Artificial Neural Networks

Lecturer

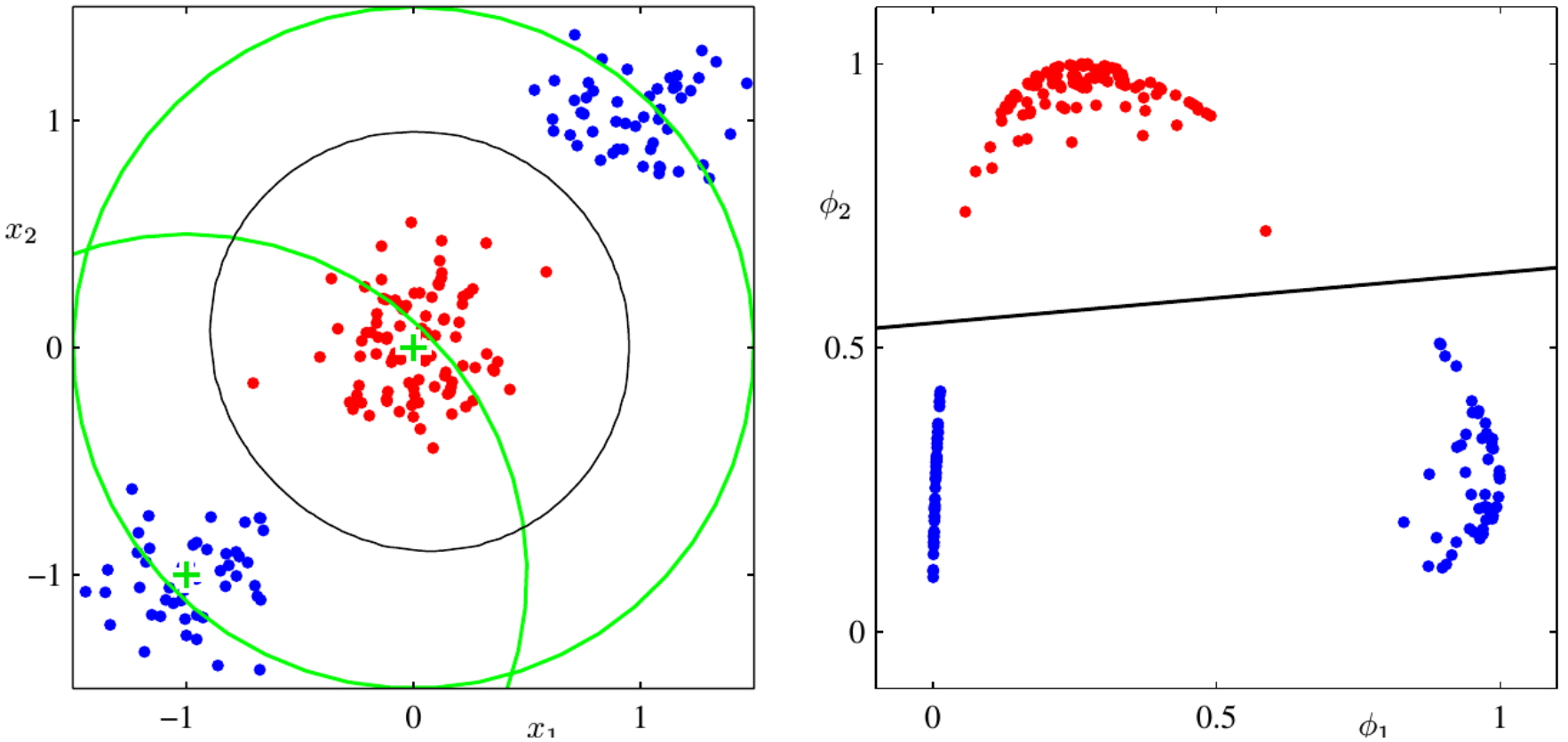
Ali Katanforoush

Shahid Beheshti University

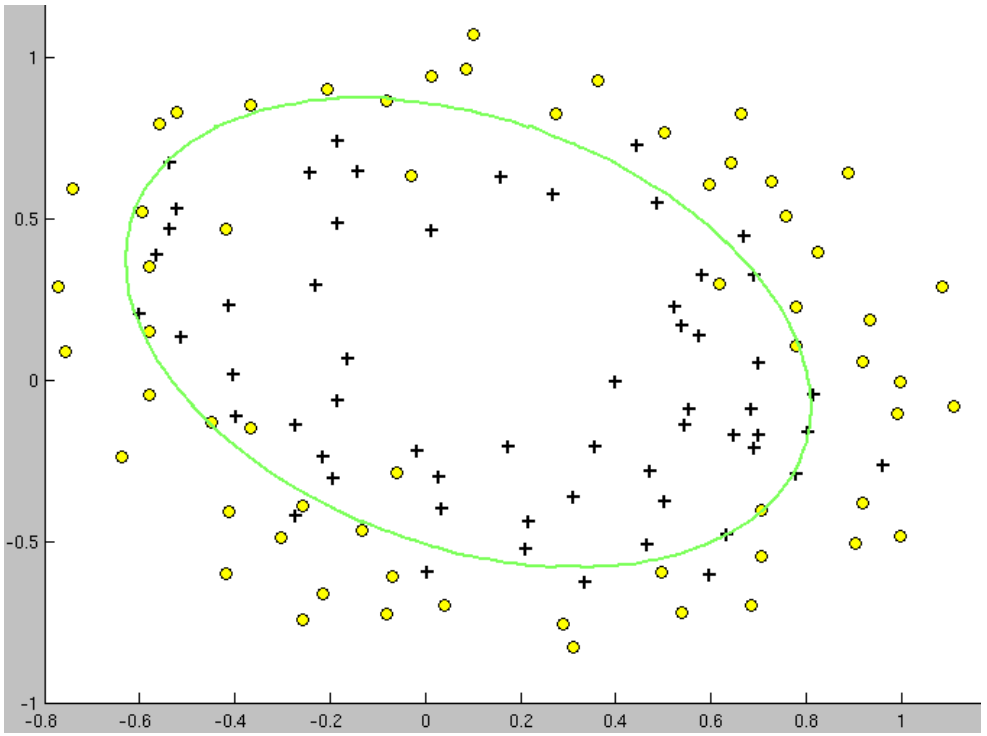
Department of Computer Science

Autumn 2017

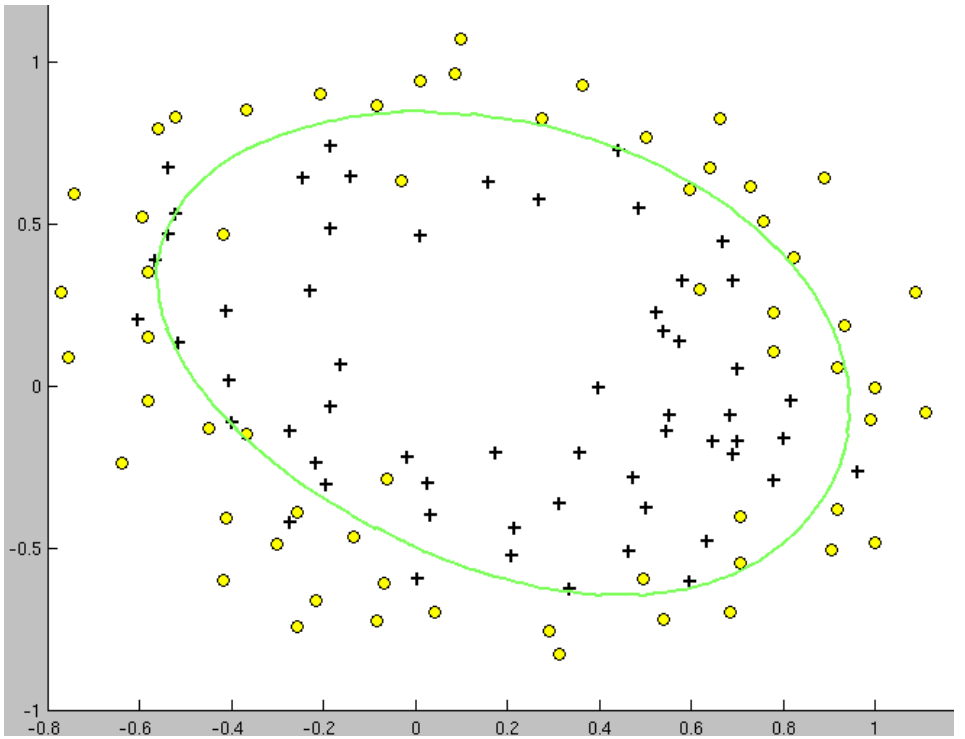
Nonlinear boundary by nonlinear basis Functions



Example



Logistic regression training
(transformed from 6D quadratic form feature space)



Perceptron training
(transformed from 10D cubic form feature space)

Logistic regression

We can directly express the class probability as a sigmoid:

* without implicitly having an underlying Gaussian

Substitute the *"raw input"*, *X*, with some *"higer order features"*, Φ;

$$p(\mathcal{C}_1 \mid \phi) = \sigma(\mathbf{w}^T \phi)$$

$$\sigma(a) = \frac{1}{1+\exp(-a)}$$

Logistic regression

We can directly express the class probability as a sigmoid:

* without implicitly having an underlying Gaussian

Substitute the *"raw input"*, *X*, with some *"higer order features"*, Φ;

$$p(\mathcal{C}_1 \mid \phi) = \sigma(\mathbf{w}^T \phi)$$

The likelihood:

$$p(\mathcal{D}_1 \mid \mathbf{w}) = \prod_{n=1}^{N} y_n^{t_n} (1 - y_n)^{1-t_n} \qquad y_n = \sigma(\mathbf{w}^T \phi(\mathbf{x}_n))$$

Logistic regression

Here we estimate *M* weights via *maximum likelihood approach*;

$$\log p(\mathcal{D} \mid \mathbf{w}) = \sum_{n=1}^{N} \{t_n \log y_n + (1 - t_n) \log(1 - y_n)\}$$

$$\nabla \log p(\mathcal{D} \mid \mathbf{w}) = \sum_{n=1}^{N} \left\{ t_n \frac{y_n' \phi(\mathbf{x}_n)}{y_n} - (1 - t_n) \frac{y_n' \phi(\mathbf{x}_n)}{1 - y_n} \right\}$$

$$= \sum_{n=1}^{N} (t_n - y_n) \phi(\mathbf{x}_n)$$

$$y_n' = y_n(1 - y_n)$$

Logistic regression

Here we estimate *M* weights via *maximum likelihood approach*;

$$\nabla E(\mathbf{w}) = \sum_{n=1}^{N} (y_n - t_n)\boldsymbol{\phi}_n$$

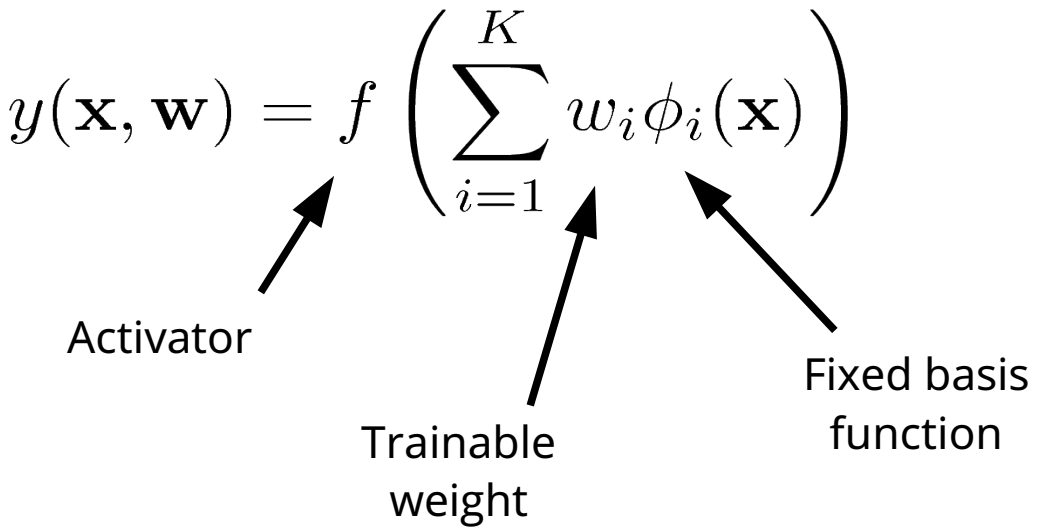
Simple learning rule (*gradient descent*)

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

The Multi-layer Perceptron (MLP)

Motivation

Problem: With our linear methods, we can train the weights but not the basis functions:



Motivation

Problem: With our linear methods, we can train the weights but not the basis functions:

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{i=1}^{K} w_i \phi_i(\mathbf{x})\right)$$

Can we ***learn*** the basis functions?

Motivation

Problem: With our linear methods, we can train the weights but not the basis functions:

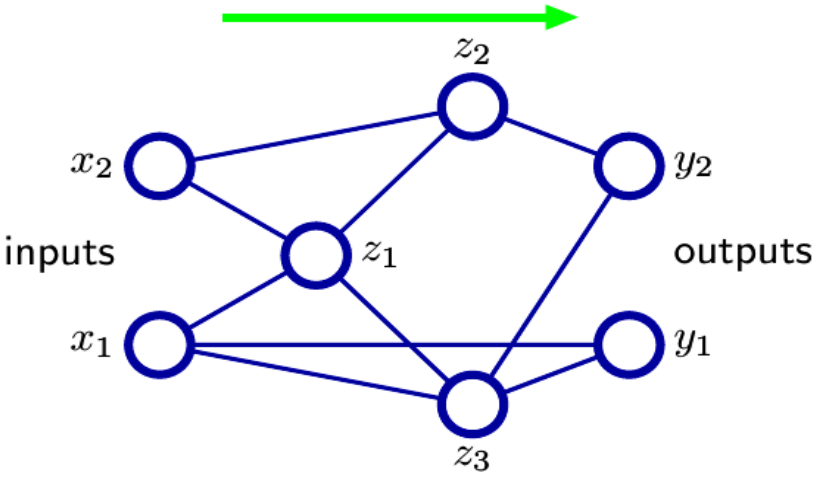
$$y(\mathbf{x}, \mathbf{w}, \{\mathbf{w}^i\}) = f\left(\sum_{i=1}^{K} w_i \phi_i(\mathbf{x}, \mathbf{w}^i)\right)$$

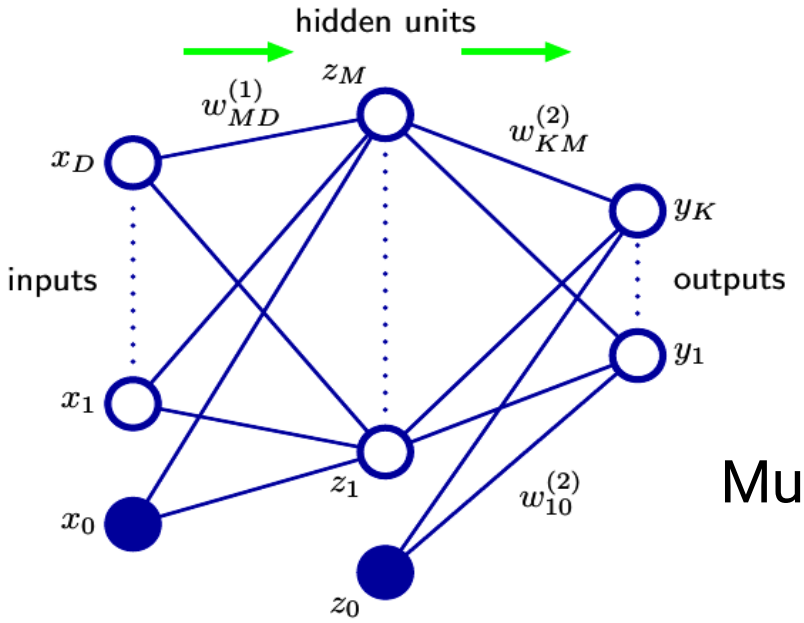
Can we ***learn*** the basis functions?

Maybe reuse ideas from our previous technology? Adjustable weight vectors?

Neural Networks

Feed-forward
(a typical example)





Multi Layer Perceptron

Feed-forward neural network

Two levels of “linear regression” (or classification):

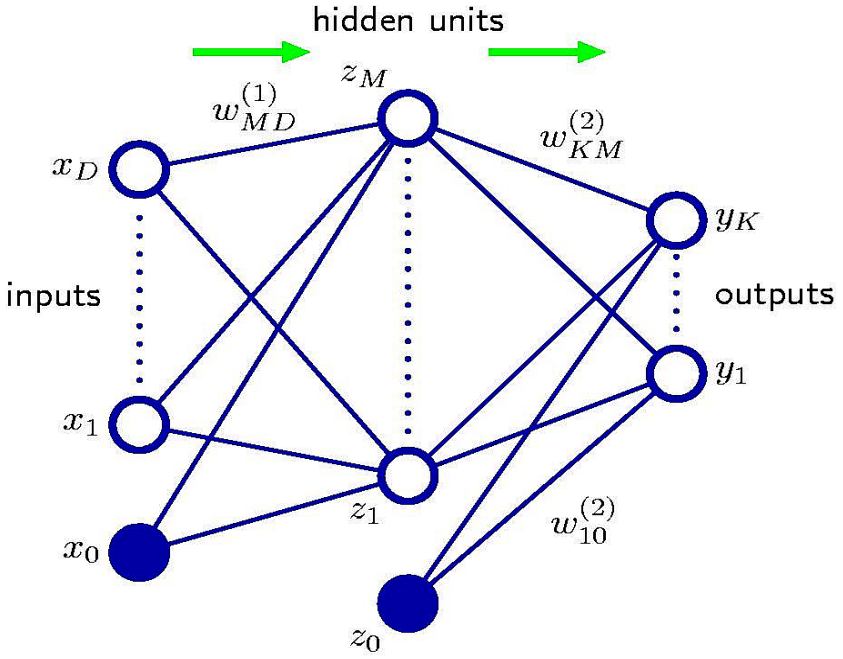
$$y(\mathbf{x}, \mathbf{w}^{(1)}, \mathbf{w}^{(2)}) = f\left(\sum_{i=1}^{K} w_i^{(2)} z_i\right)$$

Output layer

$$z_i = h\left(\sum_{j=1}^{M} w_{ij}^{(1)} x_j\right)$$

Hidden layer

Multi Layer Perceptron



$$y_K(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_{kj}^{(2)} \cdot h\left(\sum_{i=0}^{D} w_{ji}^{(1)} \cdot x_i\right)\right)$$

Activation Function

- Note 1. Nested application of the identity function (even any linear transformation) DOES NOT yield any more stronger model than linear models.
- Note 2. Activation of output layer also DOES NOT form the intrinsic shape of decision boundary. So, use a function simply interpreting the desired output.
- Note 3. Activation of hidden layer has a crucial effect on the shape of decision boundary, consequently the power of classifier.

$$y_K(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_{kj}^{(2)} \cdot h\left(\sum_{i=0}^{D} w_{ji}^{(1)} \cdot x_i\right)\right)$$

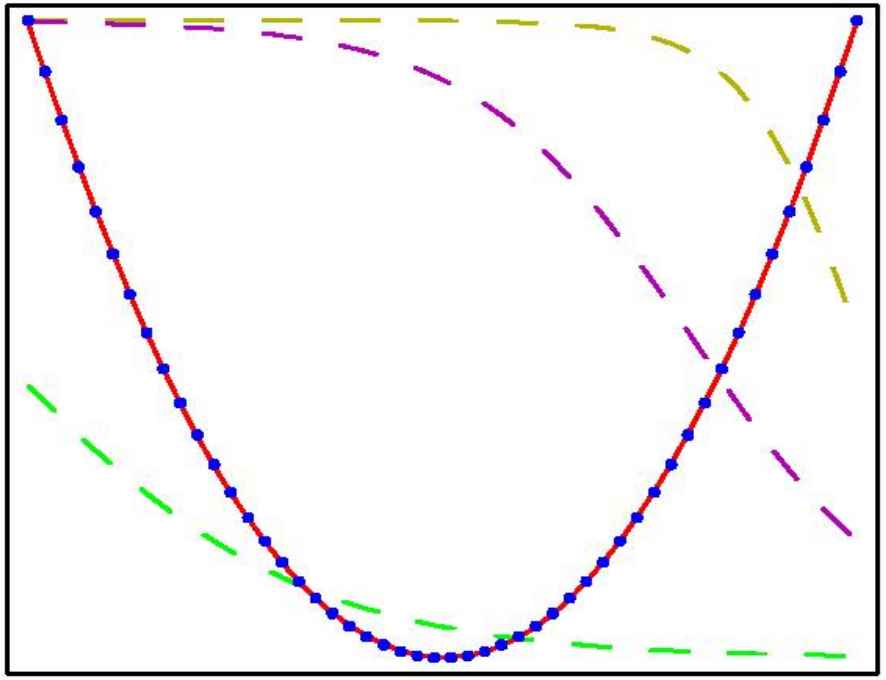
Activation Function

- Note 3. Activation of hidden layer has a crucial effect on the shape of decision boundary, consequently the power of classifier.
  - Sigmoid-like functions are adequate for classification and regression problems. (Logit, tanh, erf, sign, . . .).
    - Note. Logit and tanh are linearly equivalent.
  - Locally supported functions are adequate for regression and clustering problems. (RBFs like Gaussian, multi-quadratic).

$$y_K(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_{kj}^{(2)} \cdot h\left(\sum_{i=0}^{D} w_{ji}^{(1)} \cdot x_i\right)\right)$$

Example

$$y(x) = x^2$$

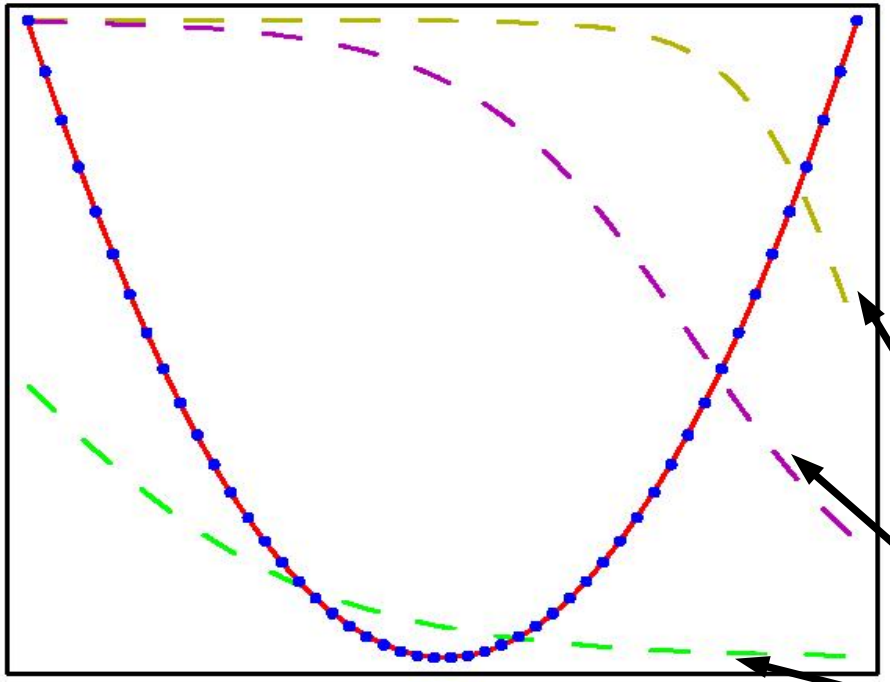


tanh activation function for 3 hidden layers

Linear activation function for output layer

Example

$$y(x) = x^2$$



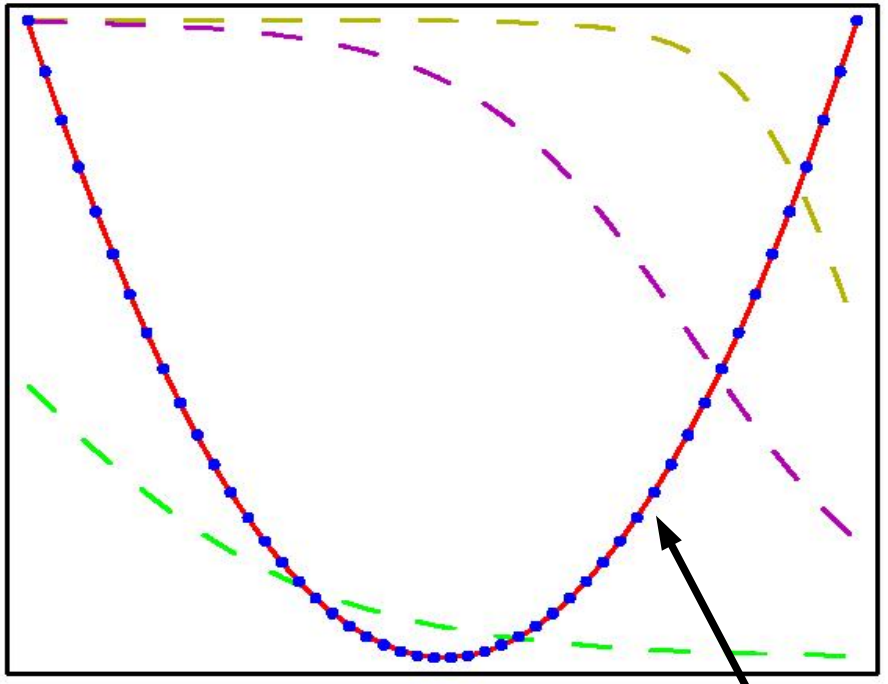
tanh activation function for 3 hidden layers

Linear activation function for output layer

$$z_i = \tanh\left(w_{1i}^{(1)} x + w_{0i}^{(1)}\right)$$

Example

$$y(x) = x^2$$



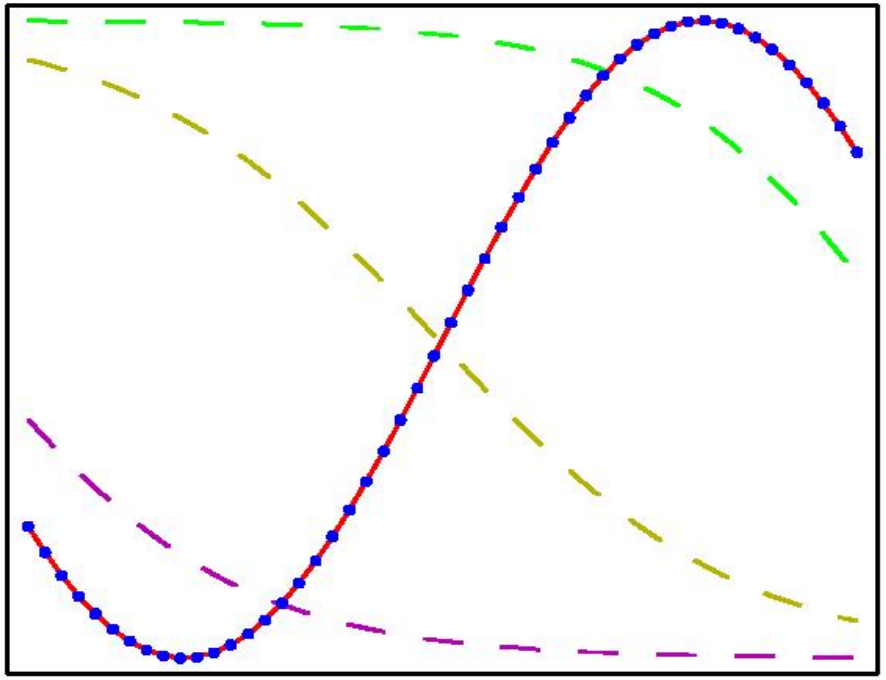
tanh activation function for 3 hidden layers

Linear activation function for output layer

$$y = w_0^{(2)} + w_1^{(2)} \cdot z_1 + w_2^{(2)} \cdot z_2 + w_3^{(2)} \cdot z_3$$

Example

$$y(x) = \sin(x)$$

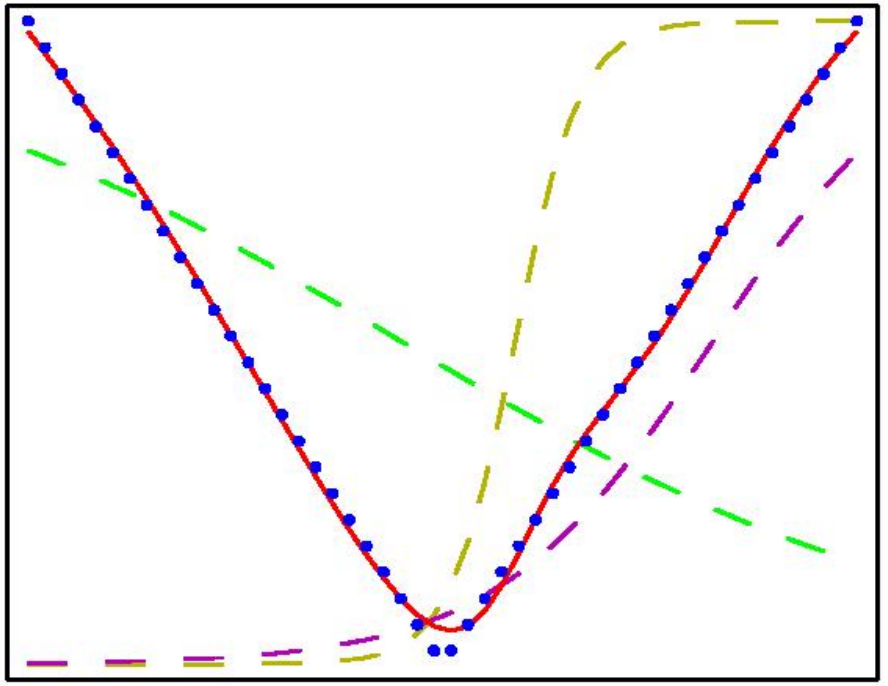


tanh activation function for 3 hidden layers

Linear activation function for output layer

Example

$$y(x) = |x|$$

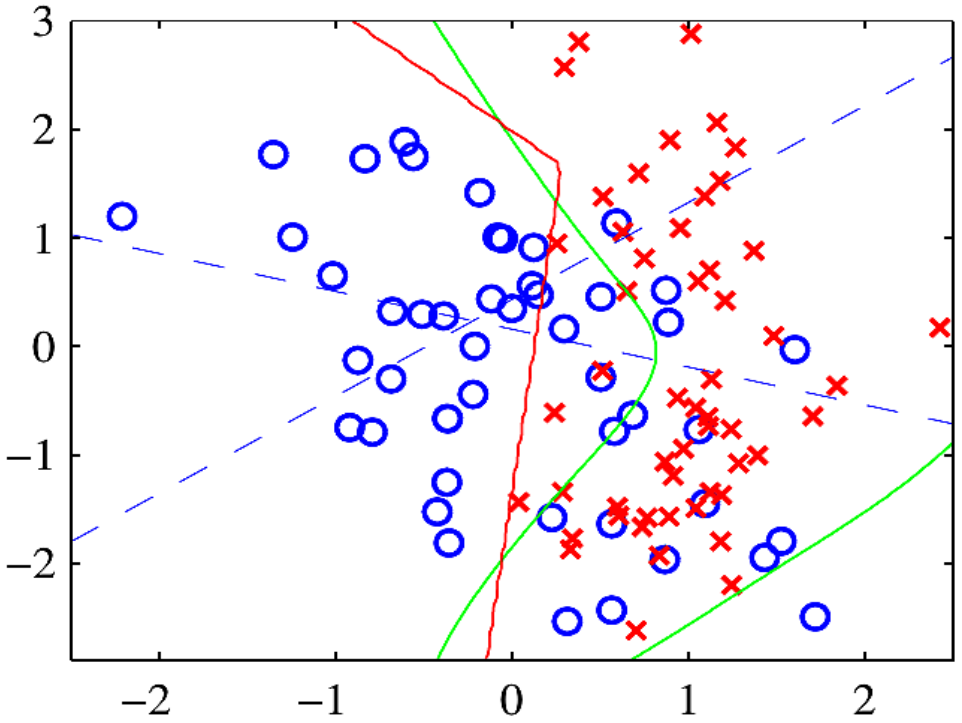


tanh activation function for 3 hidden layers

Linear activation function for output layer

Example

Example of the solution of a simple two-class classification problem involving synthetic data using a neural network having two inputs, two hidden units with 'tanh' activation functions, and a single output having a logistic sigmoid activation function. The dashed blue lines show the $z = 0.5$ contours for each of the hidden units, and the red line shows the $y = 0.5$ decision surface for the network. For comparison, the green line denotes the optimal decision boundary computed from the distributions used to generate the data.



Network training

If we can give the network a probabilistic interpretation, we can use our “standard” estimation methods for training.

Network training

If we can give the network a probabilistic interpretation, we can use our “standard” estimation methods for training.

Regression problems:

$$p(t \mid \mathbf{x}, \mathbf{w}) = \mathcal{N}\left(t \mid y(\mathbf{x}, \mathbf{w}), \sigma^2\right)$$

$$\hat{\mathbf{w}}_{\mathrm{ML}} = \underset{\mathbf{w}}{\operatorname{argmax}}\, p(t \mid \mathbf{x}, \mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}}\, E(\mathbf{w})$$

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2$$

Network training

If we can give the network a probabilistic interpretation, we can use our “standard” estimation methods for training.

Classification problems:

$$p(t \mid \mathbf{x}, \mathbf{w}) = y(\mathbf{x}, \mathbf{w})^t \left\{1 - y(\mathbf{x}, \mathbf{w})\right\}^{1-t}$$

$$y = \sigma(a)$$

$$\hat{\mathbf{w}}_{\mathrm{ML}} = \underset{\mathbf{w}}{\operatorname{argmax}}\, p(t \mid \mathbf{x}, \mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}}\, E(\mathbf{w})$$

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \left\{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\right\}$$

Network training

If we can give the network a probabilistic interpretation, we can use our “standard” estimation methods for training.

Training by minimizing an ***error function***.

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2$$

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\{t_n \ln y_n + (1 - t_n)\ln(1 - y_n)\}$$

Network training

If we can give the network a probabilistic interpretation, we can use our “standard” estimation methods for training.

Training by minimizing an ***error function***.

Typically, we cannot minimize analytically but need numerical optimization algorithms.

$$E(\mathbf{w}) = \frac{1}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2$$

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\{t_n \ln y_n + (1 - t_n)\ln(1 - y_n)\}$$

Network training

If we can give the network a probabilistic interpretation, we can use our “standard” estimation methods for training.

Training by minimizing an ***error function***.

Typically, we cannot minimize analytically but need numerical optimization algorithms.

Notice: There will typically be more than one minimum of the error function (due to symmetries).

Notice: At best, we can find local minima.

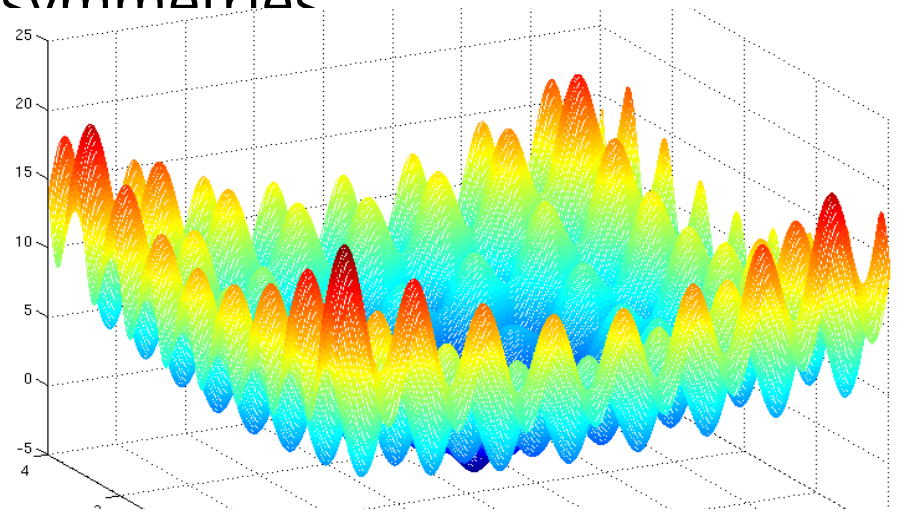
Parameter optimization

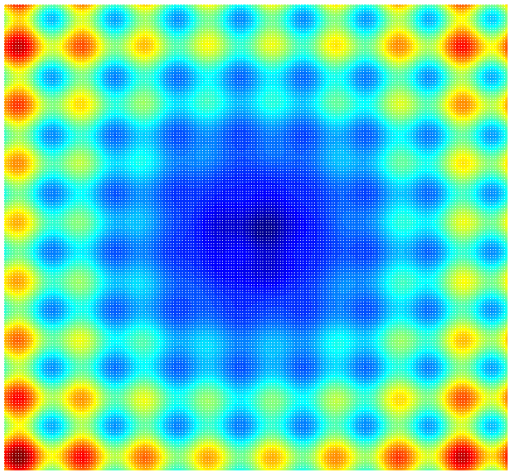
Numerical optimization is beyond the scope of this class – you can (should) get away with just using appropriate libraries.

$$\hat{\mathbf{w}} = \text{minimize}(E, \mathbf{w}_0)$$

Weight-space Symmetries

- Dealing with optimization problems in high dimensional spaces poses as an impractical approach!
  - In high dimension spaces, functions with even simple formulation can have many local minimas.
- BUT good news for MLP! The weight-space is highly symmetrical => many local minimas are identical wrt symmetries

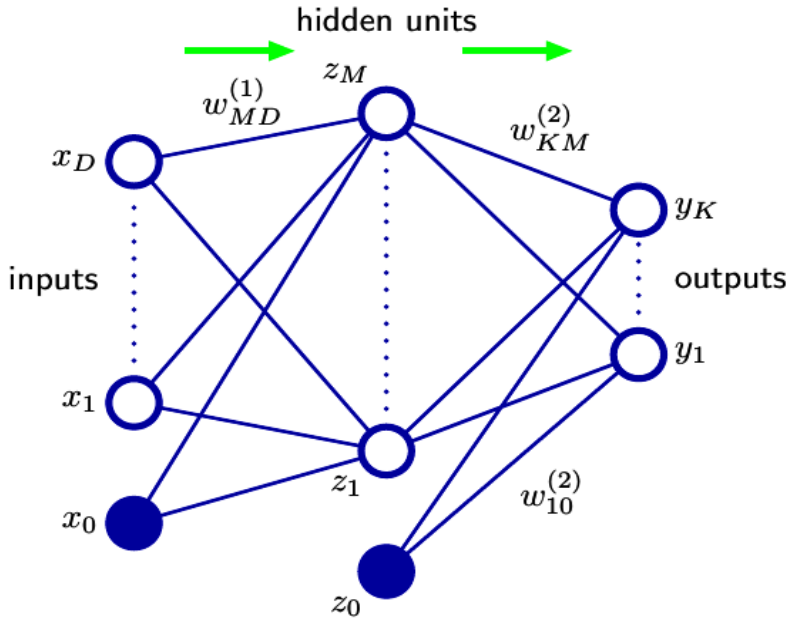




Weight-space Symmetries

- MLP training is insensitive wrt arrangement of neuron in hidden layer.
  - If net is optimal with weights $\left(w_{i1}^{(1)},\ldots,w_{iD}^{(1)}\right),\ldots,\left(w_{j1}^{(1)},\ldots,w_{jD}^{(1)}\right),\ldots$ $\left(w_{1\mathrm{i}}^{(2)},\ldots,w_{Ki}^{(2)}\right),\ldots,\left(w_{1\mathrm{j}}^{(2)},\ldots,w_{Kj}^{(2)}\right)$

    then equivalently is optimal with



$$\left(w_{j1}^{(1)},\ldots,w_{jD}^{(1)}\right),\ldots,\left(w_{i1}^{(1)},\ldots,w_{iD}^{(1)}\right),\ldots,$$
$$\left(w_{1\mathrm{j}}^{(2)},\ldots,w_{Kj}^{(2)}\right),\ldots,\left(w_{1\mathrm{i}}^{(2)},\ldots,w_{Ki}^{(2)}\right)$$

MLP is invariant to every permutations on hidden layer => each local minima is equiv. to $M!$ minimas else.

Weight-space Symmetries

$$y_K(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j=0}^{M} w_{kj}^{(2)} \cdot h\left(\sum_{i=0}^{D} w_{ji}^{(1)} \cdot x_i\right)\right)$$

- If an odd function is used as hidden layer activation function (as tanh is so), then MLP is not changed if all in-weights and out-weights of any hidden neuron be negated.
  - If net is optimal with weights $(w_{i1}^{(1)}, \ldots, w_{iD}^{(1)}), \ldots$ $(w_{1\mathrm{i}}^{(2)}, \ldots, w_{Ki}^{(2)})$

    then equivalently is optimal with $(-w_{i1}^{(1)}, \ldots, -w_{iD}^{(1)}), \ldots$ $(-w_{1\mathrm{i}}^{(2)}, \ldots, -w_{Ki}^{(2)})$

MLP is invariant to weight-negation on every hidden neuron

=> each local minima is equiv. to 2^M minimas else => $2^M M!$

Back propagation

Back propagation is an efficient algorithm for computing both the error function and the gradient.

Back propagation

Back propagation is an efficient algorithm for computing both the error function and the gradient.

For iid data:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

$$\nabla E(\mathbf{w}) = \sum_{k=1}^{K} \frac{\partial E}{\partial w_k} = \sum_{k=1}^{K} \left(\sum_{n=1}^{N} \frac{\partial E_n}{\partial w_k} \right)$$

Back propagation

Back propagation is an efficient algorithm for computing both the error function and the gradient.

For iid data:

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

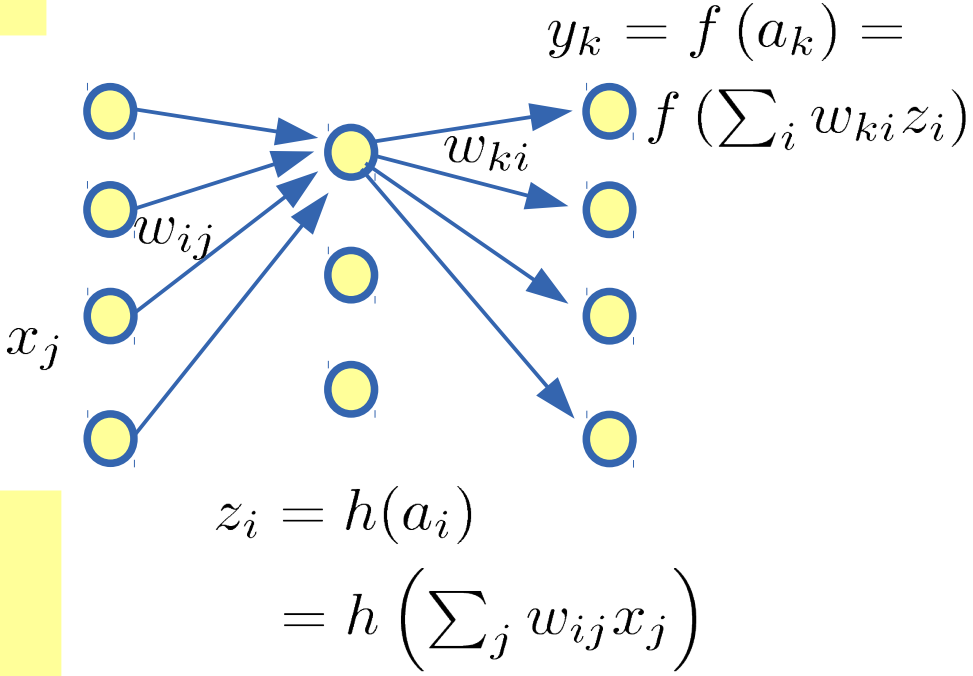
$$\nabla E(\mathbf{w}) = \sum_{k=1}^{K} \frac{\partial E}{\partial w_k} = \sum_{k=1}^{K} \left(\sum_{n=1}^{N} \frac{\partial E_n}{\partial w_k} \right)$$

We just focus on this term

Back propagation

After running the feed-forward algorithm, we know the values z_i and y_k.

$$\frac{\partial E_n}{\partial w_{ki}} = \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial w_{ki}}$$
$$= \delta_k z_i$$



we need to compute the δ s *(errors).*

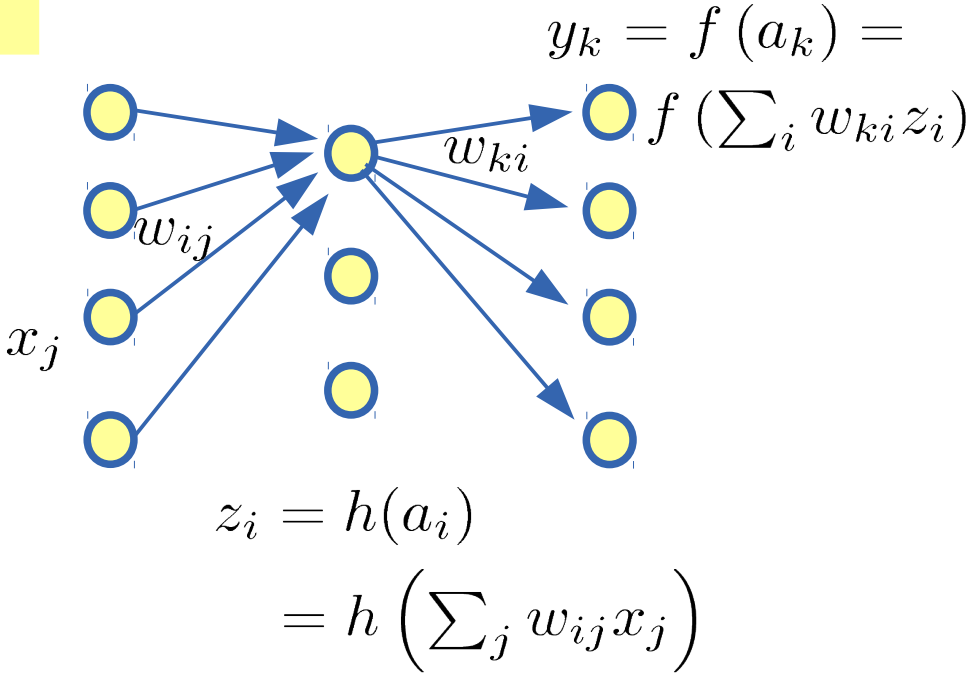
Back propagation

The choice of error function and output activator typically makes it simple to deal with the output layer

$$y_k = f(a_k) = a_k$$

$$E_n(\mathbf{w}) = \frac{1}{2}\sum_k \{y_{kn} - t_{kn}\}^2$$

$$\delta_k = \frac{\partial E_n}{\partial a_k} = (y_{kn} - t_{kn}) \cdot 1$$



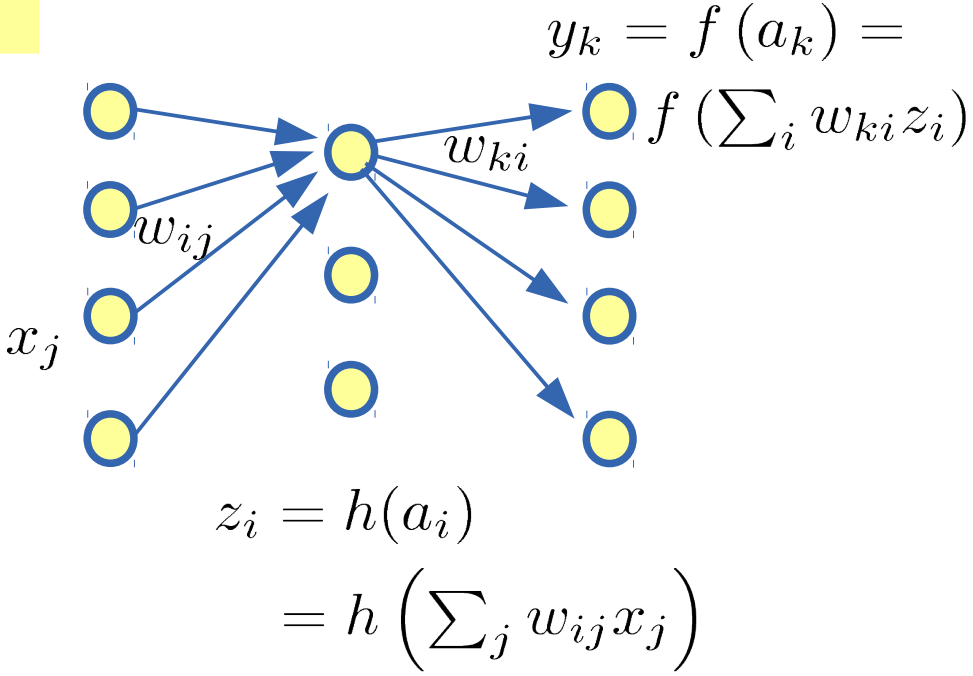
Back propagation

The choice of error function and output activator typically makes it simple to deal with the output layer

$$y_k = f(a_k) = \frac{\exp(a_k)}{\sum_l \exp(a_l)}$$

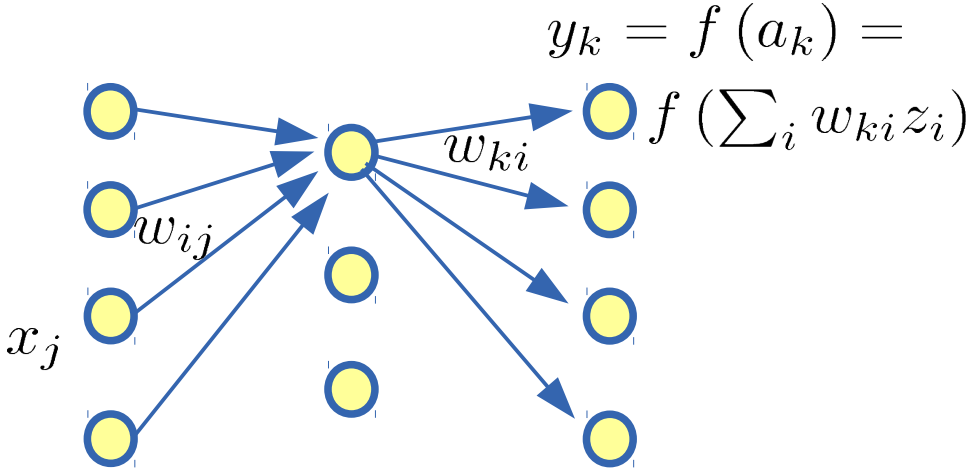
$$E_n(\mathbf{w}) = -\sum_k \underbrace{\mathbf{1}(t_n == k)}_{\text{one-hot-encoding: } t_{kn}} \ln(y_{kn})$$

$$\delta_k = \frac{\partial E_n}{\partial a_k} = (y_{kn} - t_{kn})$$



Back propagation

For hidden layers, we apply the chain rule



$$\delta_i = \frac{\partial E_n}{\partial a_i} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_i}$$

$$= \sum_k \delta_k \frac{\partial a_k}{\partial a_i} = \sum_k \delta_k \frac{\partial a_k}{\partial z_i}\frac{\partial z_i}{\partial a_i}$$

$$= \sum_k \delta_k w_{ki} \frac{\partial z_i}{\partial a_i} = h'(a_i) \sum_k w_{ki}\delta_k$$

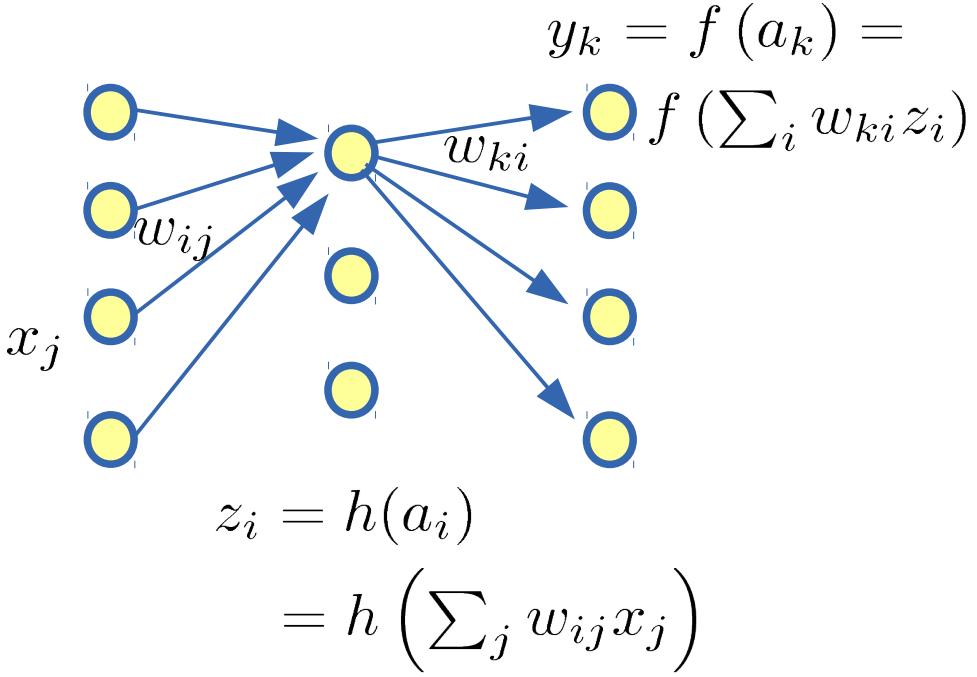
Calculating error and gradient

Algorithm

1. Apply feed forward to calculate hidden and output variables and then the error.

2. Calculate output errors and propagate errors back to get the gradient.

3. Iterate ad lib.



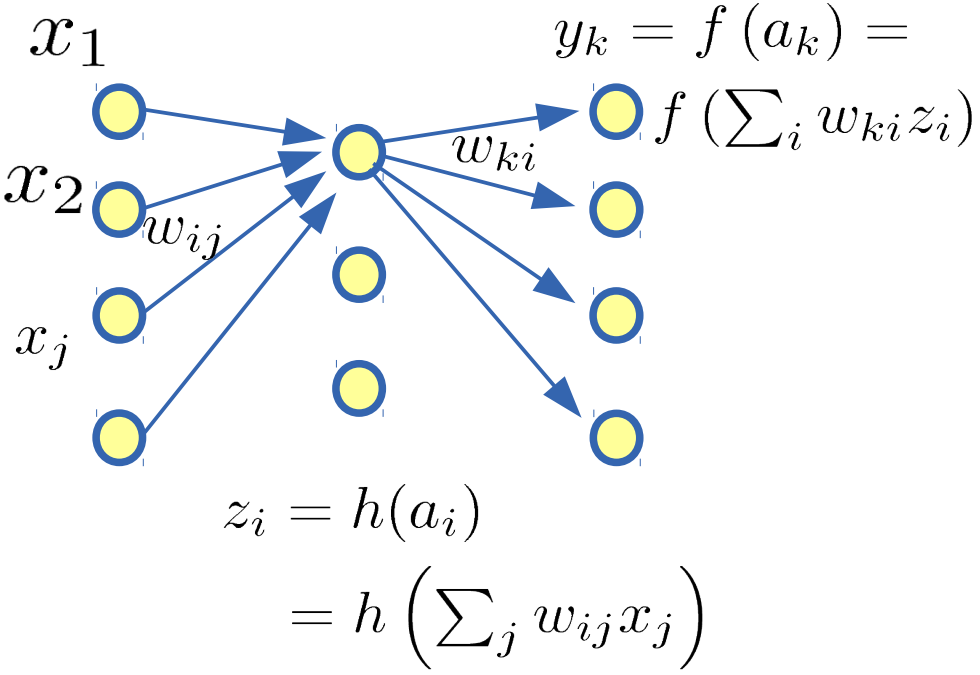
Calculating error and gradient

Algorithm

1. Apply feed forward to calculate hidden and output variables and then the error.

2. Calculate output errors and propagate errors back to get the gradient.

3. Iterate ad lib.



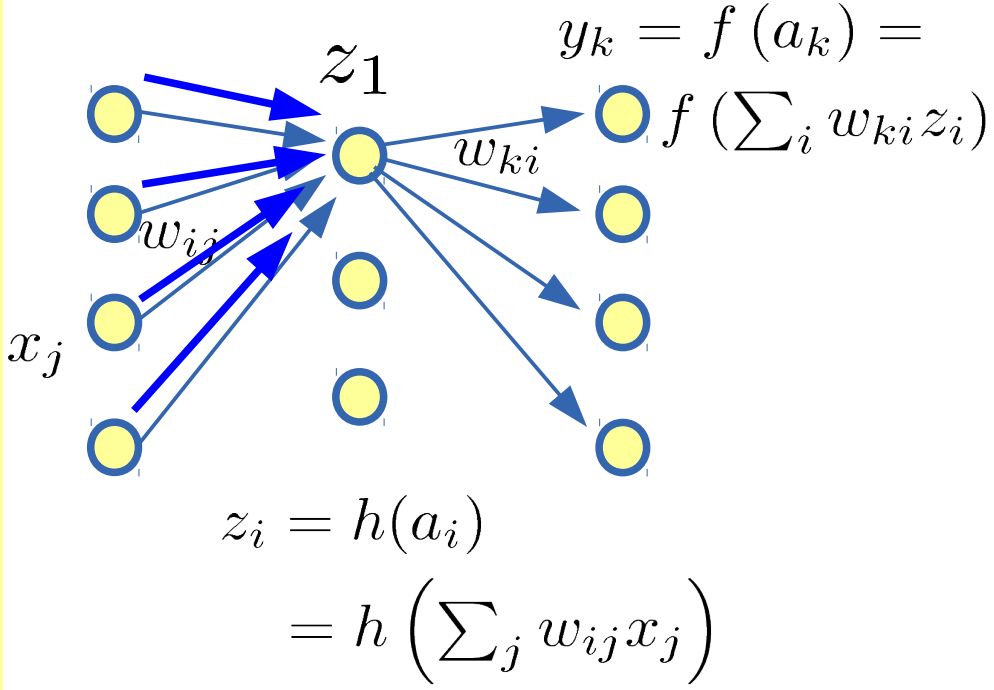
Calculating error and gradient

Algorithm

1. Apply feed forward to calculate hidden and output variables and then the error.

2. Calculate output errors and propagate errors back to get the gradient.

3. Iterate ad lib.



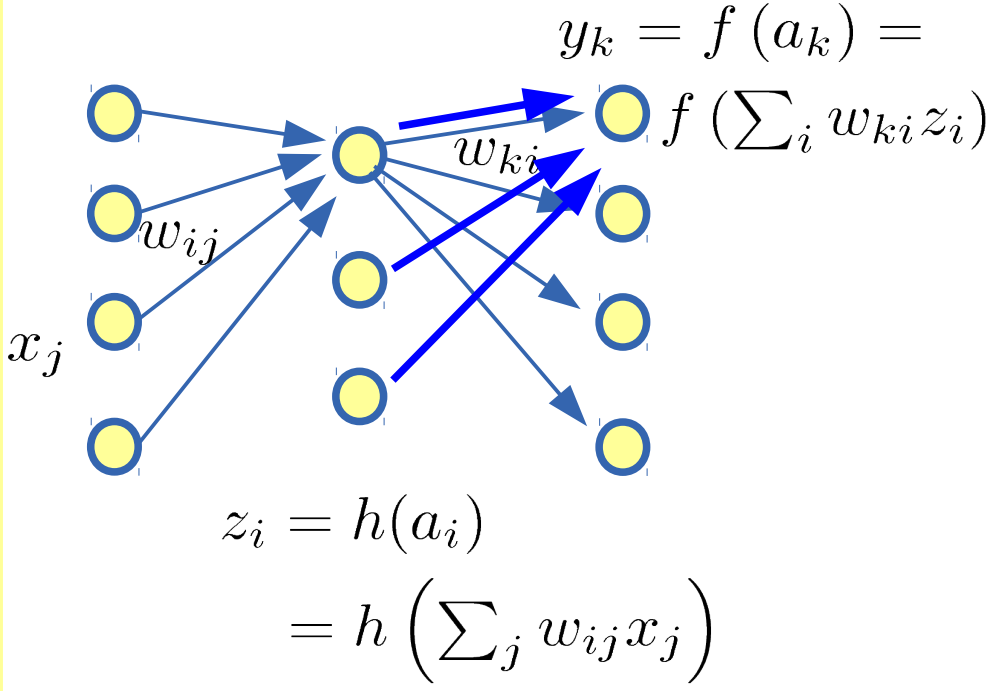
Calculating error and gradient

Algorithm

1. Apply feed forward to calculate hidden and output variables and then the error.

2. Calculate output errors and propagate errors back to get the gradient.

3. Iterate ad lib.



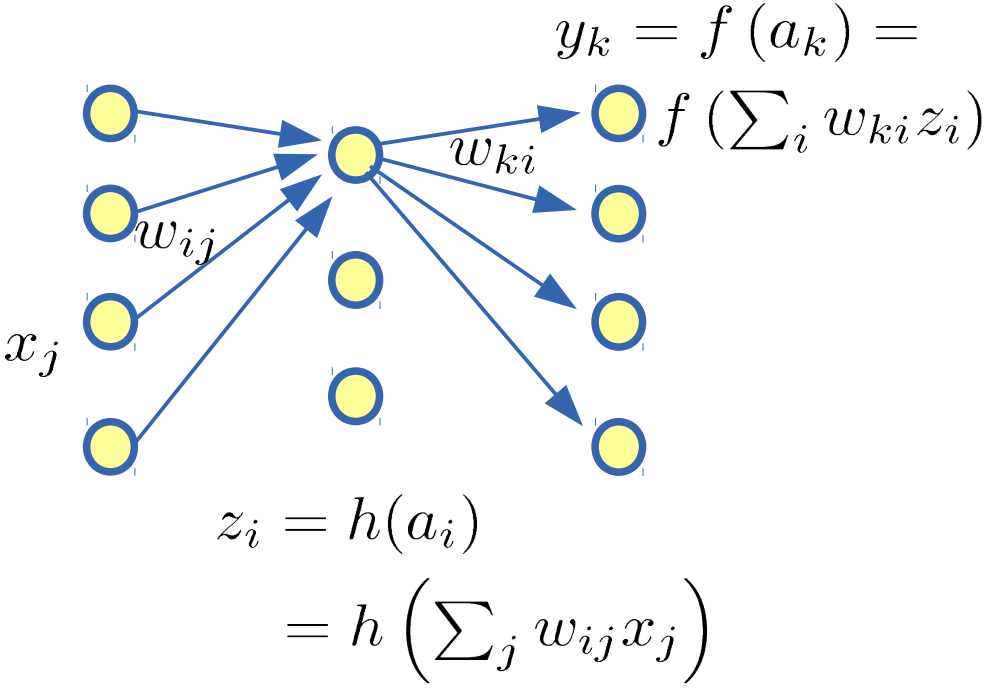
Calculating error and gradient

Algorithm

1. Apply feed forward to calculate hidden and output variables and then the error.

2. Calculate output errors and propagate errors back to get the gradient.

3. Iterate ad lib.



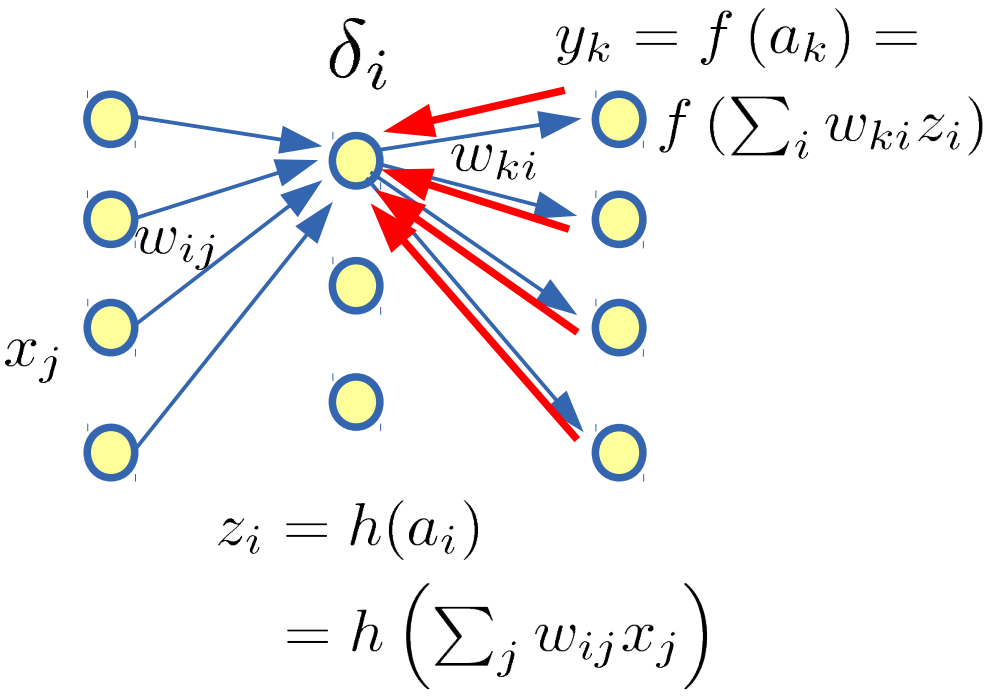
Calculating error and gradient

Algorithm

1. Apply feed forward to calculate hidden and output variables and then the error.

2. Calculate output errors and propagate errors back to get the gradient.

3. Iterate ad lib.



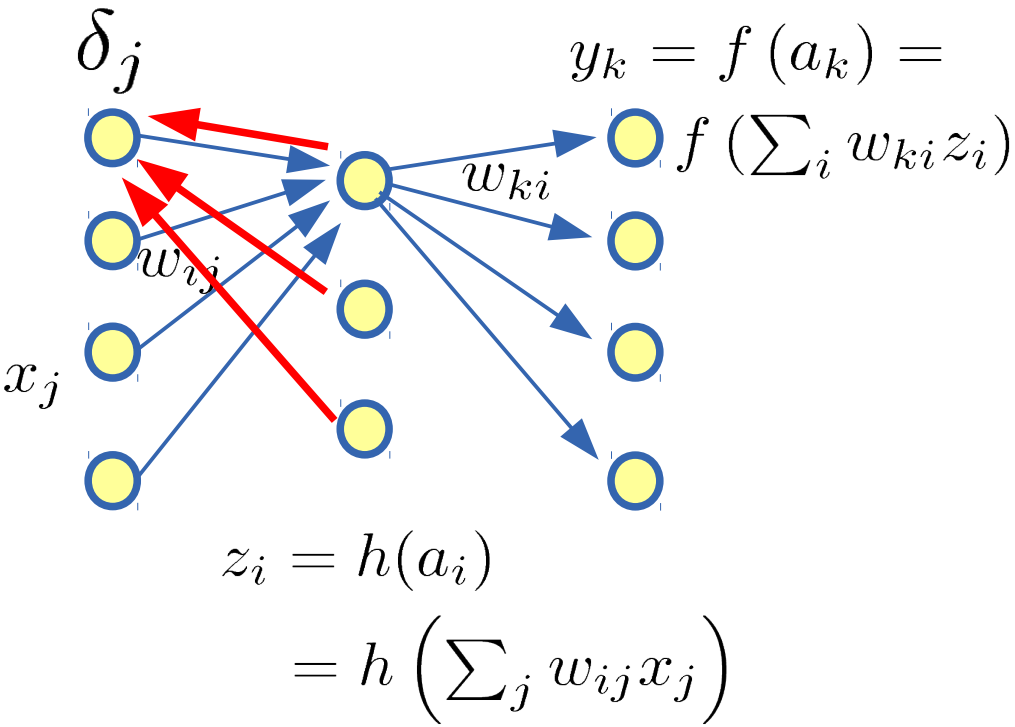
Calculating error and gradient

Algorithm

1. Apply feed forward to calculate hidden and output variables and then the error.

2. Calculate output errors and propagate errors back to get the gradient.

3. Iterate ad lib.



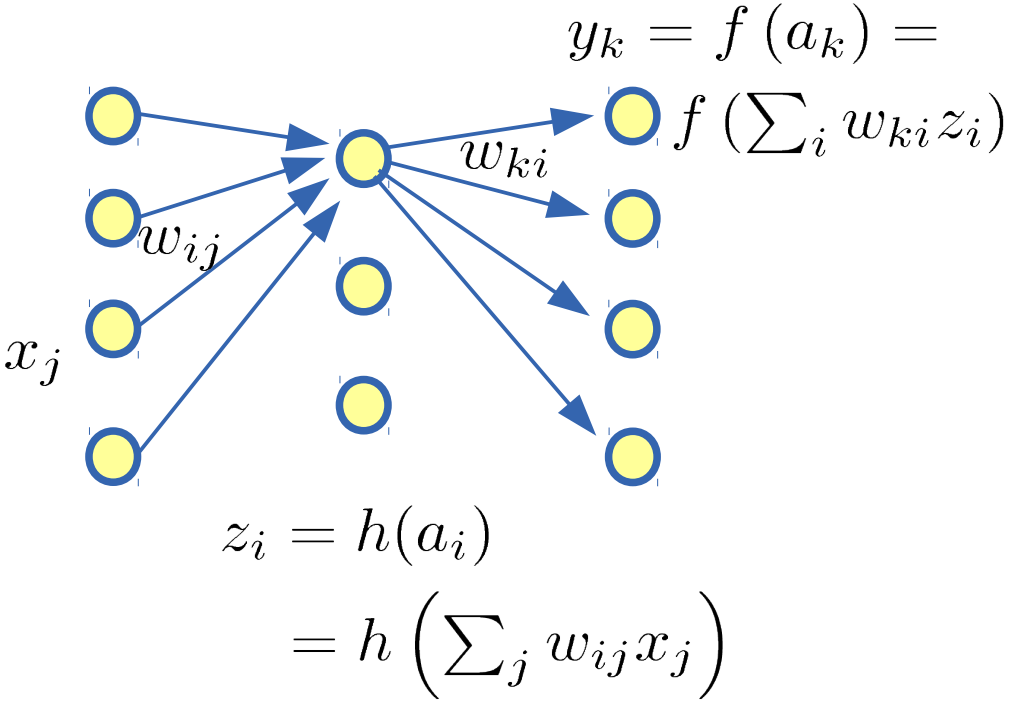
Calculating error and gradient

Algorithm

1. Apply feed forward to calculate hidden and output variables and then the error.

2. Calculate output errors and propagate errors back to get the gradient.

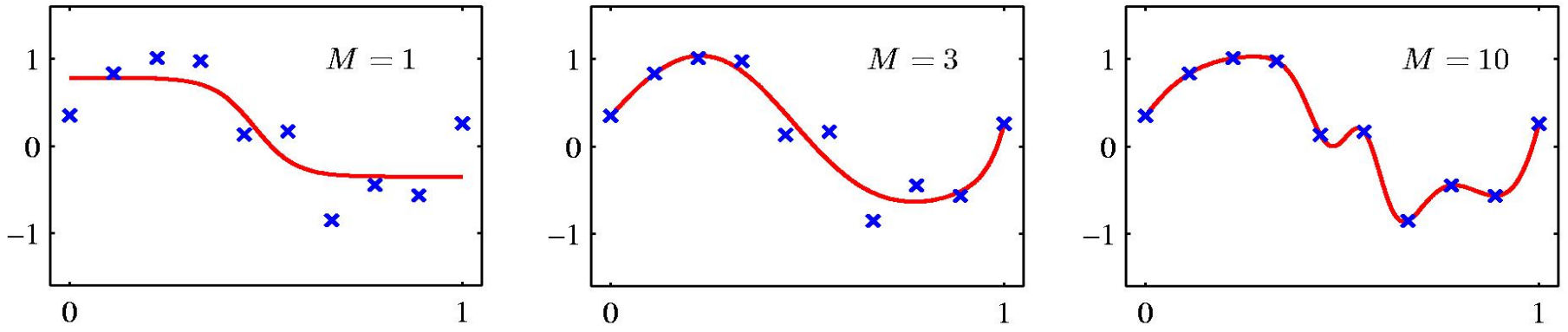
3. Iterate ad lib.



Overfitting

The complexity of the network is determined by the hidden layer – and overfitting is still a problem.

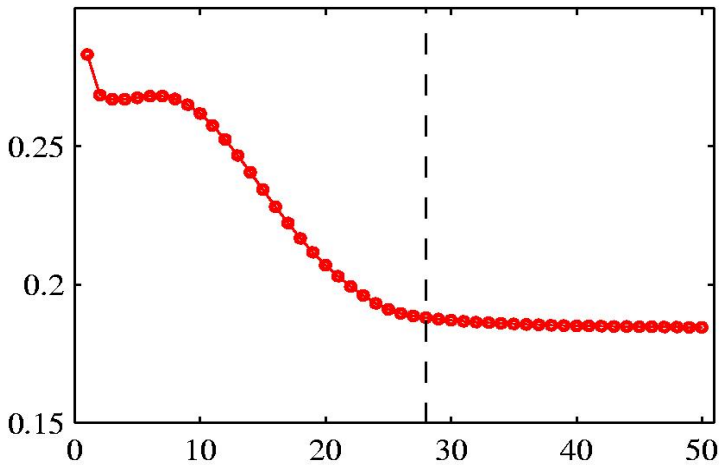
Use e.g. test datasets to alleviate this problem.



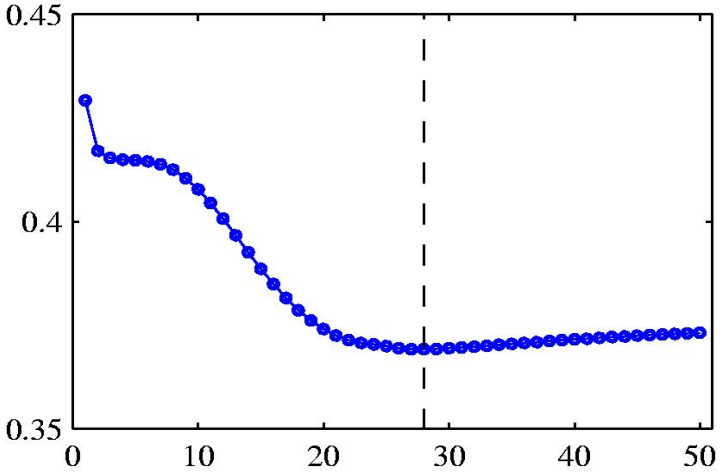
Early stopping

...or just stop the training when overfitting starts to occur.

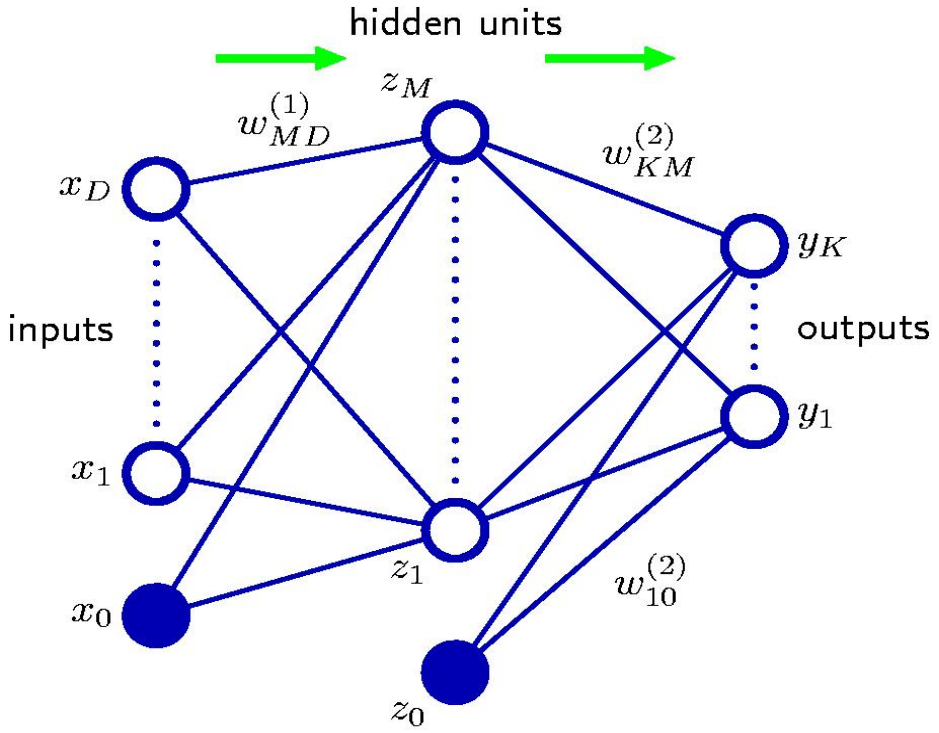
Training data



Validation data



Summary



- Neural network as general regression/classification models
- Feed-forward evaluation
- Maximum-likelihood framework
- Error back-propagation to obtain gradient