Lecture 16: (Over/Under)Fitting Neural Networks

Administrative: COVID19

As of 3/9 there are no cases reported in Michigan.

But we should still be cautious:

- **Stay home if you feel sick.**
- **Wash your hands frequently.**
- **Avoid touching your face.**
- **Avoid large gatherings.**

Administrative: COVID19

- **Lectures are recorded:** You are encouraged to watch from home.
- **Project group size**: Previously we said group size 3-5. We will now allow groups of size 1-5 so you can work alone if you prefer.
- **Office Hours**: We will start experimenting with virtual office hours this week. Stay tuned on Piazza for details.
- **Poster Session** is cancelled. We will ask you to submit a video describing your project instead; details to follow.

Administrative: Project Proposal

- Project proposal is due **tomorrow**, **3/11 11:59pm**
- 2 page writeup in CVPR format
- You should answer the following:
  - Who are you working with? Groups of 1-5
  - What problem are you trying to solve? Applying vision to an interesting application? Re-implement a paper other other exiting method? Try to implement some new idea? All are fine!
  - How are you going to try and solve the problem? You don't need to have all details figured out, but you should have an idea of how you'll approach it
  - What do you need to attack this problem? Datasets, code, computing resources? What is your plan for getting access to these things?
  - How will you measure success? What evaluation metrics will you use?

Administrative: Homework 4

- Homework 4 was released yesterday;
  will be due **Friday 3/20, 11:59pm**
- We should cover everything you need for this assignment by Thursday’s lecture

Model Complexity
Underfitting / Overfitting

(Over/Under)fitting and Complexity

Let's fit a polynomial: given x, predict y

$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \cdots + w_F x^F$$

Note: can do non-linear regression with copies of x

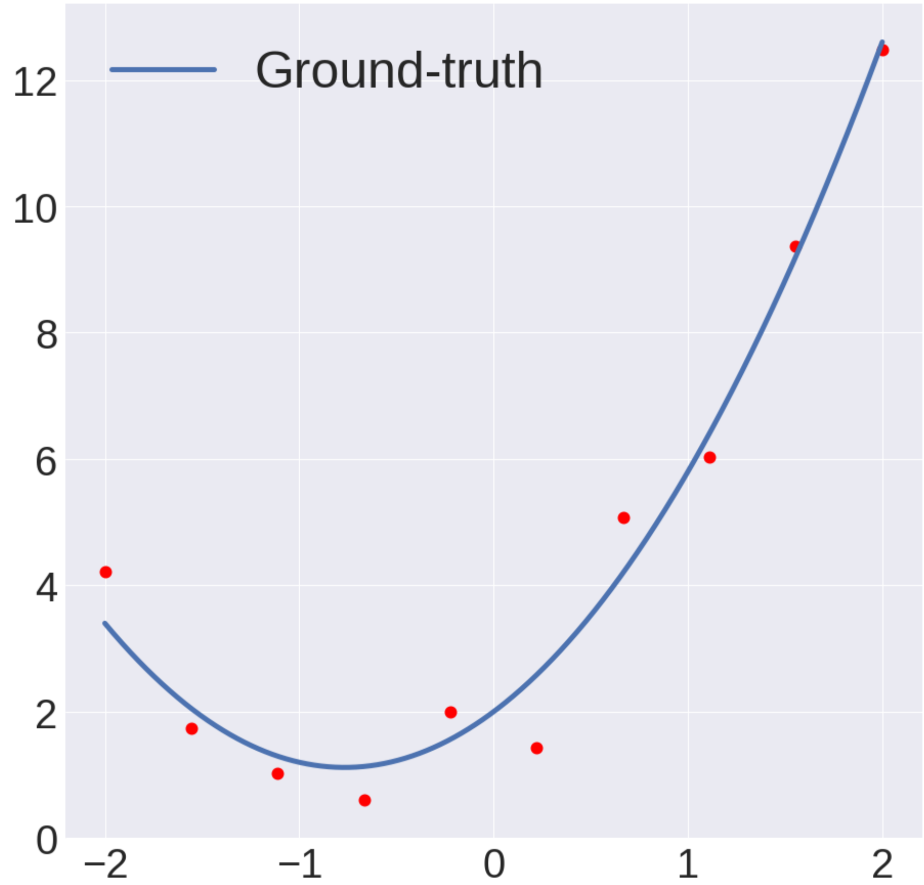
$$\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} x_1^F & \cdots & x_1^2 & x_1 & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ x_N^F & \cdots & x_N^2 & x_N & 1 \end{bmatrix} \begin{bmatrix} w_F \\ \vdots \\ w_2 \\ w_1 \\ w_0 \end{bmatrix}$$

Matrix of all polynomial degrees

Weights: one per polynomial degree

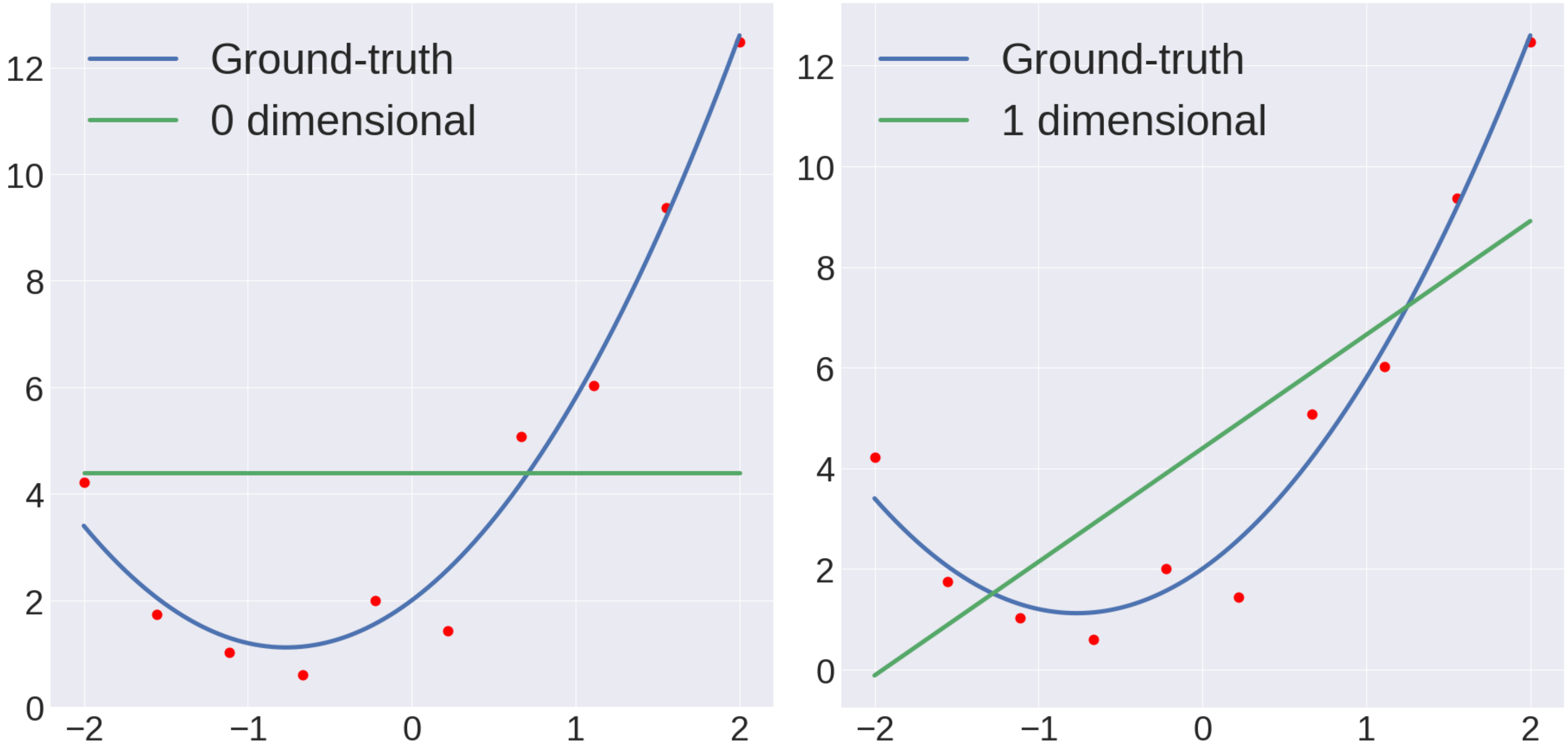
(Over/Under)fitting and Complexity

Ground-Truth: $1.5x^2 + 2.3x+2 + N(0,0.5)$



Underfitting

Ground-Truth: $1.5x^2 + 2.3x+2 + N(0,0.5)$

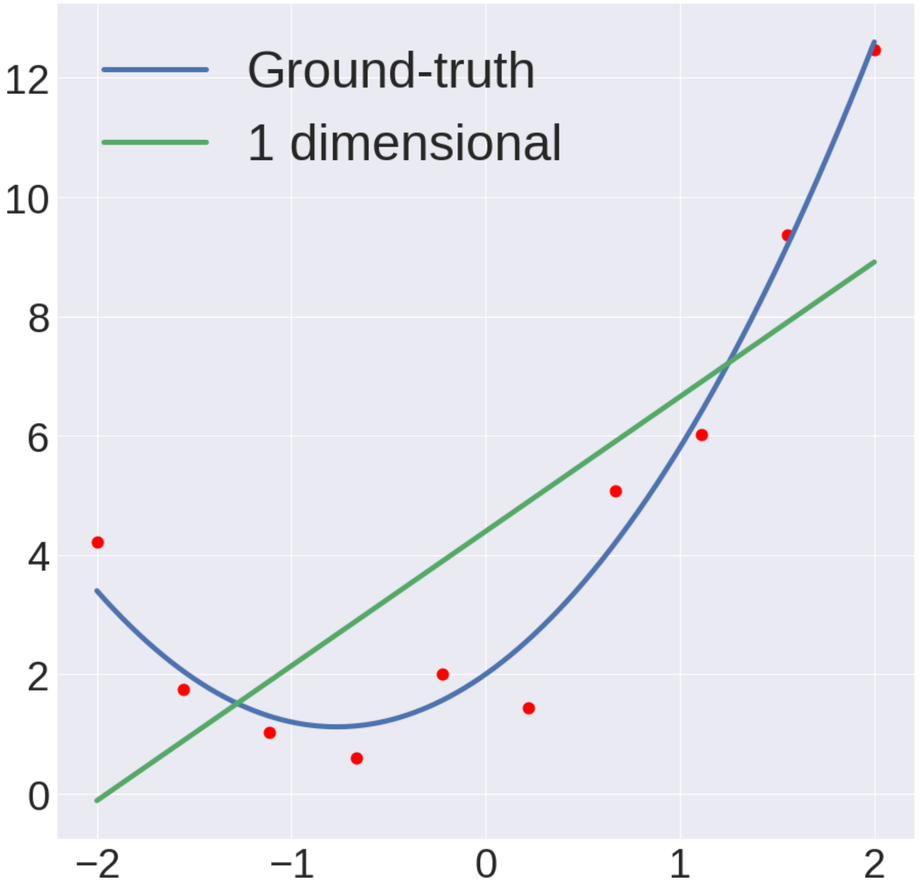


Underfitting

Ground-Truth: $1.5x^2 + 2.3x+2 + N(0,0.5)$

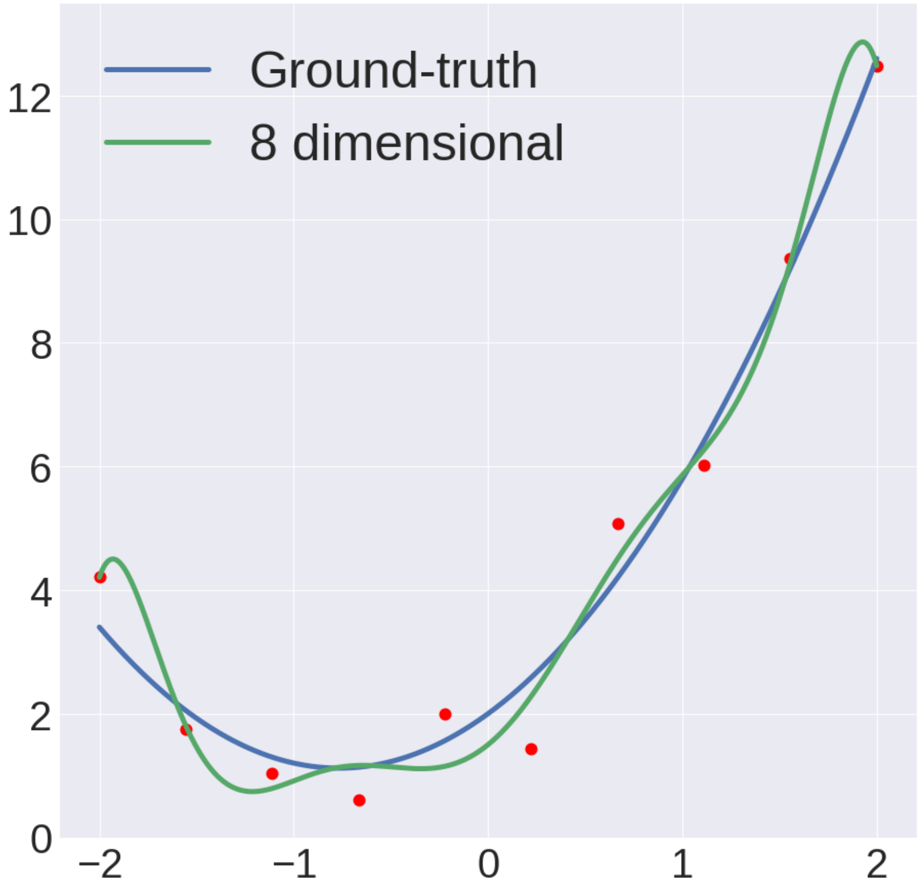
Model isn't "complex" enough to fit the data

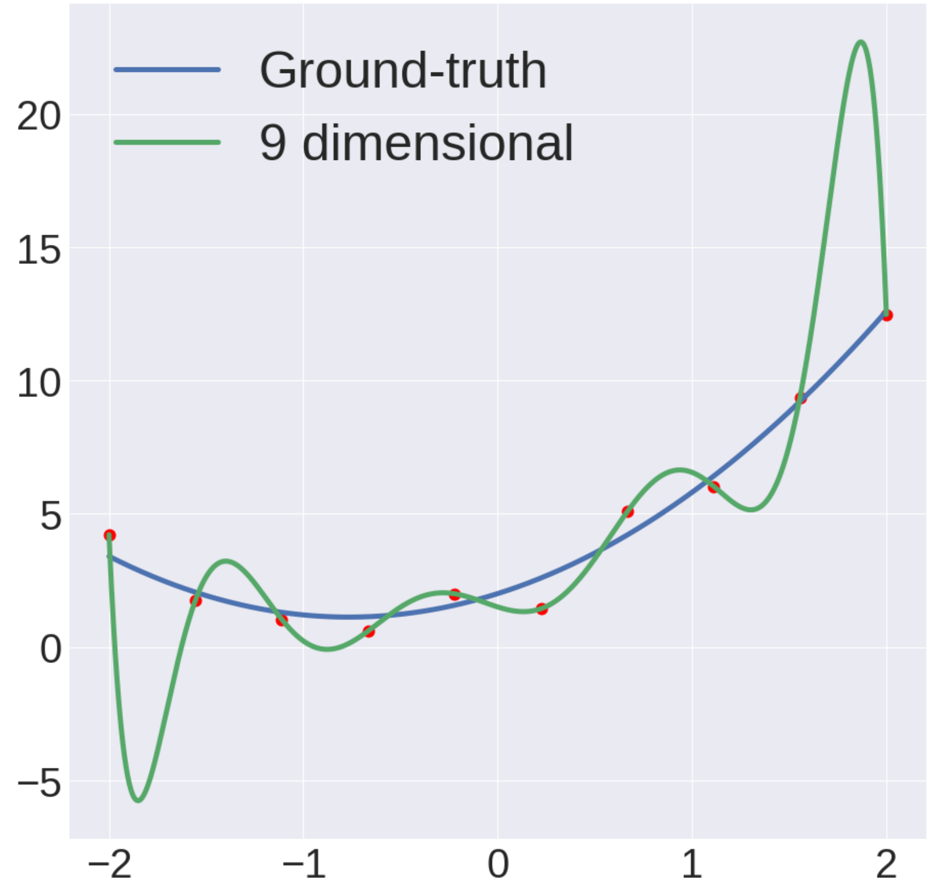
Bias (statistics): Error intrinsic to the model.



Overfitting

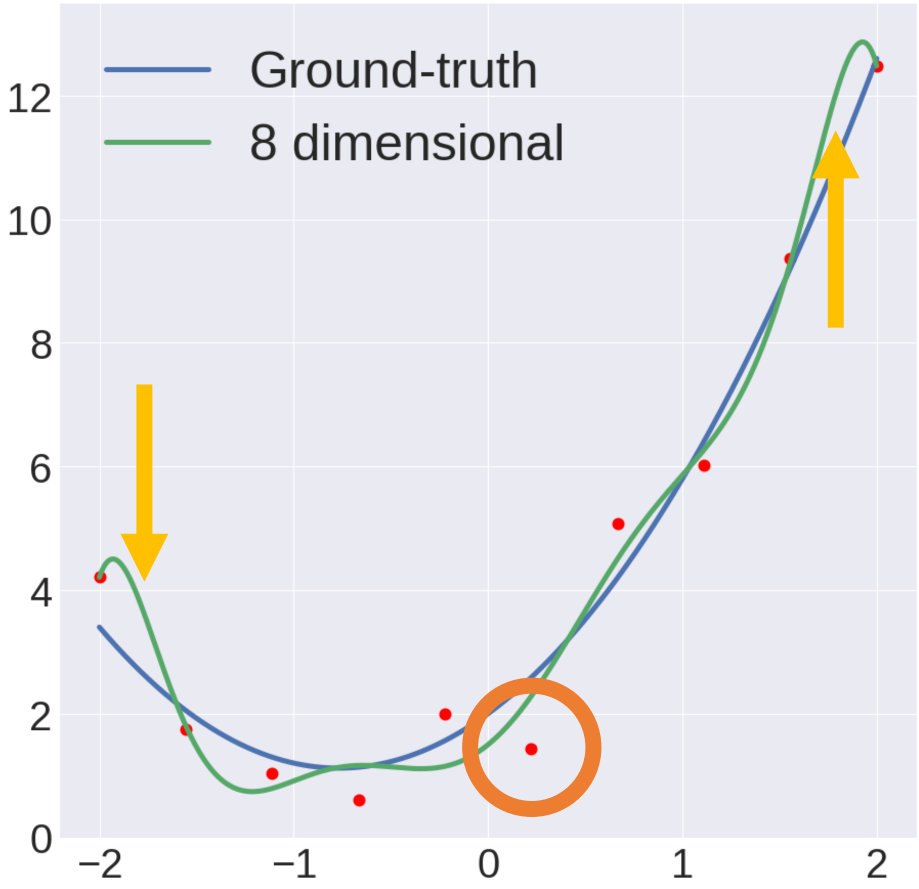
Ground-Truth: $1.5x^2 + 2.3x+2 + N(0,0.5)$

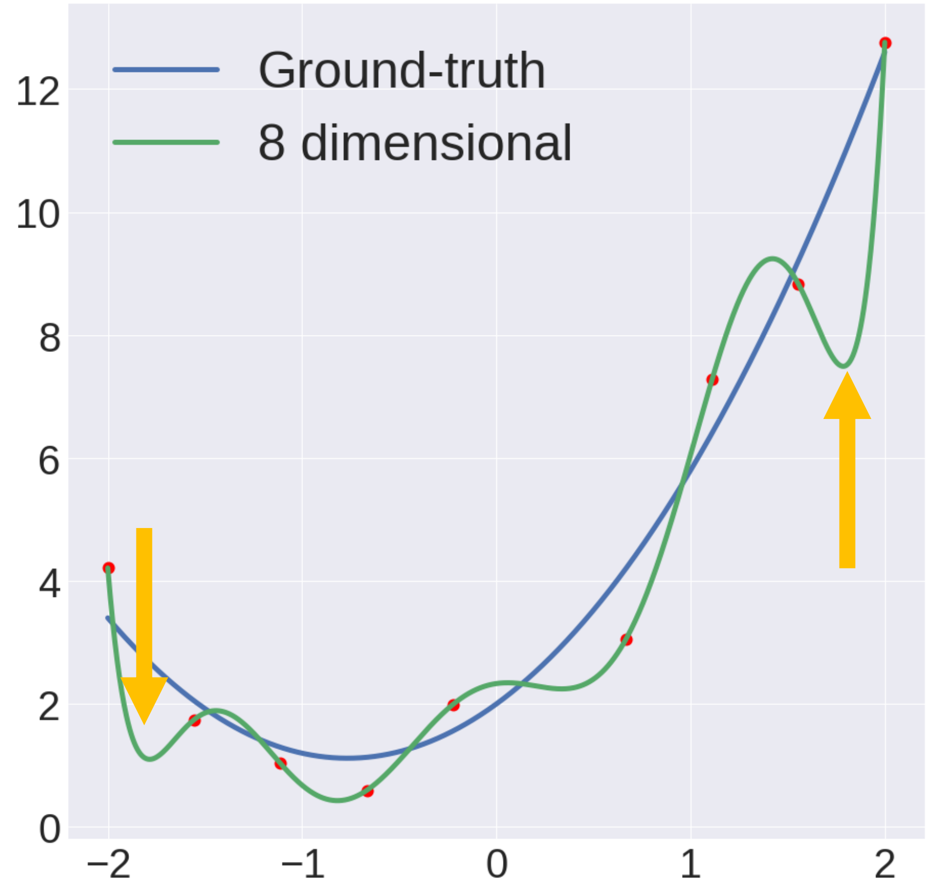




Overfitting

Model has high ***variance***: remove **one point**, and model changes dramatically





(Continuous) Model Complexity

$$\arg\min_{W} \lambda \|W\|_2^2 + \underbrace{\sum_{i=1}^{n} -\log\left(\frac{\exp((Wx)_{y_i})}{\sum_k \exp((Wx)_k)}\right)}$$

Regularization: penalty for complex model

Pay penalty for negative log-likelihood of correct class

Intuitively: big weights = more complex model

Model 1: $0.01*x_1 + 1.3*x_2 + -0.02*x_3 + -2.1x_4 + 10$

Model 2: $37.2*x_1 + 13.4*x_2 + 5.6*x_3 + -6.1x_4 + 30$

Fitting a Model

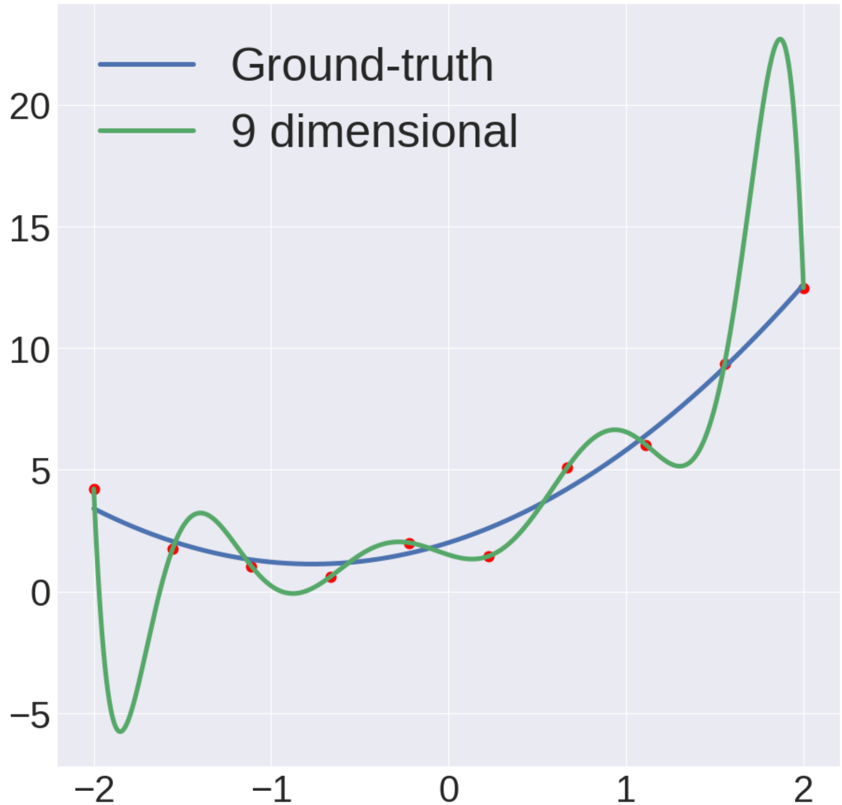
Again, fitting polynomial, but with regularization

$$\arg\min_{\mathrm{w}} ||\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}|| + \lambda ||\boldsymbol{w}||$$

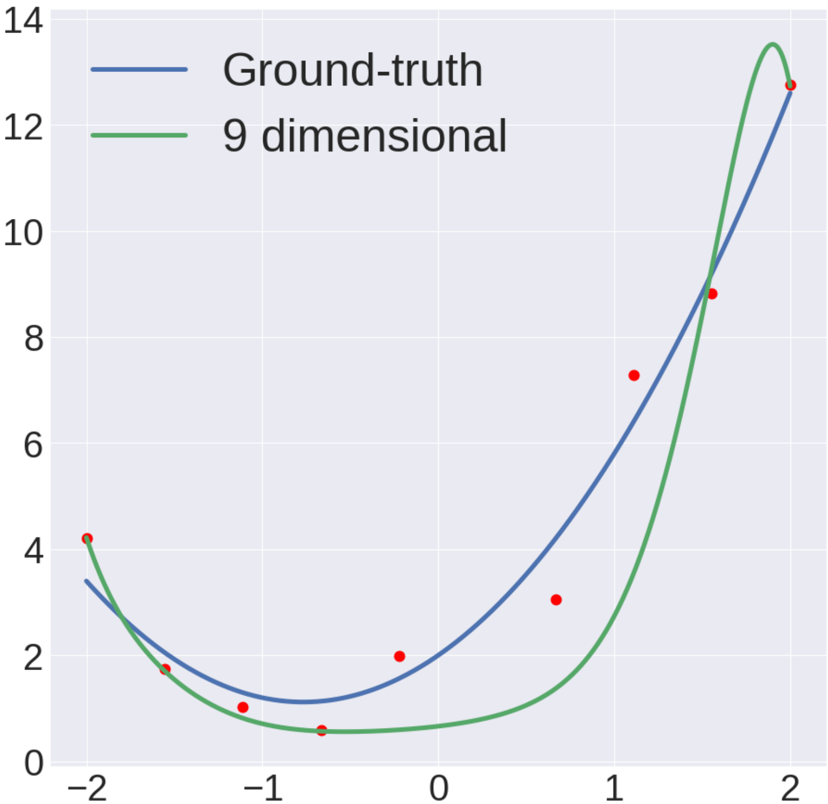
$$\begin{bmatrix} x_1^F & \cdots & x_1^2 & x_1 & 1 \\ \vdots & \ddots & \vdots & \vdots & \vdots \\ x_N^F & \cdots & x_N^2 & x_N & 1 \end{bmatrix} \begin{bmatrix} w_F \\ \vdots \\ w_0 \end{bmatrix}$$

Adding Regularization

No regularization:
fits all data points



Regularization:
can't fit all data points



Bias / Variance Tradeoff

Error on new data comes from combination of:

1. **Bias**: model is oversimplified and can't fit the underlying data
2. **Variance**: you don't have the ability to estimate your model from limited data
3. **Inherent**: the data is intrinsically difficult

Bias and variance trade-off. Fixing one hurts the other. You can prove theorems about this.

Underfitting and Overfitting

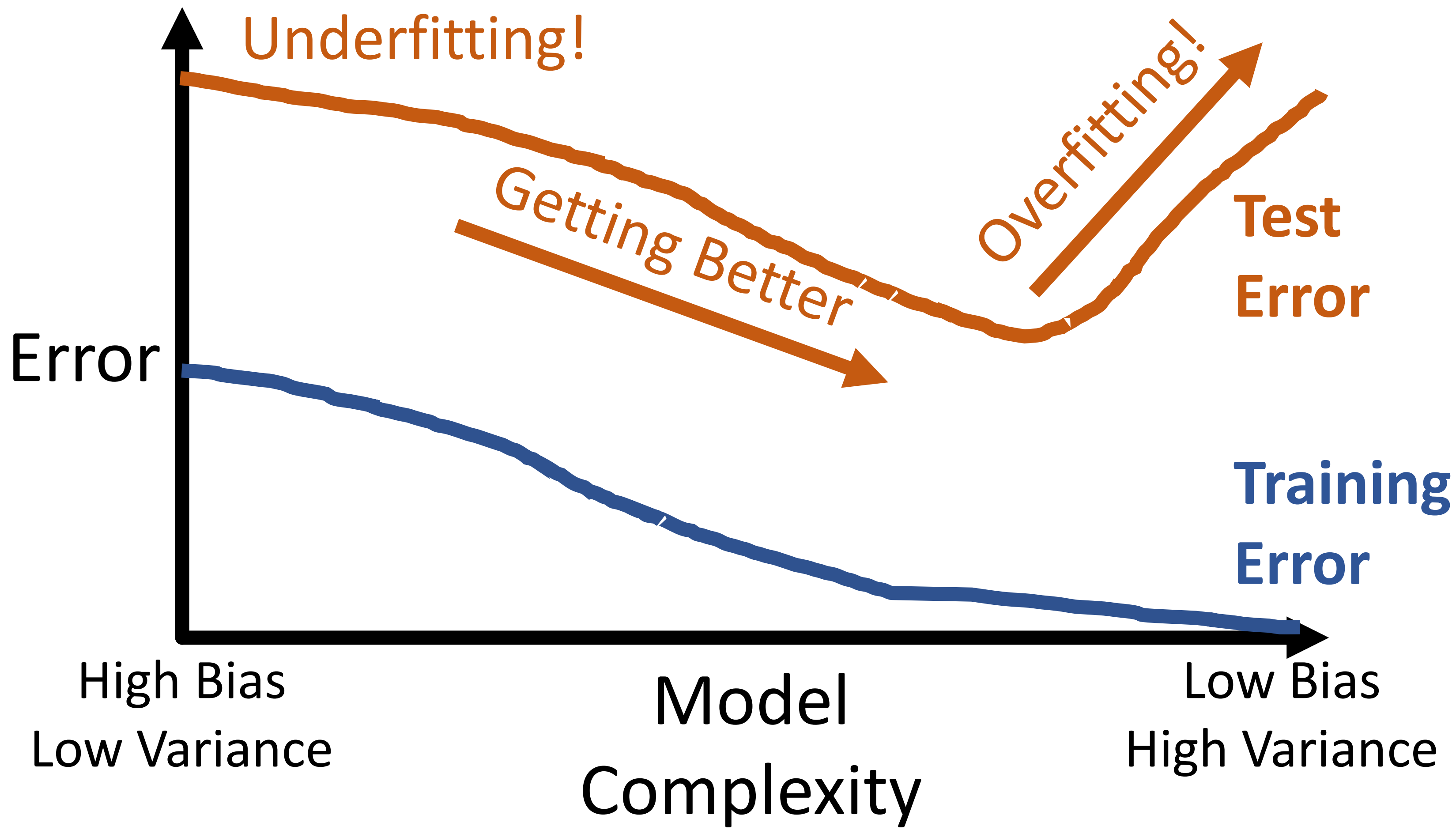


Diagram adapted from: D. Hoiem

Underfitting and Overfitting

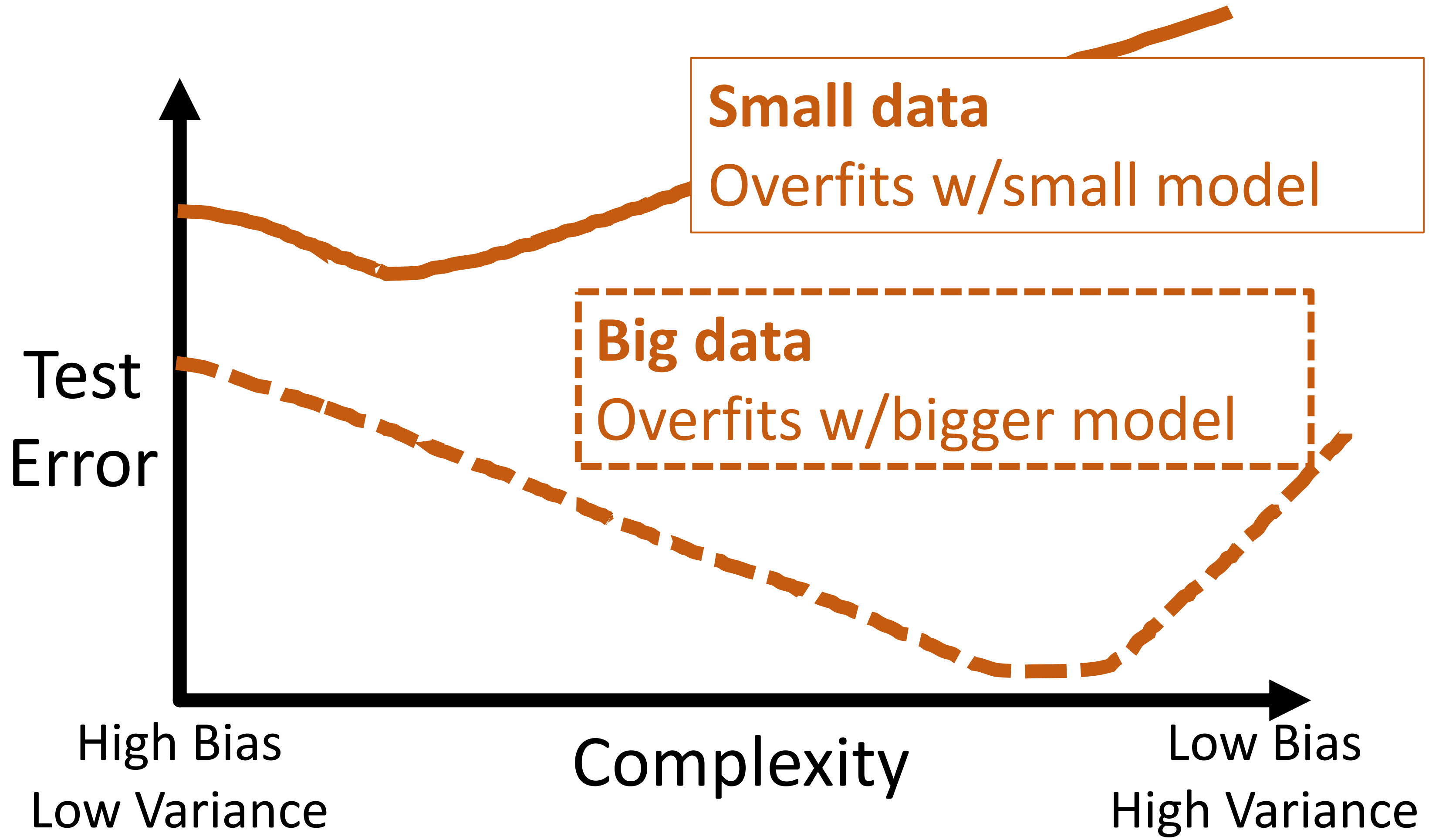
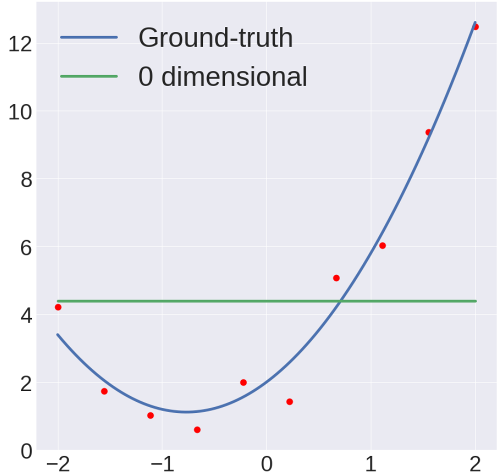
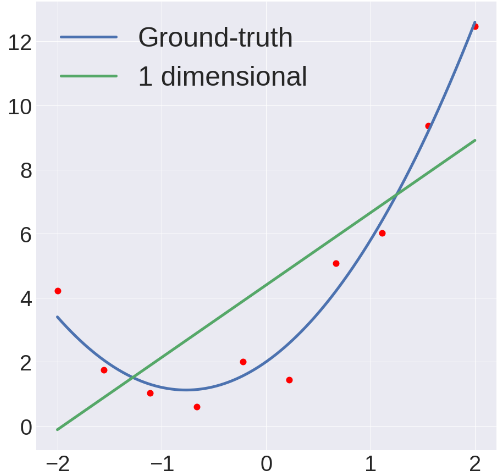


Diagram adapted from: D. Hoiem

Underfitting



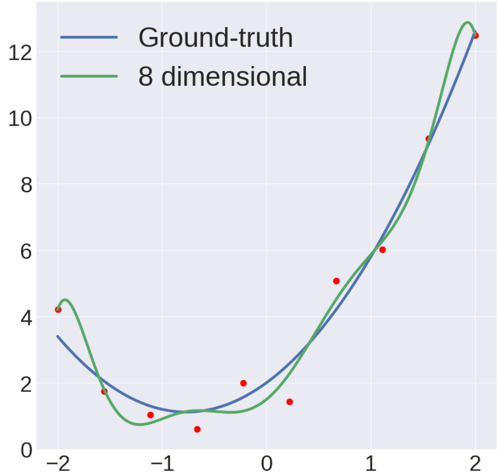


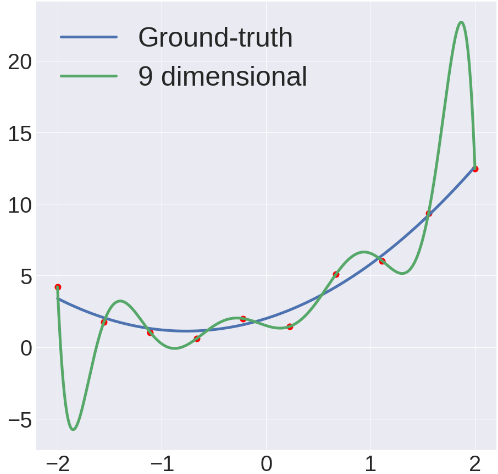
Do poorly on both training and validation data due to bias.

Solution:

1. More features
2. More powerful model
3. Reduce regularization

Overfitting





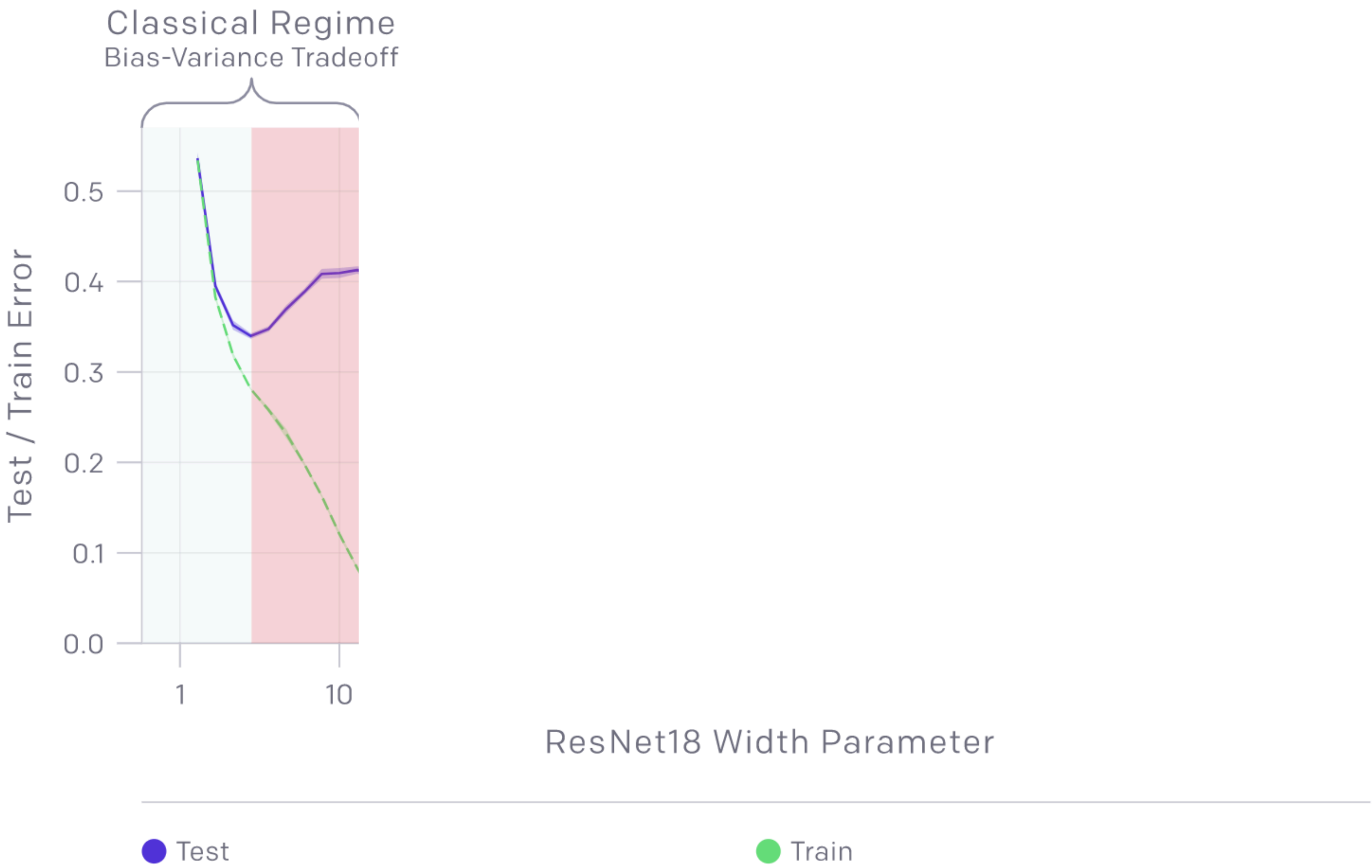
Do well on training data, but poorly on validation data due to variance

Solution:

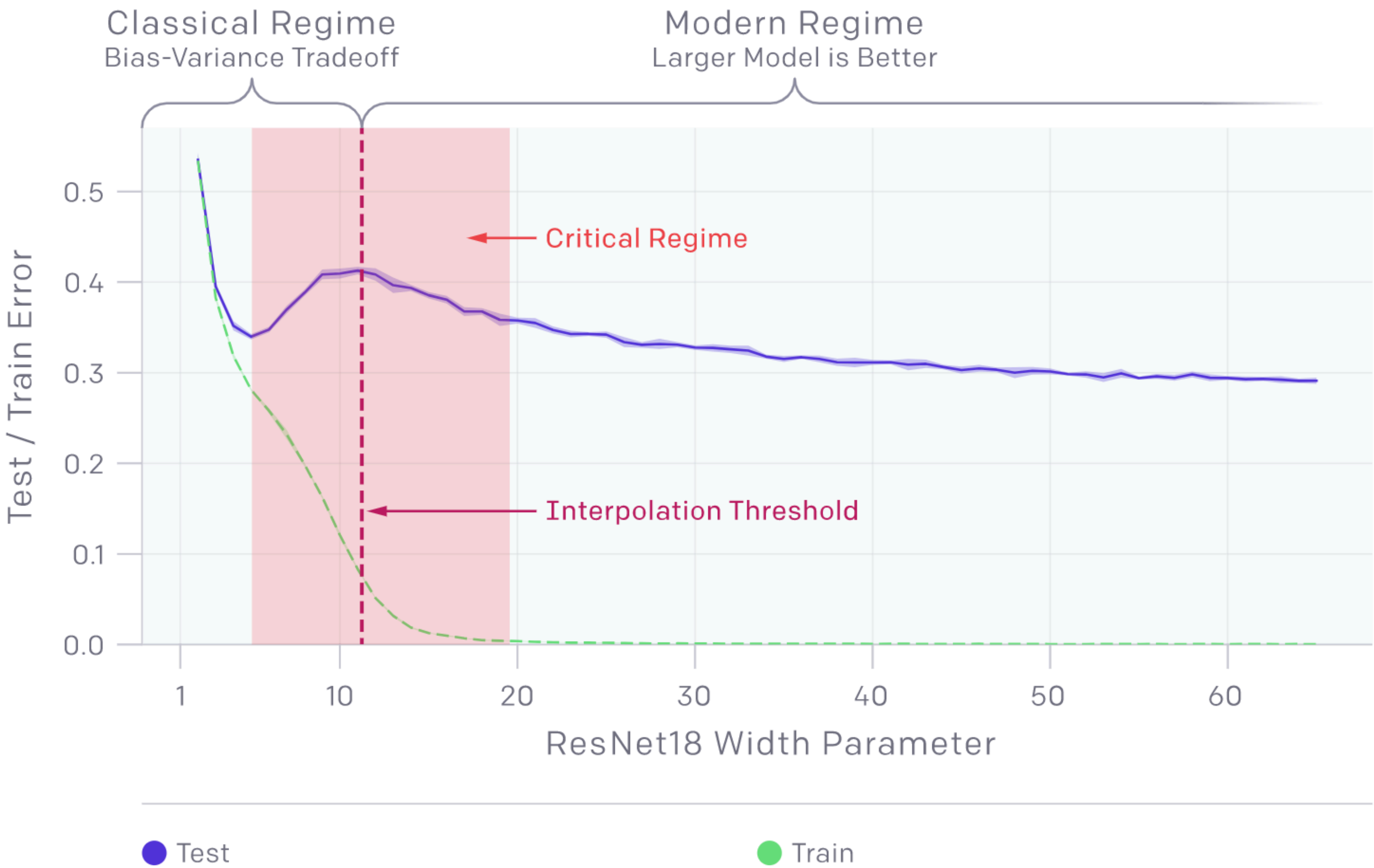
1. More data
2. Less powerful model
3. Regularize your model more

Heuristic: First make sure you *can* overfit, then stop overfitting.

Double Descent



Double Descent



Advani and Saxe, “High-dimensional dynamics of generalization error in neural networks”, 2017

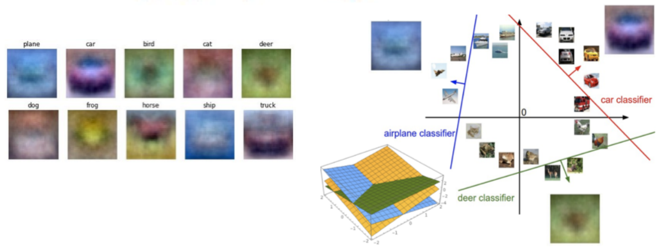
Geiger et al, “The jamming transition as a paradigm to understand the loss landscape of deep neural networks”, 2018

Belkin et al, “Reconciling modern machine learning practice and the bias-variance trade-off”, 2018

Nakkiran et al, “Deep Double Descent: Where Bigger Models and More Data Hurt”, 2019

Where we are:

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



1. Use **Linear Models** for image classification problems

2. Use **Loss Functions** to express preferences over different choices of weights

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

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

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

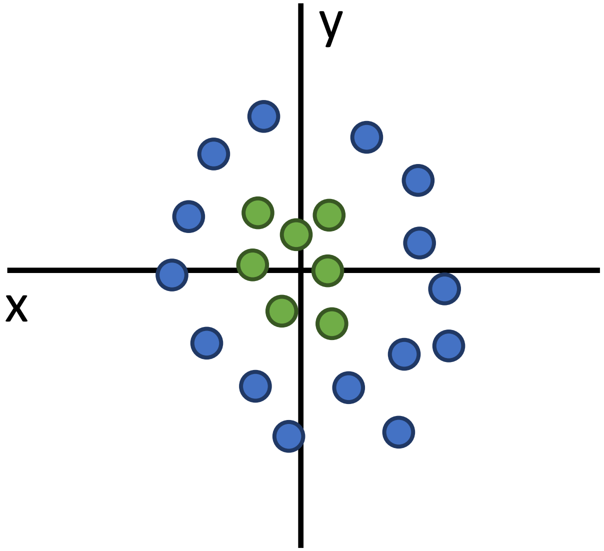
3. Use **Stochastic Gradient Descent** to minimize our loss functions and train the model

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

4. Add **Regularization** to control overfitting

Problem: Linear Classifiers not enough

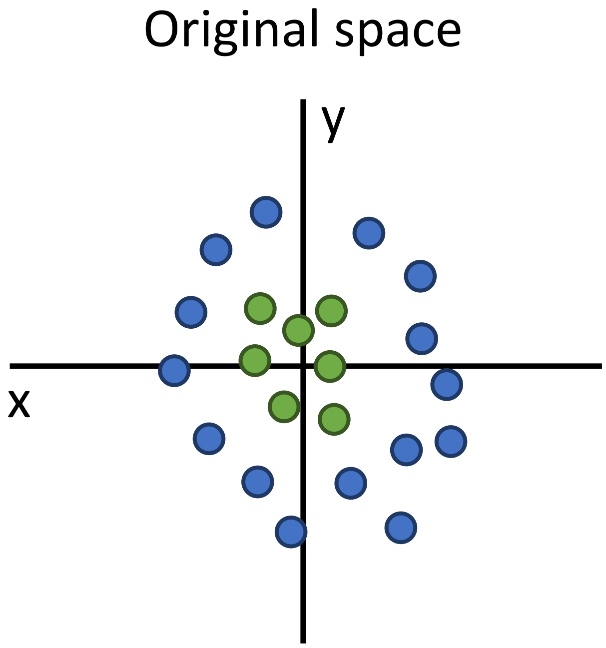
Geometric Viewpoint



Visual Viewpoint

One template per class:
Can't recognize different
modes of a class

One solution: **Feature Transforms**

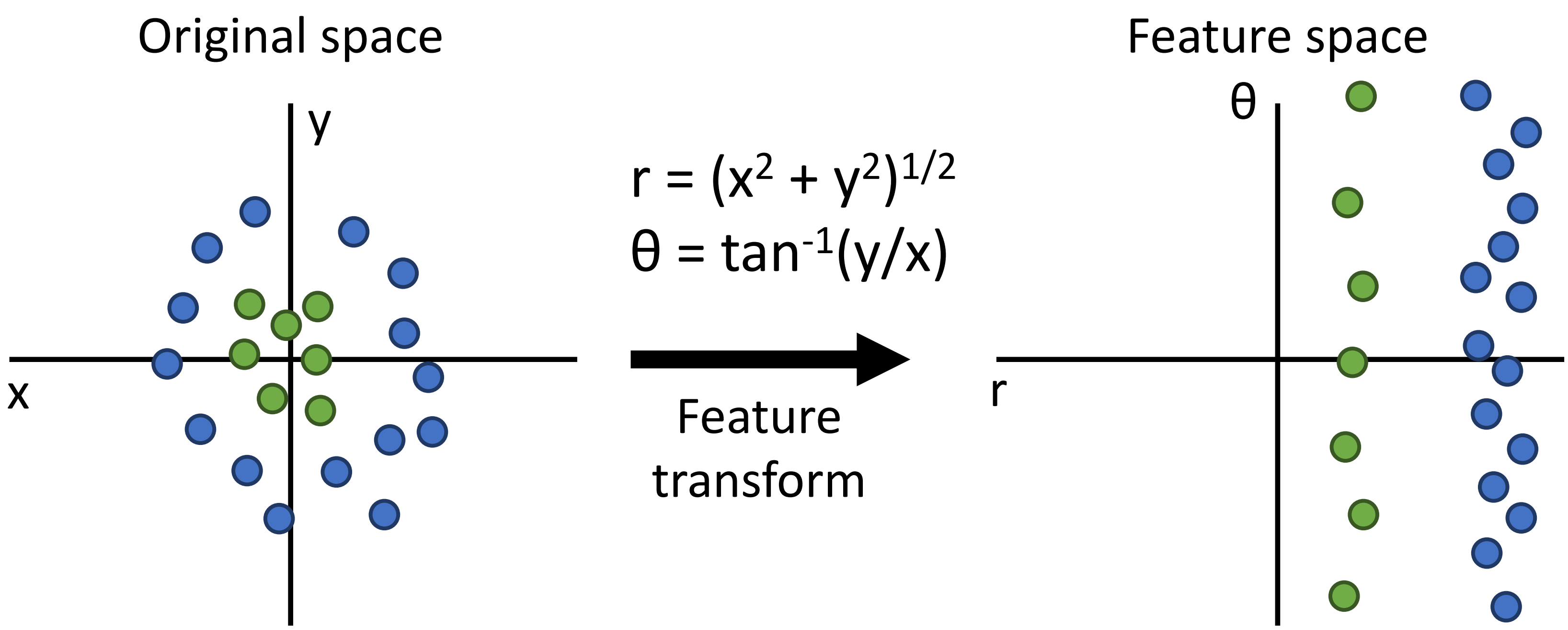


$r = (x^2 + y^2)^{1/2}$

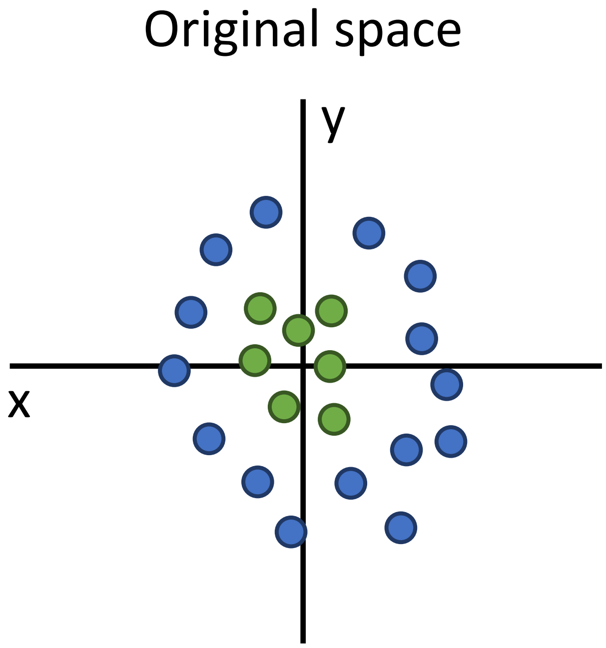
$\theta = \tan^{-1}(y/x)$

Feature transform

One solution: **Feature Transforms**



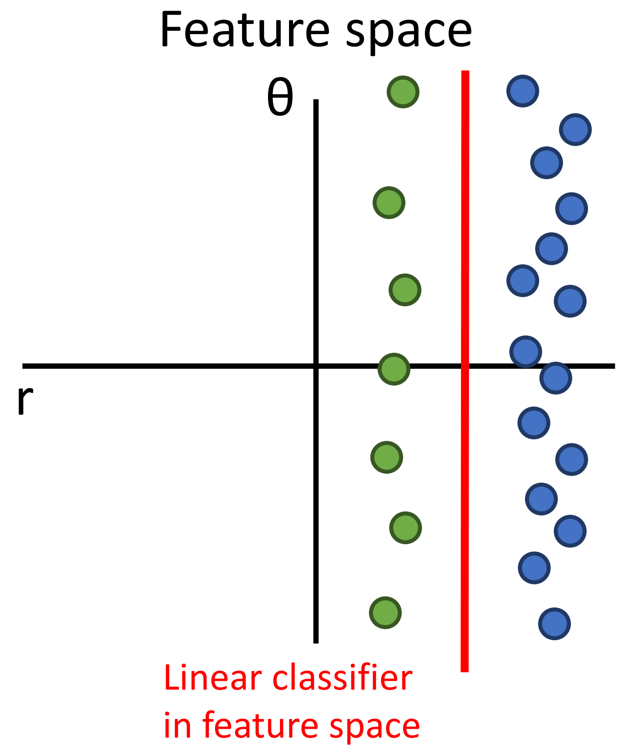
One solution: **Feature Transforms**



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

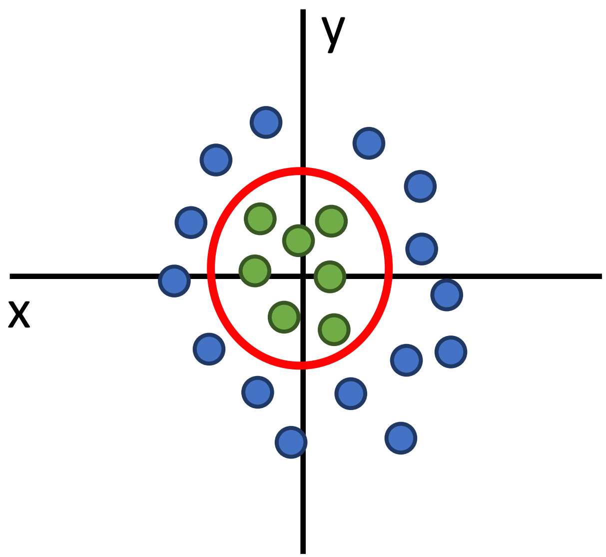
$\theta = \tan^{-1}(y/x)$

Feature transform



One solution: **Feature Transforms**

Original space

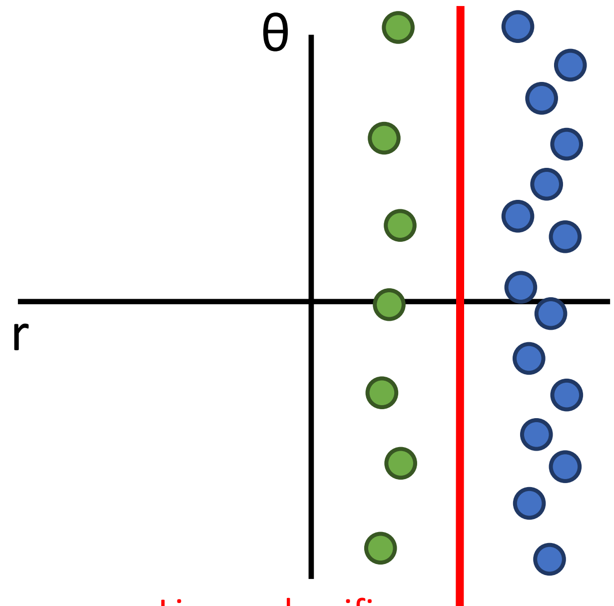


$r = (x^2 + y^2)^{1/2}$

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

Feature transform

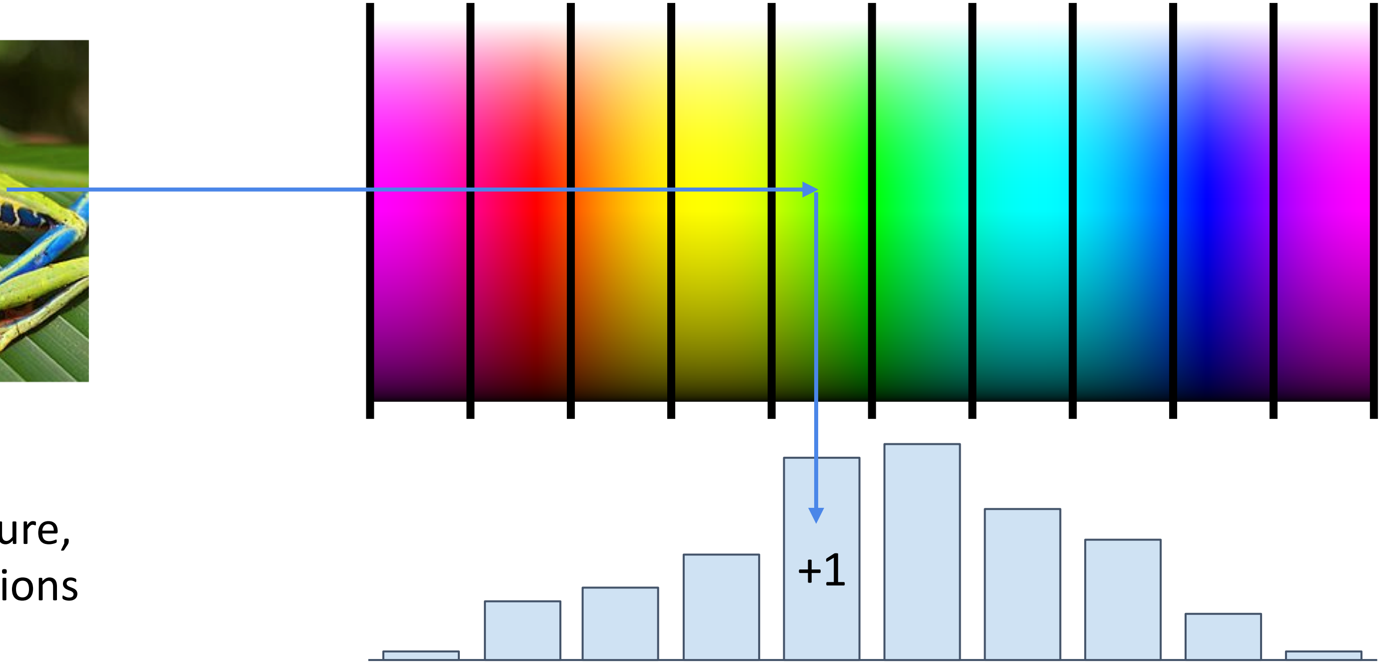
Feature space



Nonlinear classifier in original space!

Linear classifier in feature space

Image Features: Color Histogram



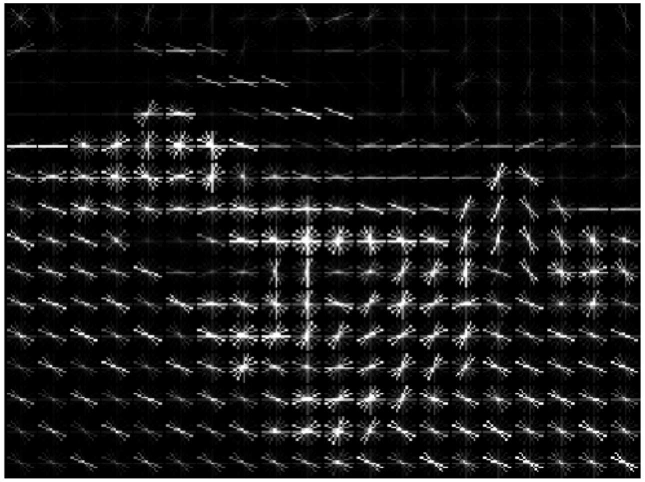
Ignores texture,
spatial positions

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)

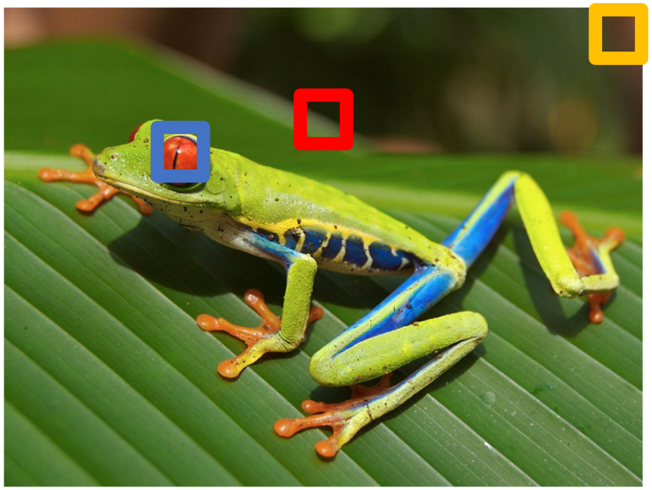


1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

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

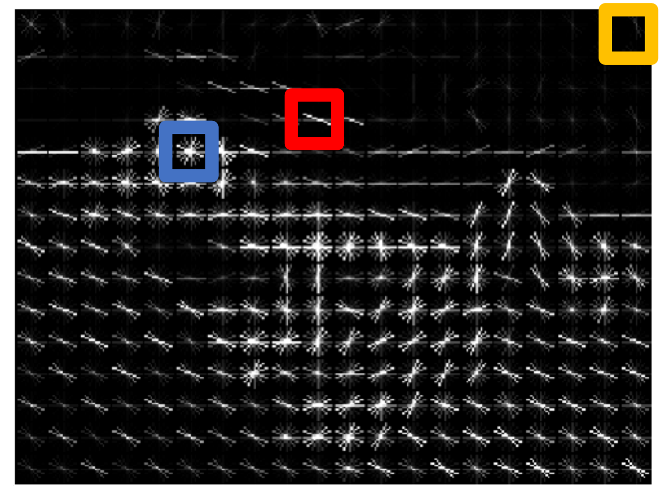
Image Features: Histogram of Oriented Gradients (HoG)



Weak edges

Strong diagonal edges

Edges in all directions



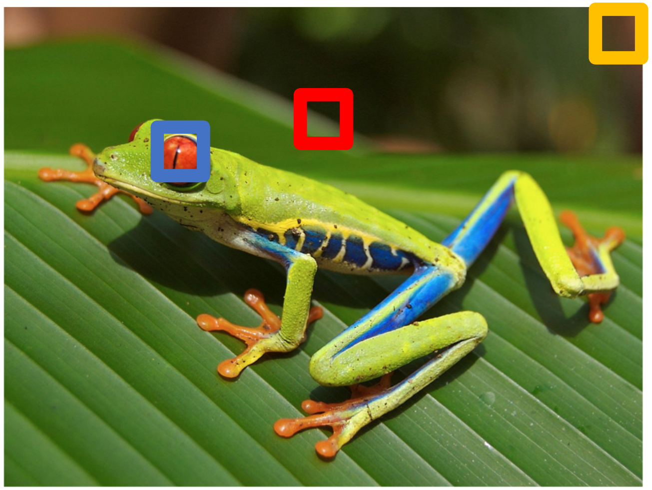
1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

Example: 320x240 image gets divided into 40x30 bins; 8 directions per bin; 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)

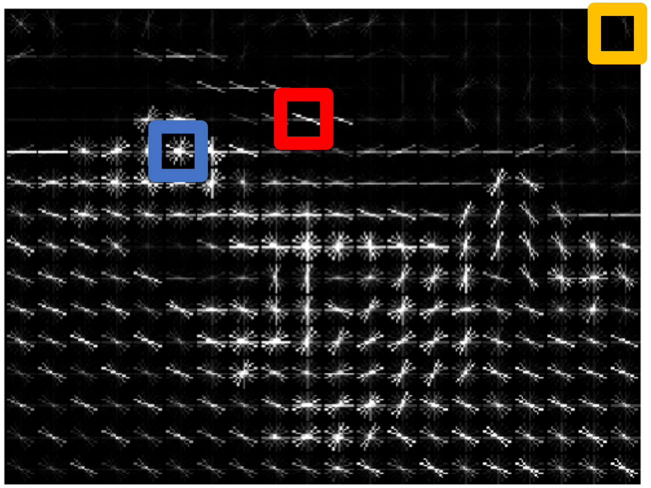
Captures texture and position, robust to small image changes



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

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

Image Features: Bag of Words

Learn a feature transform from data!

Step 1: Build codebook

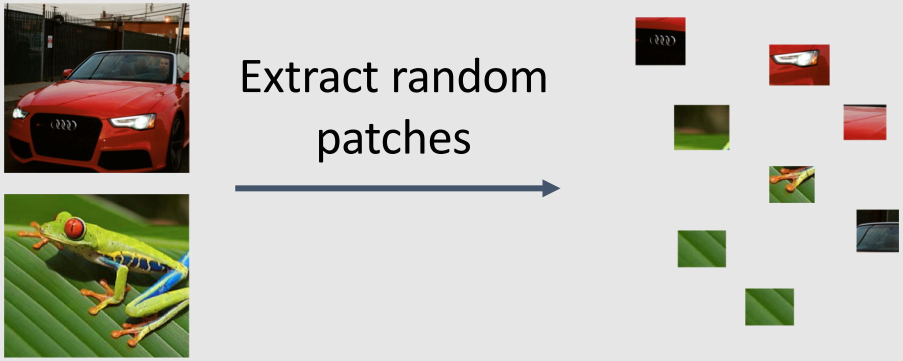
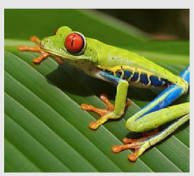


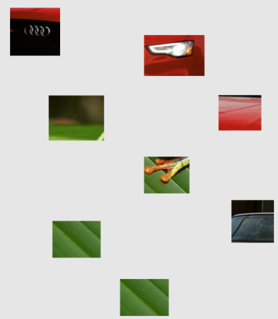
Image Features: Bag of Words

Learn a feature transform from data!

Step 1: Build codebook



Extract random patches



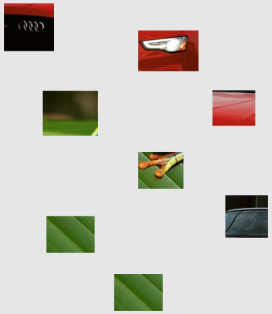
Cluster patches to form "codebook" of "visual words"

Image Features: Bag of Words

Learn a feature transform from data!

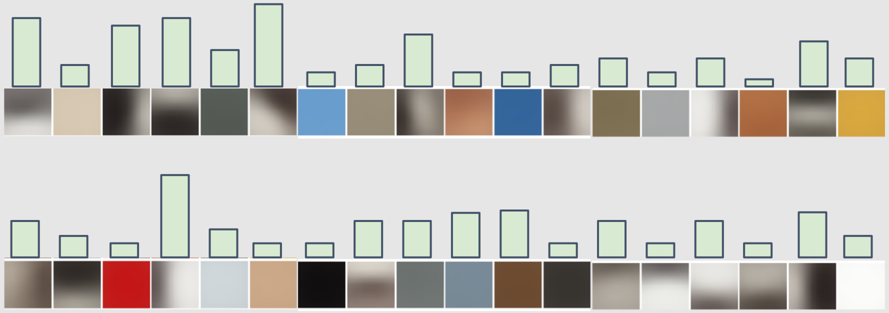
Step 1: Build codebook

Extract random patches



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

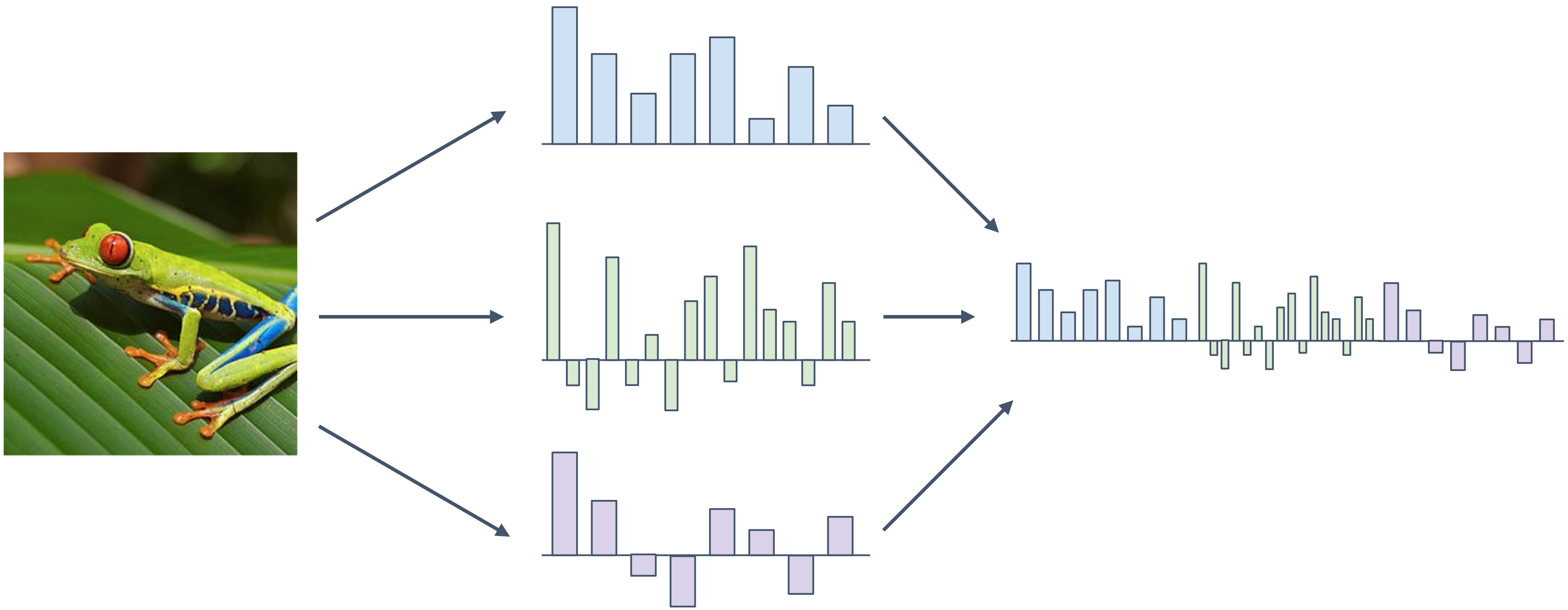
Step 2: Encode images



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

Image Features

Common trick: Combine multiple feature transforms



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

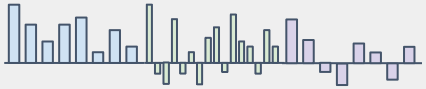
One-vs-all SVM learning with SGD

Late fusion of SIFT and color systems

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

Image Features vs Neural Networks

Feature Extraction



f

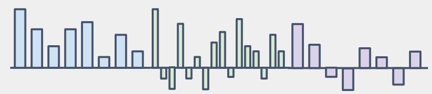
10 numbers giving scores for classes

training

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

Image Features vs Neural Networks

Feature Extraction

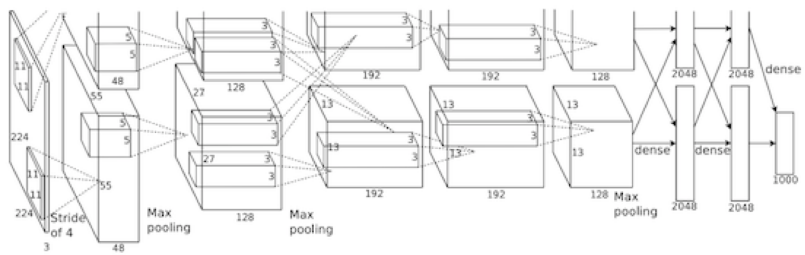


f

10 numbers giving scores for classes

training

Deep Neural Network



10 numbers giving scores for classes

training

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

Neural Networks

Input image: $x \in \mathbb{R}^D$
Category scores: $s \in \mathbb{R}^C$

Linear Classifier:

$$s = Wx$$
$$W \in \mathbb{R}^{C \times D}$$

In practice we add a learnable bias
+b after each matrix multiply

Neural Networks

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

Category scores: $s \in \mathbb{R}^C$

Linear Classifier:

$$s = Wx$$
$$W \in \mathbb{R}^{C \times D}$$

2-layer Neural Net:

$$s = \mathrm{W}_2 \max(0, W_1 x)$$
$$W_1 \in \mathbb{R}^{H \times D}$$
$$W_2 \in \mathbb{R}^{C \times H}$$

In practice we add a learnable bias
+b after each matrix multiply

Neural Networks

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

Category scores: $s \in \mathbb{R}^C$

Linear Classifier:

$$s = Wx$$

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

2-layer Neural Net:

$$s = \mathrm{W}_2 \max(0, W_1 x)$$

$$W_1 \in \mathbb{R}^{H \times D}$$

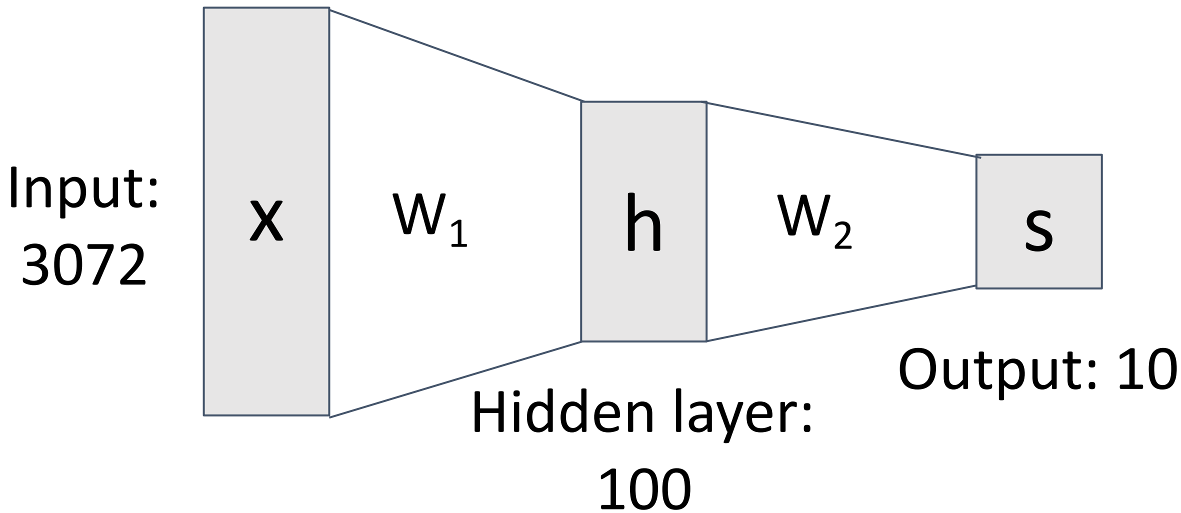
$$W_2 \in \mathbb{R}^{C \times H}$$

3-layer Neural Net:

$$s = \mathrm{W}_3 \max(0, W_2 \max(0, W_1 x))$$

Neural Networks

Two-Layer Neural Network: $s = \mathrm{W}_2 \max(0, W_1 x)$



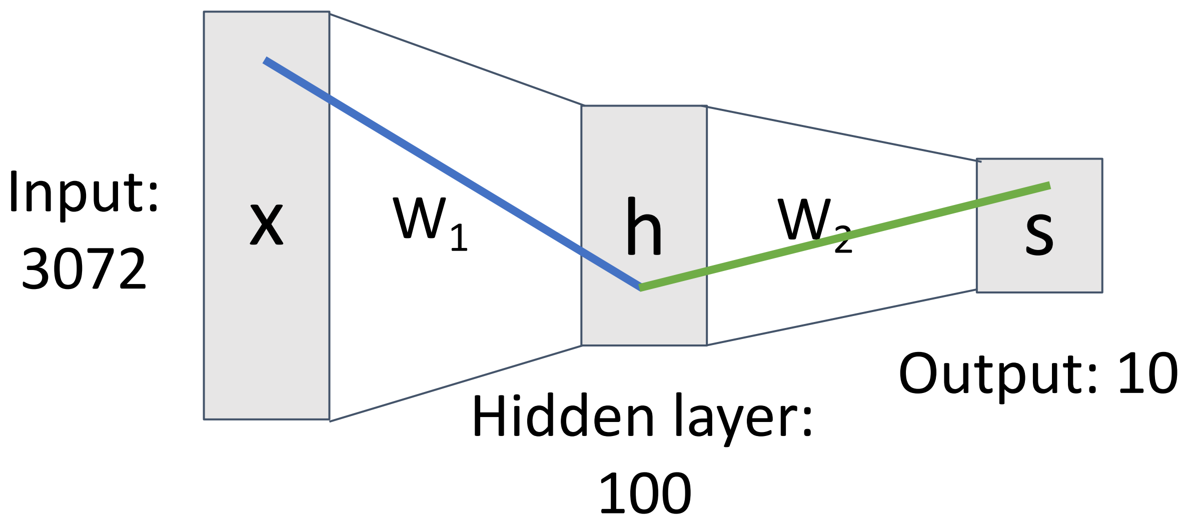
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times D}$$

Neural Networks

Two-Layer Neural Network: $s = W_2 \max(0, W_1 x)$

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



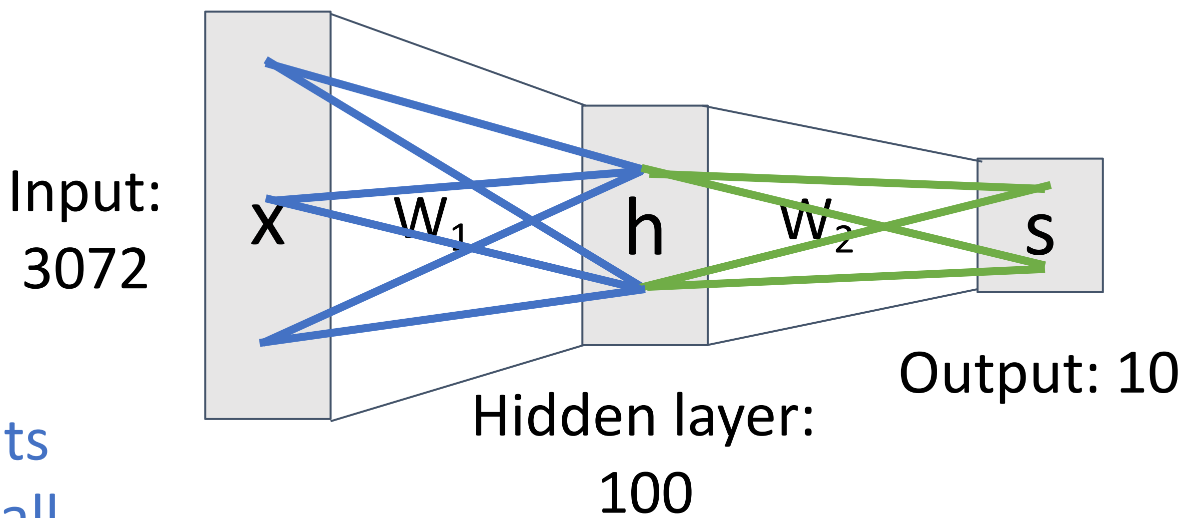
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H\times D}, W_2 \in \mathbb{R}^{C\times D}$$

Neural Networks

Two-Layer Neural Network: $s = W_2 \max(0, W_1 x)$

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



All elements of x affect all elements of h

All elements of h affect all elements of s

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

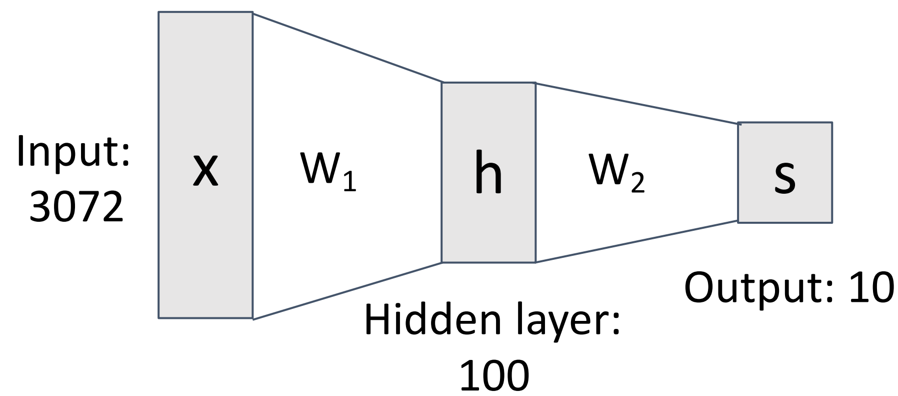
Neural Networks

Linear classifier: $s = Wx$

One template per class

Two-Layer Neural Network:

$s = \mathrm{W}_2 \max(0, W_1 x)$



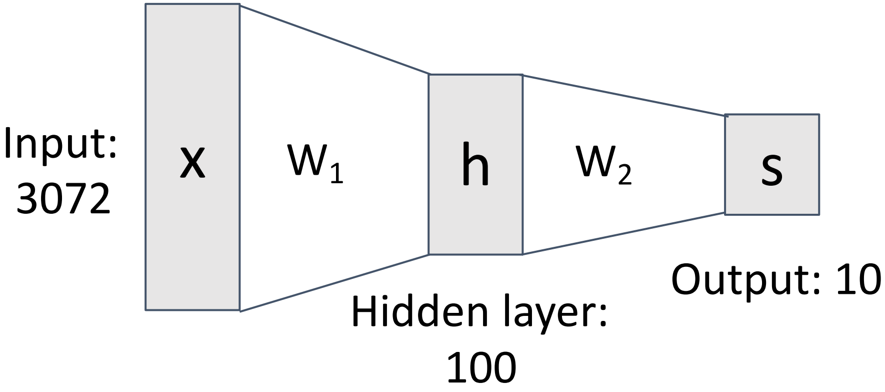
$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times D}$

Neural Networks

Neural Network:
First layer is a bank of templates
Second layer recombines templates

Two-Layer Neural Network:

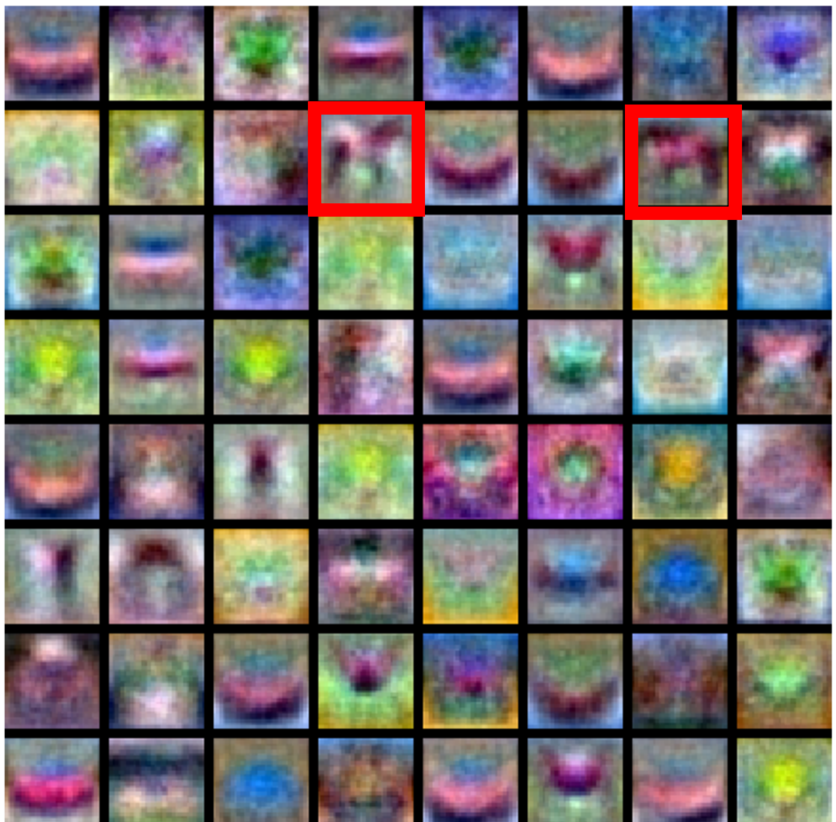
$$s = \mathrm{W}_2 \max(0, W_1 x)$$



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

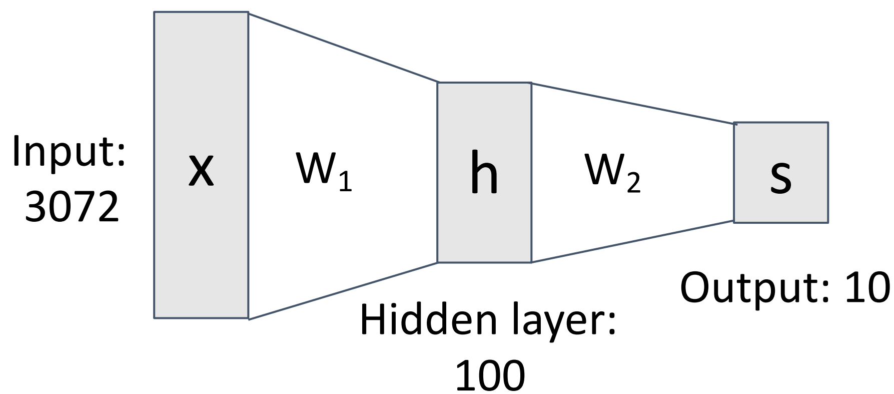
Neural Networks

Different templates can cover different modes of a class!



Two-Layer Neural Network:

$$s = \mathrm{W}_2 \max(0, W_1 x)$$



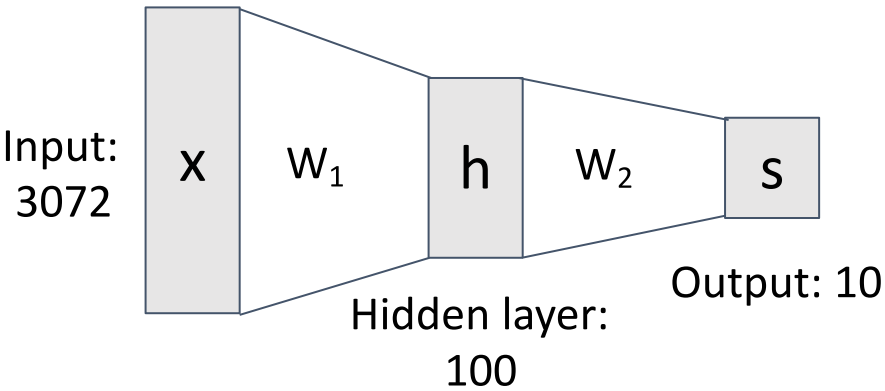
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times D}$$

Neural Networks

Many templates not interpretable:
"Distributed representation"

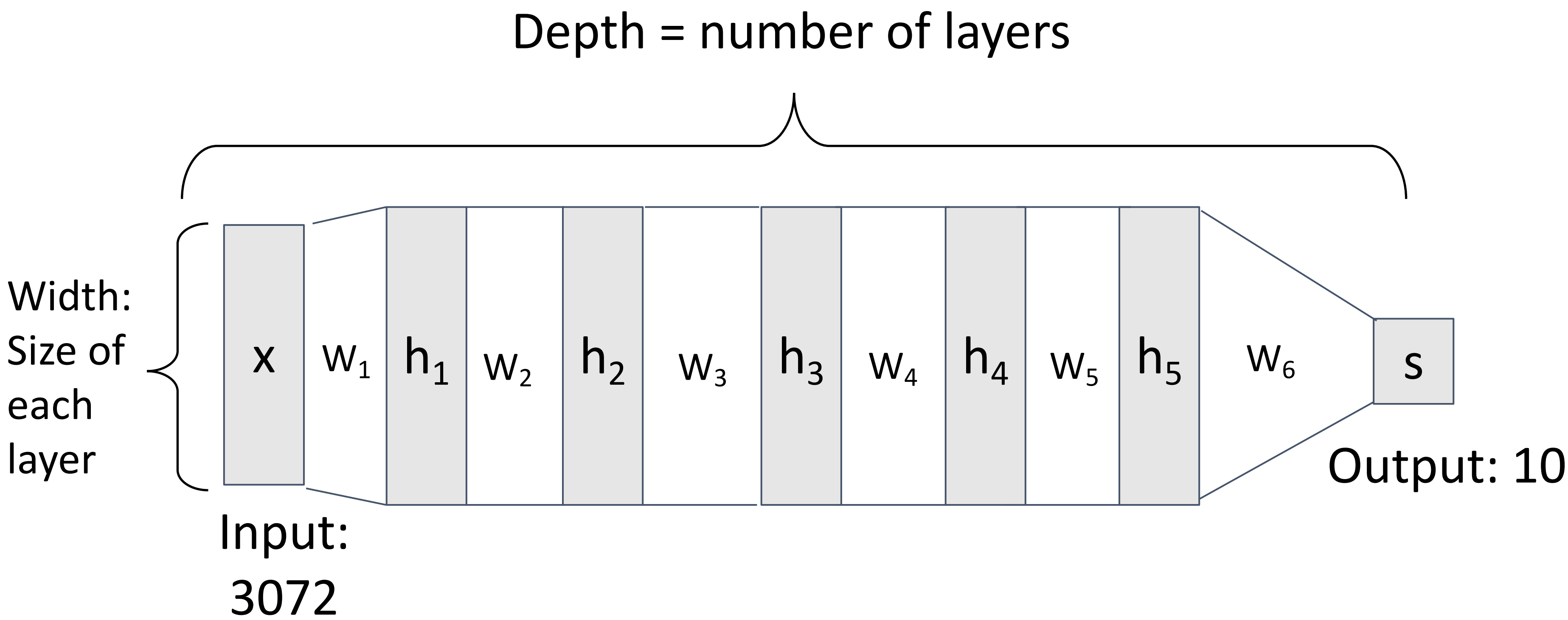
Two-Layer Neural Network:

$s = W_2 \max(0, W_1 x)$



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

Deep Neural Networks



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

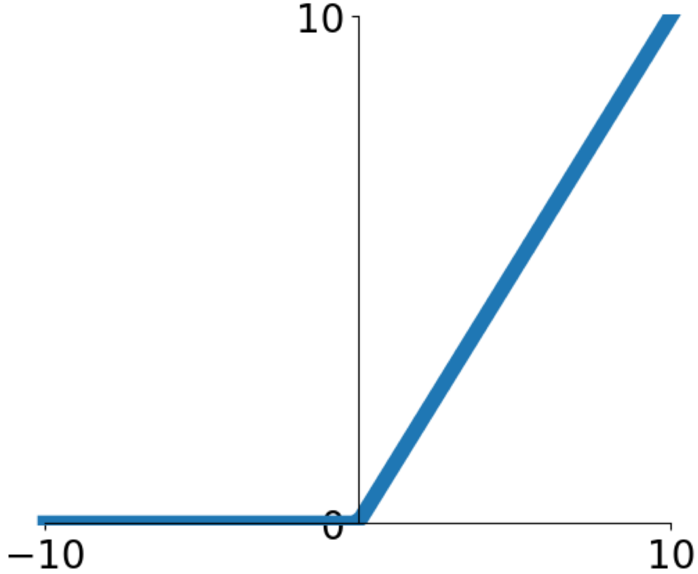
Activation Functions

2-layer Neural Network

$$s = W_2 \max(0, W_1 x)$$

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

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



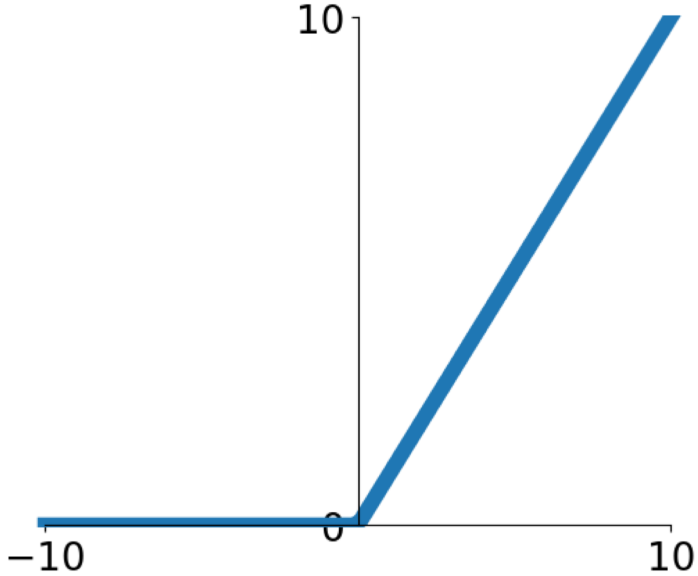
Activation Functions

2-layer Neural Network

$$s = W_2 \max(0, W_1 x)$$

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?

$$s = W_2 W_1 x$$

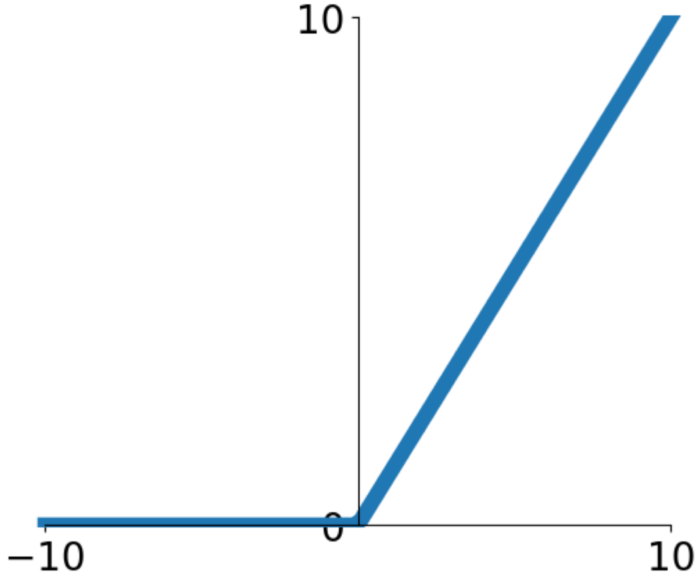
Activation Functions

2-layer Neural Network

$$s = W_2 \max(0, W_1 x)$$

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?

$$s = W_2 W_1 x$$

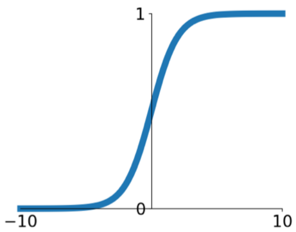
A: We get a linear classifier!

$$W_3 = W_2 W_1 \in \mathbb{R}^{C \times D}$$
$$s = W_3 x$$

Activation Functions

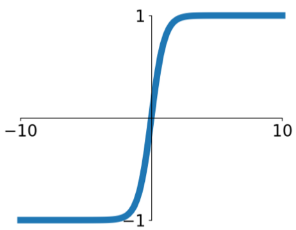
Sigmoid

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



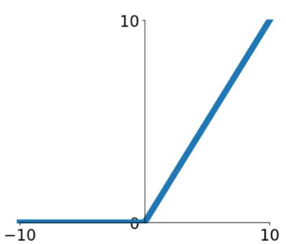
tanh

$\tanh(x)$



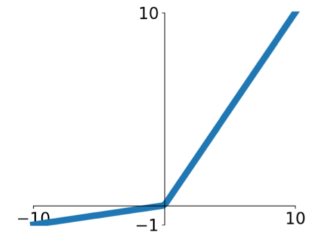
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

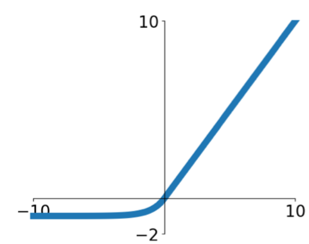


Maxout

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

ELU

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

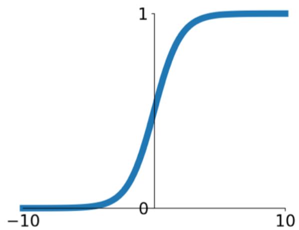


Activation Functions

ReLU is a good default choice

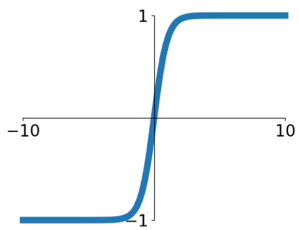
Sigmoid

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



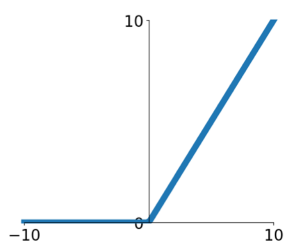
tanh

$\tanh(x)$



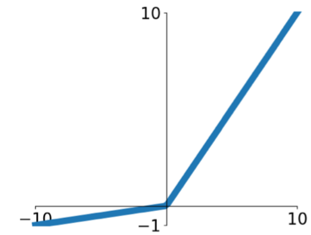
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

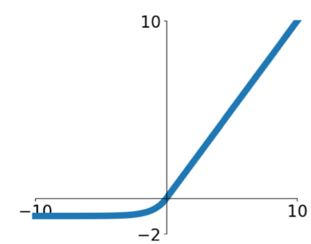


Maxout

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

ELU

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



Neural Net in <20 lines!

```
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
```

Neural Net in <20 lines!

Initialize weights and data

```
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
```

Neural Net in <20 lines!

Initialize weights and data

Compute loss (sigmoid activation, L2 loss)

```
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
```

Neural Net in <20 lines!

Initialize weights and data

Compute loss (sigmoid activation, L2 loss)

Compute gradients

```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
```

Neural Net in <20 lines!

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

```

Initialize weights and data

```python
N, Din, H, Dout = 64, 1000, 100, 10
x, y = randn(N, Din), randn(N, Dout)
w1, w2 = randn(Din, H), randn(H, Dout)
for t in range(10000):
```

Compute loss (sigmoid activation, L2 loss)

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

Compute gradients

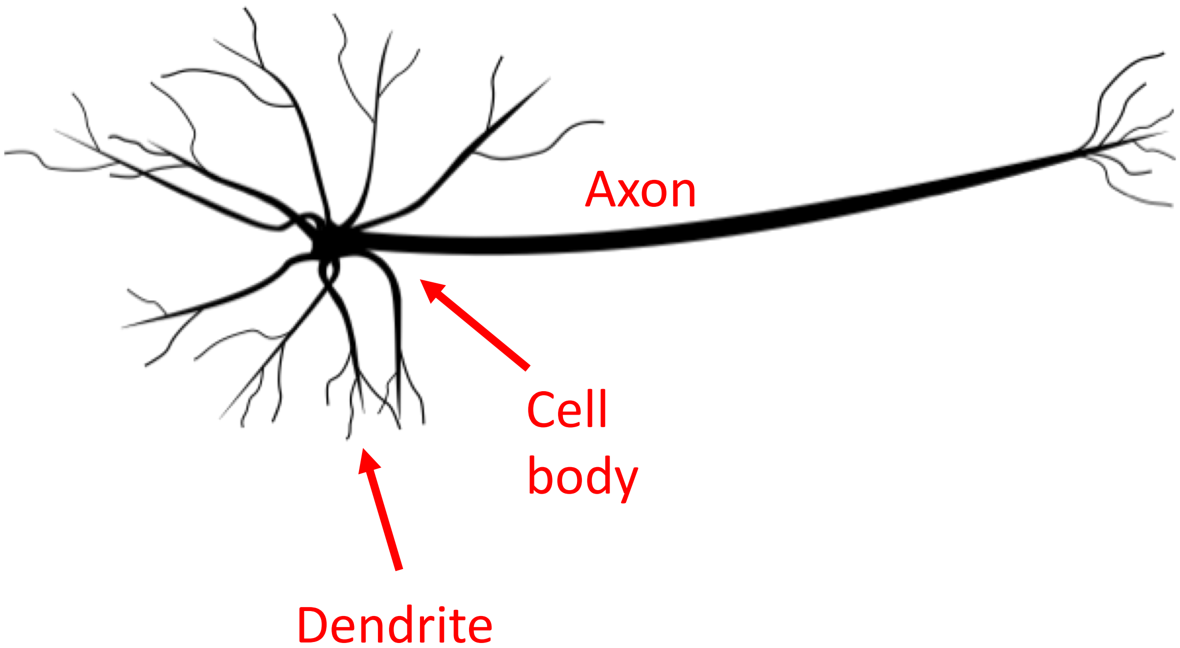
```python
  dy_pred = 2.0 * (y_pred - y)
  dw2 = h.T.dot(dy_pred)
  dh = dy_pred.dot(w2.T)
  dw1 = x.T.dot(dh * h * (1 - h))
```

SGD step

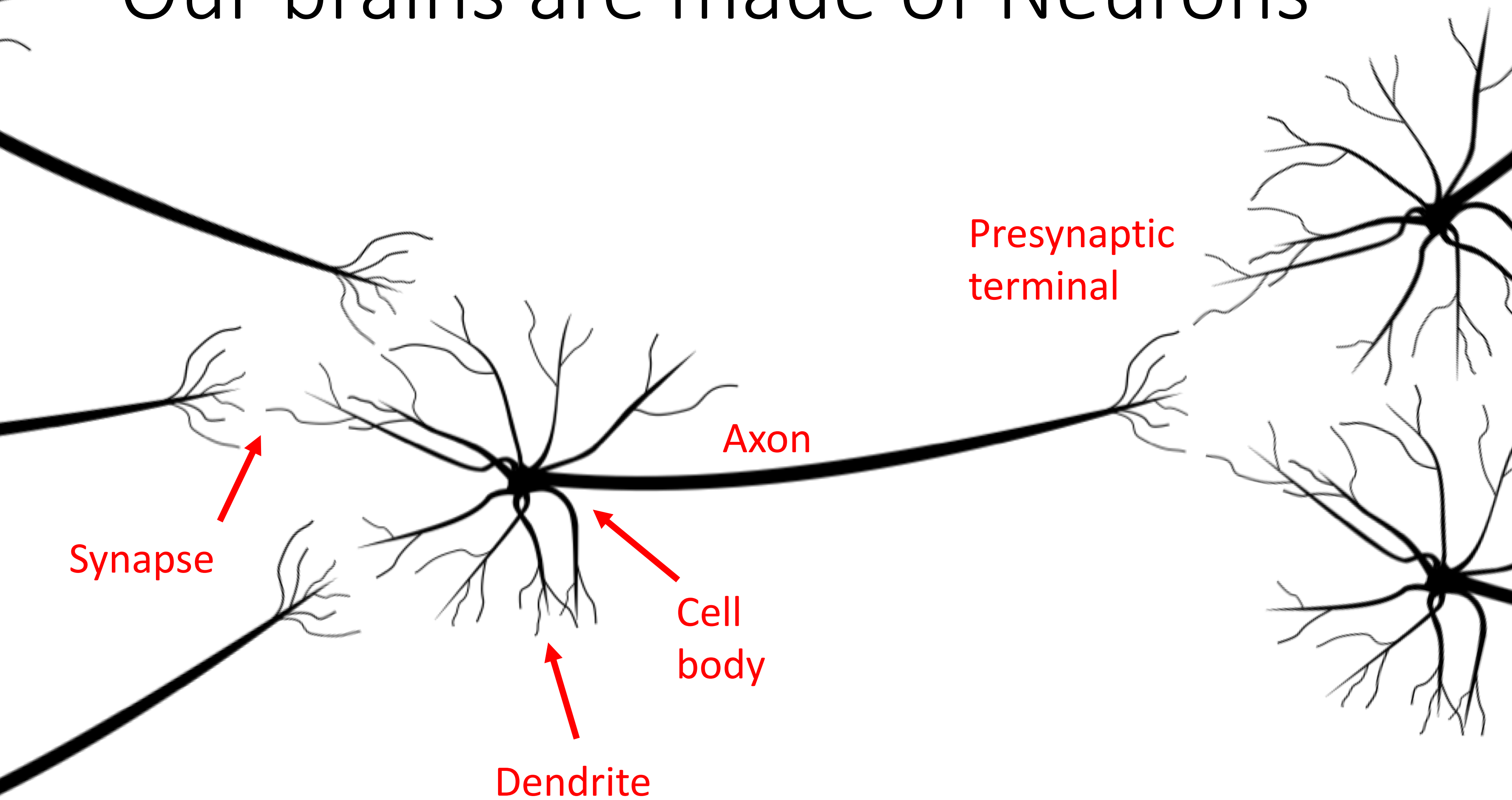
```python
  w1 -= 1e-4 * dw1
  w2 -= 1e-4 * dw2
```

“Neural” Networks

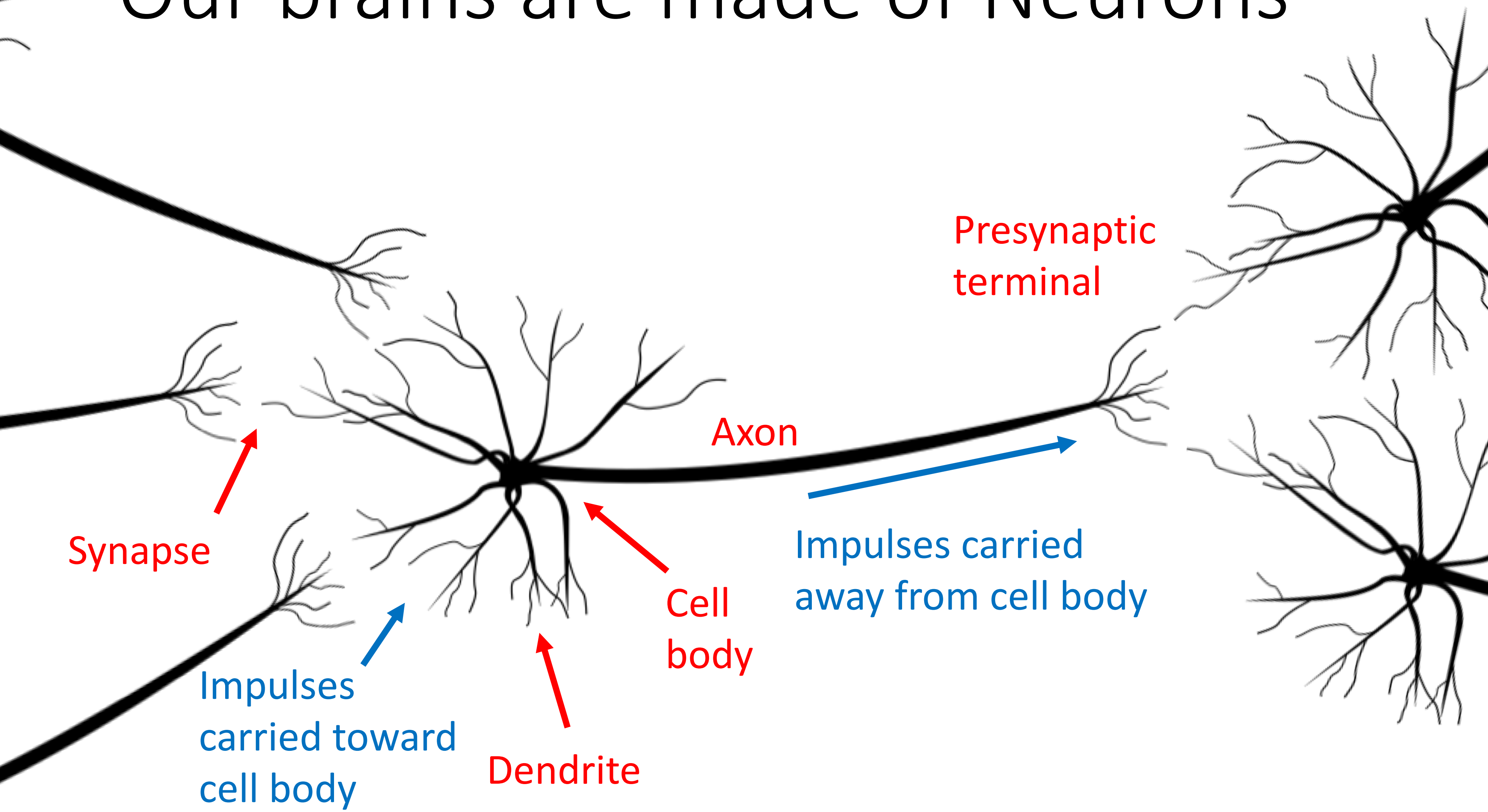
Our brains are made of Neurons



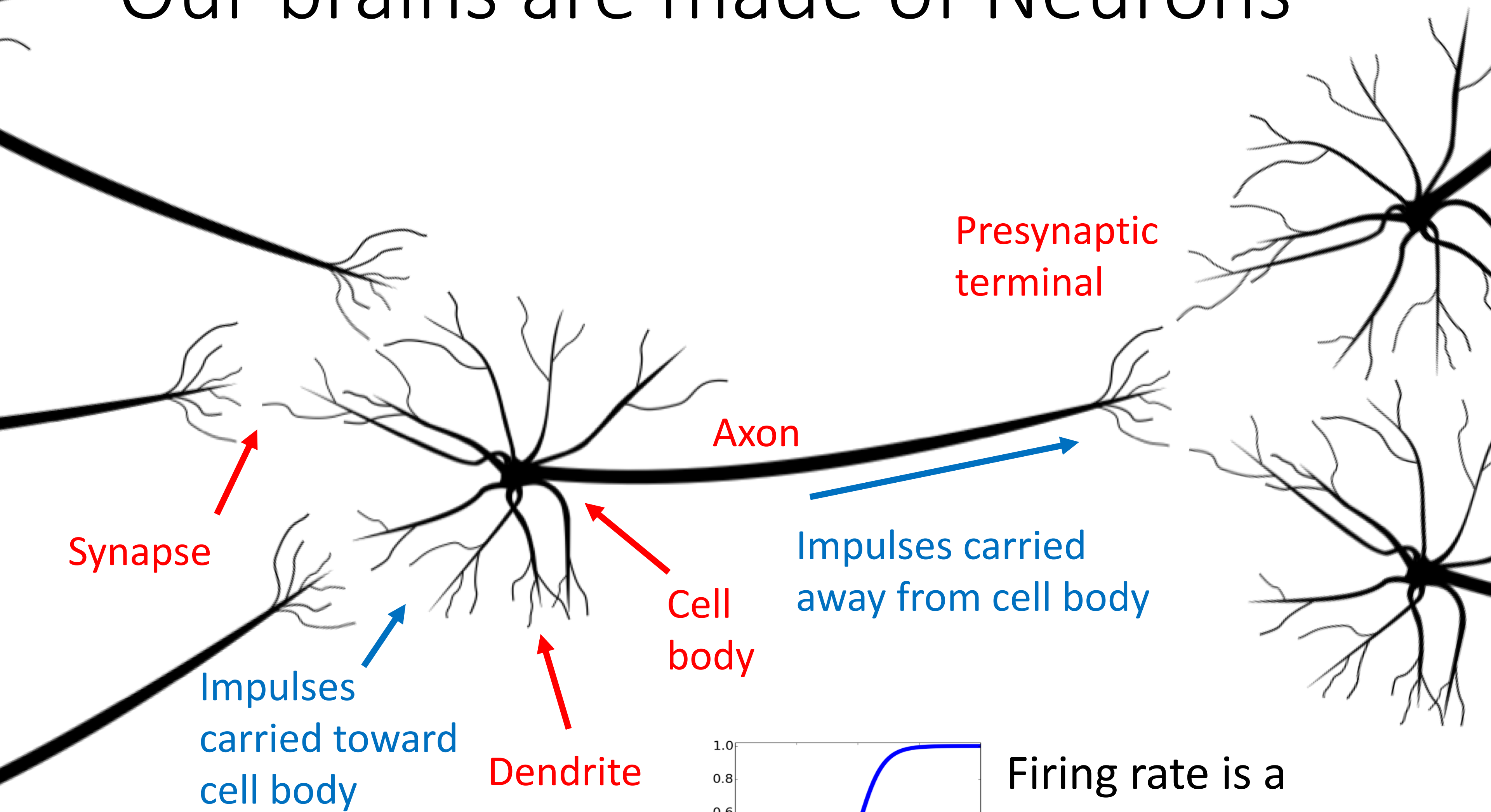
Our brains are made of Neurons

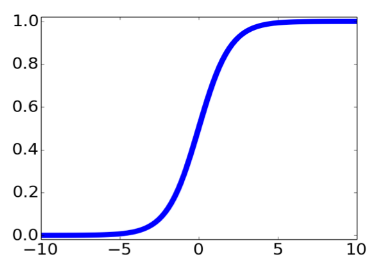


Our brains are made of Neurons

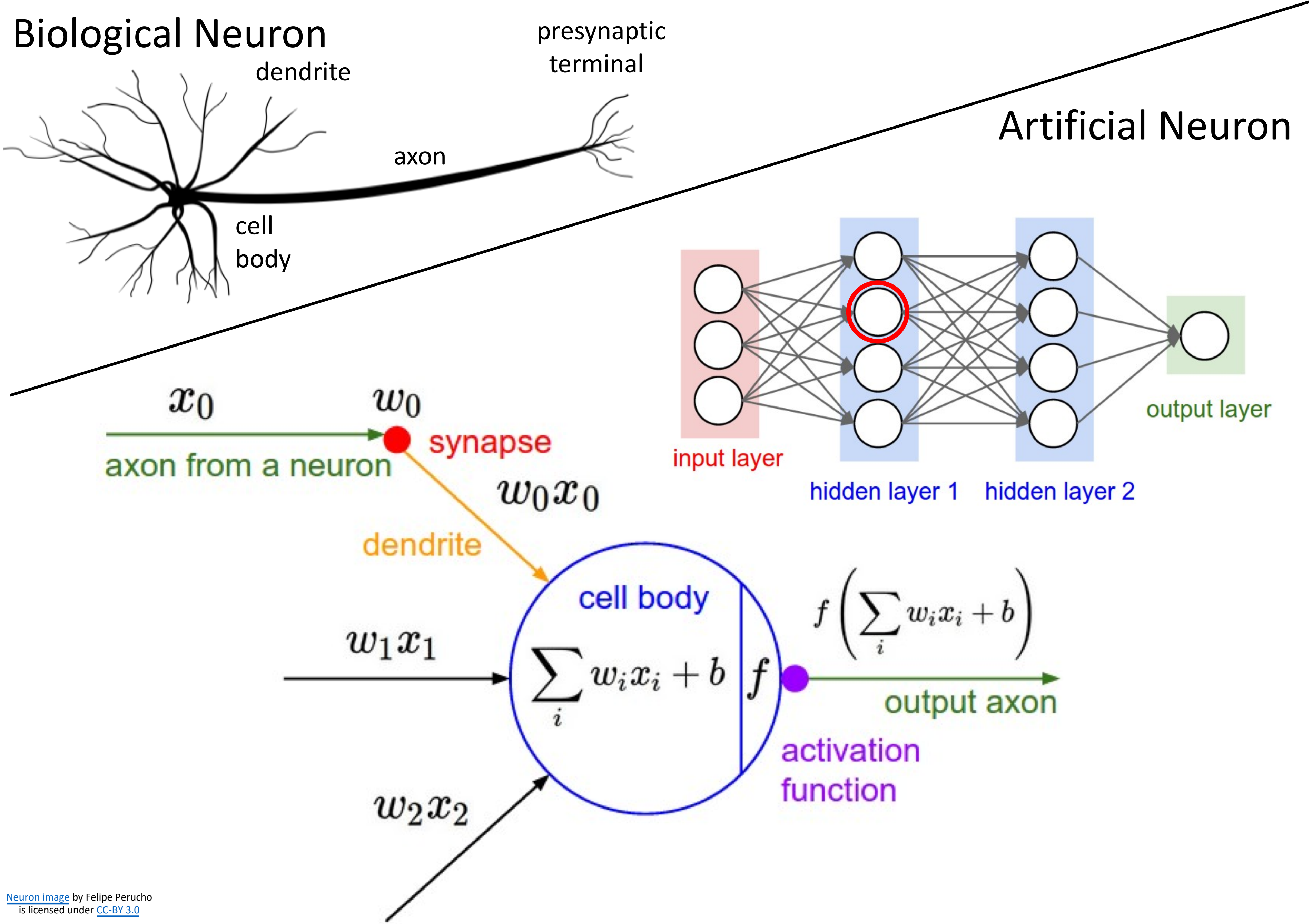


Our brains are made of Neurons

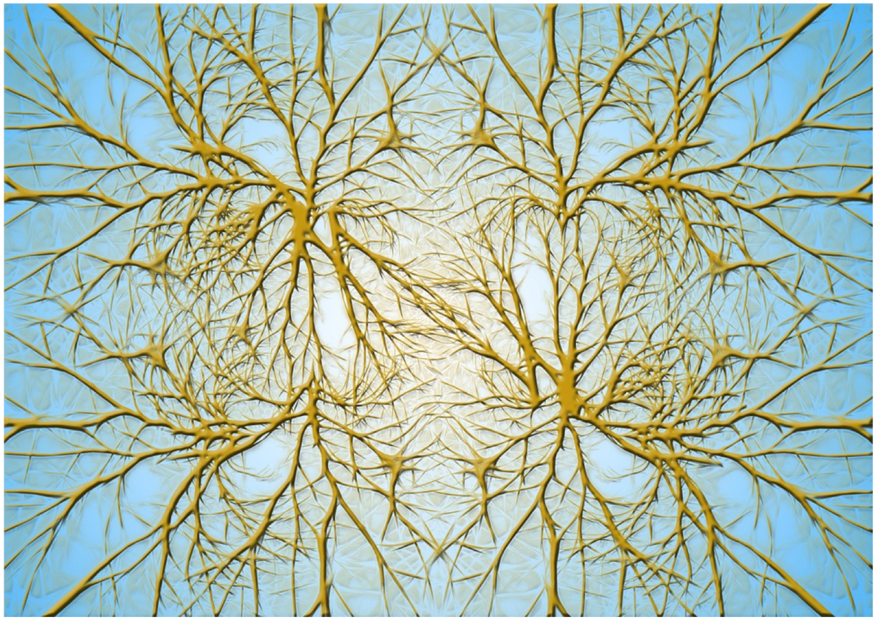




Firing rate is a nonlinear function of inputs

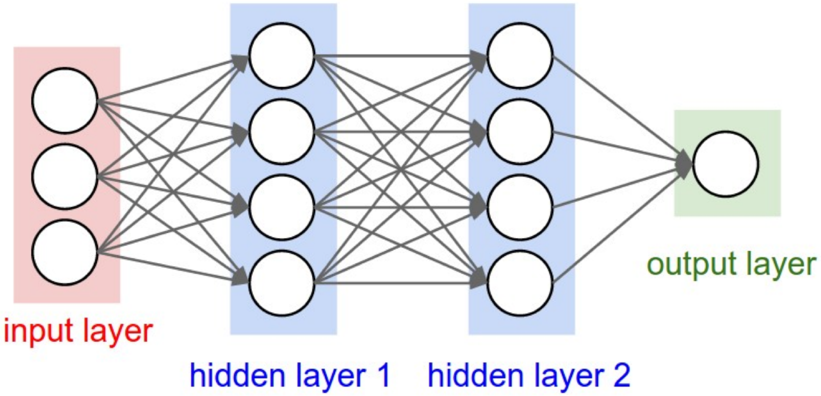

Biological Neuron
dendrite
presynaptic terminal
axon
cell body
Artificial Neuron
input layer
hidden layer 1
hidden layer 2
output layer
x_0
w_0
synapse
axon from a neuron
$w_0 x_0$
dendrite
cell body
$\sum_i w_i x_i + b$
f
$f\left(\sum_i w_i x_i + b\right)$
$w_1 x_1$
output axon
activation function
$w_2 x_2$

Biological Neurons:
Complex connectivity patterns



This image is CC0 Public Domain

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



Be very careful with brain analogies!

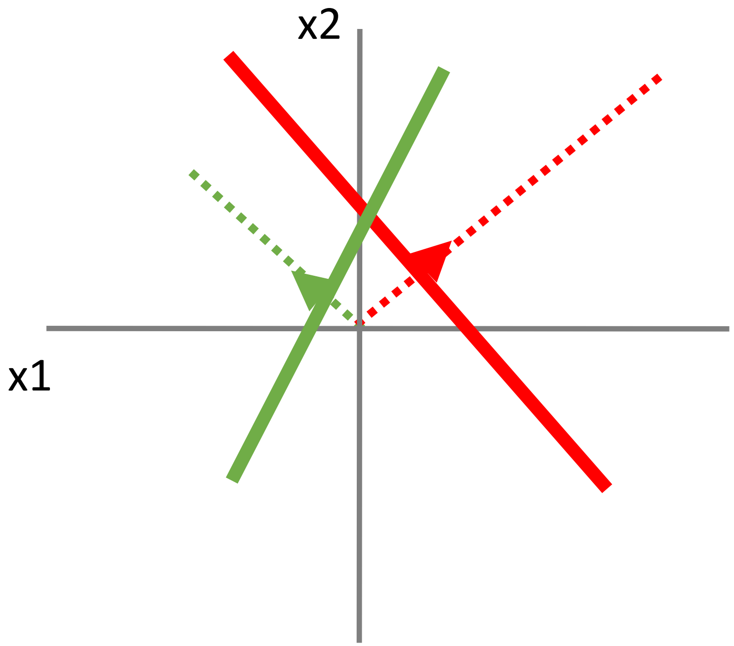
Biological Neurons:

- Many different types
- Can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system
- Can have feedback, time-dependent
- Probably don’t learn via gradient descent

[Dendritic Computation. London and Hausser]

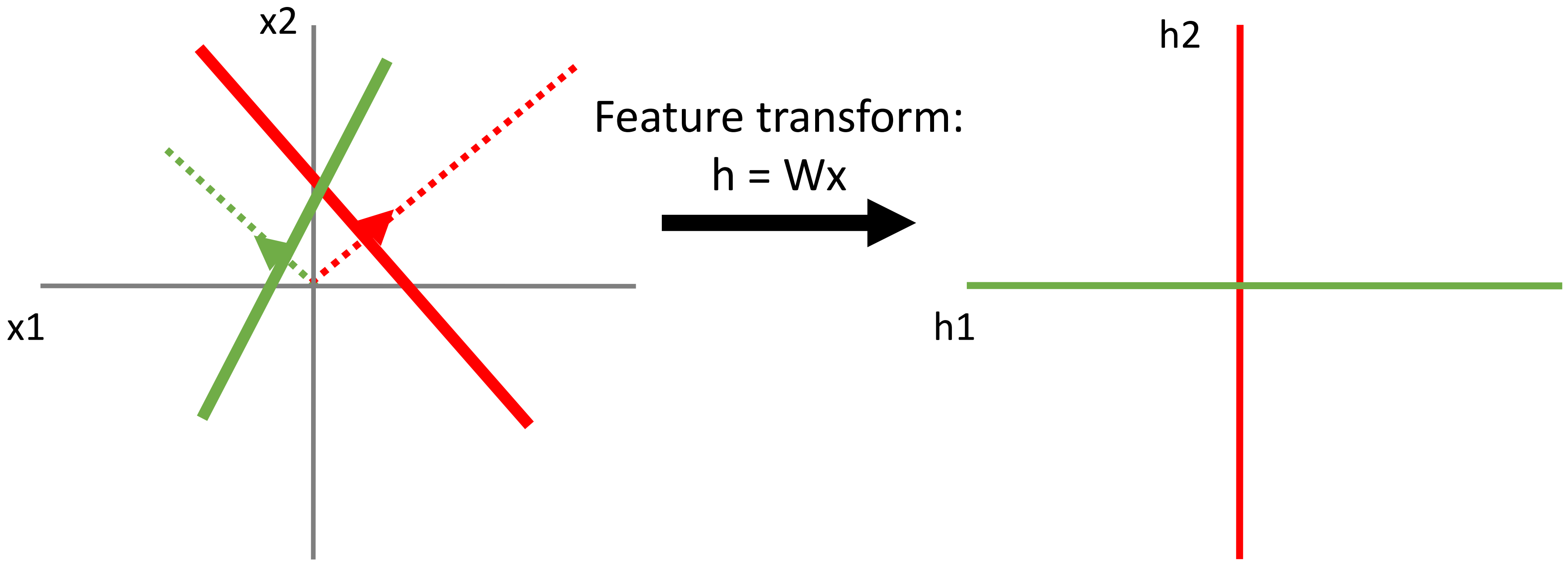
Space Warping

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



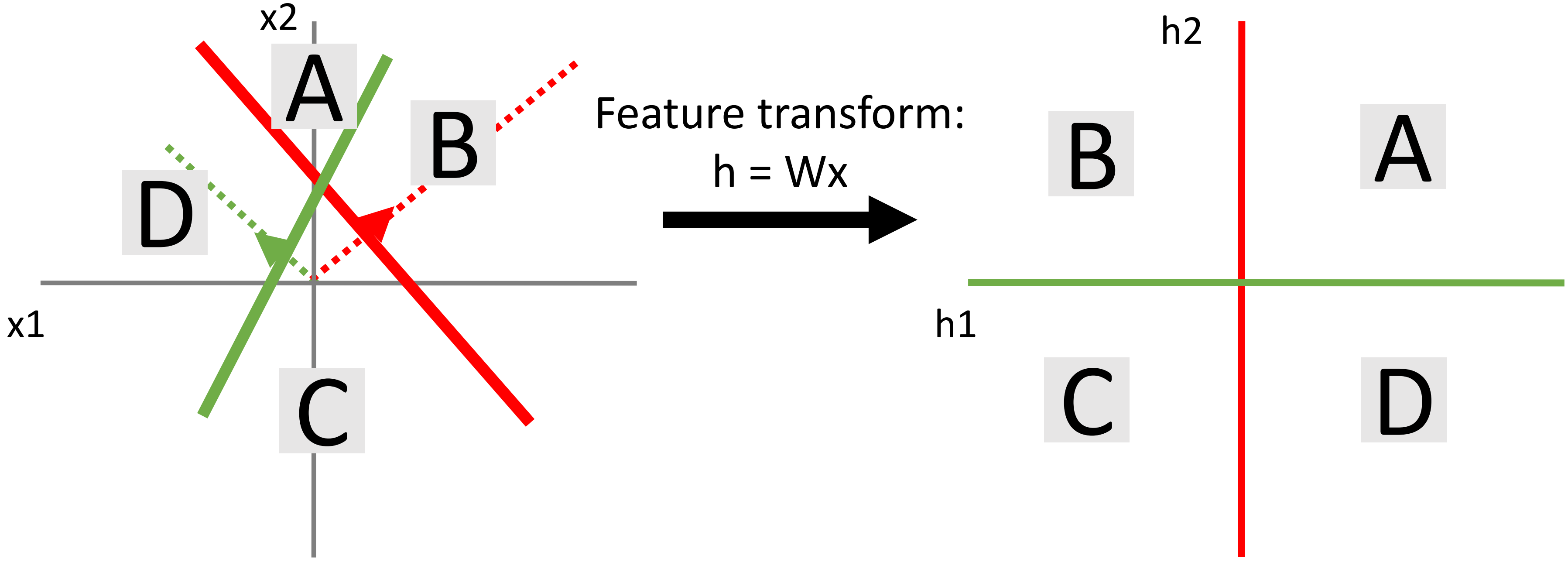
Space Warping

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



Space Warping

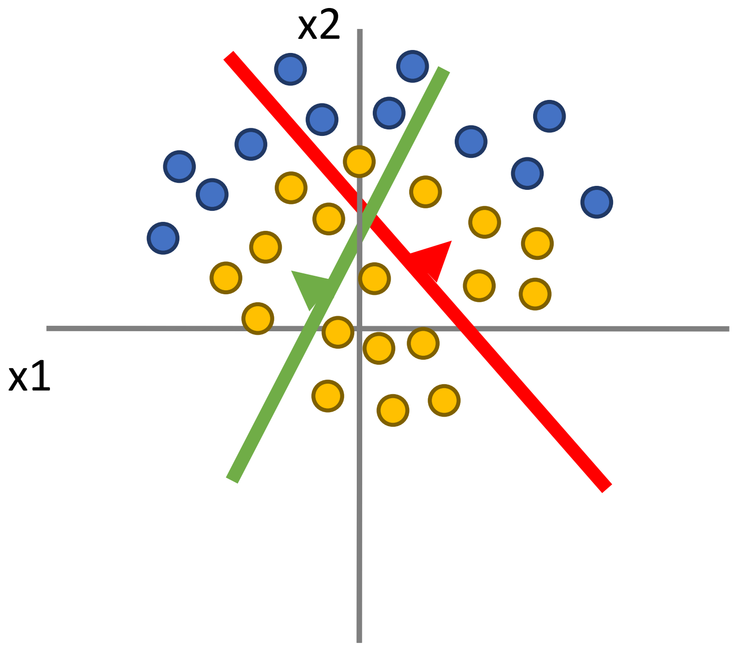
Consider a linear transform: h = Wx
Where x, h are both 2-dimensional



Space Warping

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

Points not linearly separable in original space

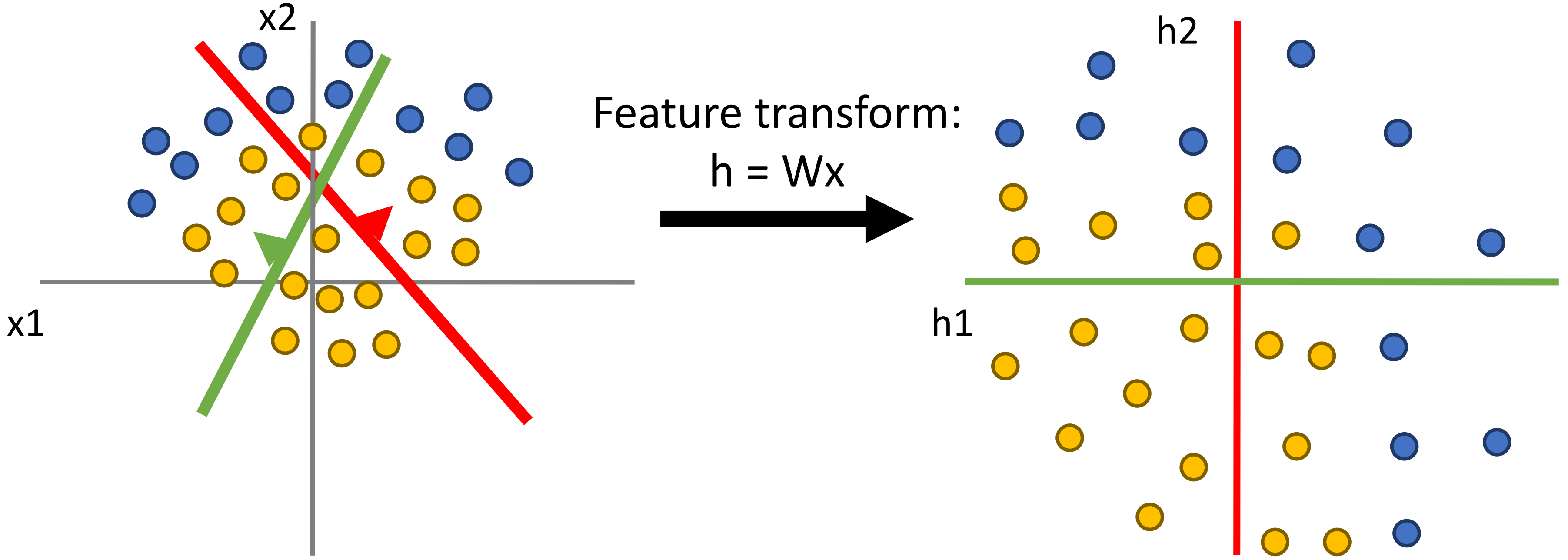


Space Warping

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

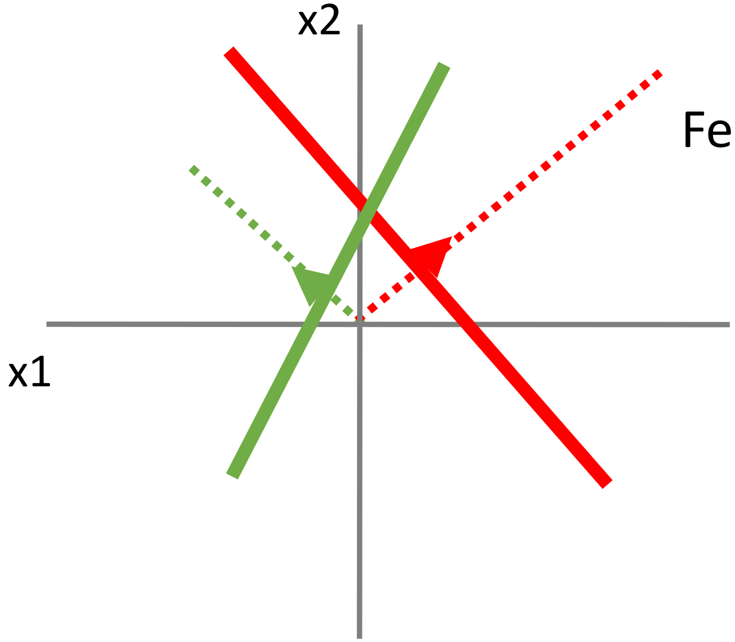
Points not linearly separable in original space

Points not linearly separable in feature space

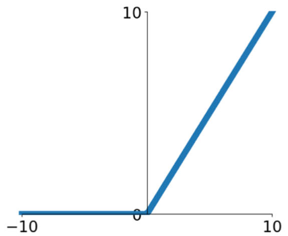


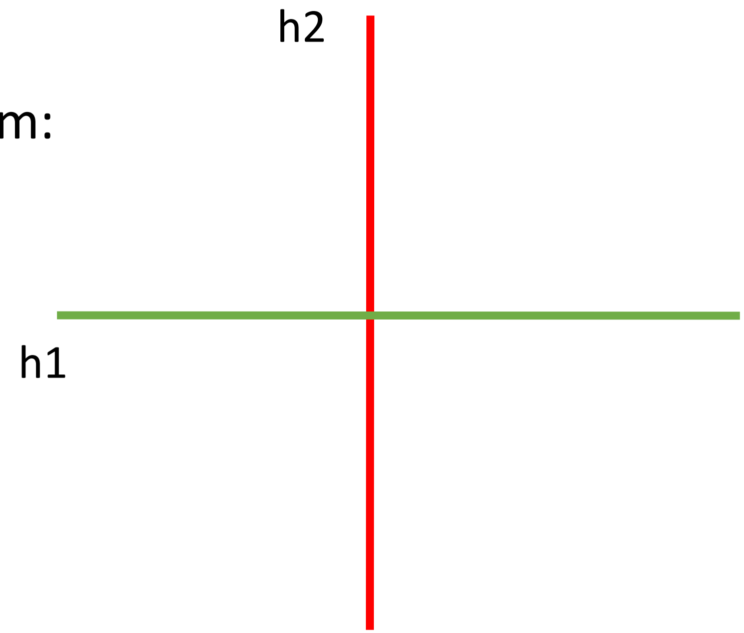
Space Warping

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



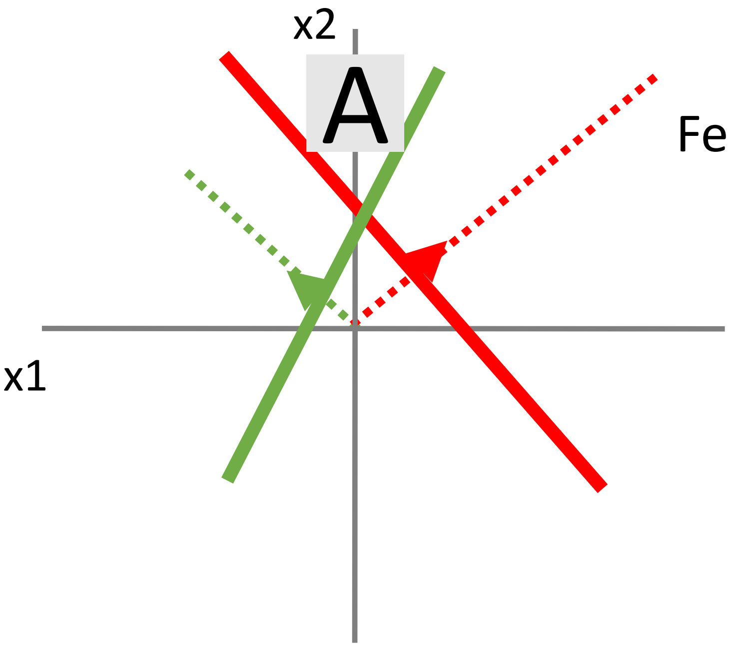
Feature transform:
h = ReLU(Wx)



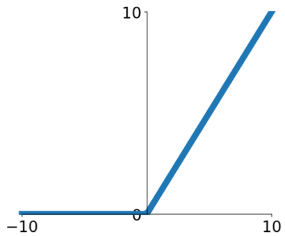


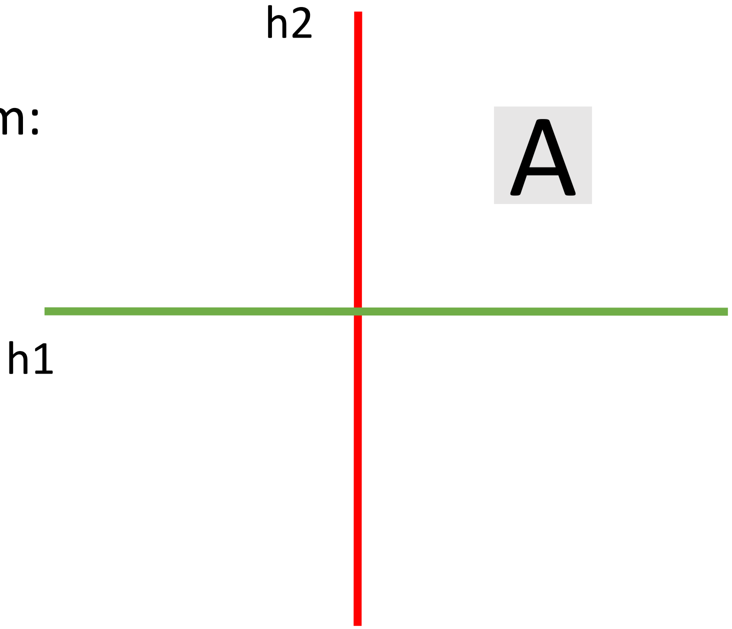
Space Warping

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



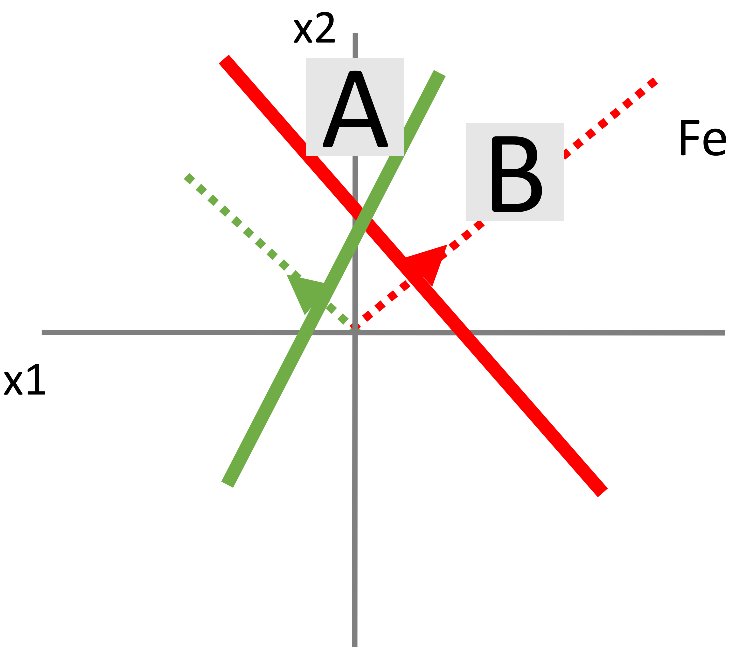
Feature transform:
h = ReLU(Wx)



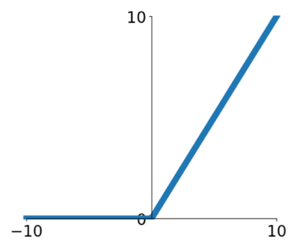


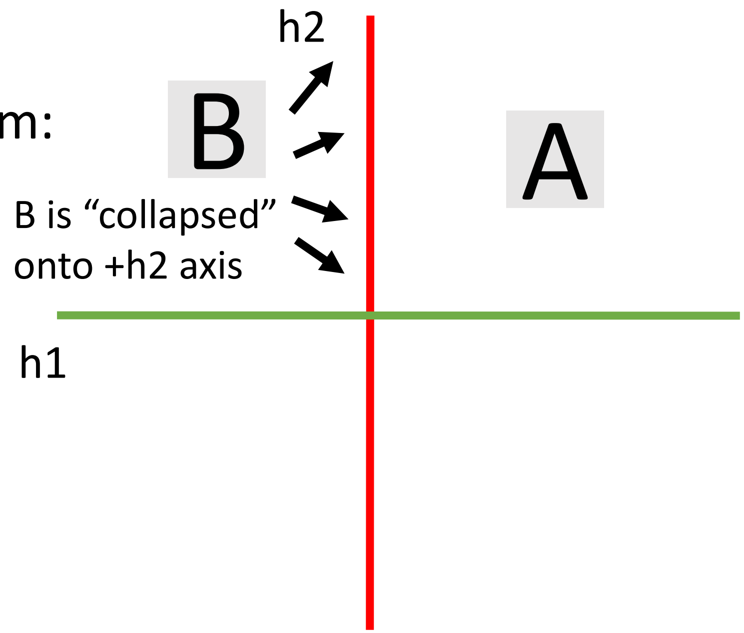
Space Warping

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



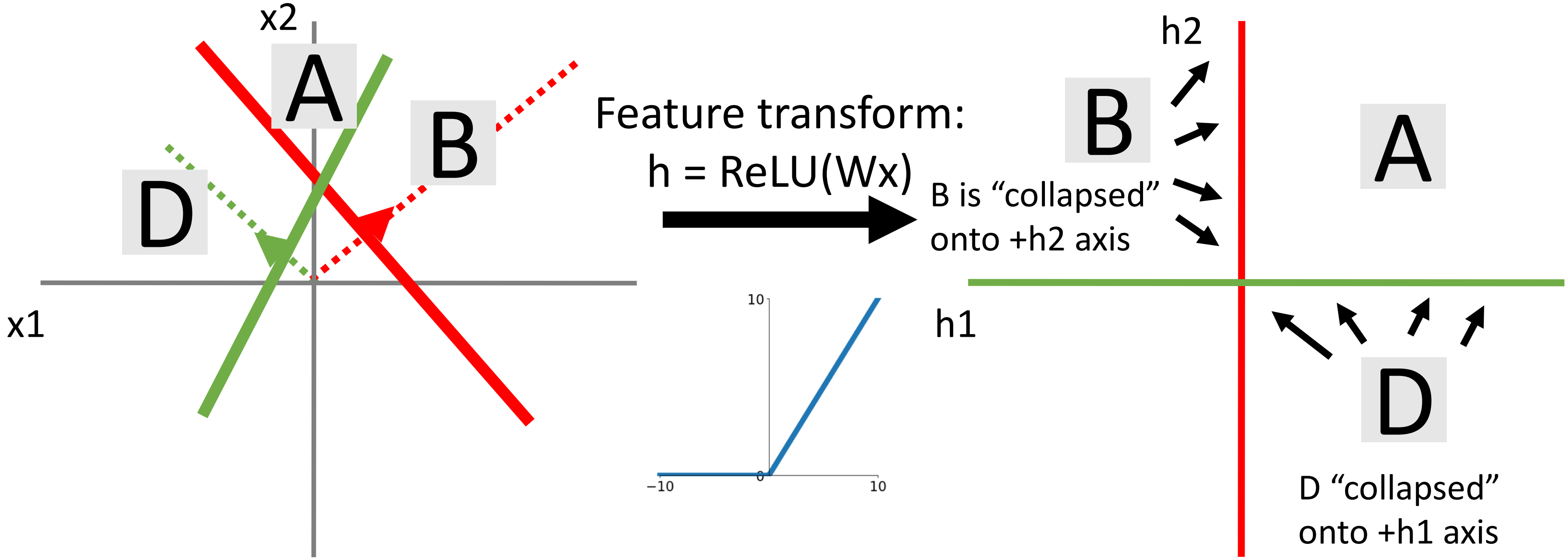
Feature transform:
h = ReLU(Wx)





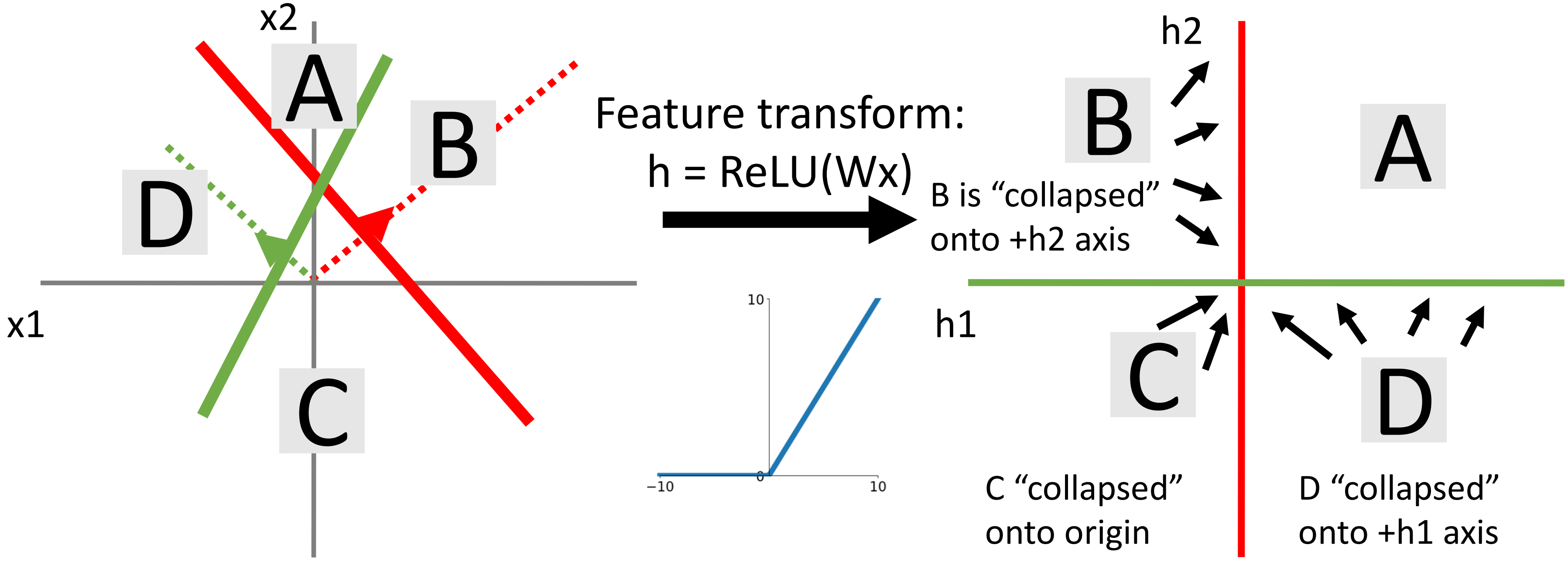
Space Warping

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



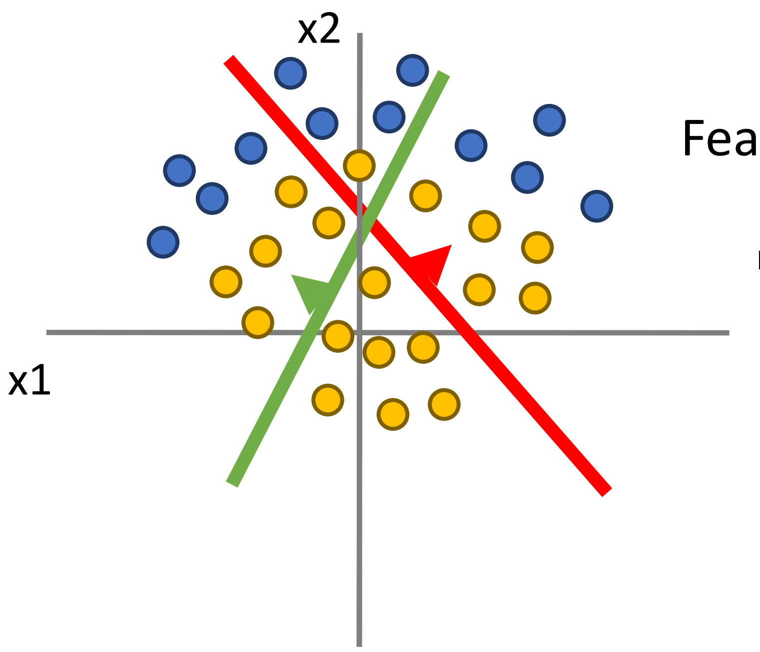
Space Warping

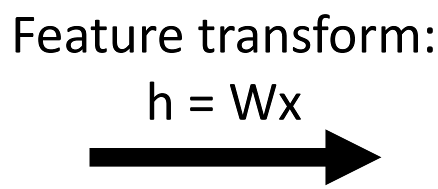
Consider a neural net hidden layer:
h = ReLU(Wx) = max(0, Wx)
Where x, h are both 2-dimensional

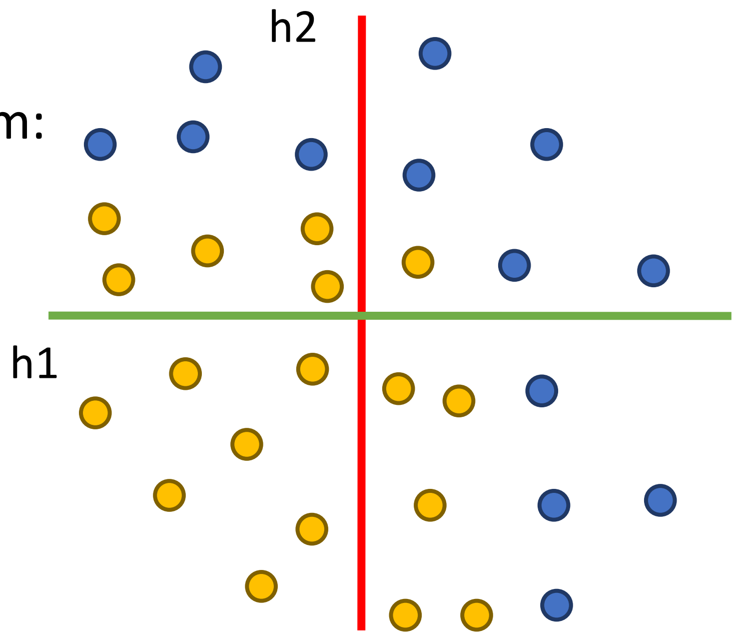


Space Warping

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



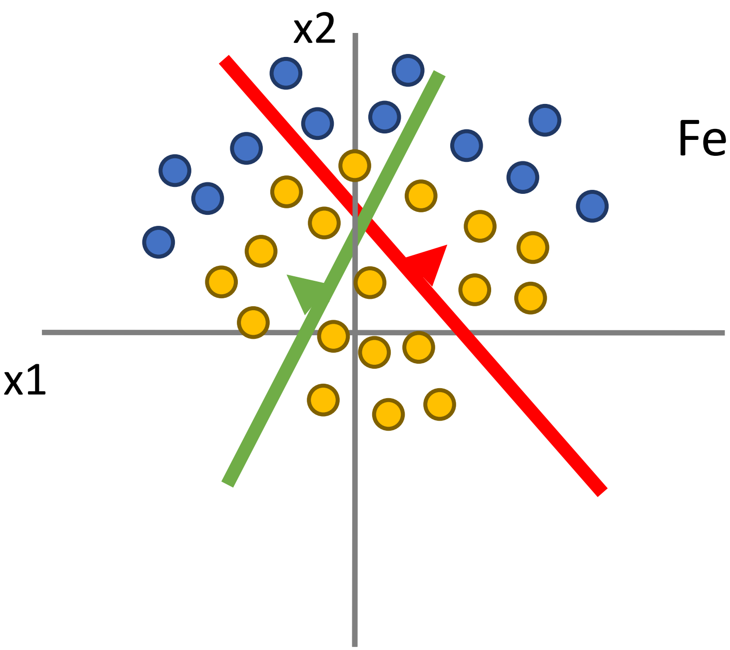




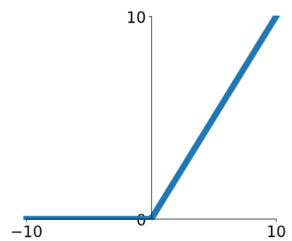
Points not linearly
separable in original space

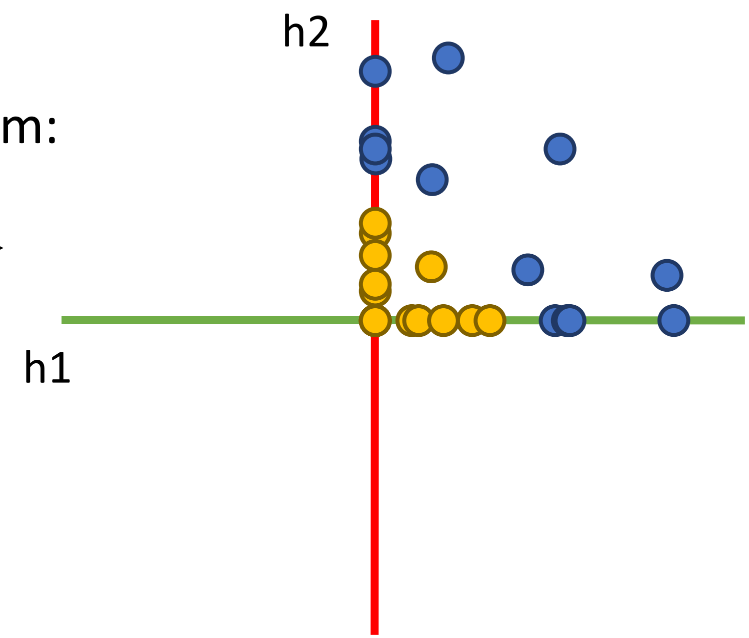
Space Warping

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



Feature transform:
h = ReLU(Wx)

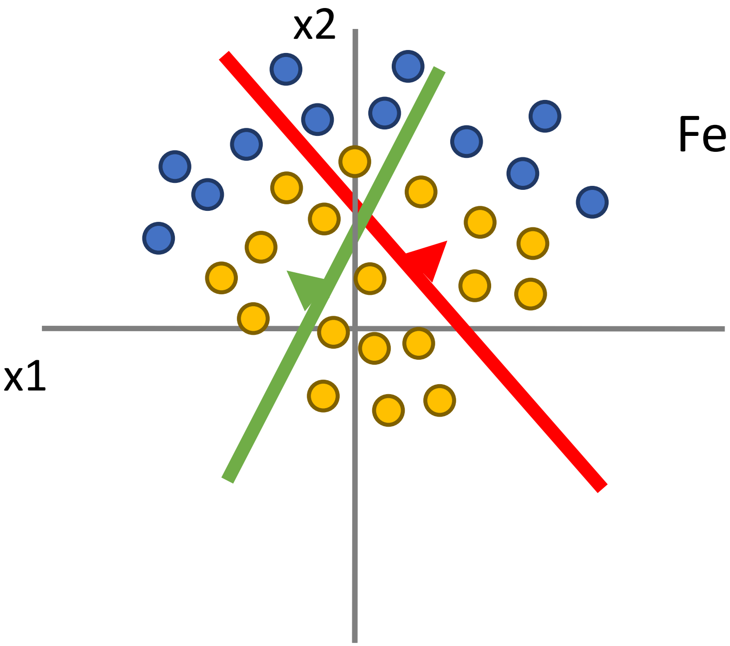




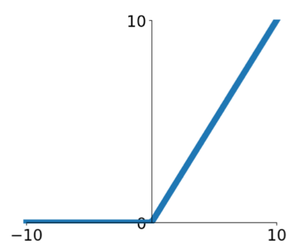
Points not linearly separable in original space

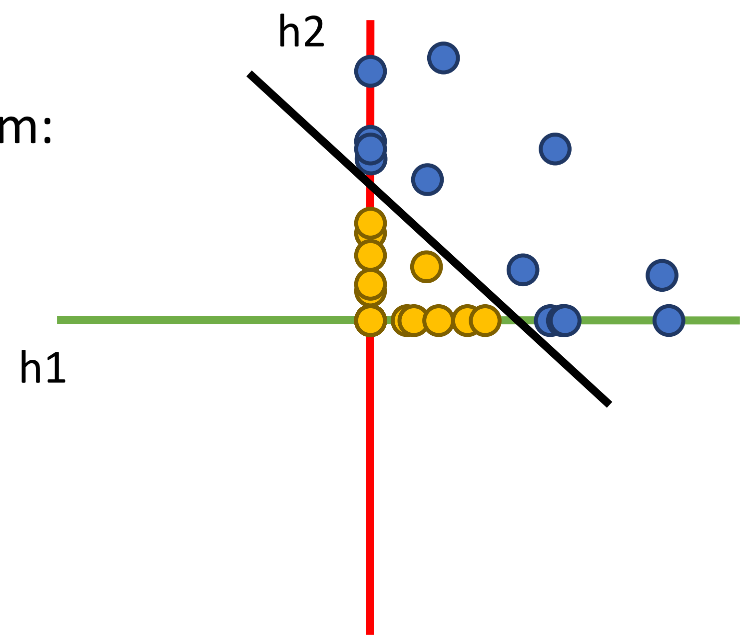
Space Warping

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



Feature transform:
h = ReLU(Wx)





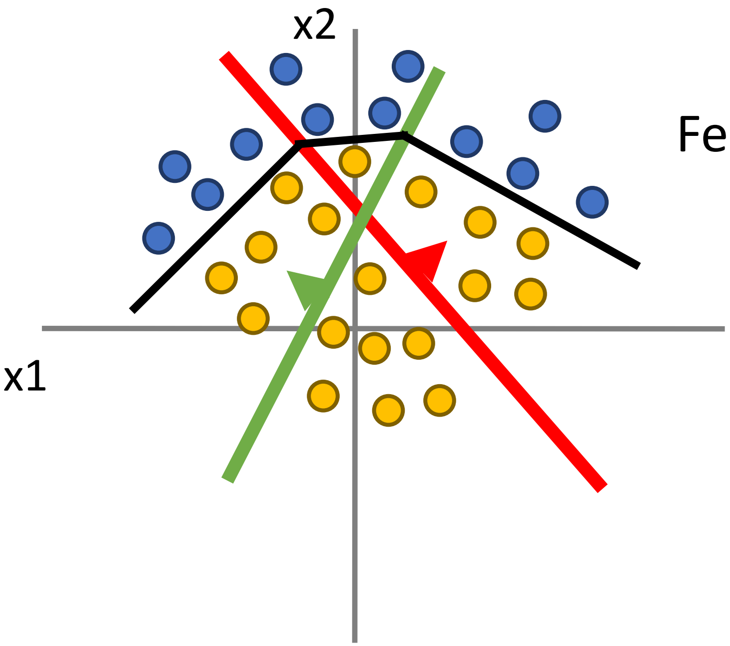
Points not linearly separable in original space

Points are linearly separable in features space!

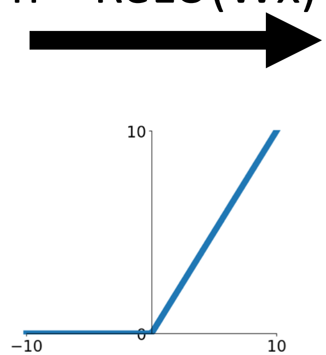
Space Warping

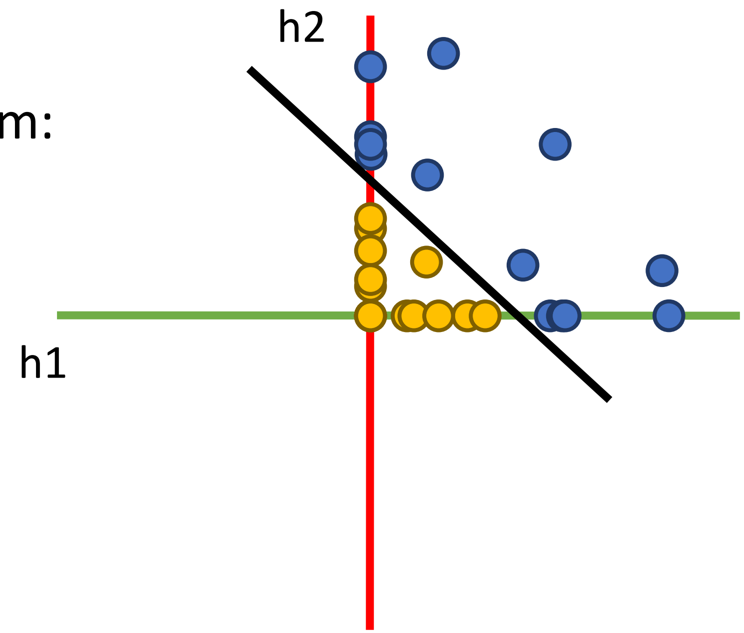
Linear classifier in feature space gives nonlinear classifier in original space

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



Feature transform:
h = ReLU(Wx)

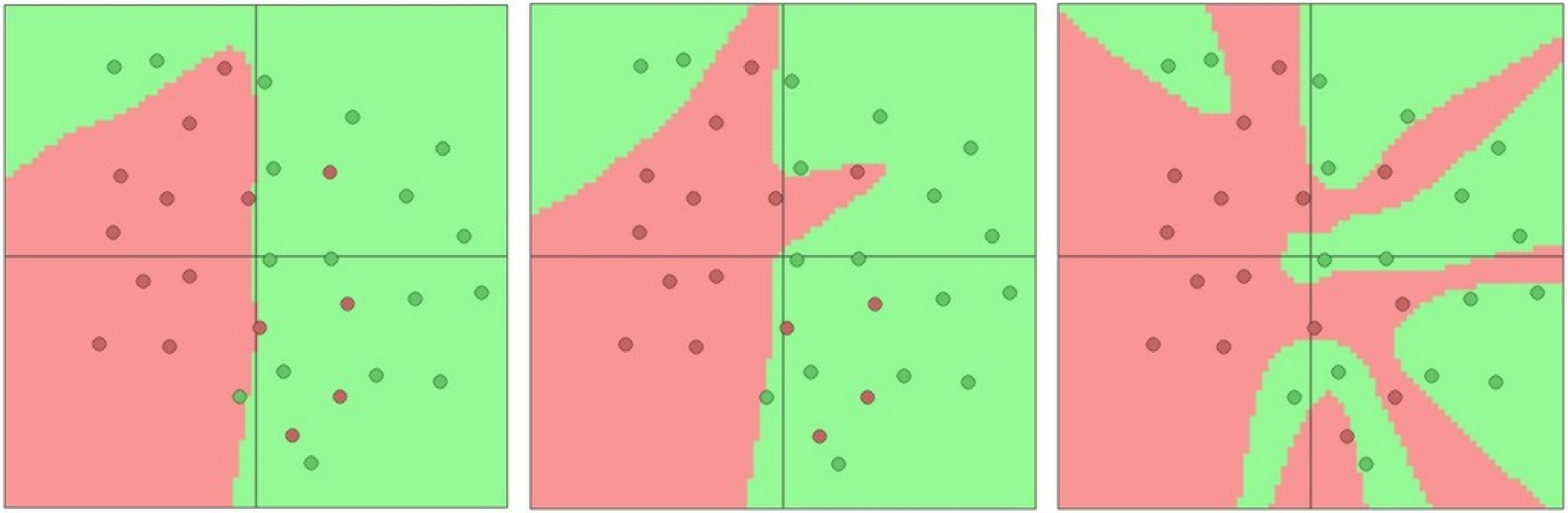




Points not linearly separable in original space

Points are linearly separable in features space!

Neural Networks Web Demo



(Web demo with ConvNetJS:
http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html)

Next Time: How to compute gradients? Backpropagation