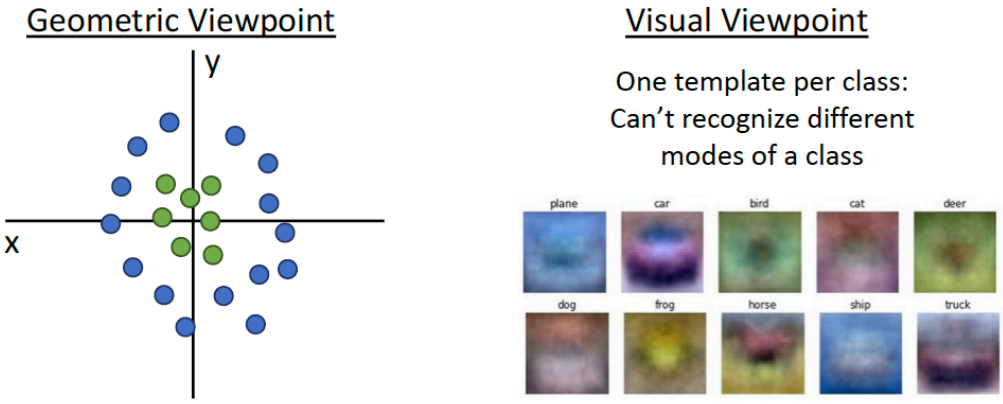
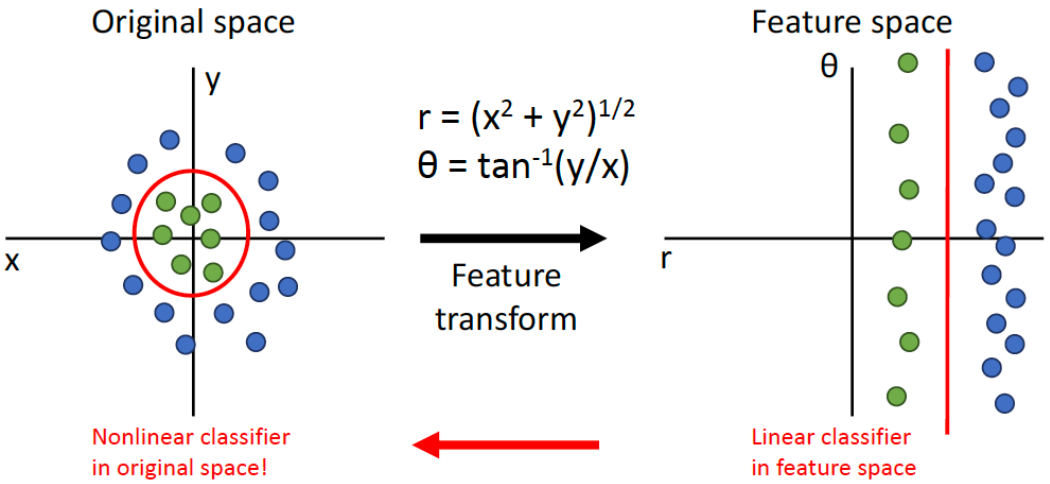
# Lecture 5 - Neural Networks

# 1. Problem and Solution

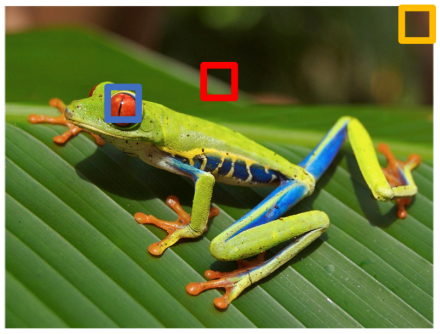
- **Problem:** Linear Classifiers aren't that powerful




- **Solution:**
  - Feature Transforms



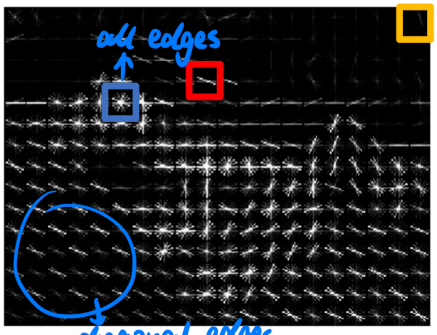

  - Image Features
    - Histogram of Oriented Gradients (HOG)



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

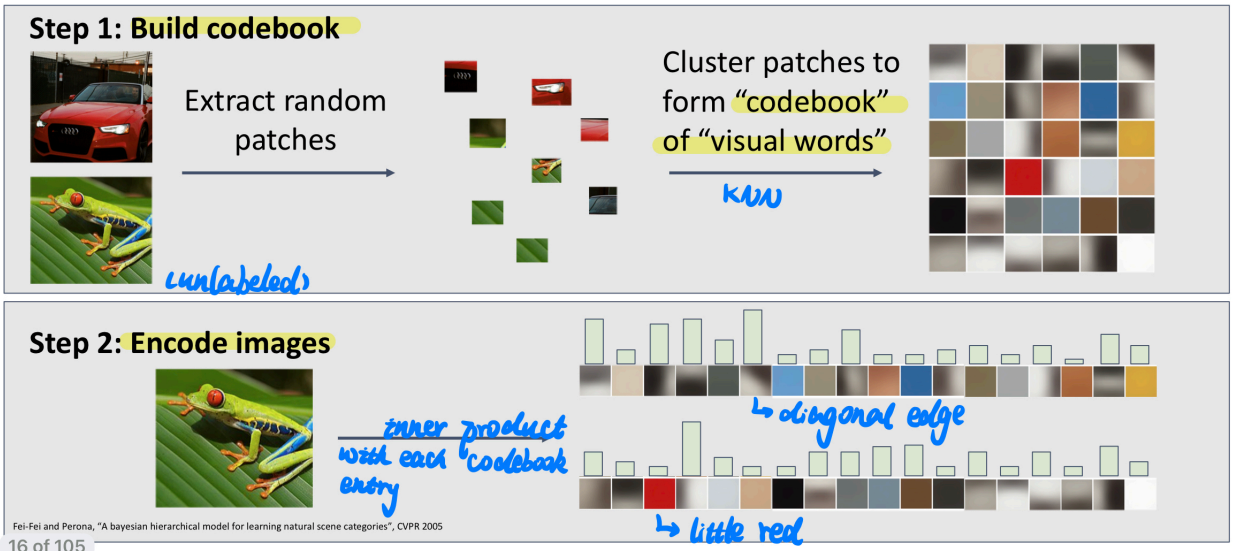
Captures texture and position, robust to small image changes

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

8x8

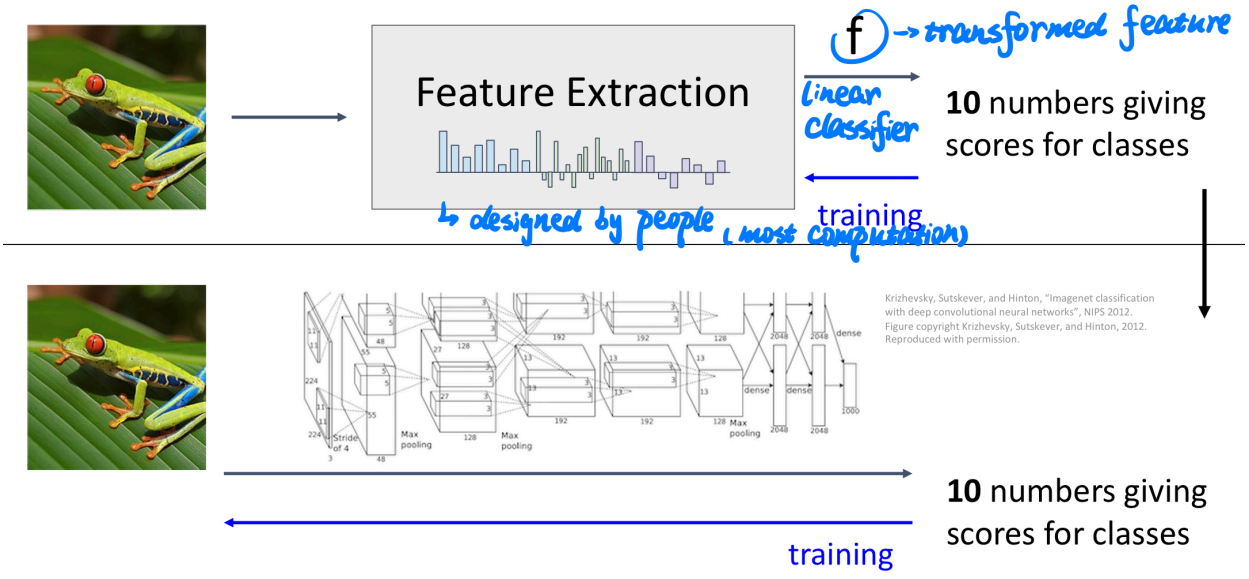
Lowe, "Object recognition from local scale-invariant features", ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

- Bag of Words (Data-Driven!)






- Image Features vs Neural Networks



# 2. Neural Networks

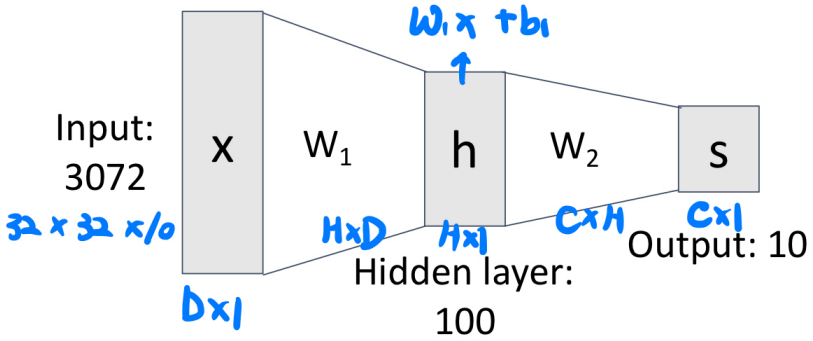
**Input**: $x \in \mathbb{R}^D$ **Output**: $f(x) \in \mathbb{R}\text{^}C$

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

**Now:** Two-Layer Neural Network: $f(x) = W_2 \max(0, W_1 x + b_1) + b2$
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:
$f(x) = W_3 \max(0, W_2 \max(0, W_1 x + b_1) + b_2) + b_3$

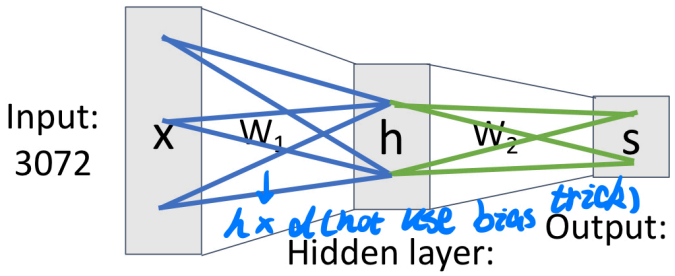
--> $max(0, W_1 x + b_1)$ and $max(0, W_2\ max(0, W_1 x + b_1) + b2)$ Are the new features




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

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 $h_j$

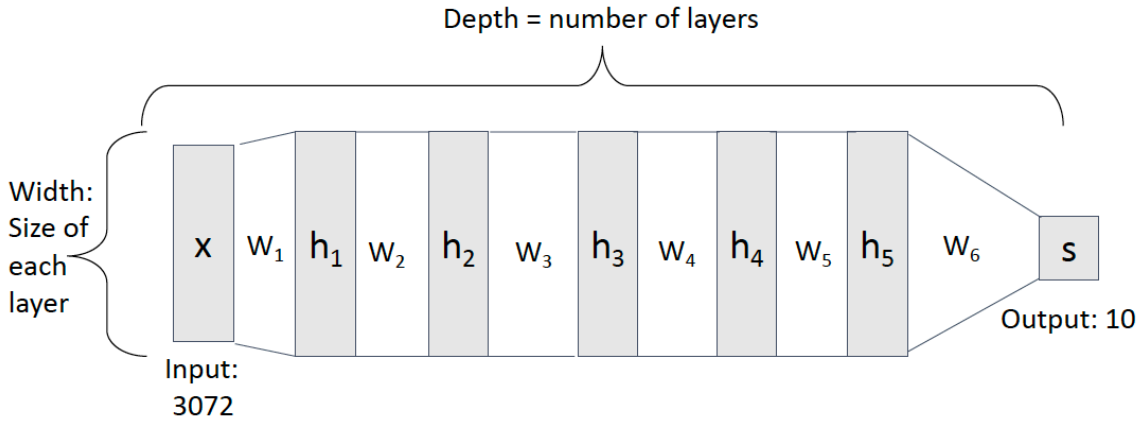
All elements of h affect all elements of s

=> all entries in all layers are connected

Fully-connected neural network

Also "Multi-Layer Perceptron" (MLP)

# Deep Neural Networks




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

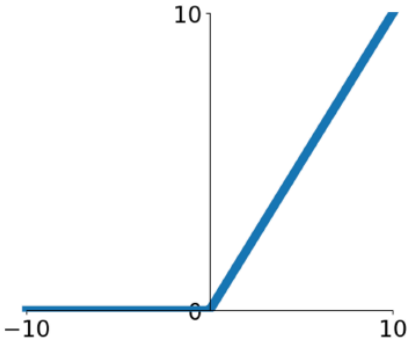
- **Features**
  - Neural net: first layer is bank of templates; second layer recombines templates (activation)
  - Can use **different templates to cover multiple modes** of a class! --> fix multi-modality problem in linear classifier

  - "Distributed representation": Most templates not interpretable! --> cannot interpreted as a single class!
- **Activation Functions**

## 2-layer Neural Network

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

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




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

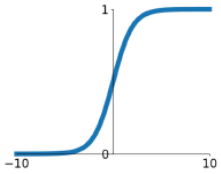
**Q**: What happens if we build a neural network with no activation function?

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

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

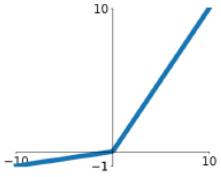
## Sigmoid

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



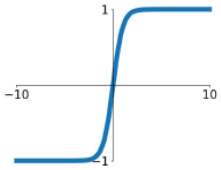

## Leaky ReLU

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




## tanh

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

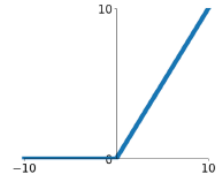



## Softplus

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

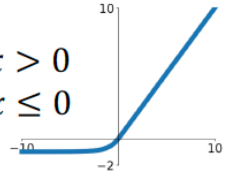
## ReLU

$$\max(0, x)$$



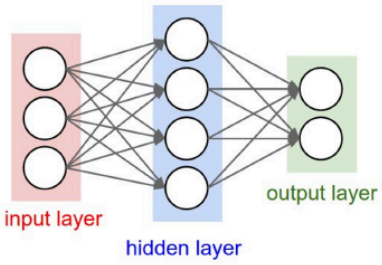

## ELU

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




**RELU is a good default choice for most problems**

## Neural Net in <20 lines!




Initialize weights and data

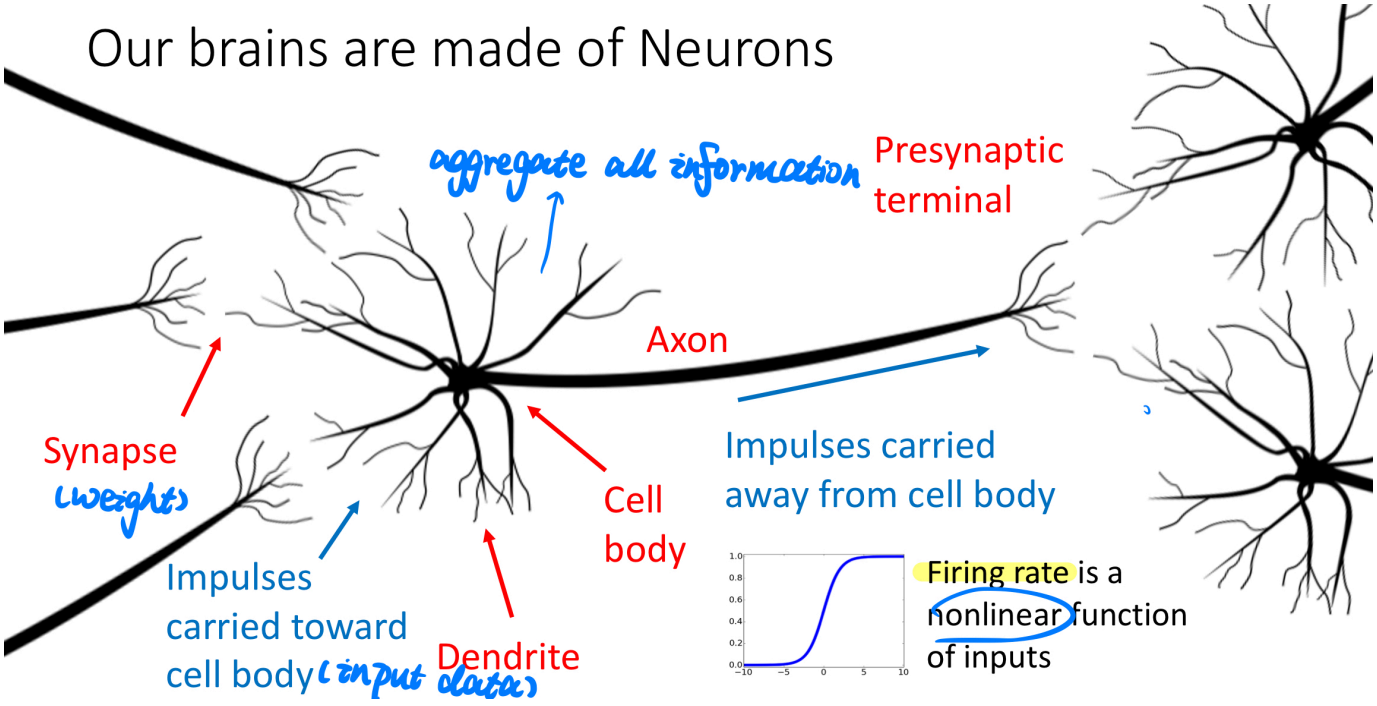
Compute loss (sigmoid activation, L2 loss)

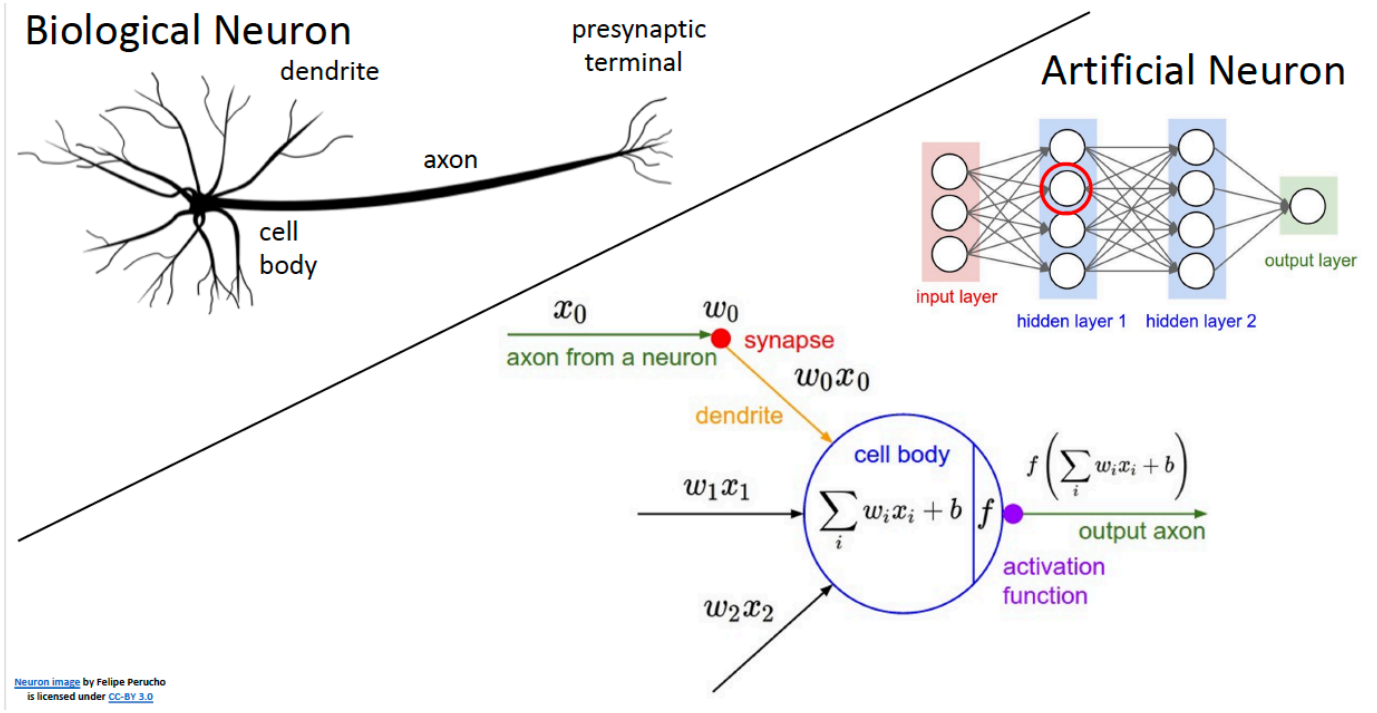
Compute gradients

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
```

- **Brain Analogy**



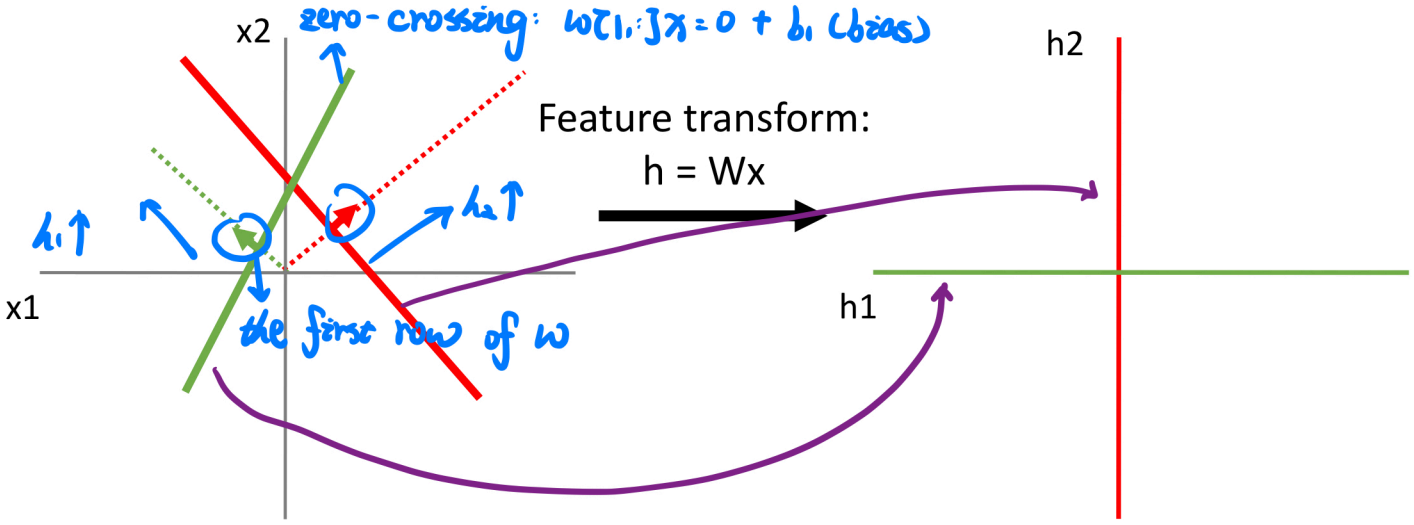

--> each neuron is a siingle scalar entry in the hidden layer

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

## 3. Space Warping

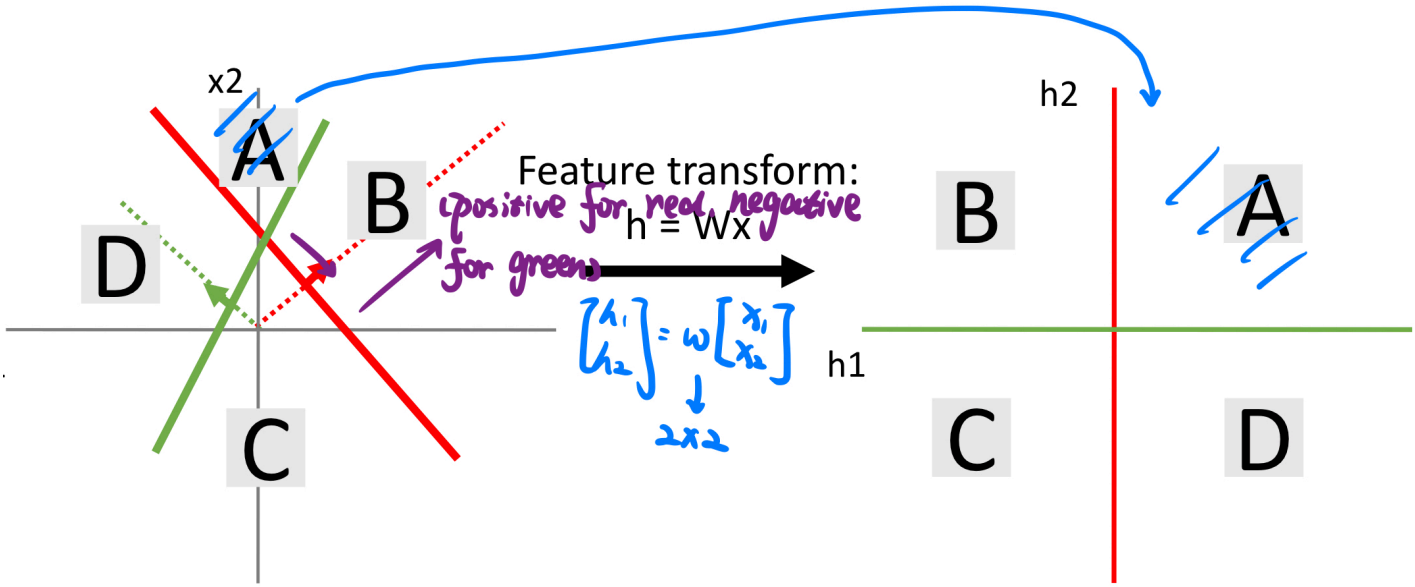
# Space Warping

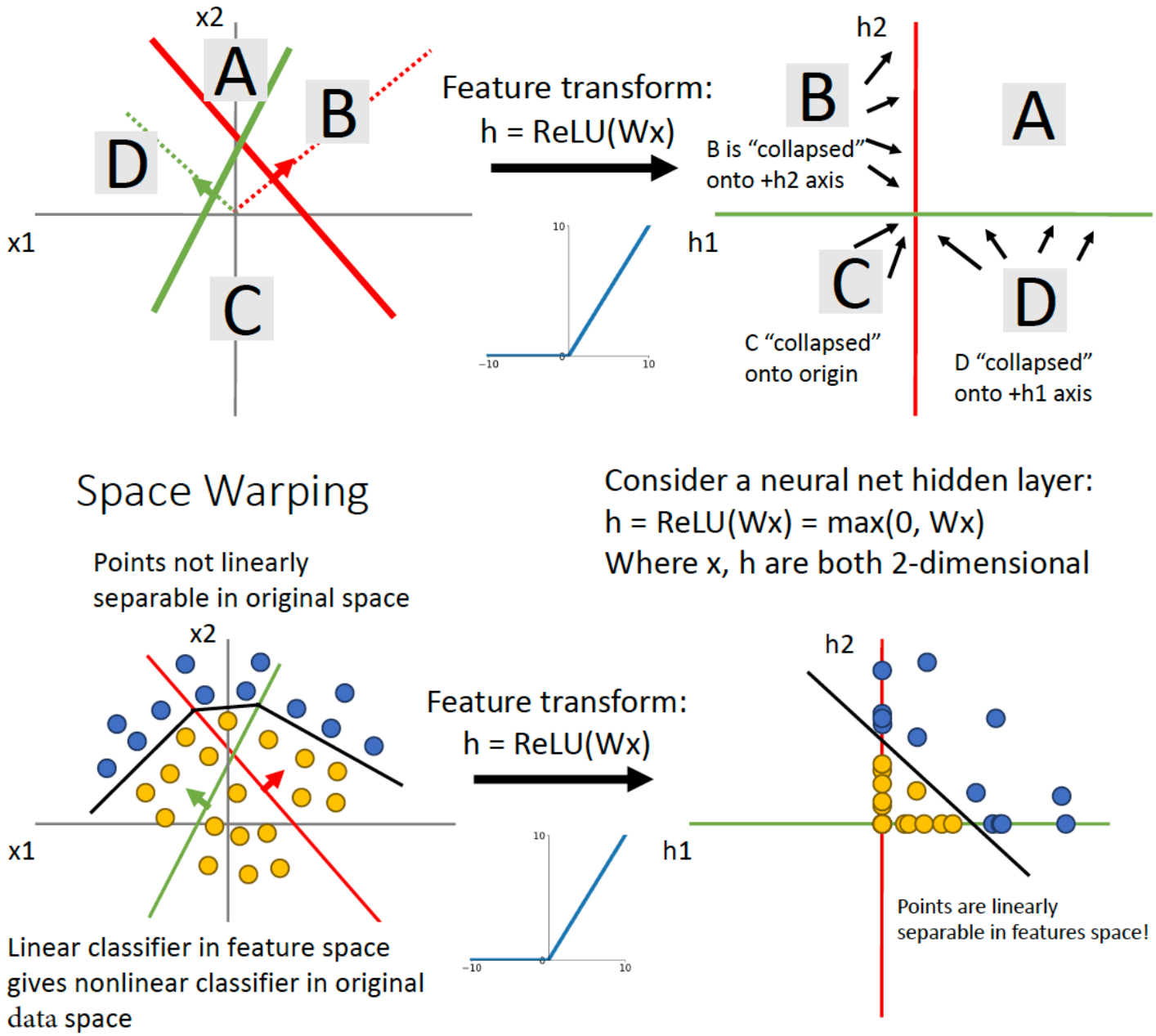
Consider a linear transform: h = Wx
Where x, h are both 2-dimensional




# Space Warping

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

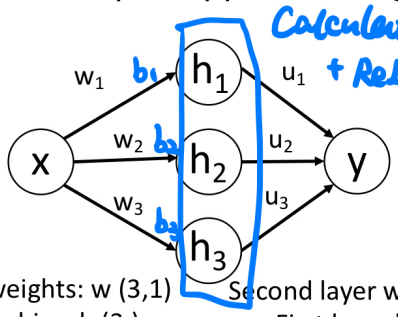





## 4. Universal Approaximation

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

## Example: Approximating a function f: R -> R with a two-layer ReLU network




Calculation + ReLU

single

First layer weights: w (3,1)
First layer bias: b (3,)

Second layer weights: u (1,3)
First layer bias: p (1,)

Output is a sum of shifted, scaled ReLUs:

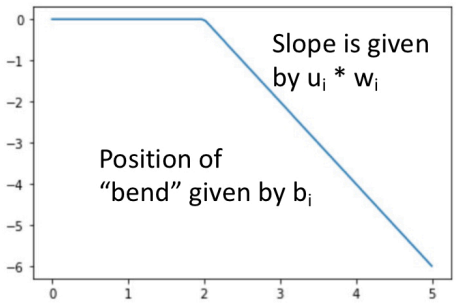
$h1 = \max(0, w1 * x + b1)$
$h2 = \max(0, w2 * x + b2)$
$h3 = \max(0, w3 * x + b3)$
$y = u1 * h1 + u2 * h2 + u3 * h3 + p$

$y = u1 * \max(0, w1 * x + b1)$
$+ u2 * \max(0, w2 * x + b2)$
$+ u3 * \max(0, w3 * x + b3)$
$+ p$

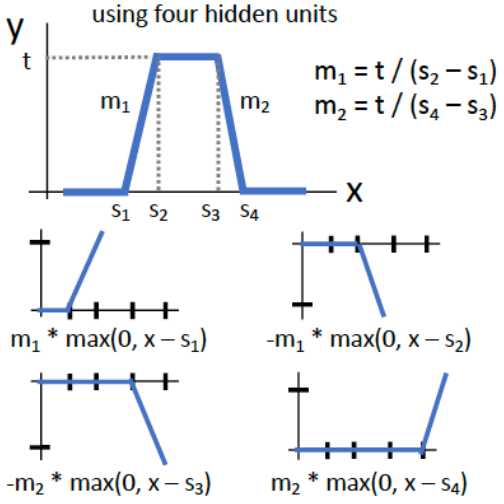
=> 3 shifted + scaled ReLUs

↳ single bias

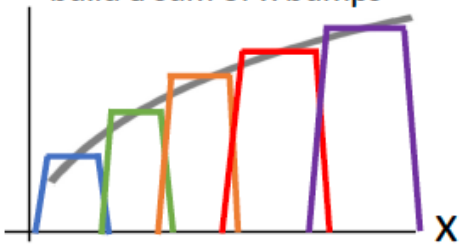
Flip left / right based on sign of $w_i$




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!

---

**Reality check: Networks don't really learn bumps! --> only for mathematical expressions**

Universal approximation tells us:

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