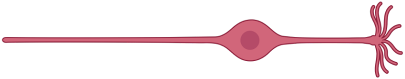
Neural Networks: Perceptrons

BIOE 498/598 PJ

Spring 2021

The artificial neuron

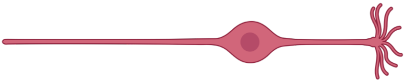
A neuron connects a series on inputs (dendrites) to an output (axon).

output: y  $\left.\begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{matrix}\right\}$ inputs

The artificial neuron

A neuron connects a series on inputs (dendrites) to an output (axon).

output: y

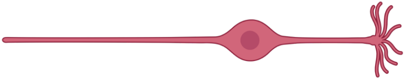


$$\left.\begin{matrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{matrix}\right\} \text{inputs}$$

Our model needs to include two processes:

1. The cell body (soma) combines all of the n inputs.
2. If the combined input exceeds a threshold, the output fires.

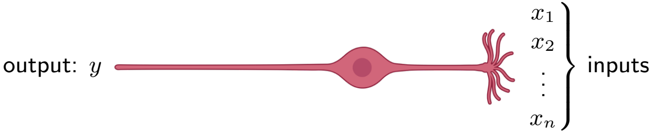
Modeling the artificial neuron

output: y  $x_1, x_2, \vdots, x_n$ $\Big\}$ inputs

Let's model the combined input z as a linear combination of the inputs.

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w} \cdot \mathbf{x}$$

Modeling the artificial neuron



Let's model the combined input z as a linear combination of the inputs.

$$z = w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = \mathbf{w} \cdot \mathbf{x}$$

For now, let's assume the neuron "fires" based on the sign of z:

$$y = \begin{cases} +1, & \mathbf{w} \cdot \mathbf{x} > 0 \\ -1, & \mathbf{w} \cdot \mathbf{x} < 0 \end{cases}$$

Wait, what happened to the intercept?

Our perceptron fires using the rule

$$y = \begin{cases} +1, & \mathbf{w} \cdot \mathbf{x} > 0 \\ -1, & \mathbf{w} \cdot \mathbf{x} < 0 \end{cases}$$

Does this mean the perceptron hyperplane always passes through the origin?

Wait, what happened to the intercept?

Our perceptron fires using the rule

$$y = \begin{cases} +1, & \mathbf{w} \cdot \mathbf{x} > 0 \\ -1, & \mathbf{w} \cdot \mathbf{x} < 0 \end{cases}$$

Does this mean the perceptron hyperplane always passes through the origin?

No. We use a common ML trick to move the *bias* (intercept) into the weight vector and expand $\mathbf{x}$ with a dummy dimension containing 1.

$$\mathbf{w} \cdot \mathbf{x} = b \Leftrightarrow \begin{pmatrix} w_1 \\ w_2 \\ -b \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ 1 \end{pmatrix} = 0$$

Summary (so far)

- A perceptron is a simplistic model of a single neuron.
- A perceptron can learn to perform simple classification tasks using an update rule.

Summary (so far)

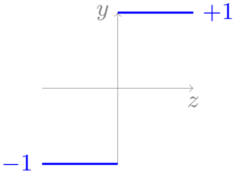
- A perceptron is a simplistic model of a single neuron.
- A perceptron can learn to perform simple classification tasks using an update rule.
- **Imagine what a network of millions of perceptrons can learn!**

Any nonlinearity will do

Any nonlinear function can be an activation function.

Sign/step activation

$$\mathrm{sgn}(z) = \begin{cases} +1, & z > 0 \\ -1, & z < 0 \end{cases}$$

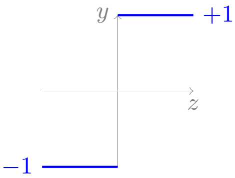


Any nonlinearity will do

Any nonlinear function can be an activation function.

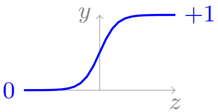
Sign/step activation

$$\operatorname{sgn}(z) = \begin{cases} +1, & z > 0 \\ -1, & z < 0 \end{cases}$$



Sigmoid activation

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

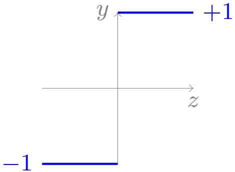


Any nonlinearity will do

Any nonlinear function can be an activation function.

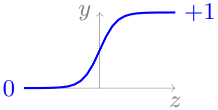
Sign/step activation

$$\mathrm{sgn}(z) = \begin{cases} +1, & z > 0 \\ -1, & z < 0 \end{cases}$$



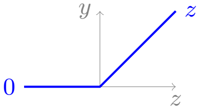
Sigmoid activation

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



Rectified linear unit activation

$$\mathrm{ReLU}(z) = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases}$$



Multi-neuron (wide) perceptrons

Neural networks use multiple neurons to learn different features from the **same inputs**.

$$
\begin{tikzcd}
y_1 = \sigma(\mathbf{w}_1 \cdot \mathbf{x}) & \\
y_2 = \sigma(\mathbf{w}_2 \cdot \mathbf{x}) & \\
y_3 = \sigma(\mathbf{w}_3 \cdot \mathbf{x}) & \mathbf{x} \arrow[l] \arrow[ul] \arrow[uul] \arrow[ddl] \\
\vdots & \\
y_m = \sigma(\mathbf{w}_m \cdot \mathbf{x}) &
\end{tikzcd}
$$

Multi-neuron (wide) perceptrons

Neural networks use multiple neurons to learn different features from the **same inputs**.

$$
\begin{tikzcd}
 & y_1 = \sigma(\mathbf{w}_1 \cdot \mathbf{x}) \arrow[ddl] & \\
 & y_2 = \sigma(\mathbf{w}_2 \cdot \mathbf{x}) \arrow[dl] & \\
y_{\text{out}} = \sigma(\mathbf{w}_{\text{out}} \cdot \mathbf{y}) & y_3 = \sigma(\mathbf{w}_3 \cdot \mathbf{x}) \arrow[l] & \mathbf{x} \arrow[uul] \arrow[ul] \arrow[l] \arrow[ddl] \\
 & \vdots & \\
 & y_m = \sigma(\mathbf{w}_m \cdot \mathbf{x}) \arrow[uul] &
\end{tikzcd}
$$

The outputs of each neuron are collected into a single neuron to predict the final class.

A matrix formalism for perceptrons

Consider a stack of m neurons that are all connected to the same input $\mathbf{x}$.

$$
\begin{aligned}
z_1 &= \mathbf{w}_1 \cdot \mathbf{x} \\
z_2 &= \mathbf{w}_2 \cdot \mathbf{x} \\
&\vdots \\
z_m &= \mathbf{w}_m \cdot \mathbf{x}
\end{aligned}
$$

A matrix formalism for perceptrons

Consider a stack of m neurons that are all connected to the same input $\mathbf{x}$.

$$
\begin{aligned}
z_1 &= \mathbf{w}_1 \cdot \mathbf{x} \\
z_2 &= \mathbf{w}_2 \cdot \mathbf{x} \\
&\vdots \\
z_m &= \mathbf{w}_m \cdot \mathbf{x}
\end{aligned}
$$

The stack can be written as the product of the input $\mathbf{x}$ and a weight matrix

$$\mathbf{z} = \mathbf{W}\mathbf{x}$$

where each row in $\mathbf{W}$ contins the weights for a single neuron

$$\mathbf{W} = \begin{pmatrix} \leftarrow \mathbf{w}_1 \rightarrow \\ \leftarrow \mathbf{w}_2 \rightarrow \\ \vdots \\ \leftarrow \mathbf{w}_m \rightarrow \end{pmatrix}.$$

Elementwise activation functions

Let's define an *elementwise* sigmoid activation function

$$\boldsymbol{\sigma}(\mathbf{z}) = \begin{pmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \vdots \\ \sigma(z_n) \end{pmatrix}.$$

Elementwise activation functions

Let's define an *elementwise* sigmoid activation function

$$\boldsymbol{\sigma}(\mathbf{z}) = \begin{pmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \vdots \\ \sigma(z_n) \end{pmatrix}.$$

A stack of m neurons can be written as

$$\begin{aligned} \mathbf{z} &= \mathbf{W}\mathbf{x} \\ \mathbf{y} &= \boldsymbol{\sigma}(\mathbf{z}) \end{aligned}$$

Elementwise activation functions

Let's define an *elementwise* sigmoid activation function

$$\boldsymbol{\sigma}(\mathbf{z}) = \begin{pmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \vdots \\ \sigma(z_n) \end{pmatrix}.$$

A stack of m neurons can be written as

$$\mathbf{z} = \mathbf{W}\mathbf{x}$$
$$\mathbf{y} = \boldsymbol{\sigma}(\mathbf{z})$$

or, more succinctly as

$$\mathbf{y} = \boldsymbol{\sigma}(\mathbf{W}\mathbf{x})$$

Elementwise activation functions

Let's define an *elementwise* sigmoid activation function

$$\boldsymbol{\sigma}(\mathbf{z}) = \begin{pmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \vdots \\ \sigma(z_n) \end{pmatrix}.$$

A stack of m neurons can be written as

$$\begin{aligned} \mathbf{z} &= \mathbf{W}\mathbf{x} \\ \mathbf{y} &= \boldsymbol{\sigma}(\mathbf{z}) \end{aligned}$$

or, more succinctly as

$$\mathbf{y} = \boldsymbol{\sigma}(\mathbf{W}\mathbf{x})$$

where

$$\dim(\mathbf{y}) = m \times 1, \quad \dim(\mathbf{z}) = m \times 1$$
$$\dim(\mathbf{W}) = m \times n, \quad \dim(\mathbf{x}) = n \times 1$$

Completing our matrix formalism

$$\begin{tikzcd}[column sep=small]
 & y_{1,1} = \sigma(z_{1,1}) \leftarrow z_{1,1} = \mathbf{w}_{1,1} \cdot \mathbf{x} \arrow[dl] & \\
 & y_{1,2} = \sigma(z_{1,2}) \leftarrow z_{1,2} = \mathbf{w}_{1,2} \cdot \mathbf{x} \arrow[dl] & \\
y_{2,1} = \sigma(z_{2,1}) \leftarrow z_{2,1} = \mathbf{w}_{2,1} \cdot \mathbf{y}_1 & y_{1,3} = \sigma(z_{1,3}) \leftarrow z_{1,3} = \mathbf{w}_{1,3} \cdot \mathbf{x} \arrow[l] & \mathbf{x} \arrow[ul] \arrow[uul] \arrow[l] \arrow[ddl] \\
 & \vdots & \\
 & y_{1,m} = \sigma(z_{1,m}) \leftarrow z_{1,m} = \mathbf{w}_{1,m} \cdot \mathbf{x} \arrow[uul] &
\end{tikzcd}$$

Completing our matrix formalism

$$
\begin{tikzcd}
 & y_{1,1} = \sigma(z_{1,1}) \leftarrow z_{1,1} = \mathbf{w}_{1,1} \cdot \mathbf{x} \arrow[ddl] & \\
 & y_{1,2} = \sigma(z_{1,2}) \leftarrow z_{1,2} = \mathbf{w}_{1,2} \cdot \mathbf{x} \arrow[dl] & \\
y_{2,1} = \sigma(z_{2,1}) \leftarrow z_{2,1} = \mathbf{w}_{2,1} \cdot \mathbf{y}_1 & y_{1,3} = \sigma(z_{1,3}) \leftarrow z_{1,3} = \mathbf{w}_{1,3} \cdot \mathbf{x} \arrow[l] & \mathbf{x} \arrow[uul] \arrow[ul] \arrow[l] \arrow[ddl] \\
 & \vdots & \\
 & y_{1,m} = \sigma(z_{1,m}) \leftarrow z_{1,m} = \mathbf{w}_{1,m} \cdot \mathbf{x} \arrow[uul] &
\end{tikzcd}
$$

$$\mathbf{y}_2 = \boldsymbol{\sigma}(\mathbf{z}_2) \quad \leftarrow \quad \mathbf{z}_2 = \mathbf{W}_2 \mathbf{y}_1 \quad \leftarrow \quad \mathbf{y}_1 = \boldsymbol{\sigma}(\mathbf{z}_1) \quad \leftarrow \quad \mathbf{z}_1 = \mathbf{W}_1 \mathbf{x}$$

Completing our matrix formalism

$$\begin{tikzcd}[column sep=small]
 & y_{1,1} = \sigma(z_{1,1}) \leftarrow z_{1,1} = \mathbf{w}_{1,1} \cdot \mathbf{x} \arrow[dl] & \\
 & y_{1,2} = \sigma(z_{1,2}) \leftarrow z_{1,2} = \mathbf{w}_{1,2} \cdot \mathbf{x} \arrow[l] & \\
{\color{blue} y_{2,1} = \sigma(z_{2,1}) \leftarrow z_{2,1} = \mathbf{w}_{2,1} \cdot \mathbf{y}_1} & y_{1,3} = \sigma(z_{1,3}) \leftarrow z_{1,3} = \mathbf{w}_{1,3} \cdot \mathbf{x} \arrow[l] & \mathbf{x} \arrow[l] \arrow[ul] \arrow[uul] \arrow[dl] \\
 & \vdots & \\
 & y_{1,m} = \sigma(z_{1,m}) \leftarrow z_{1,m} = \mathbf{w}_{1,m} \cdot \mathbf{x} \arrow[uul] &
\end{tikzcd}$$

$$\mathbf{y}_2 = \boldsymbol{\sigma}(\mathbf{z}_2) \quad \leftarrow \quad \mathbf{z}_2 = \mathbf{W}_2\mathbf{y}_1 \quad \leftarrow \quad \mathbf{y}_1 = \boldsymbol{\sigma}(\mathbf{z}_1) \quad \leftarrow \quad \mathbf{z}_1 = \mathbf{W}_1\mathbf{x}$$

$$\mathbf{y}_2 = \boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x})))$$

Completing our matrix formalism

$$y_{1,1} = \sigma(z_{1,1}) \leftarrow z_{1,1} = \mathbf{w}_{1,1} \cdot \mathbf{x}$$

$$y_{1,2} = \sigma(z_{1,2}) \leftarrow z_{1,2} = \mathbf{w}_{1,2} \cdot \mathbf{x}$$

$$y_{2,1} = \sigma(z_{2,1}) \leftarrow z_{2,1} = \mathbf{w}_{2,1} \cdot \mathbf{y}_1 \qquad y_{1,3} = \sigma(z_{1,3}) \leftarrow z_{1,3} = \mathbf{w}_{1,3} \cdot \mathbf{x} \longleftarrow \mathbf{x}$$

$$\vdots$$

$$y_{1,m} = \sigma(z_{1,m}) \leftarrow z_{1,m} = \mathbf{w}_{1,m} \cdot \mathbf{x}$$

$$\mathbf{y}_2 = \boldsymbol{\sigma}(\mathbf{z}_2) \quad \leftarrow \quad \mathbf{z}_2 = \mathbf{W}_2\mathbf{y}_1 \quad \leftarrow \quad \mathbf{y}_1 = \boldsymbol{\sigma}(\mathbf{z}_1) \quad \leftarrow \quad \mathbf{z}_1 = \mathbf{W}_1\mathbf{x}$$

$$\mathbf{y}_2 = \boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x})))$$

In general, a network with d layers is

$$\mathbf{y}_d = \boldsymbol{\sigma}(\mathbf{W}_d(\boldsymbol{\sigma}(\mathbf{W}_{d-1}(\cdots\boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x}))))))).$$

Deep learning with neural networks

$$\mathbf{y}_d = \boldsymbol{\sigma}(\mathbf{W}_d(\boldsymbol{\sigma}(\mathbf{W}_{d-1}(\cdots\boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x})))))))$$

- The number of neurons in each layer i is the *width* of the layer.
  - If the $(i-1)$th layer has n outputs and the ith layer has m outputs, the weight matrix $\mathbf{W}_i$ has dimensions $m \times n$.
  - The dimensions of the inputs $\mathbf{x}$ and outputs $\mathbf{y}_d$ are fixed by the problem.
  - Layer 1 is called the *input layer*, and layer d is the *output layer*.
  - We can use as many nodes as we want in the *hidden* layers.

Deep learning with neural networks

$$\mathbf{y}_d = \boldsymbol{\sigma}(\mathbf{W}_d(\boldsymbol{\sigma}(\mathbf{W}_{d-1}(\cdots\boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x})))))))$$

- The number of neurons in each layer i is the *width* of the layer.
  - If the $(i-1)$th layer has n outputs and the ith layer has m outputs, the weight matrix $\mathbf{W}_i$ has dimensions $m \times n$.
  - The dimensions of the inputs $\mathbf{x}$ and outputs $\mathbf{y}_d$ are fixed by the problem.
  - Layer 1 is called the *input layer*, and layer d is the *output layer*.
  - We can use as many nodes as we want in the *hidden* layers.
- The number of layers d is the *depth* of the neural network.

Deep learning with neural networks

$$\mathbf{y}_d = \boldsymbol{\sigma}(\mathbf{W}_d(\boldsymbol{\sigma}(\mathbf{W}_{d-1}(\cdots \boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x})))))))$$

- The number of neurons in each layer i is the *width* of the layer.
  - If the $(i-1)$th layer has n outputs and the ith layer has m outputs, the weight matrix $\mathbf{W}_i$ has dimensions $m \times n$.
  - The dimensions of the inputs $\mathbf{x}$ and outputs $\mathbf{y}_d$ are fixed by the problem.
  - Layer 1 is called the *input layer*, and layer d is the *output layer*.
  - We can use as many nodes as we want in the *hidden* layers.
- The number of layers d is the *depth* of the neural network.
- *Deep learning* means $d > 2$.

The importance of nonlinearity

The nonlinear functions (like $\boldsymbol{\sigma}$) sandwiched between the layers are critical to deep learning.

Let's imagine what would happen if we removed them:

$$\begin{aligned}
\mathbf{y}_d &= \boldsymbol{\sigma}(\mathbf{W}_d(\boldsymbol{\sigma}(\mathbf{W}_{d-1}(\cdots\boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x}))))))) \\
&= \mathbf{W}_d(\mathbf{W}_{d-1}(\cdots\mathbf{W}_2(\mathbf{W}_1\mathbf{x}))) \\
&= \mathbf{W}_d\mathbf{W}_{d-1}\cdots\mathbf{W}_2\mathbf{W}_1\mathbf{x} \\
&= \widetilde{\mathbf{W}}\mathbf{x}
\end{aligned}$$

The importance of nonlinearity

The nonlinear functions (like $\boldsymbol{\sigma}$) sandwiched between the layers are critical to deep learning.

Let's imagine what would happen if we removed them:

$$\begin{aligned}
\mathbf{y}_d &= \boldsymbol{\sigma}(\mathbf{W}_d(\boldsymbol{\sigma}(\mathbf{W}_{d-1}(\cdots \boldsymbol{\sigma}(\mathbf{W}_2(\boldsymbol{\sigma}(\mathbf{W}_1\mathbf{x}))))))) \\
&= \mathbf{W}_d(\mathbf{W}_{d-1}(\cdots \mathbf{W}_2(\mathbf{W}_1\mathbf{x}))) \\
&= \mathbf{W}_d\mathbf{W}_{d-1}\cdots\mathbf{W}_2\mathbf{W}_1\mathbf{x} \\
&= \widetilde{\mathbf{W}}\mathbf{x}
\end{aligned}$$

Without the activation functions, the entire neural network reduces to a single linear system!

Why do we want deep networks?

- The *Universal Approximation Theorem* states that given enough neurons, a 2-layer (input/output) perceptron can learn any reasonable function.
- Neural networks are therefore universal function approximators.

Why do we want deep networks?

- The *Universal Approximation Theorem* states that given enough neurons, a 2-layer (input/output) perceptron can learn any reasonable function.
- Neural networks are therefore universal function approximators.
- Unfortunately, the theorem does not tell us how many neurons we need to approximate a given function.
- For complicated functions, evidence suggests the number is enormous!

Why do we want deep networks?

- The *Universal Approximation Theorem* states that given enough neurons, a 2-layer (input/output) perceptron can learn any reasonable function.
- Neural networks are therefore universal function approximators.
- Unfortunately, the theorem does not tell us how many neurons we need to approximate a given function.
- For complicated functions, evidence suggests the number is enormous!
- Our brains are very deep, so it's reasonable to believe that deep networks learn more efficiently than wide ones.
- In practice this is almost certainly true.
- Deep learning reduces the total number of neurons needed to learn a function since each of the d layers needs fewer than $1/d$-times the number of neurons.

Why do deeper networks learn better?

We can understand deep networks using examples from *feature engineering*.

Imagine you wanted to learn a Michaelis-Menten function.

Why do deeper networks learn better?

We can understand deep networks using examples from *feature engineering*.

Imagine you wanted to learn a Michaelis-Menten function.

Single Layer

$$\begin{tikzcd} & V_{\mathsf{max}} \arrow[dl] \\ \dfrac{V_{\mathsf{max}}[S]}{K_m + [S]} & {[S]} \arrow[l] \\ & K_m \arrow[ul] \end{tikzcd}$$

Why do deeper networks learn better?

We can understand deep networks using examples from *feature engineering*.

Imagine you wanted to learn a Michaelis-Menten function.

Single Layer

$$\begin{tikzcd}
 & V_{\text{max}} \arrow[dl] \\
\frac{V_{\text{max}}[S]}{K_m + [S]} & {[S]} \arrow[l] \\
 & K_m \arrow[ul]
\end{tikzcd}$$

Two Layers

$$\begin{tikzcd}
 & & V_{\text{max}} \arrow[dl] \\
 & V_{\text{max}}[S] \arrow[dl] & \\
\frac{V_{\text{max}}[S]}{K_m + [S]} & & {[S]} \arrow[ul] \arrow[dl] \\
 & K_m + [S] \arrow[ul] & \\
 & & K_m \arrow[ul]
\end{tikzcd}$$

Why do deeper networks learn better?

We can understand deep networks using examples from *feature engineering*.

Imagine you wanted to learn a Michaelis-Menten function.

Single Layer

$$\begin{tikzcd}
 & V_{\mathsf{max}} \arrow[dl] \\
\dfrac{V_{\mathsf{max}}[S]}{K_m + [S]} & {[S]} \arrow[l] \\
 & K_m \arrow[ul]
\end{tikzcd}$$

Two Layers

$$\begin{tikzcd}
 & & V_{\mathsf{max}} \arrow[dl] \\
 & V_{\mathsf{max}}[S] \arrow[dl] & \\
\dfrac{V_{\mathsf{max}}[S]}{K_m + [S]} & & {[S]} \arrow[ul] \arrow[dl] \\
 & K_m + [S] \arrow[ul] & \\
 & & K_m \arrow[ul]
\end{tikzcd}$$

Each layer in the network only needs to improve the features for the next layer.

Summary

- Deep neural networks are built from layers in artificial neurons.
- Each neuron has the power of a linear classifier.
- Layers **must** be separated by nonlinear activation functions.
- Neural networks can learn nearly any function, but deep networks learn more efficiently.
- Each layer creates features for the subsequent layers to improve learning.

Summary

- Deep neural networks are built from layers in artificial neurons.
- Each neuron has the power of a linear classifier.
- Layers **must** be separated by nonlinear activation functions.
- Neural networks can learn nearly any function, but deep networks learn more efficiently.
- Each layer creates features for the subsequent layers to improve learning.
- **Next time:** Training a neural network to learn Q-factors.