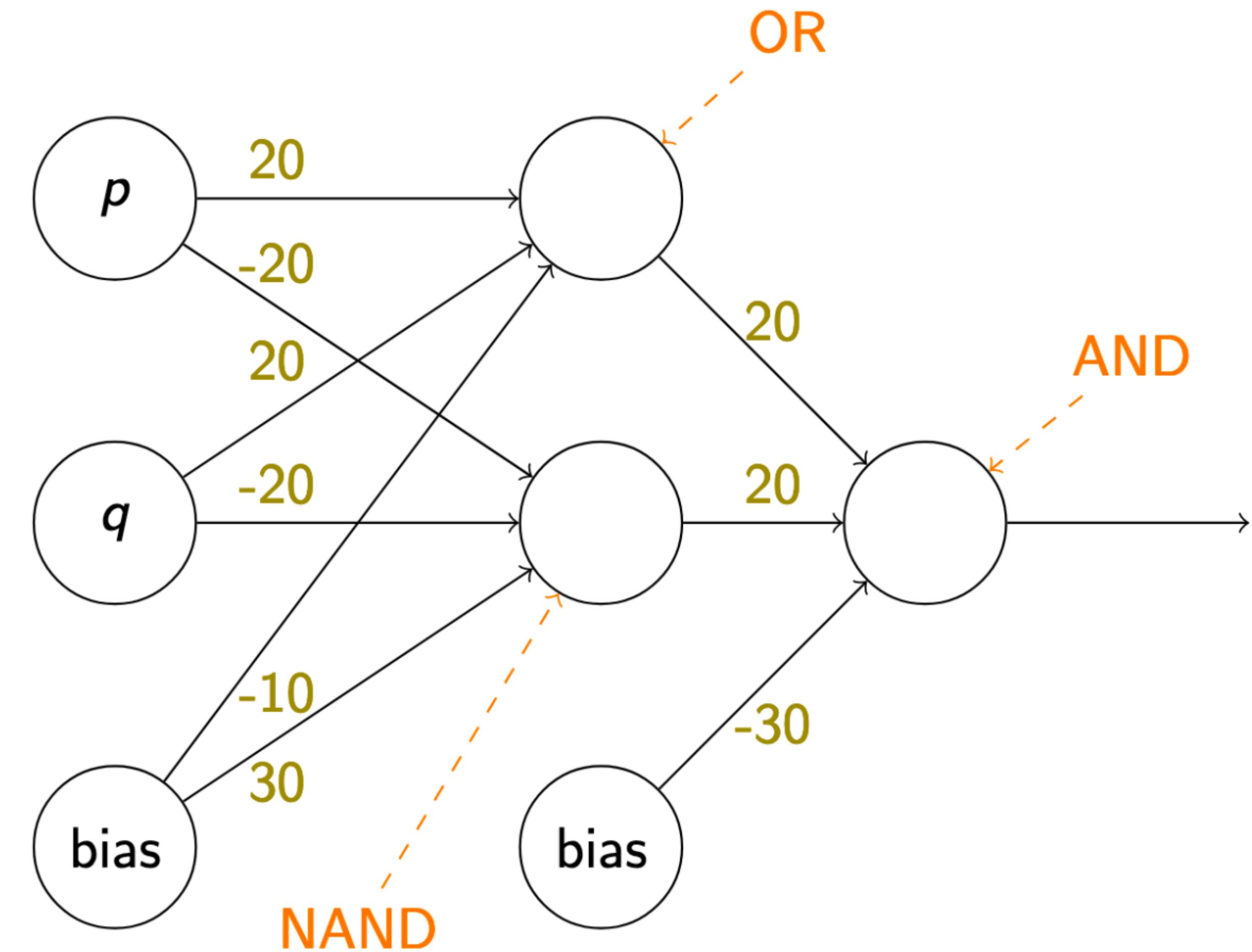


XOR is not linearly separable

# Computing XOR

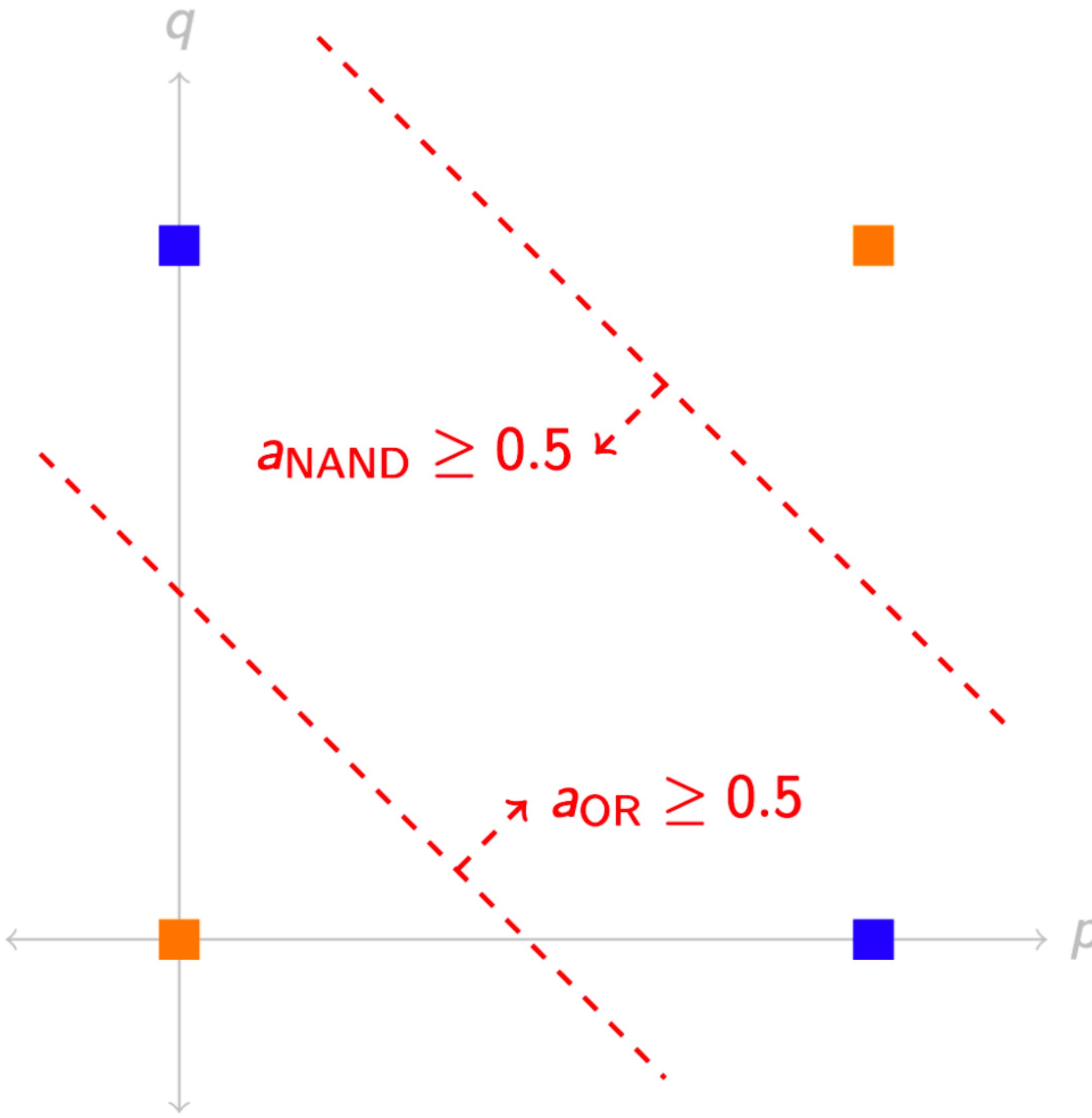


# Computing XOR



Exercise: show that  
NAND behaves as described.

# Computing XOR



# Key Ideas

- Hidden layers compute high-level / abstract features of the input
  - Via training, will *learn which features* are helpful for a given task
  - Caveat: doesn't always learn much more than shallow features
- Doing so *increases the expressive power* of a neural network
  - Strictly more functions can be computed with hidden layers than without

# Expressive Power

# Expressive Power

- Neural networks with *one* hidden layer are *universal function approximators*

# Expressive Power

- Neural networks with *one* hidden layer are *universal function approximators*
- Let  $f: [0,1]^m \rightarrow \mathbb{R}$  be continuous and  $\epsilon > 0$ . Then there is a one-hidden-layer neural network  $g$  with sigmoid activation such that  $|f(\mathbf{x}) - g(\mathbf{x})| < \epsilon$  for all  $\mathbf{x} \in [0,1]^m$ .

# Expressive Power

- Neural networks with *one* hidden layer are *universal function approximators*
- Let  $f: [0,1]^m \rightarrow \mathbb{R}$  be continuous and  $\epsilon > 0$ . Then there is a one-hidden-layer neural network  $g$  with sigmoid activation such that  $|f(\mathbf{x}) - g(\mathbf{x})| < \epsilon$  for all  $\mathbf{x} \in [0,1]^m$ .
- Generalizations (diff activation functions, less bounded, etc.) exist.

# Expressive Power

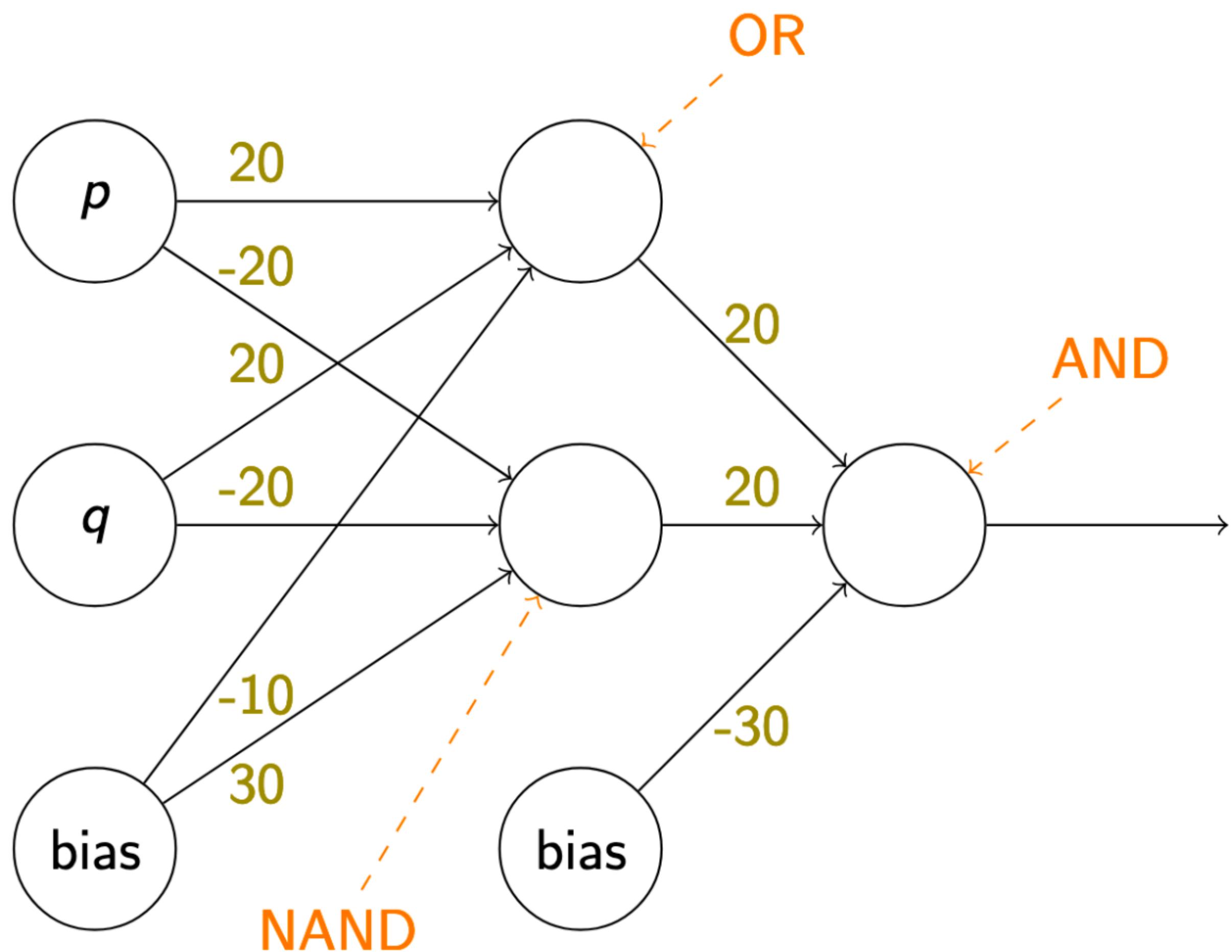
- Neural networks with *one* hidden layer are *universal function approximators*
- Let  $f: [0,1]^m \rightarrow \mathbb{R}$  be continuous and  $\epsilon > 0$ . Then there is a one-hidden-layer neural network  $g$  with sigmoid activation such that  $|f(\mathbf{x}) - g(\mathbf{x})| < \epsilon$  for all  $\mathbf{x} \in [0,1]^m$ .
- Generalizations (diff activation functions, less bounded, etc.) exist.
- But:
  - Size of the hidden layer is *exponential* in  $m$
  - How does one *find/learn* such a good approximation?

# Expressive Power

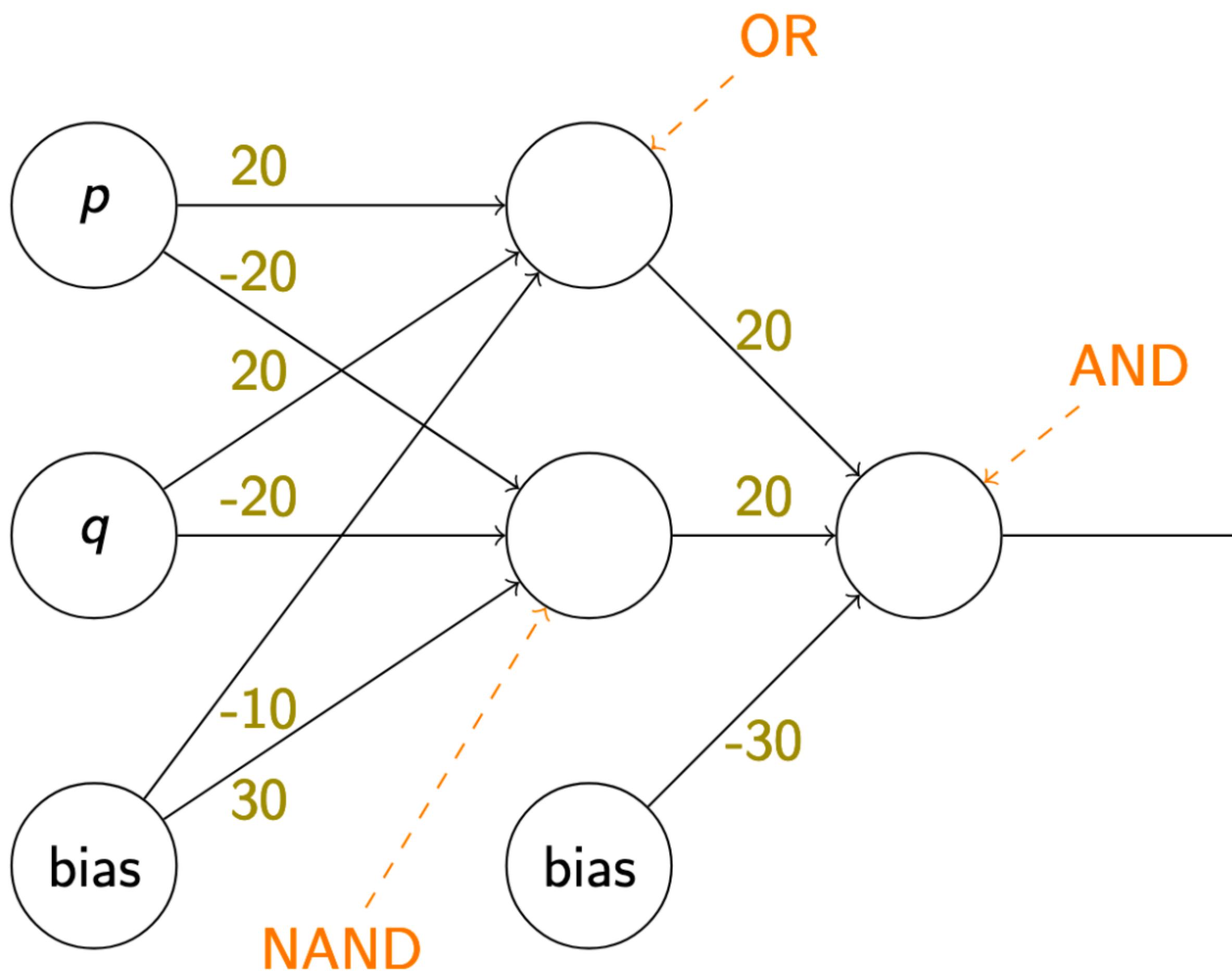
- Neural networks with *one* hidden layer are *universal function approximators*
- Let  $f: [0,1]^m \rightarrow \mathbb{R}$  be continuous and  $\epsilon > 0$ . Then there is a one-hidden-layer neural network  $g$  with sigmoid activation such that  $|f(\mathbf{x}) - g(\mathbf{x})| < \epsilon$  for all  $\mathbf{x} \in [0,1]^m$ .
- Generalizations (diff activation functions, less bounded, etc.) exist.
- But:
  - Size of the hidden layer is *exponential* in  $m$
  - How does one *find/learn* such a good approximation?
- Nice walkthrough: <http://neuralnetworksanddeeplearning.com/chap4.html>

# Feed-forward networks aka Multi-layer perceptrons (MLP)

# XOR Network

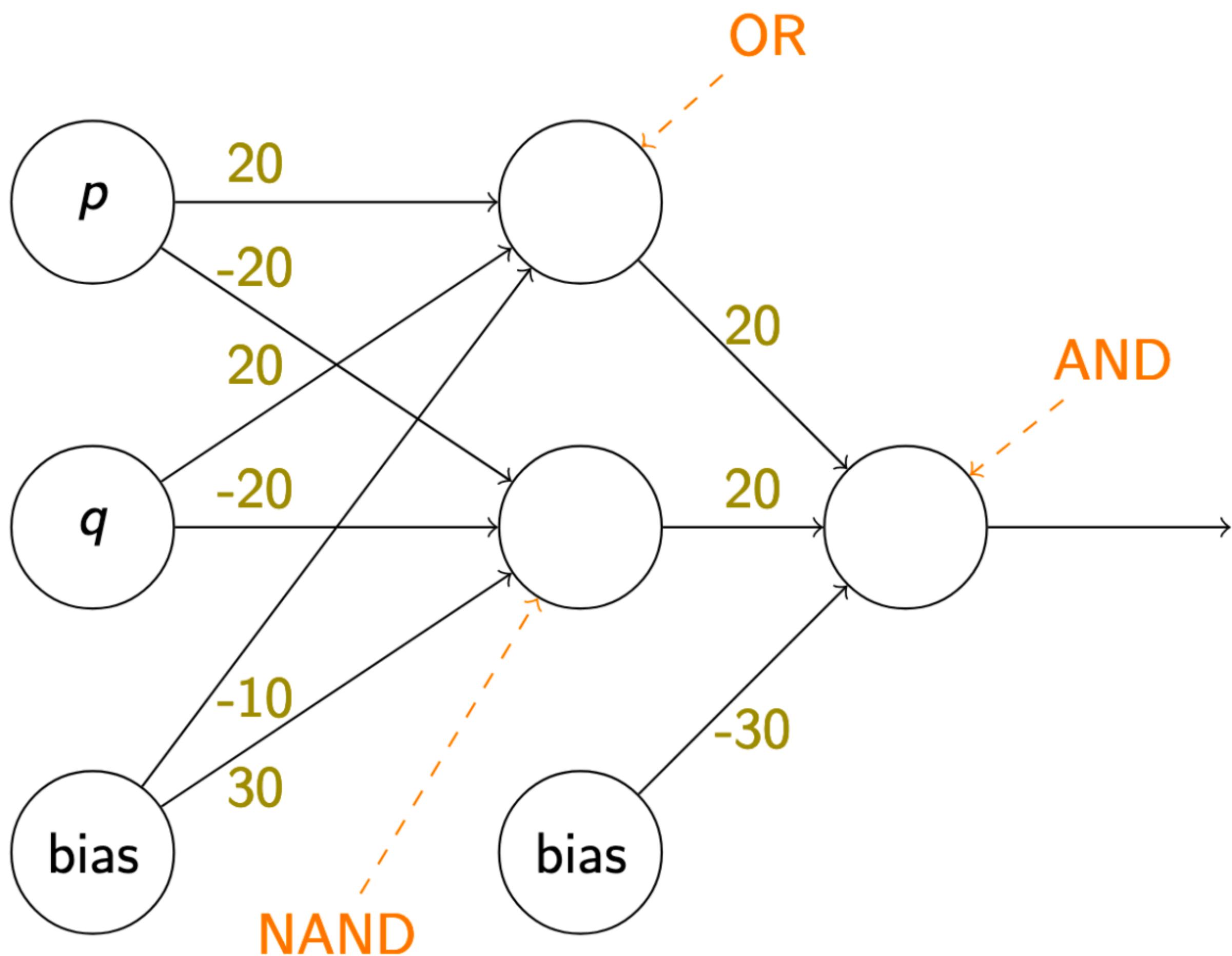


# XOR Network



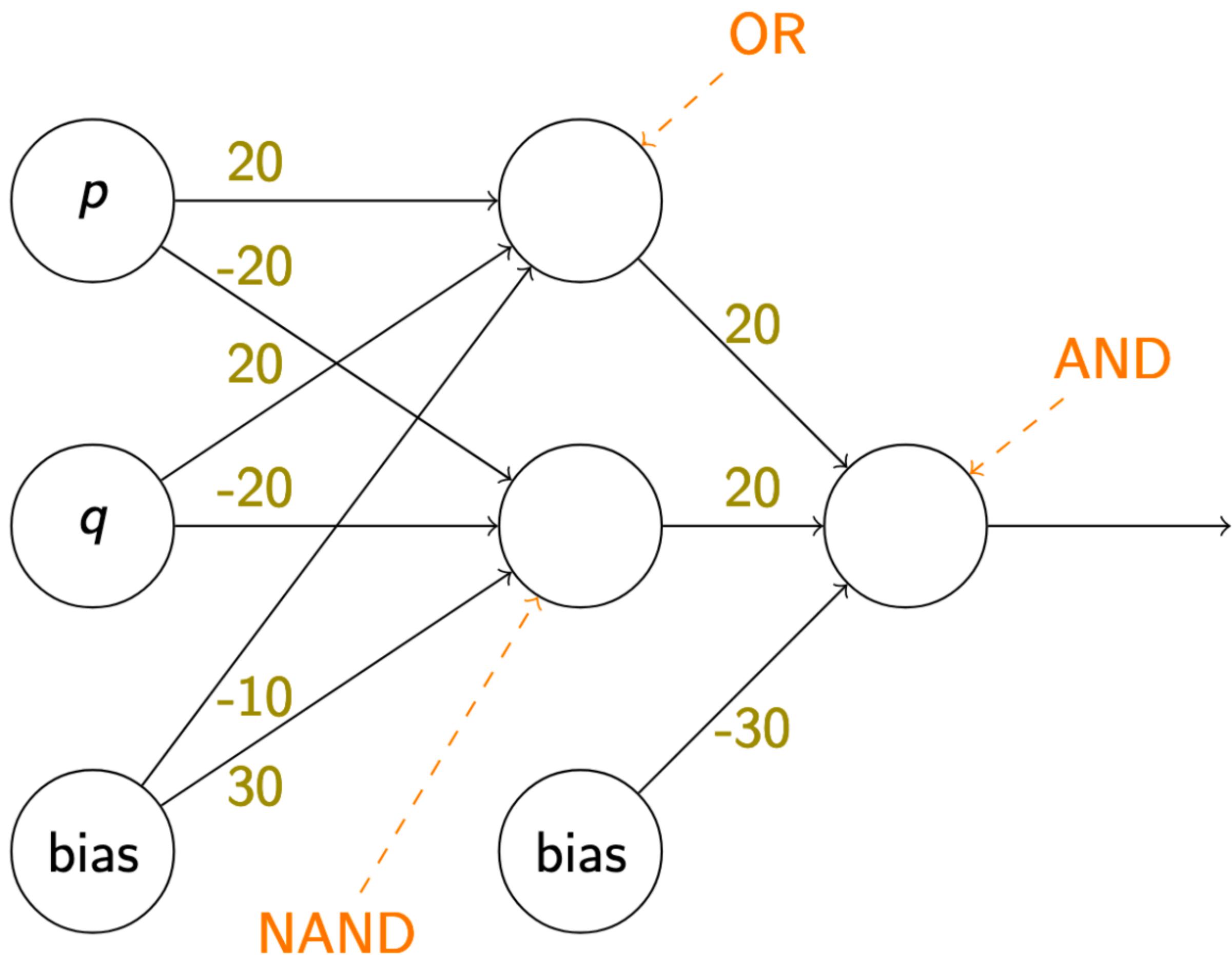
$$a_{\text{and}} = \sigma(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}})$$

# XOR Network



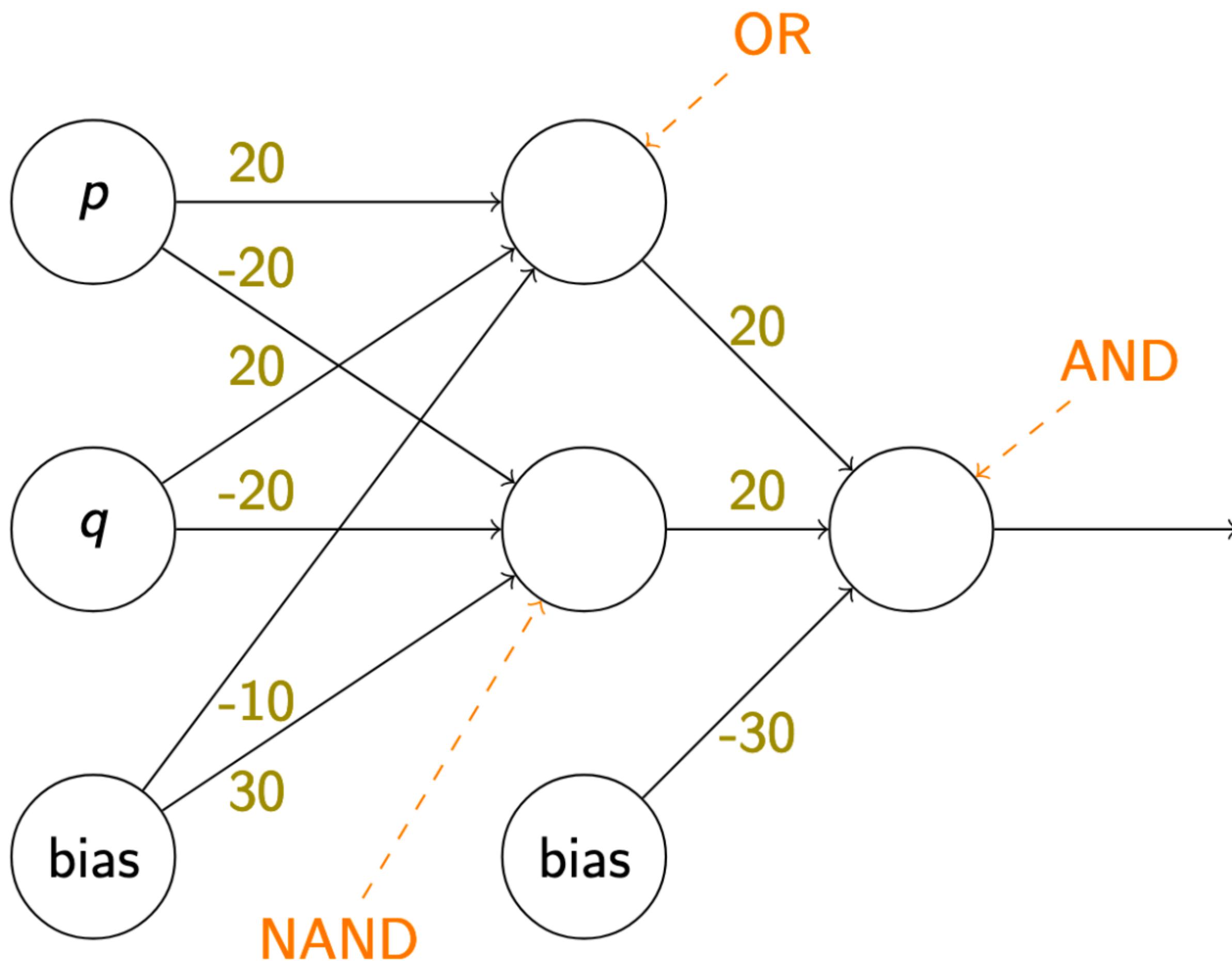
$$\begin{aligned}a_{\text{and}} &= \sigma(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}) \\&= \sigma \left( [a_{\text{or}} \quad a_{\text{nand}}] \begin{bmatrix} w_{\text{or}}^{\text{and}} \\ w_{\text{nand}}^{\text{and}} \end{bmatrix} + b^{\text{and}} \right)\end{aligned}$$

# XOR Network



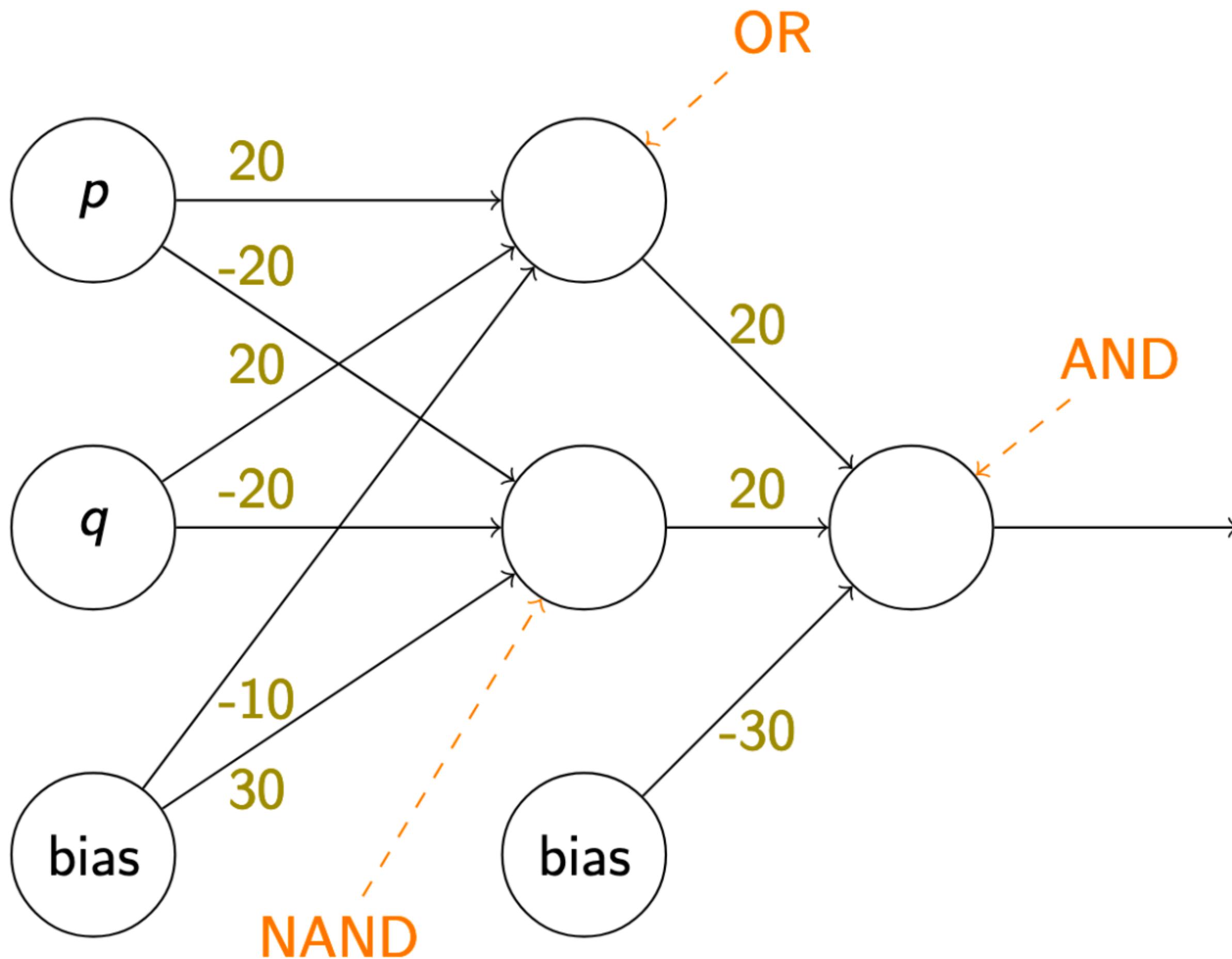
$$\begin{aligned}a_{\text{and}} &= \sigma(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}) \\&= \sigma \left( [a_{\text{or}} \quad a_{\text{nand}}] \begin{bmatrix} w_{\text{or}}^{\text{and}} \\ w_{\text{nand}}^{\text{and}} \end{bmatrix} + b^{\text{and}} \right)\end{aligned}$$

# XOR Network



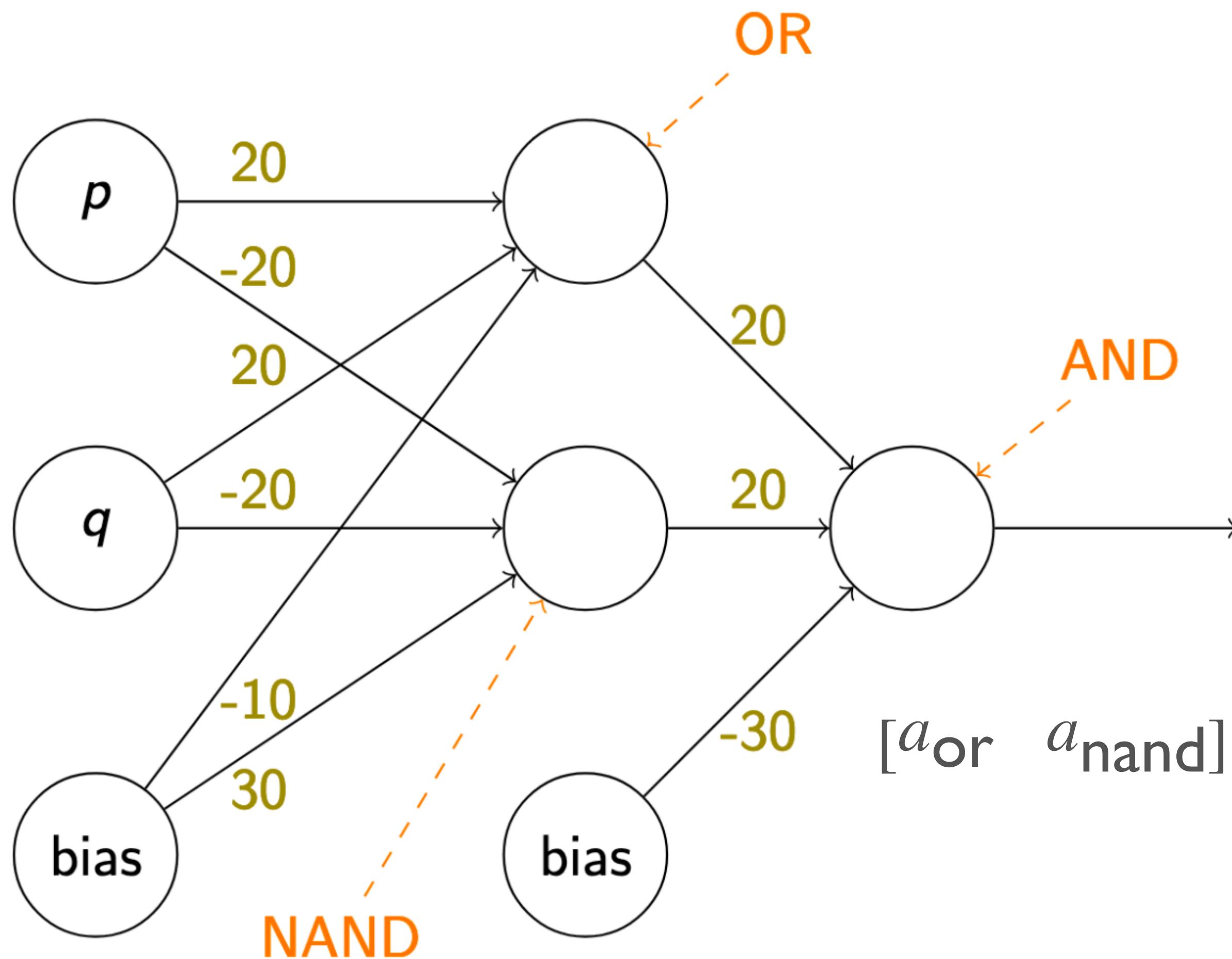
$$\begin{aligned}
 a_{\text{and}} &= \sigma(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}) \\
 &= \sigma \left( [a_{\text{or}} \quad a_{\text{nand}}] \begin{bmatrix} w_{\text{or}}^{\text{and}} \\ w_{\text{nand}}^{\text{and}} \end{bmatrix} + b^{\text{and}} \right) \\
 a_{\text{or}} &= \sigma(w_p^{\text{or}} \cdot a_p + w_q^{\text{or}} \cdot a_q + b^{\text{or}})
 \end{aligned}$$

# XOR Network



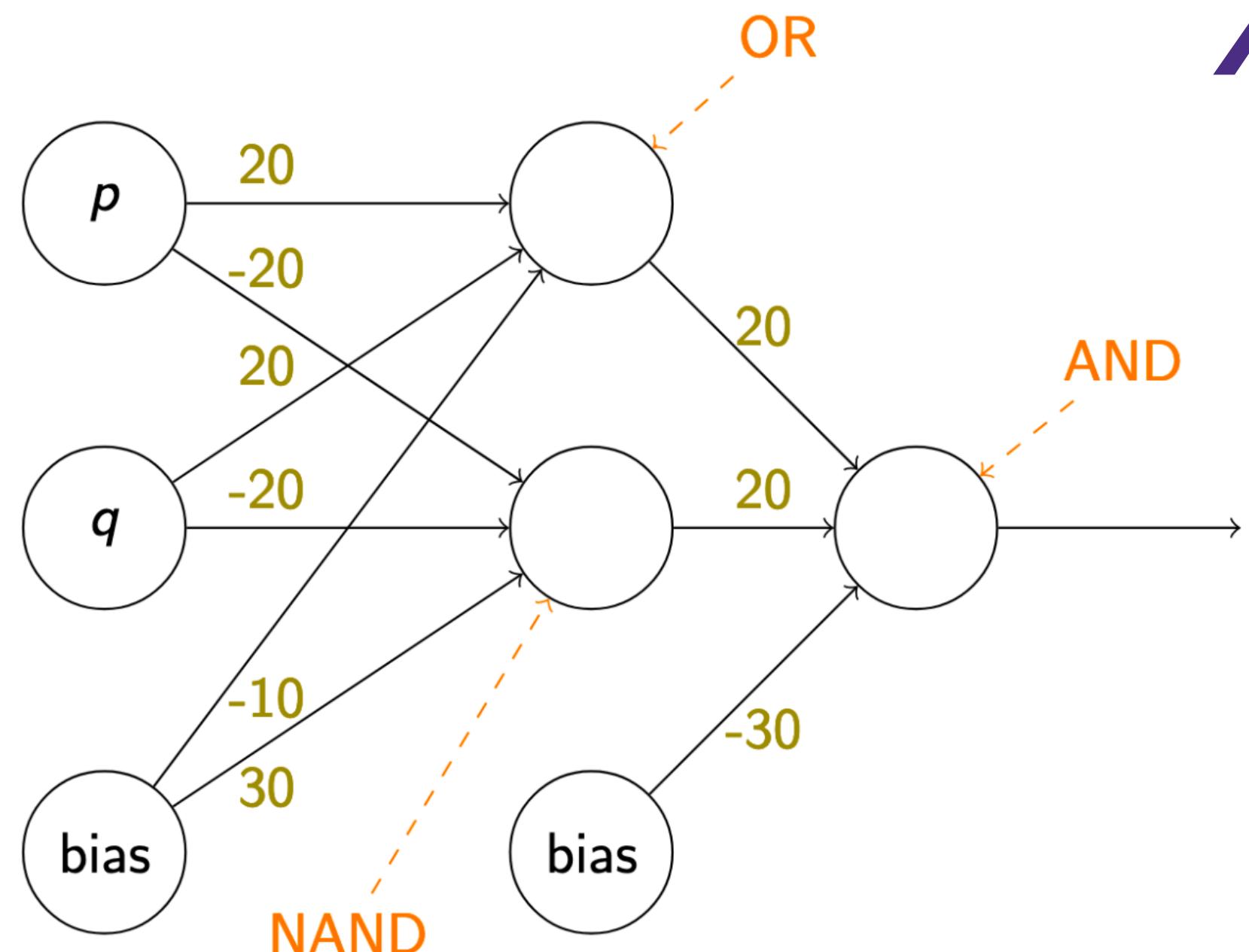
$$\begin{aligned}
 a_{\text{and}} &= \sigma(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}) \\
 &= \sigma \left( [a_{\text{or}} \quad a_{\text{nand}}] \begin{bmatrix} w_{\text{or}}^{\text{and}} \\ w_{\text{nand}}^{\text{and}} \end{bmatrix} + b^{\text{and}} \right) \\
 a_{\text{or}} &= \sigma(w_p^{\text{or}} \cdot a_p + w_q^{\text{or}} \cdot a_q + b^{\text{or}}) \\
 a_{\text{nand}} &= \sigma(w_p^{\text{nand}} \cdot a_p + w_q^{\text{nand}} \cdot a_q + b^{\text{nand}})
 \end{aligned}$$

# XOR Network



$$\begin{aligned}
 a_{\text{and}} &= \sigma(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}) \\
 &= \sigma \left( [a_{\text{or}} \quad a_{\text{nand}}] \begin{bmatrix} w_{\text{or}}^{\text{and}} \\ w_{\text{nand}}^{\text{and}} \end{bmatrix} + b^{\text{and}} \right) \\
 [a_{\text{or}} \quad a_{\text{nand}}] &= \sigma \left( [a_p \quad a_q] \begin{bmatrix} w_p^{\text{or}} & w_p^{\text{nand}} \\ w_q^{\text{or}} & w_q^{\text{nand}} \end{bmatrix} + [b^{\text{or}} \quad b^{\text{nand}}] \right)
 \end{aligned}$$

# XOR Network



$$a_{\text{and}} = \sigma \left( \sigma \left( \begin{bmatrix} a_p & a_q \end{bmatrix} \begin{bmatrix} w_p^{\text{or}} & w_p^{\text{nand}} \\ w_q^{\text{or}} & w_q^{\text{nand}} \end{bmatrix} + \begin{bmatrix} b^{\text{or}} & b^{\text{nand}} \end{bmatrix} \right) \begin{bmatrix} w_{\text{or}}^{\text{and}} \\ w_{\text{nand}}^{\text{and}} \end{bmatrix} + b^{\text{and}} \right)$$

$$a_{\text{and}} = \sigma \left( w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}} \right)$$

$$= \sigma \left( \begin{bmatrix} a_{\text{or}} & a_{\text{nand}} \end{bmatrix} \begin{bmatrix} w_{\text{or}}^{\text{and}} \\ w_{\text{nand}}^{\text{and}} \end{bmatrix} + b^{\text{and}} \right)$$

# Generalizing

$$a_{\text{and}} = \sigma \left( \sigma \left( \begin{bmatrix} a_p & a_q \end{bmatrix} \begin{bmatrix} w_p^{\text{or}} & w_p^{\text{nand}} \\ w_q^{\text{or}} & w_q^{\text{nand}} \end{bmatrix} + \begin{bmatrix} b^{\text{or}} & b^{\text{nand}} \end{bmatrix} \right) \begin{bmatrix} w^{\text{and}} \\ w^{\text{or}} \\ w^{\text{nand}} \end{bmatrix} + b^{\text{and}} \right)$$

# Generalizing

$$a_{\text{and}} = \sigma \left( \sigma \left( \begin{bmatrix} a_p & a_q \end{bmatrix} \begin{bmatrix} w_p^{\text{or}} & w_p^{\text{nand}} \\ w_q^{\text{or}} & w_q^{\text{nand}} \end{bmatrix} + \begin{bmatrix} b^{\text{or}} & b^{\text{nand}} \end{bmatrix} \right) \begin{bmatrix} w^{\text{and}} \\ w^{\text{or}} \\ w^{\text{nand}} \end{bmatrix} + b^{\text{and}} \right)$$

$$\hat{y} = f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right)$$

# Generalizing

$$a_{\text{and}} = \sigma \left( \sigma \left( \begin{bmatrix} a_p & a_q \end{bmatrix} \begin{bmatrix} w_p^{\text{or}} & w_p^{\text{nand}} \\ w_q^{\text{or}} & w_q^{\text{nand}} \end{bmatrix} + \begin{bmatrix} b^{\text{or}} & b^{\text{nand}} \end{bmatrix} \right) \begin{bmatrix} w^{\text{and}} \\ w^{\text{or}} \\ w^{\text{nand}} \end{bmatrix} + b^{\text{and}} \right)$$

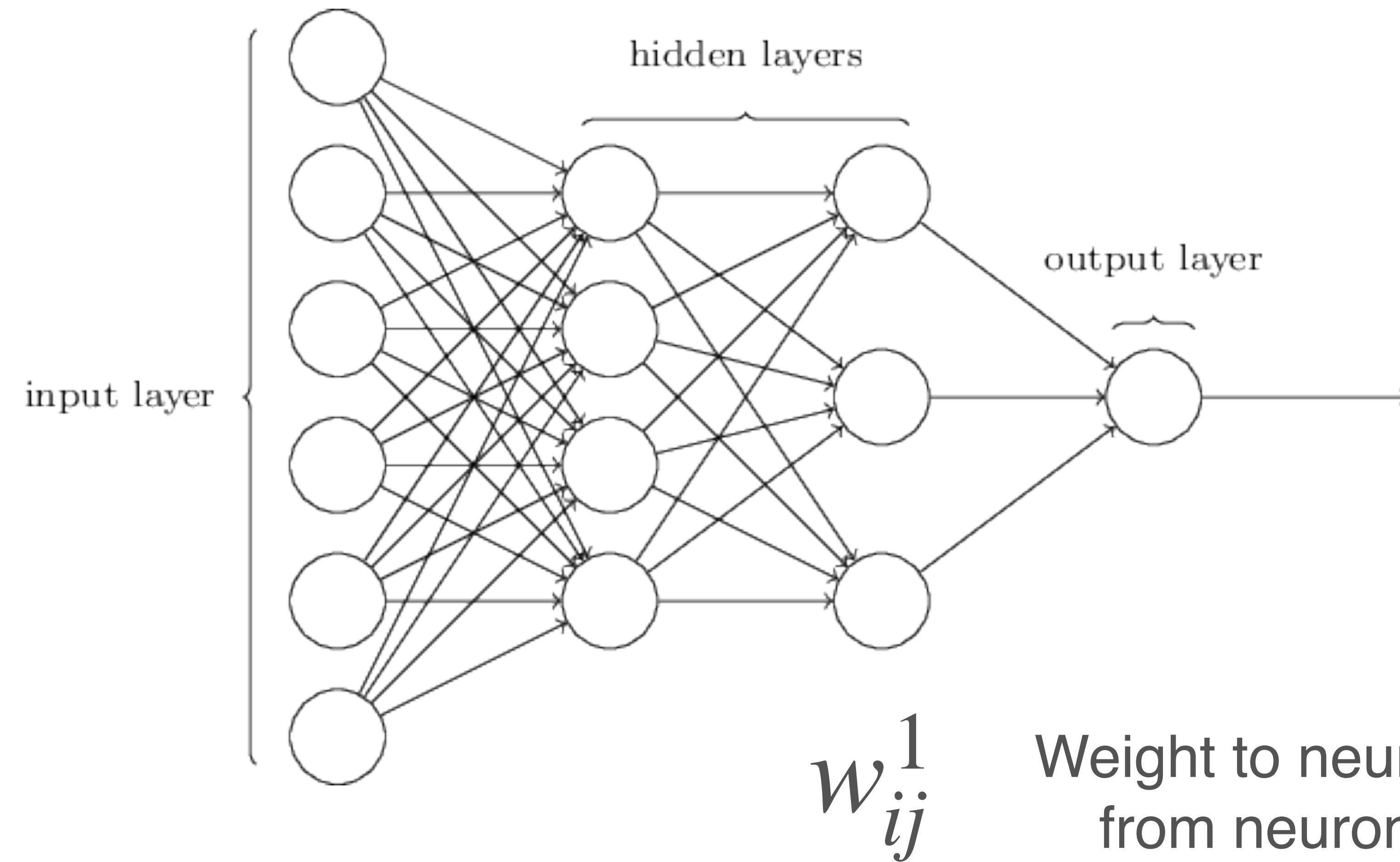
$$\hat{y} = f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right)$$

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

# Some terminology

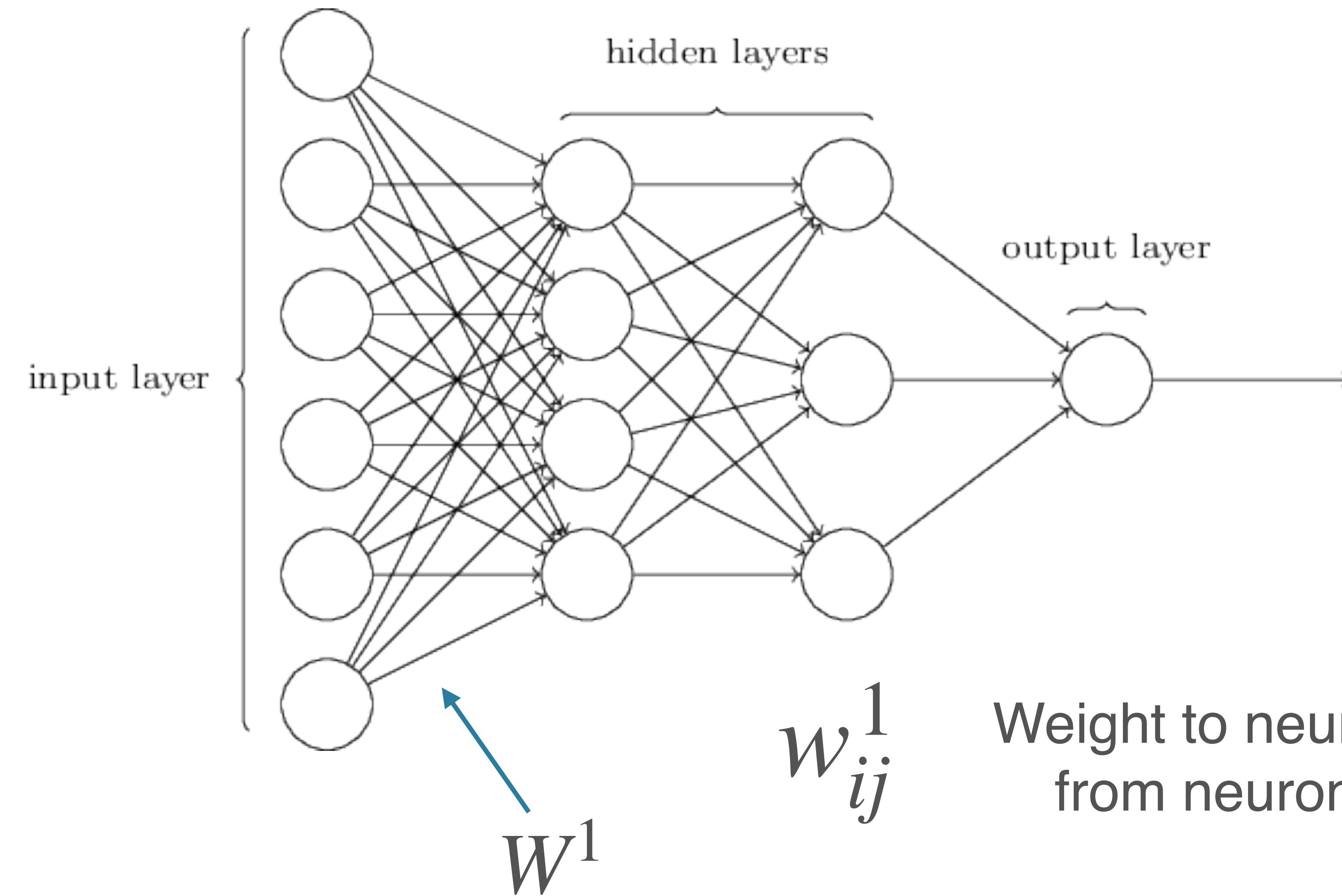
- Our XOR network is a *feed-forward neural network* with *one hidden layer*
  - Aka a multi-layer perceptron (MLP)
  - Input nodes: 2; output nodes: 1
  - Activation function: sigmoid

# General MLP



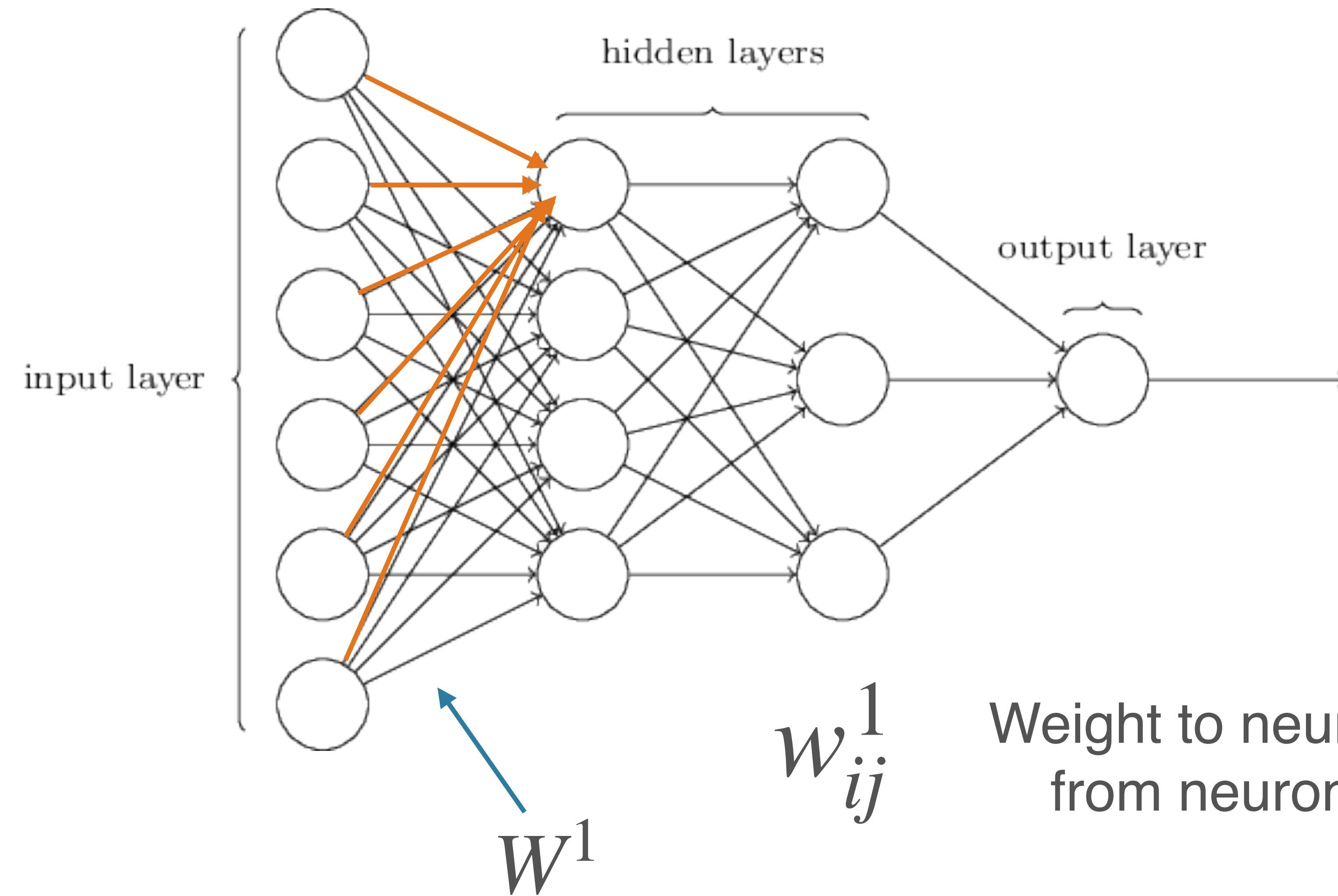
[source](#)

# General MLP



[source](#)

# General MLP



source

Weight to neuron j in layer 1  
from neuron i in layer 0

# General MLP

# General MLP

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

# General MLP

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

$$x = [x_0 \quad x_1 \quad \cdots \quad x_{n_0}]$$

Shape:  $(1, n_0)$

# General MLP

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

$$x = [x_0 \quad x_1 \quad \cdots \quad x_{n_0}]$$

Shape:  $(1, n_0)$

$$W^1 = \begin{bmatrix} w_{00}^1 & w_{10}^1 & \cdots & w_{0n_1}^1 \\ w_{10}^1 & w_{11}^1 & \cdots & w_{1n_1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_00}^1 & w_{n_01}^1 & \cdots & w_{n_0n_1}^1 \end{bmatrix}$$

# General MLP

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

$$x = [x_0 \quad x_1 \quad \cdots \quad x_{n_0}]$$

Shape:  $(1, n_0)$

$$W^1 = \begin{bmatrix} w_{00}^1 & w_{10}^1 & \cdots & w_{0n_1}^1 \\ w_{10}^1 & w_{11}^1 & \cdots & w_{1n_1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_00}^1 & w_{n_01}^1 & \cdots & w_{n_0n_1}^1 \end{bmatrix}$$

Shape:  $(n_0, n_1)$

$n_0$ : number of neurons in layer 0 (input)

$n_1$ : number of neurons in layer 1

# General MLP

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

$$x = [x_0 \ x_1 \ \cdots \ x_{n_0}]$$

Shape:  $(1, n_0)$

$$b^1 = [b_0^1 \ b_1^1 \ \cdots \ b_{n_1}^1]$$

Shape:  $(1, n_1)$

$$W^1 = \begin{bmatrix} w_{00}^1 & w_{10}^1 & \cdots & w_{0n_1}^1 \\ w_{10}^1 & w_{11}^1 & \cdots & w_{1n_1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_00}^1 & w_{n_01}^1 & \cdots & w_{n_0n_1}^1 \end{bmatrix}$$

Shape:  $(n_0, n_1)$

$n_0$ : number of neurons in layer 0 (input)

$n_1$ : number of neurons in layer 1

# Parameters of an MLP

- Weights and biases
  - For each layer  $l$ :  $n_l(n_{l-1} + 1)$
  - $n_l n_{l-1}$  weights;  $n_l$  biases
- With  $n$  hidden layers (considering the output as a hidden layer):

$$\sum_{i=1}^n n_i(n_{i-1} + 1)$$

# Hyper-parameters of an MLP

# Hyper-parameters of an MLP

- Input size, output size
  - Usually fixed by your problem / dataset
  - Input: image size, vocab size; number of “raw” features in general
  - Output: 1 for binary classification or simple regression, number of labels for classification, ...

# Hyper-parameters of an MLP

- Input size, output size
  - Usually fixed by your problem / dataset
  - Input: image size, vocab size; number of “raw” features in general
  - Output: 1 for binary classification or simple regression, number of labels for classification, ...
- *Number* of hidden layers

# Hyper-parameters of an MLP

- Input size, output size
  - Usually fixed by your problem / dataset
  - Input: image size, vocab size; number of “raw” features in general
  - Output: 1 for binary classification or simple regression, number of labels for classification, ...
- *Number* of hidden layers
- For each hidden layer:
  - Size
  - Activation function

# Hyper-parameters of an MLP

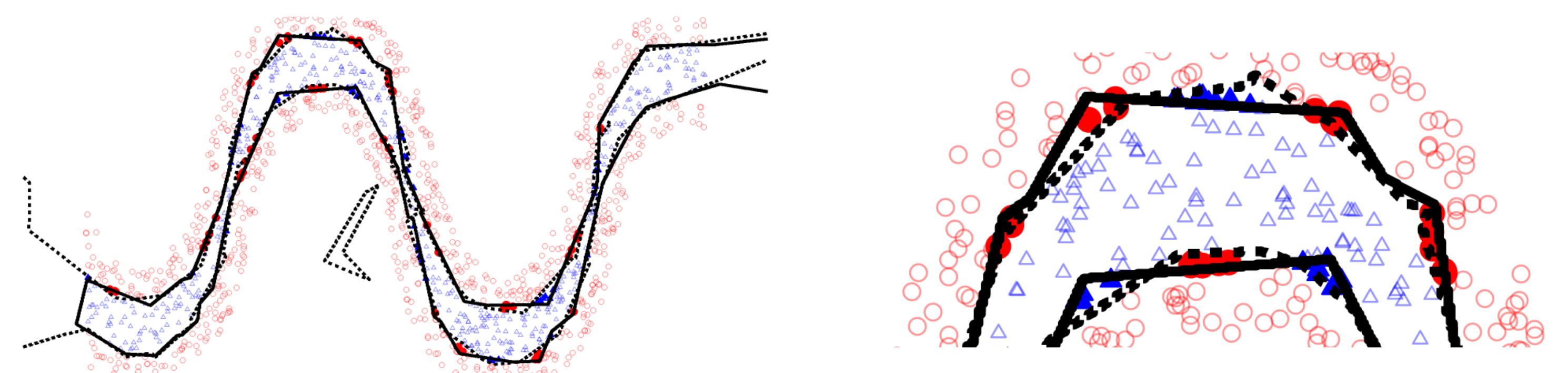
- Input size, output size
  - Usually fixed by your problem / dataset
  - Input: image size, vocab size; number of “raw” features in general
  - Output: 1 for binary classification or simple regression, number of labels for classification, ...
- *Number* of hidden layers
- For each hidden layer:
  - Size
  - Activation function
- Others: initialization, regularization (and associated values), learning rate / training, ...

# The Deep in Deep Learning

- The Universal Approximation Theorem says that one hidden layer suffices for arbitrarily-closely approximating a given function
- Empirical drawbacks: Super-exponentially many neurons; hard to discover
- “Deep and narrow” >> “Shallow and wide” (some theoretical analysis)
  - In principle allows hierarchical features to be learned
  - More well-behaved w/r/t optimization

# The Deep in Deep Learning

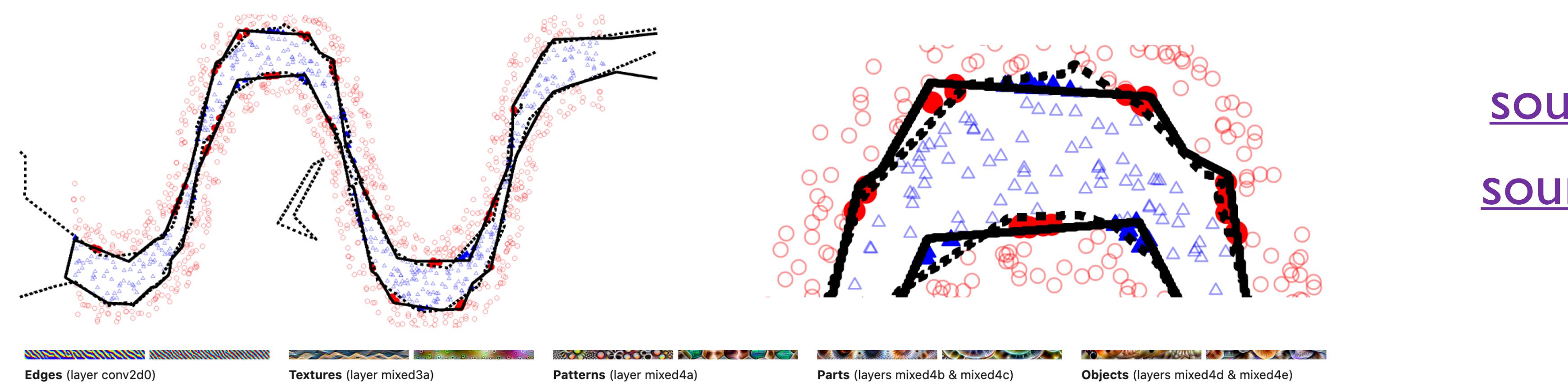
- The Universal Approximation Theorem says that one hidden layer suffices for arbitrarily-closely approximating a given function
- Empirical drawbacks: Super-exponentially many neurons; hard to discover
- “Deep and narrow” >> “Shallow and wide” (some theoretical analysis)
  - In principle allows hierarchical features to be learned
  - More well-behaved w/r/t optimization



source

# The Deep in Deep Learning

- The Universal Approximation Theorem says that one hidden layer suffices for arbitrarily-closely approximating a given function
- Empirical drawbacks: Super-exponentially many neurons; hard to discover
- “Deep and narrow” >> “Shallow and wide” (some theoretical analysis)
  - In principle allows hierarchical features to be learned
  - More well-behaved w/r/t optimization



# Activation Functions

- Note: *non-linear* activation functions are essential
- MLP: linear transformation, followed by a point-wise non-linearity, repeated several times over
- Without the non-linearity, would just have several linear transformations
  - Composition of linear transformations is *also* linear!

# Activation Functions

- Note: *non-linear* activation functions are essential
- MLP: linear transformation, followed by a point-wise non-linearity, repeated several times over
- Without the non-linearity, would just have several linear transformations
  - Composition of linear transformations is *also* linear!

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

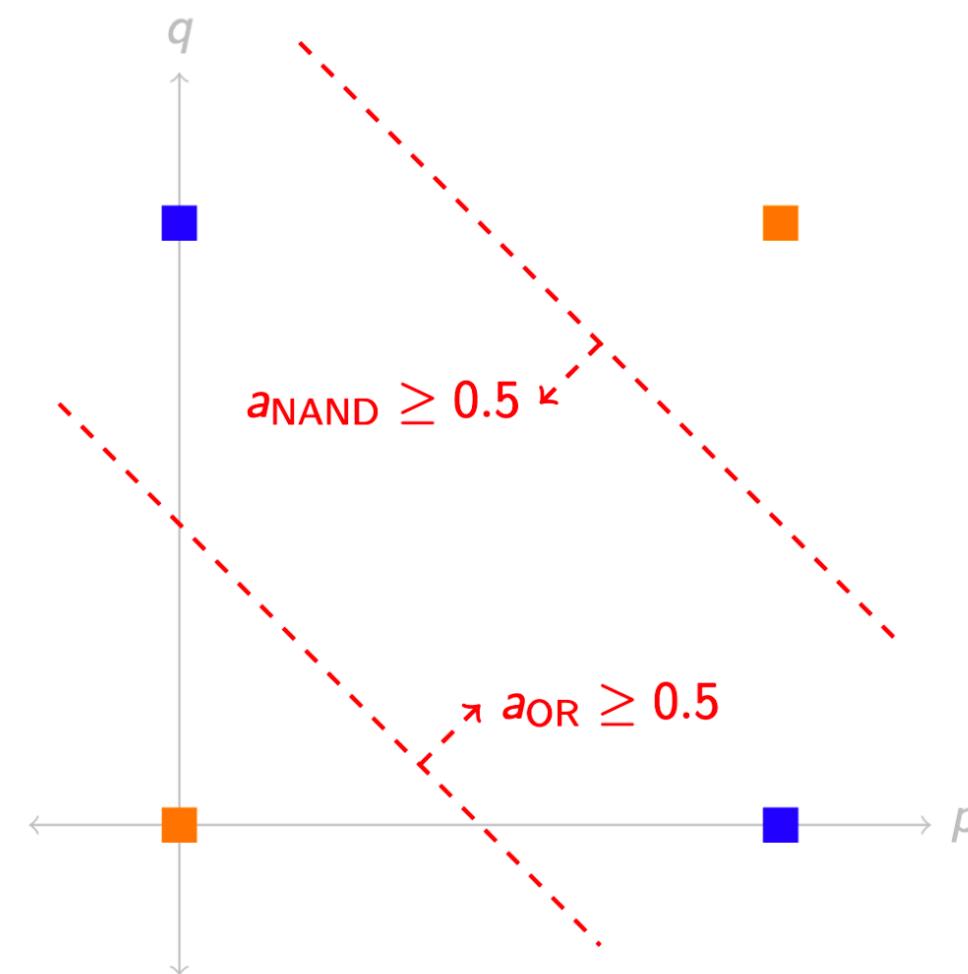
# Non-linearity, cont.

# Non-linearity, cont.

- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions

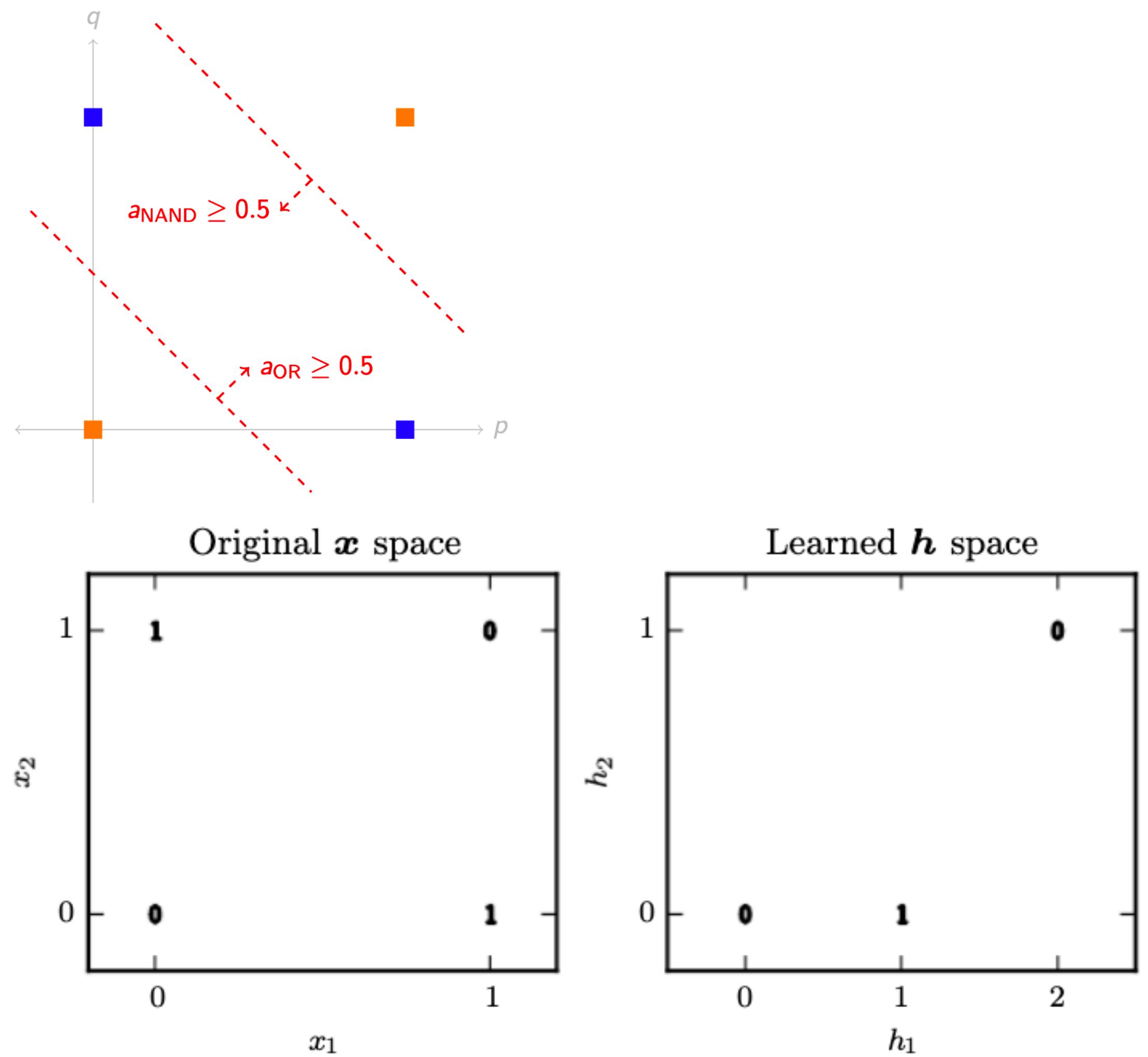
# Non-linearity, cont.

- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions
- One perspective: integrating extracted features



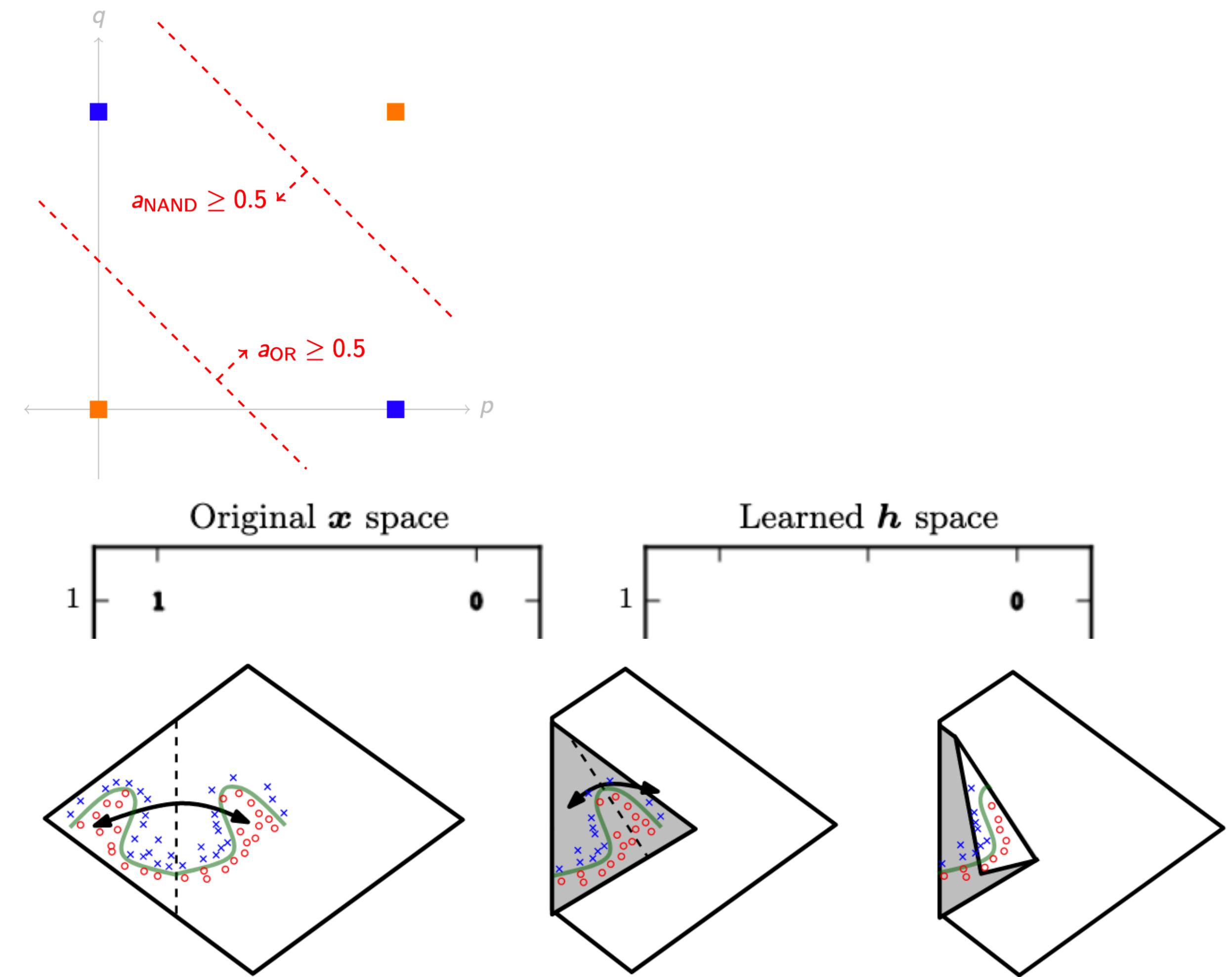
# Non-linearity, cont.

- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions
- One perspective: integrating extracted features
- An equivalent perspective:
  - Transforming the input space (source; p. 169)
  - This is a *non-linear* transformation
  - Space folding intuition more generally



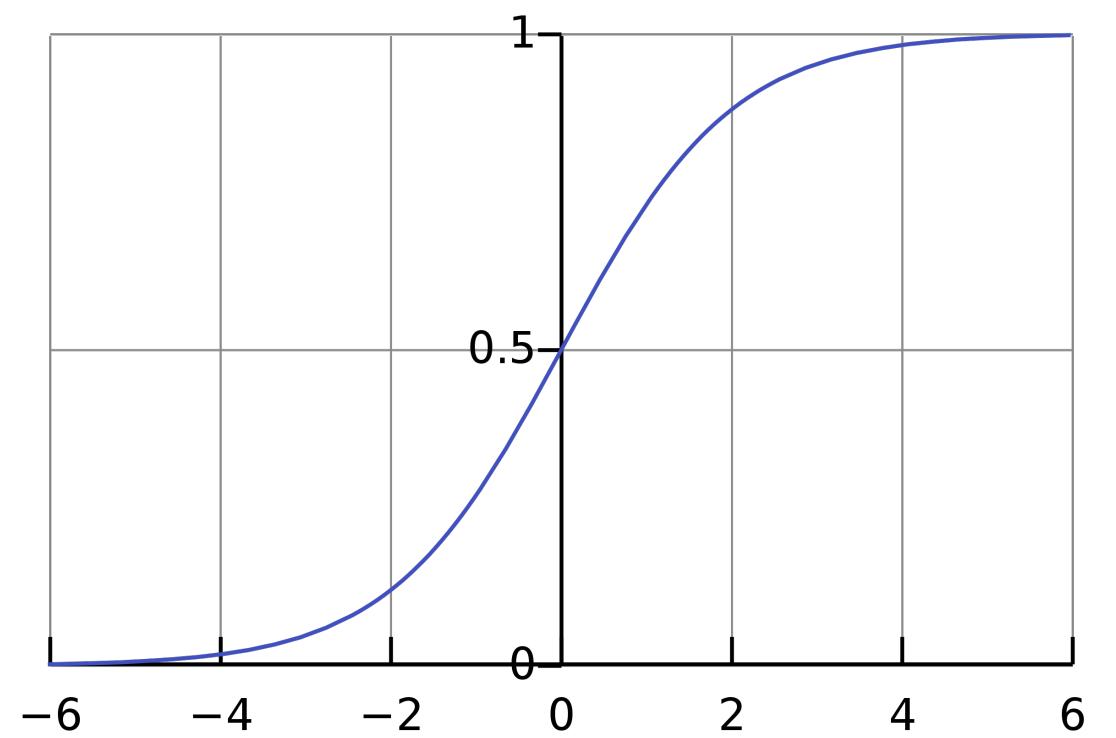
# Non-linearity, cont.

- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions
- One perspective: integrating extracted features
- An equivalent perspective:
  - Transforming the input space (source; p. 169)
  - This is a *non-linear* transformation
  - Space folding intuition more generally



# Activation Functions: Hidden Layer

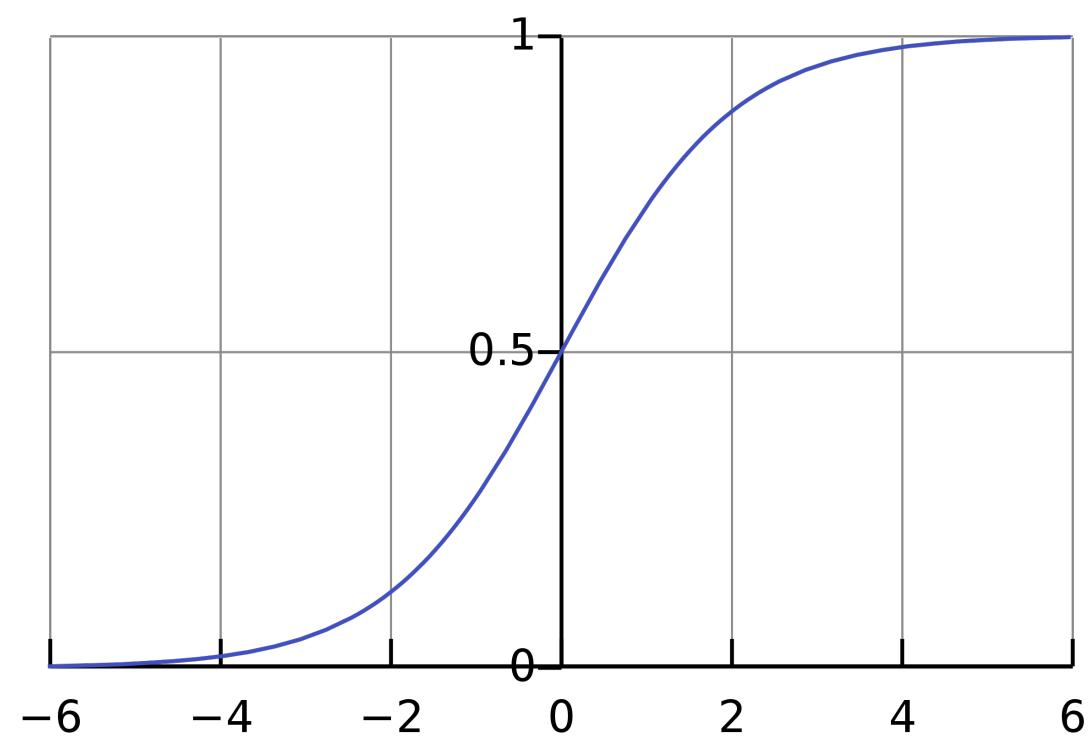
sigmoid



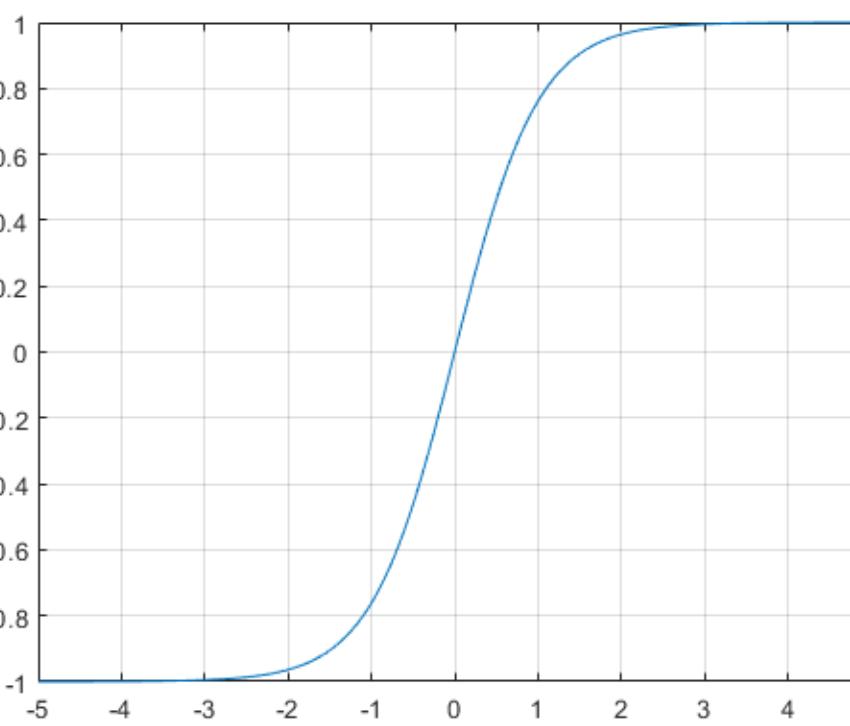
$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

# Activation Functions: Hidden Layer

sigmoid



tanh

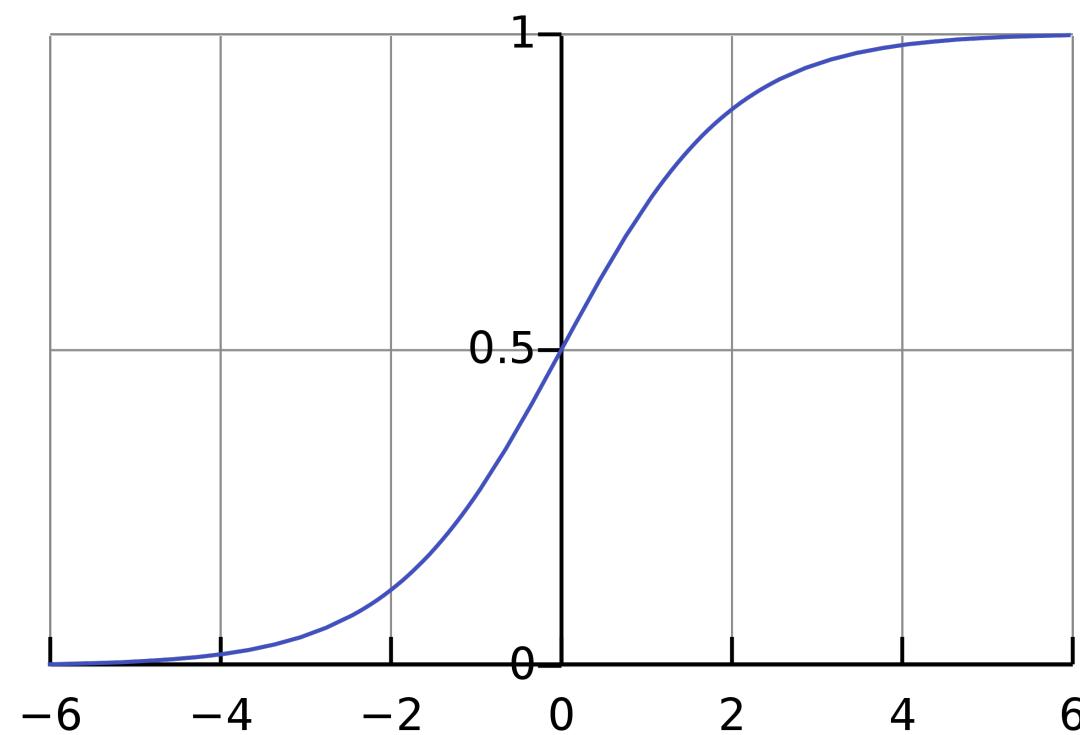


$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

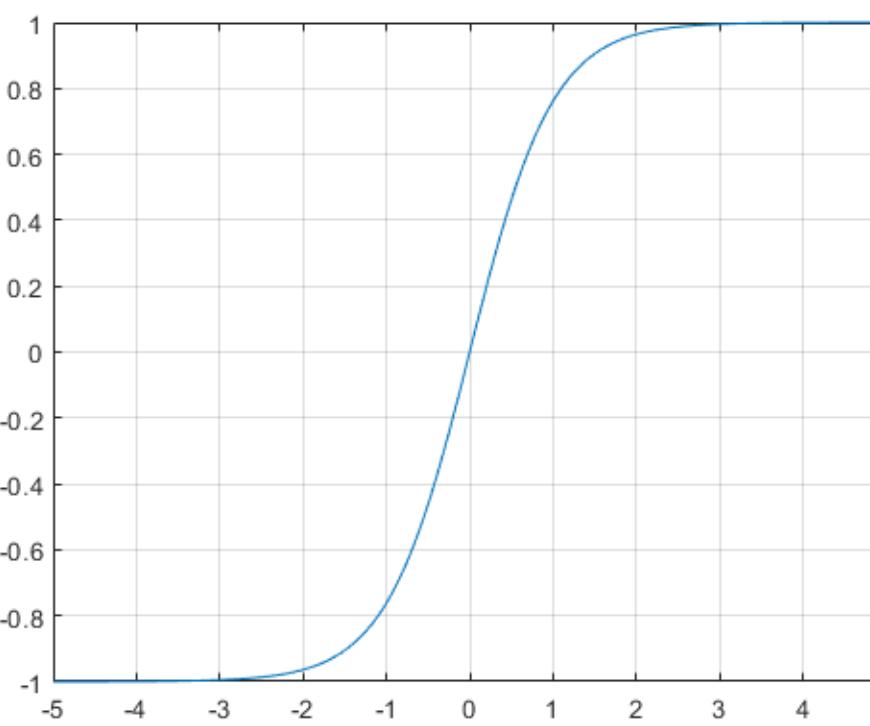
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

# Activation Functions: Hidden Layer

sigmoid



tanh



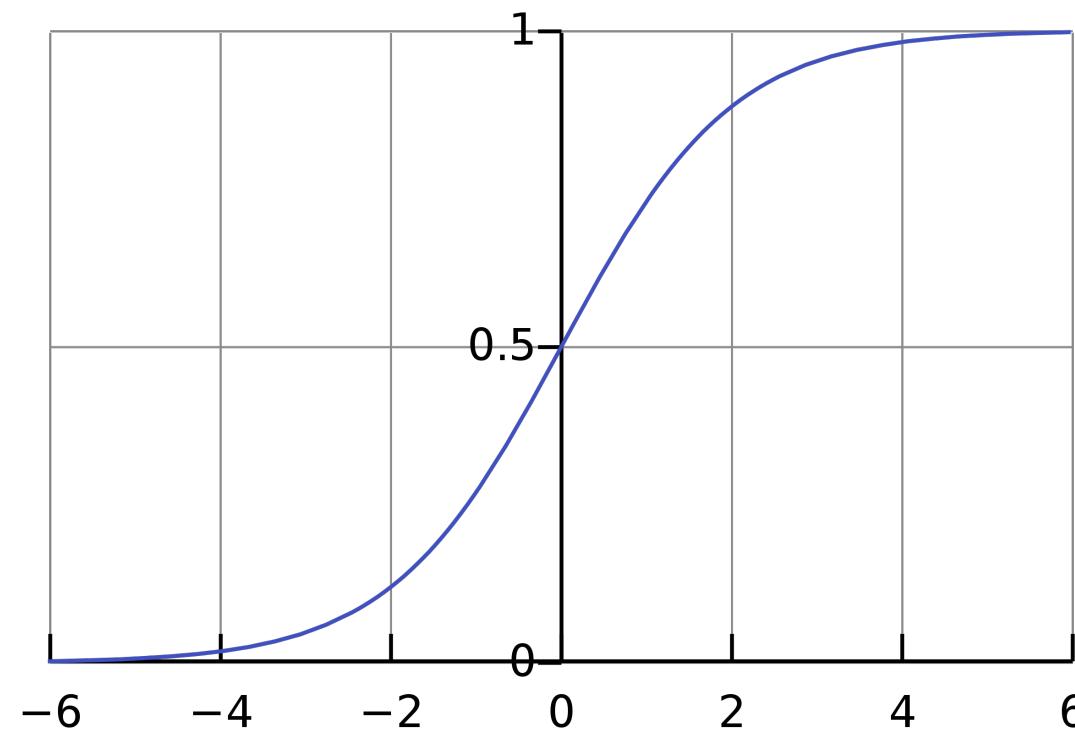
$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

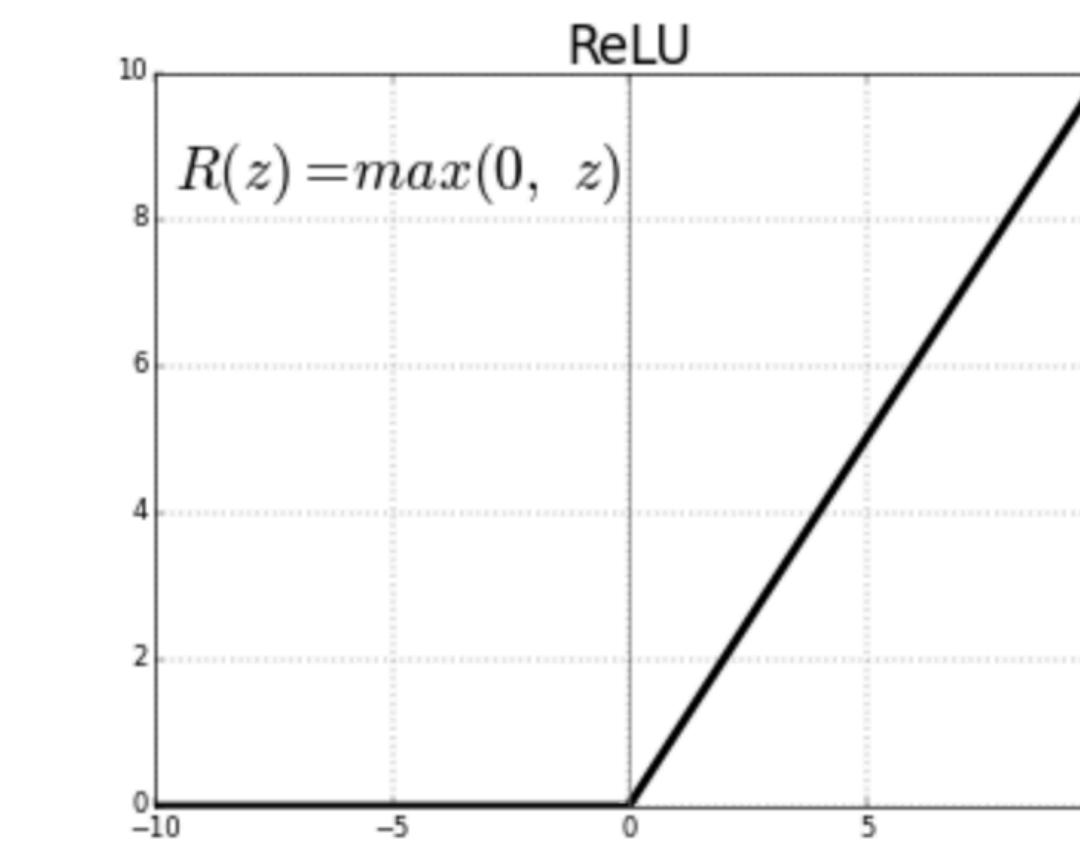
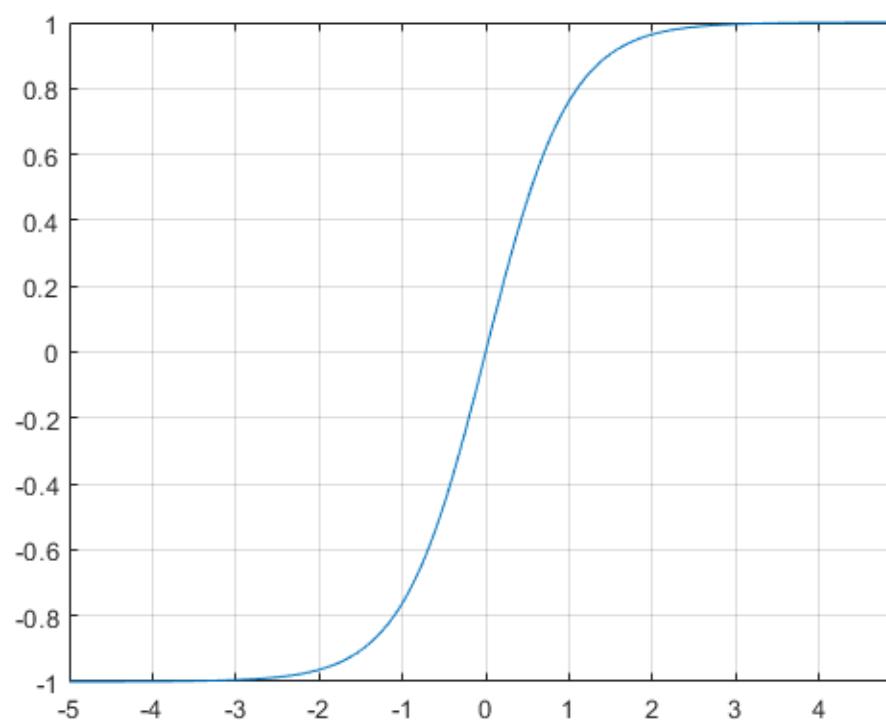
Problem: derivative “saturates” (nearly 0)  
everywhere except near origin

# Activation Functions: Hidden Layer

sigmoid



tanh



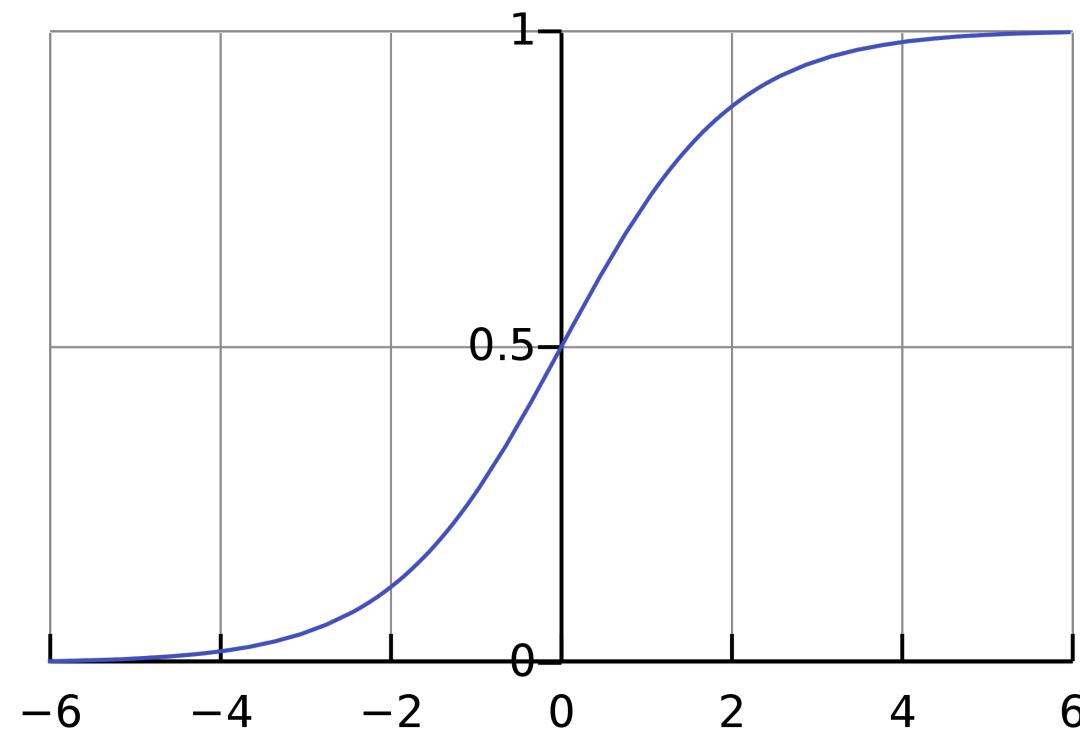
$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

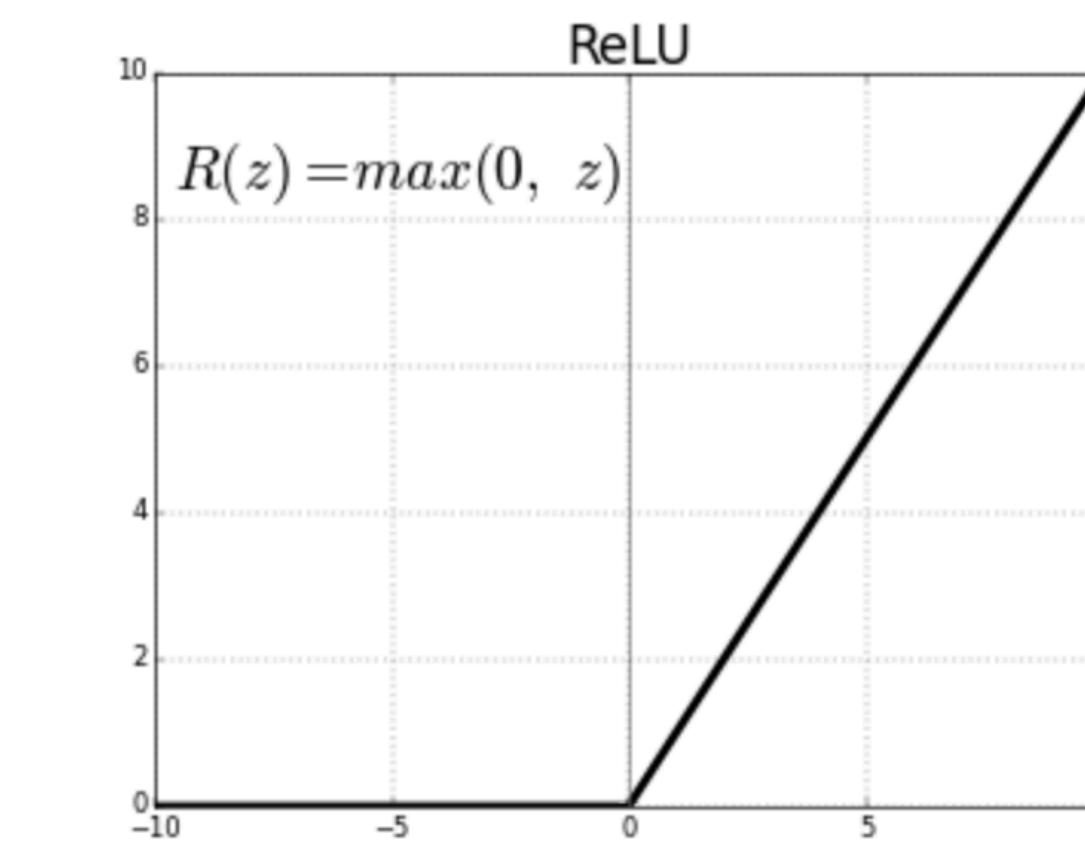
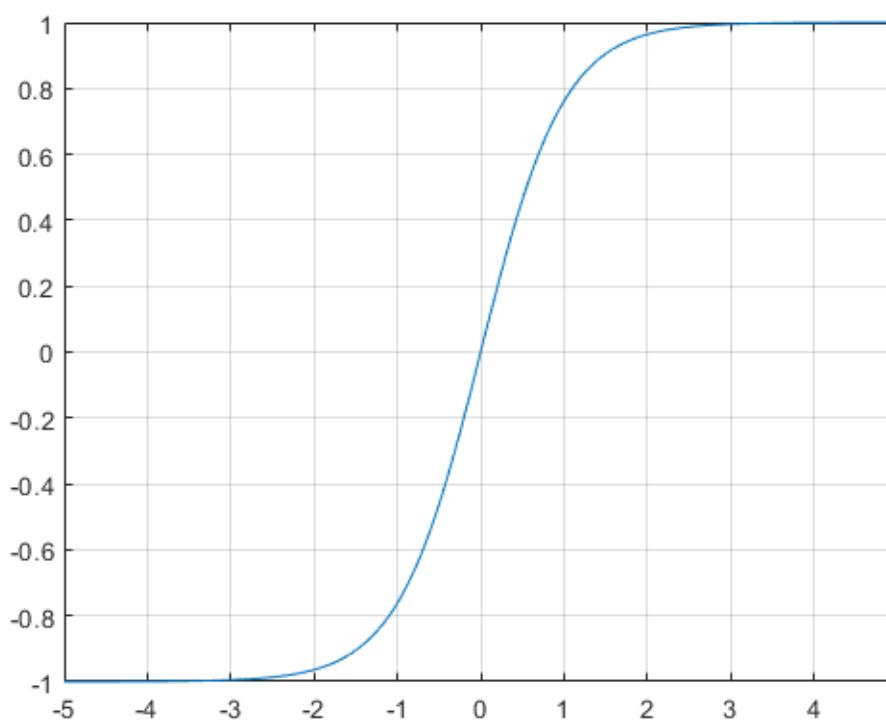
Problem: derivative “saturates” (nearly 0)  
everywhere except near origin

# Activation Functions: Hidden Layer

sigmoid



tanh



$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

- Use ReLU by default
- Generalizations:
  - Leaky
  - ELU
  - Softplus
  - ...

Problem: derivative “saturates” (nearly 0)  
everywhere except near origin

# Activation Functions: Output Layer

- Depends on the task!
- Regression (continuous output(s)): none!
  - Just use final linear transformation
- Binary classification: sigmoid
  - Also for *multi-label* classification
- Multi-class classification: softmax
  - Terminology: the inputs to a softmax are called *logits*
  - [there are sometimes other uses of the term, so beware]

$$\text{softmax}(x)_i = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

# Mini-batch computation

# Computing with a Single Input

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( xW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

$$x = [x_0 \ x_1 \ \cdots \ x_{n_0}]$$

Shape:  $(1, n_0)$

$$b^1 = [b_0^1 \ b_1^1 \ \cdots \ b_{n_1}^1]$$

Shape:  $(1, n_1)$

$$W^1 = \begin{bmatrix} w_{00}^1 & w_{10}^1 & \cdots & w_{0n_1}^1 \\ w_{10}^1 & w_{11}^1 & \cdots & w_{1n_1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_00}^1 & w_{n_01}^1 & \cdots & w_{n_0n_1}^1 \end{bmatrix}$$

Shape:  $(n_0, n_1)$

$n_0$ : number of neurons in layer 0 (input)

$n_1$ : number of neurons in layer 1

# Mini-batch Gradient Descent (from lecture 2)

initialize parameters / build model

for each epoch:

```
data = shuffle(data)
batches = make_batches(data)
```

for each batch in batches:

```
outputs = model(batch)
loss = loss_fn(outputs, true_outputs)
compute gradients
update parameters
```

# Computing with Mini-batches

- Bad idea:

```
for each batch in batches:  
    for each datum in batch:  
        outputs = model(datum)  
        loss = loss_fn(outputs, true_outputs)  
        compute gradients  
        update parameters
```

# Computing with a Batch of Inputs

# Computing with a Batch of Inputs

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( XW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

# Computing with a Batch of Inputs

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( XW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

$$X = \begin{bmatrix} x_0^0 & x_1^0 & \dots & x_{n_0}^0 \\ x_1^0 & x_1^1 & \dots & x_{n_0}^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^n & x_1^n & \dots & x_{n_0}^n \end{bmatrix}$$

Shape:  $(n, n_0)$

$n$ : batch\_size

# Computing with a Batch of Inputs

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( XW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

$$X = \begin{bmatrix} x_0^0 & x_1^0 & \dots & x_{n_0}^0 \\ x_1^0 & x_1^1 & \dots & x_{n_0}^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^n & x_1^n & \dots & x_{n_0}^n \end{bmatrix} \quad W^1 = \begin{bmatrix} w_{00}^1 & w_{01}^1 & \cdots & w_{0n_1}^1 \\ w_{10}^1 & w_{11}^1 & \cdots & w_{1n_1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_00}^1 & w_{n_01}^1 & \cdots & w_{n_0n_1}^1 \end{bmatrix}$$

Shape:  $(n, n_0)$

$n$ : batch\_size

# Computing with a Batch of Inputs

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( XW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

$$X = \begin{bmatrix} x_0^0 & x_1^0 & \dots & x_{n_0}^0 \\ x_1^0 & x_1^1 & \dots & x_{n_0}^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^n & x_1^n & \dots & x_{n_0}^n \end{bmatrix} \quad W^1 = \begin{bmatrix} w_{00}^1 & w_{01}^1 & \cdots & w_{0n_1}^1 \\ w_{10}^1 & w_{11}^1 & \cdots & w_{1n_1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_00}^1 & w_{n_01}^1 & \cdots & w_{n_0n_1}^1 \end{bmatrix}$$

Shape:  $(n, n_0)$

$n$ : batch\_size

Shape:  $(n_0, n_1)$

$n_0$ : number of neurons in layer 0 (input)

$n_1$ : number of neurons in layer 1

# Computing with a Batch of Inputs

$$\hat{y} = f_n \left( f_{n-1} \left( \cdots f_2 \left( f_1 \left( XW^1 + b^1 \right) W^2 + b^2 \right) \cdots \right) W^n + b^n \right)$$

$$X = \begin{bmatrix} x_0^0 & x_1^0 & \dots & x_{n_0}^0 \\ x_1^0 & x_1^1 & \dots & x_{n_0}^1 \\ \vdots & \vdots & \ddots & \vdots \\ x_1^n & x_1^n & \dots & x_{n_0}^n \end{bmatrix}$$

Shape:  $(n, n_0)$   
 $n$ : batch\_size

$$W^1 = \begin{bmatrix} w_{00}^1 & w_{01}^1 & \cdots & w_{0n_1}^1 \\ w_{10}^1 & w_{11}^1 & \cdots & w_{1n_1}^1 \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_00}^1 & w_{n_01}^1 & \cdots & w_{n_0n_1}^1 \end{bmatrix}$$

Shape:  $(n_0, n_1)$   
 $n_0$ : number of neurons in layer 0 (input)  
 $n_1$ : number of neurons in layer 1

$$b^1 = \begin{bmatrix} b_0^1 & b_1^1 & \dots & b_{n_1}^1 \end{bmatrix}$$

Shape:  $(1, n_1)$   
Added to each row of  $XW^1$

# Note on mini-batches and shape

# Note on mini-batches and shape

- Most modern neural net libraries (e.g. PyTorch) expect the *first* dimension of matrices/tensors to be a batch size
  - Produce a sequence of representations, *for each item* in the batch
  - e.g. (batch\_size, input\_size) → (batch\_size, hidden\_size) → (batch\_size, output\_size)

# Note on mini-batches and shape

- Most modern neural net libraries (e.g. PyTorch) expect the *first* dimension of matrices/tensors to be a batch size
  - Produce a sequence of representations, *for each item* in the batch
  - e.g. (batch\_size, input\_size) → (batch\_size, hidden\_size) → (batch\_size, output\_size)
- In principle, can be higher than 2-dimensional
  - Images: (batch\_size, width, height, 3)
  - Sequences: (batch\_size, seq\_len, representation\_size)

# Note on mini-batches and shape

- Most modern neural net libraries (e.g. PyTorch) expect the *first* dimension of matrices/tensors to be a batch size
  - Produce a sequence of representations, *for each item* in the batch
  - e.g. (batch\_size, input\_size) → (batch\_size, hidden\_size) → (batch\_size, output\_size)
- In principle, can be higher than 2-dimensional
  - Images: (batch\_size, width, height, 3)
  - Sequences: (batch\_size, seq\_len, representation\_size)
- Two comments:
  - In your code, **annotate every tensor** with a comment saying intended shape
  - When debugging, look at shapes early on!!

# Homework 2

# Next Time

- Further abstraction: *computation graph*
- Backpropagation algorithm for computing gradients
  - Using forward/backward API for nodes in a comp graph