Neural Network Introduction

Ling 282/482: Deep Learning for Computational Linguistics

C.M. Downey

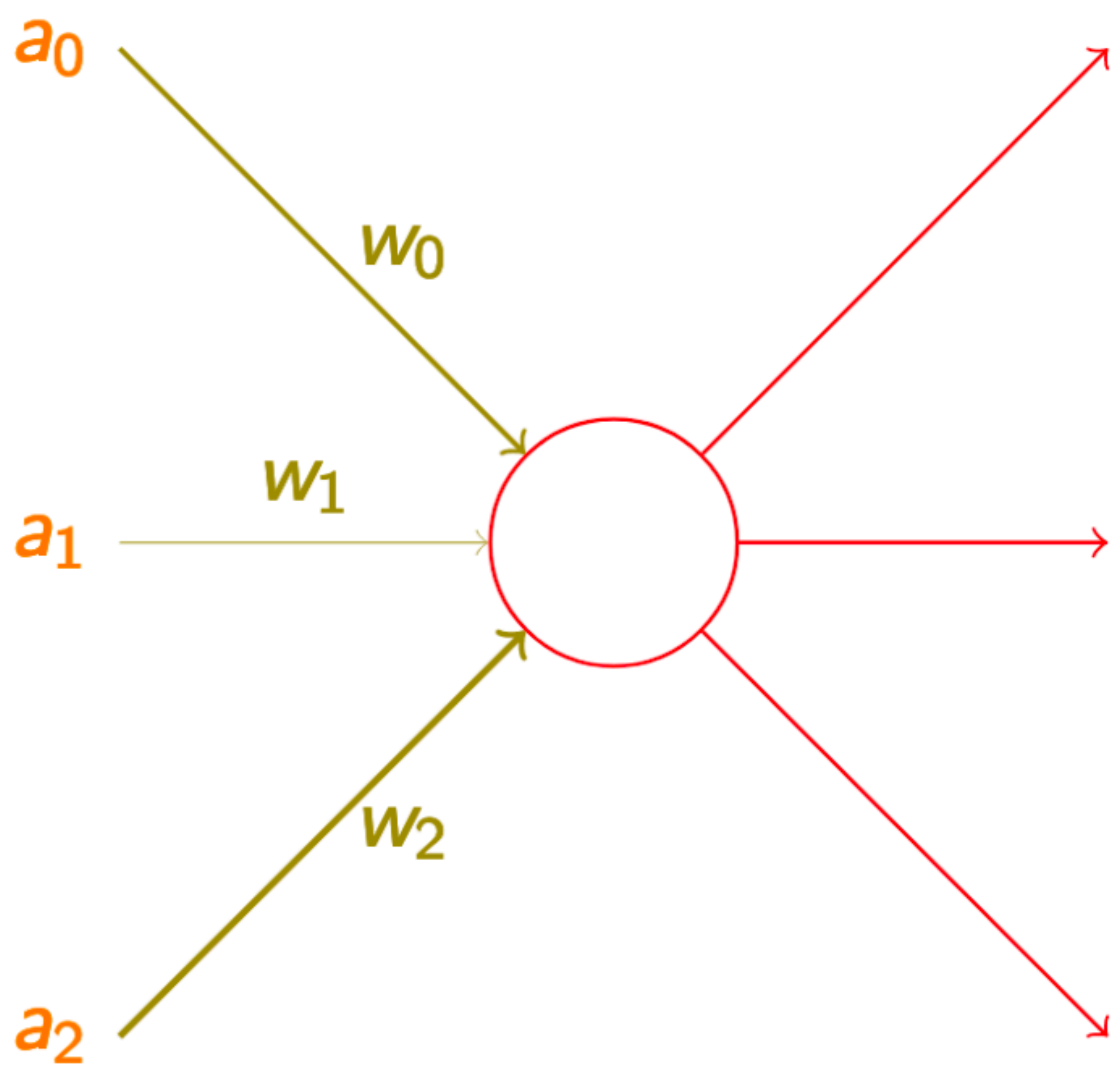
Fall 2024

Plan for Today

- Last time:
  - Computational graph abstraction
  - Backpropagation algorithm
- Today: intro to **feed-forward neural networks**
  - Basic computation + expressive power
  - Multilayer perceptrons
  - Mini-batches
  - Hyper-parameters and regularization

Computation: Basic Example

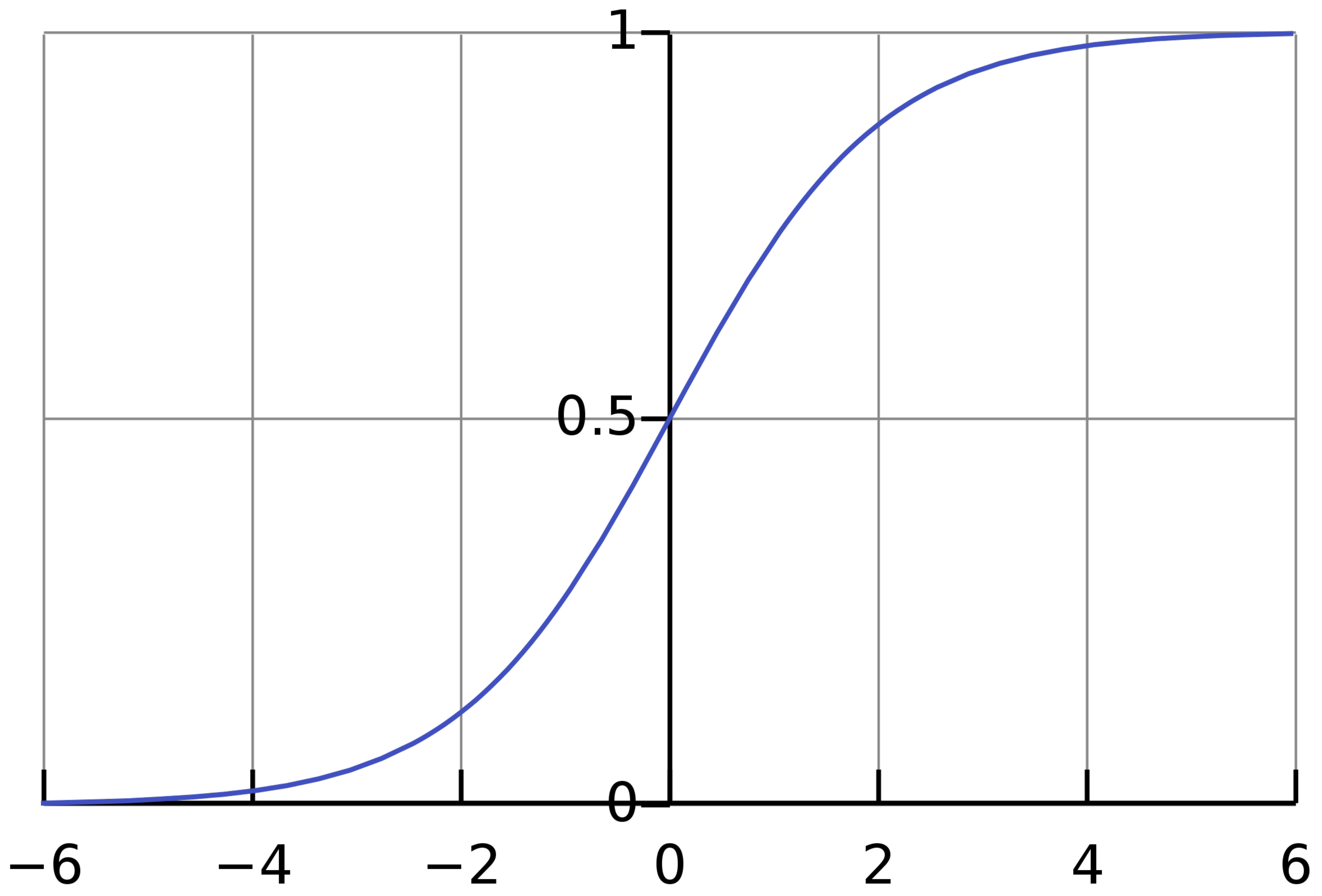
Artificial Neuron



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

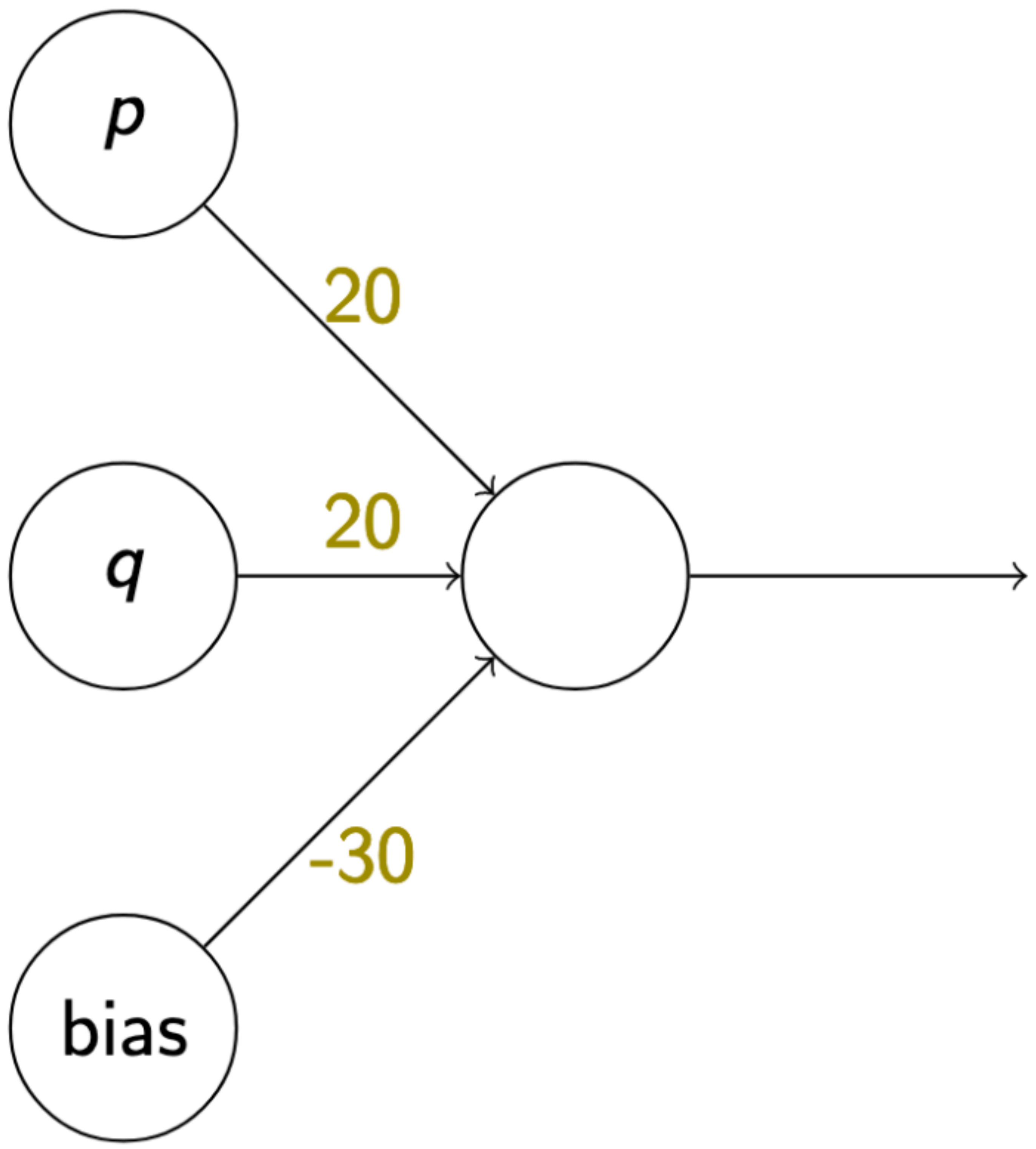
https://github.com/shanest/nn-tutorial

Activation Function: Sigmoid



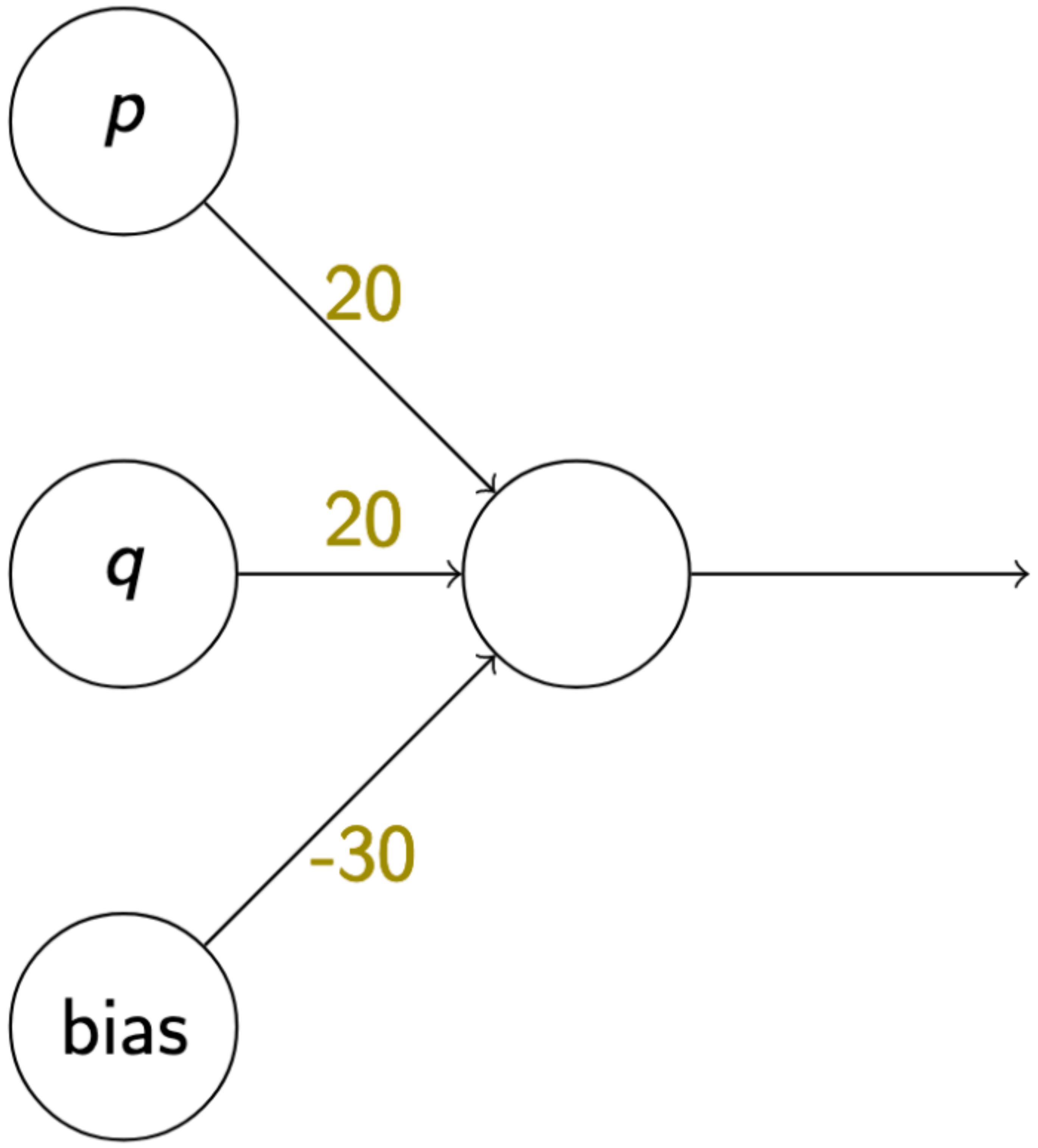
$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

Computing a Boolean function



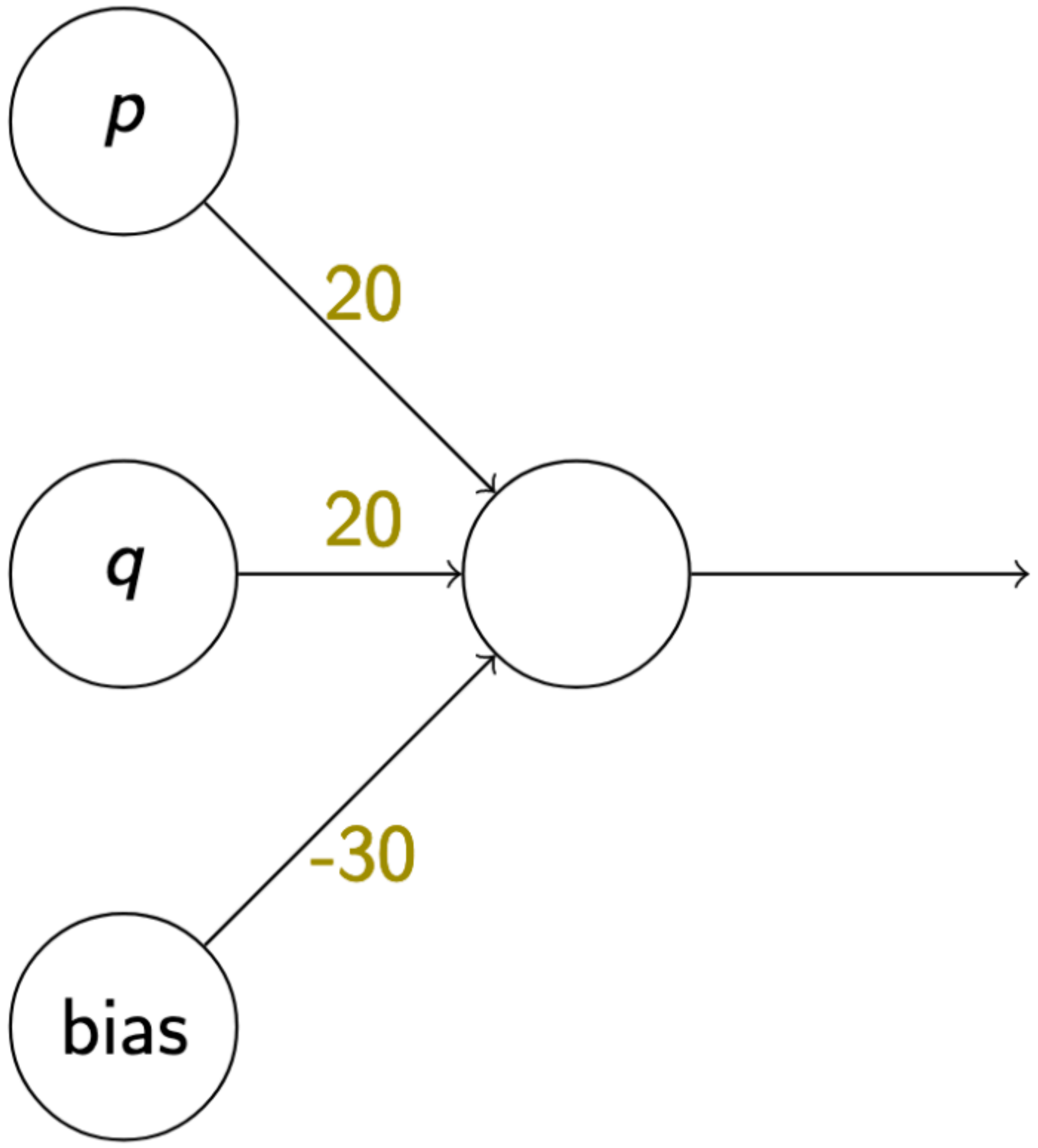
Computing a Boolean function

p	q	a



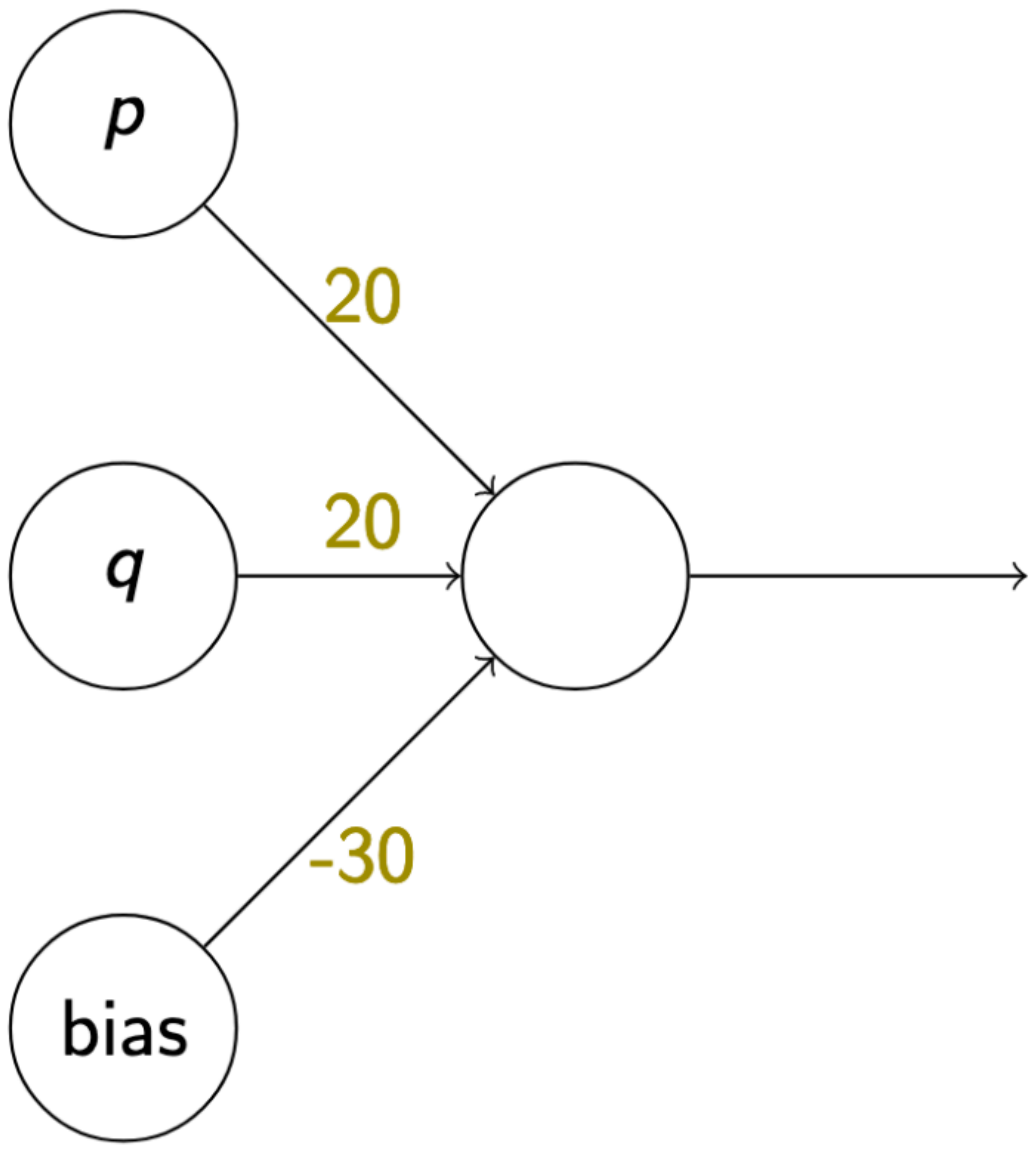
Computing a Boolean function

p	q	a
1	1	1



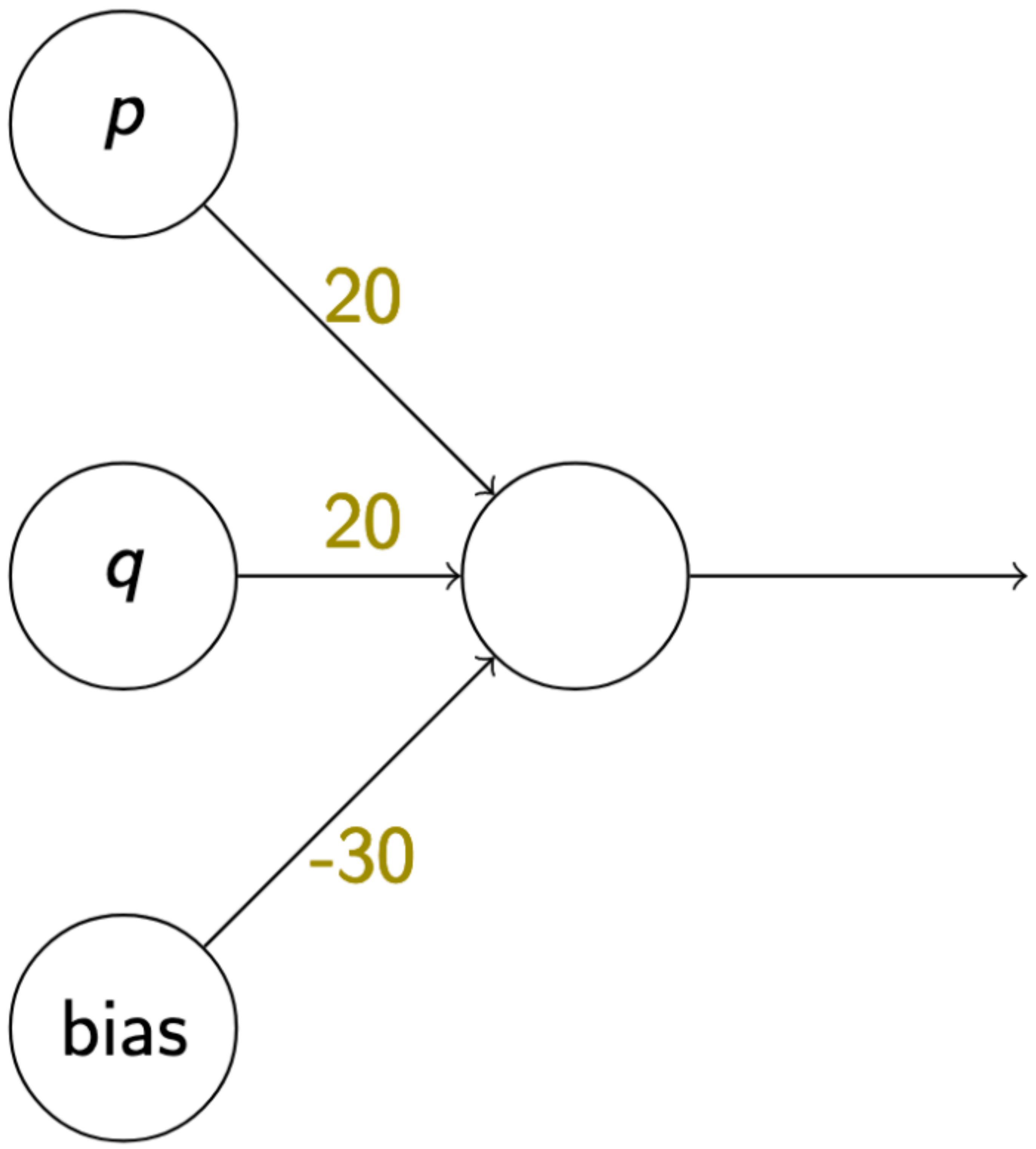
Computing a Boolean function

p	q	a
1	1	1
1	0	0



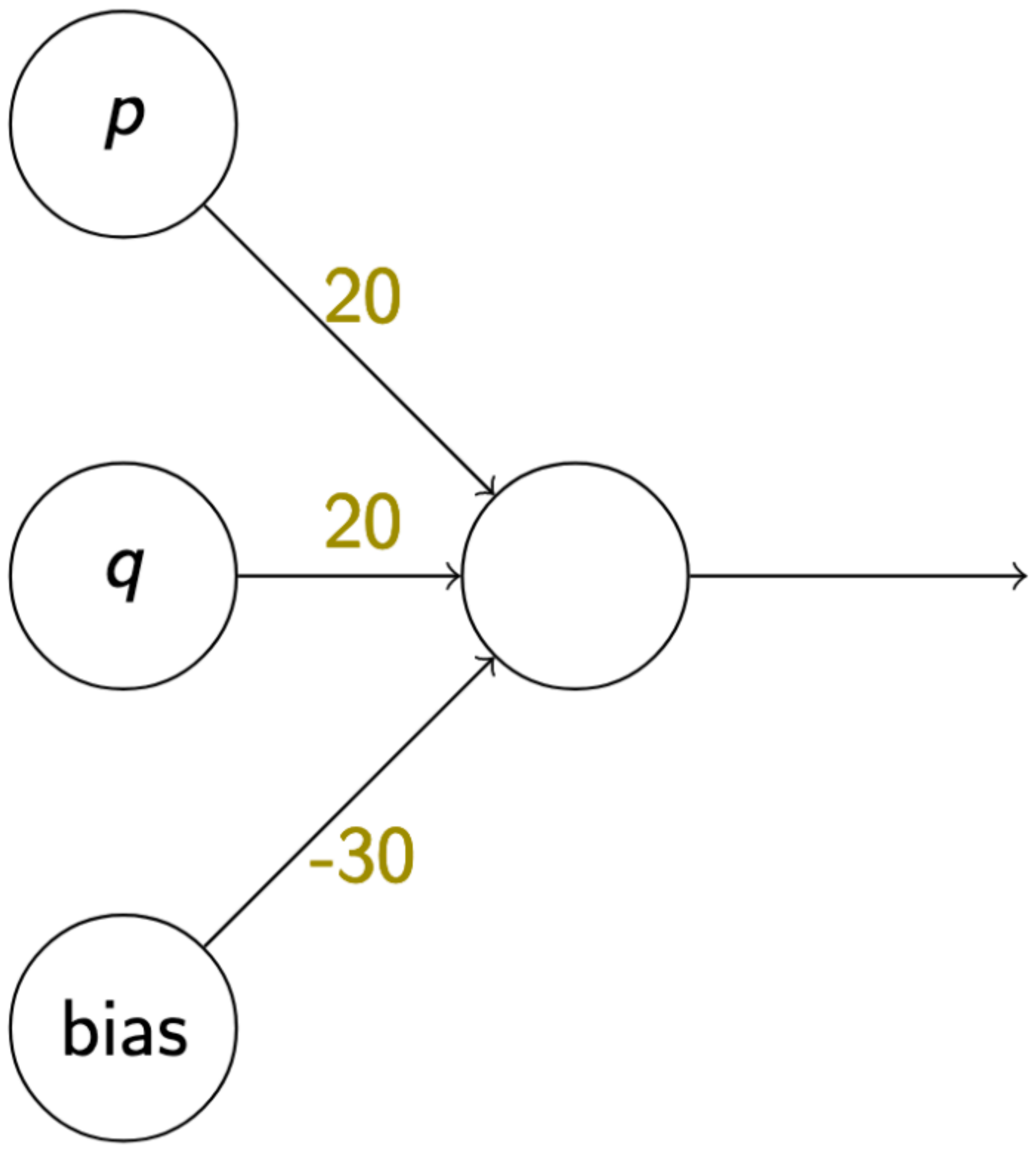
Computing a Boolean function

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

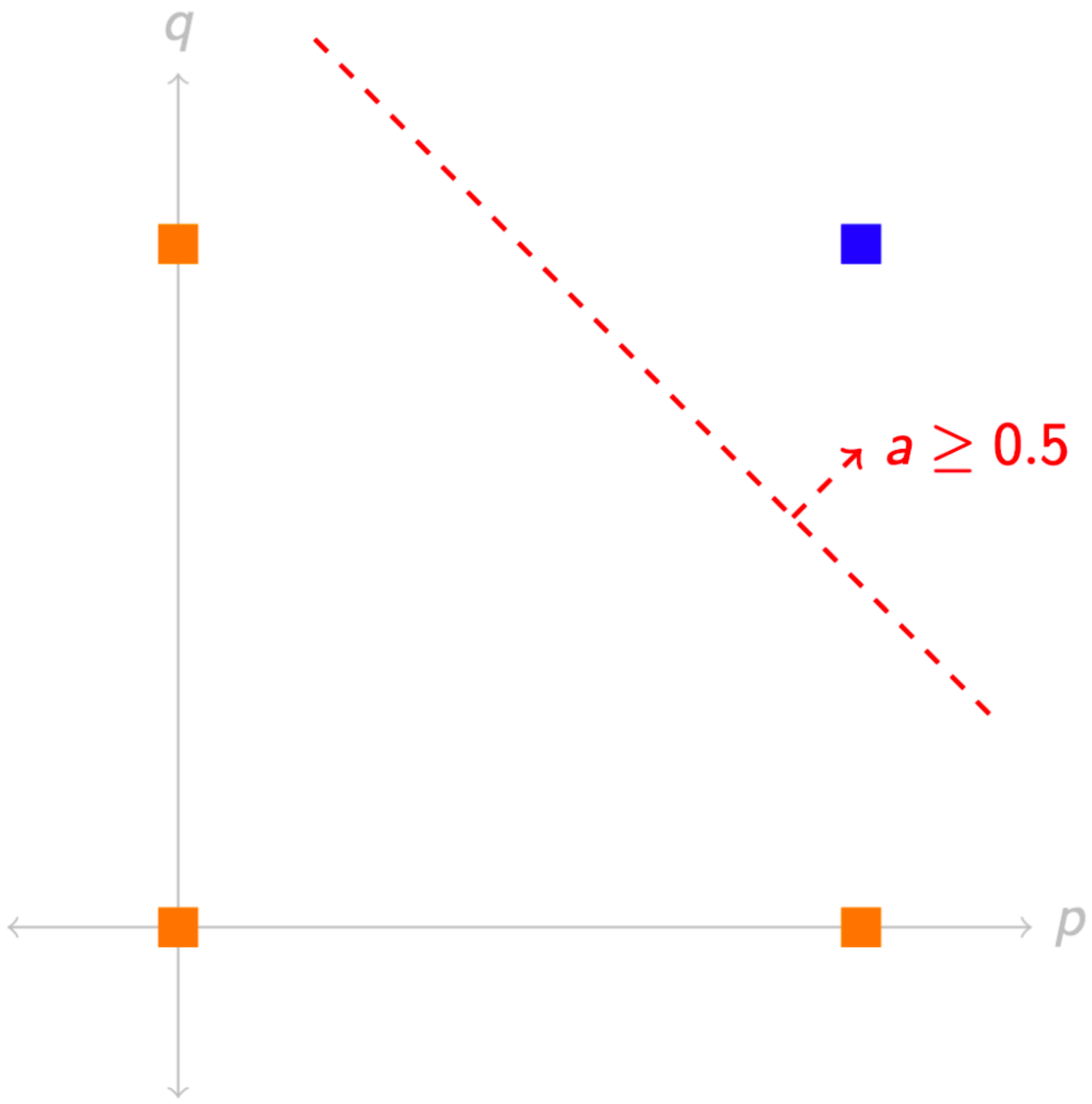


Computing a Boolean function

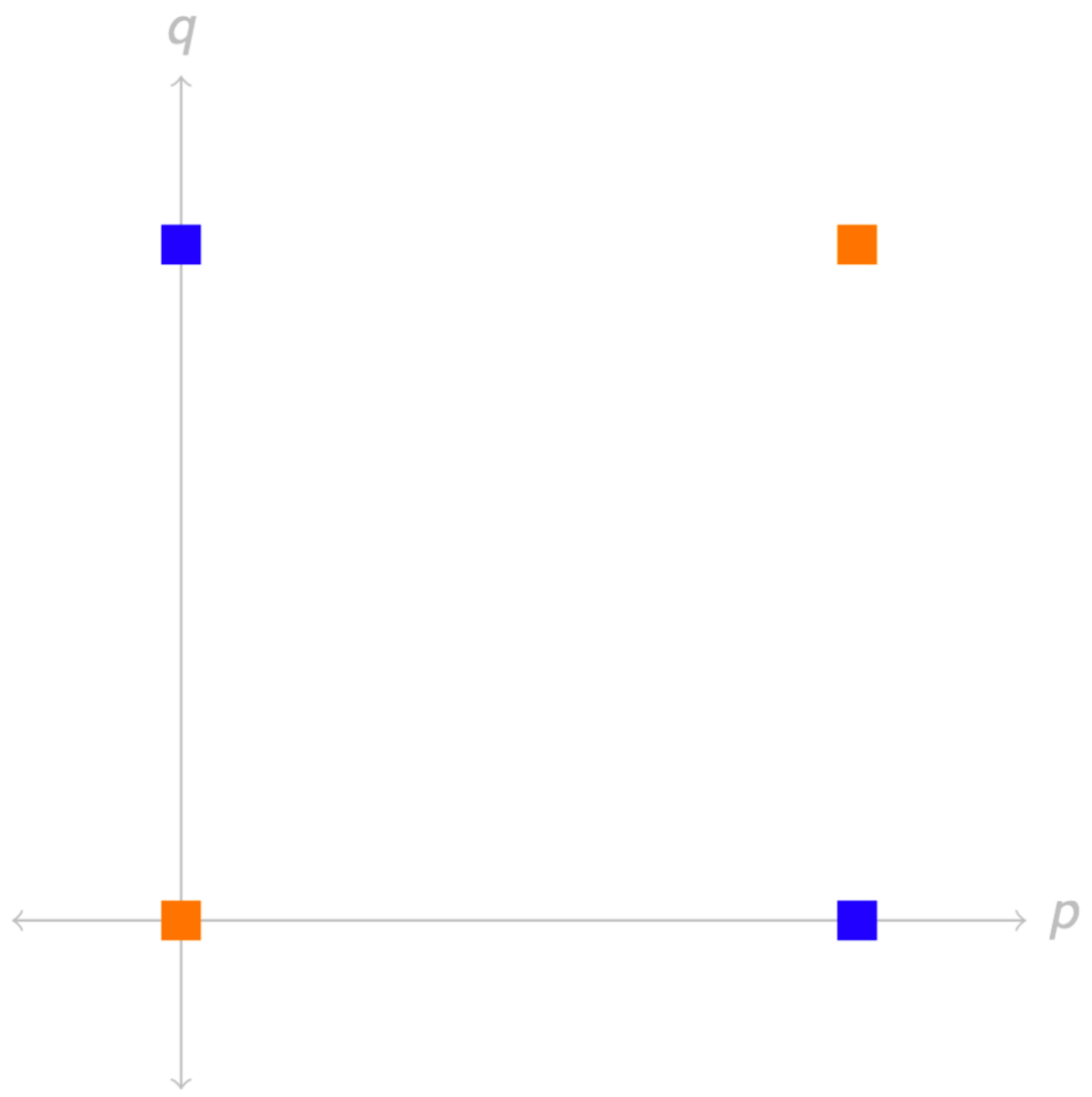
p	q	a
1	1	1
1	0	0
0	1	0
0	0	0



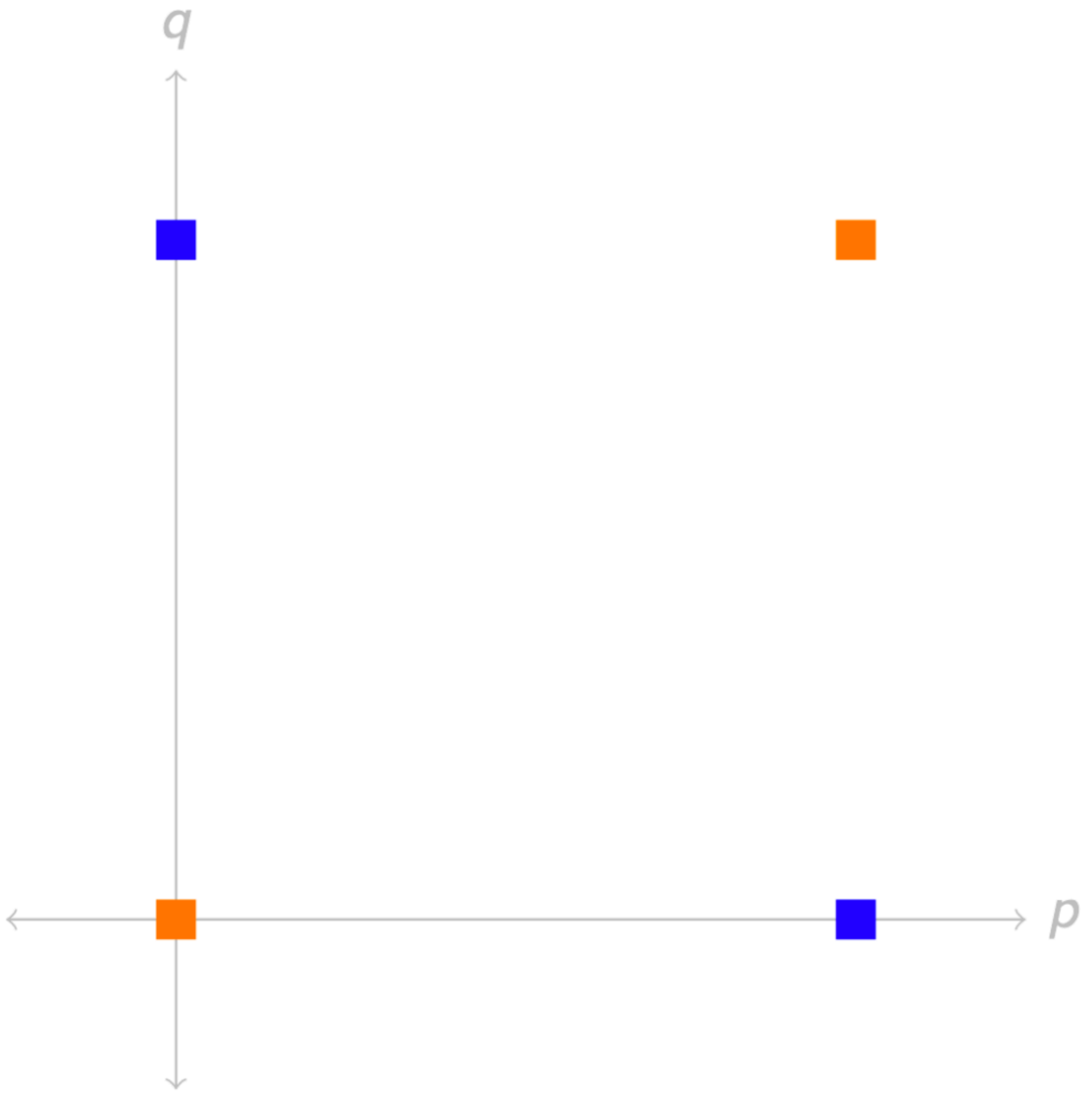
Computing 'and'



The XOR problem

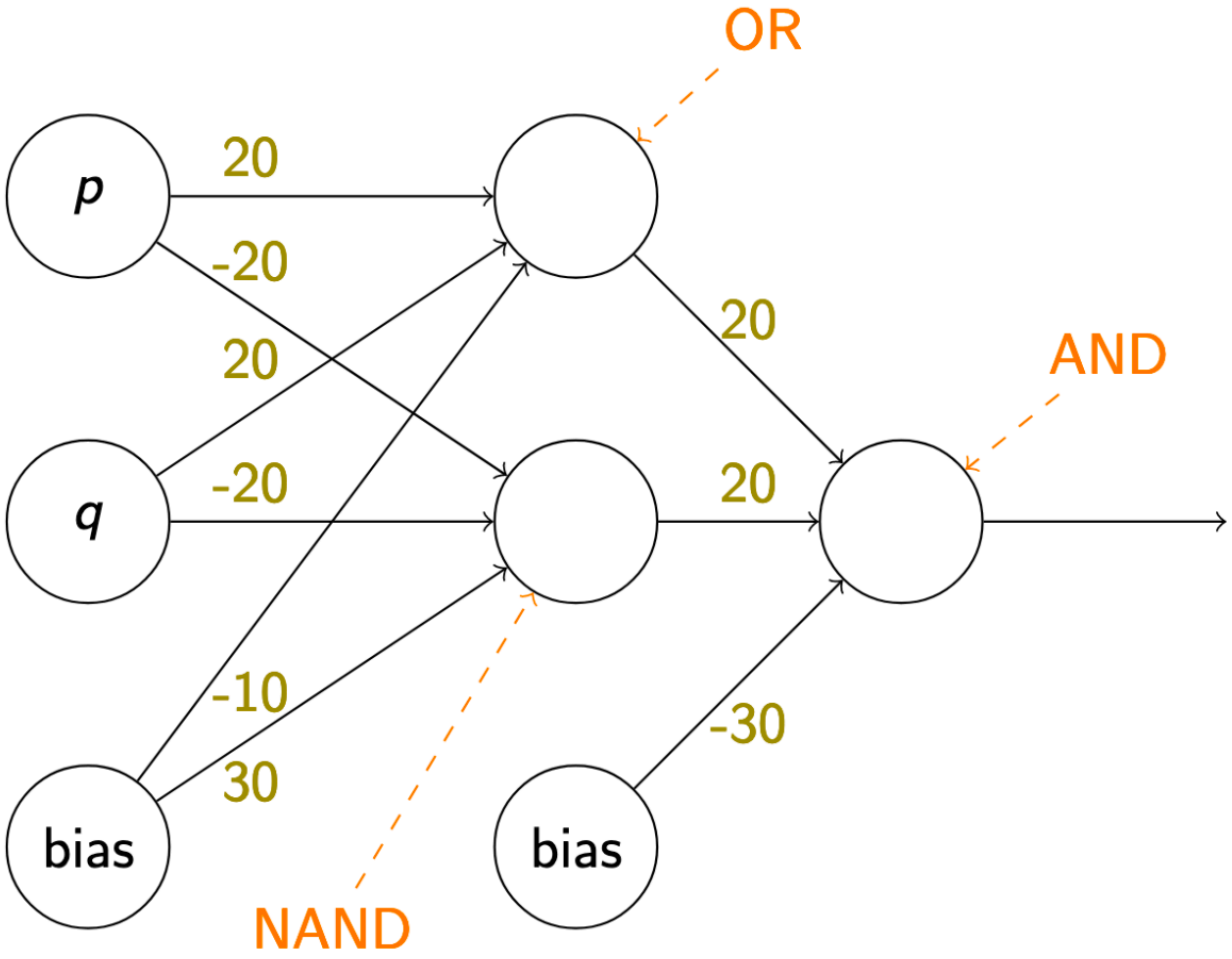


The XOR problem

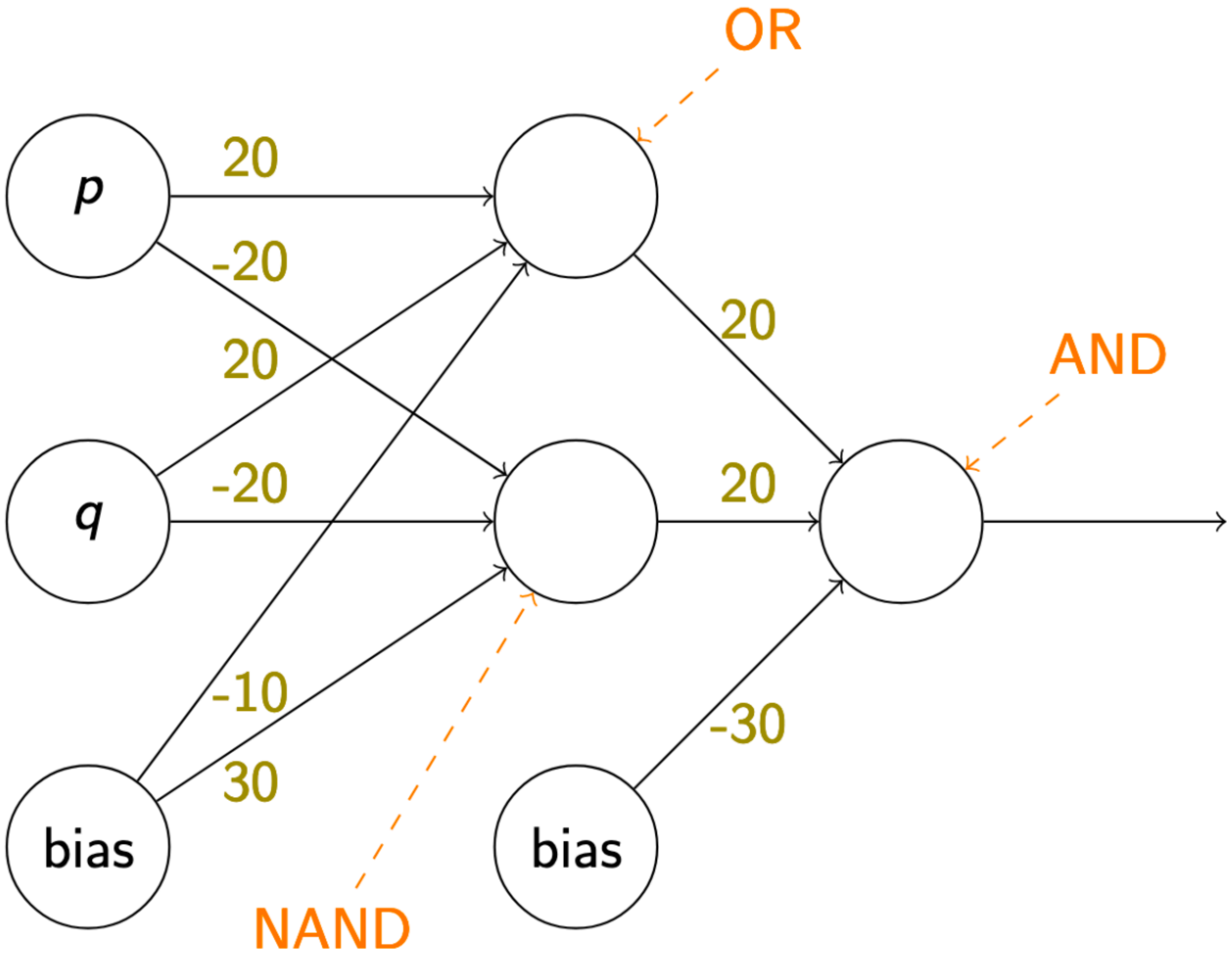


XOR is not linearly separable

Computing XOR

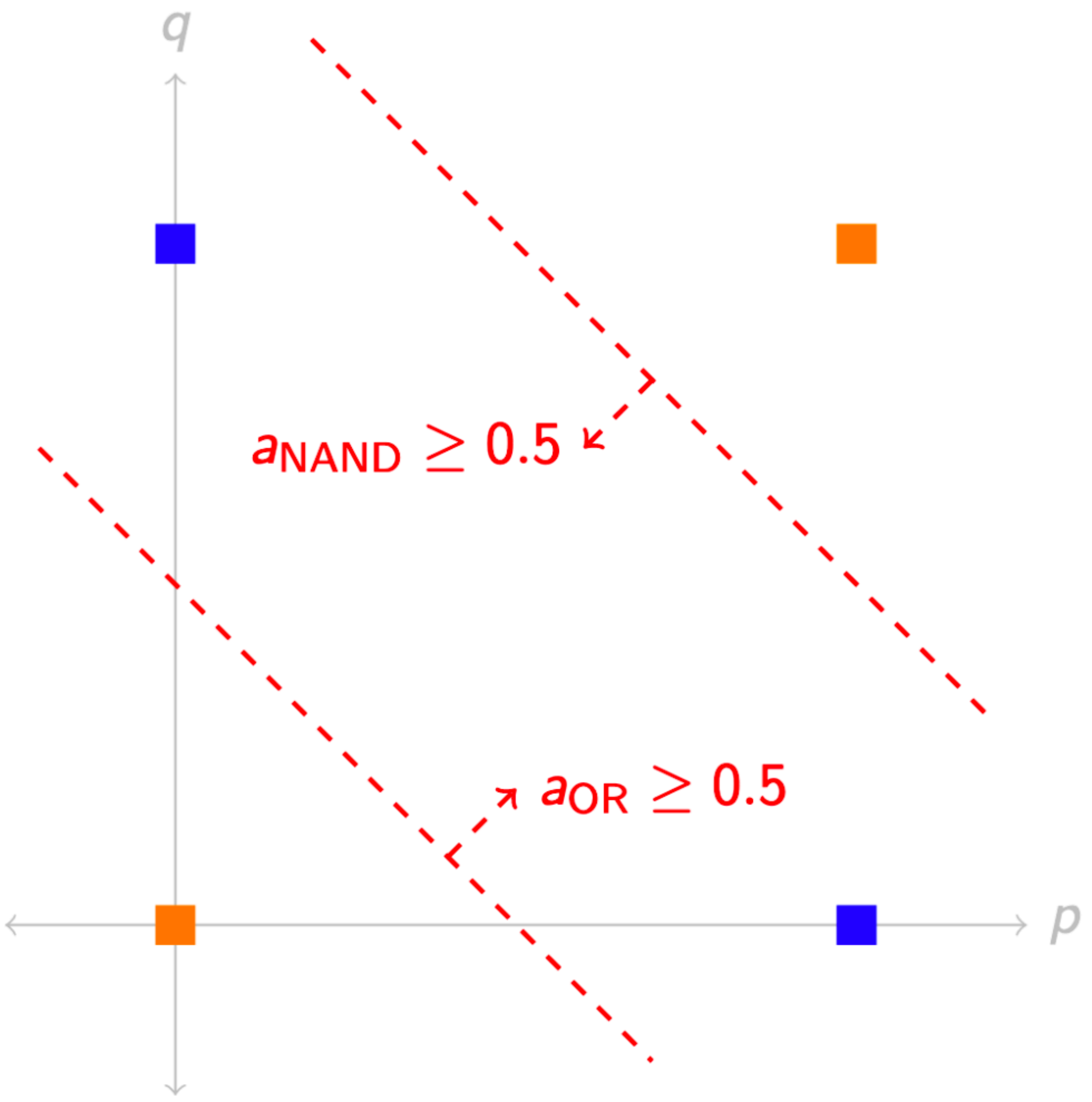


Computing XOR



Exercise: show that
NAND behaves as described.

Computing XOR



Key Ideas

- **Hidden layers**: intermediate layers of representation that are **not directly used** as outputs
  - 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
- Adding hidden layers **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 $|f(x) - g(x)| < \epsilon$ for all $x \in [0,1]^m$.

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 $|f(x) - g(x)| < \epsilon$ for all $x \in [0,1]^m$.
- 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 $|f(x) - g(x)| < \epsilon$ for all $x \in [0,1]^m$.
- Generalizations (diff activation functions, less bounded, etc.) exist.
- But:
  - Size of the hidden layer is must be **exponential in *m***
  - How does one find/learn such a good approximation?

Expressive Power

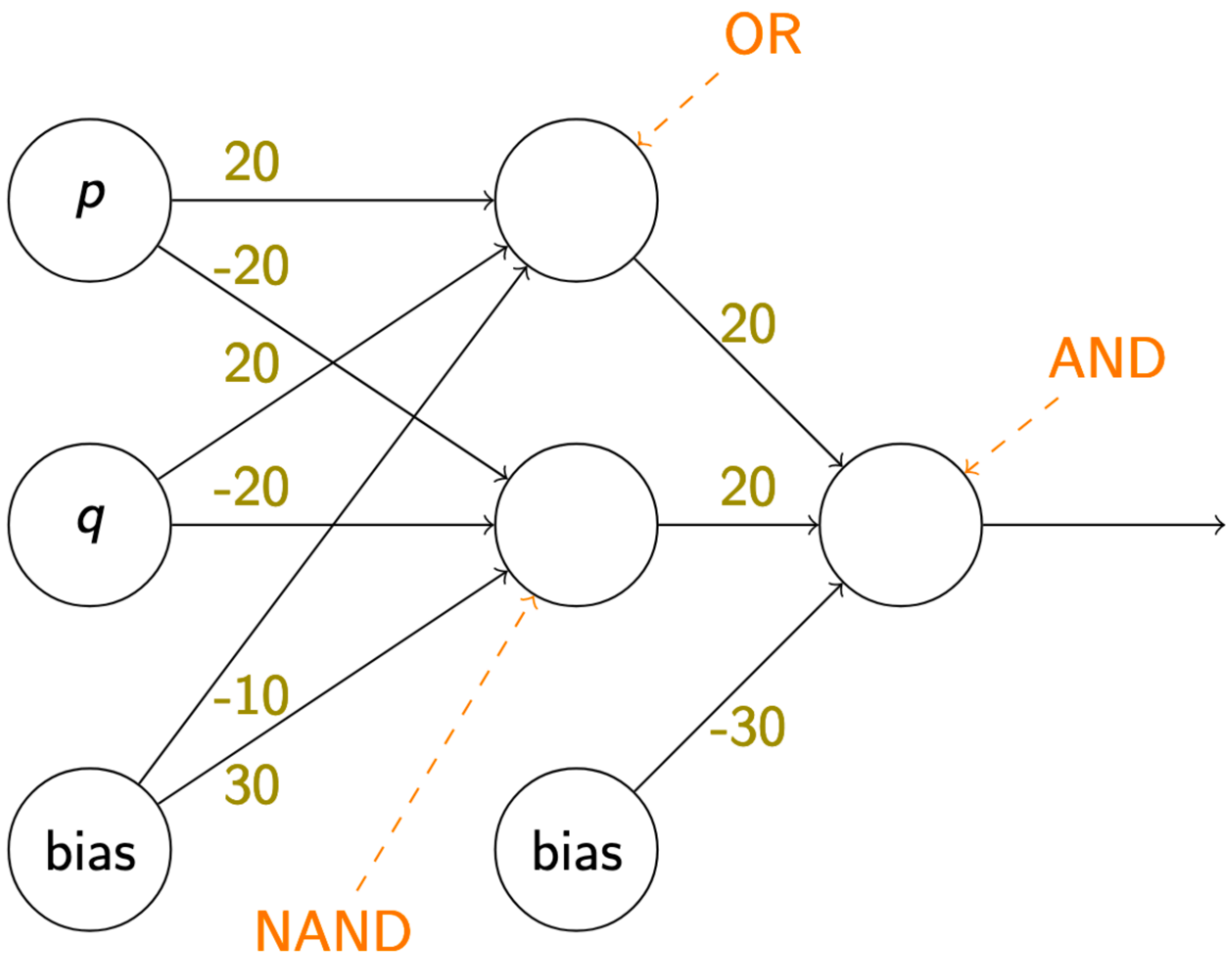
- Neural networks with **one hidden layer** are **universal function approximators**
- Let $f : [0,1]^m \to \mathbb{R}$ be continuous and $\epsilon > 0$. Then there is a one-hidden-layer neural network g with sigmoid activation such that $|f(x) - g(x)| < \epsilon$ for all $x \in [0,1]^m$.
- Generalizations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

Expressive Power

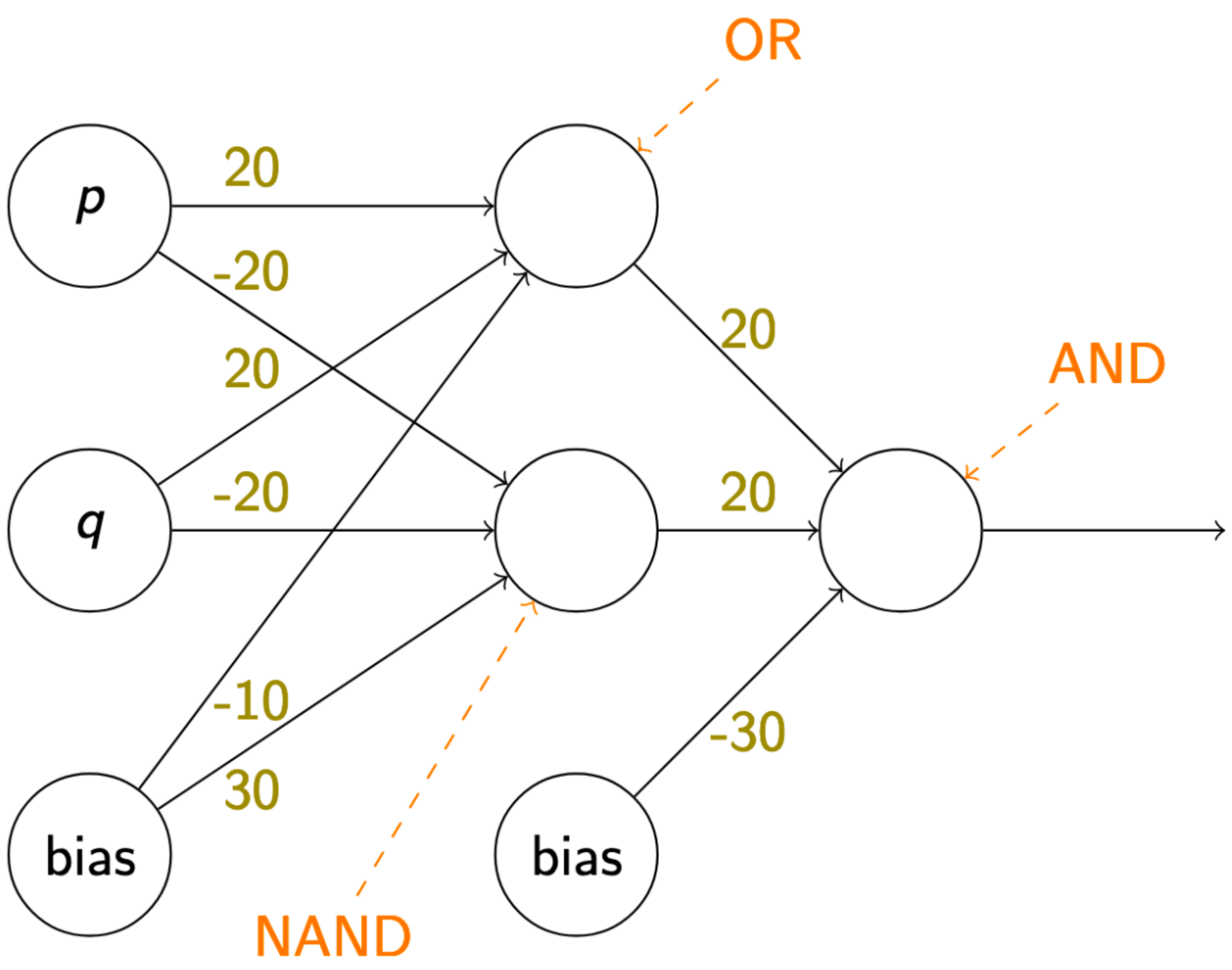
- Neural networks with **one hidden layer** are **universal function approximators**
- Let $f : [0,1]^m \to \mathbb{R}$ be continuous and $\epsilon > 0$. Then there is a one-hidden-layer neural network g with sigmoid activation such that $|f(x) - g(x)| < \epsilon$ for all $x \in [0,1]^m$.
- Generalizations (diff activation functions, less bounded, etc.) exist.
- But:
  - Size of the hidden layer is must be **exponential in *m***
  - How does one find/learn such a good approximation?
- Nice walkthrough: http://neuralnetworksanddeeplearning.com/chap4.html
- See also GBC 6.4.1 for more references, generalizations, discussion

Feed-forward networks
aka Multi-Layer Perceptrons (MLP)

XOR Network

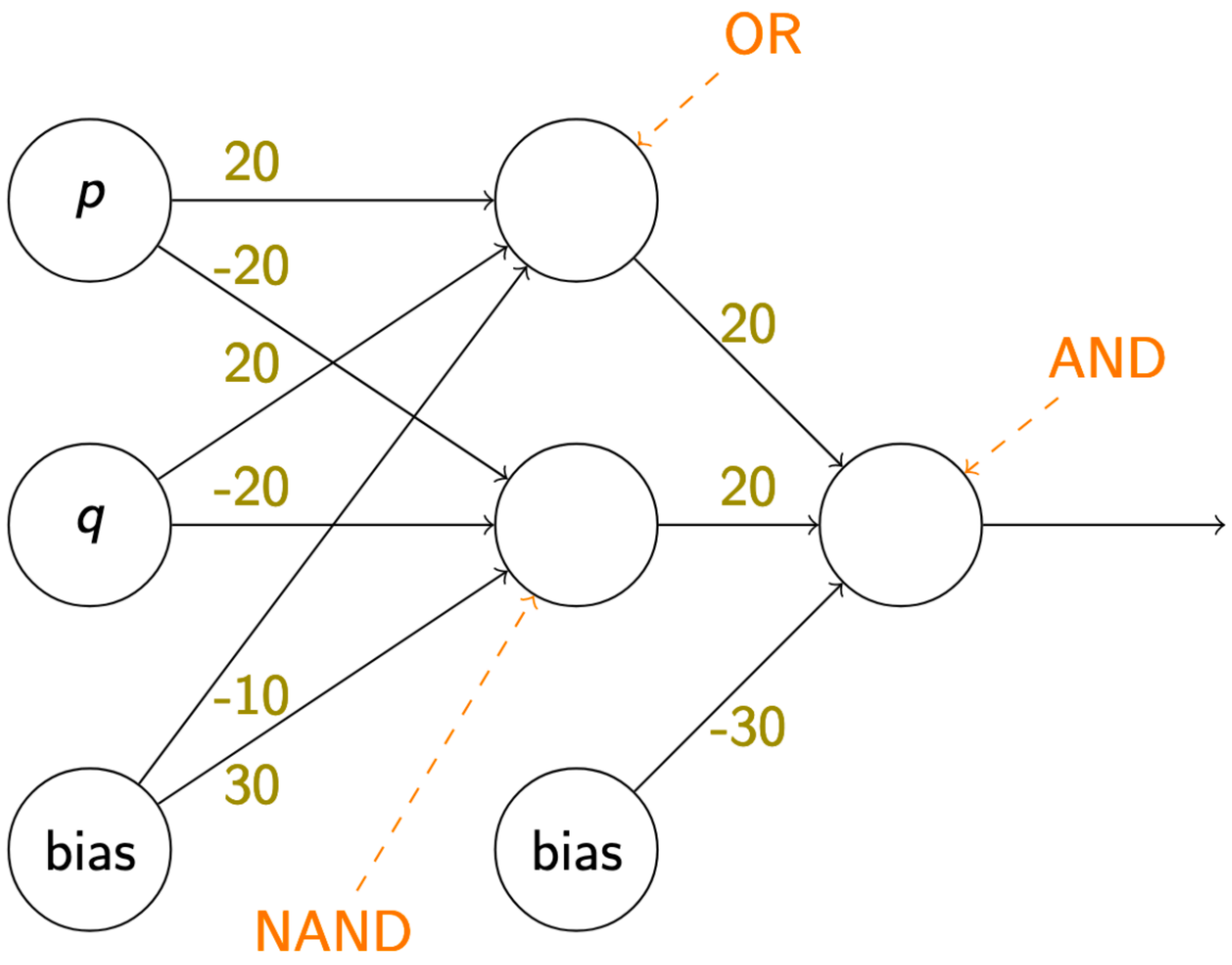


XOR Network



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

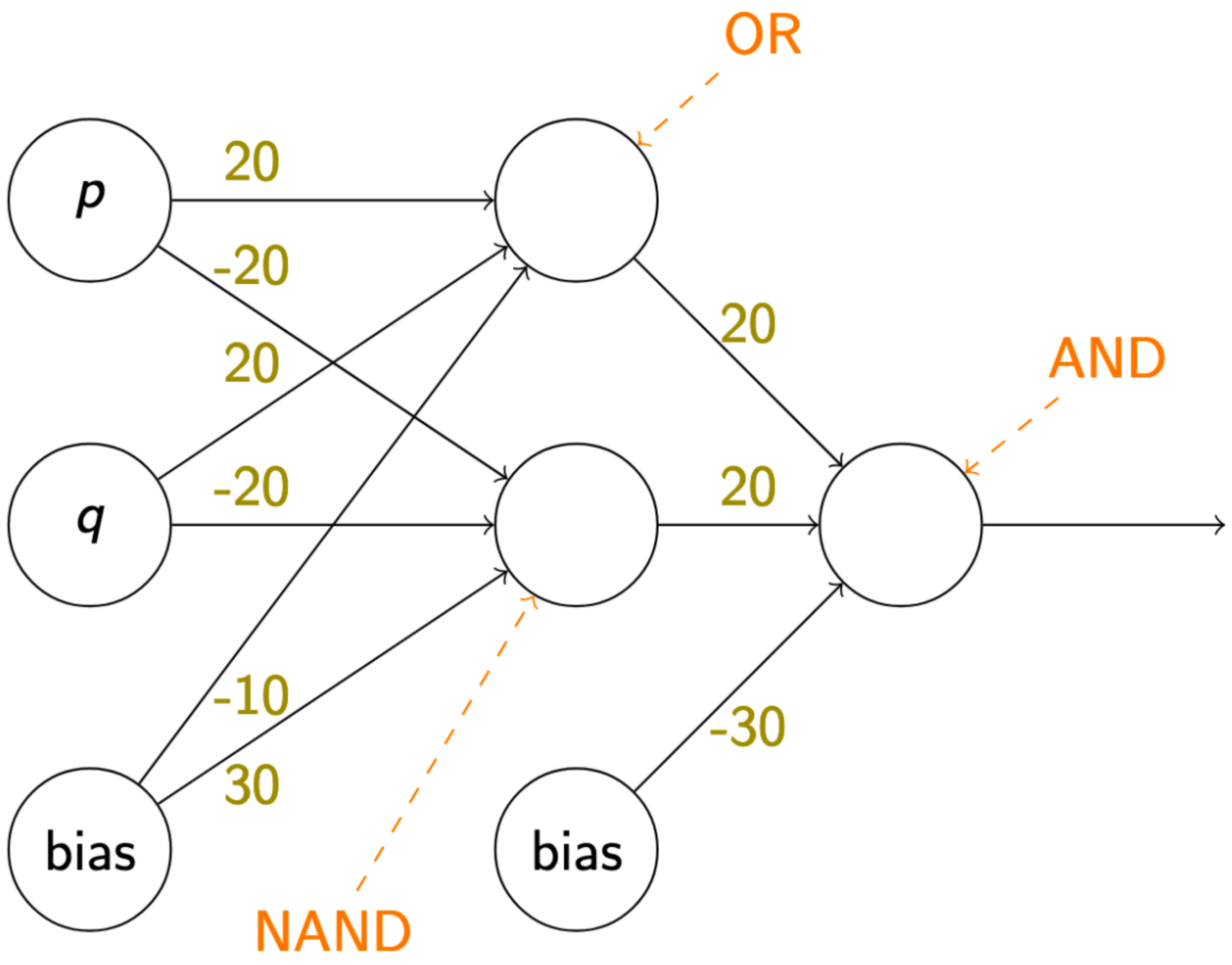
XOR Network



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

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

XOR Network

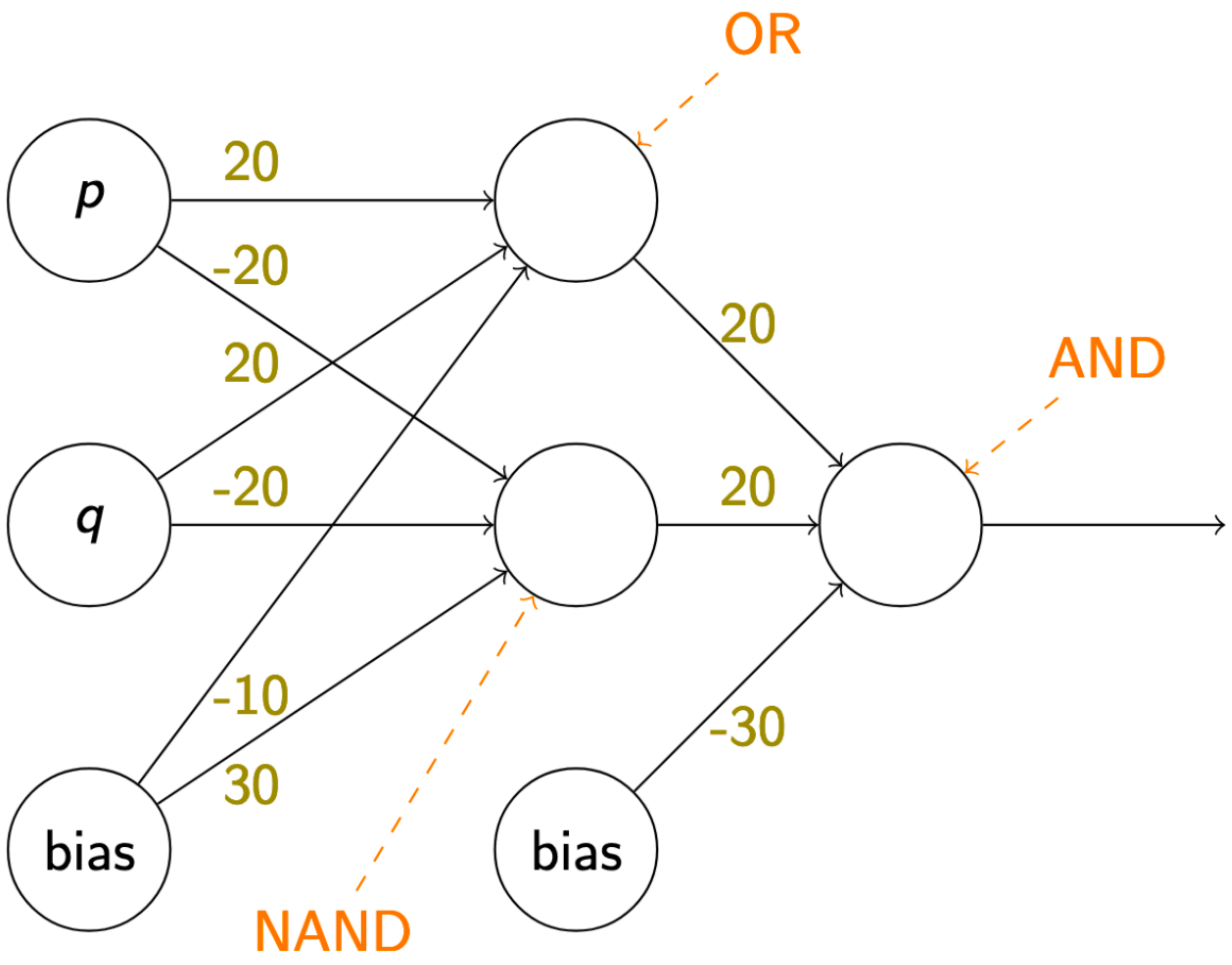


$$a_{\text{or}} = \sigma\left(\begin{bmatrix} w_p^{\text{or}} & w_q^{\text{or}} \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix} + b^{\text{or}}\right)$$

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

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

XOR Network

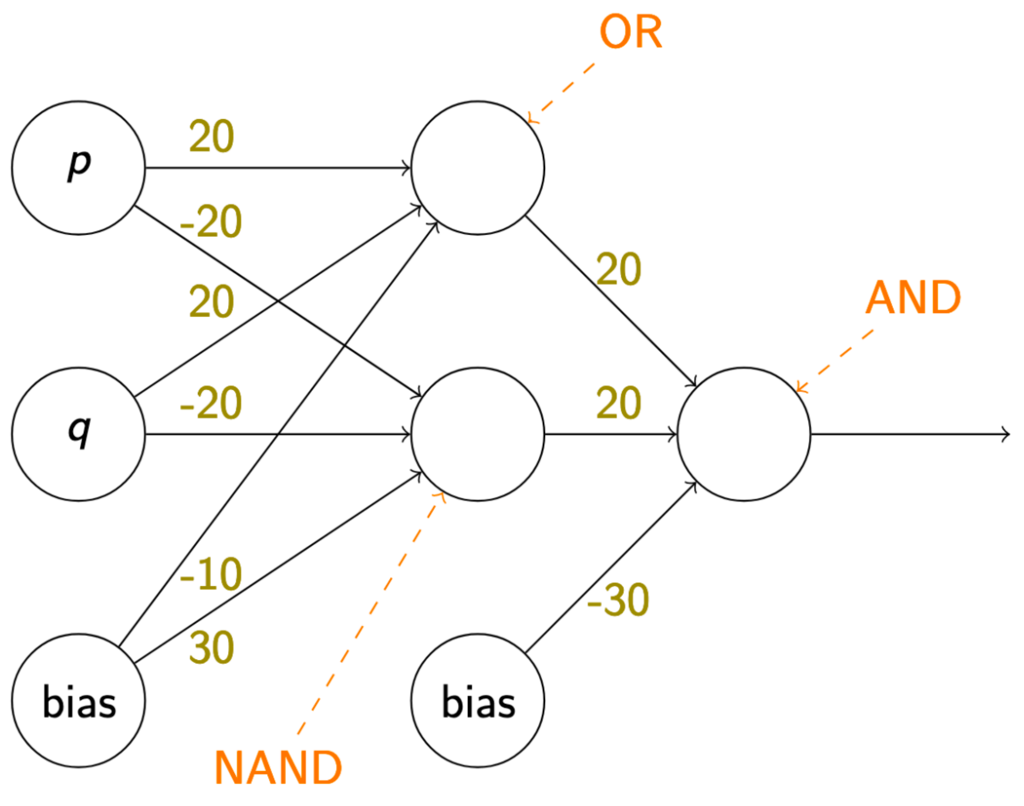


$$a_{\text{or}} = \sigma\left(\begin{bmatrix} w_p^{\text{or}} & w_q^{\text{or}} \end{bmatrix}\begin{bmatrix} p \\ q \end{bmatrix} + b^{\text{or}}\right)$$

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

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

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



XOR Network

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

Generalizing

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

Generalizing

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

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

Generalizing

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

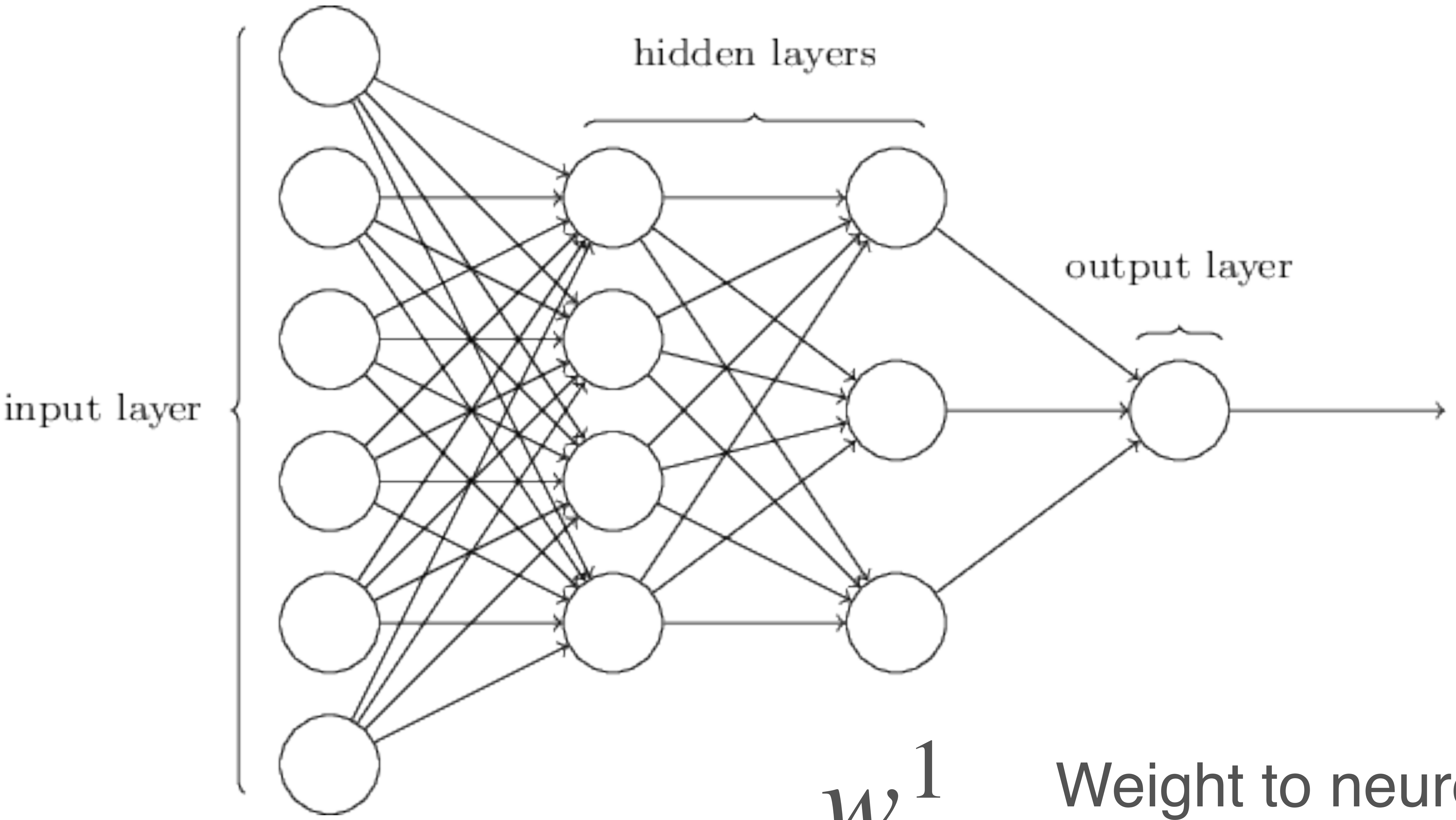
$$\hat{y} = f_2\left(W^2 \cdot f_1\left(W^1 x + b^1\right) + b^2\right)$$

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

Some terminology

- Our XOR network is a **feed-forward** neural network with **one hidden layer**
  - Aka a multi-layer perceptron (MLP)
- 2 input nodes
- 1 output node
- 1 hidden layer with 2 neurons
- Sigmoid activation function

General MLP



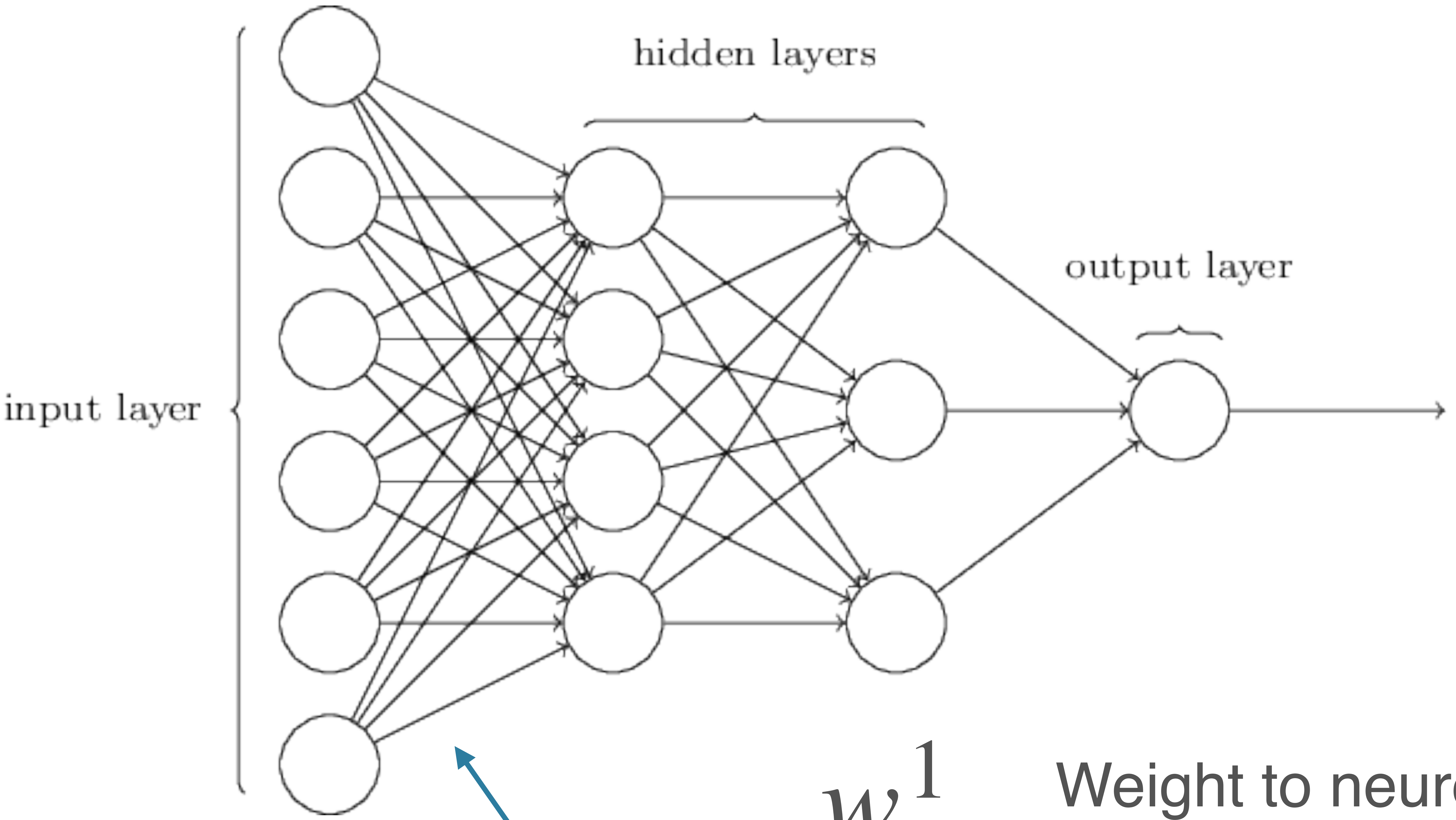
source

$$w_{ji}^{1}$$

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

General MLP

source



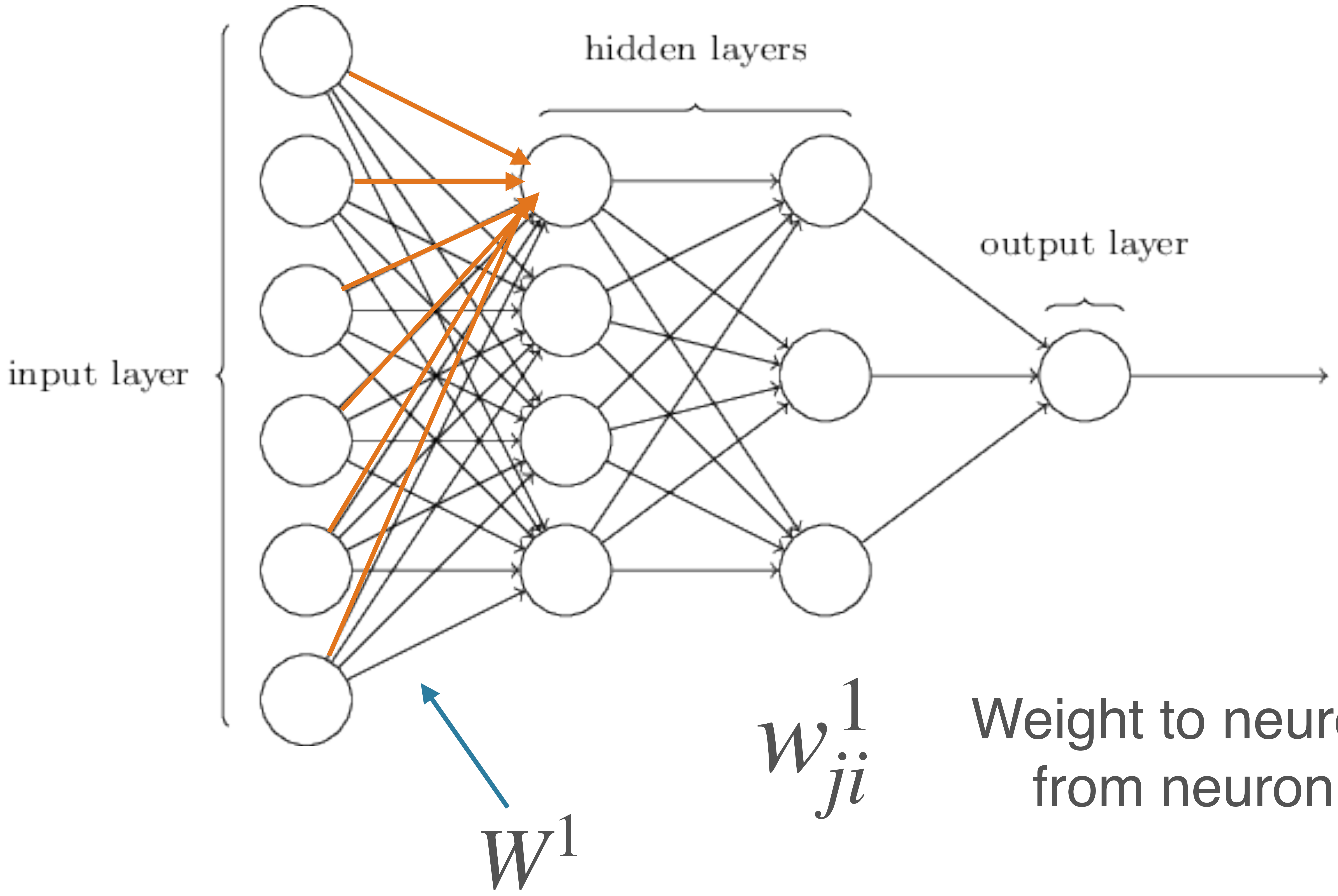
W^1

w_{ji}^1

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

General MLP

source



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

General MLP

General MLP

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

General MLP

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

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

shape: $(n_0, 1)$

General MLP

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

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

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

shape: $(n_0, 1)$

shape: (n_1, n_0)

n_0: dimension of input (layer 0)

n_1: output dimension of layer 1

General MLP

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

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

shape: (n_1, n_0)

n_0: dimension of input (layer 0)

n_1: output dimension of layer 1

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

shape: $(n_0, 1)$

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

shape: $(n_1, 1)$

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 & 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 & 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 & 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**

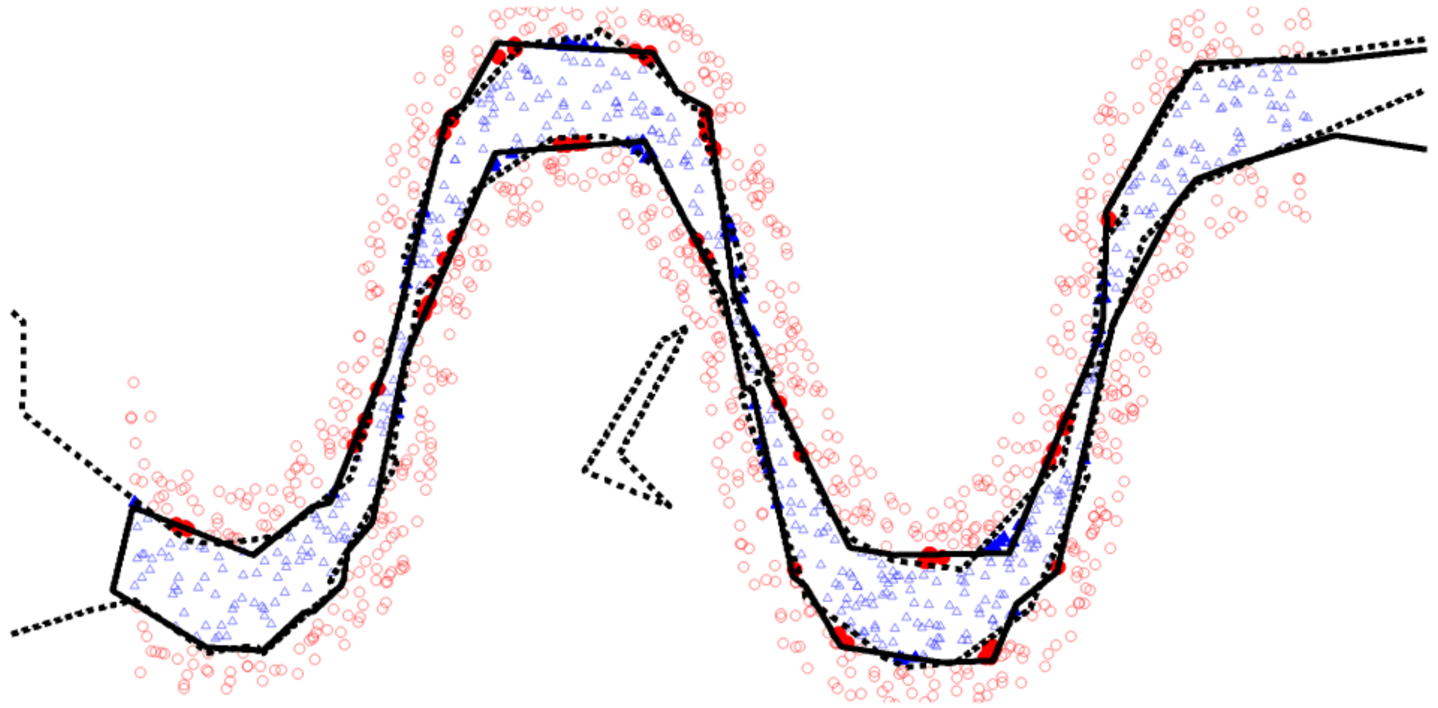
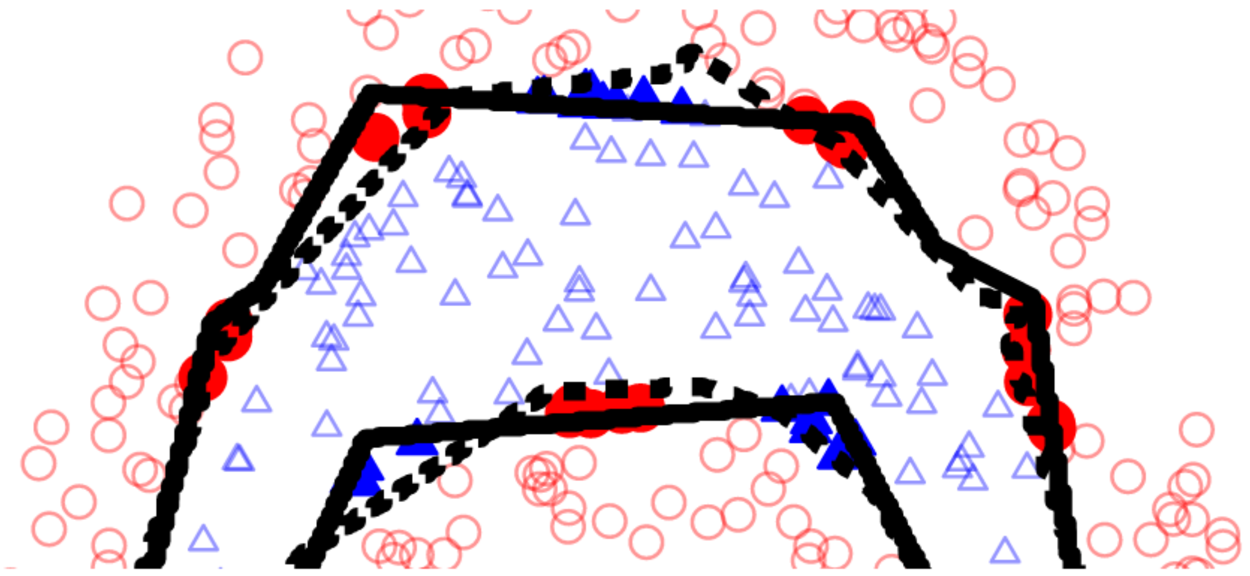
Hyper-parameters of an MLP

- **Input & 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, …

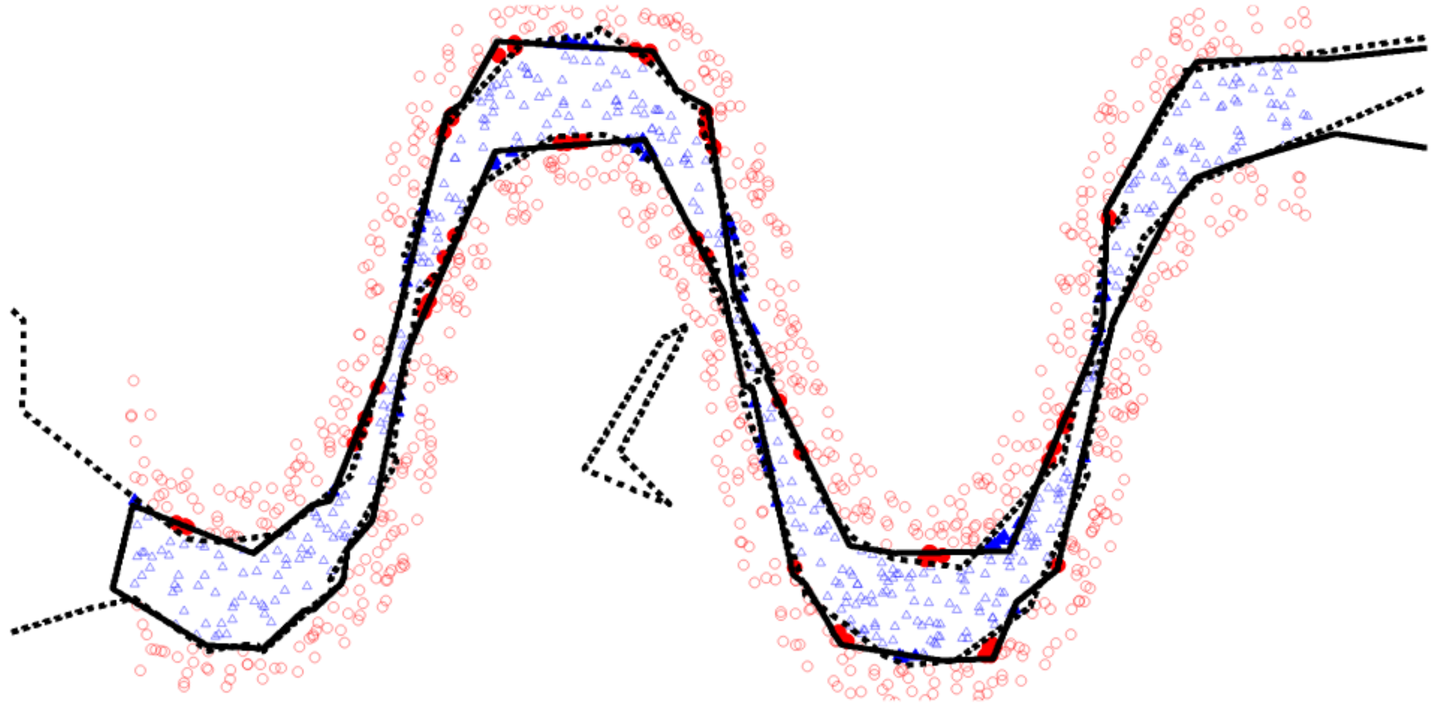
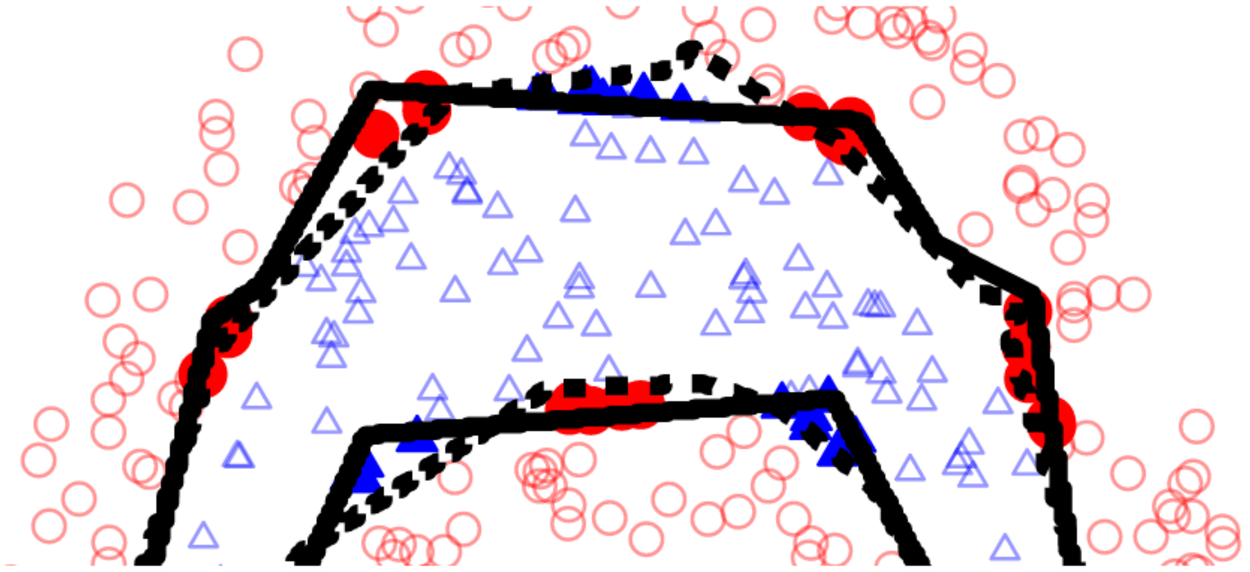
The Deep in Deep Learning

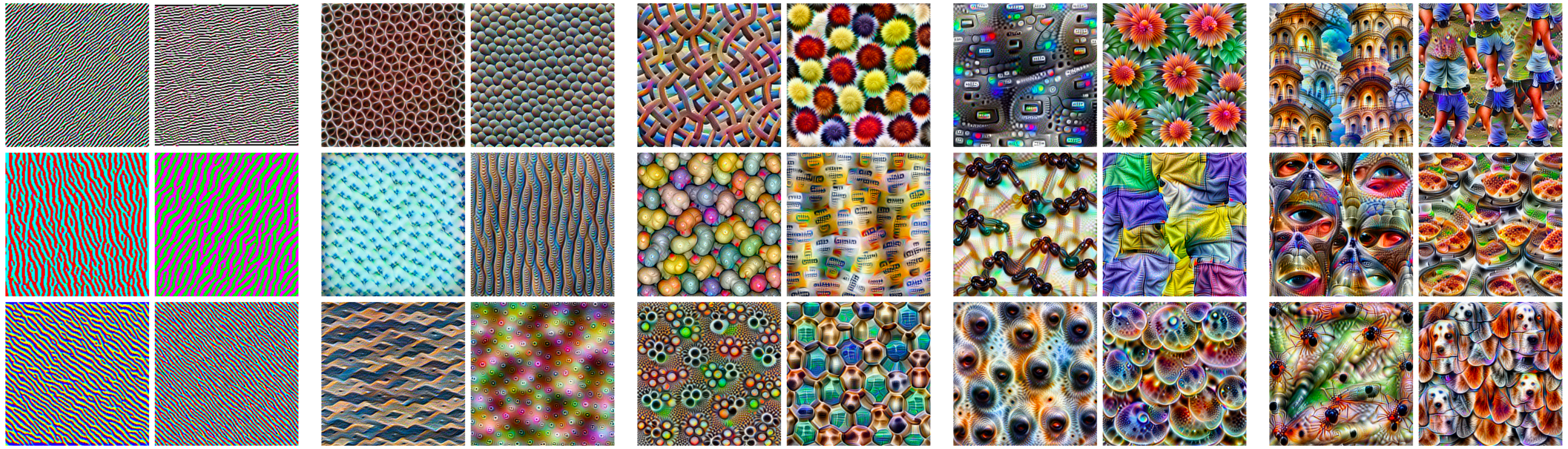
- The Universal Approximation Theorem says that **one hidden layer suffices** for arbitrarily-closely approximating a given function
- Empirical drawbacks: Super-exponentially many neurons; hard to discover
- “Deep and narrow” >> “Shallow and wide” (some theoretical analysis)
  - In principle allows 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 d   discover source
- "Deep and sis)
  - In principle
  - 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 d   discover
  source
- “Deep and … sis)
  - In principle
  - 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

- **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

- **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!**

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

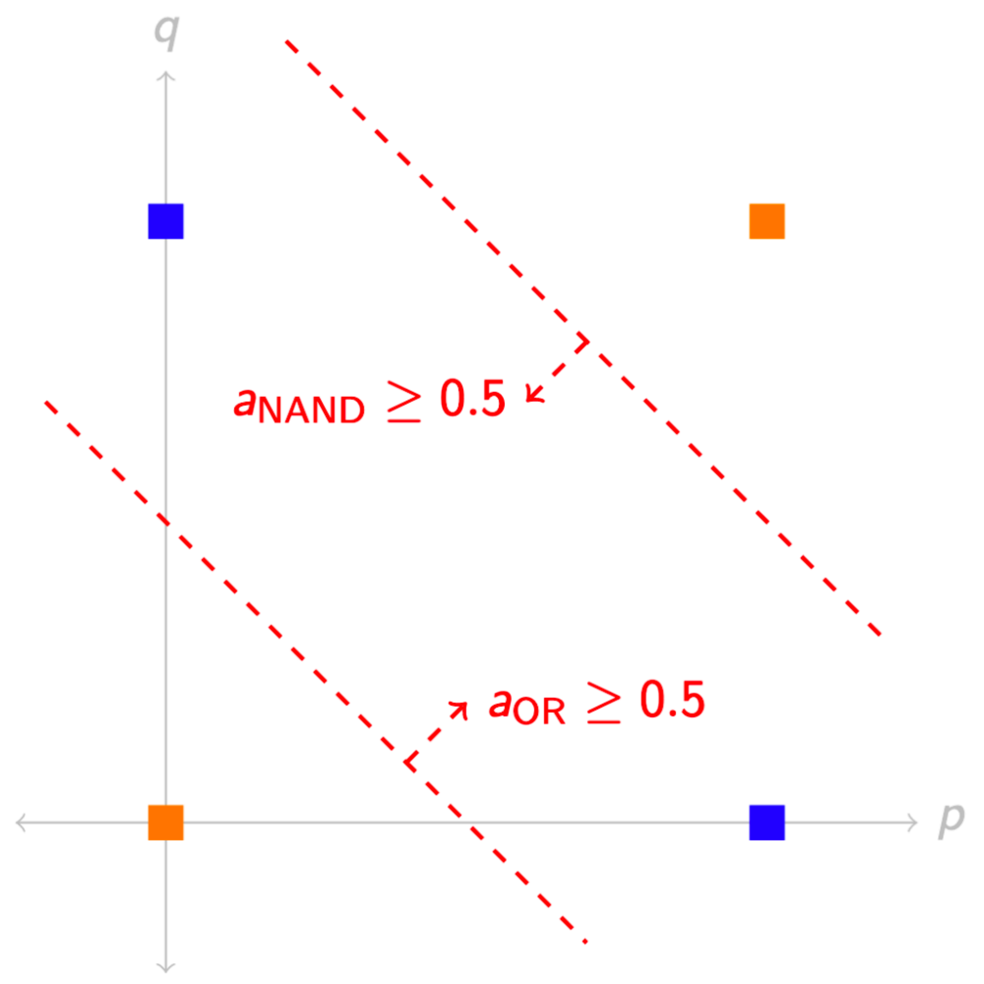
Non-linearity, cont.

Non-linearity, cont.

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

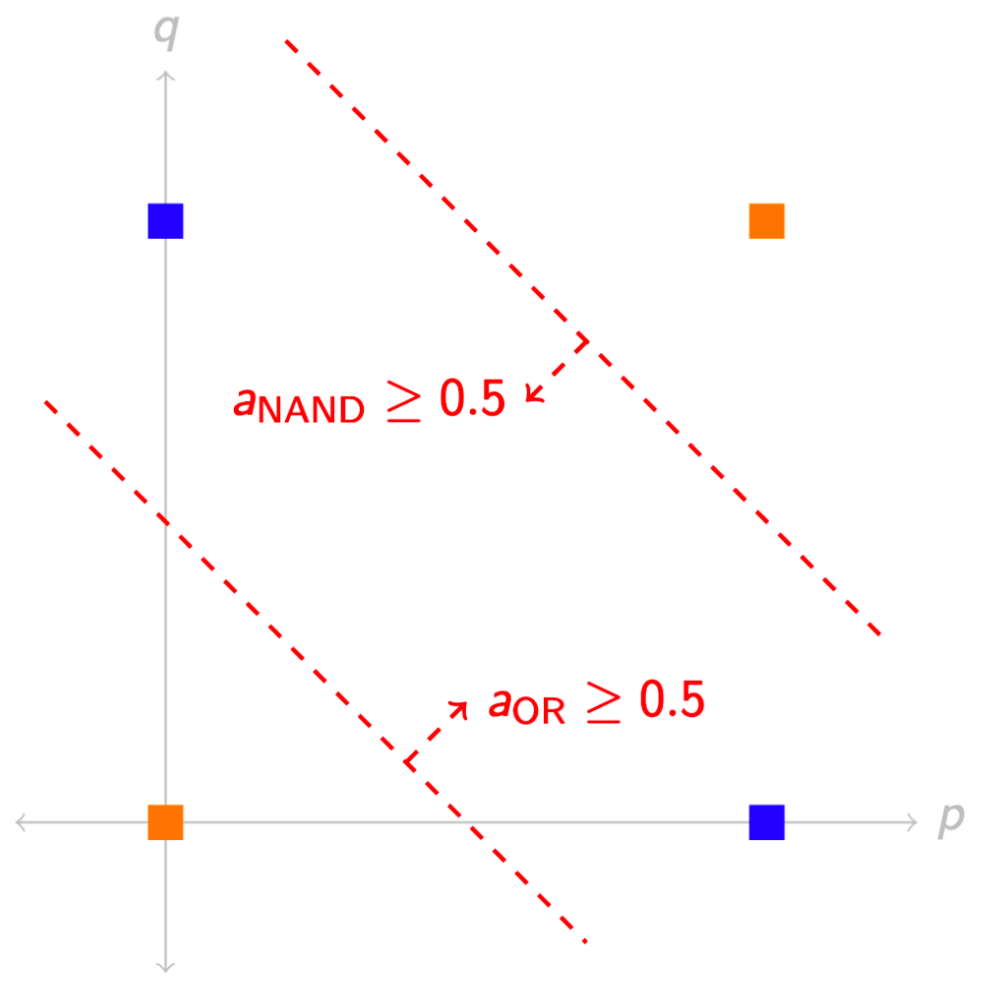
Non-linearity, cont.

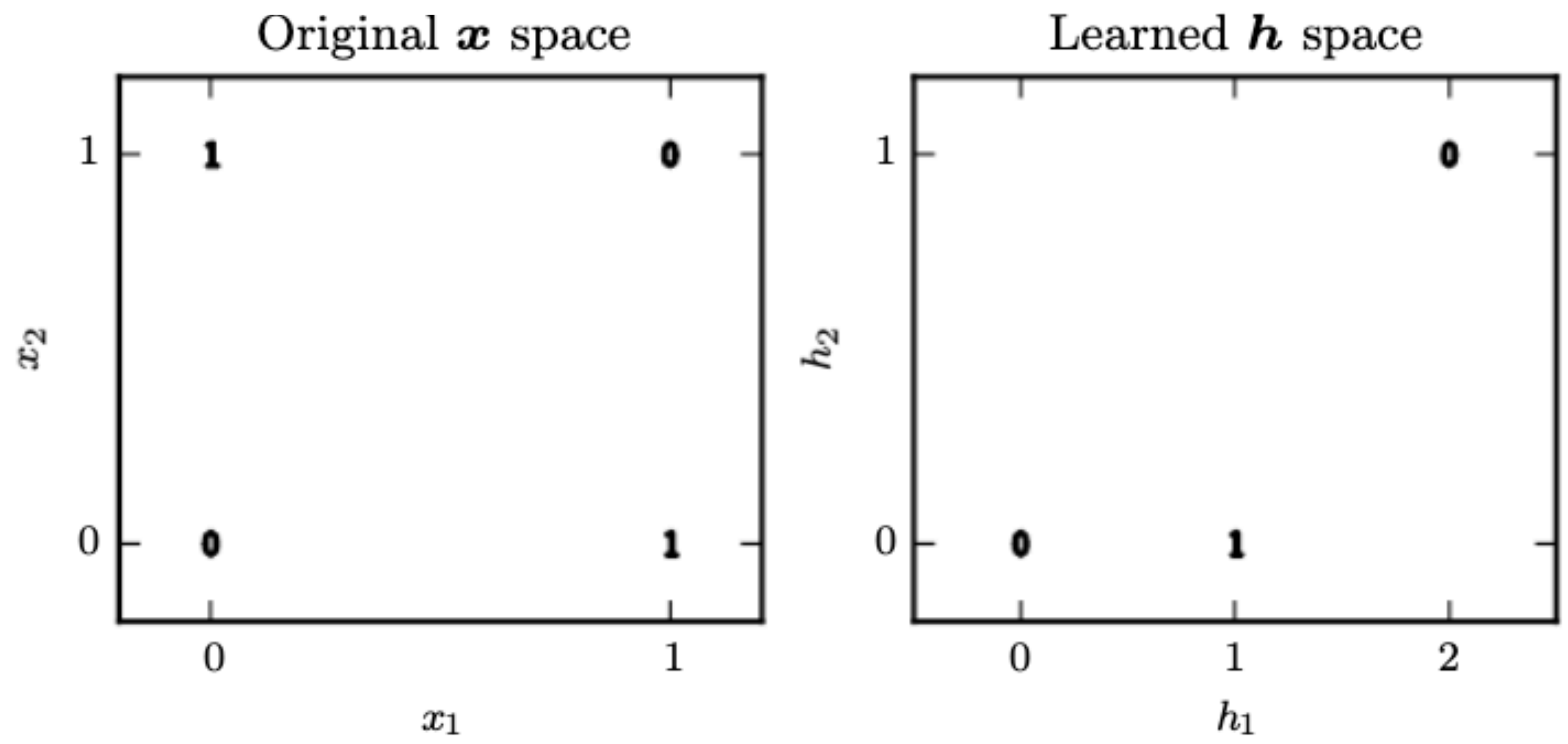
- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions
- One perspective: integrating extracted features



Non-linearity, cont.

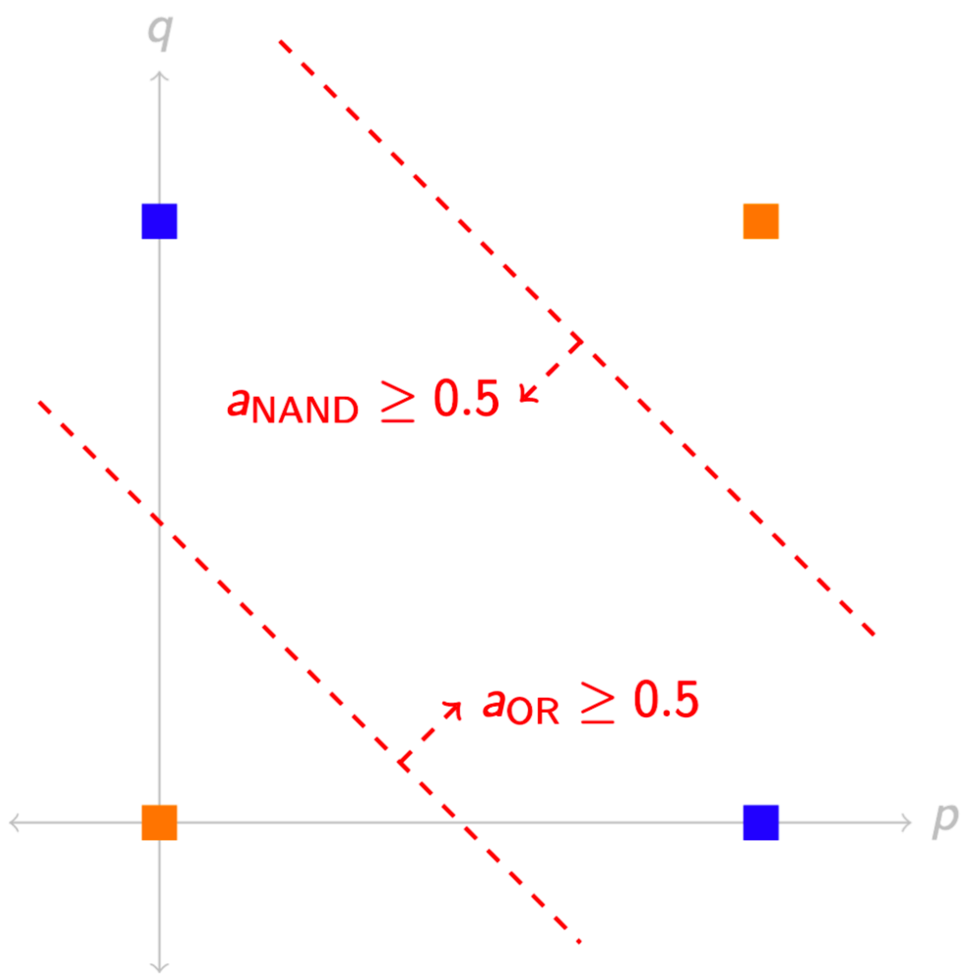
- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions
- One perspective: integrating extracted features
- An equivalent perspective:
  - Transforming the input space (source; p. 169)
  - This is a *non-linear* transformation
  - Space folding intuition more generally (also GBC sec 6.4.1)

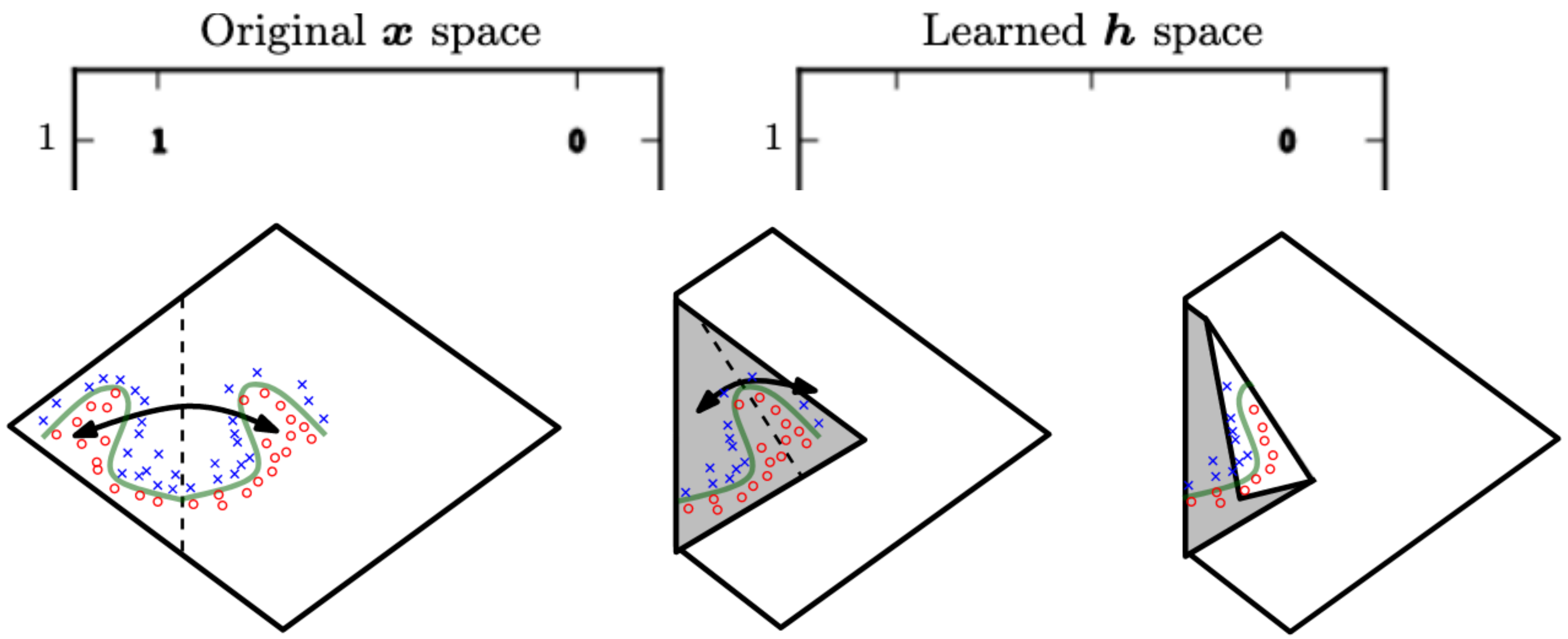




Non-linearity, cont.

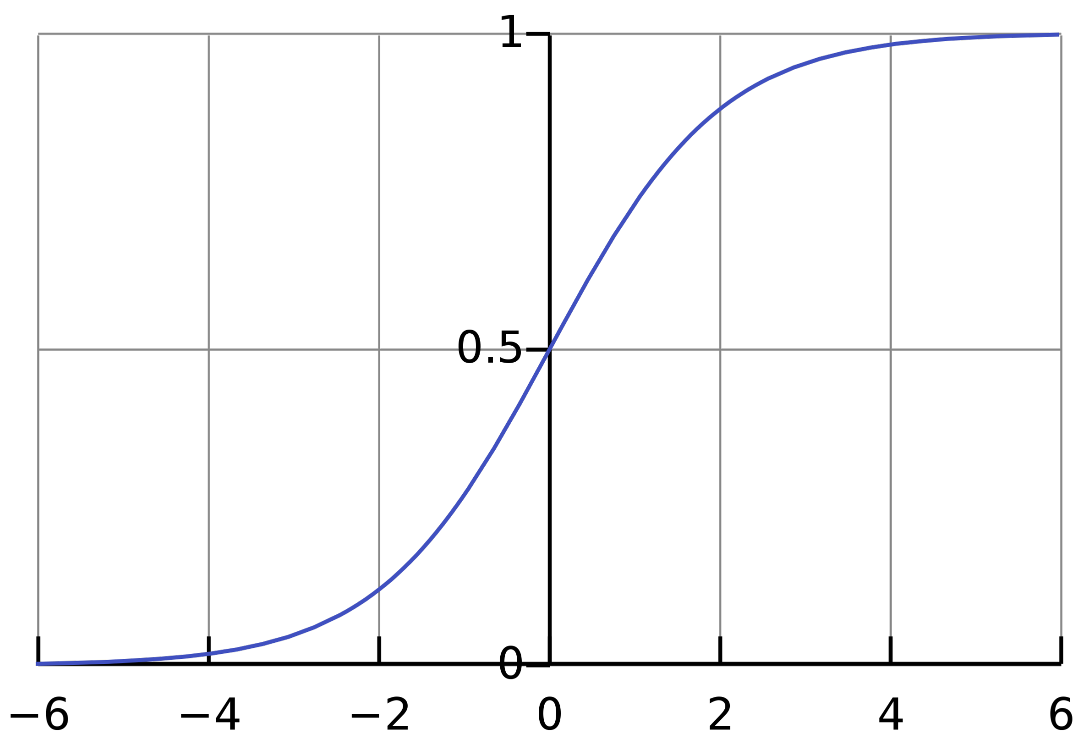
- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions
- One perspective: integrating extracted features
- An equivalent perspective:
  - Transforming the input space (source; p. 169)
  - This is a *non-linear* transformation
  - Space folding intuition more generally (also GBC sec 6.4.1)





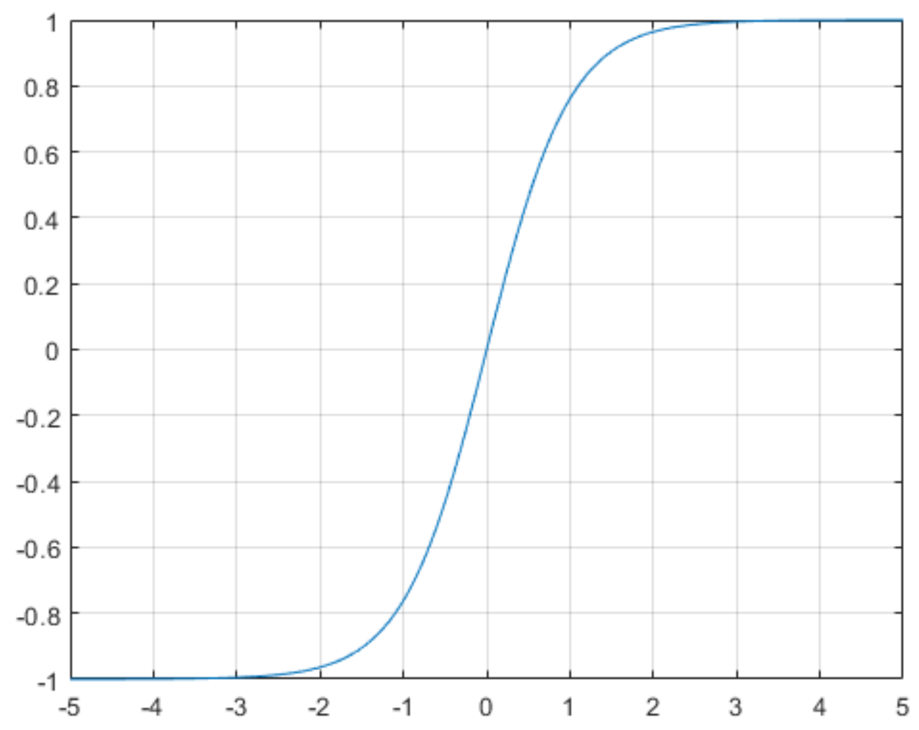
Activation Functions: Hidden Layer

sigmoid



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

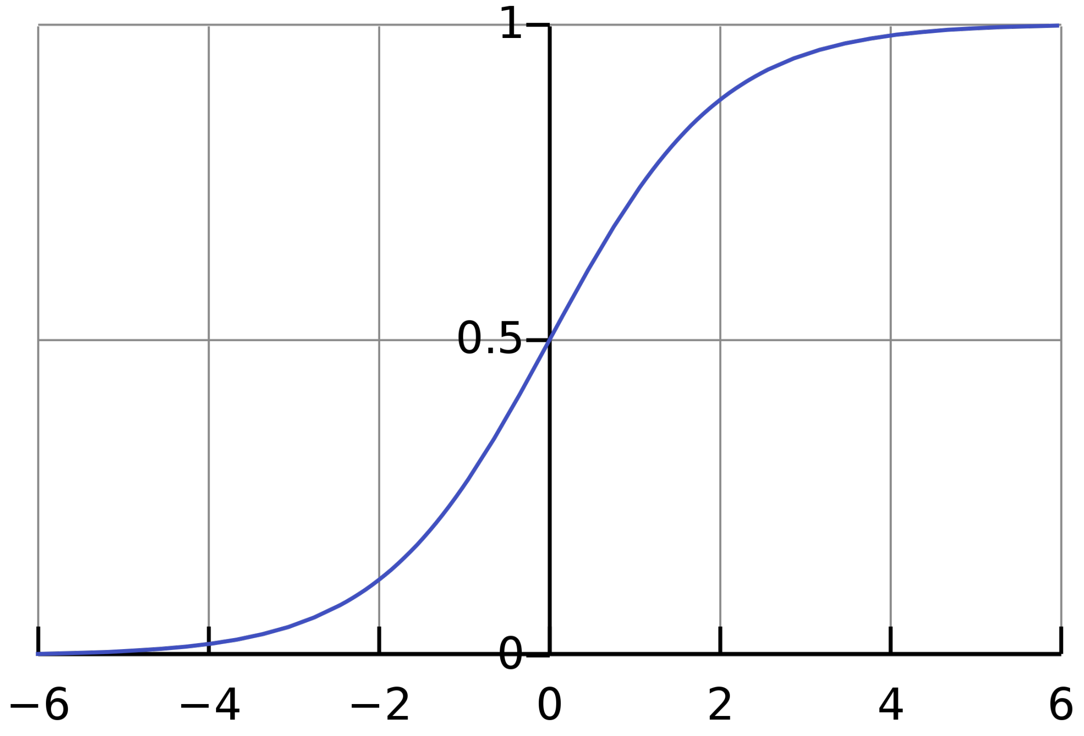
tanh



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

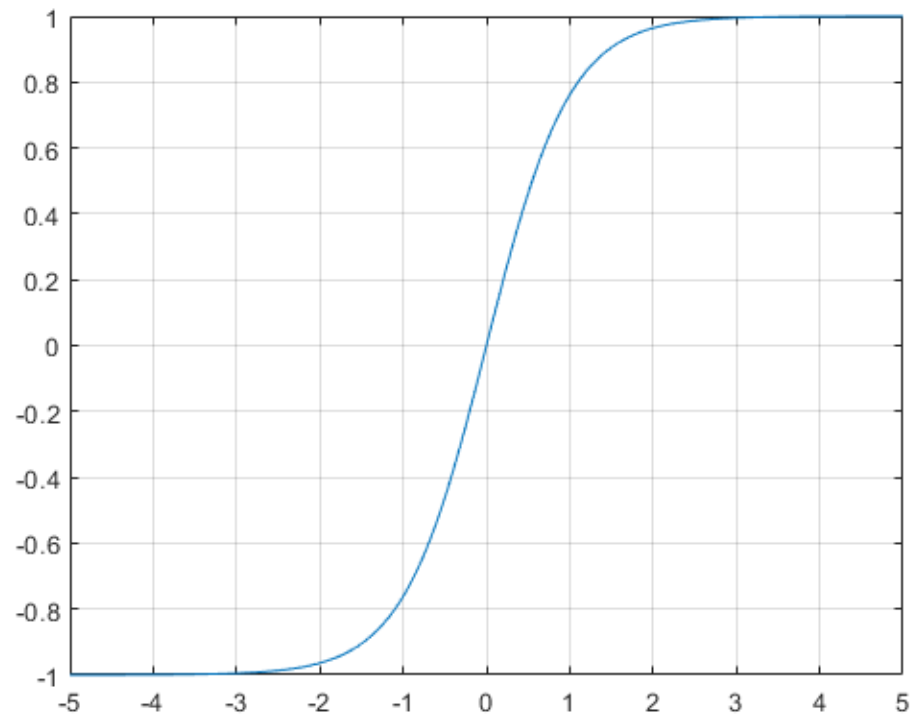
Activation Functions: Hidden Layer

sigmoid



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

tanh

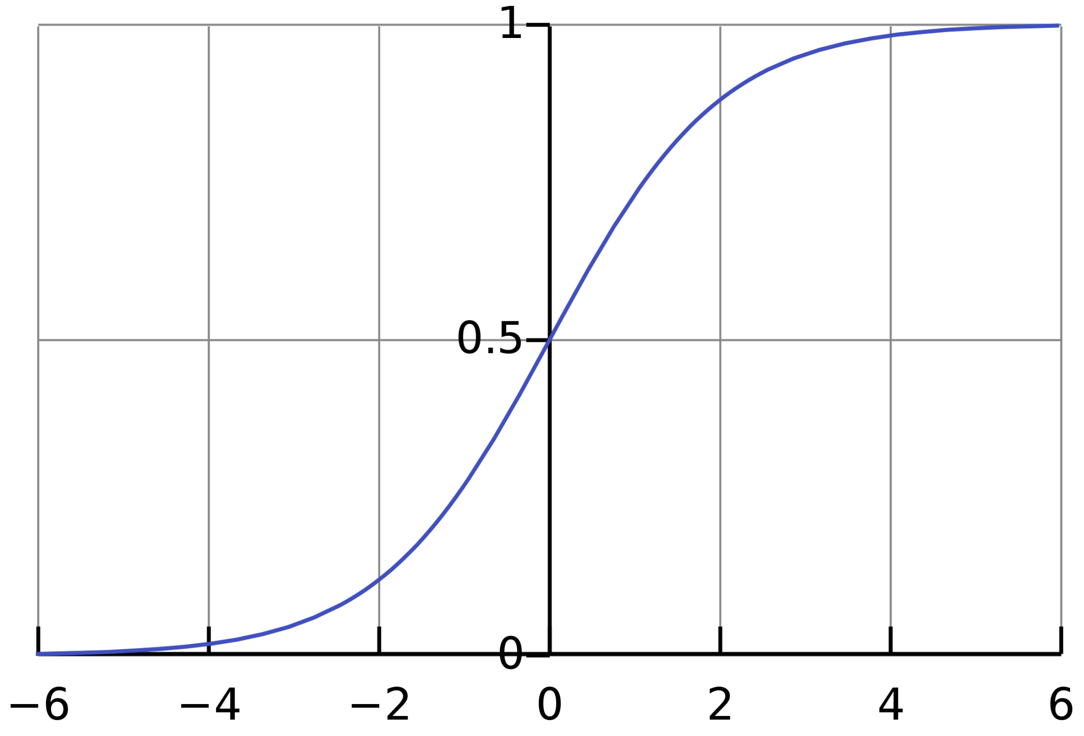


$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

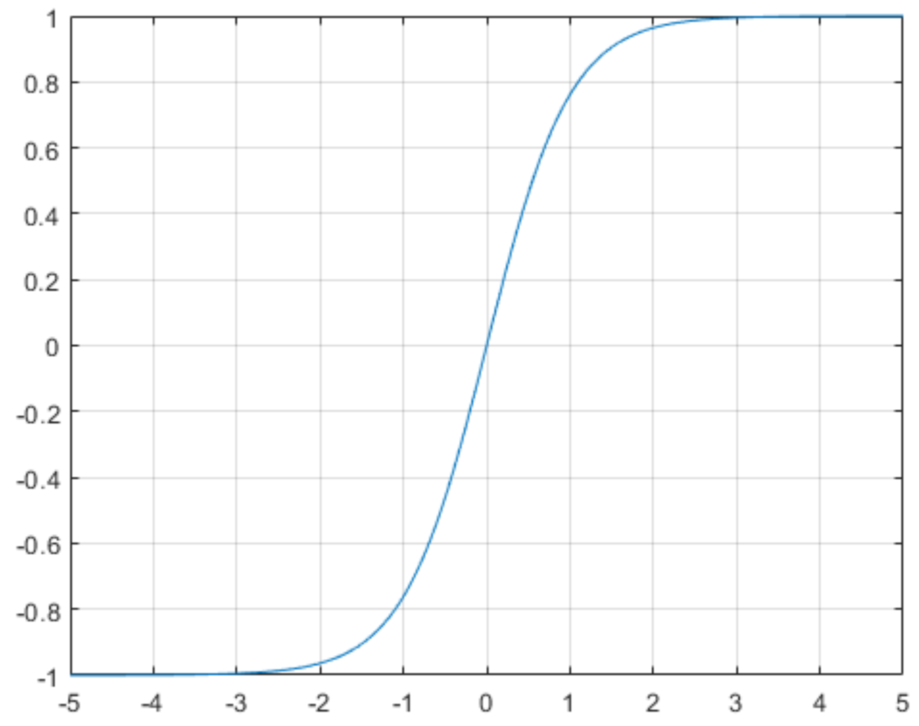
Problem with these two: derivative "saturates" (nearly 0) everywhere except near origin

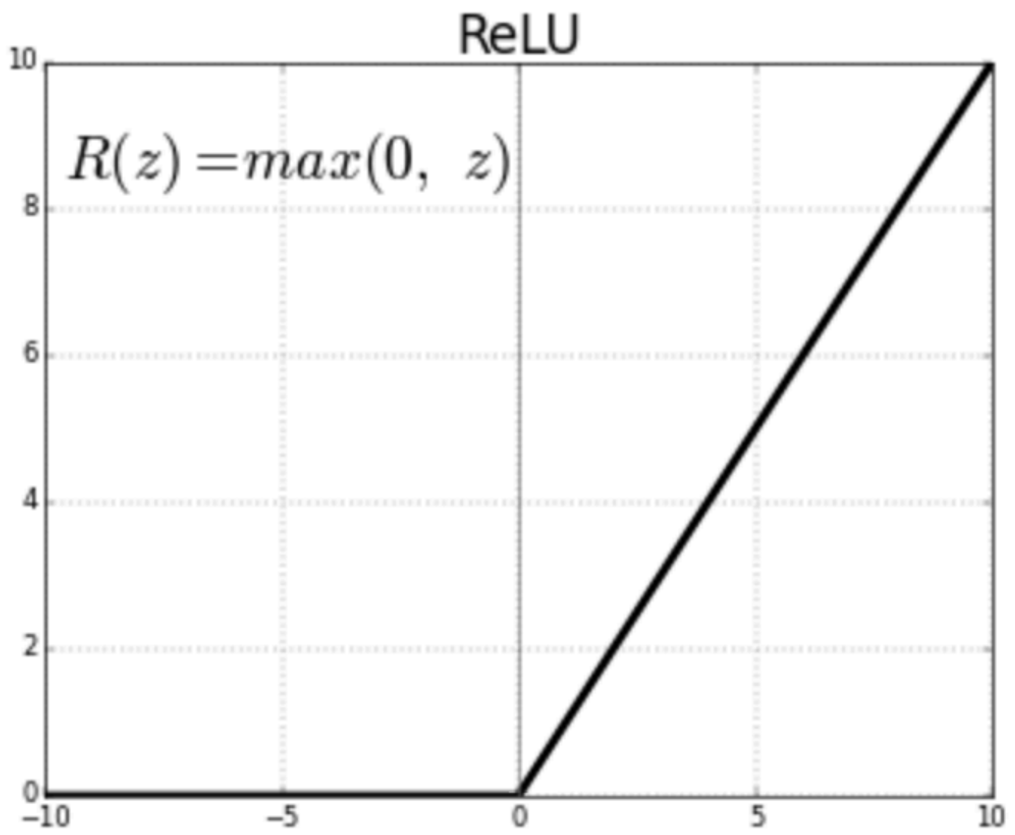
Activation Functions: Hidden Layer

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 with these two: derivative "saturates" (nearly 0) everywhere except near origin

ReLU does not saturate. Good "default"

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

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

Computing with a Single Input

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

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

Shape: $(n_0, 1)$

Computing with a Single Input

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

$$W^1 = \begin{bmatrix} w_{00} & w_{10} & \cdots & w_{0n_0} \\ w_{10} & w_{11} & \cdots & w_{1n_0} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_10} & w_{n_11} & \cdots & w_{n_1n_0} \end{bmatrix}$$

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

Shape: $(n_0, 1)$

Shape: (n_1, n_0)

n_0: dimension of input (layer 0)

n_1: output dimension of layer 1

Computing with a Single Input

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

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

Shape: $(n_0, 1)$

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

Shape: (n_1, n_0)

n_0: dimension of input (layer 0)

n_1: output dimension of layer 1

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

Shape: $(n_1, 1)$

Mini-batch Gradient Descent

```
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

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

Computing with a Batch of Inputs

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

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

Shape: (n_0, k)

k: batch_size

Computing with a Batch of Inputs

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

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

Shape: (n_0, k)

k: batch_size

Computing with a Batch of Inputs

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

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

Shape: (n_0, k)
k: batch_size

Shape: (n_1, n_0)
n_0: dimension of input (layer 0)
n_1: output dimension of layer 1

Computing with a Batch of Inputs

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

$$X = \begin{bmatrix} x_0^0 & x_0^1 & \dots & x_0^k \\ x_1^0 & x_1^1 & \dots & x_1^k \\ \vdots & \vdots & \ddots & \vdots \\ x_{n_0}^0 & x_{n_0}^1 & \dots & x_{n_0}^k \end{bmatrix} \quad W^1 = \begin{bmatrix} w_{00} & w_{01} & \cdots & w_{0n_1} \\ w_{10} & w_{11} & \cdots & w_{1n_1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_00} & w_{n_01} & \cdots & w_{n_0n_1} \end{bmatrix} \quad b^1 = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n_1} \end{bmatrix}$$

Shape: (n_0, k)
k: batch_size

Shape: (n_1, n_0)
n_0: dimension of input (layer 0)
n_1: output dimension of layer 1

Shape: $(n_1, 1)$
Added to each col. of W^1X

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** (tensor rather than just matrix)
  - 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** (tensor rather than just matrix)
  - Images: (batch_size, width, height, 3)
  - Sequences: (batch_size, seq_len, representation_size)
- Two comments:
  - In your code, **annotate every tensor** with a comment showing **intended shape**
  - When debugging, look at shapes early on!!

Note on mini-batches and shape

Note on mini-batches and shape

- **Warning**: PyTorch / NN libraries typically multiply matrices in the **opposite direction** as Linear Algebra notation

Note on mini-batches and shape

- **Warning**: PyTorch / NN libraries typically multiply matrices in the **opposite direction** as Linear Algebra notation
- e.g. the input to an MLP should be (batch_size, embedding_size) rather than (embedding_size, batch_size)

Note on mini-batches and shape

- **Warning**: PyTorch / NN libraries typically multiply matrices in the **opposite direction** as Linear Algebra notation
- e.g. the input to an MLP should be (batch_size, embedding_size) rather than (embedding_size, batch_size)
- The **last dimension of the input** should match the **first dimension of the weights**

Note on mini-batches and shape

- **Warning**: PyTorch / NN libraries typically multiply matrices in the **opposite direction** as Linear Algebra notation
- e.g. the input to an MLP should be (batch_size, embedding_size) rather than (embedding_size, batch_size)
- The **last dimension of the input** should match the **first dimension of the weights**
- You can think of it as these libraries preferring $x^T W^T$ to Wx

Note on mini-batches and shape

- **Warning**: PyTorch / NN libraries typically multiply matrices in the **opposite direction** as Linear Algebra notation
- e.g. the input to an MLP should be (batch_size, embedding_size) rather than (embedding_size, batch_size)
- The **last dimension of the input** should match the **first dimension of the weights**
- You can think of it as these libraries preferring $x^T W^T$ to Wx
  - (The result of this multiplication is the same, just transposed)

Next Time

- Feedforward models for language
  - “Deep Averaging Network”
  - Feedforward language model
- Training regularization
- Language model quality metrics