Neural Network Introduction

LING 575K Deep Learning for NLP

Shane Steinert-Threlkeld

April 5 2021

Announcements

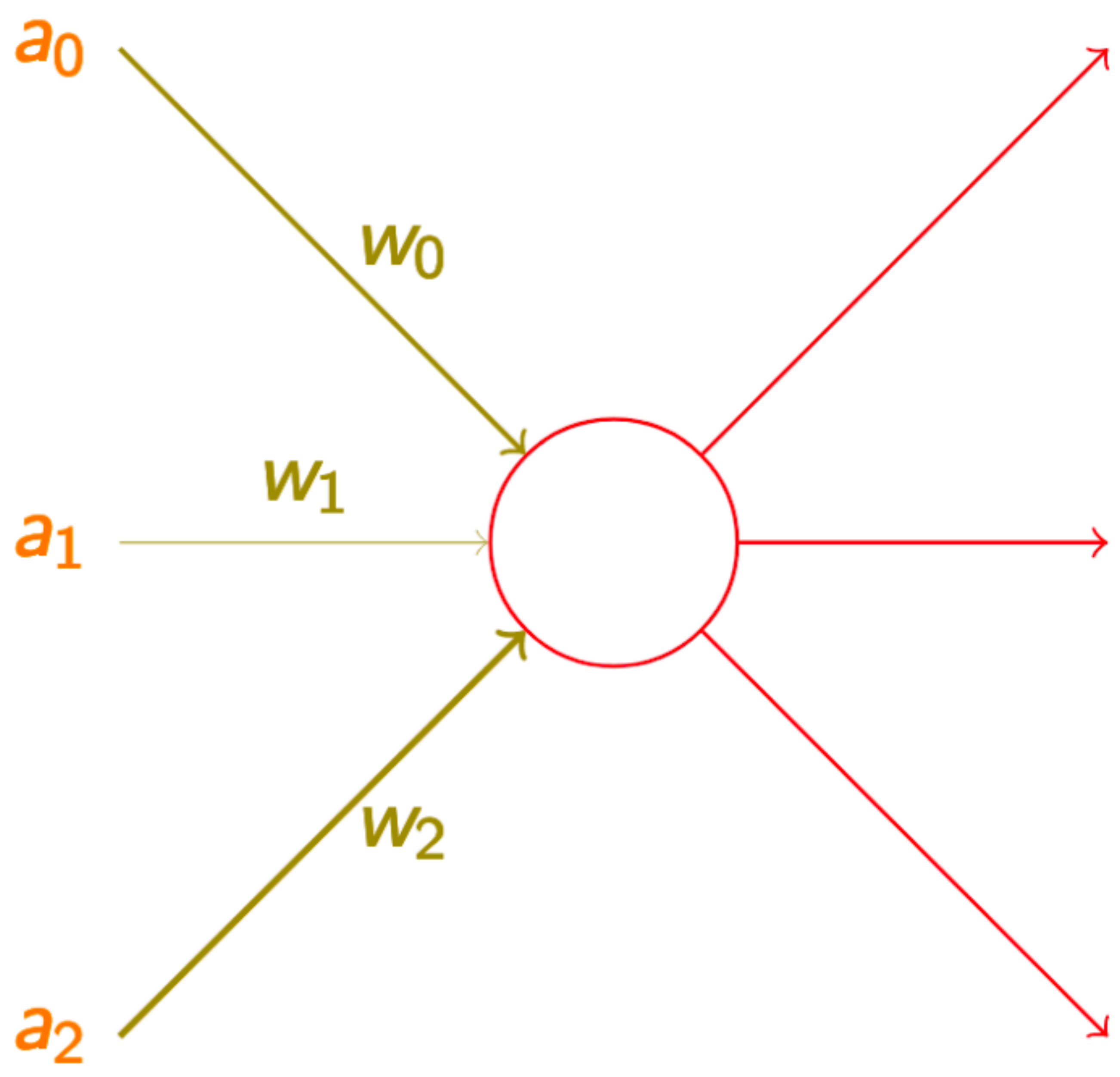
- HW1 due tomorrow night, upload readme and hw1.tar.gz to Canvas
  - NB: separate files!
  - Do not put readme inside of tar.gz
- indices_to_tokens: no error handling
- You can/should use `Vocabulary.from_text_files` to build your vocab object
  - Factory design pattern allows for different initialization 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

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

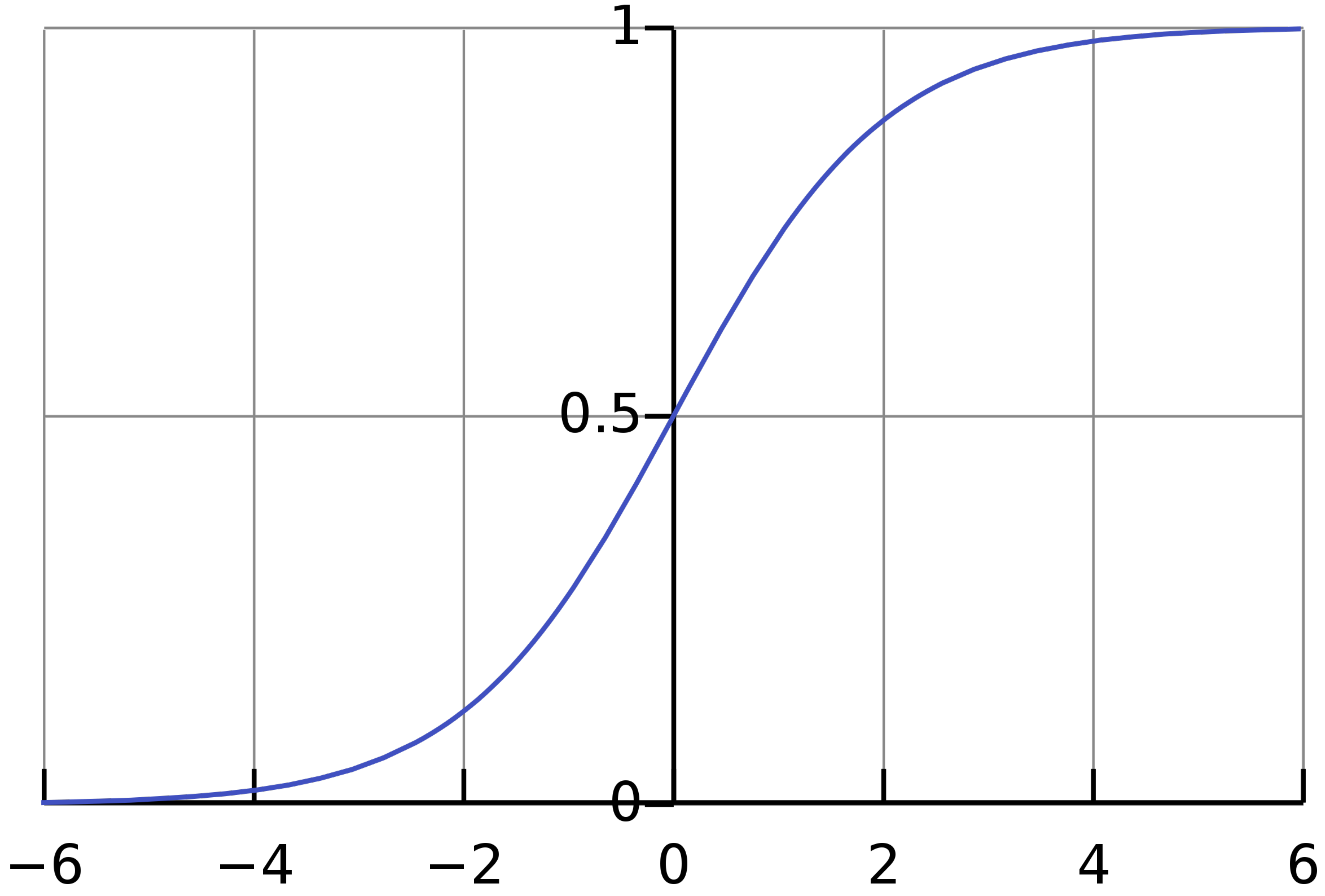
Artificial Neuron



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

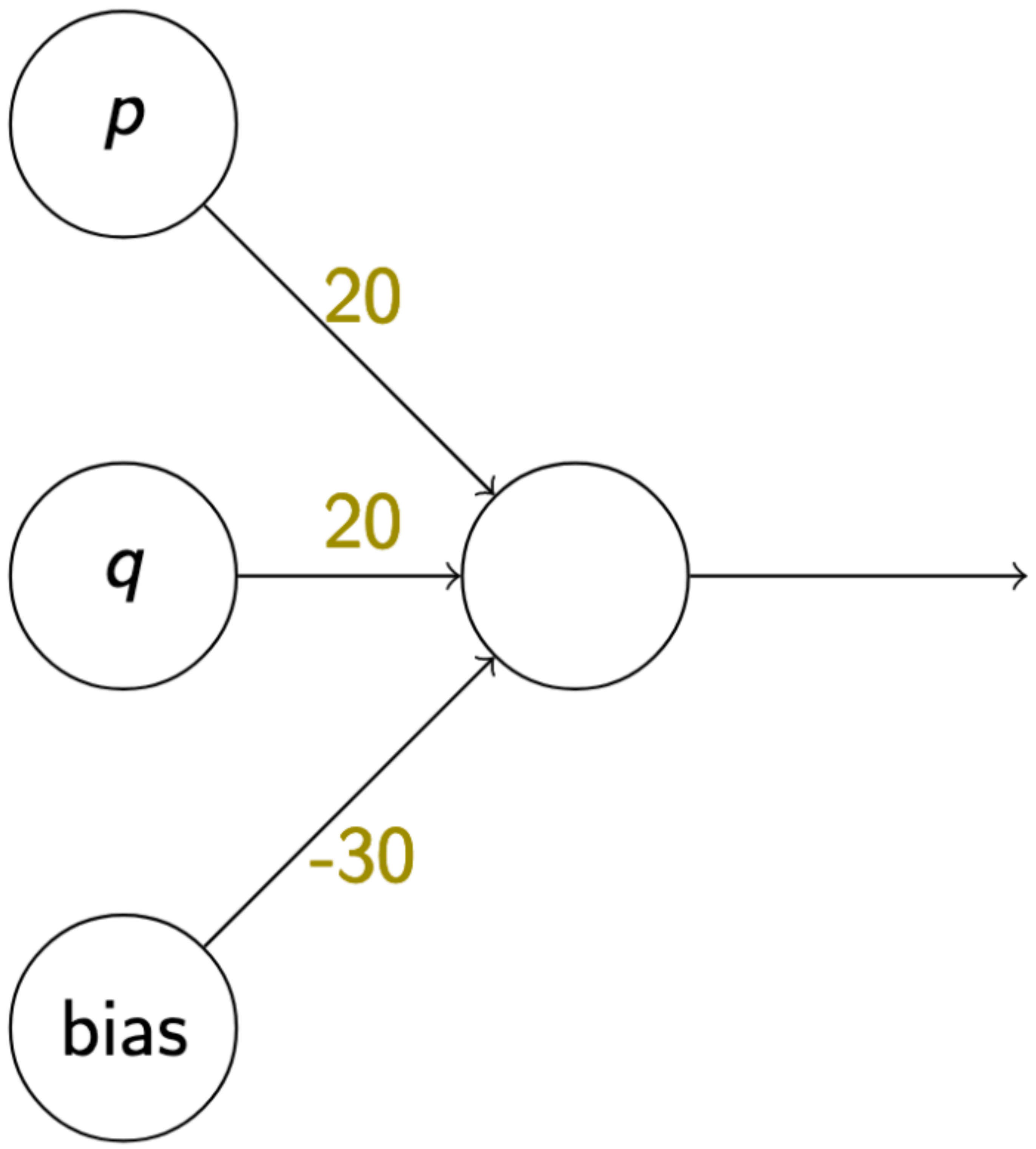
https://github.com/shanest/nn-tutorial

Activation Function: Sigmoid



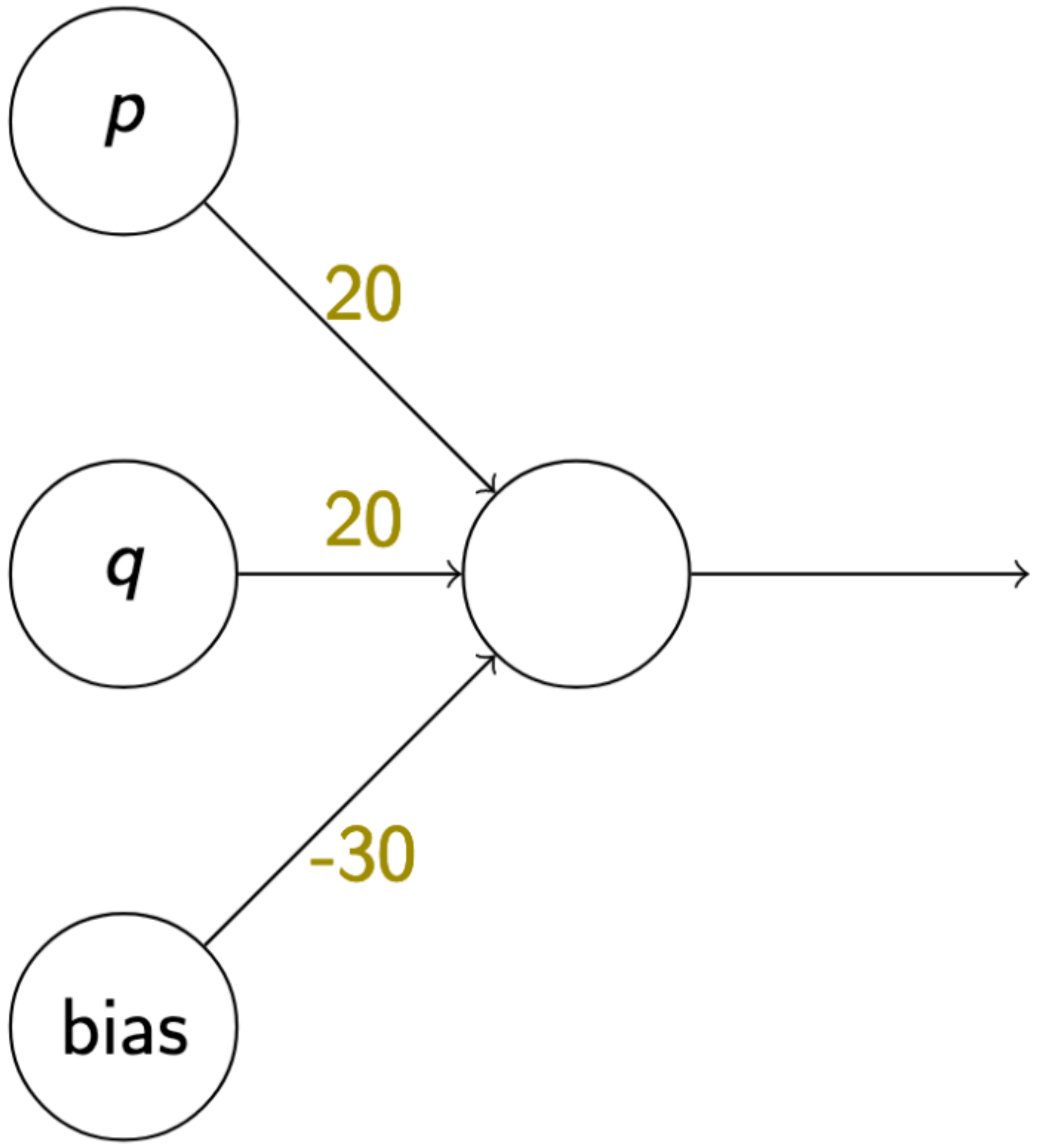
$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

Computing a Boolean function



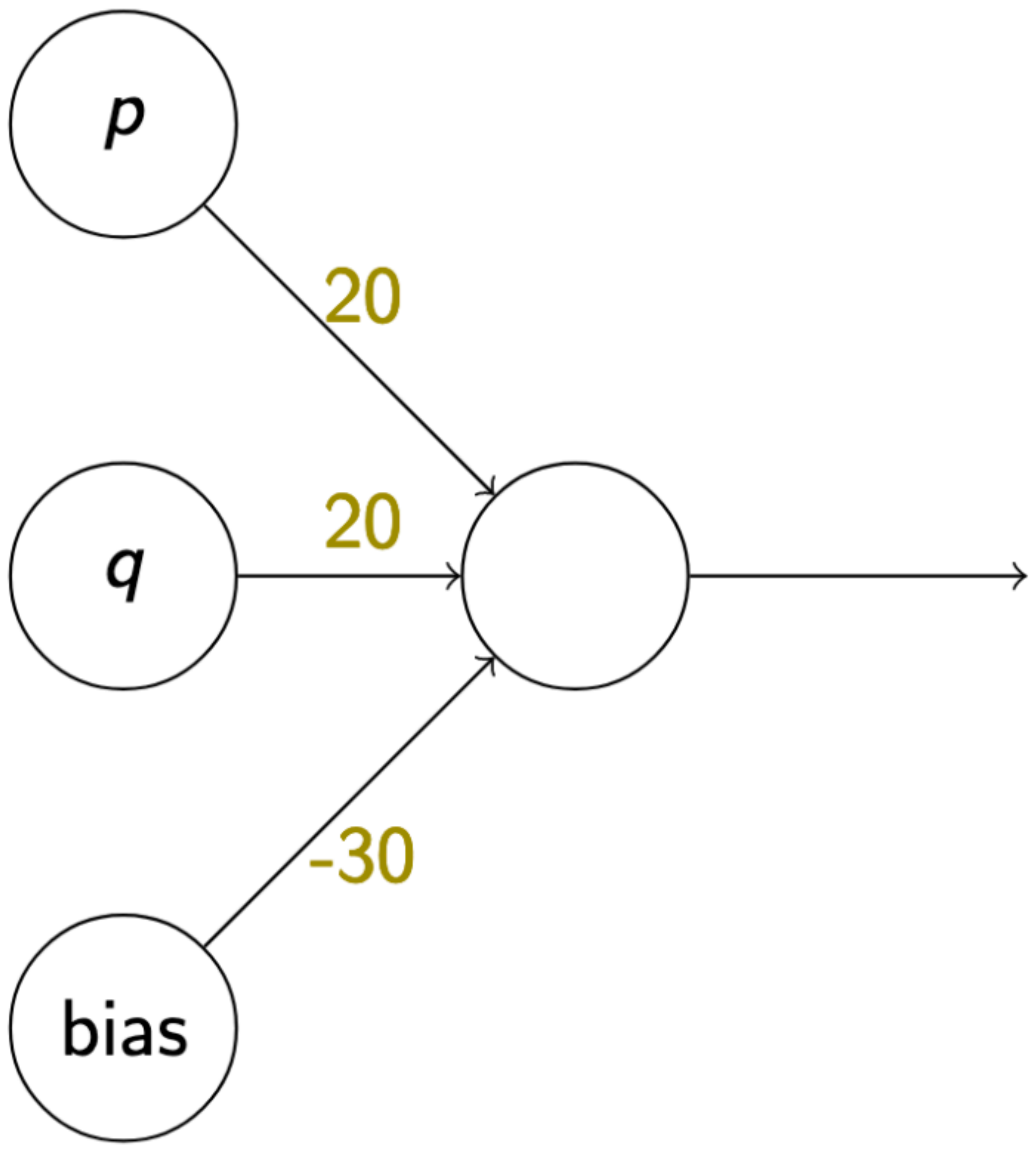
Computing a Boolean function

p	q	a



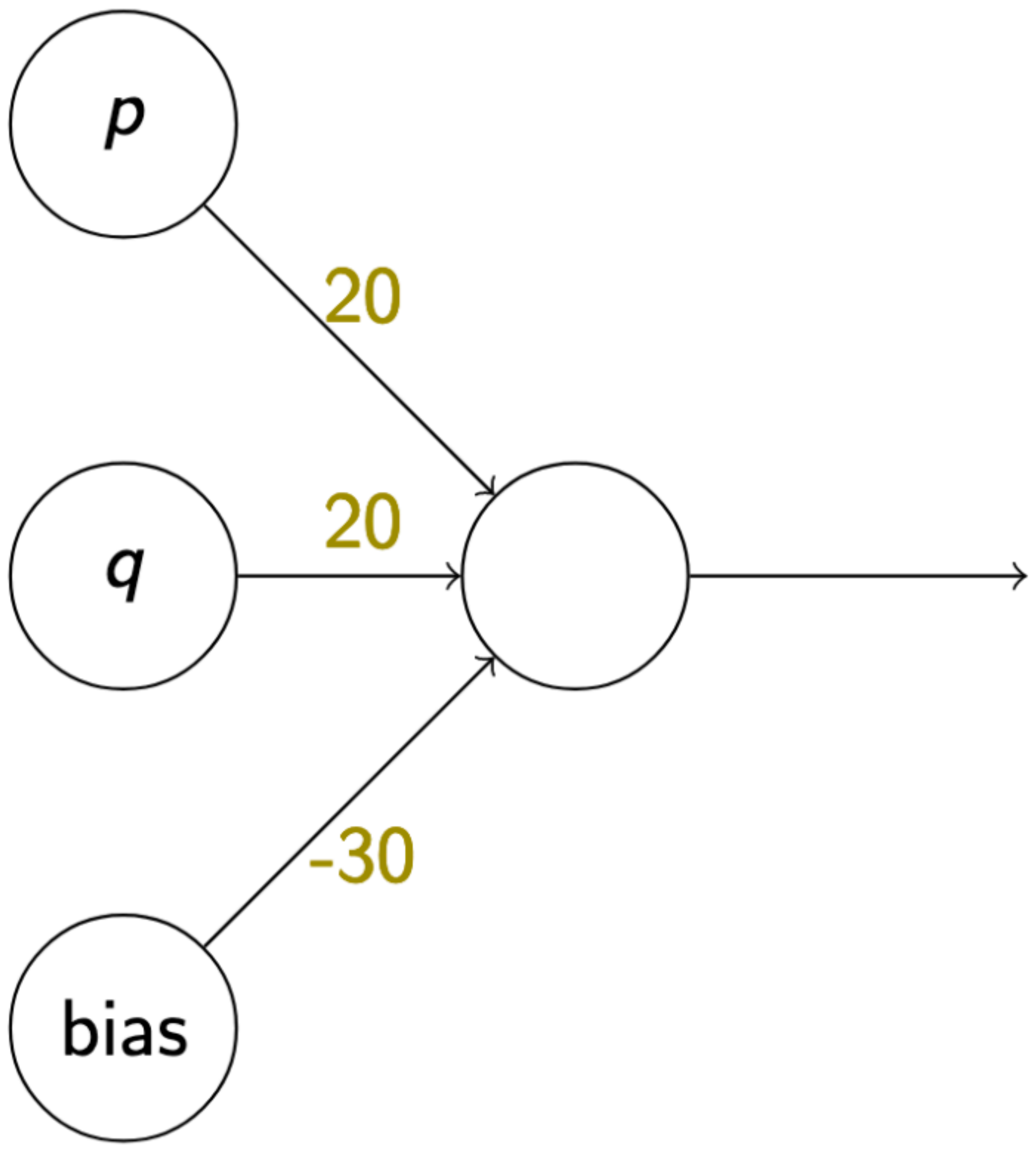
Computing a Boolean function

p	q	a
1	1	1



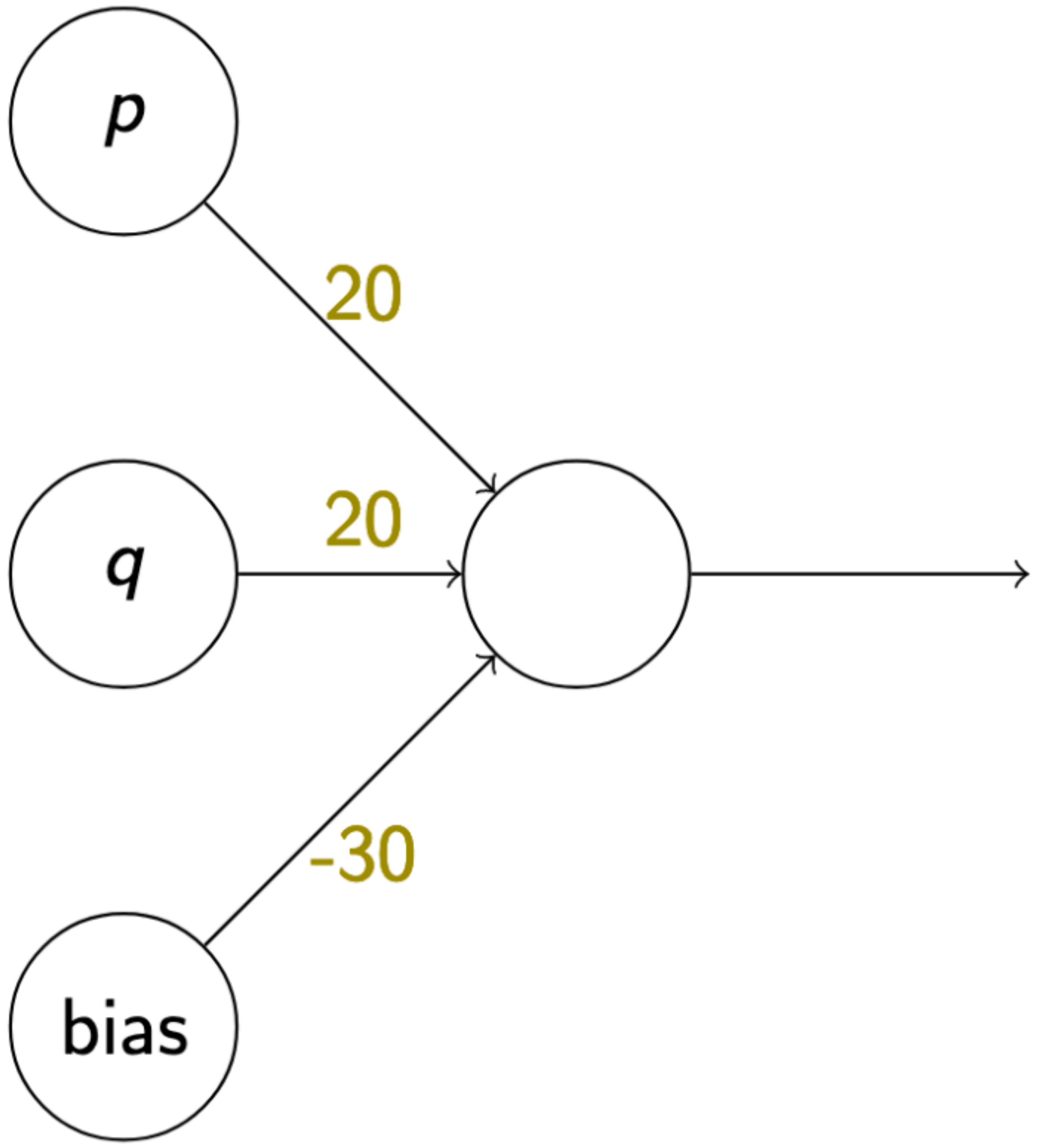
Computing a Boolean function

p	q	a
1	1	1
1	0	0



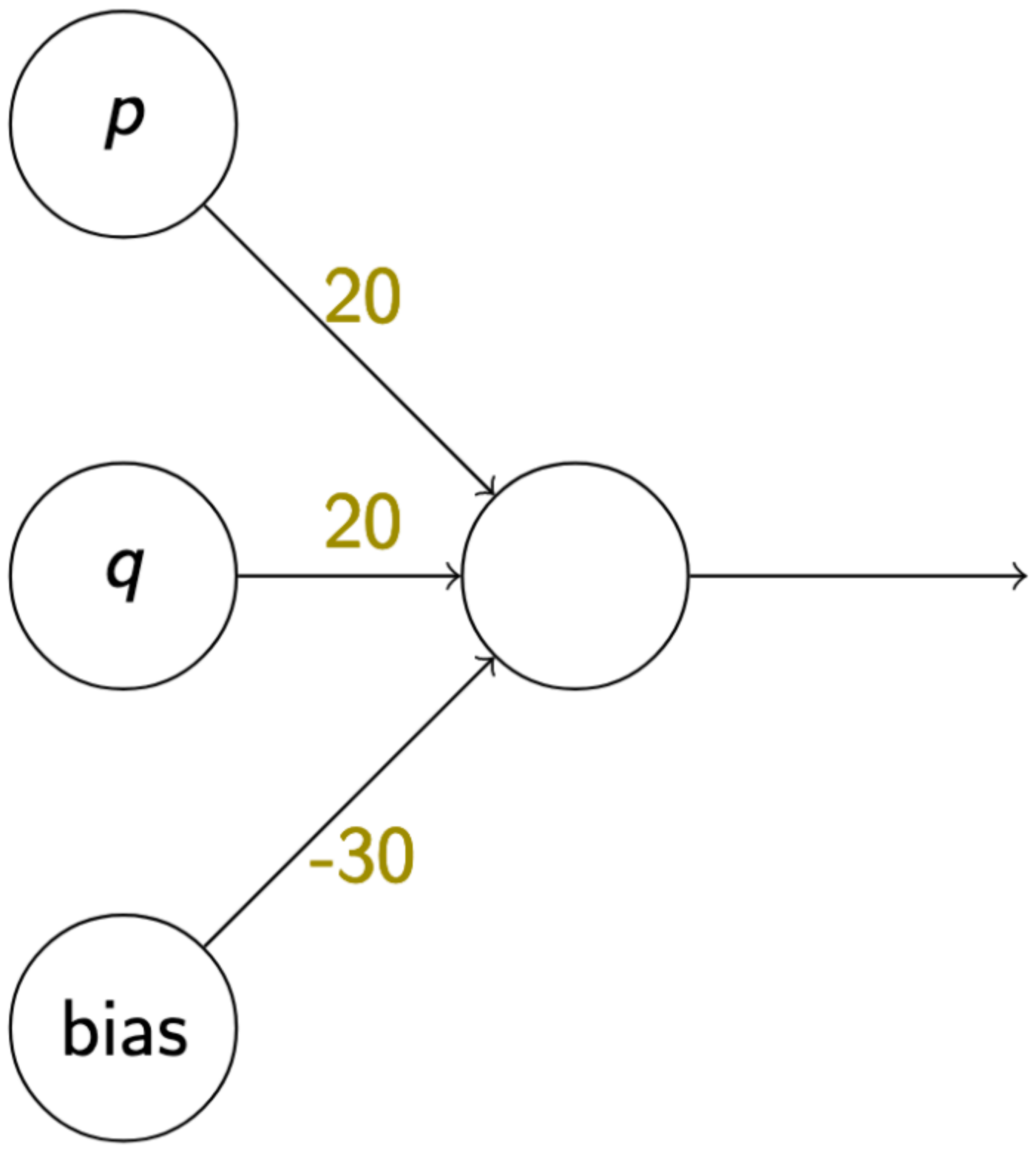
Computing a Boolean function

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

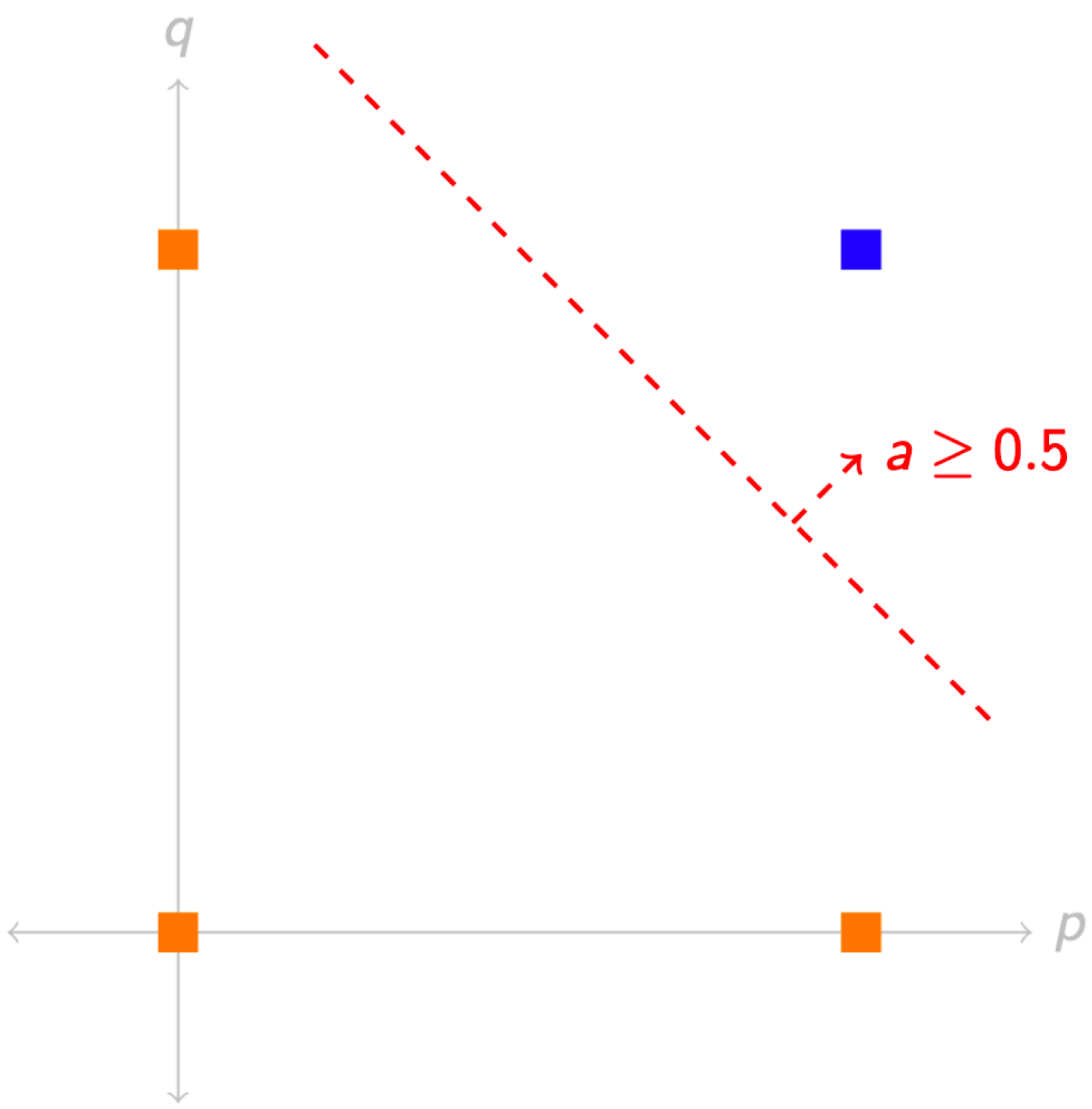


Computing a Boolean function

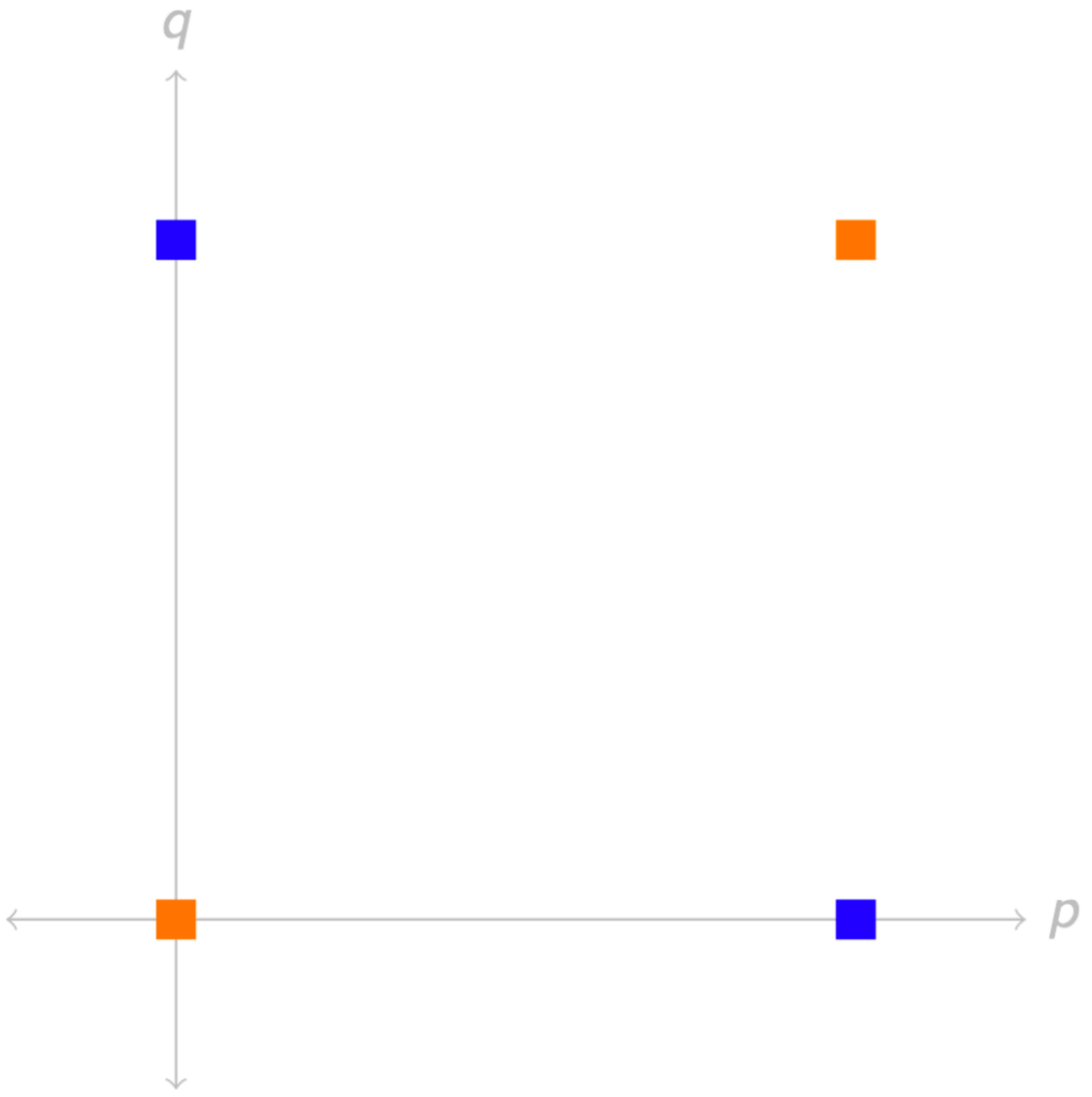
p	q	a
1	1	1
1	0	0
0	1	0
0	0	0



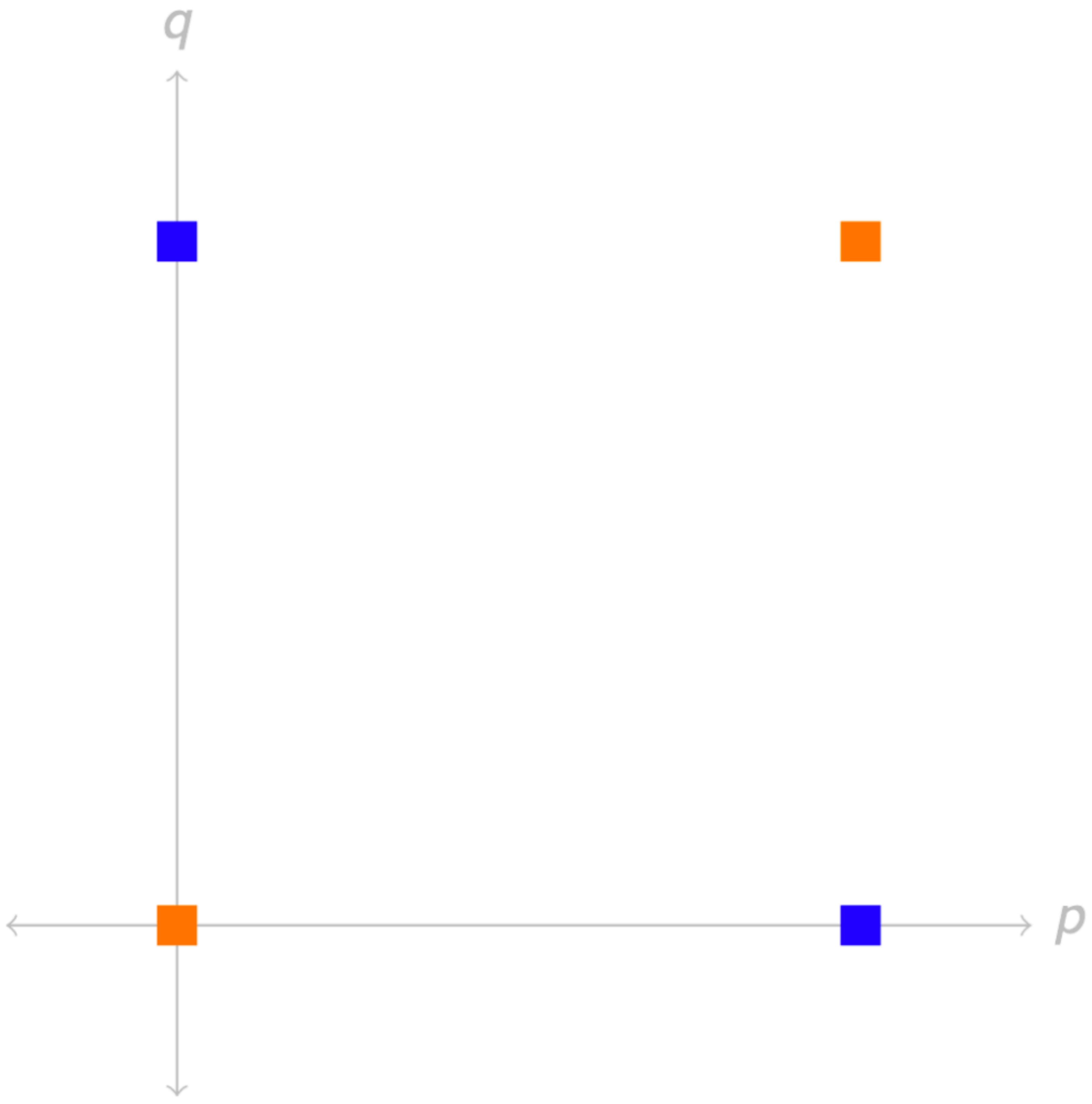
Computing 'and'



The XOR problem

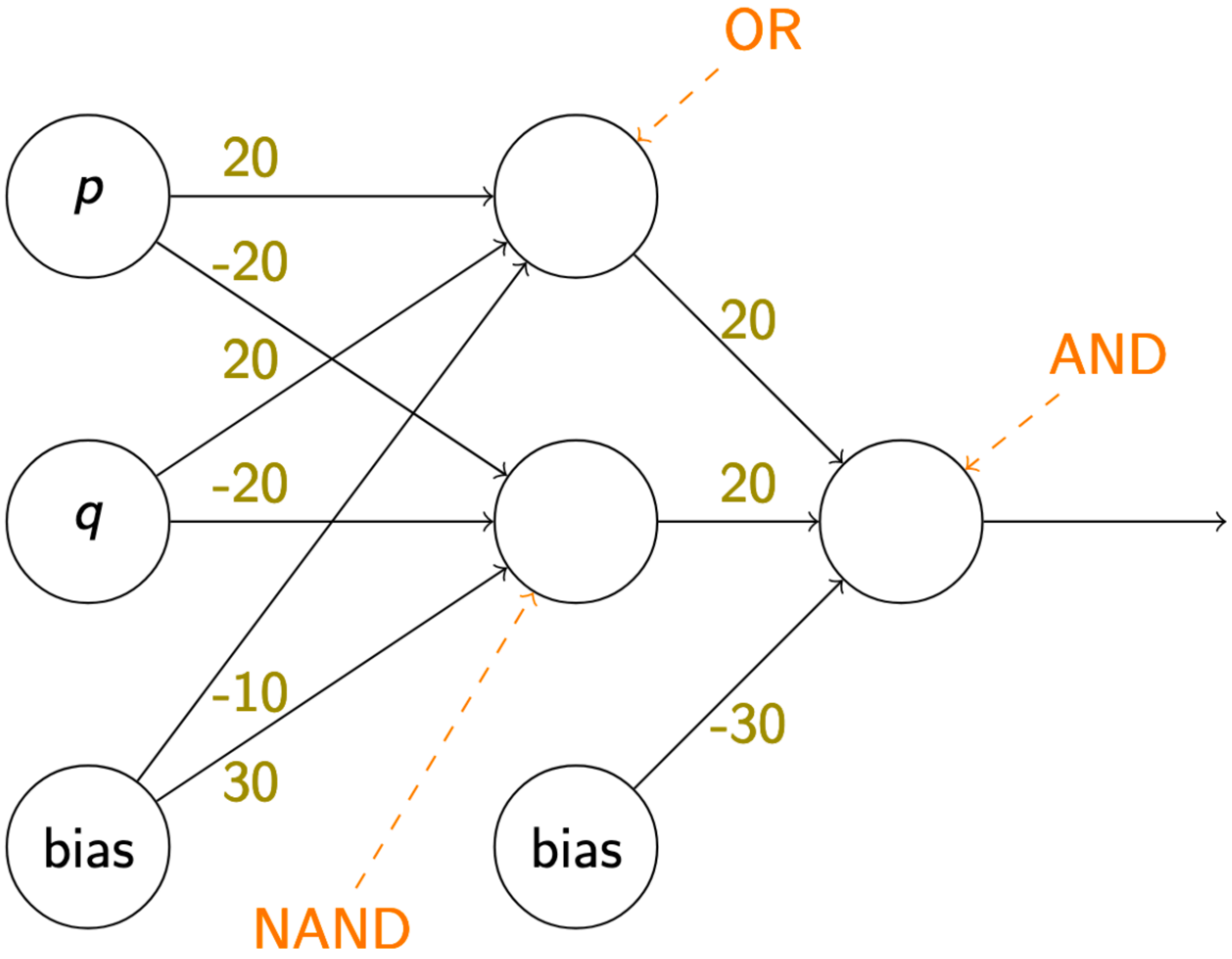


The XOR problem

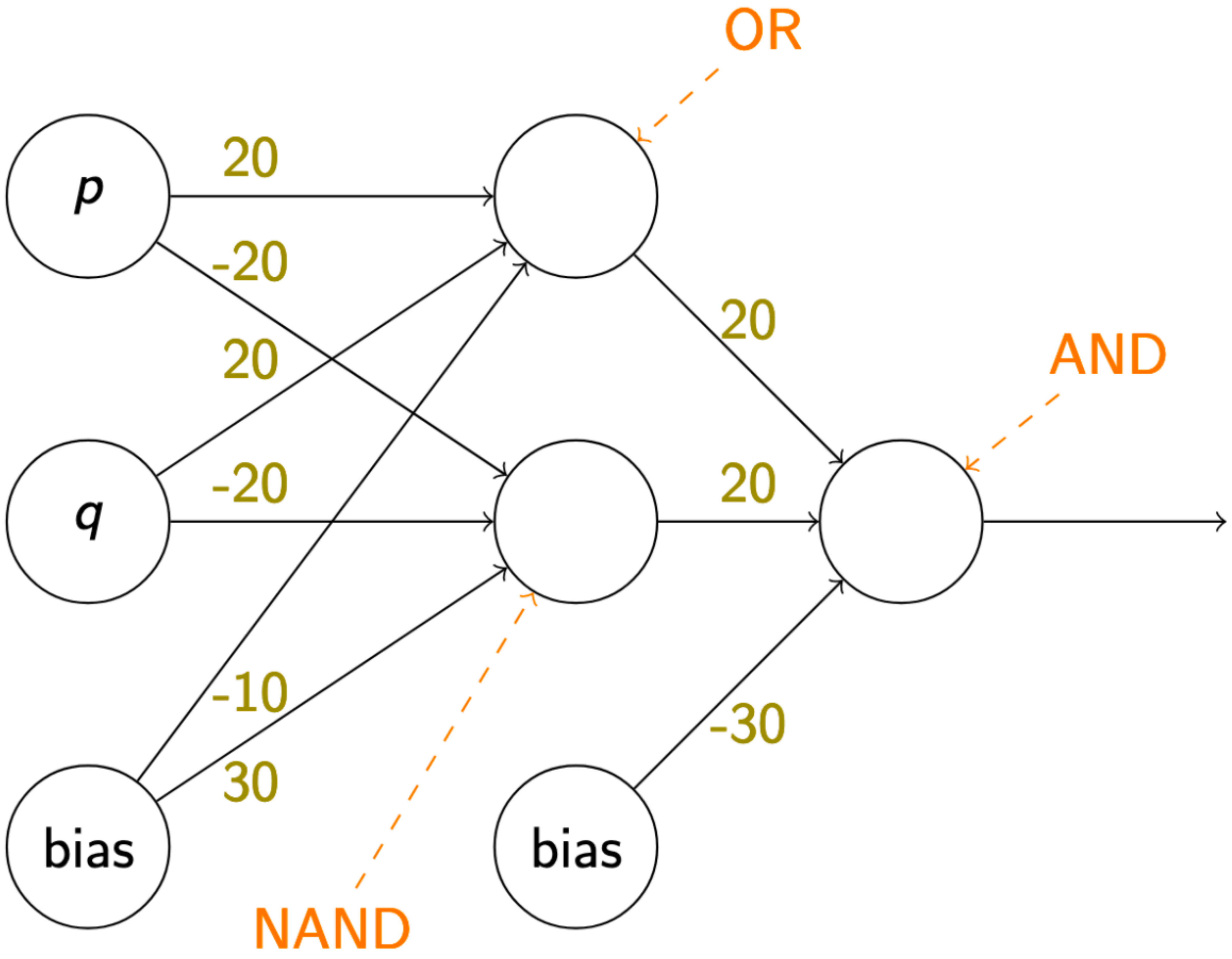


XOR is not linearly separable

Computing XOR

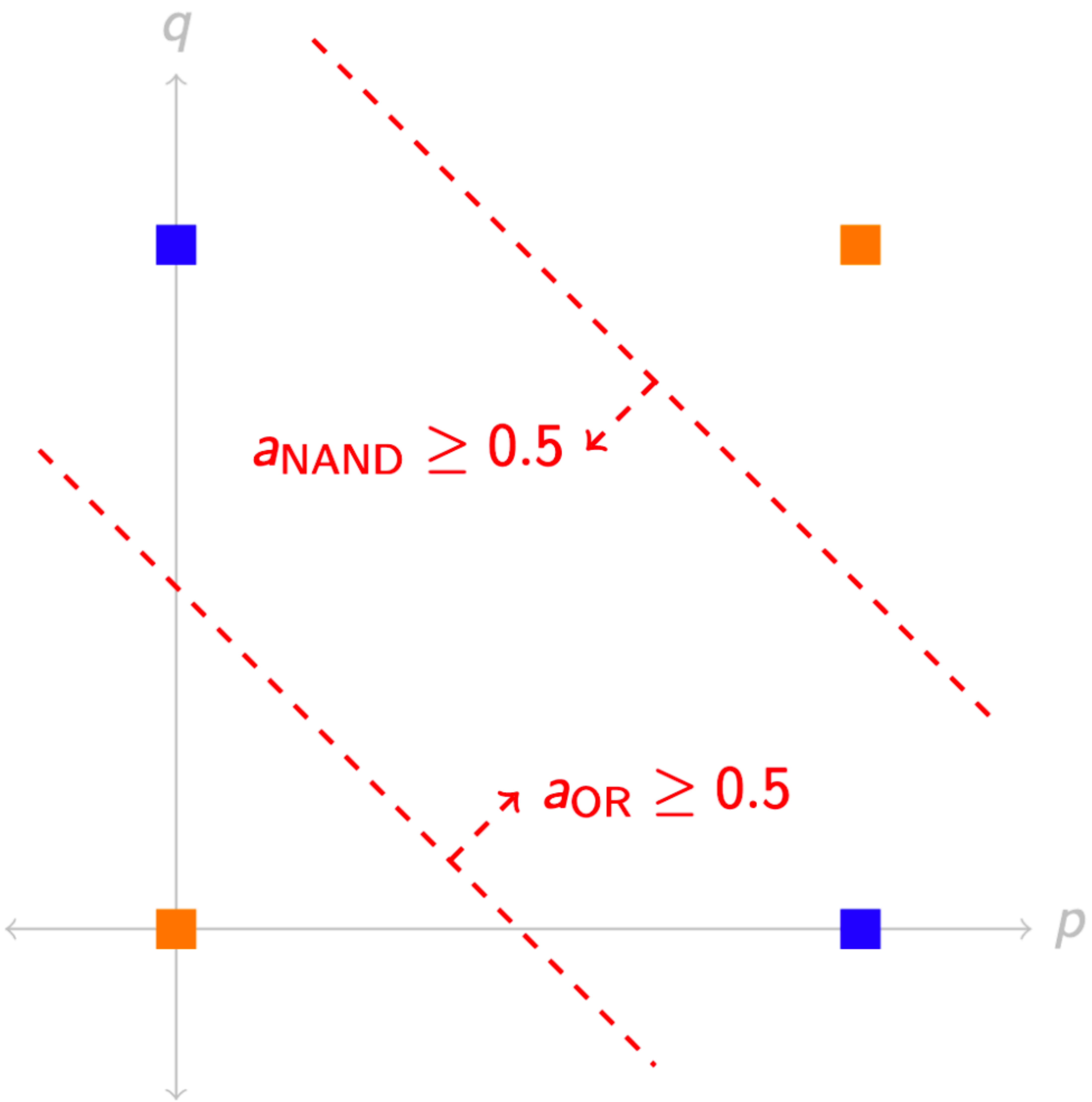


Computing XOR



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 $f : [0,1]^m \to \mathbb{R}$ 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 $f : [0,1]^m \to \mathbb{R}$ 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

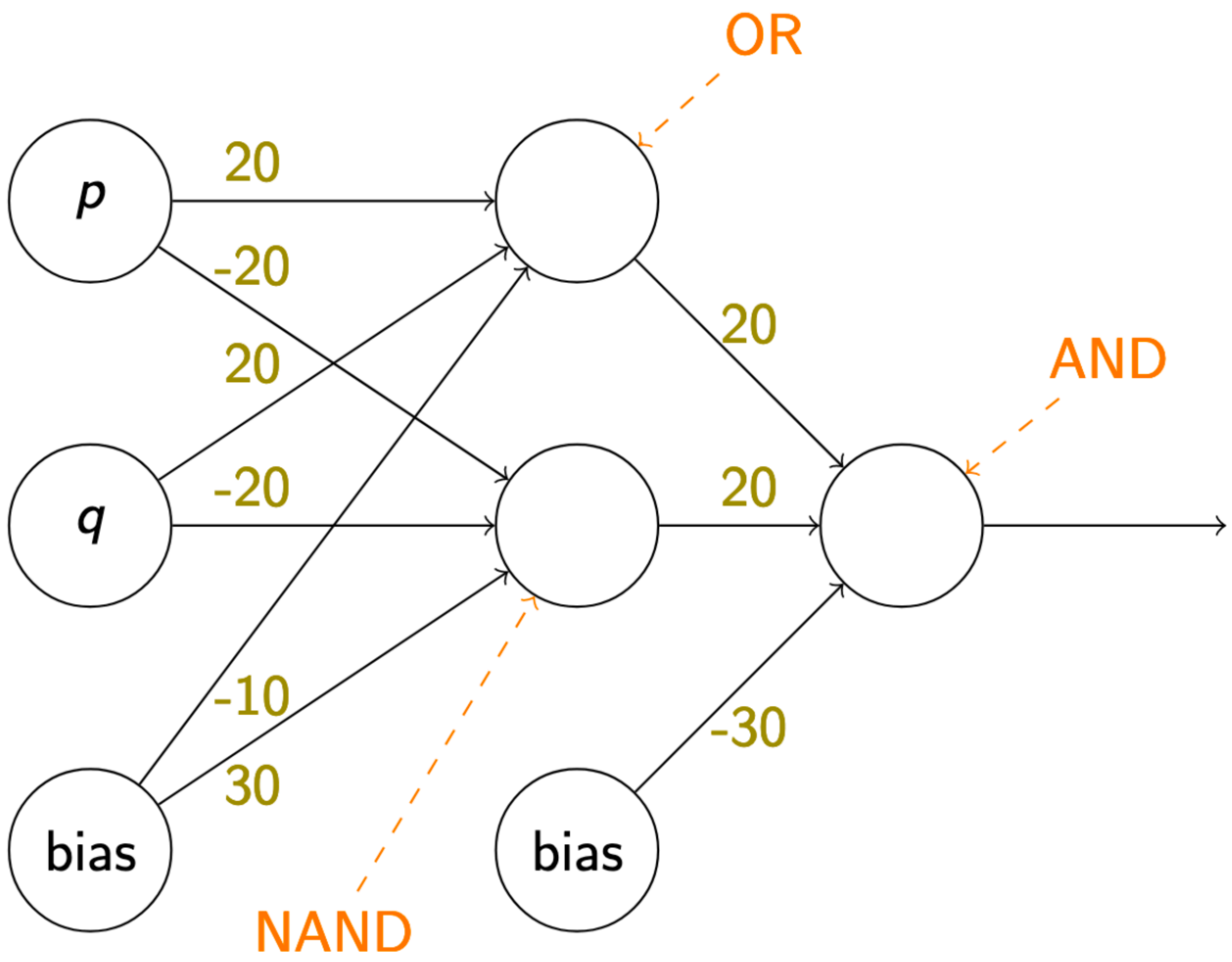
- Neural networks with *one* hidden layer are *universal function approximators*
- Let $f : [0,1]^m \to \mathbb{R}$ be continuous and $\epsilon > 0$. Then there is a one-hidden-layer neural network g with sigmoid activation such that $|f(\mathbf{x}) - g(\mathbf{x})| < \epsilon$ for all $\mathbf{x} \in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

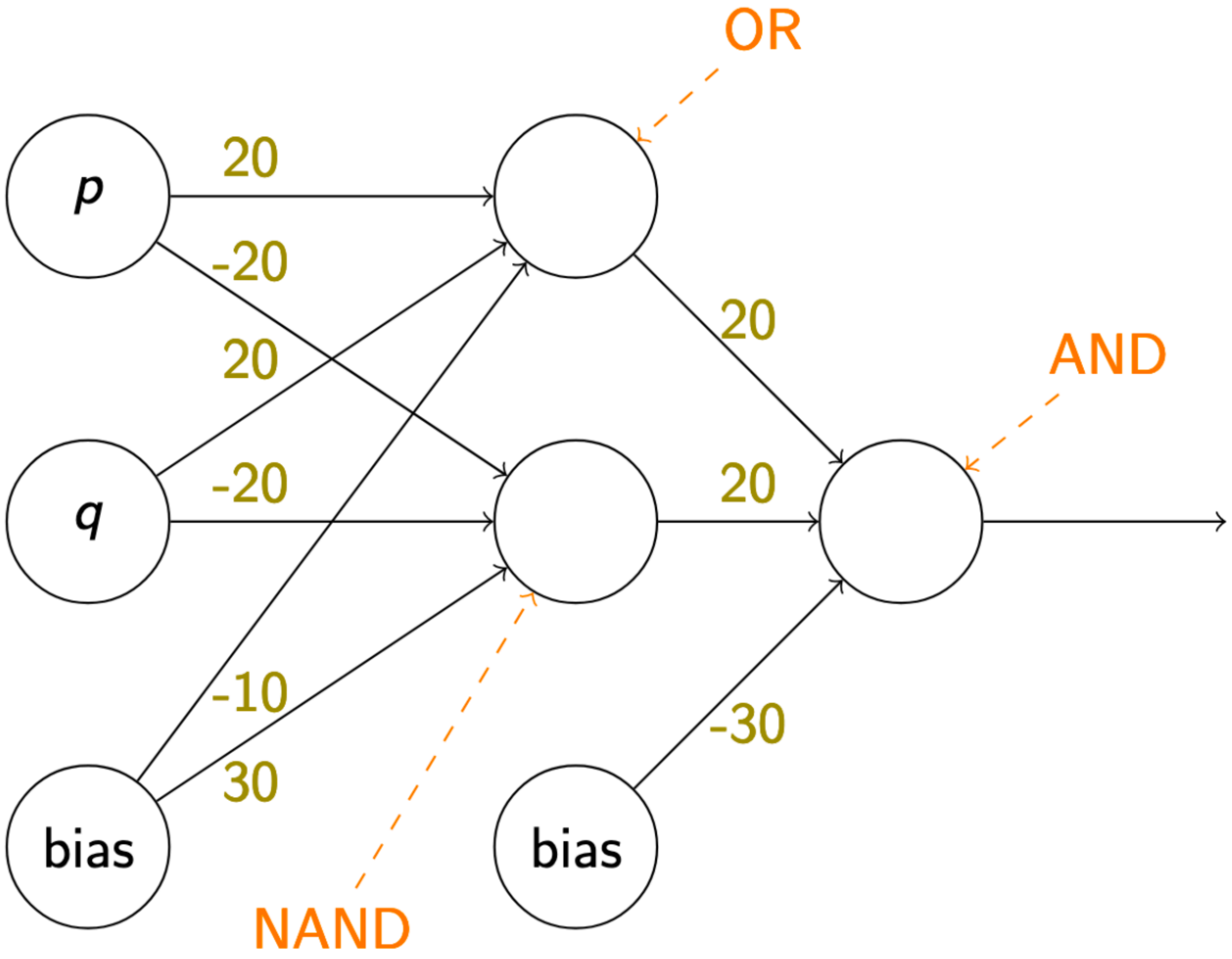
- Neural networks with *one* hidden layer are *universal function approximators*
- Let $f : [0,1]^m \to \mathbb{R}$ be continuous and $\epsilon > 0$. Then there is a one-hidden-layer neural network g with sigmoid activation such that $|f(\mathbf{x}) - g(\mathbf{x})| < \epsilon$ for all $\mathbf{x} \in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

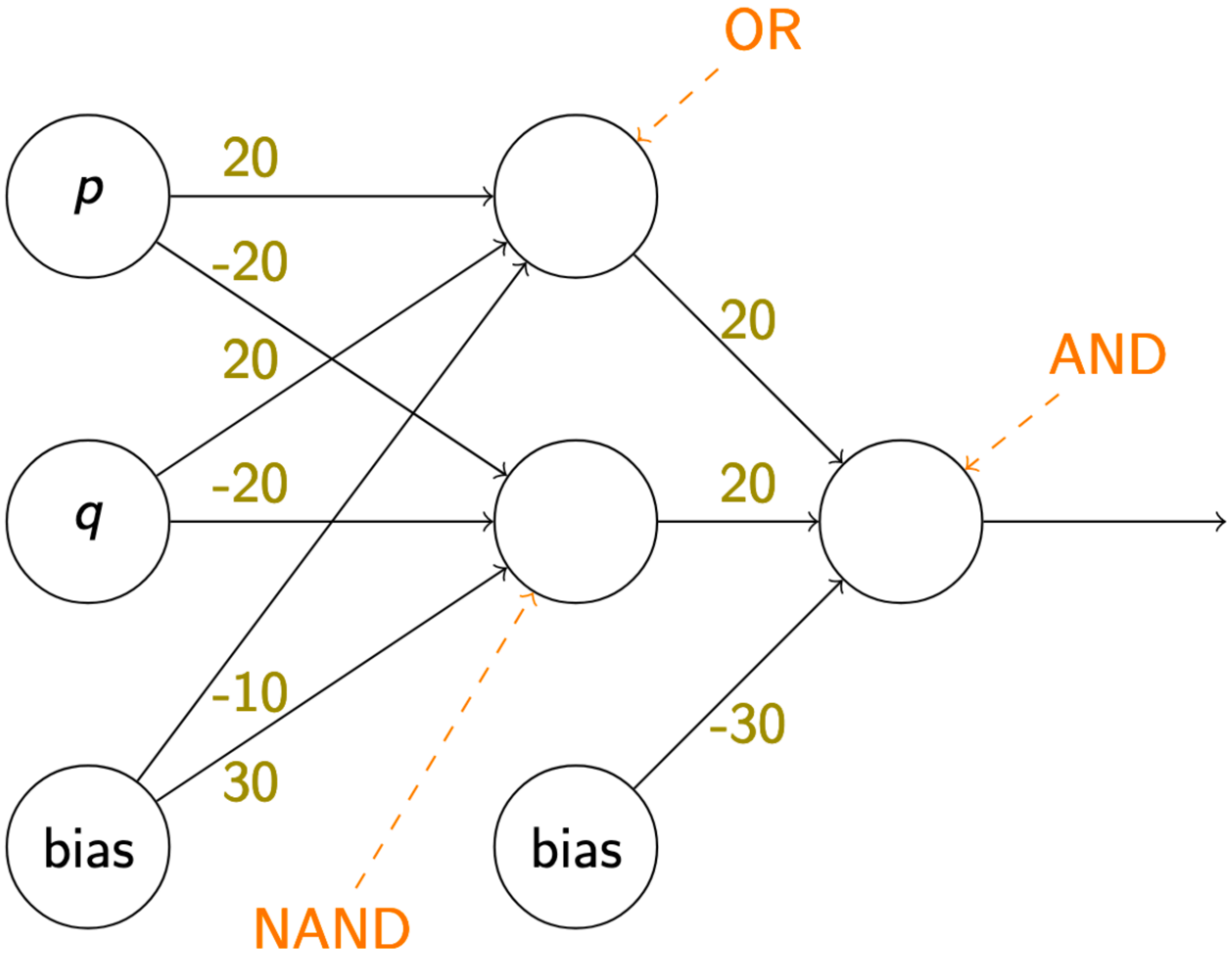


XOR Network



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

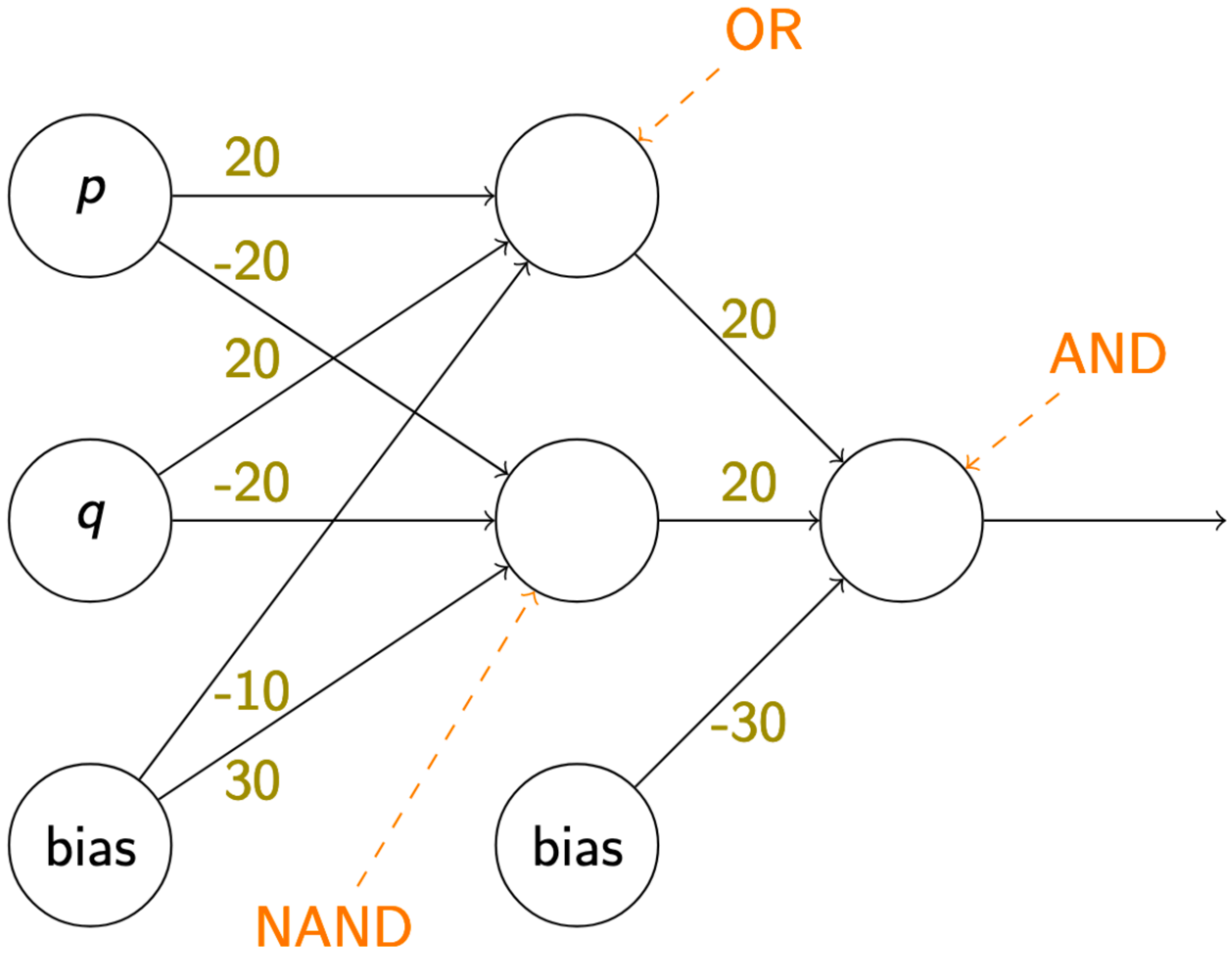
XOR Network



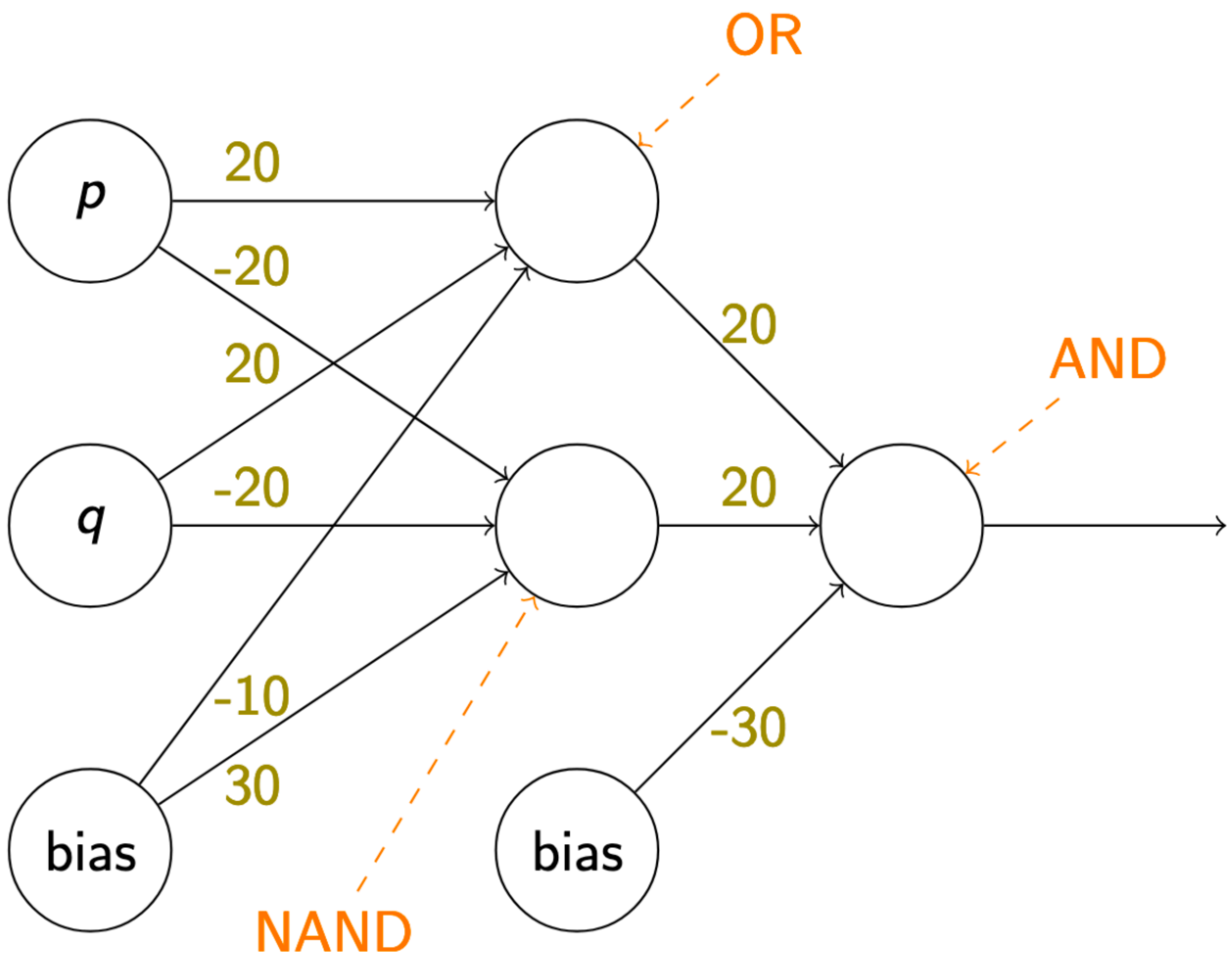
$$a_{\text{and}} = \sigma\left(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}\right)$$

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

XOR Network

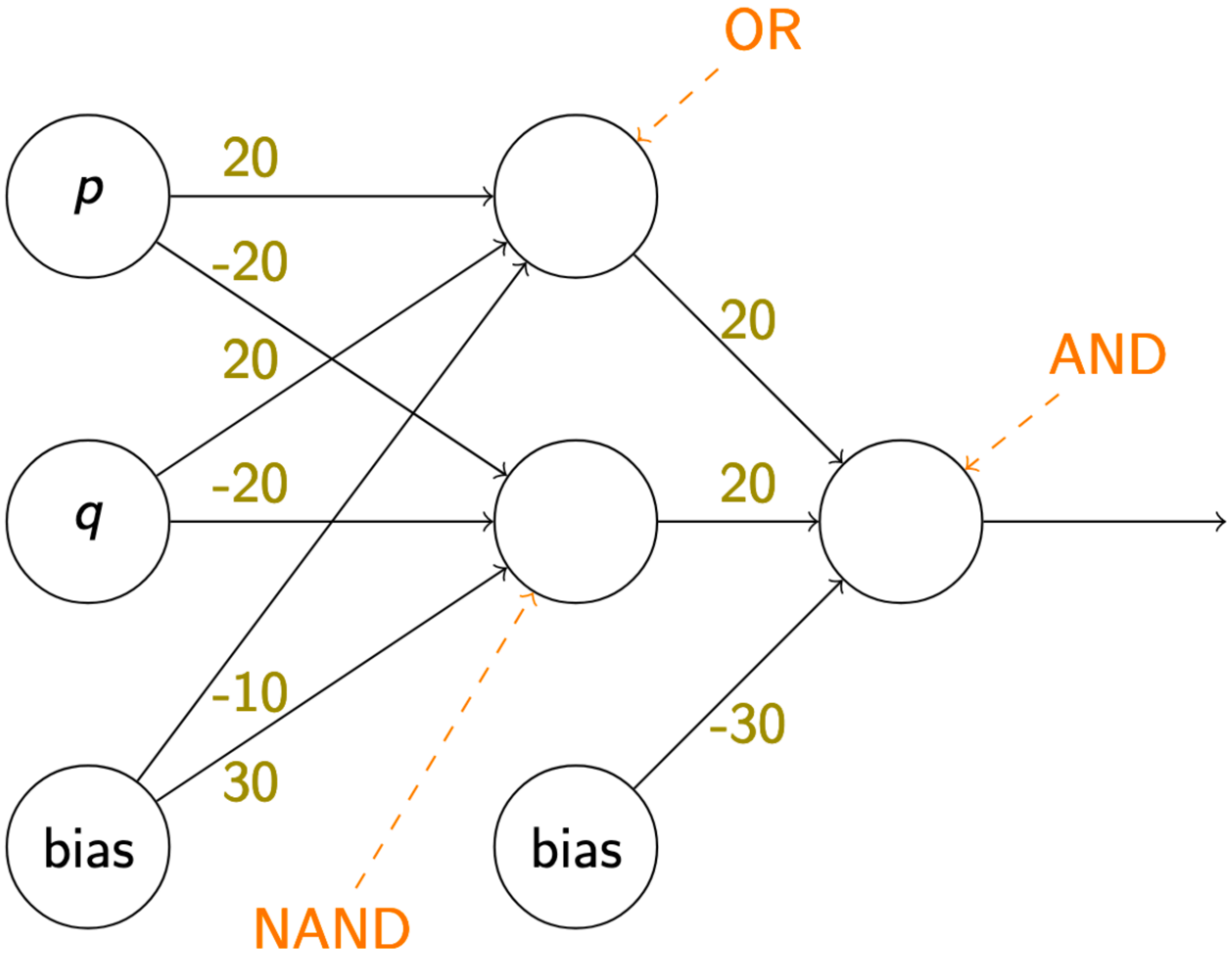


XOR Network



$$a_{\text{or}} = \sigma\left(w_p^{\text{or}} \cdot a_p + w_q^{\text{or}} \cdot a_q + b^{\text{or}}\right)$$

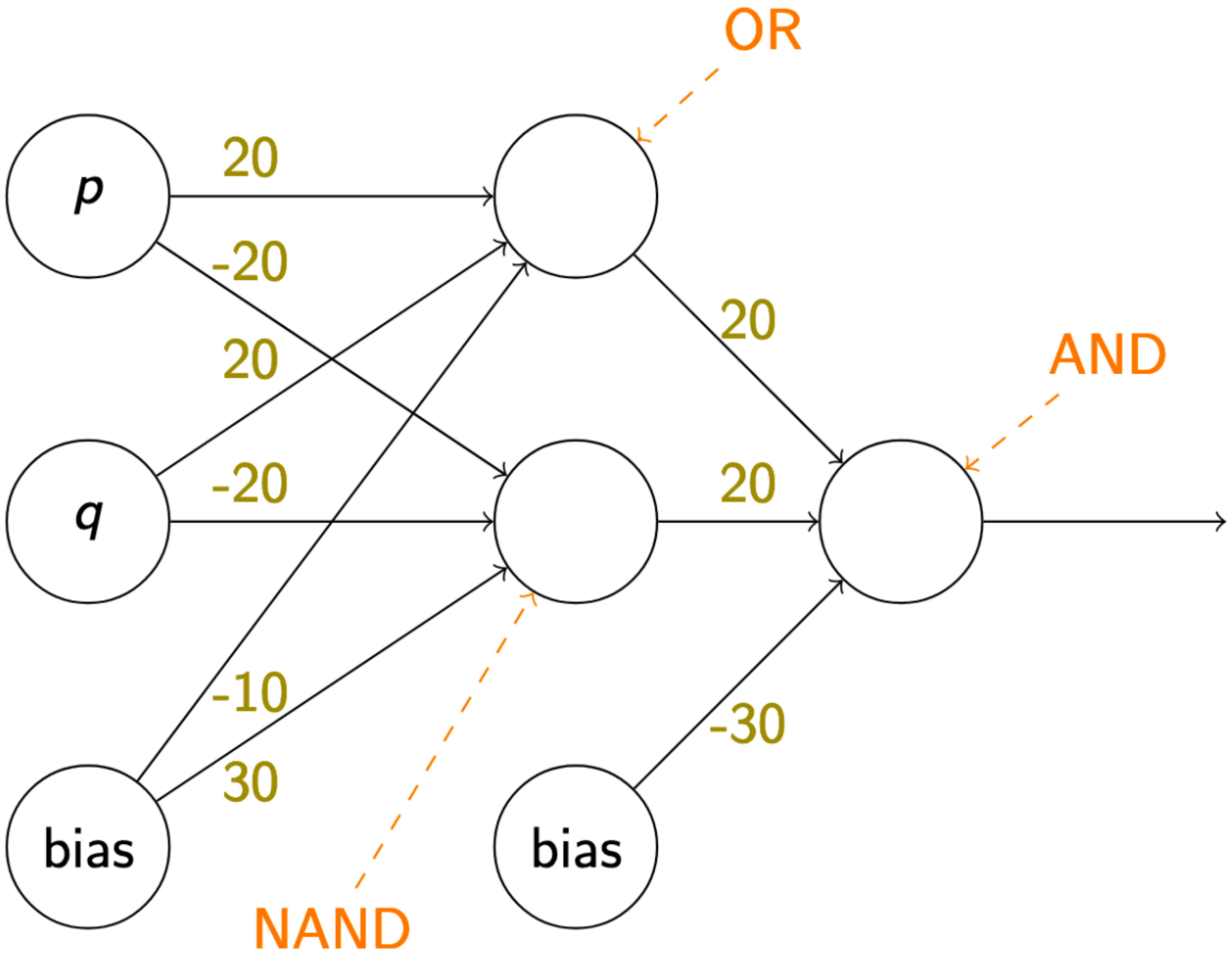
XOR Network



$$a_{\text{or}} = \sigma\left(w_p^{\text{or}} \cdot a_p + w_q^{\text{or}} \cdot a_q + b^{\text{or}}\right)$$

$$a_{\text{nand}} = \sigma\left(w_p^{\text{nand}} \cdot a_p + w_q^{\text{nand}} \cdot a_q + b^{\text{nand}}\right)$$

XOR Network

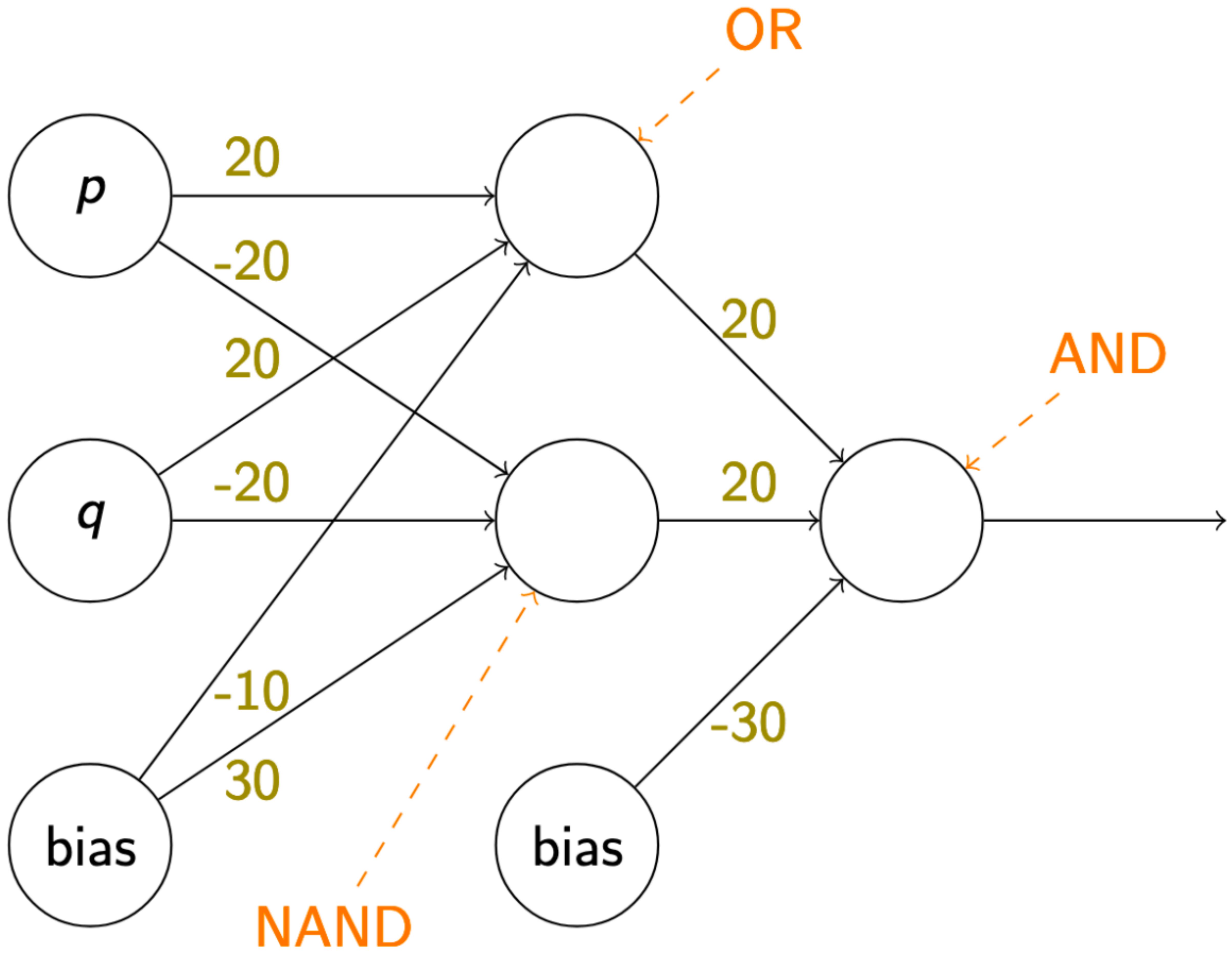


$$a_{\text{and}} = \sigma\left(w^{\text{and}}_{\text{or}} \cdot a_{\text{or}} + w^{\text{and}}_{\text{nand}} \cdot a_{\text{nand}} + b^{\text{and}}\right)$$

$$a_{\text{or}} = \sigma\left(w^{\text{or}}_{p} \cdot a_p + w^{\text{or}}_{q} \cdot a_q + b^{\text{or}}\right)$$

$$a_{\text{nand}} = \sigma\left(w^{\text{nand}}_{p} \cdot a_p + w^{\text{nand}}_{q} \cdot a_q + b^{\text{nand}}\right)$$

XOR Network



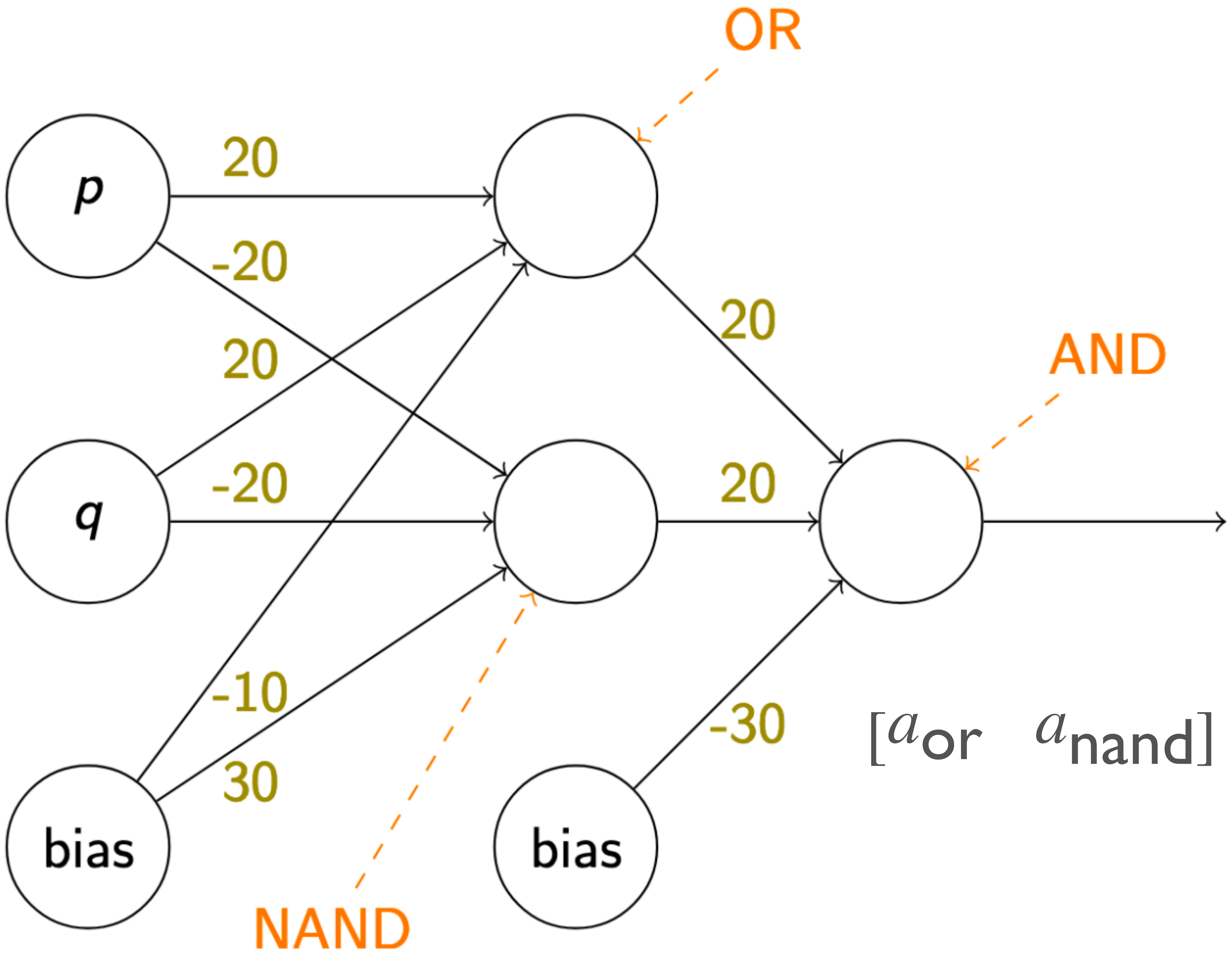
$$a_{\text{and}} = \sigma\left(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}\right)$$

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

$$a_{\text{or}} = \sigma\left(w_p^{\text{or}} \cdot a_p + w_q^{\text{or}} \cdot a_q + b^{\text{or}}\right)$$

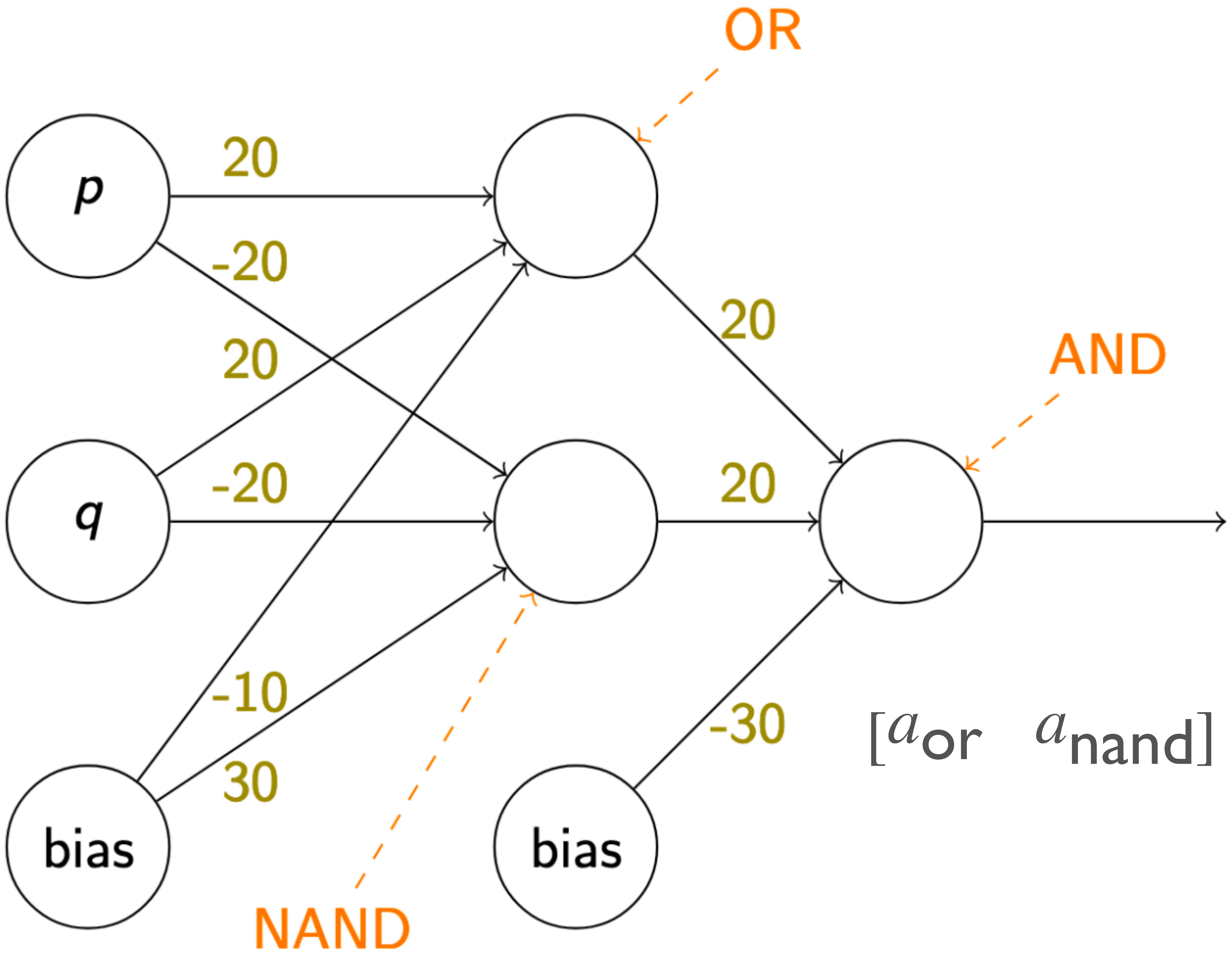
$$a_{\text{nand}} = \sigma\left(w_p^{\text{nand}} \cdot a_p + w_q^{\text{nand}} \cdot a_q + b^{\text{nand}}\right)$$

XOR Network



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

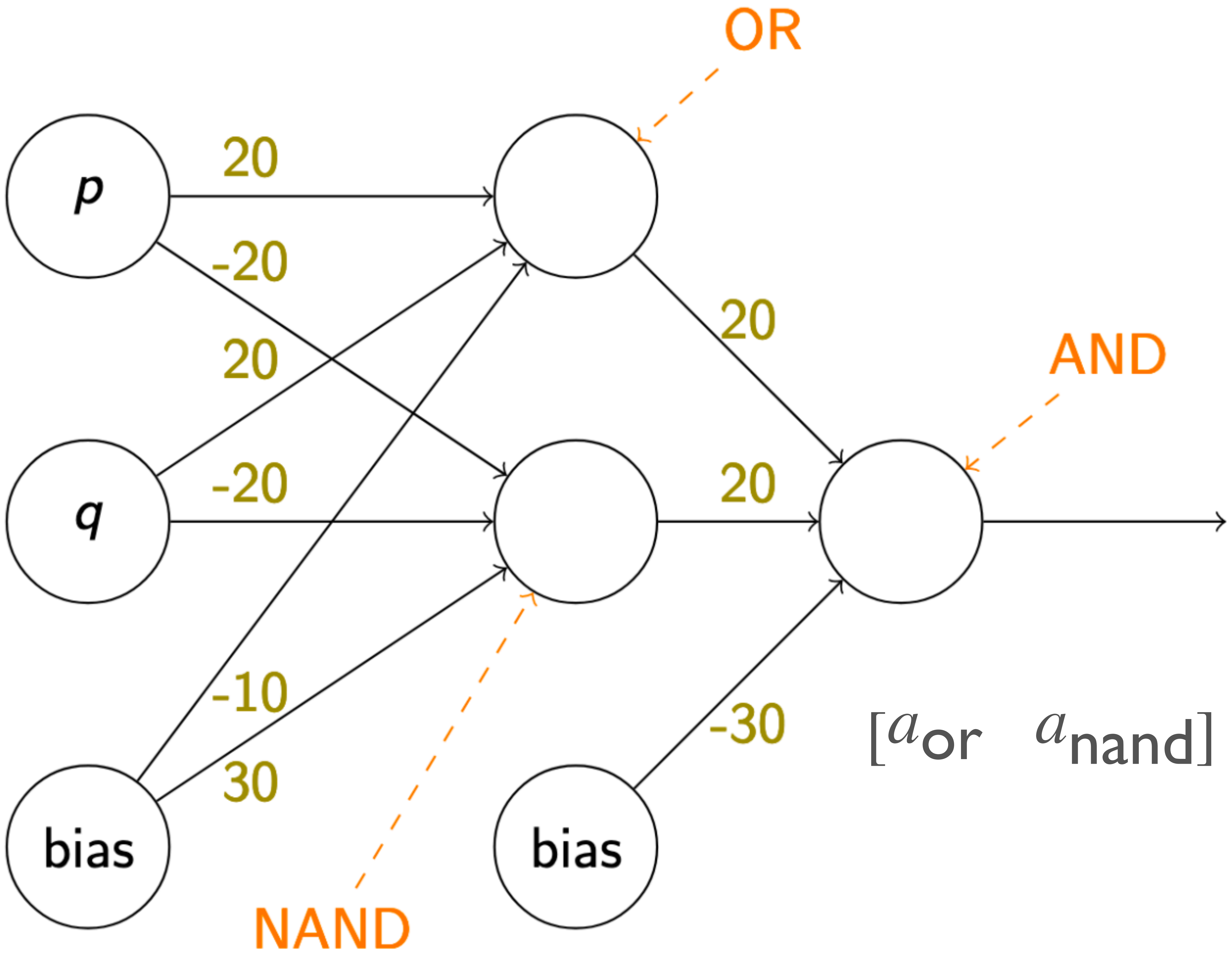
XOR Network



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

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

XOR Network

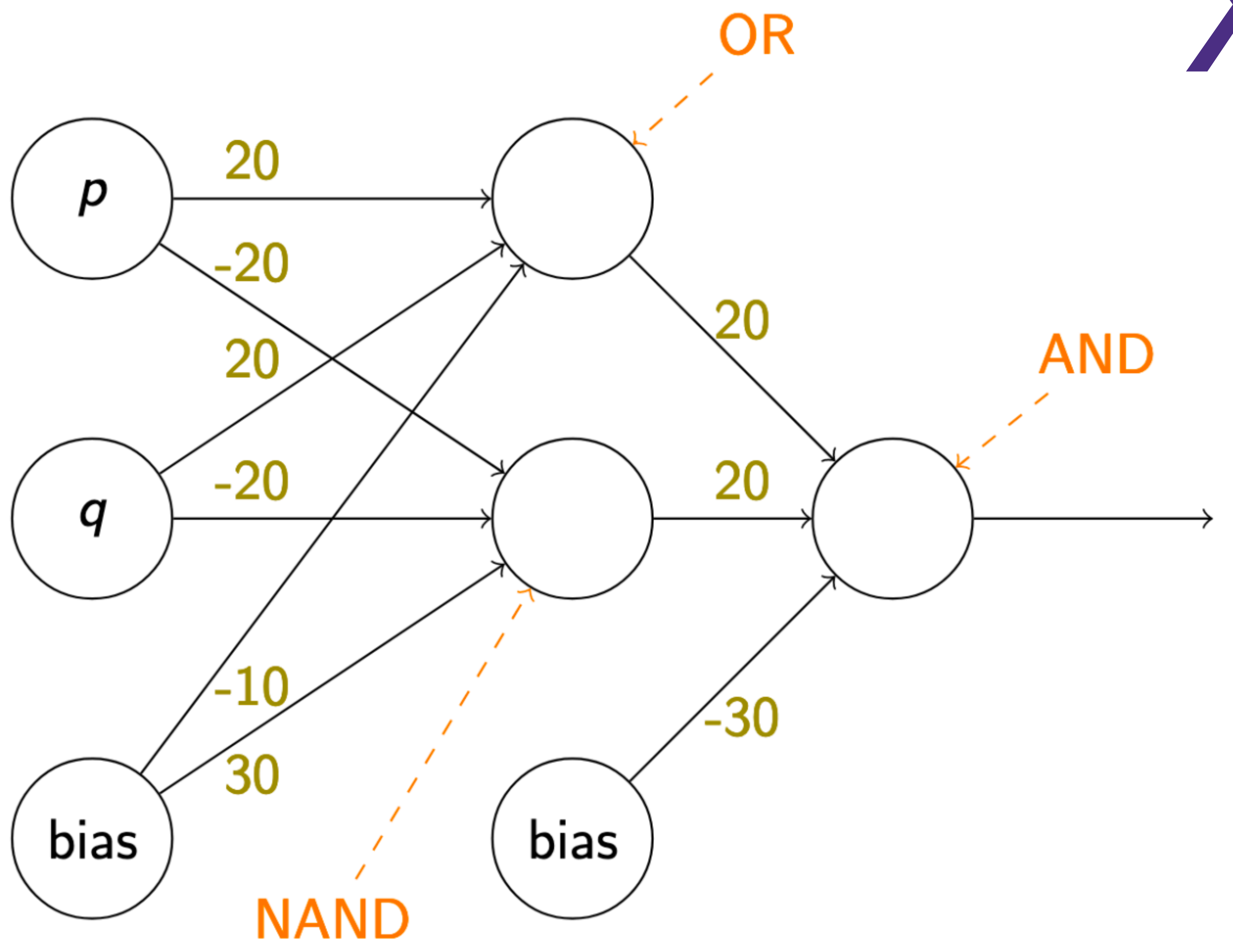


$$a_{\text{and}} = \sigma\left(w_{\text{or}}^{\text{and}} \cdot a_{\text{or}} + w_{\text{nand}}^{\text{and}} \cdot a_{\text{nand}} + b^{\text{and}}\right)$$

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

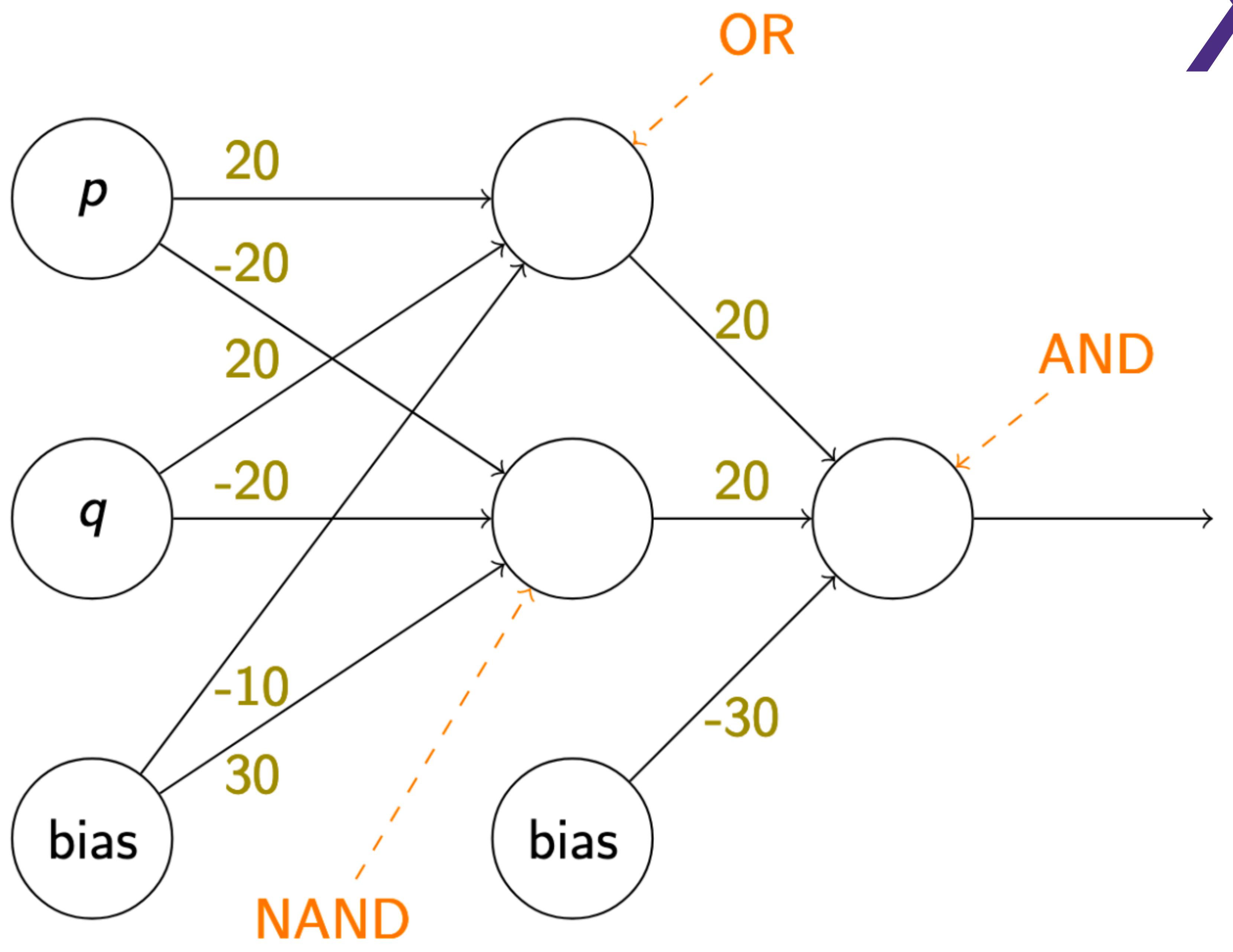
$$\begin{bmatrix} a_{\text{or}} & a_{\text{nand}} \end{bmatrix} = \sigma\left(\begin{bmatrix} a_p & a_q \end{bmatrix} \begin{bmatrix} w_p^{\text{or}} & w_p^{\text{nand}} \\ w_q^{\text{or}} & w_q^{\text{nand}} \end{bmatrix} + \begin{bmatrix} b^{\text{or}} & b^{\text{nand}} \end{bmatrix}\right)$$

XOR Network



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

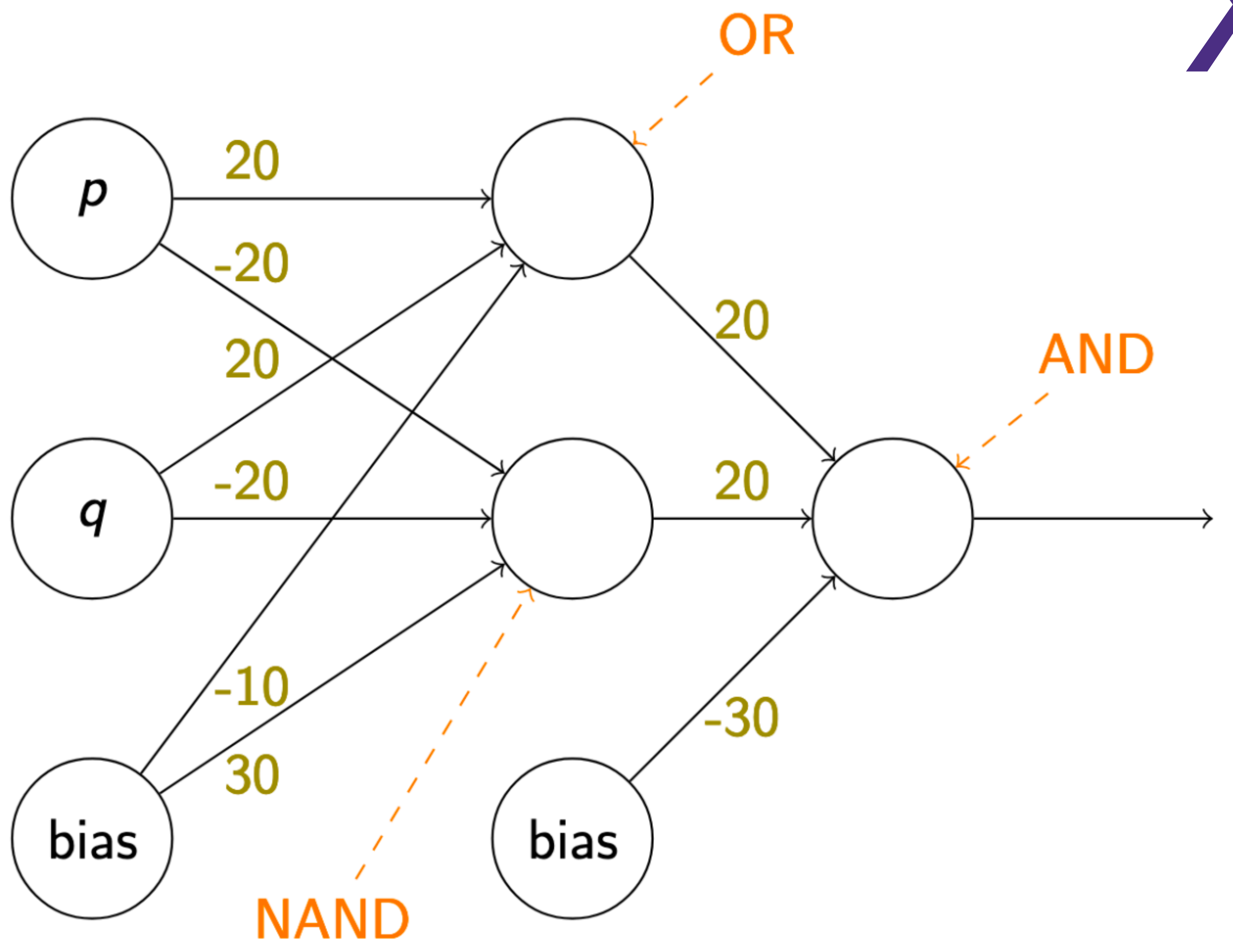
XOR Network



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

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

XOR Network



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

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

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

Generalizing

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

Generalizing

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

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

Generalizing

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

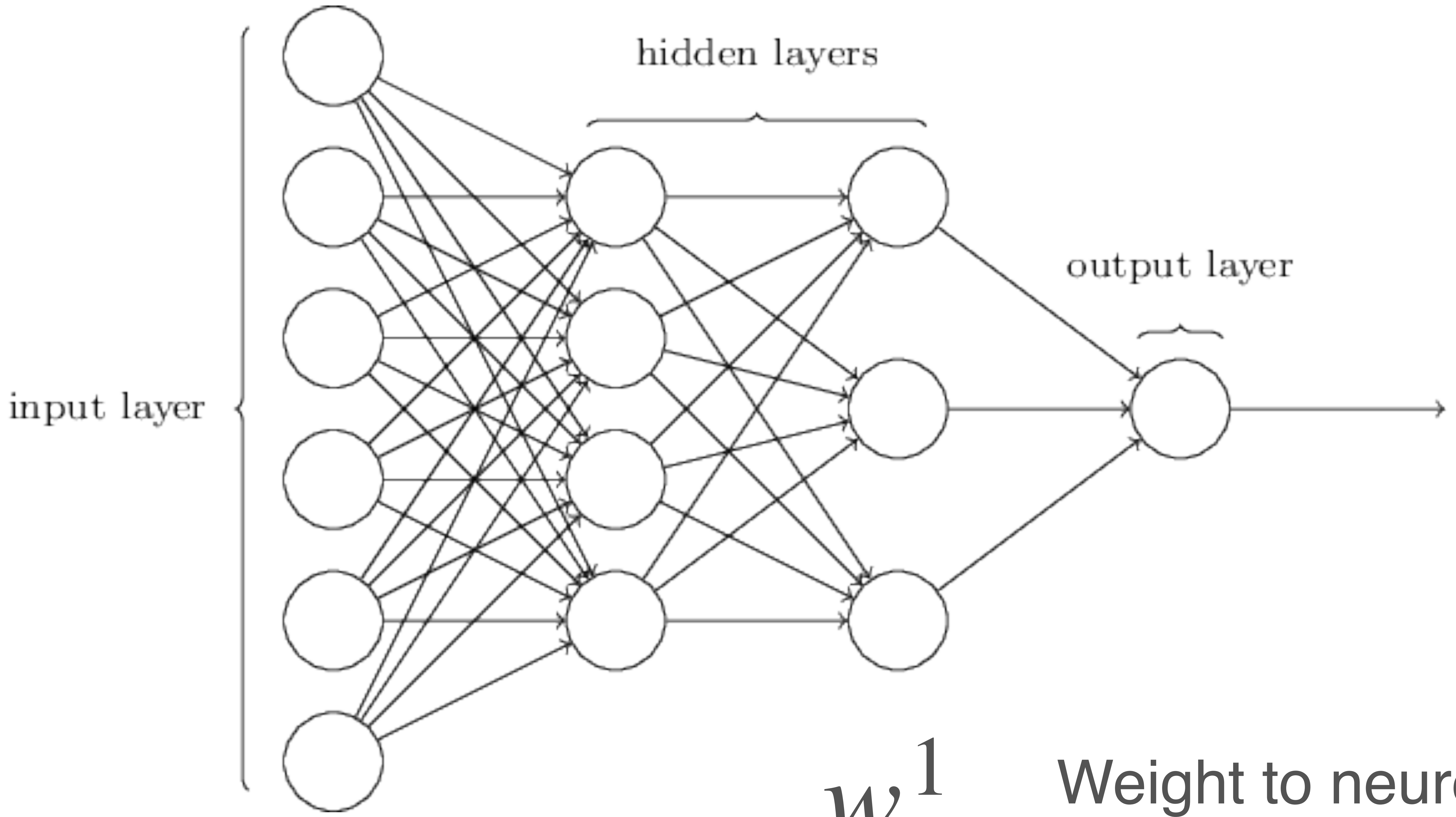
$$\hat{y} = f_2\left(f_1\left(xW^1 + b^1\right)W^2 + b^2\right)$$

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

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

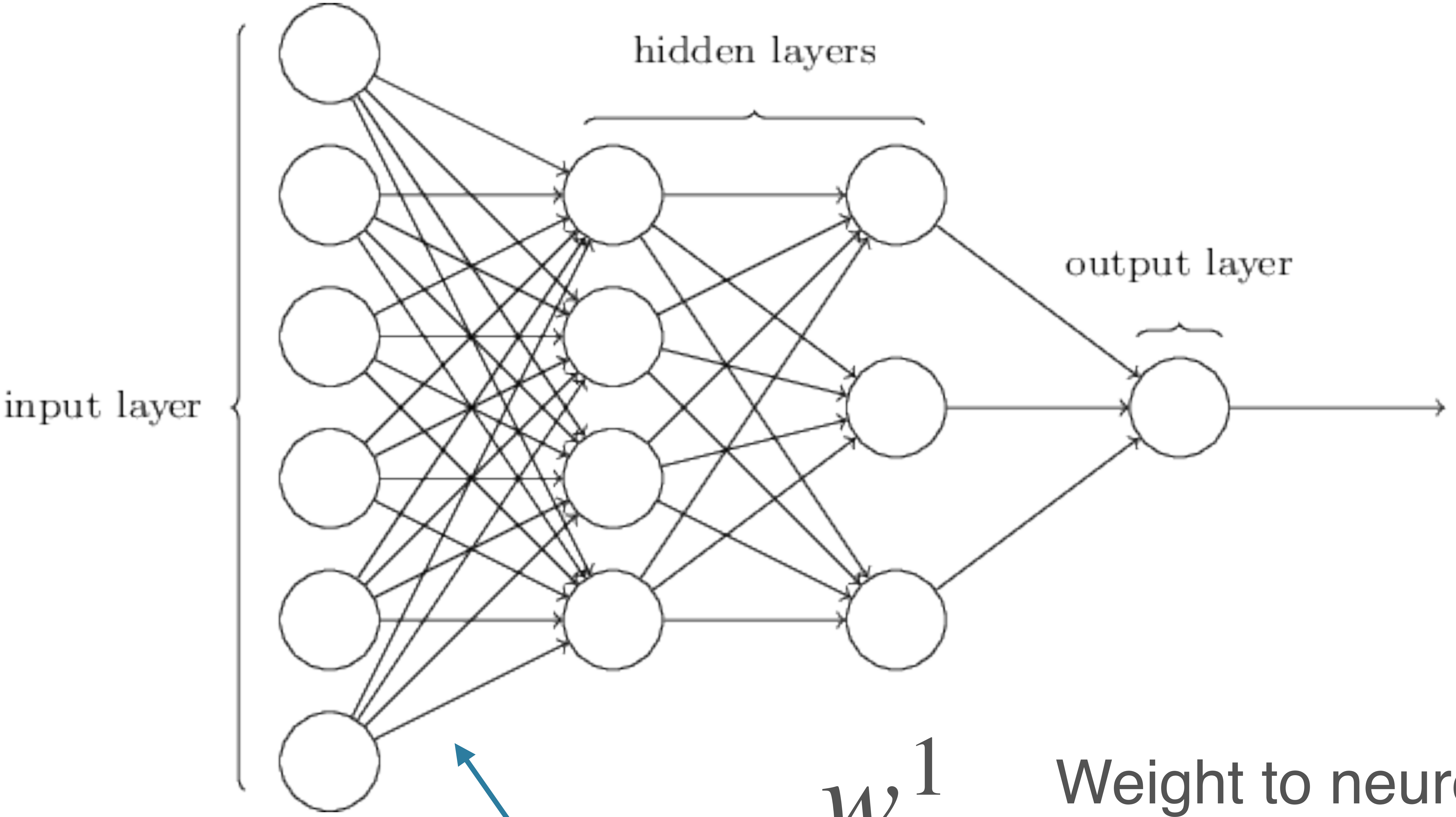


source

$$w_{ij}^{1}$$

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

General MLP



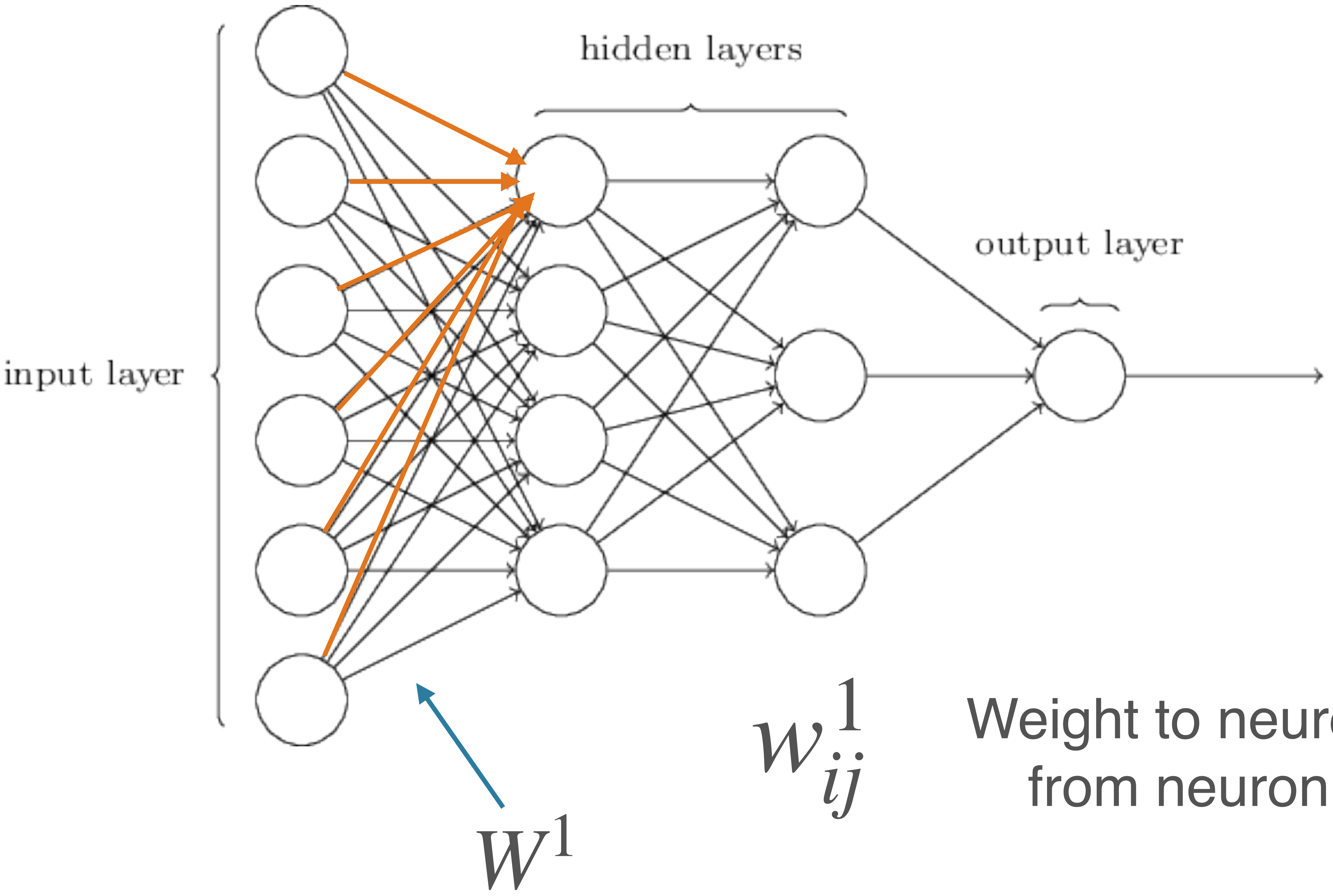
source

W^1

w_{ij}^1 Weight to neuron j in layer 1 from neuron i in layer 0

General MLP

source



w_{ij}^1 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

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

$$x = \begin{bmatrix} x_0 & x_1 & \cdots & x_{n_0} \end{bmatrix}$$

Shape: $(1, n_0)$

General MLP

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

$$x = \begin{bmatrix} x_0 & x_1 & \cdots & x_{n_0} \end{bmatrix}$$

Shape: $(1, n_0)$

$$W^1 = \begin{bmatrix} w^1_{00} & w^1_{10} & \cdots & w^1_{0n_1} \\ w^1_{10} & w^1_{11} & \cdots & w^1_{1n_1} \\ \vdots & \vdots & \ddots & \vdots \\ w^1_{n_0 0} & w^1_{n_0 1} & \cdots & w^1_{n_0 n_1} \end{bmatrix}$$

General MLP

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

$$x = \begin{bmatrix} x_0 & x_1 & \cdots & x_{n_0} \end{bmatrix}$$

Shape: $(1, n_0)$

$$W^1 = \begin{bmatrix} w^1_{00} & w^1_{10} & \cdots & w^1_{0n_1} \\ w^1_{10} & w^1_{11} & \cdots & w^1_{1n_1} \\ \vdots & \vdots & \ddots & \vdots \\ w^1_{n_0 0} & w^1_{n_0 1} & \cdots & w^1_{n_0 n_1} \end{bmatrix}$$

Shape: (n_0, n_1)

n_0: number of neurons in layer 0 (input)

n_1: number of neurons in layer 1

General MLP

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

$$x = \begin{bmatrix} x_0 & x_1 & \cdots & x_{n_0} \end{bmatrix}$$

Shape: $(1, n_0)$

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

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

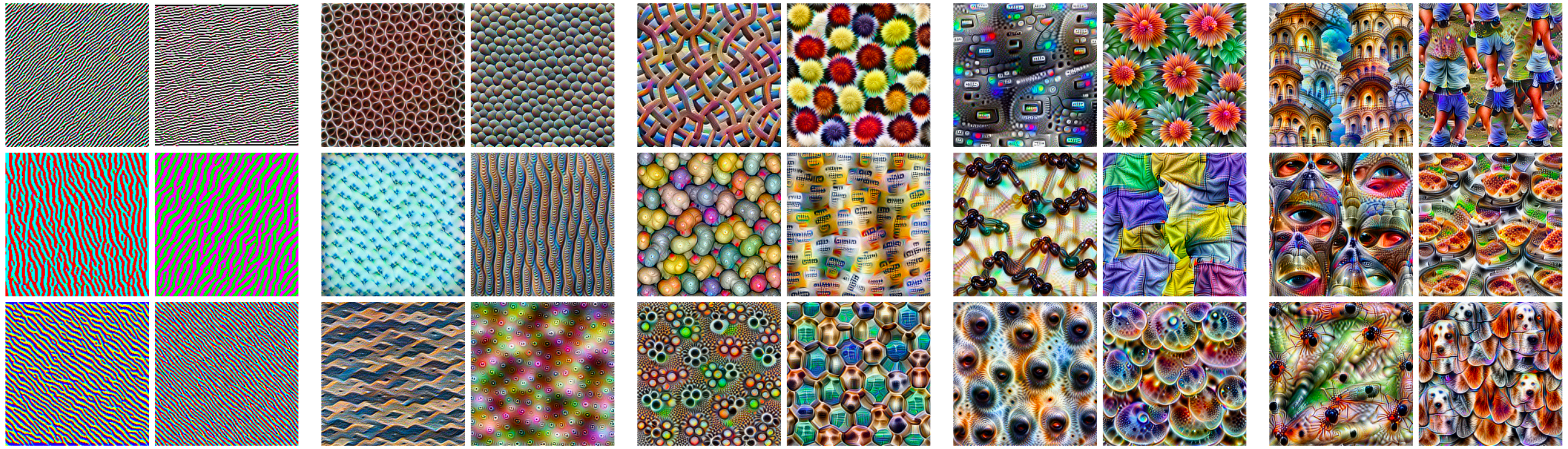
- Input size, output size
  - Usually fixed by your problem / dataset
  - Input: image size, vocab size; number of “raw” features in general
  - Output: 1 for binary classification or simple regression, number of labels for classification, …
- *Number* of hidden layers
- For each hidden layer:
  - Size
  - 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”
  - In principle allows hierarchical features to be learned
  - More well-behaved w/r/t optimization



Edges (layer conv2d0) **Textures** (layer mixed3a) **Patterns** (layer mixed4a) **Parts** (layers mixed4b & mixed4c) **Objects** (layers mixed4d & mixed4e)

source

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!

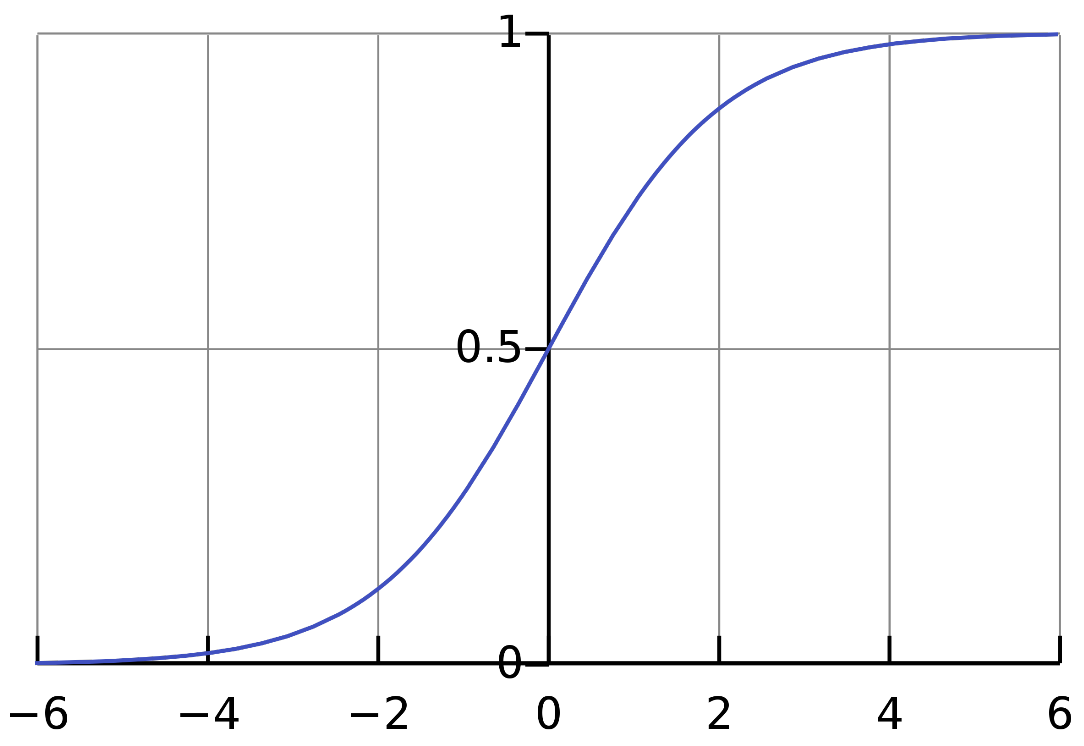
Activation Functions

- 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!

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

Activation Functions: Hidden Layer

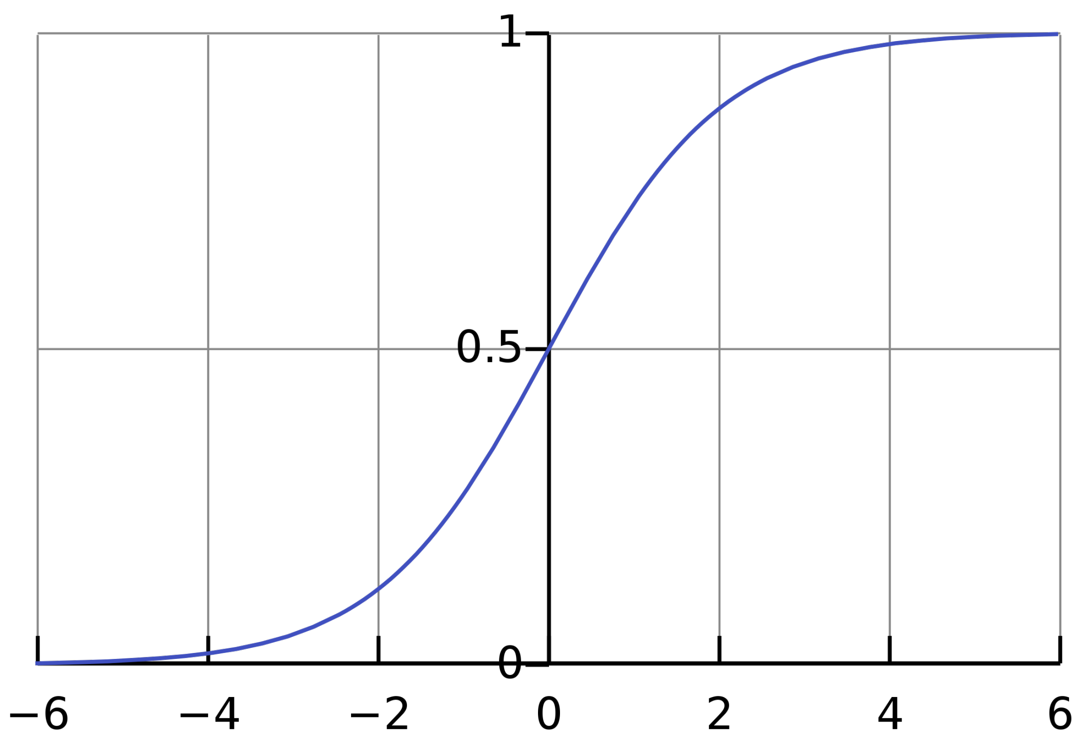
sigmoid



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

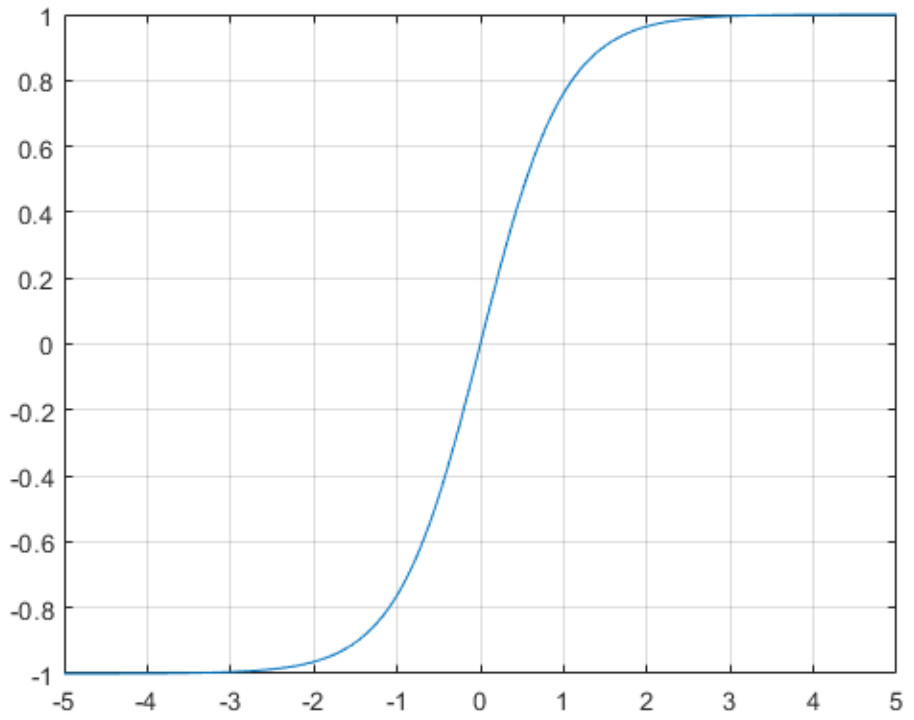
Activation Functions: Hidden Layer

sigmoid



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

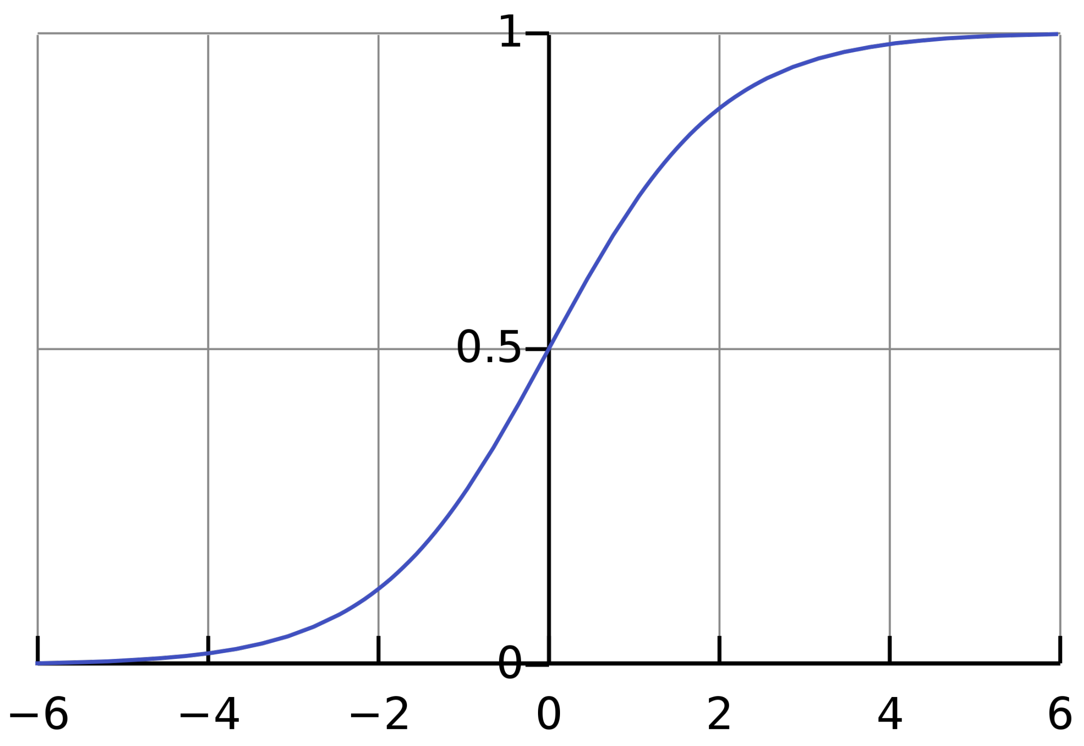
tanh



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

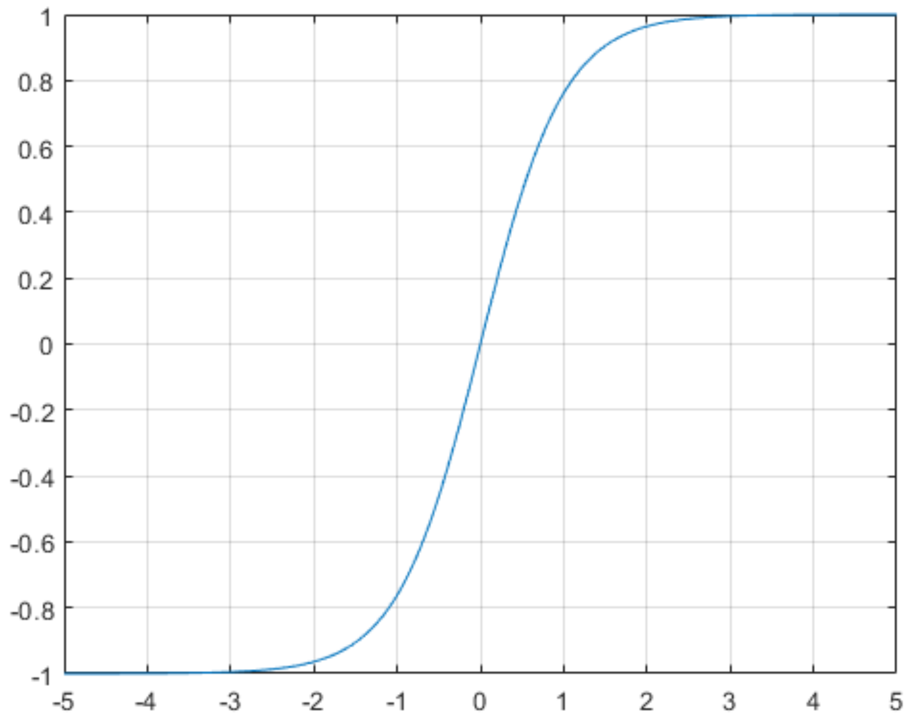
Activation Functions: Hidden Layer

sigmoid



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

tanh

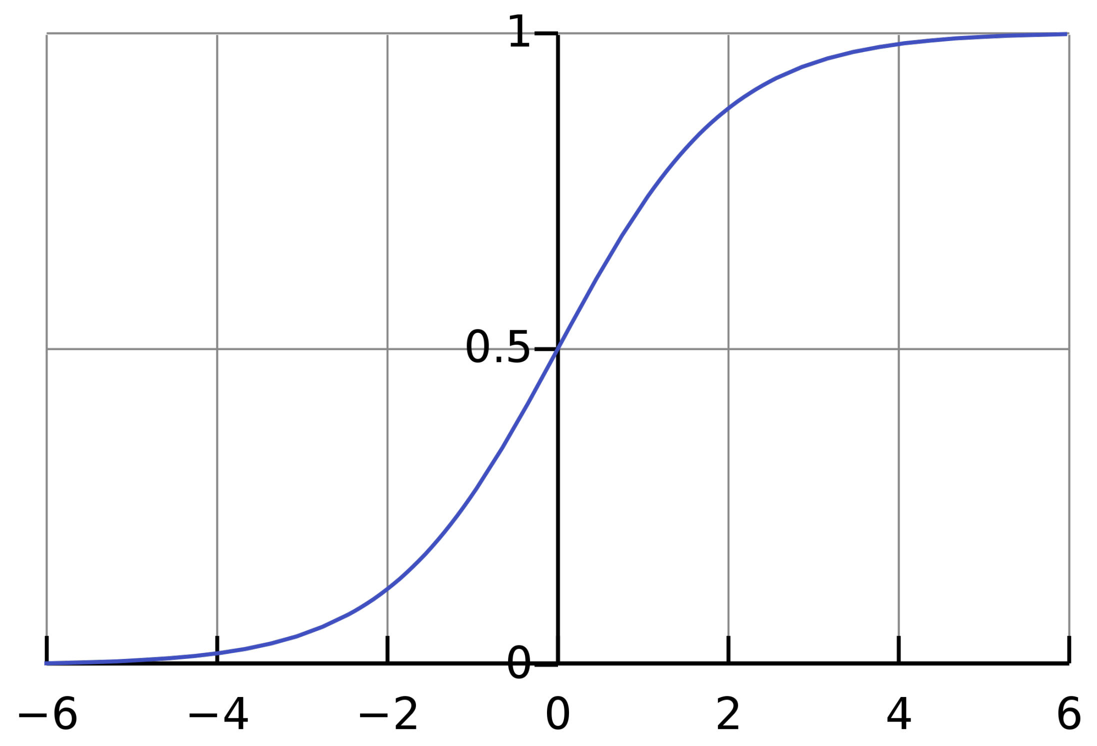


$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

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

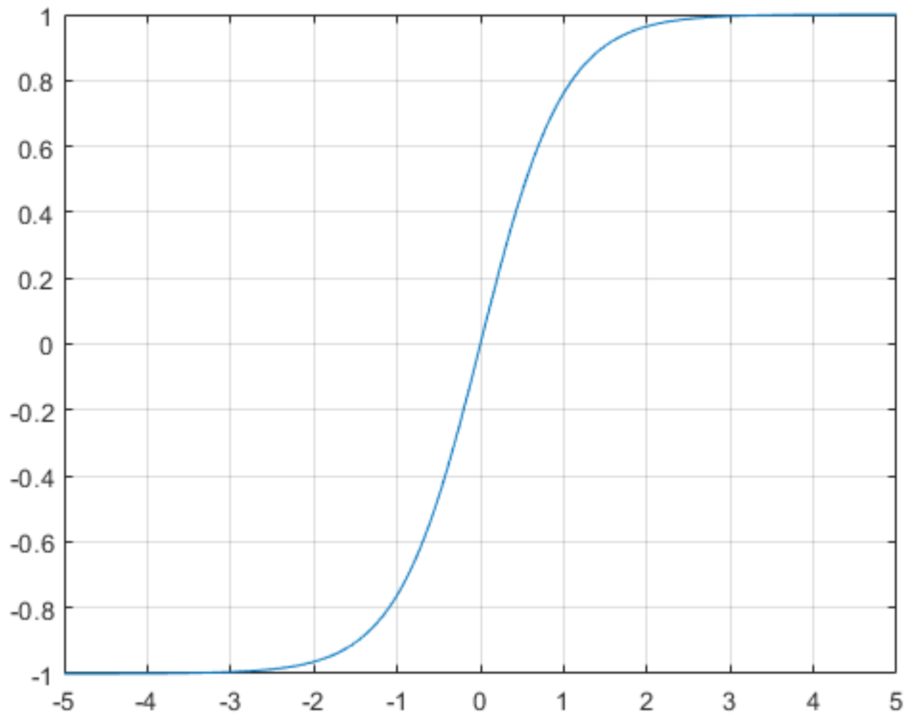
Activation Functions: Hidden Layer

sigmoid

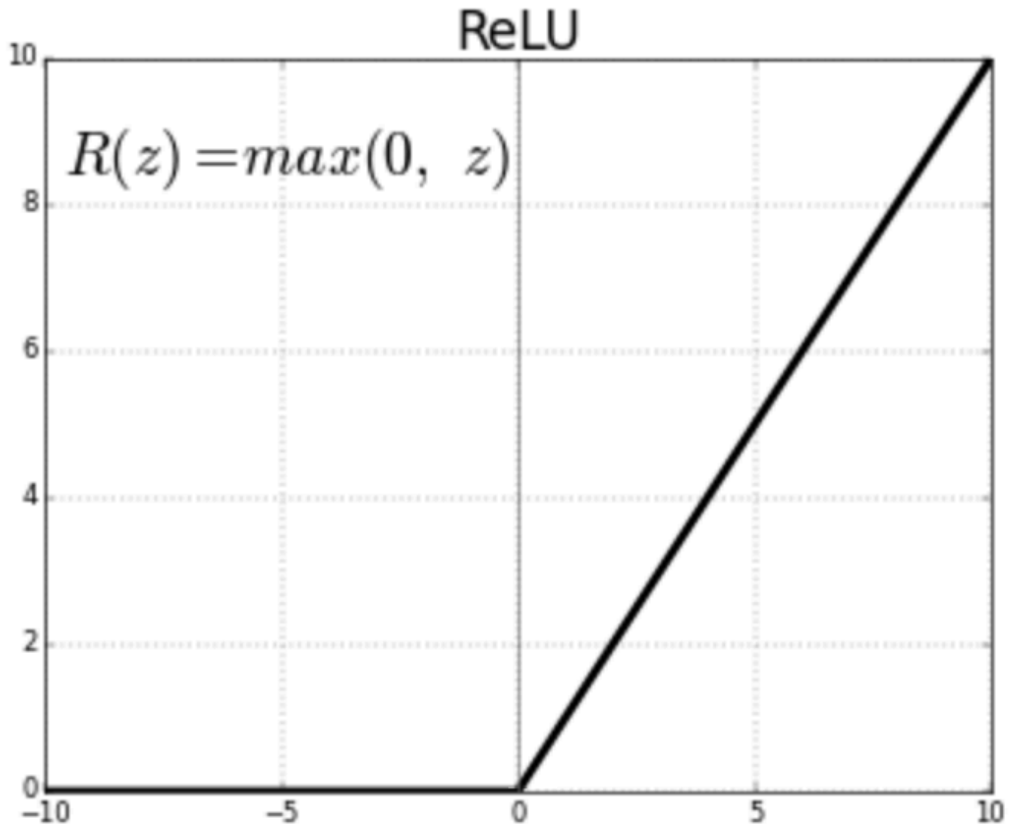


$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

tanh



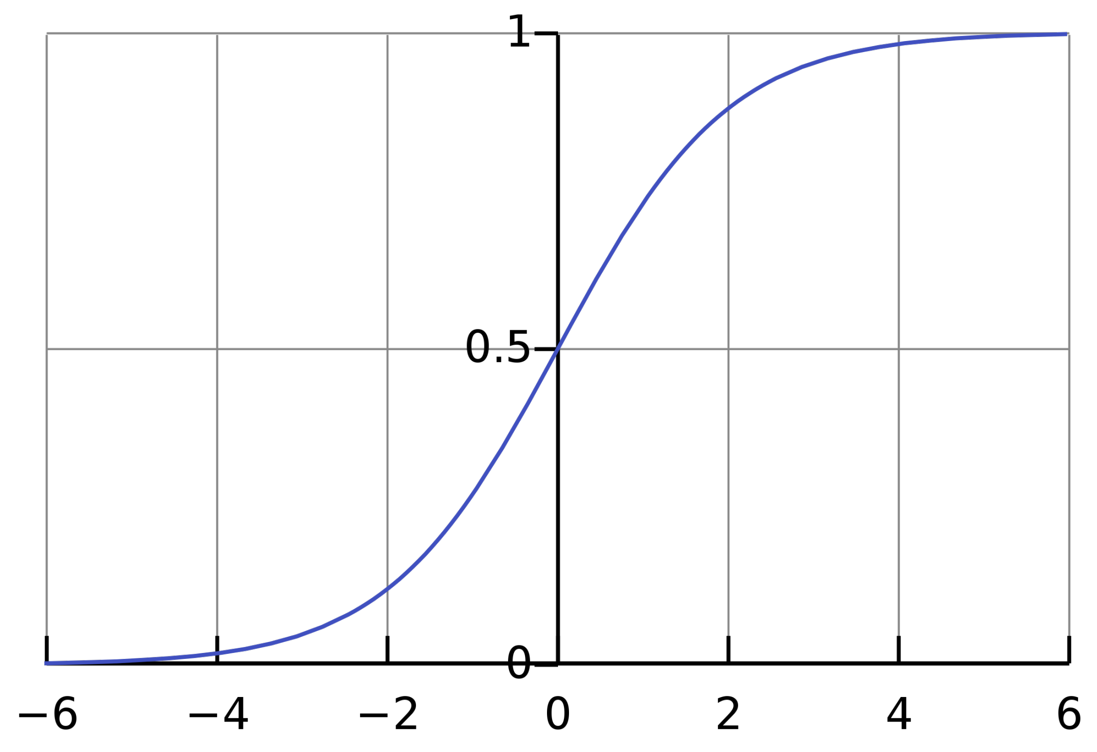
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$



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

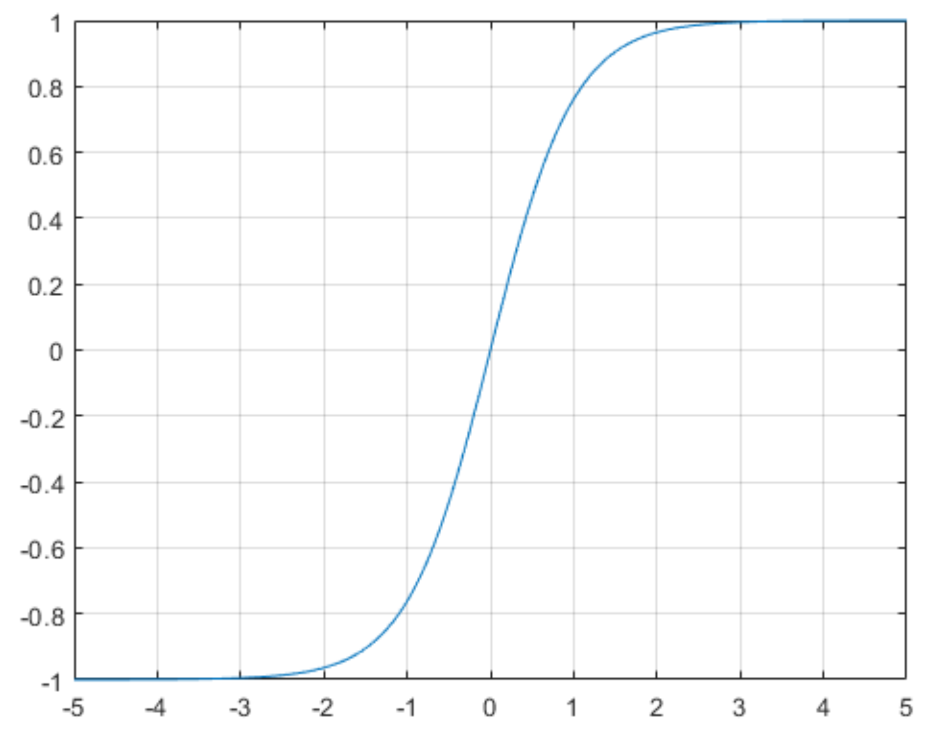
Activation Functions: Hidden Layer

sigmoid

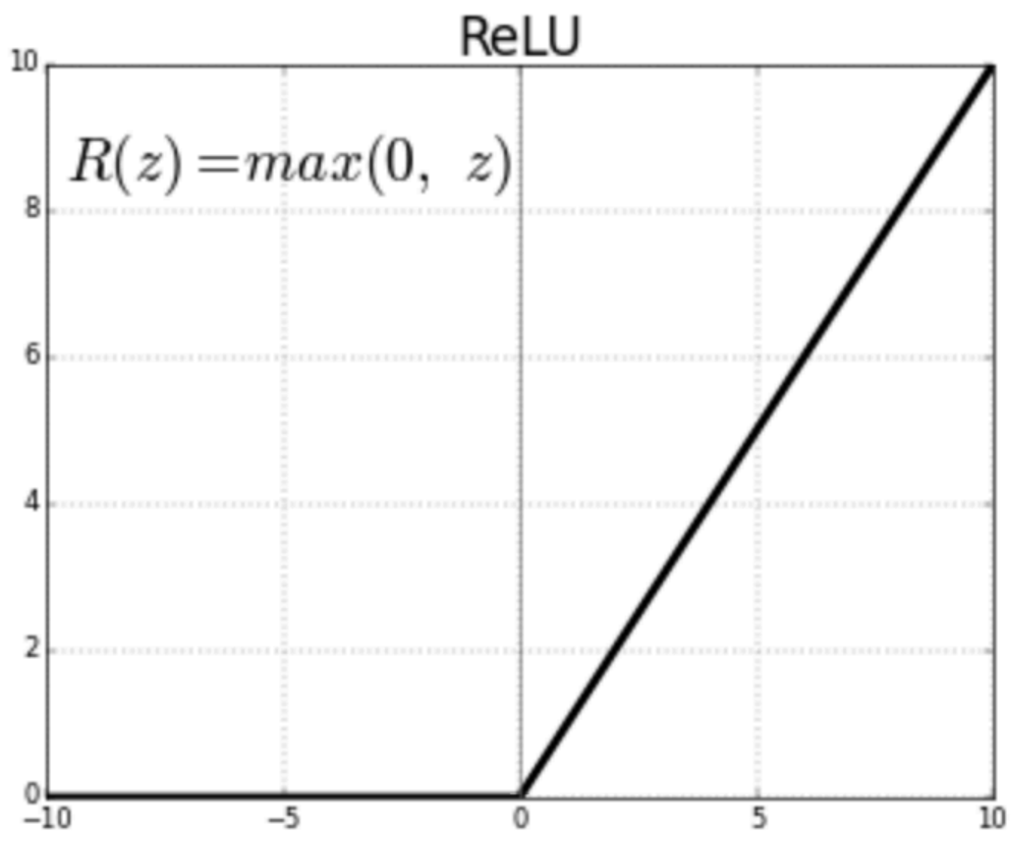


$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

tanh



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



- 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]

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

Mini-batch computation

Computing with a Single Input

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

$$x = \begin{bmatrix} x_0 & x_1 & \cdots & x_{n_0} \end{bmatrix}$$

Shape: $(1, n_0)$

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

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

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

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

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

Shape: (n, n_0)

n: batch_size

Computing with a Batch of Inputs

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

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

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!!

Additional Training Notes

Early stopping

source

Early stopping

- One: Pick # of epochs, hope for no overfitting

source

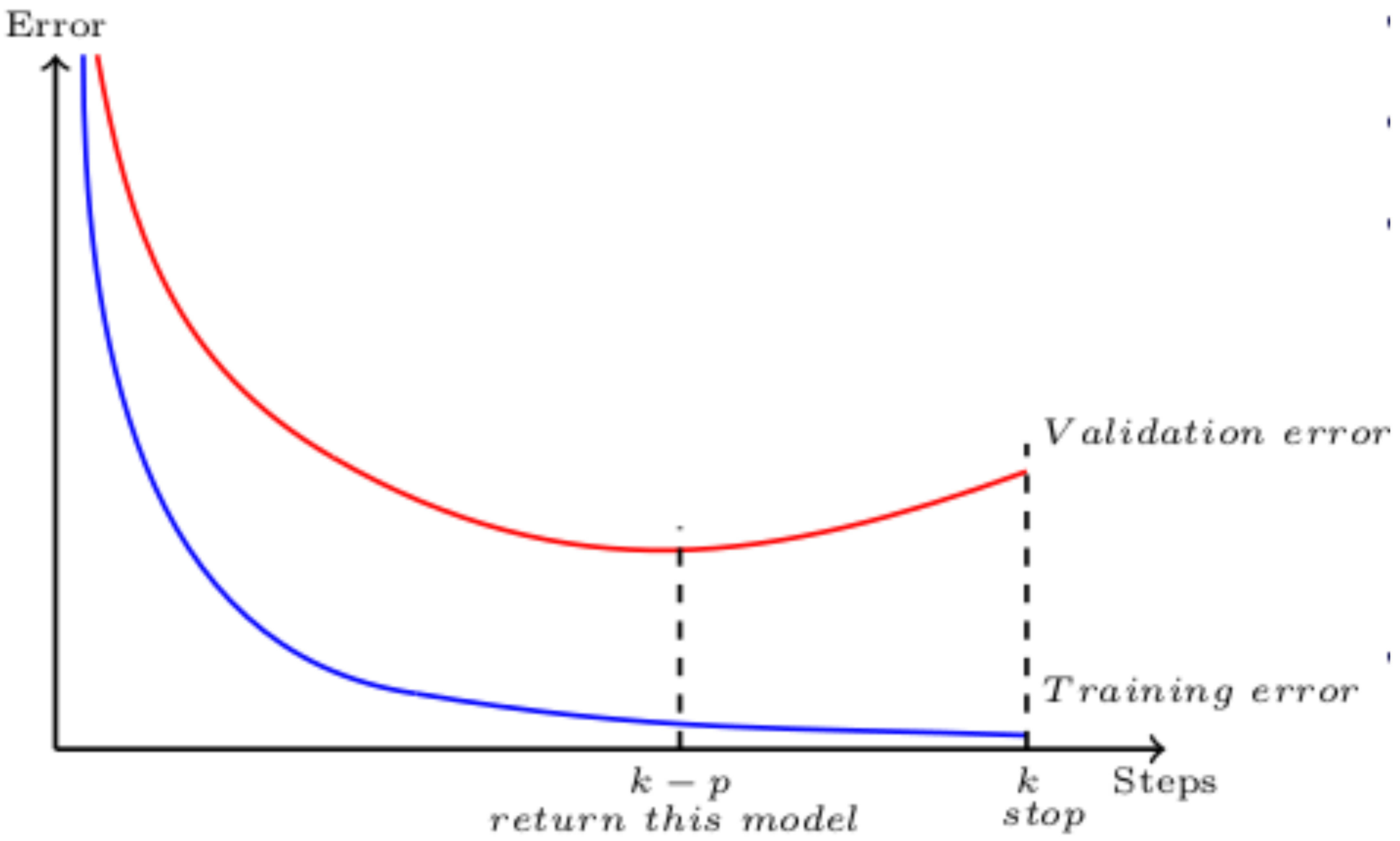
Early stopping

- One: Pick # of epochs, hope for no overfitting
- Better: pick max # of epochs, and “patience”
  - Halt when validation error does not improve over patience-many epochs

source

Early stopping

- One: Pick # of epochs, hope for no overfitting
- Better: pick max # of epochs, and “patience”
  - Halt when validation error does not improve over patience-many epochs



[source]

Regularization

- NNs are often *overparameterized*, so regularization helps
- L1/L2: $\mathscr{L}'(\theta, y) = \mathscr{L}(\theta, y) + \lambda \|\theta\|^2$
- Dropout (2012):
  - *During training*, randomly turn off X% of neurons in each layer
  - (Don't do this during testing/predicting)
- Batch Normalization (2015)

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

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

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

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

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

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

Hyper-parameters

- In addition to the model architecture ones mentioned earlier
- Optimizer: SGD, Adam, Adagrad, RMSProp, ….
  - Optimizer-specific hyper-parameters: learning rate, alpha, beta, …
  - NB: backprop computes gradients; optimizer uses them to update parameters
- Regularization: L1/L2, Dropout, BN, …
  - regularizer-specific ones: e.g. dropout rate
- Batch size
- Number of epochs to train for
  - Early stopping criterion (e.g. number of epochs, “patience”)

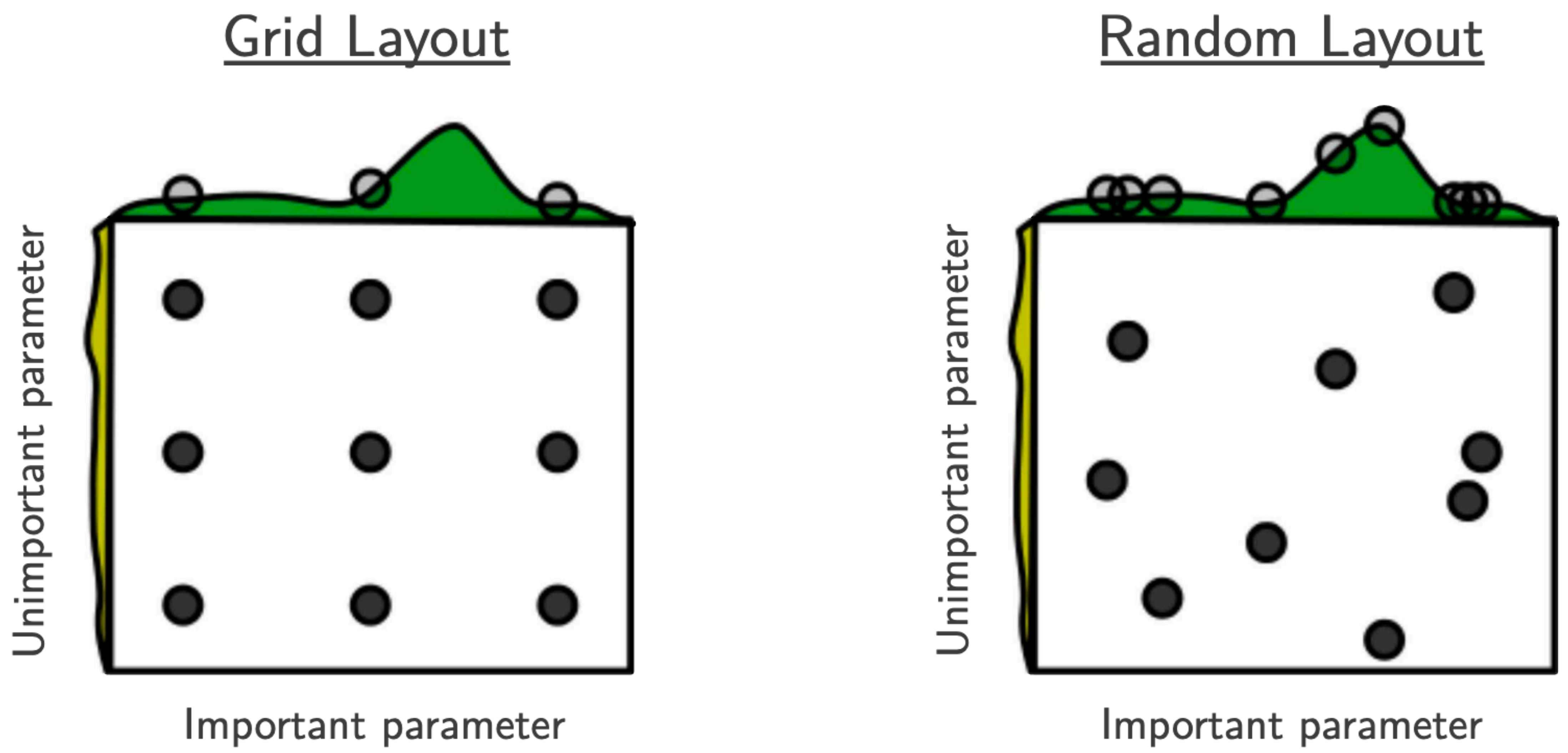
A note on hyper-parameter tuning

- Grid search: specify range of values for each hyper-parameter, try all possible combinations thereof
- Random search: specify possible values for all parameters, randomly sample values for each, stop when some criterion is met

Bergstra and Bengio 2012

A note on hyper-parameter tuning

- Grid search: specify range of values for each hyper-parameter, try all possible combinations thereof
- Random search: specify possible values for all parameters, randomly sample values for each, stop when some criterion is met



Bergstra and Bengio 2012

Homework 2

Learning Goals

- Understand skip-gram with negative sampling in more detail
  - Compute various derivatives in order to get the gradient of the loss with respect to the parameters
- Learn how to translate math into code, for
  - The model forward pass
  - Gradient computations

Understanding Word2Vec

- Count parameters
- Understand sigmoid, and the role it plays in SGNS
  - Compute its derivative
- Compute the gradient of L_{CE} with respect to parameters
  - Done in stages
  - Uses:
    - Logarithm rules
    - Derivative of logarithm
    - Addition / product / chain rule for derivatives

Implementing Word2Vec

- SGNS will be implemented in raw numpy
- We provide the entire training loop, but various methods that are called need to be filled in
  - Data processing: generating positive and negative samples
  - Model computation: implement the $P(1 \mid w, c; \theta)$ computation
  - Gradient computation: compute ∇L_{CE} w/r/t each of the relevant parameters

Training Word Vectors

- Finally, you will train word vectors by iterating through the SST training set
- Plot the vectors of a list of words, using PCA for dimensionality reduction
  - We provide all of this code!
- Describe any trends you see in the embeddings

Next Time

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