Neural Networks

CS 550: Machine Learning

Classifiers and Discriminant Functions

(revisited)

- A classifier is represented with a set of discriminant functions $g_j(x)$ for $j = 1, 2, \ldots C$
- A given instance x is then classified with the class C_j for which the discriminant function $g_j(x)$ is the maximum

1. **Likelihood-based approaches**
2. **Discriminant-based approaches**

Likelihood-Based Approaches

(revisited)

- They estimate class probabilities on the training samples and then use them to define the discriminant functions

$$
\begin{aligned}
g_j(x) &= \hat{P}(C_j \mid x) \\
&= \frac{\hat{P}(x \mid C_j) \cdot \hat{P}(C_j)}{\hat{P}(x)} \\
&\equiv \underbrace{\hat{P}(x \mid C_j) \cdot \hat{P}(C_j)}_{\text{For each class, estimate the likelihood and the prior from the training samples that belong to this class}}
\end{aligned}
$$

For each class, estimate the likelihood and the prior from the training samples that belong to this class

Discriminant-Based Approaches

(revisited)

- They learn discriminant functions directly on the training samples
- They make an assumption on the form of the discriminant functions and learn their parameters from the training samples without estimating the class probabilities
- **Linear discriminants assume that each discriminant function is a linear combination of the input features**

Linear Discriminants

- They define $g_j(x)$ as a linear combination of the input features

number of the input dimensions

$$g_j(x \mid W_j) = \sum_{i=1}^{d} W_{ij}\, x_i + W_{0j}$$

weight vector for the j-th class

let's define $x_0 = 1$

$$g_j(x \mid W_j) = \sum_{i=0}^{d} W_{ij}\, x_i$$

Learning involves learning the parameters (weights) W_j for each class C_j from the training samples

For that, we will define a criterion function and learn the weights that minimize/maximize this function

Linear Discriminants

- They yield hyperplane decision boundaries
- Consider the 2-class classification problem

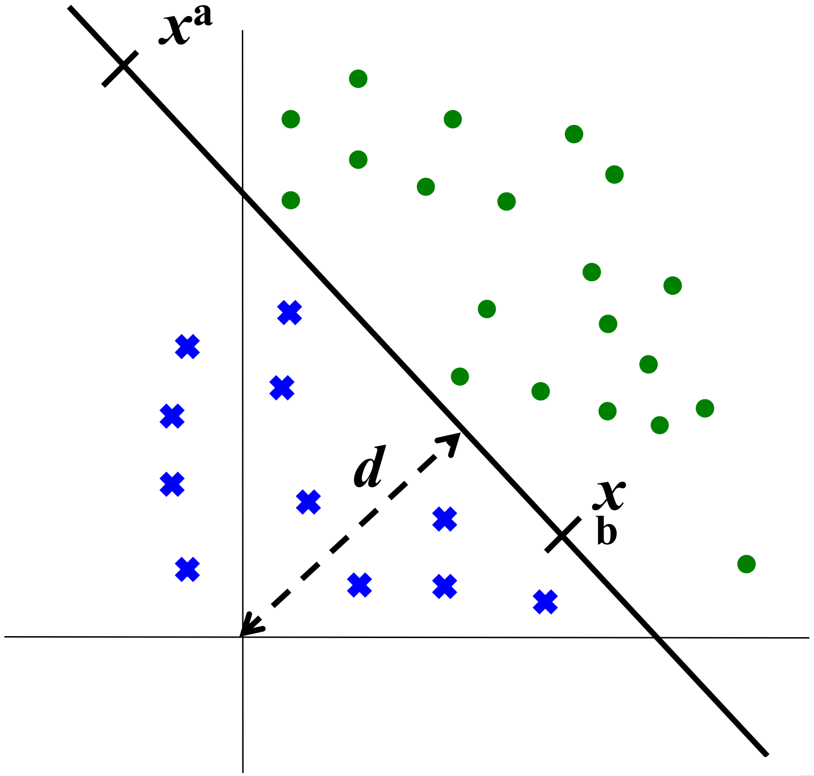
$$g_1(x) = \sum_{i=1}^{d} W_{i1}\, x_i \; + \; W_{01}$$

$$\left.\begin{aligned} g_1(x) &= W_1^{\mathsf{T}}\, x \; + \; W_{01} \\ g_2(x) &= W_2^{\mathsf{T}}\, x \; + \; W_{02} \end{aligned}\right\} \quad g(x) = g_1(x) - g_2(x) \qquad \text{choose} \quad \begin{cases} C_1 & \text{if } g(x) \geq 0 \\ C_2 & \text{otherwise} \end{cases}$$

$$g(x) = \left(W_1 \; - \; W_2\right)^{\mathsf{T}} x + \left(W_{01} \; - \; W_{02}\right)$$

$$g(x) = \underbrace{W^{\mathsf{T}} x \; + \; W_0}_{\text{This is another linear function}}$$

Linear Discriminants



$$g(x) = W^T x + W_0$$

Let's take two points on the decision plane

$$g(x^a) = g(x^b)$$

$$W^T x^a + W_0 = W^T x^b + W_0$$

$$\underbrace{W^T (x^a - x^b) = 0}$$

W determines the hyperplane's orientation
(W is normal to any vector on the hyperplane)

W_0 determines the hyperplane's location with respect to the origin

Linear Discriminants

We consider the multiclass classification as

1. one-against-one OR 2. one-against-all

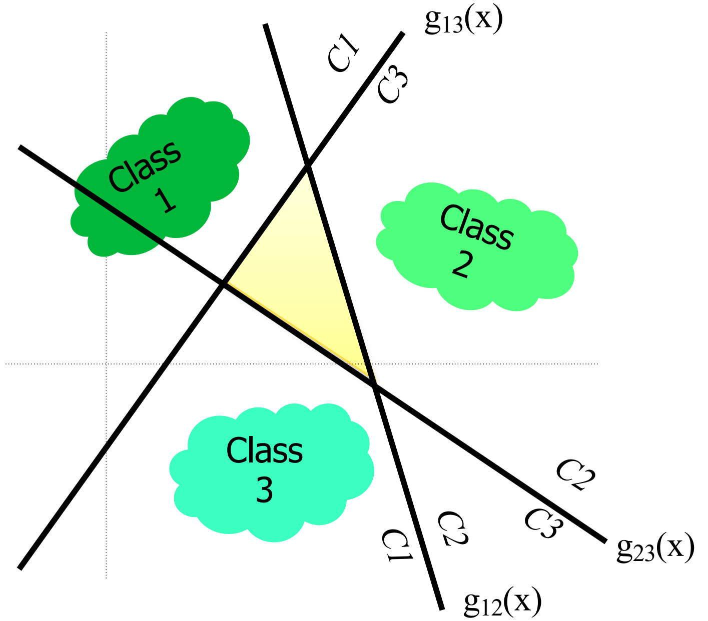
C (C – 1) / 2 discriminants

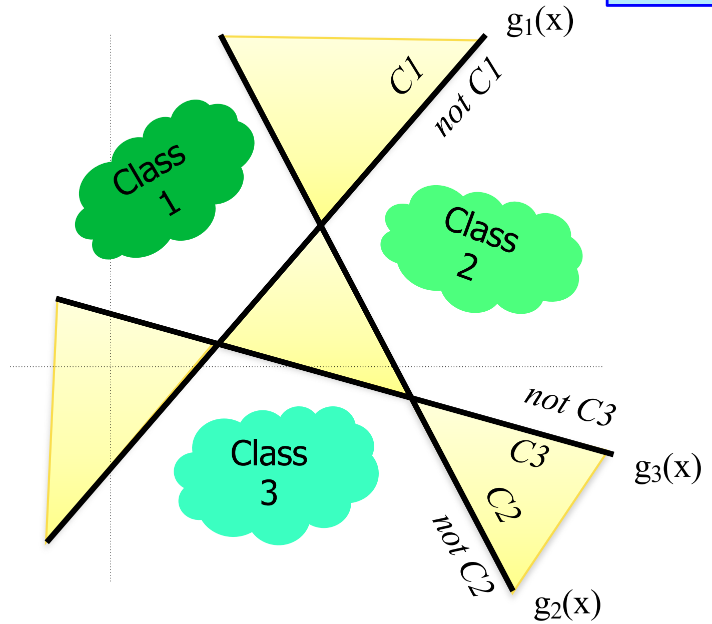
$$g_{kj}(x) = \begin{cases} \geq 0 & \text{if} \quad x \in C_k \\ < 0 & \text{if} \quad x \notin C_j \\ \text{don't care otherwise} \end{cases}$$

C discriminants

$$g_k(x) = \begin{cases} \geq 0 & \text{if} \quad x \in C_k \\ < 0 & \text{if} \quad x \notin C_k \end{cases}$$

To resolve ambiguities, we may assign x to the class for which the discriminant is highest (it is also possible to reject classification or combine the discriminants in a different way)

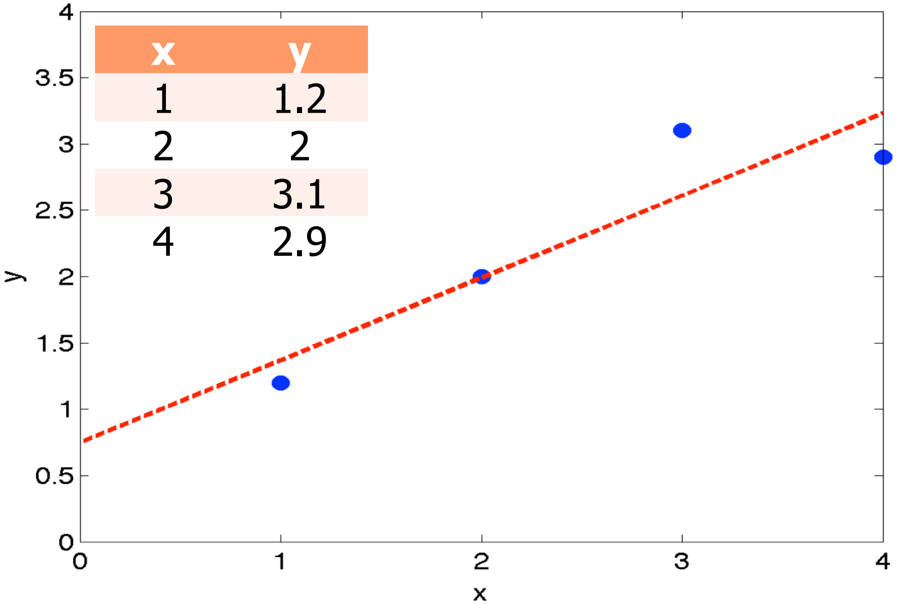




How to Learn

Although we will use linear discriminants for classification, let's first consider a linear regression problem

Construct a linear model on the following data points



construct a linear model

$$f(x) = W\,x + W_0$$

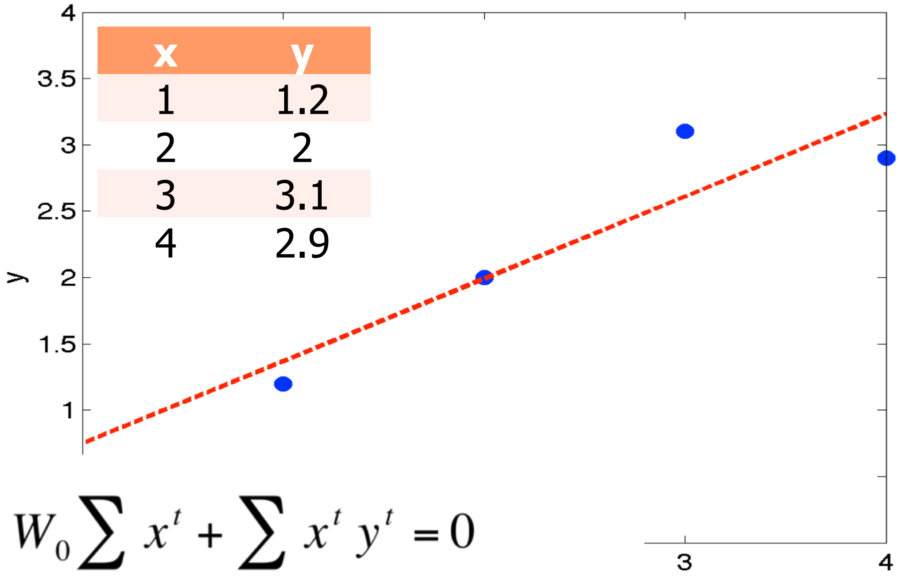
define a criterion function

$$\underbrace{loss(W\,,\,W_0)}_{\textbf{Sum of squared errors}} = \frac{1}{2}\sum_t \left(f(x^t) - y^t\right)^2$$

select W and W_0 that minimize this error on the training samples

$$\frac{\partial loss}{\partial W} = 0 \qquad \frac{\partial loss}{\partial W_0} = 0$$

Analytical Solution



x	y
1	1.2
2	2
3	3.1
4	2.9

$$loss(W, W_0) = \frac{1}{2} \sum_t \left(f(x^t) - y^t \right)^2$$

$$f(x) = W x + W_0$$

$$\frac{\partial loss}{\partial W} = \frac{1}{2}\, 2 \sum_t \left(W x^t + W_0 - y^t \right) x^t = W \sum_t x^{t^2} + W_0 \sum_t x^t + \sum_t x^t y^t = 0$$

$$\frac{\partial loss}{\partial W_0} = \frac{1}{2}\, 2 \sum_t \left(W x^t + W_0 - y^t \right) = W \sum_t x^t + W_0 N + \sum_t y^t = 0$$

<u>In our example</u>

$$\sum x^t = 10$$
$$\sum y^t = 9.2$$
$$\sum x^{t^2} = 30$$
$$\sum x^t y^t = 26.1$$
$$N = 4$$

$$\left.\begin{array}{l} 30\ W + 10\ W_0 - 26.1 = 0 \\ 10\ W + 4\ W_0 - 9.2 = 0 \end{array}\right\}$$

2 unknowns
2 equations $\Rightarrow$ $W = 0.62$, $W_0 = 0.75$

$$f(x) = 0.62\ x\ +\ 0.75$$

In many cases, there is no analytical solution
(If the linear system has a singular matrix,
no solution or multiple solutions exist)
➡ **ITERATIVE OPTIMIZATION METHODS**

Gradient Descent Algorithm

- One commonly used iterative optimization method
- Goal is to find the parameters that minimize the loss
  - Starting with random parameters, it iteratively updates them in the direction of the steepest descent (in the opposite direction of the gradient) until the gradient is zero (or small enough)

start with random weights W_i

do

$$\Delta W_i = -\eta \frac{\partial loss_{ALL}}{\partial W_i} \quad \text{for all } i$$

$$W_i = W_i + \Delta W_i \quad \text{for all } i$$

until convergence

It finds the nearest minimum, which could be local

It does not guarantee to find the global minimum

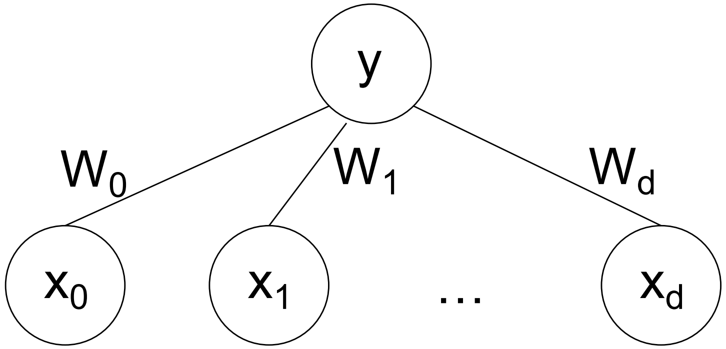
η is the learning rate, which determines how much to move in the direction of the steepest descent

→ *if it is too small, convergence is slow*

→ *if it is too large, we may overshoot the minimum (divergence might occur)*

Regression

Let's derive the update rules for regression



$$loss_{ALL}(W) = \sum_t loss^t$$

$$loss = \frac{1}{2}\left(f(x) - y\right)^2$$

$$f(x) = net$$

$$net = \sum_i x_i W_i$$

$$\Delta W_i = -\eta \frac{\partial loss_{ALL}(W)}{\partial W_i}$$

$$\frac{\partial loss}{\partial W_i} = \frac{\partial loss}{\partial net} \frac{\partial net}{\partial W_i}$$

$$\frac{\partial loss}{\partial W_i} = \delta x_i$$

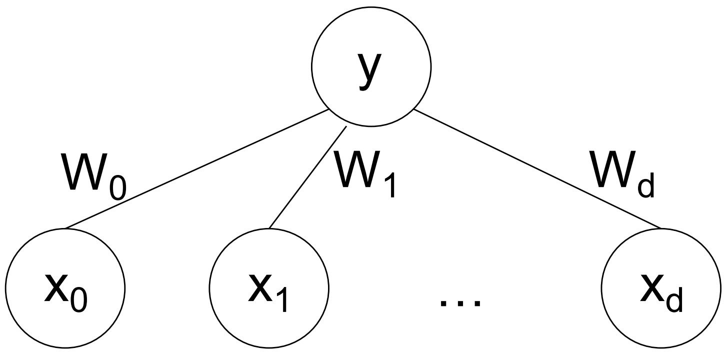
$$\delta = \left(net - y\right)$$

$$\Delta W_i = -\eta \sum_t \delta x_i$$

$$\boxed{\Delta W_i = \eta \sum_t \left(y^t - net^t\right) x^t{}_i}$$

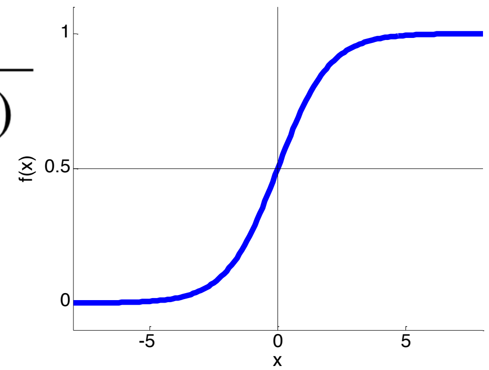
Classification (Logistic Regression)

Let's derive the update rules for 2-class classification



Logarithmic sigmoid function

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



$$y = 1 \quad \text{if } x \in C_1$$

$$y = 0 \quad \text{if } x \in C_2$$

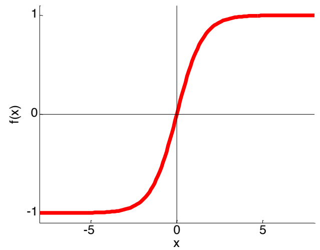
$$\boxed{f(x) = \sigma(net)}$$

$$net = \sum_i x_i W_i$$

$$\sigma'(net) = \sigma(net)\,\big(1 - \sigma(net)\big)$$

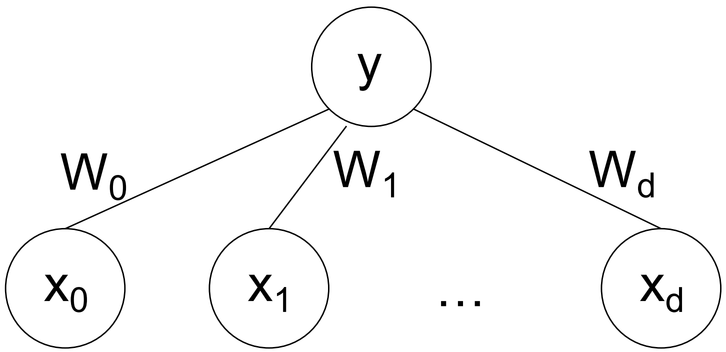
Hyperbolic tangent sigmoid function

$$f(x) = a \tanh(b\,x) = a \left[\frac{\exp(b\,x) - \exp(-b\,x)}{\exp(b\,x) + \exp(-b\,x)} \right]$$



Classification (Logistic Regression)

Let's derive the update rules for 2-class classification



$$\boxed{f(x) = \sigma(net)}$$

$$net = \sum_i x_i W_i$$

$$\frac{\partial loss}{\partial W_i} = \frac{\partial loss}{\partial net} \frac{\partial net}{\partial W_i}$$

$$\frac{\partial loss}{\partial W_i} = \delta x_i$$

$$loss_{ALL}(W) = \sum_t loss^t$$

$$\underbrace{loss = -y \log f(x) \; -}_{\text{Cross entropy}} \; (1-y) \log (1 - f(x))$$

$$\underbrace{loss = \frac{1}{2} \left(f(x) - y\right)^2}_{\text{Squared error}}$$

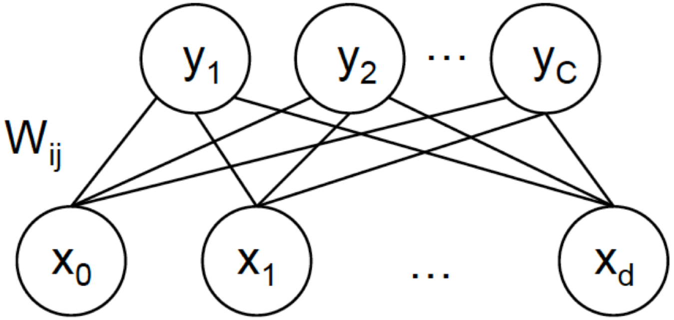
When squared error is used

$$\boxed{\delta = \left(\sigma(net) - y\right) \sigma'(net)}$$

$$\boxed{\Delta W_i = \eta \sum_t \left(y^t - \sigma(net^t)\right) \sigma(net^t) \left(1 - \sigma(net^t)\right) x^t{}_i}$$

Classification

Let's derive the update rules for multiclass classification



Define output as a C-dimensional vector

$$y_j = 1 \quad \text{if } x \in C_j$$
$$y_j = 0 \quad \text{if } x \notin C_j$$

$$f_j(x) = softmax(net_j)$$

$$net_j = \sum_i x_i W_{ij}$$

$$softmax(net_j) = \frac{\exp(net_j)}{\sum_m \exp(net_m)}$$

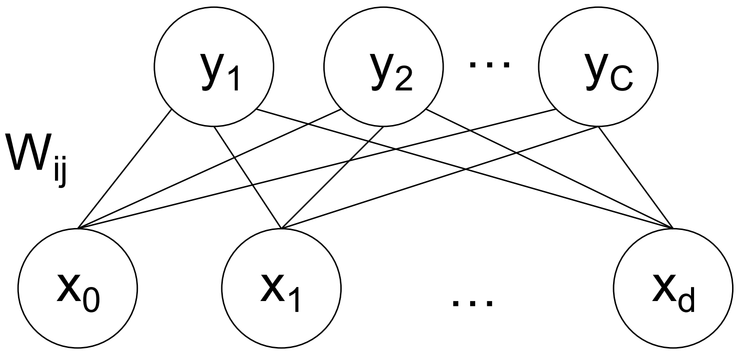
$$\frac{\partial\, softmax(net_k)}{\partial\, net_j} = softmax(net_j)\left(\delta_{jk} - softmax(net_k)\right)$$

$$\delta_{jk} = \underbrace{\begin{cases} 1 & \text{if } j = k \\ 0 & \text{if } j \neq k \end{cases}}_{\text{Kronecker delta}}$$

Kronecker delta

Classification

Let's derive the update rules for multiclass classification



$$f_j(x) = softmax(net_j)$$

$$net_j = \sum_i x_i W_{ij}$$

$$\frac{\partial loss}{\partial W_{ij}} = \frac{\partial loss}{\partial net_j} \frac{\partial net_j}{\partial W_{ij}}$$

$$\frac{\partial loss}{\partial W_{ij}} = \delta_j x_i$$

$$loss_{ALL}(W) = \sum_t loss^t$$

$$\underbrace{loss = -\sum_k y_k \log f_k(x)}_{\text{Cross entropy}}$$

$$\underbrace{loss = \frac{1}{2} \sum_k \left(f_k(x) - y_k\right)^2}_{\text{Squared error}}$$

When squared error is used

$$\boxed{\delta_j = \sum_k \left(softmax(net_k) - y_k\right) \frac{\partial\, softmax(net_k)}{\partial\, net_j}}$$

$$\boxed{\Delta W_{ij} = \eta \sum_t \sum_k \left(y_k^t - s(net_k^t)\right) s(net_j^t)\, (\delta_{jk} - s(net_k^t))\, x^t{}_i}$$

softmax(net$^t{}_k$)

Batch learning algorithm

start with random weights W_{ij}

do

EPOCH

compute $f_j(x^t) = softmax\left(\sum_i x^t{}_i\, W_{ij}\right)$ for all t and j

compute $\Delta W_{ij} = -\eta \sum_t \delta_j\, x_i^t$ for all i and j

update $W_{ij} = W_{ij} + \Delta W_{ij}$ for all i and j

until convergence

Stochastic learning algorithm

start with random weights W_{ij}

do

EPOCH

for all (x^t, y^t) in random order

compute $f_j(x^t) = softmax\left(\sum_i x^t{}_i\, W_{ij}\right)$ for all j

compute $\Delta W_{ij} = -\eta\, \delta_j\, x_i^t$ for all i and j

update $W_{ij} = W_{ij} + \Delta W_{ij}$ for all i and j

until convergence

Mini-batch stochastic learning algorithm is a good tradeoff

Batch or Stochastic Learning?

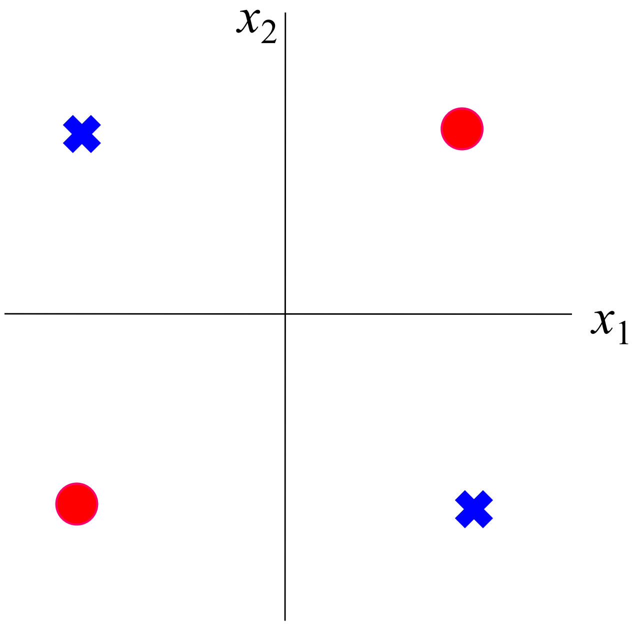
- Batch learning allows using some second-order techniques that
  - Cannot be easily incorporated into stochastic learning
- Stochastic training is preferred for most applications
  - Especially when datasets are highly redundant
  - It is typically faster than batch training
- Mini-batch stochastic learning is a good tradeoff

Adding Nonlinearity

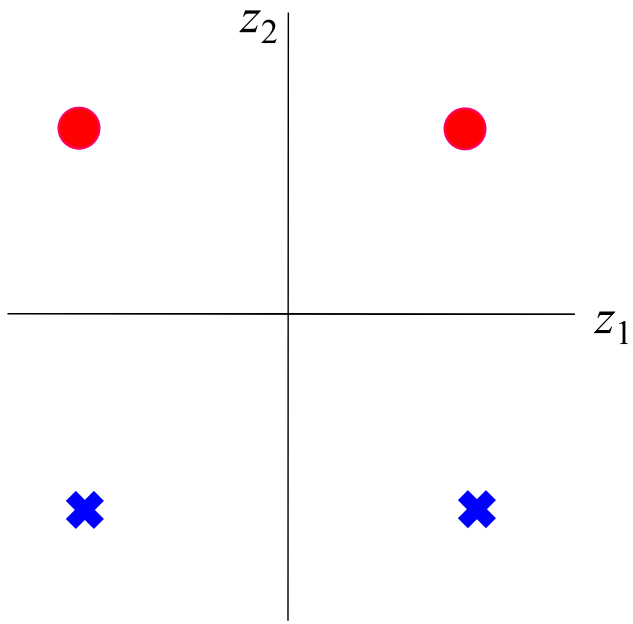
- Linear discriminants yield hyperplane decision boundaries

- If they are not sufficient to construct a “good” model
  1. We may transform the space into a new one using nonlinear mappings and construct linear discriminants on the transformed space → **SVMs**
  2. We may learn the nonlinearity at the same time as the linear discriminants → **ANNs**

XOR Problem



$$z_1 = x_1$$
$$z_2 = x_1 \, x_2$$

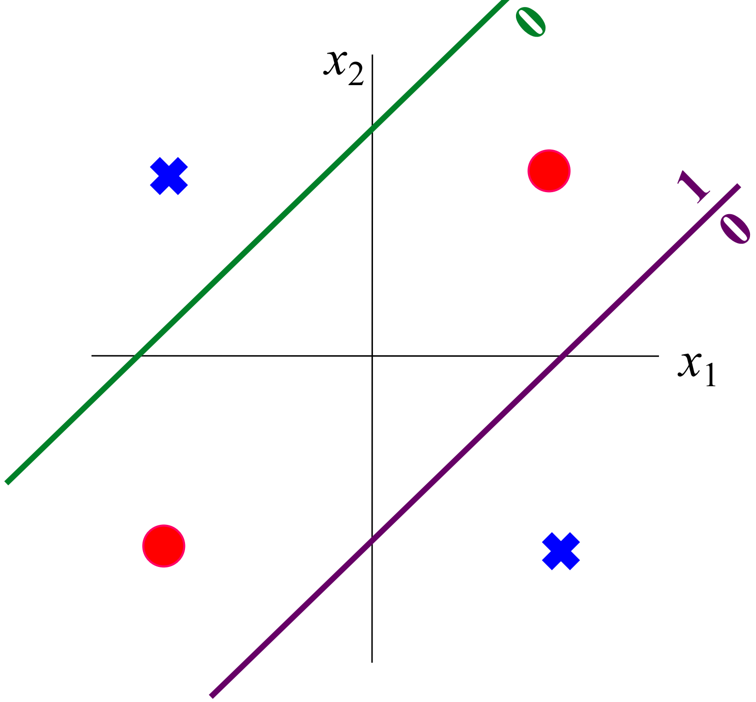


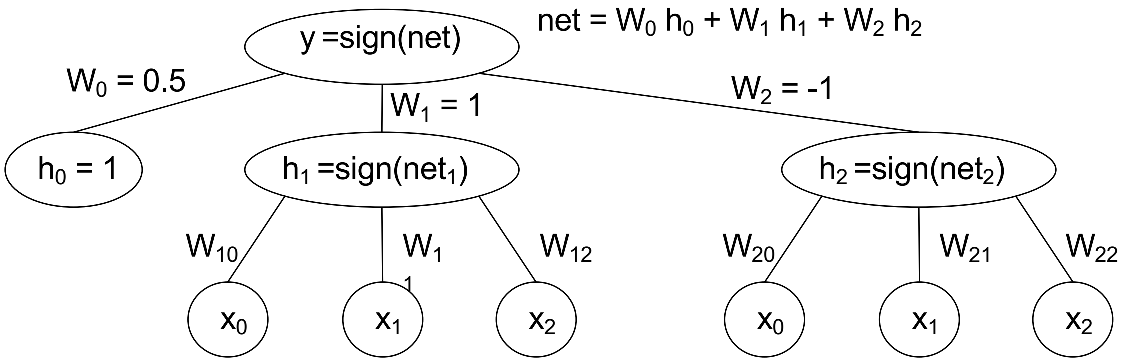
Support vector machines use the idea of nonlinear mapping to find a linearly separable space

XOR Problem

$$net_1 = W_{12}\ x_2 + W_{11}\ x_1 + W_{10}$$
$$h_1 = sign(\,net_1\,)$$

$$net_2 = W_{22}\ x_2 + W_{21}\ x_1 + W_{20}$$
$$h_2 = sign(\,net_2\,)$$

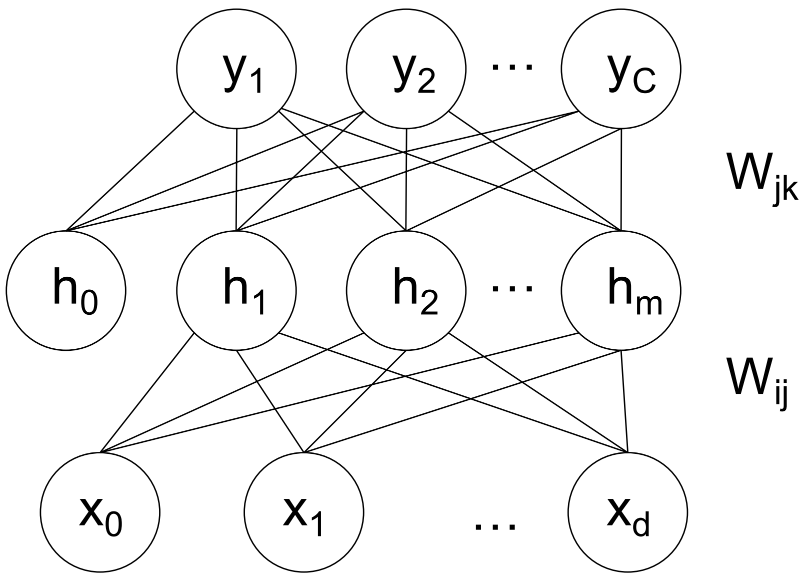




Neural networks learn the nonlinearity at the same time as the linear discriminants
(learn all the weights at the same time)

Multilayer Perceptrons

- Also contain hidden layers in addition to input and output layers



Hidden units h_j's can be viewed as new "features" obtained by combining x_i's

A deeper architecture with nonlinear activations is more expressive than a shallow one

<u>In this network</u>

1. Each hidden unit computes its net activation $$net_j = \sum_i x_i W_{ij}$$
2. Each hidden unit emits an output that is a nonlinear function of its activation $$h_j = \sigma(net_j)$$
3. Each output unit computes its net activation $$net_k = \sum_j h_j W_{jk}$$
4. Each output units emits an output $$y_k = g(net_k)$$

How to Learn?

- In linear discriminants, we select the weights to minimize a loss function defined on the difference between the actual and computed output values
- In multilayer structures, we can also select the hidden-to-output-layer weights to minimize a loss function defined on the actual and computed output values
- However, we cannot select the input-to-hidden-layer weights in a similar way since we do not know the actual values of the hidden units
- Thus, to learn the input-to-hidden-layer weights, we propagate the loss function (defined on the output values) from the output layer to the corresponding hidden layer
  → **BACKPROPAGATION ALGORITHM**

Backpropagation Algorithm

Let's derive the update rules for multiclass classification

$$net_j = \sum_i x_i W_{ij}$$

$$h_j = \sigma(net_j)$$

$$net_k = \sum_j h_j W_{jk}$$

$$f_k(x) = softmax(net_k)$$

$$loss_{ALL}(W) = \sum_t loss^t$$

$$\underbrace{loss = \frac{1}{2} \sum_k \left(f_k(x) - y_k\right)^2}_{\text{Squared error}}$$

$$\Delta W_{jk} = -\eta \frac{\partial loss_{ALL}(W)}{\partial W_{jk}}$$

Hidden-to-output-layer weights

$$\frac{\partial loss}{\partial W_{jk}} = \frac{\partial loss}{\partial net_k} \frac{\partial net_k}{\partial W_{jk}}$$

$$\frac{\partial loss}{\partial W_{jk}} = \delta_k h_j$$

$$\delta_k = \sum_m \left(softmax(net_m) - y_m\right) \frac{\partial\, softmax(net_m)}{\partial\, net_k}$$

Backpropagation Algorithm

Let’s derive the update rules for multiclass classification

$$net_j = \sum_i x_i W_{ij}$$

$$h_j = \sigma(net_j)$$

$$net_k = \sum_j h_j W_{jk}$$

$$f_k(x) = softmax(net_k)$$

$$loss_{ALL}(W) = \sum_t loss^t$$

$$\underbrace{loss = \frac{1}{2} \sum_k \left(f_k(x) - y_k\right)^2}_{\text{Squared error}}$$

$$\Delta W_{ij} = -\eta \frac{\partial loss_{ALL}(W)}{\partial W_{ij}}$$

$$\frac{\partial loss}{\partial W_{ij}} = \frac{\partial loss}{\partial net_j} \frac{\partial net_j}{\partial W_{ij}}$$

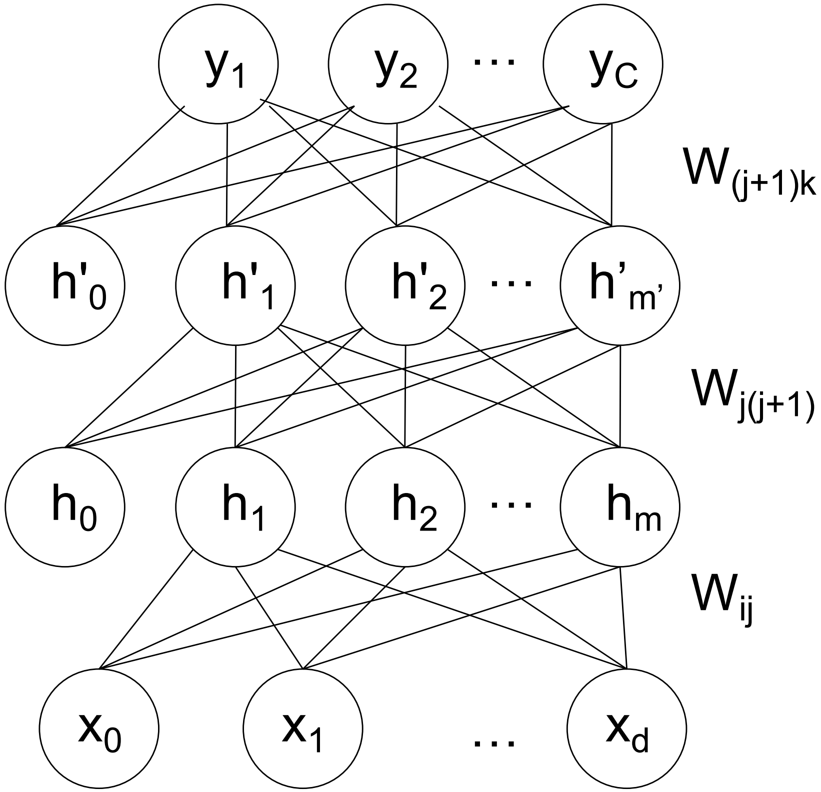
$$\frac{\partial loss}{\partial W_{ij}} = \delta_j x_i$$

$$\delta_j = \sum_k \frac{\partial loss}{\partial net_k} \frac{\partial net_k}{\partial net_j} = \left[\sum_k \delta_k W_{jk}\right] \sigma'(net_j)$$

Input-to-hidden layer weights

Exercise: Derive the update rules for regression

More Hidden Layers



Input-to-first-hidden layer weights

$$\frac{\partial loss}{\partial W_{ij}} = \frac{\partial loss}{\partial net_j} \frac{\partial net_j}{\partial W_{ij}}$$

$$\frac{\partial loss}{\partial W_{ij}} = \delta_j x_i$$

$$\delta_j = \sum_{(j+1)} \frac{\partial loss}{\partial net_{(j+1)}} \frac{\partial net_{(j+1)}}{\partial net_j}$$

$$\delta_j = \left[\sum_{(j+1)} \delta_{(j+1)} W_{j(j+1)} \right] \sigma'(net_j)$$

δ_j may vanish after repeated multiplication
This makes deep architectures hard to train (when the initial values of the weights are not "good" enough)

Activation Function

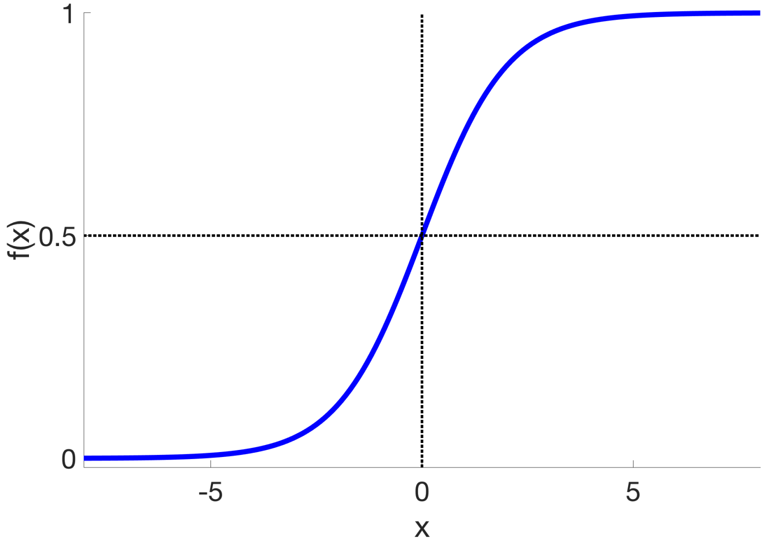
- The activation function $\sigma(.)$ must be nonlinear and $\sigma(.)$ and $\sigma'(.)$ must be defined throughout the range of their argument (for backpropagation)

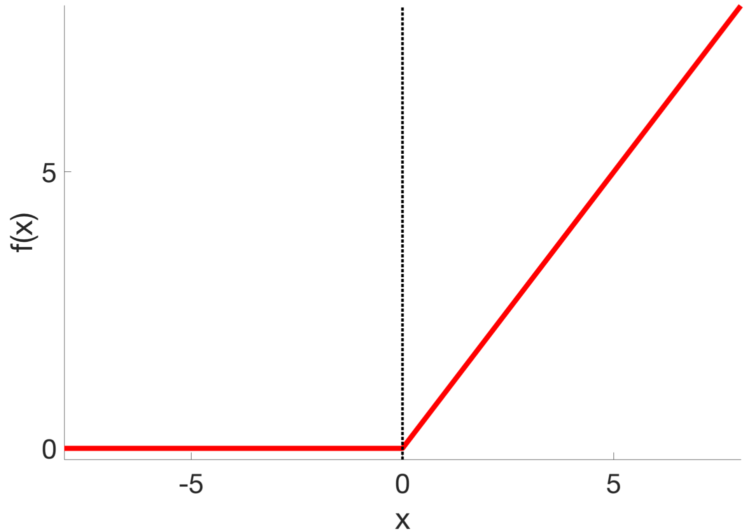
  - Logarithmic sigmoid function

$$f(x) = \frac{1}{1+\exp(-x)}$$

  - Rectified linear unit (ReLU)

$$f(x) = \max(0\ , x)$$





Target Output Values

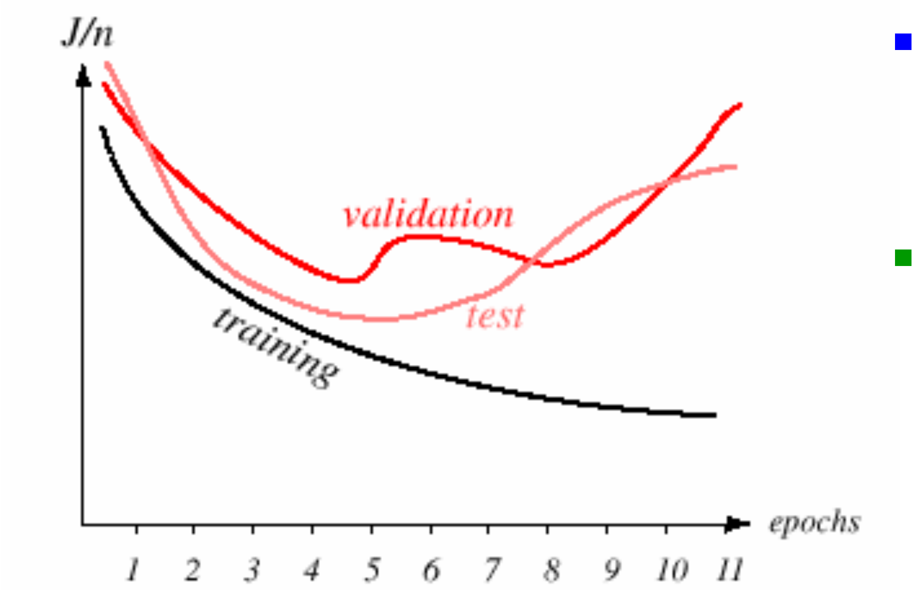
- In classification, a target value can be represented by
  - 0/1 target values
    - Outputs represent posterior probabilities
    - Using the softmax function, the maximum output is transformed towards 1.0 and all others reduced to 0.0
  - -1/+1 target values
    - Outputs do not represent posterior properties
- A proper activation function should be used in the output layer
  - Depending on the selected target value representation

When to Stop

- When the change in the loss function is smaller than some preset value θ

$$\left\|\nabla loss_{ALL}(W)\right\| \leq \theta$$

- When a minimum is reached on the validation set



- Training error ultimately reaches an asymptotic value
- The error on an independent test set is virtually always higher
  - Although it usually decreases, it can also increase or oscillate

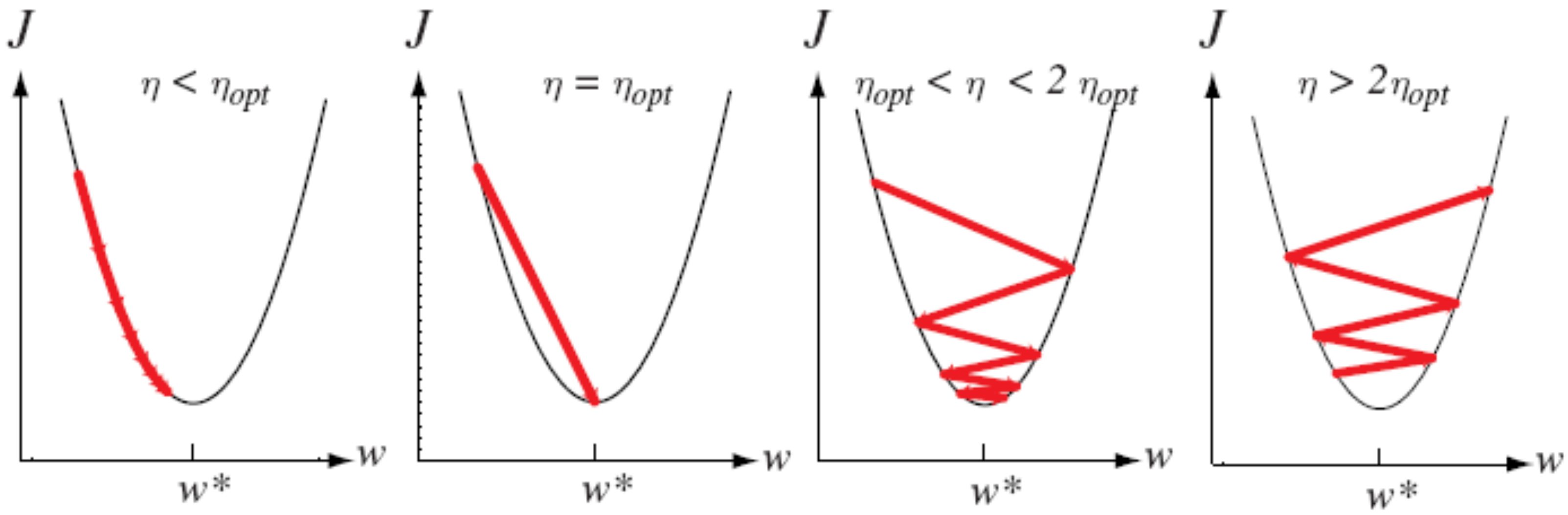
Initializing Weights

- We cannot initialize the weights to zero
- We want to have uniform learning
  - All weights reach their final equilibrium at about the same time
  - For that, with standardized data, we choose weights randomly from a uniform distribution $-\omega < w < \omega$
  - When the sigmoid function is used (for calculating an output and/or for defining hidden units in MLPs)
    - If ω is chosen too small, the net will be too small
    - If ω is chosen too large, sigmoid may saturate even before learning begins
    - Set ω such that sigmoid is in its linear range

Learning Rate

- In principle, if the learning rate is small enough, it ensures the convergence
  - Its value determines only the speed
  - Not the final weight values
- In practice, the learning rate can indeed affect the quality of the final network
  - Since networks are not fully trained most of the time

Learning Rate



- The optimal learning rate leads to the local minimum in one step
- The optimal rate is found as $$\boxed{\eta_{opt} = \left(\frac{\partial^2 loss_{ALL}(W)}{\partial W^2} \right)^{-1}}$$
- The system converges for $\boxed{\eta < \eta_{opt}}$ and $\boxed{\eta_{opt} < \eta < 2\,\eta_{opt}}$
  - But training is needlessly slow
- It is found that the system diverge if $\boxed{\eta > 2\,\eta_{opt}}$

Learning Rate

- Thus, in order to have rapid and uniform learning
  - For each weight, calculate $\partial^2 loss_{ALL}(W)/\partial W^2$ and set the optimal learning rate separately (not so much practical)

- For typical networks that use sigmoid functions
  - η=0.1 is a good choice to start with
    - It should be lowered if the loss function diverges
    - It should be raised if learning seems unduly slow

- During training, it is also possible to change the learning rate η as a function of time (epoch number)

Regularization

- Adding regularization term to the loss function reduces sensitivity to training samples and decreases the risk of overfitting

$$loss_{ALL}(W) = \frac{1}{T}\sum_{t} loss^{t} + \|W\|$$

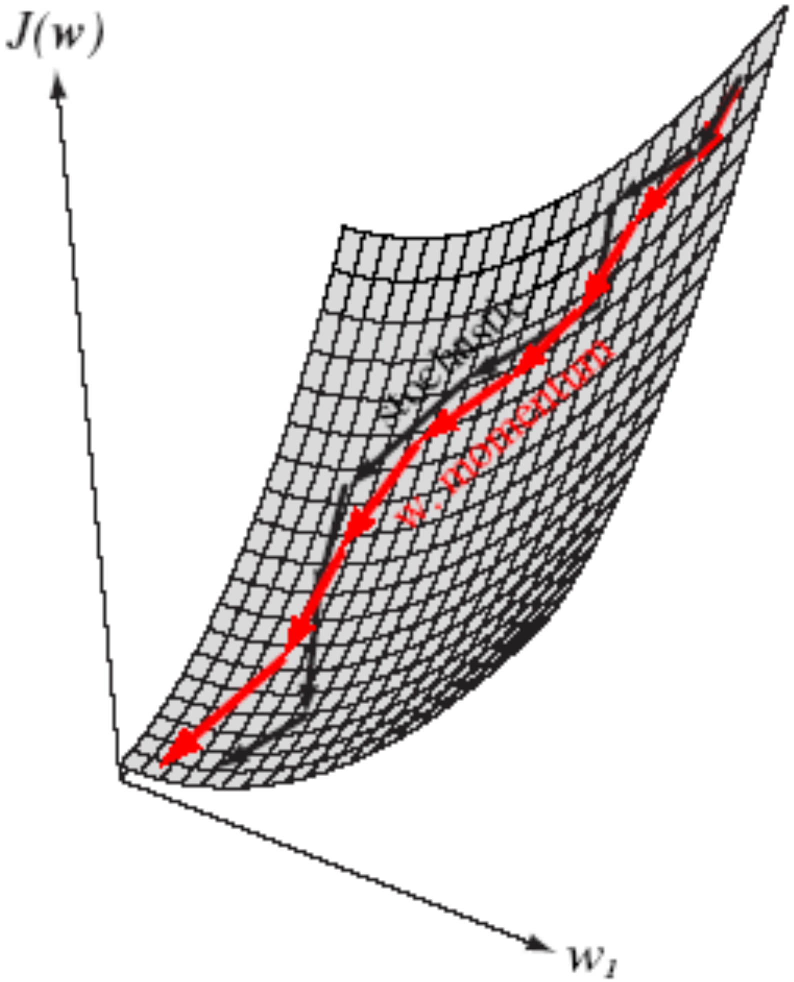
$$loss_{ALL}(W) = \underbrace{\frac{1}{2T}\sum_{t}\sum_{k}\left(f_k(x^t) - y_k^t\right)^2}_{\text{Mean squared error}} + \underbrace{\frac{\lambda}{2T}\|W\|_2^2}_{\text{L2-regularization term}} \qquad \text{where } \|W\|_2^2 = \sum_{i} W_i^2$$

Regularization

Dropout regularization

- During training, in each iteration, randomly drop out units (also their incoming and outgoing connections) with probability p to sample a "thinned" network and train it
- Training can be seen as training a collection of different thinned networks with extensive weight sharing
- Training typically takes longer
- In testing, consider the entire network where the weights are scaled down by multiplying them a factor of 1 – p

Momentum



- Error surfaces often have plateaus
  - Regions in which the derivative is very small
  - Such plateaus may arise especially when there are too many weights such that the loss function only weakly depends on any of them

- Momentum allows to learn more quickly when there are such plateaus

Momentum

- For stochastic learning algorithm, we include some fraction of the previous weight updates into the learning rule

$$w^{(t+1)} = w^{(t)} + (1-\alpha)\,\Delta w^{(t)} + \alpha\,\Delta w^{(t-1)}$$

  - Parameter α should be nonnegative and less than 1
  - If $\alpha = 0$, it is the same as the standard gradient descent
  - If $\alpha = 1$, the weight vector moves with constant velocity
  - Values typically used are $\alpha \cong 0.9$

- The use of momentum increases stability
- It may speed up the learning process

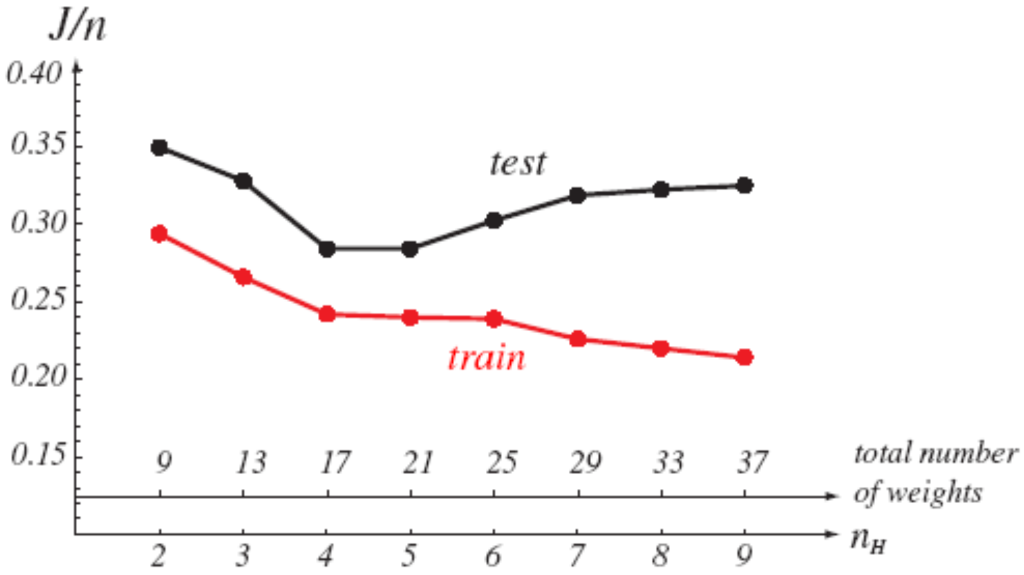
Network Topology

The number of hidden units

- It controls the expressive power of the network
  - Thus, the complexity of the decision boundary

- There is no foolproof method to set the number of hidden units before training
  - If samples are well-separated
    - few hidden units are enough
  - If samples have complicated densities
    - more hidden units may be necessary

Network Topology



- If too much hidden units,
  - The network is tuned to the particular dataset (overfitting)
    - Training error can become small, but test error is unacceptably high
- If too few hidden units,
  - The network does not have enough free parameters to fit the training data well
    - Training and test errors are high