Feed-Forward Neural Networks

Ling 282/482: Deep Learning for Computational Linguistics

C.M. Downey

Fall 2025

Note on Random Seeds

Note on Random Seeds

- In word2vec.py / util.py:

```
def set_seed(seed: int) -> None:
    """Sets various random seeds. """
    random.seed(seed)
    np.random.seed(seed)
```

Note on Random Seeds

- In word2vec.py / util.py:

```
def set_seed(seed: int) -> None:
    """Sets various random seeds. """
    random.seed(seed)
    np.random.seed(seed)
```

- Random seed
  - Behavior of **pseudo-random number generators** is determined by their “seed” value
  - If not specified, determined by e.g. # of seconds since 1970
  - **Same seed —> same (non-random) behavior**

Note on Random Seeds

- In word2vec.py / util.py:

```
def set_seed(seed: int) -> None:
    """Sets various random seeds. """
    random.seed(seed)
    np.random.seed(seed)
```

- Random seed
  - Behavior of **pseudo-random number generators** is determined by their "seed" value
  - If not specified, determined by e.g. # of seconds since 1970
  - **Same seed —> same (non-random) behavior**
- **Sources of randomness** in DL: **shuffling the data** each epoch, **weight initialization**, negative sampling, …

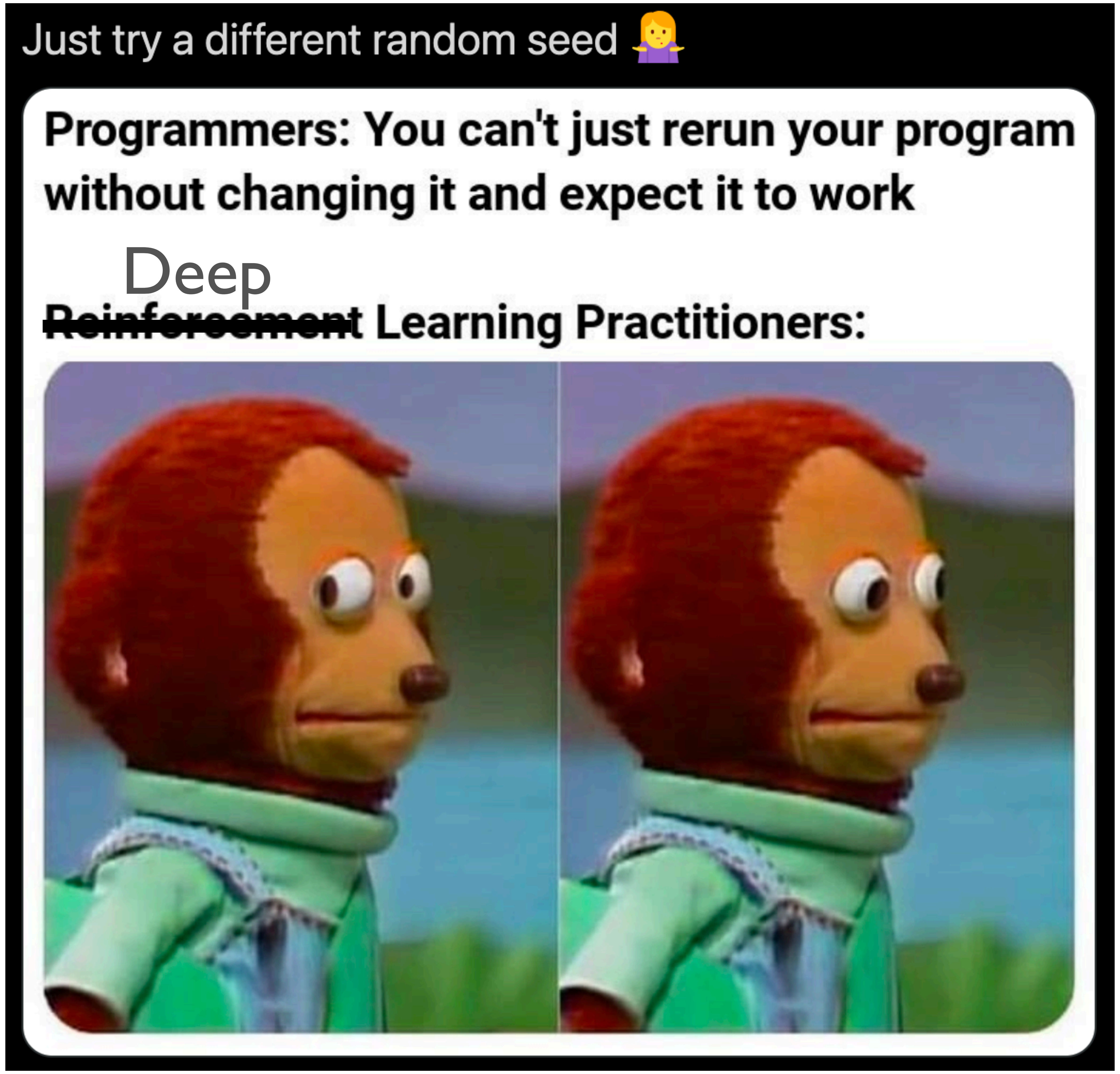
Note on Random Seeds

- In word2vec.py / util.py:

```
def set_seed(seed: int) -> None:
    """Sets various random seeds. """
    random.seed(seed)
    np.random.seed(seed)
```

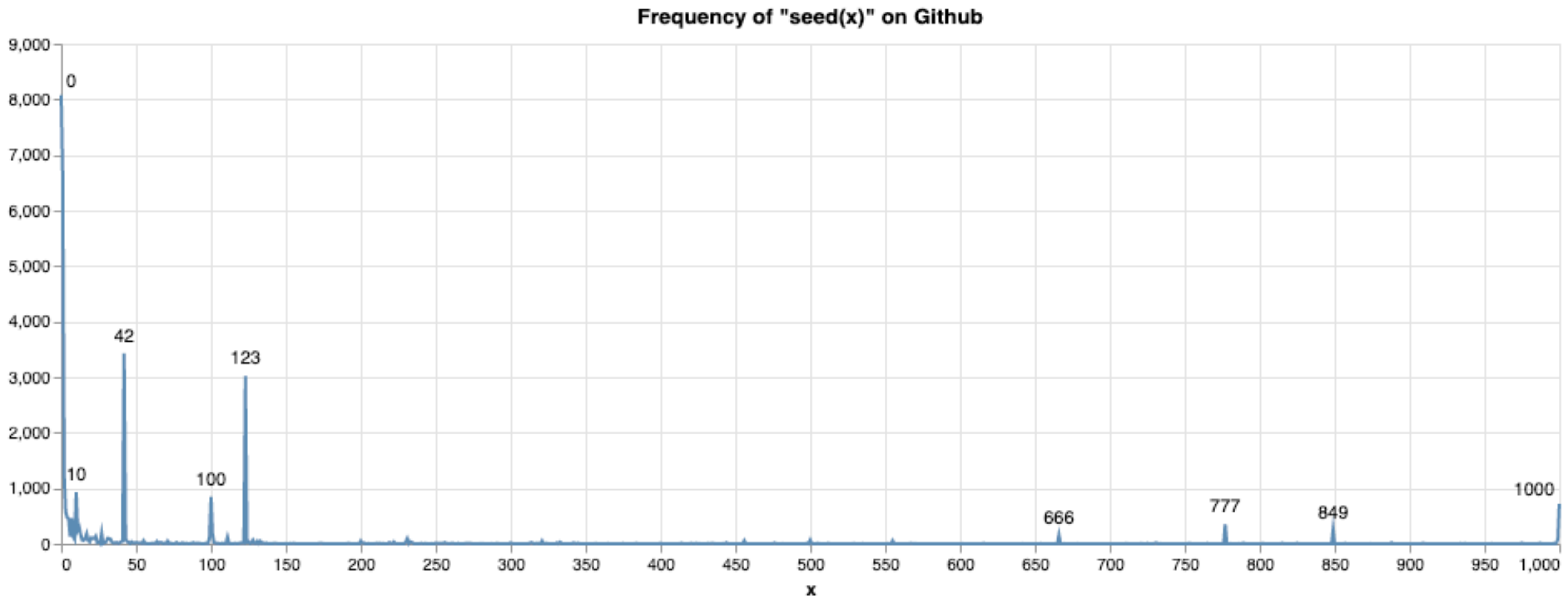
- Random seed
  - Behavior of **pseudo-random number generators** is determined by their “seed” value
  - If not specified, determined by e.g. # of seconds since 1970
  - **Same seed —> same (non-random) behavior**
- **Sources of randomness** in DL: **shuffling the data** each epoch, **weight initialization**, negative sampling, …
- Very important for **reproducibility!**
  - In general, run on several seeds and report means / std’s

Random Seeds and Reproducibility



Random Seeds, cont

- Ideally: “randomly generate” seeds, but save/store them!
- Random seed is not a hyper-parameter! (Some discussions in these threads.)

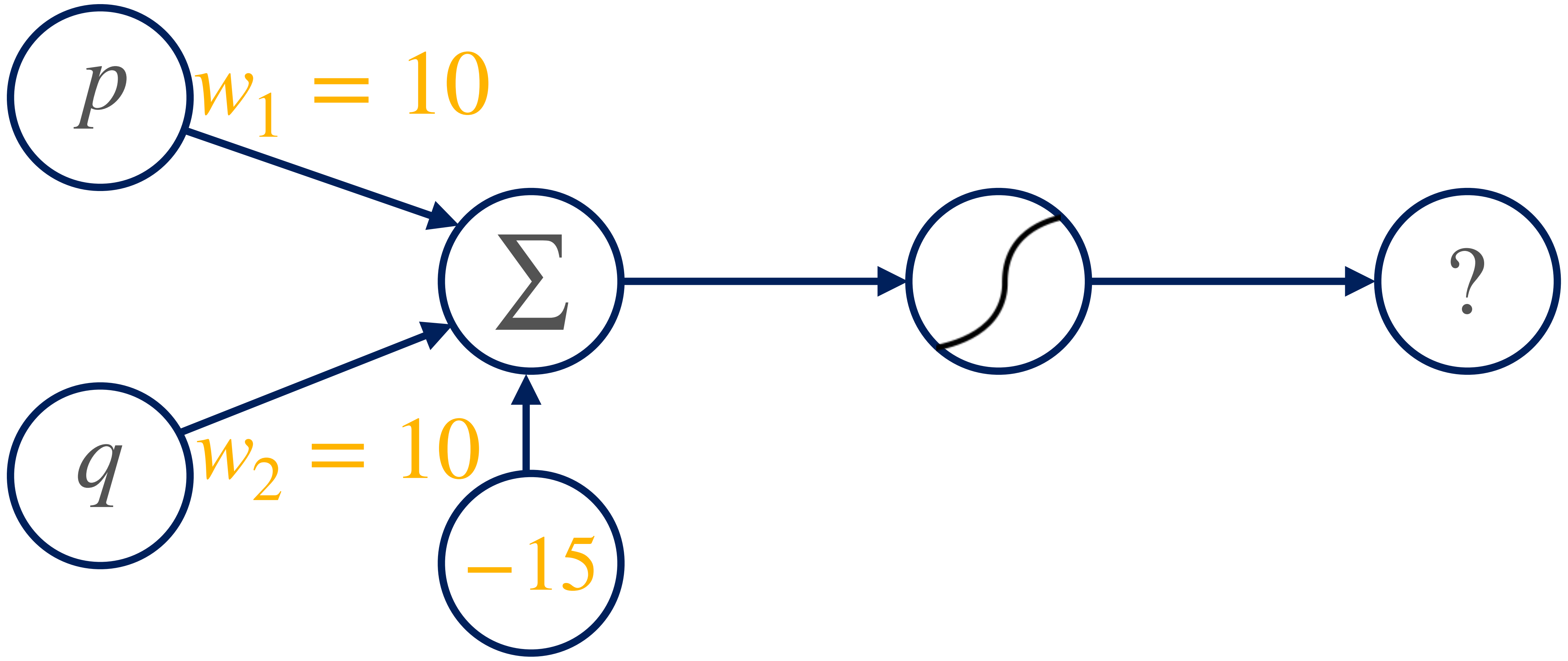


source

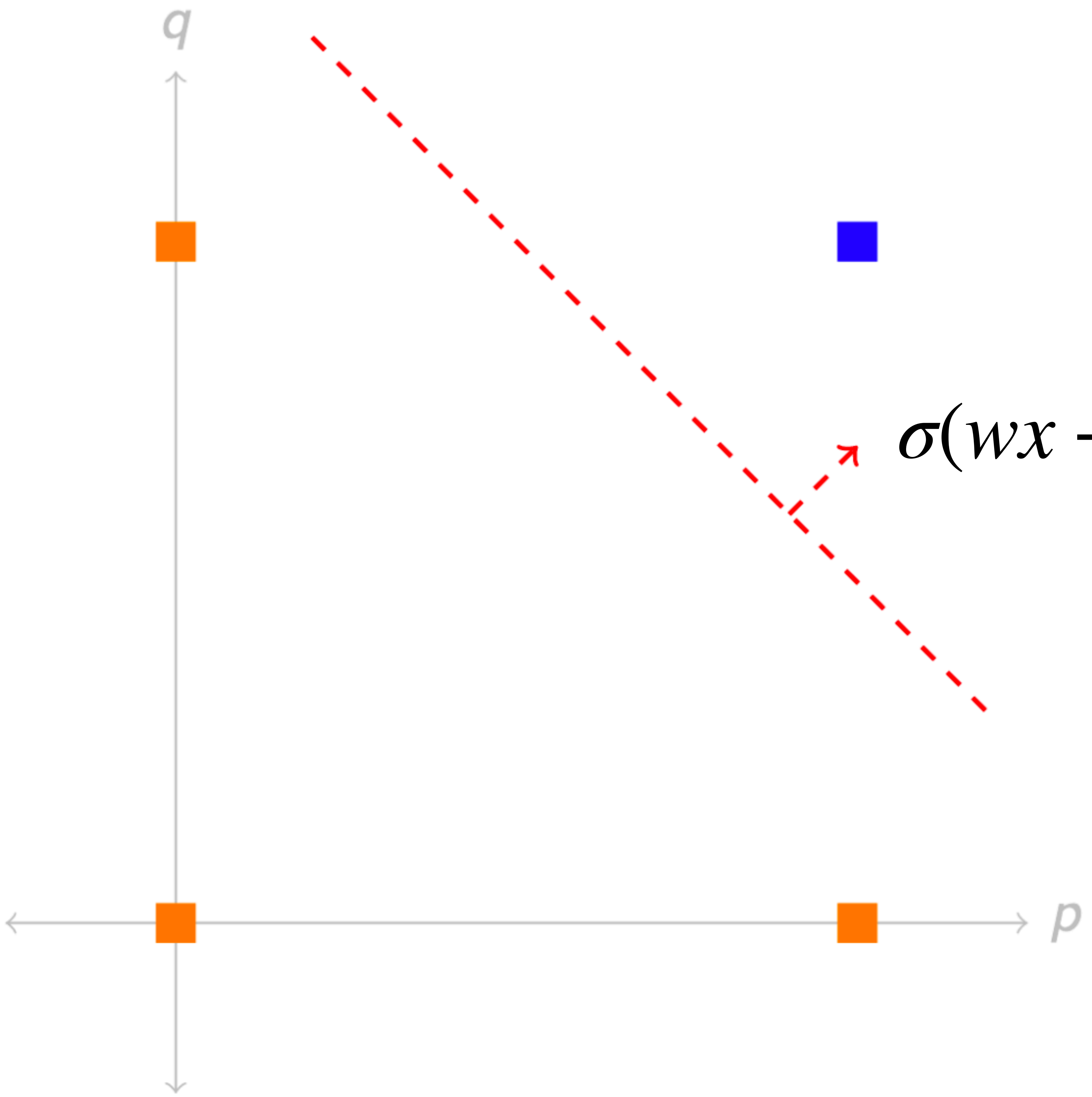
Feed-forward Neural Networks

Recall: AND Perceptron

p	q	p∧q
0	0	0
1	0	0
0	1	0
1	1	1

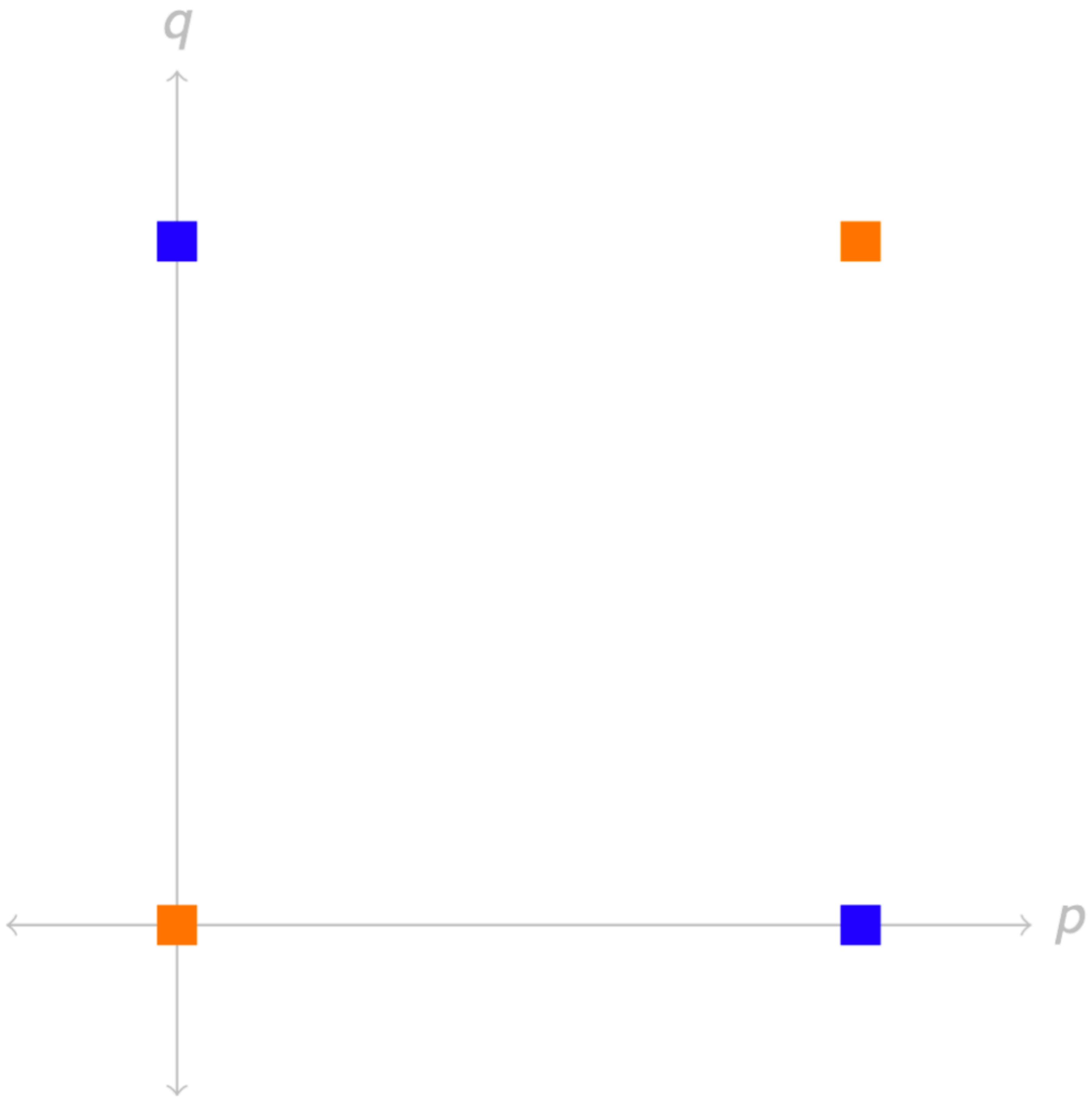


AND Linear Separation

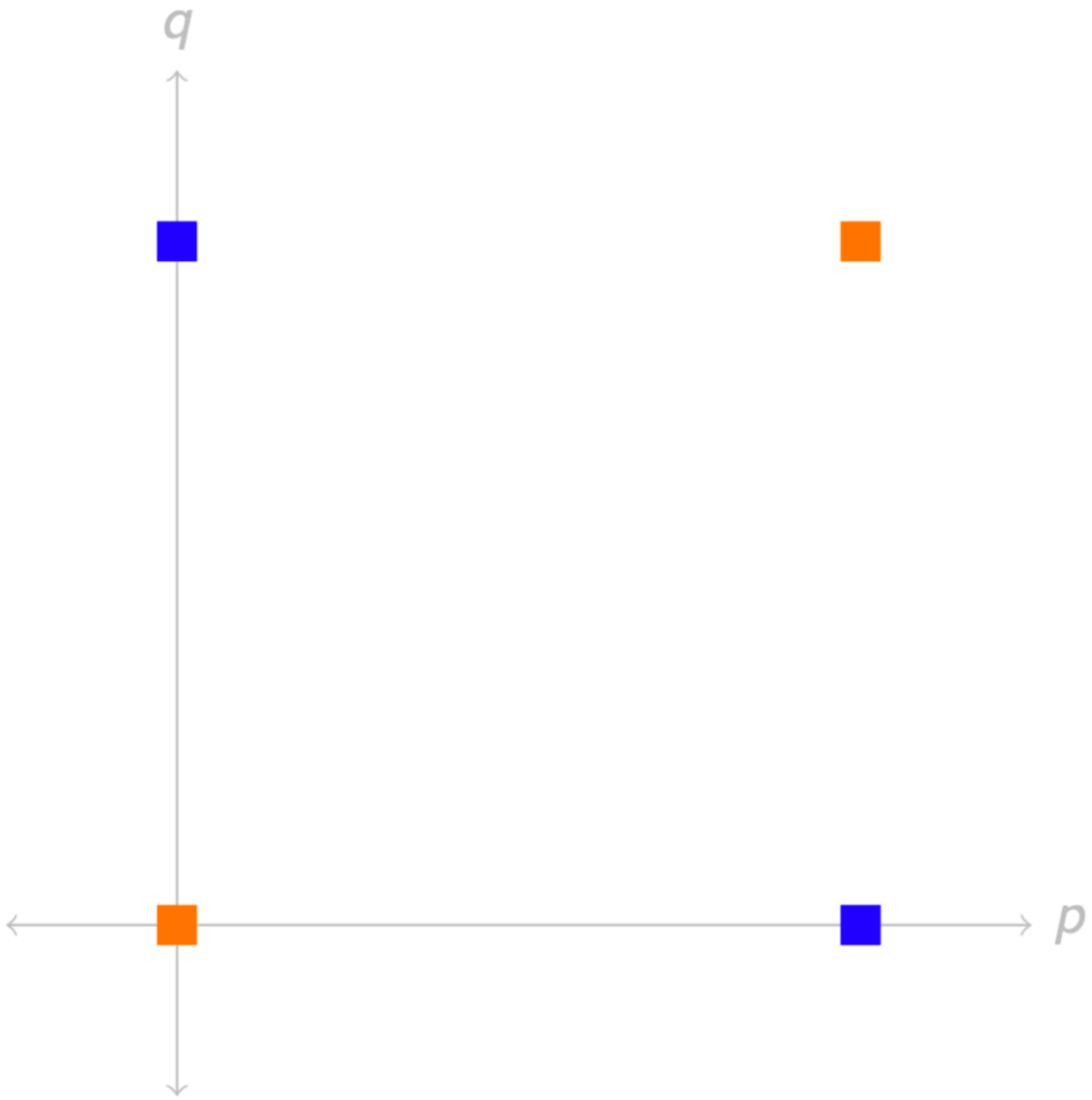


$$\sigma(wx + b) \geq 0.5$$

The XOR problem

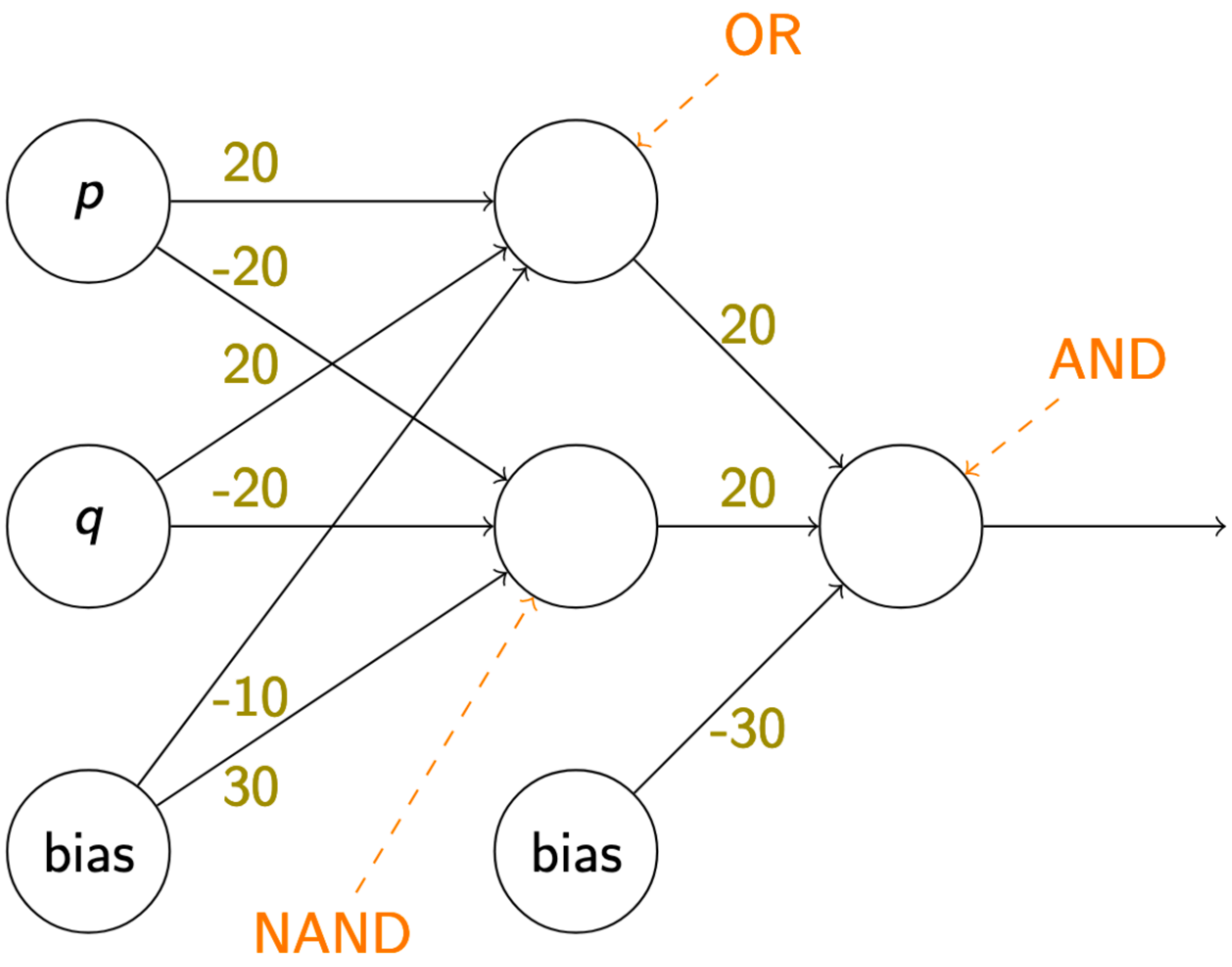


The XOR problem



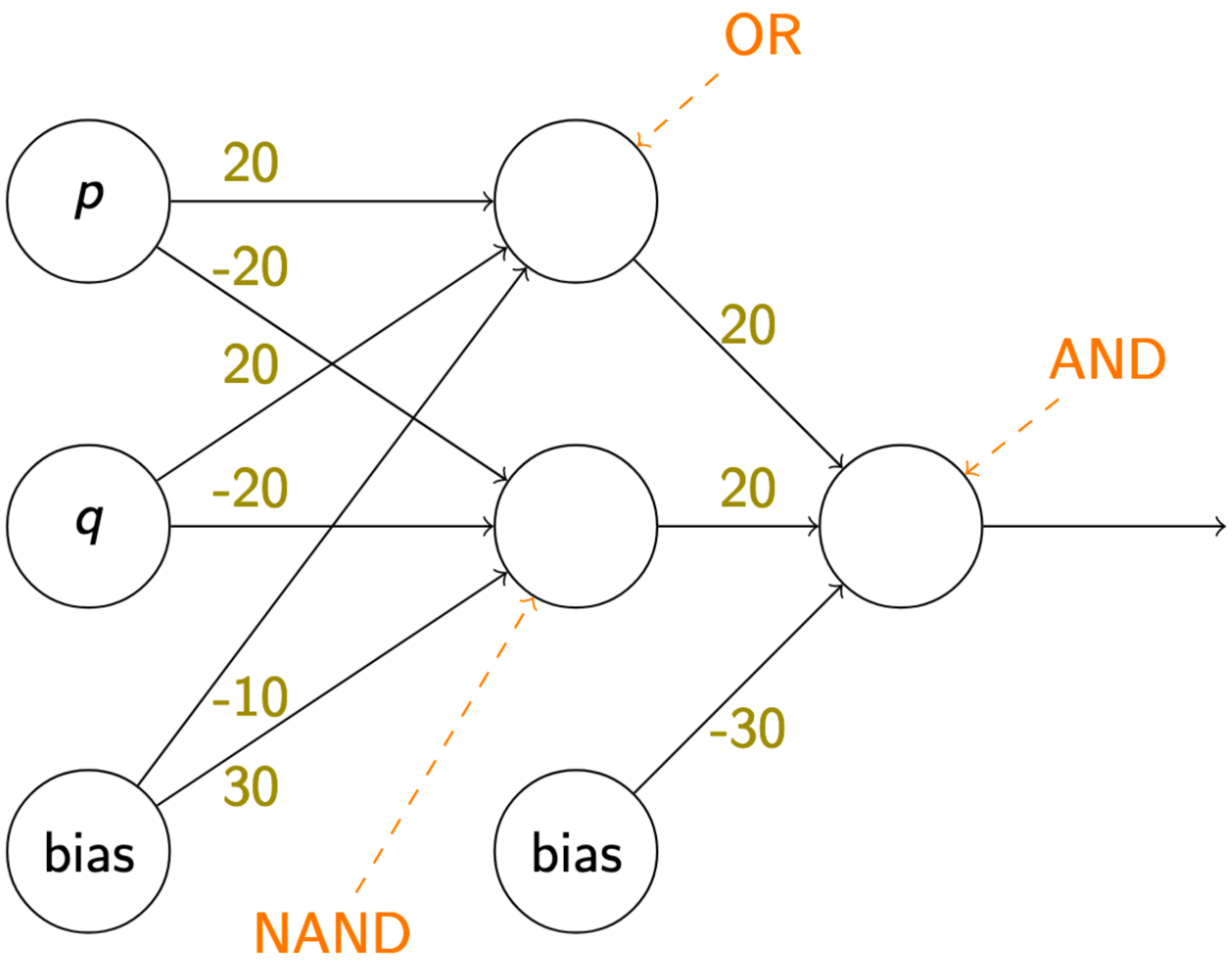
XOR is not linearly separable

Modeling XOR



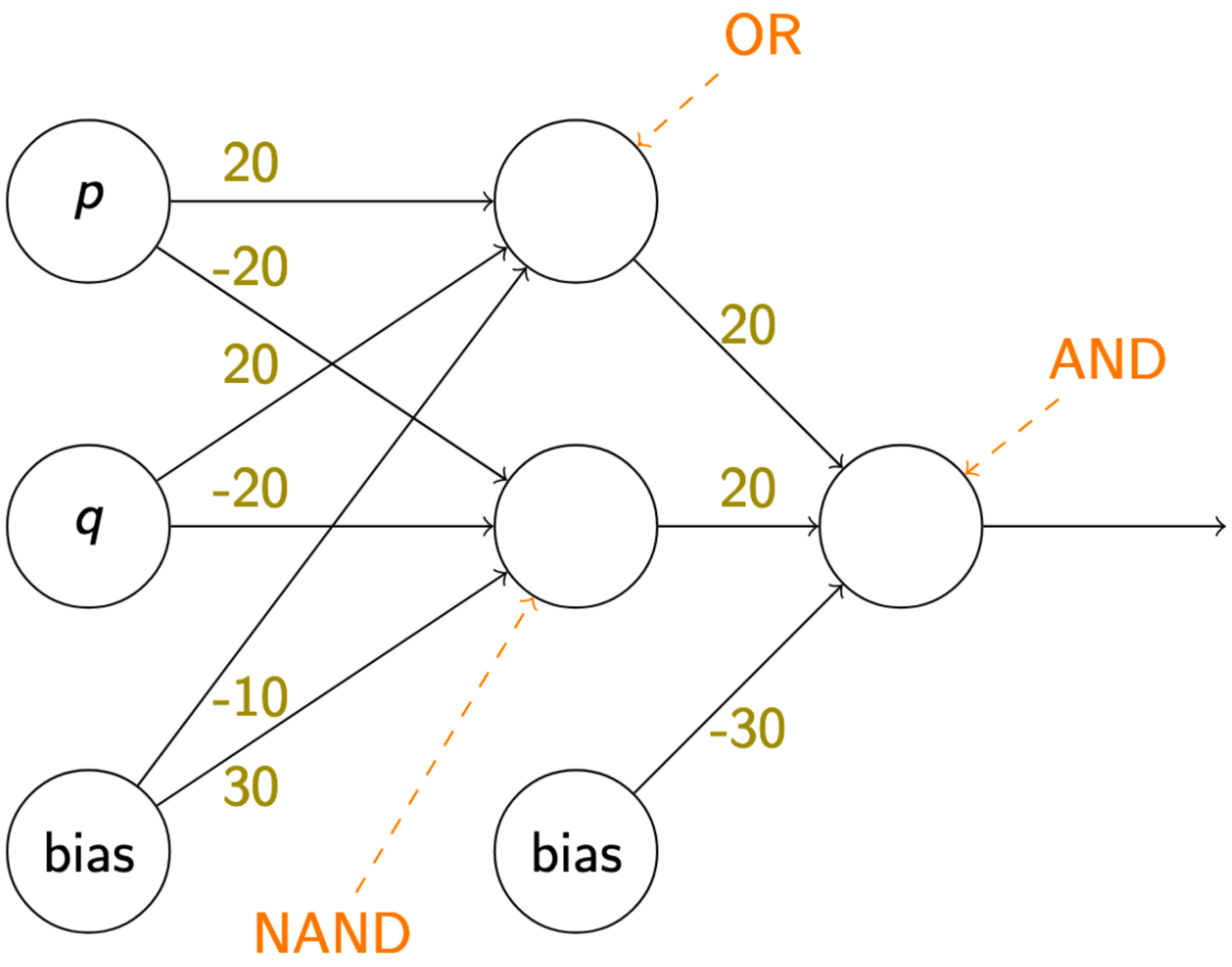
Modeling XOR

- XOR is **decomposable** into other logical functions
  - (p OR q) AND (p NAND q)



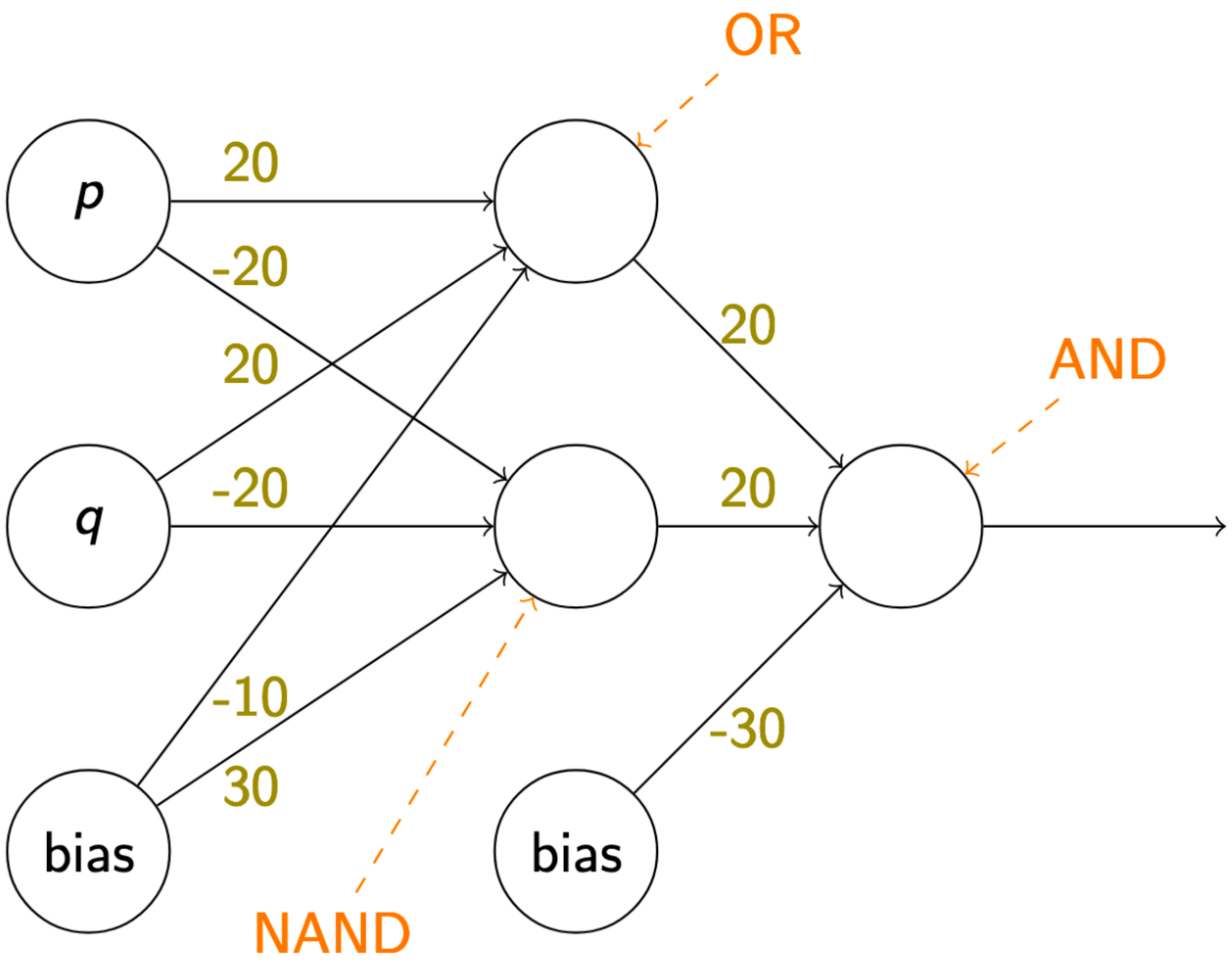
Modeling XOR

- XOR is **decomposable** into other logical functions
  - (p OR q) AND (p NAND q)
- We can represent this with a **Multi-Layer Perceptron (MLP)**



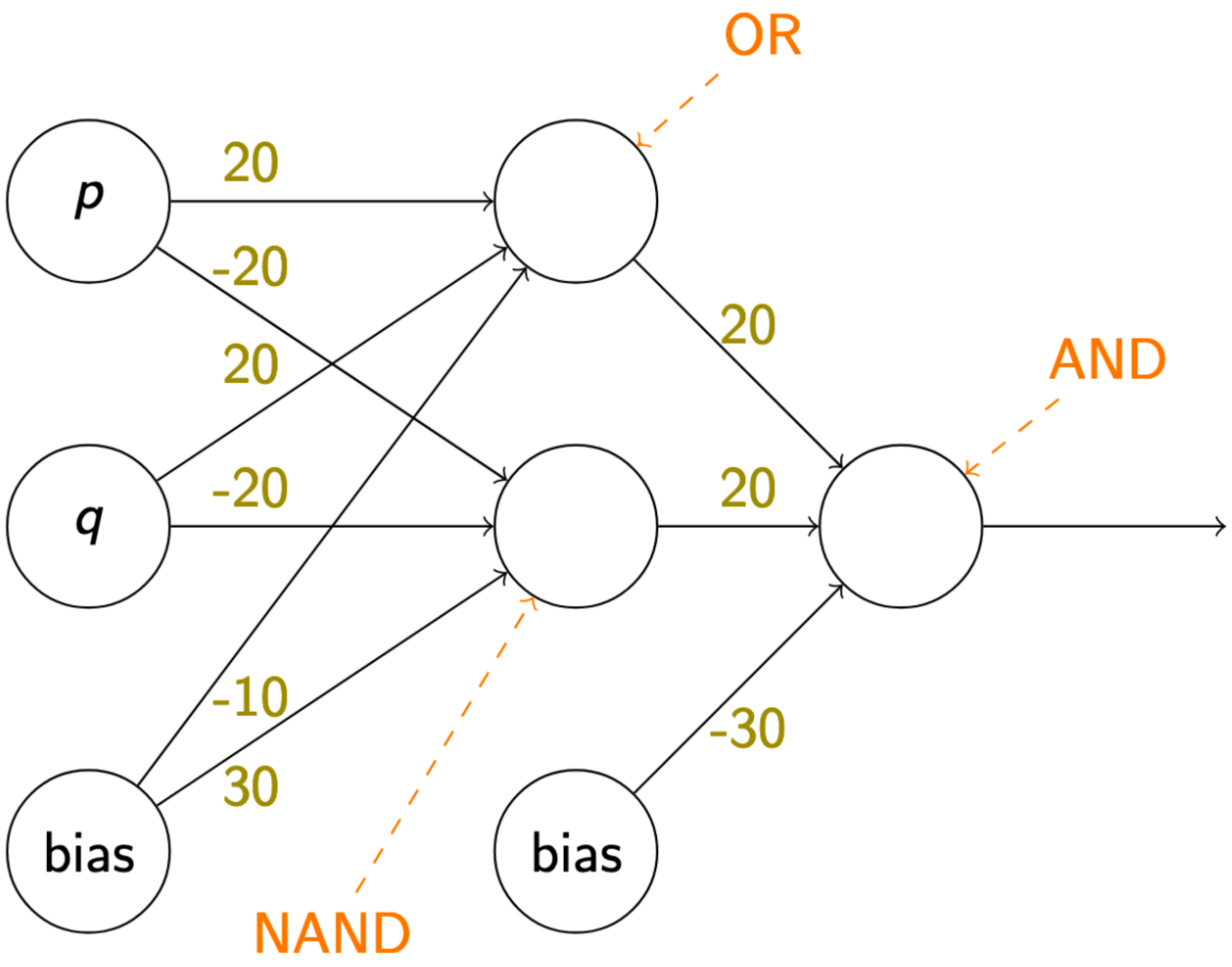
Modeling XOR

- XOR is **decomposable** into other logical functions
  - (p OR q) AND (p NAND q)
- We can represent this with a **Multi-Layer Perceptron (MLP)**
- XOR isn't linearly separable, but OR, NAND, AND are!

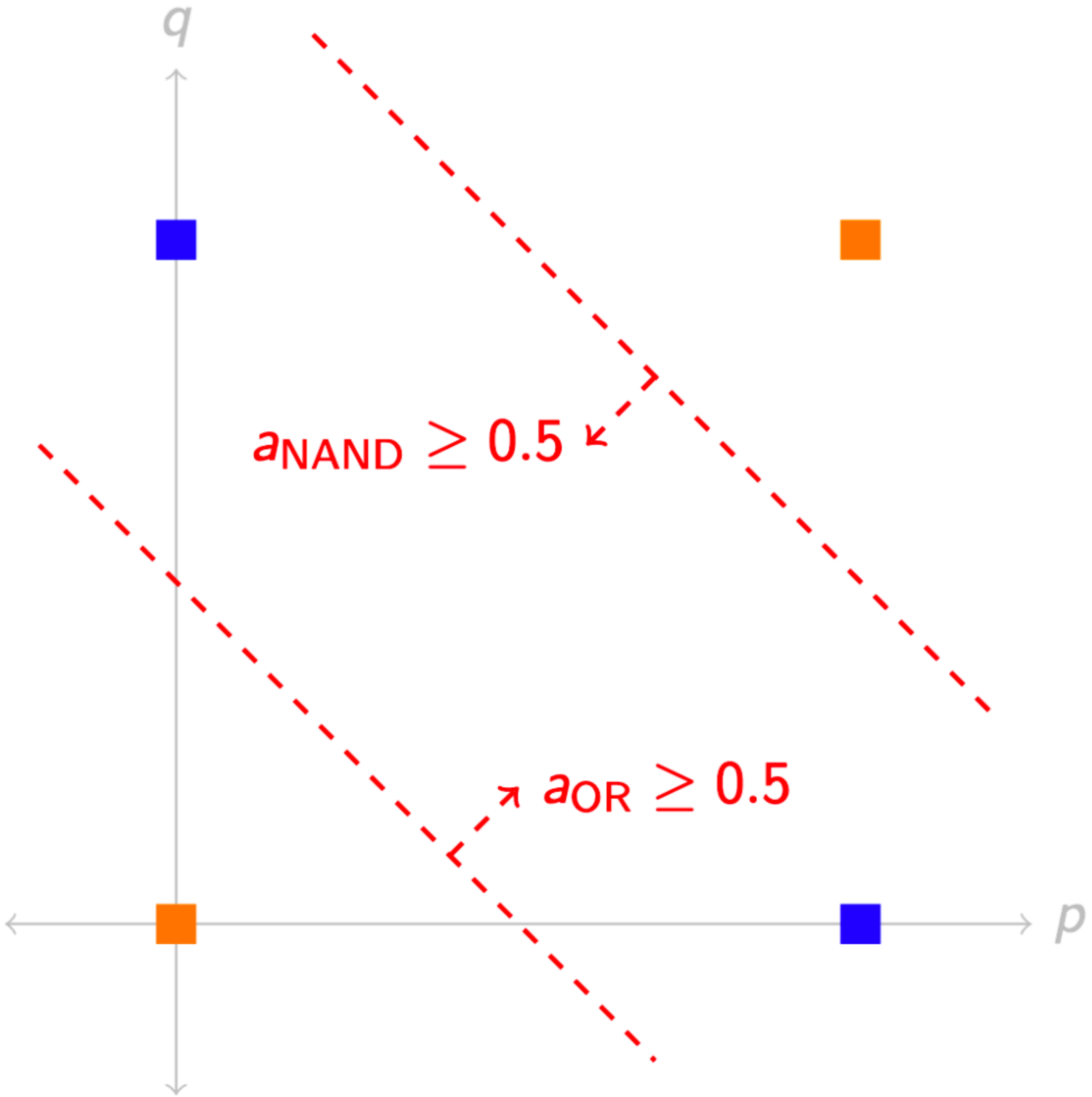


Modeling XOR

- XOR is **decomposable** into other logical functions
  - (p OR q) AND (p NAND q)
- We can represent this with a **Multi-Layer Perceptron (MLP)**
- XOR isn't linearly separable, but OR, NAND, AND are!
- Exercise: verify this perceptron does what we say it does



Modeling XOR



Key Ideas

Key Ideas

- **Hidden layers**: intermediate layers that are **not directly used** as outputs

Key Ideas

- **Hidden layers**: intermediate layers that are **not directly used** as outputs
  - Compute **high-level / abstract features** of the input

Key Ideas

- **Hidden layers**: intermediate layers 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

Key Ideas

- **Hidden layers**: intermediate layers 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

Key Ideas

- **Hidden layers**: intermediate layers 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

Key Ideas

- **Hidden layers**: intermediate layers 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

Key Ideas

- **Hidden layers**: intermediate layers 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
  - ***Technically** one hidden layer is all you need (see next slide)

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 $|f(x) - g(x)| < \epsilon$ 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 $f : [0,1]^m \rightarrow \mathbb{R}$ 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?

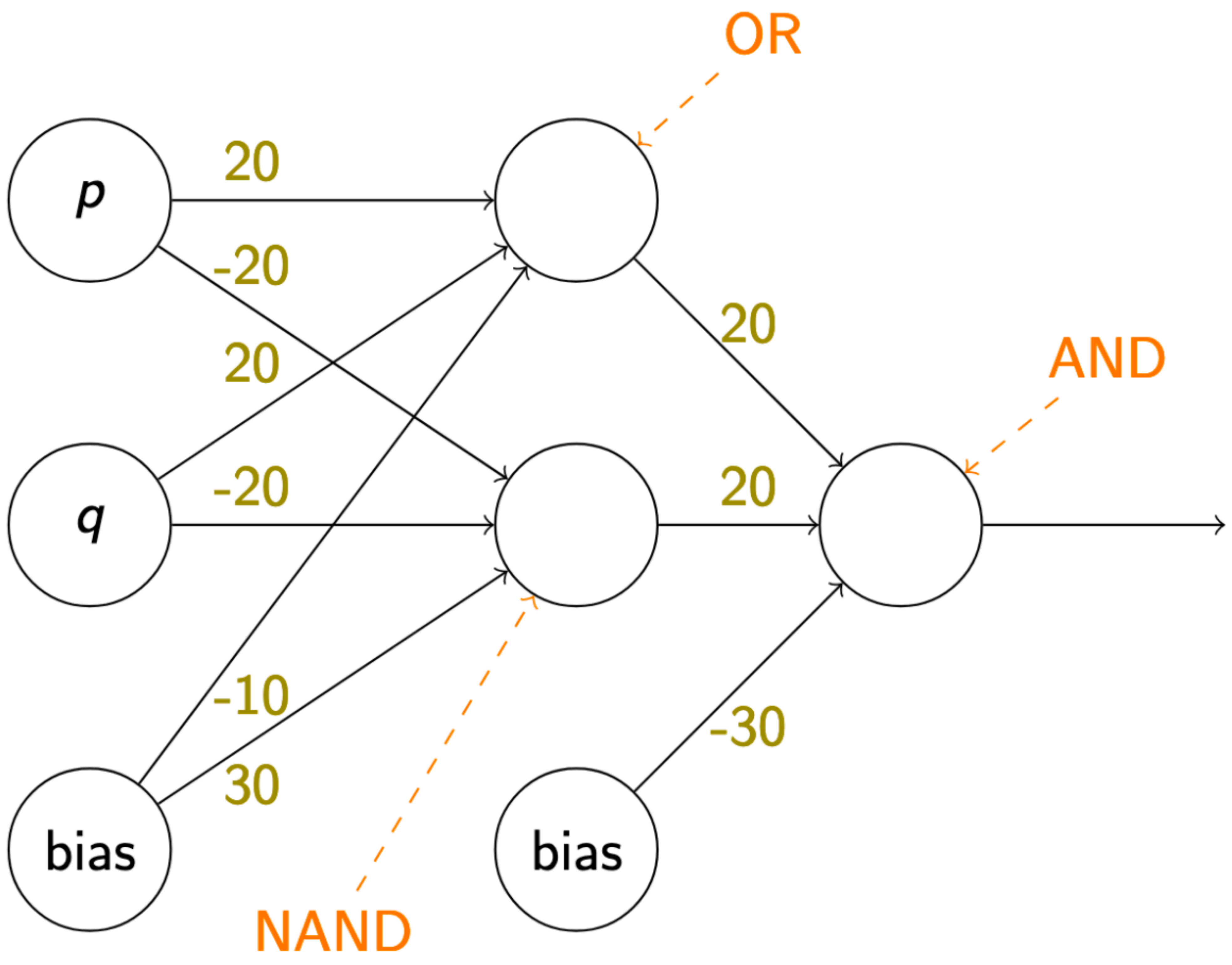
Expressive Power

- 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

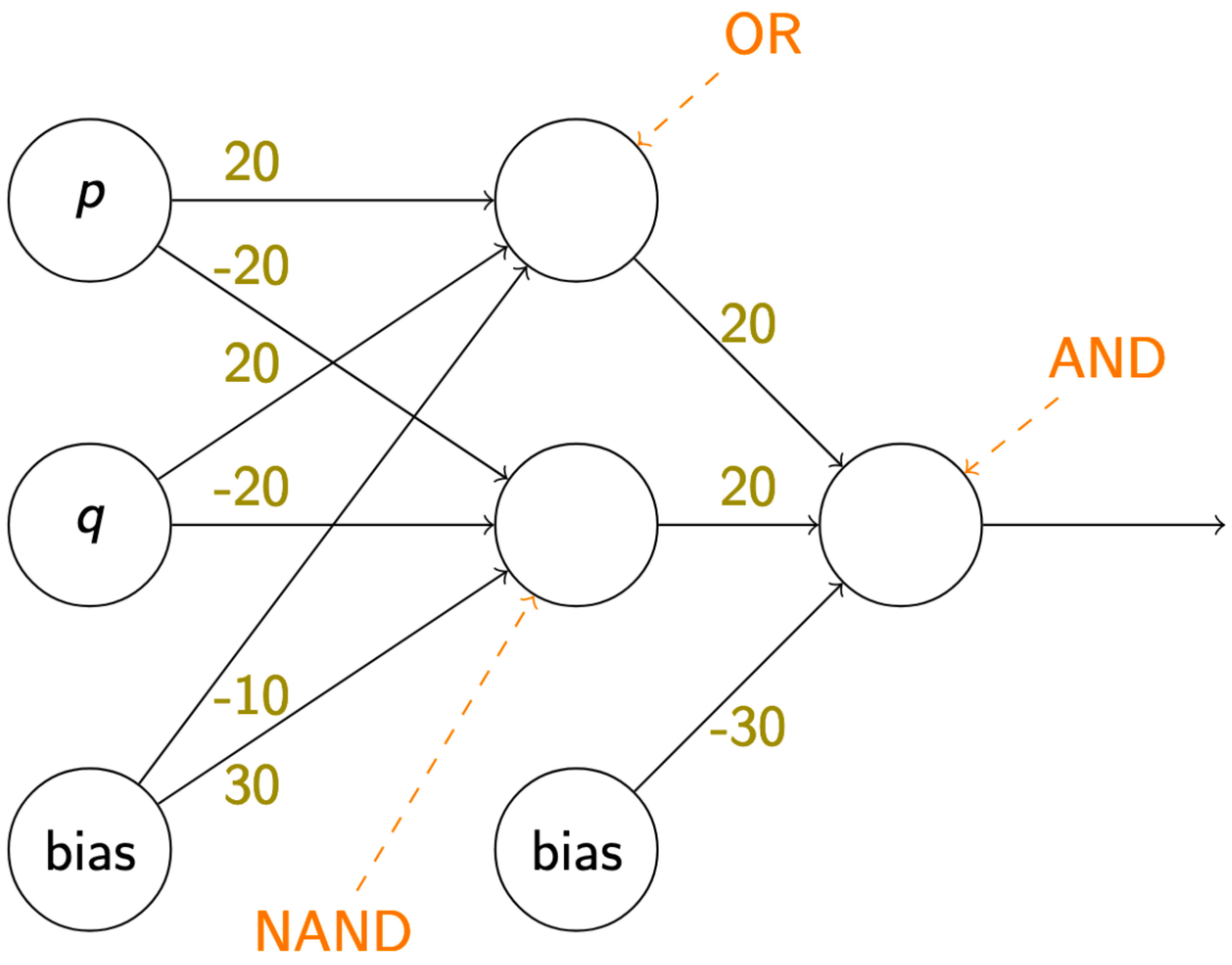
Expressive Power

- 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

XOR Network

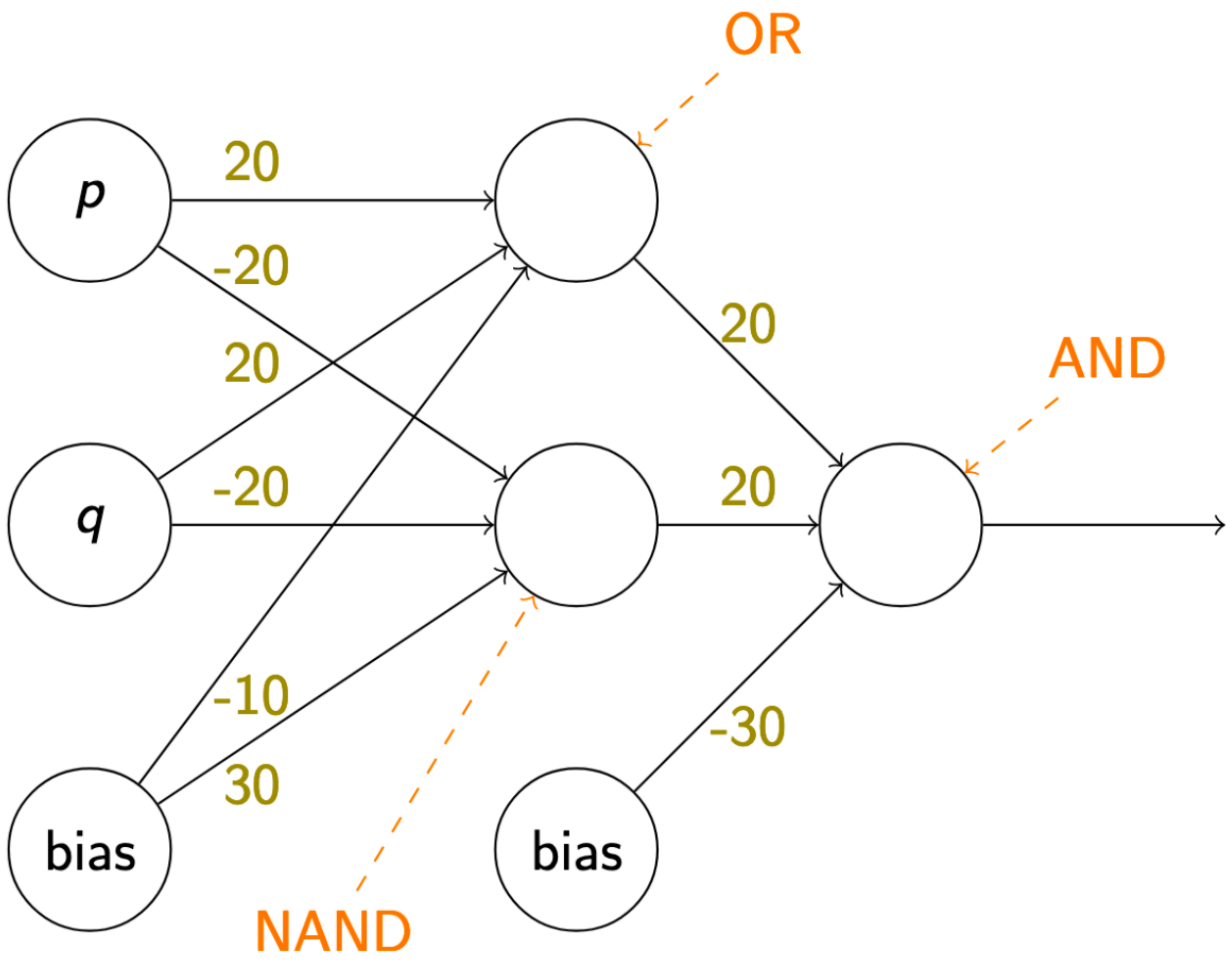


XOR Network



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

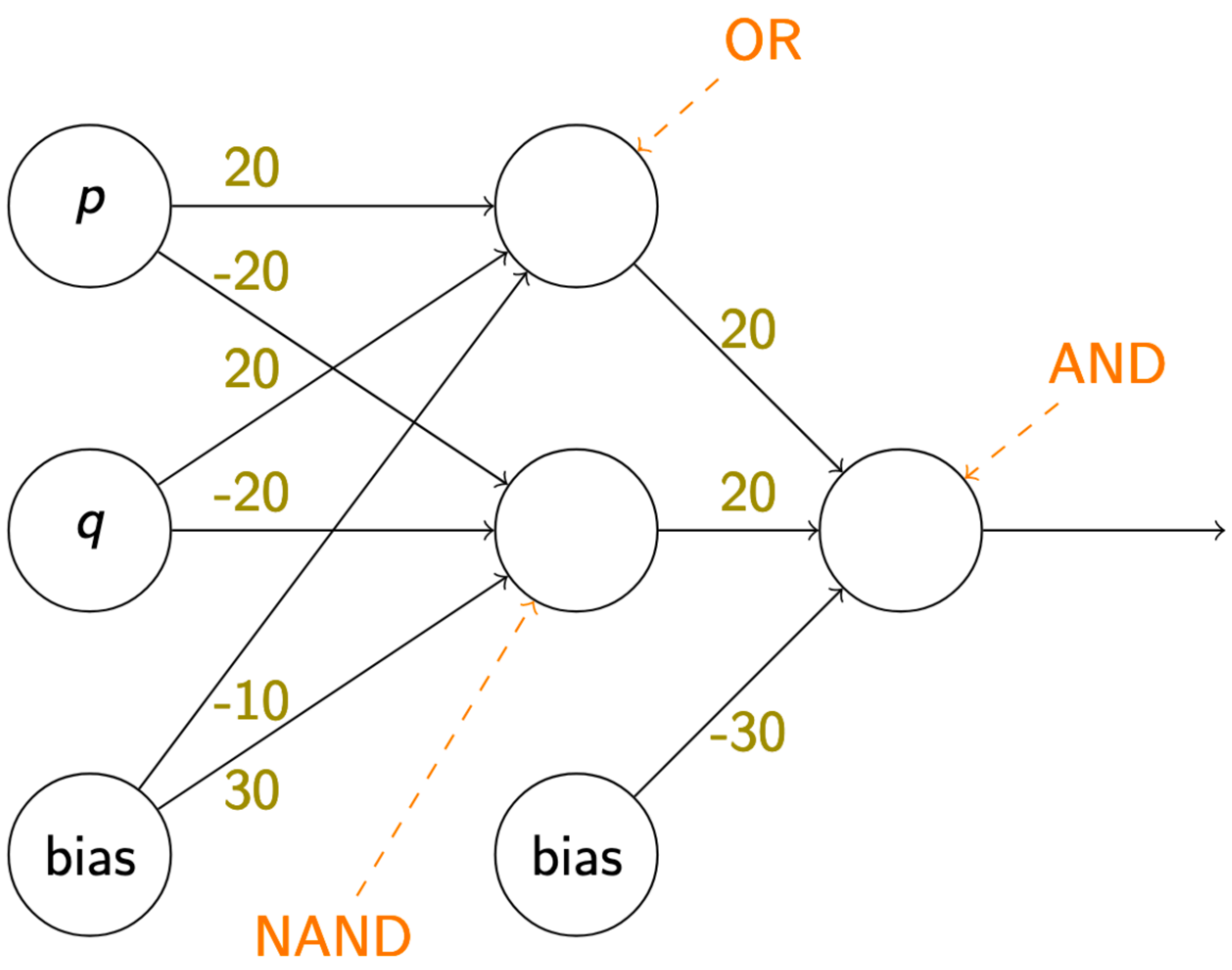
XOR Network



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

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

XOR Network

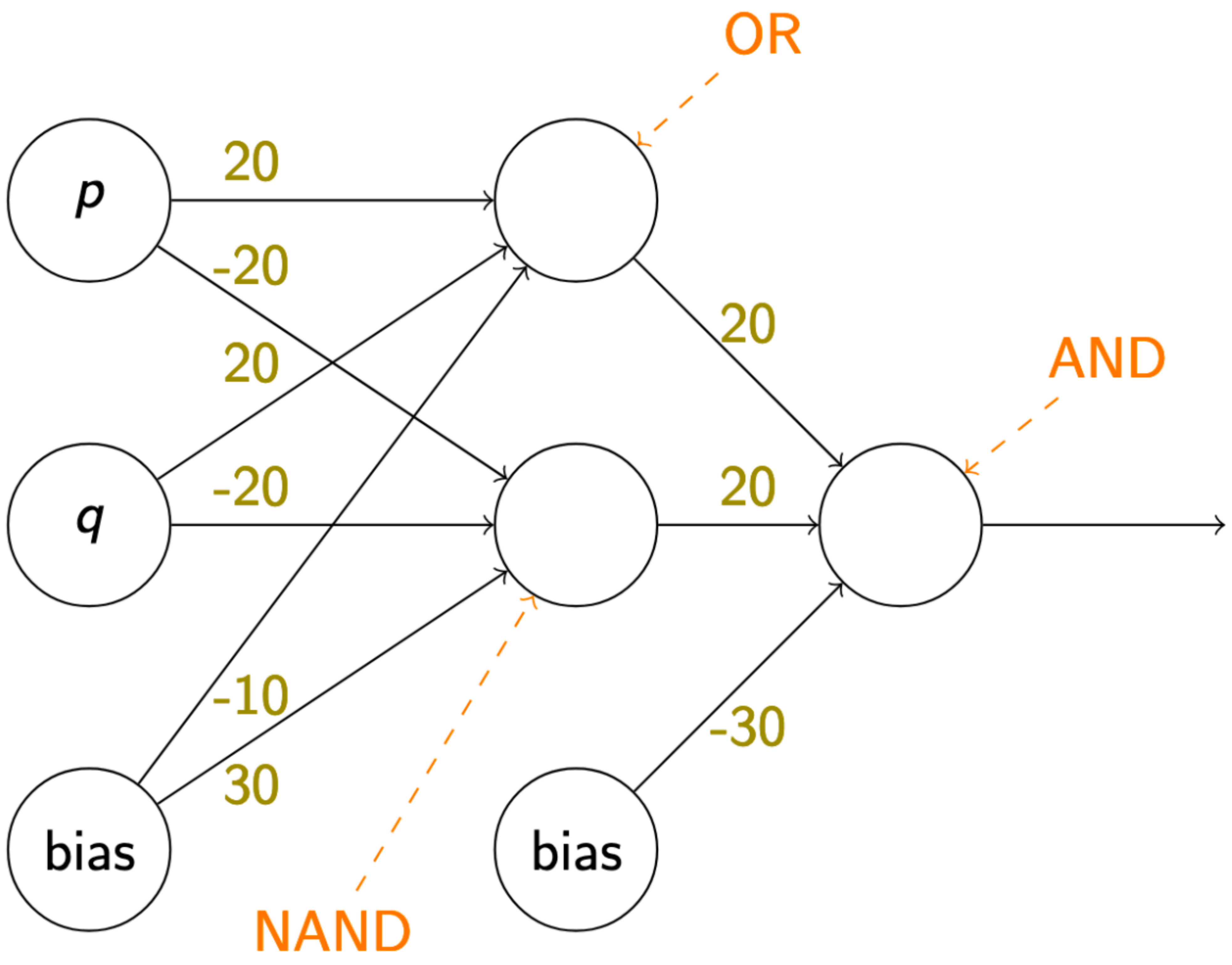


$$a_{\text{or}} = \sigma\left(\begin{bmatrix} w_p^{\text{or}} & w_q^{\text{or}} \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix} + b^{\text{or}}\right)$$

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

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

XOR Network

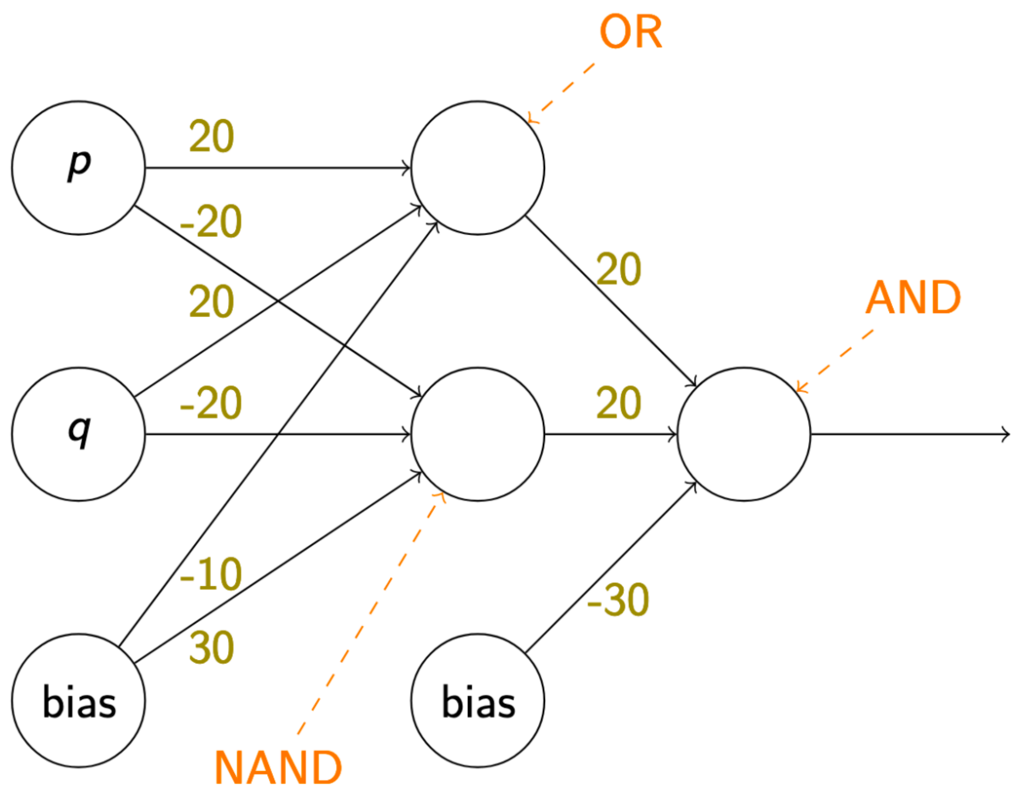


$$a_{\text{or}} = \sigma\left(\begin{bmatrix} w_p^{\text{or}} & w_q^{\text{or}} \end{bmatrix} \begin{bmatrix} p \\ q \end{bmatrix} + b^{\text{or}}\right)$$

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

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

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



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

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

Generalizing

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

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

Generalizing

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

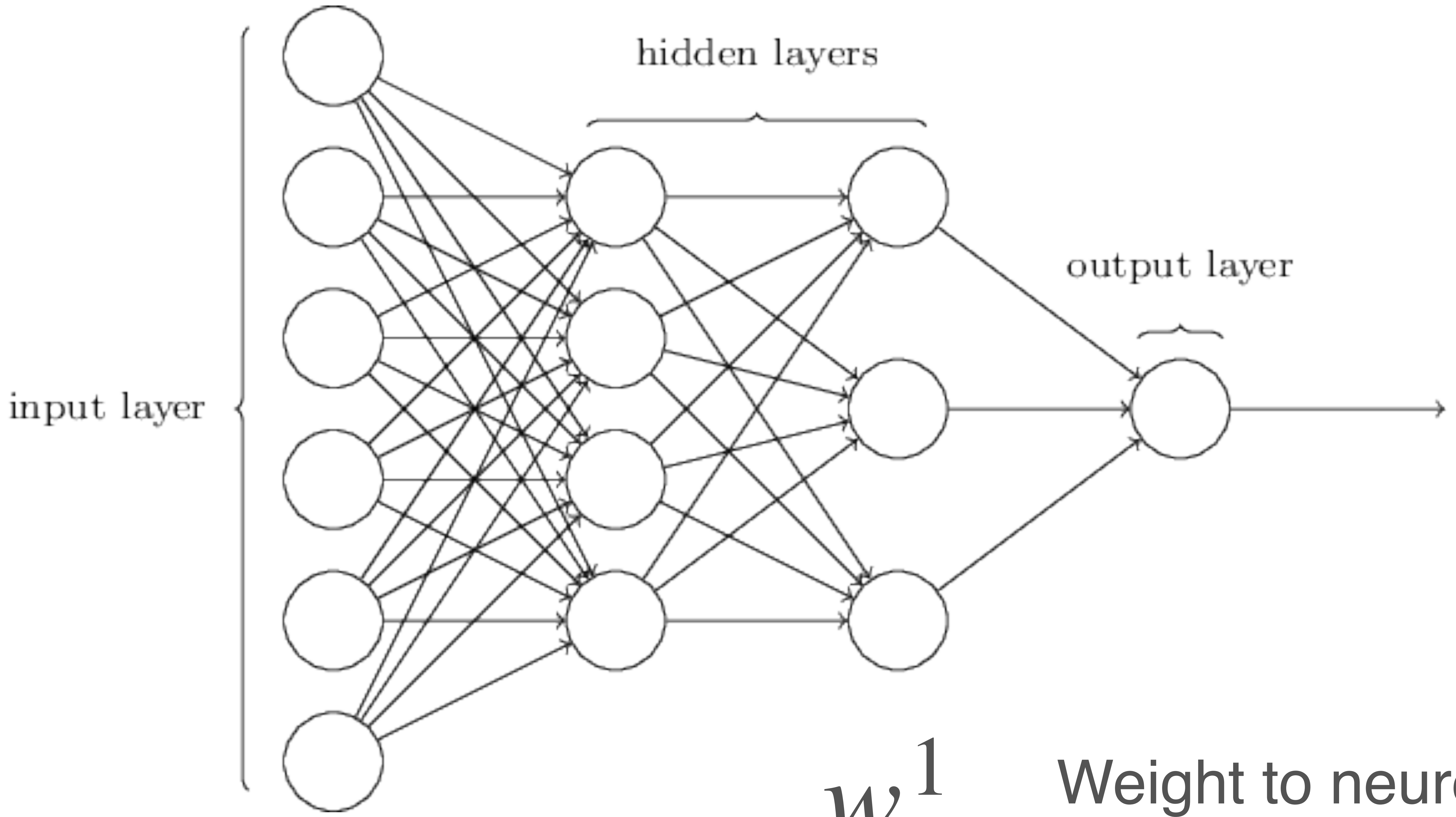
$$\hat{y} = f_2\left(W^2 \cdot f_1\left(W^1 x + b^1\right) + b^2\right)$$

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

Some terminology

- Our XOR network is a **feed-forward** neural network with **one hidden layer**
  - Also called a **Multi-Layer Perceptron (MLP)**
- 2 input nodes
- 1 output node
- 1 hidden layer with 2 neurons
- Sigmoid activation function

General MLP

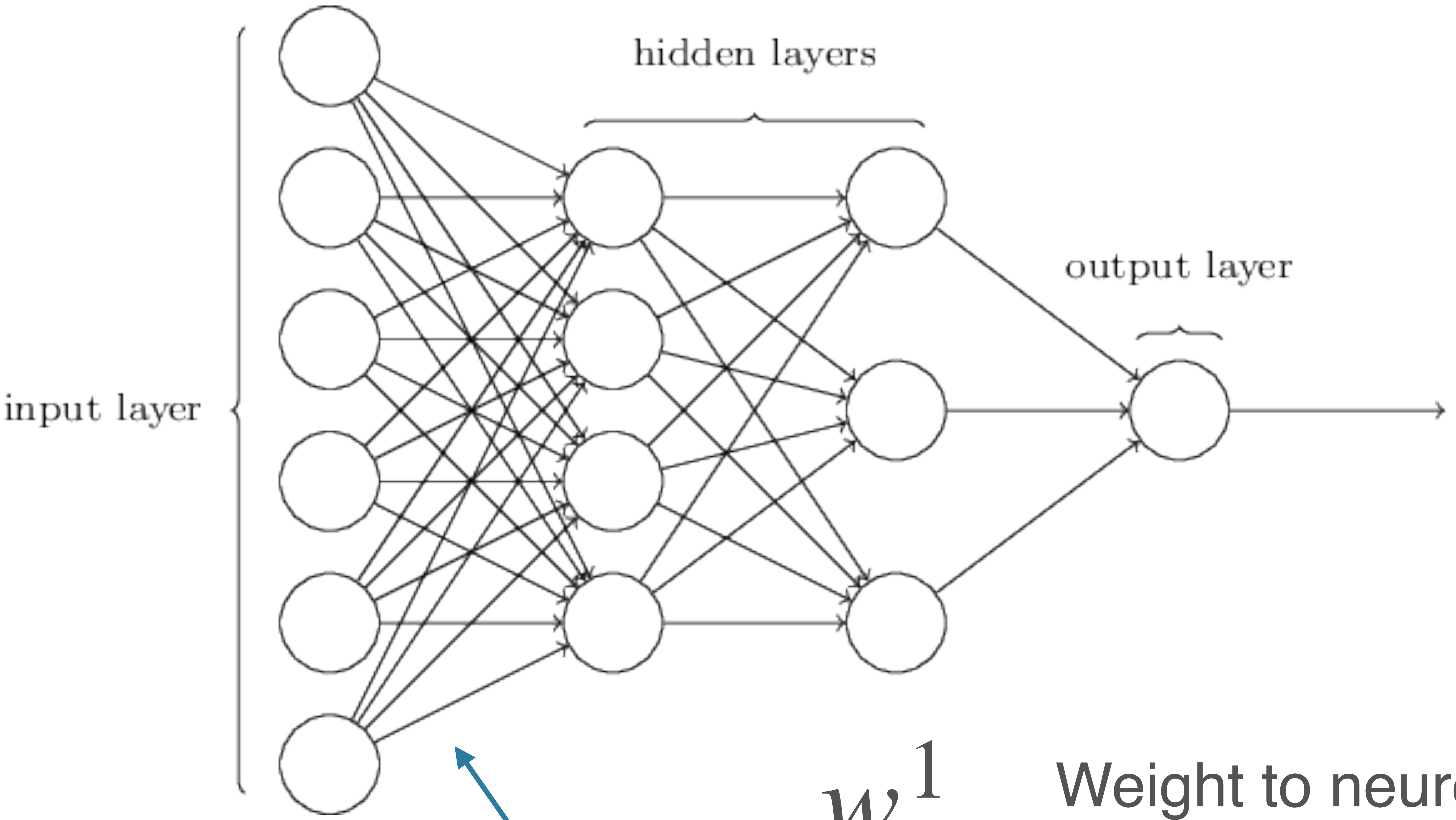


source

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

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

General MLP



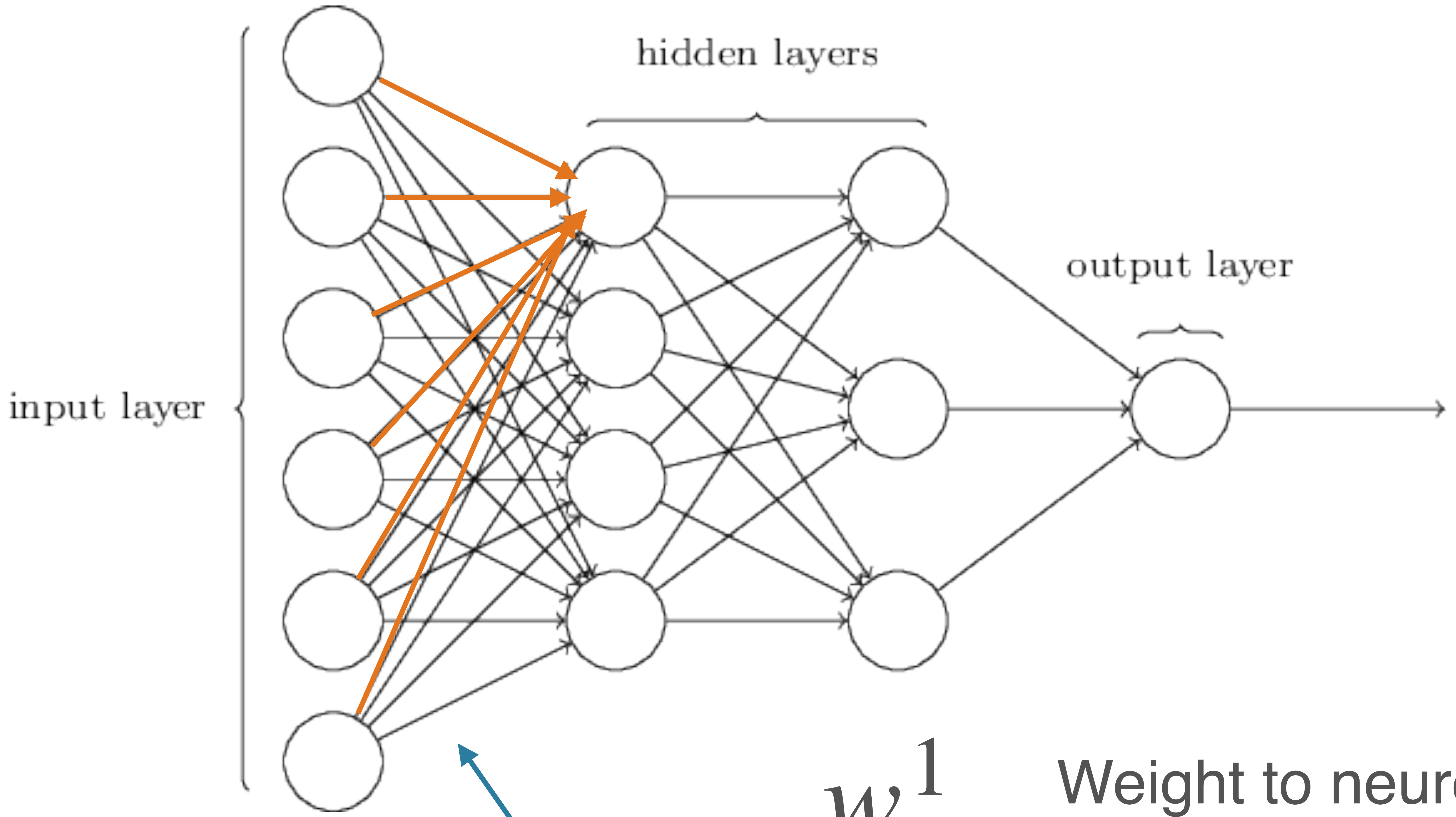
source

W^1

w_{ji}^1

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

General MLP



source

W^1

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

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

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

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: $(n_0, 1)$

General MLP

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

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

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

shape: $(n_0, 1)$

shape: (n_1, n_0)

n_0: dimension of input (layer 0)

n_1: output dimension of layer 1

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

shape: $(n_1, 1)$

Parameters of an MLP

- Weights and biases
  - For **each layer** l: $n_l(n_{l-1} + 1)$
  - $n_l \cdot n_{l-1}$ weights; n_l biases
- With *k* hidden layers (considering the output as a hidden layer):
  - $\sum_{i=1}^{k} n_i(n_{i-1} + 1)$ trainable parameters

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

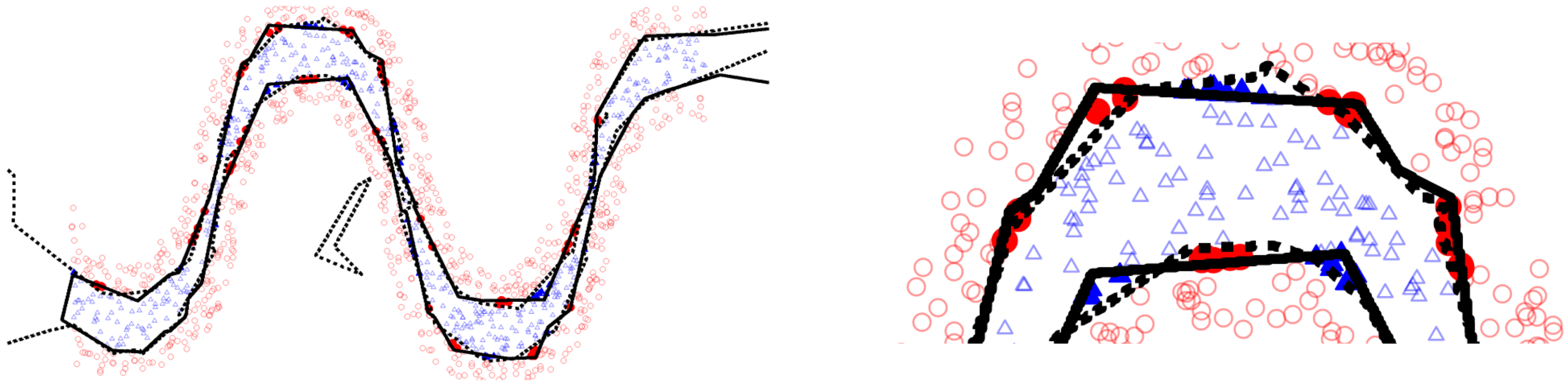
- **Input & output size**
  - Usually **fixed by your problem / dataset**
  - Input: image size, vocab size; number of “raw” features in general
  - Output: 1 for binary classification or simple regression, number of labels for classification, …
- **Number** of hidden layers
- For each hidden layer: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

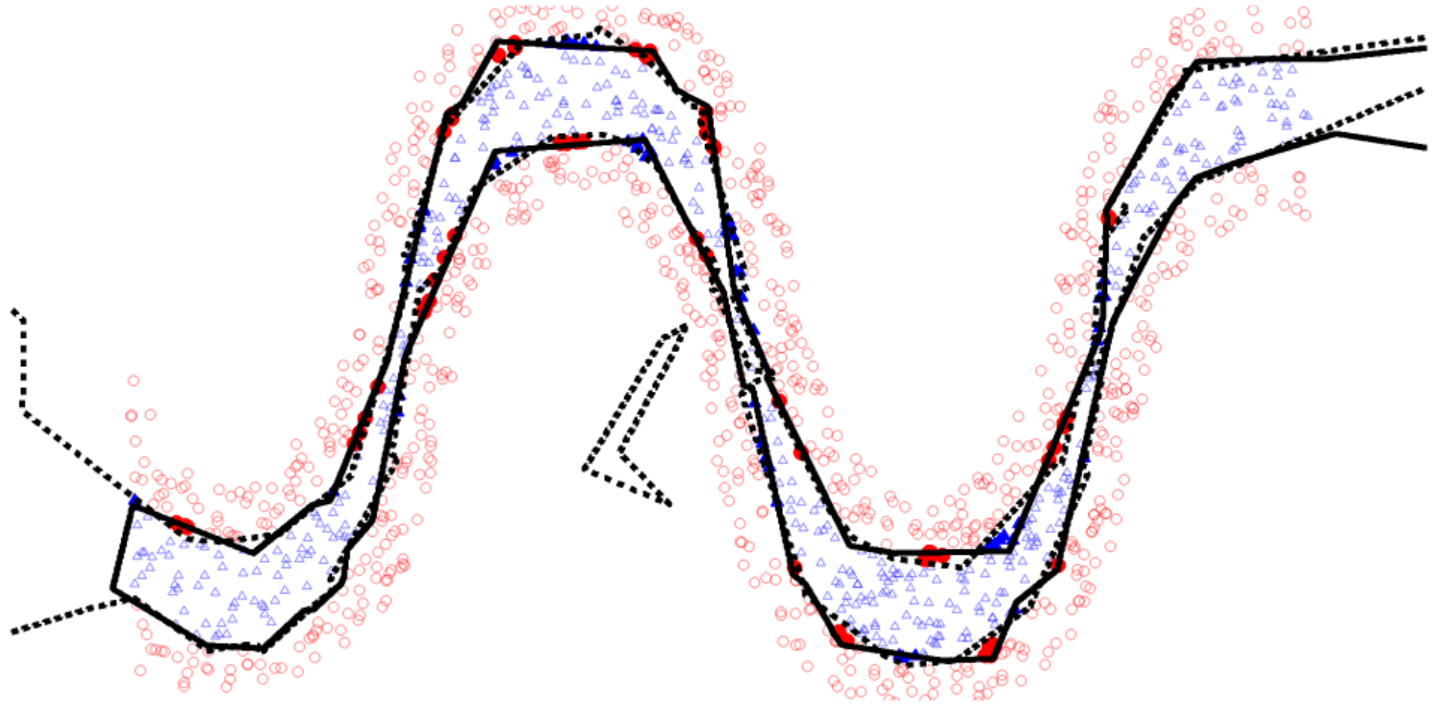


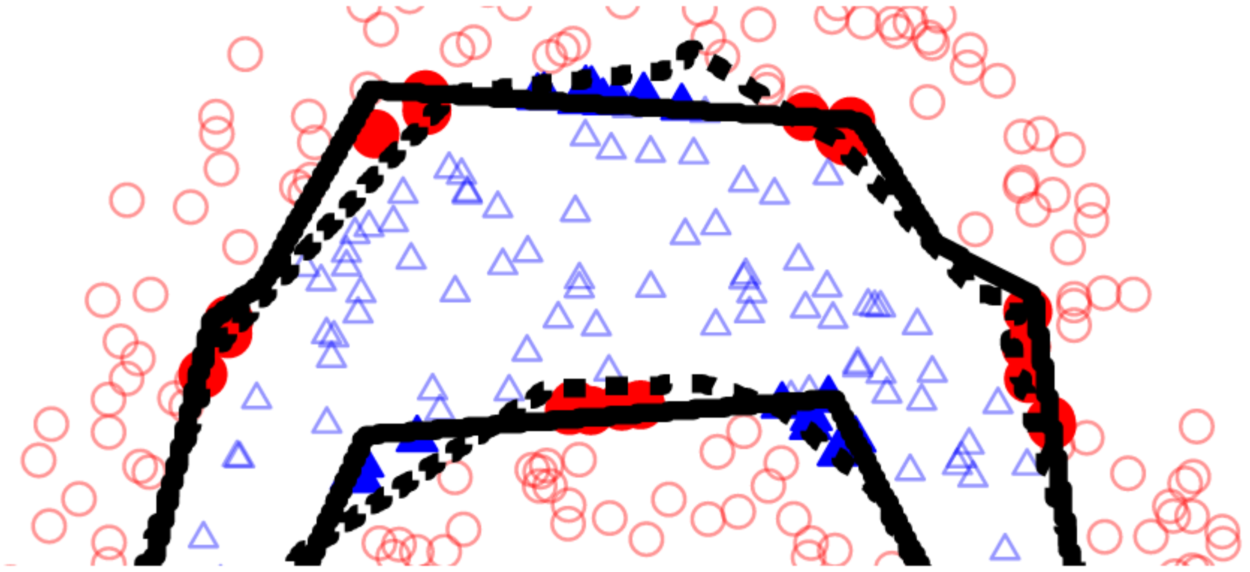
discover
source

sis)

The Deep in Deep Learning

- The Universal Approximation Theorem says that **one hidden layer suffices** for arbitrarily-closely approximating a given function
- Empirical d



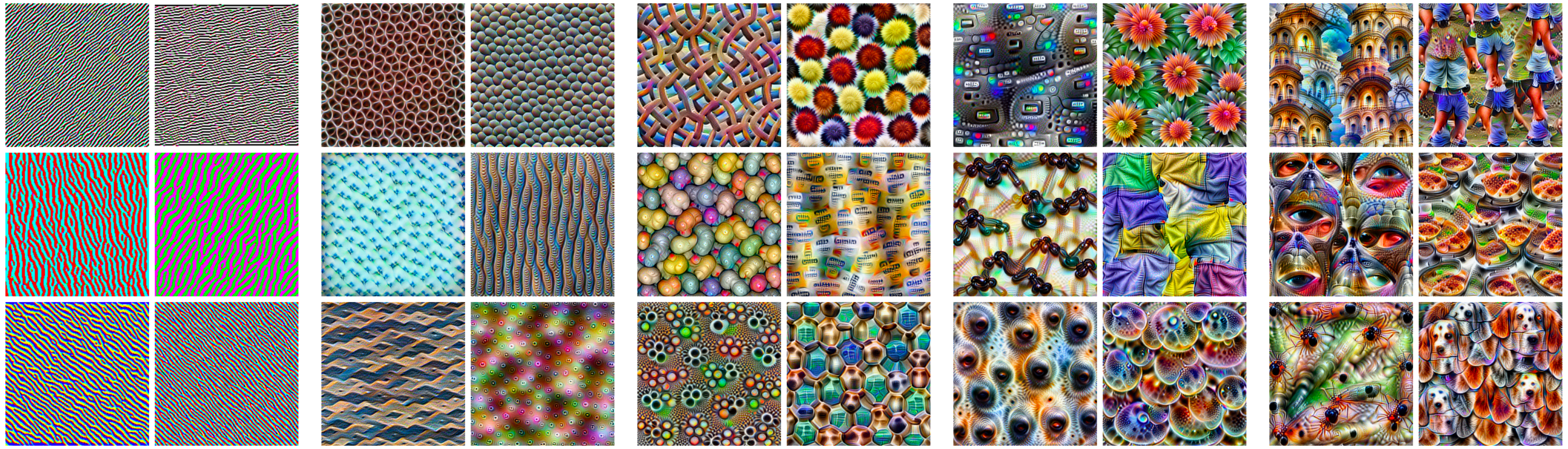


discover
source

- “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 non-linearity, repeated several times over
- Without the non-linearity, would just have several linear transformations
  - **Composition** of linear transformations is **also linear!**

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

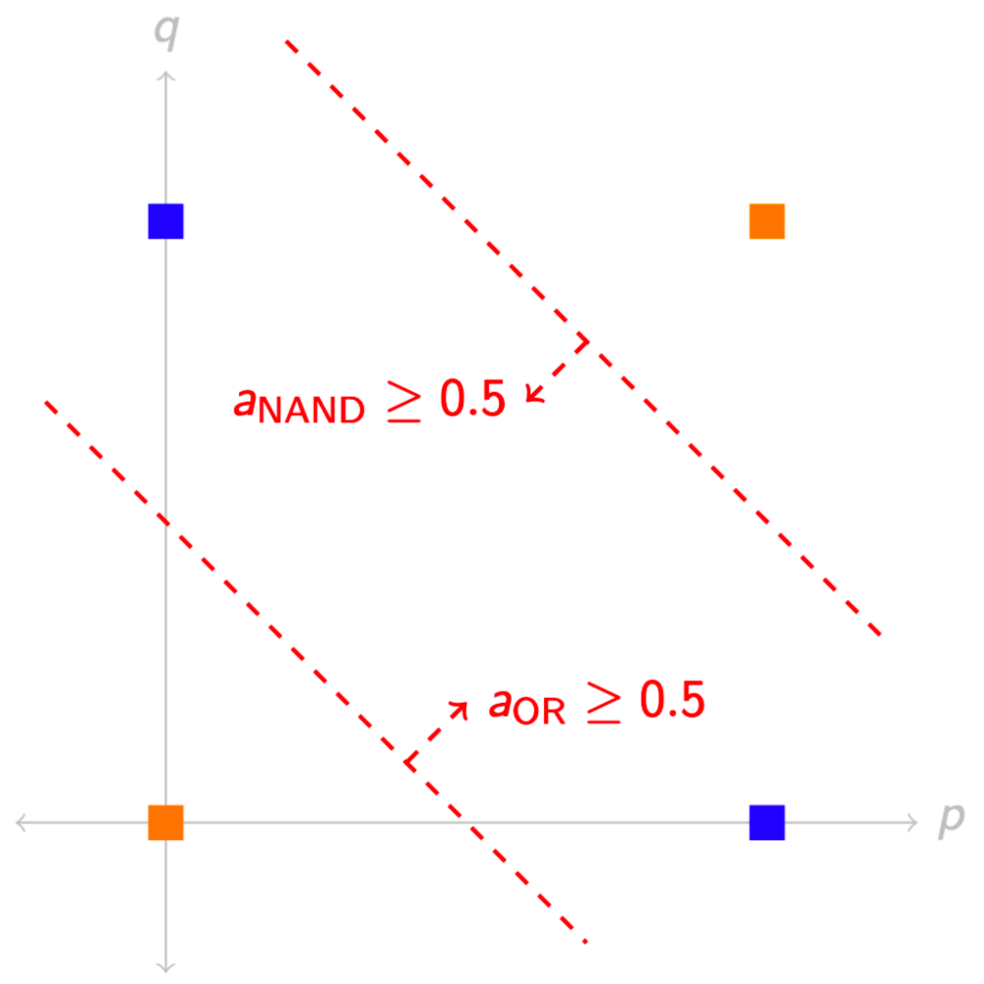
Non-linearity, cont.

Non-linearity, cont.

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

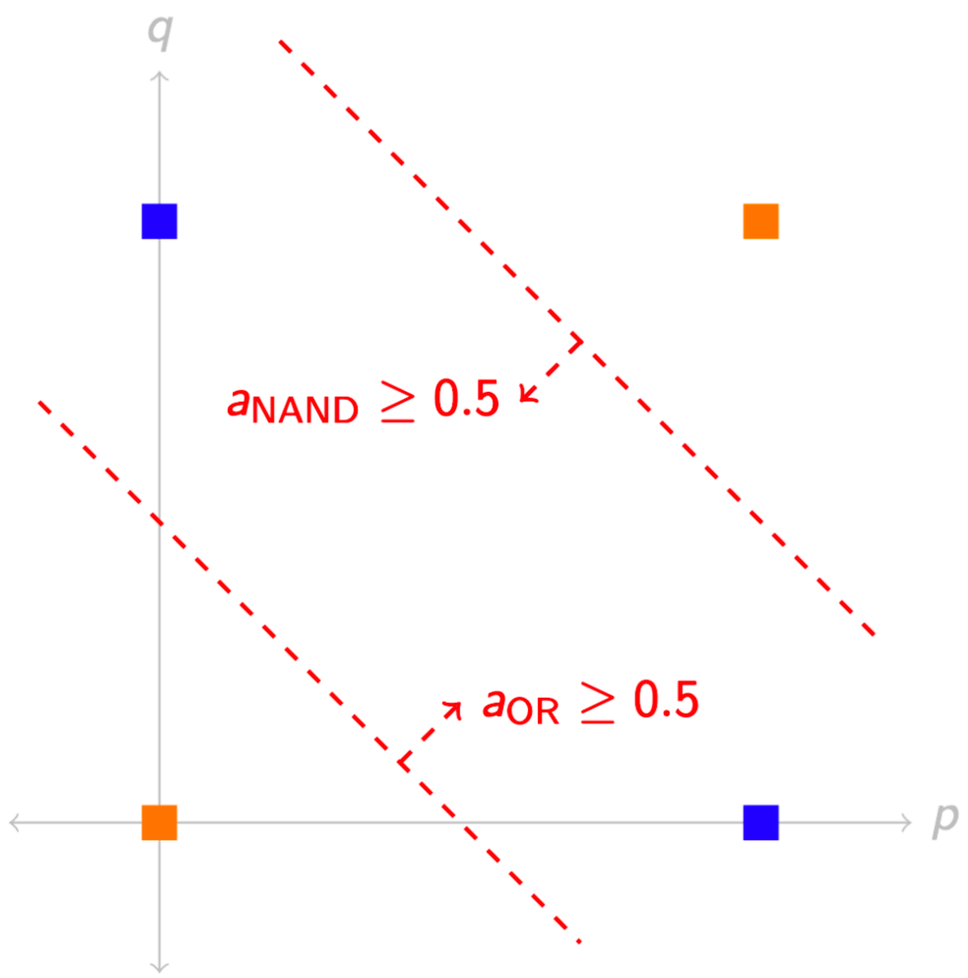
Non-linearity, cont.

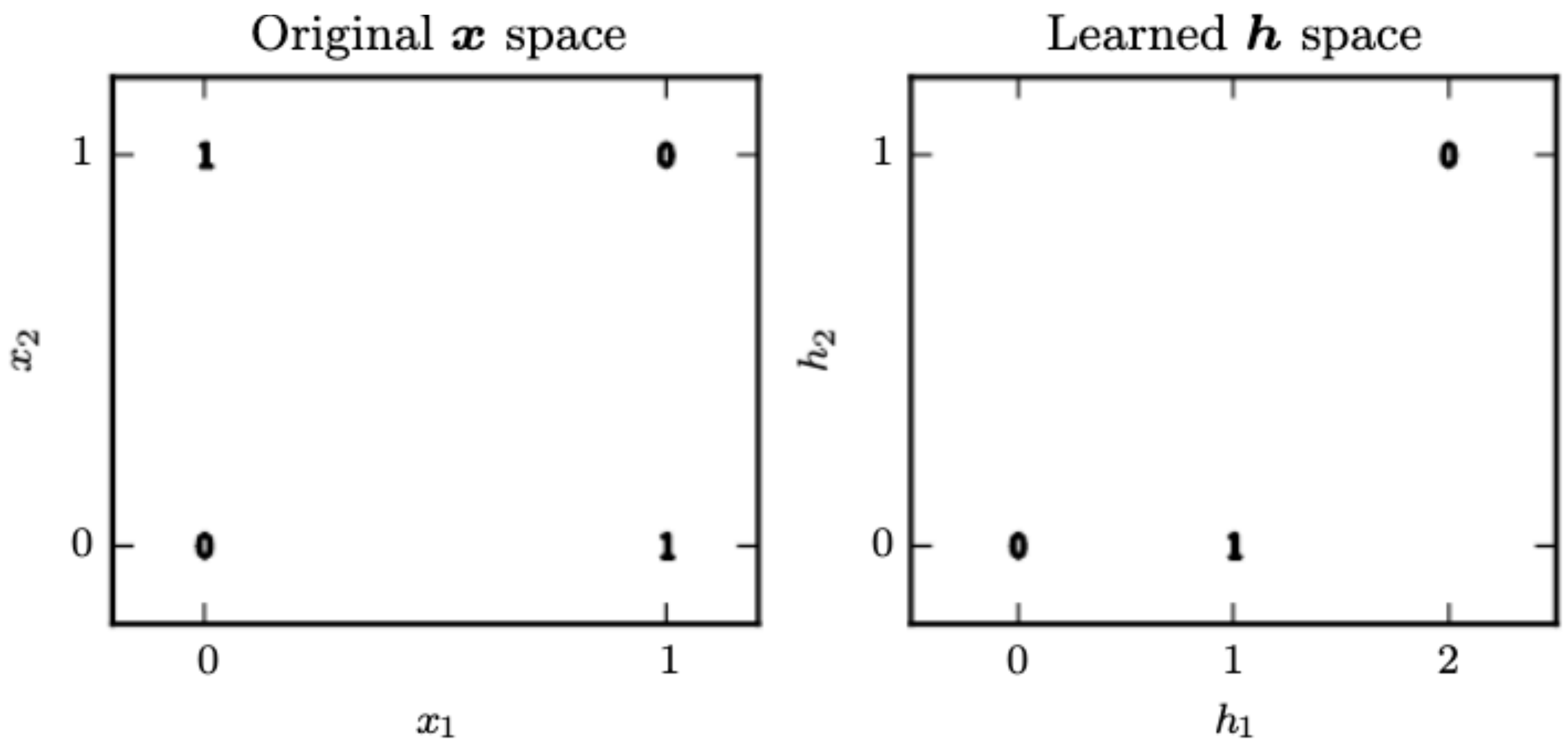
- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions
- One perspective: integrating extracted features



Non-linearity, cont.

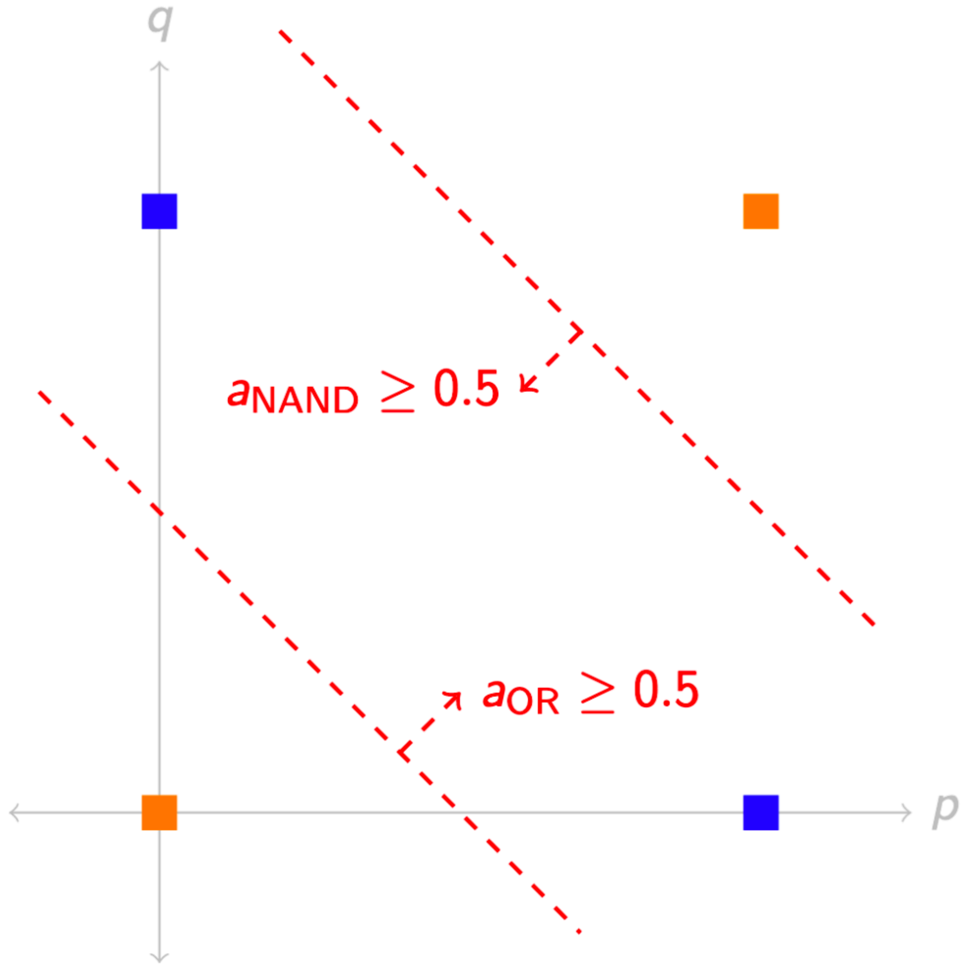
- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions
- One perspective: integrating extracted features
- An equivalent perspective:
  - Transforming the input space (source; p. 169)
  - This is a *non-linear* transformation
  - Space folding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

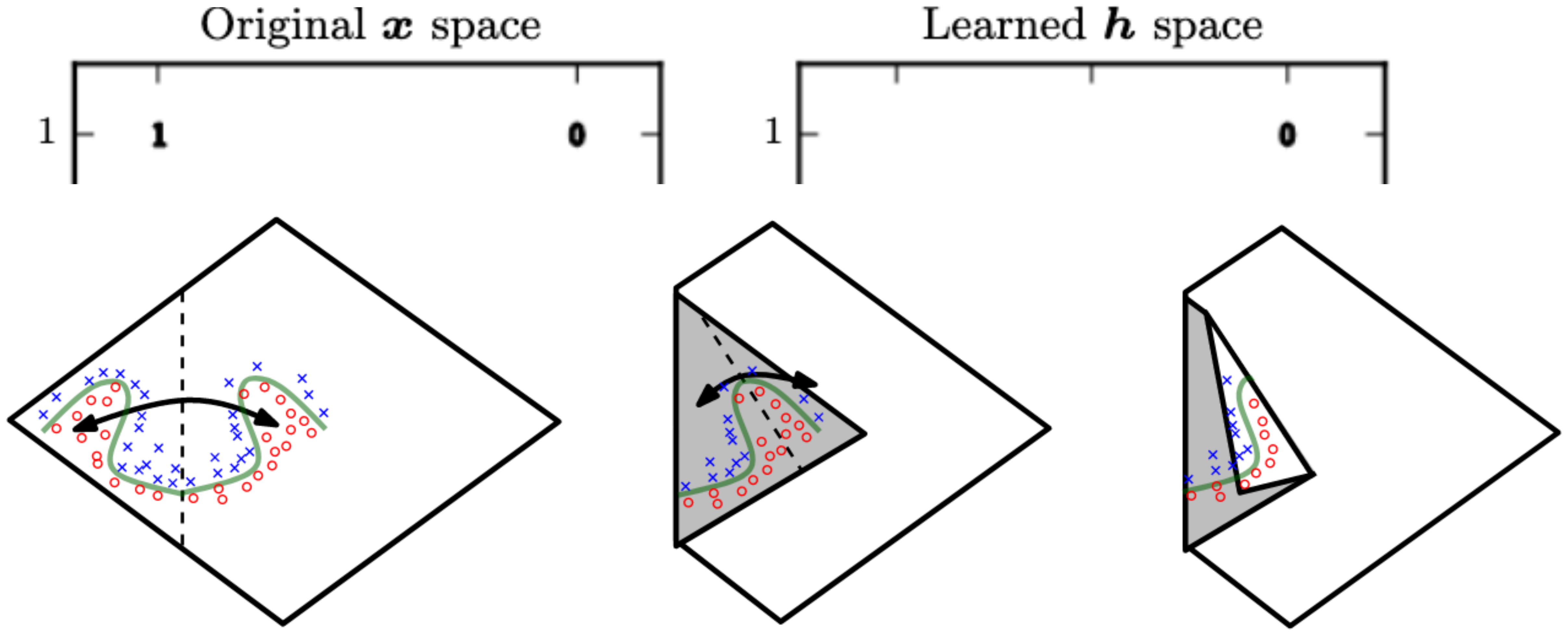




Non-linearity, cont.

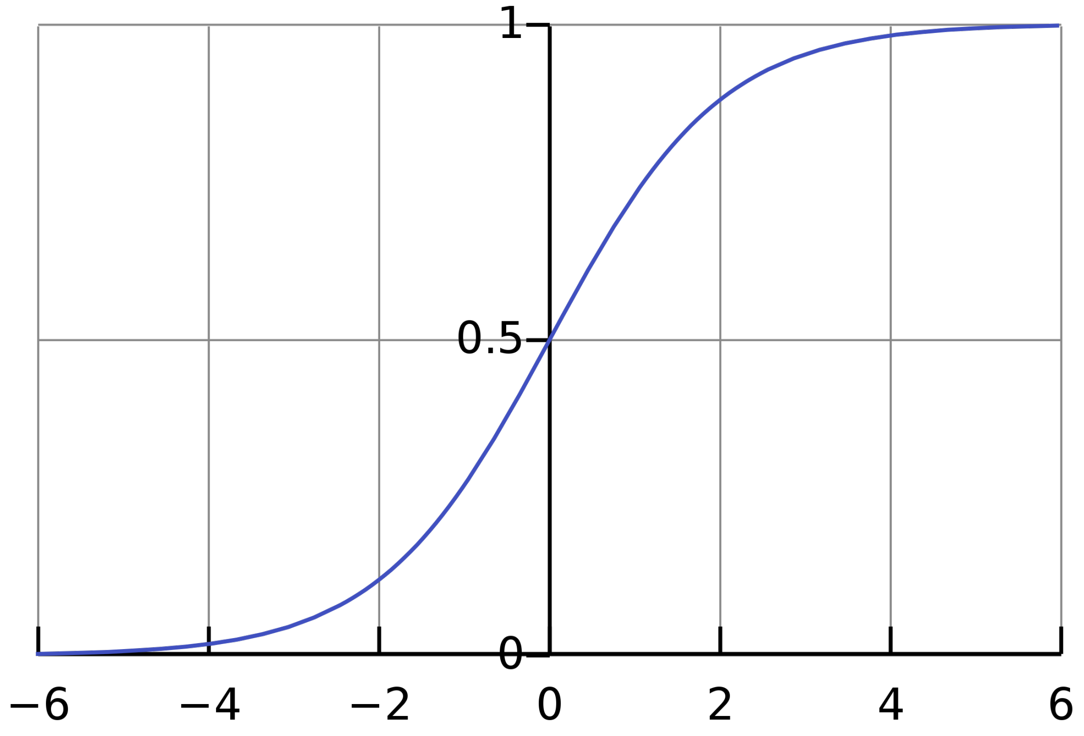
- Recall: XOR was not computable by a single neuron because the latter can only compute *linearly separable* functions
- One perspective: integrating extracted features
- An equivalent perspective:
  - Transforming the input space (source; p. 169)
  - This is a *non-linear* transformation
  - Space folding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





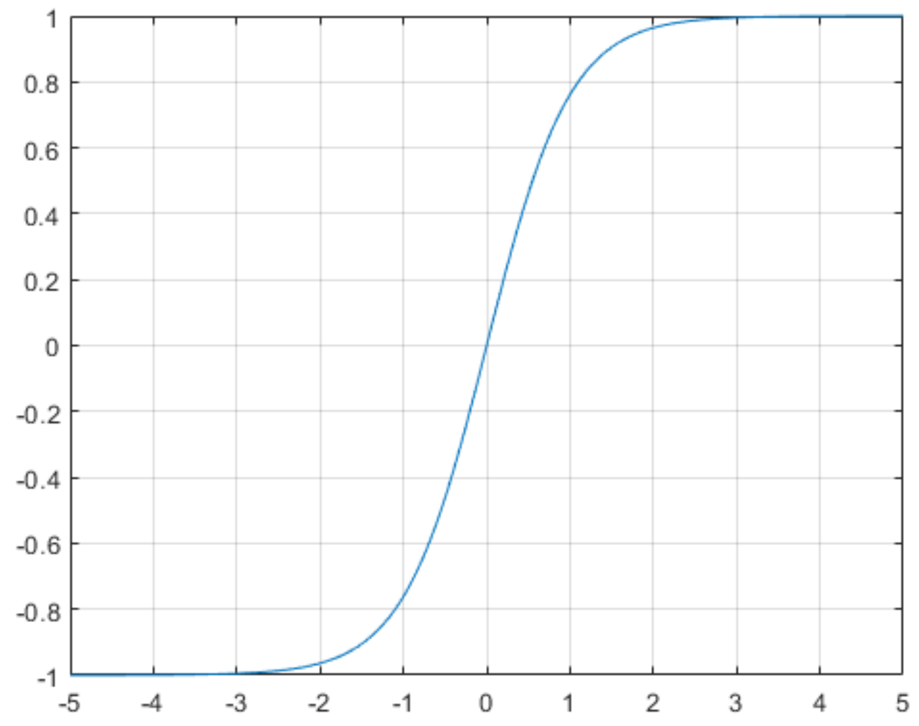
Activation Functions: Hidden Layer

sigmoid



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

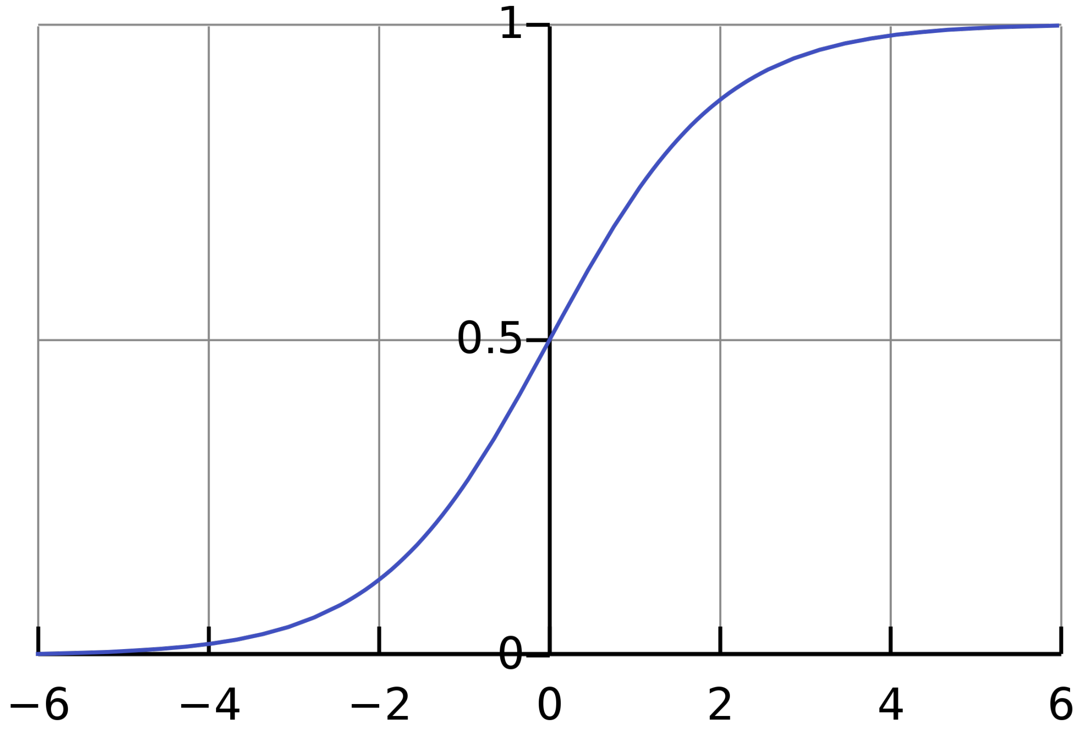
tanh



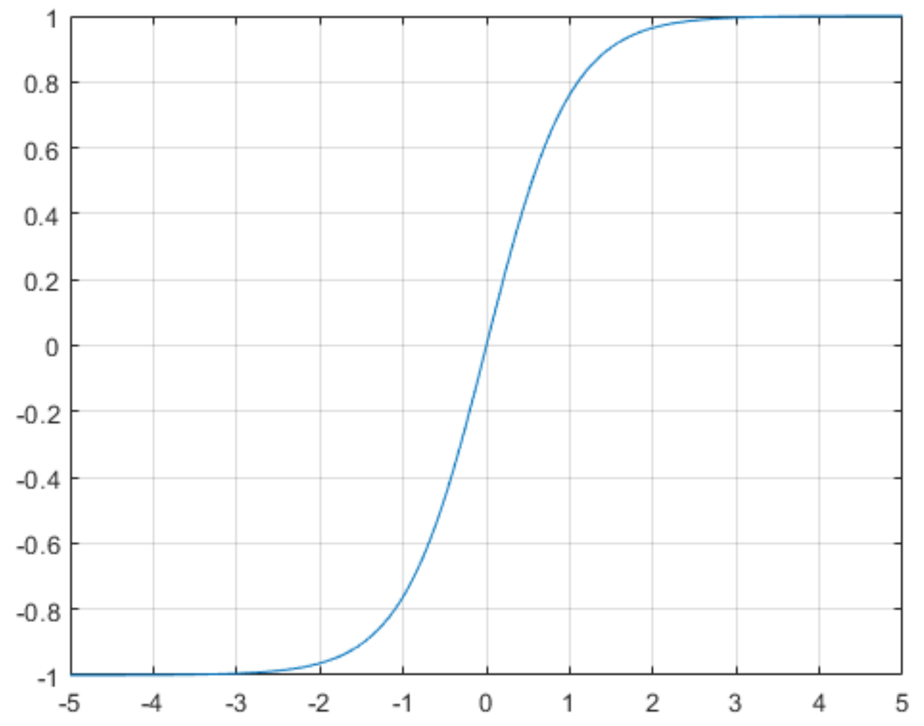
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

Activation Functions: Hidden Layer

sigmoid



tanh



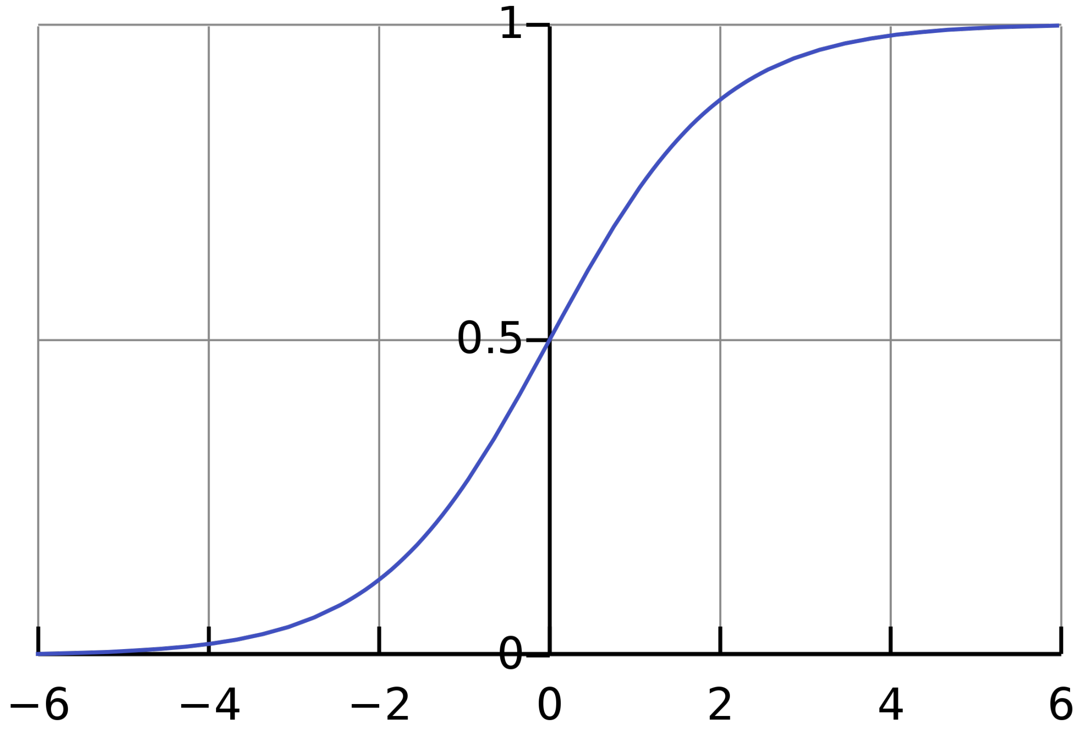
$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{e^x}{e^x+1}$$

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

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

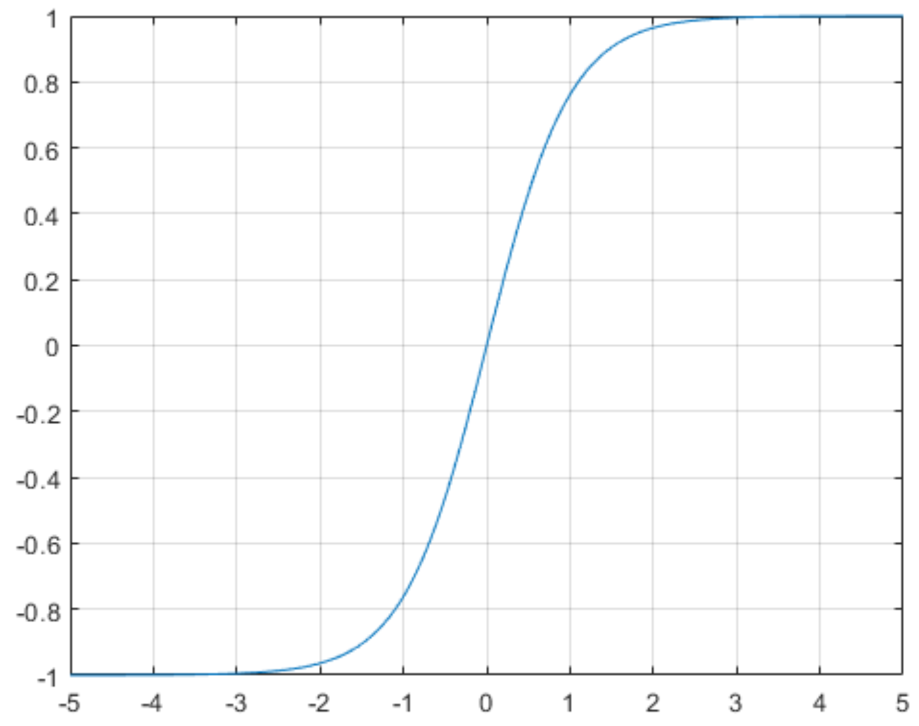
Activation Functions: Hidden Layer

sigmoid

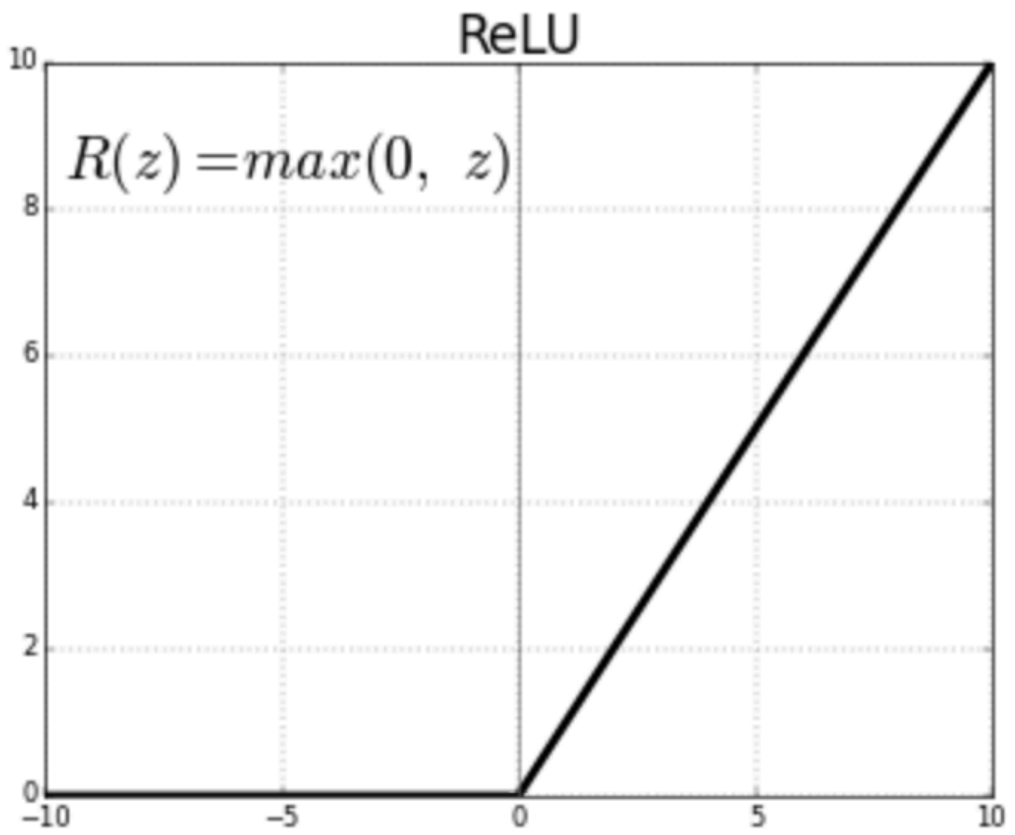


$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1}$$

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

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_1, n_0)
n_0: dimension of input (layer 0)
n_1: output dimension of layer 1

Shape: $(n_0, 1)$

Computing with a Single Input

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

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

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: $(n_1, 1)$

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 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

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

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

Shape: (n_0, k)
k: batch_size

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

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

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

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 $W^1 X$

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)

Neural Probabilistic Language Model

Language Modeling

- A language model parametrized by θ computes: $P_\theta(w_1, \ldots, w_n)$
- Typically (though we'll see variations): $P_\theta(w_1, \ldots, w_n) = \prod_i P_\theta(w_i \mid w_1, \ldots, w_{i-1})$
- E.g. of labeled data: "Today is the seventh day of 282." —>
  - (<s>, Today)
  - (<s> Today, is)
  - (<s> Today is, the)
  - (<s> Today is the, seventh)

N-gram LMs

- Dominant approach for a long time uses **n-grams**:

$$P_\theta(w_i \mid w_1, \ldots, w_{i-1}) \approx P_\theta(w_i \mid w_{i-1}, w_{i-2}, \ldots, w_{i-n})$$

- Estimate the probabilities by **counting** in a corpus
  - Fancy variants (back-off, smoothing, etc)
- Some problems:
  - **Huge number of parameters**: $\approx |V|^n$
  - Doesn't generalize to unseen n-grams

Neural LM

- Core idea behind the Neural Probabilistic LM
  - Make **n-gram assumption**
  - But: learn **word embeddings**
  - “n-gram of word vectors”
  - Probabilities represented by a neural network, not counts

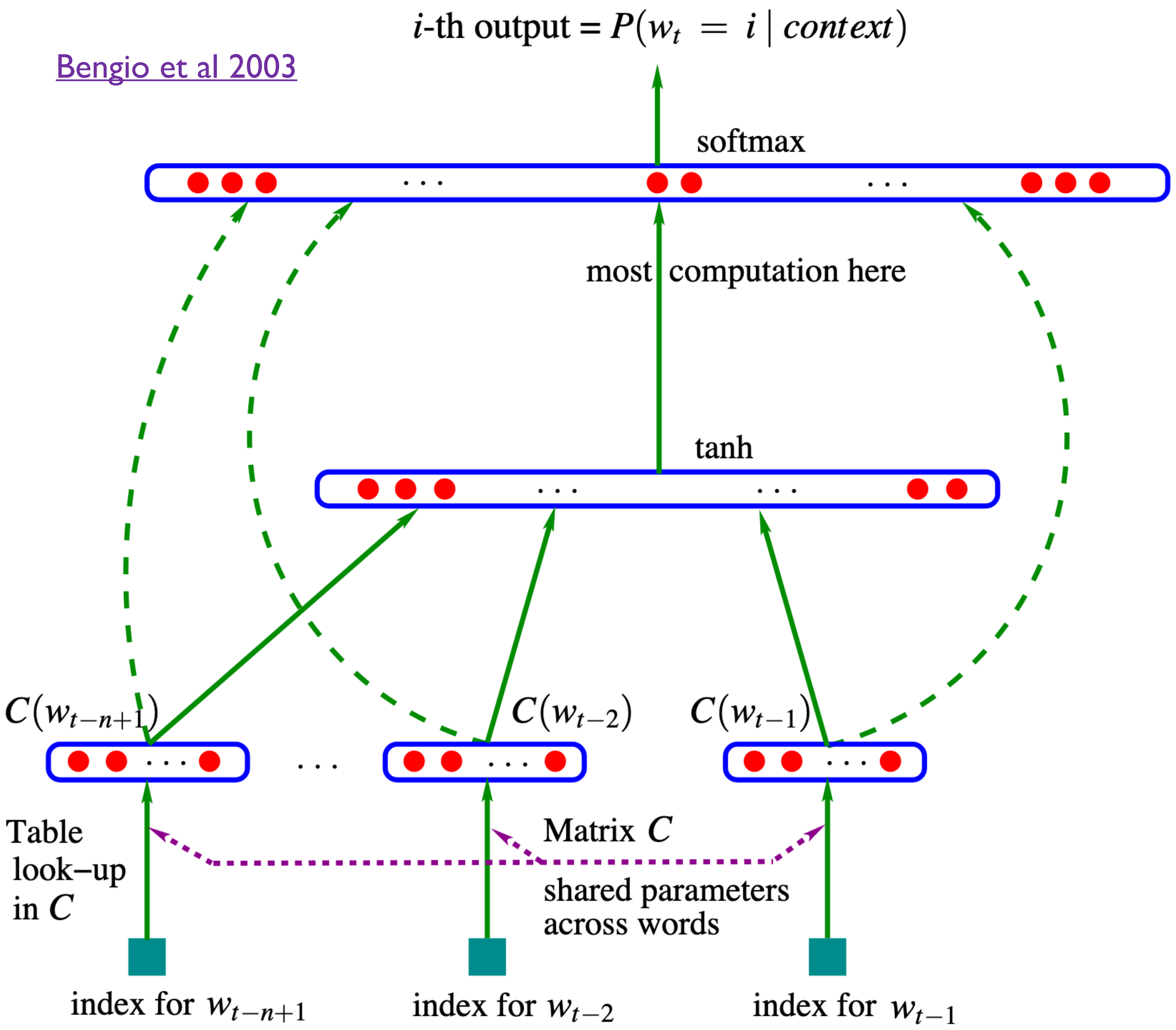
Pros of Neural LM

- Number of parameters:
  - Significantly lower, thanks to **“low”-dimensional embeddings**
- Generalization: embeddings enable **generalizing to similar words**

```
                    The cat is walking in the bedroom
to                  A dog was running in a room
and likewise to     The cat is running in a room
                    A dog is walking in a bedroom
                    The dog was walking in the room
```

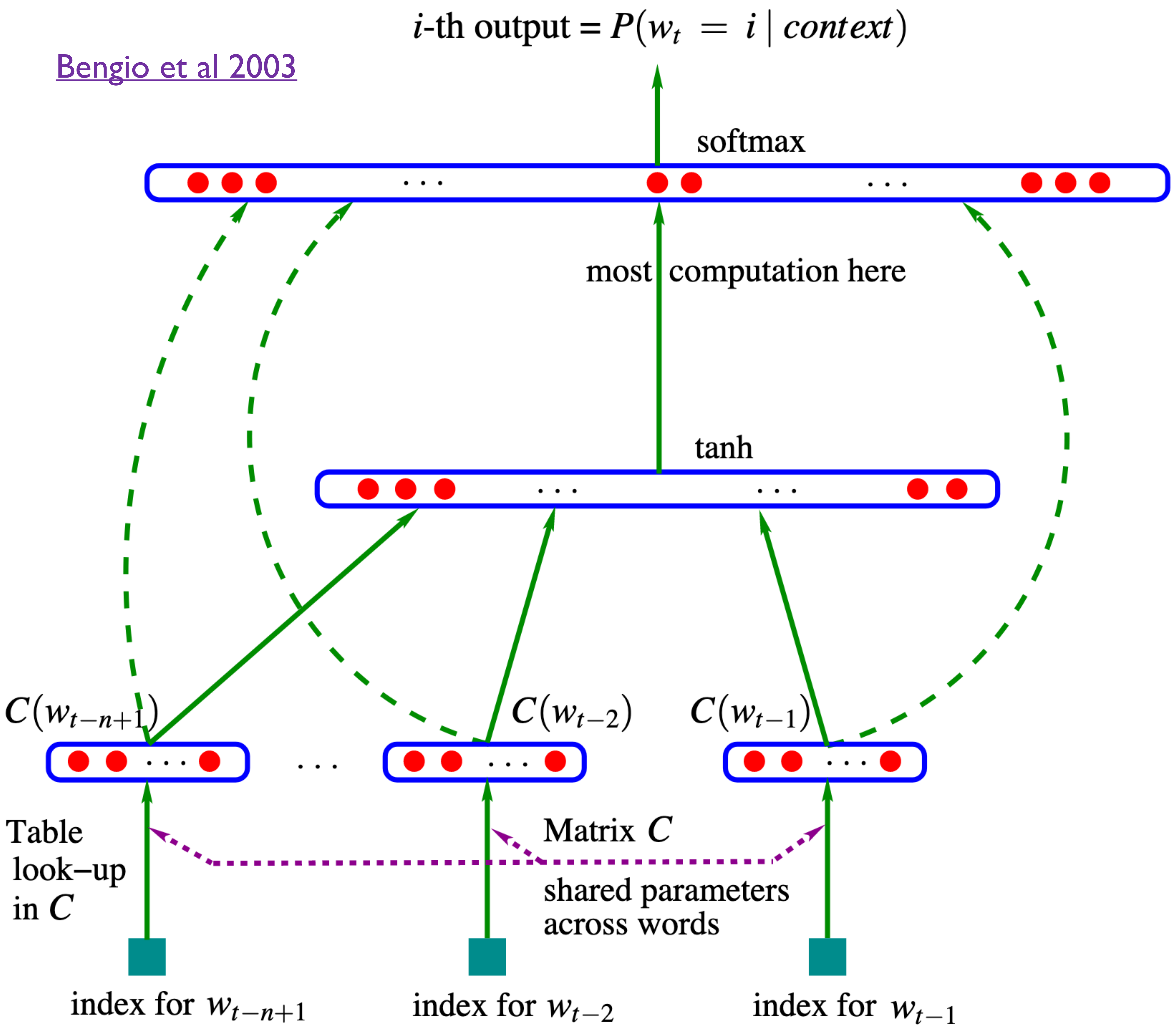
Neural LM Architecture

Bengio et al 2003



Neural LM Architecture

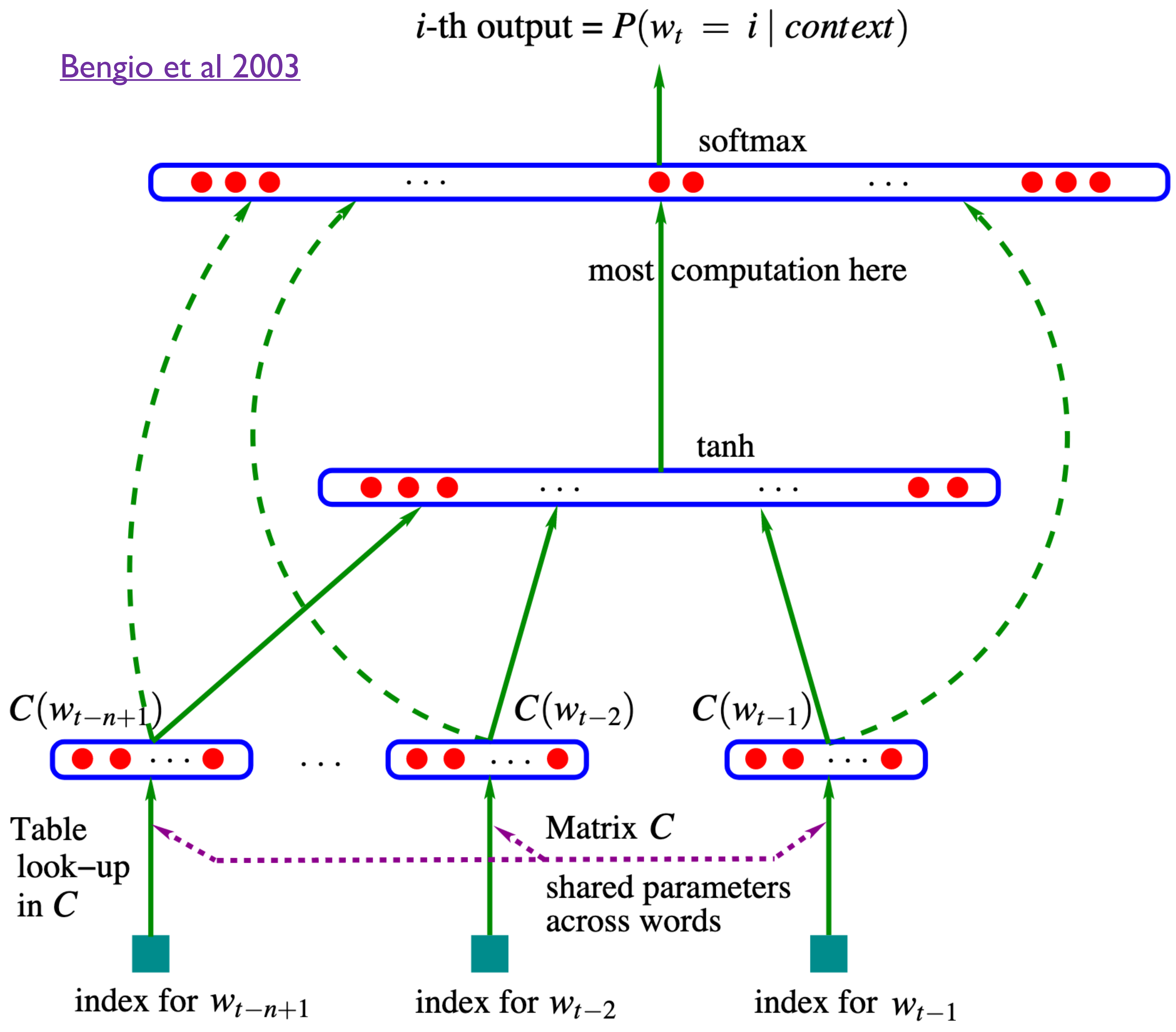
Bengio et al 2003



w_t: one-hot vector

Neural LM Architecture

Bengio et al 2003

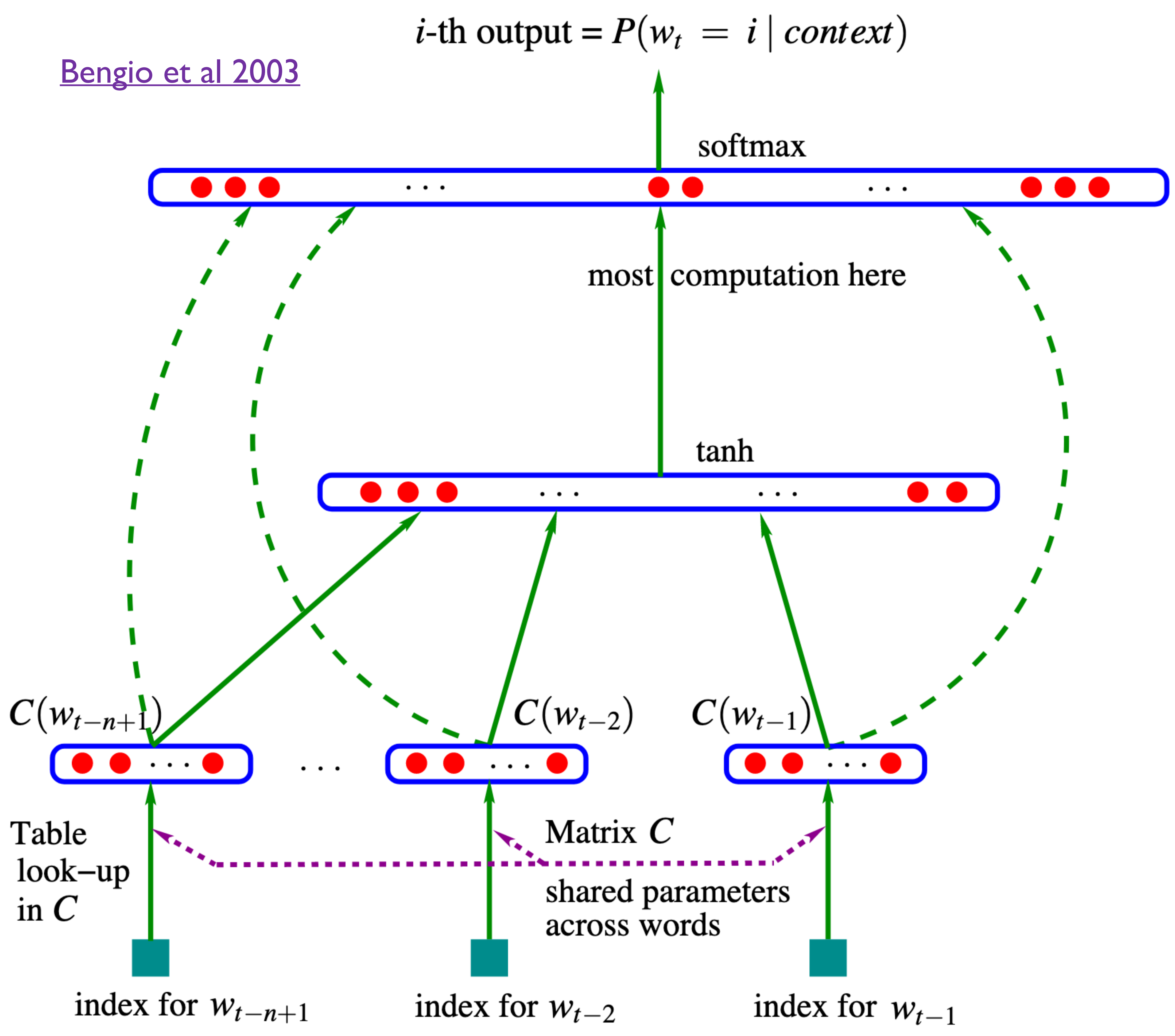


embeddings = concat$(Cw_{t-1}, Cw_{t-2}, \ldots, Cw_{t-(n+1)})$

w_t: one-hot vector

Neural LM Architecture

Bengio et al 2003

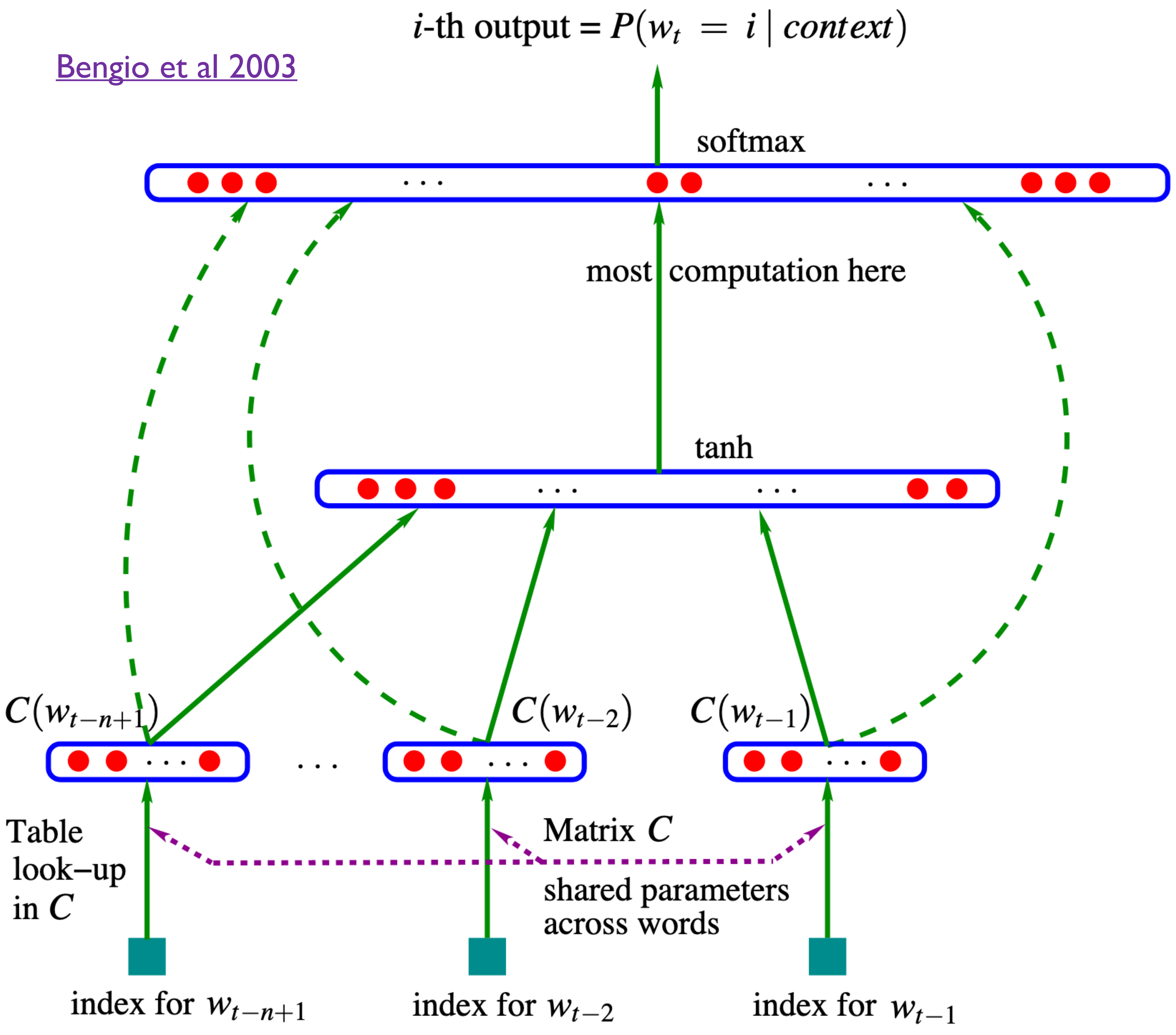


$$\text{hidden} = \tanh(W^1 \cdot \text{embeddings} + b^1)$$

$$\text{embeddings} = \text{concat}(Cw_{t-1}, Cw_{t-2}, \ldots, Cw_{t-(n+1)})$$

w_t: one-hot vector

Neural LM Architecture



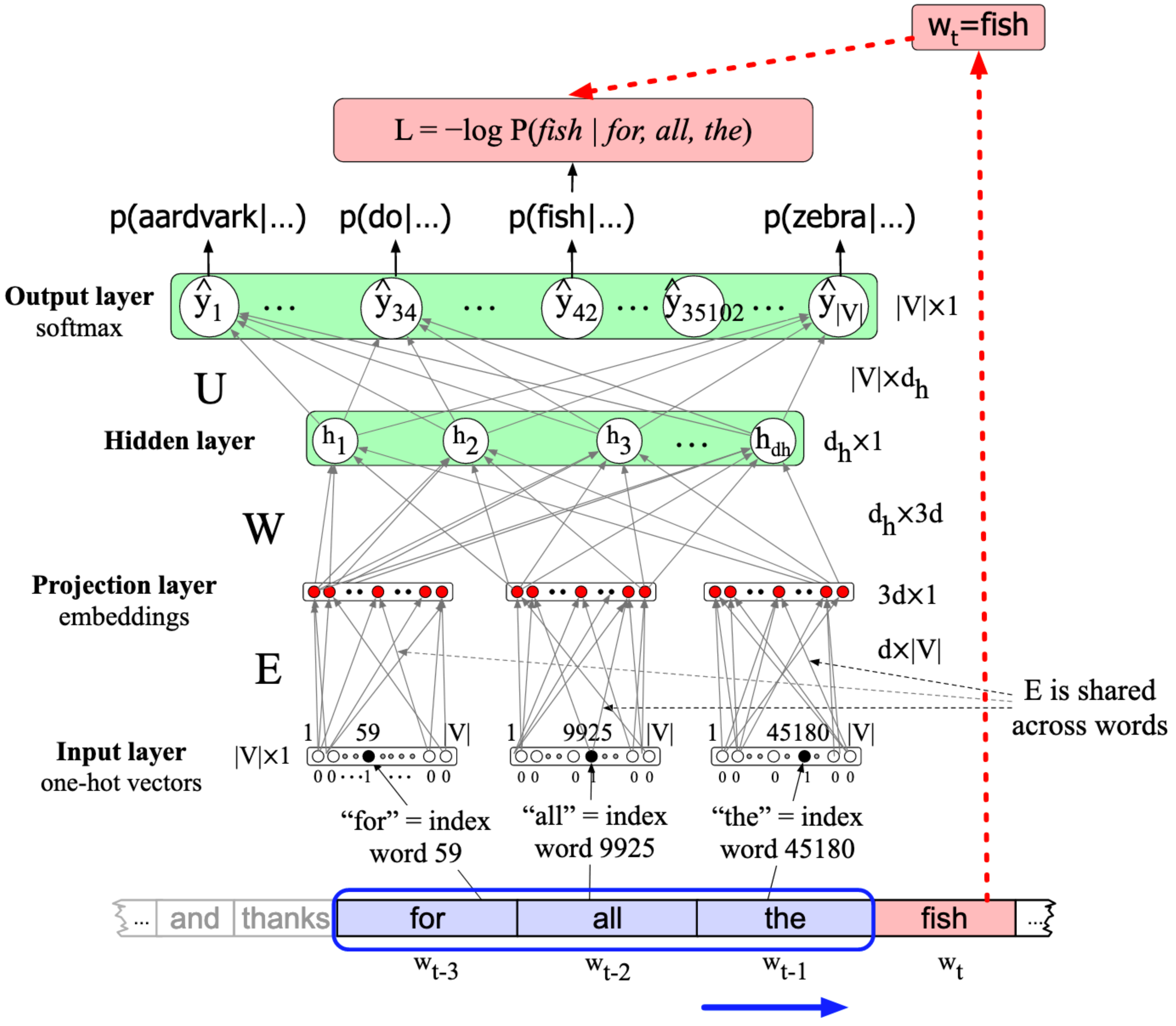
$$\text{probabilities} = \text{softmax}(W^2 \cdot \text{hidden} + b^2)$$

$$\text{hidden} = \tanh(W^1 \cdot \text{embeddings} + b^1)$$

$$\text{embeddings} = \text{concat}(Cw_{t-1}, Cw_{t-2}, \ldots, Cw_{t-(n+1)})$$

w_t: one-hot vector

More Detailed Diagram of Architecture



JM sec 7.5

Output and Loss for Classification

$$\text{logits} = W \cdot \text{hidden} + b$$

$$\hat{y} = \text{probs} = \text{softmax}(\text{logits})$$

$$\ell_{CE}(\hat{y}, y) = -\sum_{i=0}^{|\text{classes}|} y_i \log \hat{y}_i$$

One hot for true class label

Evaluation of LMs

- **Extrinsic**: use in other NLP systems
- **Intrinsic**: intuitively, want probability of a test corpus
  - **Perplexity**: inverse probability, weighted by size of corpus
  - **Lower** is better!
  - **Only comparable w/ same vocab**

Perplexity

$$PP(W) = P(w_1 w_2 \cdots w_N)^{-1/N}$$

$$= \sqrt[N]{\frac{1}{P(w_1 w_2 \cdots w_N)}}$$

$$= \sqrt[N]{\frac{1}{\prod_{i=0}^{N} P(w_i \mid w_1, \ldots, w_{i-1})}}$$

$$= 2^{-\frac{1}{N}\sum_{i=0}^{N} \log P(w_i | w_1, \ldots, w_{i-1})}$$

More Complete Picture of This Model

Revisiting Simple Neural Probabilistic Language Models

Simeng Sun and **Mohit Iyyer**
College of Information and Computer Sciences
University of Massachusetts Amherst
`{simengsun, miyyer}@cs.umass.edu`

Abstract

Recent progress in language modeling has been driven not only by advances in neural architectures, but also through hardware and optimization improvements. In this paper, we revisit the neural probabilistic language model (NPLM) of Bengio et al. (2003), which simply concatenates word embeddings within a fixed window and passes the result through a feed-forward network to predict the next word. When scaled up to modern hardware, this model (despite its many limitations) performs

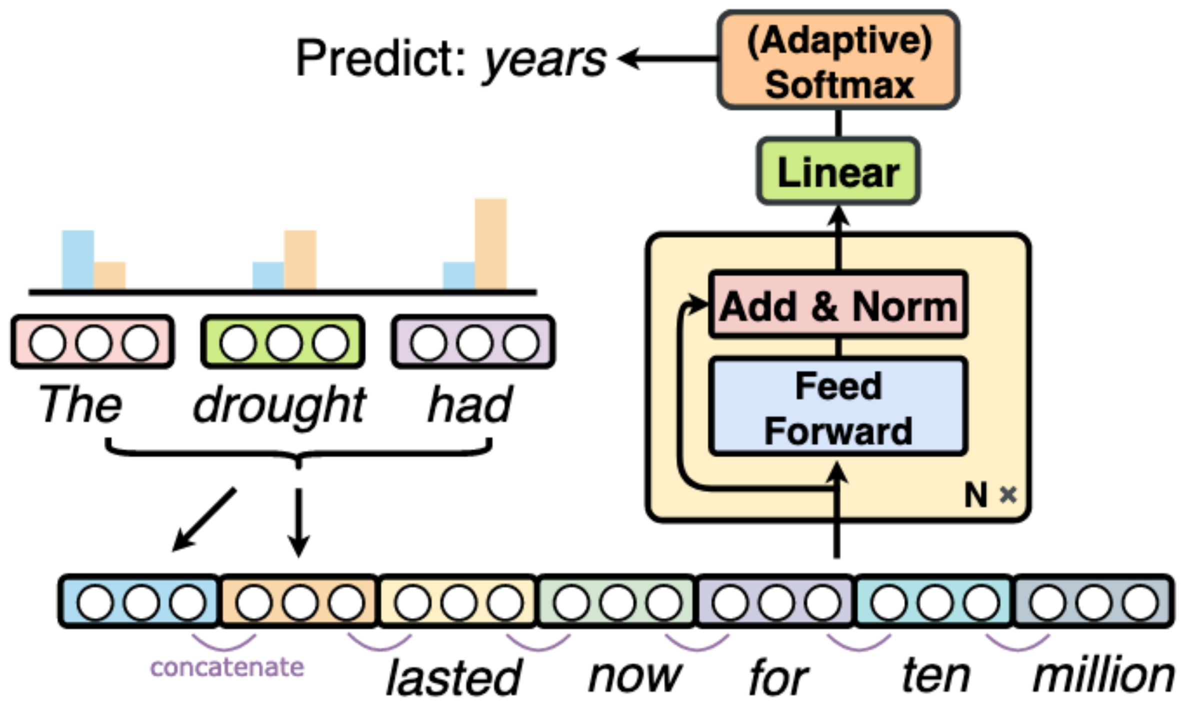
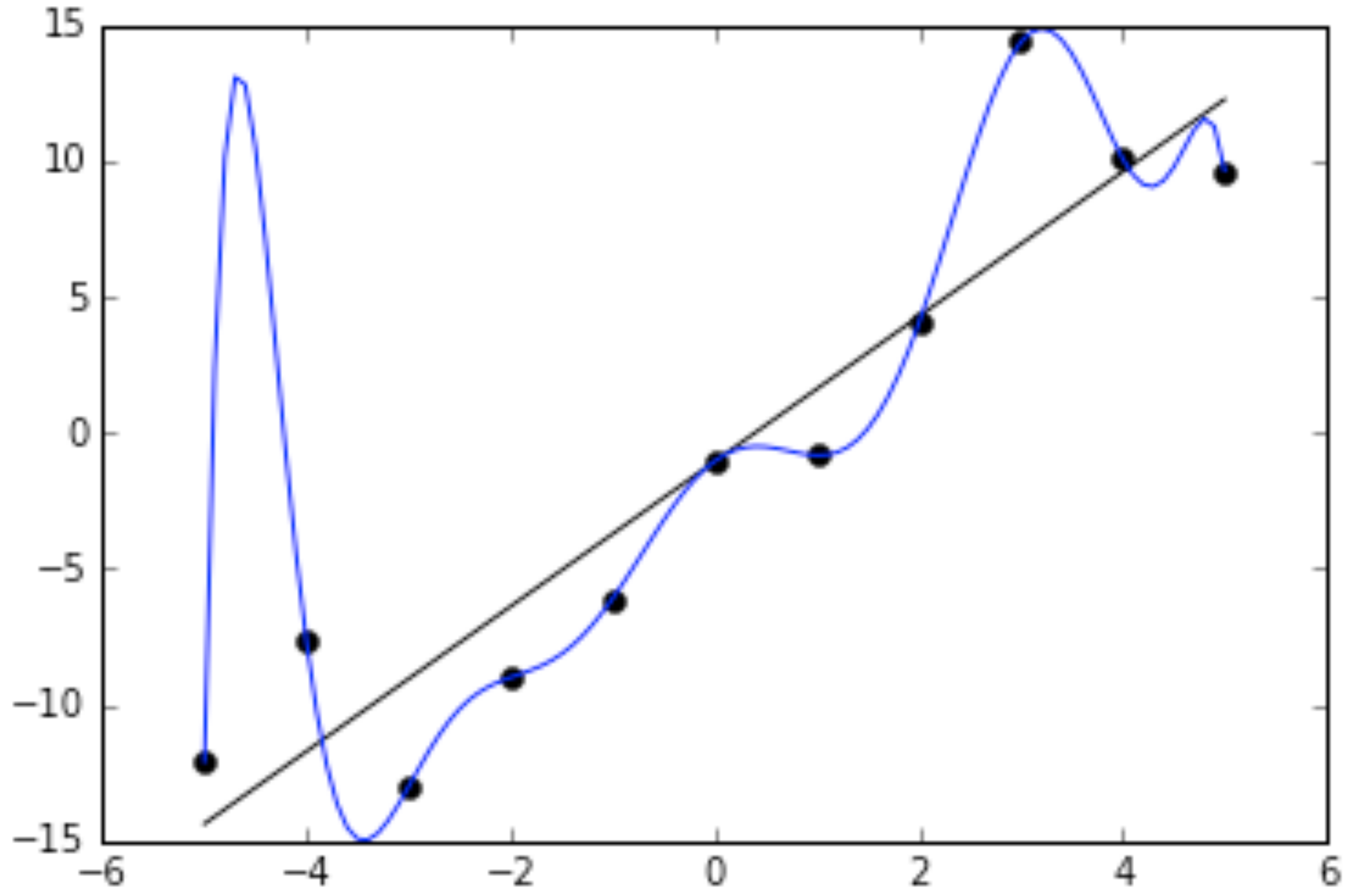


Figure 1: A modernized version of the neural probabilistic language model of Bengio et al. (2003), which

Additional Training Notes: Regularization and Hyper-Parameters

Overfitting

- Over-fitting: model too closely mimics the training data
  - Therefore, **cannot *generalize* well**
- Common when models are "**over-parameterized**"
  - E.g. fitting a high-degree polynomial
  - Neural models are typically over-parameterized
- Key questions:
  - How to detect overfitting?
  - How to prevent it?

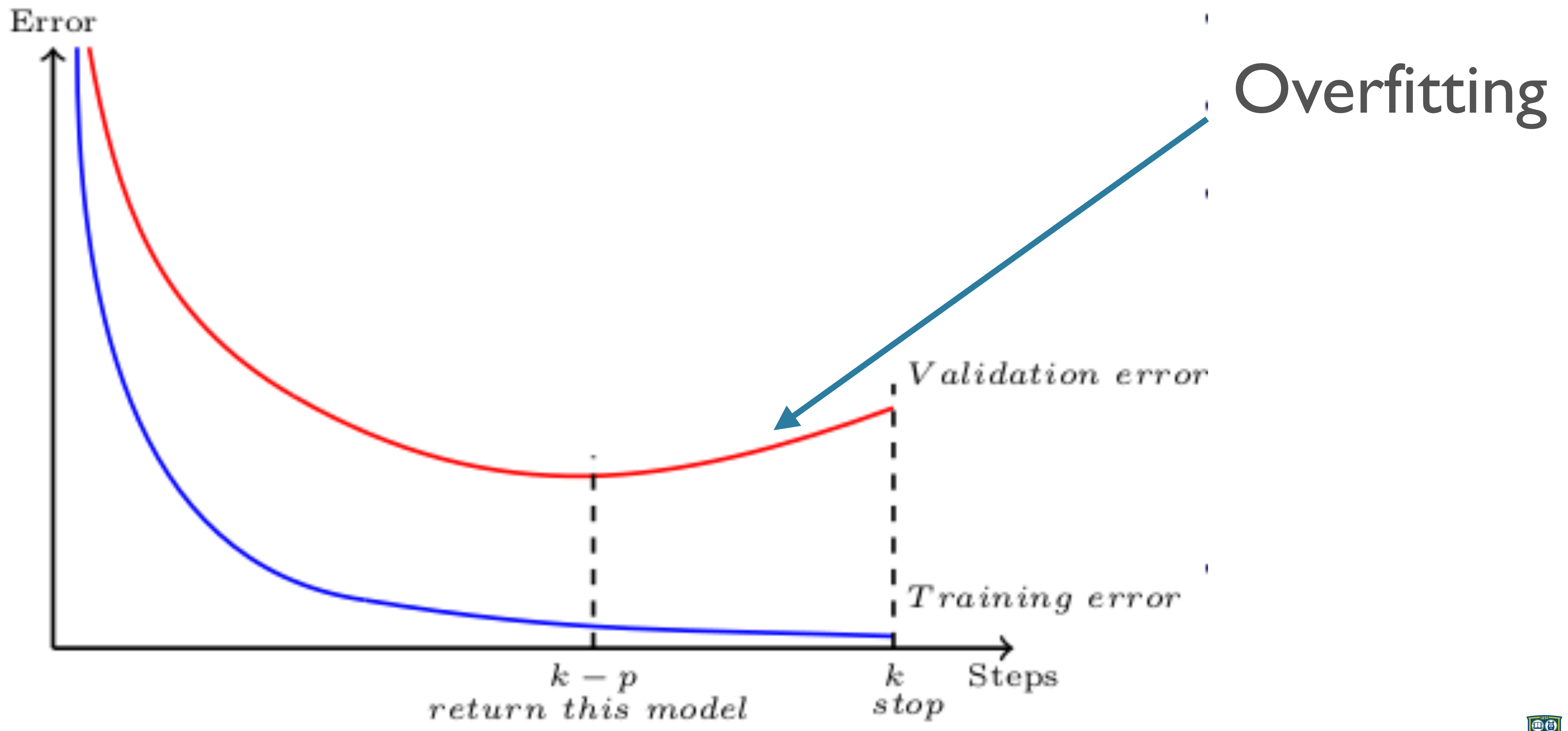


Train, Dev, Test Set Splits

- Split total data into three chunks: train, dev (aka valid), test
  - Common: 70/15/15, 80/10/10%
- **Train**: used for individual model training, as we've seen so far
- **Dev/valid**:
  - Evaluation during training
  - Hyper-parameter tuning
  - Model selection
- **Test**:
  - Final evaluation; **DO NOT TOUCH** otherwise

Early stopping

- Naive idea: 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: $\mathscr{L}'(\theta, y) = \mathscr{L}(\theta, y) + \lambda ||\theta||^2$
  - (penalty for **higher magnitude** parameters)
- Dropout:
  - During training, randomly **turn off X% of neurons** in each layer
  - (Don't do this during testing/predicting)
- Batch Normalization / Layer Norm
- Batch size choice can also be regulating

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

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

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

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

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

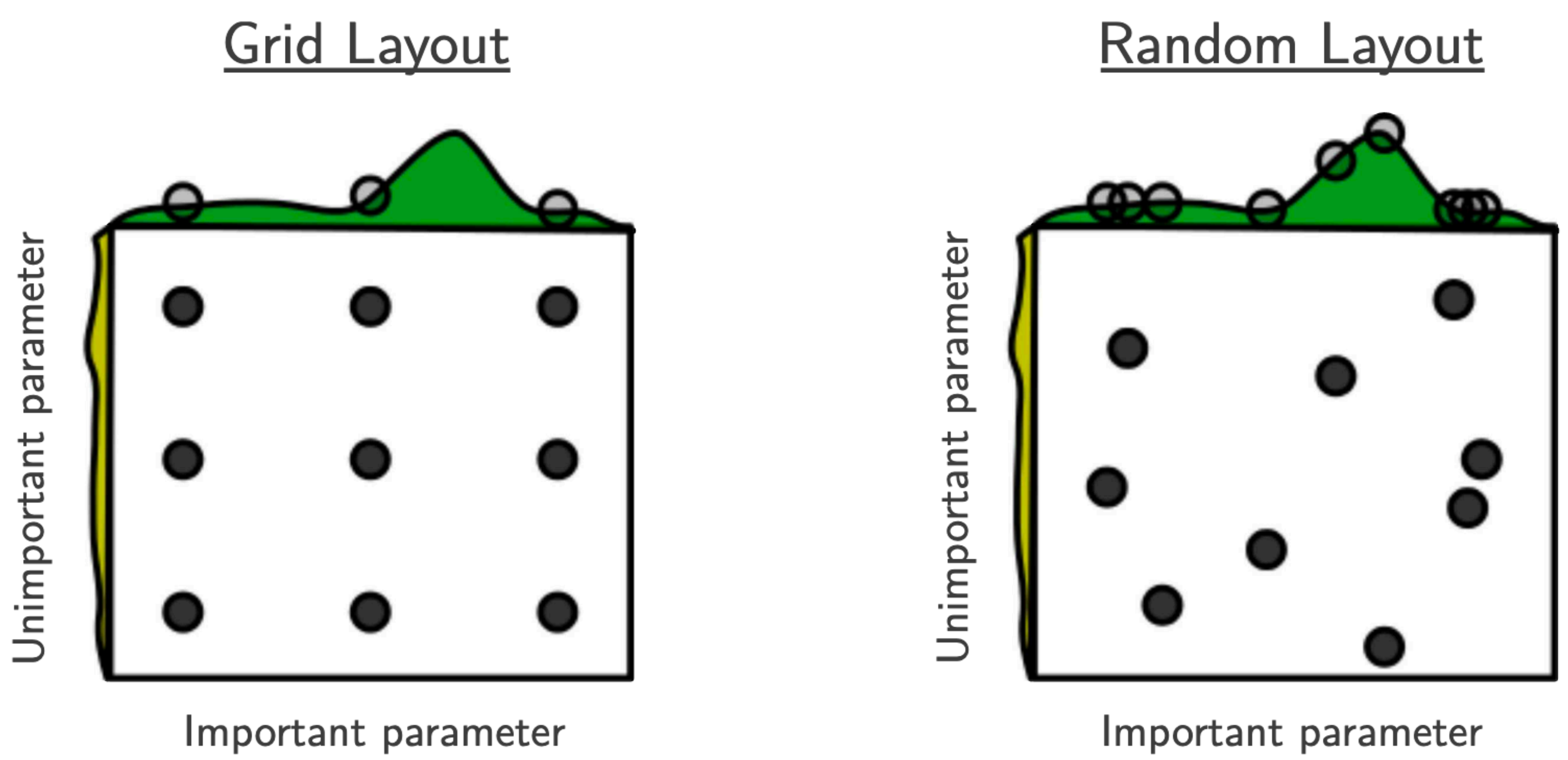
$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Hyper-parameters

- In addition to the model architecture ones mentioned earlier
- **Optimizer**: SGD, Adam, Adagrad, RMSProp, ….
  - Optimizer-specific hyper-parameters: learning rate, alpha, beta, …
  - (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. patience)

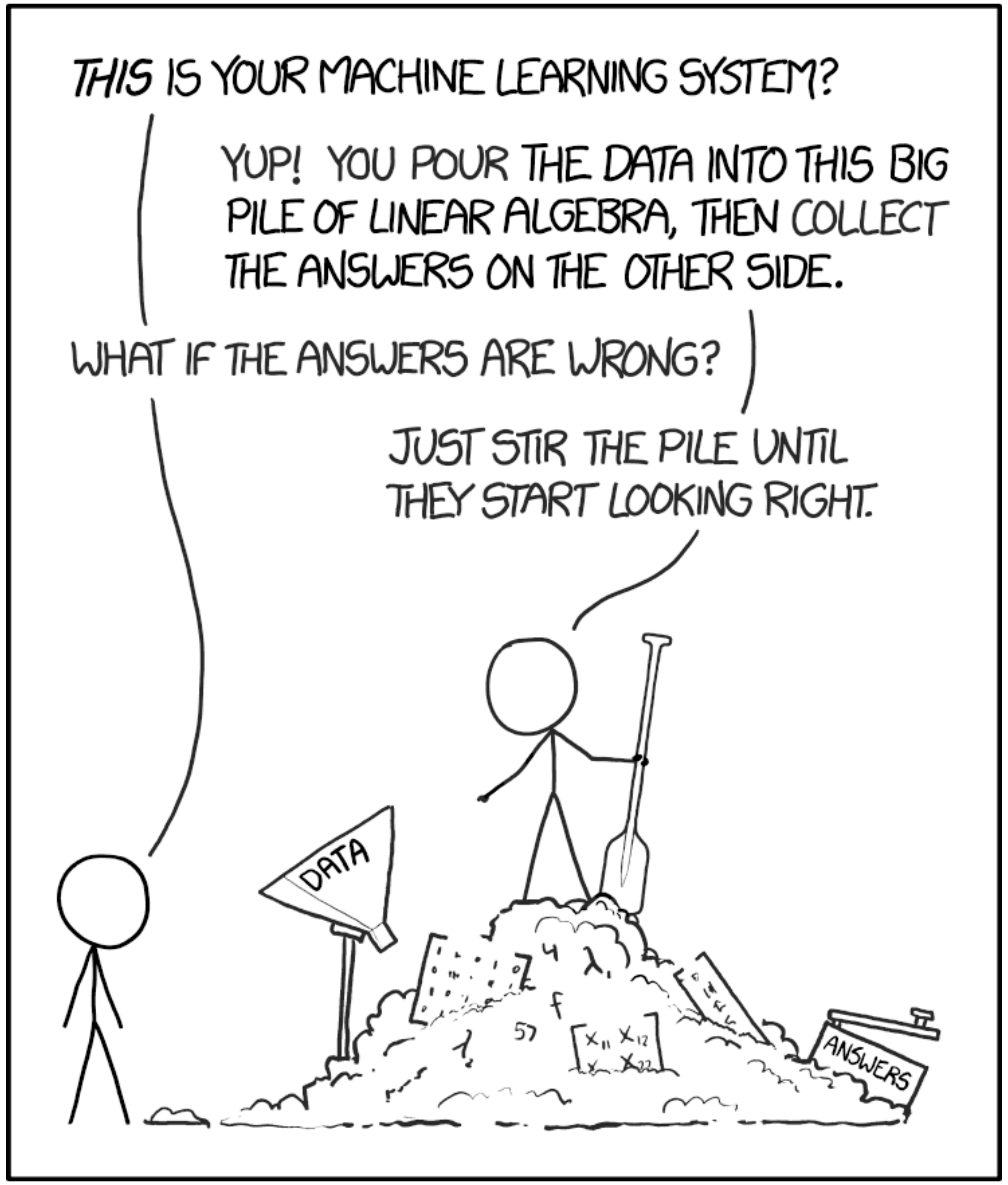
A note on hyper-parameter tuning

- **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

Craft/Art of Deep Learning



https://xkcd.com/1838/

Some Practical Pointers

- Hyper-parameter tuning and the like are not the focus of this course
- For some helpful hand-on advice about training NNs from scratch, debugging under “silent failures”, etc:
  - http://karpathy.github.io/2019/04/25/recipe/

Adagrad

- “Adaptive Gradients”
- Key idea: **adjust the learning rate per parameter**
- Frequent features —> more updates
- Adagrad will make the learning rate **smaller for those**

Adagrad

- Let $g_{t,i} := \nabla_{\theta_{t,i}} \mathscr{L}$
- SGD: $\theta_{t+1,i} = \theta_{t,i} - \alpha g_{t,i}$
- Adagrad: $\theta_{t+1,i} = \theta_{t,i} - \dfrac{\alpha}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$

$$G_{t,i} = \sum_{k=0}^{t} g_{k,i}^2$$

Accumulated change to parameter i over time

Adagrad

- Pros:
  - “Balances” parameter importance
  - **Less manual tuning** of learning rate needed (0.01 default)
- Cons:
  - $G_{t,i}$ increases monotonically, so **step-size always gets smaller**
- Newer optimizers try to have the pros without the cons
- Resources:
  - Original paper (veeery math-y): https://jmlr.org/papers/volume12/duchi11a/duchi11a.pdf
  - Overview of optimizers: https://ruder.io/optimizing-gradient-descent/index.html#adagrad