17. Neural Networks

STA3142 Statistical Machine Learning

Kibok Lee

Assistant Professor of

Applied Statistics / Statistics and Data Science

May 14, 2024

** Slides adapted from EECS498/598 @ Univ. of Michigan by Justin Johnson*

연세대학교
YONSEI UNIVERSITY

Assignment 4

- Due **Friday 5/17, 11:59pm**
- Topics
  - K-Means and Gaussian Mixture Models
  - Principal Component Analysis
- Please read the instruction carefully!
  - Submit one pdf and one zip file separately
  - Write your code only in the designated spaces
  - Do not import additional libraries
  - …
- If you feel difficult, consider to take option 2.

Midterm Questions

- Questions got >= 3 votes:
  - 1.1~1.4, 1.8, 1.12, 2.1, 3~

Recap: Rough Plan

- We have **13 weeks (39 hours)** of lectures.

- **6 hours** for intro & math/python review
- **15 hours** for basic ML & supervised learning
  - Regression, classification, kernel methods, validation, …
- **3 hours** for unsupervised learning
- **12 hours** for neural networks
- **3 hours** for others
  - Reinforcement learning, summary
  - Note: You can take **STA3145** for RL

Note: Other Useful ML Topics

- **Probabilistic graphical model (PGM)**
  - Bayesian networks, Markov random fields, conditional random fields, (restricted) Boltzmann machine
  - **Gradient-based neural networks are preferred these days**
- **Hidden Markov model (HMM)**
  - For sequential data; **RNNs and Transformers are good replacements**
- **Tree-based models (useful for tabular data)**
  - Decision tree and random forest
  - Bootstrapping, bagging, and boosting
- **Statistical learning theory (for theoretical ML)**
  - Probably approximately correct (PAC) learning and VC dimension

Data Representations

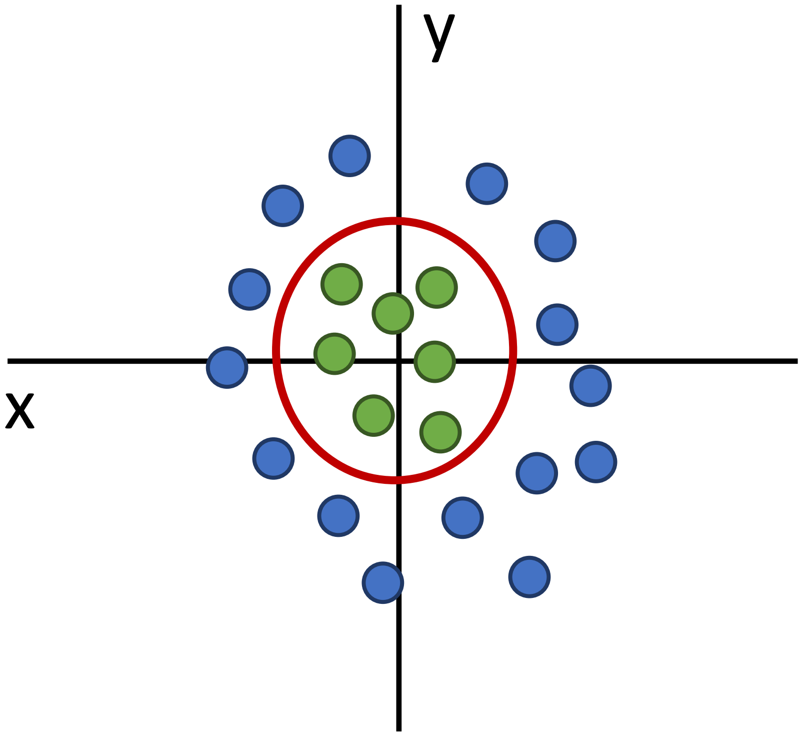
- The success of ML applications relies on having a **good representation** of the data.
  - Such that linear models perform well on top of the representation

- ML practitioners have put lots of efforts in **feature engineering**.
  - Based on domain expert's knowledge
    - E.g., Computer vision: color histogram, HoG, BoW, …
  - Time-consuming hand-tuning
  - (Arguably) a key limiting factor in advancing the state-of-the-arts

- Can we develop good representations **with less human effort**?

Beyond Linear Models: **Kernel Methods**

Original space

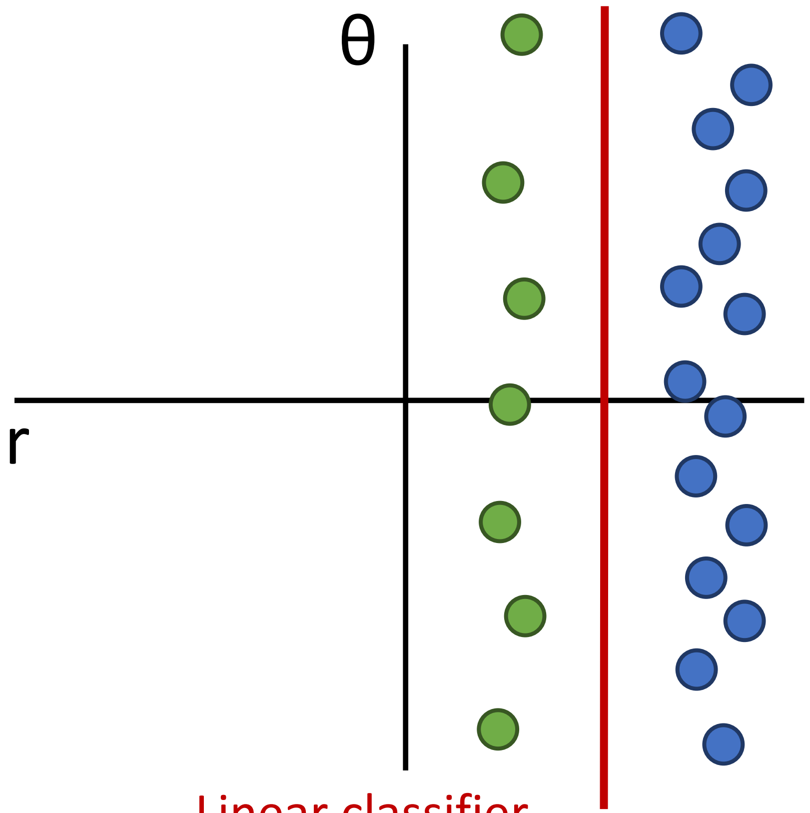


Nonlinear classifier
in original space!

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

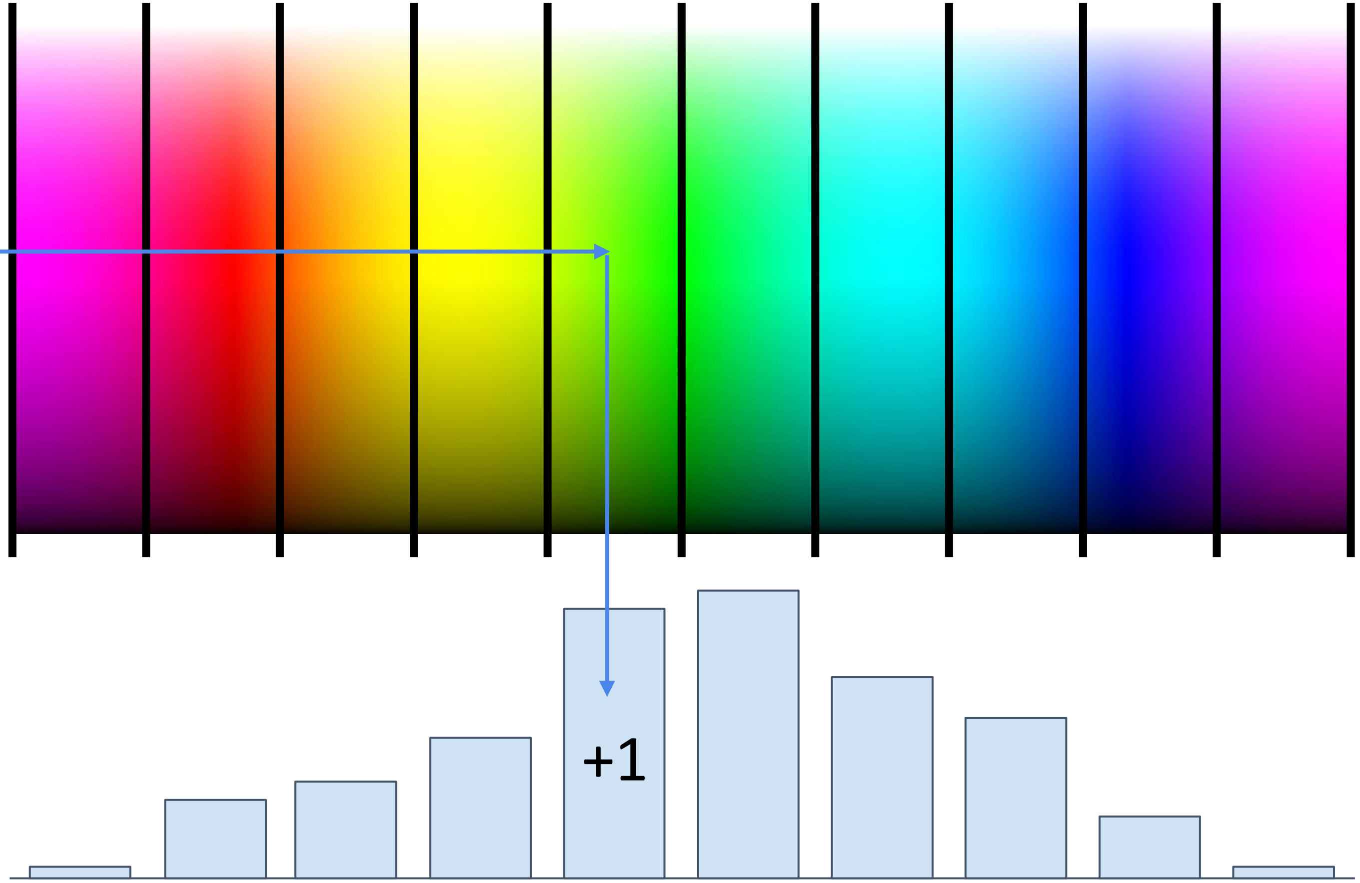
Feature transform

Feature space



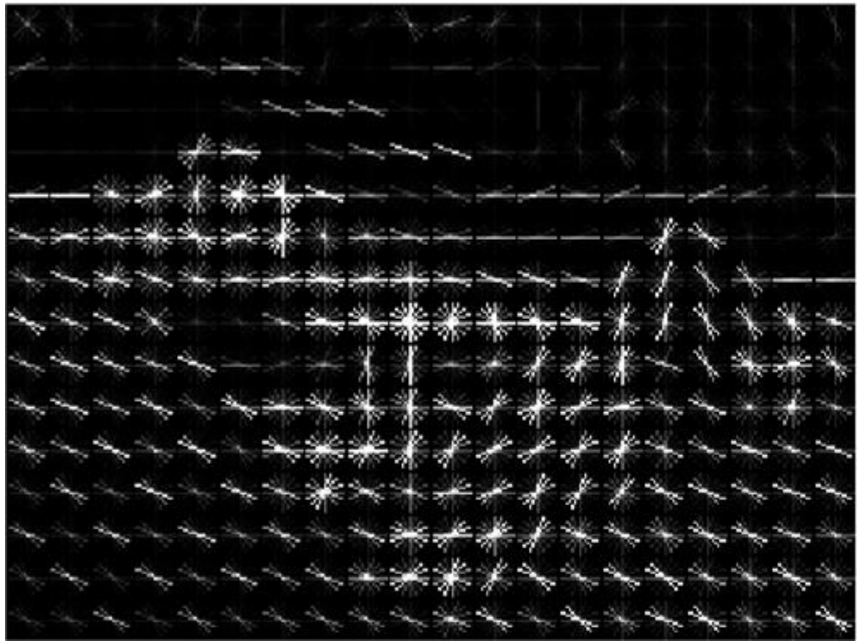
Linear classifier
in feature space

Image Features: Color Histogram



Ignores texture,
spatial positions

Image Features: Histogram of Oriented Gradients (HoG)

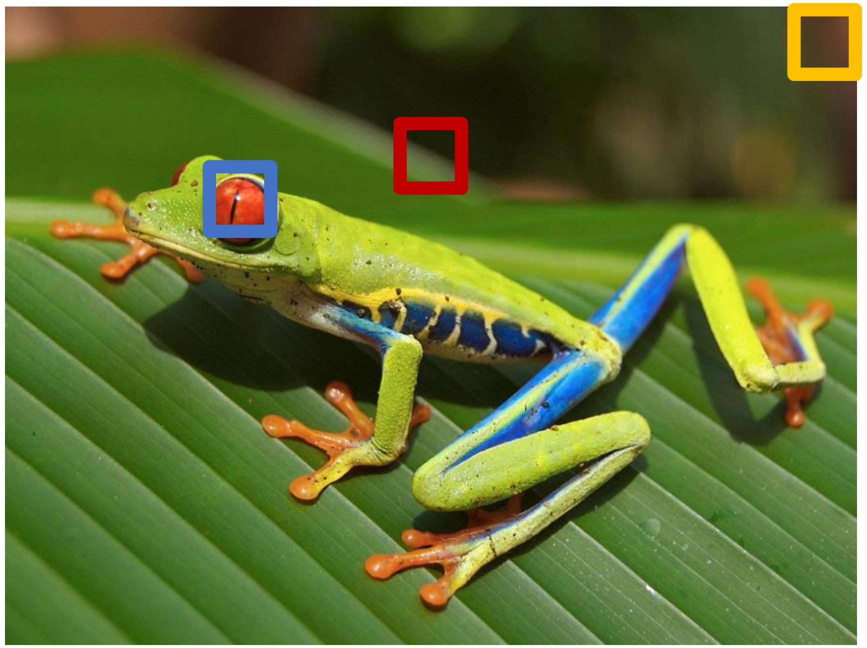


1. Compute edge direction / strength at each pixel
2. Divide image into 8x8 regions
3. Within each region compute a histogram of edge directions weighted by edge strength

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

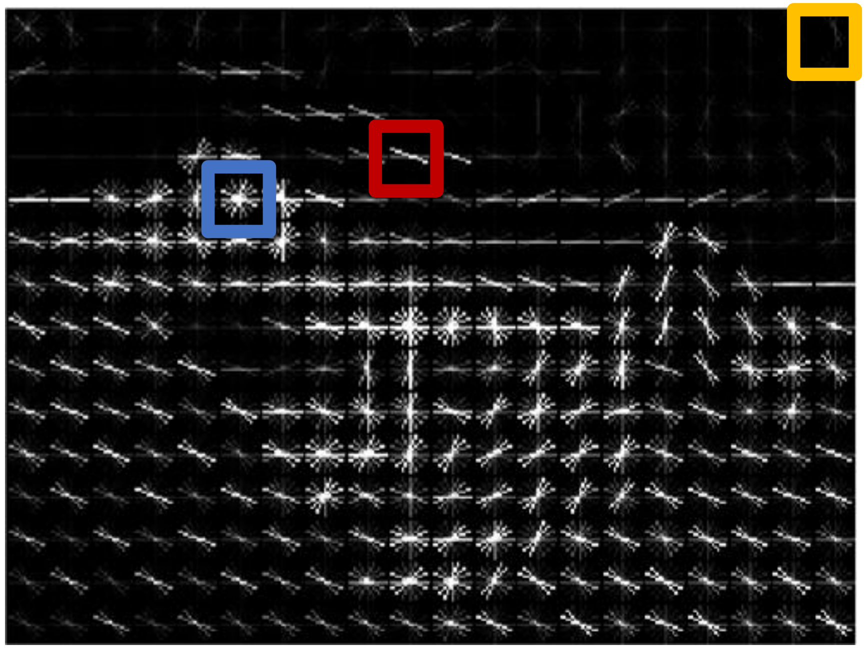
Image Features: Histogram of Oriented Gradients (HoG)



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

Captures texture and position, robust to small image changes

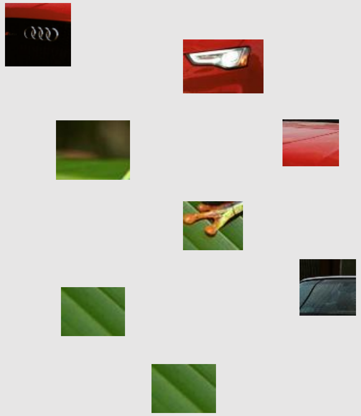
Example: 320x240 image gets divided into 40x30 bins; 8 directions per bin; feature vector has 30*40*9 = 10,800 numbers

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 (Data-Driven!)

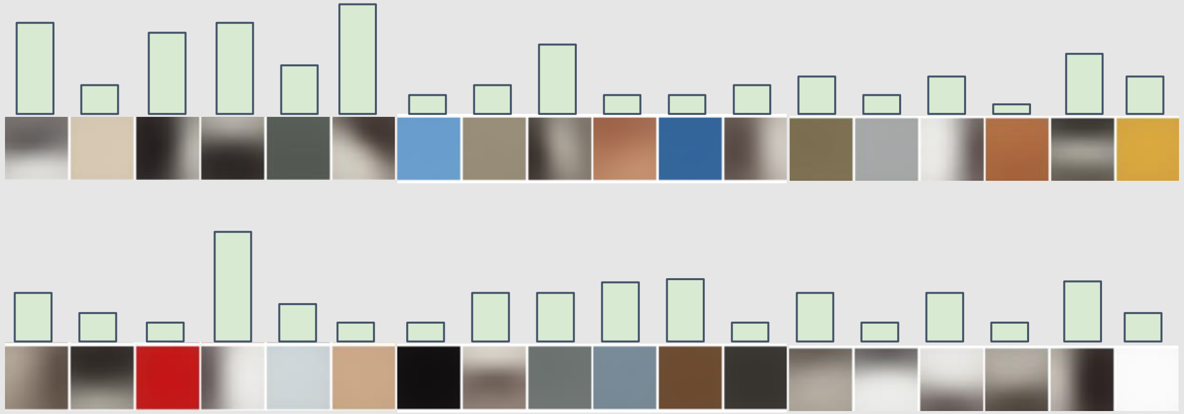
Step 1: Build codebook

Extract random patches



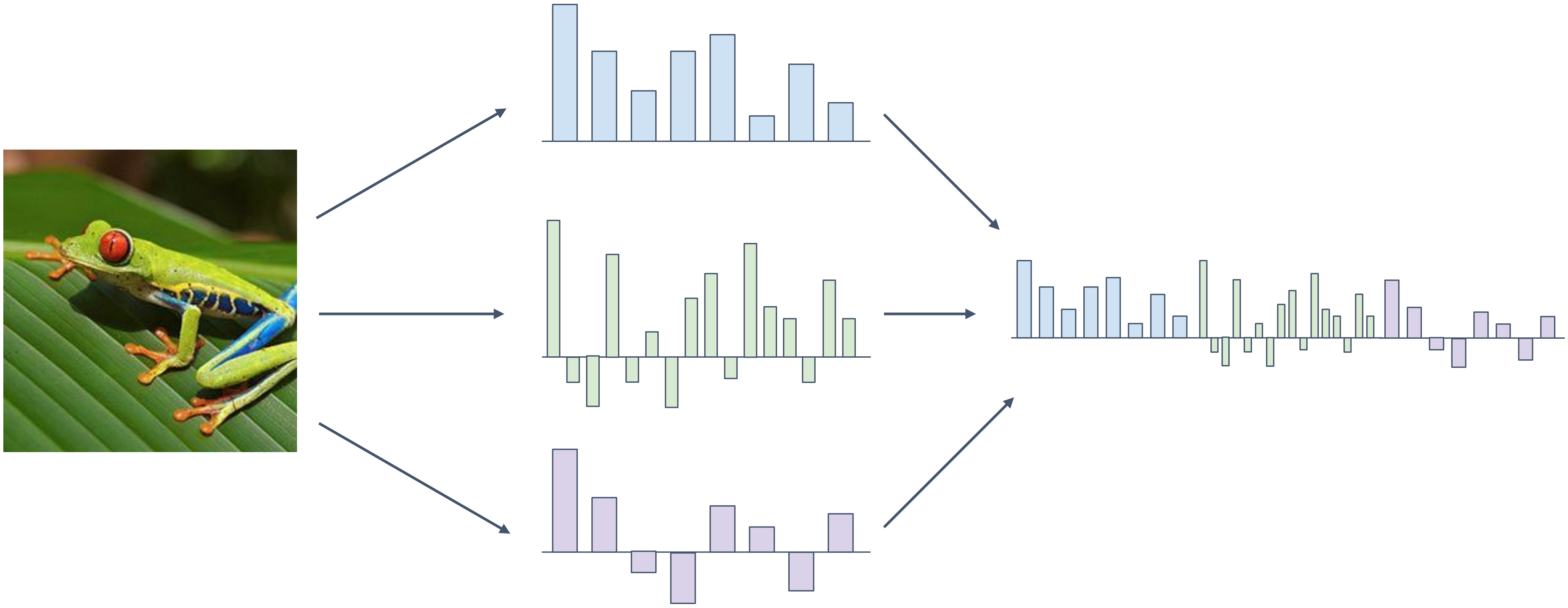
Cluster patches to form "codebook" of "visual words"

Step 2: Encode images



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

Image Features



Example: Winner of 2011 ImageNet challenge

Low-level feature extraction ≈ 10k patches per image

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

$\Big\}$ 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

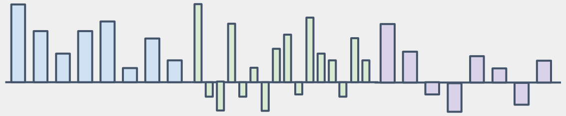
One-vs-all SVM learning with SGD

Late fusion of SIFT and color systems

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

Image Features

Feature Extraction



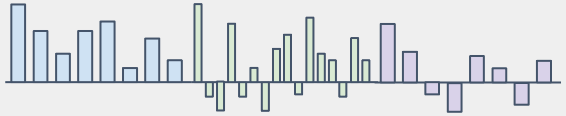
f

training

10 numbers giving scores for classes

Image Features vs. Neural Networks

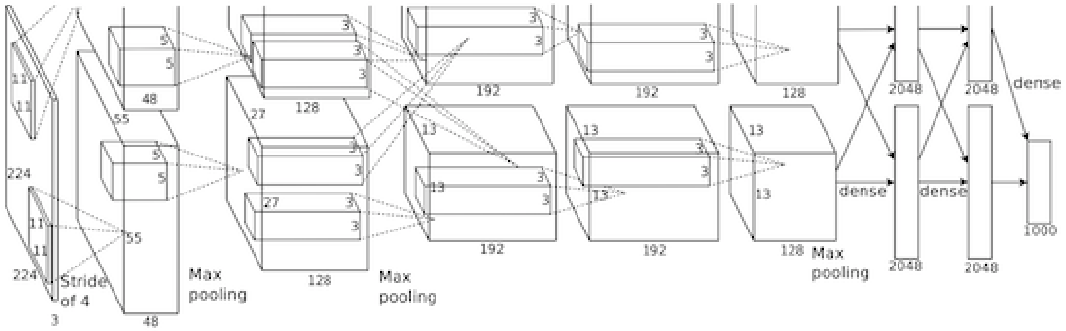
Feature Extraction



f

10 numbers giving scores for classes

training



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

10 numbers giving scores for classes

training

Neural Networks

Linear classifiers

This image is CC0 1.0 public domain

Neural Networks

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

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

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

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

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

Neural Networks

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

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

Feature Extraction
Linear Classifier

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

Neural Networks

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

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

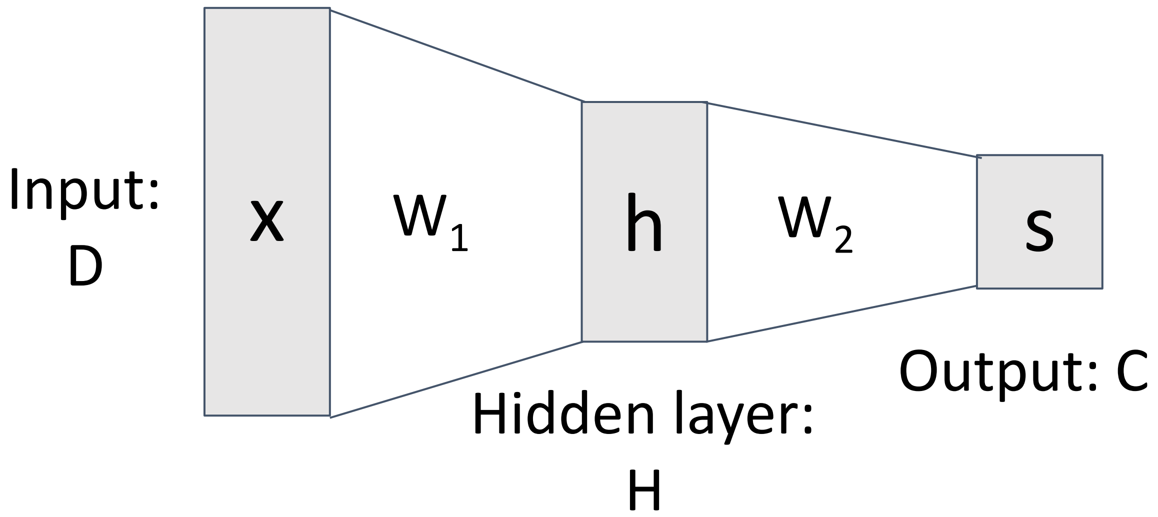
Now: Two-Layer Neural Network: $f(x) = W_2\, g(W_1 x + b_1) + b_2$
Learnable parameters: $W_1 \in \mathbb{R}^{H \times D}, b_1 \in \mathbb{R}^H, W_2 \in \mathbb{R}^{C \times H}, b_2 \in \mathbb{R}^C$

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

Neural Networks

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

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



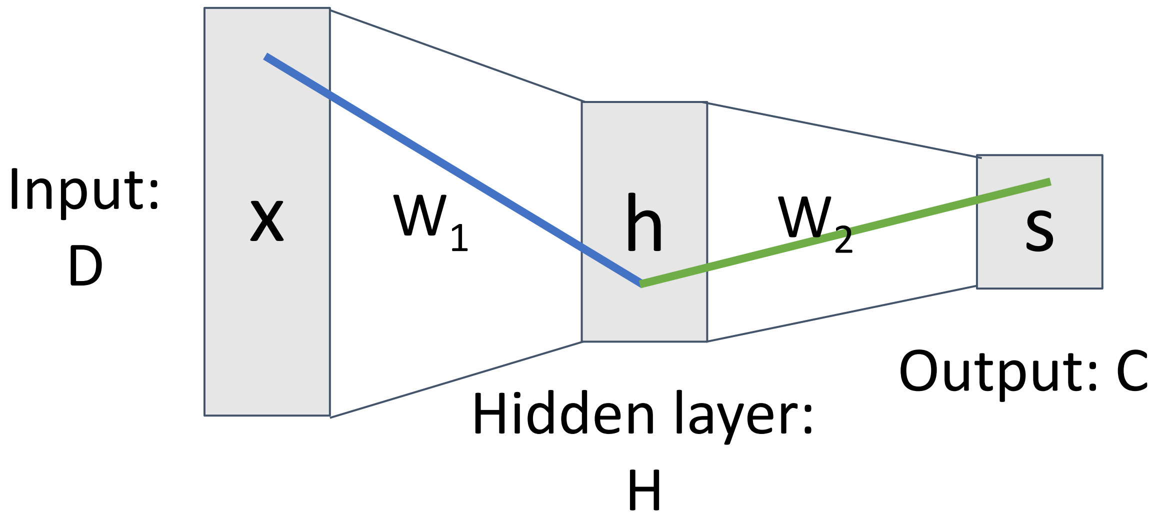
$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

Neural Networks

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

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

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

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

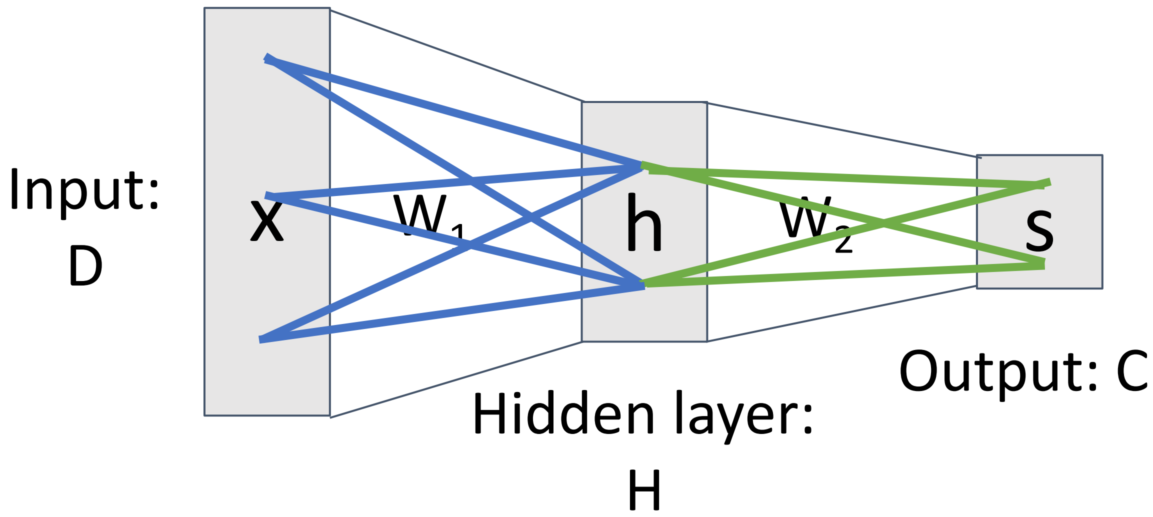
Neural Networks

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

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

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

All elements of x affect all elements of h

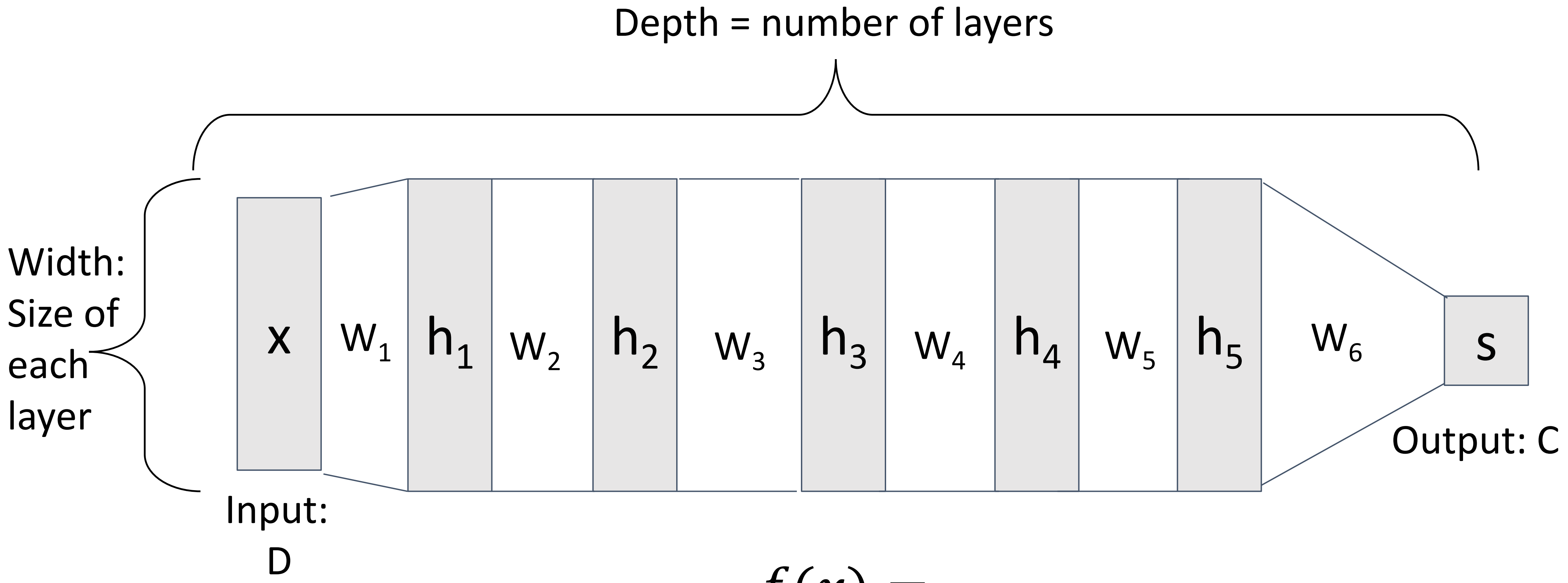


Element (i, j) of W_2 gives the effect on s_i from h_j

All elements of h affect all elements of s

Fully-connected neural network
a.k.a. "Multi-Layer Perceptron" (MLP)

Deep Neural Networks



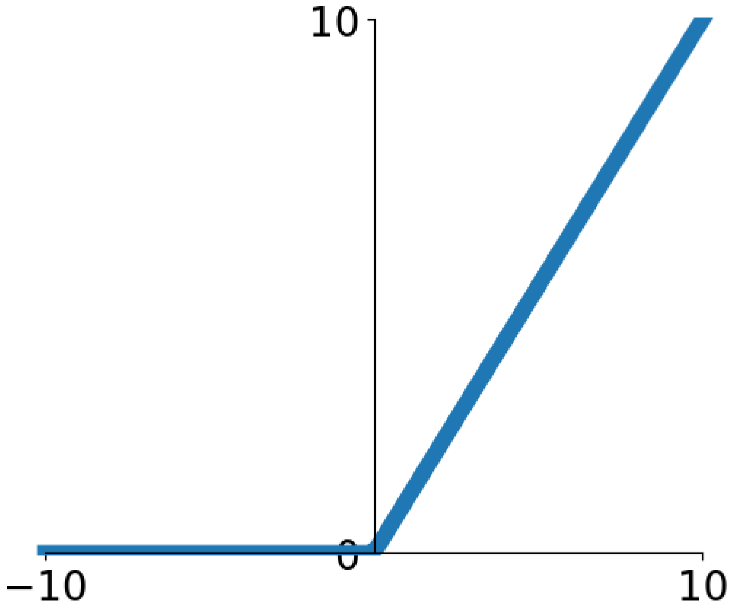
$$f(x) =$$
$$W_6\, g(W_5\, g(W_4\, g(W_3\, g(W_2\, g(W_1 x + b_1) + b_2) + b_3) + b_4) + b_5) + b_6$$

Activation Functions

2-layer Neural Network

$$f(x) = W_2 \boxed{g}(W_1 x + b_1) + b_2$$

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?

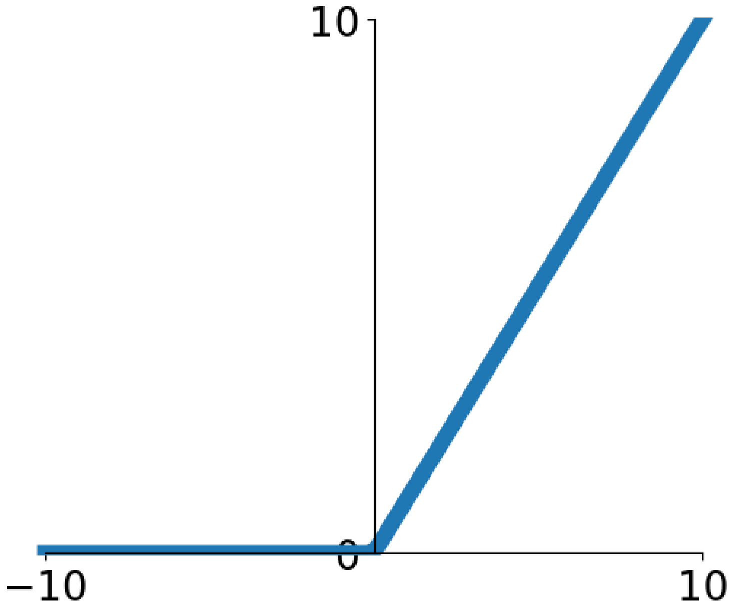
$$f(x) = W_2(W_1 x + b_1) + b_2$$

Activation Functions

2-layer Neural Network

$$f(x) = W_2 \boxed{g}(W_1 x + b_1) + b_2$$

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?

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

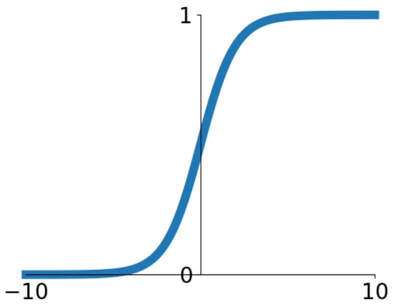
A: We end up with a linear classifier!

Activation Functions

ReLU is a good default choice for most problems

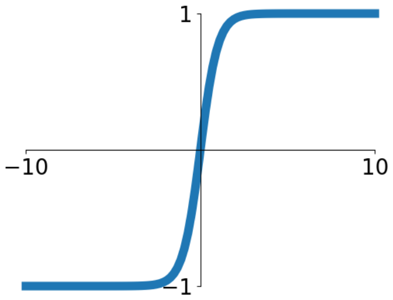
Sigmoid

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



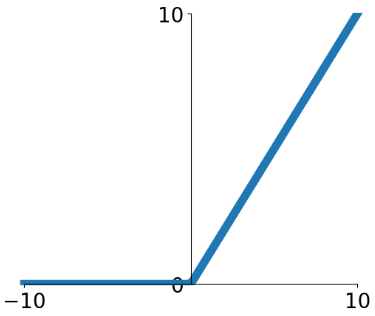
tanh

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



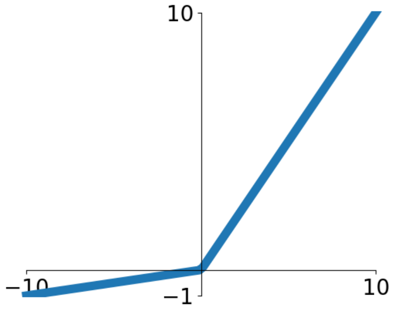
ReLU

$$\max(0, x)$$



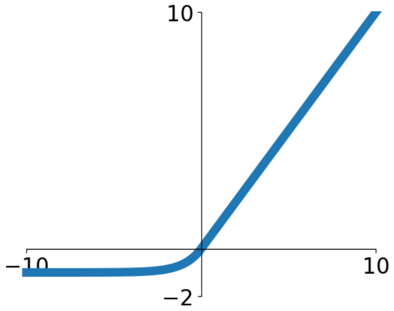
Leaky ReLU

$$\max(0.1x, x)$$



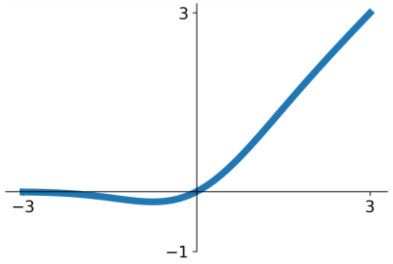
ELU

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



GELU

$$= 0.5x\left[1 + \mathrm{erf}(x/\sqrt{2})\right]$$
$$\approx x\sigma(1.702x)$$



Recap: Cross-Entropy Loss (Softmax Regression)

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

Softmax probability vector: $p(y|x) \in [0,1]^C$

$\boldsymbol{k}$**-th class probability**: $p(y = k|x) = \frac{\exp(s_k)}{\sum_j \exp(s_j)} \in [0,1]$

Cross-Entropy Loss for a training data $\left(x^{(i)}, y^{(i)}\right)$:

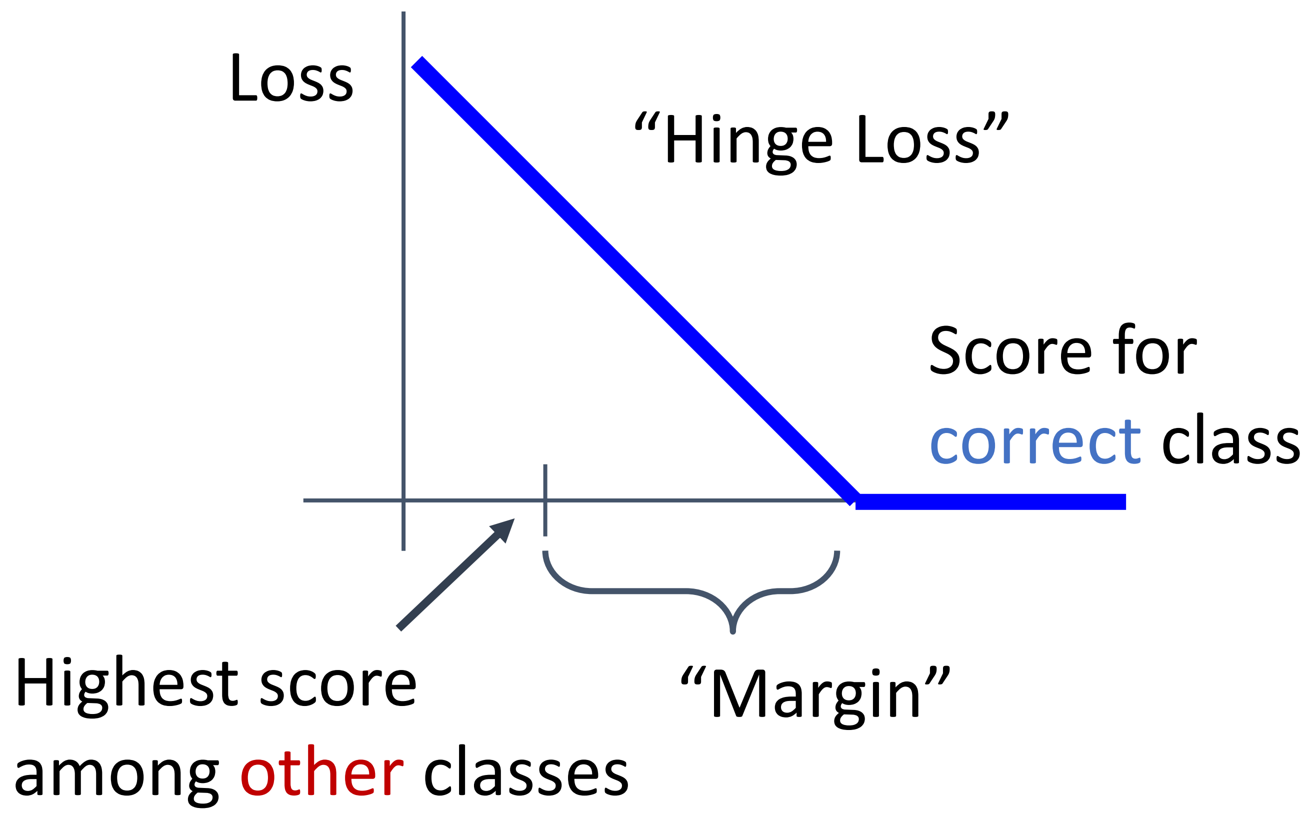
$$L_i = -\log p\left(y = y^{(i)}|x^{(i)}\right) = -\log \frac{\exp\left(s_{y^{(i)}}\right)}{\sum_j \exp\left(s_j\right)}$$

Note: we don't need softmax at test time; simply take

$$y = \underset{k}{\operatorname{argmax}}\, f_k(x)$$

Recap: Multiclass SVM Loss (Hinge Loss)

"The score of the correct class should be higher than all the other scores"



Given a training data $\left(x^{(i)}, y^{(i)}\right)$

Let $s^{(i)} = f\left(x^{(i)}\right)$ be scores.

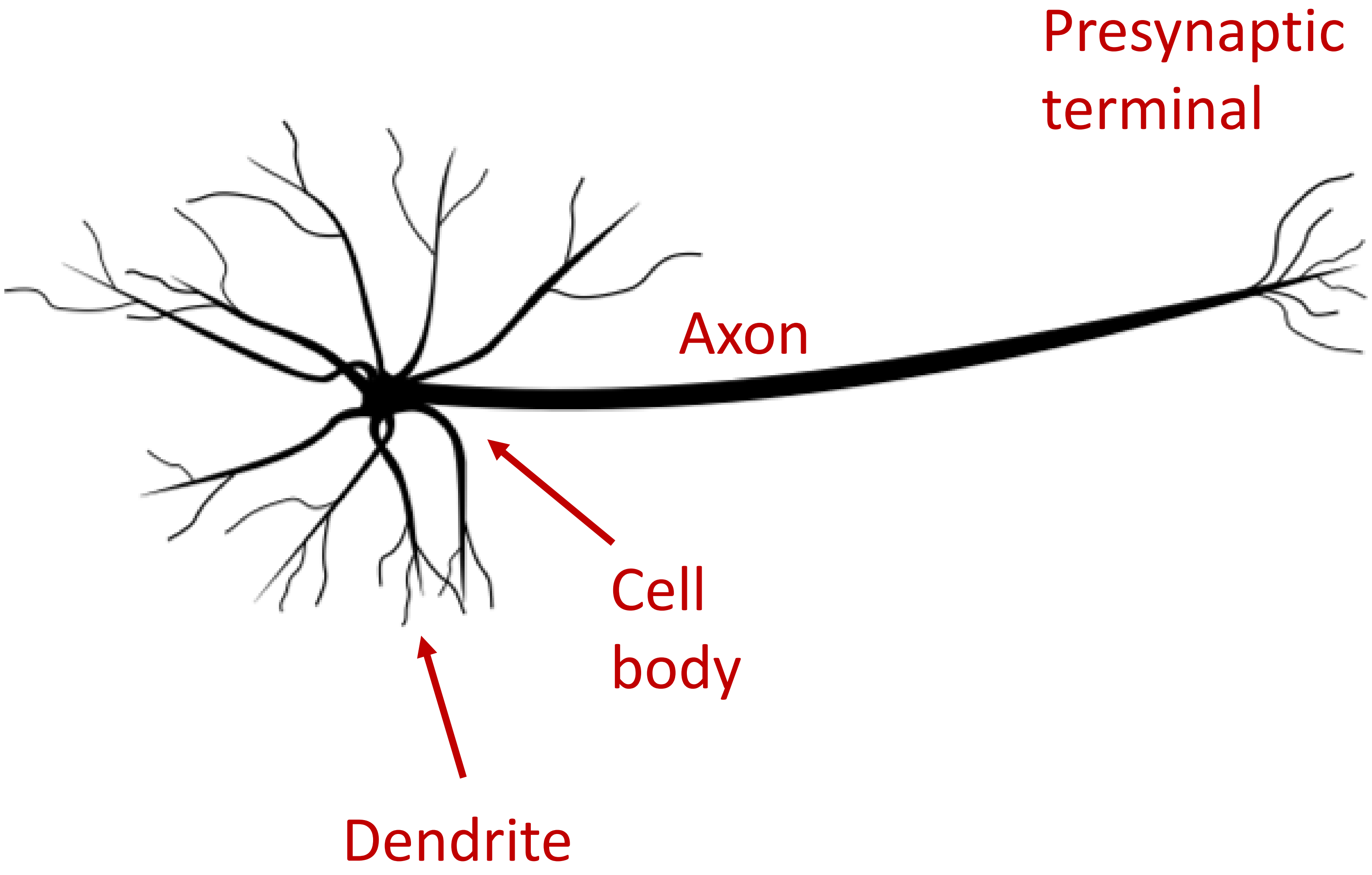
The SVM loss has the form:

$$L_i = \sum_{j \neq y^{(i)}} \max\left(0, s_j^{(i)} - s_{y^{(i)}}^{(i)} + 1\right)$$

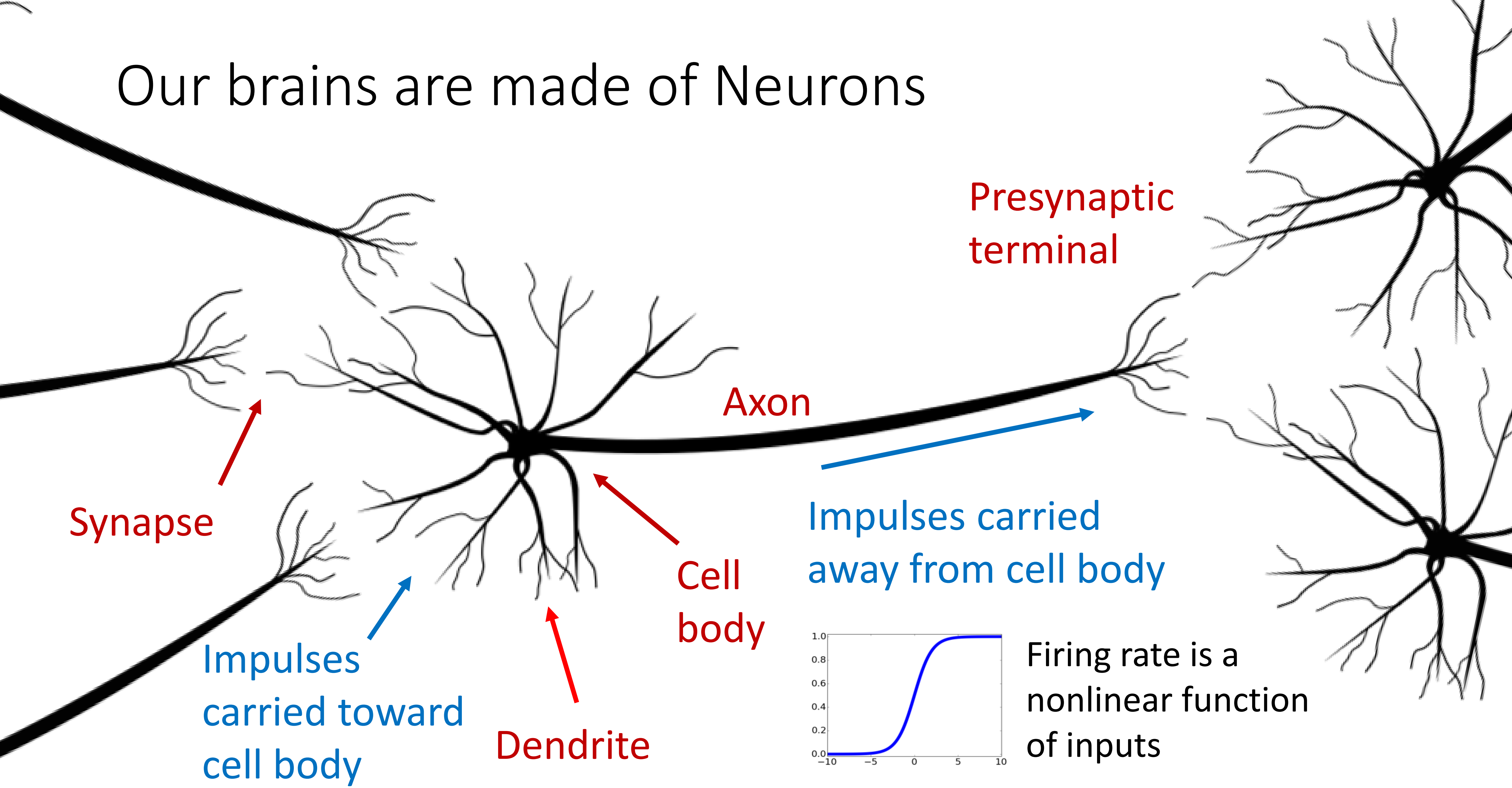
Convexity

- Most linear classifiers optimize a convex function
  - Linear layer $s = f(x; W) = Wx$
  - Cross-entropy loss $L_i = -\log(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}})$
  - SVM $L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$
  - L1/L2 regularization $L = \frac{1}{N} \sum_{i=1}^{N} L_i + R(W)$
- Most neural networks need non-convex optimization
  - Few or no guarantees about convergence (mostly falls in a local optimum)
  - Empirically it seems to work anyway
  - Active area of research

Our brains are made of Neurons

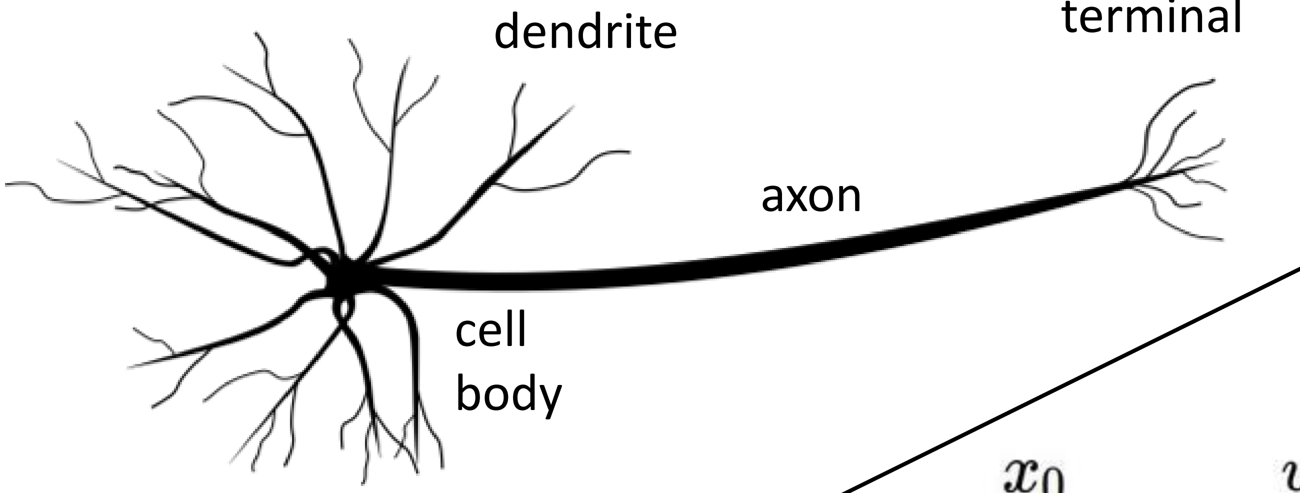


Our brains are made of Neurons

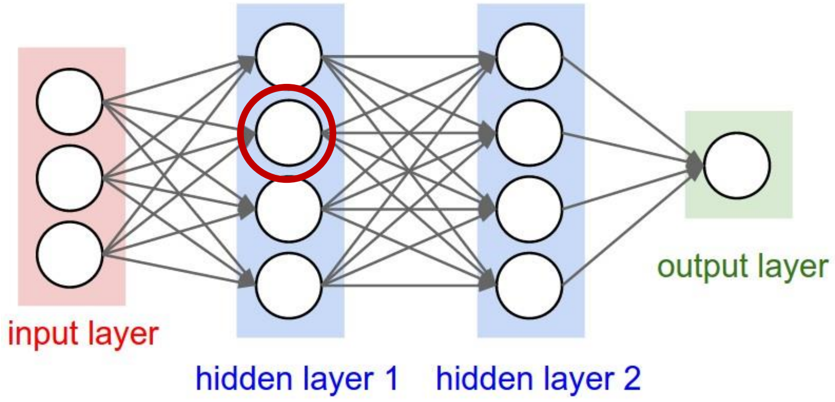


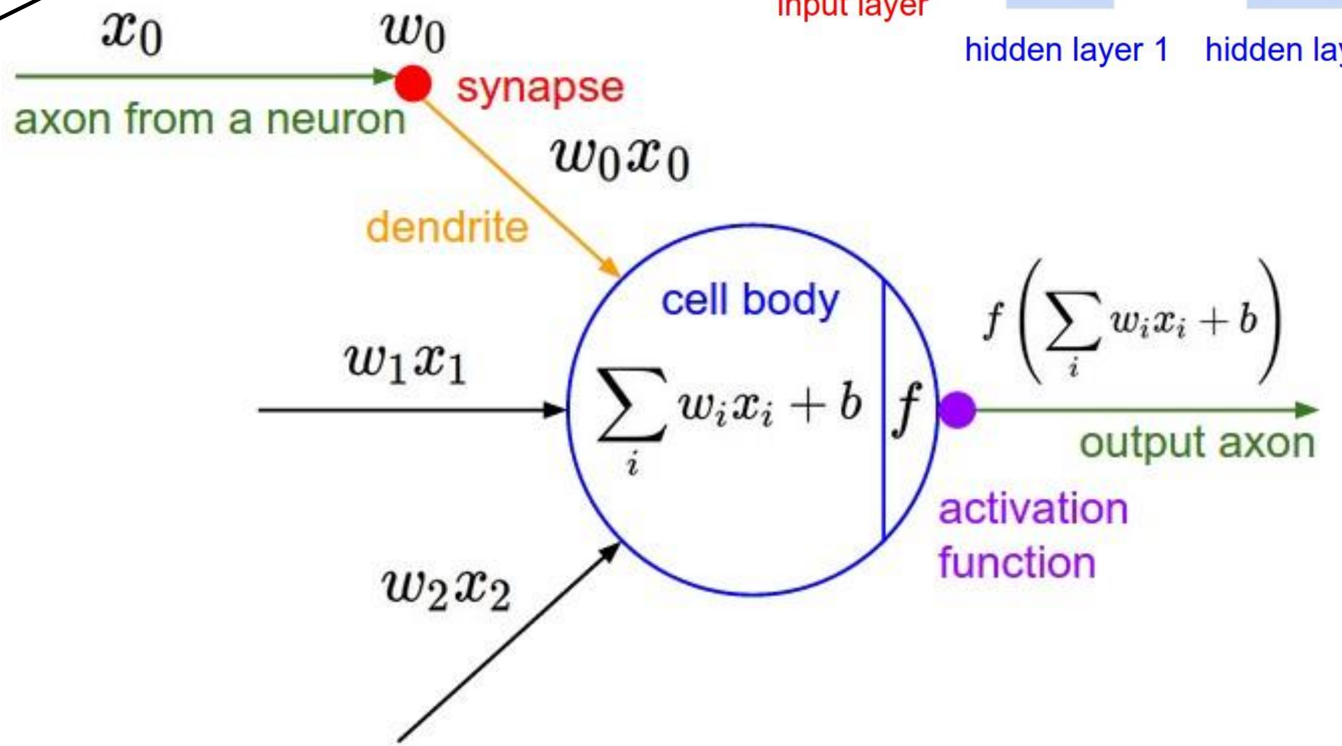
Firing rate is a nonlinear function of inputs

Biological Neuron

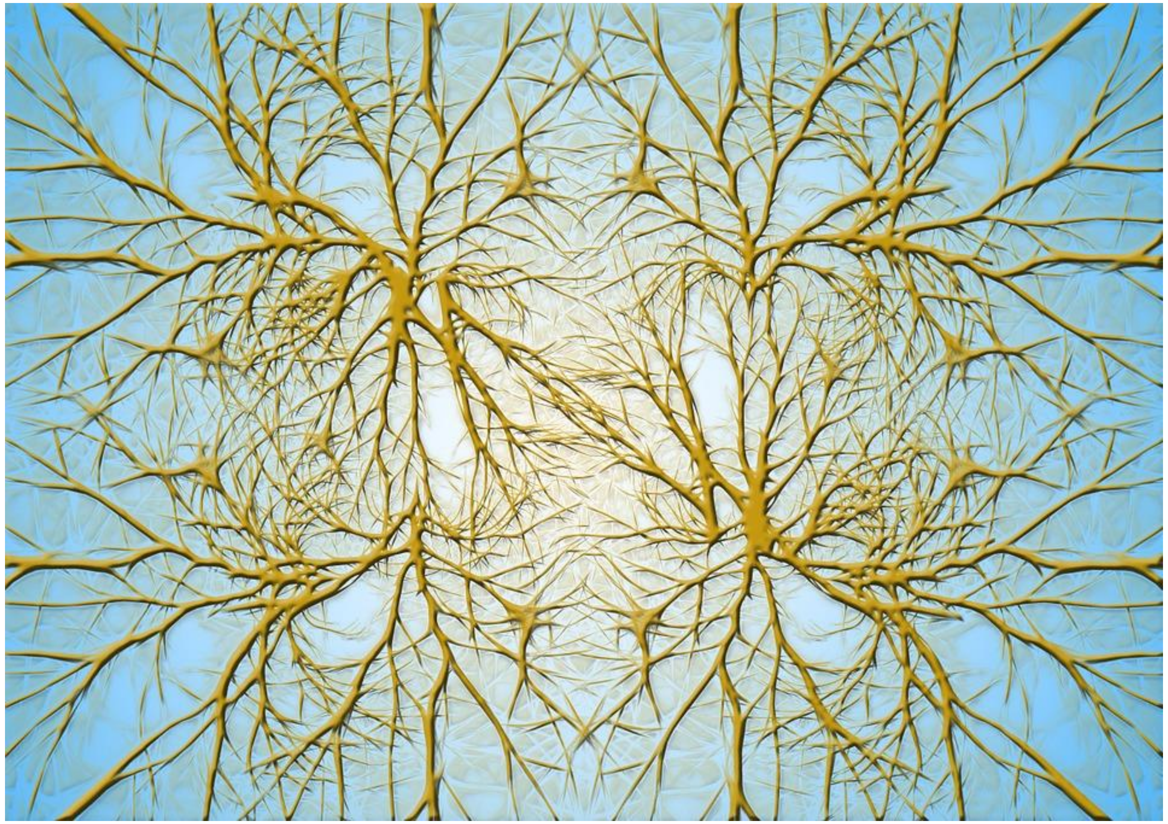


Artificial Neuron



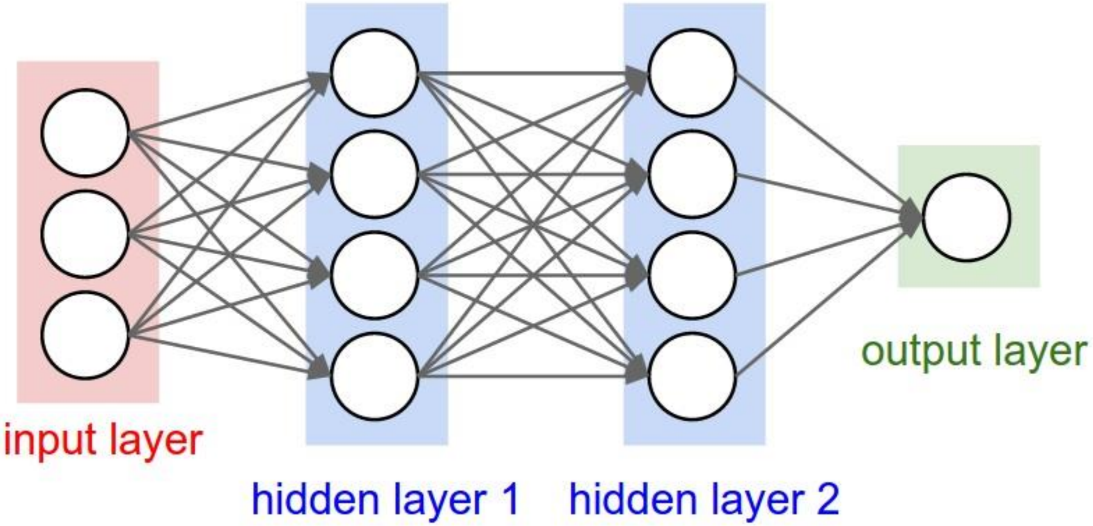


Biological Neurons:
Complex connectivity patterns

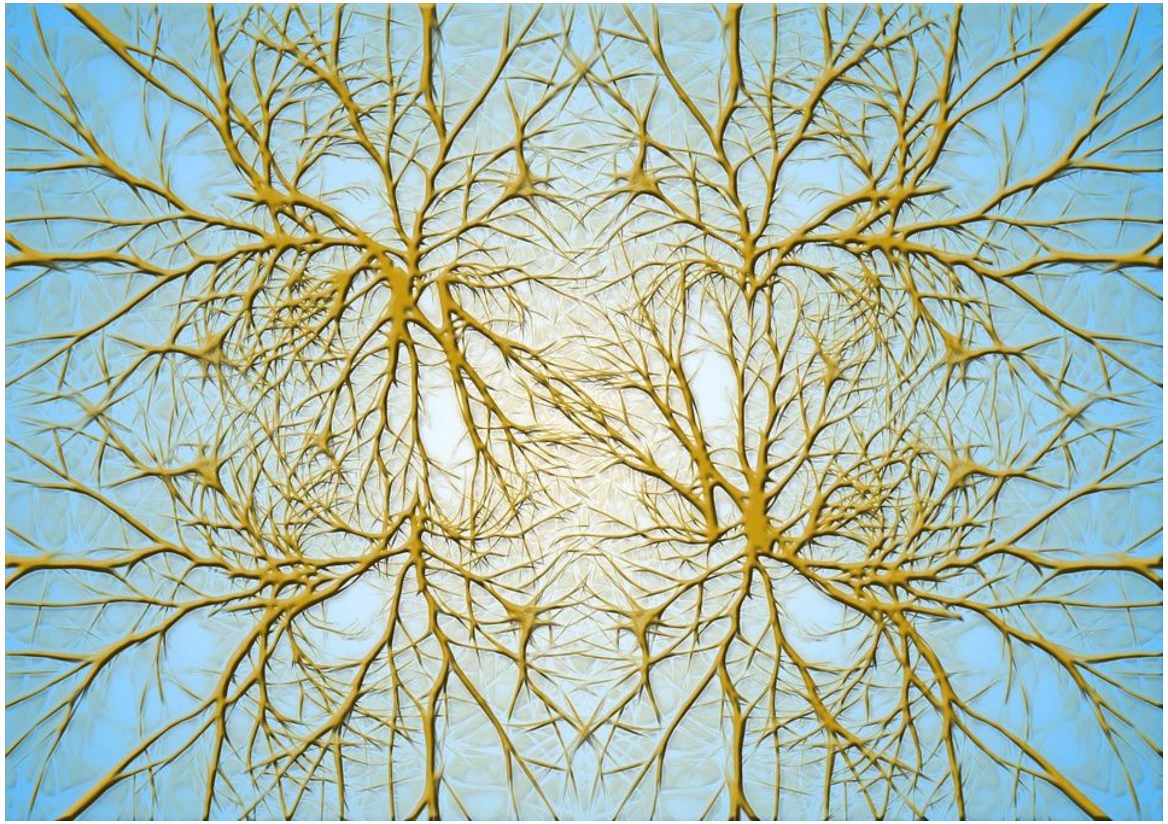


This image is CC0 Public Domain

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

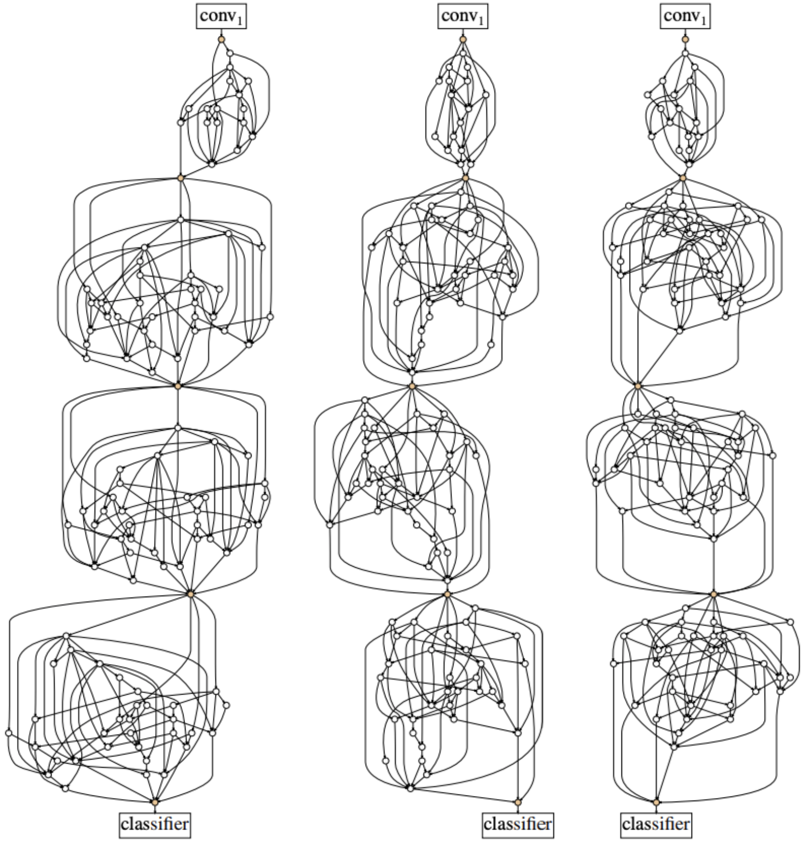


Biological Neurons:
Complex connectivity patterns



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

Be very careful with brain analogies!

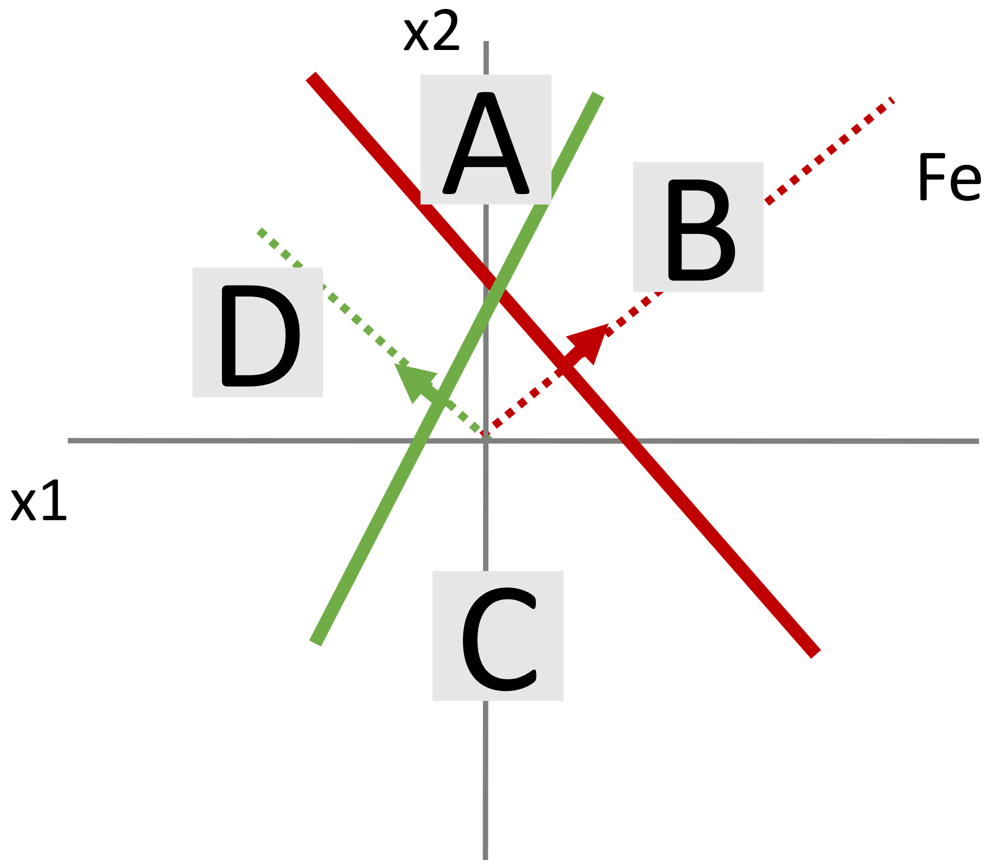
Biological Neurons:

- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system
- Abstracting a neuron by “firing rate” isn’t enough; temporal sequences of activations matter too (spiking neural networks)

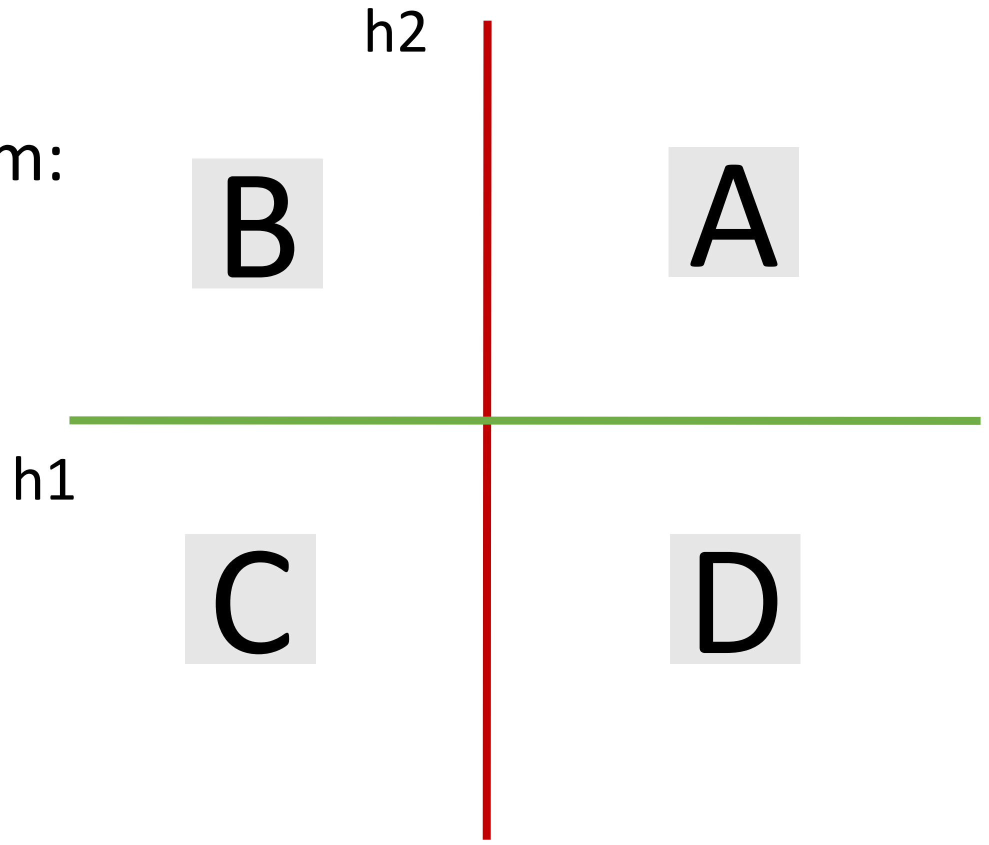
[Dendritic Computation. London and Hausser]

Space Warping

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



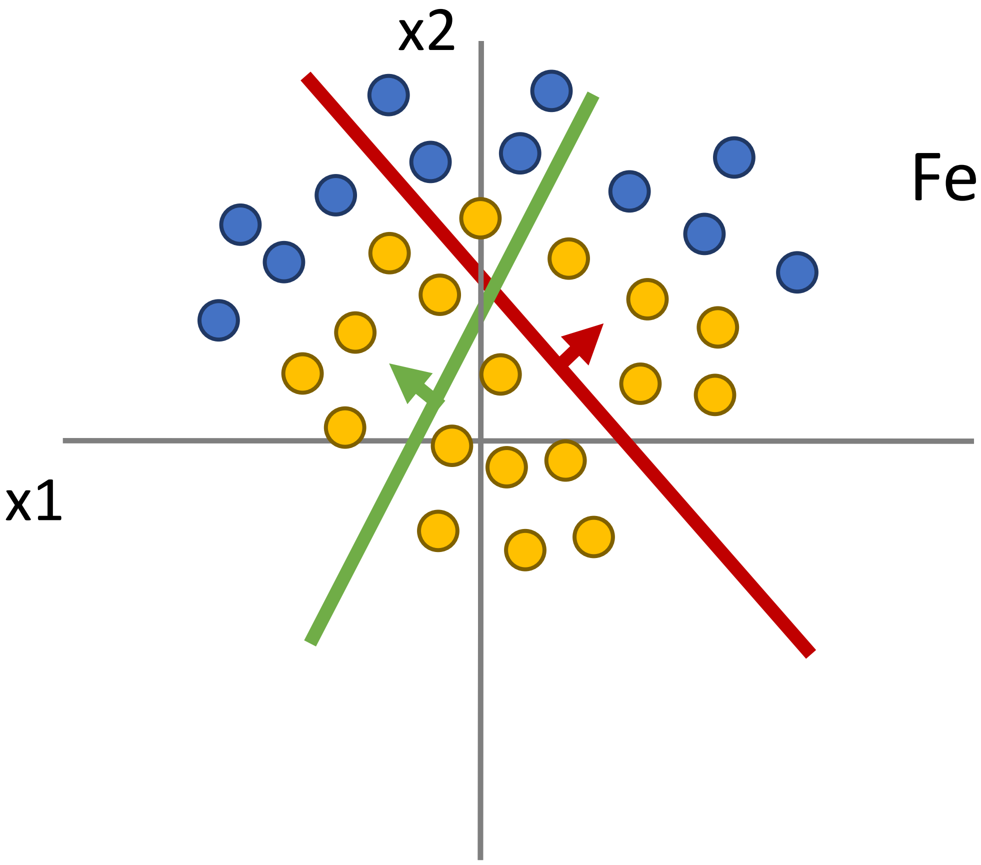
Feature transform:
$h = Wx$



Space Warping

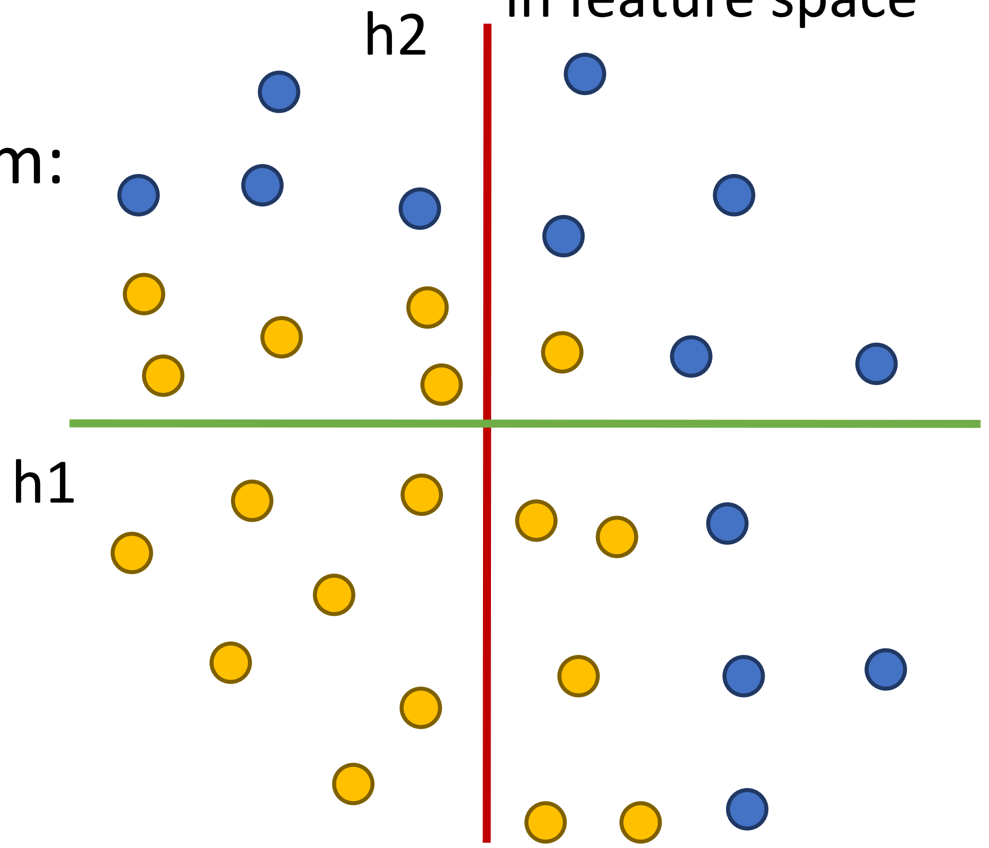
Consider a linear transform: h = Wx
Where x, h are both 2-dimensional

Points not linearly separable in original space



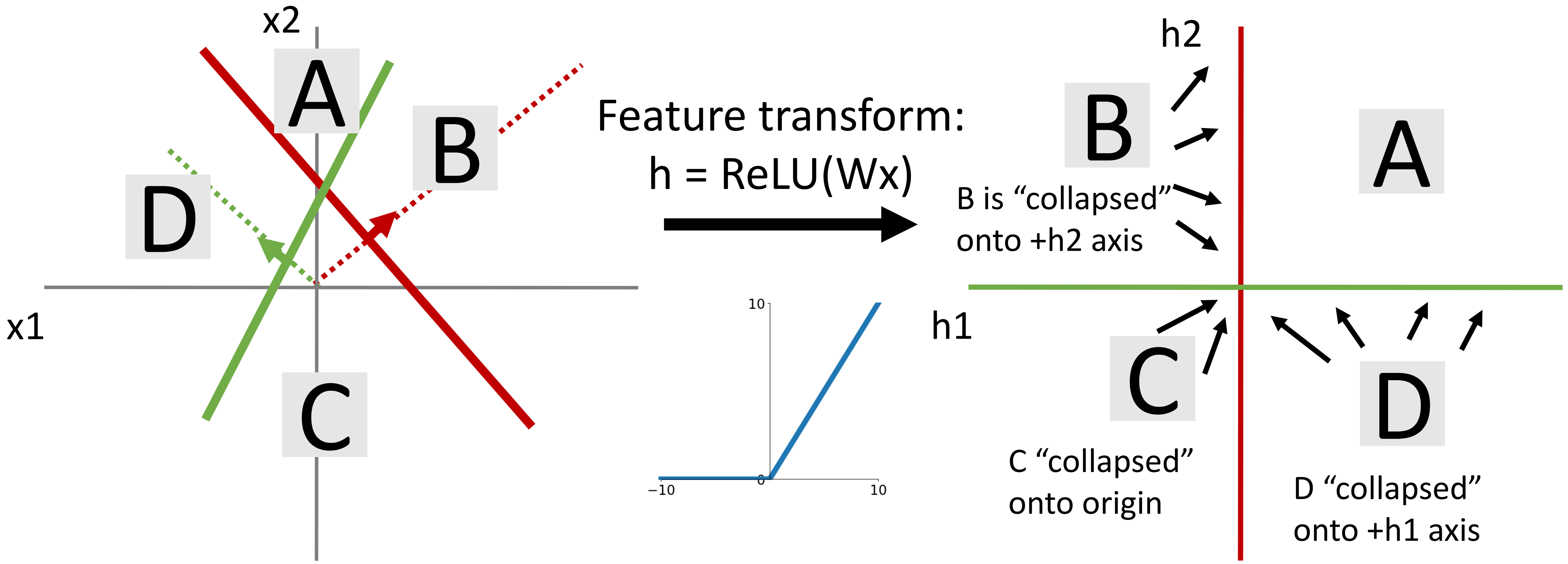
Feature transform:
h = Wx

Not linearly separable in feature space



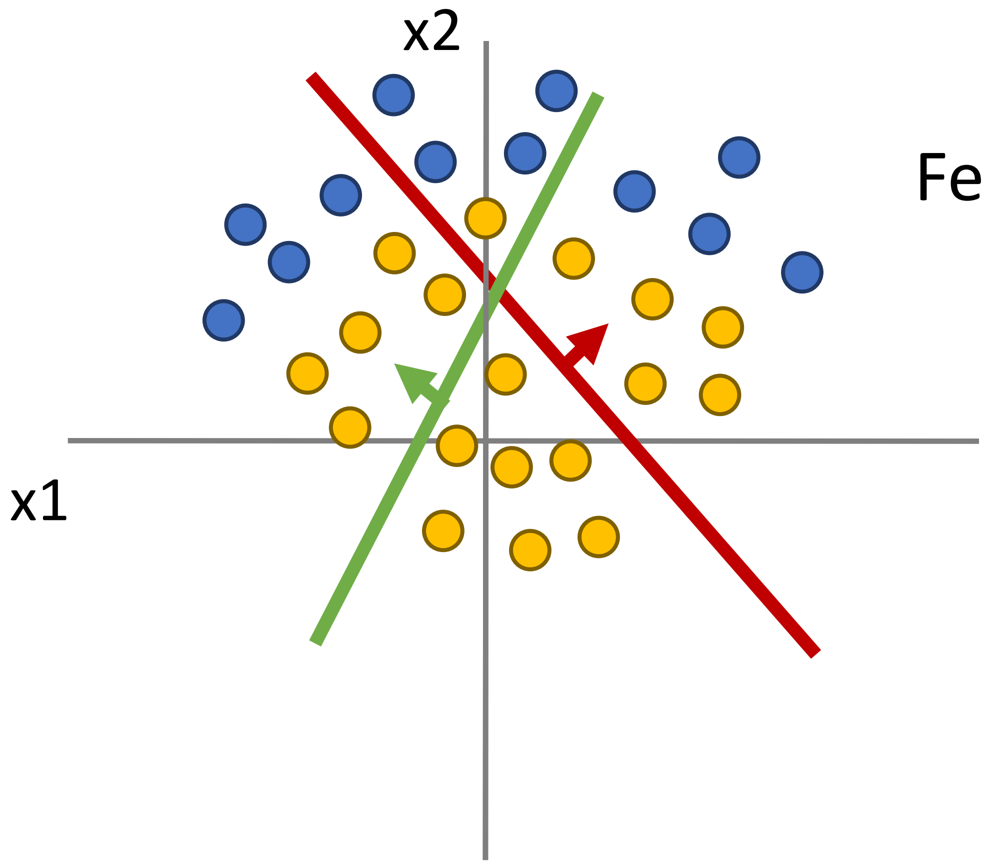
Space Warping

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



Space Warping

Points not linearly separable in original space



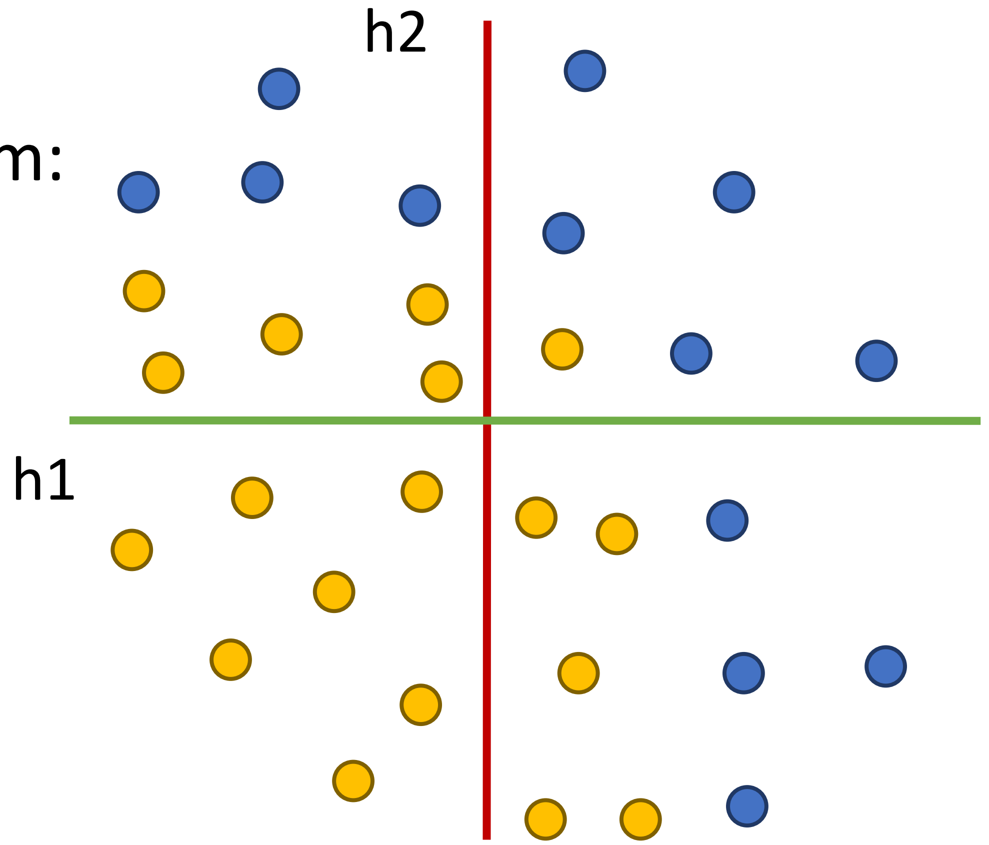
Consider a neural net hidden layer:

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

Where x, h are both 2-dimensional

Feature transform:

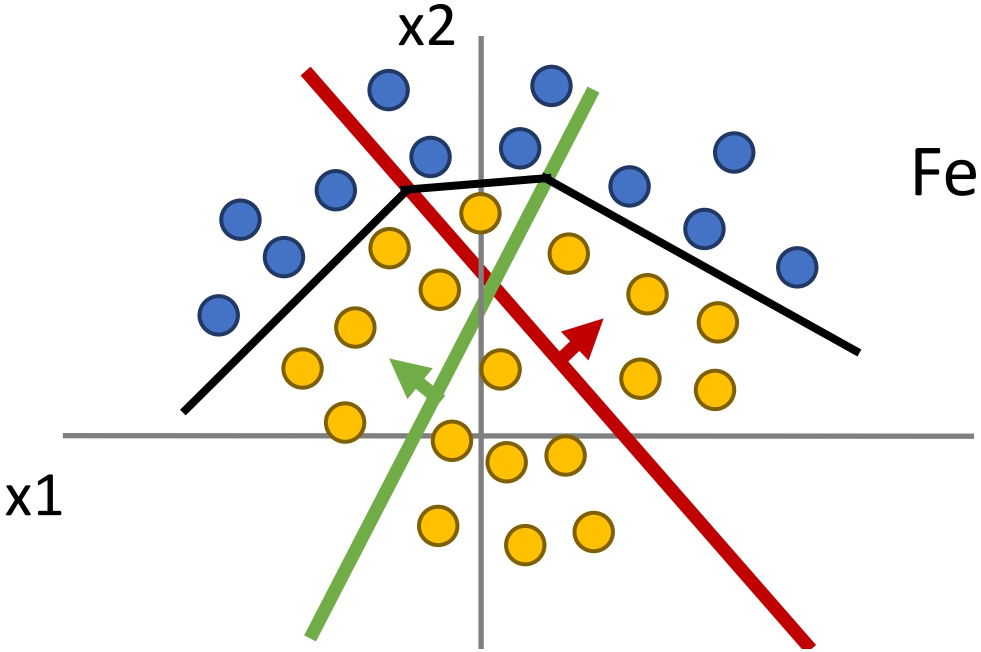
h = Wx



Space Warping

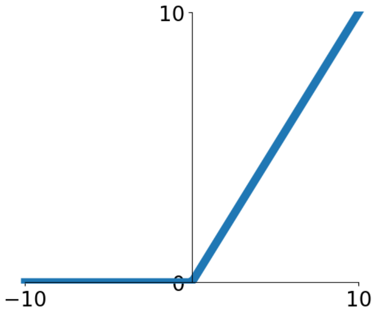
Consider a neural net hidden layer:
$h = \text{ReLU}(Wx) = \max(0, Wx)$
Where x, h are both 2-dimensional

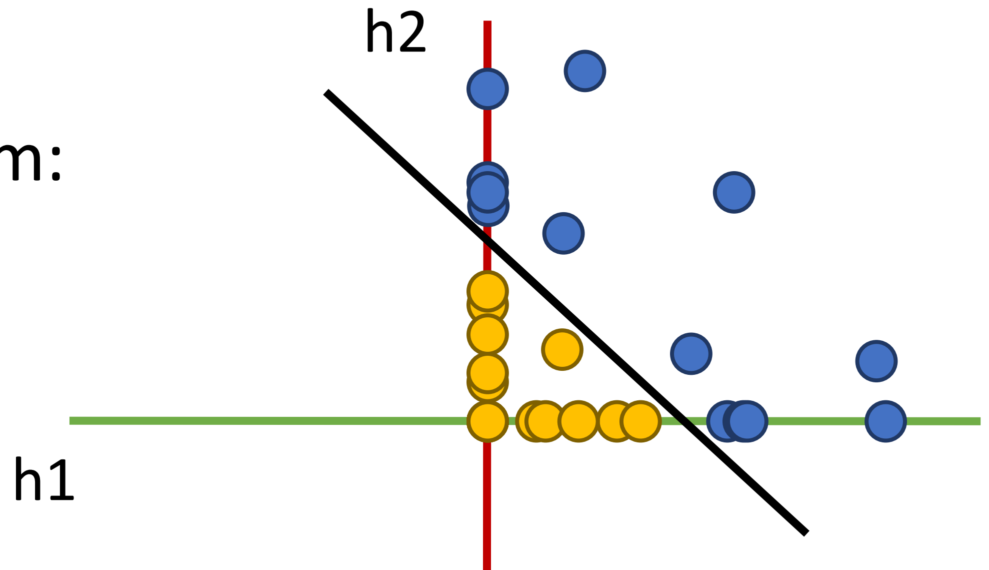
Points not linearly separable in original space



Linear classifier in feature space gives nonlinear classifier in original space

Feature transform:
$h = \text{ReLU}(Wx)$

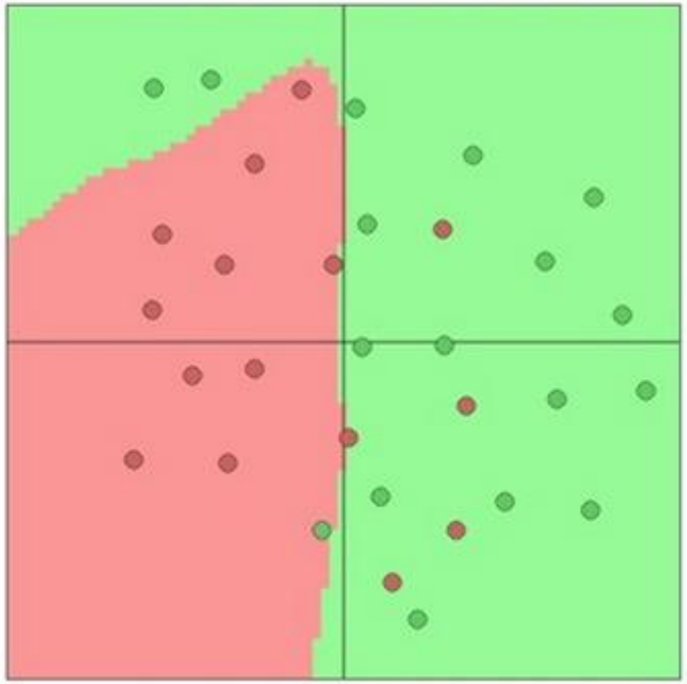




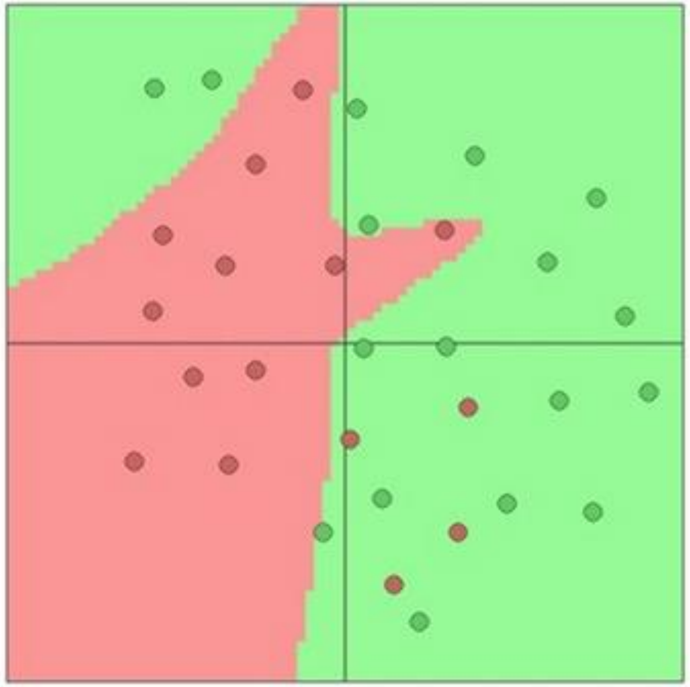
Points are linearly separable in features space!

Setting the number of layers and their sizes

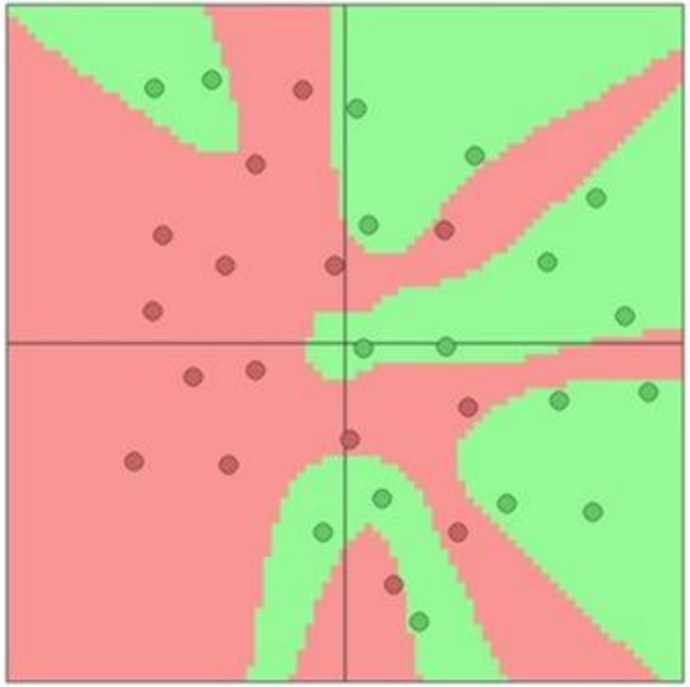
3 hidden units



6 hidden units

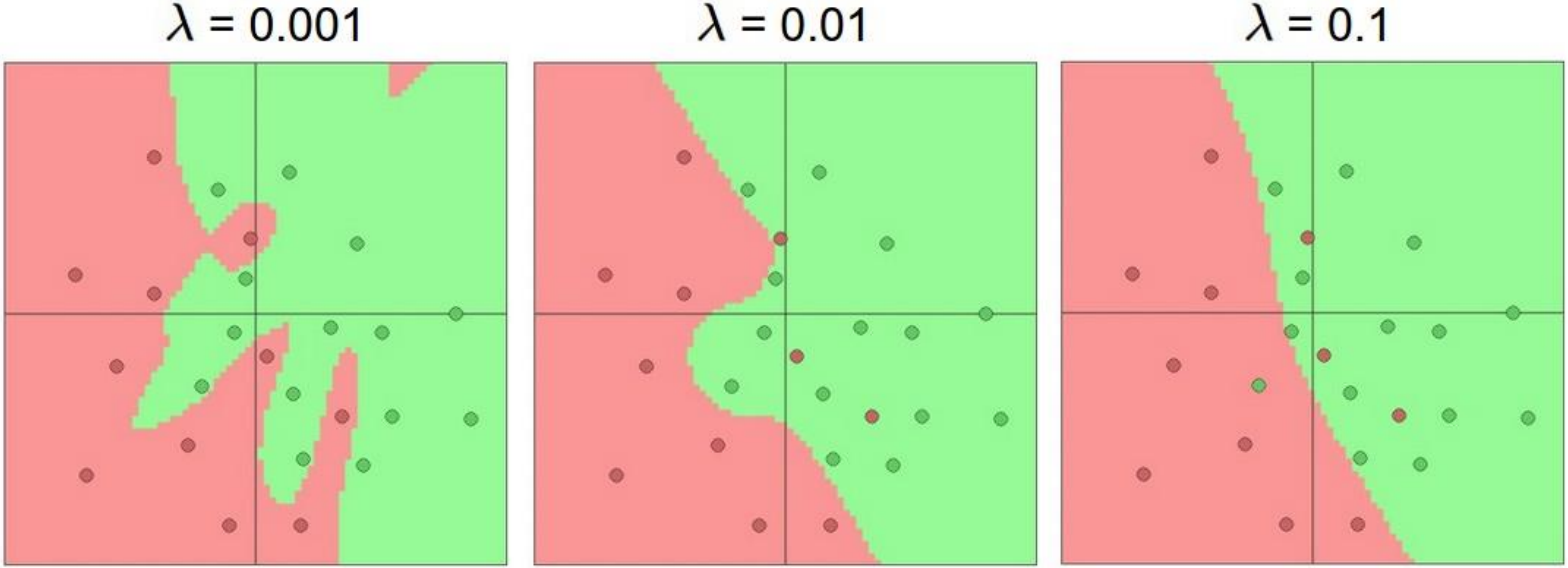


20 hidden units



More hidden units = more capacity

Don’t regularize with size; instead use stronger L2



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

Dropout as Regularization

Regularization: Add term to the loss

$$L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \boxed{\lambda R(W)}$$

In common use:

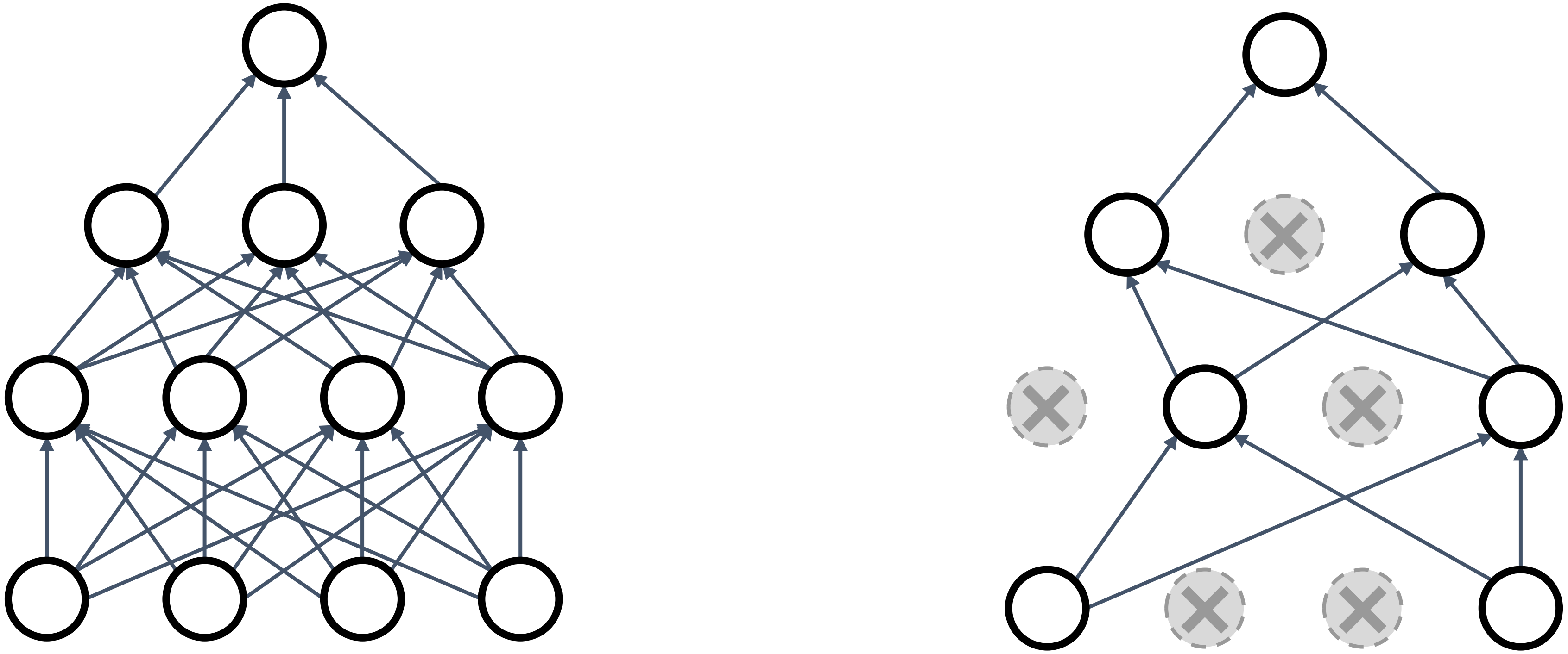
L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$ (Weight decay)

L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2) $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

Dropout

In each forward pass, randomly set some neurons to zero
Probability of dropping is a hyperparameter; 0.5 is common

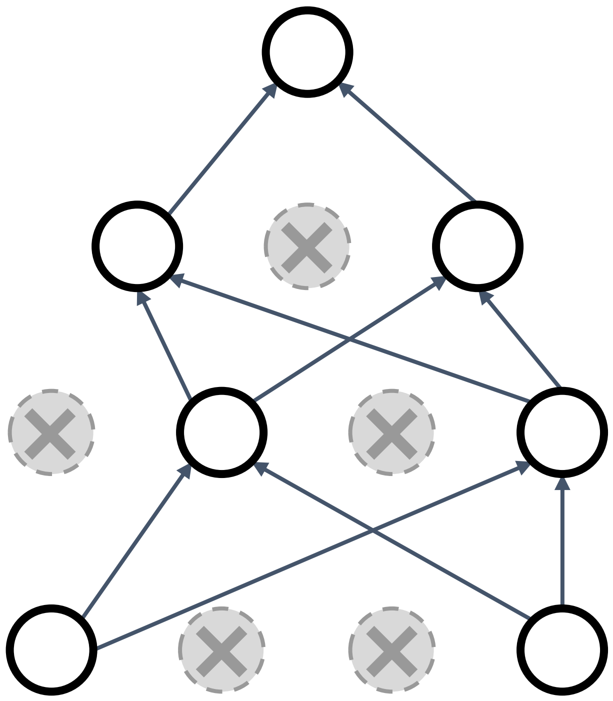


Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

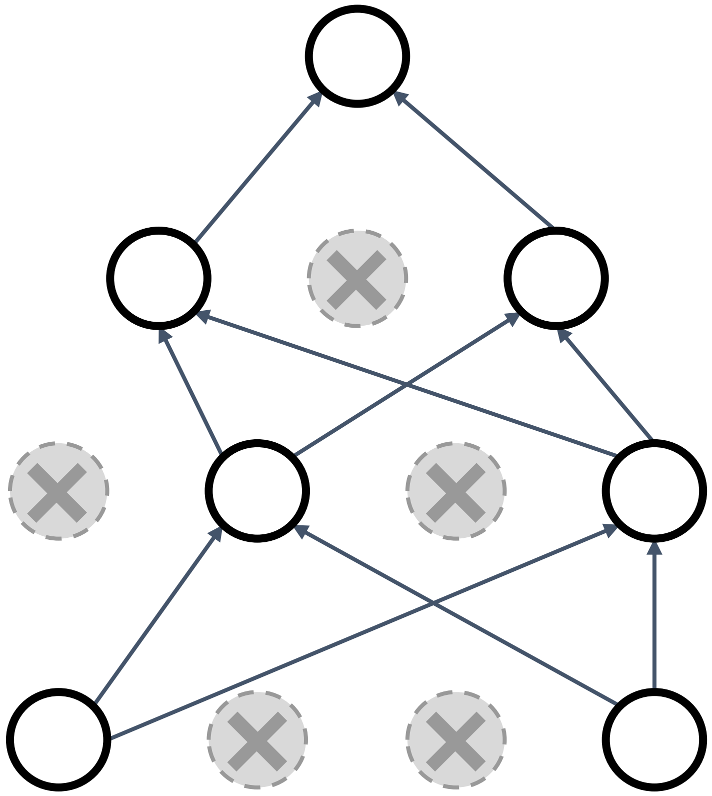
Dropout

```
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)
```

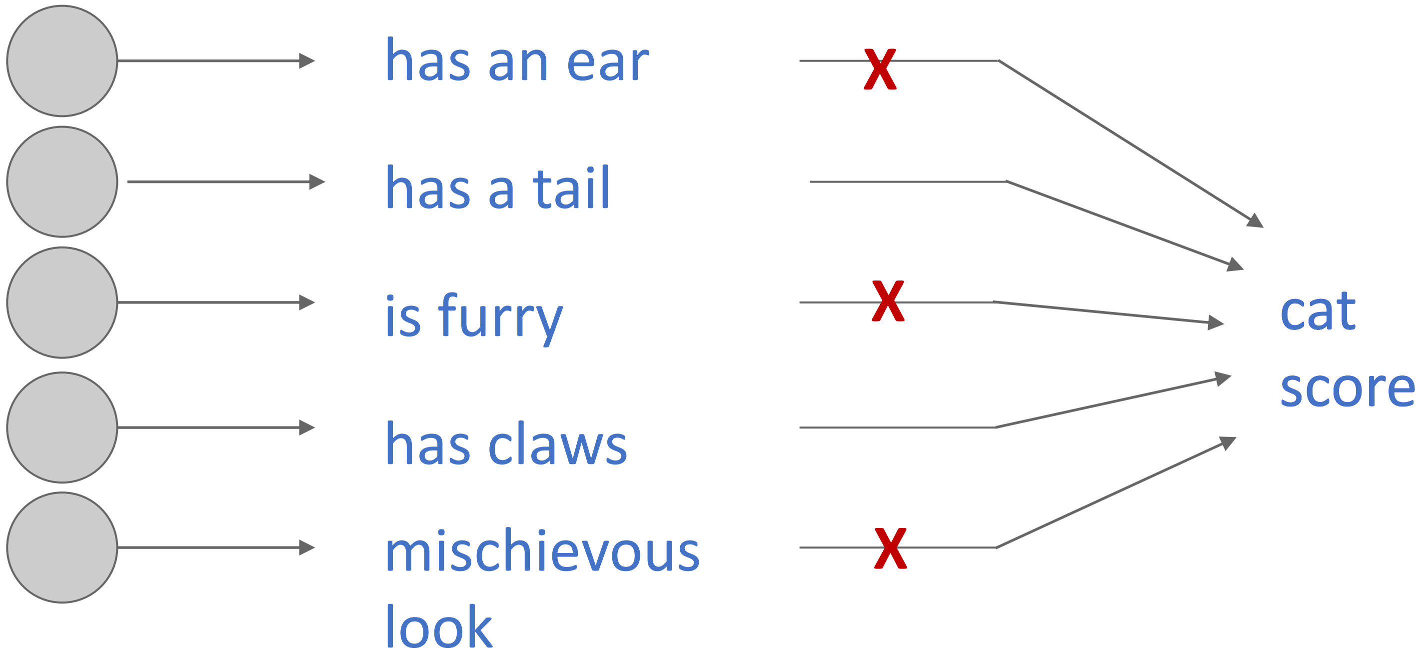
Example forward pass with a 3-layer network using dropout



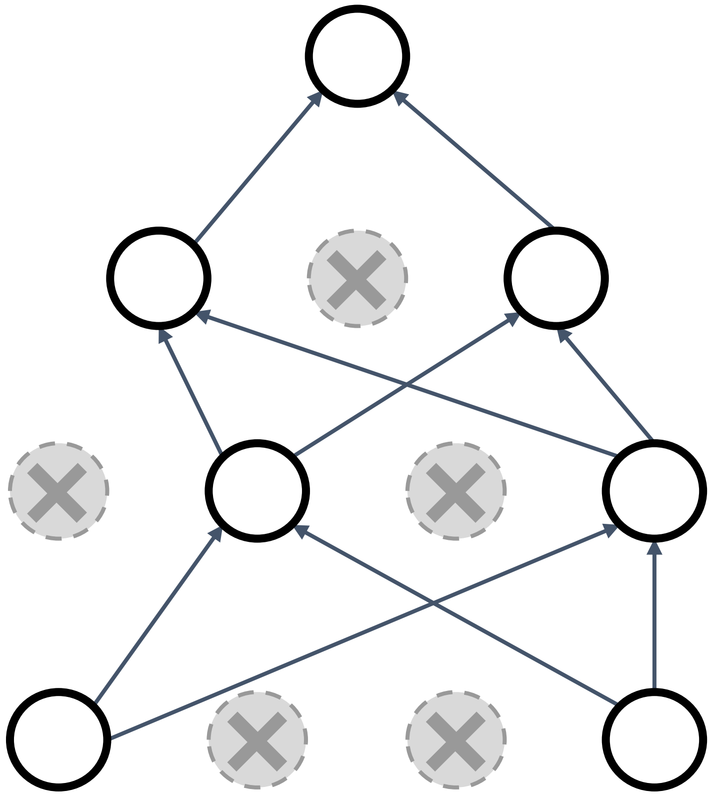
Dropout



Forces the network to have a redundant representation; Prevents **co-adaptation** of features



Dropout



Another interpretation:

Dropout is training a large **ensemble** of models (that share parameters).

Each binary-masked one is one model

An FC layer with 4096 units has $2^{4096} \sim 10^{1233}$ possible masks!
Only $\sim 10^{82}$ atoms in the universe...

Dropout: Test Time

Output (label)

Input (image)

Dropout makes our output random!

$$\boldsymbol{y} = f_W(\boldsymbol{x}, \boldsymbol{z})$$

Random mask

Want to "average out" the randomness at test-time

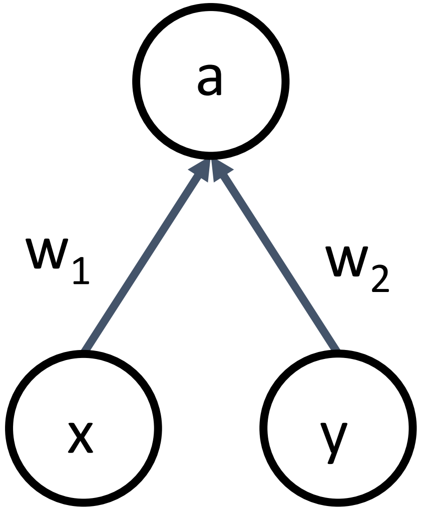
$$y = f(x) = E_z[f(x,z)] = \int p(z) f(x,z) dz$$

But this integral seems hard …

Dropout: Test Time

Want to approximate the integral $y = f(x) = E_z[f(x,z)] = \int p(z) f(x,z) dz$

Consider a single neuron:



At test time we have: $E[a] = w_1 x + w_2 y$

During training we have: $E[a] = \frac{1}{4}(w_1 x + w_2 y) + \frac{1}{4}(w_1 x + 0y)$
$+ \frac{1}{4}(0x + 0y) + \frac{1}{4}(0x + w_2 y)$
$= \frac{1}{2}(w_1 x + w_2 y)$

At test time, drop nothing and **multiply** by dropout probability

Dropout: Test Time

```python
def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

At test time all neurons are active always

=> We must scale the activations so that for each neuron:

<u>output at test time</u> = <u>expected output at training time</u>

Dropout: Summary

```python
""" Vanilla Dropout: Not recommended implementation (see notes below) """

p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  """ X contains the data """

  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = np.random.rand(*H1.shape) < p # first dropout mask
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = np.random.rand(*H2.shape) < p # second dropout mask
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) * p # NOTE: scale the activations
  H2 = np.maximum(0, np.dot(W2, H1) + b2) * p # NOTE: scale the activations
  out = np.dot(W3, H2) + b3
```

drop in forward pass

scale at test time

“Inverted Dropout” Is More Common in Practice

```python
p = 0.5 # probability of keeping a unit active. higher = less dropout

def train_step(X):
  # forward pass for example 3-layer neural network
  H1 = np.maximum(0, np.dot(W1, X) + b1)
  U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p!
  H1 *= U1 # drop!
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p!
  H2 *= U2 # drop!
  out = np.dot(W3, H2) + b3

  # backward pass: compute gradients... (not shown)
  # perform parameter update... (not shown)

def predict(X):
  # ensembled forward pass
  H1 = np.maximum(0, np.dot(W1, X) + b1) # no scaling necessary
  H2 = np.maximum(0, np.dot(W2, H1) + b2)
  out = np.dot(W3, H2) + b3
```

Drop and scale during training

test time is unchanged!

NN as a Universal Approximator

Neural Networks as Universal Approximation

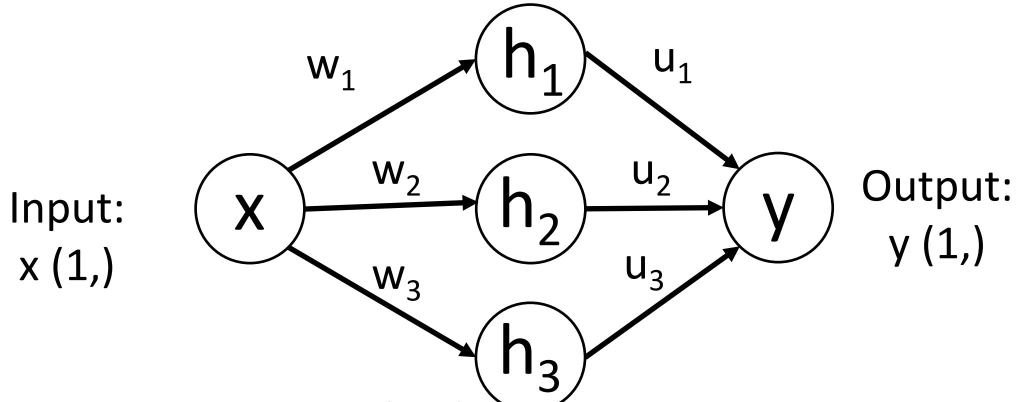
- A neural network with one hidden layer can approximate any(*) function $f: R^N \rightarrow R^M$ with arbitrary precision.

- For example, two-layer Sigmoid/ReLU networks with arbitrary number of hidden units

(*) 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



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

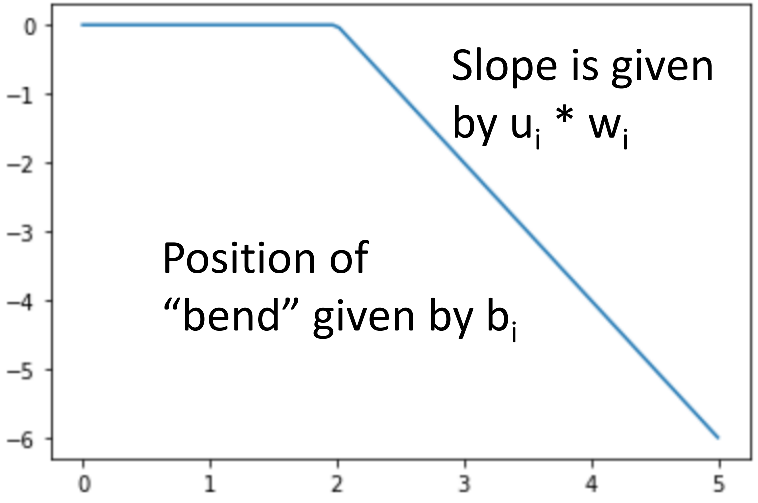
Second layer weights: u (1,3)
First layer bias: p (1,)

h1 = max(0, w1 * x + b1)
h2 = max(0, w2 * x + b2)
h3 = max(0, w3 * x + b3)
y = u1 * h1 + u2 * h2 + u3 * h3 + p

y = u1 * max(0, w1 * x + b1)
+ u2 * max(0, w2 * x + b2)
+ u3 * max(0, w3 * x + b3)
+ p

Output is a sum of shifted, scaled ReLUs:

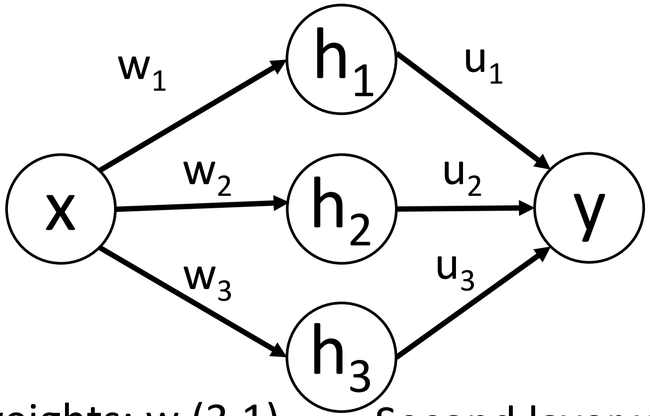
Flip left / right based on sign of w_i



Universal Approximation

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

Input:
x (1,)

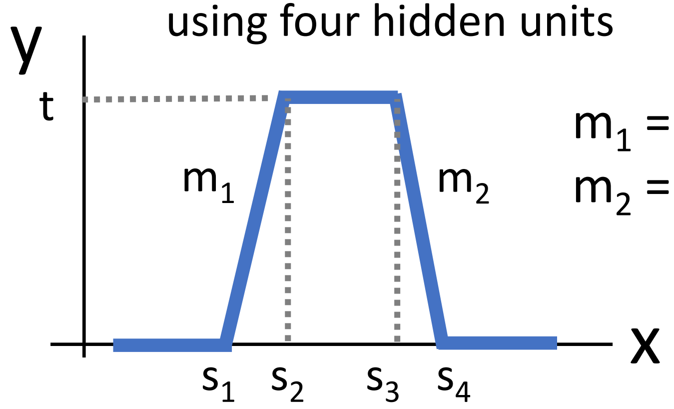


Output:
y (1,)

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

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

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



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

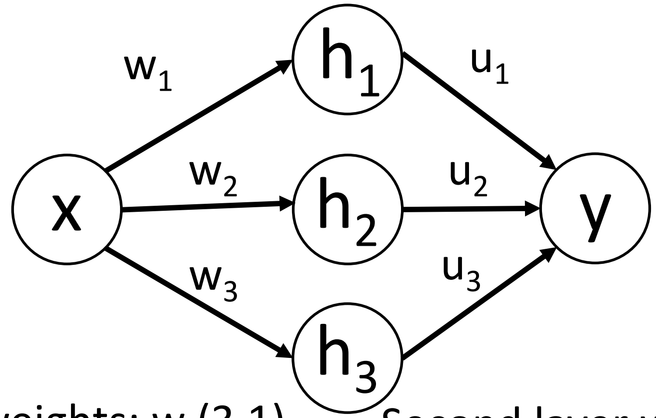
h1 = max(0, w1 * x + b1)
h2 = max(0, w2 * x + b2)
h3 = max(0, w3 * x + b3)
y = u1 * h1 + u2 * h2 + u3 * h3 + p

y = u1 * max(0, w1 * x + b1)
+ u2 * max(0, w2 * x + b2)
+ u3 * max(0, w3 * x + b3)
+ p

Universal Approximation

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

Input:
x (1,)



Output:
y (1,)

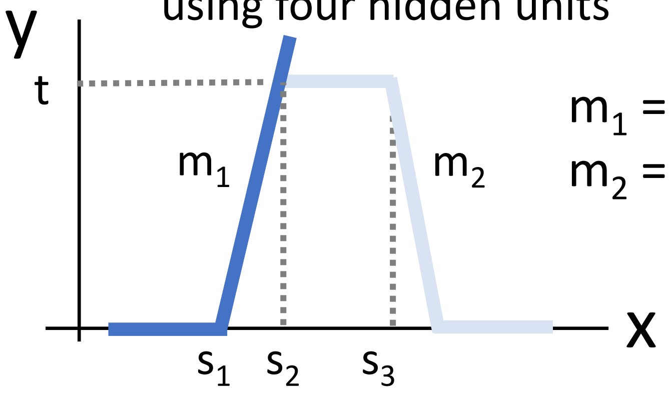
First layer weights: w (3,1)
First layer bias: b (3,)

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

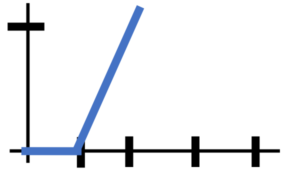
h1 = max(0, w1 * x + b1)
h2 = max(0, w2 * x + b2)
h3 = max(0, w3 * x + b3)
y = u1 * h1 + u2 * h2 + u3 * h3 + p

y = u1 * max(0, w1 * x + b1)
+ u2 * max(0, w2 * x + b2)
+ u3 * max(0, w3 * x + b3)
+ p

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



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

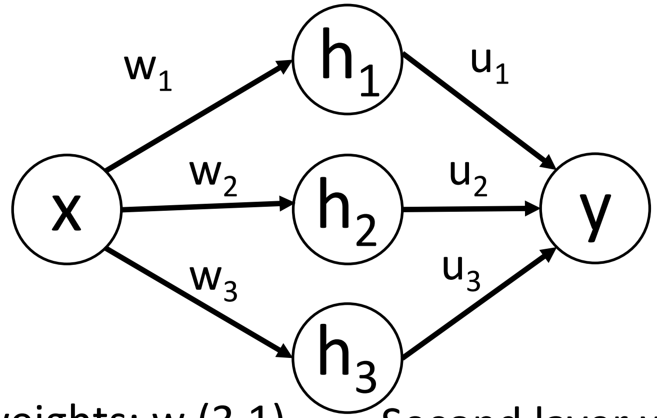


m_1 * max(0, x – s_1)

Universal Approximation

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

Input:
x (1,)



Output:
y (1,)

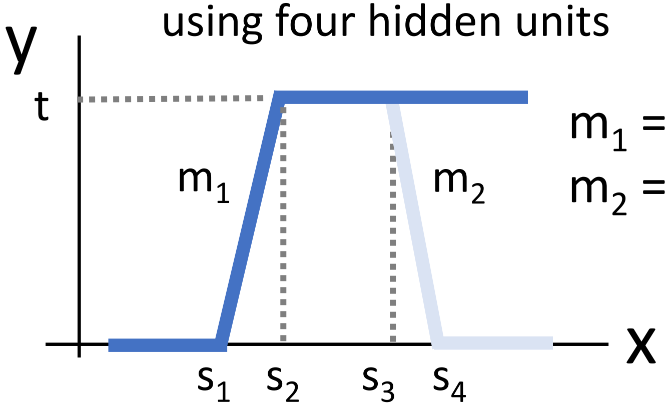
First layer weights: w (3,1)
First layer bias: b (3,)

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

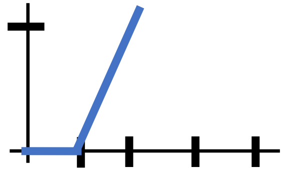
h1 = max(0, w1 * x + b1)
h2 = max(0, w2 * x + b2)
h3 = max(0, w3 * x + b3)
y = u1 * h1 + u2 * h2 + u3 * h3 + p

y = u1 * max(0, w1 * x + b1)
+ u2 * max(0, w2 * x + b2)
+ u3 * max(0, w3 * x + b3)
+ p

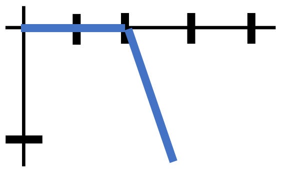
We can build a "bump function" using four hidden units



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



m_1 * max(0, x – s_1)

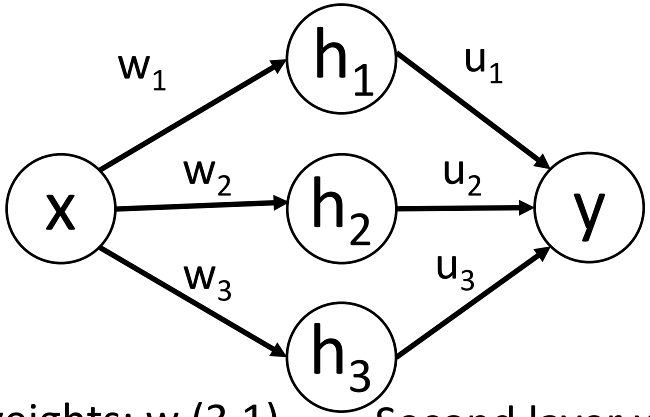


-m_1 * max(0, x – s_2)

Universal Approximation

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

Input:
x (1,)



Output:
y (1,)

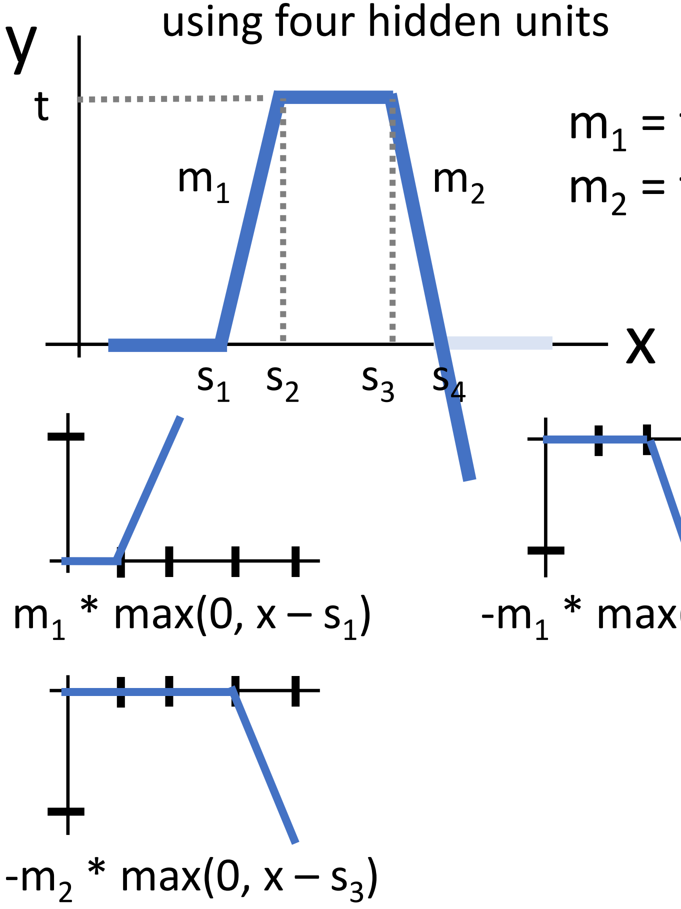
First layer weights: w (3,1)
First layer bias: b (3,)

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

h1 = max(0, w1 * x + b1)
h2 = max(0, w2 * x + b2)
h3 = max(0, w3 * x + b3)
y = u1 * h1 + u2 * h2 + u3 * h3 + p

y = u1 * max(0, w1 * x + b1)
+ u2 * max(0, w2 * x + b2)
+ u3 * max(0, w3 * x + b3)
+ p

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

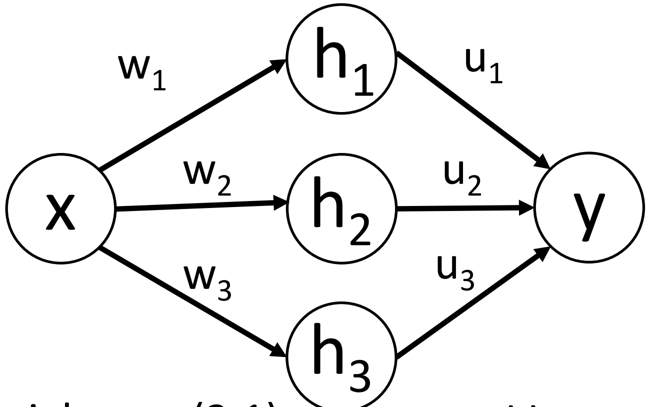


$m_1 = t / (s_2 – s_1)$
$m_2 = t / (s_4 – s_3)$

Universal Approximation

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

Input:
x (1,)



Output:
y (1,)

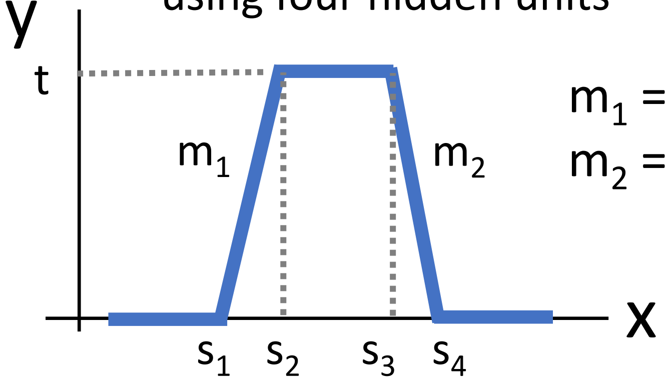
First layer weights: w (3,1)
First layer bias: b (3,)

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

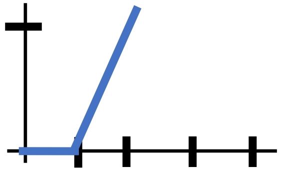
h1 = max(0, w1 * x + b1)
h2 = max(0, w2 * x + b2)
h3 = max(0, w3 * x + b3)
y = u1 * h1 + u2 * h2 + u3 * h3 + p

y = u1 * max(0, w1 * x + b1)
 + u2 * max(0, w2 * x + b2)
 + u3 * max(0, w3 * x + b3)
 + p

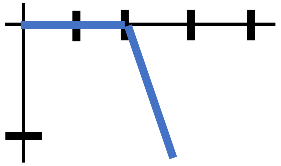
We can build a "bump function" using four hidden units



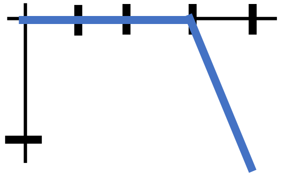
$m_1 = t / (s_2 - s_1)$
$m_2 = t / (s_4 - s_3)$



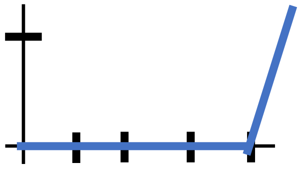
$m_1 * \max(0, x - s_1)$



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



$-m_2 * \max(0, x - s_3)$

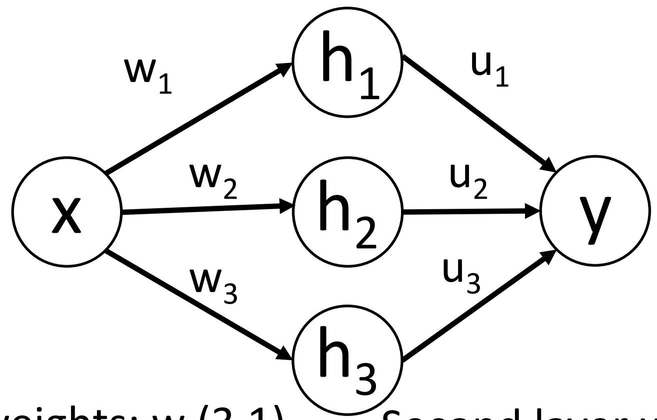


$m_2 * \max(0, x - s_4)$

Universal Approximation

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

Input:
x (1,)



Output:
y (1,)

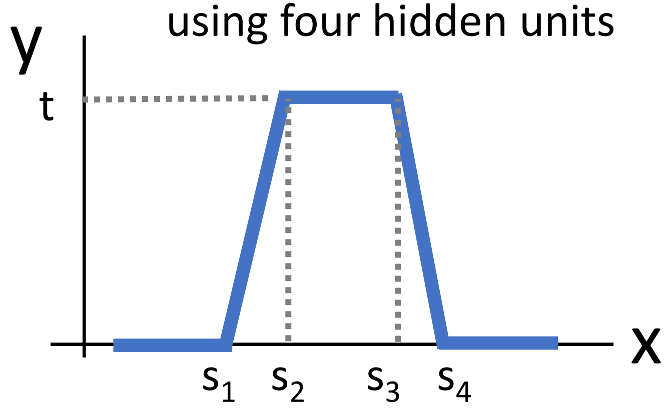
First layer weights: w (3,1)
First layer bias: b (3,)

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

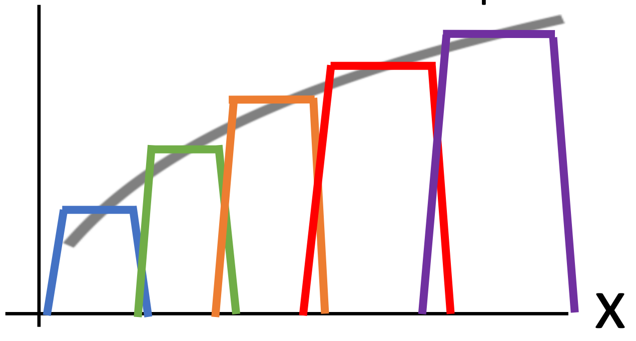
h1 = max(0, w1 * x + b1)
h2 = max(0, w2 * x + b2)
h3 = max(0, w3 * x + b3)
y = u1 * h1 + u2 * h2 + u3 * h3 + p

y = u1 * max(0, w1 * x + b1)
 + u2 * max(0, w2 * x + b2)
 + u3 * max(0, w3 * x + b3)
 + p

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



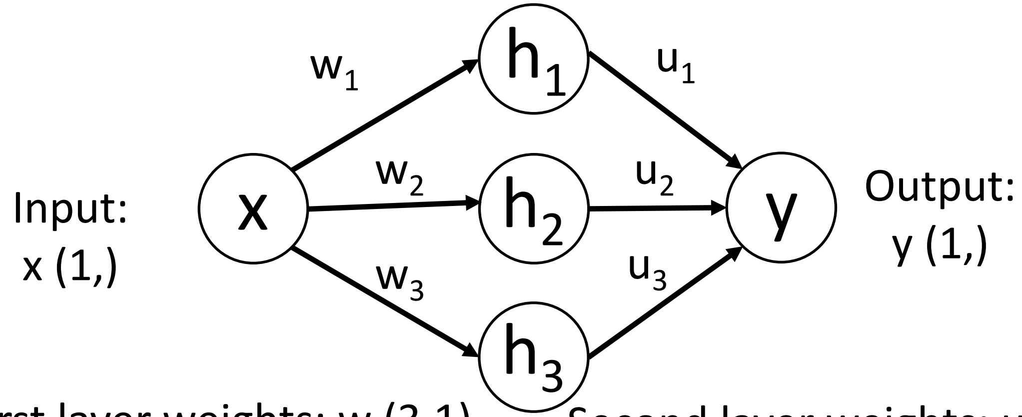
With 4K hidden units, we can build a sum of K bumps



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,)

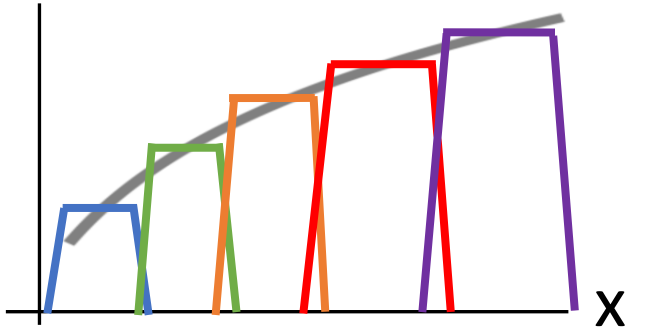
Second layer weights: u (1,3)
First layer bias: p (1,)

$$
\begin{aligned}
h1 &= \max(0, w1 * x + b1) \\
h2 &= \max(0, w2 * x + b2) \\
h3 &= \max(0, w3 * x + b3) \\
y &= u1 * h1 + u2 * h2 + u3 * h3 + p
\end{aligned}
$$

$$
\begin{aligned}
y = u1 &* \max(0, w1 * x + b1) \\
+ u2 &* \max(0, w2 * x + b2) \\
+ u3 &* \max(0, w3 * x + b3) \\
+ p &
\end{aligned}
$$

What about…
- Gaps between bumps?
- Other nonlinearities?
- Higher-dimensional functions?

See [Nielsen, Chapter 4]

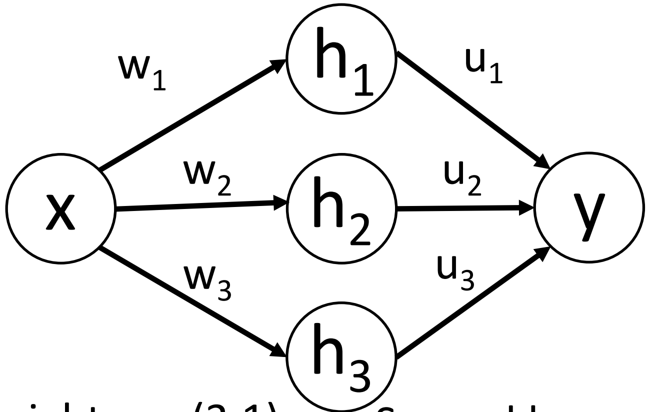


Approximate functions with bumps!

Universal Approximation

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

Input:
x (1,)

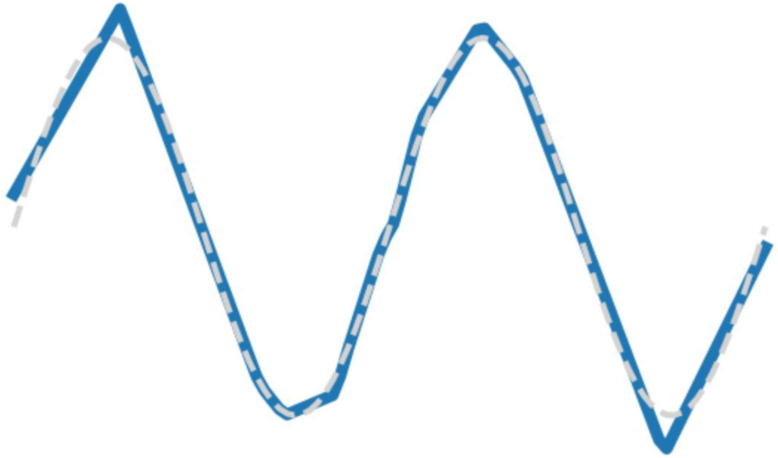


Output:
y (1,)

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

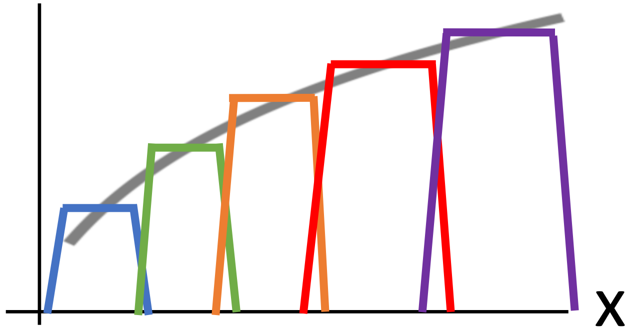
Second layer weights: u (1,3)
First layer bias: p (1,)

Reality check: Networks don’t really learn bumps!



h1 = max(0, w1 * x + b1)
h2 = max(0, w2 * x + b2)
h3 = max(0, w3 * x + b3)
y = u1 * h1 + u2 * h2 + u3 * h3 + p

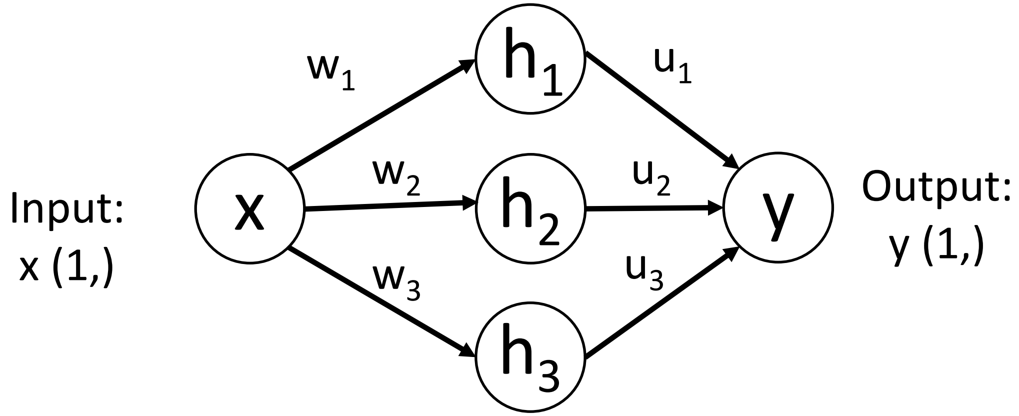
y = u1 * max(0, w1 * x + b1)
+ u2 * max(0, w2 * x + b2)
+ u3 * max(0, w3 * x + b3)
+ p



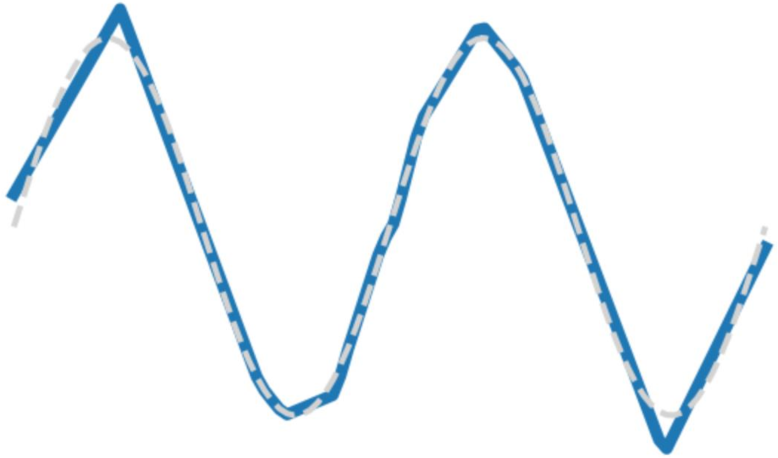
Approximate functions with bumps!

Universal Approximation

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



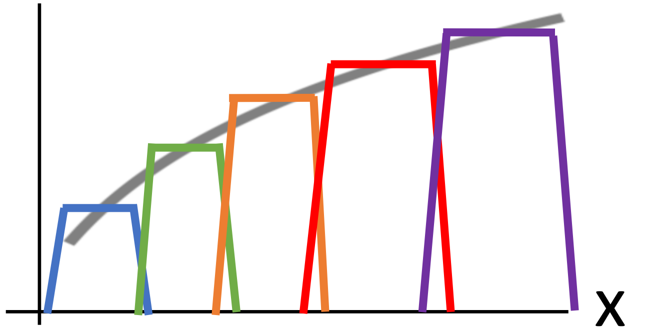
Reality check: Networks don't really learn bumps!



Universal approximation tells us:
- Neural nets can represent any function

Universal approximation DOES NOT tell us:
- Whether we can actually learn any function with SGD
- How much data we need to learn a function

Remember: k-NN is also a universal approximator!



Approximate functions with bumps!

NN Optimization

Problem: How to compute (complex) gradients?

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

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

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

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

If we can compute $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}, \frac{\partial L}{\partial b_1}, \frac{\partial L}{\partial b_2}$ then we can optimize with SGD

(Bad) Idea: Derive $\nabla_W L$ on paper

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

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

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \lambda \sum_k W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

$$\nabla_W L = \nabla_W \left(\frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

Problem: Very tedious: Lots of matrix calculus, need lots of paper

Problem: What if we want to change loss? e.g., use softmax instead of SVM? Need to re-derive from scratch. Not modular!

Problem: Not feasible for very complex models!

Better Idea: Backpropagation by **Chain Rule**

$$x \xrightarrow{W_1} h_1 \xrightarrow{W_2} h_2 \xrightarrow{W_3} h_3 \xrightarrow{W_4} L$$

D_0 $\quad$ D_1 $\quad$ D_2 $\quad$ D_3 $\quad$ scalar

Chain rule

$$\frac{\partial L}{\partial x} = \left(\frac{\partial h_1}{\partial x}\right)\left(\frac{\partial h_2}{\partial h_1}\right)\left(\frac{\partial h_3}{\partial h_2}\right)\left(\frac{\partial L}{\partial h_3}\right)$$

[D_0 x D_1] [D_1 x D_2] [D_2 x D_3] [D_3]

e.g., (i, j)-th element in W_2

$$\frac{\partial L}{\partial W_2(i,j)} = \left(\frac{\partial h_2}{\partial W_2(i,j)}\right)\left(\frac{\partial h_3}{\partial h_2}\right)\left(\frac{\partial L}{\partial h_3}\right)$$

[D_2] [D_2 x D_3] [D_3]

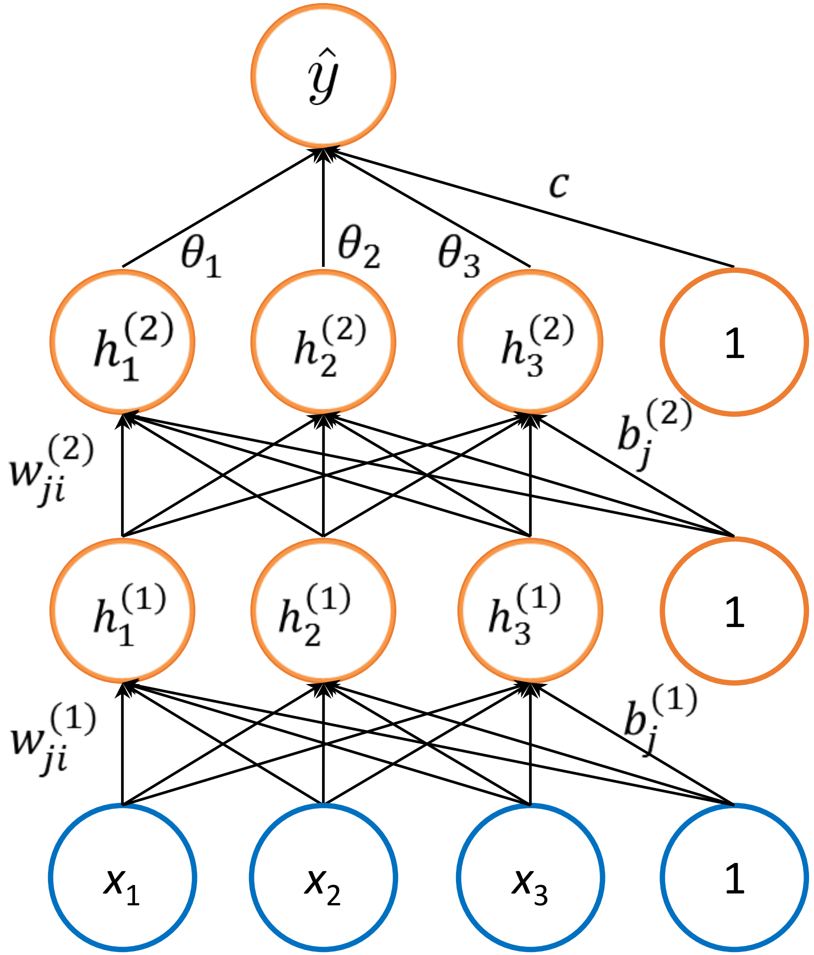
Example: 2-Layer NN Forward Pass

- Input $\mathbf{x}$
- Output $\hat{y}$
- Target y
- L2 Loss function $L = (\hat{y} - y)^2$

First hidden layer $$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)})$$

Second hidden layer $$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(1)} + b_j^{(2)})$$

Output $$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$



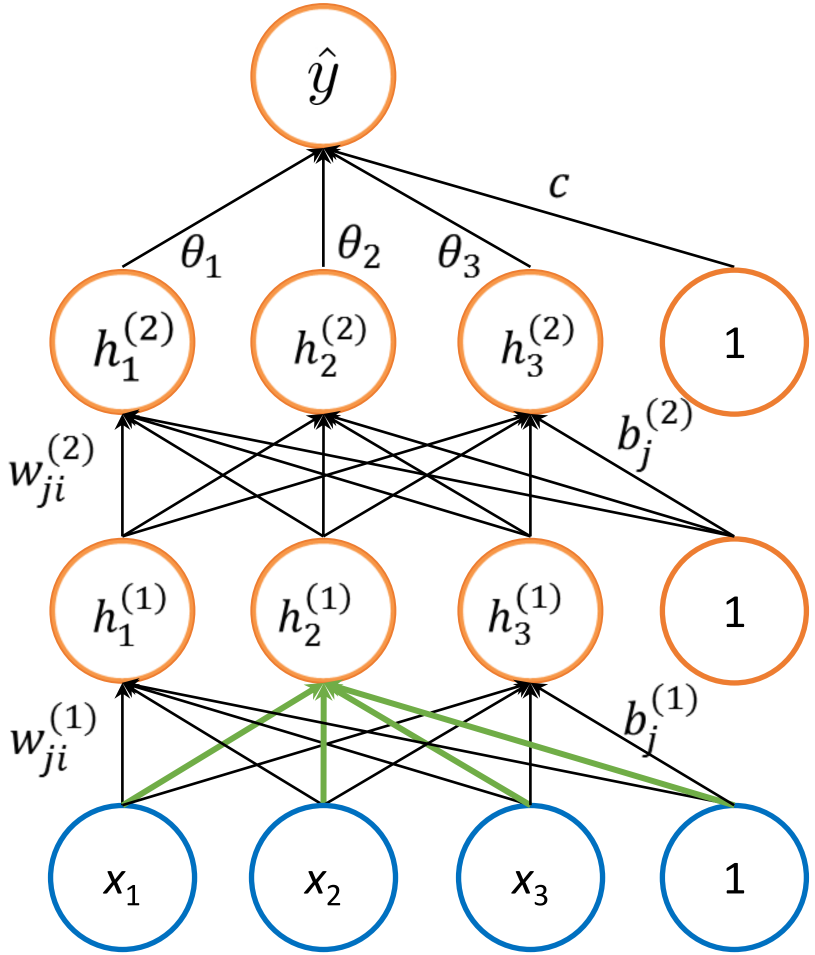
Example: 2-Layer NN Forward Pass

- Input $\mathbf{x}$
- Output $\hat{y}$
- Target y
- L2 Loss function $L = (\hat{y} - y)^2$

First hidden layer $$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)})$$

Second hidden layer $$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(1)} + b_j^{(2)})$$

Output $$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$



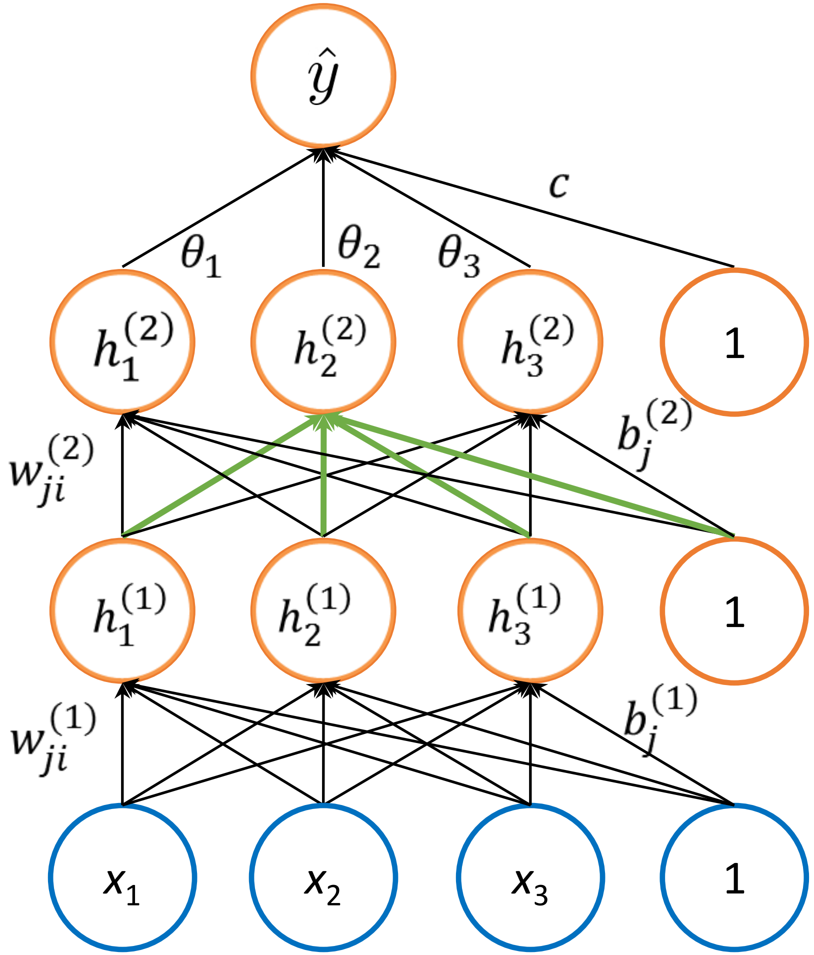
Example: 2-Layer NN Forward Pass

- Input $\mathbf{x}$
- Output $\hat{y}$
- Target y
- L2 Loss function $L = (\hat{y} - y)^2$

First hidden layer $$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)})$$

Second hidden layer $$\boxed{h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(1)} + b_j^{(2)})}$$

Output $$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$



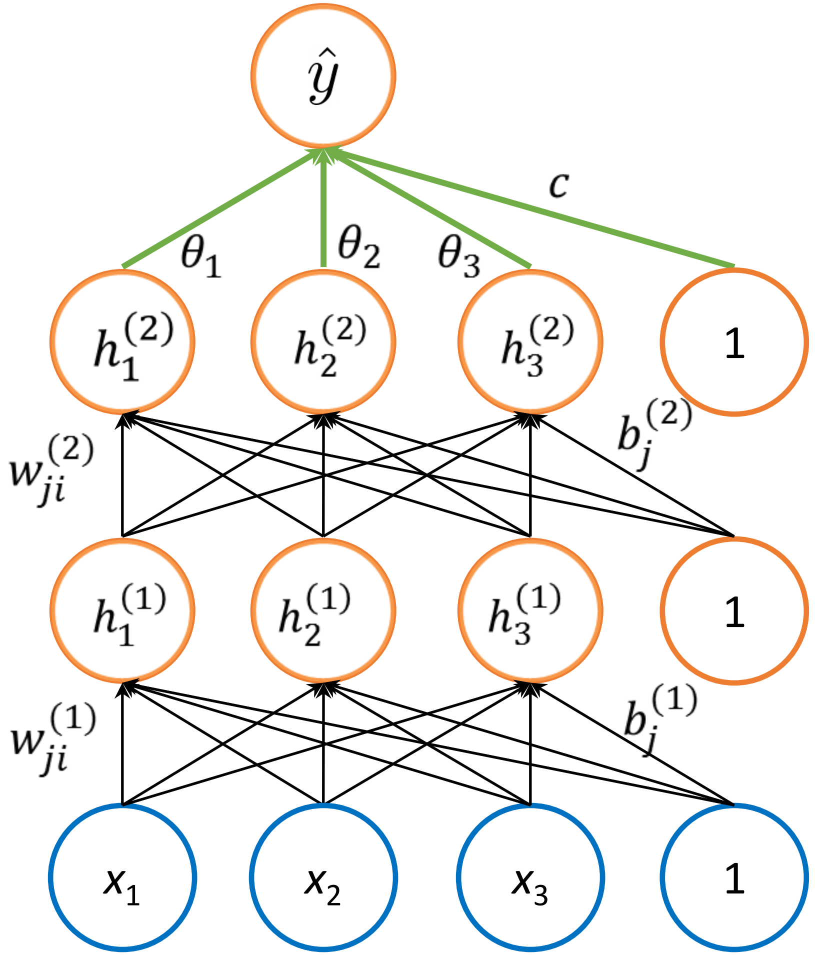
Example: 2-Layer NN Forward Pass

- Input $\mathbf{x}$
- Output $\hat{y}$
- Target y
- L2 Loss function $L = (\hat{y} - y)^2$

First hidden layer $$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)})$$

Second hidden layer $$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(1)} + b_j^{(2)})$$

Output $$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$



Example: 2-Layer NN Backward Pass

- Compute gradients w.r.t.

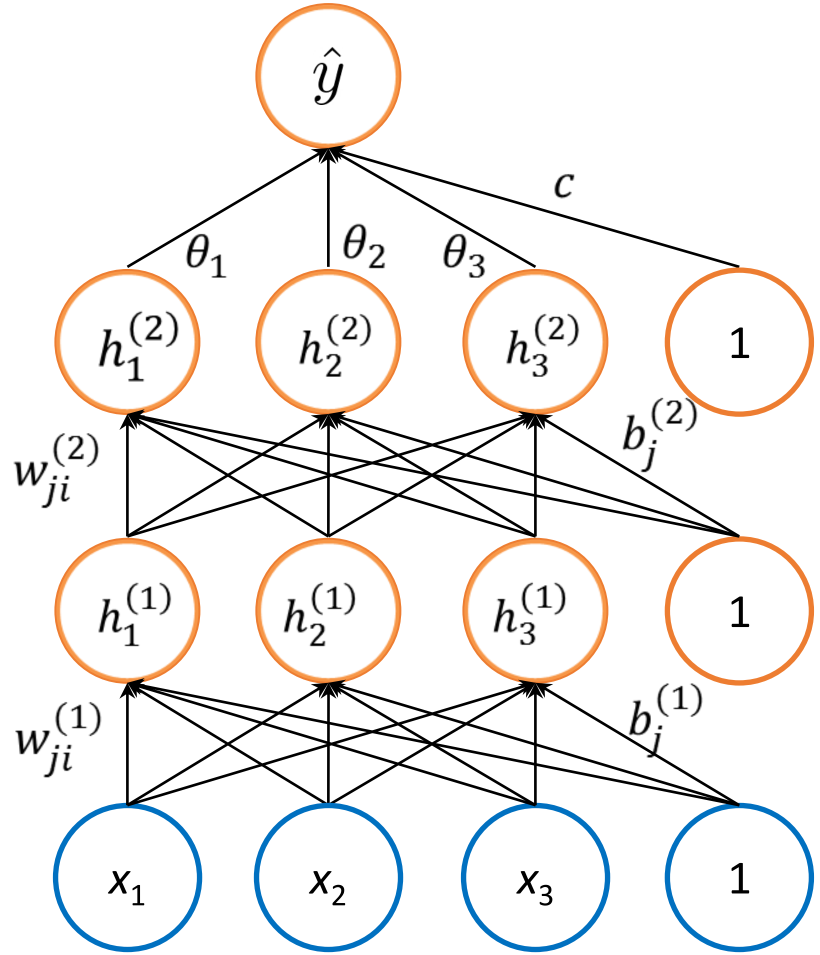
$$\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \theta, c\}$$

$$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)}), \forall j$$

$$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(i)} + b_j^{(2)}), \forall j$$

$$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$

$$L = (\hat{y} - y)^2$$



Example: 2-Layer NN Backward Pass

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

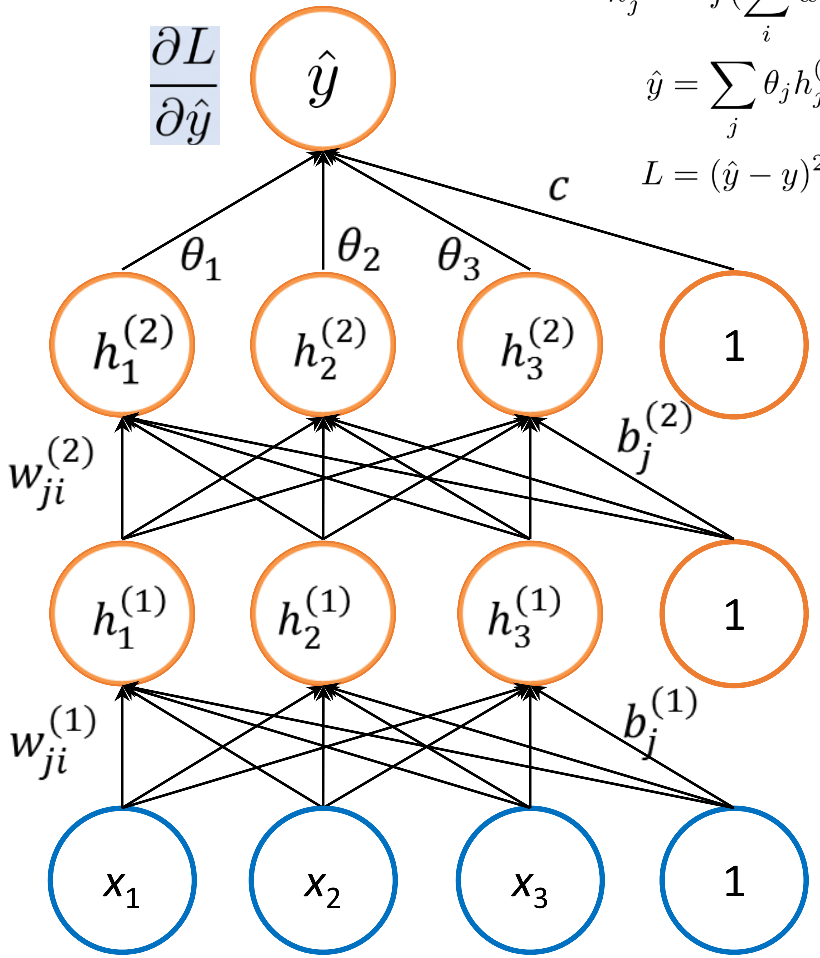
$$\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \theta, c\}$$

$$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)}), \forall j$$

$$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(i)} + b_j^{(2)}), \forall j$$

$$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$

$$L = (\hat{y} - y)^2$$



Example: 2-Layer NN Backward Pass

$$\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \theta, c\}$$

$$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)}), \forall j$$

$$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(i)} + b_j^{(2)}), \forall j$$

$$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$

$$L = (\hat{y} - y)^2$$

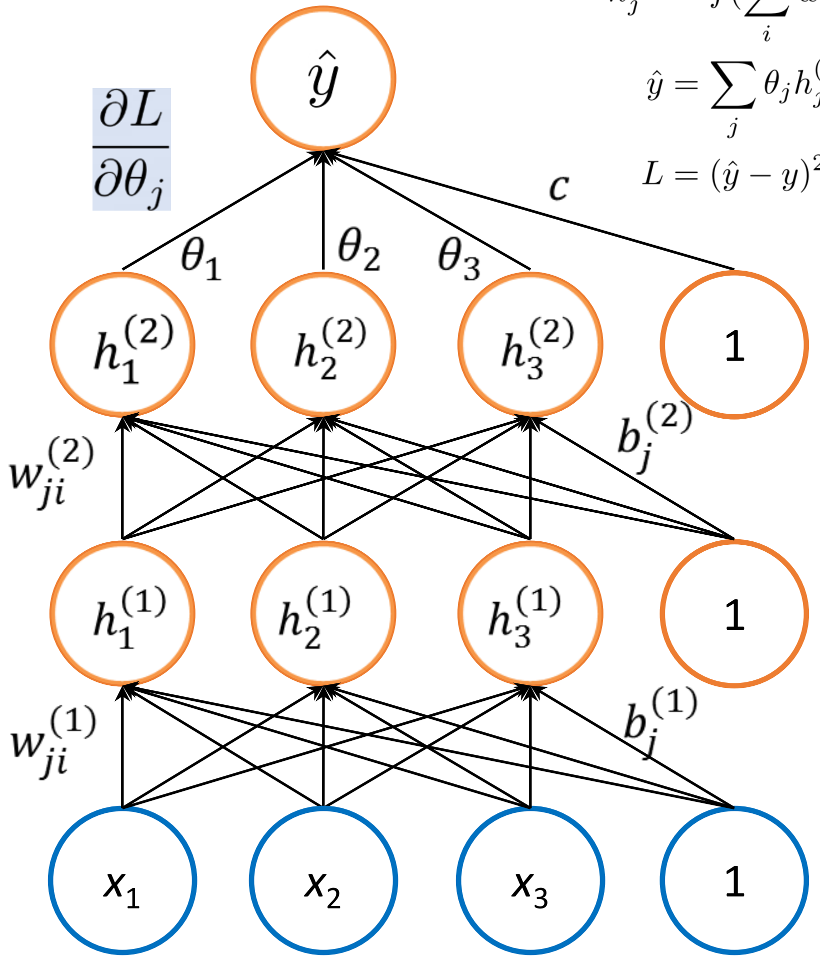
$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\frac{\partial L}{\partial \theta_j} = \frac{\partial \hat{y}}{\partial \theta_j} \frac{\partial L}{\partial \hat{y}} = h_j^{(2)} \frac{\partial L}{\partial \hat{y}}$$

Downstream gradient

Local gradient

Upstream gradient



Example: 2-Layer NN Backward Pass

$$\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \theta, c\}$$

$$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)}), \forall j$$

$$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(i)} + b_j^{(2)}), \forall j$$

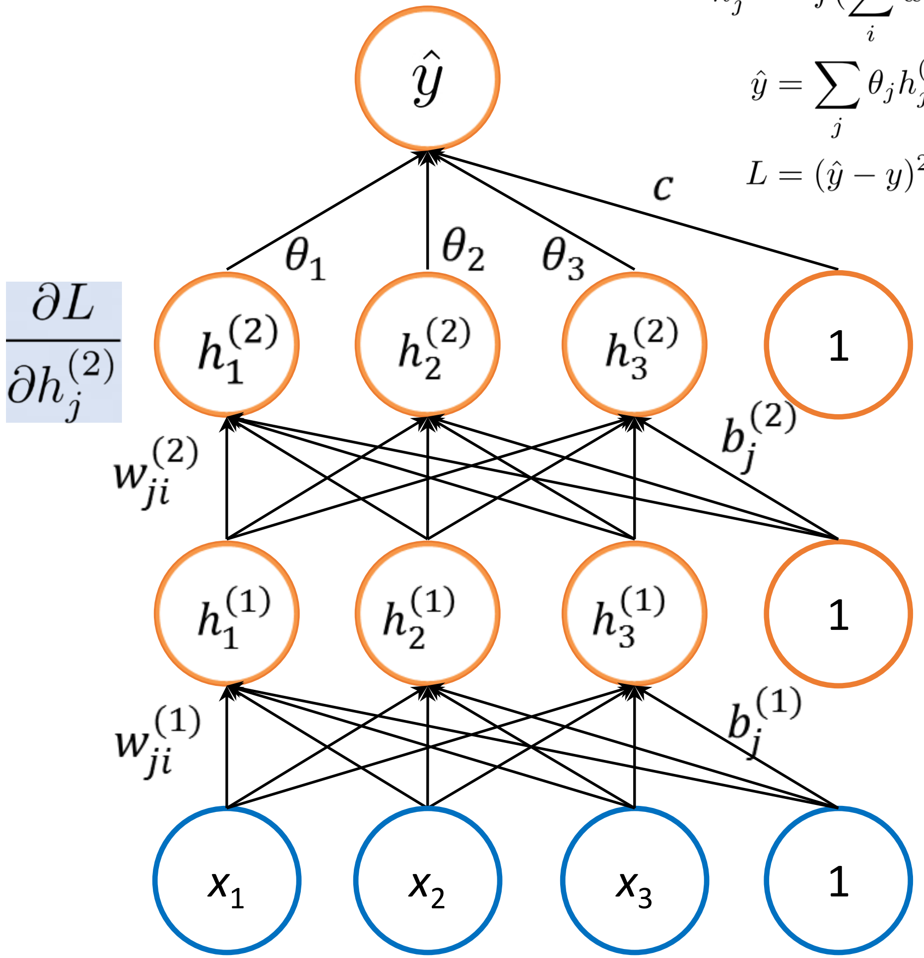
$$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$

$$L = (\hat{y} - y)^2$$

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\frac{\partial L}{\partial \theta_j} = \frac{\partial \hat{y}}{\partial \theta_j} \frac{\partial L}{\partial \hat{y}} = h_j^{(2)} \frac{\partial L}{\partial \hat{y}}$$

$$\frac{\partial L}{\partial h_j^{(2)}} = \frac{\partial \hat{y}}{\partial h_j^{(2)}} \frac{\partial L}{\partial \hat{y}} = \theta_j \frac{\partial L}{\partial \hat{y}}$$



Example: 2-Layer NN Backward Pass

$$\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \theta, c\}$$

$$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)}), \forall j$$

$$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(i)} + b_j^{(2)}), \forall j$$

$$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$

$$L = (\hat{y} - y)^2$$

$$z_j^{(2)} = \sum_i h_i^{(1)} w_{ji}^{(2)} + b_j^{(2)}$$

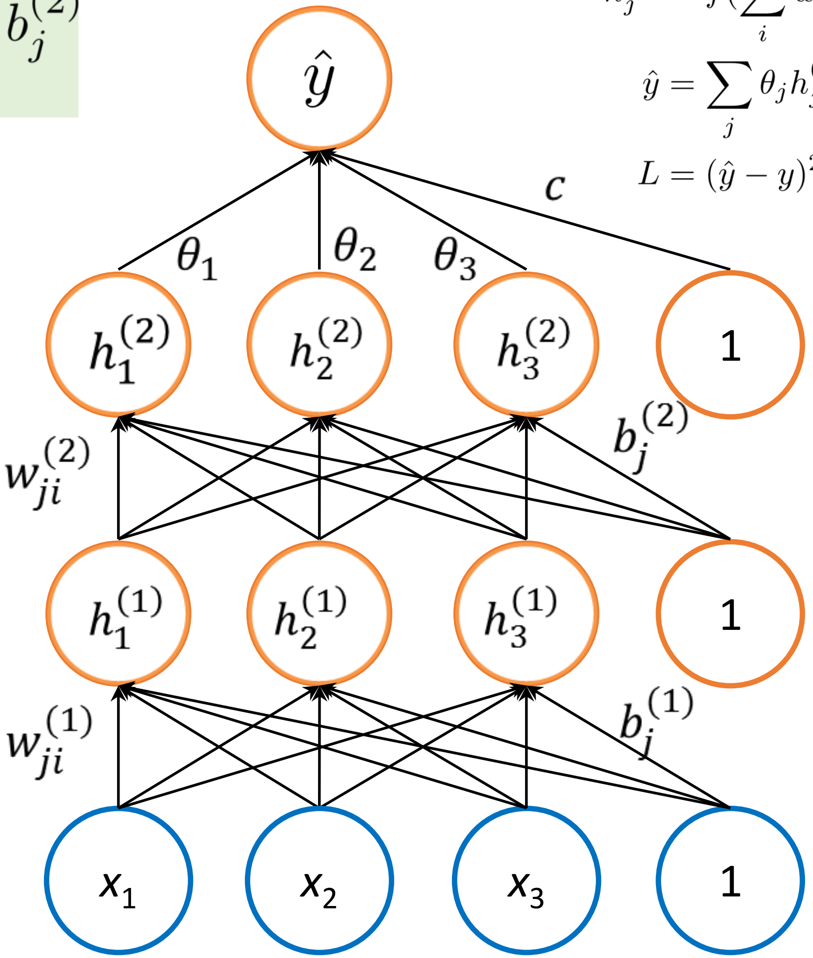
$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\frac{\partial L}{\partial \theta_j} = \frac{\partial \hat{y}}{\partial \theta_j}\frac{\partial L}{\partial \hat{y}} = h_j^{(2)} \frac{\partial L}{\partial \hat{y}}$$

$$\frac{\partial L}{\partial h_j^{(2)}} = \frac{\partial \hat{y}}{\partial h_j^{(2)}}\frac{\partial L}{\partial \hat{y}} = \theta_j \frac{\partial L}{\partial \hat{y}}$$

$$\frac{\partial L}{\partial w_{ji}^{(2)}} = \frac{\partial h_j^{(2)}}{\partial w_{ji}^{(2)}} \frac{\partial L}{\partial h_j^{(2)}} = f'(z_j^{(2)}) h_i^{(1)} \frac{\partial L}{\partial h_j^{(2)}}$$

$$\frac{\partial L}{\partial w_{ji}^{(2)}}$$



Example: 2-Layer NN Backward Pass

$$\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \theta, c\}$$

$$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)}), \forall j$$

$$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(i)} + b_j^{(2)}), \forall j$$

$$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$

$$L = (\hat{y} - y)^2$$

$$z_j^{(2)} = \sum_i h_i^{(1)} w_{ji}^{(2)} + b_j^{(2)}$$

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

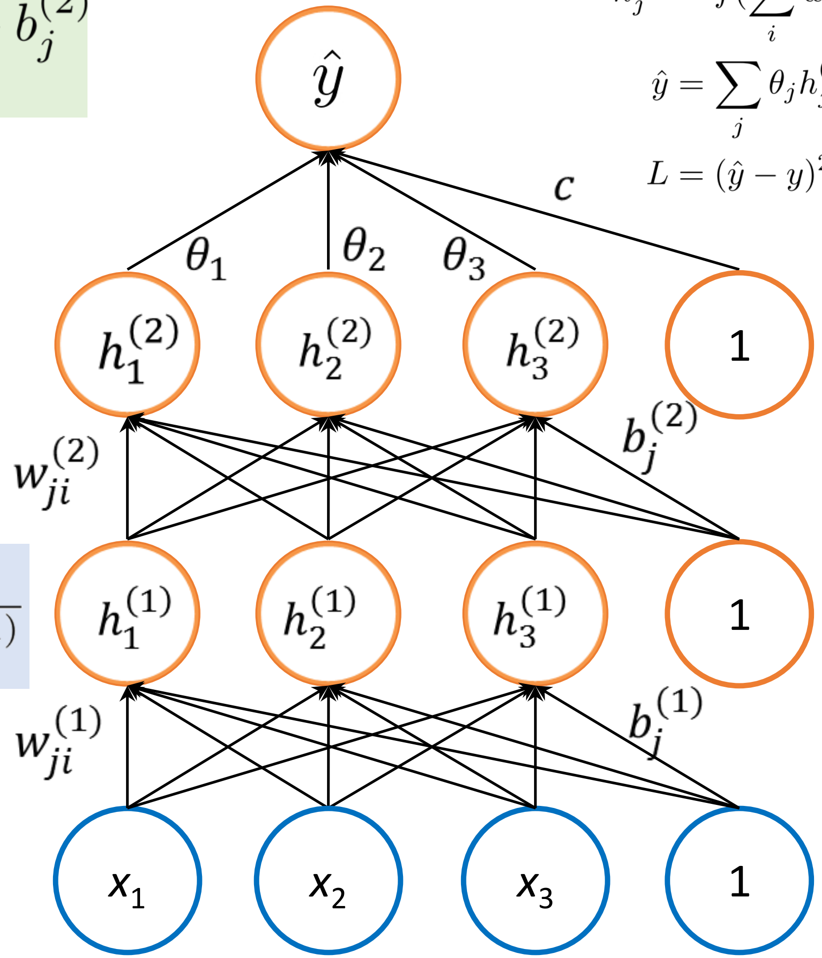
$$\frac{\partial L}{\partial \theta_j} = \frac{\partial \hat{y}}{\partial \theta_j} \frac{\partial L}{\partial \hat{y}} = h_j^{(2)} \frac{\partial L}{\partial \hat{y}}$$

$$\frac{\partial L}{\partial h_j^{(2)}} = \frac{\partial \hat{y}}{\partial h_j^{(2)}} \frac{\partial L}{\partial \hat{y}} = \theta_j \frac{\partial L}{\partial \hat{y}}$$

$$\frac{\partial L}{\partial w_{ji}^{(2)}} = \frac{\partial h_j^{(2)}}{\partial w_{ji}^{(2)}} \frac{\partial L}{\partial h_j^{(2)}} = f'(z_j^{(2)}) h_i^{(1)} \frac{\partial L}{\partial h_j^{(2)}}$$

$$\frac{\partial L}{\partial h_i^{(1)}} = \sum_j \frac{\partial h_j^{(2)}}{\partial h_i^{(1)}} \frac{\partial L}{\partial h_j^{(2)}} = \sum_j f'(z_j^{(2)}) w_{ji}^{(2)} \frac{\partial L}{\partial h_j^{(2)}}$$

$$\frac{\partial L}{\partial h_i^{(1)}}$$



Example: 2-Layer NN Backward Pass

$$\{W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}, \theta, c\}$$

$$h_j^{(1)} = f(\sum_i w_{ji}^{(1)} x_i + b_j^{(1)}), \forall j$$

$$h_j^{(2)} = f(\sum_i w_{ji}^{(2)} h_i^{(i)} + b_j^{(2)}), \forall j$$

$$\hat{y} = \sum_j \theta_j h_j^{(2)} + c$$

$$L = (\hat{y} - y)^2$$

$$z_j^{(2)} = \sum_i h_i^{(1)} w_{ji}^{(2)} + b_j^{(2)}$$

$$z_i^{(1)} = \sum_k x_k w_{ik}^{(1)} + b_i^{(1)}$$

$$\frac{\partial L}{\partial \hat{y}} = 2(\hat{y} - y)$$

$$\frac{\partial L}{\partial \theta_j} = \frac{\partial \hat{y}}{\partial \theta_j} \frac{\partial L}{\partial \hat{y}} = h_j^{(2)} \frac{\partial L}{\partial \hat{y}}$$

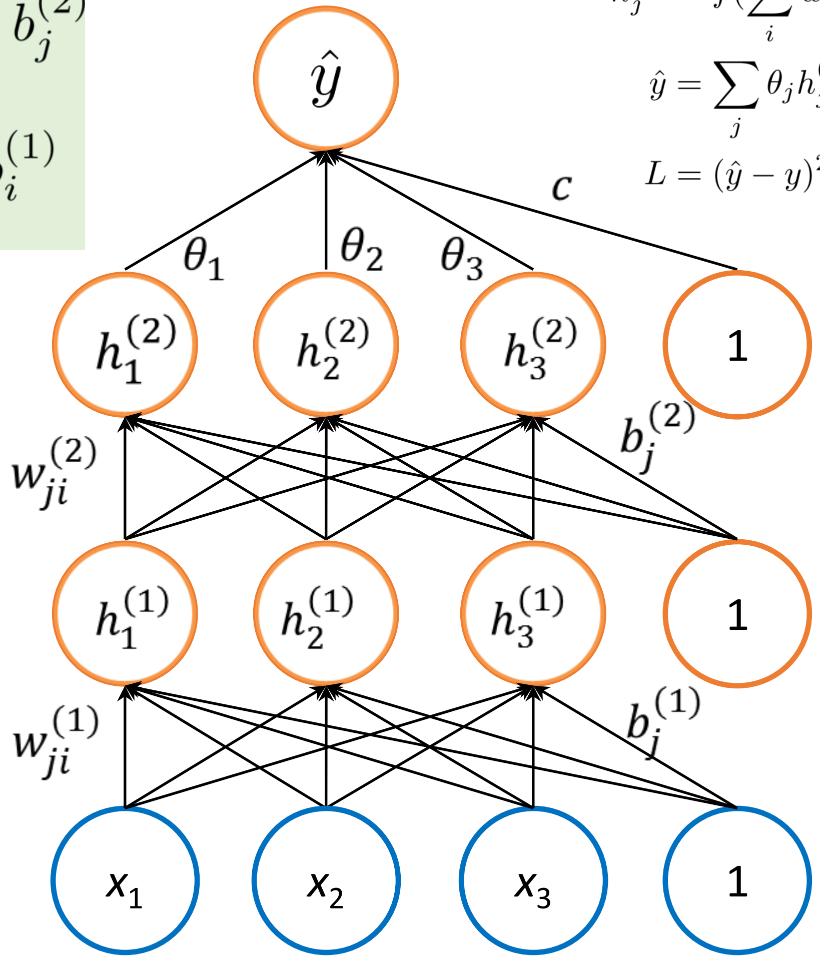
$$\frac{\partial L}{\partial h_j^{(2)}} = \frac{\partial \hat{y}}{\partial h_j^{(2)}} \frac{\partial L}{\partial \hat{y}} = \theta_j \frac{\partial L}{\partial \hat{y}}$$

$$\frac{\partial L}{\partial w_{ji}^{(2)}} = \frac{\partial h_j^{(2)}}{\partial w_{ji}^{(2)}} \frac{\partial L}{\partial h_j^{(2)}} = f'(z_j^{(2)}) h_i^{(1)} \frac{\partial L}{\partial h_j^{(2)}}$$

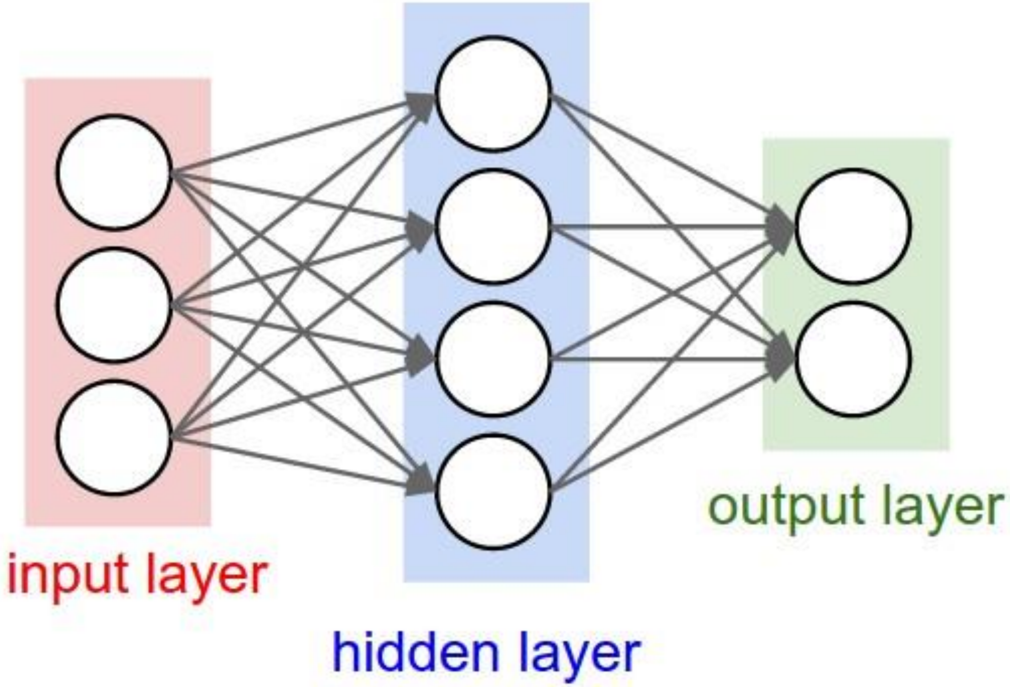
$$\frac{\partial L}{\partial h_i^{(1)}} = \sum_j \frac{\partial h_j^{(2)}}{\partial h_i^{(1)}} \frac{\partial L}{\partial h_j^{(2)}} = \sum_j f'(z_j^{(2)}) w_{ji}^{(2)} \frac{\partial L}{\partial h_j^{(2)}}$$

$$\frac{\partial L}{\partial w_{ik}^{(1)}}$$

$$\frac{\partial L}{\partial w_{ik}^{(1)}} = \frac{\partial h_i^{(1)}}{\partial w_{ik}^{(1)}} \frac{\partial L}{\partial h_i^{(1)}} = f'(z_i^{(1)}) x_k \frac{\partial L}{\partial h_i^{(1)}}$$

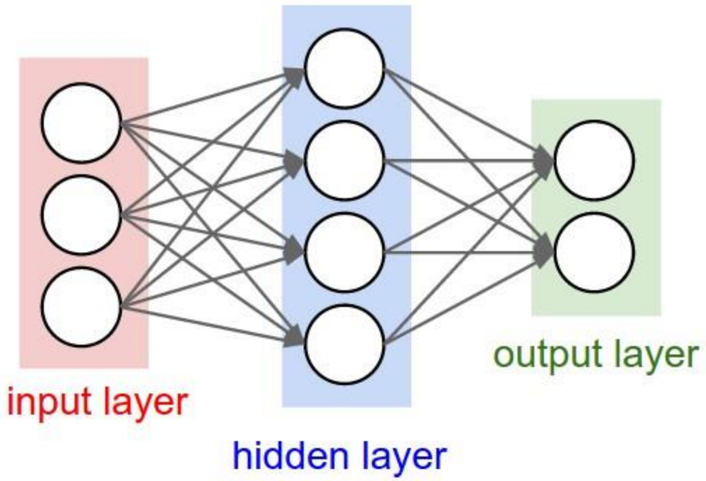


Neural Net in <20 lines!




```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!



Initialize weights and data

Compute loss (sigmoid activation, L2 loss)

Compute gradients

SGD step

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

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

Better Idea: Backpropagation by **Chain Rule**

$$x \xrightarrow{W_1} h_1 \xrightarrow{W_2} h_2 \xrightarrow{W_3} h_3 \xrightarrow{W_4} L$$

D_0 D_1 D_2 D_3 scalar

Chain rule

$$\frac{\partial L}{\partial x} = \left(\frac{\partial h_1}{\partial x}\right)\left(\frac{\partial h_2}{\partial h_1}\right)\left(\frac{\partial h_3}{\partial h_2}\right)\left(\frac{\partial L}{\partial h_3}\right)$$

[D_0 x D_1] [D_1 x D_2] [D_2 x D_3] [D_3]

e.g., (i, j)-th element in W_2

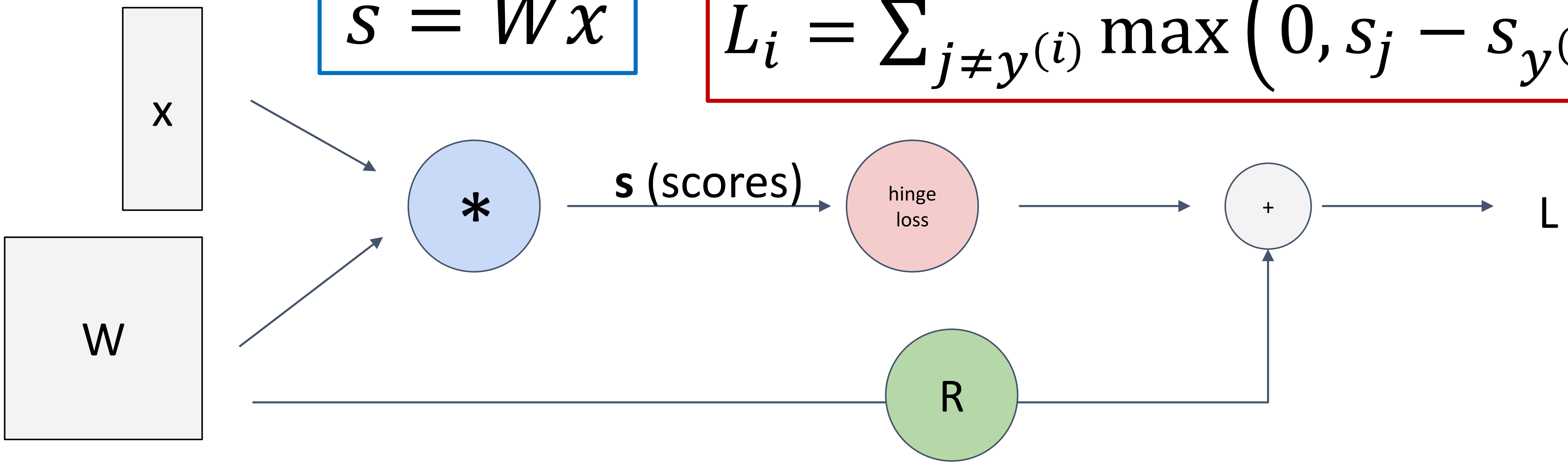
$$\frac{\partial L}{\partial W_2(i,j)} = \left(\frac{\partial h_2}{\partial W_2(i,j)}\right)\left(\frac{\partial h_3}{\partial h_2}\right)\left(\frac{\partial L}{\partial h_3}\right)$$

[D_2] [D_2 x D_3] [D_3]

Better Idea: **Computational Graphs**

$$\boxed{s = Wx}$$

$$\boxed{L_i = \sum_{j \neq y^{(i)}} \max\left(0, s_j - s_{y^{(i)}} + 1\right)}$$



$$\boxed{R(W)}$$

Next: Backpropagation