DeepRob

Lecture 5
Neural Networks
University of Minnesota

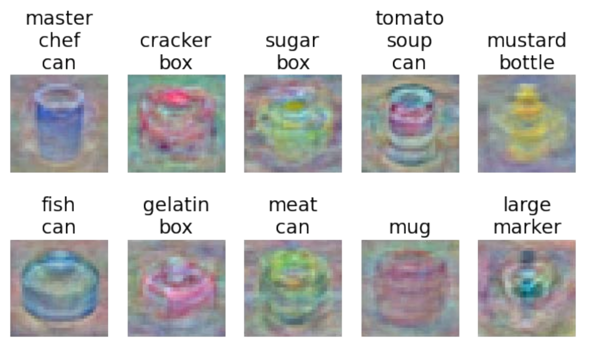
Project 1—Reminder

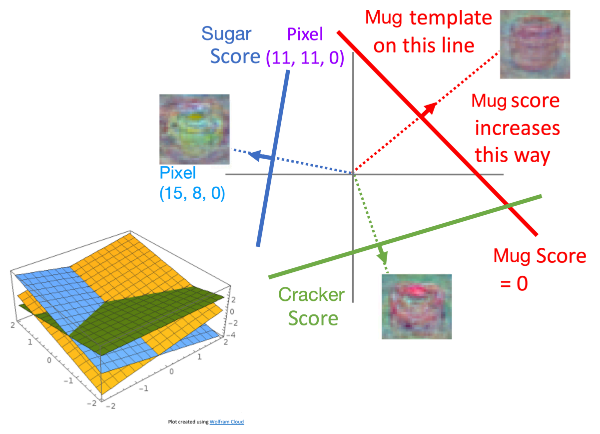
- Instructions and code available on the website
  - Here: https://rpm-lab.github.io/CSCI5980-F24-DeepRob/projects/project1/
- Uses Python, PyTorch and Google Colab
- Implement KNN, linear SVM, and linear softmax classifiers
- **Autograder is available!**
- **Due Monday, Sept 29th 11:59 PM CT**

Recap from Previous Lectures

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

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





$$L_i = -\log\left(\frac{\exp^{s_{y_i}}}{\sum_i \exp^{s_j}}\right)$$ **Softmax**

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

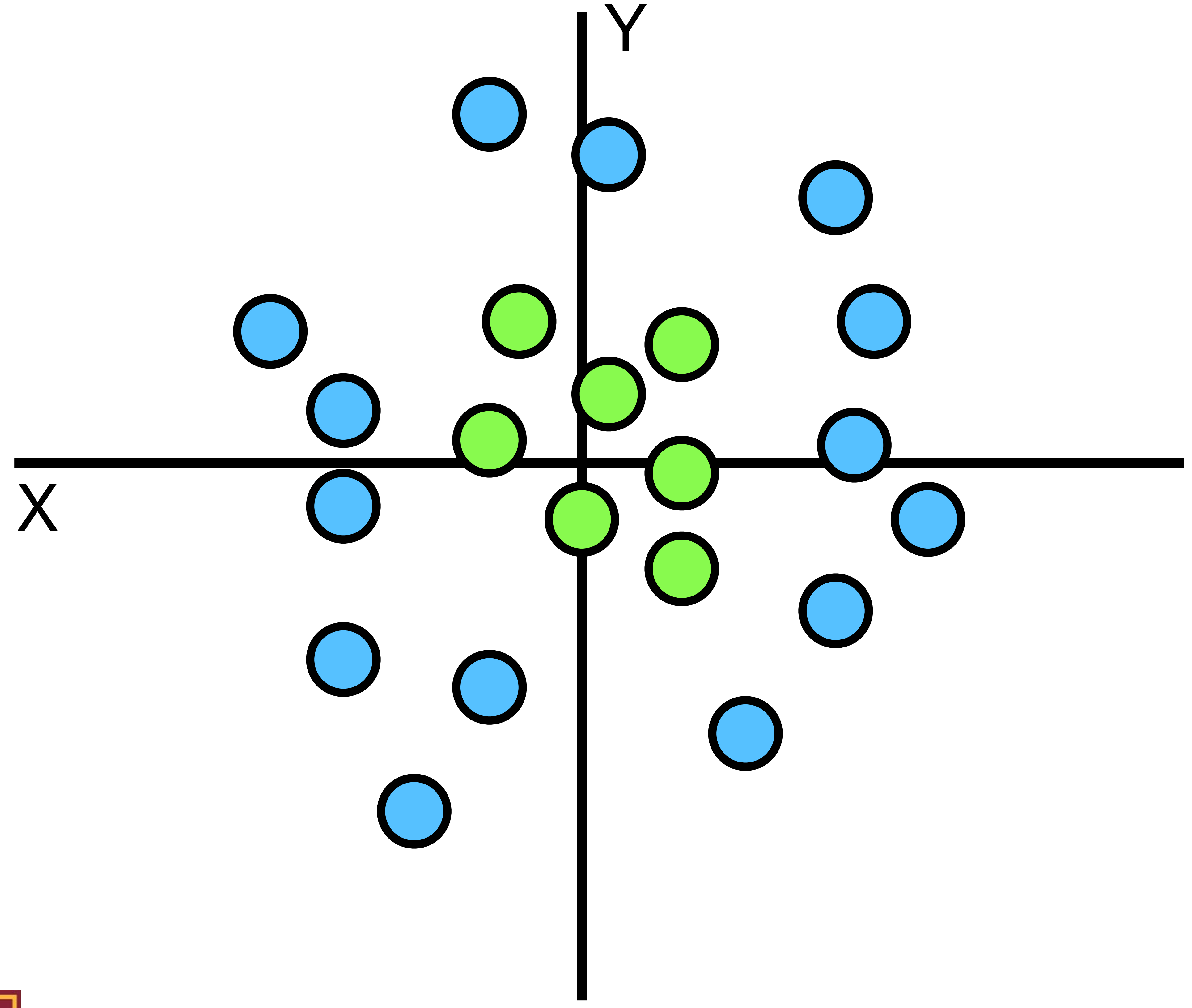
$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$

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

Neural Networks

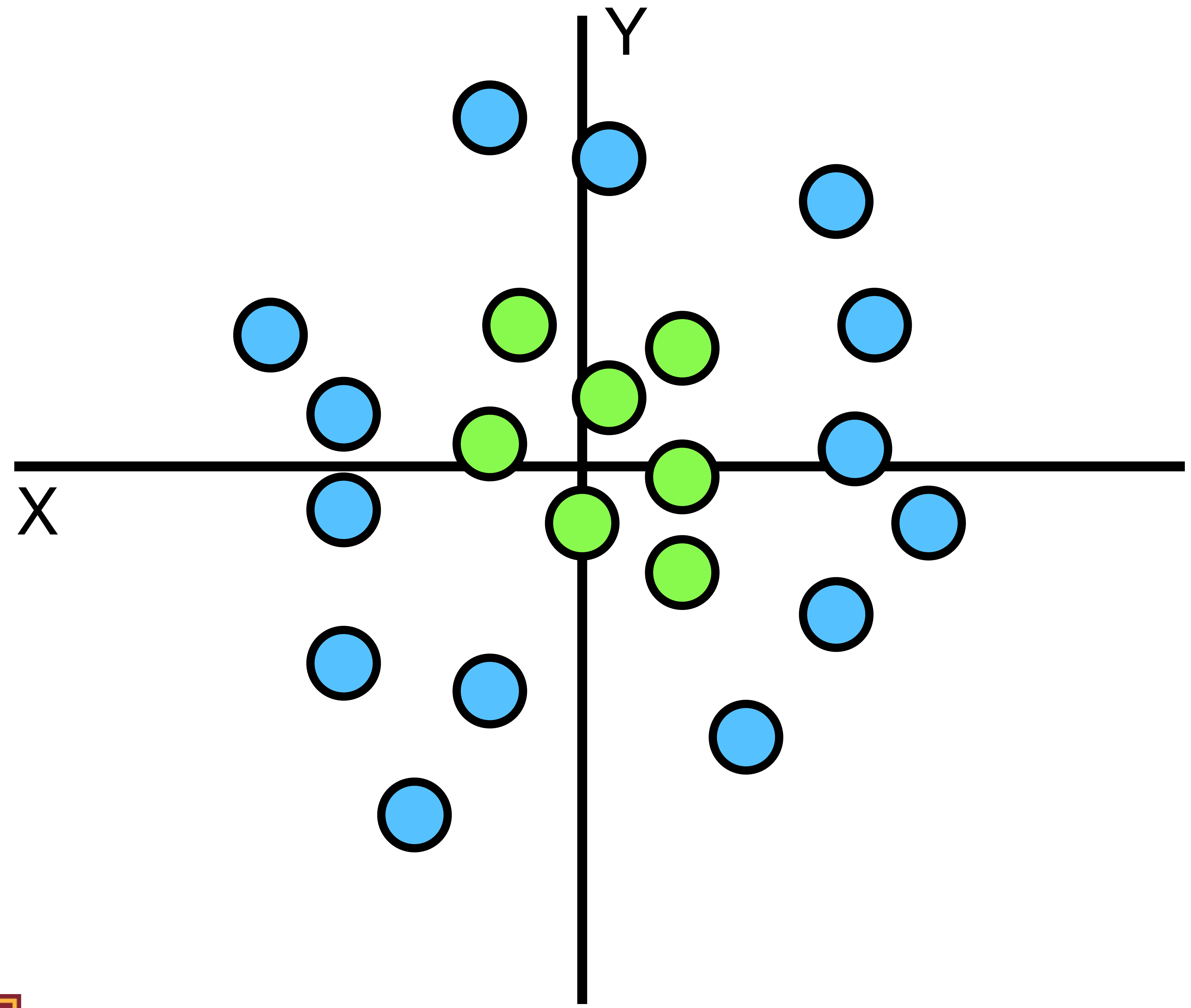
Problem: Linear Classifiers aren't that powerful

Geometric Viewpoint



Problem: Linear Classifiers aren't that powerful

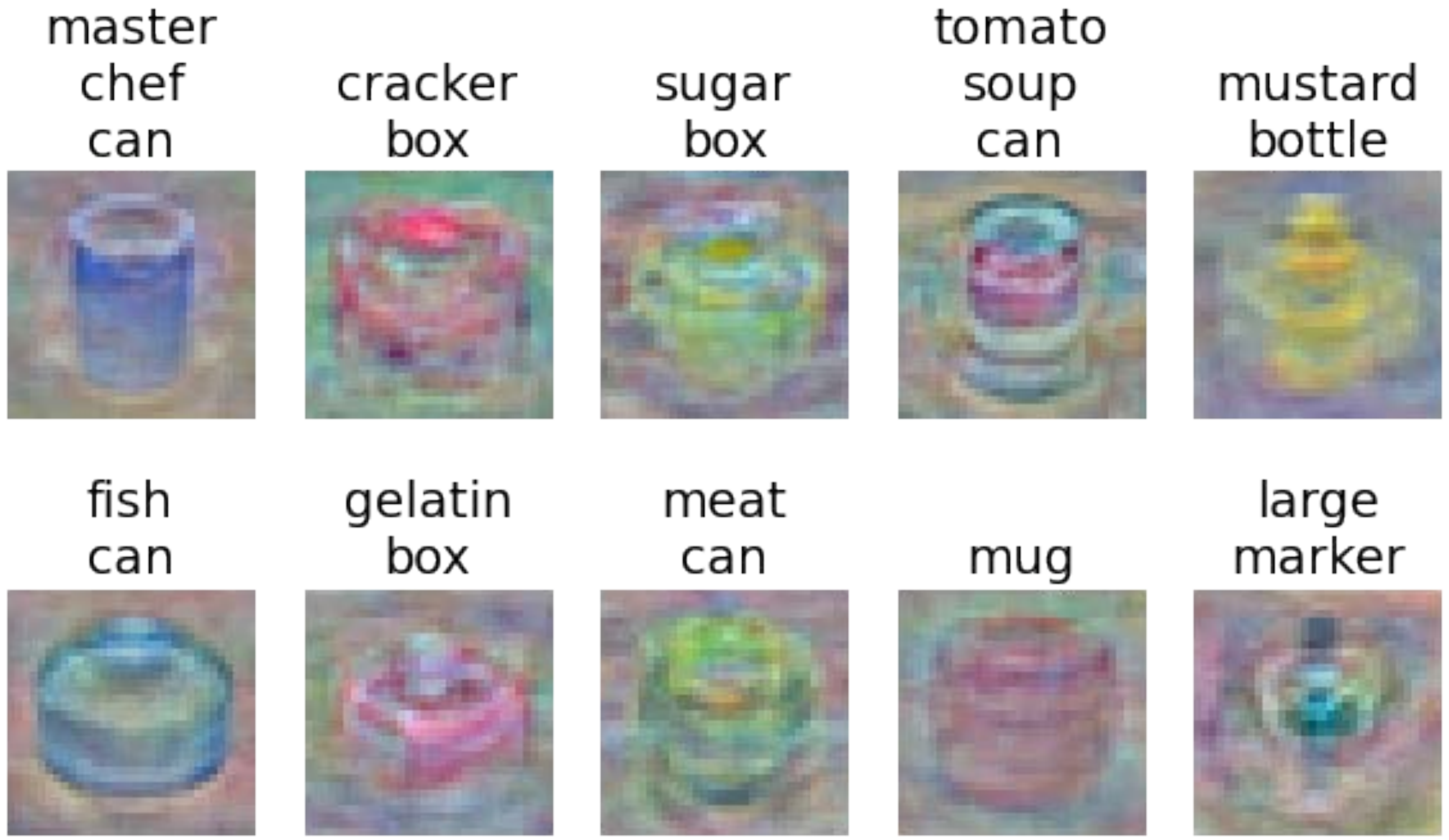
Geometric Viewpoint



Visual Viewpoint

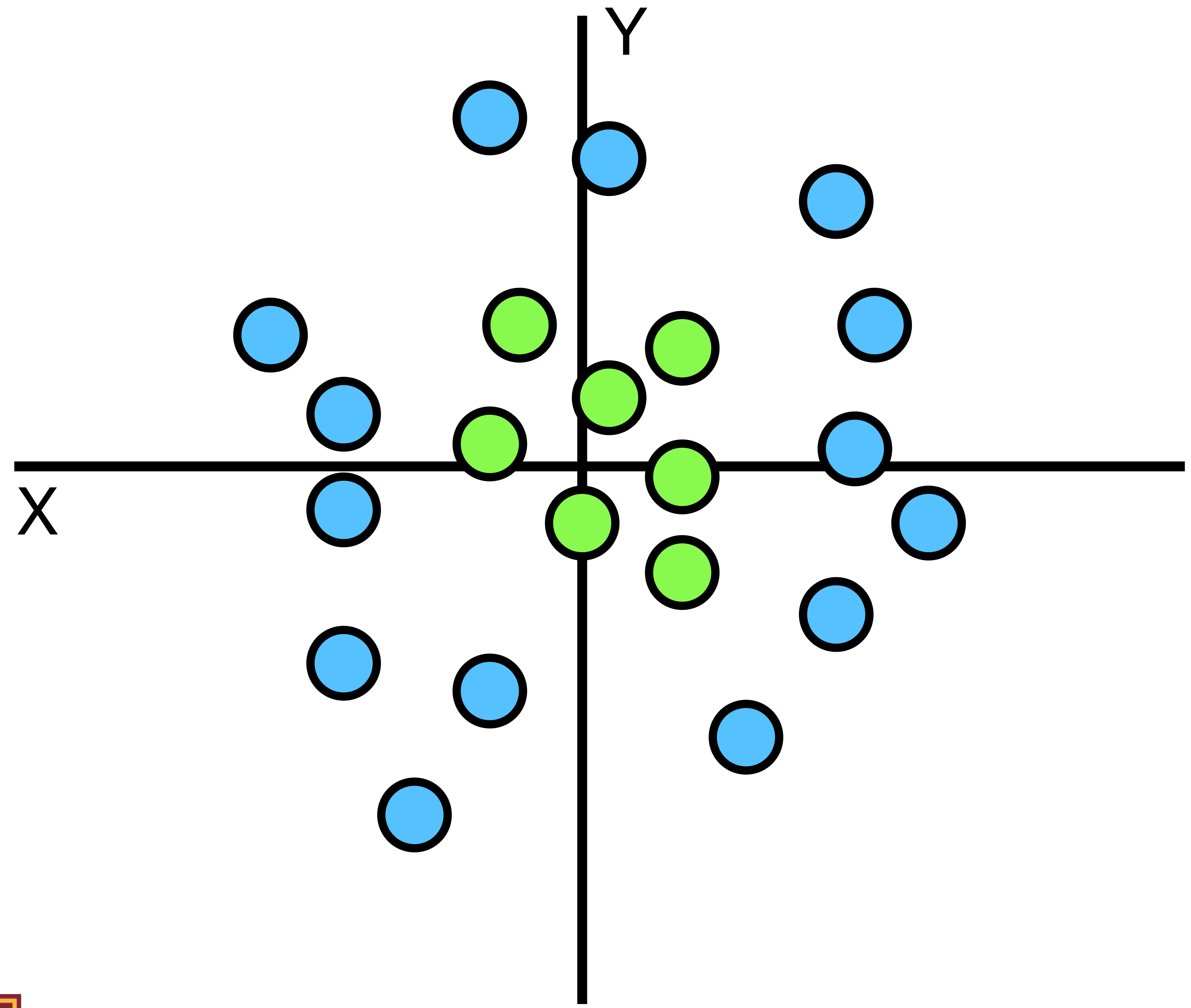
One template per class:

Can't recognize different modes of a class



One solution: Feature Transforms

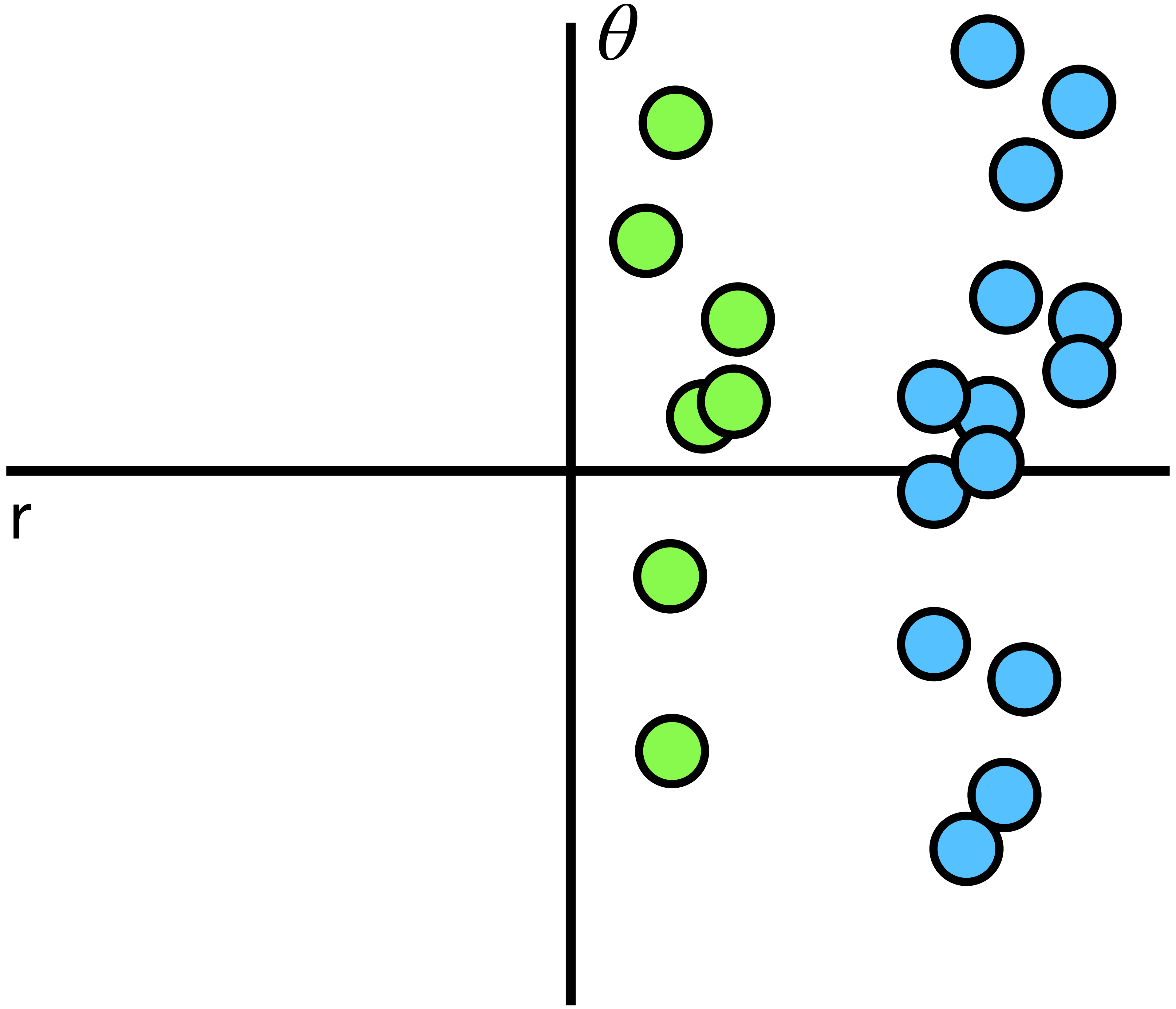
Original space



$$r = (x^2 + y^2)^{1/2}$$
$$\theta = \tan^{-1}(y/x)$$

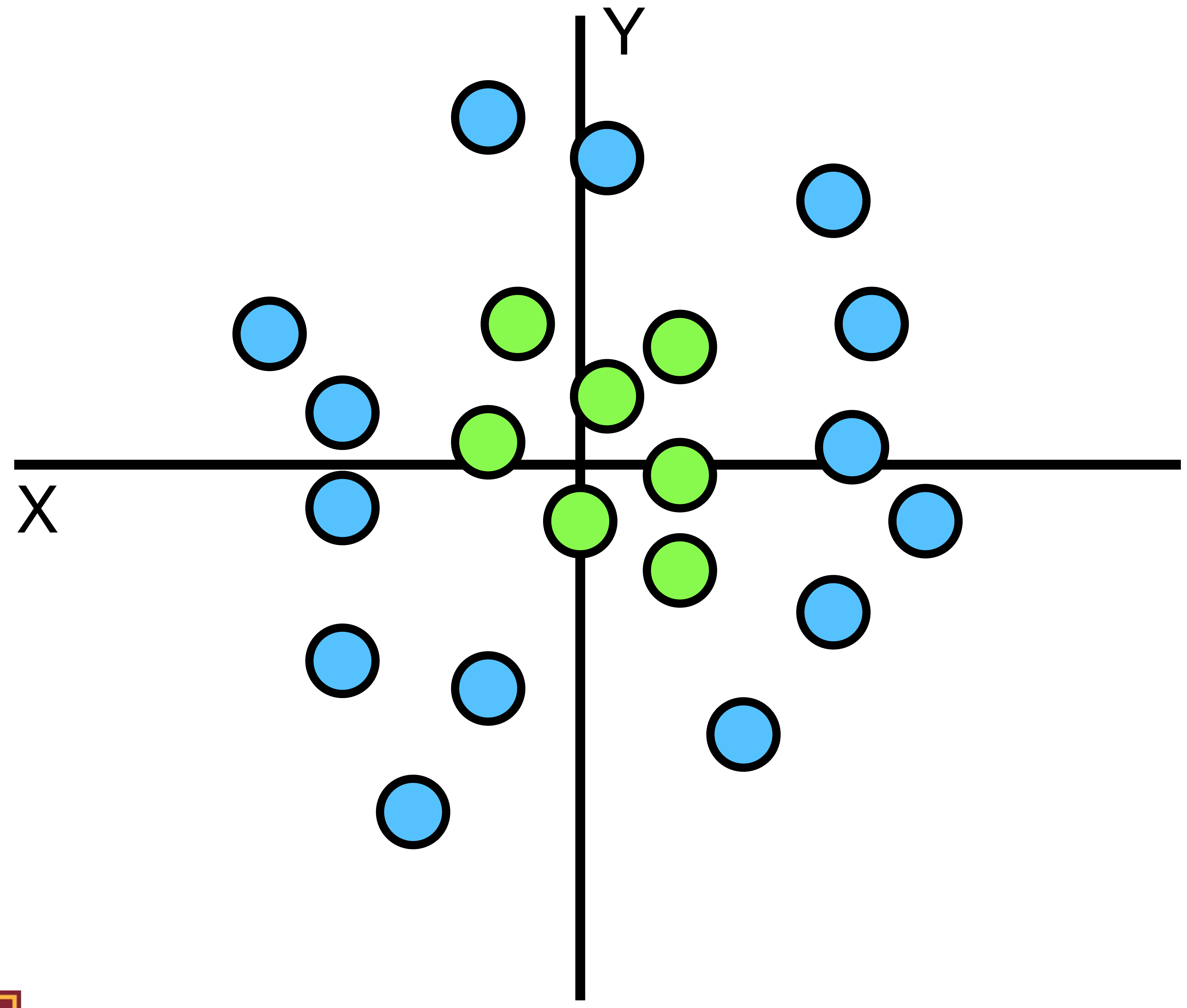
Feature Transform

Feature space



One solution: Feature Transforms

Original space

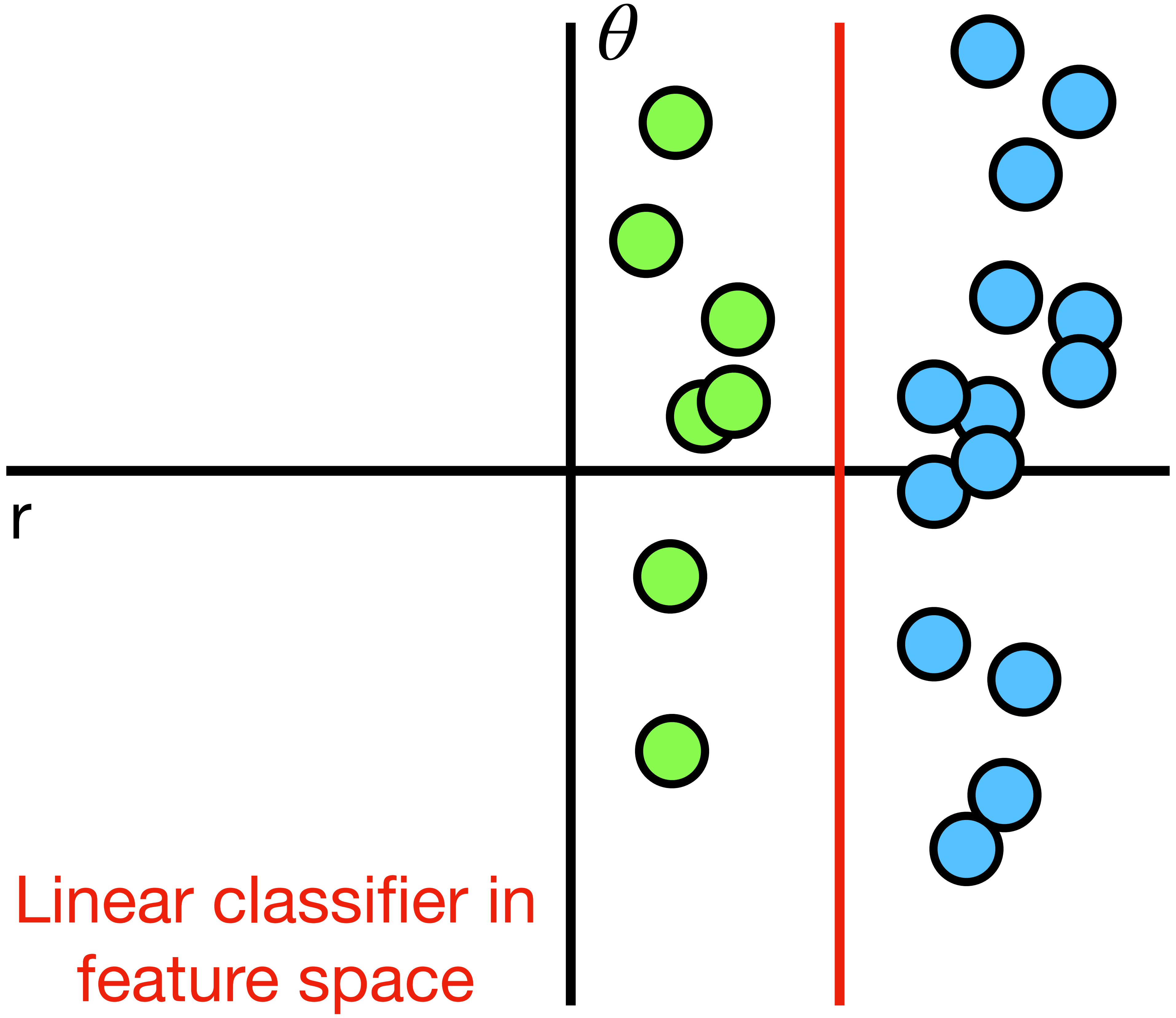


$$r = (x^2 + y^2)^{1/2}$$

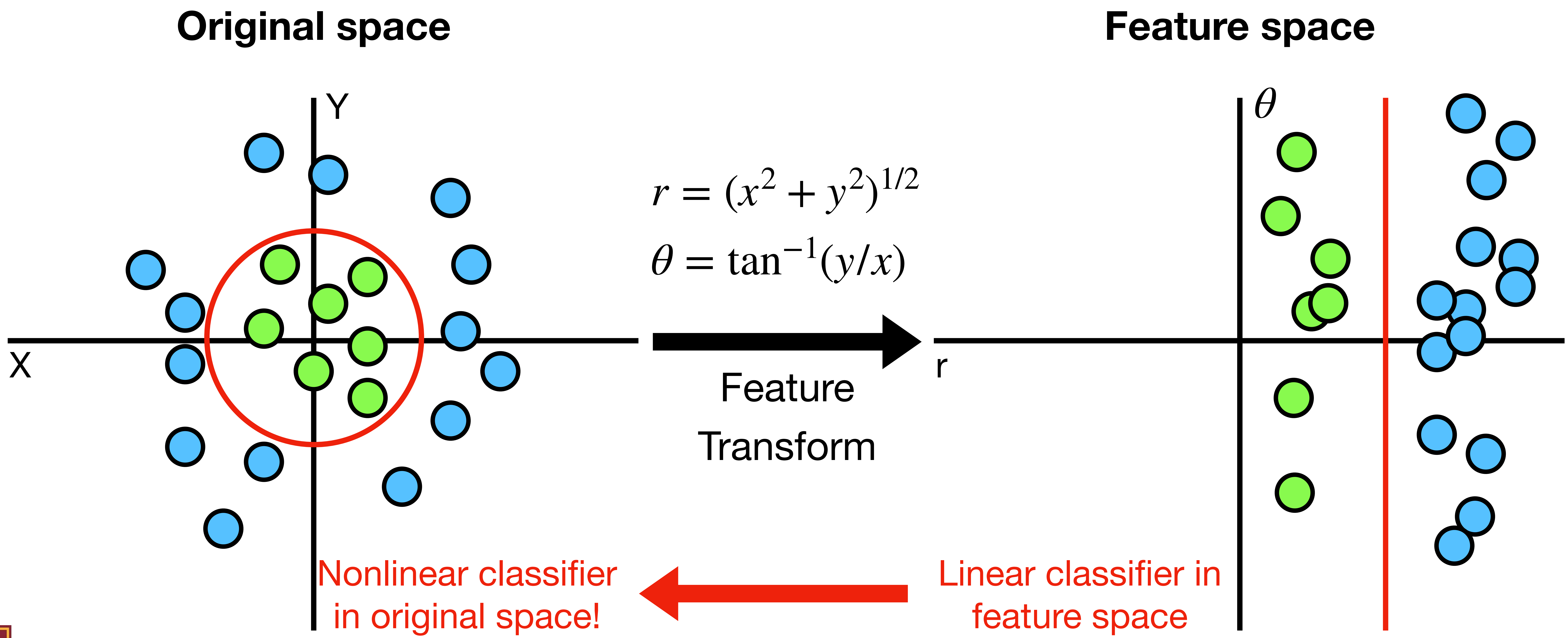
$$\theta = \tan^{-1}(y/x)$$

Feature Transform

Feature space



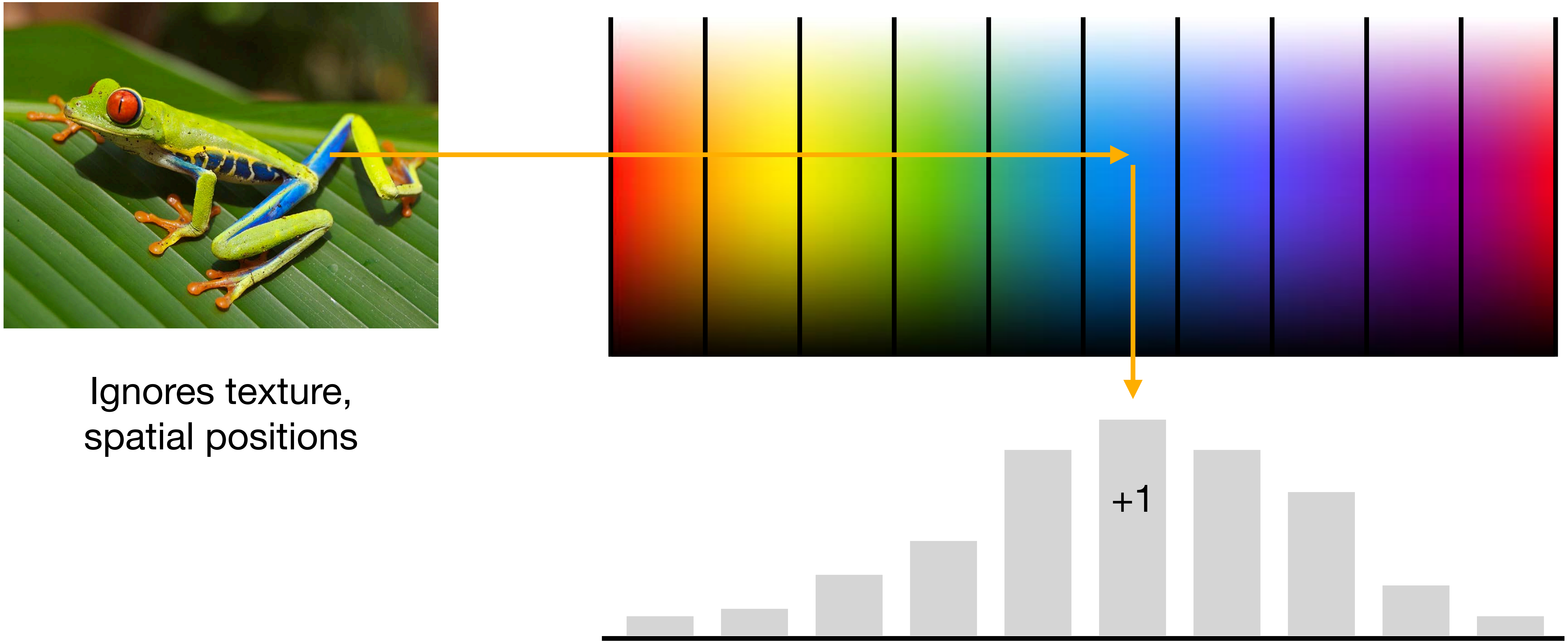
One solution: Feature Transforms



$$r = (x^2 + y^2)^{1/2}$$

$$\theta = \tan^{-1}(y/x)$$

Image Features: Color Histogram



Ignores texture, spatial positions

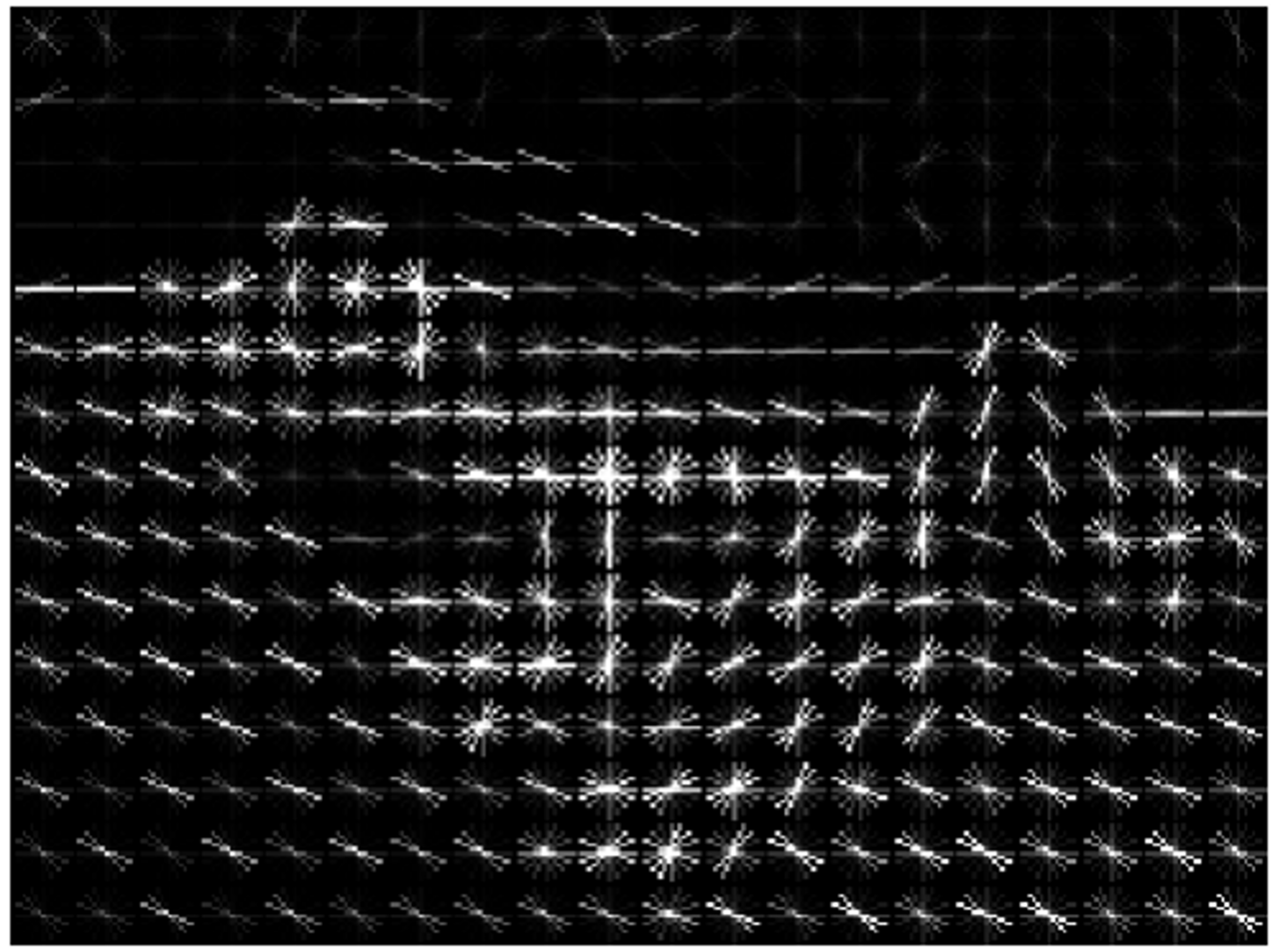
Frog image is in the public domain

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 direction 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 8x8 regions
3. Within each region compute a histogram of edge direction weighted by edge strength

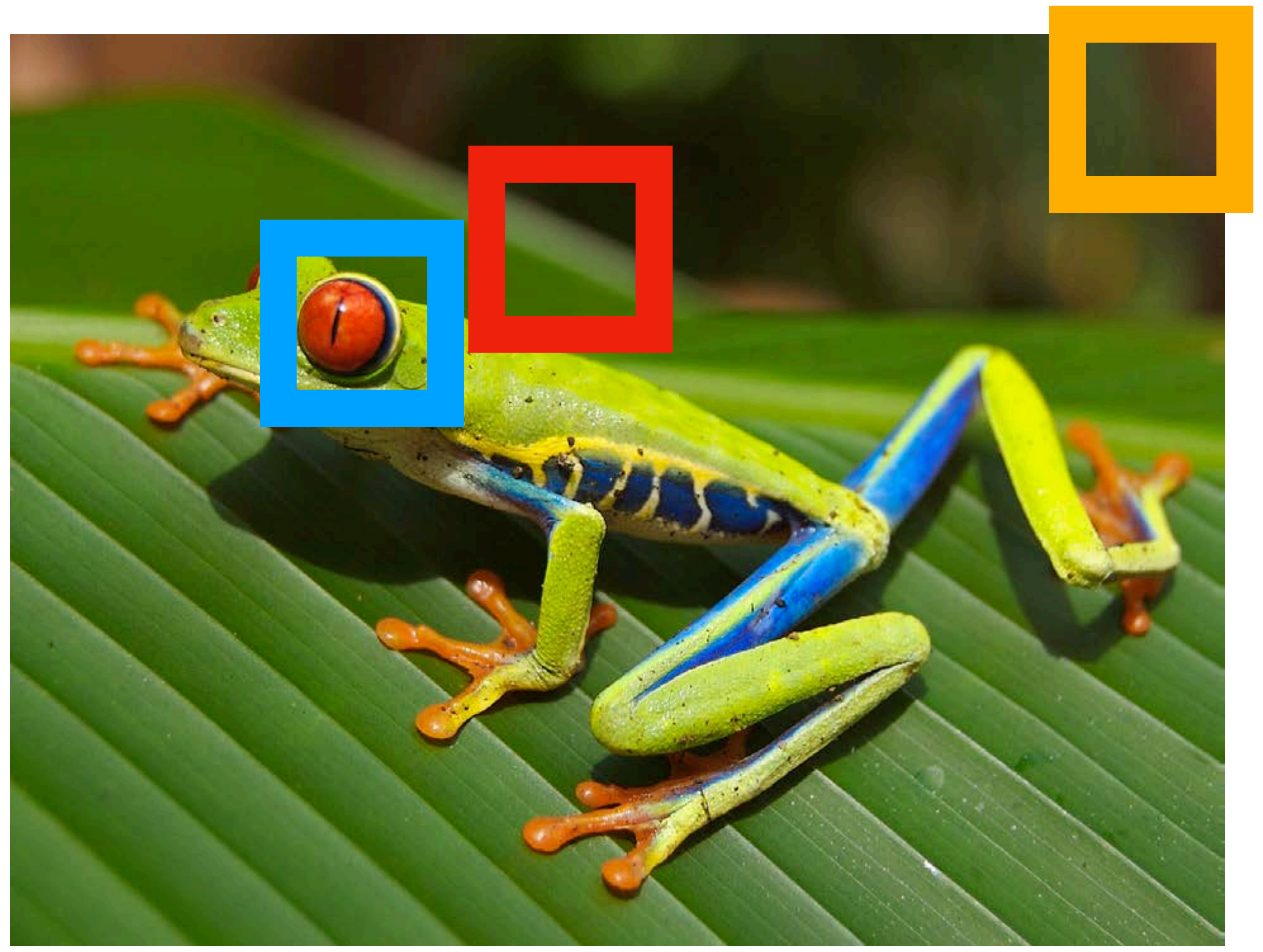
Example: 320x240 image gets divided into 40x30 bins;

9 directions per bin;

feature vector has 30*40*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

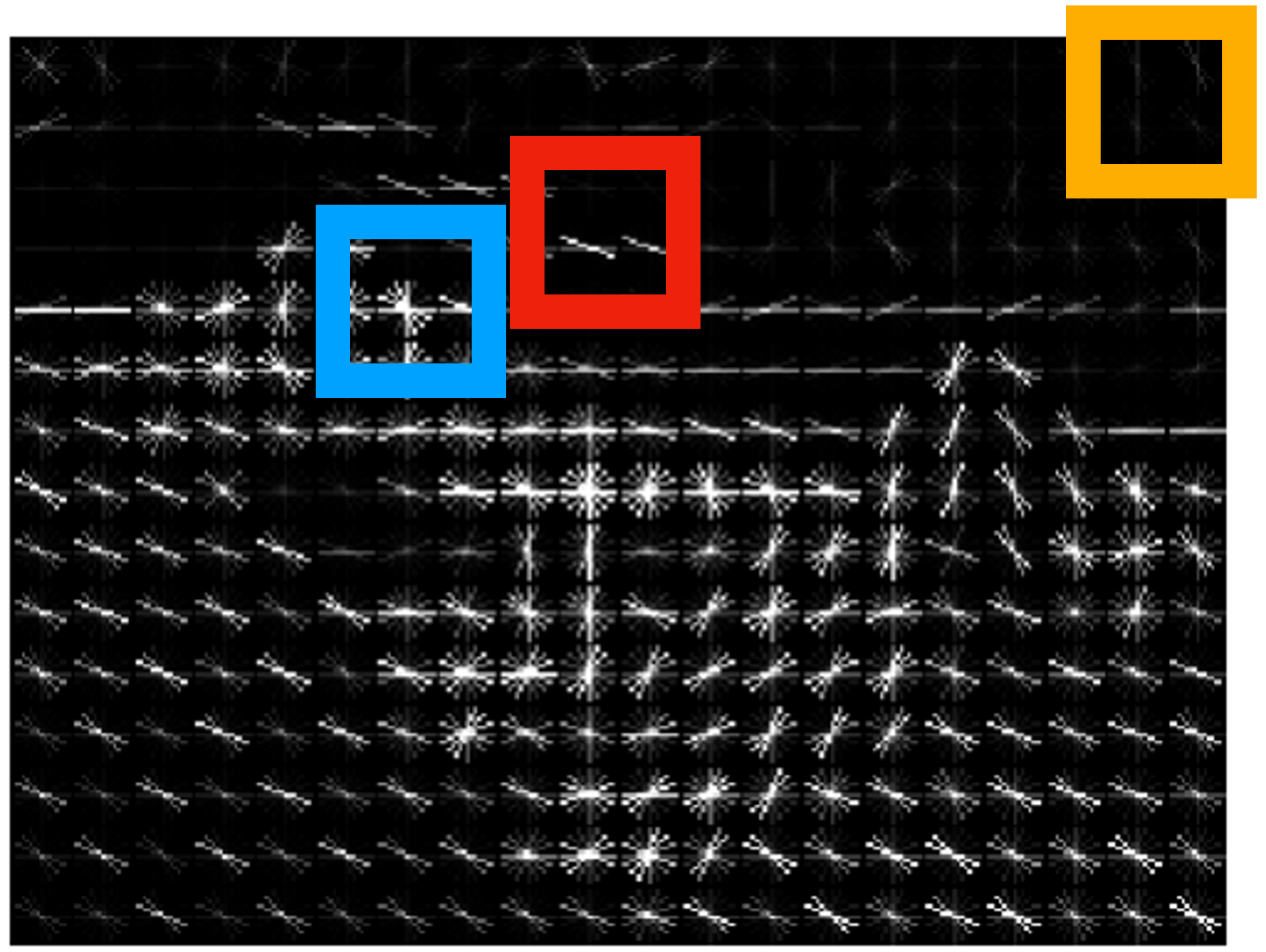
Image Features: Histogram of Oriented Gradients (HoG)



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 direction weighted by edge strength

Capture texture and position, robust to small image changes

Example: 320x240 image gets divided into 40x30 bins;

9 directions per bin;

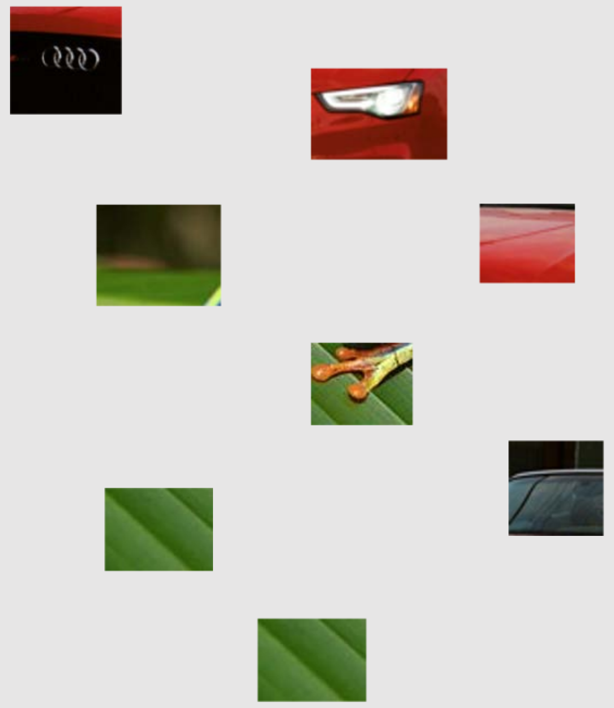
feature vector has 30*40*9 = 10,800 numbers

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”

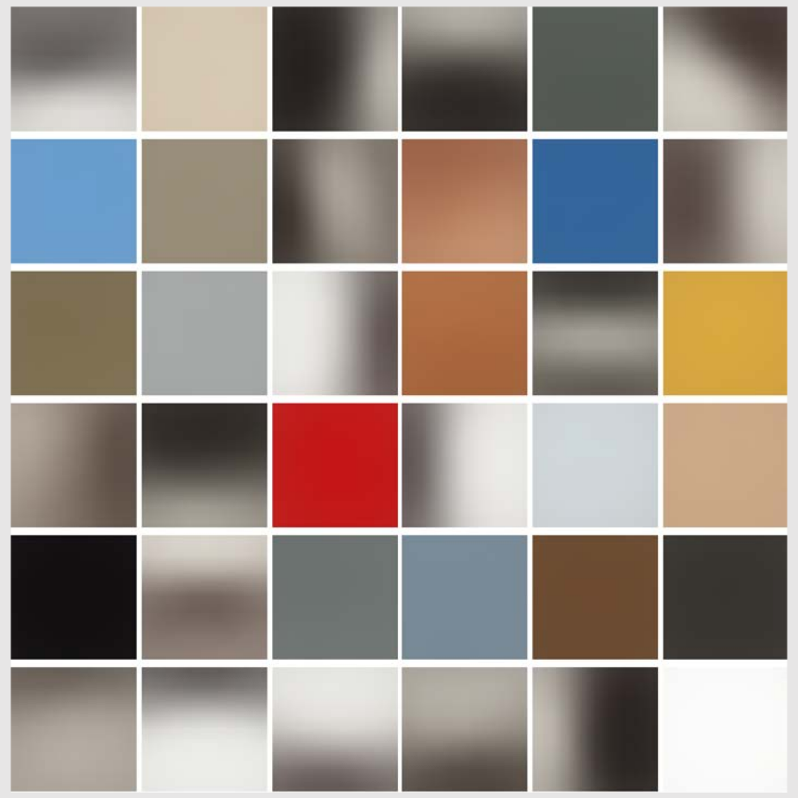
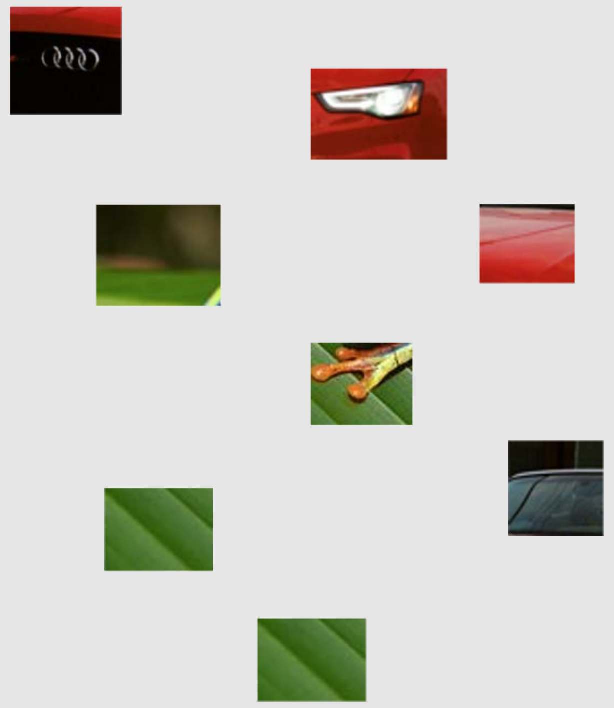


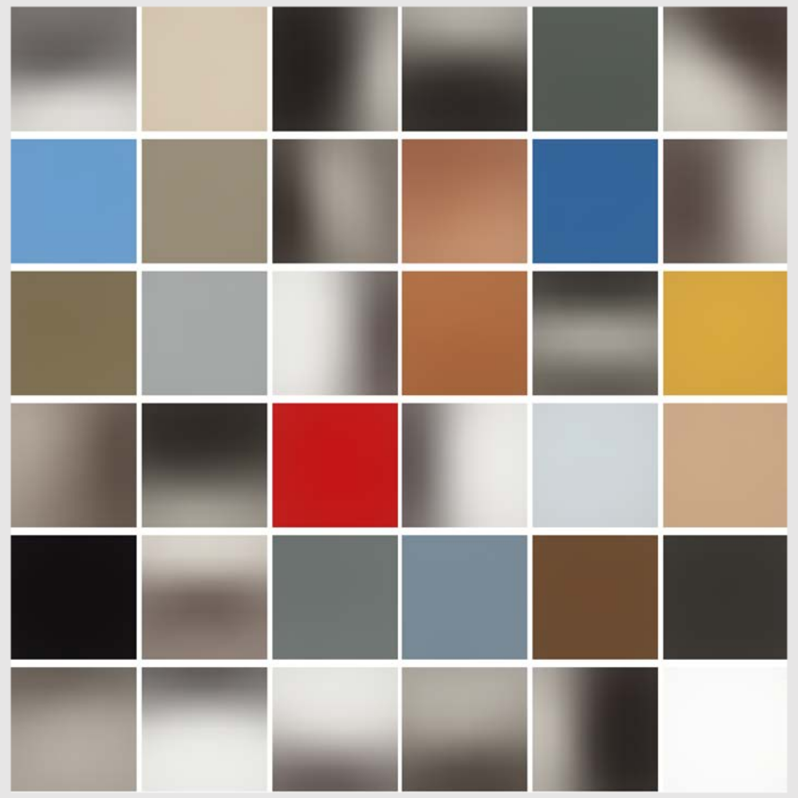
Image Features: Bag of Words (Data-Driven!)

Step 1: Build codebook

Extract random patches →

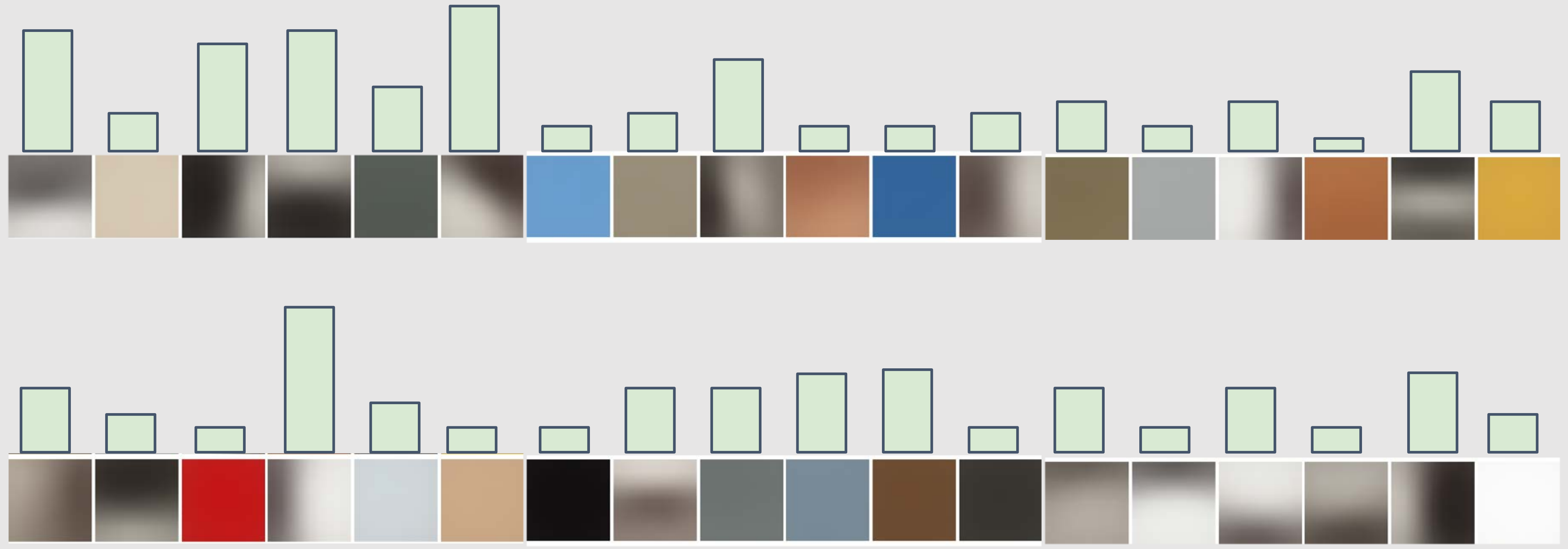


Cluster patches to form “codebook” of “visual words” →



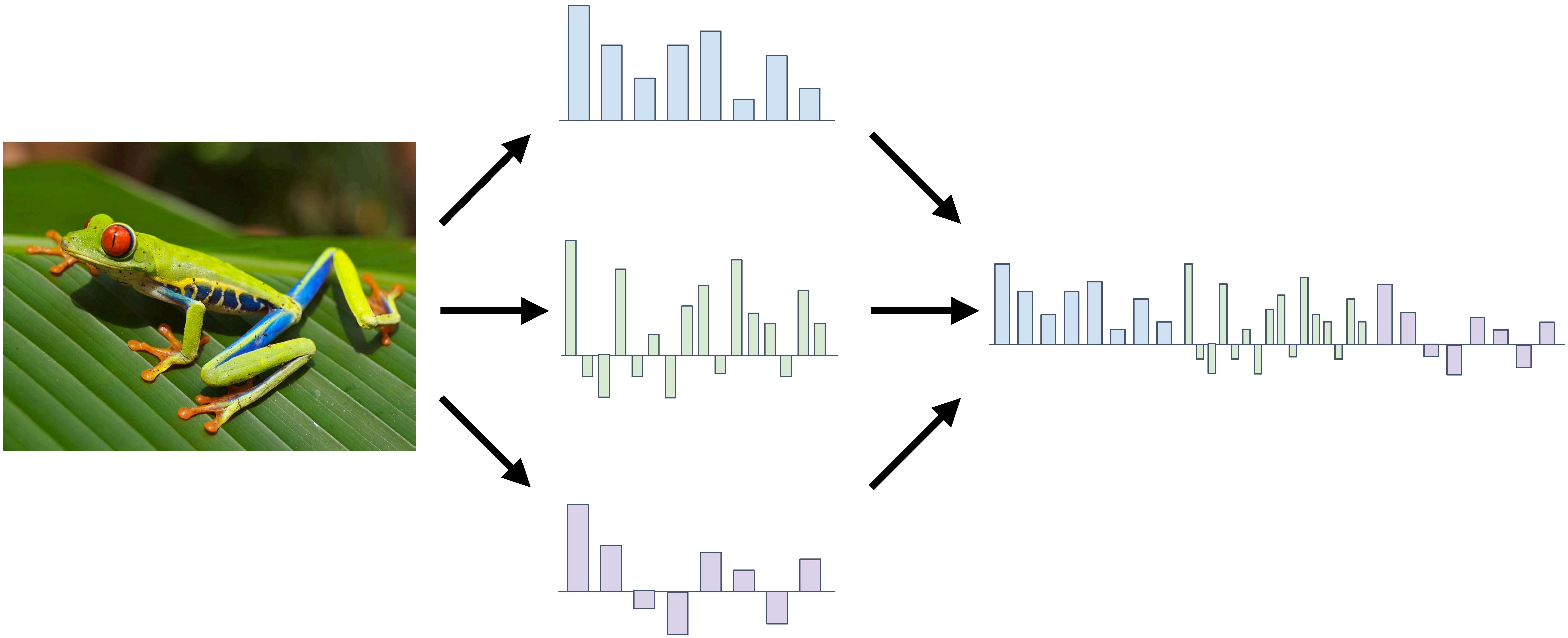
Step 2: Encode images

→



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-dims
- Color: 96-dim

} Reduced to 64-dim with PCA

FV extraction and compression:

- N=1024 Gaussians, R=4 regions $\rightarrow$ 520K dim x 2
- Compression: G=8, b=1 bit per dimension

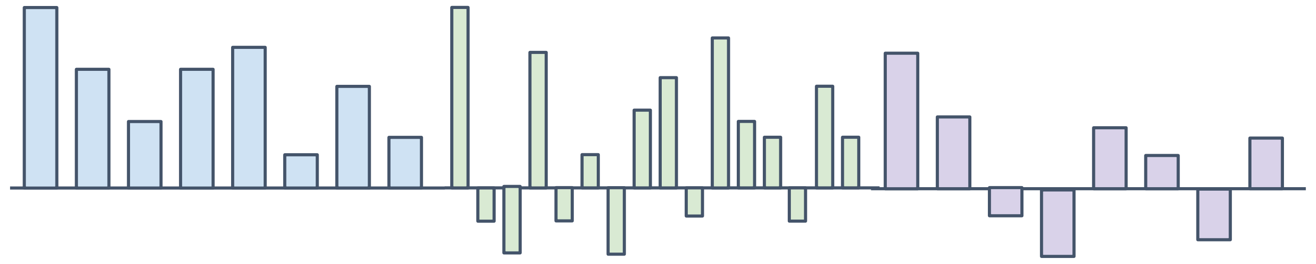
One-vs-all SVM learning with SGD

Late fusion of SIFT and color systems

F. Perronnin, J. Sánchez, "Compressed Fisher vectors for LSVRC", PASCAL VOC / ImageNet workshop, ICCV, 2011.

Image Features

Feature Extraction



f

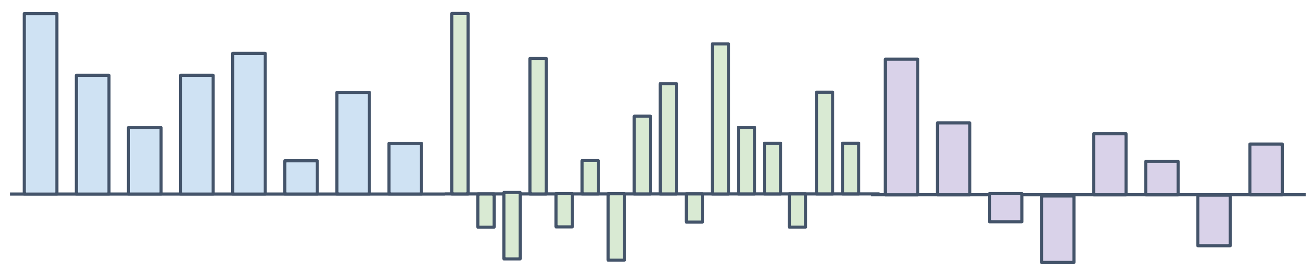
10 numbers giving scores for classes

training

Image Features vs Neural Networks

Image Features vs Neural Networks

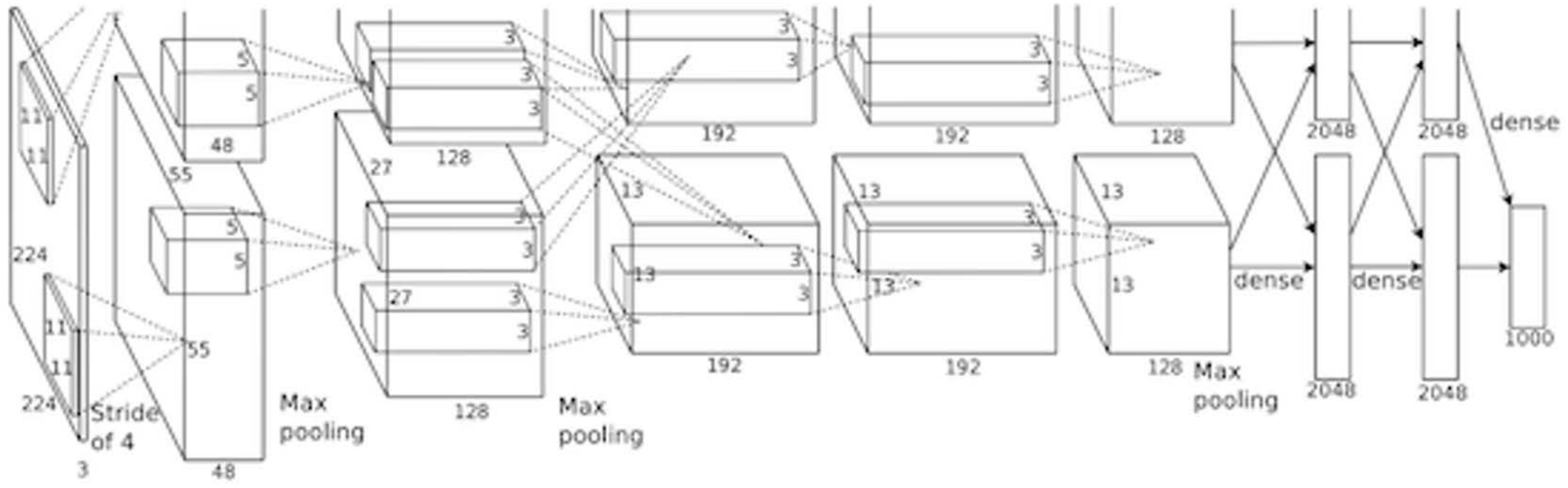
Feature Extraction



f

10 numbers giving scores for classes

training



Krizhevsky, Sutskever, and Hinton, "Imagenet classification with deep convolutional neural networks", NIPS 2012.

10 numbers giving scores for classes

training

Neural Networks

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

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

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

Neural Networks

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

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

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

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

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

Feature Extraction

Linear Classifier

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

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

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

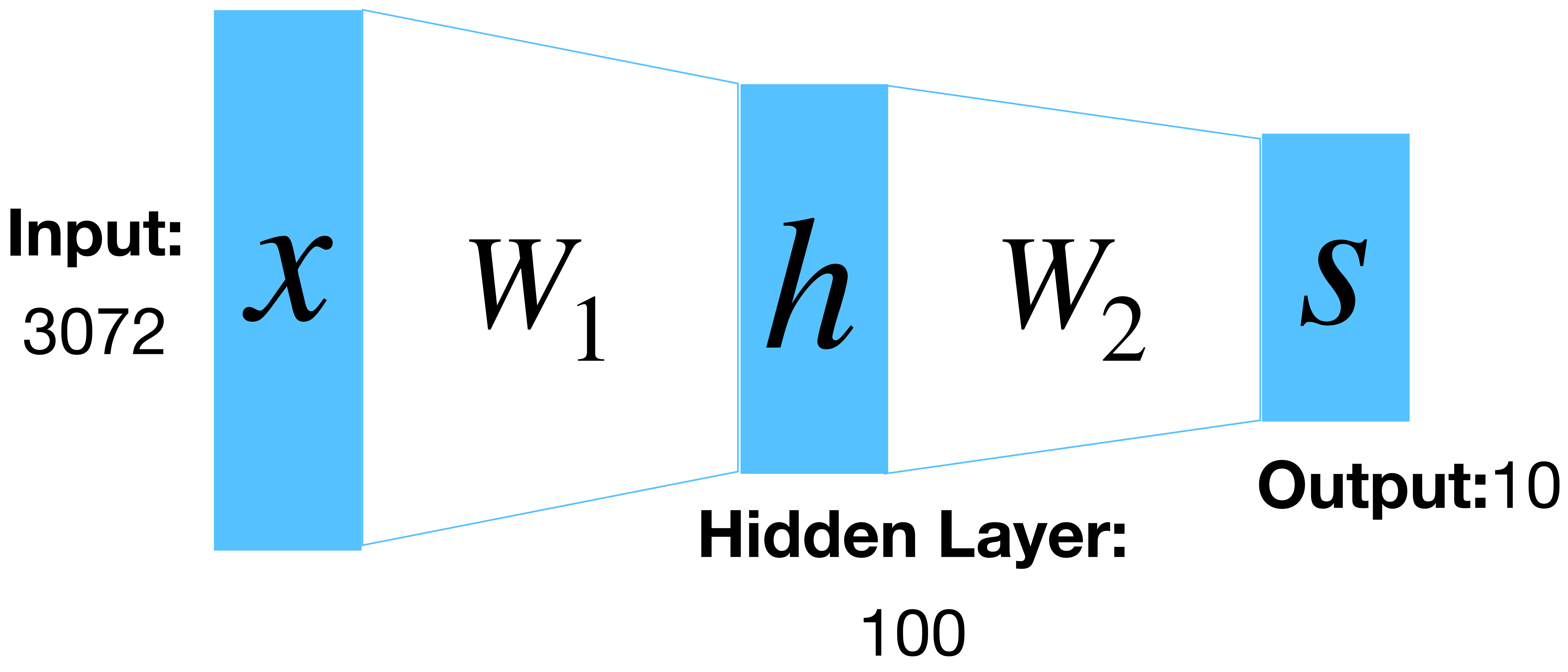
Or Three-Layer Neural Network:

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

Neural Networks

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

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



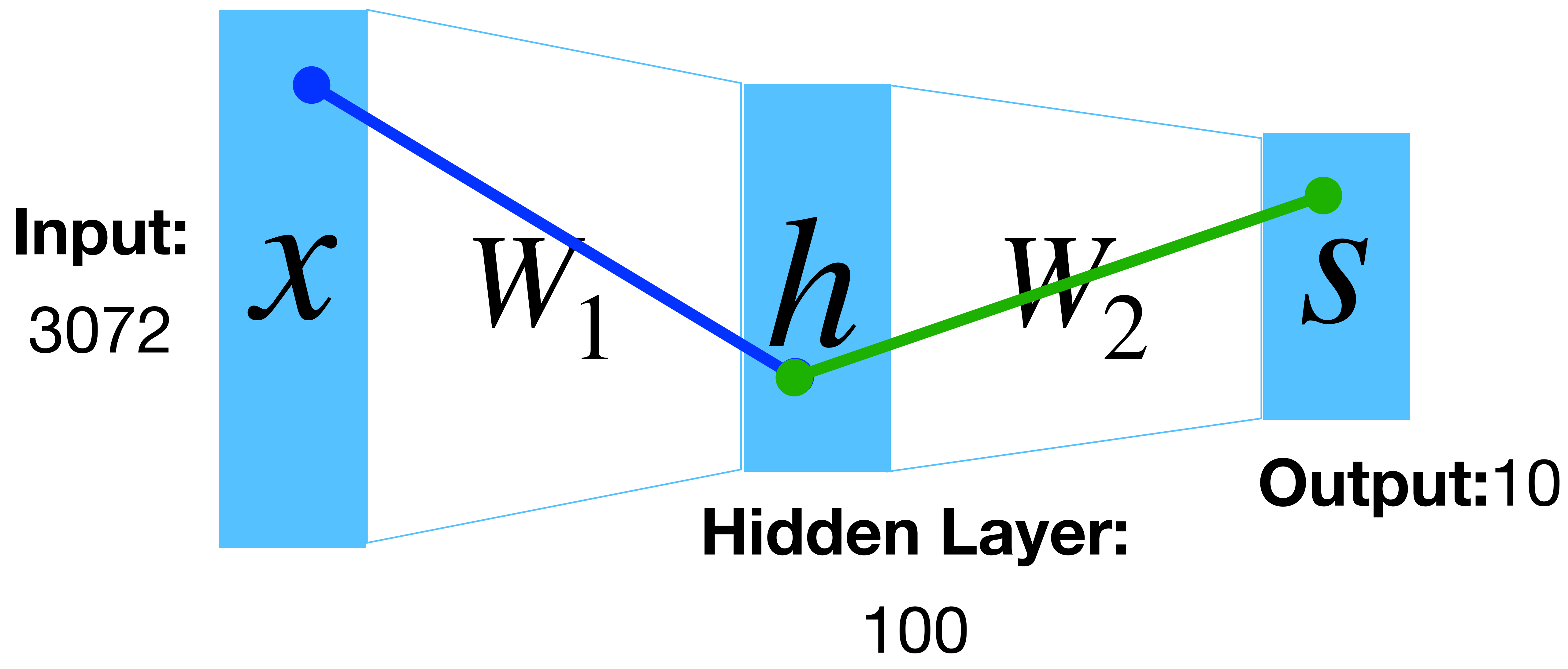
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

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

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

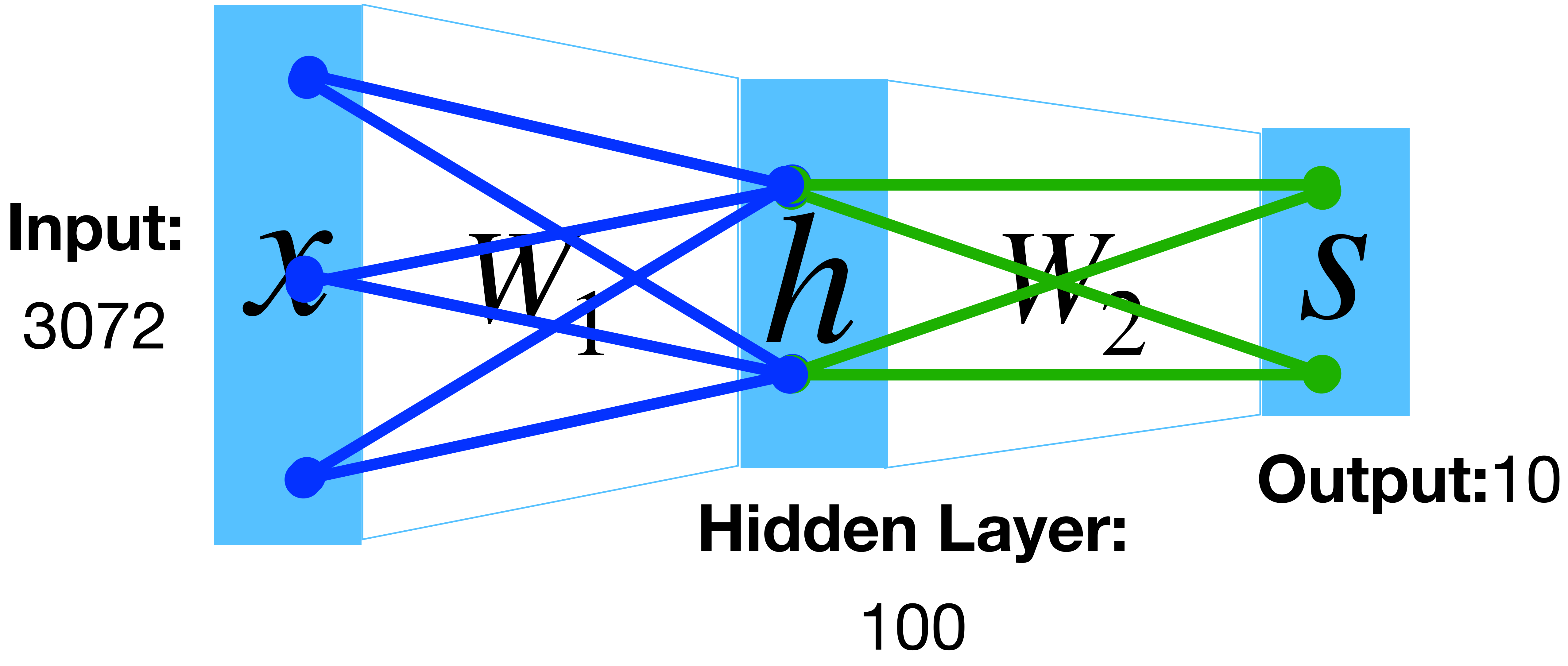
Neural Networks

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

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

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

All elements of x affect all elements of h



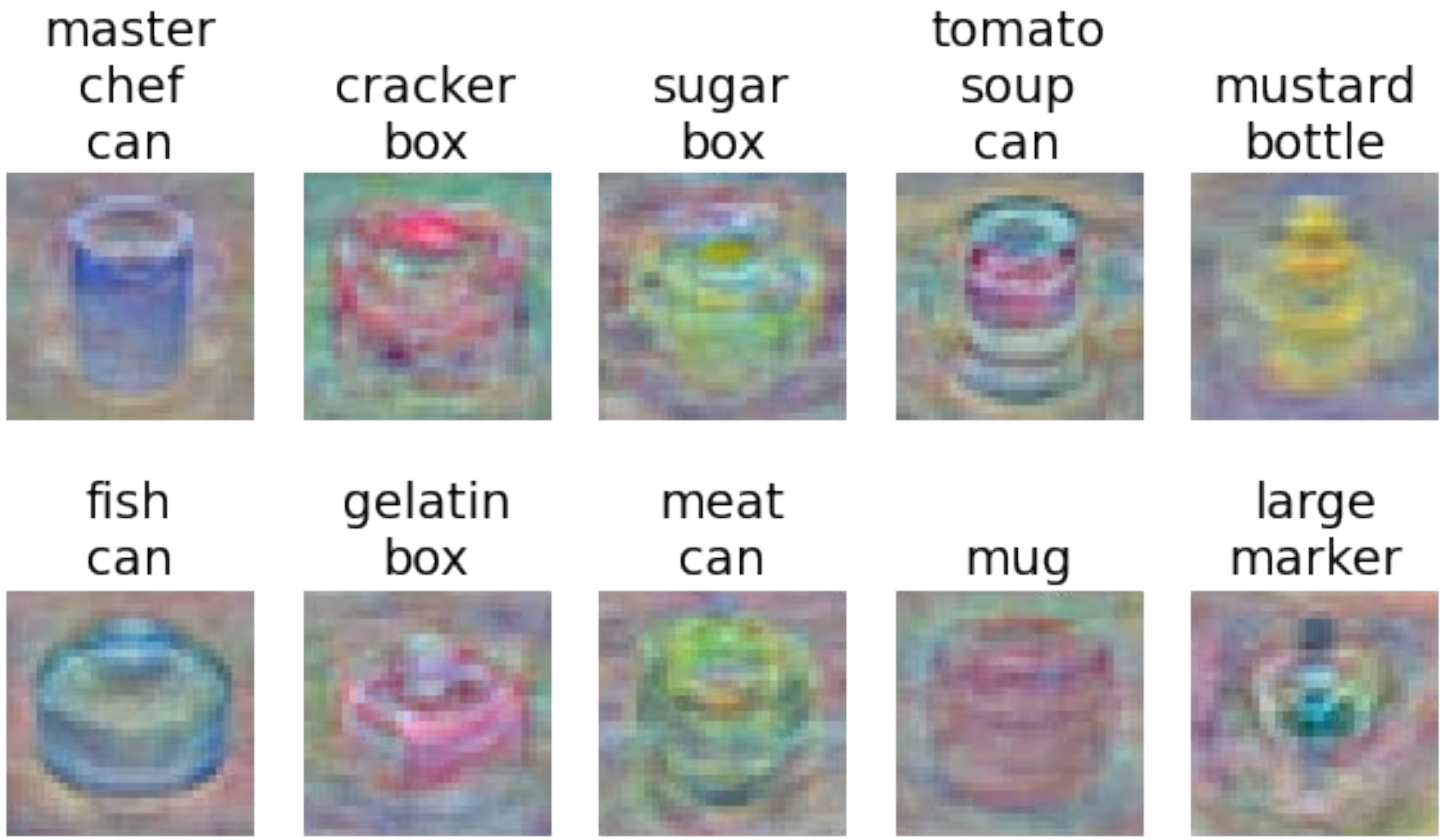
Element (i, j) of W_2 gives the effect on s_i from h_j

All elements of h affect all elements of s

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

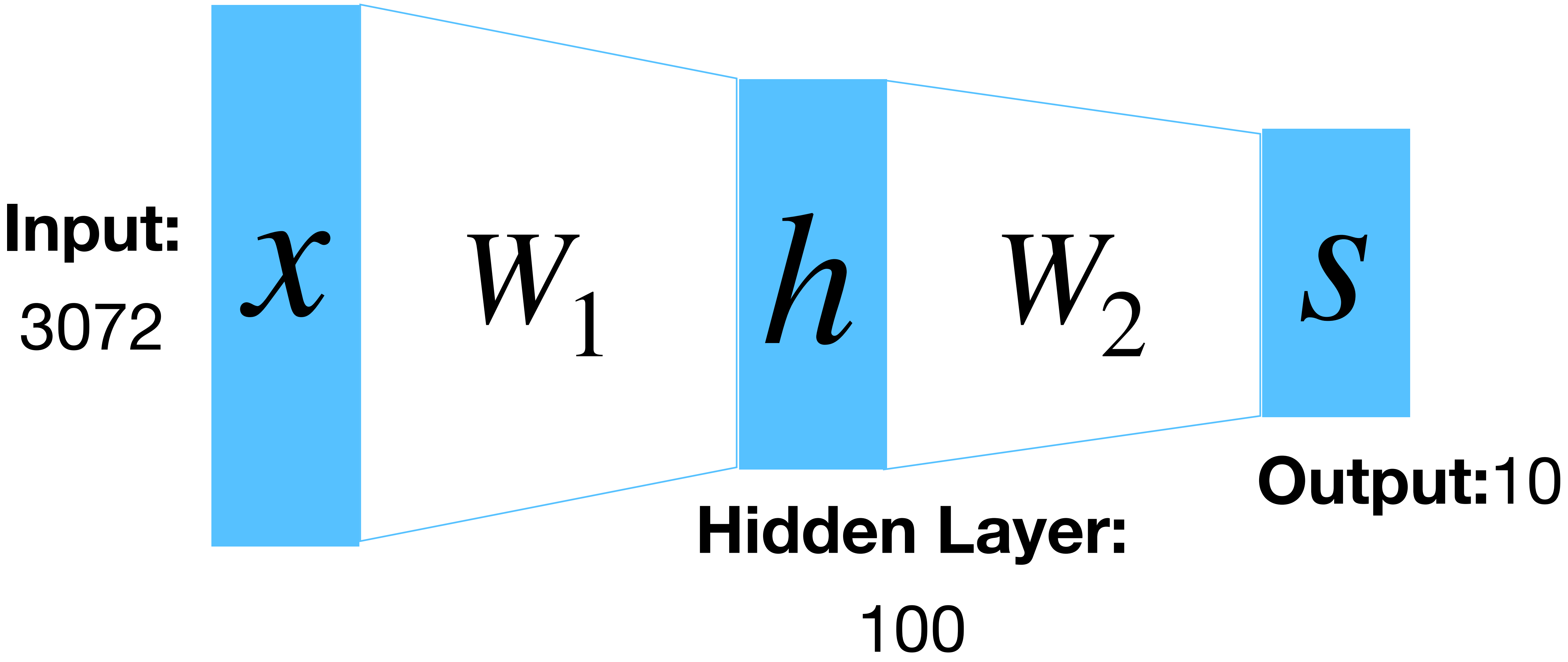
Neural Networks

Linear classifier: One template per class



Before: Linear score function

Now: Two-Layer Neural Network:



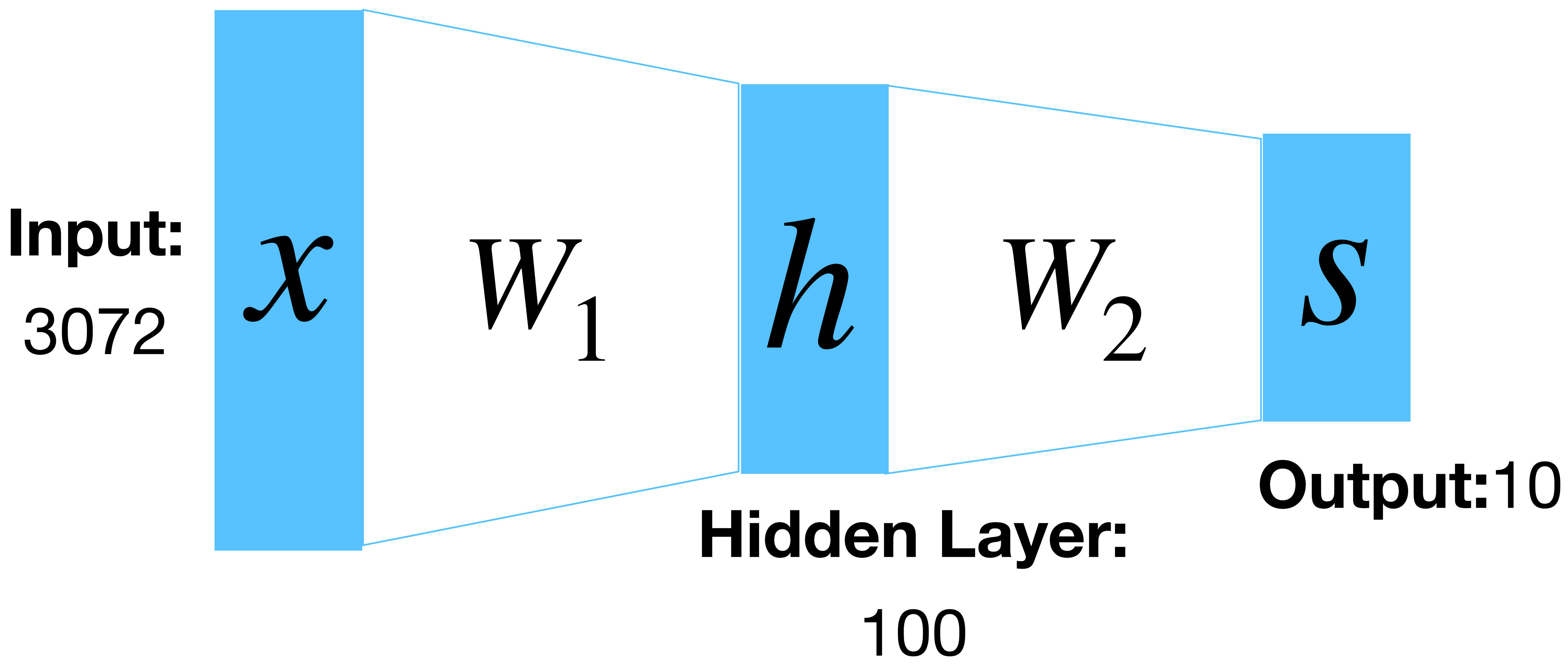
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

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

Before: Linear score function

Now: Two-Layer Neural Network:



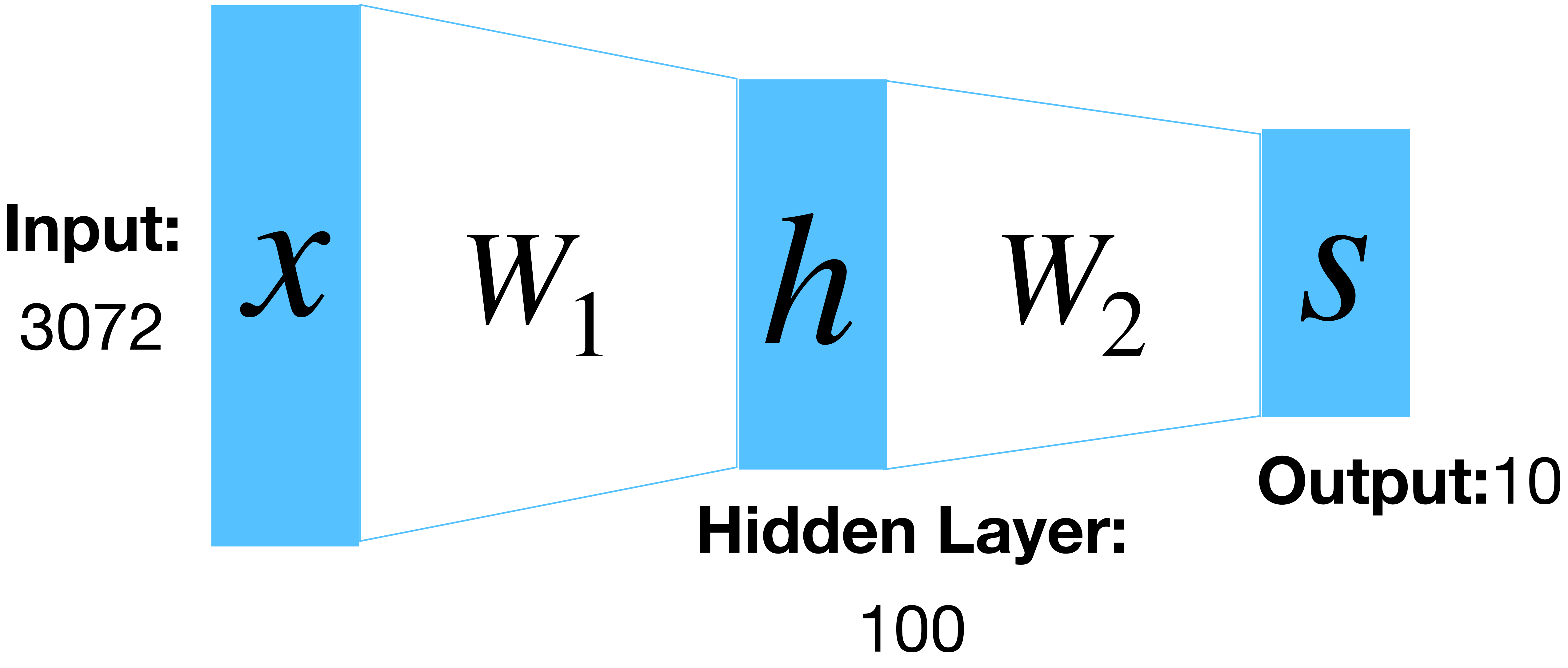
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

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

Before: Linear score function

Now: Two-Layer Neural Network:



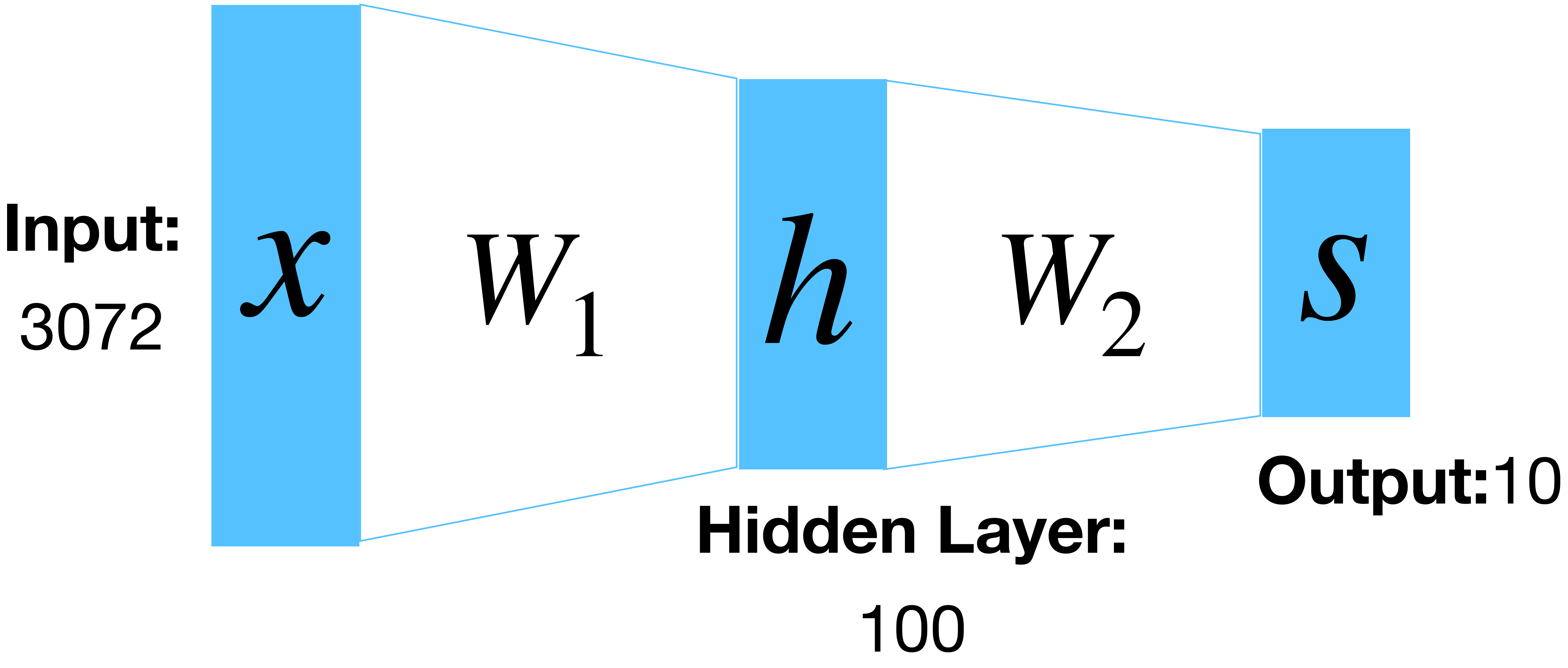
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

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

Before: Linear score function

Now: Two-Layer Neural Network:



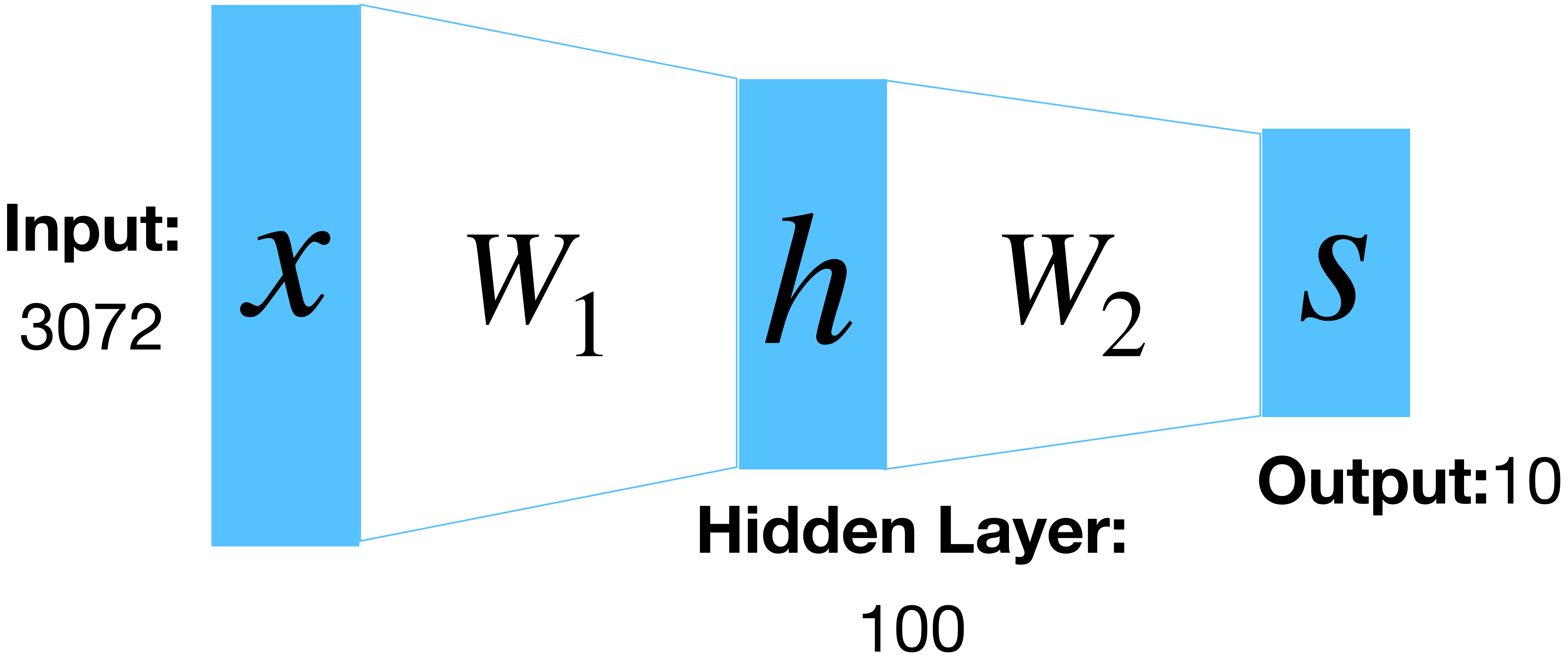
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

"Distributed representation": Most templates not interpretable!

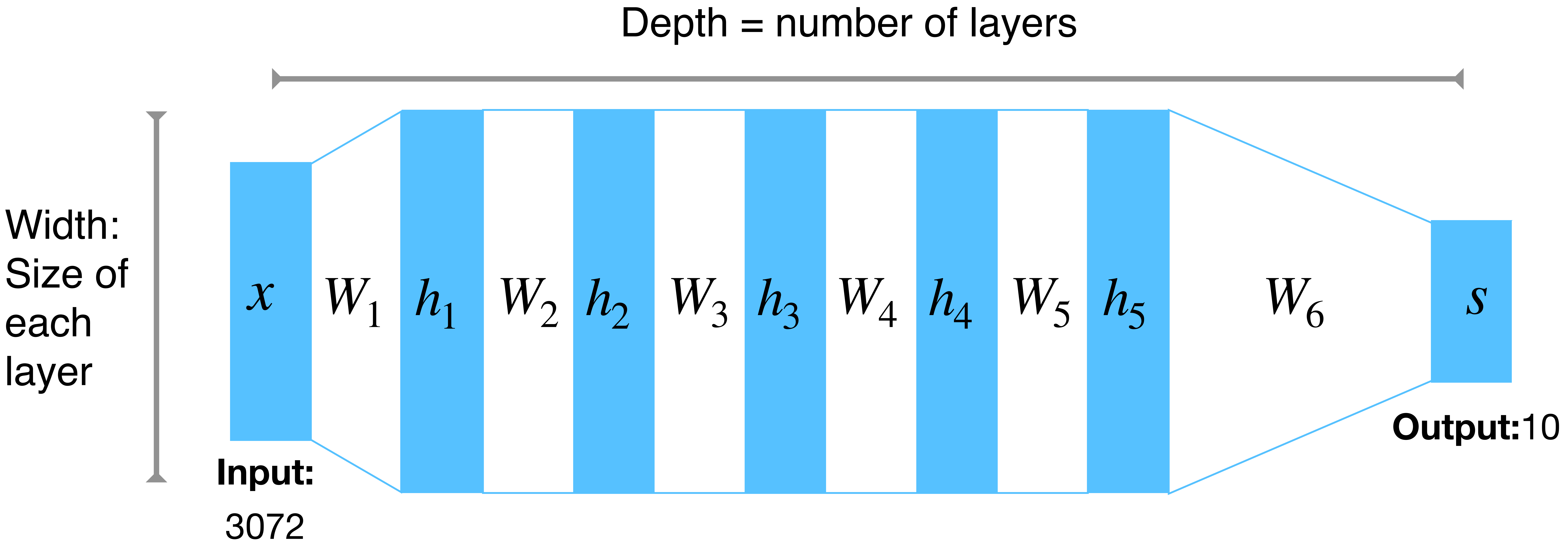
Before: Linear score function

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



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

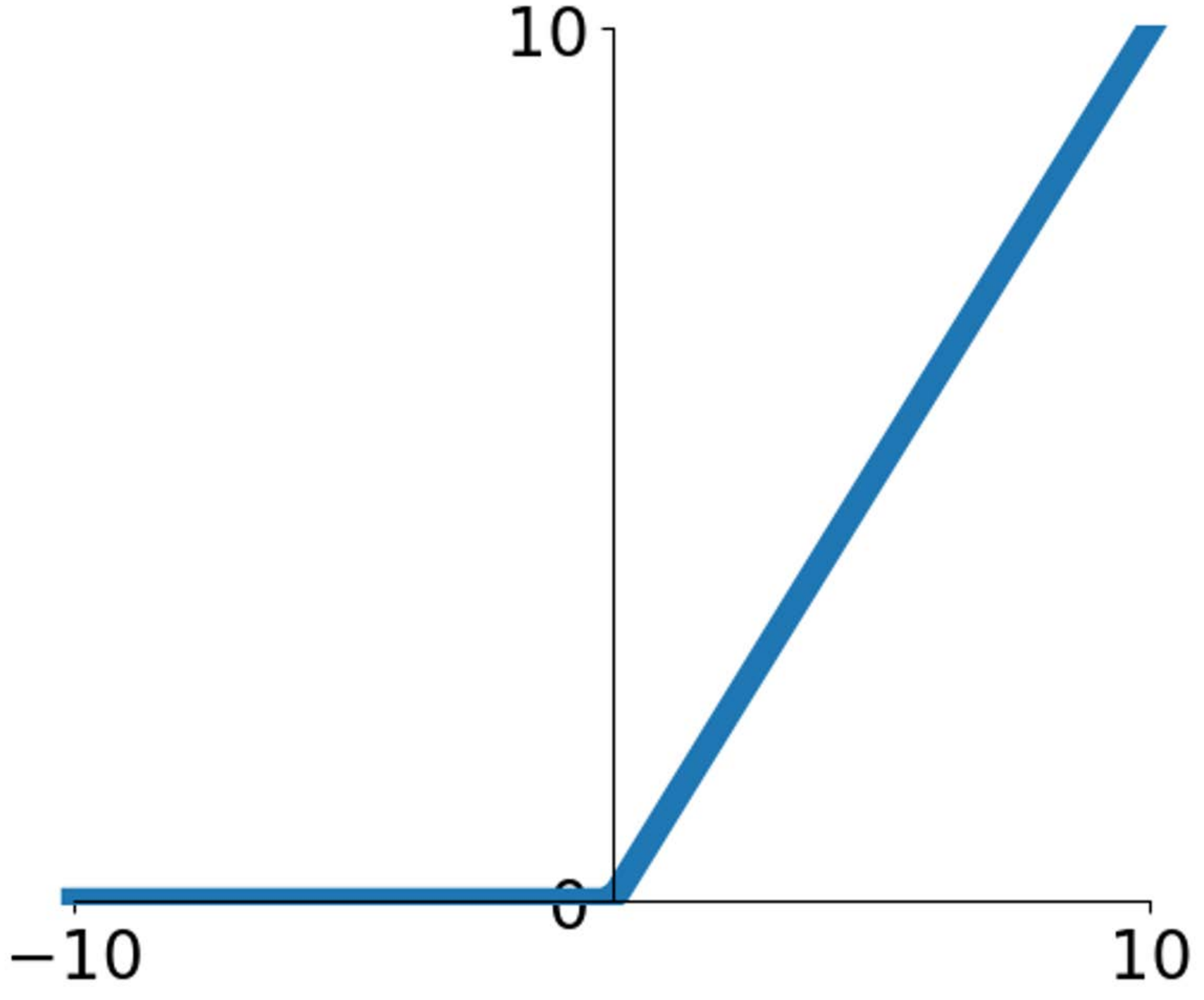
Activation Functions

2-Layer Neural Network

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

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

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

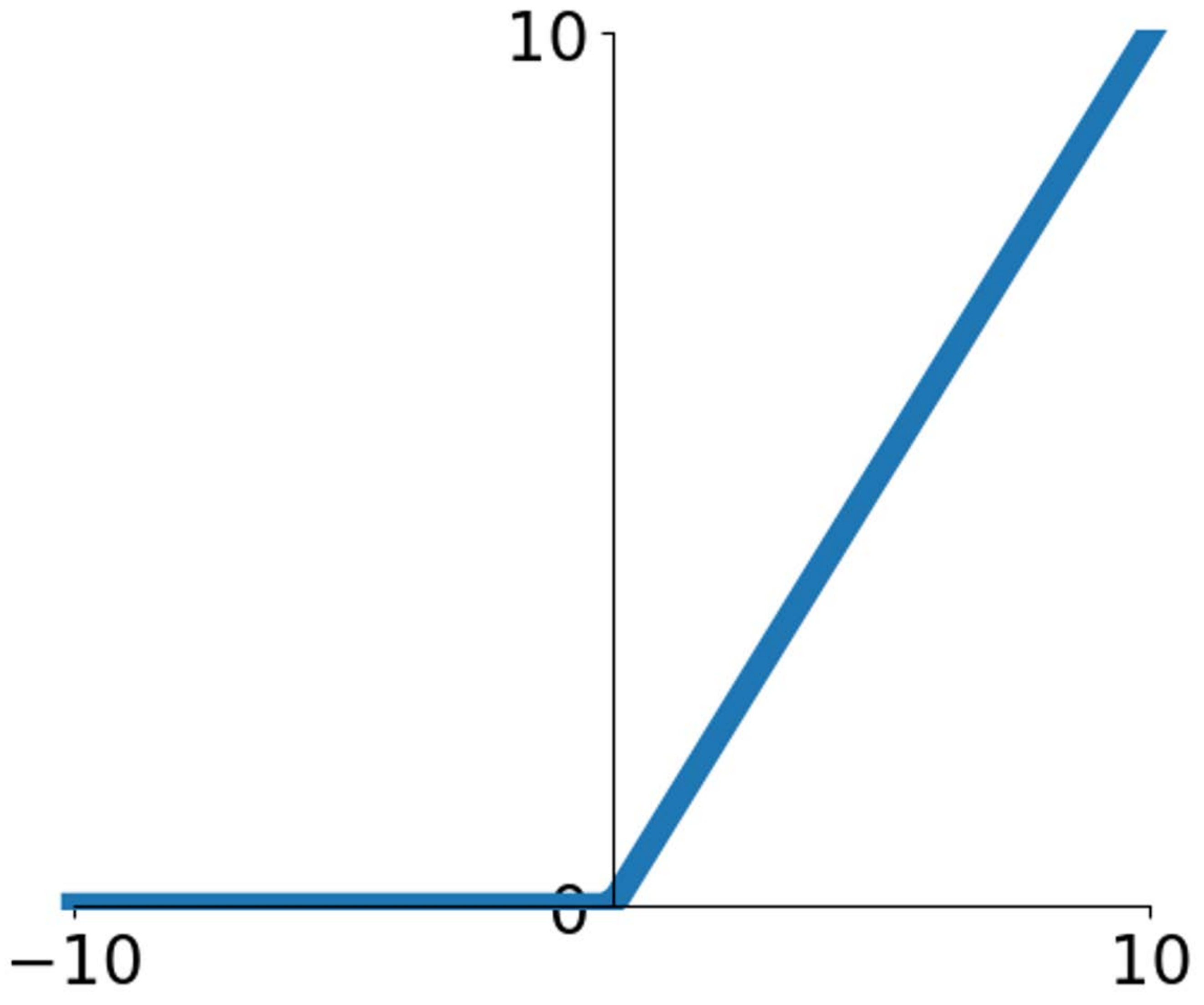


Activation Functions

2-Layer Neural Network

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

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

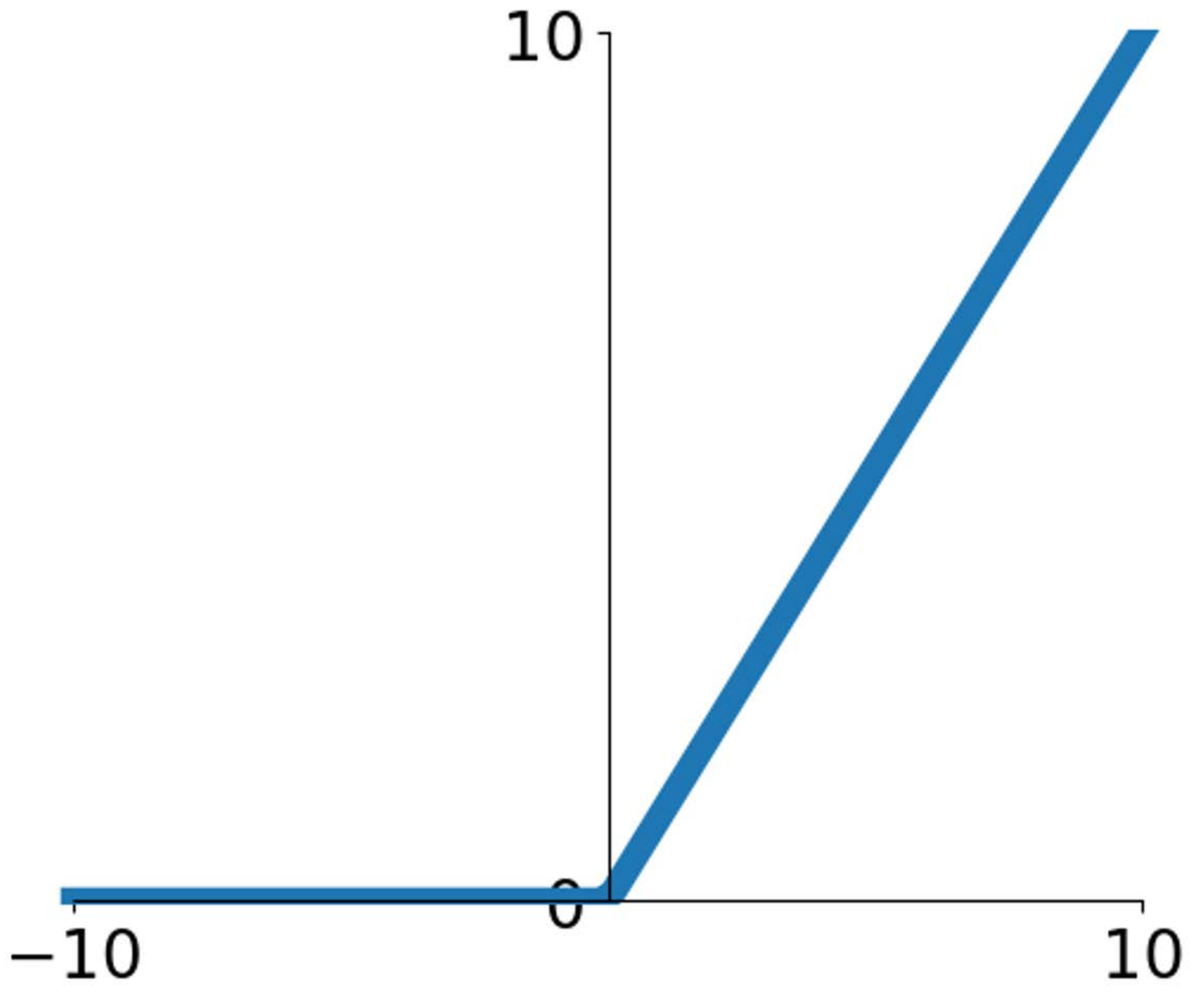
$$f(x) = W_2(W_1 x + b_1) + b_2$$

Activation Functions

2-Layer Neural Network

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

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

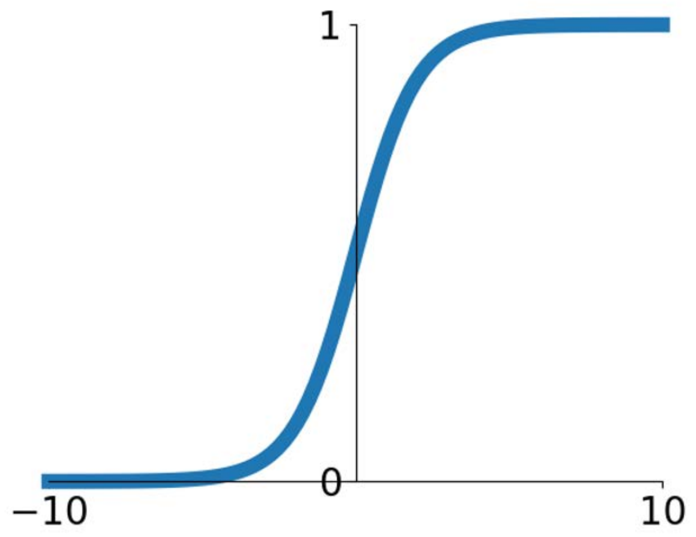
$$f(x) = W_2(W_1 x + b_1) + b_2$$
$$= (W_1 W_2)x + (W_2 b_1 + b_2)$$

A: We end up with a linear classifier

Activation Functions

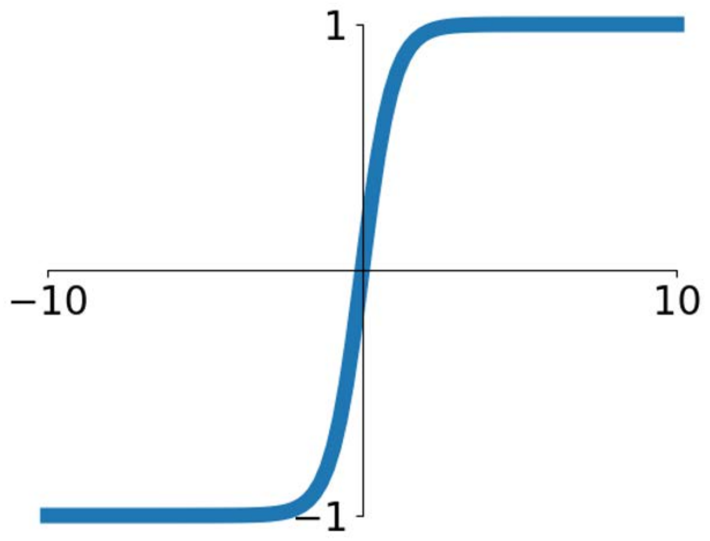
Sigmoid

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



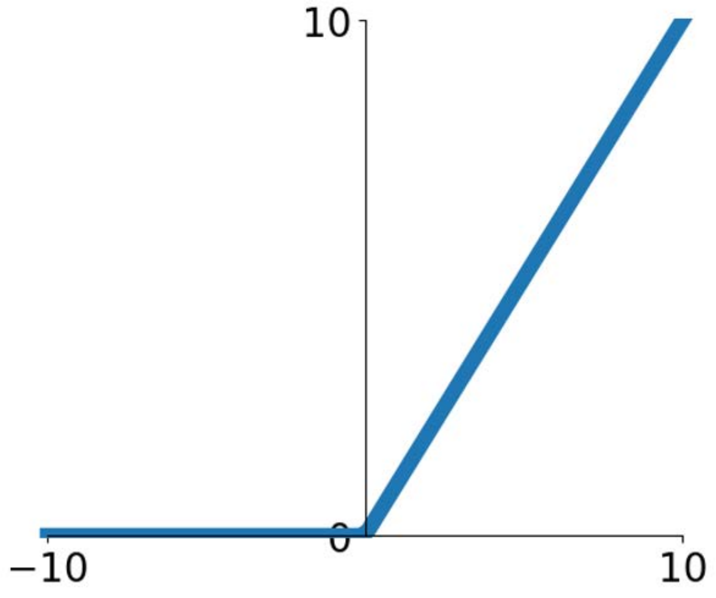
tanh

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



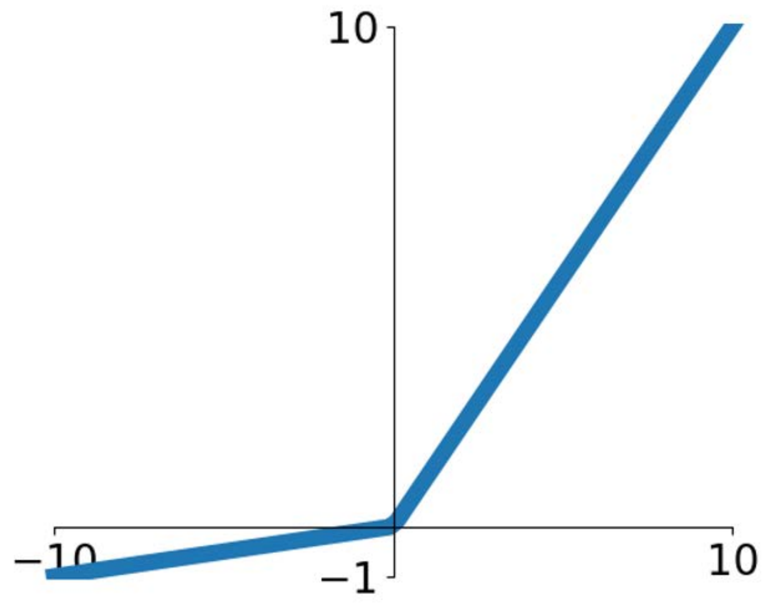
ReLU

$$\max(0,x)$$



Leaky ReLU

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

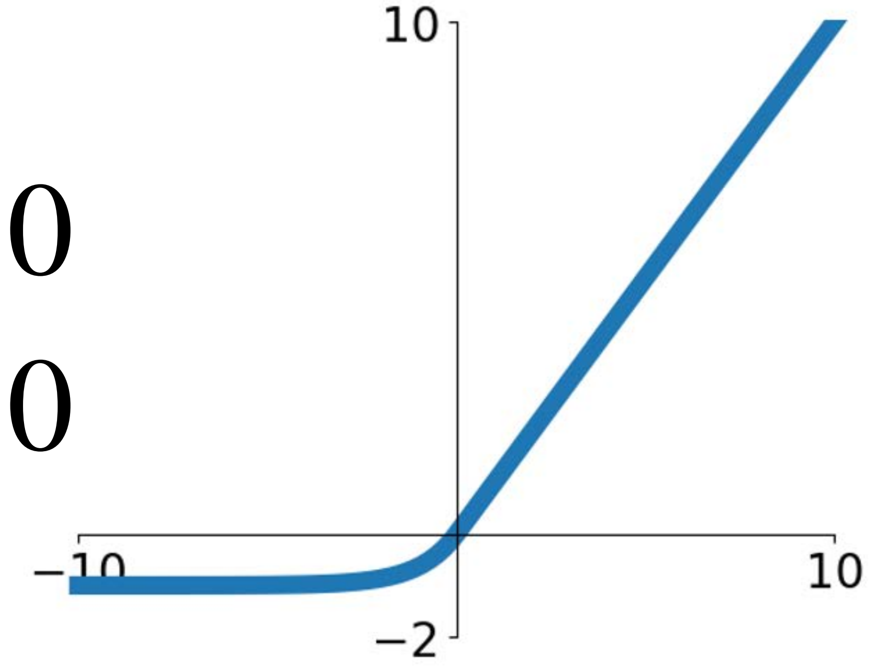


Softplus

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

ELU

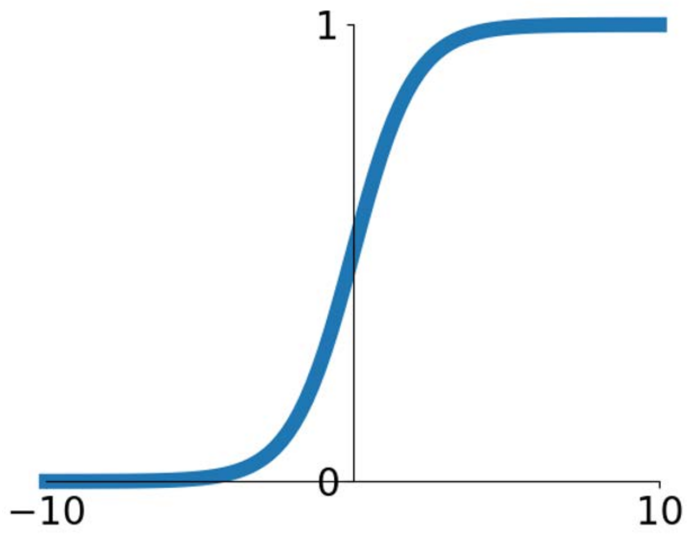
$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) -), & x \leq 0 \end{cases}$$



Activation Functions

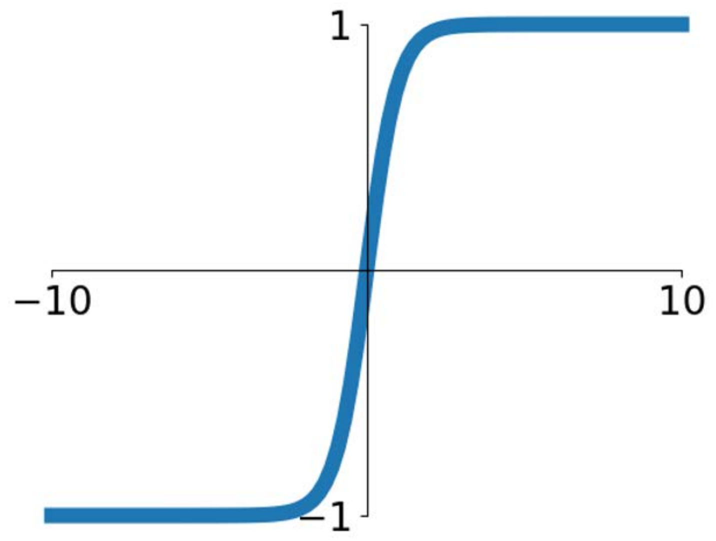
Sigmoid

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



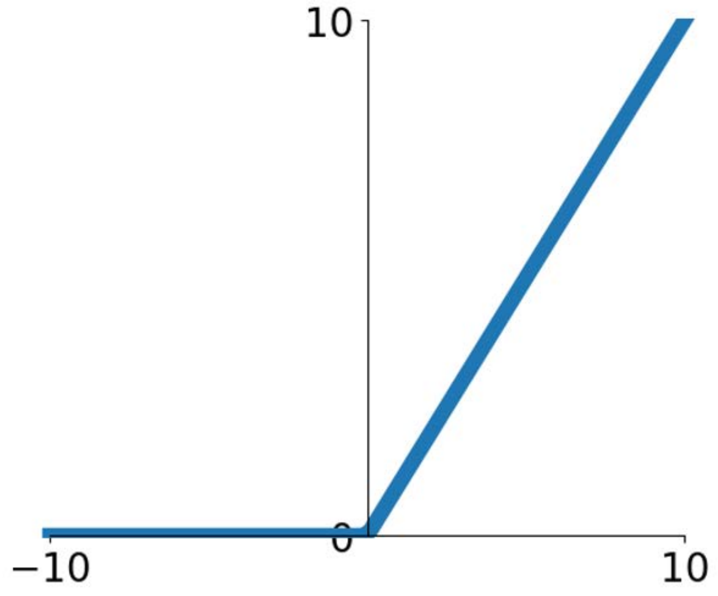
tanh

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



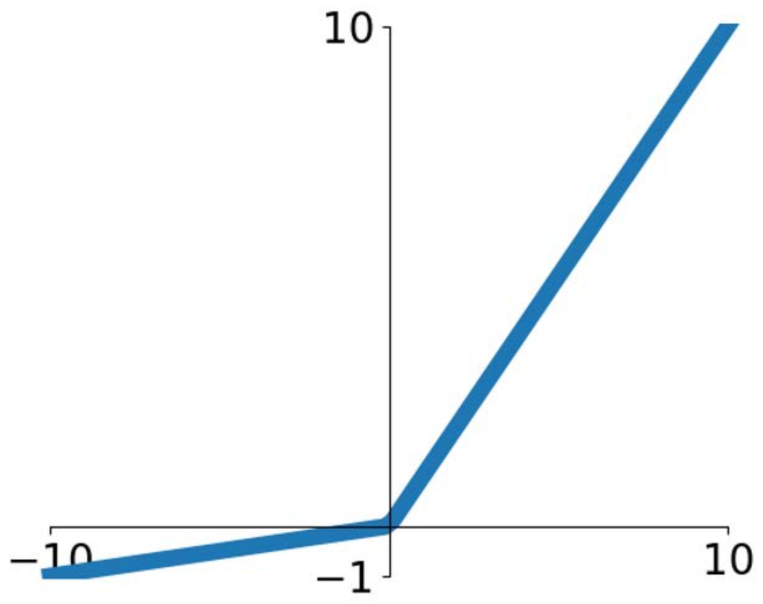
ReLU

$\max(0,x)$



Leaky ReLU

$\max(0.2x, x)$

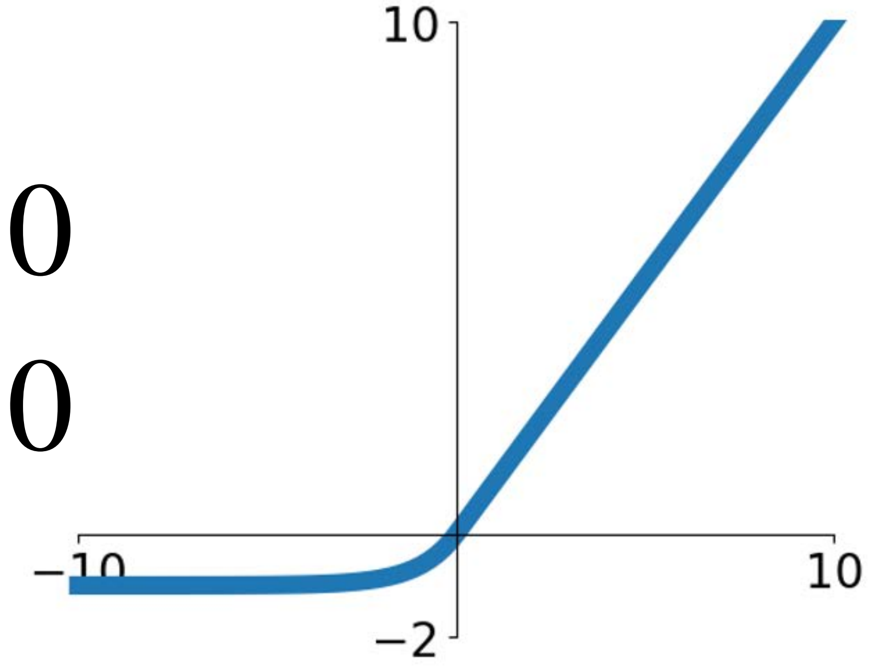


Softplus

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

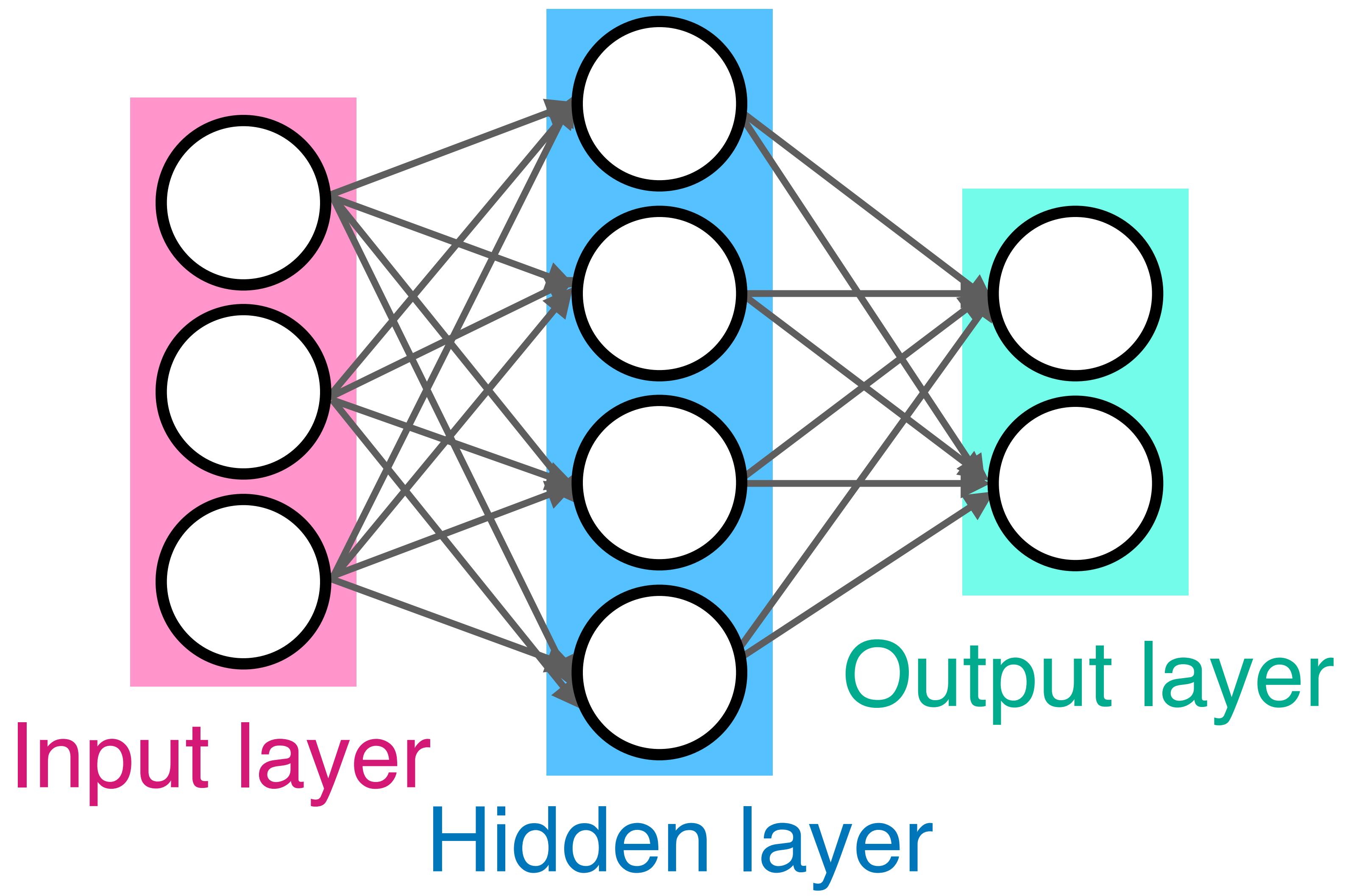
ELU

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



ReLU is a good default choice for most problems

Neural Net in <20 lines!



Initialize weights and data (lines 4–6)

Compute loss (Sigmoid activation, L2 loss) (lines 8–10)

Compute gradients (lines 11–14)

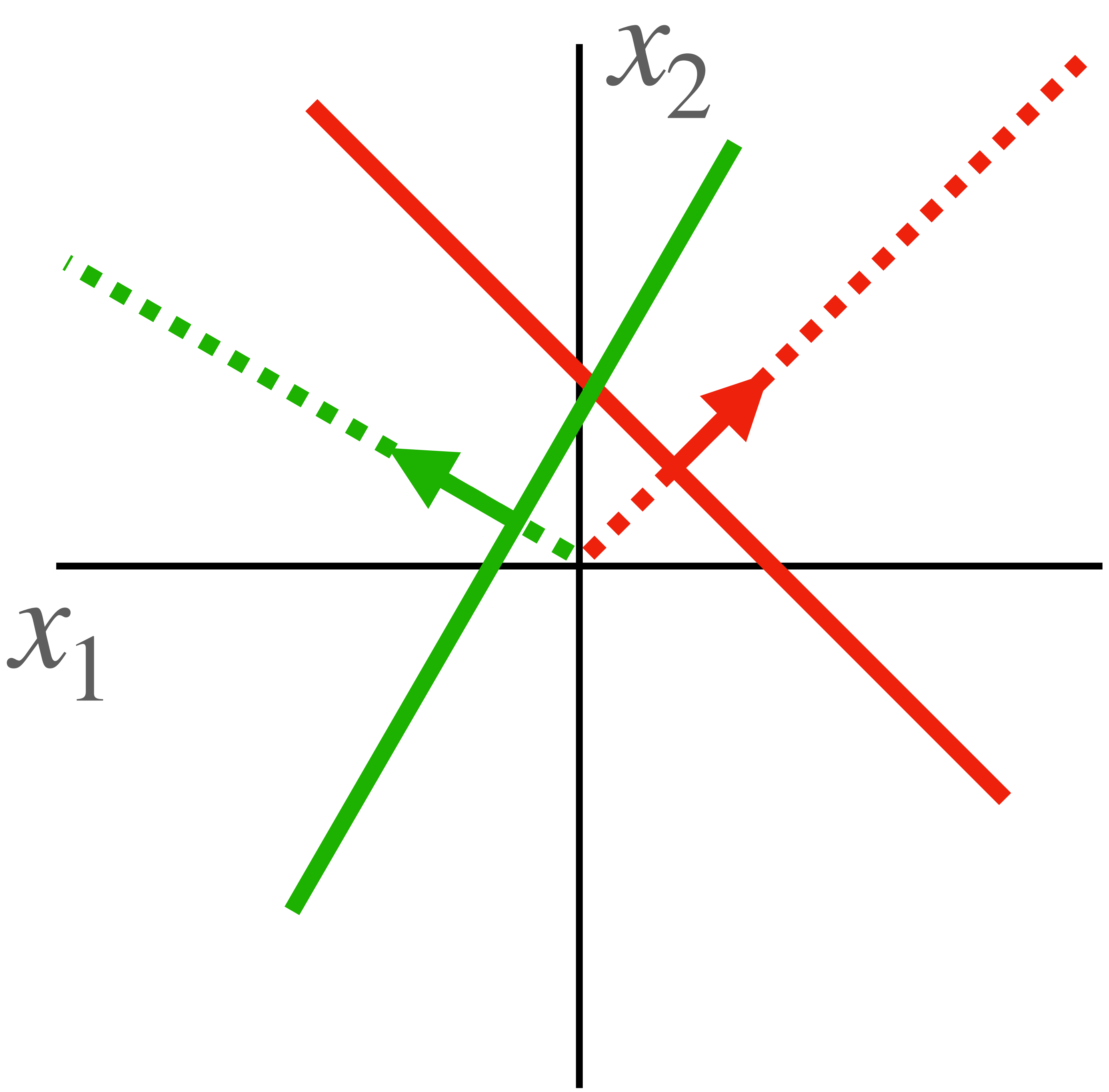
SGD step (lines 15–16)

```python
import numpy as np
from numpy.random import randn

N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)
for t in range(10000):
  h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
  dy_pred = 2.0 * (y_pred - y)
  dw2 = h.T.dot(dy_pred)
  dh = dy_pred.dot(w2.T)
  dw1 = x.T.dot(dh * h * (1 - h))
  w1 -= 1e-4 * dw1
  w2 -= 1e-4 * dw2
```

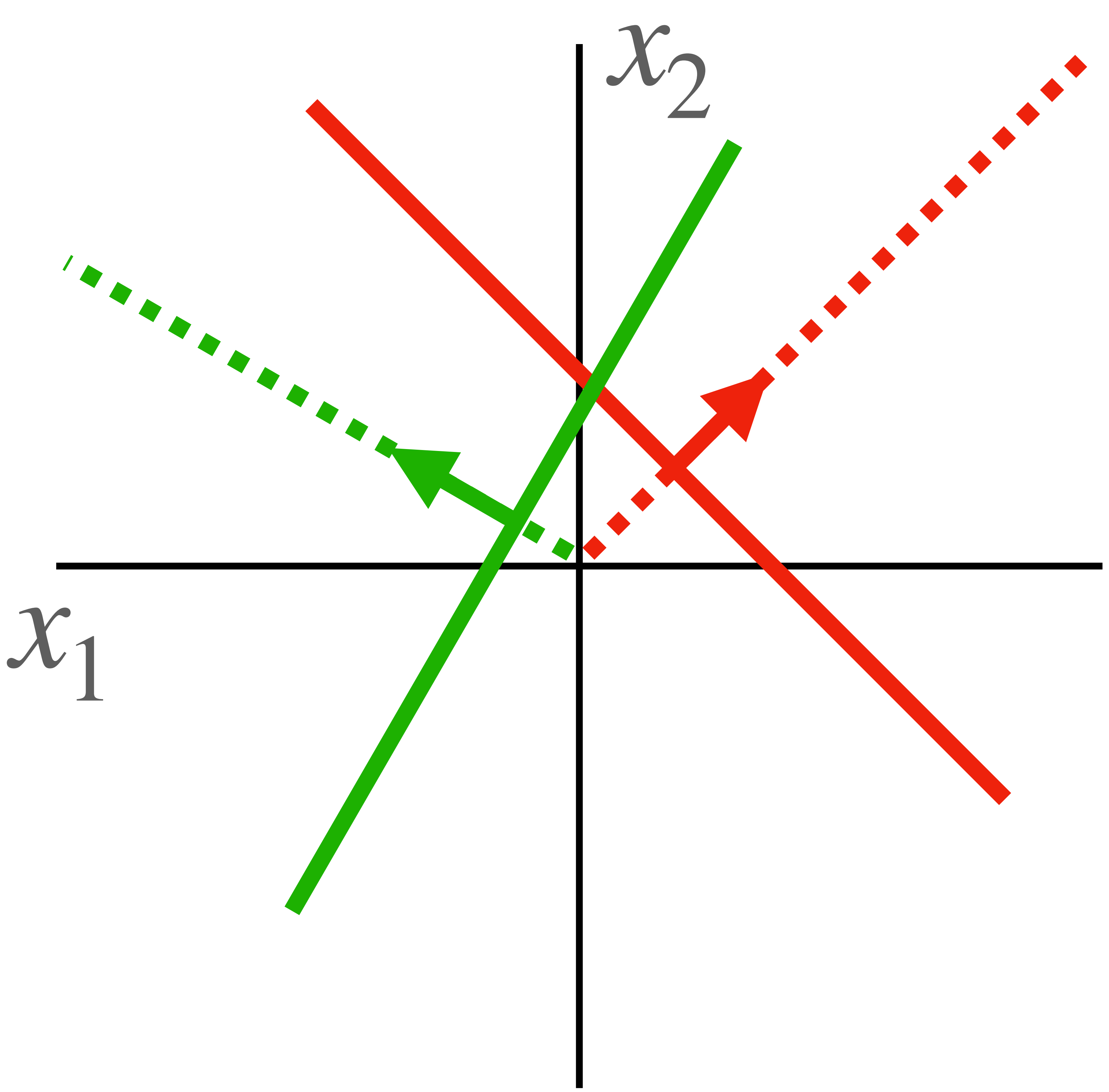
Space Warping

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



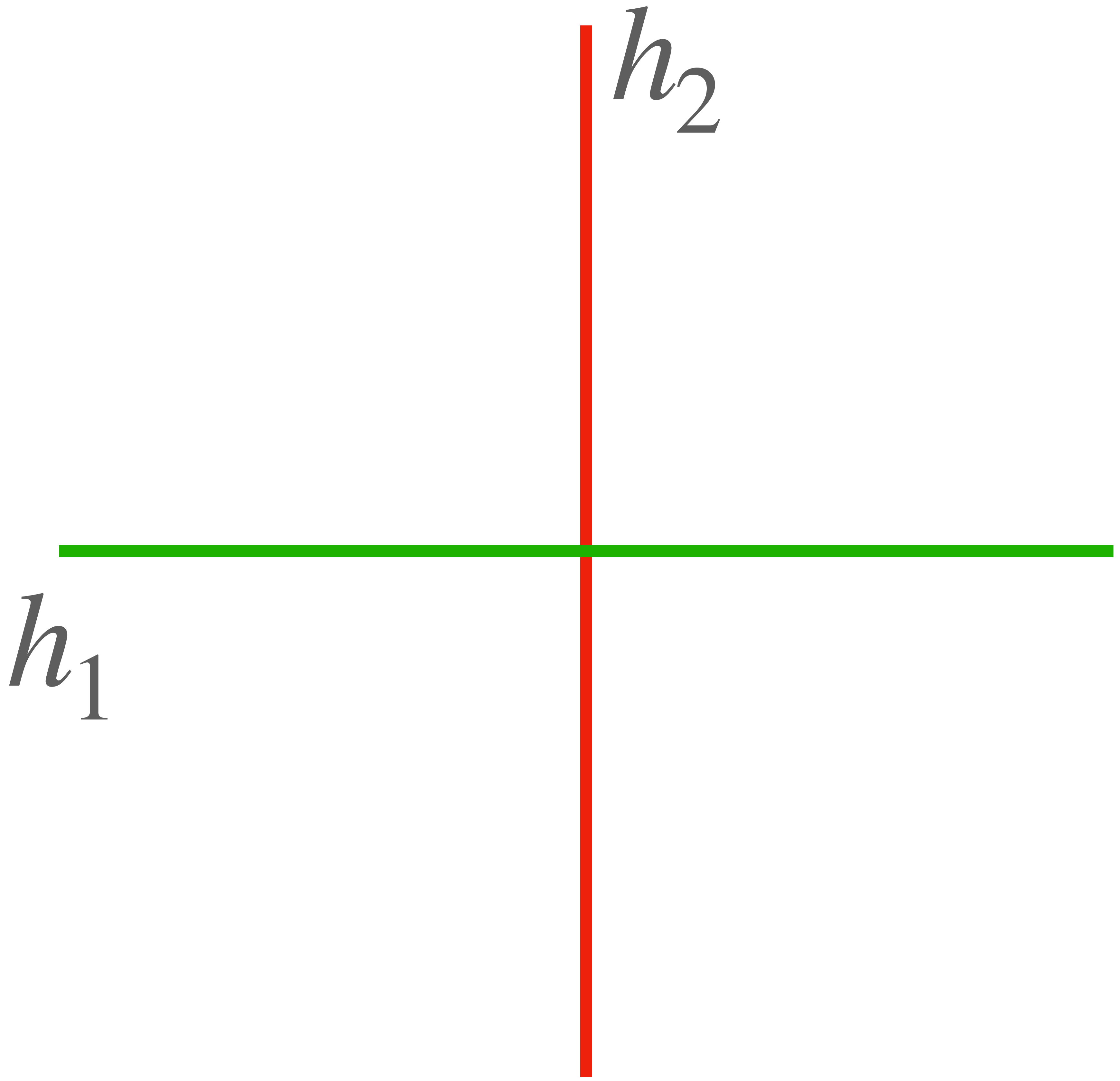
Space Warping

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



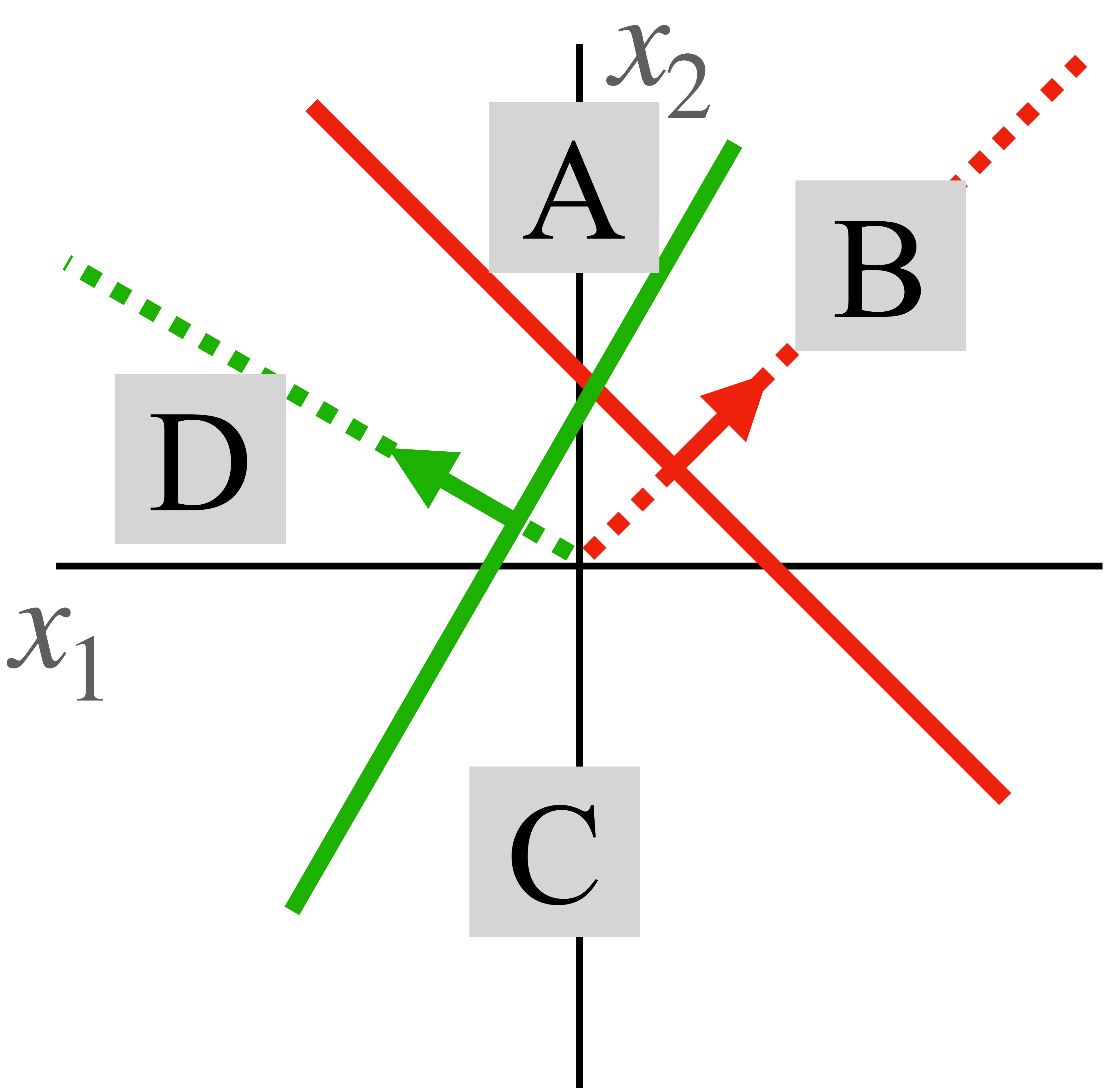
Feature transform:

$$h = Wx + b$$

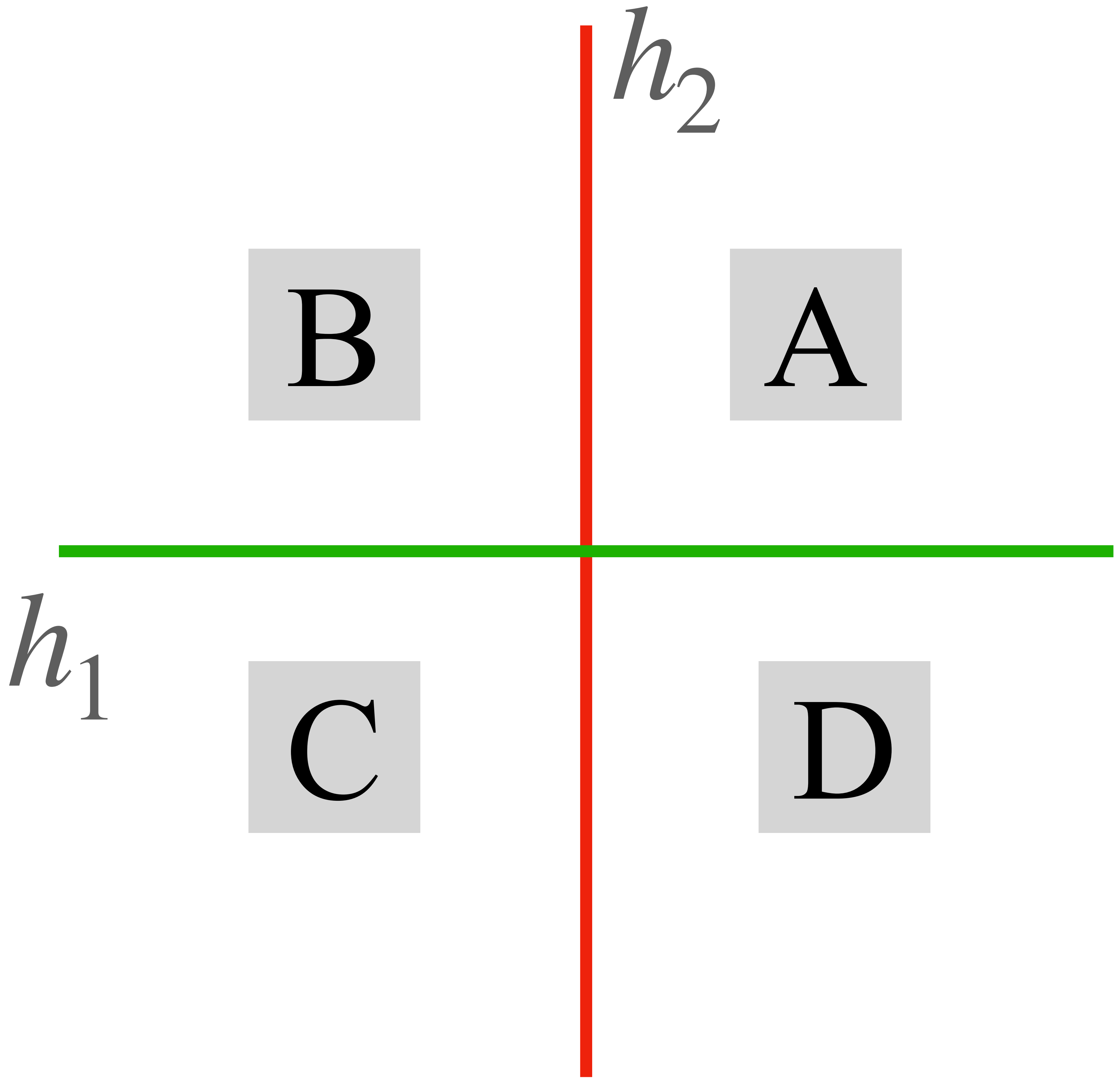


Space Warping

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

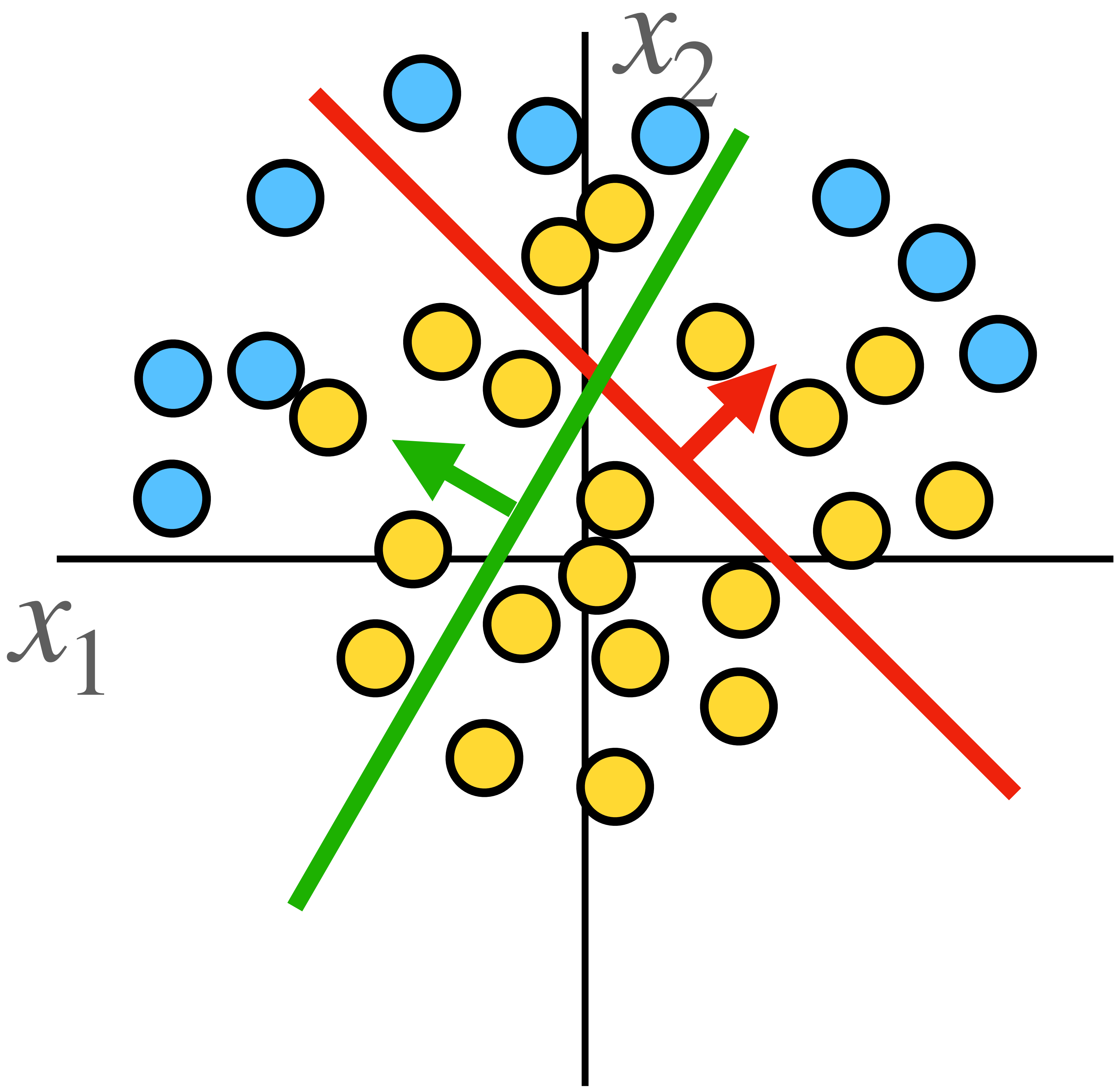


Feature transform:
$h = Wx + b$



Space Warping

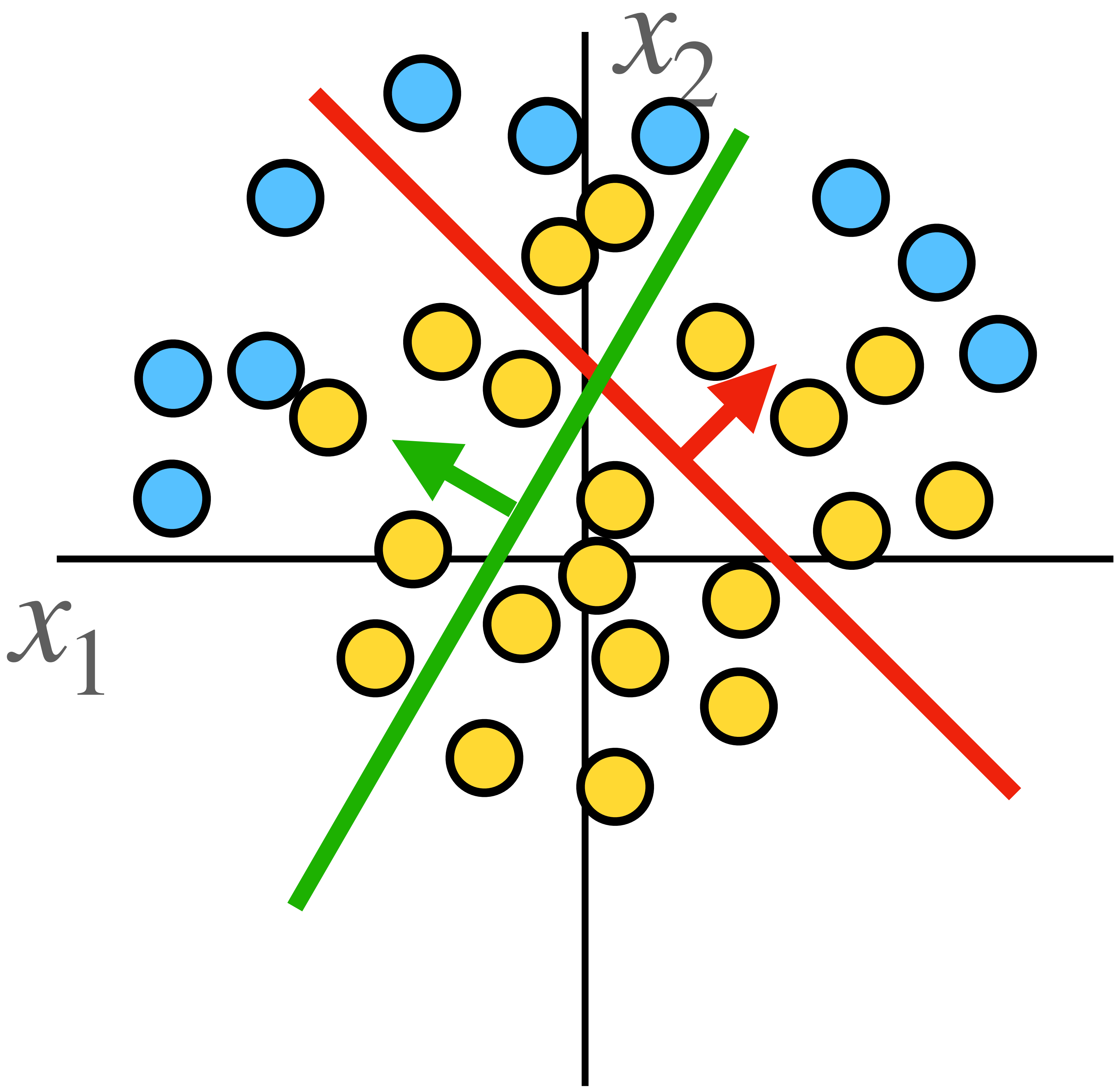
Points not linearly separable in original space



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

Space Warping

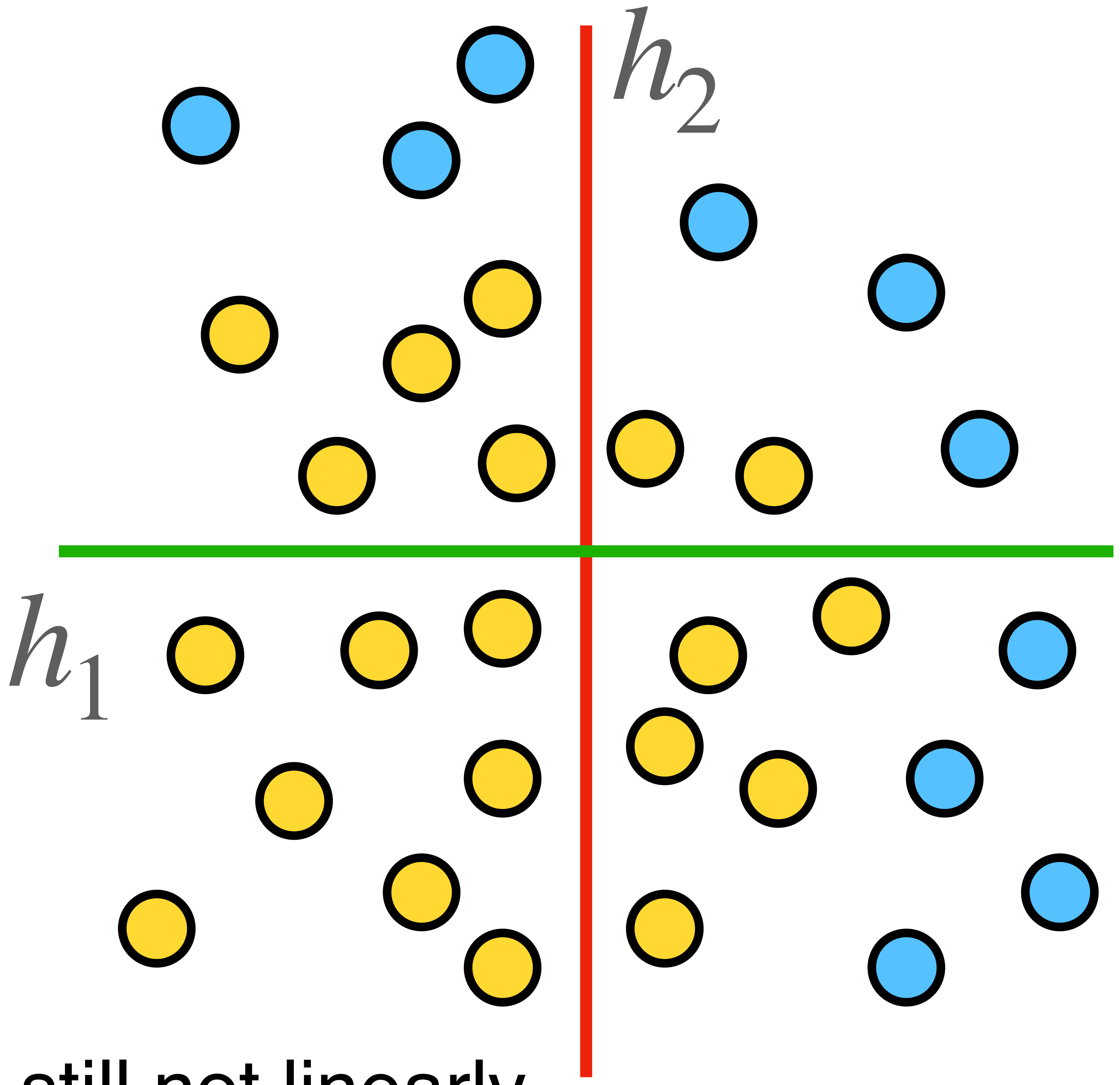
Points not linearly separable in original space



Feature transform:

$$h = Wx + b$$

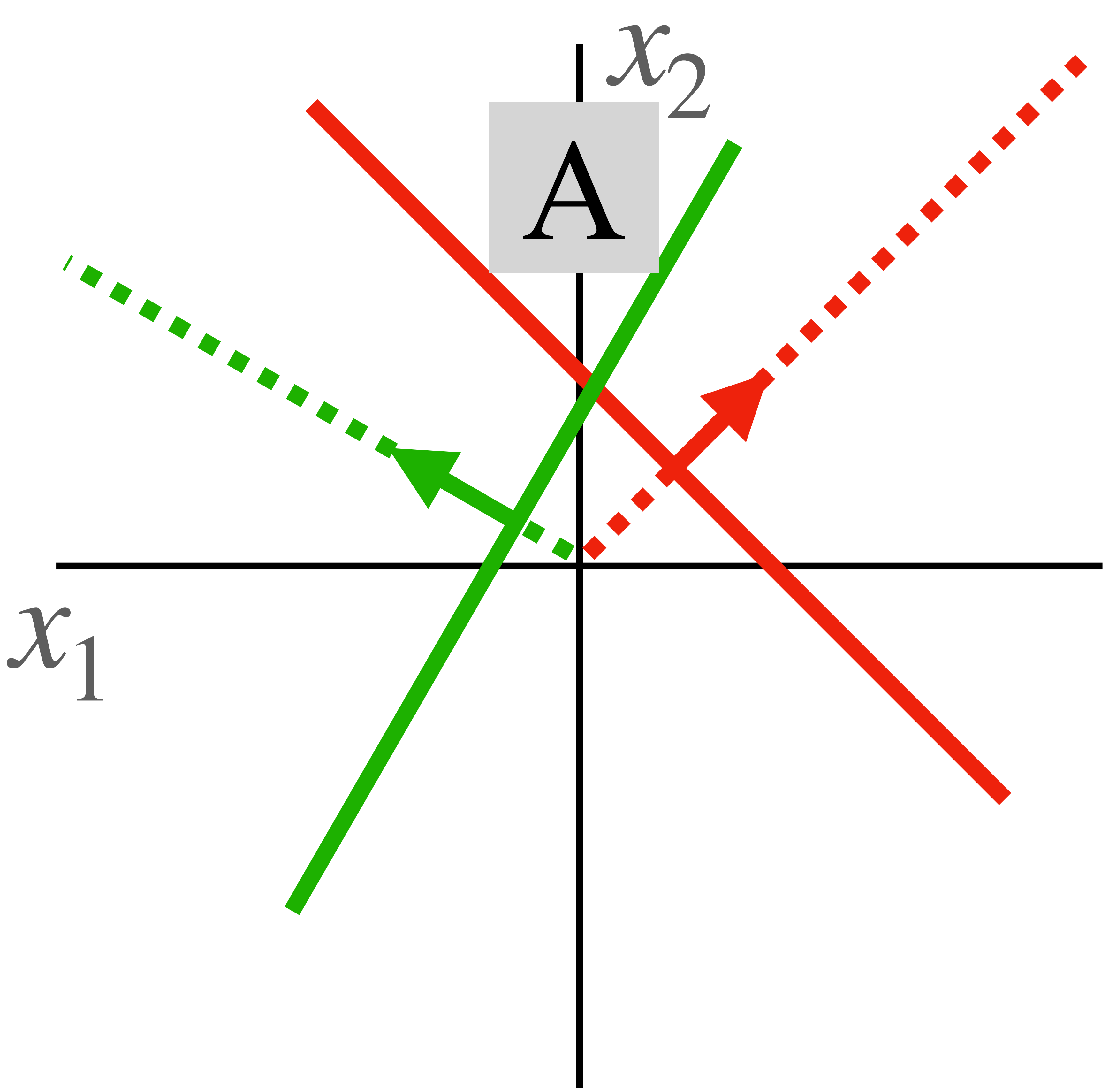
Consider a linear transform: $h = Wx + b$ where x, b, h are each 2-dimensional



Points still not linearly separable in feature space

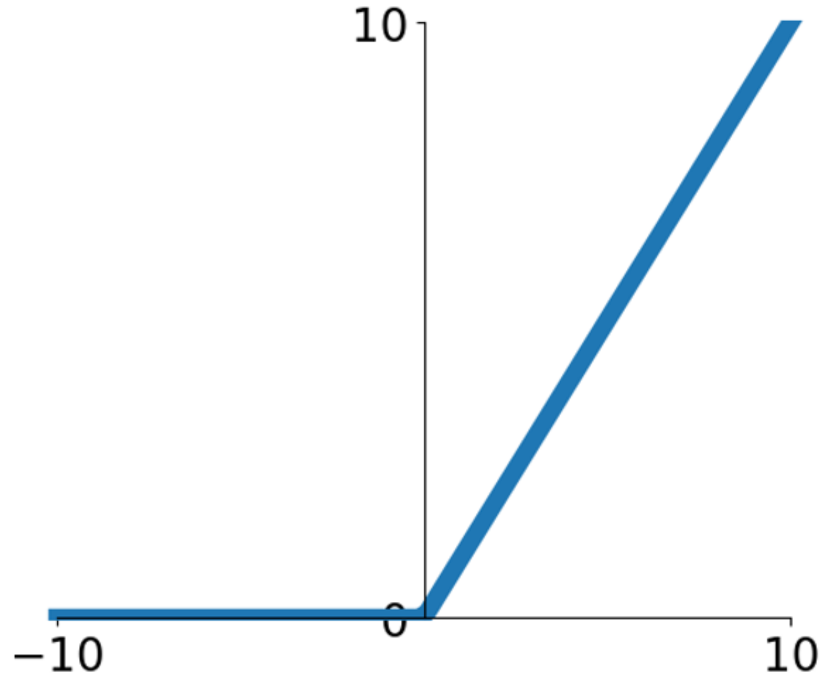
Space Warping

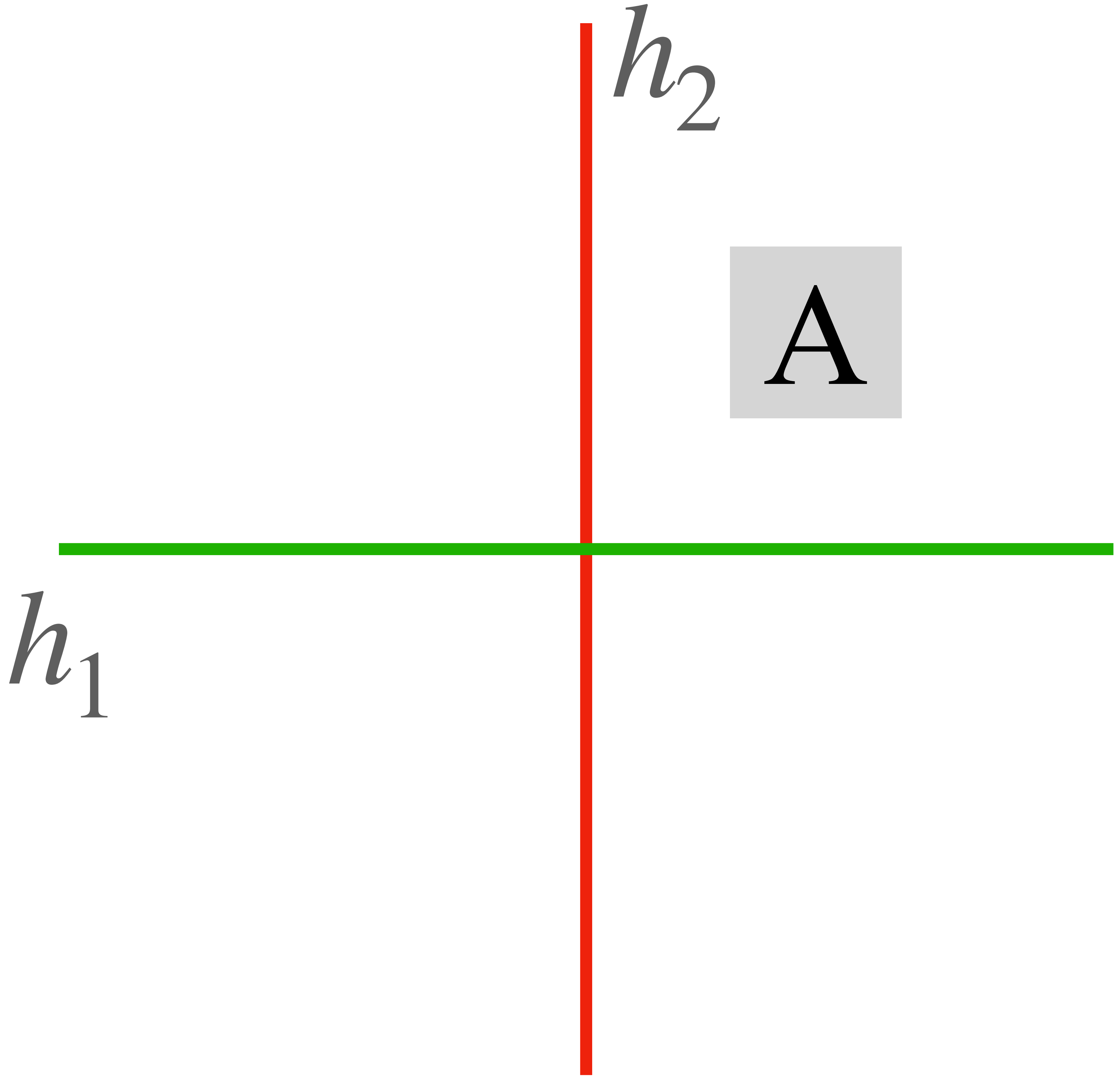
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

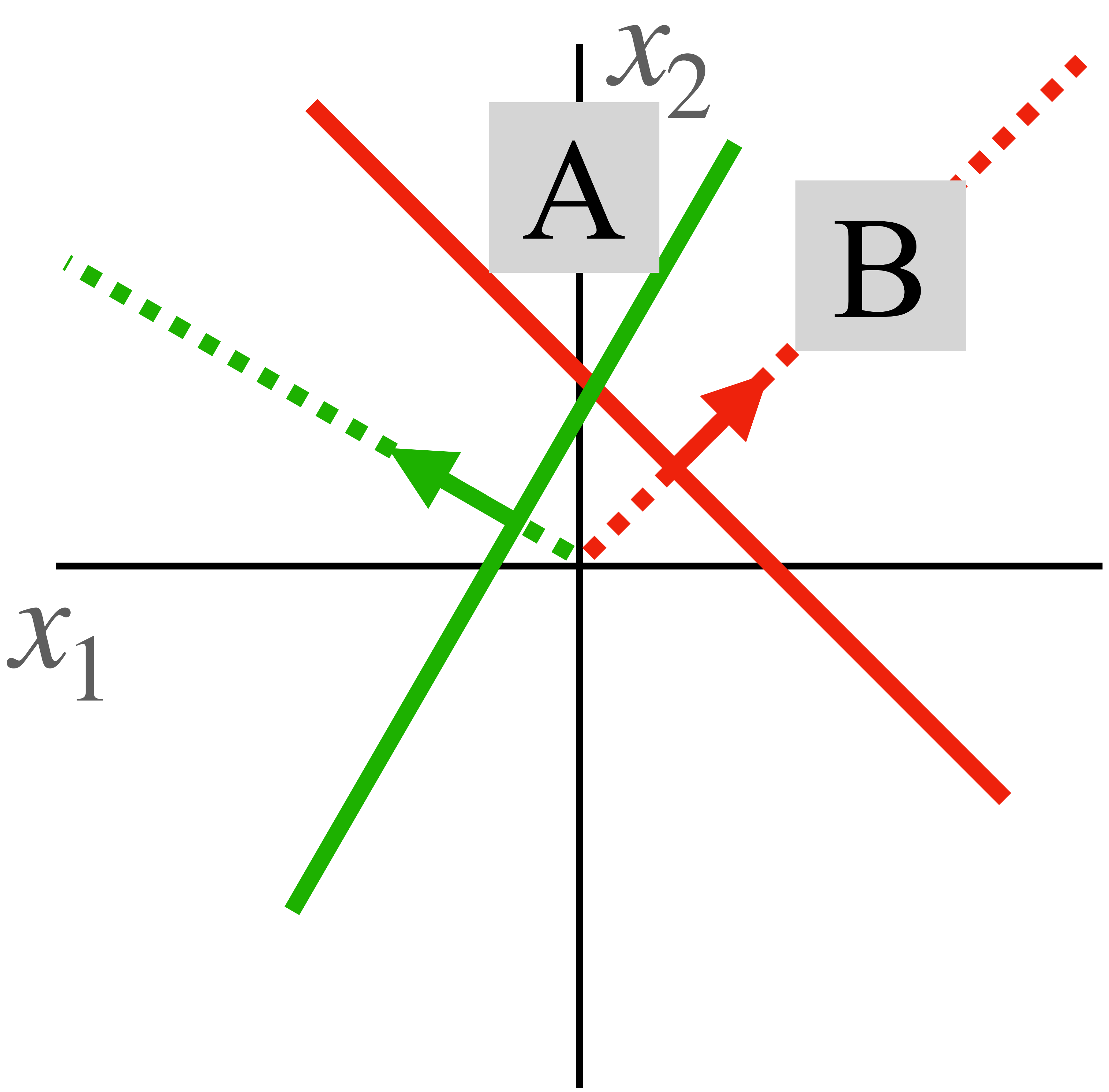
$h = ReLU(Wx + b)$





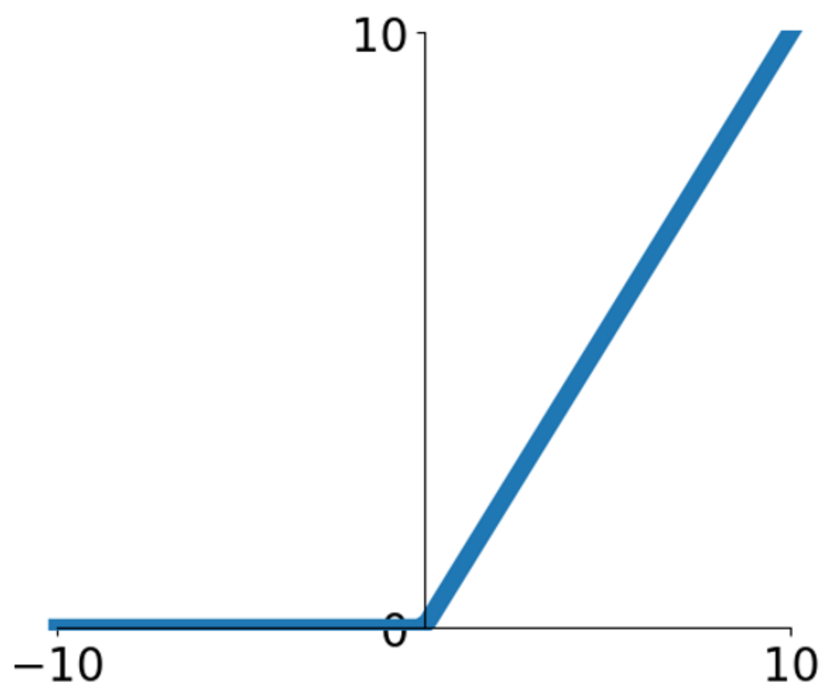
Space Warping

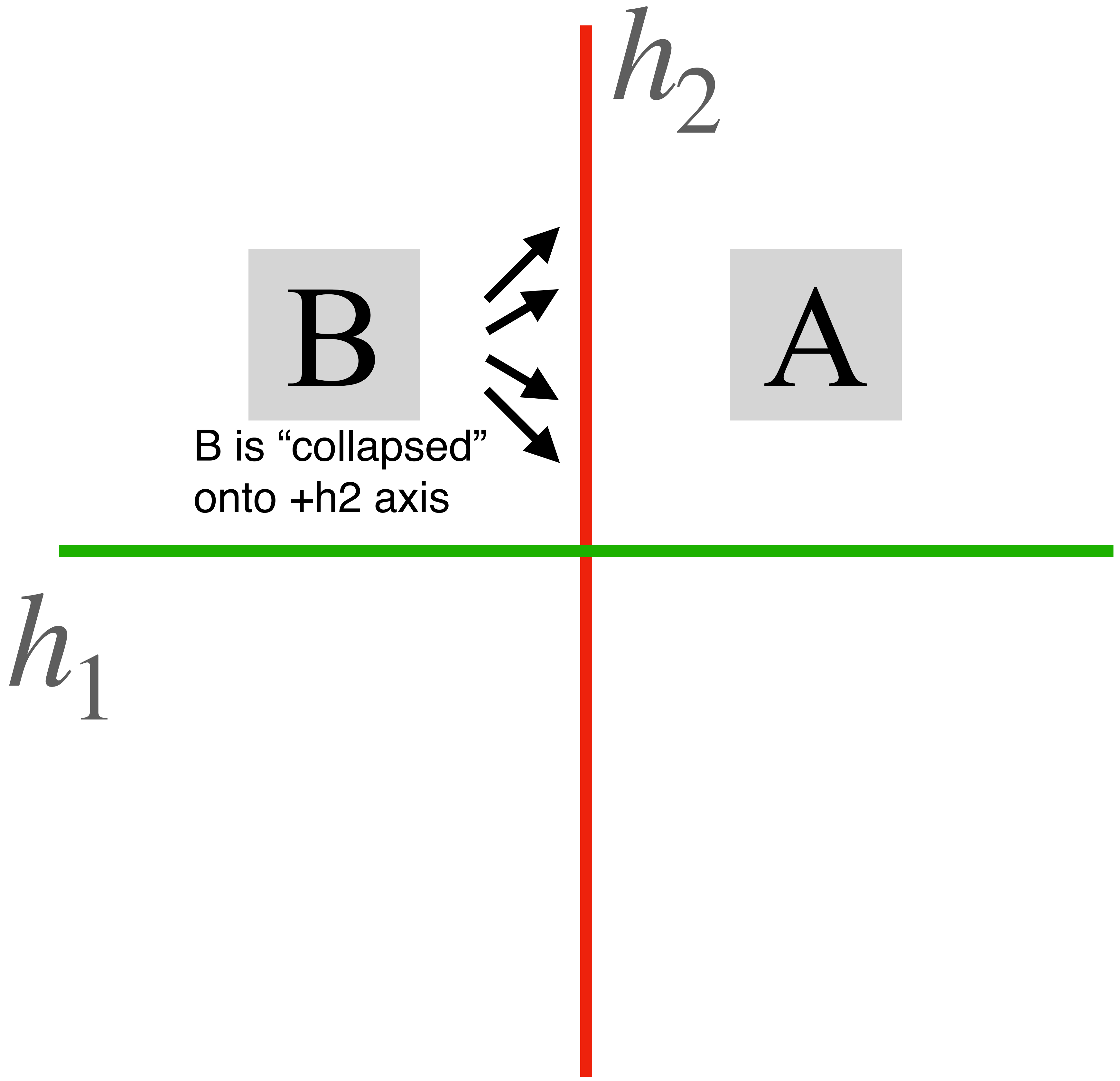
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

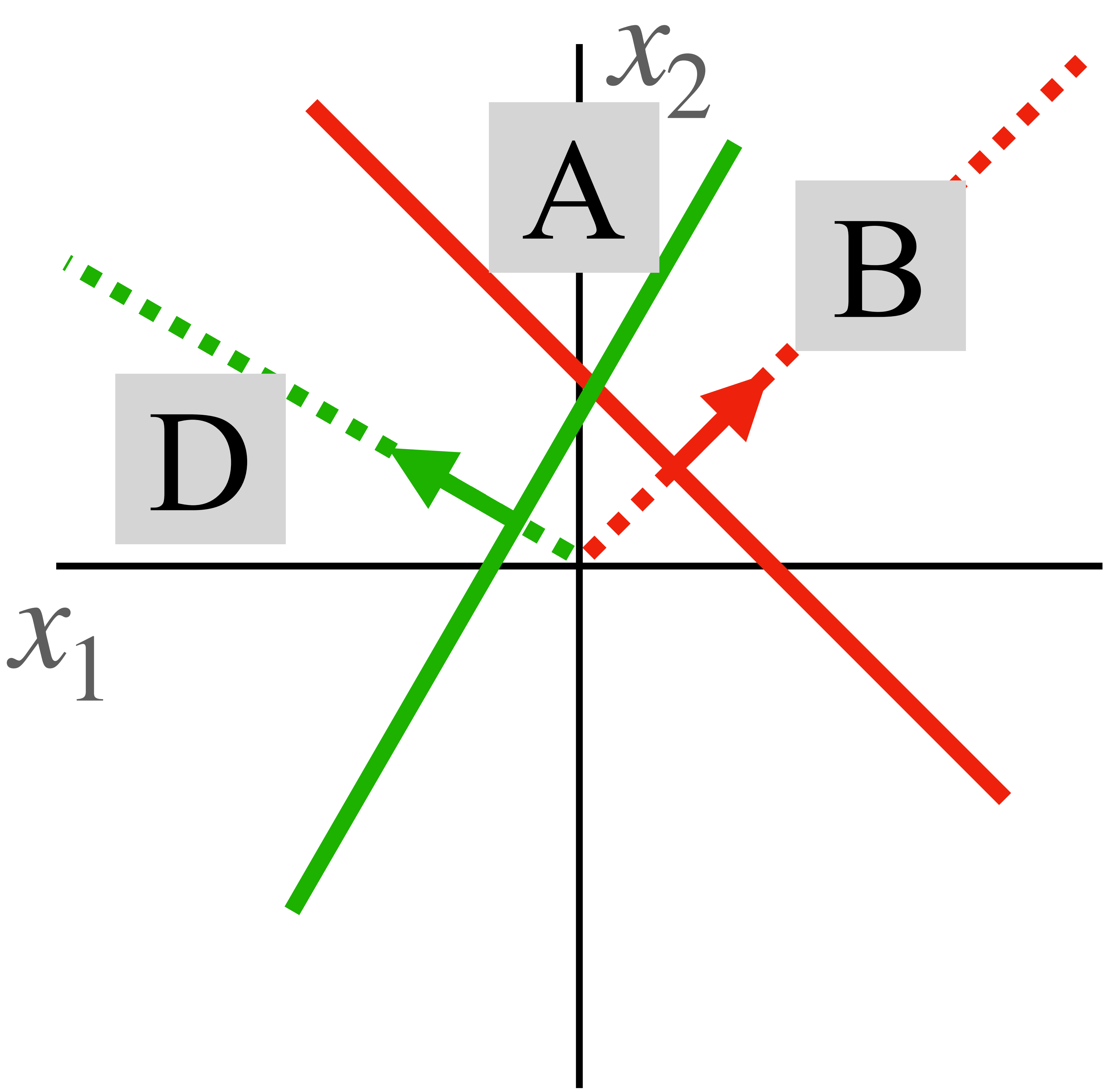
$$h = ReLU(Wx + b)$$





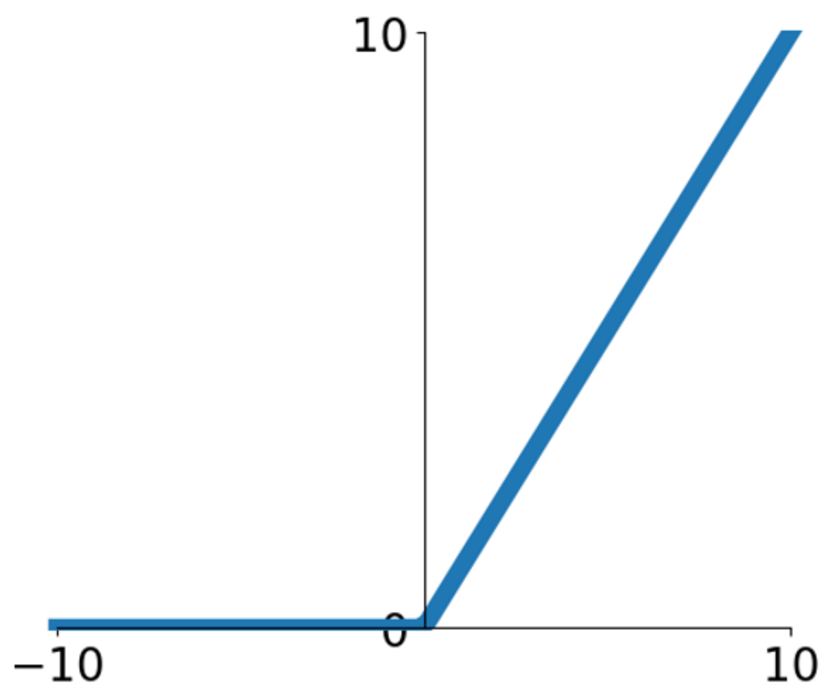
Space Warping

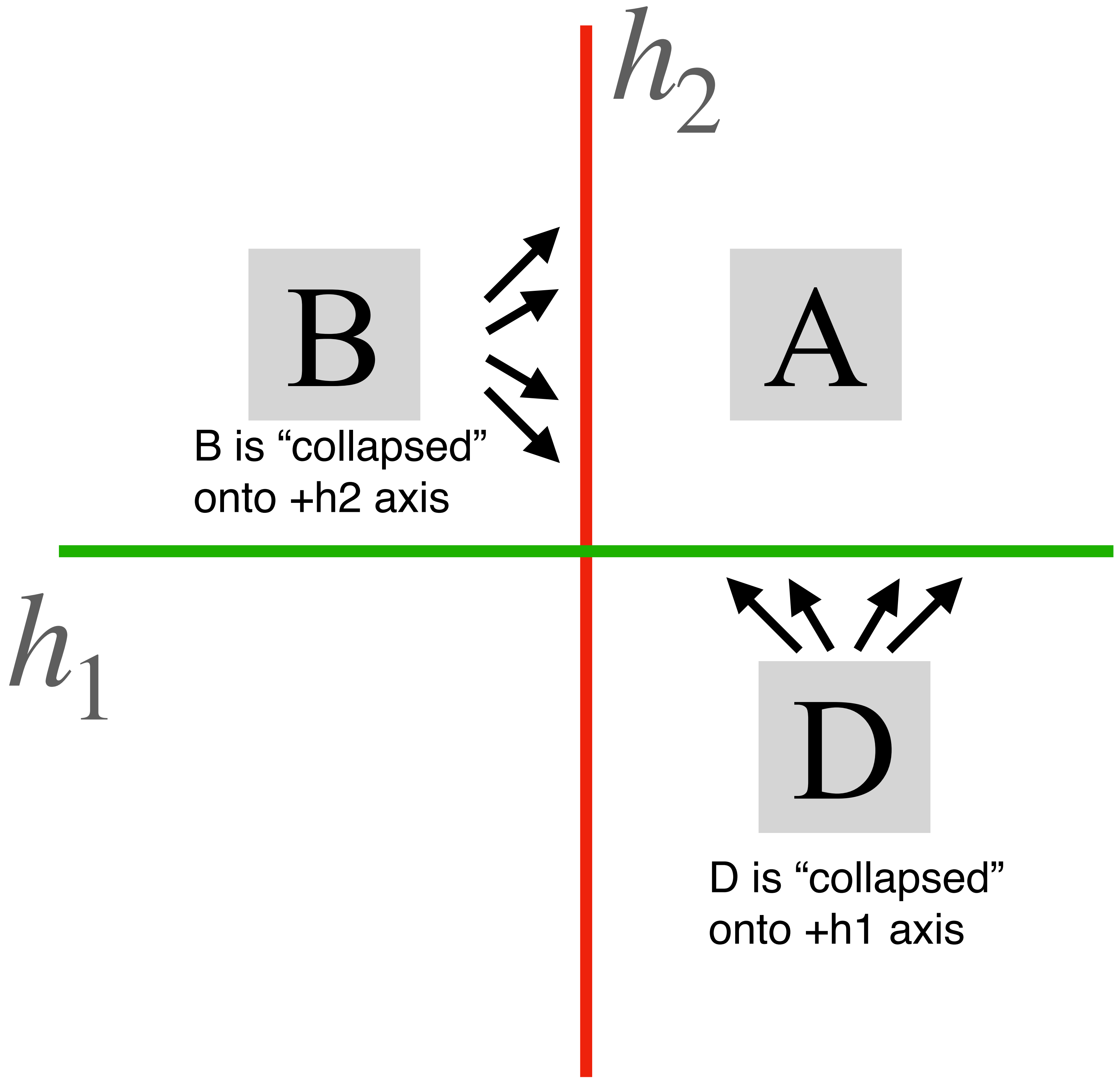
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

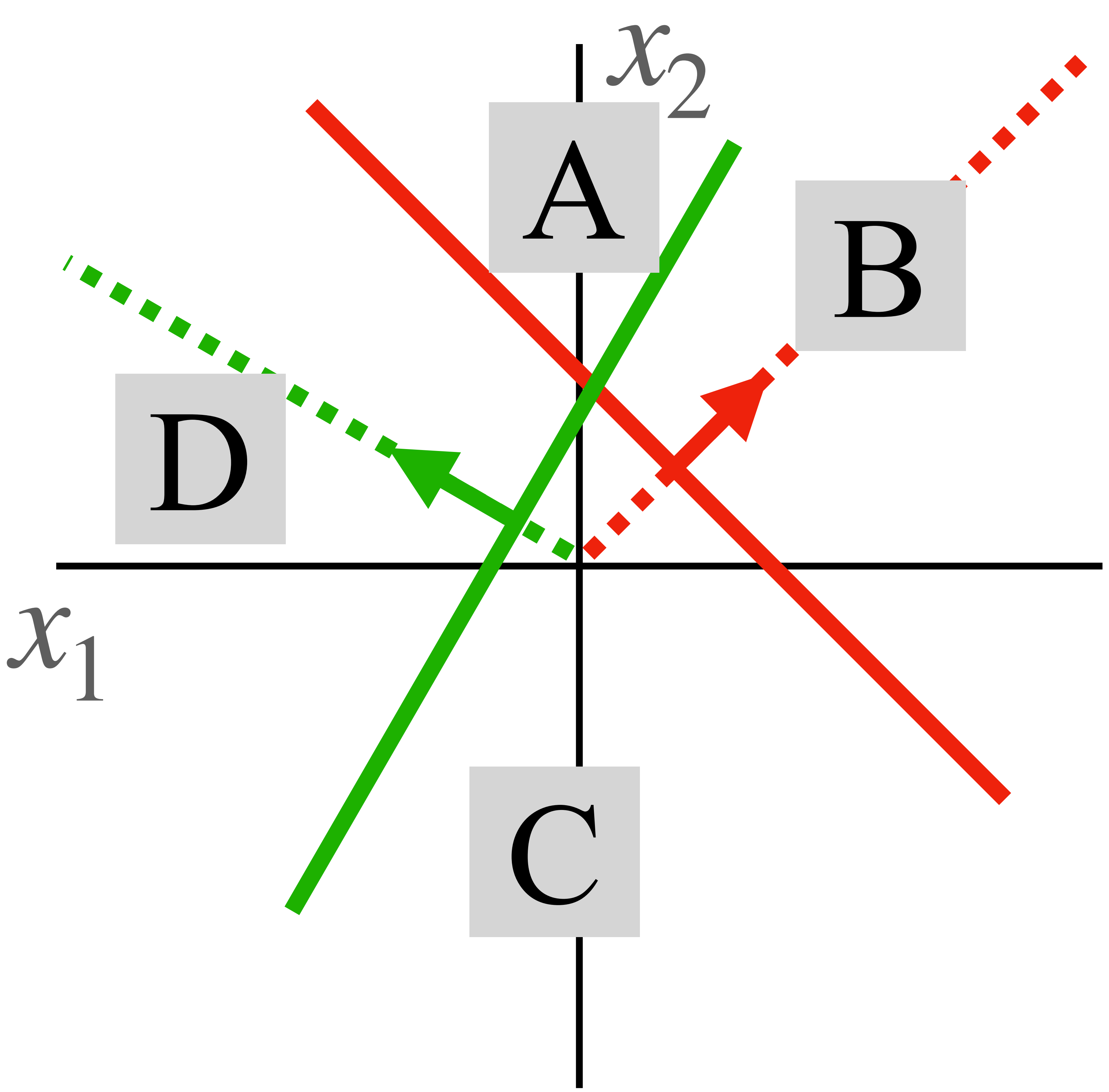
$h = ReLU(Wx + b)$





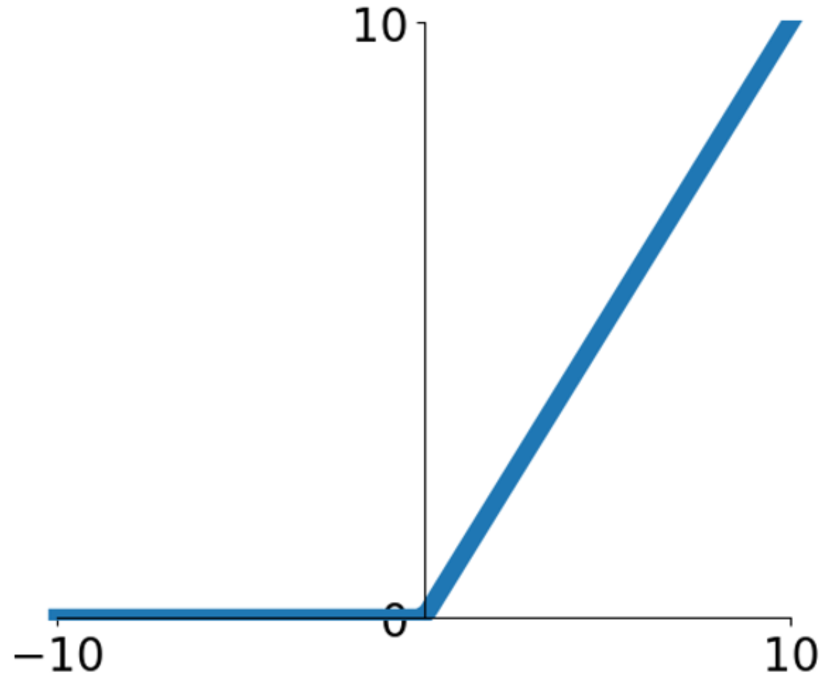
Space Warping

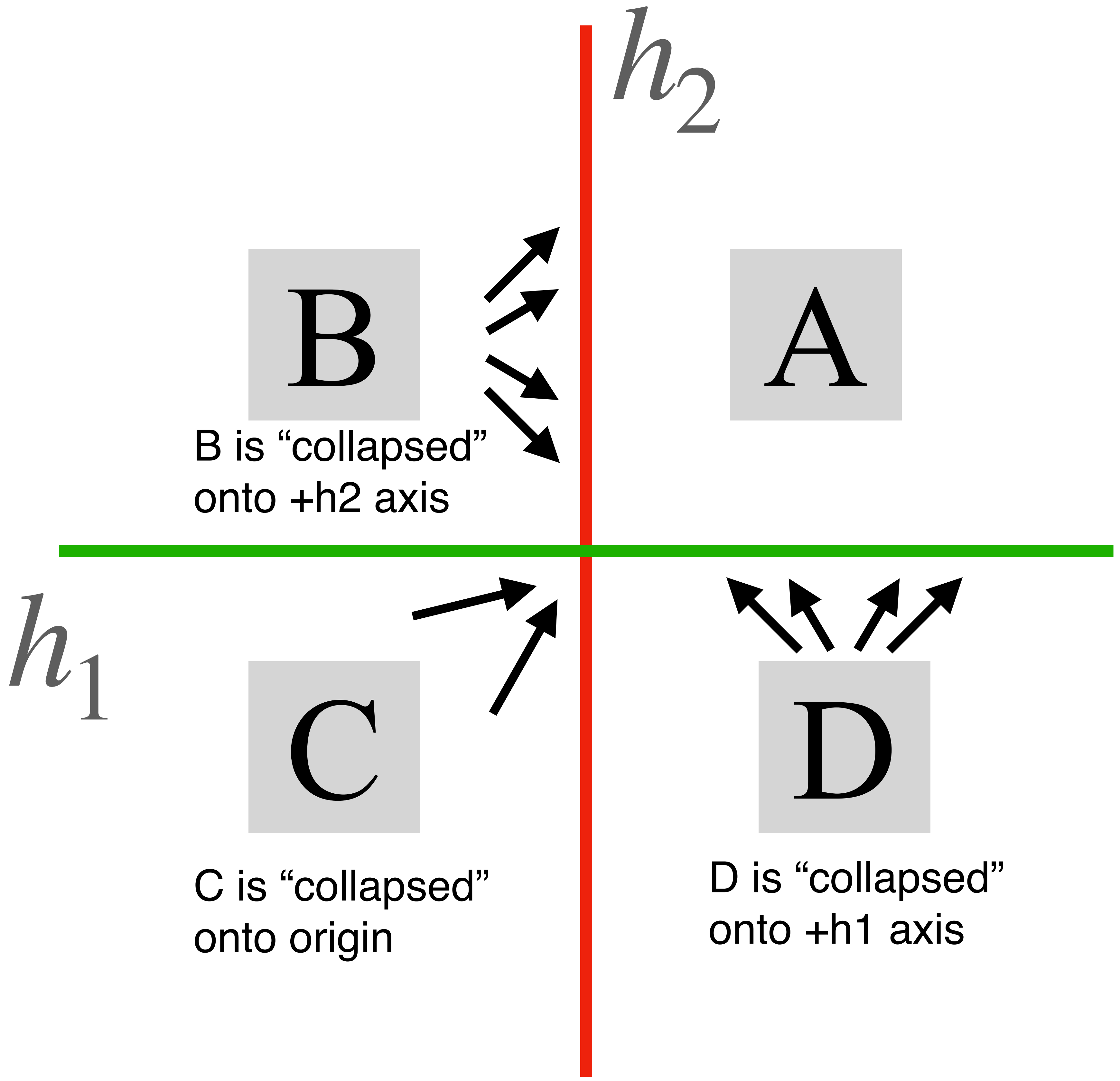
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

$h = ReLU(Wx + b)$

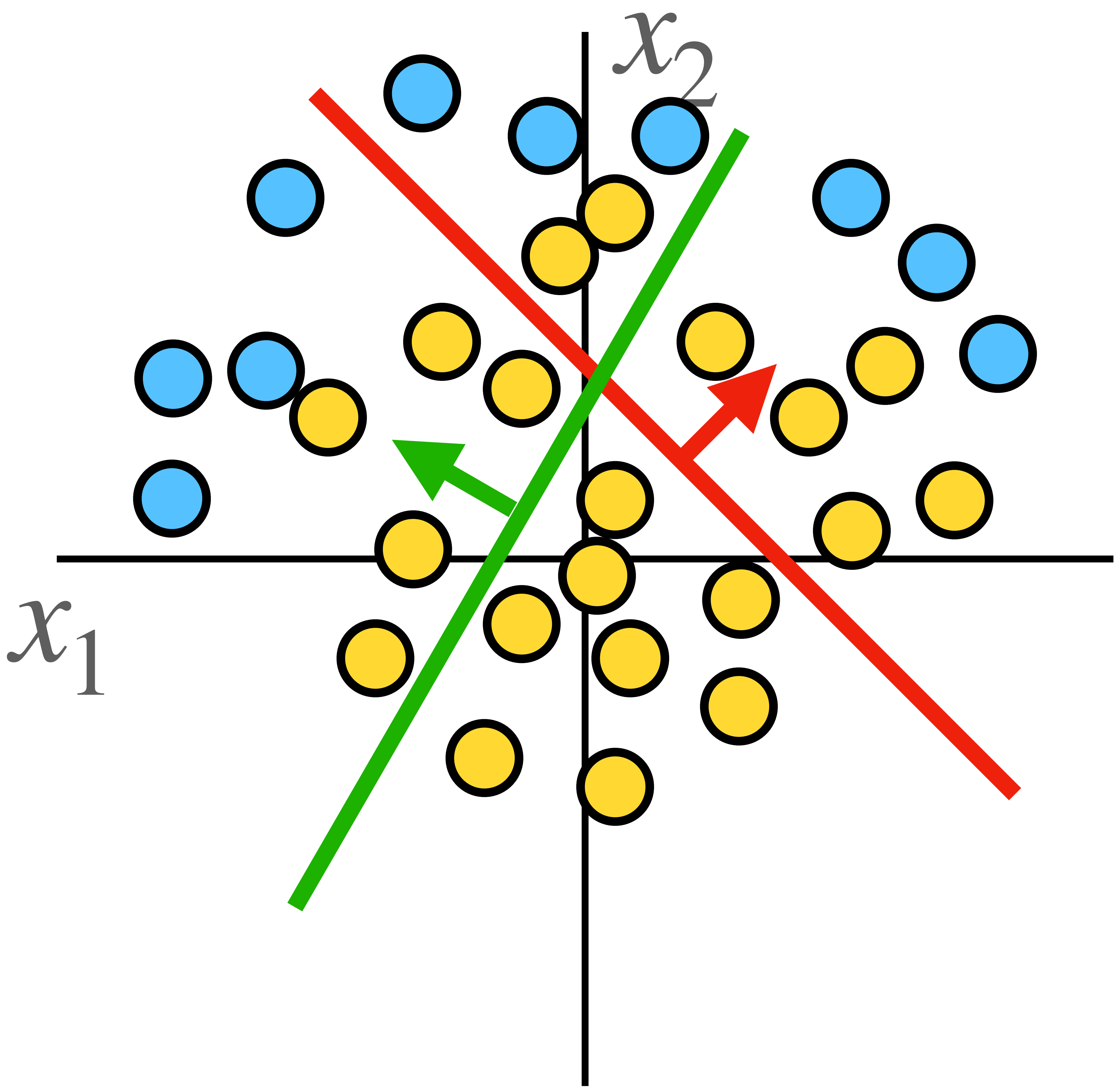




Space Warping

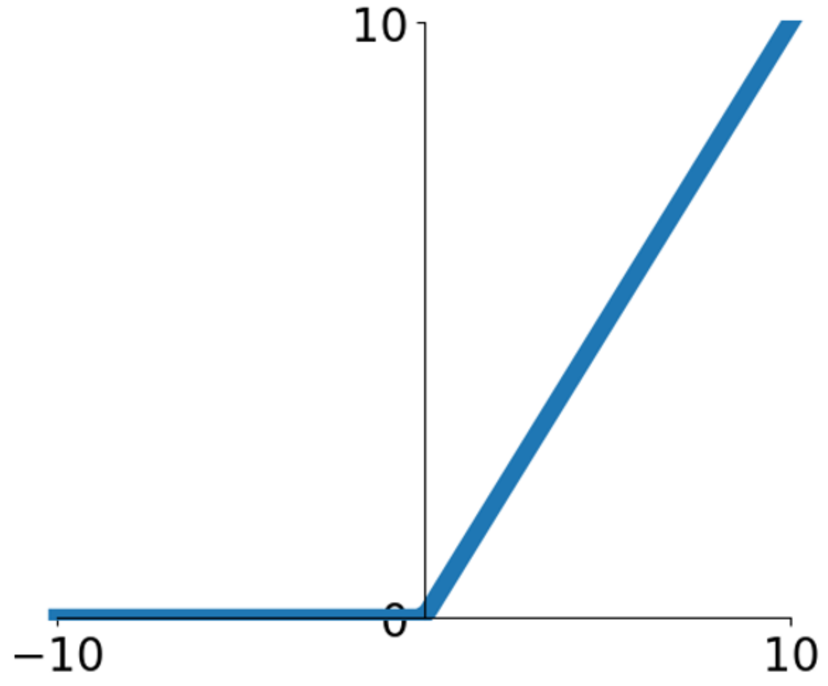
Points not linearly separable in original space

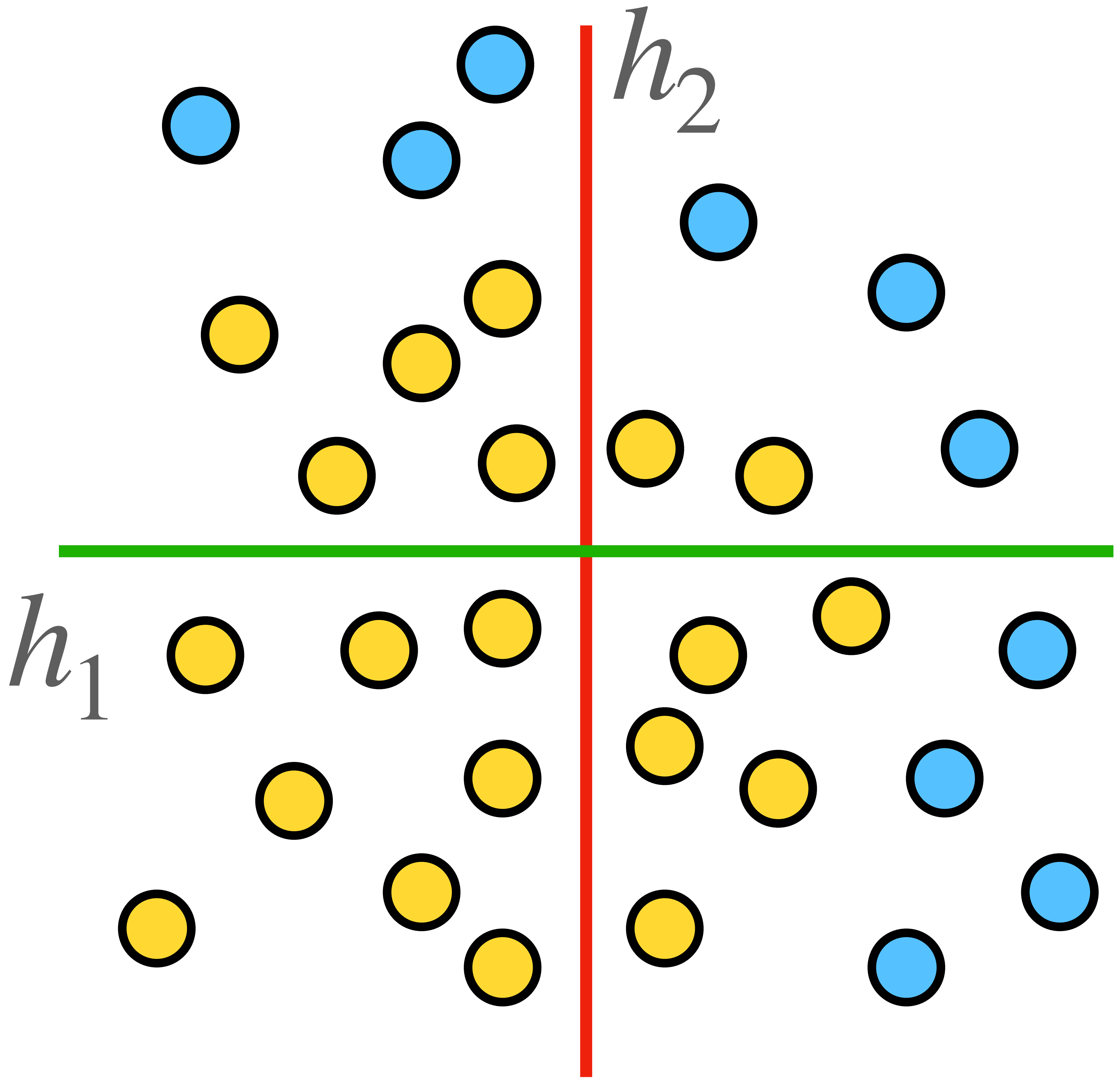
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

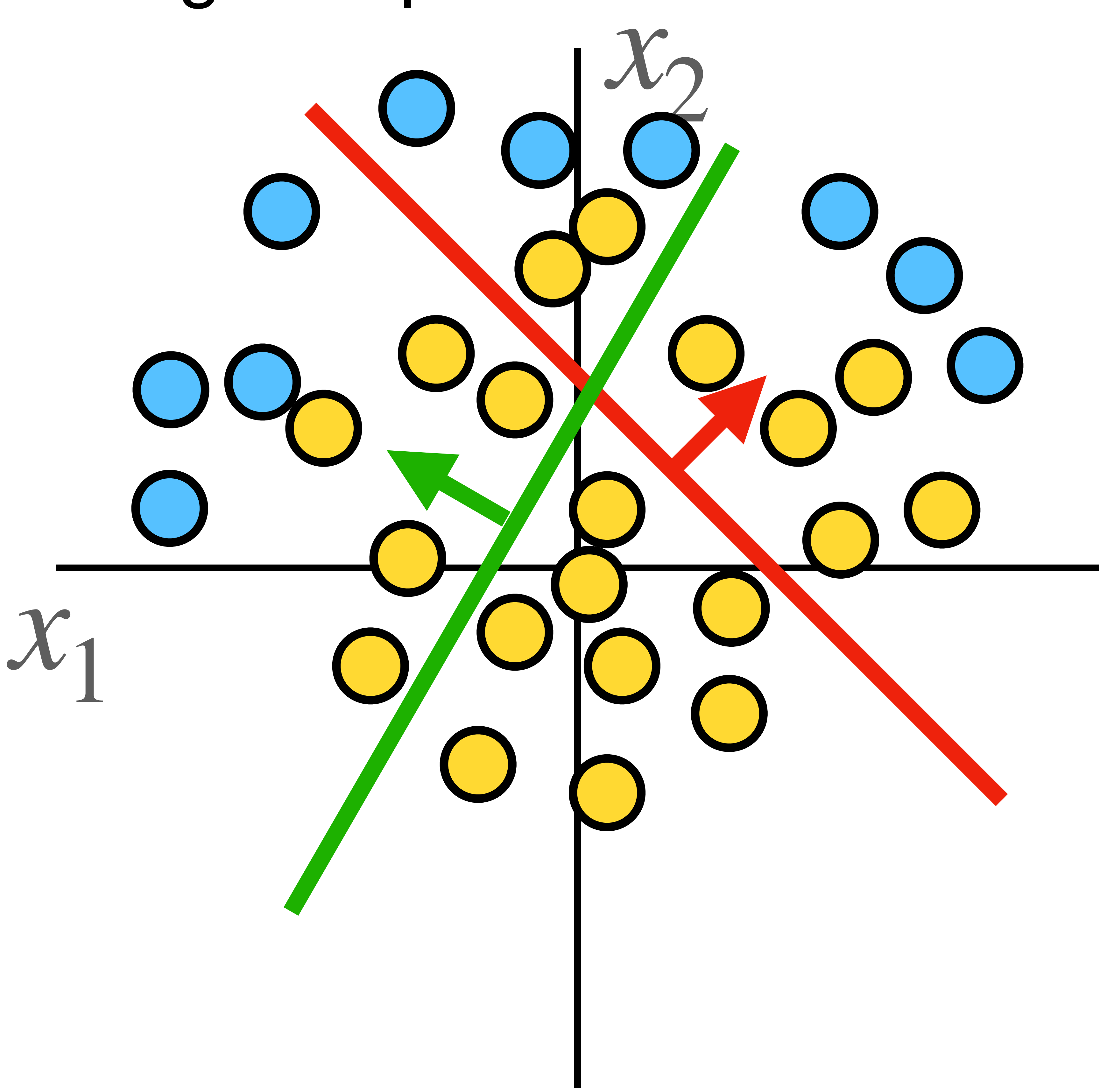
$h = ReLU(Wx + b)$





Space Warping

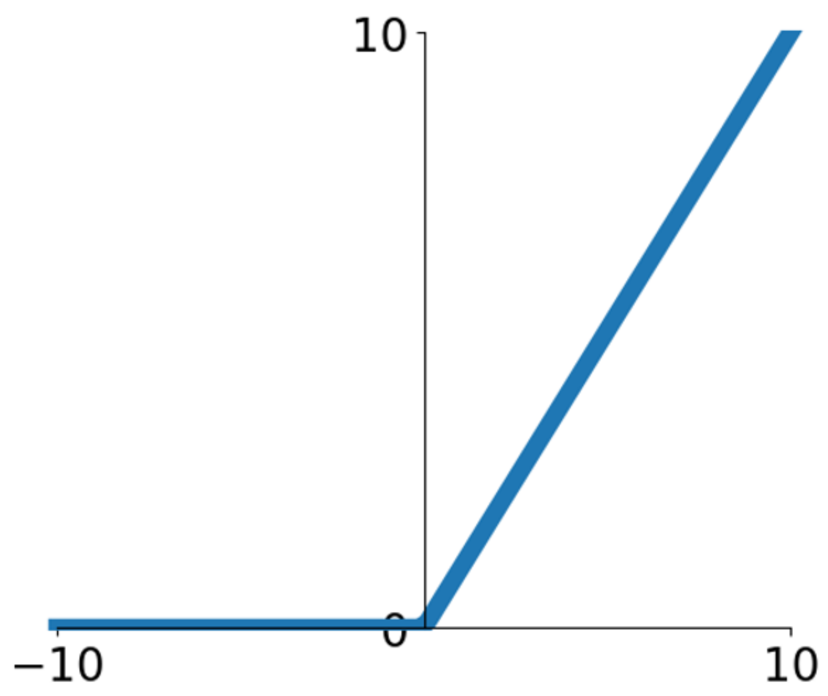
Points not linearly separable in original space

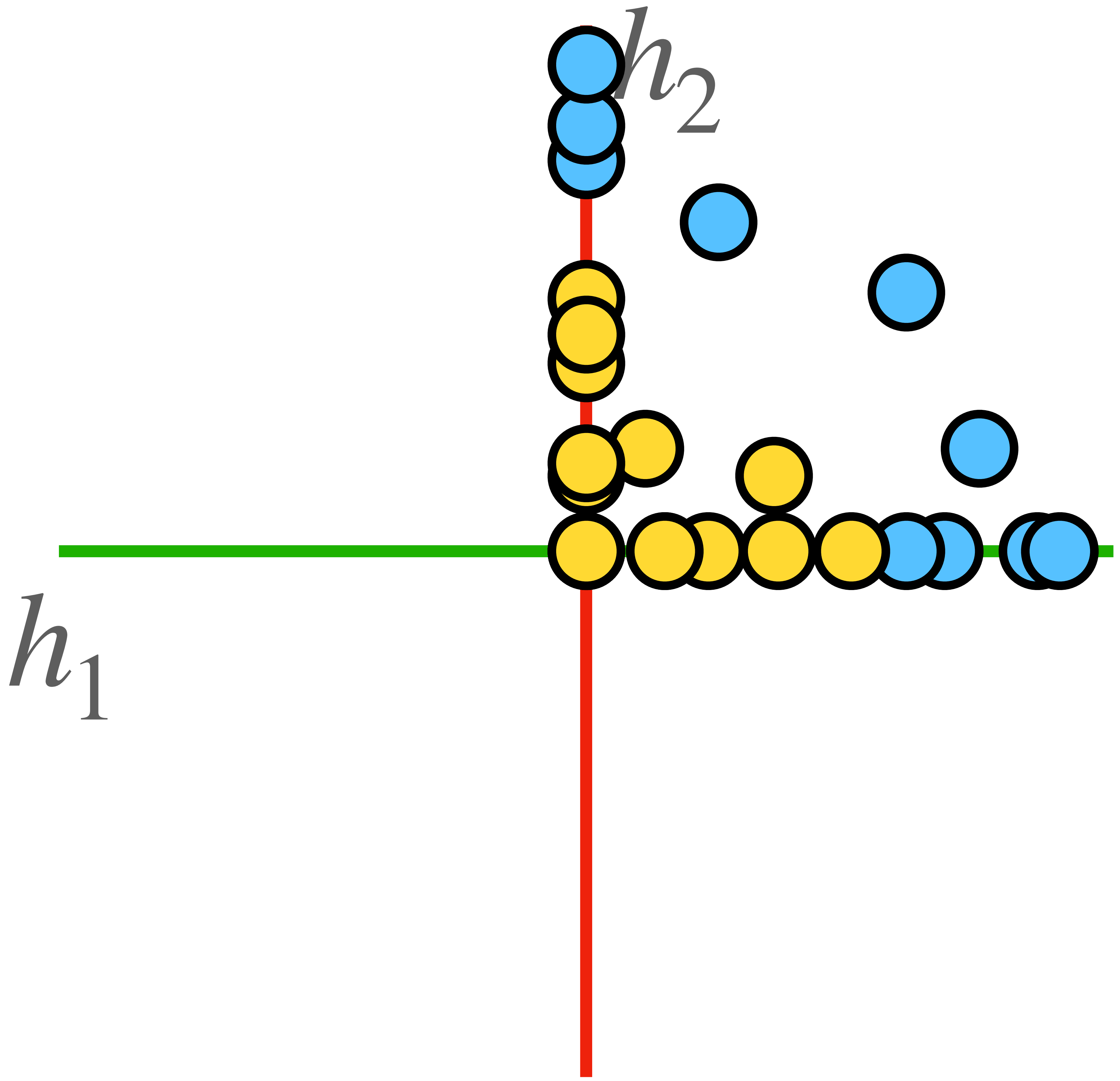


Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional

Feature transform:

$h = ReLU(Wx + b)$

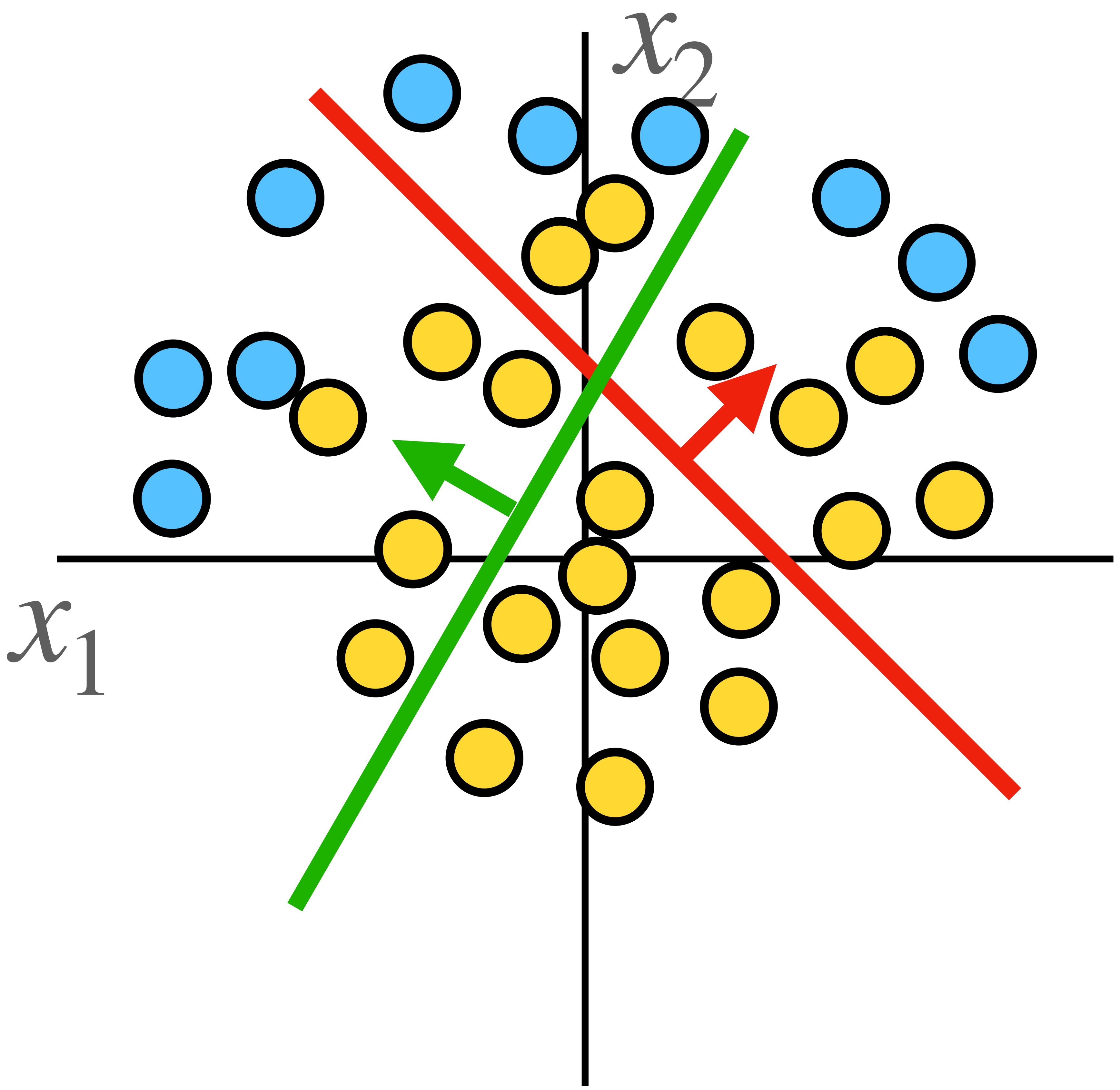




Space Warping

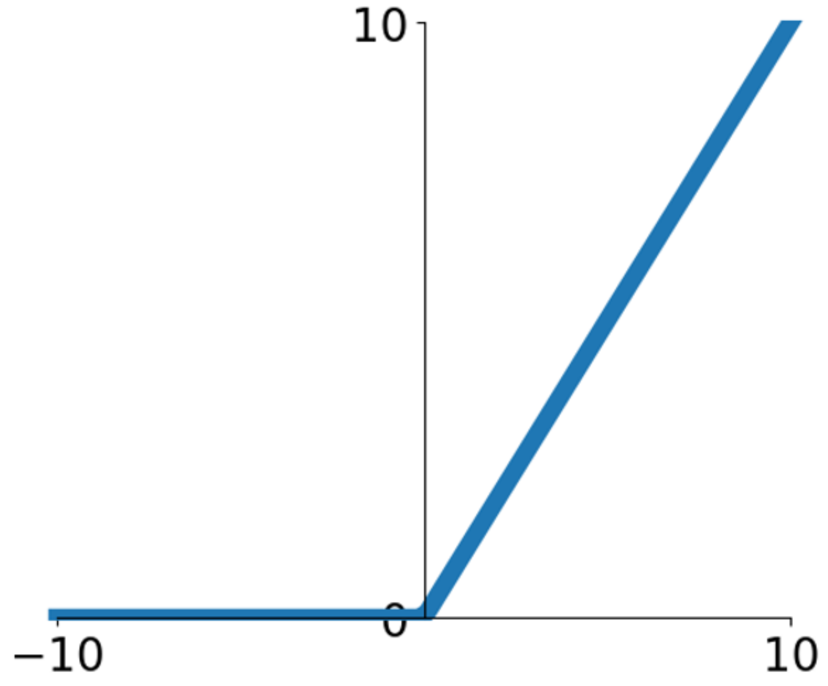
Points not linearly separable in original space

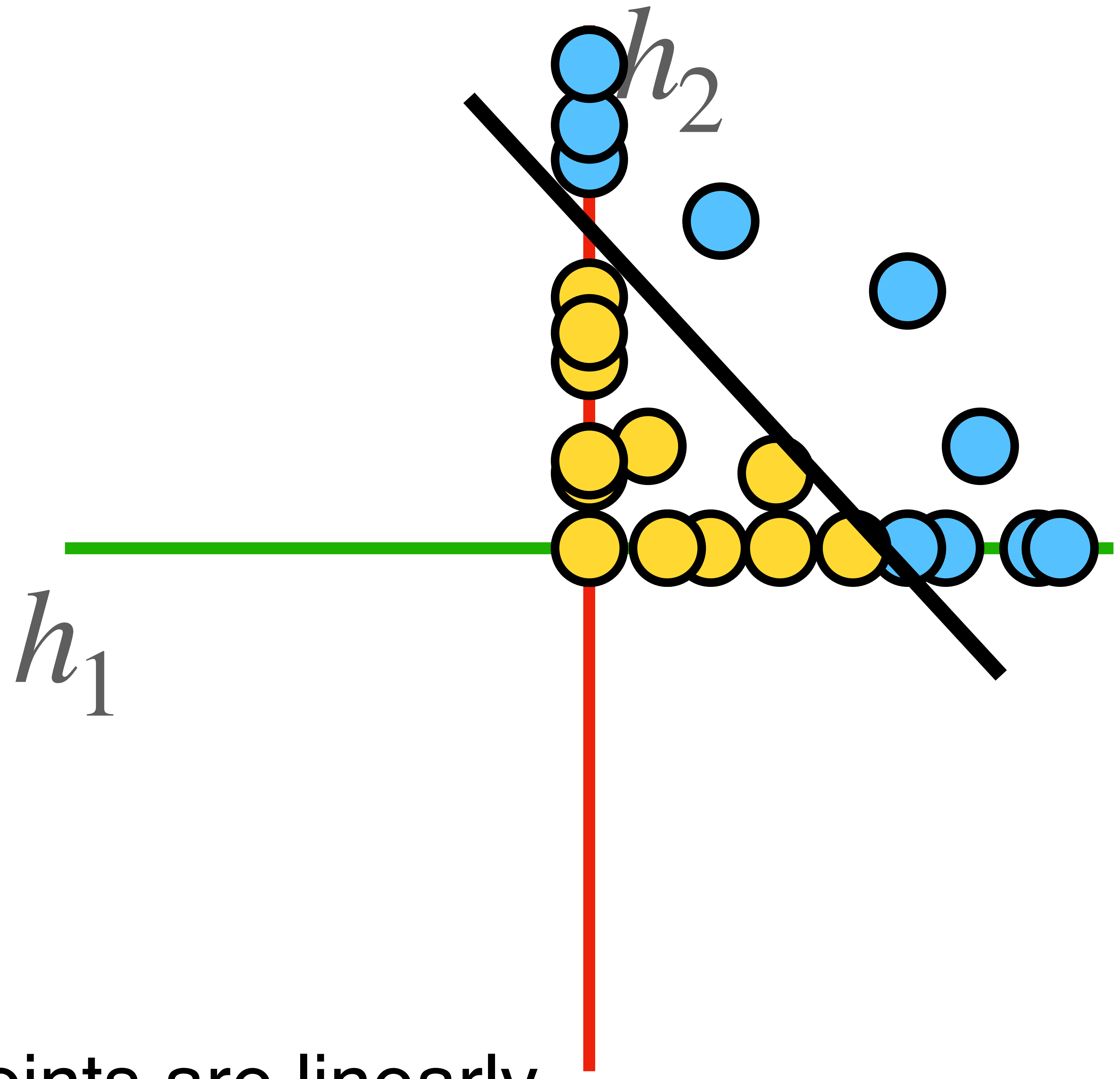
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

$h = ReLU(Wx + b)$



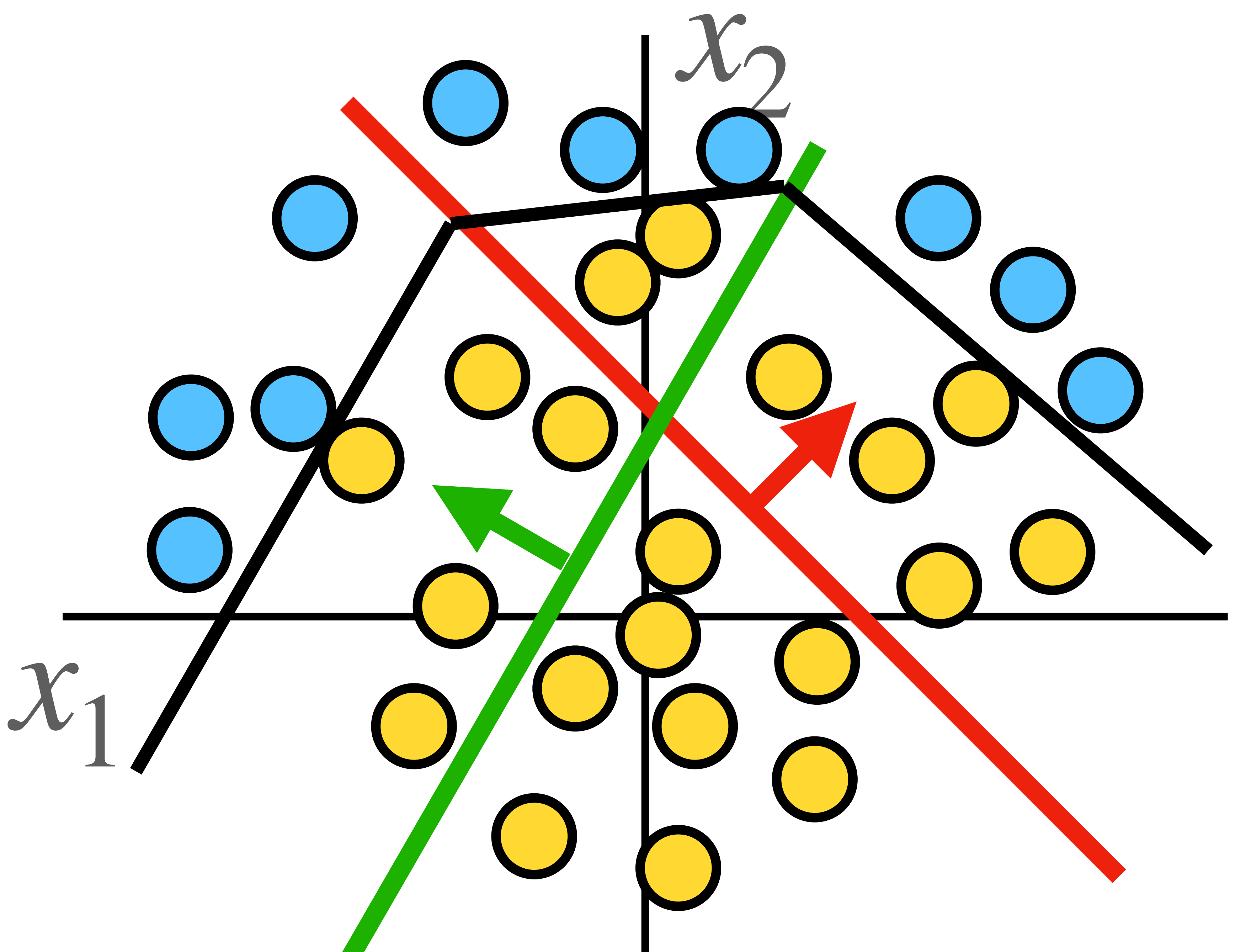


Points are linearly separable in feature space!

Space Warping

Points not linearly separable in original space

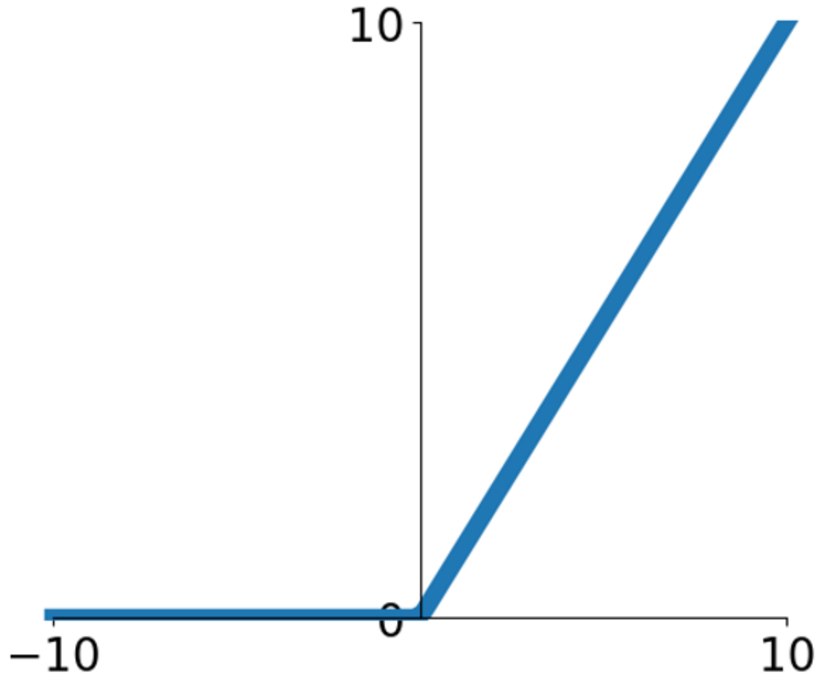
Consider a neural net hidden layer: $h = ReLU(Wx + b) = \max(0, Wx + b)$ where x, b, h are each 2-dimensional

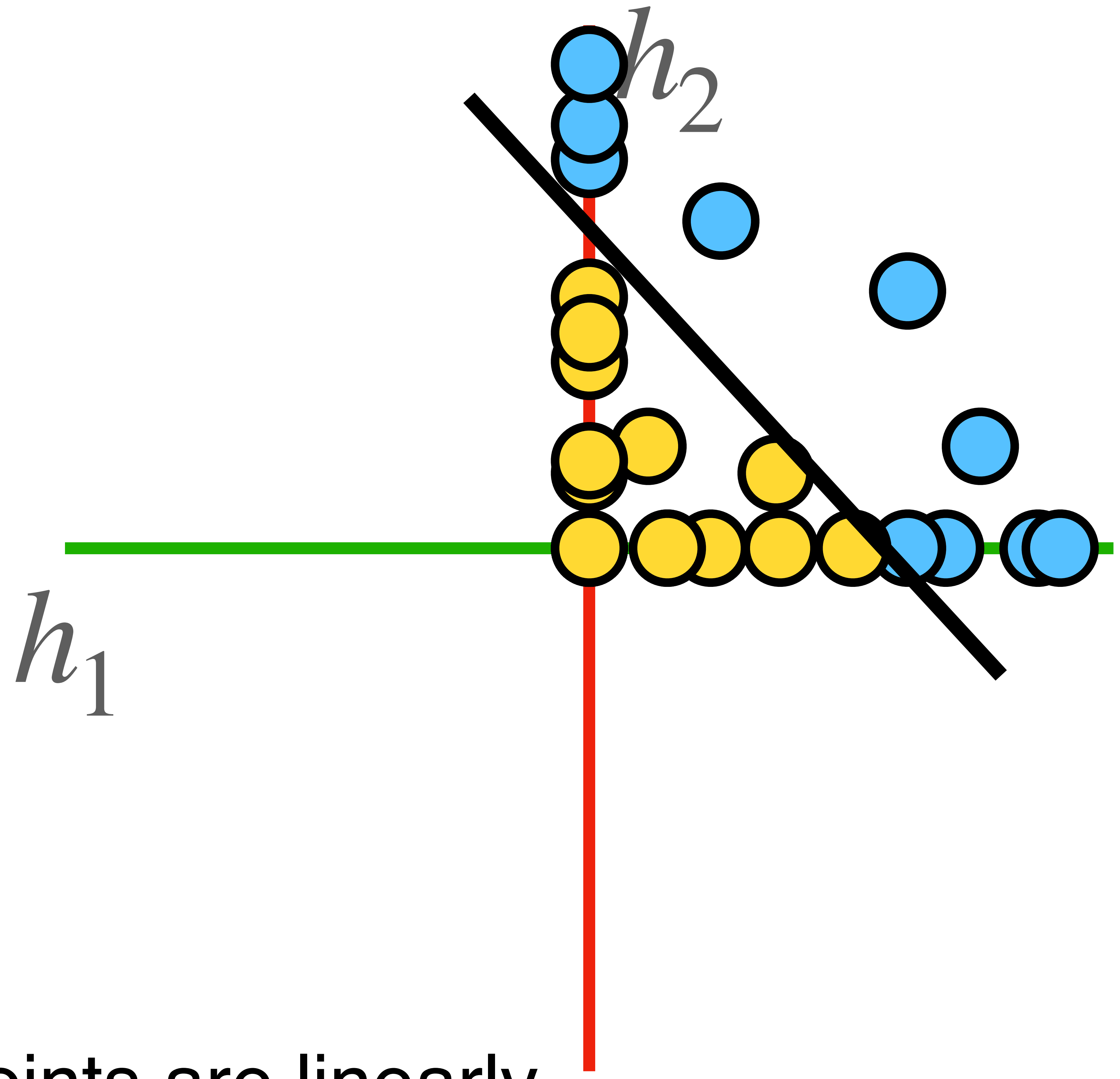


Linear classifier in feature space gives nonlinear classifier in original space

Feature transform:

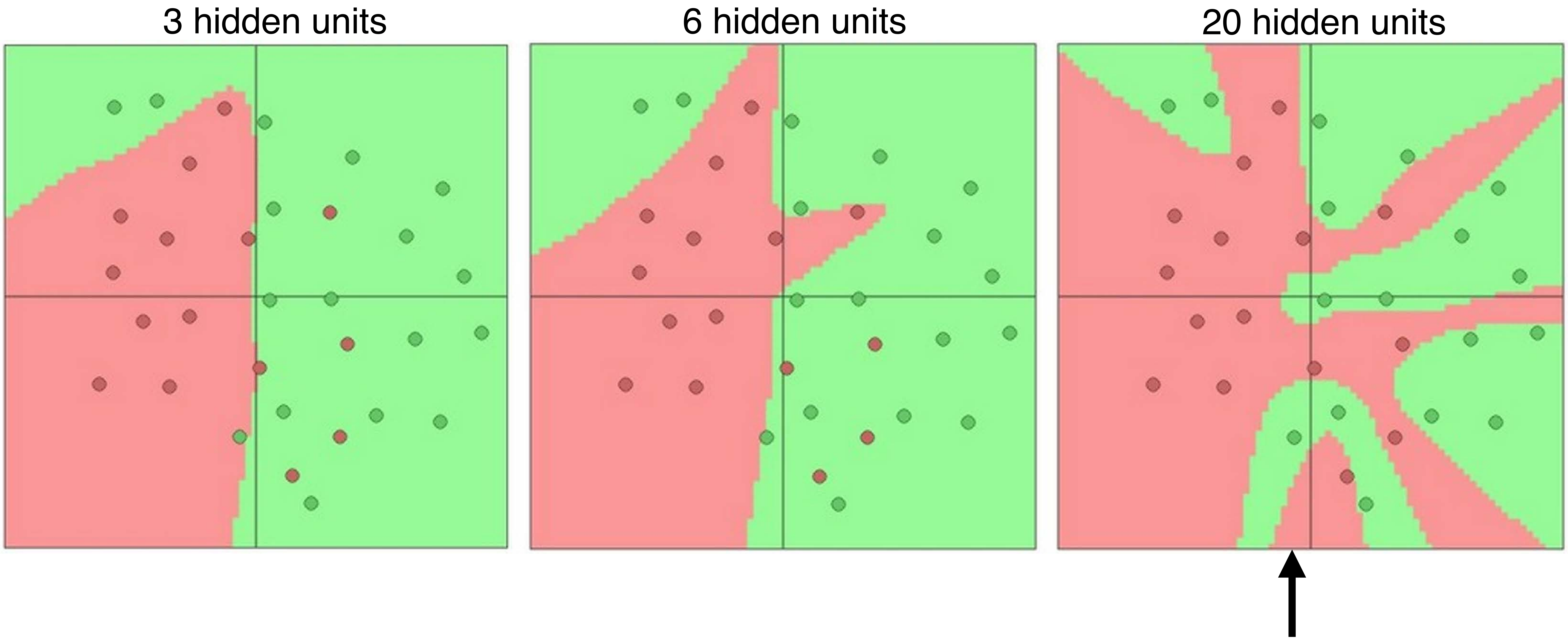
$h = ReLU(Wx + b)$





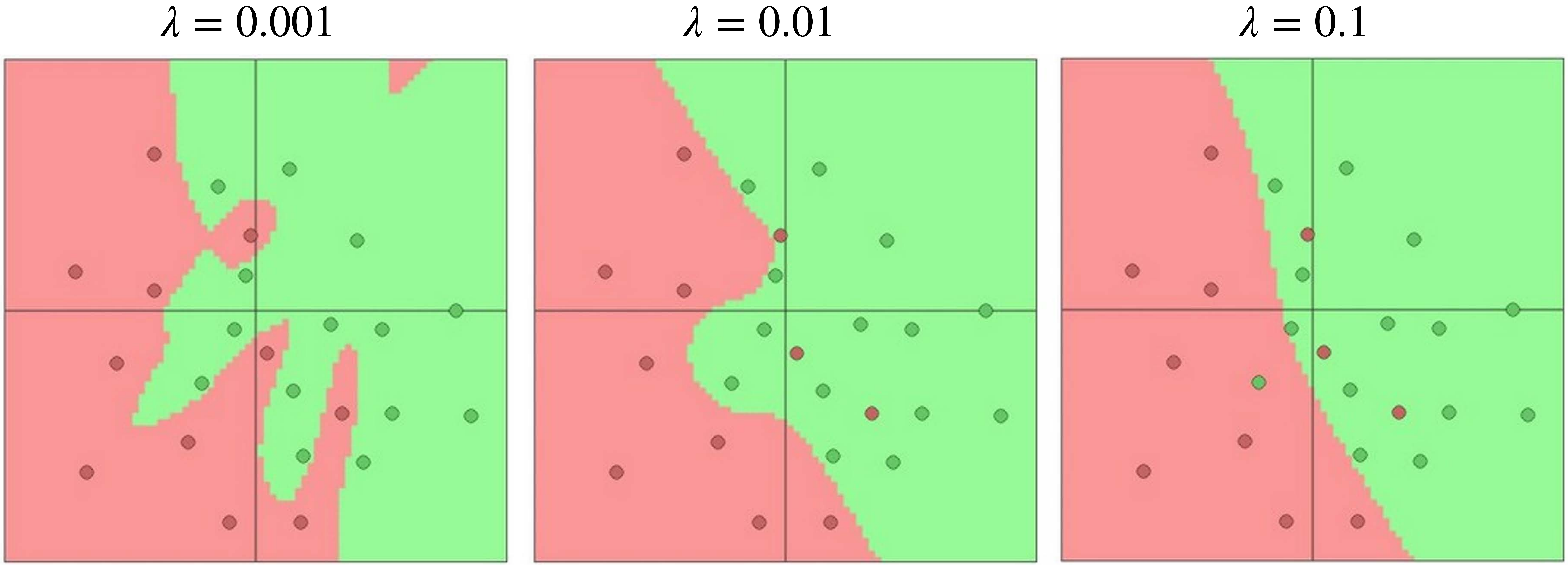
Points are linearly separable in feature space!

Setting the number of layers and their sizes



More hidden units = more capacity

Don't regularize with size; instead use stronger L2



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

Universal Approximation

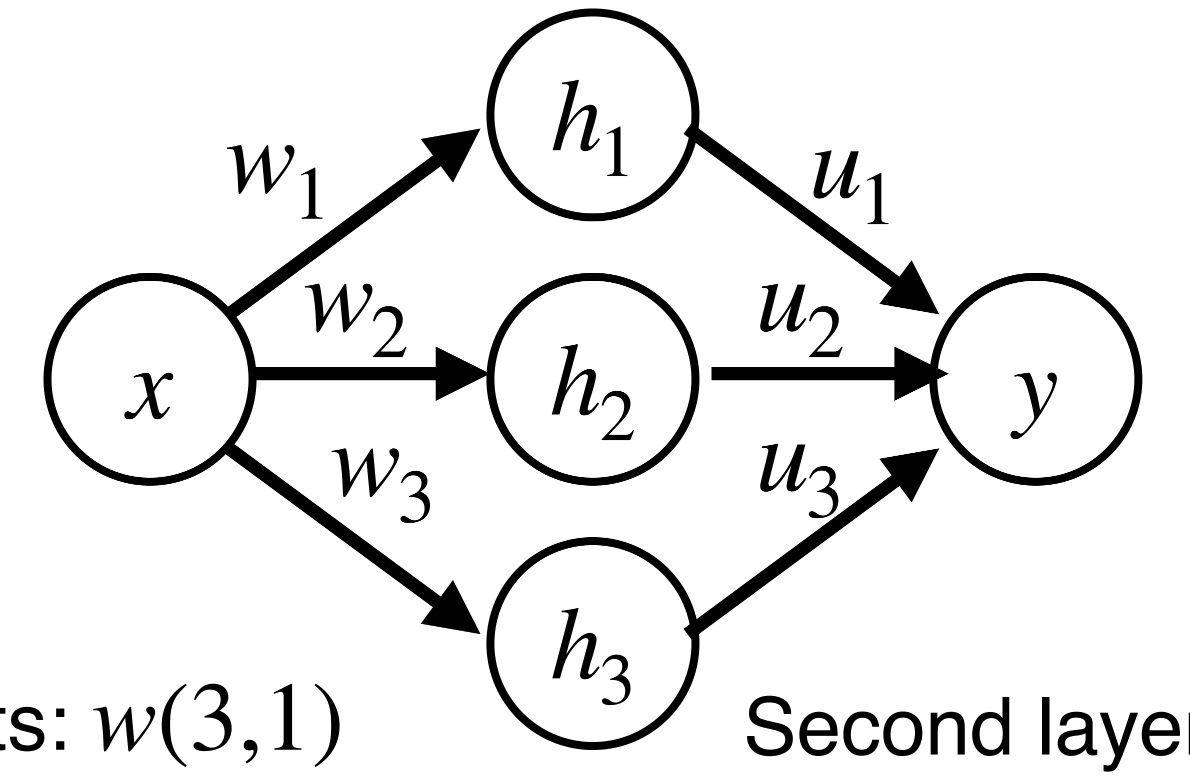
A neural network with one hidden layer can approximate any function $f : \mathbb{R}^N \to \mathbb{R}^M$ with arbitrary precision*

*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 $f : \mathbb{R} \to \mathbb{R}$ with a two-layer ReLU network

Input: $x(1,)$



Output: $y(1,)$

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

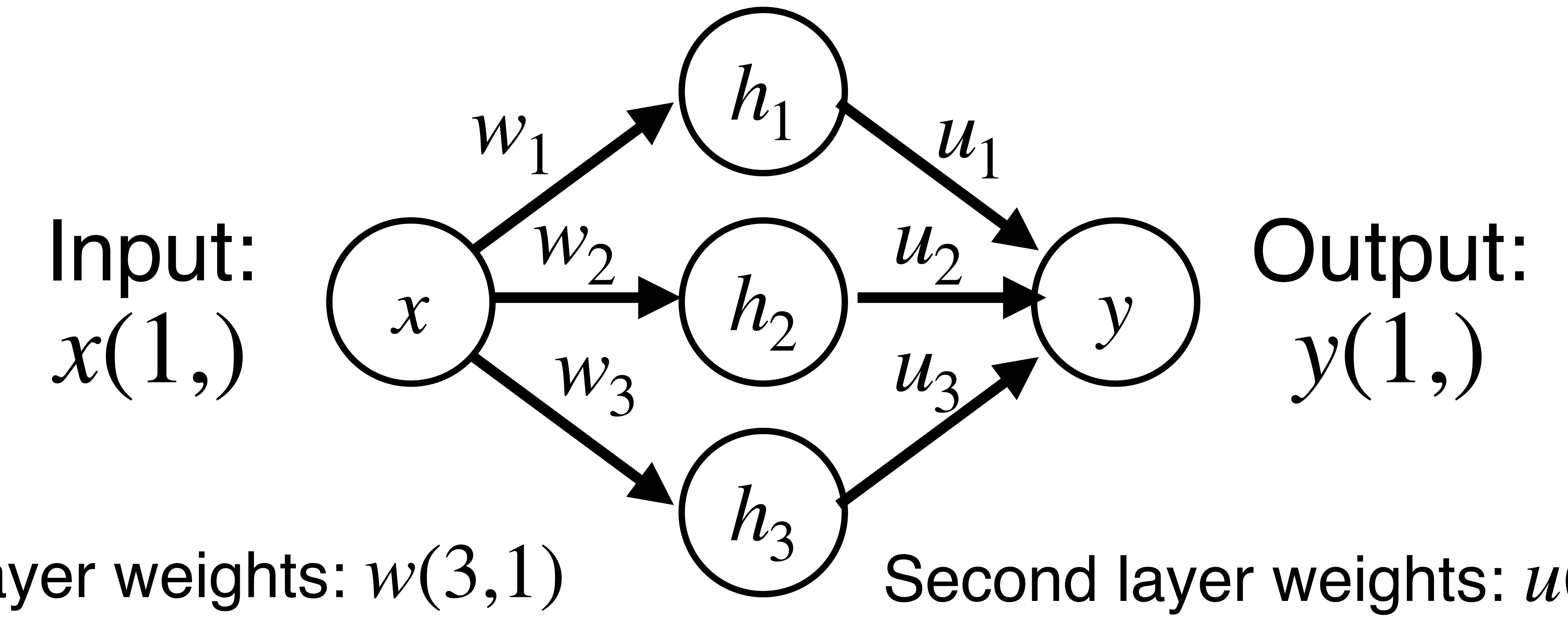
First layer bias: $b(3,)$

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

First layer bias: $p(1,)$

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: $b(3,)$

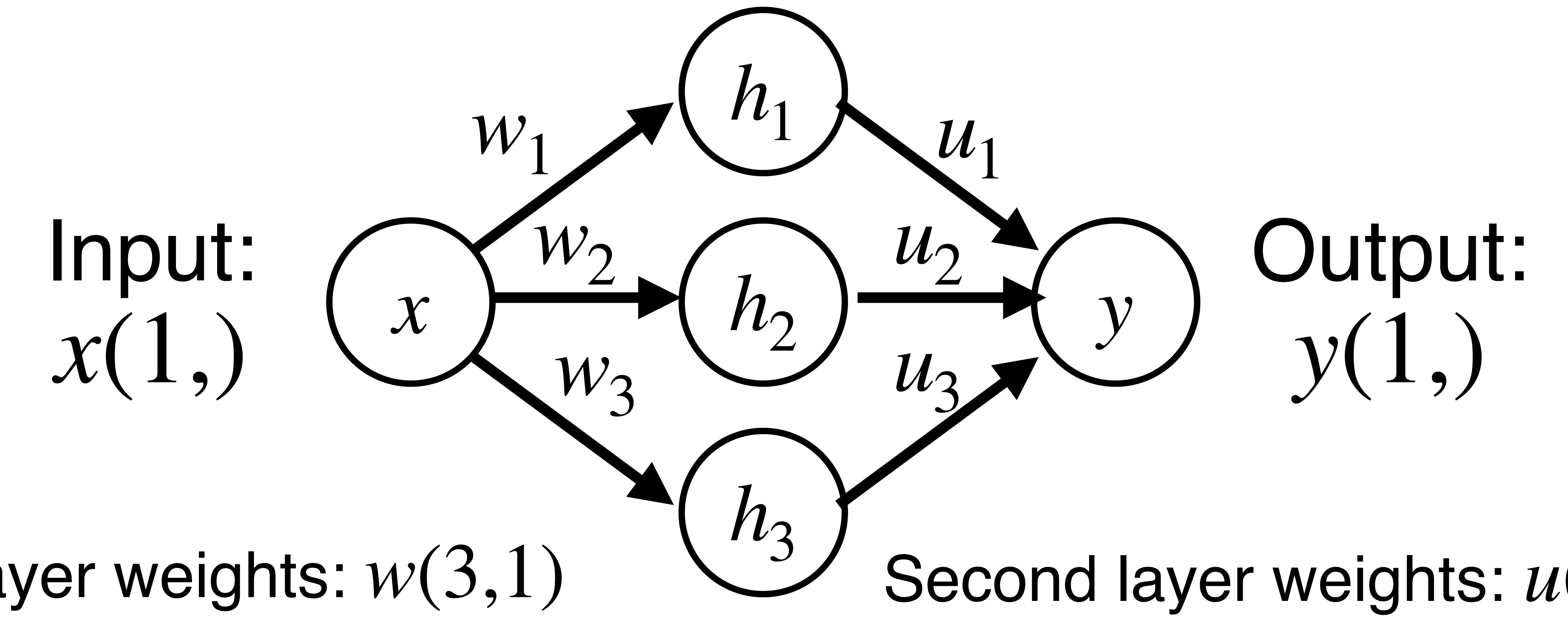
Second layer weights: $u(1,3)$

First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

Universal Approximation

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



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

First layer bias: $b(3,)$

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

First layer bias: $p(1,)$

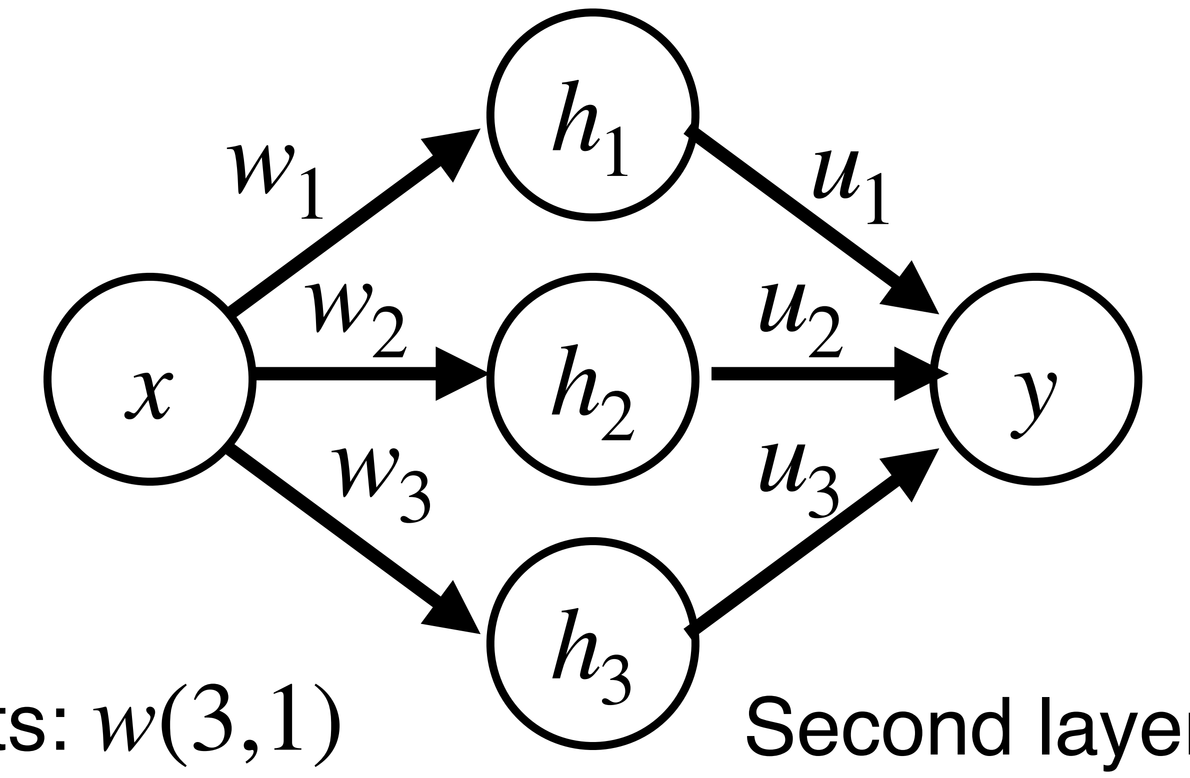
$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1) + u_2 \max(0, w_2 x + b_2) + u_3 \max(0, w_3 x + b_3) + p$$

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

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

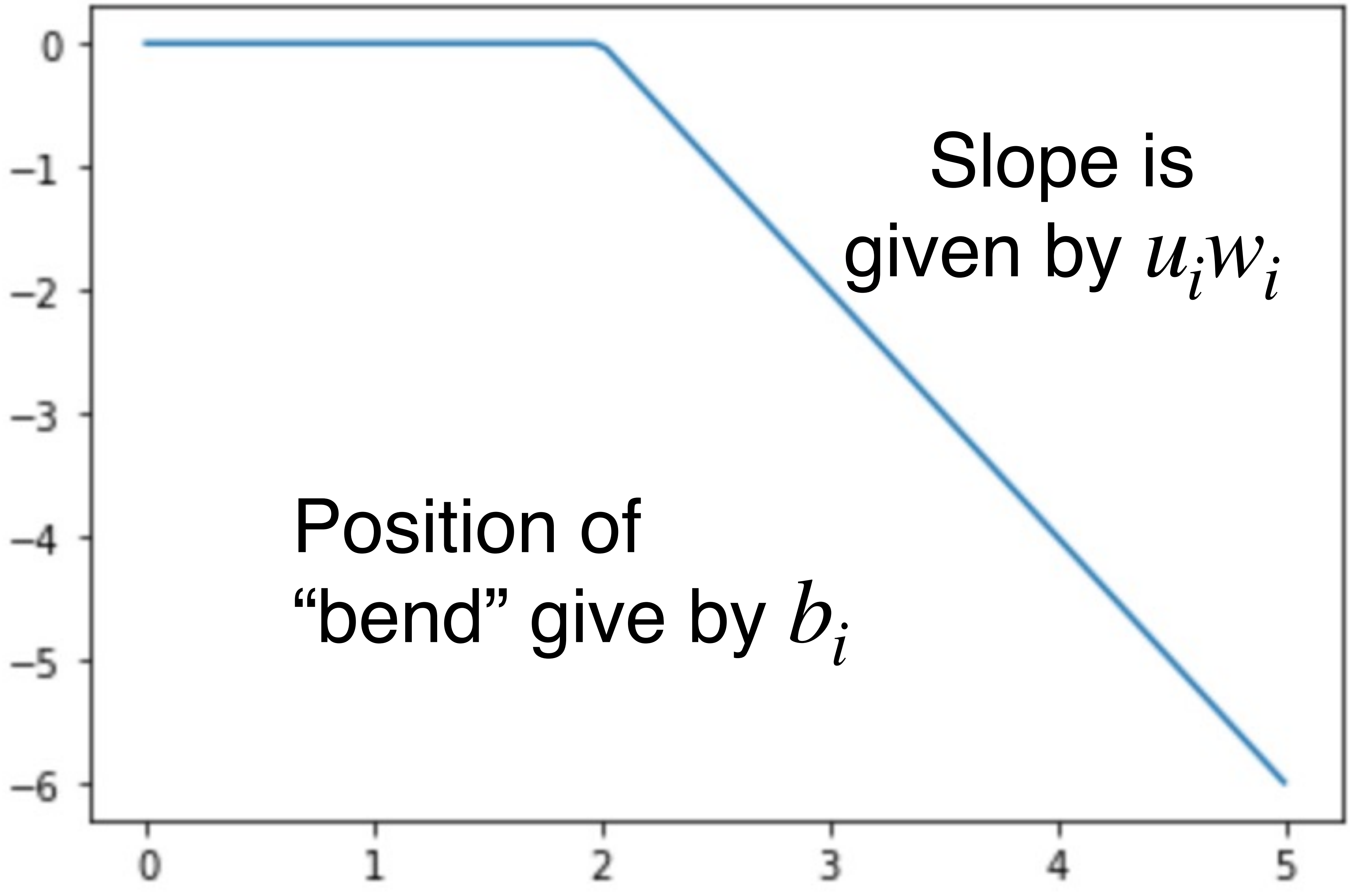
First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

Output is a sum of shifted, scaled ReLUs:

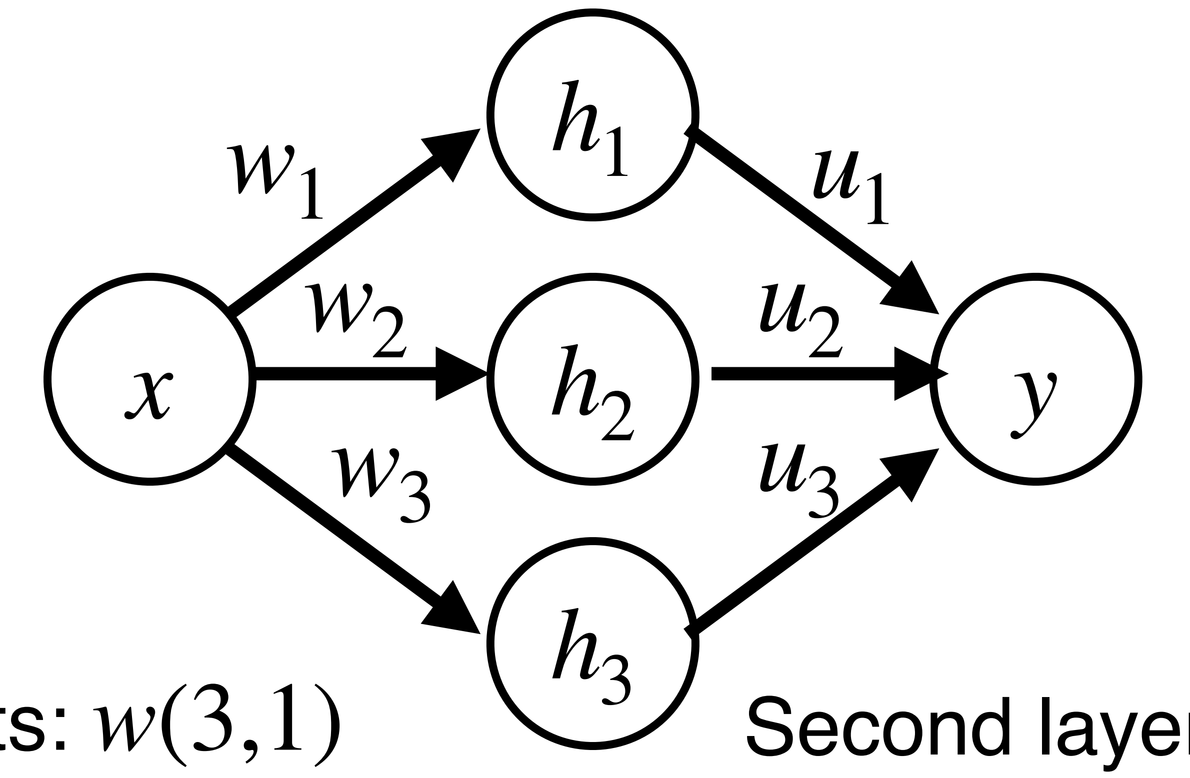
Flip left / right based on sign of w_i



Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

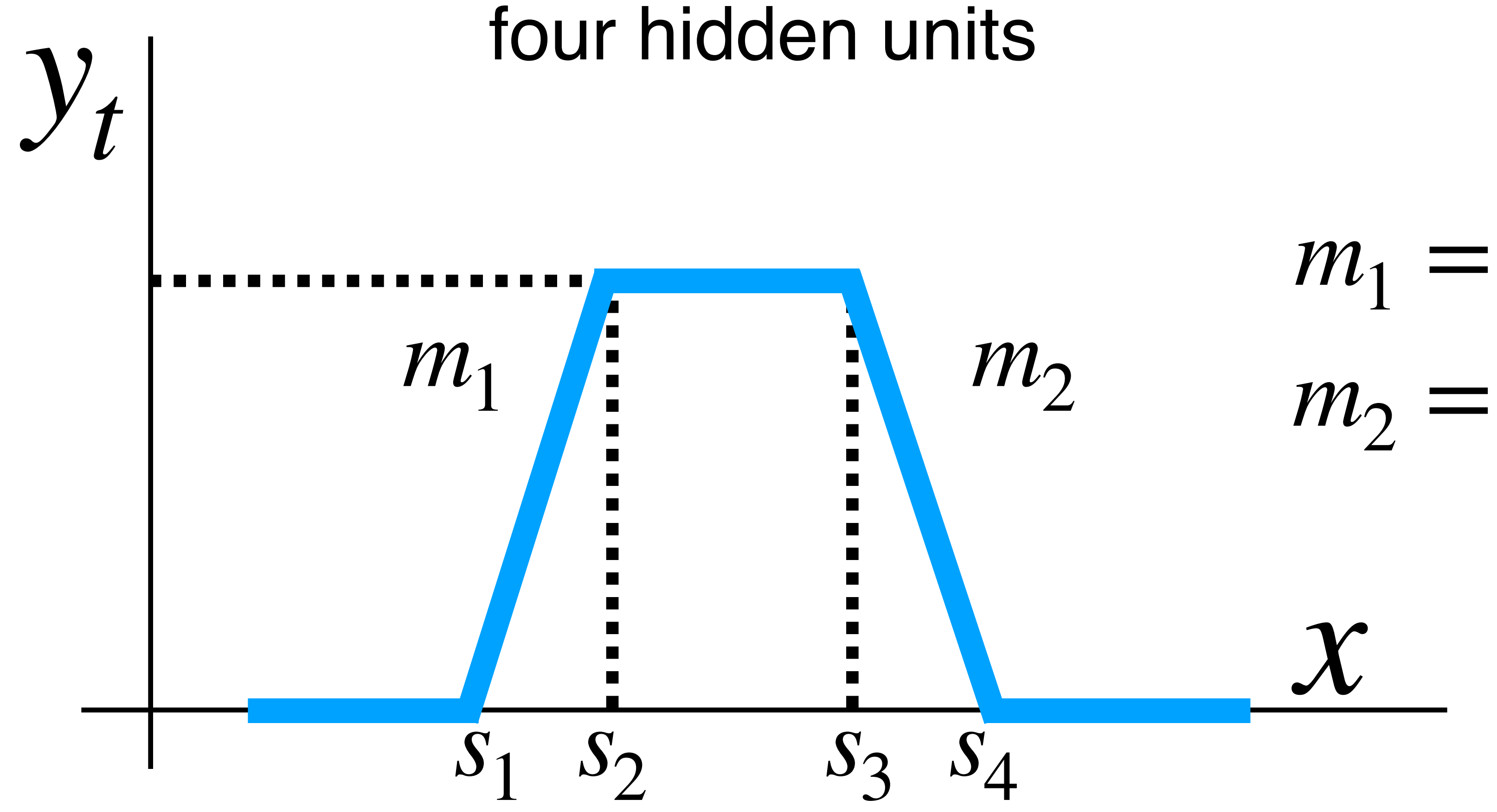
Second layer weights: $u(1,3)$

First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

We can build a "bump function" using four hidden units

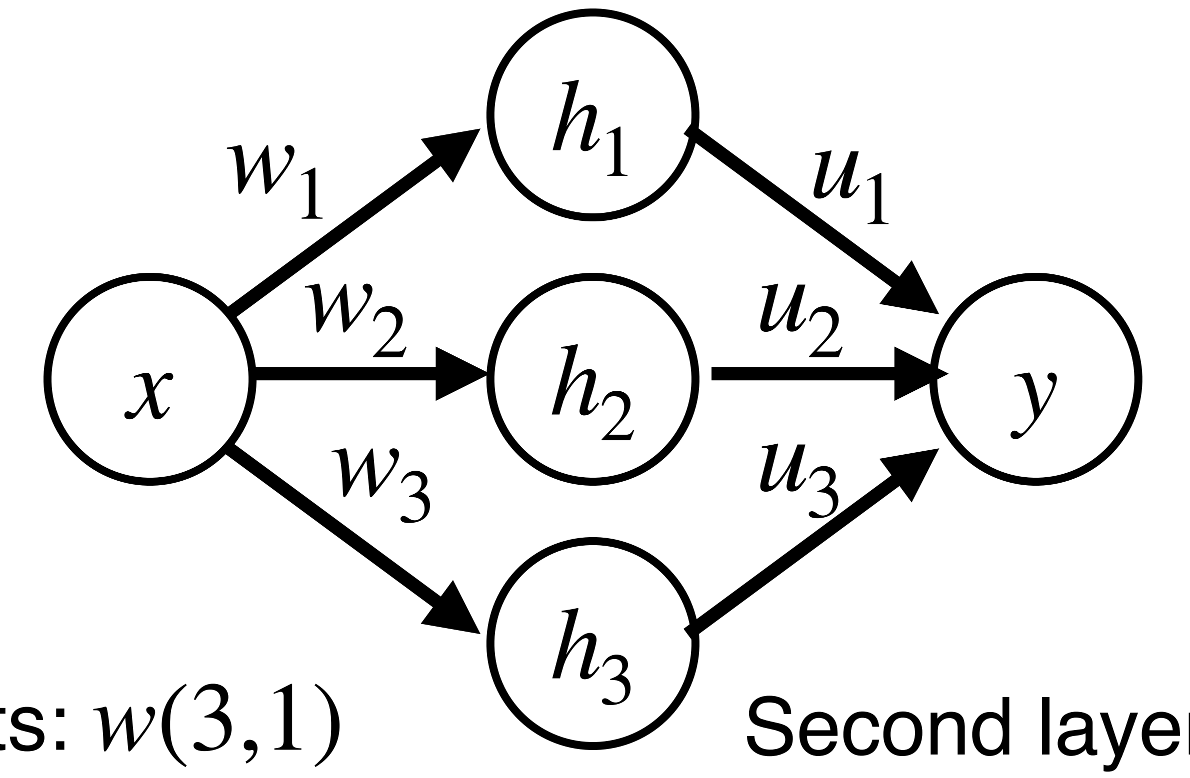


$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

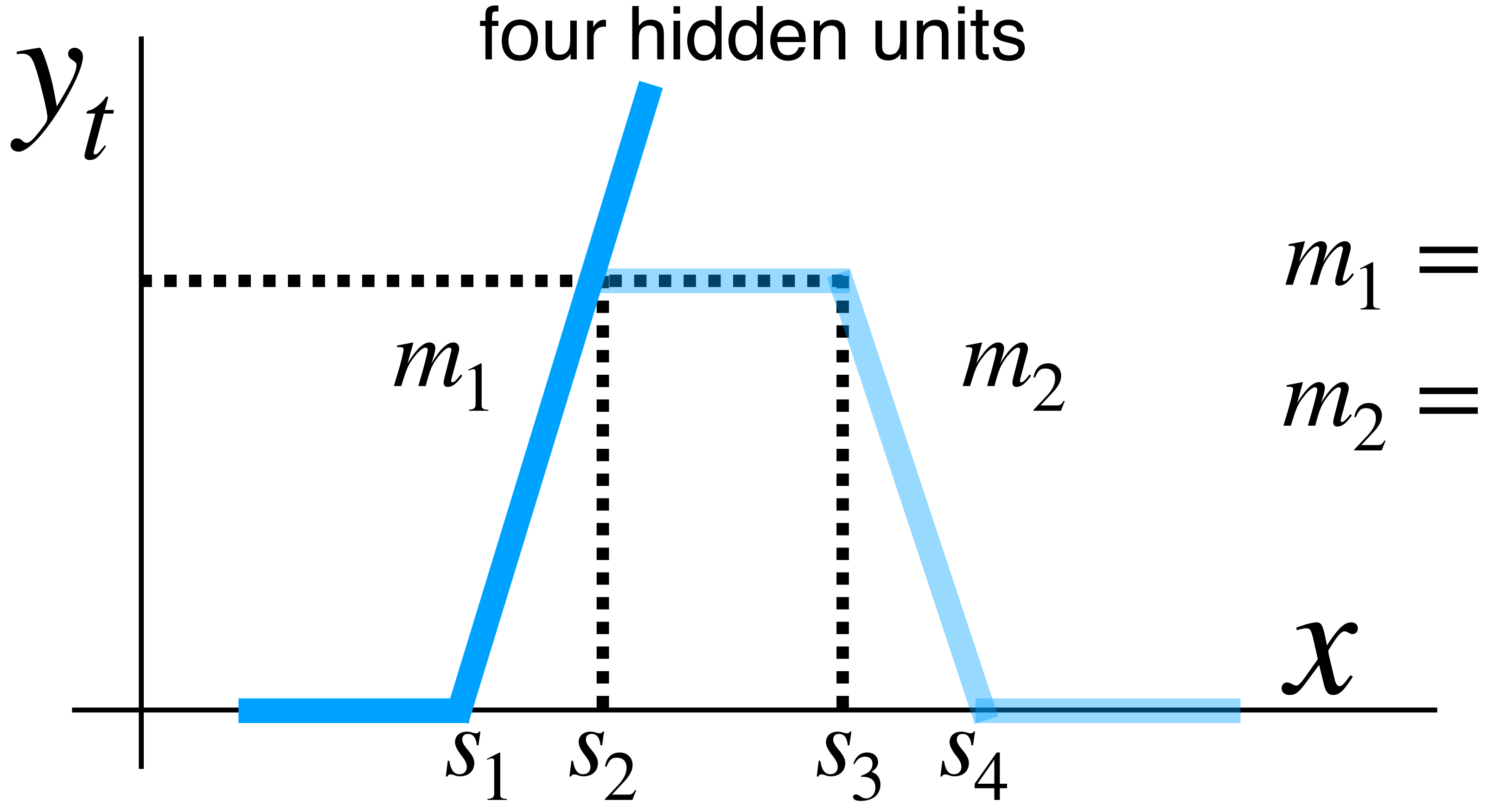
Second layer weights: $u(1,3)$

First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

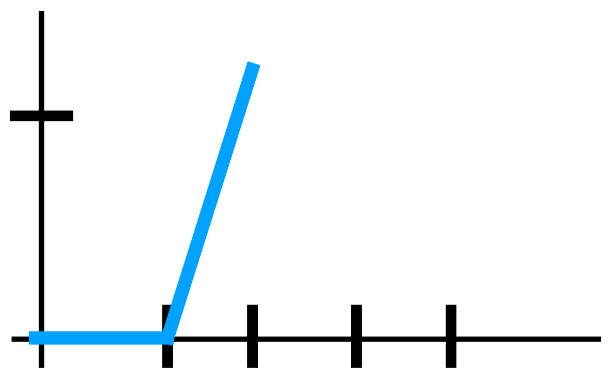
$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

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



$m_1 = t/(s_2 - s_1)$

$m_2 = t/(s_4 - s_3)$

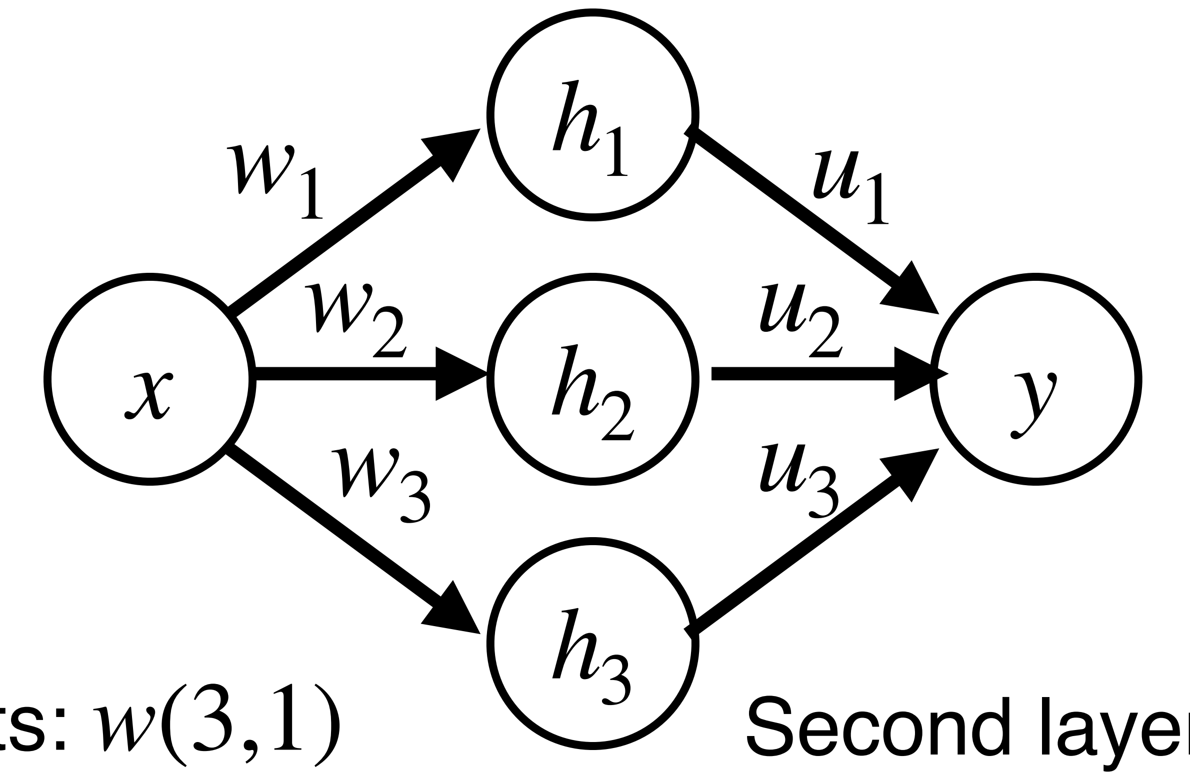


$m_1 \max(0, x - s_1)$

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

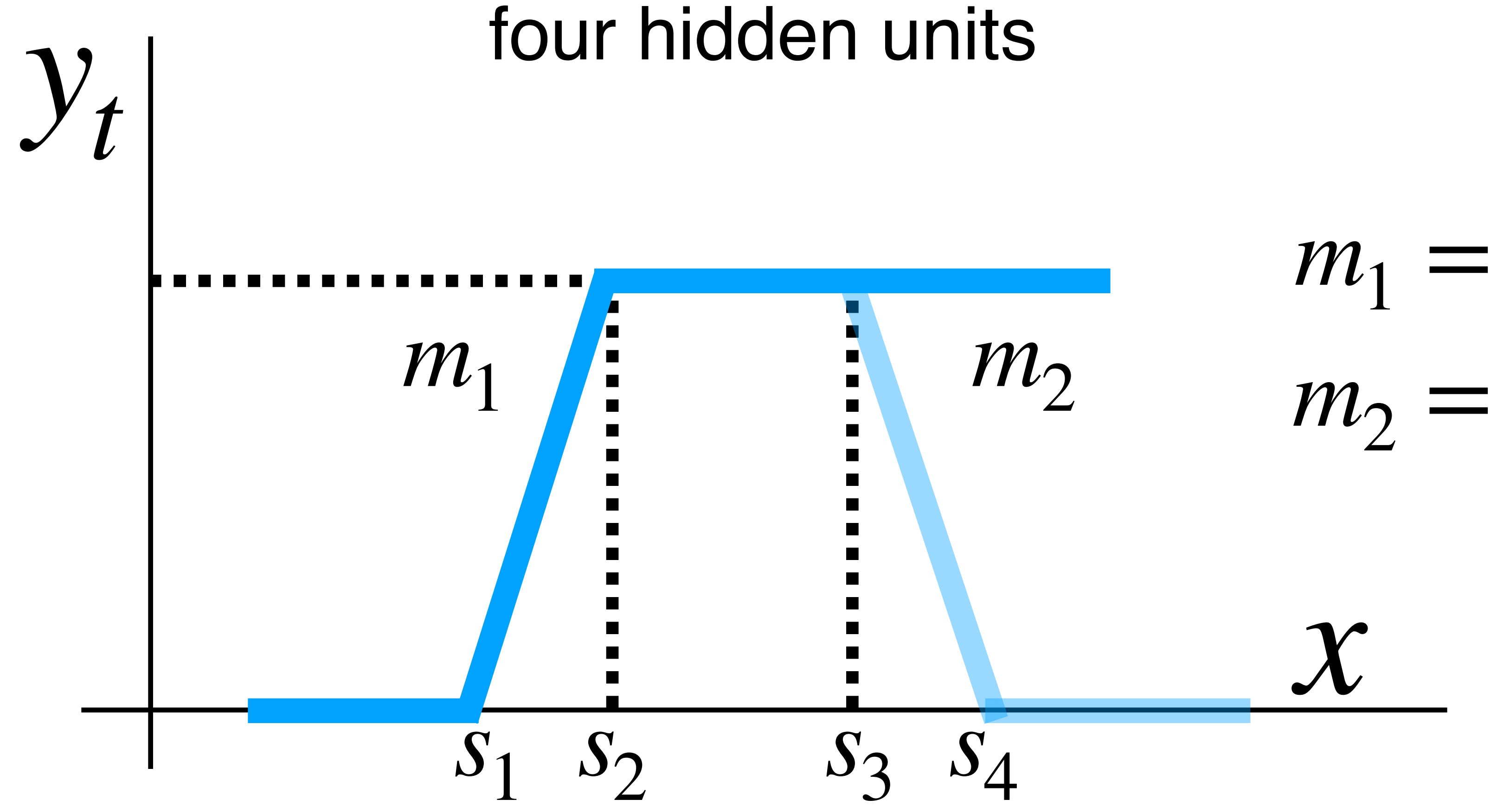
Second layer weights: $u(1,3)$

First layer bias: $p(1,)$

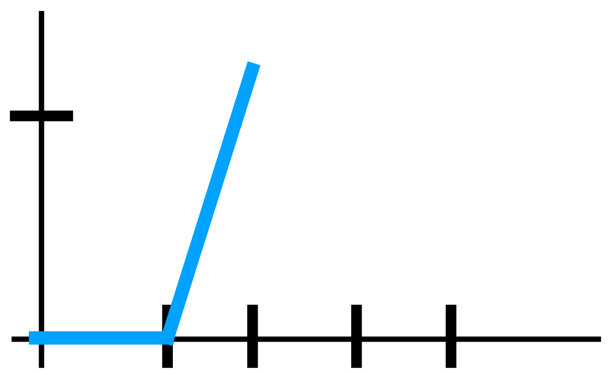
$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

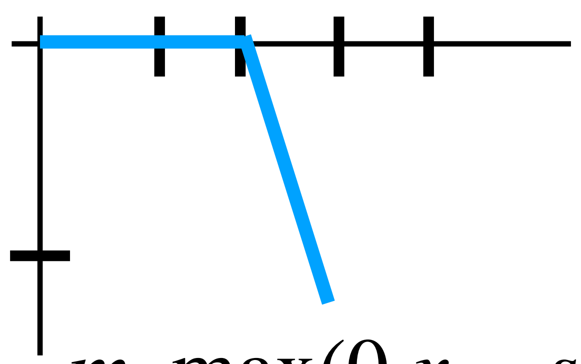
We can build a "bump function" using four hidden units



$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$



$m_1 \max(0, x - s_1)$

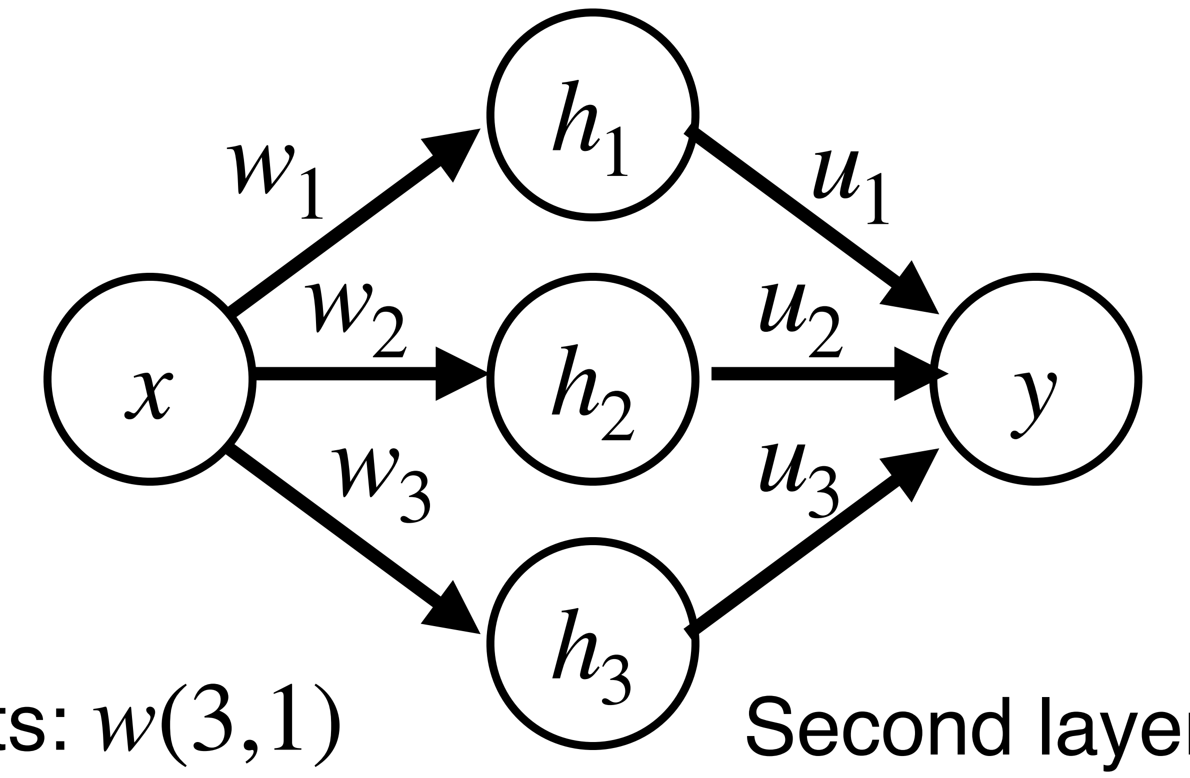


$-m_1 \max(0, x - s_2)$

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

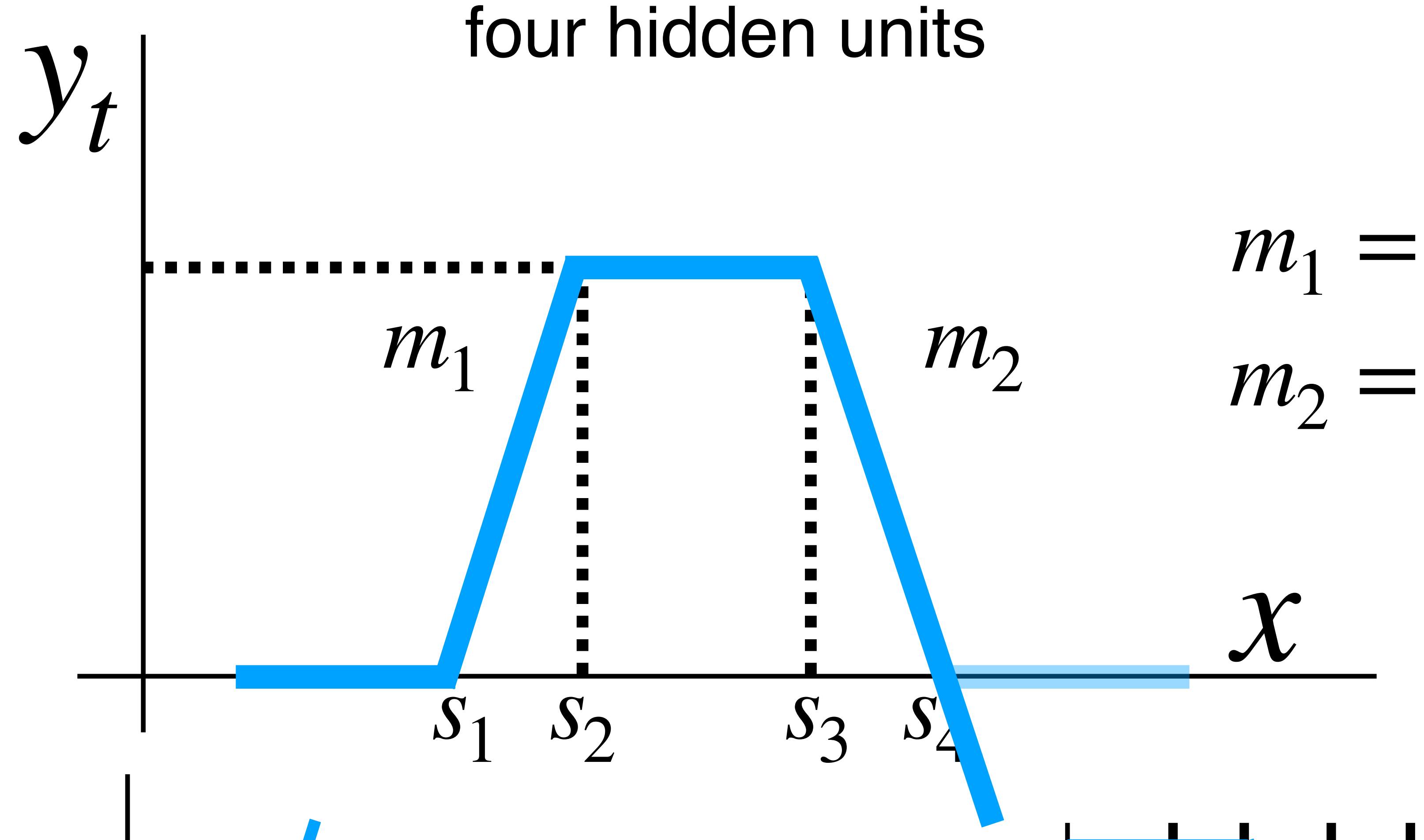
Second layer weights: $u(1,3)$

First layer bias: $p(1,)$

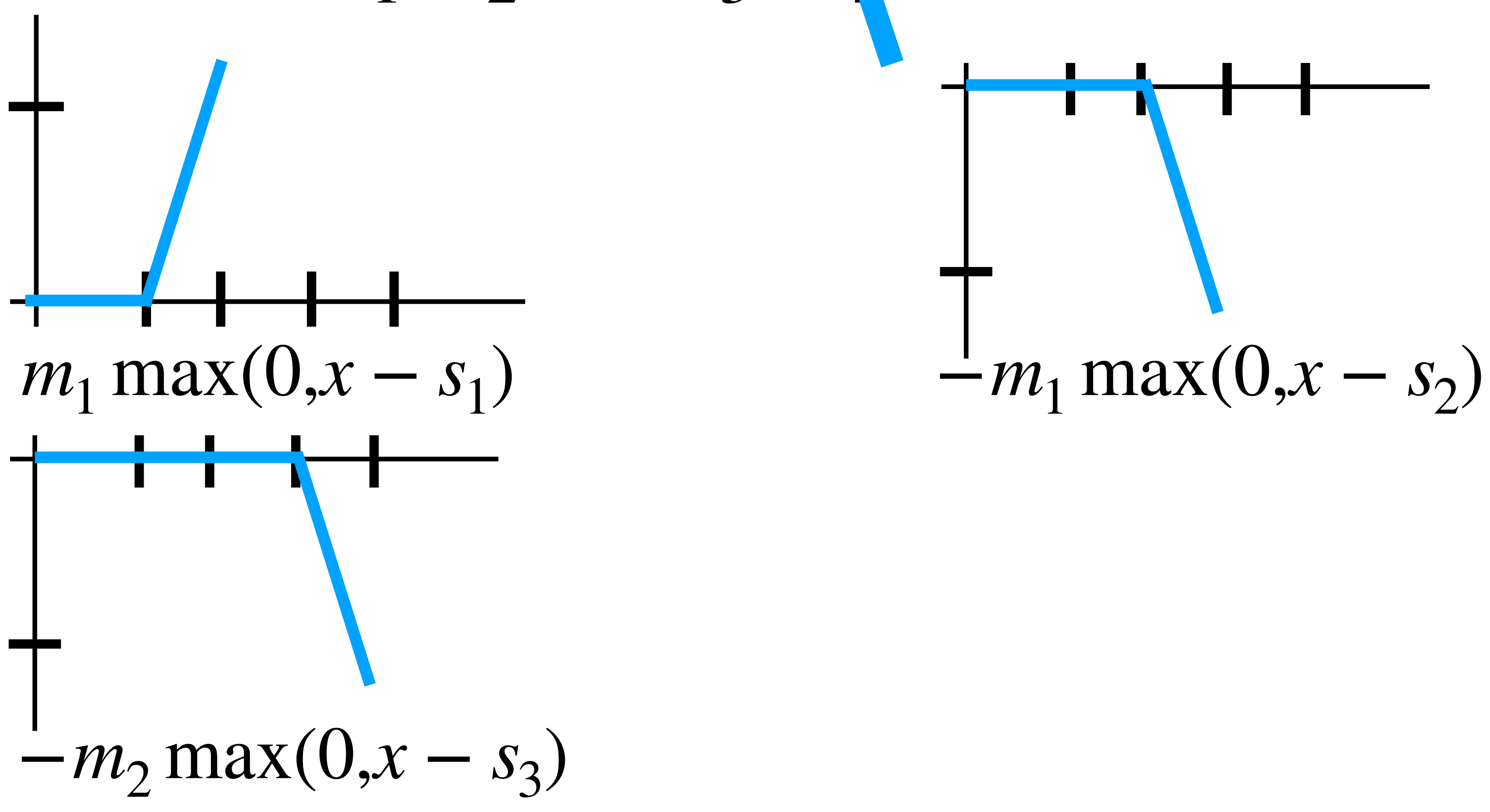
$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

We can build a "bump function" using four hidden units

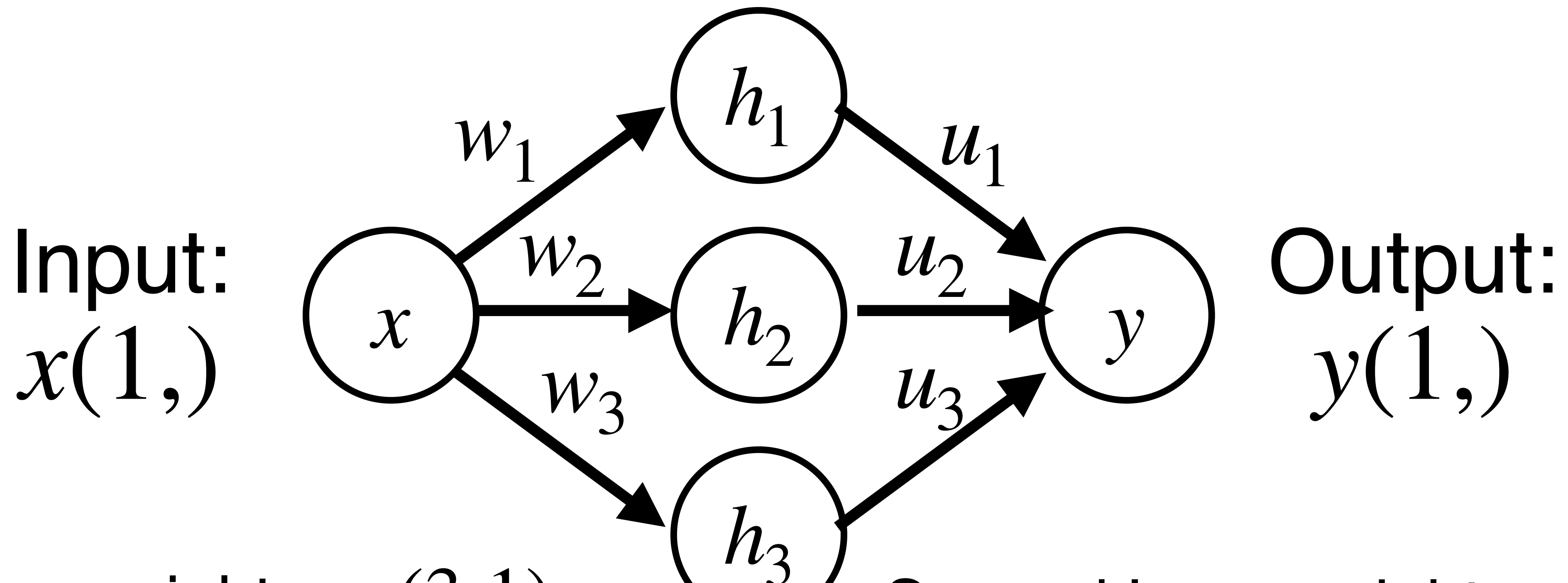


$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$



Universal Approximation

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



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

First layer bias: $b(3,)$

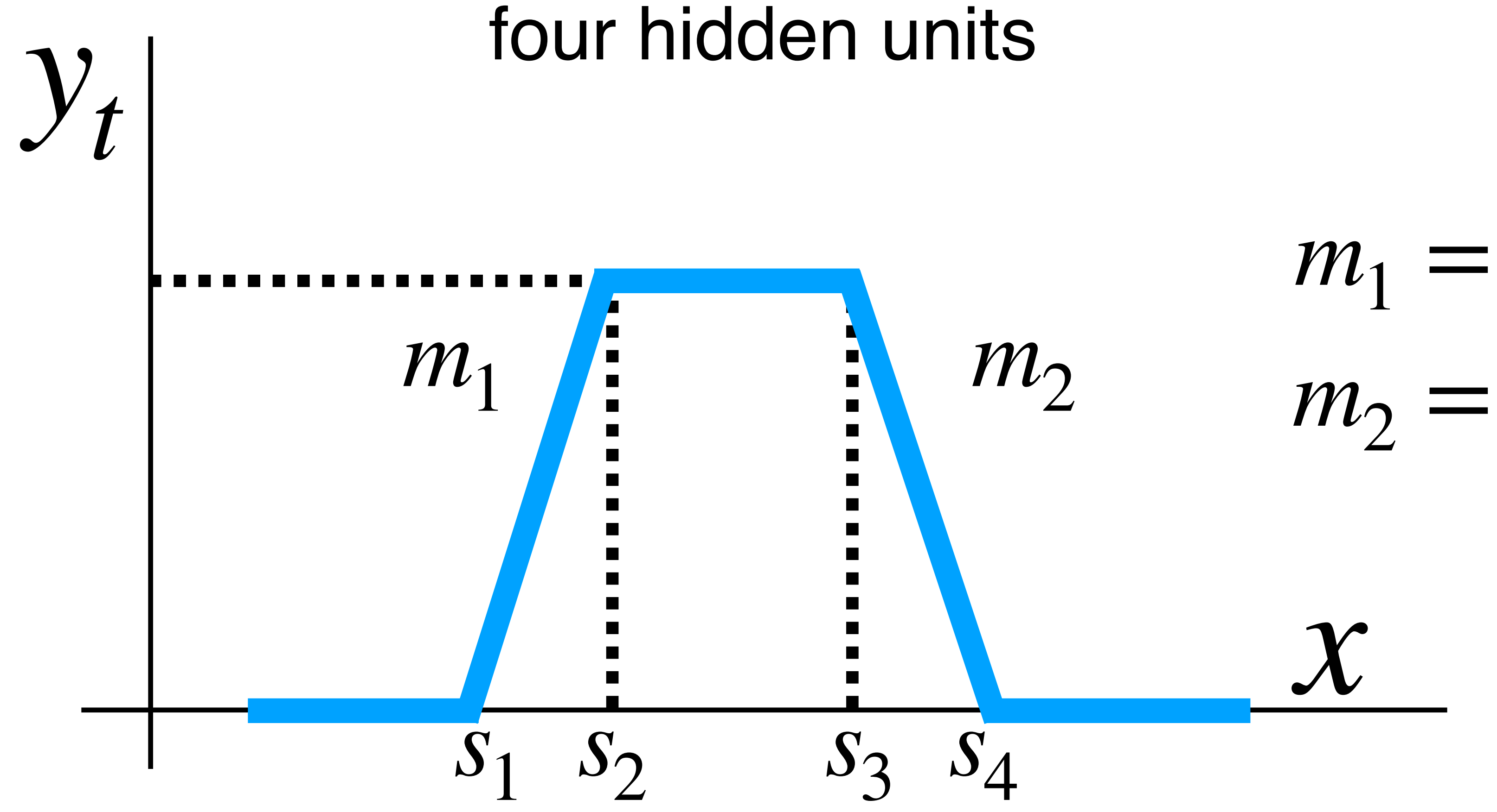
Second layer weights: $u(1,3)$

First layer bias: $p(1,)$

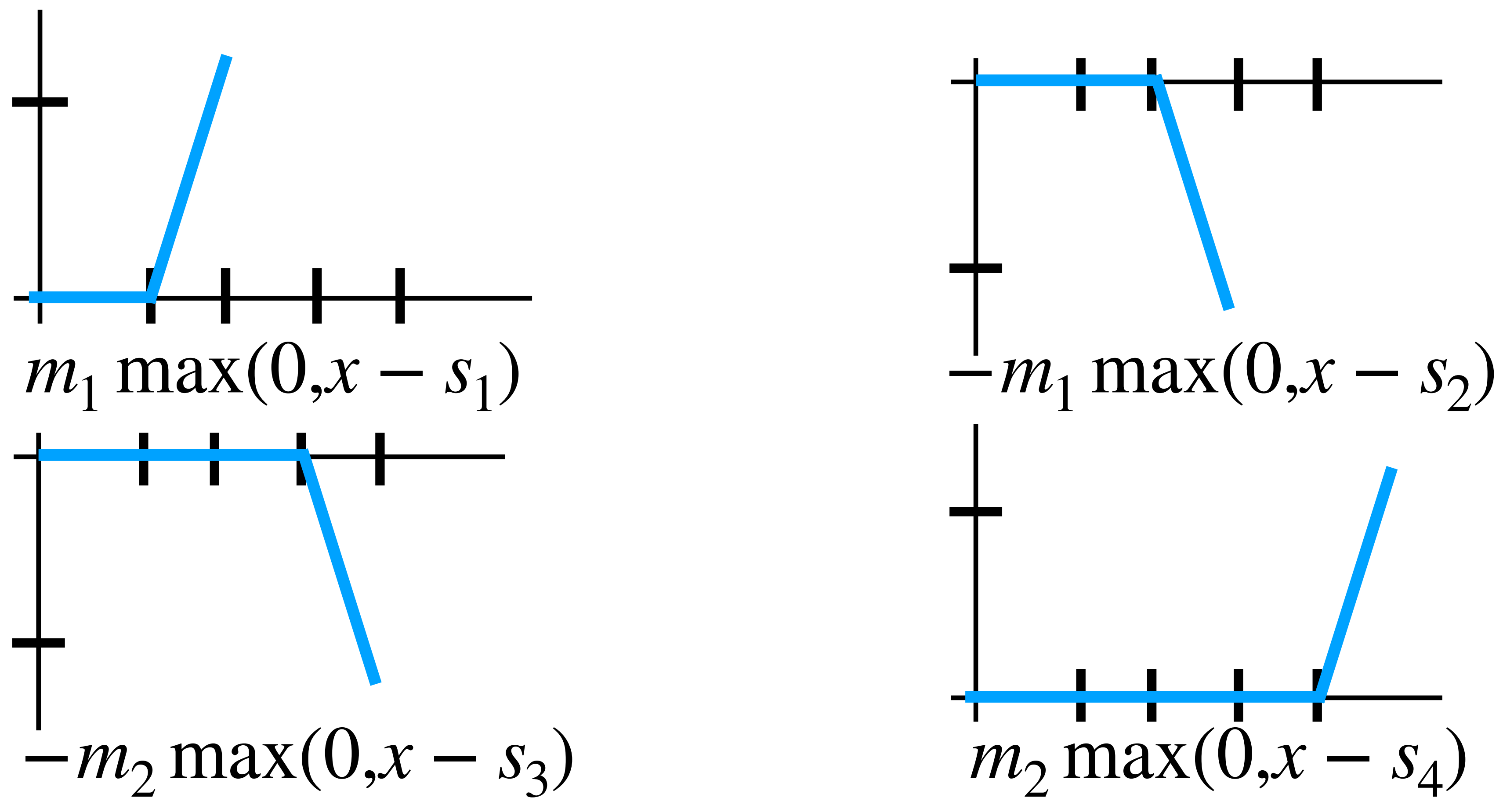
$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

We can build a "bump function" using four hidden units



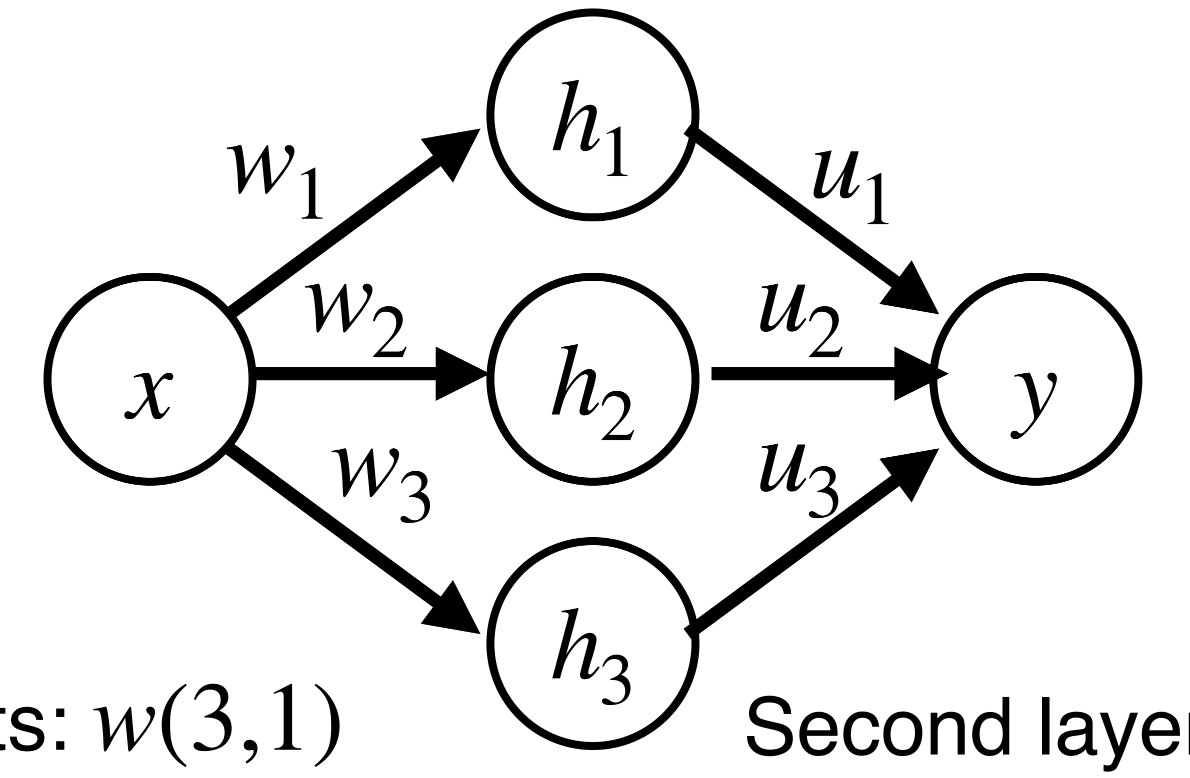
$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$



Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

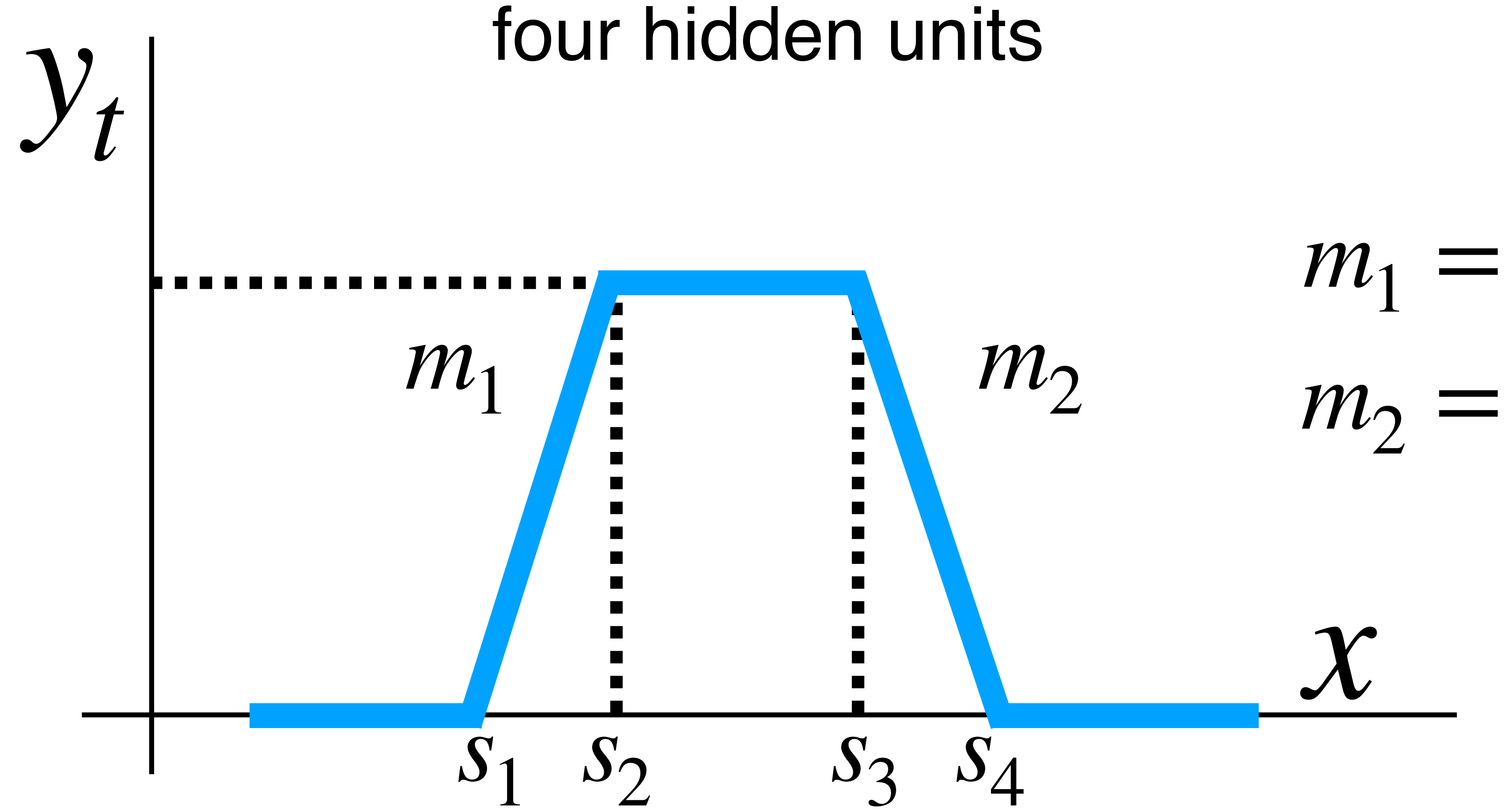
Second layer weights: $u(1,3)$

First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

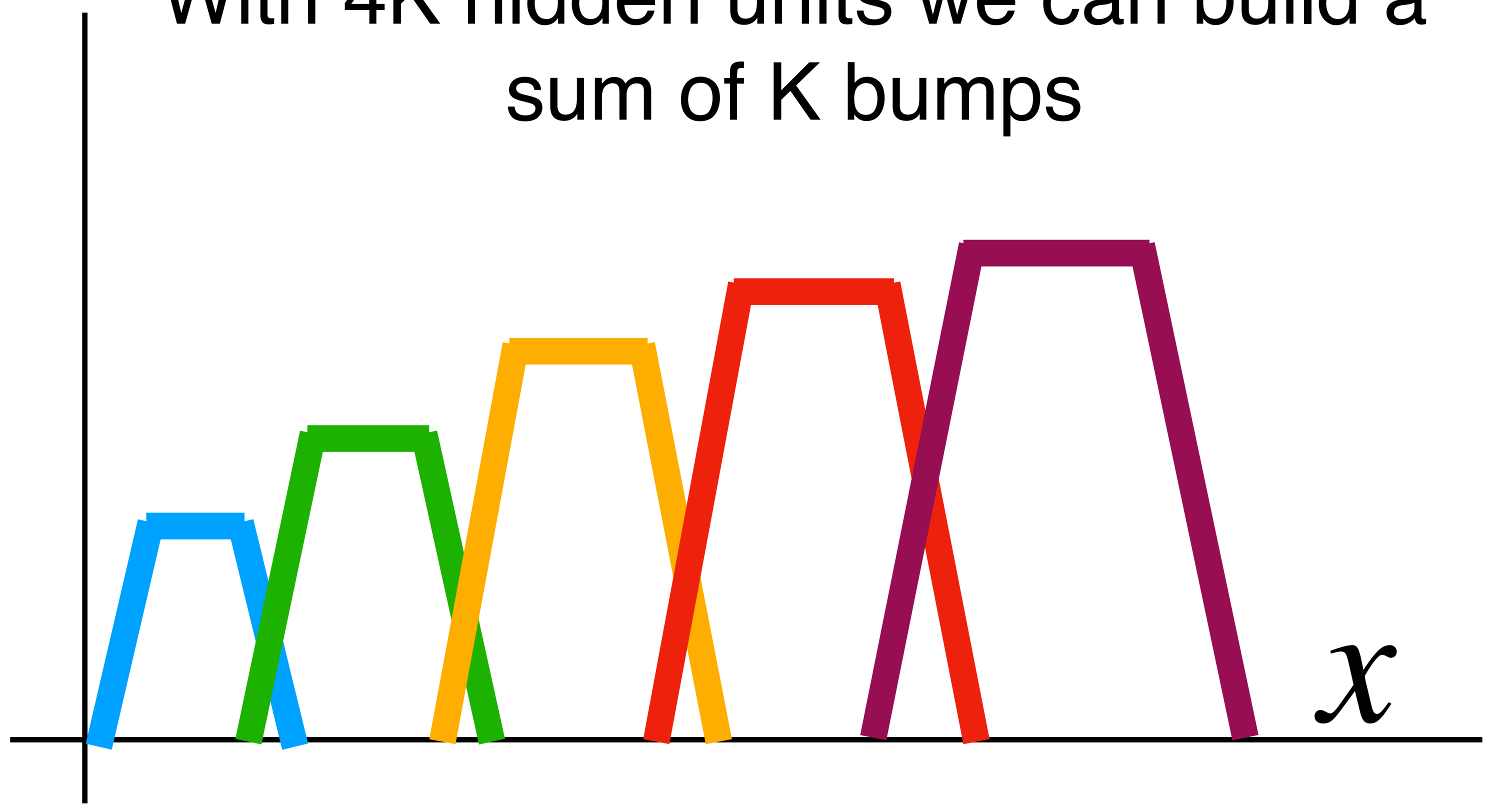
$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

We can build a "bump function" using four hidden units



$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$

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

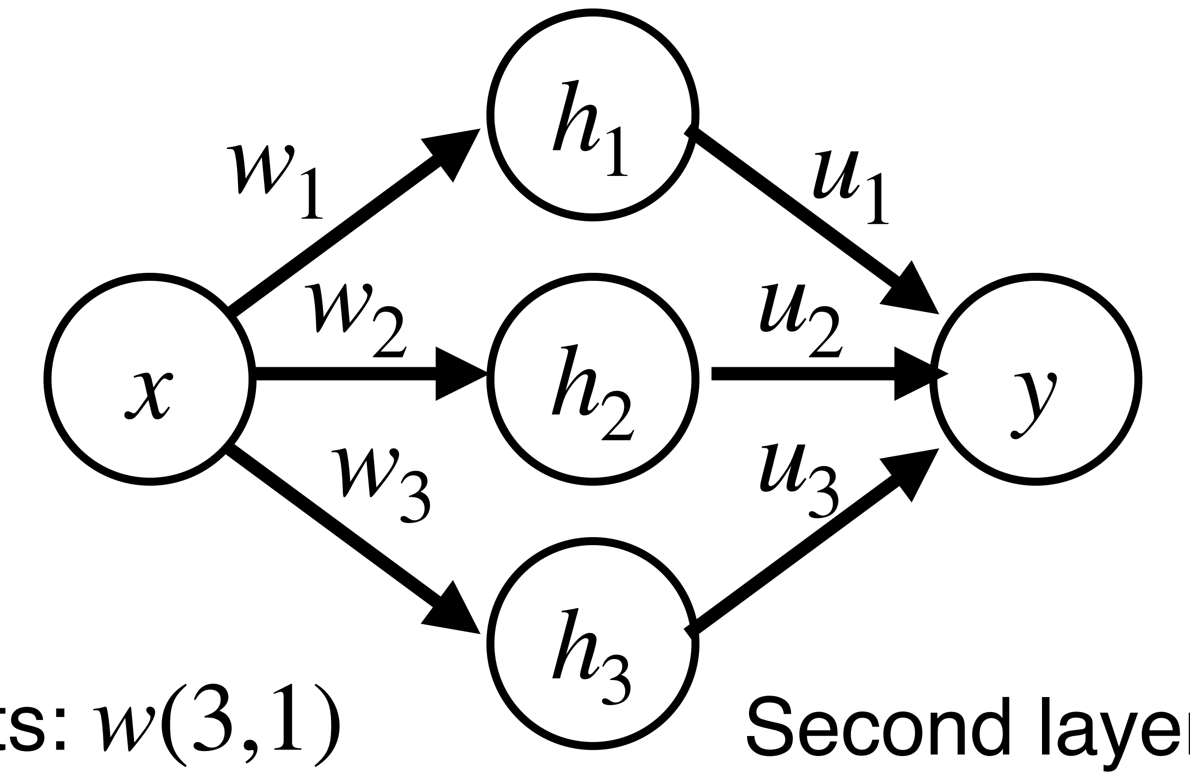


Approximate functions with bumps!

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

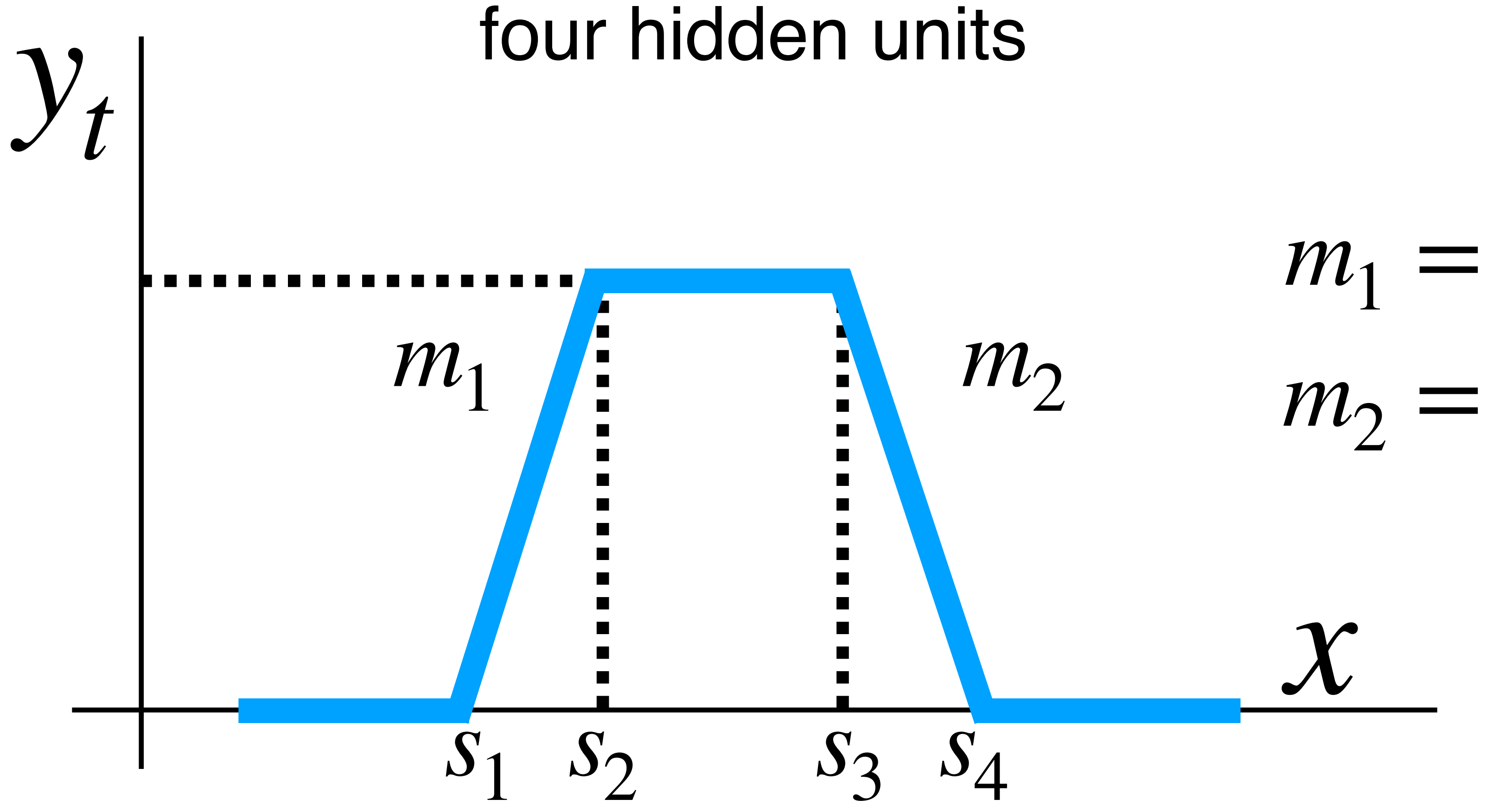
Second layer weights: $u(1,3)$

First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

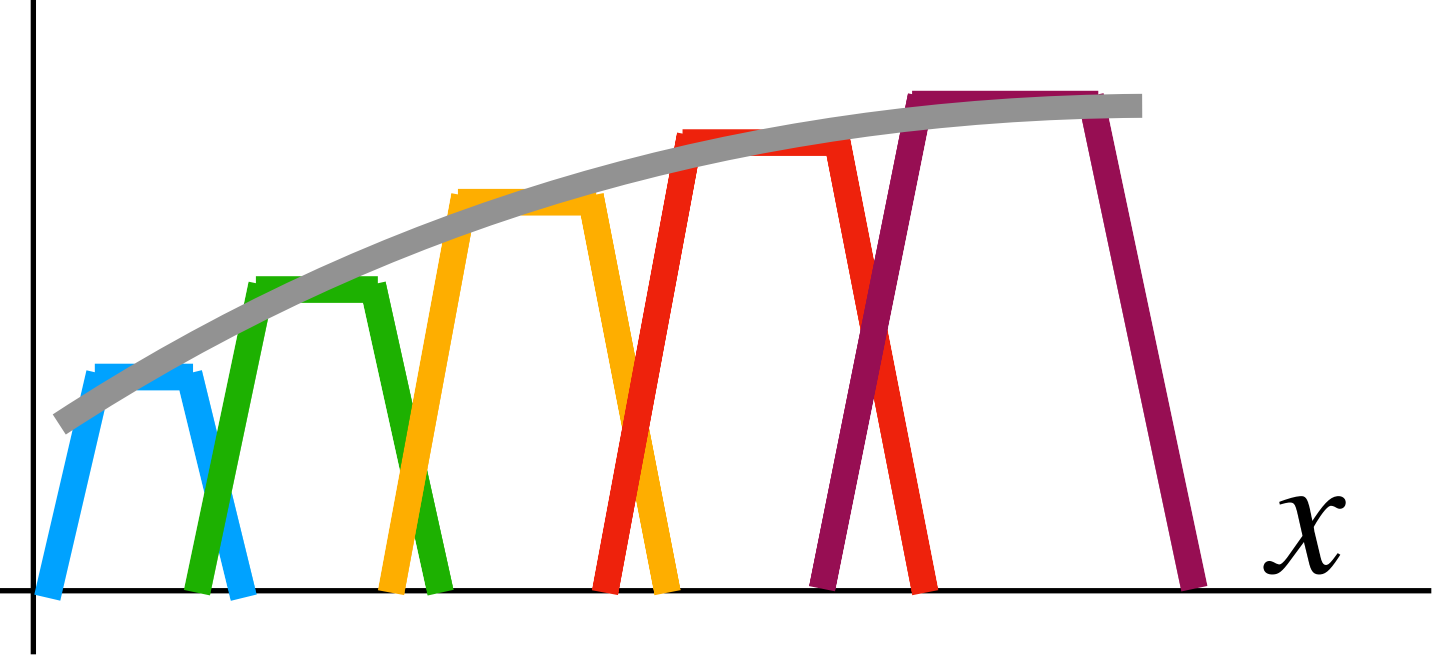
$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

We can build a "bump function" using four hidden units



$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$

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

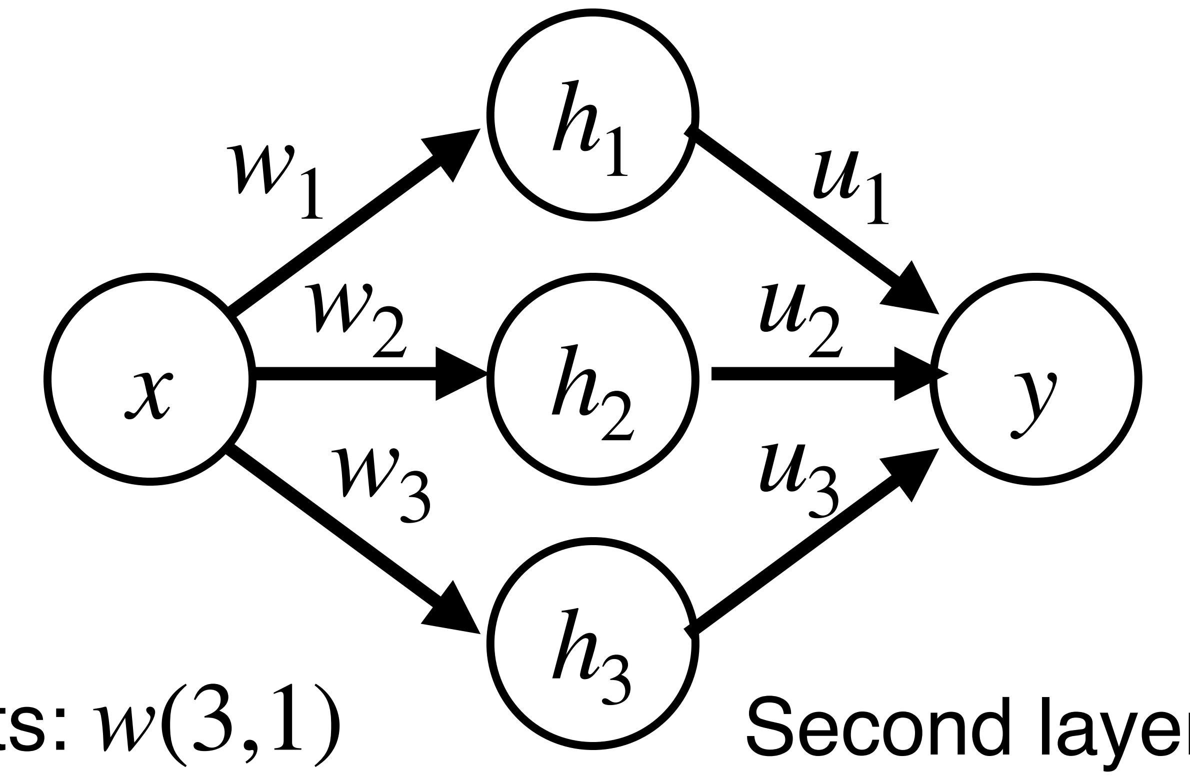


Approximate functions with bumps!

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

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

First layer bias: $p(1,)$

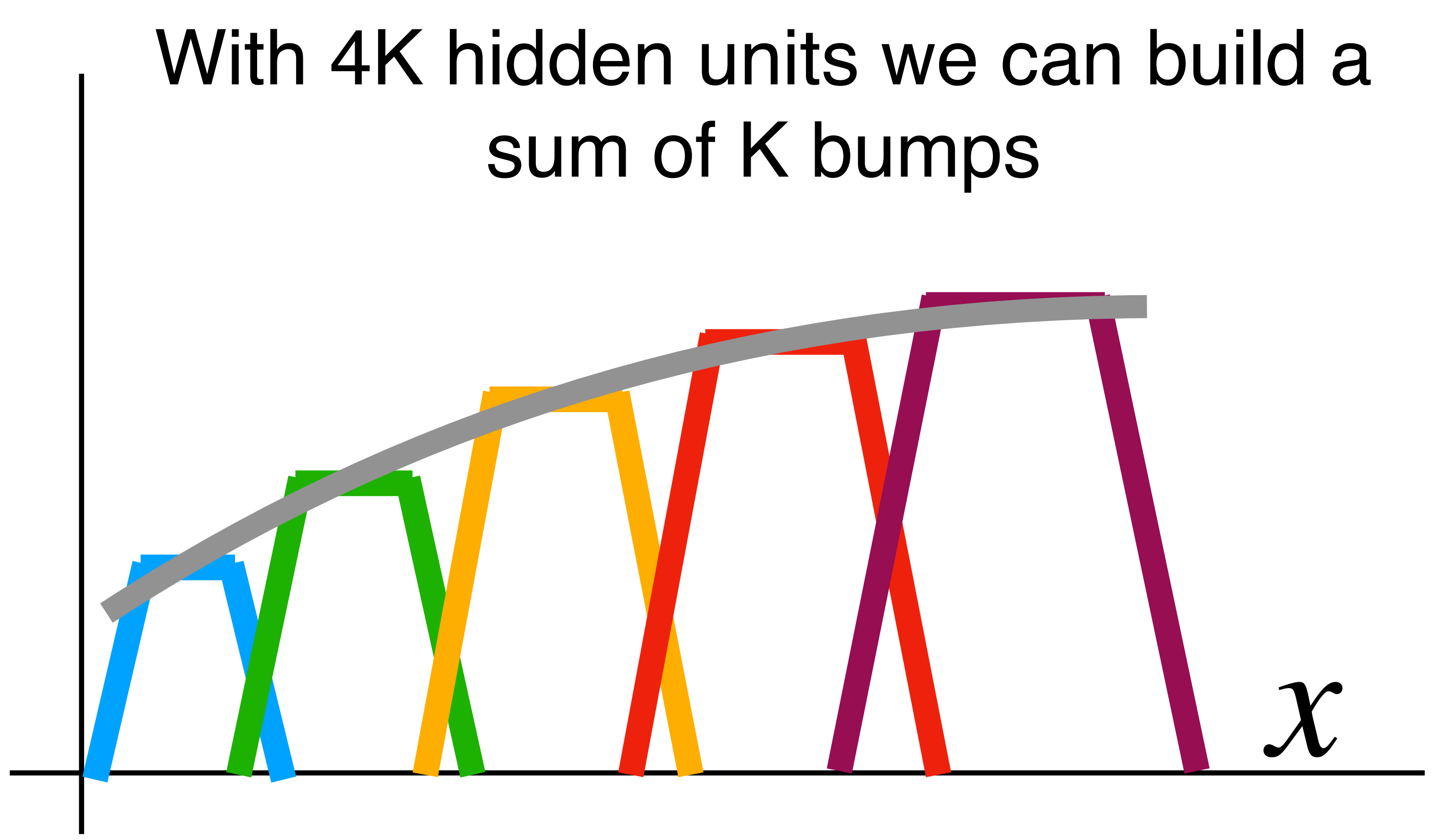
$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

What about …

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

See Nielsen, Chapter 4

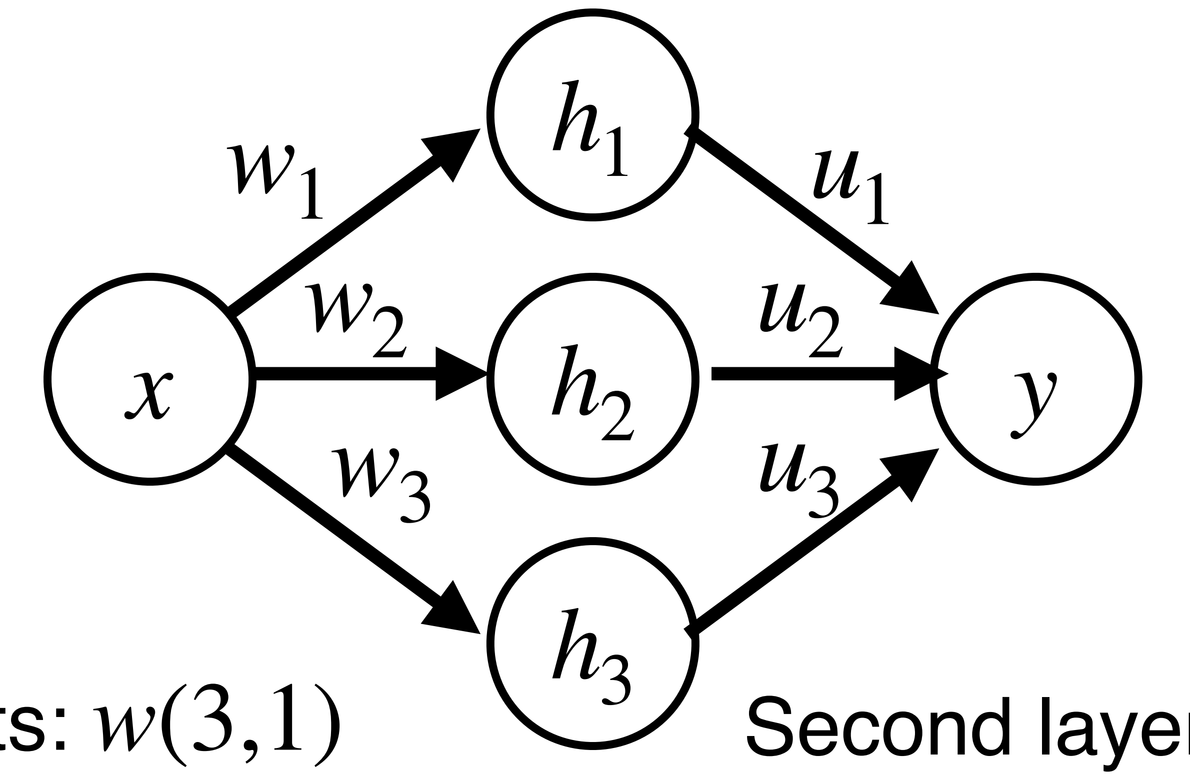


Approximate functions with bumps!

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

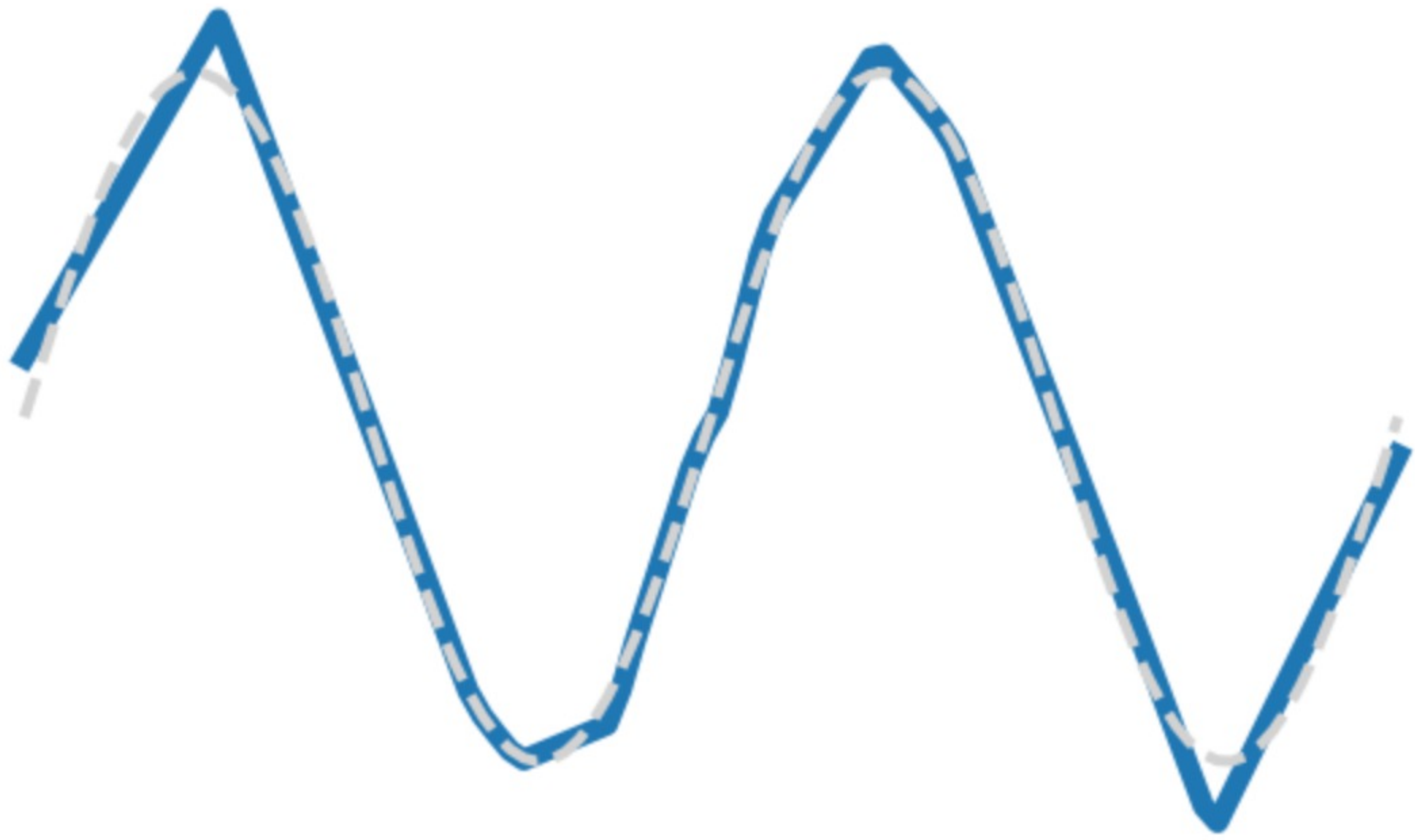
Second layer weights: $u(1,3)$

First layer bias: $p(1,)$

$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

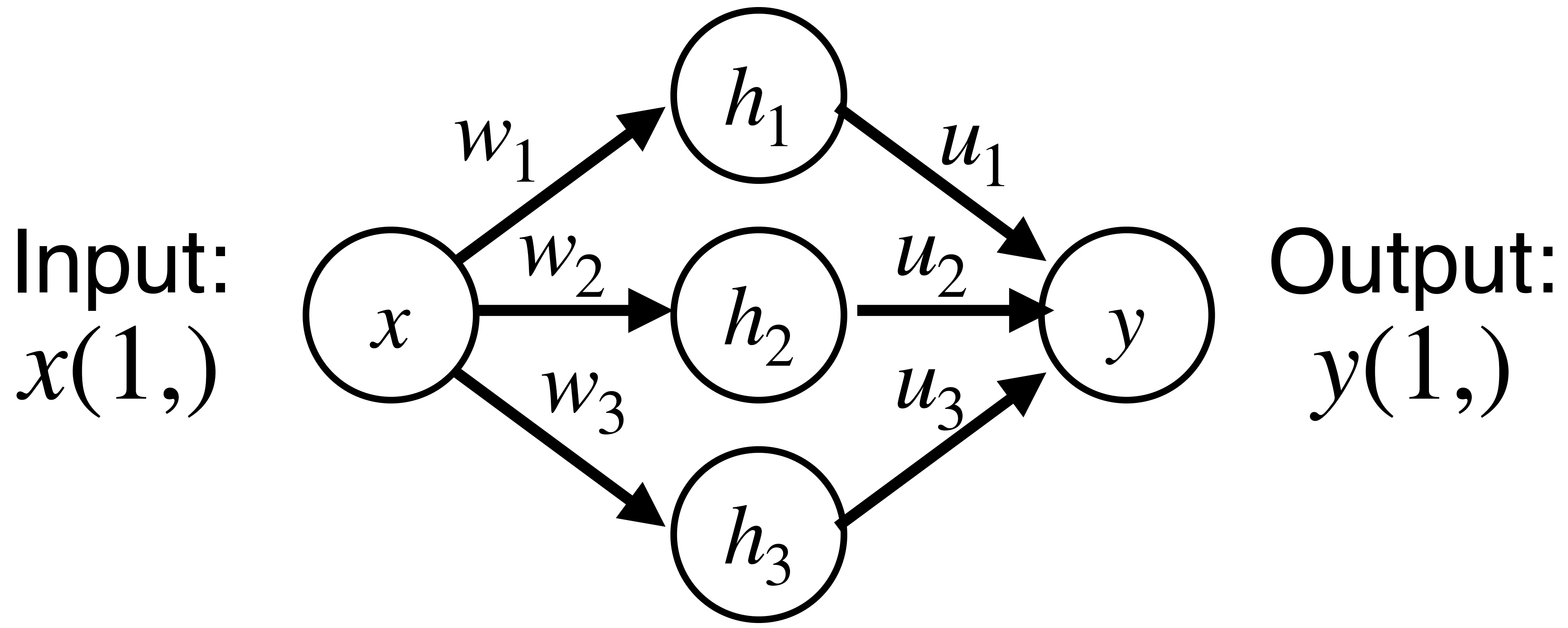
Reality check: Networks don't really learn bumps!



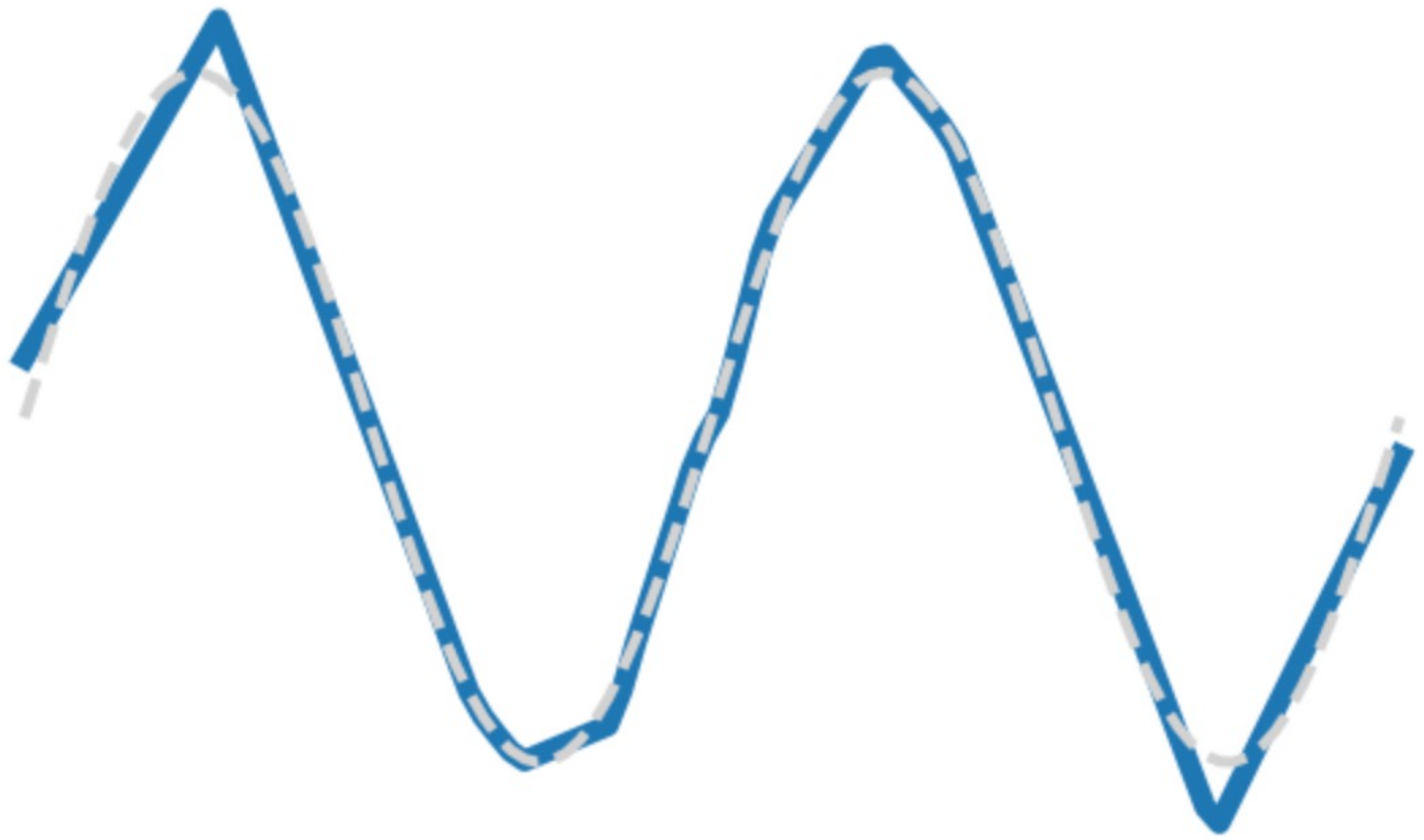
Approximate functions with bumps!

Universal Approximation

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



Reality check: Networks don't really learn bumps!



Universal approximation tells us:

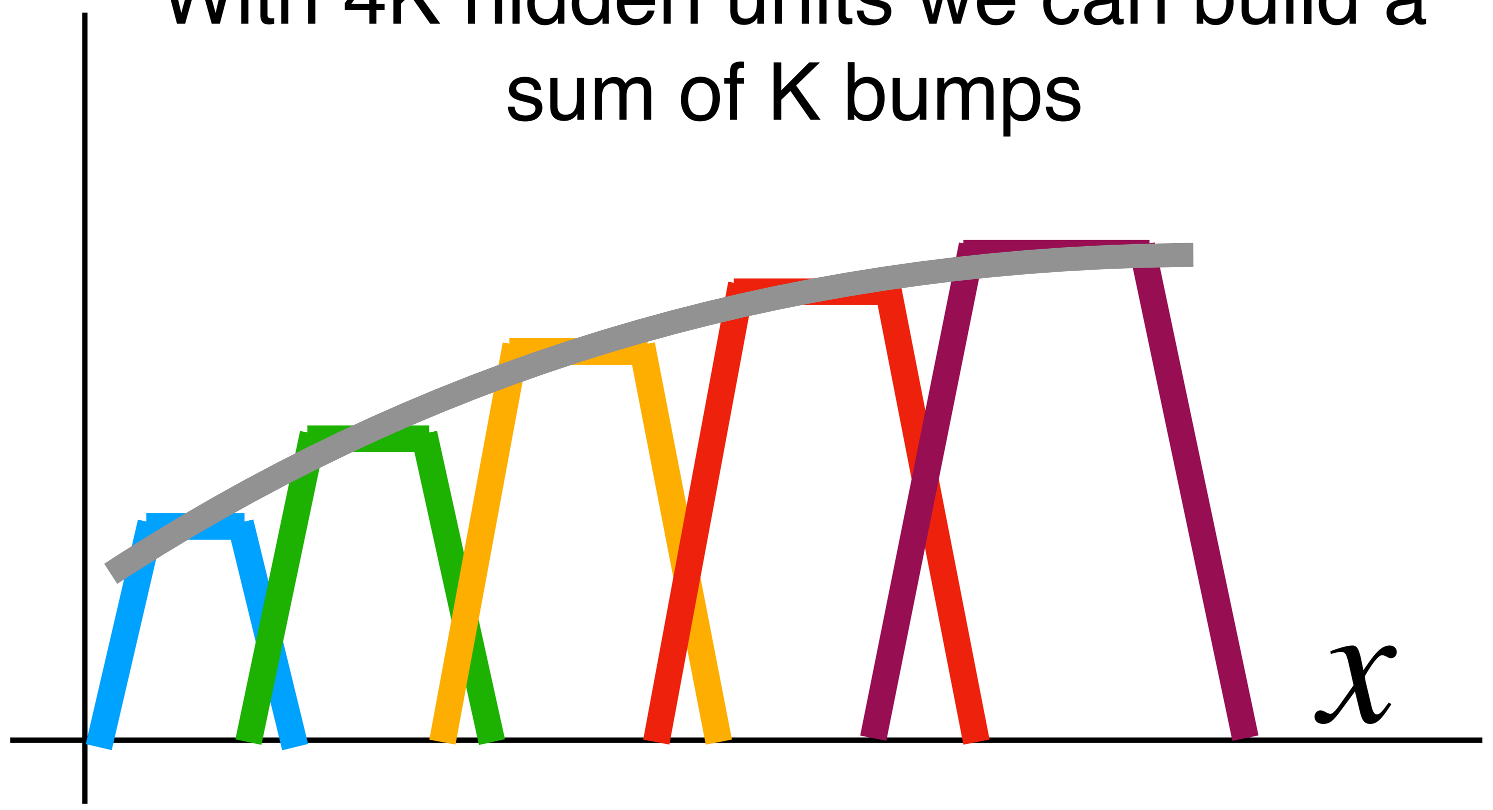
- Neural nets can represent any function

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!

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



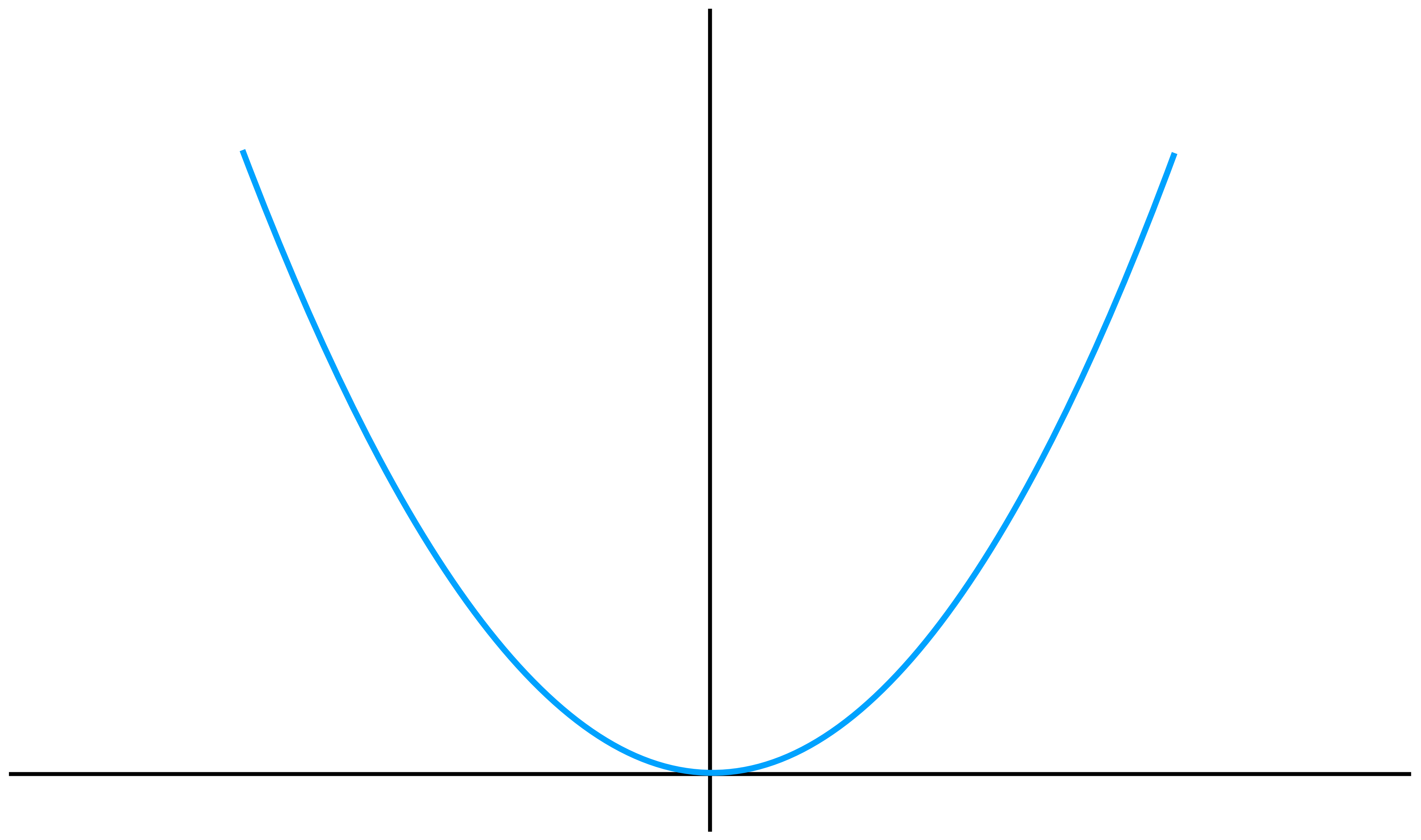
Approximate functions with bumps!

Convex Functions

A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ 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:

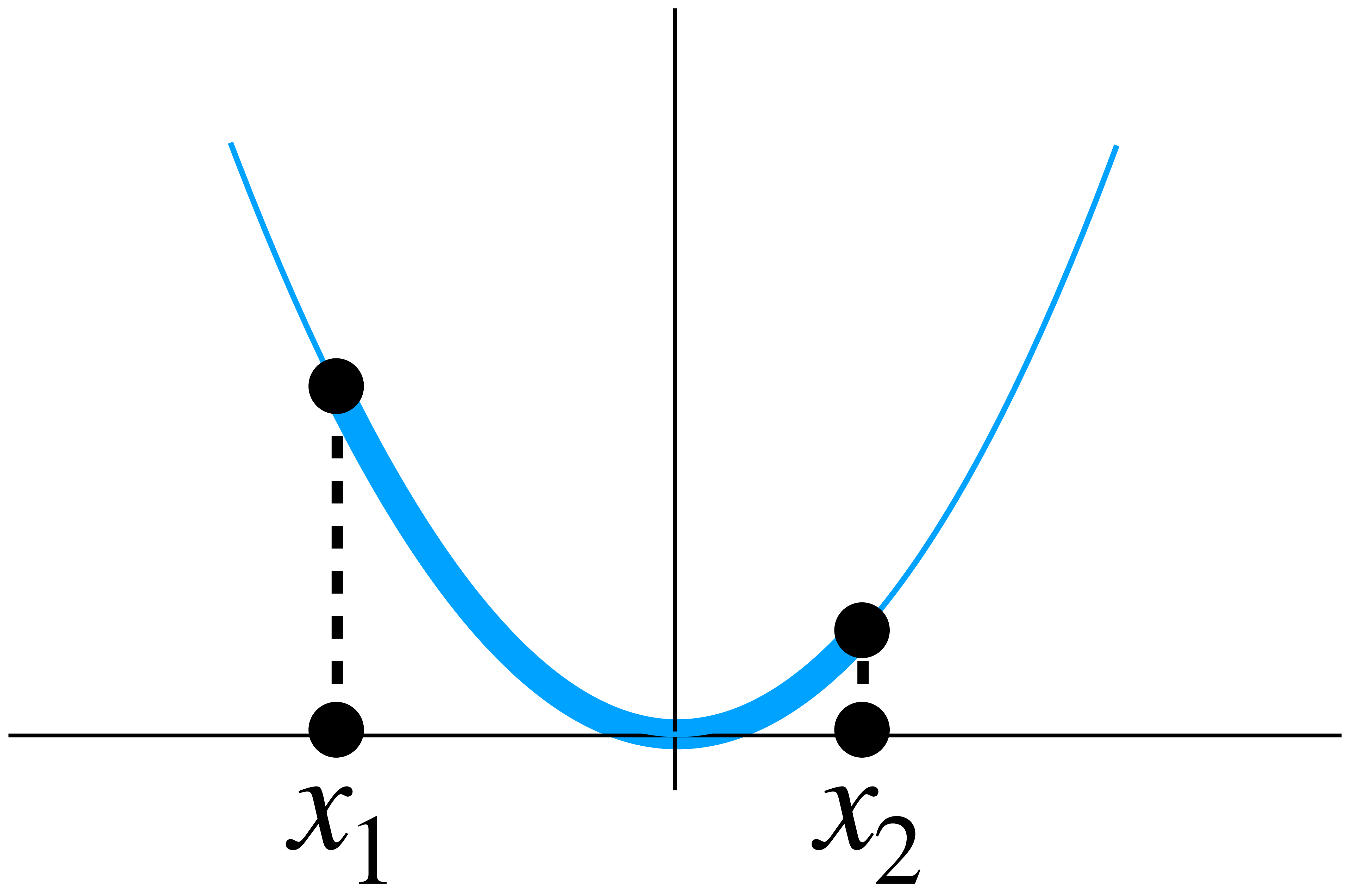


Convex Functions

A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ 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:

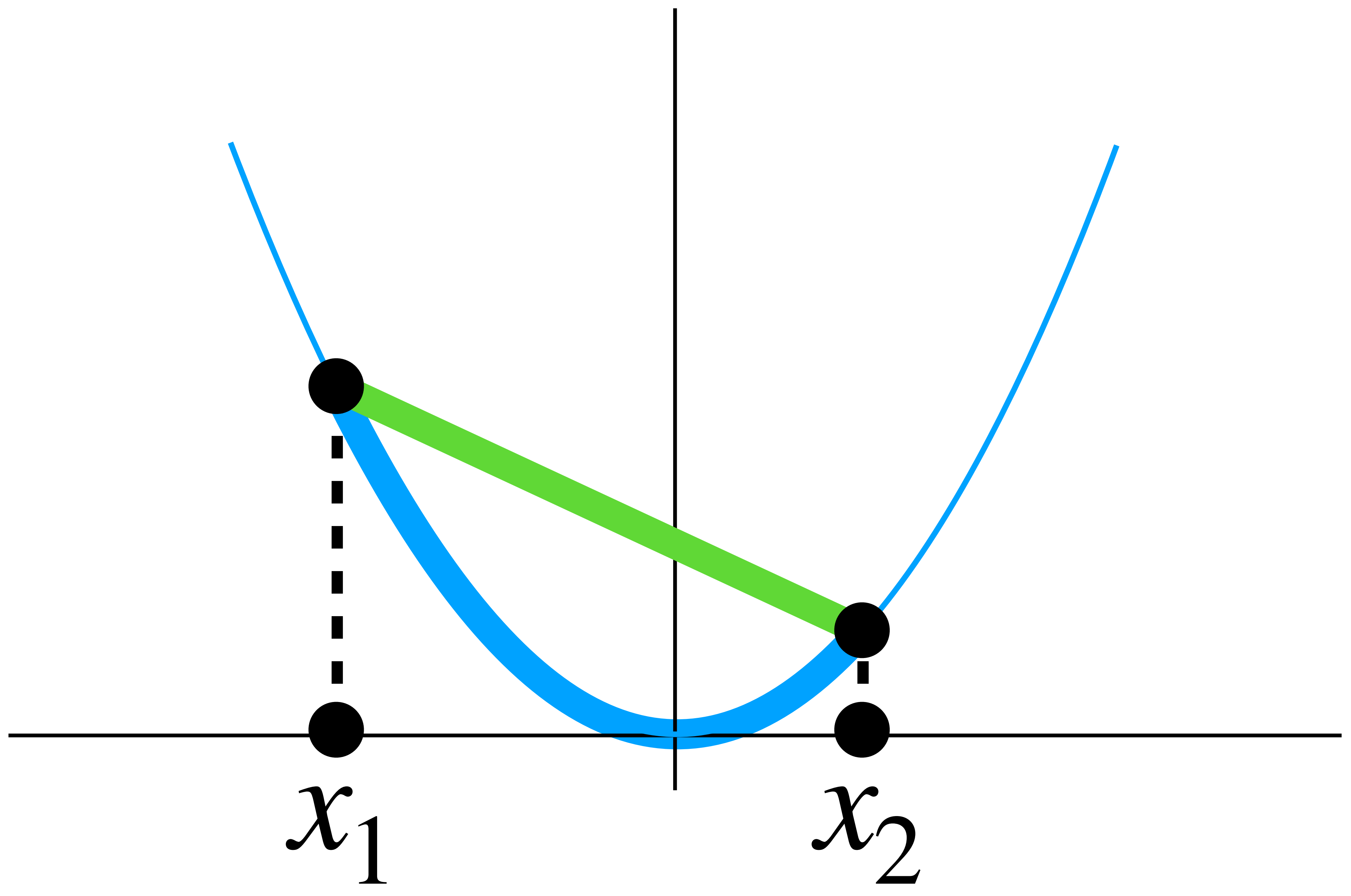


Convex Functions

A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ 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:

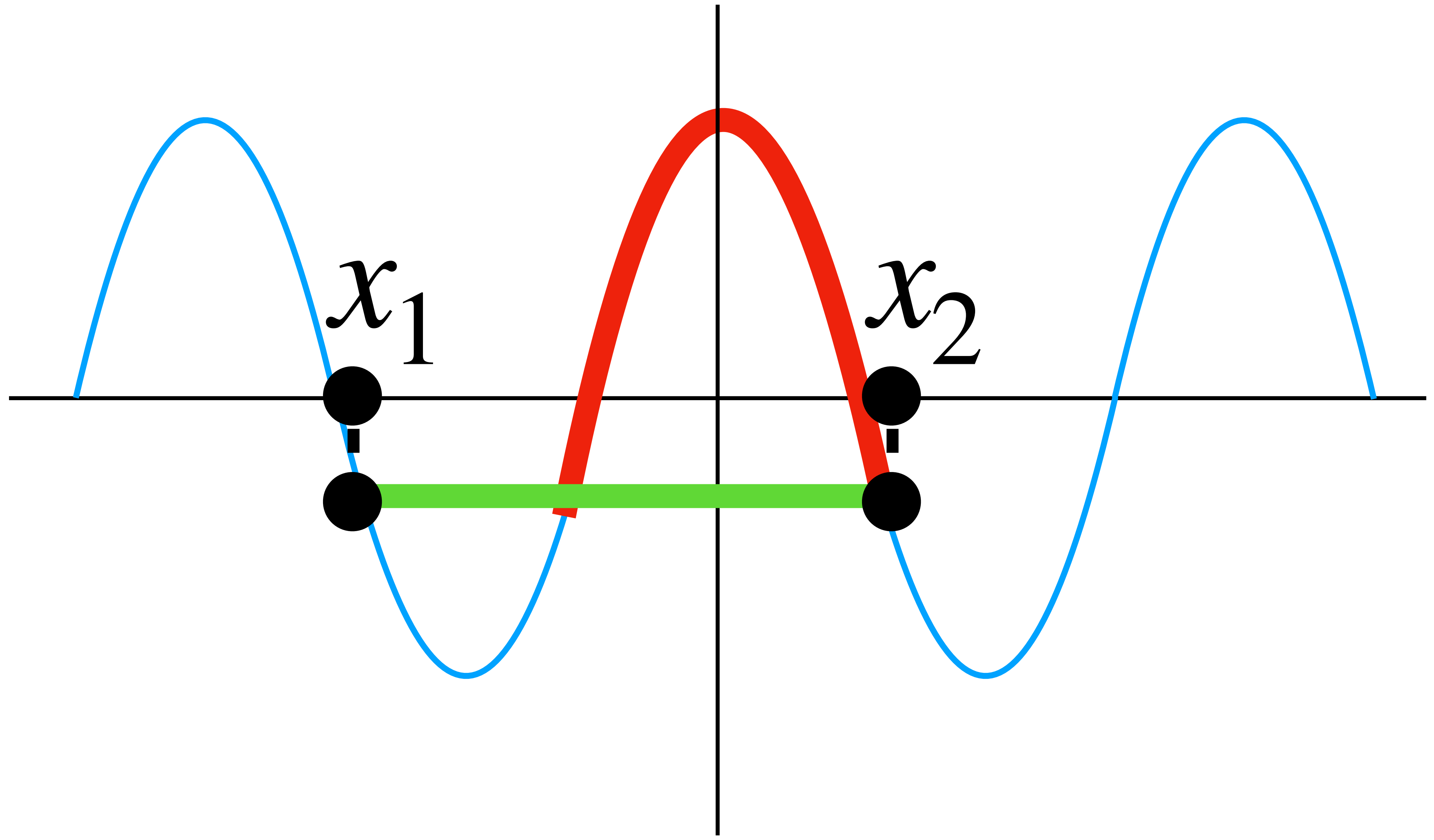


Convex Functions

A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ 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:



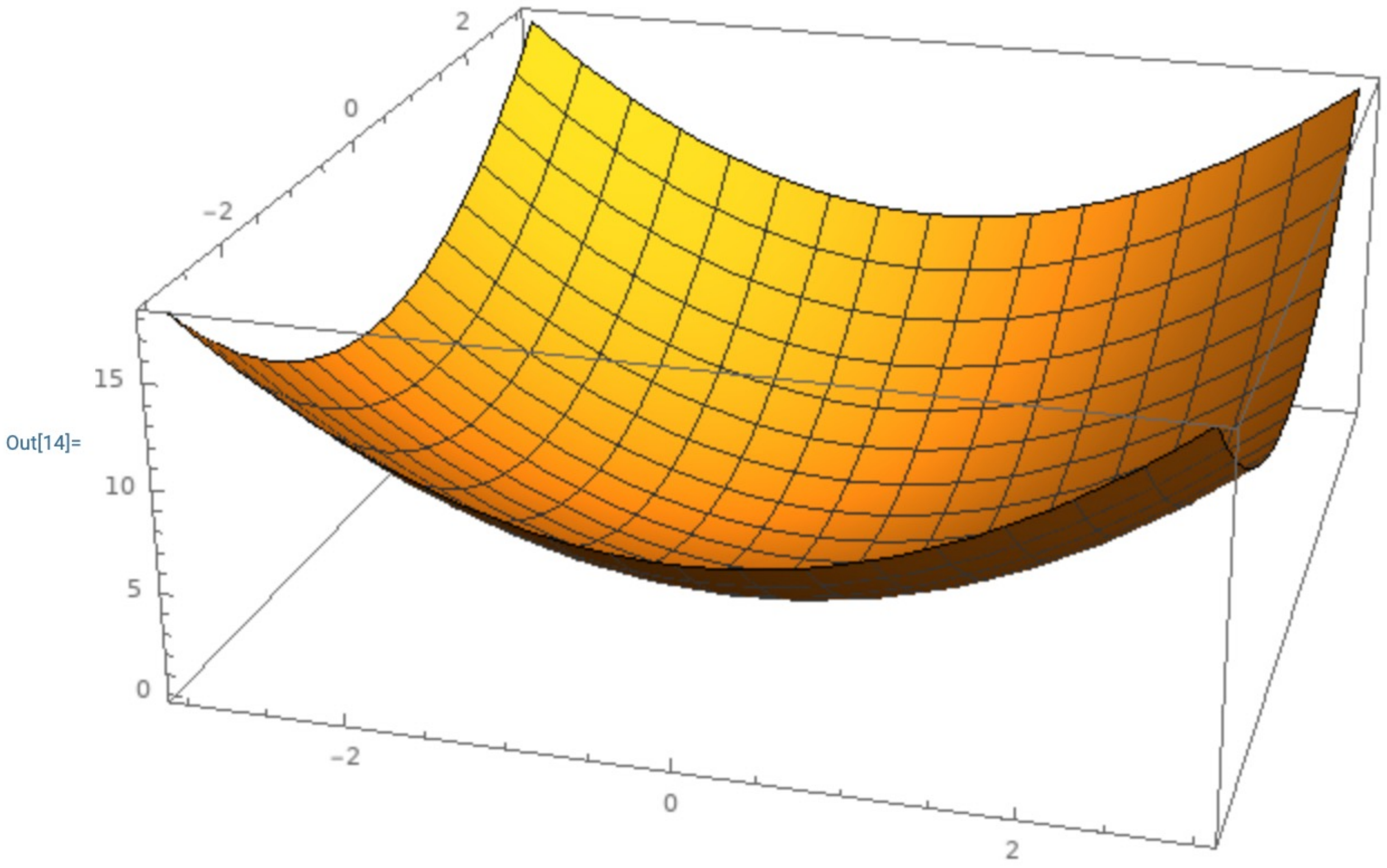
Convex Functions

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!**

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

$$L_i = -\log(\frac{e^{s_{y_i}}}{\sum + je^{s_j}})$$ Softmax

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

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

Convex Functions

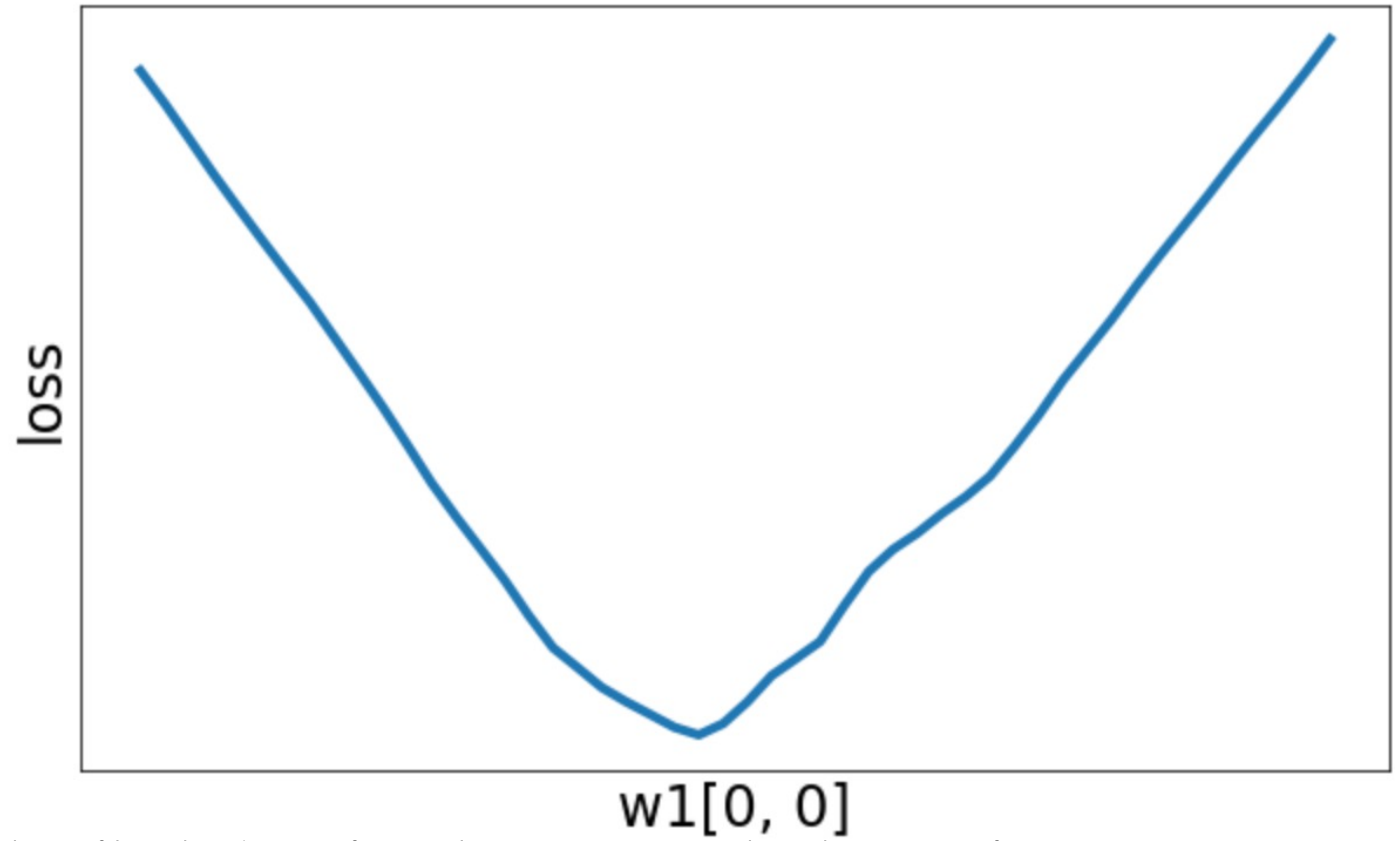
A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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:



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

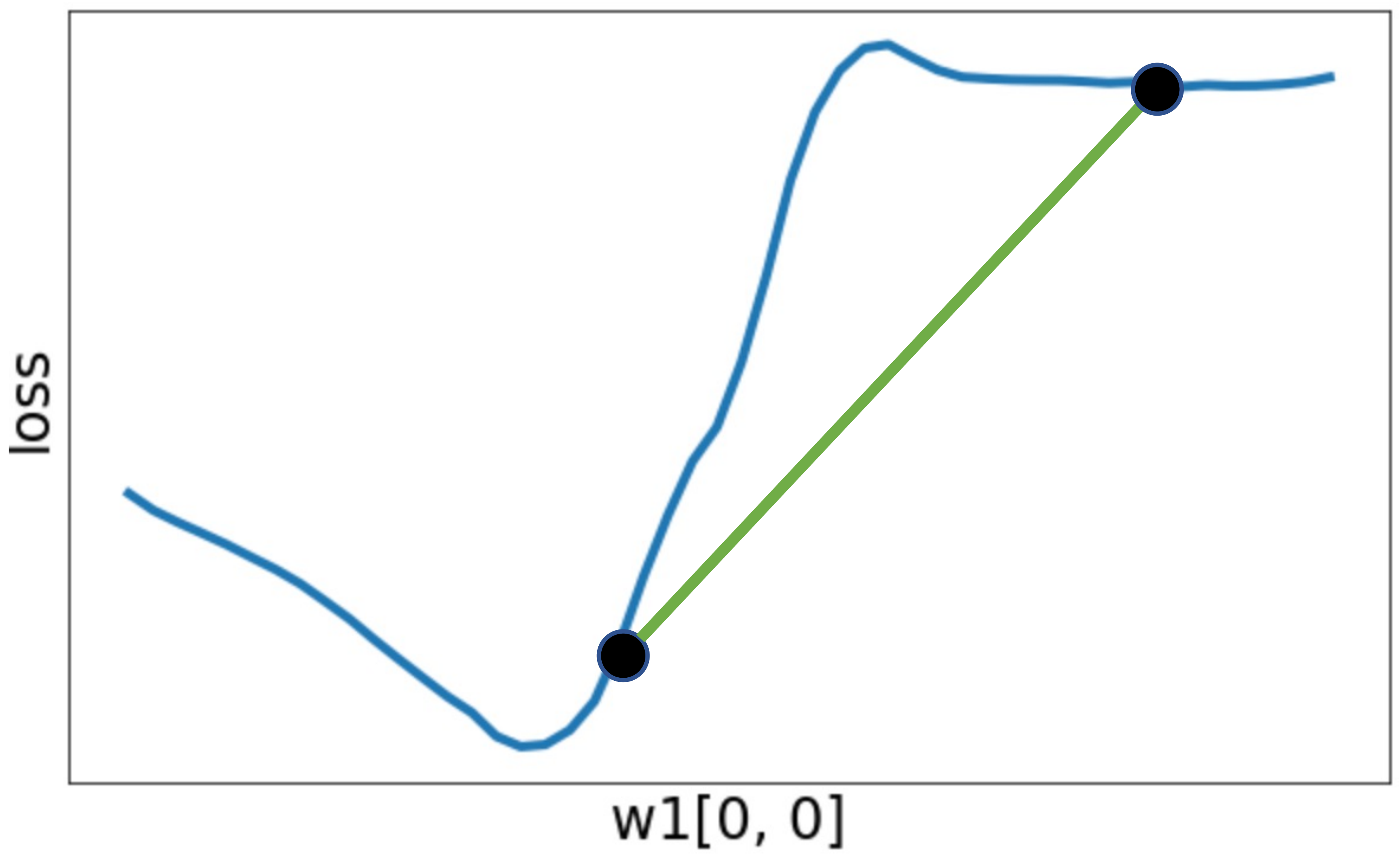
A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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

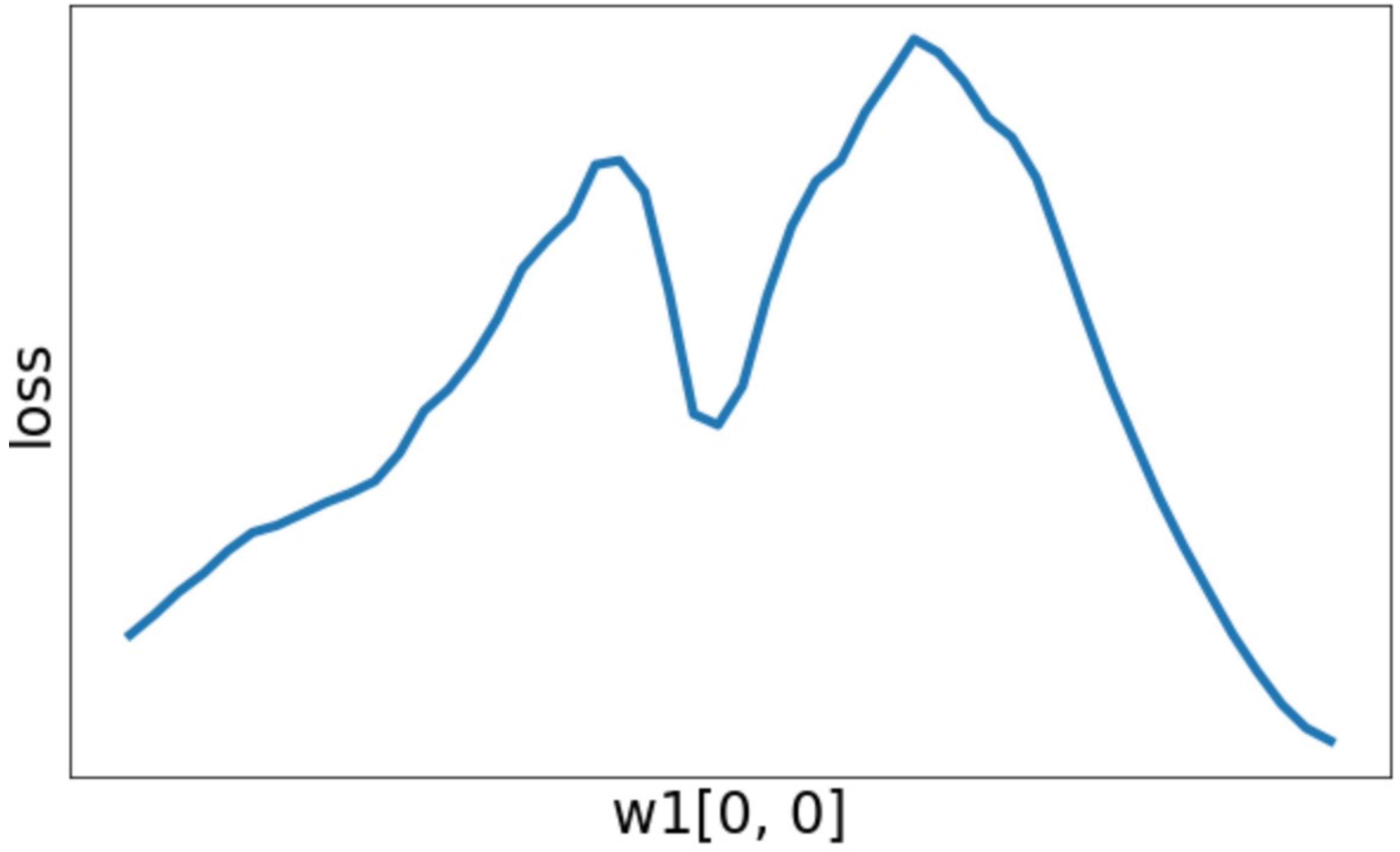
A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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

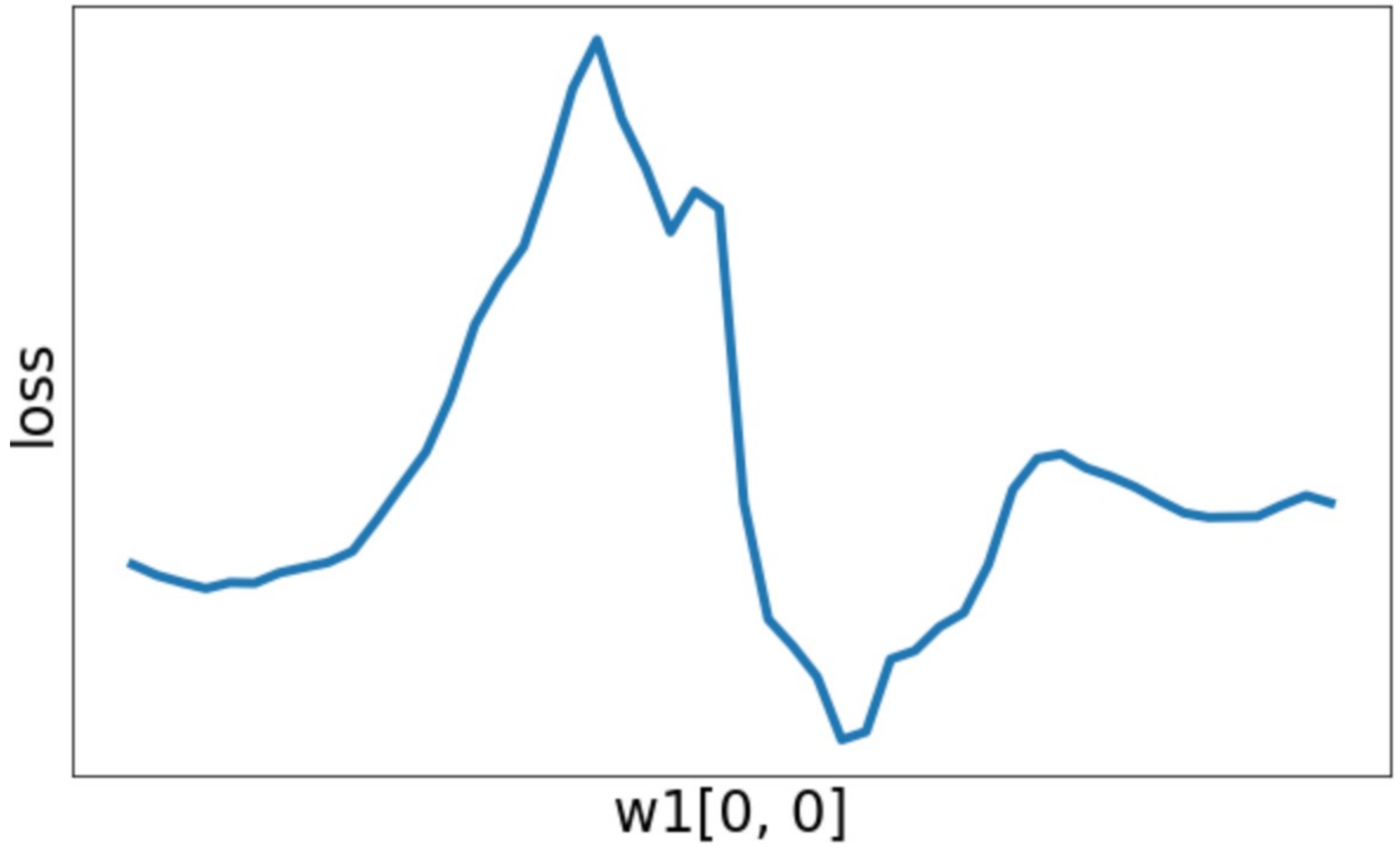
A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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 $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ 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

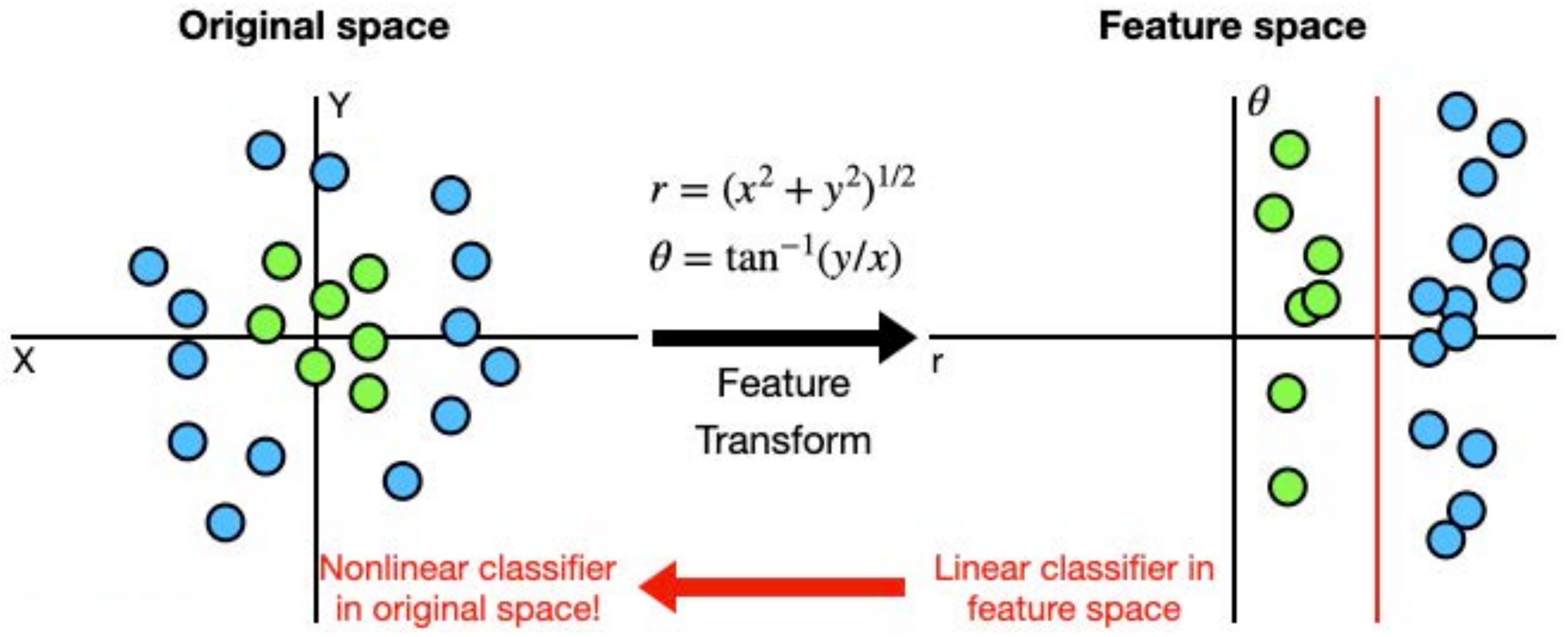
Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum***

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

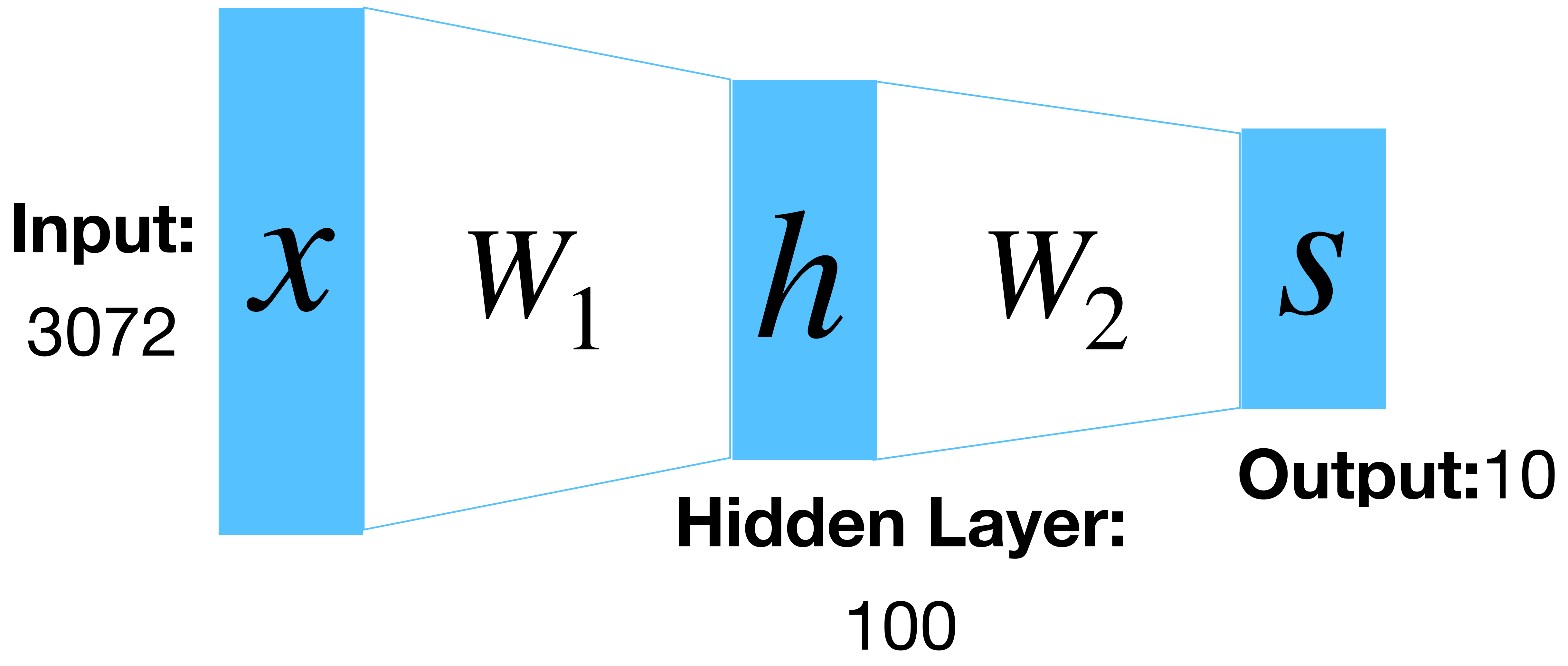


Neural Networks as learnable feature transforms

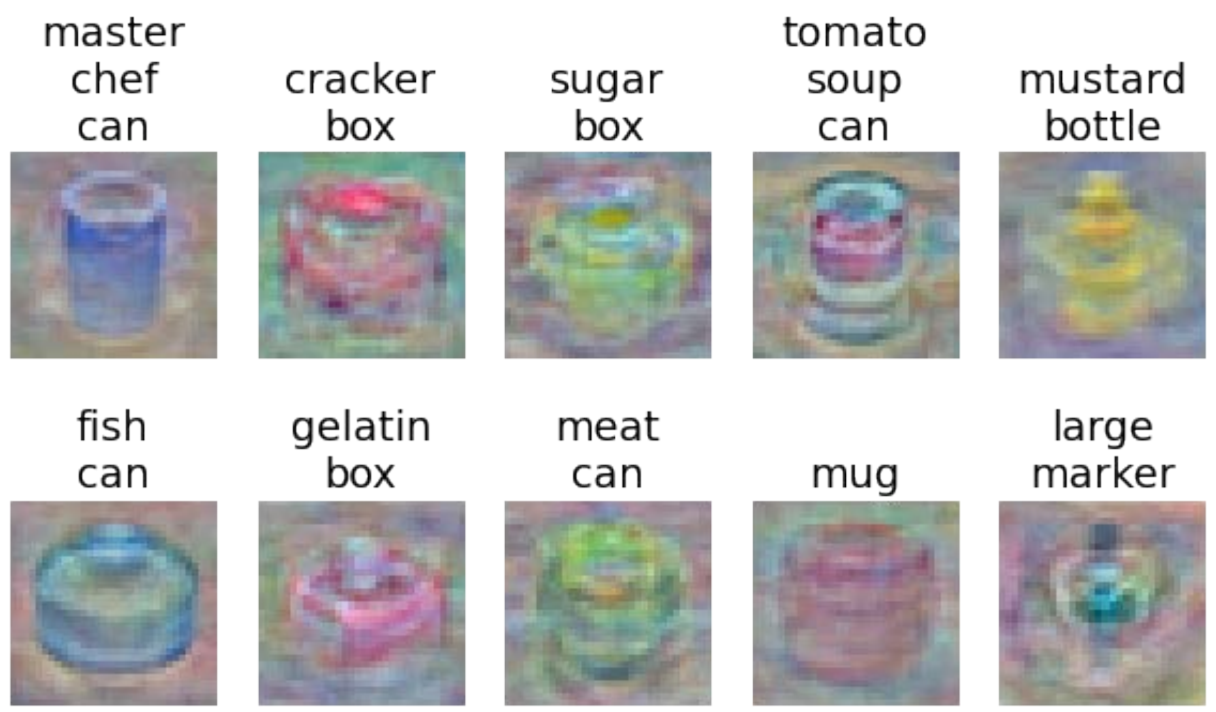
Summary

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

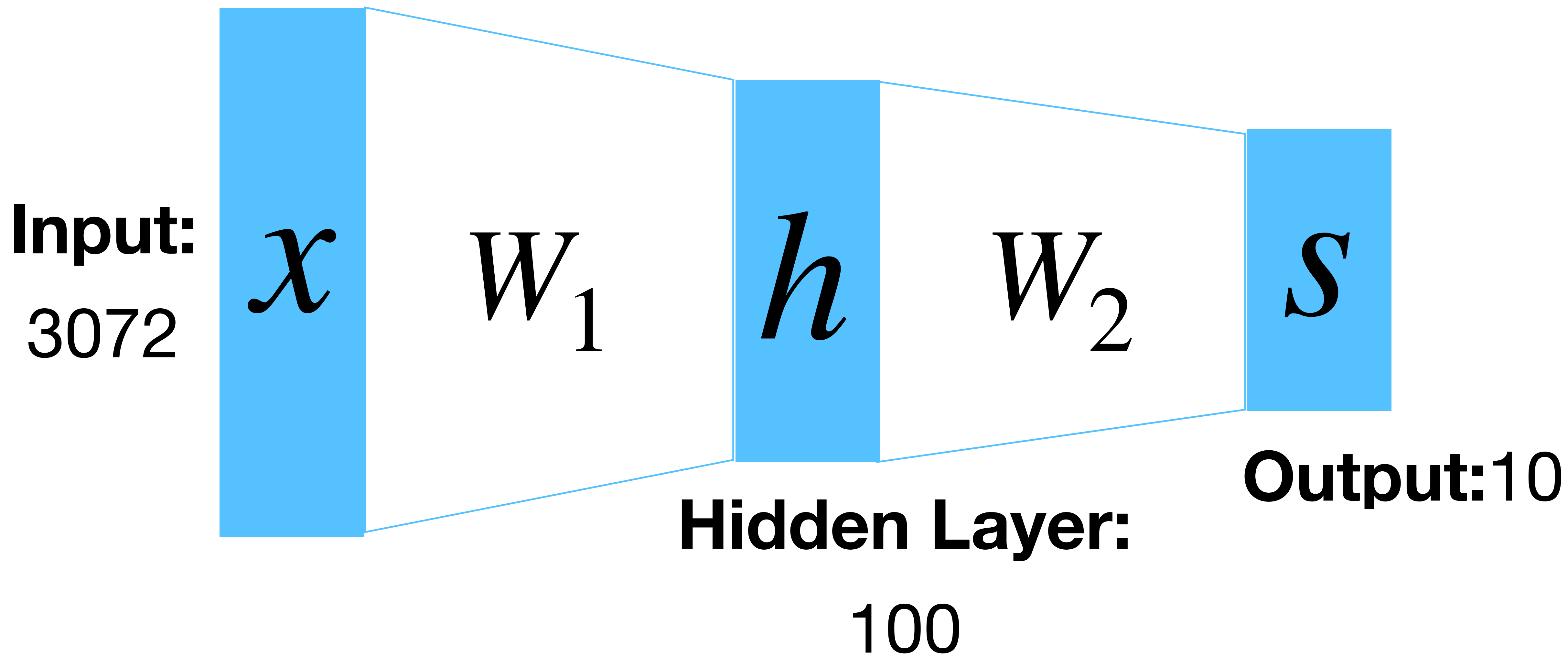


Neural networks: Many reusable templates

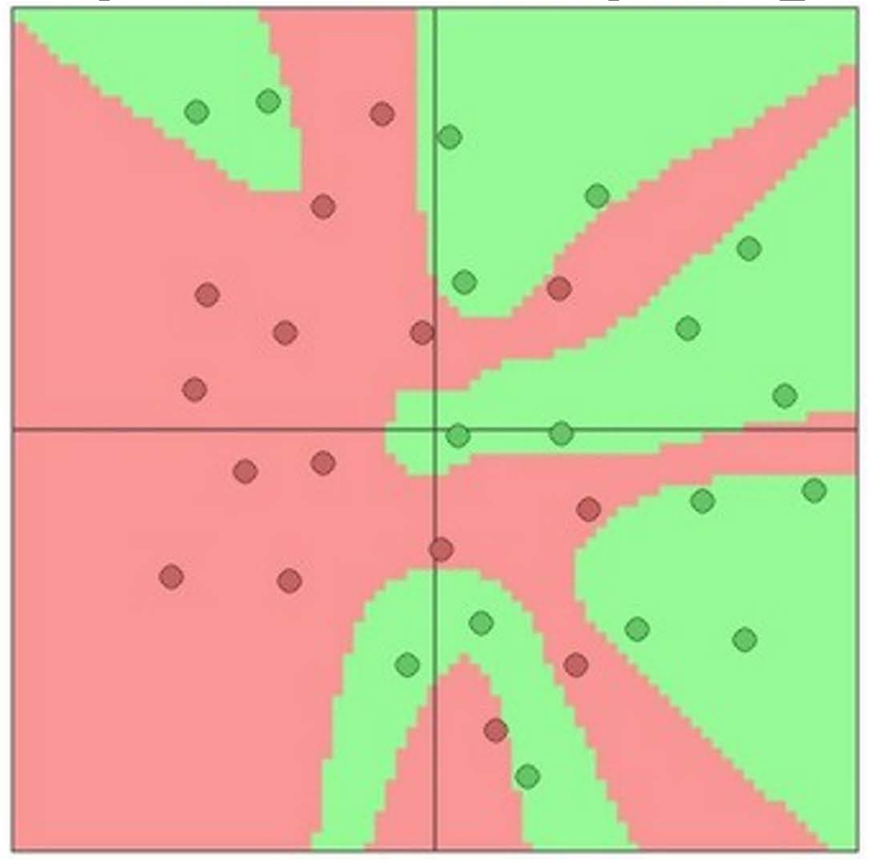
Summary

From linear classifiers to fully-connected networks

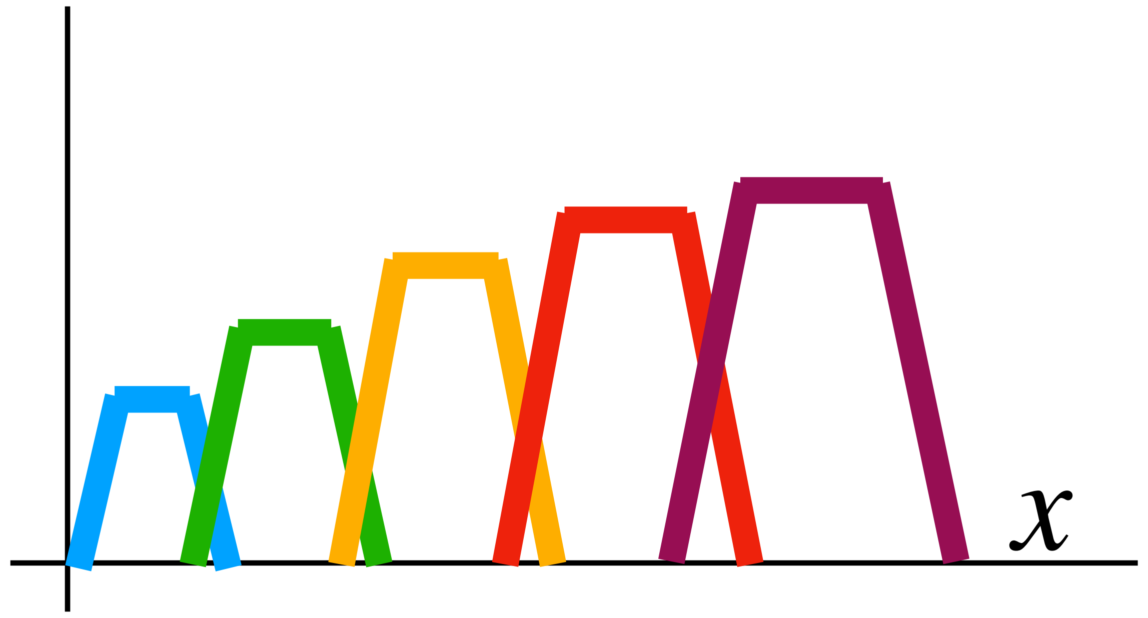
$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$



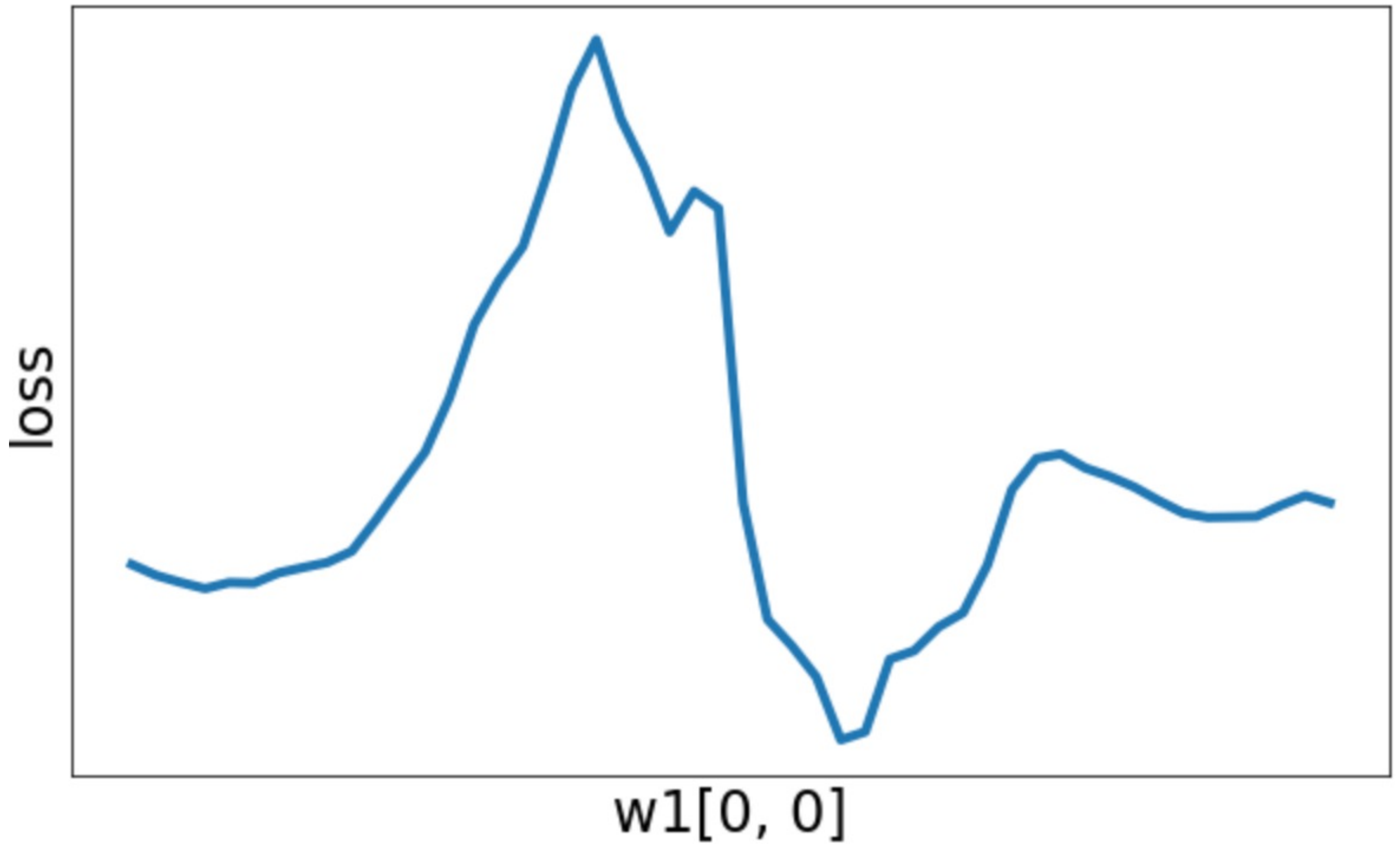
Space Warping



Universal approximation



Nonconvex



Problem: How to compute gradients?

$s = W_2 \max(0, W_1 x + b_!) + b_2$ Nonlinear score function

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

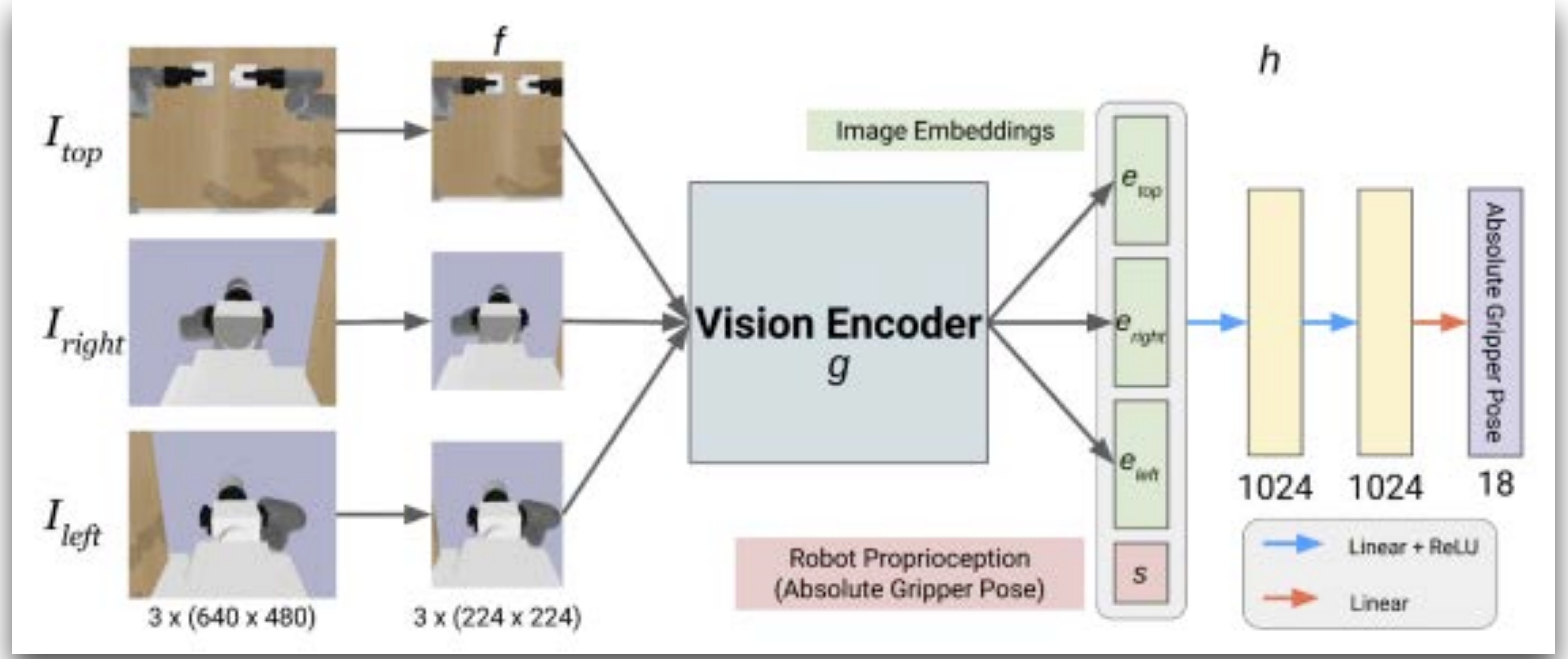
$R(W) = \sum_k W_k^2$ L2 regularization

$L(W_1, W_2, b_1, b_2) = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda R(W_1) + \lambda R(W_2)$ 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 time: Backpropagation

Problem Statements



C. Ku, C. Winge, R. Diaz, W. Yuan and K. Desingh, "Evaluating Robustness of Visual Representations for Object Assembly Task Requiring Spatio-Geometrical Reasoning," *2024 IEEE International Conference on Robotics and Automation (ICRA)*, Yokohama, Japan, 2024, pp. 831-837, doi: 10.1109/ICRA57147.2024.10610774.

III. IMITATION LEARNING FRAMEWORK

The goal of imitation learning is to train a policy $\pi : \mathcal{O} \rightarrow \mathcal{A}$ that maps all observations to an action that will progress the robot towards executing the task. Our dataset $\mathcal{D} = \{(O_i, a_i)\}_{i=1,\ldots N}$ consists of N observation-action pairs. Each observation $O_i = ((I_v)_{v\in V}, s)$ is a tuple of RGB images I_v from view $v \in V$ and the current state of the robot $s \in \mathcal{S}$. Each action $a_i \in \mathcal{A}$ is the action the expert performs during demonstration.

Our architecture shown in Fig. 2 is inspired by imitation learning evaluation frameworks from Robomimic [18], R3M [10], and MVP [11]. In our implementation, the policy π_θ consists of three parts: the image preprocessor f, the image encoder g_ϕ, and the policy head h_ψ. The image preprocessor $f : \mathcal{I}^{640\times480} \rightarrow \mathcal{I}^{224\times224}$ crops and resizes the original RGB image to a consistant size for fair comparison of all vision encoder models (ViT-B/16 requires this size). The image encoder $g_\phi : \mathcal{I}^{224\times224} \rightarrow \mathbb{R}^D$ is a neural network that deterministically maps an RGB image I_v to a D-dimensional image embedding e_v. The policy head $h_\psi : \mathbb{R}^D \times \ldots \times \mathbb{R}^D \times \mathcal{S} \rightarrow \mathcal{A}$ is a multi-layer perceptron on top of concatenated image embeddings $(e_v)_{v\in V}$ and robot state s which produces the final output action a. In summary, output of the policy $\pi_\theta(O) = h_\psi(g_\phi(f(I_{v_1})), \ldots, g_\phi(f(I_{v_{|V|}})), s) = \hat{a}$ is the predicted action. We train either the parameters ψ (frozen g) or both ϕ and ψ (unfrozen g) by back-propagating the mean squared error loss $\mathcal{L} = MSE(a, \hat{a})$.

We choose $\mathcal{S} \subseteq SE(3)\times SE(3)$ and $\mathcal{A} \subseteq SE(3)\times SE(3)$ to be absolute gripper poses of both arms. For each arm, there are 3 values representing xyz position and 6 values representing the first two columns of the rotation matrix [25], so we represent both as a 18 dimensional vector. The reason for this choice is explained in Sec. V.

Problem Statements

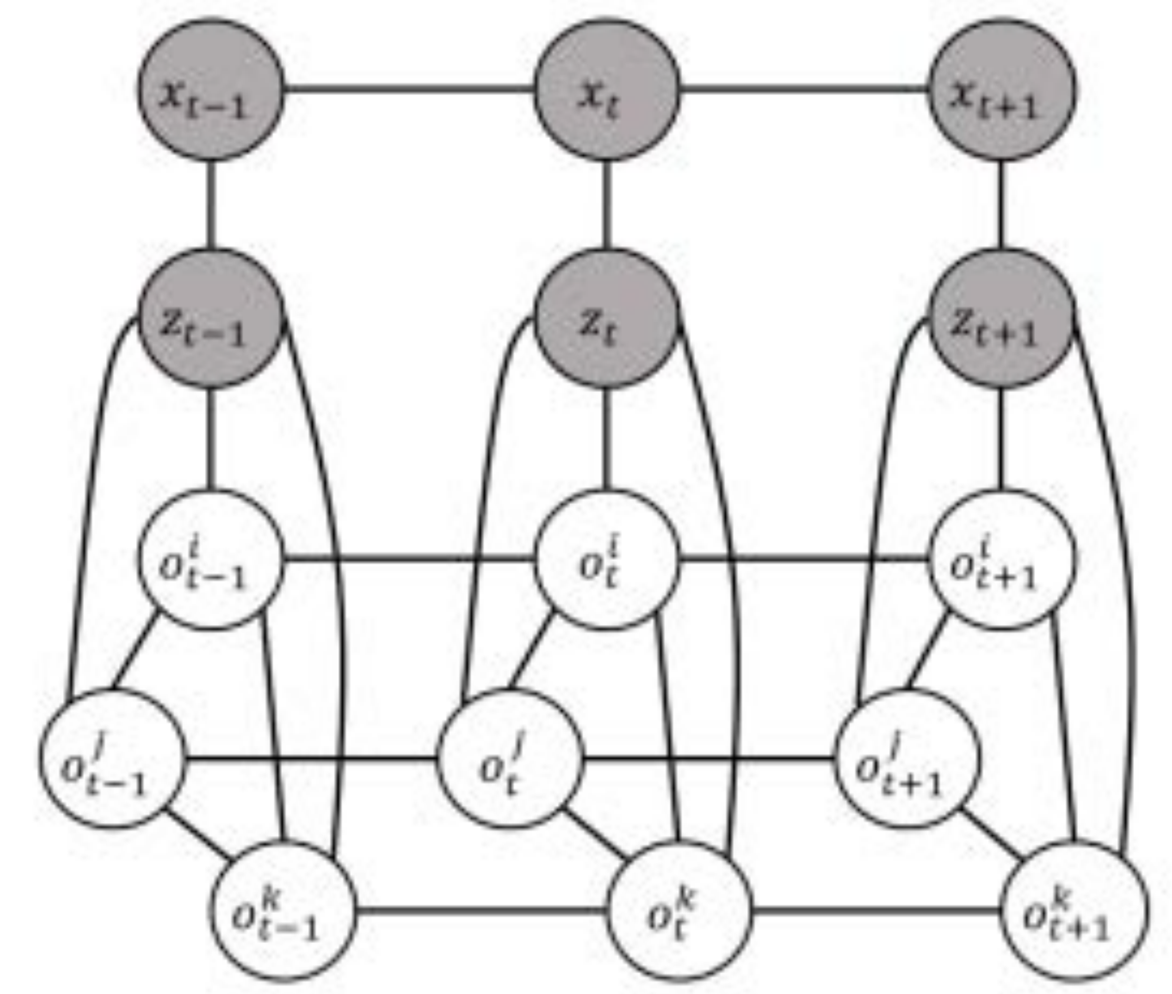


Fig. 2: Graphical model of the semantic mapping problem. Observed variables are robot poses x_t and observations z_t. Unknown variables are objects $\{o^1, o^2, \cdots, o^N\}$. We compute the posterior over objects while modeling contexual relations between all pairs of objects at each time point, and temporal consistency of each object across consecutive time points.

Z. Zeng, Y. Zhou, O. C. Jenkins and K. Desingh, "Semantic Mapping with Simultaneous Object Detection and Localization," *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Madrid, Spain, 2018, pp. 911-918, doi: 10.1109/IROS.2018.8594205.

III. Problem Formulation

We focus on semantic mapping at the object level. Our proposed *CT-Map* method maintains a belief over object classes and poses across an observed scene. We assume that the robot stays localized in the environment through an external localization routine (e.g., ORB-SLAM [23]). The semantic map is composed by a set of N objects $O = \{o^1, o^2, \cdots, o^N\}$. Each object $o^i = \{o^c, o^g, o^\psi\}$ contains the object class $o^c \in \mathcal{C}$, object geometry o^g, and object pose o^ψ, where $\mathcal{C}$ is the set of object classes $\mathcal{C} = \{c_1, c_2, \cdots, c_n\}$.

At time t, the robot is localized at x_t. The robot observes $z_t = \{I_t, S_t\}$, where I_t is the observed RGB-D image, and S_t are semantic measurements. The semantic measurements $s_k = \{s^s_k, s^b_k\} \in S_t$ are returned by an object detector (as explained in section V-A), which contains: 1) a object detection score vector s^s_k, with each element in s^s_k denoting the detection confidence of each object class, and 2) a 2D bounding box s^b_k.

We probabilistically formalize the semantic mapping problem in the form of a CRF, as shown in Figure 2. Robot pose x_t and observation z_t are known. The set of objects O are unknown variables. We model the contextual dependencies between objects and the temporal consistency of each individual object over time. The posterior probability of the semantic map is expressed as:

$$p(O_{0:T} | x_{0:T}, z_{0:T}) = \frac{1}{Z} \prod_{t=0}^{T} \prod_{i=1}^{N} \phi_p(o^i_t, o^i_{t-1}, u^i_{t-1}) \phi_m(o^i_t, x_t, z_t) \prod_{i,j} \phi_c(o^i_t, o^j_t) \quad (1)$$

where Z is a normalization constant, and action applied to object o^i at time t is denoted by u^i_t. ϕ_p is the *prediction potential* that models the temporal consistency of the object poses. ϕ_m is the *measurement potential* that accounts for the observation model given 3D mesh of objects. ϕ_c is the *context potential* that captures the contextual relations between objects.

Problem Statements

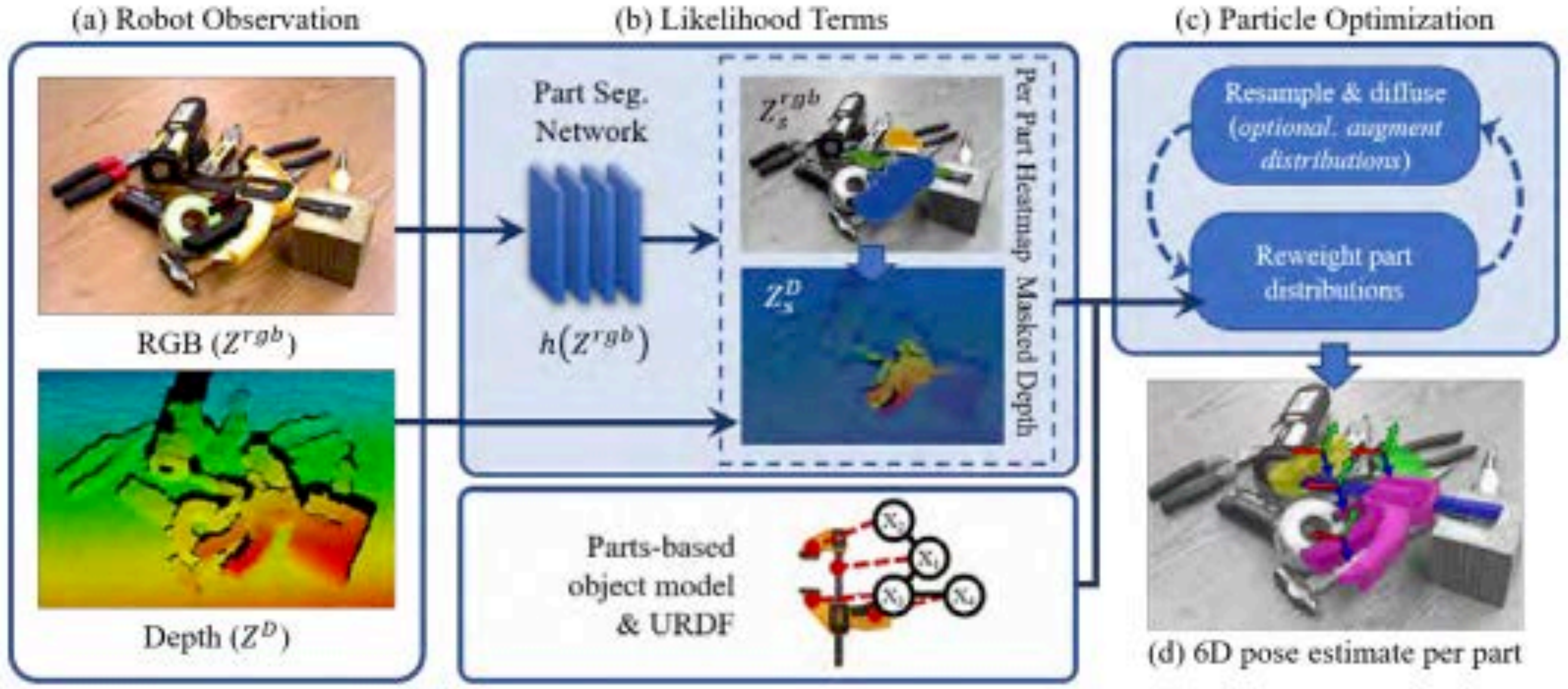


Fig. 3: The inference pipeline. (a) The robot observes a scene as an RGB-D image, $\mathcal{Z} = (Z^{rgb}, Z^D)$. (b) The RGB image is passed through a trained part segmentation network, $h(Z^{rgb})$, that generates a pixel-wise heatmap for the P_k parts of an object class of interest, $\{Z_s^{rgb}\}_{s=1}^{P_k}$ (in this example, the clamp, which has one fully occluded part). The heatmaps are used to generate masked depth images, $\{Z_s^D\}_{s=1}^{P_k}$ (c) The inference is initialized with part poses using these heatmaps and the depth image. Hypotheses are iteratively reweighed using Equation 4, and resampled with importance sampling. (d) The inference process generates an estimate of the 6D pose of each part. (Best viewed in color).

J. Pavlasek, S. Lewis, K. Desingh and O. C. Jenkins, "Parts-Based Articulated Object Localization in Clutter Using Belief Propagation," *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Las Vegas, NV, USA, 2020, pp. 10595-10602, doi: 10.1109/IROS45743.2020.9340908.

III. Problem Statement

Given a scene containing objects O, such that $\{O_k\}_{k=1}^K$ is the set of K relevant objects, we wish to localize each object O_k. The state of an object O_k is represented by the set of part poses $\mathcal{X} = \{X_i\}_{i=1}^{P_k}$, where X_s is the 6D pose of an articulating rigid part s of O_k, with P_k parts. Each object O_k in the scene is estimated independently.

This estimation problem is formulated as a Markov Random Field (MRF). Let $G = (V, E)$ denote an undirected graph with nodes V and edges E. An example MRF is illustrated in Figure 2. The joint probability of the graph G is expressed as:

$$p(\mathcal{X}, \mathcal{Z}) \propto \prod_{(s,t)\in E} \psi_{s,t}(X_s, X_t) \prod_{s\in V} \phi_s(X_s, Z_s) \qquad (1)$$

where $\mathcal{X}$ denotes the hidden state variables to be inferred and $\mathcal{Z}$ denotes the observed sensor information in the form of an RGB-D image. The function $\psi_{s,t}$ is the *pairwise potential*, describing the correspondence between part poses based on the articulation constraints, and ϕ_s is the *unary potential*, describing the correspondence of a part pose X_s with its observation Z_s. The problem of pose estimation of an articulated model O_k is interpreted as the problem of estimating the marginal distribution of each part pose, called the belief, $bel(X_s)$.

In addition to the sensor data, the articulation constraints and 3D geometry of the object, in the form of a Unified Robot Description Format (URDF), and the 3D mesh models of the objects are provided as inputs. We assume that the object articulations are produced by either fixed, prismatic or revolute joints. We consider scenes which contain only one instance of an object. In Section IV, our proposed inference mechanism is detailed, along with a description of our modelling of the potentials in Equation 1.

Brainstorming task due 09/23

- Pick one of the ideas that you listed.
- Abstract the ideas into a general formulation.
- Write down variables with mathematical notations.
- Write down assumptions and technique that is suitable for this.
- Search for a paper that suits your formulation.

An example:

[1]: Robot Making Coffee using a Mug and K-Cup Pod Coffee Maker: A Robot with a single arm and a camera should be able to sense the mug on the table, and pick and place it in the K-Cup Pod Coffee Maker. Then, it should open the K-Cup lid. Pick up the K-cup from a tray, insert it into the coffee maker, closer the lid. The robot should press the button and wait for the coffee to be poured. The robot should pick up the filled coffee mug and place it on the table.

↓

Given an observation of the scene in the form of RGB image $I \in R^{w \times h \times 3}$, that contains three objects $O = \{o_{mug}, o_{kcup}, o_{coffee-m}\}$, this project aims to develop a learning method that can produce actions $A = (x, y, z, \pi, \phi, \rho, g)$ where x, y, z are the gripper position to reach, π, ϕ, ρ are the orientation of the gripper, together representing $SE(3)$ pose of the gripper. g denotes whether the gripper should be open or close at the end of the action. For this task of coffee making, we aim to develop an end-to-end learning method that can implicitly determine the objects' and their locations, as well as actions to execute from a number of demonstration data that can go from $I \rightarrow A$.

↓

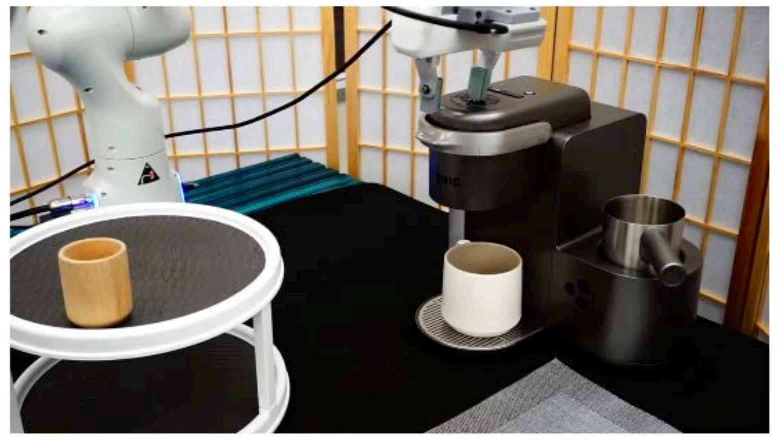
VIOLA: Imitation Learning for Vision-Based Manipulation with Object Proposal Priors

Yifeng Zhu[1] Abhishek Joshi[1] Peter Stone[1, 2] Yuke Zhu[1]

[1]The University of Texas at Austin [2]Sony AI

Paper | Video | Code | Bibtex

6th Conference on Robot Learning, Auckland, New Zealand



DeepRob

Lecture 5
Neural Networks
University of Minnesota