CS163: Deep Learning for Computer Vision

Lecture 5: Neural Network/Multilayer Perceptron

Courtesy of Prof. Justin Johnson at Umich and the team of Stanford CS231n for the part of slides

Annoucement

- Assignment 1 is due this weekend (Sunday Oct 20)
- Assignment 2 is released at https://github.com/UCLAdeepvision/CS163-Assignments-2024Fall/tree/main/Assignment2
  - It is due on Sunday, Nov 3.
  - There is a MiniPlaces classification challenge for this assignment, and the top 20% of students can get a 10-point bonus. You can access the challenge here: https://www.kaggle.com/t/9cb9ae990d5f4576b5fb207ee9bf39dc

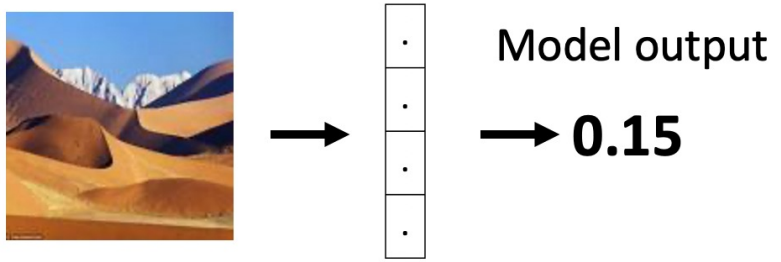
Announcement

- Course project: by the end of week 4, form group and book presentation slot:
  - List of topics: https://docs.google.com/spreadsheets/d/16jvVrBh0Cubjs0jANzVaXGRupXCxUhOWKFYq9hJcKng/edit?usp=sharing
  - Team & Presentation slot: https://docs.google.com/spreadsheets/d/1YEOOUwaPlMJ2gD_VG7AT7SG4XoNLTNmIGD9h-m0xXsQ/edit?usp=sharing
- Registration info for Google cloud education credit has been sent out (See the announcement in the Bruinlearn if you miss it)
  - Check the Tutorial slide in the discussion session last Friday

Previous Week:

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

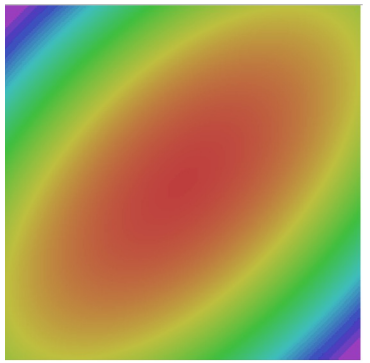
$$s = f(x; W) = Wx$$



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

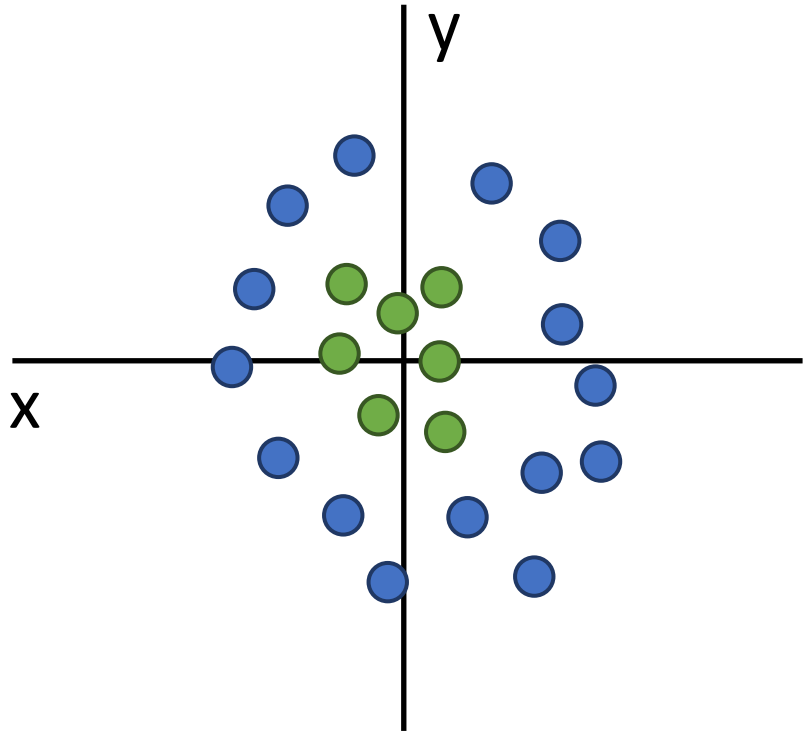
$$L = \frac{1}{N}\sum_{i=1}^{N} L_i + R(W)$$

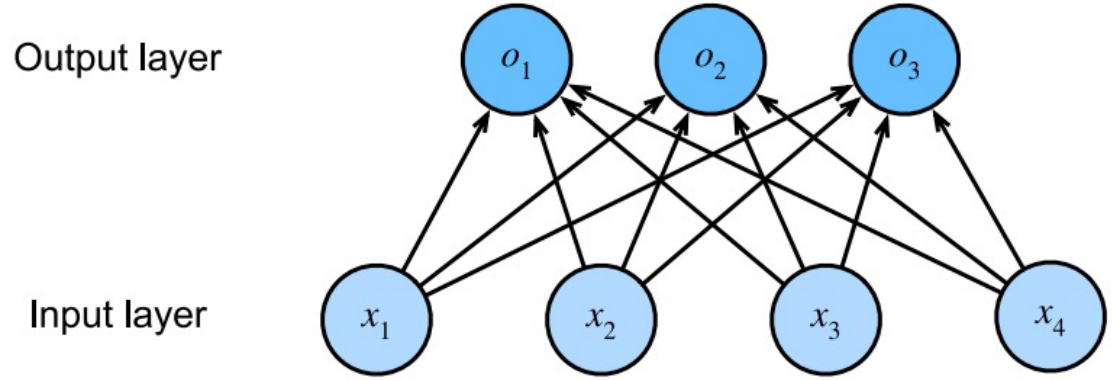
```
v = 0
for t in range(num_steps):
  dw = compute_gradient(w)
  v = rho * v + dw
  w -= learning_rate * v
```



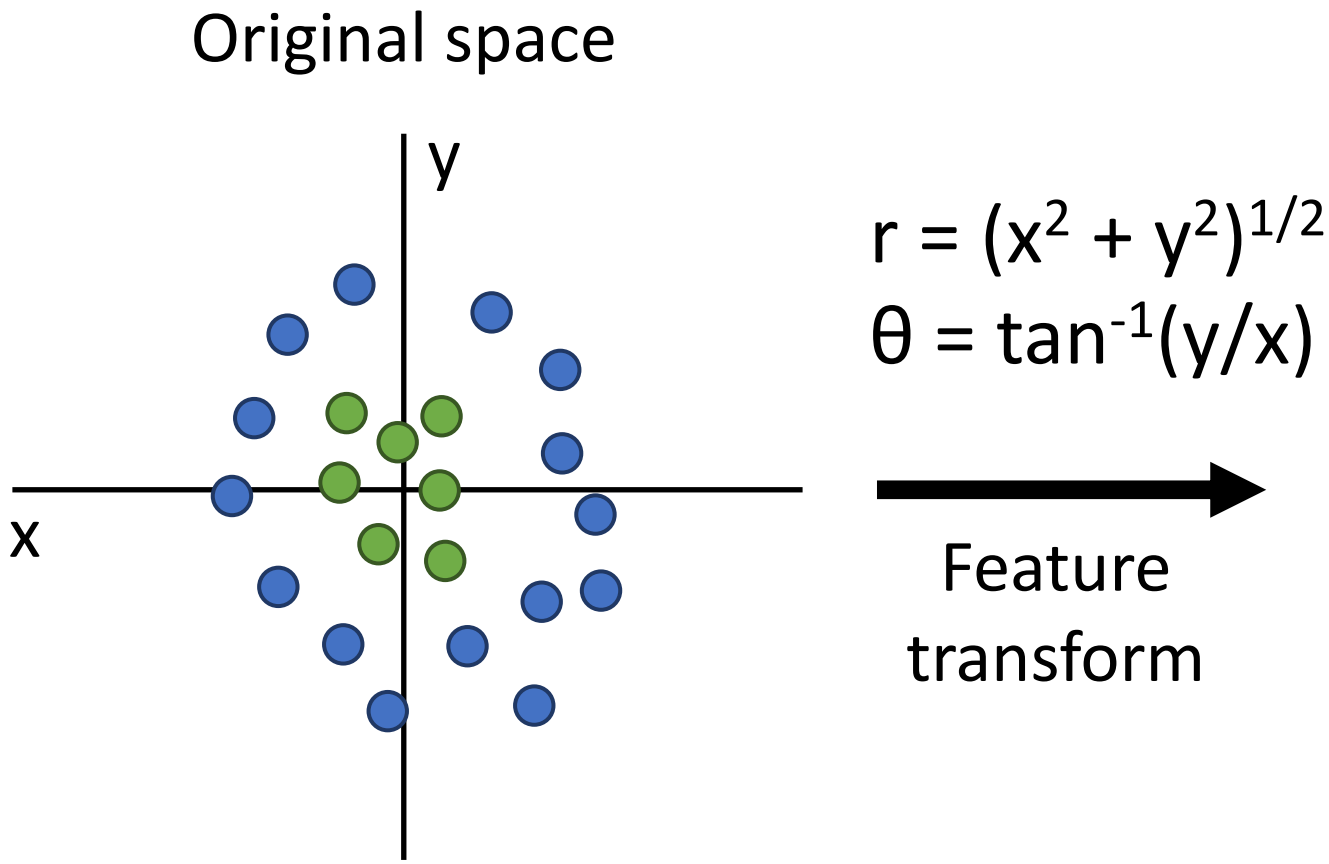
Problem: Linear Classifiers aren't that powerful

Linear model only has linear decision boundary

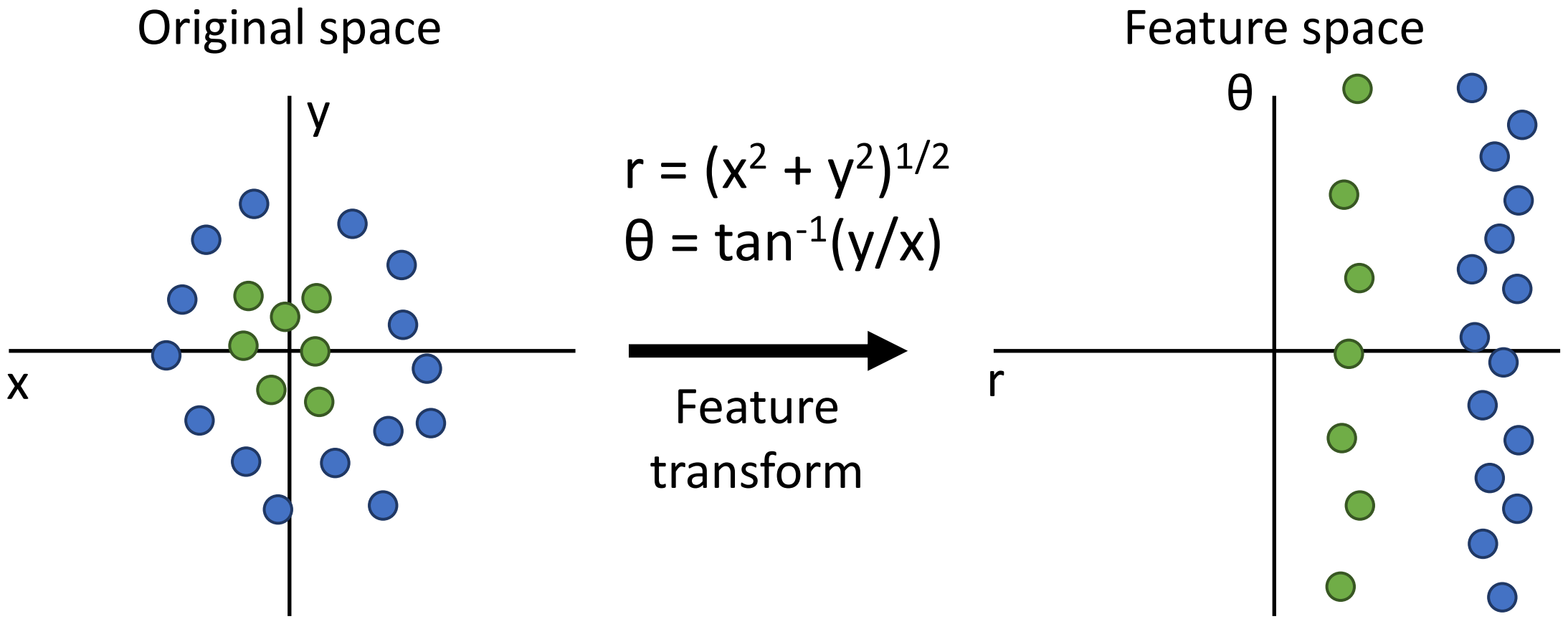




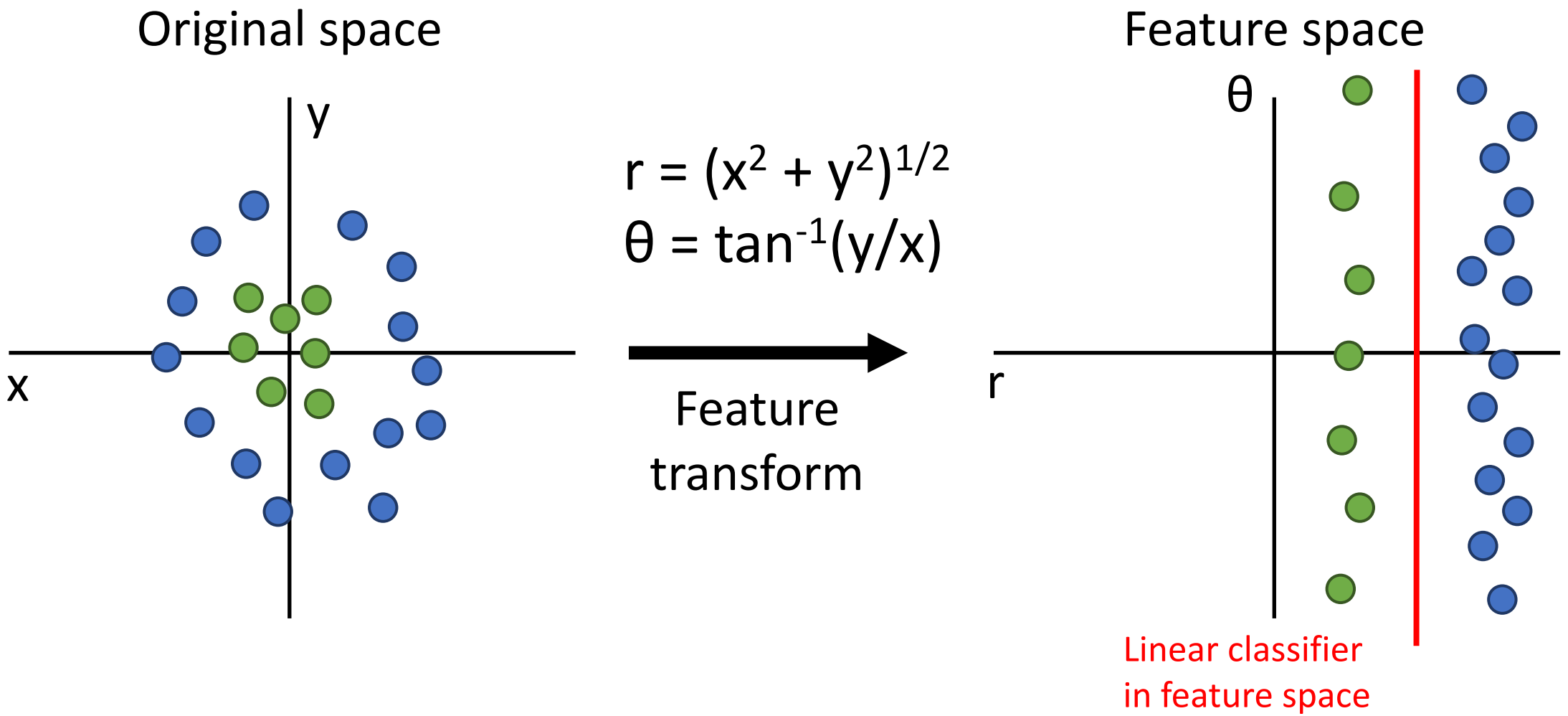
One solution: **Feature Transforms**



One solution: **Feature Transforms**



One solution: **Feature Transforms**



One solution: **Feature Transforms**

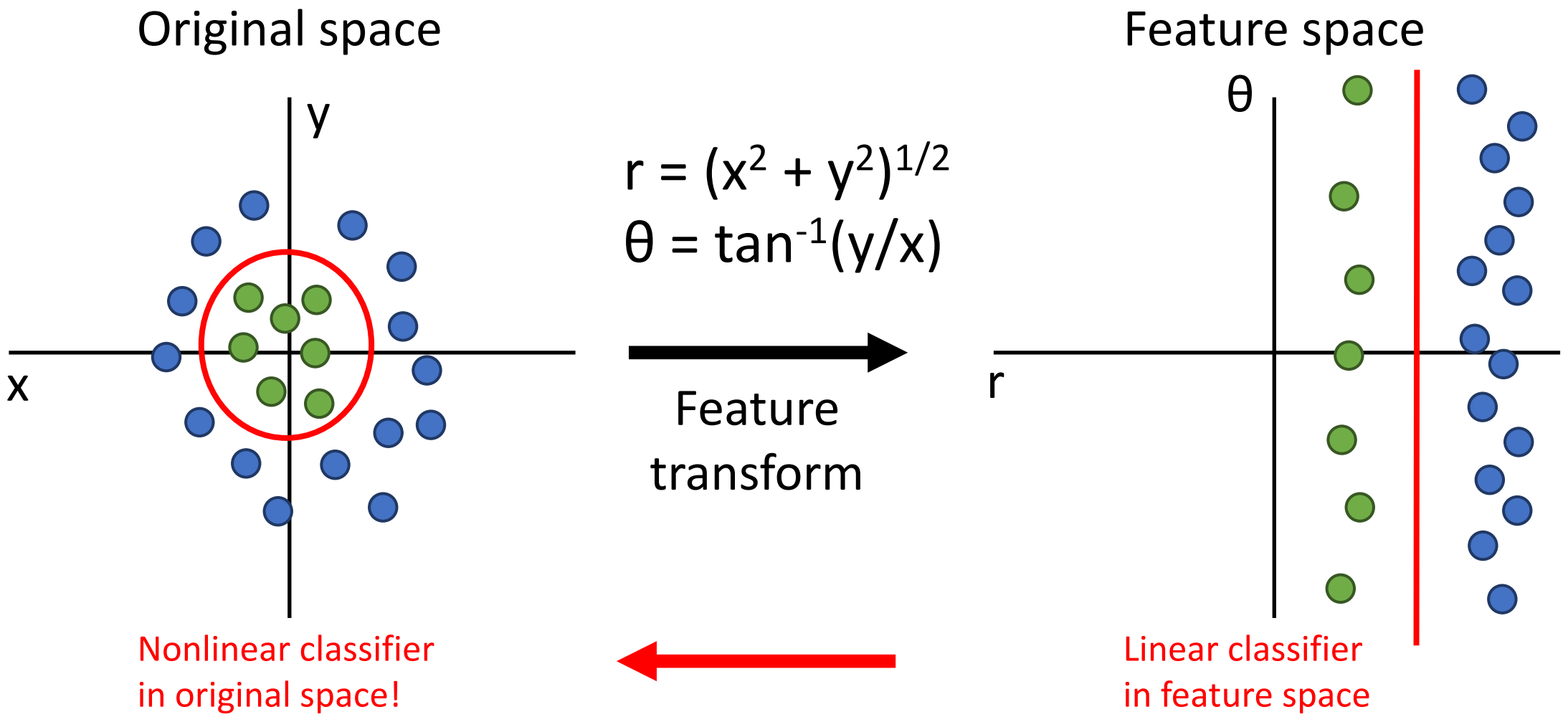
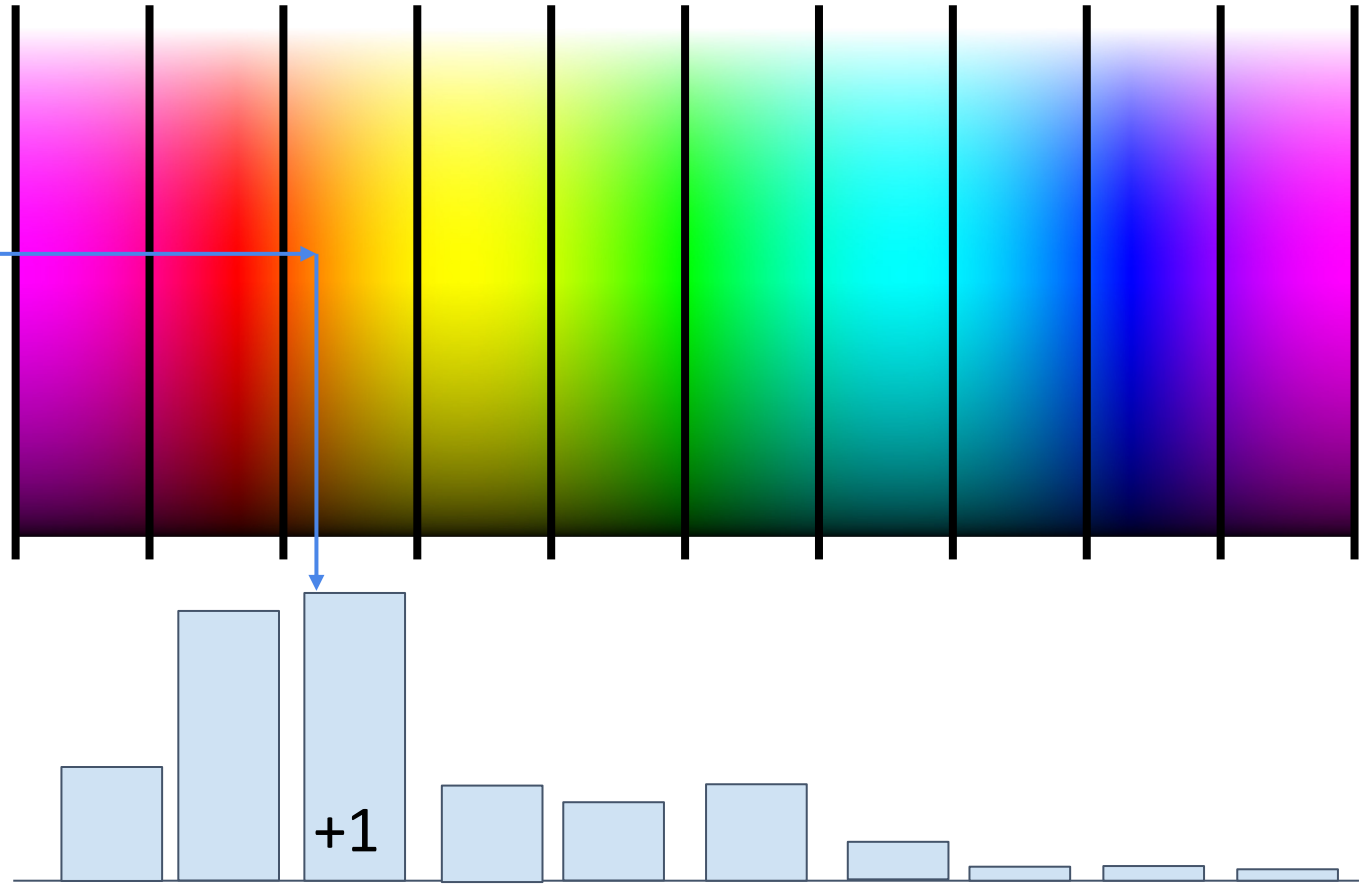


Image Features: Color Histogram



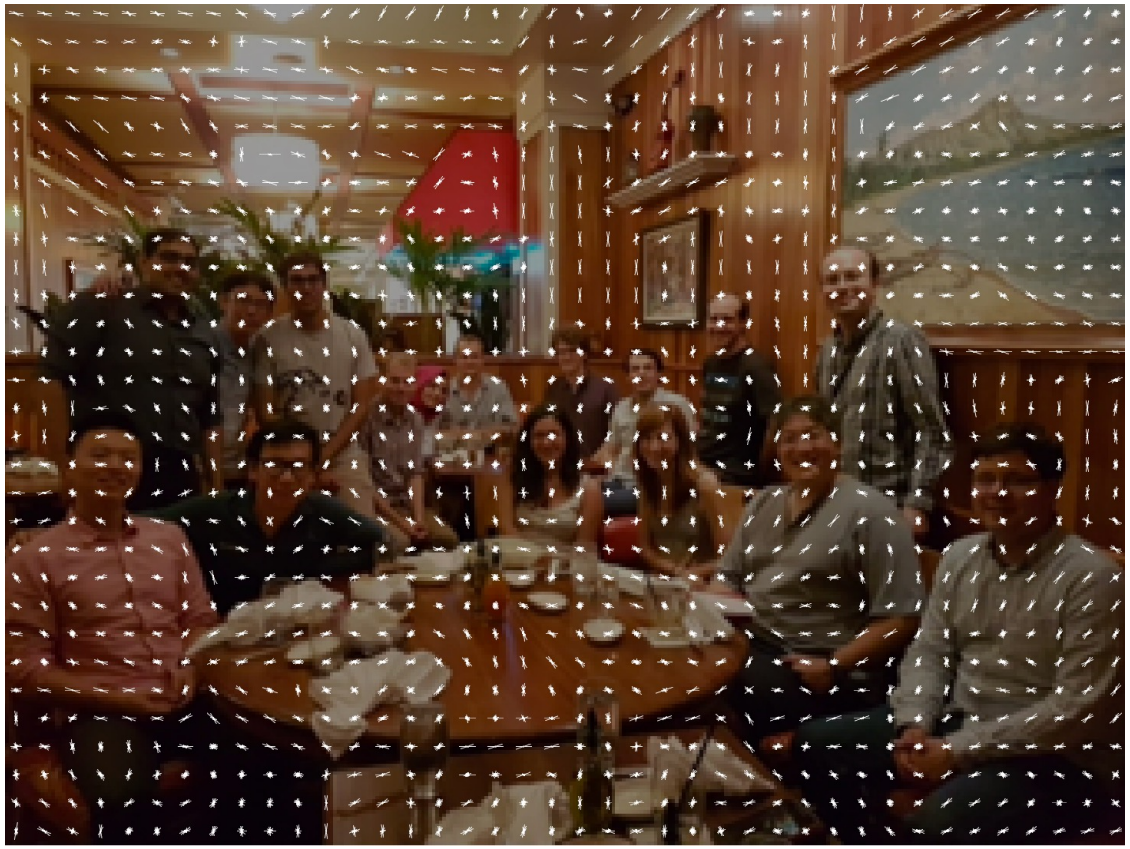
Ignores texture,
spatial positions

Image Features: Histogram of Oriented Gradients (HoG)

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

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

Image Features: Histogram of Oriented Gradients (HoG)



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

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

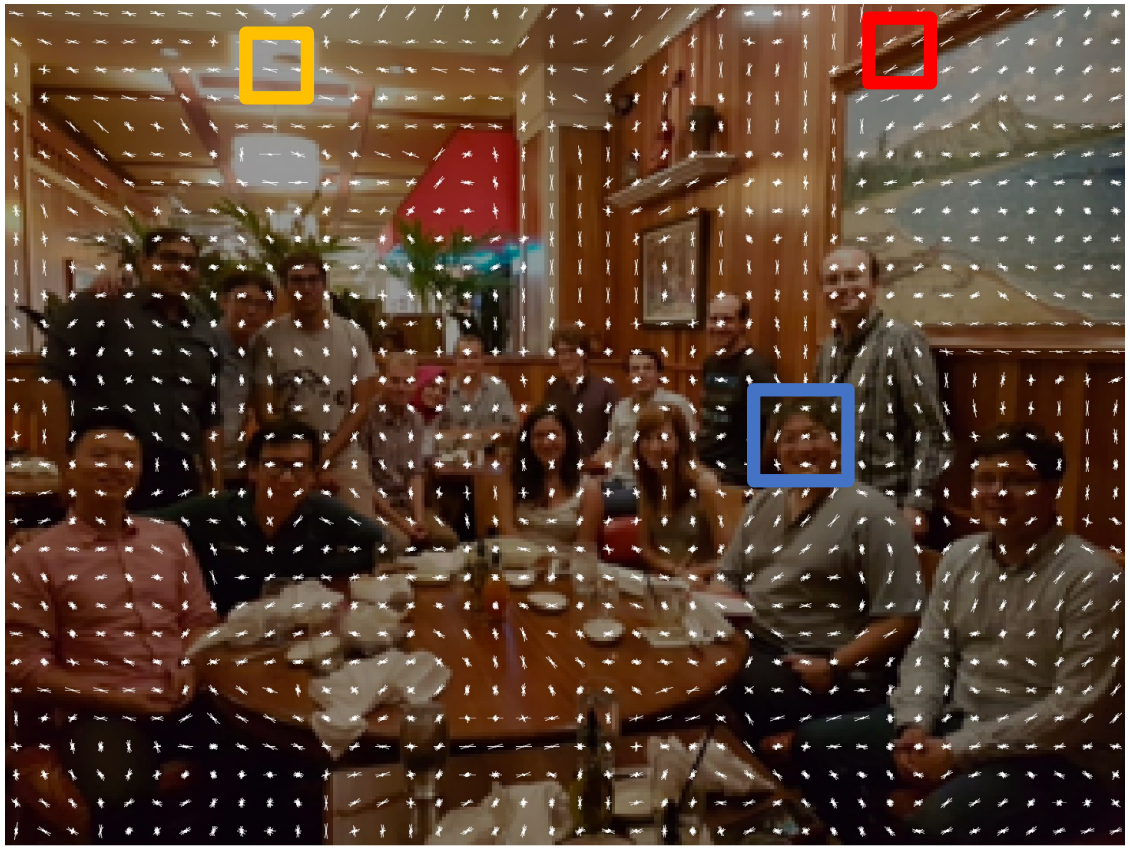
Lowe, “Object recognition from local scale-invariant features”, ICCV 1999
Dalal and Triggs, "Histograms of oriented gradients for human detection," CVPR 2005

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 directions weighted by edge strength

Example: 320x240 image gets divided into 40x30 bins; 8 directions per bin plus zero direction (no edge); feature vector has 30*40*9 = 10,800 numbers

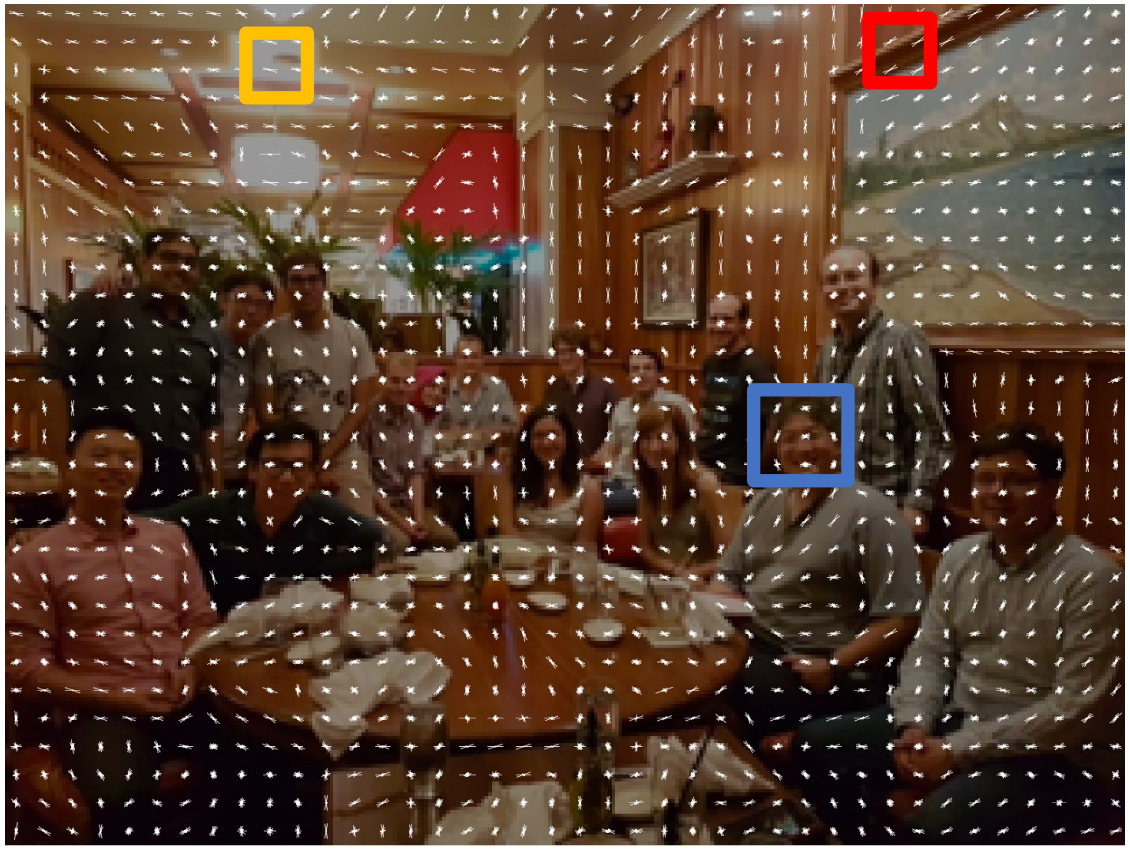
Image Features: Histogram of Oriented Gradients (HoG)

Weak edges

Strong diagonal edges

Edges in all directions

Captures texture and position, robust to small image changes



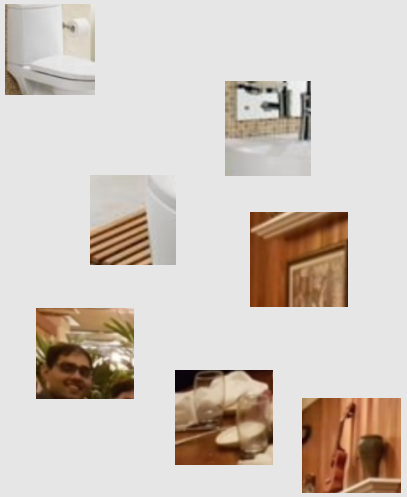
1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

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

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

Step 1: Build codebook

Extract random patches

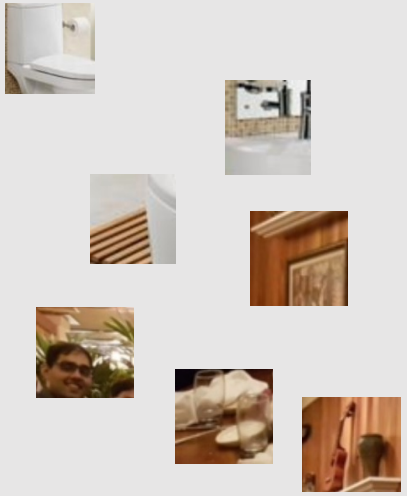


Cluster patches to form “codebook” of “visual words”

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

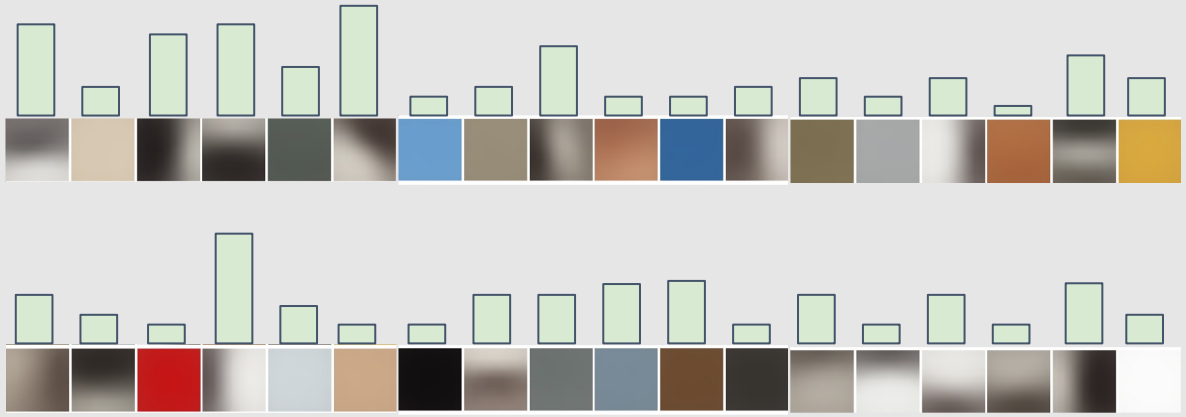
Step 1: Build codebook

Extract random patches



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

Step 2: Encode images



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

Image Features

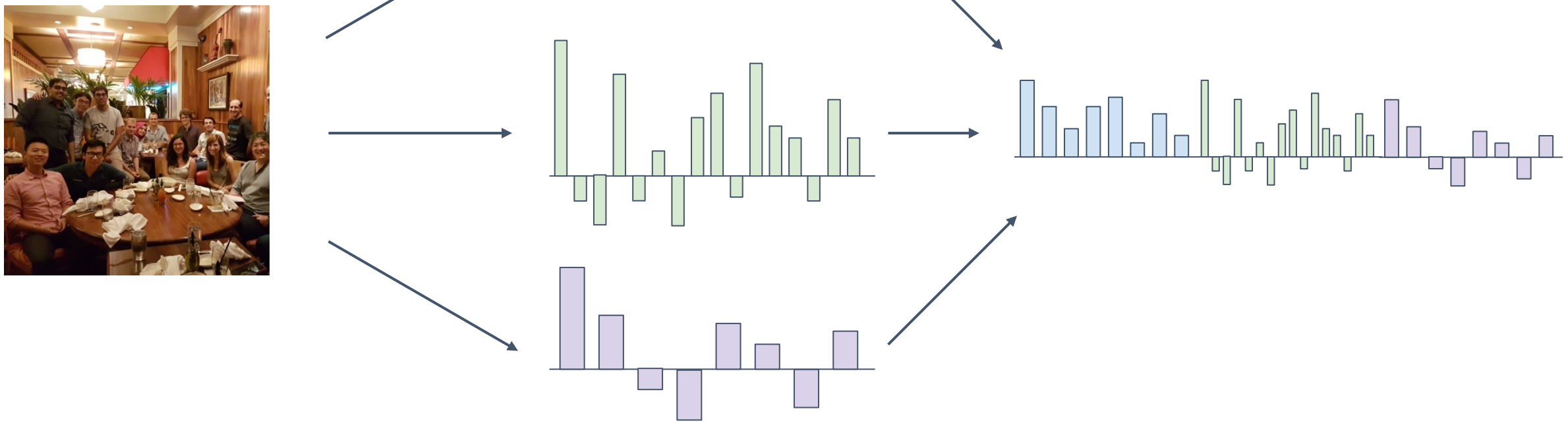
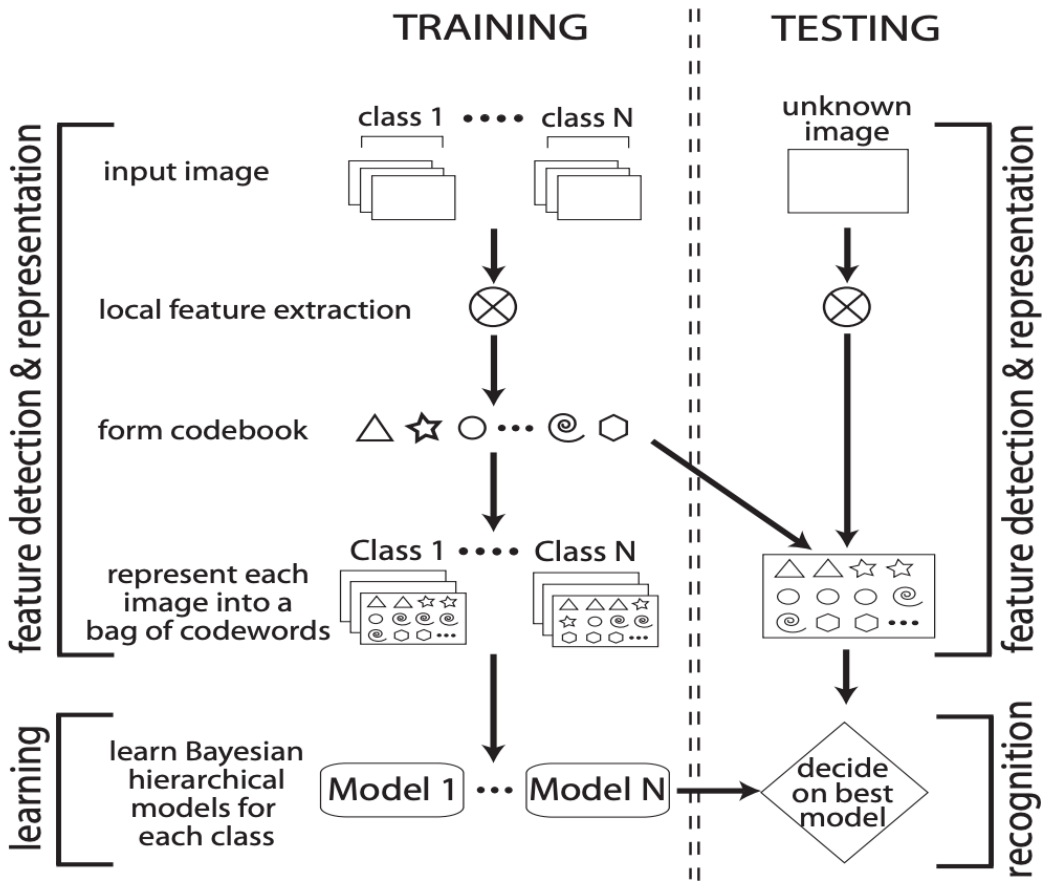
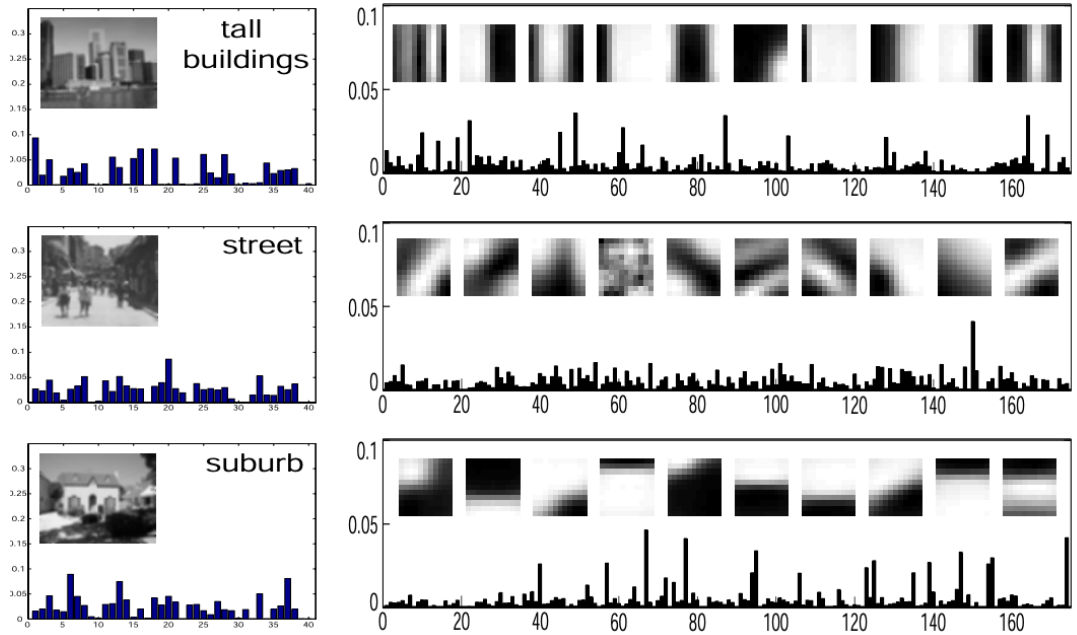


Image Features: Bag of Words (Data-Driven!)



13 scene classification



Feifei and Perona. A Bayesian Hierarchical Model for Learning Natural Scene Categories. CVPR'05.

Example: Winner of 2011 ImageNet challenge

Low-level feature extraction ≈ 10k patches per image

- SIFT: 128-dim
- color: 96-dim

} reduced to 64-dim with PCA

FV extraction and compression:

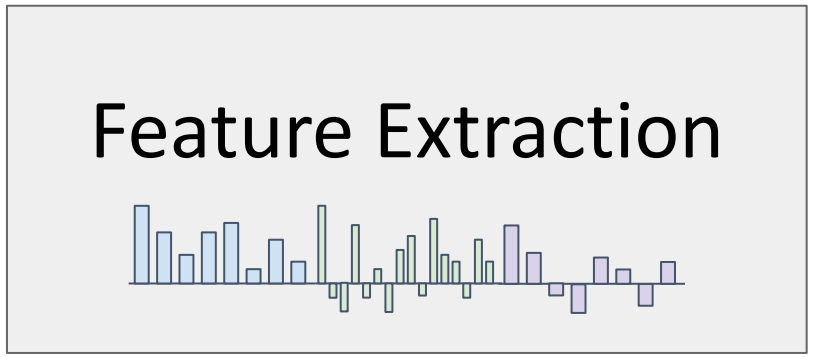
- N=1,024 Gaussians, R=4 regions ⇨ 520K dim x 2
- compression: G=8, b=1 bit per dimension

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

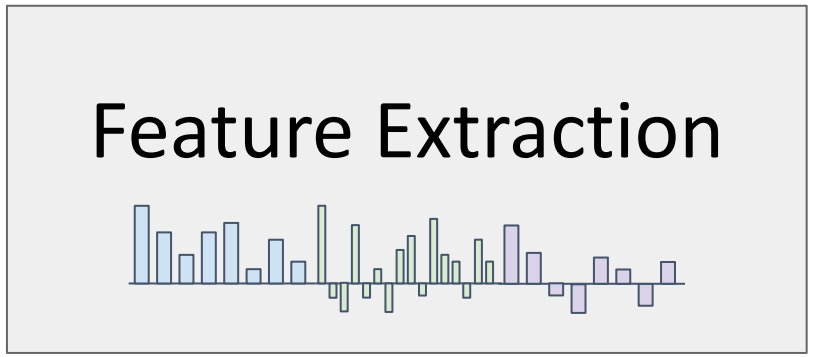


f

20 numbers giving scores for classes

training

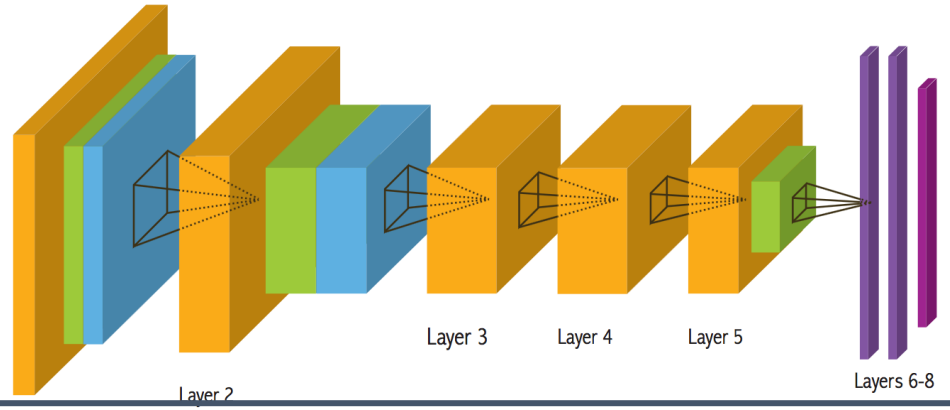
Image Features vs Neural Networks



f

20 numbers giving scores for classes

training



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

20 numbers giving scores for classes

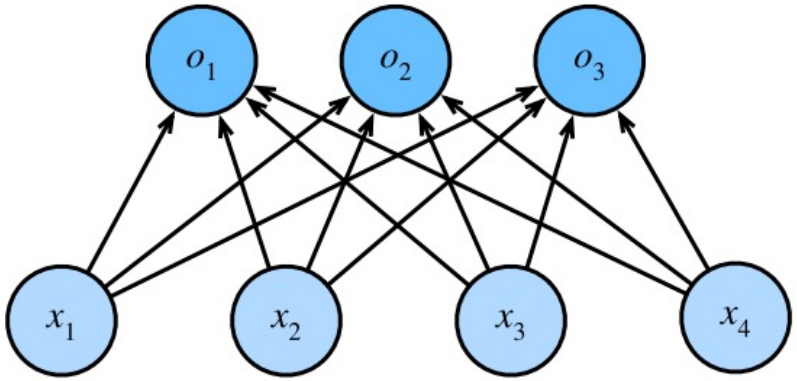
training

Neural Networks

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

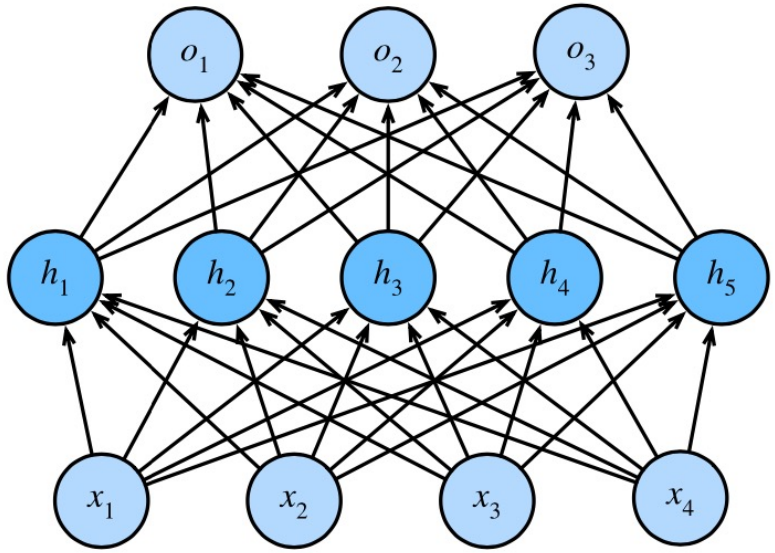
Before: Linear Classifier: $f(x) = Wx + b$

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



Neural Networks

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

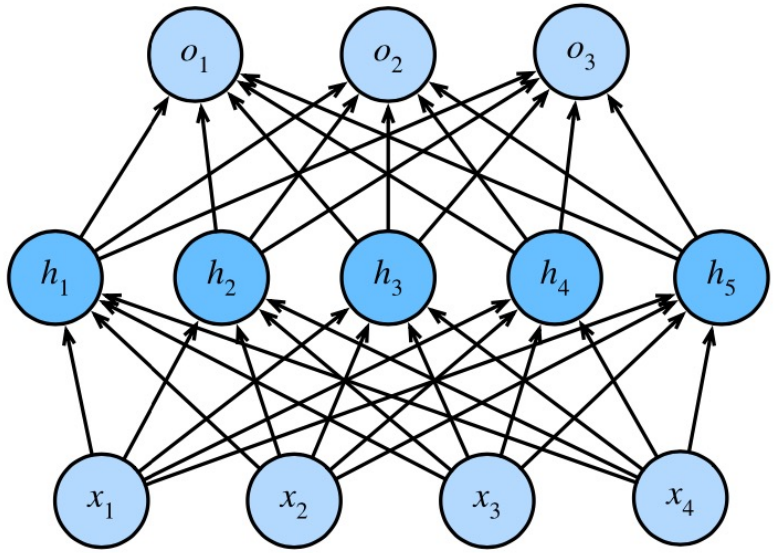


Before: Linear Classifier: $f(x) = Wx + b$
Learnable parameters: $W \in \mathbb{R}^{D \times C}, b \in \mathbb{R}^C$

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

Neural Networks

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

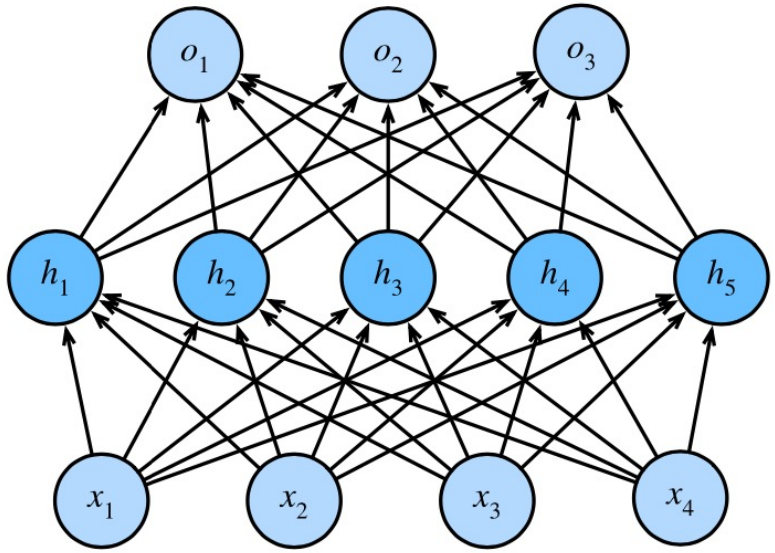


Before: Linear Classifier: $f(x) = Wx + b$
Learnable parameters: $W \in \mathbb{R}^{D \times C}, b \in \mathbb{R}^C$

Now: Two-Layer Neural Network: $f(x) = W_2 \max(0, W_1 x + b_1) + b2$
Learnable parameters: $W_1 \in \mathbb{R}^{H \times D}, b_1 \in \mathbb{R}^H, W_2 \in \mathbb{R}^{C \times H}, b_2 \in \mathbb{R}^C$

Neural Networks

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



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

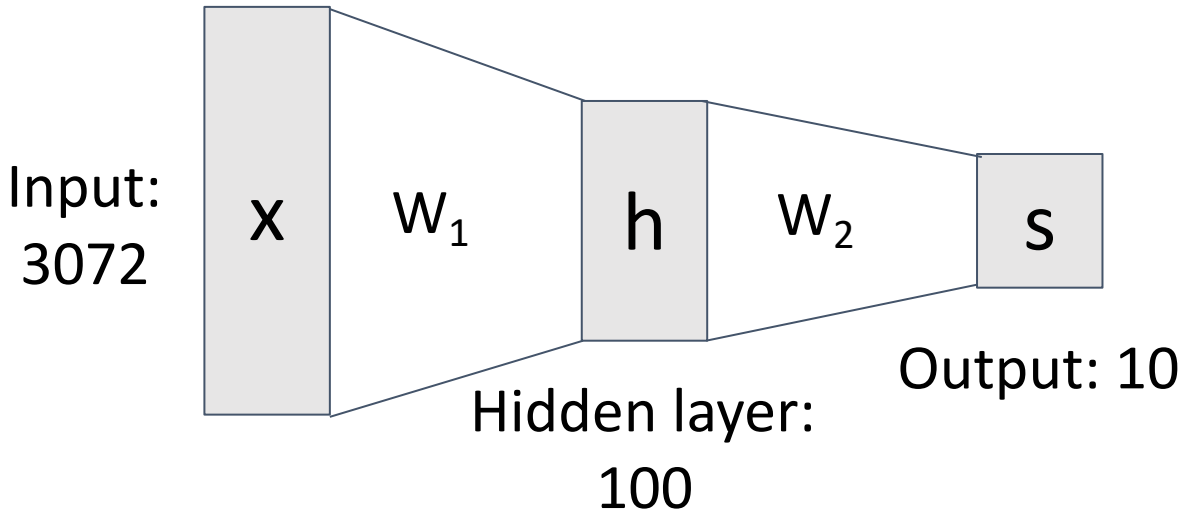
Now: Two-Layer Neural Network: $f(x) = W_2 \max(0, W_1 x + b_1) + b2$
Learnable parameters: $W_1 \in \mathbb{R}^{H \times D}, b_1 \in \mathbb{R}^H, W_2 \in \mathbb{R}^{C \times H}, b_2 \in \mathbb{R}^C$

Or Three-Layer Neural Network:
$f(x) = W_3 \max(0, W_2 \max(0, W_1 x + b_1) + b_2) + b_3$

Neural Networks

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

Now: 2-layer Neural Network $f(x) = W_2 \max(0, W_1 x + b_1) + b_2$



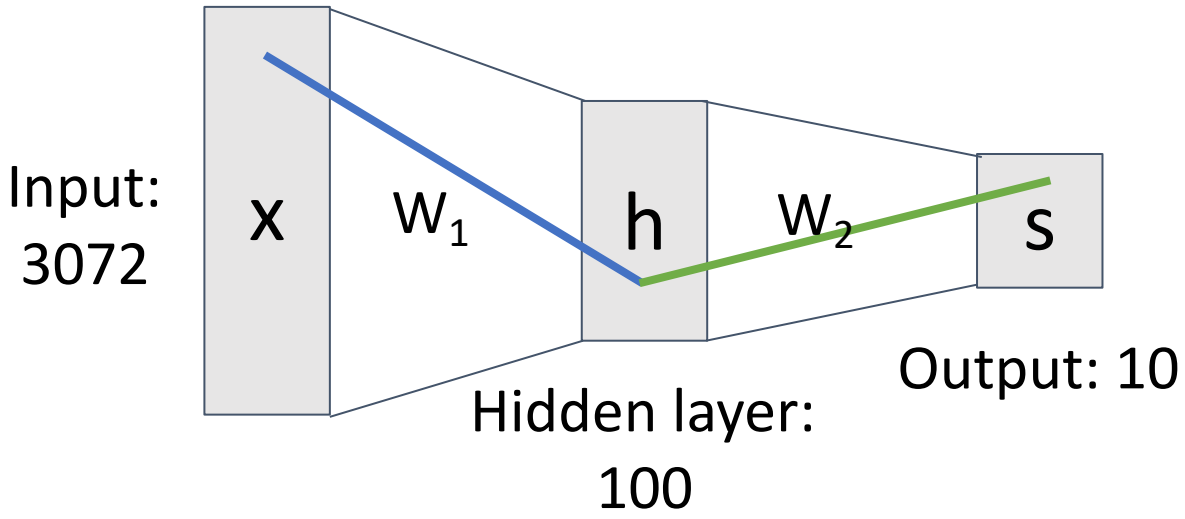
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

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

Now: 2-layer Neural Network $f(x) = W_2 \max(0, W_1 x + b_1) + b_2$

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



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

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

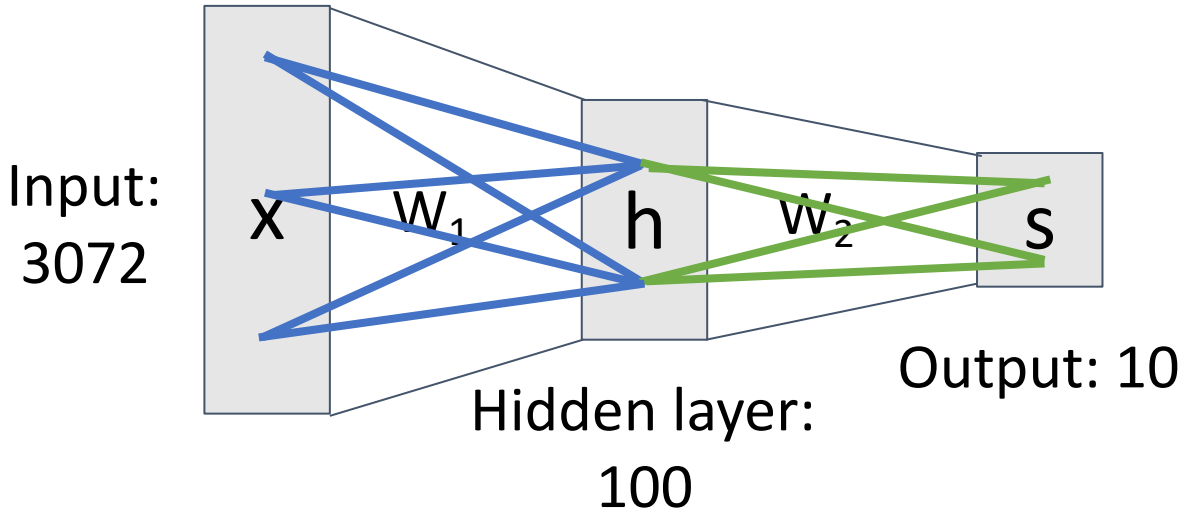
Neural Networks

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

Now: 2-layer Neural Network $f(x) = W_2 \max(0, W_1 x + b_1) + b_2$

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

All elements of x affect all elements of h



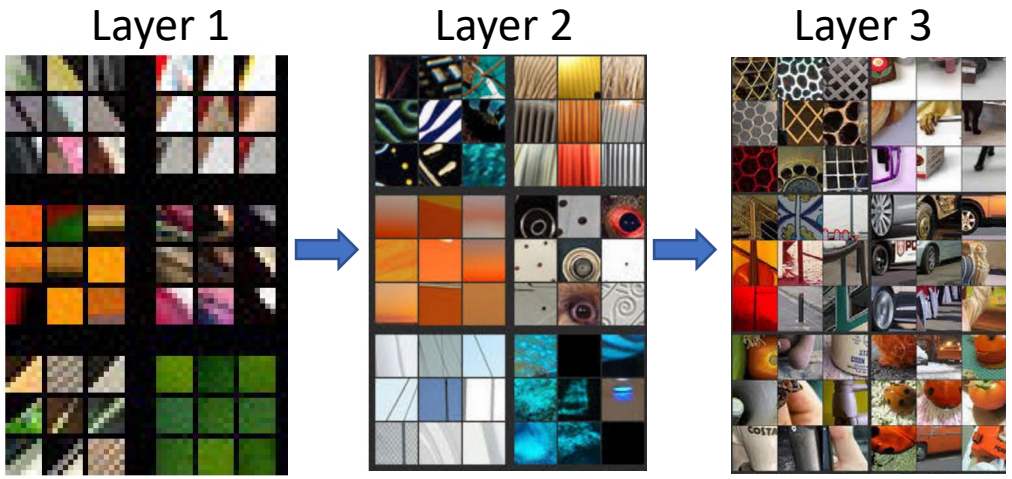
Element (i, j) of W_2 gives the effect on s_i from h_j

All elements of h affect all elements of s

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

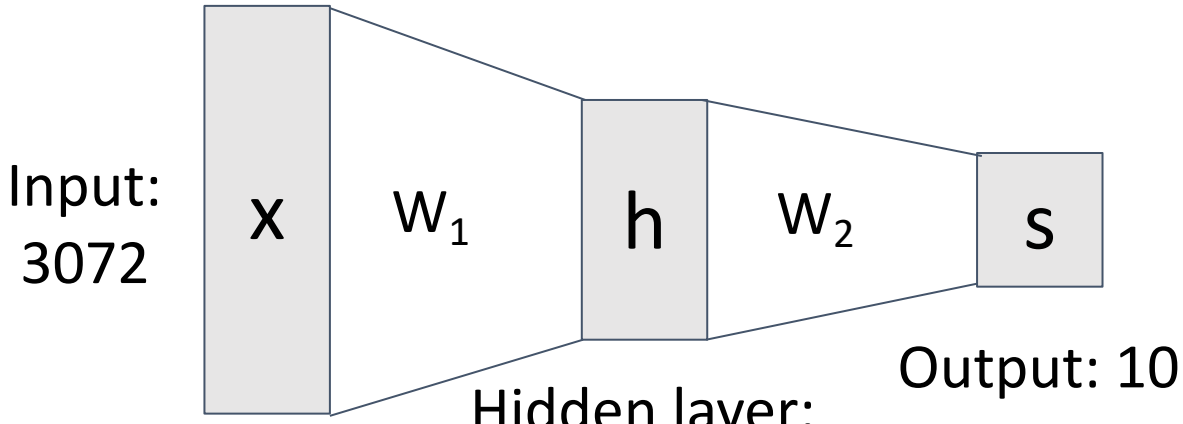
Neural Networks

Neural net: Later layer compose features from previous layer's activation



Hierarchical compositional features

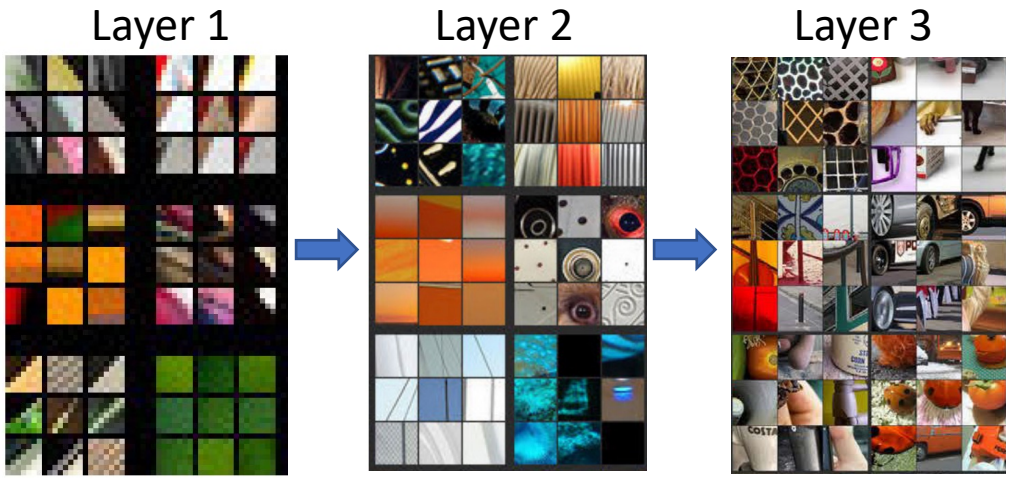
2-layer Neural Network



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

Neural Networks

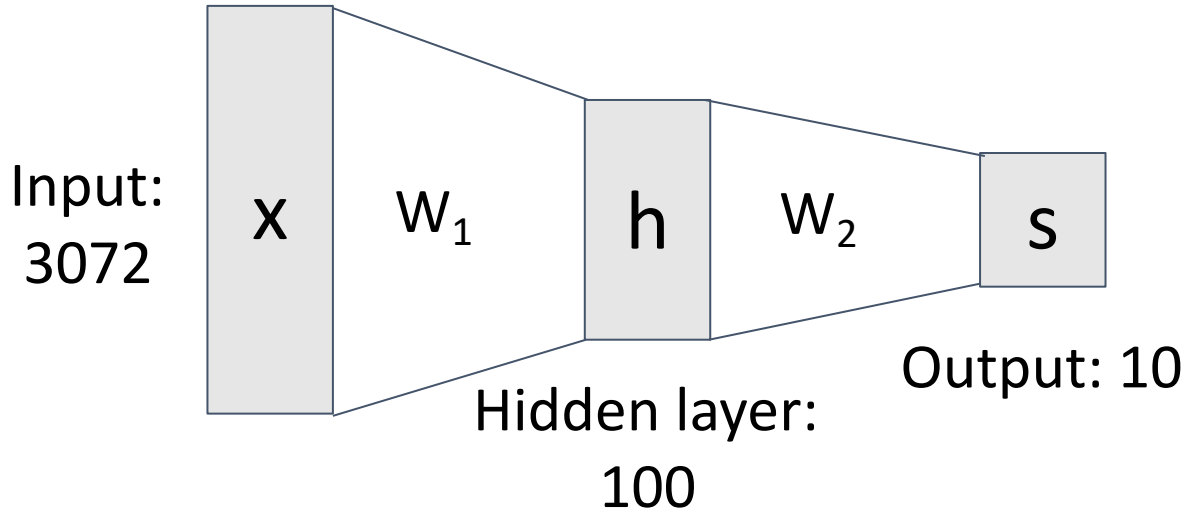
Neural net: Later layer compose features from previous layer's activation



Hierarchical compositional features

Understanding the learned representation of neural networks is an active research topic

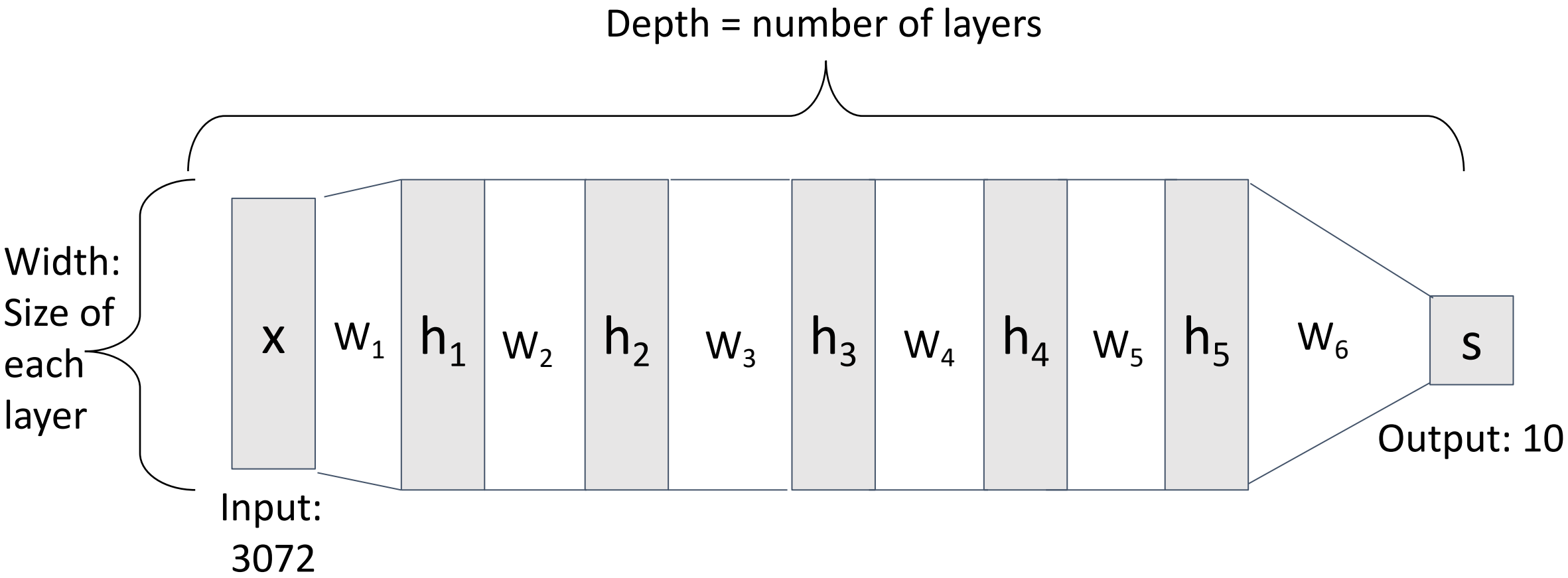
2-layer Neural Network



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

Zeiler and Fergus. ECCV'15

Deep Neural Networks



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

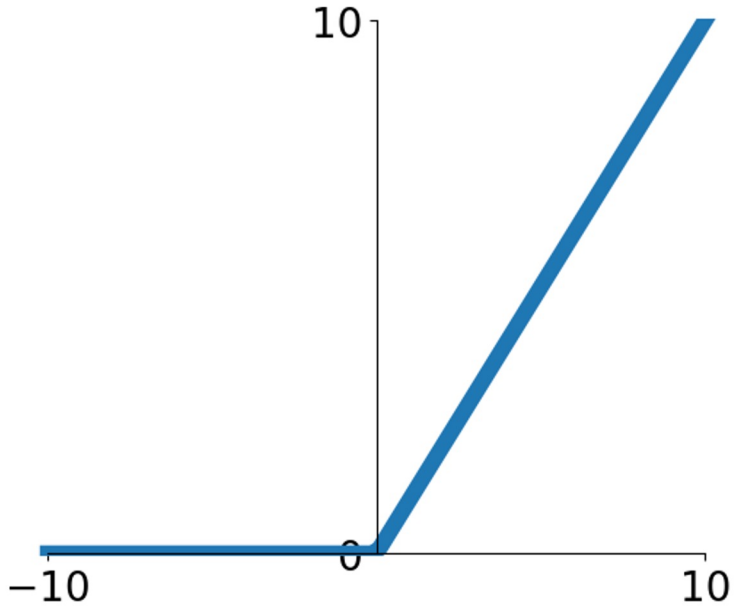
Activation Functions

2-layer Neural Network

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

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

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

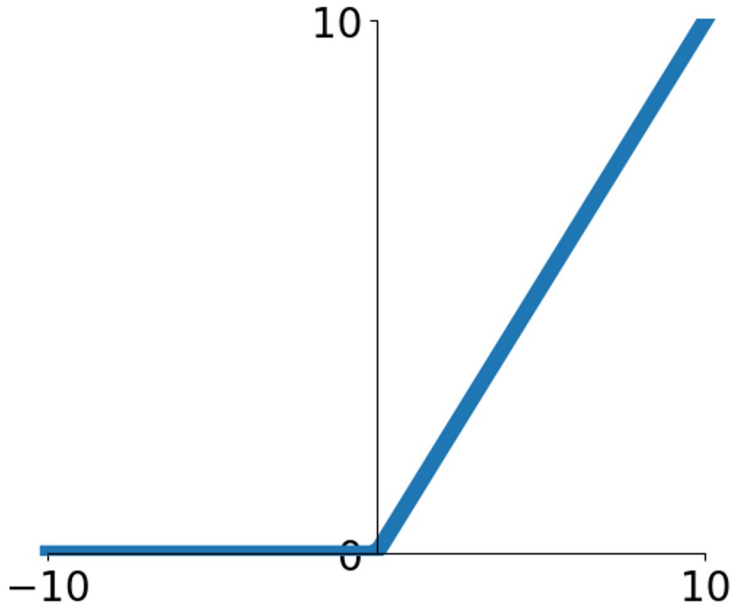


Activation Functions

2-layer Neural Network

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

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



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

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

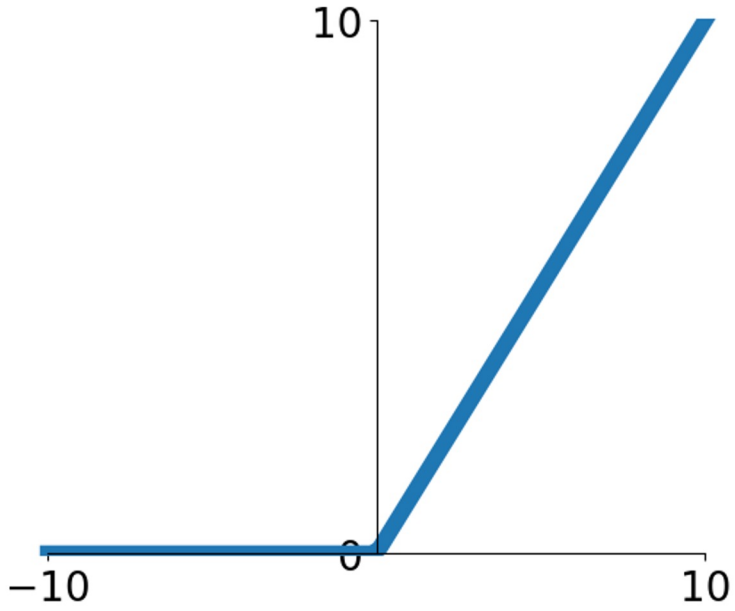
$$f(x) = W_2(W_1 x + b_1) + b_2$$

Activation Functions

2-layer Neural Network

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

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



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

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

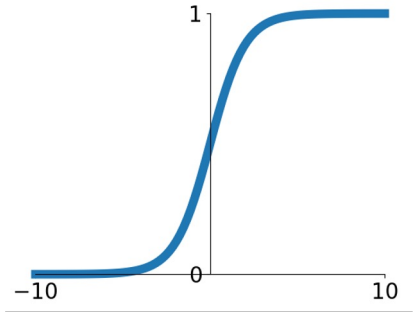
$$f(x) = W_2(W_1 x + b_1) + b_2$$
$$= (W_1 W_2)x + (W_2 b_1 + b_2)$$

A: We end up with a linear classifier!

Activation Functions

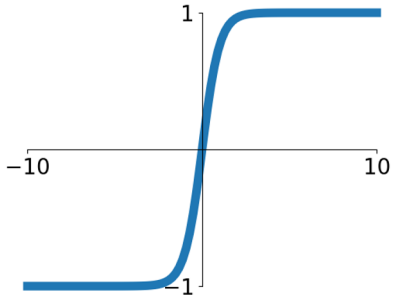
Sigmoid

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



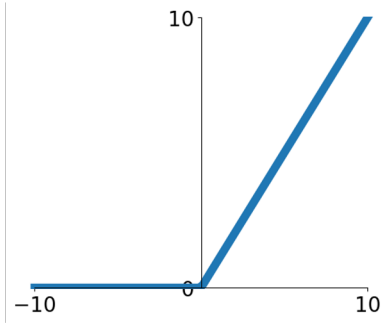
tanh

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



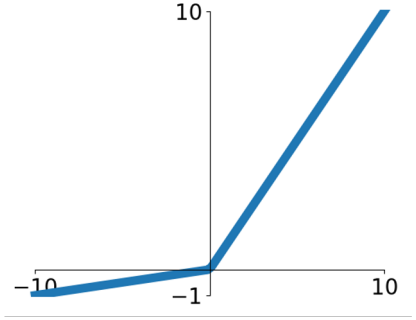
ReLU

$$\max(0, x)$$



Leaky ReLU

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

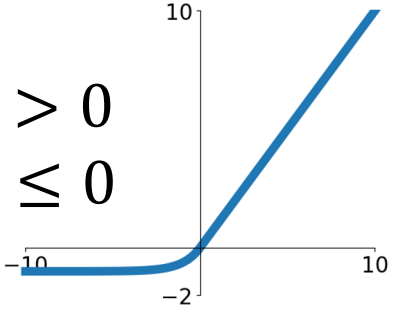


Softplus

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

ELU

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

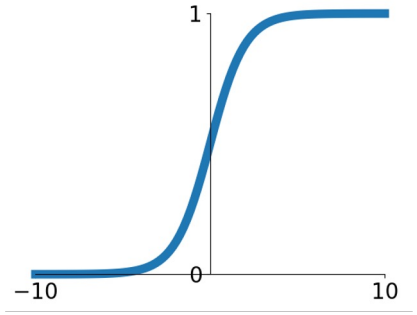


Activation Functions

ReLU is a good default choice for most problems

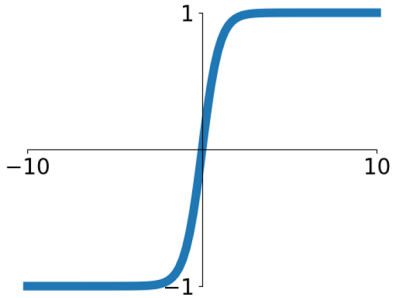
Sigmoid

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



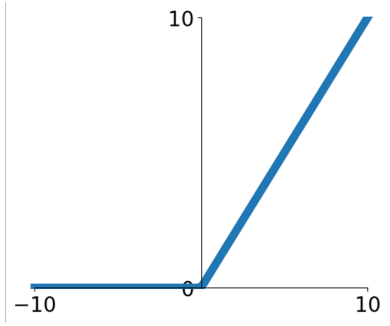
tanh

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



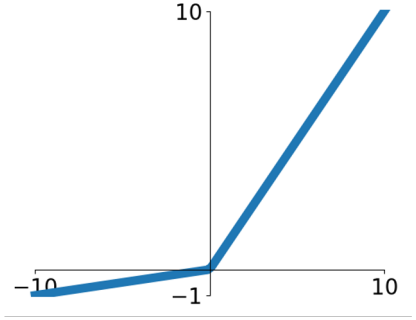
ReLU

$$\max(0, x)$$



Leaky ReLU

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

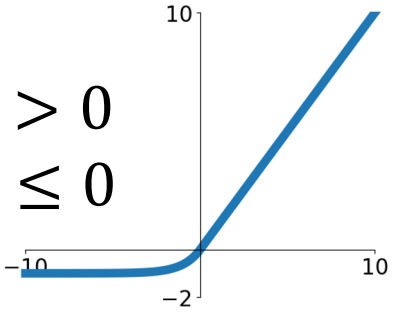


Softplus

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

ELU

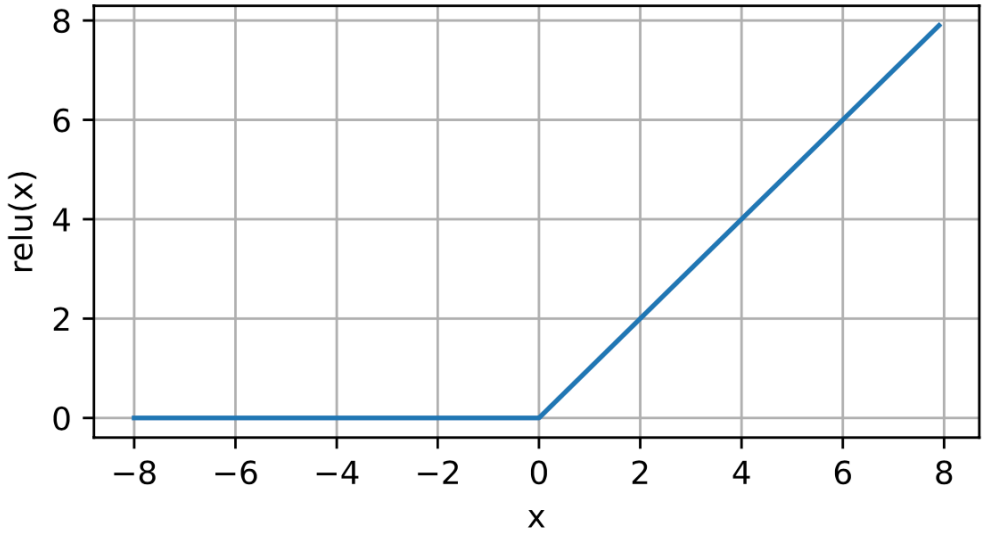
$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \leq 0 \end{cases}$$



Why ReLU

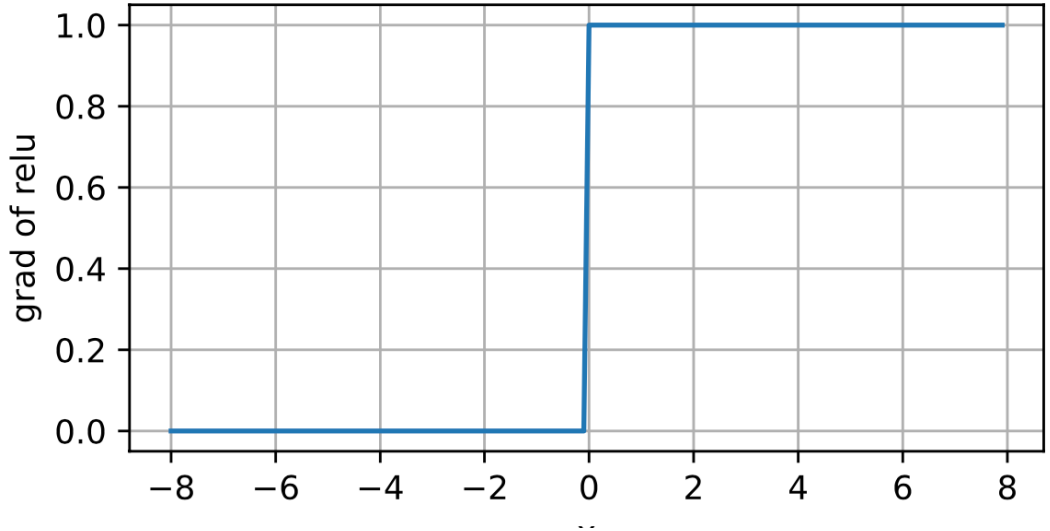
Forward pass

```
x = torch.arange(-8.0, 8.0, 0.1, requires_grad=True)
y = torch.relu(x)
d2l.plot(x.detach(), y.detach(), 'x', 'relu(x)', figsize=(5, 2.5))
```



Backward pass

```
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of relu', figsize=(5, 2
```



its derivatives are particularly well behaved: either they vanish or they just let the argument through. This makes optimization better behaved and it mitigated the well-documented problem of vanishing gradients

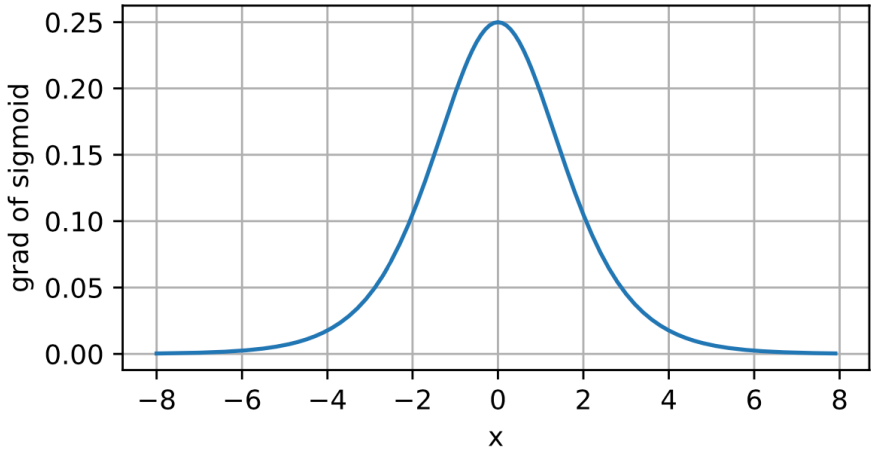
Issues with sigmoid and tanh

sigmoid is also called squashing function, turn (-inf, inf) into (0,1)

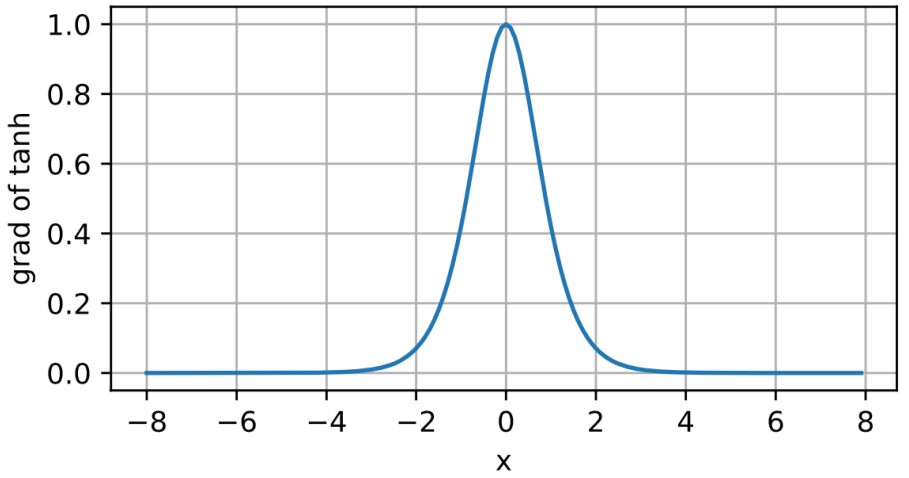
$$\frac{d}{dx}\operatorname{sigmoid}(x) = \frac{\exp(-x)}{(1+\exp(-x))^2} = \operatorname{sigmoid}(x)\,(1-\operatorname{sigmoid}(x))\,.$$

$$\frac{d}{dx}\tanh(x) = 1 - \tanh^2(x).$$

```
y = torch.sigmoid(x)
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of sigmoid', figsize=(5, 2.5))
```

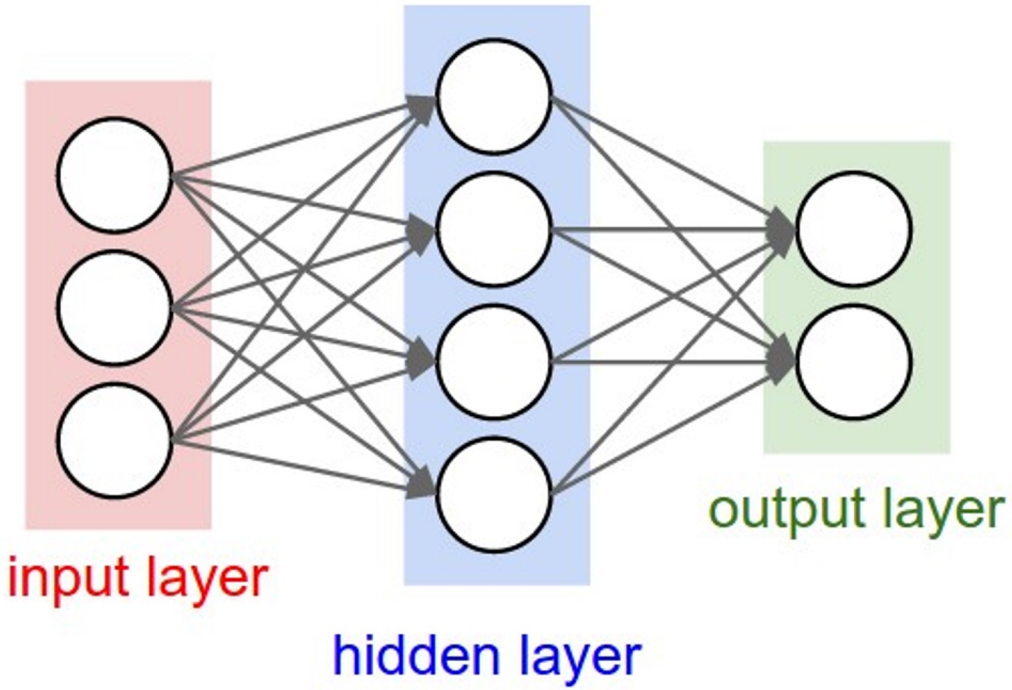



```
y = torch.tanh(x)
y.backward(torch.ones_like(x), retain_graph=True)
d2l.plot(x.detach(), x.grad, 'x', 'grad of tanh', figsize=(5, 2.5))
```



As the input diverges from 0 in either direction, the derivative approaches 0, vanishing gradient happens.

Neural Net in <20 lines!

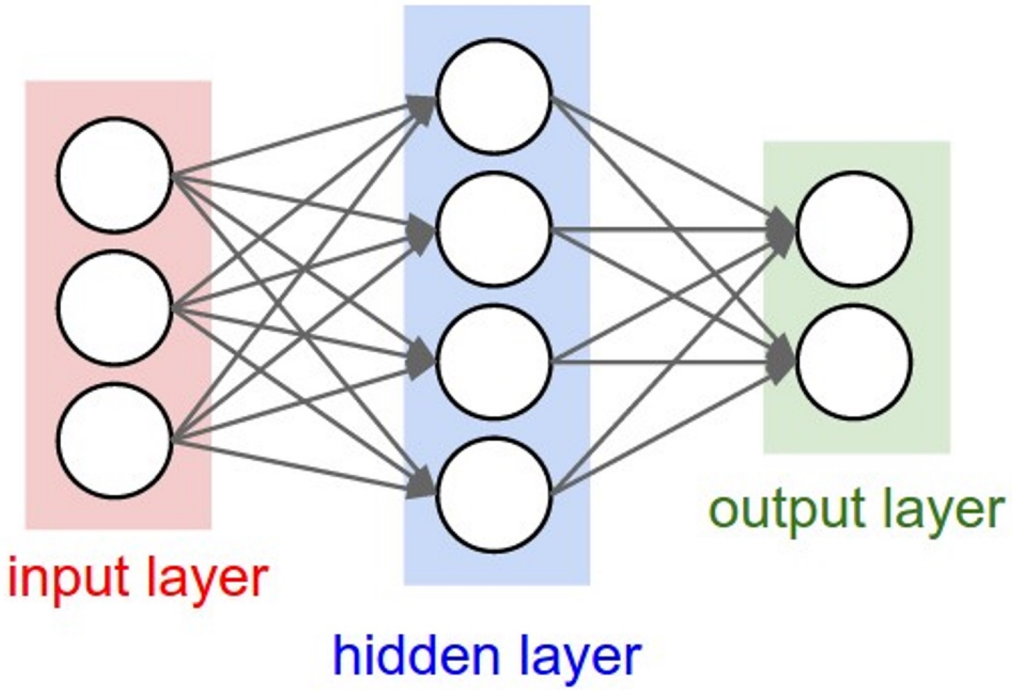



```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = nn.Parameter(torch.randn(
    num_inputs, num_hiddens, requires_grad=True) * 0.01)
b1 = nn.Parameter(torch.zeros(num_hiddens, requires_grad=True))
W2 = nn.Parameter(torch.randn(
    num_hiddens, num_outputs, requires_grad=True) * 0.01)
b2 = nn.Parameter(torch.zeros(num_outputs, requires_grad=True))

params = [W1, b1, W2, b2]
```

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(X@W1 + b1)  # Here '@' stands for matrix multiplication
    return (H@W2 + b2)
```

Neural Net in <20 lines!




```
num_inputs, num_outputs, num_hiddens = 784, 10, 256

W1 = nn.Parameter(torch.randn(
    num_inputs, num_hiddens, requires_grad=True) * 0.01)
b1 = nn.Parameter(torch.zeros(num_hiddens, requires_grad=True))
W2 = nn.Parameter(torch.randn(
    num_hiddens, num_outputs, requires_grad=True) * 0.01)
b2 = nn.Parameter(torch.zeros(num_outputs, requires_grad=True))

params = [W1, b1, W2, b2]
```

```
def relu(X):
    a = torch.zeros_like(X)
    return torch.max(X, a)
```

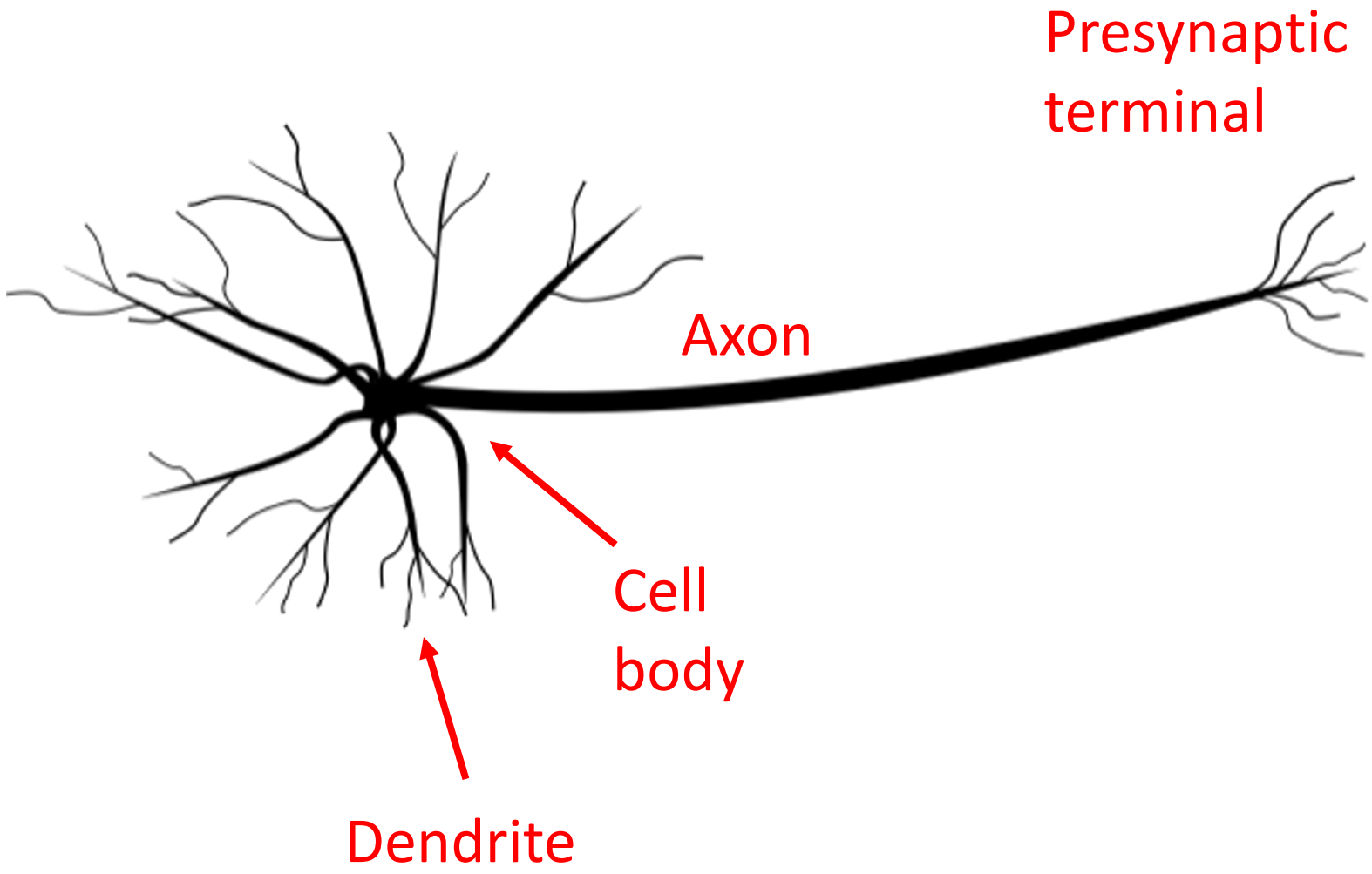
```
def net(X):
    X = X.reshape((-1, num_inputs))
    H = relu(X@W1 + b1)  # Here '@' stands for matrix multiplication
    return (H@W2 + b2)
```

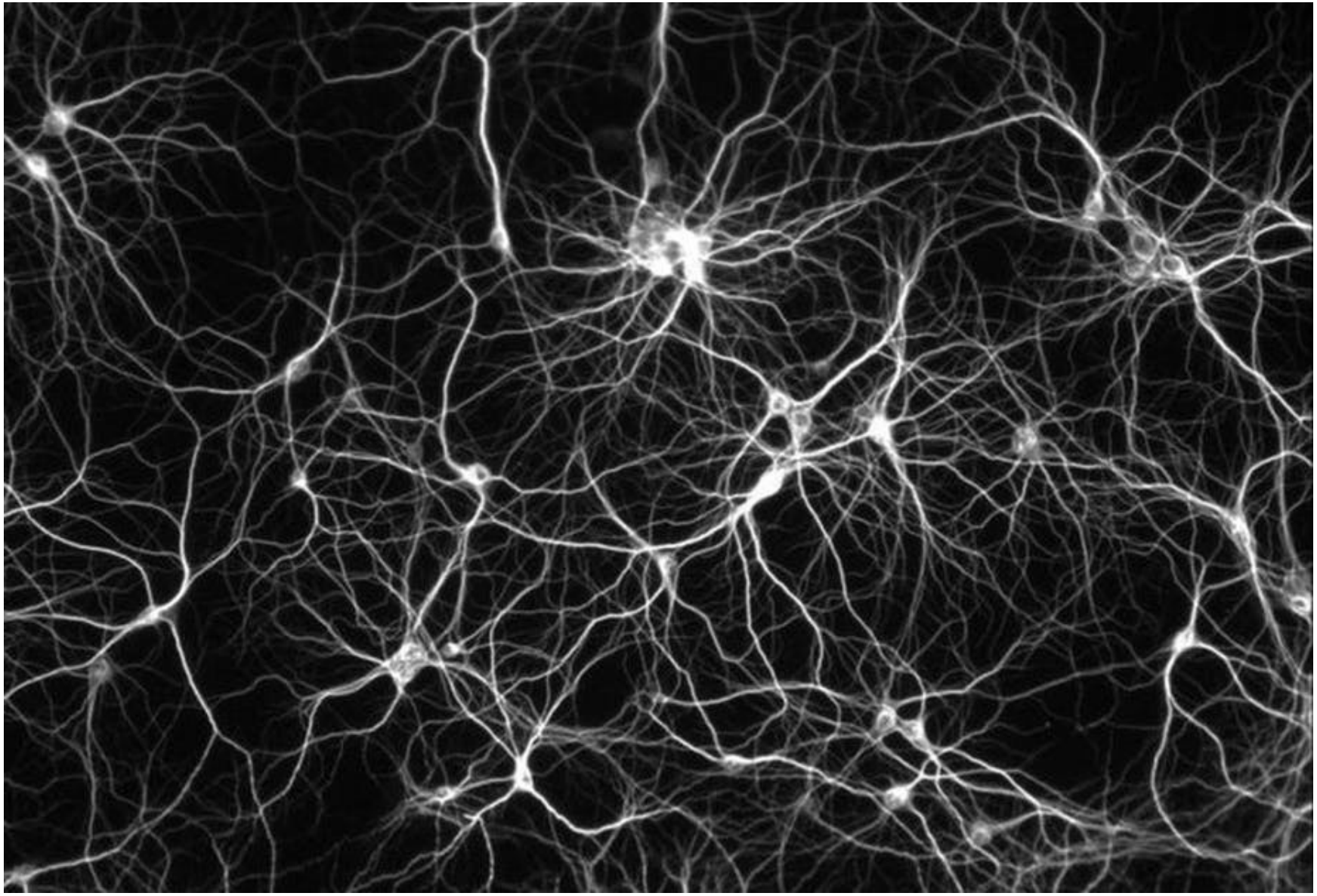
```
loss = nn.CrossEntropyLoss()
```

```
num_epochs, lr = 10, 0.1
updater = torch.optim.SGD(params, lr=lr)
d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, updater)
```

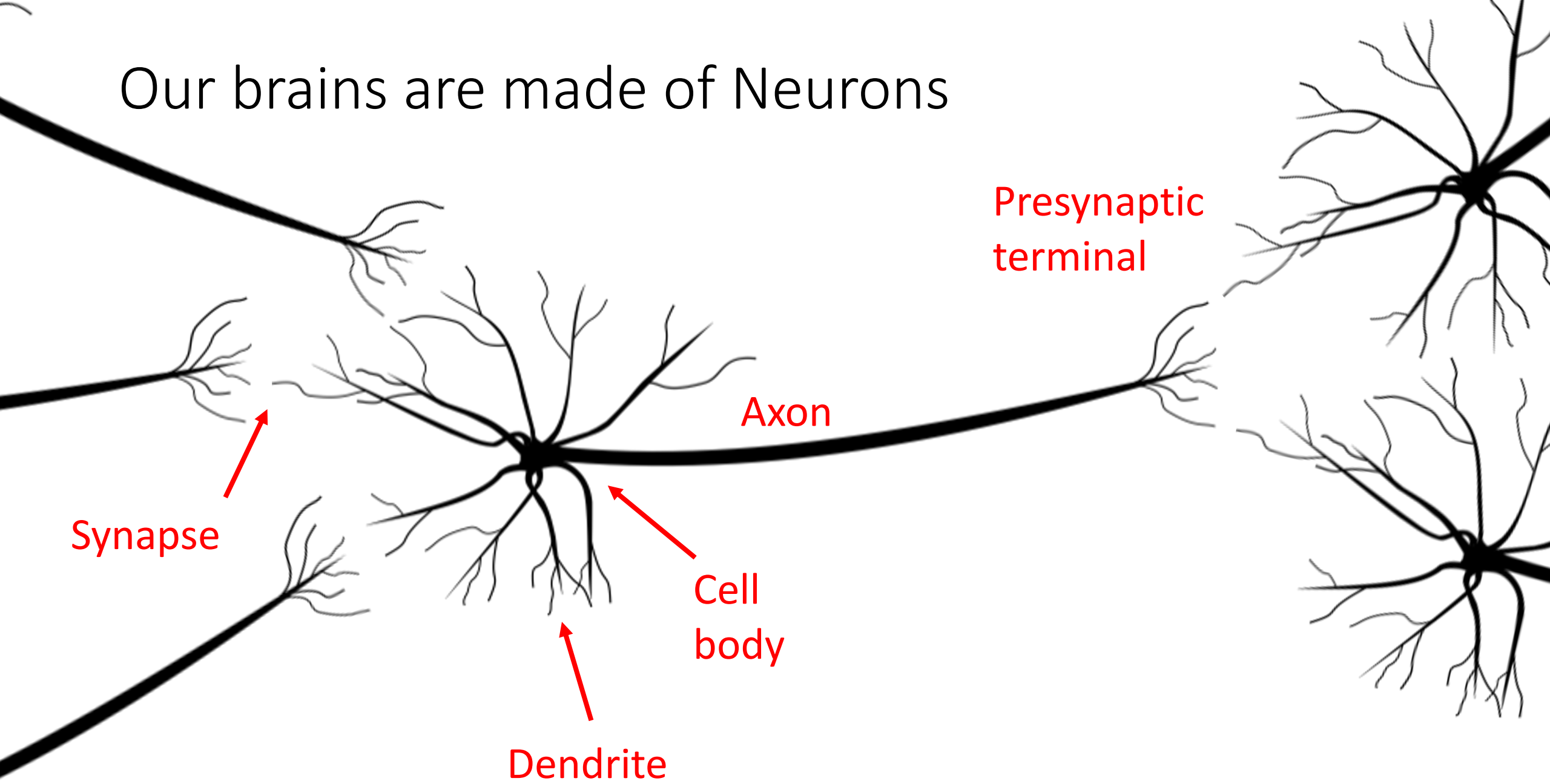
Analogy between Artificial Neural Network (ANN) and Human Brain

Our brains are made of Neurons

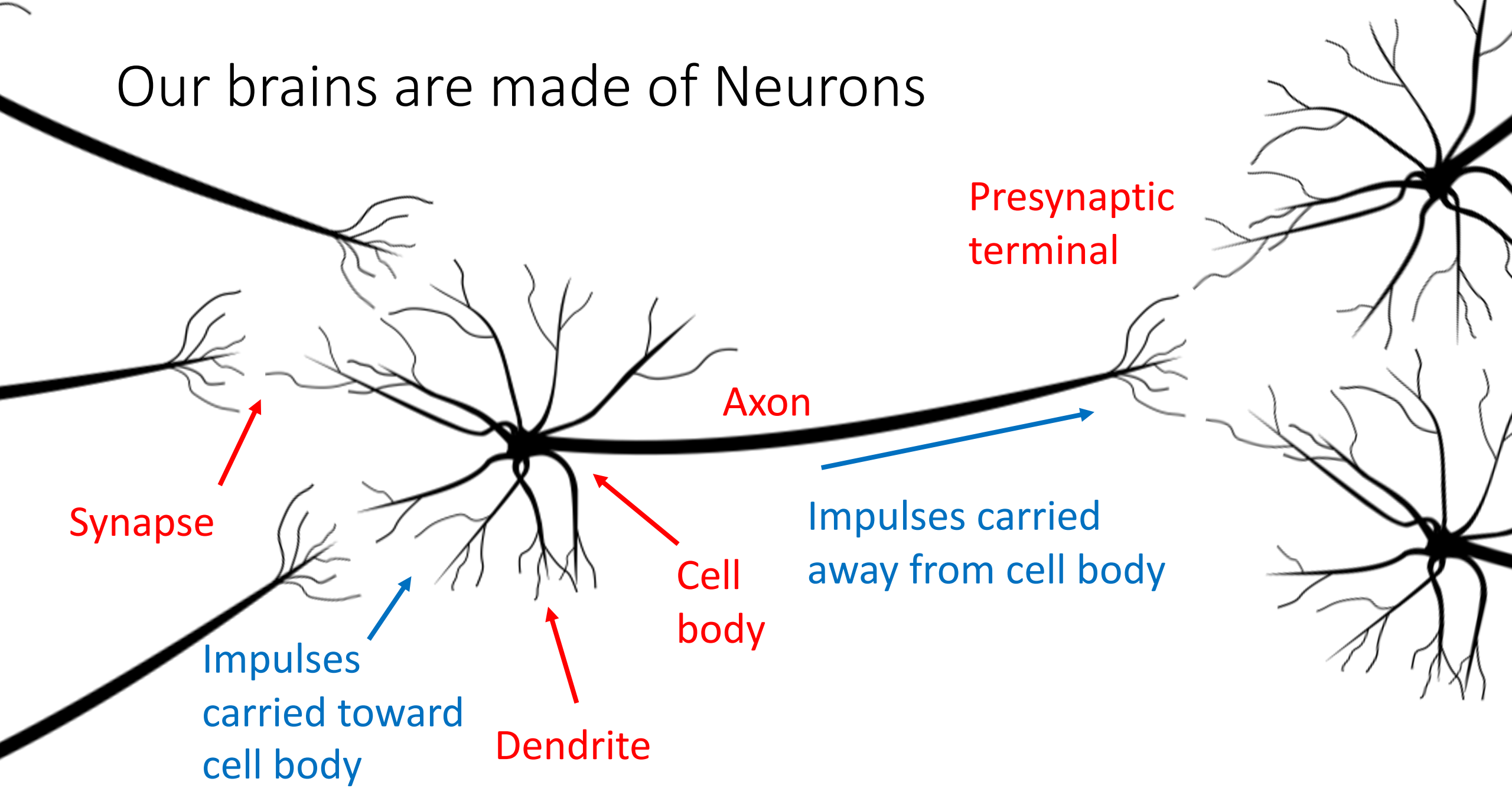




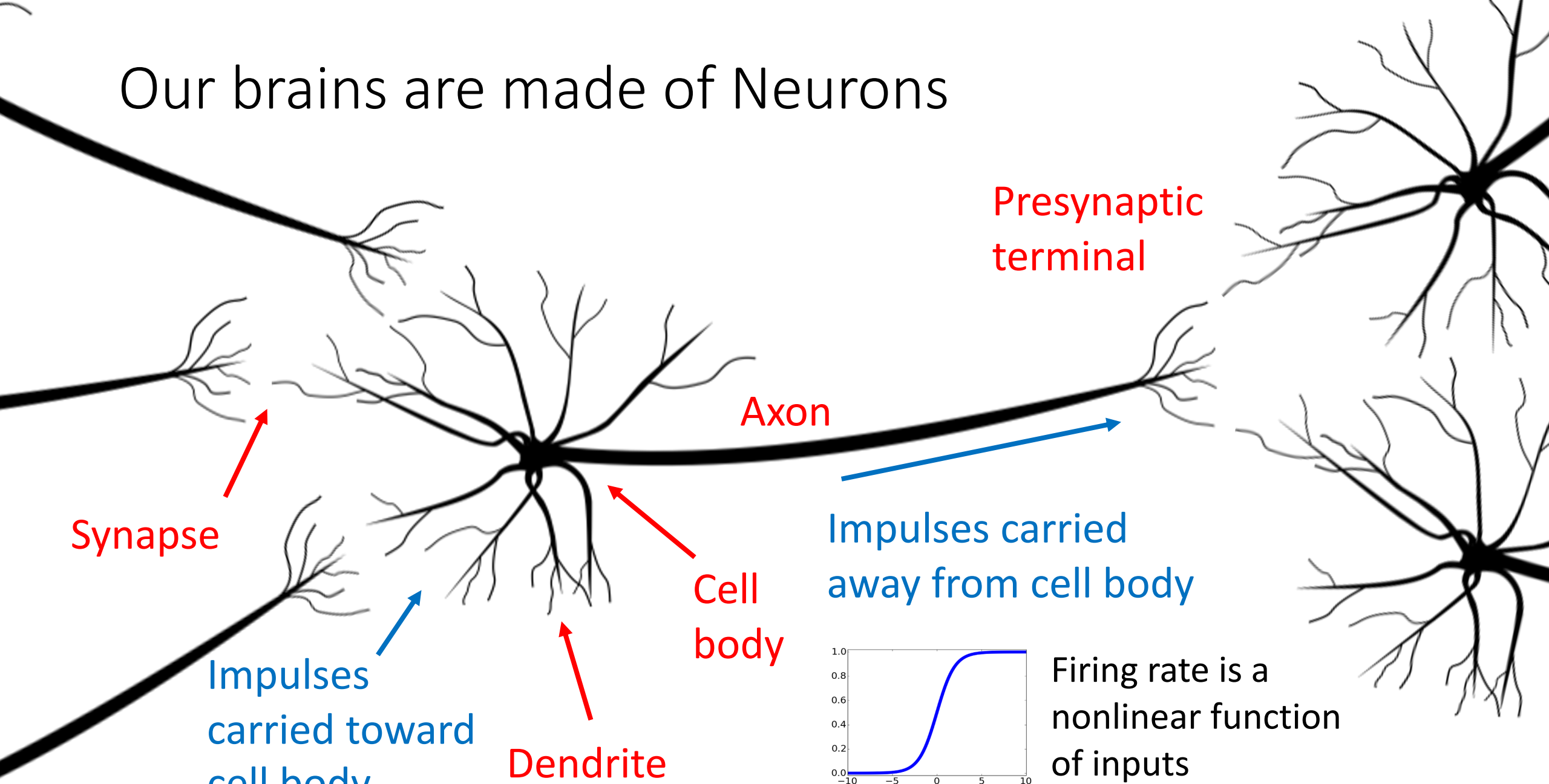
Our brains are made of Neurons



Our brains are made of Neurons

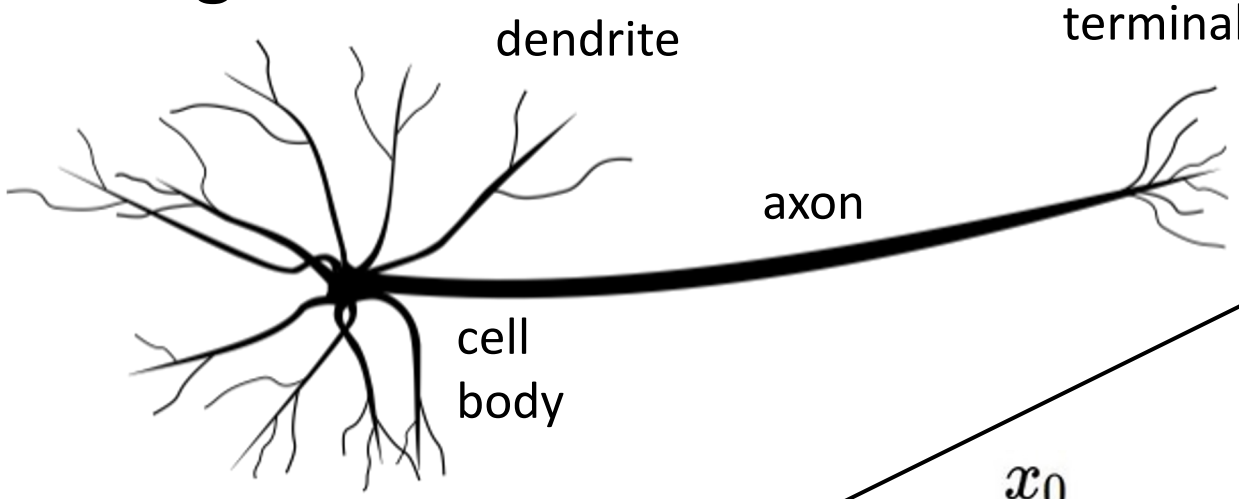


Our brains are made of Neurons

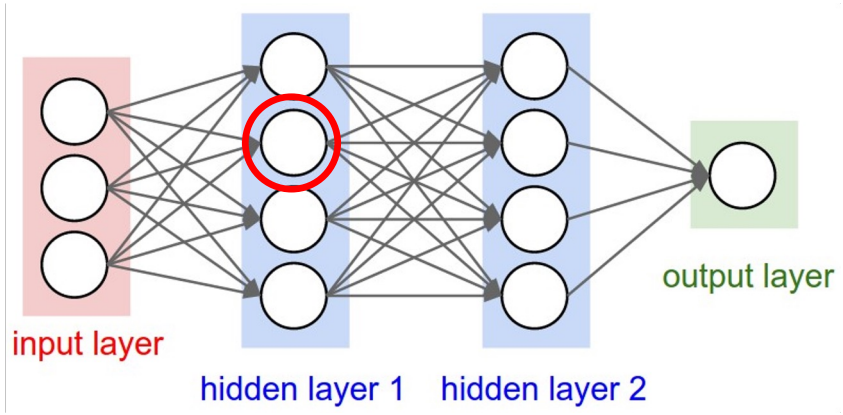


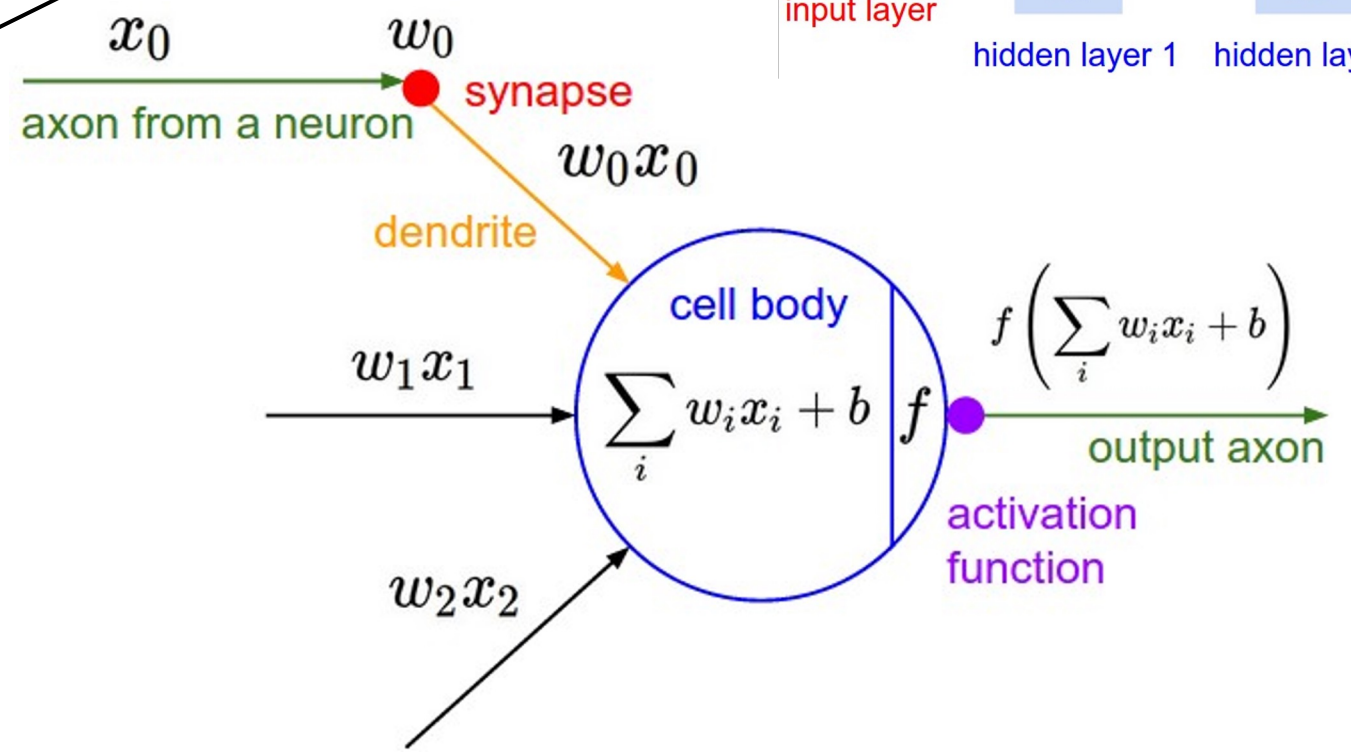
Firing rate is a nonlinear function of inputs

Biological Neuron

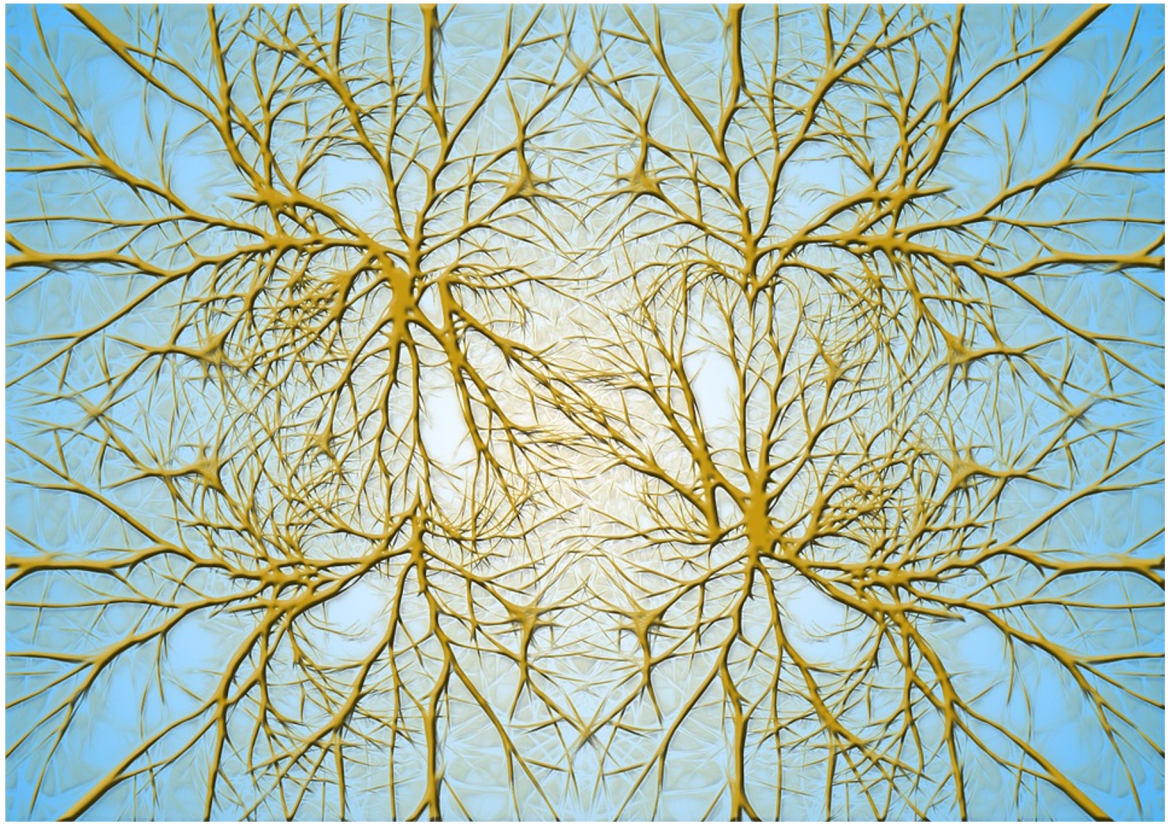


Artificial Neuron

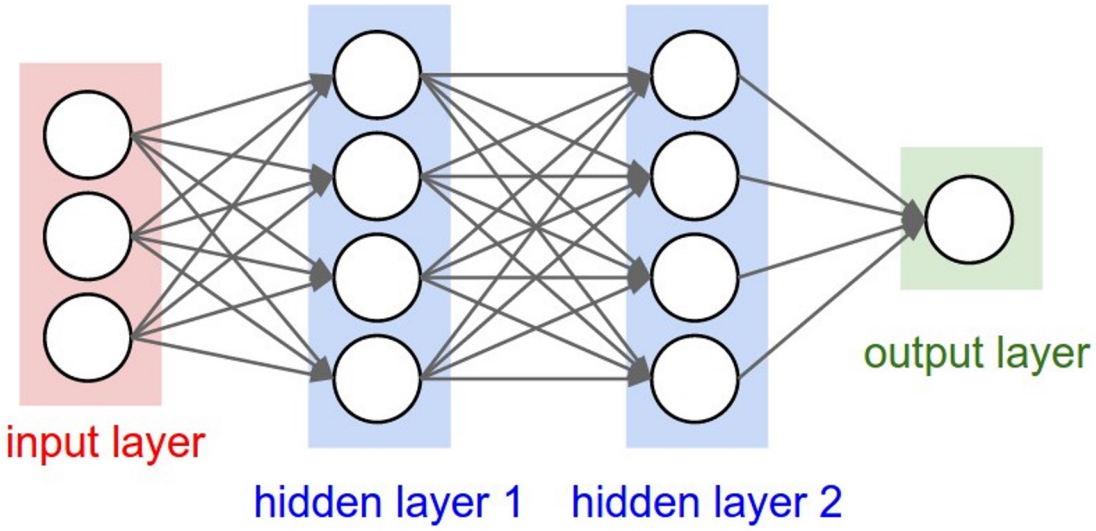




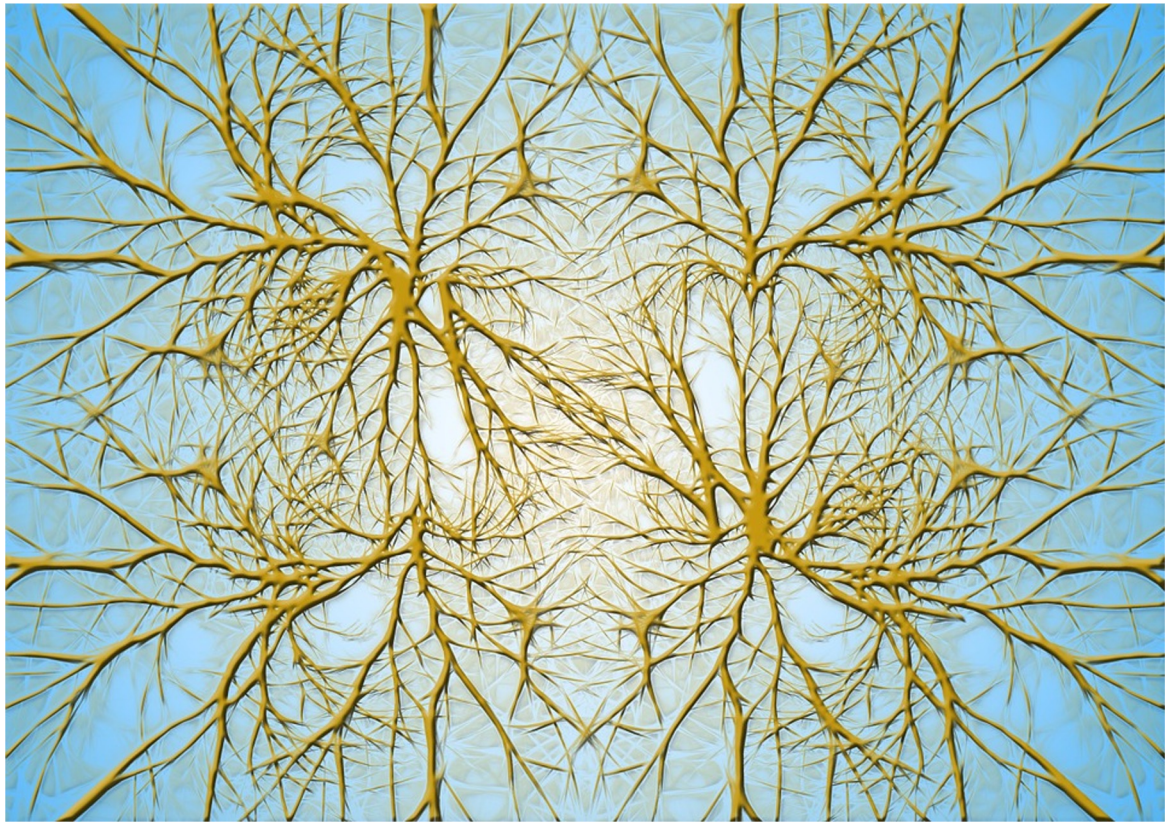
Biological Neurons:
Complex connectivity patterns



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

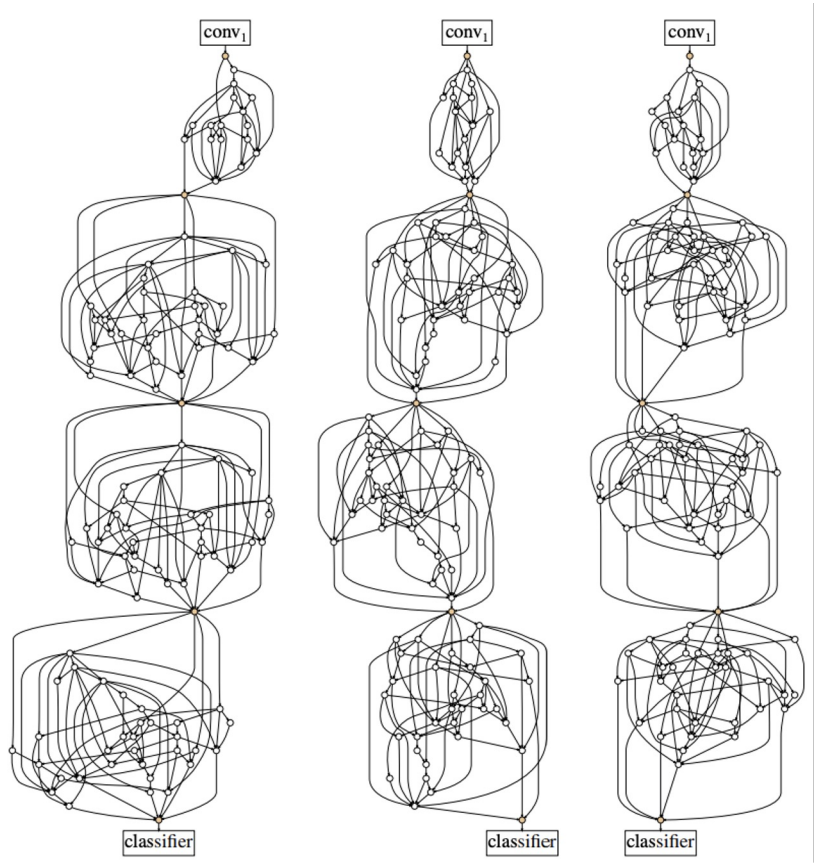


Biological Neurons: Complex connectivity patterns



This image is CC0 Public Domain

But neural networks with random connections can work too!

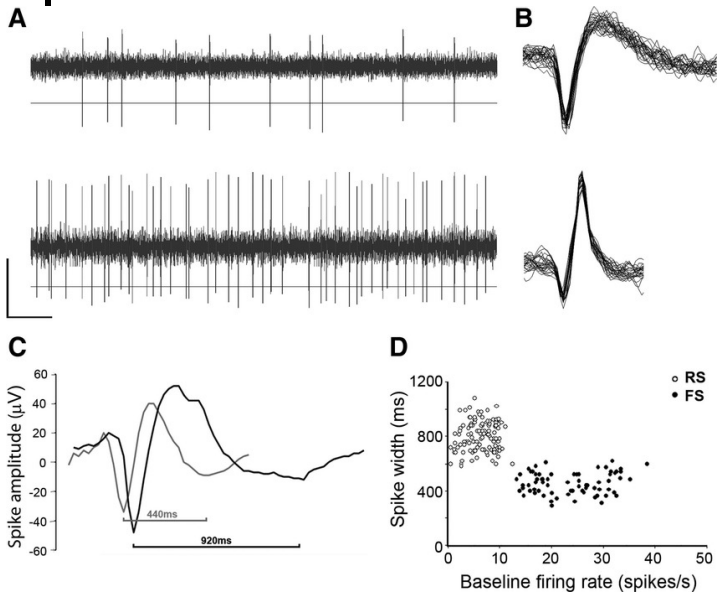


Xie et al, "Exploring Randomly Wired Neural Networks for Image Recognition", ICCV 2019

Are neural networks biologically inspired? Weakly

Biological Neurons:

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



[Dendritic Computation. London and Hausser]

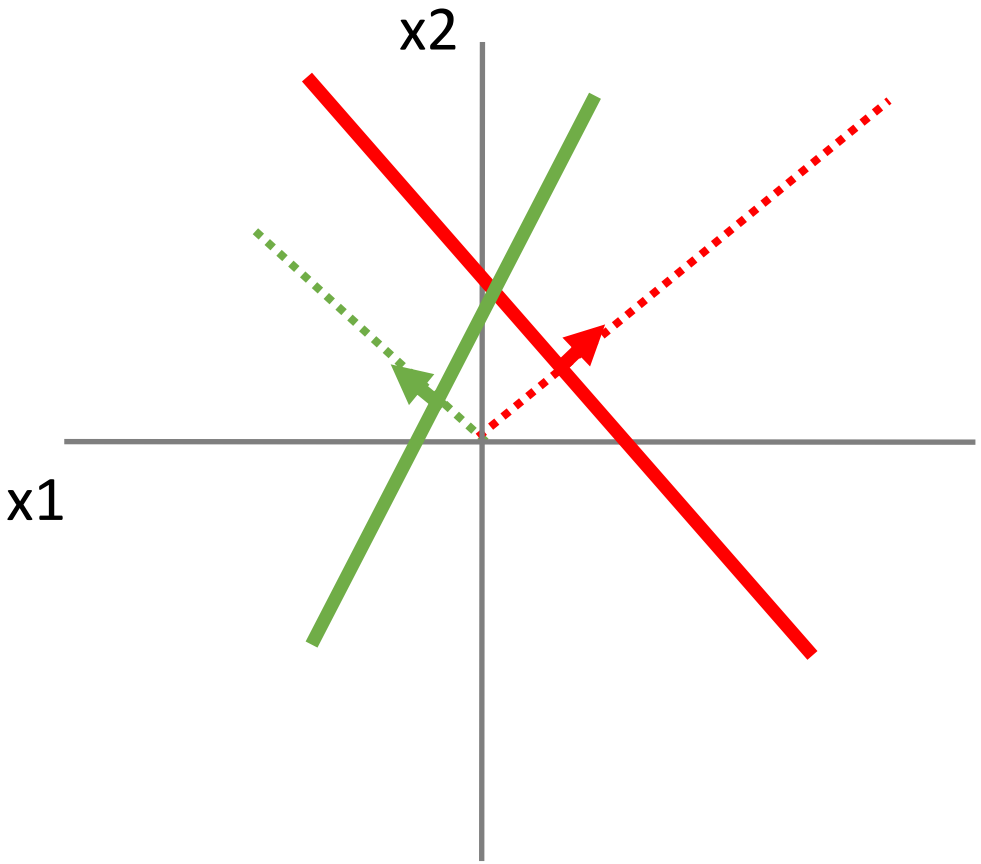
Differential Involvement of Excitatory and Inhibitory Neurons of Cat Motor Cortex in Coincident Spike Activity Related to Behavioral Context

Intuition on why 2-layer NN + ReLU works

Space Warping

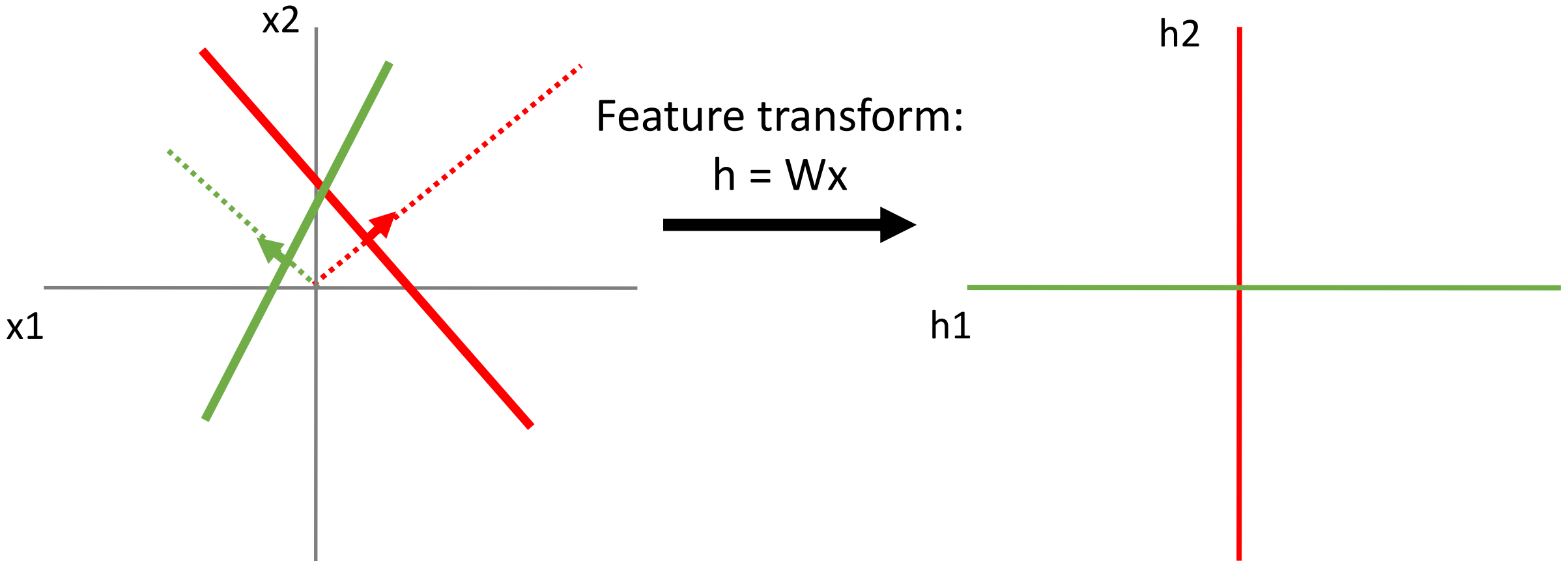
Space Warping

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



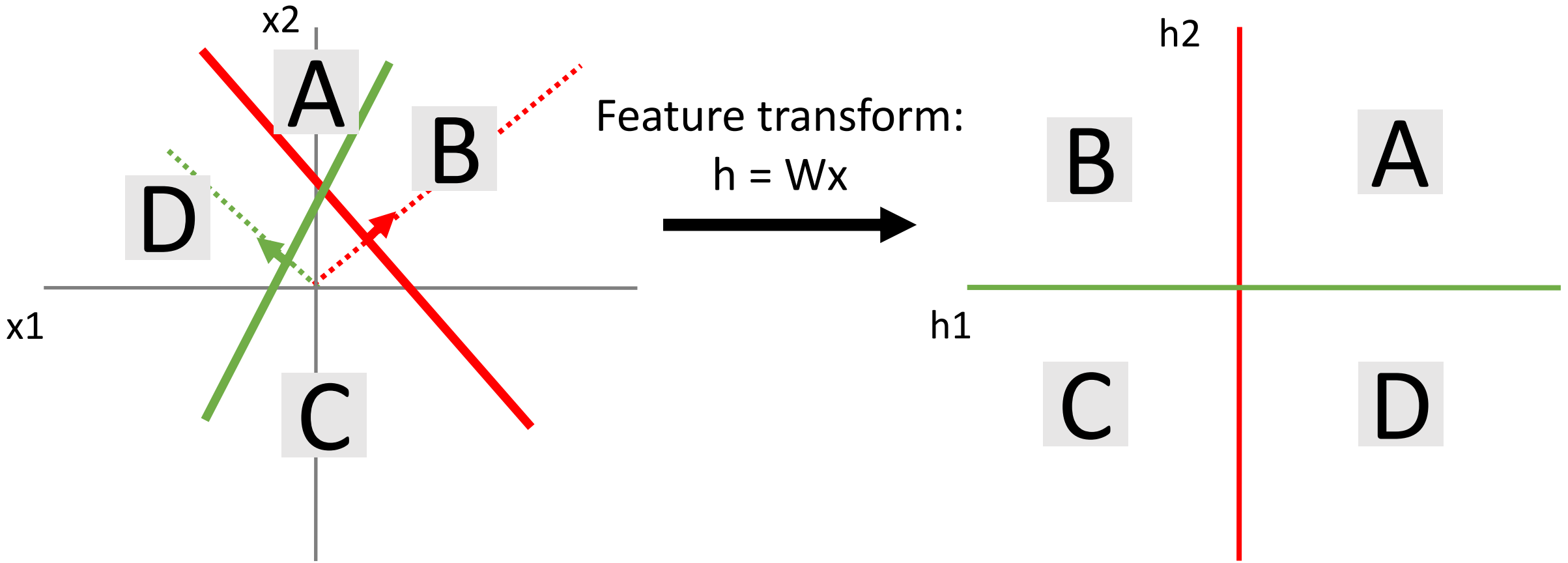
Space Warping

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



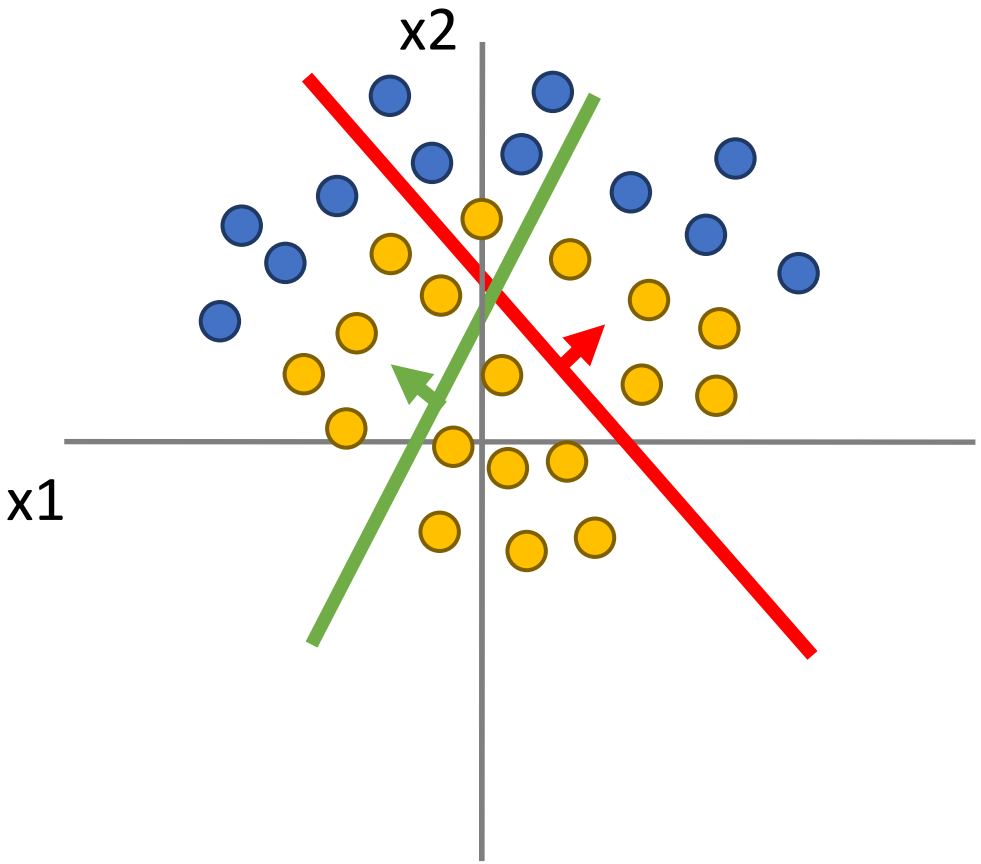
Space Warping

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



Space Warping

Points not linearly separable in original space

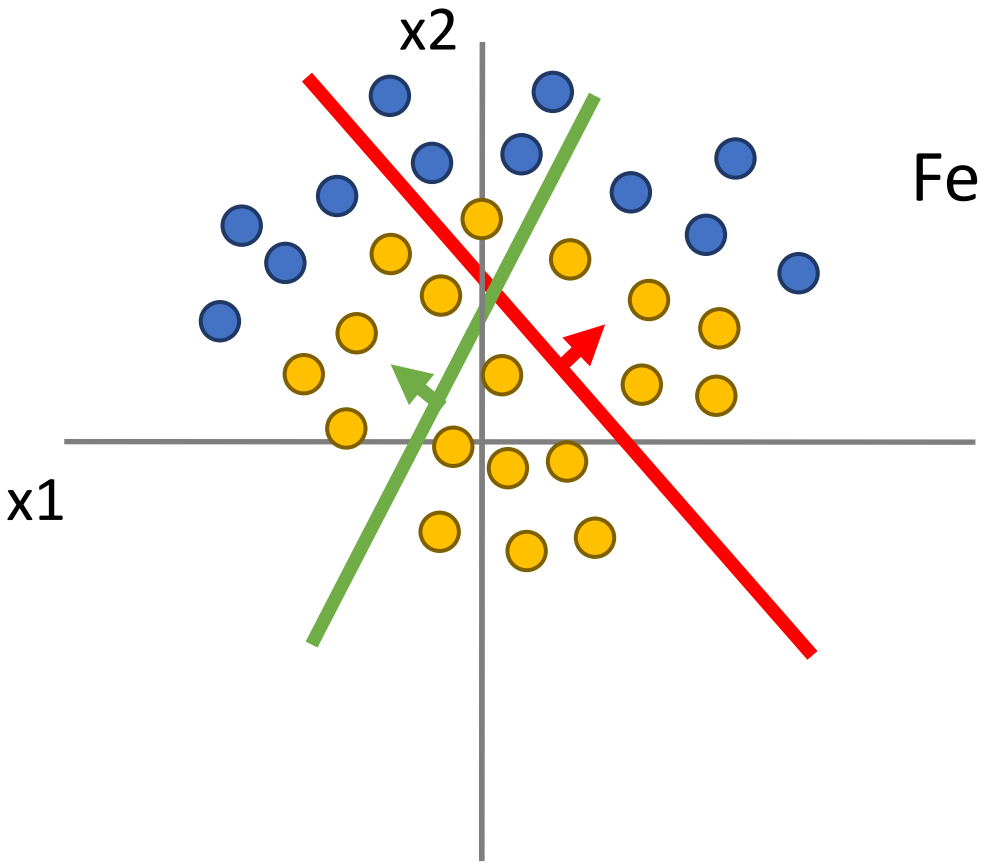


Consider a linear transform: h = Wx
Where x, h are both 2-dimensional

Space Warping

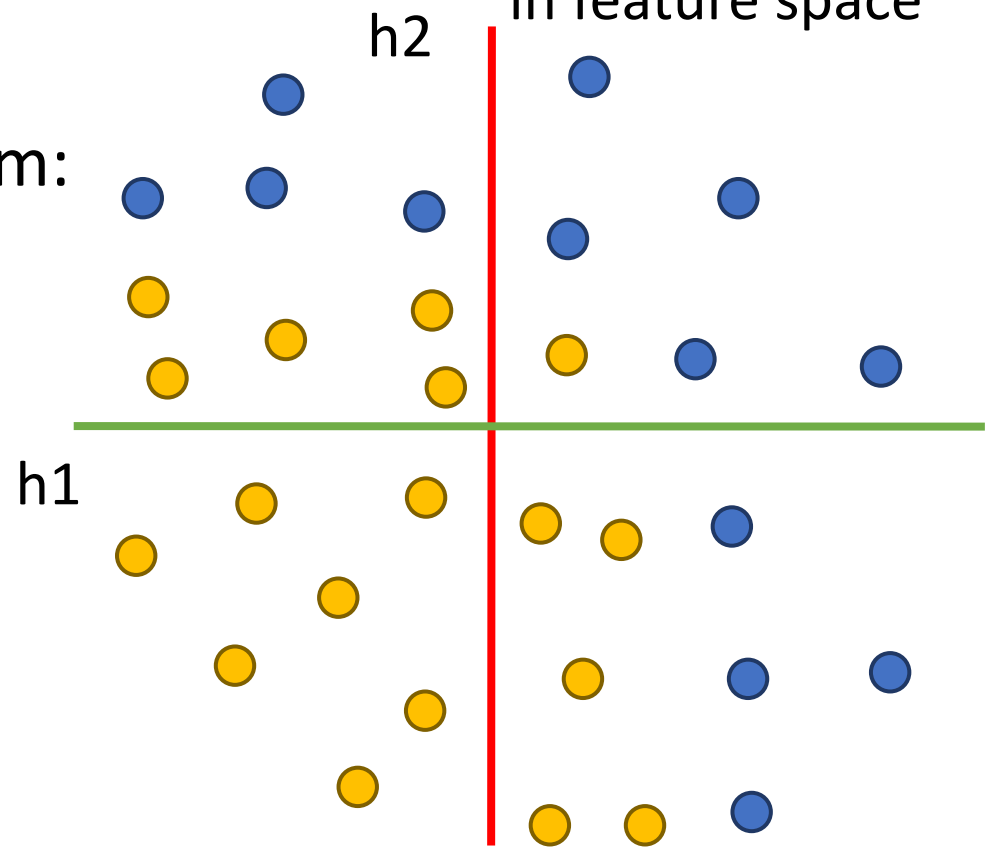
Consider a linear transform: h = Wx
Where x, h are both 2-dimensional

Points not linearly separable in original space



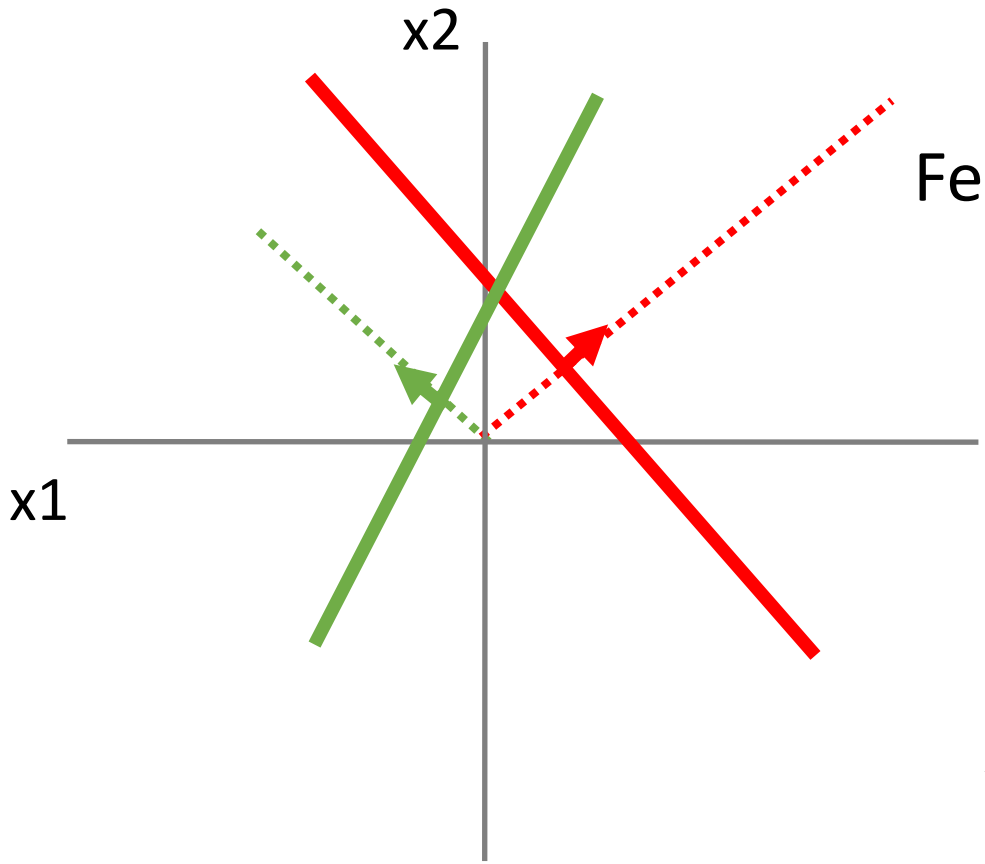
Feature transform:
h = Wx

Not linearly separable in feature space

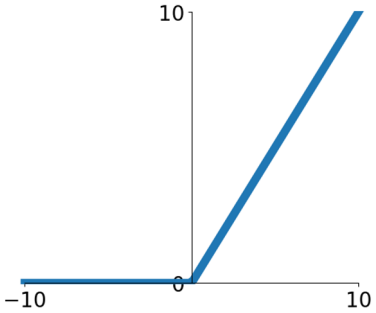


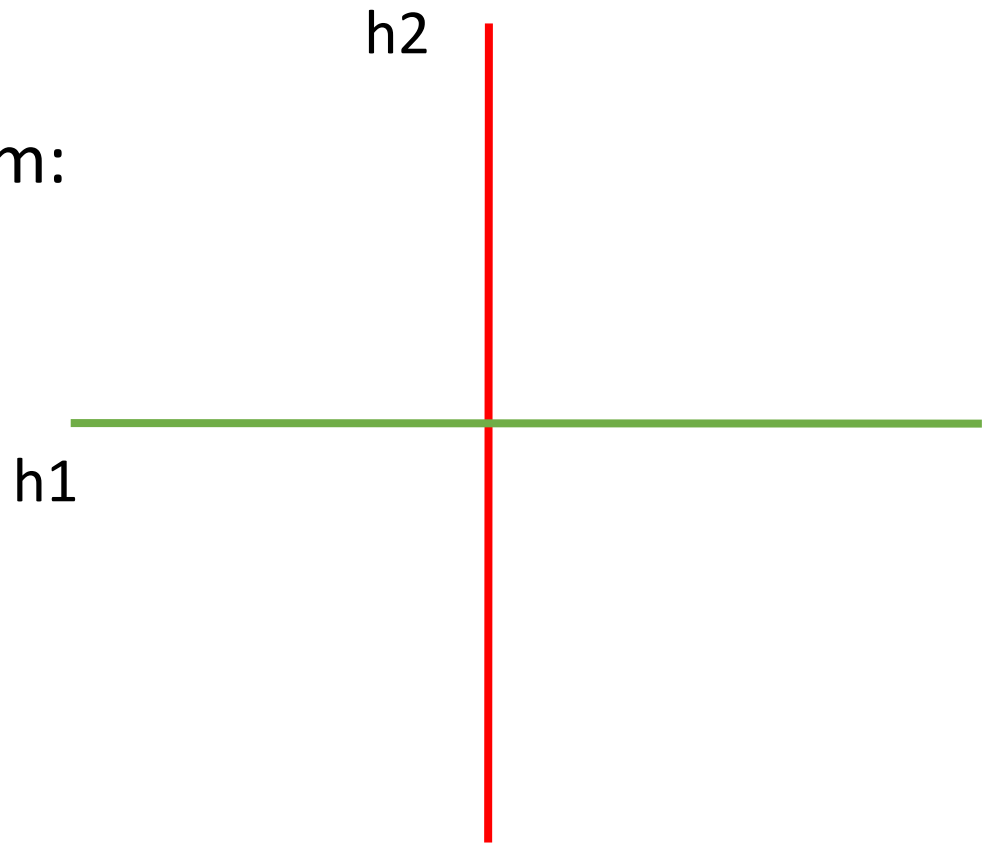
Space Warping

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



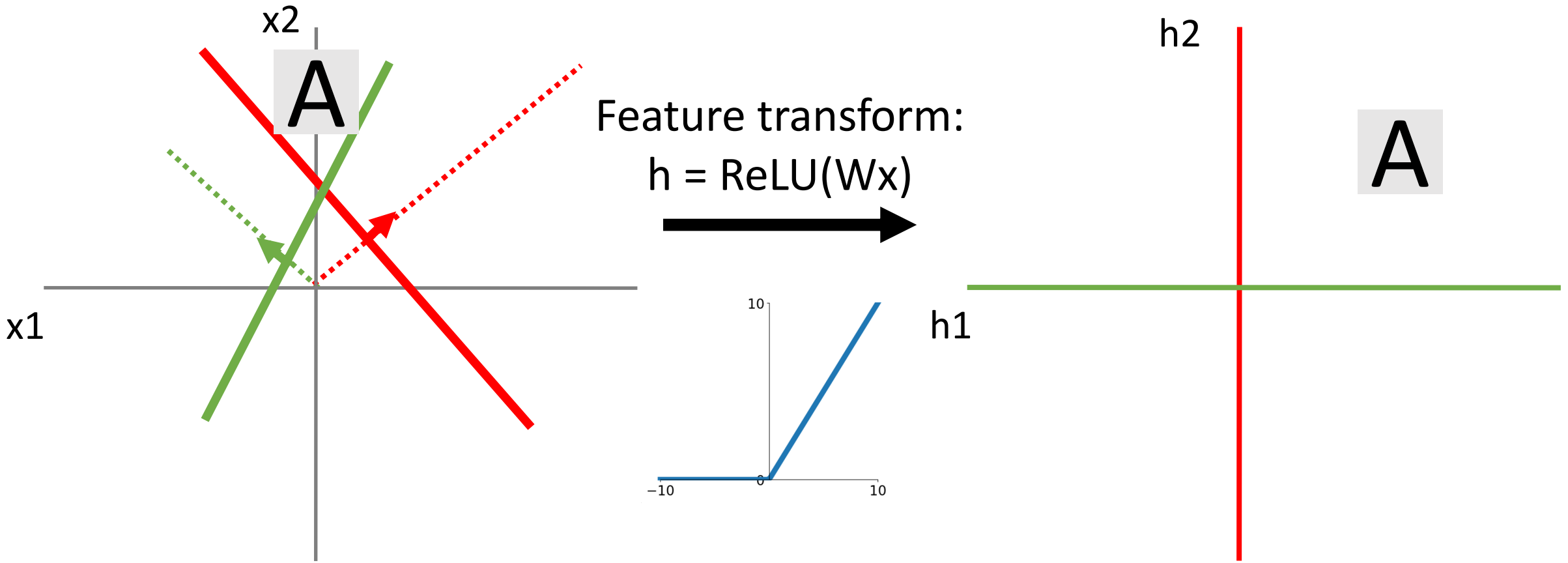
Feature transform:
h = ReLU(Wx)





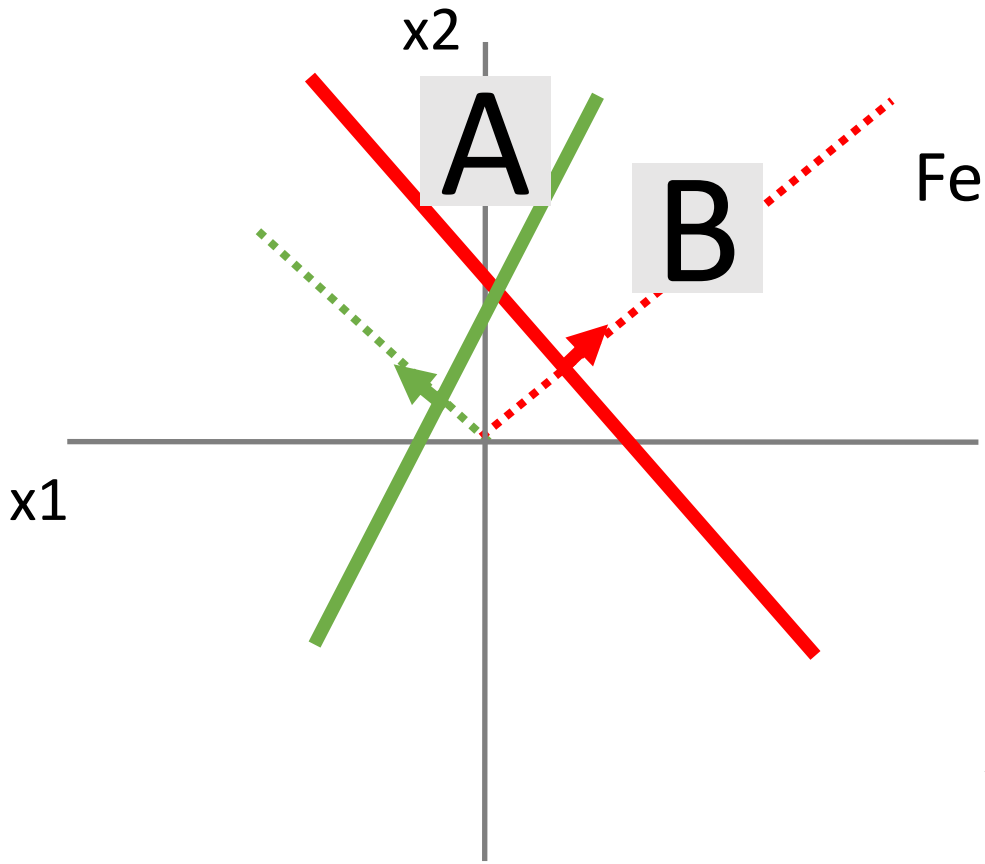
Space Warping

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

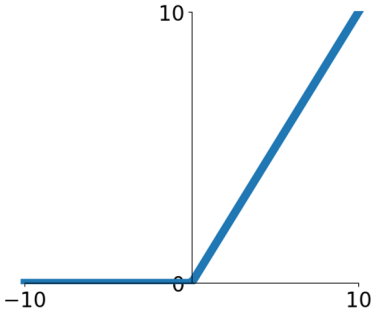


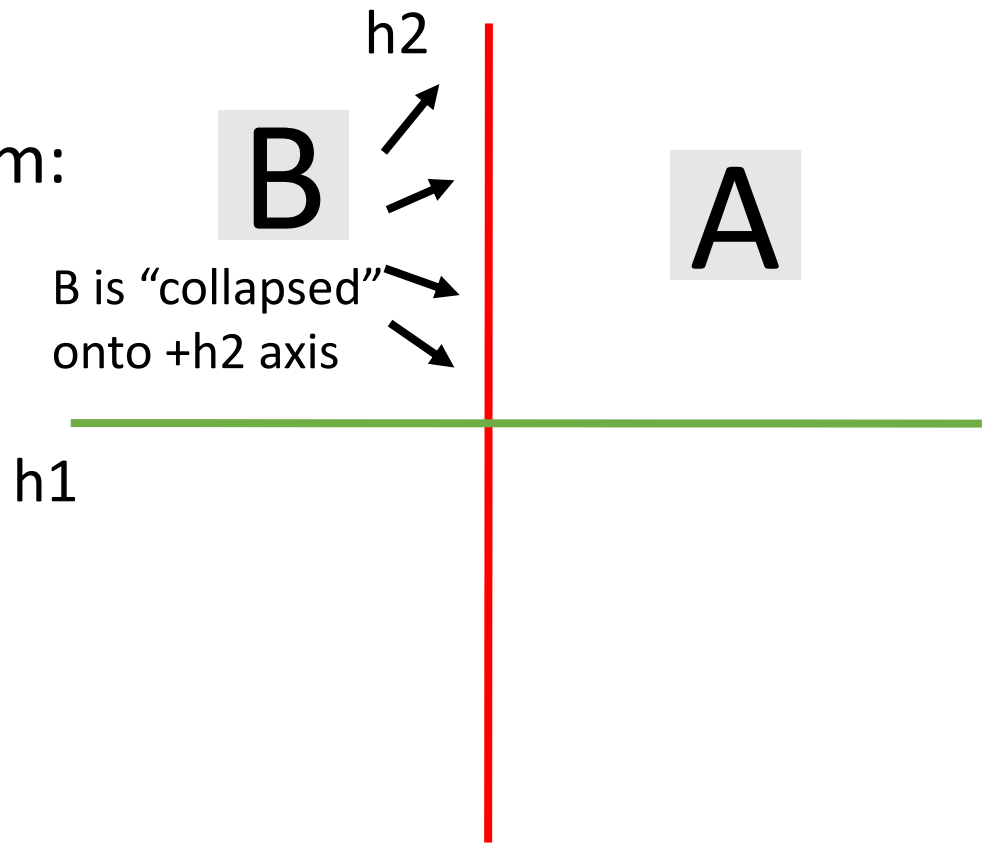
Space Warping

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



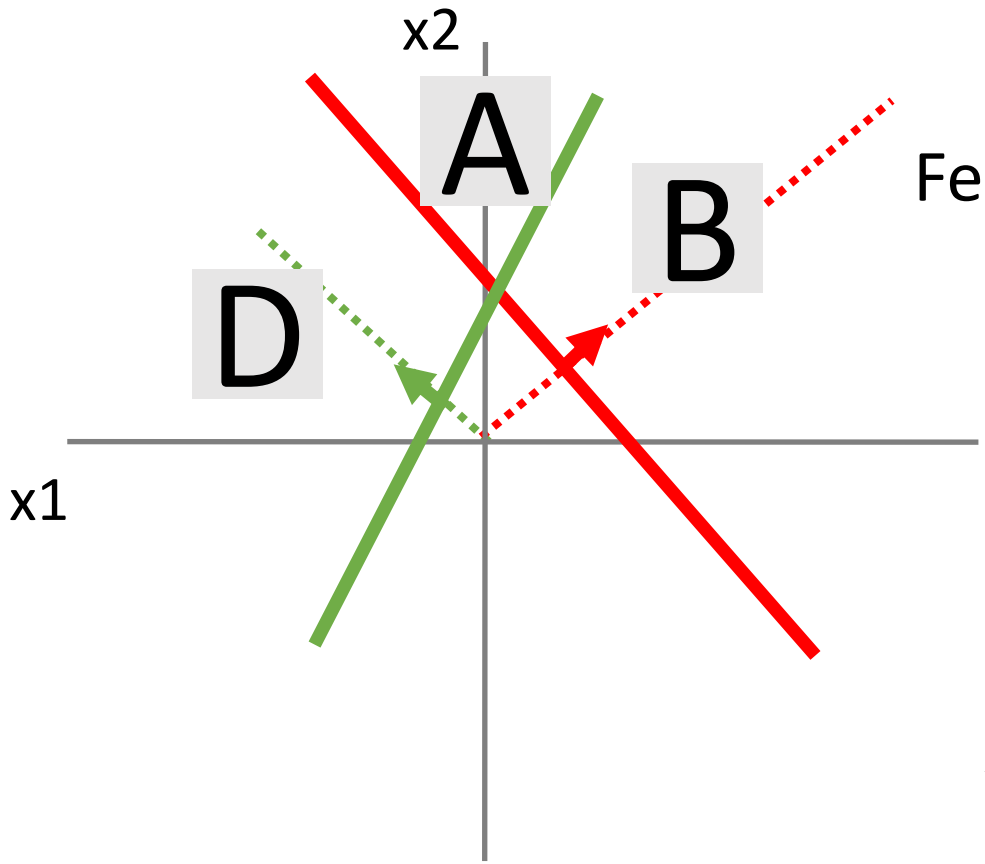
Feature transform:
h = ReLU(Wx)



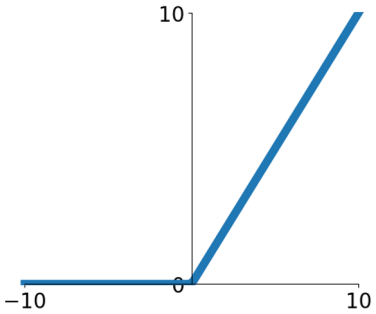


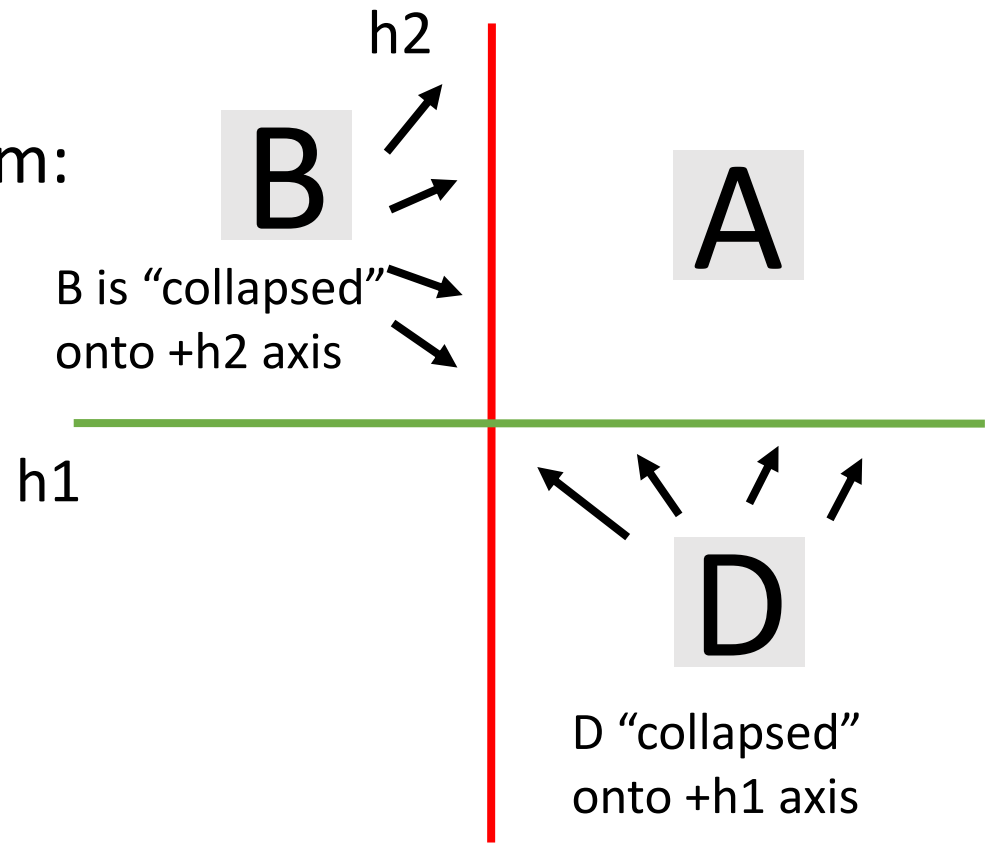
Space Warping

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



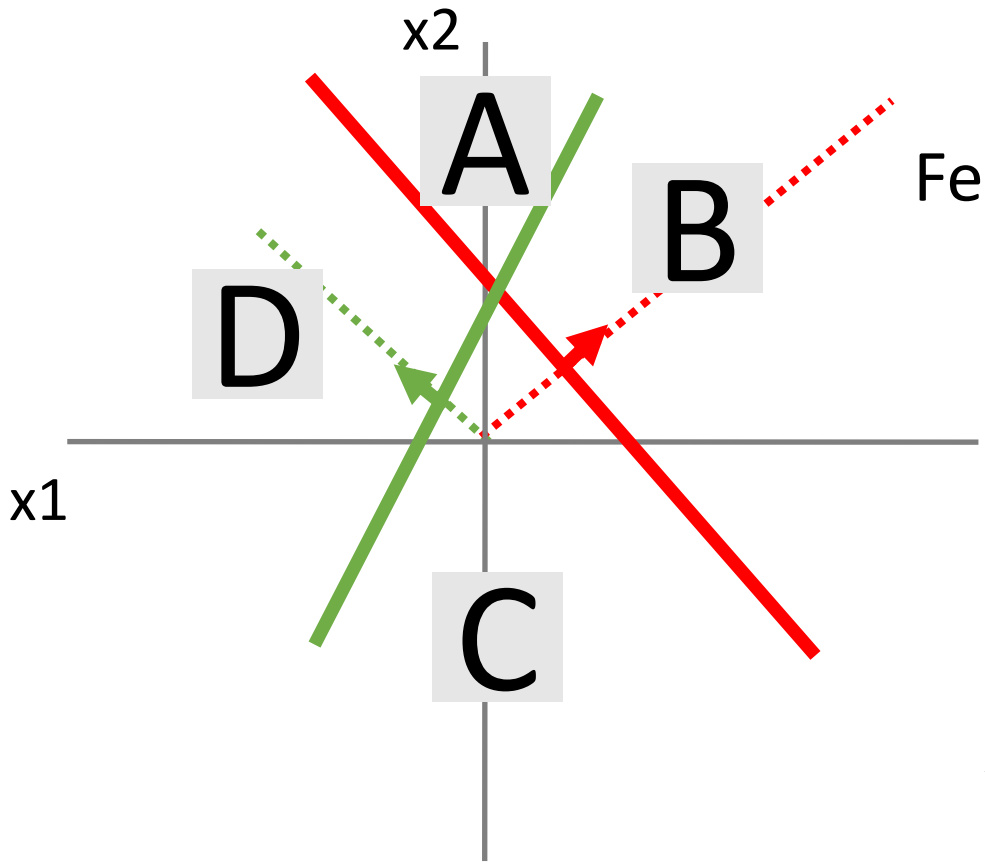
Feature transform:
h = ReLU(Wx)



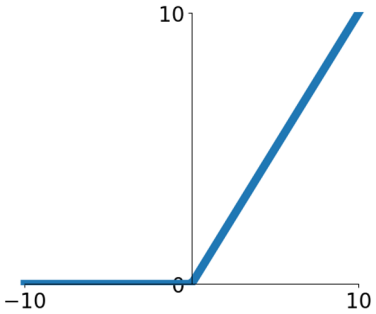


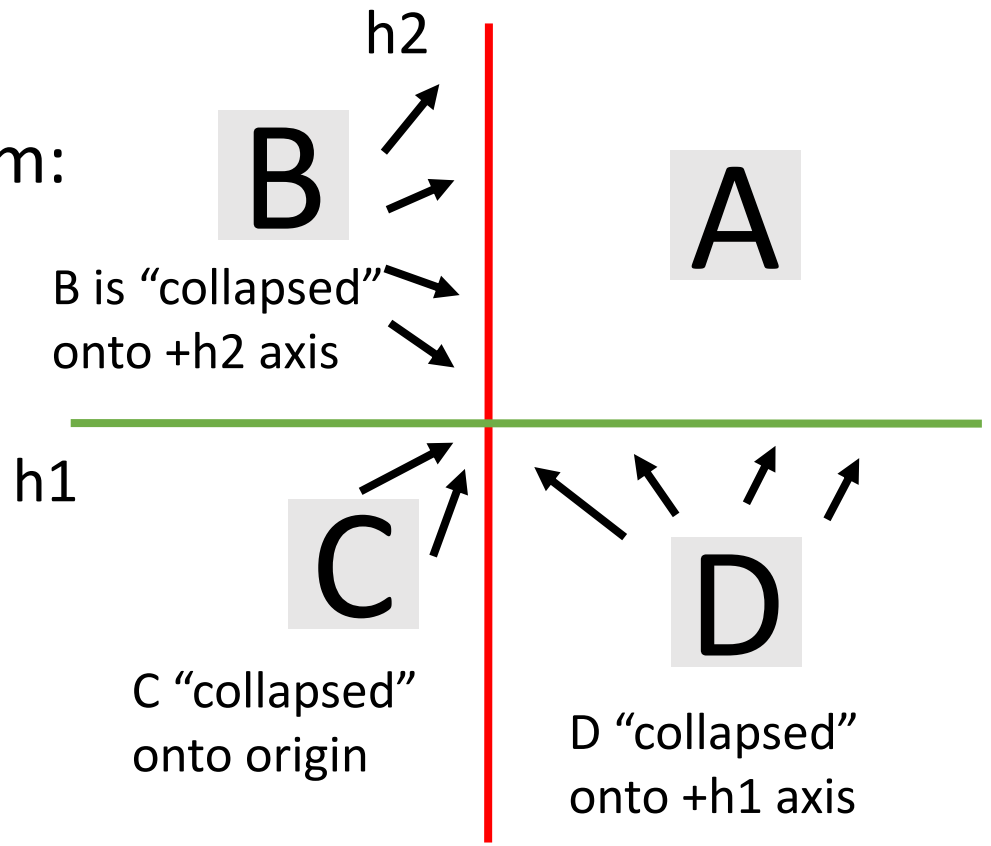
Space Warping

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



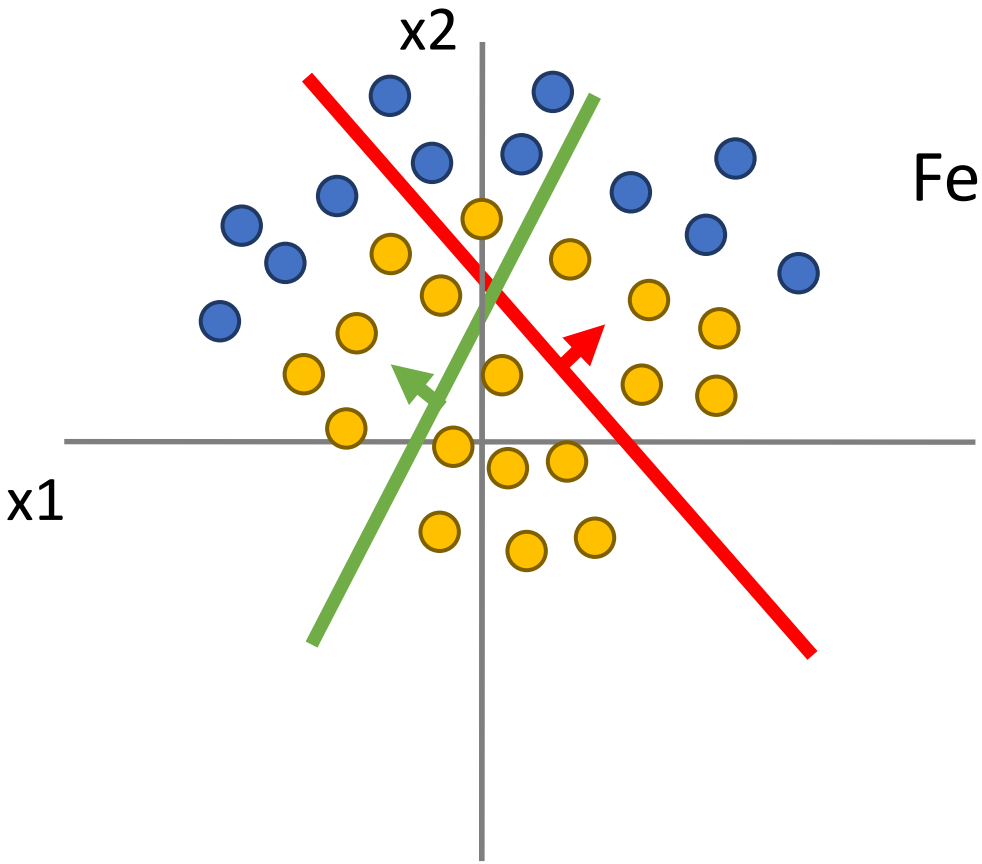
Feature transform:
h = ReLU(Wx)





Space Warping

Points not linearly separable in original space



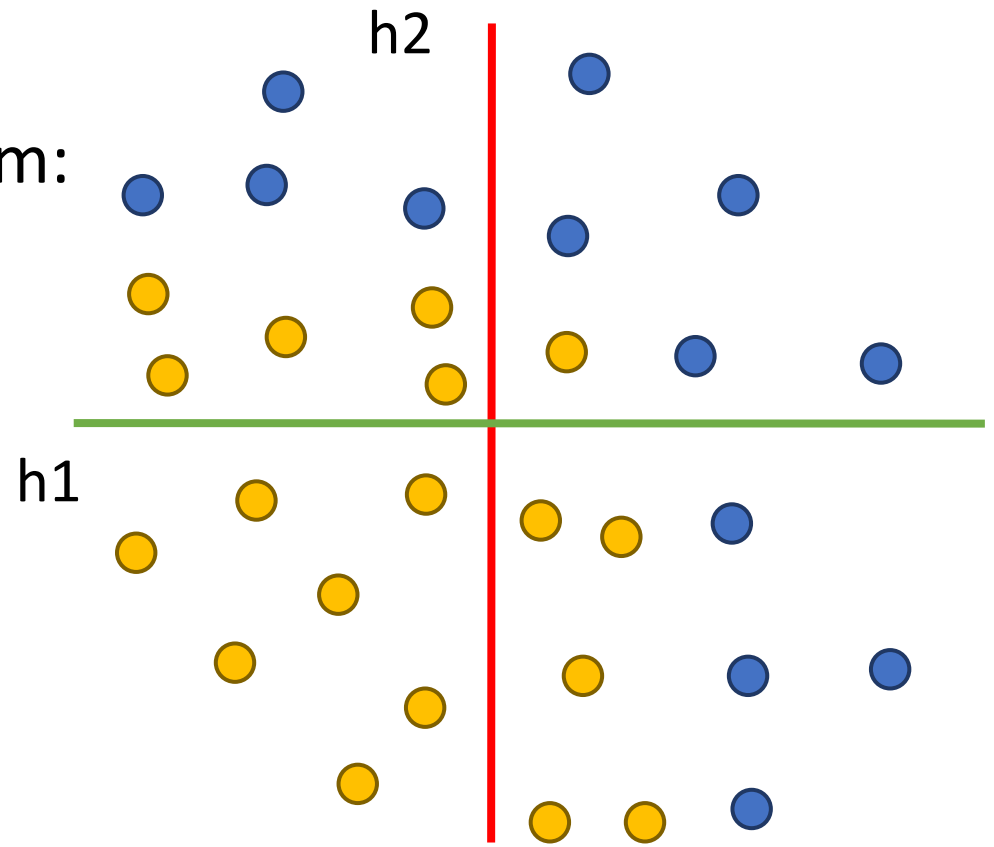
Feature transform:

h = Wx

Consider a neural net hidden layer:

h = ReLU(Wx) = max(0, Wx)

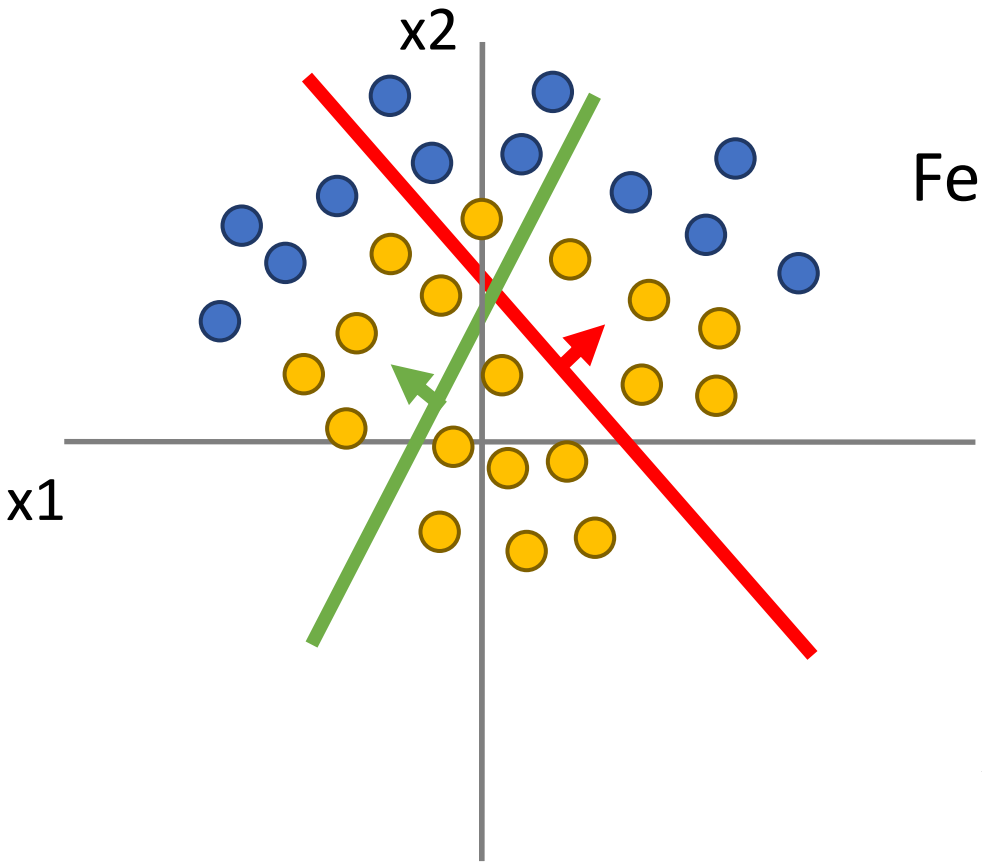
Where x, h are both 2-dimensional



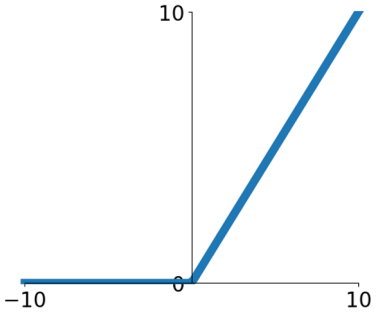
Space Warping

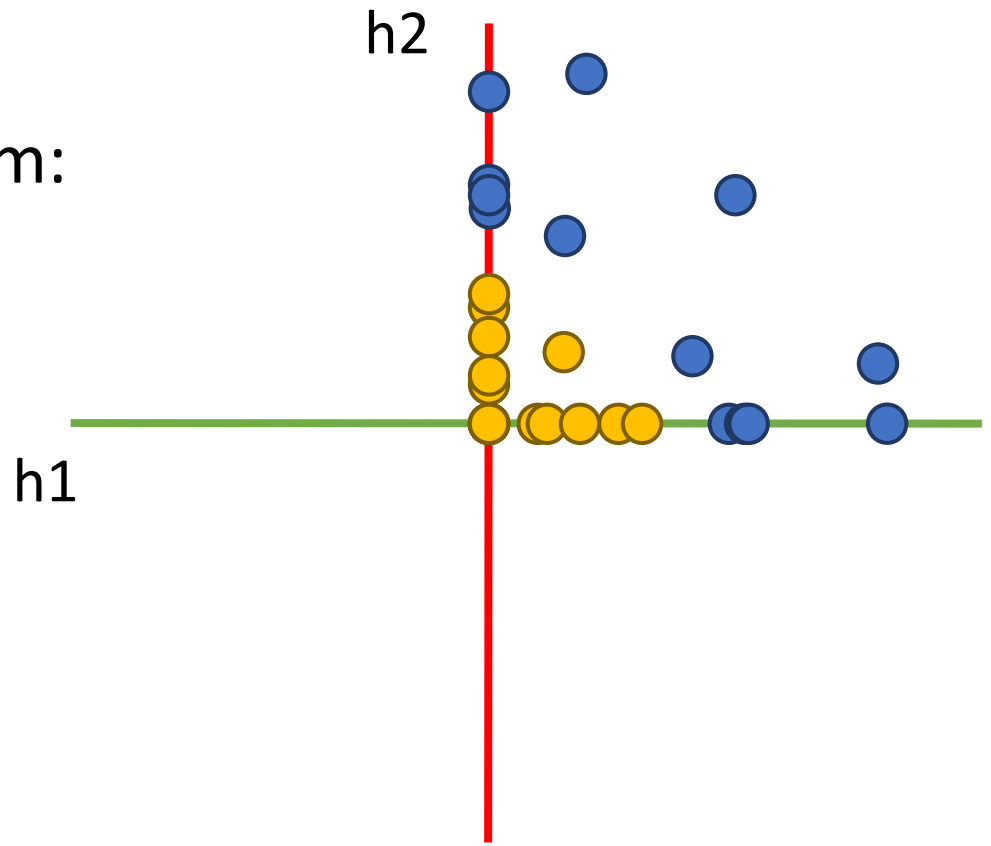
Consider a neural net hidden layer:
h = ReLU(Wx) = max(0, Wx)
Where x, h are both 2-dimensional

Points not linearly separable in original space



Feature transform:
h = ReLU(Wx)

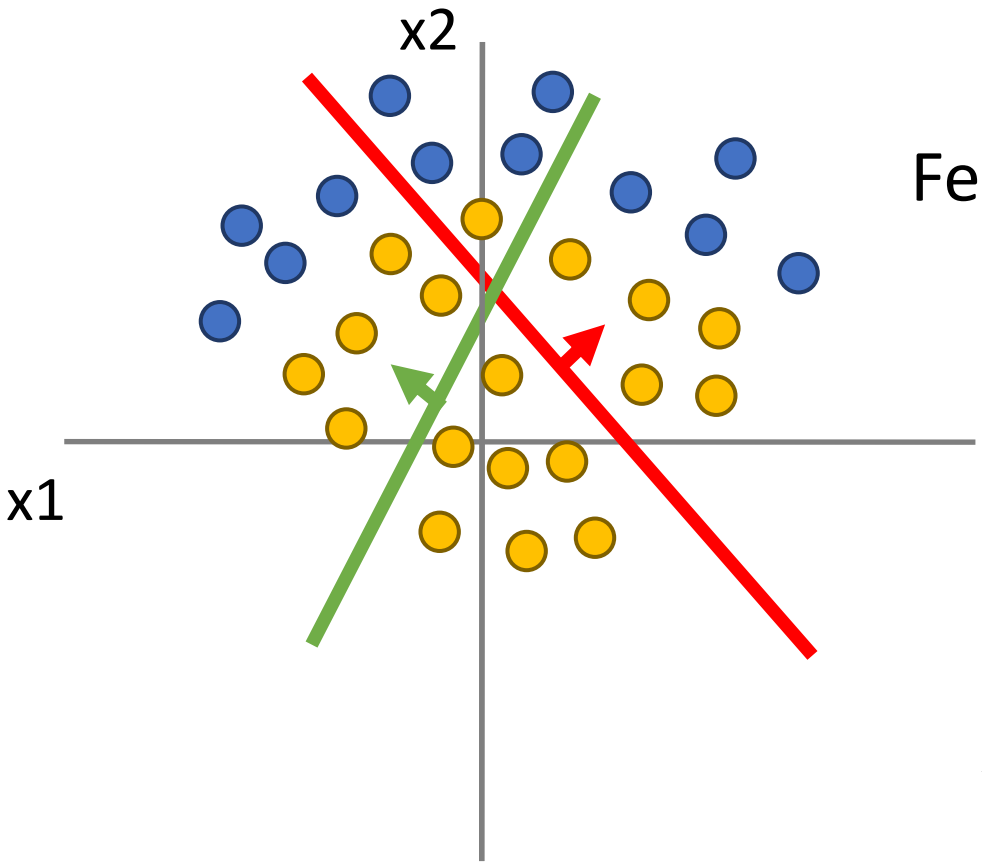




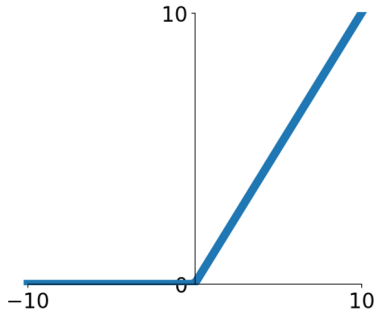
Space Warping

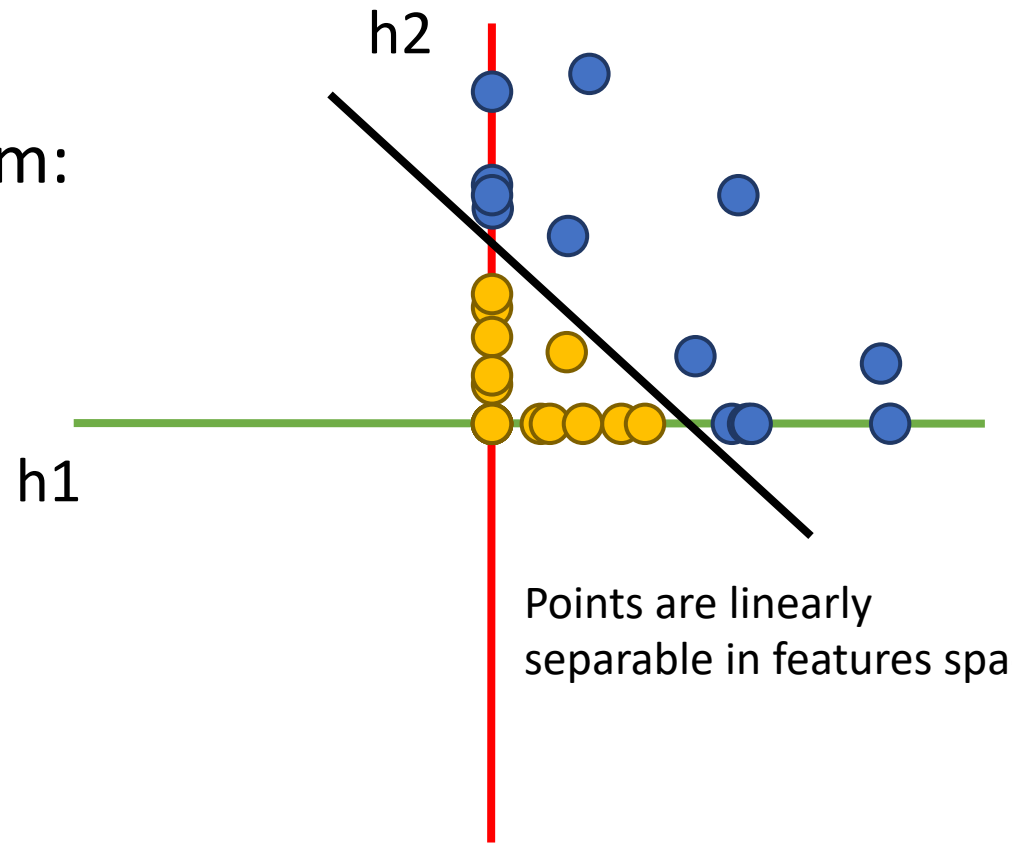
Consider a neural net hidden layer:
h = ReLU(Wx) = max(0, Wx)
Where x, h are both 2-dimensional

Points not linearly separable in original space



Feature transform:
h = ReLU(Wx)

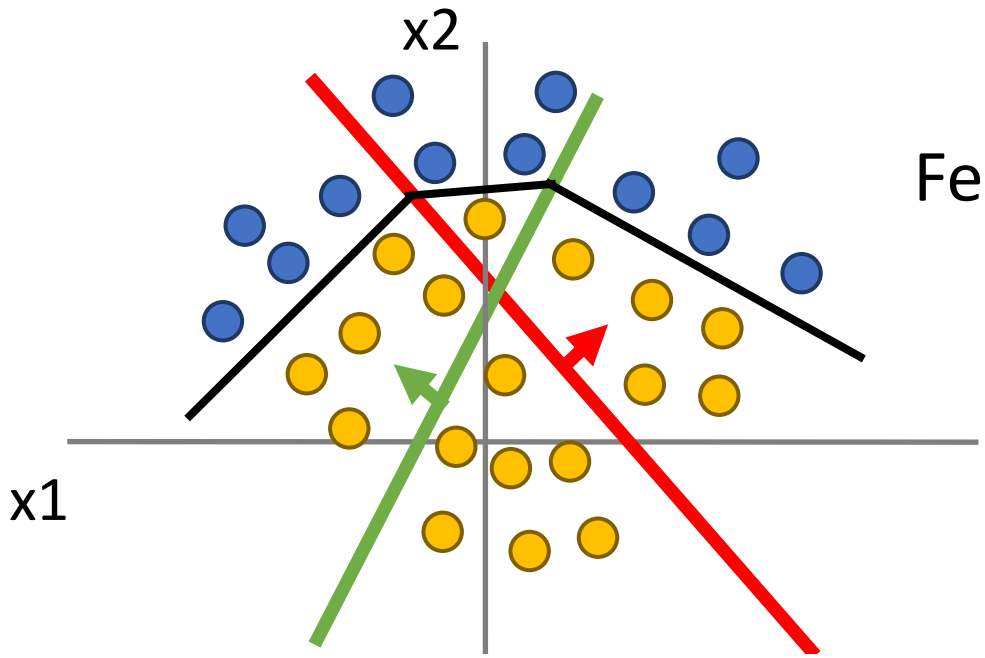




Points are linearly separable in features space!

Space Warping

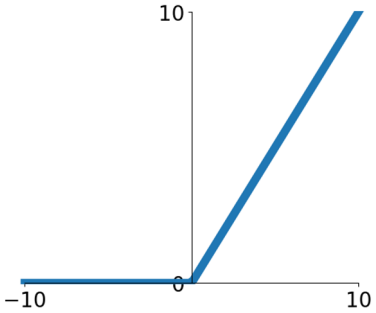
Points not linearly separable in original space

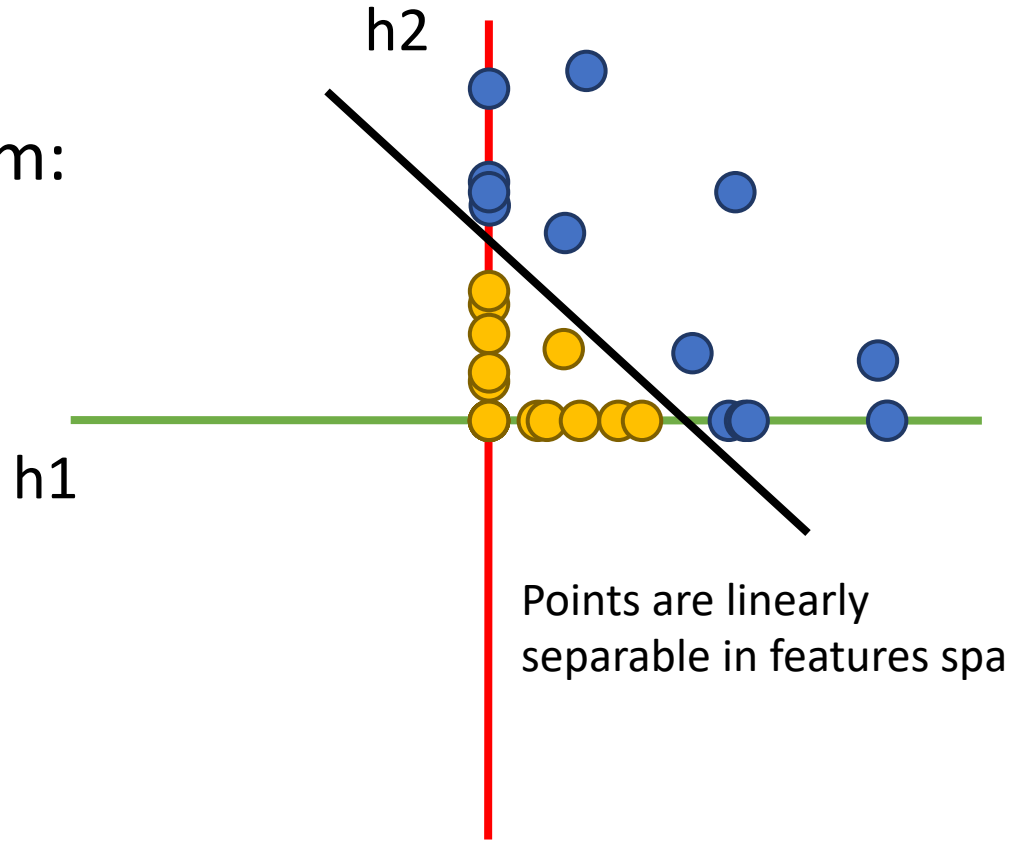


Linear classifier in feature space gives nonlinear classifier in original space

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

Feature transform:
h = ReLU(Wx)

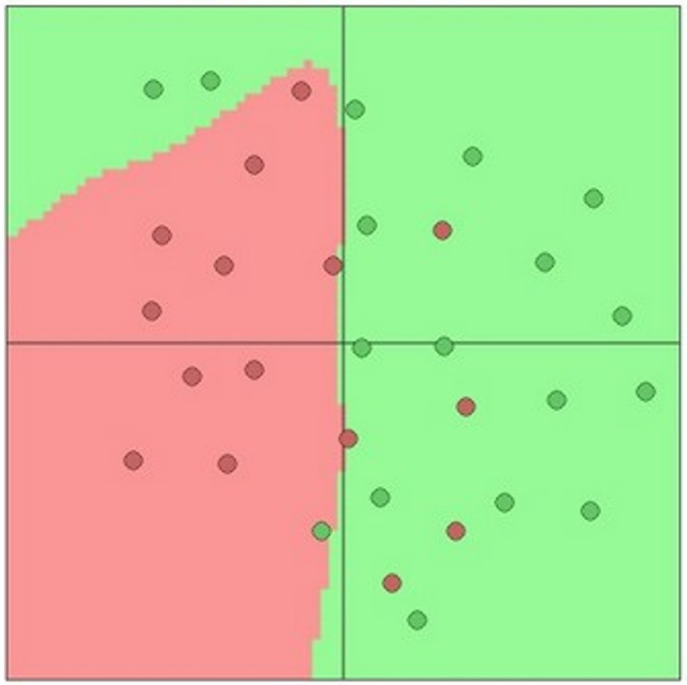




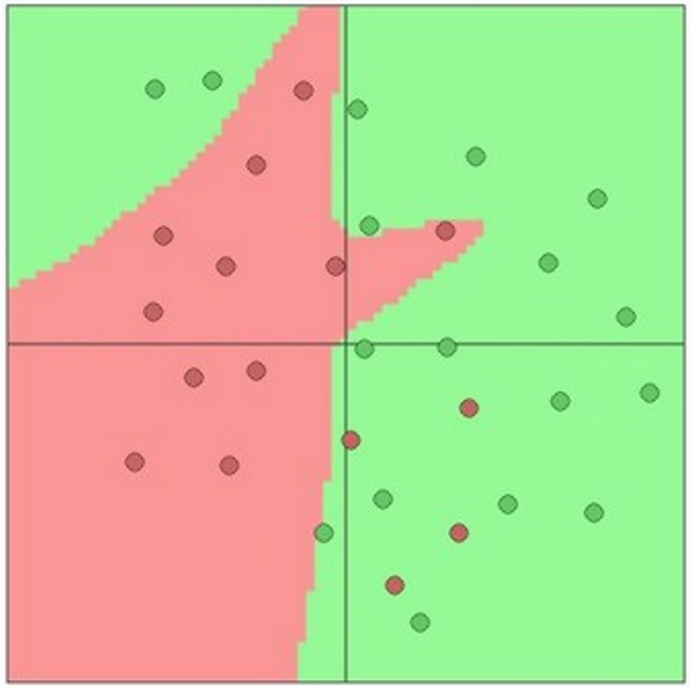
Points are linearly separable in features space!

Setting the number of layers and their sizes

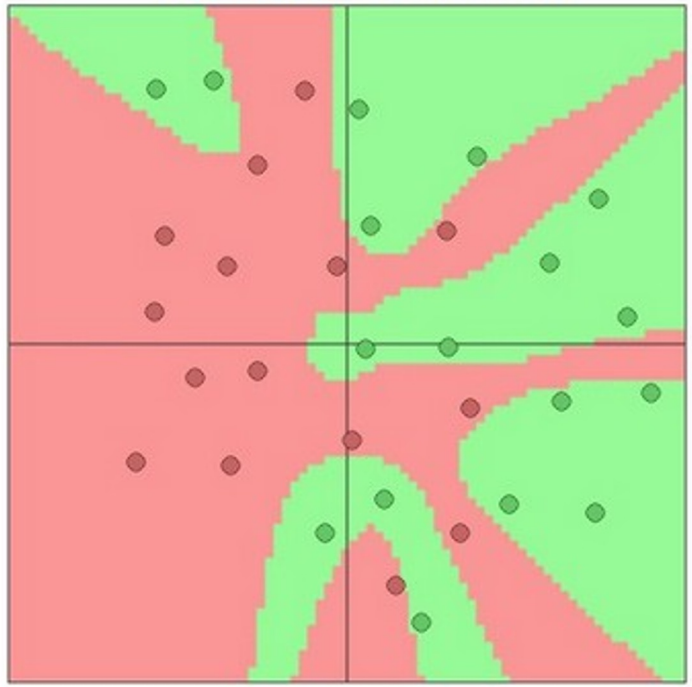
3 hidden units



6 hidden units



20 hidden units



https://playground.tensorflow.org/

A live demo

More hidden units = more capacity

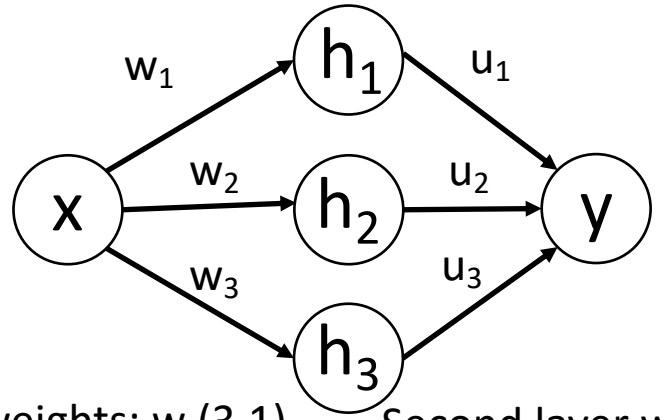
Universal Approximation

A neural network with one hidden layer can approximate any function f: R^N -> R^M with arbitrary precision*

*Many technical conditions: Only holds on compact subsets of R^N; function must be continuous; need to define “arbitrary precision”; etc

Universal Approximation

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



Input:
x (1,)

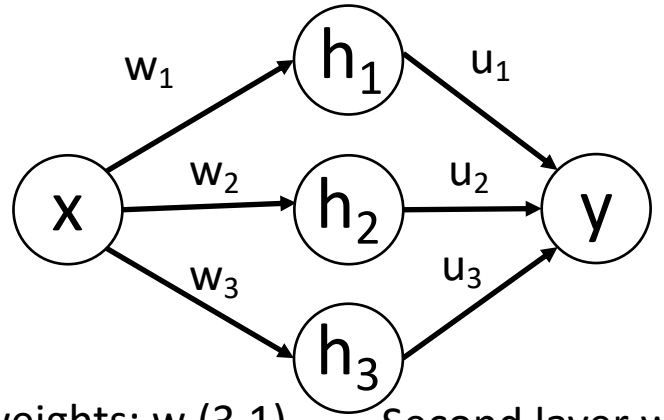
Output:
y (1,)

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

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

Universal Approximation

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



Input:
x (1,)

Output:
y (1,)

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

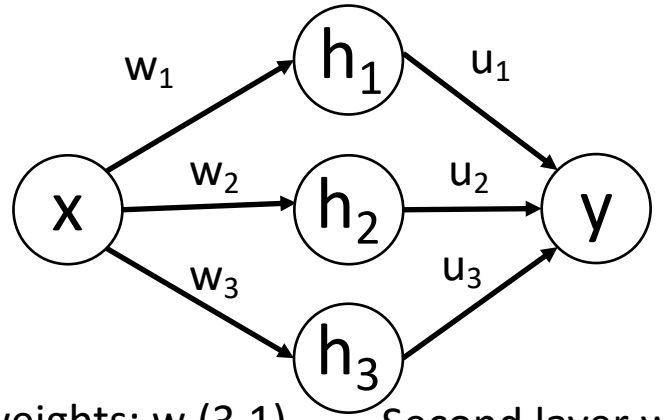
Second layer weights: u (1,3)
Second layer bias: p (1,)

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

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

Universal Approximation

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



Input:
x (1,)

Output:
y (1,)

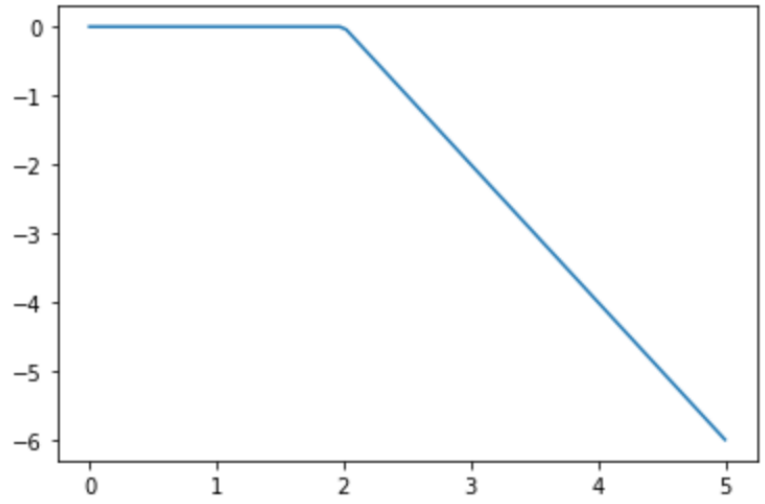
First layer weights: w (3,1)
First layer bias: b (3,)

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

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

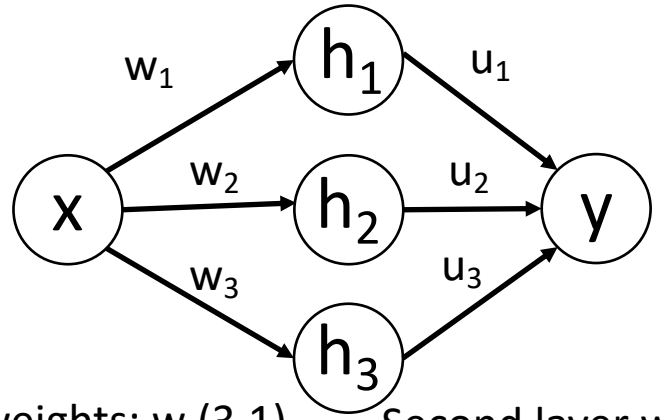
$$y = u_1 * \max(0, w_1 * x + b_1) + u_2 * \max(0, w_2 * x + b_2) + u_3 * \max(0, w_3 * x + b_3) + p$$

Output is a sum of shifted, scaled ReLUs:



Universal Approximation

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



Input:
x (1,)

Output:
y (1,)

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

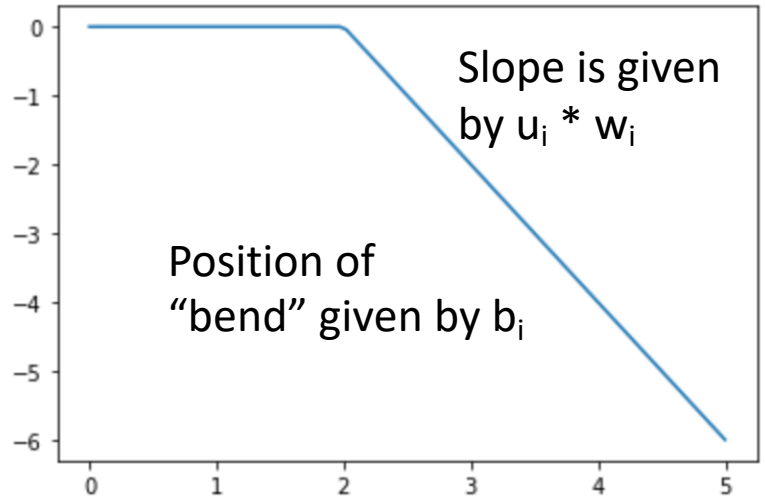
Second layer weights: u (1,3)
Second layer bias: p (1,)

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

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

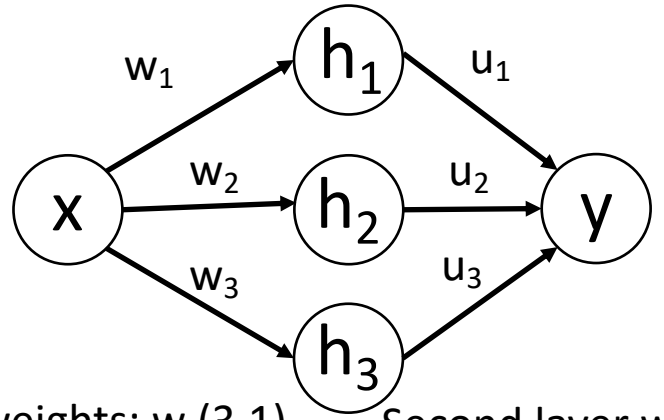
Output is a sum of shifted, scaled ReLUs:

Flip left / right based on sign of w_i



Universal Approximation

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



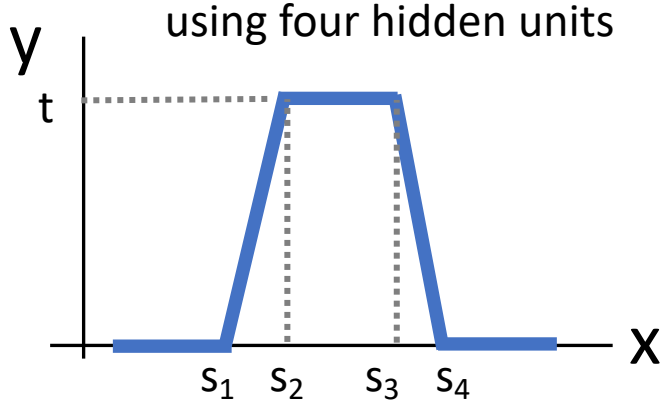
Input:
x (1,)

Output:
y (1,)

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

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

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

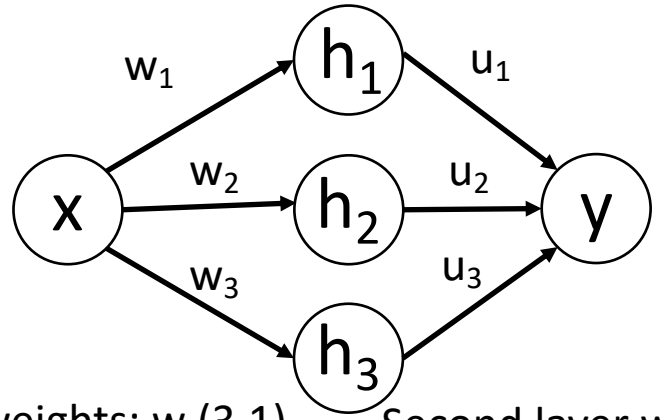


$$h_1 = \max(0, w_1 * x + b_1)$$
$$h_2 = \max(0, w_2 * x + b_2)$$
$$h_3 = \max(0, w_3 * x + b_3)$$
$$y = u_1 h_1 + u_2 * h_2 + u_3 * h_3 + p$$

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

Universal Approximation

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



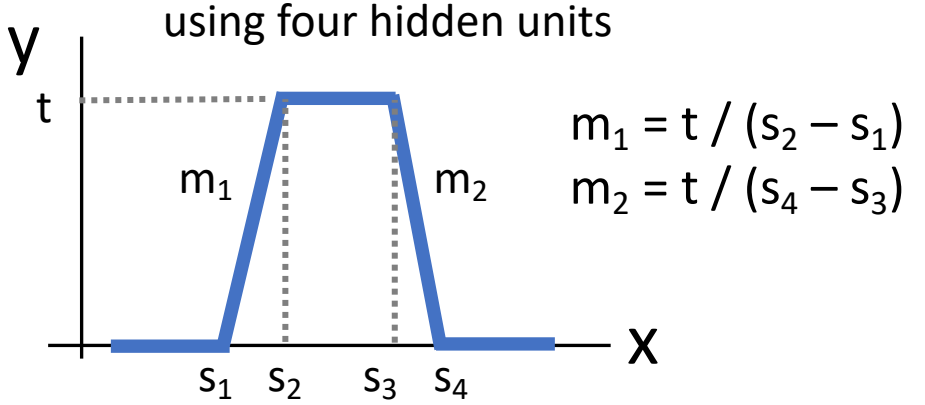
Input:
x (1,)

Output:
y (1,)

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

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

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



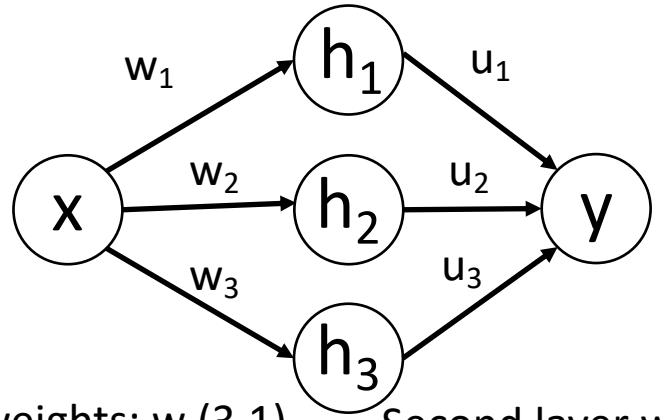
$m_1 = t / (s_2 - s_1)$
$m_2 = t / (s_4 - s_3)$

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

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

Universal Approximation

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



Input:
x (1,)

Output:
y (1,)

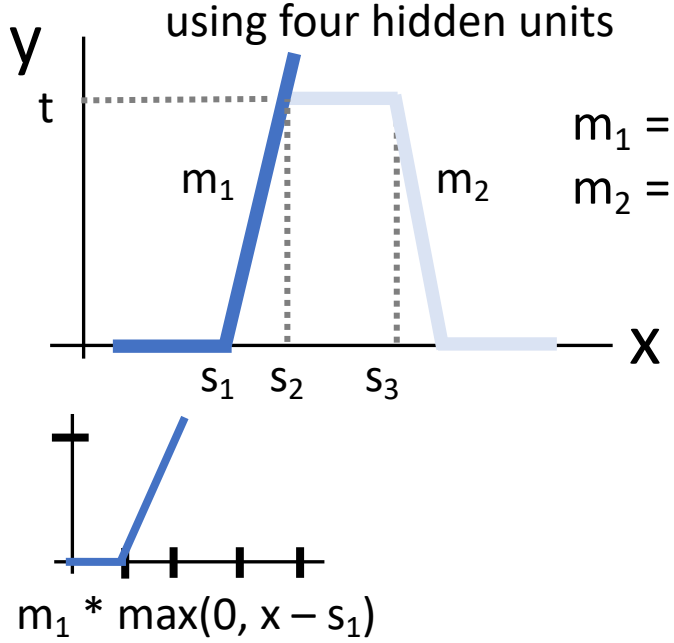
First layer weights: w (3,1)
First layer bias: b (3,)

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

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

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

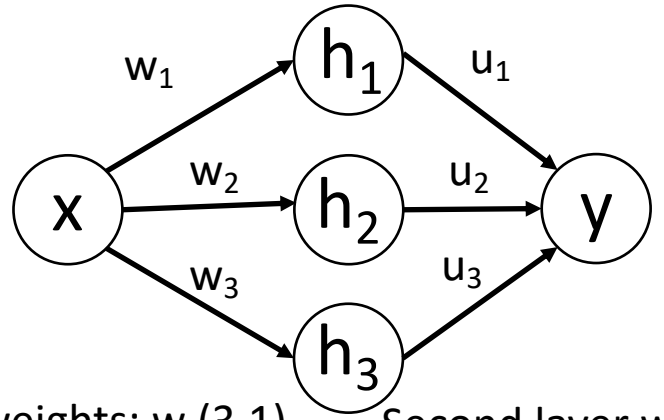
We can build a "bump function" using four hidden units



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

Universal Approximation

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



Input:
x (1,)

Output:
y (1,)

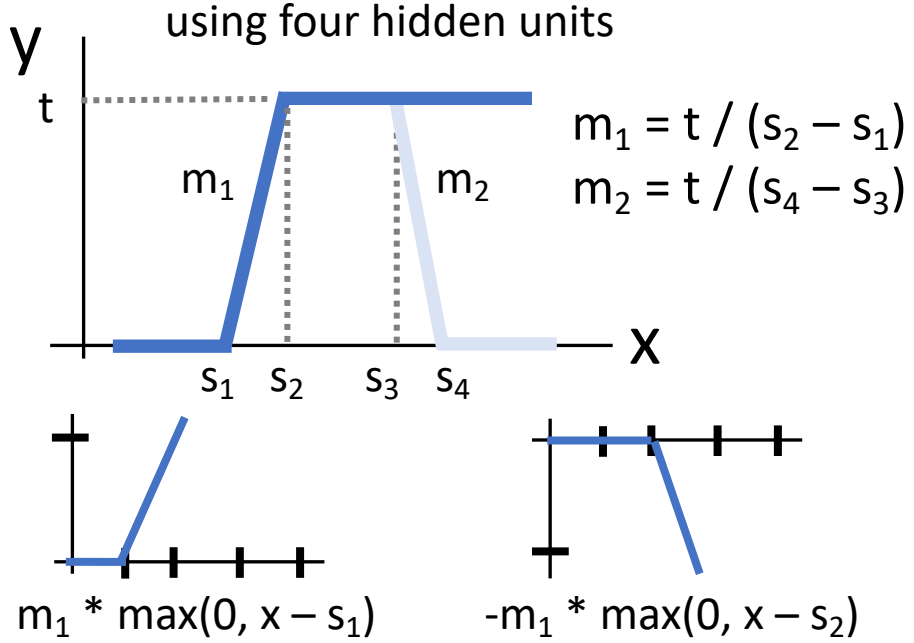
First layer weights: w (3,1)
First layer bias: b (3,)

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

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

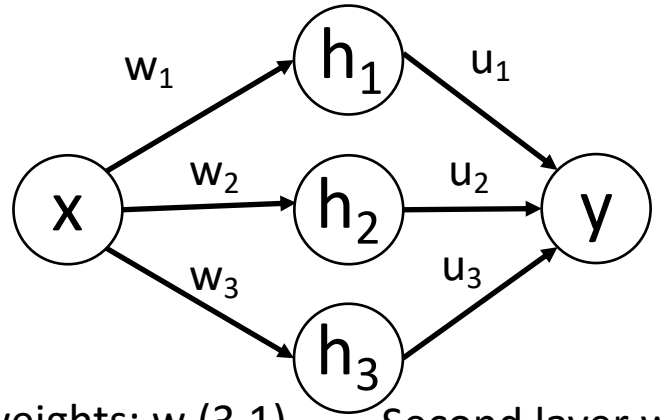
$$y = u_1 * \max(0, w_1 * x + b_1) + u_2 * \max(0, w_2 * x + b_2) + u_3 * \max(0, w_3 * x + b_3) + p$$

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



Universal Approximation

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



Input:
x (1,)

Output:
y (1,)

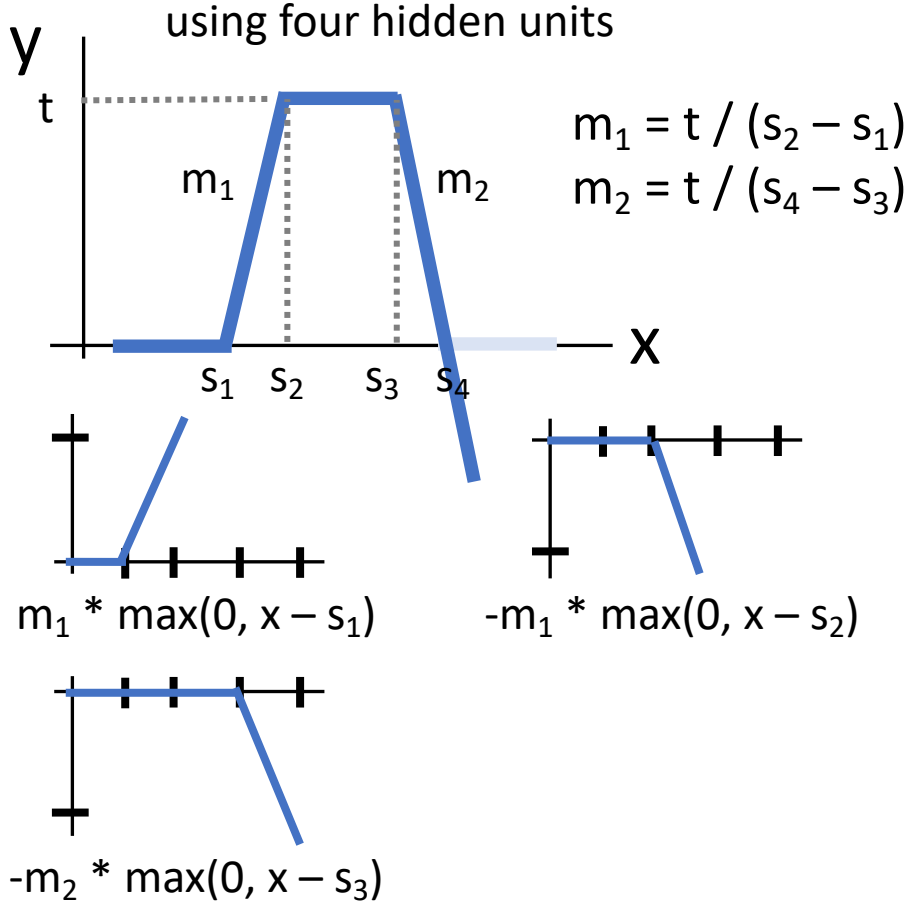
First layer weights: w (3,1)
First layer bias: b (3,)

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

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

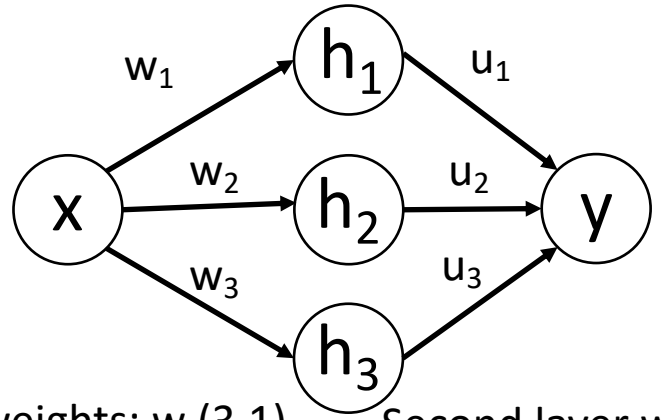
$$y = u_1 * \max(0, w_1 * x + b_1) + u_2 * \max(0, w_2 * x + b_2) + u_3 * \max(0, w_3 * x + b_3) + p$$

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



Universal Approximation

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



Input:
x (1,)

Output:
y (1,)

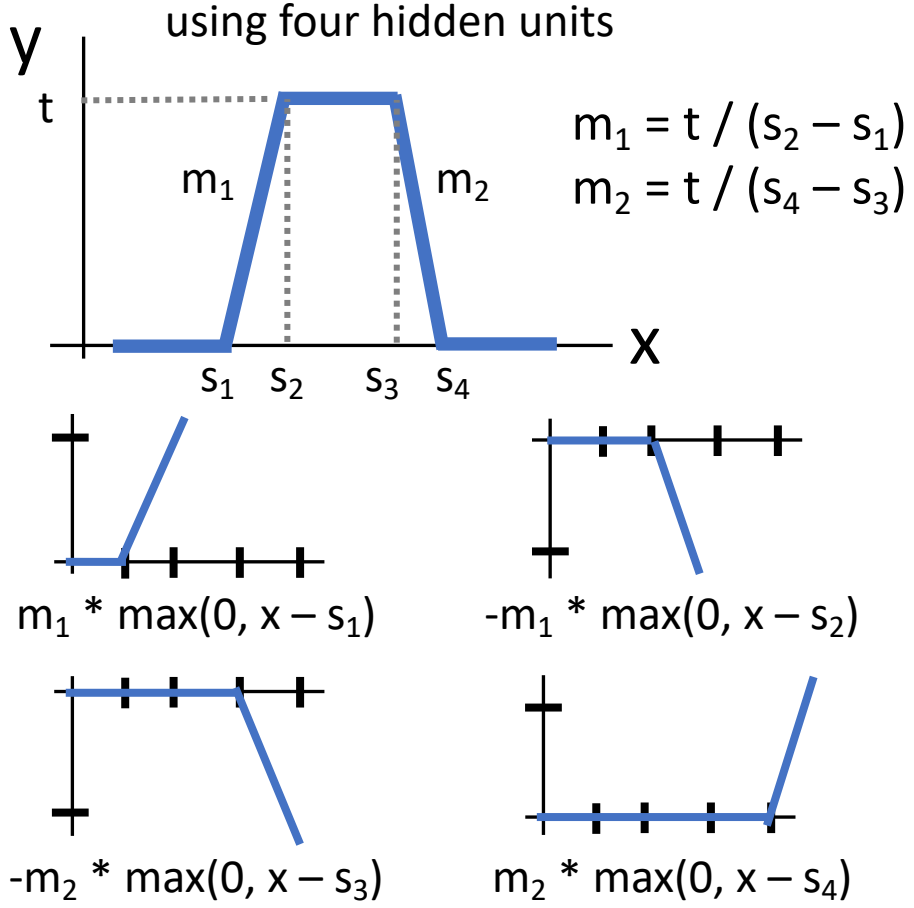
First layer weights: w (3,1)
First layer bias: b (3,)

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

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

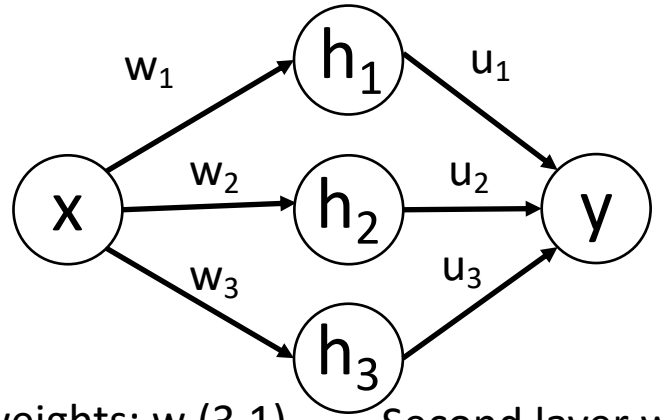
$$y = u_1 * \max(0, w_1 * x + b_1) + u_2 * \max(0, w_2 * x + b_2) + u_3 * \max(0, w_3 * x + b_3) + p$$

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



Universal Approximation

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



Input:
x (1,)

Output:
y (1,)

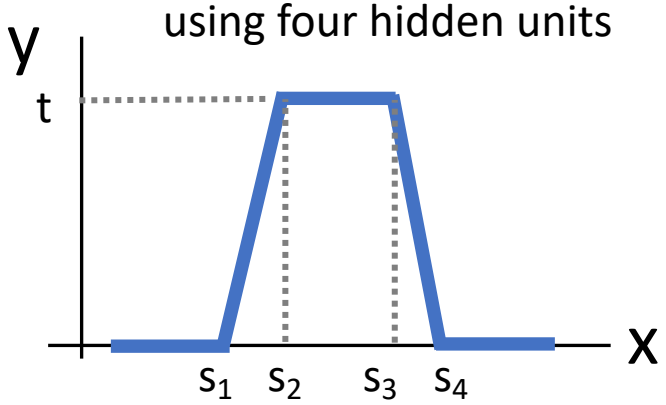
First layer weights: w (3,1)
First layer bias: b (3,)

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

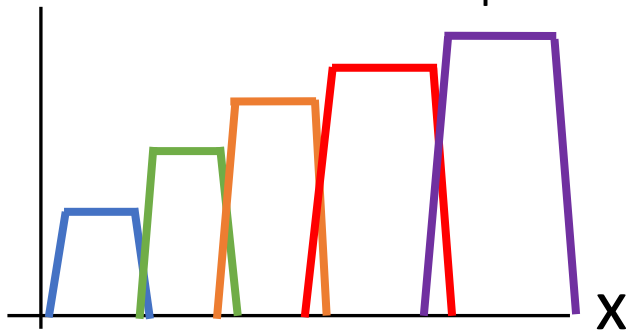
$$h_1 = \max(0, w_1 * x + b_1)$$
$$h_2 = \max(0, w_2 * x + b_2)$$
$$h_3 = \max(0, w_3 * x + b_3)$$
$$y = u_1 h_1 + u_2 * h_2 + u_3 * h_3 + p$$

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

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

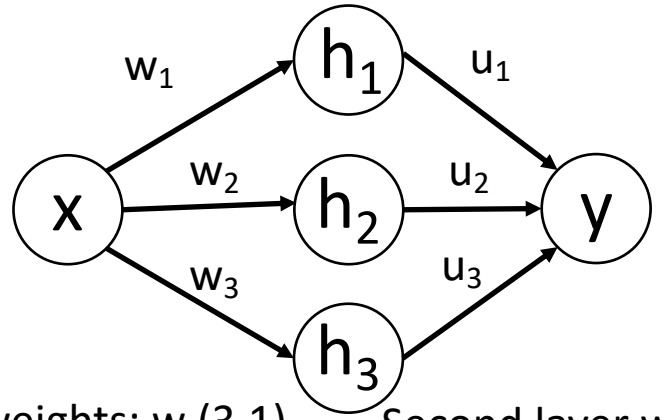


With 4K hidden units we can build a sum of K bumps



Universal Approximation

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



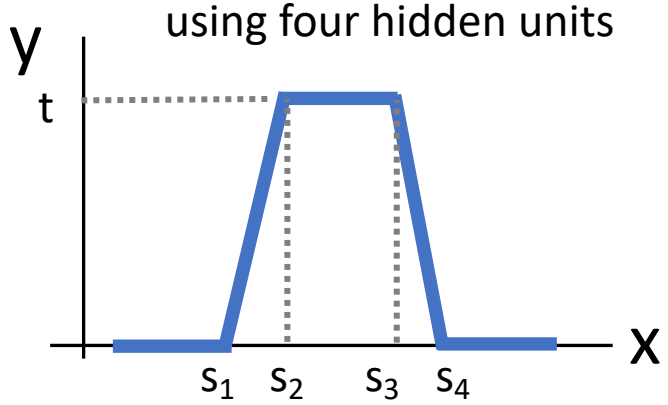
First layer weights: w (3,1)
First layer bias: b (3,)

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

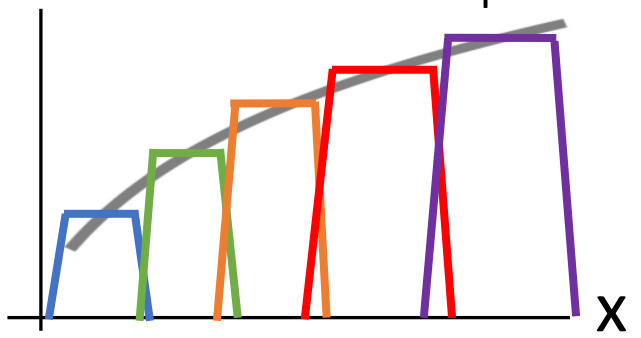
$$h_1 = \max(0, w_1 * x + b_1)$$
$$h_2 = \max(0, w_2 * x + b_2)$$
$$h_3 = \max(0, w_3 * x + b_3)$$
$$y = u_1 h_1 + u_2 * h_2 + u_3 * h_3 + p$$

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

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



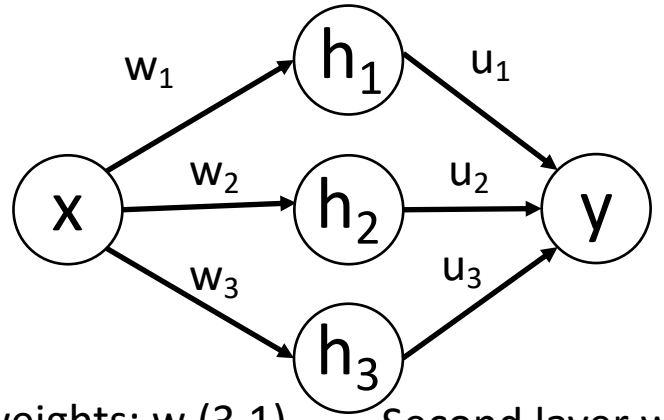
With 4K hidden units we can build a sum of K bumps



Approximate functions with bumps!

Universal Approximation

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



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

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

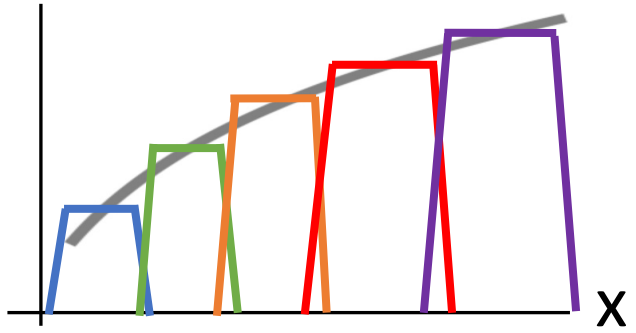
$$h_1 = \max(0, w_1 * x + b_1)$$
$$h_2 = \max(0, w_2 * x + b_2)$$
$$h_3 = \max(0, w_3 * x + b_3)$$
$$y = u_1 h_1 + u_2 * h_2 + u_3 * h_3 + p$$

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

What about…

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

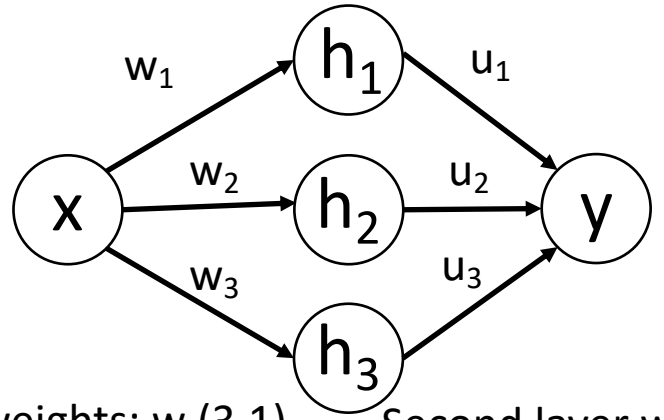
See Nielsen, Chapter 4



Approximate functions with bumps!

Universal Approximation

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



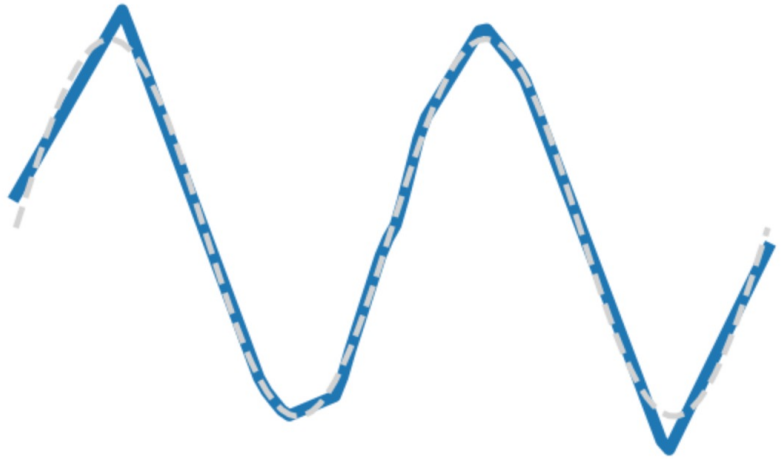
First layer weights: w (3,1)
First layer bias: b (3,)

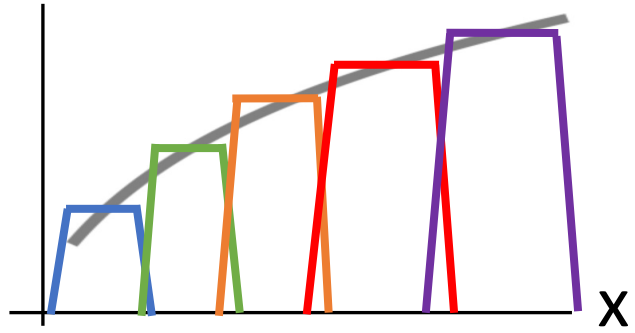
Second layer weights: u (1,3)
Second layer bias: p (1,)

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

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

Reality check: Networks don't really learn bumps!

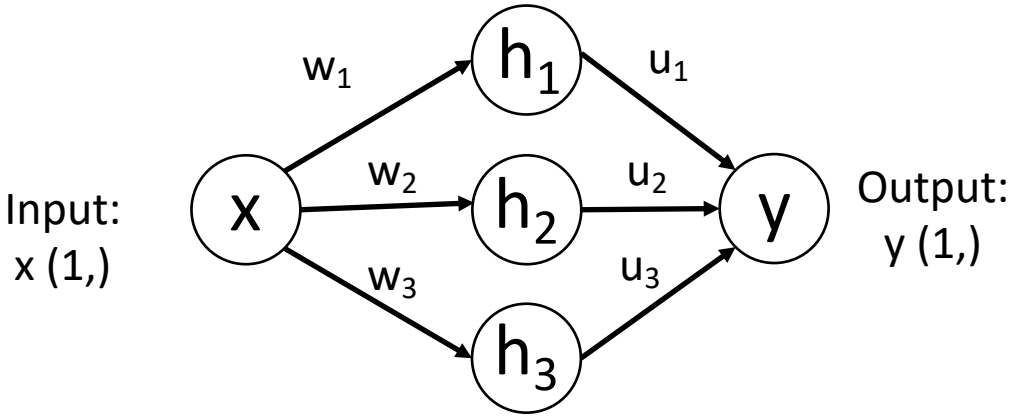




Approximate functions with bumps!

Universal Approximation

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



Universal approximation tells us:

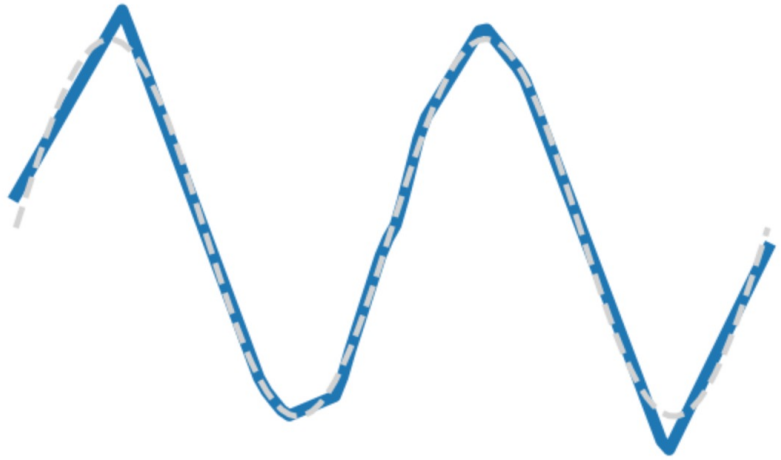
- Neural nets can represent any function

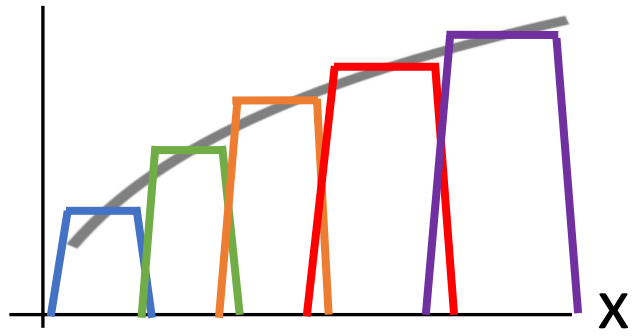
Universal approximation DOES NOT tell us:

- 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 $x_1, x_2 \in X, t \in [0, 1]$,

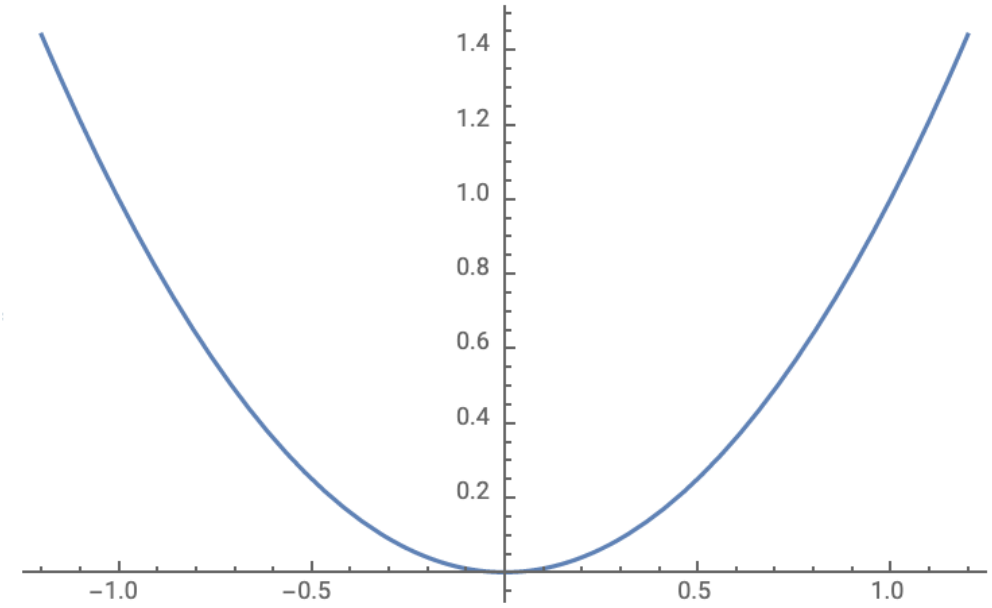
$$f(tx_1 + (1-t)x_2) \leq tf(x_1) + (1-t)f(x_2)$$

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:

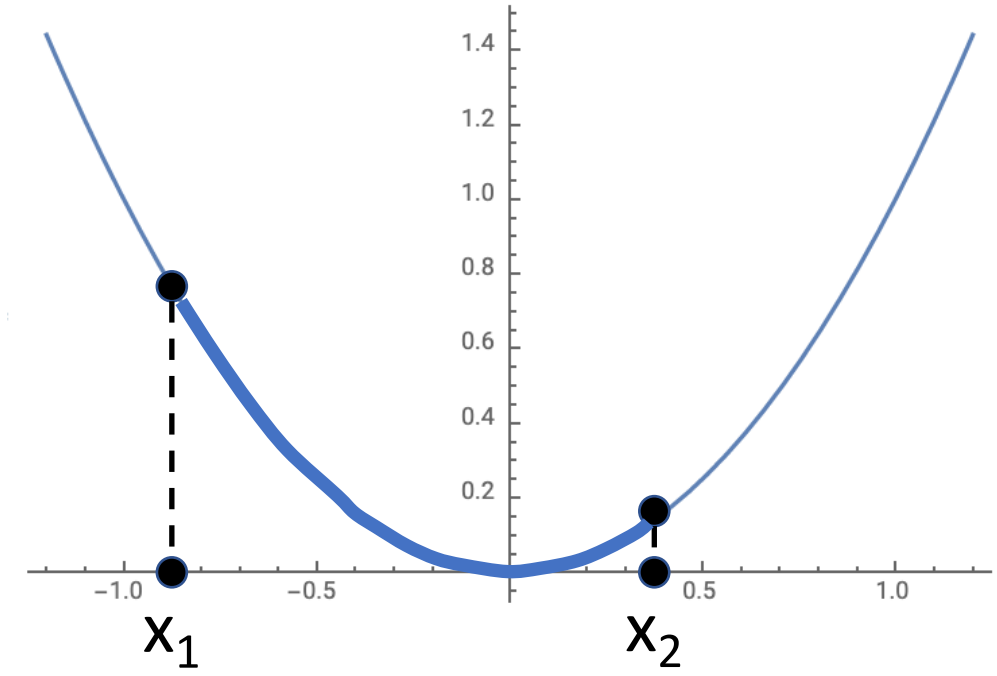


Convex Functions

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

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

Example: $f(x) = x^2$ is convex:

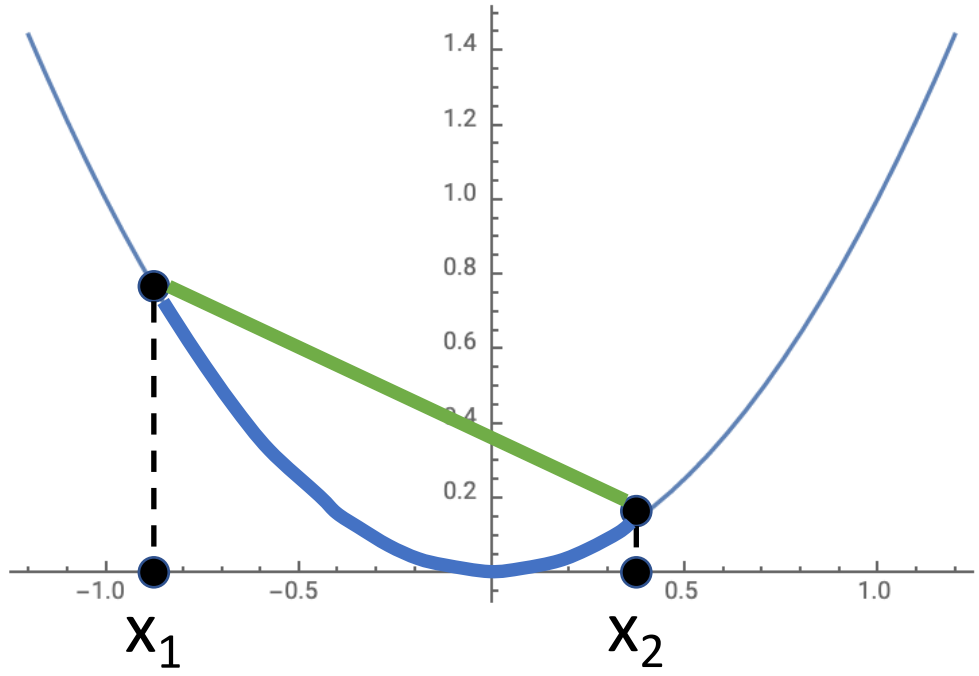


Convex Functions

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:

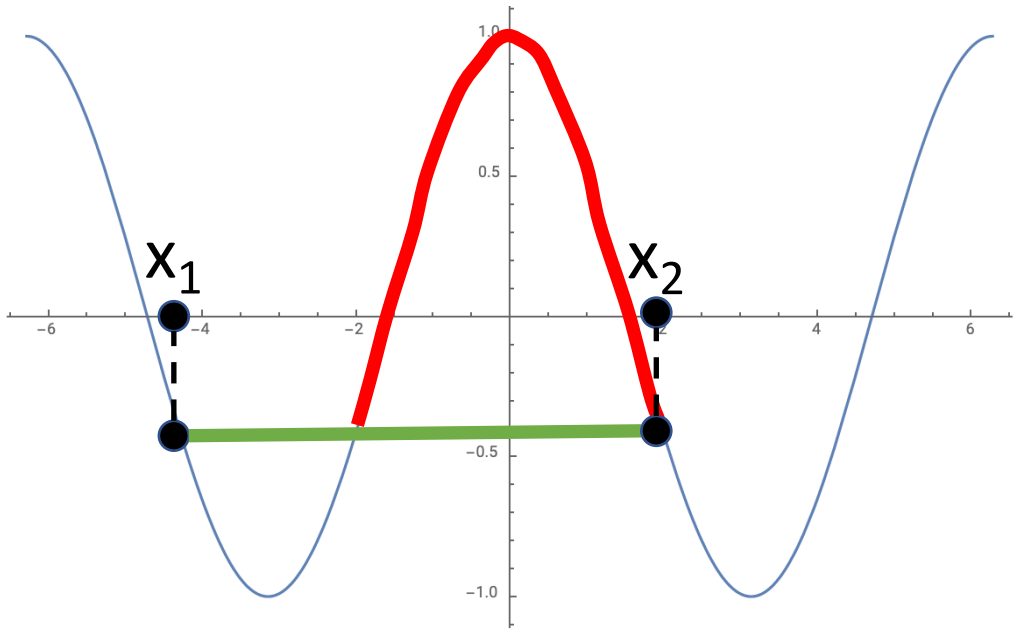


Convex Functions

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:

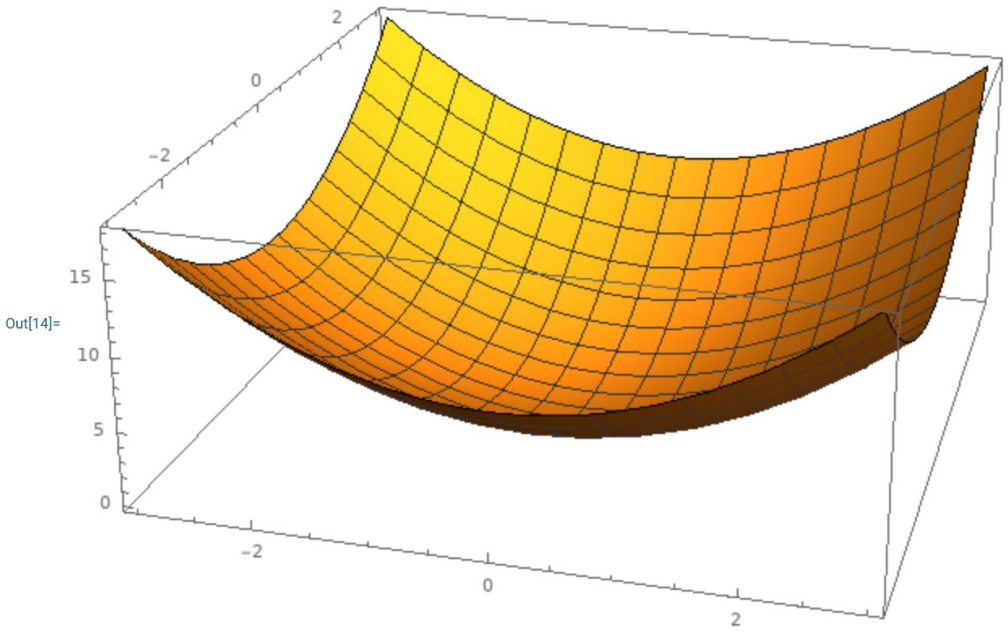


Convex Functions

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



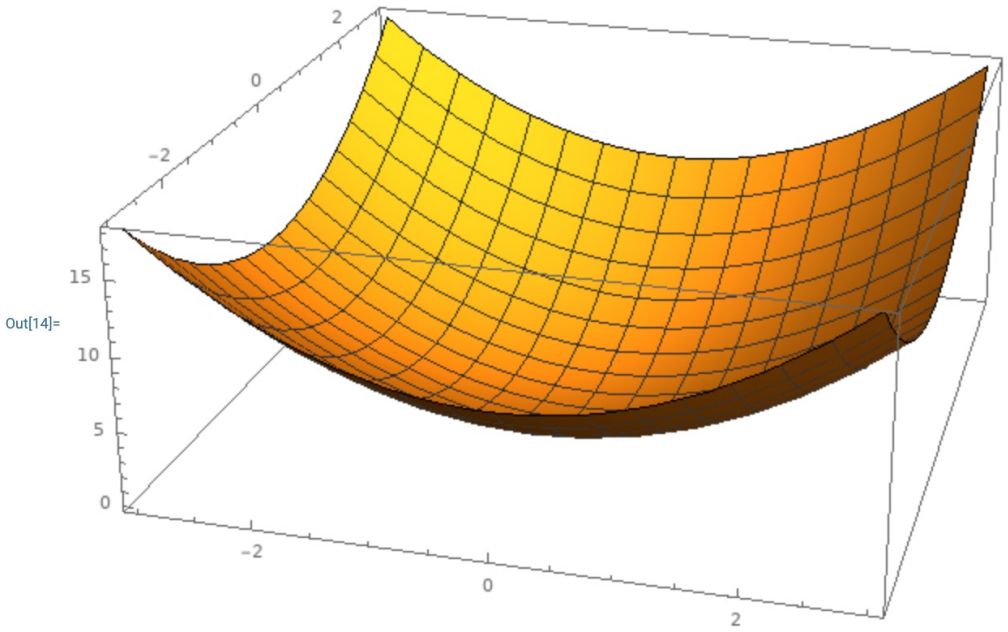
*Many technical details! See e.g. IOE 661 / MATH 663

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***

*Many technical details! See e.g. IOE 661 / MATH 663

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***

*Many technical details! See e.g. IOE 661 / MATH 663

Linear classifiers optimize a **convex function!**

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

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

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

R(W) = L2 or L1 regularization

Convex Functions

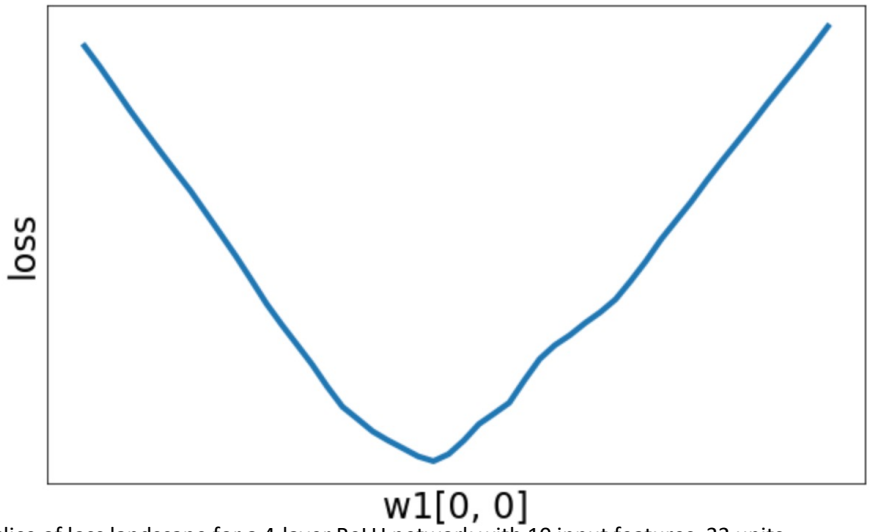
A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

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

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***

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

*Many technical details! See e.g. IOE 661 / MATH 663

Convex Functions

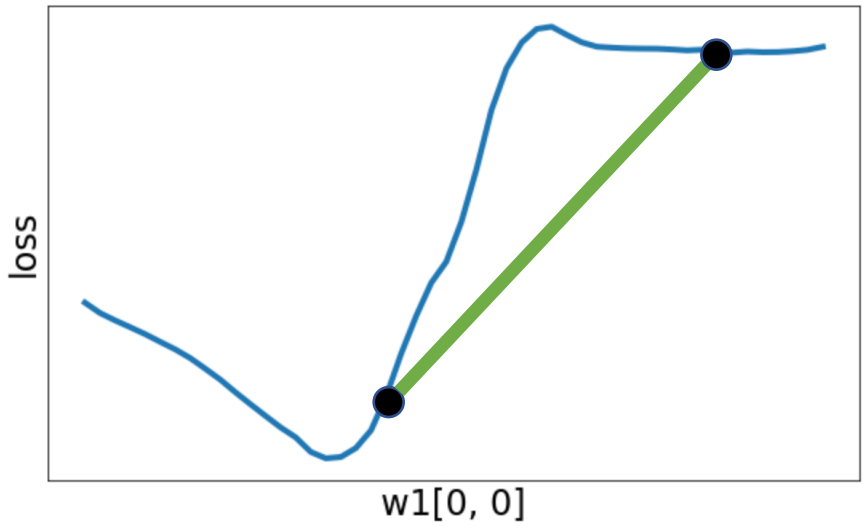
A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

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

Intuition: A convex function is a (multidimensional) bowl

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

But often clearly nonconvex:



1D slice of loss landscape for a 4-layer ReLU network with 10 input features, 32 units per hidden layer, 10 categories, with softmax loss

*Many technical details! See e.g. IOE 661 / MATH 663

Convex Functions

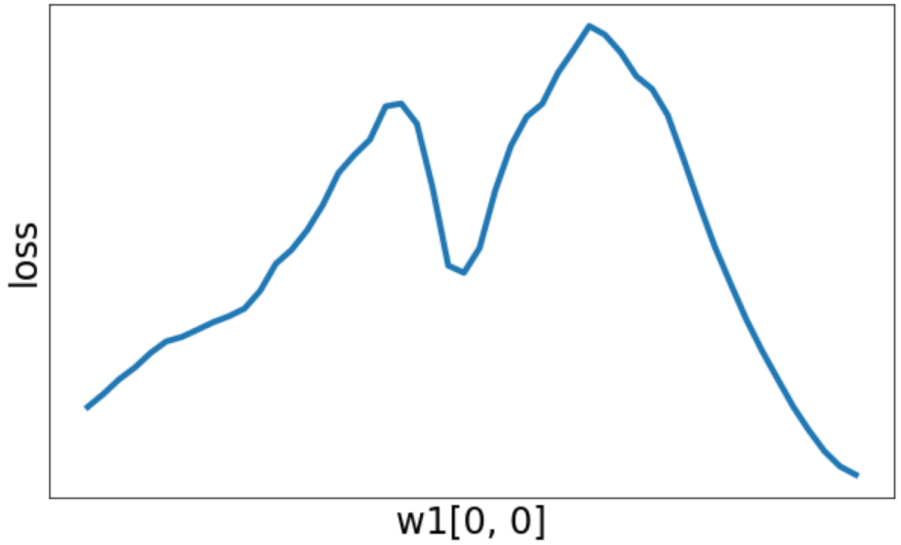
A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

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

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

*Many technical details! See e.g. IOE 661 / MATH 663

Convex Functions

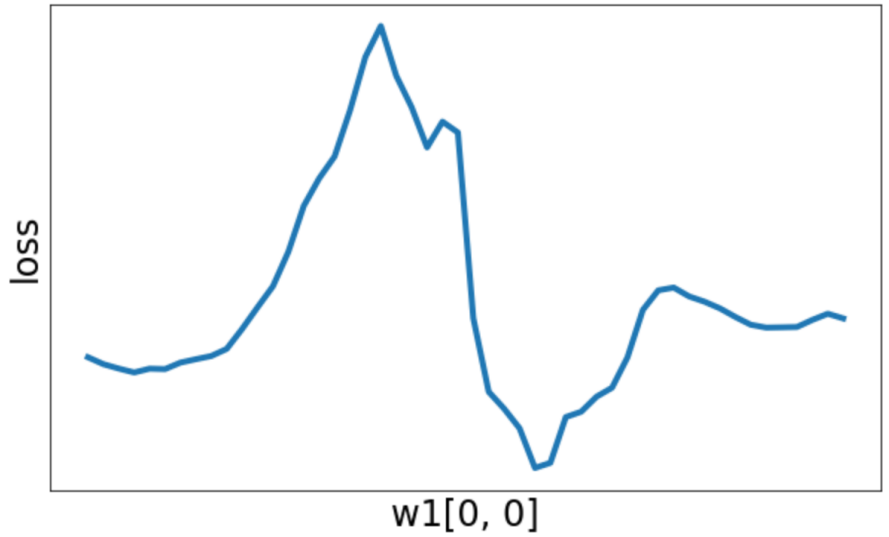
A function $f : X \subseteq \mathbb{R}^N \to \mathbb{R}$ is **convex** if for all $x_1, x_2 \in X, t \in [0, 1]$,

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

Intuition: A convex function is a (multidimensional) bowl

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

Can get very wild!



1D slice of loss landscape for a 4-layer ReLU network with 10 input features, 32 units per hidden layer, 10 categories, with softmax loss

*Many technical details! See e.g. IOE 661 / MATH 663

Convex Functions

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

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

Intuition: A convex function is a (multidimensional) bowl

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

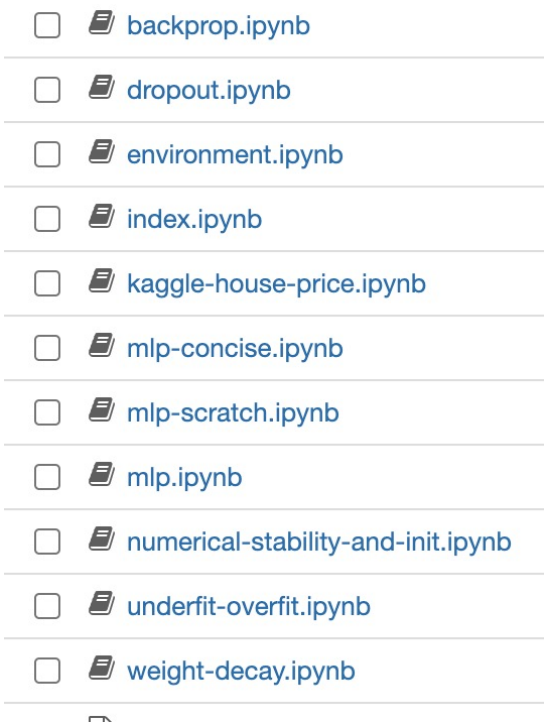
Most neural networks need **nonconvex optimization**

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

*Many technical details! See e.g. IOE 661 / MATH 663

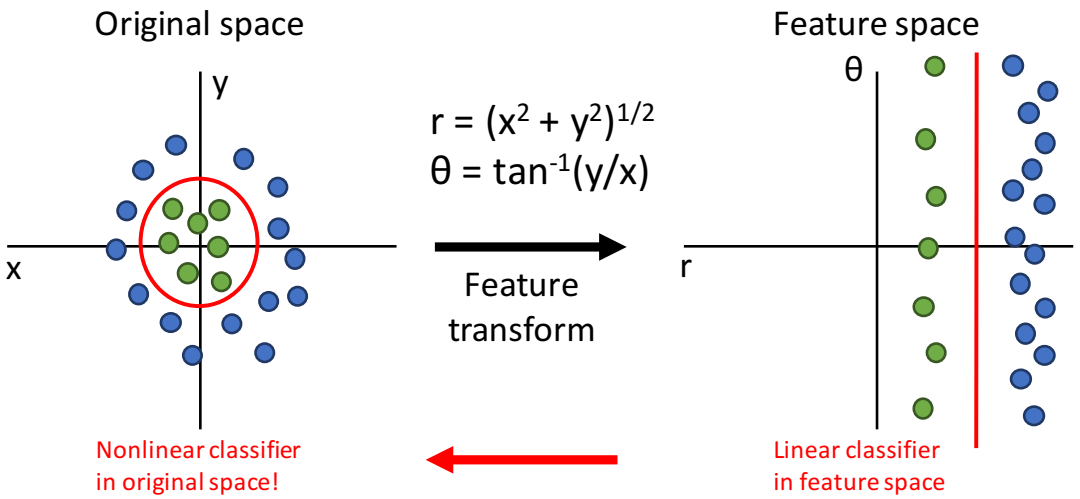
Reading and Practice

D2L textbook (https://d2l.ai/d2l-en.pdf) : Chapter 5 Multilayer Perceptrons: https://github.com/d2l-ai/d2l-pytorch-colab/tree/master/chapter_multilayer-perceptrons

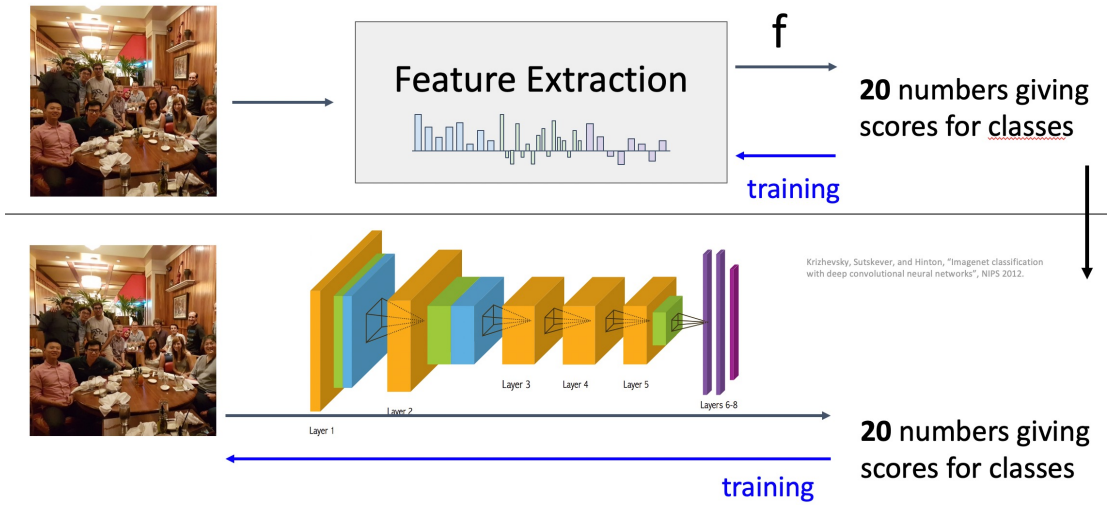


Summary

Feature transform + Linear classifier
allows nonlinear decision boundaries



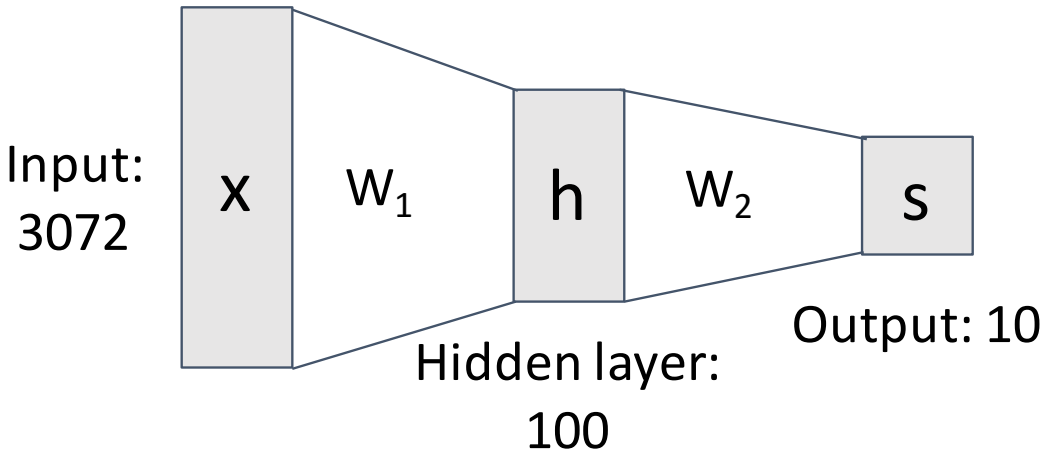
Neural Networks as learnable feature transforms



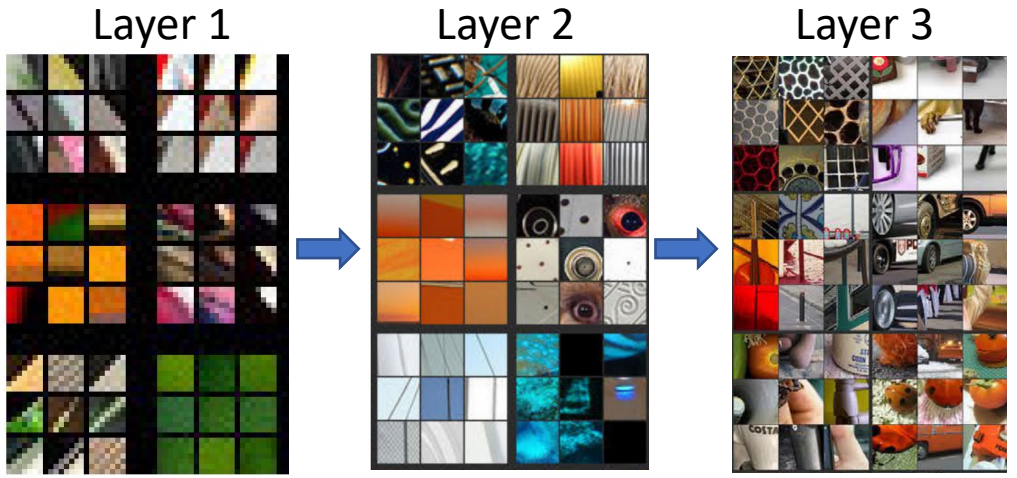
Summary

From linear classifiers to fully-connected networks

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



Neural networks: Hierarchical Compositional Features

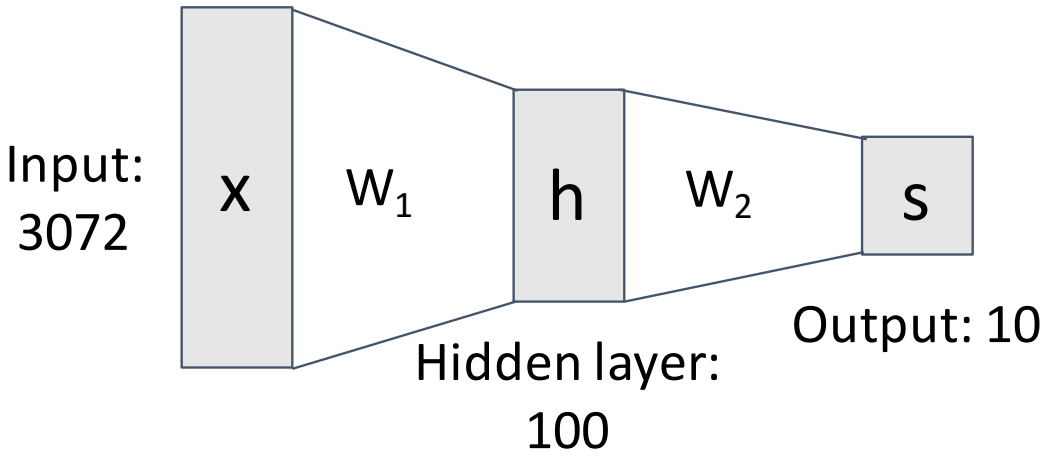


Zeiler and Fergus. ECCV'14

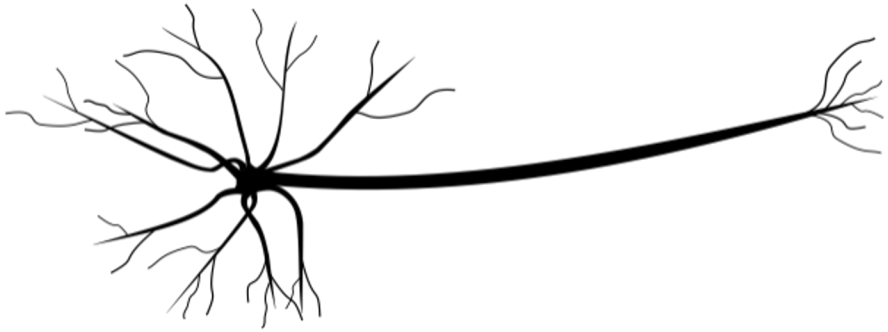
Summary

From linear classifiers to fully-connected networks

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



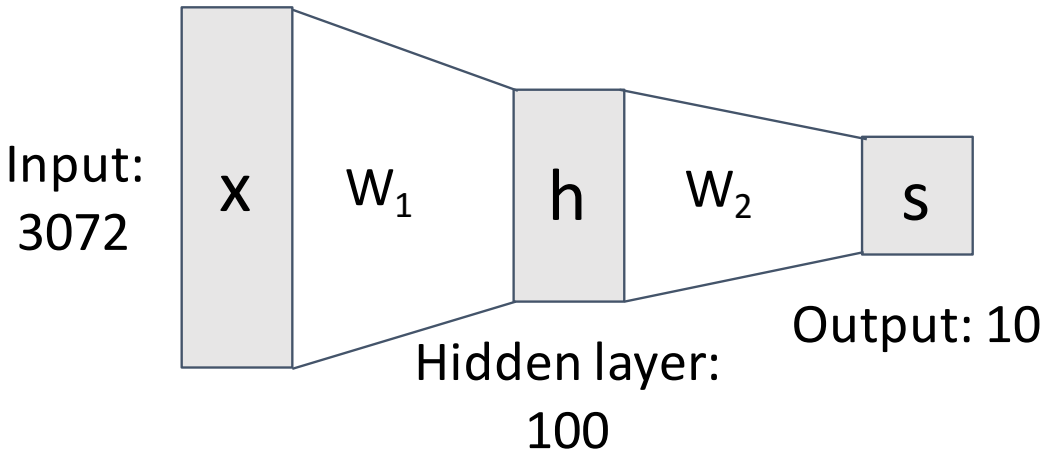
Neural networks are loosely inspired by biological neurons but be careful with analogies



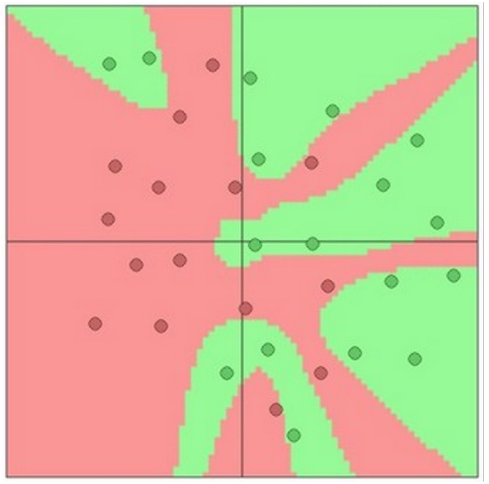
Summary

From linear classifiers to fully-connected networks

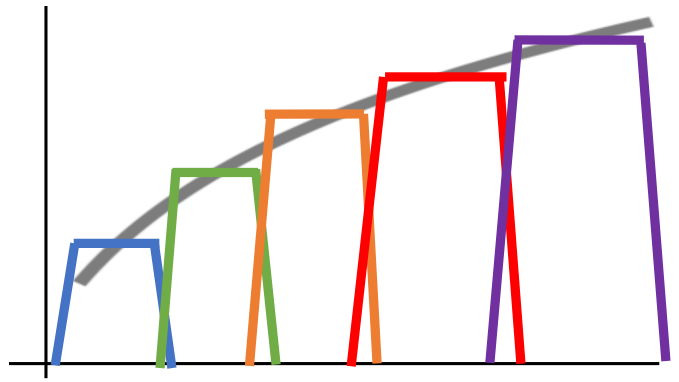
$$f(x) = W_2 \max(0, W_1 x + b_1) + b_2$$



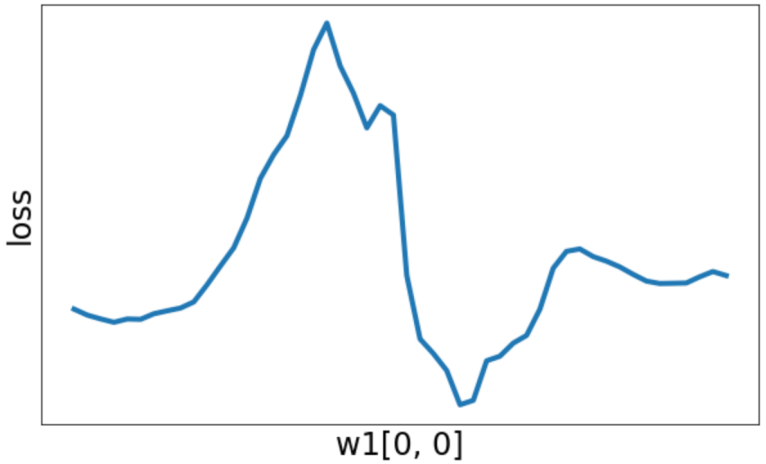
Space Warping



Universal Approximation



Nonconvex



Problem: How to compute gradients?

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

$R(W) = \sum_k W_k^2$ 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 $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial b_2}$ then we can optimize with SGD

Next time: Backpropagation