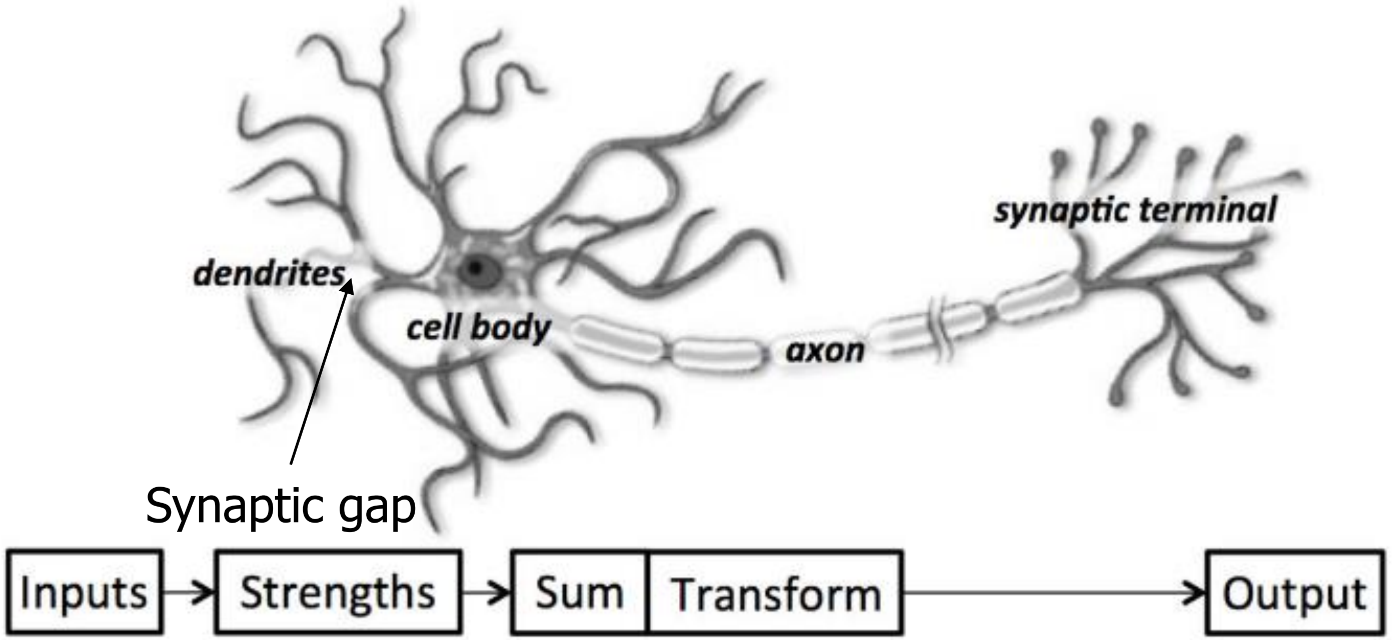
Feed-forward network (FFN)

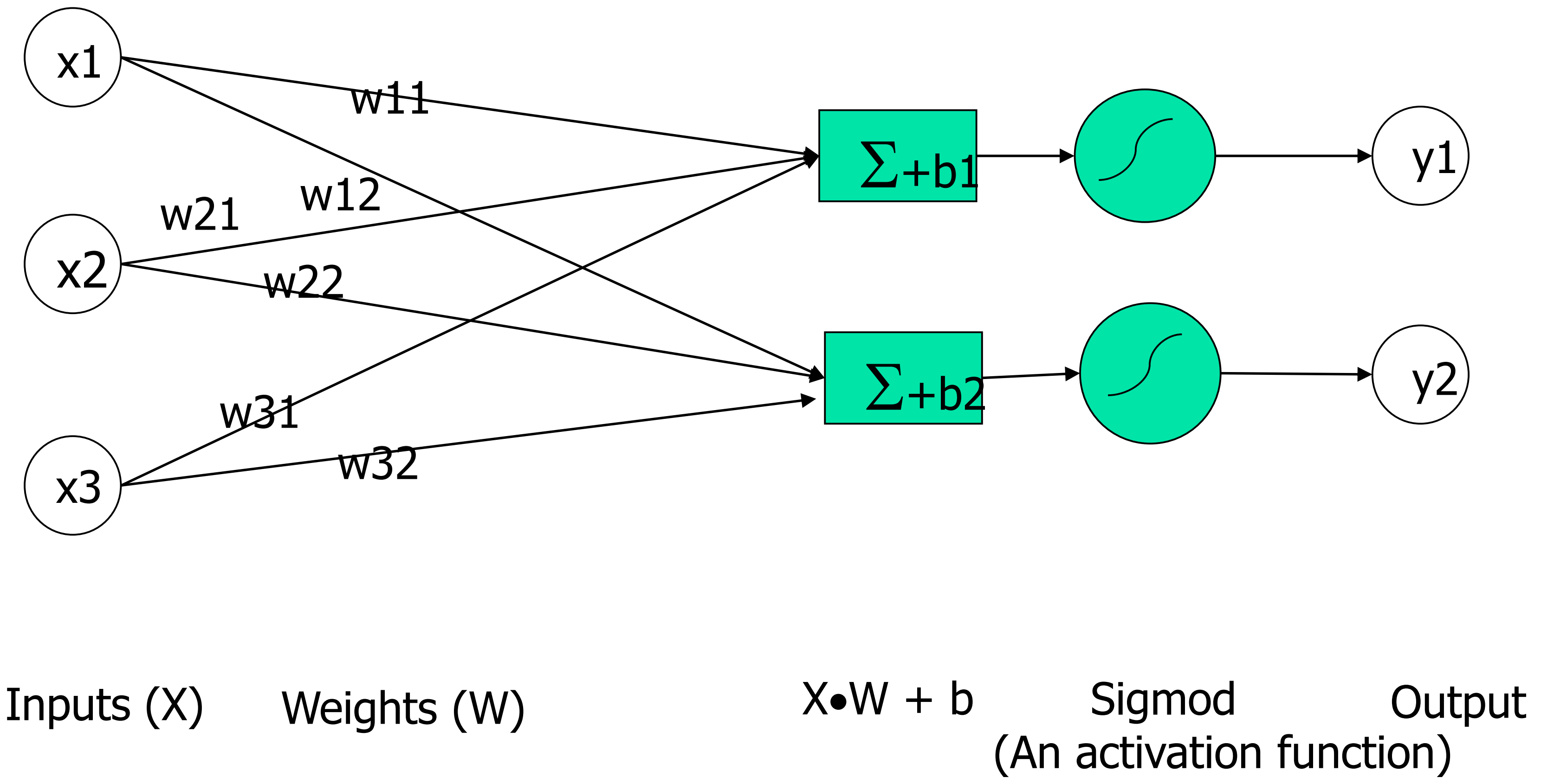
- Architecture
- Activation functions
- Weights updating (Backward propagation)
- Overfitting
- Training
- Examples

A biological neuron

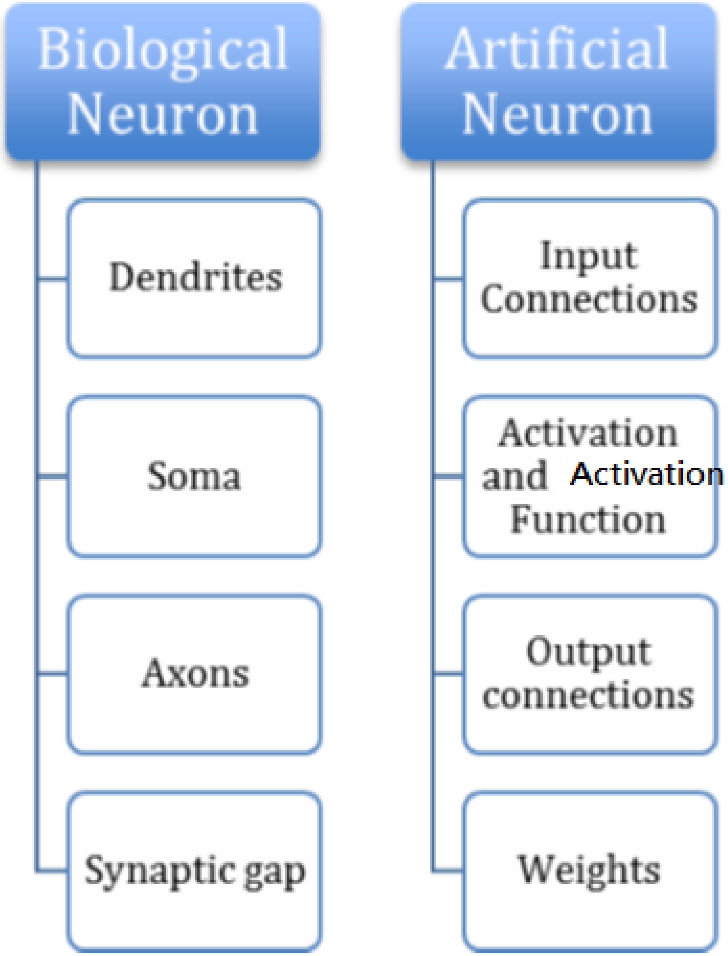


An Artificial neural (A mathematic model)

Y=activation(X*W+b)

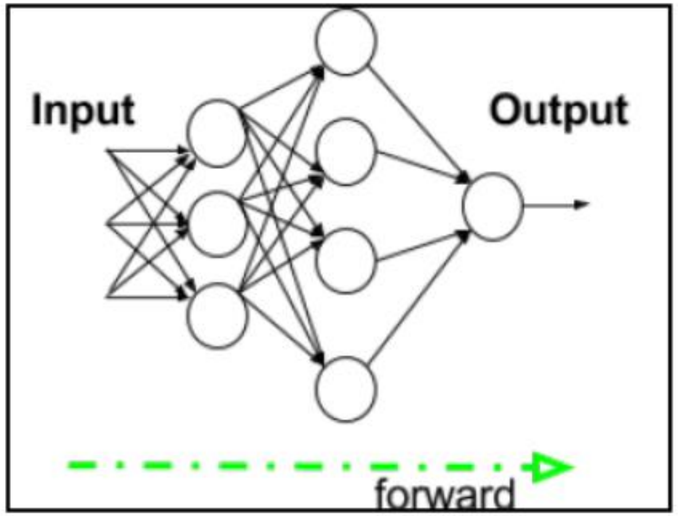


Analogy

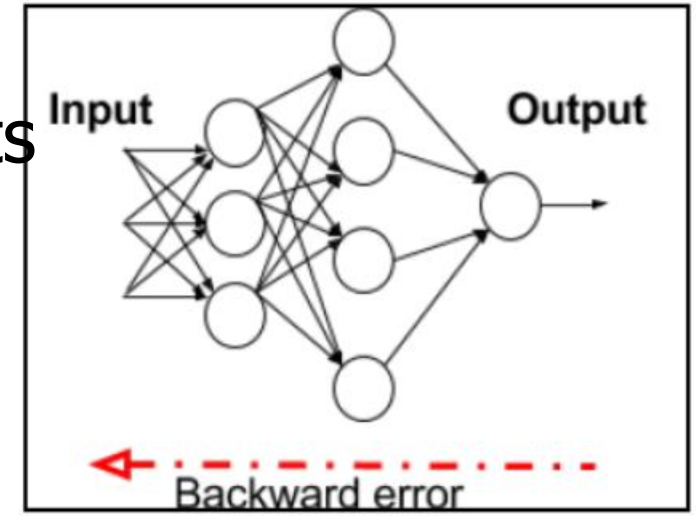

Biological Neuron
Dendrites
Soma
Axons
Synaptic gap
Artificial Neuron
Input Connections
Activation and Activation Function
Output connections
Weights

Training

- Forward propagates & calculate errors
  (t-y): t is the true value & y is the output



- Backward propagates error (actually, using gradients on error func.) to adjust weights to minimize error



- Y is a func of weights w and input x

Error func.

- Error function: E$= \frac{1}{2}\Sigma_i\left(t^{(i)} - y^{(i)}\right)^2$, here *i* denotes the *i*th sample
- Is a func. of y, which in turn is a func. of w and x

Gradient descent

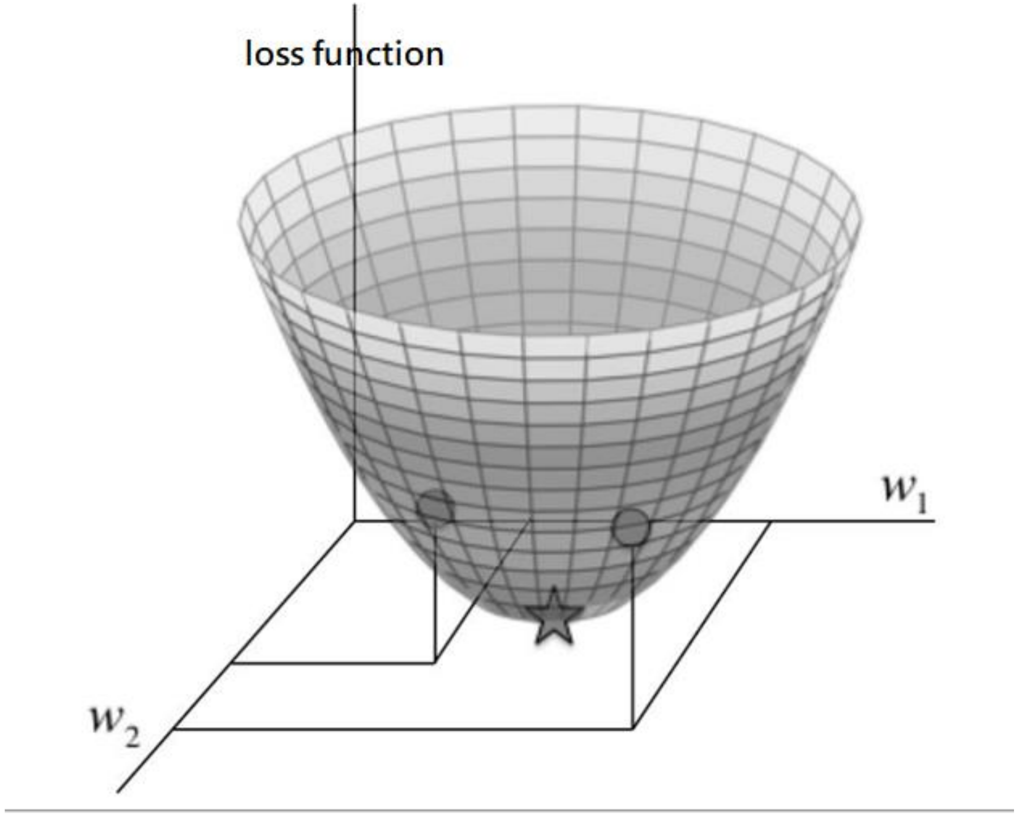
- Gradient of loss func. At (w1, w2)
- $\nabla(E) = \nabla\left(\Sigma_i\left(y^i - t^{\mathrm{i}}\right)^2\right)$=
- $\nabla(J(w))$, $J(w)$ is called the loss function, or error func

- The gradient at (w_1, w_2) is a vector, and along its head direction one can get the maximum increment on the loss function.

- It can be proved that

$$\nabla(J(w))=\left(\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}\right)$$

- Since we want to decrease the loss function, we update the parameters in proportion to - $\nabla(J(w))$, the opposite direction of the gradient



Updating rule

$$w = w - \epsilon\nabla\left(\Sigma_i\left(y^i - t^\mathrm{i}\right)^2\right) = w - \epsilon\nabla(J(w))$$

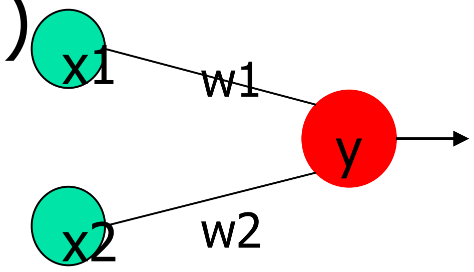
Gradient descent: move along the direction with the greatest decrement
Of function value, which is the opposite direction of the gradient
ϵ: a small step of update, called the learning rate
w: denotes the weight vector
y^i: output value of sample i
t^i: true value of sample i

Note that square function is one of the loss function, others like cross entropy.

Updating rule

- $\Delta w_k = -\epsilon \frac{\partial E}{\partial w_k}, \quad and\ w_k' = w_k + \Delta w_k$

- For a simple case (Only one layer, no activation!)
- $y^{(i)} = \Sigma_k w_k x_k^{(i)}$



- $\Delta w_k = -\epsilon \frac{\partial E}{\partial w_k}$
- $= -\epsilon \frac{\partial}{\partial w_k} \left(\frac{1}{2} \sum_i \left(t^{(i)} - y^{(i)} \right)^2 \right), for\ all\ sample\ i$
- $= \sum_i \varepsilon (t^{(i)} - y^{(i)}) \frac{\partial y^{(i)}}{\partial w_k}$
- $= \sum_i \epsilon x_k^{(i)} (t^{(i)} - y^{(i)})$ (Quite simple!)

With Sigmoid activation function

- z=$\sum_k w_k x_k$
- $y = \frac{1}{1+e^{-z}}$
- $\frac{\partial z}{\partial w_k}$=$x_k$

$$\frac{dy}{dz} = \frac{e^{-z}}{\left(1+e^{-z}\right)^2}$$

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

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

$$= y(1-y)$$

$$\frac{\partial y}{\partial w_k} = \frac{dy}{dz} \frac{\partial z}{\partial w_k} = x_k y(1-y)$$

Sigmoid case (contd.) (One hidden layer)

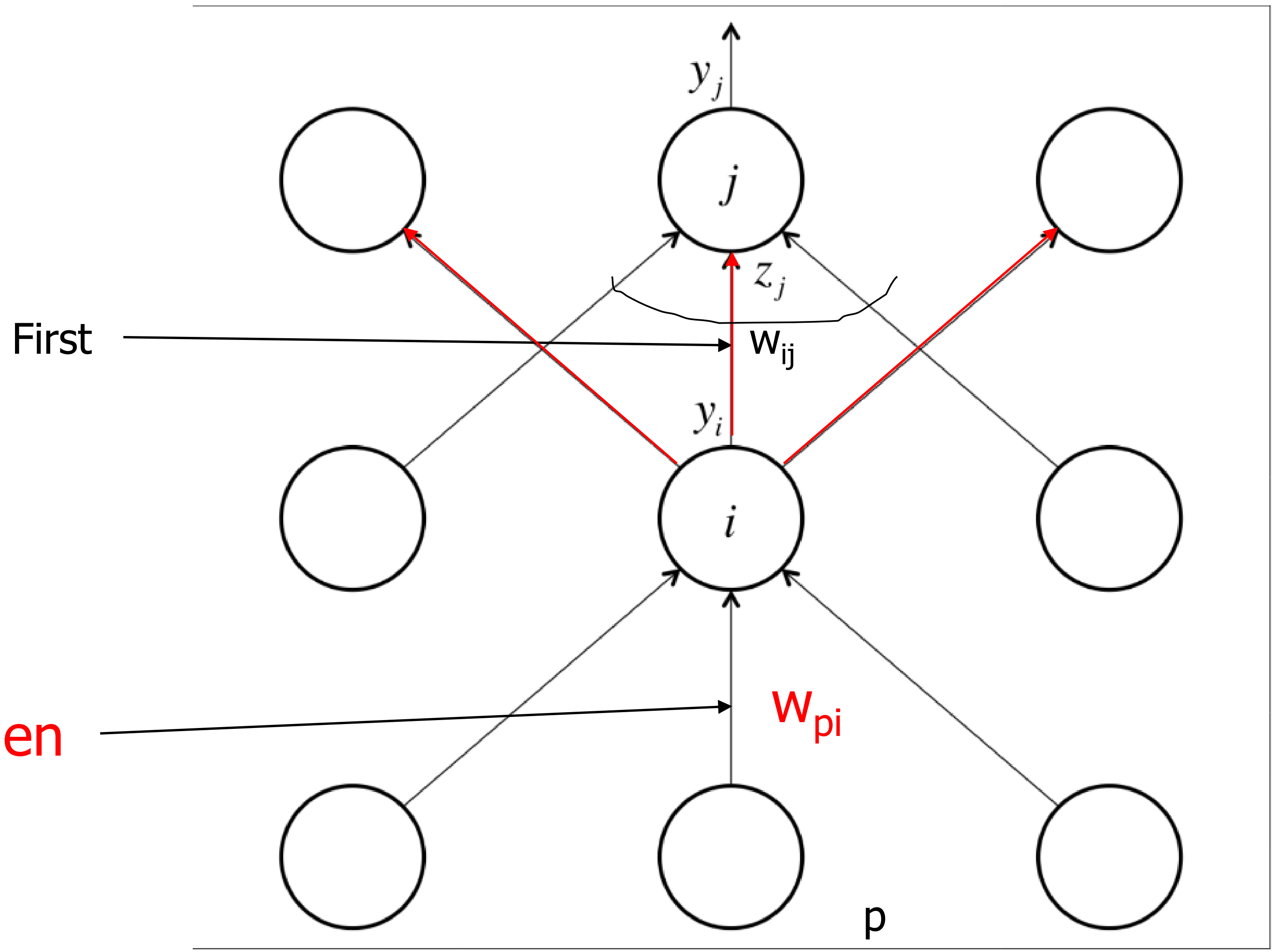
Given E$= \frac{1}{2}\Sigma_i\left(t^{(i)} - y^{(i)}\right)^2$, we have only one output node and many samples in this case

$$\frac{\partial E}{\partial w_k} = \Sigma_i \frac{\partial E}{\partial y^{(i)}} \frac{\partial y^{(i)}}{\partial w_k} = -\Sigma_i x_k^{(i)} y^{(i)}\left(1 - y^{(i)}\right)\left(t^{(i)} - y^{(i)}\right)$$

$$\Delta w_k = \Sigma_i \epsilon x_k^{(i)} y^{(i)}\left(1 - y^{(i)}\right)\left(t^{(i)} - y^{(i)}\right)$$

y(1-y) is the extra term to account for derivative of the sigmoid func.

Backward propagation (many hidden layers) for one sample



Key point: Gradient of internal node $\frac{\partial E}{\partial y_i}$

- For output layer:

$$E = \frac{1}{2}\Sigma_{j \in output}\left(t_j - y_j\right)^2 \Rightarrow \frac{\partial E}{\partial y_j} = -\left(t_j - y_j\right)$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial y_i}$$

$$\frac{\partial y_j}{\partial y_i} = \frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial y_i}; \quad and\ y_j = \sigma(z_j)$$

- For an internal node i: its gradient is affected by all of its output nodes

$$\frac{\partial \mathrm{E}}{\partial y_i} = \sum_j \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial y_i} = \sum_j \frac{\partial E}{\partial y_j}\underbrace{\frac{\partial y_j}{\partial z_j}}_{y_j(1-y_j)}\underbrace{\frac{\partial z_j}{\partial y_i}}_{w_{ij}}$$

Here, $z_j = \Sigma_k w_{kj} y_k$

Input to node j

Consider the sigmod func. between input of node j and output of node j

Put together, we have:

$$\frac{\partial E}{\partial y_i} = \sum_j w_{ij} y_j \left(1 - y_j\right) \frac{\partial E}{\partial y_j}$$

Now we know how to calculate the derivative of E with respect to any y_i

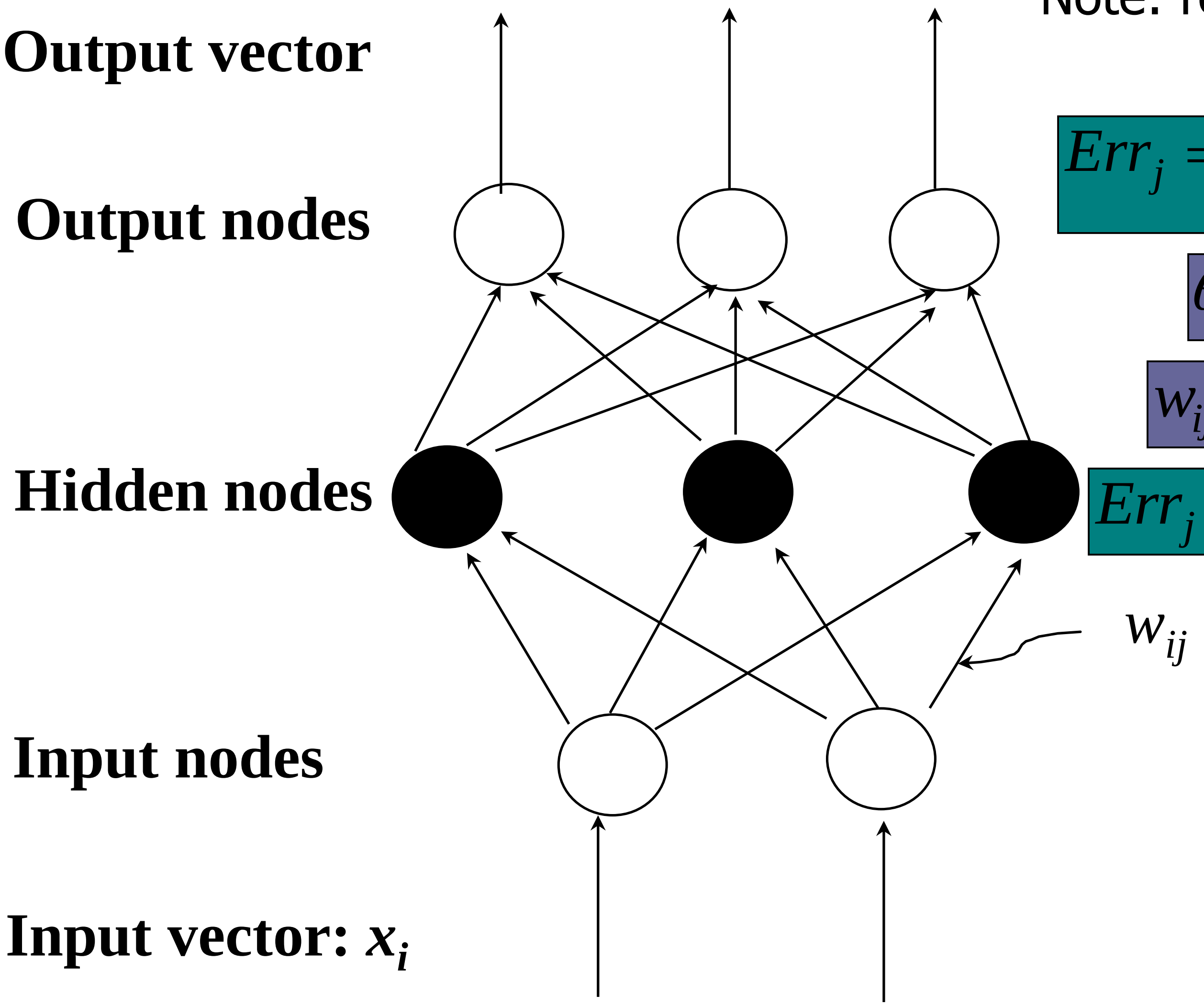
Now consider the node p at the previous layer of node i

$$\boxed{\frac{\partial E}{\partial y_i} = \sum_j w_{ij} y_j (1 - y_j) \frac{\partial E}{\partial y_j}}$$

$p \longrightarrow i \longrightarrow j$

$$\frac{\partial E}{\partial w_{pi}} = \frac{\partial E}{\partial y_i} \frac{\partial y_i}{\partial z_i} \frac{\partial z_i}{\partial w_{pi}} = y_i (1 - y_i) \frac{\partial E}{\partial y_i} y_p$$

multiple output nodes, different notation



Note: replace y_j with O_j in the previous eq.

$$Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}$$

$$\theta_j = \theta_j + (l)Err_j$$

$$w_{ij} = w_{ij} + (l)Err_j O_i$$

$$Err_j = O_j(1-O_j)(T_j - O_j)$$

$$O_j = \frac{1}{1+e^{-I_j}}$$

$$I_j = \sum_i w_{ij} O_i + \theta_j$$

The general updating rule

- Find the gradient with respect to y_i:

$$\frac{\partial E}{\partial y_i} = \sum_j w_{ij} y_j \left(1 - y_j\right) \frac{\partial E}{\partial y_j}$$

- Find the gradient with respect to w_{ij}:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial z_j} \frac{\partial z_j}{\partial w_{ij}} = y_i y_j \left(1 - y_j\right) \frac{\partial E}{\partial y_j}$$

- Where :

$$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j} = y_j \left(1 - y_j\right) \frac{\partial E}{\partial y_j}$$

The general updating rule

- Updating rule: summarize all the update from many different samples

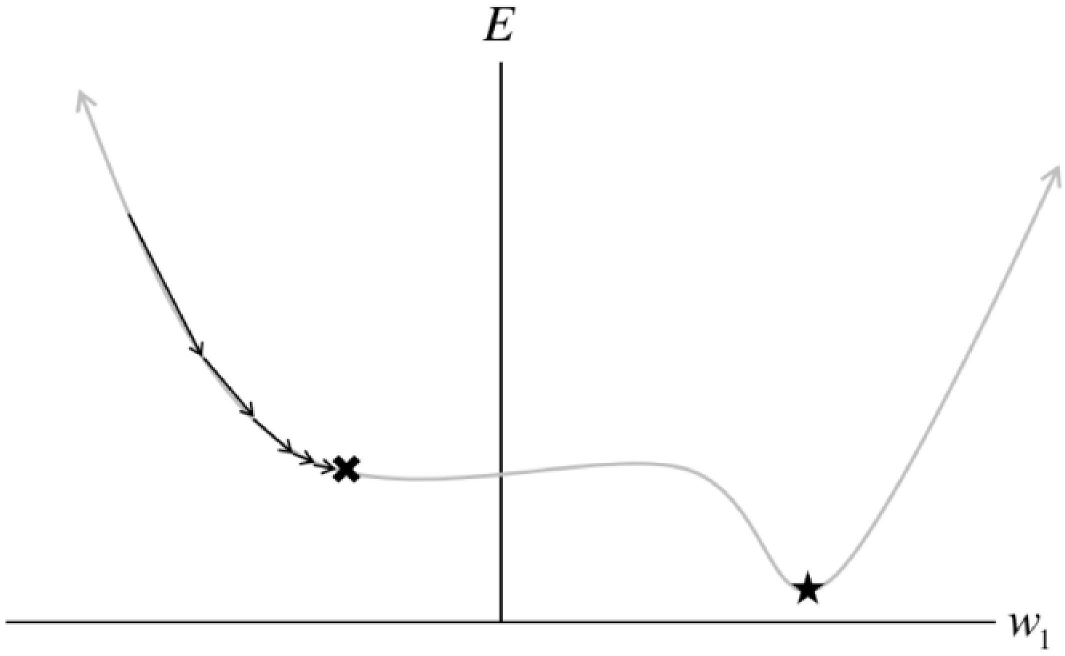
$$\Delta w_{ij} = -\sum_{k \in epoch} \epsilon y_i^{(k)} y_j^{(k)} (1 - y_j^{(k)}) \frac{\partial E^{(k)}}{\partial y_j^{(k)}}$$

Terminating condition

- All Δw_{ij} in the previous epoch were so small as to be below some specified threshold, or
- The percentage of samples misclassified in the previous epoch is below some threshold, or
- A prespecified number of epochs has expired.

In practice, several hundreds of thousands of epochs may be required before the weights converge.

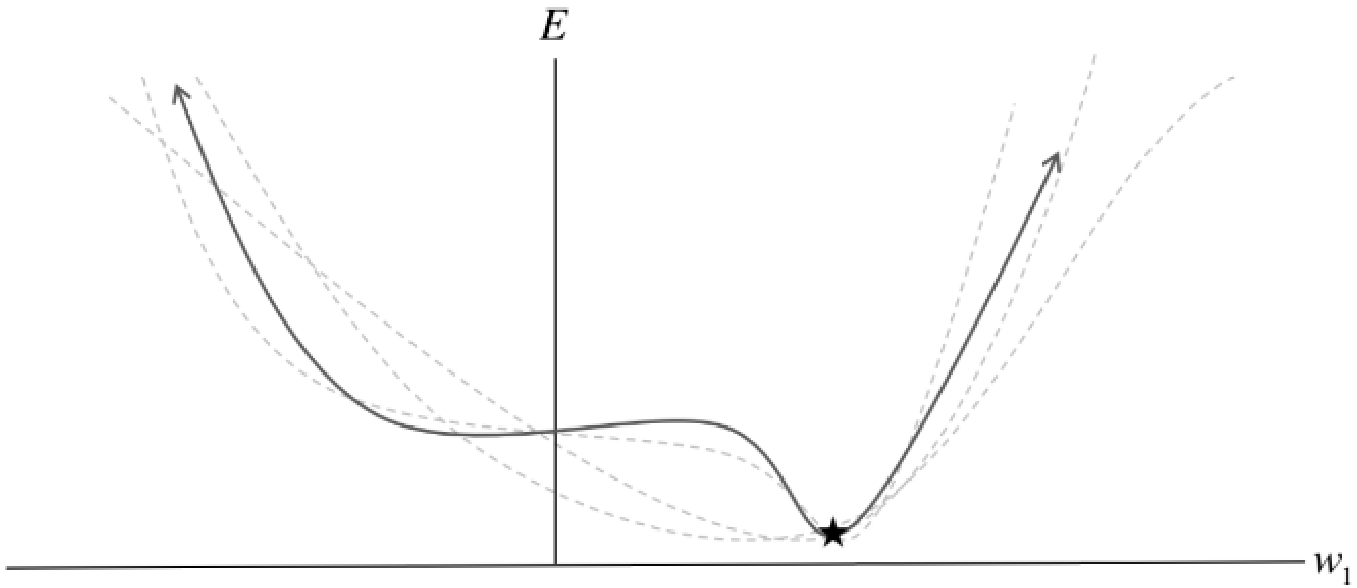
Batch gradient descent



Batch gradient descent

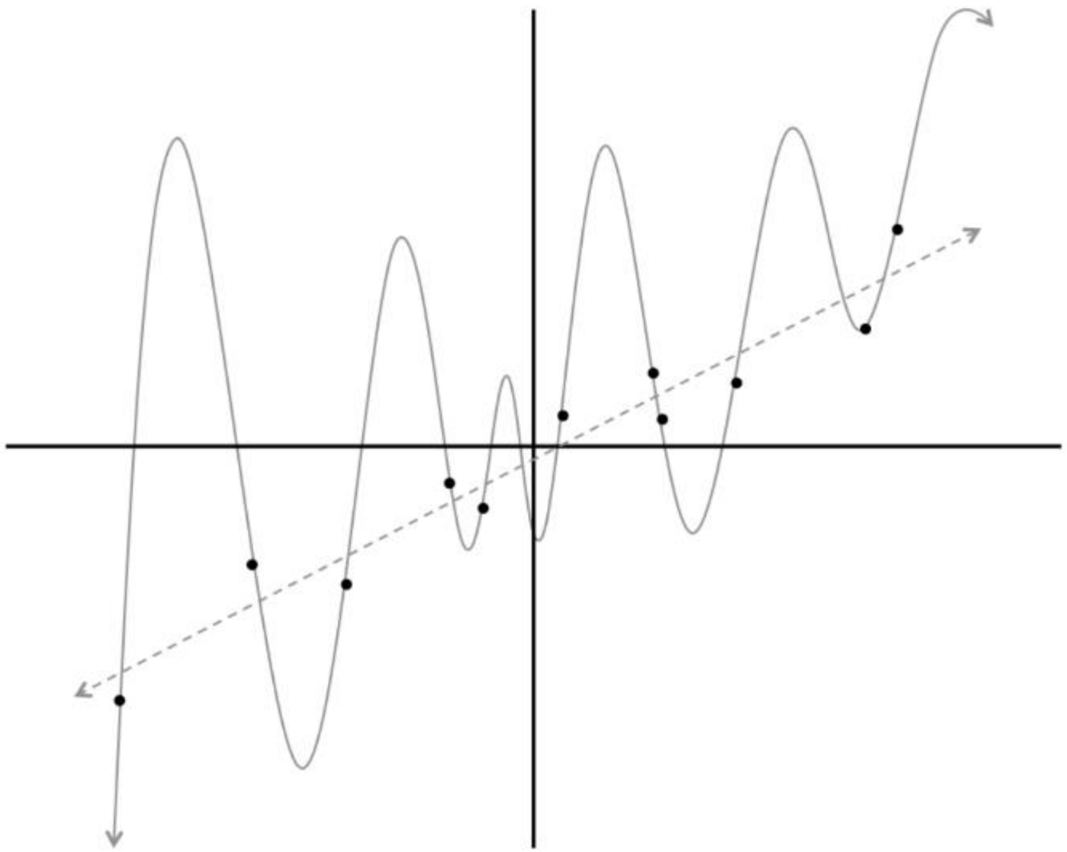
Minibatch

$$\Delta w_{ij} = -\sum_{k \in minibatch} \epsilon y_i^{(k)} y_j^{(k)} \left(1 - y_j^{(k)}\right) \frac{\partial E^{(k)}}{\partial y_j^{(k)}}$$



Minibatch has many different search paths to avoid local minimum

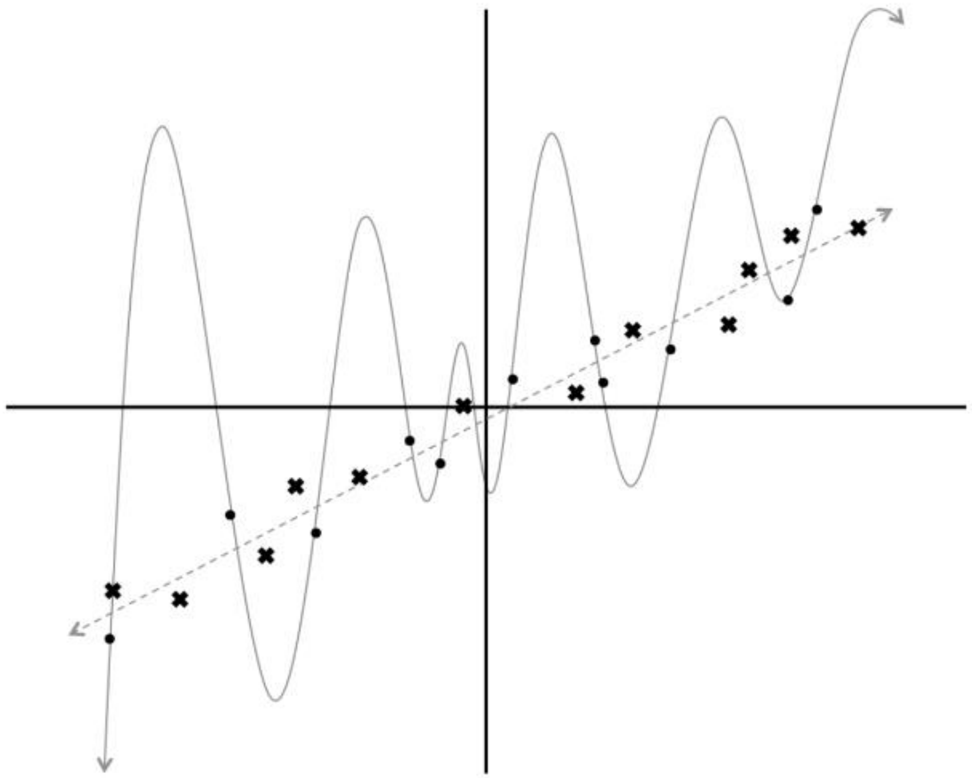
overfitting



Linear vs. polynomial of power of 12

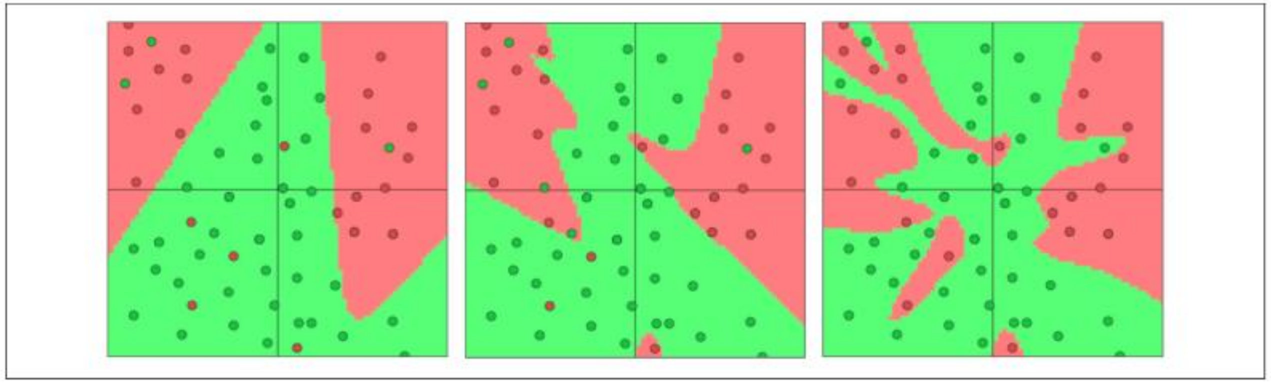
Ploynomial overfit

- For testing data



Neural net overfitting

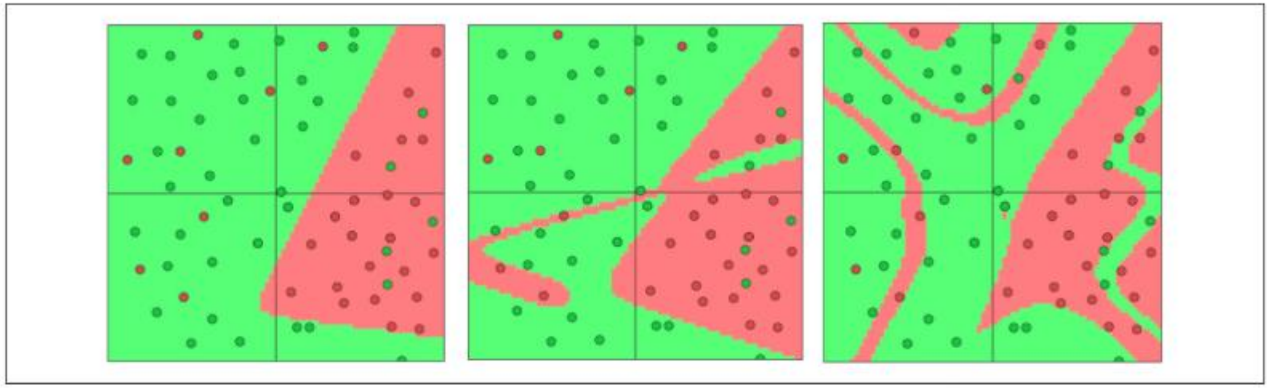
Two inputs, one hidden layer, softmax activation with two outputs



3, 6, 20 neurons, respectively, in the hidden layer

Neural network overfitting

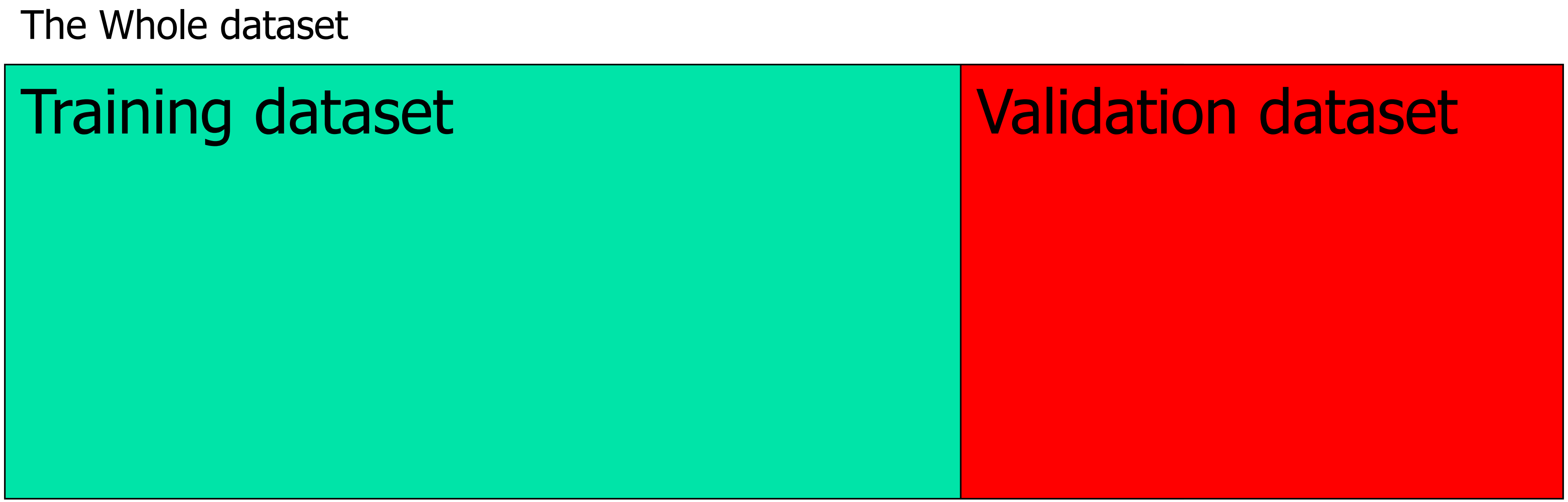
Hidden layer contains 3 neurons

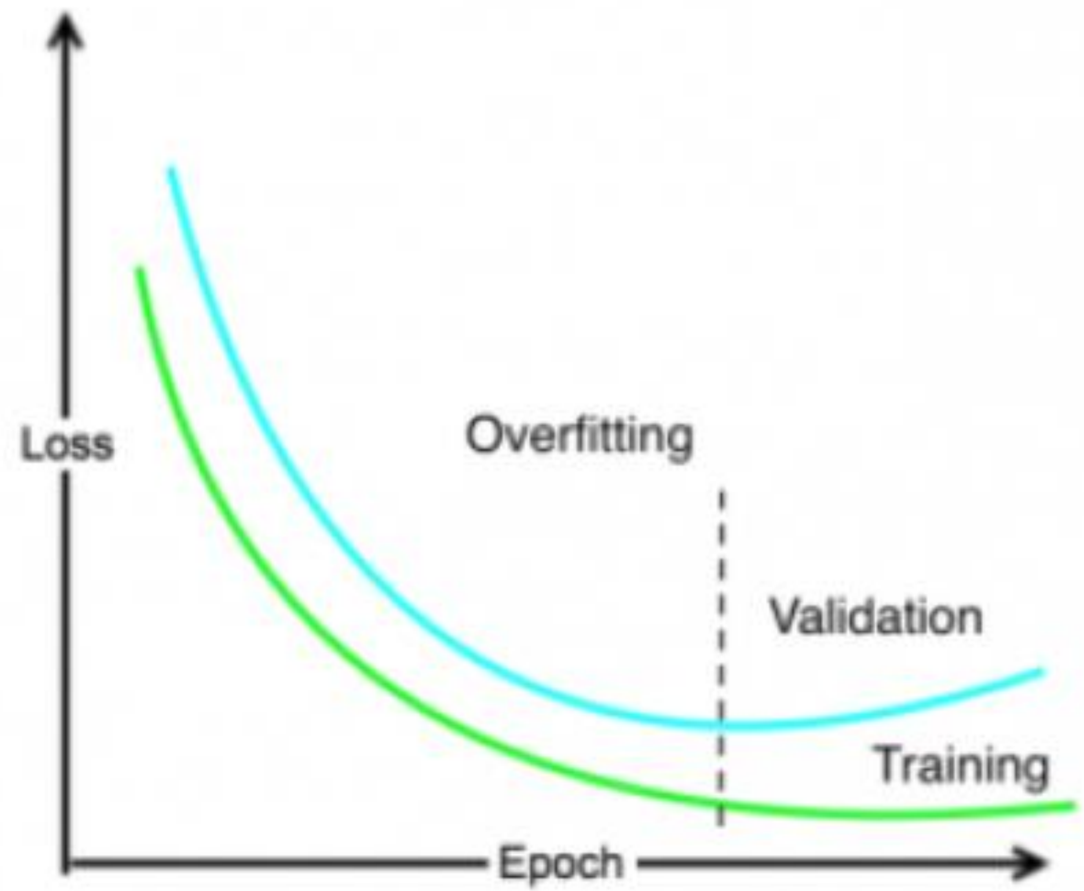


With 1, 2, 4 hidden layers, respectively,

To prevent overfitting

- Use validation dataset

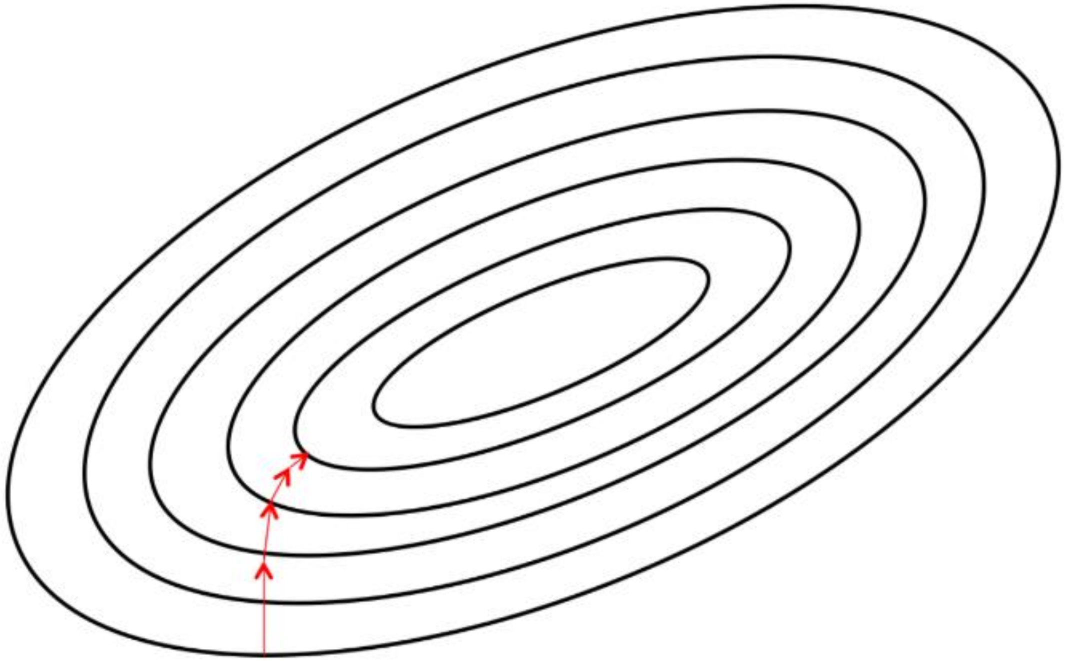


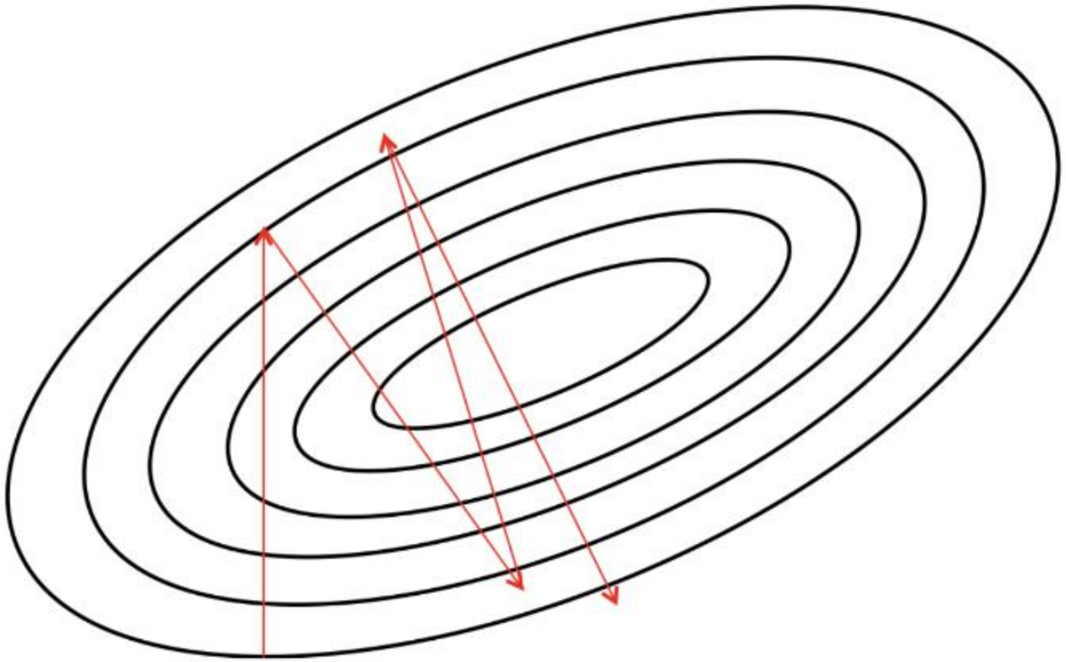

Loss
Overfitting
Validation
Training
Epoch

Hyperparameter optimization

- To find the best hyperparameters, ϵ, and minibatch size
- Use validation dataset
- Grid search, $\epsilon \in \{0.001, 0.01, 0.1\}$, batch size $\in \{16, 64, 128, \ldots\}$
- Use all combinations to find one with best loss value

Hyperparameter ϵ



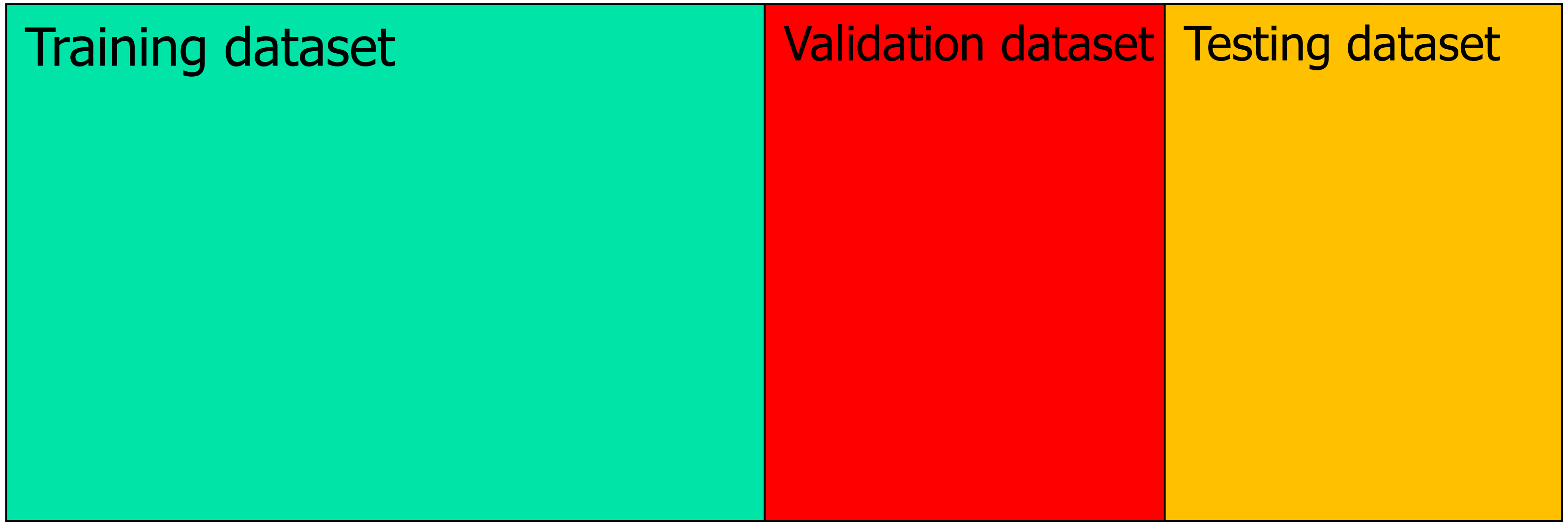


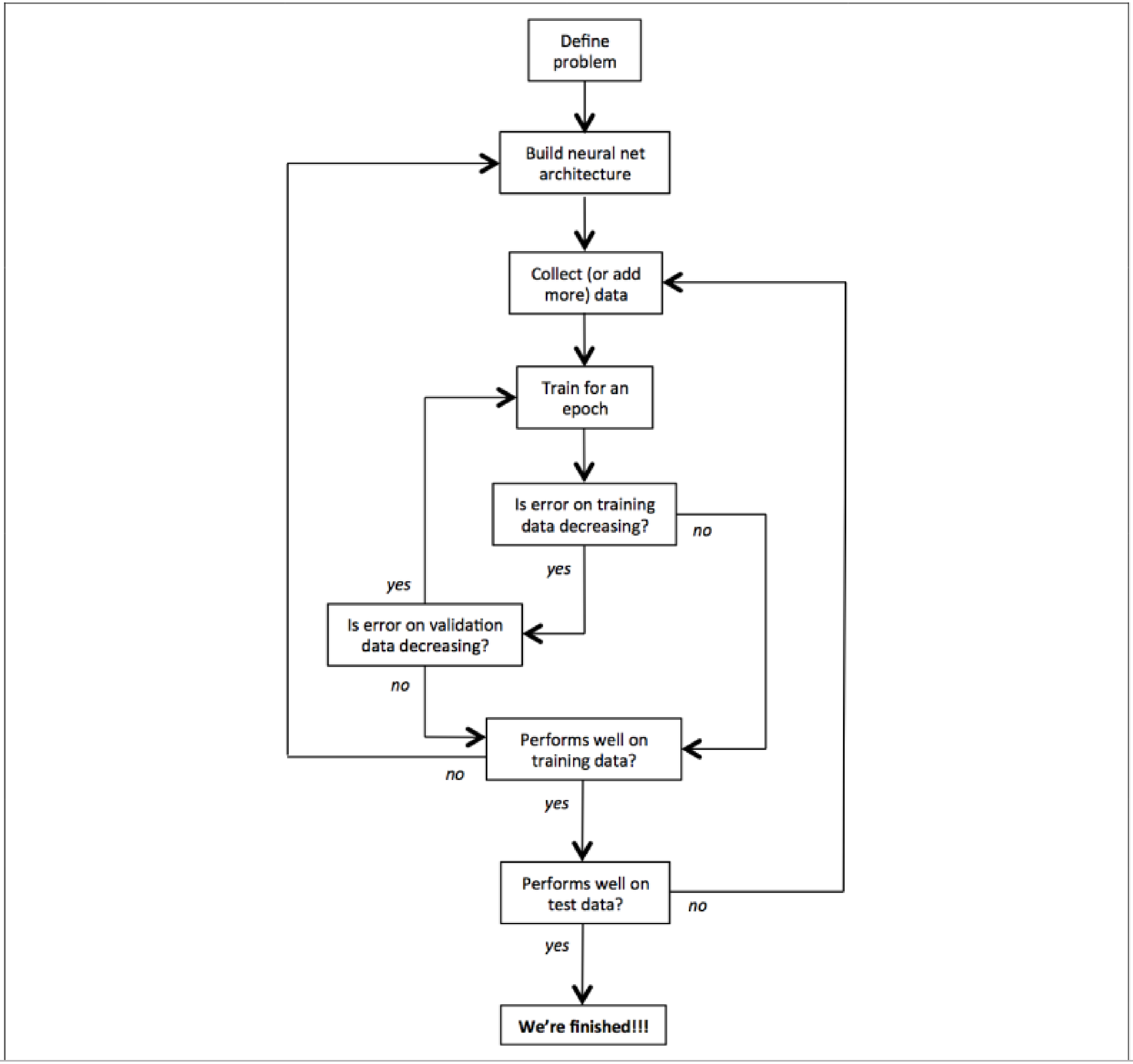
$\in$ too large

The training process

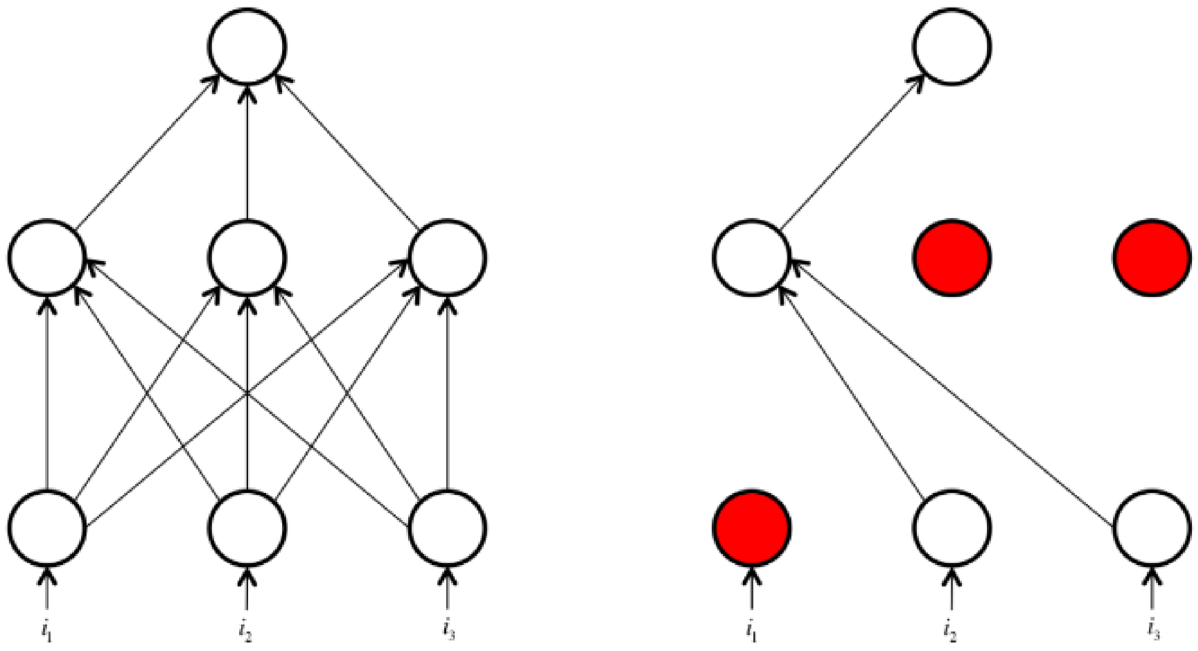
- Separate dataset into training, validation and testing

The Whole dataset




Define problem
Build neural net architecture
Collect (or add more) data
Train for an epoch
Is error on training data decreasing?
no
yes
yes
Is error on validation data decreasing?
no
Performs well on training data?
no
yes
Performs well on test data?
no
yes
We're finished!!!

Dropout (to avoid overfitting)



P: prob. To keep, then output should be divided by p for the node not dropped out

To speed up training dnn

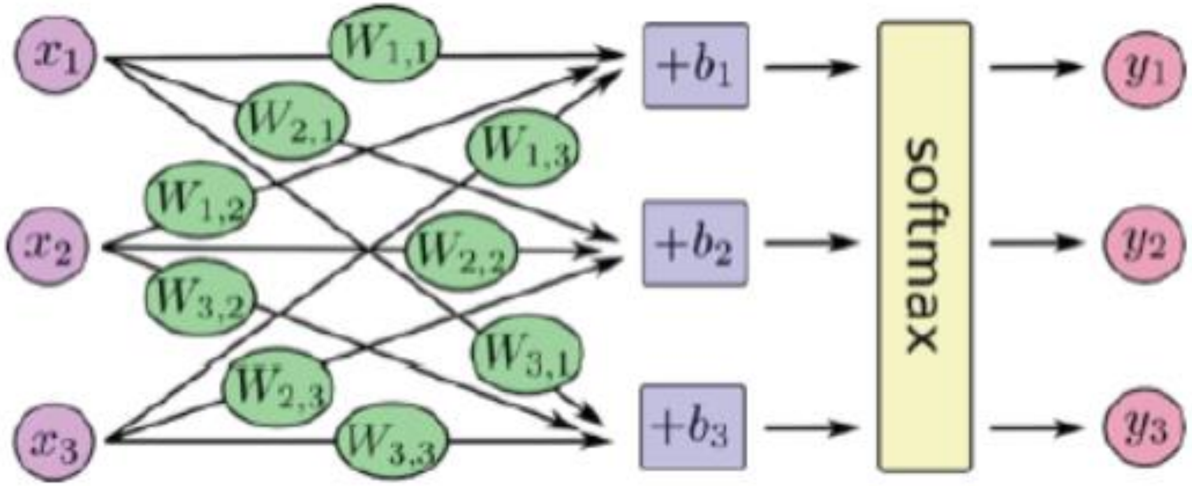
- Local optimum, saddle point
- Second order optimization methods: methods with momentum(SGD+ Momentum), AgaGrad, RMSProp, Adam
- One class for the optimization of DNN

Softmax

- To transform activation into probability, the larger the activation, the high the probability
- $P(y = i|x) = softmax_i(Wx + b) = \frac{e^{w_i^{x}+b_i}}{\sum_j e^{w_j x+b_j}}$

The activation of input i = the amount of input to node i

- Note that the probability of a softmax will never be zero!


x_1
x_2
x_3
$W_{1,1}$
$W_{2,1}$
$W_{1,2}$
$W_{1,3}$
$W_{2,2}$
$W_{3,2}$
$W_{3,1}$
$W_{2,3}$
$W_{3,3}$
$+b_1$
$+b_2$
$+b_3$
softmax
y_1
y_2
y_3

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax} \begin{pmatrix} W_{1,1}x_1 + W_{1,2}x_2 + W_{1,3}x_3 + b_1 \\ W_{2,1}x_1 + W_{2,2}x_2 + W_{2,3}x_3 + b_2 \\ W_{3,1}x_1 + W_{3,2}x_2 + W_{3,3}x_3 + b_3 \end{pmatrix}$$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

```
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

More on backpropagation

- **//x** is an input example, **t** is its corresponding output vector
- For each (**x, t**), in the training examples **Do**
  propagate the input forward though the network

1. input x to the network and compute o_u of every output unit u in the network

2. For each network output unit k, calculate its error term δ_k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \qquad (1)$$

3. For each hidden unit h, calculate its error term δ_h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{hk} \delta_k \qquad (2)$$

4. Update each network weight w_{ij}

$w_{ij} = w_{ij} + \Delta w_{ij}$, where $\Delta w_{ij} = \eta \delta_j x_{ij}$ (3)

More on cross entropy

$$H = \sum_{c=1}^{C} \sum_{i=1}^{n} -y_{c,i} log_2(p_{c,i})$$

P_{ci} is the probability of the predicted i^{th} class, it will never be 0 (softmax output) , otherwise a big problem.
$Y_{c,i}$ is the true class probability (usually, one-hot encoding)

Cross entropy is a measure of how similar two distributions are.

Example

Model 1							
		Predicted prob.			Actual class (one hot encoding)		
		boy	girl	other	boy	girl	other
data1	boy	0.4	0.3	0.3	1	0	0
data2	girl	0.3	0.4	0.3	0	1	0
data3	boy	0.5	0.2	0.3	1	0	0
data4	other	0.8	0.1	0.1	0	0	1
		Error rate= ¼=25% CROSS ENTROPY =6.966					

Boy's cross entropy=-(1*log(0.4)+1*log(0.5)) = 2.322 (taking $\log_2$)
Girl's cross entropy = -(1*log(0.4))=1.322
Other's cross entropy=-(1*log(0.1))=3.322
Overall cross entropy=6.966

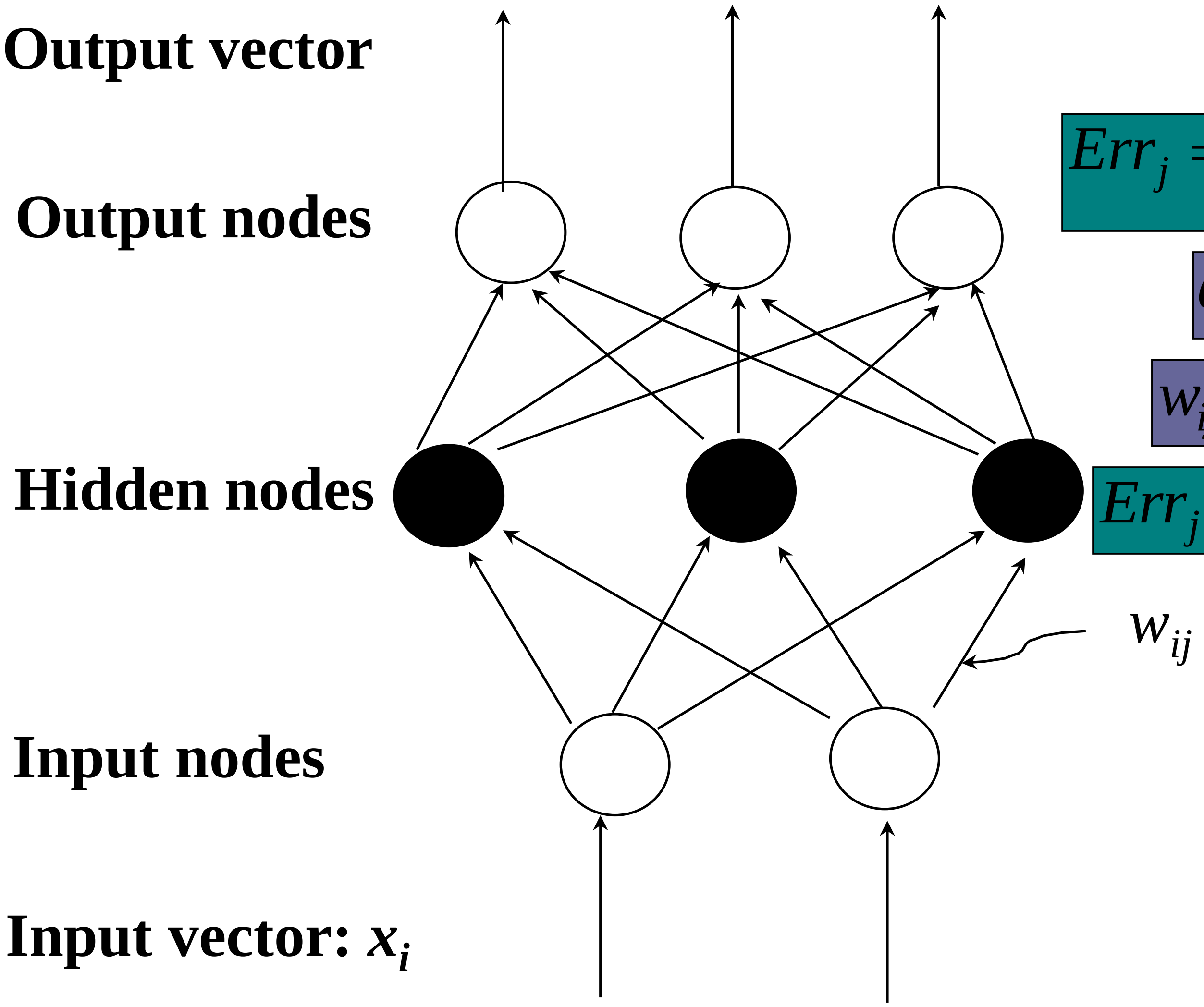
Model 2							
		Predicted prob.			Actual class (one hot encoding)		
		boy	girl	other	boy	girl	other
data1	boy	0.7	0.1	0.2	1	0	0
data2	girl	0.1	0.8	0.1	0	1	0
data3	boy	0.9	0.1	0.0	1	0	0
data4	other	0.4	0.3	0.3	0	0	1
		Error rate= ¼=25% CROSS ENTROPY =2.725					

$=-(1 \times \log(0.7) + 1 \times \log(0.8) + 1 \times \log(0.9) + 1 \times \log(0.3)) = 2.725$

Total = 2.725

Model 2 is better in terms of cross entropy

multiple output nodes, different notation



$$Err_j = O_j(1-O_j)\sum_k Err_k w_{jk}$$

$$\theta_j = \theta_j + (l)Err_j$$

$$w_{ij} = w_{ij} + (l)Err_j O_i$$

$$Err_j = O_j(1-O_j)(T_j - O_j)$$

$$O_j = \frac{1}{1+e^{-I_j}}$$

$$I_j = \sum_i w_{ij} O_i + \theta_j$$

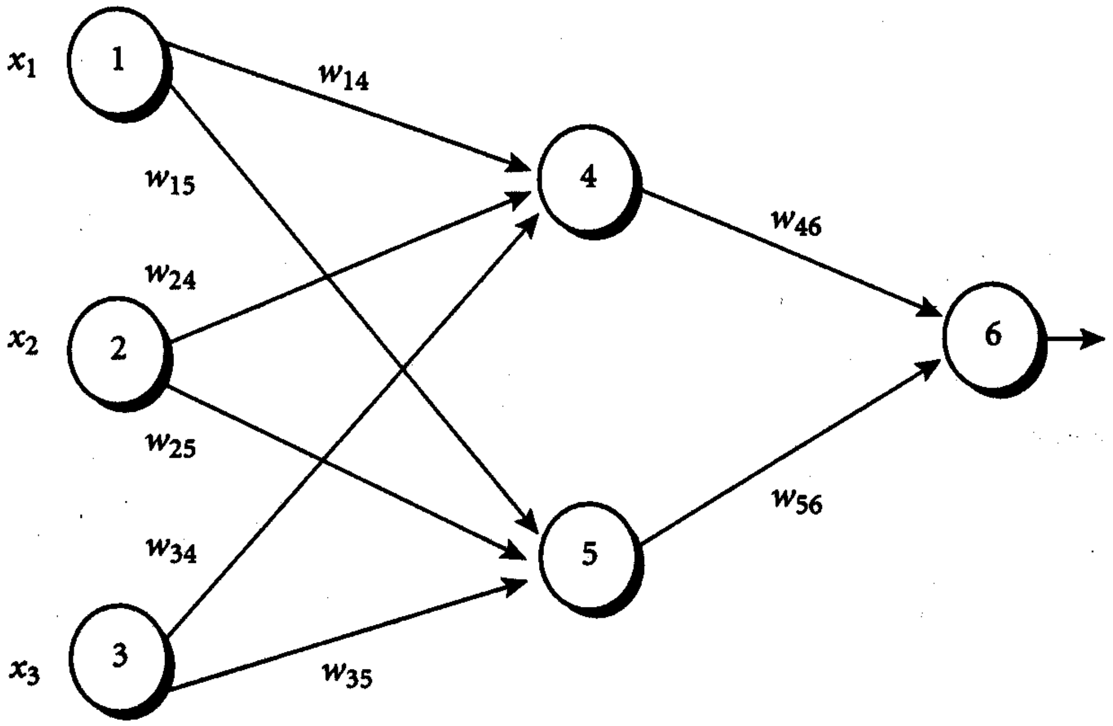


Figure 7.11 An example of a multilayer feed-forward neural network.

Table 7.3 Initial input, weight, and bias values.

x_1	x_2	x_3	w_{14}	w_{15}	w_{24}	w_{25}	w_{34}	w_{35}	w_{46}	w_{56}	θ_4	θ_5	θ_6
1	0	1	0.2	−0.3	0.4	0.1	−0.5	0.2	−0.3	−0.2	−0.4	0.2	0.1

Table 7.4 The net input and output calculations.

Unit j	Net input, I_j	Output, O_j
4	$0.2 + 0 - 0.5 - 0.4 = -0.7$	$1/(1 + e^{0.7}) = 0.332$
5	$-0.3 + 0 + 0.2 + 0.2 = 0.1$	$1/(1 + e^{-0.1}) = 0.525$
6	$(-0.3)(0.332) - (0.2)(0.525) + 0.1 = -0.105$	$1/(1 + e^{0.105}) = 0.474$

Table 7.5 Calculation of the error at each node.

Unit j	Err_j
6	$(0.474)(1-0.474)(1-0.474)=0.1311$
5	$(0.525)(1-0.525)(0.1311)(-0.2)=-0.0065$
4	$(0.332)(1-0.332)(0.1311)(-0.3)=-0.0087$

Table 7.6 Calculations for weight and bias updating.

Weight or bias	*New value*
w_{46}	$-0.3+(0.9)(0.1311)(0.332)=-0.261$
w_{56}	$-0.2+(0.9)(0.1311)(0.525)=-0.138$
w_{14}	$0.2+(0.9)(-0.0087)(1)=0.192$
w_{15}	$-0.3+(0.9)(-0.0065)(1)=-0.306$
w_{24}	$0.4+(0.9)(-0.0087)(0)=0.4$
w_{25}	$0.1+(0.9)(-0.0065)(0)=0.1$
w_{34}	$-0.5+(0.9)(-0.0087)(1)=-0.508$
w_{35}	$0.2+(0.9)(-0.0065)(1)=0.194$
θ_6	$0.1+(0.9)(0.1311)=0.218$
θ_5	$0.2+(0.9)(-0.0065)=0.194$
θ_4	$-0.4+(0.9)(-0.0087)=-0.408$