DeepRob

Lecture 4
Neural Networks
University of Michigan | Department of Robotics

Recap: Regularization

Loss Function

$$L(W) = \boxed{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \boxed{\lambda R(W)}$$

Data Loss **Regularization**

Simple examples:

"Ridge regression" <u>L2 regularization:</u> $R(W) = \sum_{k,l} W_{k,l}^2$

"LASSO regression" L1 regularization: $R(W) = \sum_{k,l} |W_{k,l}|$

Recap: Regularization

Example:

$$x = [1,1,1,1]$$

$$w_1 = [1,0,0,0]$$

$$w_2 = [0.25,0.25,0.25,0.25]$$

L2 Regularization

$$R(W) = \sum_{k,l} W_{k,l}^2$$

Tend to shrink coefficients **Evenly**

$$w_1^T x = w_2^T x = 1$$

Same predictions, so data loss will always be the same

Recap: Regularization

Loss Function

$$L(W) = \boxed{\frac{1}{N}\sum_{i=1}^{N} L_i(f(x_i, W), y_i)} + \boxed{\lambda R(W)}$$

Data Loss **Regularization**

Simple examples:

"Ridge regression" <u>L2 regularization:</u> $R(W) = \sum_{k,l} W_{k,l}^2$

"LASSO regression" L1 regularization: $R(W) = \sum_{k,l} |W_{k,l}|$

<u>Useful for feature selection</u>

How to find a good W*?

Optimization

$$w^* = \arg\min_w L(w)$$

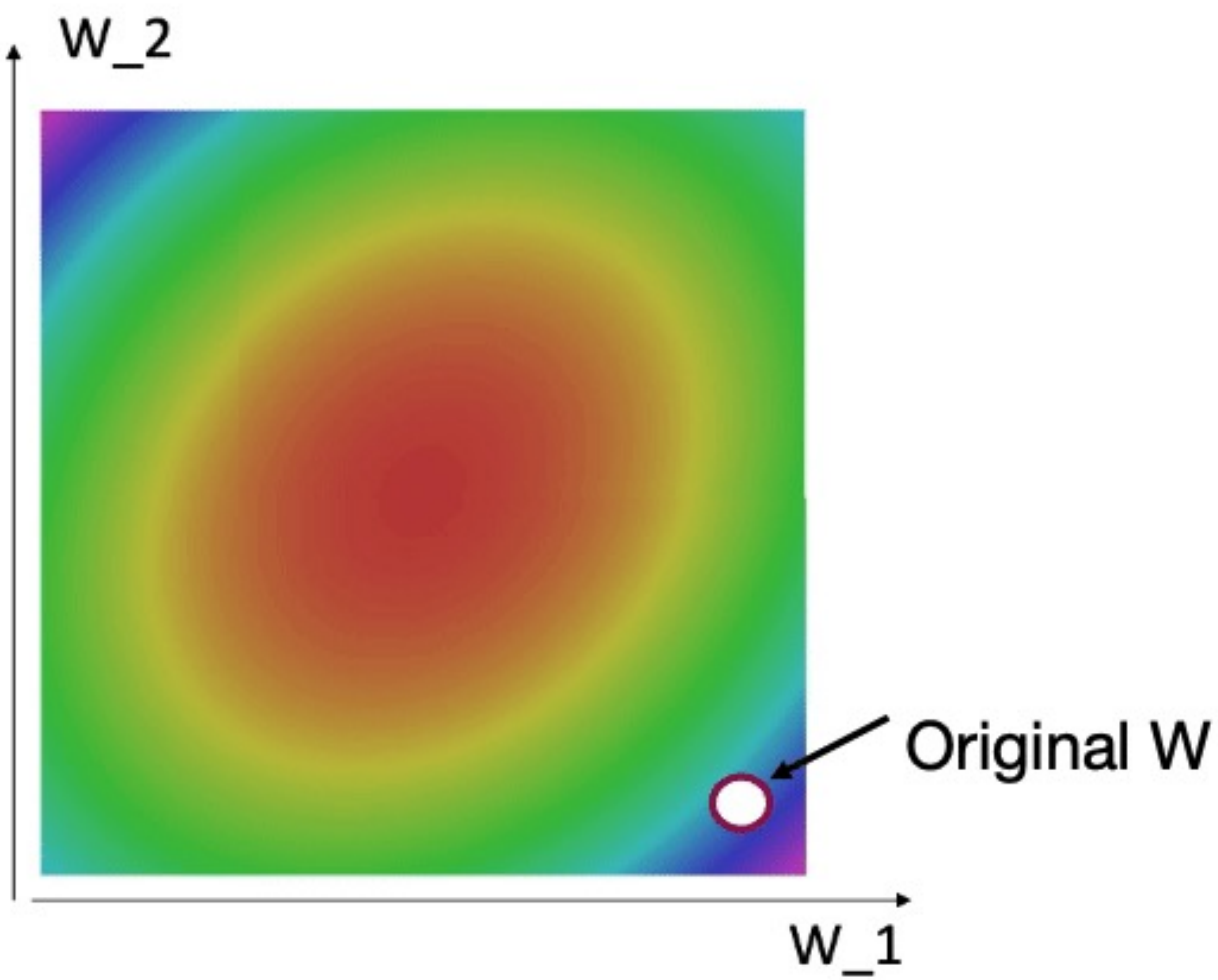
Gradient Descent

Numeric gradient: approximate, slow, easy to write

Analytic gradient: exact, fast, error-prone

$$\lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Recap: Optimization



SGD

$$w_{t+1} = w_t - \alpha \nabla L(w_t)$$

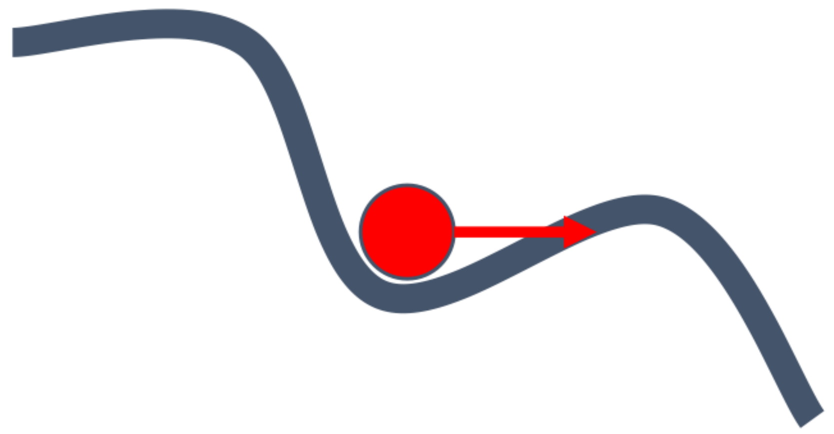
```
for t in range(num_steps):
  dw = compute_gradient(w)
  w -= learning_rate * dw
```

SGD + Momentum

$$v_{t+1} = \rho v_t + \nabla L(w_t)$$
$$w_{t+1} = w_t - \alpha v_{t+1}$$

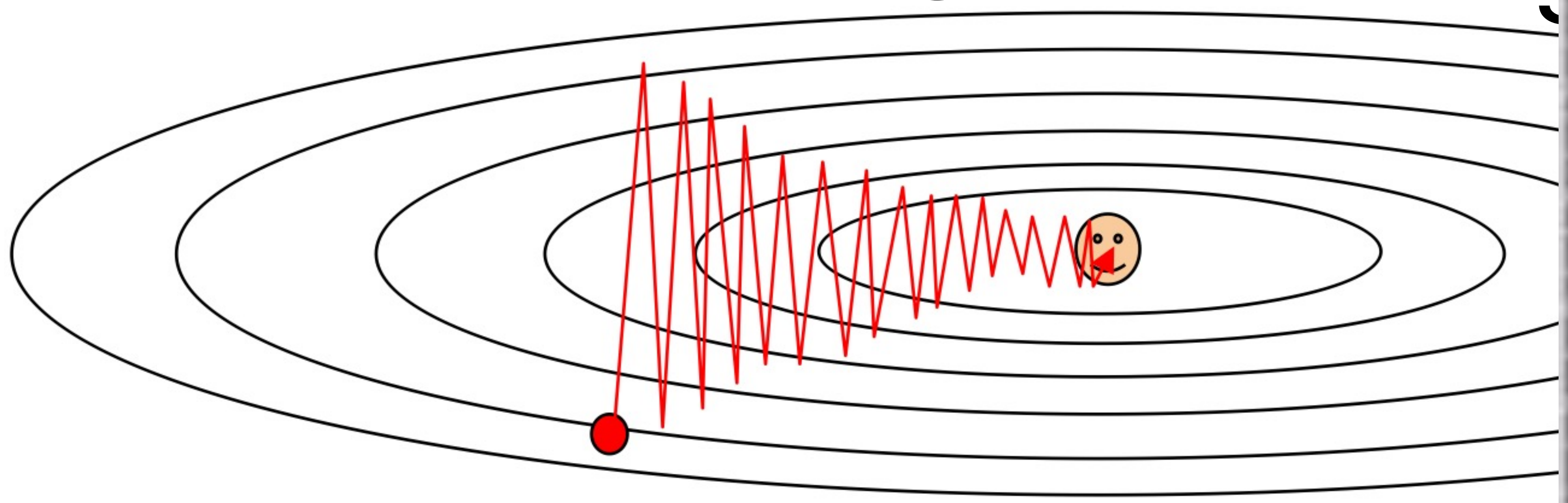
SGD + Momentum

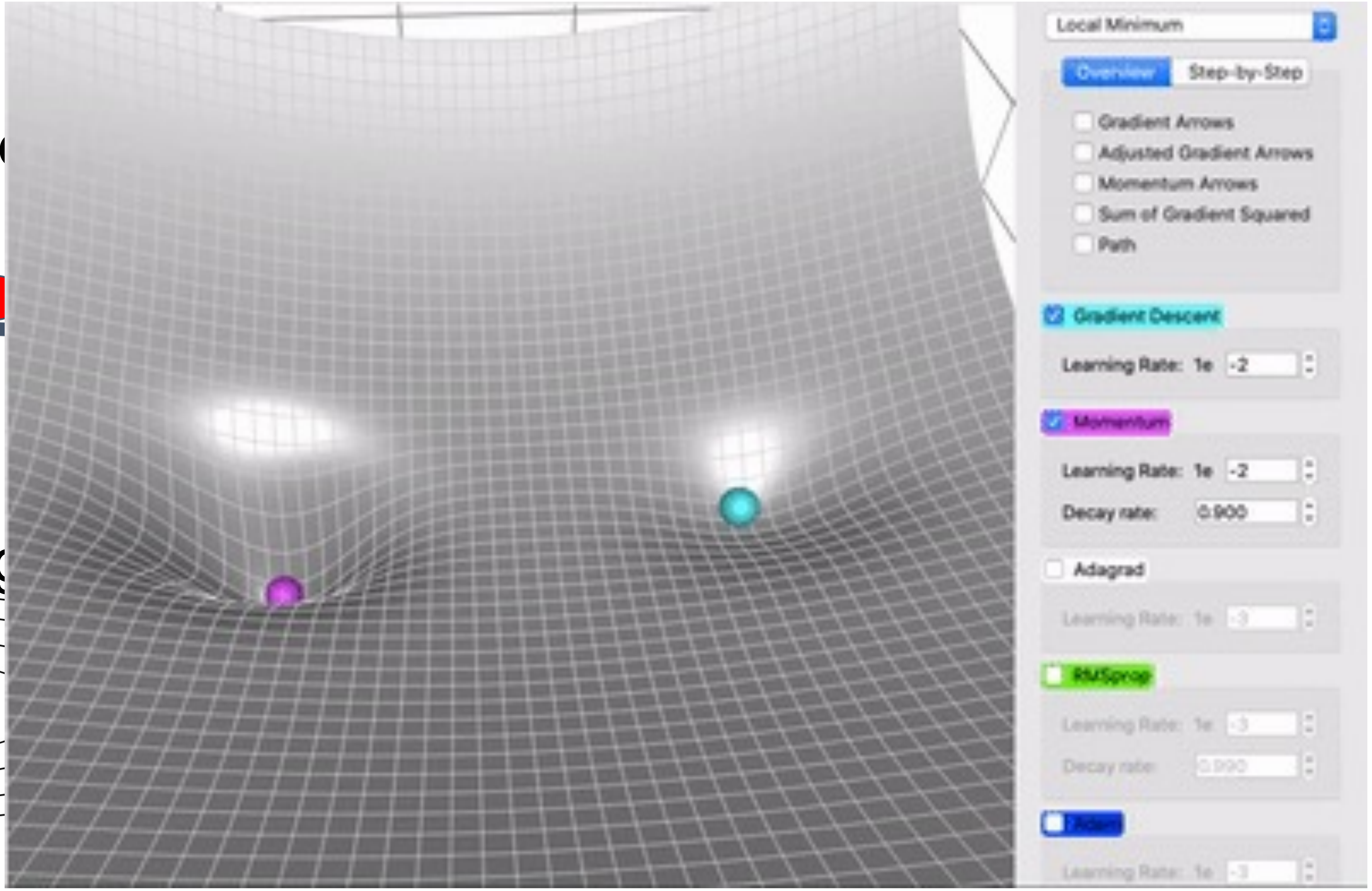
Local Minima



Saddle Po

Poor Conditioning



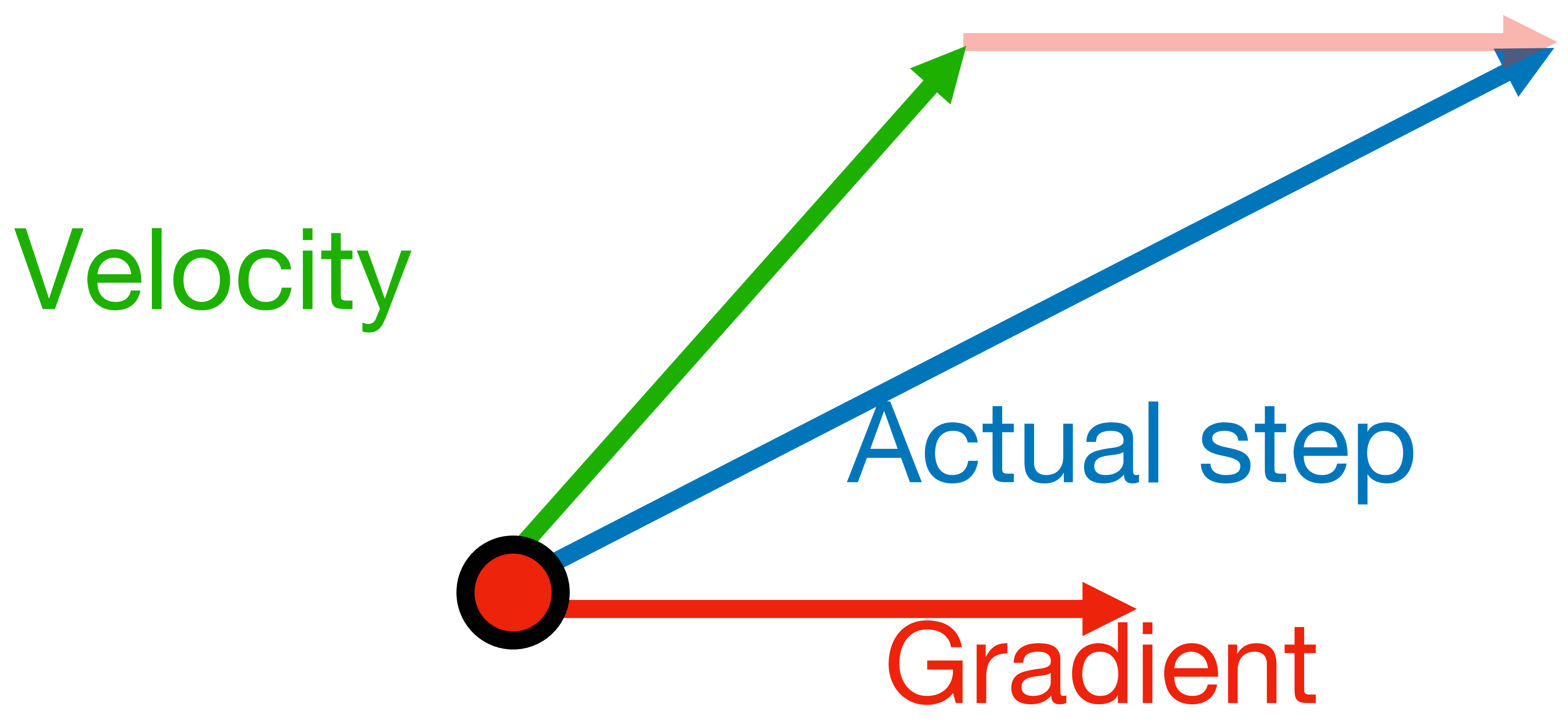


SGD SGD+Momentum

Sutskever et al, "On the importance of initialization and momentum in deep learning," ICML 2013

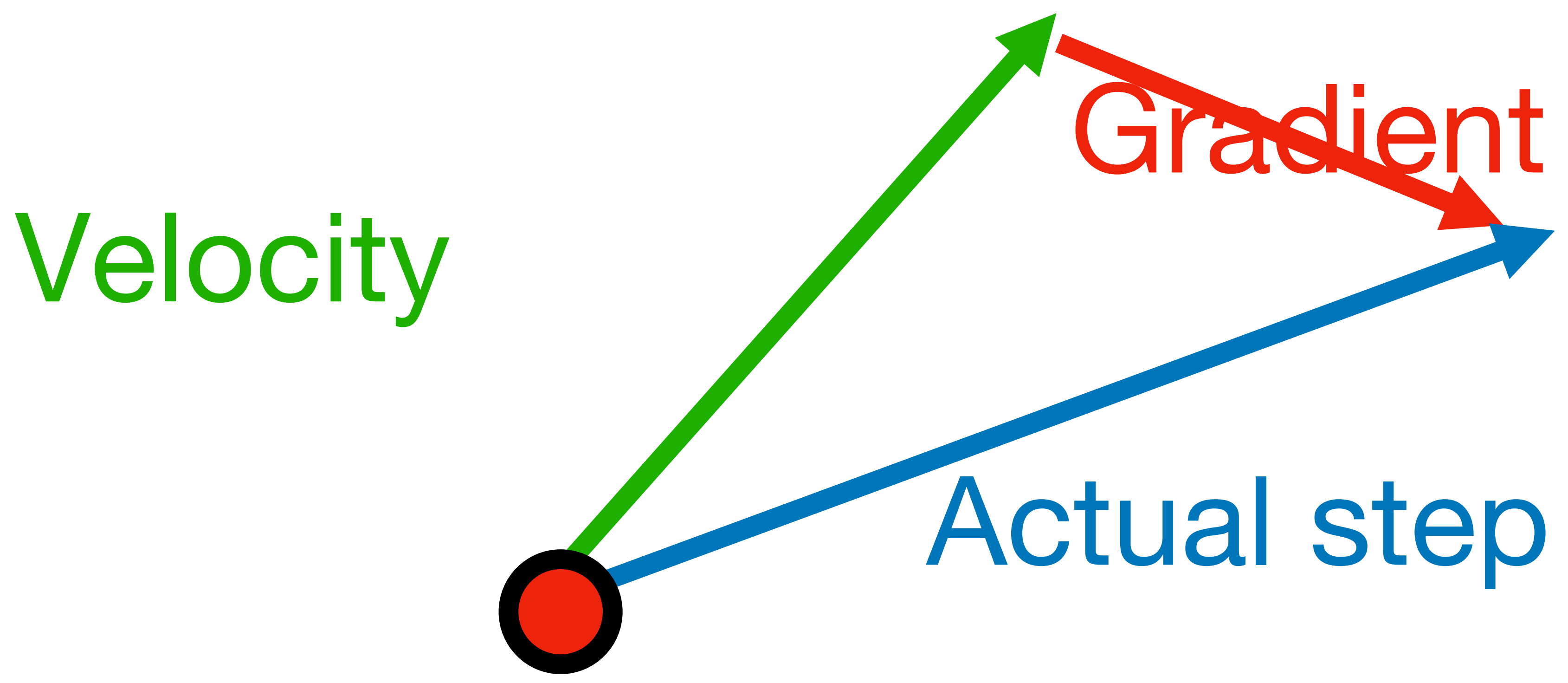
SGD + Momentum

Momentum update:



Combine gradient at current point with velocity to get step used to update weights

Nesterov Momentum



"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

"Per-parameter learning rates" or "adaptive learning rates"

Duchi et al, "Adaptive sub gradient methods for online learning and stochastic optimization," JMLR 2011

SGD in PyTorch

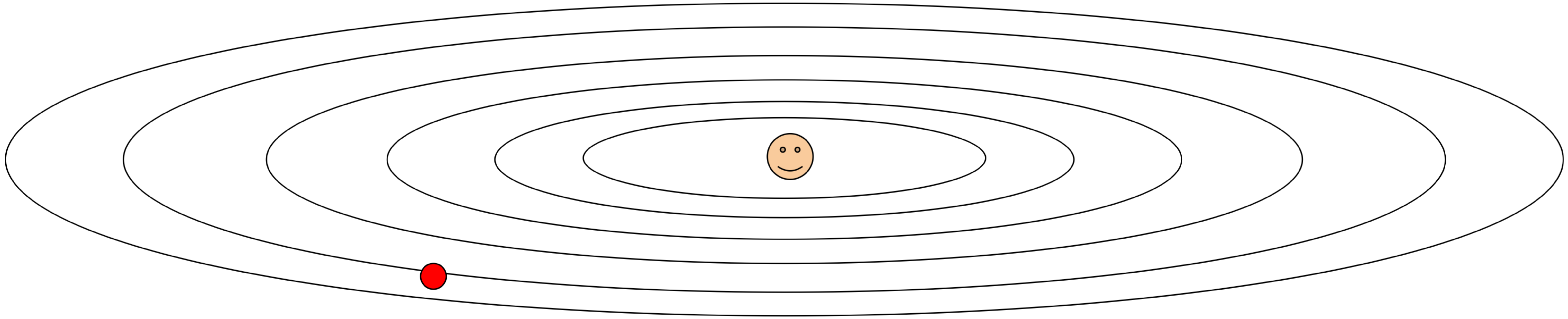
```
train_dataloader =
torch.utils.data.DataLoader(train_dataset, batch_size=64,
shuffle=True)

optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
```

Duchi et al, “Adaptive sub gradient methods for online learning and stochastic optimization,” JMLR 2011

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

Problem: AdaGrad will slow over many iterations



Q: What happens with AdaGrad?

Progress along “steep” directions is damped; progress along “flat” directions is accelerated

Duchi et al, “Adaptive sub gradient methods for online learning and stochastic optimization,” JMLR 2011

RMSProp: "Leaky AdaGrad"

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared += dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

AdaGrad

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Kingma and Ba, "Adam: A method for stochastic optimization," ICLR 2015

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

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

SGD+Momentum

Kingma and Ba, “Adam: A method for stochastic optimizatio

Adam (almost): RMSProp + Momentum

```
moment1 = 0
moment2 = 0
for t in range(1, num_steps + 1):  # Start at t = 1
  dw = compute_gradient(w)
  moment1 = beta1 * moment1 + (1 - beta1) * dw
  moment2 = beta2 * moment2 + (1 - beta2) * dw * dw
  w -= learning_rate * moment1 / (moment2.sqrt() + 1e-7)
```

Adam

Momentum

AdaGrad / RMSProp

```
grad_squared = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dw * dw
  w -= learning_rate * dw / (grad_squared.sqrt() + 1e-7)
```

RMSProp

Kingma and Ba, “Adam: A method for stochastic optimization,” ICLR 2015

Adam: Very common in Practice!

for input to the CNN; each colored pixel in the image yields a 7D one-hot vector. Following common practice, the network is trained end-to-end using stochastic gradient descent with the Adam optimizer [22]. We anneal the learning rate to 0 using a half cosine schedule without restarts [28].

Bakhtin, van der Maaten, Johnson, Gustafson, and Girshick, NeurIPS 2019

We train all models using Adam [23] with learning rate 10^{-4} and batch size 32 for 1 million iterations; training takes about 3 days on a single Tesla P100. For each mini-batch we first update f, then update D_{img} and D_{obj}.

Johnson, Gupta, and Fei-Fei, CVPR 2018

ganized into three residual blocks. We train for 25 epochs using Adam [27] with learning rate 10^{-4} and 32 images per batch on 8 Tesla V100 GPUs. We set the `cubify` thresh-

Gkioxari, Malik, and Johnson, ICCV 2019

sampled with each bit drawn uniformly at random. For gradient descent, we use Adam [29] with a learning rate of 10^{-3} and default hyperparameters. All models are trained with batch size 12. Models are trained for 200 epochs, or 400 epochs if being trained on multiple noise layers.

Zhu, Kaplan, Johnson, and Fei-Fei, ECCV 2018

16 dimensional vectors. We iteratively train the Generator and Discriminator with a batch size of 64 for 200 epochs using Adam [22] with an initial learning rate of 0.001.

Gupta, Johnson, et al, CVPR 2018

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3, 5e-4, 1e-4 is a great starting point for many models!

AdamW: Decouple Weight Decay

"weight decay" in L2 regularization

```
final_loss = loss + wd * all_weights.pow(2).sum() / 2
w = w - lr * w.grad - lr * wd * w
```

"weight decay" in AdamW

```
CLASS torch.optim.AdamW(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0.01, amsgrad=False, *, maximize=False, foreach=None, capturable=False, differentiable=False, fused=None) [SOURCE]
```

Loshchilov and Hunter, "Decoupled Weight Decay Regularization," ICLR 2019

Optimization Algorithm Comparison

Algorithm	Tracks first moments (Momentum)	Tracks second moments (Adaptive learning rates)	Leaky second moments	Bias correction for moment estimates
SGD	x	x	x	x
SGD+Momentum	✓	x	x	x
Nesterov	✓	x	x	x
AdaGrad	x	✓	x	x
RMSProp	x	✓	✓	x
Adam	✓	✓	✓	✓

In practice:

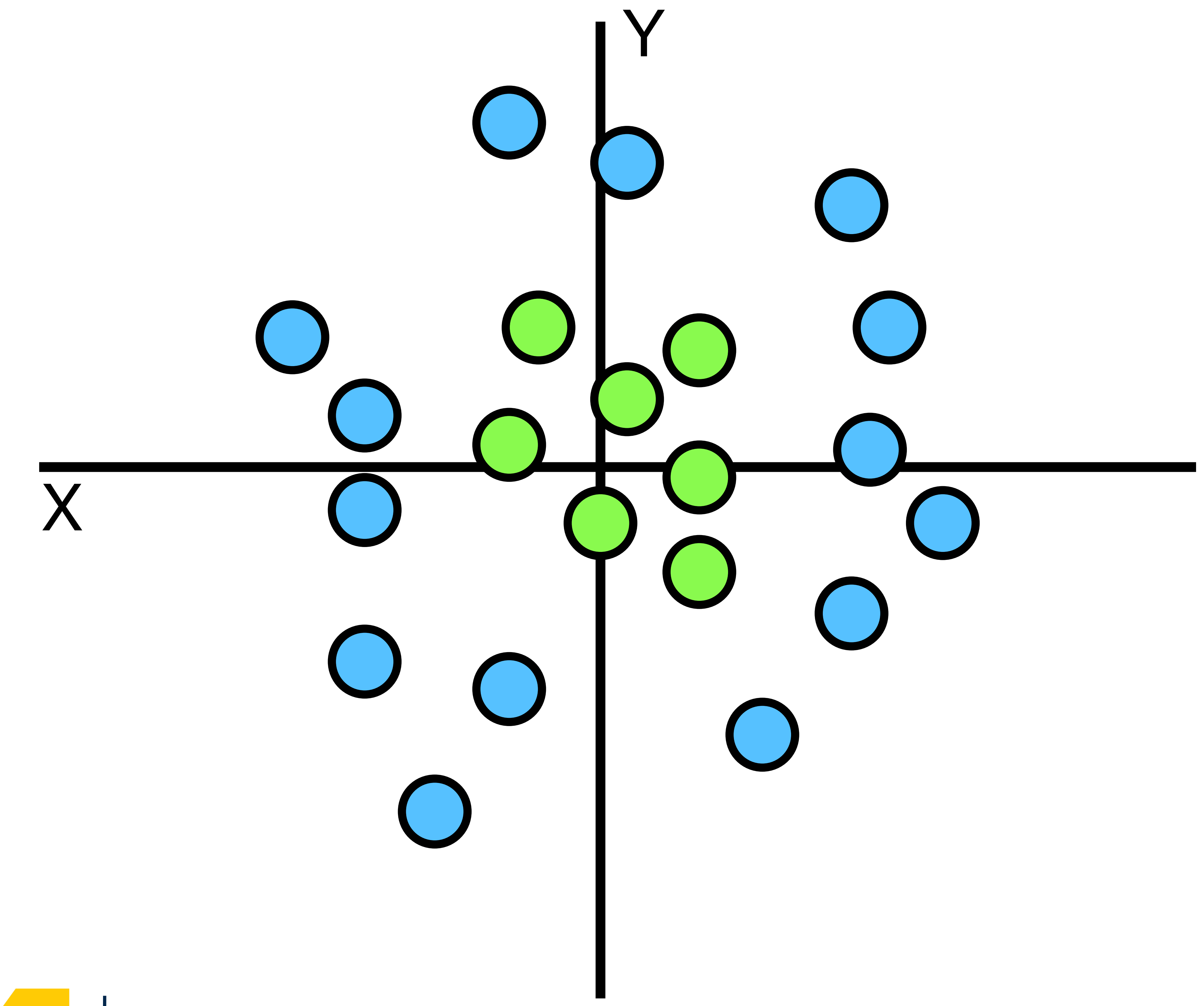
- **Adam** is a good default choice in many cases
  SGD+Momentum can outperform Adam but may require more tuning.

- If you can afford to do full batch updates then try out second-order optimization (e.g., **L-BFGS)**, and don't forget to disable all sources of noise

Neural Networks

Problem: Linear Classifiers aren't that powerful

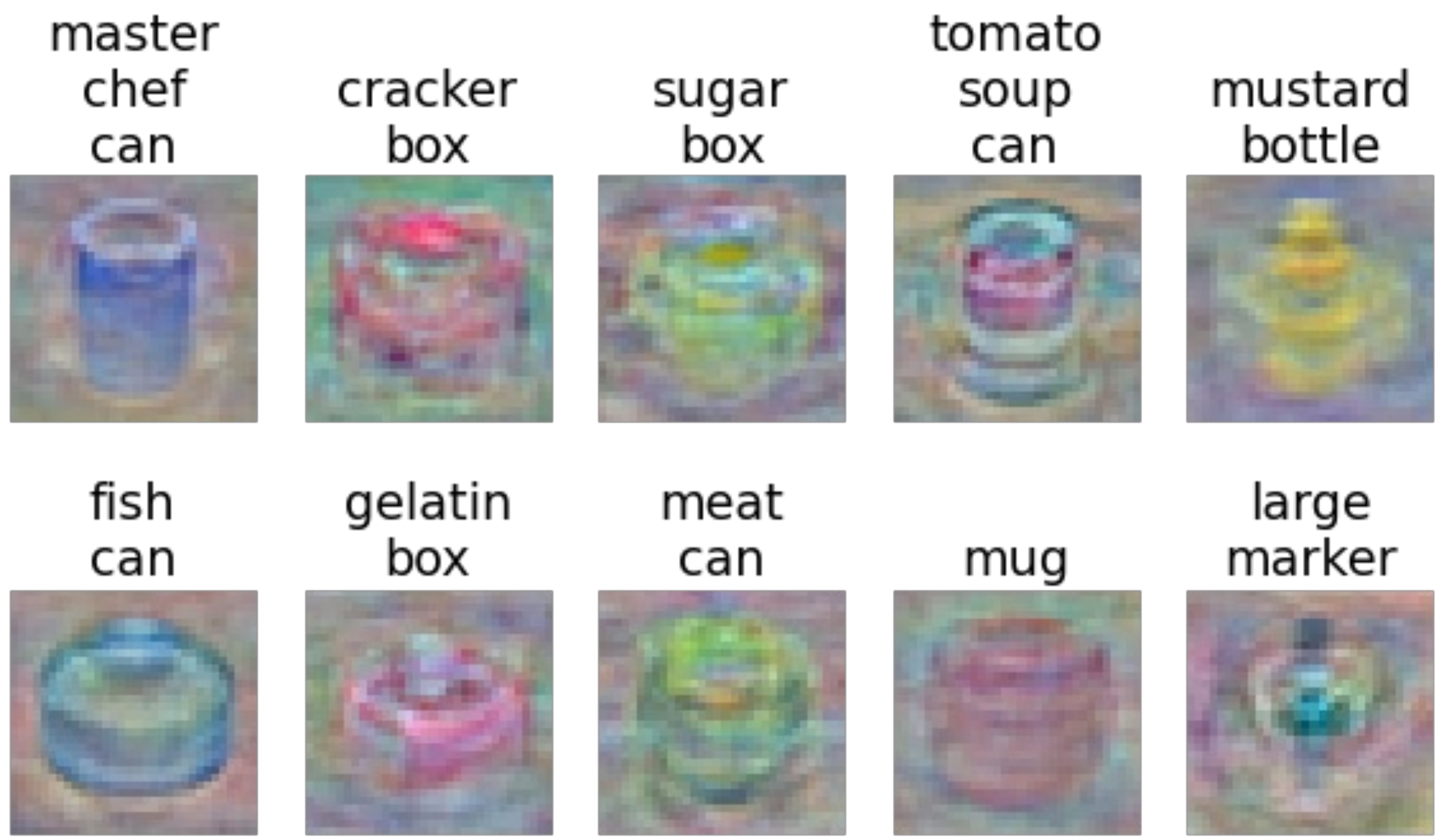
Geometric Viewpoint



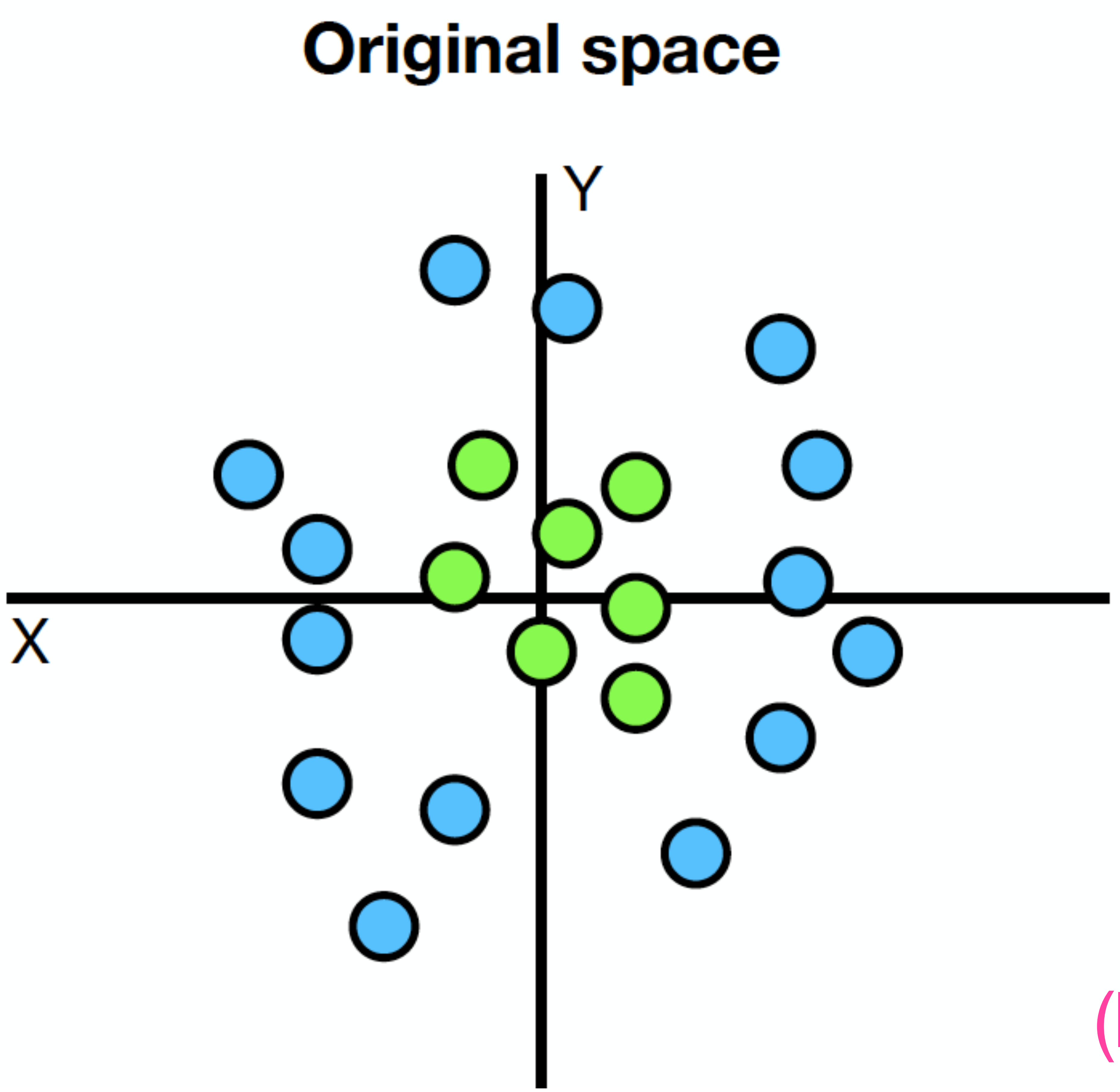
Visual Viewpoint

One template per class:

Can't recognize different modes of a class



One solution: Feature Transformation

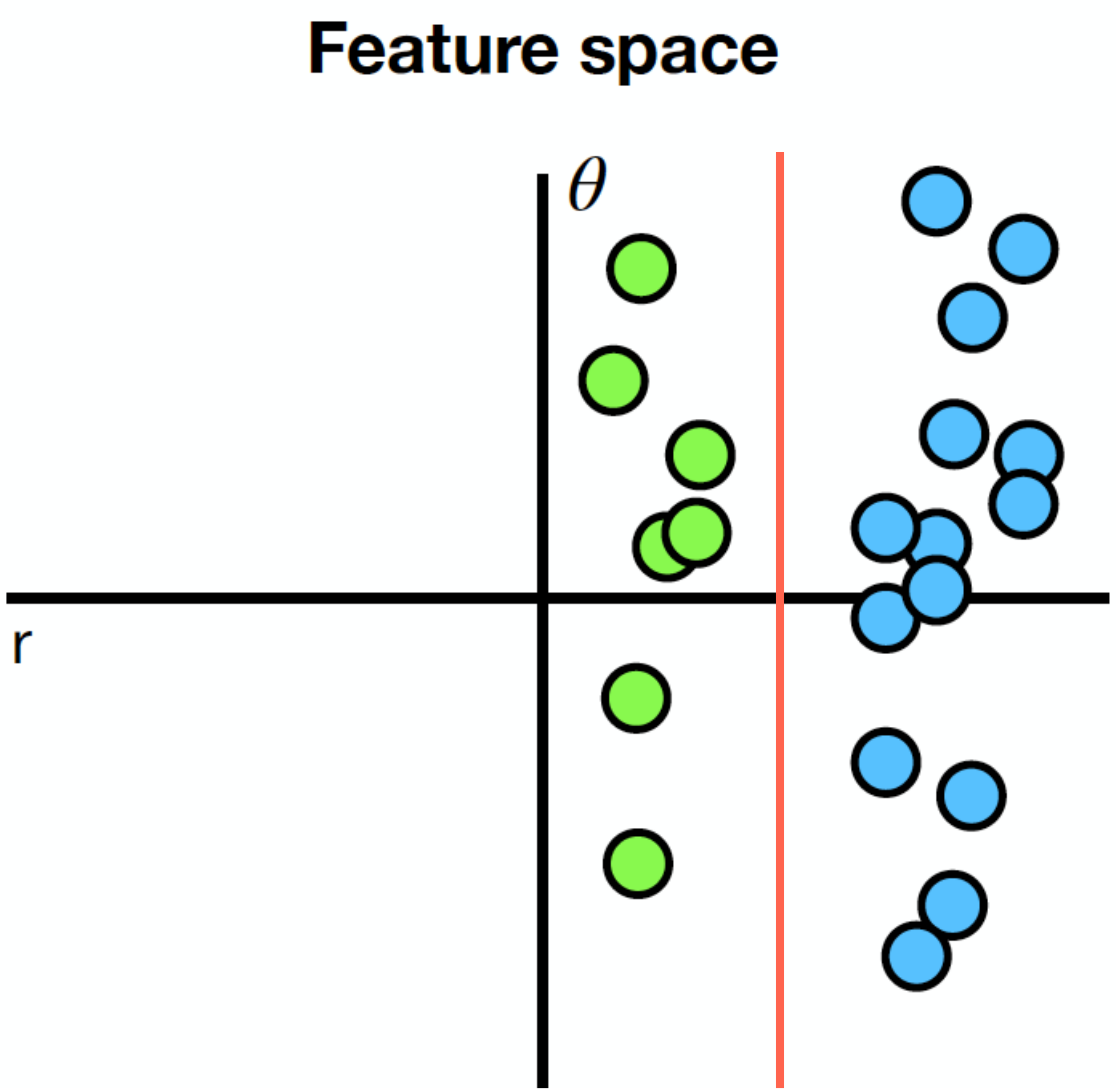


$$r = (x^2 + y^2)^{1/2}$$

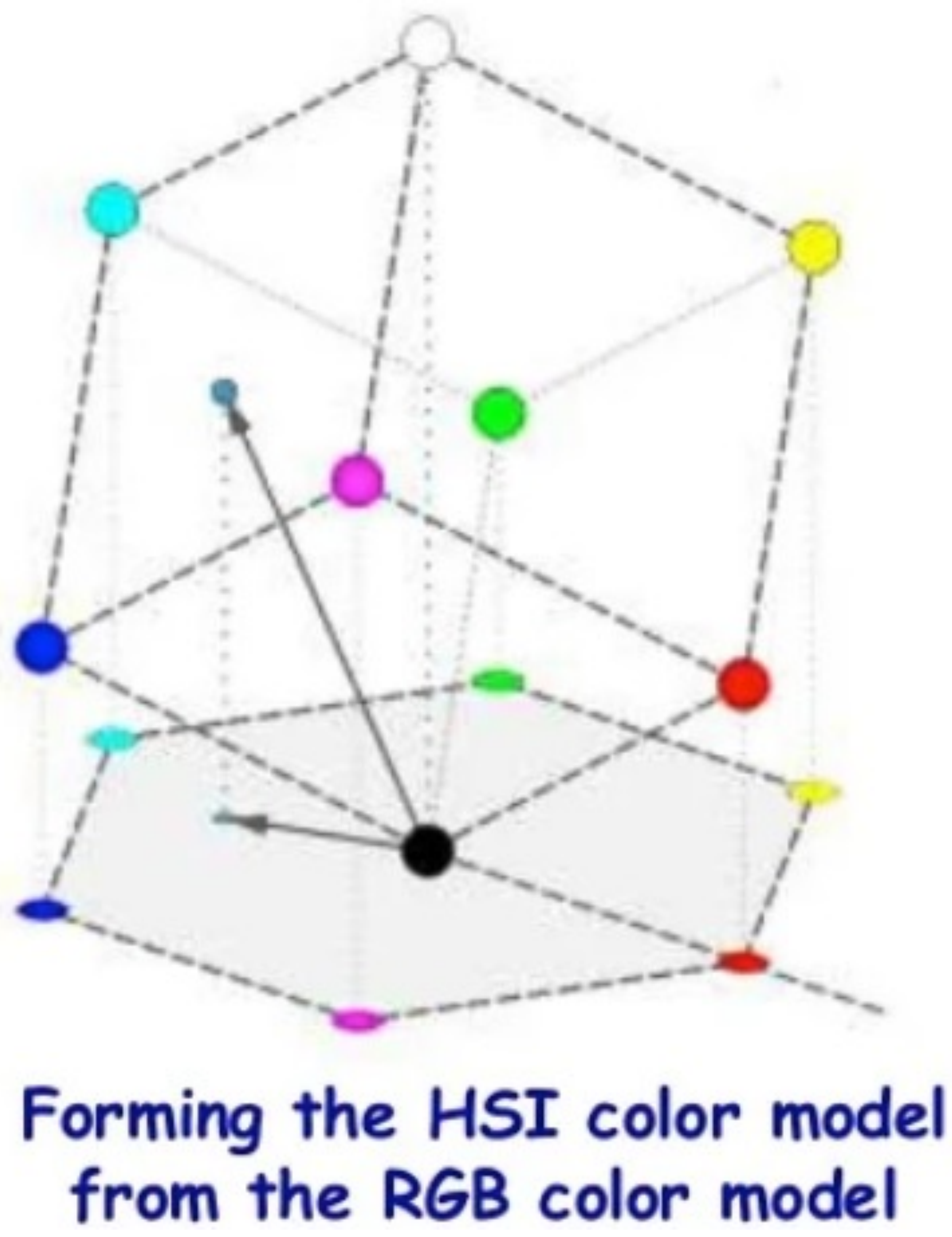
$$\theta = \tan^{-1}(y/x)$$

Feature Transform

(hand-crafted!)



RGB – HSV color space



- **H**ue, **S**aturation, **V**alue
- Nonlinear – reflects topology of colors by coding **hue** as an angle

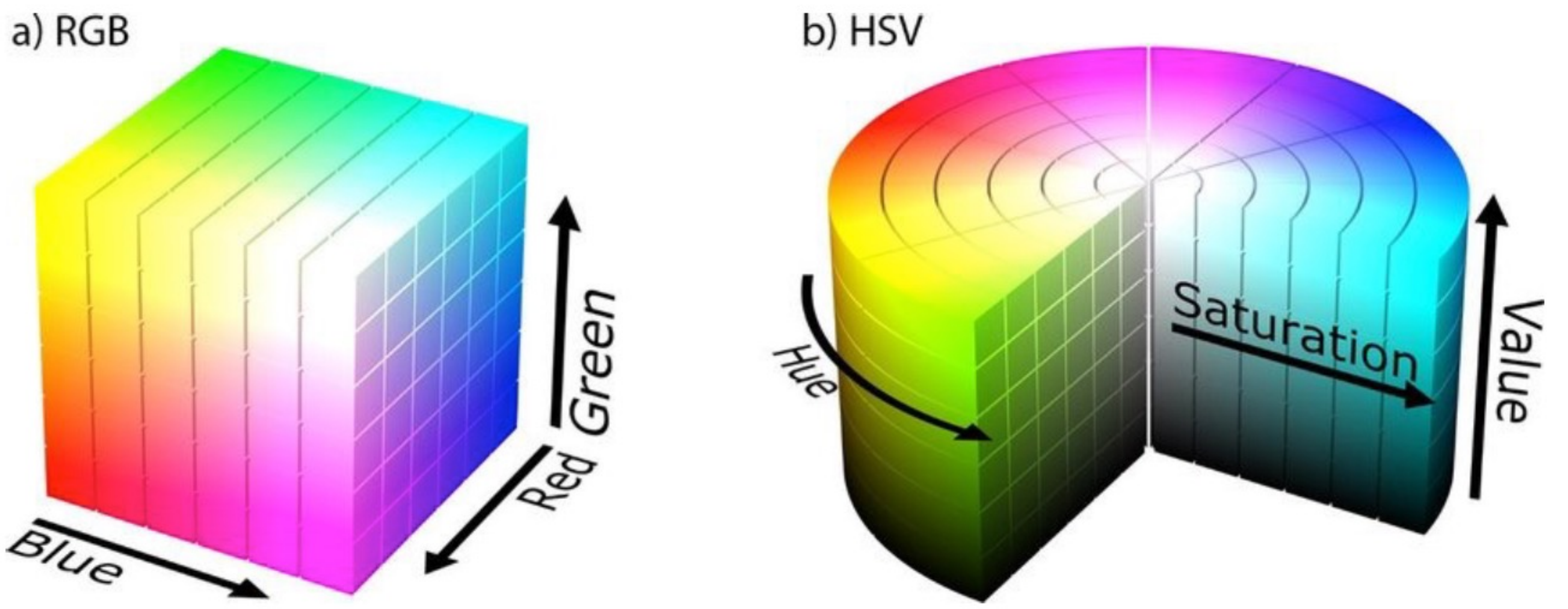
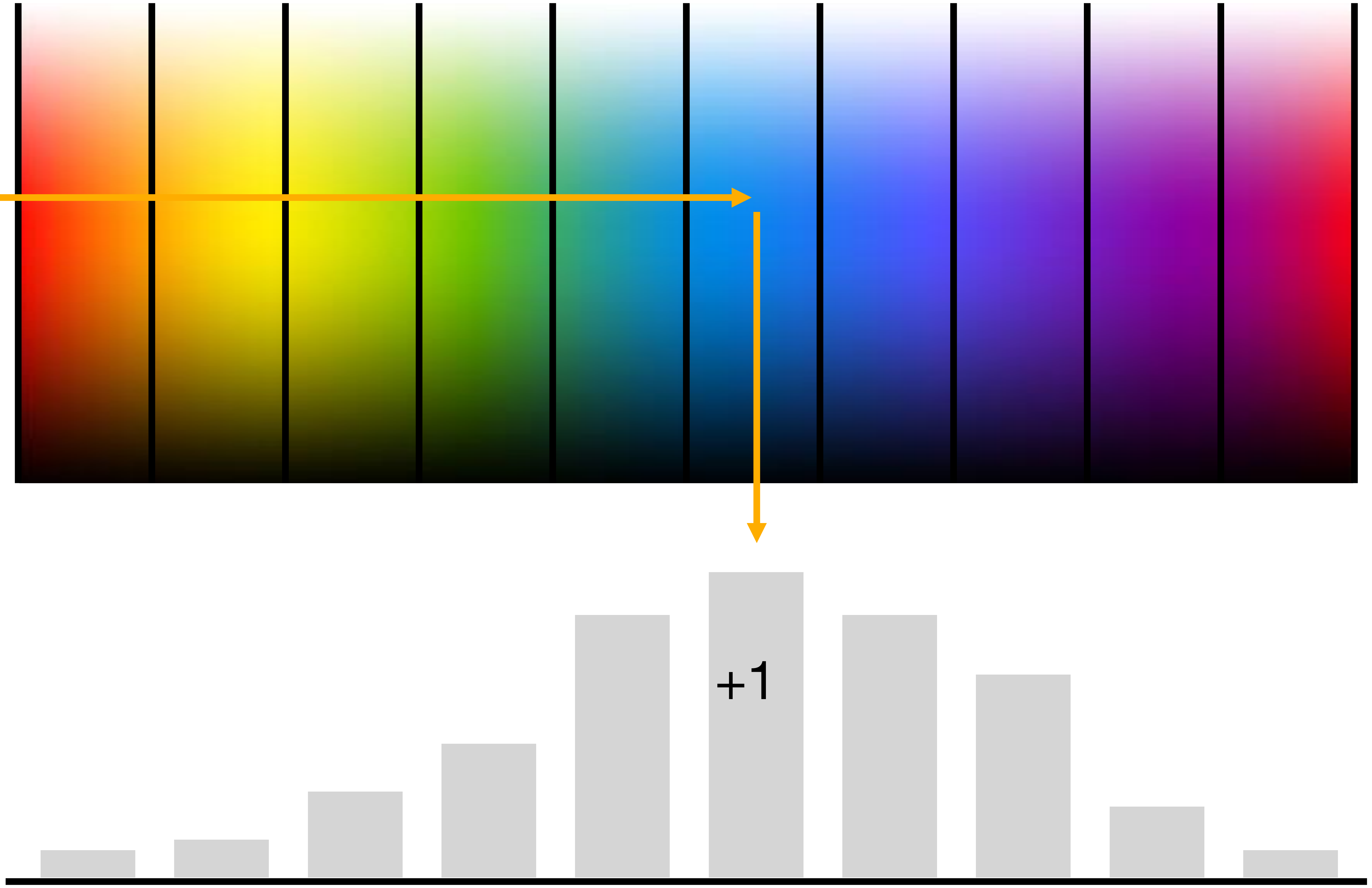


Image Features: Color Histogram



Ignores texture,
spatial positions

Frog image is in the public domain

Image Features: Color Quantization

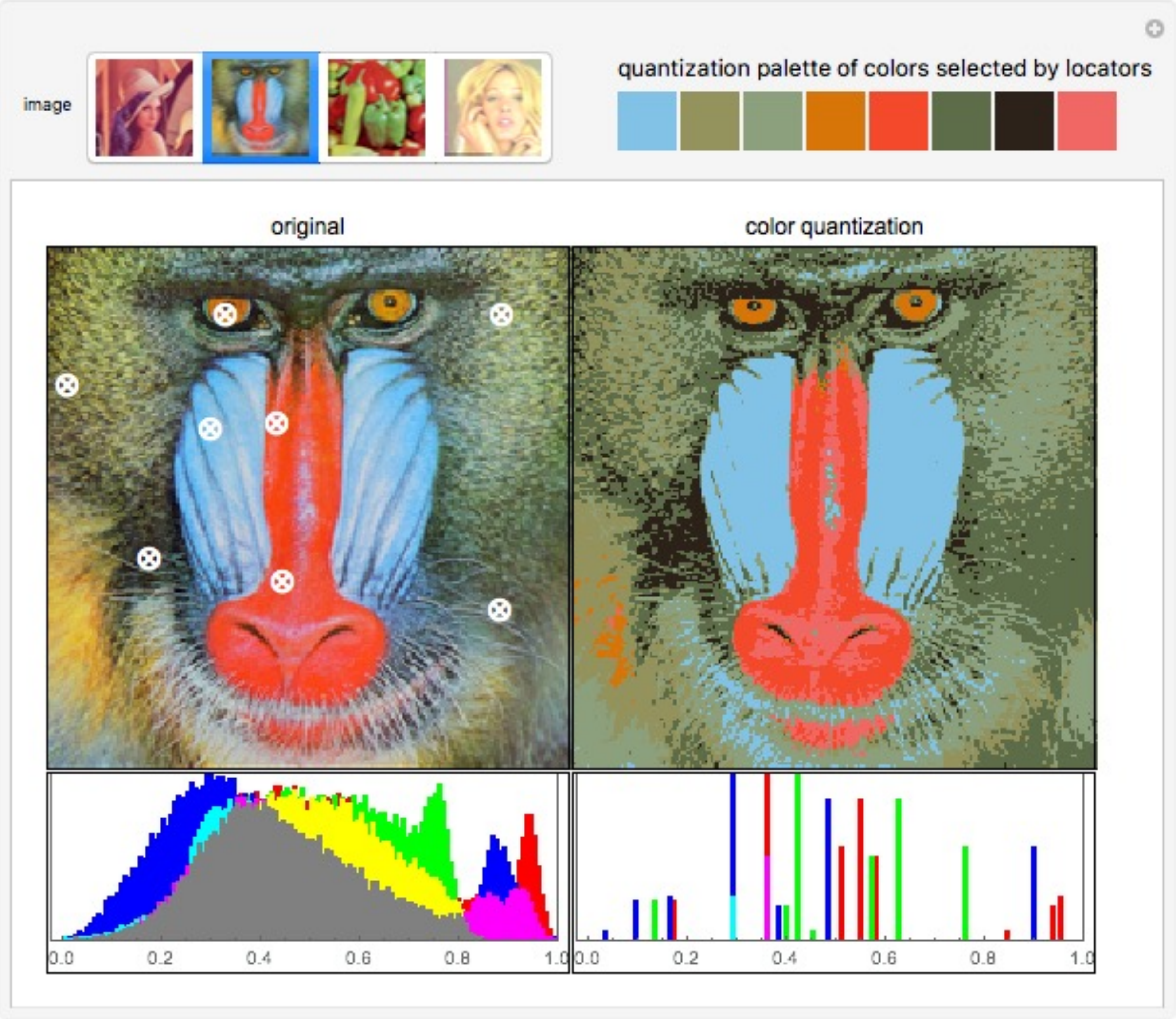
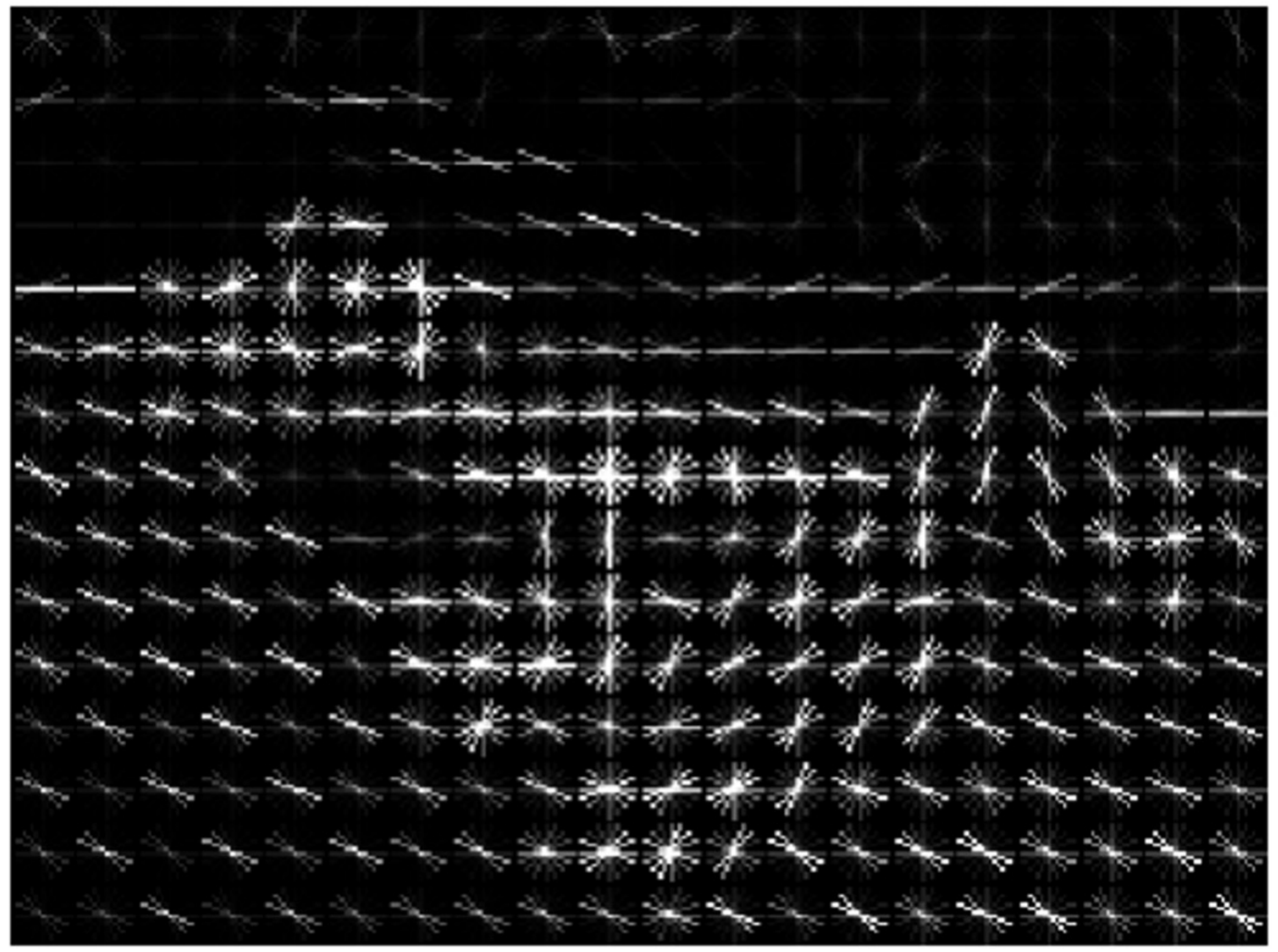


Image Features: Histogram of Oriented Gradients (HoG)



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

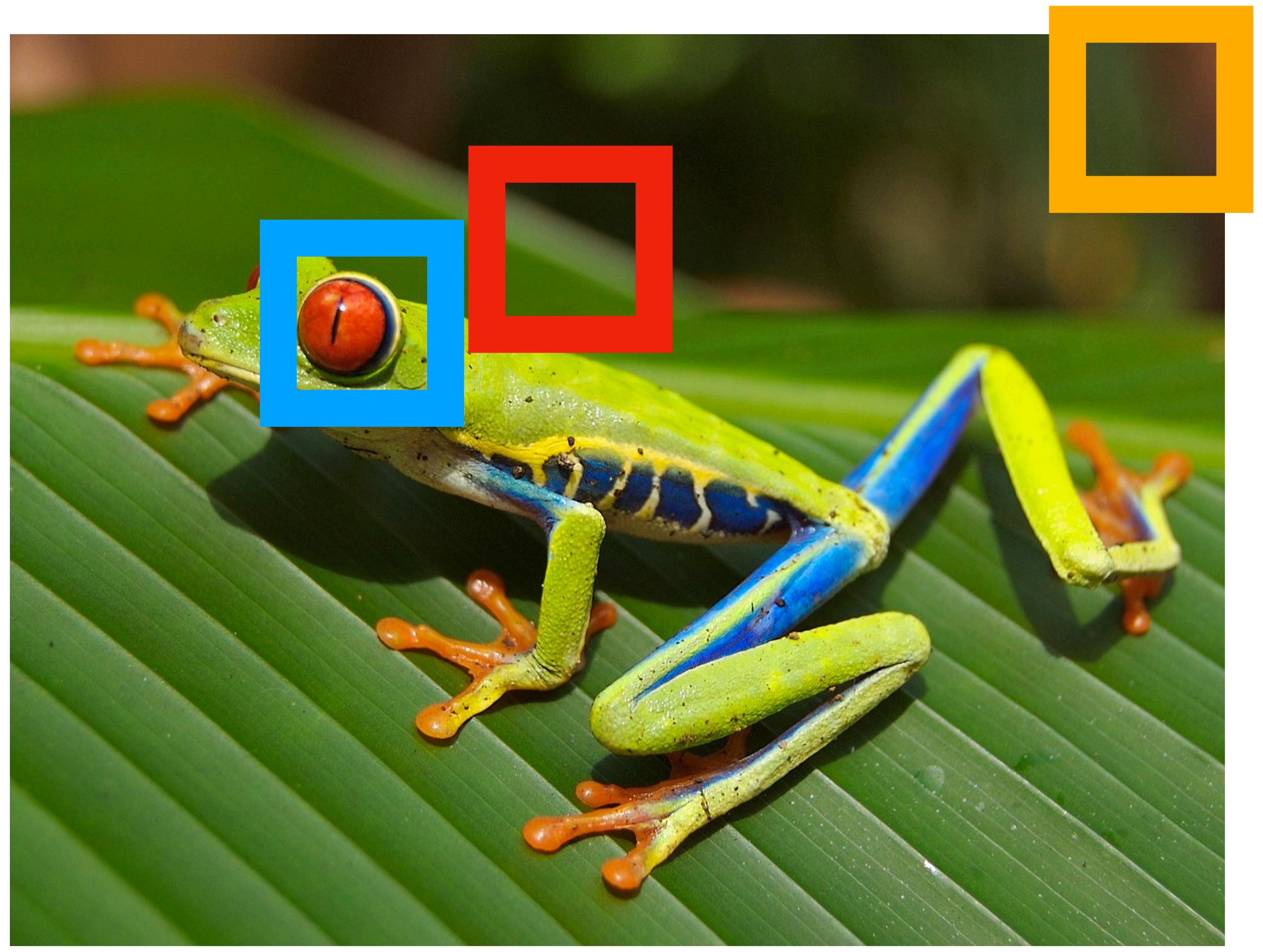
Example: 320x240 image gets divided into 40x30 bins;

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

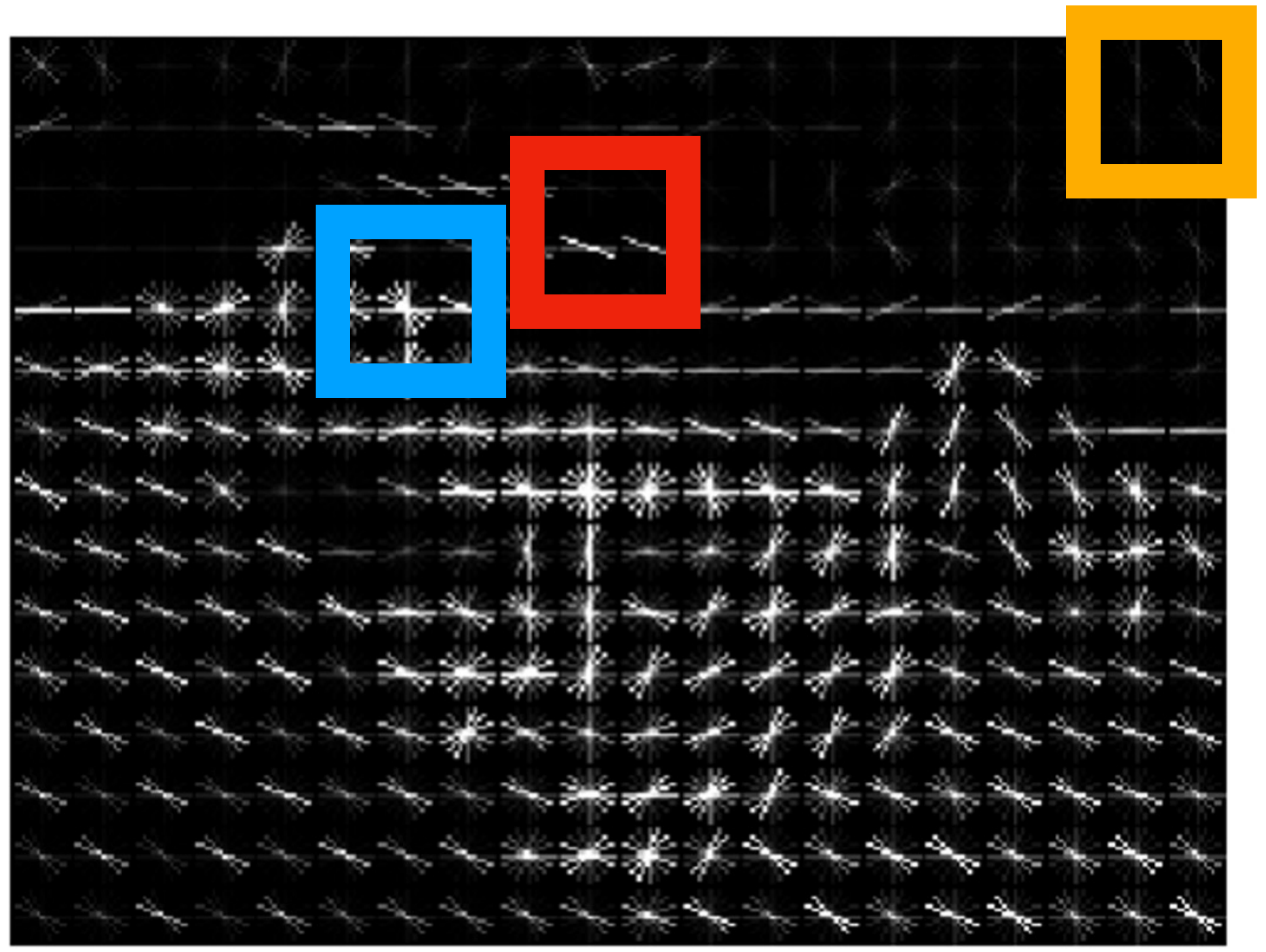
Image Features: Histogram of Oriented Gradients (HoG)



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

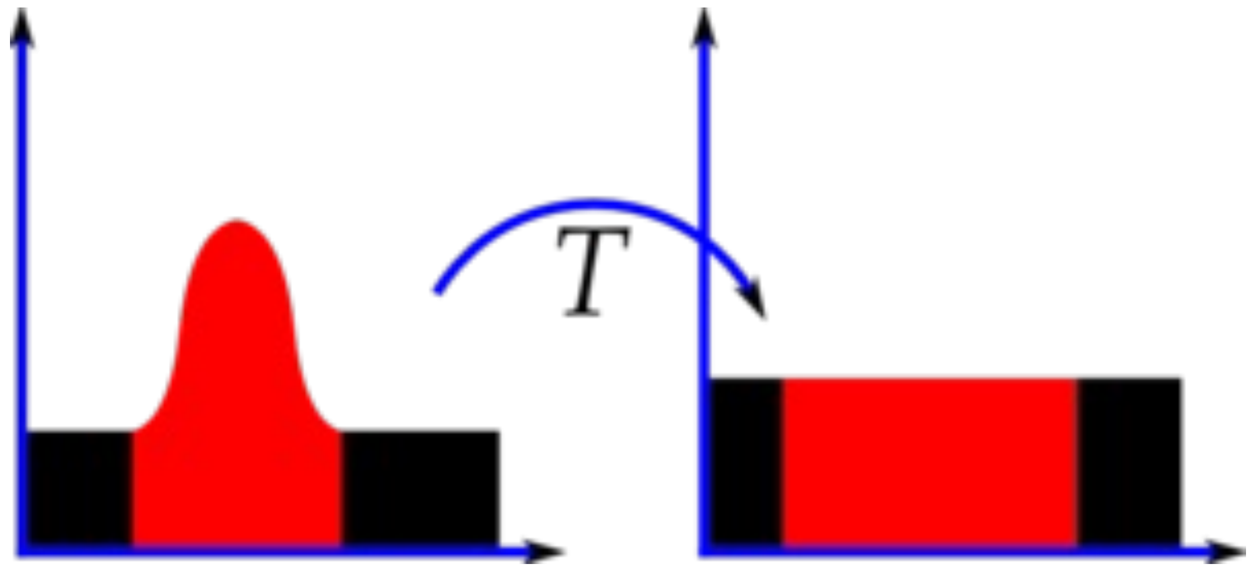
Capture texture and position, robust to small image changes

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

9 directions per bin;

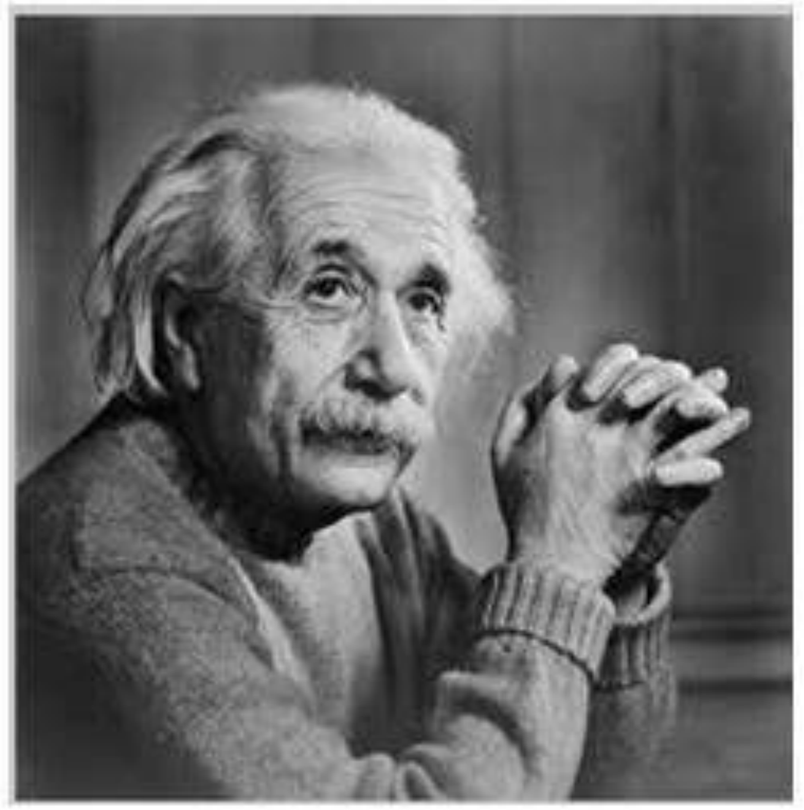
feature vector has 30*40*9 = 10,800 numbers

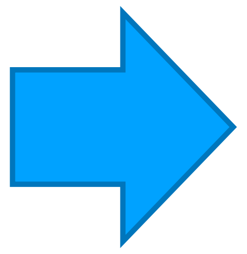
Image Features: Histogram Equalization



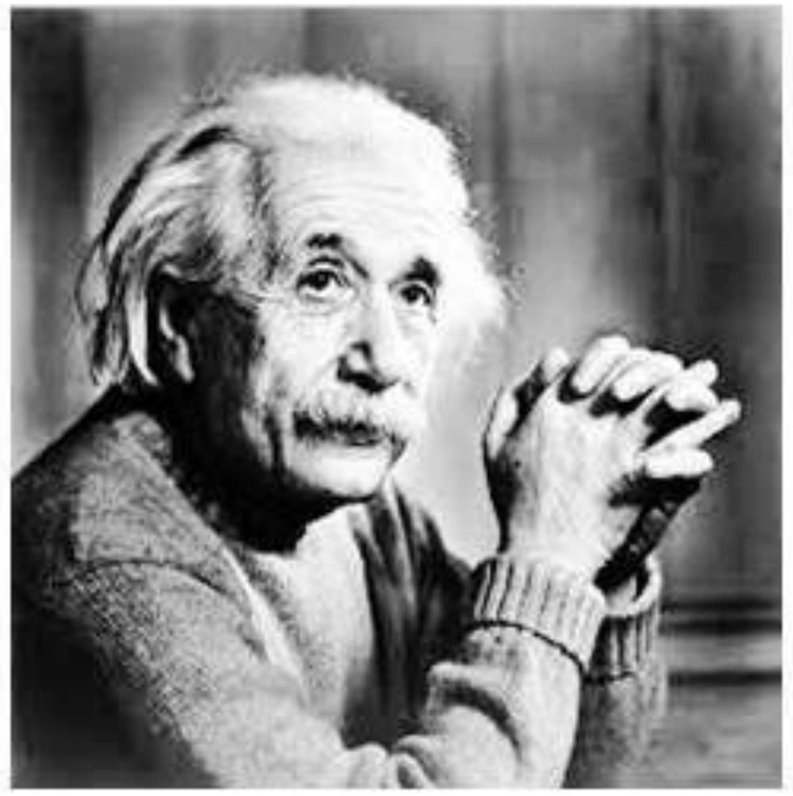
torchvision.transforms.functional.equalize

Old image

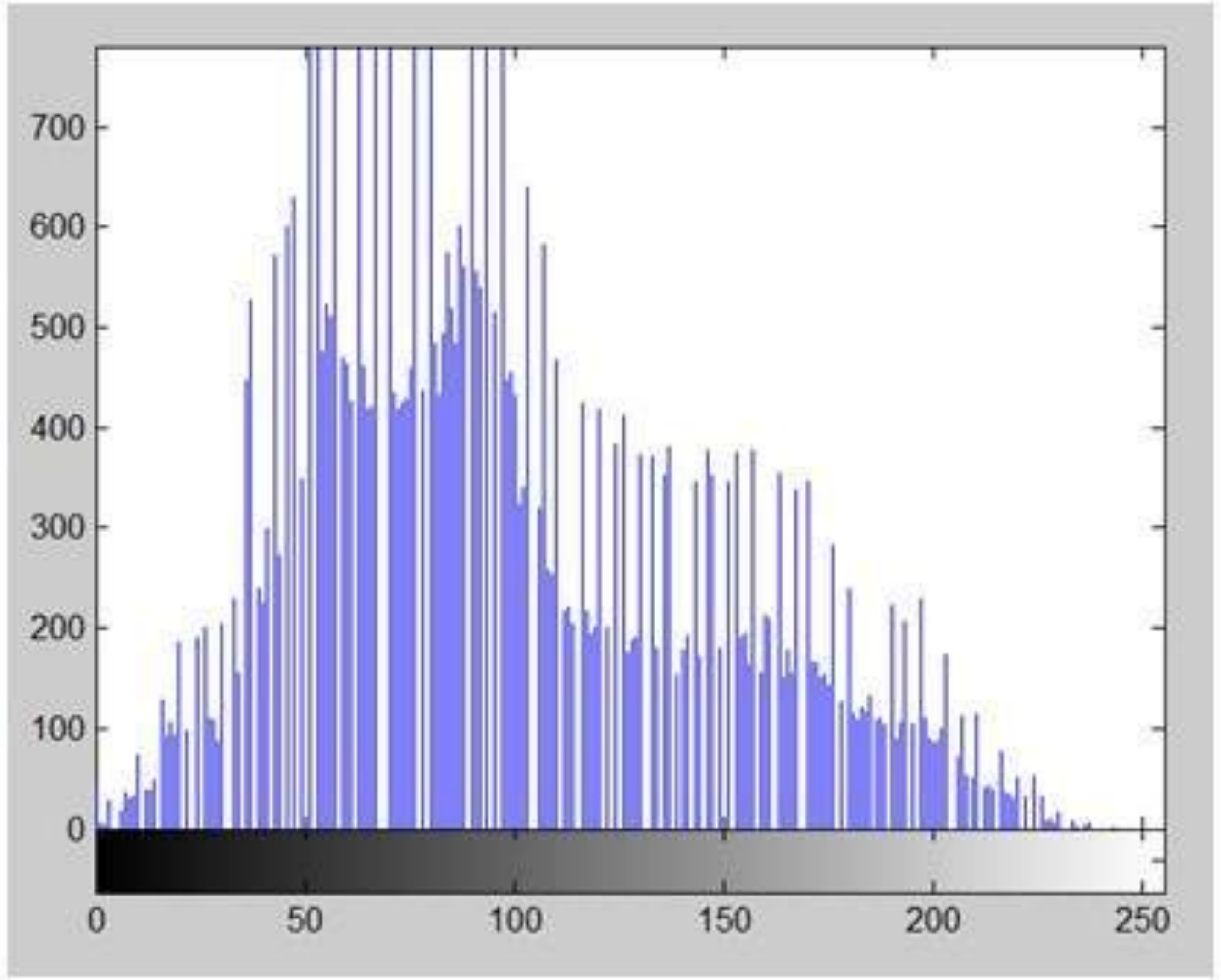




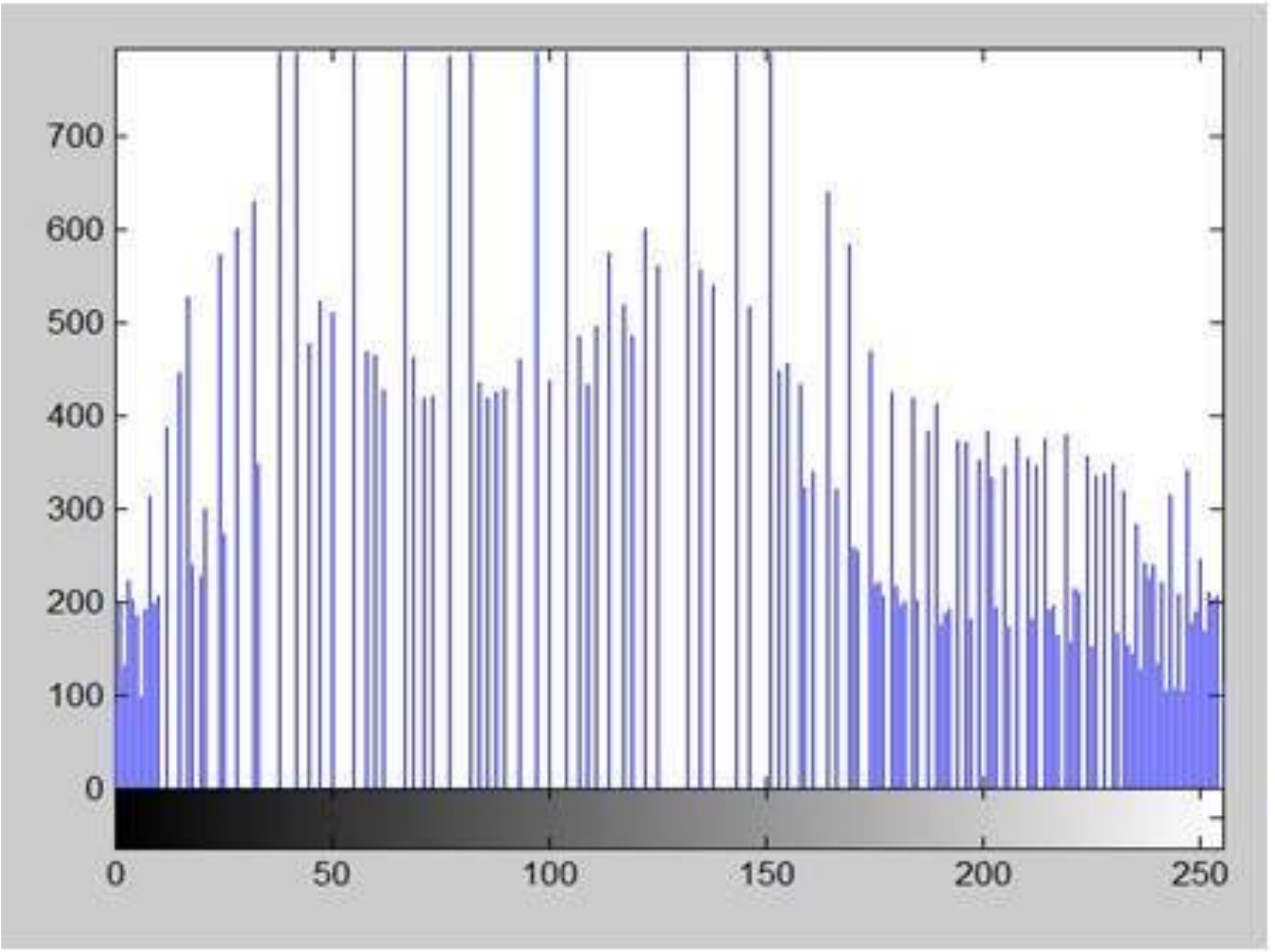
New Image



Old Histogram



New Histogram

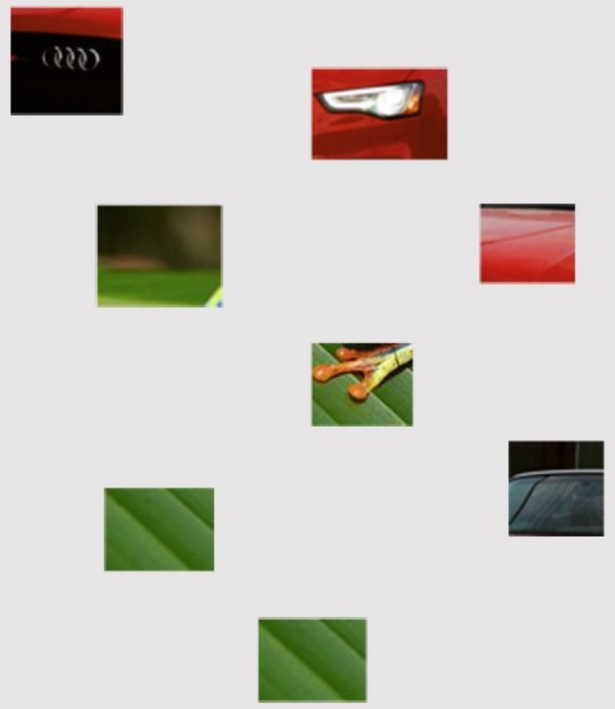


DeepRob

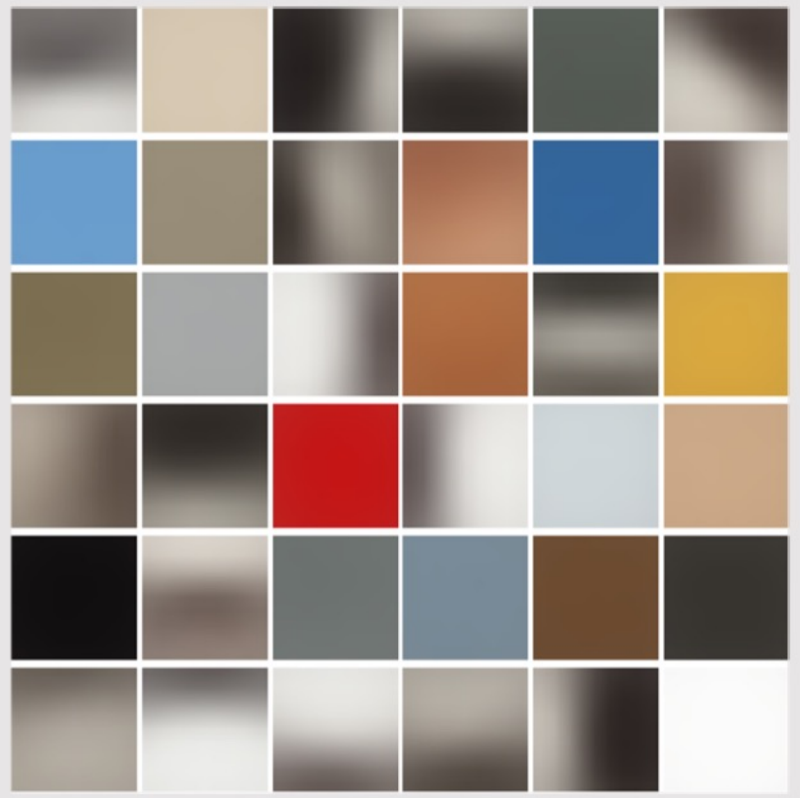
Image Features: Bag of Words (Data-Driven!)

Step 1: Build codebook

Extract random patches



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

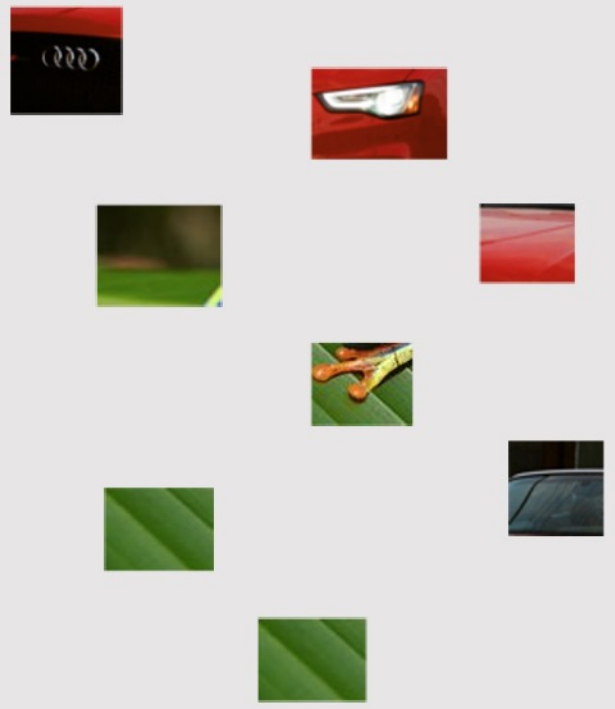


Fei-Fei and Perona, “A bayesian hierarchical model for learning natural scene categories,” CVPR 2005

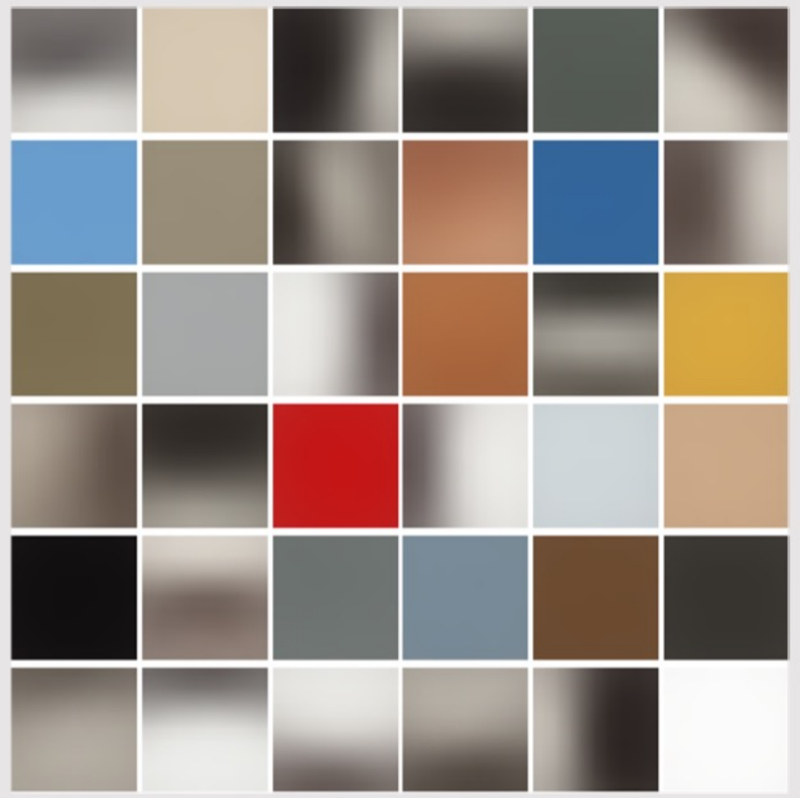
Image Features: Bag of Words (Data-Driven!)

Step 1: Build codebook

Extract random patches



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



Step 2: Encode images

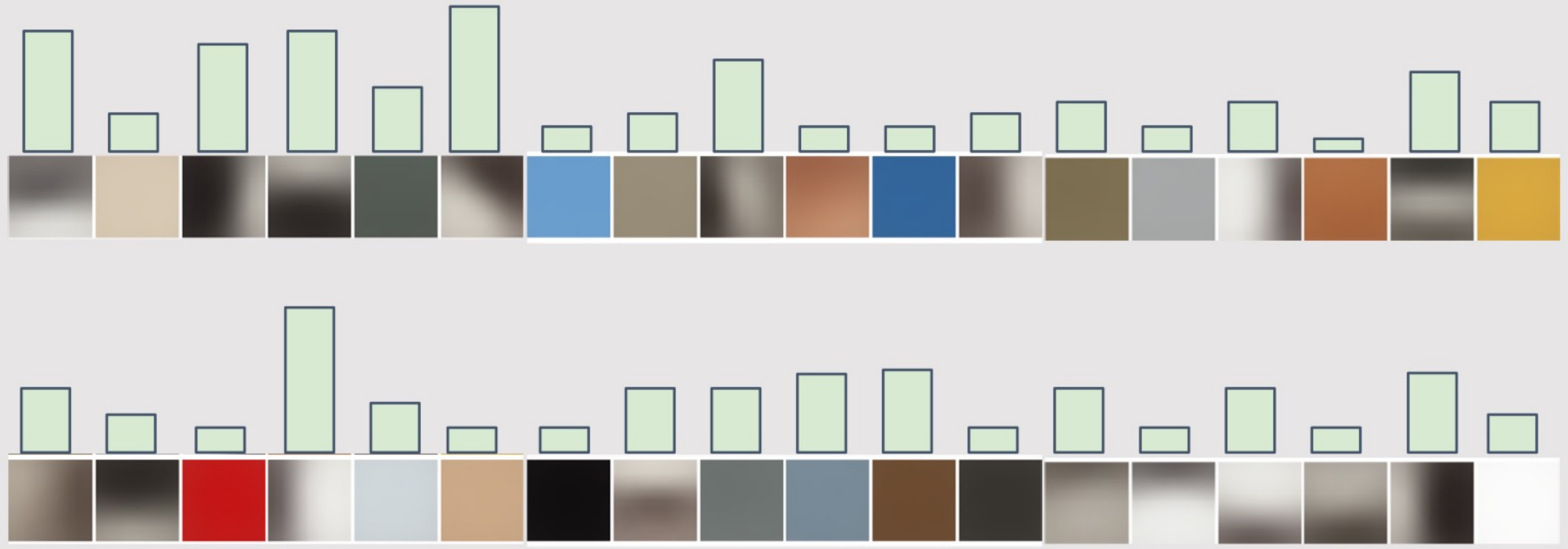
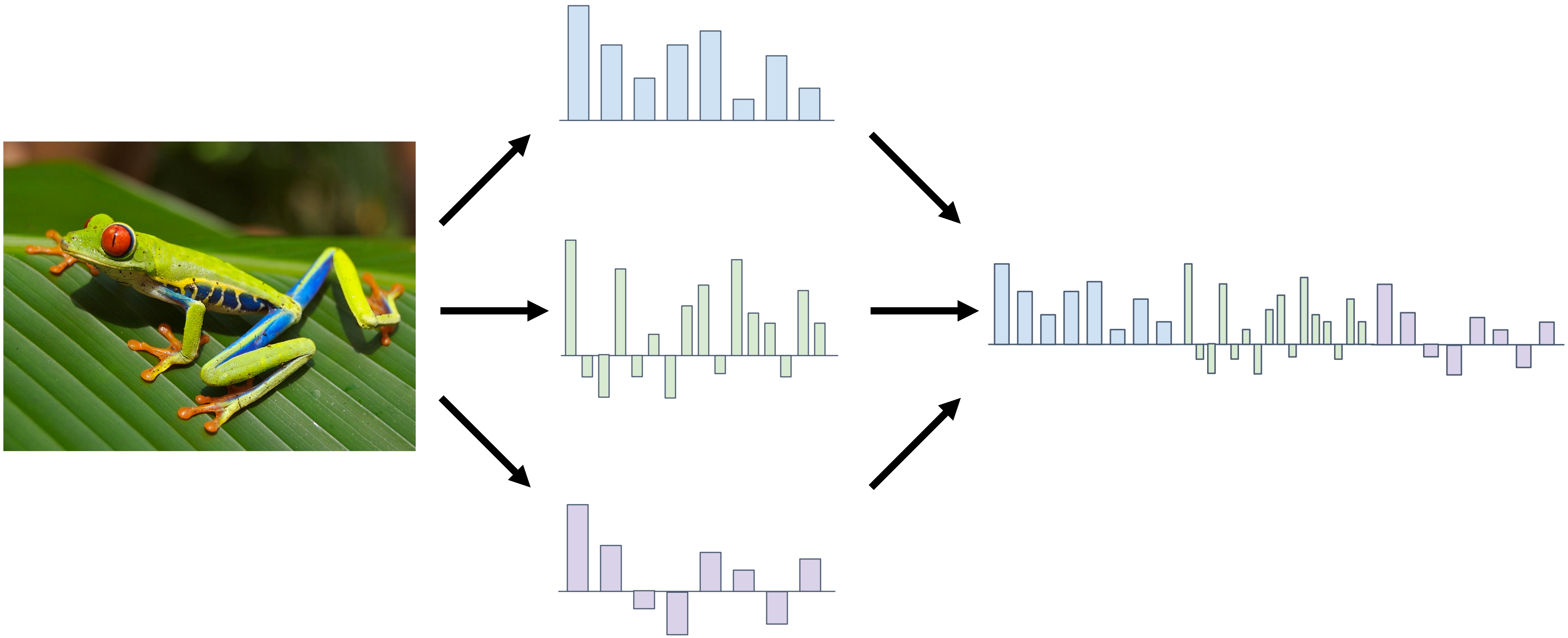


Image Features



Example: Winner of 2011 ImageNet Challenge

Low-level feature extraction ≈ 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 → 520K dim x 2
- Compression: G=8, b=1 bit per dimension

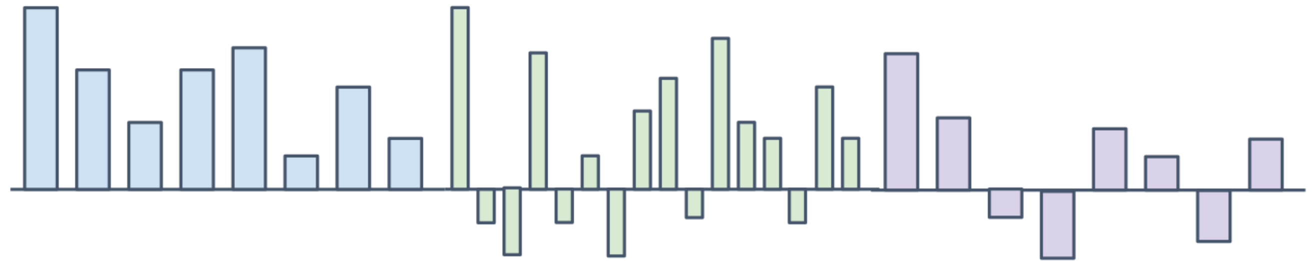
One-vs-all SVM learning with SGD

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



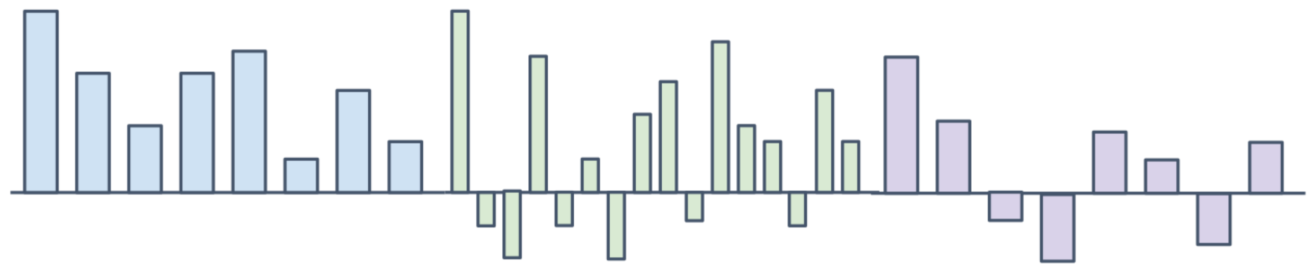
f

10 numbers giving scores for classes

training

Image Features vs Neural Networks

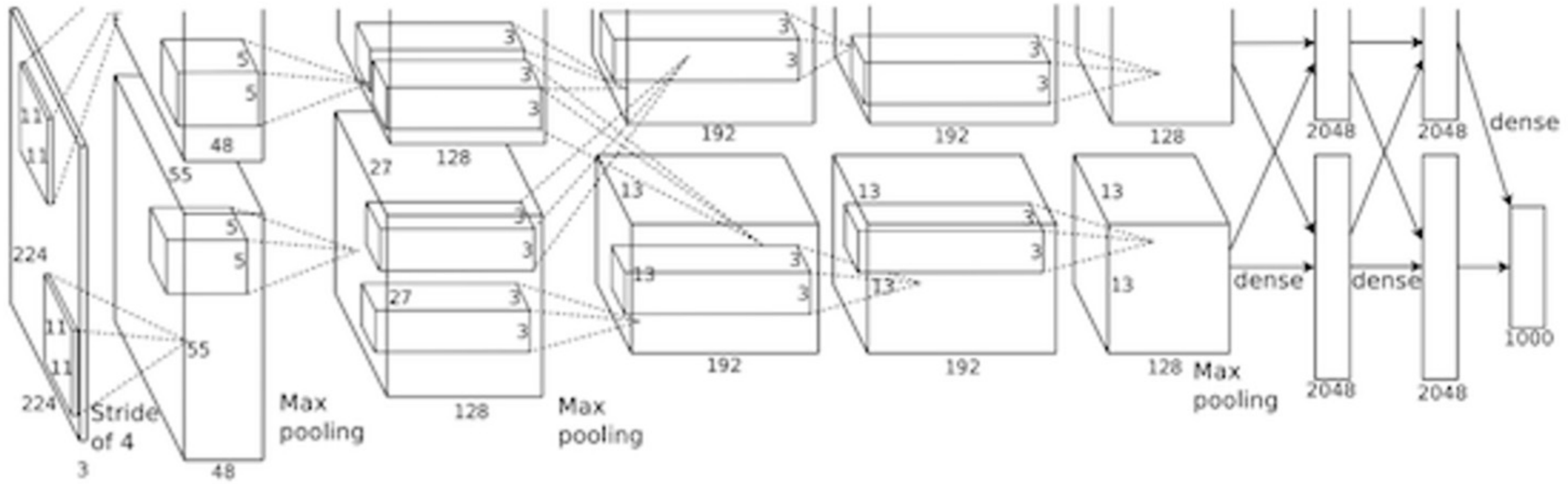
Feature Extraction



f

10 numbers giving scores for classes

training



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

10 numbers giving scores for classes

"Trained from data"

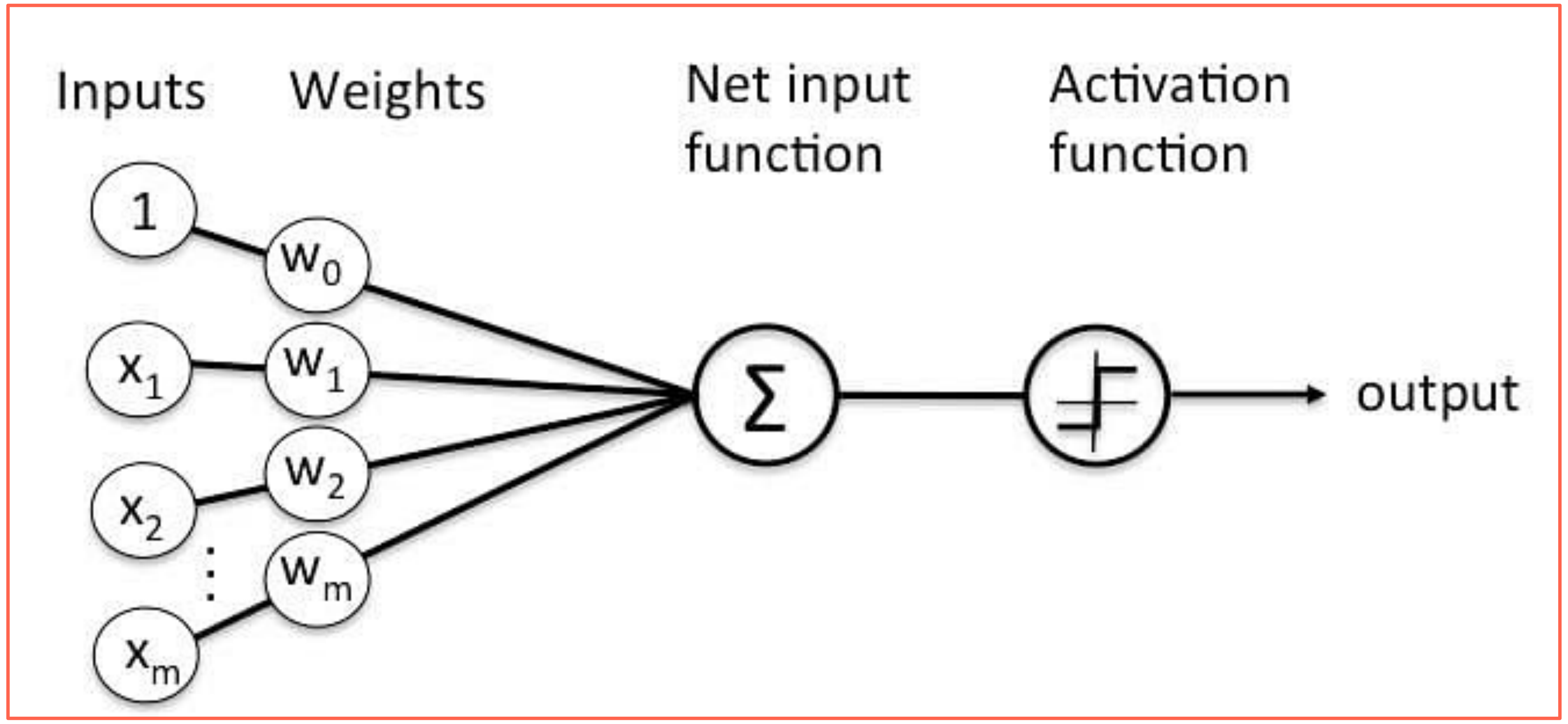
training

Neural Networks

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

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

Rosenblatt's Perceptron



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

•An *adder*

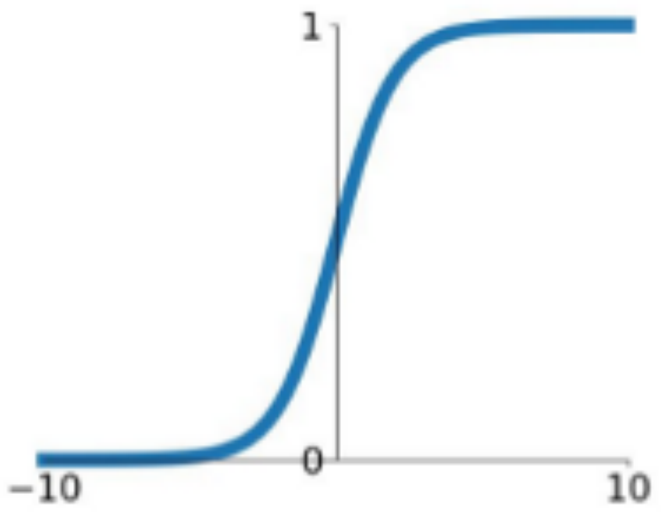
•An *activation function* (e.g., Rectified Linear Unit/ReLU, Sigmoid function, etc.)

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

Activation Function

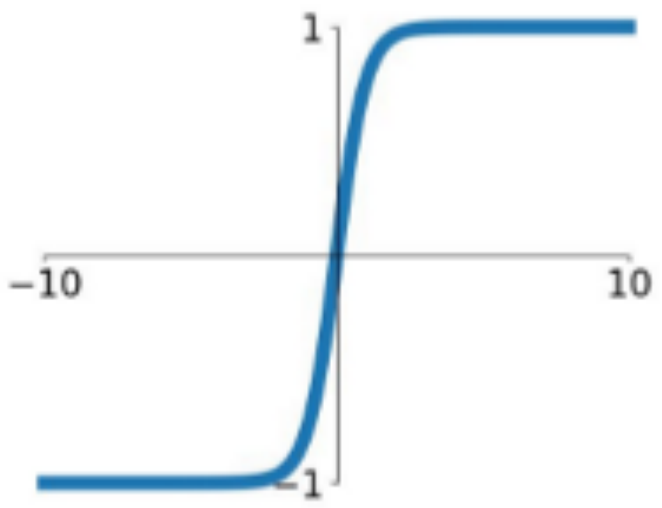
Sigmoid

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



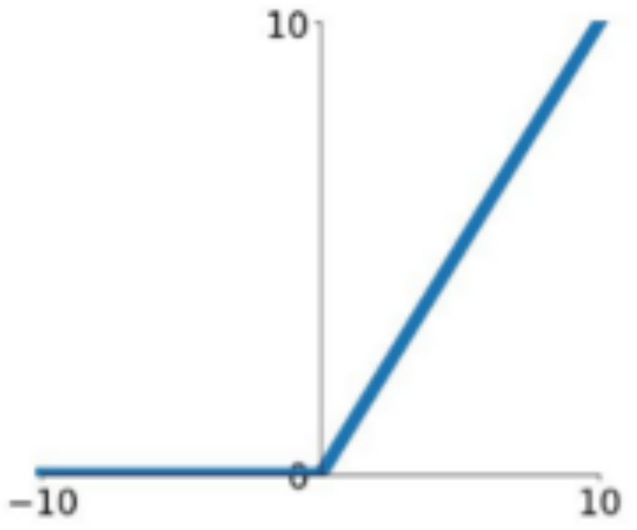
tanh

$\tanh(x)$



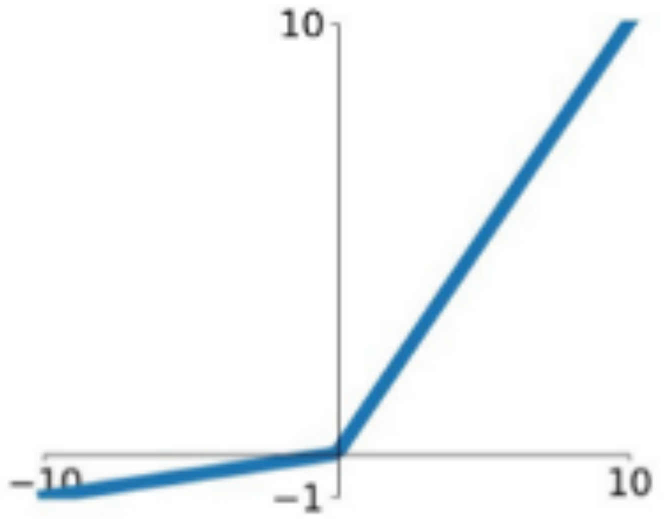
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

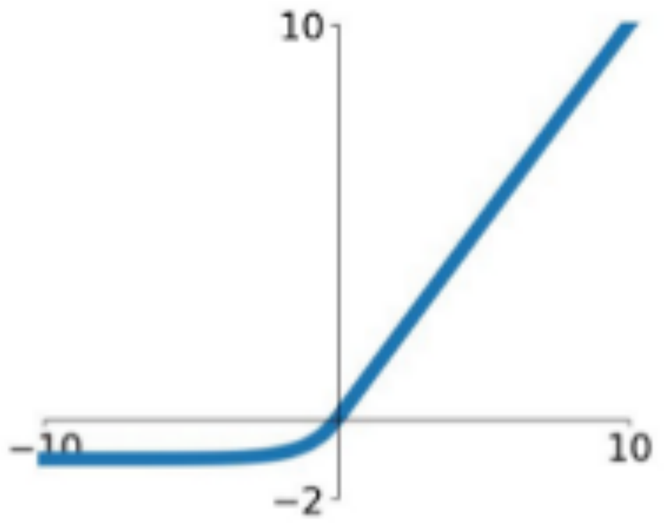


Maxout

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

ELU

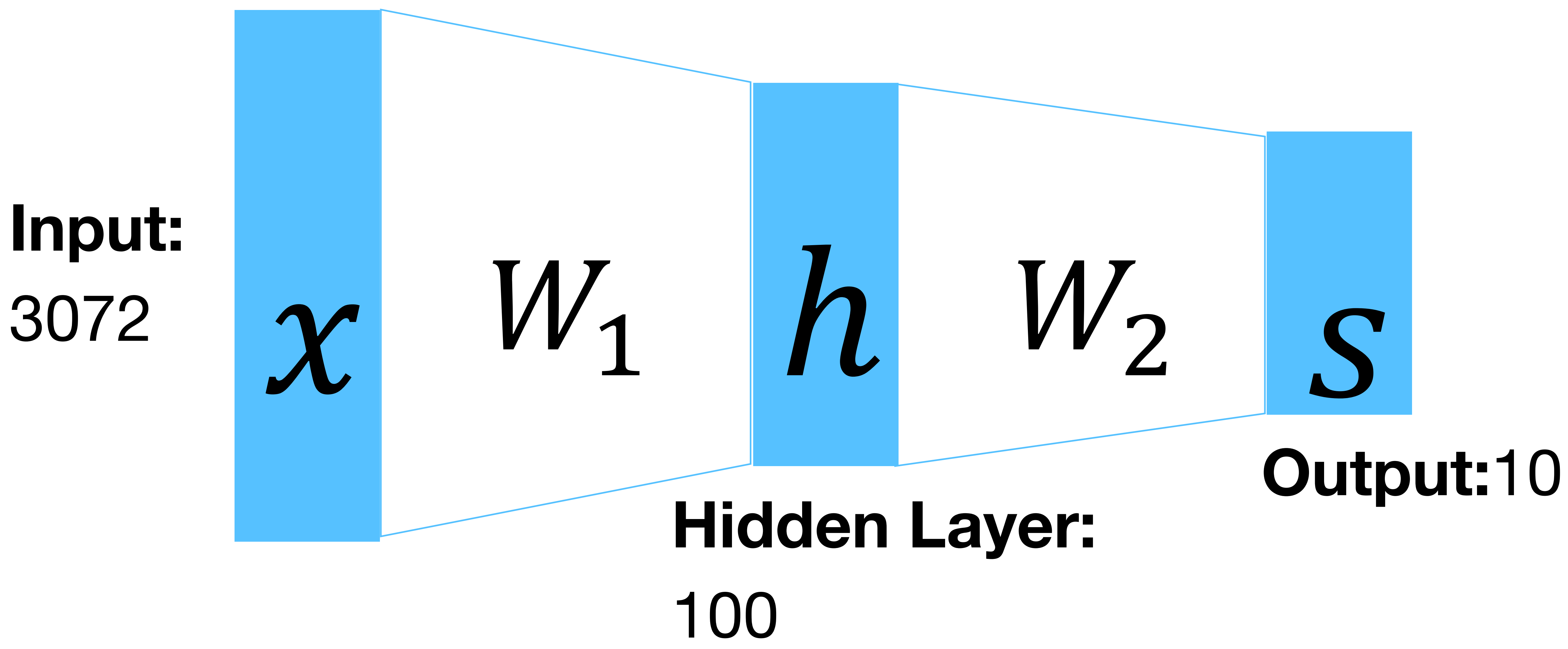
$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$



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

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

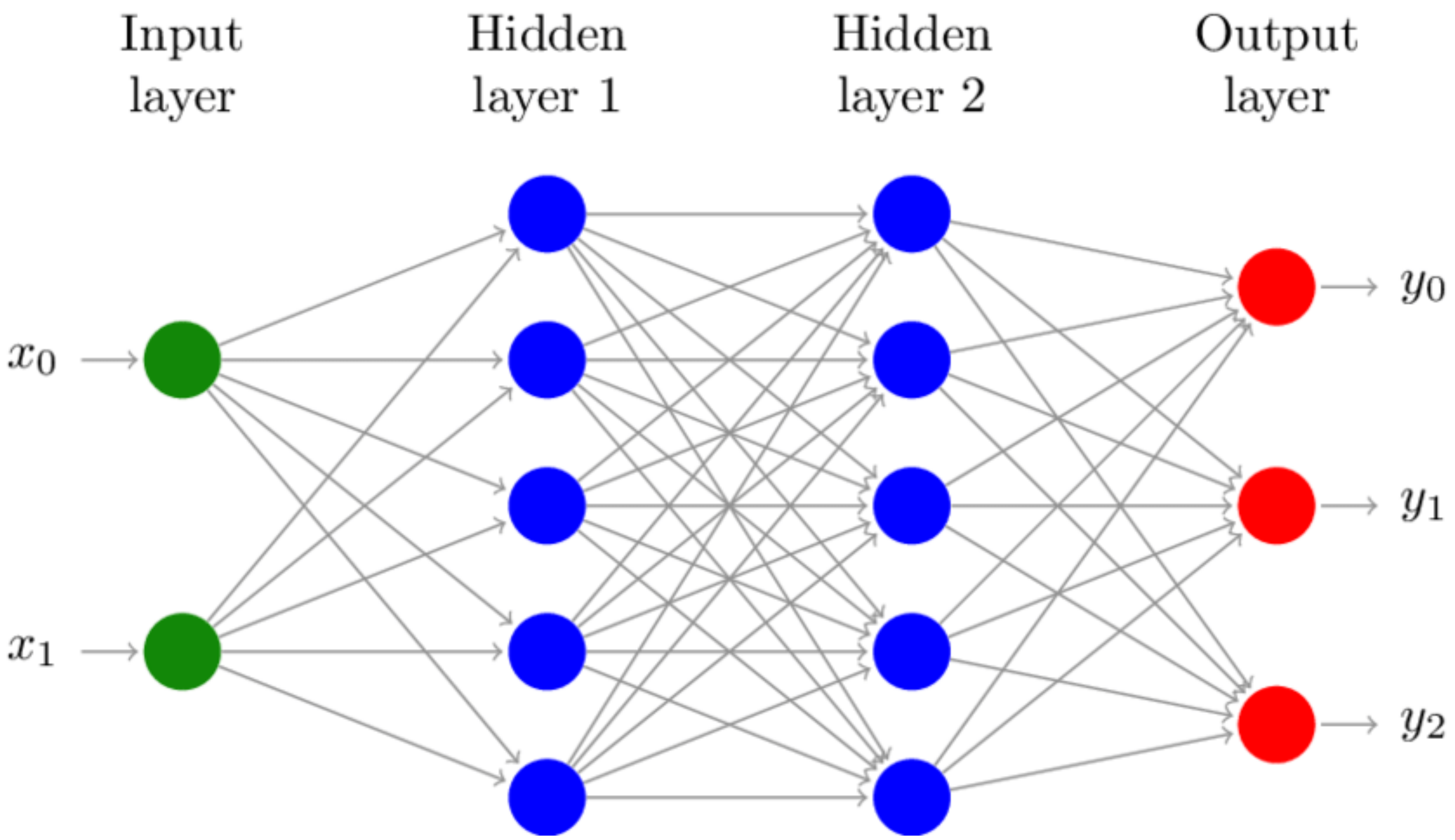
Learnable parameters: $W_1 \in \mathbb{R}^{H\times D}, b_1 \in \mathbb{R}^H, W_2 \in \mathbb{R}^{C\times H}, b_2 \in \mathbb{R}^C$

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 – MLP

"Fully Connected"

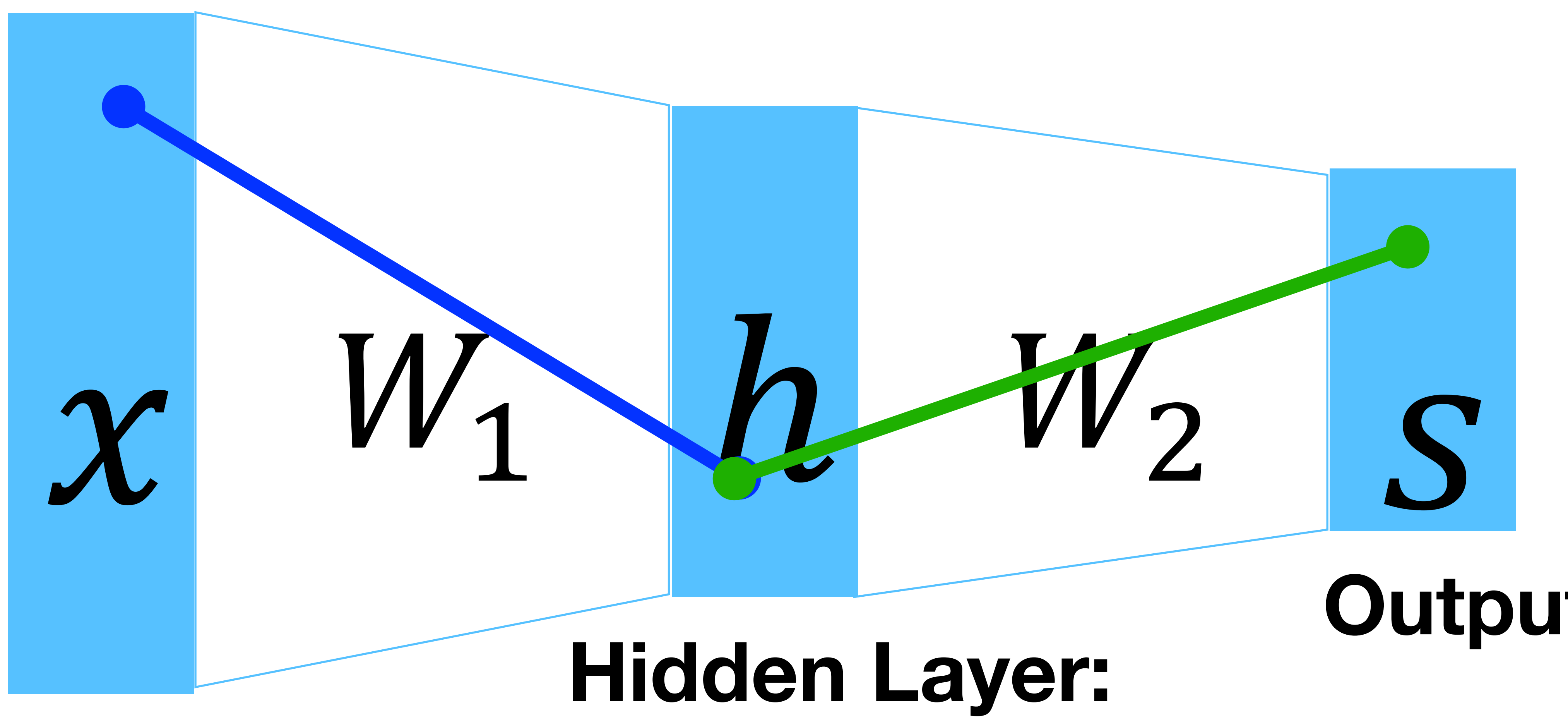


Neural Networks - MLP

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

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

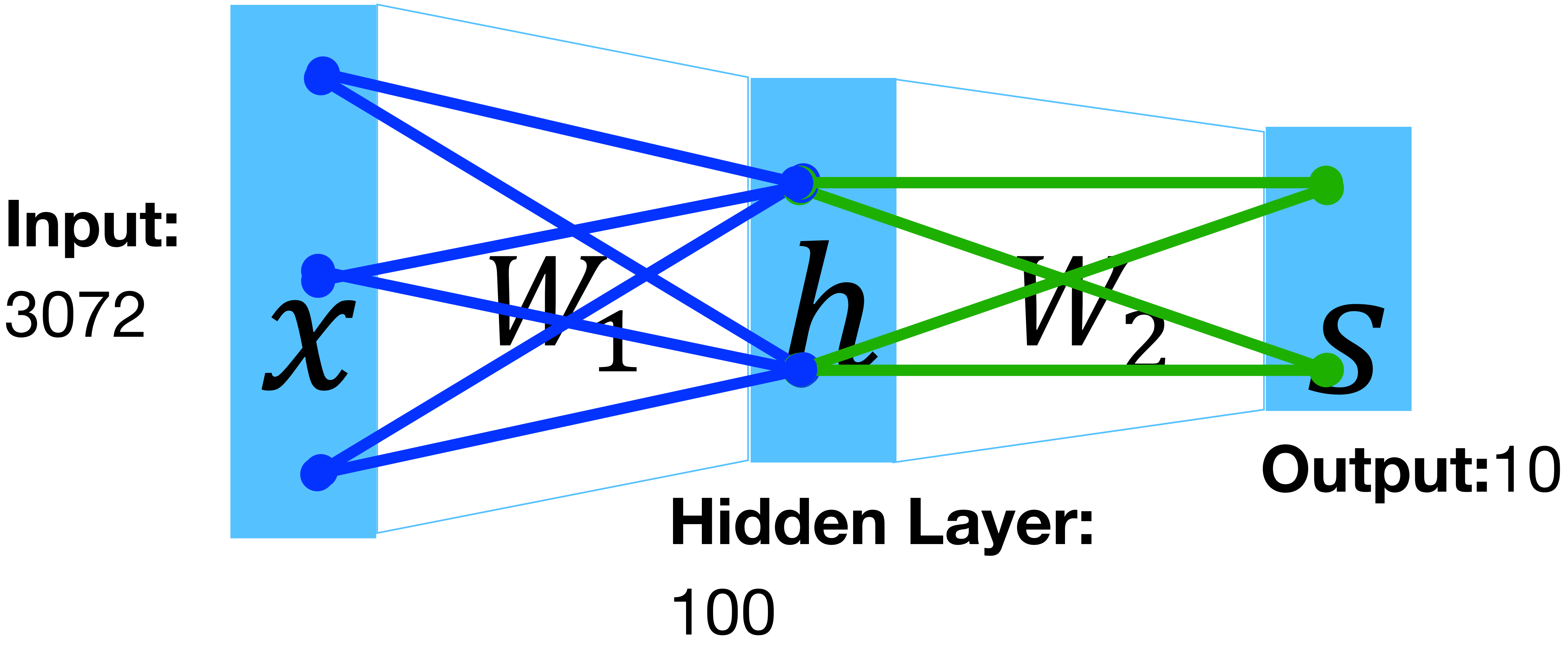
Neural Networks - MLP

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



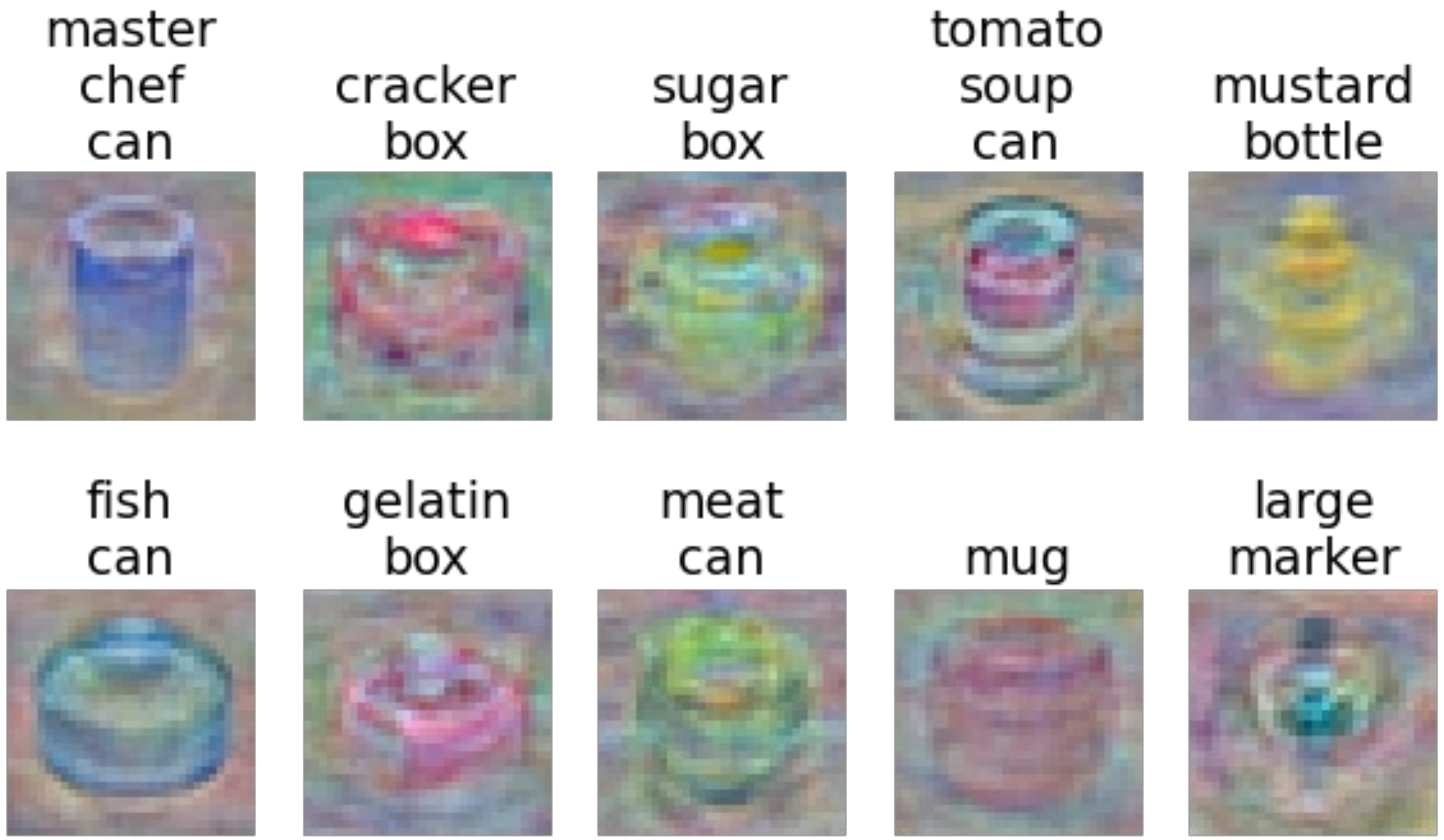
Element (i, j) of W_2 gives the effect on s_i from h_j

All elements of h affect all elements of s

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

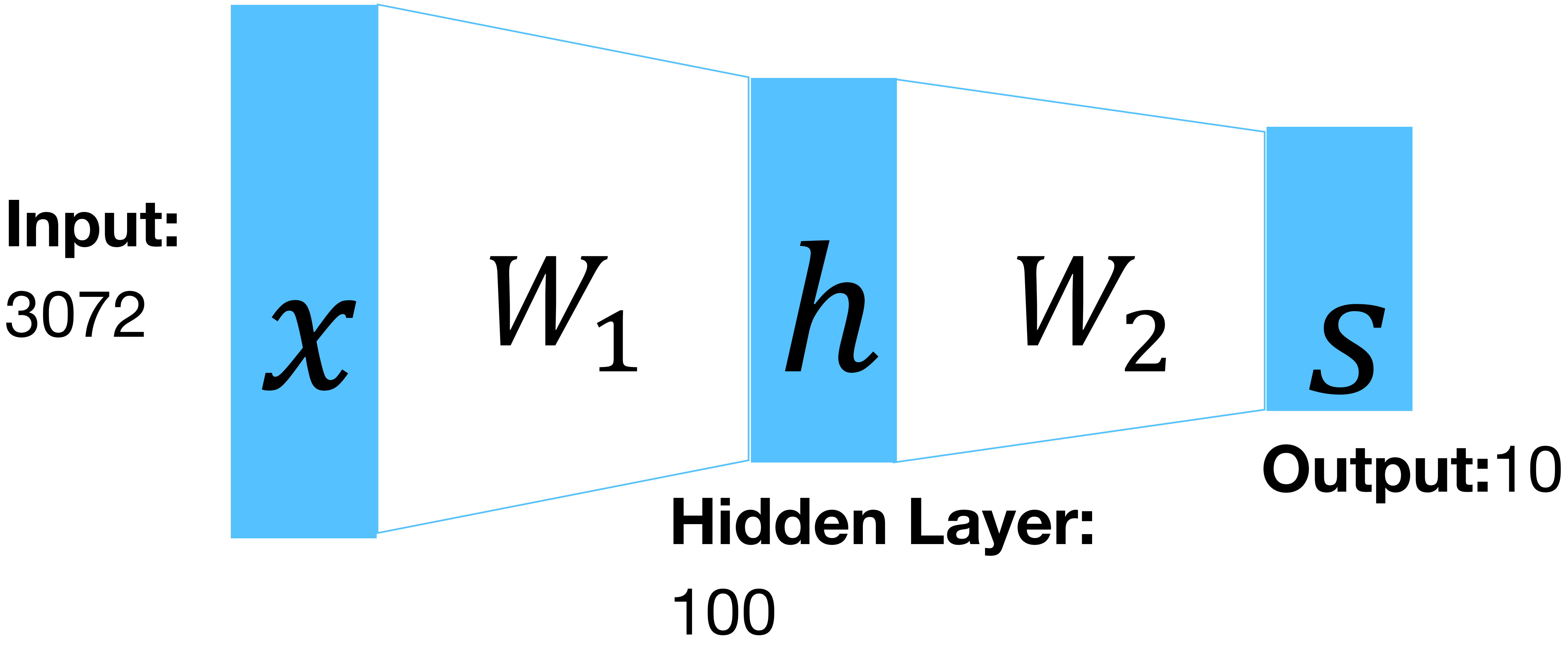
Neural Networks

Linear classifier: One template per class



Before: Linear score function

Now: Two-Layer Neural Network:



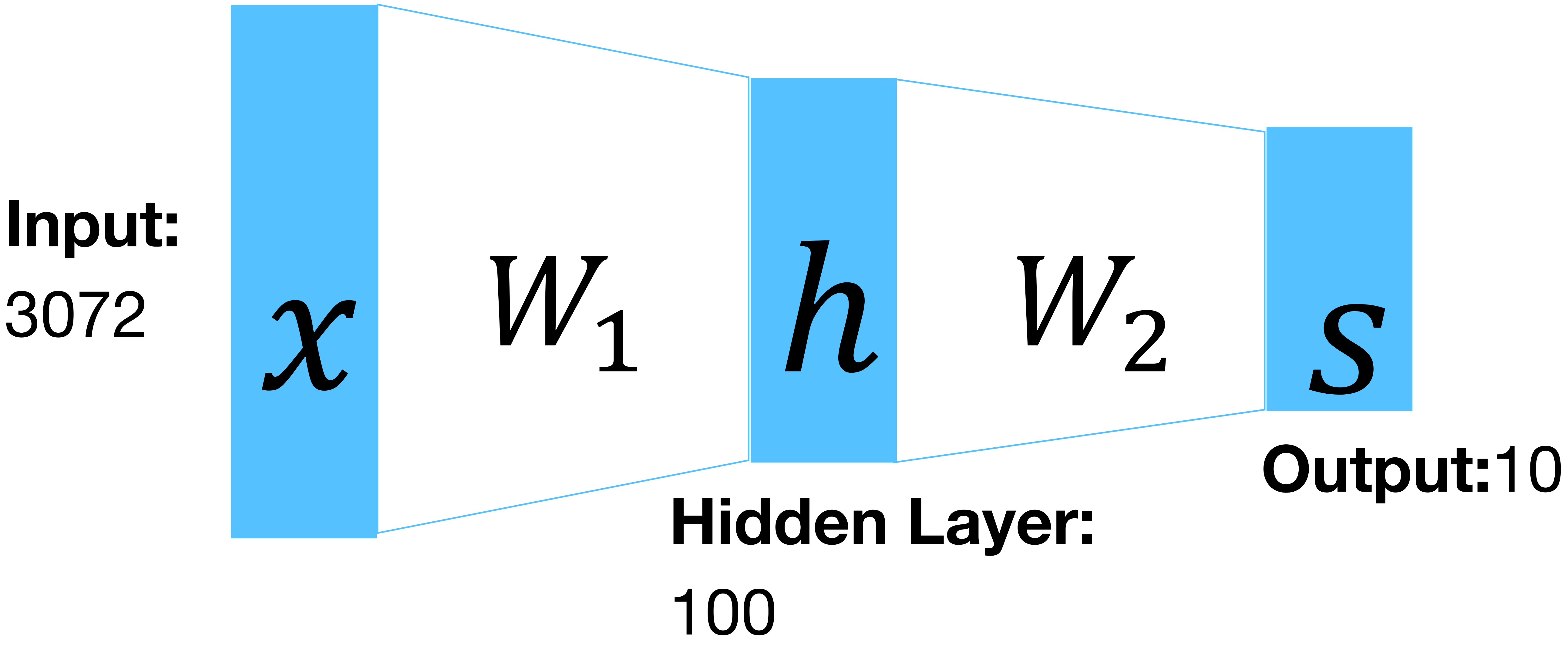
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

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

Before: Linear score function

Now: Two-Layer Neural Network:



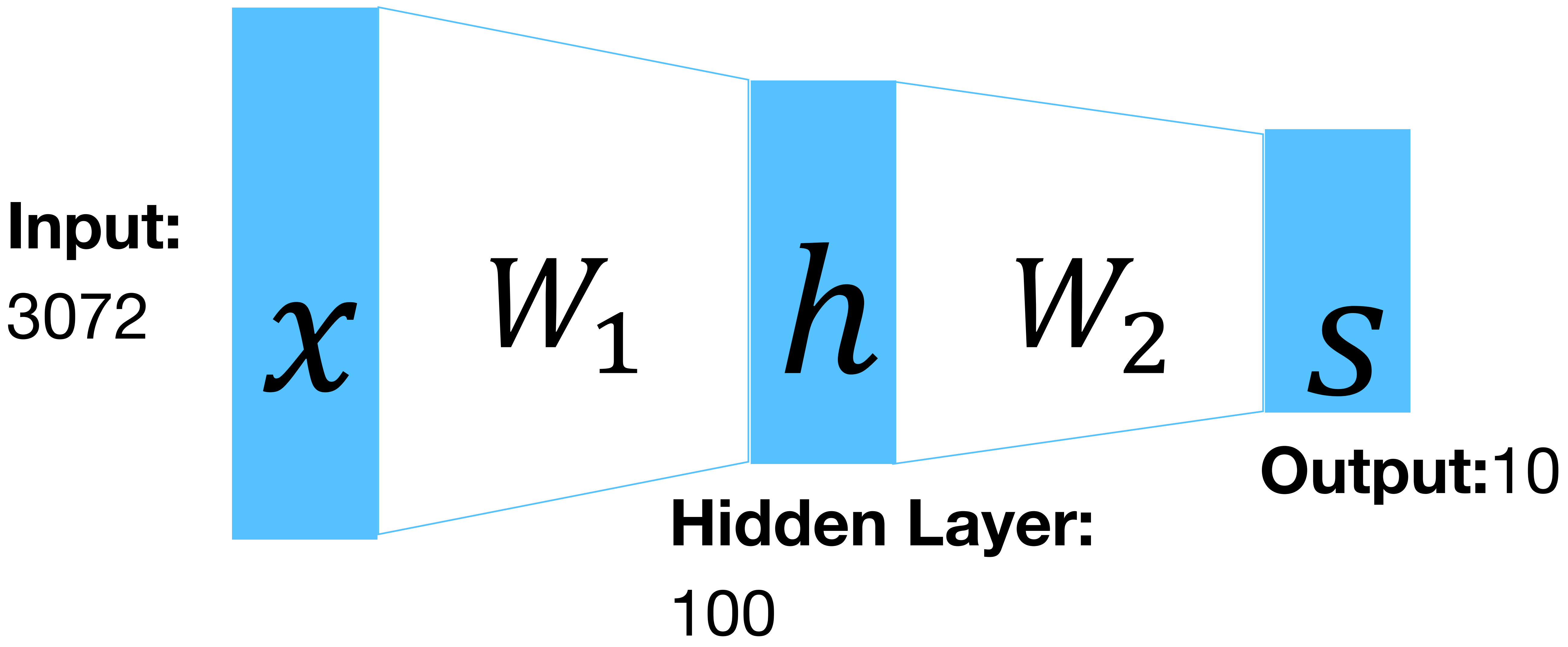
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H\times D}, W_2 \in \mathbb{R}^{C\times H}$$

Neural Networks

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

Before: Linear score function

Now: Two-Layer Neural Network:



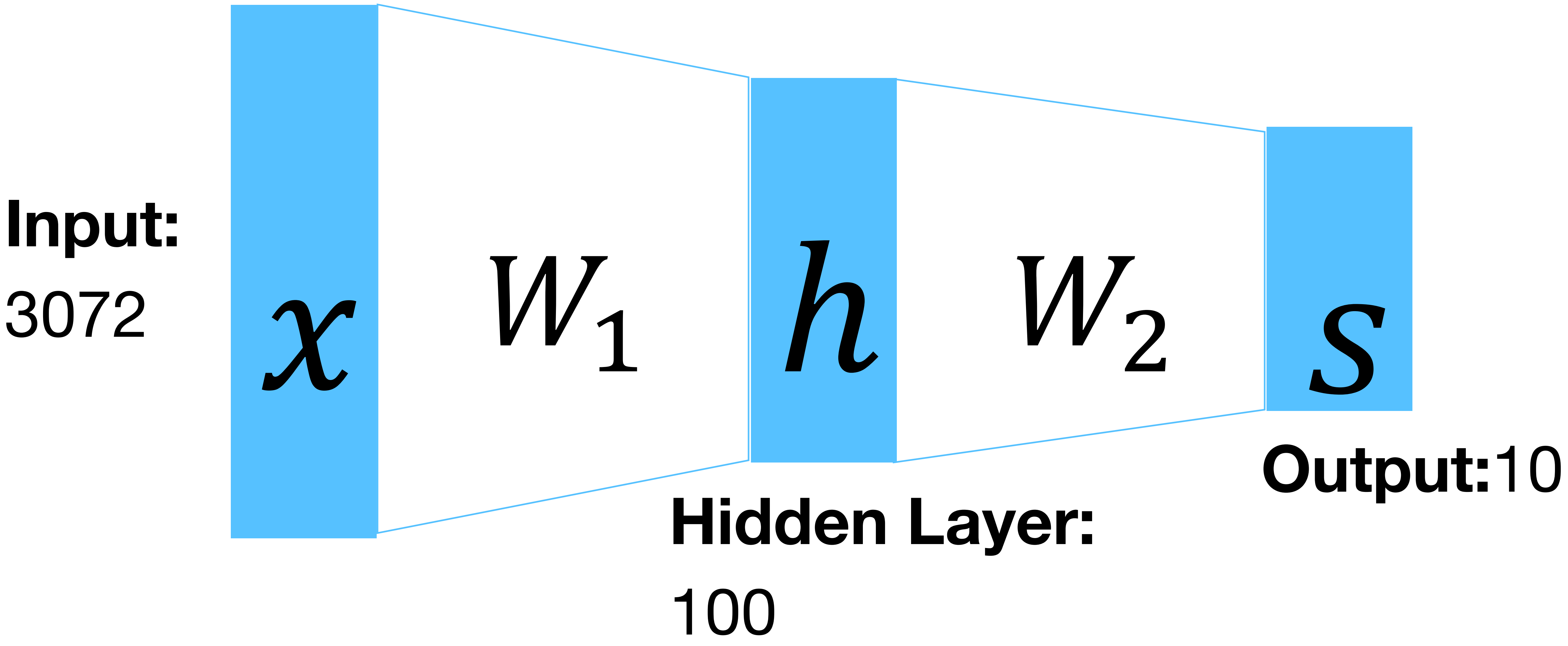
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

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

Before: Linear score function

Now: Two-Layer Neural Network:



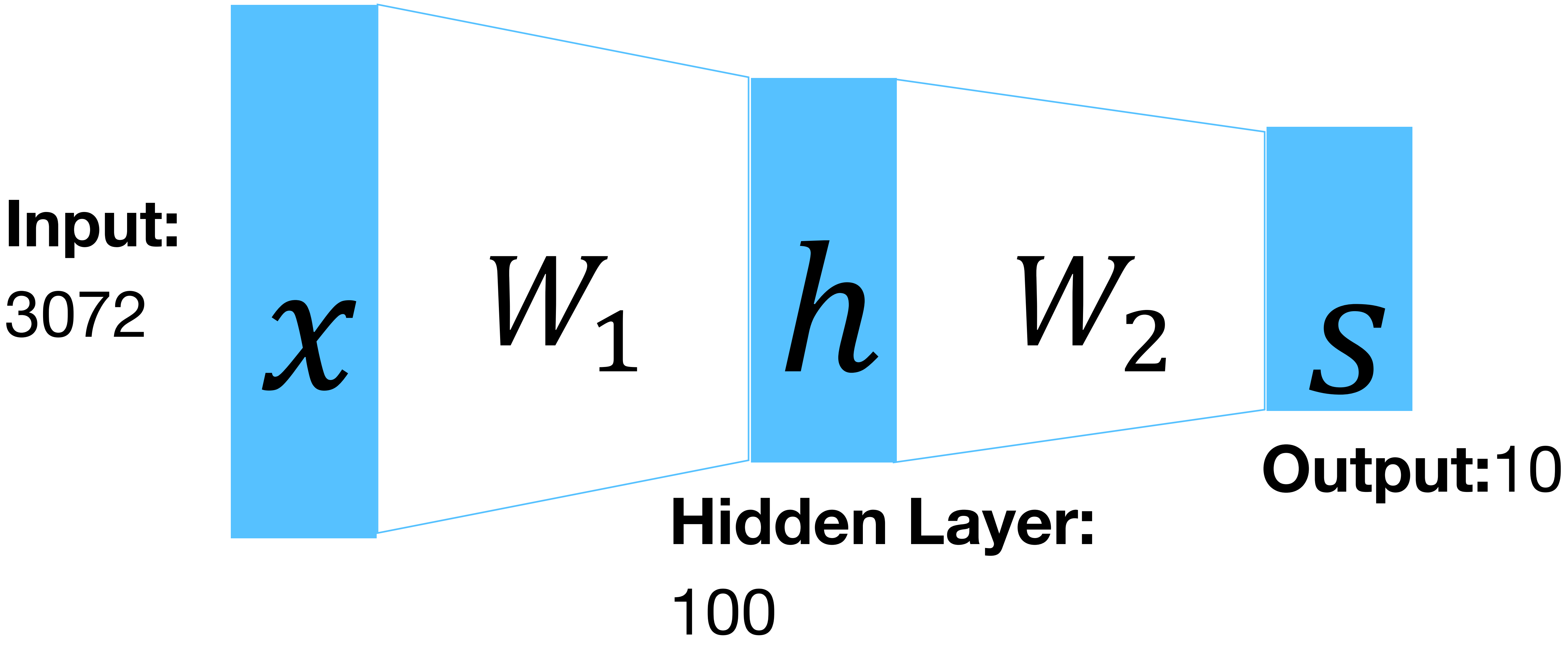
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H\times D}, W_2 \in \mathbb{R}^{C\times H}$$

Neural Networks

"Distributed representation": Most templates not interpretable!

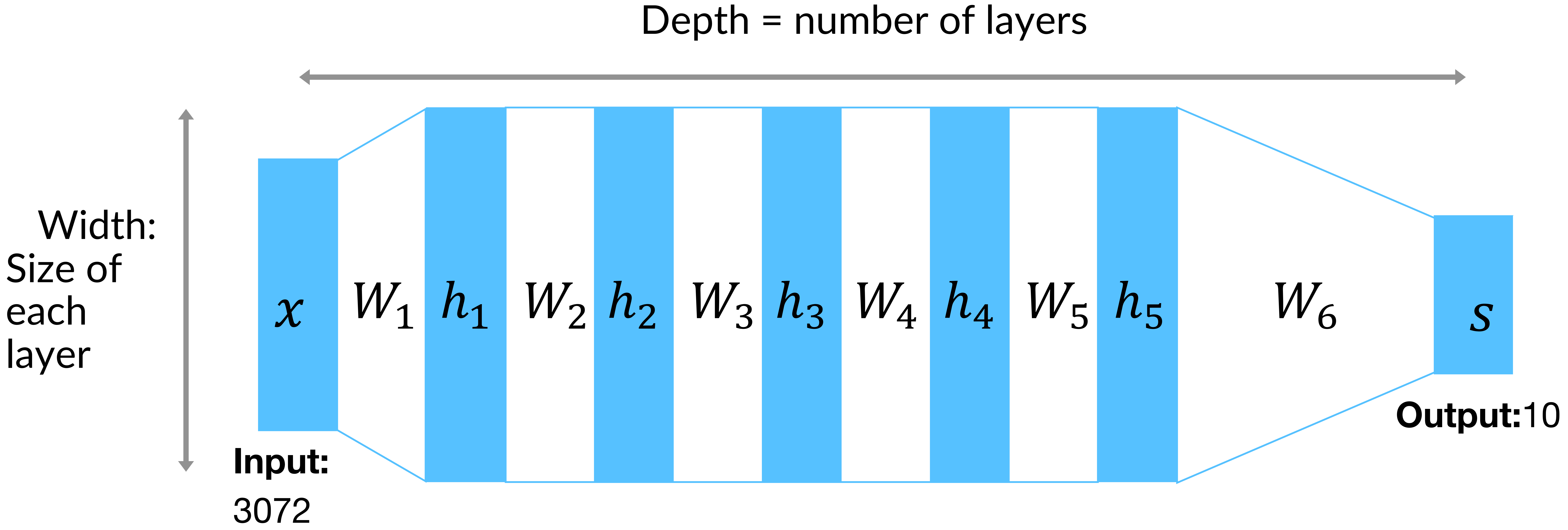
Before: Linear score function

Now: Two-Layer Neural Network:



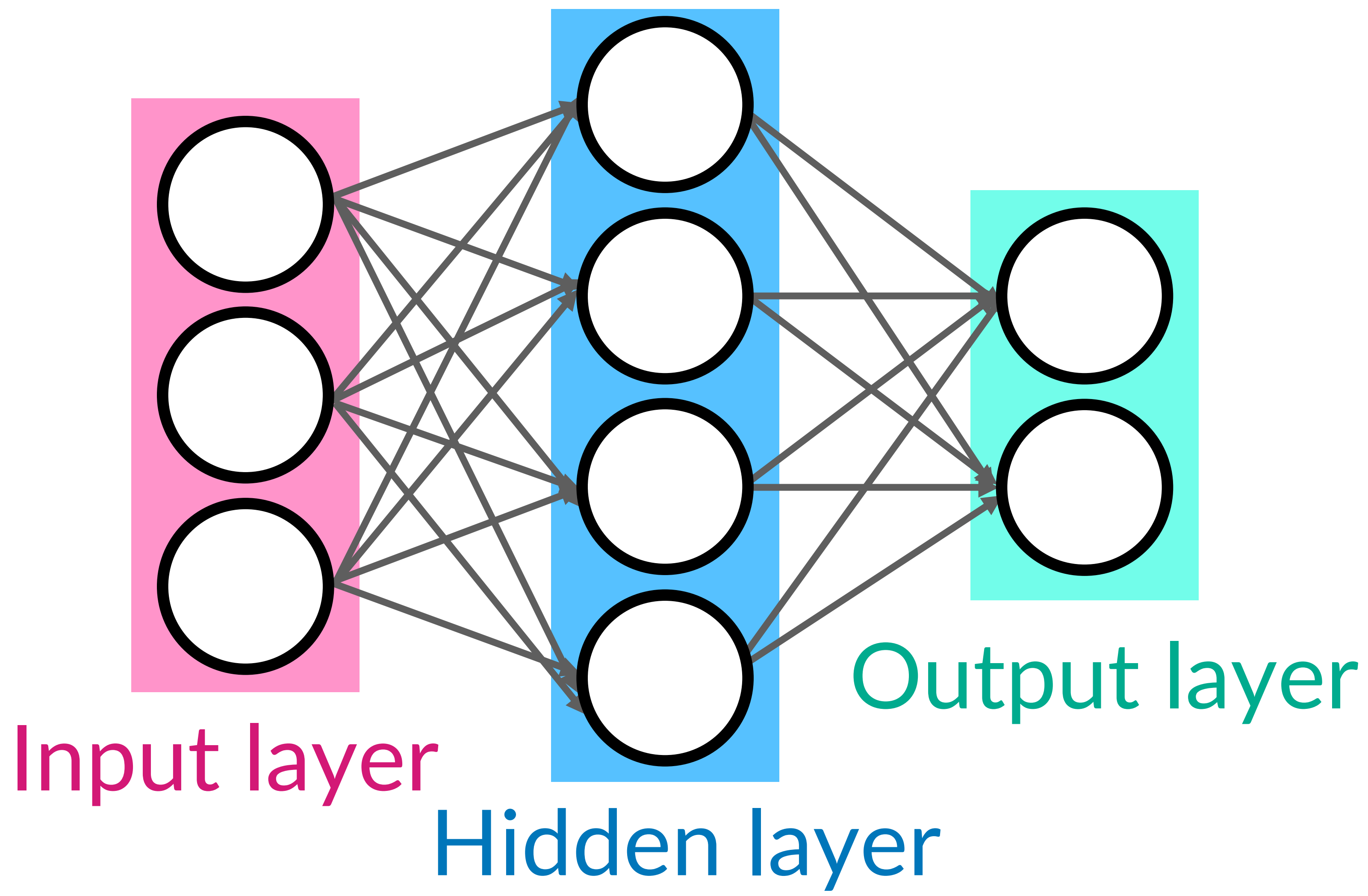
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Deep Neural Networks



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

Neural Net in <20 lines!



Initialize weights and data

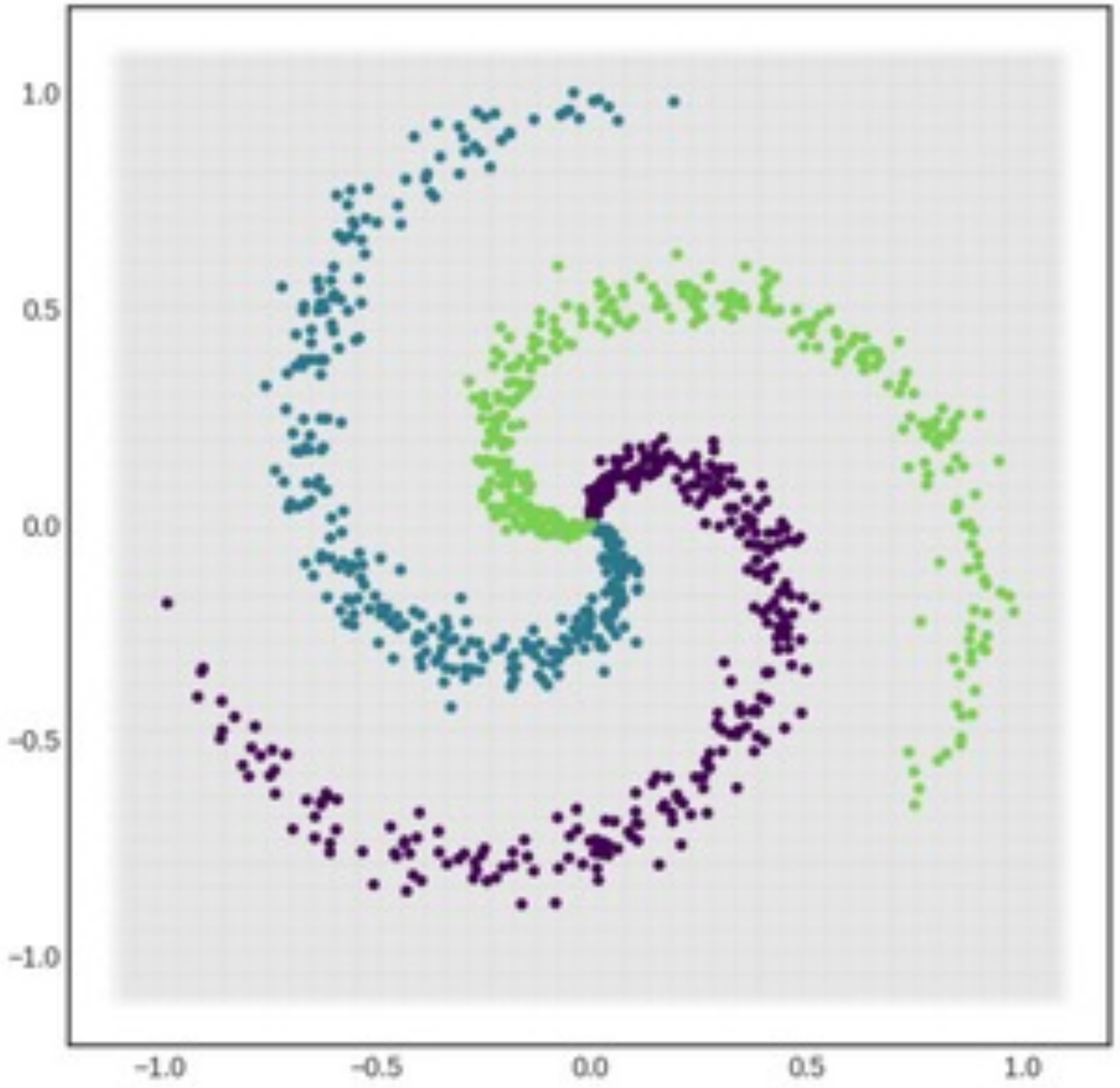
Compute loss (Sigmoid activation, L2 loss)

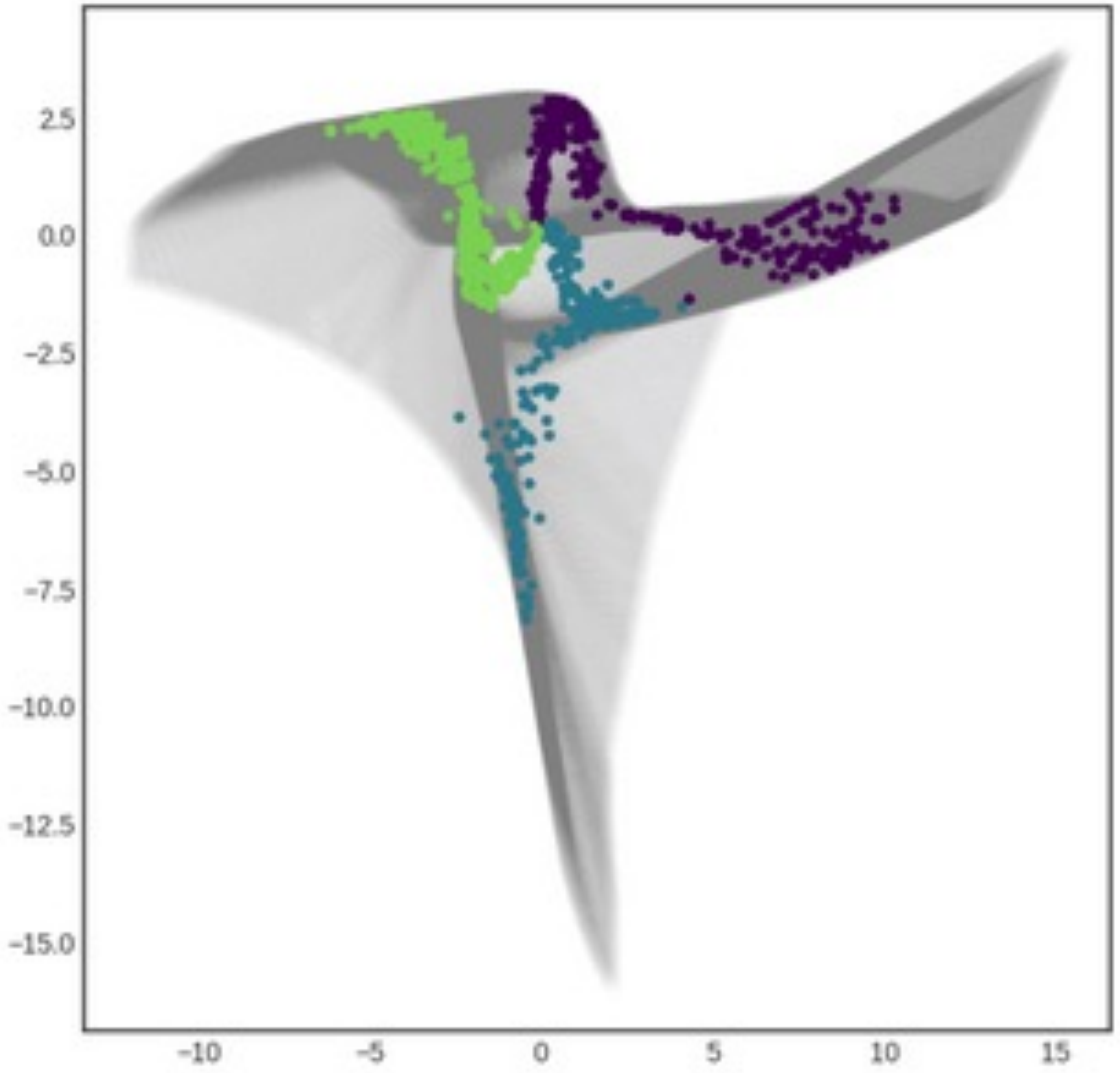
Compute gradients

SGD step

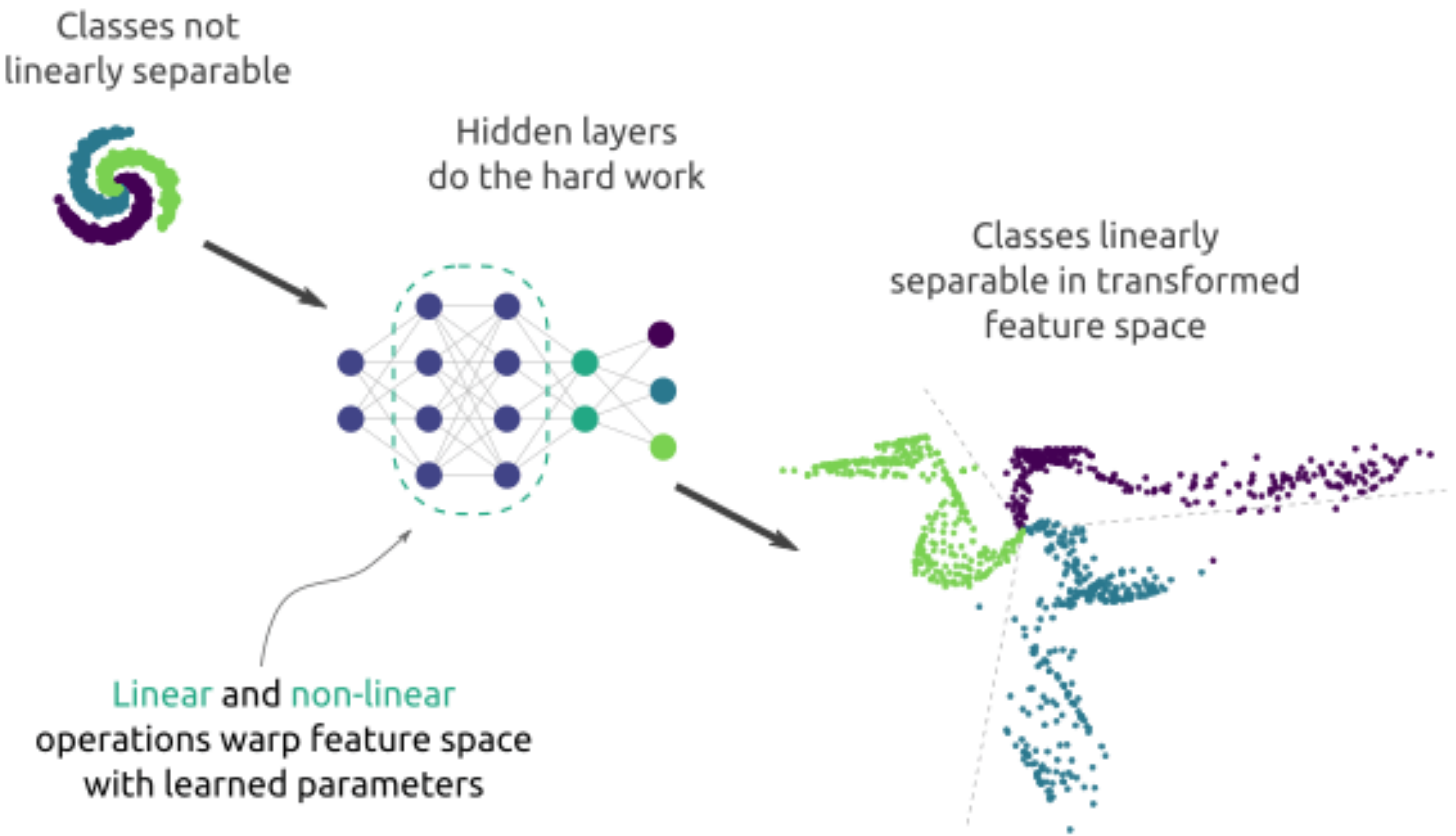
```python
import numpy as np
from numpy.random import randn

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
```

Feature Space Warping



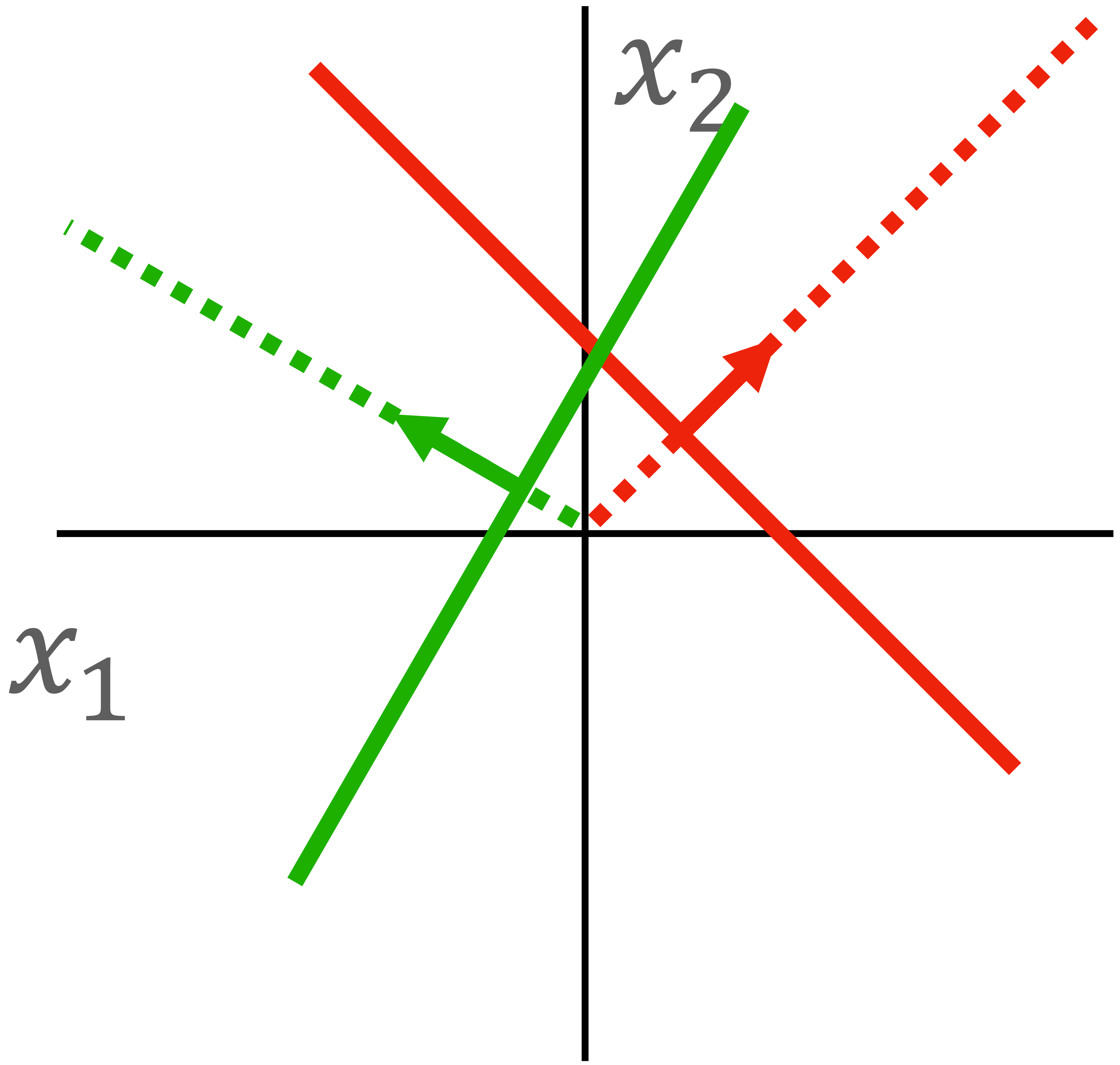


Feature Space Warping



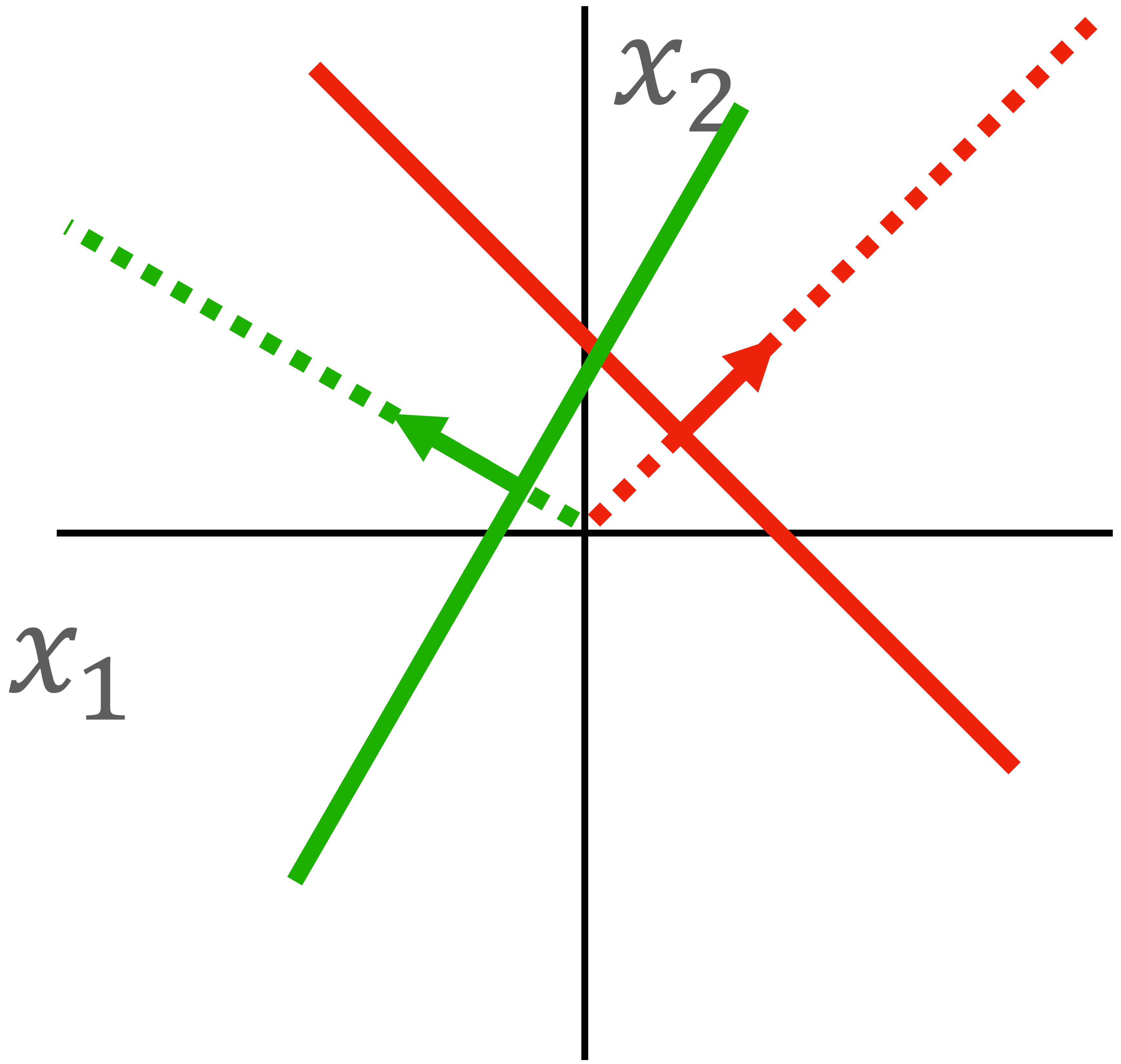
Feature Space Warping

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

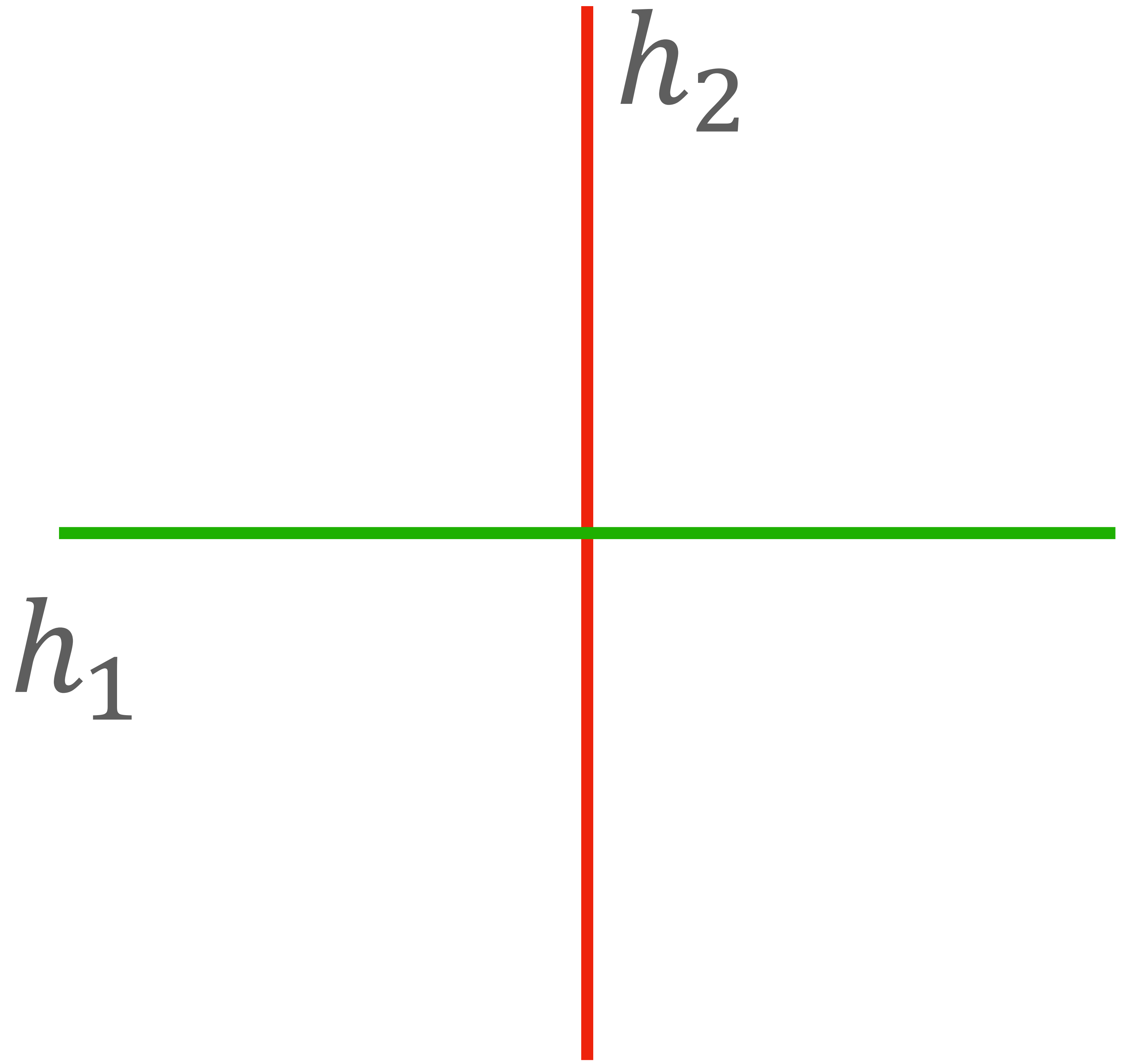


Feature Space Warping

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

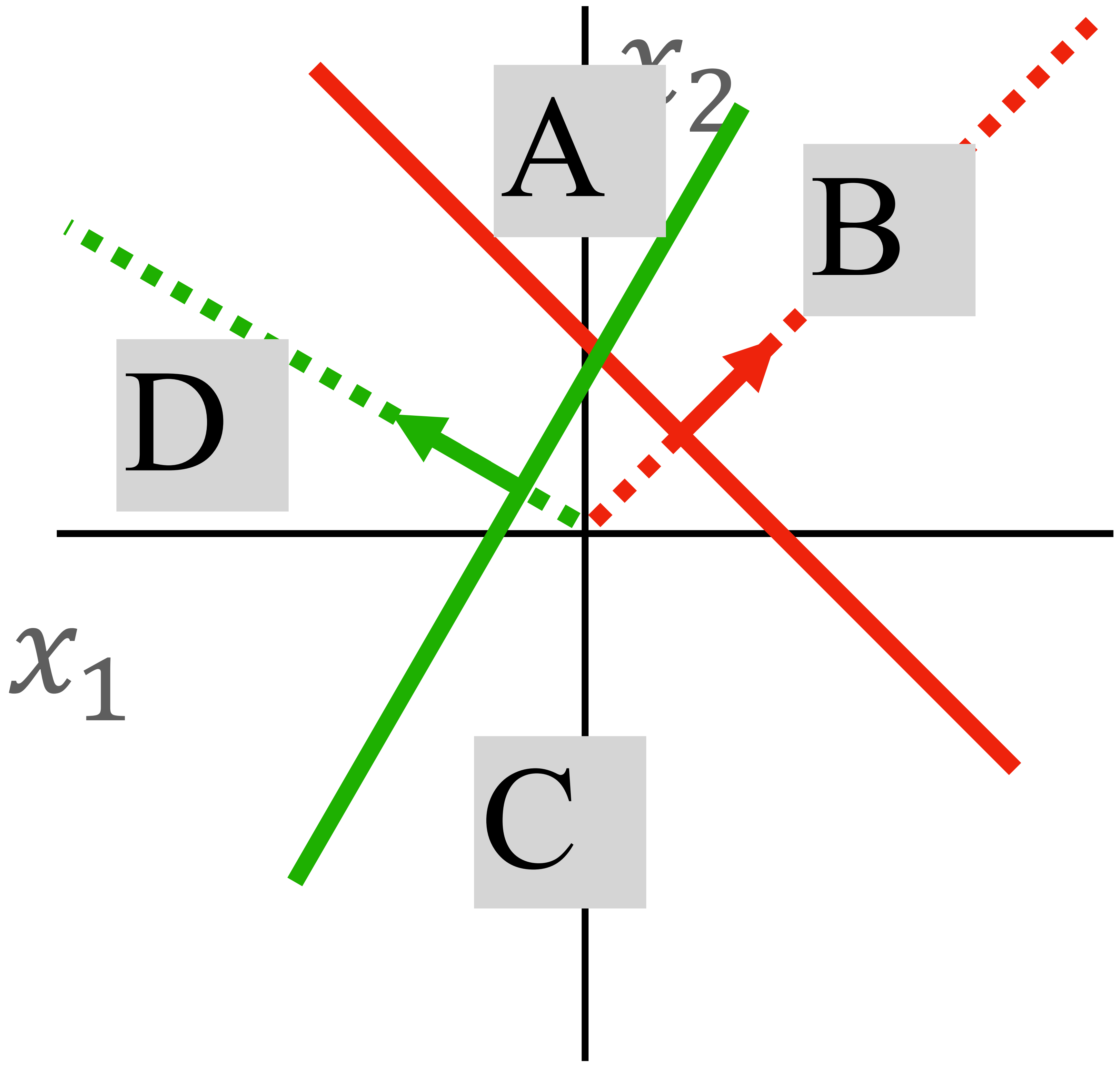


Feature transform:
$h = Wx + b$

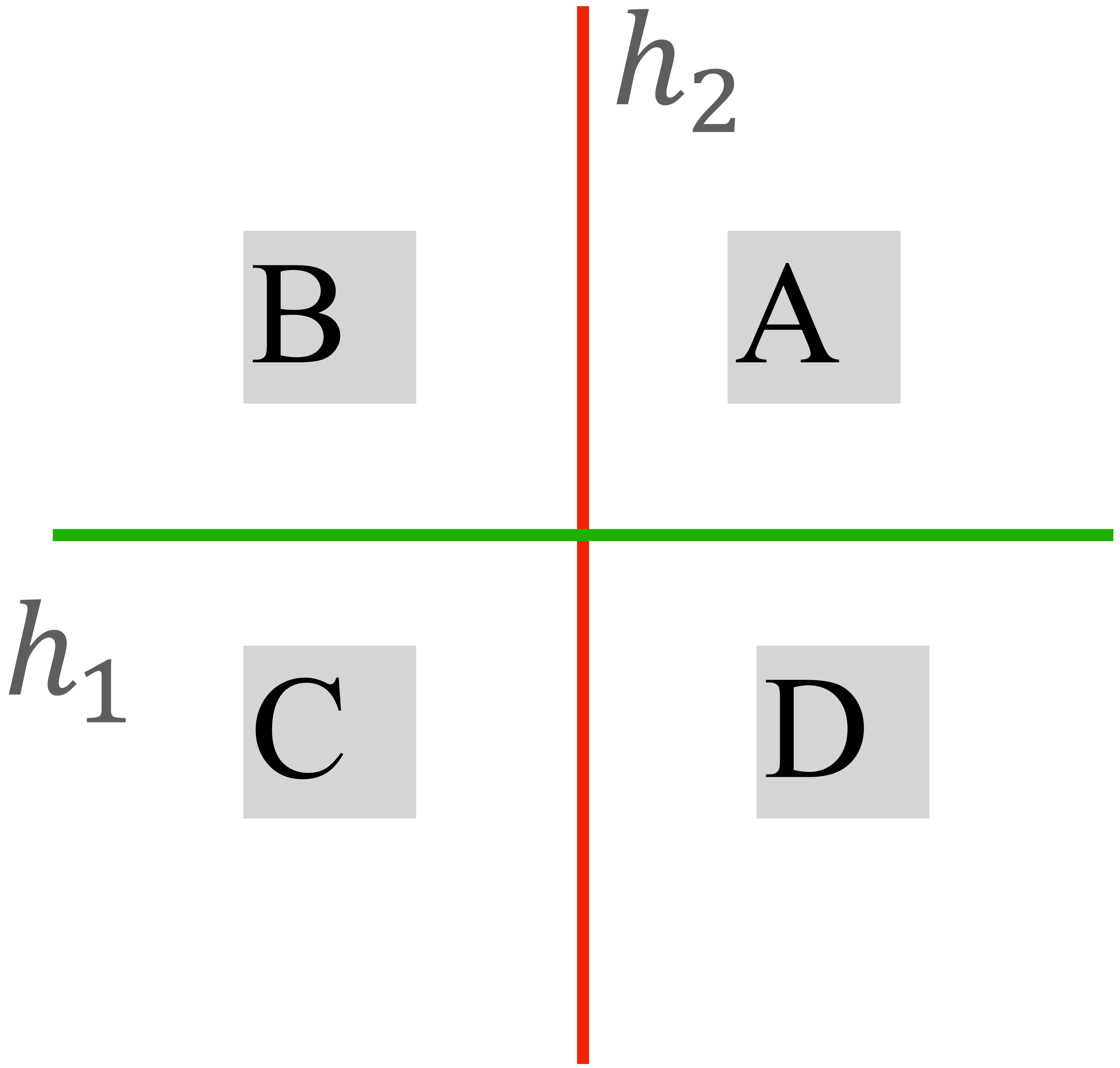


Feature Space Warping

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

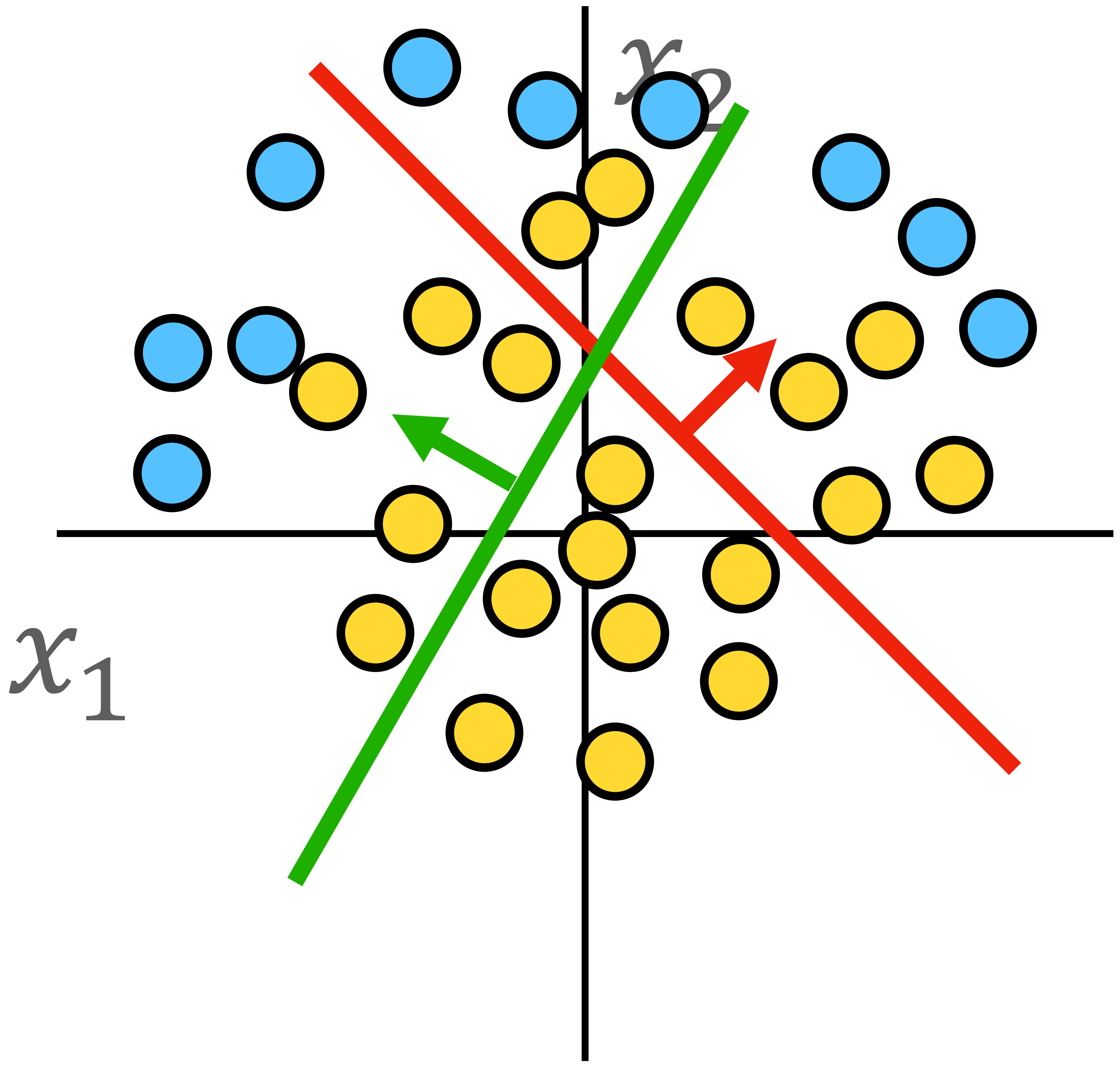


Feature transform:
$h = Wx + b$



Feature Space Warping

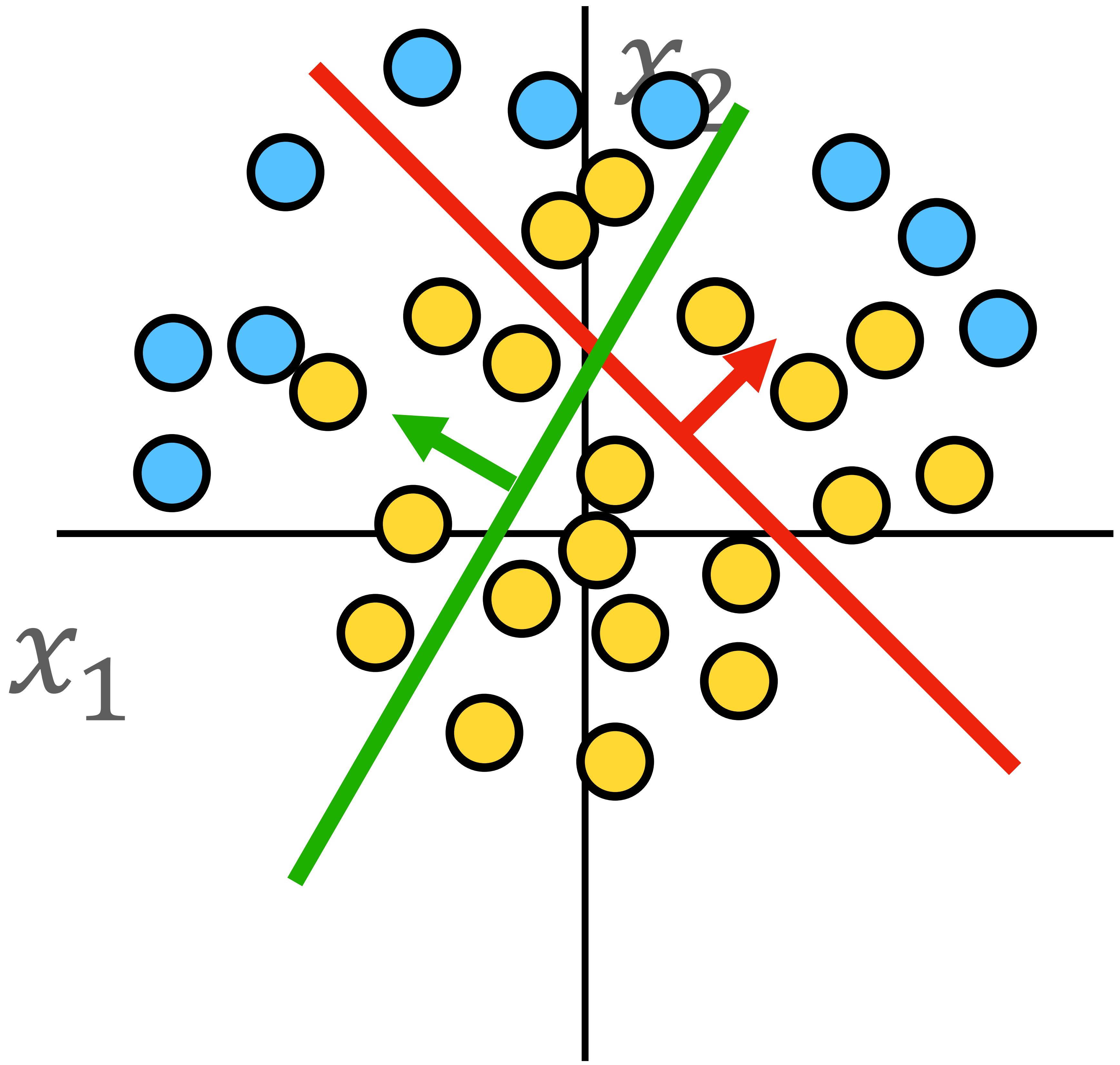
Points not linearly separable in original space



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

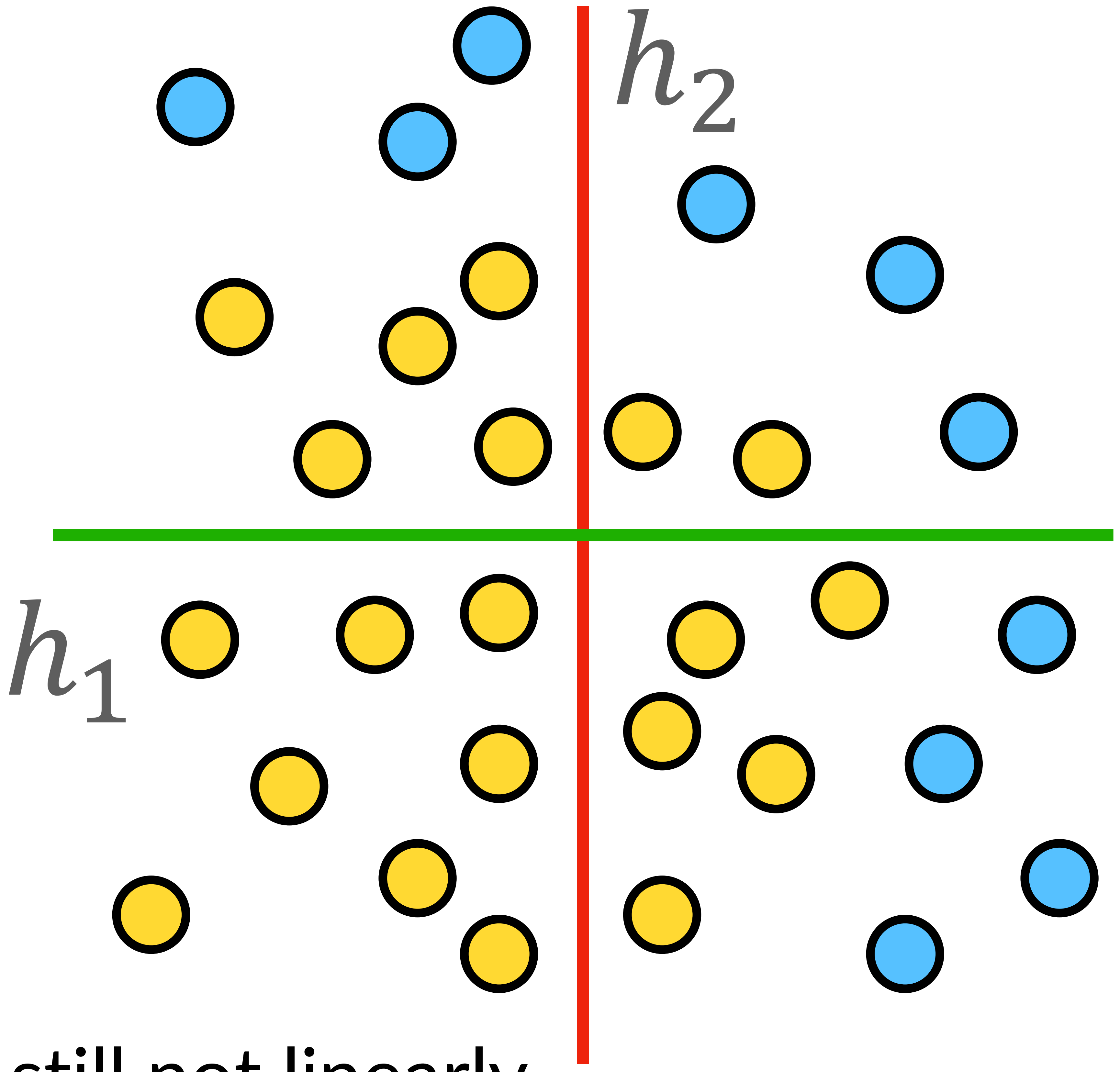
Feature Space Warping

Points not linearly separable in original space



Feature transform:
$h = Wx + b$

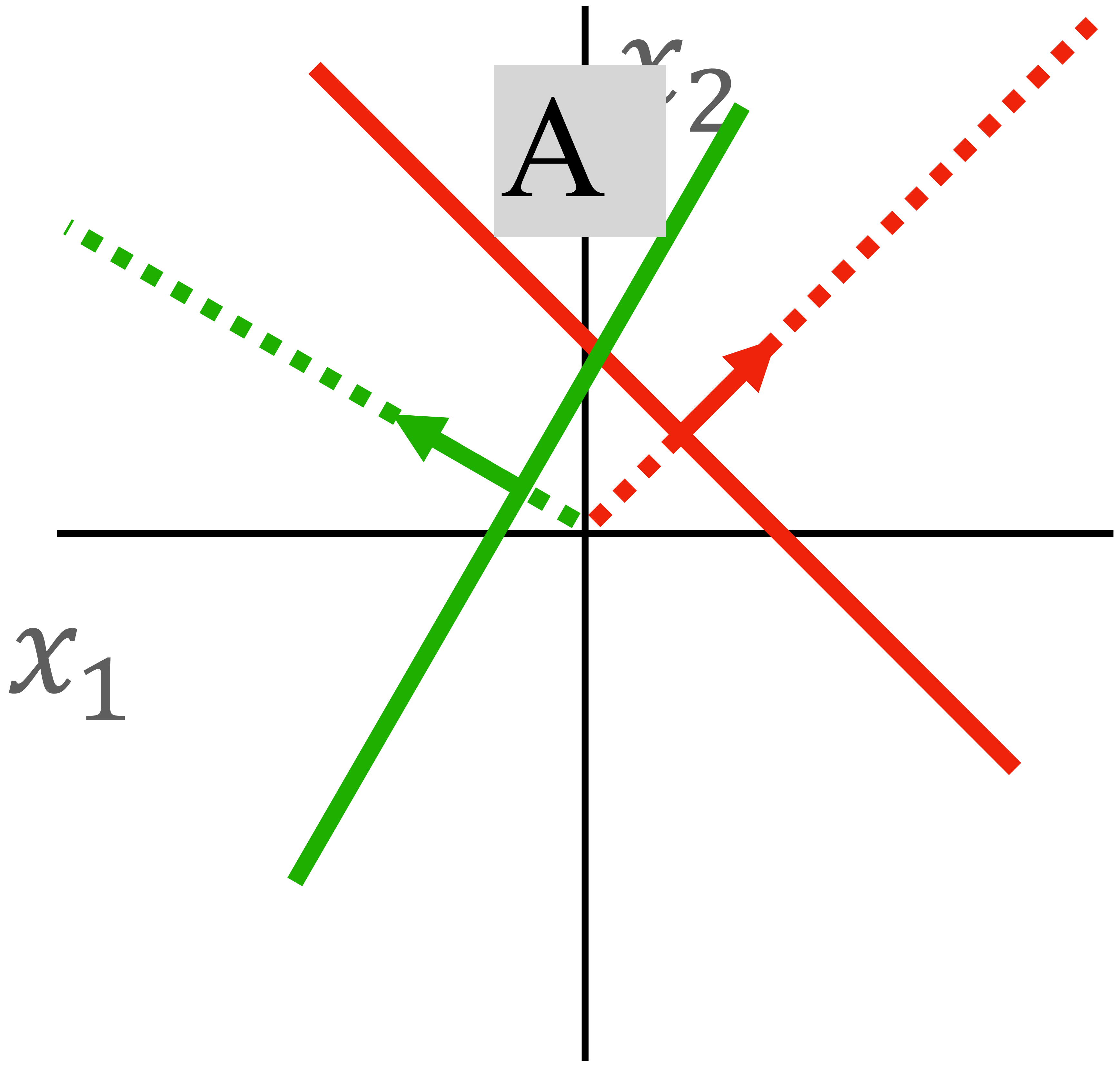
Consider a linear transform: $h = Wx + b$ where x, b, h are each 2-dimensional



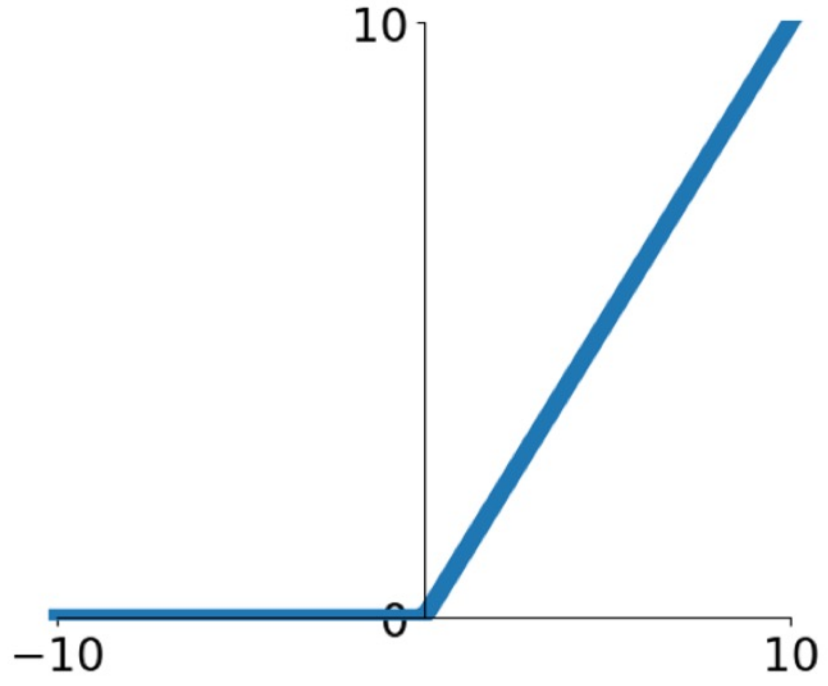
Points still not linearly separable in feature space

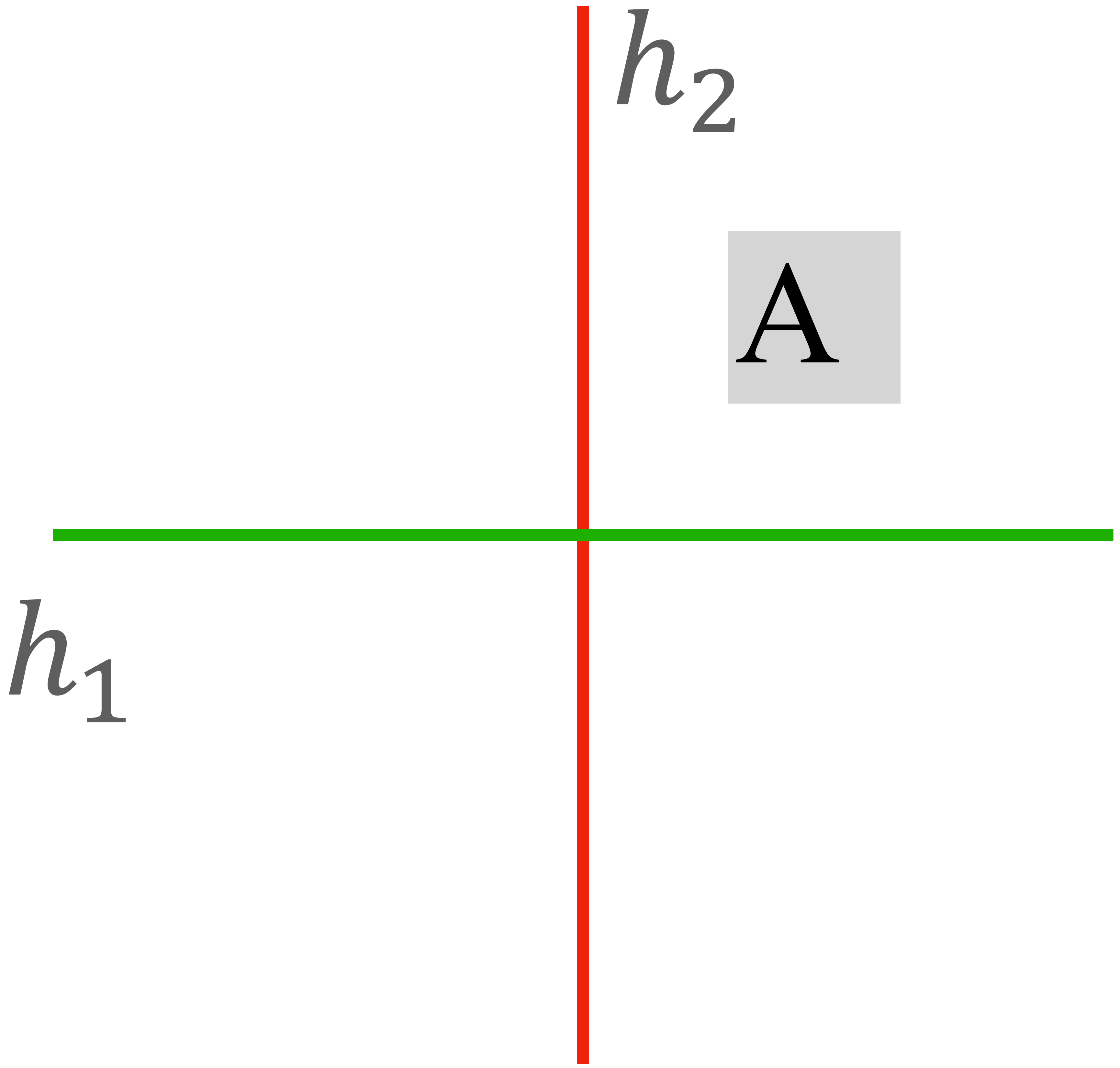
Feature Space Warping

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



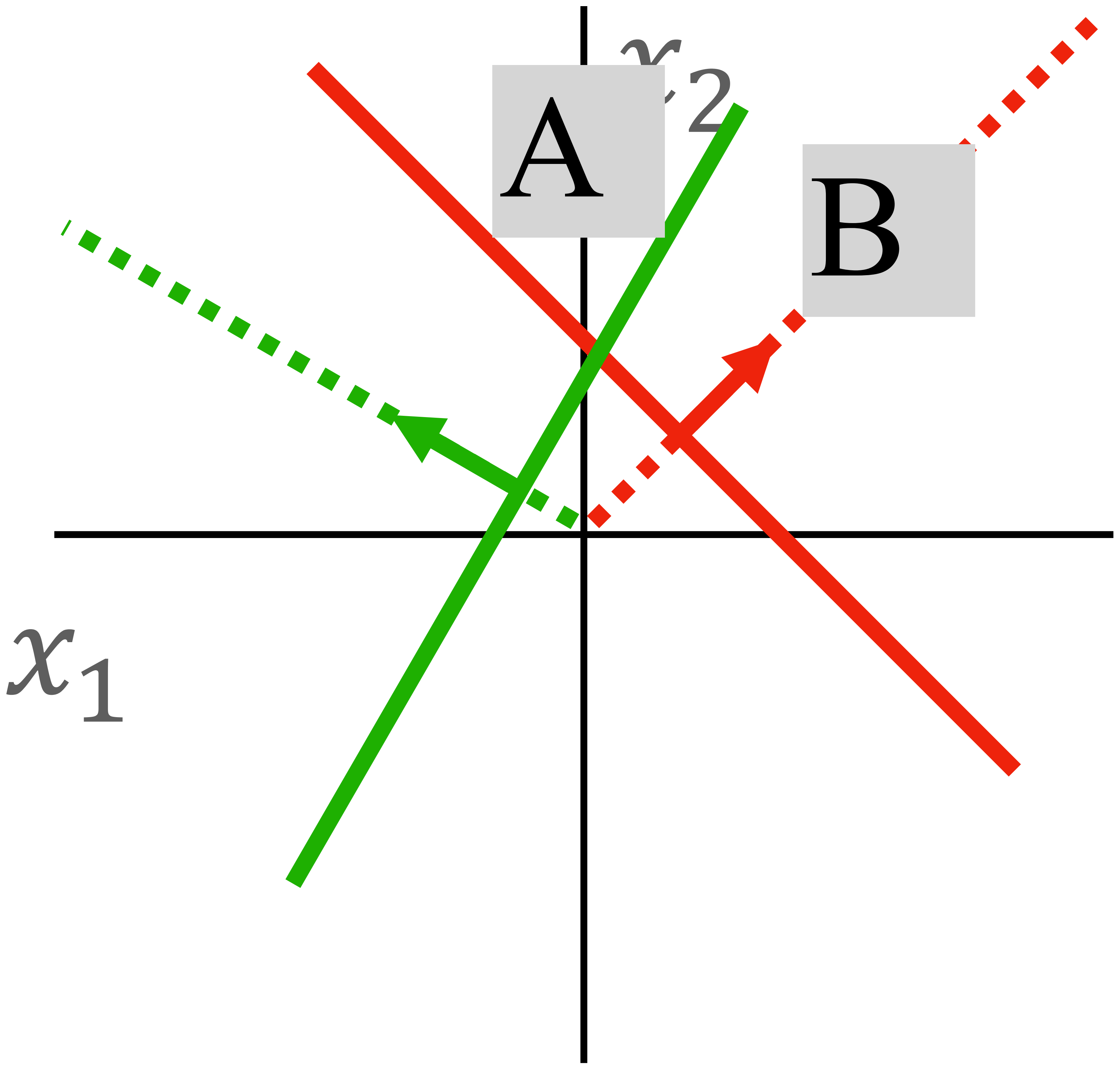
Feature transform:
$$h = ReLU(Wx + b)$$





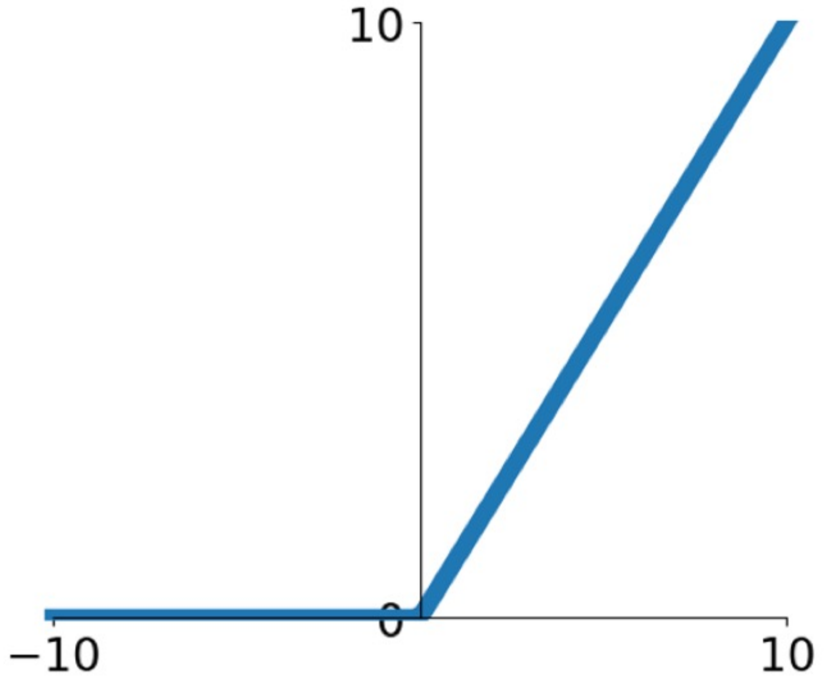
Feature Space Warping

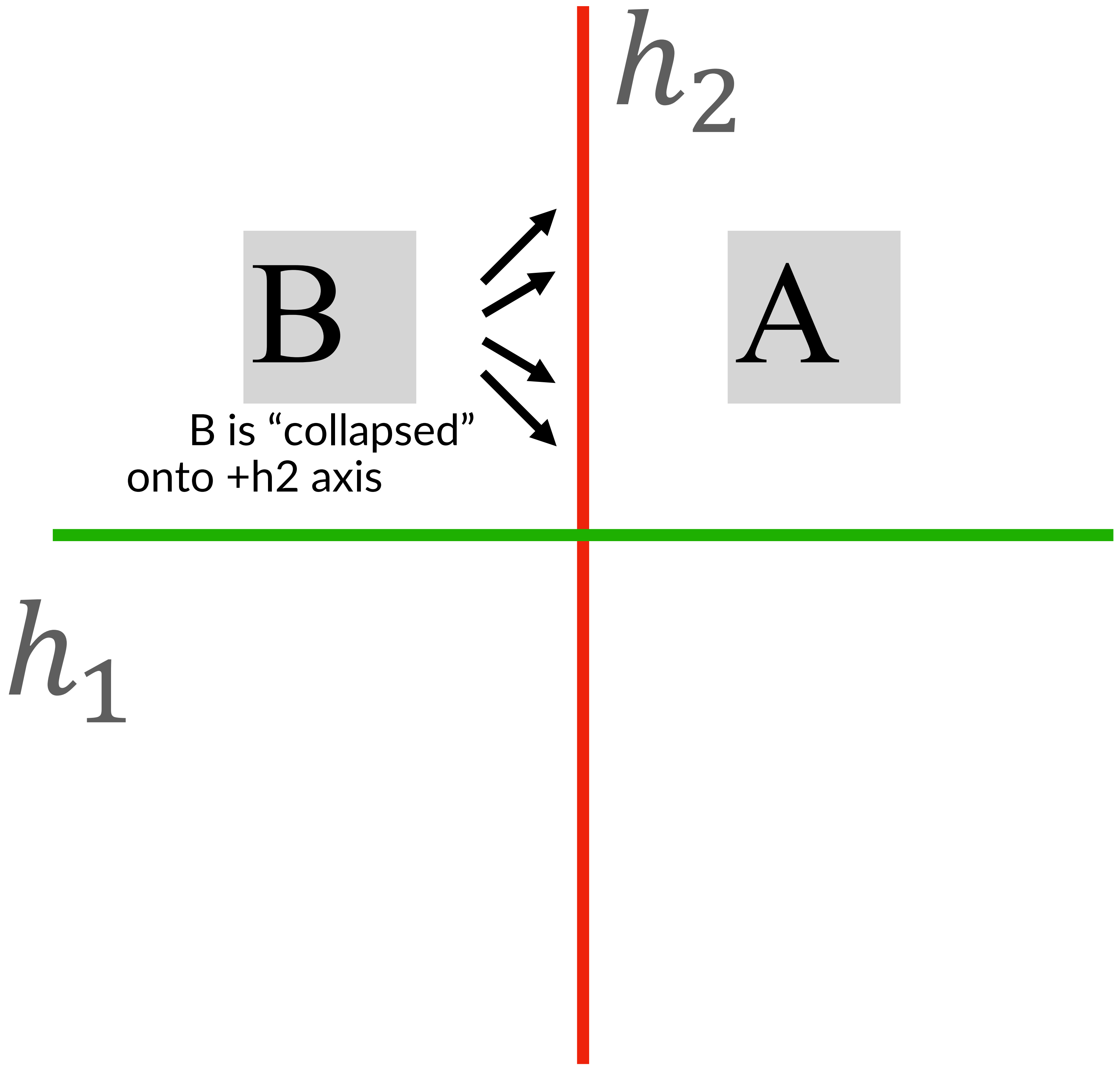
Consider a neural net hidden layer: $h = ReLU(Wx + b) = max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

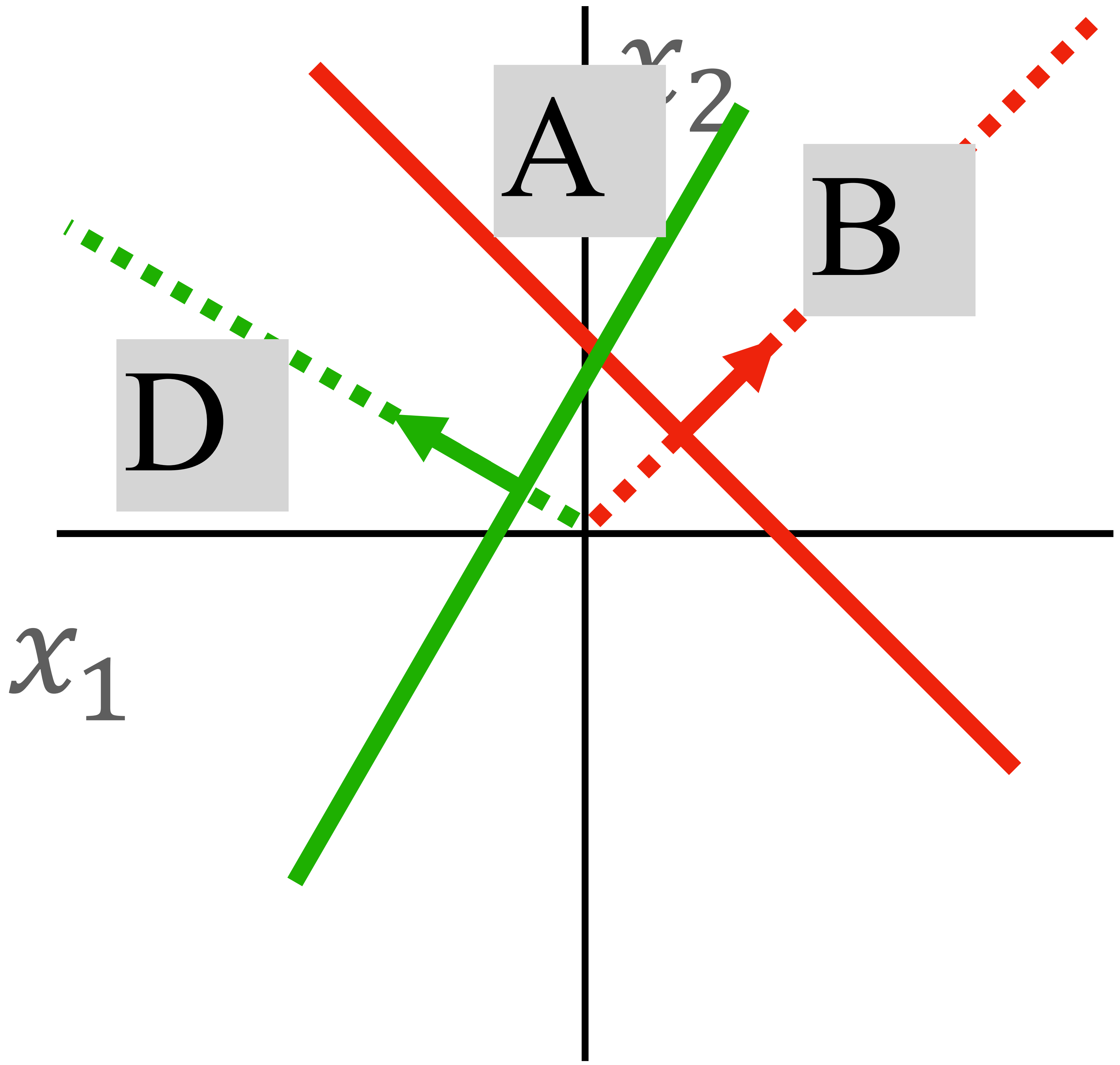
$$h = ReLU(Wx + b)$$



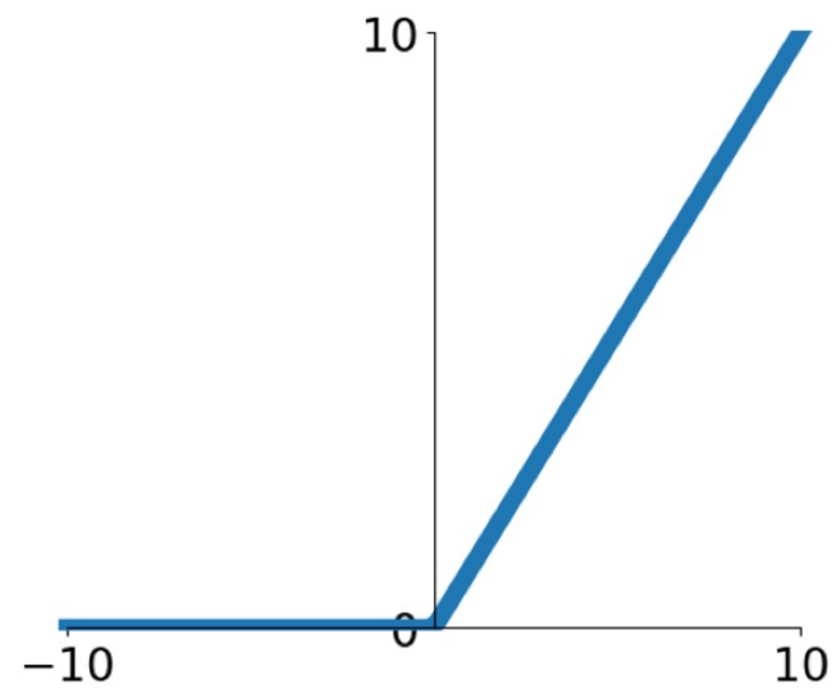


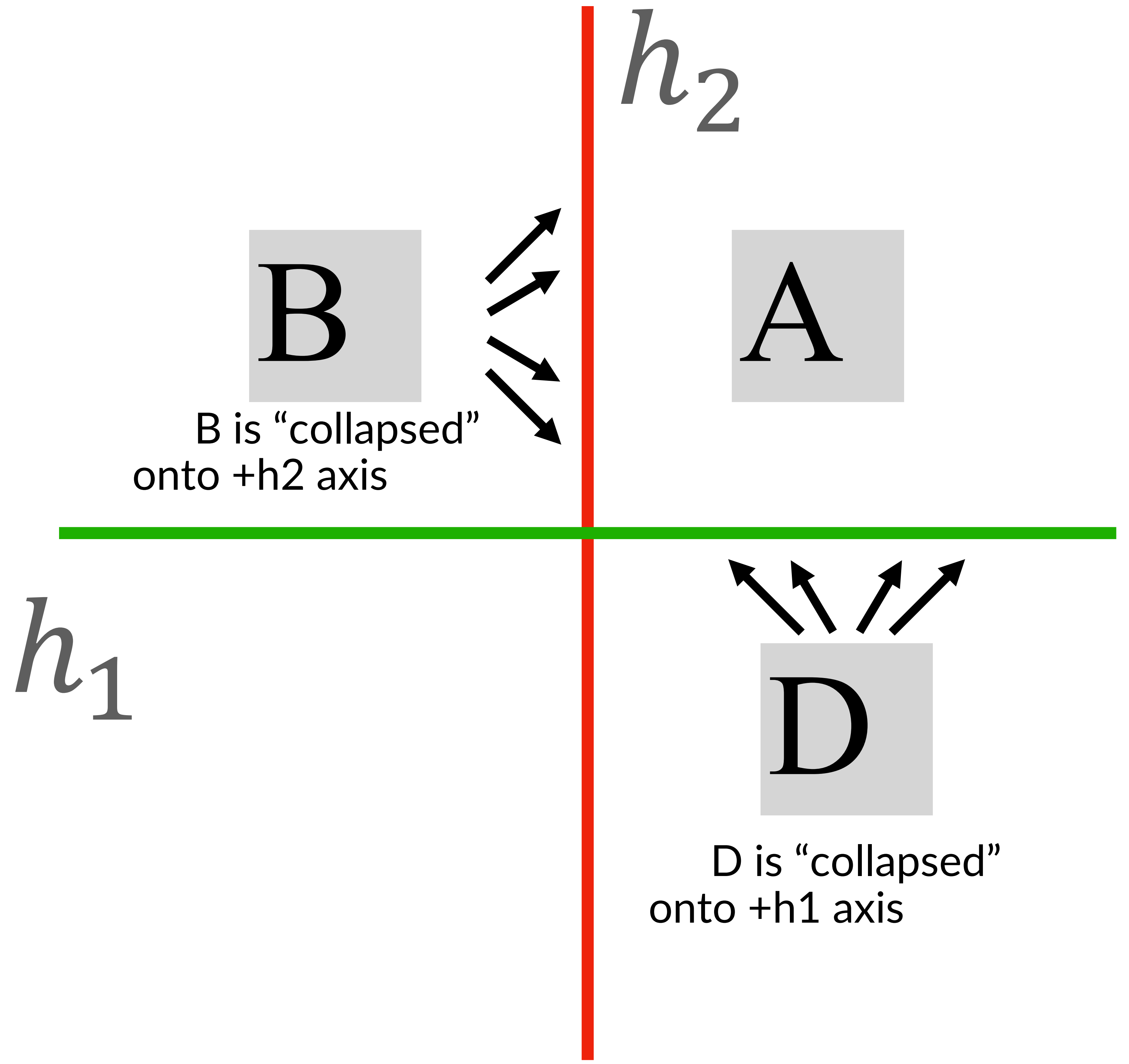
Feature Space Warping

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



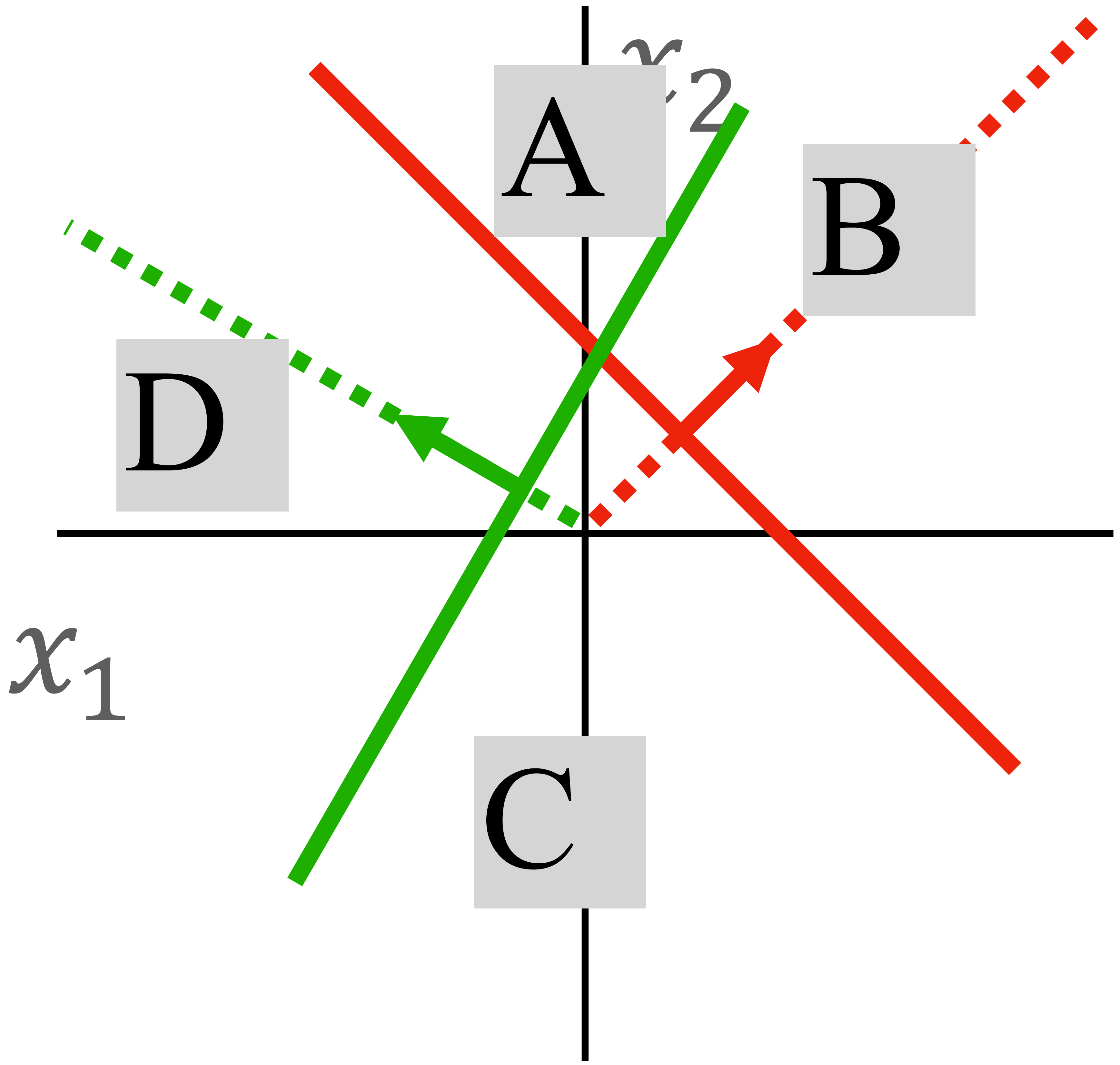
Feature transform:
$h = ReLU(Wx + b)$



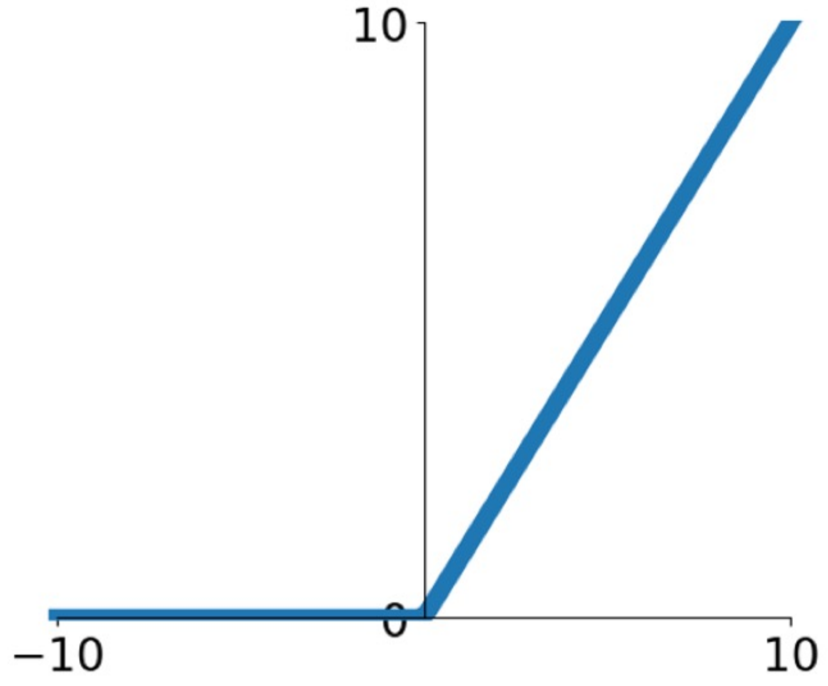


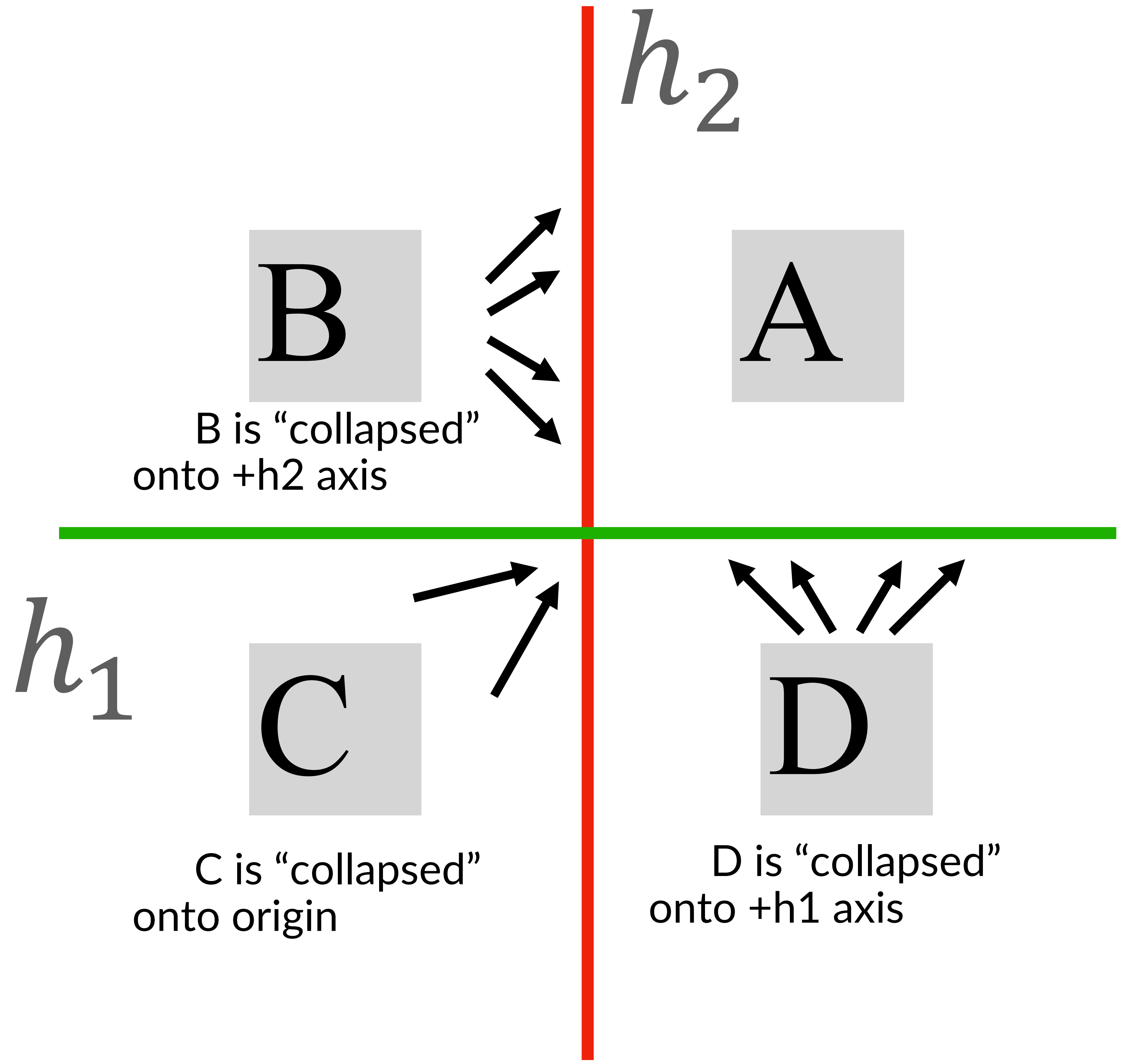
Feature Space Warping

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



Feature transform:
$h = ReLU(Wx + b)$

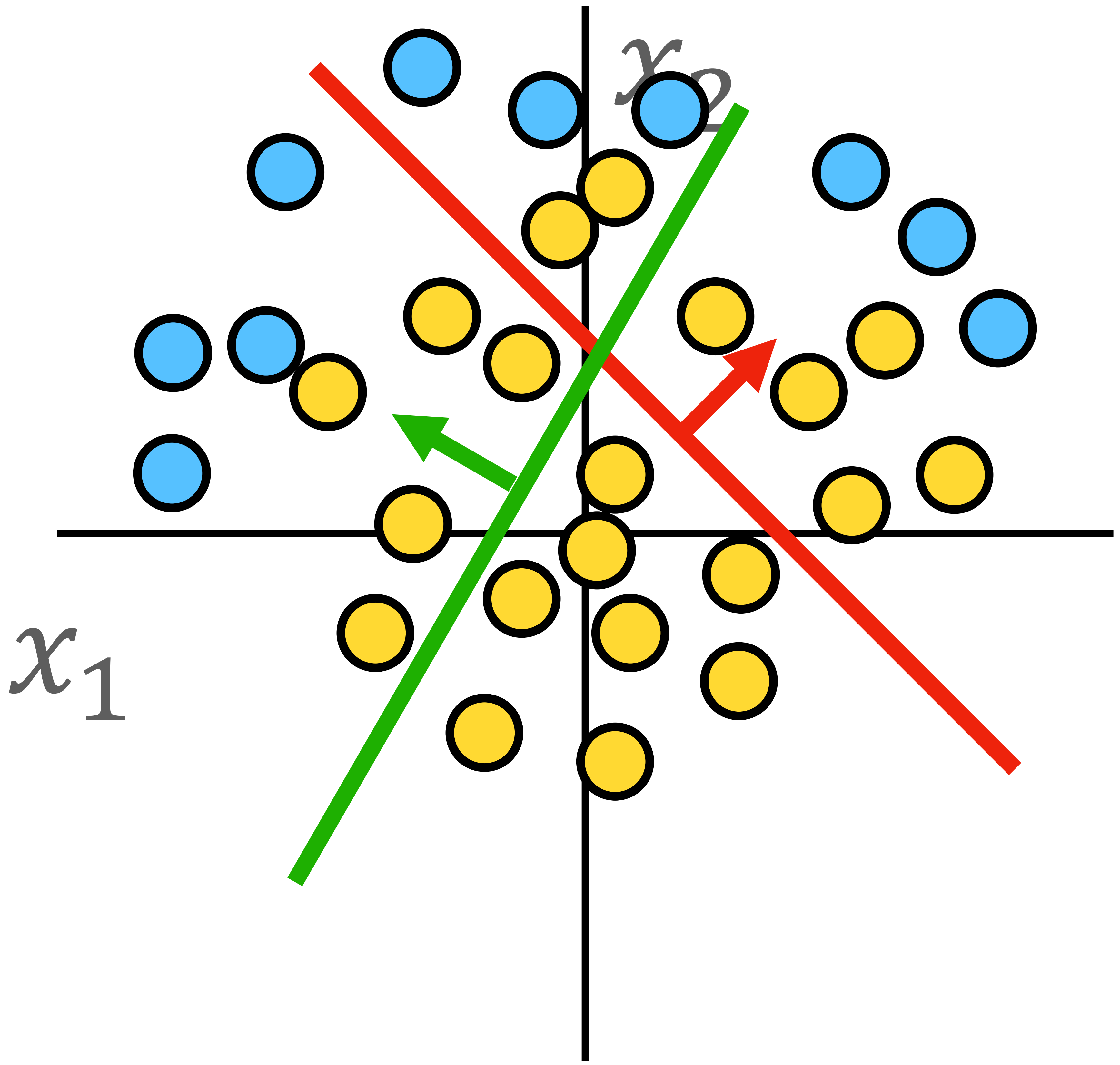




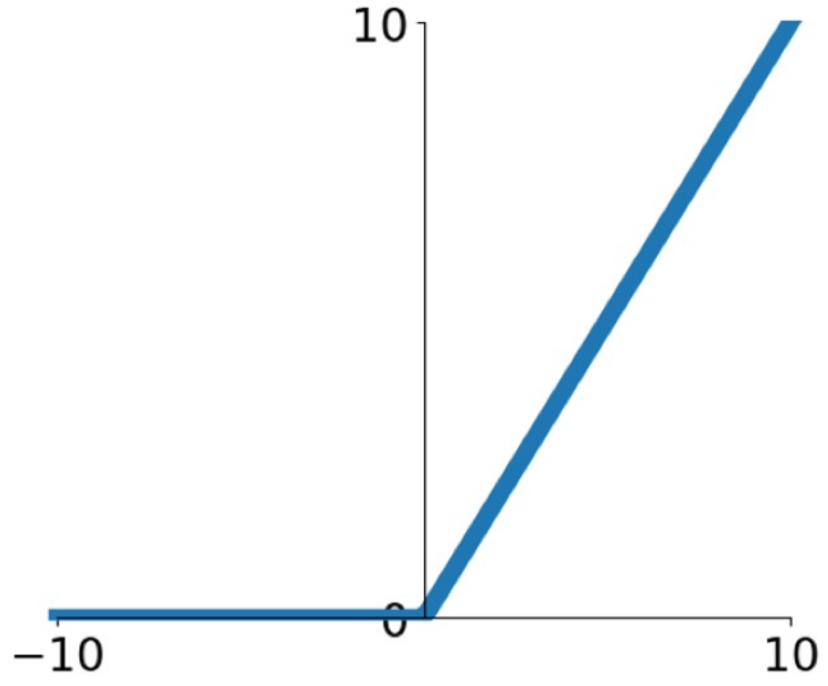
Feature Space Warping

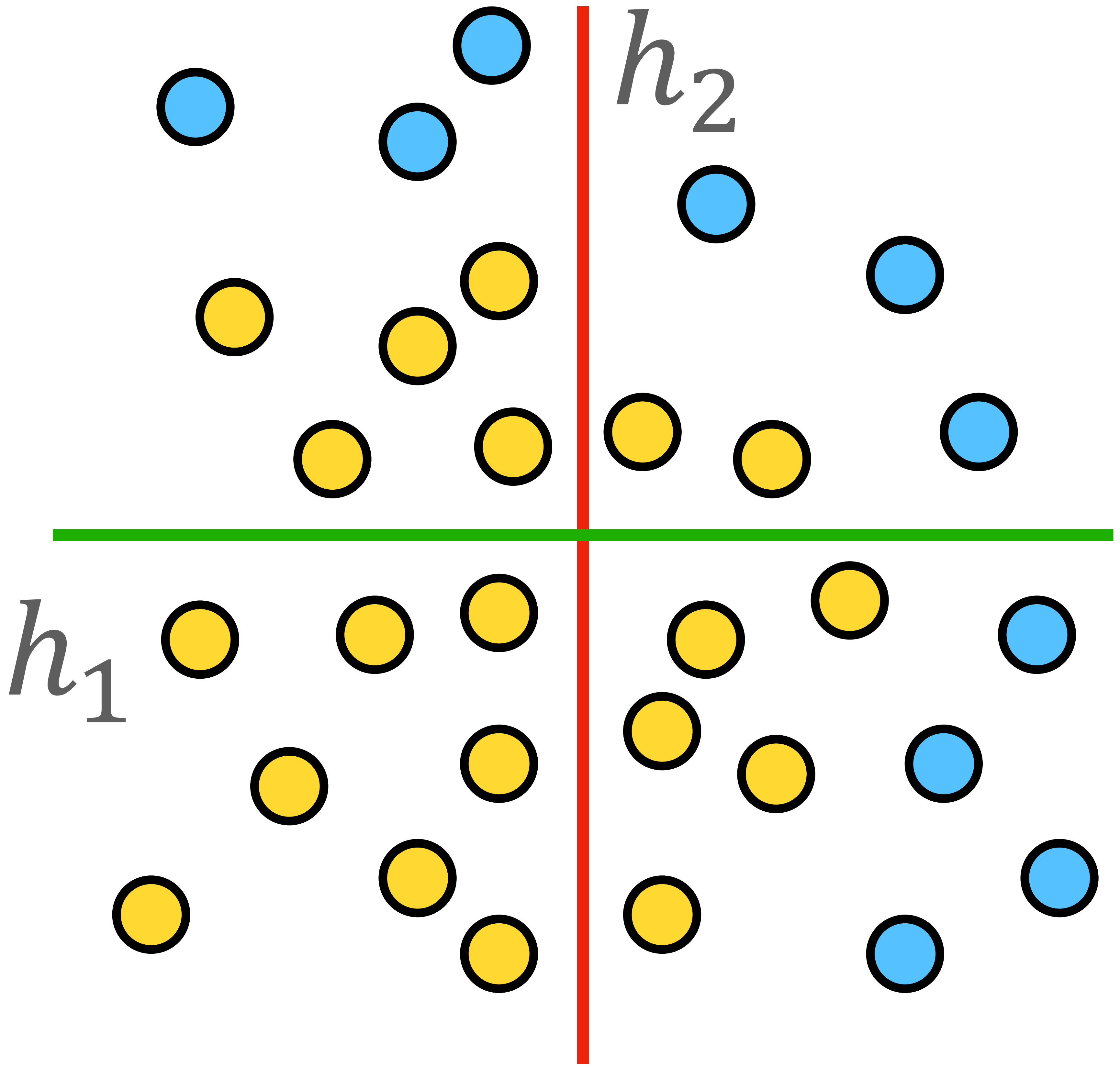
Points not linearly separable in original space

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



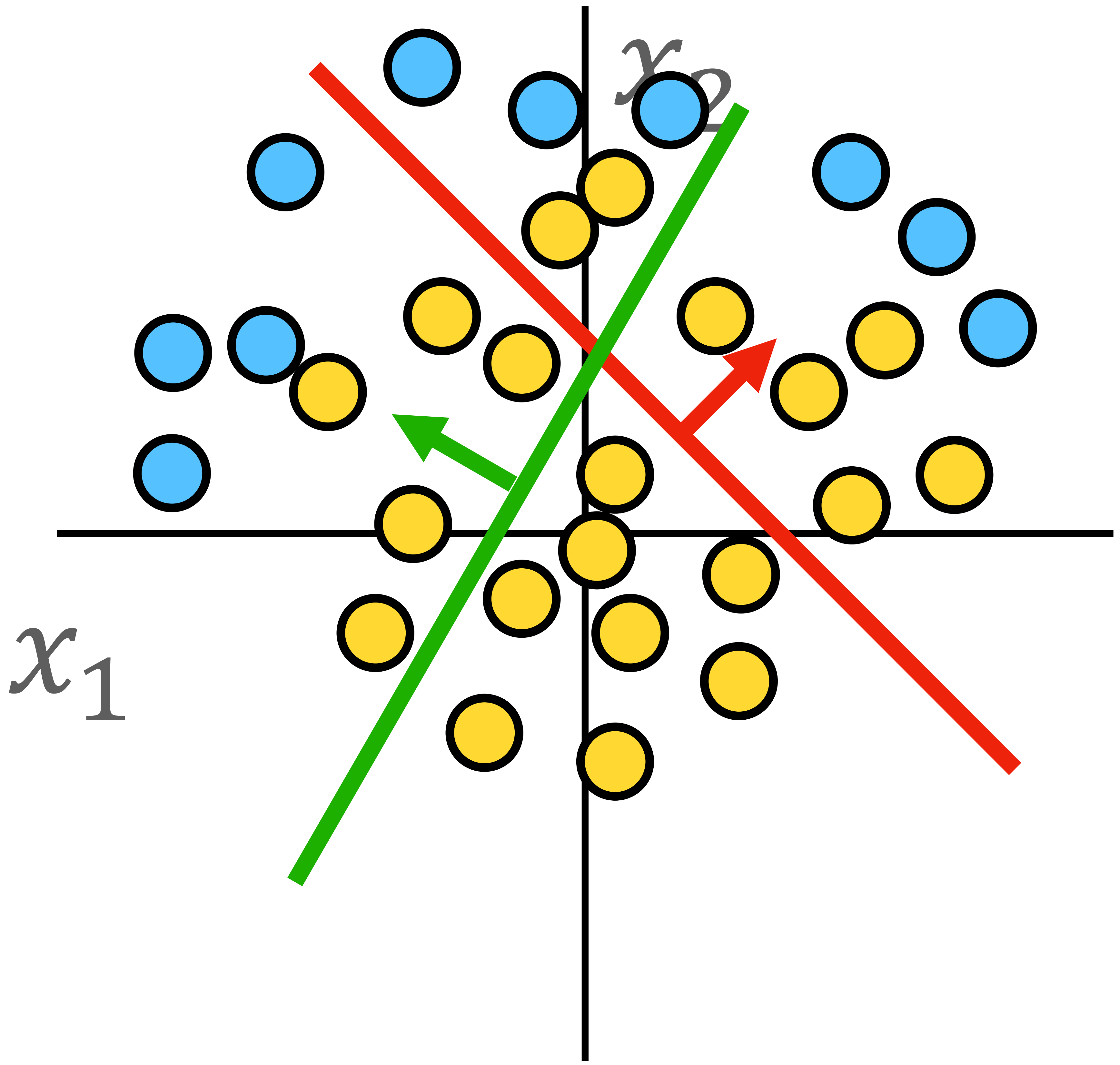
Feature transform: $h = ReLU(Wx + b)$





Feature Space Warping

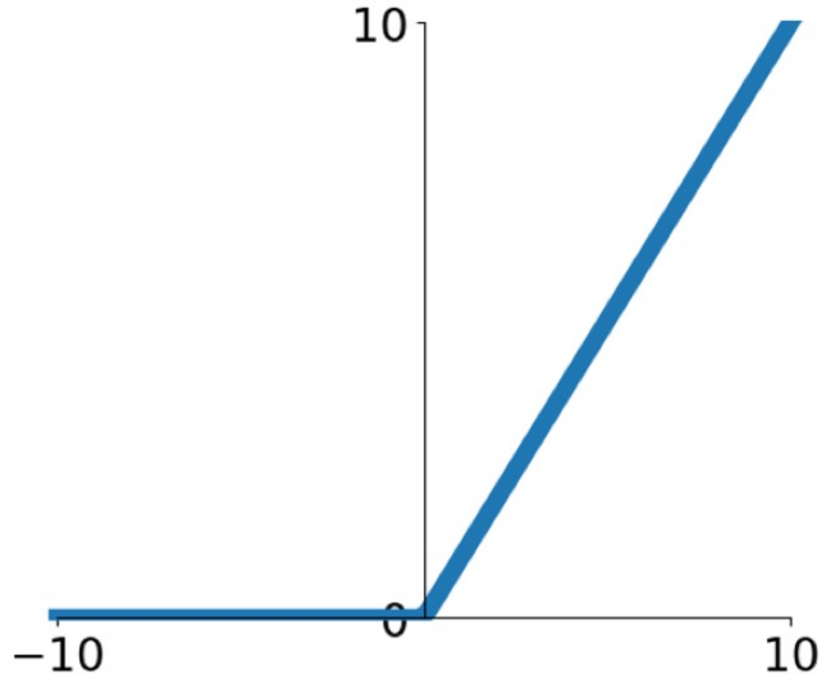
Points not linearly separable in original space

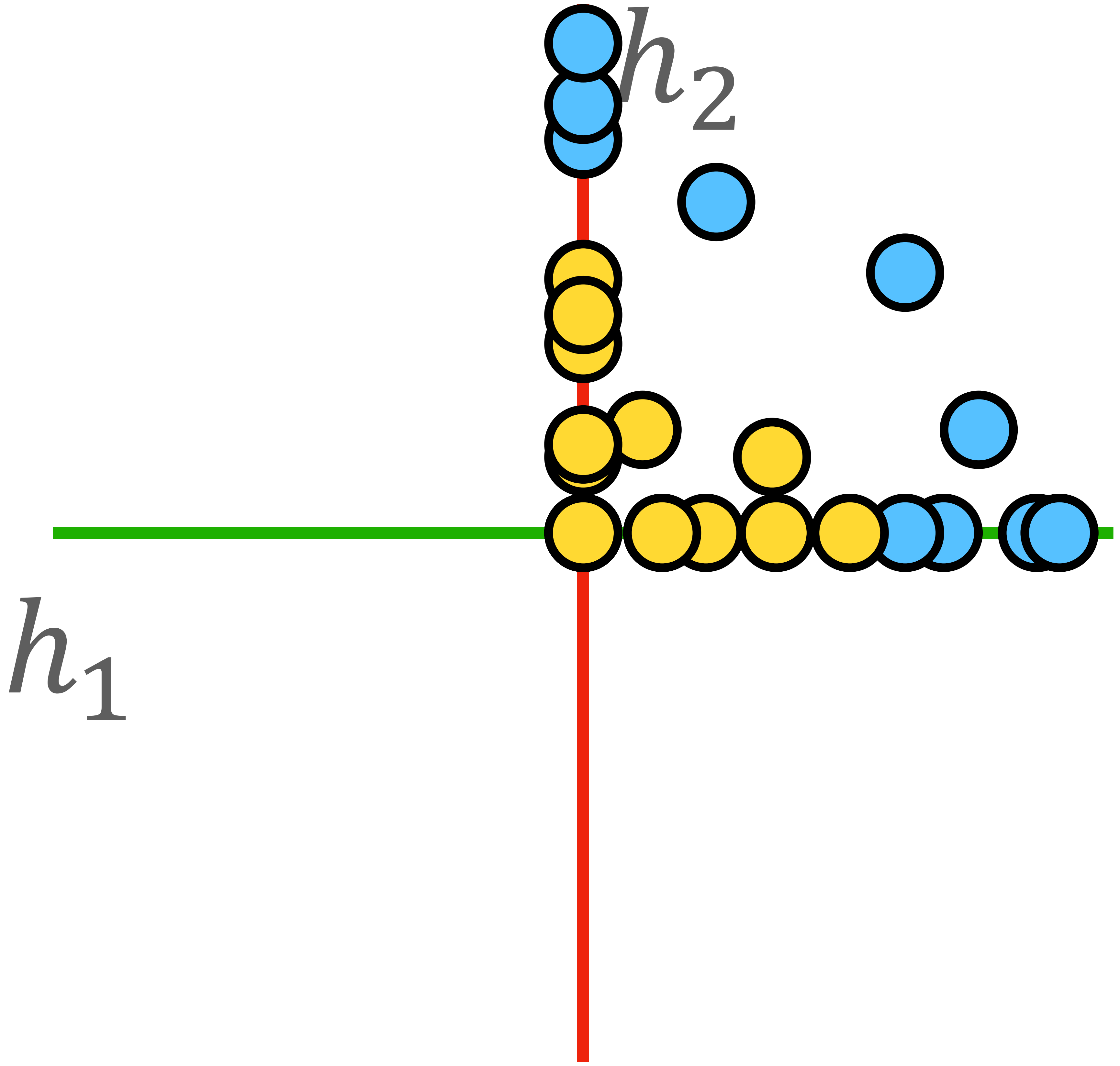


Consider a neural net hidden layer: $h = ReLU(Wx + b) = max(0, Wx + b)$ where x, b, h are each 2-dimensional

Feature transform:

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

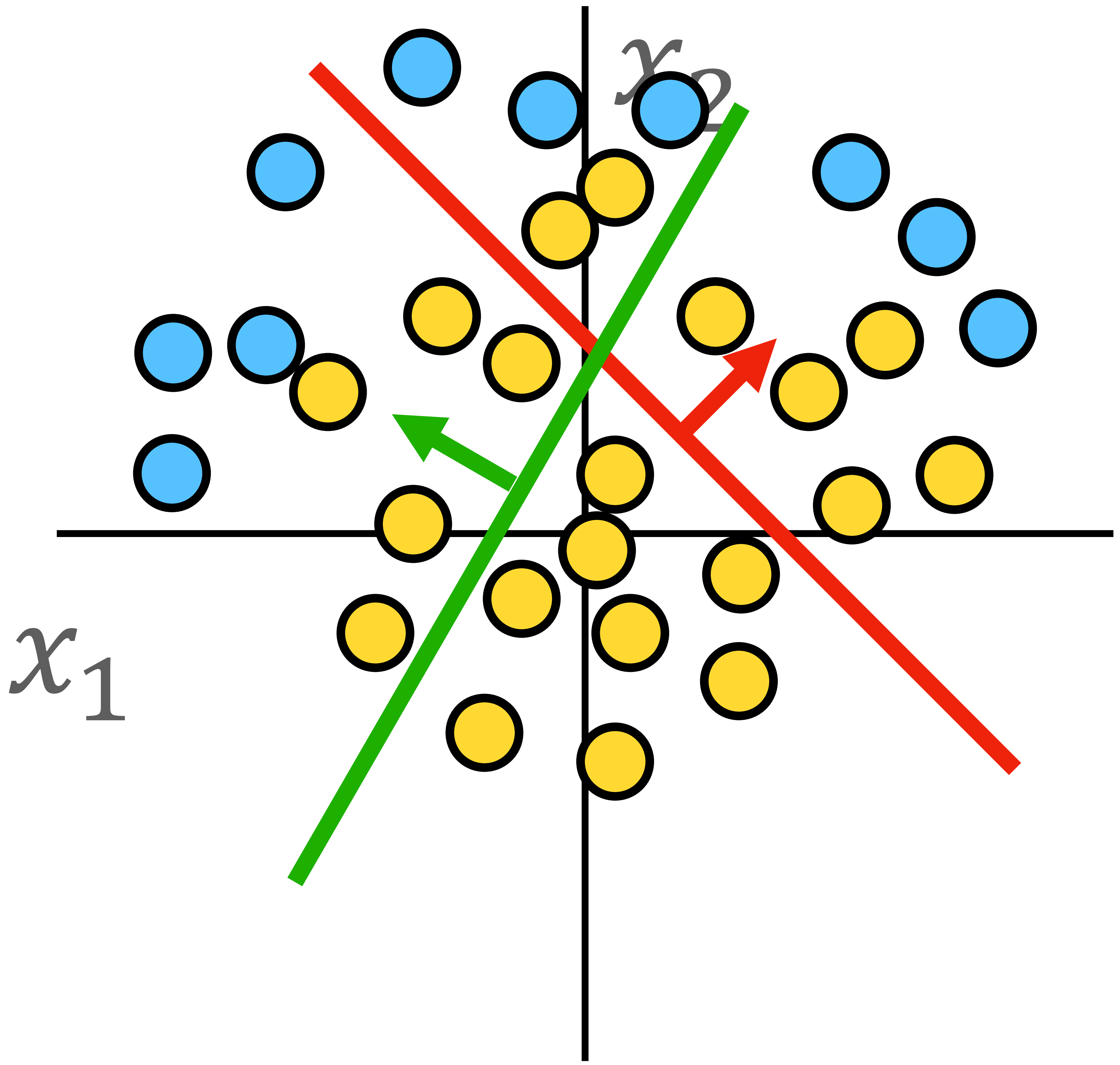




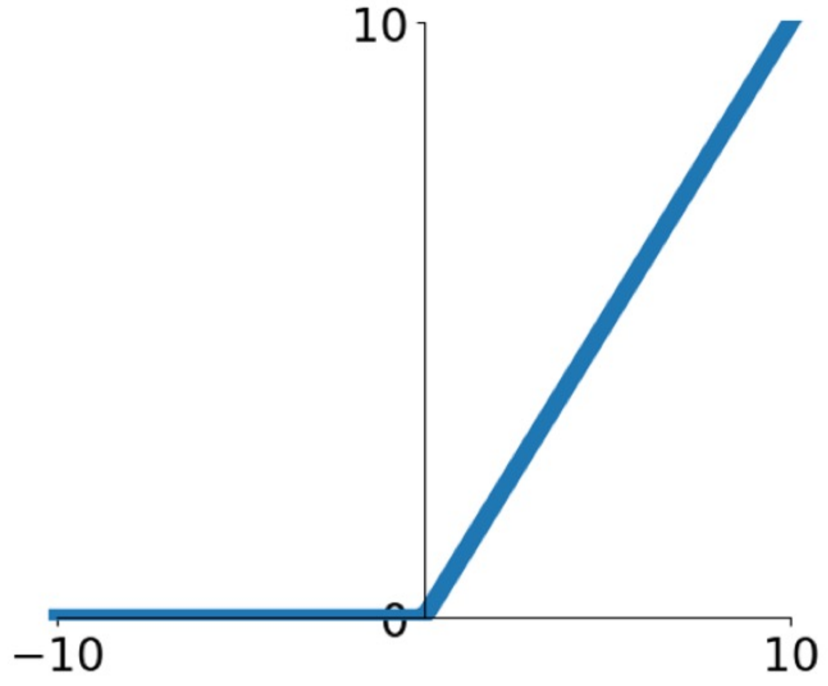
Feature Space Warping

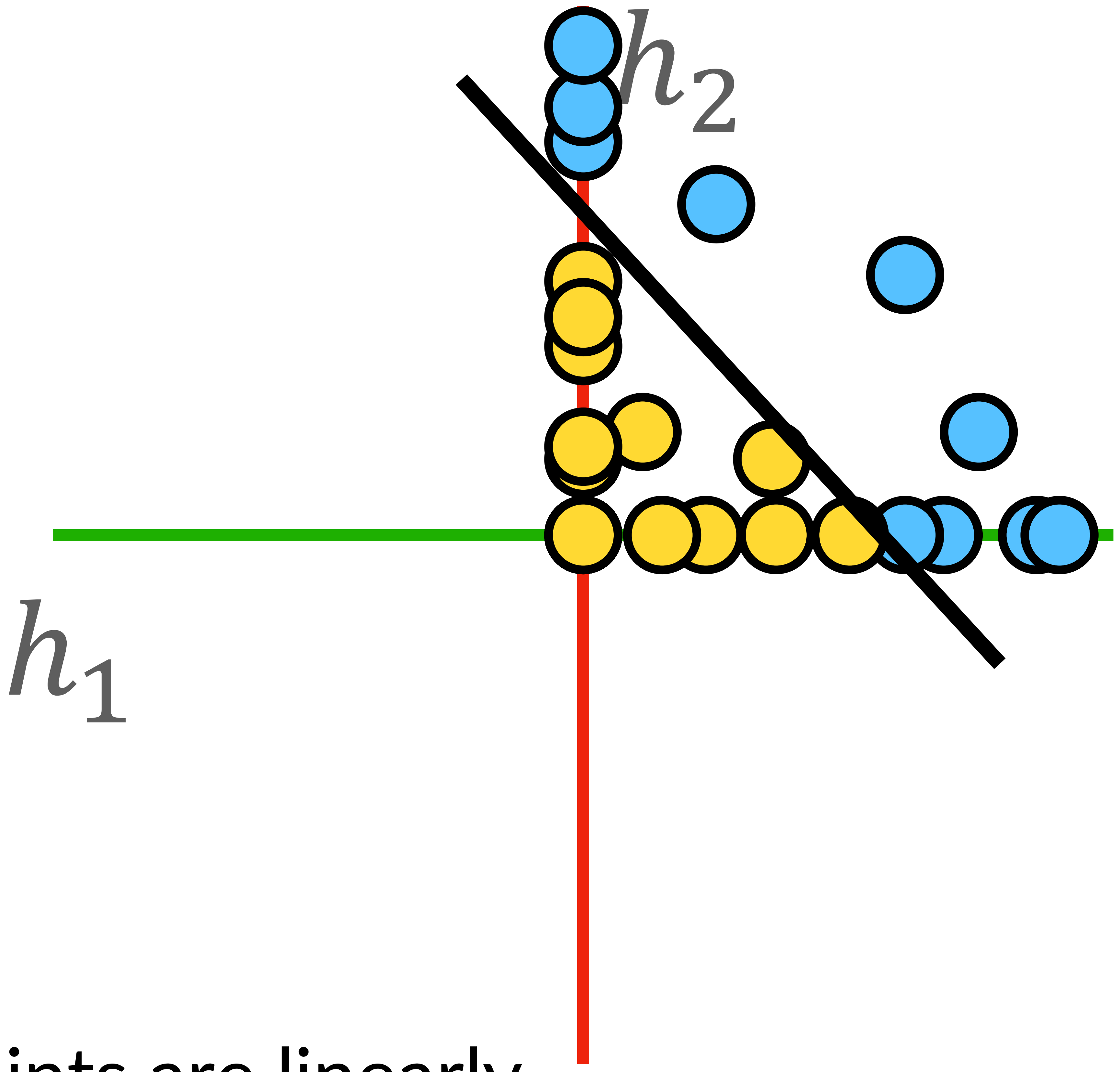
Points not linearly separable in original space

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



Feature transform: $h = ReLU(Wx + b)$



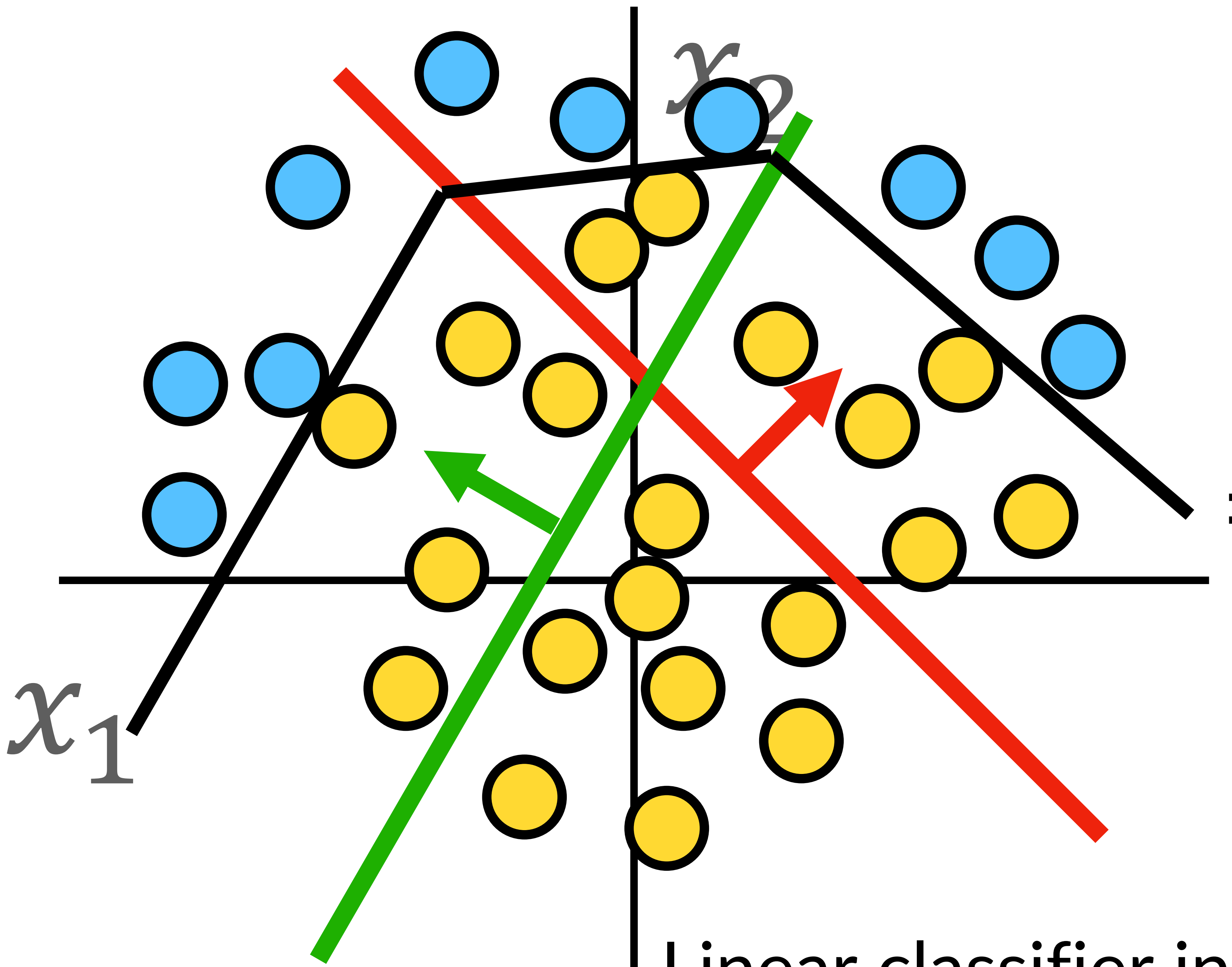


Points are linearly separable in feature space!

Feature Space Warping

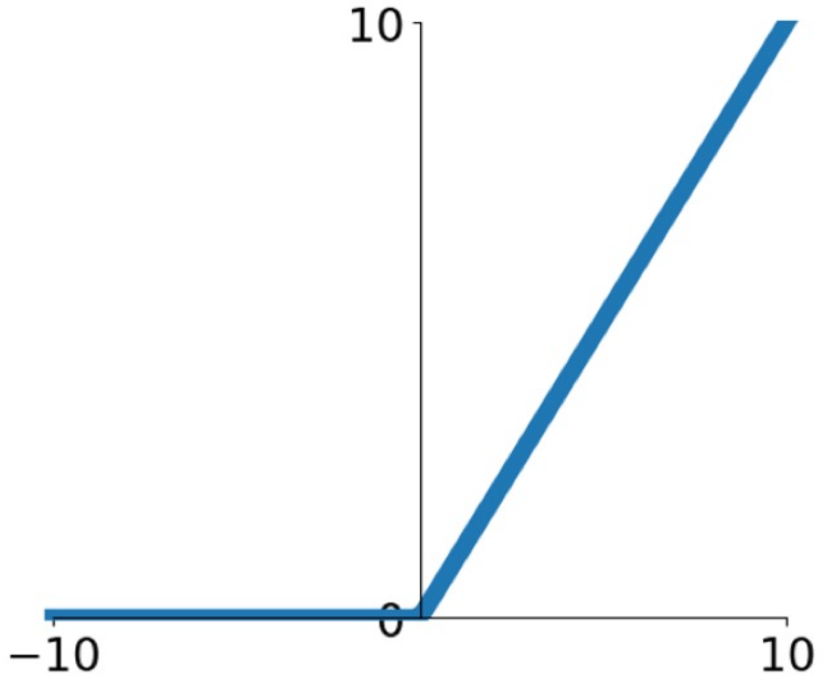
Points not linearly separable in original space

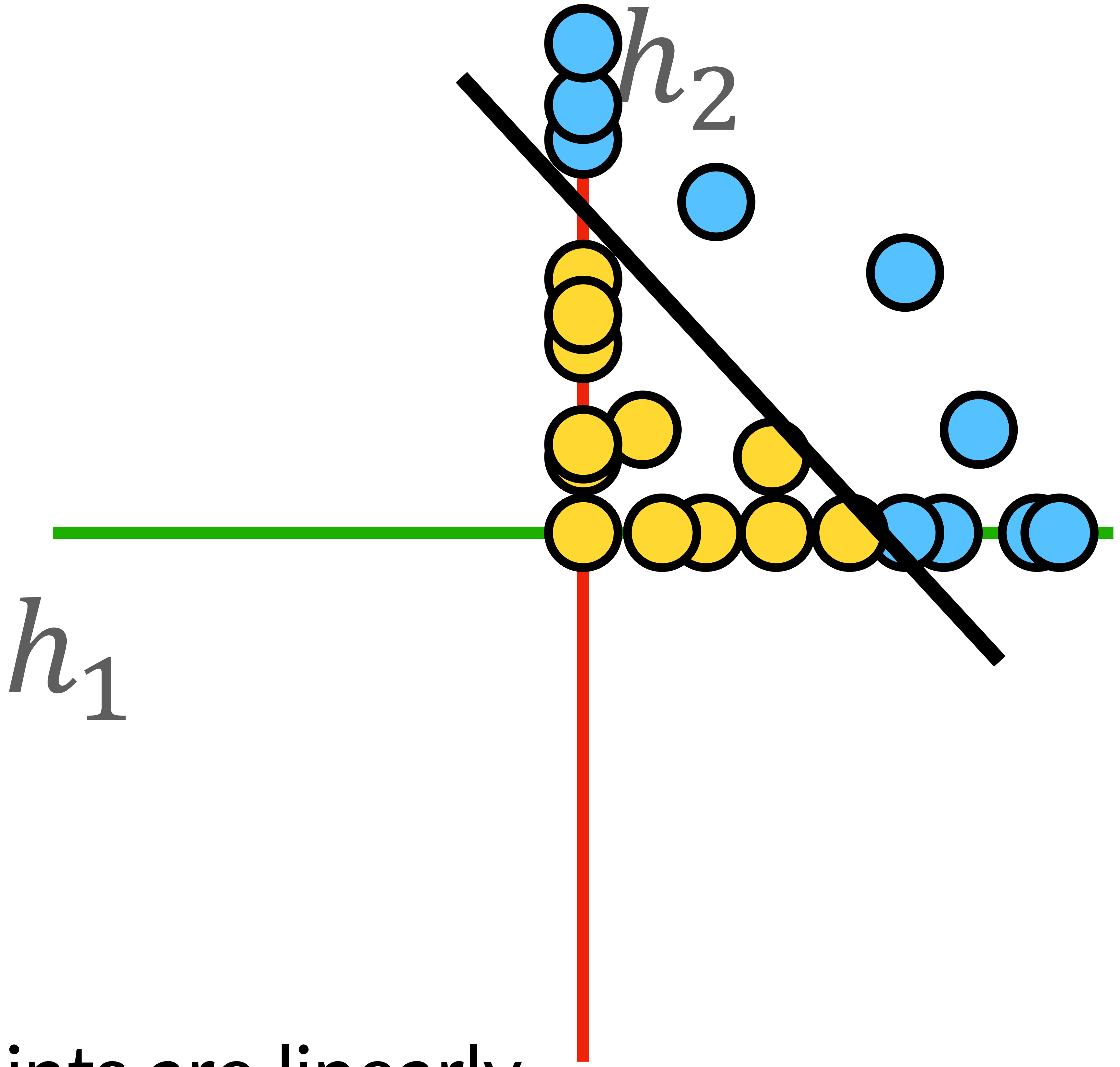
Consider a neural net hidden layer: $h = ReLU(Wx + b) = max(0, Wx + b)$ where x, b, h are each 2-dimensional



Feature transform:

$h = ReLU(Wx + b)$

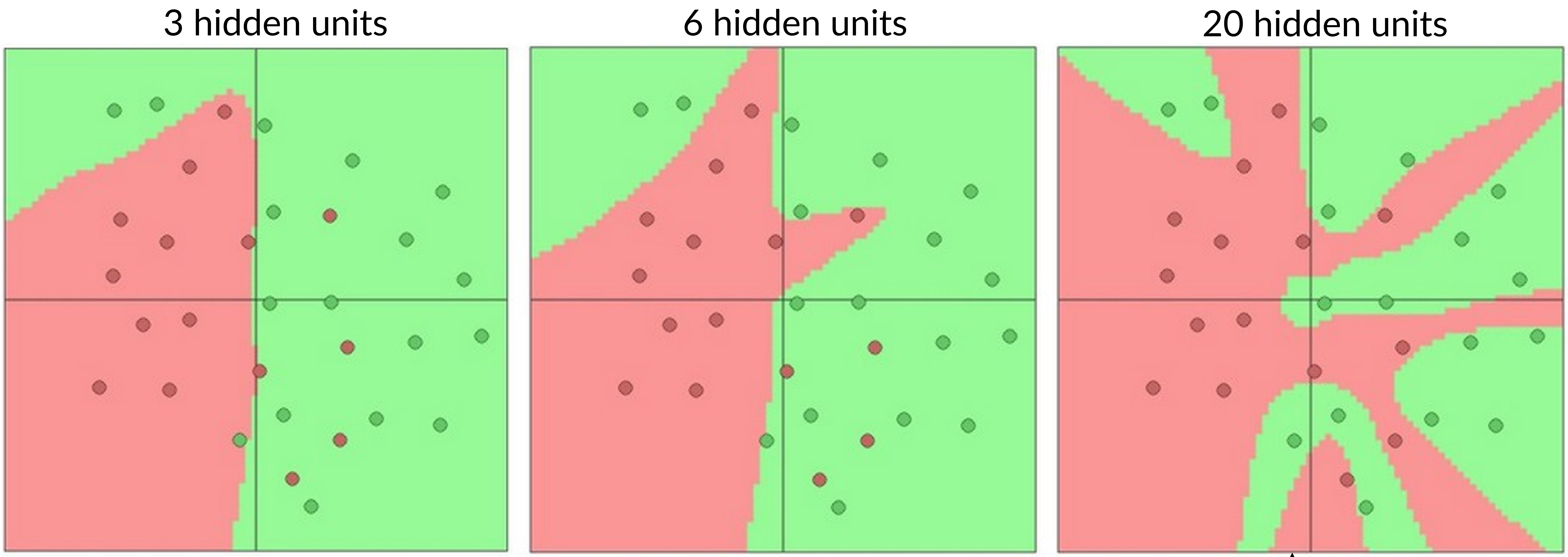




Linear classifier in feature space gives nonlinear classifier in original space

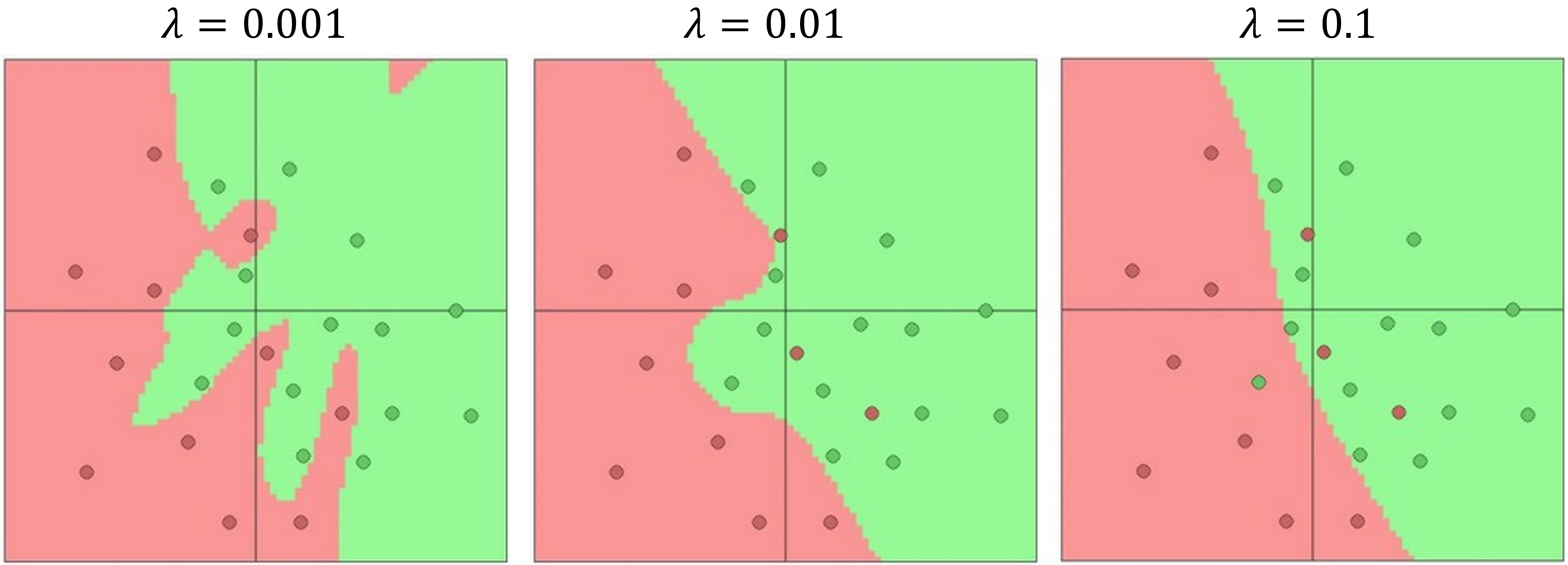
Points are linearly separable in feature space!

Setting the number of layers and their sizes



More hidden units = more capacity

Don’t regularize with size; instead use stronger L2



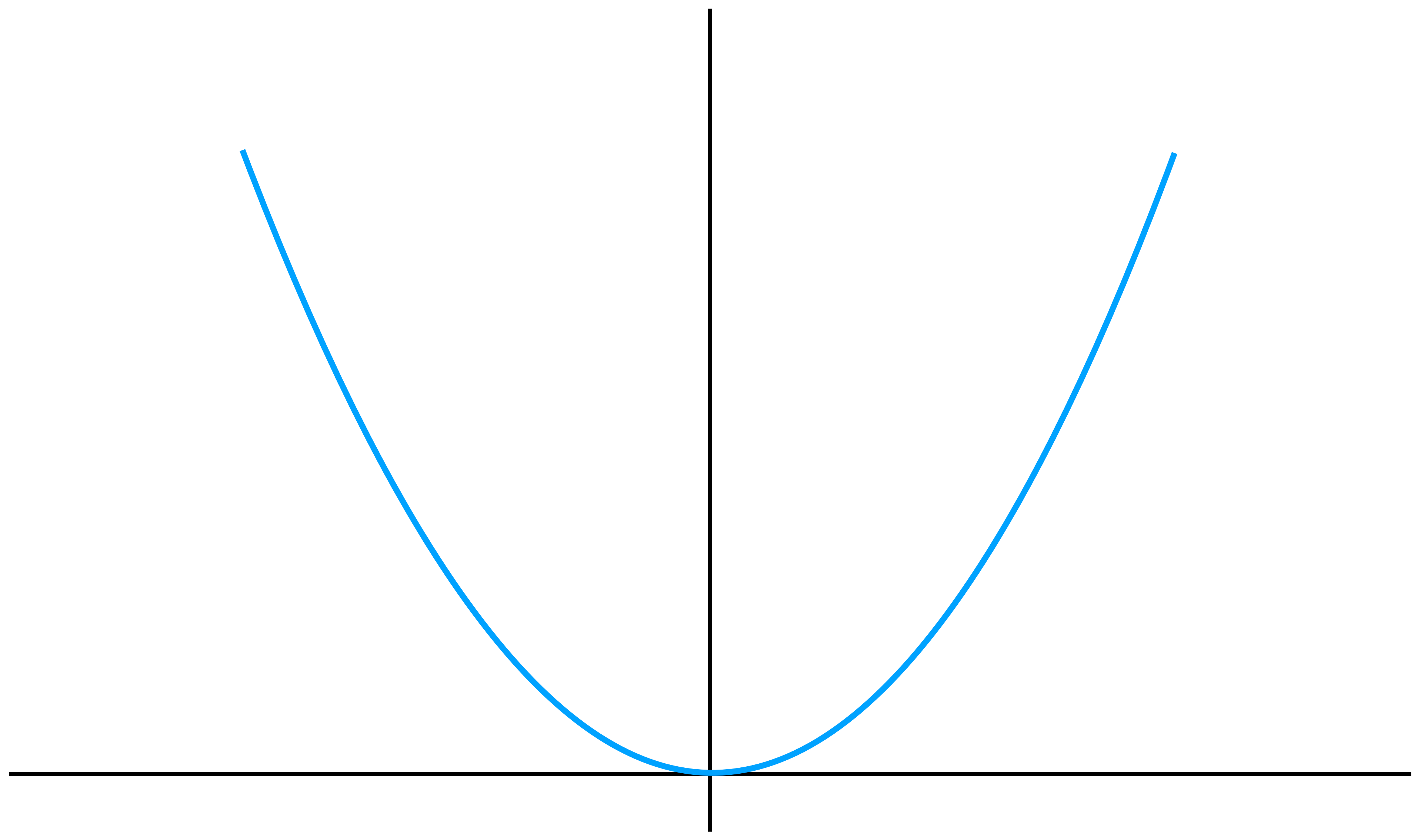
Web demo with ConvNetJS:
https://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html

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:

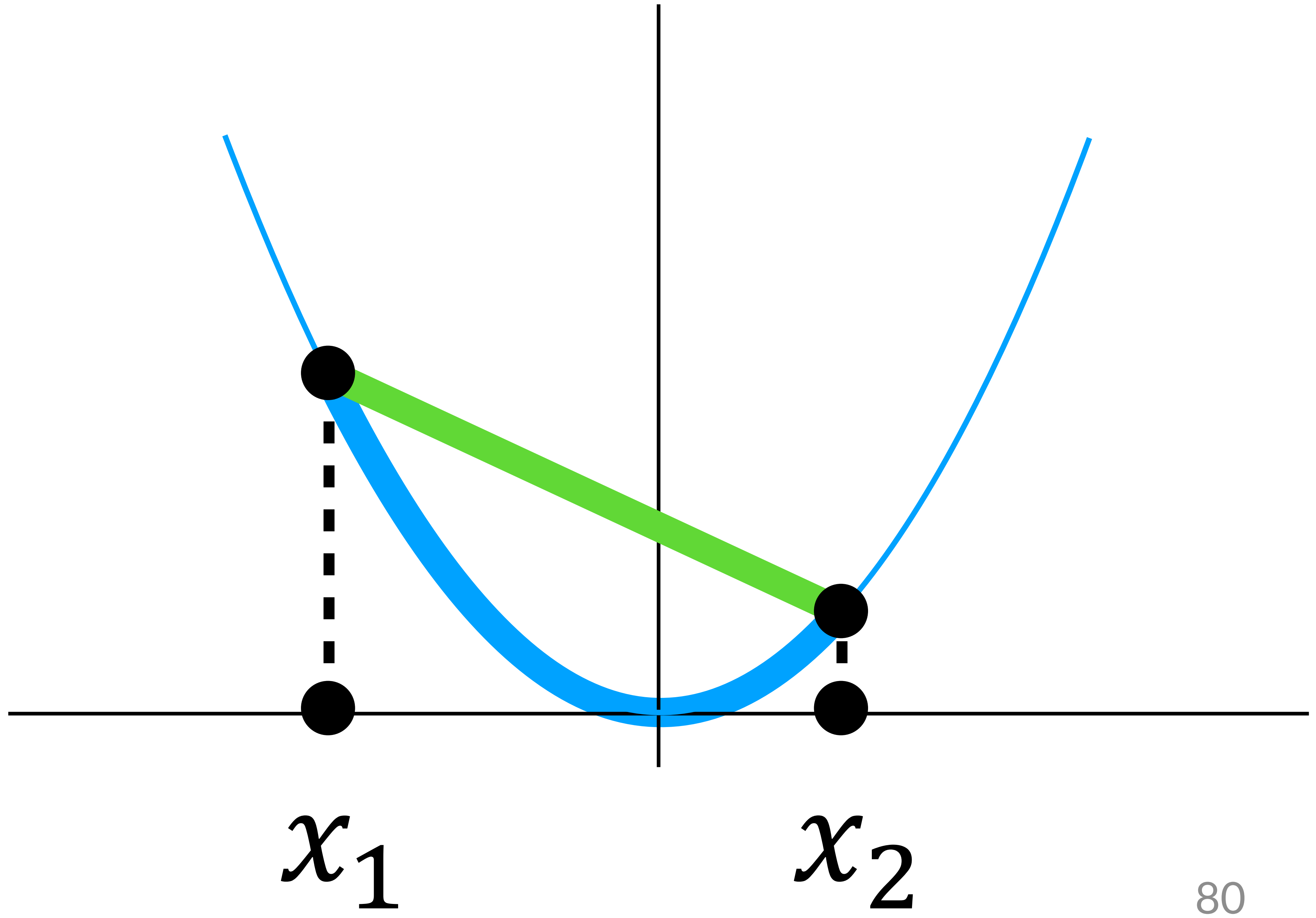


Convex Functions

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:

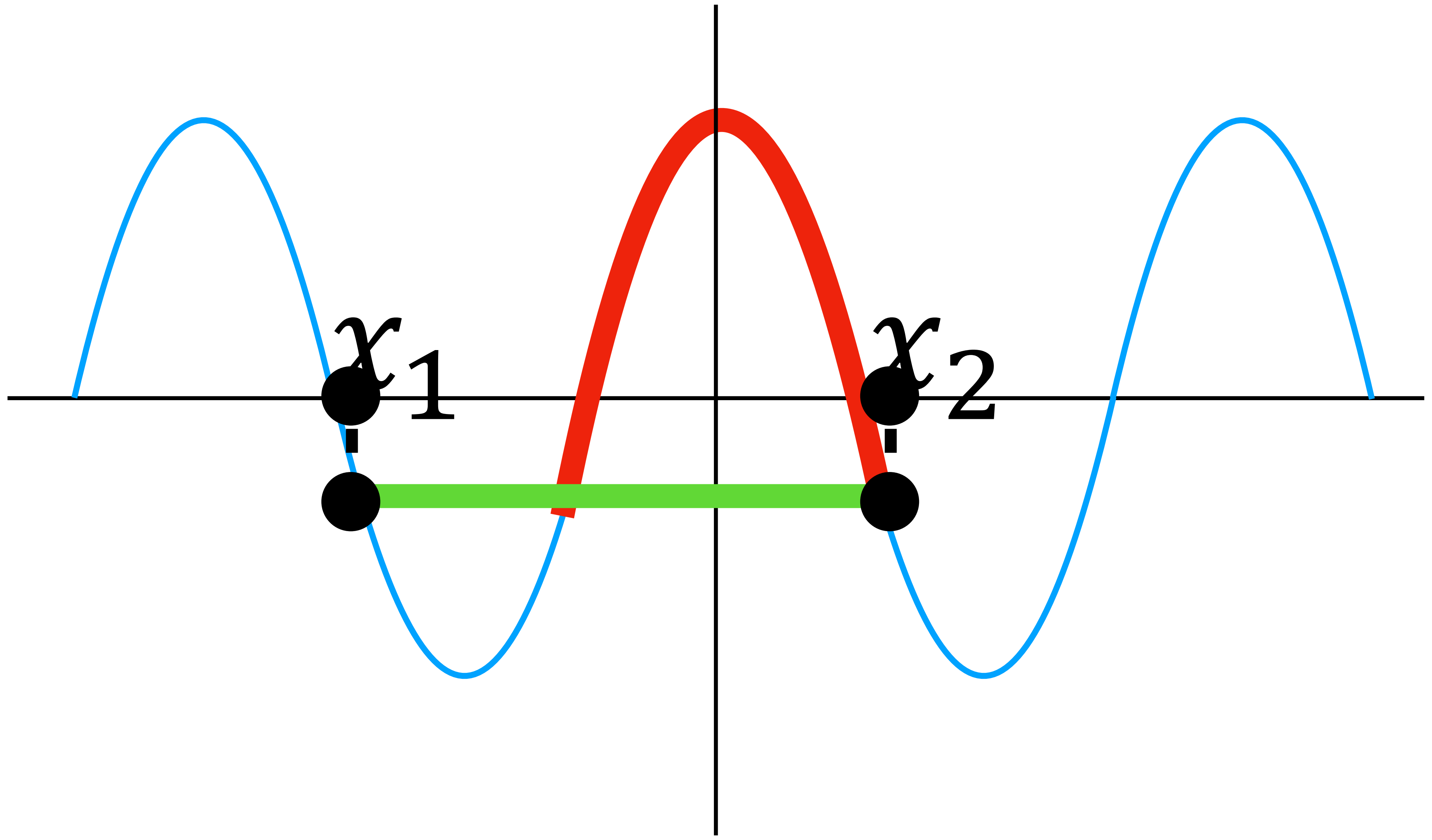


Convex Functions

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:



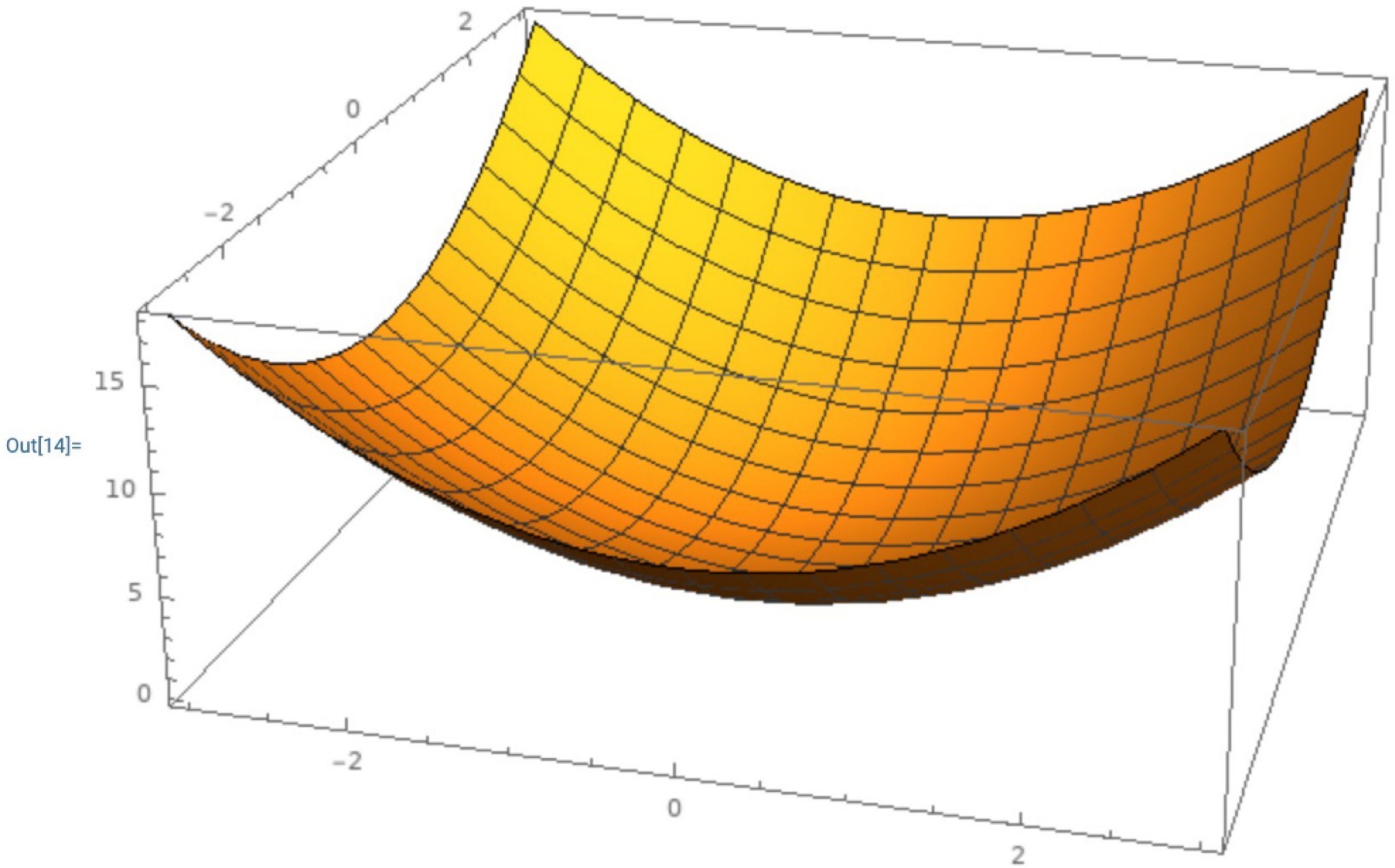
Convex Functions

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

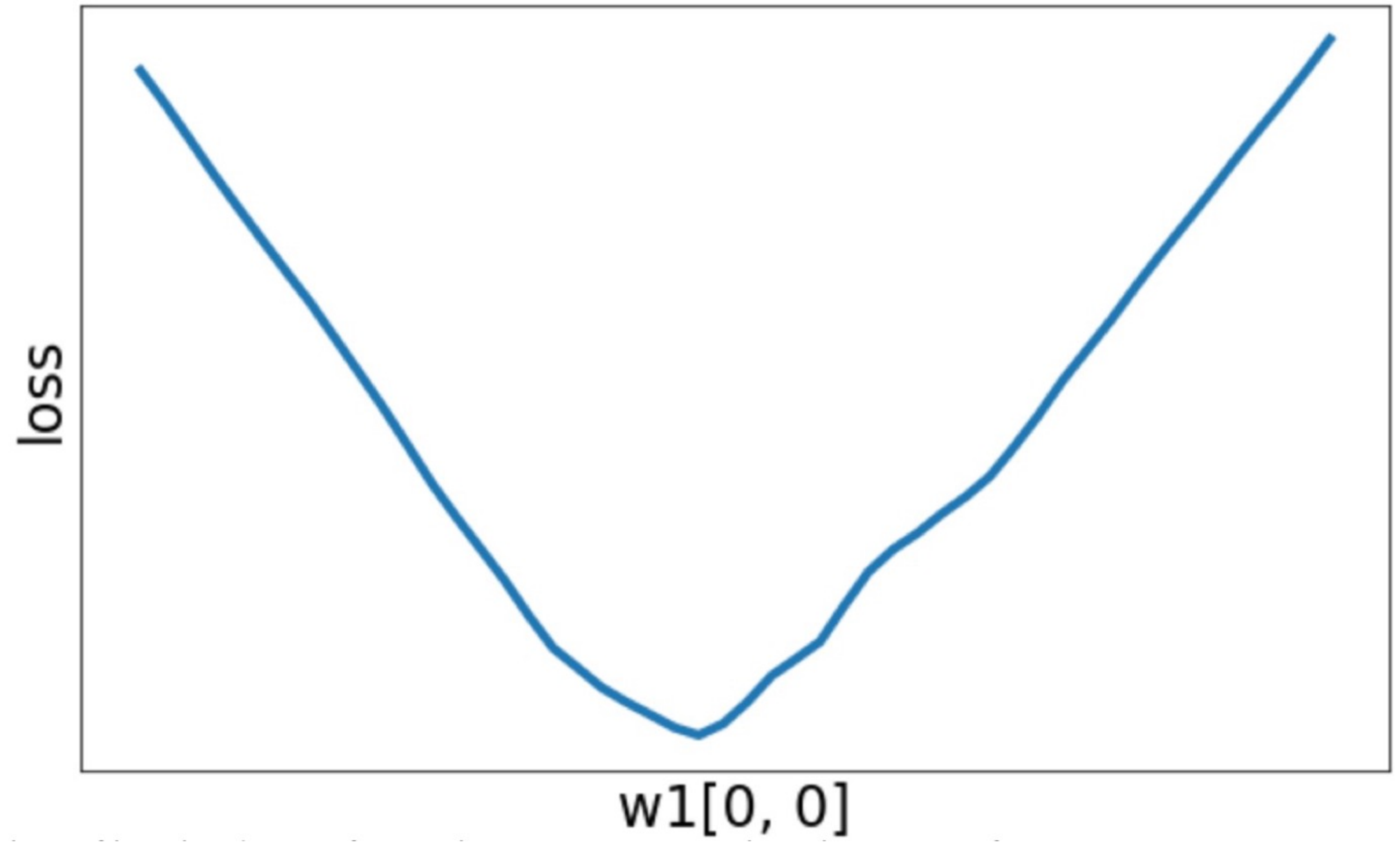
A function $f: X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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***

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

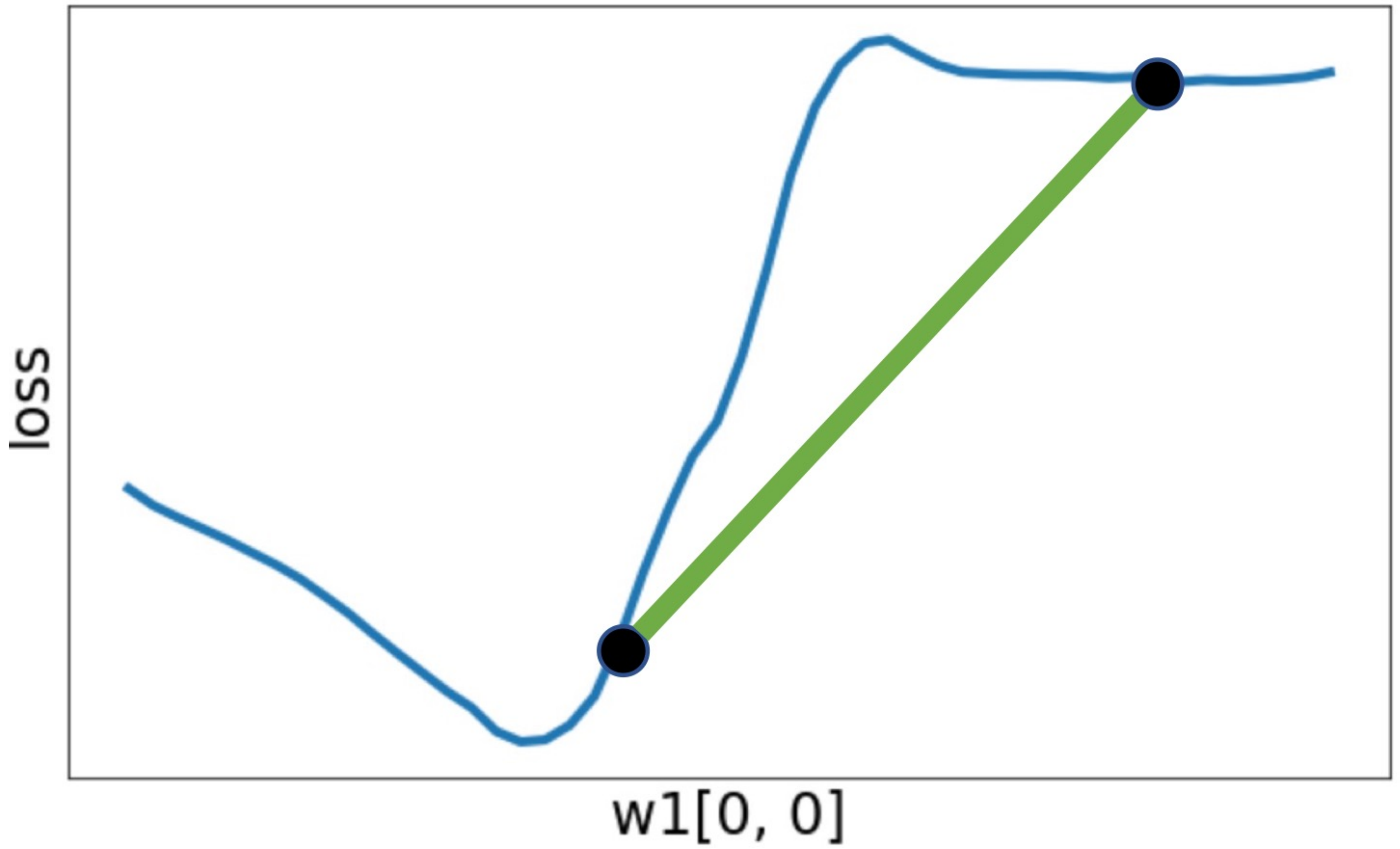
A function $f: X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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

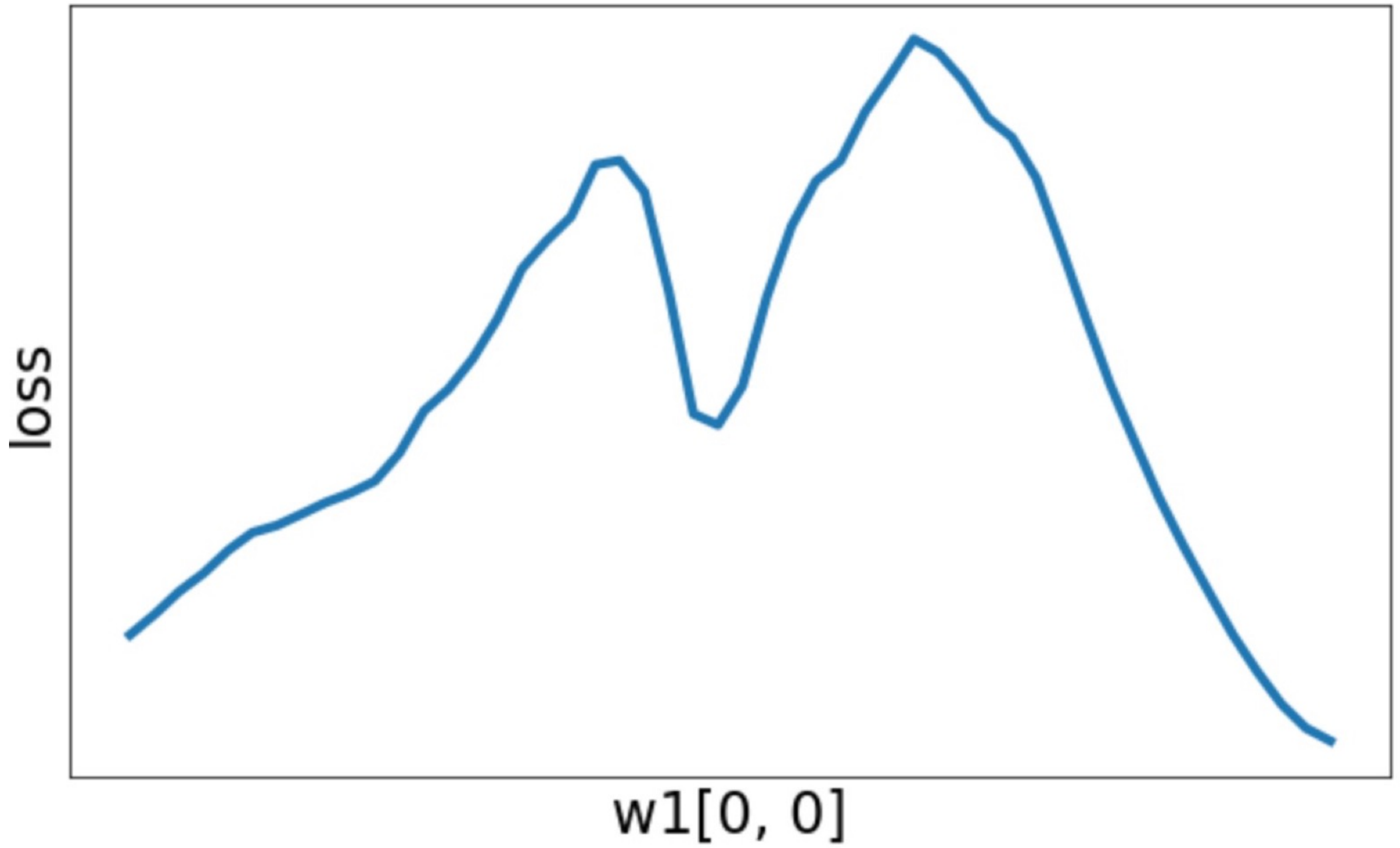
A function $f: X \subseteq \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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

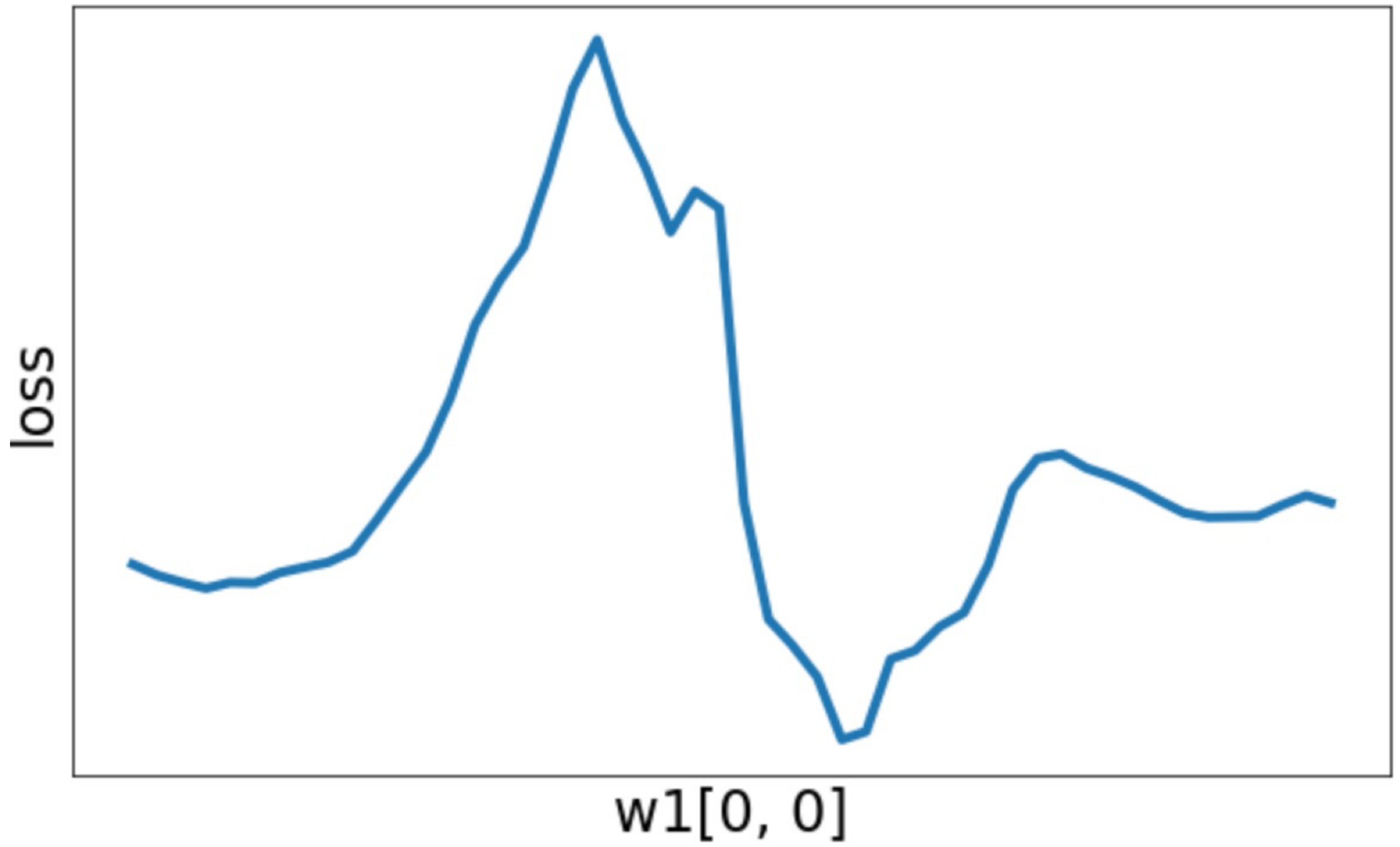
A function $f: X \subseteq \mathbb{R}^N \rightarrow \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0,1]$,

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

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

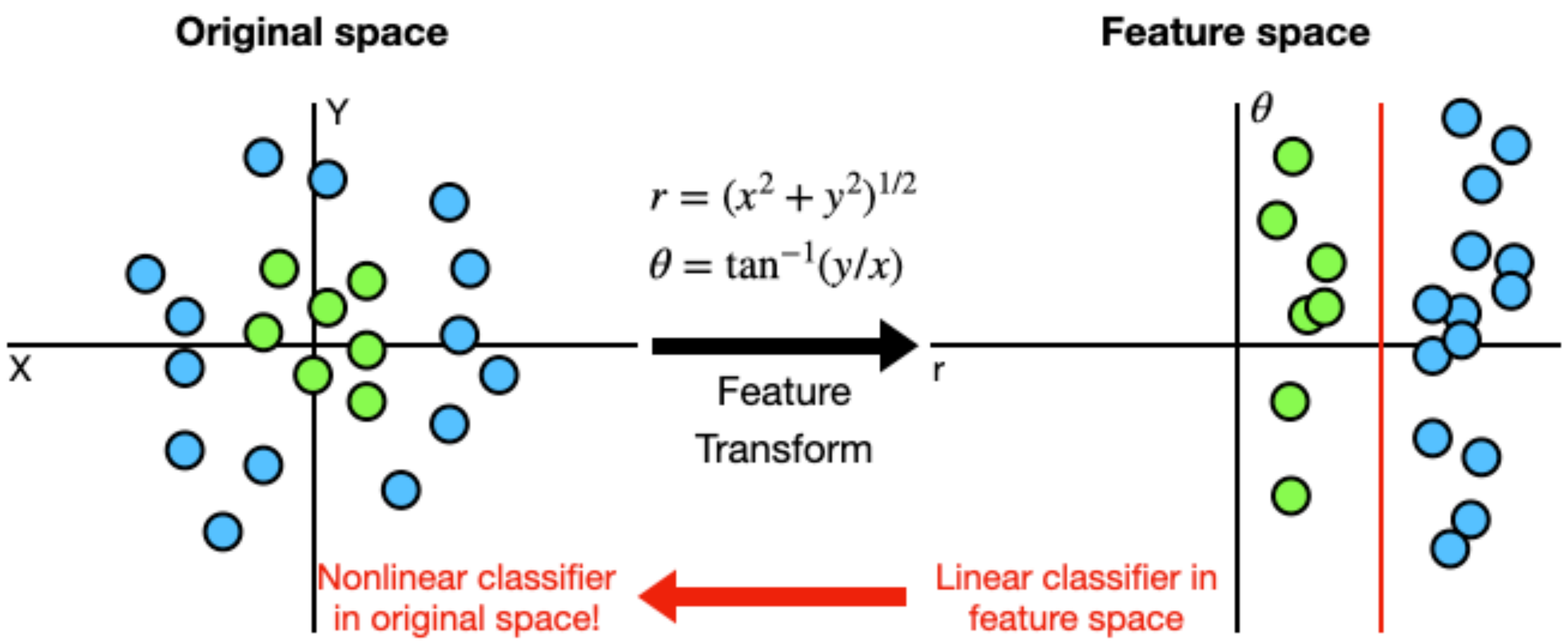
Generally speaking, convex functions are **easy to optimize**: can derive theoretical guarantees about **converging to global minimum***

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

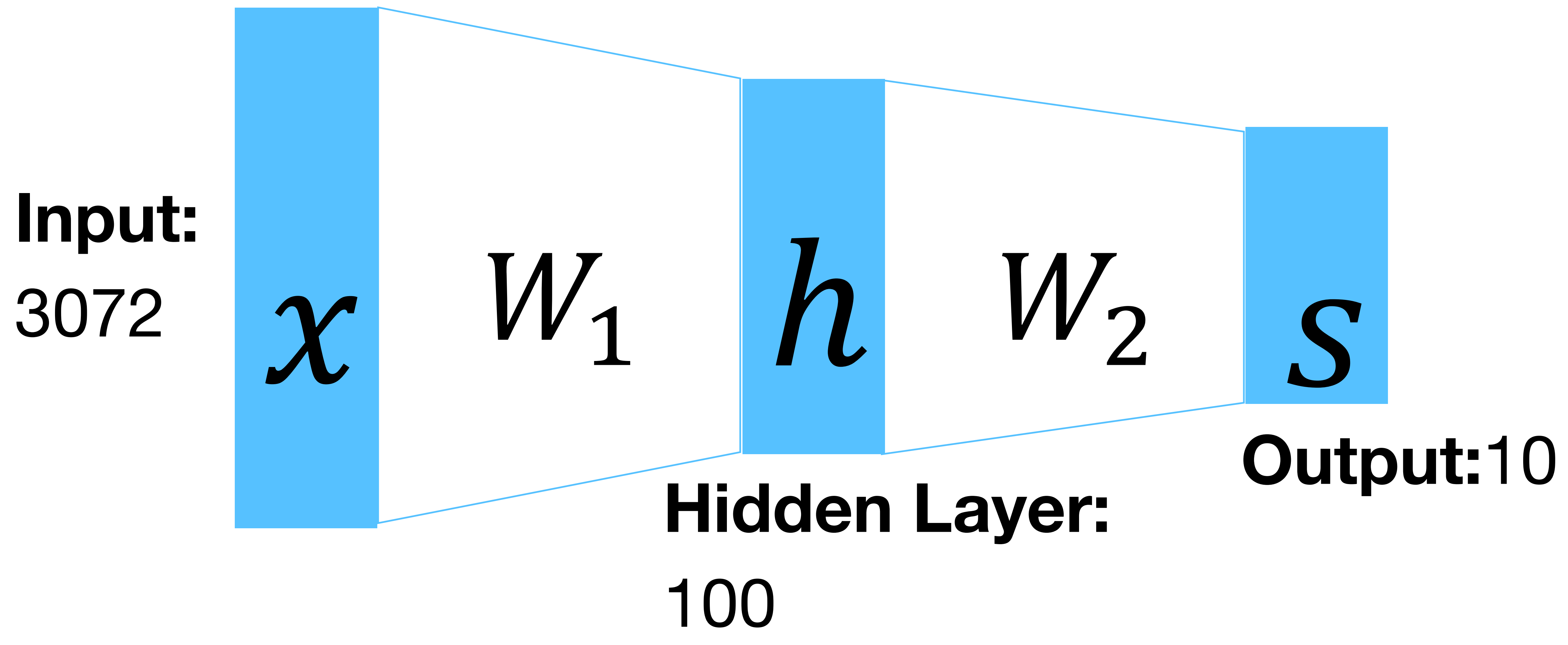


Neural Networks as learnable feature transforms

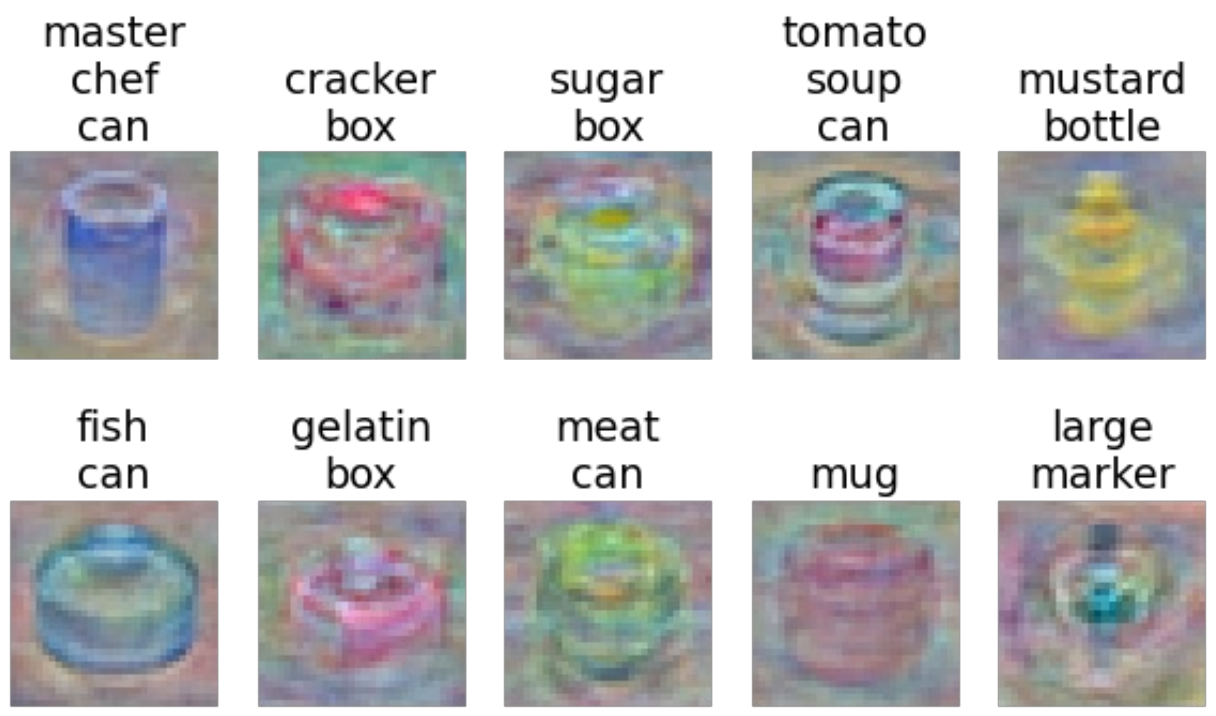
Summary

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

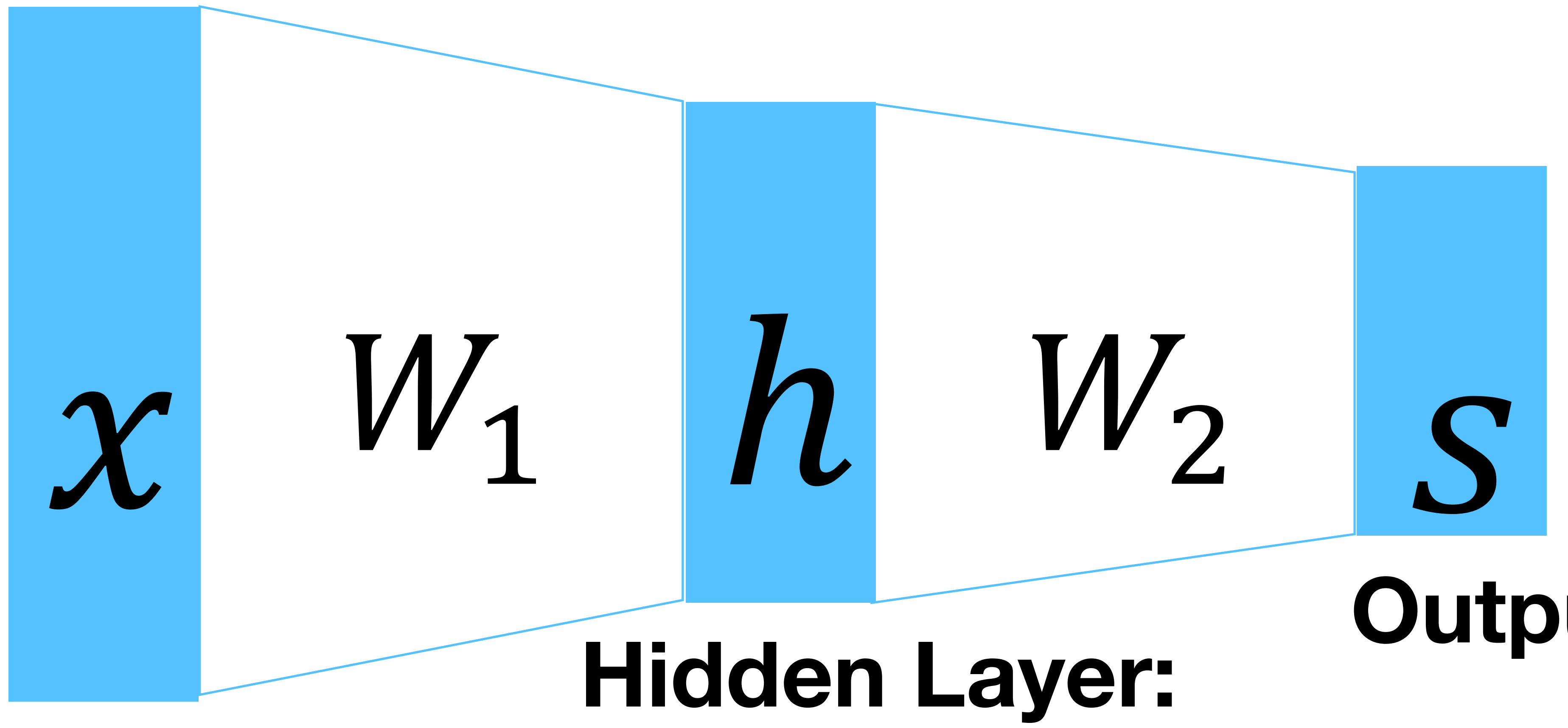


Neural networks: Many reusable templates

Summary

From linear classifiers to
fully-connected networks

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

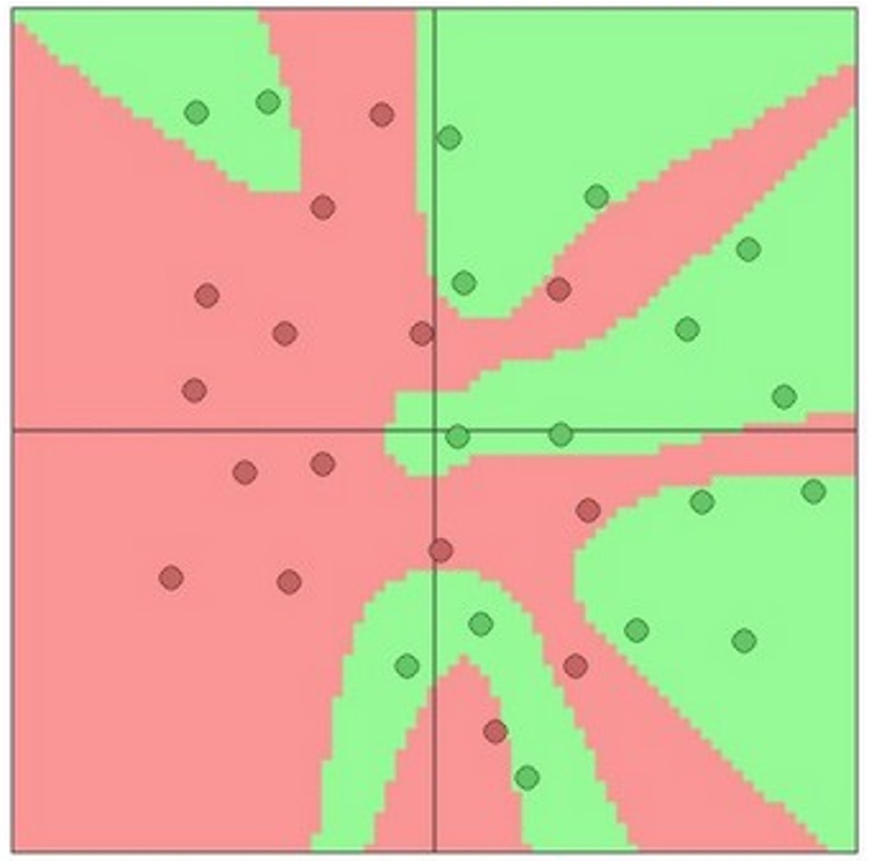


Input: 3072

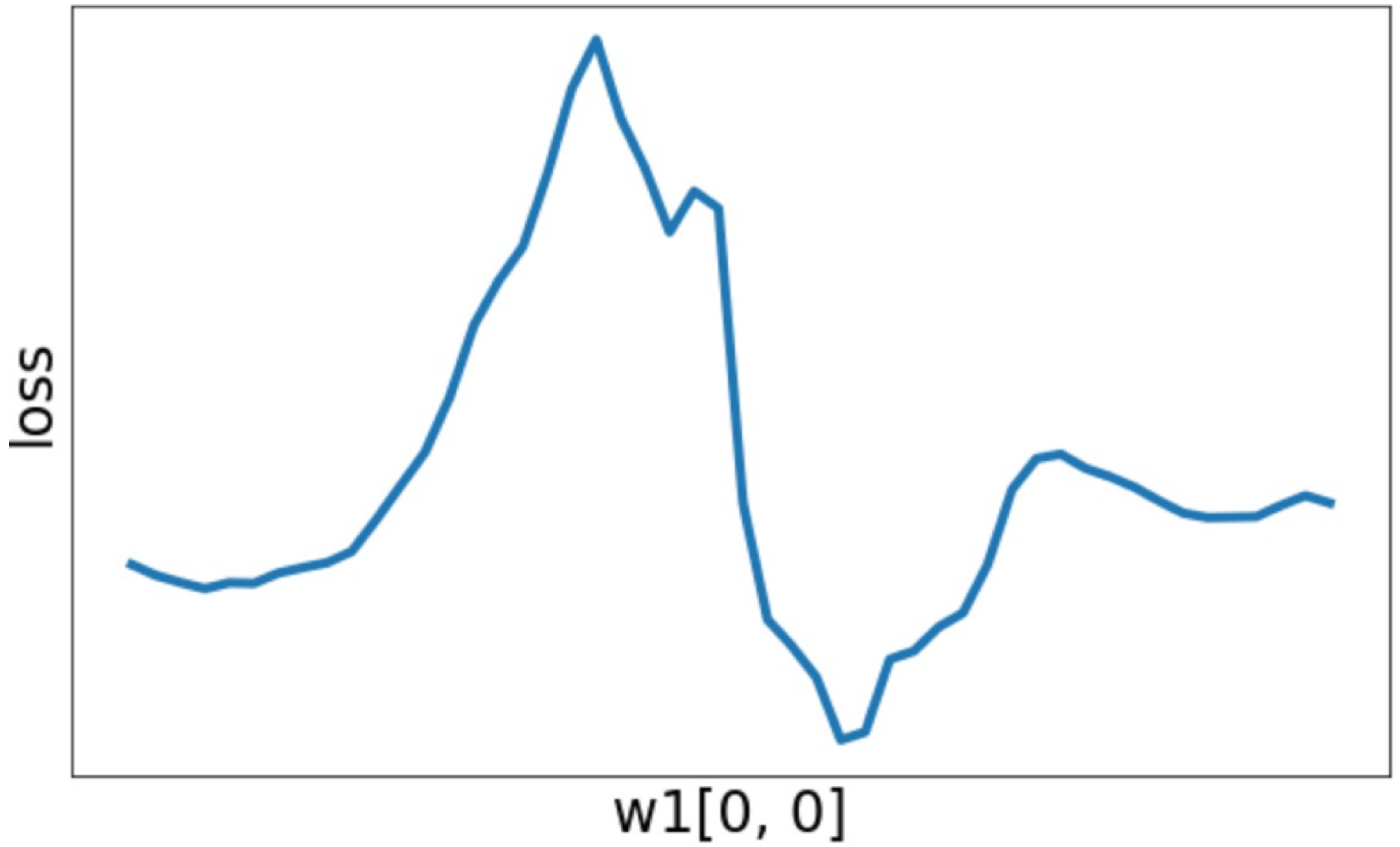
Hidden Layer: 100

Output: 10

Feature Space Warping



Nonconvex



DeepRob

Lecture 4
Neural Networks
University of Michigan | Department of Robotics

DeepRob