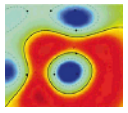
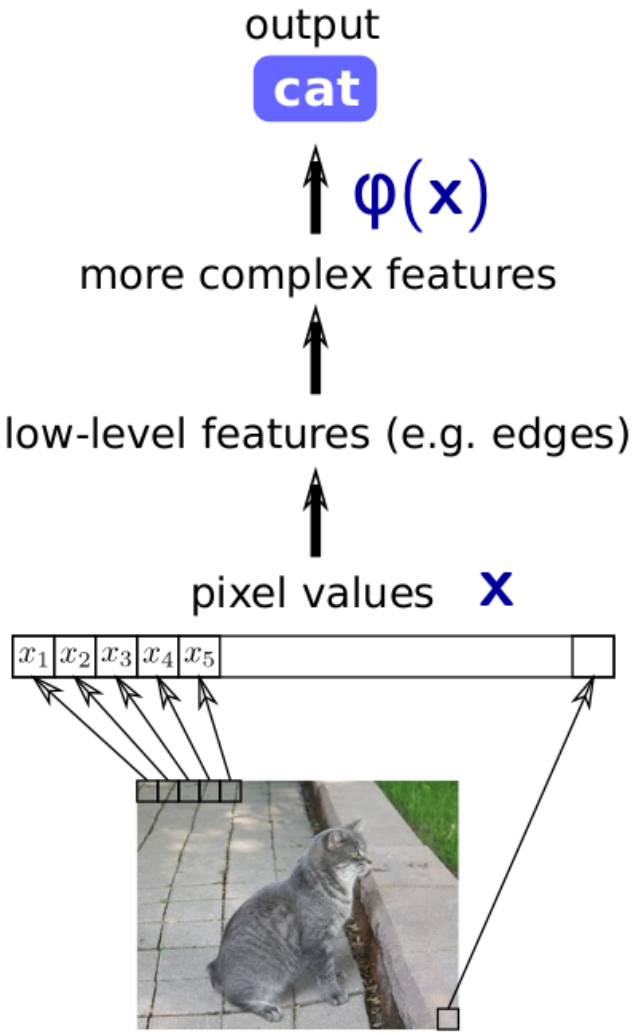
SoSe 2018: Deep Neural Networks

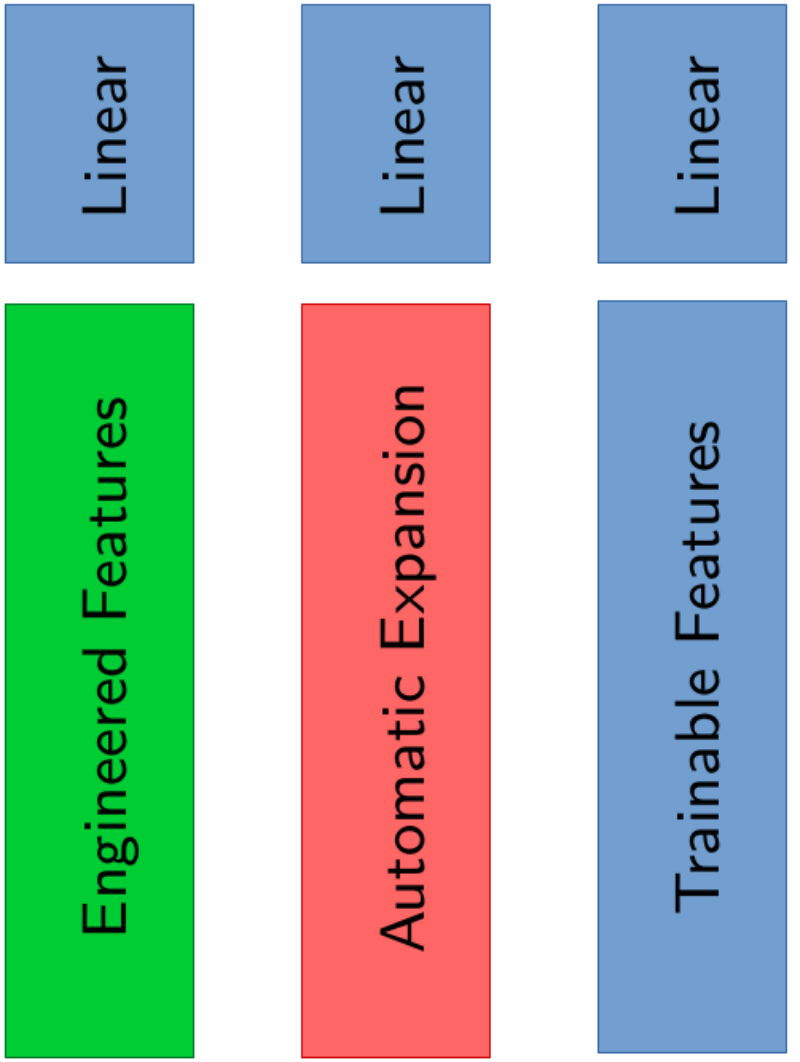
Lecture 2: Representation and Propagation

Machine Learning Group
Technische Universität Berlin

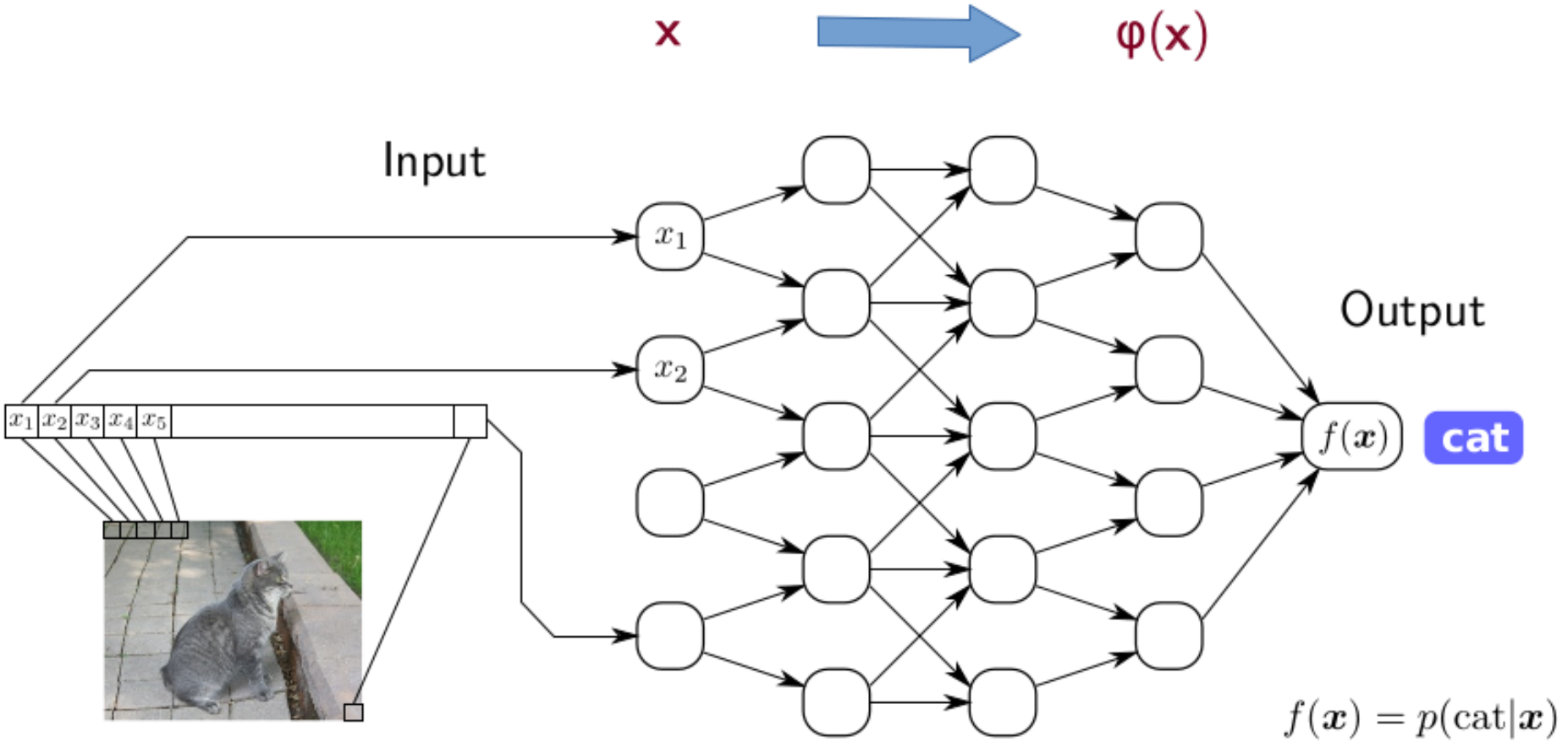


Recap: Deep Representations

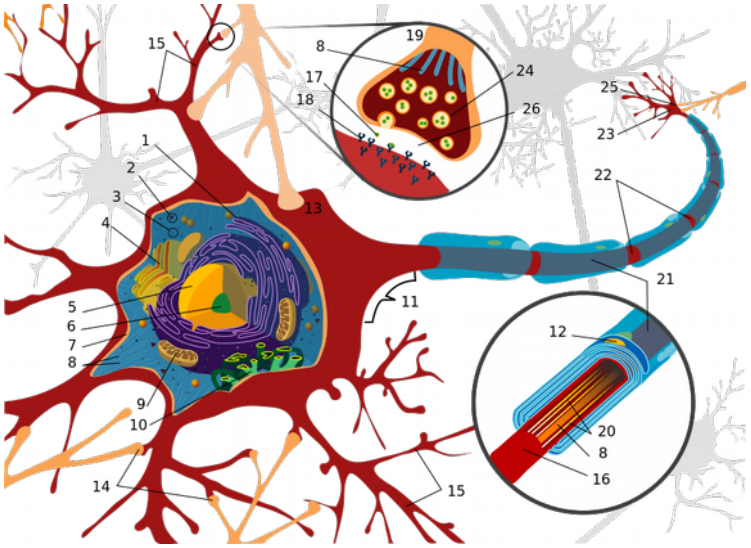




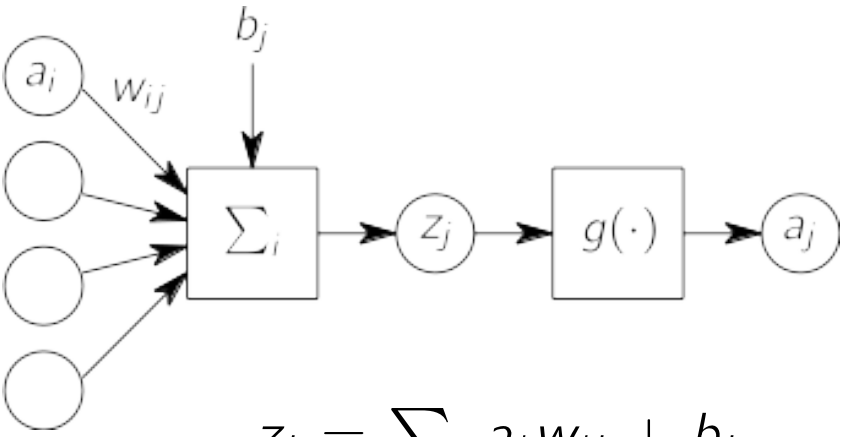
Recap: Graphical View of a Neural Network



Recap: The Neuron Building Block



Highly sophisticated physical system, with spatio-temporal dynamics.



$$z_j = \sum_i a_i w_{ij} + b_j$$

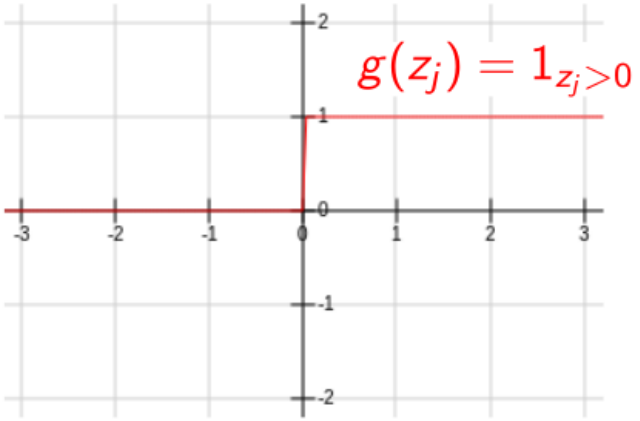
$$a_j = g(z_j)$$

Simple multivariate and nonlinear function.

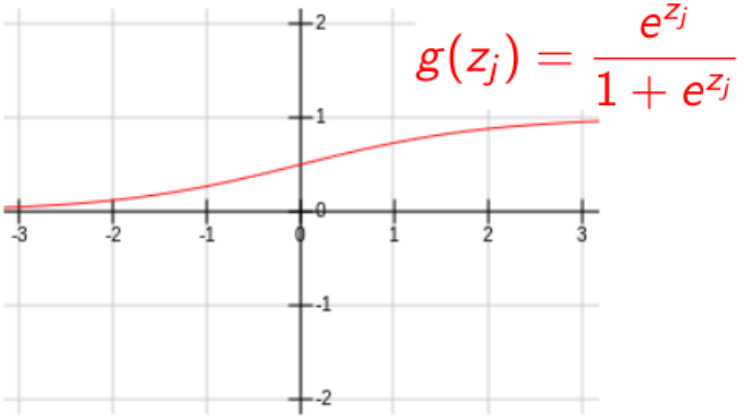
Common denominator: (1) They both have an adaptation mechanism. (2) Ability to represent abstractions derives from interconnecting a large number of them.

Examples of Activation Functions

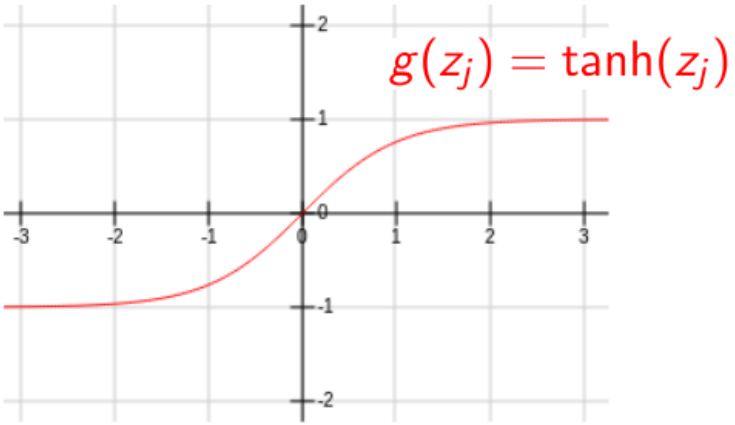
threshold function



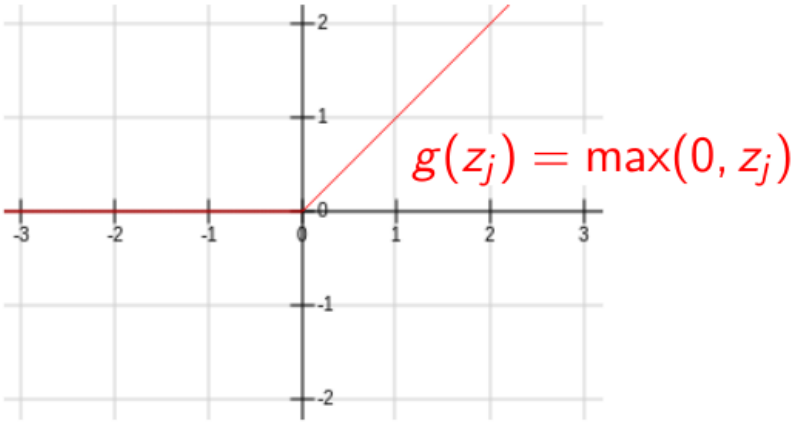
logistic sigmoid



hyperbolic tangent

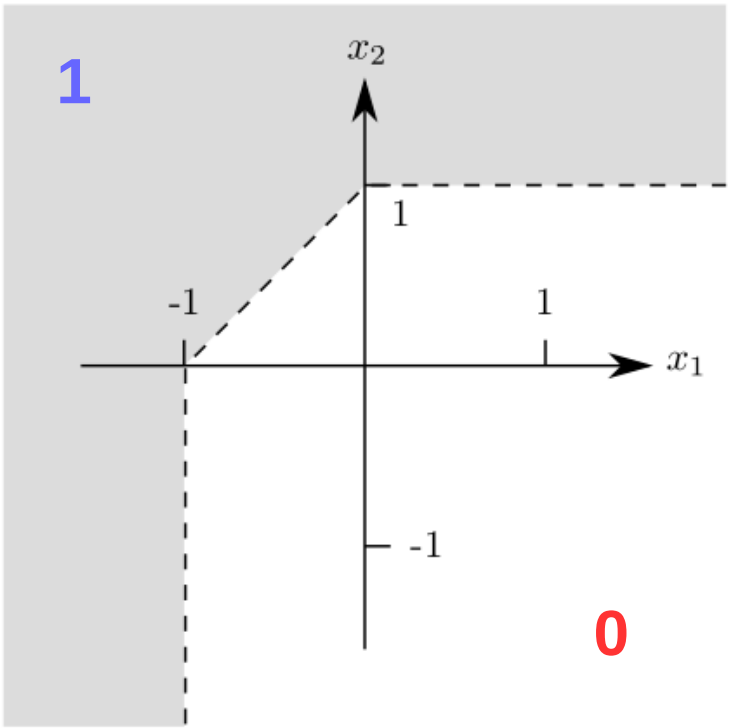


rectified linear unit

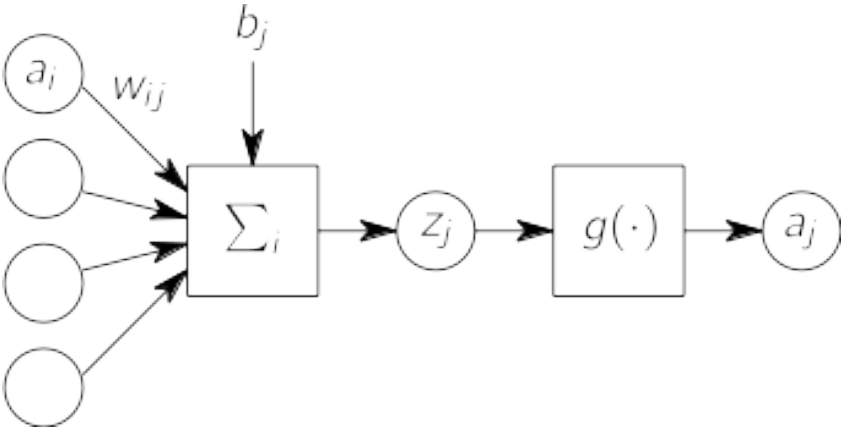


Representing Decision Boundaries (1)

Decision boundary to implement in $\mathbb{R}^2$



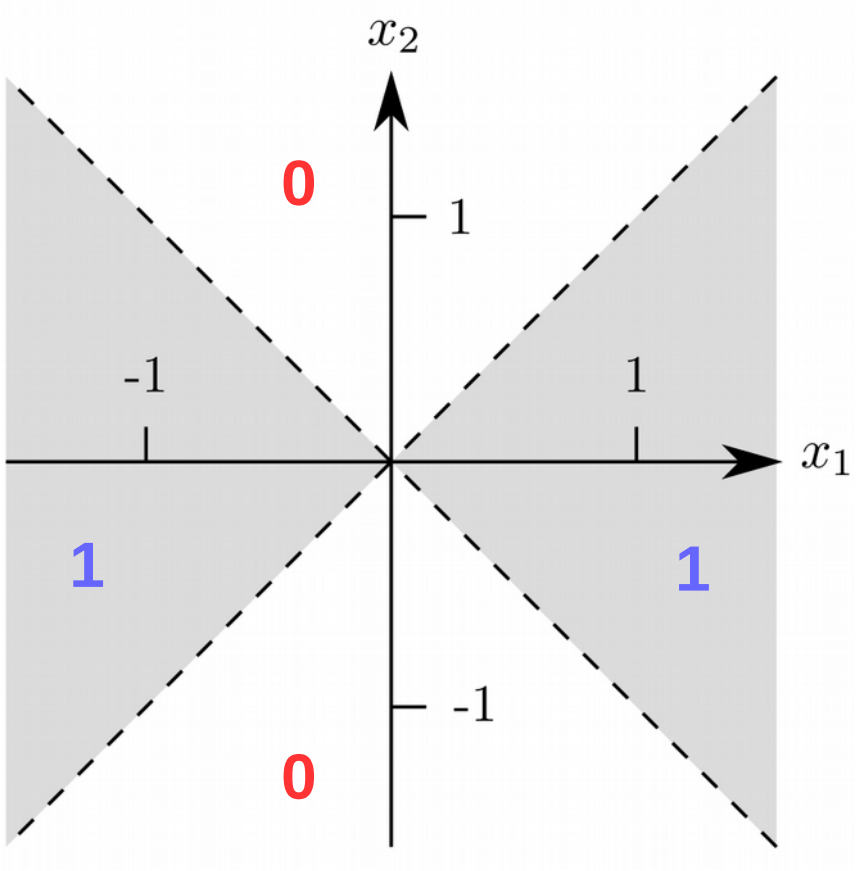
Type of neuron available



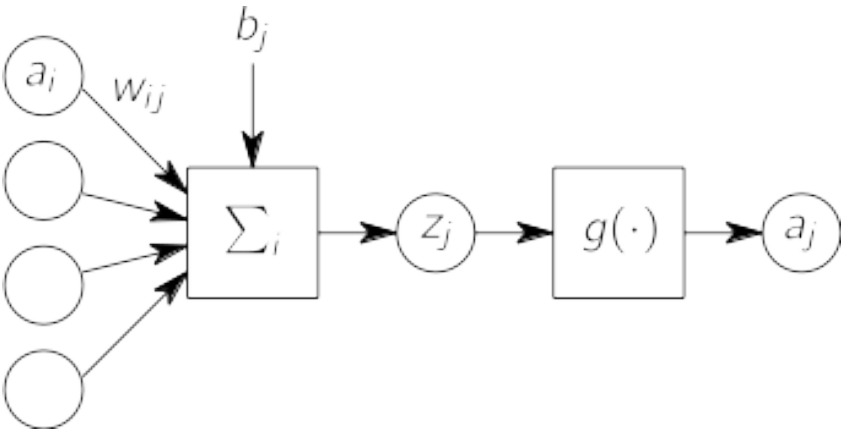
$$z_j = \sum_i a_i w_{ij} + b_j$$

$$a_j = 1_{z_j > 0}$$

Representing Decision Boundaries (2)



Type of neuron available



$$z_j = \sum_i a_i w_{ij} + b_j$$

$$a_j = 1_{z_j > 0}$$

Two Useful Properties of Neural Networks

1. Universal Approximation

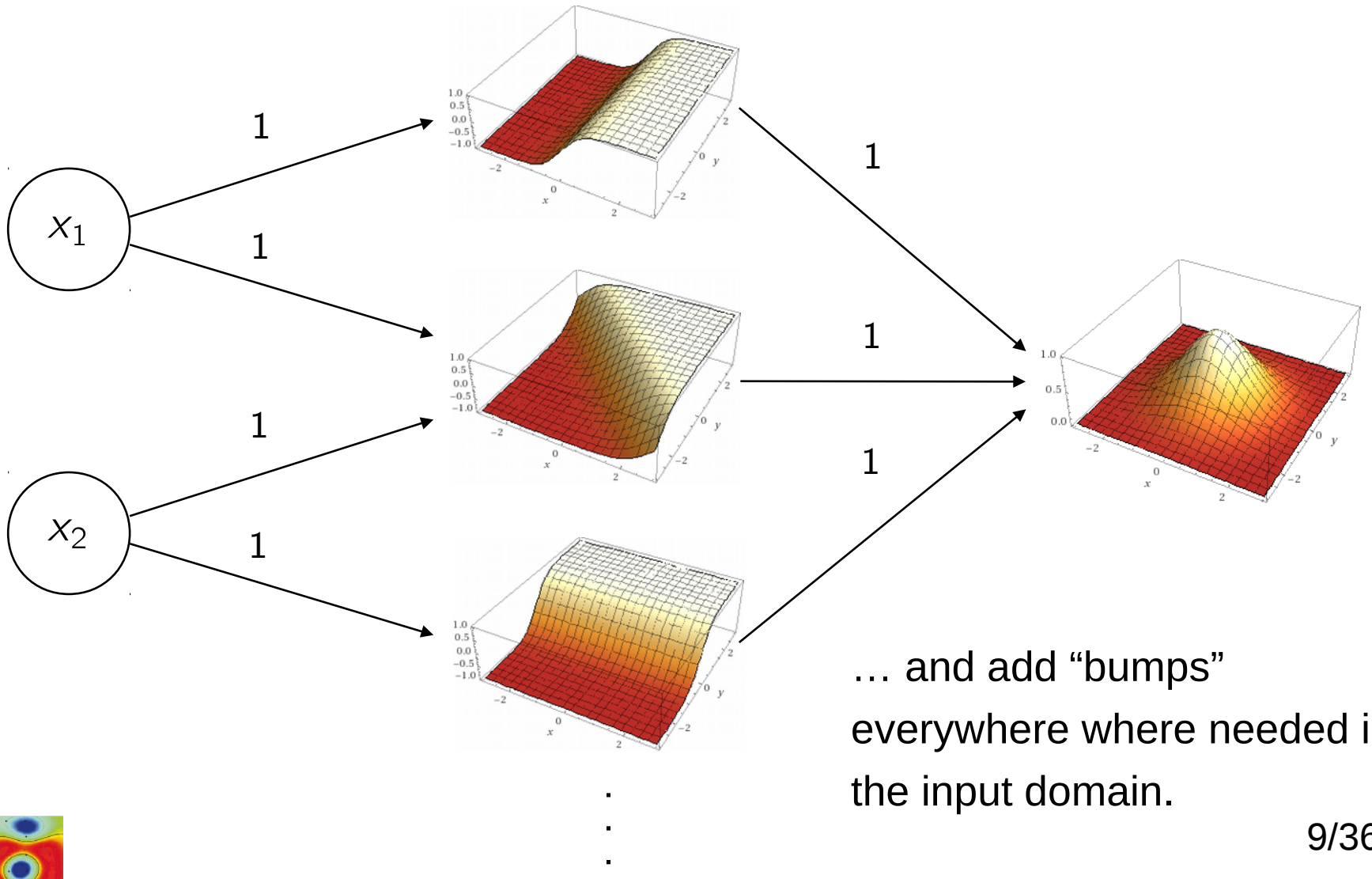
With enough neurons, a neural network can represent *any* nonlinear function

2. Compactness

In practice, a limited number of neurons is sufficient to implement the nonlinear function.

Universal Approximation

The basic idea:



… and add “bumps” everywhere where needed in the input domain.

Proof of Universal Approximation

Sketch proof taken from the book Bishop'95 Neural Network for Pattern Recognition, p. 130–131, (after Jones'90 and Blum&Li'91):

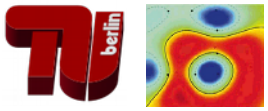
Consider the special class of functions

$$y : \mathbb{R}^2 \to \mathbb{R} \quad \text{where input variables are called} \quad x_1, x_2$$

We will show that any two-layer network with threshold functions as nonlinearity can approximate $y(x_1,x_2)$ up to arbitrary accuracy.

We first observe that any function of x_2 (with x_1 fixed) can be approximated as an infinite Fourier series.

$$y(x_1, x_2) \simeq \sum_s A_s(x_1) \cdot \cos(s \cdot x_2)$$



Proof of Universal Approximation

We first observe that any function of x_2 (with x_1 fixed) can be approximated as an infinite Fourier series.

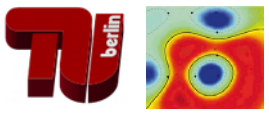
$$y(x_1, x_2) \simeq \sum_s A_s(x_1) \cdot \cos(s \cdot x_2)$$

Similarly, the coefficients themselves can be expressed as an infinite Fourier series:

$$y(x_1, x_2) \simeq \sum_s \sum_l A_{sl} \cdot \cos(l x_1) \cdot \cos(s x_2)$$

We now make use of a trigonometric identity to write the function above as a sum of cosines:

$$\cos(\alpha)\cos(\beta) = \tfrac{1}{2}\cos(\alpha + \beta) + \tfrac{1}{2}\cos(\alpha - \beta)$$



Proof of Universal Approximation

We now make use of a trigonometric identity to write the function above as a sum of cosines:

$$\cos(\alpha)\cos(\beta) = \tfrac{1}{2}\cos(\alpha+\beta) + \tfrac{1}{2}\cos(\alpha-\beta)$$

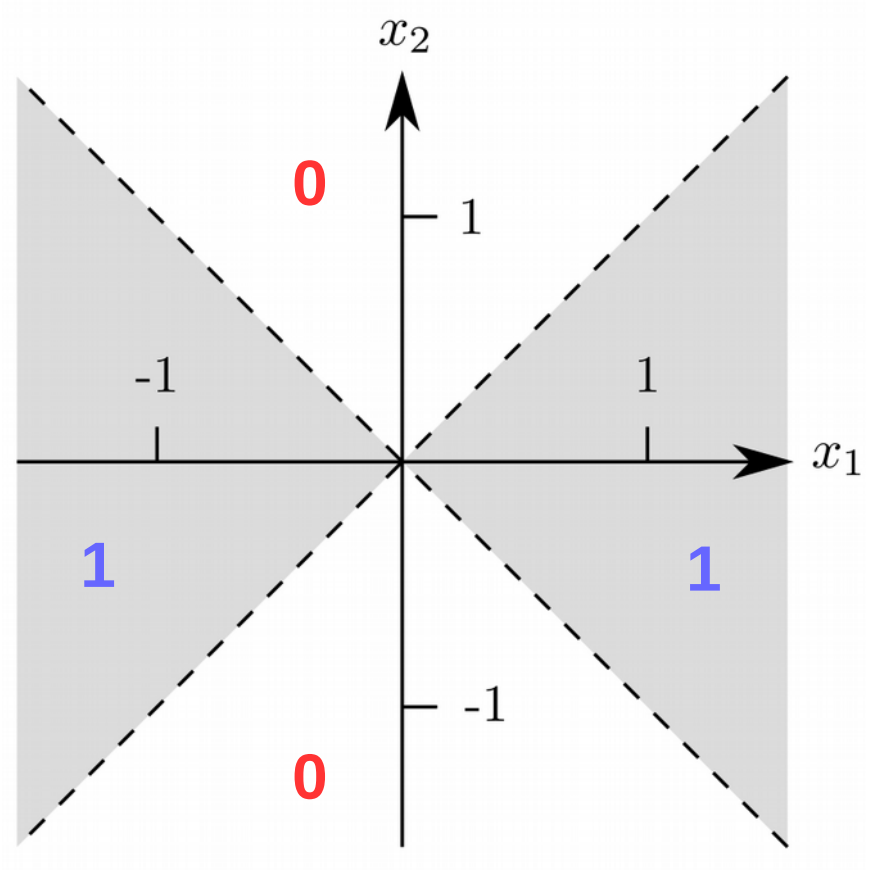
Thus, the function to approximate can be written as a sum of cosines, where each of them receives a linear combination of the input variables:

$$y(x_1, x_2) \simeq \sum_{j=1}^{\infty} v_j \cdot \cos(x_1 w_{1j} + x_2 w_{2j})$$

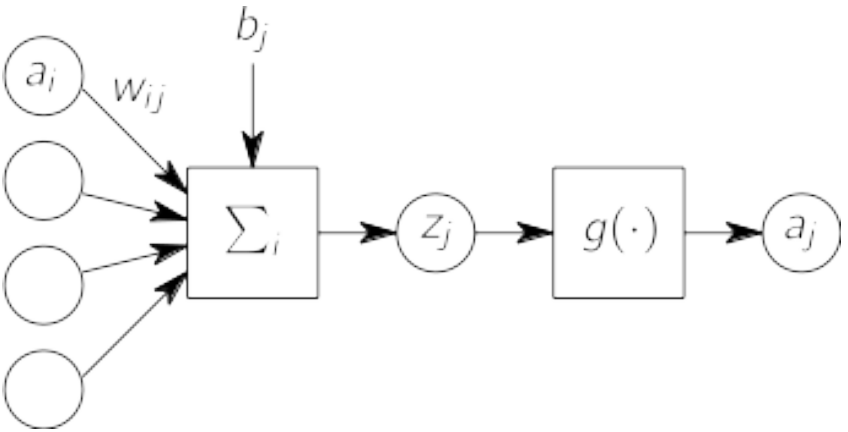
This is a two-layer neural network, except for the cosine nonlinearity. The latter can however be approximated by a superposition of a large number of step functions.

$$cos(z) = \lim_{\tau \to 0} \sum_i \underbrace{[\cos(\tau \cdot (i+1)) - \cos(\tau \cdot i)]}_{\text{constant}} \cdot \underbrace{1_{z > \tau \cdot (i+1)}}_{\text{step function}} + \text{const.}$$

Representing Decision Boundaries (2)



Type of neuron available



$$z_j = \sum_i a_i w_{ij} + b_j$$

$$a_j = 1_{z_j > 0}$$

Two Useful Properties of Neural Networks

1. Universal Approximation

With enough neurons, a neural network can represent *any* nonlinear function

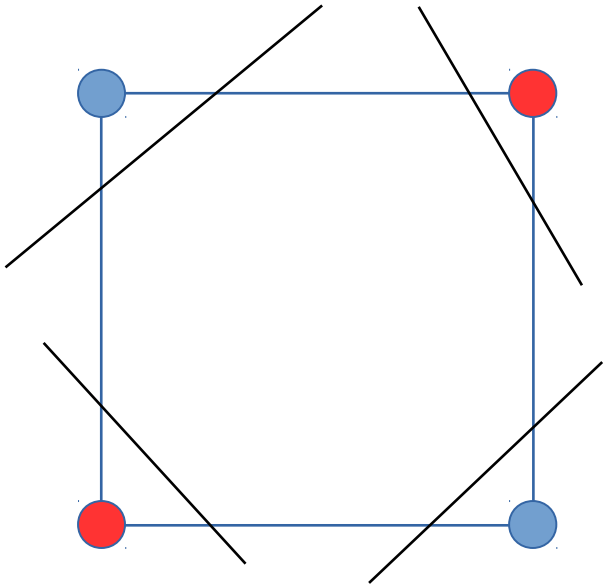
2. Compactness

In practice, a limited number of neurons is sufficient to implement the nonlinear function.

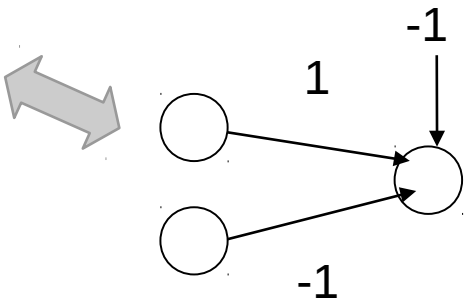
Compactness of Approximation

Function to approximate $f : \{0,1\}^d \rightarrow \{-1,1\}$

This can be achieved by a network with 2^d neurons.



Dedicate one neuron to each data point (template detector).

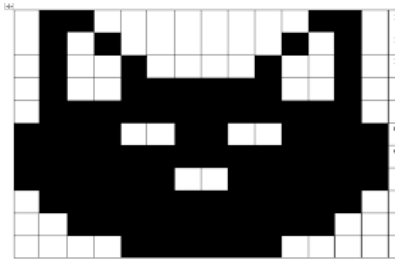


Question: do we really need exponentially many neurons in practice?

Compactness of Approximation

Function to approximate $f : \{0,1\}^d \to \{-1,1\}$

$x \in \{0,1\}^d$

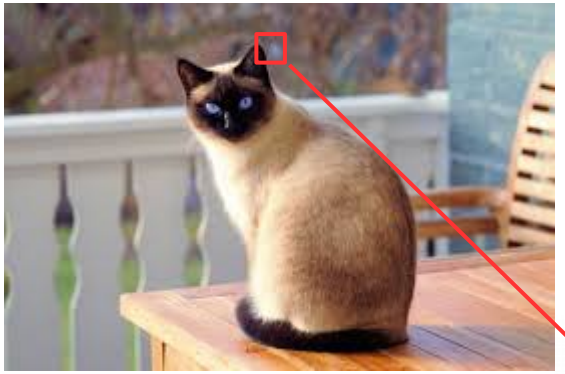


Question: how many neurons to detect all possible cats?

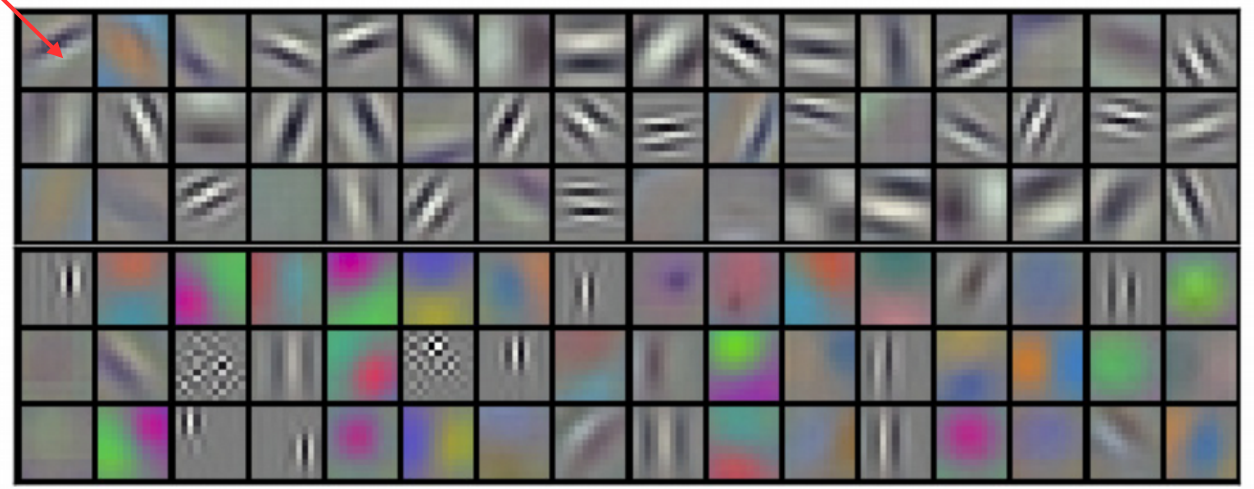
Observation 1: Data to classify lies in a lower-dimensional manifold. → Project the data on the low-dimensional manifold before applying detection neurons.

Observation 2: Problem has compositional structures (e.g. left ear can be detected independently from right ear). → Detect individual objects in the first layer.

Compactness of Approximation

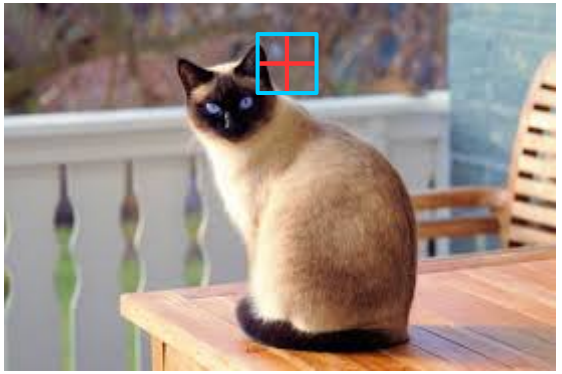


Observation: To detect all meaningful content of a patch, one will need much less than $16^{\#\text{pixels}}$ neurons. Perhaps only a few hundreds (e.g. 400) neurons.



Reason: Natural images occupy a low-dimensional manifold in the pixel space.

Compactness of Approximation



Furthermore, it doesn't matter where the feature exactly is. E.g. a cat with an ear translated by a few pixels is still a cat.

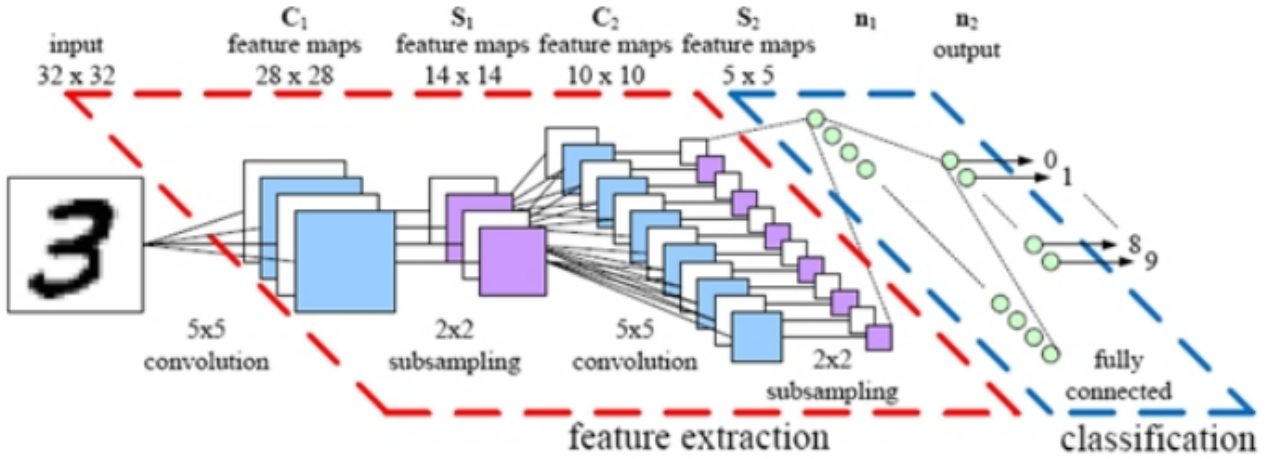
Idea: pool features in adjacent patches.

Result: Dimensionality of the input space is further reduced without removing class information.

before detection		**before pooling**		**after pooling**
$16^{\#\text{pixels}}$	$\gg$	$400^{\#\text{patches}}$	$\gg$	$400^{\#\text{patches}/4}$

Compactness of Approximation

A practical example of neural network that compactly approximates classes is the the convolutional neural network.

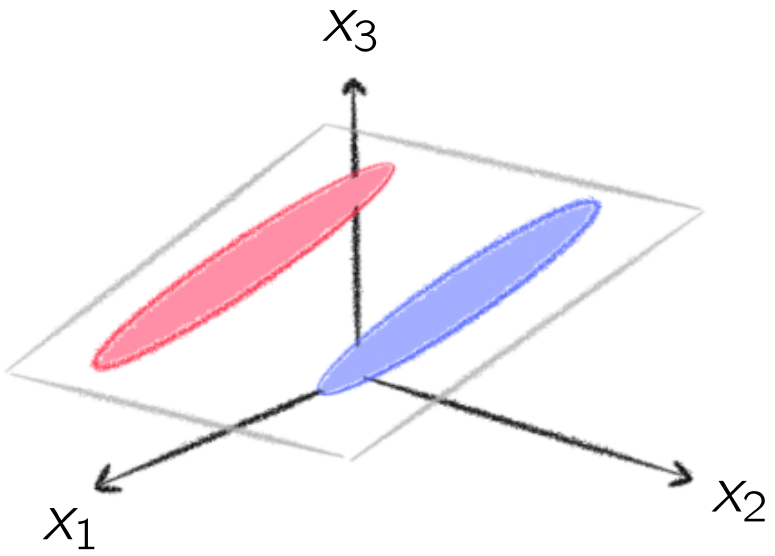


(source: Peemen et al. 2011: Speed sign detection and recognition by convolutional neural networks).

Convolutional neural network applies repeatedly detection and pooling layers (→ progressively reduce input dimensionality).

Already conceptualized in the 1980's (Fukushima's Neocognitron).

Geometrical Intuition in $\mathbb{R}^3$



Step 1:

Project on lower-dimensional data manifold

$$\mathbb{R}^3 \rightarrow \mathbb{R}^2$$

example, convolutional layers

Step 2:

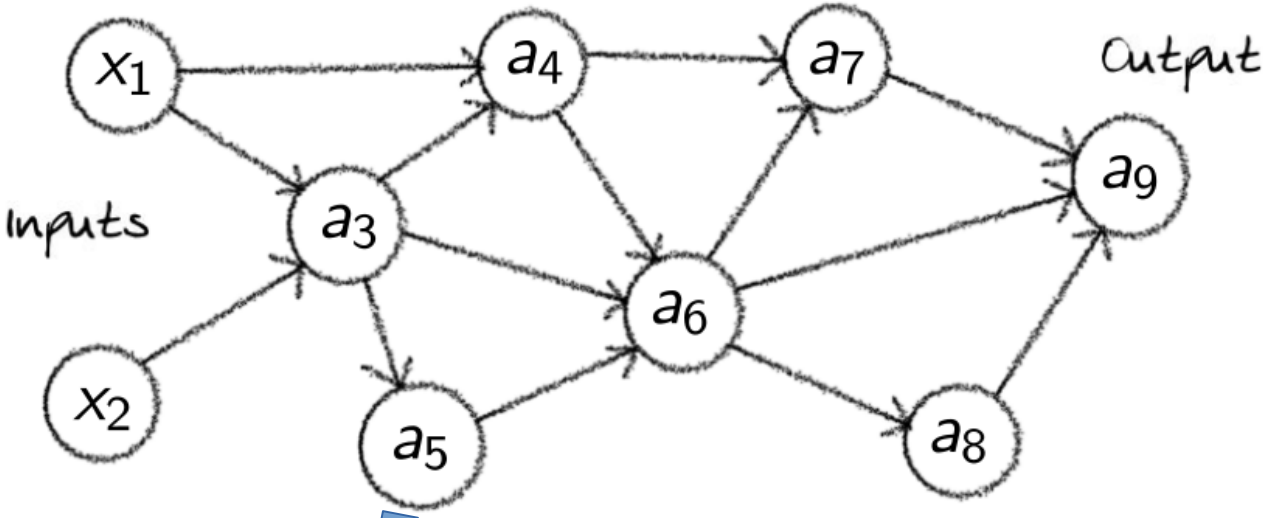
Project on class-discriminative subspace

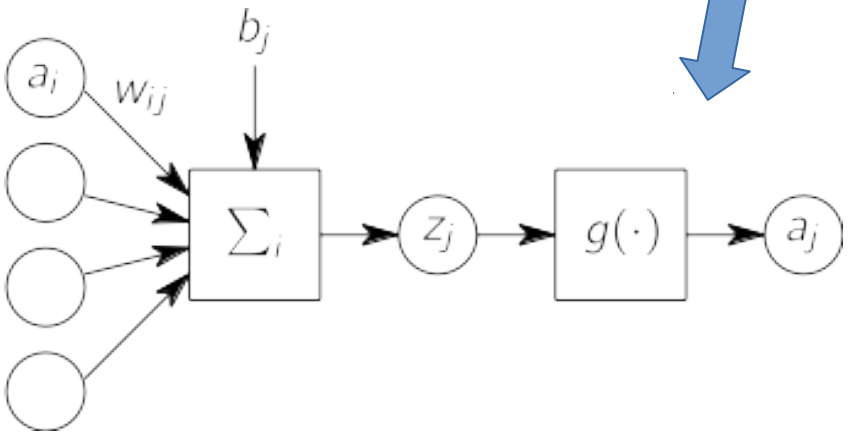
$$\mathbb{R}^2 \rightarrow \mathbb{R}^1$$

example, pooling layers

Part 2: Propagation in a Neural Network

Example of a Simple Neural Network



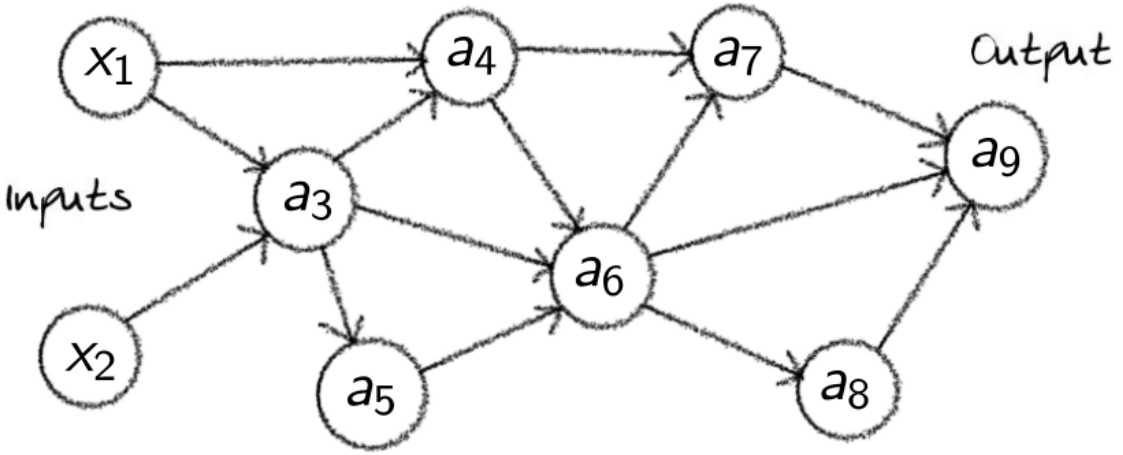


$$z_j = \sum_i a_i w_{ij} + b_j$$

$$a_j = g(z_j)$$

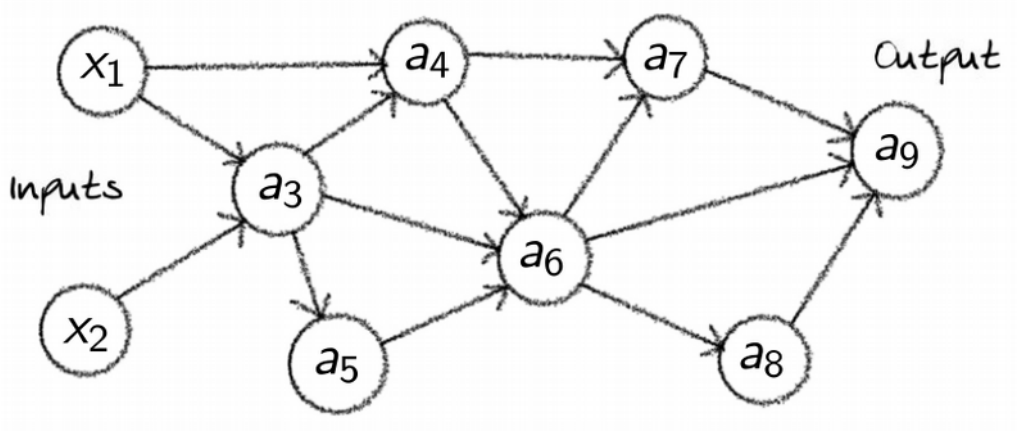
Question: how to compute the forward pass?

Computing the Forward Pass



$$z_3 = x_1 \cdot w_{13} + x_2 \cdot w_{23} + b_3 \qquad a_3 = g(z_3)$$
$$z_4 = x_1 \cdot w_{14} + a_3 \cdot w_{34} + b_4 \qquad a_4 = g(z_4)$$
$$z_5 = a_3 \cdot w_{35} + b_5 \qquad a_5 = g(z_5)$$
$$z_6 = a_3 \cdot w_{36} + a_4 \cdot w_{46} + a_5 \cdot w_{56} + b_6 \qquad a_6 = g(z_6)$$
$$z_7 = a_4 \cdot w_{47} + a_6 \cdot w_{67} + b_7 \qquad a_7 = g(z_7)$$
$$z_8 = a_6 \cdot w_{68} + b_8 \qquad a_8 = g(z_8)$$
$$z_9 = a_6 \cdot w_{69} + a_7 \cdot w_{79} + a_8 \cdot w_{89} + b_9 \qquad a_9 = g(z_9)$$

Computing the Forward Pass



Layered architectures
run a **for** loop from input to output

General feed-forward architecture
require a graph traversal algorithm

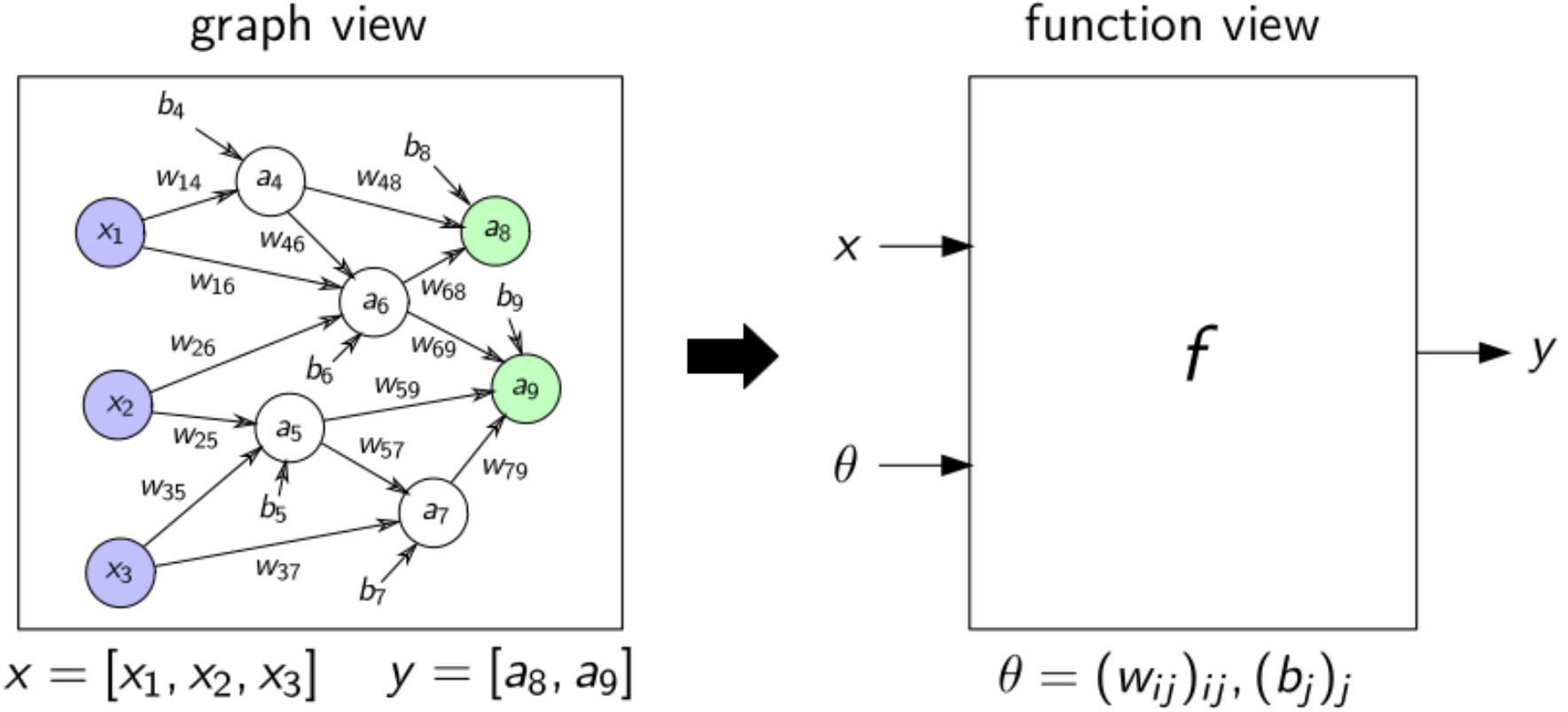
General graphs
may not converge
(due to recurrent connections)

```
def activate(self):
    self.z = self.b
    for neuron,connection in self.incoming:
        if neuron.a == None: neuron.activate()
        self.z += neuron.a * connection.w
        self.a = g(self.z)


for neuron in outputs: neuron.activate()
```

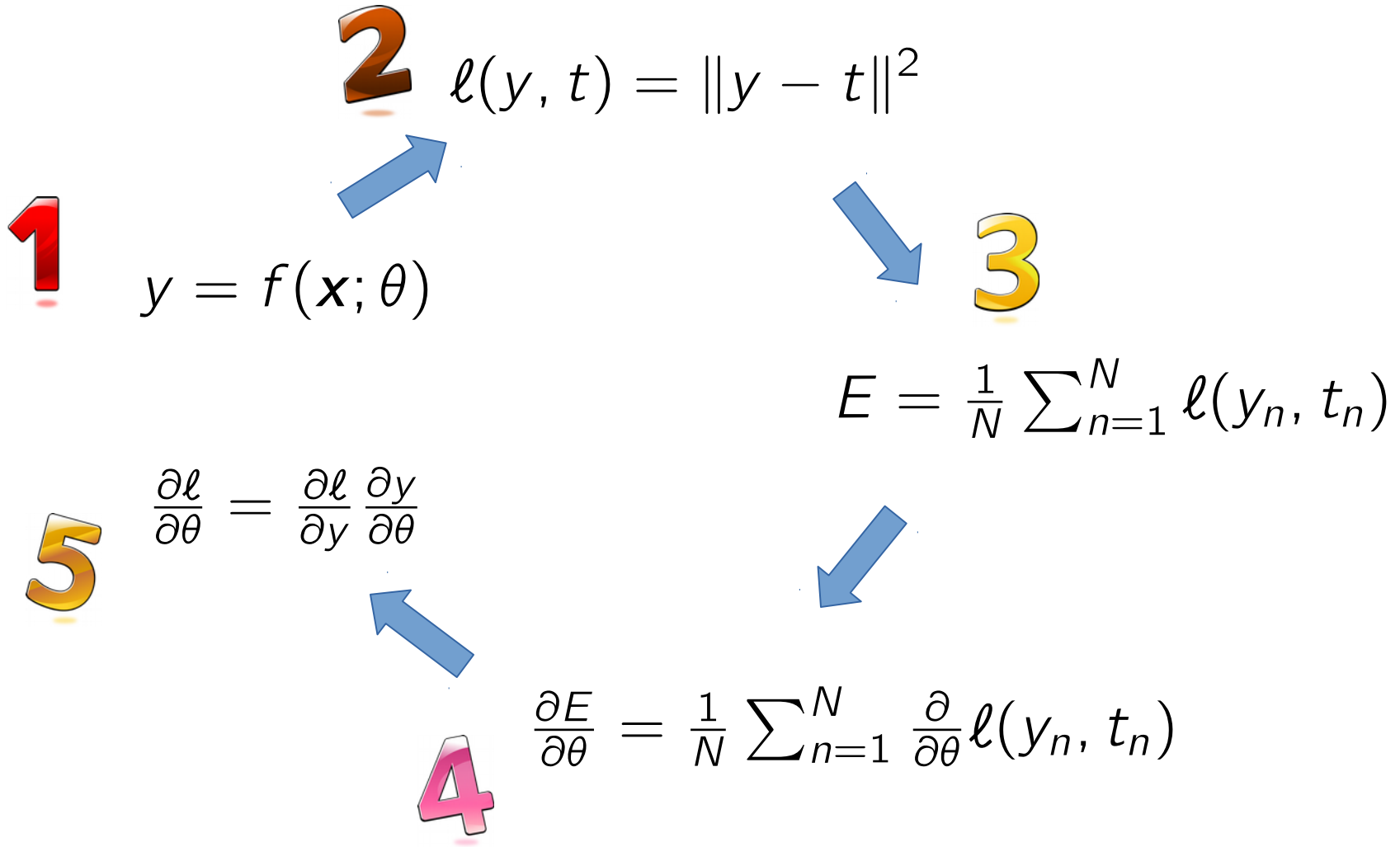
Recap: How to Learn in a Neural Network

Observation: A neural network is a function of both its inputs and parameters.



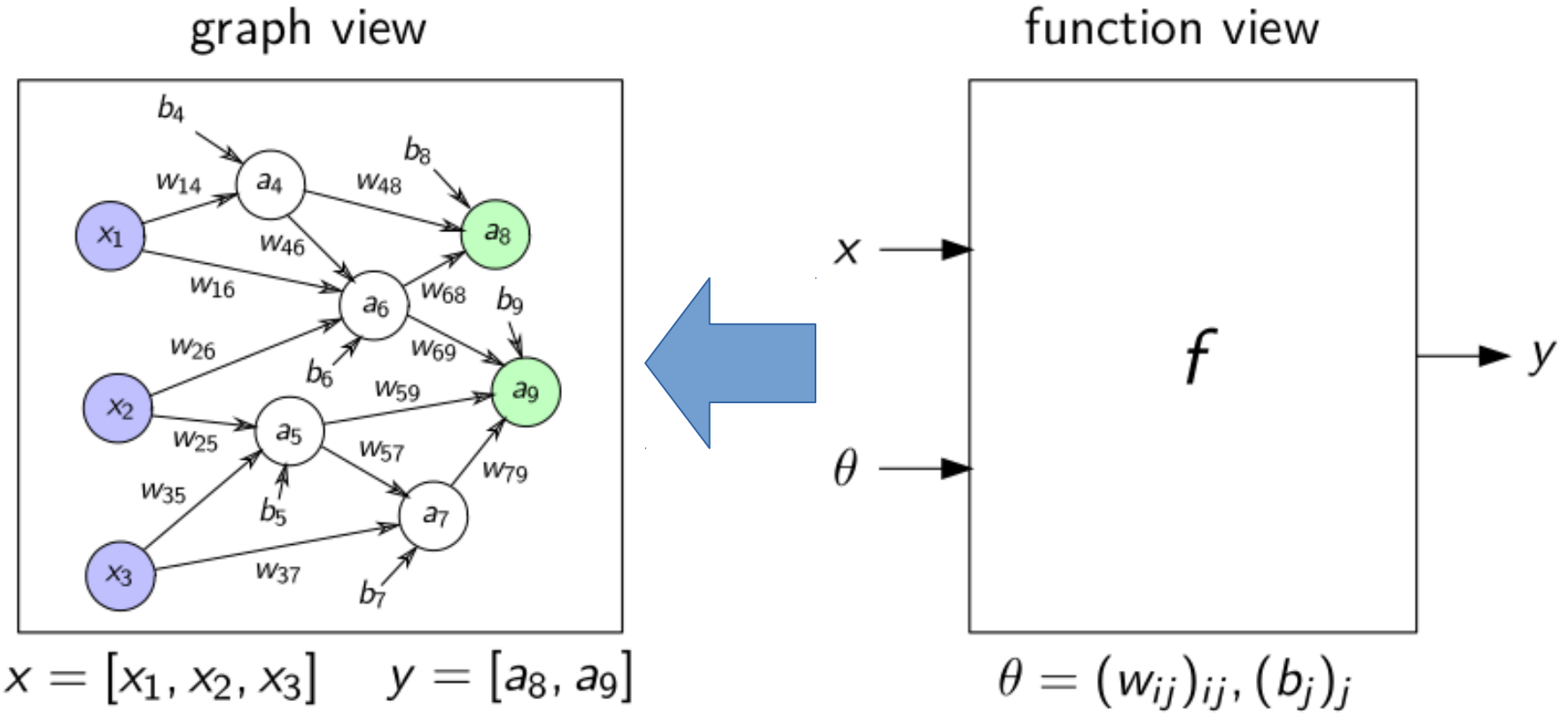
Learn by evaluating some error function and following the gradients.

How to Learn in a Neural Network

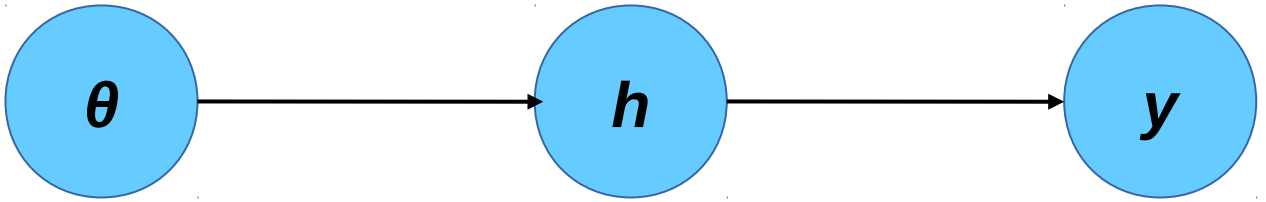


How to Learn in a Neural Network

Question: How to propagate gradients in the neural network graph



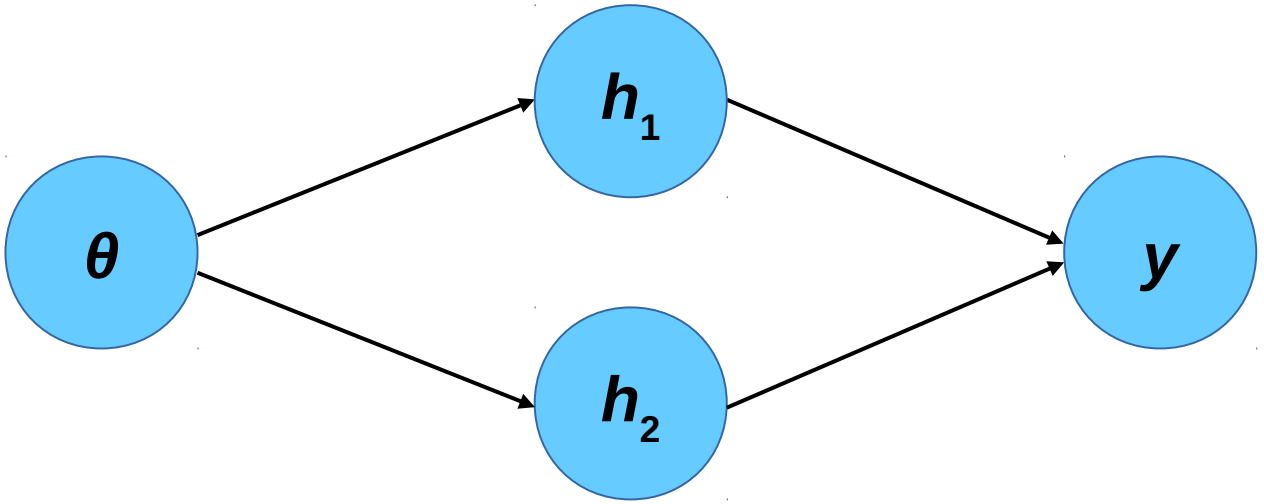
Chain Rule



forward: $y(h(\theta))$

backward: $\frac{\partial y}{\partial \theta} = \frac{\partial h}{\partial \theta} \cdot \frac{\partial y}{\partial h}$

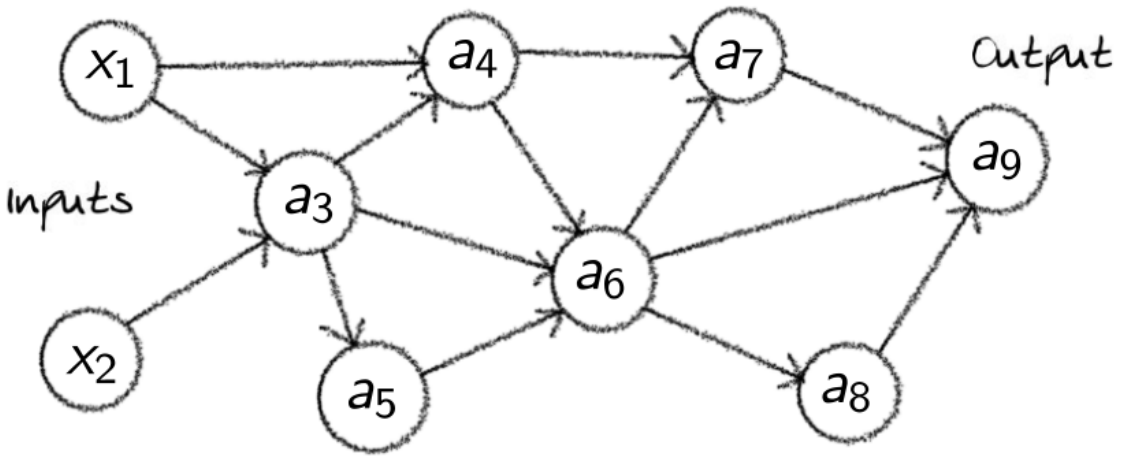
Multivariate Chain Rule



forward: $y(h_1(\theta), h_2(\theta))$

backward: $$\frac{\partial y}{\partial \theta} = \frac{\partial h_1}{\partial \theta} \cdot \frac{\partial y}{\partial h_1} + \frac{\partial h_2}{\partial \theta} \cdot \frac{\partial y}{\partial h_2}$$

Backward Propagation (1)



$$f(x) = a_9 = g(z_9)$$

Shortcut notation:

$$\delta_i = \partial E / \partial z_i$$

Error function:

$$E = (f(x) - t)^2$$

Error gradients w.r.t. neurons:

$$\delta_9 = 2 \cdot (f(x) - t) \cdot g'(z_9)$$

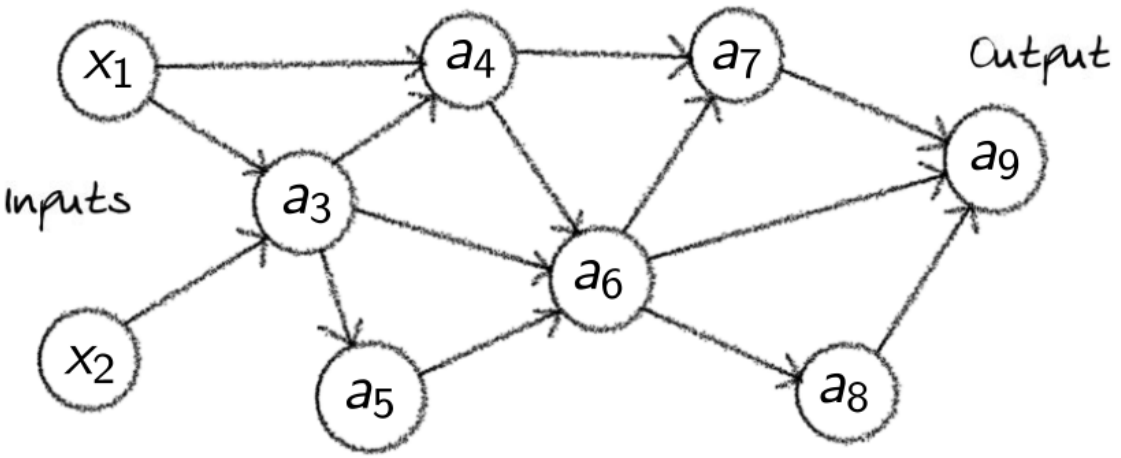
$$\delta_8 = \delta_9 \cdot \partial z_9 / \partial z_8 = \delta_9 \cdot w_{89} \cdot g'(z_8)$$

$$\delta_7 = \delta_9 \cdot \partial z_9 / \partial z_7 = \delta_9 \cdot w_{79} \cdot g'(z_7)$$

$$\delta_6 = \delta_9 \cdot \partial z_9 / \partial z_6 + \delta_8 \cdot \partial z_8 / \partial z_6 + \delta_7 \cdot \partial z_7 / \partial z_6$$

$$= [\delta_9 \cdot w_{69} + \delta_8 \cdot w_{68} + \delta_7 \cdot w_{67}] \cdot g'(z_6)$$

Backward Propagation (2)



$$f(x) = a_9 = g(z_9)$$

Shortcut notation:

$$\delta_i = \partial E / \partial z_i$$

Recap from previous slide (derivative of neuron 6):

$$\delta_6 = [\delta_9 \cdot w_{69} + \delta_8 \cdot w_{68} + \delta_7 \cdot w_{67}] \cdot g'(z_6)$$

Error derivatives for parameters of neuron 6:

$$\partial \mathcal{E} / \partial b_6 = \delta_6 \cdot \partial z_6 / \partial b_6 = \delta_6 \cdot 1$$

$$\partial \mathcal{E} / \partial w_{56} = \delta_6 \cdot \partial z_6 / \partial w_{56} = \delta_6 \cdot a_5$$

$$\partial \mathcal{E} / \partial w_{46} = \delta_6 \cdot \partial z_6 / \partial w_{46} = \delta_6 \cdot a_4$$

$$\partial \mathcal{E} / \partial w_{36} = \delta_6 \cdot \partial z_6 / \partial w_{36} = \delta_6 \cdot a_3$$

Neural Networks for Classification

So far, we have optimized the model minimize mean-square error. This is a reasonable choice for regression problem but what about classification?

$$y \in \mathbb{R} \qquad\qquad t \in \{-1, 1\}$$

$$\ell(y, t) = 1_{\mathrm{sgn}(y) \neq t} = 1_{y \cdot t \leq 0}$$

$$\frac{\partial \ell}{\partial \theta} = ?$$

Neural Networks for Classification

True objective is not differentiable

$$\ell(y, t) = 1_{y \cdot t \leq 0}$$

Idea: use a surrogate loss e.g. negative log-likelihood

$$\ell(y, t) = -\log y \cdot 1_{t>0} - \log(1 - y) \cdot 1_{t<0}$$

where $y = p_\theta(t = 1|\boldsymbol{x}) \in [0, 1]$ should be interpreted as the probability of the first class.

Remaining question:
how to ensure that *y* describes a probability?

Neural Networks for Classification

$$y = p_\theta(t = 1|\boldsymbol{x}) \in [0, 1]$$

Remaining question:
how to ensure that *y* describes a probability score?

Use a top-layer nonlinearity that maps real values to probabilities, e.g. the sigmoid function:

$$y = \frac{\exp(z)}{1 + \exp(z)}$$

or the softmax function for multiclass problems:

$$y_k = \frac{\exp(z_k)}{\sum_{k'} \exp(z_{k'})}$$

Other Loss Functions

Epsilon-sensitive loss

Enforce regression accuracy with some epsilon-value tolerance.

$$\ell = \left(y - t - \max\left(\min\left(y - t, \varepsilon\right), -\varepsilon\right)\right)^2$$

Log-likelihood loss

Useful when predicting the parameters of a probability model of which targets are samples. (when predicting the mean of a Gaussian probability model, the loss reduces to MSE, when predicting the parameters of a heavy tailed distribution, the training becomes more robust to outliers)

Structured losses (for trees, graphs)

Summary

(Deep) neural networks can represent any function given sufficiently many neurons (*universality*).

On most problems, only a limited number of neurons will be necessary (*compactness*)

Neural network error gradient can be computed using backpropagation (*chain rule*)

For this to work, the neural network and the loss function need to be differentiable.