Neural Networks

Sewoong Oh

CSE446
University of Washington

Recall Multi-class logistic regression

- data: categorical y in $\boxed{\{c_1, \ldots, c_k\}}$ with k categories
- model: linear vector-function makes a linear prediction

$\underline{\underline{\hat{y} \in \mathbb{R}^k}}$

$\{x_i, y_i = c_1 = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}\}$

$$\hat{y}_i = f(x_i) = w^T x_i \in \mathbb{R}^k$$

$c_2 = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$

with model parameter matrix $w \in \mathbb{R}^{d \times k}$

$$f(x_i) = \begin{bmatrix} f_1(x_i) \\ f_2(x_i) \\ \vdots \\ f_k(x_i) \end{bmatrix} = \begin{bmatrix} w_{1,0} + w_{1,1}x[1] + w_{1,2}x[2] + \cdots \\ w_{2,0} + w_{2,1}x[1] + w_{2,2}x[2] + \cdots \\ \vdots \\ w_{k,0} + w_{k,1}x[1] + w_{k,2}x[2] + \cdots \end{bmatrix} = f(x_i)_j = \sum_{l=1}^{d} w_{jl} x_i[l]$$

$$w = \begin{bmatrix} w[:,1] & w[:,2] & \cdots & w[:,k] \end{bmatrix}$$

- Logistic regression

2 classes

$$\mathbb{P}(y_i = -1 \,|\, x_i) = \frac{1}{1 + e^{w^T x_i}}$$

$$\mathbb{P}(y_i = +1 \,|\, x_i) = \frac{1}{1 + e^{-w^T x_i}}$$

k classes

$$\mathbb{P}(y_i = c_1 \,|\, x_i) = \frac{e^{w[:,1]^T x_i}}{e^{w[:,1]^T x_i} + \cdots + e^{w[:,k]^T x_i}}$$

$$\vdots$$

$$\mathbb{P}(y_i = c_k \,|\, x_i) = \frac{e^{w[:,k]^T x_i}}{e^{w[:,1]^T x_i} + \cdots + e^{w[:,k]^T x_i}}$$

Maximum Likelihood Estimator

$$\text{maximize}_w \;\; \frac{1}{n} \sum_{i=1}^{n} \log(\mathbb{P}(y_i \,|\, x_i))$$

$$\text{maximize}_{w \in \mathbb{R}^d} \;\; \frac{1}{n} \sum_{i=1}^{n} \log\left(\frac{1}{1 + e^{-y_i w^T x_i}} \right)$$

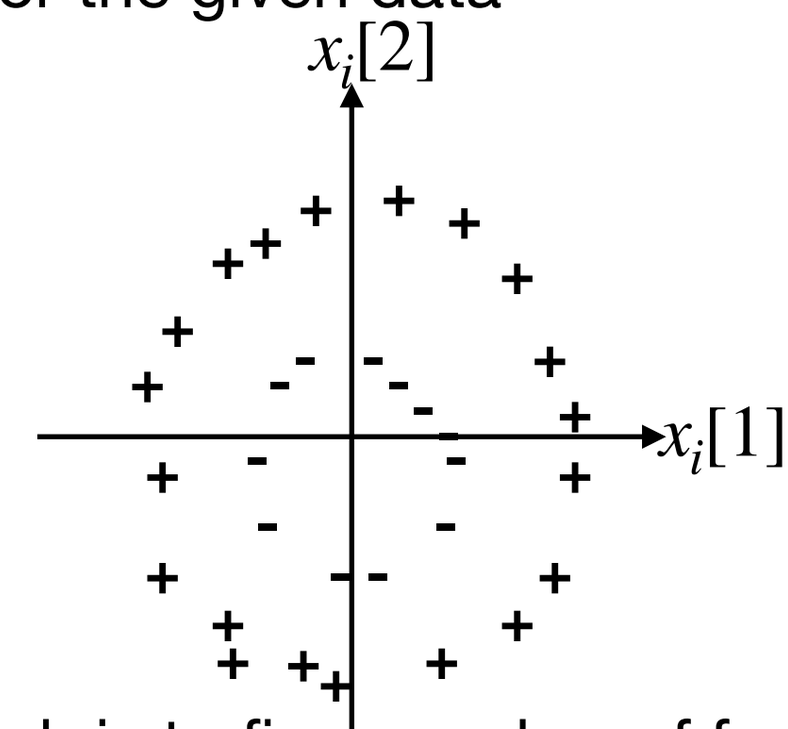
$$\text{maximize}_{w \in \mathbb{R}^{d \times k}} \sum_{i=1}^{n} \sum_{j=1}^{k} \mathbf{I}\{y_i = j\} \log\left(\frac{e^{w[:,k]^T x_i}}{\sum_{j'=1}^{k} e^{w[:,j']^T x_i}} \right)$$

Neural Network

- for classification and regression, we studied linear models

$$f_w(x) \;=\; w_0 + w_1 h_1(x) + w_2 h_2(x) + \cdots + w_k h_k(x)$$

- without domain knowledge, typical machine learning starts out with a large number k of features, and use regularization to select a small number of features that matter for the given data



- an alternative approach is to fix a number of features to be used in advance, and learn the features adapted to the data
- most successful approach in this direction is
  Feed-forward Neural network also called
  Multilayer Perceptron (MLP)
- the term neural originates from an attempt to make a connection to information processing in biological systems

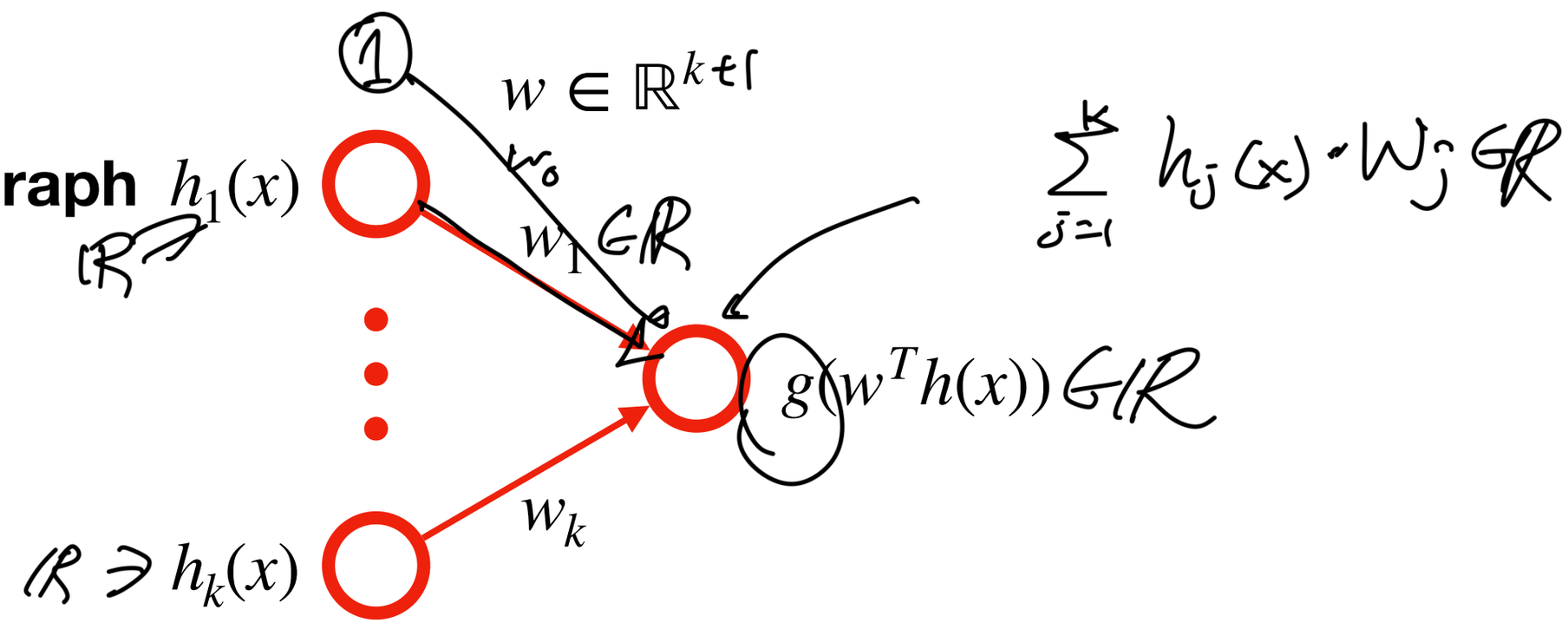
Feed-forward neural network

- Feed-forward neural network is a multi-layer generalization of **logistic regression**
- recall logistic regression predict the probability that the label is +1 by

$$\mathbf{P}(y = +1 \,|\, x) \;\approx\; f_w(x) \;=\; g(\underbrace{w_0 + w_1 h_1(x) + w_2 h_2(x) + \cdots + w_k h_k(x)}_{w^T h(x)})$$

where the sigmoid function is used: $g(a) = \dfrac{1}{1 + e^{-a}}$

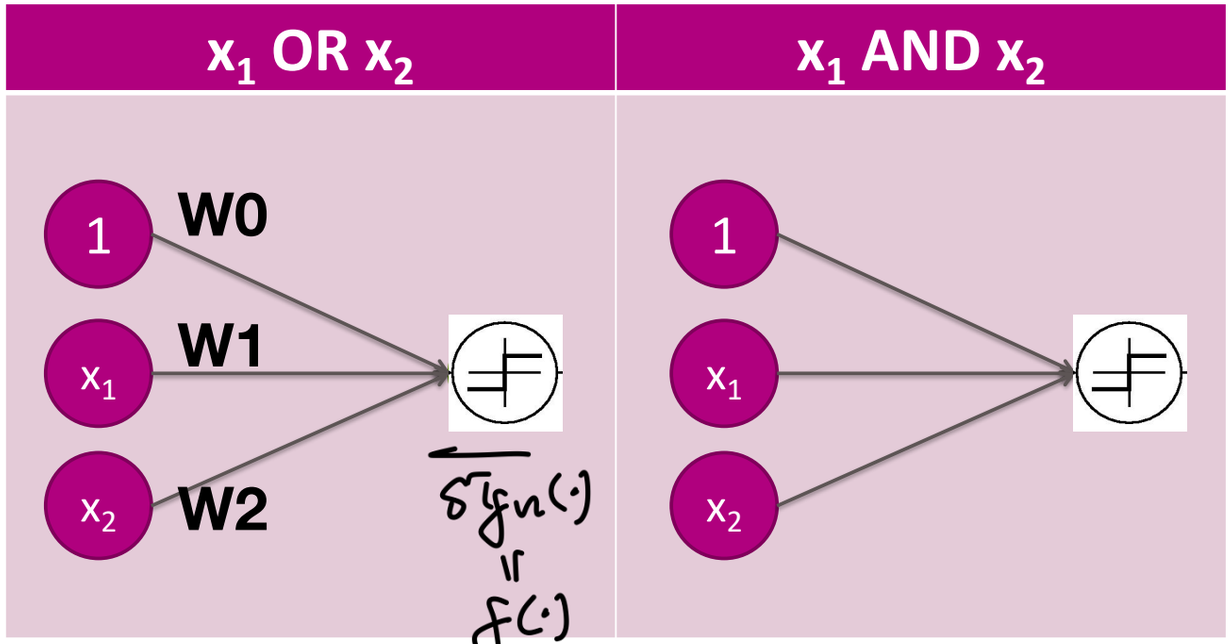
- **Computational graph**



- instead of using predefined features $h_j(x)$'s, we will replace them by parametric functions and **learn the features from data**
- the idea is to recursively apply (a version of) logistic regression in multiple layers

What can be represented by a linear classifier?

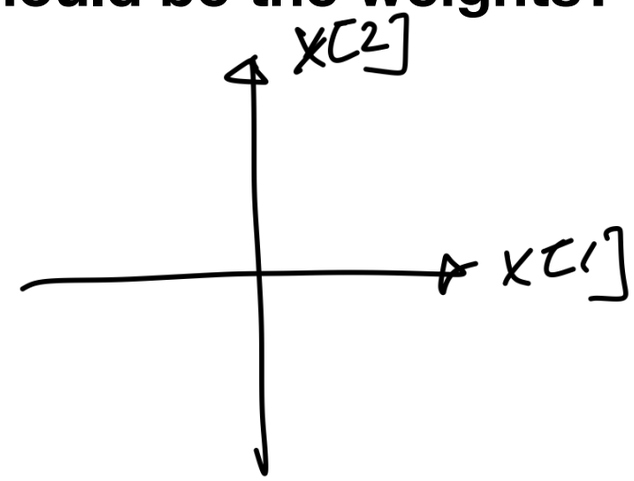
- x[1] x[2] y
- 0 0 0
- 0 1 1
- 1 0 1
- 1 1 1



- x[1] x[2] y
- 0 0 0
- 0 1 0
- 1 0 0
- 1 1 1

$f_w(x[1], x[2]) = (w_0 + w_1 x[1] + w_2 x[2]$

What should be the weights?



Note that there is a one-to-one correspondence between a linear classifier and a neural network of the above form

What cannot be learned?

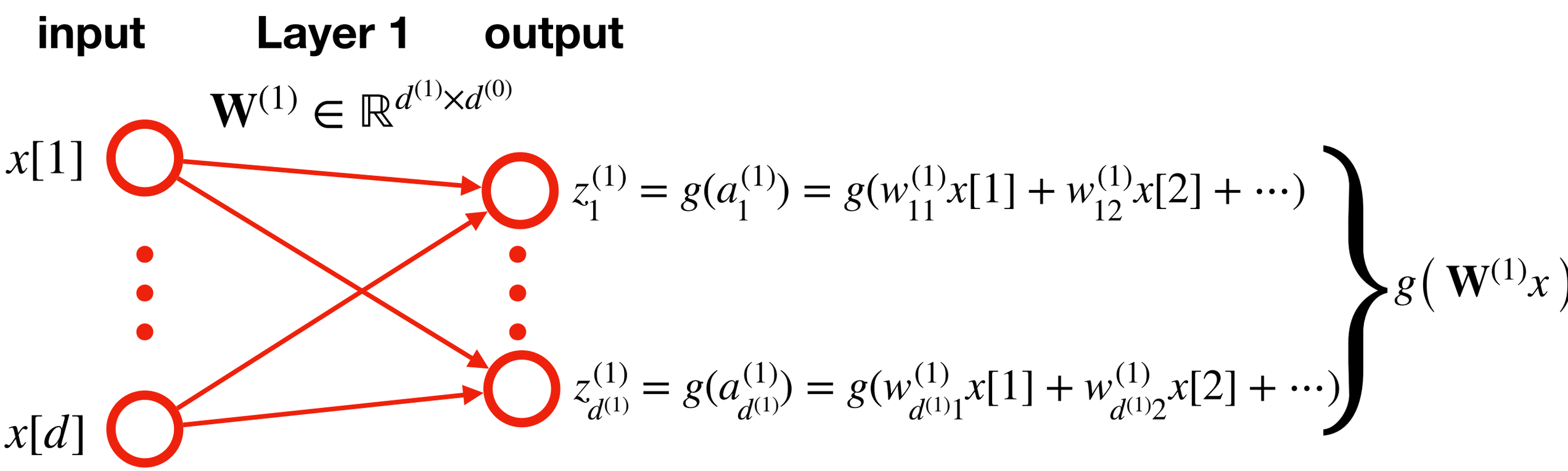
- **the first layer** (i.e. $\ell = 1$) has $d^{(0)} = d$ input dimension, and the input data is x
- and $d^{(1)}$ **nodes** or **units,**
  each node first computes **input activations** $a_k^{(1)}$:

$$a_k^{(1)} = \sum_{j=1}^{d^{(0)}} w_{kj}^{(1)} x_j \in \mathbb{R}, \qquad \text{for } k \in \{1, \ldots, d^{(1)}\}$$

  such that $a^{(1)} = \mathbf{W}^{(1)} x \in \mathbb{R}^{d^{(1)}}$ with **weight** $\mathbf{W}^{(1)} \in \mathbb{R}^{d^{(1)} \times d^{(0)}}$
  where we ignored the constant term w_{k0}'s (called bias) for notational convenience
- then outputs **output activation** $z_k^{(1)}$**:**

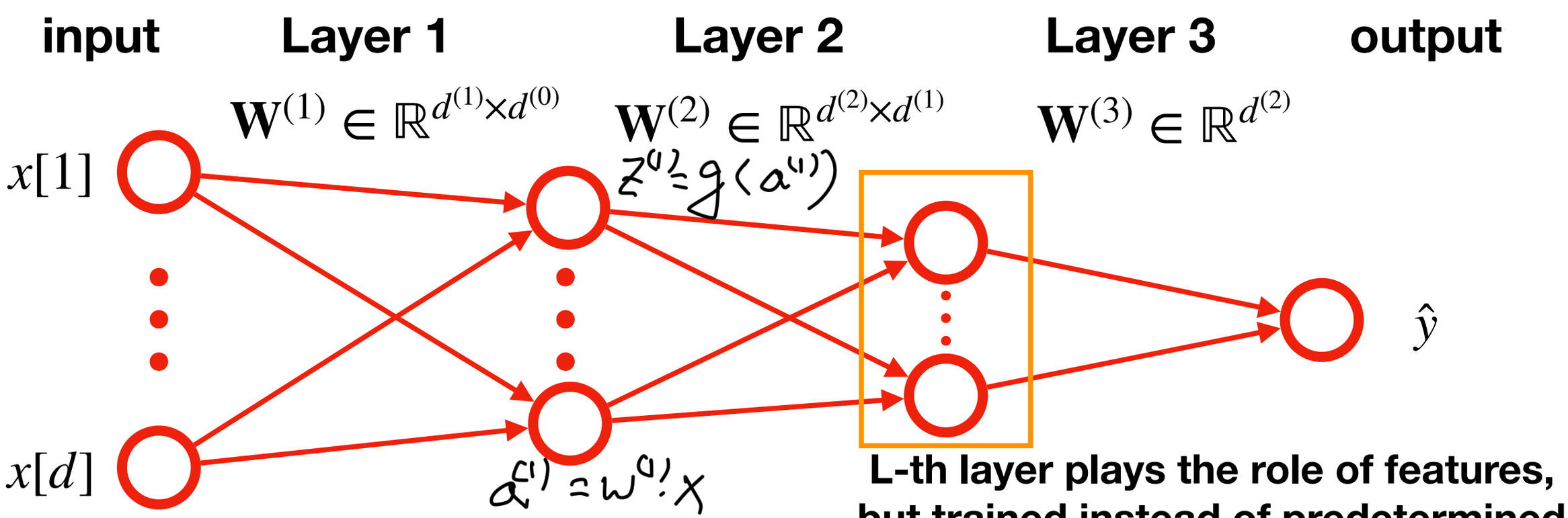
$$z_k^{(1)} = g\left(a_k^{(1)} \right) \qquad \text{for } k \in \{1, \ldots, d^{(1)}\}$$

  where $g(\cdot)$ can be a function of choice



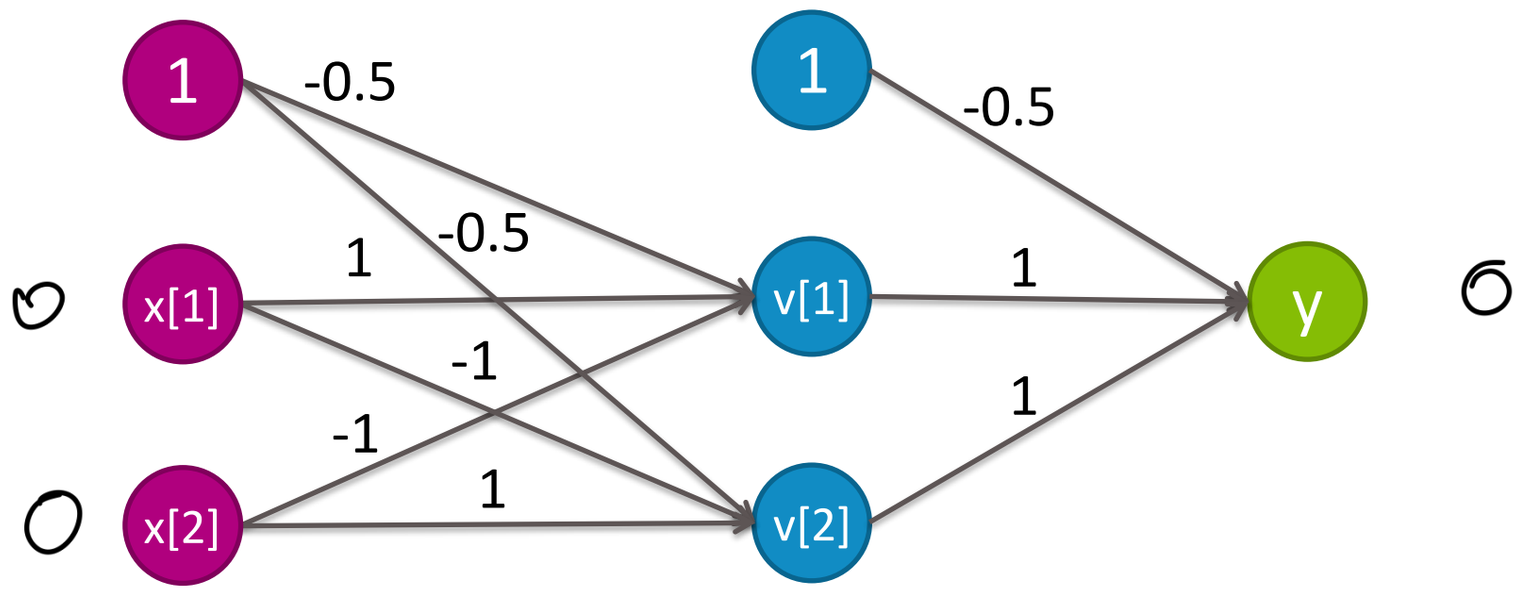
Feed-forward neural network

- using the convention that $z^{(0)} = x$, each layer computes
$$z^{(\ell)} \;=\; g\big(\,\mathbf{W}^{(\ell)} z^{(\ell-1)}\,\big),$$
where $g(\,\cdot\,)$ is entry-wise applied to a vector
- after L-**hidden layers**, the output is the input activation at level $L+1$:
$$\hat{y} \;=\; f_{\mathbf{W}^{(1)}\ldots,\mathbf{W}^{(L+1)}}(x) \;=\; a^{(L+1)} \;=\; \sum_{j=1}^{d^{(L)}} w_j^{(L+1)} z_j^{(L)}$$
- if there are more than 1 output (for example in the multi class classification problem), we compute a vector activation inputs of a dimension that we want



L-th layer plays the role of features, but trained instead of predetermined

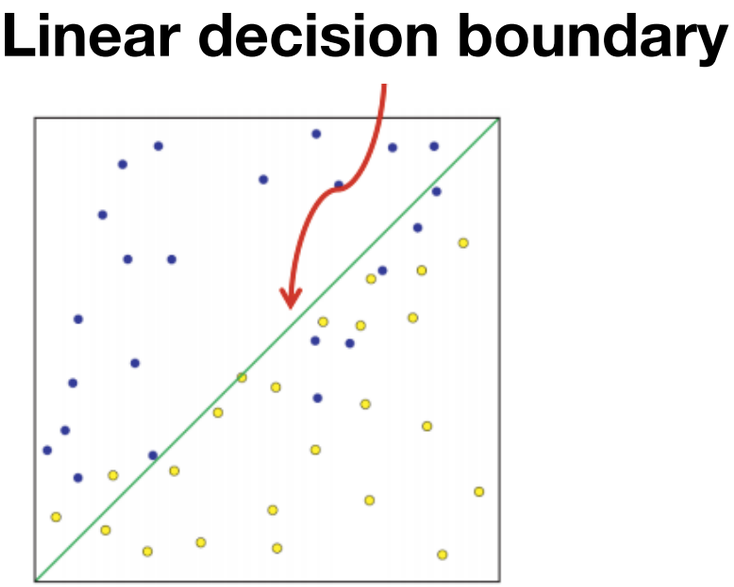
XOR as a 2-layer neural network



this is a special case with $g(a) = a$

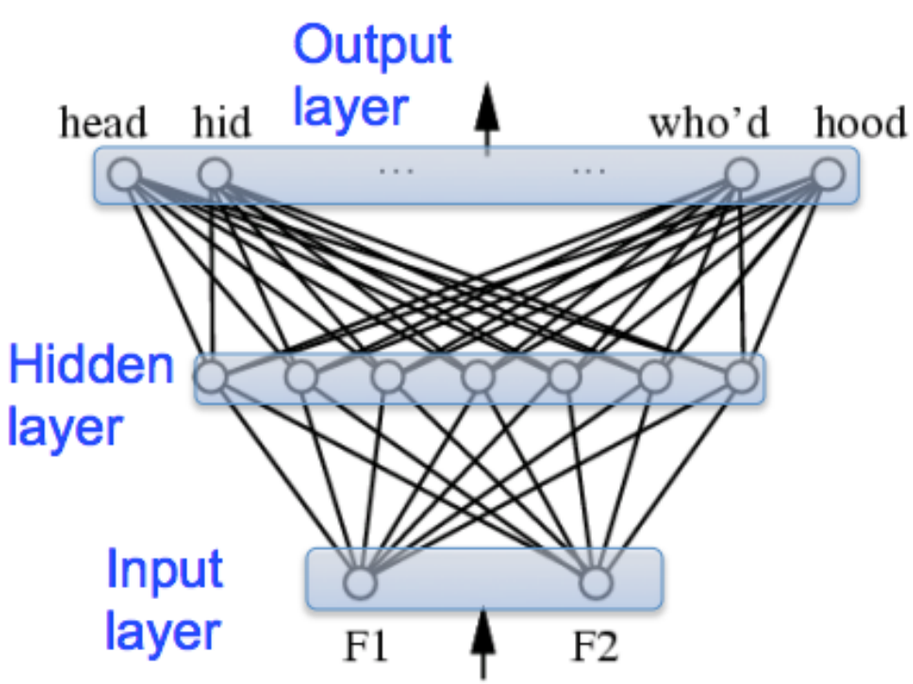
Example of 2-layer neural network in action

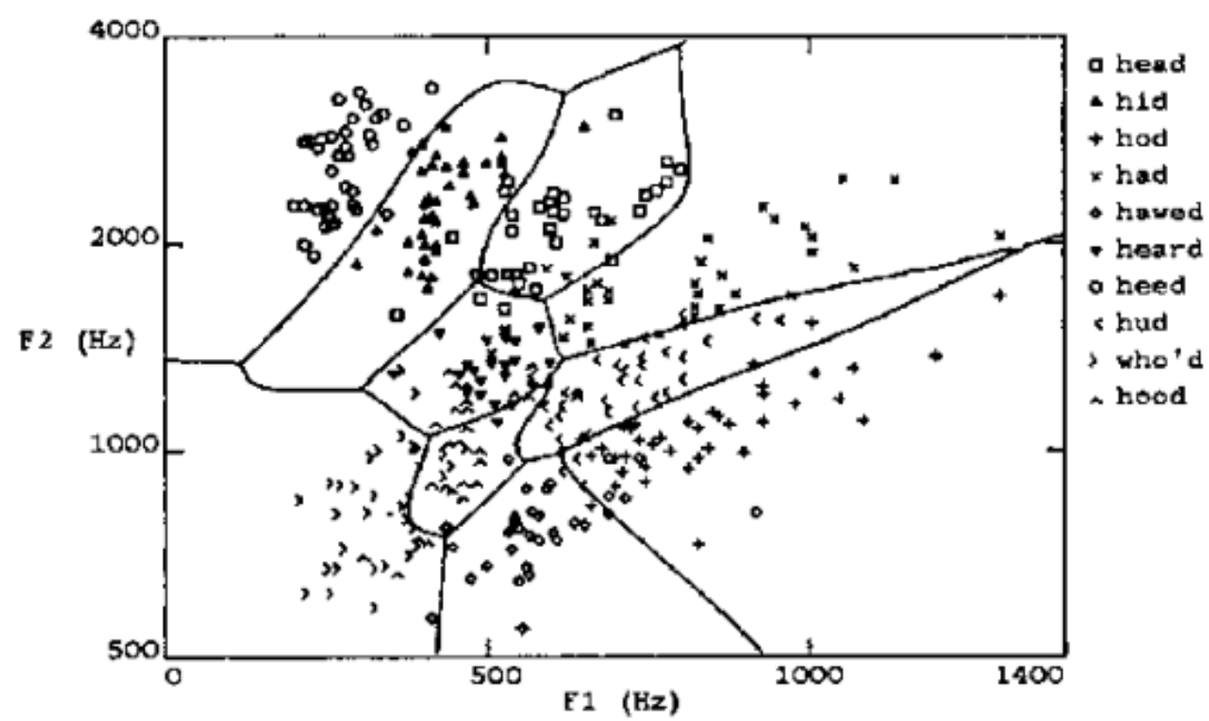
1-layer neural networks
only represents linear classifiers



Example: 2-layer neural network trained to distinguish vowel sounds using 2 formants (features)

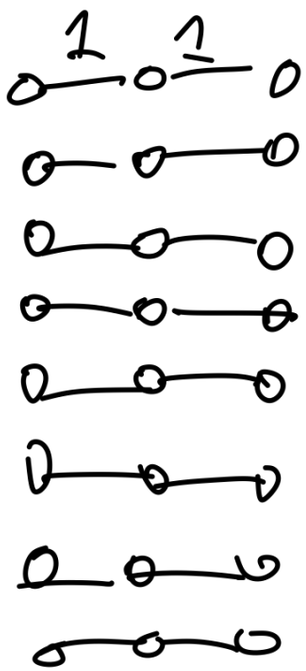
a highly non-linear decision boundary can be learned from 2-layer neural networks



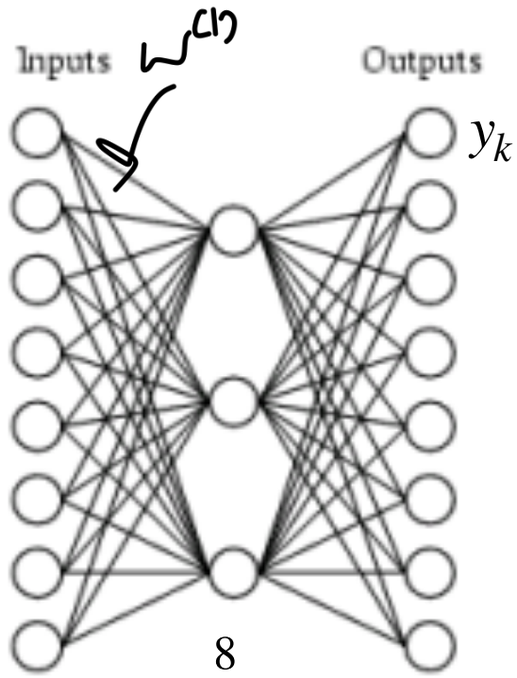


Representation power of a 2-layer neural network

A target function: Identity



Input x	z	Output y
10000000	001 →	10000000
01000000	111 →	01000000
00100000	101 →	00100000
00010000	000 →	00010000
00001000	100 →	00001000
00000100	→	00000100
00000010	→	00000010
00000001	→	00000001

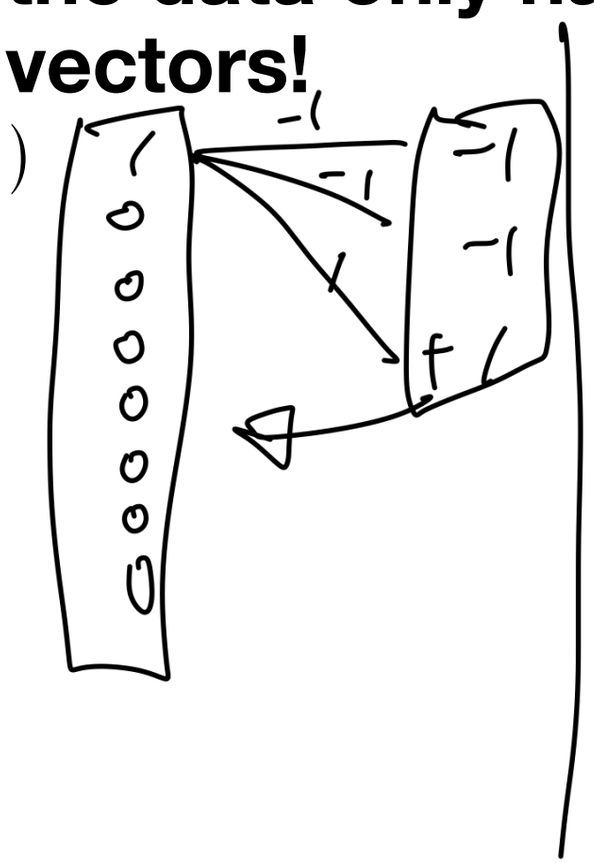


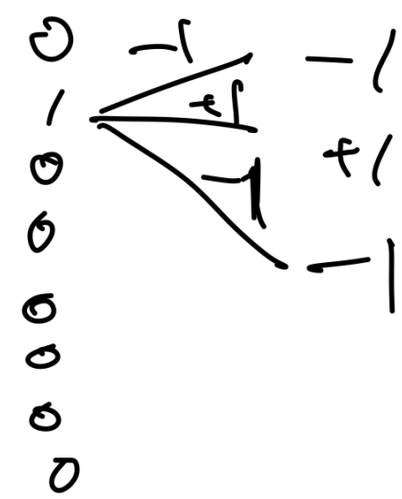
$$y_k = \text{sign}\Big(\sum_{j=1}^{3} w_{kj}^{(2)} z_j - 0.5 \Big)$$

$x \in \{0,1\}^8$

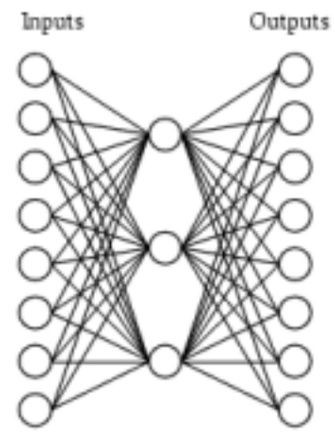
$$z_k = \sum_{j=1}^{8} w_{kj}^{(1)} x_j$$

- can such function be learned?
- if we are manually designing functions, then 3 hidden (binary values) nodes are enough.
- the reason is that there is some simplicity or pattern in the data that we want to represent: **although it is 8-dimensional, the data only has basis vectors!**





A network:



Learned hidden layer representation:

Input		Hidden Values				Output
10000000	→	.89	.04	.08	→	10000000
01000000	→	.01	.11	.88	→	01000000
00100000	→	.01	.97	.27	→	00100000
00010000	→	.99	.97	.71	→	00010000
00001000	→	.03	.05	.02	→	00001000
00000100	→	.22	.99	.99	→	00000100
00000010	→	.80	.01	.98	→	00000010
00000001	→	.60	.94	.01	→	00000001

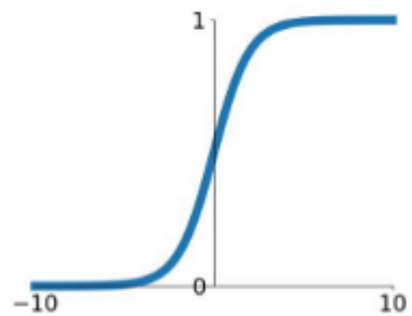
Nonlinear activation function

$g(\cdot): \mathbb{R} \to \mathbb{R}$

- popular choices of activation function includes

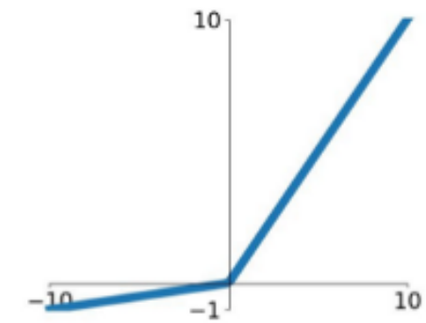
Sigmoid

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



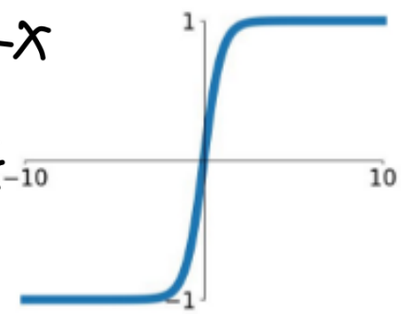
Leaky ReLU

$\max(0.1x, x)$



tanh

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

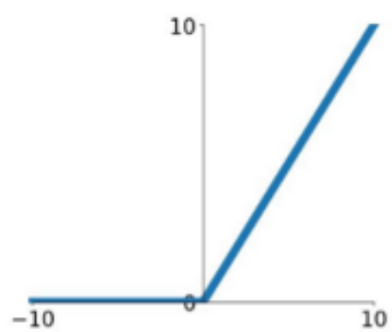


Maxout

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

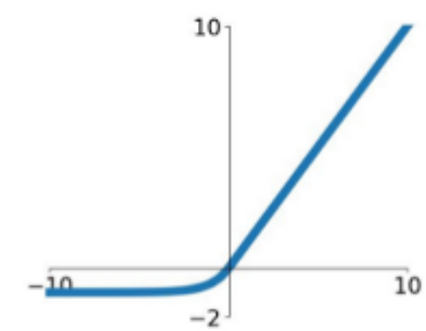
ReLU

$\max(0, x)$



ELU

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



$f_W(x) = \underbrace{W^{(L)} \cdots W^{(2)} W^{(1)}} \cdot x$

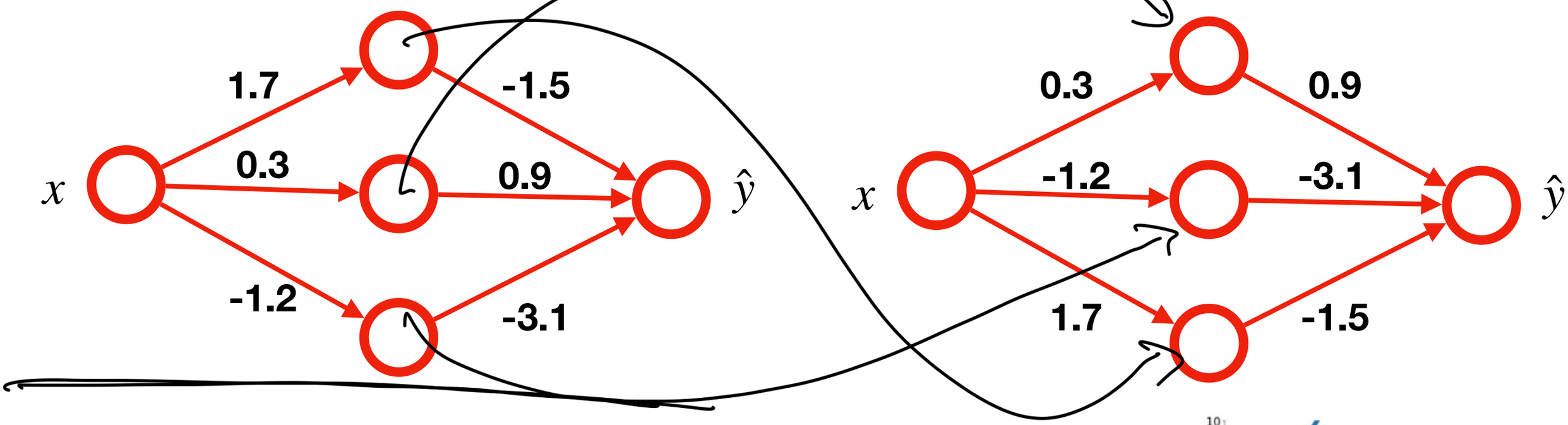
$= W \cdot x$

$\cdots g(W^{(2)} g(W^{(1)} x))$

Symmetry in the weights

$f_W(x)$:

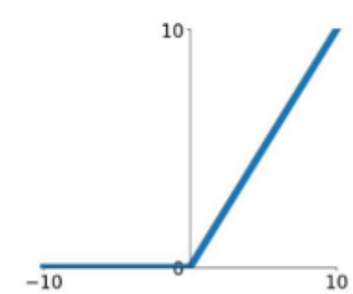
- whichever non-linear activation function is used,
  the following symmetry gives equivalent weights with identical outputs

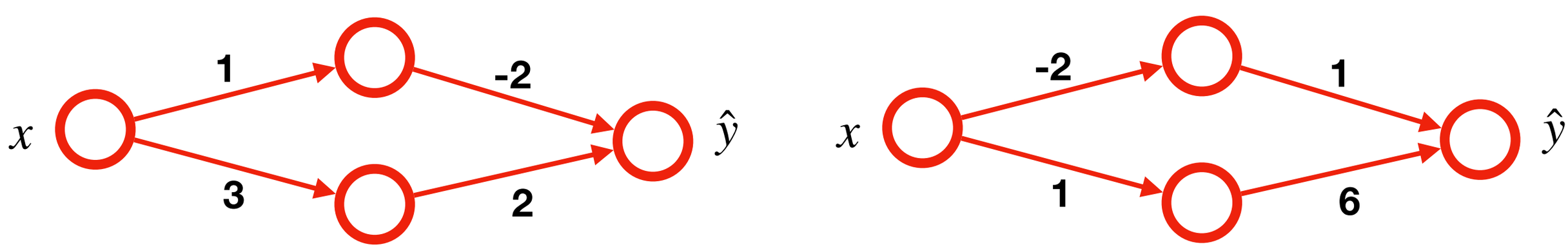


- if ReLU activation is used,
  then $g(c\,x) = c\,g(x)$ and

ReLU

$\max(0, x)$





Training

- let $\mathbf{W} = (\mathbf{W}^{(1)}, \ldots, \mathbf{W}^{(L+1)})$ denote all the weights of the neural network
- the empirical risk is defined the same way as

$$\mathscr{L}(\mathbf{W}) \;=\; \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f_{\mathbf{W}}(x_i))$$

- however, even for squared loss or logistic loss, the objective is no longer a convex function
- still, we apply (stochastic) gradient descent
- **back-propagation** algorithm efficiently computes the gradient using the computation graph
- we will focus on the example of squared loss

$$\mathscr{L}(\mathbf{W}) \;=\; \frac{1}{n}\sum_{i=1}^{n} (y_i - f_{\mathbf{W}}(x_i))^2$$

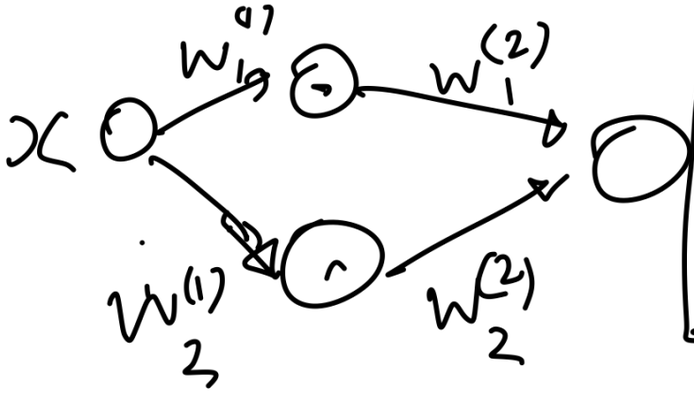
$$\boxed{\nabla_{W^{(\ell)}_{kj}} f_W(x_i)}$$

$$\Sigma \|$$

$$f_W(x_i)$$

$$\boxed{d^{(6)} \times d^{(5)} + d^{(5)} \cdot d^{(4)} + \cdots}$$

but back-propagation works for any loss



$$\boxed{\hat{y} = \begin{bmatrix} w_1^{(2)} & w_2^{(2)} \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}}$$

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

$$\frac{\partial \ell(y, \hat{y})}{\partial w_2^{(1)}} = ?$$

Forward pass

$$\begin{bmatrix} a_1 \\ a_2 \end{bmatrix} = \begin{bmatrix} w_1^{(1)} \cdot x \\ w_2^{(1)} \cdot x \end{bmatrix}$$

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} g(a_1) \\ g(a_2) \end{bmatrix}$$

$$\boxed{\hat{y} = w_1^{(2)} g(a_1) + w_2^{(2)} g(a_2)}$$

$$\hat{y} = w_1^{(2)} z_1 + w_2^{(2)} z_2$$

Backward Pass

$$\delta^{(2)} = \frac{\ell(y, \hat{y})}{\partial \hat{y}} = 2(\hat{y} - y) \longrightarrow \frac{\partial \ell(y, \hat{y})}{\partial w_1^{(2)}} = \frac{\partial \ell}{\partial \hat{y}} \boxed{\frac{\partial \hat{y}}{\partial w_1^{(2)}}}$$

data

F.P.

$$\nabla_{w^{(2)}} \ell(y, \hat{y}) = \begin{bmatrix} 2(\hat{y} - y) \begin{bmatrix} z_1 \\ z_2 \end{bmatrix} \end{bmatrix}$$

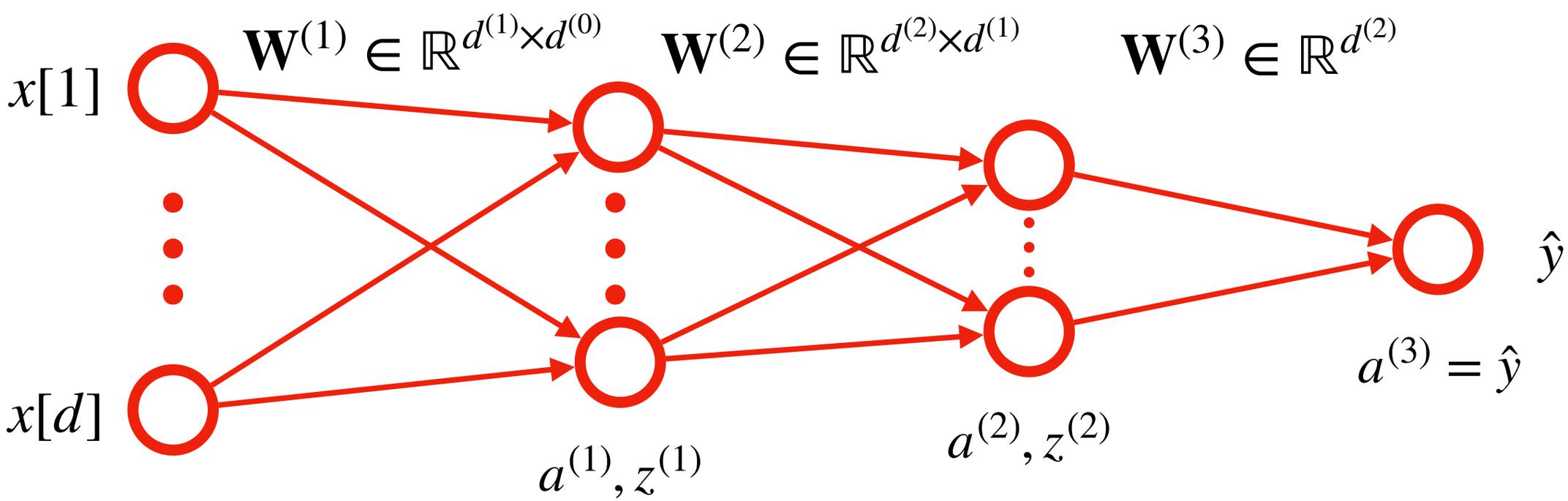
$$\delta^{(1)} = \begin{bmatrix} \frac{\partial \ell(y, \hat{y})}{\partial a_1} \\ \frac{\partial \ell(y, \hat{y})}{\partial a_2} \end{bmatrix} = \frac{\partial \ell(y, \hat{y})}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a_1} = 2(\hat{y} - y) \begin{bmatrix} w_1^{(2)} g'(a_1) \\ w_2^{(2)} \cdot g'(a_2) \end{bmatrix}$$

Back-propagation for computing gradient

- for a given model $\mathbf{W}^{(1)}, \mathbf{W}^{(2)}, \ldots, \mathbf{W}^{(L+1)}$**,** we compute the gradient exactly as follows
- we explain how to do it for a single input case, i.e.

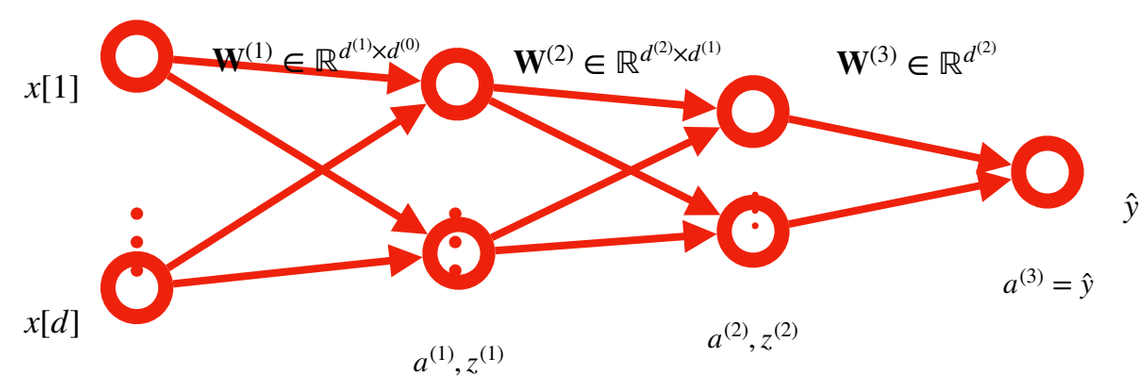
$$\ell(y, \hat{y}) = (y - f_{\mathbf{W}}(x))^2$$

  we can easily generalize it when there are n data points in the training data
- **Forward pass**
  - starting from a single input x, go forward (from input to output layer), compute and store the variables $a^{(1)}, z^{(1)}, a^{(2)}, z^{(2)}, \ldots, a^{(L)}, z^{(L)}, a^{(L+1)}$



Back-propagation for computing gradient

- **Backward pass**
  - we want to compute $\nabla_{w_{kj}^{(l)}} \ell(y, \underbrace{f_{\mathbf{W}}(x)}_{\hat{y}})$ for all k, j, l
  - instead of writing the function explicitly, and writing the gradient explicitly, we will use recursion
  - we will do it backwards from output to input
  - define $\delta_j^{(l)} \triangleq \dfrac{\partial \ell(y, \hat{y})}{\partial a_j^{(l)}}$
  - if we have all $\delta_j^{(l)}$'s then we can compute all derivatives w.r.t $w_{kj}^{(l)}$'s:



(a) $$\frac{\partial \ell(y, \hat{y})}{\partial w_{kj}^{(l)}} = \frac{\partial \ell(y, \hat{y})}{\partial a_k^{(l)}} \frac{\partial a_k^{(l)}}{\partial w_{kj}^{(l)}} = \delta_k^{(l)} z_j^{(l-1)}$$

which follows from

$$a_k^{(l)} = \sum_j w_{kj}^{(l)} z_j^{(l-1)} \text{ and } \frac{\partial a_k^{(l)}}{\partial w_{kj}^{(l)}} = z_j^{(l-1)}$$

Back-propagation for computing gradient

- we can now recursively compute all $\delta_j^{(l)}$'s and hence all derivatives $\nabla_{w_{kj}^{(l)}} \ell(y, \hat{y})$'s
- starting from the output layer where $\hat{y} = a^{(L+1)}$

$$\boxed{\delta^{(L+1)}} \triangleq \frac{\partial \ell(y, \hat{y})}{\partial a^{(L+1)}} = 2(a^{L+1)} - y) = 2(\hat{y} - y)$$

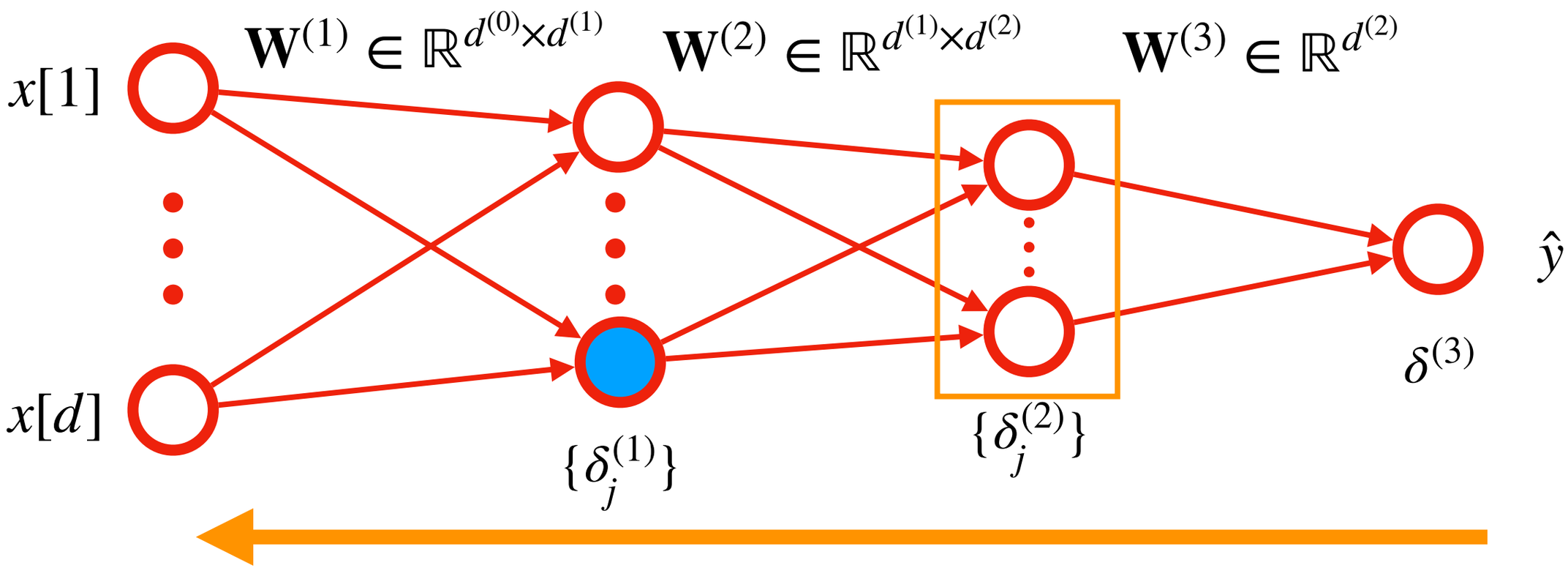
as $\ell(y, \hat{y}) = (y - \hat{y})^2$, and there is no subscript k as $\delta^{(L+1)}$ is a scalar

- we apply (a) to get the output layer derivatives

$$\boxed{\frac{\partial \ell(y, \hat{y})}{\partial w_j^{(L+1)}}} = \delta^{(L+1)} z_j^{(L)} = 2(\hat{y} - y)\, z_j^{(L)}$$

Precomputed from froward pass

- now we can recursively compute $\delta_j^{(l)}$'s using $\delta_k^{(l+1)}$'s



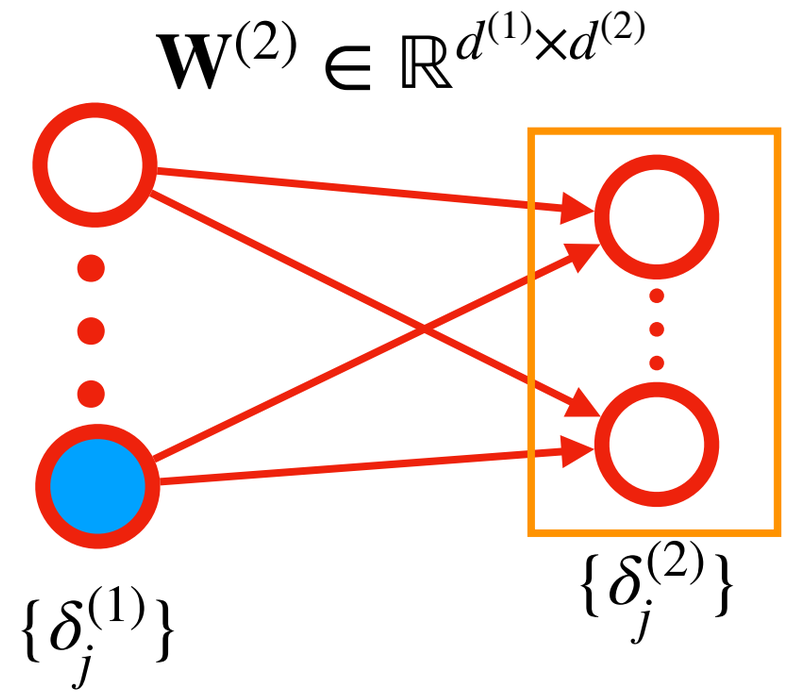
- It follows from the computation graph that $\delta_j^{(1)}$ depends on the loss $\ell(y, \hat{y})$ only through $\delta_j^{(l+1)}$'s

- using the chain rule,

$$\delta_j^{(l)} \triangleq \frac{\partial \ell(y, \hat{y})}{\partial a_j^{(l)}}$$

$$= \sum_{k=1}^{d^{(l+1)}} \frac{\partial \ell(y, \hat{y})}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}}$$

(b) $$= \sum_{k=1}^{d^{l+1)}} \delta_k^{(l+1)} \frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}}$$



- to finish the recursion, we need to compute the second term in the summand

$$a_k^{(l+1)} = \sum_{c=1}^{d^{(l)}} w_{kc}^{(l+1)} z_c^{(l)} = \sum_{c=1}^{d^{(l)}} w_{kc}^{(l+1)} g(a_c^{(l)})$$

which implies

$$\frac{\partial a_k^{(l+1)}}{\partial a_j^{(l)}} = w_{kj}^{(l+1)} g'(a_j^{(l)})$$

- substituting it back in (b), we get

$$\delta_j^{(l)} = g'(a_j^{(l)}) \sum_{k=1}^{d^{(l+1)}} w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

Back-propagation for computing $\nabla_{\mathbf{W}}(y - f_{\mathbf{W}}(x))^2$

- **Forward pass:**
  - compute $a^{(1)}, z^{(1)}, a^{(2)}, \cdots, a^{(L)}, z^{(L)}, a^{(L+1)} = \hat{y}$
- **Backward pass:**
  - initialize: $\delta^{(L+1)} \;=\; 2(\hat{y} - y)$

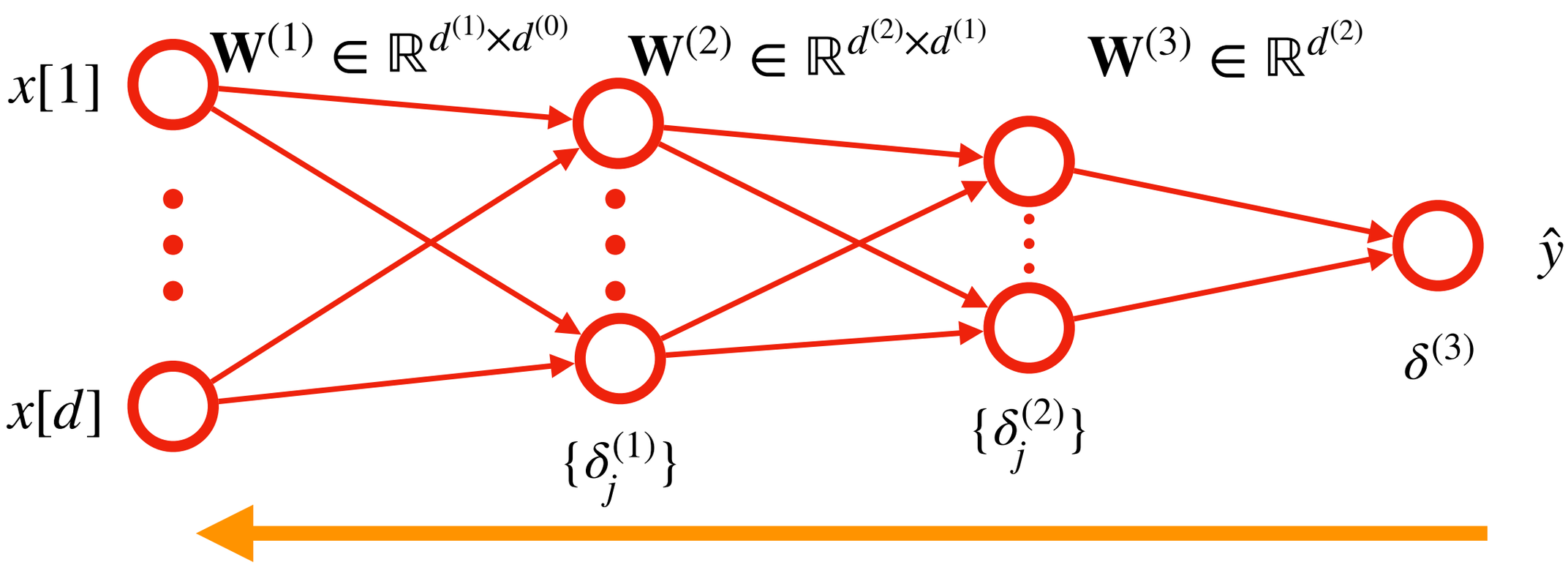
$$\frac{\partial \ell(y, \hat{y})}{\partial w_j^{(L+1)}} \;=\; 2(\hat{y} - y)\, z_j^{(L)}$$

  - recursively compute:

$$\delta_j^{(l)} \;=\; g'(a_j^{(l)}) \sum_{k=1}^{d^{(l+1)}} w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

  and the derivatives

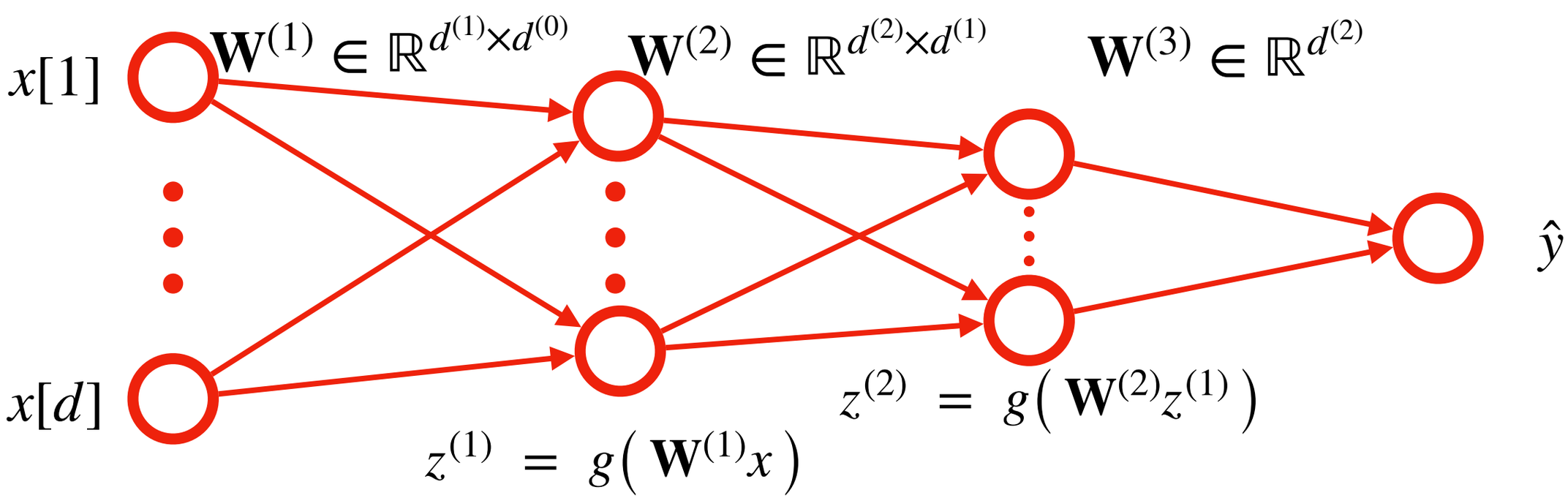
$$\frac{\partial \ell(y, \hat{y})}{\partial w_{kj}^{(l)}} \;=\; \delta_k^{(l)} z_j^{(l-1)}$$



Time complexity of evaluating the function value

- suppose addition, subtraction, multiplication, division, and evaluating $g(\,\cdot\,)$ of scalars take "one unit" of time
- the time complexity to compute $\hat{y} = f_{\mathbf{W}}(x)$ is

$$\underbrace{2 \times d^{(0)} \times d^{(1)}}_{a^{(1)}=\mathbf{W}^{(1)}x} + \underbrace{d^{(1)}}_{z^{(1)}=g(a^{(1)})} + 2 \times d^{(1)} \times d^{(2)} + d^{(2)} + \cdots + \underbrace{2 \times d^{(L)}}_{\hat{y}=\mathbf{W}^{(L+1)}z^{(L)}}$$

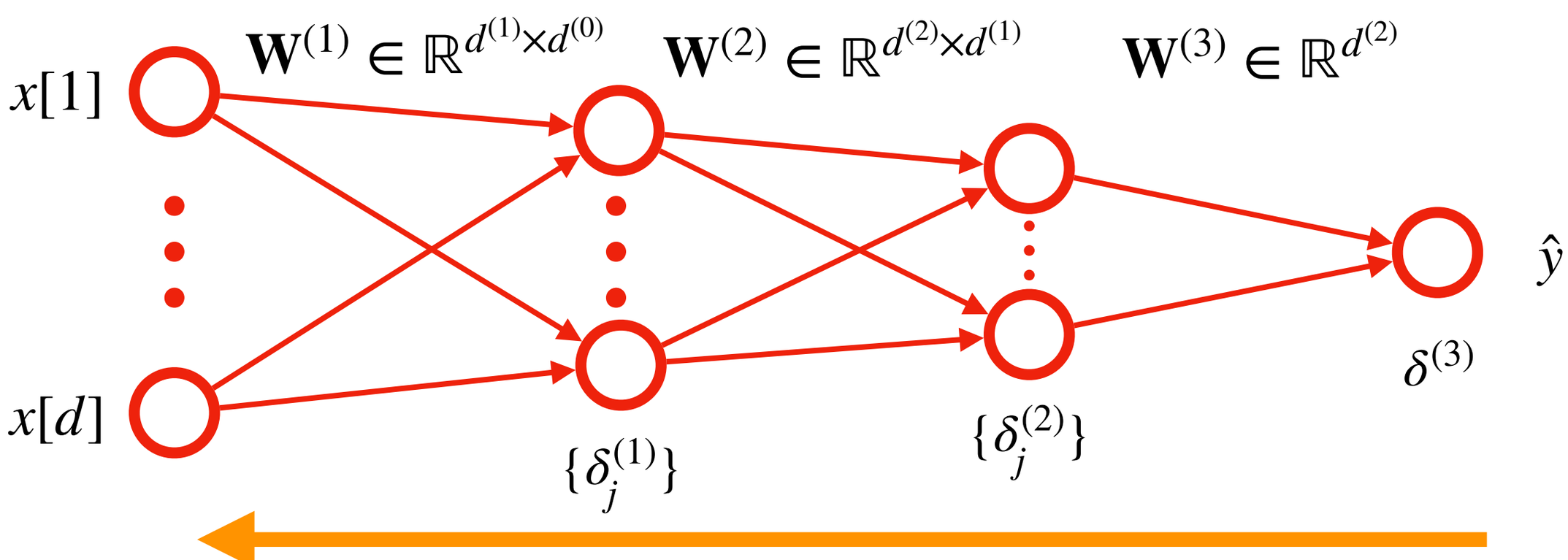


Time-complexity of evaluating the gradient

- suppose $g'(\cdot)$ can be evaluated in time similar to $g(\cdot)$ (say five unit time)

- using back-propagation algorithm, the total time to compute both $\ell(y, f_{\mathbf{W}}(x))$ and $\nabla_{\mathbf{W}} \ell(y, f_{\mathbf{W}}(x))$ is within a constant factor (e.g. a factor of five) of the time required to compute just $\ell(y, f_{\mathbf{W}}(x))$

$$\underbrace{2}_{\delta^{(L+1)}=2(a^{L+1}-y)} + \underbrace{d^{(L)}}_{\frac{\partial \ell(y,\hat{t})}{\partial w^{(L)}}=\delta^{(L+1)}z^{(L)}} + \underbrace{d^{(L-1)} \times (2d^{(L)}+1)}_{\delta_j^{(L-1)}=g'(a_j^{(L-1)})(\mathbf{W}_j^{(L)})^T\delta^{(L)}} + \cdots$$



Why is back-propagation so fast?

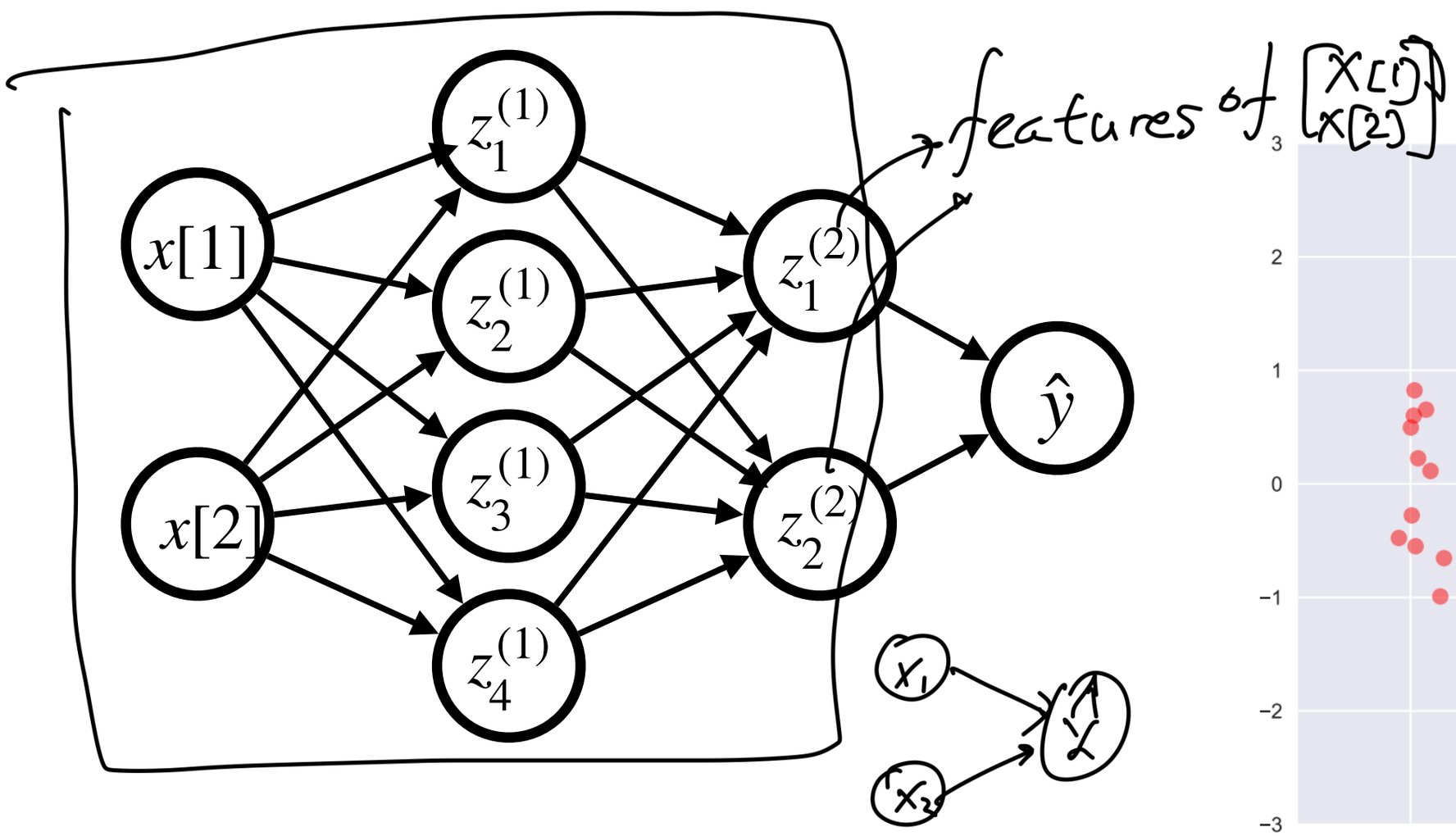
- evaluating a single derivative $\dfrac{\partial \ell(y, f_{\mathbf{W}}(x))}{\partial \mathbf{W}_{kj}^{(l)}}$ takes as much time as computing $\ell(y, f_{\mathbf{W}}(x))$

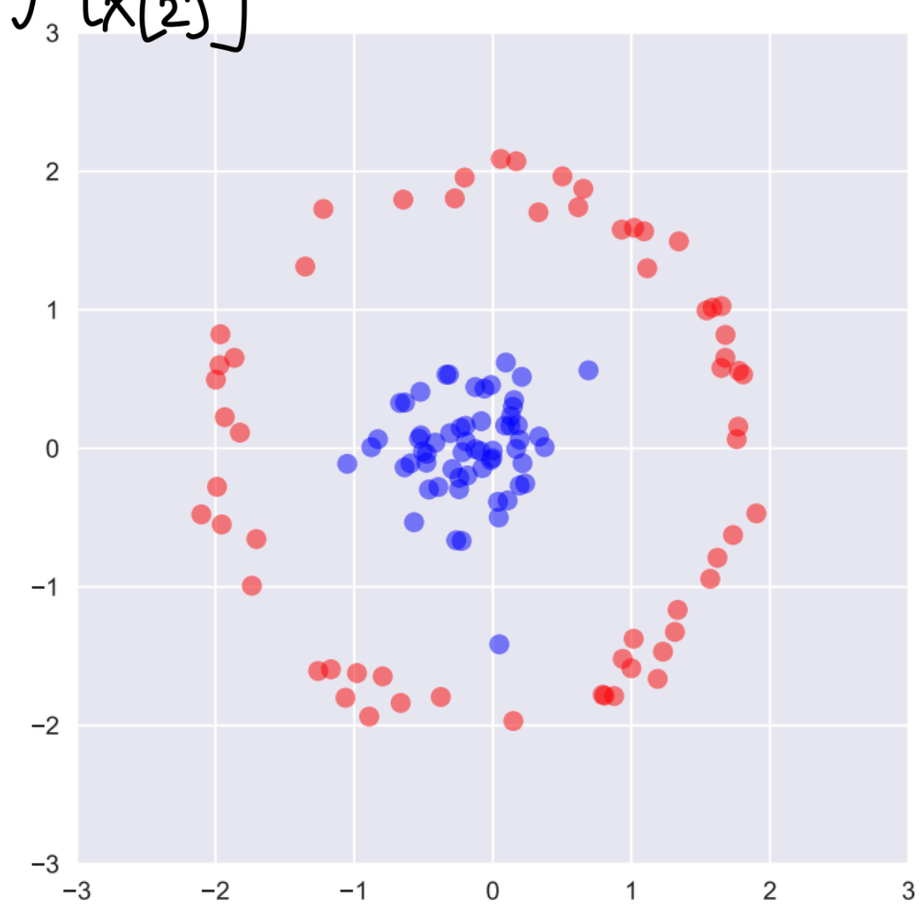
- back-propagation simultaneously computes all of them

- this result that evaluating the gradient takes a similar amount of time as just evaluating the function is known as **Baur-Strassen theorem** from
  - Walter Baur and Volker Strassen. “The complexity of partial derivatives.”, 1983.
  - Andreas Griewank and Andrea Walther. “Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.”, 2008.

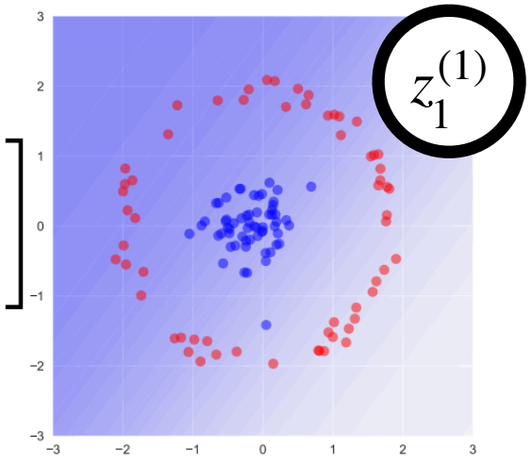
Example: classification

- Logistic loss $\ell(y, \hat{y}) = \log(1 + e^{y\hat{y}})$
- 2-hidden layers ($L = 2$)
- sigmoid activation $g(a) = \dfrac{1}{1 + e^{-a}}$
- Weights $\mathbf{W} \in \mathbb{R}^{25}$ (including the bias terms for each of the 7 nodes)

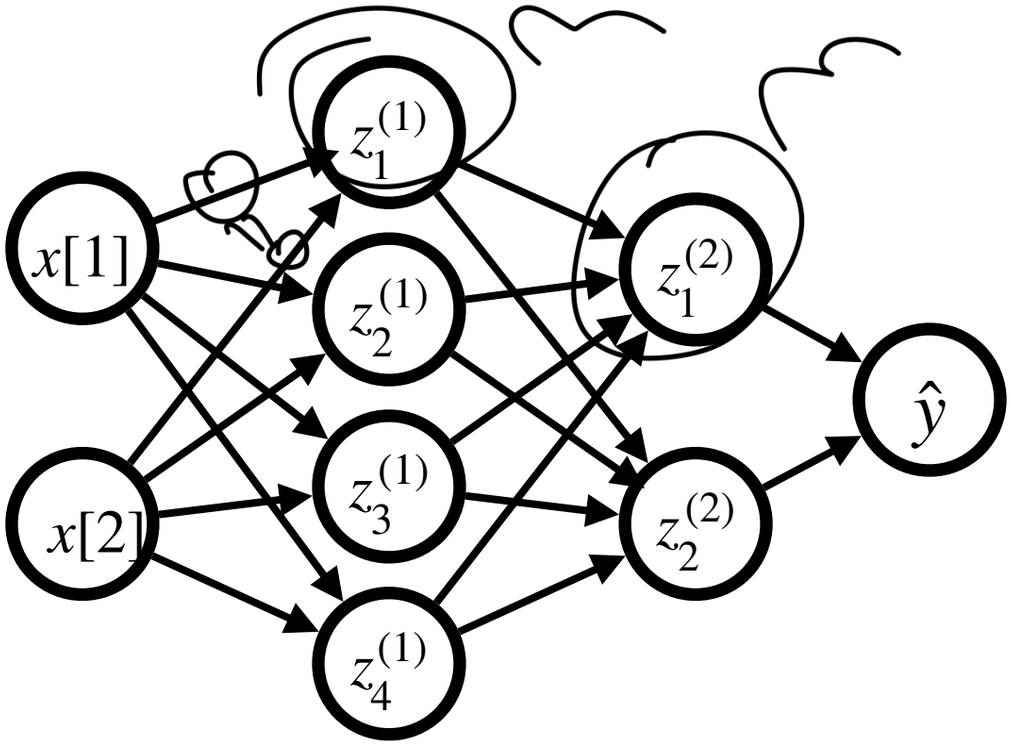




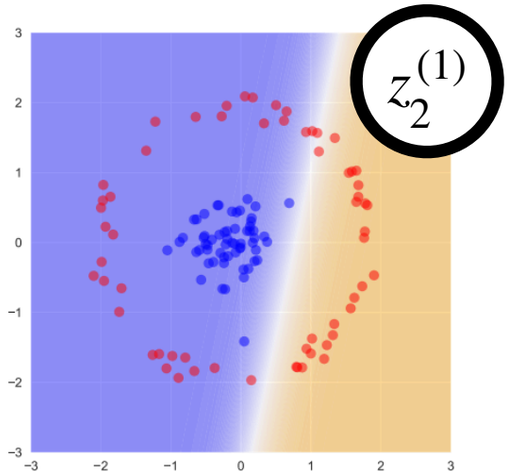
$$\begin{bmatrix} 0.63 \\ -0.89 \\ 0.66 \end{bmatrix}$$



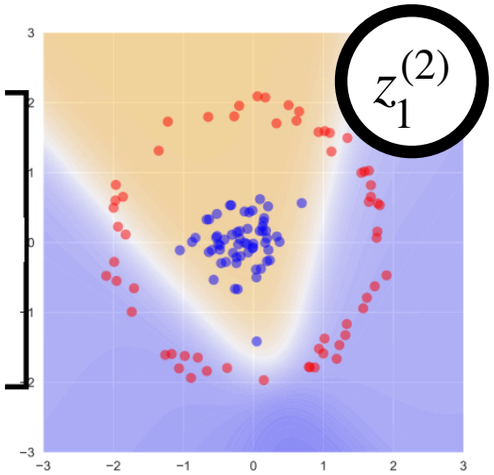
$= 0.63 - 0.89\,x[1] + 0.66\,x[2]$



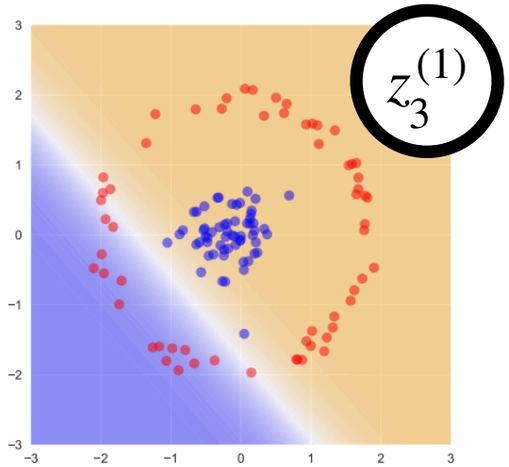
$$\begin{bmatrix} 3.5 \\ -4.5 \\ 0.94 \end{bmatrix}$$



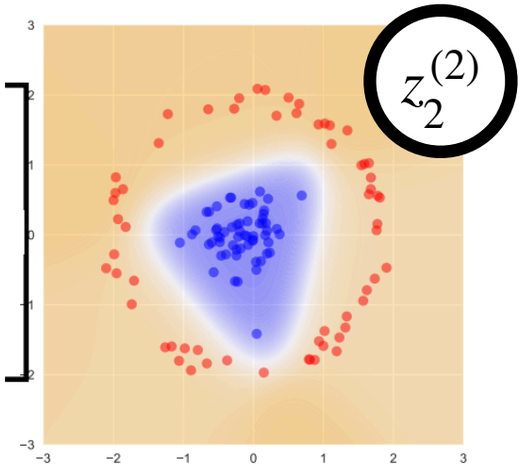
$$\begin{bmatrix} 1.6 \\ -0.81 \\ -3.5 \\ 1.8 \\ 2.3 \end{bmatrix}$$



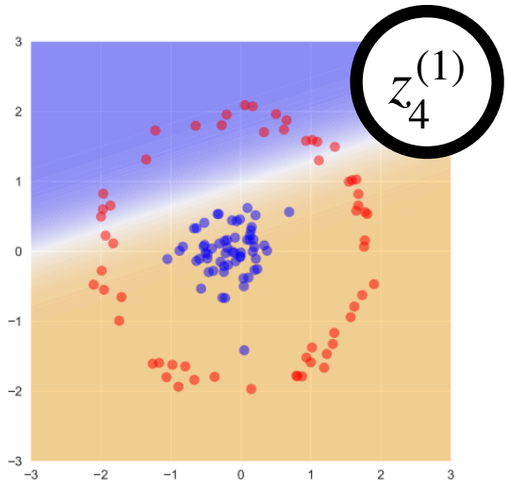
$$\begin{bmatrix} -4.4 \\ -2.9 \\ -2.5 \end{bmatrix}$$



$$\begin{bmatrix} -1.6 \\ -0.18 \\ 5.4 \\ -5.6 \\ -6.8 \end{bmatrix}$$

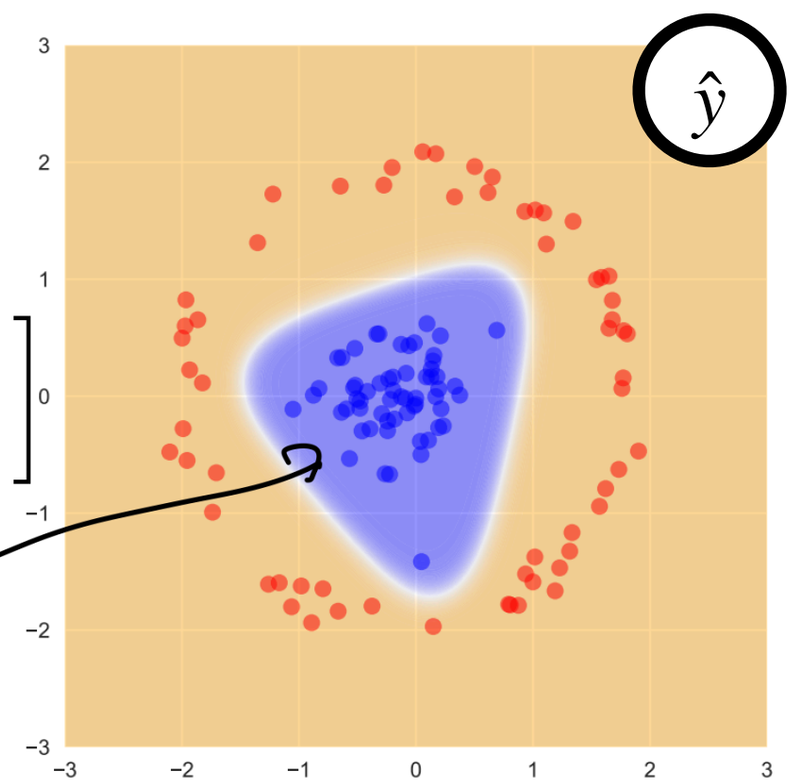


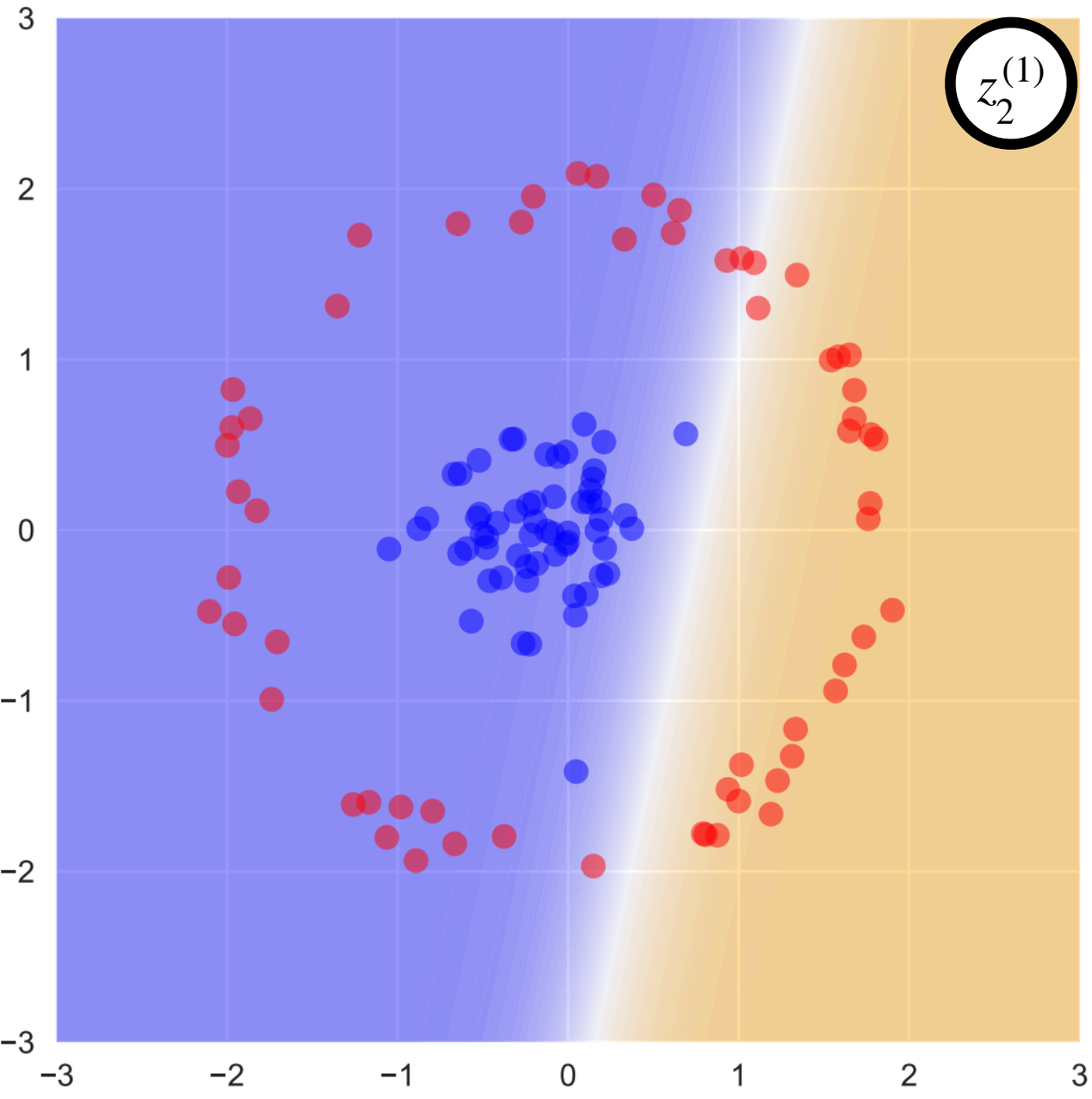
$$\begin{bmatrix} -3.7 \\ -1.3 \\ 3.9 \end{bmatrix}$$

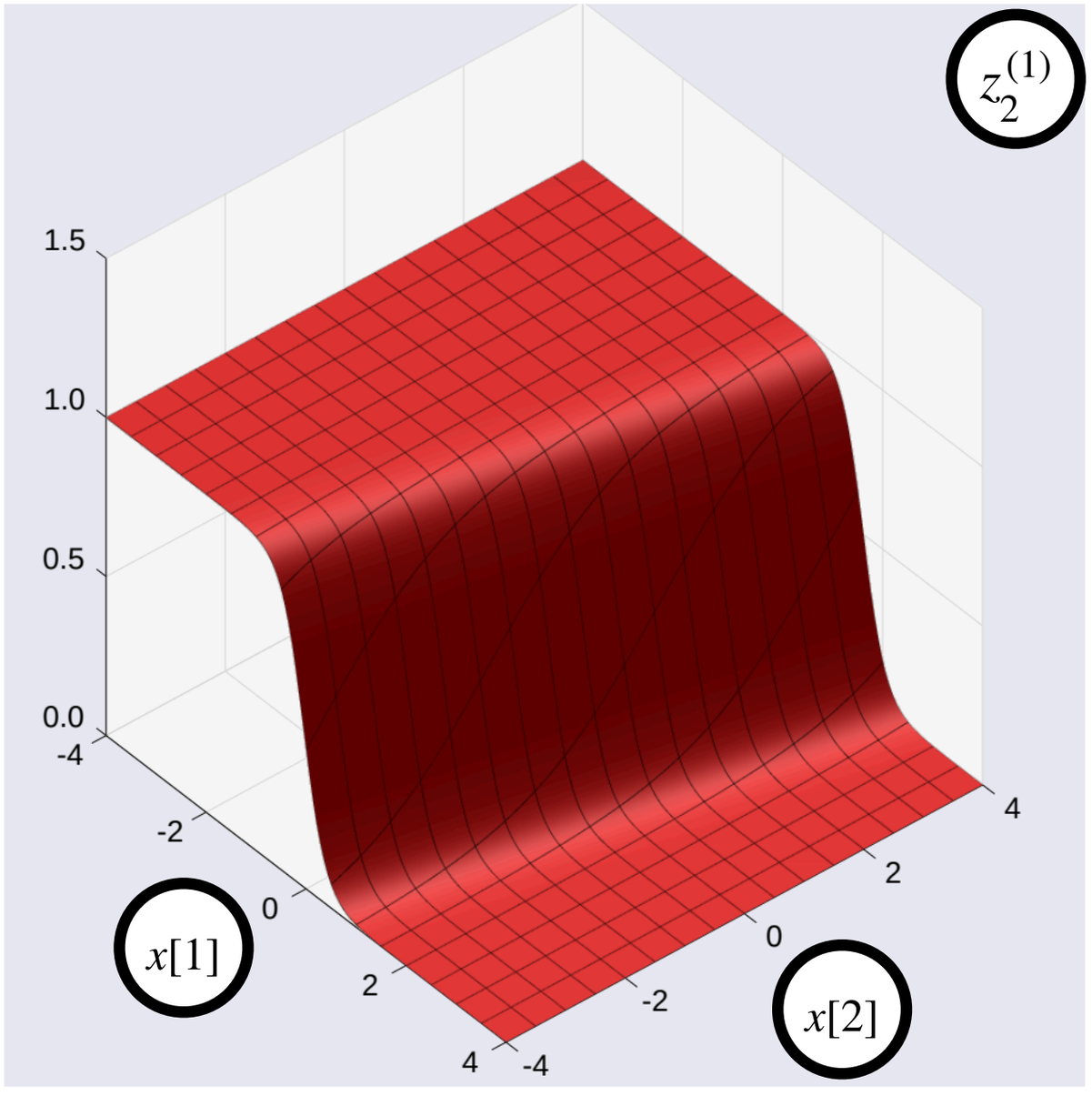


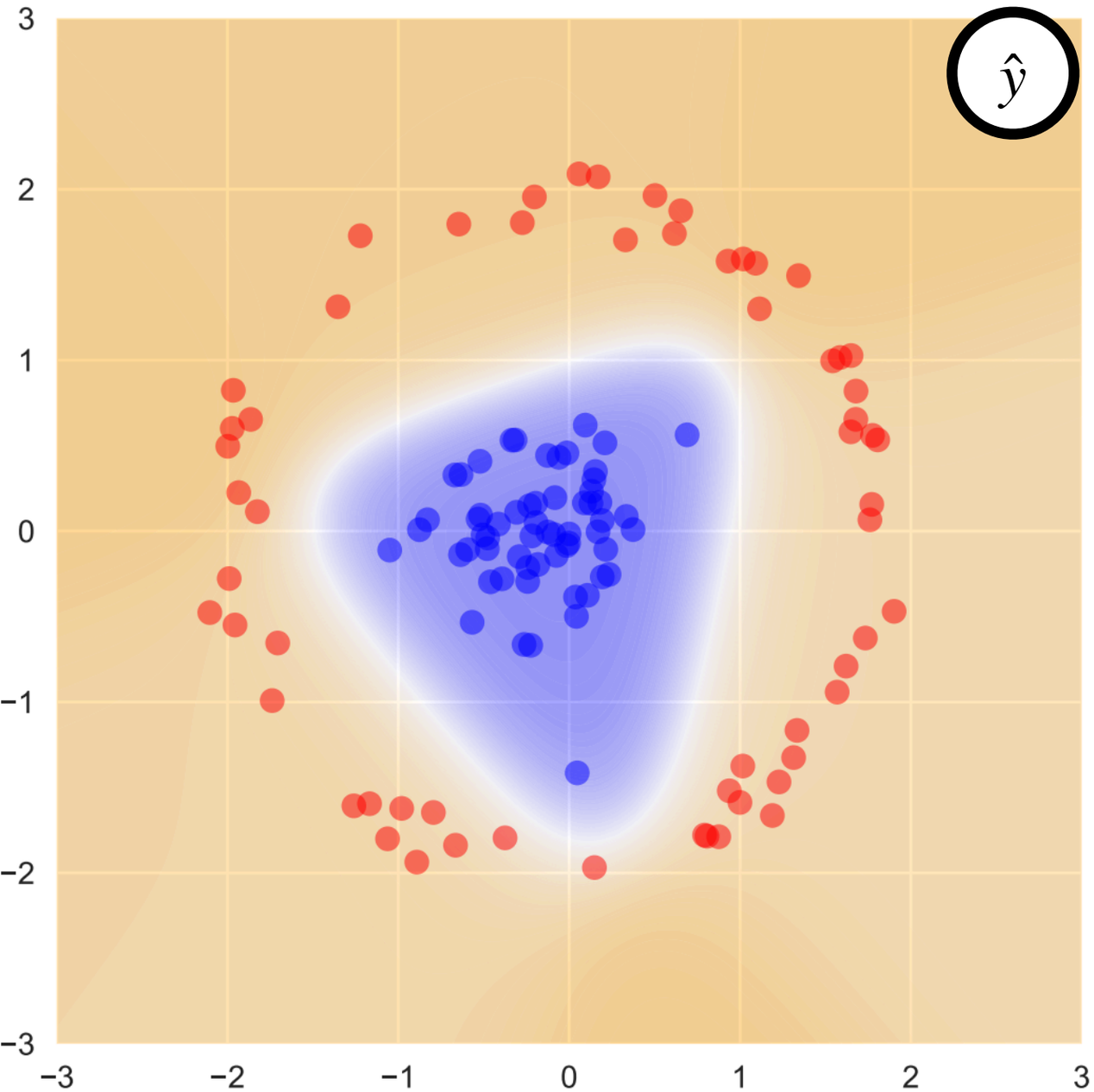
$$\begin{bmatrix} -5.2 \\ 0.62 \\ 13 \end{bmatrix}$$

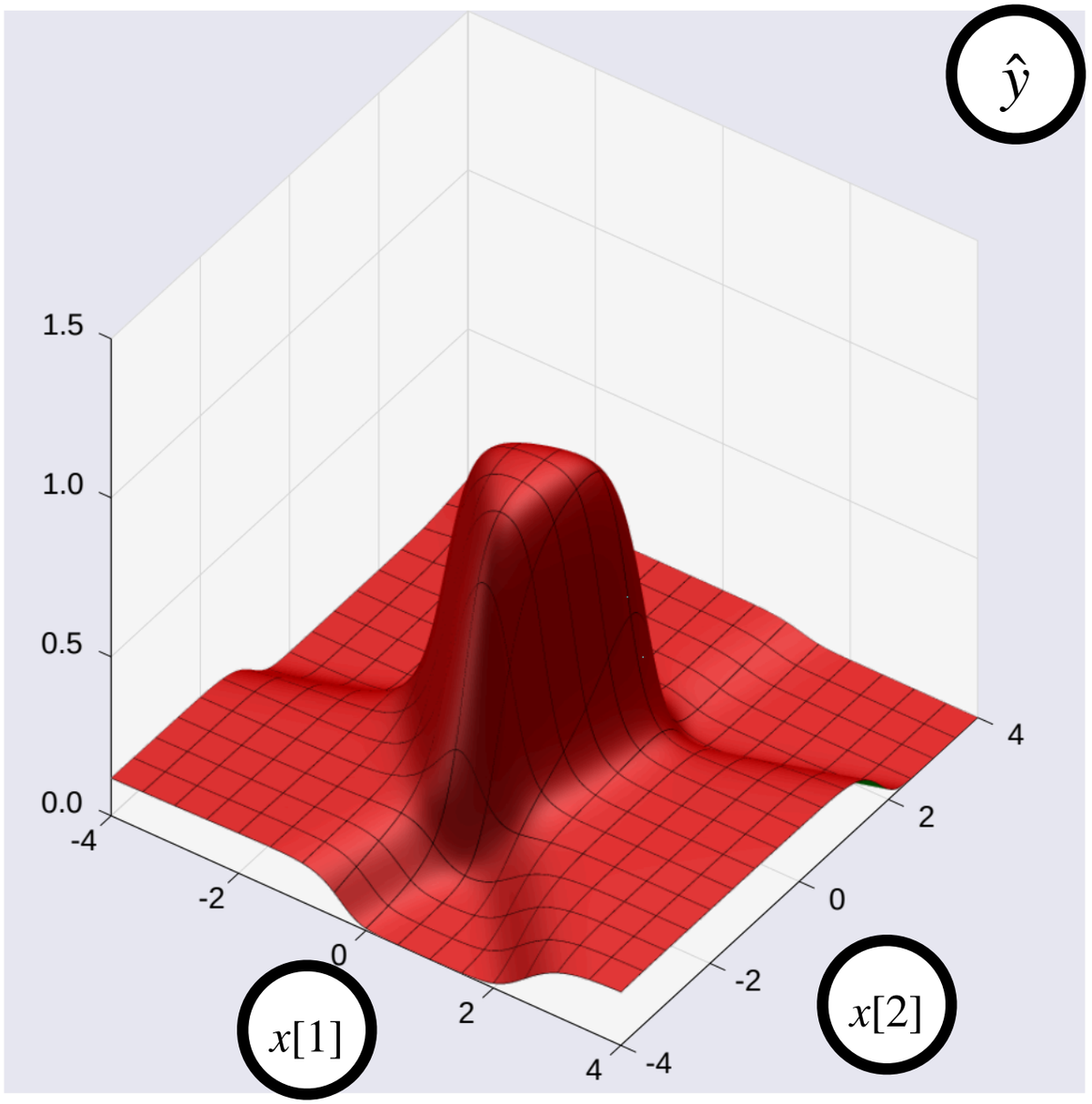
$\mathbb{I}(x_1^2 + x_2^2 \leq 2)$




$z_2^{(1)}$
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3


$z_2^{(1)}$
1.5
1.0
0.5
0.0
-4
-2
0
2
4
$x[1]$
4
2
0
-2
-4
$x[2]$


$\hat{y}$
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3

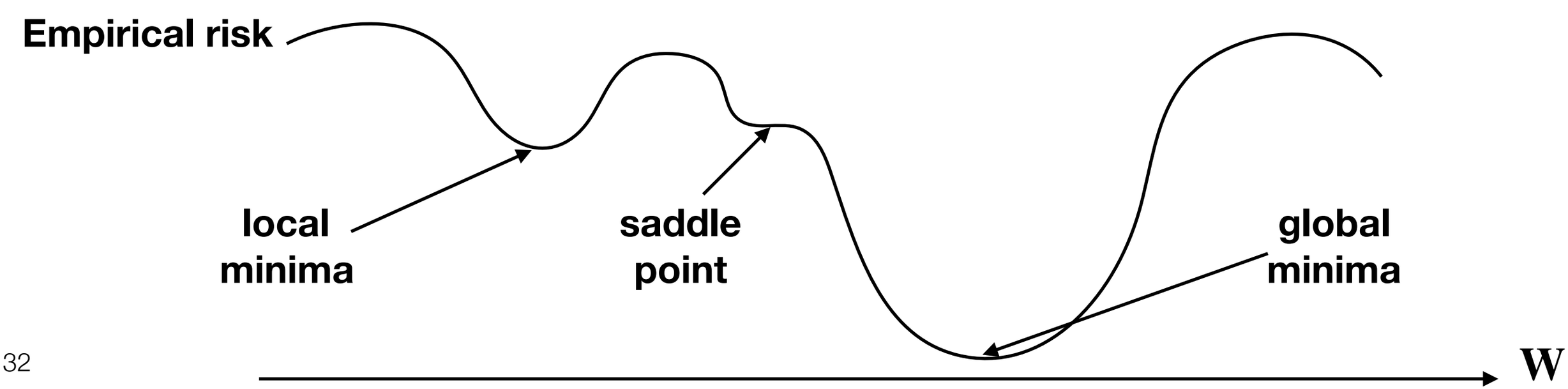

$\hat{y}$
1.5
1.0
0.5
0.0
-4
-2
0
2
4
$x[1]$
4
2
0
-2
-4
$x[2]$

Optimization on non-convex function

- to train a neural network, we use back-propagation to compute

$$\mathbf{W} \leftarrow \mathbf{W} - \eta \sum_{\mathbf{i=1}}^{\mathbf{n}} \nabla_{\mathbf{W}} \ell\left(\mathbf{y_i}, \mathbf{f_W}(\mathbf{x_i}) \right)$$

- however, gradient descent (or stochastic gradient descent) does not converge to global minima
  - we should not expect any efficient algorithm to find the global minima
  - instead, gradient descent stops moving when gradient is zero (or small, in practice)
  - in practice converging to local minima is inevitable, but one should avoid stopping at saddle points (points with gradient zero but are not local minima or maxima) if possible



Training feed-forward neural networks

- **initialization**
  - for convex optimization, we can initialize with $\mathbf{W} = 0$, and GD will find the optimal solution
  - for feed-forward neural network, we should not initialize with $\mathbf{W} = 0$ as it is a saddle point (the iterate never moves)

- back-propagation
  - recursively compute:

$$\delta_j^{(l)} = g'(a_j^{(l)}) \sum_{k=1}^{d^{(l+1)}} w_{kj}^{(l+1)} \delta_k^{(l+1)}$$

and the derivatives

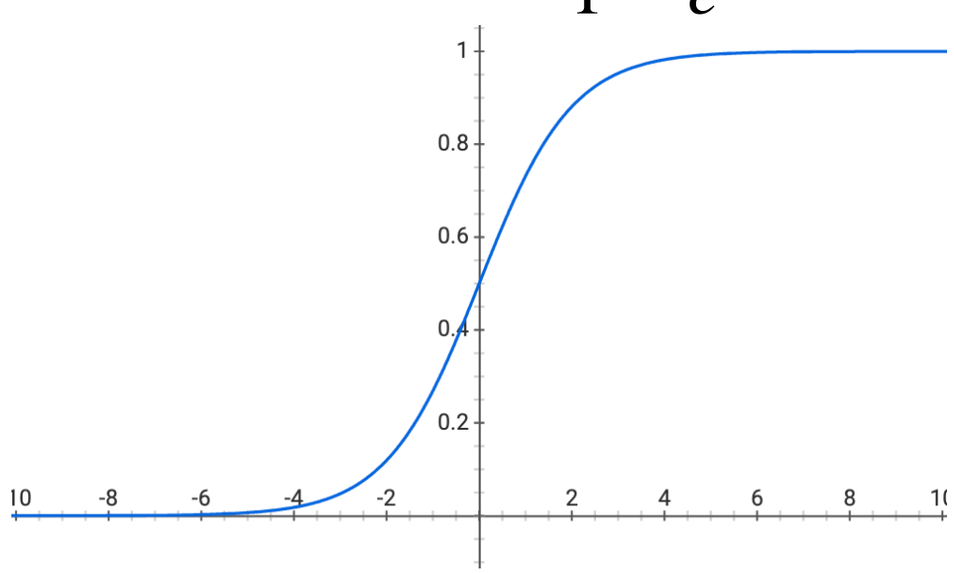
$$\frac{\partial \ell(y, \hat{y})}{\partial w_{kj}^{(l)}} = \delta_k^{(l)} z_j^{(l-1)}$$

  - initializing with **small** $\mathbf{W}$ is bad, as initial gradient is small, and takes long time for the empirical risk to decrease
  - initializing with **large** $\mathbf{W}$ can also be bad, as we show next
  - how do we choose the right initialization?
    - **Xavier initialization:** initialize s.t. each $a_j^{(l)}$ has unit variance over the training examples x_i's

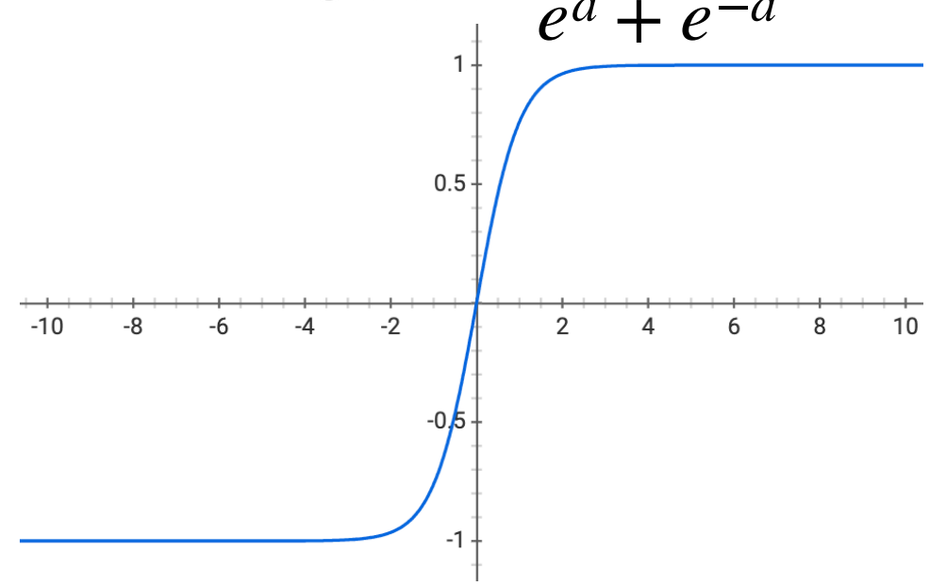
Training feed-forward neural networks

- **Saturation/vanishing gradients**
  - this applies to **sigmoid** or **tanh** activation functions
  - when $a_j^{(l)}$'s have large magnitude (either because of large initialization, large learning rates, etc.)
  - when using sigmoid or tanh, one should be careful about vanishing gradients

sigmoid $g(a) = \dfrac{1}{1 - e^{-a}}$



tanh $g(a) = \dfrac{e^{a} - e^{-a}}{e^{a} + e^{-a}}$



Auto-differentiation

- the ability to automatically differentiate functions has become a core ML tool
- a breakthrough mathematical result behind the success is that all derivatives can be computed in time similar to the runtime of evaluating the function itself
- Understanding auto-differentiation allows us to better understand how software like PyTorch and TensorFlow work, and hence better utilize those tools to get the desired result (in training ML models)
- back-propagation is a special case of auto-differentiation

Auto-differentiation

- **Computational model**
  - for a real valued function $f(\mathbf{W}) : \mathbb{R}^{d^*} \rightarrow \mathbb{R}$, we seek to compute the gradient $\nabla_{\mathbf{W}} f(\mathbf{W})$
  - we first need to specify the function $f(\mathbf{W})$
  - suppose
    $f(w_1, w_2) = (\overbrace{\sin(2\pi w_1/w_2)}^{z_2} + 3w_1/w_2 - \exp(2w_2)) \times (3w_1/w_2 - \exp(2w_2))$
  - here is a program that computes $f(w_1, w_2)$
    - **input:** $z_0 = (w_1, w_2)$

      $z_1 = w_1/w_2$

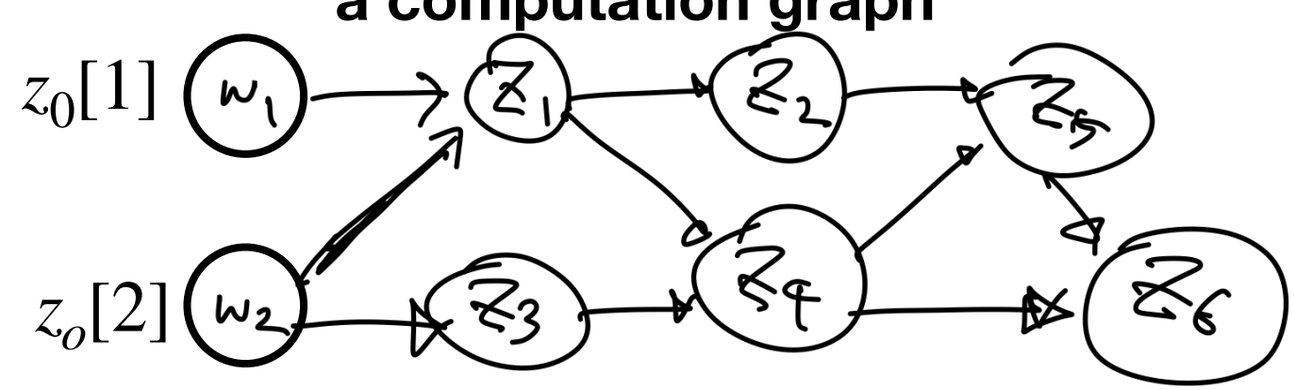
      $z_2 = \sin(2\pi z_1)$

      $z_3 = \exp(2w_2)$

      $z_4 = 3z_1 - z_3$

      $z_5 = z_2 + z_4$
    - **output:** $z_6 = z_4 \times z_5$

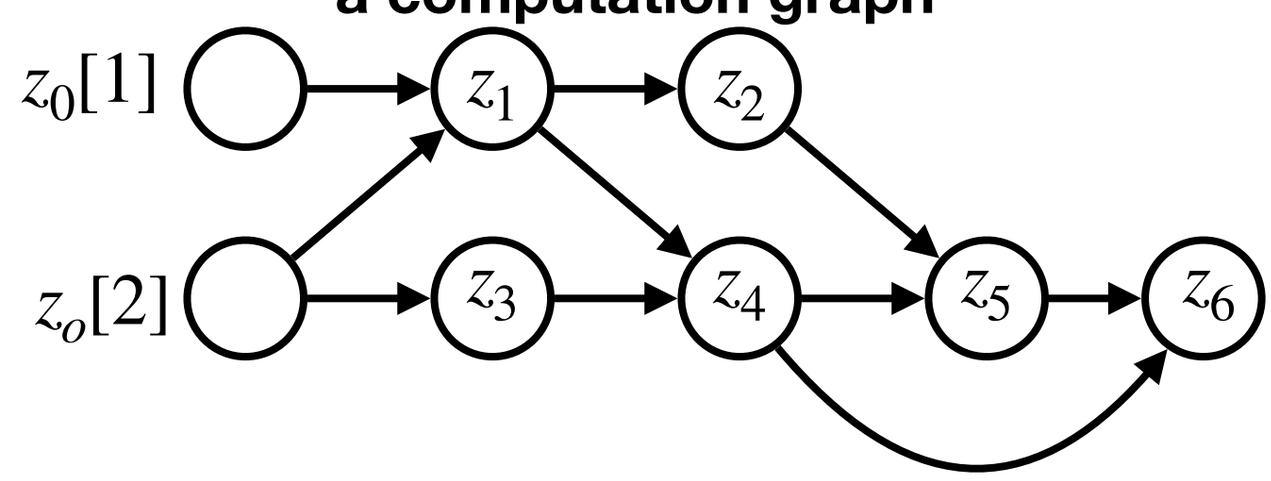
The program can be translated into a computation graph



Auto-differentiation

- **Computational model**
  - for a real valued function $f(\mathbf{W}) : \mathbb{R}^{d^*} \rightarrow \mathbb{R}$, we seek to compute the gradient $\nabla_{\mathbf{W}} f(\mathbf{W})$
  - we first need to specify the function $f(\mathbf{W})$
  - suppose
    $f(w_1, w_2) \;=\; (\sin(2\pi w_1/w_2) + 3w_1/w_2 - \exp(2w_2)) \times (3w_1/w_2 - \exp(2w_2))$
  - here is a program that computes $f(w_1, w_2)$
    - **input:** $z_0 = (w_1, w_2)$
    - $z_1 = w_1/w_2$
    - $z_2 = \sin(2\pi z_1)$
    - $z_3 = \exp(2w_2)$
    - $z_4 = 3z_1 - z_3$
    - $z_5 = z_2 + z_4$
    - **output:** $z_6 = z_4 \times z_5$

The program can be translated into a computation graph



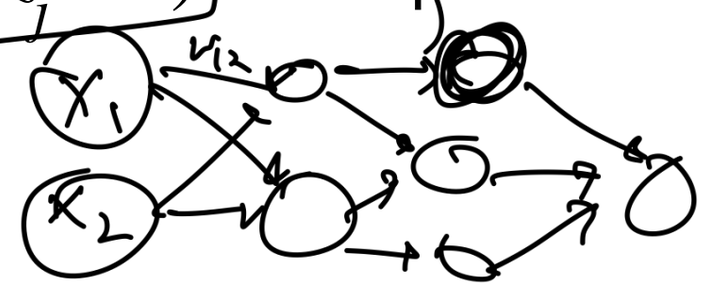
Auto-differentiation

- such a program is called an **evaluation trace**
- we give an abstract definition of evaluation trace
  - we define a set of differentiable functions $h \in \mathscr{H}$
  - we use functions from $\mathscr{H}$ for intermediate variables, and create an evaluation trace
  - all intermediate variables $z_1, \ldots, z_t$ will be **scalars**
  - each variable is a node in computational graph
  - only the input $z_0 = w \in \mathbb{R}^d$ is a vector, which is represented by d nodes: $z_0[1] = w_1, \ldots, z_0[d] = w_d$

  - **input**: $z_0 = w$
  - $z_1 = h_1$(a fixed subset of parent variables in z_0)
    $\vdots$
  - $z_t = h_t$(a fixed subset of parent variables in $z_{0:t-1}$)
    $\vdots$
  - **output**: $z_T = h_T$(a fixed subset of parent variabels in $z_{0:T-1}$)

Auto-differentiation

- we will assume that $\mathscr{H}$ only contains
  - 1. **affine transformation**, e.g. $z_4 = 3z_1 - z_3$ $\quad z_5 = z_1^3 z_4^2$
  - 2. **product of variables**, or some power of variables, e.g. $z_1 = w_1/w_2$
  - 3. **one-dimensional differentiable function**, e.g. $z_3 = \exp(2w_2)$
    - note that we only allow one-dimensional input, i.e. $z_3 = \exp(z_1 + z_2)$ is not allowed $\quad z_3 = z_1 + z_2$, $z_4 = \exp(z_3)$
- the computation graph we studied for feed-forward neural networks is not a special case of the above model, as
  - the parameter $\mathbf{W}$ we want to take gradient with respect to, is not at the input
  - and $h(\sum_j w_{kj}^{(l)} z_j^{(l-1)})$ is not permitted, as it is not a one-dimensional function



Auto-differentiation

- auto-differentiation uses the chain rule to differentiate a function represented by its evaluation trace and compute $\nabla_w f(w)$
- the insight is that z_t affects target only through its children nodes
- we work backwards
- $$\frac{\partial z_T}{\partial z_T} = 1$$
- we use the chain rule:

$$\frac{\partial z_T}{\partial z_t} = \sum_{c \in \text{children of } z_t} \frac{\partial z_T}{\partial z_c} \frac{\partial z_c}{\partial z_t}$$

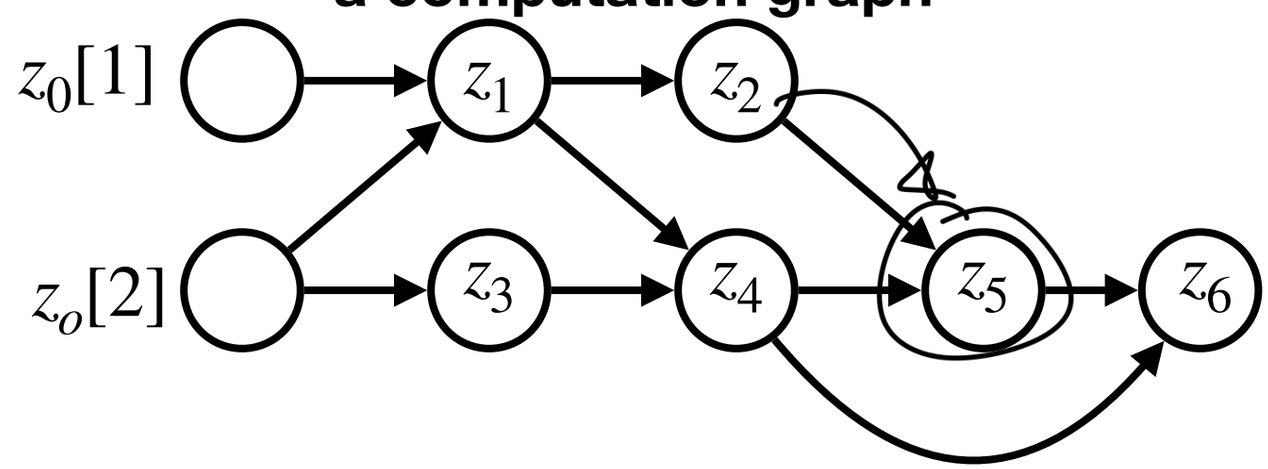
where a child node is a node that z_t directly points to

$$\frac{\partial z_T}{\partial z_0[1]} = \frac{\partial f(w)}{\partial f(w_1)}$$

$$f(w_1, w_2) \;=\; (\sin(2\pi w_1/w_2) + 3w_1/w_2 - \exp(2w_2)) \times (3w_1/w_2 - \exp(2w_2))$$

- $z_1 = w_1/w_2$
- $z_2 = \sin(2\pi z_1)$
- $z_3 = \exp(2w_2)$
- $z_4 = 3z_1 - z_3$
- $z_5 = z_2 + z_4$
- **output:** $z_6 = z_4 \times z_5$

The program can be translated into a computation graph



$$\frac{\partial z_6}{\partial z_5} = z_4$$

$$\frac{\partial z_6}{\partial z_2} = \frac{\partial z_6}{\partial z_5} \cdot \frac{\partial z_5}{\partial z_2} = z_4 \cdot 1$$

$$\cancel{\frac{\partial z_6}{\partial z_4} = z_5}$$

$$\frac{\partial z_6}{\partial z_5} \cdot \frac{\partial z_5}{\partial z_4} + \frac{\partial z_6}{\partial z_6} \cdot \frac{\partial z_6}{\partial z_4}$$

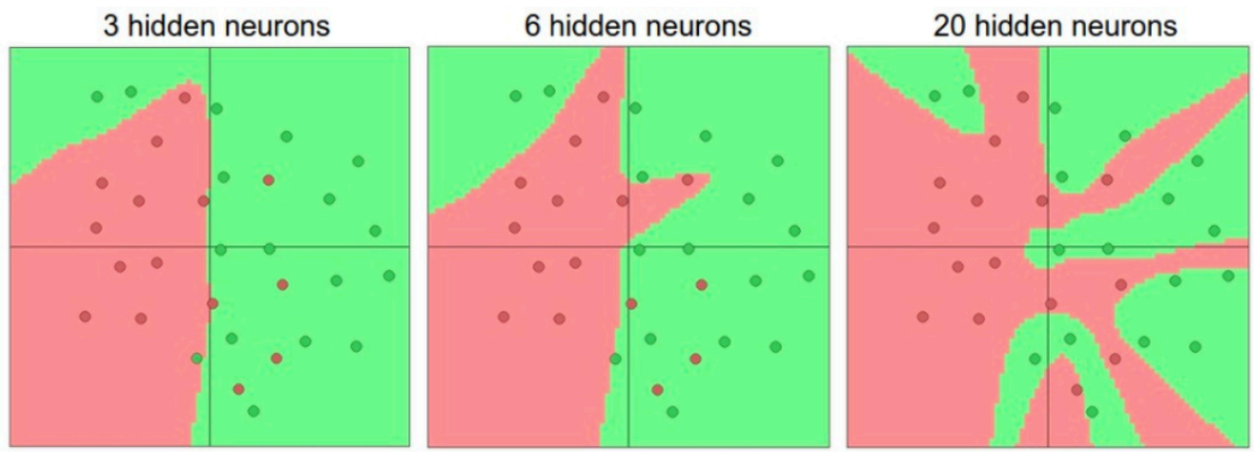
Auto-differentiation

- **the reverse mode of auto-differentiation**
  - forward pass
    - compute $f(w)$ and store all intermediate variables
  - backward pass
    - initialize: $\frac{\partial z_T}{\partial z_T} = 1$
    - for $t = T - 1, \ldots, 0$
      - $\frac{\partial z_T}{\partial z_t} = \sum_{c \in \text{children of } z_t} \frac{\partial z_T}{\partial z_c} \frac{\partial z_c}{\partial z_t}$
  - return: $\frac{\partial z_T}{\partial z_0} = \nabla_w f(w)$

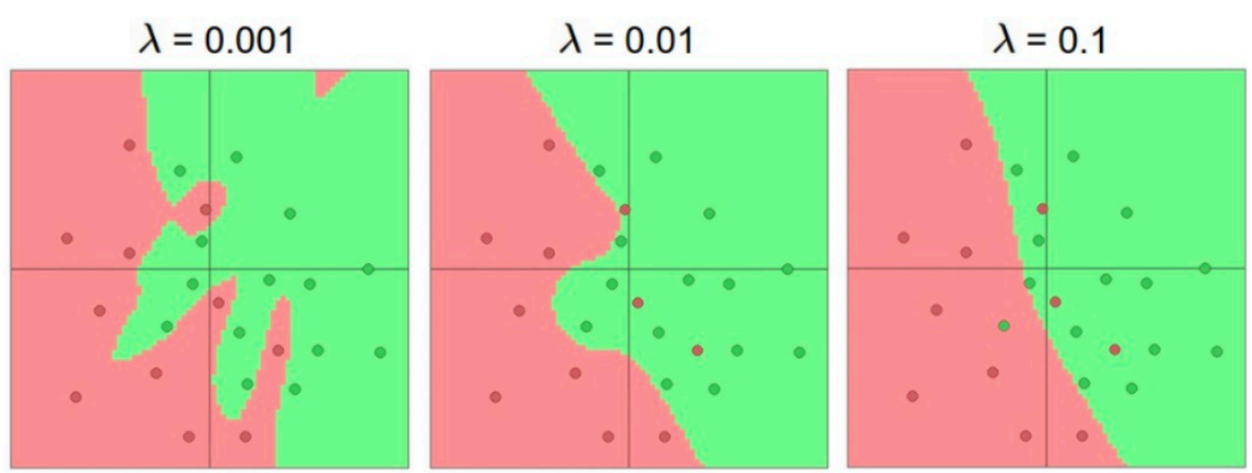
Over-fitting and regularization in neural networks

Representation power vs. size and regularization

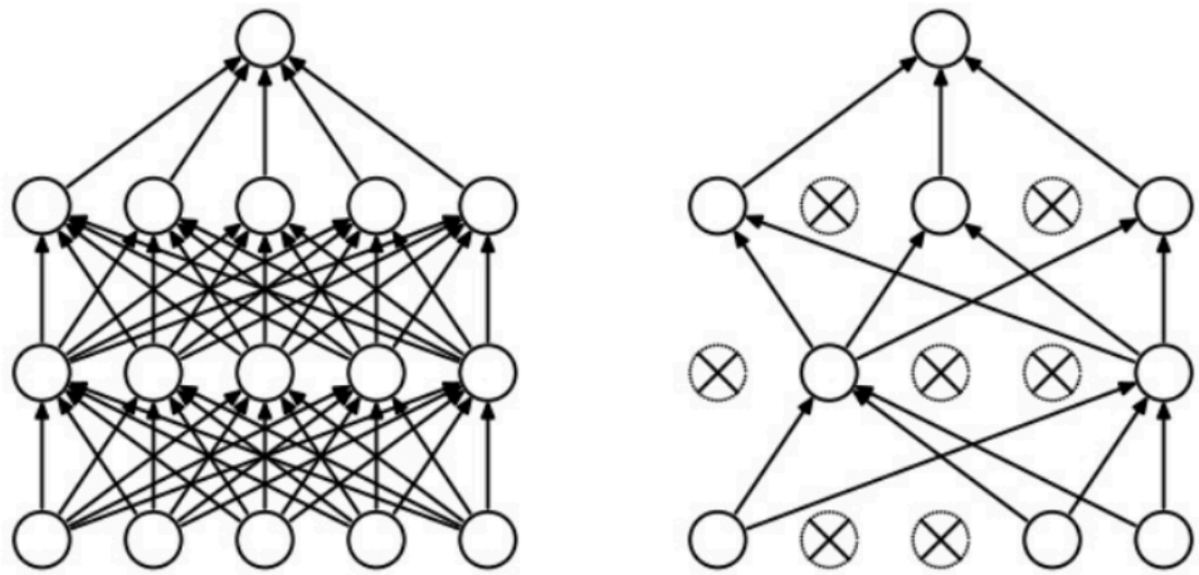
- more layers and more nodes in each layer gives larger representation power, but can lead to overfitting



- larger regularization coefficient gives smoother surface, potentially avoiding overfitting



Dropout (yet another regularization technique)



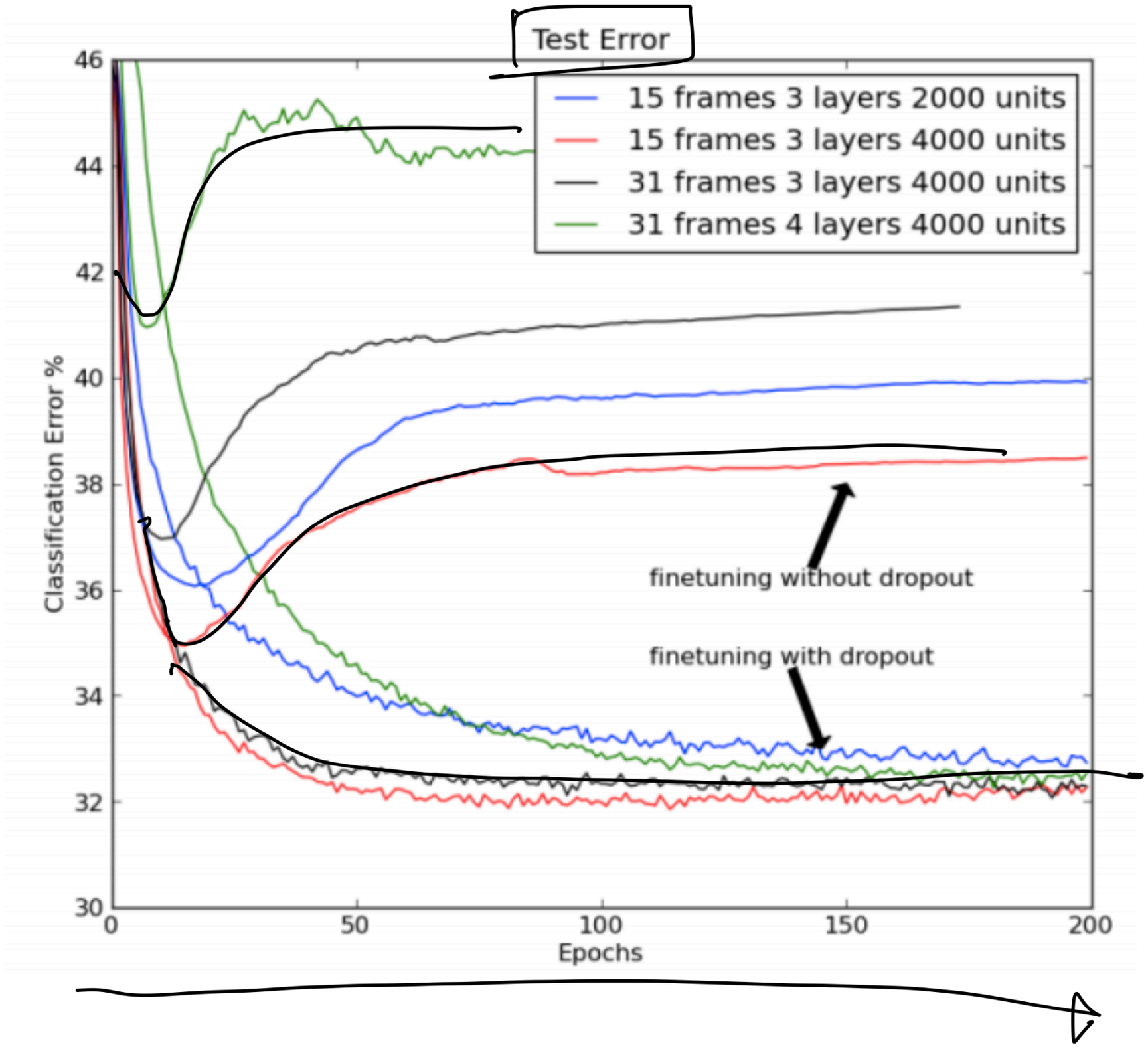
(a) Standard Neural Net (b) After applying dropout.

- **Dropout** is another recently introduced (["Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Srivastava, Hinton, Krizhevsky, Sutskever, Salakhutdinov, 2014]) technique for regularization
- at training, each "neuron" is active with some probability p, and set to zero otherwise
- at testing, all neurons are active, but scaled by p

Drop out encourages all nodes to contribute equally, as nodes are randomly unavailable

- pseudo code

```
p = 0.5
# probability of keeping a unit active.
def train_step(X):
      # forward pass for example 2-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
      out = np.dot(W2, H1) + b2
      # 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
      out = np.dot(W2, H1) + b2
```



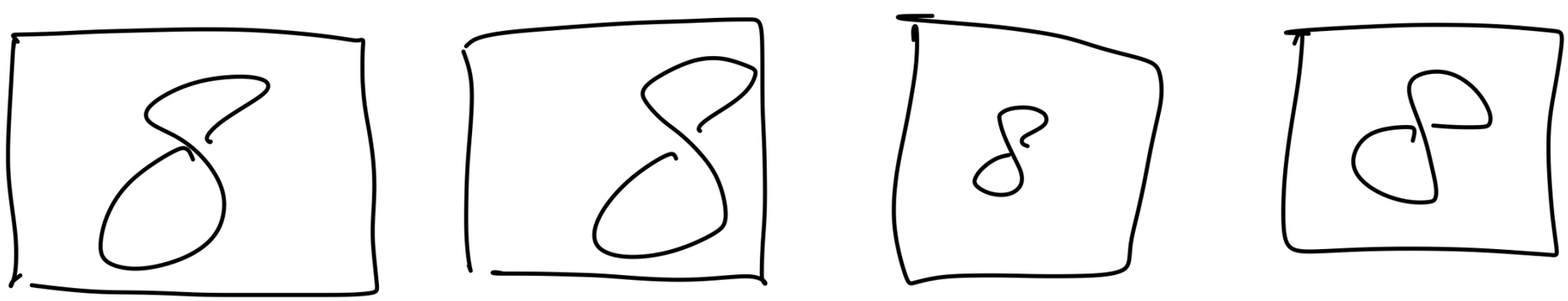
50% dropout for hidden layer and 20% dropout for input layer

Convolutional Neural Networks

Invariances

- in many applications, we know that the prediction should be unchanged or **invariant** when the input is transformed in certain ways
  - for example, in image classification, the prediction should be invariant to
    - changing position (translation invariance),
    - changing its size (scale invariance), or
    - small rotations (rotation invariance)
  - for speech, faster or slower time scale should not change the meaning

- however, these invariant transforms significantly change the raw data, making it challenging to train an invariant model
- one solution is **data augmentation**, which is including many invariant transformed versions of the training data

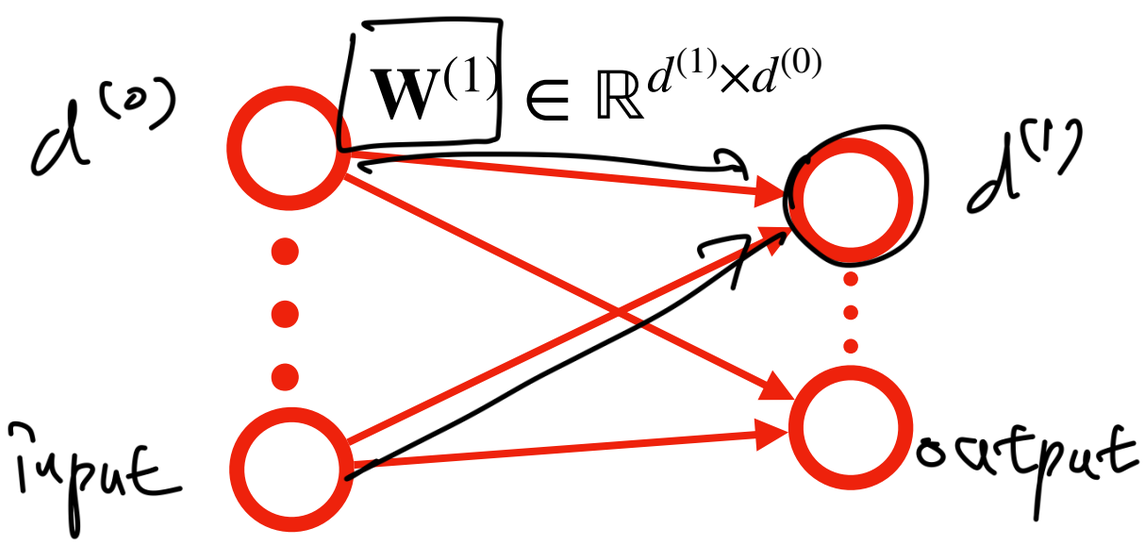


- this often times is impractical due to the increase in training data, and training time

Convolutional networks

- in a specific but popular example of image classification, we want to design **neural network architectures** that captures the invariances that we want to impose (translation, small rotation, scale)
- also, a key aspect of image data is that close-by pixels are likely to be more related than distant pixels (locality)
- convolutional neural networks exploit this property by first extracting **local features** and merge those local features in later stages when constructing higher order features

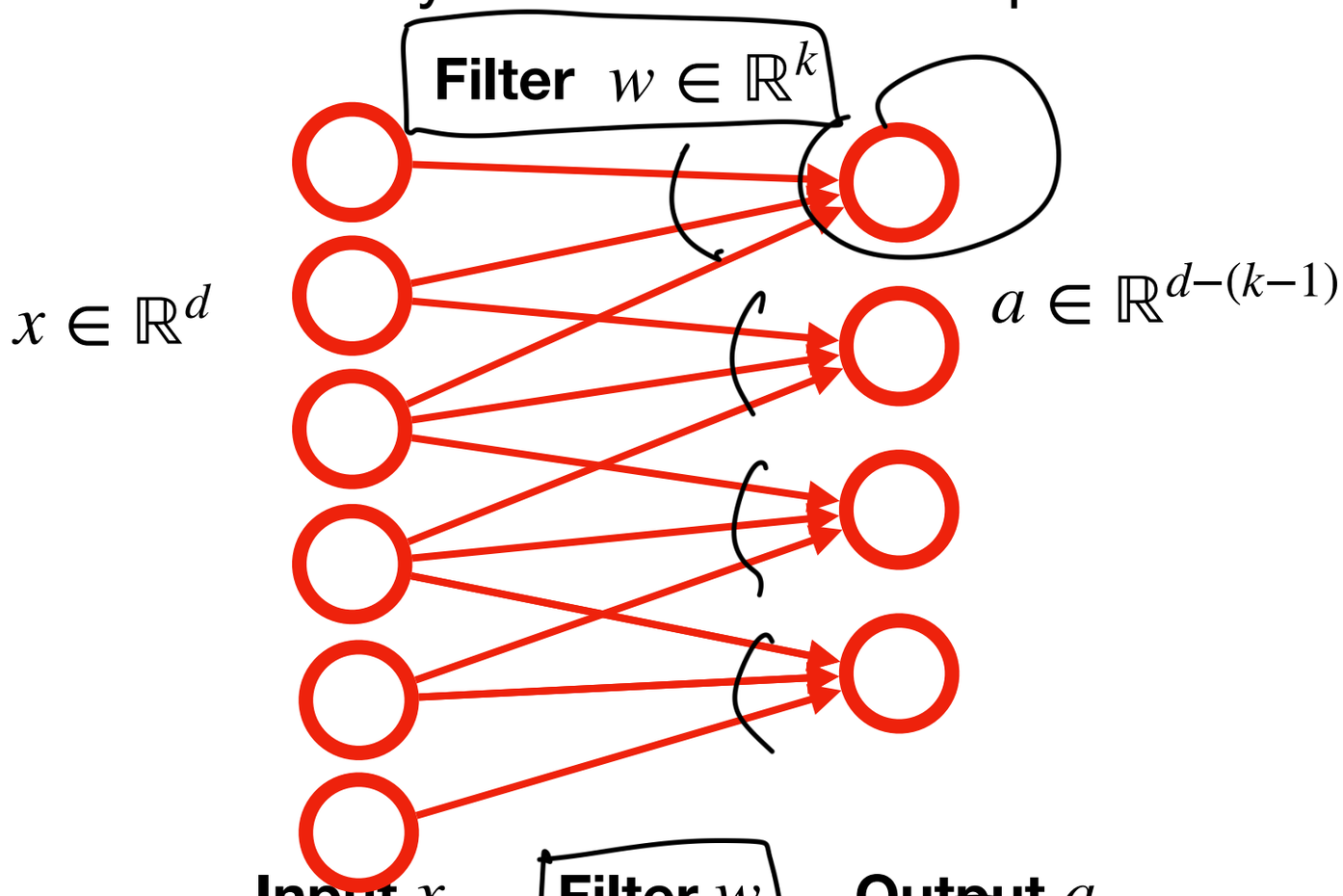
- recall that **a fully connected layer** of feed-forward neural network extracts linear features and applies a non-linear activation



$$a^{(1)} = \mathbf{W}^{(1)} x$$

Example: 1-d convolution

- **convolutional layer** extracts the same local features, but for many locations of the input

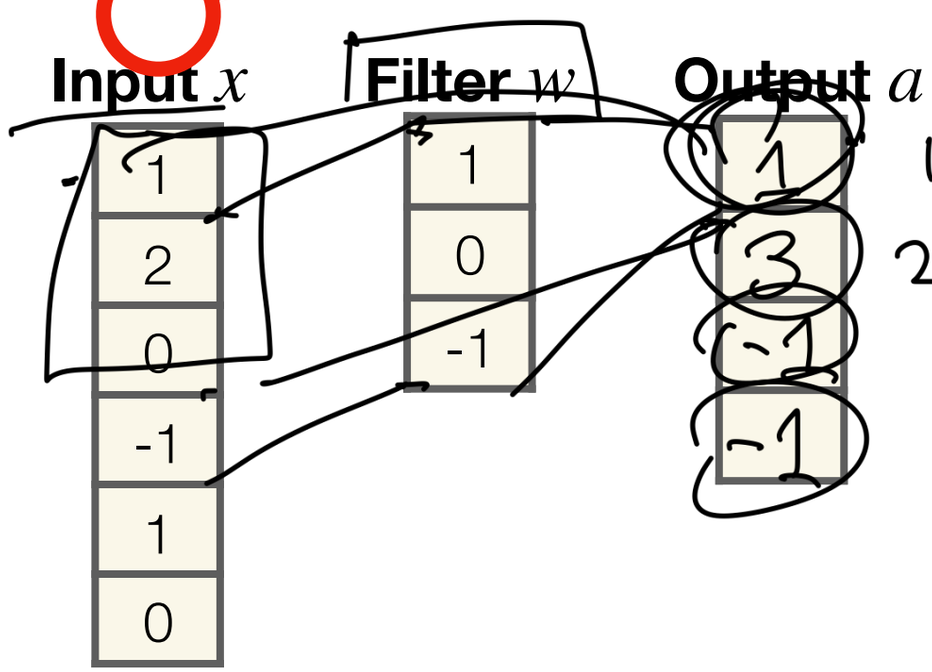


Convolution operation

$$a = w \star x$$

$$\boxed{a_i = \sum_{j=1}^{k} w_j x_{i+j-1}}$$

Applies the same filter through a sliding window on the input



$1 \cdot 1 + 2 \cdot 0 + 0(-1)$

$2 + 0 + 1$

$(0,-1,1) \cdot (1,0,-1) = -1$

$(-1,1,0) \cdot (1,0,-1) =$

- reduces number of parameters
- captures locality
- imposes translation invariance

2-d convolution

- consider an image $x \in \mathbb{R}^{d \times d}$ and a filter $w \in \mathbb{R}^{k \times k}$

Input x

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Filter w

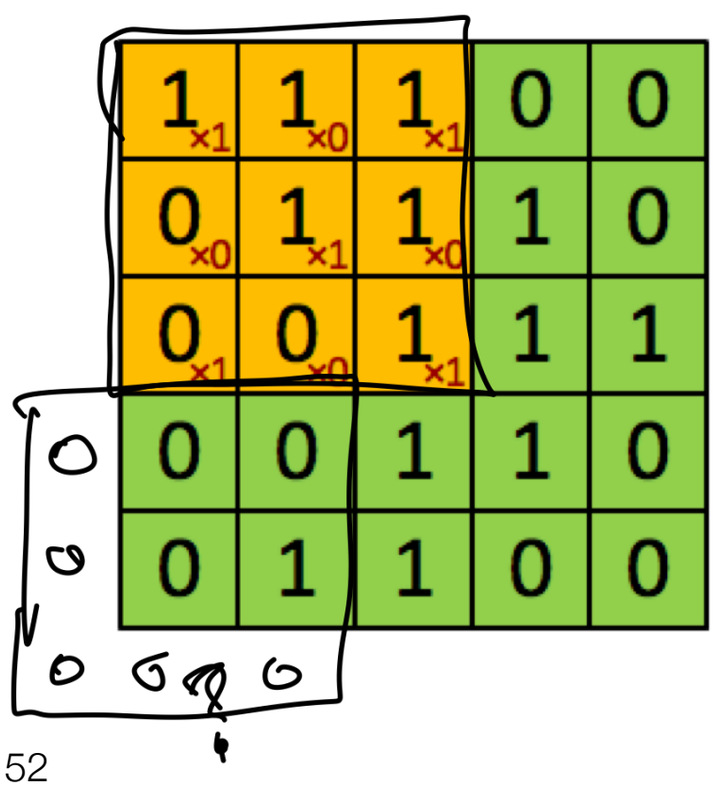
1	0	1
0	1	0
1	0	1

Convolution operation

$$a = w \star x$$

$$a_{i_1,i_2} = \sum_{j_1,j_2=1}^{k} w_{j_1,j_2} x_{i_1+j_1-1,i_2+j_2-1}$$

Output a

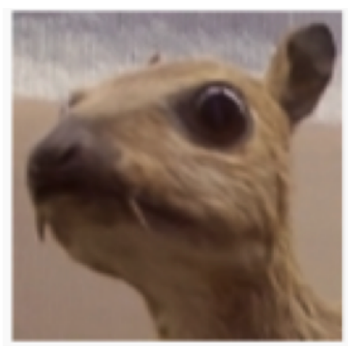


4	3	

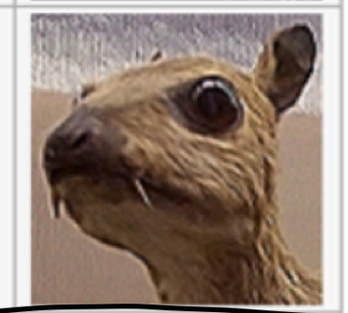
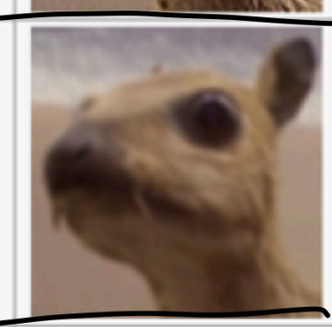
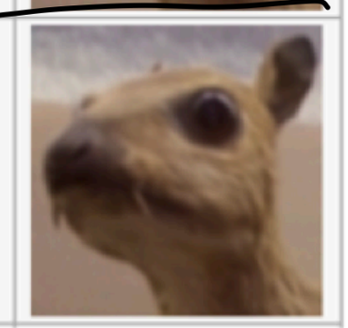
Role of the filter

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

Input x



$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

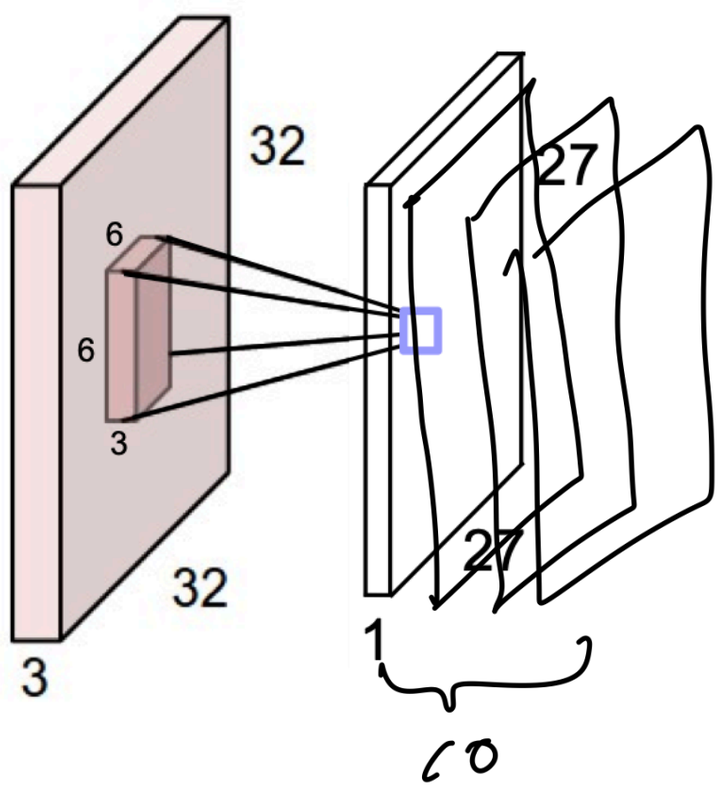
	Filter w	**Output** a
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

3-d convolution (single filter case)

- input $x \in \mathbb{R}^{d \times d \times 3}$, filter $w \in \mathbb{R}^{k \times k \times 3}$, output $a \in \mathbb{R}^{(d-(k-1)) \times (d-(k-1))}$

Convolution operation

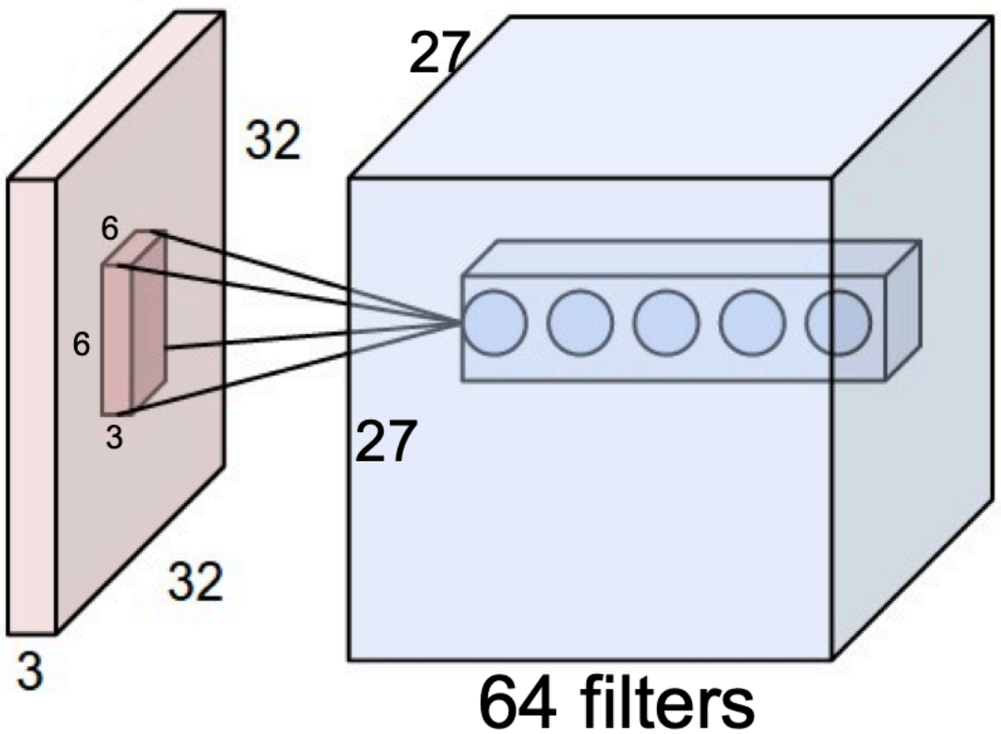
$$a = w \star x$$



$$a_{i_1,i_2} = \sum_{c=1}^{3} \sum_{j_1,j_2=1}^{k} w_{j_1,j_2,c} x_{i_1+j_1-1,i_2+j_2-1,c}$$

3-d convolution (one convolutional layer)

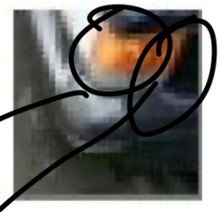
- input $x \in \mathbb{R}^{d \times d \times 3}$, M filters $w_m \in \mathbb{R}^{k \times k \times 3}$, output $a \in \mathbb{R}^{(d-(k-1)) \times (d-(k-1))}$



- M convolutions are computed
- this is still a linear operation
- we apply a non-linear activation at the output

Input

$x \in \mathbb{R}^{d \times d \times 3}$



5×5 **filters** w_m

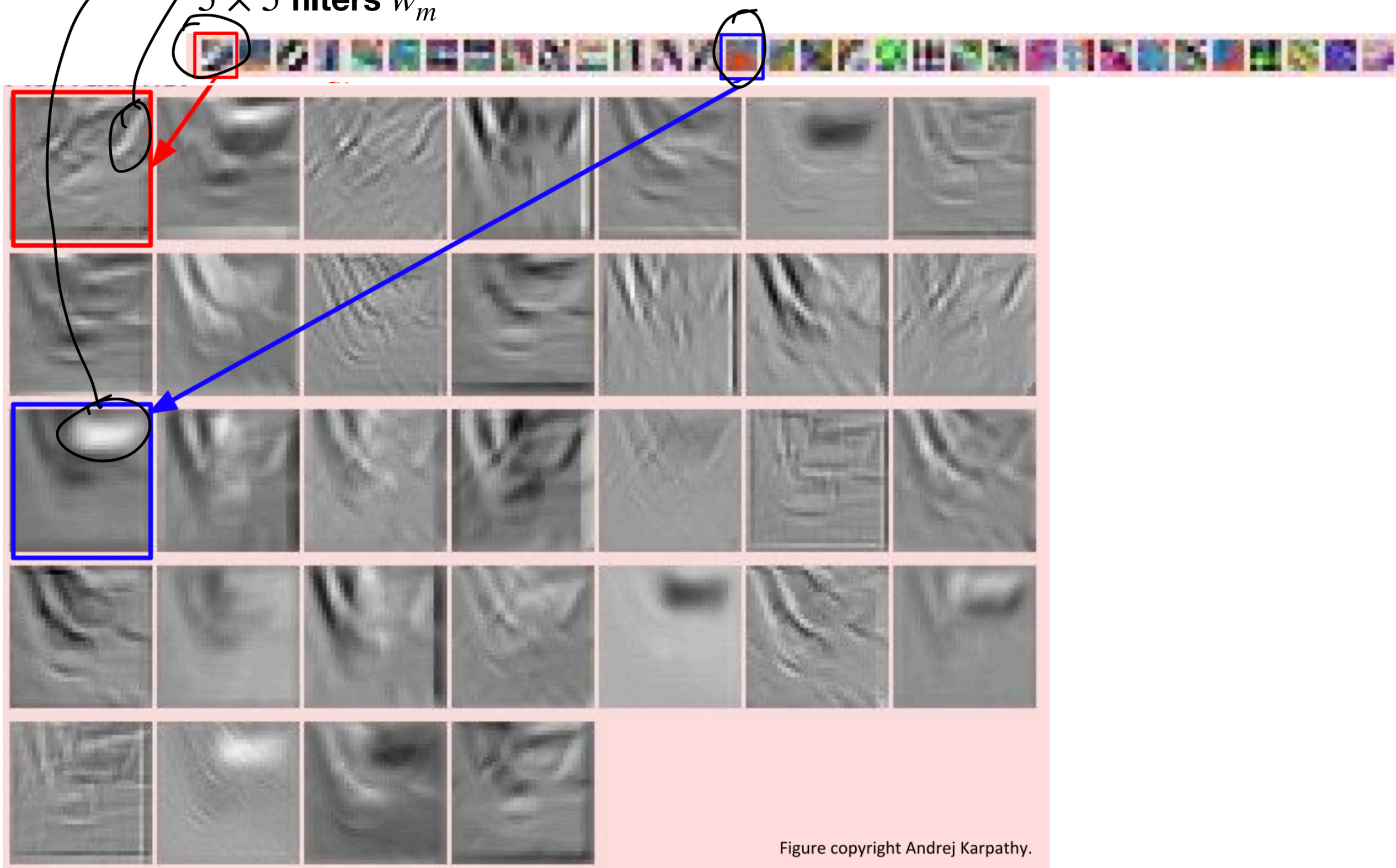
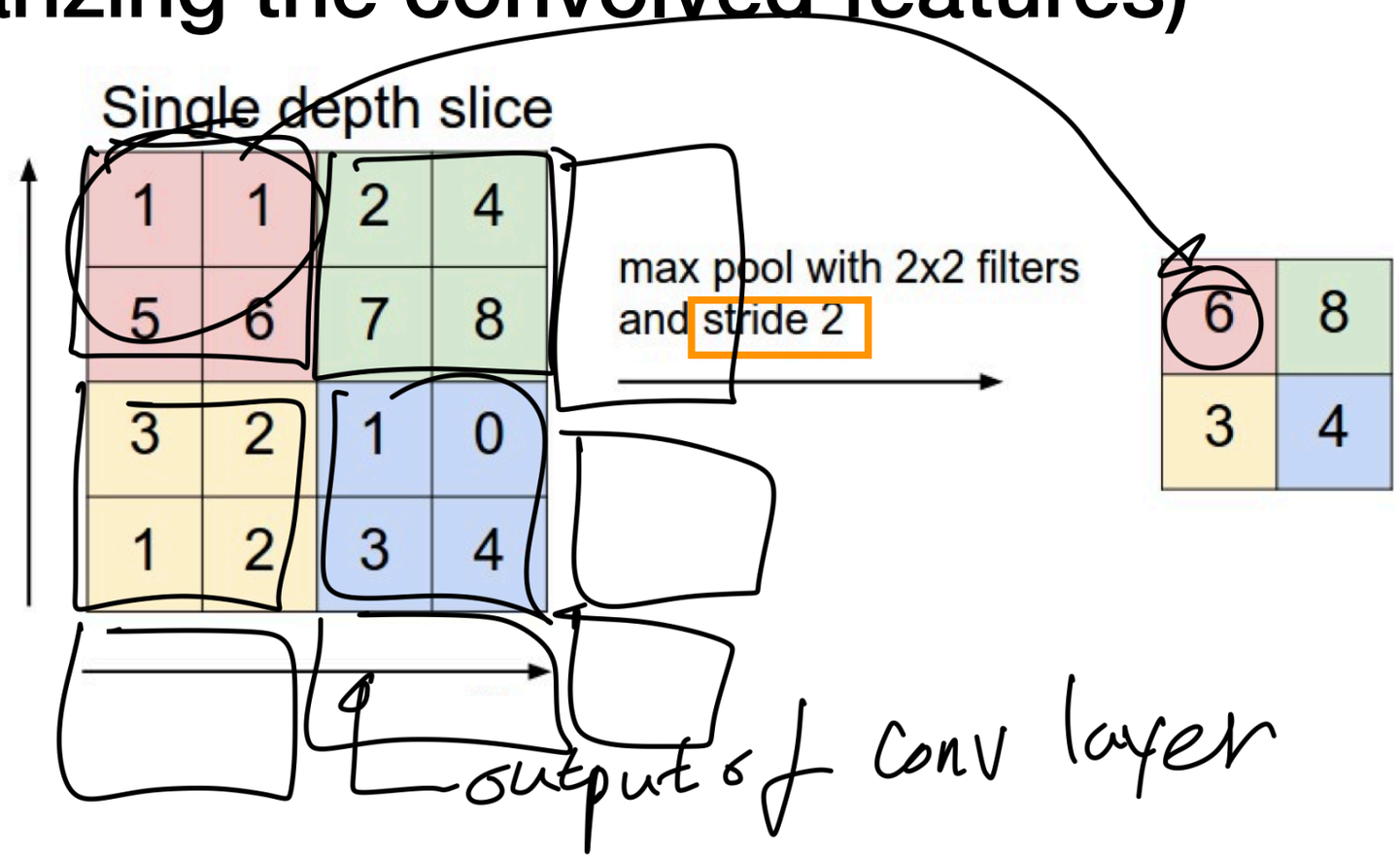
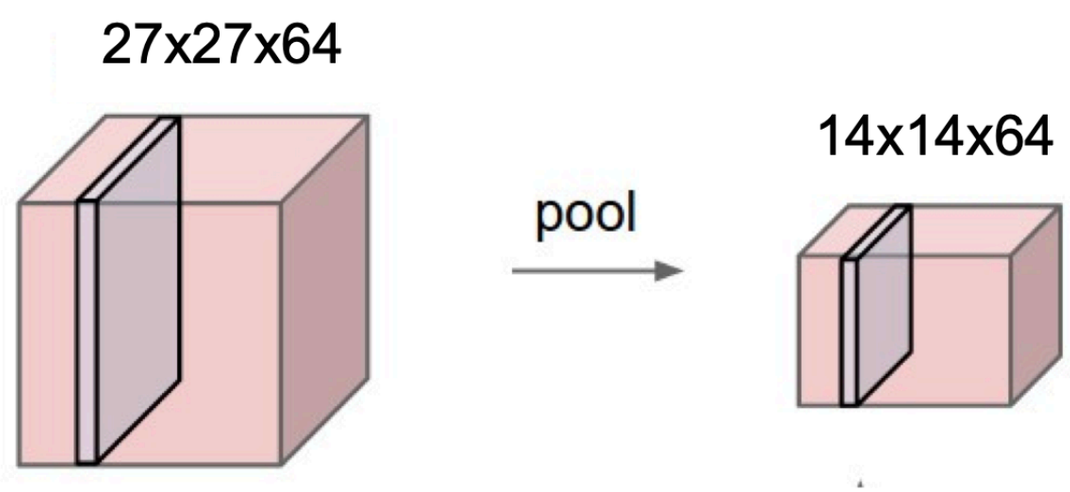


Figure copyright Andrej Karpathy.

Pooling (locally summarizing the convolved features)

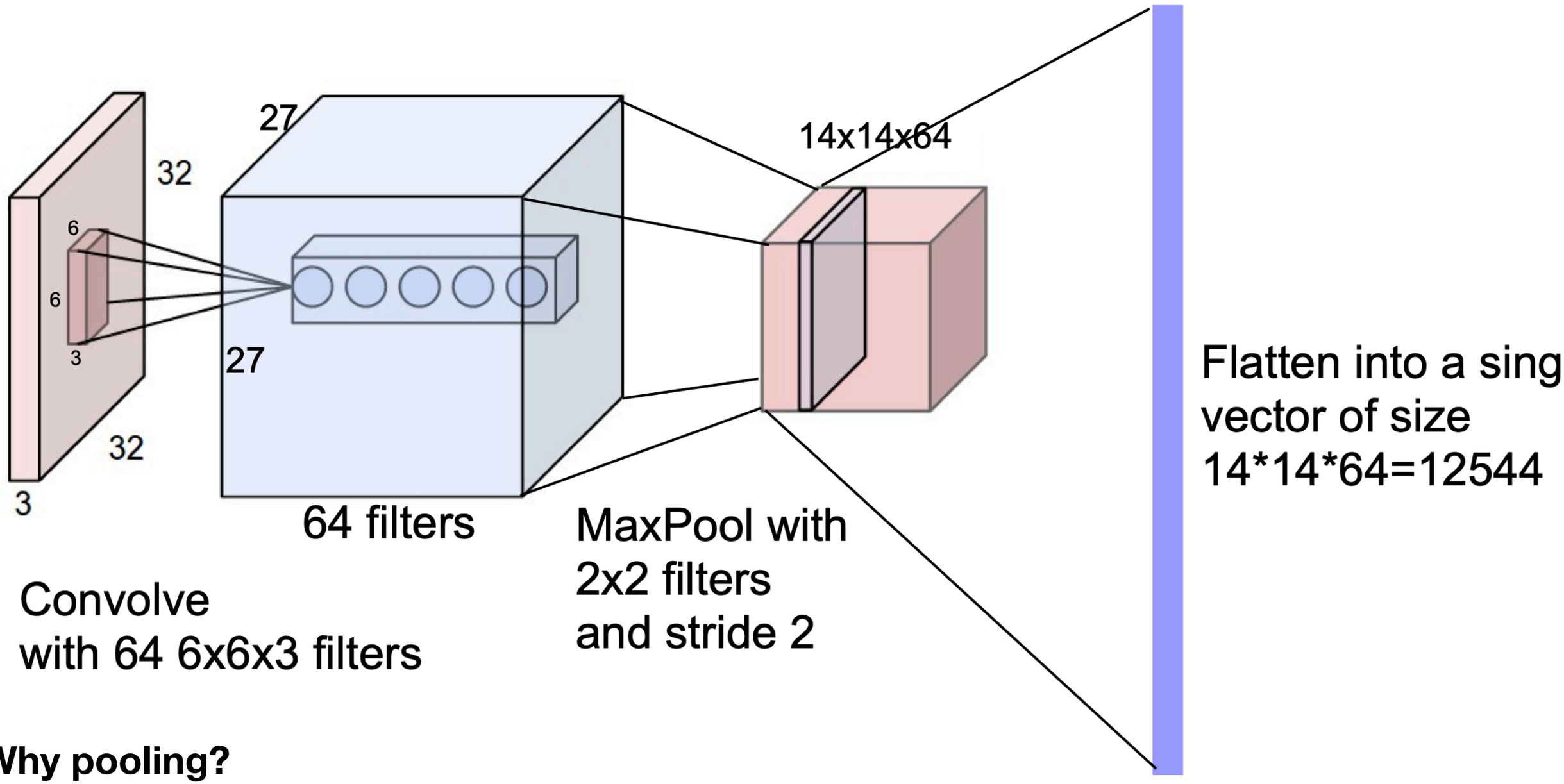
Pooling reduces the dimension and can be interpreted as “This filter had a high response in this general region”





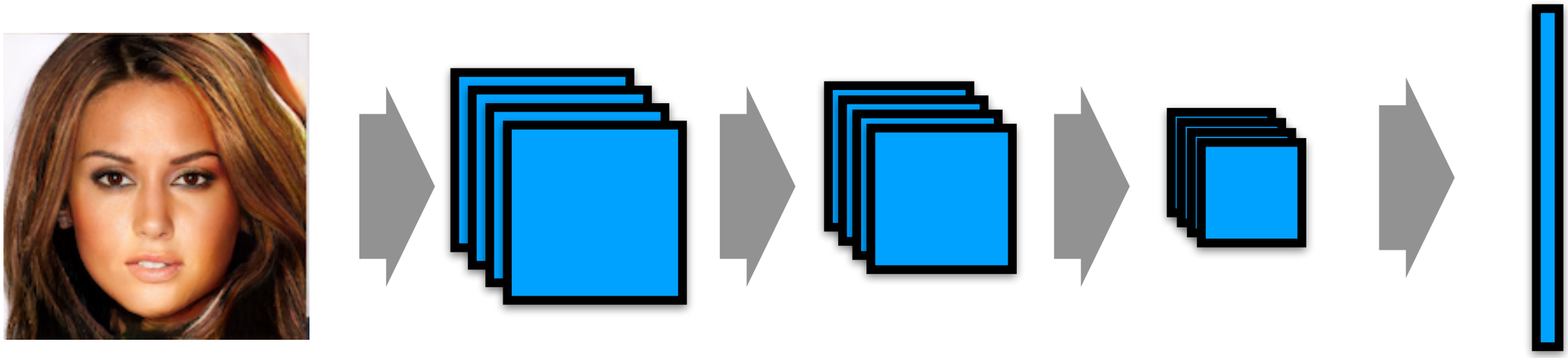
- other functionals include, max pooling, average pooling, and L2-norm pooling
- it should be a function independent of the input permutation (to impose shift invariance)

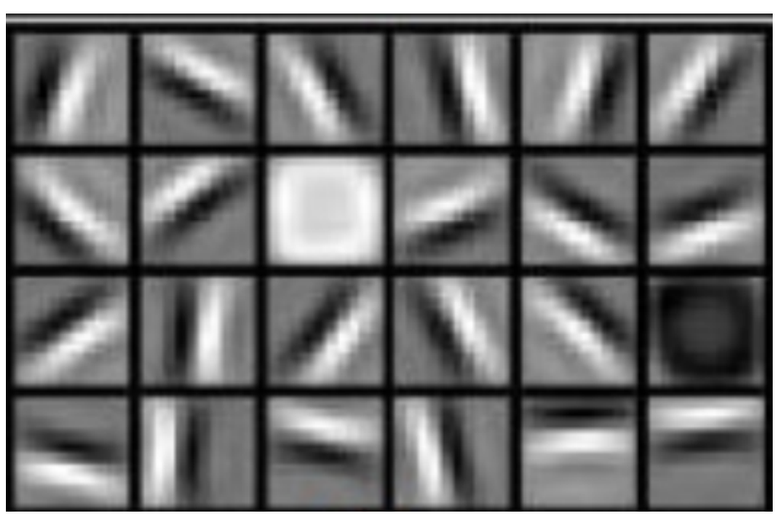
A single convolutional + pooling layer as feature extraction

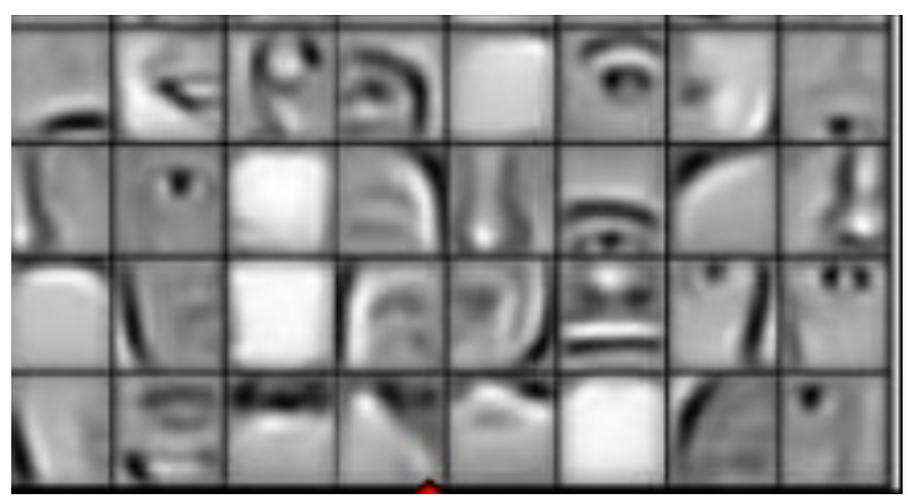


Why pooling?
- if we used pooling with a larger stride, then the output feature is (almost) invariant to small shifts

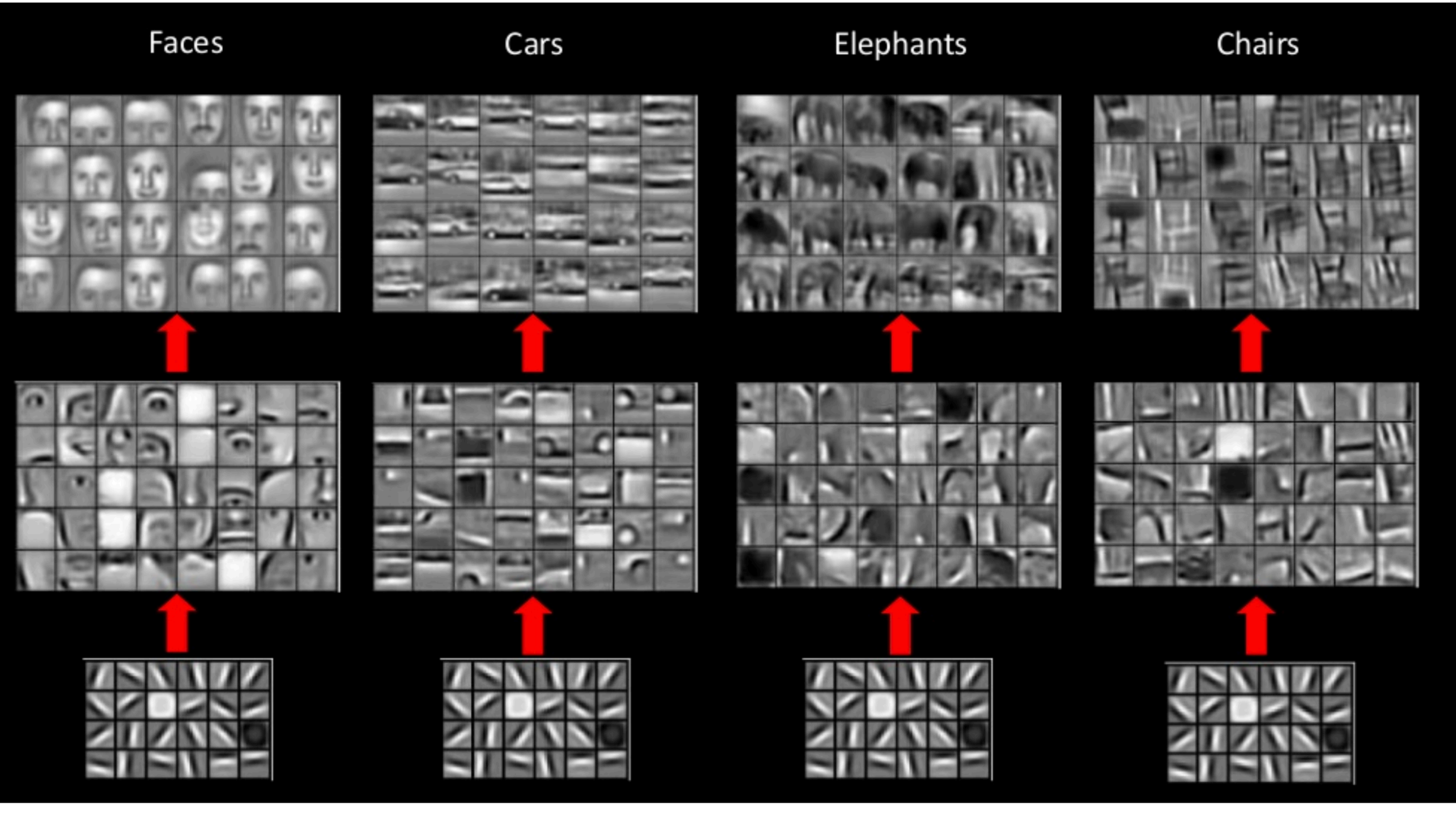
Hidden convolutional layers learn patterns of increasing complexity





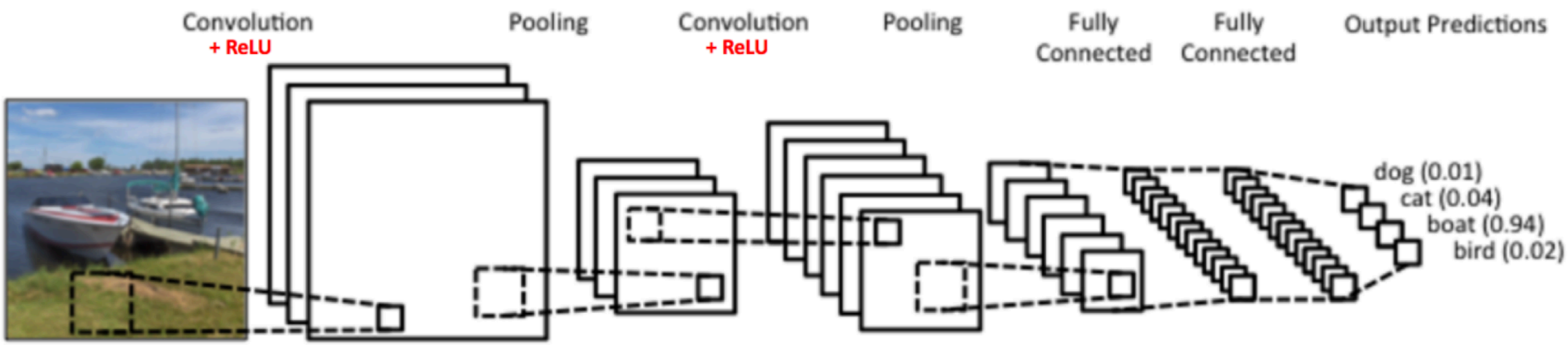


Deep neural networks learns non-linear features



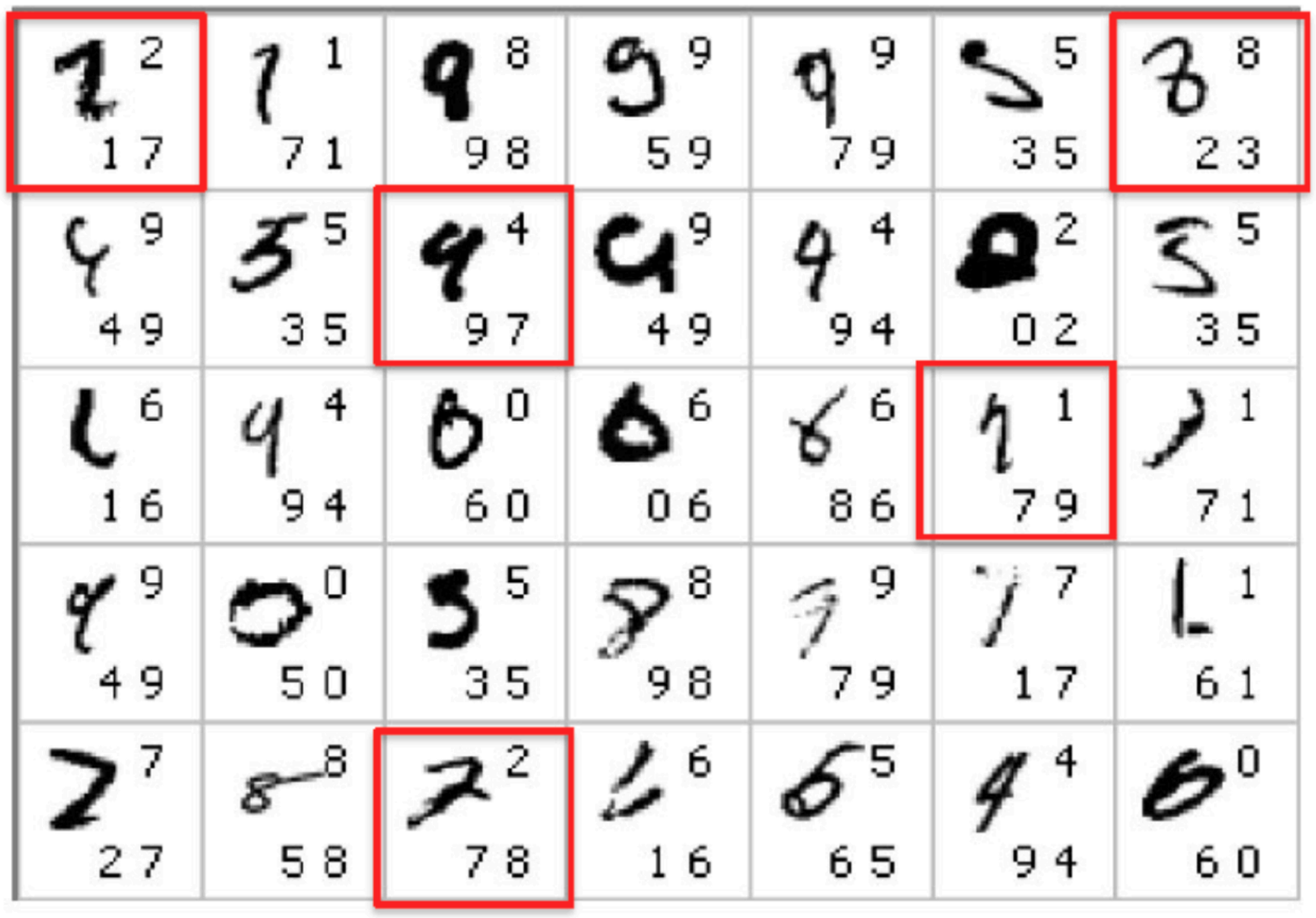
Convolutional neural network (LeNet 1990's)

- A convolutional neural network consists of multiple convolution, pooling, and fully connected layers



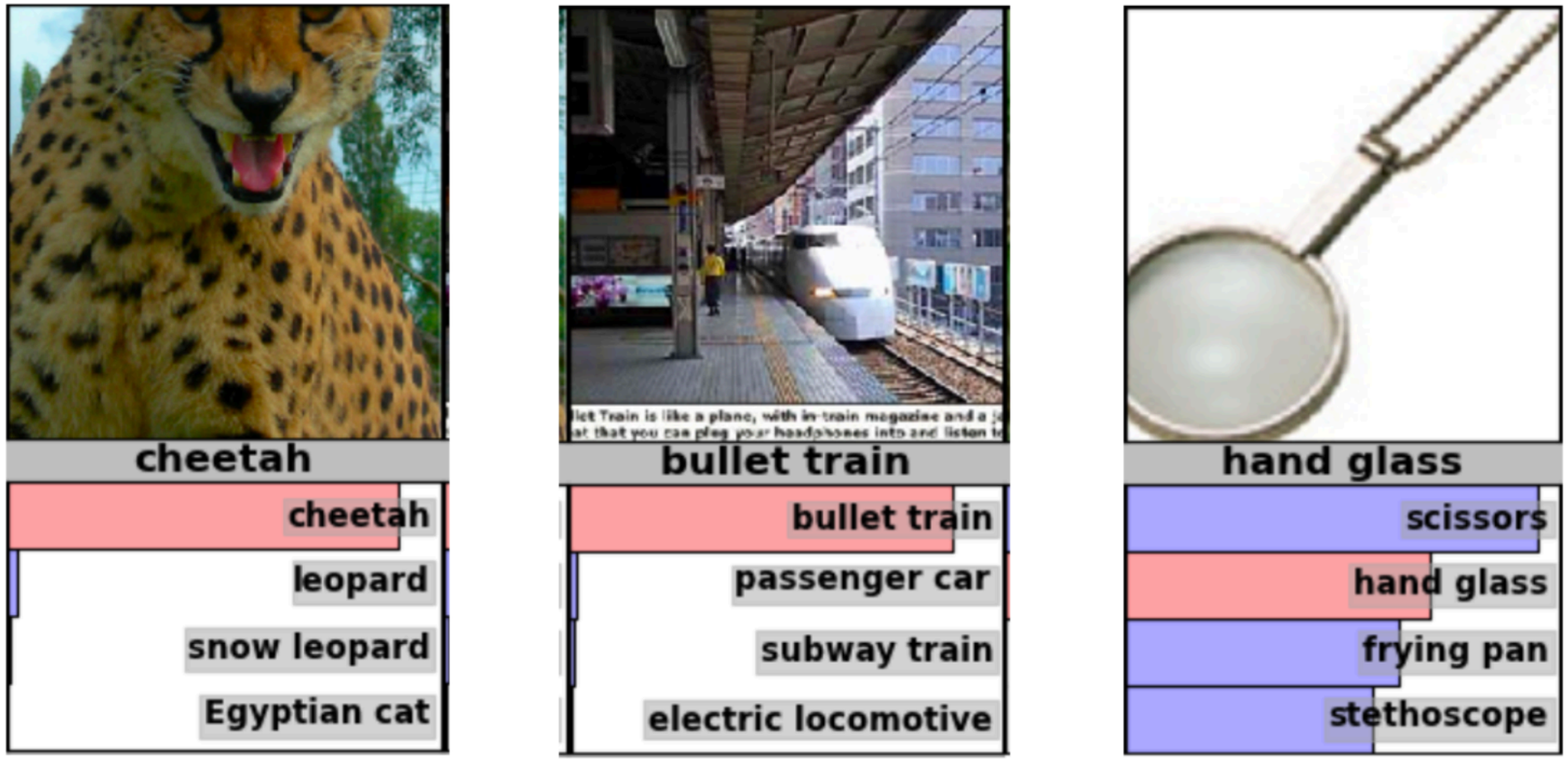
- 82 error made by LeNet

- 35 error made by Ciresan et al.
- further, most of the time the true answer is in the top-2 prediction
- idea: train with transformed samples ← Data Augmentation



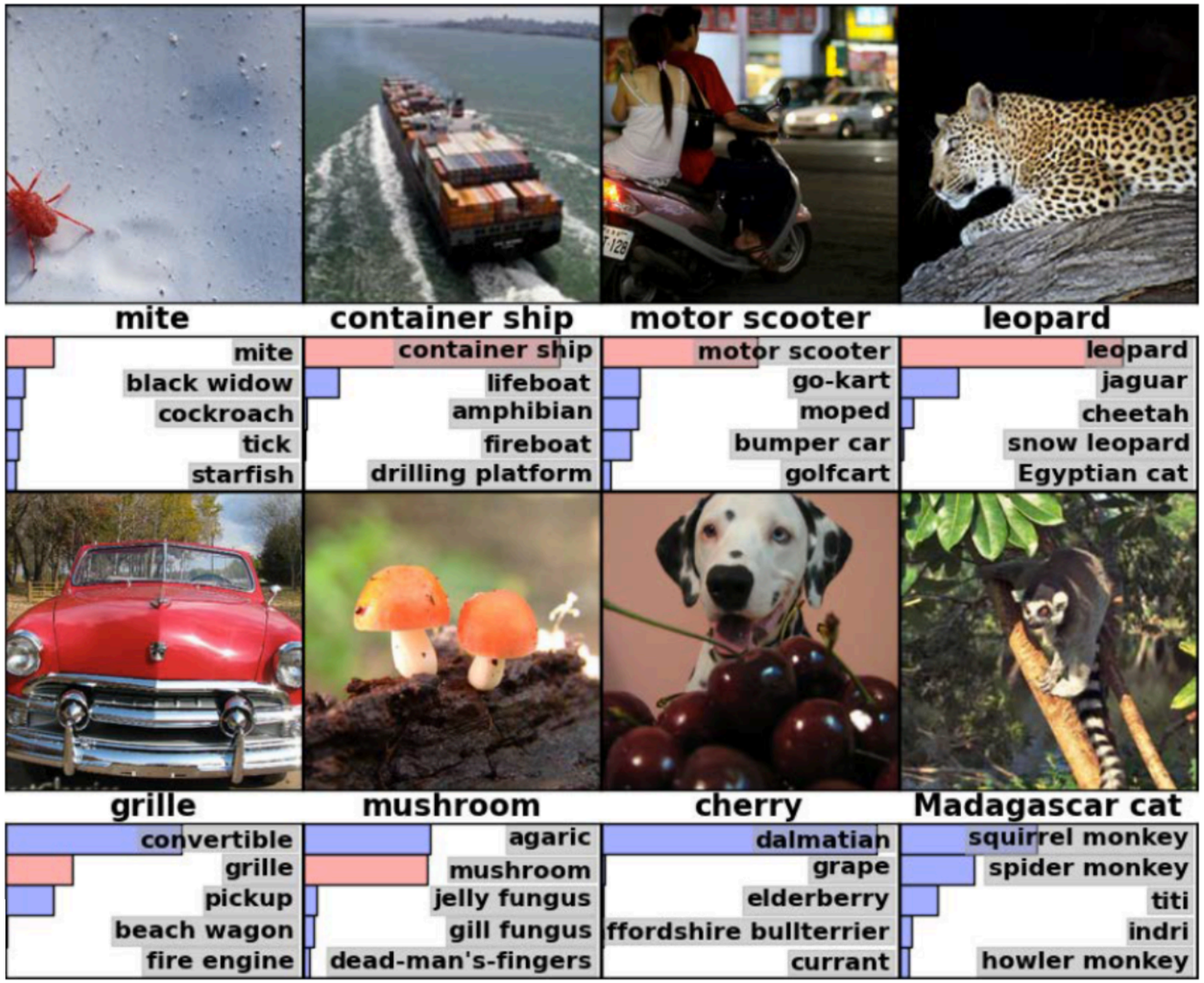
ILSVRC-2012 challenge on ImageNet

- 28×28 grey-scale to 256×256 color
- 10 classes to 1,000 classes
- multiple objects
- natural 3-d scene



winner: AlexNet

- Alex Krizhevsky, Ilya Sutskever and Geoff Hinton, 2012
- mirror image
- subsampling to get 224×224 patches from 256×256 images
- ReLU activation is faster to train and more expressive
- Dropout to regularize



Why are convolutional neural networks so successful?

- Convolutional neural network imposes **translation invariance** (via **pooling** and **weight sharing**)

- and significantly reduces the number of parameters (by **weight sharing**)

- Structured sparse connections capture **locality** of images

- these are main reasons for the success of deep learning for computer vision
- which is central to the popularity of deep learning

- Feed-forward neural network would not scale to images
  - CIFAR-10 images are only 32*32*3 and a single neuron at the first layer will have 32*32*3 = 3072 weights

  - this gets worse for practical size images
  - Convolution allows us to extract many relevant features, with small number of parameters

demo: https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html