

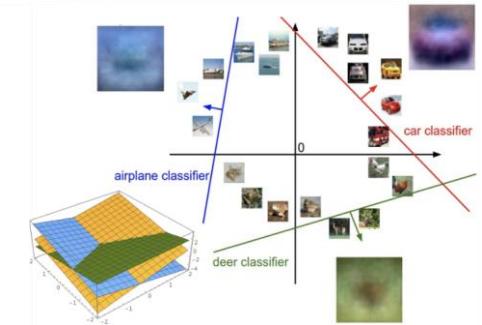
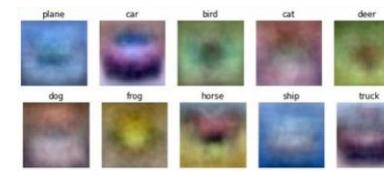
# Lecture 3: Neural Networks



Where we are:

1. Use **Linear Models** for image classification problems
2. Use **Loss Functions** to express preferences over different choices of weights
3. Use **Regularization** to prevent overfitting to training data
4. Use **Stochastic Gradient Descent** to minimize our loss functions and train the model

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

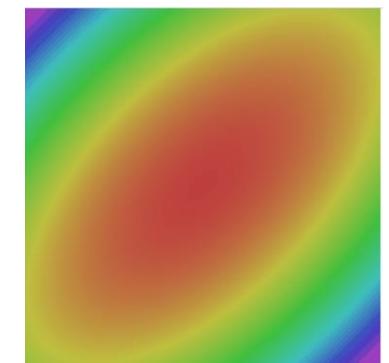


$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$
 Softmax

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

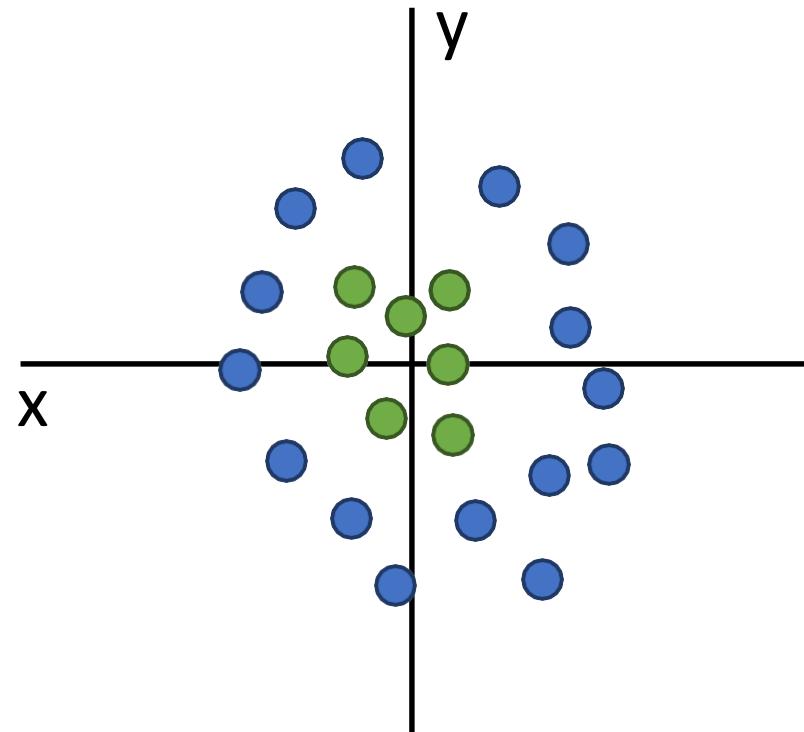
$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W)$$

```
v = 0
for t in range(num_steps):
    dw = compute_gradient(w)
    v = rho * v + dw
    w -= learning_rate * v
```



# Problem: Linear Classifiers aren't that powerful

## Geometric Viewpoint

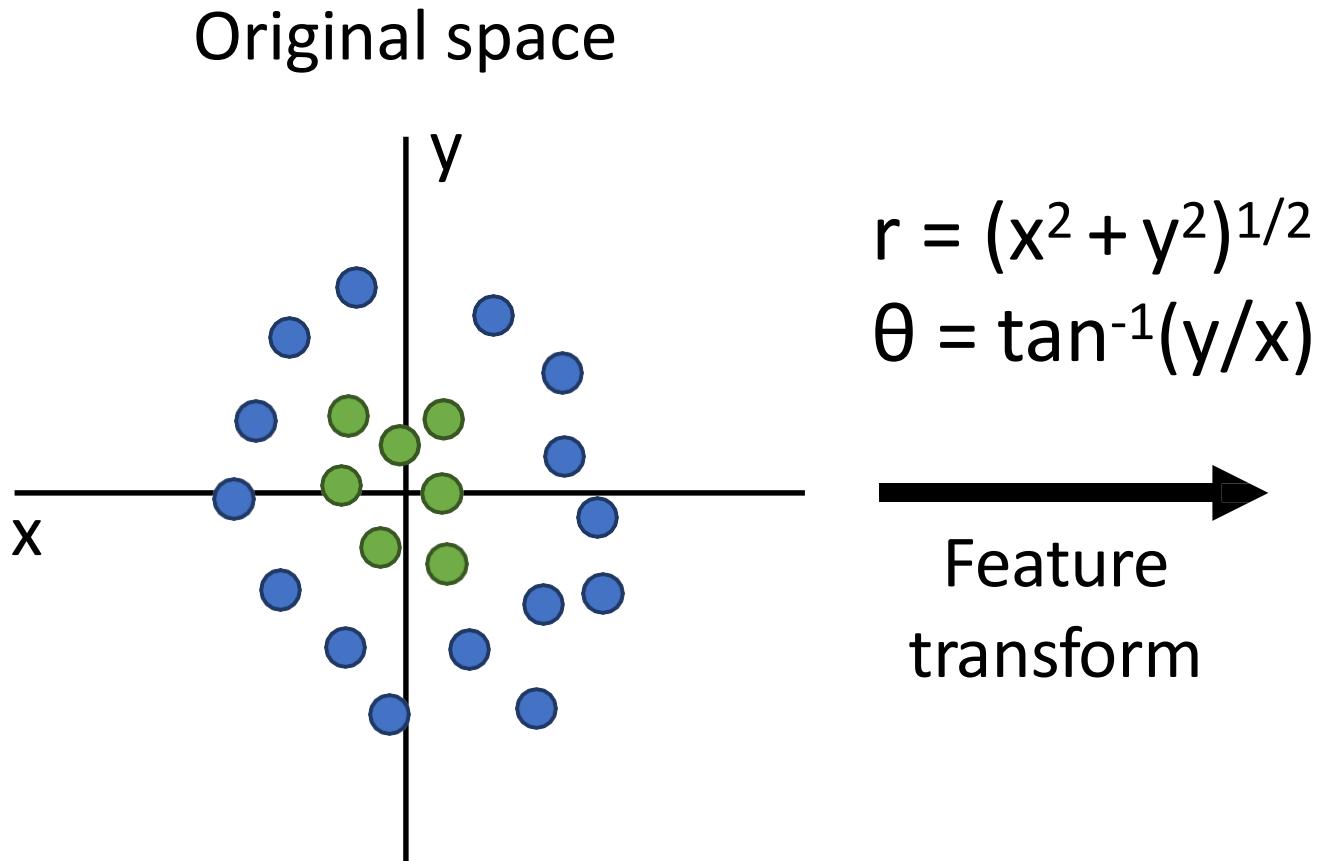


## Visual Viewpoint

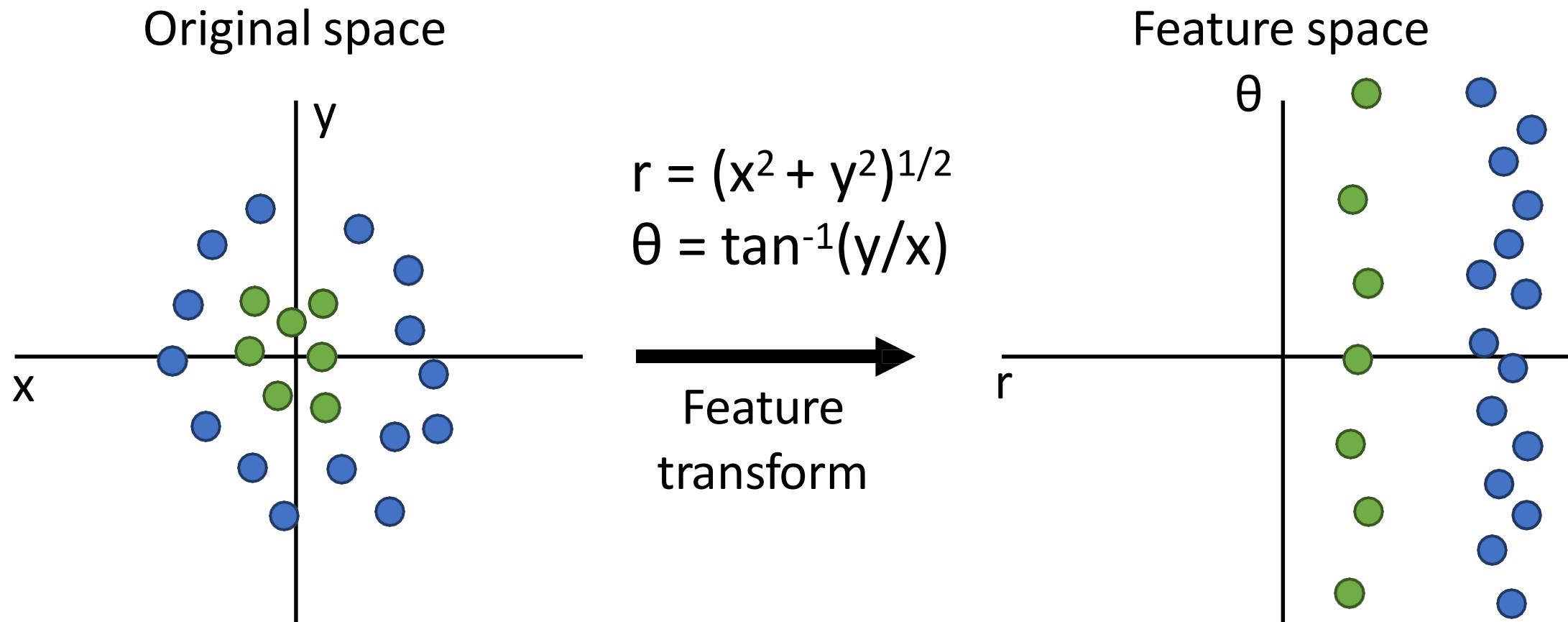
One template per class:  
Can't recognize different  
modes of a class



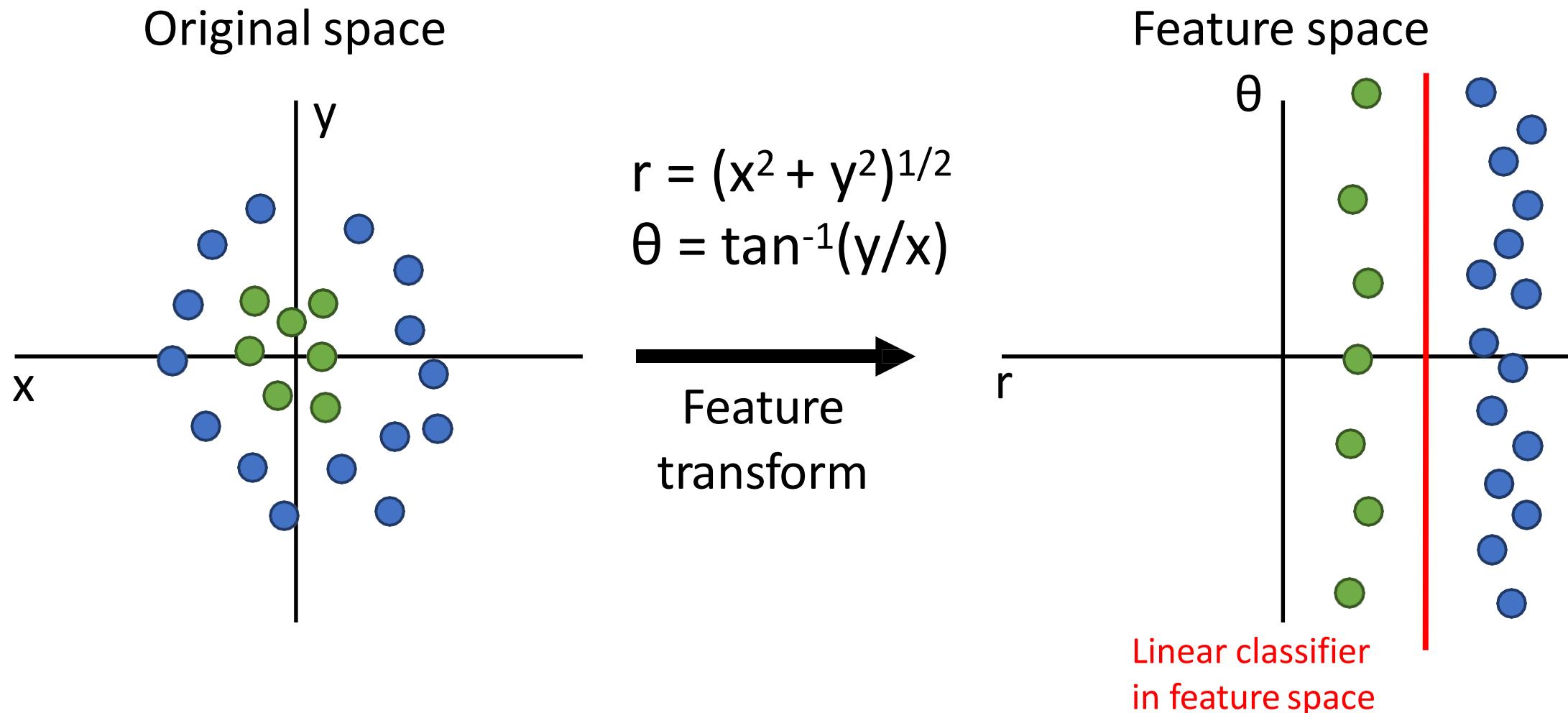
# One solution: Feature Transforms



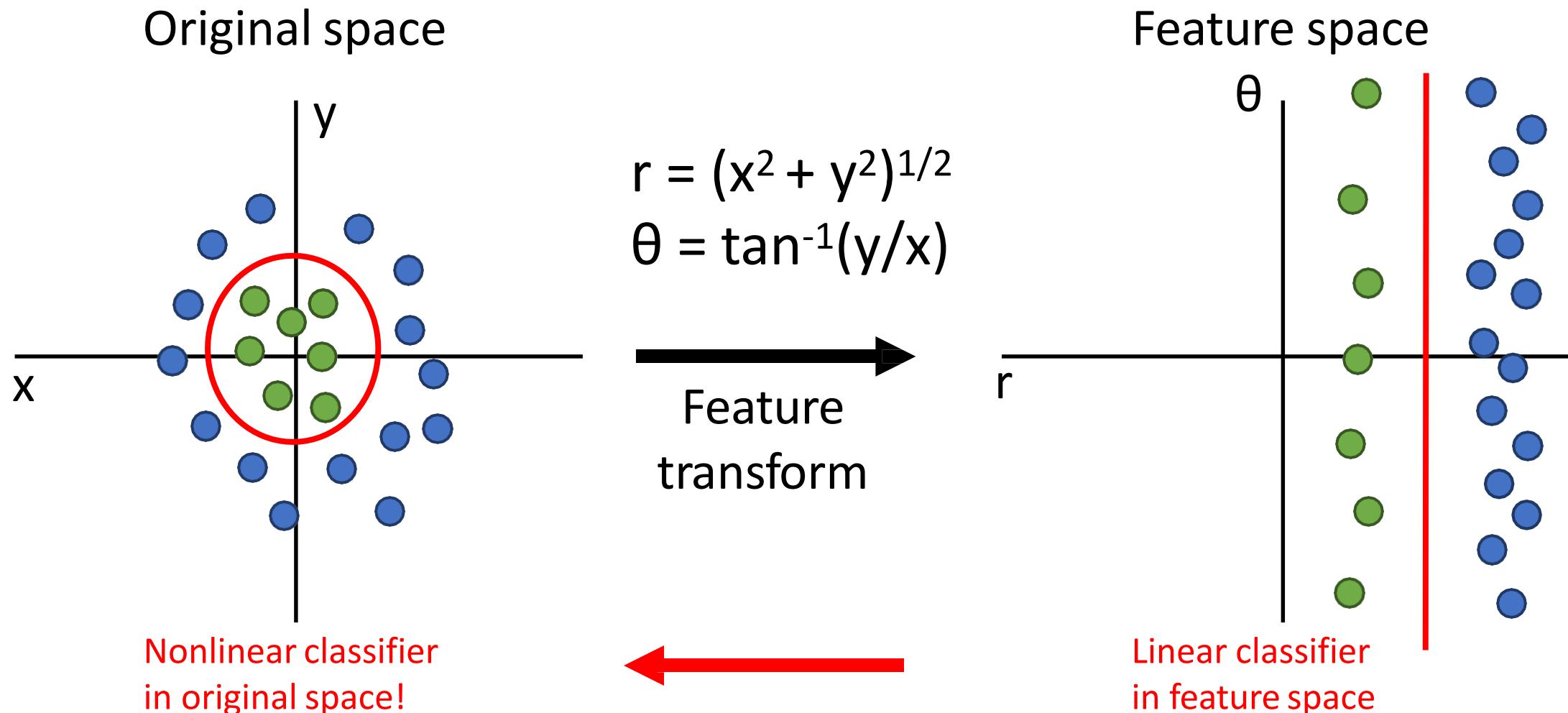
# One solution: Feature Transforms



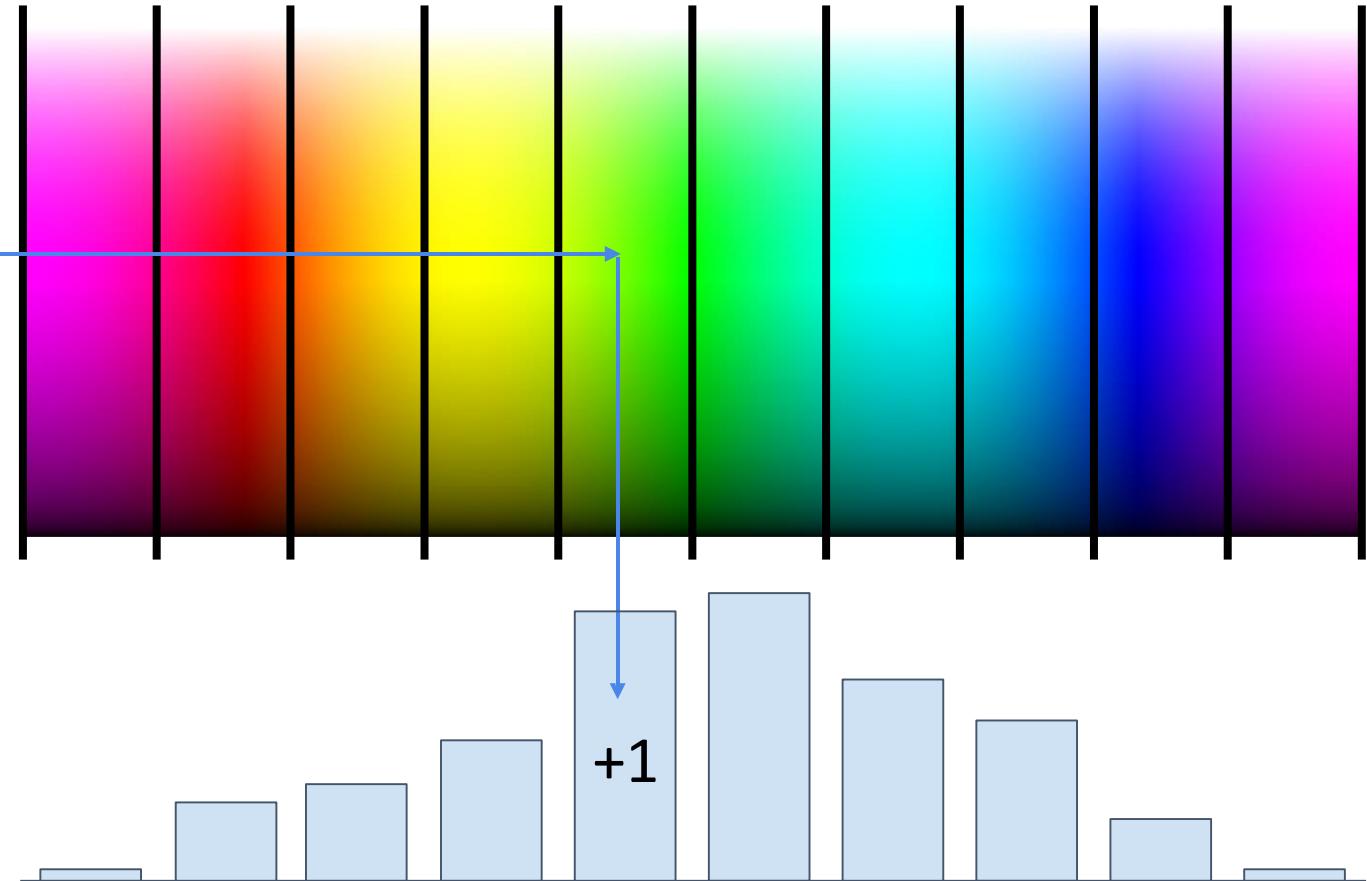
# One solution: Feature Transforms



# One solution: Feature Transforms



# Image Features: Color Histogram



Ignores texture,  
spatial positions

[Frog image](#) is in the public domain

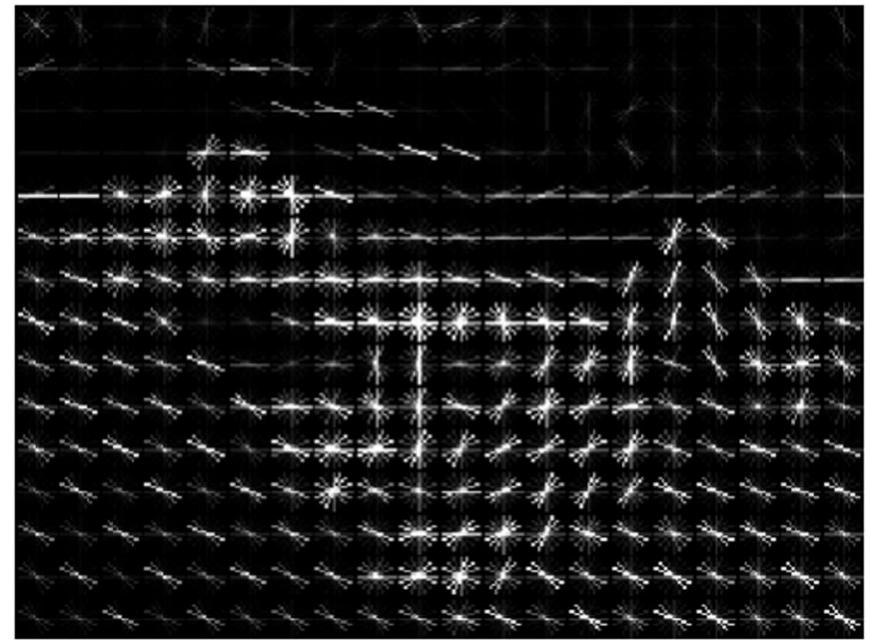
# Image Features: Histogram of Oriented Gradients (HoG)



1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

Lowe, "Object recognition from local scale-invariant features", ICCV 1999  
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

# Image Features: Histogram of Oriented Gradients (HoG)

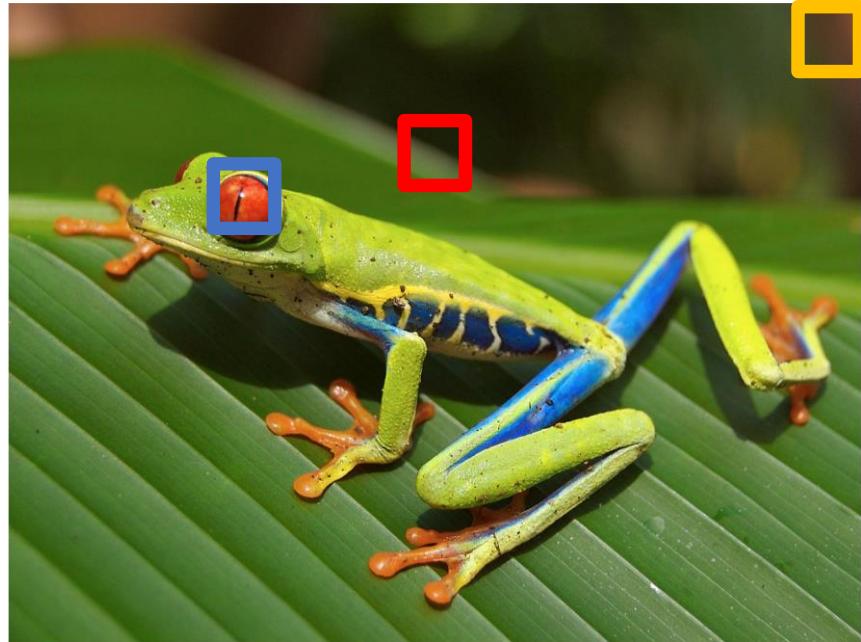


1. Compute edge direction / strength at each pixel
2. Divide image into  $8 \times 8$  regions
3. Within each region compute a histogram of edge directions weighted by edge strength

Example:  $320 \times 240$  image gets divided into  $40 \times 30$  bins; 8 directions per bin; feature vector has  $30 \times 40 \times 9 = 10,800$  numbers

Lowe, "Object recognition from local scale-invariant features", ICCV 1999  
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

# Image Features: Histogram of Oriented Gradients (HoG)

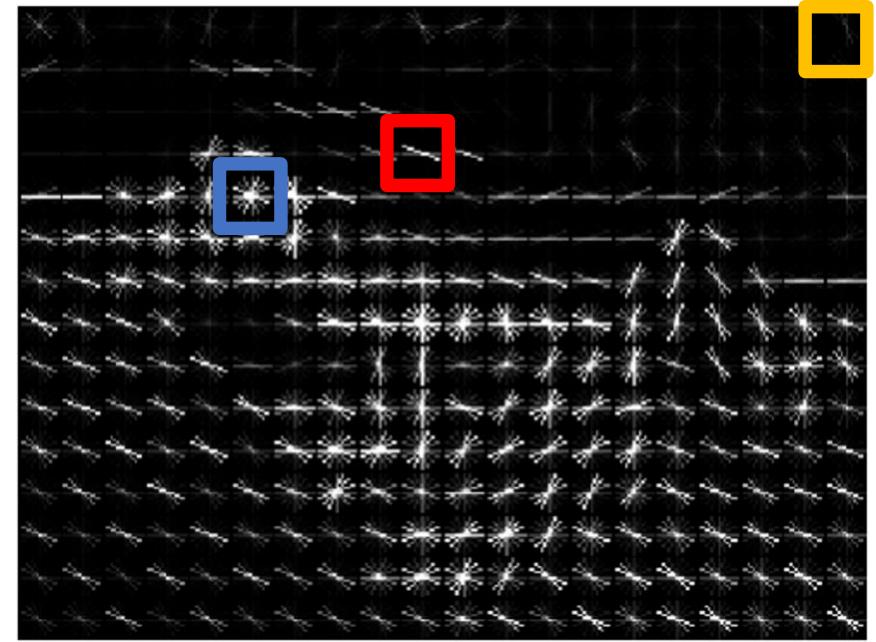


Weak edges

Strong diagonal edges



Edges in all directions

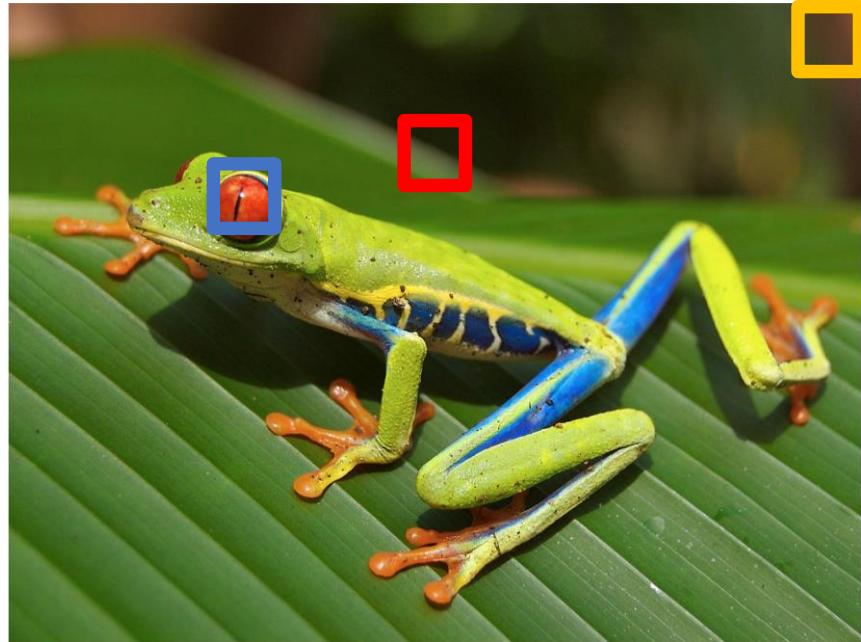


1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

Example: 320x240 image gets divided into 40x30 bins; 8 directions per bin; feature vector has  $30 \times 40 \times 9 = 10,800$  numbers

Lowe, "Object recognition from local scale-invariant features", ICCV 1999  
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

# Image Features: Histogram of Oriented Gradients (HoG)

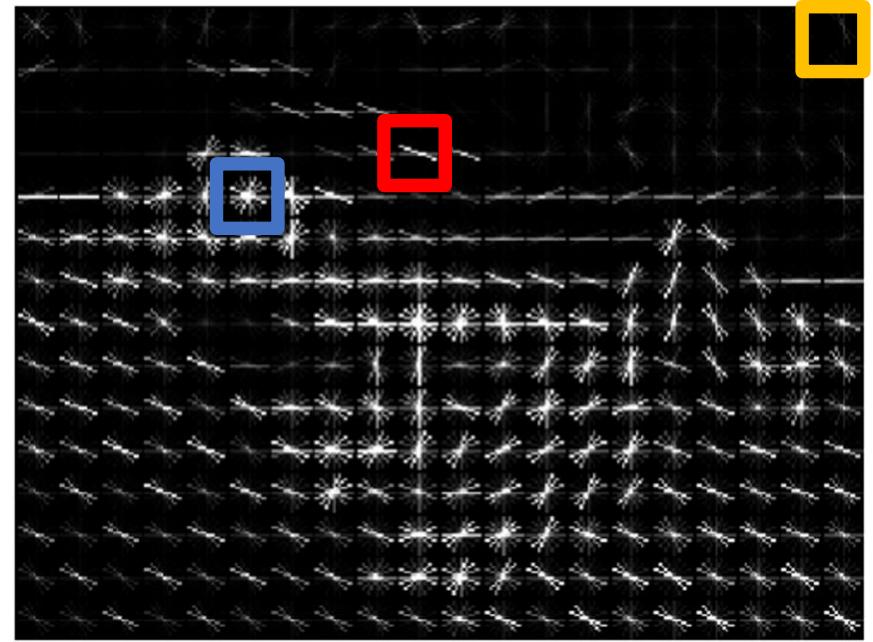


1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

Weak edges  
Strong diagonal edges

→  
Edges in all directions

Captures  
texture and  
position,  
robust to  
small image  
changes



Example: 320x240 image gets divided into 40x30 bins; 8 directions per bin; feature vector has  $30 \times 40 \times 9 = 10,800$  numbers

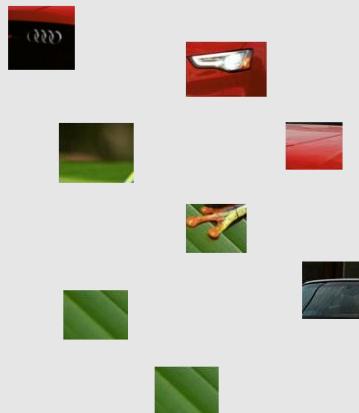
Lowe, "Object recognition from local scale-invariant features", ICCV 1999  
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

# Image Features: Bag of Words (Data-Driven!)

## Step 1: Build codebook



Extract random patches



Cluster patches to form “codebook” of “visual words”

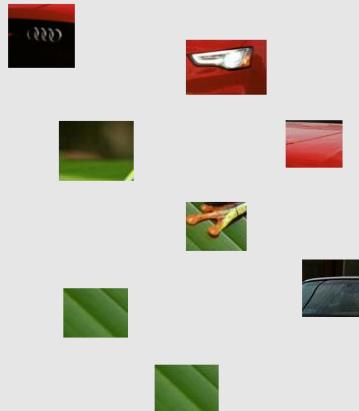


# Image Features: Bag of Words (Data-Driven!)

## Step 1: Build codebook



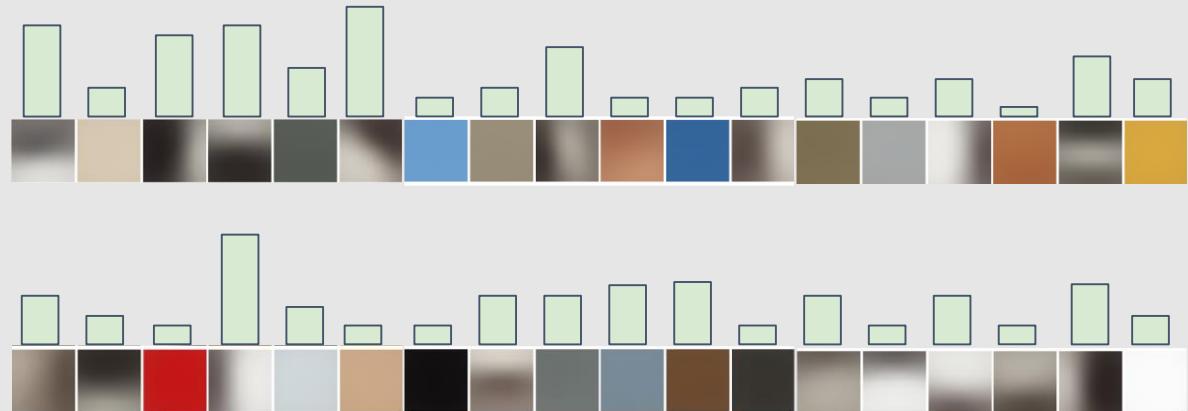
Extract random patches



Cluster patches to form “codebook” of “visual words”

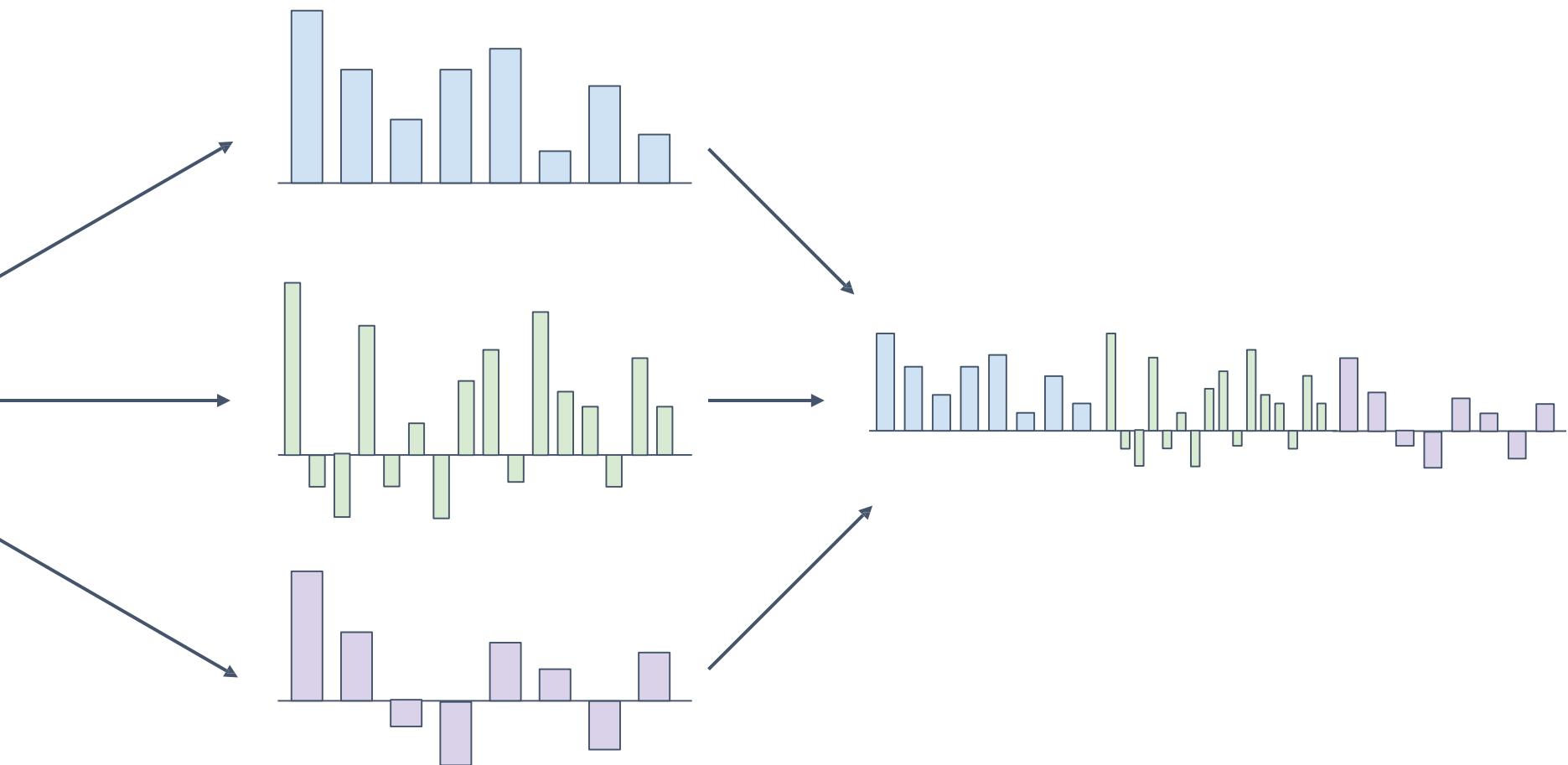


## Step 2: Encode images



Fei-Fei and Perona, “A bayesian hierarchical model for learning natural scene categories”, CVPR 2005

# Image Features



# Example: Winner of 2011 ImageNet challenge

Low-level feature extraction  $\approx$  10k patches per image

- SIFT: 128-dim
  - color: 96-dim
- }
- reduced to 64-dim with PCA

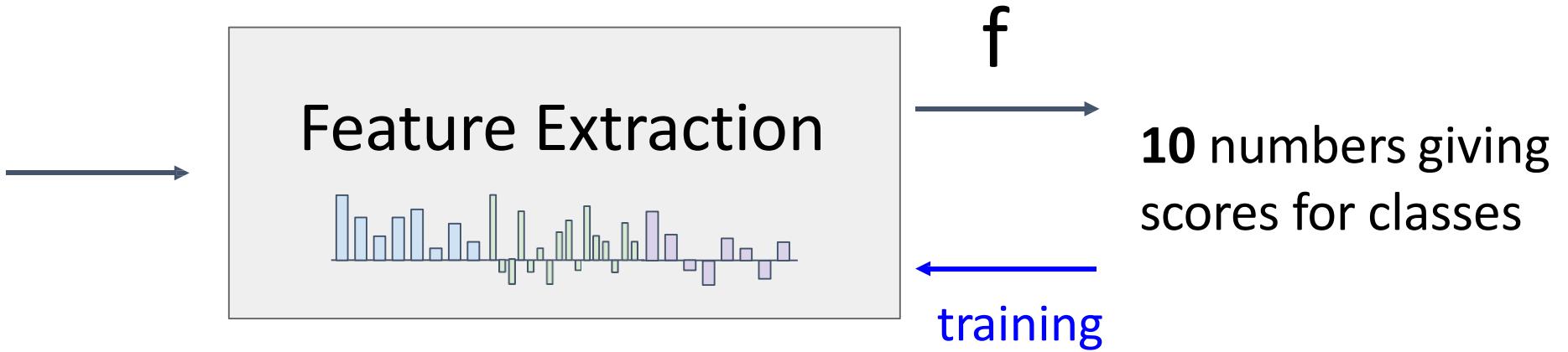
FV extraction and compression:

- $N=1,024$  Gaussians,  $R=4$  regions  $\Rightarrow 520K \text{ dim} \times 2$
- compression:  $G=8$ ,  $b=1$  bit per dimension

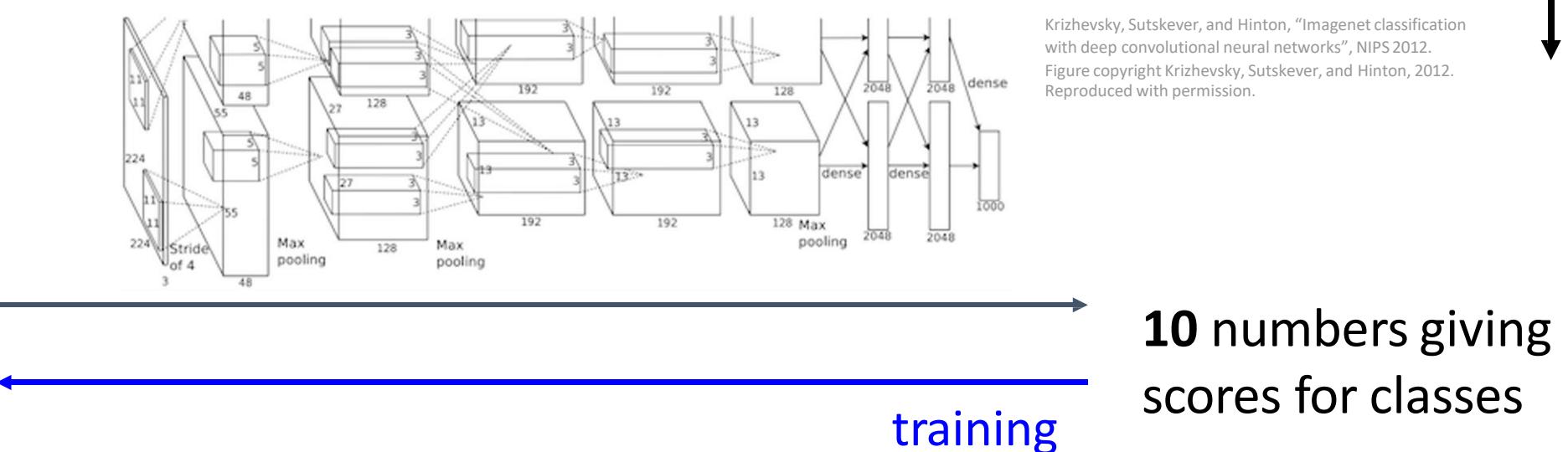
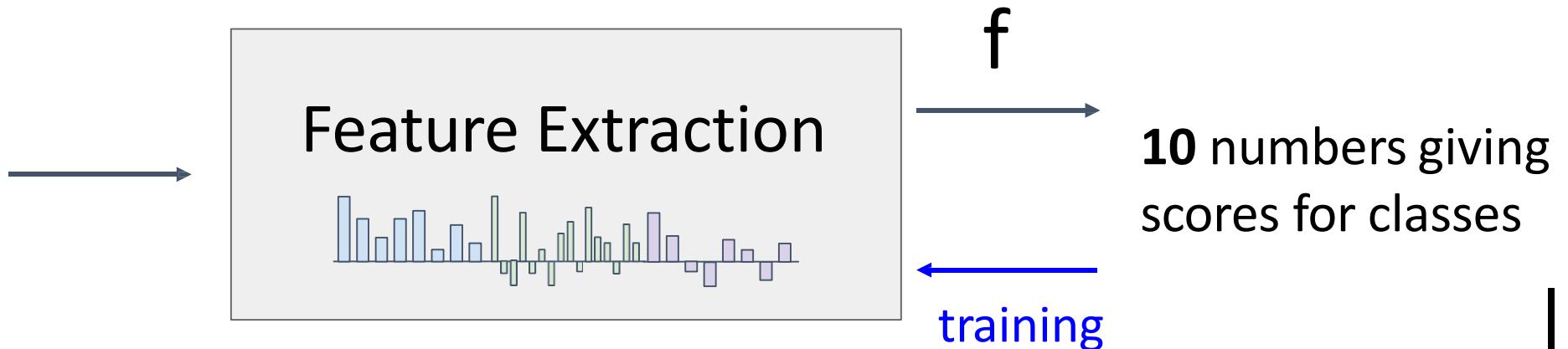
One-vs-all SVM learning with SGD

Late fusion of SIFT and color systems

# Image Features



# Image Features vs Neural Networks



# Neural Networks

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

**Before:** Linear Classifier:  $f(x) = Wx + b$

Learnable parameters:  $W \in \mathbb{R}^{C \times D}, b \in \mathbb{R}^C$

# Neural Networks

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

**Before:** Linear Classifier:  $f(x) = Wx + b$

Learnable parameters:  $W \in \mathbb{R}^{C \times D}, b \in \mathbb{R}^C$

**Now:** Two-Layer Neural Network:  $f(x) = W_2 \max(0, W_1 x + b_1) + b_2$

# Neural Networks

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

**Before:** Linear Classifier:  $f(x) = Wx + b$

Learnable parameters:  $W \in \mathbb{R}^{C \times D}, b \in \mathbb{R}^C$

**Now:** Two-Layer Neural Network:  $f(x) = W_2 \max(0, W_1 x + b_1) + b_2$

Learnable parameters:  $W_1 \in \mathbb{R}^{H \times D}, b_1 \in \mathbb{R}^H, W_2 \in \mathbb{R}^{C \times H}, b_2 \in \mathbb{R}^C$

# Neural Networks

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

**Before:** Linear Classifier:  $f(x) = Wx + b$

Learnable parameters:  $W \in \mathbb{R}^{C \times D}, b \in \mathbb{R}^C$

**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:

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

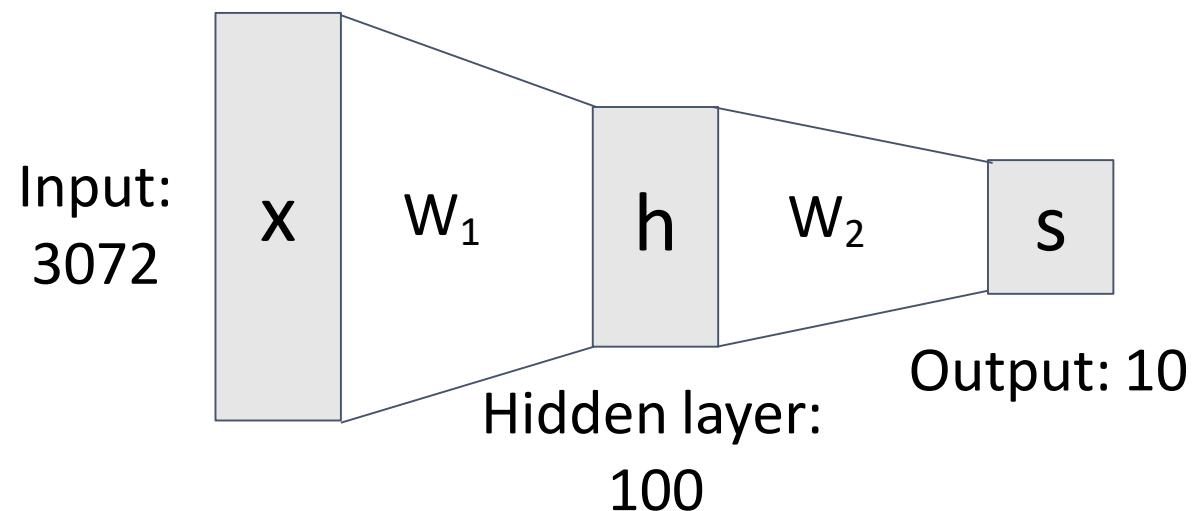
# Neural Networks

**Before:** Linear classifier

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

**Now:** 2-layer Neural Network

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



# Neural Networks

**Before:** Linear classifier

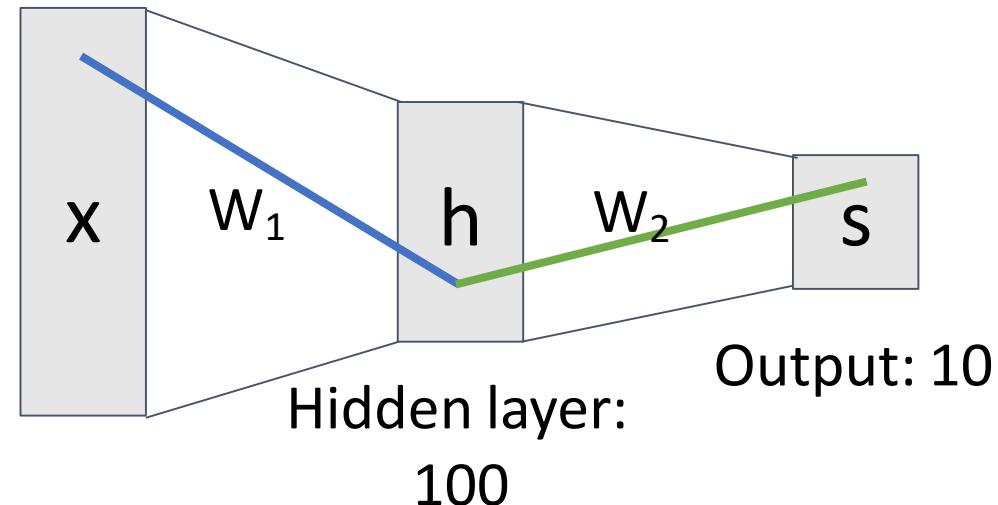
$$f(x) = Wx + b$$

**Now:** 2-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  $x_j$

Input:  
3072



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

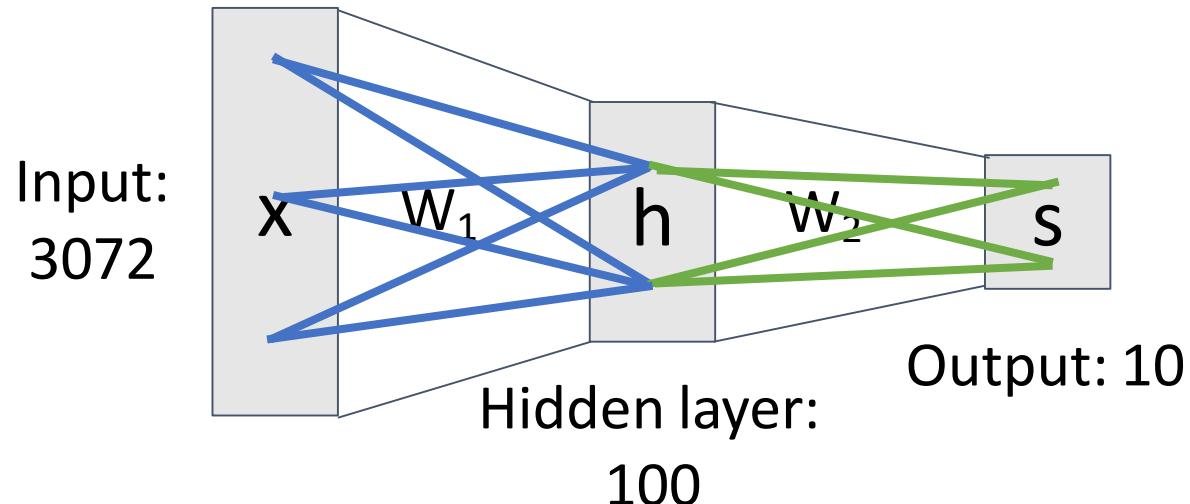
**Before:** Linear classifier

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

**Now:** 2-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  $x_j$



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

**(Before) Linear score function:**

Linear classifier: One template per class



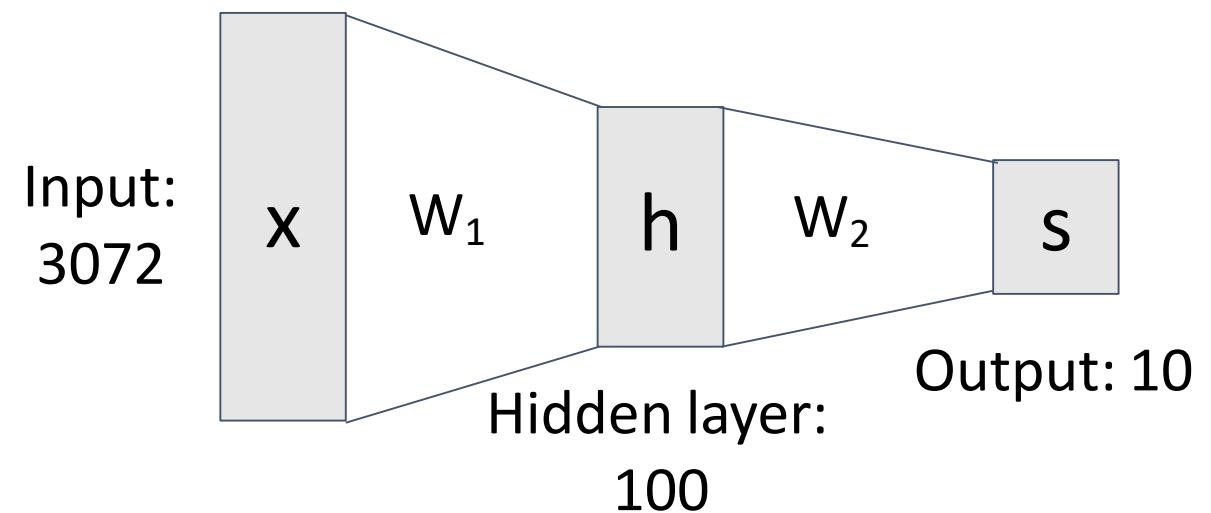
# Neural Networks

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



**(Before)** Linear score function:

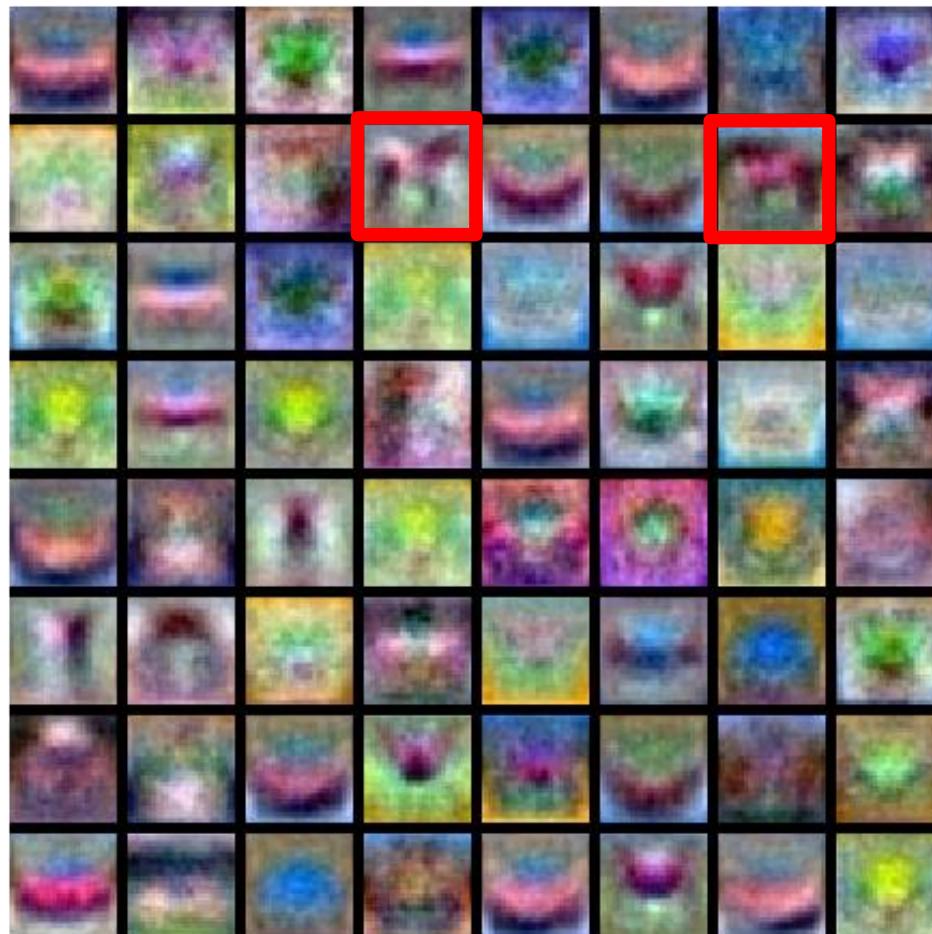
**(Now)** 2-layer Neural Network



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

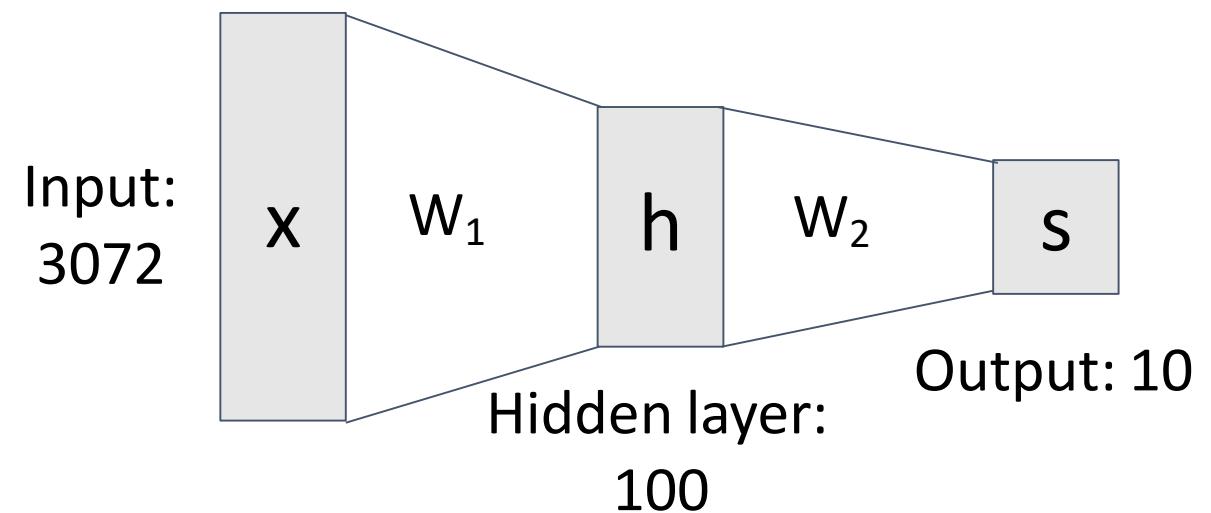
# Neural Networks

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



**(Before)** Linear score function:

**(Now)** 2-layer Neural Network



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

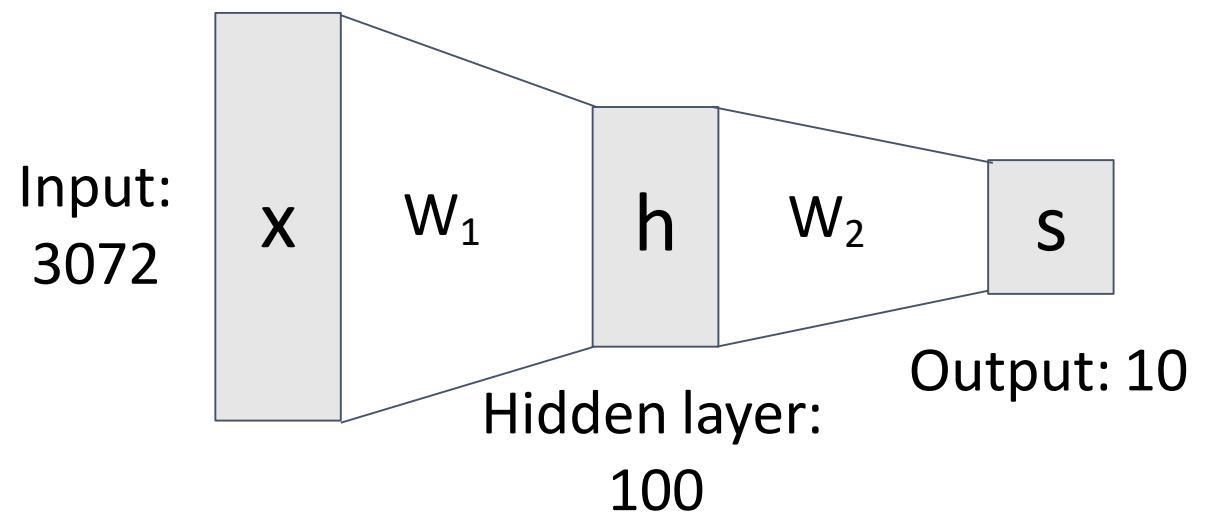
# Neural Networks

“Distributed representation”:  
Most templates not interpretable!



**(Before)** Linear score function:

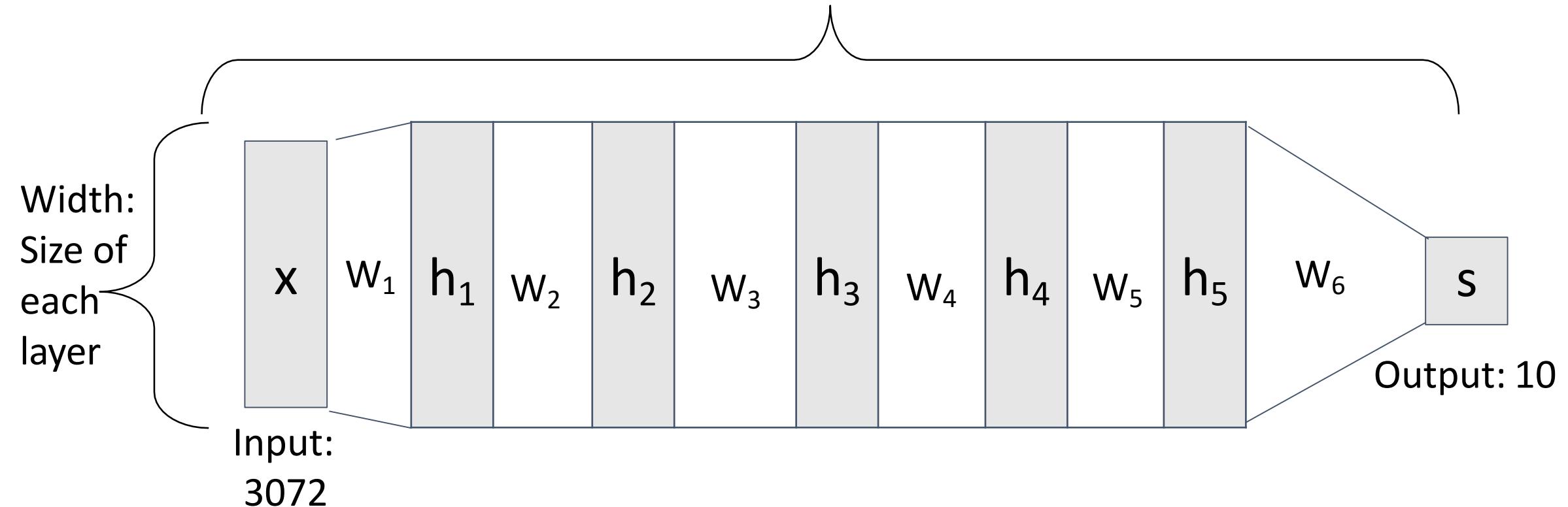
**(Now)** 2-layer Neural Network



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

# Deep Neural Networks

Depth = number of layers



$$f(x) = W_6 \max\left(0, W_5 \max\left(0, W_4 \max\left(0, W_3 \max\left(0, W_2 \max(0, W_1 x)\right)\right)\right)\right)$$

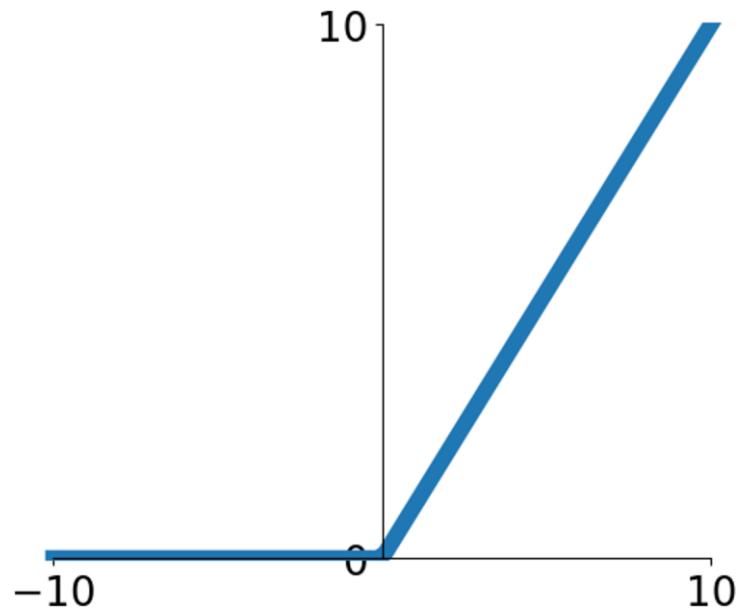
# Activation Functions

## 2-layer Neural Network

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

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

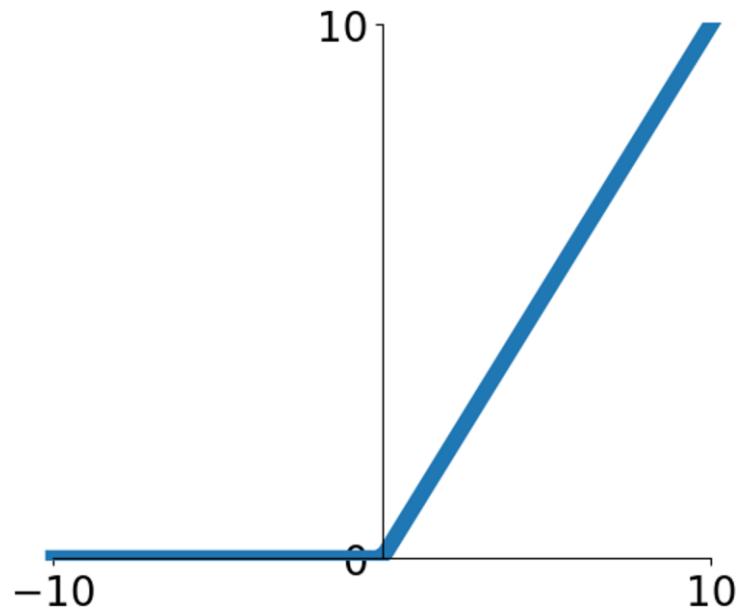
This is called the **activation function** of  
the neural network



# Activation Functions

## 2-layer Neural Network

The function  $ReLU(z) = \max(0, z)$  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

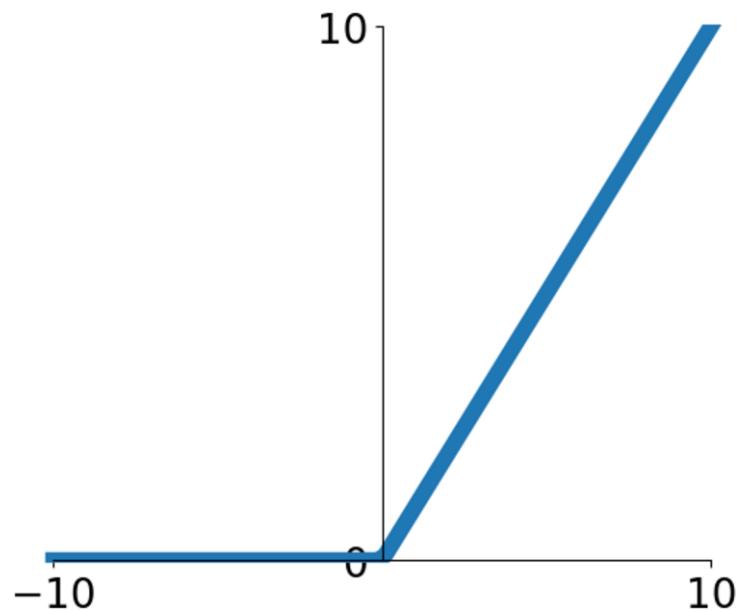
**Q:** What happens if we build a neural network with no activation function?

$$\begin{aligned} f(x) &= W_2(W_1 x + b_1) + b_2 \\ &= W_2 W_1 x + W_2 b_1 + b_2 \end{aligned}$$

# Activation Functions

## 2-layer Neural Network

The function  $ReLU(z) = \max(0, z)$  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?

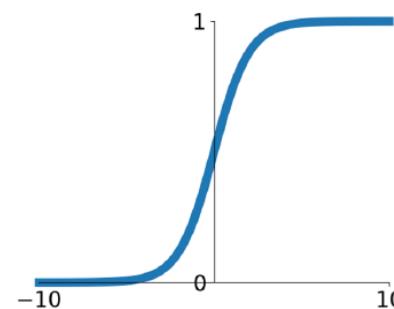
$$\begin{aligned} f(x) &= W_2(W_1 x + b_1) + b_2 \\ &= W_2 W_1 x + W_2 b_1 + b_2 \end{aligned}$$

**A:** We end up with a linear classifier!

# Activation Functions

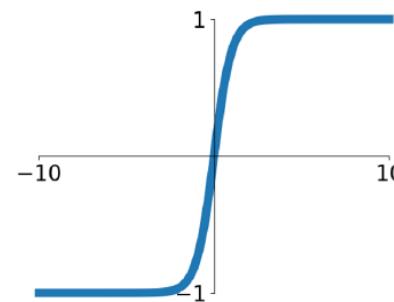
## Sigmoid

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



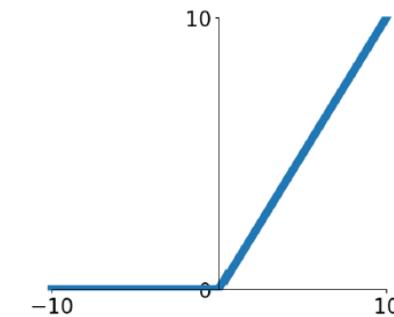
## tanh

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



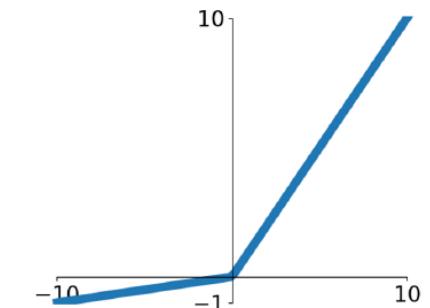
## ReLU

$$\max(0, x)$$



## Leaky ReLU

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

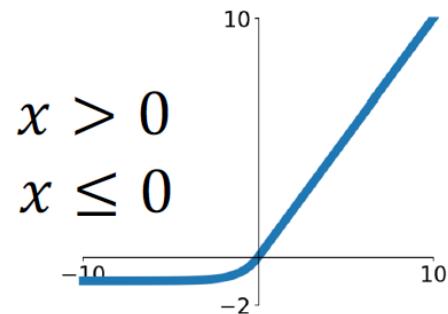


## Softplus

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

## ELU

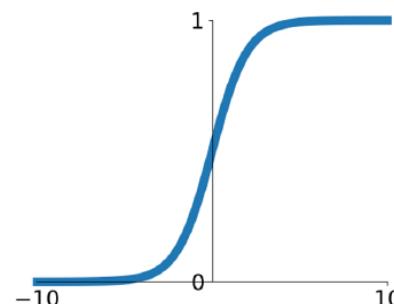
$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \leq 0 \end{cases}$$



# Activation Functions

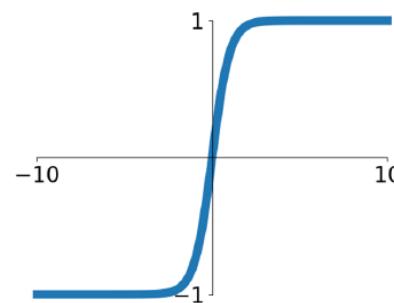
## Sigmoid

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



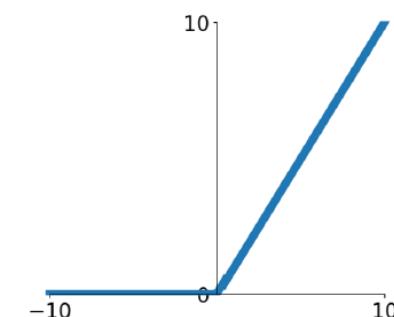
## tanh

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



## ReLU

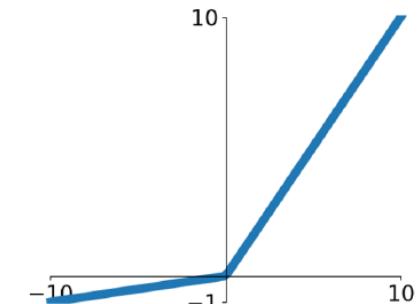
$$\max(0, x)$$



ReLU is a good default choice  
for most problems

## Leaky ReLU

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

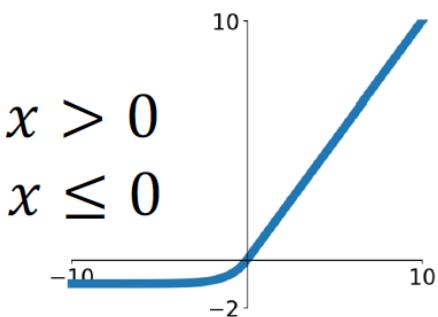


## Softplus

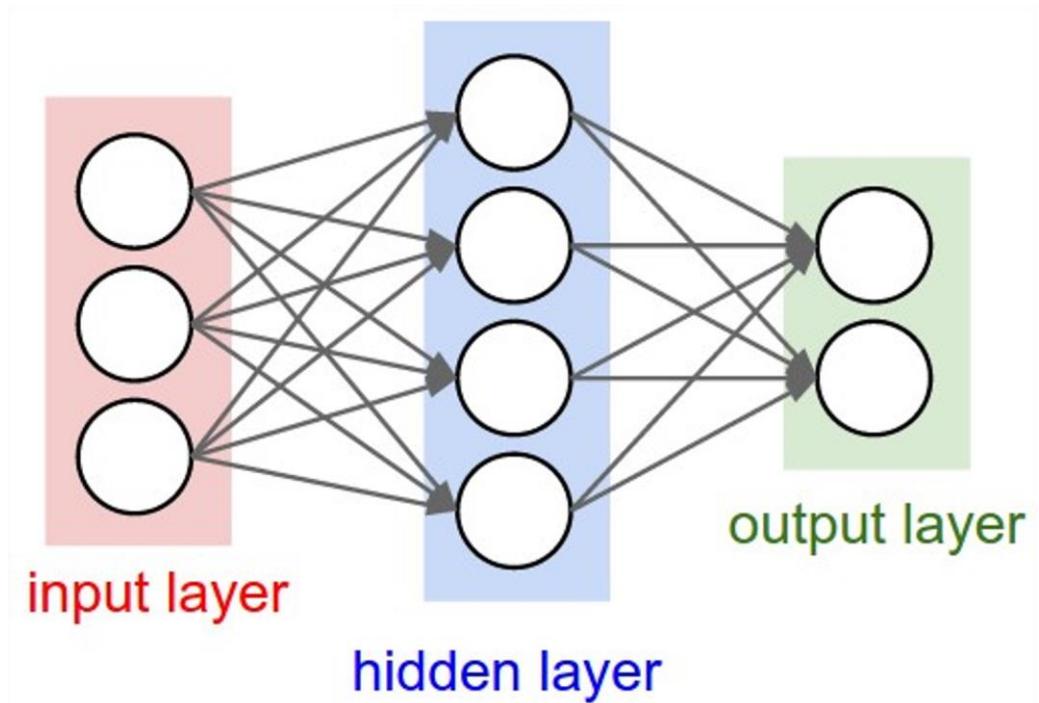
$$\log(1 + \exp(x))$$

## ELU

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

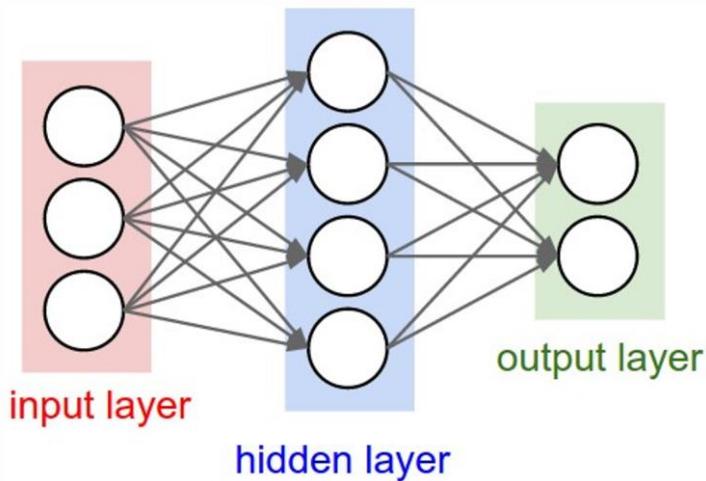


# Neural Net in <20 lines!



```
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
```

# Neural Net in <20 lines!

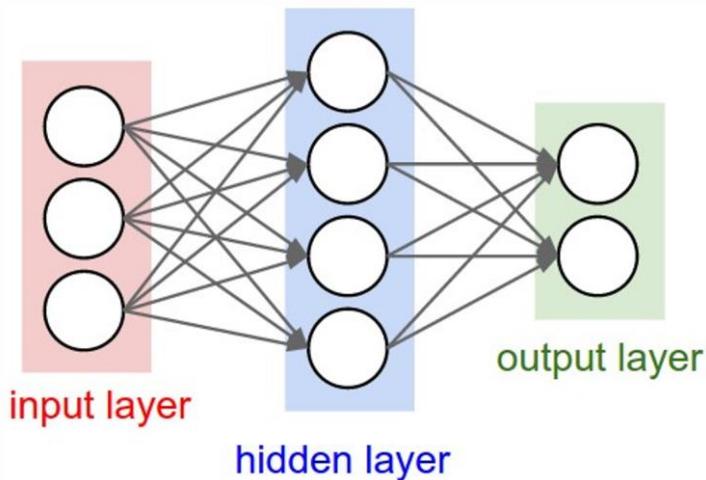


Initialize weights  
and data



```
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
```

# Neural Net in <20 lines!

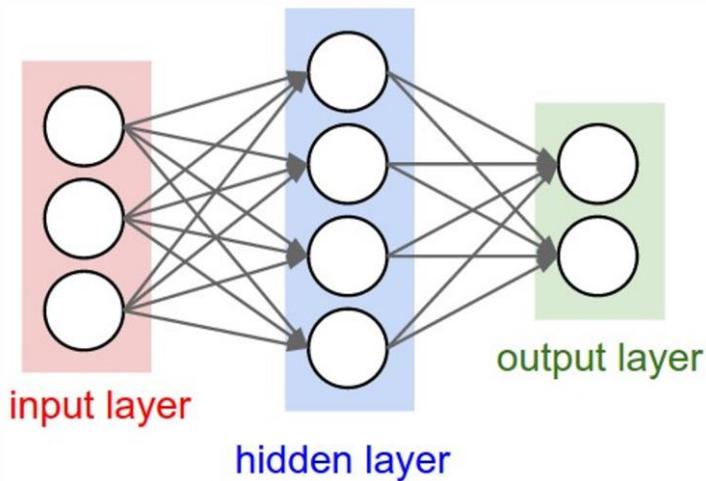


Initialize weights  
and data

Compute loss  
(sigmoid activation,  
L2 loss)

```
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
```

# Neural Net in <20 lines!



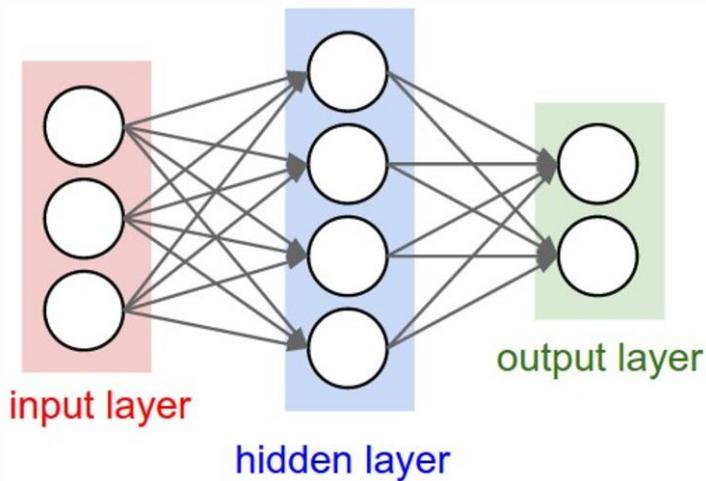
Initialize weights  
and data

Compute loss  
(sigmoid activation,  
L2 loss)

Compute  
gradients

```
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
```

# Neural Net in <20 lines!



Initialize weights  
and data

Compute loss  
(sigmoid activation,  
L2 loss)

Compute  
gradients

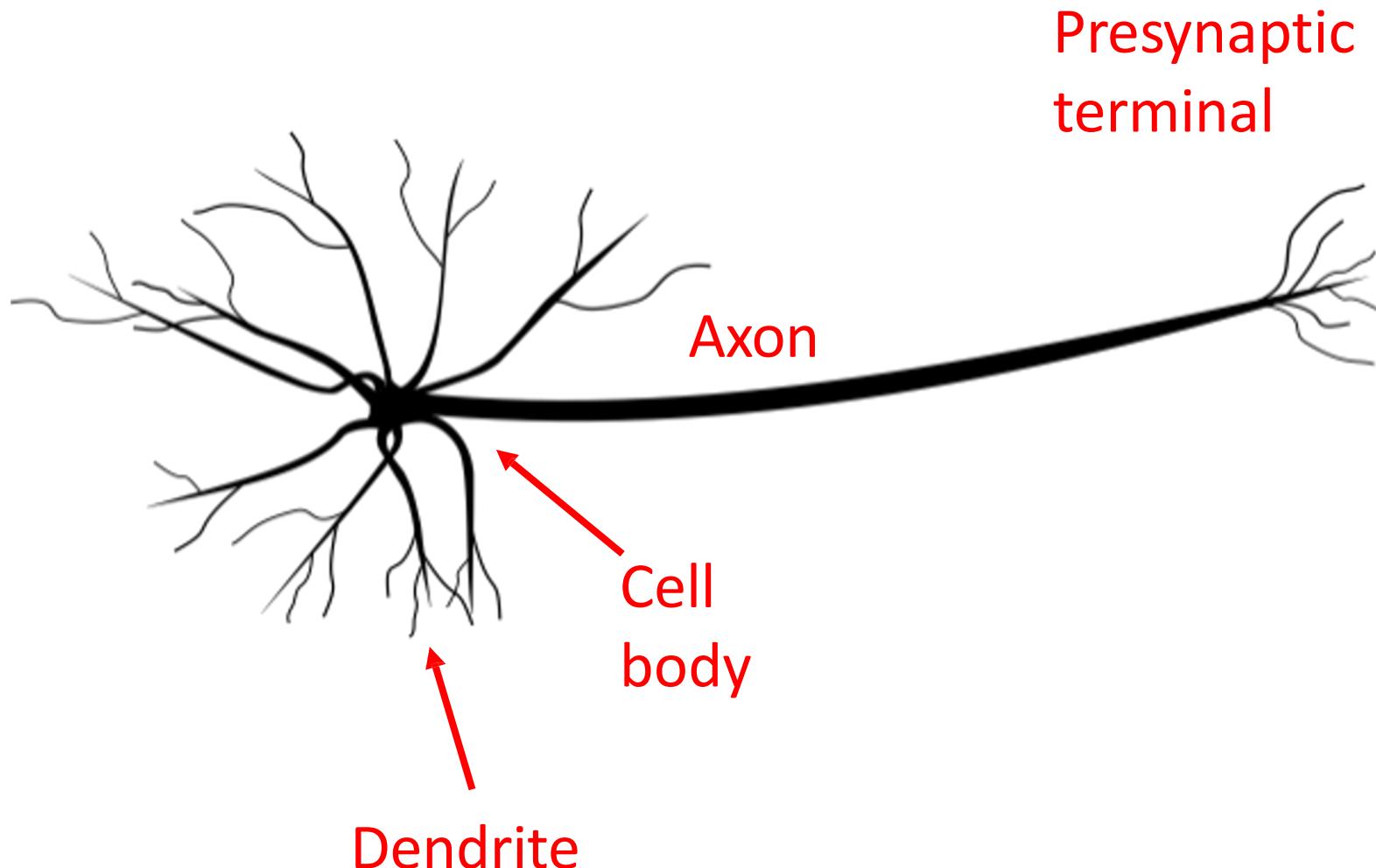
SGD  
step

```
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
```



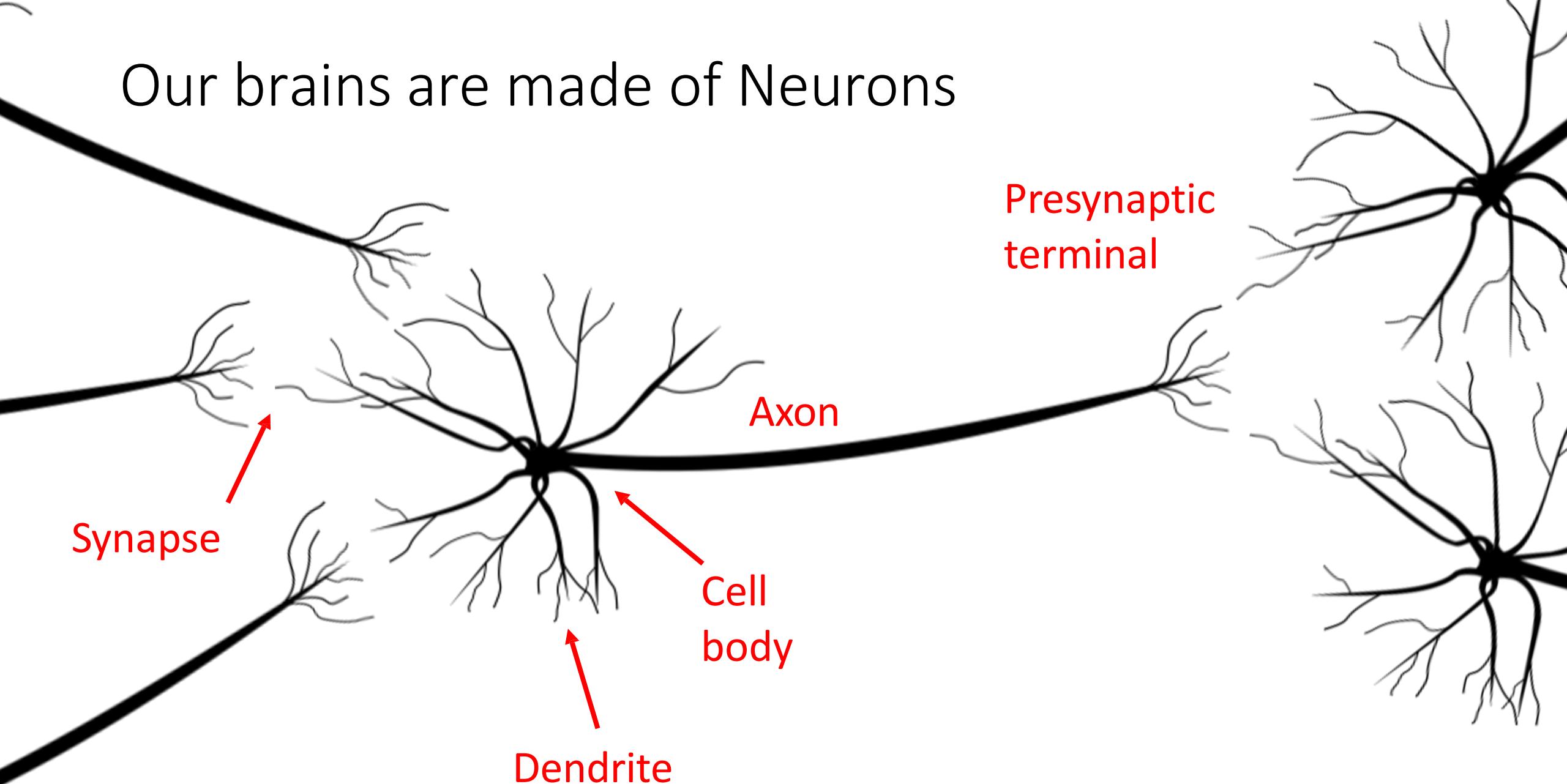
This image by [Fotis Bobolas](#) is  
licensed under [CC-BY 2.0](#)

# Our brains are made of Neurons

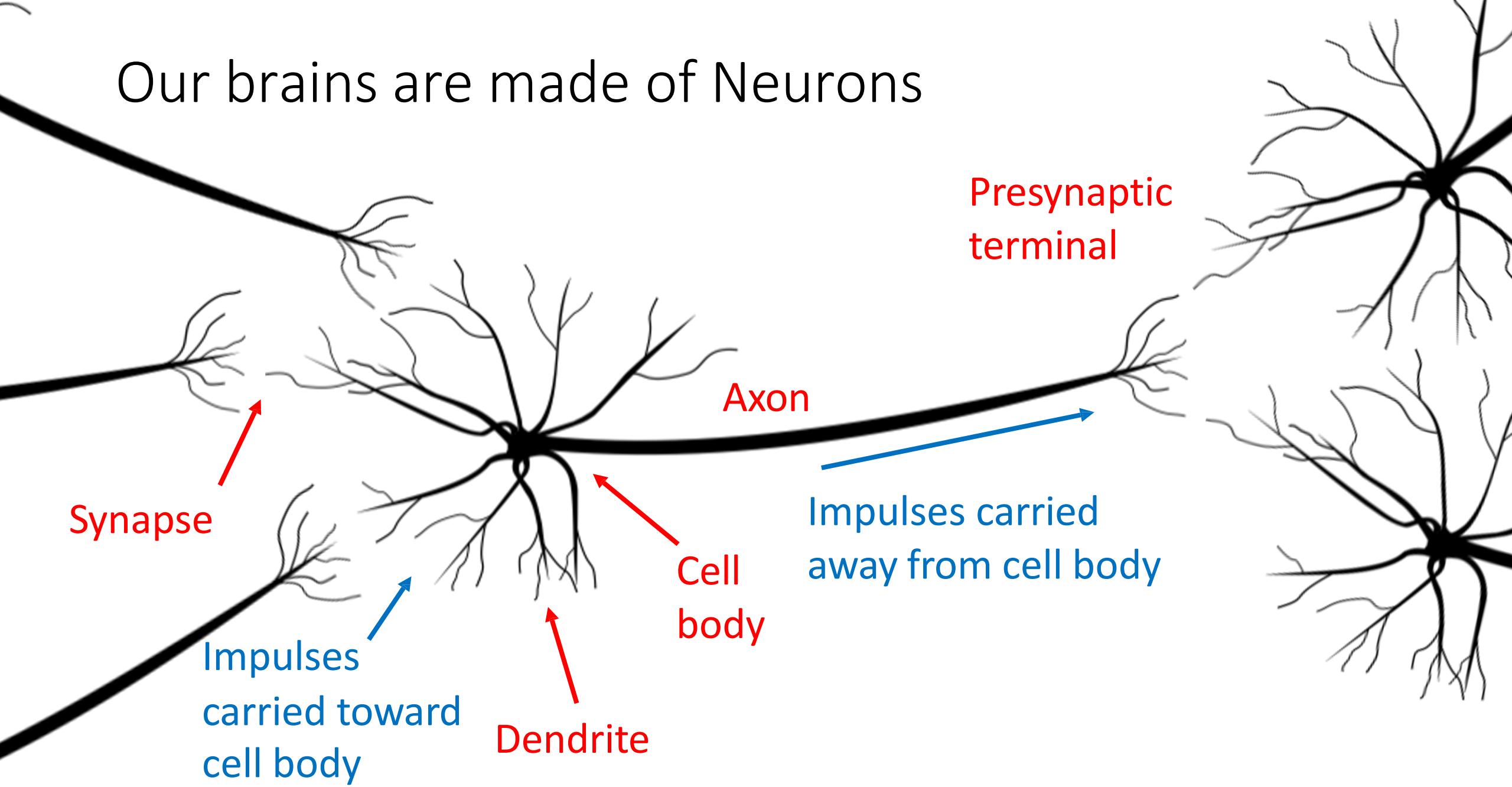


[Neuron image](#) by Felipe Perucho  
is licensed under [CC-BY 3.0](#)

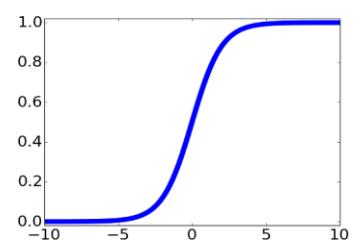
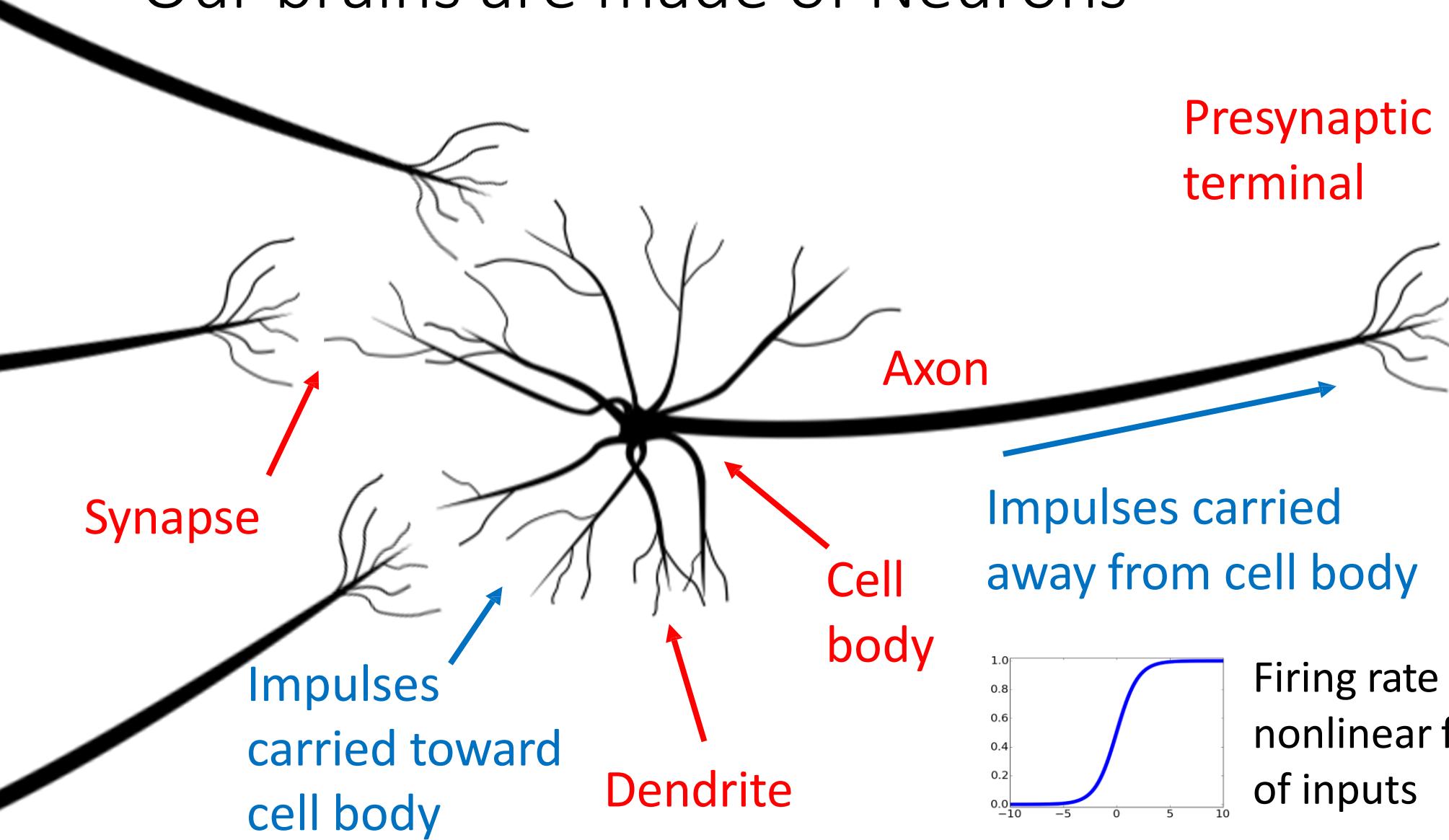
# Our brains are made of Neurons



# Our brains are made of Neurons

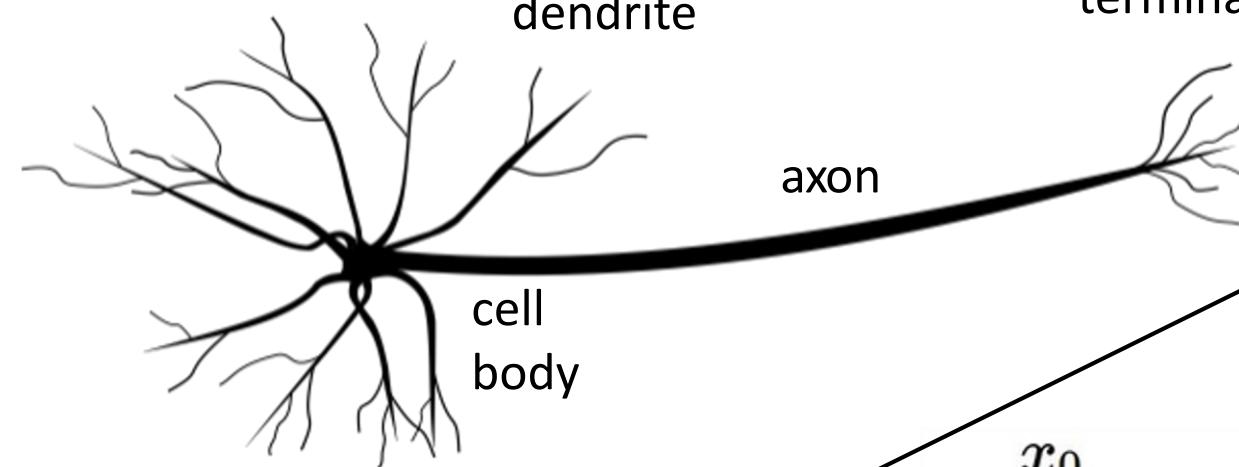


# Our brains are made of Neurons

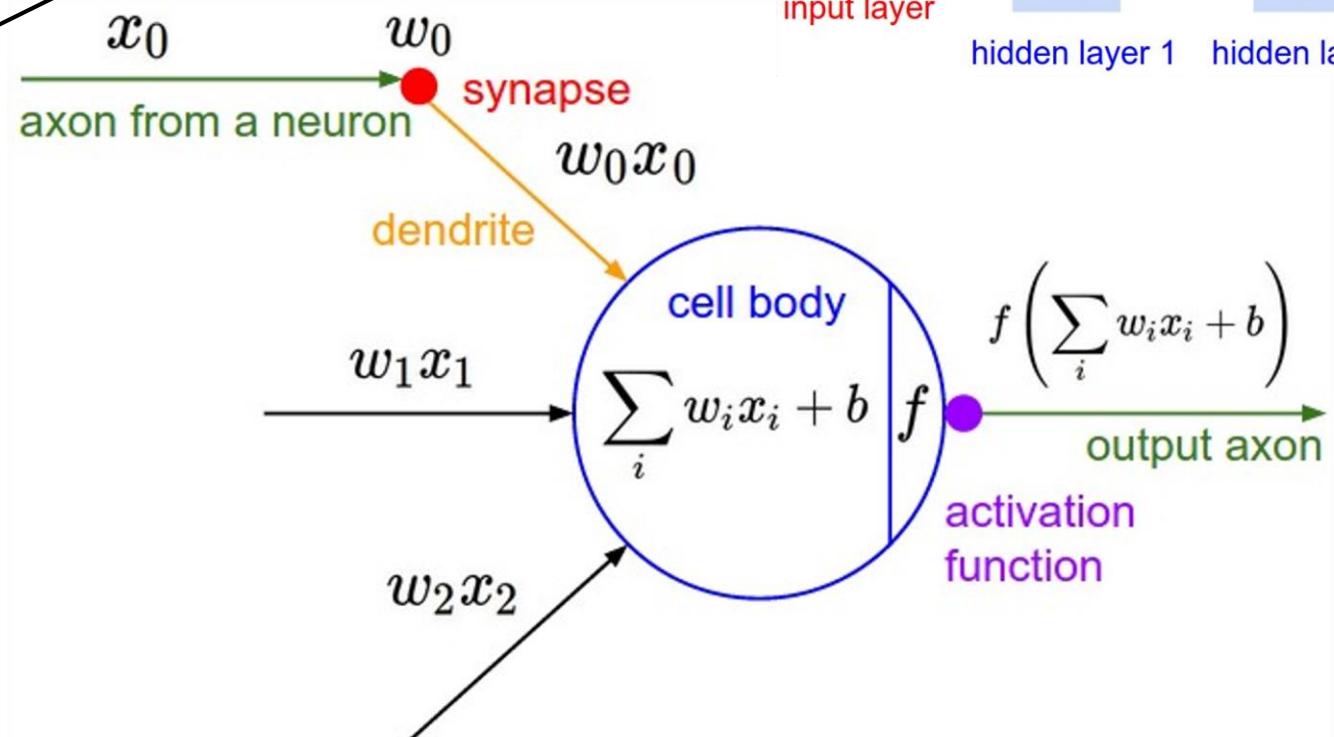
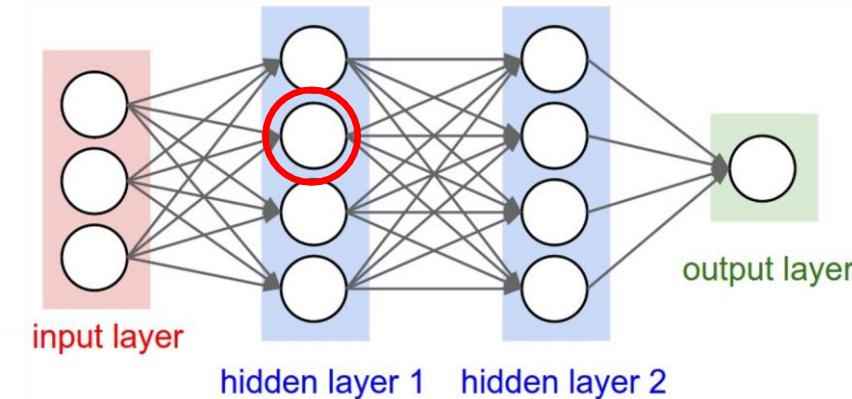


Firing rate is a  
nonlinear function  
of inputs

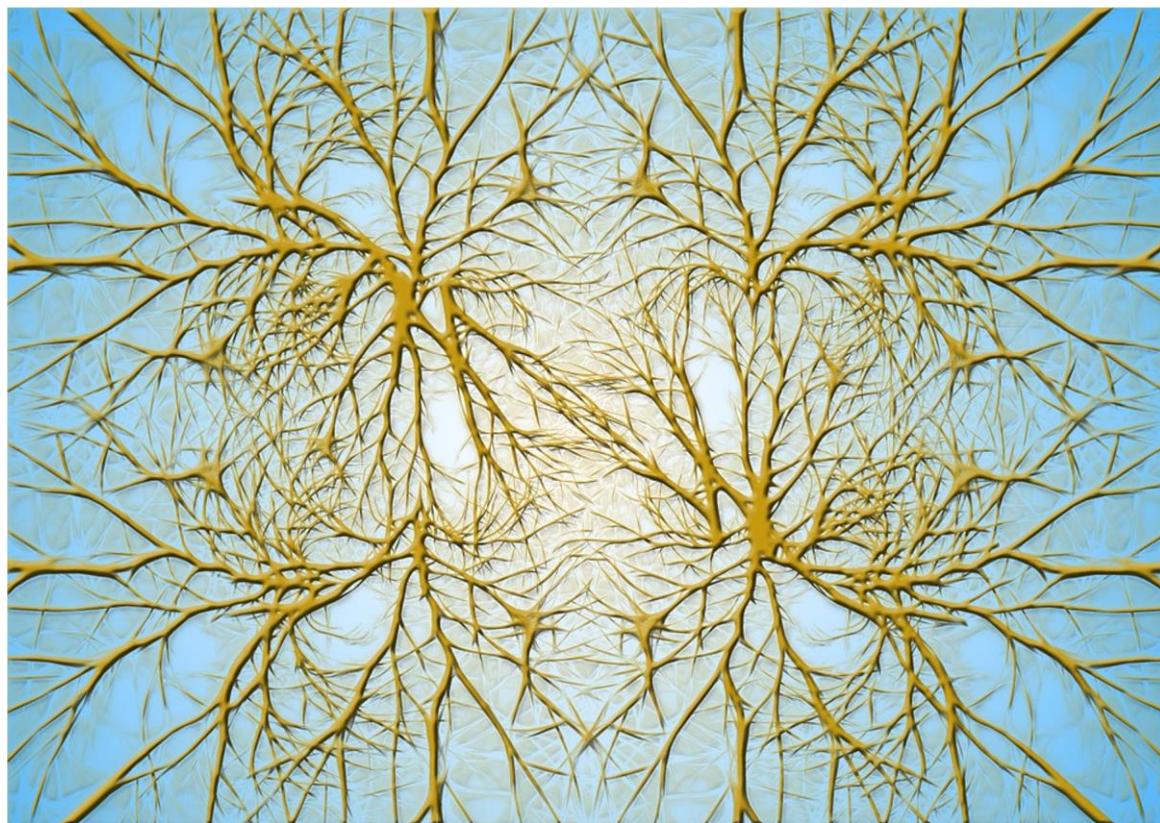
# Biological Neuron



# Artificial Neuron

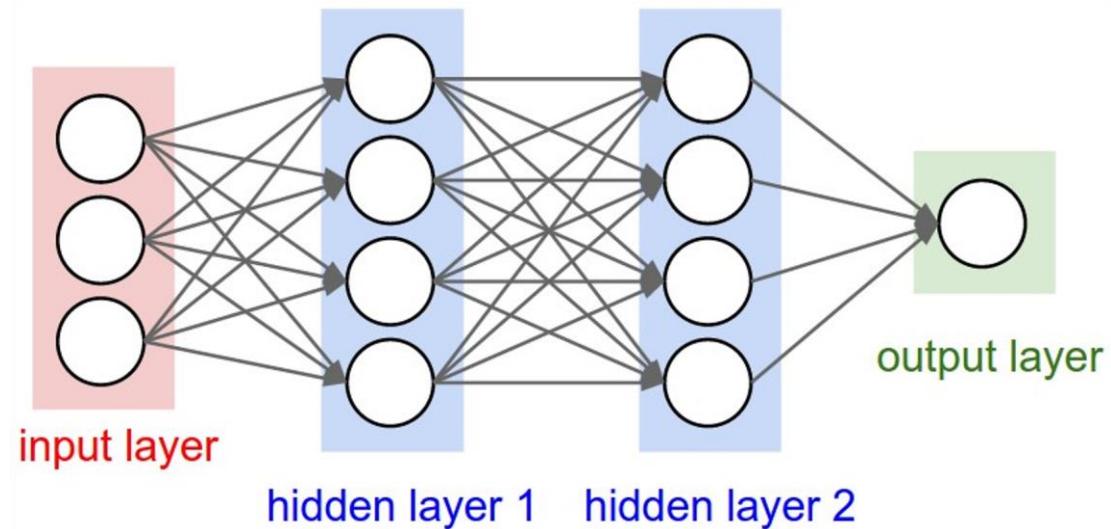


## Biological Neurons: Complex connectivity patterns

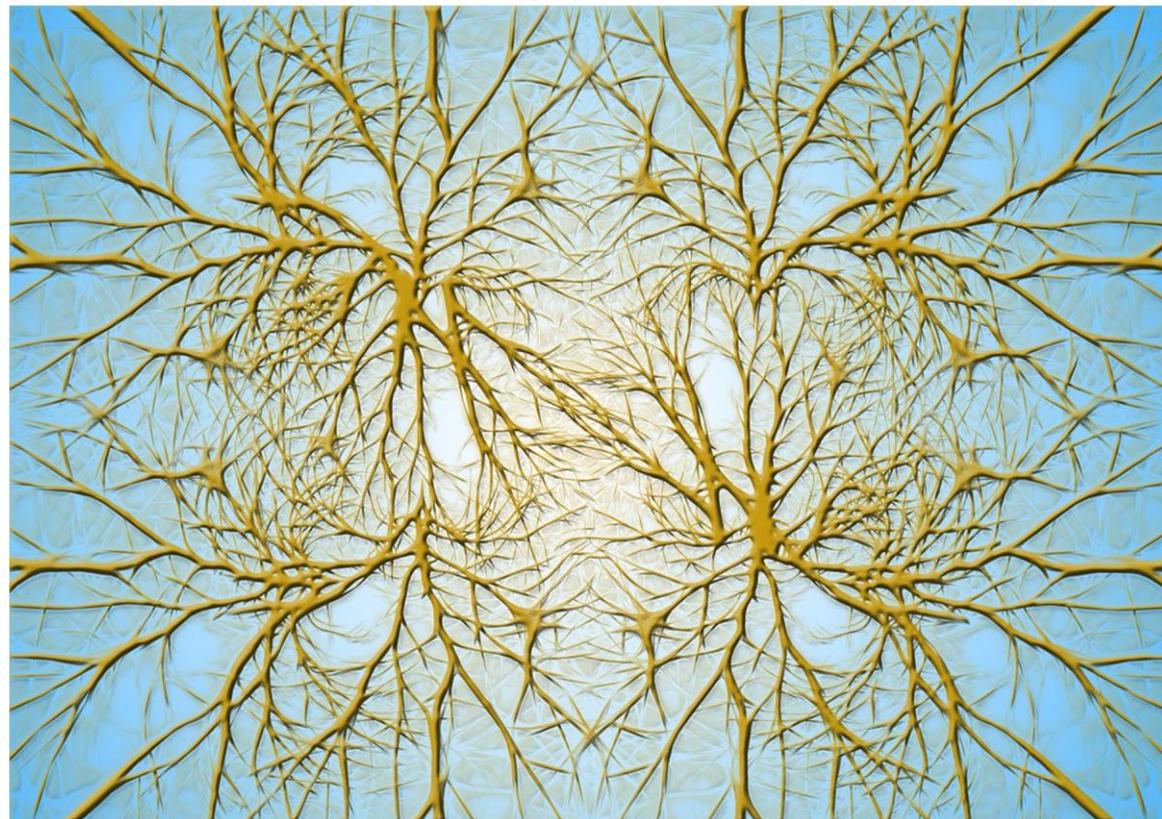


[This image is CC0 Public Domain](#)

Neurons in a neural network:  
Organized into regular layers for  
computational efficiency

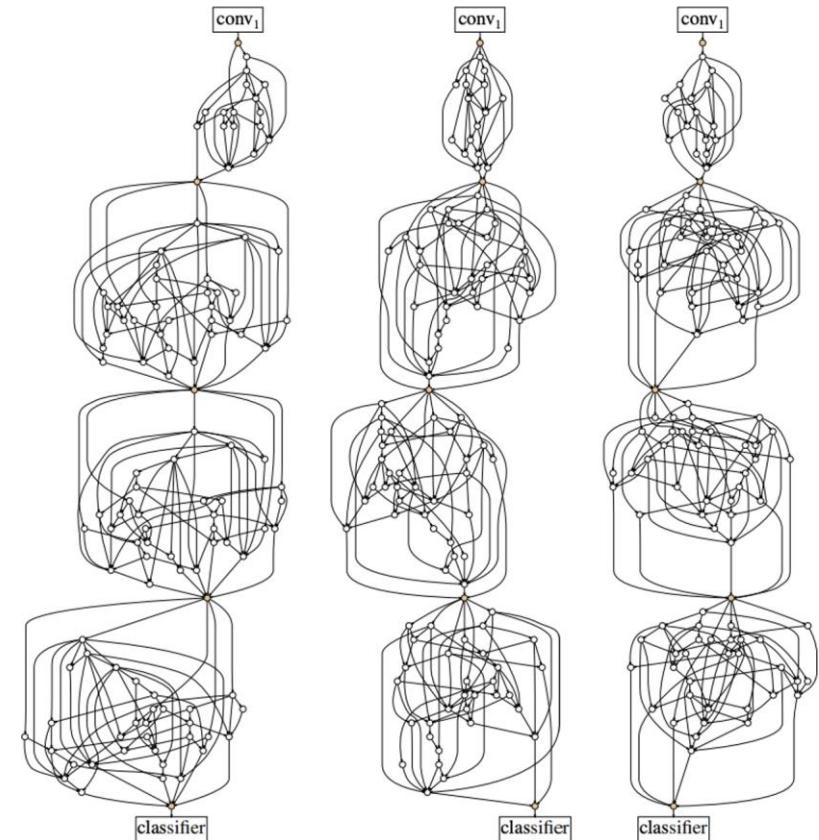


# Biological Neurons: Complex connectivity patterns



[This image is CC0 Public Domain](#)

But neural networks with random connections can work too!



Xie et al, "Exploring Randomly Wired Neural Networks for Image Recognition", ICCV 2019

# Be very careful with brain analogies!

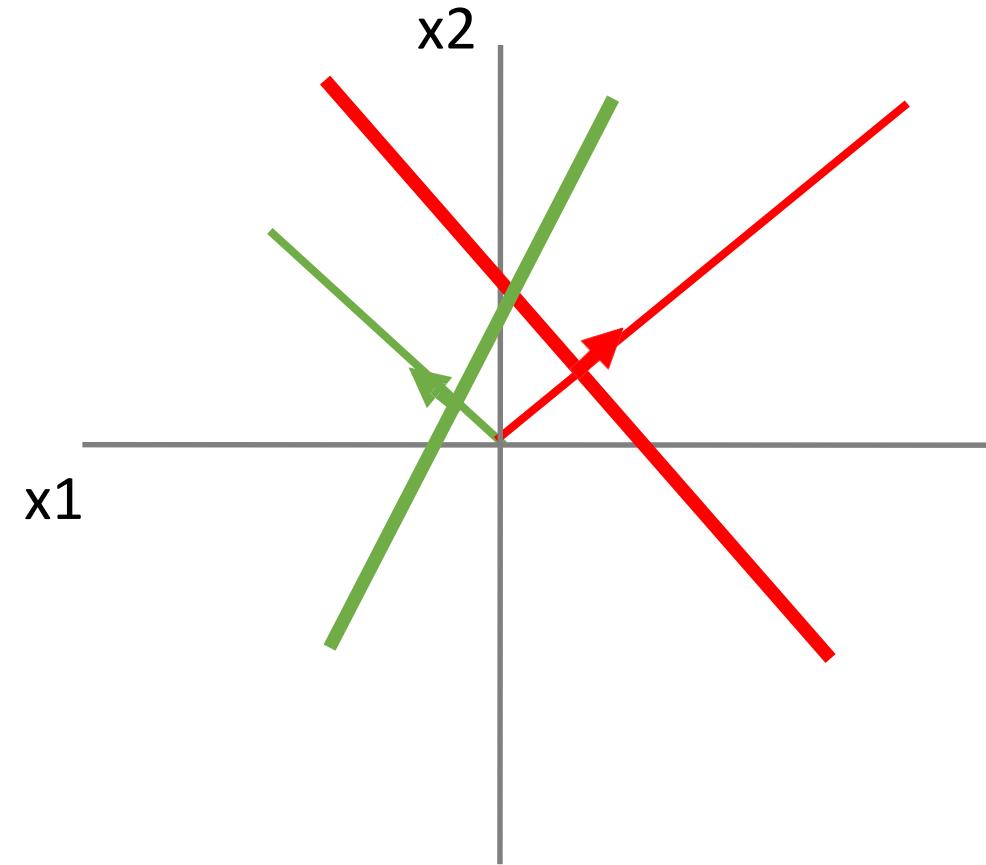
## **Biological Neurons:**

- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system
- Rate code may not be adequate

[Dendritic Computation. London and Häusser]

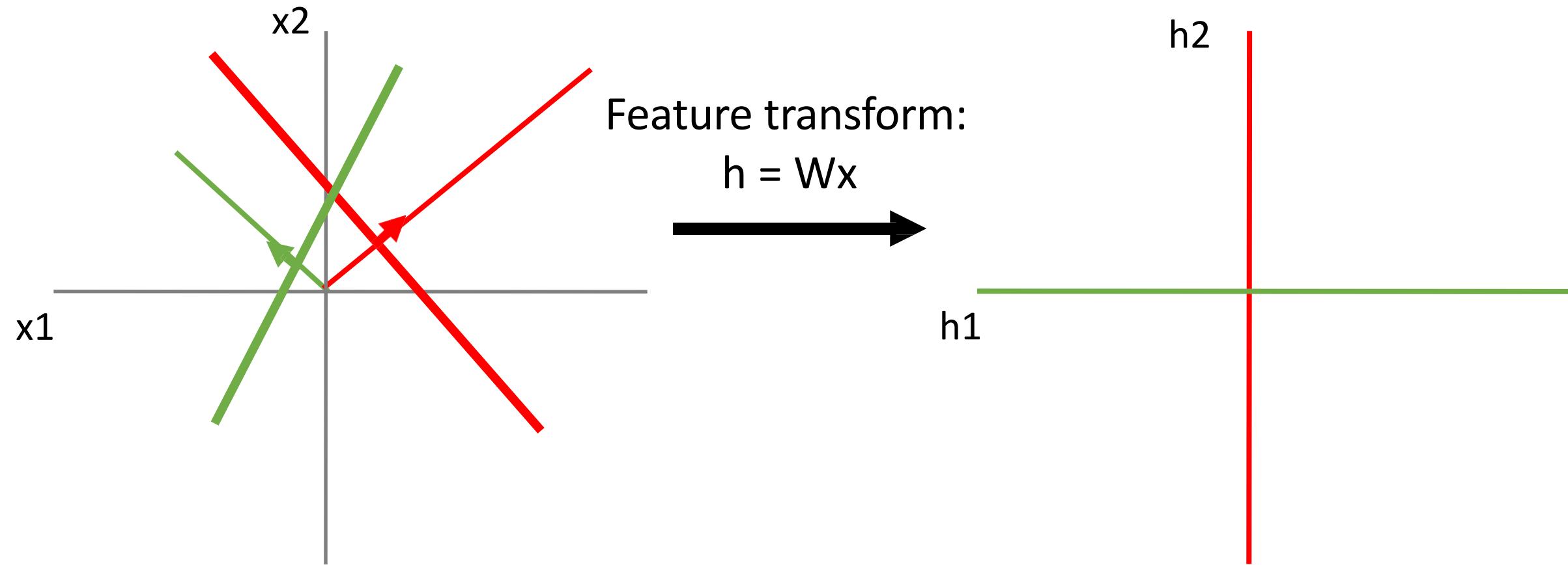
# Space Warping

Consider a linear transform:  $h = Wx$   
Where  $x, h$  are both 2-dimensional



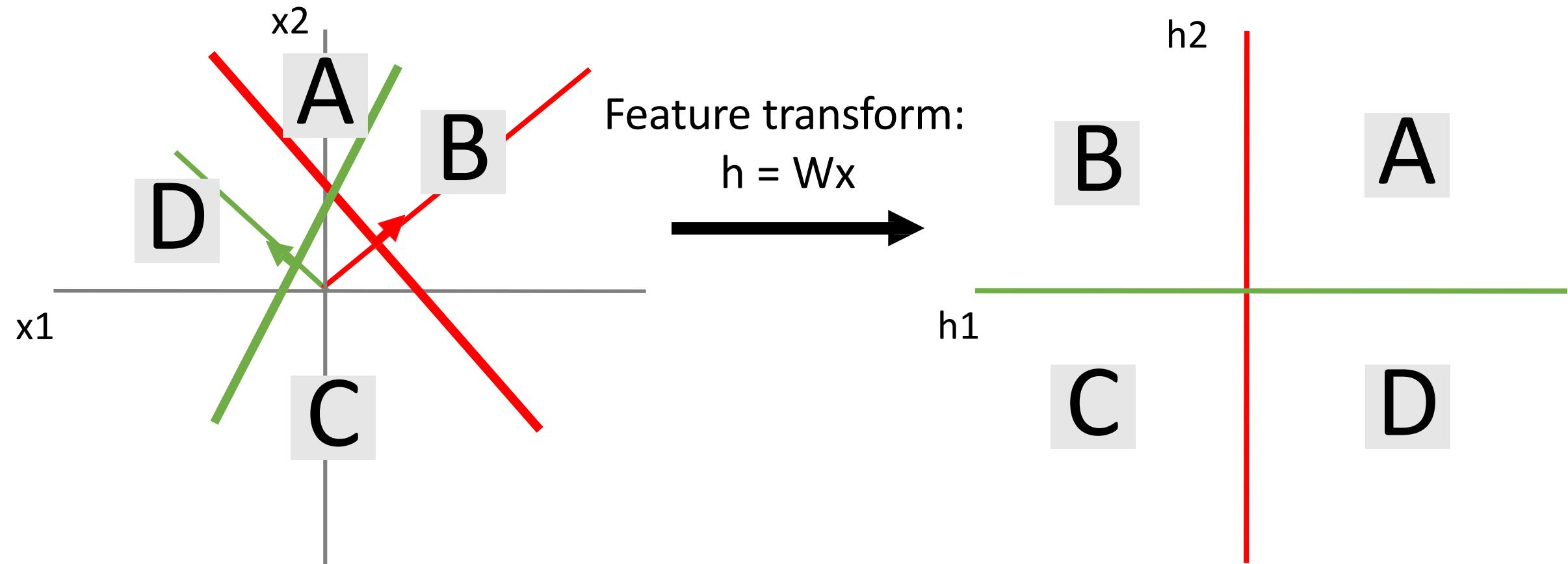
# Space Warping

Consider a linear transform:  $h = Wx$   
Where  $x, h$  are both 2-dimensional



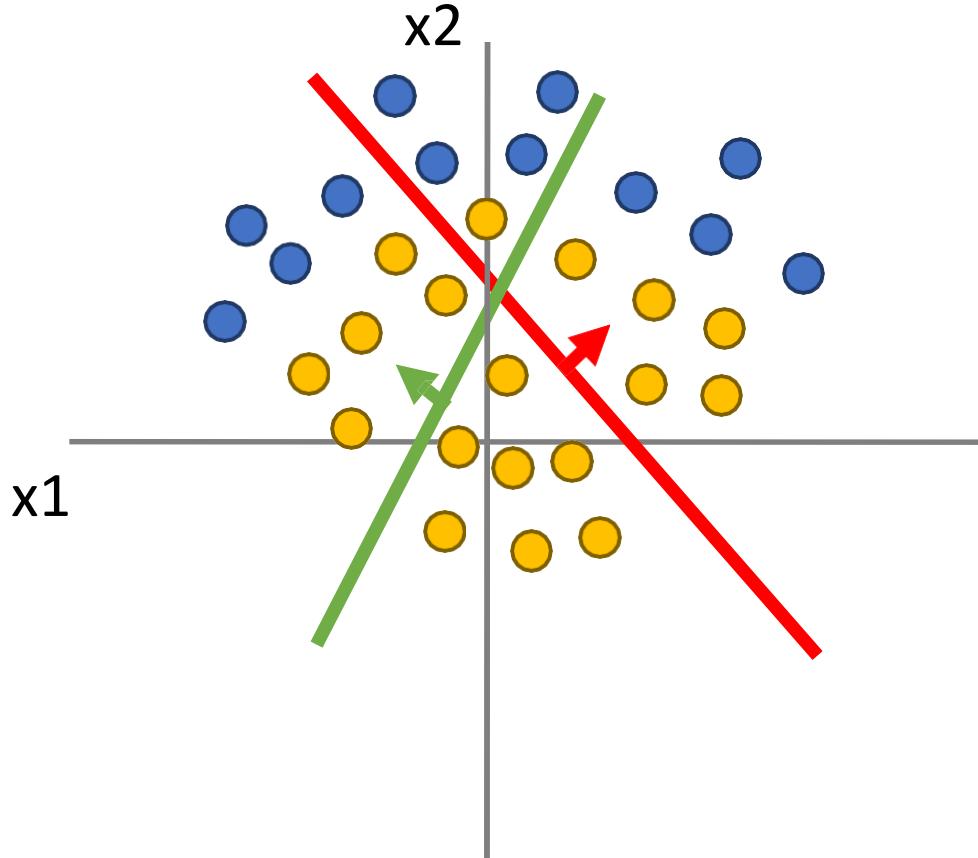
# Space Warping

Consider a linear transform:  $h = Wx$   
Where  $x, h$  are both 2-dimensional



# Space Warping

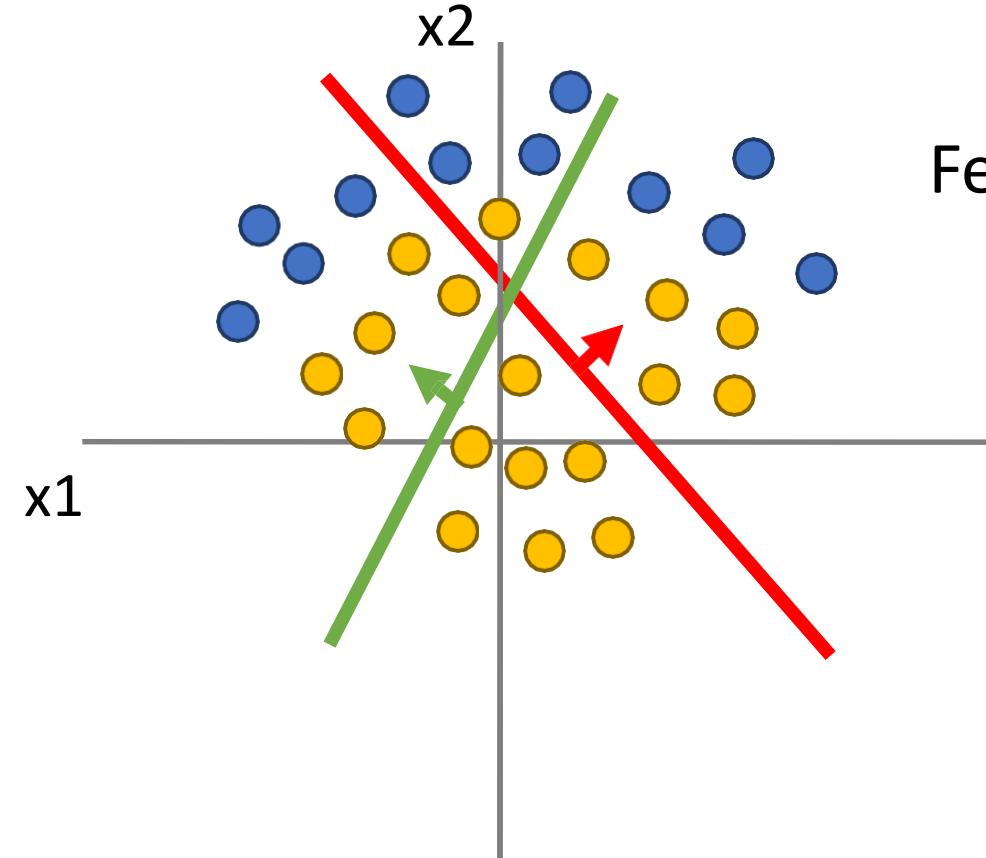
Points not linearly  
separable in original space



Consider a linear transform:  $h = Wx$   
Where  $x, h$  are both 2-dimensional

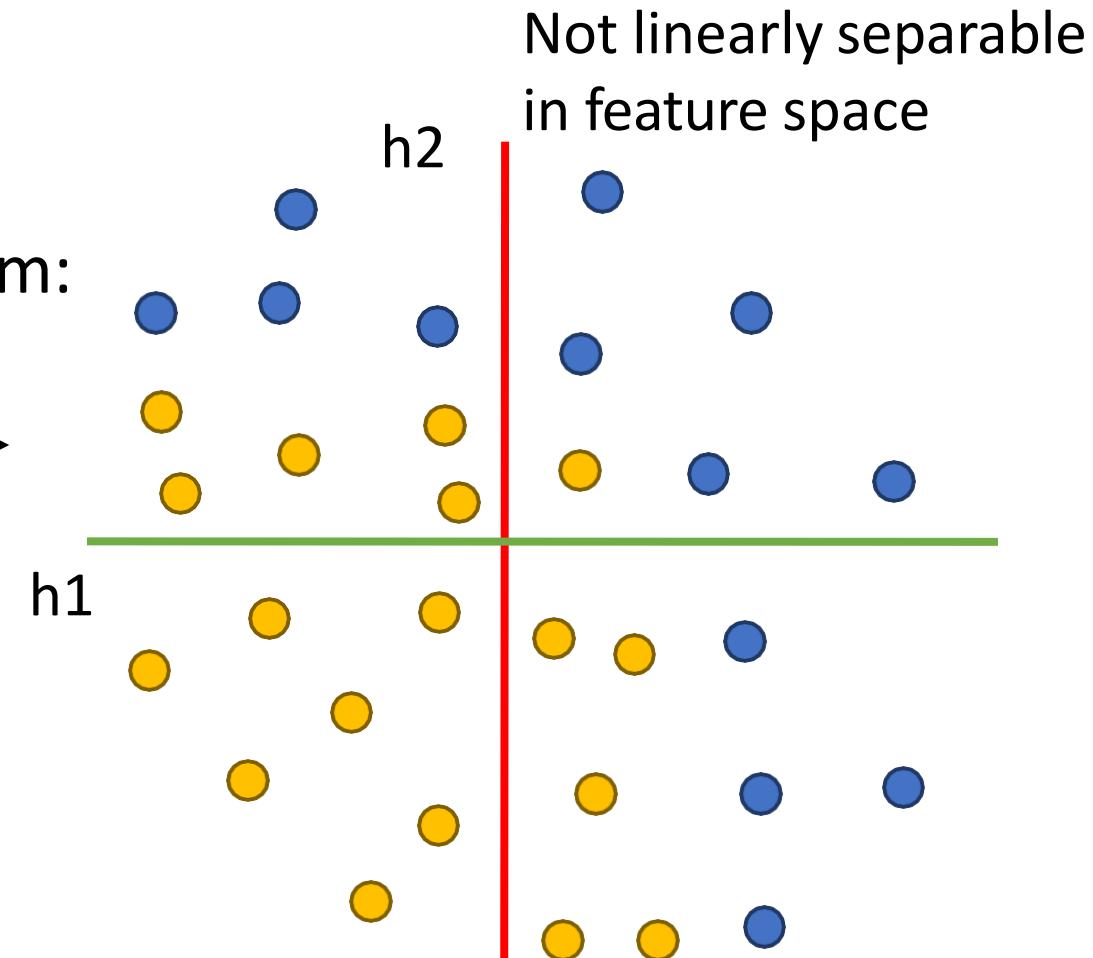
# Space Warping

Points not linearly  
separable in original space



Feature transform:  
 $h = Wx$

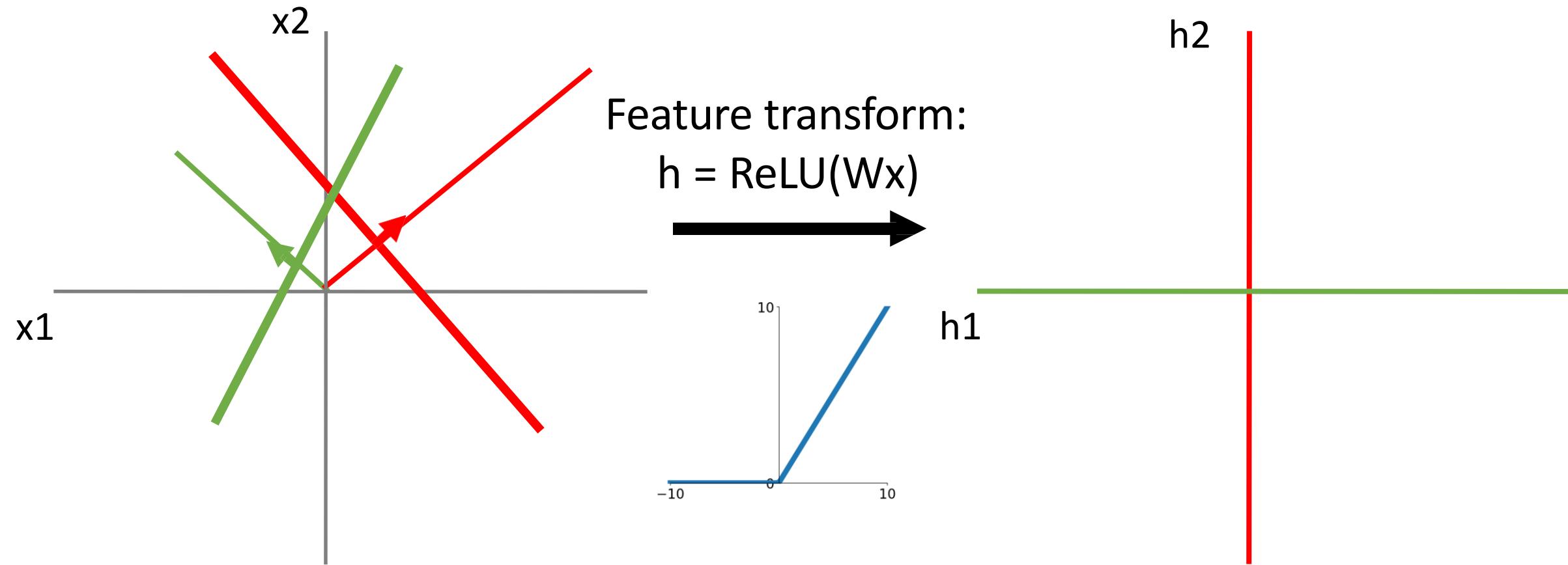
Consider a linear transform:  $h = Wx$   
Where  $x, h$  are both 2-dimensional



Not linearly separable  
in feature space

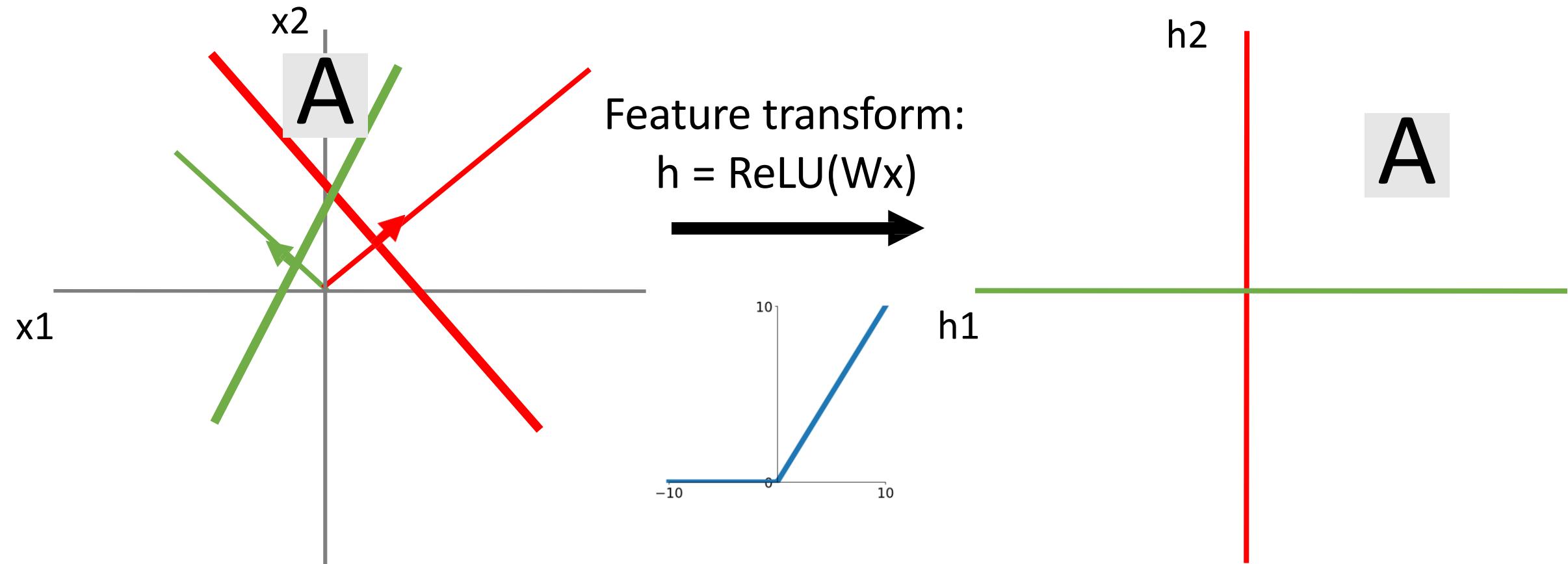
# Space Warping

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



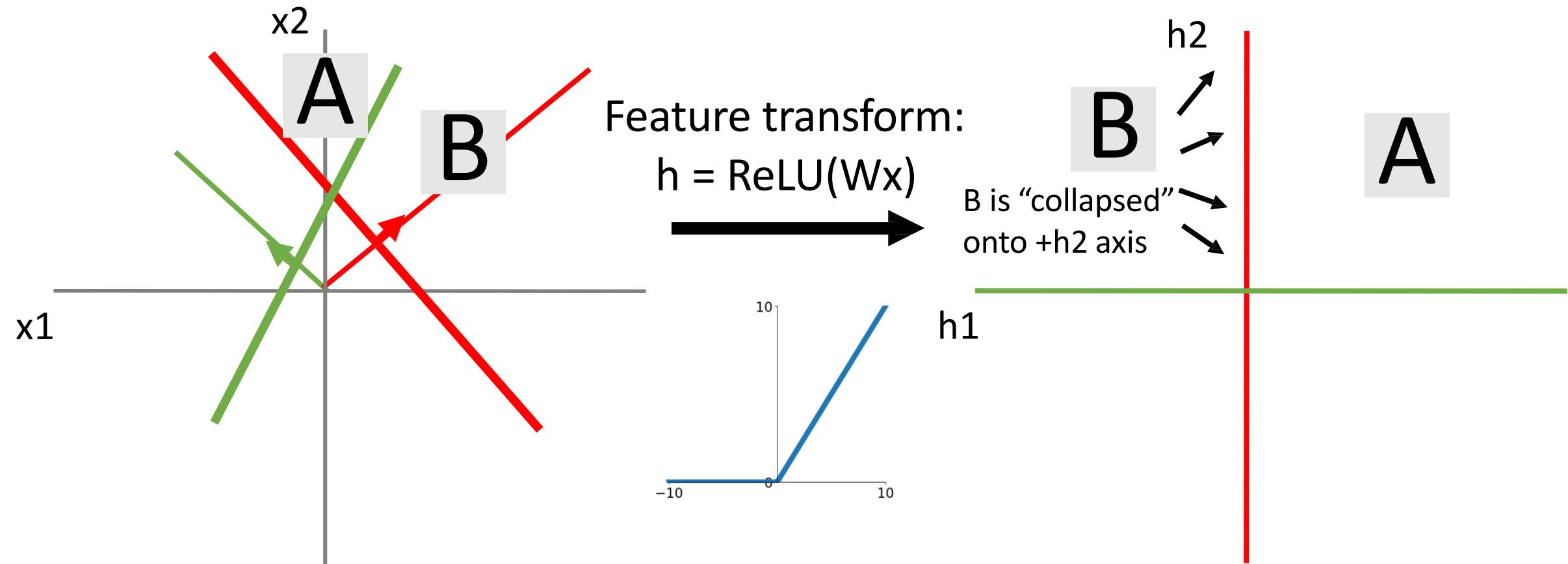
# Space Warping

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



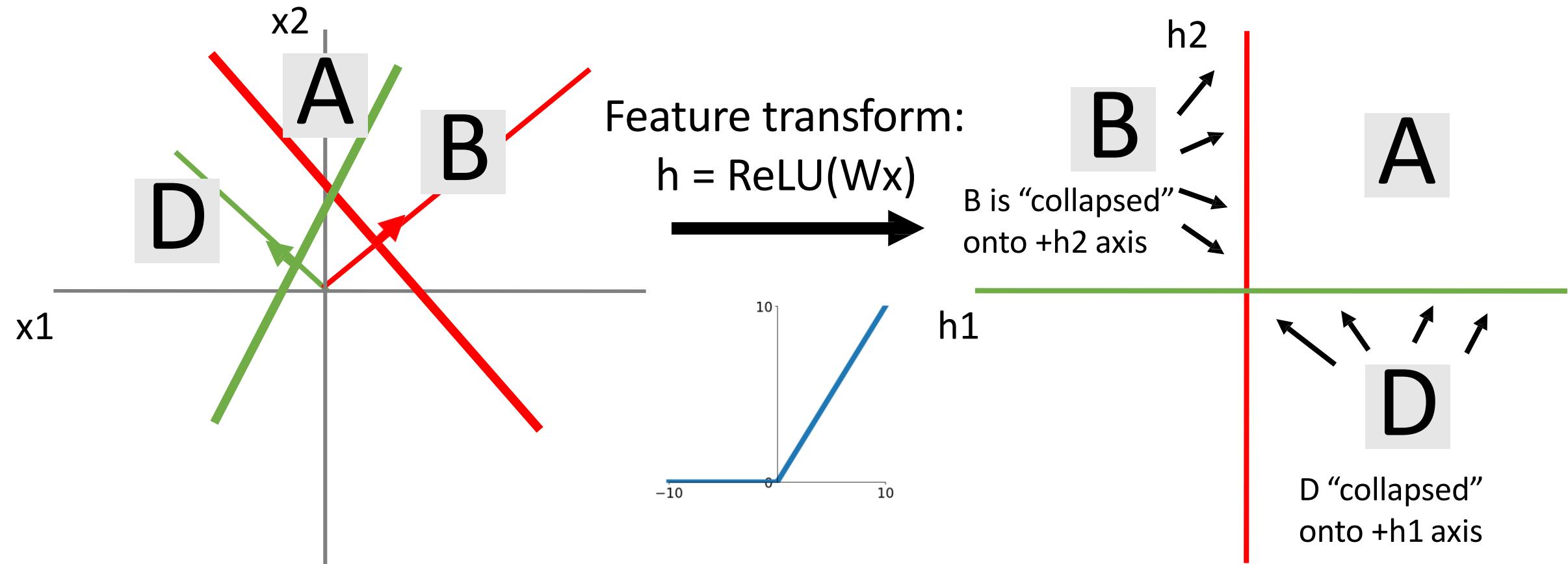
# Space Warping

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



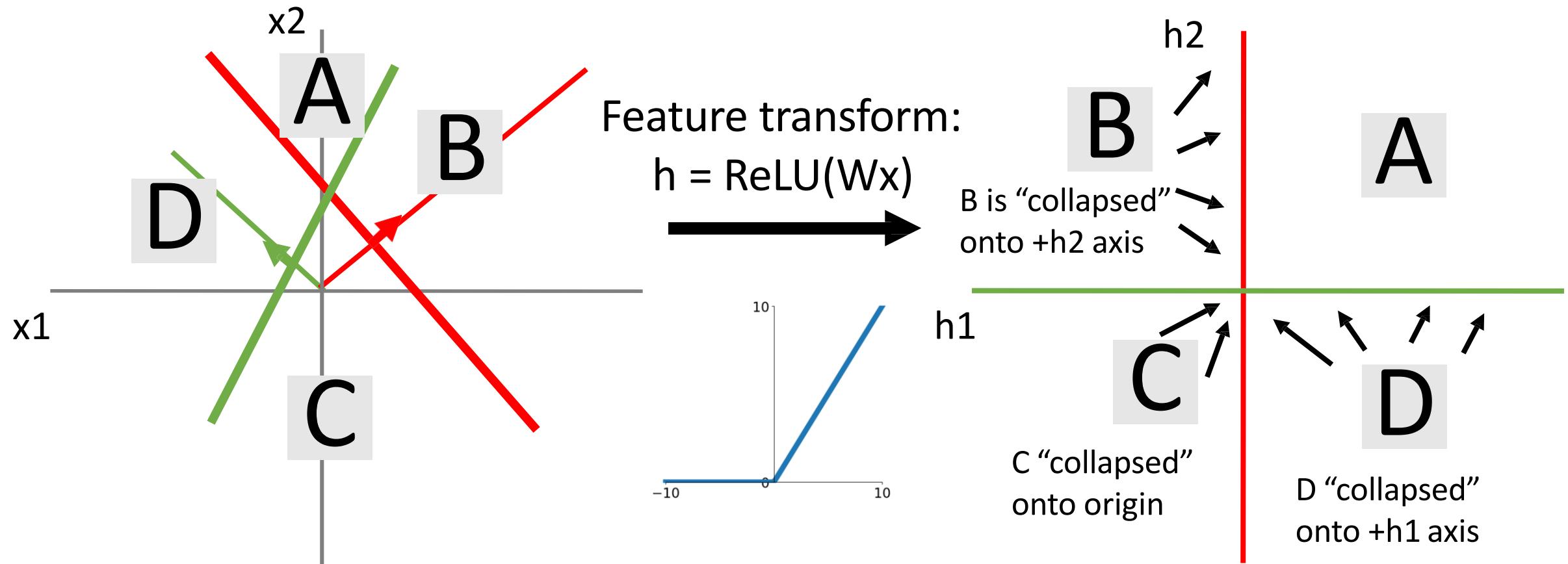
# Space Warping

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



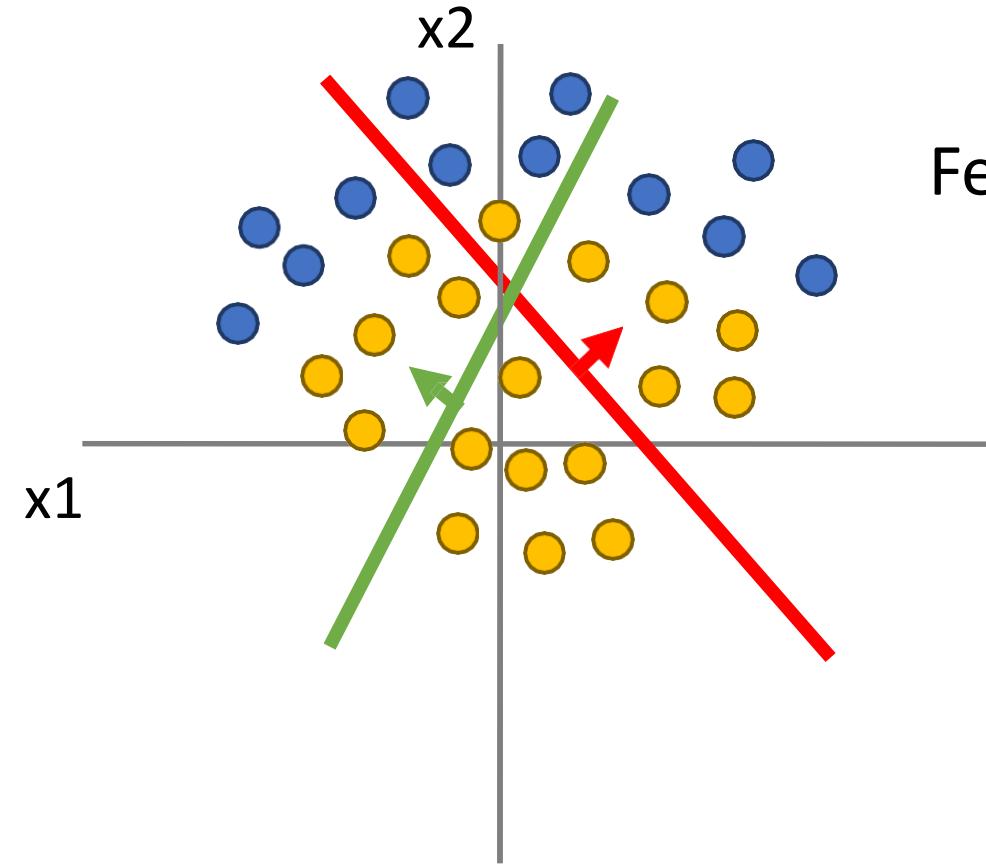
# Space Warping

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

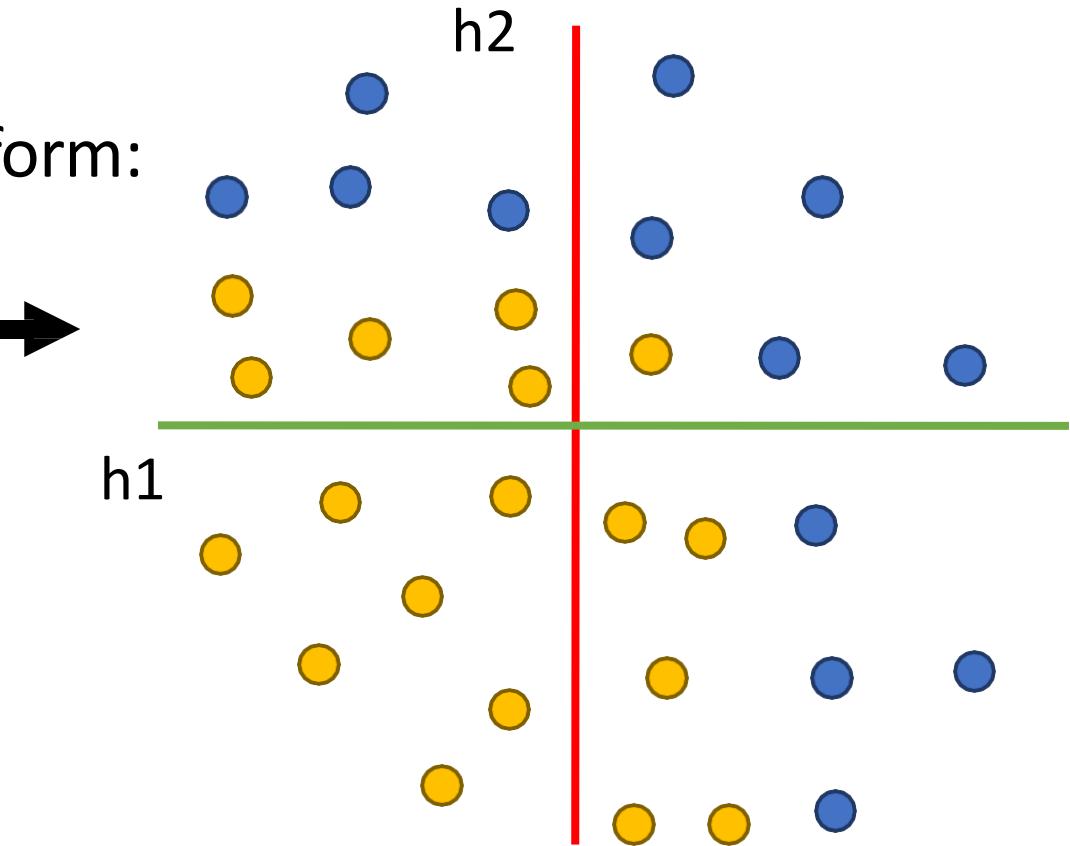


# Space Warping

Points not linearly  
separable in original space



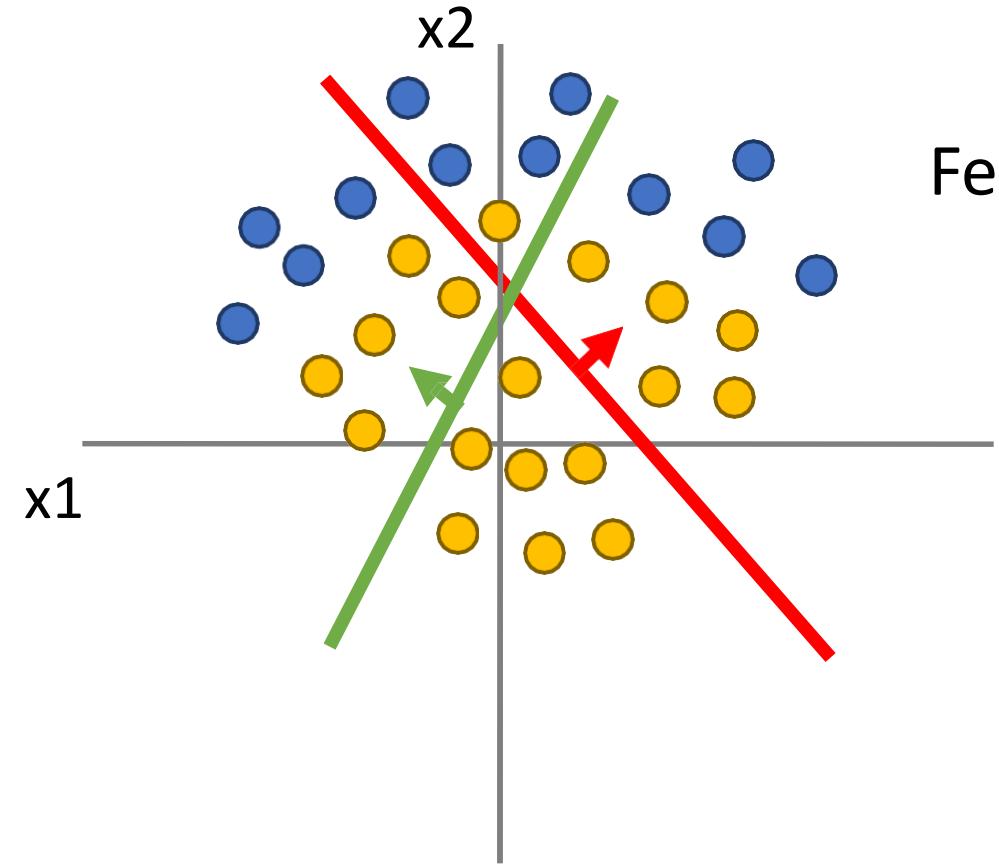
Feature transform:  
 $h = Wx$



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

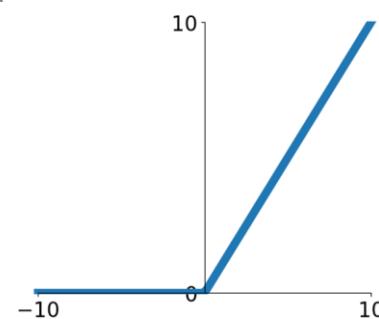
# Space Warping

Points not linearly  
separable in original space

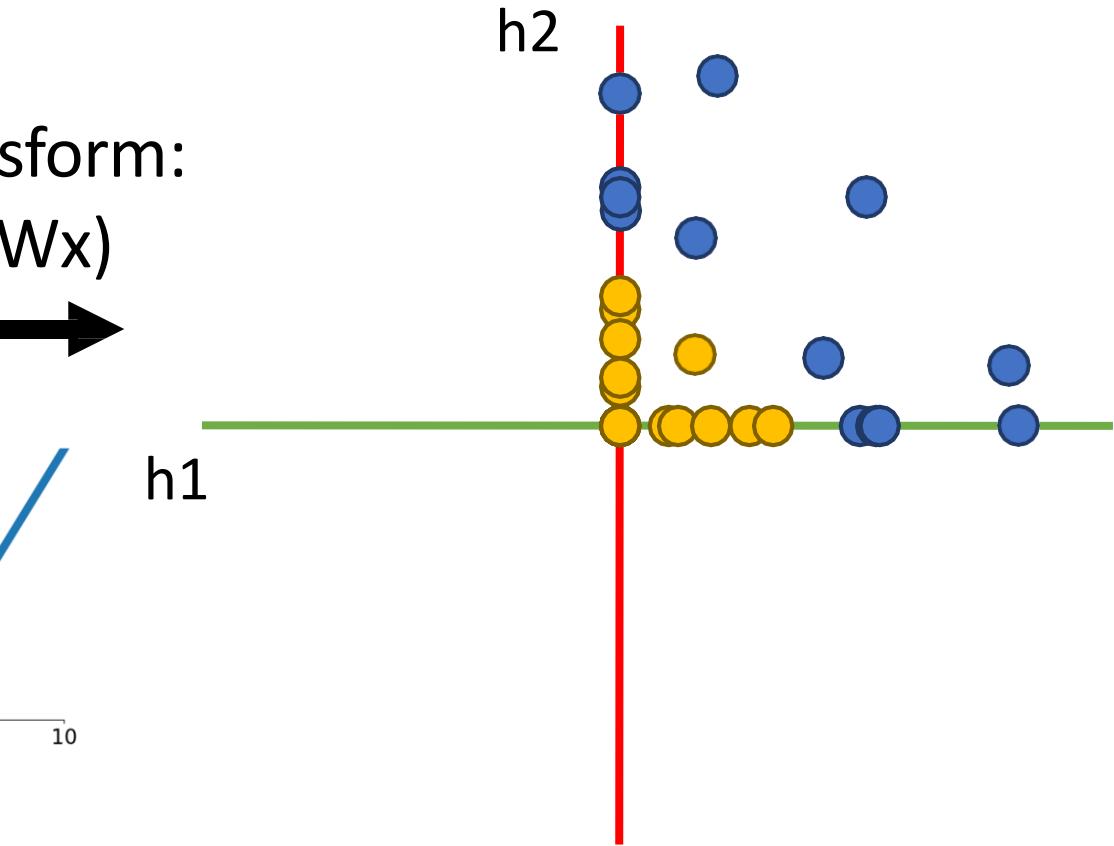


Feature transform:

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



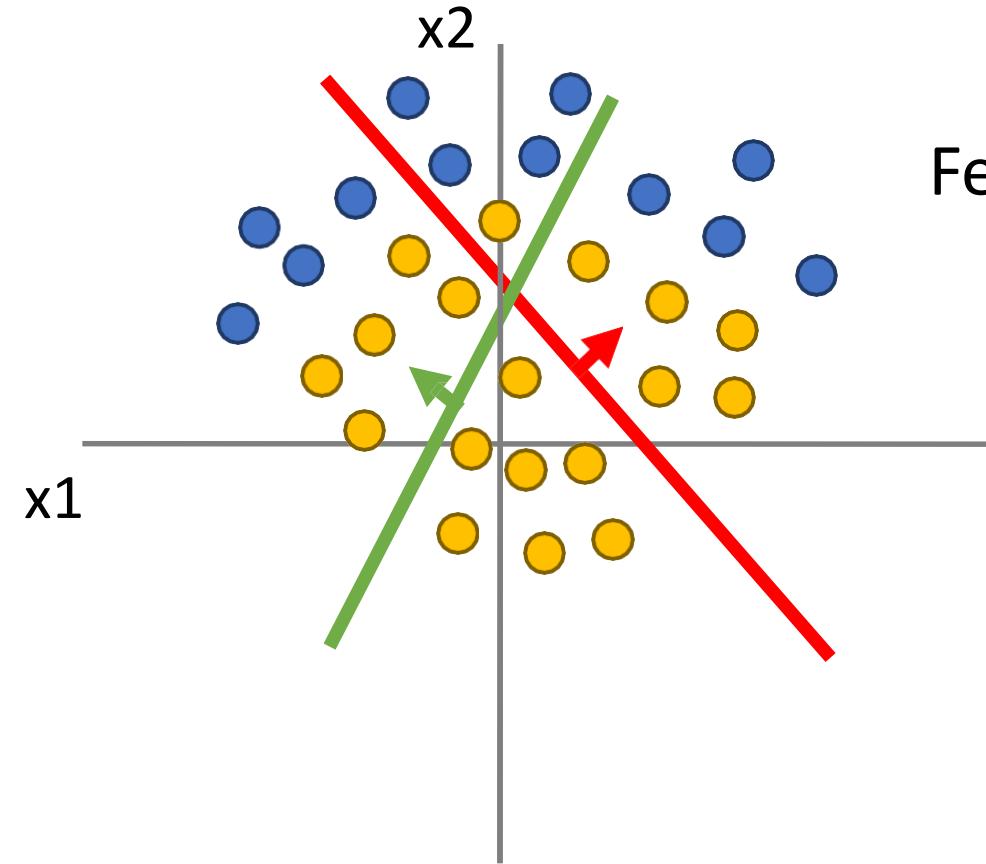
$h_1$



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

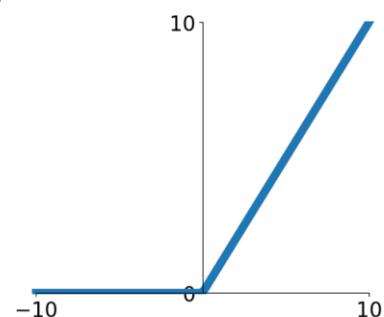
# Space Warping

Points not linearly  
separable in original space

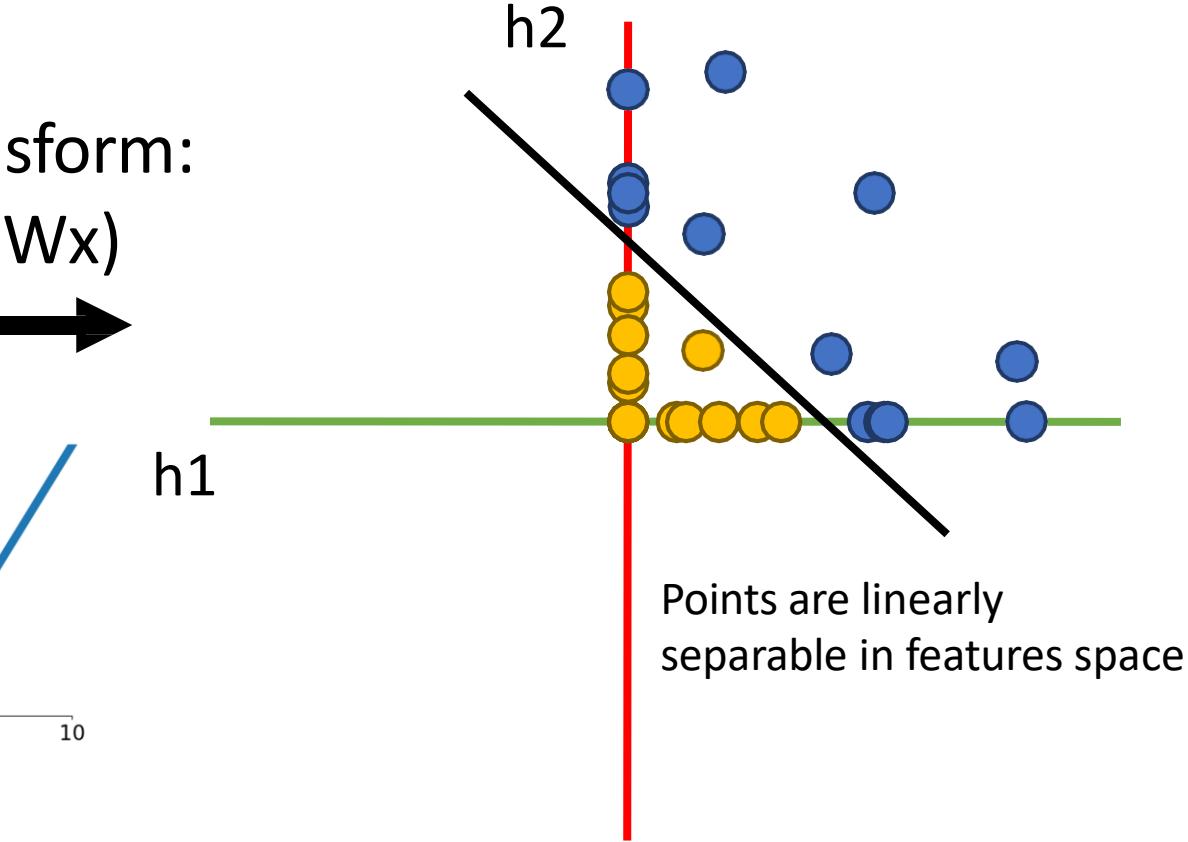


Feature transform:

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



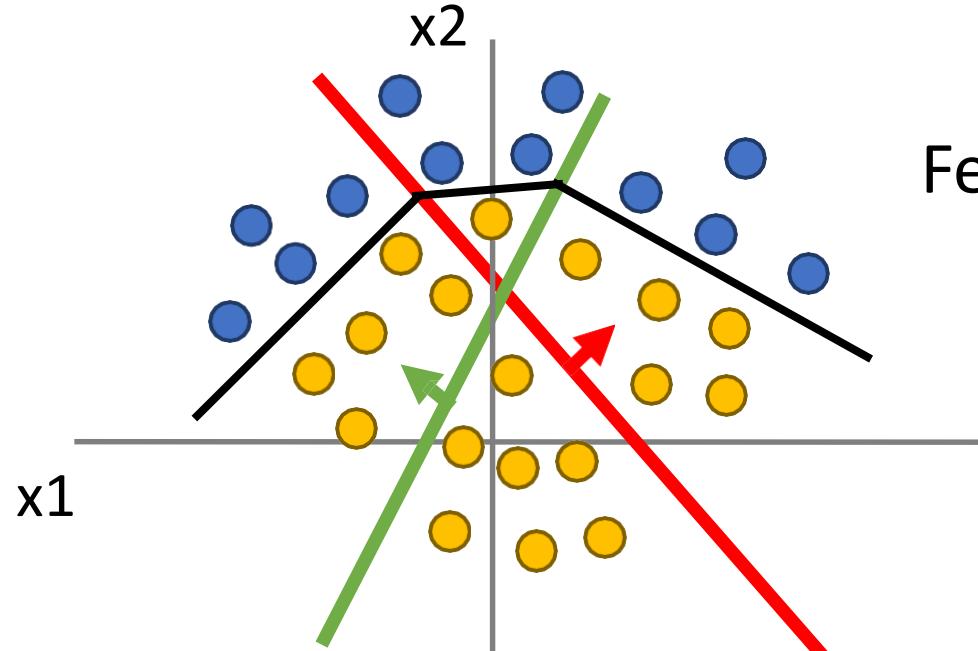
$h_1$



Points are linearly  
separable in features space!

# Space Warping

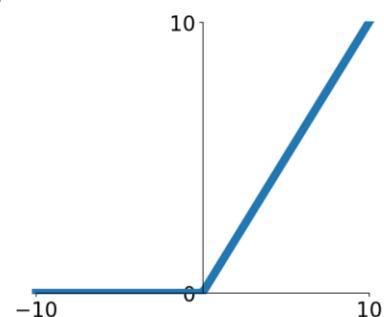
Points not linearly separable in original space



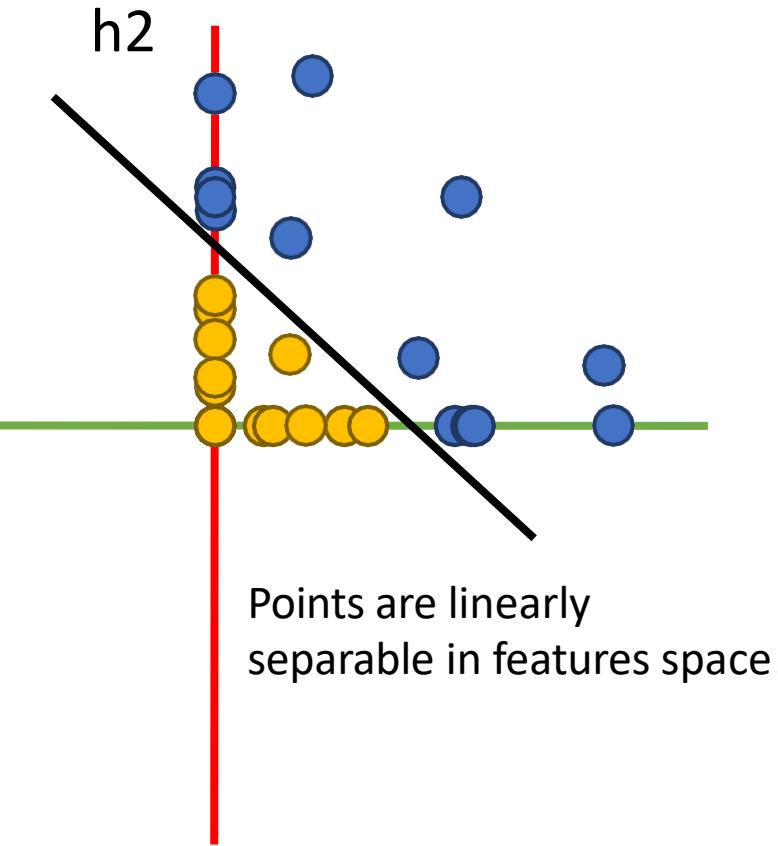
Linear classifier in feature space gives nonlinear classifier in original space

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

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



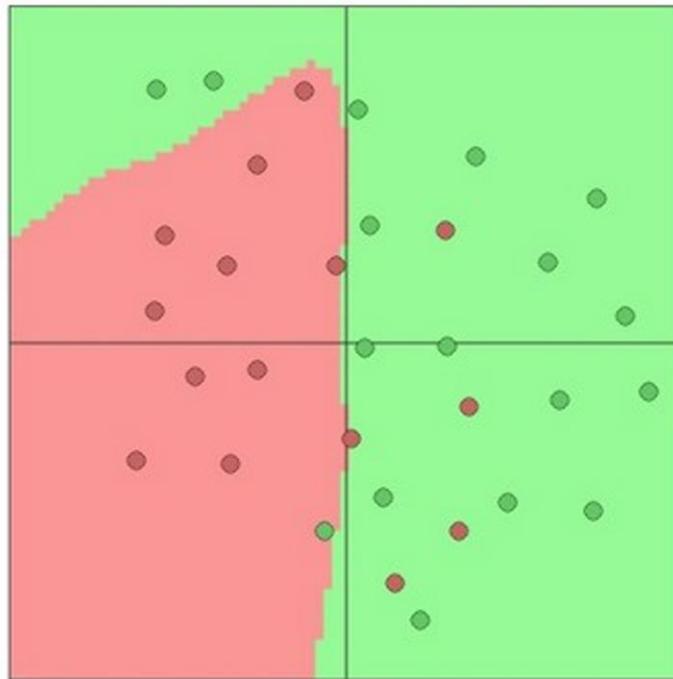
$h_1$



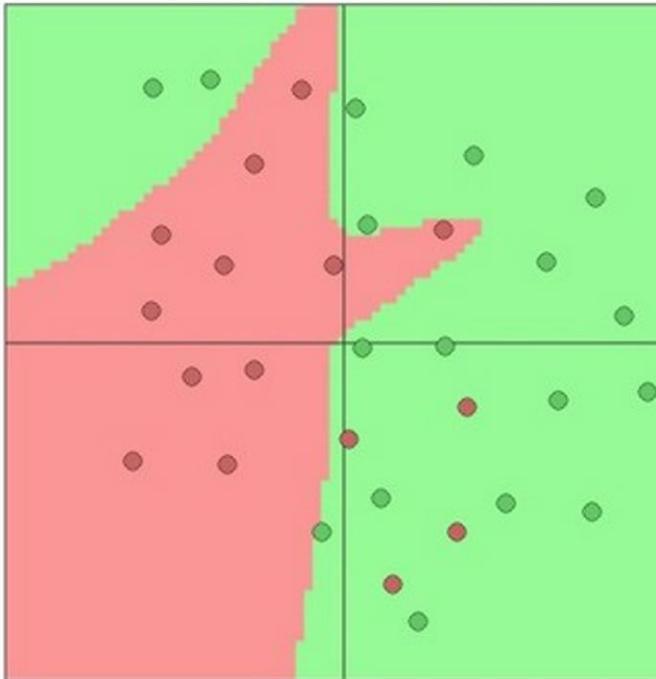
Points are linearly separable in features space!

# Setting the number of layers and their sizes

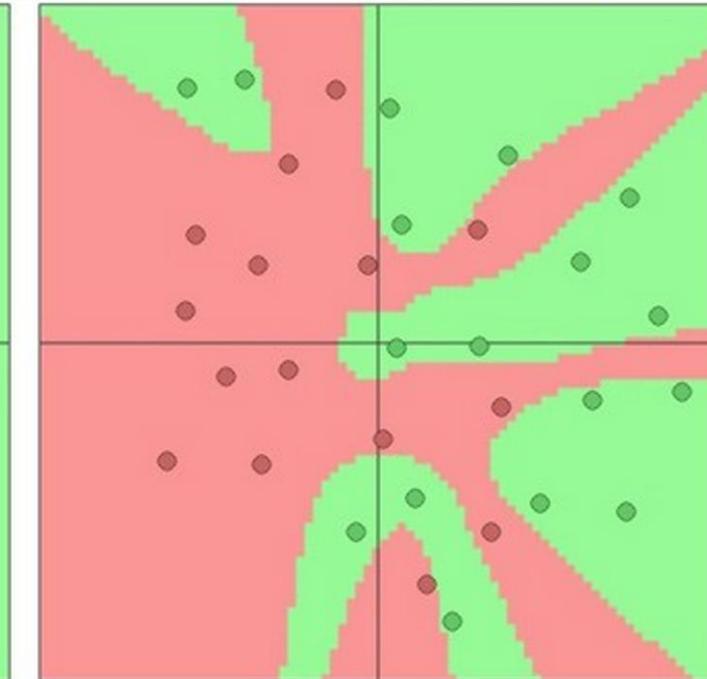
3 hidden units



6 hidden units



20 hidden units



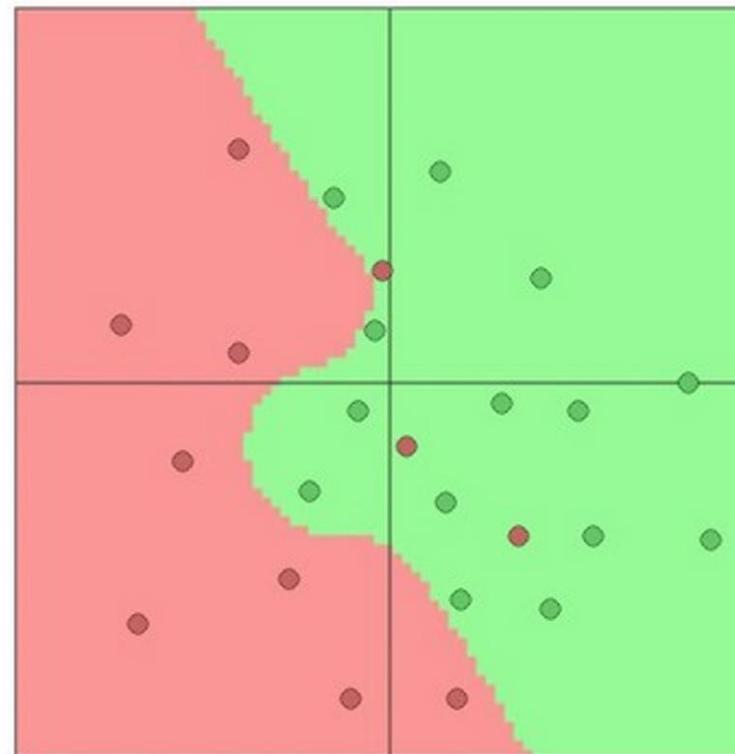
More hidden units = more capacity

Don't regularize with size; instead use stronger L2

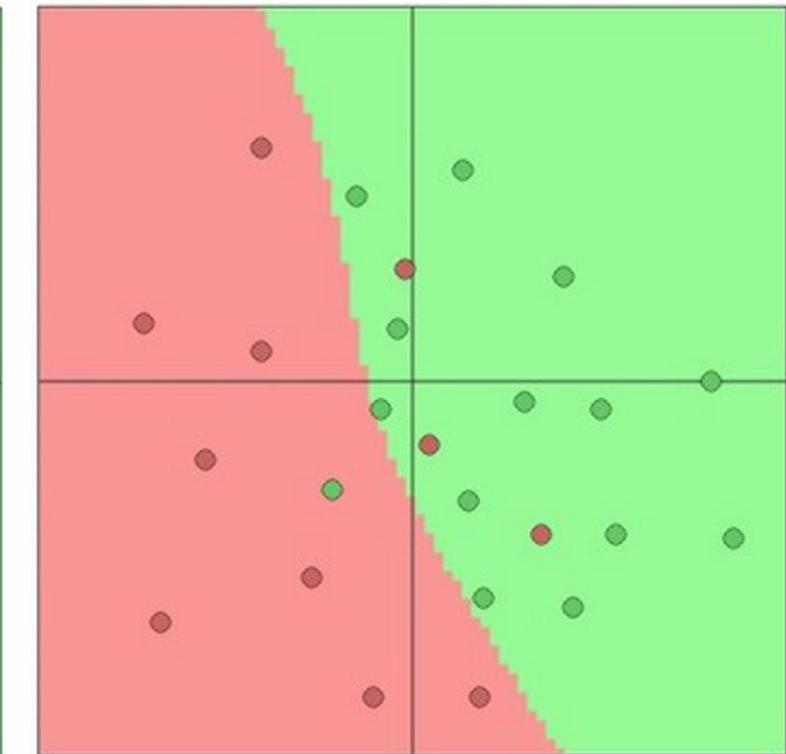
$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.1$



(Web demo with ConvNetJS:

<http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>)

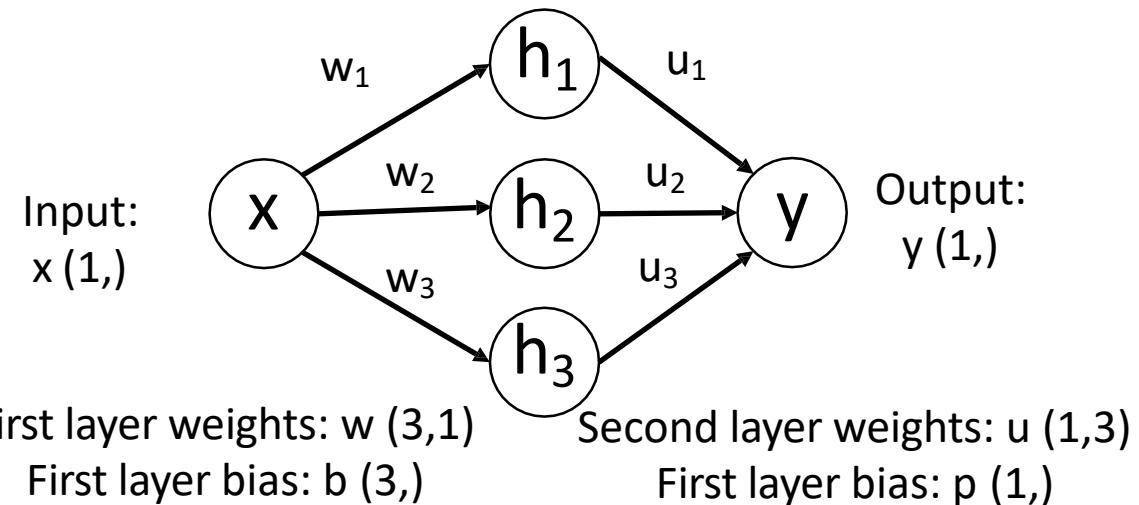
# Universal Approximation

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

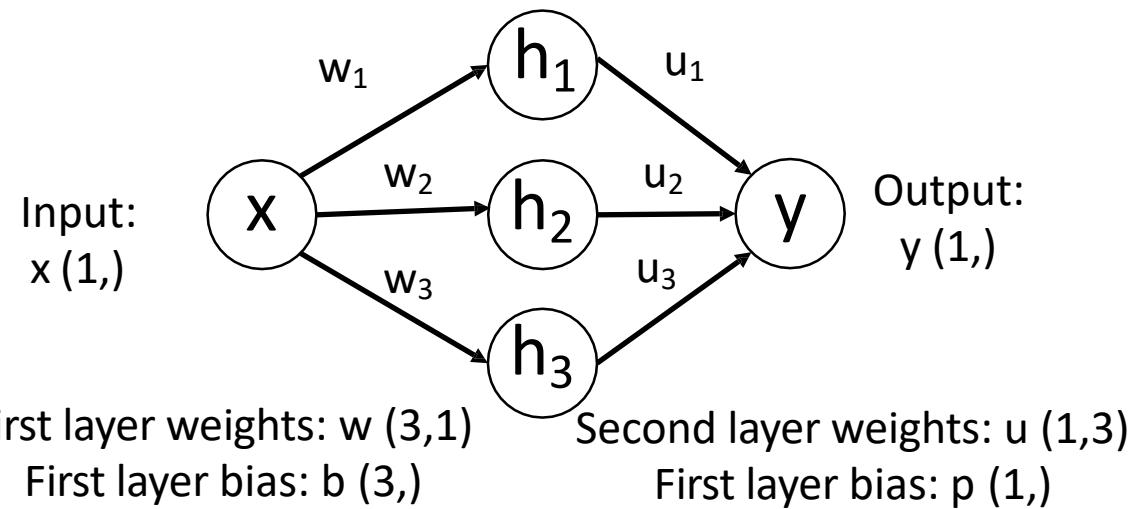
# Universal Approximation

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



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

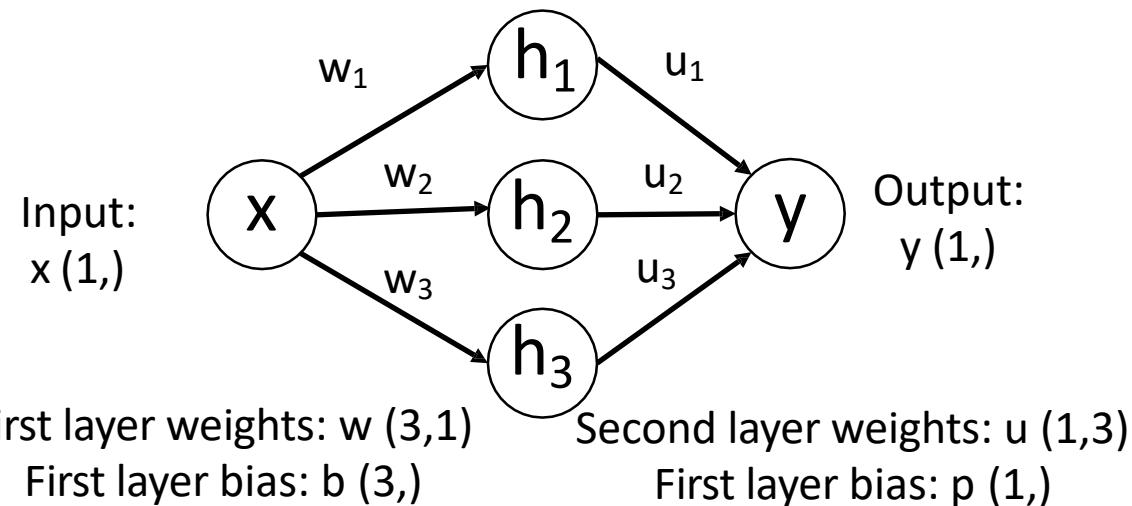
$$h_2 = \max(0, w_2 * x + b_2)$$

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

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

# Universal Approximation

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



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

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

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

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

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

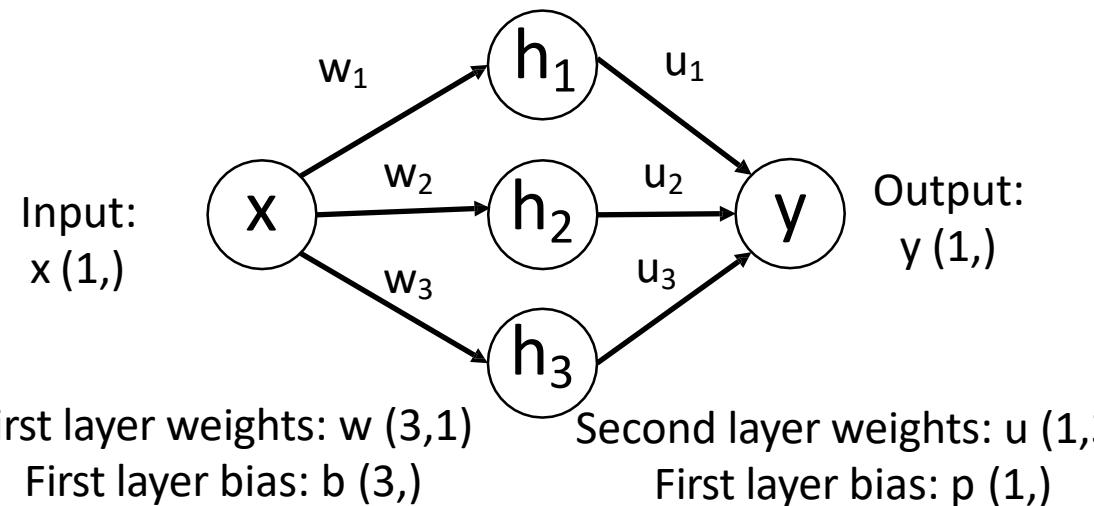
$$+ u_2 * \max(0, w_2 * x + b_2)$$

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

$$+ p$$

# Universal Approximation

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



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

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

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

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

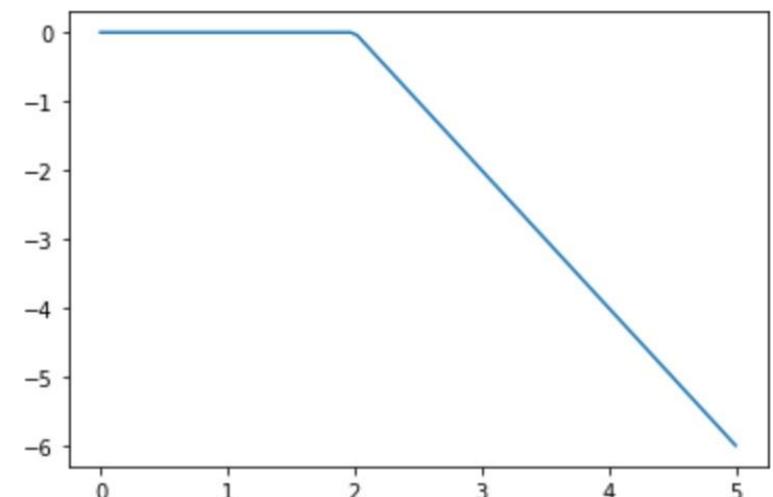
$$y = u_1 * \max(0, w_1 * x + b_1)$$

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

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

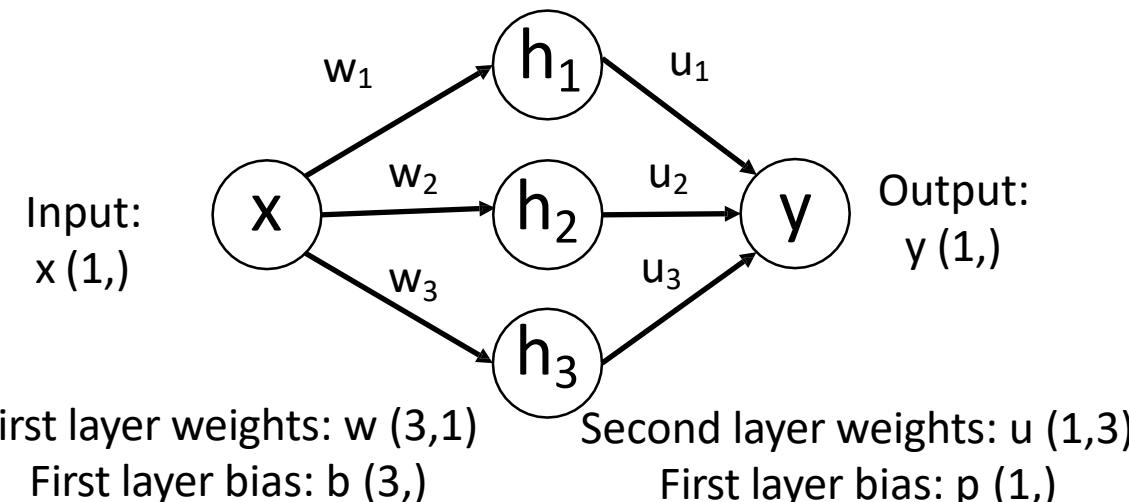
$$+ p$$

Output is a sum of shifted, scaled ReLUs:



# Universal Approximation

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



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

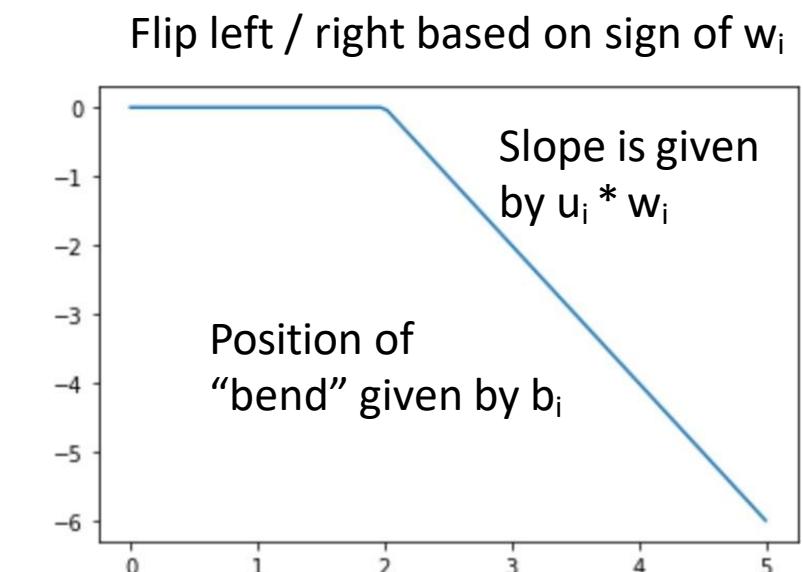
$$h_2 = \max(0, w_2 * x + b_2)$$

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

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

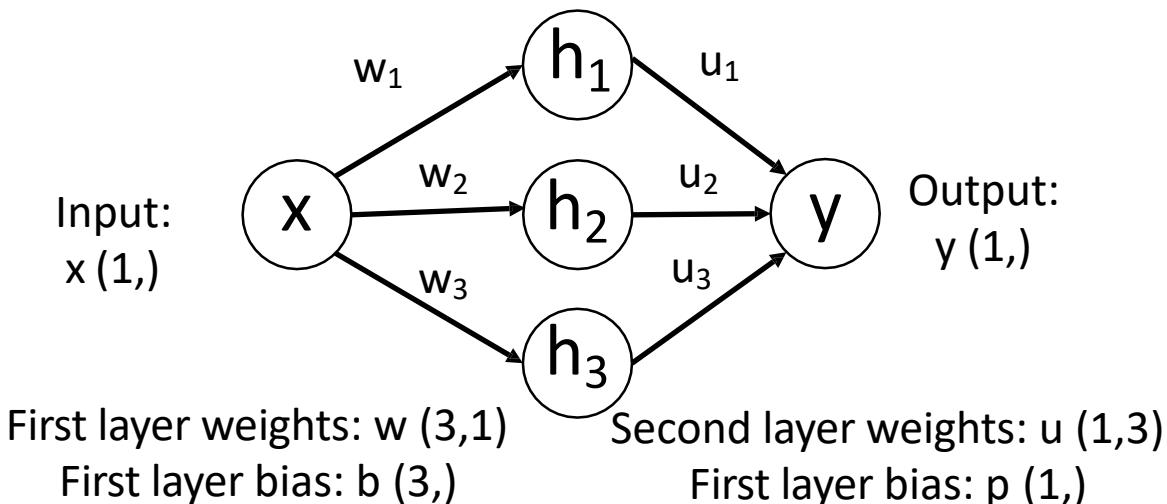
$$\begin{aligned} y = & u_1 * \max(0, w_1 * x + b_1) \\ & + u_2 * \max(0, w_2 * x + b_2) \\ & + u_3 * \max(0, w_3 * x + b_3) \\ & + p \end{aligned}$$

Output is a sum of shifted, scaled ReLUs:



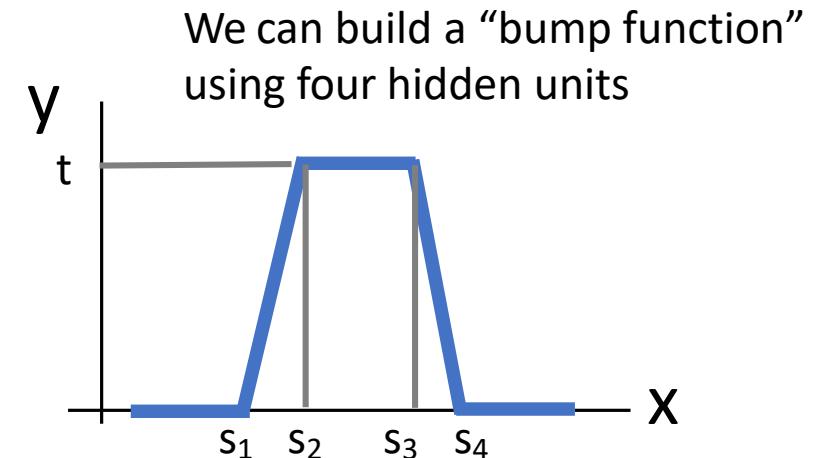
# Universal Approximation

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



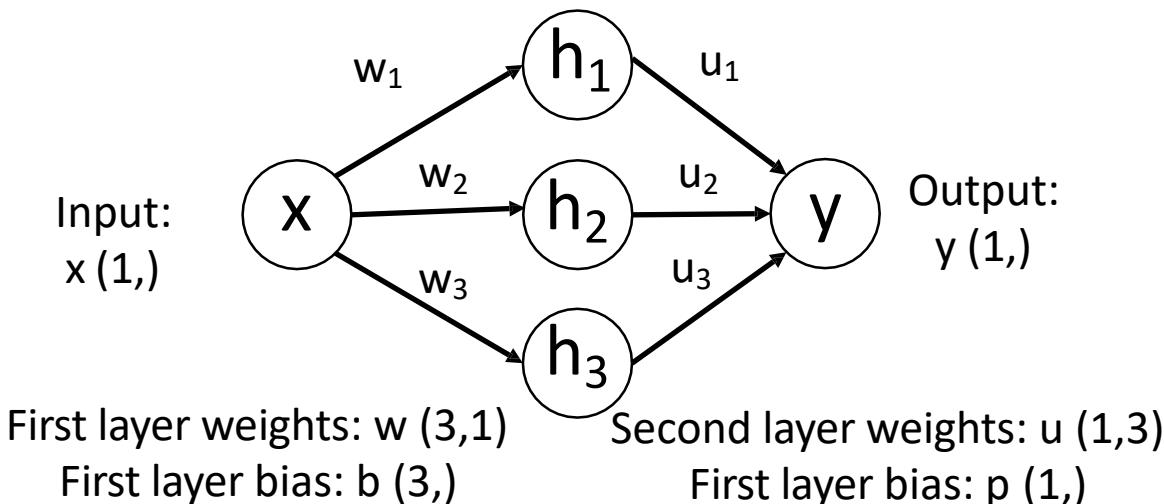
$$\begin{aligned} h_1 &= \max(0, w_1 * x + b_1) \\ h_2 &= \max(0, w_2 * x + b_2) \\ h_3 &= \max(0, w_3 * x + b_3) \\ y &= u_1 * h_1 + u_2 * h_2 + u_3 * h_3 + p \end{aligned}$$

$$\begin{aligned} y &= u_1 * \max(0, w_1 * x + b_1) \\ &\quad + u_2 * \max(0, w_2 * x + b_2) \\ &\quad + u_3 * \max(0, w_3 * x + b_3) \\ &\quad + p \end{aligned}$$



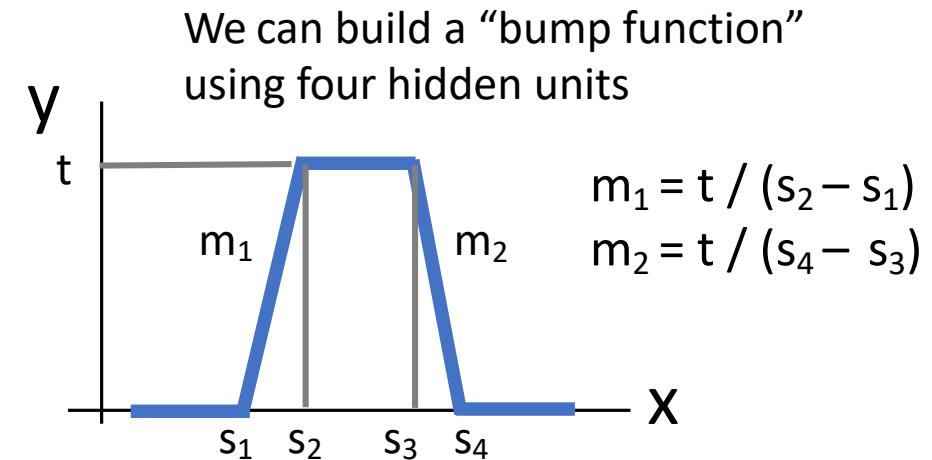
# Universal Approximation

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



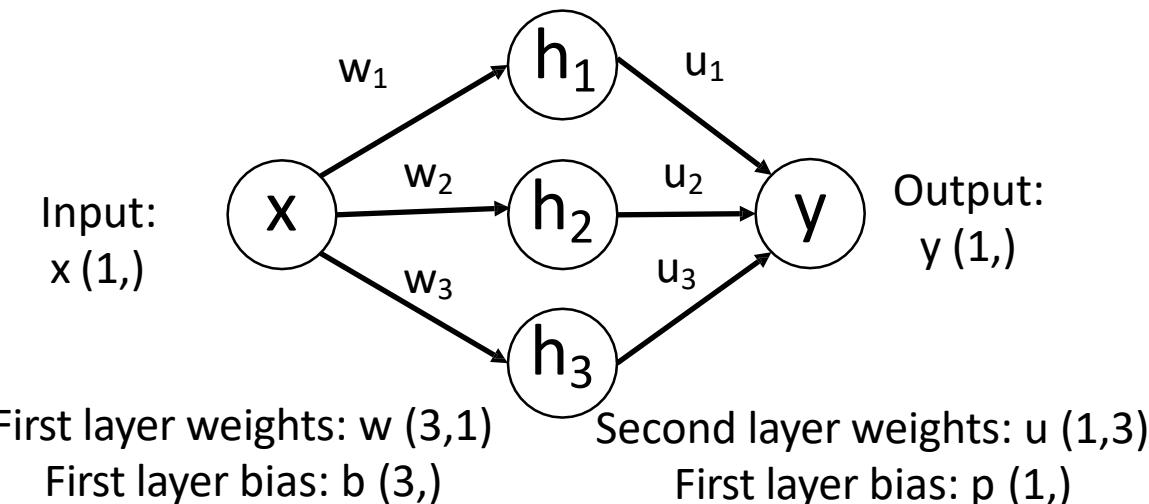
$$\begin{aligned} h_1 &= \max(0, w_1 * x + b_1) \\ h_2 &= \max(0, w_2 * x + b_2) \\ h_3 &= \max(0, w_3 * x + b_3) \\ y &= u_1 * h_1 + u_2 * h_2 + u_3 * h_3 + p \end{aligned}$$

$$\begin{aligned} y &= u_1 * \max(0, w_1 * x + b_1) \\ &\quad + u_2 * \max(0, w_2 * x + b_2) \\ &\quad + u_3 * \max(0, w_3 * x + b_3) \\ &\quad + p \end{aligned}$$



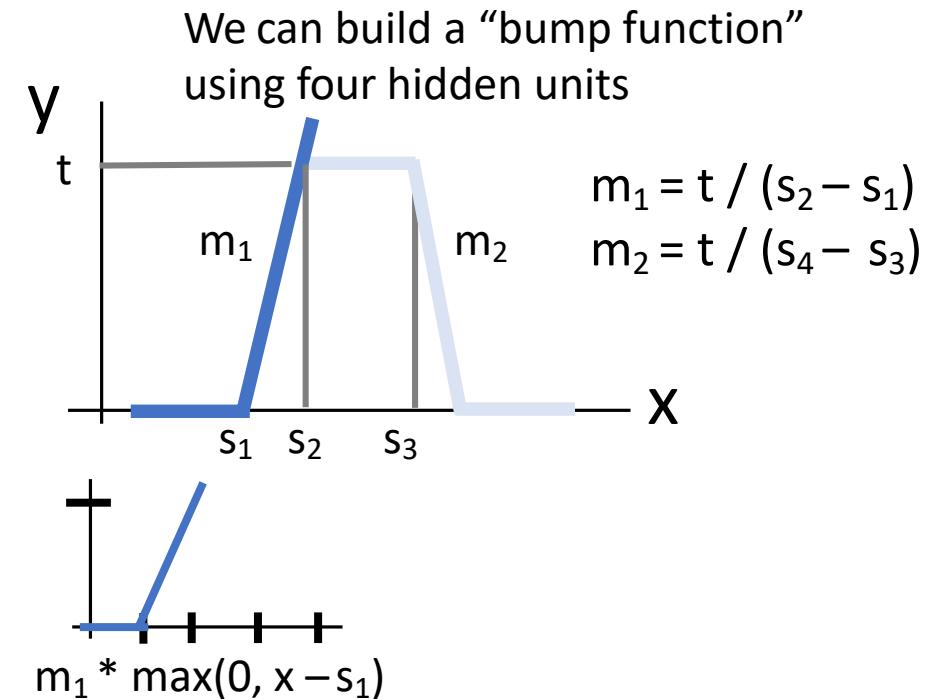
# Universal Approximation

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



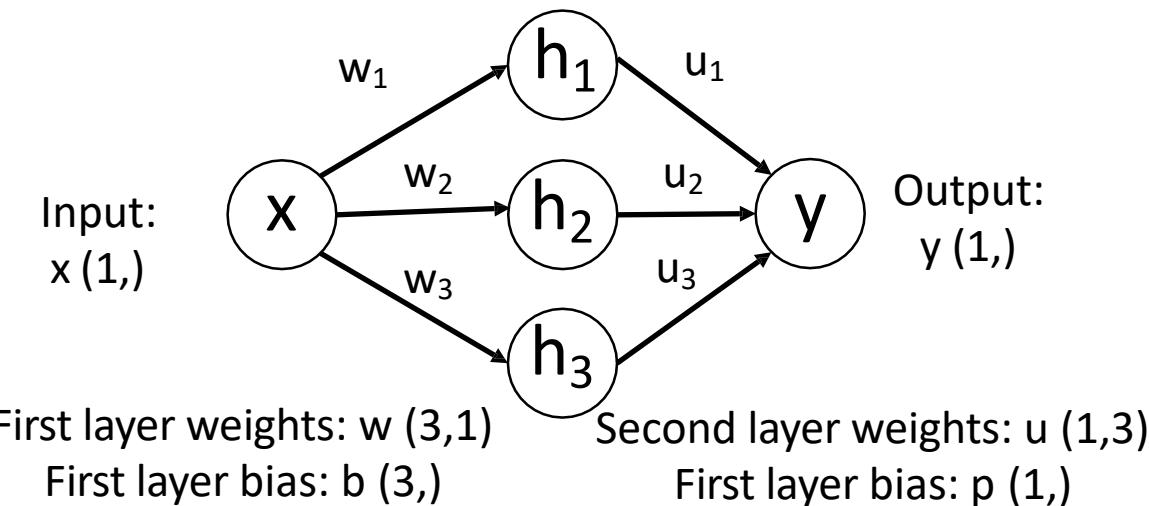
$$\begin{aligned} h_1 &= \max(0, w_1 * x + b_1) \\ h_2 &= \max(0, w_2 * x + b_2) \\ h_3 &= \max(0, w_3 * x + b_3) \\ y &= u_1 * h_1 + u_2 * h_2 + u_3 * h_3 + p \end{aligned}$$

$$\begin{aligned} y &= u_1 * \max(0, w_1 * x + b_1) \\ &\quad + u_2 * \max(0, w_2 * x + b_2) \\ &\quad + u_3 * \max(0, w_3 * x + b_3) \\ &\quad + p \end{aligned}$$



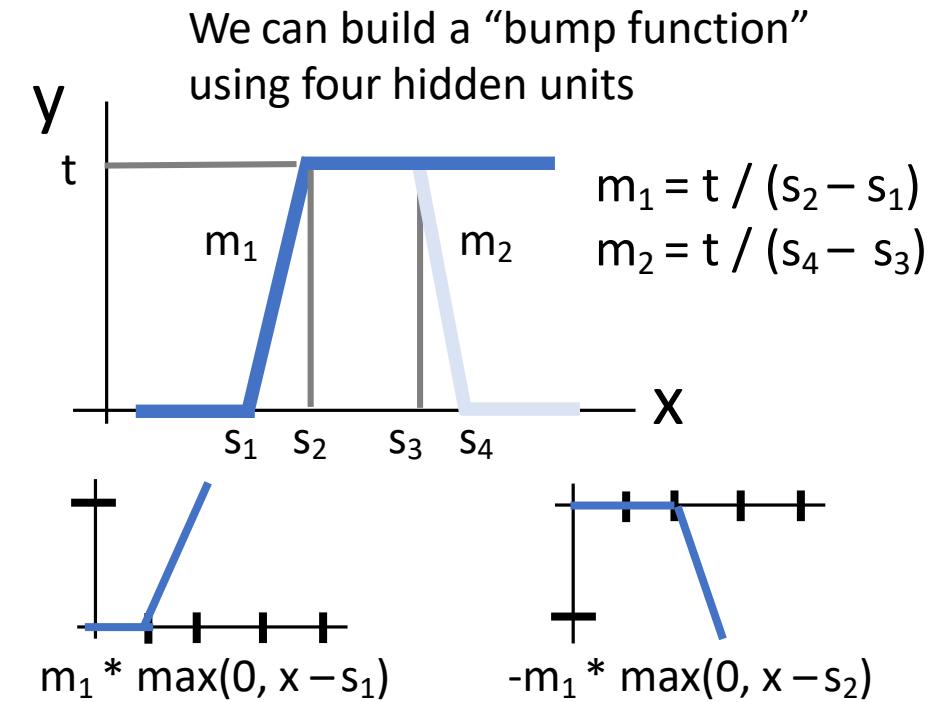
# Universal Approximation

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



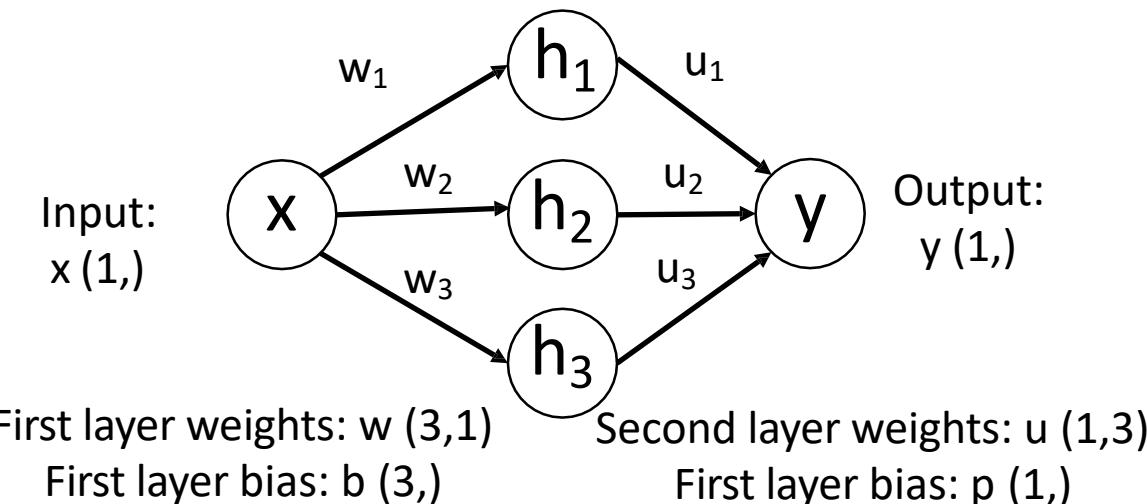
$$\begin{aligned} h_1 &= \max(0, w_1 * x + b_1) \\ h_2 &= \max(0, w_2 * x + b_2) \\ h_3 &= \max(0, w_3 * x + b_3) \\ y &= u_1 * h_1 + u_2 * h_2 + u_3 * h_3 + p \end{aligned}$$

$$\begin{aligned} y &= u_1 * \max(0, w_1 * x + b_1) \\ &\quad + u_2 * \max(0, w_2 * x + b_2) \\ &\quad + u_3 * \max(0, w_3 * x + b_3) \\ &\quad + p \end{aligned}$$



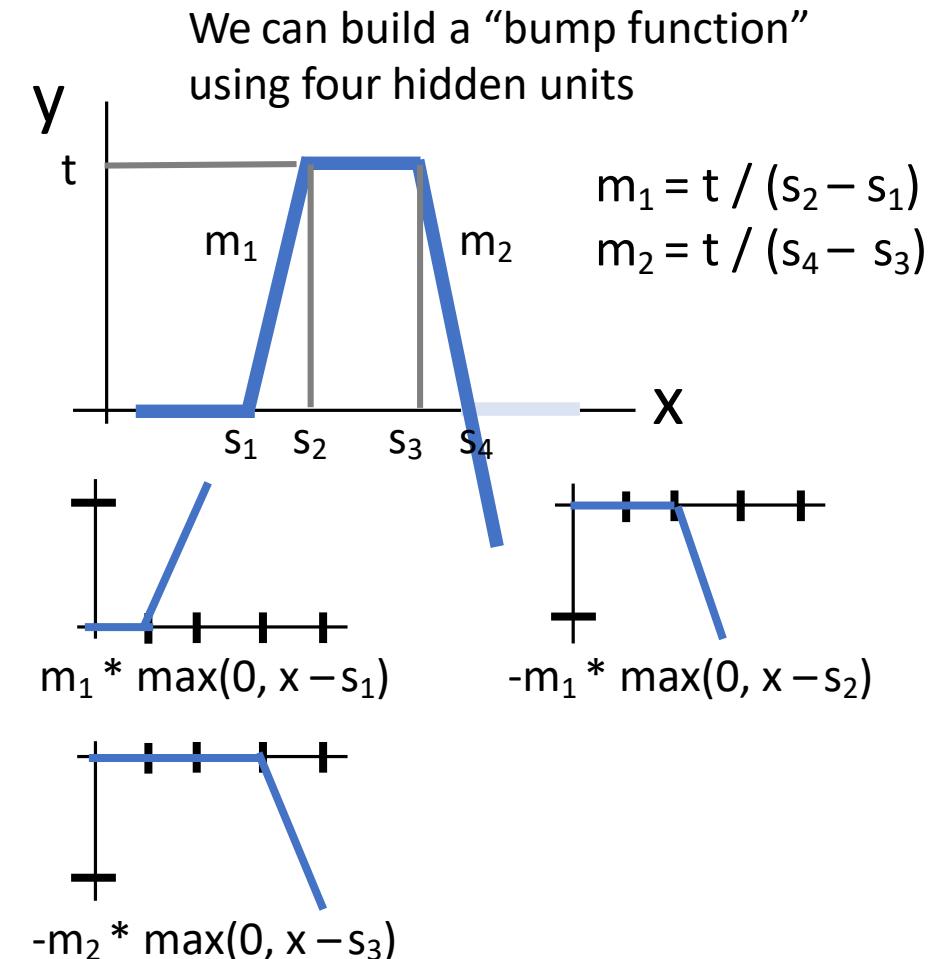
# Universal Approximation

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



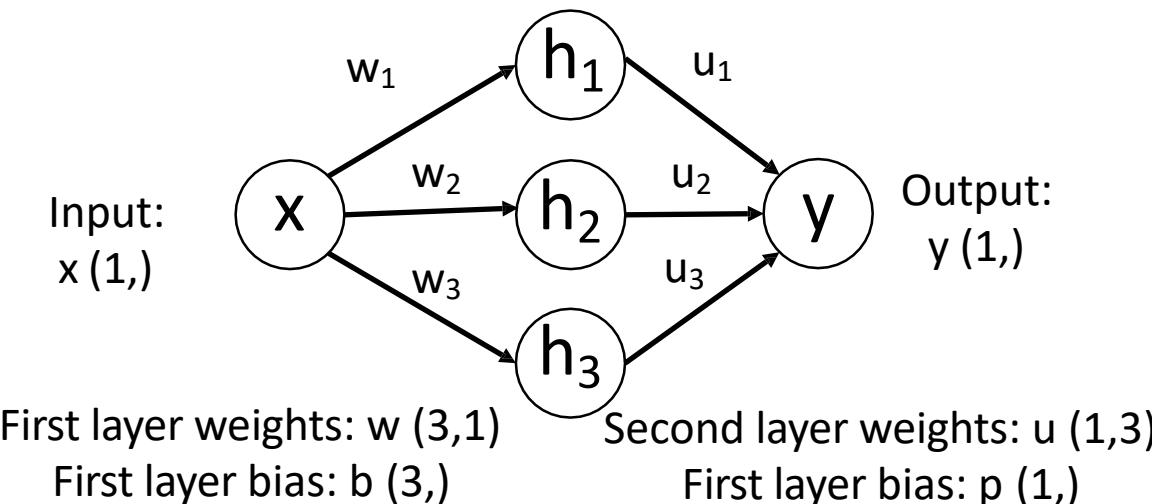
$$\begin{aligned} h_1 &= \max(0, w_1 * x + b_1) \\ h_2 &= \max(0, w_2 * x + b_2) \\ h_3 &= \max(0, w_3 * x + b_3) \\ y &= u_1 * h_1 + u_2 * h_2 + u_3 * h_3 + p \end{aligned}$$

$$\begin{aligned} y &= u_1 * \max(0, w_1 * x + b_1) \\ &\quad + u_2 * \max(0, w_2 * x + b_2) \\ &\quad + u_3 * \max(0, w_3 * x + b_3) \\ &\quad + p \end{aligned}$$



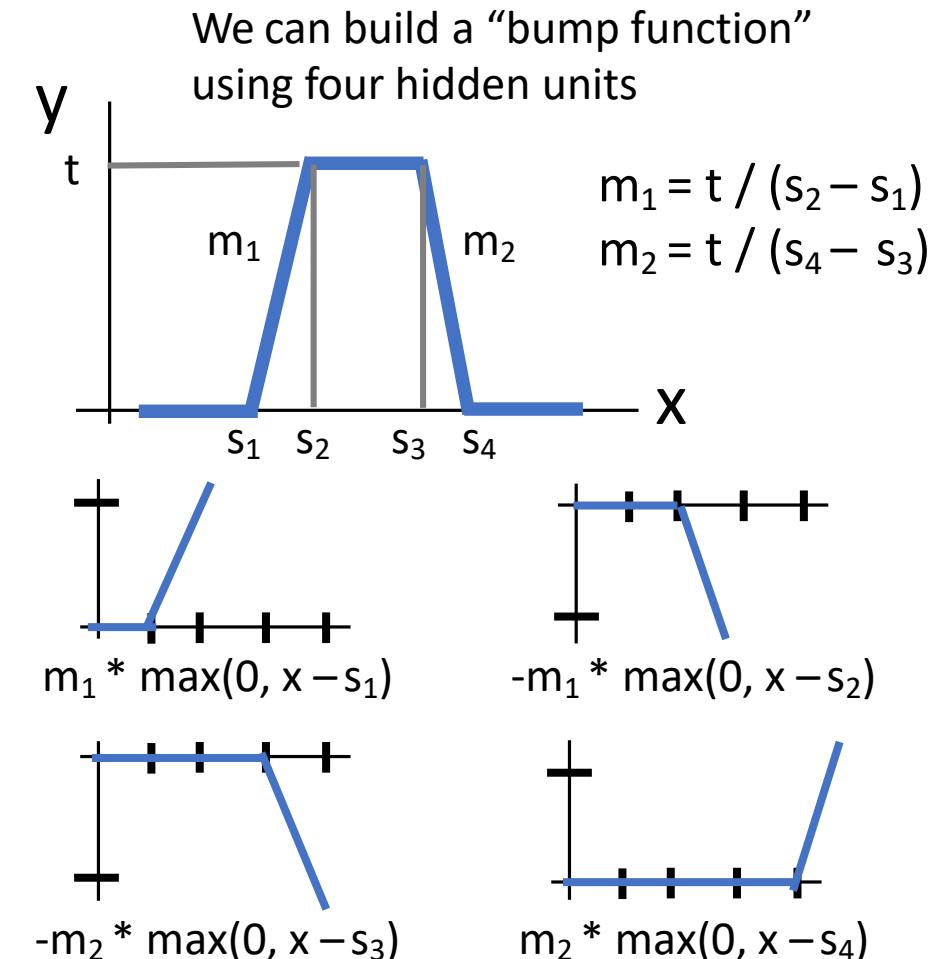
# Universal Approximation

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



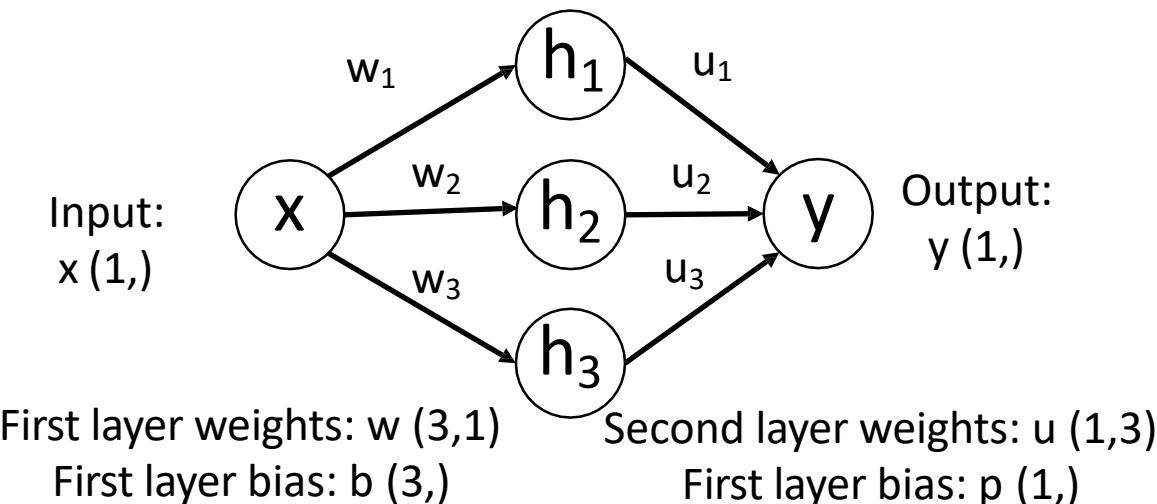
$$\begin{aligned} h_1 &= \max(0, w_1 * x + b_1) \\ h_2 &= \max(0, w_2 * x + b_2) \\ h_3 &= \max(0, w_3 * x + b_3) \\ y &= u_1 * h_1 + u_2 * h_2 + u_3 * h_3 + p \end{aligned}$$

$$\begin{aligned} y &= u_1 * \max(0, w_1 * x + b_1) \\ &\quad + u_2 * \max(0, w_2 * x + b_2) \\ &\quad + u_3 * \max(0, w_3 * x + b_3) \\ &\quad + p \end{aligned}$$



# Universal Approximation

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



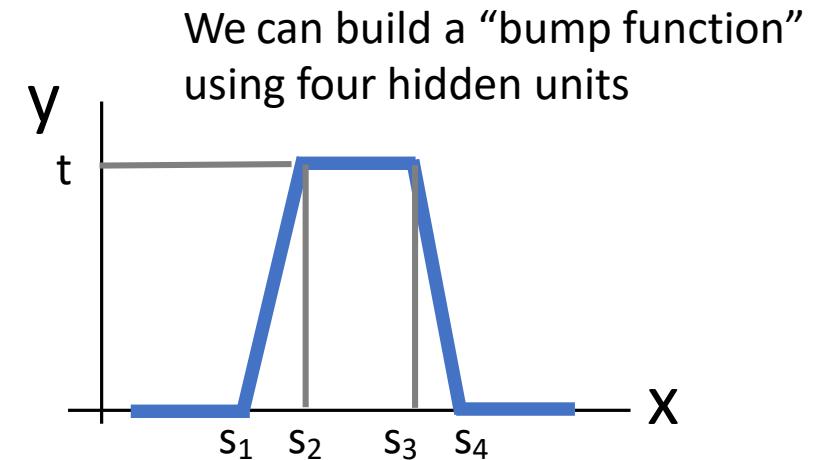
$$h_1 = \max(0, w_1 * x + b_1)$$

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

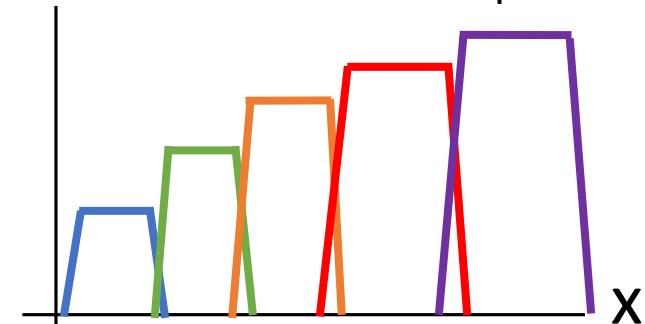
$$h_3 = \max(0, w_3 * x + b_3)$$

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

$$\begin{aligned} y = & u_1 * \max(0, w_1 * x + b_1) \\ & + u_2 * \max(0, w_2 * x + b_2) \\ & + u_3 * \max(0, w_3 * x + b_3) \\ & + p \end{aligned}$$

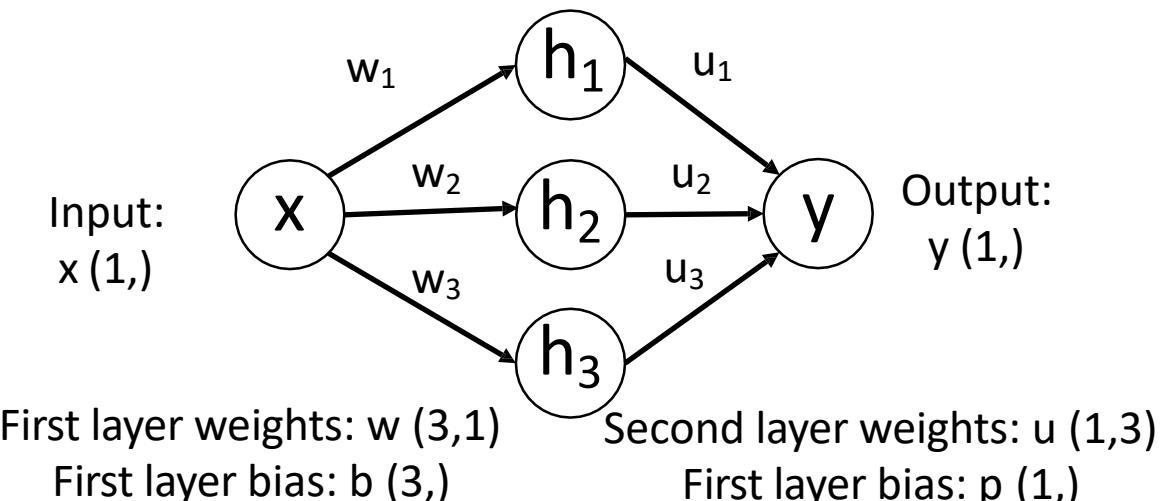


With 4K hidden units we can build a sum of K bumps



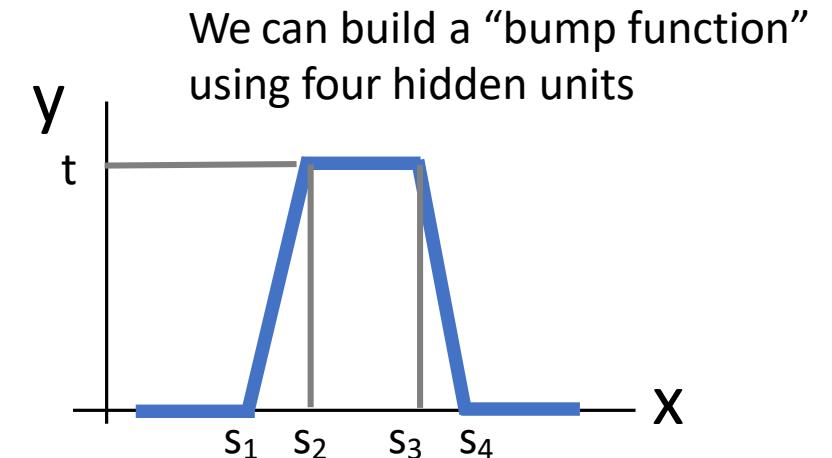
# Universal Approximation

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

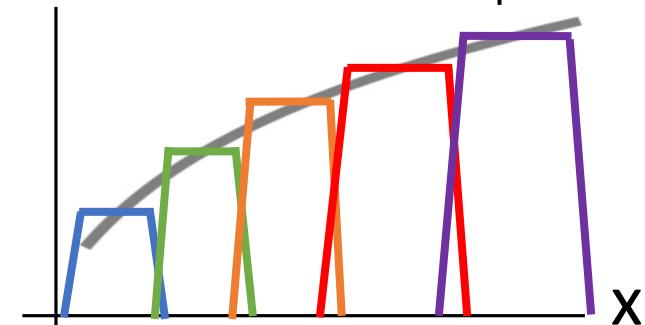


$$\begin{aligned} h_1 &= \max(0, w_1 * x + b_1) \\ h_2 &= \max(0, w_2 * x + b_2) \\ h_3 &= \max(0, w_3 * x + b_3) \\ y &= u_1 * h_1 + u_2 * h_2 + u_3 * h_3 + p \end{aligned}$$

$$\begin{aligned} y &= u_1 * \max(0, w_1 * x + b_1) \\ &\quad + u_2 * \max(0, w_2 * x + b_2) \\ &\quad + u_3 * \max(0, w_3 * x + b_3) \\ &\quad + p \end{aligned}$$



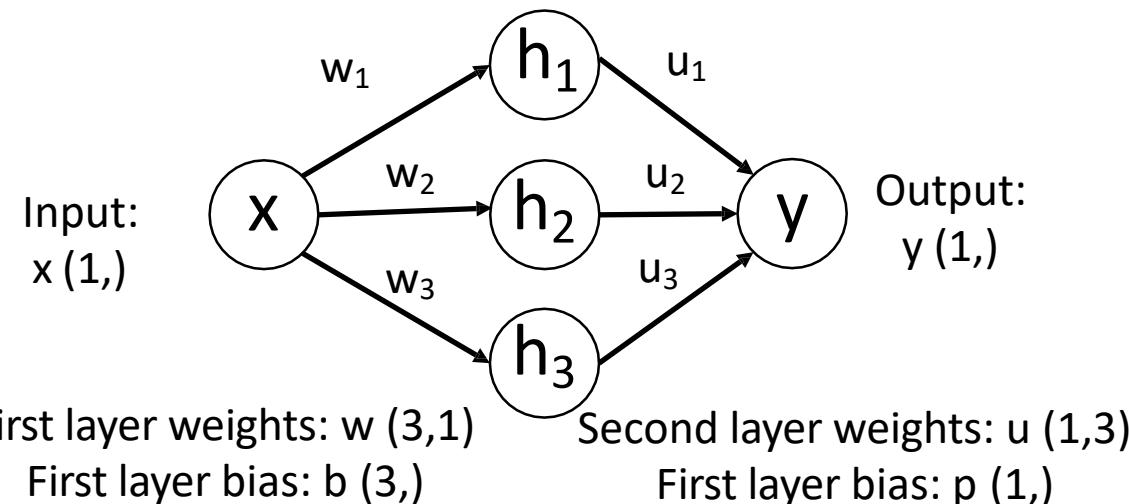
With 4K hidden units we can build a sum of K bumps



Approximate functions with bumps!

# Universal Approximation

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



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

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

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

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

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

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

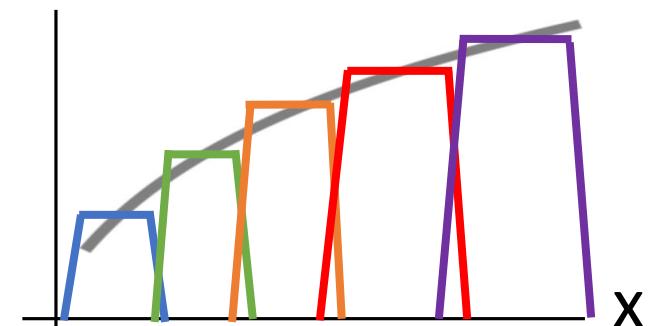
$$+ u_3 * \max(0, w_3 * x + b_3)$$

$$+ p$$

What about...

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

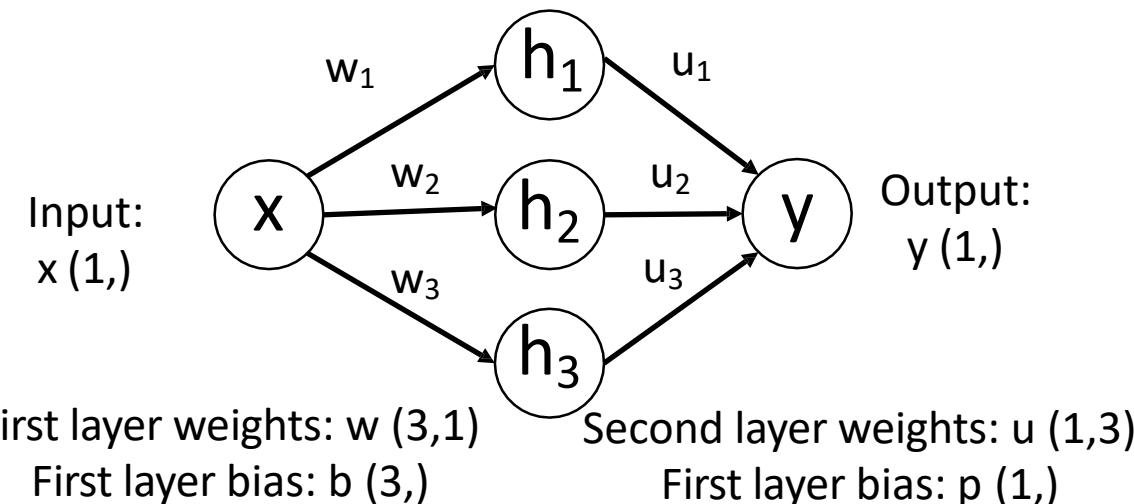
See [Nielsen, Chapter 4](#)



Approximate functions with bumps!

# Universal Approximation

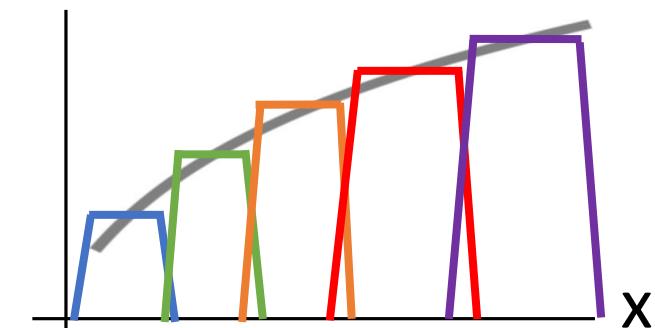
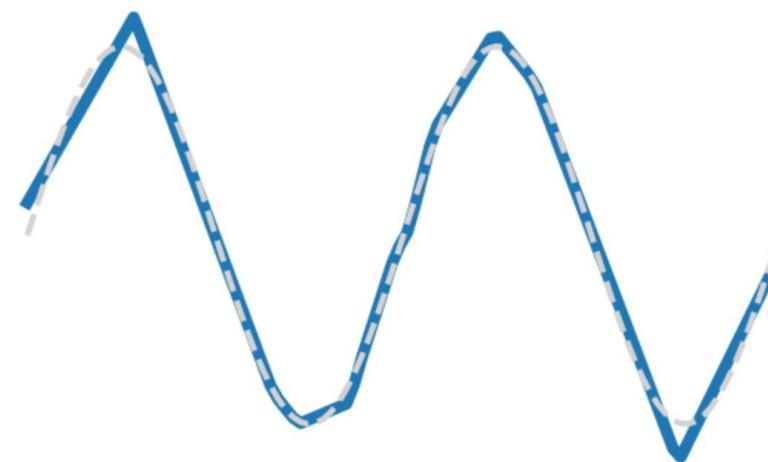
Example: Approximating a function  $f: \mathbb{R} \rightarrow \mathbb{R}$  with a two-layer ReLU network



$$\begin{aligned} h_1 &= \max(0, w_1 * x + b_1) \\ h_2 &= \max(0, w_2 * x + b_2) \\ h_3 &= \max(0, w_3 * x + b_3) \\ y &= u_1 * h_1 + u_2 * h_2 + u_3 * h_3 + p \end{aligned}$$

$$\begin{aligned} y &= u_1 * \max(0, w_1 * x + b_1) \\ &\quad + u_2 * \max(0, w_2 * x + b_2) \\ &\quad + u_3 * \max(0, w_3 * x + b_3) \\ &\quad + p \end{aligned}$$

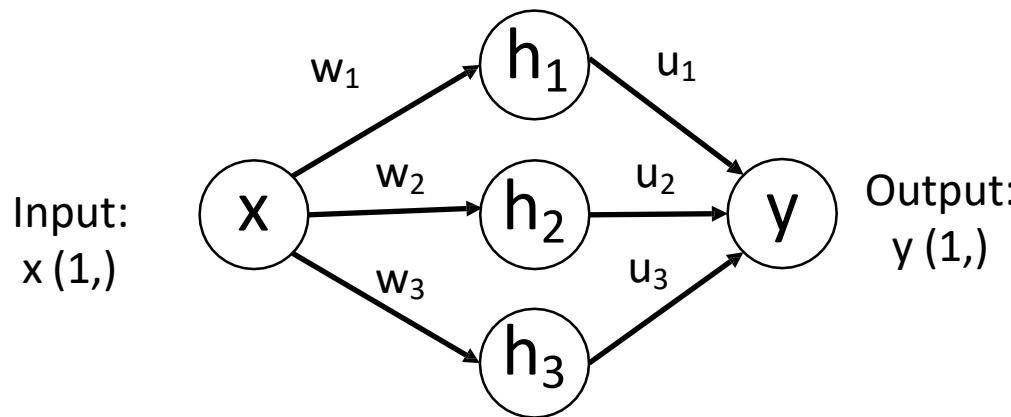
Reality check: Networks don't really learn bumps!



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:

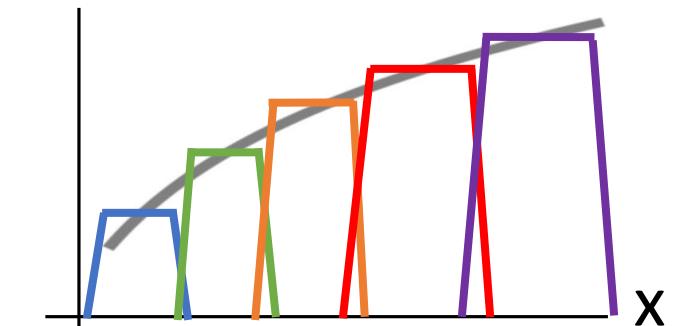
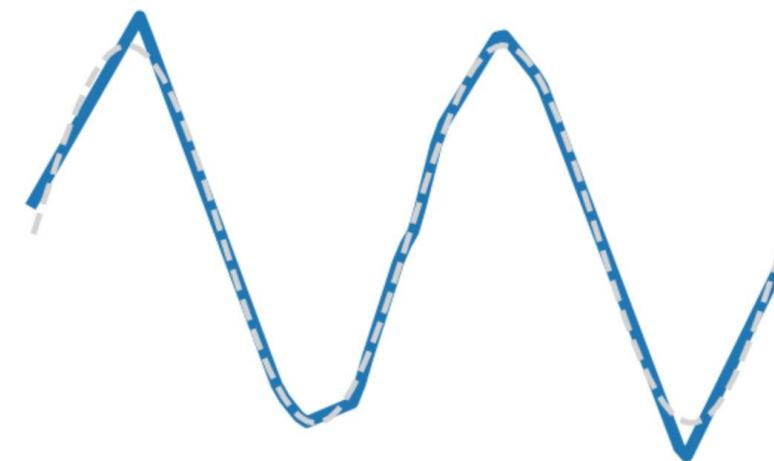
- Neural nets can represent any function

Universal approximation DOES NOT tell us:

- 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!



Approximate functions with bumps!

# Convex Functions

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

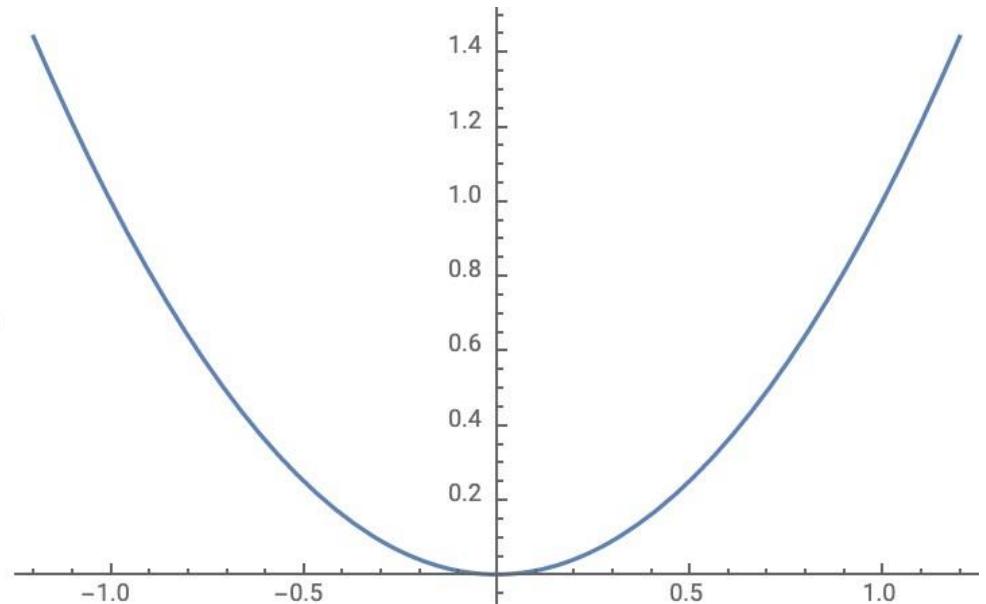
$$f(tx_1 + (1 - t)x_2) \leq tf(x_1) + (1 - t)f(x_2)$$

# 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:

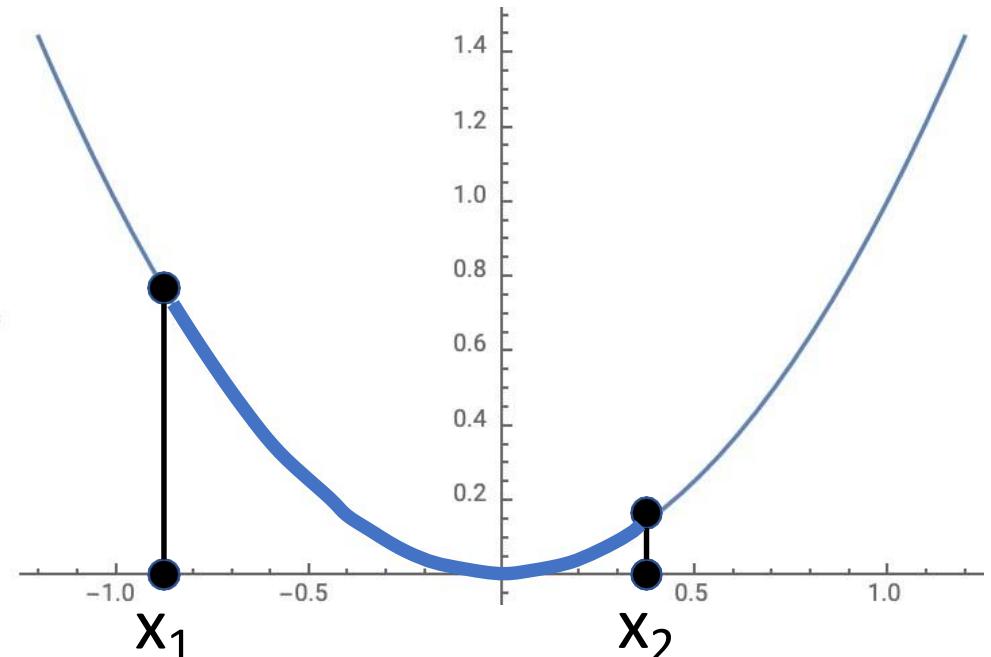


# Convex Functions

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:

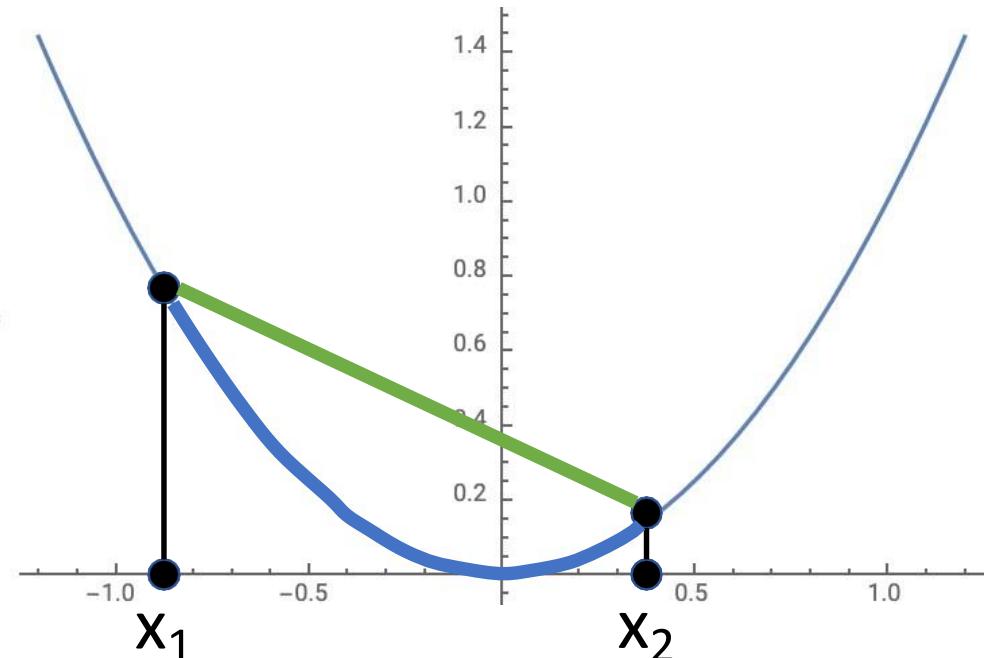


# Convex Functions

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:

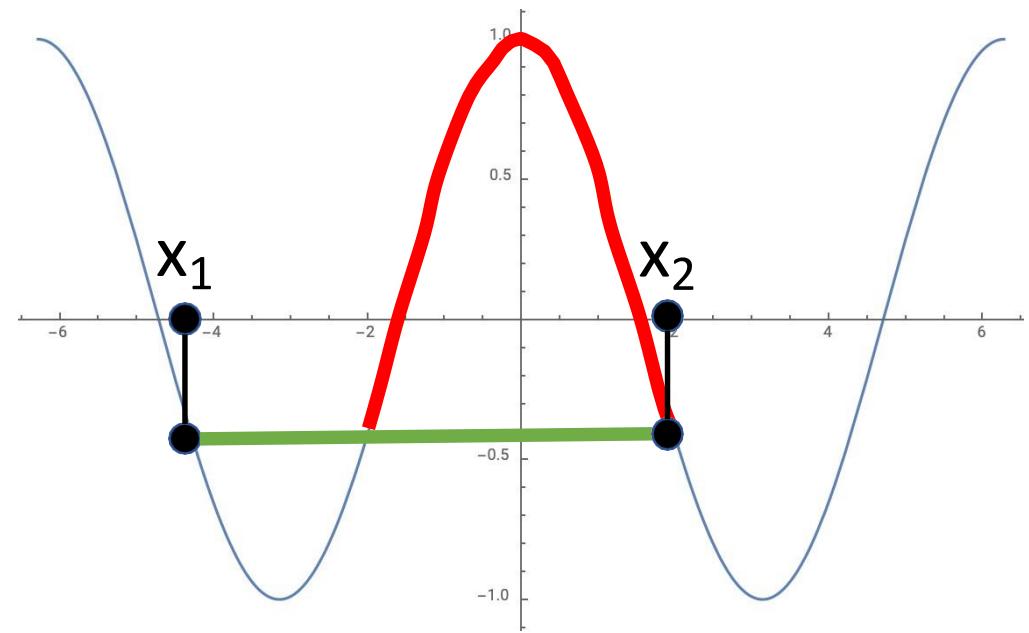


# Convex Functions

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:

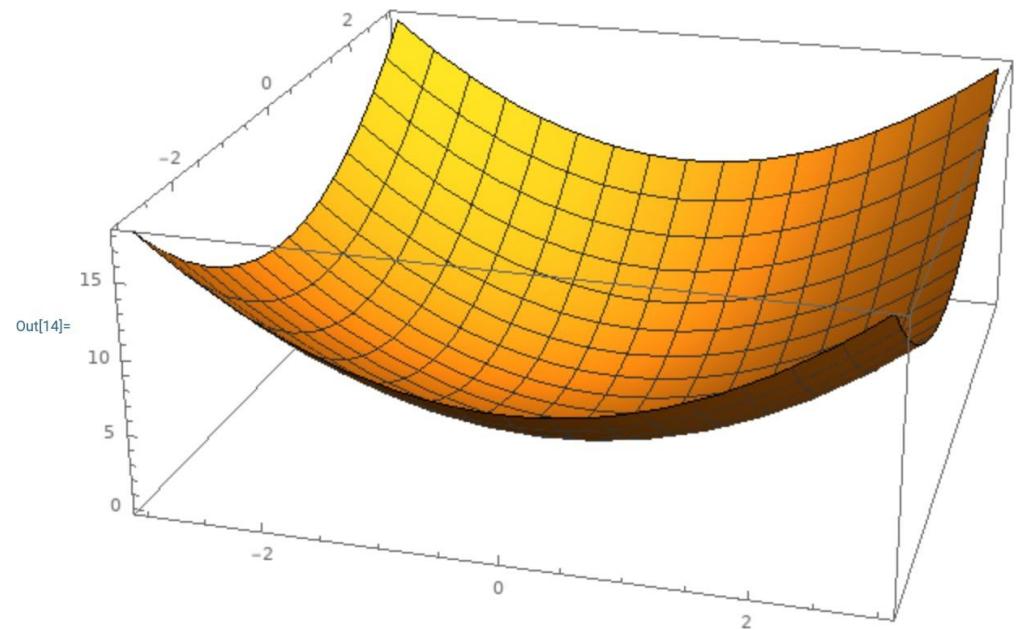


# Convex Functions

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



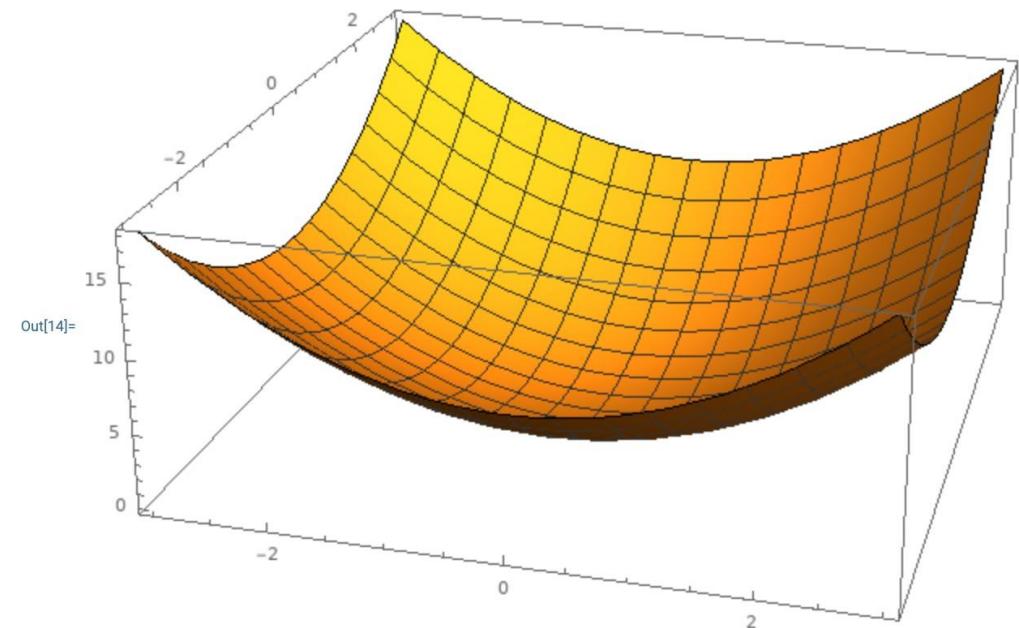
# Convex Functions

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!**

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

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{sj}}\right) \text{ Softmax}$$

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

$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W)$$

R(W) = L2 or L1 regularization

# Convex Functions

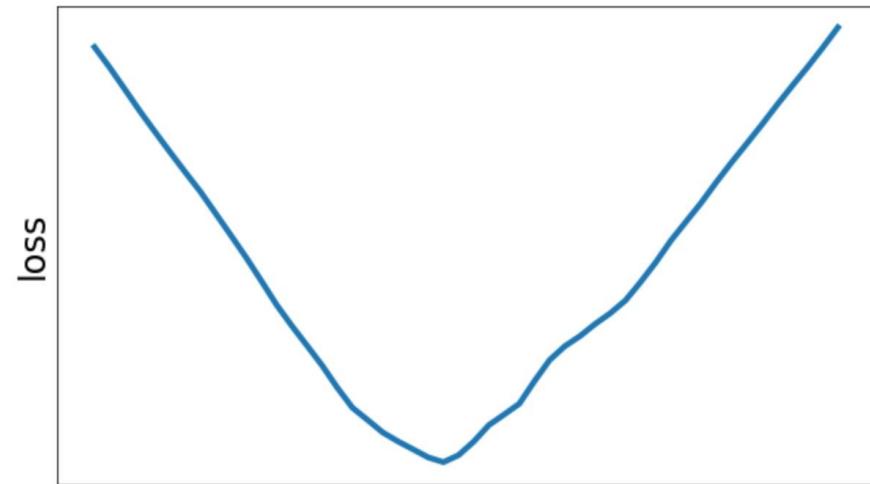
A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

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

**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:



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

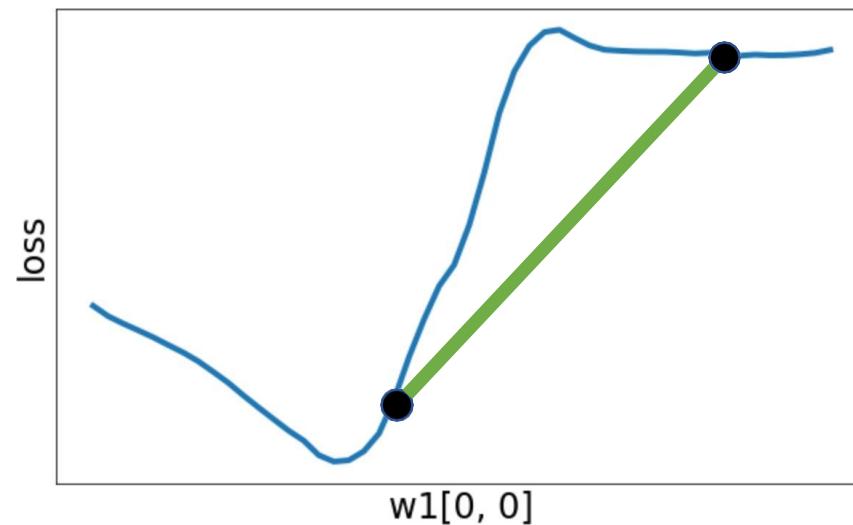
A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

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

**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\***

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

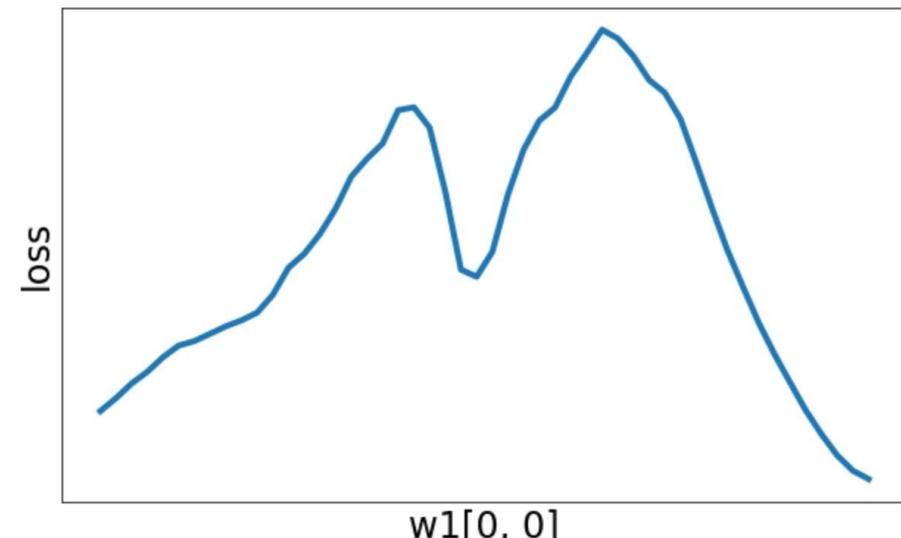
A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

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

**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

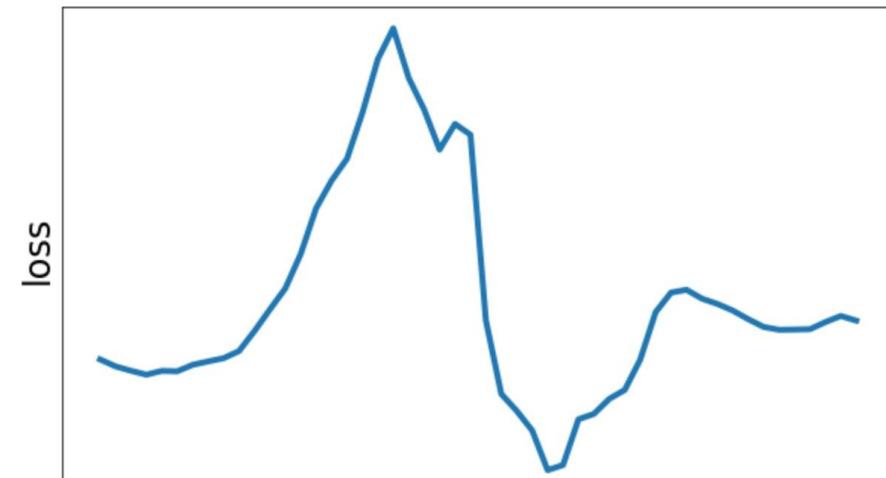
A function  $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$  is **convex** if for all  $x_1, x_2 \in X, t \in [0, 1]$ ,

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

**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

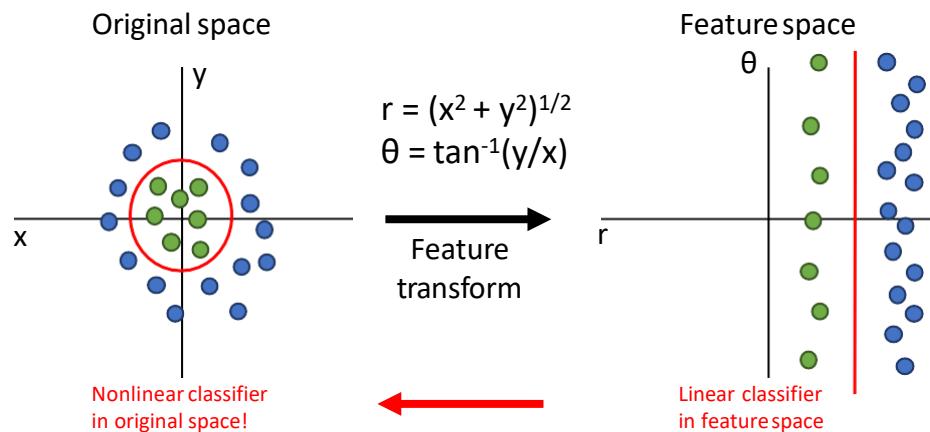
Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum\***

Most neural networks need **nonconvex optimization**

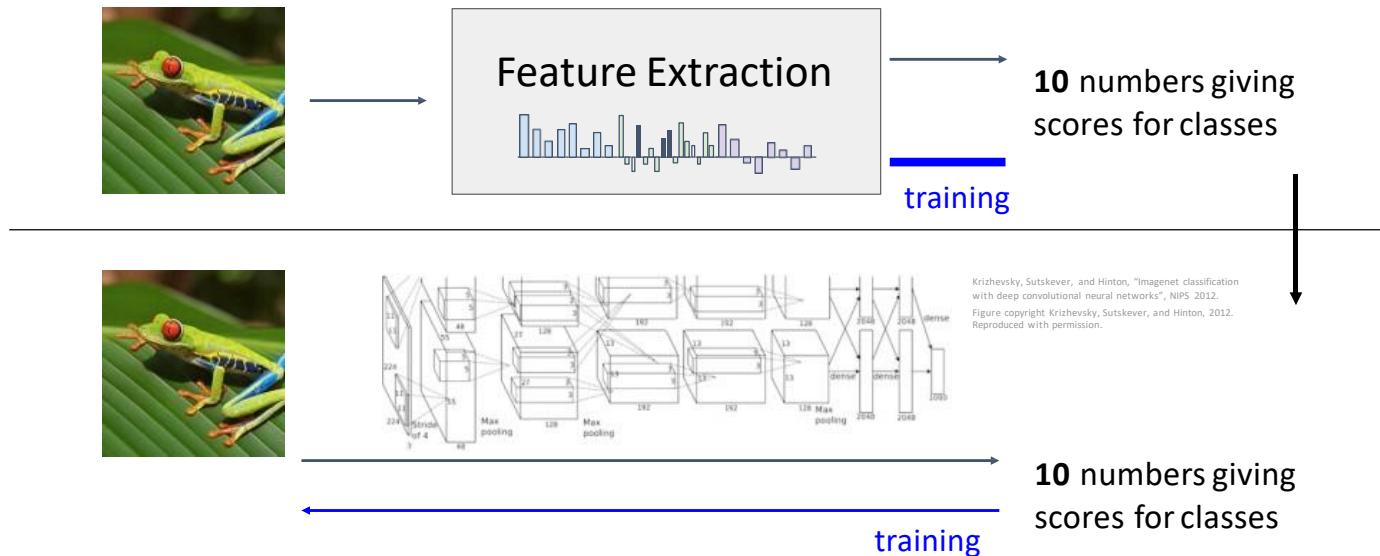
- Few or no guarantees about convergence
- Empirically it seems to work anyway
- Active area of research

# Summary

Feature transform + Linear classifier  
allows nonlinear decision boundaries



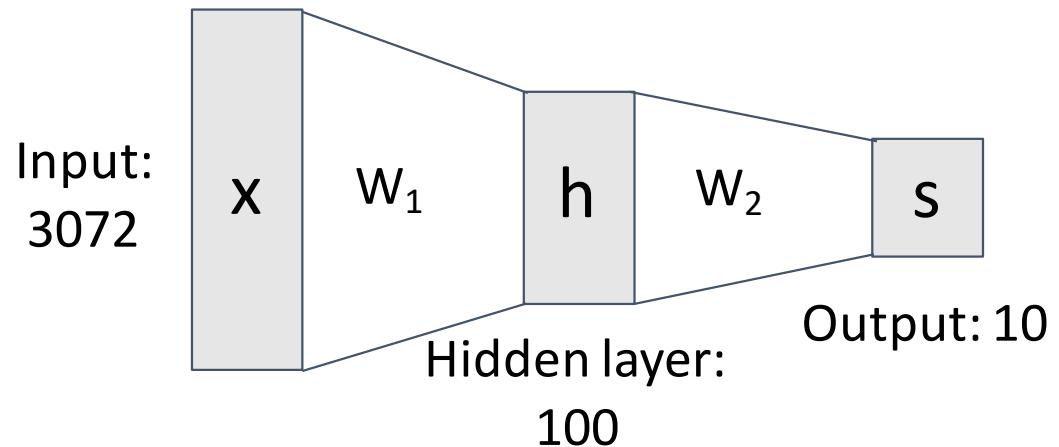
Neural Networks as learnable feature transforms



# 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



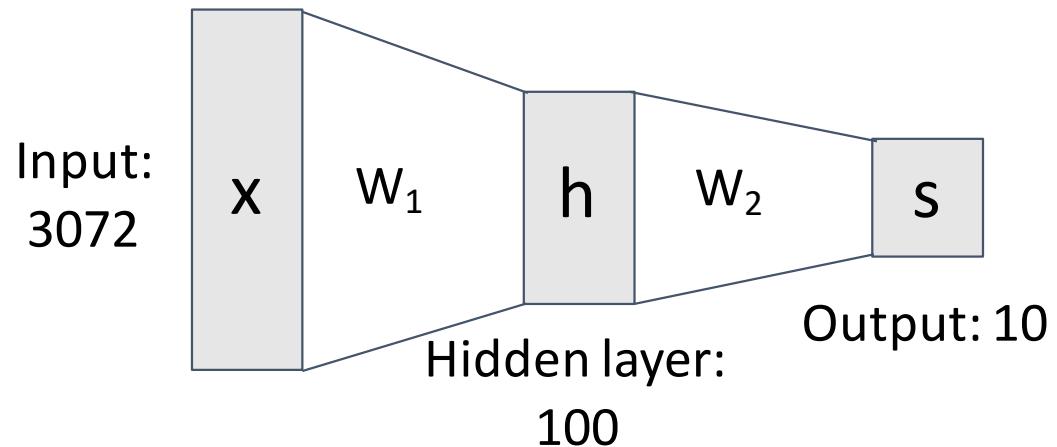
Neural networks: Many reusable templates



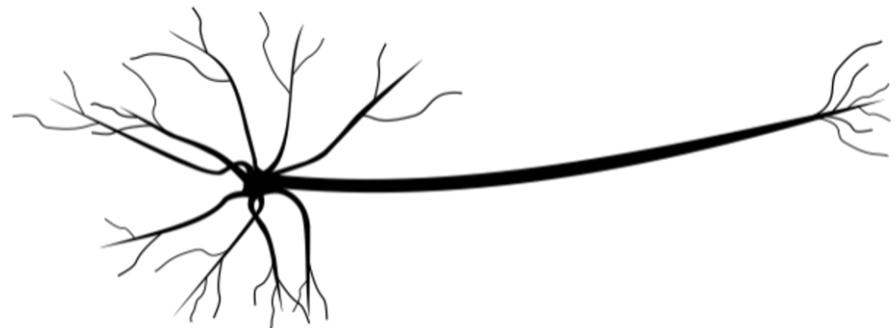
# Summary

From linear classifiers to  
fully-connected networks

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



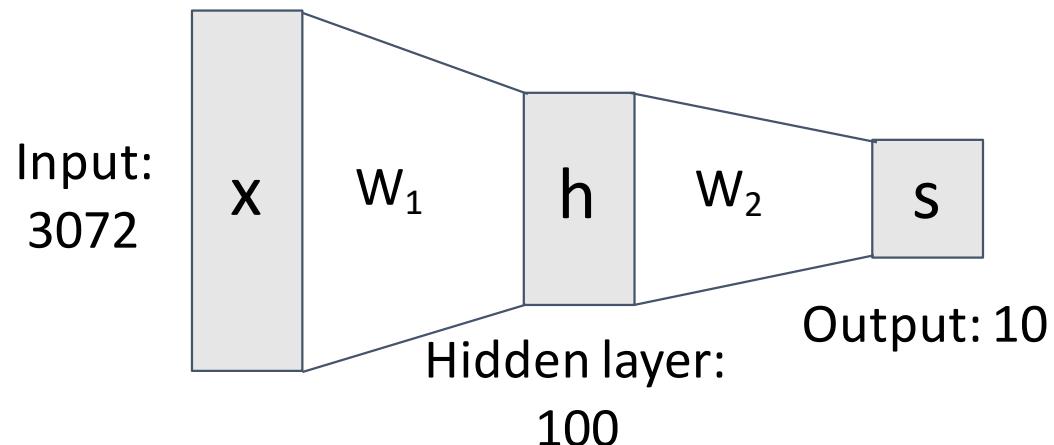
Neural networks loosely inspired by biological neurons but be careful with analogies



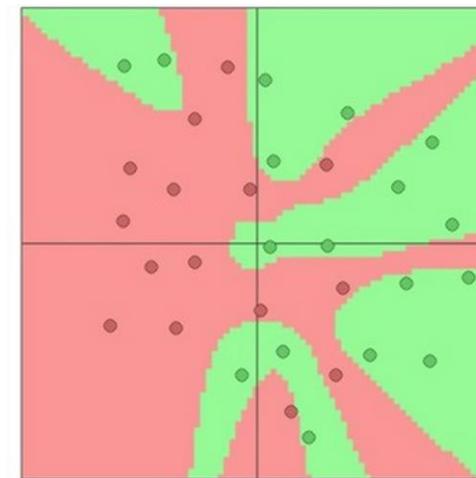
# Summary

From linear classifiers to  
fully-connected networks

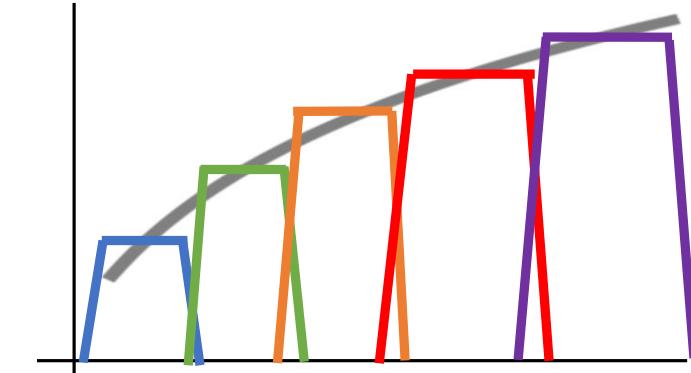
$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$



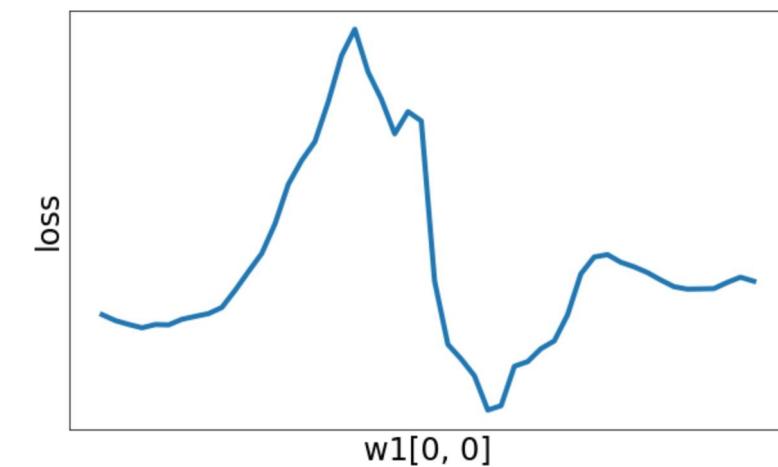
## Space Warping



## Universal Approximation



## Nonconvex



# Problem: How to compute gradients?

$$s = W_2 \max(0, W_1 x + b_1) + b_2 \quad \text{Nonlinear score function}$$

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

$$R(W) = \sum_k W_k^2 \quad \text{L2 Regularization}$$

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

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