

# ROB 430/599: Deep Learning for Robot Perception and Manipulation (DeepRob)

---

Lecture 5: Neural Networks (cont'd)

01/27/2026



# Aha Slides (In-class participation)

<https://ahaslides.com/FSKOC>



# Today

---

- Neural Networks
  - Review: Image Features (5min)
  - Neural Networks, Activation Functions (10min)
  - Space Warping (10min)
  - Universal Approximation (10min)
  - Convex Function (10min)
- Summary and Takeaways (5min)

# Neural Networks

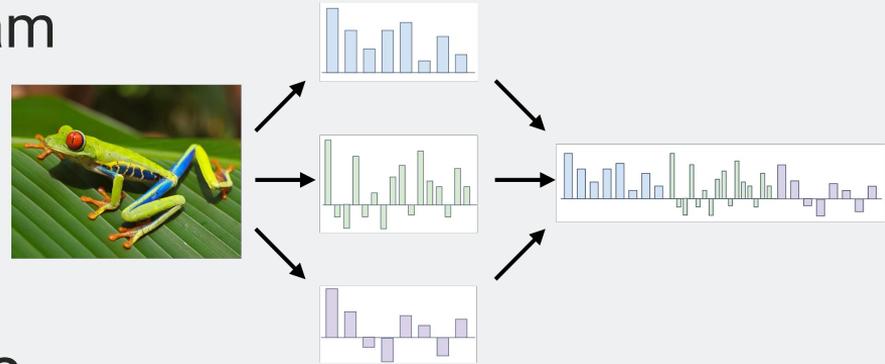
# Recap:

---

Problem: Linear Classifiers aren't that powerful

Solution 1: Image Features

- Feature transformation; histogram
- HoG
- BoW



Solution 2: Non-linear classifiers  
(e.g., Neural Network)

# Neural Networks (Overview)

**Input:**  $x \in \mathbb{R}^D$

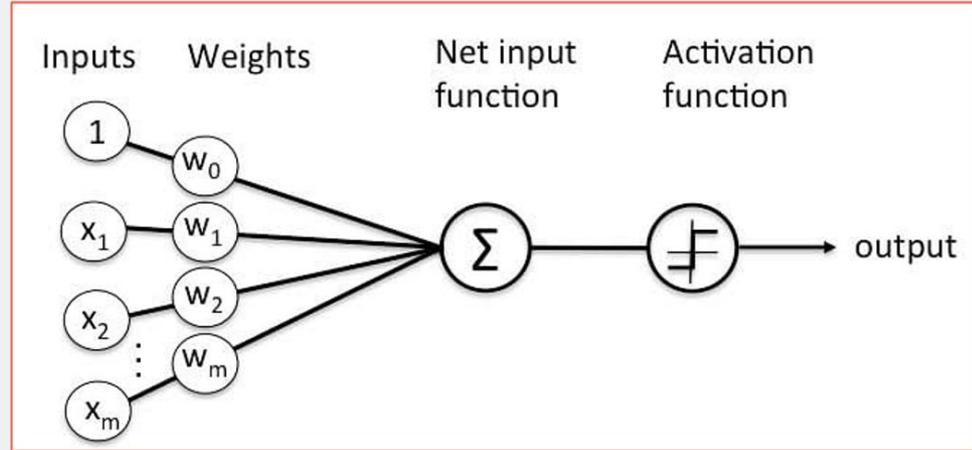
**Output:**  $f(x) \in \mathbb{R}^C$

## Rosenblatt's Perceptron

- A set of *synapses* each of which is characterized by a *weight* (which includes a *bias*).

- An *adder*

- An *activation function* (e.g., Rectified Linear Unit/ReLU, Sigmoid function, etc.)



$$y_k = \phi \left( \sum_{j=1}^m w_{kj} x_j + b_k \right)$$

# Recap: Neural Networks

---

**Input:**  $x \in \mathbb{R}^D$

**Output:**  $f(x) \in \mathbb{R}^C$

**Before:** Linear Classifier:  $f(x) = Wx + b$   
Learnable parameters:  $W \in \mathbb{R}^{D \times C}, b \in \mathbb{R}^C$

Feature Extraction

Linear Classifier

→ **Now:** Two-Layer Neural Network:  $f(x) = W_2 \max(0, W_1 x + b_1) + b_2$   
Learnable parameters:  $W_1 \in \mathbb{R}^{H \times D}, b_1 \in \mathbb{R}^H, W_2 \in \mathbb{R}^{C \times H}, b_2 \in \mathbb{R}^C$

Or Three-Layer Neural Network:

$$f(x) = W_3 \max(0, W_2 \max(0, W_1 x + b_1) + b_2) + b_3$$

# Neural Networks

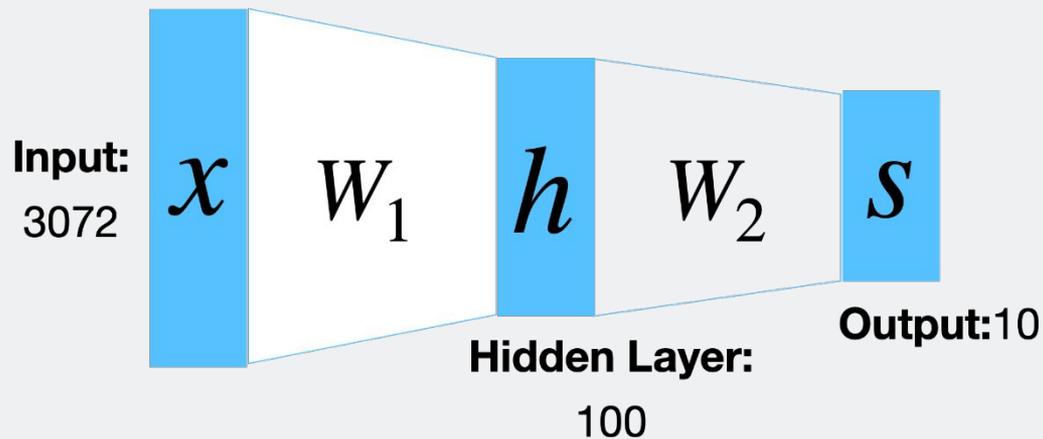
---

**Before:** Linear Classifier:

$$f(x) = Wx + b$$

➔ **Now:** Two-Layer Neural Network:

$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

# Neural Networks

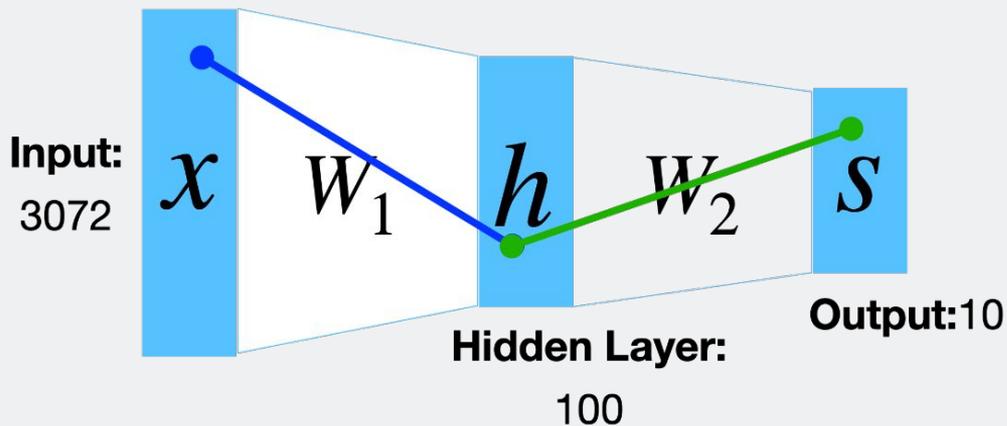
**Before:** Linear Classifier:

$$f(x) = Wx + b$$

**Now:** Two-Layer Neural Network:

$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$

Element  $(i, j)$  of  $W_1$   
gives the effect on  
 $h_i$  from  $x_j$



Element  $(i, j)$  of  $W_2$   
gives the effect on  
 $s_i$  from  $h_j$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

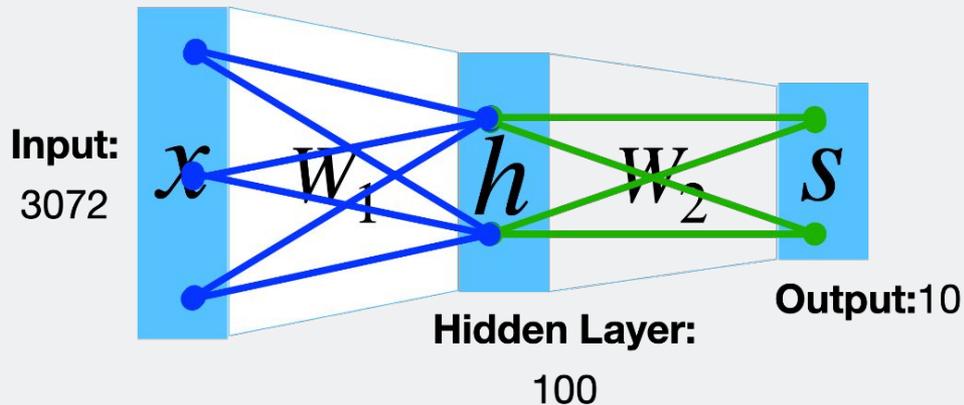
# Neural Networks

**Before:** Linear Classifier:

$$f(x) = Wx + b$$

**Now:** Two-Layer Neural Network:

$$f(x) = W_2 \max(0, W_1x + b_1) + b_2$$



Element  $(i, j)$  of  $W_1$  gives the effect on  $h_i$  from  $x_j$

All elements of  $x$  affect all elements of  $h$

Element  $(i, j)$  of  $W_2$  gives the effect on  $s_i$  from  $h_j$

All elements of  $h$  affect all elements of  $s$

Fully-connected neural network also  
"Multi-Layer Perceptron" (MLP)

# Neural Networks

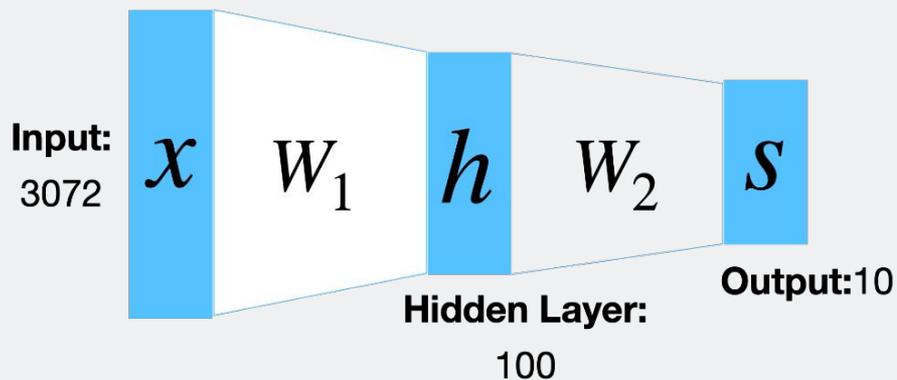
Recall:

Linear classifier: One template per class



**Before:** Linear score function

**Now:** Two-Layer Neural Network:



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

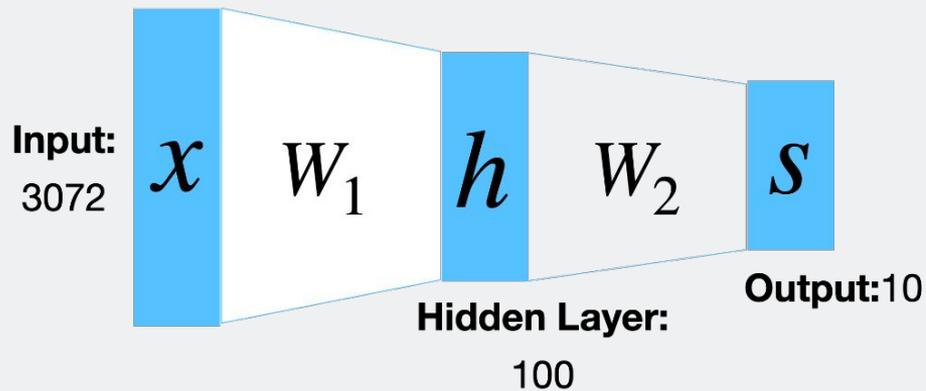
# Neural Networks

Neural net: first layer is bank of templates;  
Second layer recombines templates



**Before:** Linear score function

**Now:** Two-Layer Neural Network:



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

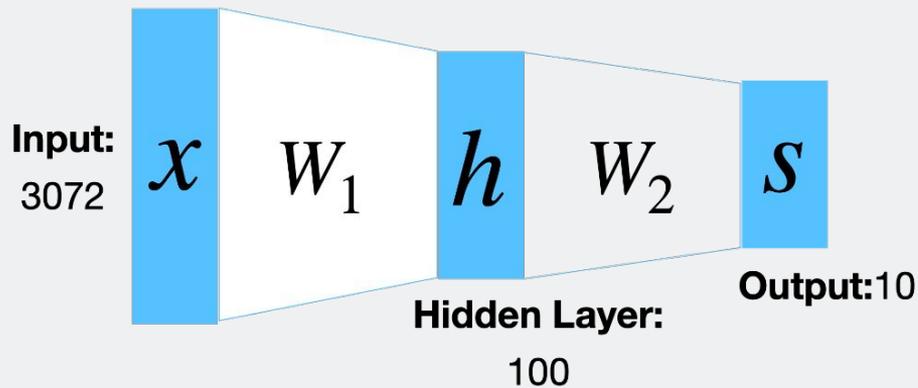
# Neural Networks

Can use different templates to cover multiple modes of a class!



**Before:** Linear score function

**Now:** Two-Layer Neural Network:



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

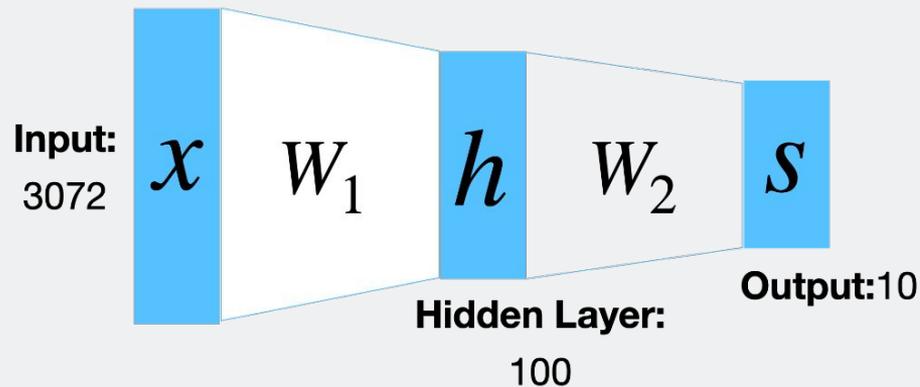
# Neural Networks

Can use different templates to cover multiple modes of a class!



**Before:** Linear score function

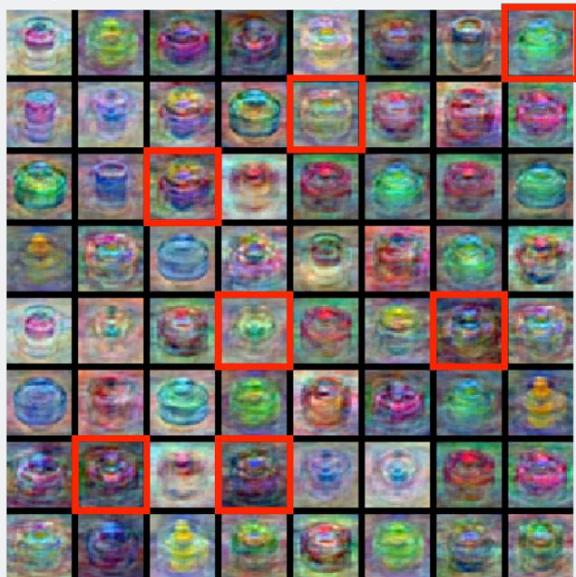
**Now:** Two-Layer Neural Network:



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

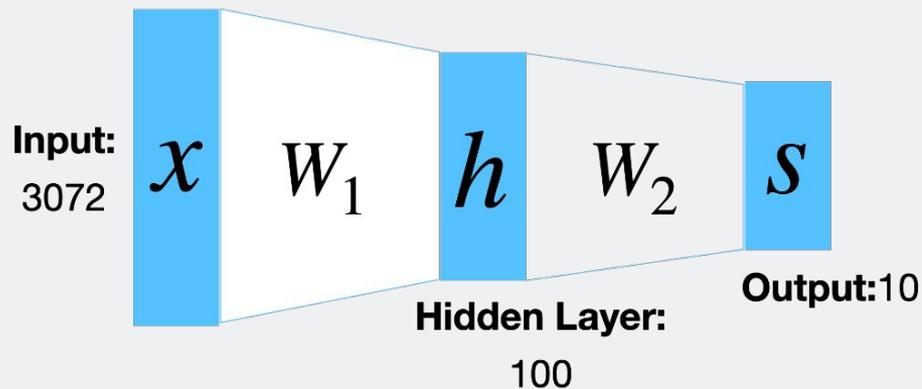
# Neural Networks

“Distributed representation”: Most templates not interpretable!



**Before:** Linear score function

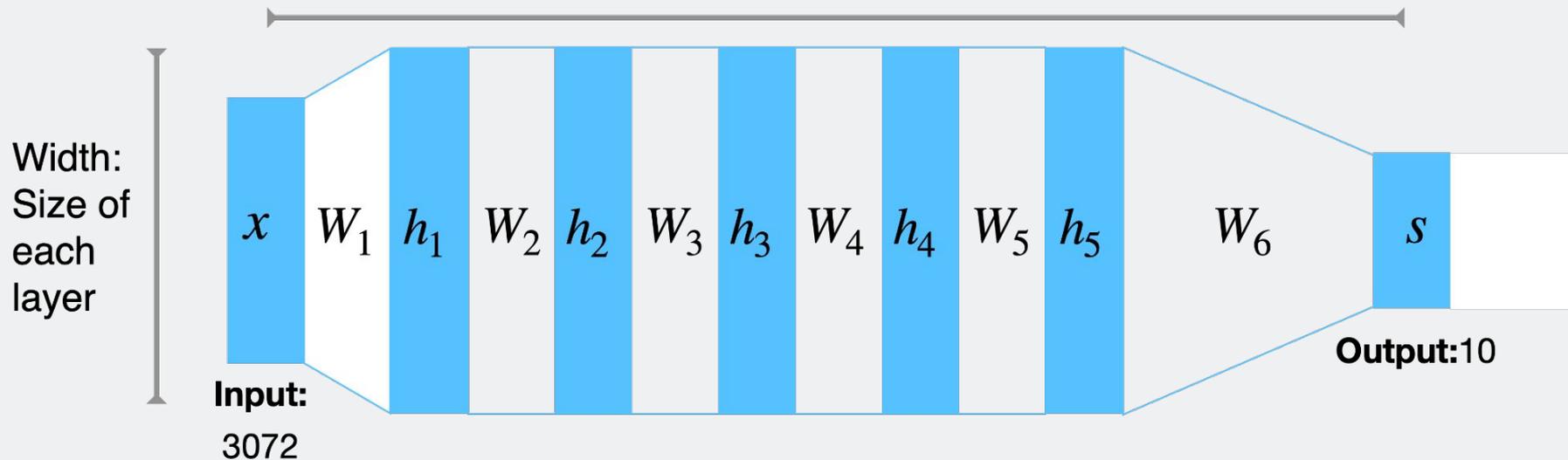
**Now:** Two-Layer Neural Network:



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

# Deep Neural Networks

Depth = number of layers



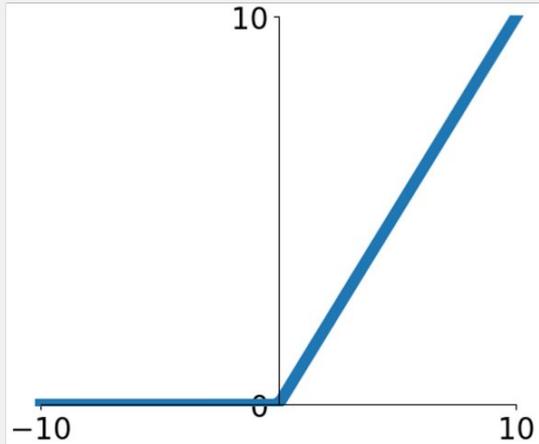
$$s = W_6 \max(0, W_5 \max(0, W_4 \max(0, W_3 \max(0, W_2 \max(0, W_1 x))))))$$

# Neural Networks: Activation Functions

---

## 2-Layer Neural Network

The activation  $ReLU(z) = \max(0, z)$  is called “Rectified Linear Unit”



$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$

This is called the **activation function** of the neural network

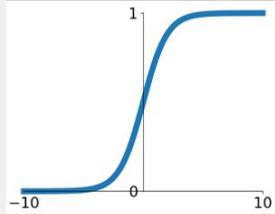
Q: What happens if we build a neural network with no activation function?

Linear

# Activation Functions

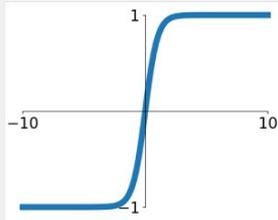
## Sigmoid

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



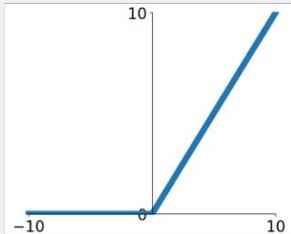
## tanh

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



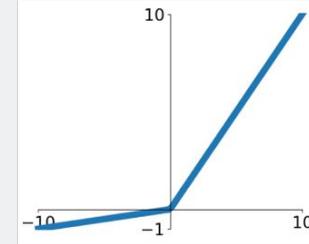
## ReLU

$$\max(0, x)$$



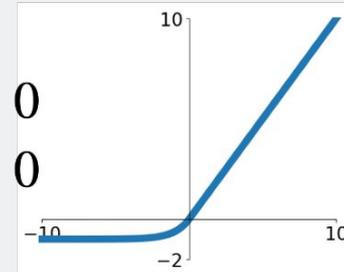
## Leaky ReLU

$$\max(0.2x, x)$$



## ELU

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \leq 0 \end{cases}$$



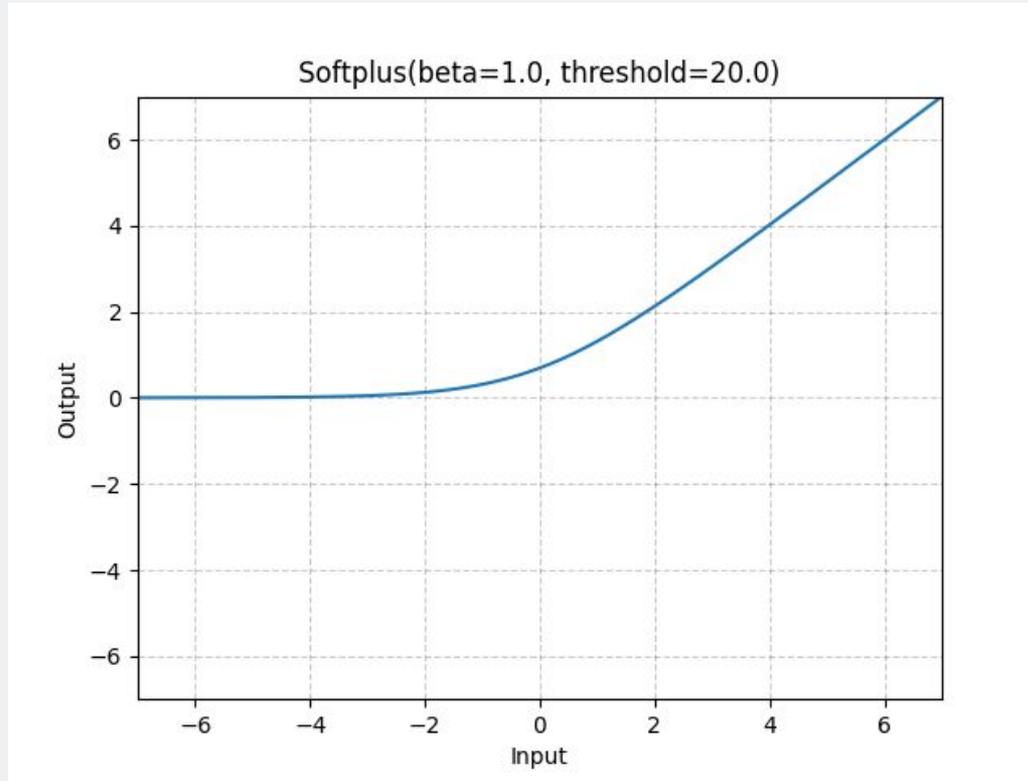
# Activation Functions

---

## Softplus

$$\log(1 + \exp(x))$$

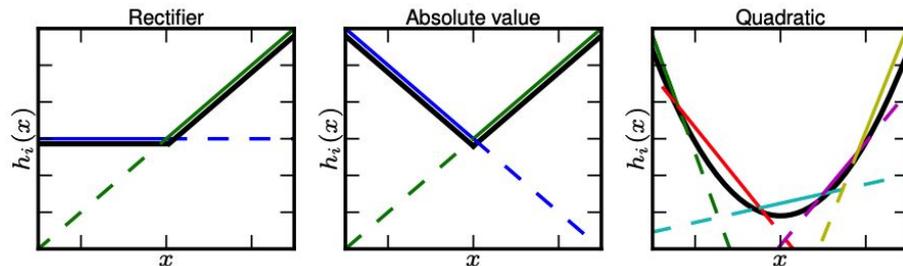
<https://pytorch.org/docs/stable/generated/torch.nn.Softplus.html>



# Activation Functions

## Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$



*Figure 1.* Graphical depiction of how the maxout activation function can implement the rectified linear, absolute value rectifier, and approximate the quadratic activation function. This diagram is 2D and only shows how maxout behaves with a 1D input, but in multiple dimensions a maxout unit can approximate arbitrary convex functions.

<https://proceedings.mlr.press/v28/goodfellow13.pdf>

# Maxout example

---

## Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Given input vector  $x = [1 \ 2 \ 3 \ 4]^T$

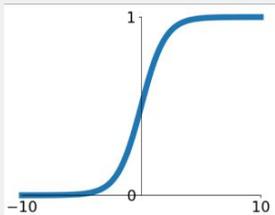
$$w_1 = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{bmatrix}, b_1 = [-1 \ -1 \ ]^T \text{ and } w_2 = \begin{bmatrix} 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix}, b_2 = [1 \ 1]^T$$

$$\text{MaxOut}(x) = ???$$

# Activation Functions

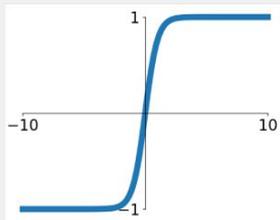
## Sigmoid

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



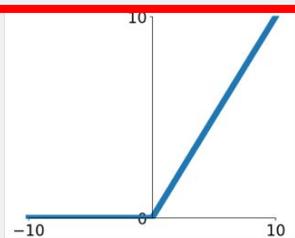
## tanh

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



## ReLU

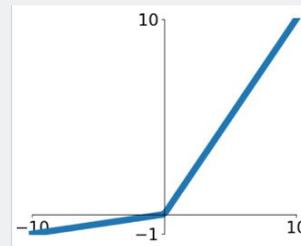
$$\max(0, x)$$



ReLU is a good default choice for most problems

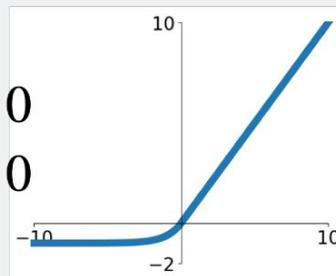
## Leaky ReLU

$$\max(0.2x, x)$$

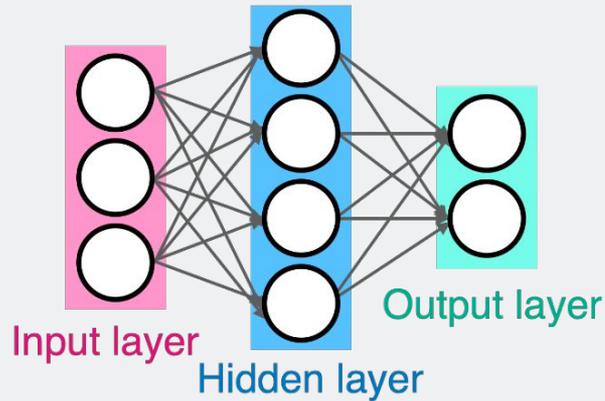


## ELU

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \leq 0 \end{cases}$$



# Neural Network in 20 Lines



Initialize weights and data

Compute loss (Sigmoid activation, L2 loss)

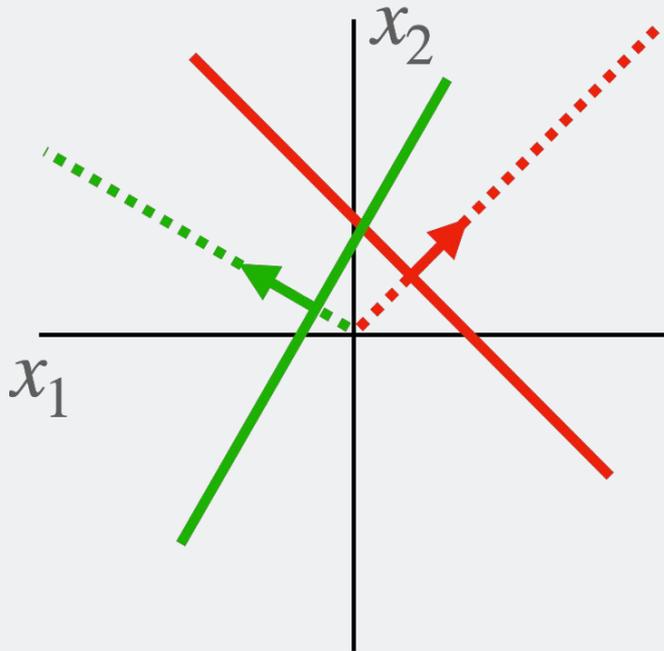
Compute gradients

SGD step

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, Din, H, Dout = 64, 1000, 100, 10
5 x, y = randn(N, Din), randn(N, Dout)
6 w1, w2 = randn(Din, H), randn(H, Dout)
7 for t in range(10000):
8     h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
9     y_pred = h.dot(w2)
10    loss = np.square(y_pred - y).sum()
11    dy_pred = 2.0 * (y_pred - y)
12    dw2 = h.T.dot(dy_pred)
13    dh = dy_pred.dot(w2.T)
14    dw1 = x.T.dot(dh * h * (1 - h))
15    w1 -= 1e-4 * dw1
16    w2 -= 1e-4 * dw2
```

# Space Warping

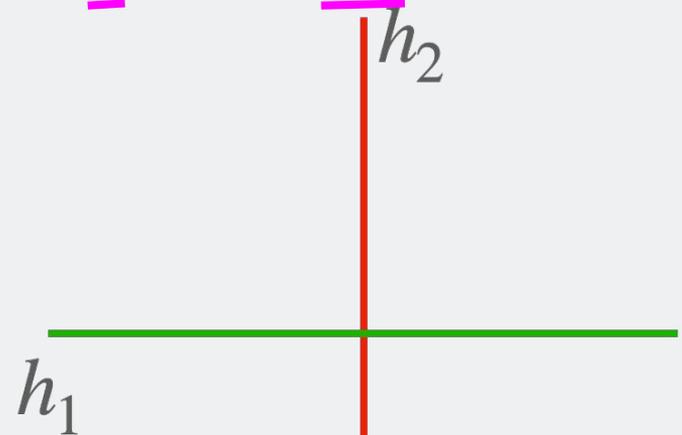
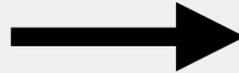
# Space Warping



Consider a linear transform:  $h = Wx + b$   
where  $x, b, h$  are each 2-dimensional

Feature transform:

$$h = Wx + b$$

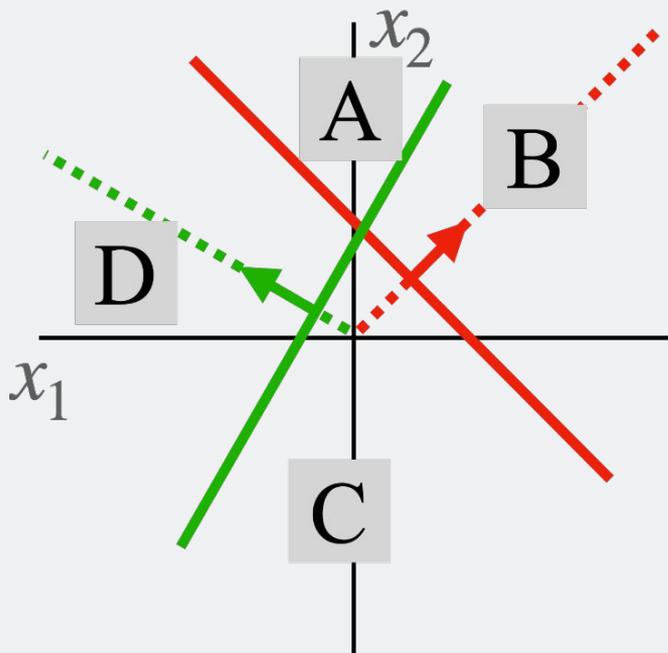


Green line -  $h_1=0$

Red line -  $h_2=0$

Dash line: where  $h_1$  and  $h_2$   
are increasing fastest

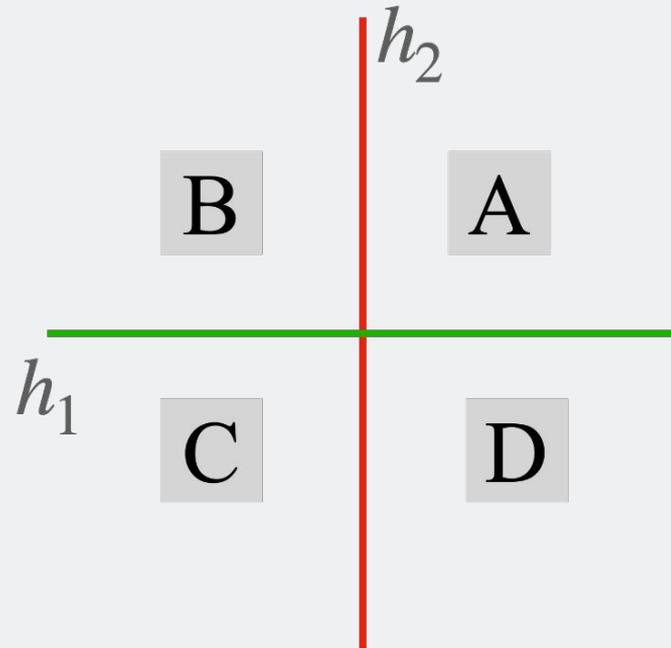
# Space Warping



Consider a linear transform:  $h = Wx + b$   
where  $x, b, h$  are each 2-dimensional

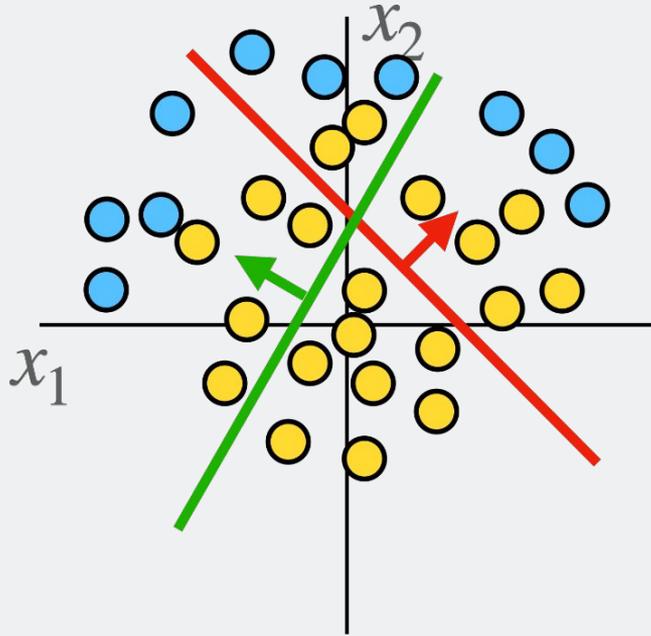
Feature transform:

$$h = Wx + b$$



# Space Warping

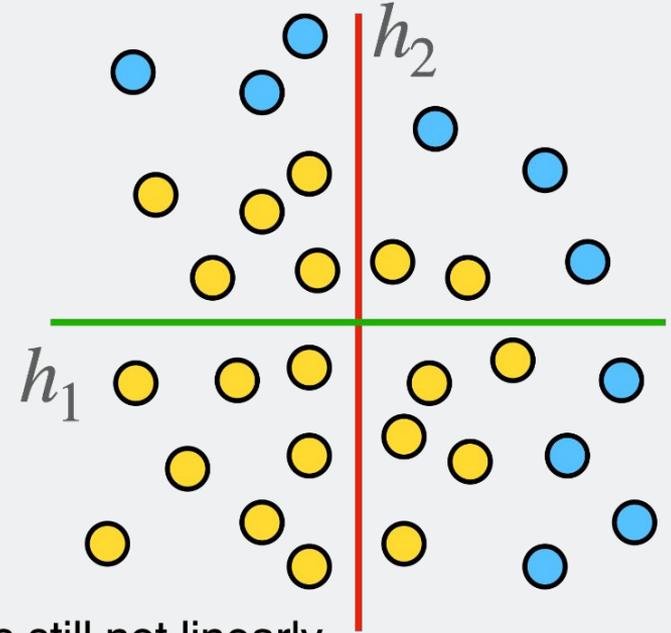
Points not linearly separable  
in original space



Consider a linear transform:  $h = Wx + b$   
where  $x, b, h$  are each 2-dimensional

Feature transform:

$$h = Wx + b$$

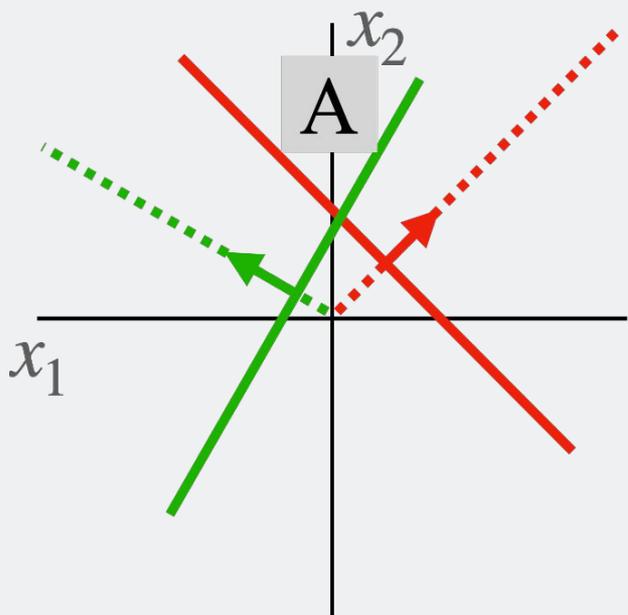


Why?

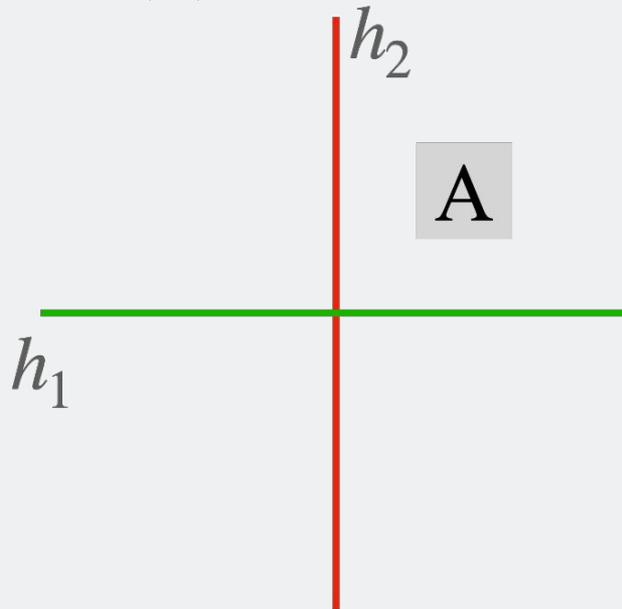
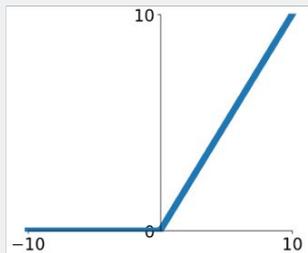
Points still not linearly  
separable in feature space

# Space Warping

Consider a neural net hidden layer:  $h = \text{ReLU}(Wx + b)$   
 $= \max(0, Wx + b)$  where  $x, b, h$  are each 2-dimensional

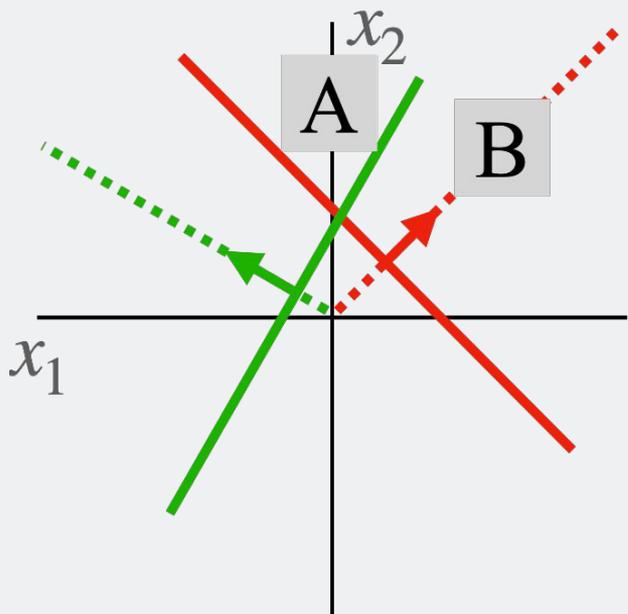


Feature transform:  
 $h = \text{ReLU}(Wx + b)$

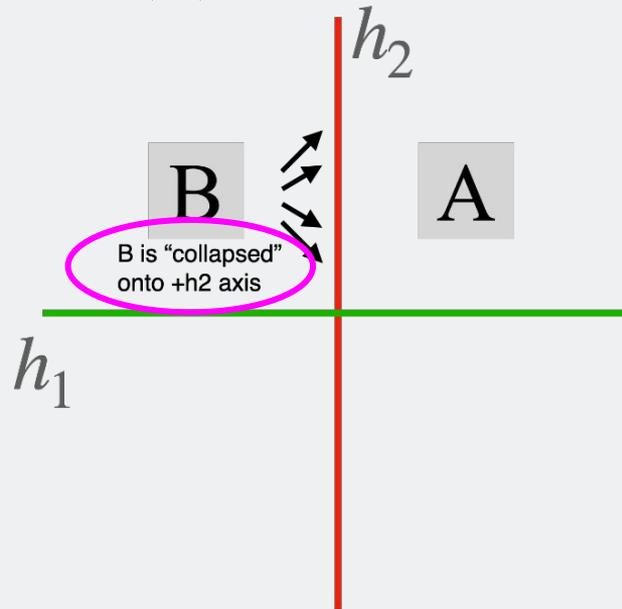
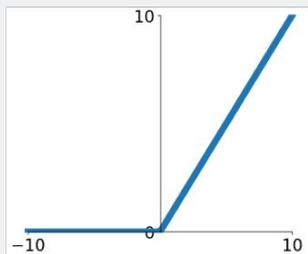


# Space Warping

Consider a neural net hidden layer:  $h = \text{ReLU}(Wx + b)$   
 $= \max(0, Wx + b)$  where  $x, b, h$  are each 2-dimensional

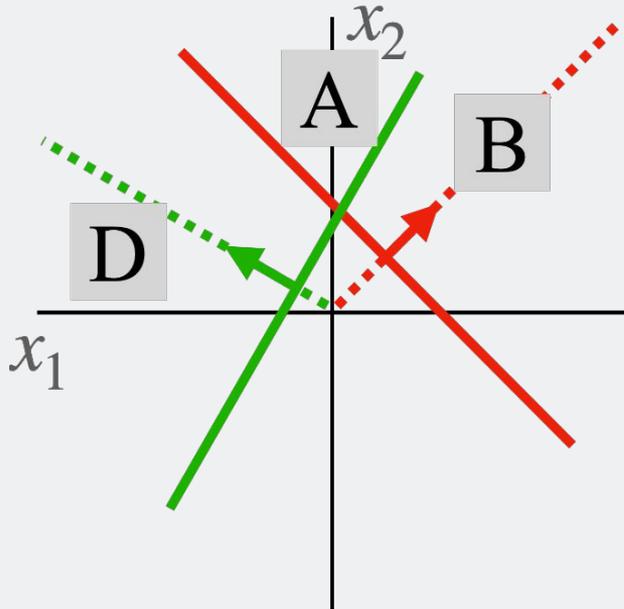


Feature transform:  
 $h = \text{ReLU}(Wx + b)$

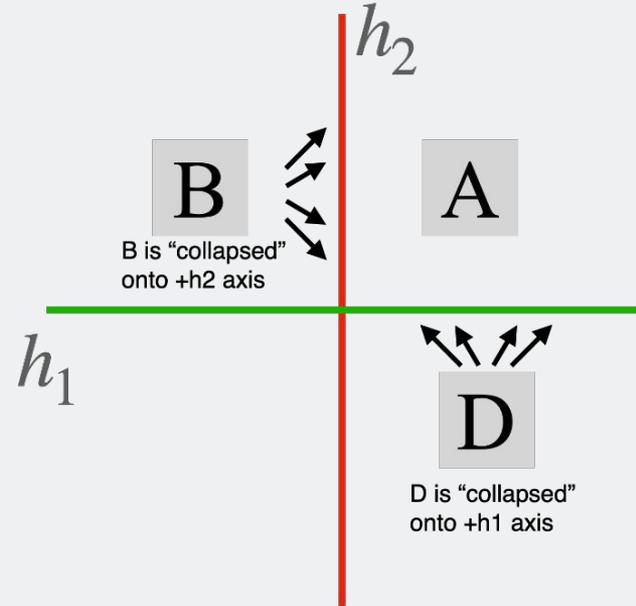
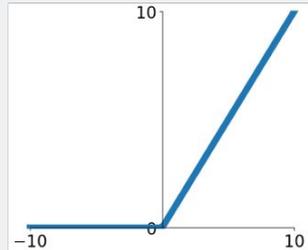


# Space Warping

Consider a neural net hidden layer:  $h = \text{ReLU}(Wx + b)$   
 $= \max(0, Wx + b)$  where  $x, b, h$  are each 2-dimensional

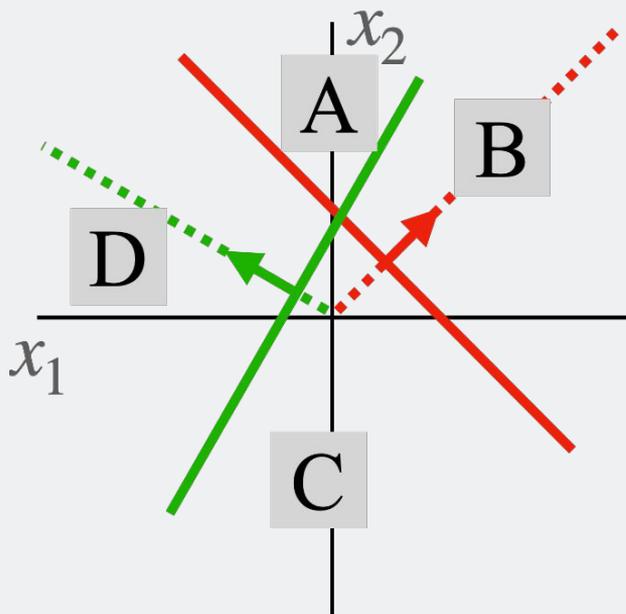


Feature transform:  
 $h = \text{ReLU}(Wx + b)$

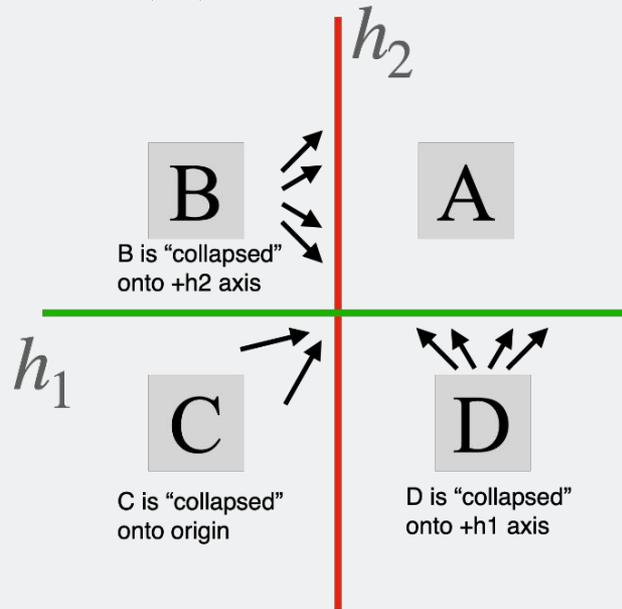
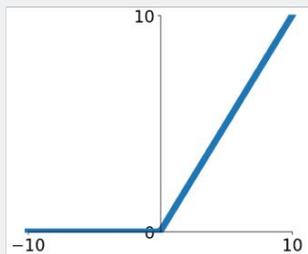


# Space Warping

Consider a neural net hidden layer:  $h = \text{ReLU}(Wx + b)$   
 $= \max(0, Wx + b)$  where  $x, b, h$  are each 2-dimensional



Feature transform:  
 $h = \text{ReLU}(Wx + b)$



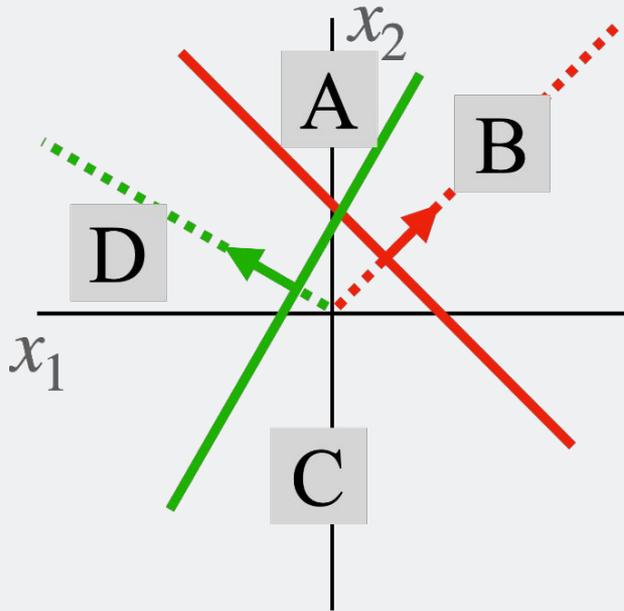
# Aha Slides (In-class participation)

<https://ahaslides.com/FSKOC>



Q1: Where would quadrant C map into?

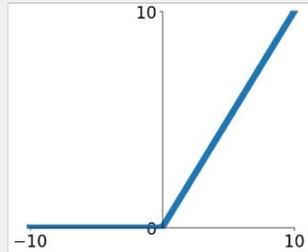
# Space Warping



Consider a neural net hidden layer:  $h = \text{ReLU}(Wx + b)$   
 $= \max(0, Wx + b)$  where  $x, b, h$  are each 2-dimensional

Feature transform:

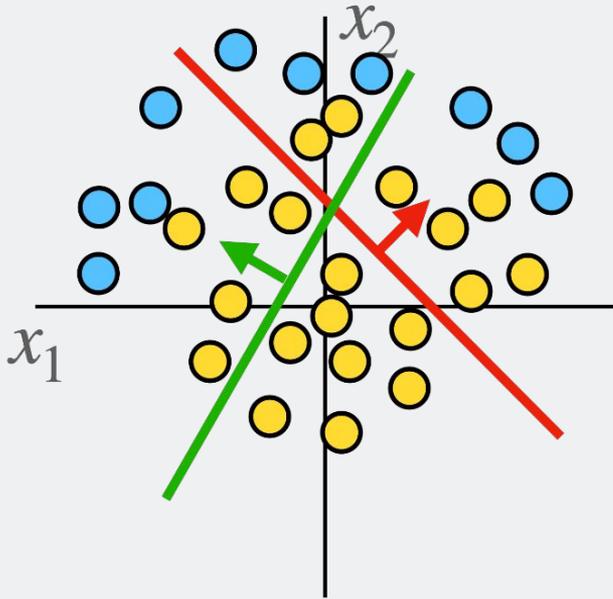
$$h = \text{ReLU}(Wx + b)$$



Q1: Where would quadrant **C** map into?

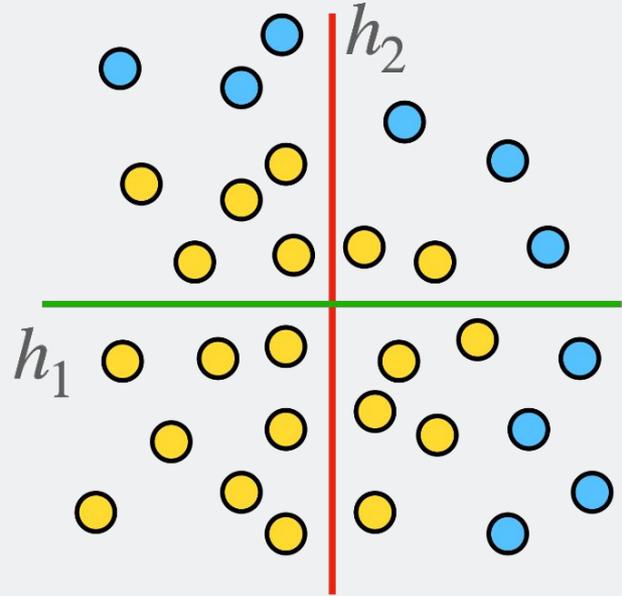
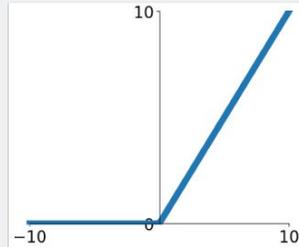
# Space Warping

Points not linearly separable in original space



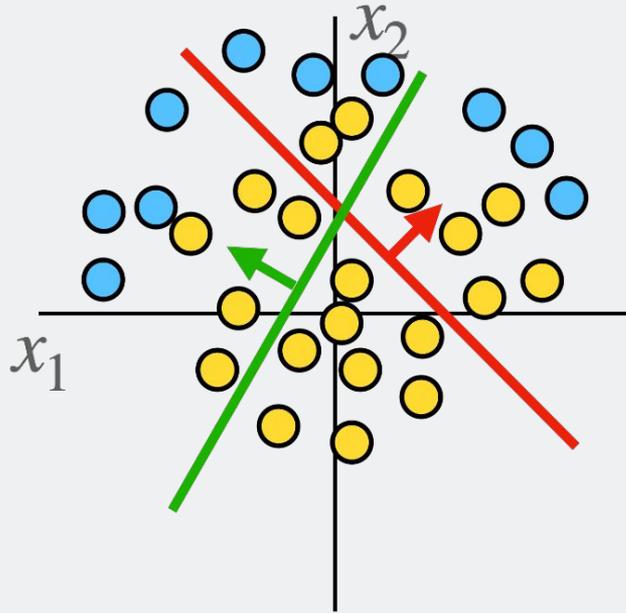
Consider a neural net hidden layer:  $h = \text{ReLU}(Wx + b)$   
 $= \max(0, Wx + b)$  where  $x, b, h$  are each 2-dimensional

Feature transform:  
 $h = \text{ReLU}(Wx + b)$



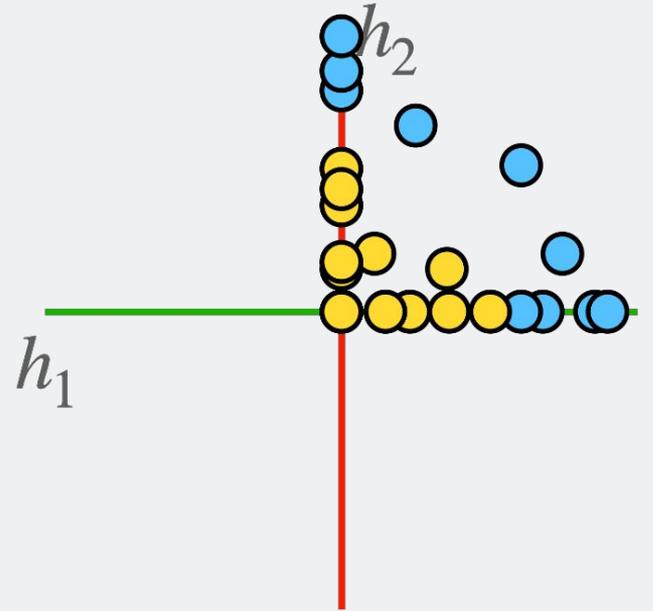
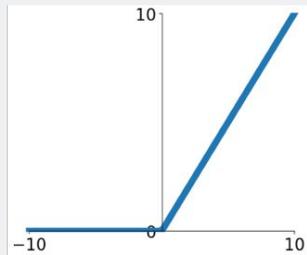
# Space Warping

Points not linearly separable in original space



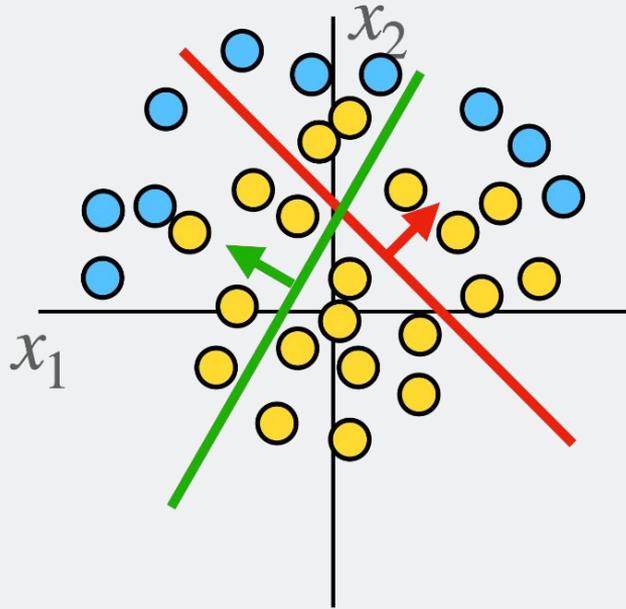
Consider a neural net hidden layer:  $h = \text{ReLU}(Wx + b)$   
 $= \max(0, Wx + b)$  where  $x, b, h$  are each 2-dimensional

Feature transform:  
 $h = \text{ReLU}(Wx + b)$



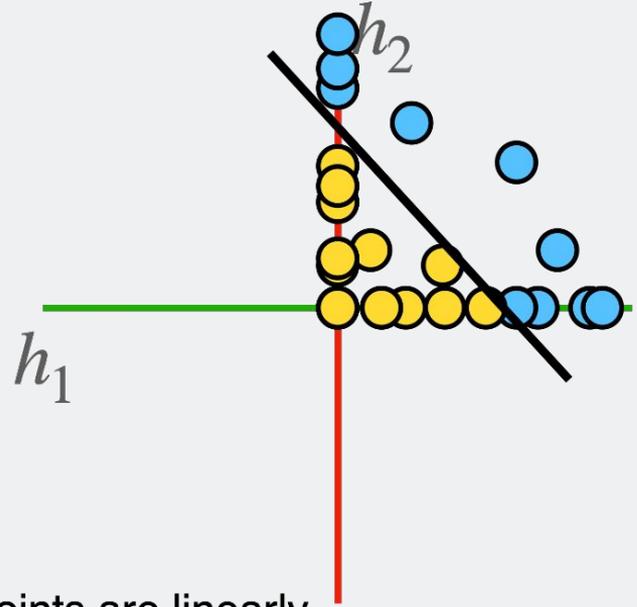
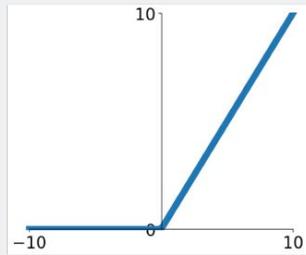
# Space Warping

Points not linearly separable in original space



Consider a neural net hidden layer:  $h = \text{ReLU}(Wx + b)$   
 $= \max(0, Wx + b)$  where  $x, b, h$  are each 2-dimensional

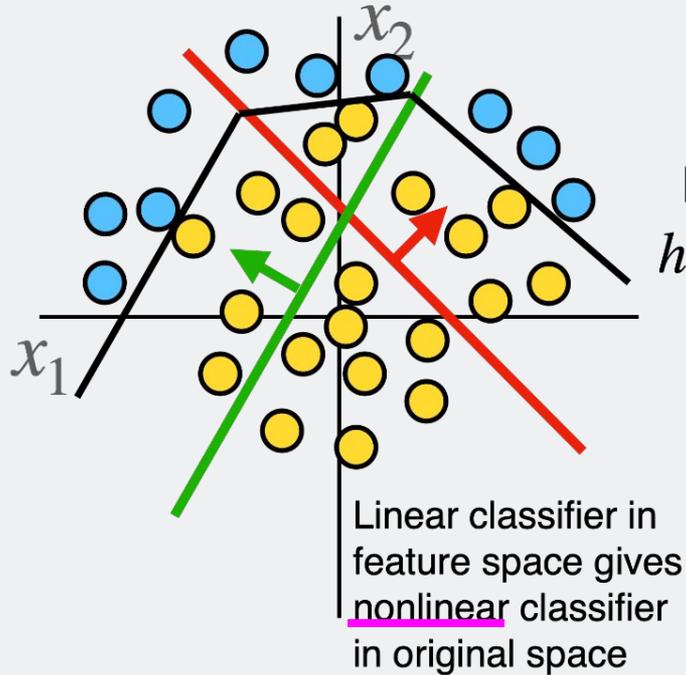
Feature transform:  
 $h = \text{ReLU}(Wx + b)$



Points are linearly separable in feature space!

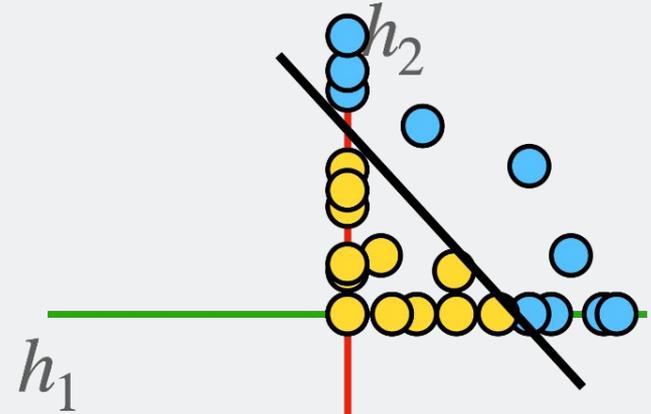
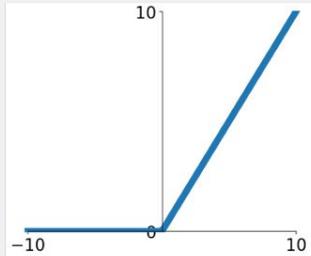
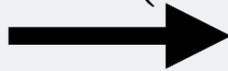
# Space Warping

Points not linearly separable in original space



Consider a neural net hidden layer:  $h = \text{ReLU}(Wx + b)$   
 $= \max(0, Wx + b)$  where  $x, b, h$  are each 2-dimensional

Feature transform:  
 $h = \text{ReLU}(Wx + b)$



Points are linearly separable in feature space!

# Aha Slides (In-class participation)

<https://ahaslides.com/FSKOC>

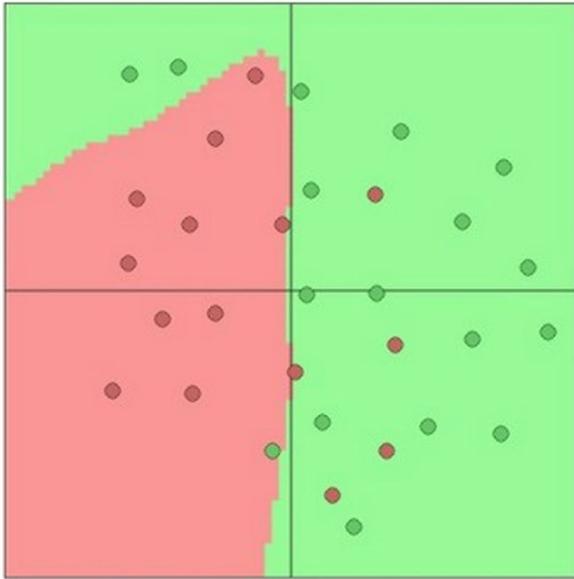


Q2, Q3: complexity

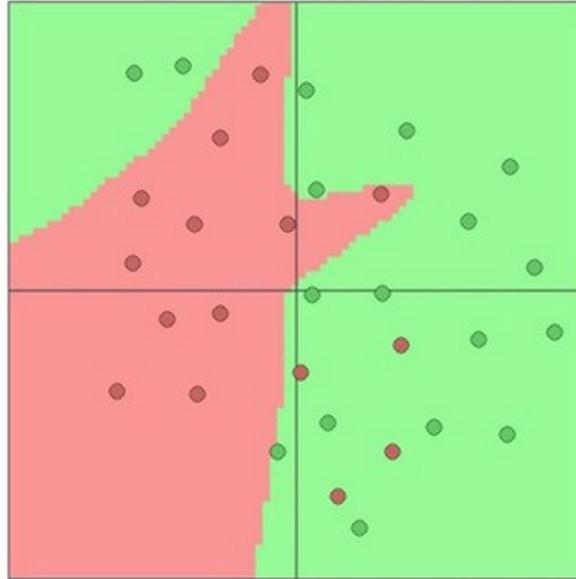
# Setting the number of layers and their sizes

---

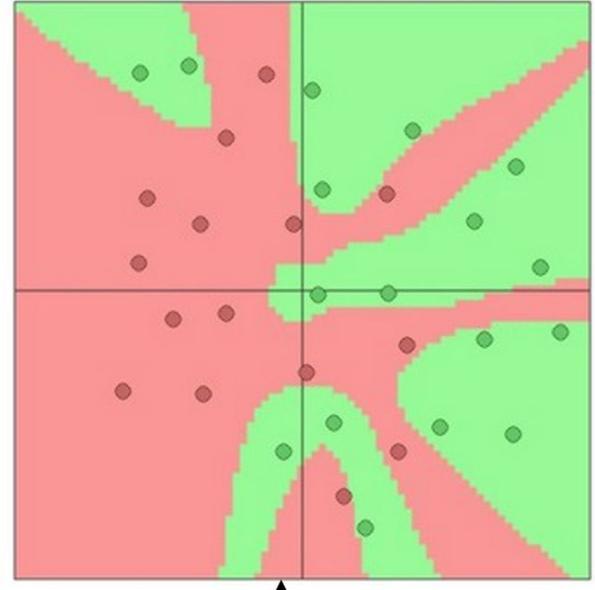
3 hidden units



6 hidden units



20 hidden units

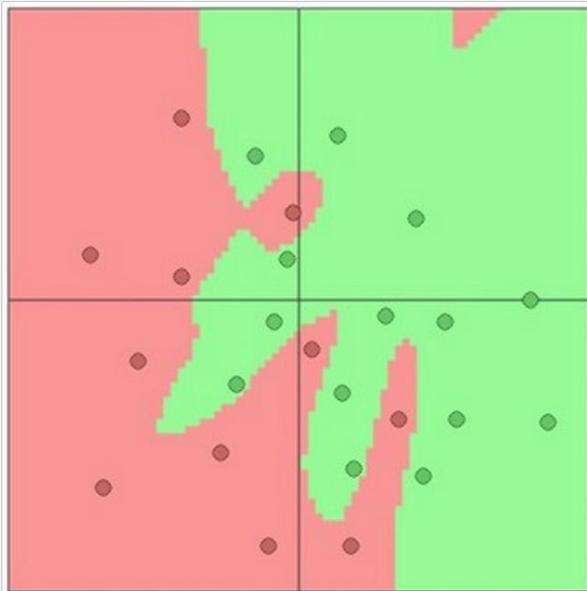


More hidden units = more capacity

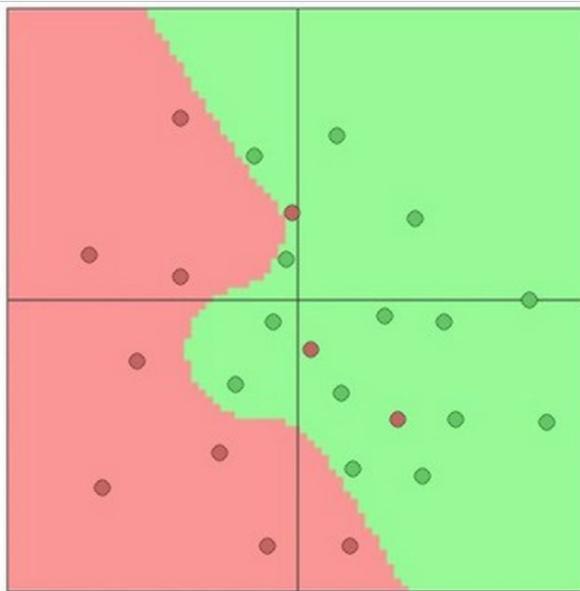
# Don't regularize with size; instead use stronger L2

---

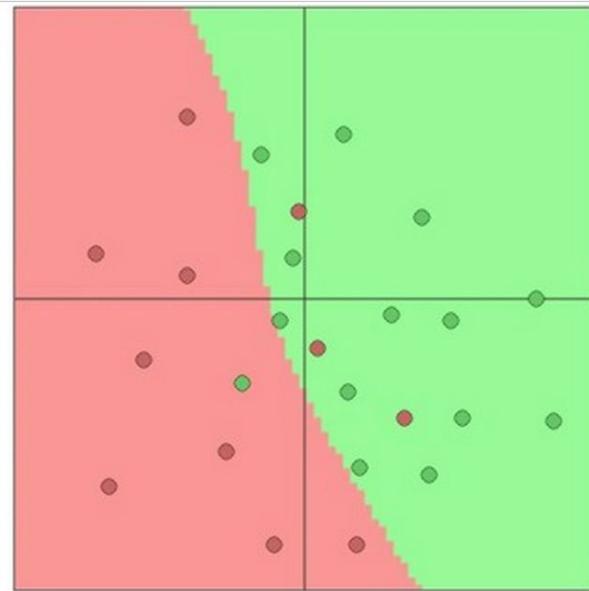
$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.1$



Web demo with ConvNetJS: <https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>

# Universal Approximation

# Universal Approximation

---

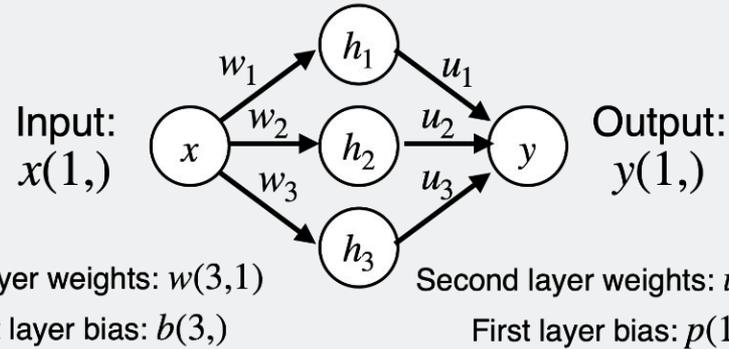
A neural network with one hidden layer can approximate any function  $f : \mathbb{R}^N \rightarrow \mathbb{R}^M$  with arbitrary precision\*

\*Many technical conditions: Only holds on compact subsets of  $\mathbb{R}^N$ ; function must be continuous; need to define "arbitrary precision"; etc.

# Universal Approximation

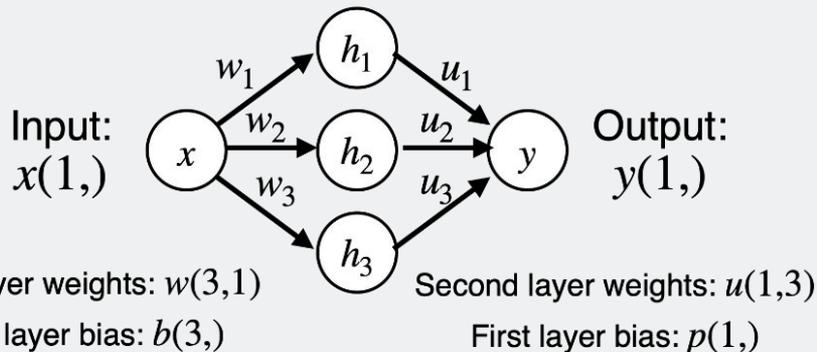
---

Example: Approximating a function  $f: \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



$$h_1 = \max(0, w_1 x + b_1)$$

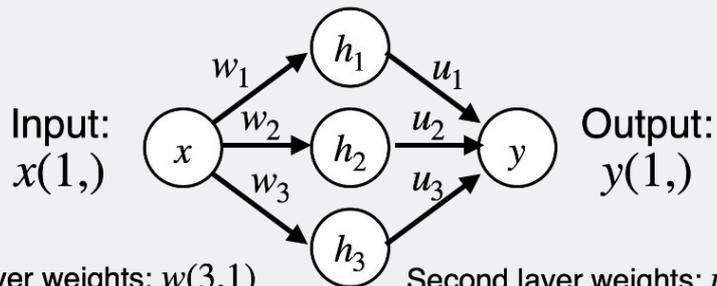
$$h_2 = \max(0, w_2 x + b_2)$$

$$h_3 = \max(0, w_3 x + b_3)$$

$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

Second layer weights:  $u(1,3)$

First layer bias:  $b(3,)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$

$$y = u_1 \max(0, w_1 x + b_1)$$

$$h_2 = \max(0, w_2 x + b_2)$$

$$+ u_2 \max(0, w_2 x + b_2)$$

$$h_3 = \max(0, w_3 x + b_3)$$

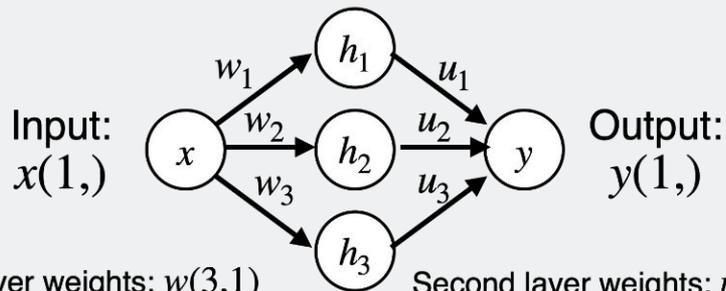
$$+ u_3 \max(0, w_3 x + b_3)$$

$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$+ p$$

# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

First layer bias:  $b(3,)$

$$h_1 = \max(0, w_1x + b_1)$$

$$h_2 = \max(0, w_2x + b_2)$$

$$h_3 = \max(0, w_3x + b_3)$$

$$y = u_1h_1 + u_2h_2 + u_3h_3 + p$$

Second layer weights:  $u(1,3)$

First layer bias:  $p(1,)$

$$y = u_1 \max(0, w_1x + b_1)$$

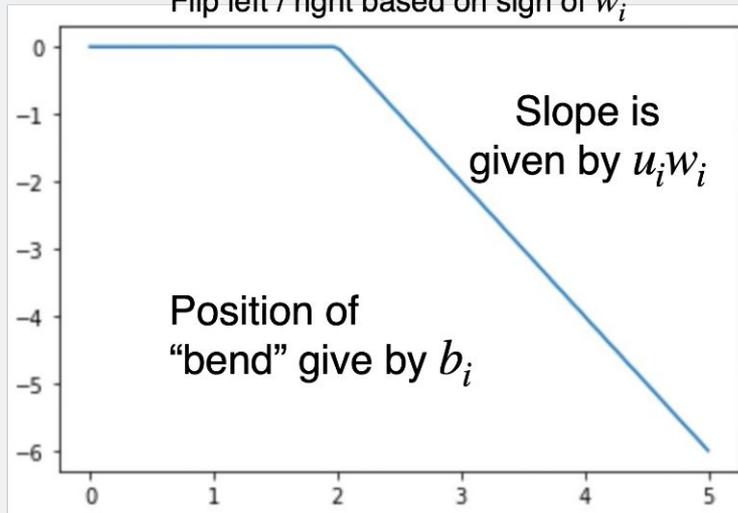
$$+ u_2 \max(0, w_2x + b_2)$$

$$+ u_3 \max(0, w_3x + b_3)$$

$$+ p$$

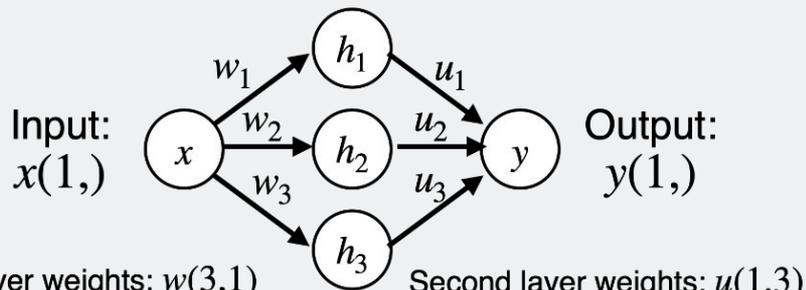
Output is a sum of shifted,  
scaled ReLUs:

Flip left / right based on sign of  $w_i$



# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer bias:  $b(3,)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1x + b_1)$$

$$h_2 = \max(0, w_2x + b_2)$$

$$h_3 = \max(0, w_3x + b_3)$$

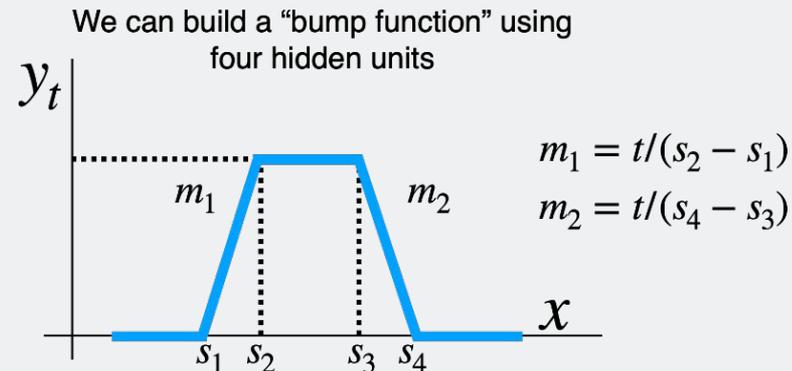
$$y = u_1h_1 + u_2h_2 + u_3h_3 + p$$

$$y = u_1 \max(0, w_1x + b_1)$$

$$+ u_2 \max(0, w_2x + b_2)$$

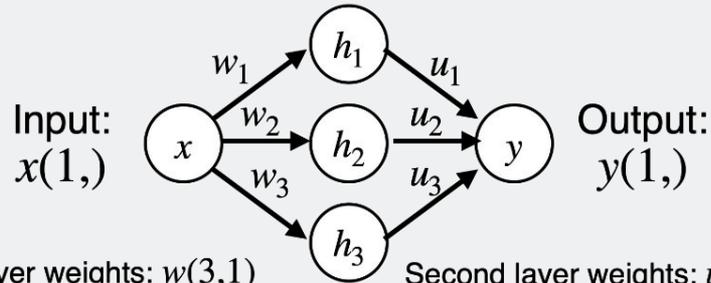
$$+ u_3 \max(0, w_3x + b_3)$$

$$+ p$$



# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

First layer bias:  $b(3,)$

Second layer weights:  $u(1,3)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$

$$h_2 = \max(0, w_2 x + b_2)$$

$$h_3 = \max(0, w_3 x + b_3)$$

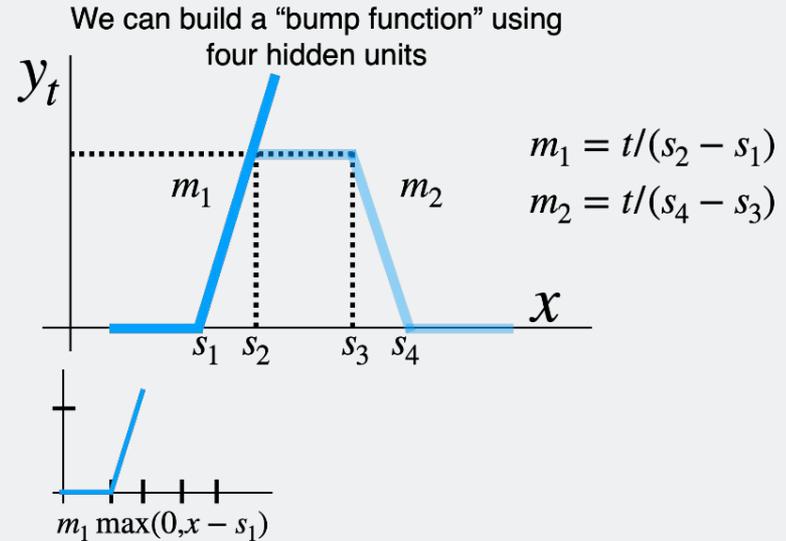
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1)$$

$$+ u_2 \max(0, w_2 x + b_2)$$

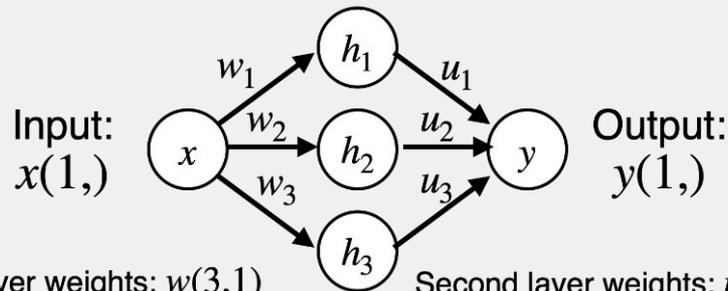
$$+ u_3 \max(0, w_3 x + b_3)$$

$$+ p$$



# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

Second layer weights:  $u(1,3)$

First layer bias:  $b(3,)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$

$$y = u_1 \max(0, w_1 x + b_1)$$

$$h_2 = \max(0, w_2 x + b_2)$$

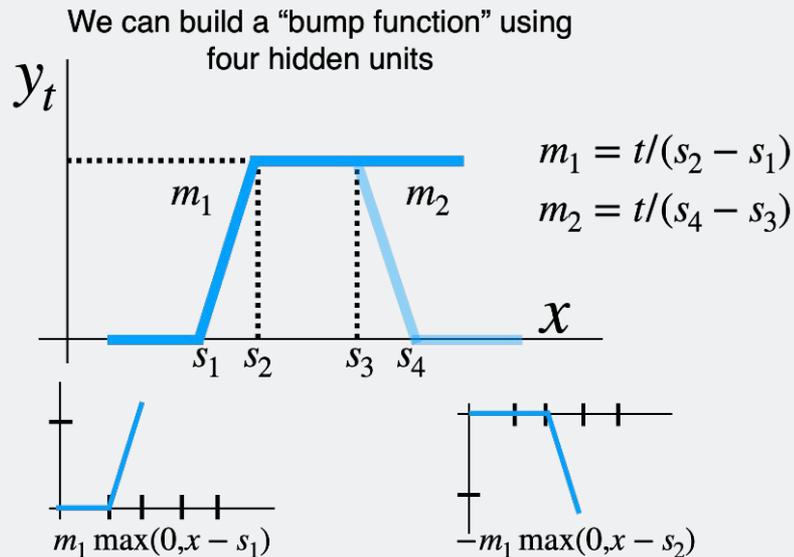
$$+ u_2 \max(0, w_2 x + b_2)$$

$$h_3 = \max(0, w_3 x + b_3)$$

$$+ u_3 \max(0, w_3 x + b_3)$$

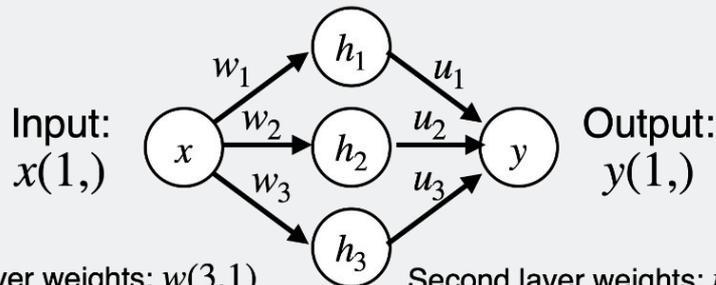
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$+ p$$



# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

First layer bias:  $b(3,)$

Second layer weights:  $u(1,3)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$

$$h_2 = \max(0, w_2 x + b_2)$$

$$h_3 = \max(0, w_3 x + b_3)$$

$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

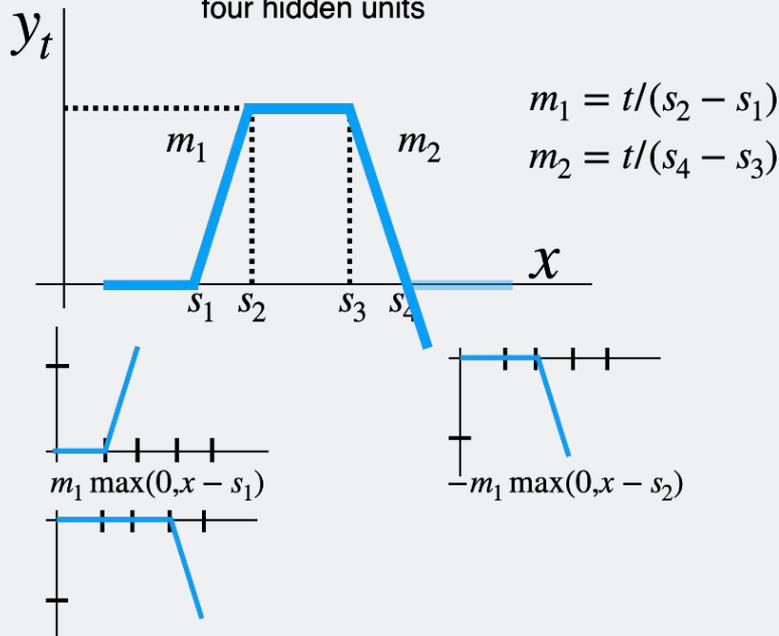
$$y = u_1 \max(0, w_1 x + b_1)$$

$$+ u_2 \max(0, w_2 x + b_2)$$

$$+ u_3 \max(0, w_3 x + b_3)$$

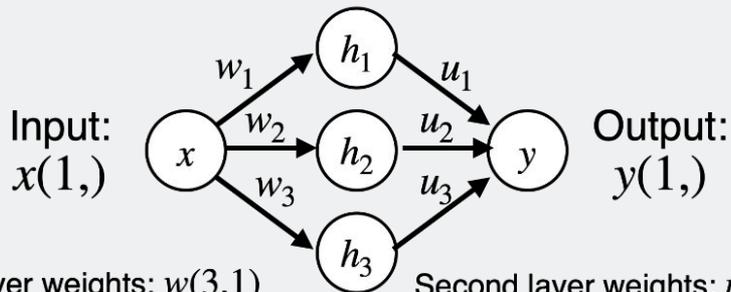
$$+ p$$

We can build a “bump function” using four hidden units



# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

First layer bias:  $b(3,)$

Second layer weights:  $u(1,3)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$

$$h_2 = \max(0, w_2 x + b_2)$$

$$h_3 = \max(0, w_3 x + b_3)$$

$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

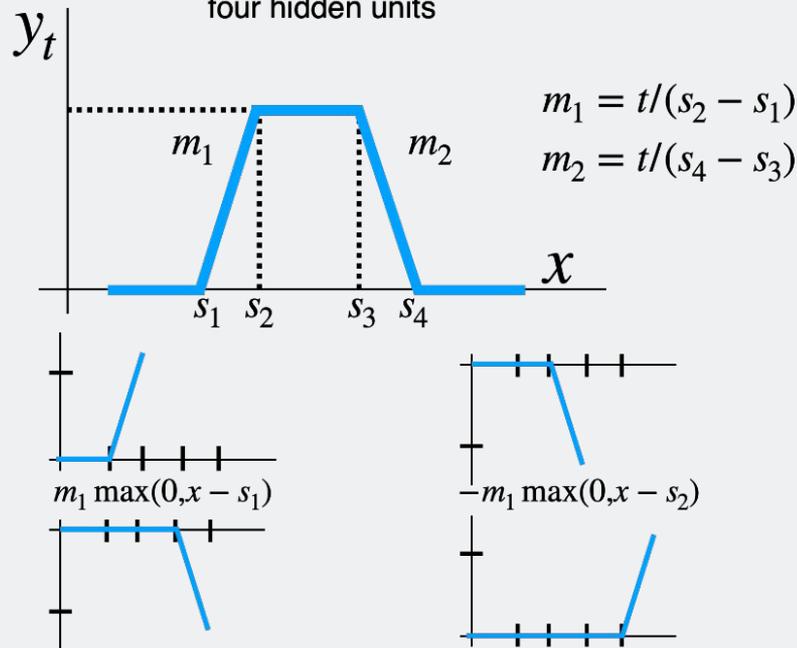
$$y = u_1 \max(0, w_1 x + b_1)$$

$$+ u_2 \max(0, w_2 x + b_2)$$

$$+ u_3 \max(0, w_3 x + b_3)$$

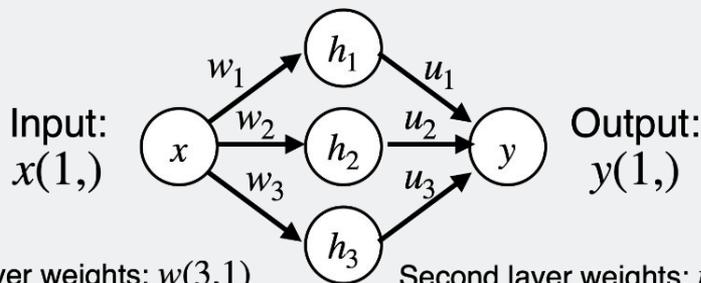
$$+ p$$

We can build a “bump function” using four hidden units



# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

First layer bias:  $b(3,)$

Second layer weights:  $u(1,3)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1x + b_1)$$

$$h_2 = \max(0, w_2x + b_2)$$

$$h_3 = \max(0, w_3x + b_3)$$

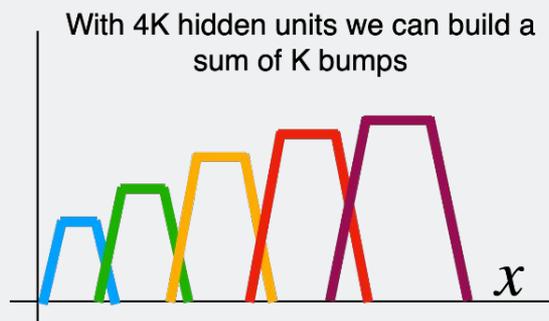
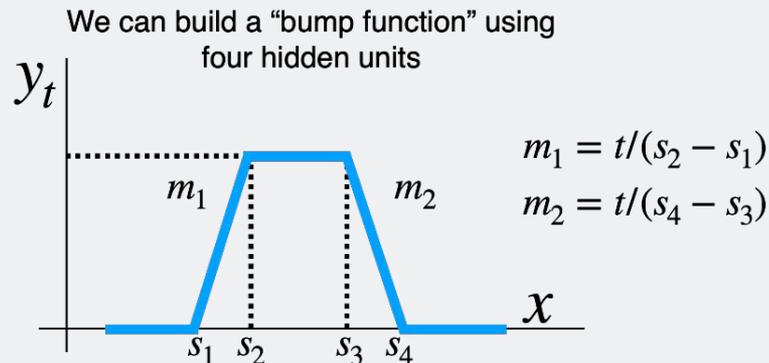
$$y = u_1h_1 + u_2h_2 + u_3h_3 + p$$

$$y = u_1 \max(0, w_1x + b_1)$$

$$+ u_2 \max(0, w_2x + b_2)$$

$$+ u_3 \max(0, w_3x + b_3)$$

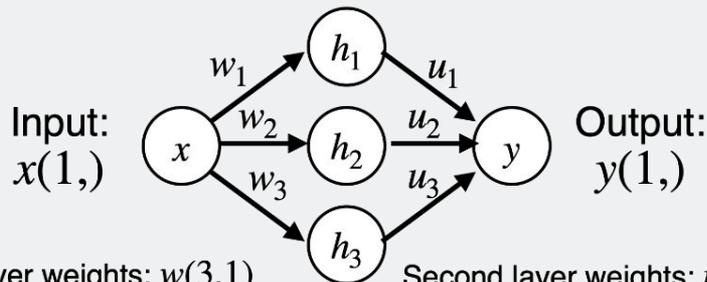
$$+ p$$



Approximate functions with bumps!

# Universal Approximation

Example: Approximating a function  $f: \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

First layer bias:  $b(3,)$

Second layer weights:  $u(1,3)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1x + b_1)$$

$$h_2 = \max(0, w_2x + b_2)$$

$$h_3 = \max(0, w_3x + b_3)$$

$$y = u_1h_1 + u_2h_2 + u_3h_3 + p$$

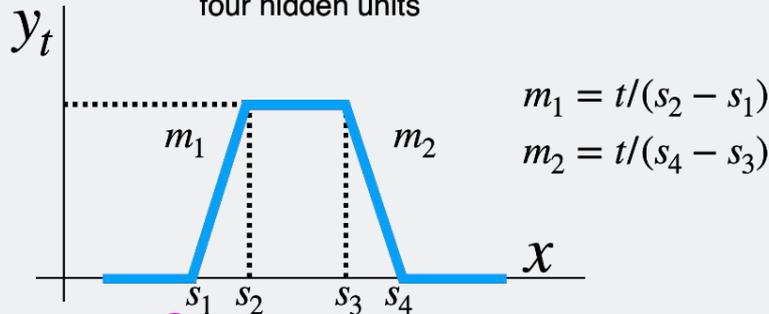
$$y = u_1 \max(0, w_1x + b_1)$$

$$+ u_2 \max(0, w_2x + b_2)$$

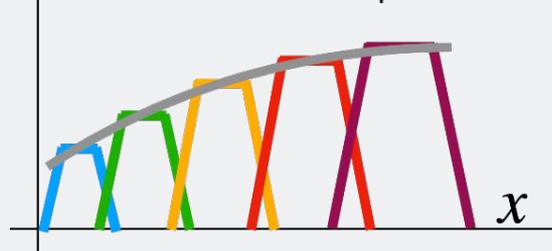
$$+ u_3 \max(0, w_3x + b_3)$$

$$+ p$$

We can build a “bump function” using four hidden units



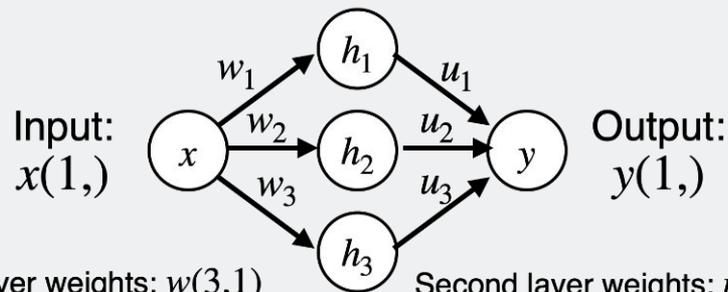
With 4K hidden units we can build a sum of K bumps



Approximate functions with bumps!

# Universal Approximation

Example: Approximating a function  $f : \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

Second layer weights:  $u(1,3)$

First layer bias:  $b(3,)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1x + b_1)$$

$$h_2 = \max(0, w_2x + b_2)$$

$$h_3 = \max(0, w_3x + b_3)$$

$$y = u_1h_1 + u_2h_2 + u_3h_3 + p$$

$$y = u_1 \max(0, w_1x + b_1)$$

$$+ u_2 \max(0, w_2x + b_2)$$

$$+ u_3 \max(0, w_3x + b_3)$$

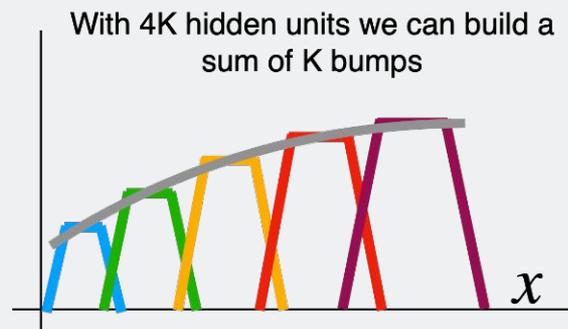
$$+ p$$

What about ...

- Gaps between bumps?
- Other nonlinearities?
- Higher-dimensional functions?

See [Nielsen, Chapter 4](http://neuralnetworksanddeeplearning.com/chap4.html)

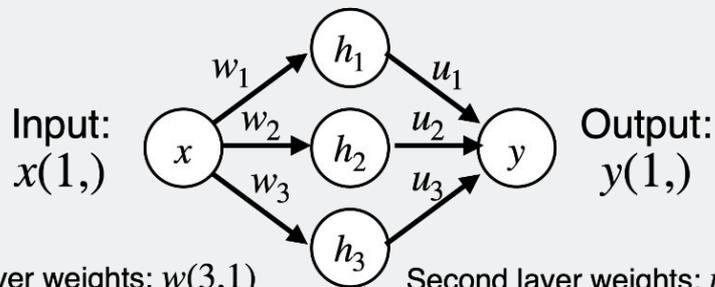
<http://neuralnetworksanddeeplearning.com/chap4.html>



Approximate functions with bumps!

# Universal Approximation

Example: Approximating a function  $f: \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



First layer weights:  $w(3,1)$

First layer bias:  $b(3,)$

Second layer weights:  $u(1,3)$

First layer bias:  $p(1,)$

$$h_1 = \max(0, w_1x + b_1)$$

$$h_2 = \max(0, w_2x + b_2)$$

$$h_3 = \max(0, w_3x + b_3)$$

$$y = u_1h_1 + u_2h_2 + u_3h_3 + p$$

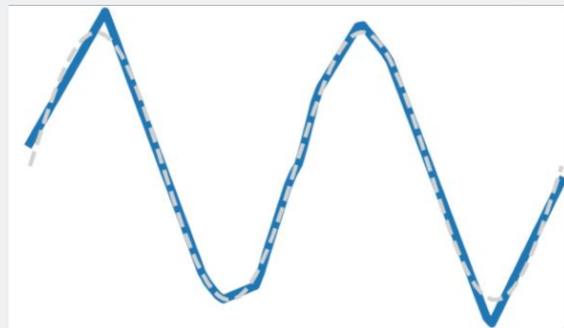
$$y = u_1 \max(0, w_1x + b_1)$$

$$+ u_2 \max(0, w_2x + b_2)$$

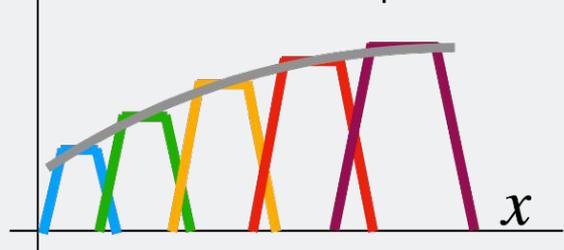
$$+ u_3 \max(0, w_3x + b_3)$$

$$+ p$$

Reality check: Networks don't really learn bumps!



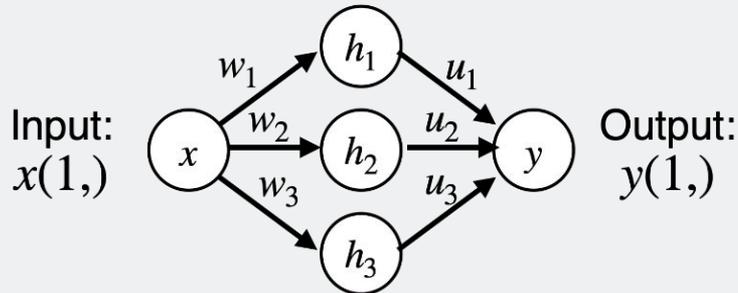
With 4K hidden units we can build a sum of K bumps



Approximate functions with bumps!

# Universal Approximation

Example: Approximating a function  $f: \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



Universal approximation tells us:

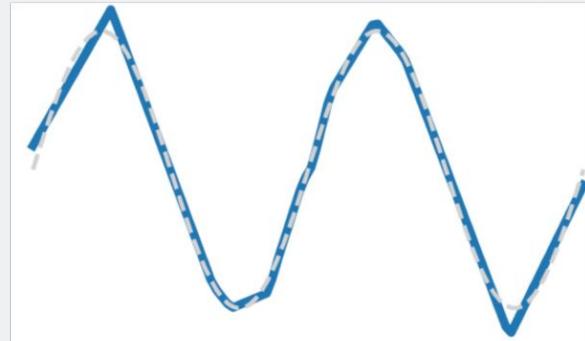
- Neural nets can represent any function “In principle”

Universal approximation **DOES NOT** tell us: “In practice”

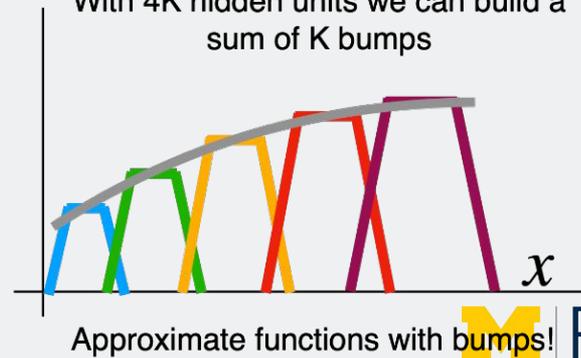
- Whether we can actually learn any function with SGD
- How much data we need to learn a function

Remember: kNN is also a universal approximator!

Reality check: Networks don't really learn bumps!



With 4K hidden units we can build a sum of K bumps



# Aha Slides (In-class participation)

<https://ahaslides.com/FSKOC>



Q4: sigmoid activation and universal approximator

# Convex Functions

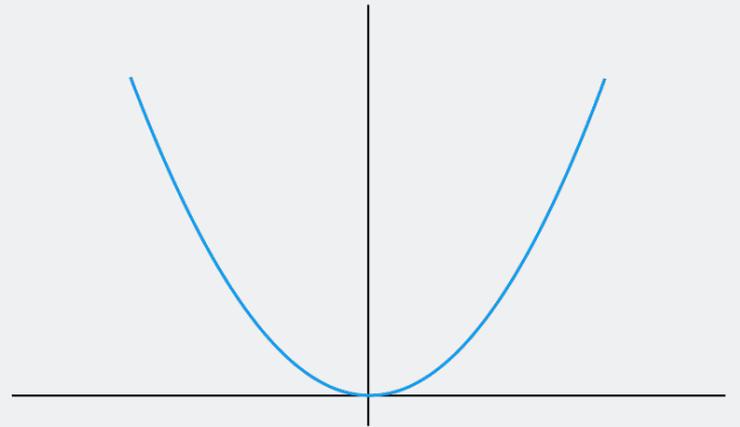
# Convex Functions

---

A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

Example:  $f(x) = x^2$  is convex:

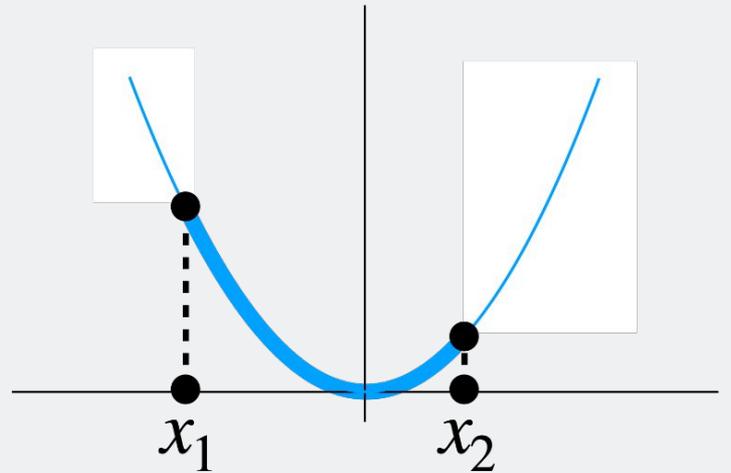


# Convex Functions

---

A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,  
 $f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$

Example:  $f(x) = x^2$  is convex:



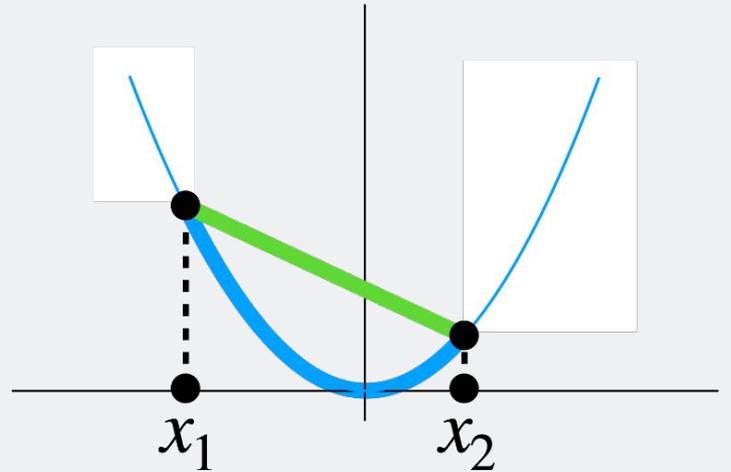
# Convex Functions

---

A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

Example:  $f(x) = x^2$  is convex:



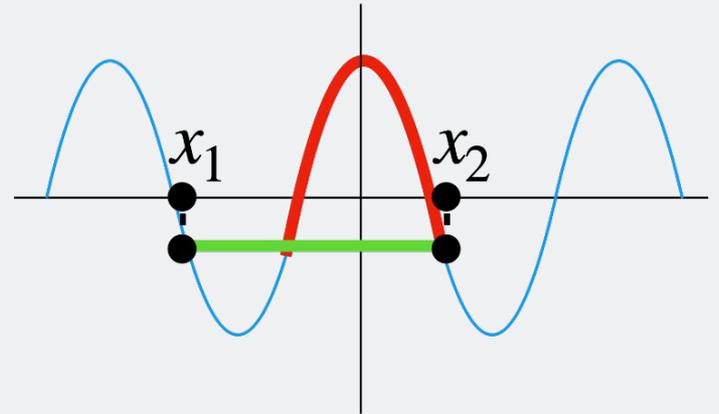
# Convex Functions

---

A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

Example:  $f(x) = \cos(x)$  is **not** convex:



# Convex Functions

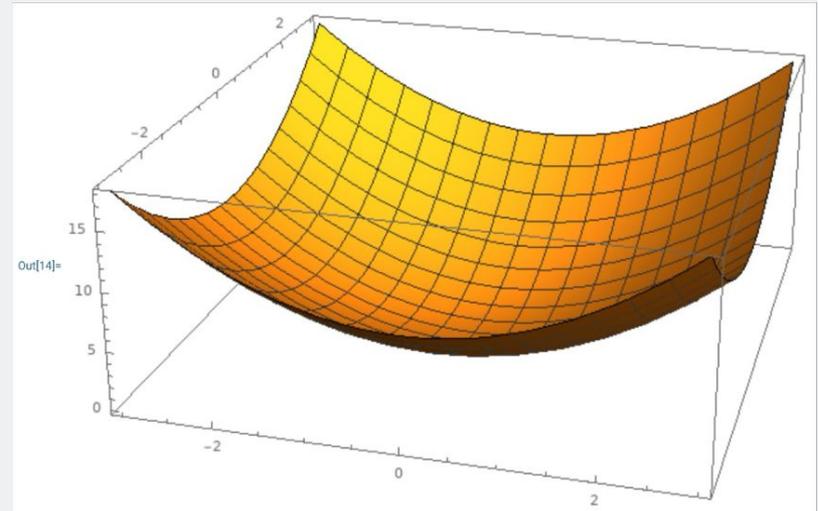
---

A function  $f: X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum\***



# Convex Functions

A function  $f: X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum\***

Linear classifiers optimize a **convex function!**

$$s = f(x; W) = Wx$$

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \quad \text{Softmax}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{SVM}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W) \quad \text{where } R(W) \text{ is L2 or L1 regularization}$$

# Convex Functions

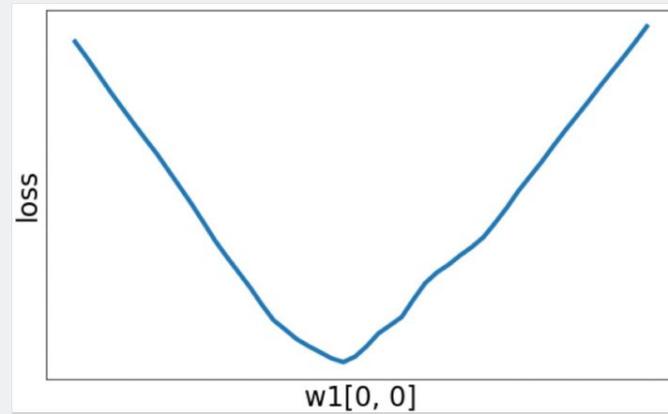
A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum\***

Neural net losses sometimes look convex-ish:



# Convex Functions

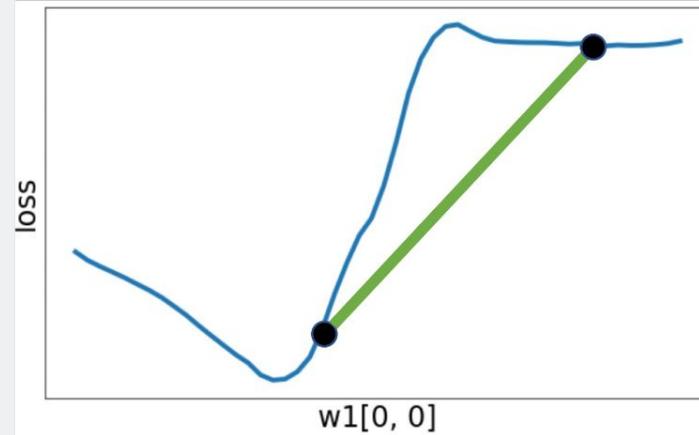
A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum\***

But often clearly nonconvex:



1D slice of loss landscape for a 4-layer ReLU network with 10 input features, 32 units per hidden layer, 10 categories, with softmax loss

# Convex Functions

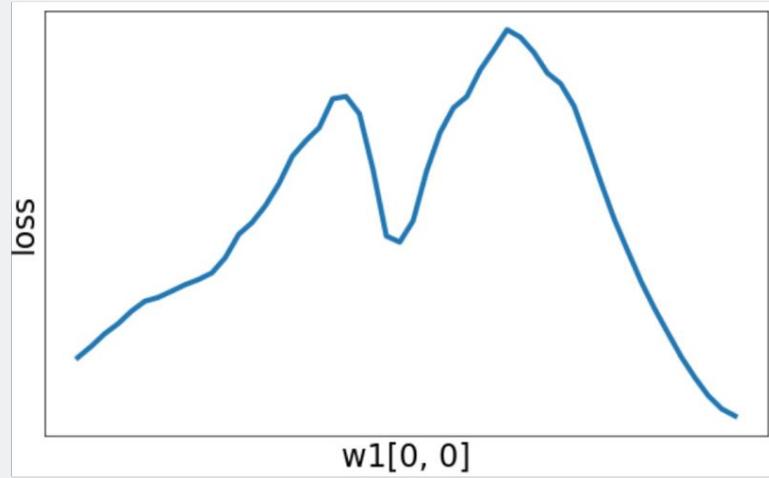
A function  $f: X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum\***

With local minima:



1D slice of loss landscape for a 4-layer ReLU network with 10 input features, 32 units per hidden layer, 10 categories, with softmax loss

# Convex Functions

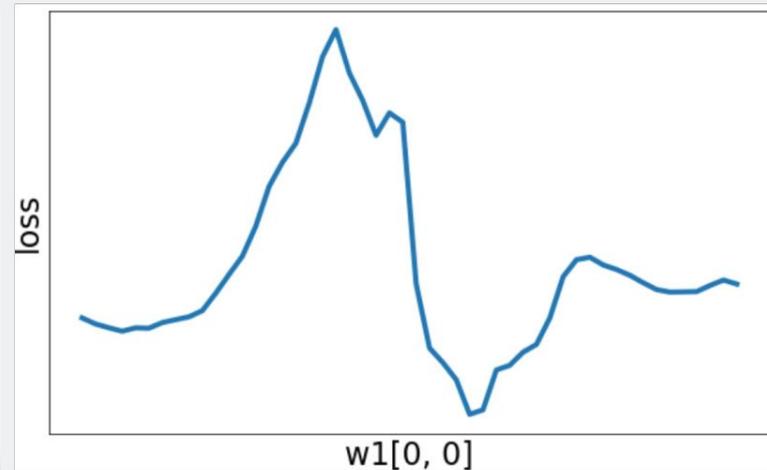
A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum\***

Can get very wild!



1D slice of loss landscape for a 4-layer ReLU network with 10 input features, 32 units per hidden layer, 10 categories, with softmax loss

# Convex Functions

---

A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

**Intuition:** A convex function is a (multidimensional) bowl

Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum\***

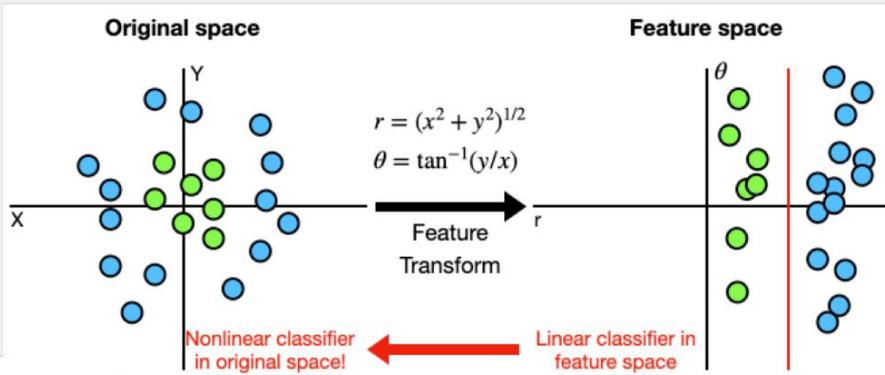
Most neural networks need **nonconvex optimization**

- Few or no guarantees about convergence
- Empirically it seems to work anyway
- Active area of research

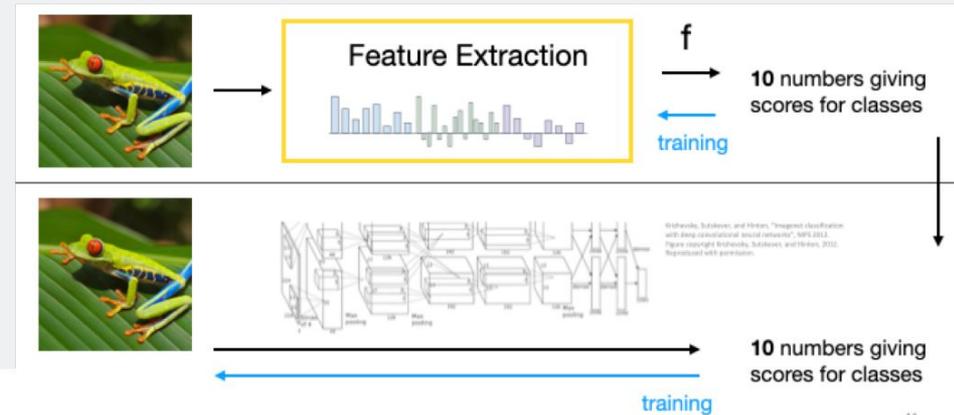
# Summary

# Summary

Feature transform + Linear classifier allows nonlinear decision boundaries



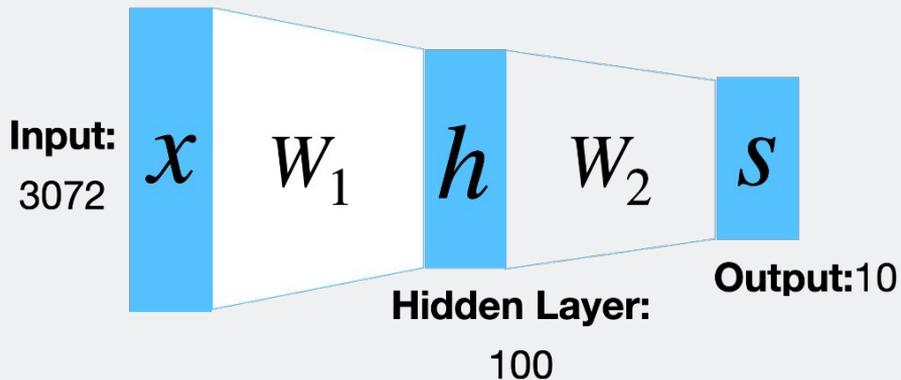
Neural Networks as learnable feature transforms



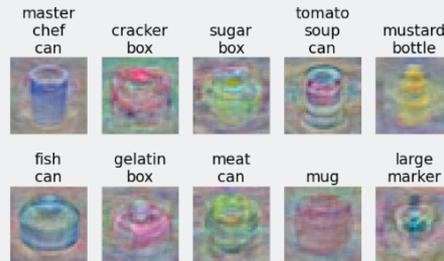
# Summary

From linear classifiers to  
fully-connected networks

$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$



Linear classifier: One template per class



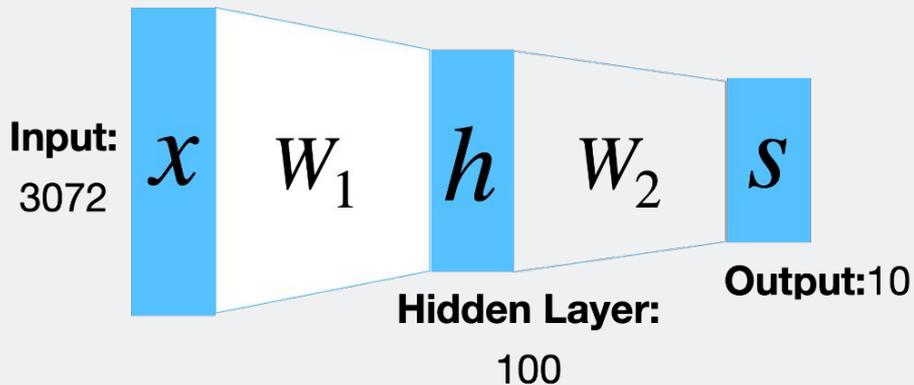
Neural networks: Many reusable templates



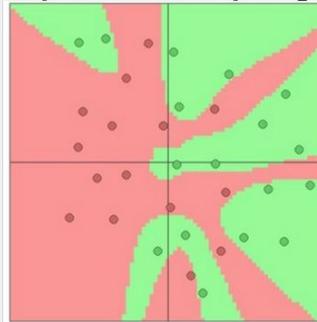
# Summary

From linear classifiers to fully-connected networks

$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$



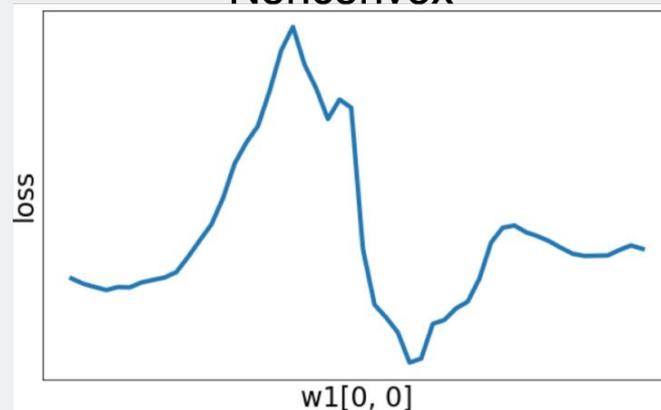
Space Warping



Universal approximation



Nonconvex



# Problem: How to compute gradients?

---

$$s = W_2 \max(0, W_1 x + b_1) + b_2$$

Nonlinear score function

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

Per-element data loss

$$R(W) = \sum_k W_k^2$$

L2 regularization

$$L(W_1, W_2, b_1, b_2) = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W_1) + \lambda R(W_2) \text{ Total loss}$$

If we can compute  $\frac{\delta L}{\delta W_1}$ ,  $\frac{\delta L}{\delta W_2}$ ,  $\frac{\delta L}{\delta b_1}$ ,  $\frac{\delta L}{\delta b_2}$  then we can optimize with SGD

Next up: Backpropagation