DeepRob

Lecture 5
Neural Networks
University of Michigan and University of Minnesota

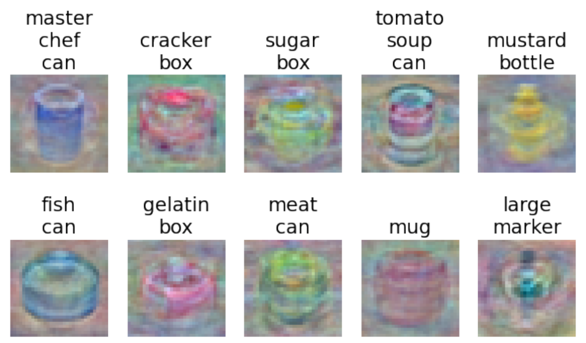
Project 1—Reminder

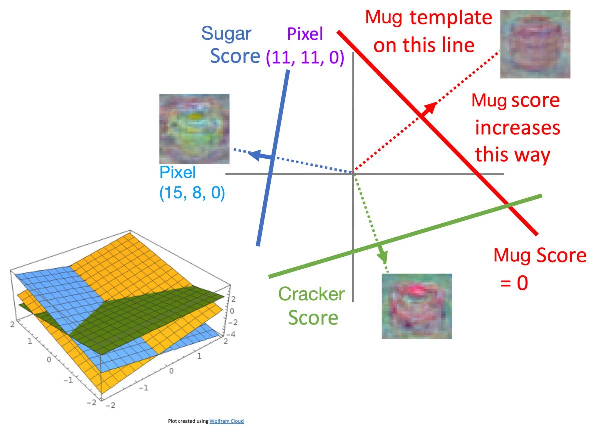
- Instructions and code available on the website
  - Here: deeprob.org/projects/project1/
- Uses Python, PyTorch and Google Colab
- Implement KNN, linear SVM, and linear softmax classifiers
- **Autograder is online and updated**
- **Due Thursday, January 26th 11:59 PM EST**

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**

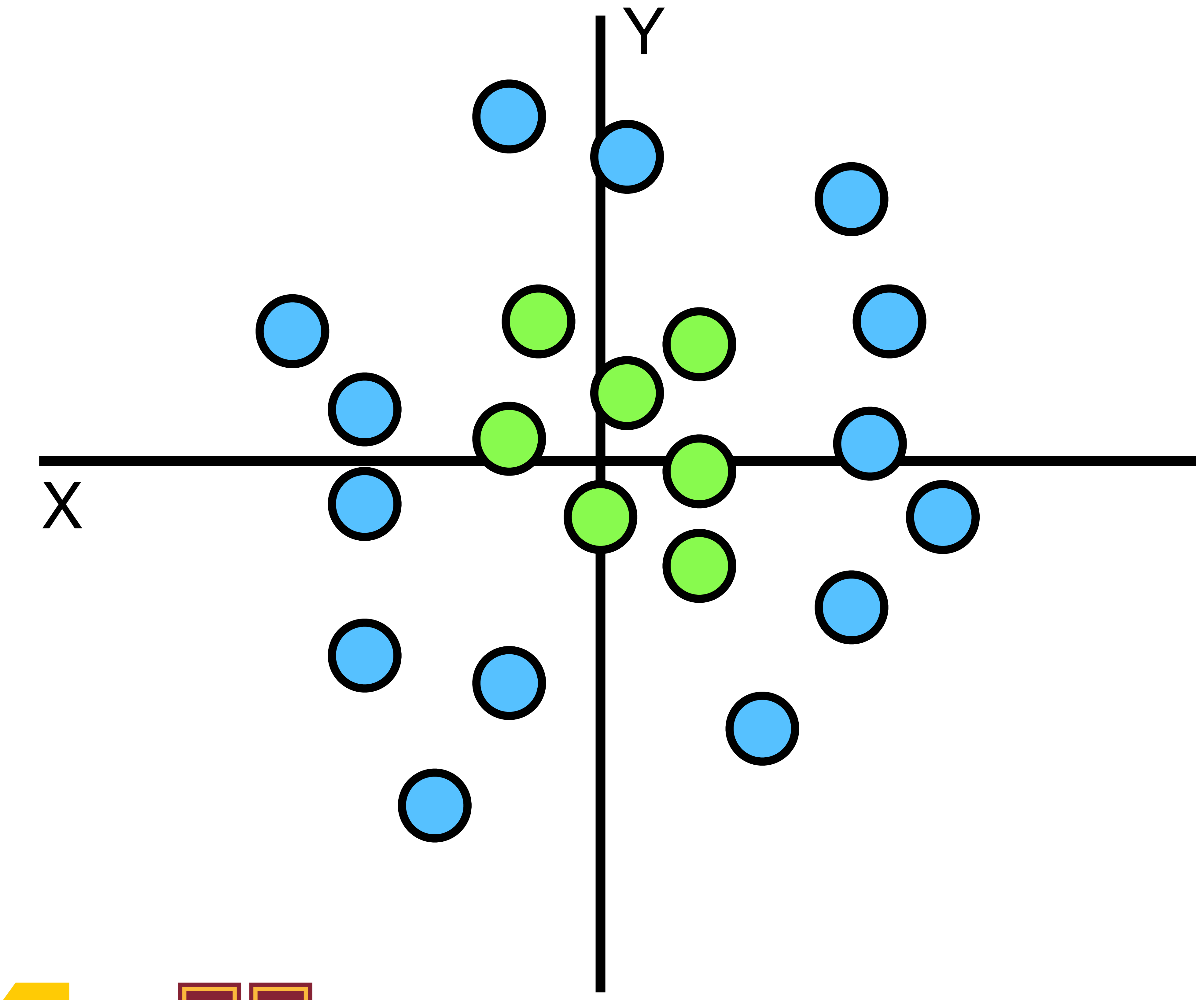
$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$

```
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

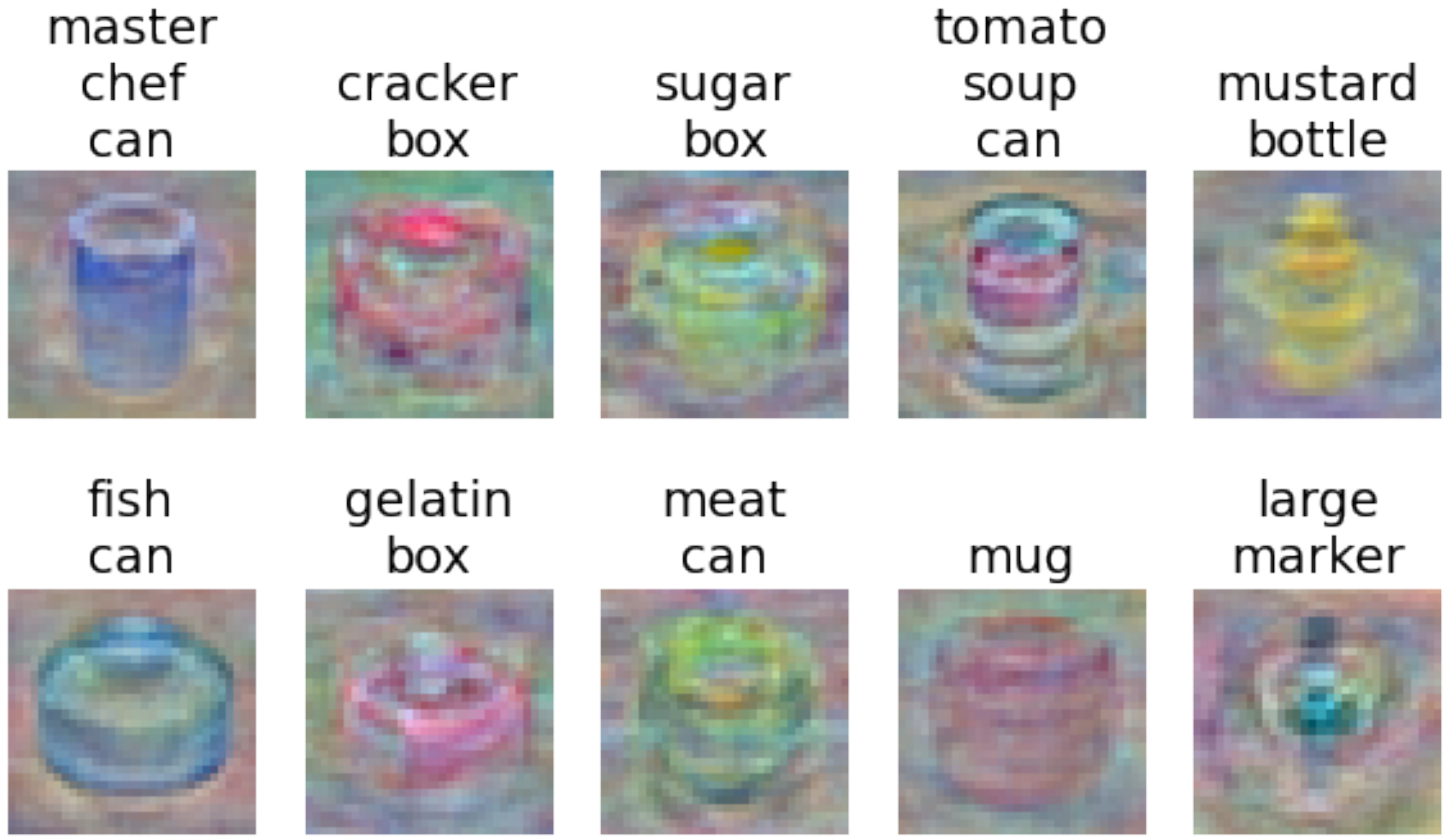
Geometric Viewpoint



Visual Viewpoint

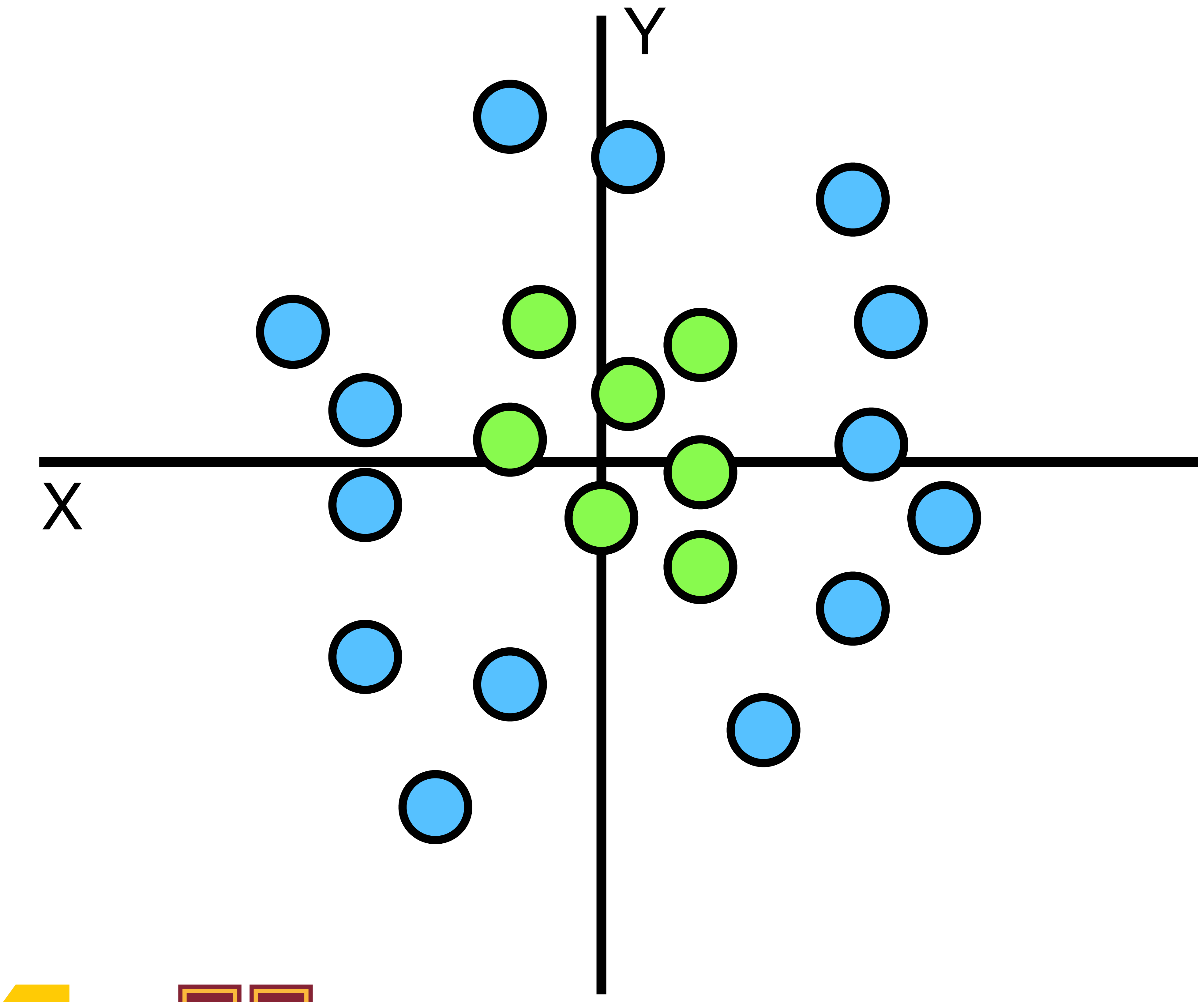
One template per class:

Can't recognize different modes of a class



One solution: Feature Transforms

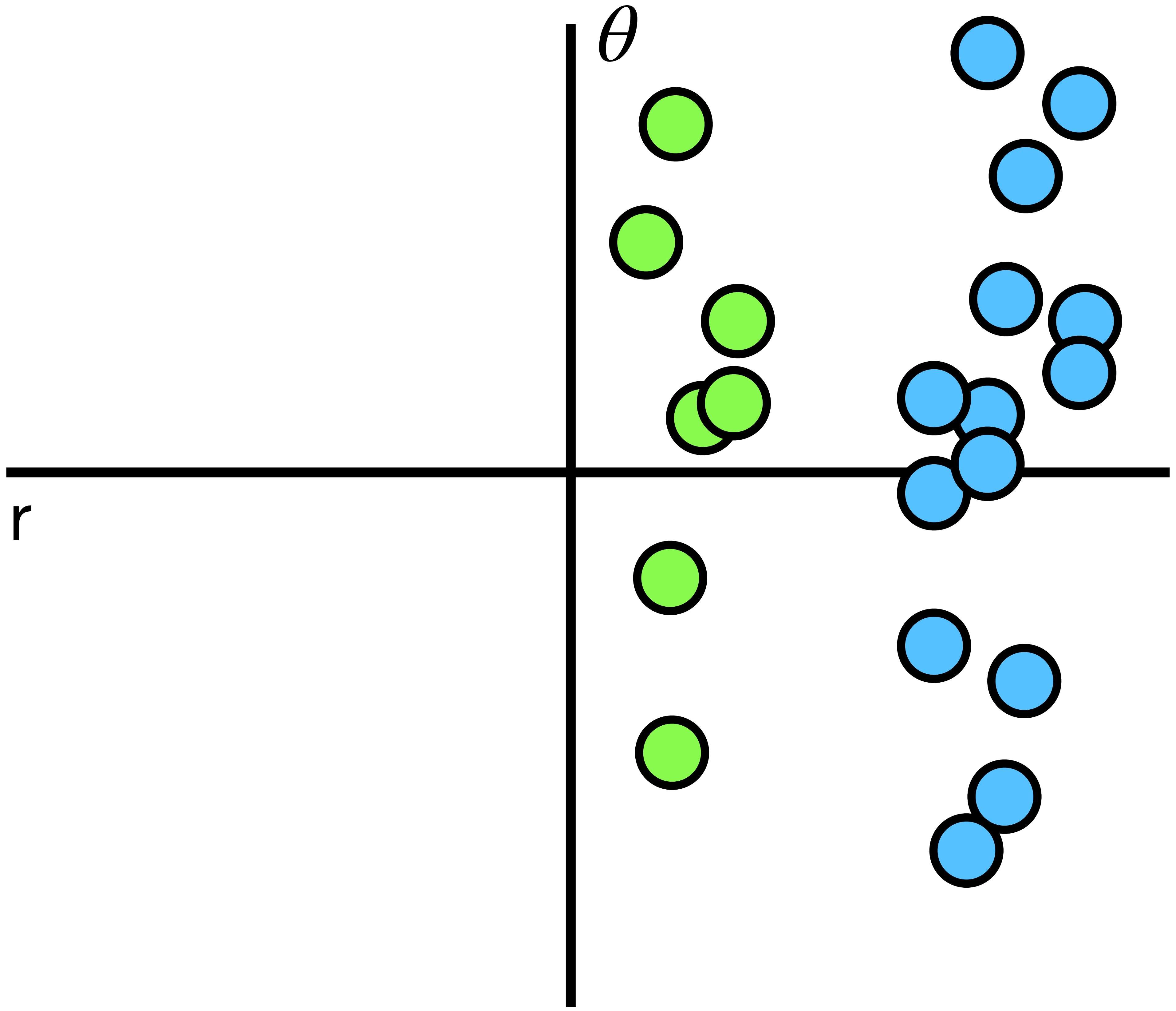
Original space



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

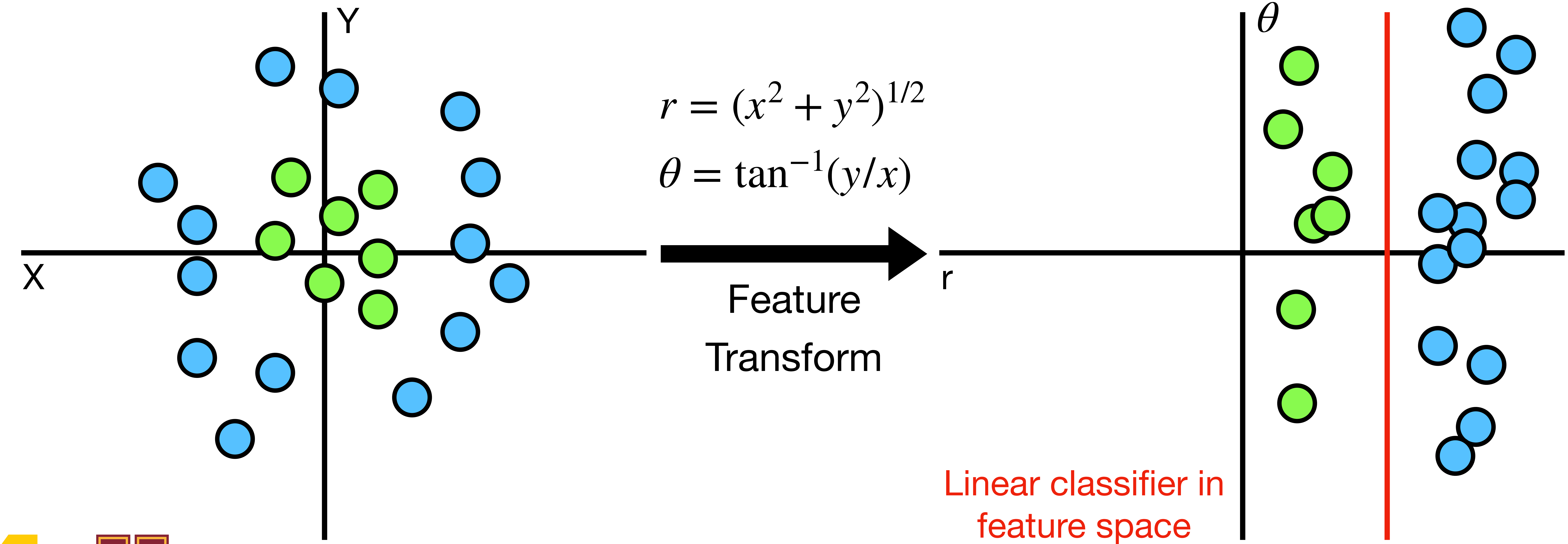
Feature Transform

Feature space



One solution: Feature Transforms

Original space | **Feature space**

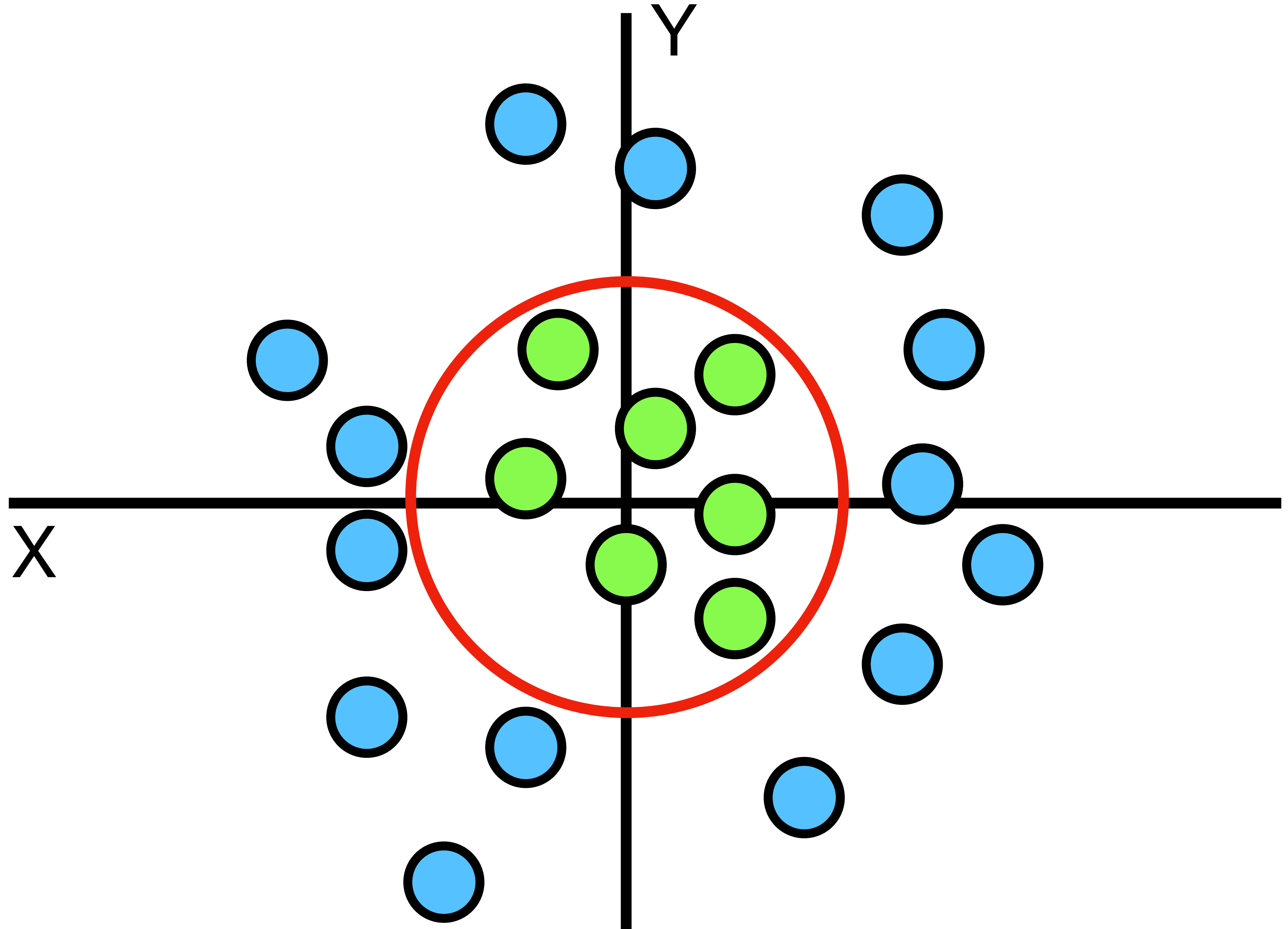


$$r = (x^2 + y^2)^{1/2}$$
$$\theta = \tan^{-1}(y/x)$$

Feature Transform

One solution: Feature Transforms

Original space

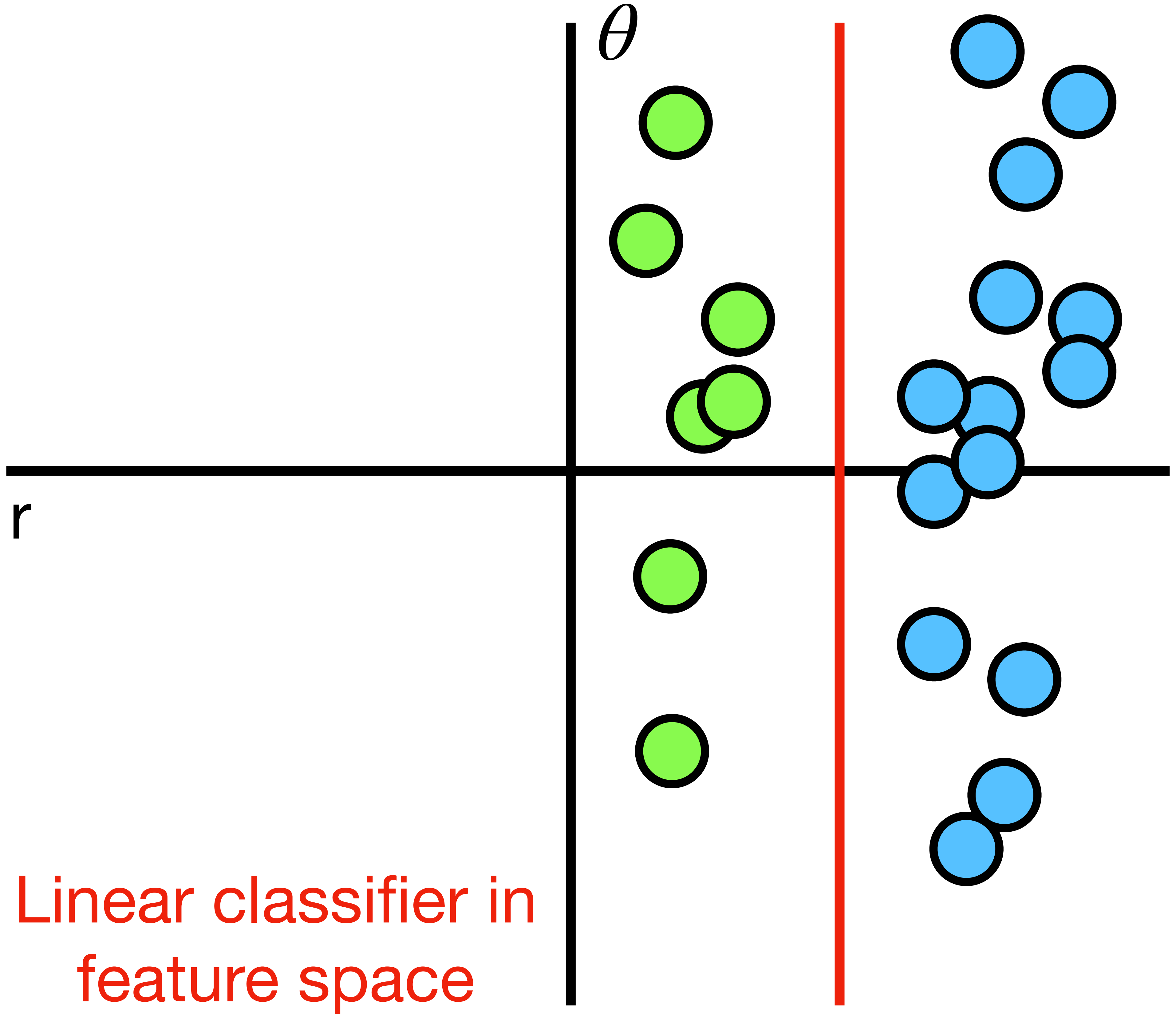


$$r = (x^2 + y^2)^{1/2}$$

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

Feature Transform

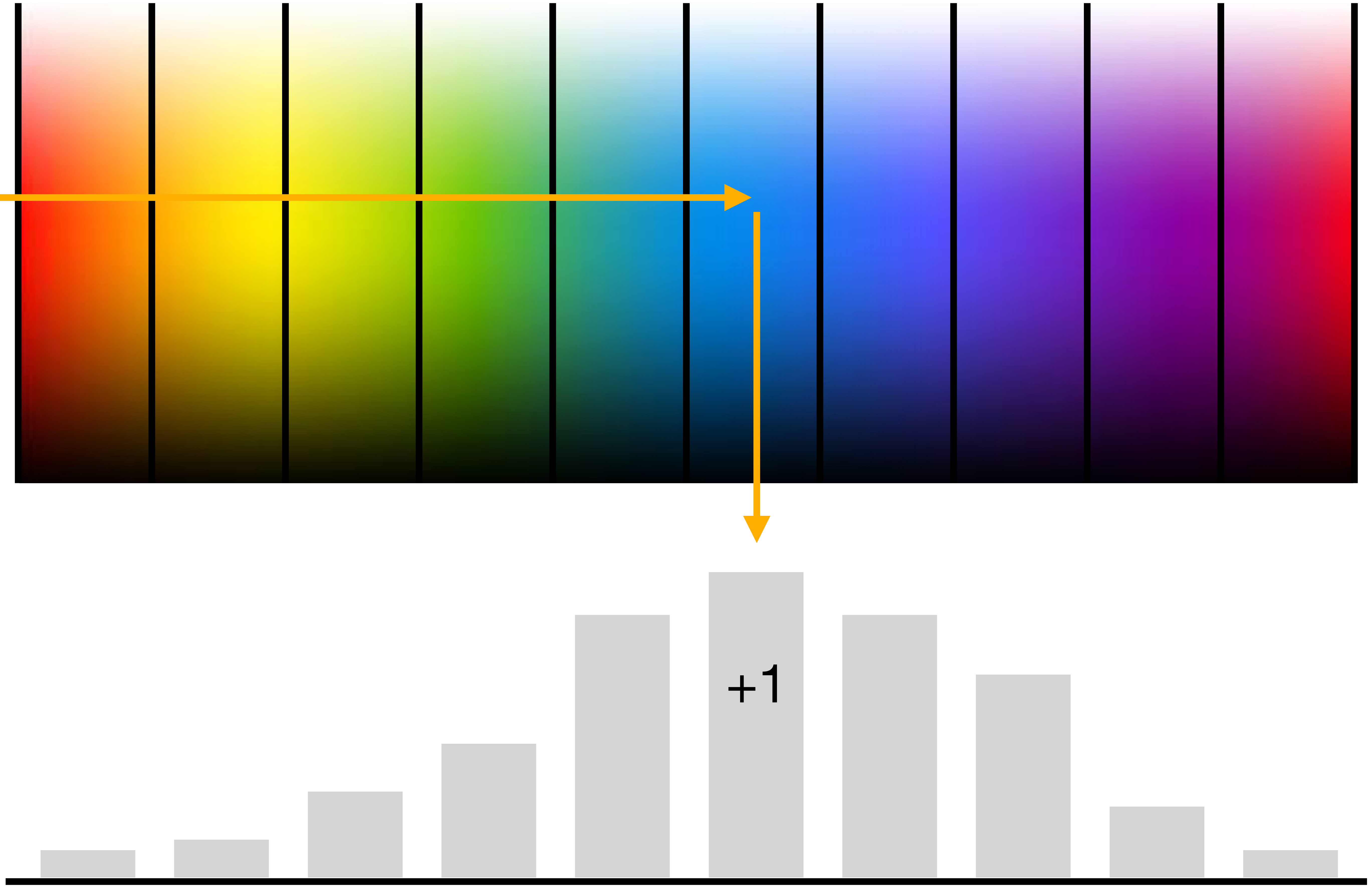
Feature space



Nonlinear classifier in original space!

Linear classifier in feature space

Image Features: Color Histogram



Ignores texture, spatial positions

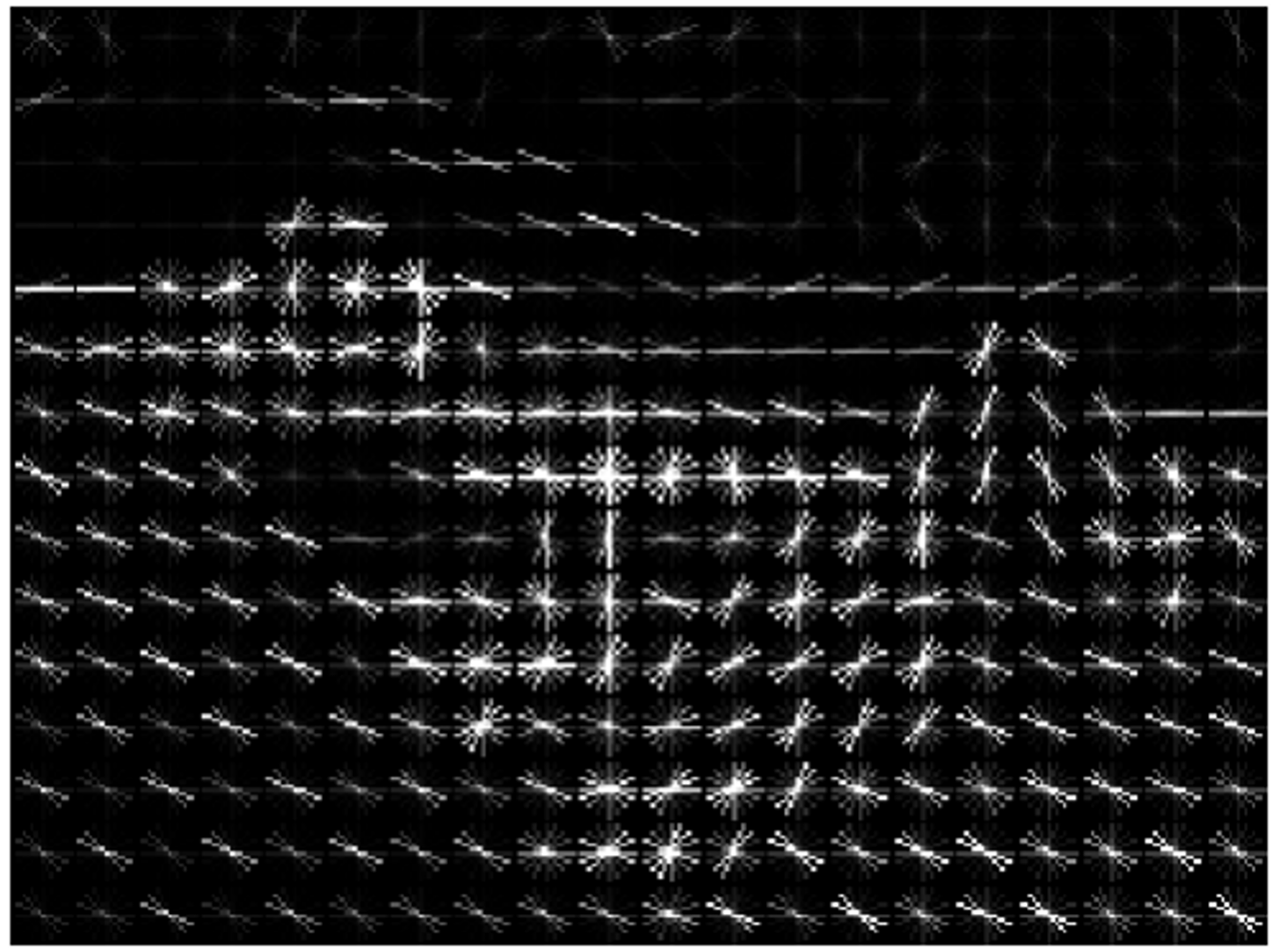
Frog image is in the public domain

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

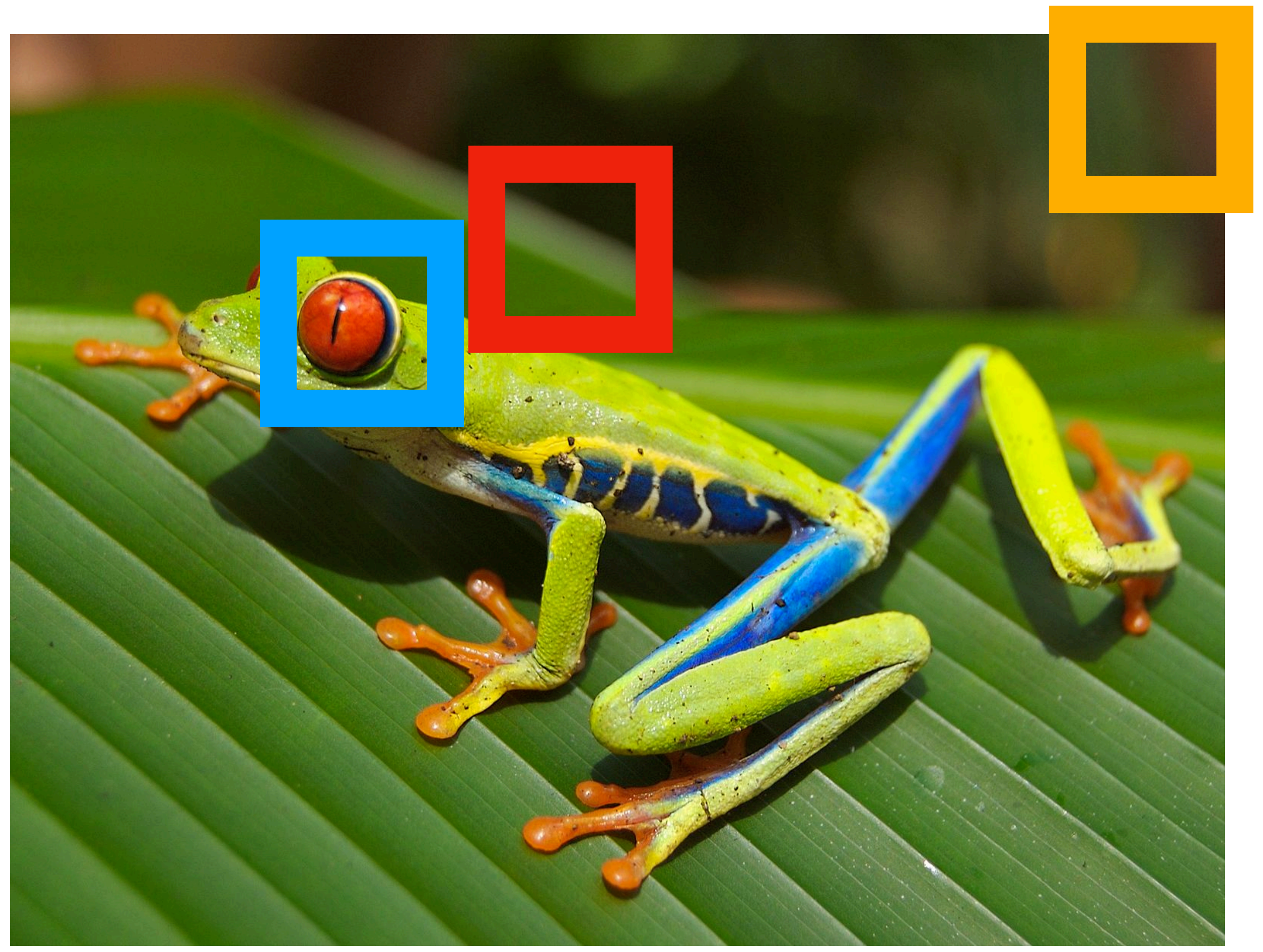
Example: 320x240 image gets divided into 40x30 bins;

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

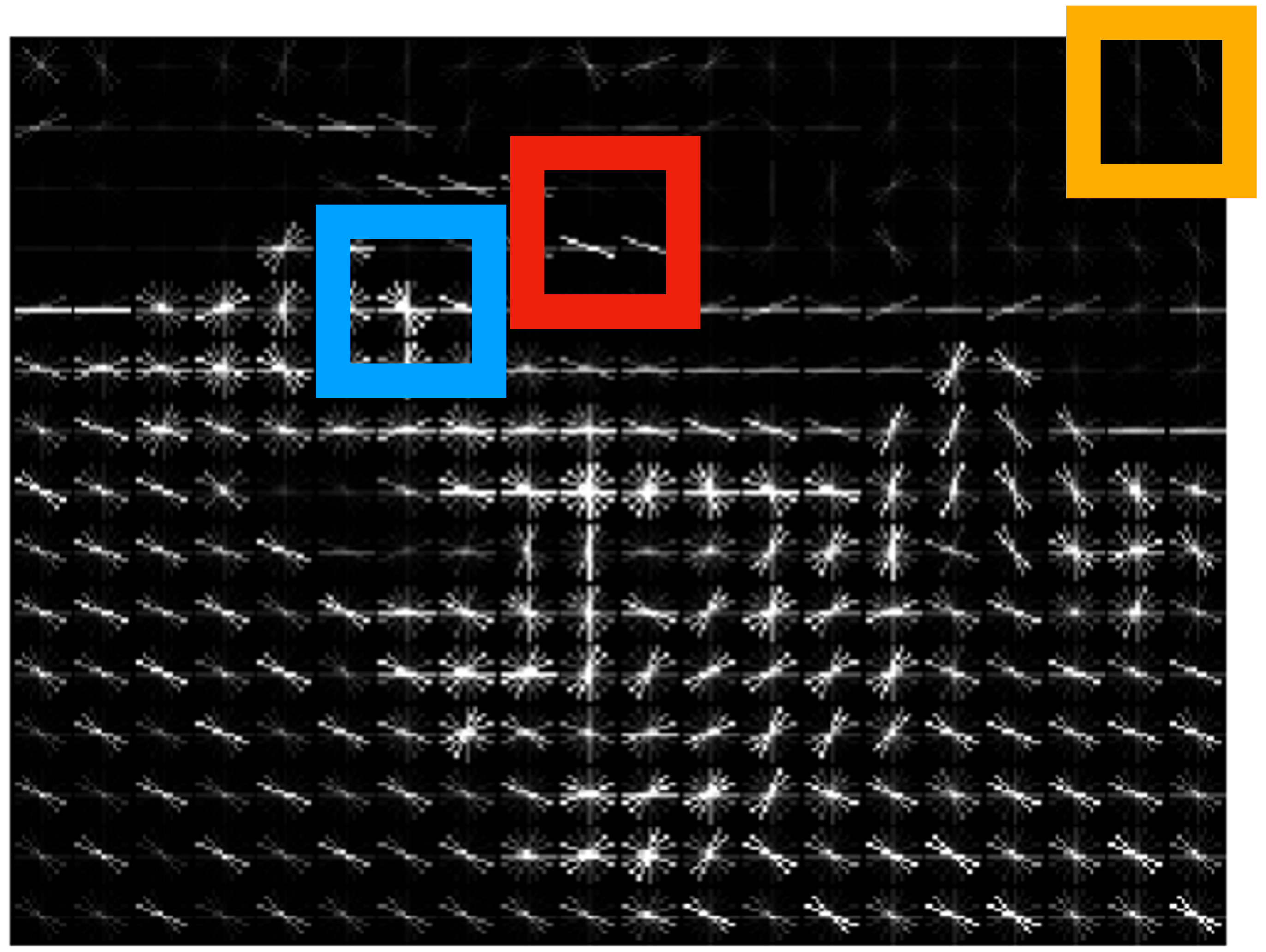
Image Features: Histogram of Oriented Gradients (HoG)



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;

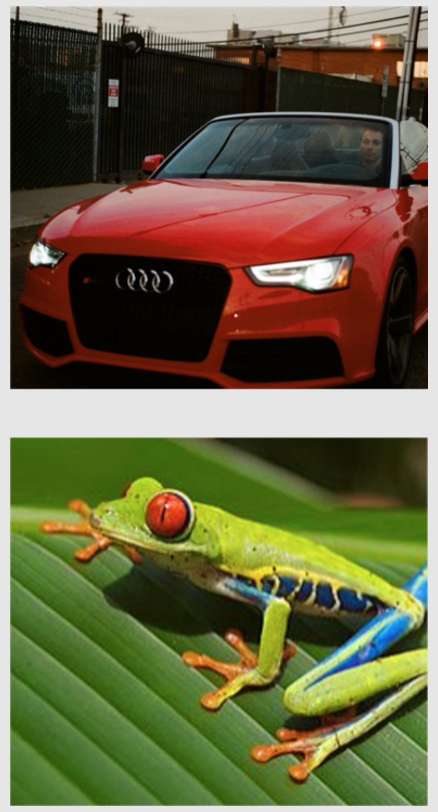
9 directions per bin;

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

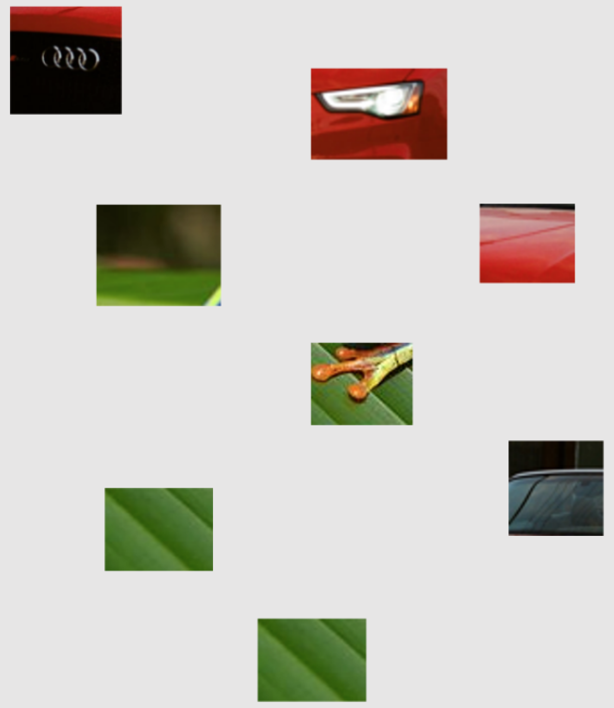
Lowe, "Object recognition from local scale-invariant features," ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

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

Step 1: Build codebook

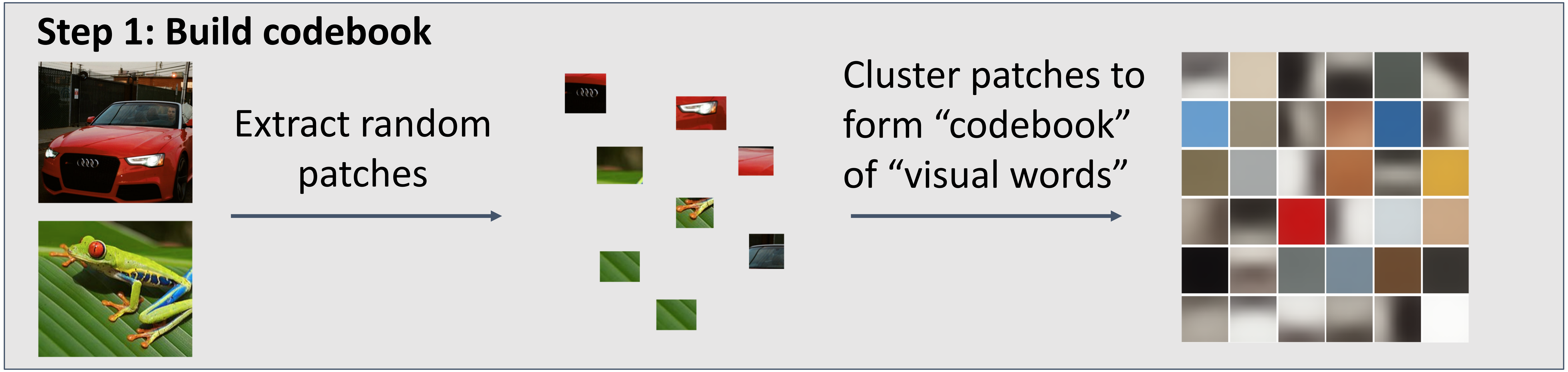


Extract random patches



Cluster patches to form "codebook" of "visual words"

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



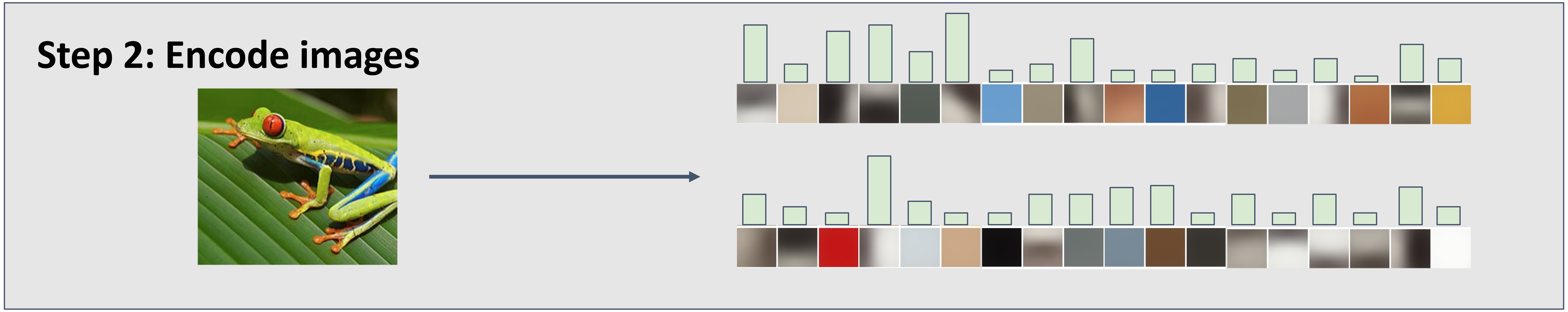
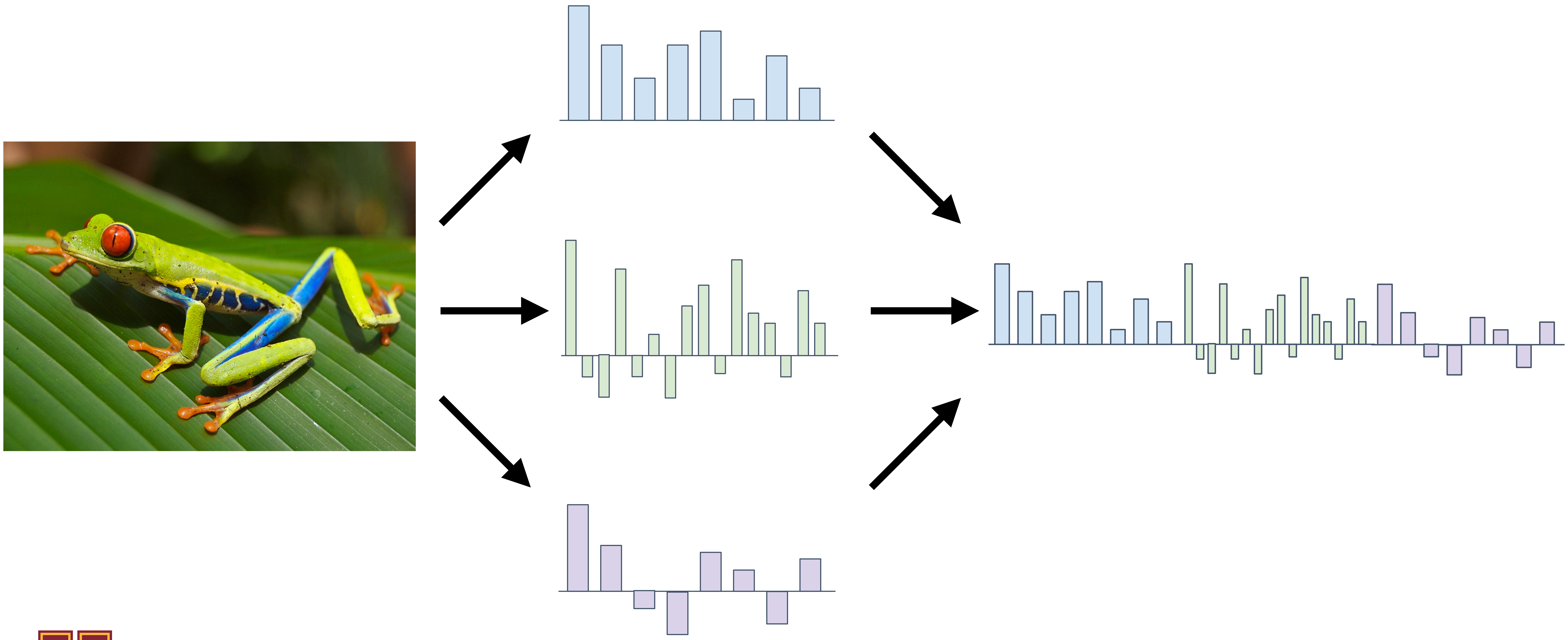


Image Features



Example: Winner of 2011 ImageNet Challenge

Low-level feature extraction ≈ 10k patches per image

$\left.\begin{array}{l}\bullet\ \text{SIFT: 128-dims}\\ \bullet\ \text{Color: 96-dim}\end{array}\right\}$ Reduced to 64-dim with PCA

FV extraction and compression:

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

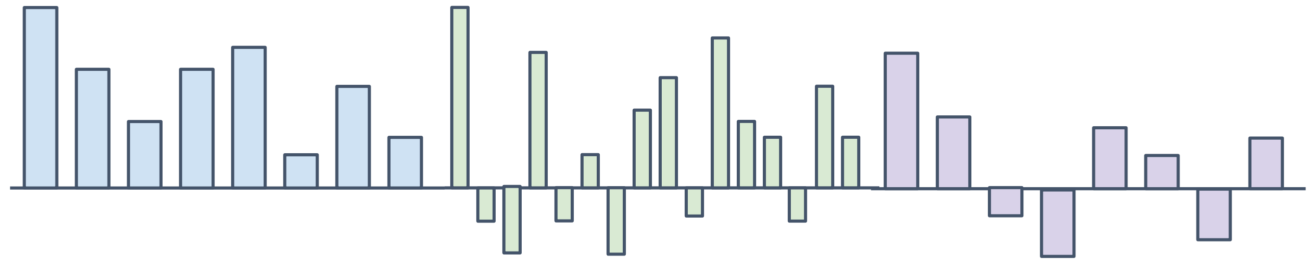
One-vs-all SVM learning with SGD

Late fusion of SIFT and color systems

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

Image Features vs Neural Networks

Feature Extraction



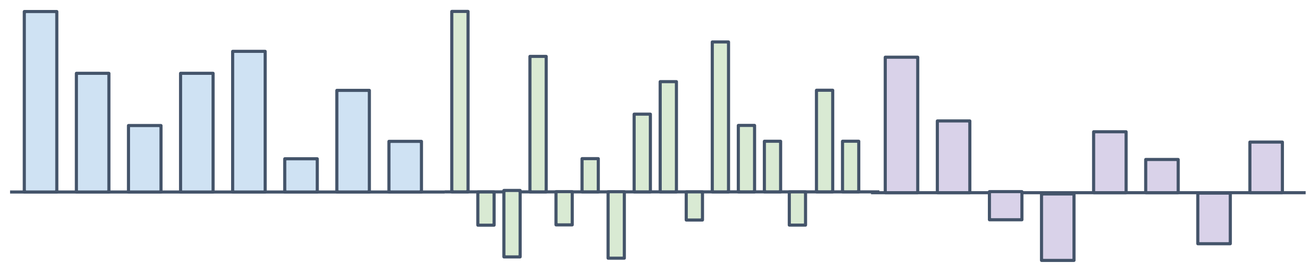
f

training

10 numbers giving scores for classes

Image Features vs Neural Networks

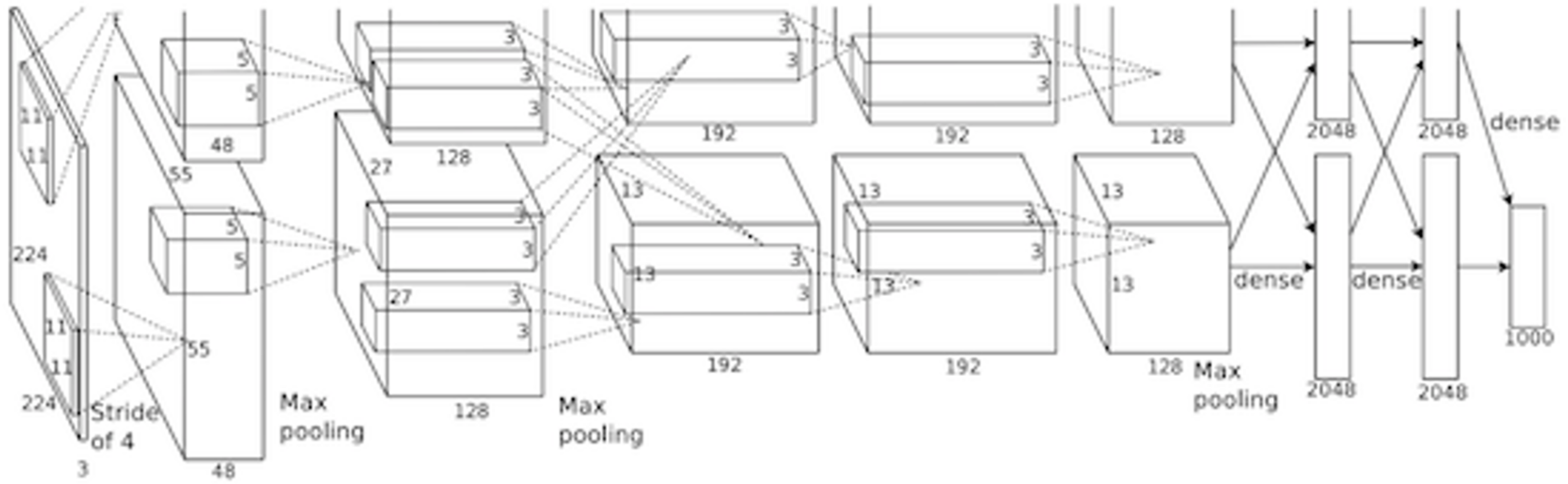
Feature Extraction



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

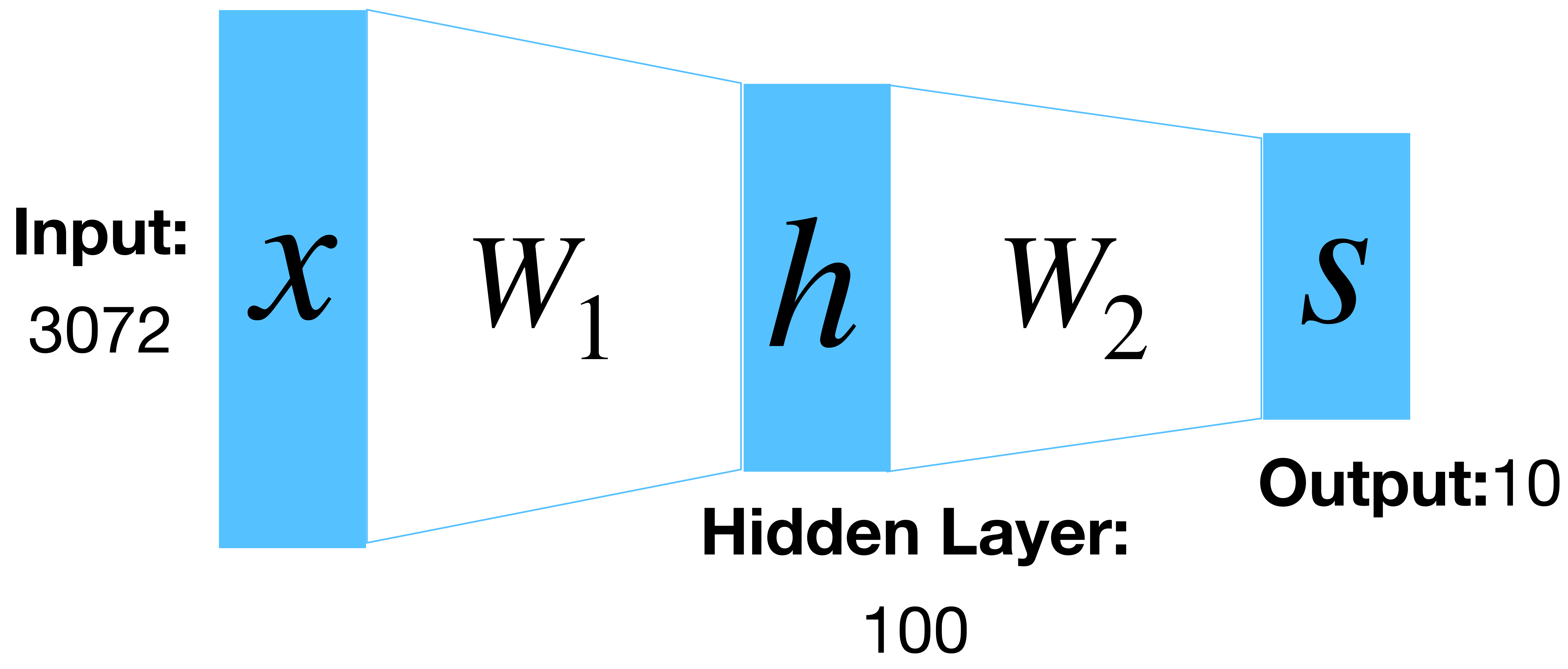
Now: Two-Layer Neural Network: $f(x) = W_2 \max(0, W_1 x + b_1) + b_2$
Learnable parameters: $W_1 \in \mathbb{R}^{H \times D}, b_1 \in \mathbb{R}^H, W_2 \in \mathbb{R}^{C \times H}, b_2 \in \mathbb{R}^C$

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



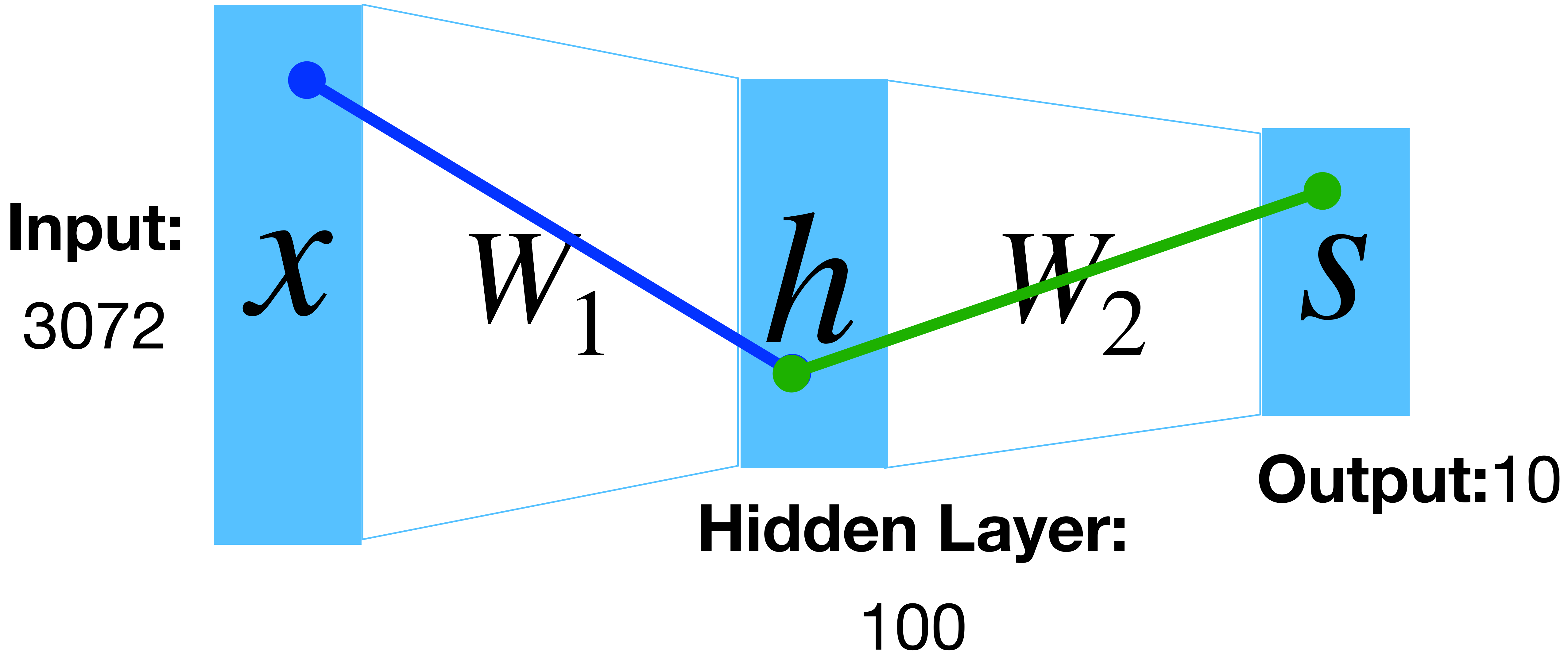
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

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

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

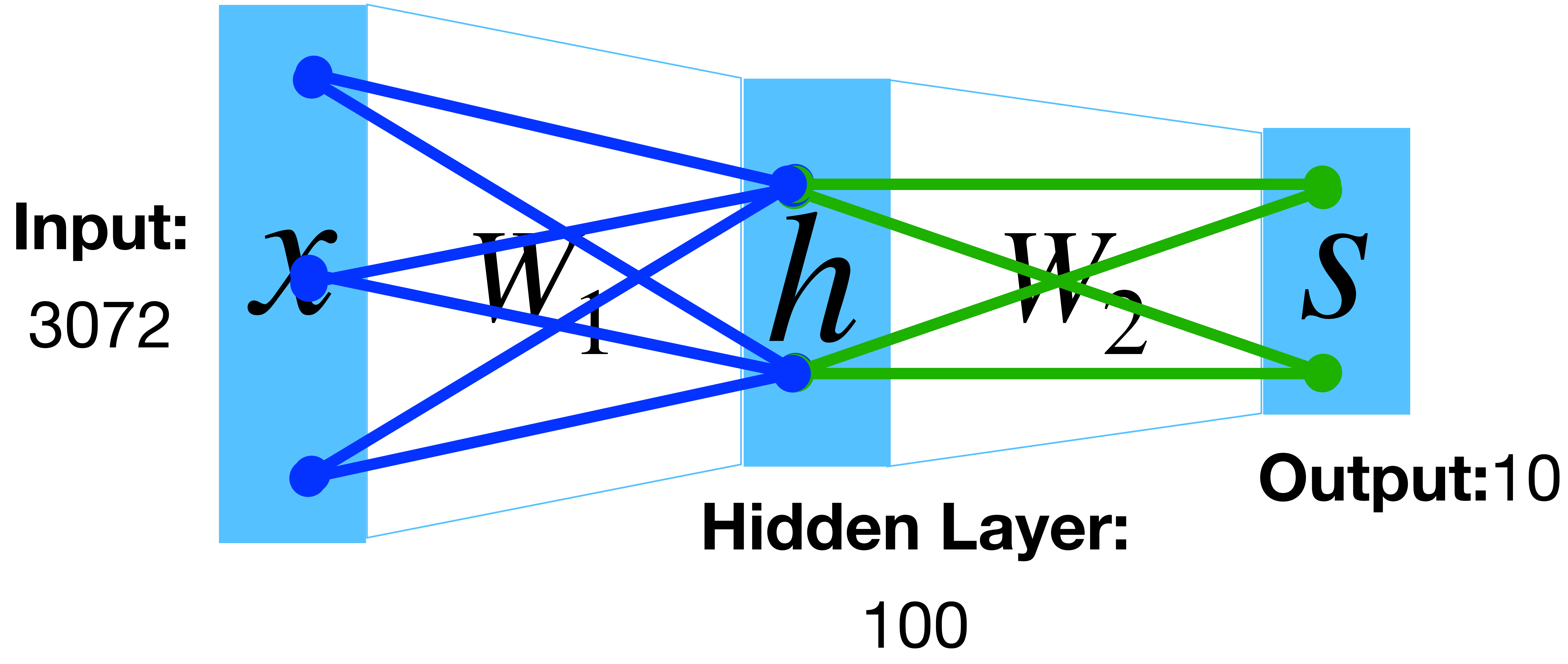
Neural Networks

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



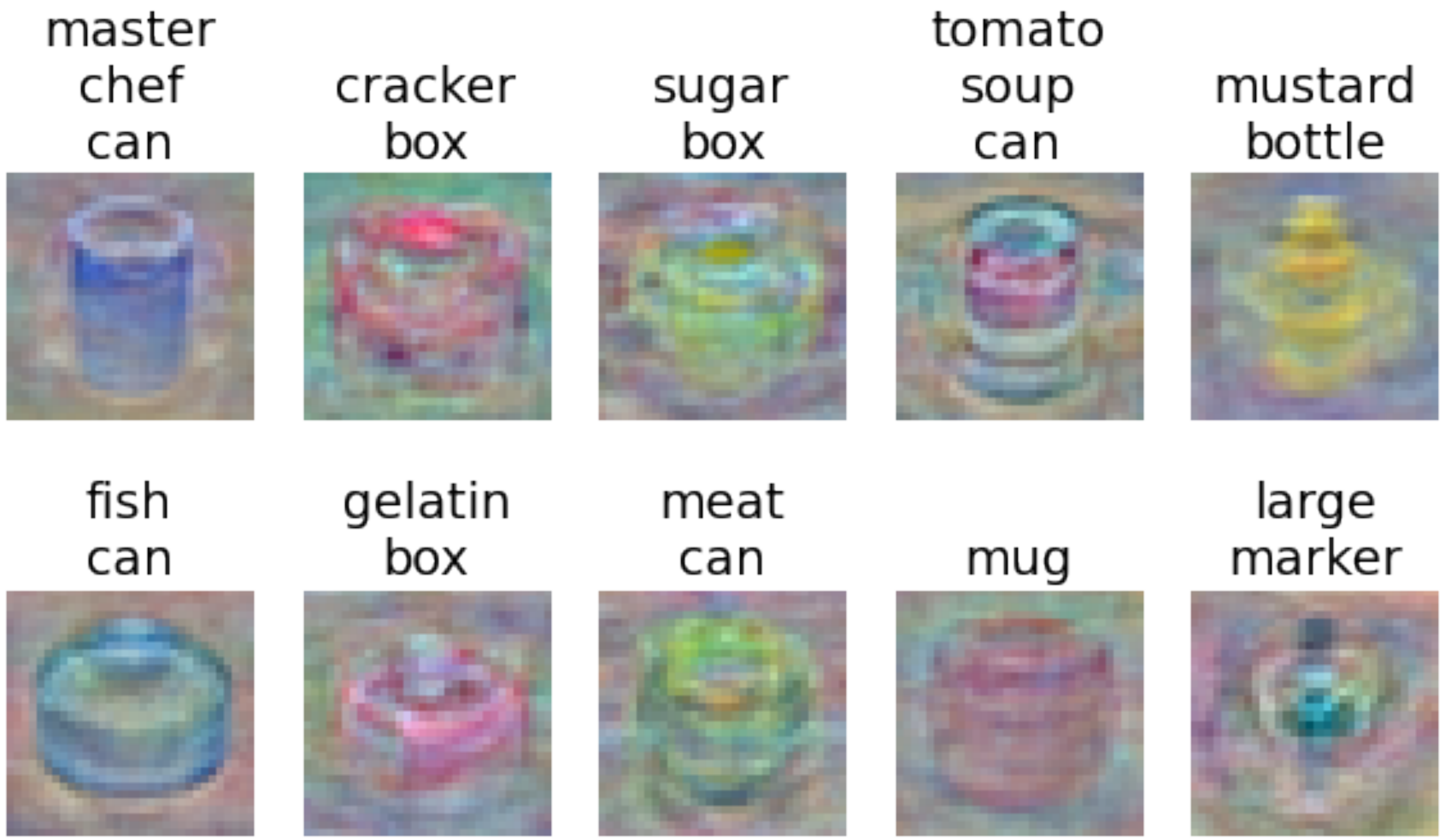
Element (i, j) of W_2 gives the effect on s_i from h_j

All elements of h affect all elements of s

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

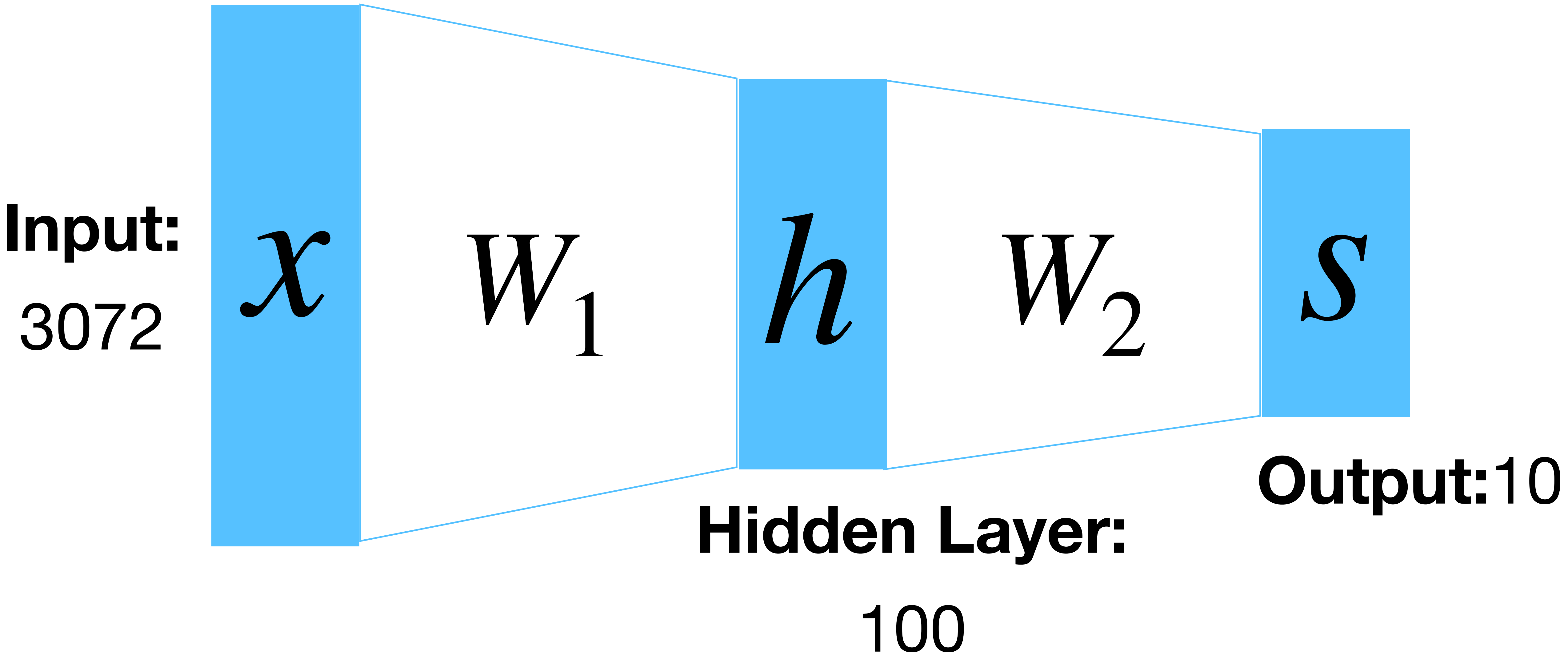
Neural Networks

Linear classifier: One template per class



Before: Linear score function

Now: Two-Layer Neural Network:



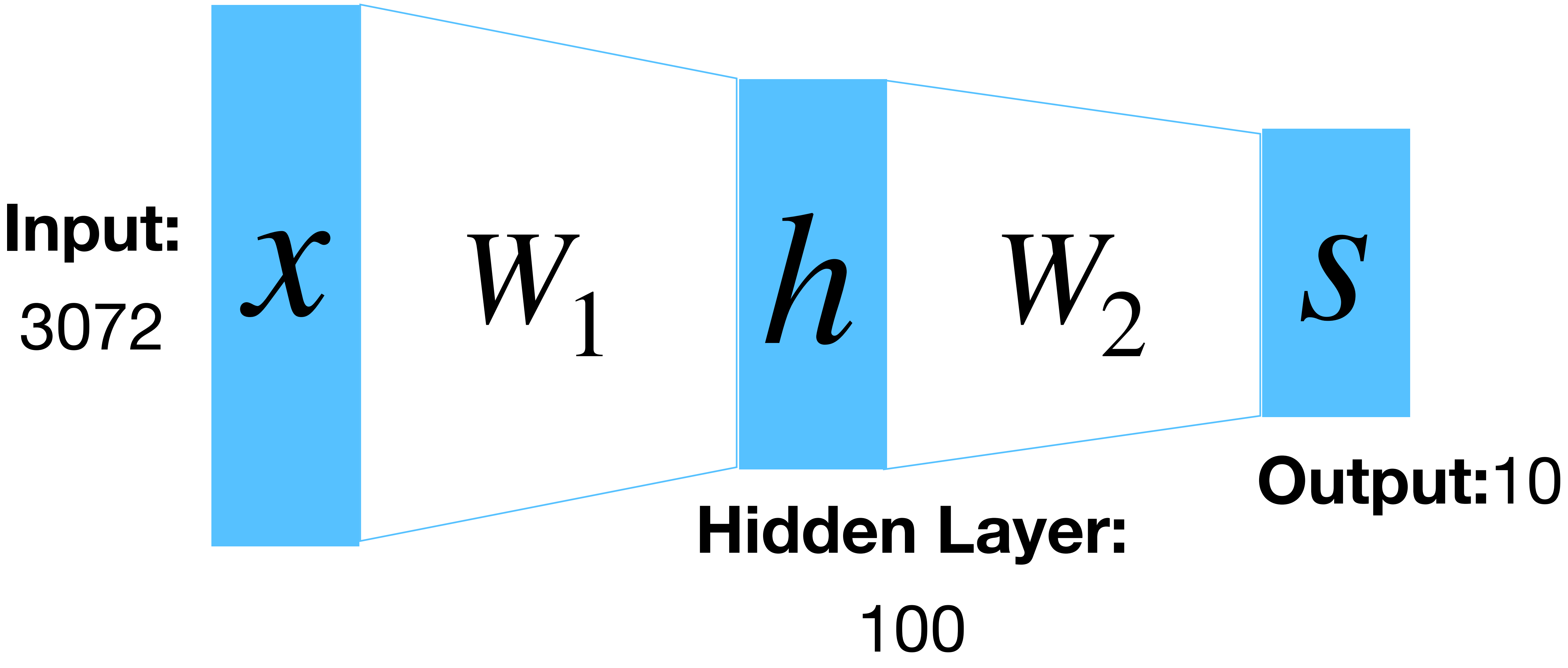
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

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

Before: Linear score function

Now: Two-Layer Neural Network:



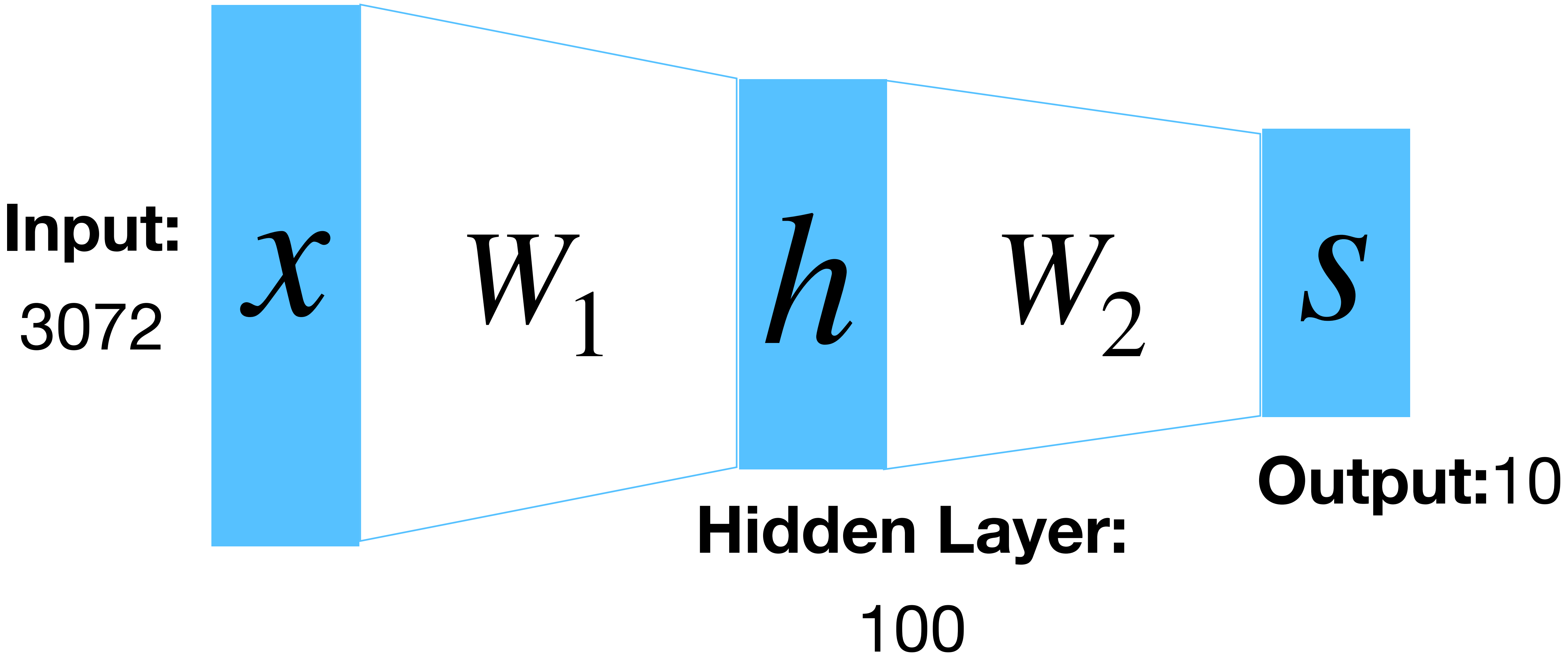
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

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

Before: Linear score function

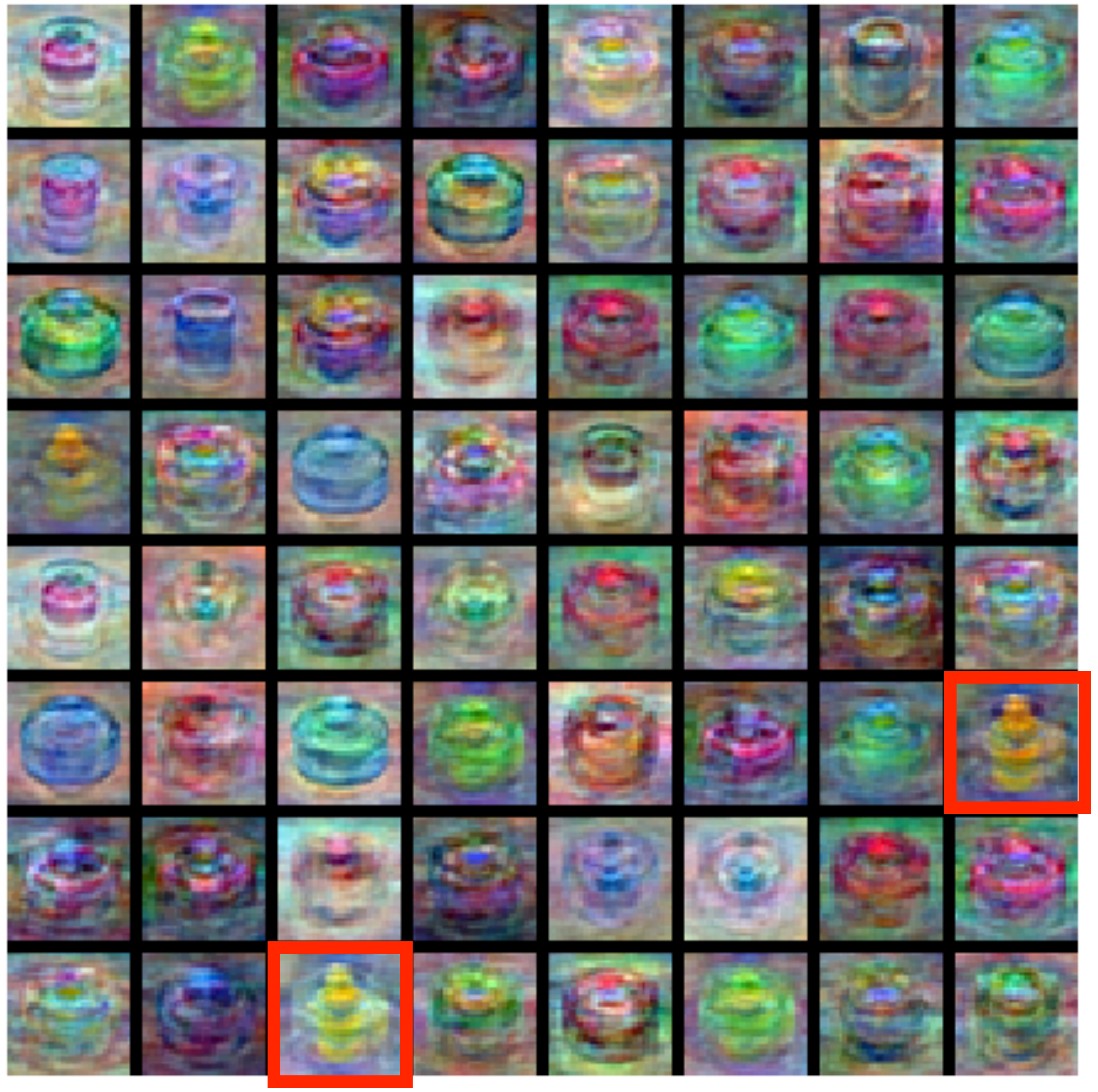
Now: Two-Layer Neural Network:



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

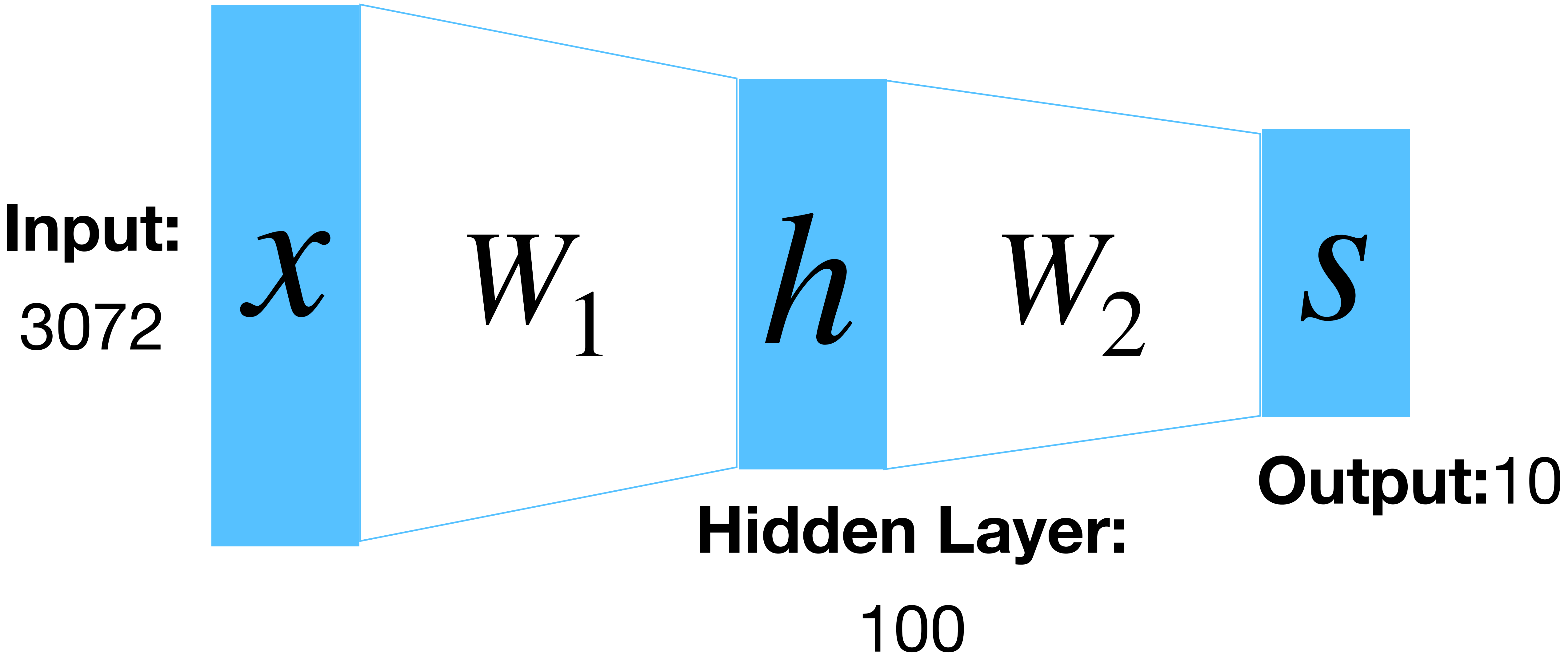
Neural Networks

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



Before: Linear score function

Now: Two-Layer Neural Network:



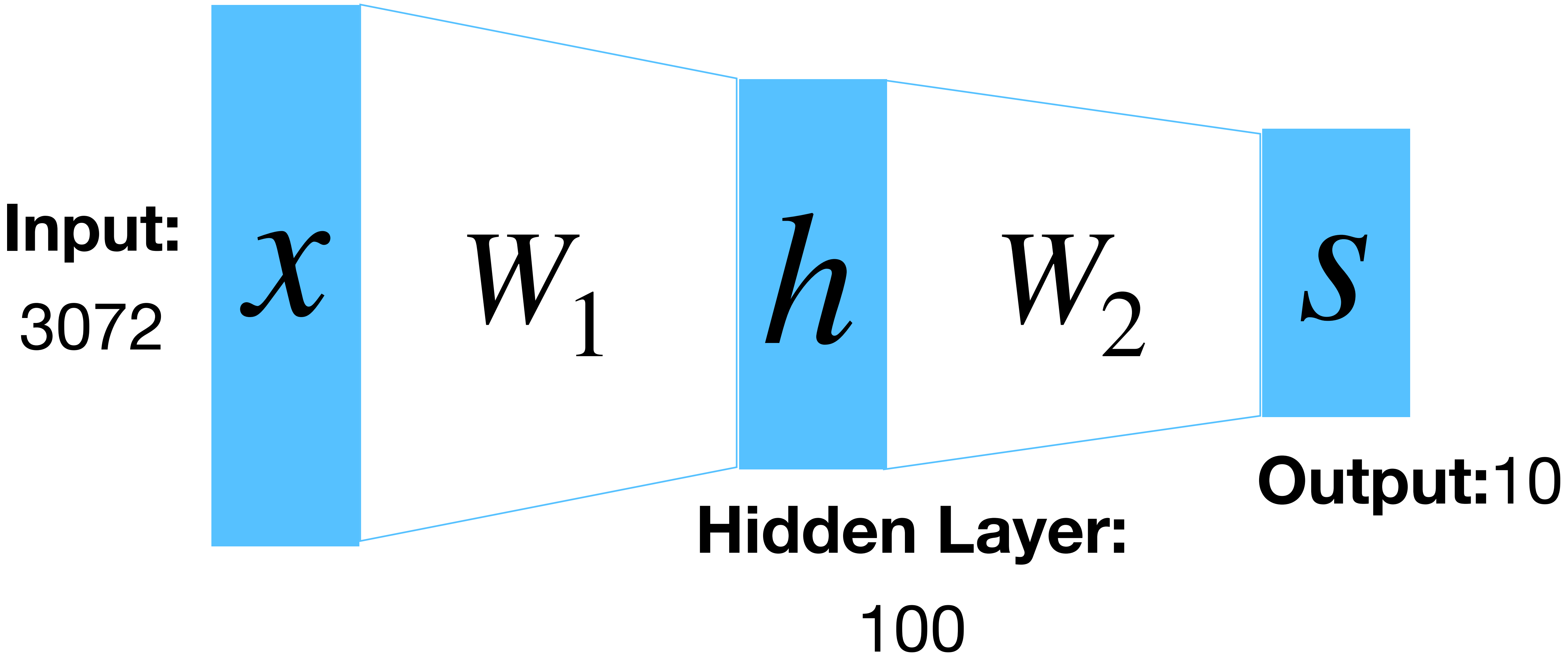
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

"Distributed representation": Most templates not interpretable!

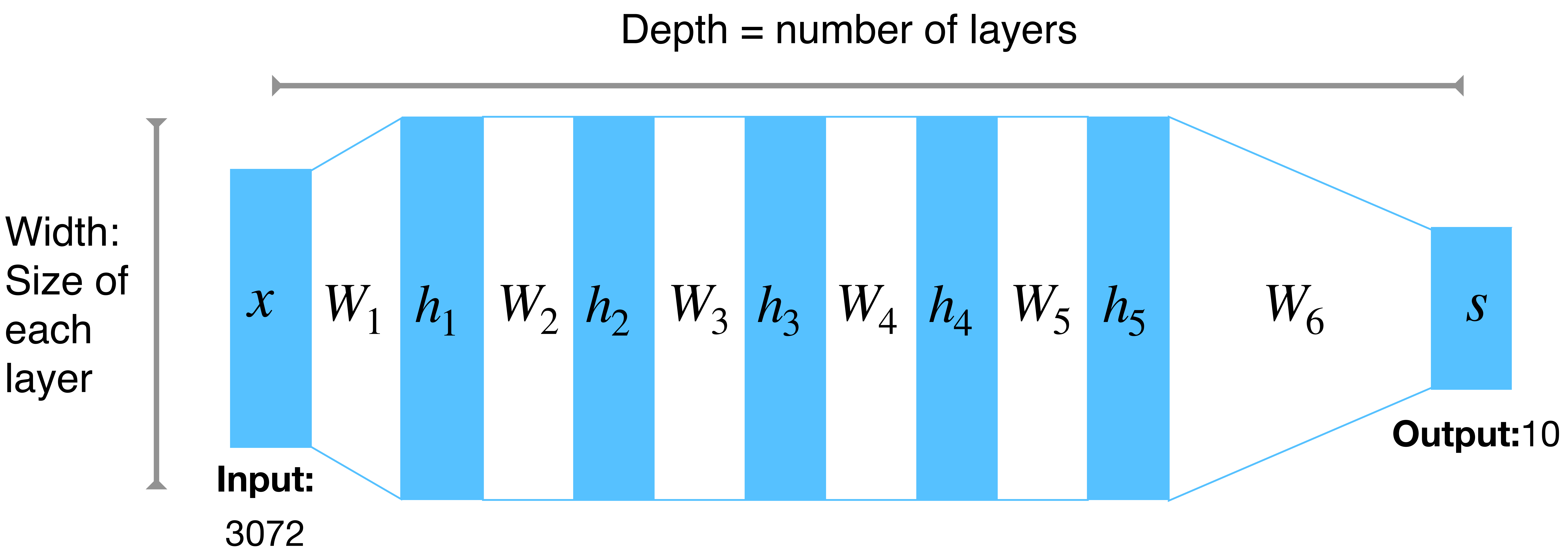
Before: Linear score function

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



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

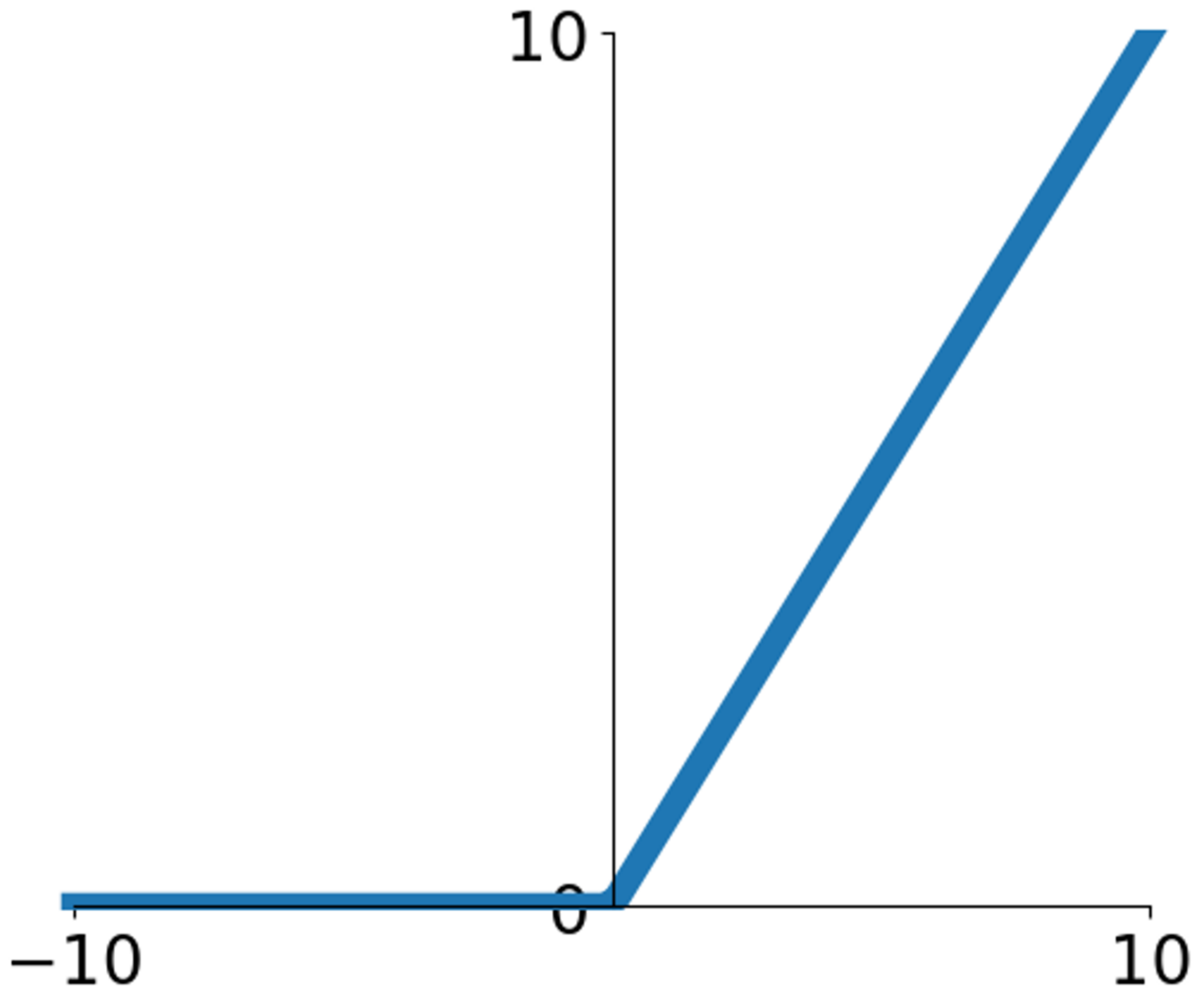
Activation Functions

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

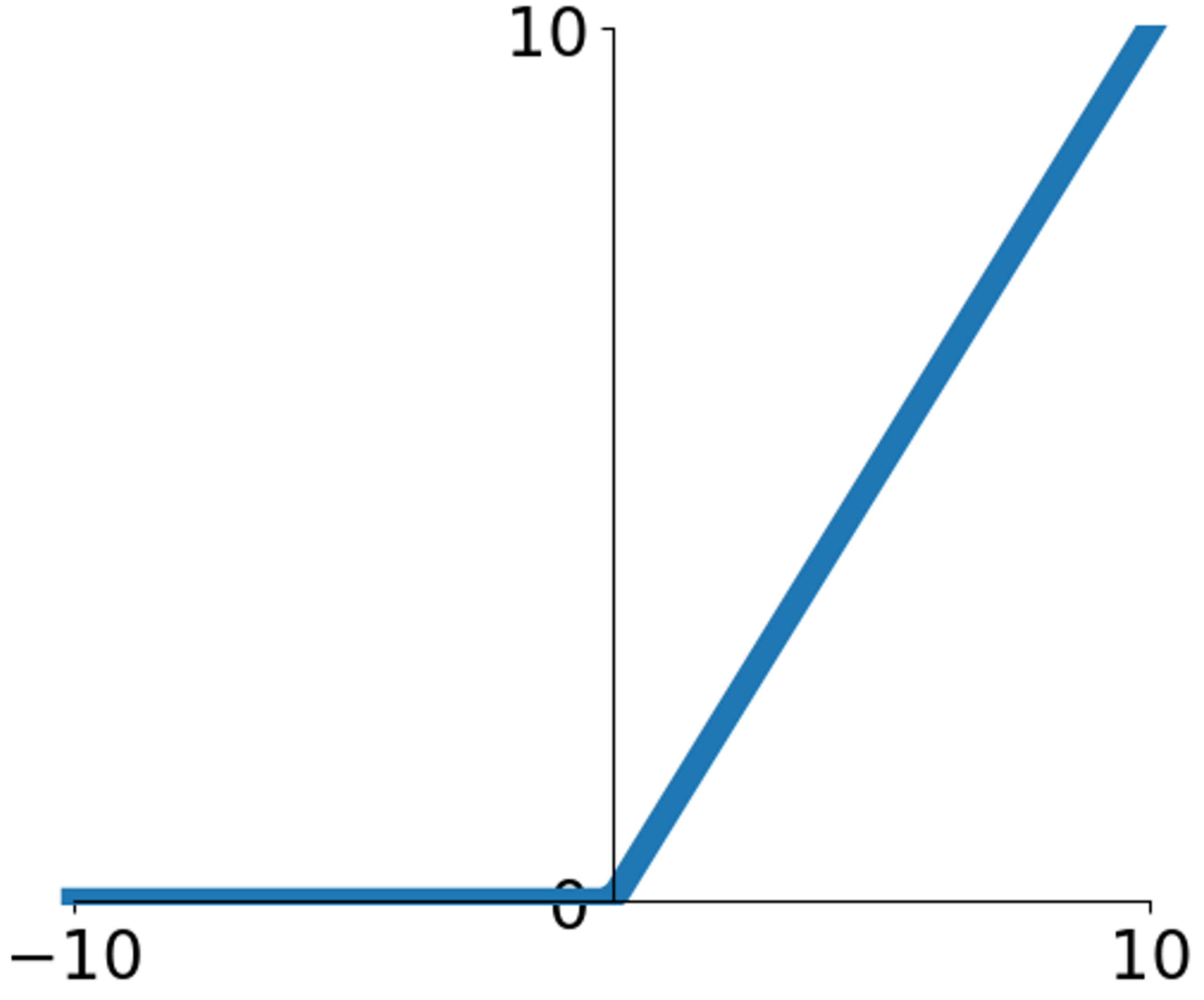


Activation Functions

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?

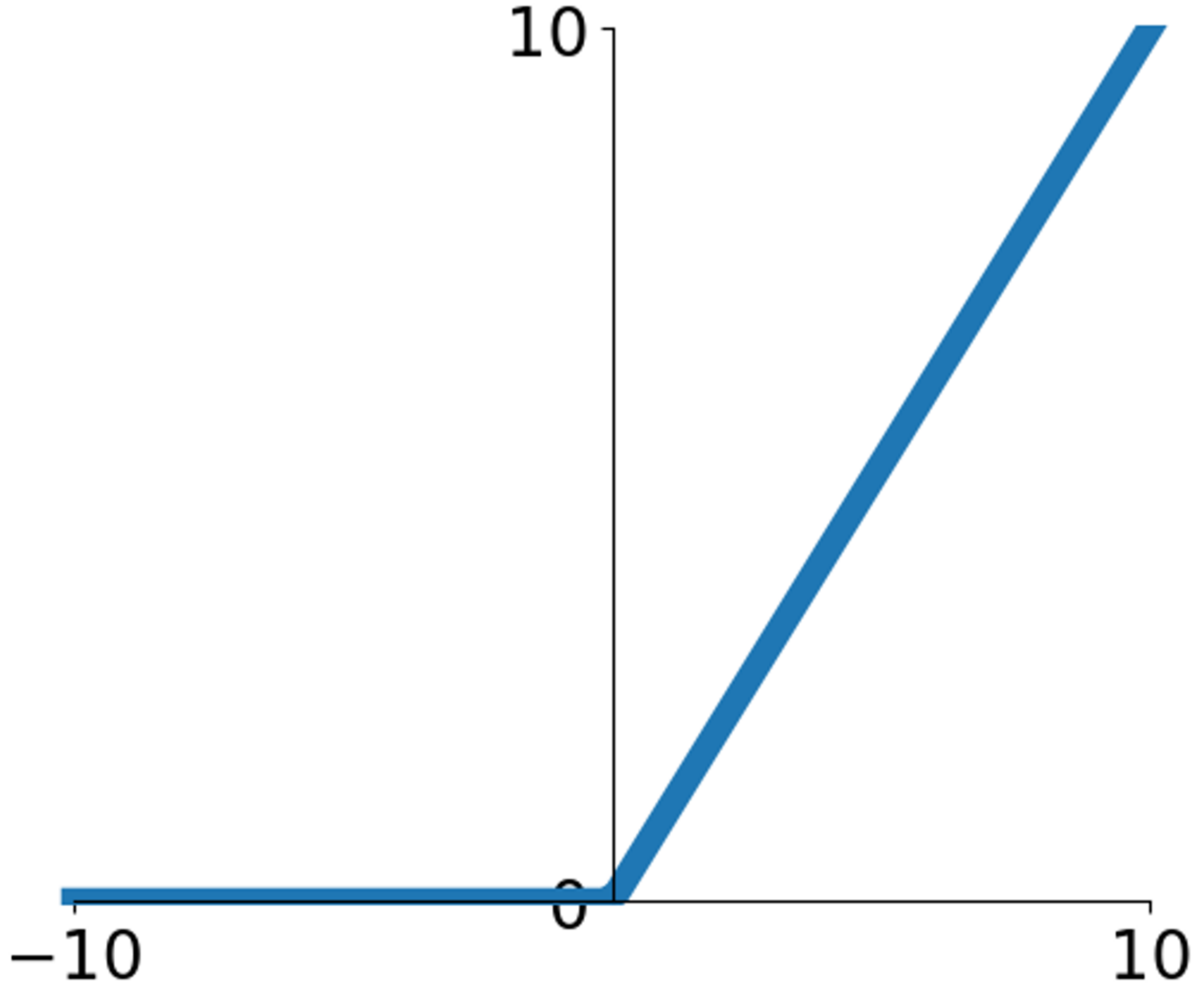
$$f(x) = W_2(W_1 x + b_1) + b_2$$

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?

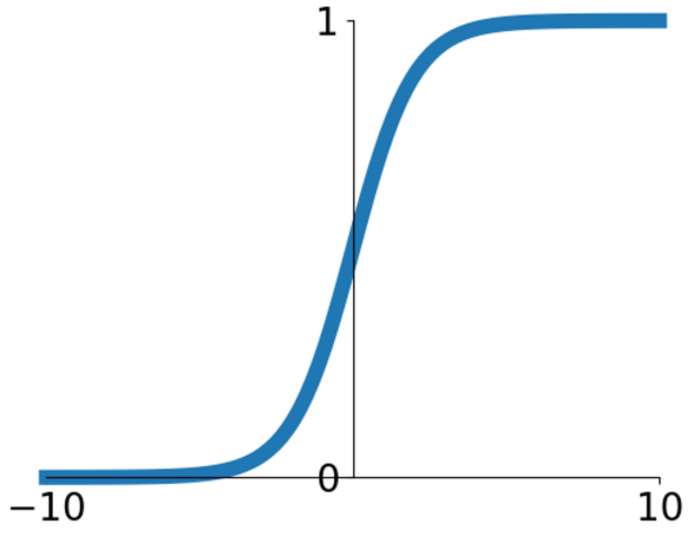
$$\begin{aligned} f(x) &= W_2(W_1 x + b_1) + b_2 \\ &= (W_1 W_2)x + (W_2 b_1 + b_2) \end{aligned}$$

A: We end up with a linear classifier

Activation Functions

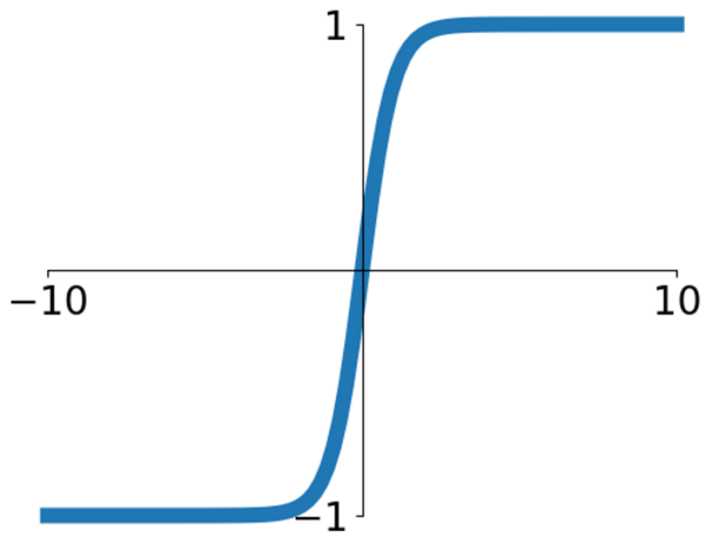
Sigmoid

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



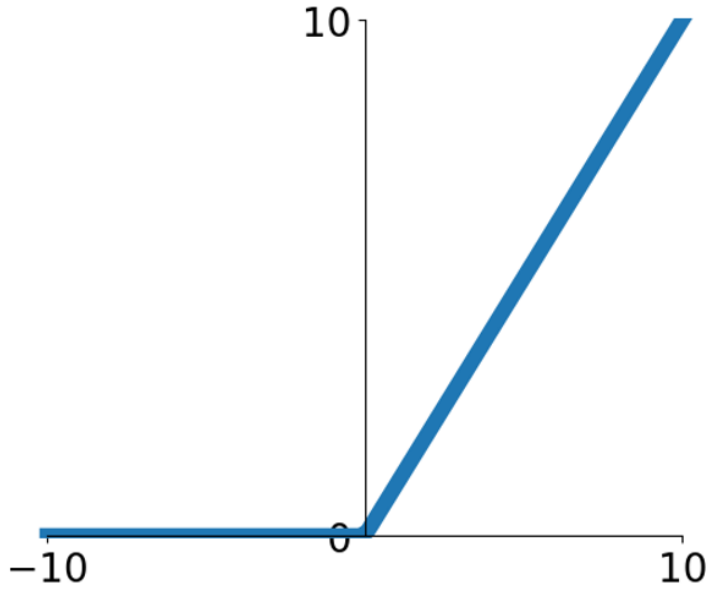
tanh

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



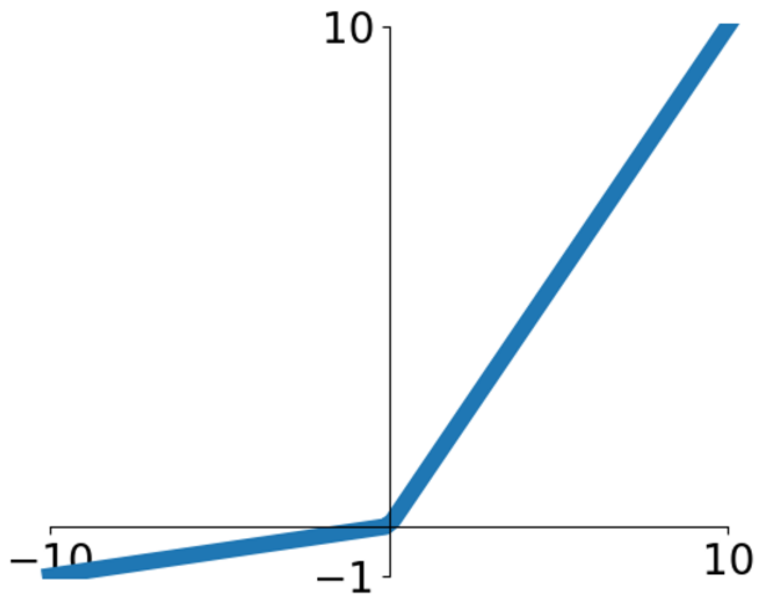
ReLU

$$\max(0,x)$$



Leaky ReLU

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

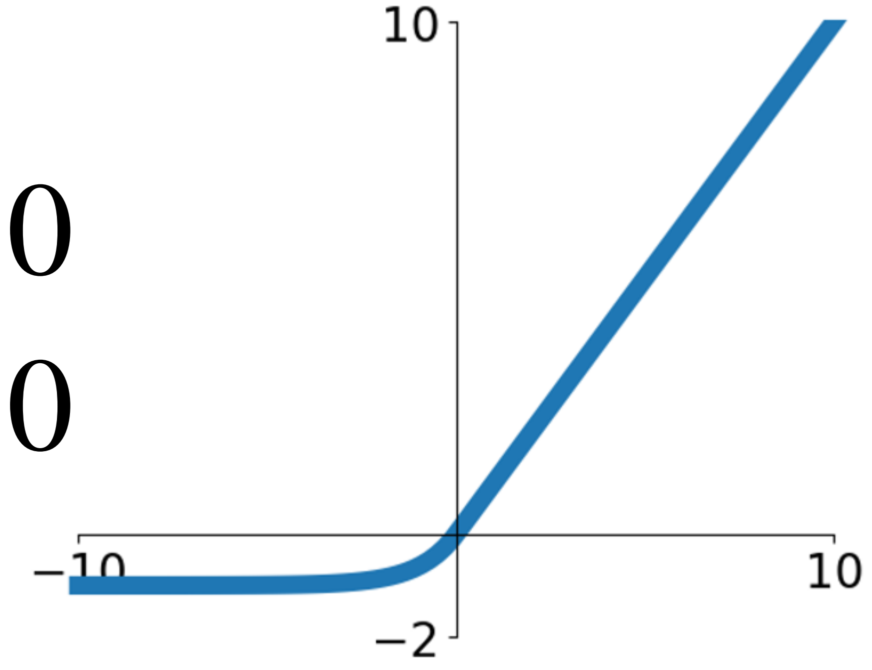


Softplus

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

ELU

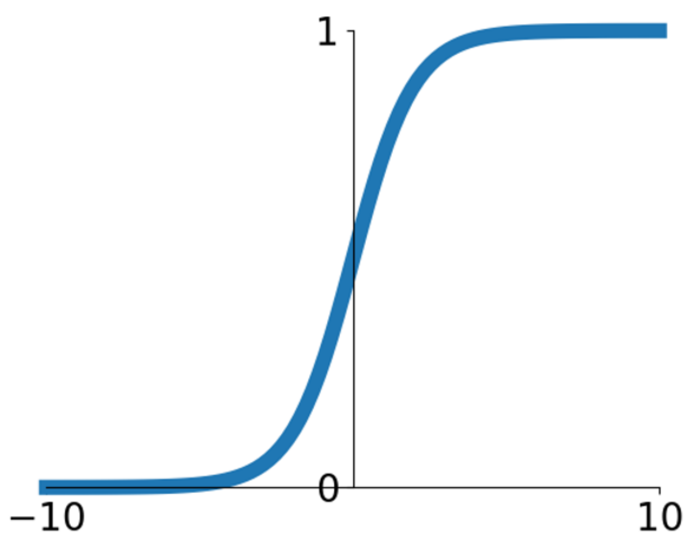
$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) -), & x \leq 0 \end{cases}$$



Activation Functions

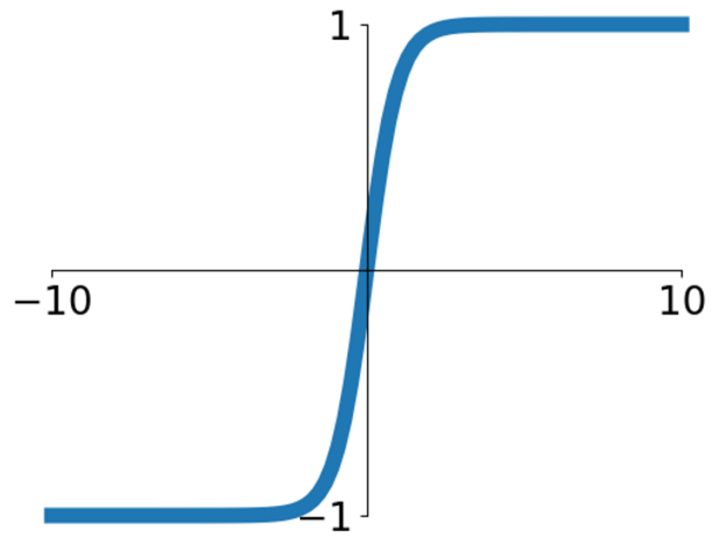
Sigmoid

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



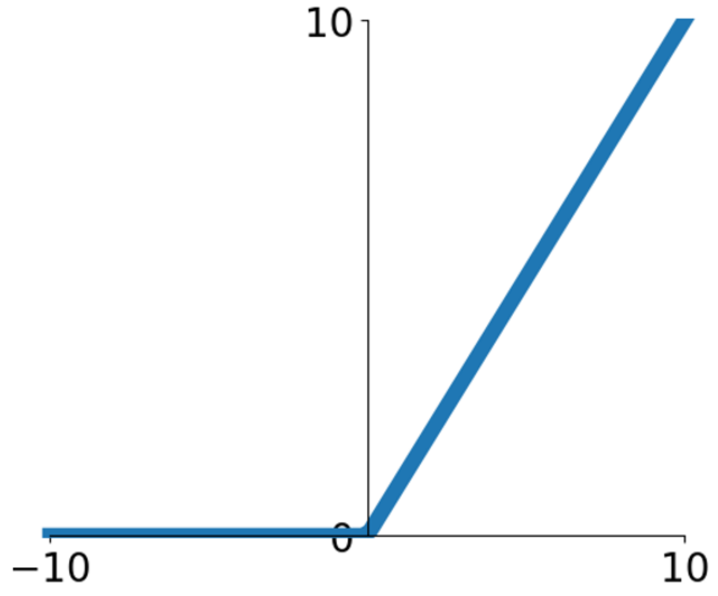
tanh

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



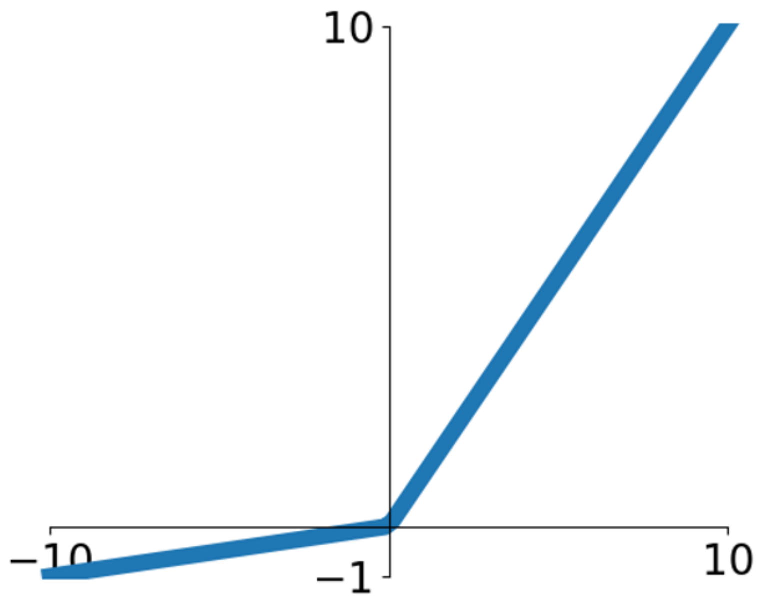
ReLU

$\max(0,x)$



Leaky ReLU

$\max(0.2x, x)$

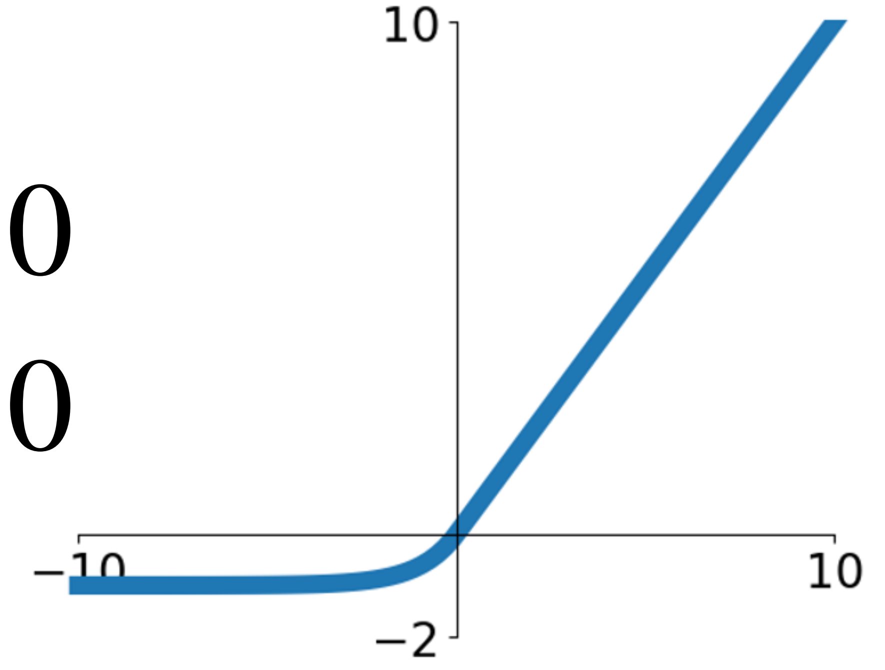


Softplus

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

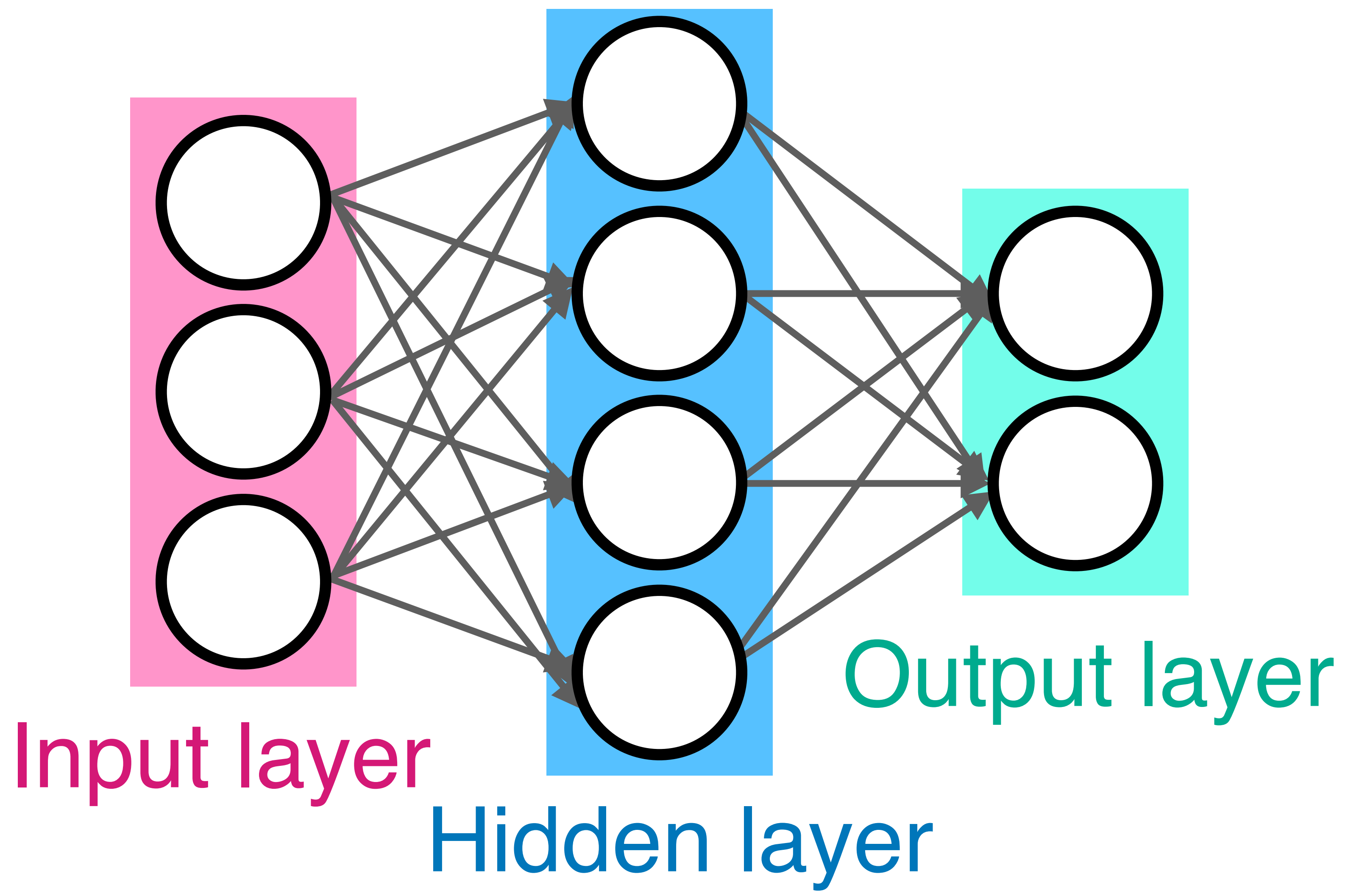
ELU

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



ReLU is a good default choice for most problems

Neural Net in <20 lines!




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

```

Initialize weights and data

```python
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):
```

Compute loss (Sigmoid activation, L2 loss)

```python
  h = 1.0 / (1.0 + np.exp(-x.dot(w1)))
  y_pred = h.dot(w2)
  loss = np.square(y_pred - y).sum()
```

Compute gradients

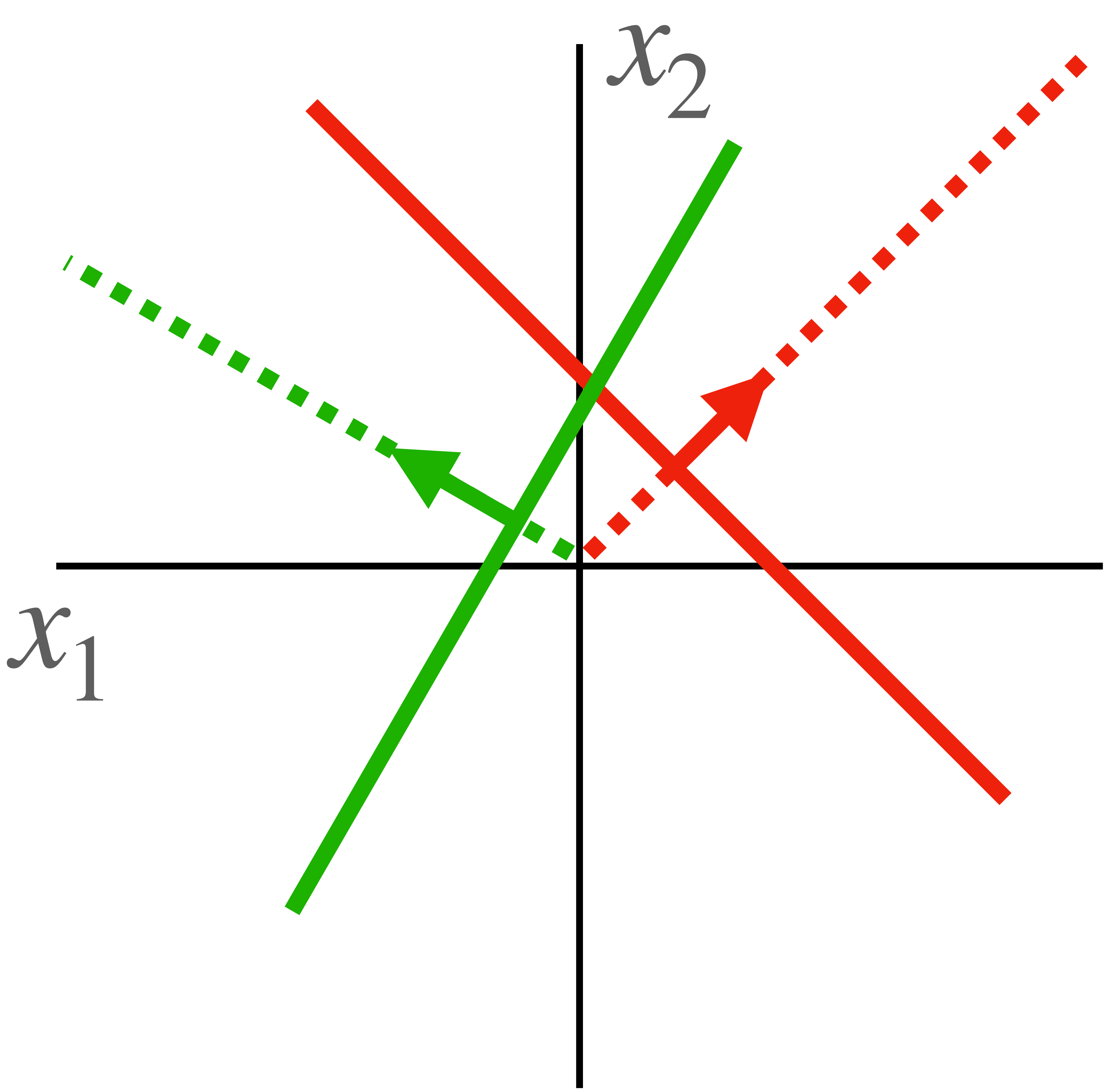
```python
  dy_pred = 2.0 * (y_pred - y)
  dw2 = h.T.dot(dy_pred)
  dh = dy_pred.dot(w2.T)
  dw1 = x.T.dot(dh * h * (1 - h))
```

SGD step

```python
  w1 -= 1e-4 * dw1
  w2 -= 1e-4 * dw2
```

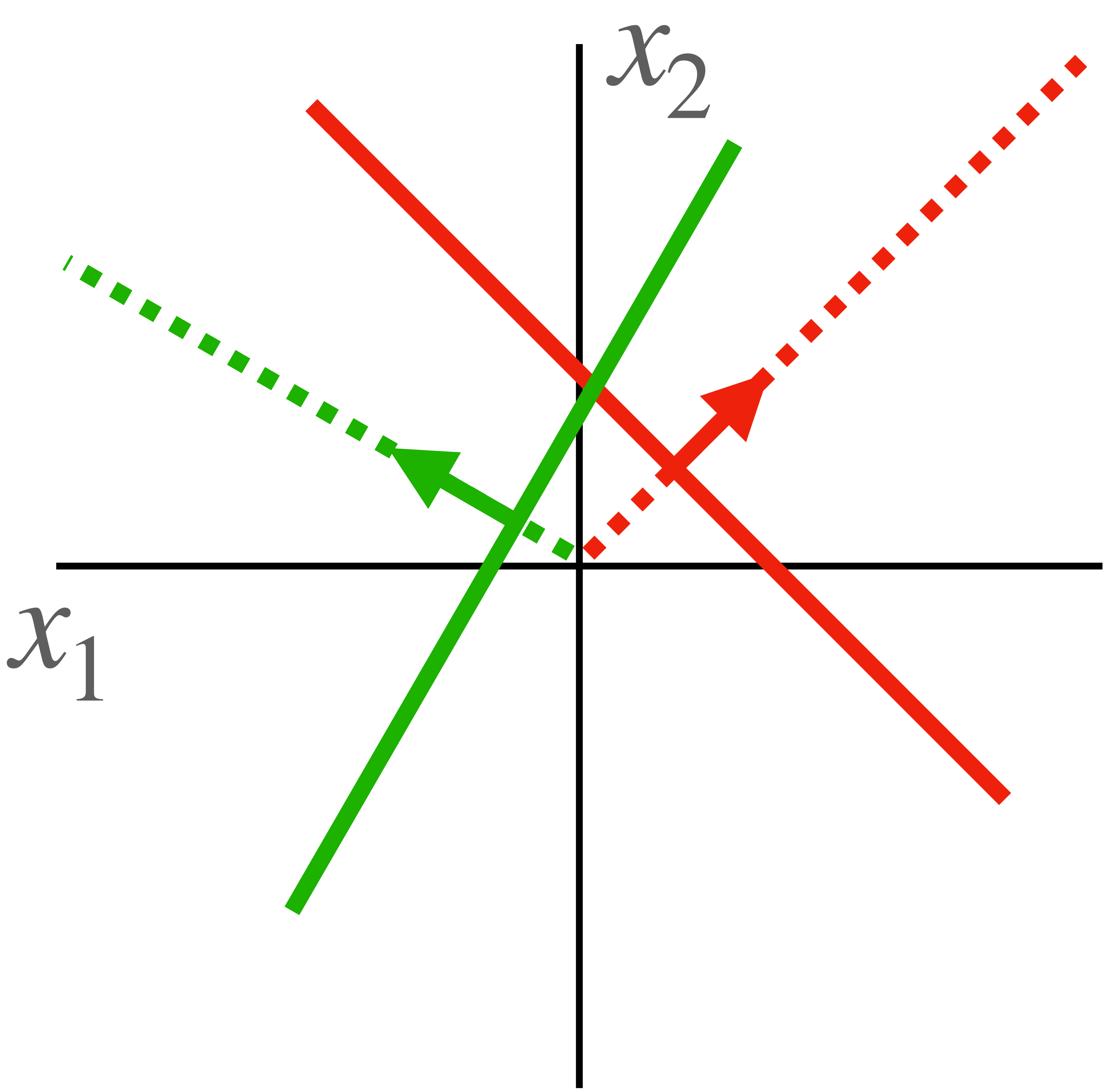
Space Warping

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



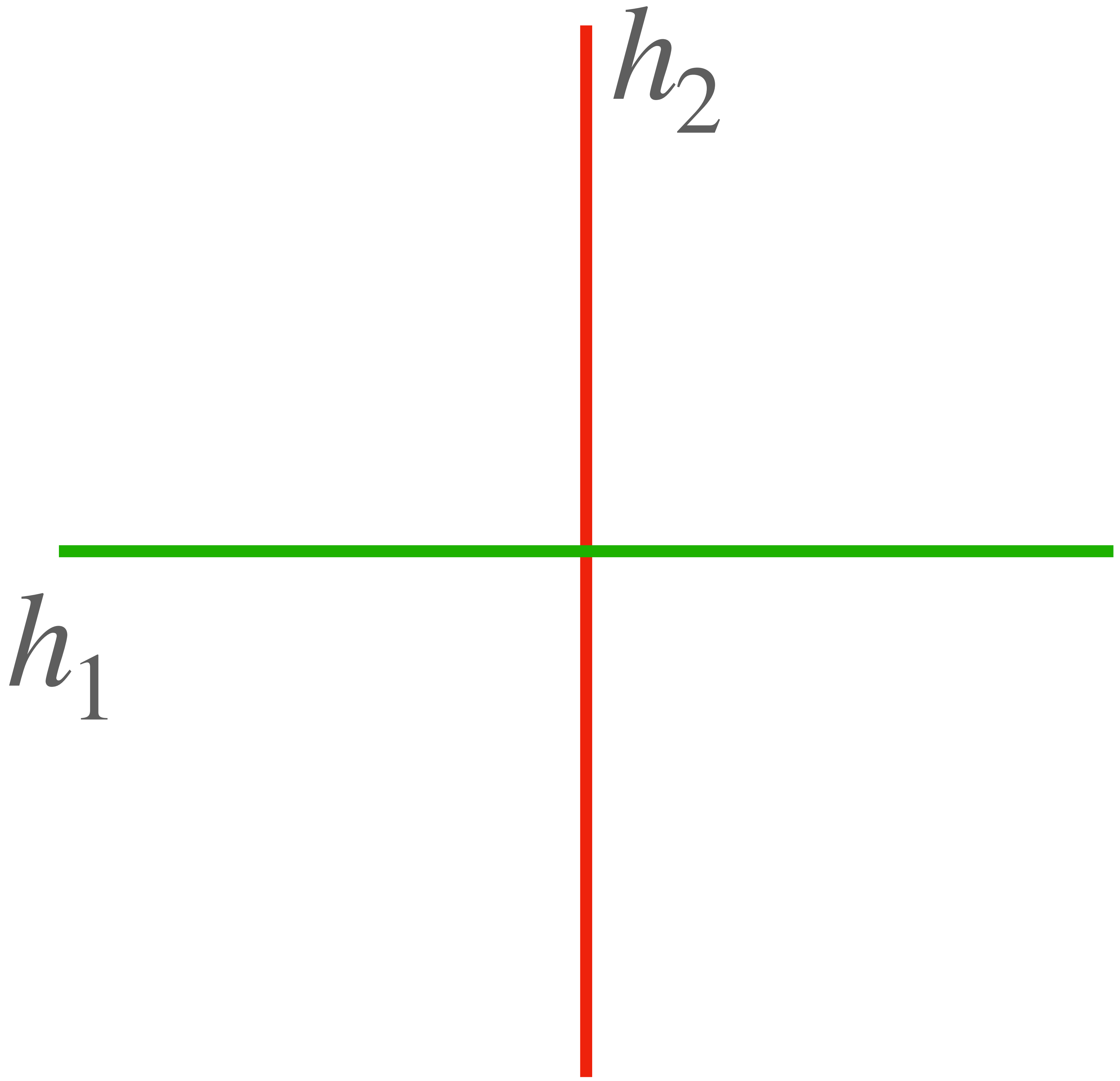
Space Warping

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



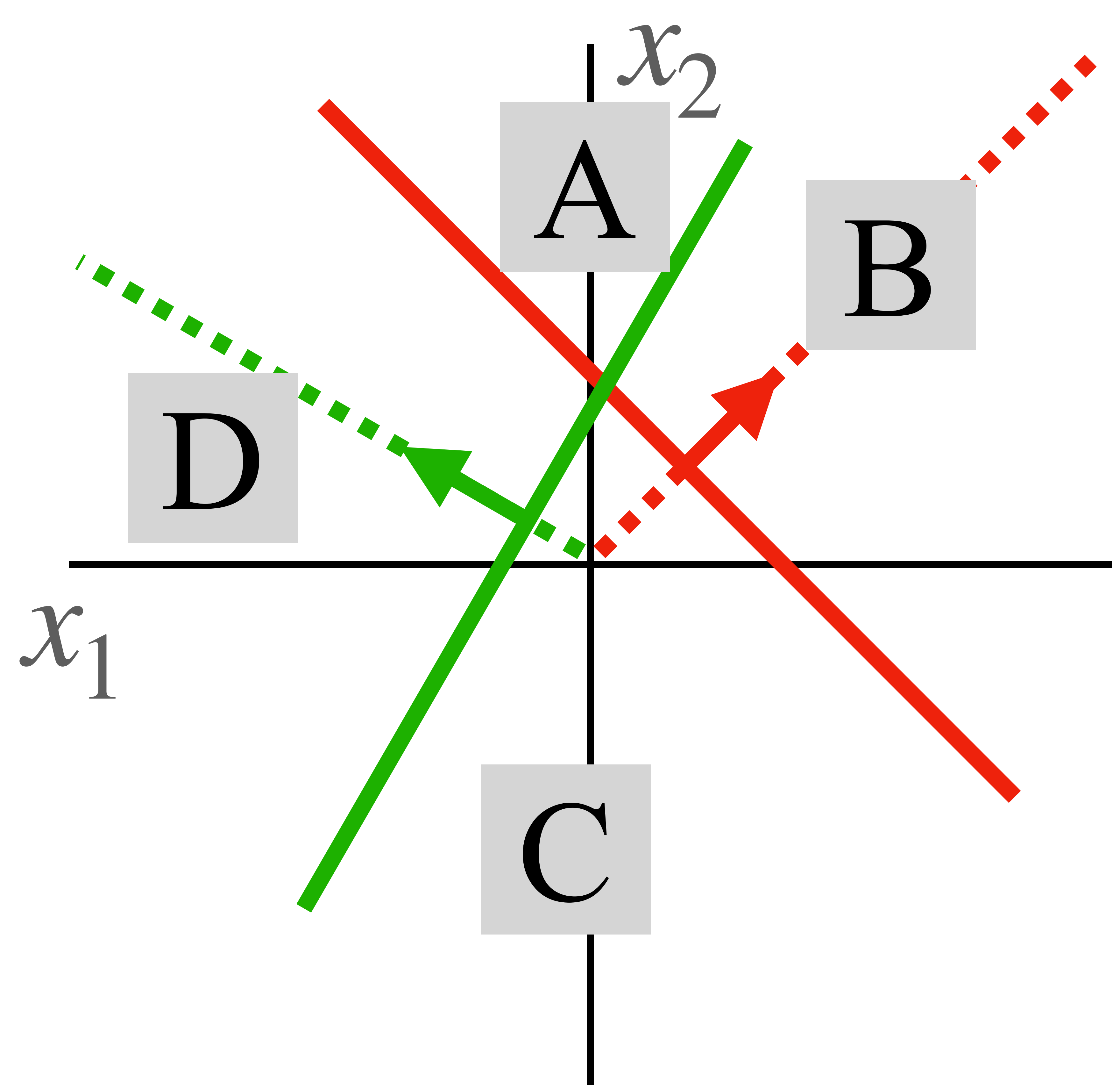
Feature transform:

$$h = Wx + b$$

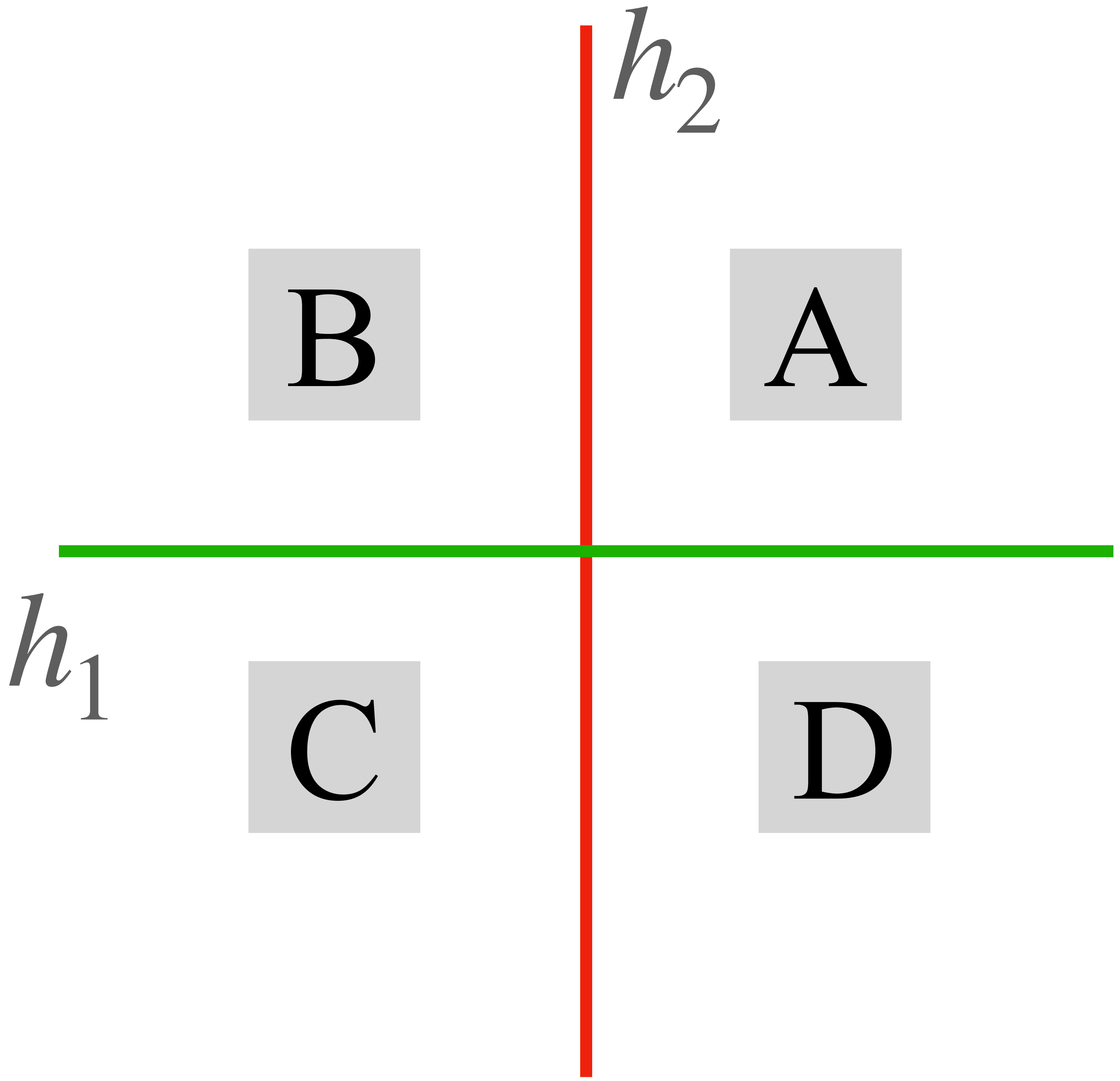


Space Warping

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

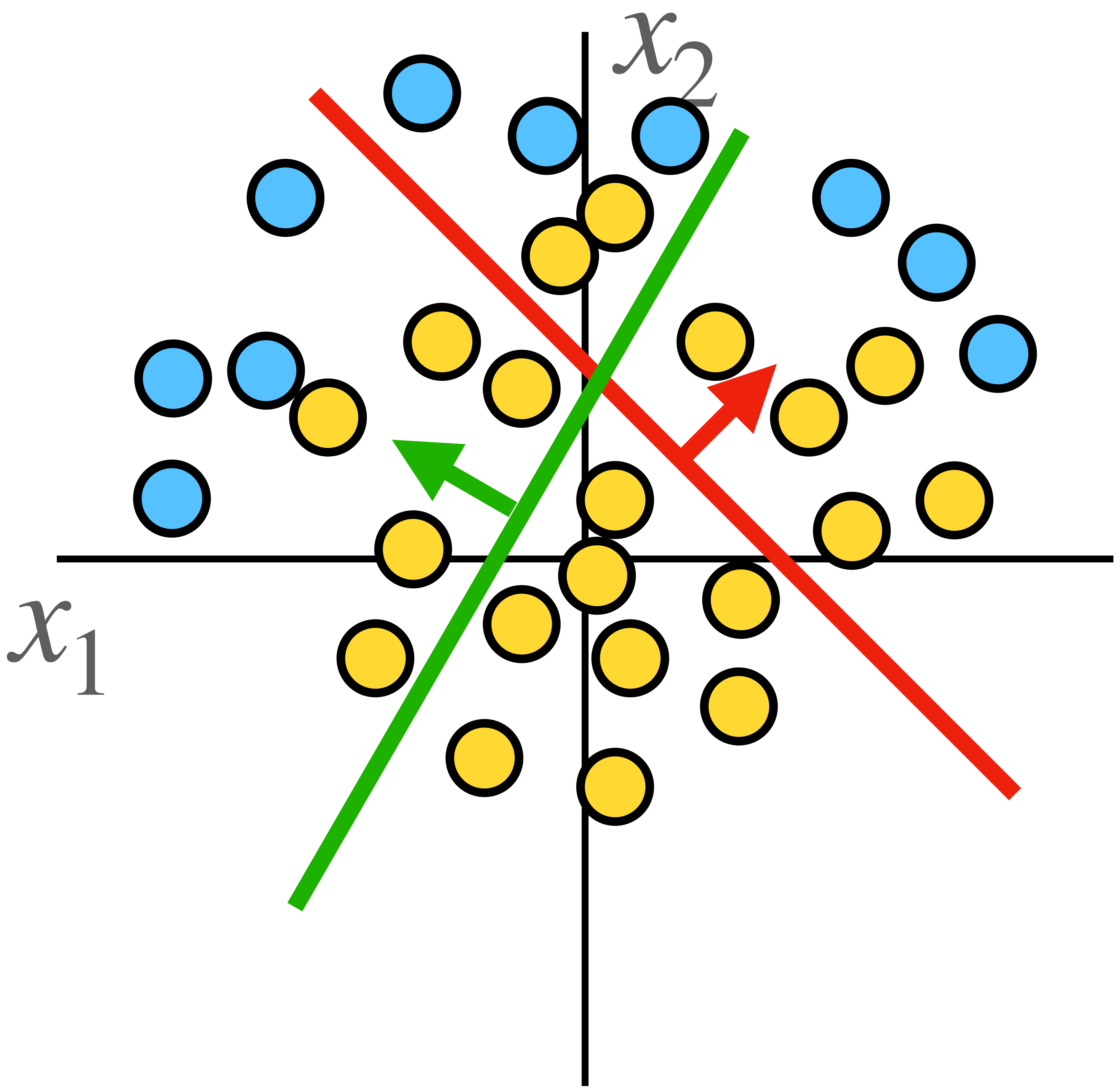


Feature transform:
$h = Wx + b$



Space Warping

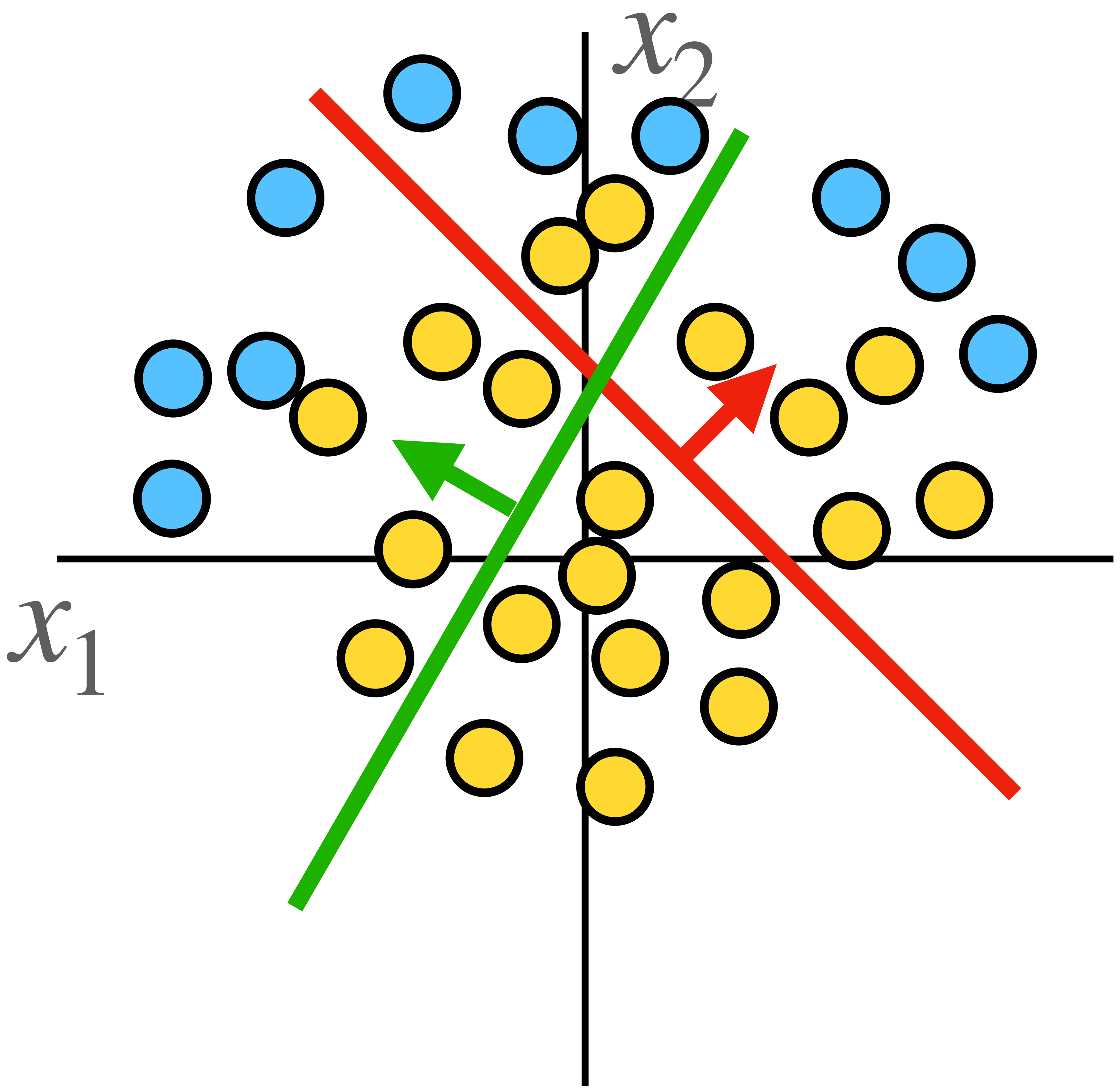
Points not linearly separable in original space



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

Space Warping

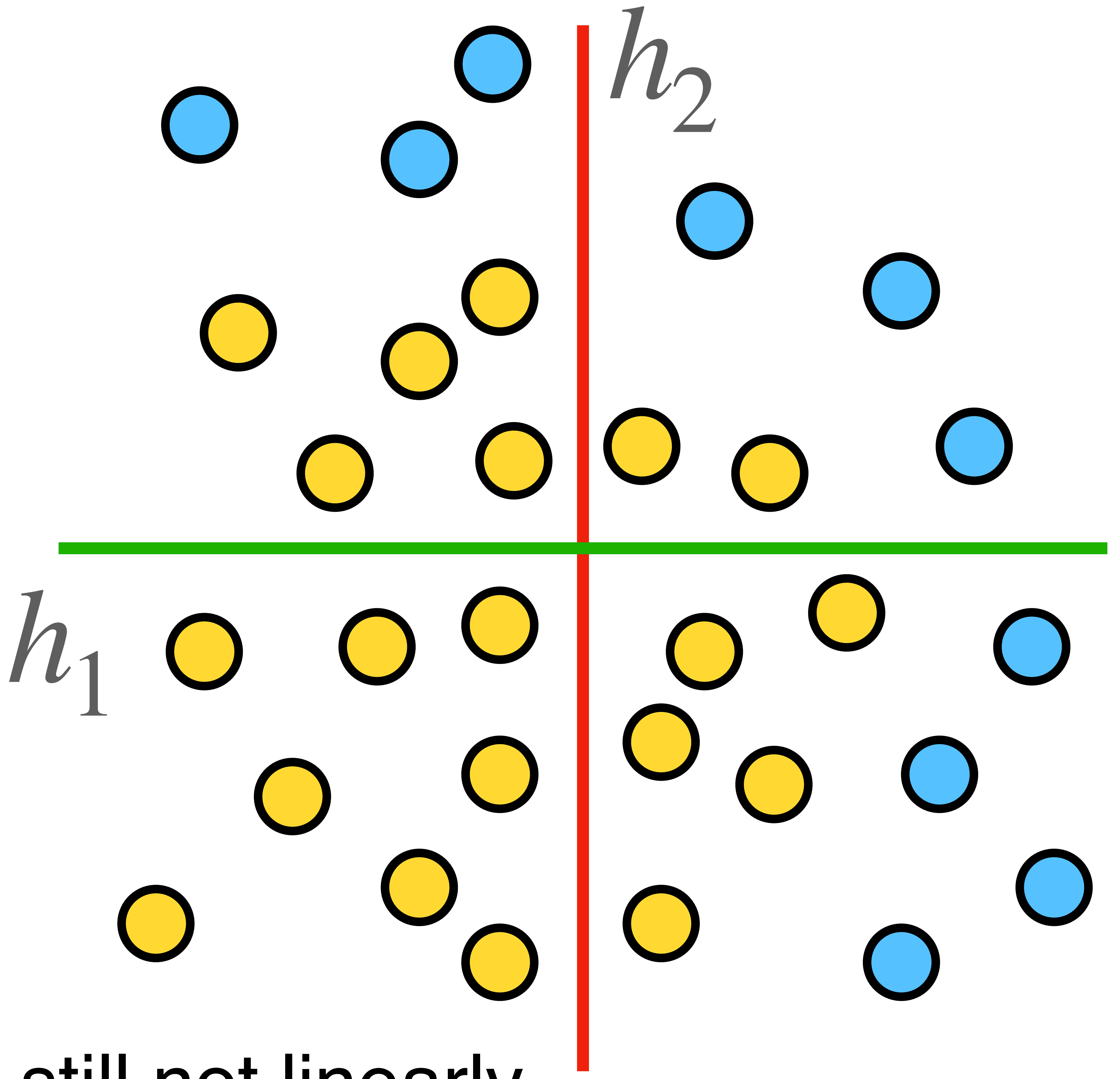
Points not linearly separable in original space



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

Feature transform:

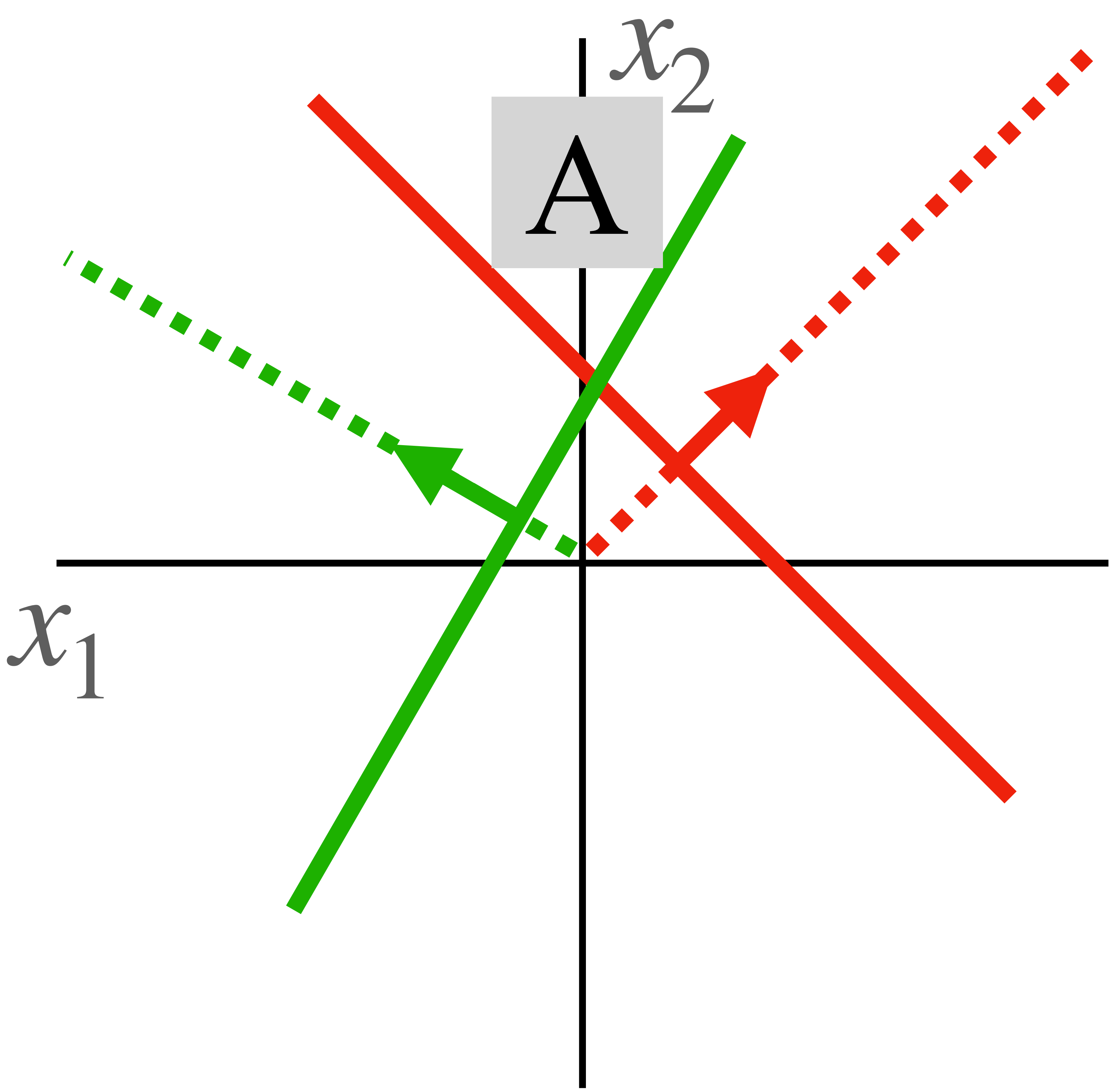
$$h = Wx + b$$



Points still not linearly separable in feature space

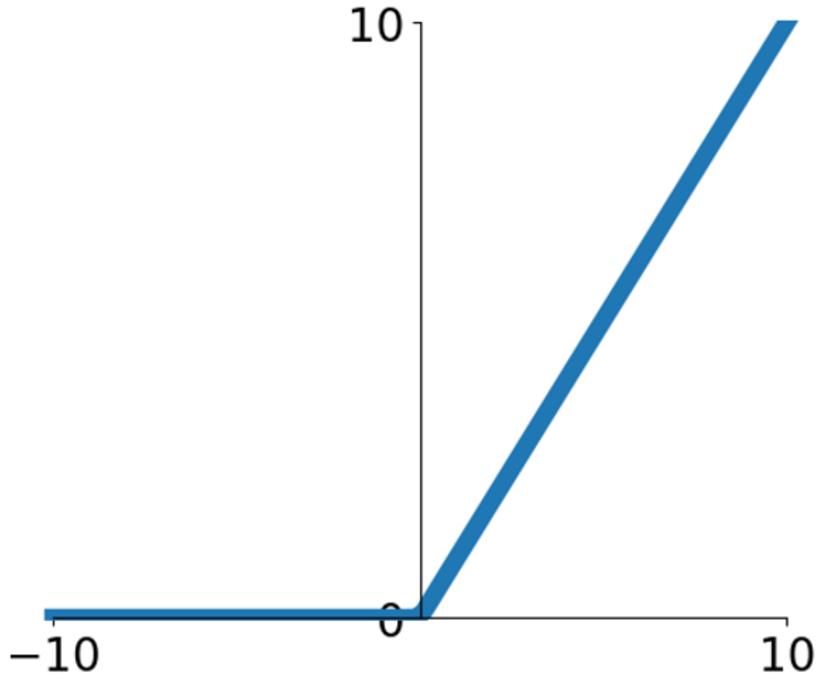
Space Warping

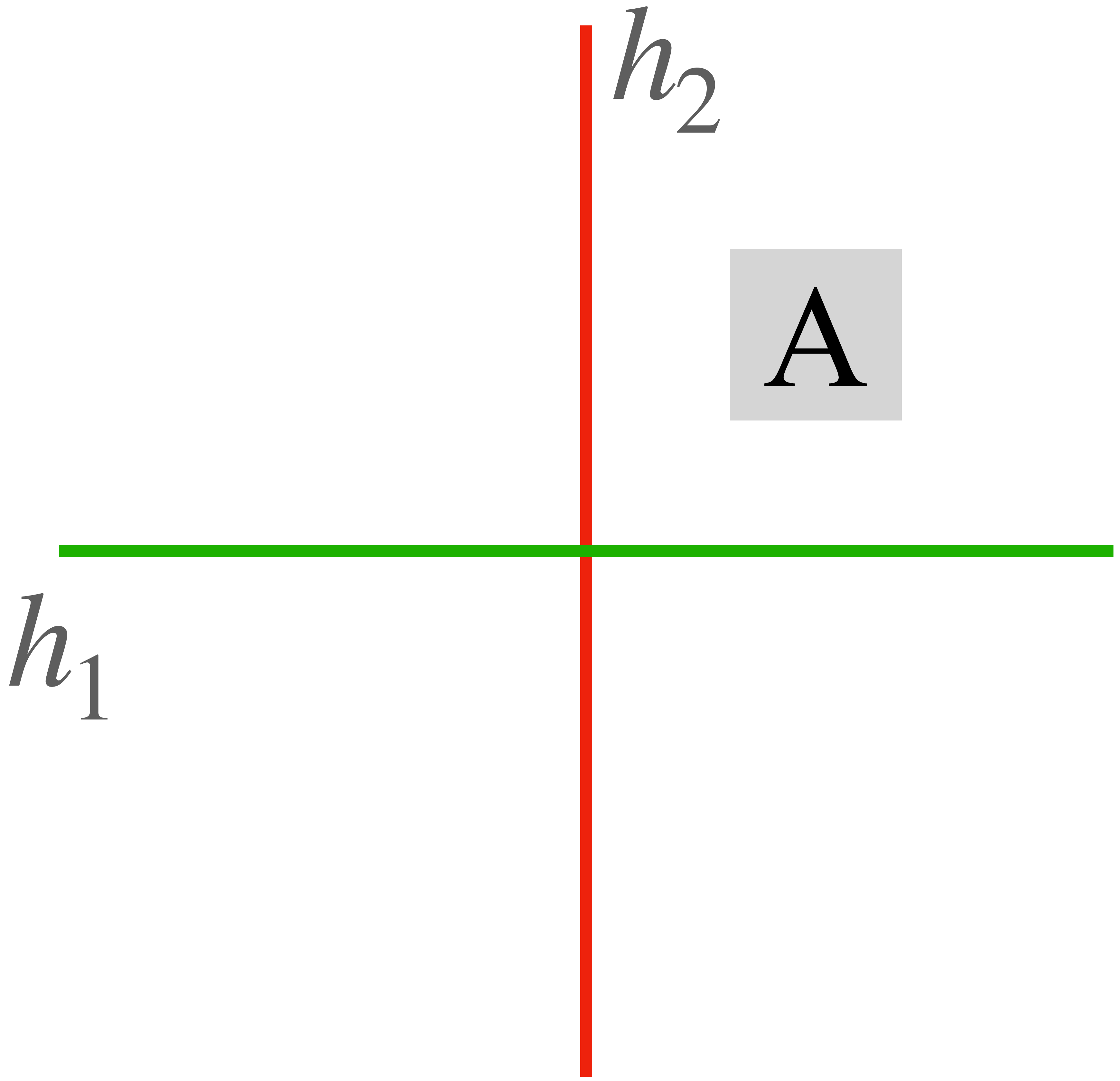
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

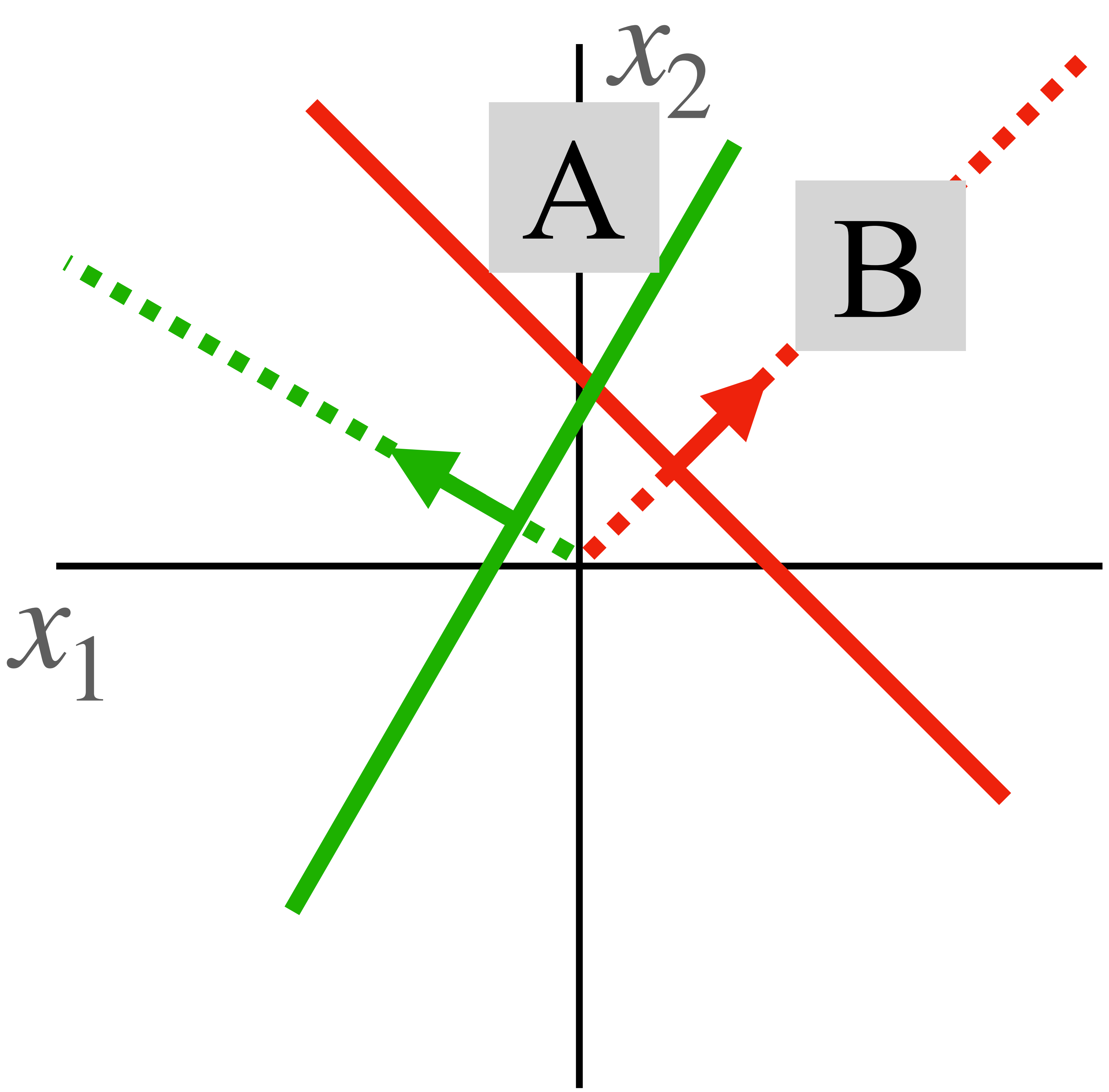
$h = ReLU(Wx + b)$





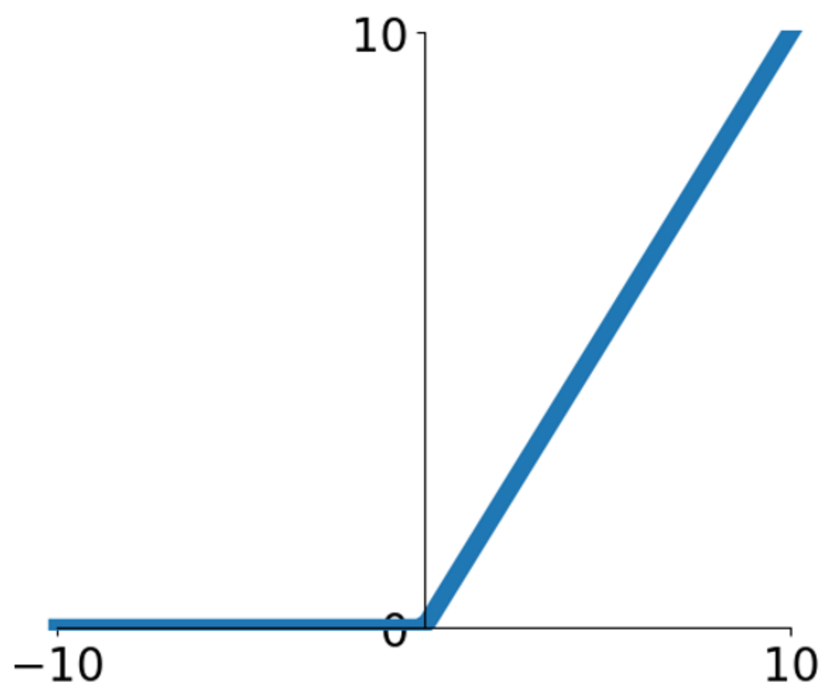
Space Warping

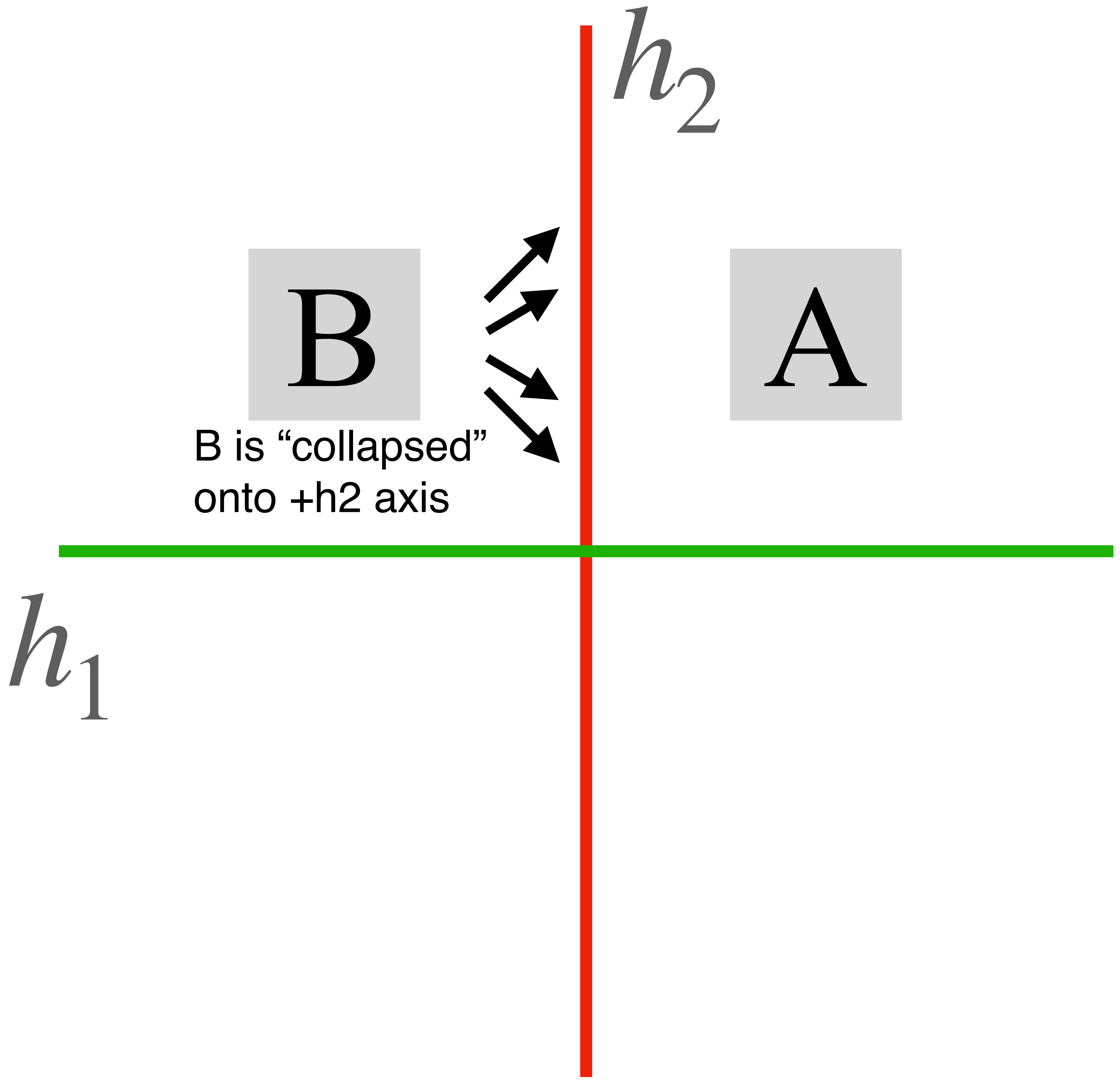
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

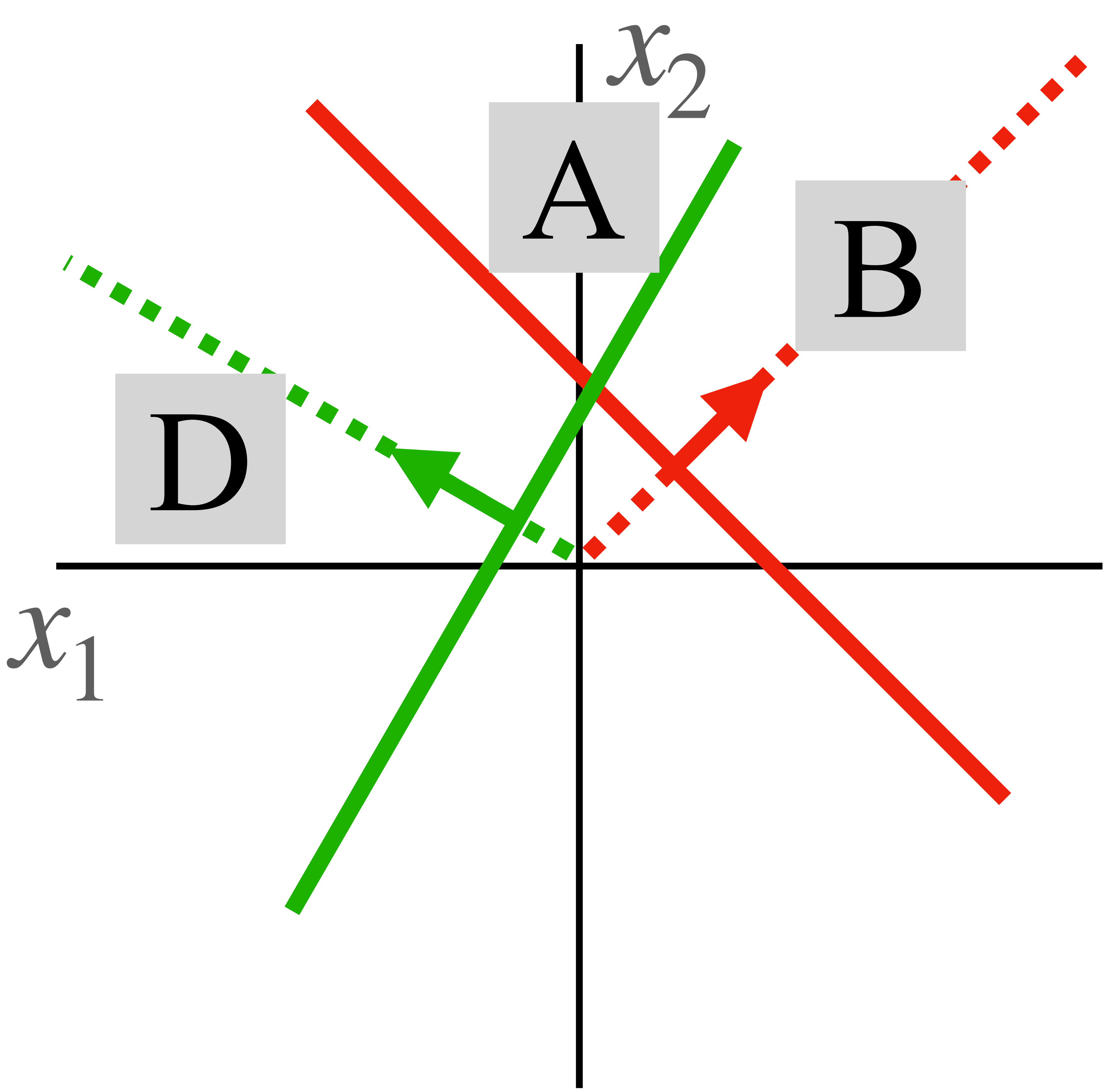
$h = ReLU(Wx + b)$





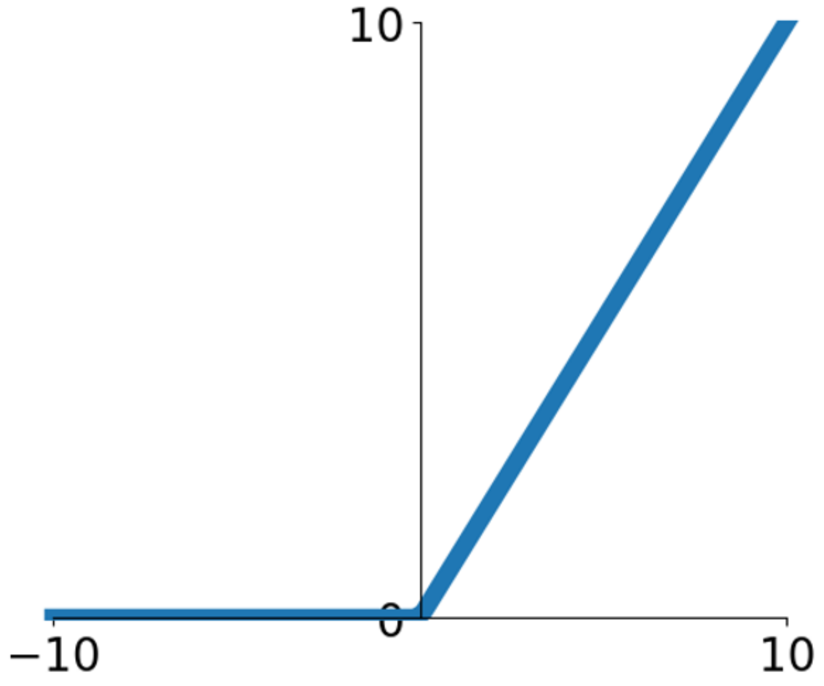
Space Warping

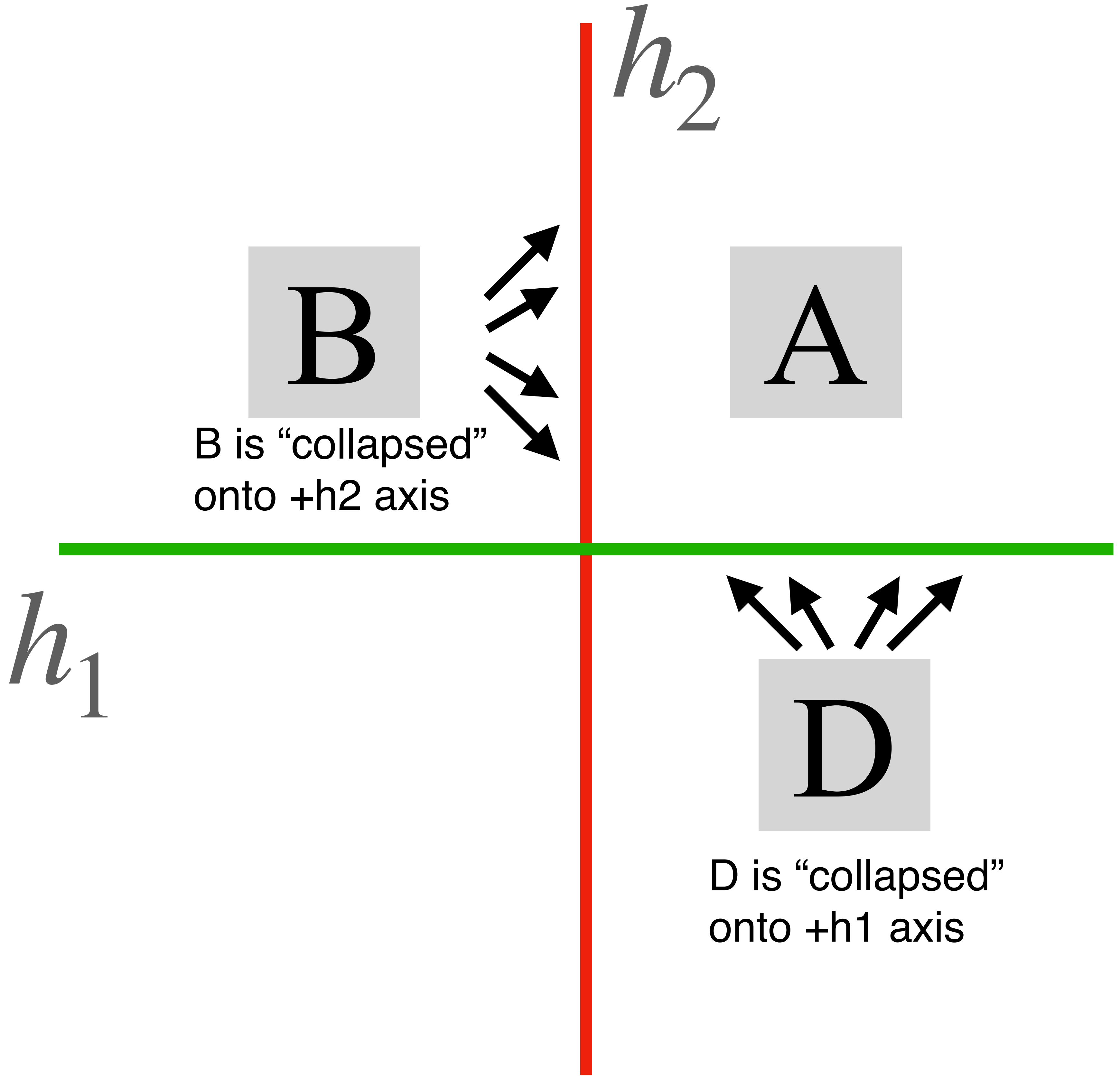
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

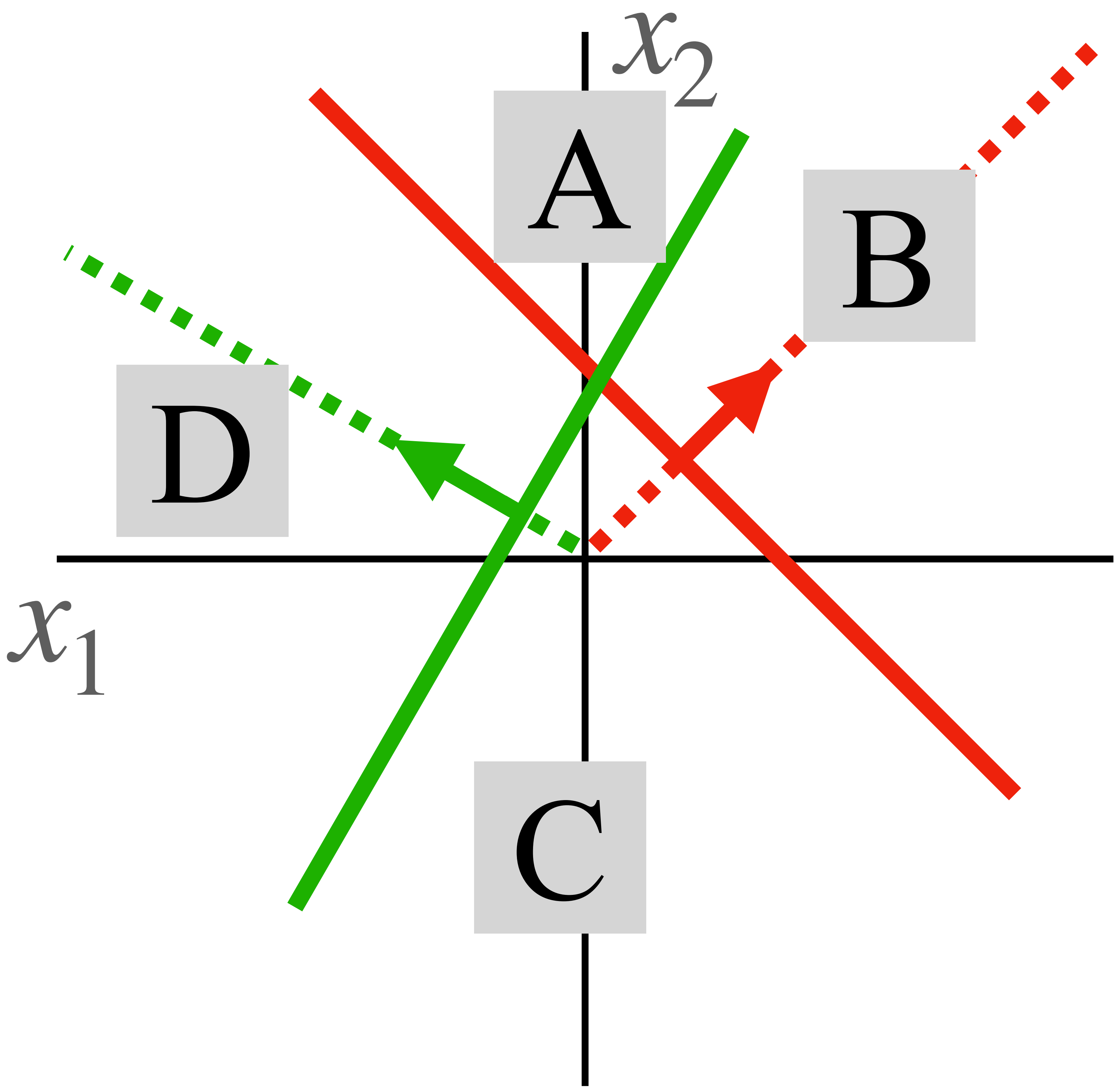
$h = ReLU(Wx + b)$





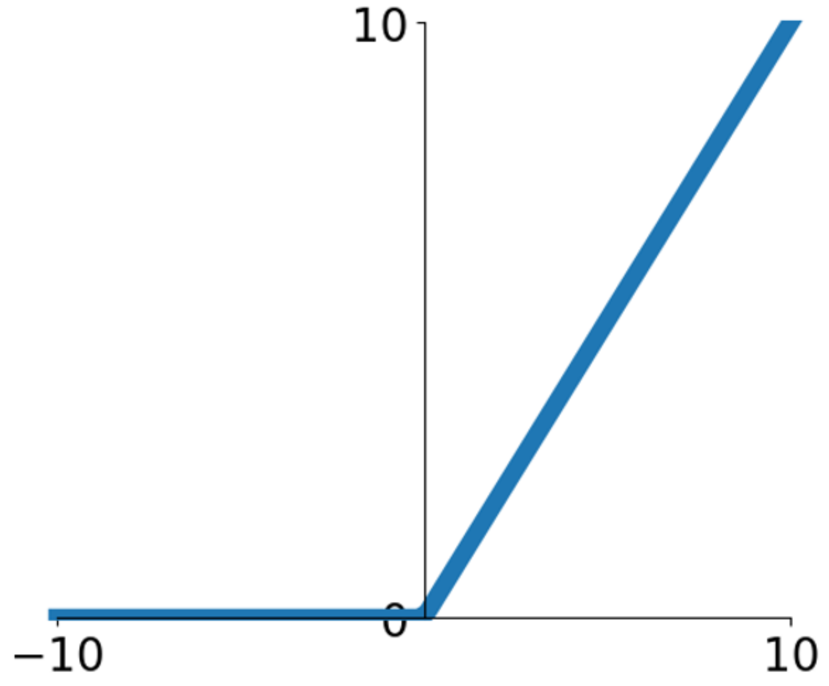
Space Warping

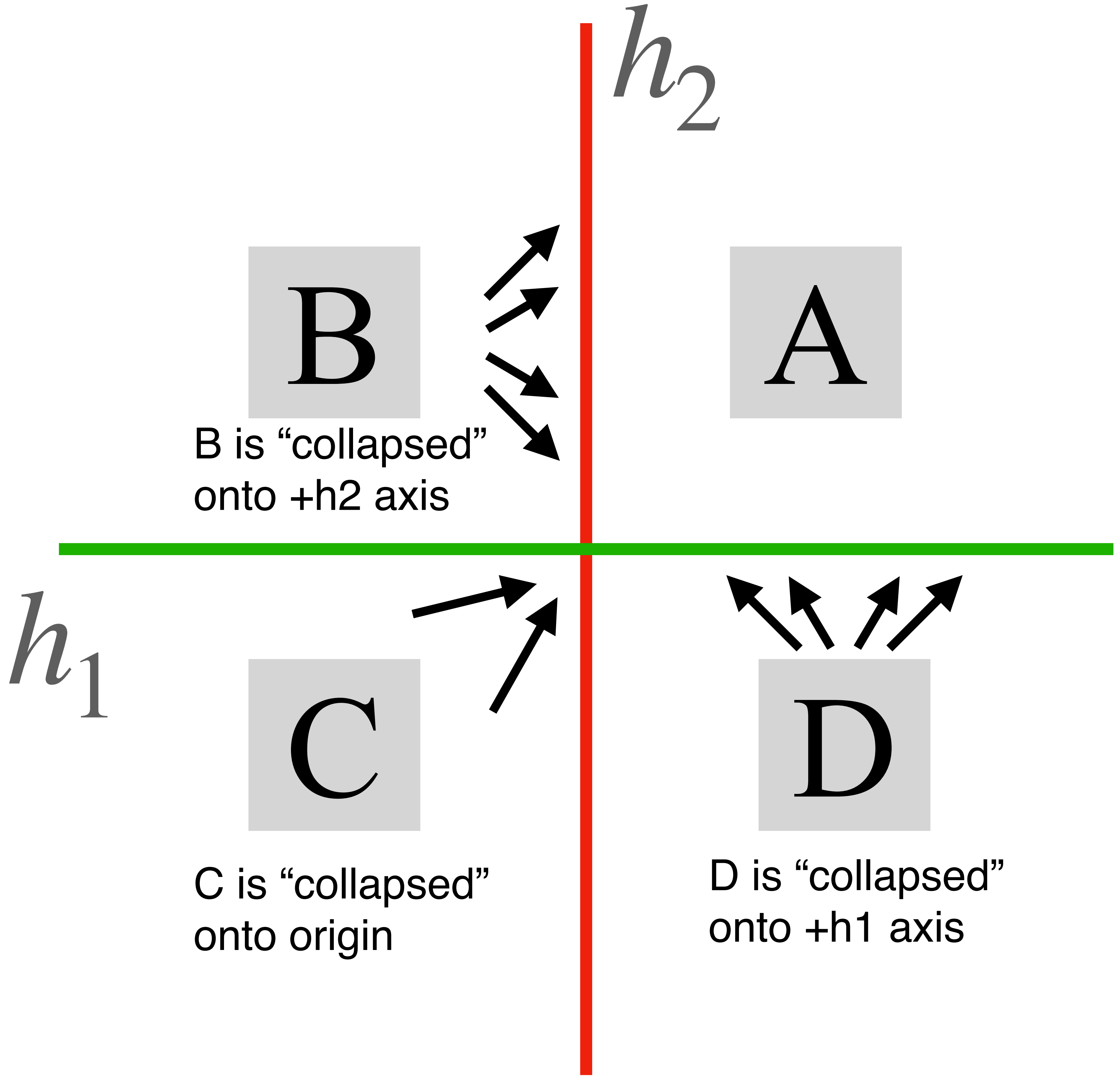
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

$h = ReLU(Wx + b)$

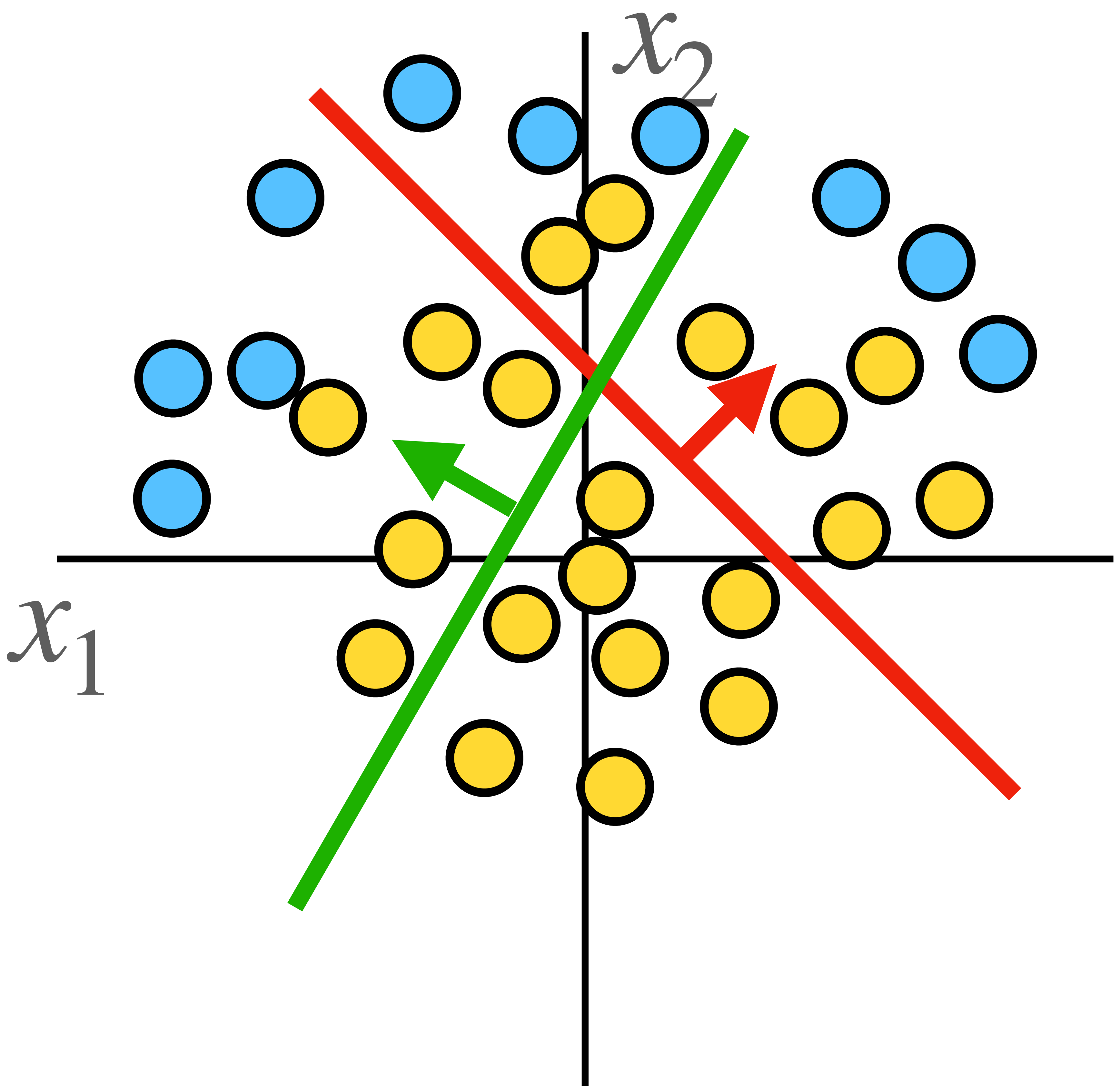




Space Warping

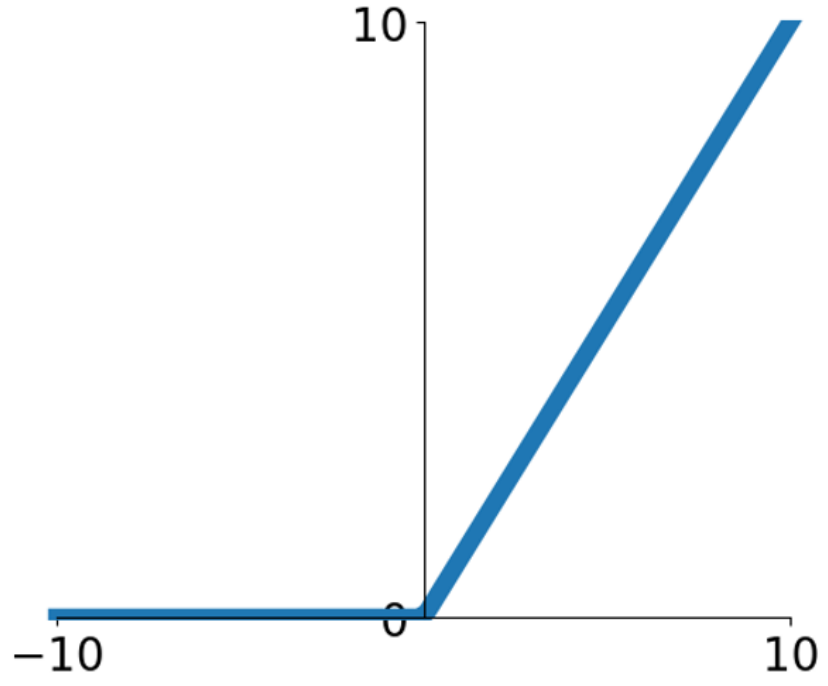
Points not linearly separable in original space

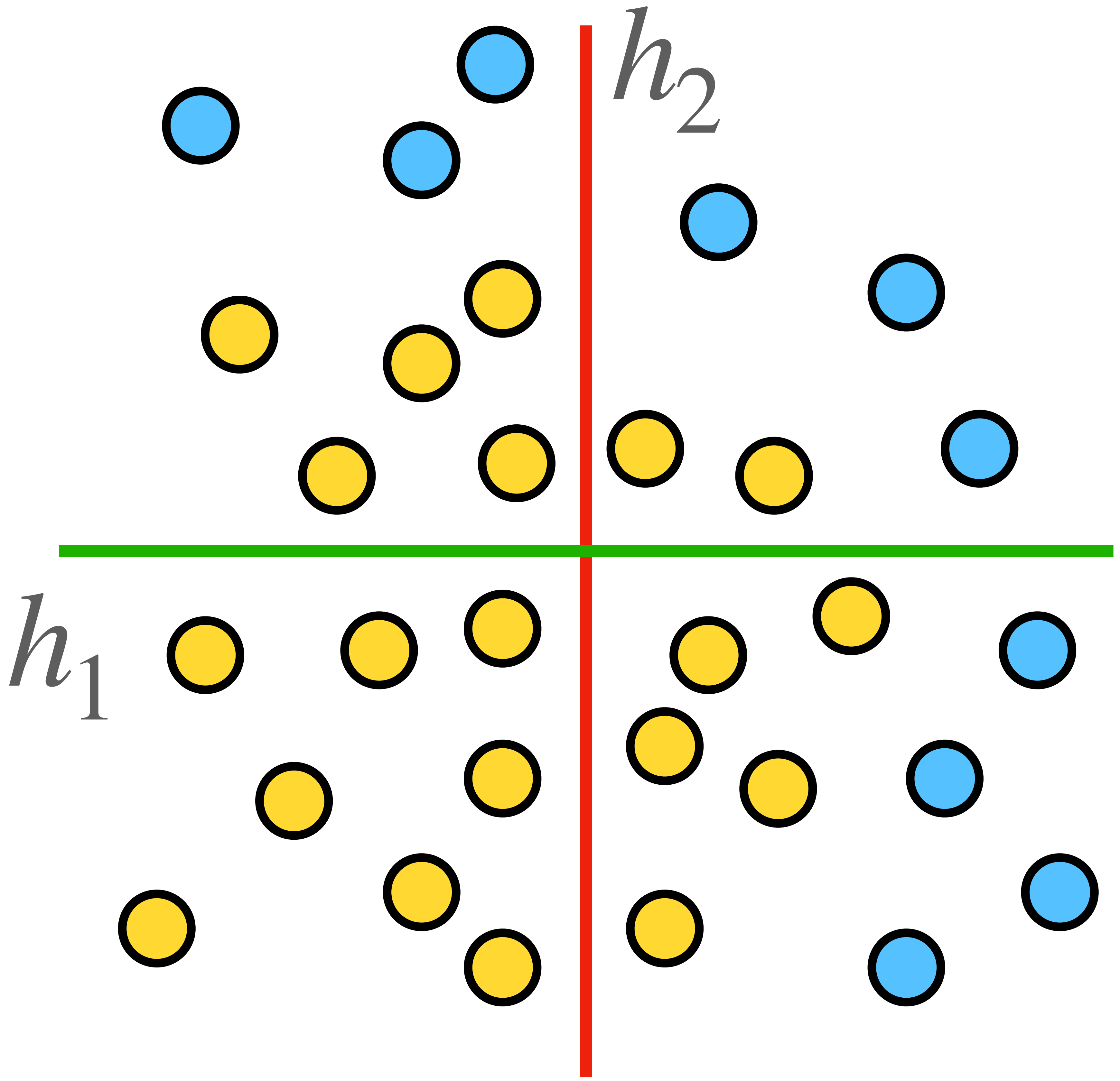
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

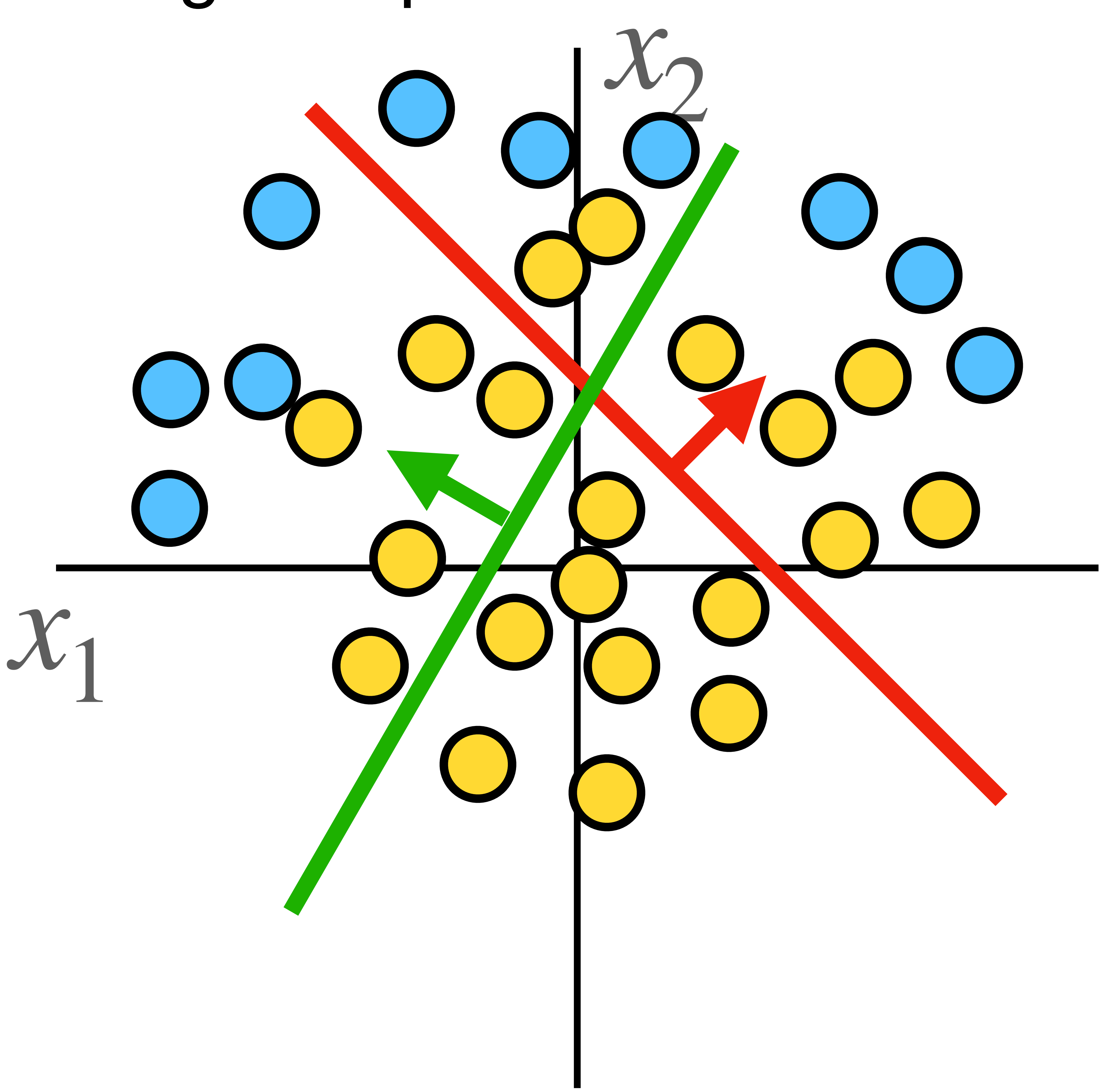
$h = ReLU(Wx + b)$





Space Warping

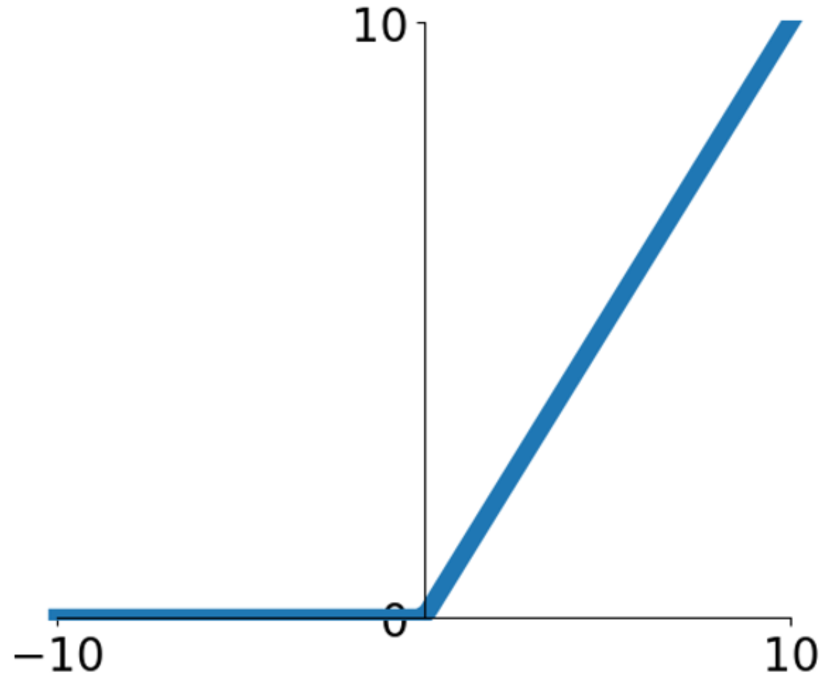
Points not linearly separable in original space

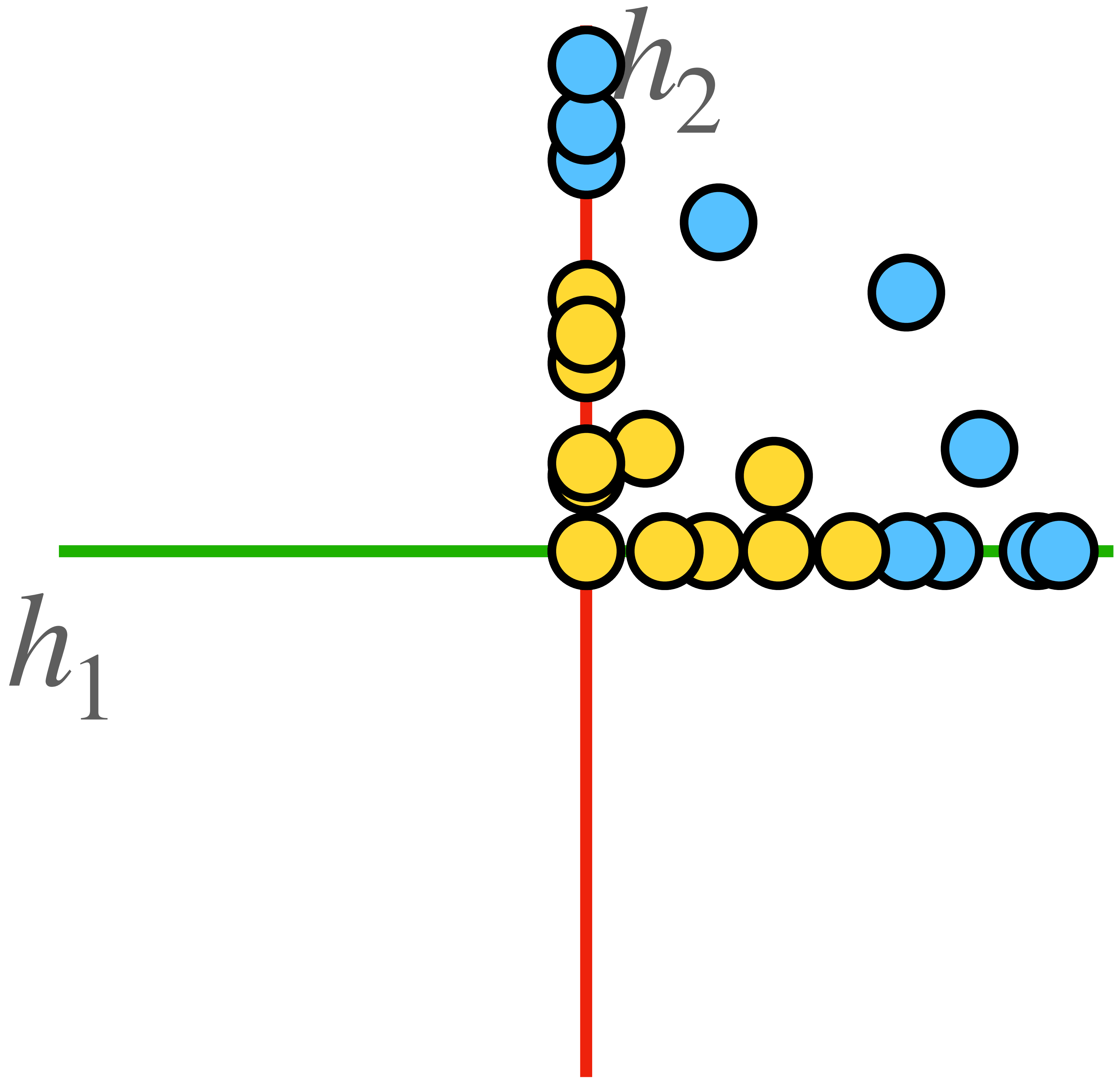


Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional

Feature transform:

$h = ReLU(Wx + b)$

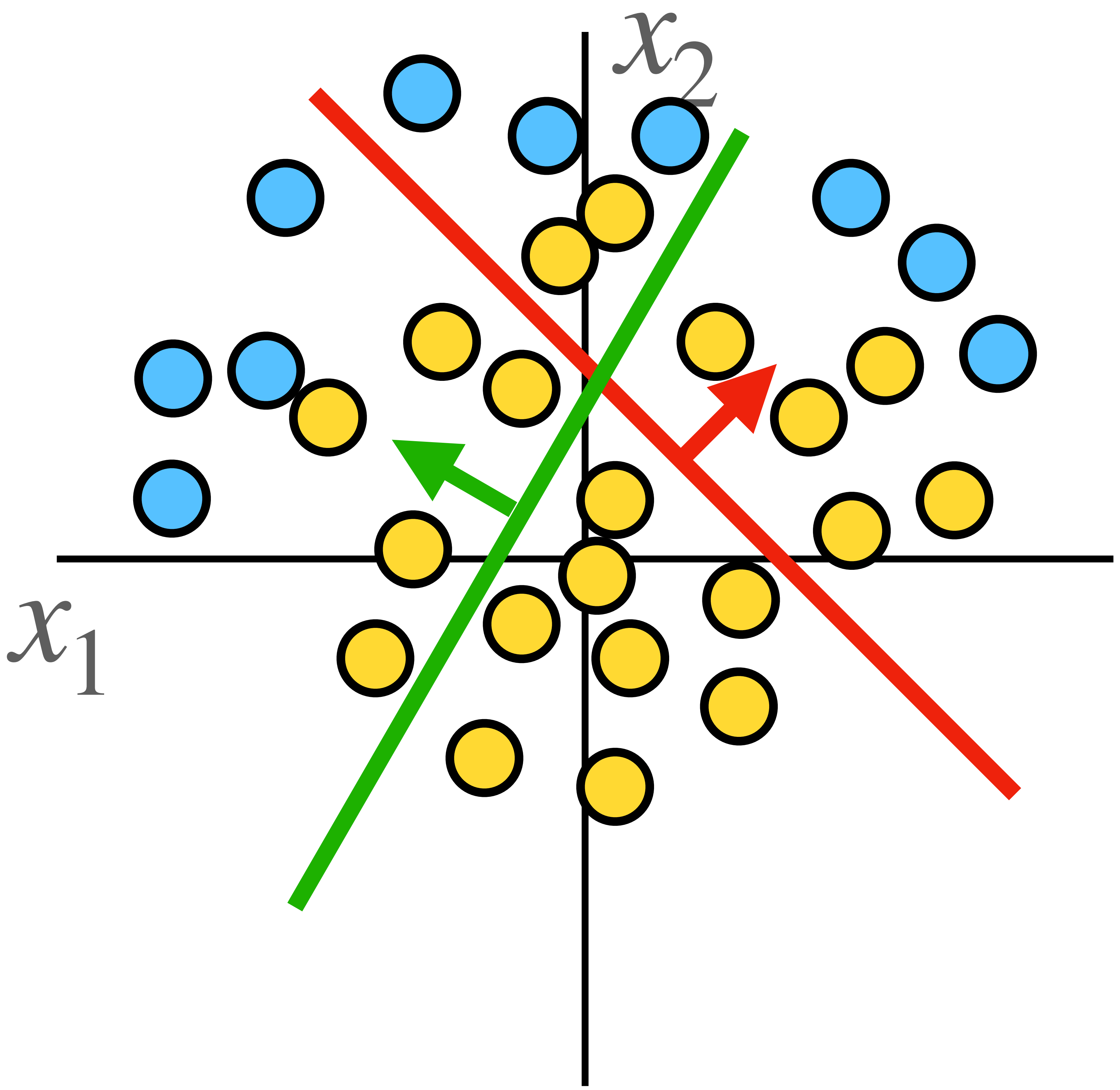




Space Warping

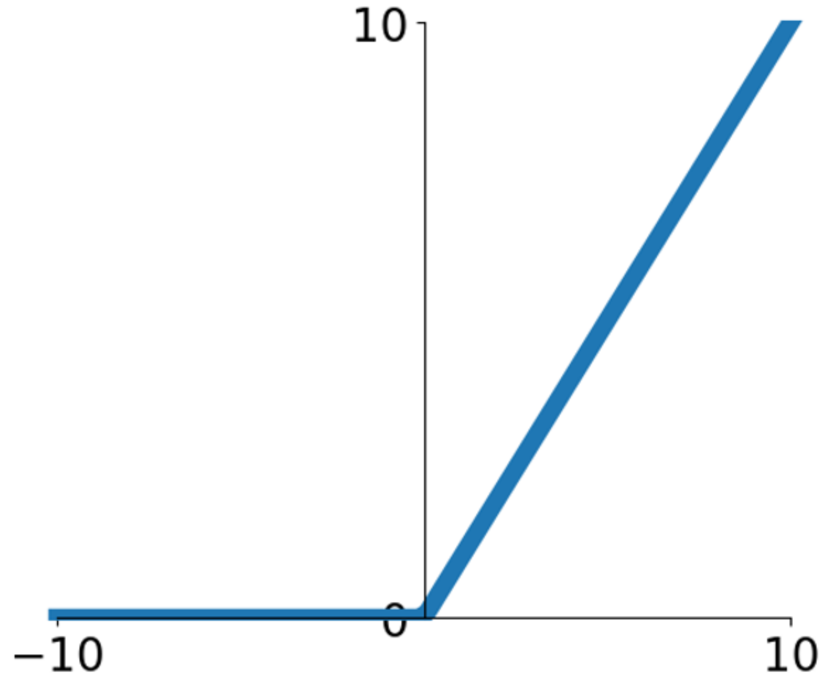
Points not linearly separable in original space

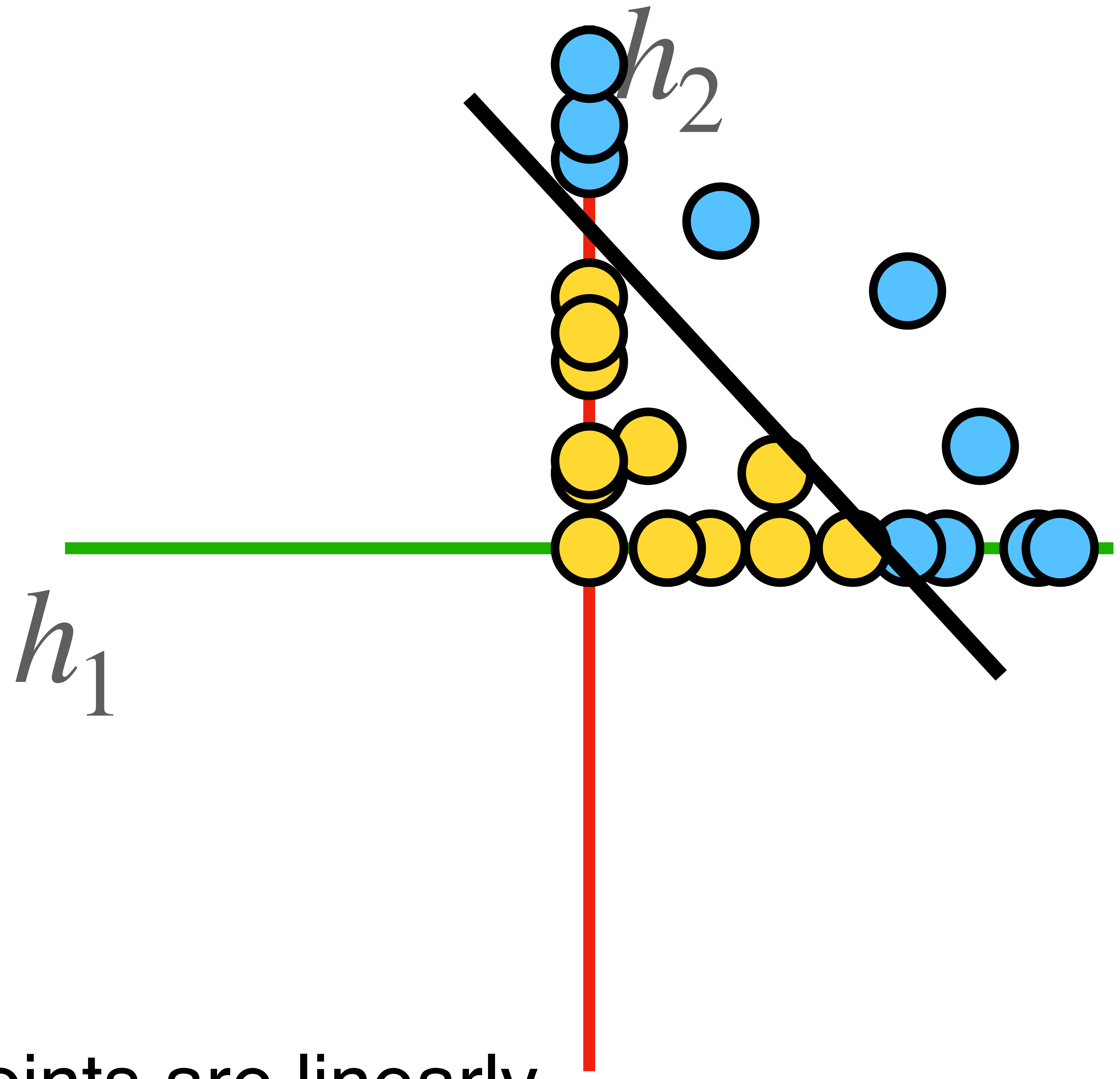
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

$h = ReLU(Wx + b)$



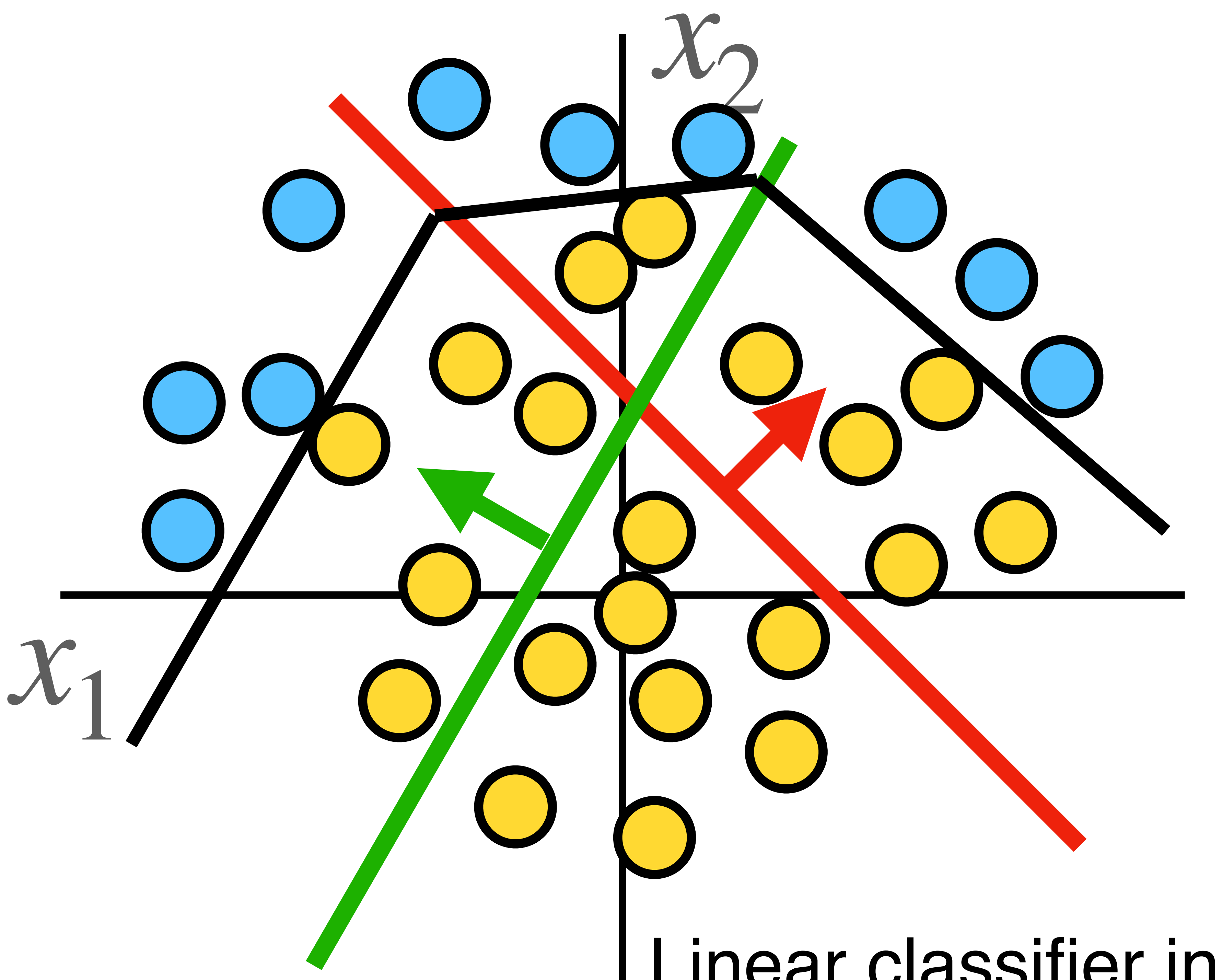


Points are linearly separable in feature space!

Space Warping

Points not linearly separable in original space

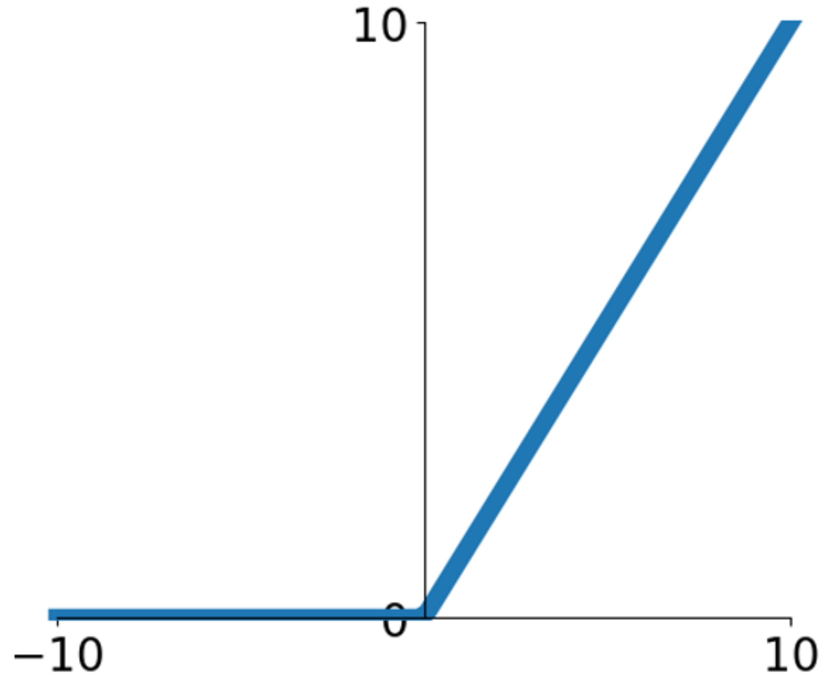
Consider a neural net hidden layer: $h = ReLU(Wx + b)$ $= \max(0, Wx + b)$ where x, b, h are each 2-dimensional

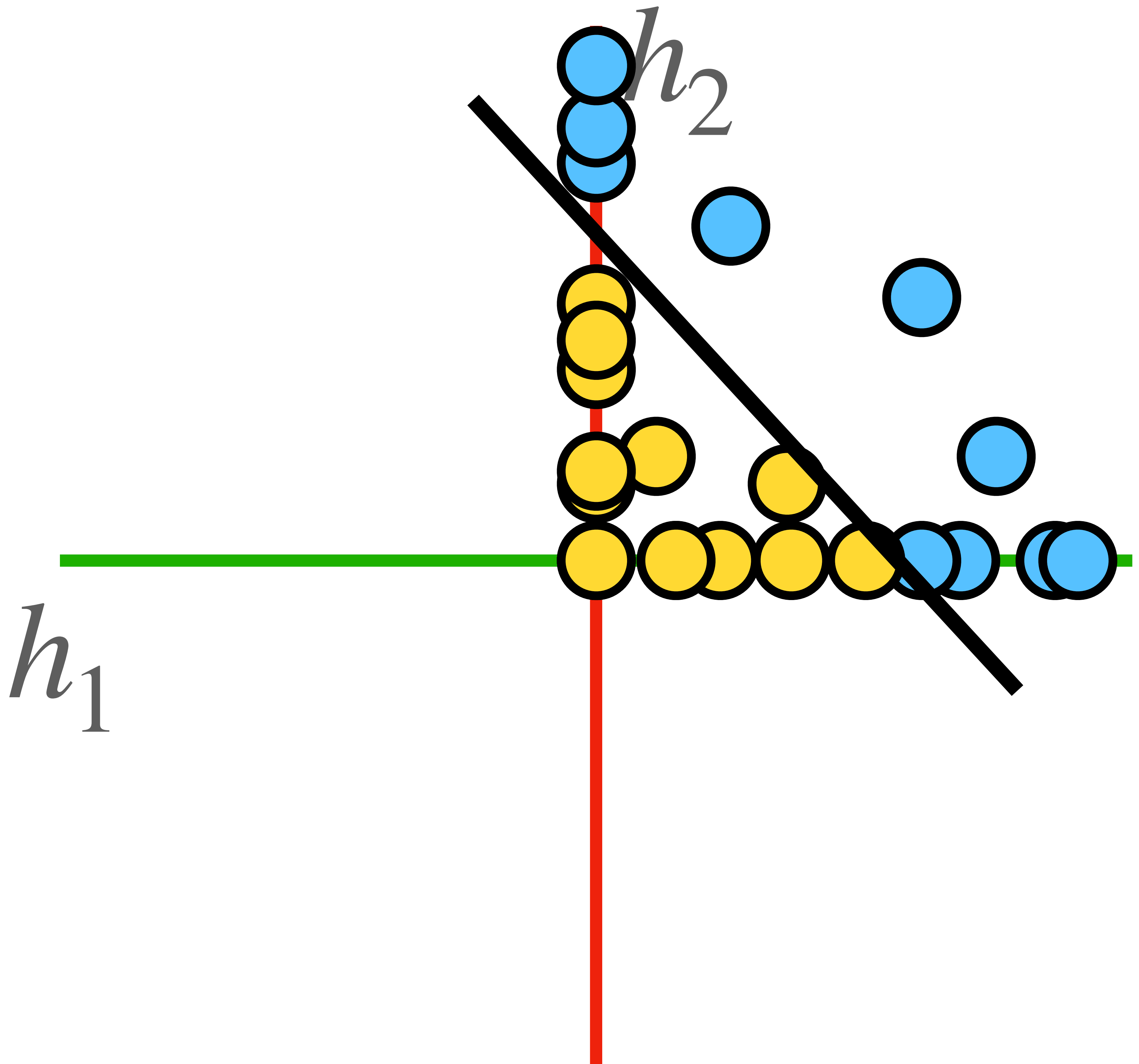


Linear classifier in feature space gives nonlinear classifier in original space

Feature transform:

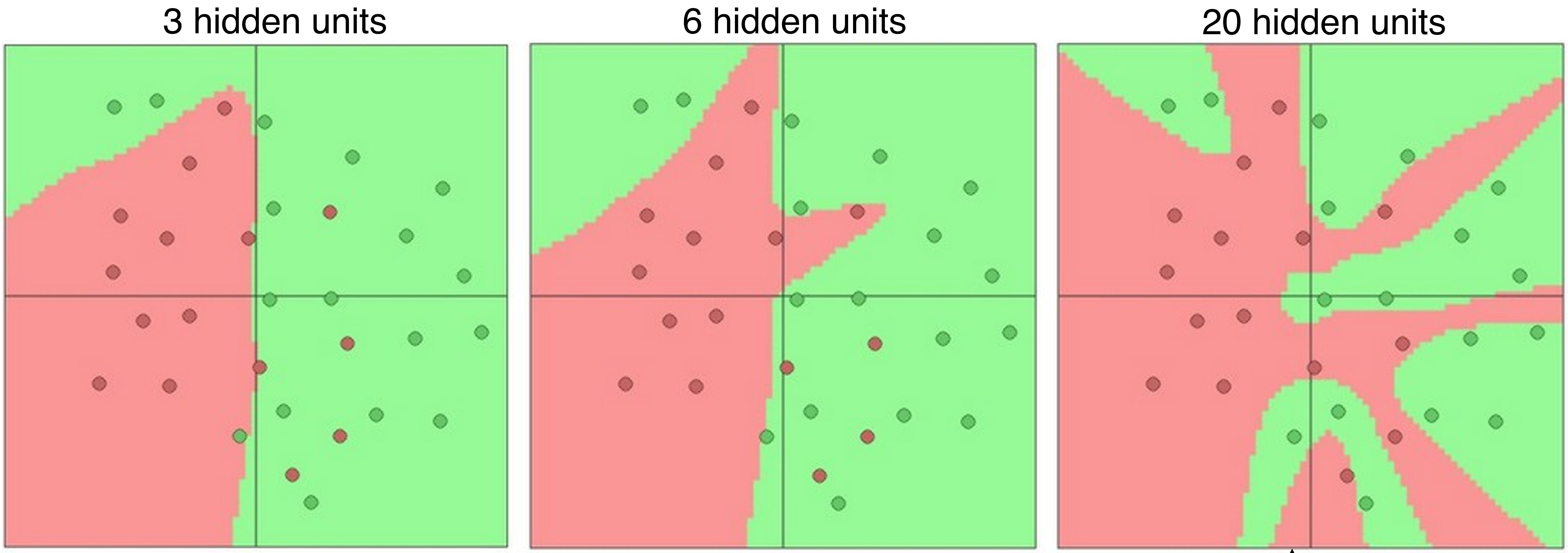
$h = ReLU(Wx + b)$





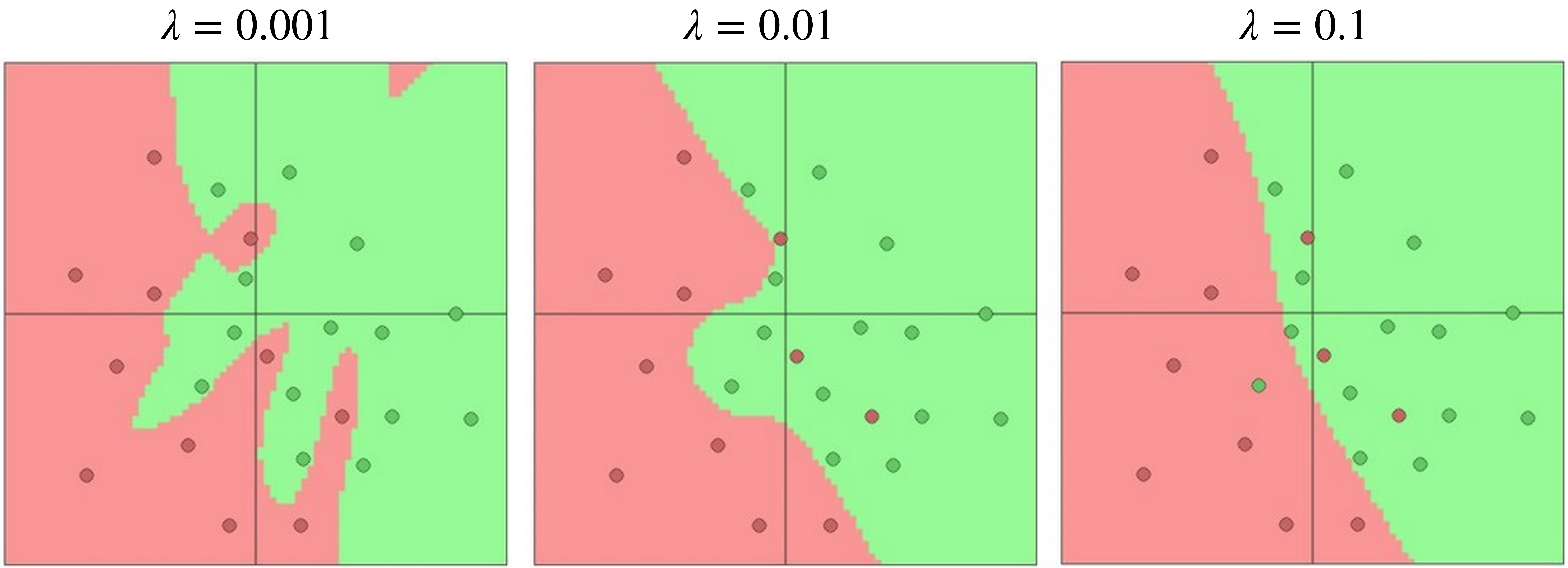
Points are linearly separable in feature space!

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

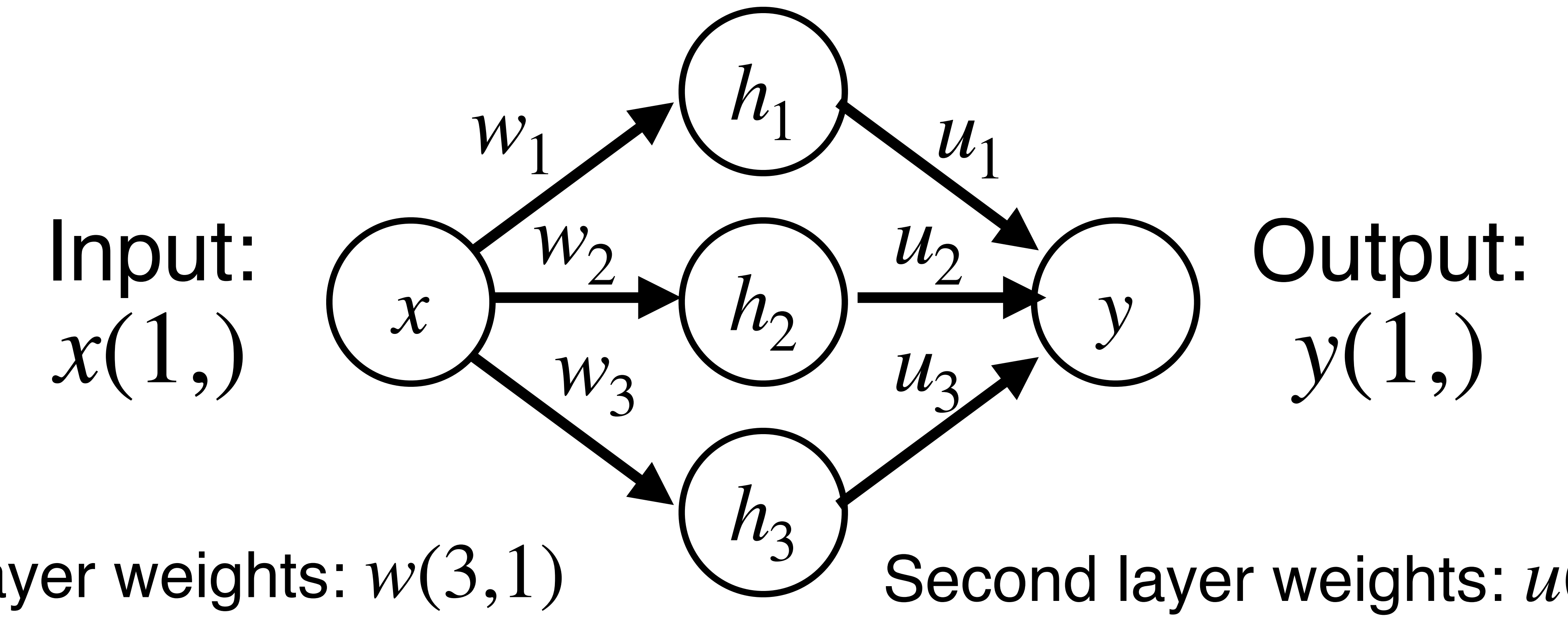
Universal Approximation

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



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

First layer bias: $b(3,)$

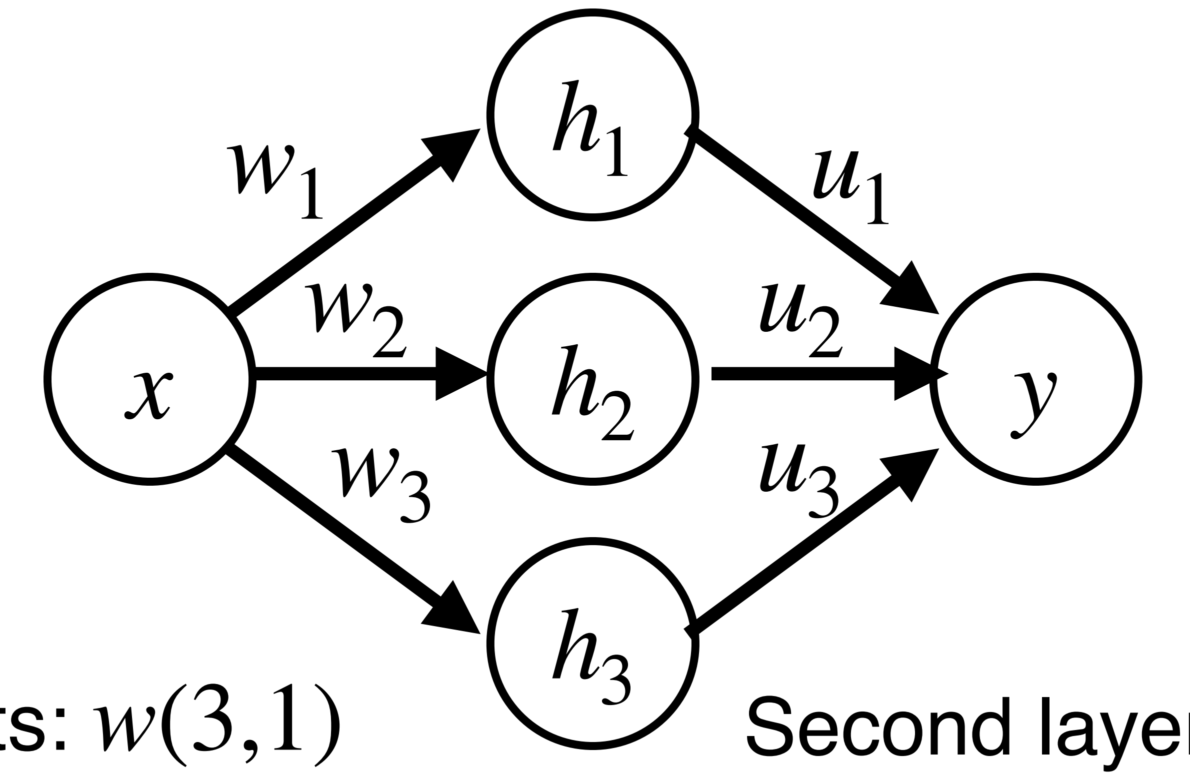
Second layer weights: $u(3,1)$

First layer bias: $p(1,)$

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: $u(3,1)$

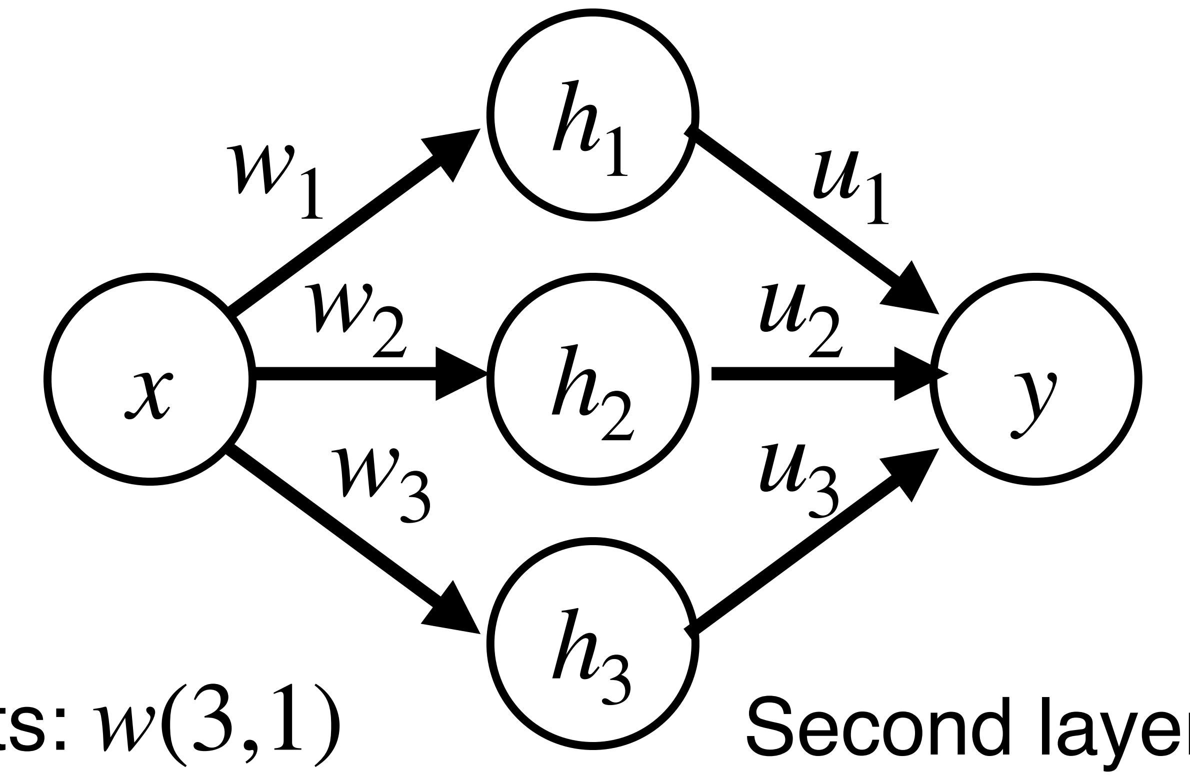
First layer bias: $p(1,)$

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

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

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

First layer bias: $p(1,)$

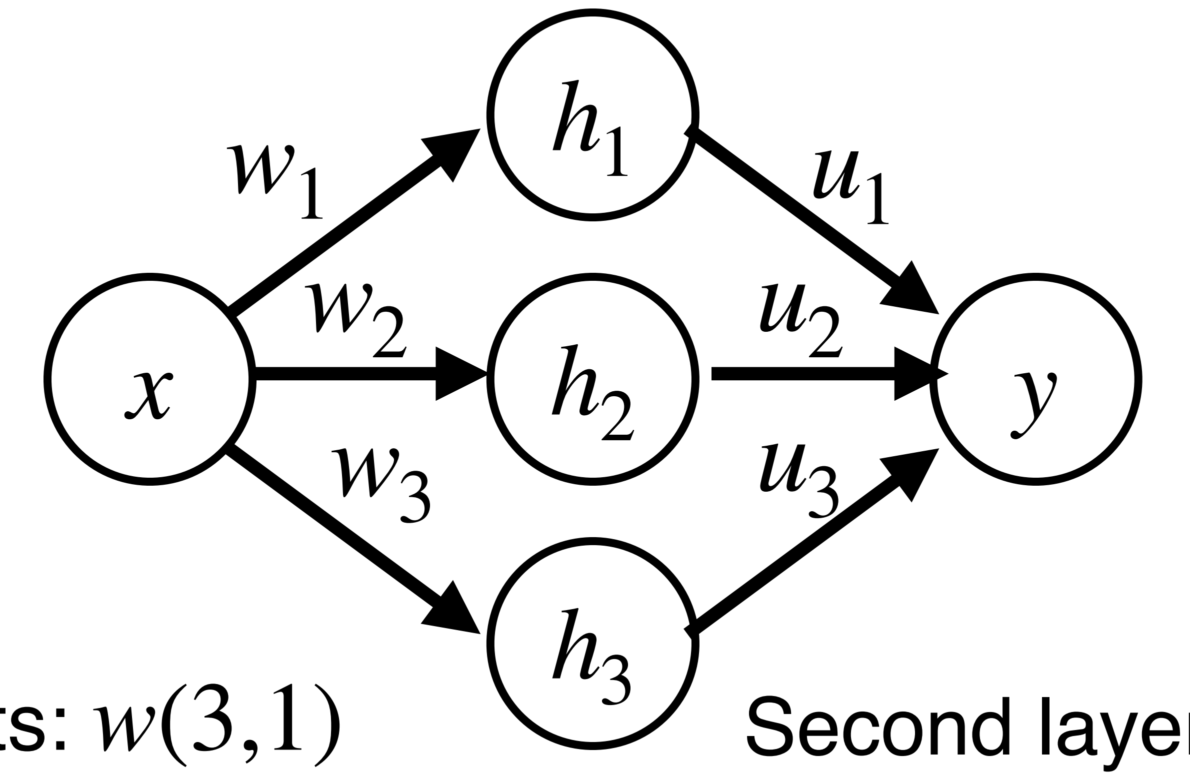
$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

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

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: $u(3,1)$

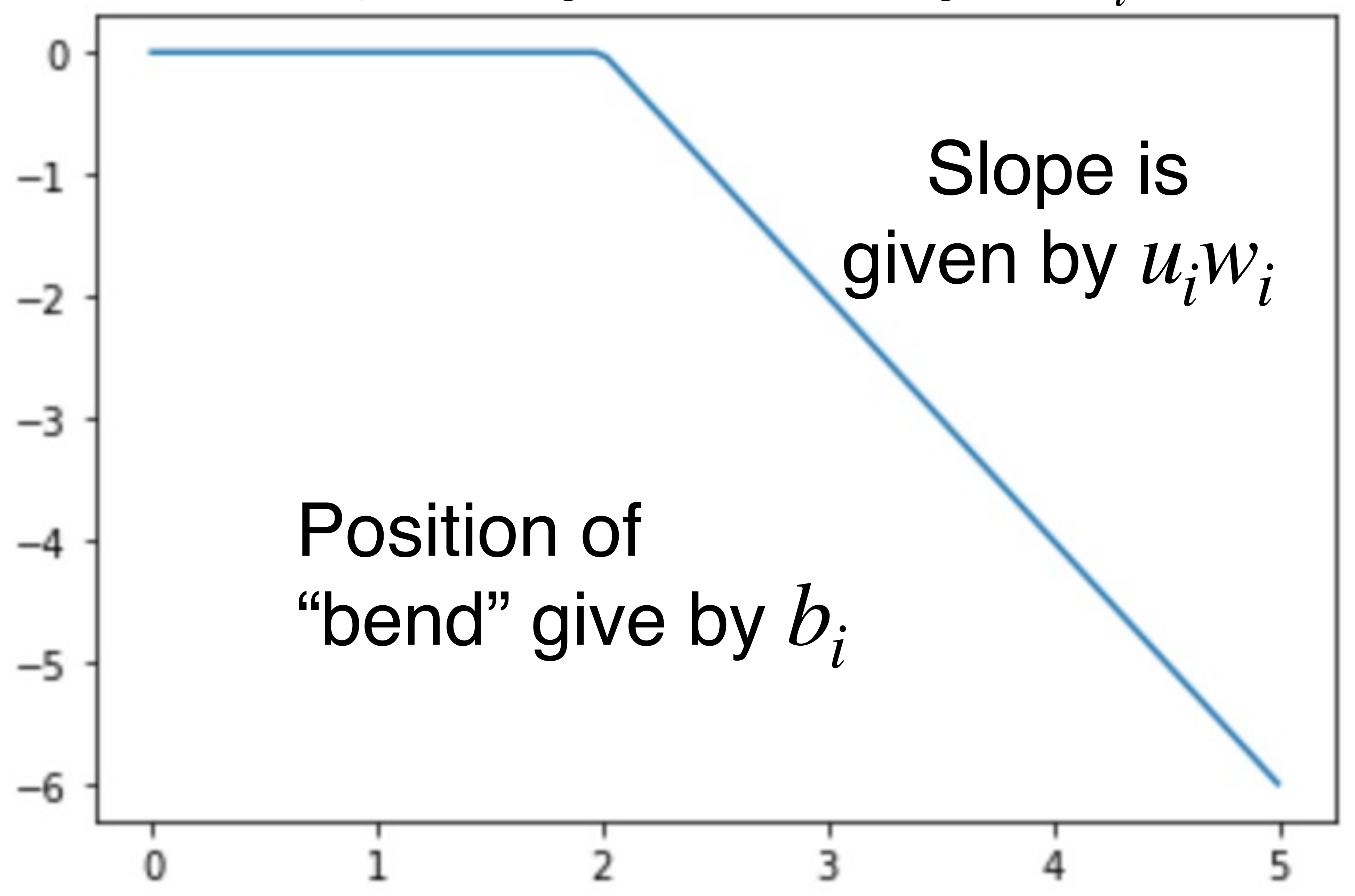
First layer bias: $p(1,)$

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

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

Output is a sum of shifted, scaled ReLUs:

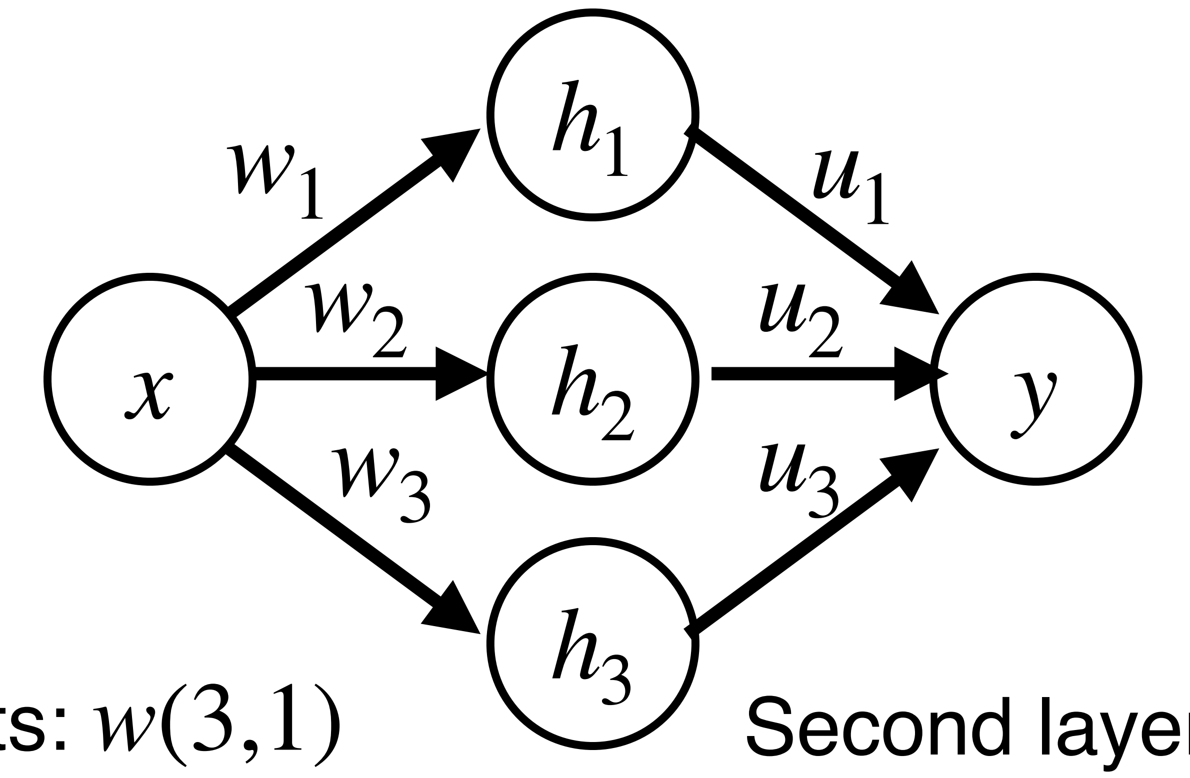
Flip left / right based on sign of w_i



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: $b(3,)$

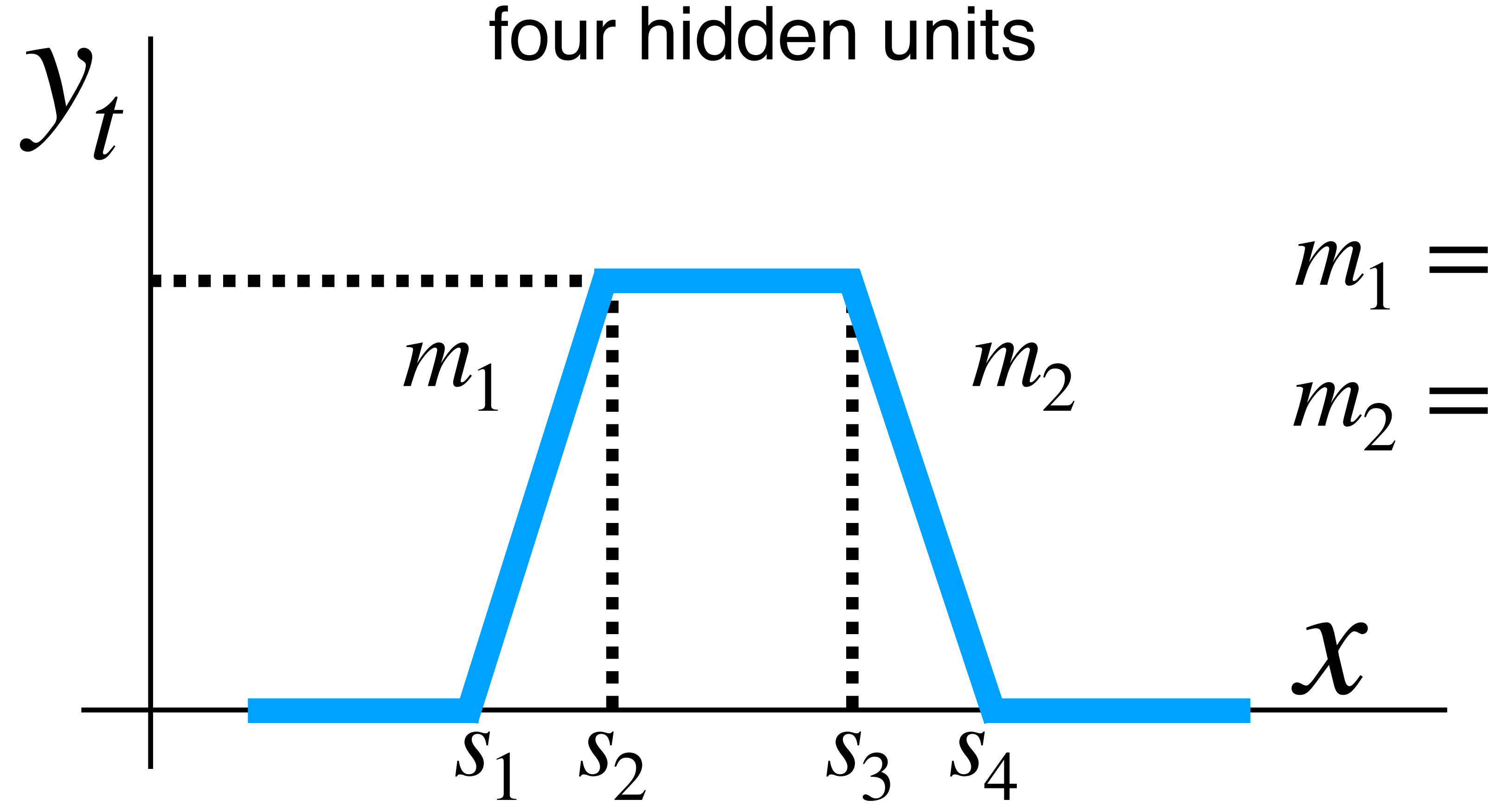
Second layer weights: $u(3,1)$

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

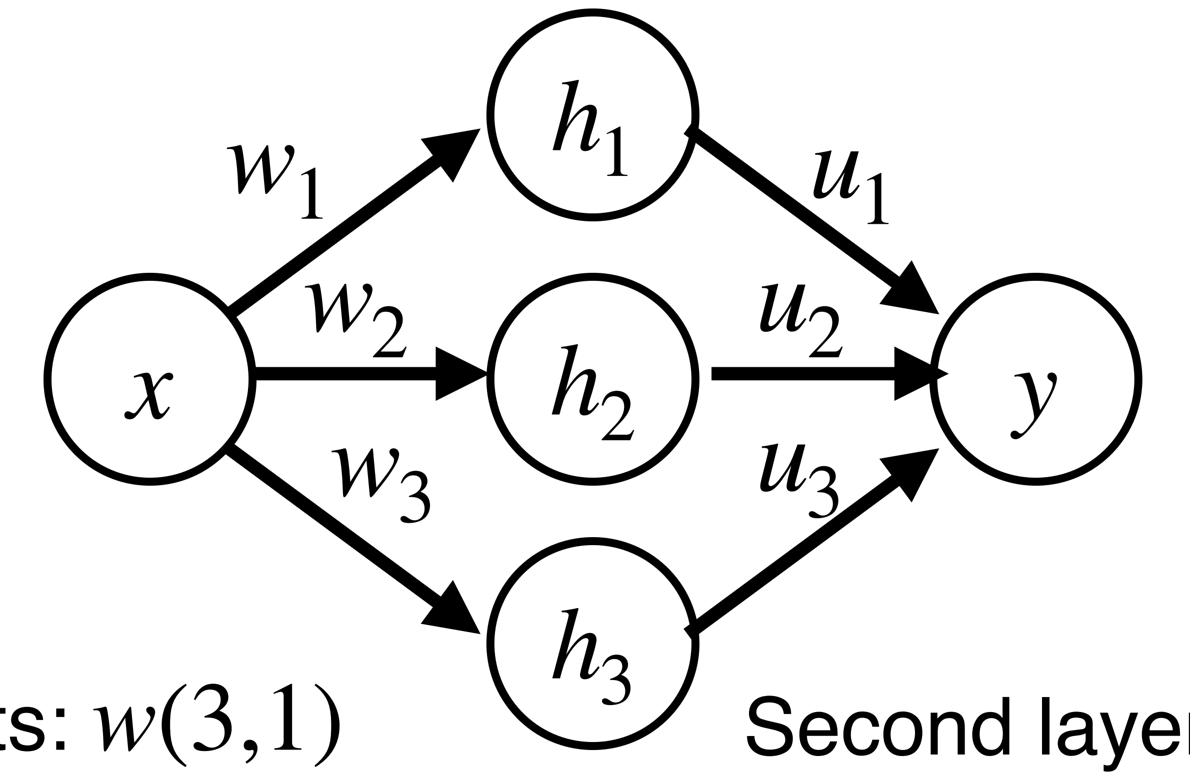


$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$

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: $b(3,)$

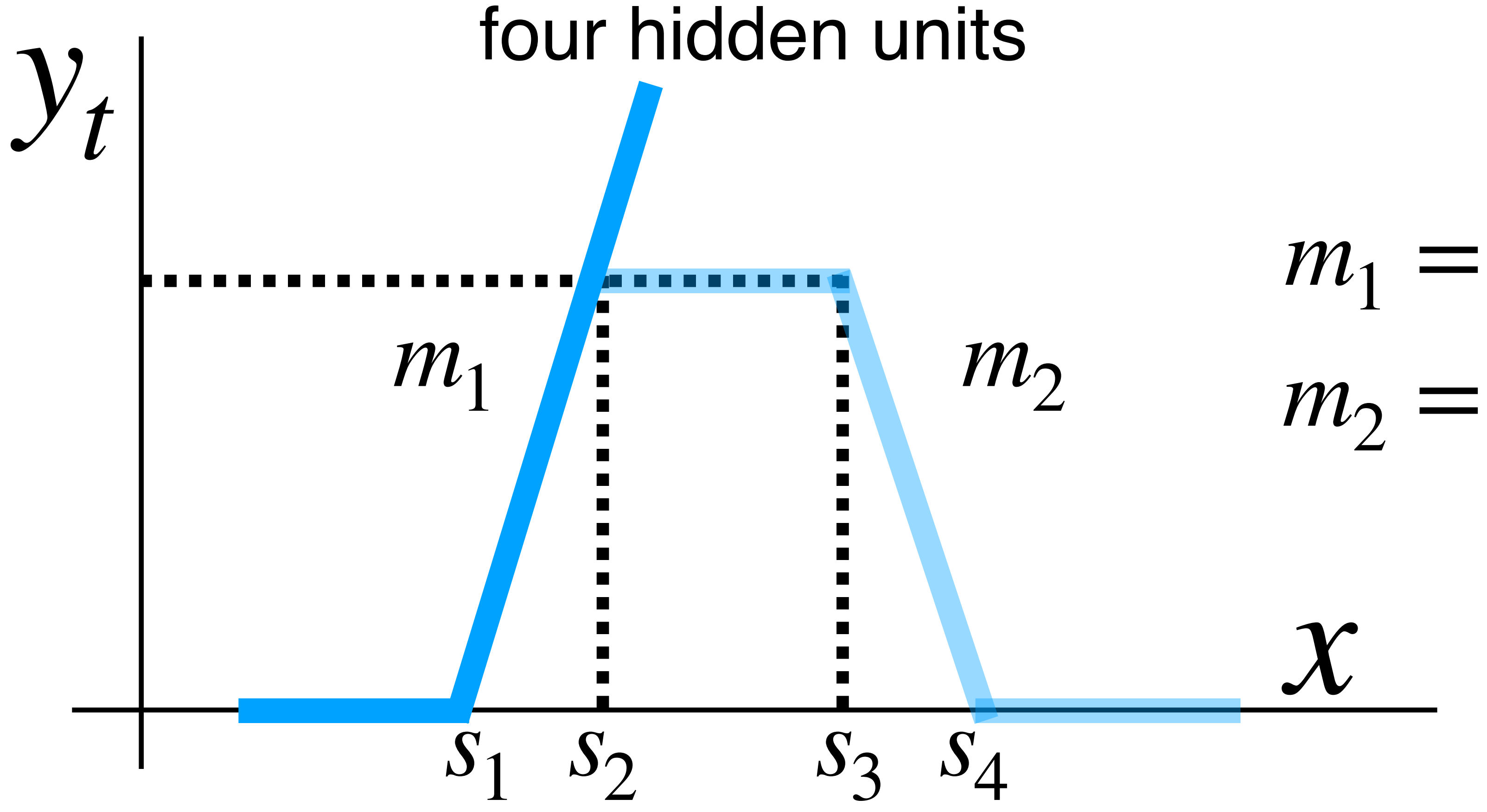
Second layer weights: $u(3,1)$

First layer bias: $p(1,)$

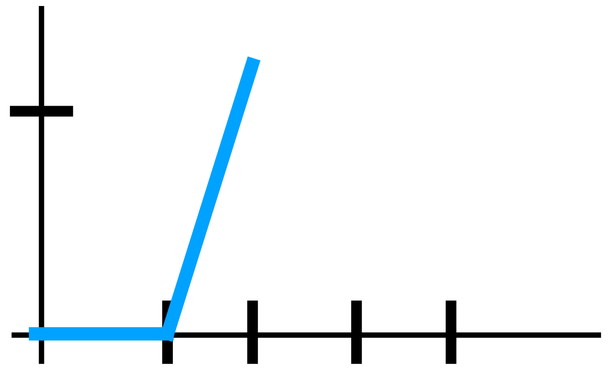
$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

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

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



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

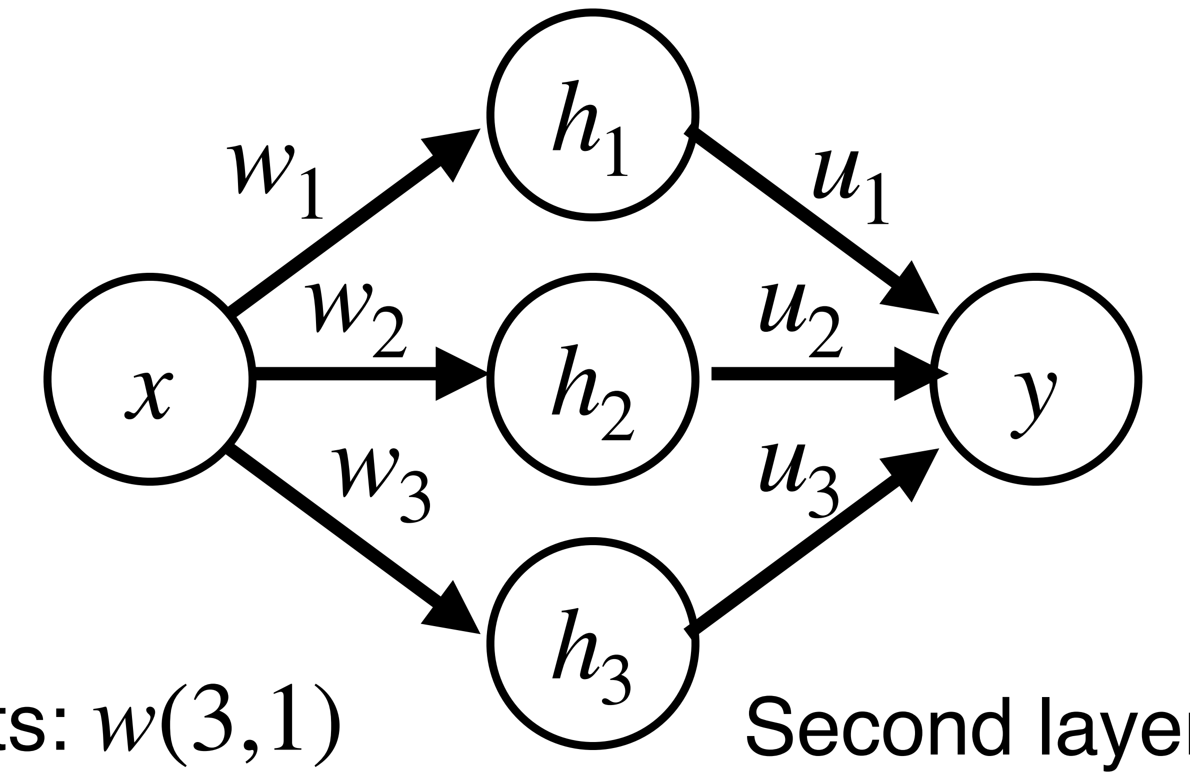


$m_1 \max(0, x - s_1)$

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

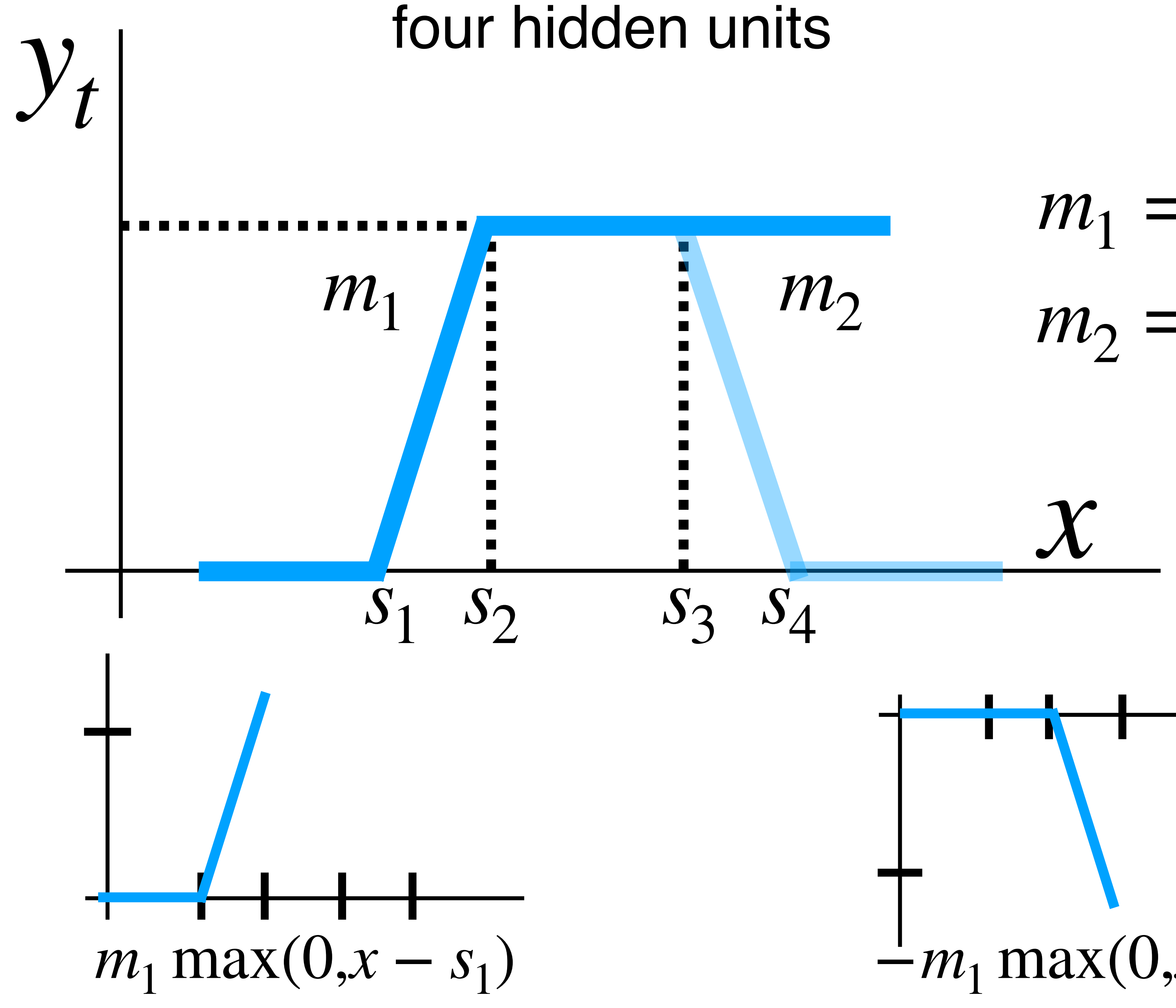
Second layer weights: $u(3,1)$

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

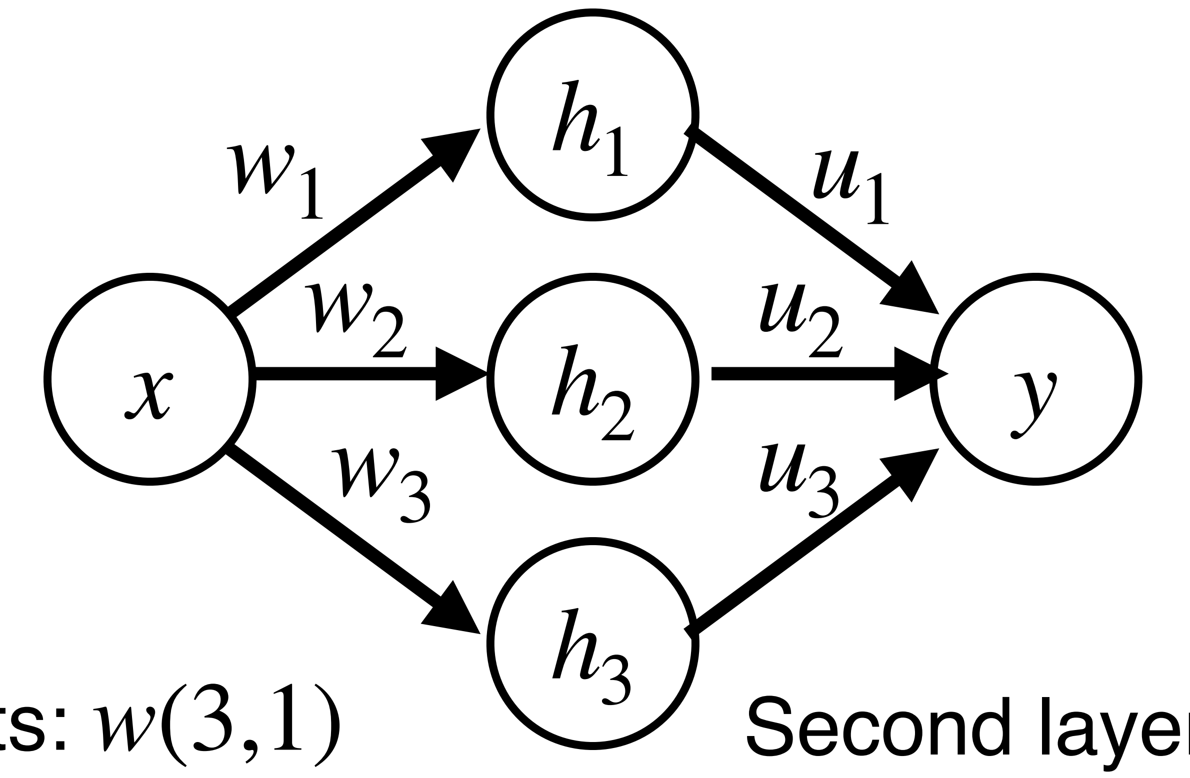


$$m_1 = t / (s_2 - s_1)$$
$$m_2 = t / (s_4 - s_3)$$

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: $b(3,)$

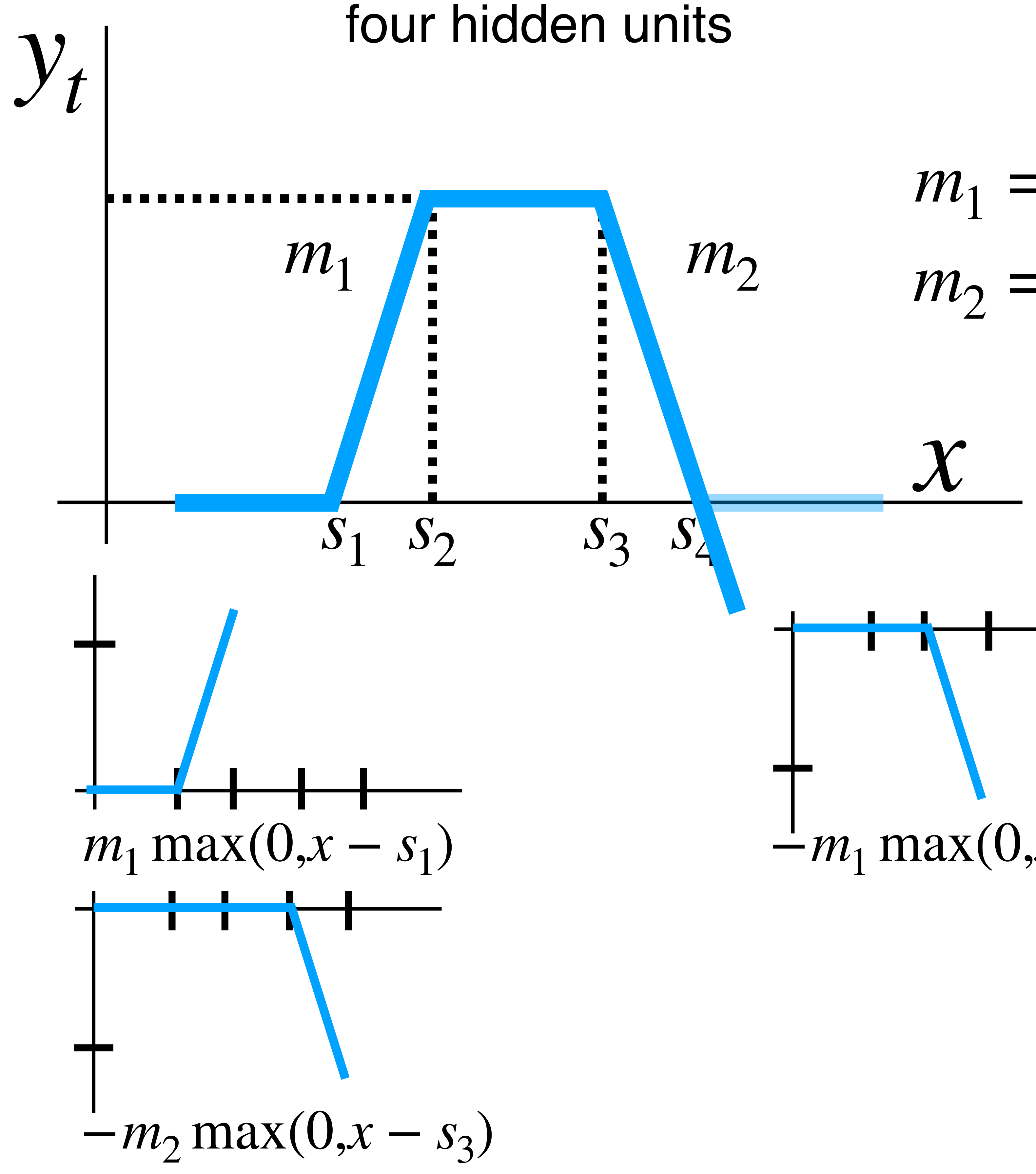
Second layer weights: $u(3,1)$

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

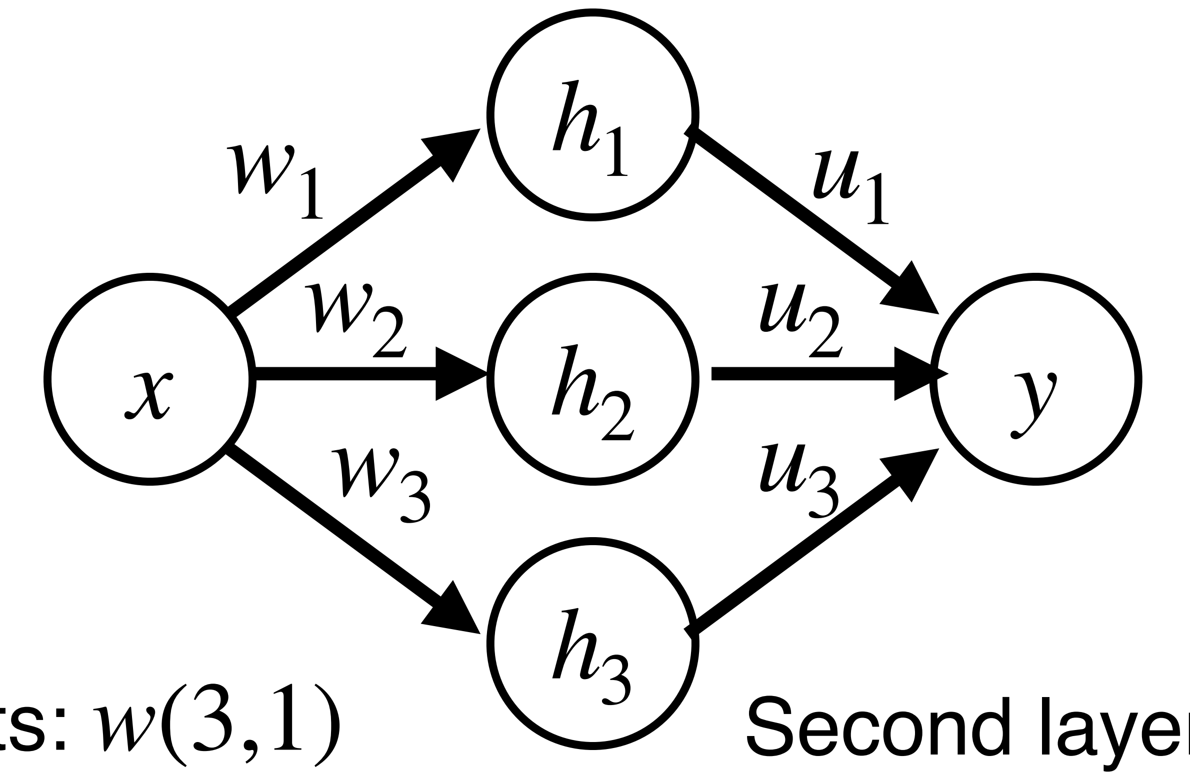


$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$

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: $b(3,)$

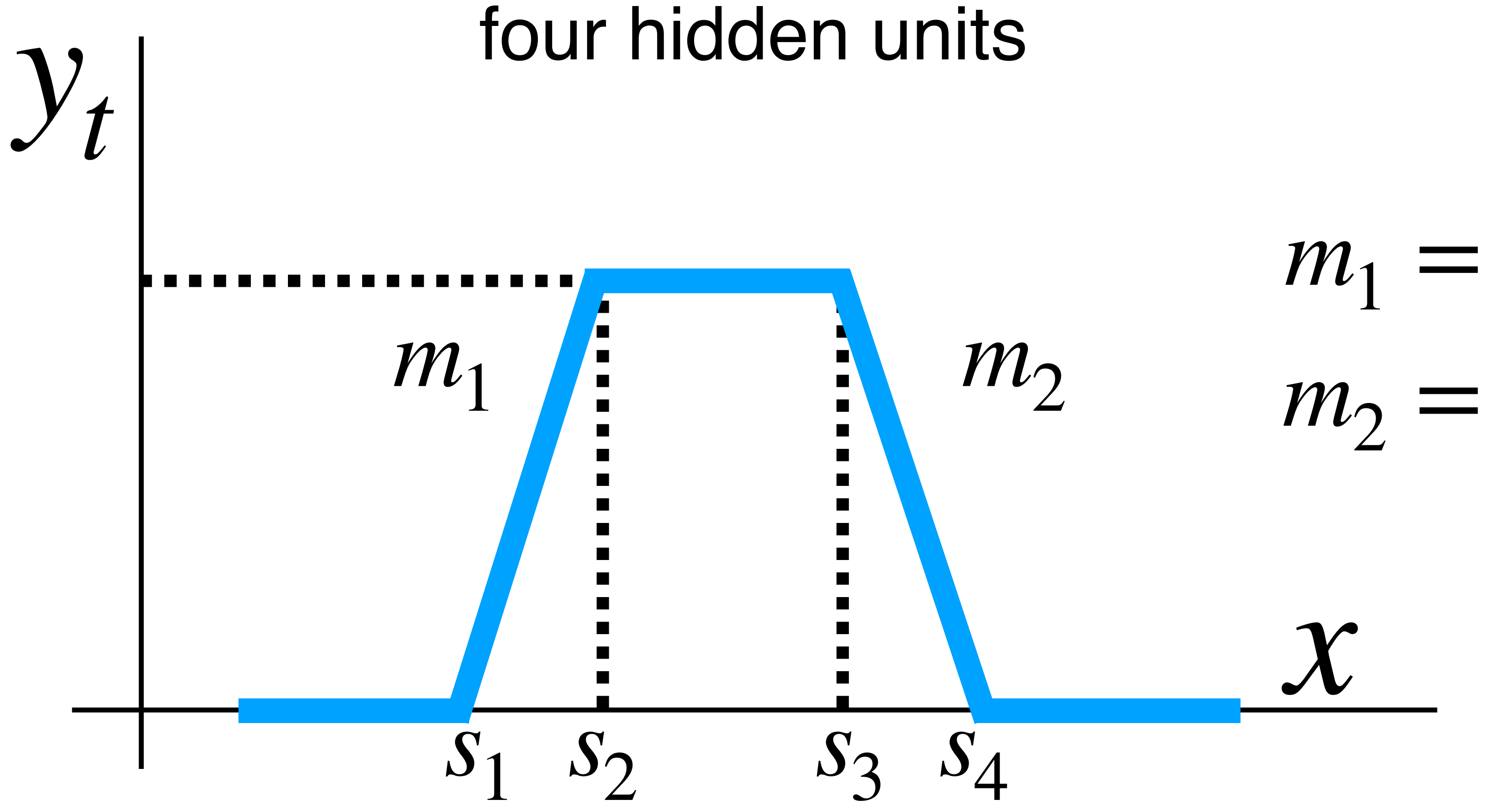
Second layer weights: $u(3,1)$

First layer bias: $p(1,)$

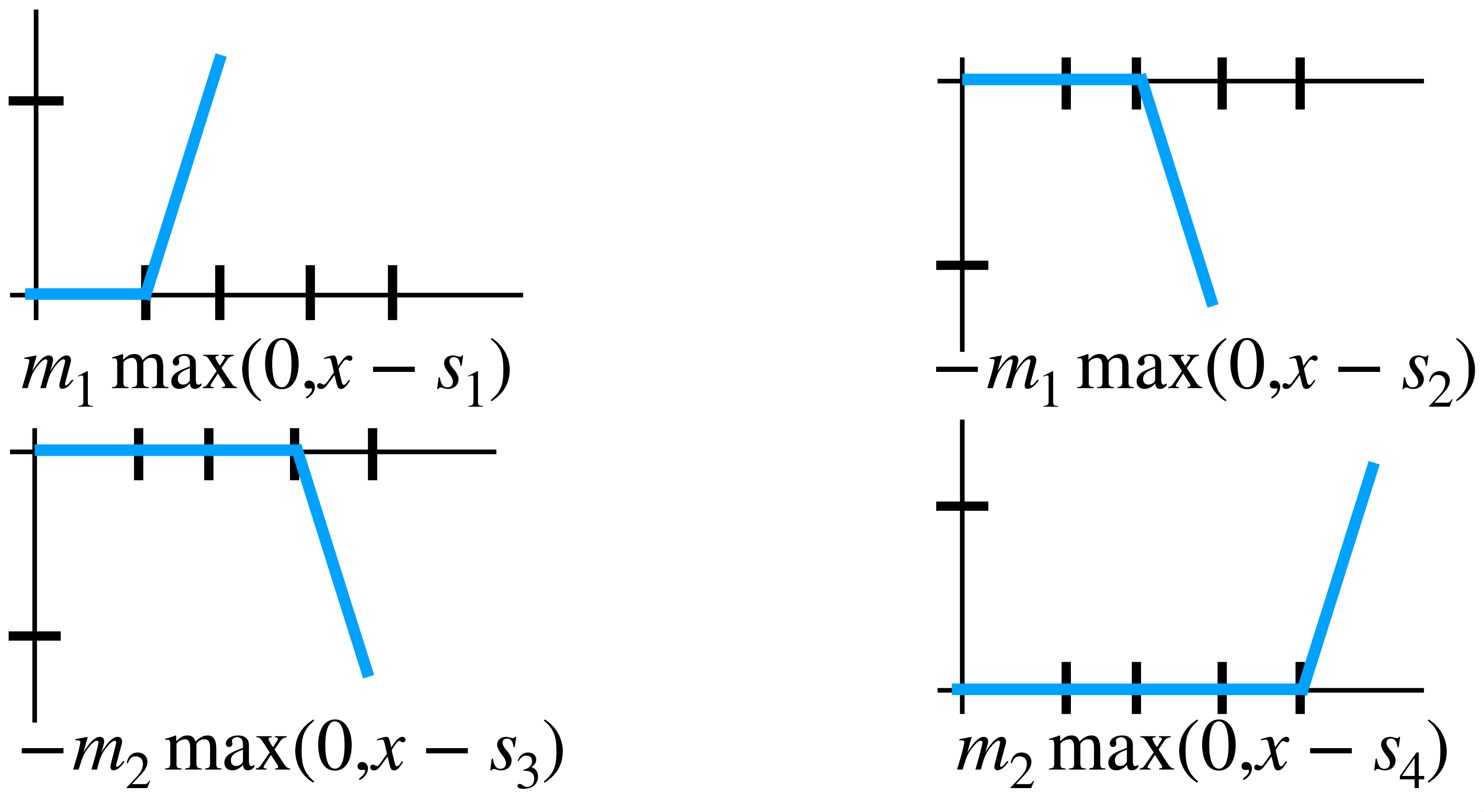
$$h_1 = \max(0, w_1 x + b_1)$$
$$h_2 = \max(0, w_2 x + b_2)$$
$$h_1 = \max(0, w_3 x + b_3)$$
$$y = u_1 h_1 + u_2 h_2 + u_3 h_3 + p$$

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

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

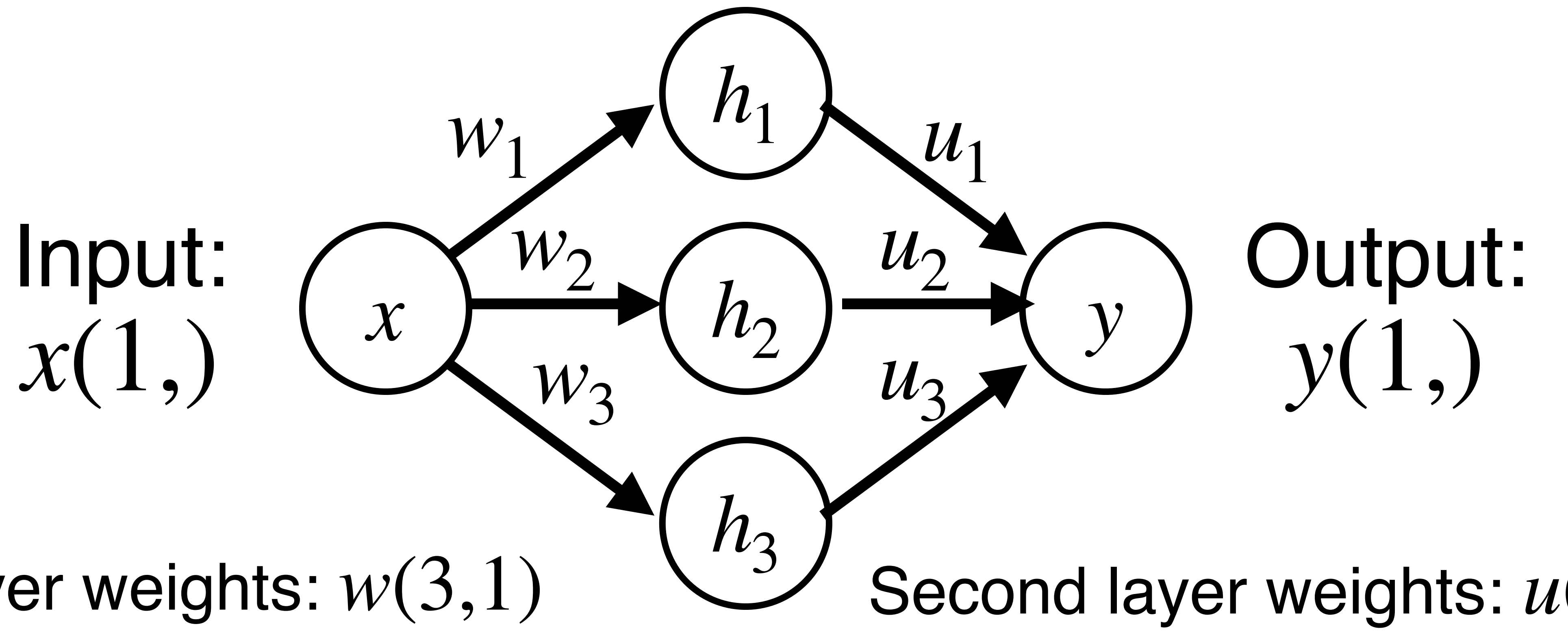


$$m_1 = t/(s_2 - s_1)$$
$$m_2 = t/(s_4 - s_3)$$



Universal Approximation

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



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

First layer bias: $b(3,)$

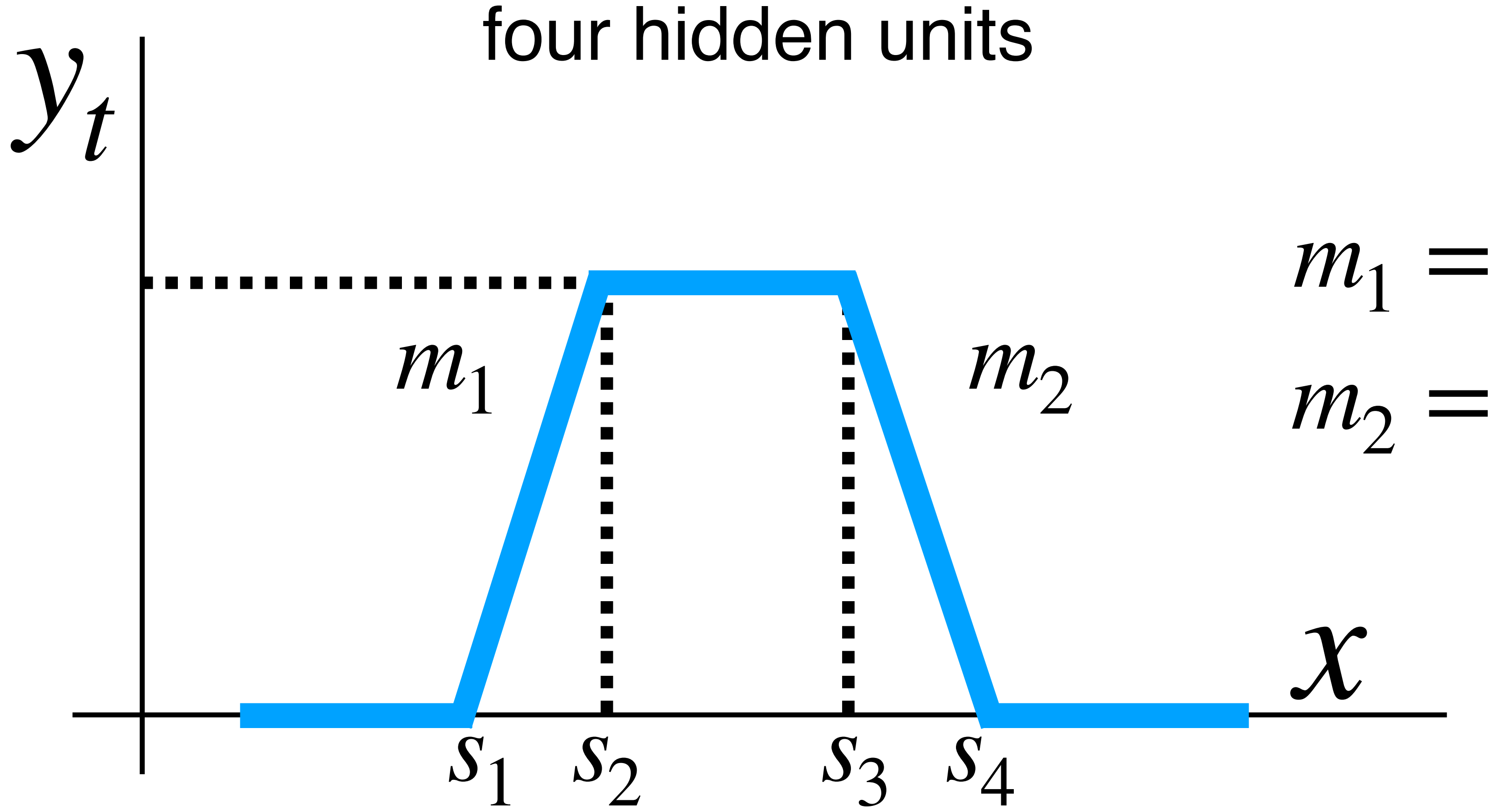
Second layer weights: $u(3,1)$

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



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

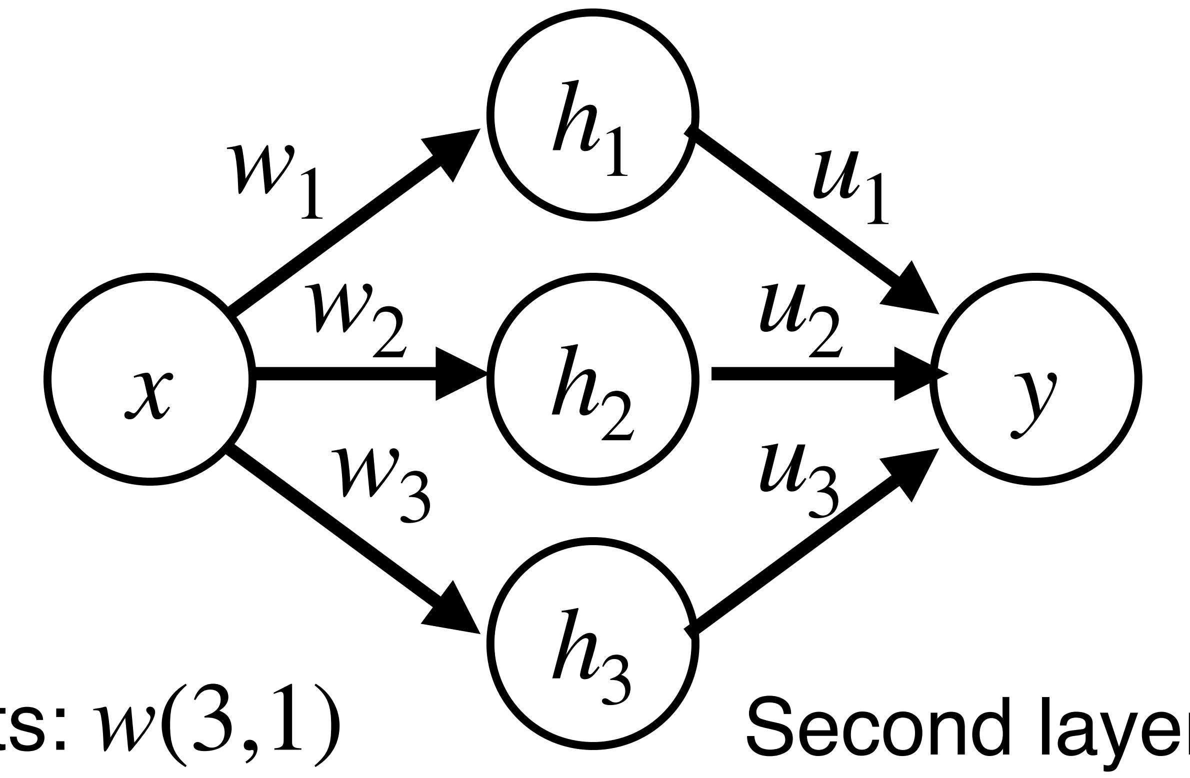
With 4K hidden units we can build a sum of K bumps

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: $b(3,)$

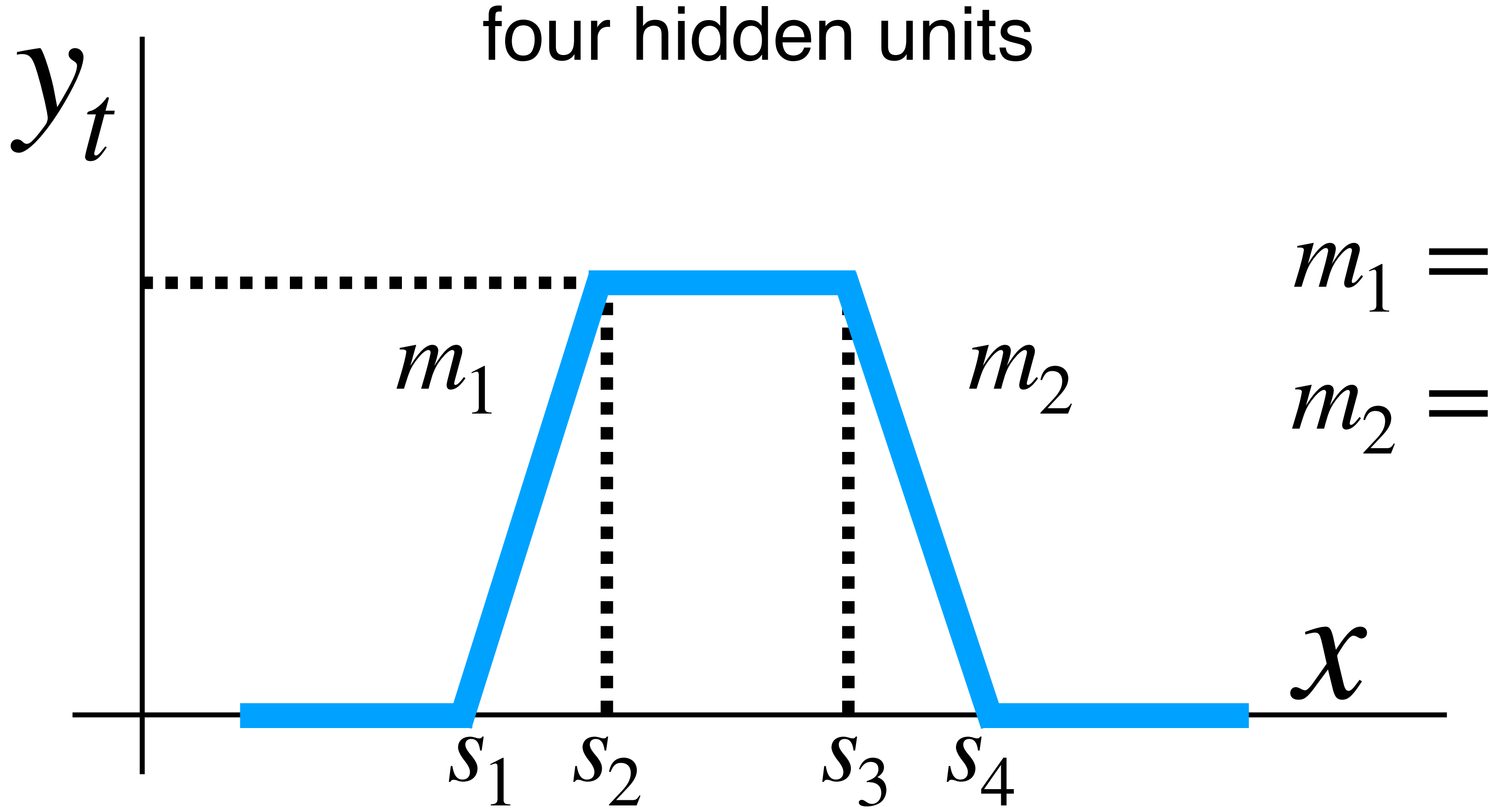
Second layer weights: $u(3,1)$

First layer bias: $p(1,)$

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

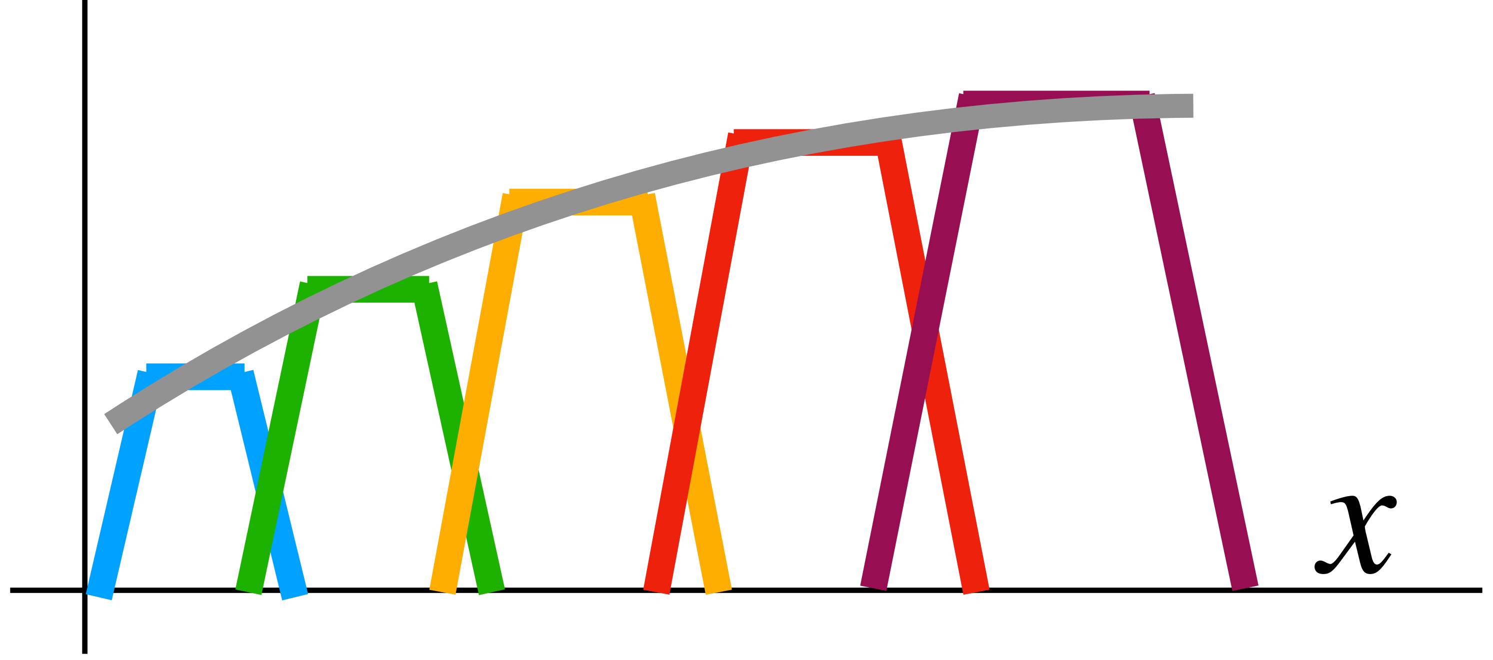
$$y = u_1 \max(0, w_1 x + b_1)$$
$$+ u_2 \max(0, w_2 x + b_2)$$
$$+ u_3 \max(0, w_3 x + b_3)$$
$$+ p$$

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

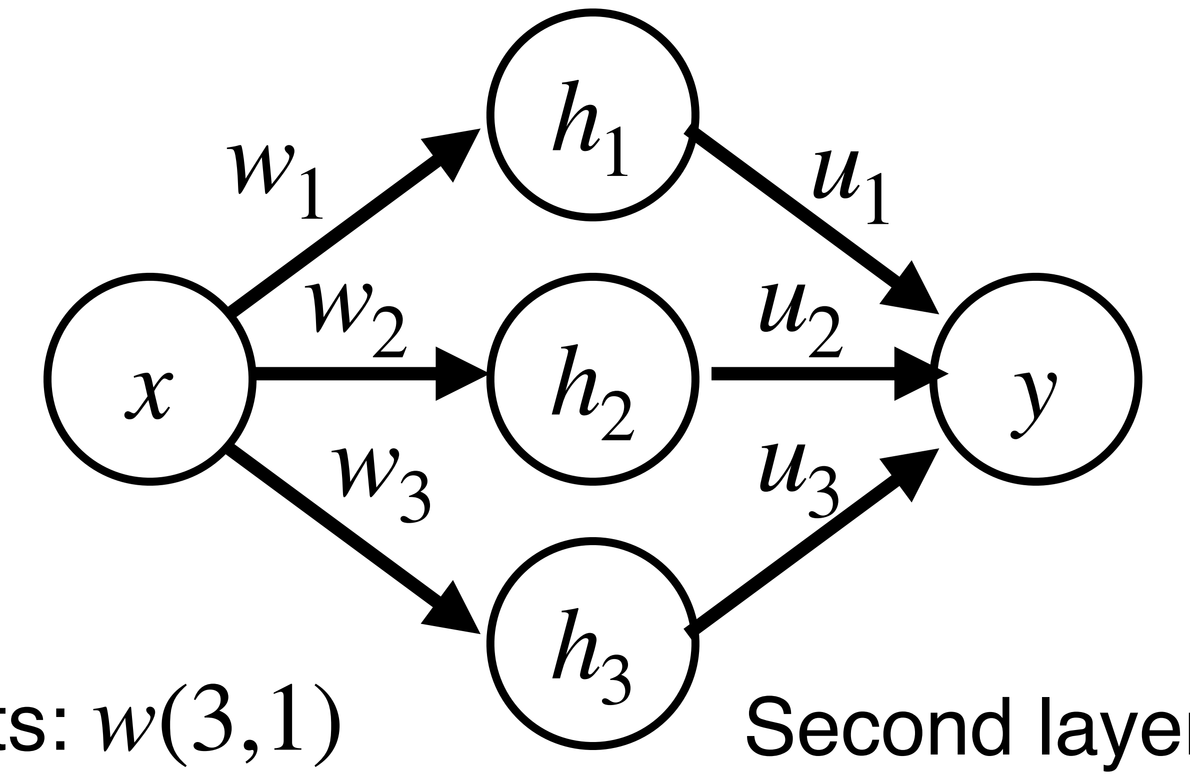


Approximate functions with bumps!

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: $u(3,1)$

First layer bias: $p(1,)$

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

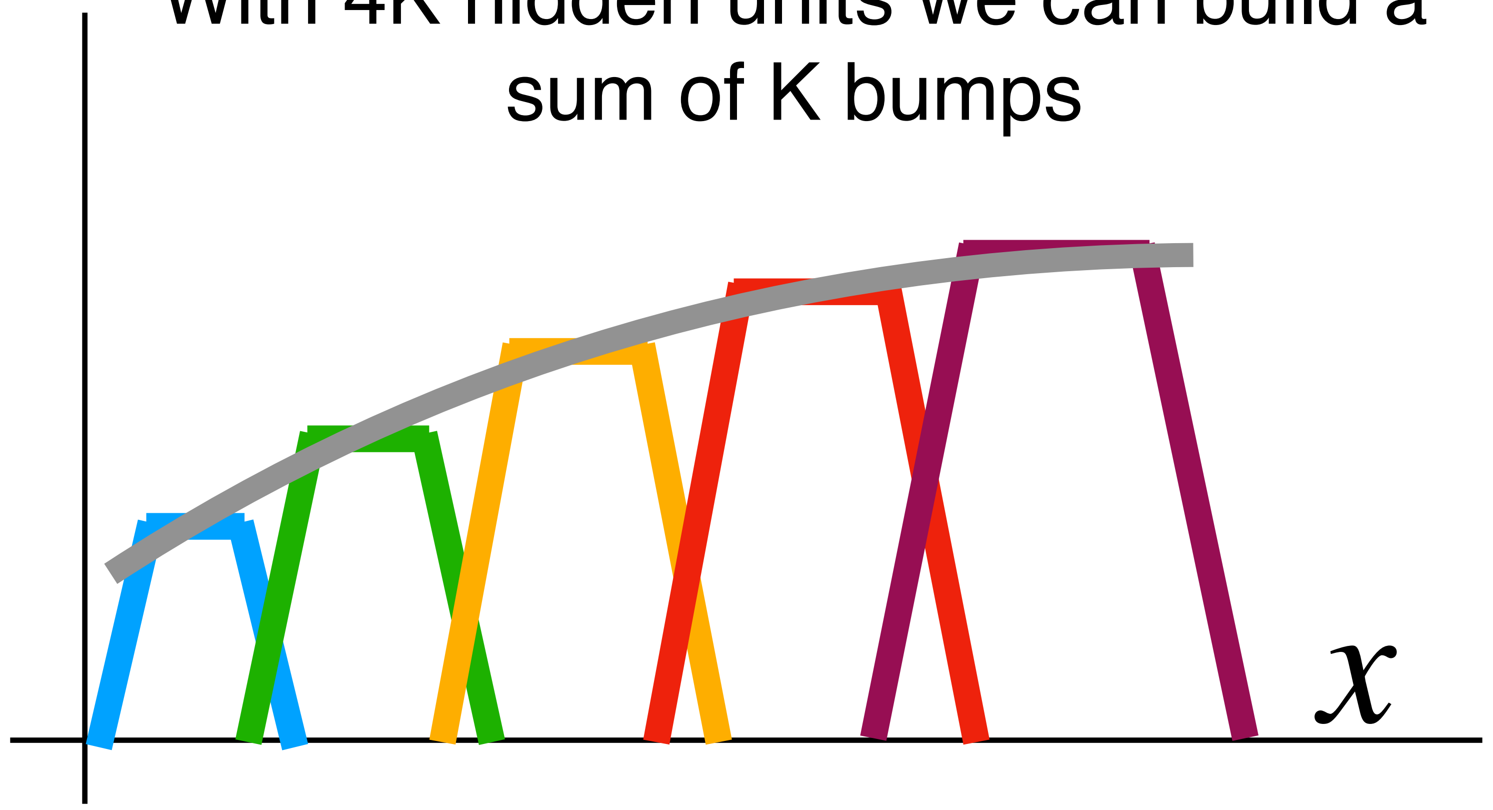
$$\begin{aligned} y = {} & u_1 \max(0, w_1 x + b_1) \\ & + u_2 \max(0, w_2 x + b_2) \\ & + u_3 \max(0, w_3 x + b_3) \\ & + p \end{aligned}$$

What about …

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

See [Nielsen, Chapter 4]

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

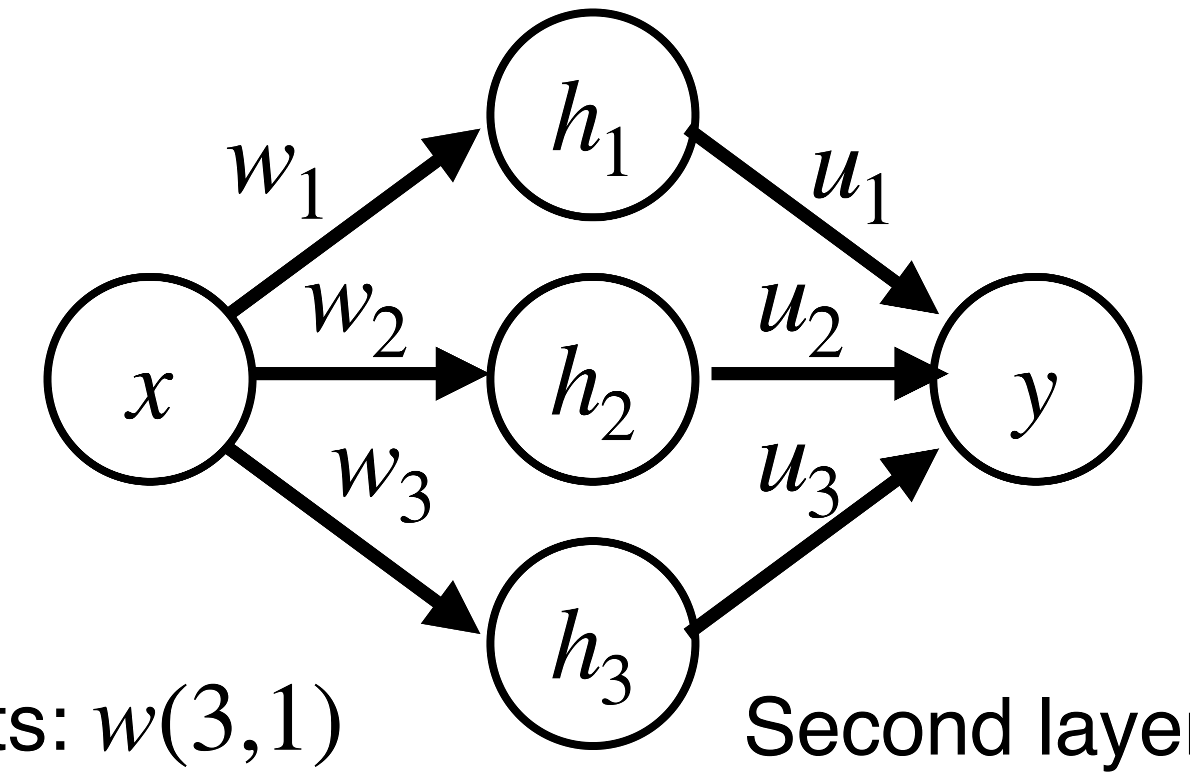


Approximate functions with bumps!

Universal Approximation

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

Input: $x(1,)$



Output: $y(1,)$

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

First layer bias: $b(3,)$

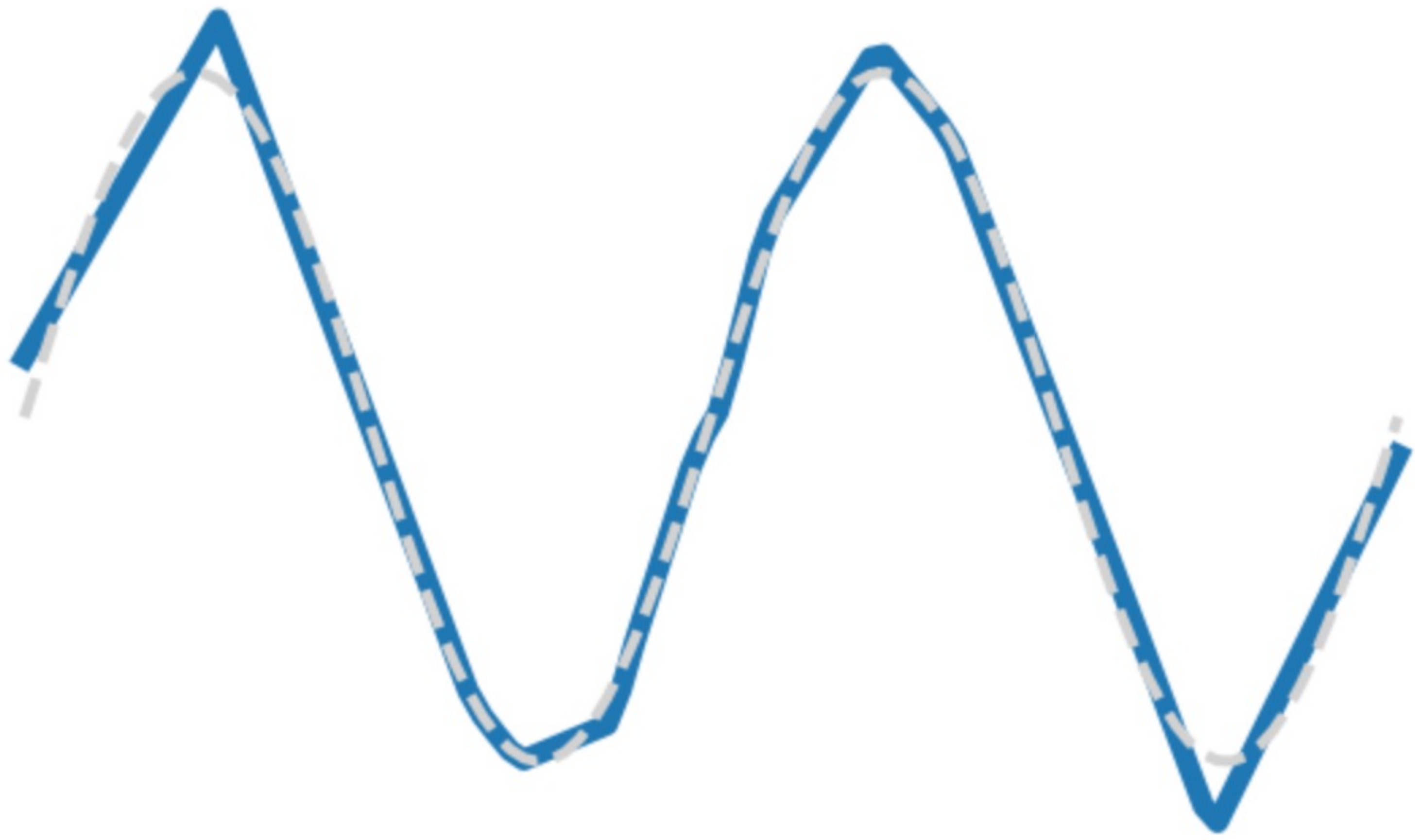
Second layer weights: $u(3,1)$

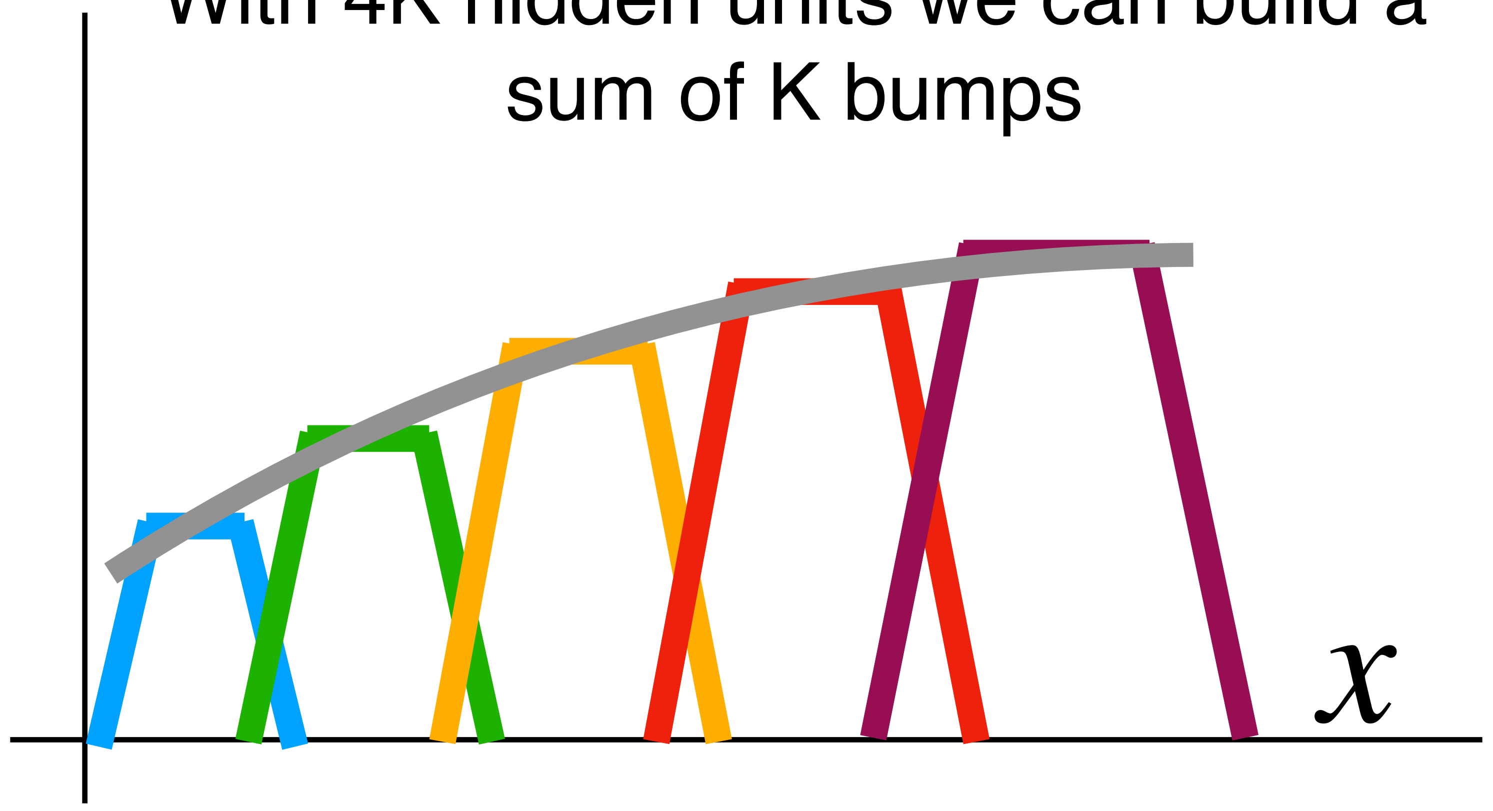
First layer bias: $p(1,)$

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

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

Reality check: Networks don't really learn bumps!

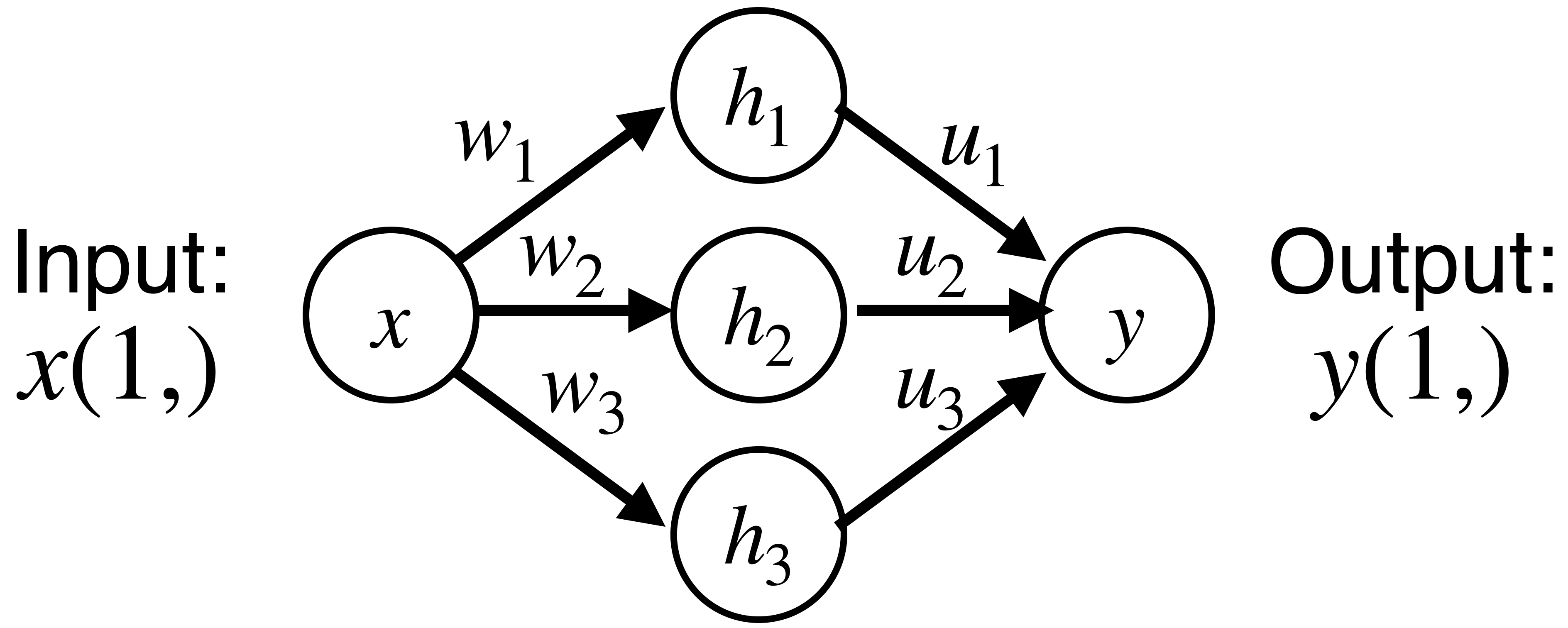




Approximate functions with bumps!

Universal Approximation

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



Universal approximation tells us:

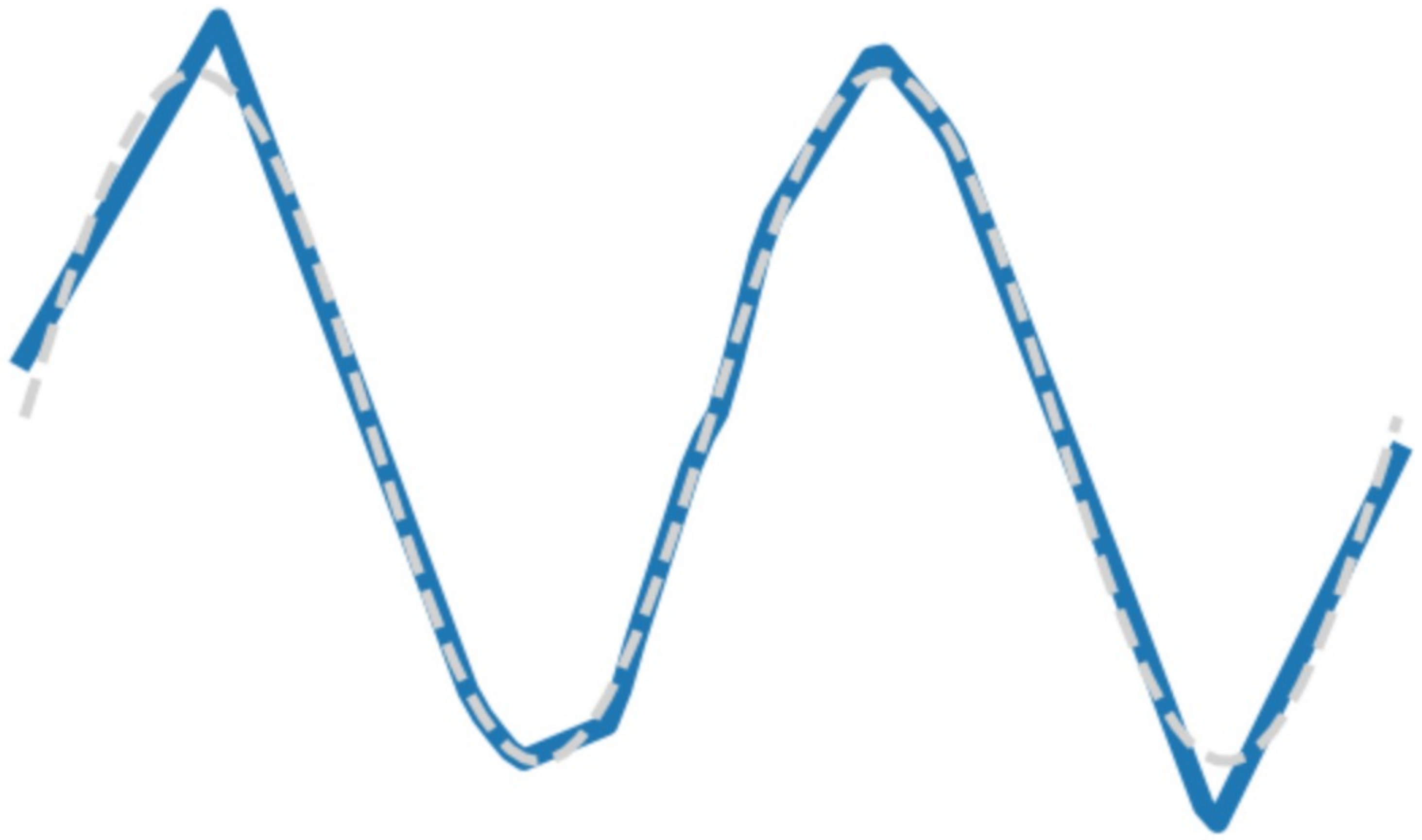
- Neural nets can represent any function

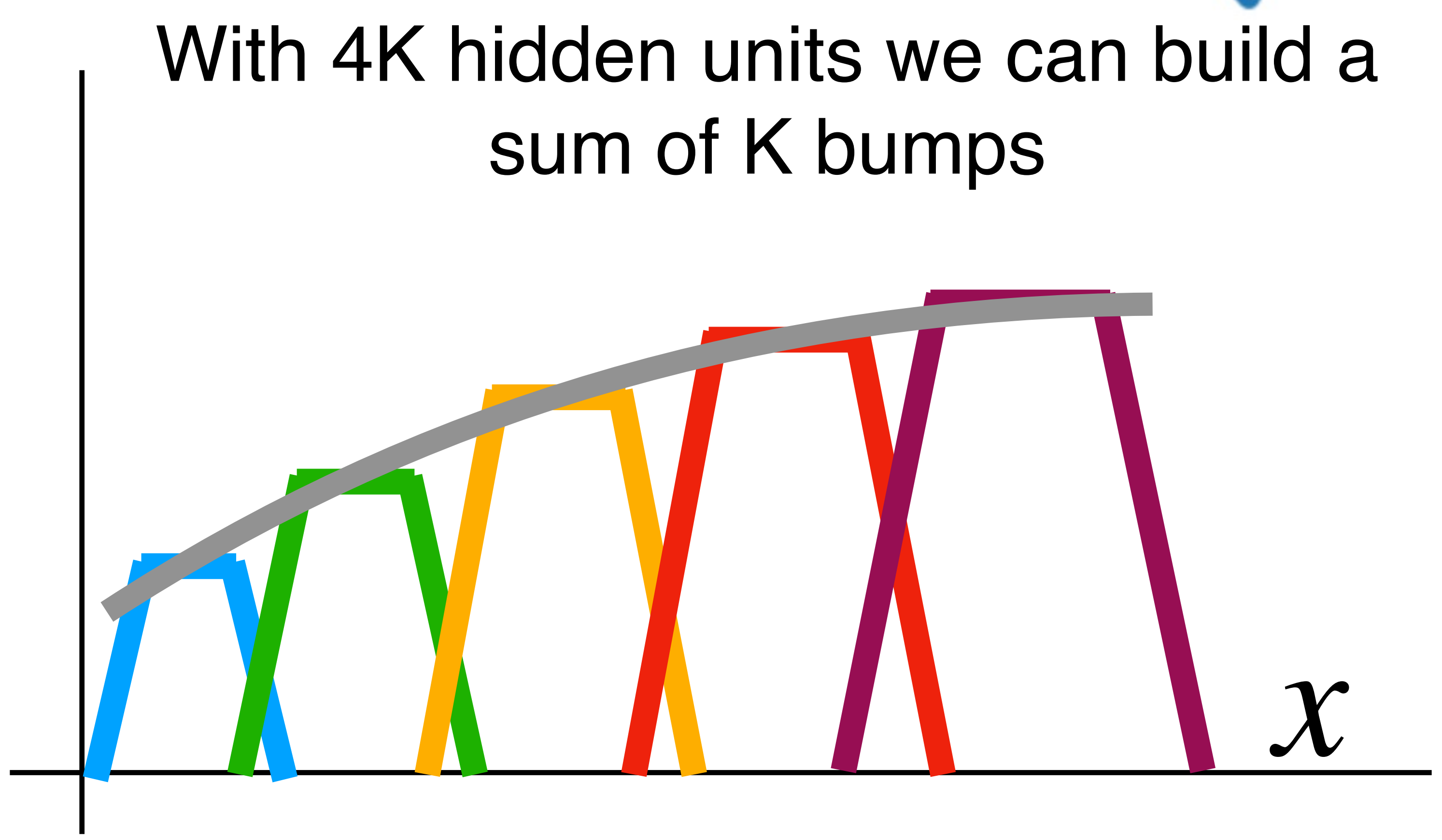
Universal approximation DOES NOT tell us:

- 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!





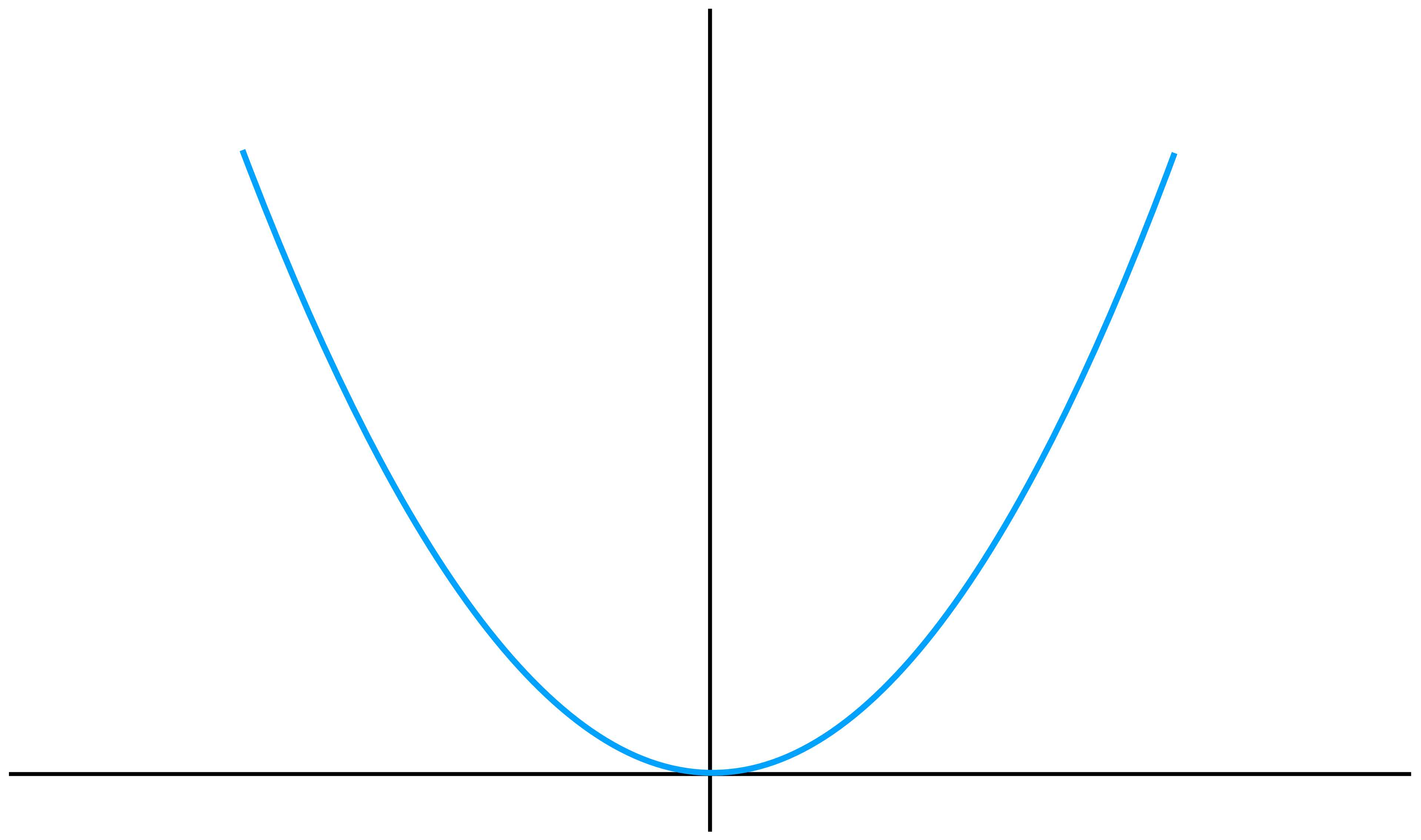
Approximate functions with bumps!

Convex Functions

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

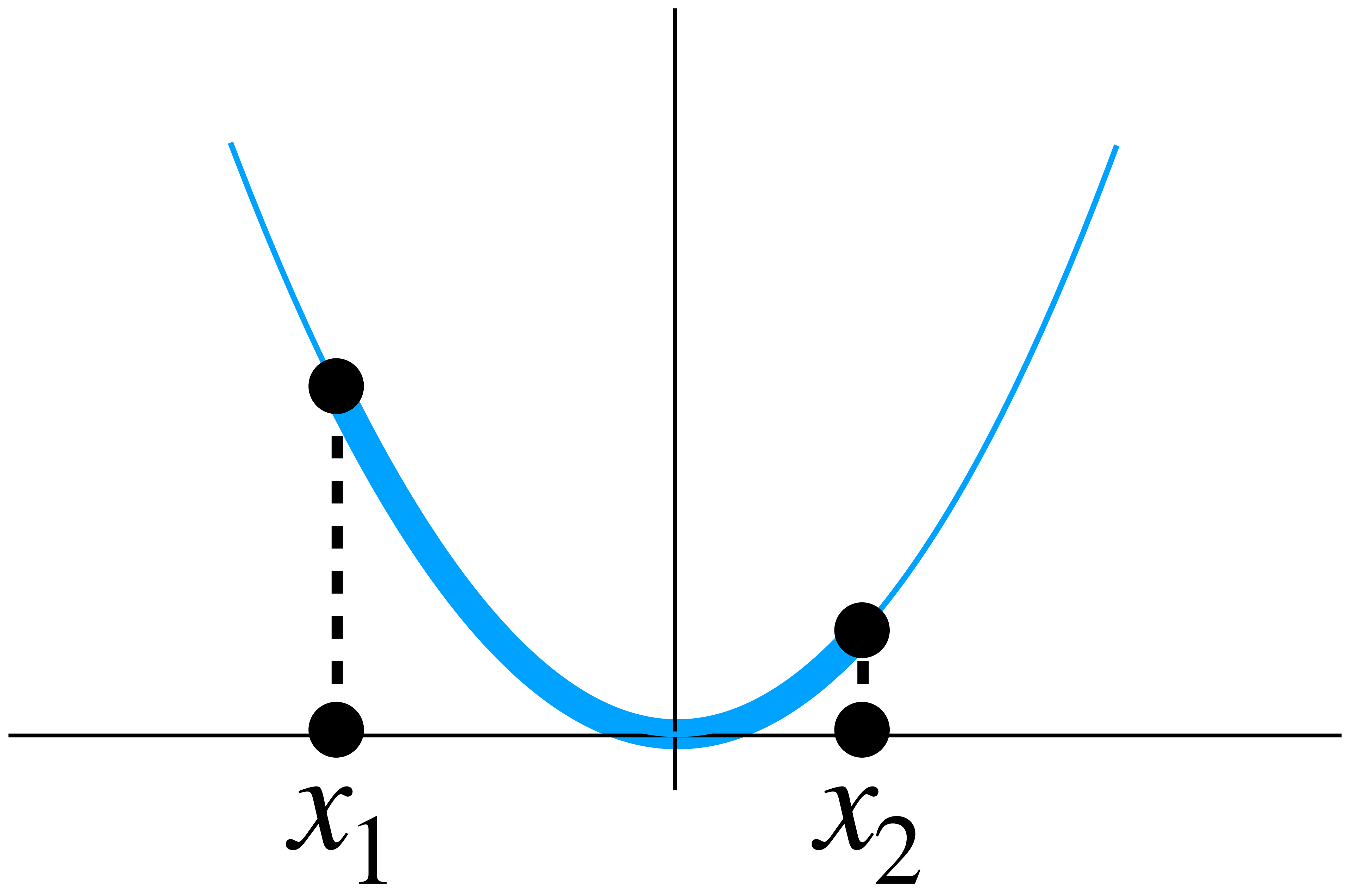


Convex Functions

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:

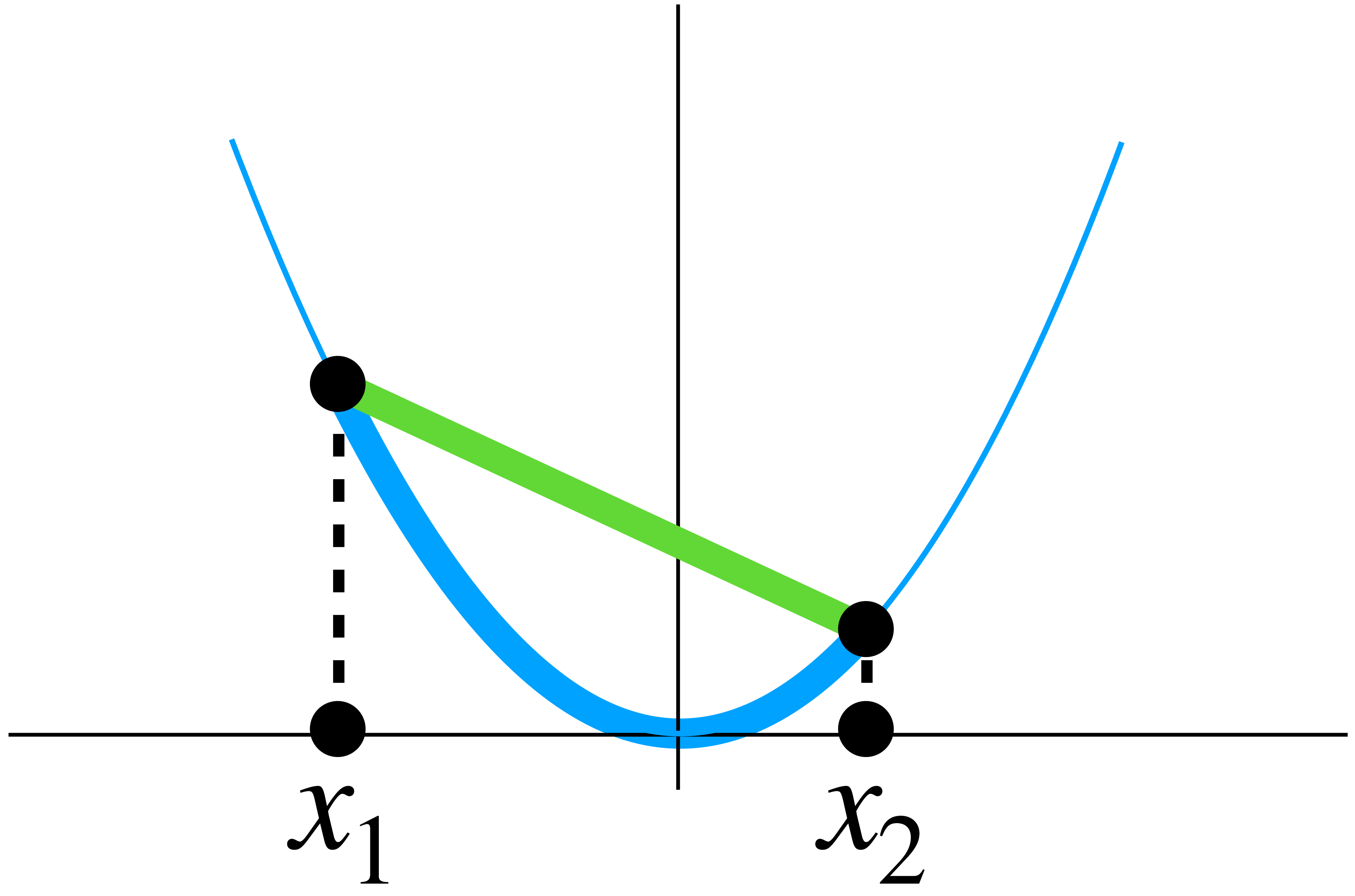


Convex Functions

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:

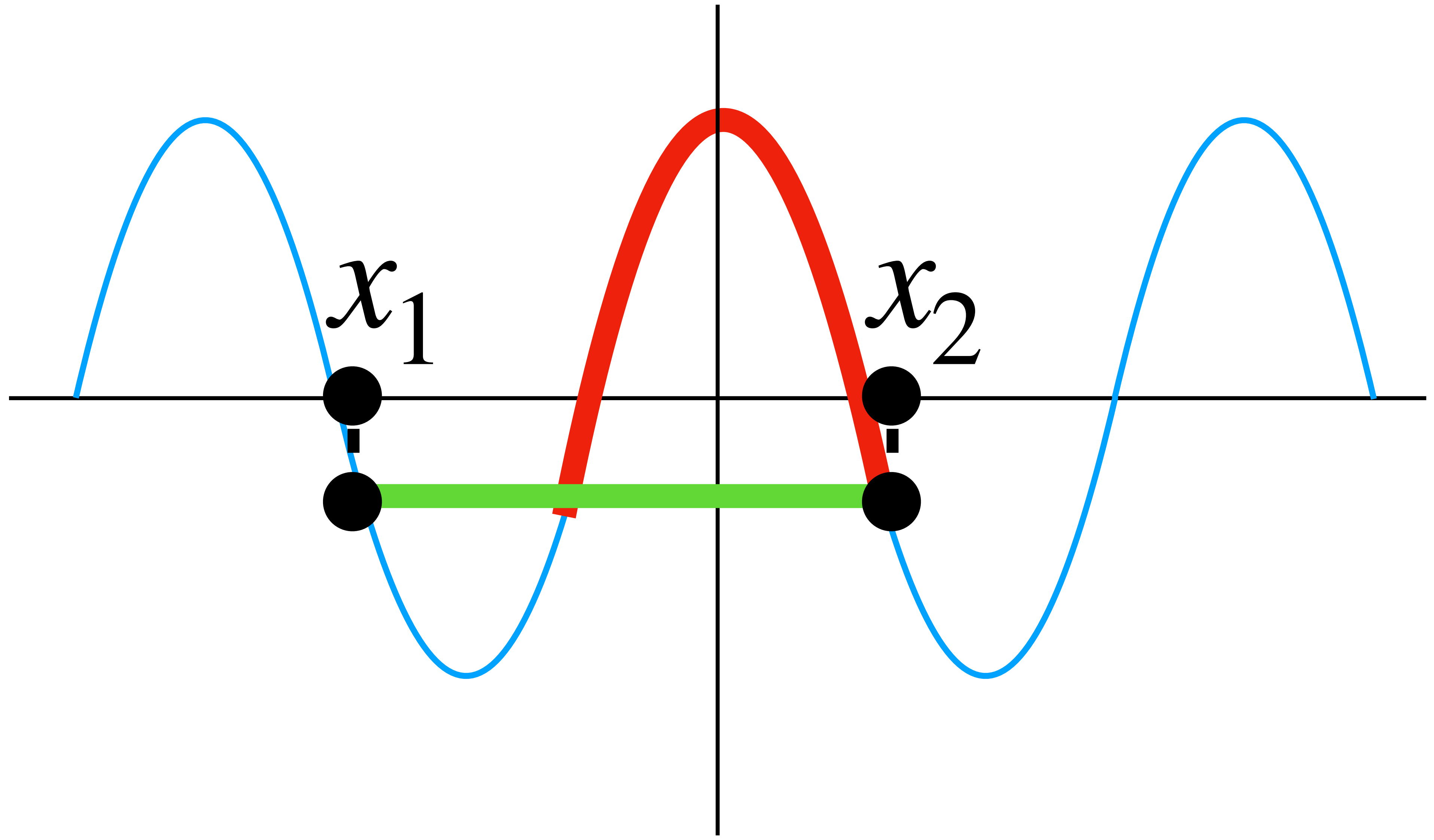


Convex Functions

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:



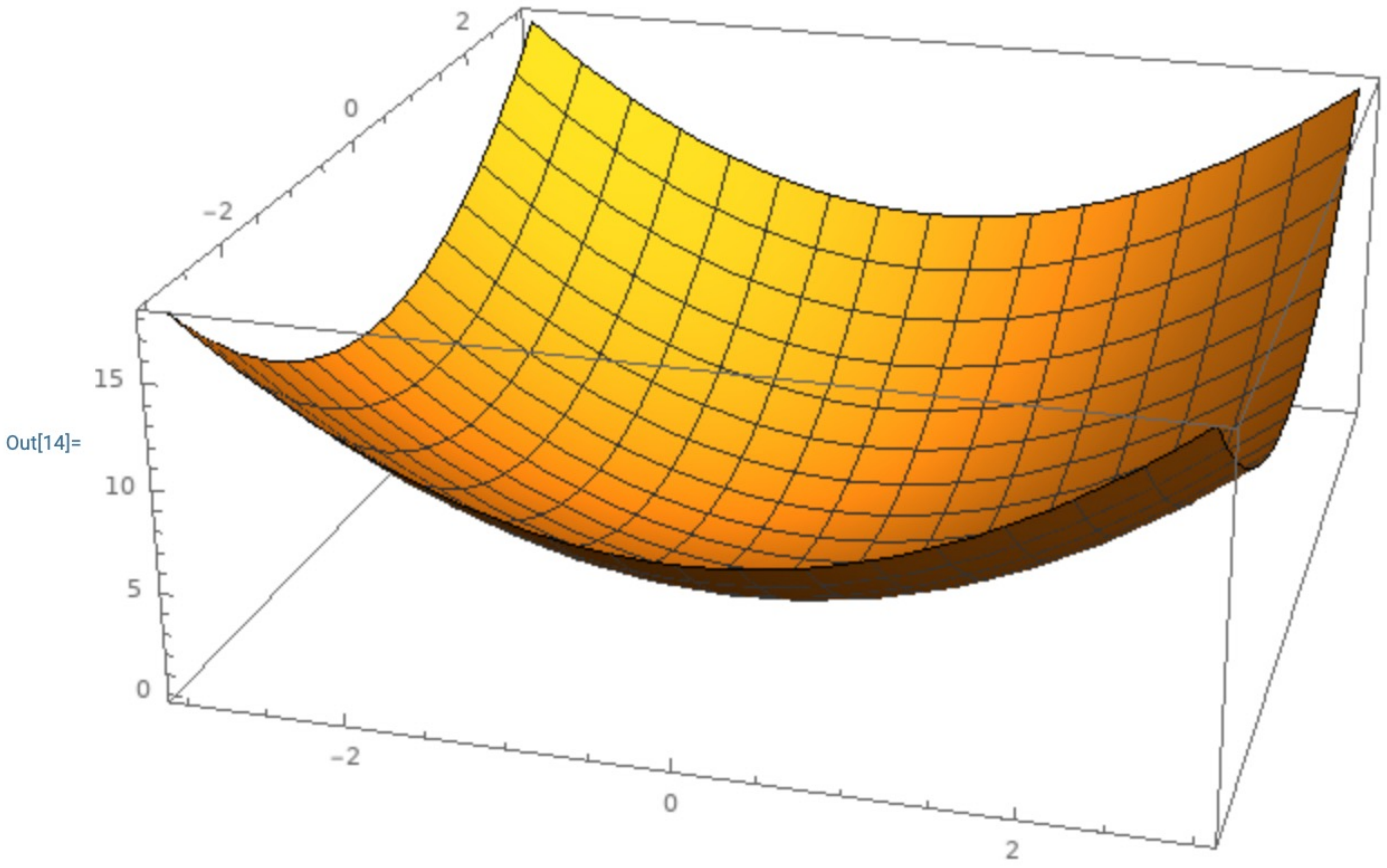
Convex Functions

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



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***

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

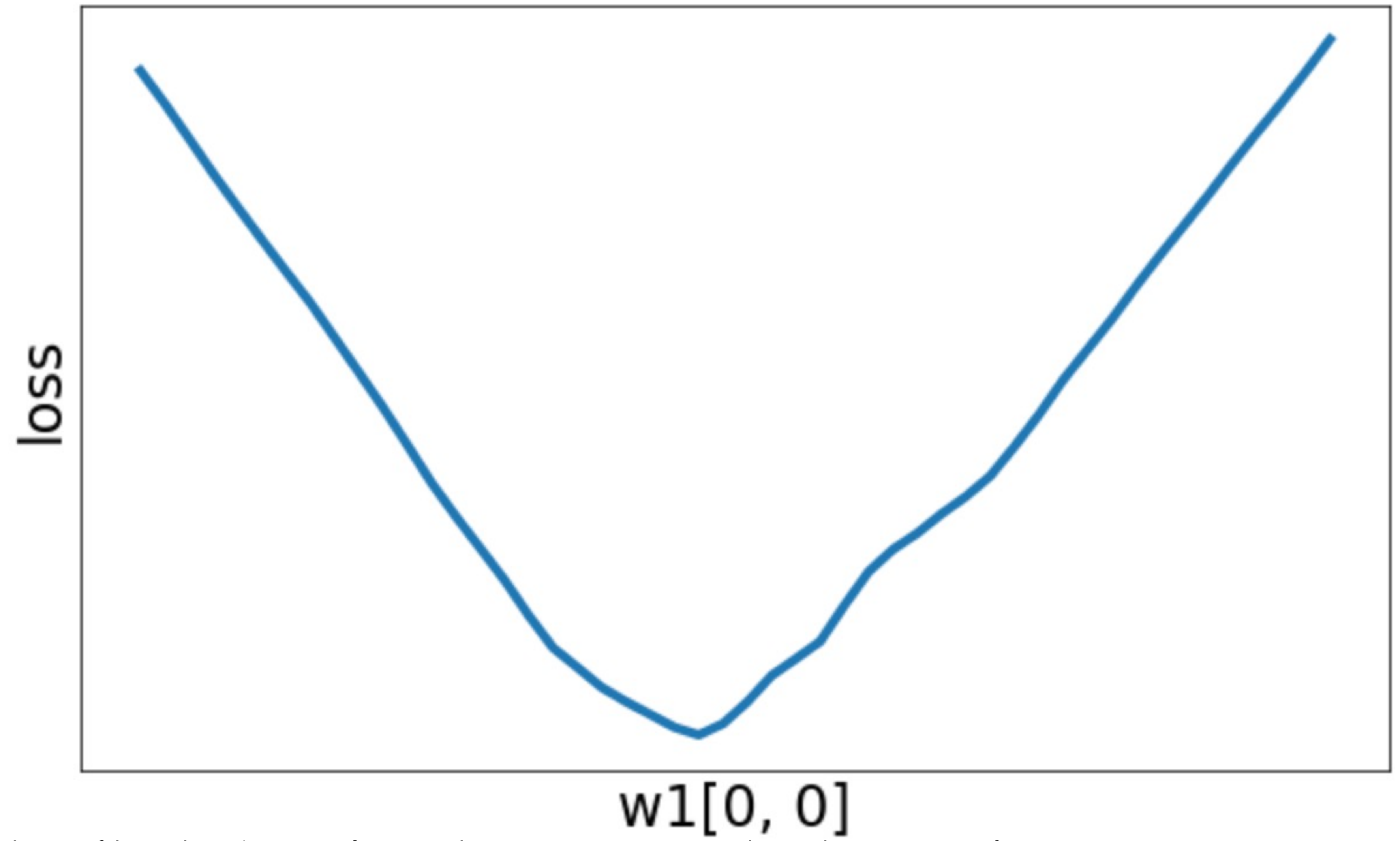
A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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

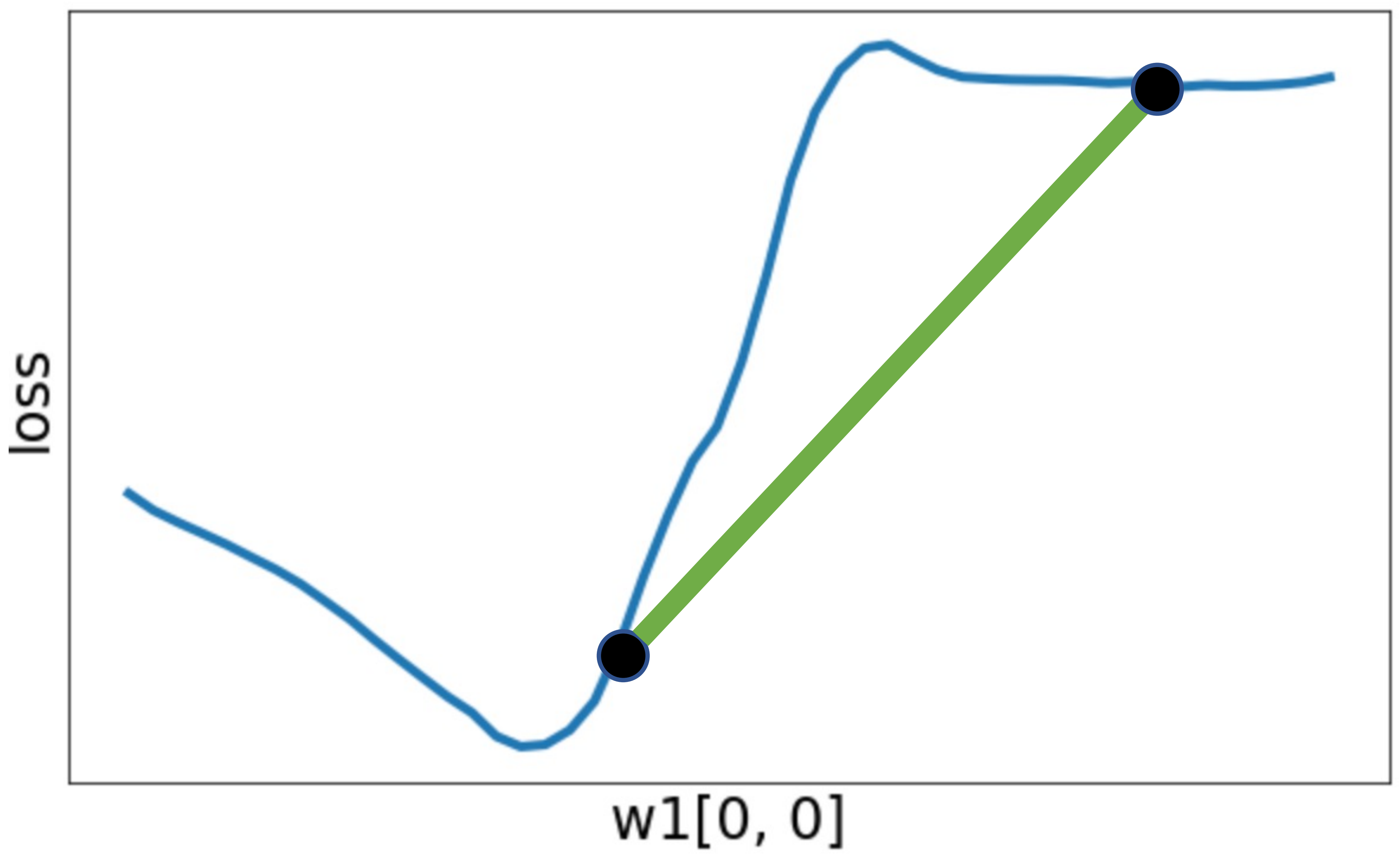
A function $f : X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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

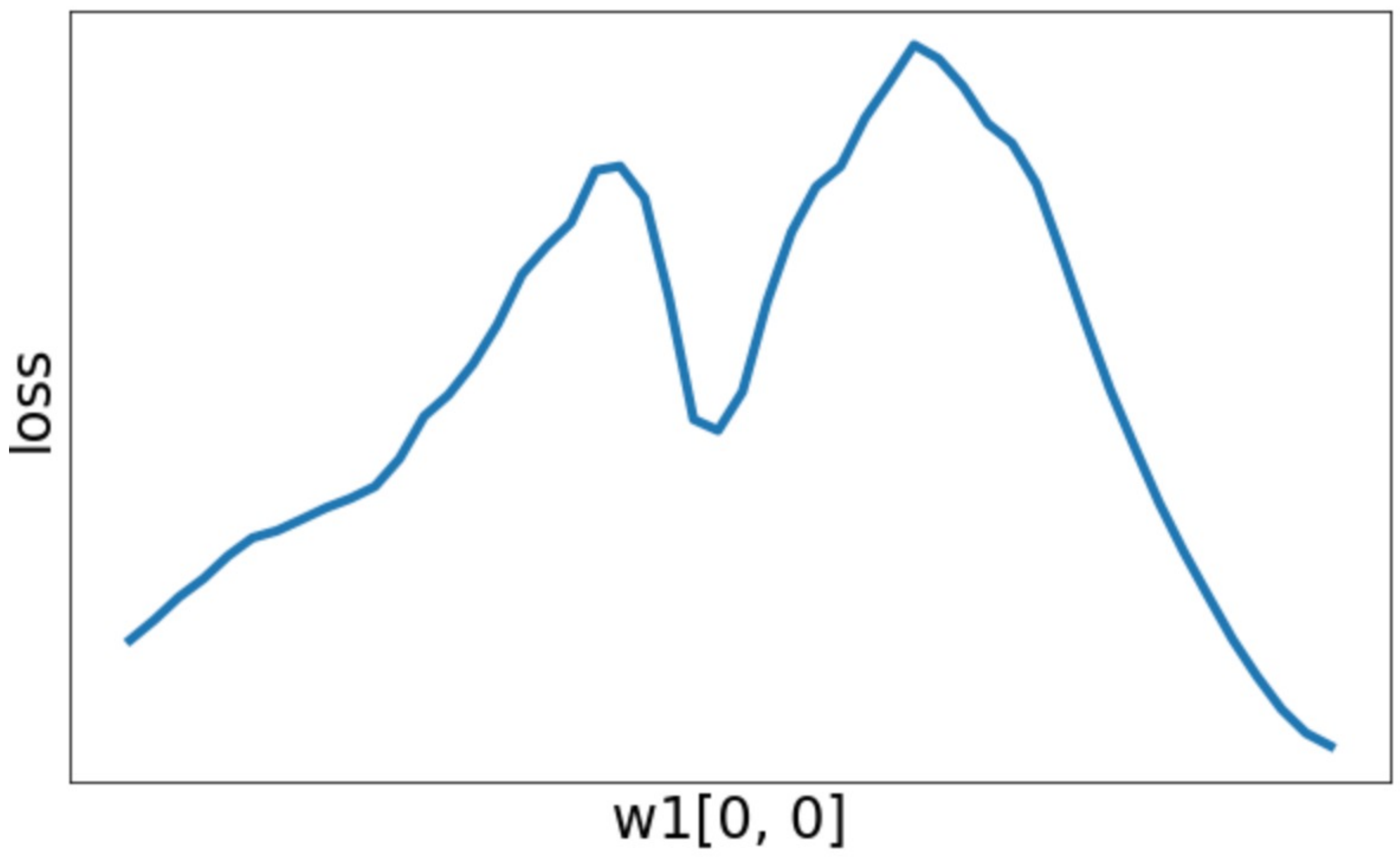
A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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***

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

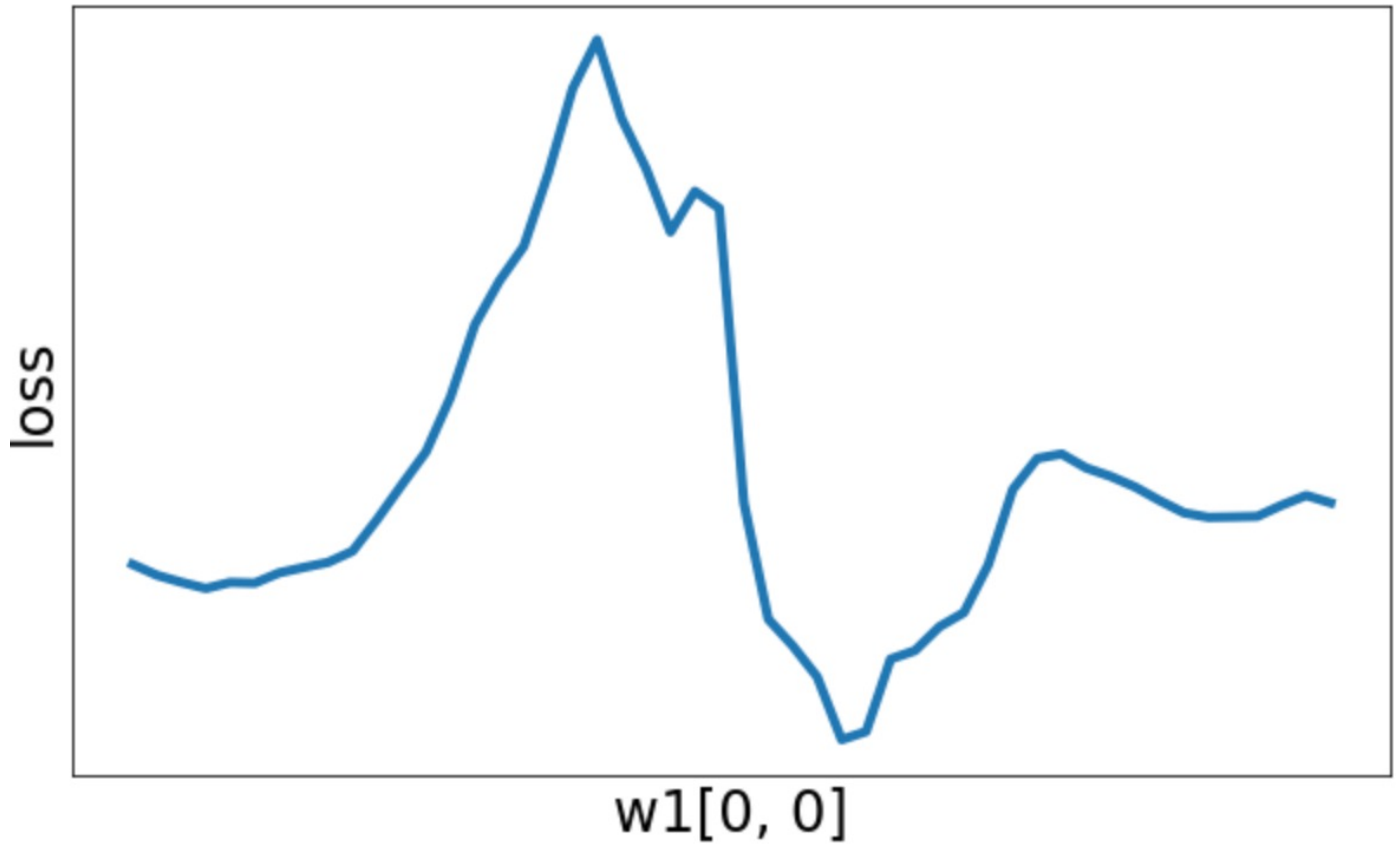
A function $f: X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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

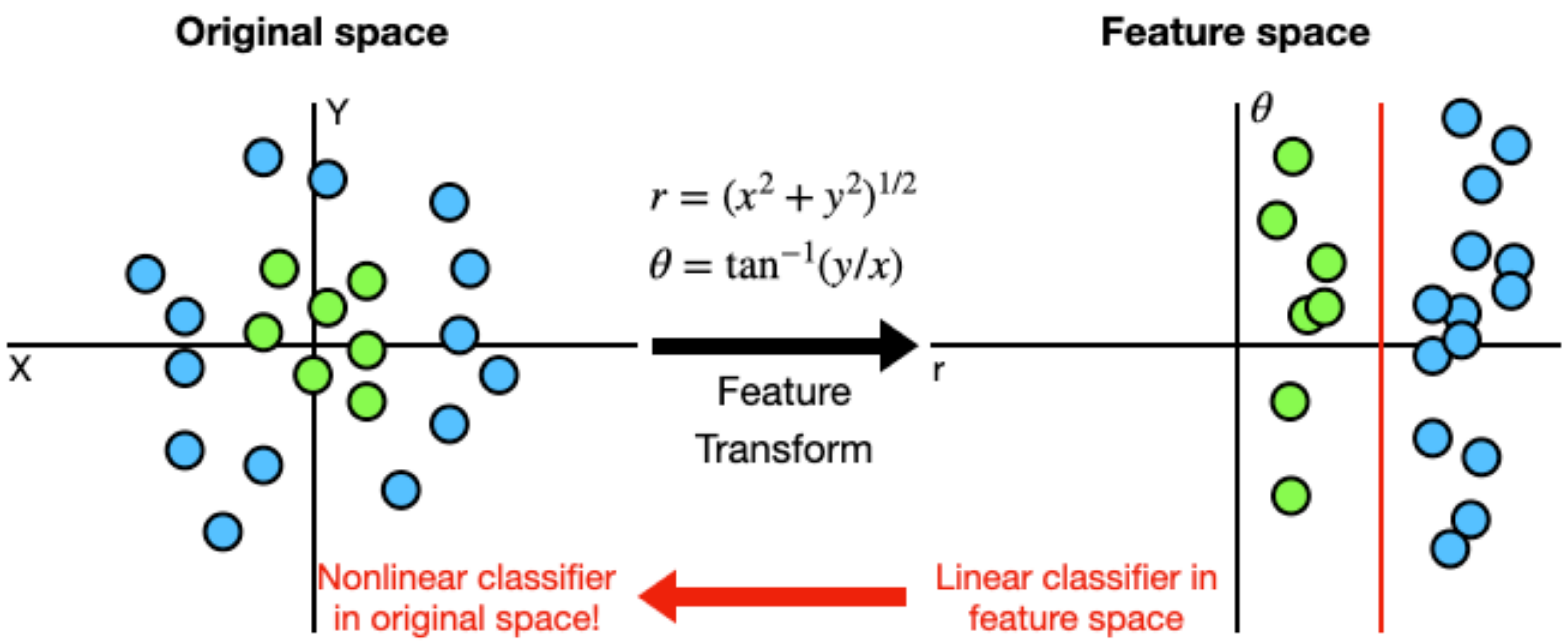
Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum***

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

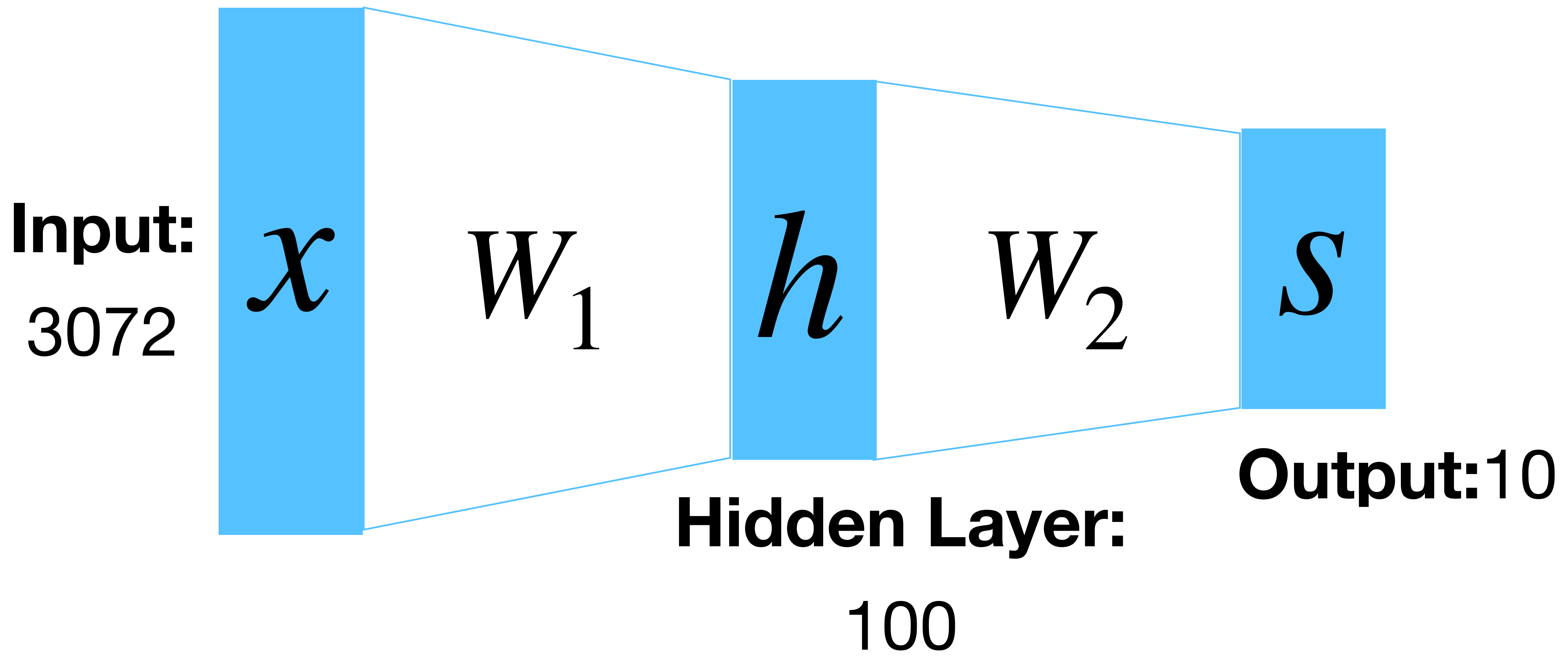


Neural Networks as learnable feature transforms

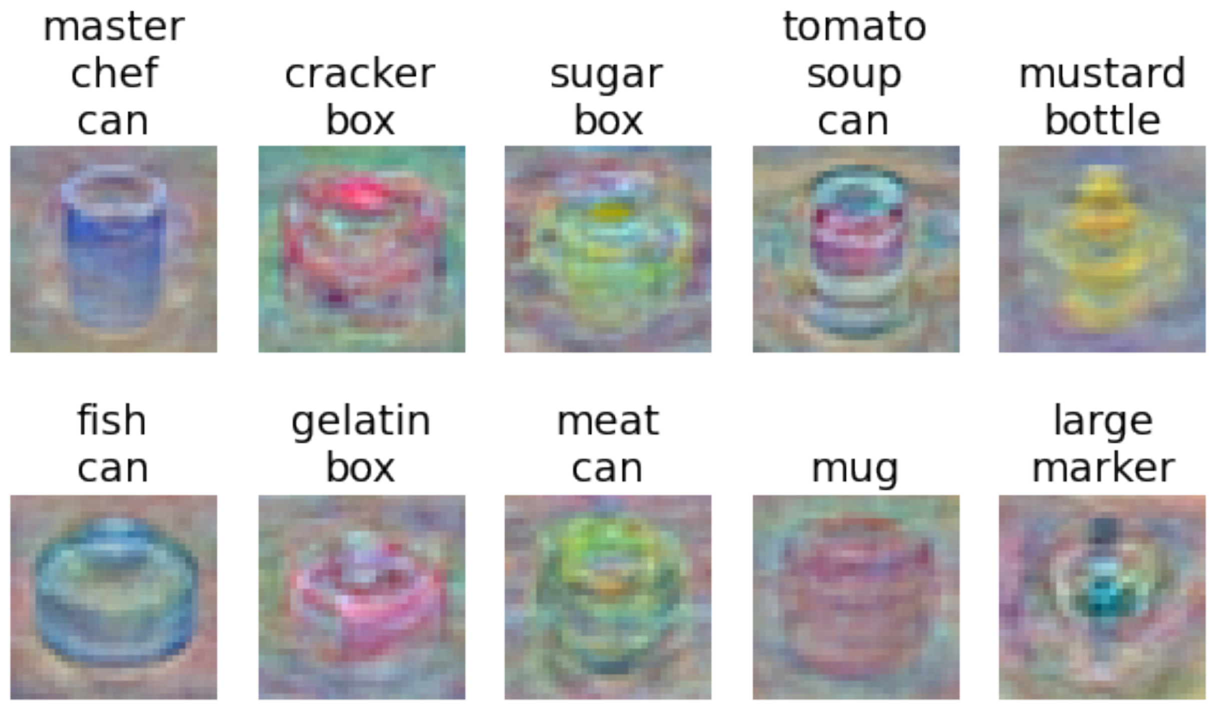
Summary

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

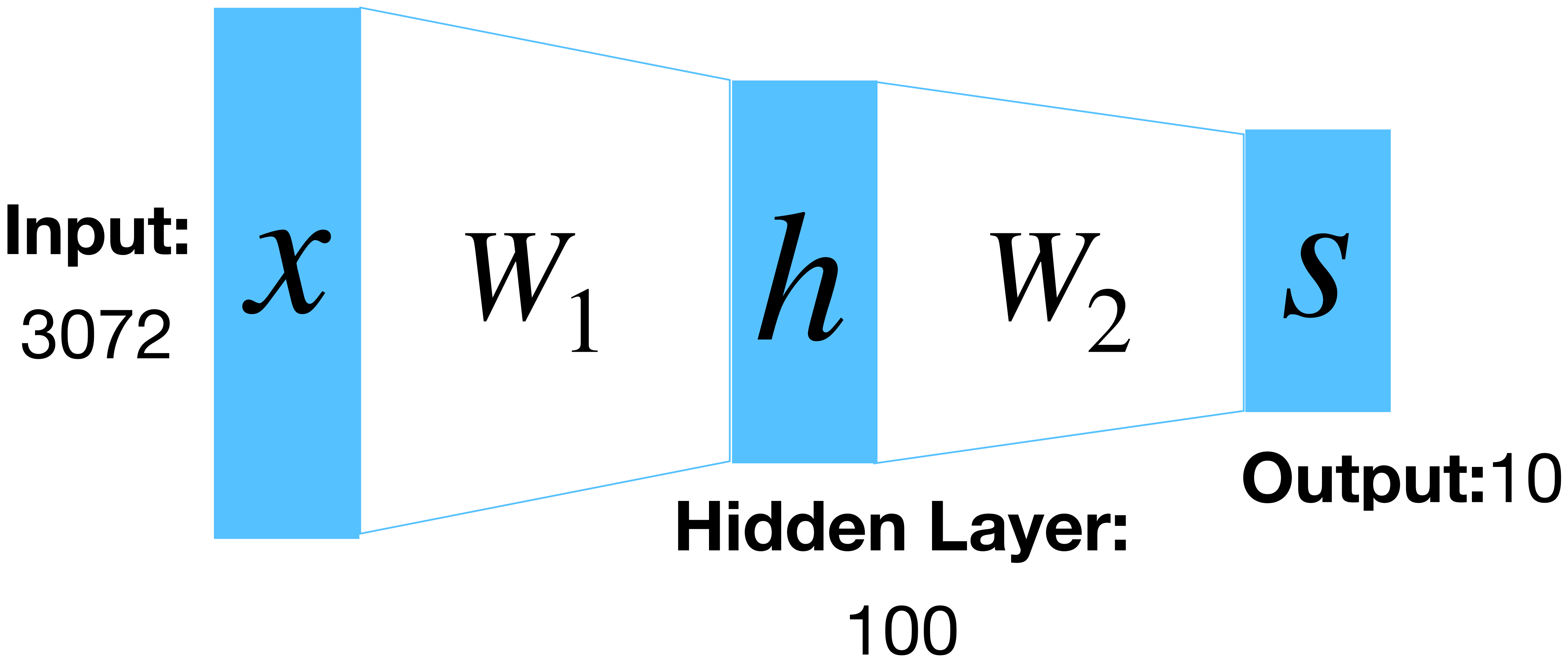


Neural networks: Many reusable templates

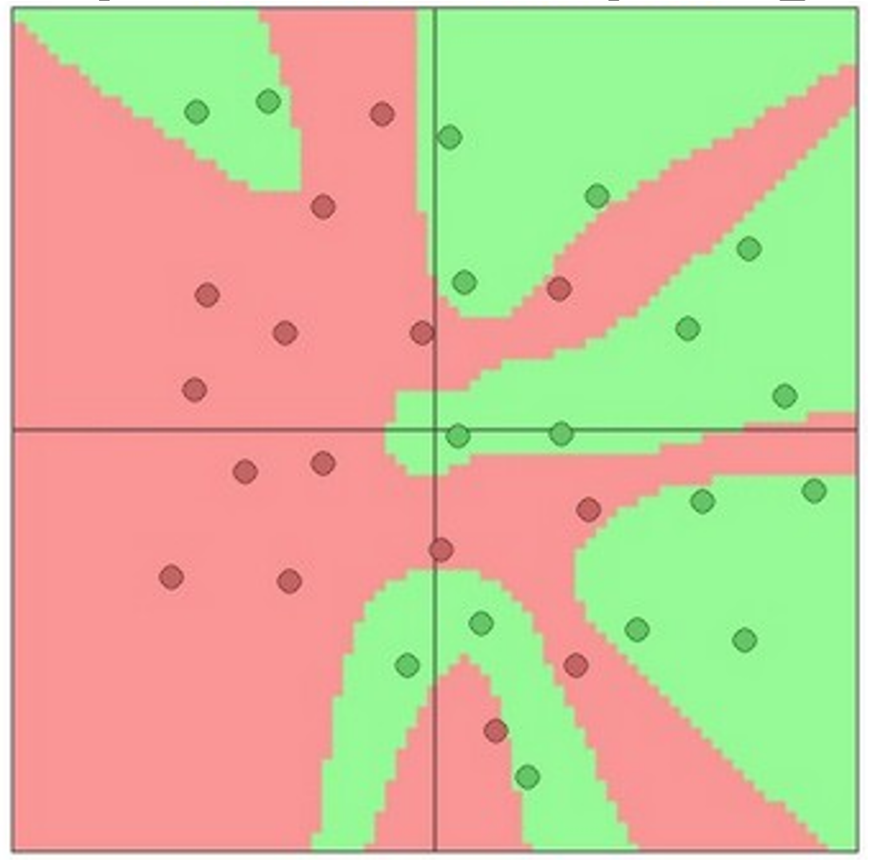
Summary

From linear classifiers to
fully-connected networks

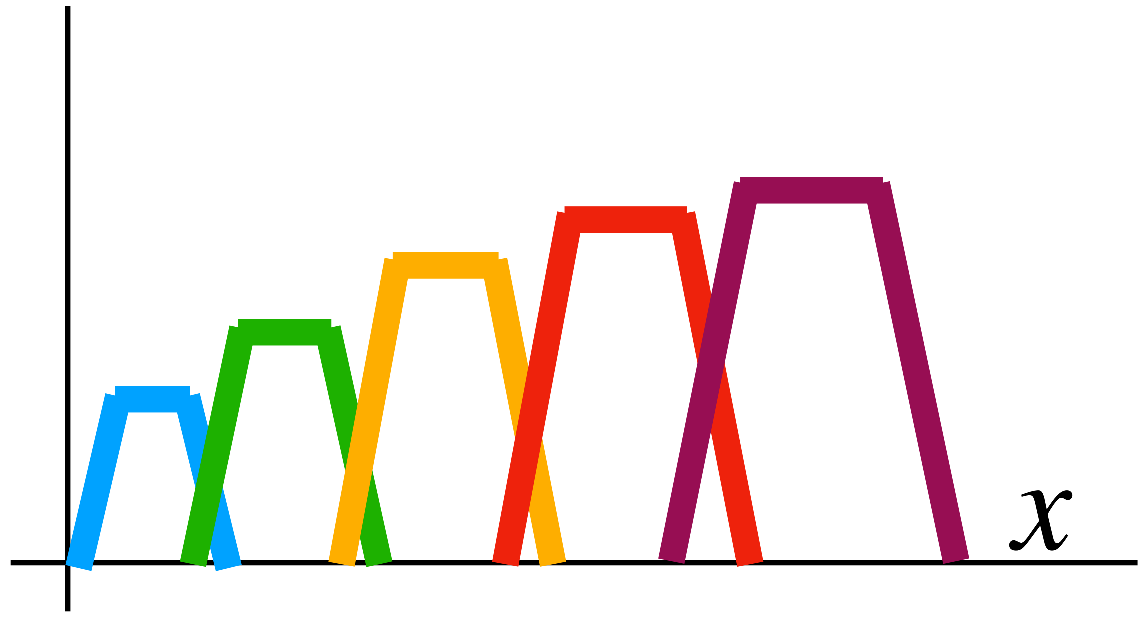
$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$



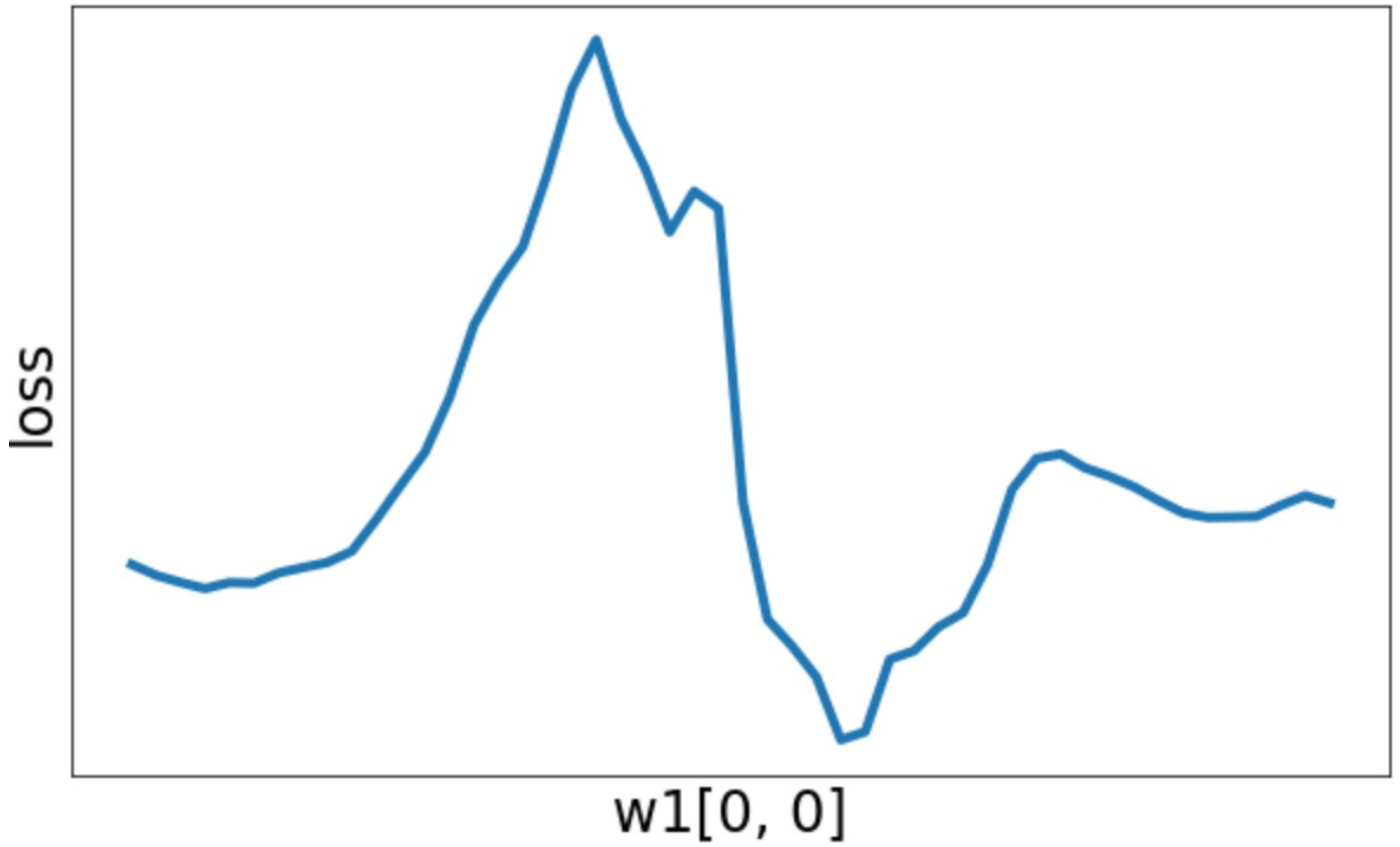
Space Warping



Universal approximation



Nonconvex



Problem: How to compute gradients?

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

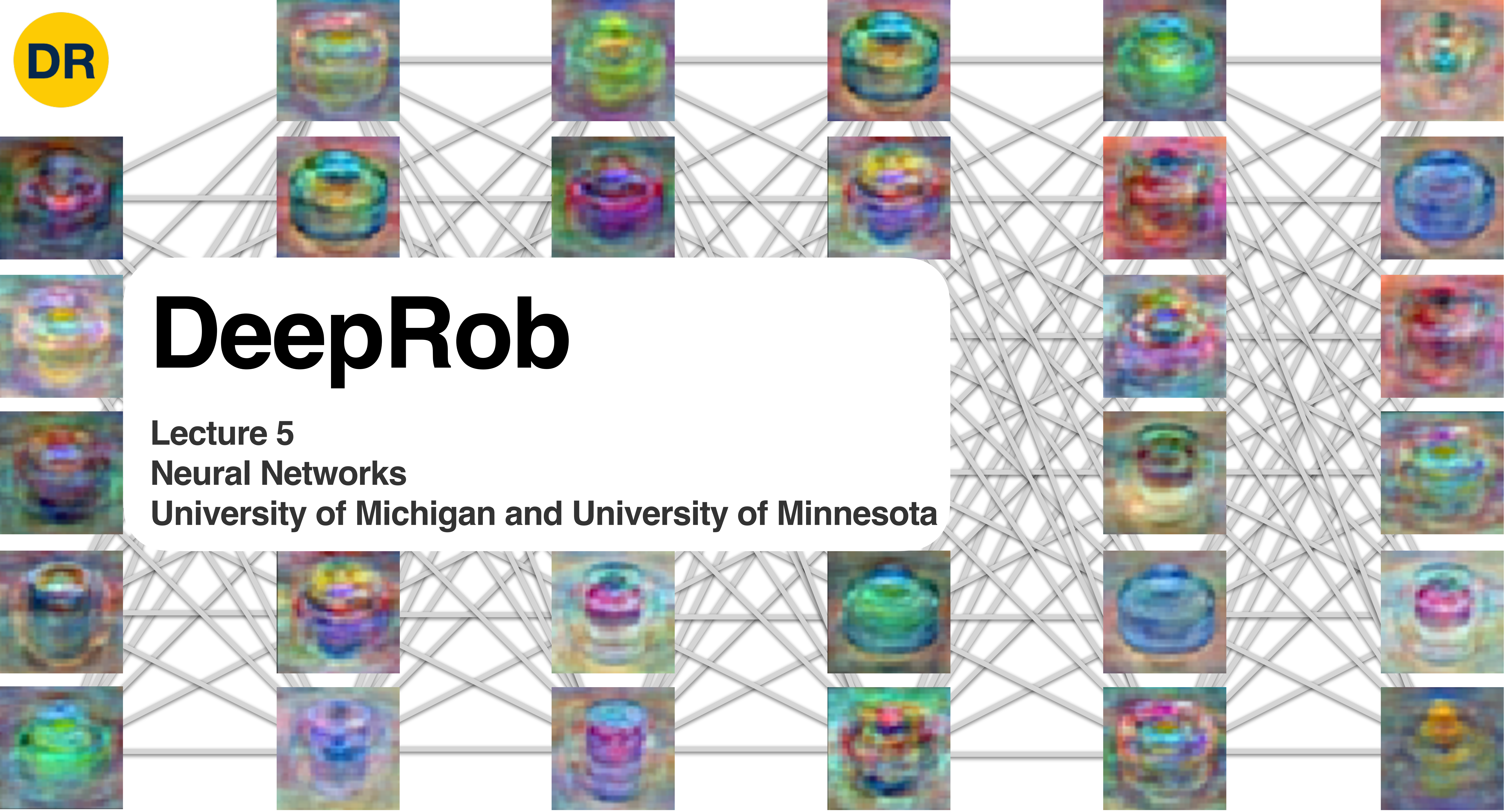
$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$ Per-element data loss

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



DeepRob

Lecture 5
Neural Networks
University of Michigan and University of Minnesota