# Neural Networks: Multi-layer Perceptrons

Spring 2021

# Review

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

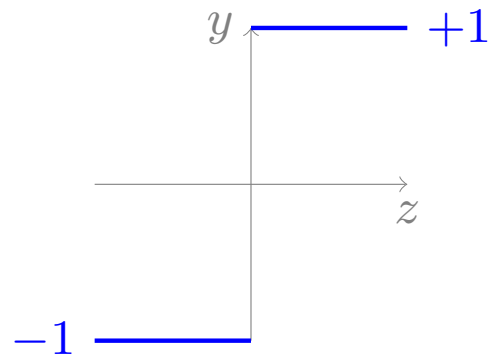
# A new activation function

Our simple perceptron computes an intermediate value $z$ using a weighted sum of the inputs

$$z = \mathbf{w} \cdot \mathbf{x}$$

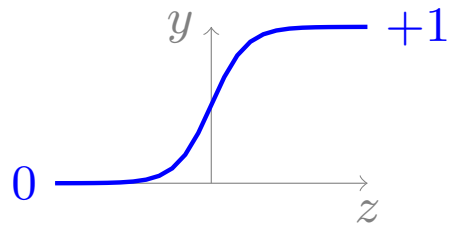
To simulate firing, we used the *sign function* for activation:

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




To avoid the discontinuity at zero, let's switch to the *sigmoid function*:

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



# Back to logistic regression?

A perceptron with a sigmoid activation function

$$y = \sigma(\mathbf{w} \cdot \mathbf{x})$$

is equivalent to a logistic regression model.

# Back to logistic regression?

A perceptron with a sigmoid activation function

$$y = \sigma(\mathbf{w} \cdot \mathbf{x})$$

is equivalent to a logistic regression model.

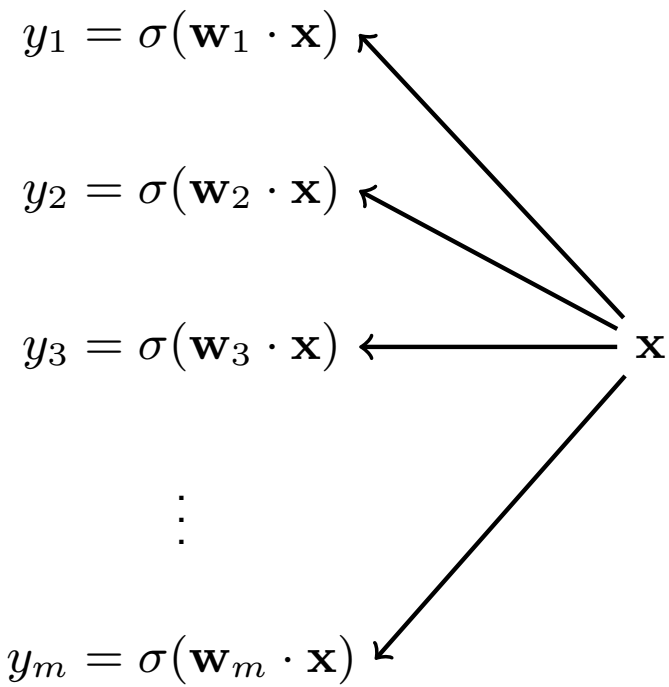
Adding the sigmoid function prevents us from using the Perceptron Update Algorithm. We must use a gradient descent-type method instead (just as we did for logistic regression).

Fortunately, the sigmoid function has a convenient derivative:

$$\frac{d}{dz}\sigma(z) = \sigma(z)[1 - \sigma(z)].$$

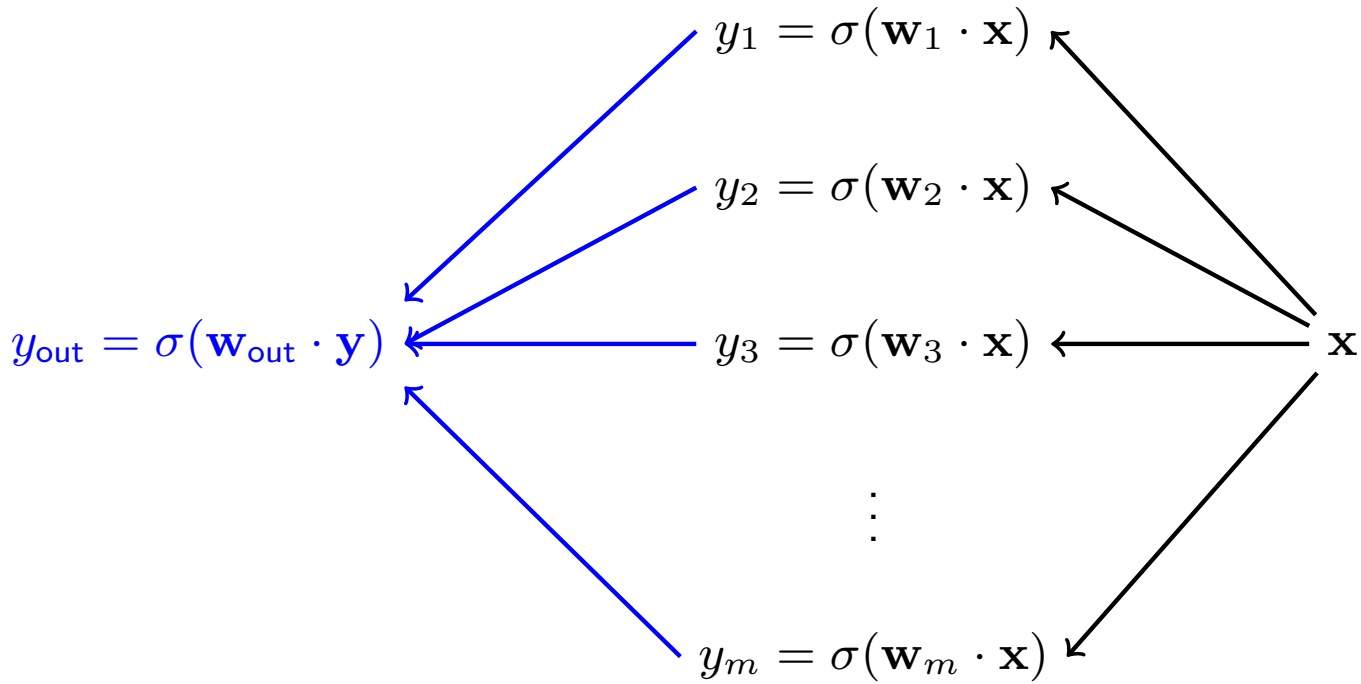
# Multi-neuron (wide) perceptrons

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



# Multi-neuron (wide) perceptrons

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




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 $\mathbf{x}$.

$$z_1 = \mathbf{w}_1 \cdot \mathbf{x}$$

$$z_2 = \mathbf{w}_2 \cdot \mathbf{x}$$

$$\vdots$$

$$z_m = \mathbf{w}_m \cdot \mathbf{x}$$

# A matrix formalism for perceptrons

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

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

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

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

# Three looks at linear systems

We've studied three classes of linear systems, differing only by what is **known** and **unknown**.

| | |
|---|---|
| $\mathbf{y} = \mathbf{A}\mathbf{x}$ | matrix multiplication |
| $\mathbf{y} = \mathbf{X}\beta$ | linear models |
| $\mathbf{z} = \mathbf{W}\mathbf{x}$ | neural networks |

# Three looks at linear systems

We've studied three classes of linear systems, differing only by what is **known** and **unknown**.

| | |
|---|---|
| $\mathbf{y} = \mathbf{A}\mathbf{x}$ | matrix multiplication |
| $\mathbf{y} = \mathbf{X}\beta$ | linear models |
| $\mathbf{z} = \mathbf{W}\mathbf{x}$ | neural networks |

Of these problems, neural networks are the most difficult to solve since $\mathbf{W}$ is a *matrix* of unknowns, not a *vector* like $\mathbf{y}$ and $\beta$.

# 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

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

# 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

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

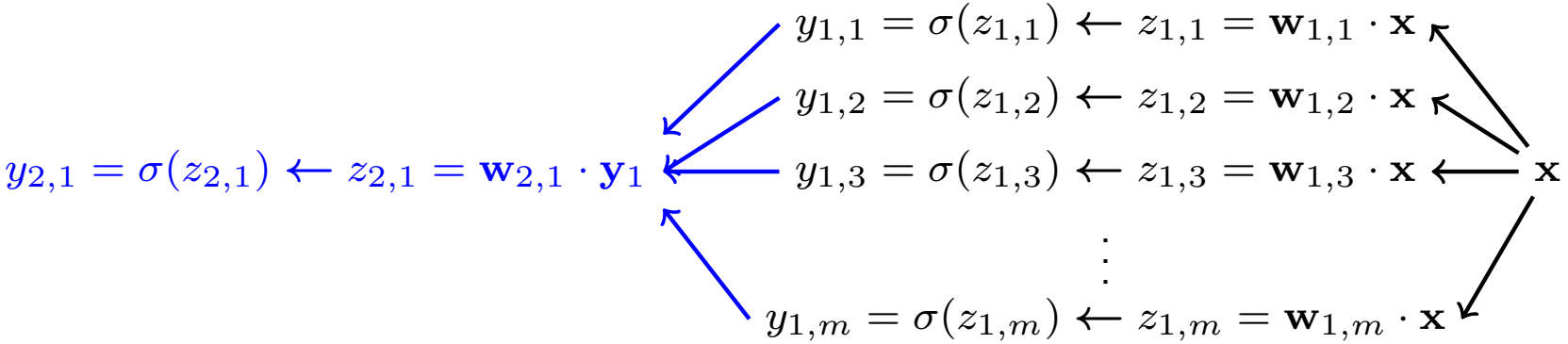
or, more succinctly as

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

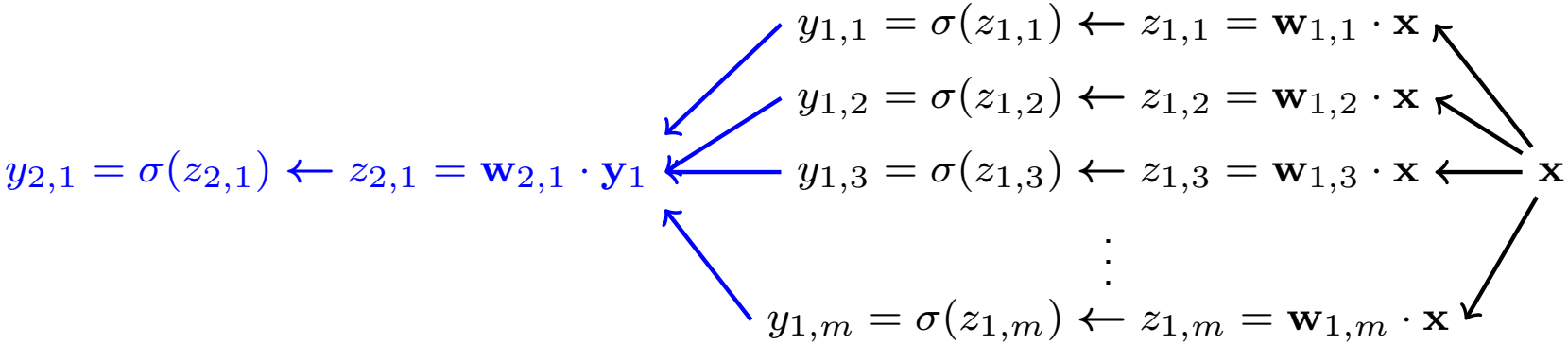
where

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

# Completing our matrix formalism

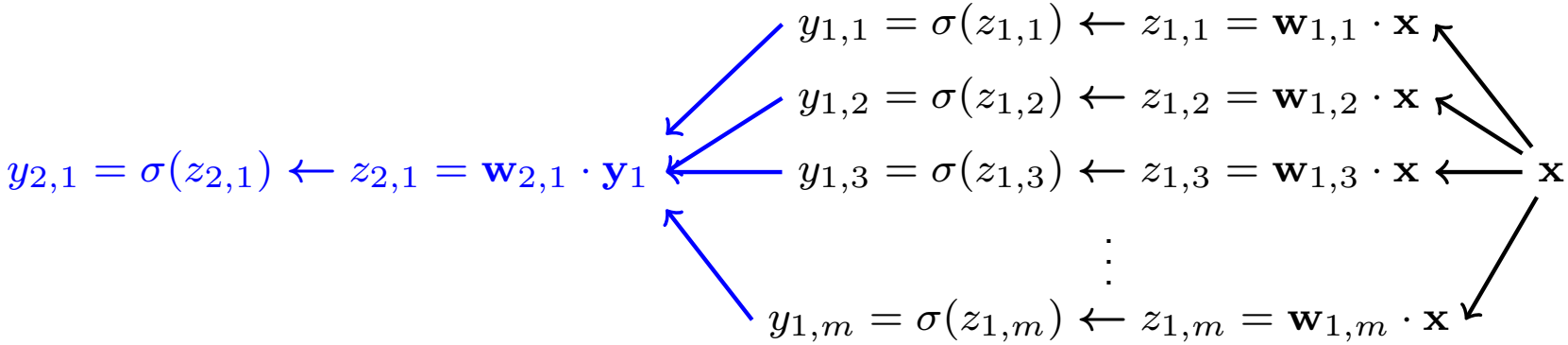


# Completing our matrix formalism




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

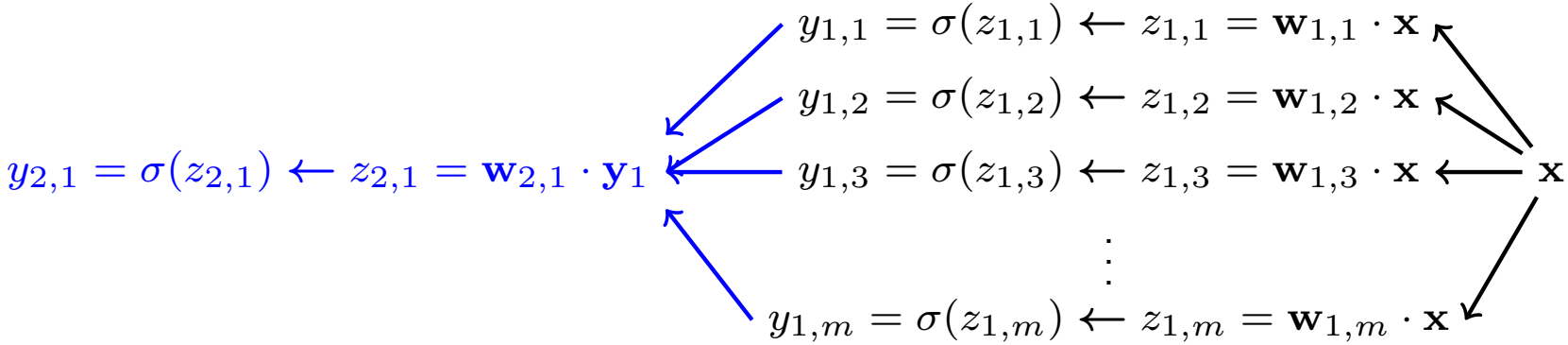
# Completing our matrix formalism




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

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

# Completing our matrix formalism




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

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

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 $i$th 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 $i$th 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 $i$th 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:

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

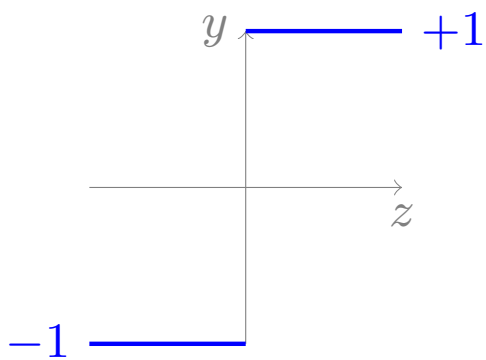
Without the activation functions, the entire neural network reduces to a single linear system!

# Any nonlinearity will do

Any nonlinear function can be an activation function.

**Sign/step activation**

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

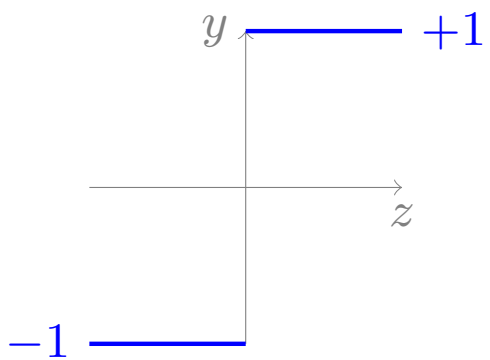


# Any nonlinearity will do

Any nonlinear function can be an activation function.

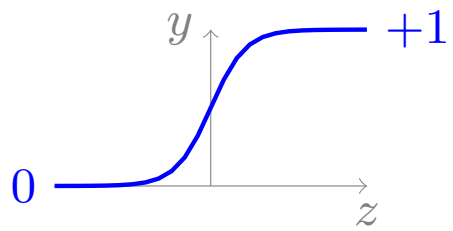
**Sign/step activation**

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




**Sigmoid activation**

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

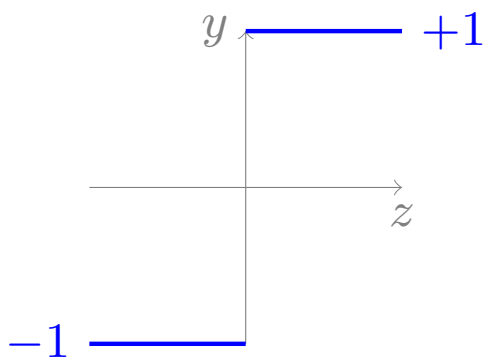


# Any nonlinearity will do

Any nonlinear function can be an activation function.

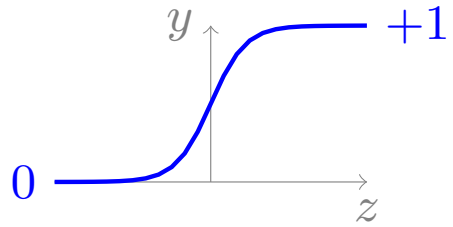
**Sign/step activation**

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



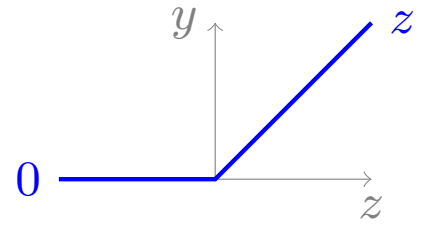

**Sigmoid activation**

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




**Rectified linear unit activation**

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



# 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?

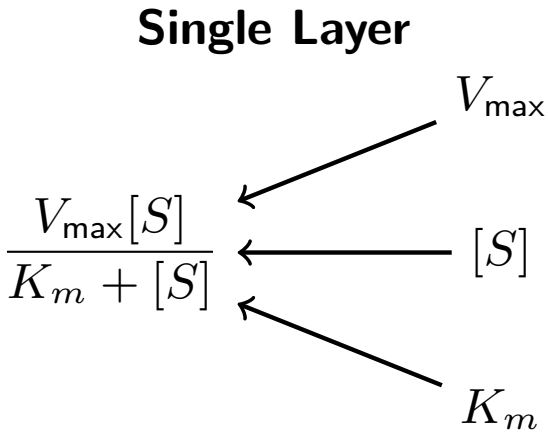
We can understand deep networks using examples from *feature engineering*.

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.

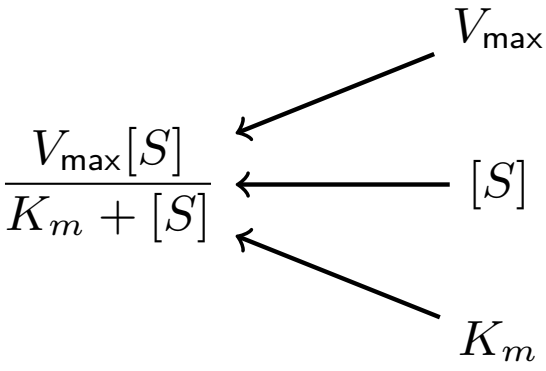


# Why do deeper networks learn better?

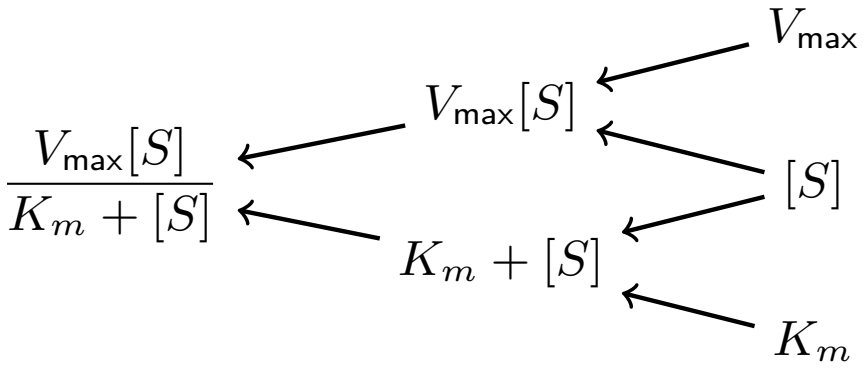
We can understand deep networks using examples from *feature engineering*.

Imagine you wanted to learn a Michaelis-Menten function.

**Single Layer**




**Two Layers**

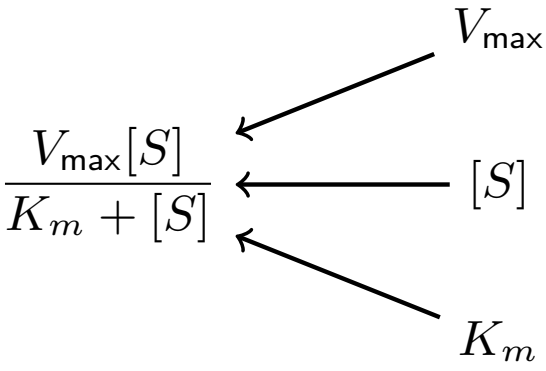


# Why do deeper networks learn better?

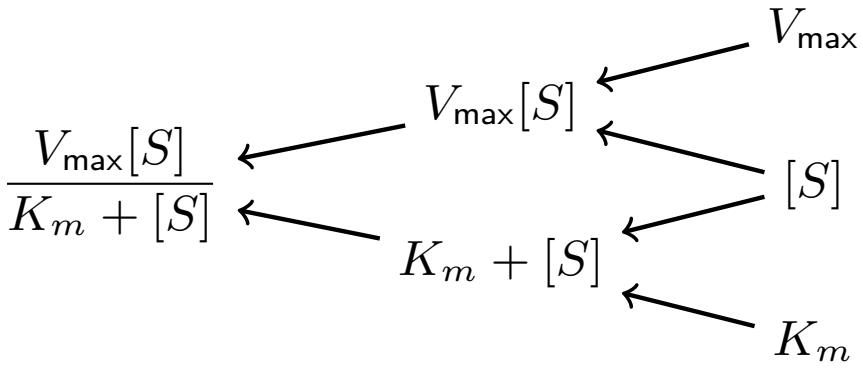
We can understand deep networks using examples from *feature engineering*.

Imagine you wanted to learn a Michaelis-Menten function.

**Single Layer**




**Two Layers**




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.

# Preparing for gradient descent

Neural networks are trained with a variation of gradient descent. Before calculating the gradients, we need to review the chain rule from calculus.

# Preparing for gradient descent

Neural networks are trained with a variation of gradient descent. Before calculating the gradients, we need to review the chain rule from calculus.

For any differentiable functions $f$ and $h$,

$$\frac{d}{dx}h(f(x)) = \frac{dh}{df}\frac{df}{dx}$$

# Preparing for gradient descent

Neural networks are trained with a variation of gradient descent. Before calculating the gradients, we need to review the chain rule from calculus.

For any differentiable functions $f$ and $h$,

$$\frac{d}{dx}h(f(x)) = \frac{dh}{df}\frac{df}{dx}$$

Similarly, for the $k$ functions $f_1$, $f_2$, ..., $f_k$,

$$\frac{d}{dx}f_k(f_{k-1}(\cdots f_2(f_1(x)))) = \frac{df_k}{df_{k-1}}\frac{df_{k-1}}{df_{k-2}}\cdots\frac{df_i}{df_{i-1}}\cdots\frac{df_2}{df_1}\frac{df_1}{dx}$$

# Preparing for gradient descent

Neural networks are trained with a variation of gradient descent. Before calculating the gradients, we need to review the chain rule from calculus.

For any differentiable functions $f$ and $h$,

$$\frac{d}{dx}h(f(x)) = \frac{dh}{df}\frac{df}{dx}$$

Similarly, for the $k$ functions $f_1$, $f_2$, ..., $f_k$,

$$\frac{d}{dx}f_k(f_{k-1}(\cdots f_2(f_1(x)))) = \frac{df_k}{df_{k-1}}\frac{df_{k-1}}{df_{k-2}}\cdots\frac{df_i}{df_{i-1}}\cdots\frac{df_2}{df_1}\frac{df_1}{dx}$$

For example, let $f(x) = (2x-3)^3$. Let $f_1(x) = 2x-3$ and $f_2(z) = z^3$. Then $f(x) = f_2(f_1(x))$ and

$$\begin{aligned}\frac{df}{dx} &= \frac{df_2}{df_1}\frac{df_1}{dx} \\ &= 3(2x-3)^2(2)\end{aligned}$$