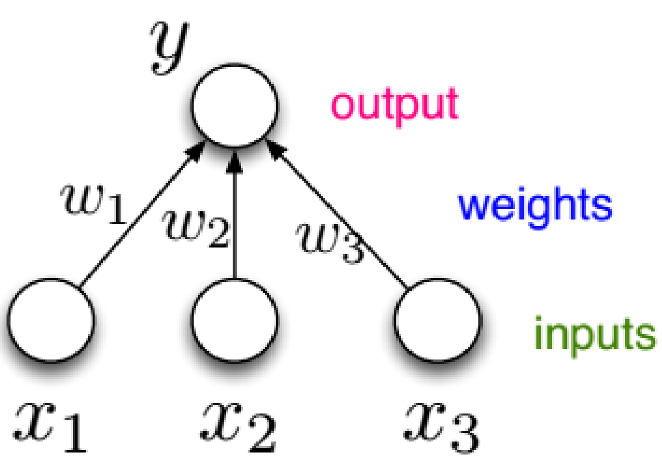
Multi layer perceptrons

Dr Suresh Sundaram

Single layer perceptron

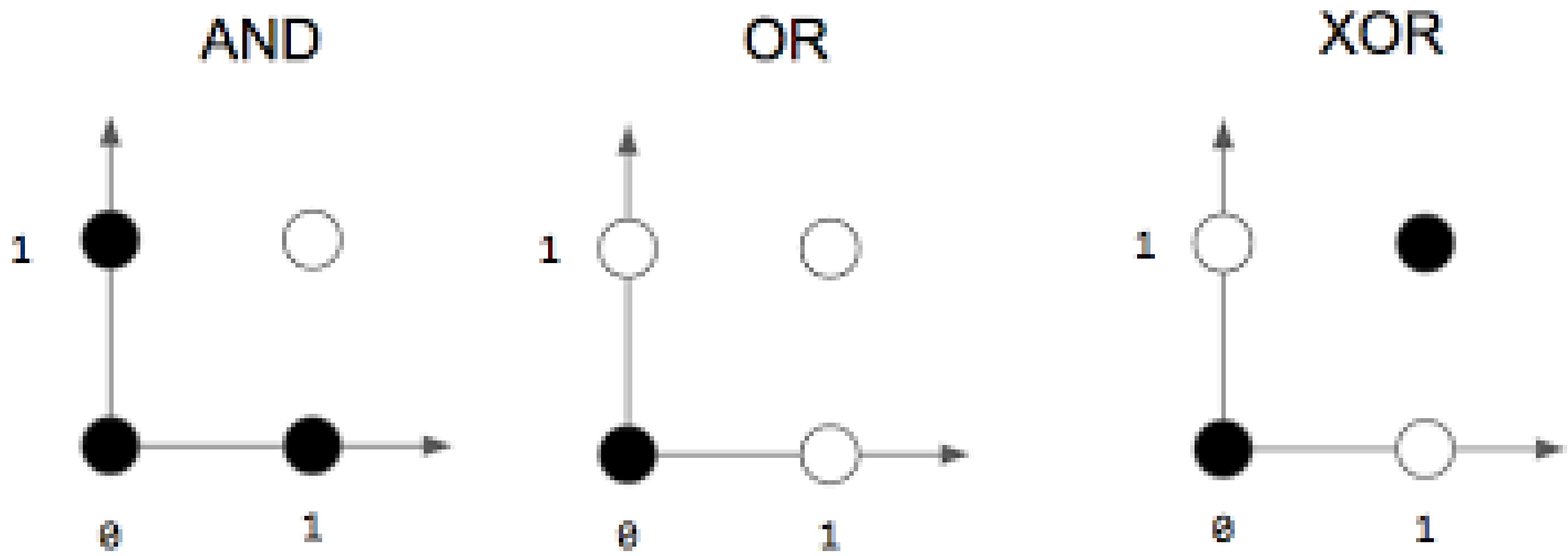


output bias i'th weight

$$y = g\left(b + \sum_i x_i w_i\right)$$

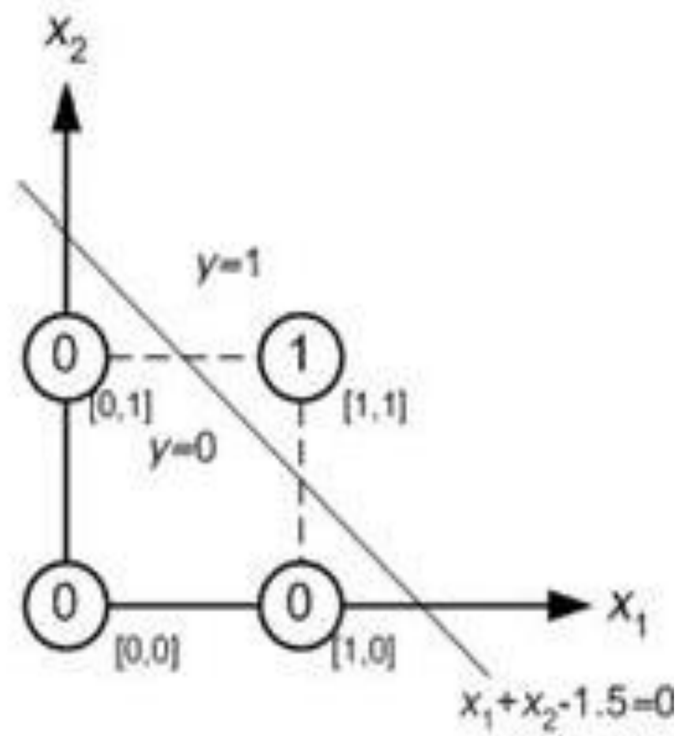
nonlinearity i'th input

Limitations of single layer perceptron

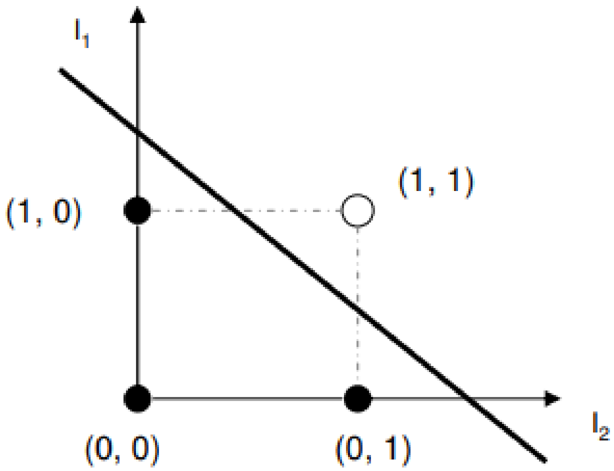


AND gate

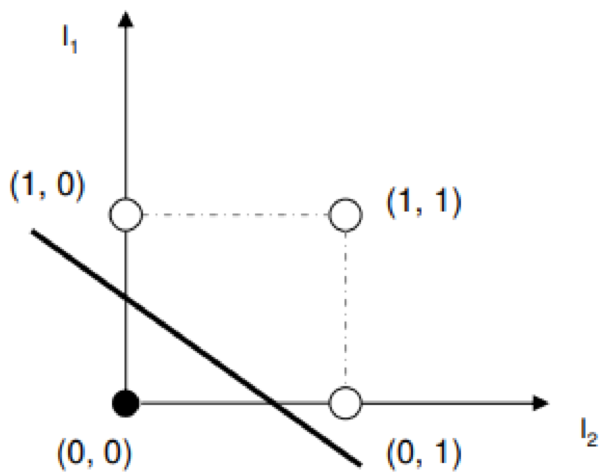
x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



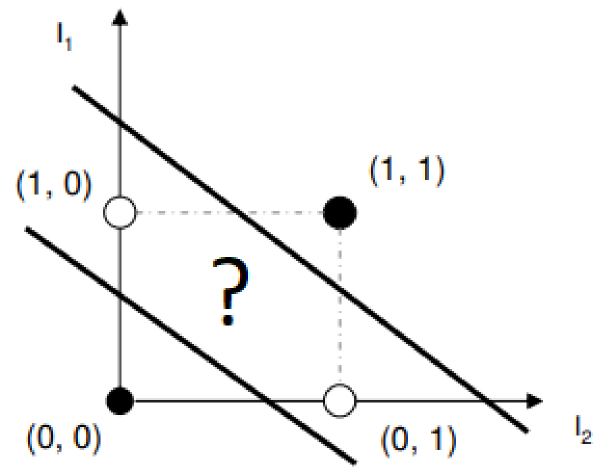
AND		
I_1	I_2	out
0	0	0
0	1	0
1	0	0
1	1	1



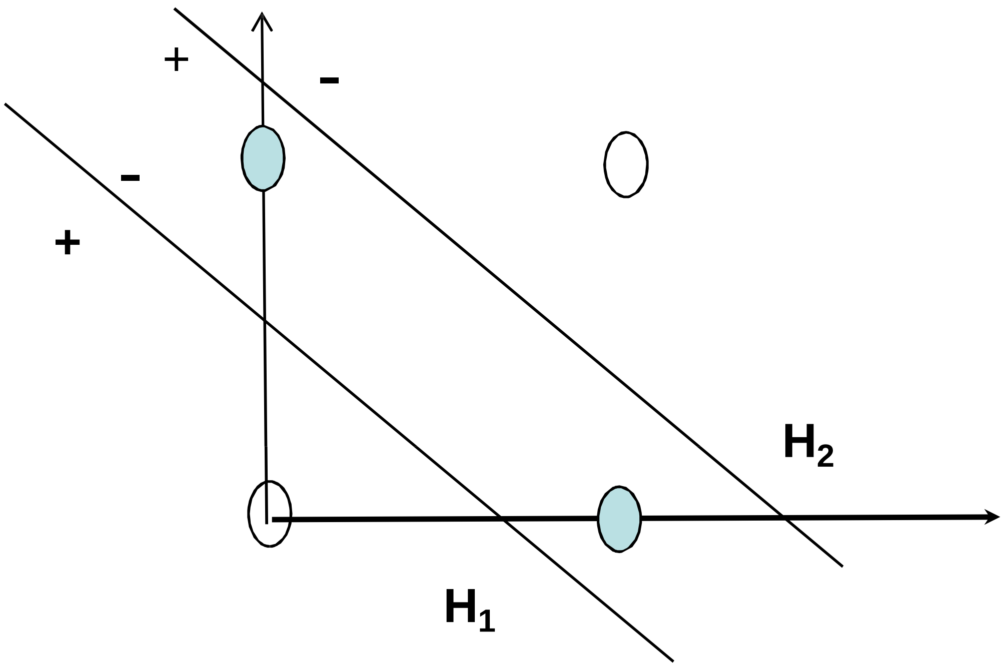
OR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	1



XOR		
I_1	I_2	out
0	0	0
0	1	1
1	0	1
1	1	0

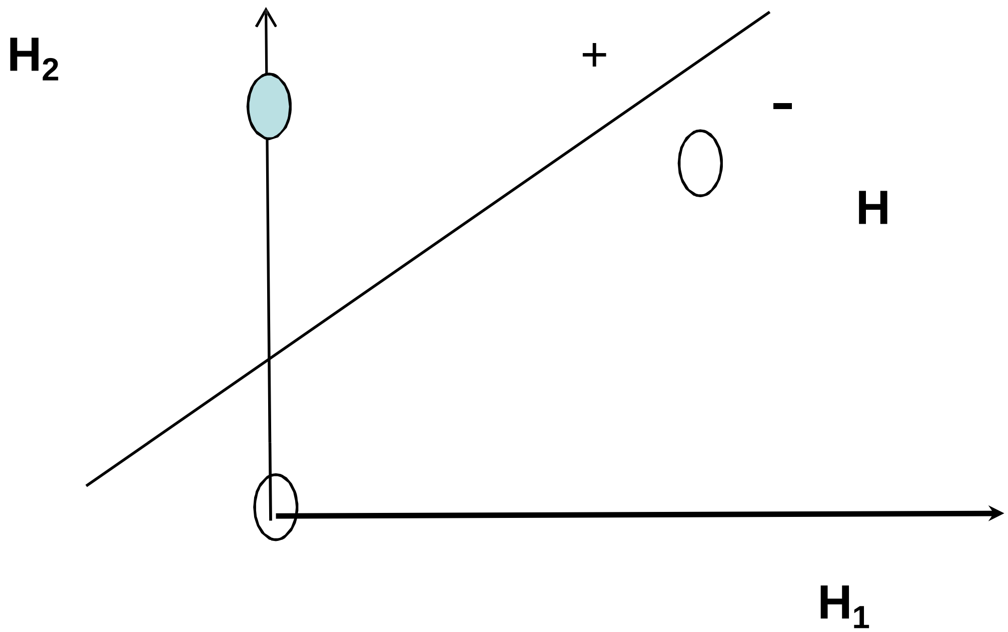


Need of multi layer perceptron : XOR Gate

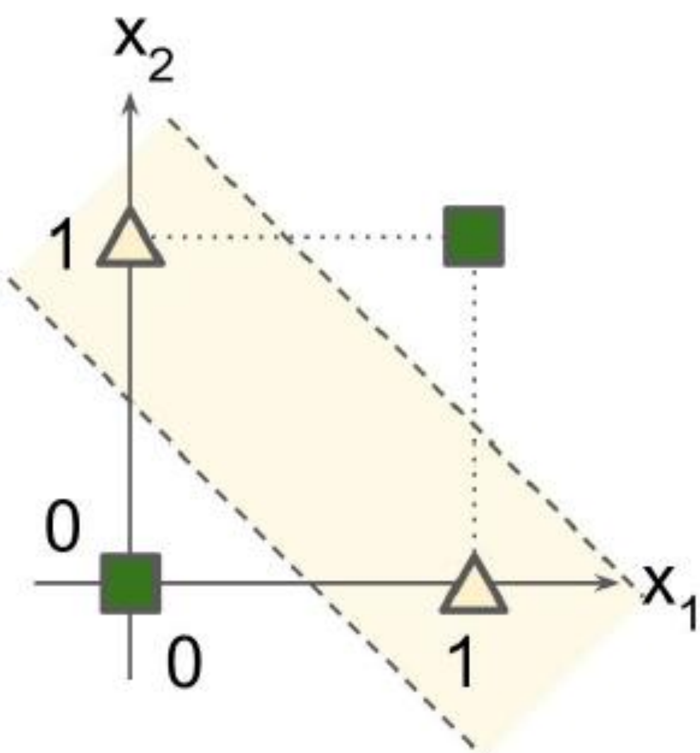


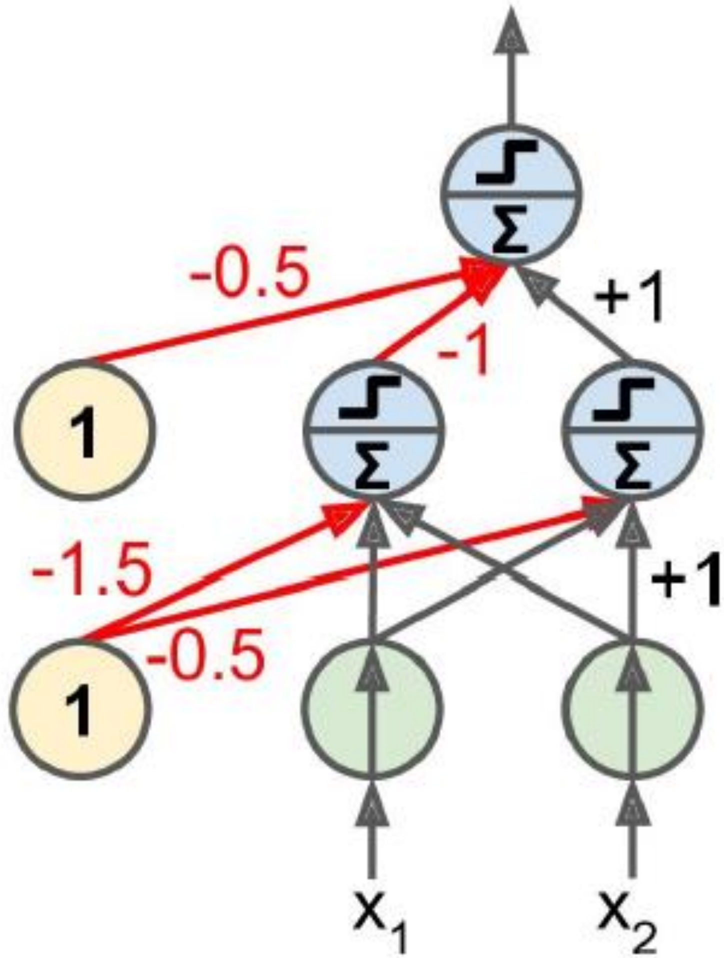
A	B	O	H_1	H_2
0	0	0	1	1
0	1	1	0	1
1	0	1	0	1
1	1	0	0	0

Need of multi layer perceptron : XOR Gate



H1	H2	H
1	1	0
0	1	1
0 0	1 0	1 0


x_2
1
0
0
1
x_1


-0.5
+1
-1
1
-1.5
+1
-0.5
1
x_1
x_2

T

- One of the most popular methods for training such multilayer networks is based on gradient descent in error — the back propagation algorithm (or generalized delta rule), a natural extension of the perceptron algorithm.

–

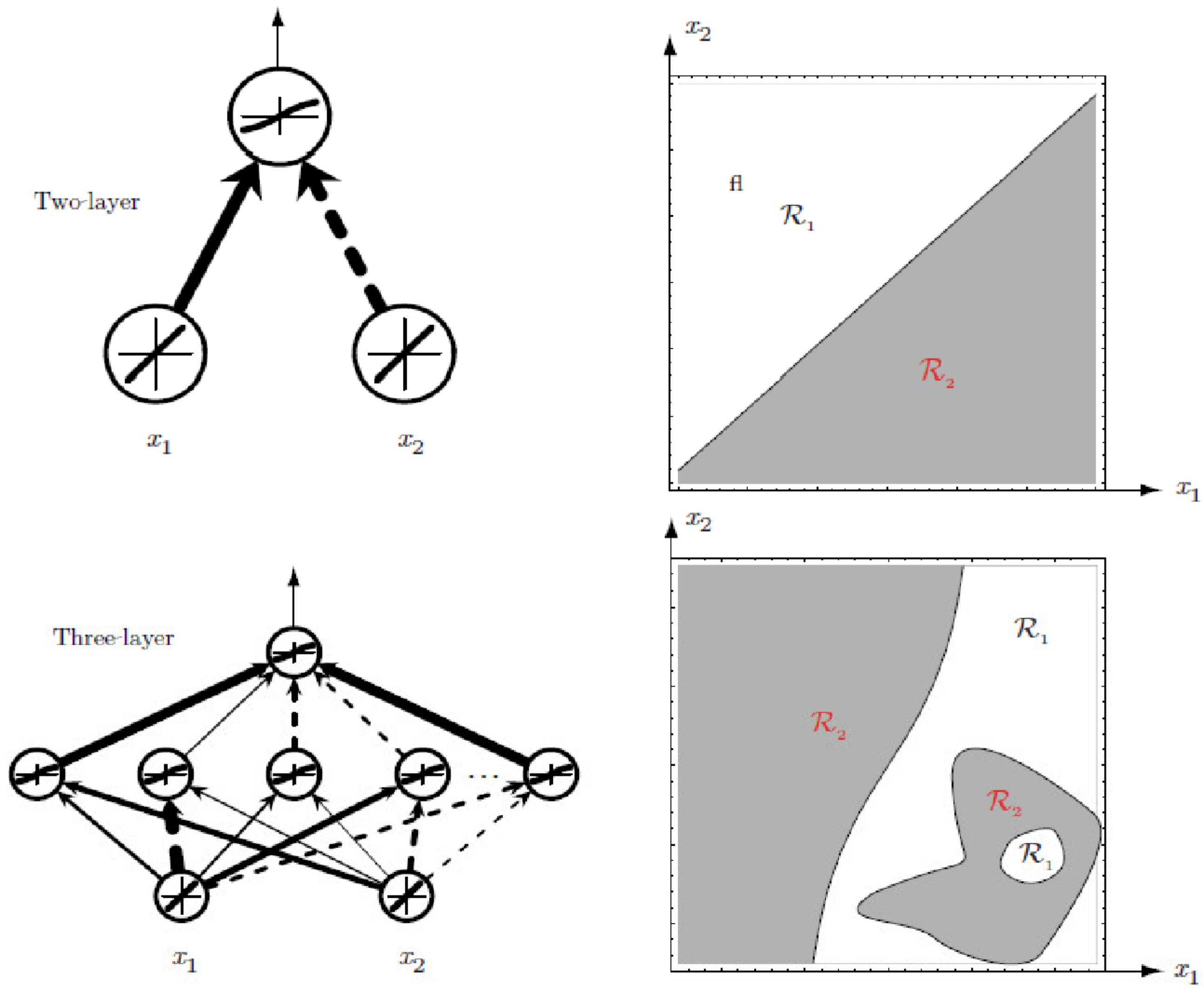
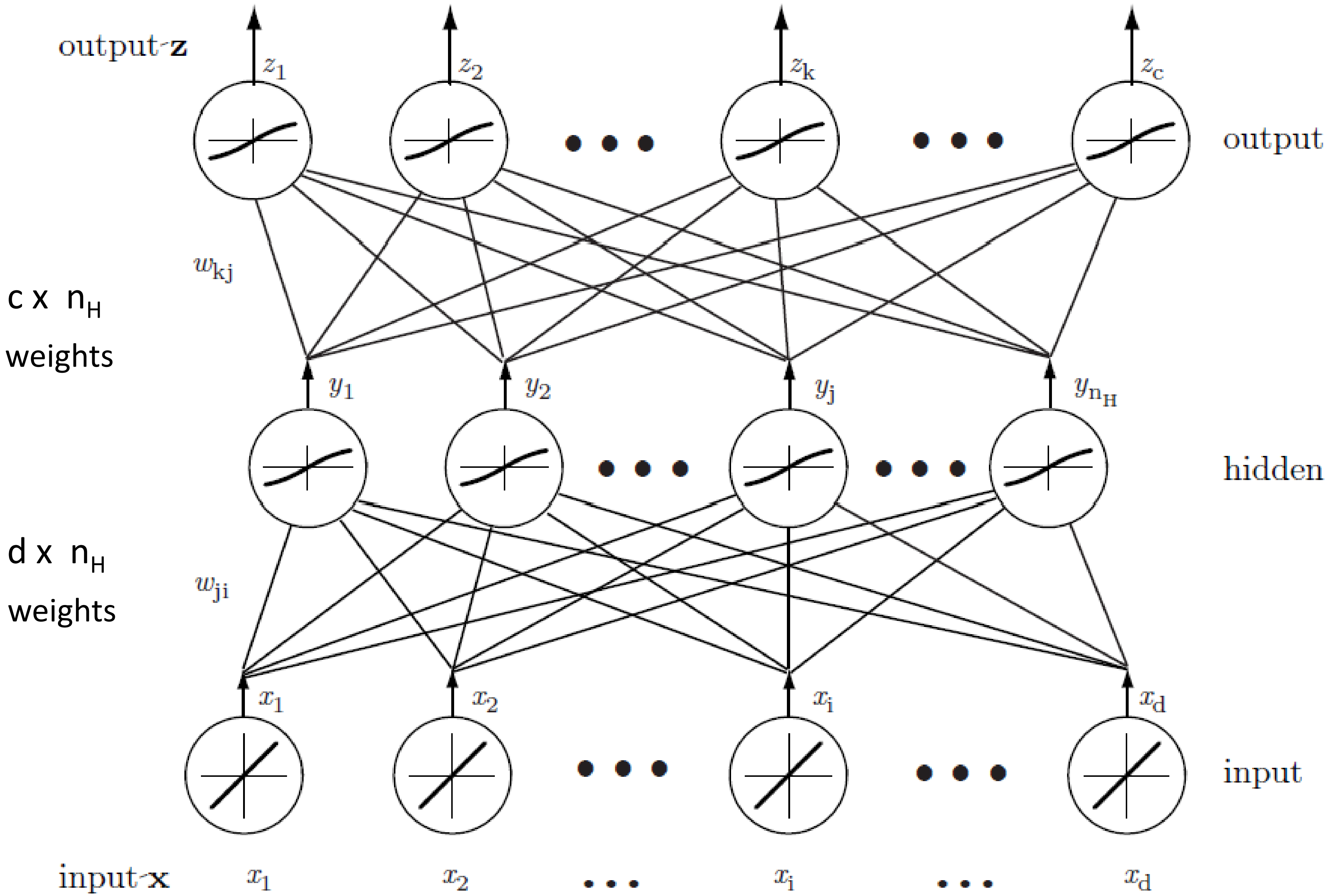


Figure 6.3: Whereas a two-layer network classifier can only implement a linear decision boundary, given an adequate number of hidden units, three-, four- and higher-layer networks can implement arbitrary decision boundaries. The decision regions need not be convex, nor simply connected.

Multi layer perceptron

- Simple three-layer neural network.

- This one consists of an input layer (having *d* input units), a hidden layer with (n_H hidden units) and an output layer (a single unit), interconnected by modifiable weights, represented by links between layers.



output z
z_1
z_2
z_k
z_c
output
w_{kj}
c x n_H
weights
y_1
y_2
y_j
y_{n_H}
hidden
d x n_H
weights
w_{ji}
x_1
x_2
x_i
x_d
input
input x
x_1
x_2
x_i
x_d

Multi layer perceptron

Net Output at input to j th hidden node

$$net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} \equiv \mathbf{w}_j^t \mathbf{x},$$

Non linearity imposed by activation function *f*(…)

$$y_j = f(net_j).$$

$$f(net) = Sgn(net) \equiv \begin{cases} 1 & \text{if } net \geq 0 \\ -1 & \text{if } net < 0, \end{cases}$$

This $f()$ is sometimes called the *transfer function* or merely "nonlinearity" of a unit,

Multi layer perceptron

Each output unit similarly computes its net activation based on the hidden unit signals as

$$net_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = \mathbf{w}_k^t \mathbf{y},$$

where the subscript k indexes units in the output layer

n_H denotes the number of hidden units

Multi layer perceptron

Each output unit then computes the nonlinear function of its net, emitting

$$z_k = f(net_k).$$

$$g_k(\mathbf{x}) \equiv z_k = f\left(\sum_{j=1}^{n_H} w_{kj}\ f\left(\sum_{i=1}^{d} w_{ji}x_i + w_{j0}\right) + w_{k0}\right).$$

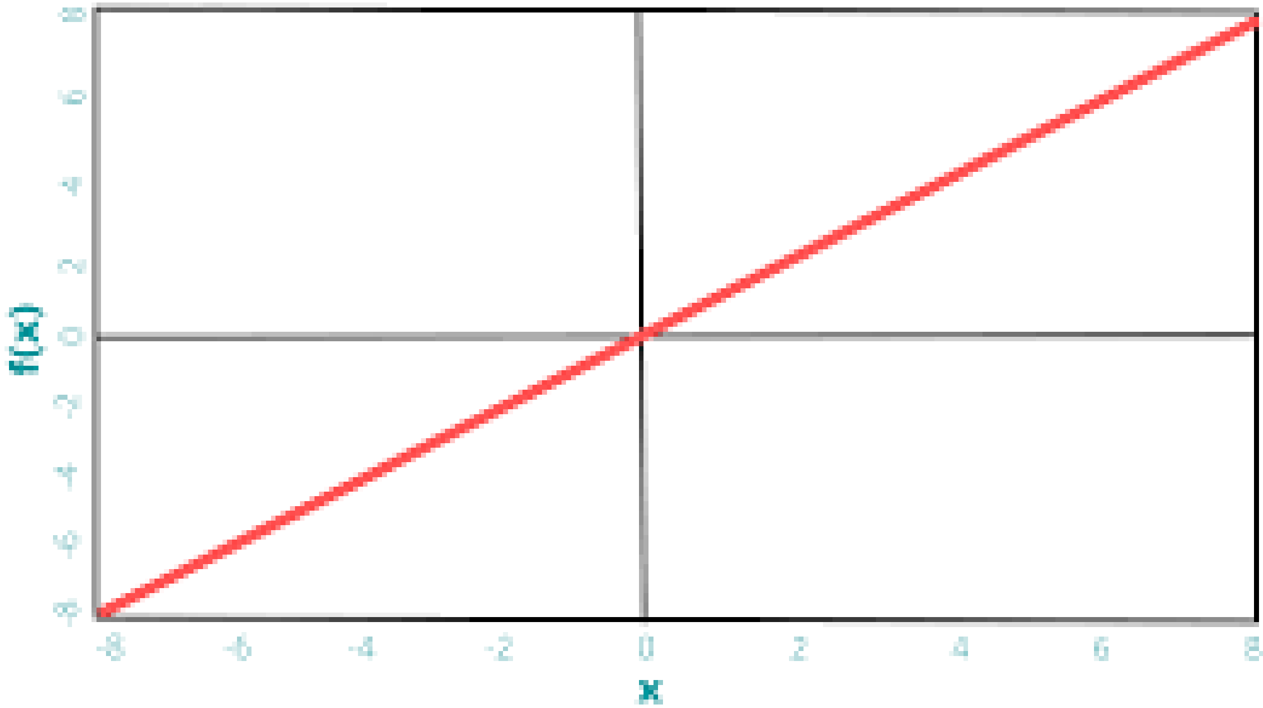
Multi layer perceptron

$$J(\mathbf{w}) \equiv 1/2 \sum_{k=1}^{c} (t_k - z_k)^2 = 1/2(\mathbf{t} - \mathbf{z})^2,$$

where $\mathbf{t}$ and $\mathbf{z}$ are the target and the network output vectors of length c; $\mathbf{w}$ represents all the weights in the network.

The backpropagation learning rule is based on gradient descent. The weights are initialized with random values, and are changed in a direction that will reduce the error:

$$\Delta \mathbf{w} = -\eta \frac{\partial J}{\partial \mathbf{w}},$$

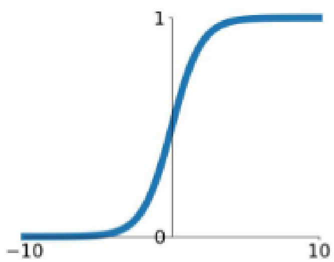


Linear Activation Function

Activation Functions

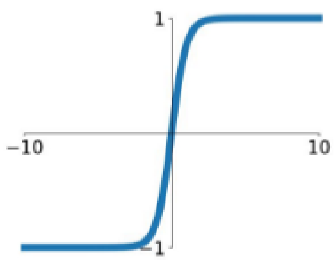
Sigmoid

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



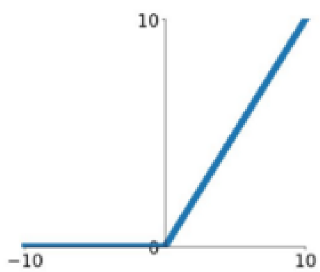
tanh

$\tanh(x)$



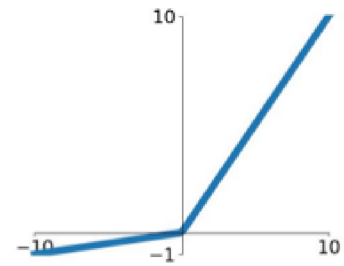
ReLU

$\max(0, x)$



Leaky ReLU

$\max(0.1x, x)$

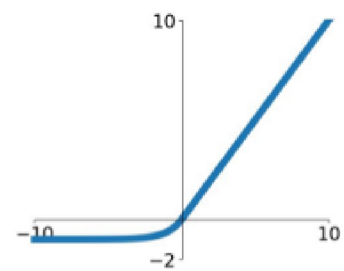


Maxout

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

ELU

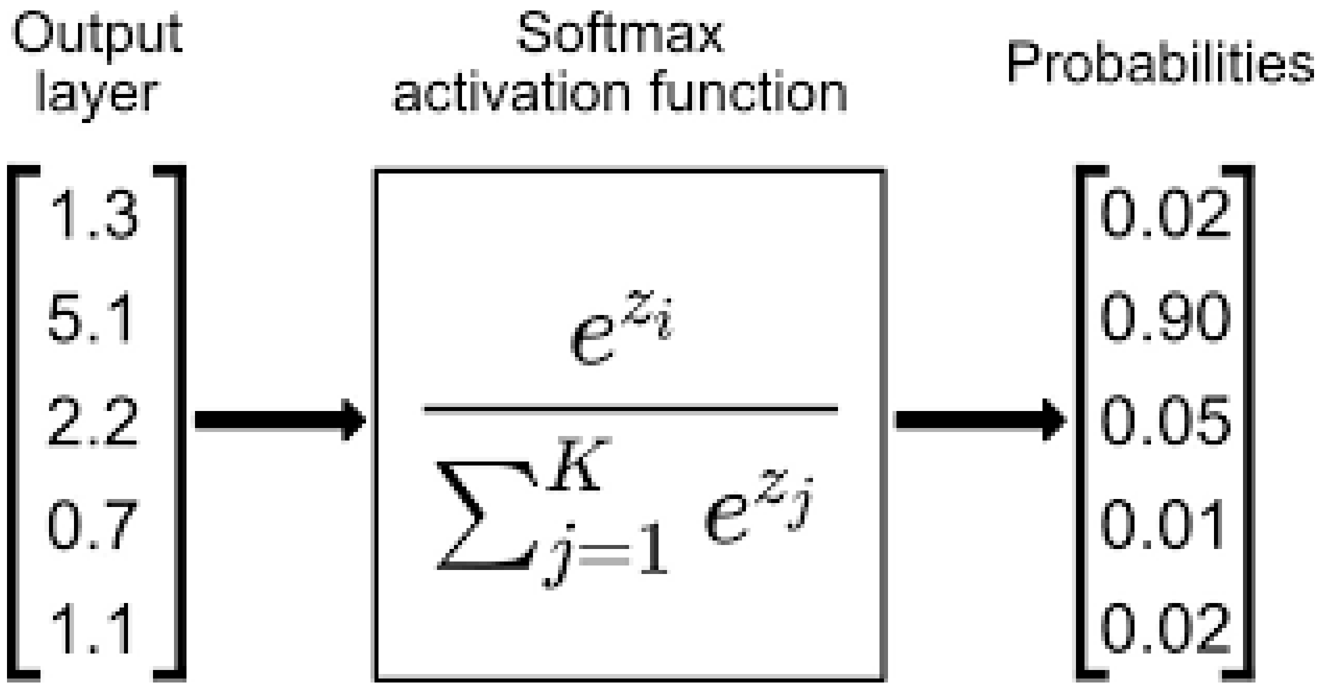
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



id	color
1	red
2	blue
3	green
4	blue

One Hot Encoding

id	color_red	color_blue	color_green
1	1	0	0
2	0	1	0
3	0	0	1
4	0	1	0



Output layer
Softmax activation function
Probabilities
1.3
5.1
2.2
0.7
1.1
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
0.02
0.90
0.05
0.01
0.02

Multi layer perceptron

MEAN SQUARE ERROR LOSS

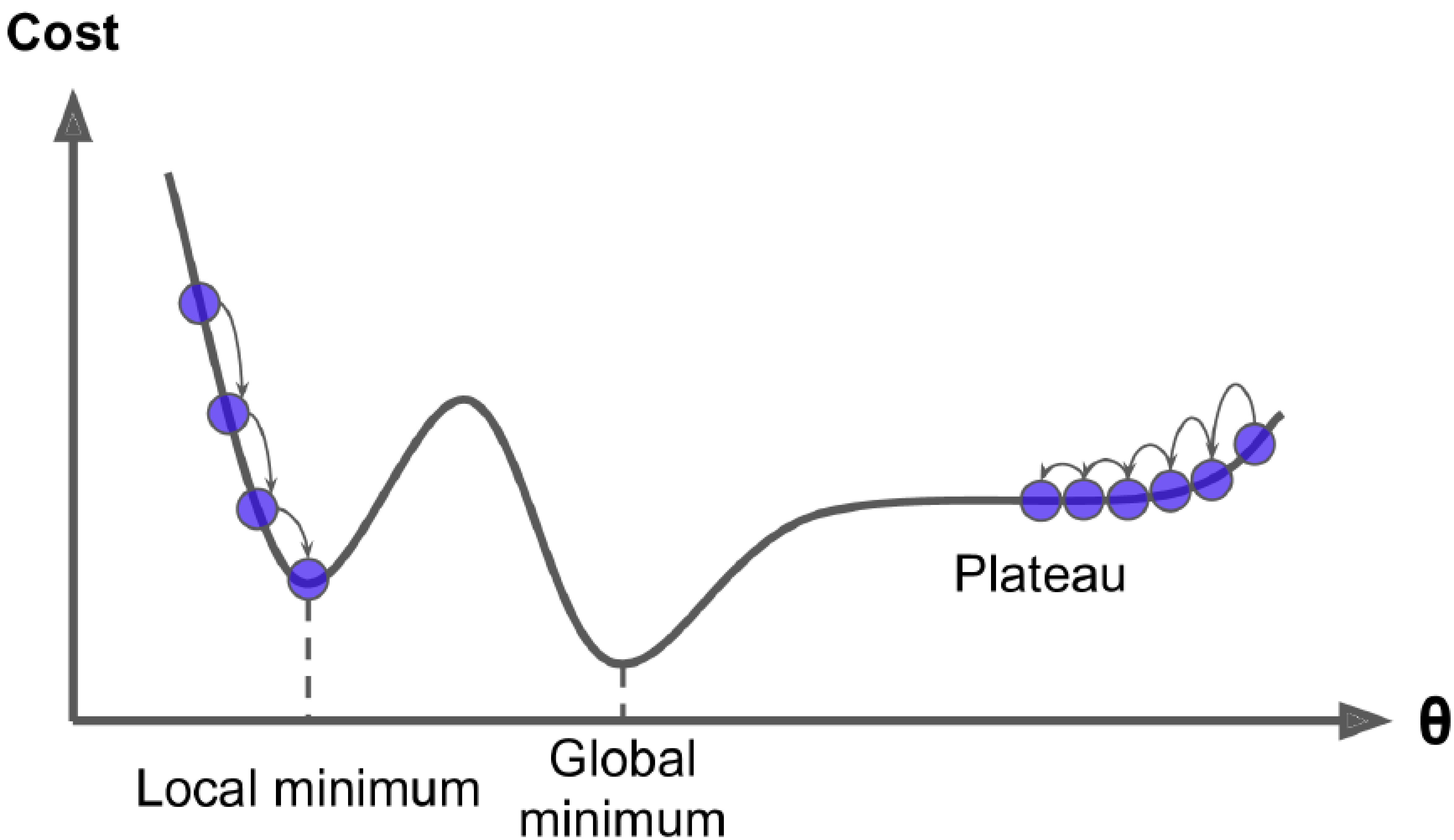
$$J(\mathrm{w}) \equiv 1/2 \sum_{k=1}^{c} (t_k - z_k)^2 = 1/2(\mathbf{t} - \mathbf{z})^2,$$

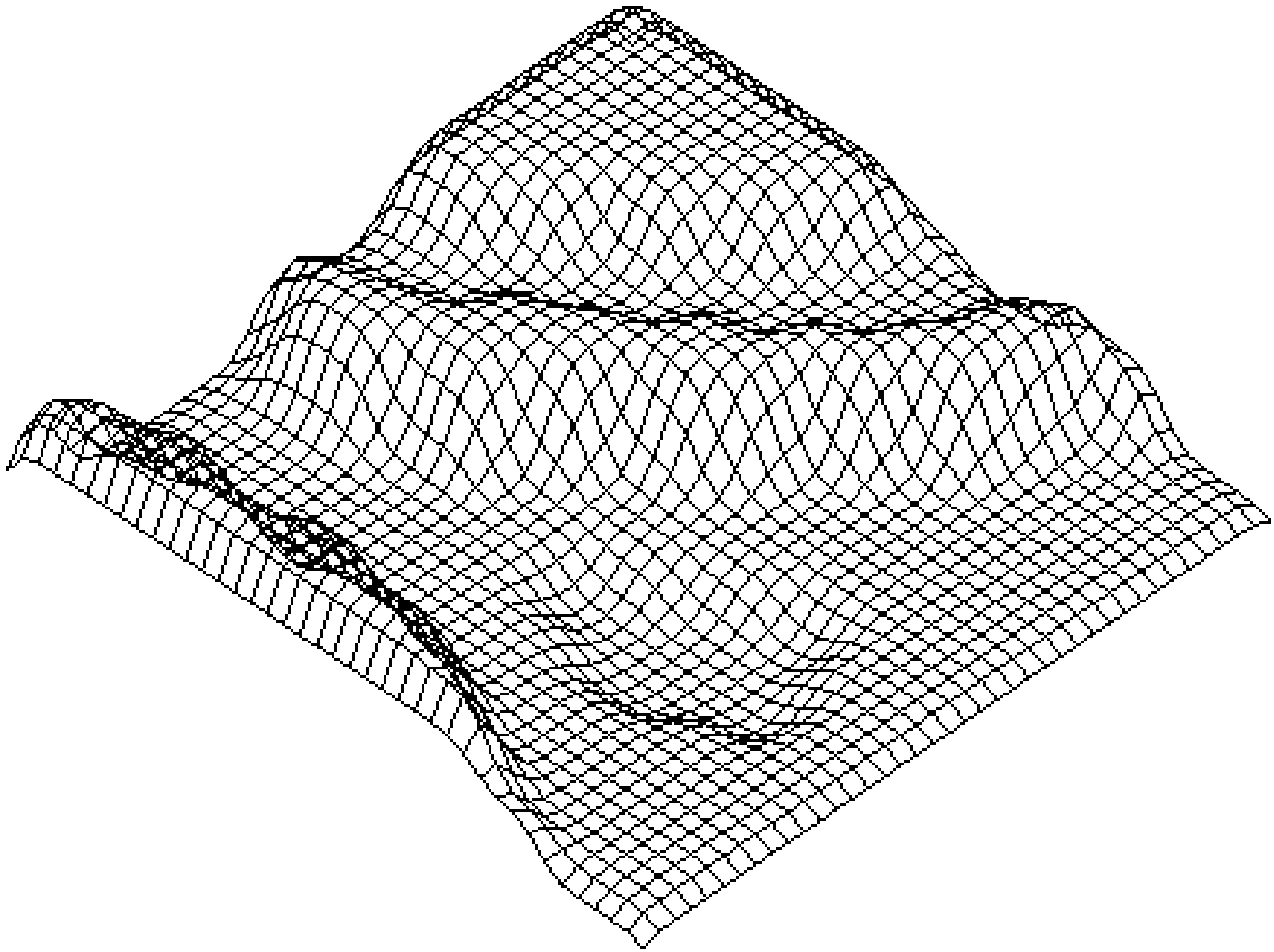
CROSS ENTROPY LOSS

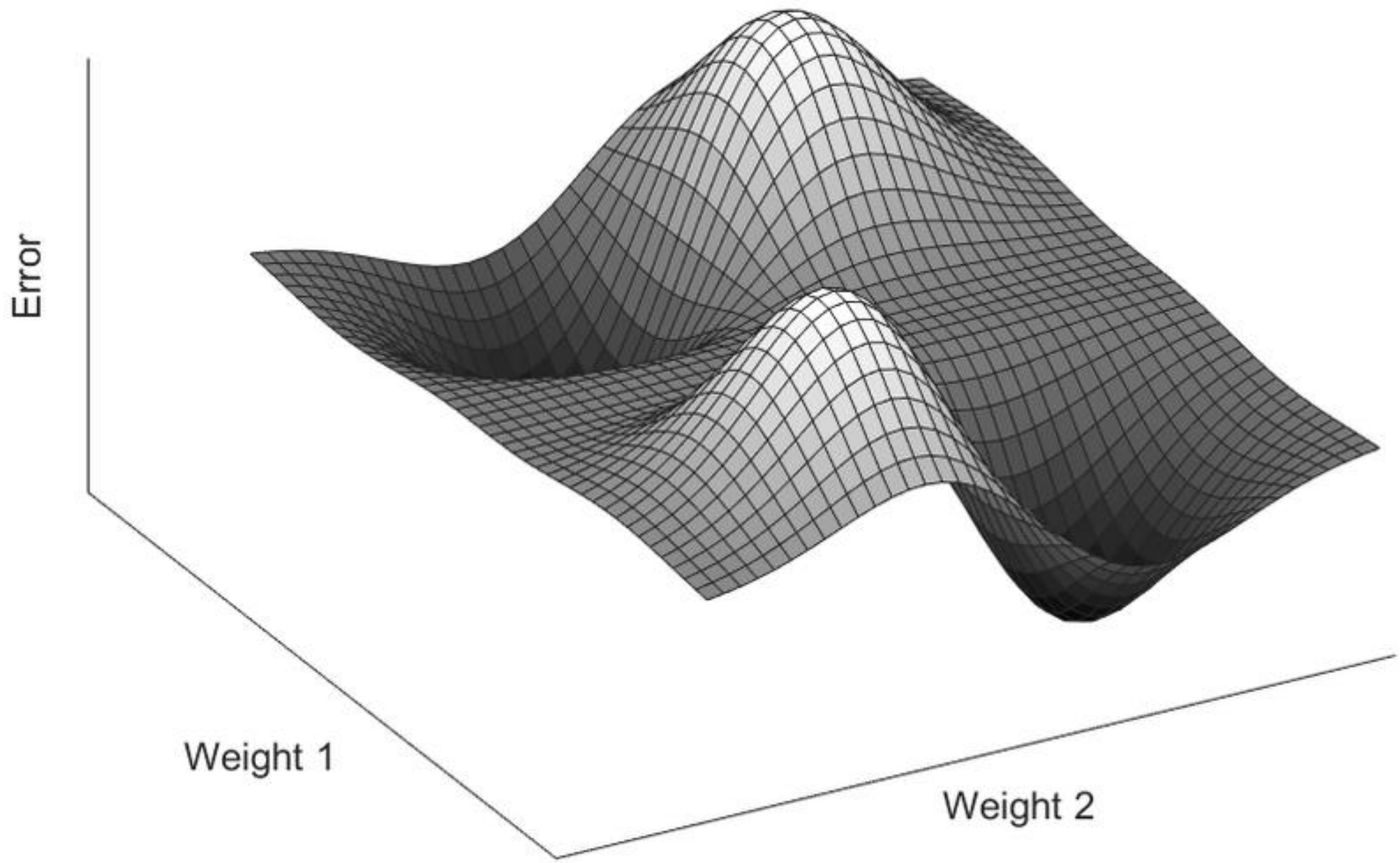
$$L_{\mathrm{CE}} = -\sum_{i=1}^{n} t_i \log(p_i), \quad \text{for n classes},$$

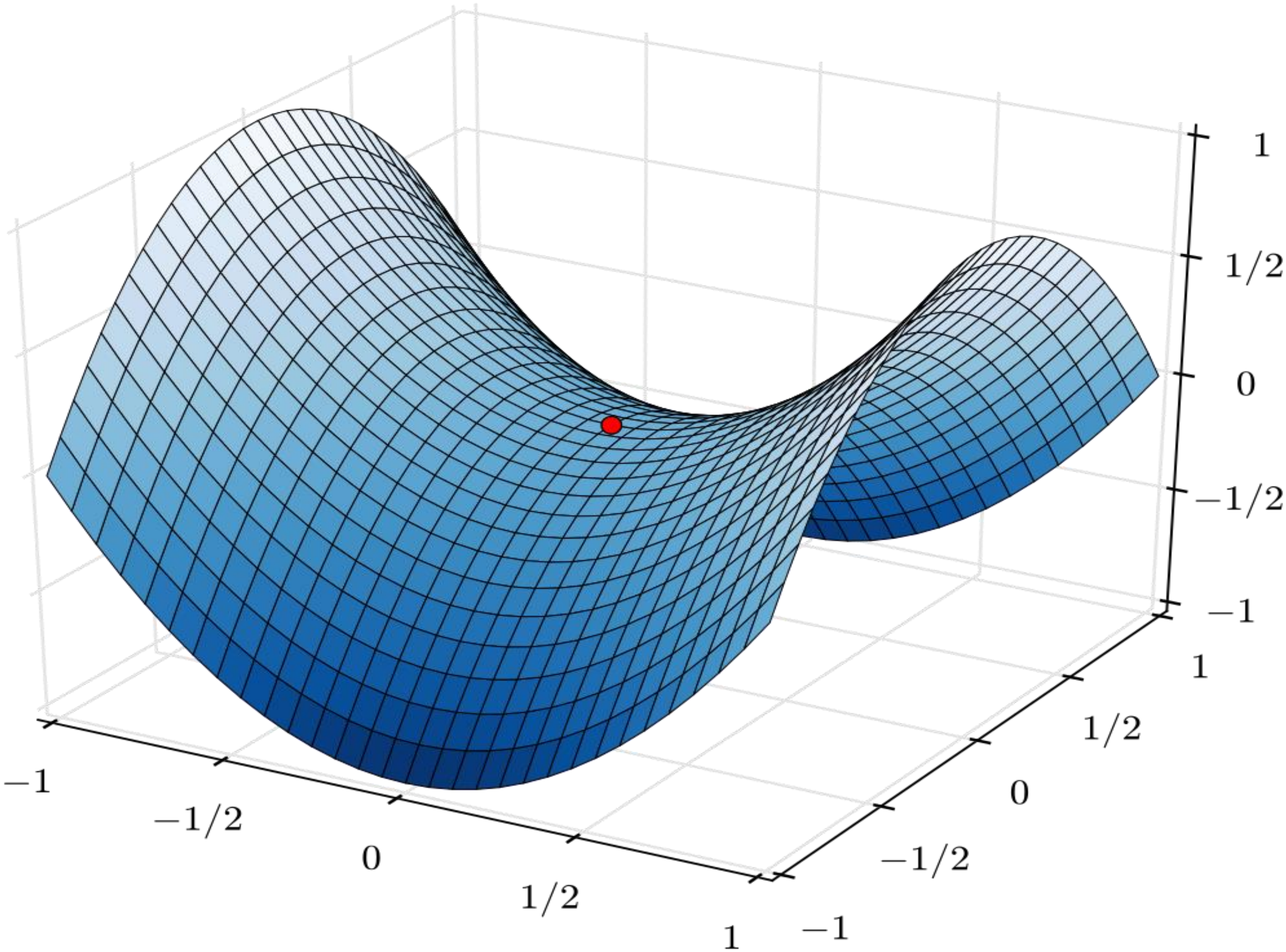
where t_i is the truth label and p_i is the Softmax probability for the i^{th} class.

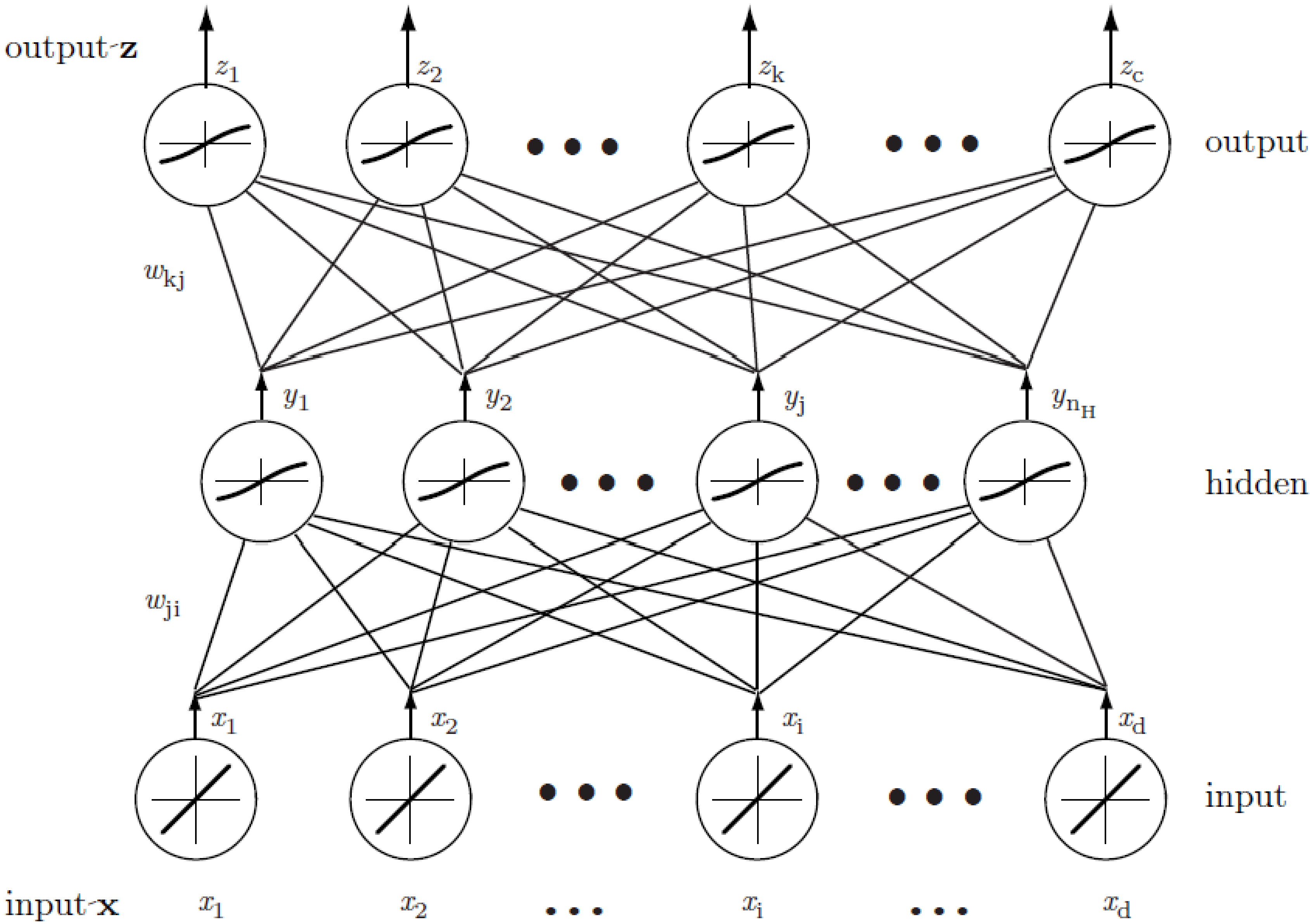
Error / cost function






Error
Weight 1
Weight 2


−1
−1/2
0
1/2
1
−1
−1/2
0
1/2
1
1
1/2
0
−1/2
−1



output z
z_1
z_2
z_k
z_c
output
w_{kj}
y_1
y_2
y_j
y_{n_H}
hidden
w_{ji}
x_1
x_2
x_i
x_d
input
input x
x_1
x_2
x_i
x_d

Multi layer perceptron

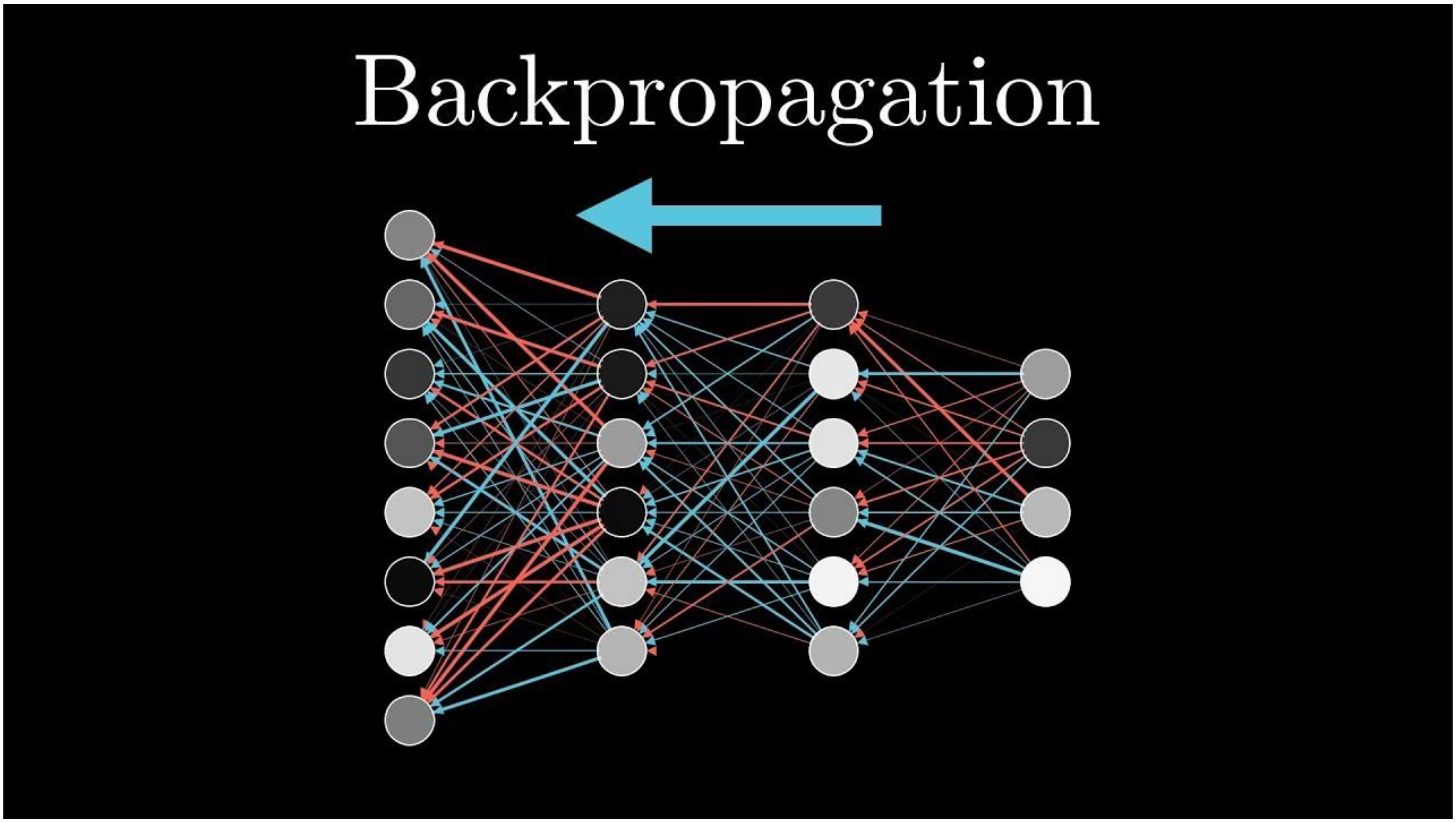
$$\Delta w_{mn} = -\eta \frac{\partial J}{\partial w_{mn}},$$

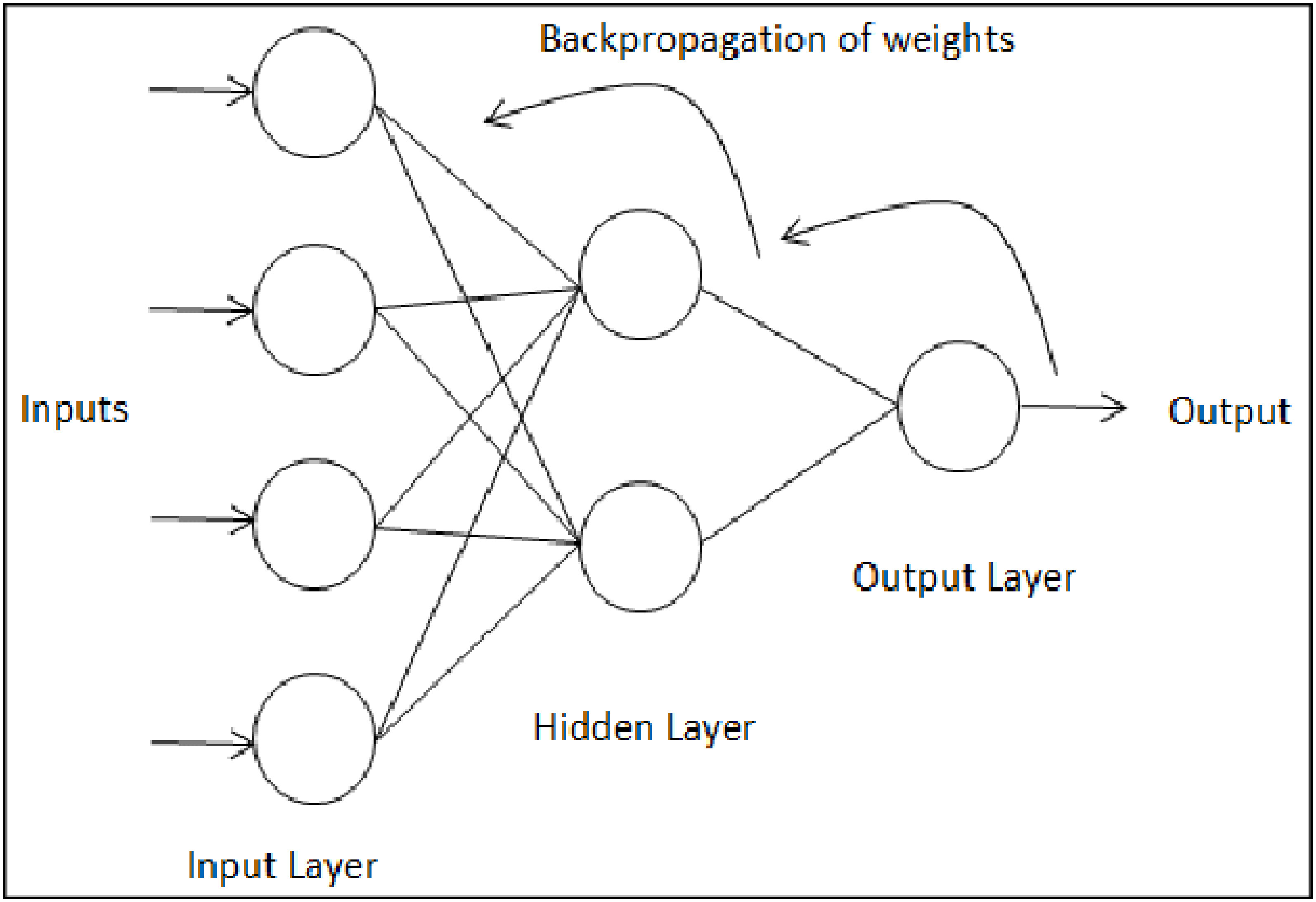
where η is the *learning rate*, and merely indicates the relative size of the change in weights.

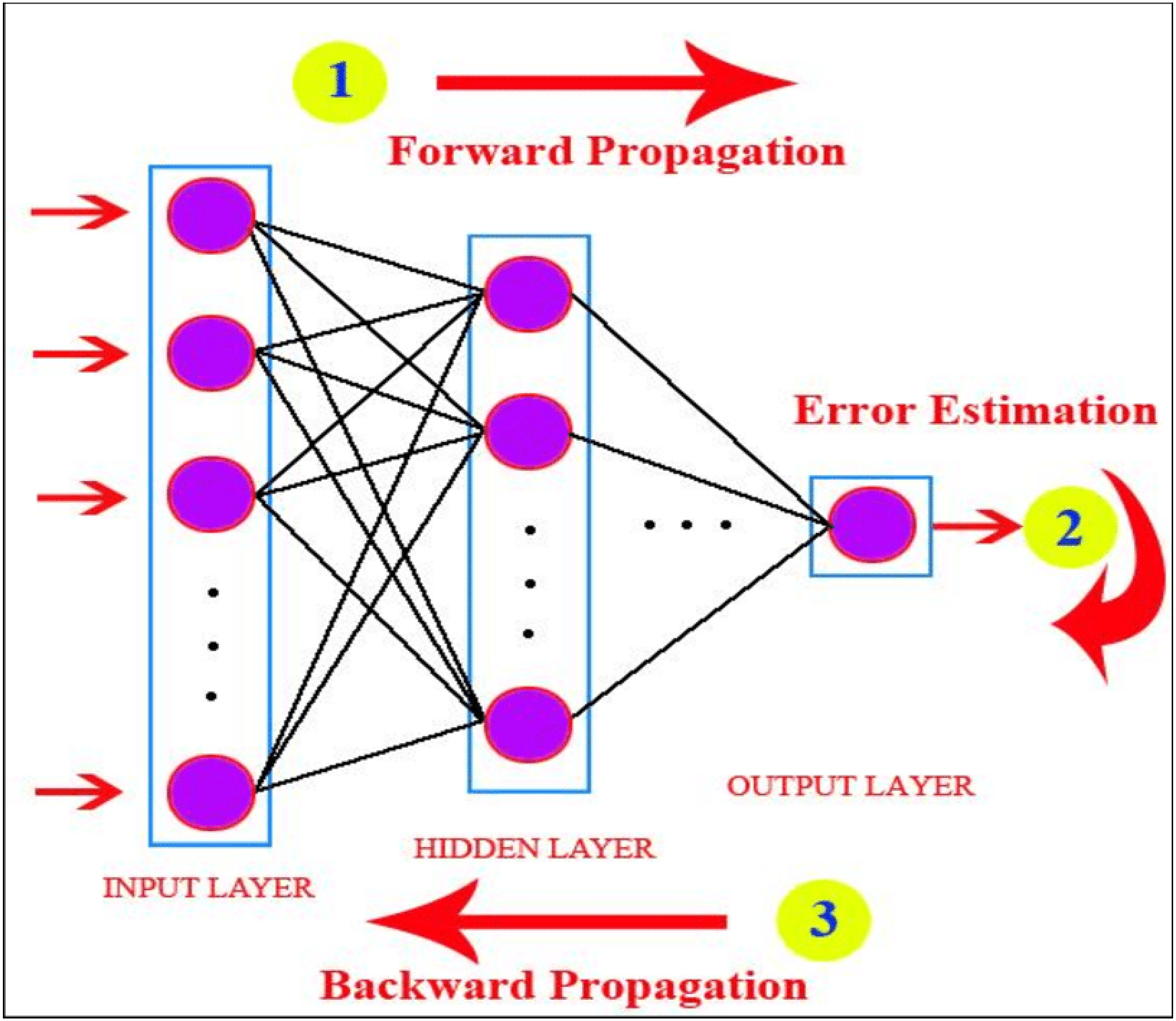
Multi layer perceptron

This iterative algorithm requires taking a weight vector at iteration m and updating it as:

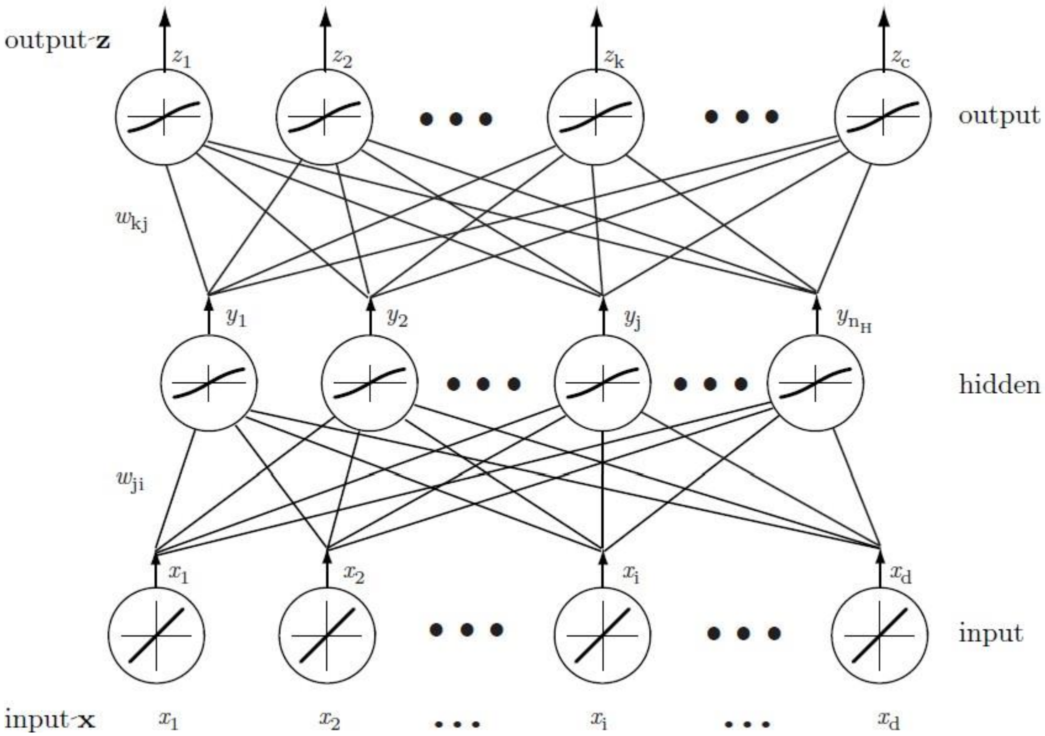
$$\mathbf{w}(m+1) = \mathbf{w}(m) + \Delta\mathbf{w}(m),$$


Backpropagation


Backpropagation of weights
Inputs
Output
Output Layer
Hidden Layer
Input Layer


1
Forward Propagation
Error Estimation
2
OUTPUT LAYER
HIDDEN LAYER
INPUT LAYER
3
Backward Propagation

Gradient descent : Back propagation



$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial net_k} \frac{\partial net_k}{\partial w_{kj}} = \delta_k \frac{\partial net_k}{\partial w_{kj}},$$

where the *sensitivity* of unit k is defined to be

$$\delta_k \equiv -\partial J / \partial net_k,$$

and describes how the overall error changes with the unit's activation.

$$\delta_k \equiv -\frac{\partial J}{\partial net_k} = -\frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial net_k} = (t_k - z_k) f'(net_k).$$

Gradient descent : Back propagation

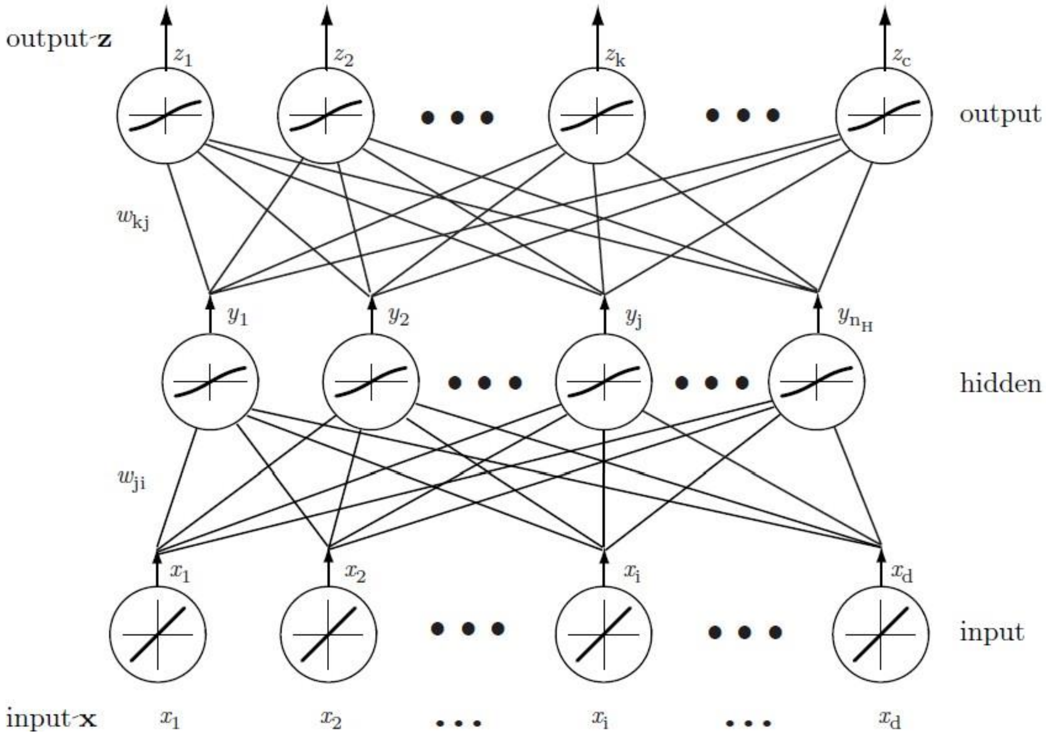
$$\frac{\partial net_k}{\partial w_{kj}} = y_j.$$

Taken together, these results give the weight update (learning rule) for the hidden-to-output weights:

$$\Delta w_{kj} = \eta \delta_k y_j = \eta (t_k - z_k) f'(net_k) y_j.$$

Gradient descent : Back propagation

The learning rule for the input-to-hidden units is more subtle,



$$\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j}\frac{\partial y_j}{\partial net_j}\frac{\partial net_j}{\partial w_{ji}}.$$

$$\begin{aligned}
\frac{\partial J}{\partial y_j} &= \frac{\partial}{\partial y_j}\left[1/2\sum_{k=1}^{c}(t_k - z_k)^2\right] \\
&= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial y_j} \\
&= -\sum_{k=1}^{c}(t_k - z_k)\frac{\partial z_k}{\partial net_k}\frac{\partial net_k}{\partial y_j} \\
&= -\sum_{k=1}^{c}(t_k - z_k)f'(net_k)w_{jk}.
\end{aligned}$$

Gradient descent : Back propagation

$$\delta_j \equiv f'(net_j) \sum_{k=1}^{c} w_{kj} \delta_k.$$

the sensitivity

at a hidden unit is simply the sum of the individual sensitivities at the output units weighted by the hidden-to-output weights w_{jk}, all multiplied by $f'(net_j)$. Thus the learning rule for the input-to-hidden weights is:

$$\Delta w_{ji} = \eta x_i \delta_j = \eta x_i f'(net_j) \sum_{k=1}^{c} w_{kj} \delta_k.$$

Gradient descent : Back propagation

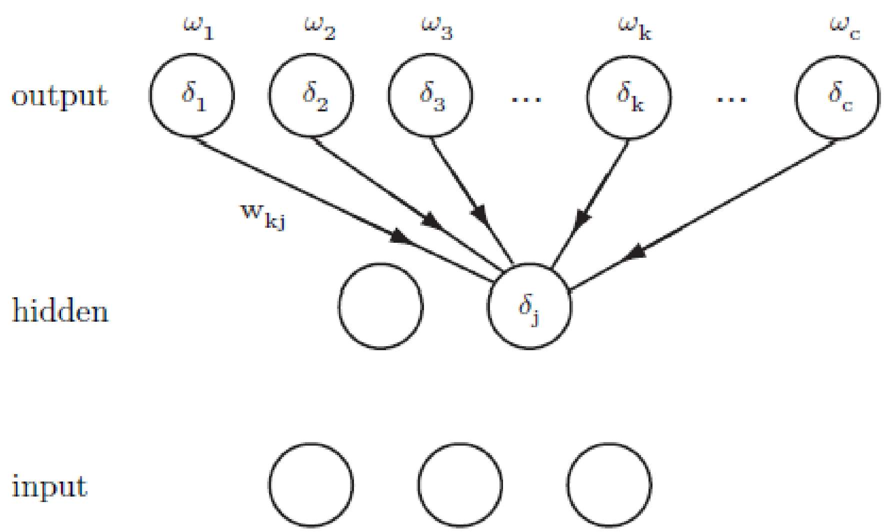


Figure 6.5: The sensitivity at a hidden unit is proportional to the weighted sum of the sensitivities at the output units: $\delta_j = f'(net_j) \sum_{k=1}^{c} w_{kj}\delta_k$. The output unit sensitivities are thus propagated "back" to the hidden units.

Training methodologies

The three most useful training protocols are: stochastic, batch and on-line.

In stochastic training (or pattern training), patterns are chosen randomly from the stochastic training set, and the network weights are updated for each pattern presentation.

In batch training, all patterns are presented to the network before learning batch (weight update) takes place.

In on-line training, each pattern is presented once and on-line protocol onlyonce; there is no use of memory for storing the patterns.

MOMENTUM BASED GRADIENT DESCENT

$$\Delta w_{ij} = (\eta * \frac{\partial E}{\partial w_{ij}})$$

weight increment

learning rate

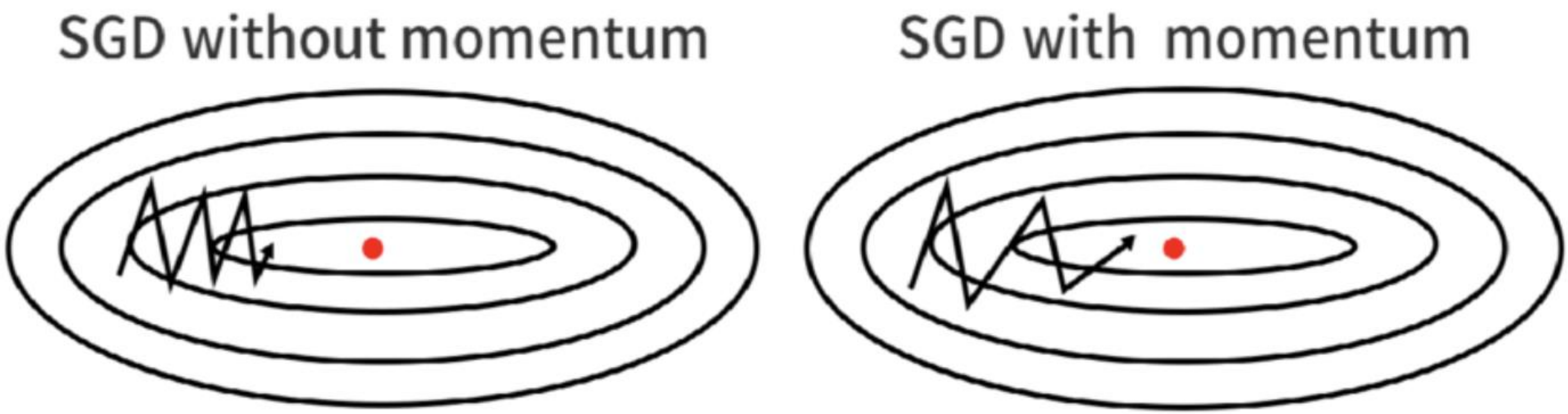
weight gradient

$$\Delta w_{ij} = (\eta * \frac{\partial E}{\partial w_{ij}}) + (\gamma * \Delta w_{ij}^{t-1})$$

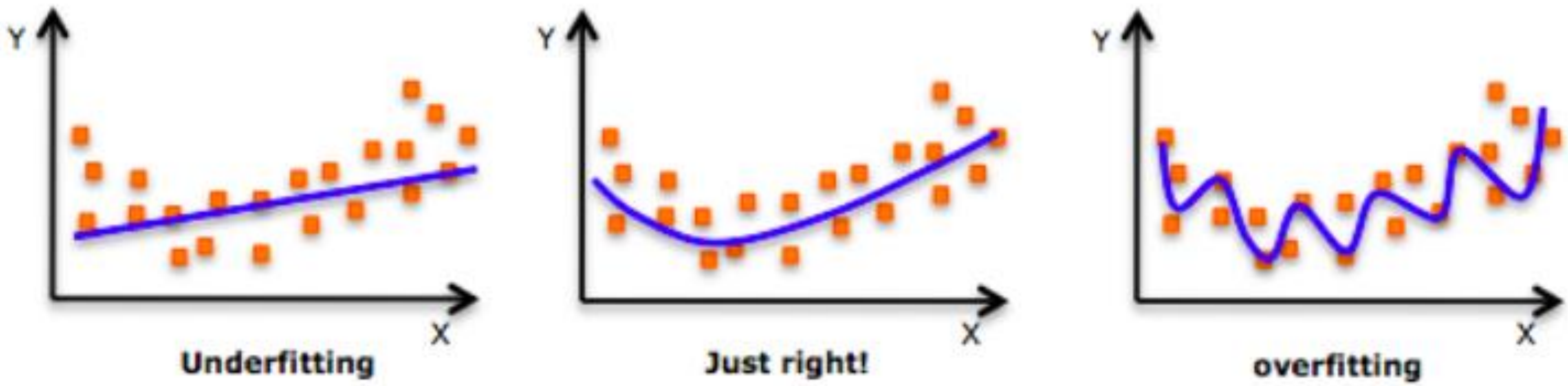
momentum factor

weight increment, previous iteration

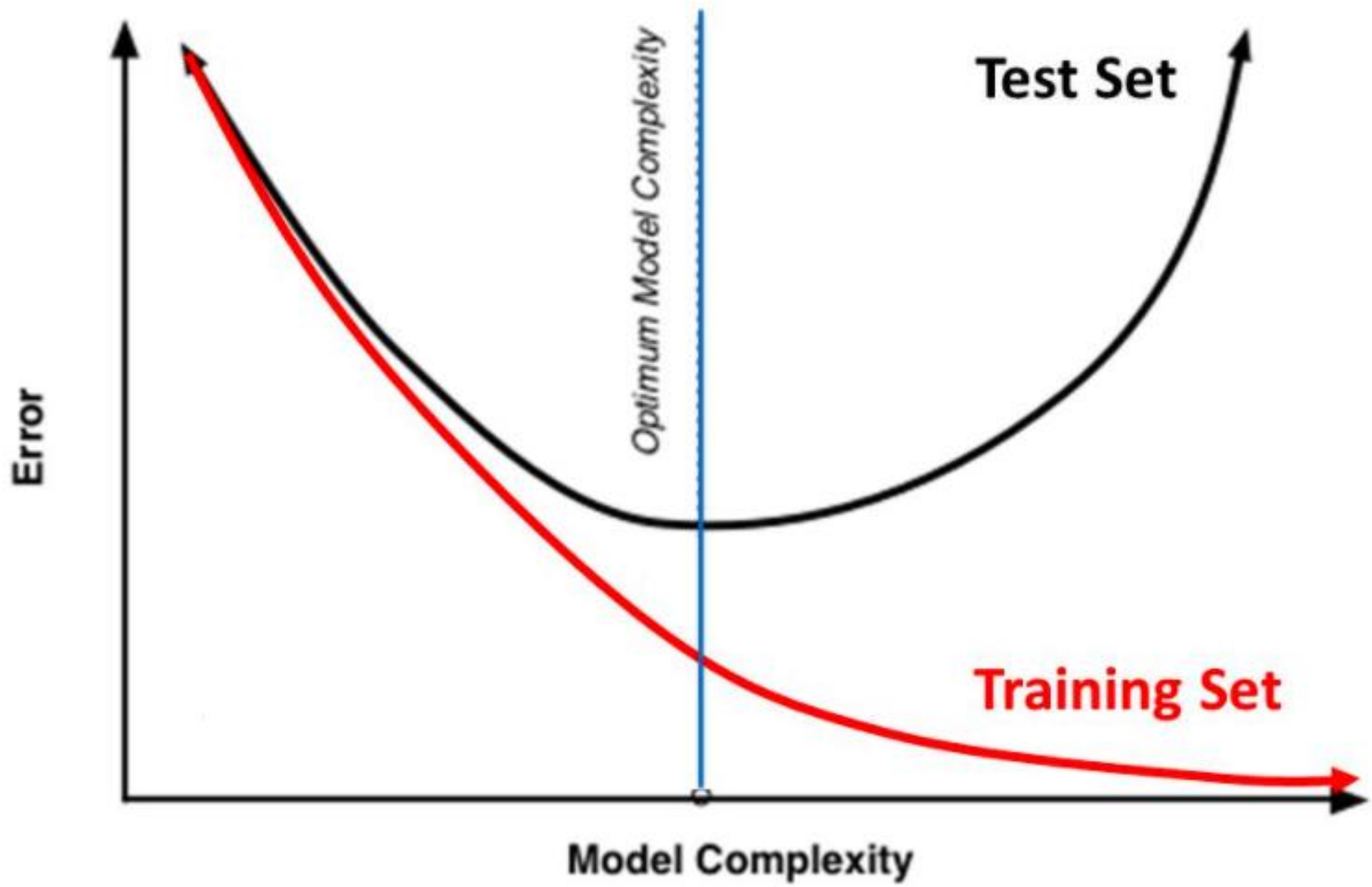
MOMENTUM BASED GRADIENT DESCENT



REGULARIZATION



Training Vs. Test Set Error



REGULARIZATION TECHNIQUES

L_2 regularization

L_1 regularization

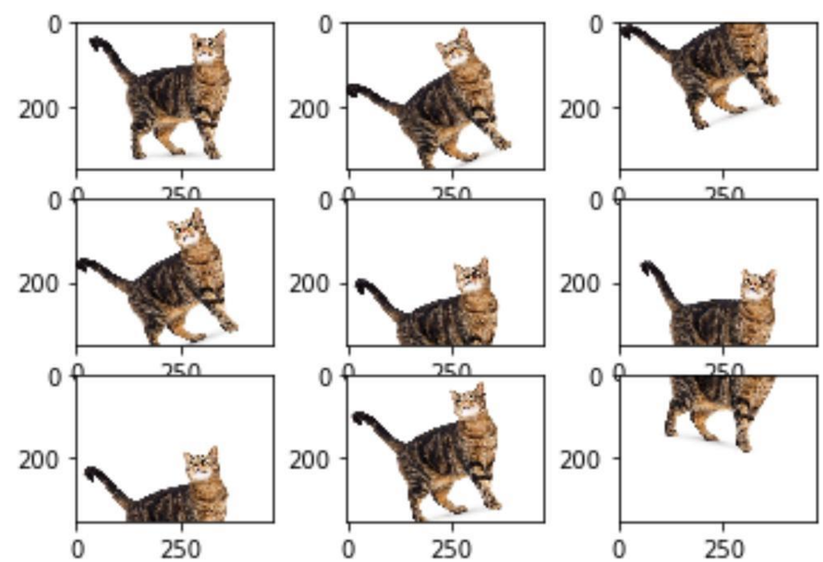
Data Augmentation

Early stopping

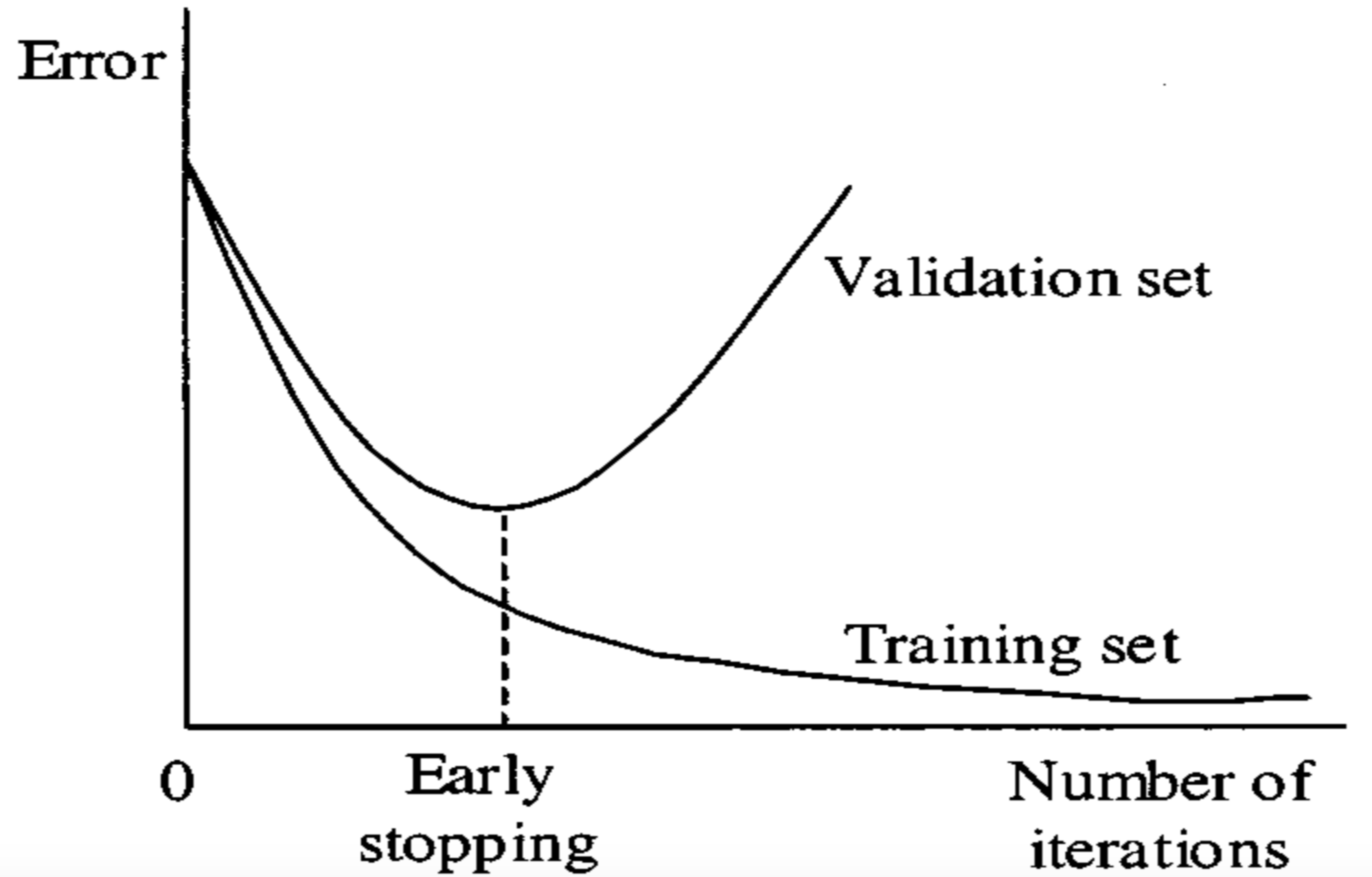
Adding noise to input

Adding noise to target

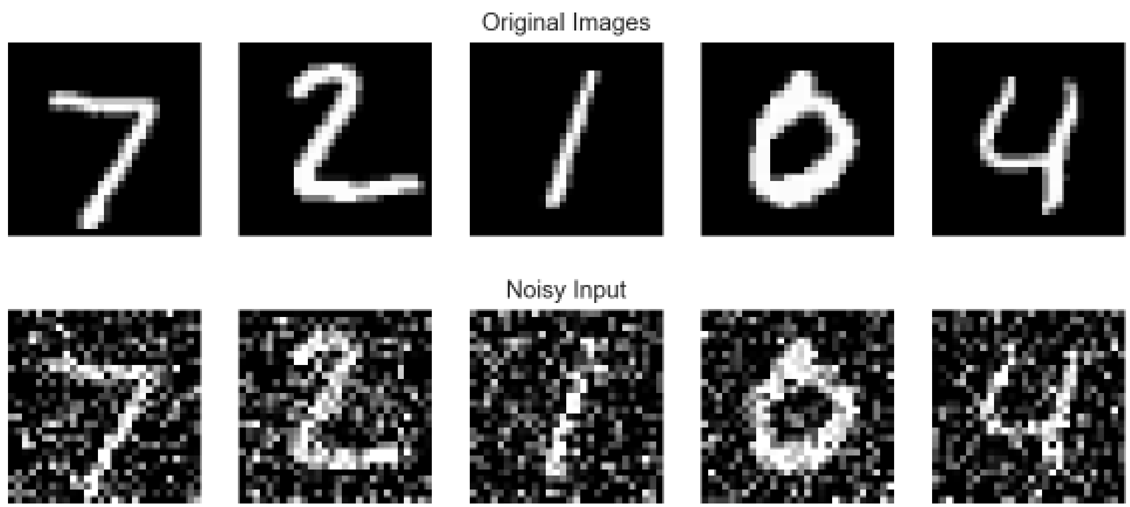
Data Augmentation



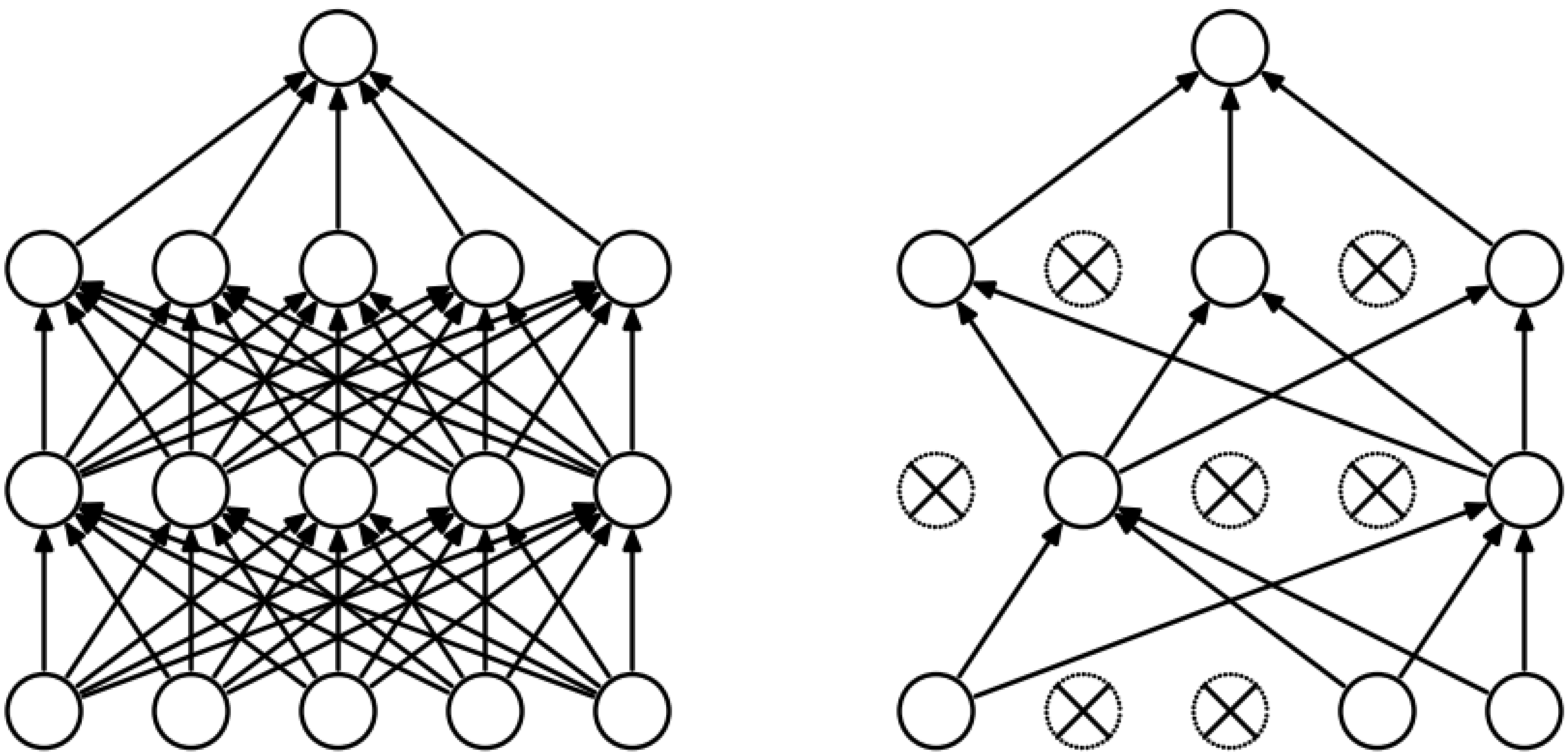
Early stopping



Adding noise to inputs



Drop-out for regularization



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

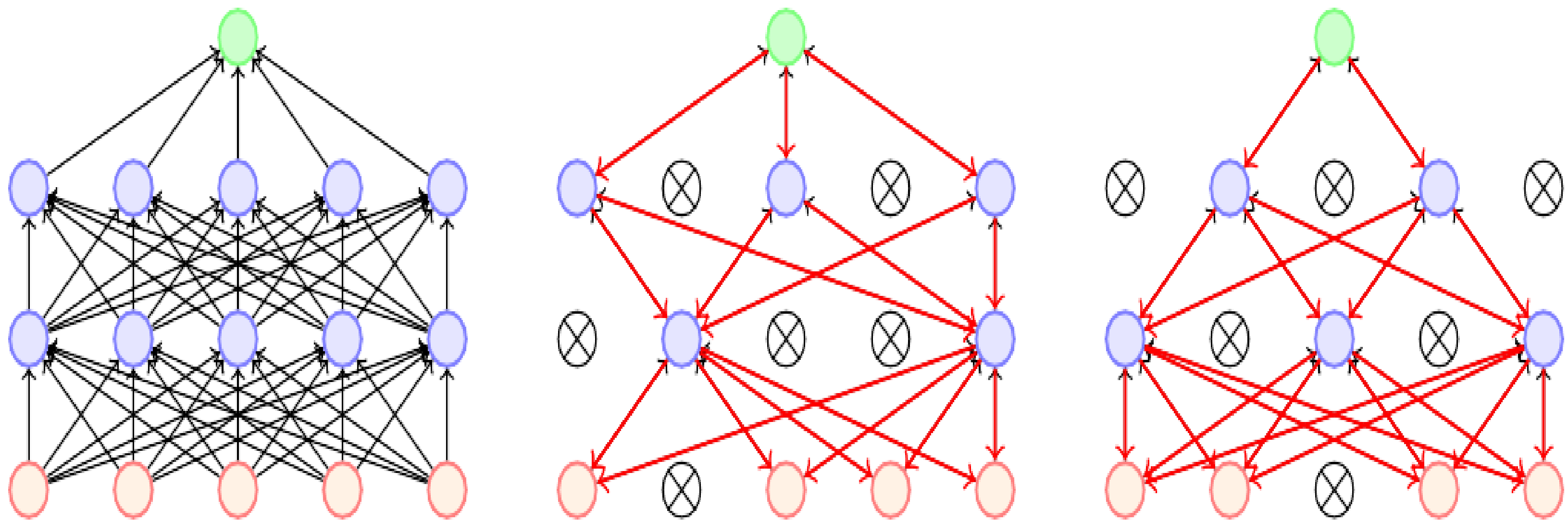
Drop-out for regularization

We initialize all the parameters (weights) of the network and start training

For the first training instance (or mini-batch), we apply dropout resulting in the thinned network

We compute the loss and back propagate

Which parameters will we update? Only those which are active



For the second training instance (or mini-batch), we again apply dropout resulting in a different thinned network

We again compute the loss and backpropagate to the active weights

If the weight was active for both the training instances then it would have received two updates by now

If the weight was active for only one of the training instances then it would have received only one updates by now