18-661 Introduction to Machine Learning

Neural Networks-I

Spring 2025

ECE – Carnegie Mellon University

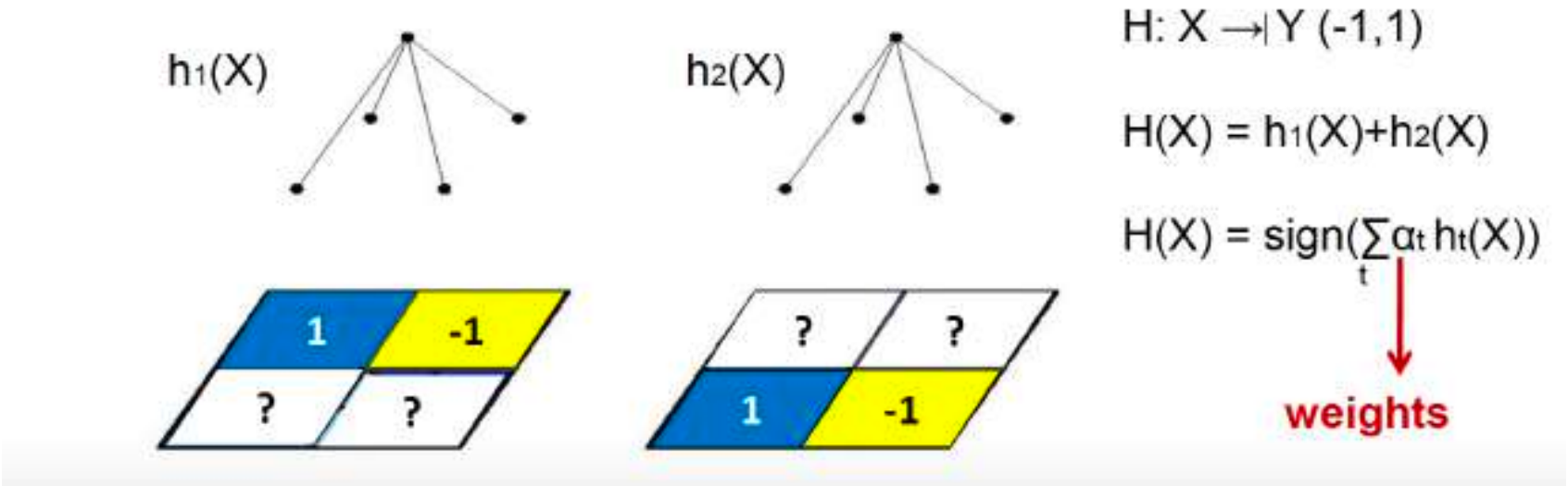
Outline

1. Review of Ensemble Methods
2. Neural Networks: Motivation
3. Neural Network Architectures and Forward Propagation
4. Choosing Activation Units
5. Choosing Neural Network Architectures

Review of Ensemble Methods

Ensemble methods

- Instead of learning a single (weak) classifier, learn many weak classifiers, preferably those that are good at different parts of the input spaces
- Predicted Class: (Weighted) Average or Majority of output of the weak classifiers
- Strength in Diversity!



Bagging

Bagging Trees (Training Phase)

- For $b = 1, 2, \cdots, B$
  - Choose n training samples $(\mathbf{x}_i, y_i)$ from $\mathcal{D}$ uniformly at random
  - Learn a decision tree h_b on these n samples
- Store the B decision trees $h_1, h_2, \ldots h_B$
- Optimal B (typically in 1000s) chosen using cross-validation

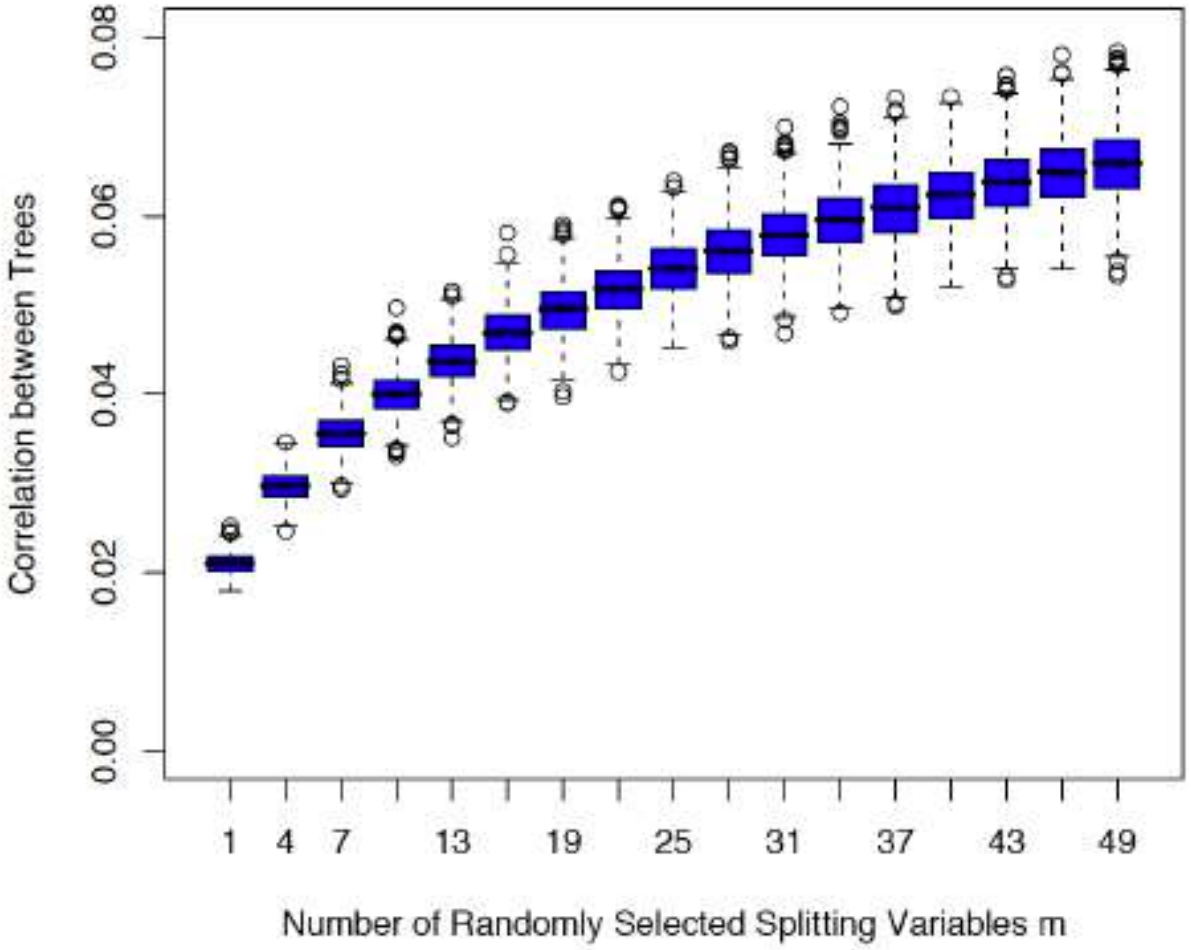
Bagging Trees (Test Phase)

- For a test unlabeled example $\mathbf{x}$
- Find the decision from each of the B trees
- Assign the majority (or most popular) label as the label for $\mathbf{x}$

Random Forests

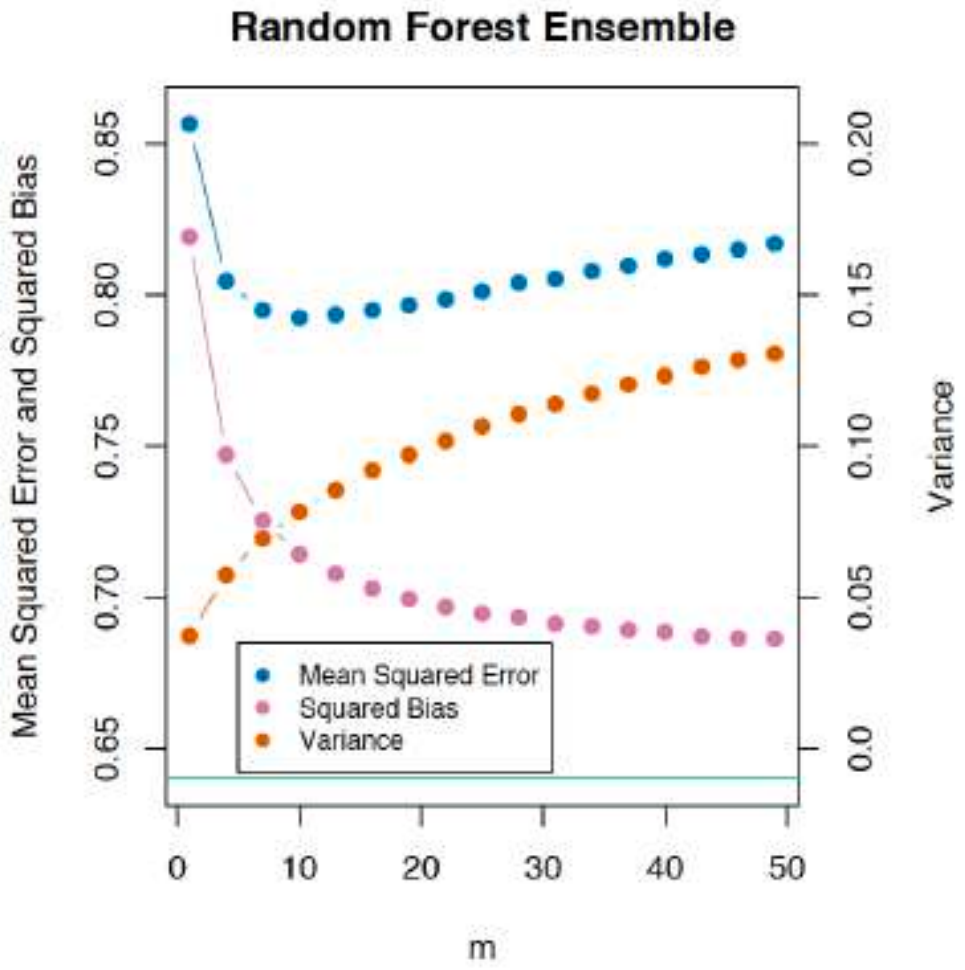
- Limitation of Bagging: If one or more features are very informative, they will be selected by almost every tree in the bag, reducing the diversity (and potentially increasing the bias).
- Key Idea behind Random Forests: Reduces correlation between trees in the bag without increasing variance too much
  - Same as bagging in terms of sampling training data
  - Before each split, select $m \leq d$ features at random as candidates for splitting $m \sim \sqrt{d}$
  - Take majority vote of B such trees

Random Forests



Increasing m, the number of splitting candidates chosen increases the correlation among the trees in the bag

Random Forests



Increasing m decreases the bias but increases variance of the output of the ensemble

Adaboost algorithm

- Given: N samples $\{\boldsymbol{x}_n, y_n\}$, where $y_n \in \{+1, -1\}$, and some way of constructing weak (or base) classifiers
- Initialize weights $w_1(n) = \frac{1}{N}$ for every training sample n
- For $t = 1$ to T
  1. Train a weak classifier $h_t(\boldsymbol{x})$ using current weights $w_t(n)$, by minimizing

     $$\epsilon_t = \sum_n w_t(n) \mathbb{I}[y_n \neq h_t(\boldsymbol{x}_n)] \quad \text{(the weighted classification error)}$$

  2. Compute contribution for this classifier: $\beta_t = \frac{1}{2} \log \frac{1-\epsilon_t}{\epsilon_t}$
  3. Update weights on each training sample n

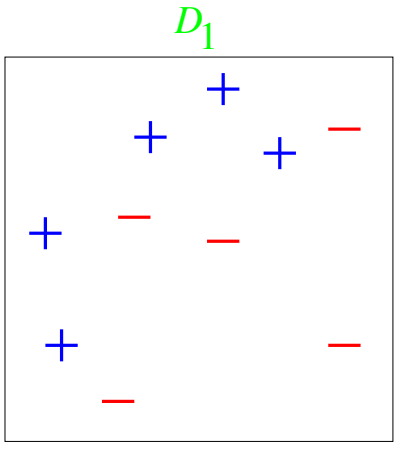
     $$w_{t+1}(n) \propto w_t(n) e^{-\beta_t y_n h_t(\boldsymbol{x}_n)}$$

     and normalize them such that $\sum_n w_{t+1}(n) = 1$.
- Output the final classifier

$$h[\boldsymbol{x}] = \text{sign}\left[\sum_{t=1}^{T} \beta_t h_t(\boldsymbol{x})\right]$$

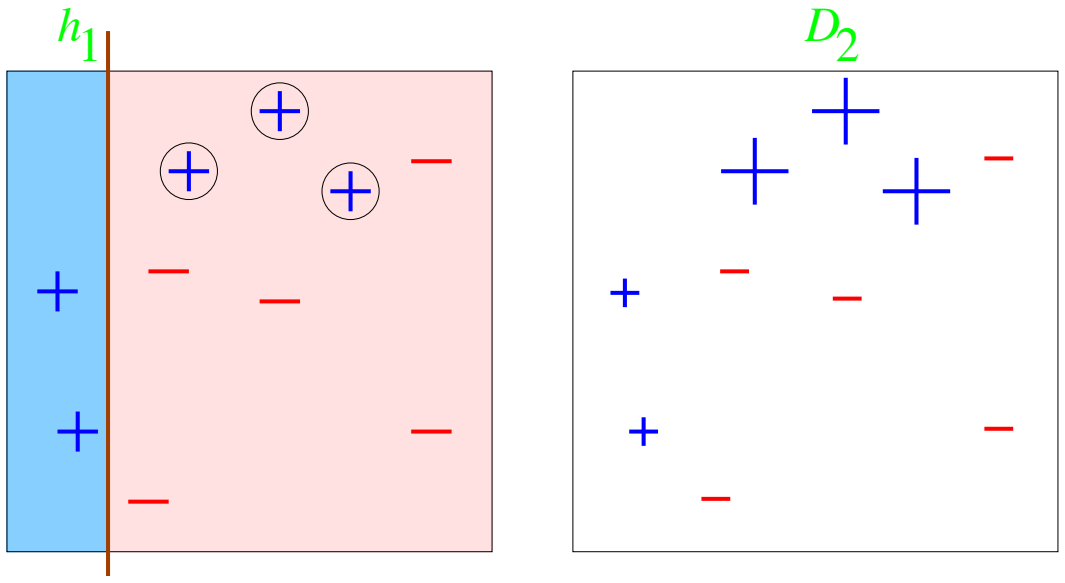
Example

10 data points and 2 features



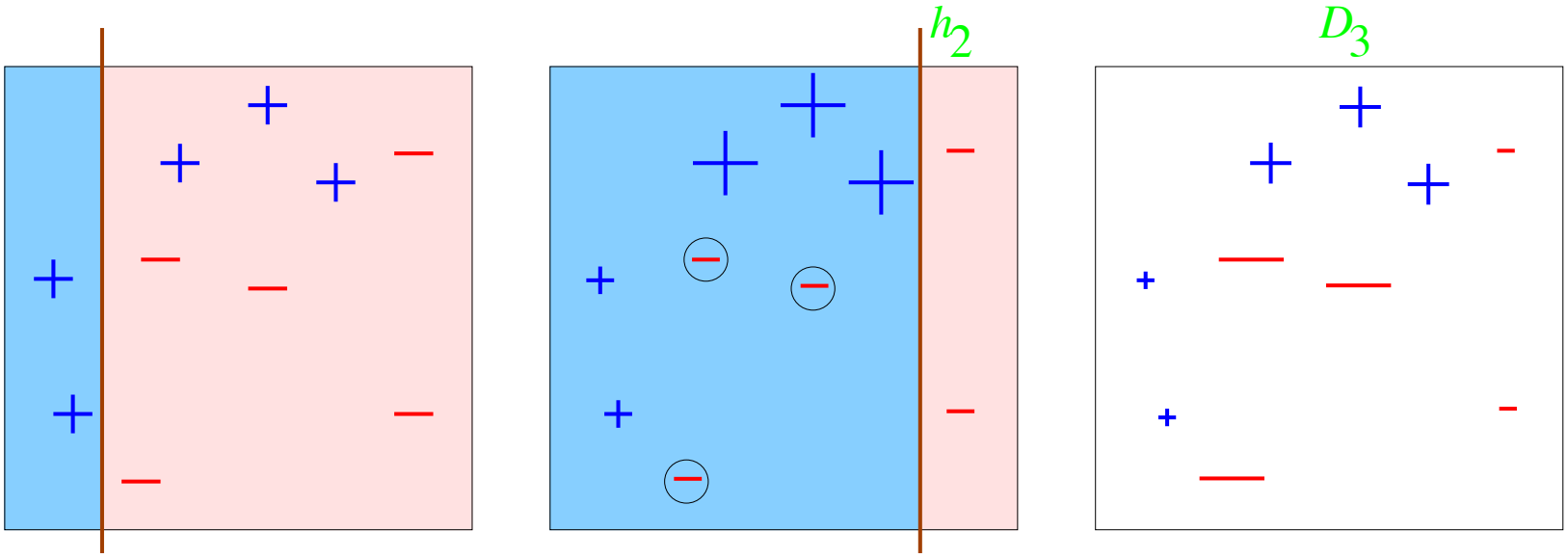
- The data points are clearly not linearly separable
- In the beginning, all data points have equal weights (the size of the data markers "+" or "-")
- Base classifier $h(\cdot)$: horizontal or vertical lines ('decision stumps')
  - Depth-1 decision trees, i.e., classify data based on a single attribute.

Round 1: $t = 1$



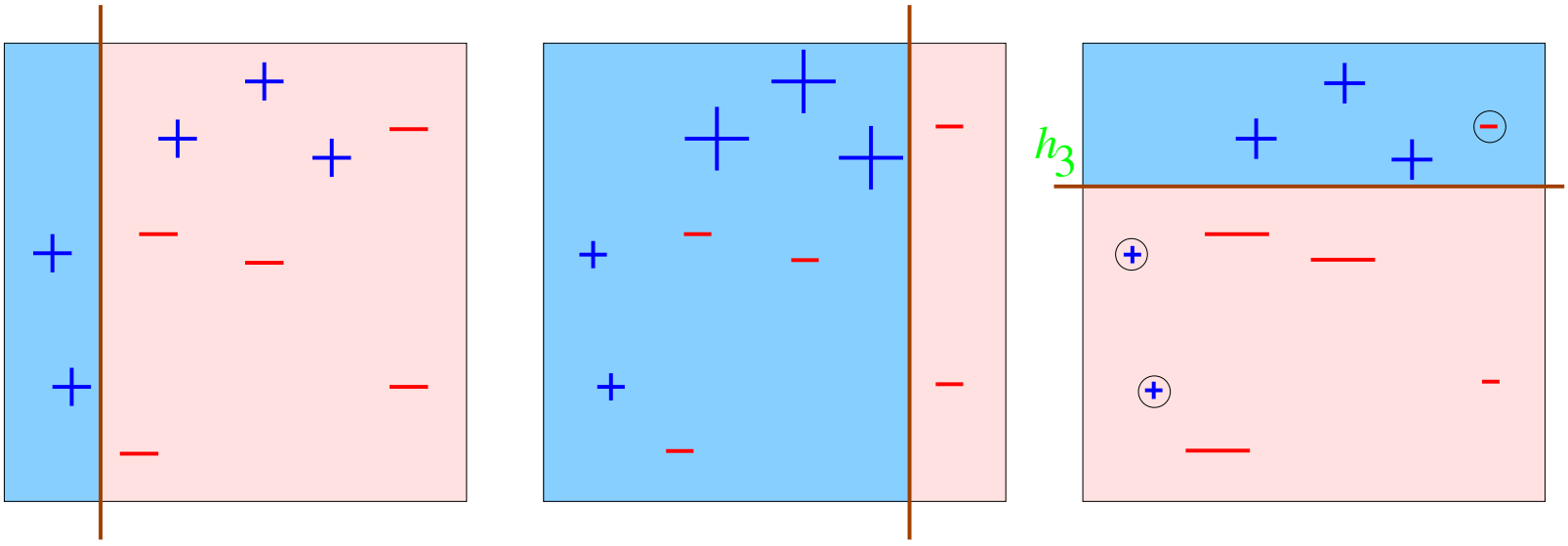
- 3 misclassified (with circles): $\epsilon_1 = 0.3 \rightarrow \beta_1 = 0.42$.
- Recompute the weights; the 3 misclassified data points receive larger weights

Round 2: $t = 2$



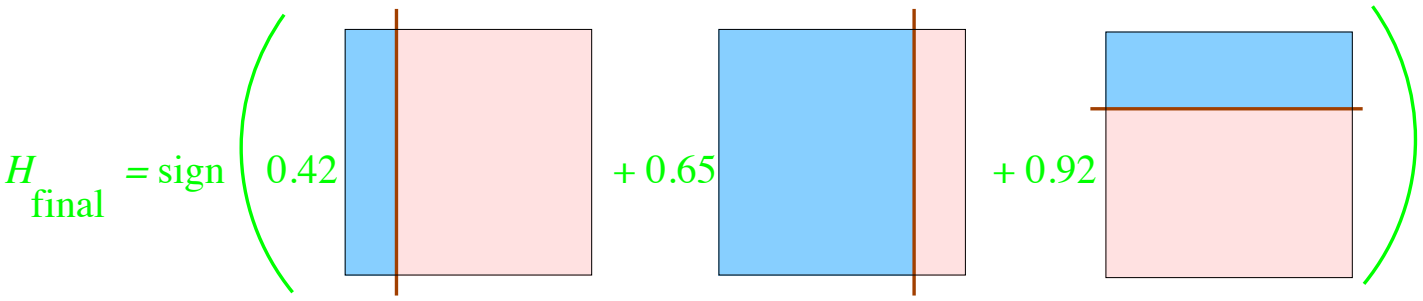
- 3 misclassified (with circles): $\epsilon_2 = 0.21 \rightarrow \beta_2 = 0.65$.
  $\epsilon_2 < 0.3$ as those 3 misclassified data points have weights < 0.1
- 3 newly misclassified data points get larger weights
- Data points classified correctly in both rounds have small weights

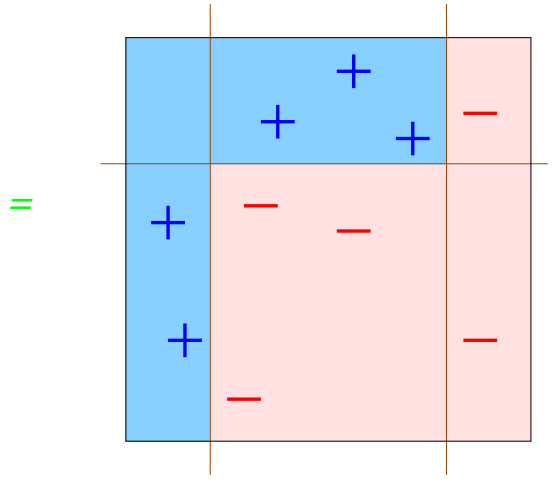
Round 3: $t = 3$



- 3 misclassified (with circles): $\epsilon_3 = 0.14 \rightarrow \beta_3 = 0.92$.
- Previously correctly classified data points are now misclassified, hence our error is low. Why?
  - Since they have been consistently classified correctly, this round's mistake will hopefully not have a huge impact on the overall prediction

Final classifier: Combining 3 classifiers





- All data points are now classified correctly!

Why does AdaBoost work?

It minimizes a loss function related to classification error.

Classification loss

- Suppose we want to have a classifier

$$h(\boldsymbol{x}) = \text{sign}[f(\boldsymbol{x})] = \begin{cases} 1 & \text{if } f(\boldsymbol{x}) > 0 \\ -1 & \text{if } f(\boldsymbol{x}) < 0 \end{cases}$$

- One seemingly natural loss function is 0-1 loss:

$$\ell(h(\boldsymbol{x}), y) = \begin{cases} 0 & \text{if } yf(\boldsymbol{x}) > 0 \\ 1 & \text{if } yf(\boldsymbol{x}) < 0 \end{cases}$$

  Namely, the function $f(\boldsymbol{x})$ and the target label y should have the same sign to avoid a loss of 1.

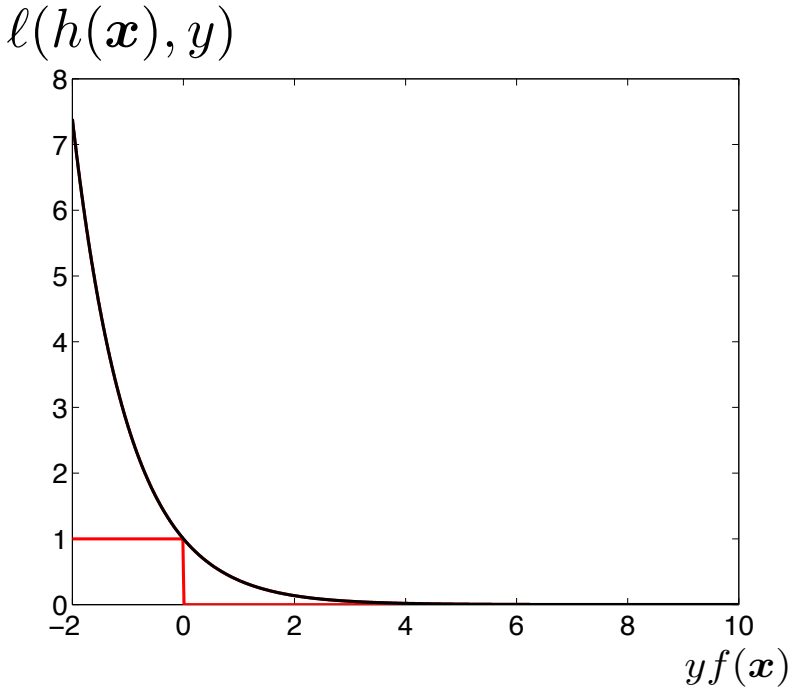
Surrogate loss

$0-1$ loss function $\ell(h(\boldsymbol{x}), y)$ is non-convex and difficult to optimize.

We can instead use a surrogate loss – what are examples?

Exponential Loss

$$\ell^{\mathrm{EXP}}(h(\boldsymbol{x}), y) = e^{-yf(\boldsymbol{x})}$$



Choosing the t-th classifier

Suppose a classifier $f_{t-1}(\boldsymbol{x})$, and want to add a weak learner $h_t(\boldsymbol{x})$

$$f(\boldsymbol{x}) = f_{t-1}(\boldsymbol{x}) + \beta_t h_t(\boldsymbol{x})$$

Note: $h_t(\cdot)$ outputs -1 or 1, as does sign $[f_{t-1}(\cdot)]$

How can we 'optimally' choose $h_t(\boldsymbol{x})$ and combination coefficient β_t?

Adaboost greedily *minimizes the exponential loss function*!

$$\begin{aligned}
(h_t^*(\boldsymbol{x}), \beta_t^*) &= \text{argmin}_{(h_t(\boldsymbol{x}),\beta_t)} \sum_n e^{-y_n f(\boldsymbol{x}_n)} \\
&= \text{argmin}_{(h_t(\boldsymbol{x}),\beta_t)} \sum_n e^{-y_n [f_{t-1}(\boldsymbol{x}_n) + \beta_t h_t(\boldsymbol{x}_n)]} \\
&= \text{argmin}_{(h_t(\boldsymbol{x}),\beta_t)} \sum_n w_t(n) e^{-y_n \beta_t h_t(\boldsymbol{x}_n)}
\end{aligned}$$

where we have used $w_t(n)$ as a shorthand for $e^{-y_n f_{t-1}(\boldsymbol{x}_n)}$

Outline

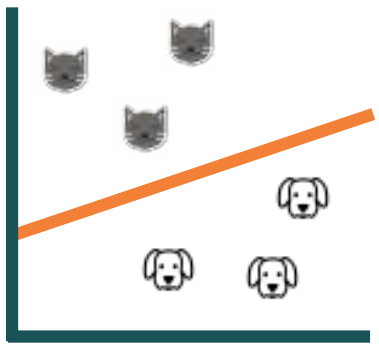
1. Review of Ensemble Methods
2. Neural Networks: Motivation
3. Neural Network Architectures and Forward Propagation
4. Choosing Activation Units
5. Choosing Neural Network Architectures

Neural Networks: Motivation

Why a "neural network"?

Many machine learning problems are easy for people but hard for machines.

input: cats and dogs

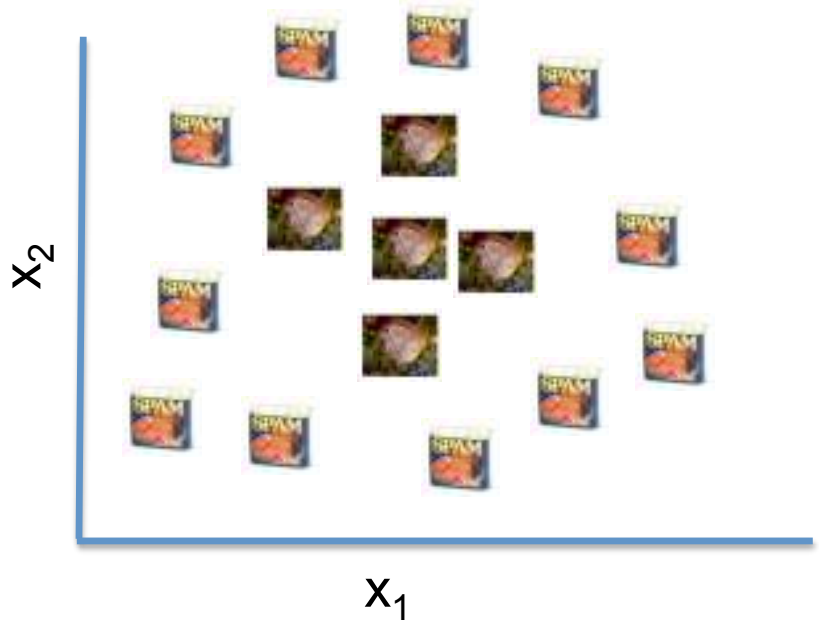


learn: $x \rightarrow y$ relationship

predict: y (*categorical*)

Each "node" in a neural network models a neuron in your brain. With enough neurons, we can learn to do very complex things.

Logistic Regression: How to handle complex boundaries?



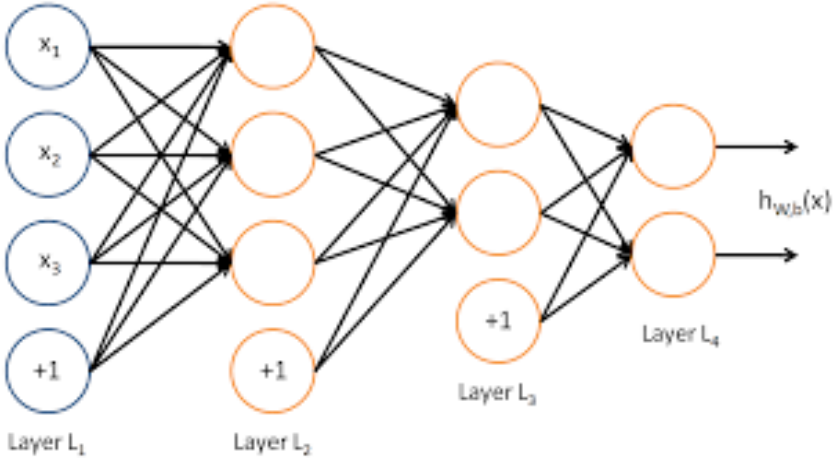
- This data is not linearly separable
- Use non-linear basis functions to add more features...

But what if we had a large number of features?

Each feature x_i is one pixel in an 100×100 input image

- Adding nonlinear (e.g. polynomial) features would result in an enormous $\phi(\mathbf{x})$
- Can we somehow only retain the important features?
- We will need to carefully hand-pick them, which can be hard and tedious...
- Neural networks automate this for us!

Neural Networks compress the set of features



- Start with feature vector **x** containing all pixels in the image
- Layer 1: distill the edges of the image
- Layer 2: distill triangles, circles, etc.
- Layer 3: recognize pointy ears, fur style etc.
- Layer 4: performs logistic regression on the features in layer 3

We cannot directly control what each layer learns; this depends on the training data.

Inspiration from Biology: How does our brain work?

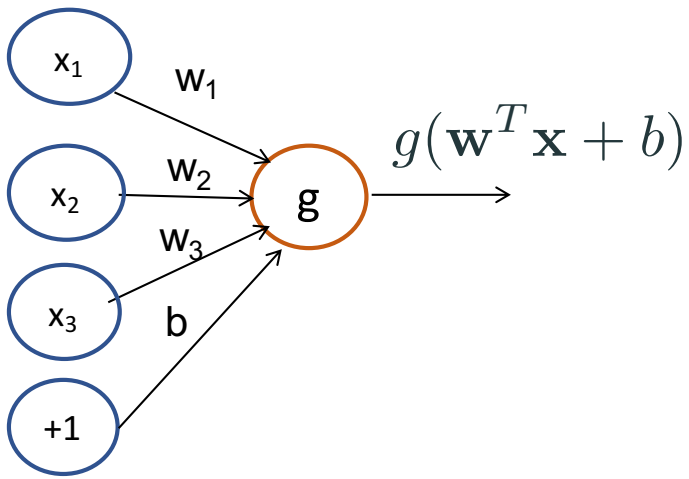
Each feature x_i is one pixel in an 100×100 input image

- Humans easily perform complex image or speech recognition tasks
- We cannot exactly describe a set of rules by which we distinguish cats vs. dogs, but we almost always know the correct answers when a new image is presented to us

Artificial Neuron model

Based on the biological insights, a mathematical model for an 'artificial' neuron was developed

- Each input x_i is multiplied by weight w_i
- Add a $+1$ input neuron, which is multiplied by the bias b
- Apply a non-linear function g to the weighted combination of the inputs, $\mathbf{w}^T\mathbf{x} + b$
- Different candidates for g: heaviside function, sigmoid, tanh, rectified linear unit, etc.

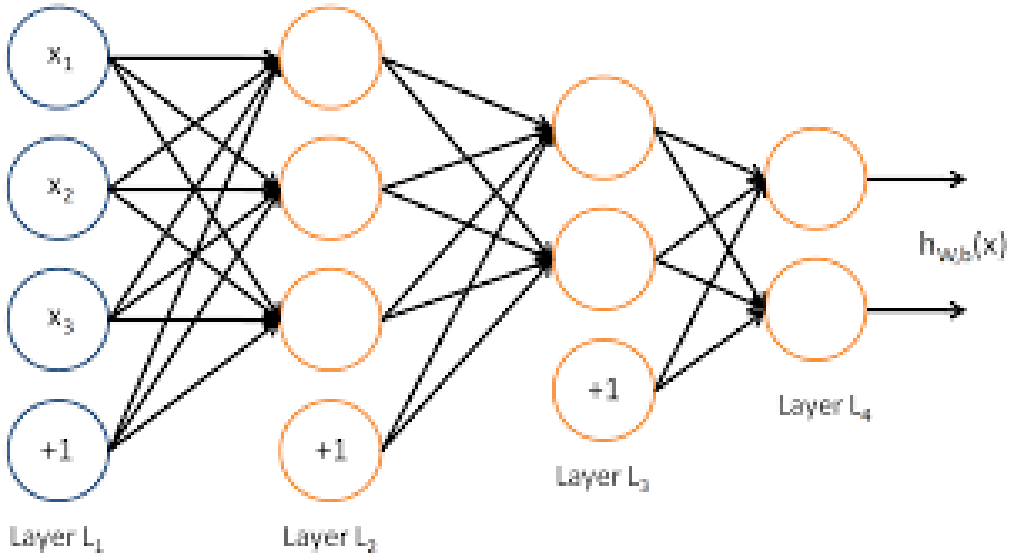


Single Artificial Neuron

Mimicking the human brain

Pass inputs through a "network" of neurons to obtain outputs.

- Neural networks are very good at handling large-scale data.
- They can learn very complex relationships.
- Requires careful configuration: what does this network look like?

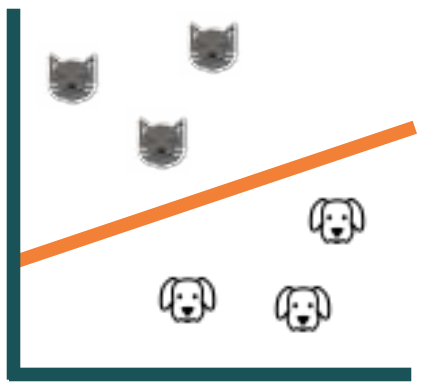


Each neuron is sometimes called a "node" or "unit" in the network. We group functions into "layers" depending on how many neurons their inputs have passed through since the original inputs.

Neural Network Architectures and Forward Propagation

Binary Logistic Regression

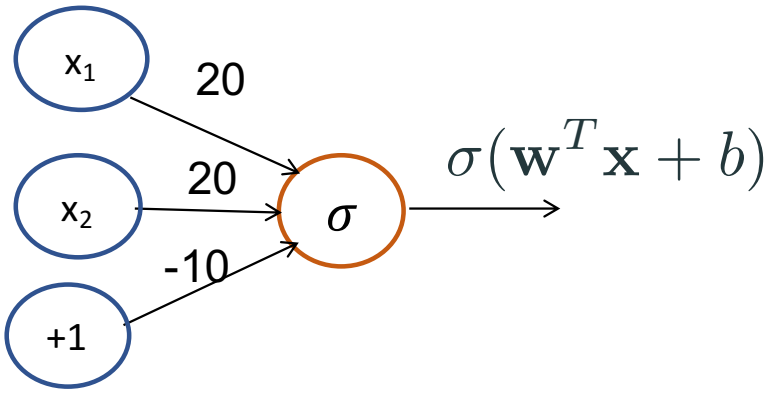
- Suppose g is the sigmoid function $\sigma(\mathbf{w}^T\mathbf{x} + b) = \frac{1}{1+e^{-(\mathbf{w}^T\mathbf{x}+b)}}$
- We can find a linear decision boundary separating two classes. The output is the probability of $\mathbf{x}$ belonging to class 1.
- This is binary logistic regression, which we already know.



linear decision boundary

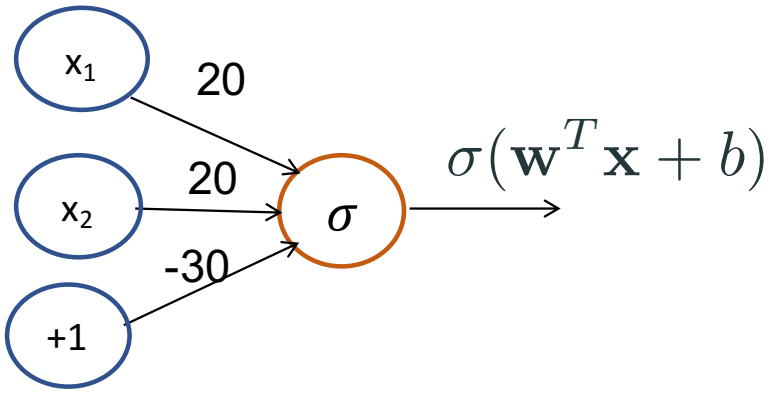
Example: Logic Gates

We can construct many common functions using just a single neuron



x_1	x_2	output
0	0	0
0	1	1
1	0	1
1	1	1

This is the OR gate



x_1	x_2	output
0	0	0
0	1	0
1	0	0
1	1	1

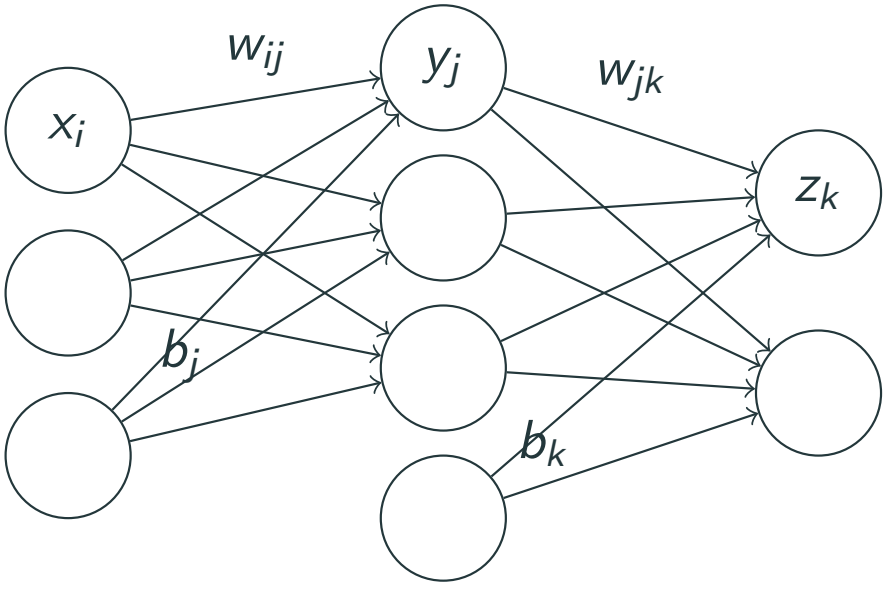
This is the AND gate

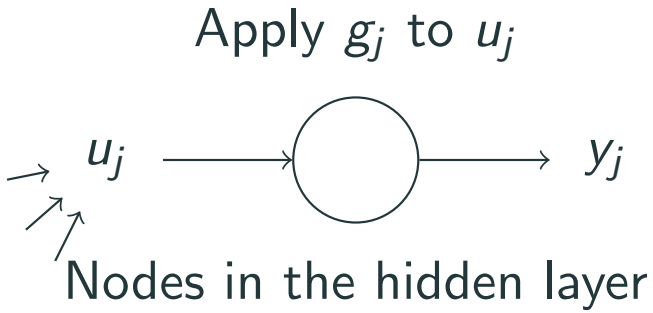
Can we build an XOR Gate?

x_1	x_2	output
0	0	0
0	1	1
1	0	1
1	1	0

- No, because this data is not linearly separable.
- We can create a combination of other logic gates $(x_1 + x_2)(\bar{x_1} + \bar{x_2})$
- Equivalent to creating a multi-layer neural network

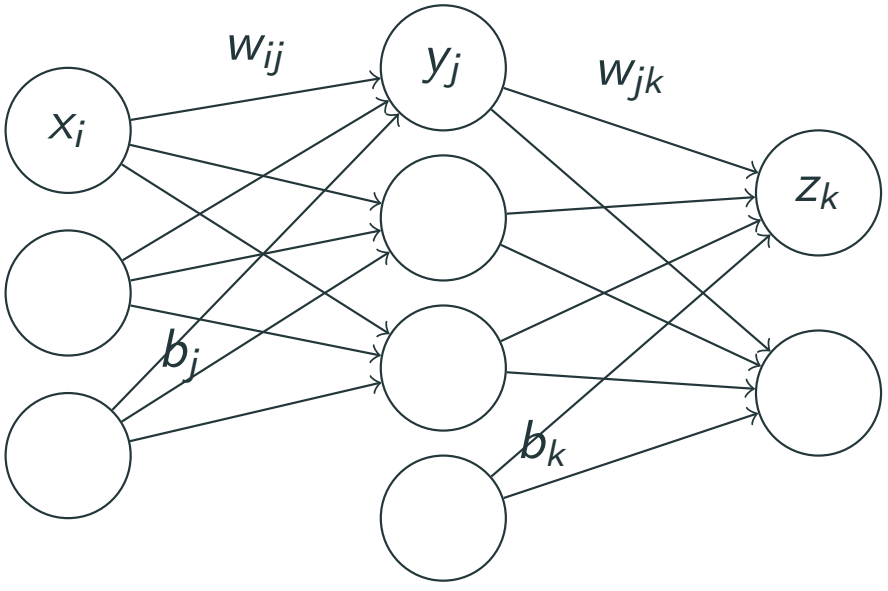
Multi-layer Neural Network

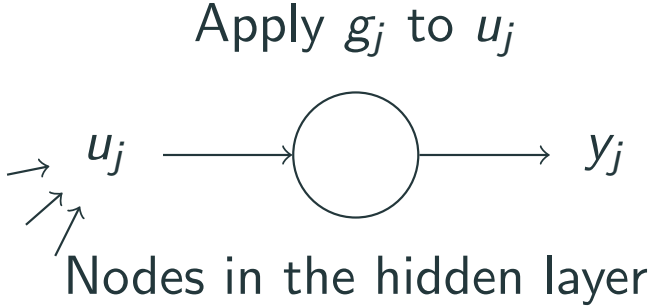




- w_{ij}: weights connecting node i in layer $(\ell - 1)$ to node j in layer ℓ.
- b_j, b_k: bias for nodes j and k.
- u_j, u_k: inputs to nodes j and k (where $u_j = b_j + \sum_i x_i w_{ij}$).
- g_j, g_k: activation function for node j (applied to u_j) and node k.
- $y_j = g_j(u_j)$, $z_k = g_k(u_k)$: output/activation of nodes j and k.
- t_k: target value for node k in the output layer.

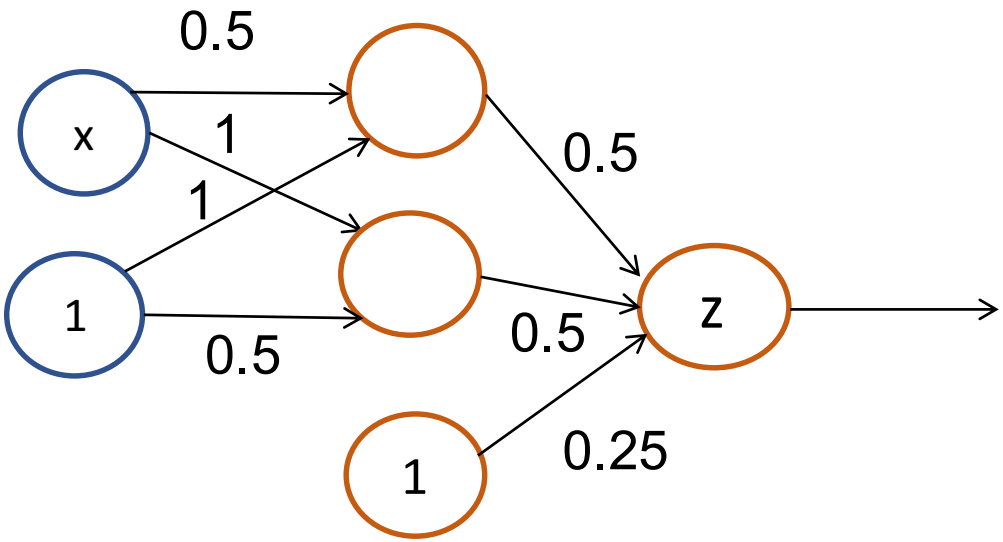
Neural Networks are very powerful





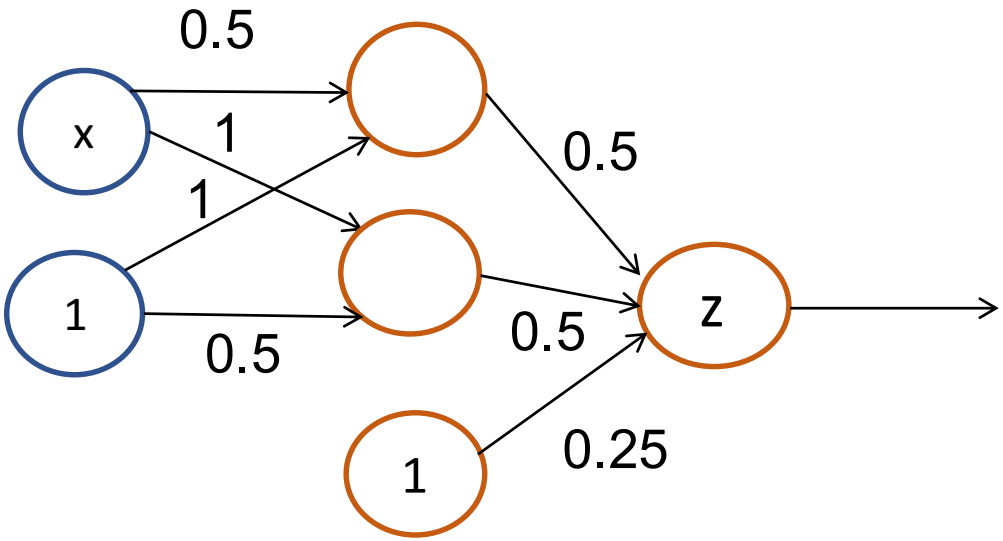
- With enough neurons and layers we can represent very complex input-output relationships
- Can be used for regression, classification, embedding, and many other ML applications

Forward-Propagation in Neural Networks



- Expressing outputs z in terms of inputs x is called forward-propagation.
- We perform forward-propagation when doing inference on a trained neural network.

Exercise: Forward-Propagation



- Outputs of the hidden layer are $\sigma(0.5x + 1)$ and $\sigma(x + 0.5)$
- Input to the last layer is $0.5\sigma(0.5x + 1) + 0.5\sigma(x + 0.5) + 0.25$
- $z = \sigma(0.5\sigma(0.5x + 1) + 0.5\sigma(x + 0.5) + 0.25)$

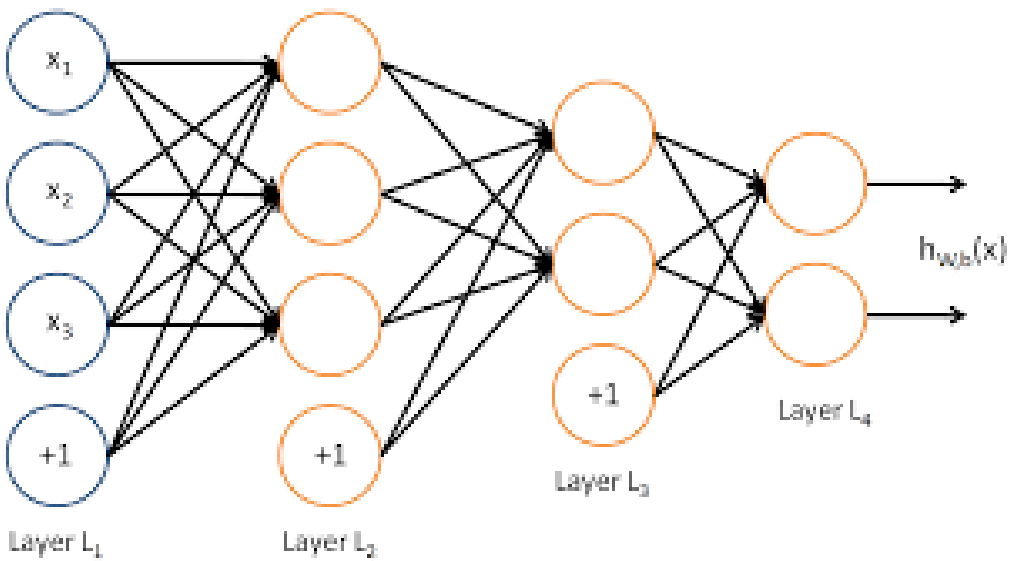
Choosing Activation Units

Activation choices for each layer

Input layer initially transforms the features.
Often uses linear, sigmoid, or tanh activations.

Hidden layers convert activated inputs to classification features.
Highly problem dependent!

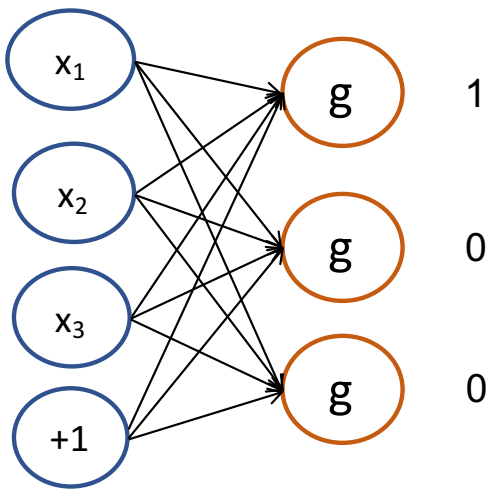
Output layer produces a classification decision.
Often, these are the probabilities of the input being in each class.



Softmax for Multi-class Regression

- If the target is takes C possible values
- If y belongs to the first class, the outputs should be $[1, 0, \ldots, 0]$
- Need to produce a vector $\hat{y}$ with $\hat{y}_i = \mathbb{P}\mathrm{r}(y = i|x)$
- Linear output layer ($g(x) = x$) first produces un-normalized log probabilities:

$$z = \mathbf{w}^T \mathbf{x} + b$$

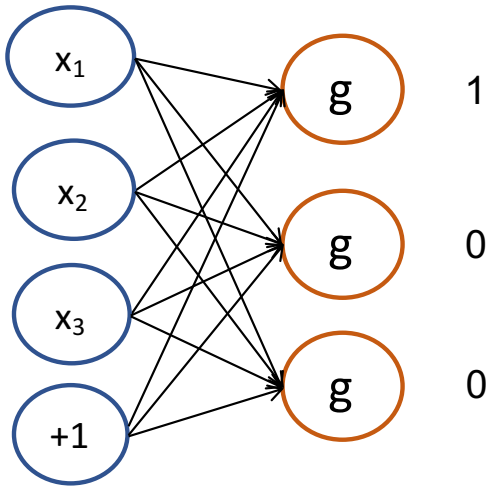


Multiclass Regression for $C = 3$

Softmax for Multi-class Regression

- Softmax:

$$\text{softmax}(z)_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

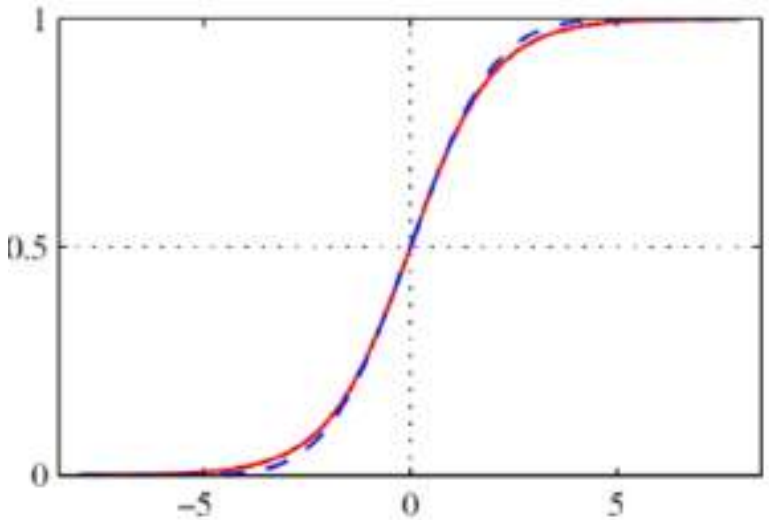


Multiclass Regression for $C = 3$

Sigmoid Units

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

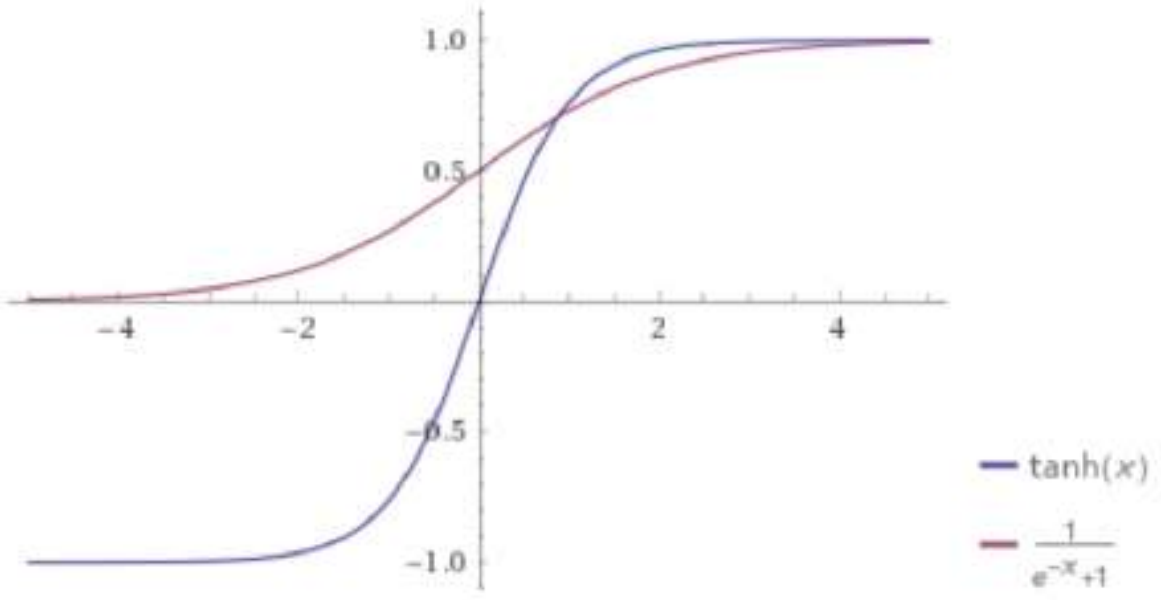
- Squashing type non-linearity: pushes output to range [0,1]



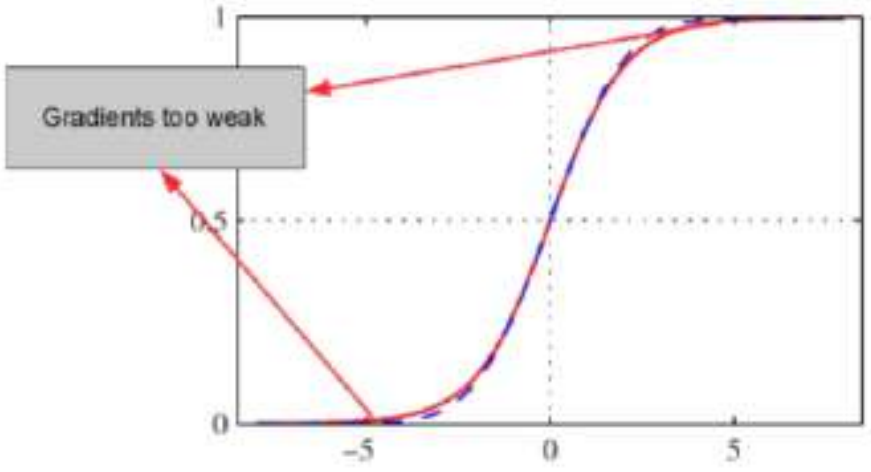
Tanh Units

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

- Related to sigmoid: $tanh(z) = 2\sigma(2z) - 1$
- Positive: Squashes output to range [-1,1], outputs are zero-centered
- Negative: Both tanh and sigmoid functions *saturate* at very small or large values



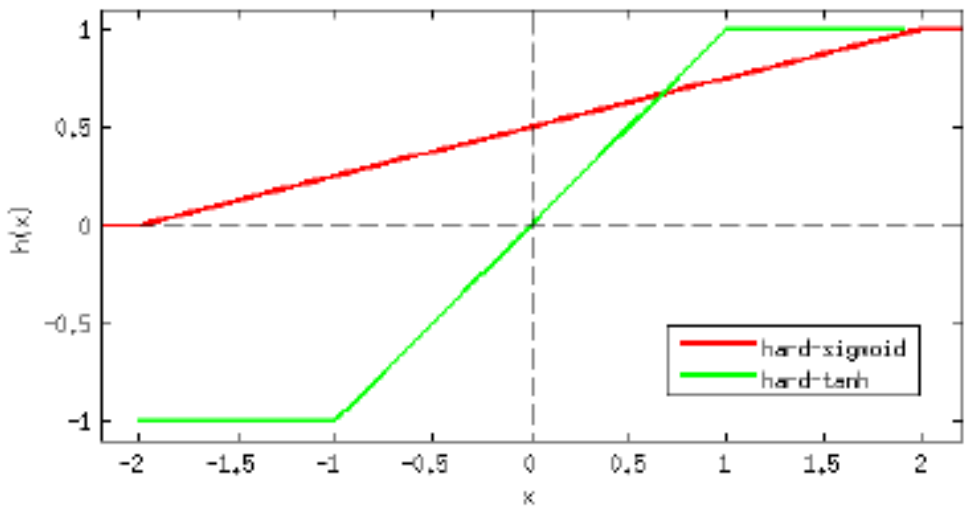
The vanishing gradients problem



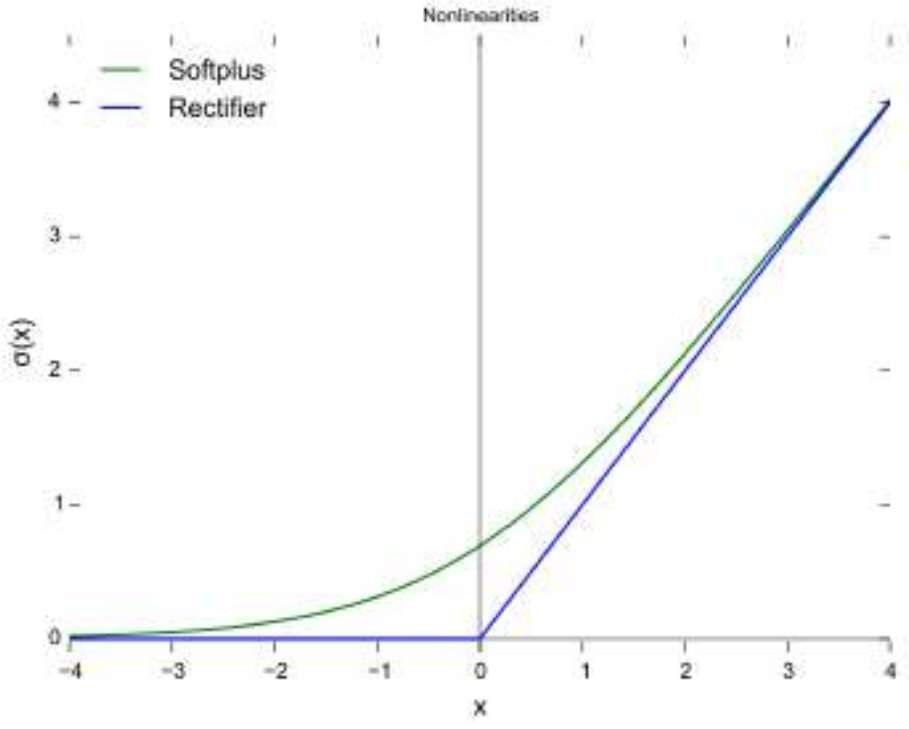
- Problem: Near-constant value across most of their domain, strongly sensitive only when z is closer to zero
- Saturation makes gradient based learning difficult (as we will see next week)

Hard Tanh and Hard Sigmoid

- To avoid the problem of vanishing gradients we can use piece-wise linear approximations to these functions
- This significantly reduces the computation complexity because gradients can take only one a few values

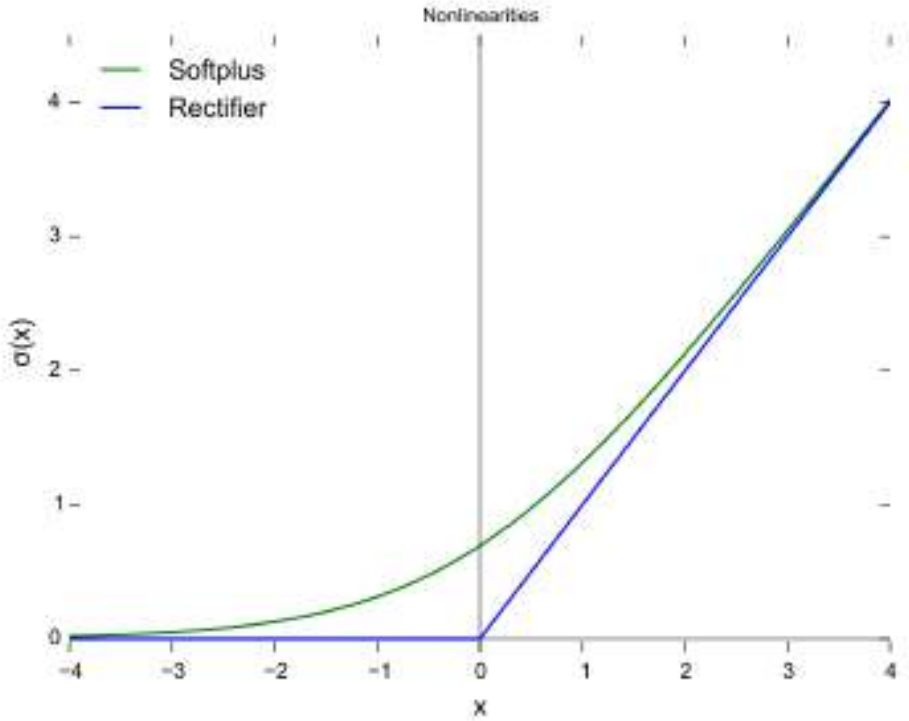


Rectified Linear Units



- Approximates the softplus function which is $\log(1 + e^z)$
- ReLu Activation function is $g(z) = max(0, z)$ with $z \in R$
- Similar to linear units. Easy to optimize!
- Give large and *consistent* gradients when active

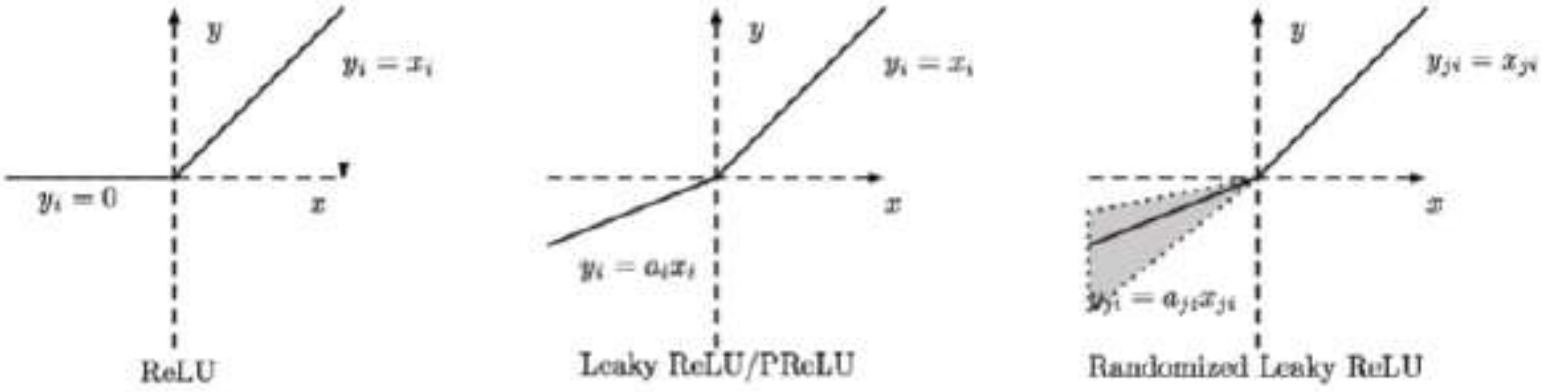
Rectified Linear Units



- Positives:
  - Gives large and *consistent* gradients (does not saturate) when active
  - Efficient to optimize, converges much faster than sigmoid or tanh
- Negatives:
  - Non zero centered output
  - Units "die" i.e when inactive they will never update

Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$
- $g(z, a)_i = max(0, z_i) + a_i min(0, z_i)$
  - **Leaky ReLU:**(Mass et al., 2013) Fix a_i to a small value e.g 0.01
  - **Parametric ReLU** (He et al., 2015) Learn a_i
  - **Randomized ReLU**(Xu et al., 2015) Sample a_i from a fixed range during training, fix during testing



Activation choices for each layer

Output layer produces a classification decision.

- Probabilities of the input being in each class.
- Often uses sigmoid, softmax, or tanh activations.

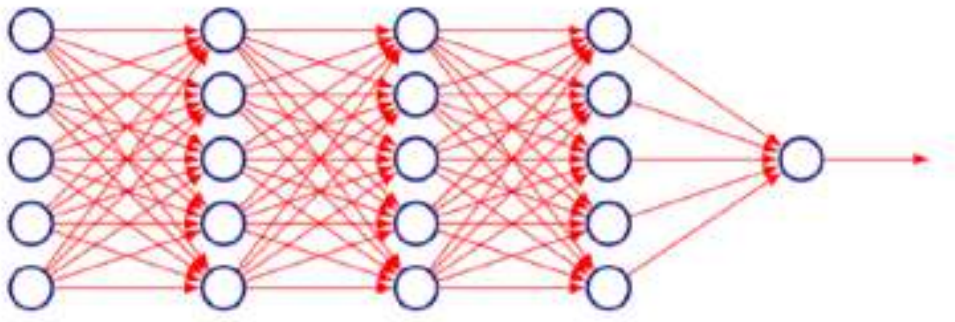
Hidden layers convert activated inputs to classification features.

- ReLU, ELU and variants are popular choices.
- Highly problem dependent!

Input layer initially transforms the features.
Often uses linear, sigmoid, or tanh activations.

Choosing Neural Network Architectures

Architecture design



- First layer: $h^{(1)} = g^{(1)}(W^{(1)^T} x + b^{(1)})$
- Second layer: $h^{(2)} = g^{(2)}(W^{(2)^T} h^{(1)} + b^{(2)})$
- How do we decide *depth, width?*
- In theory how many layers *suffice?*

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, a one hidden layer neural network can represent this function.
- But not guaranteed that our training algorithm will be able to learn that function!
- Gives no guidance on how large the network will be (exponential size in worst case)

Advantages of depth

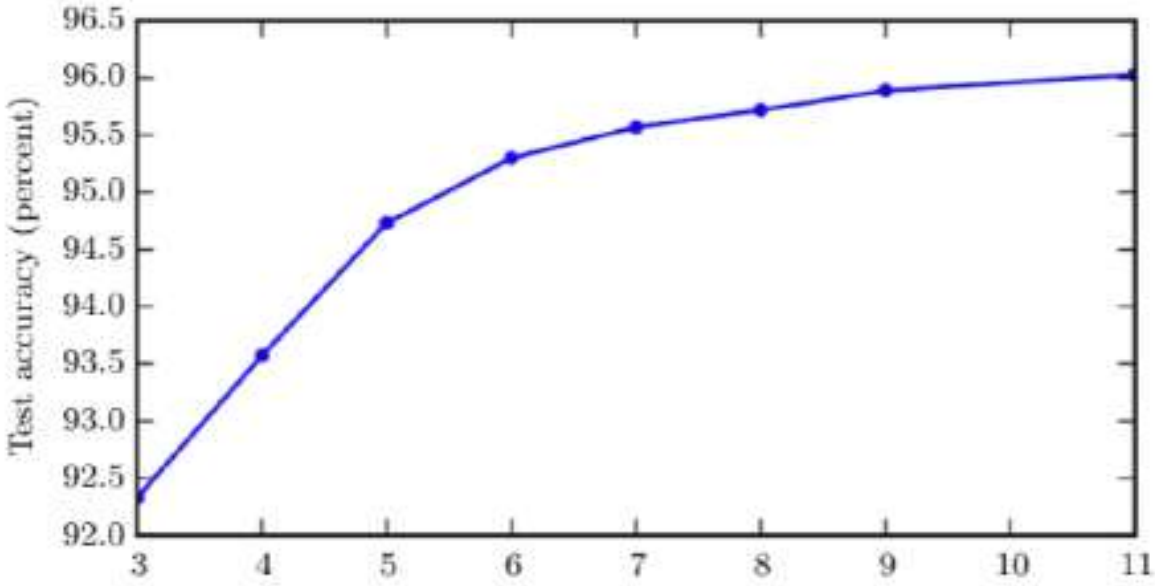
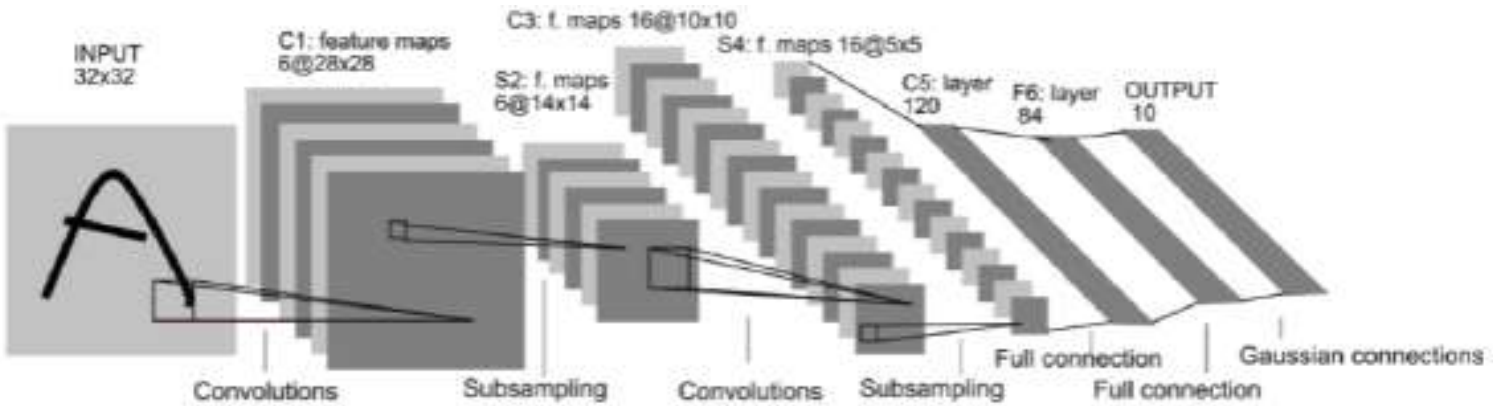


Figure 3: Goodfellow et al., 2014

- Increasing the depth of a neural network generally improves test accuracy.
- Coupled with computation advances, e.g. GPU

Deep convolutional networks

- Deep supervised neural networks are generally too difficult to train
- One notable exception: Convolutional neural networks (CNN)
- Convolutional nets were inspired by the visual system's structure.
- They typically have more than five layers, a number of layers which makes fully-connected neural networks almost impossible to train properly when initialized randomly.



Example: LeNet 5 (LeCun, 1998).

Advantages of deep convolutional networks

- Compared to standard feedforward neural networks with a similarly-sized layer
  - CNNs have much fewer connections and parameters
  - and so they are easier to train
- Usually applied to image datasets (where convolutions have a long history).

LeNet 5

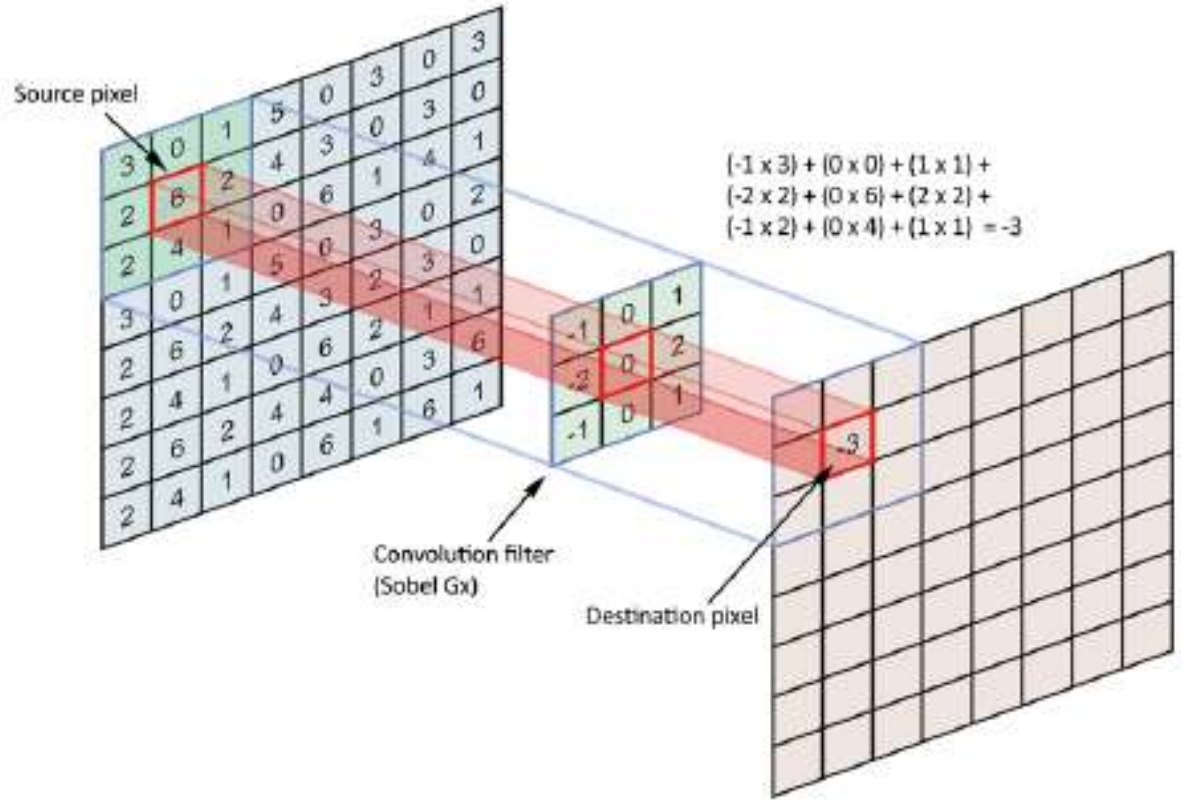
Y. LeCun, L. Bottou, Y. Bengio and P. Haffner: **Gradient-Based Learning Applied to Document Recognition**, *Proceedings of the IEEE, 86(11):2278-2324, November **1998***

Convolutional network layers

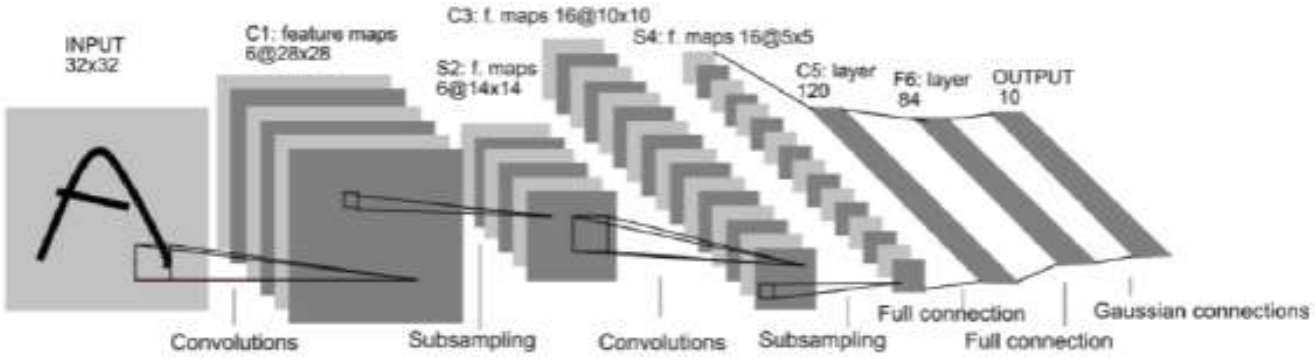
Convolve subsets of an image with a small filter.

- Each pixel in the output image is a weighted sum of the filter and a subset of the input.
- Learn the values in the filter (these are your parameters, or weights).

Many fewer parameters (and connections) than a feedforward network.



LeNet 5, Layer C1



C1: Convolutional layer with 6 feature maps of size 28X28 $C1^k (k = 1..6)$

Each unit of C1 has 5x5 receptive field in the input layer.

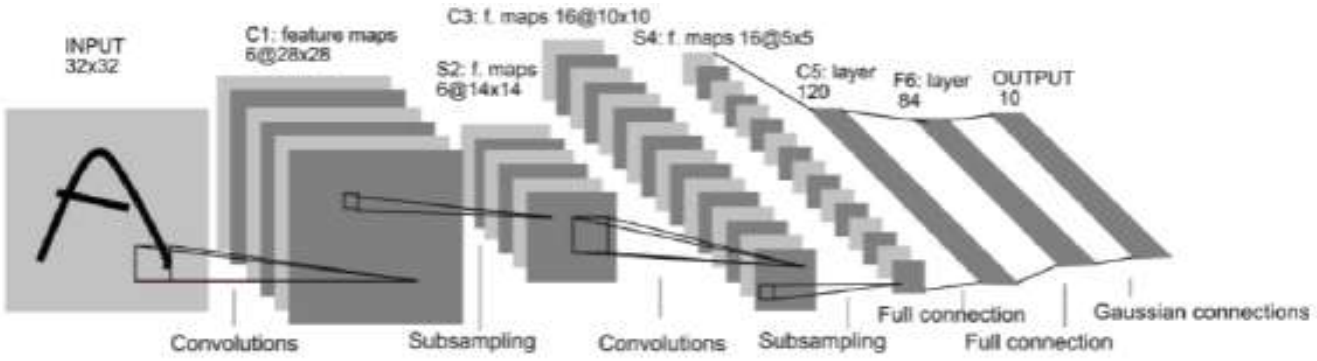
- Topological structure
- Sparse connections
- Shared weights

$$(5 * 5 + 1) * 6 = 156 \text{ parameters to learn}$$

$$\text{Connections: } 28 * 28 * (5 * 5 + 1) * 6 = 122304$$

If it was fully connected, we had (32*32+1)*(28*28)*6 parameters

LeNet 5, Layer S2



S2: Sub-sampling layer with 6 feature maps of size 14×14

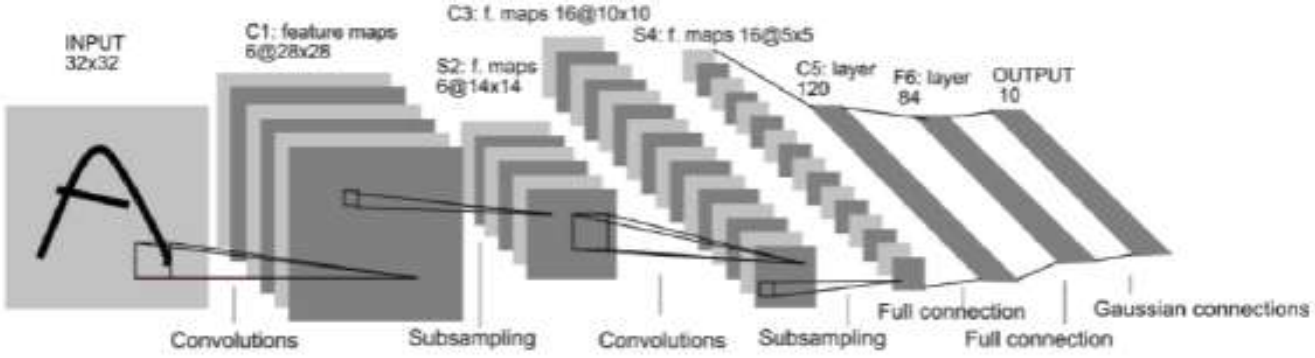
2×2 non-overlapping receptive fields in C1

$$S2_{ij}^k = tanh(w_1^k \sum_{s,t=0}^{1} C1_{2i-s,2j-t}^k + w_2^k)$$

Layer S2: 6*2=12 trainable parameters

Connections: $14 * 14 * (2 * 2 + 1) * 6 = 5880$

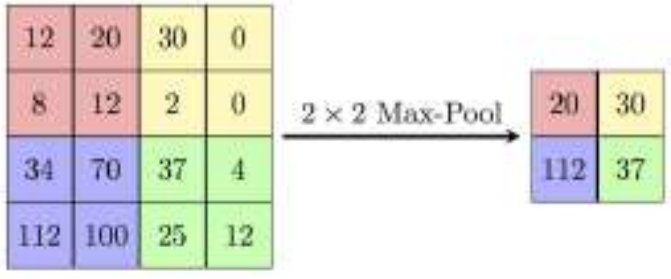
LeNet 5, Layer S2



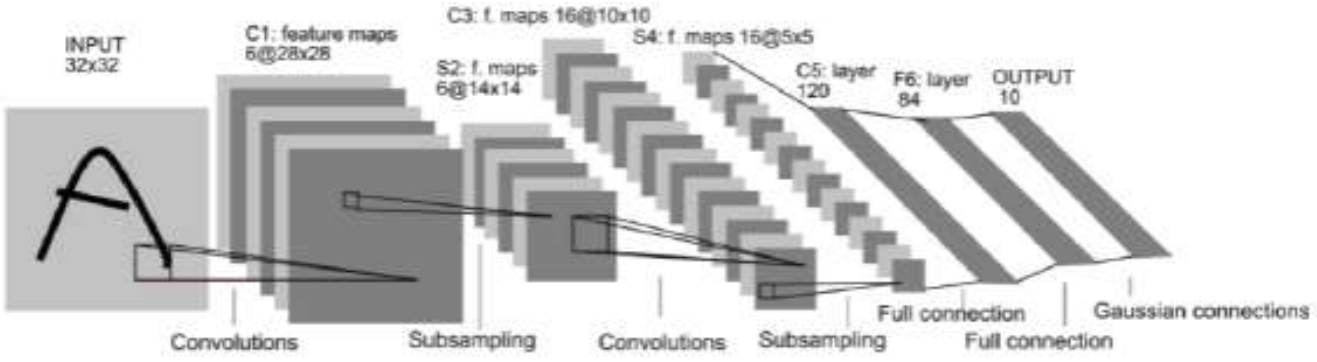
S2: Sub-sampling layer with 6 feature maps of size 14×14

2×2 non-overlapping receptive fields in C1

These days, we typically use



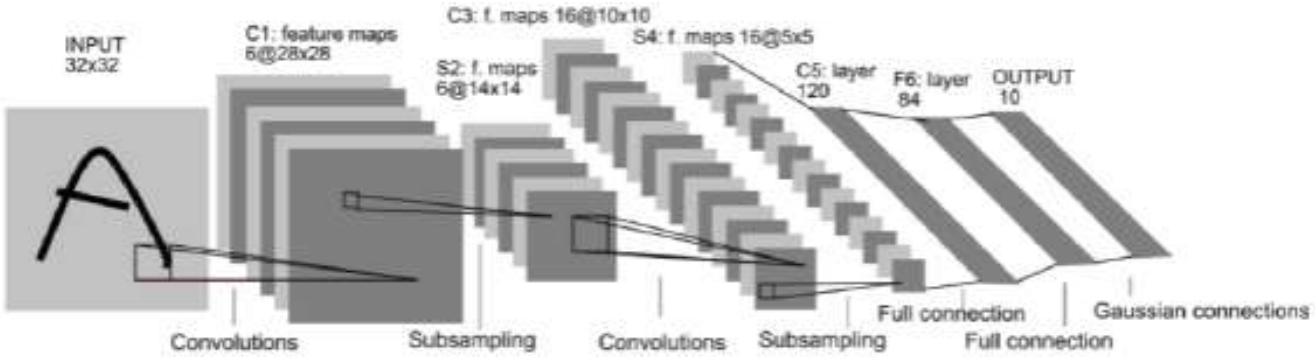
LeNet 5, Layer C5



- C5: Convolutional layer with 120 feature maps of size 1x1
- Each unit in C5 is connected to all 16 5x5 receptive fields in S4

Layer C5: $120 * (16 * 25 + 1) = 48120$ trainable parameters and connections (Fully Connected)

LeNet 5, Layer F6



- Layer F6: 84 fully connected nodes. 84*(120+1)=10164 trainable parameters and connections.
- Output layer: 10 RBF (One for each digit)

$$y_i = \sum_{j=1}^{84} (x_j - w_{ij})^2$$

Where i =1,2,..10

$84 = 7x12$, stylized image

Weight update: Backpropagation

GoogLeNet (Szegedy et al., 2015)

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Today's networks can go much deeper than LeNet!

Summary

You should know:

- Why we call these models "neural" networks.
- How to train a perceptron.
- Basic structure of a neural network.
- How to perform forward-propagation.
- Common choices for neuron activations.
- What we mean by a "deep" or "convolutional" neural network.