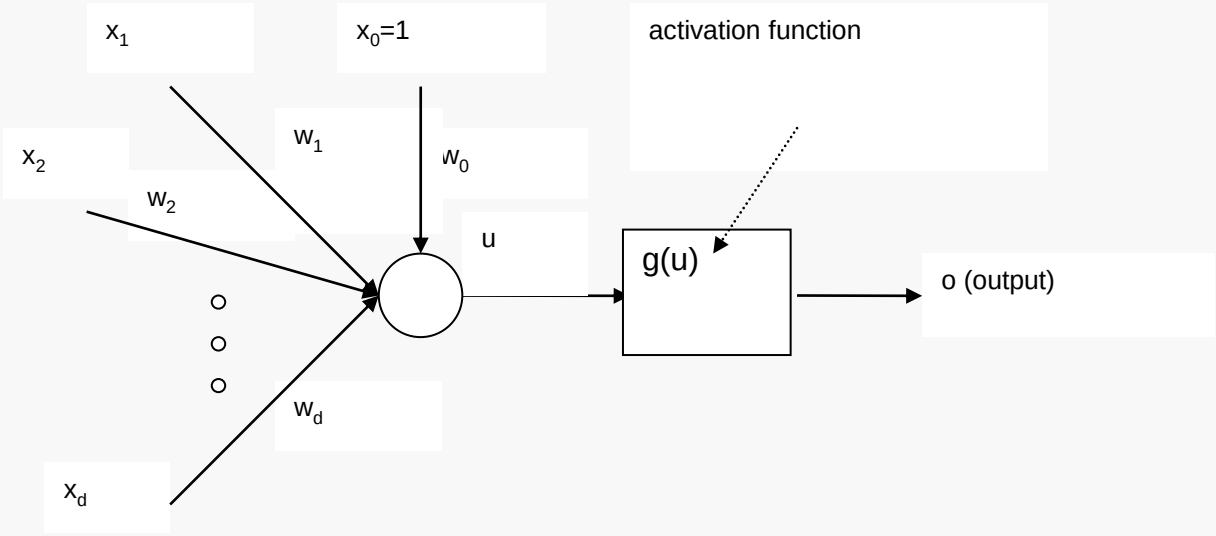
Neural Networks

Βιολογικά Νευρωνικά Δίκτυα

Artificial Neuron (generalized linear model)



- d inputs,
- input signal x_i (i=1,…,d)
- connection weights w_i, (i=1,…,d)
- bias (w_0) → weight of a connection whose input is 1

Artificial Neuron

- Two stage computation:
  - Compute activation (total input):

$$u(x)=\sum_{i=1}^{d} w_i x_i + w_0$$

  - Compute output o(x) using **activation function**

$$o(x)=g(u)$$

  - **inner–product neuron**

$$u(x)=w^T x + w_0$$

Artificial Neuron

- Alternative formulation:
  - Weight vector: $w=(w_1, w_2, \ldots, w_d)^T$
  - Extended weight vector:
    - $w_e=(w_0, w_1, w_2, \ldots, w_d)^T$
  - Extended input vector:
    - $x_e=(1, x_1, x_2, \ldots, x_d)^T$
- $u(x)=w_e^T x_e$ <=====> $u(x)=w^T x + w_0$

Activation functions

- **Step (or threshold) function):**
  - if $x<0$ then $g(x)=a$ and if $x>0$ then $g(x)=b$. Usually (a=0, b=1) or (a=-1, b=1).
  - Discontinuous for x=0.
  - **Derivative equal to zero -> difficulty in training**

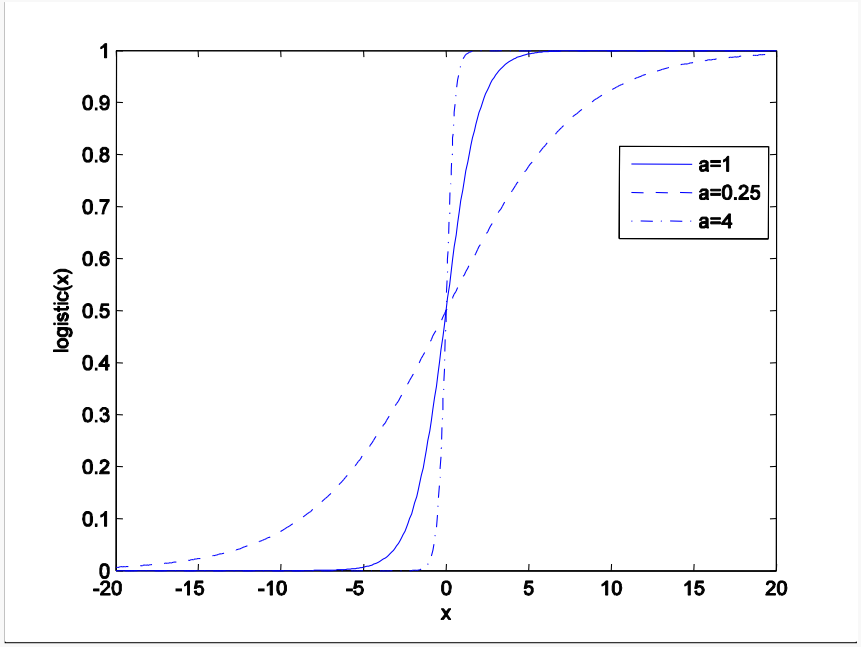
Activation functions

- **Sigmoid functions**
  1) **Logistic:**

     σ(x)=1/(1+exp(-ax))

     (a: slope, usually a=1)

     gives values in (0,1)



σ’(x)=σ(x)(1-σ(x))

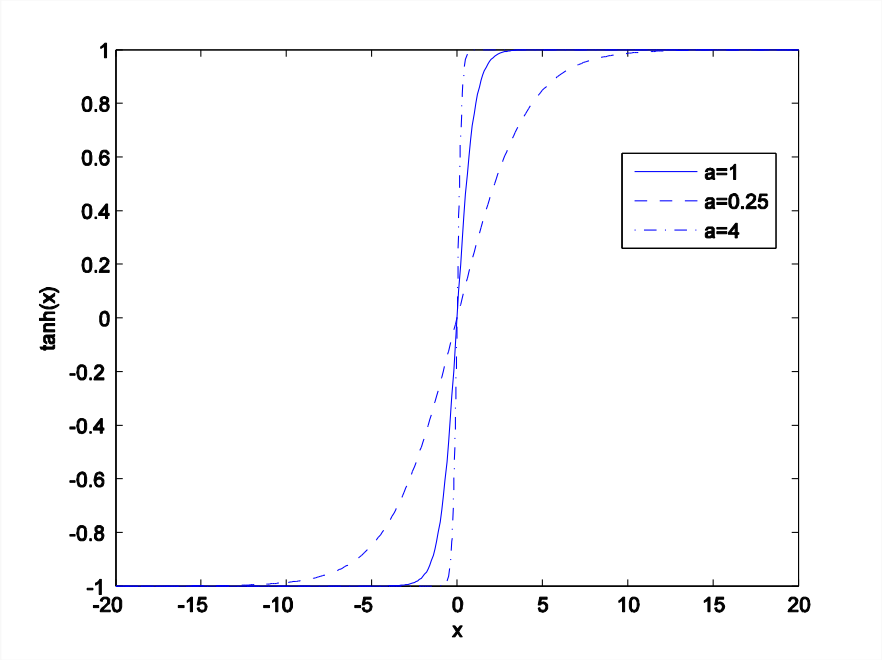
σ’’(x)=σ(x)(1-σ(x))(1-2σ(x))

Συναρτήσεις ενεργοποίησης

2) **Υπερβολική εφαπτομένη:**

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

(a: κλίση, συνήθως a=1)
δίνει τιμές στο (-1,1)
tanh'(x)=1-tanh²(x)

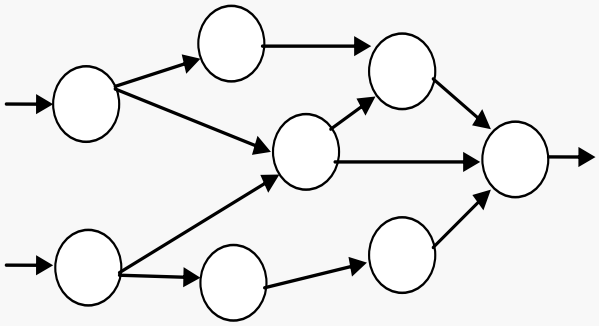


- **Γραμμική συνάρτηση**
  g(x)=x, g'(x)=1
  - μόνο στην έξοδο του δικτύου

- **Συνάρτηση RelU**
  g(x)=max(0,x)

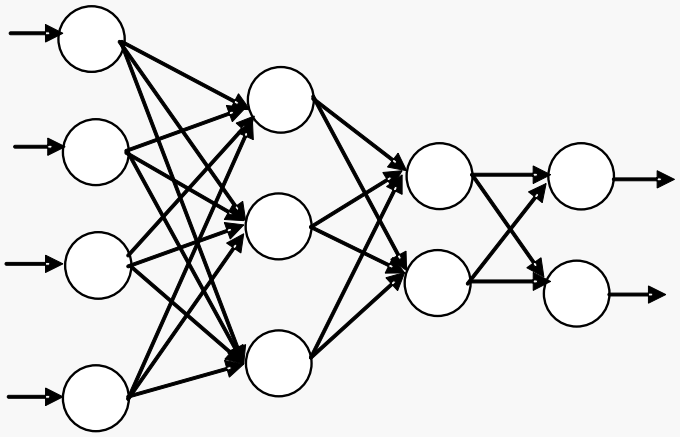
Feedfoward Neural Networks

- No feedback connections
- Computations flow from input to output.
- Implement static mapping from R^d to R^p (d inputs, p outputs).



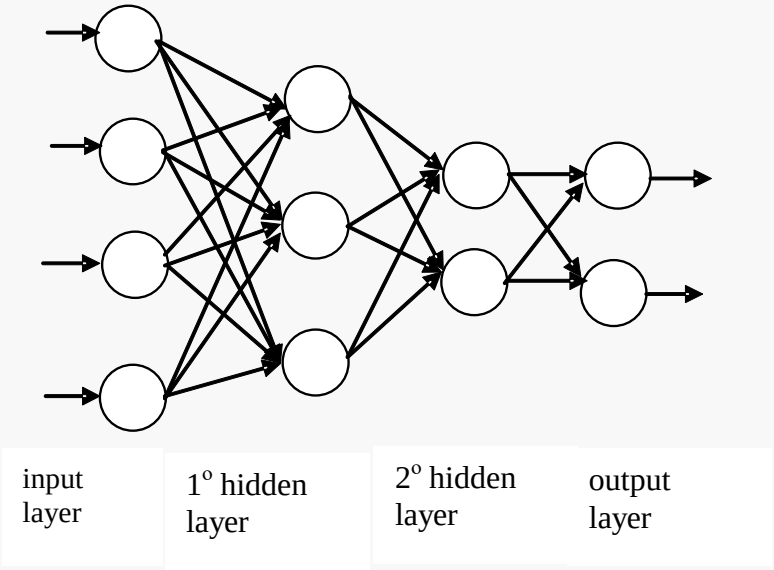
Layered Feedforward Neural Networks

- Neuron are organized in **layers:** no connections between neurons in the same layer.
- **Full connectivity between neurons in consecutive layers.**
- **At least on hidden layer: Neurons in hidden layers should have nonlinear activation (usually sigmoid).**
- **Generalized linear models with parametric basis functions $\phi(x;w)$ are special case with 1 hidden layer**



MultiLayer Perceptron (MLP)

- Layered feedforward neural network.
- Inner-product units
- Non-linear activation (e.g. logistic sigmoid) in hidden units



MultiLayer Perceptron (MLP)

- Notation i^{ℓ} : neuron i in layer ℓ
- $\mathrm{u}_{\mathrm{i}}^{(\ell)}$: total input
- $y_{\mathrm{i}}^{(\ell)}$: output
- $\delta_{\mathrm{i}}^{(\ell)}$: error
- $w_{\mathrm{i}0}^{(\ell)}$: bias (or $b_{\mathrm{i}}^{(\ell)}$)
- g_{ℓ} : activation function in layer ℓ
- d_{ℓ} : number of neurons in layer ℓ
- $w_{\mathrm{ij}}^{(\ell)}$: weight of connection from neuron $j^{\ell-1}$ to neuron i^{ℓ}

MultiLayer Perceptron(MLP)

- Consider MLP with **d** inputs, **p** outpus and **H** hidden layers. Input layer is zero, output layer is H+1. ($d_0 = d$, $d_{H+1} = p$)

- **Forward pass** (given input vector compute output vector):
- Input layer: $y_i^{(0)} = x_i$, $y_0^{(0)} = x_0 = 1$
- For h=1,…,H+1

$$u_i^{(h)} = \sum_{j=0}^{d_{h-1}} w_{ij}^{(h)} y_j^{(h-1)} \quad <==> \quad u_i^{(h)} = \sum_{j=1}^{d_{h-1}} w_{ij}^{(h)} y_j^{(h-1)} + w_{i0}^{(h)}, \quad i=1,\ldots d_h$$

$$y_i^{(h)} = g_h(u_i^{(h)}) \quad i=1,\ldots,d_h, \quad y_0^{(h)} = 1$$

- Output vector:

$$o_i = y_i^{(H+1)} \qquad i=1,\ldots,p$$

MLP Computational Capabilities

- MLP implements a mapping from input space to output space.
- The desired mapping is determined by training examples.
- **Universal approximation property**: **an MLP with at least one hidden layer with non-linear hidden units is able to implement any mapping** with arbitrary accuracy if the number of hidden units becomes arbitrarily large.
- Not useful in practice.
- Determining the number of hidden units is a **model selection** problem.

MLP computational capabilities

- The use of non-linear hidden units provides increased computational capabilities.
- MLP can solve non-linear separable problems.
- Decision regions are defined as intersections among hyperplanes.
- Usually 1 or 2 hidden layers are used. Recently there is increased interest in **deep neural networks** (more than two hidden layers).

MLP training

- Data set $\mathbf{D=\{(x^n,t^n)\}}$, n=1,…,N.
- $x^n=(x_{n1,\ldots,}x_{nd})^T$, $t^n=(t_{n1,\ldots,}t_{np})^T$ (regression problem).
- MLP with d inputs and p outputs.
- User should define the network architecture: number of hidden layers, hidden units per layer, type of activation functions.
- $o(x^n; w)$: the MLP ouput vector with input x^n, and $\mathbf{w=(w_1,w_2,\ldots,w_L)^T}$ the vector of weights and biases.
- Training: specify **w**.

MLP training

$$E(w)=\sum_{n=1}^{N} E^n(w), \quad E^n(w) = \frac{1}{2}\|t^n - o(x^n;w)\|^2 = \frac{1}{2}\sum_{m=1}^{p}(t_{nm} - o_m(x^n;w))^2$$

- E(w): error (loss) function to be minimized wrt to w (**sum of squared errors for regression**).
- Typically gradient descent approaches are used.
- The partial derivatives of E^n wrt to w_i is needed:

error backpropagation

- Given a training example (x,t), **error backpropagation computes the partial derivatives of the loss (error) function for this training example wrt to the weights,** in a feedforward neural network with inner-product units and differentiable activation functions (e.g. MLP).

Backpopagation

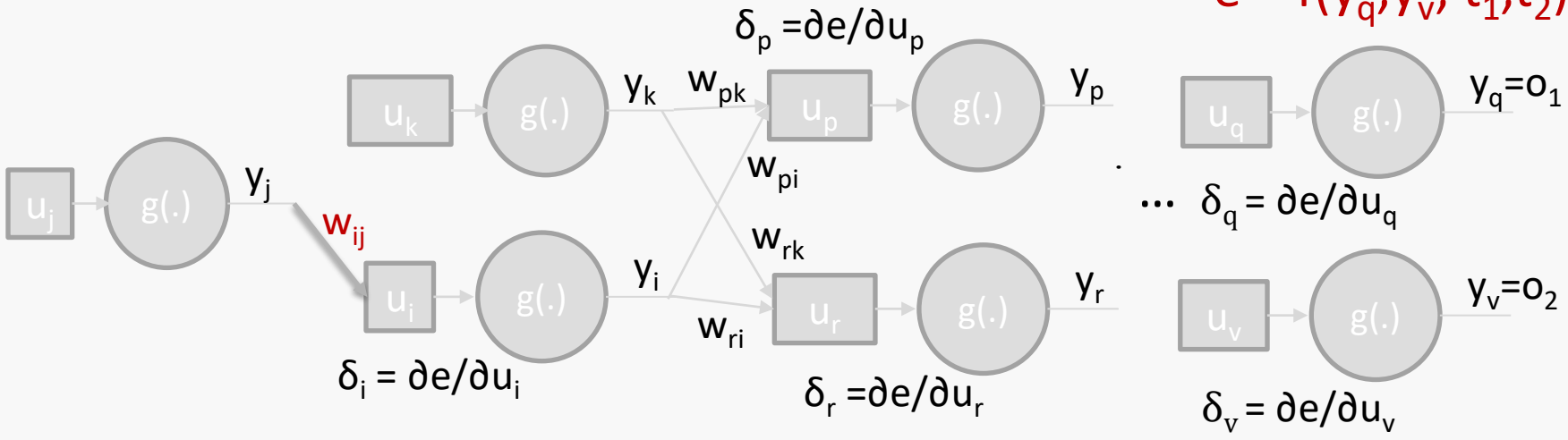
- **Forward pass**: For input x^n we compute (and store) the output y of each unit in the network.
- **Backward (reverse) pass (computes an error value δ for units in the output and hidden layers)**
  - First the errors in the output layer (H+1) are computed, comparing the network outputs o_i with the targets t_{ni}.
  - The error signals are propagated back through the hidden layers to compute the error of each hidden unit.
- **Partial derivative of a connection weight:**

(error δ of the source unit) x (output y of the destination unit)

Εξήγηση backpropagation

Σφάλμα

$e = f(y_q, y_v; t_1, t_2)$



$u_p = w_{pi} y_i + w_{pk} y_k$ $\quad \partial u_p/\partial y_i = w_{pi}$ $\quad u_r = w_{ri} y_i + w_{rk} y_k$ $\quad \partial u_r/\partial y_i = w_{ri}$

Ονομάζουμε $\delta_i = \partial e/\partial u_i$ (σφάλμα νευρώνα i)

$\delta_q = \partial e/\partial y_q \, \partial y_q/\partial u_q = \partial e/\partial y_q \; g'(u_q)$ (για νευρώνα εξόδου q, ομοίως για v)

Για εσωτερικό νευρώνα i:

$\partial e/\partial u_i = \partial e/\partial y_i \cdot \partial y_i/\partial u_i = g'(u_i) \cdot \partial e/\partial y_i$ (1)

$\partial e/\partial w_{ij} = \partial e/\partial u_i \cdot \partial u_i/\partial w_{ij}$

$\Rightarrow \partial e/\partial w_{ij} = \delta_i \cdot y_j$

$\partial e/\partial y_i = \partial e/\partial u_p \cdot \partial u_p/\partial y_i + \partial e/\partial u_r \cdot \partial u_r/\partial y_i$

$= \delta_p \cdot w_{pi} + \delta_r \cdot w_{ri}$ (2)

(1),(2) $\Rightarrow \delta_i = g'(u_i) \cdot (\delta_p \cdot w_{pi} + \delta_r \cdot w_{ri})$

Backpropagation

- **Compute δ (backward pass)**
  - Output units (layer H+1) (activation g_{H+1}) **(for squared error loss)**

$$\delta_i^{(H+1)} = g'_{H+1}(u_i^{(H+1)})(o_i - t_{ni}),\ i=1,...,p$$

$$\delta_i^{(H+1)} = (o_i - t_{ni}),\ i=1,...,p \text{ (linear acitvation)}$$

$$\delta_i^{(H+1)} = o_i(1-o_i)(o_i - t_{ni}),\ i=1,...,p \text{ (logistic activation)}$$

  - Hidden units: για επίπεδο h=H,…,1 (activation g_h)

$$\delta_i^{(h)} = g'_h(u_i^{(h)})\sum_{j=1}^{d_{h+1}} w_{ji}^{(h+1)}\delta_j^{(h+1)},\quad i=1,...,d_h$$

$$\delta_i^{(h)} = y_i^{(h)}(1-y_i^{(h)})\sum_{j=1}^{d_{h+1}} w_{ji}^{(h+1)}\delta_j^{(h+1)},\quad i=1,...,d_h \text{ (logistic activation)}$$

Backpropagation

- Partial Derivative for connection weight:

$$\frac{\partial E^n}{\partial w_{ij}^{(h)}} = \delta_i^{(h)} y_j^{(h-1)}$$

- Partial derivative for bias

$$\frac{\partial E^n}{\partial w_{i0}^{(h)}} = \delta_i^{(h)}$$

MLP training with gradient descent (batch update)

1. Weight initialization (t:=0), (random values in (-1,1)), set learning rate ρυθμού η.
2. At each iteration t (epoch), let w(t) the weight vector
   - We set: $\frac{\partial E}{\partial w_i} = 0,\ i=1,\ldots,L$
   - For n=1,…,N
     - Apply backpropagation for (x^n,t^n) and compute $\frac{\partial E^n}{\partial w_i},\ i=1,\ldots,L$
     - Update derivatives: $\frac{\partial E}{\partial w_i} := \frac{\partial E}{\partial w_i} + \frac{\partial E^n}{\partial w_i}$
   - Update weights: $w_i(t+1)=w_i(t)-\eta\frac{\partial E}{\partial w_i},\ i=1,..,L$
   - Until termination

MLP training with gradient descent (on-line update)

1. Initialize weights w(0) (random values in (-1,1)), set learning rate n. Initialize iteration counter (τ:=0), epoch counter (t:=0).
2. At the beginning of epoch t, let w(τ) the weight vector
   - Start of epoch t. For n=1,…,N
     - Apply backpropagation (x^n,t^n) to compute $\frac{\partial E^n}{\partial w_i}$, $i=1,\ldots,L$
     - Update weights: $w_i(\tau+1)=w_i(\tau)-n\frac{\partial E^n}{\partial w_i}$, $i=1,..,L$
     - τ:=τ+1
   - End of epoch t, test for termination (error difference, number of epochs, early stopping)

MLP training

- Alternative gradient-based approaches (gradient is computed through back-propagation)
  - Variants of gradient descent (e.g. use momentum term)
  - **Mini-batch approach**
  - Scaled conjugate gradients (commonly used)
  - Second-order methods (only for small networks and datasets)
    - BFGS
    - Levenberg–Marquardt
- All gradient-based methods converge to a local minimum of the loss function (multiple random restarts could be used)

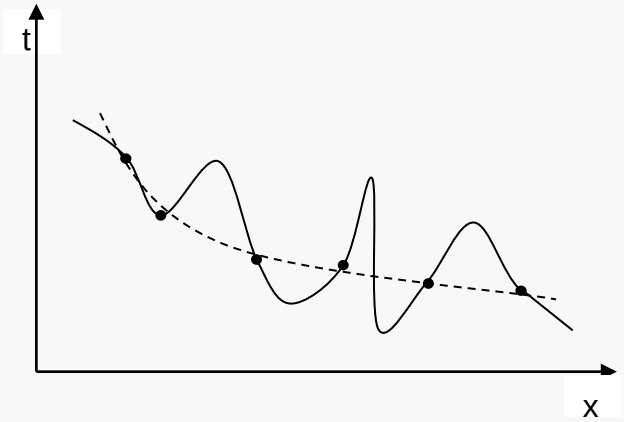
MLP for classification

- **1-out of-p encoding** for p classes $C_1,\ldots,C_p$
  - Every target vector has p components $(t_1,\ldots,t_p)$
  - The target for class C_k : $t_k=1$ and $t_i=0$ for $i\neq k$.
  - An input is classified to the class corresponding to maximum output
- It possible to use output units with sigmoid activations and train using least squares error
- It is **preferable** to **use softmax activation and train using cross-entropy** (analogous to multi-class logistic regression) (what is the error δ of the output units?)
- In the case of **two classes** we use **one logistic output** with targets **t=1** and **t=0**.
  - Train using cross-entropy for two classes (analogous to logistic regression). Decision threshold could be set to 0.5.

Learning and Generalization

Generalization

- **Generalization: successful predictions on unseen examples**
- **Occam's razor:**
  - Prefer the simplest model that fits well to the data.
- **Alternatively: Bias variance dilemma**
  - **generalization error = bias + variance**
  - **Bias:** how well the model fits the training data (small for large models)
  - **Variance:** how small perturbations in the training set affect the training results (large for large models)



Generalization

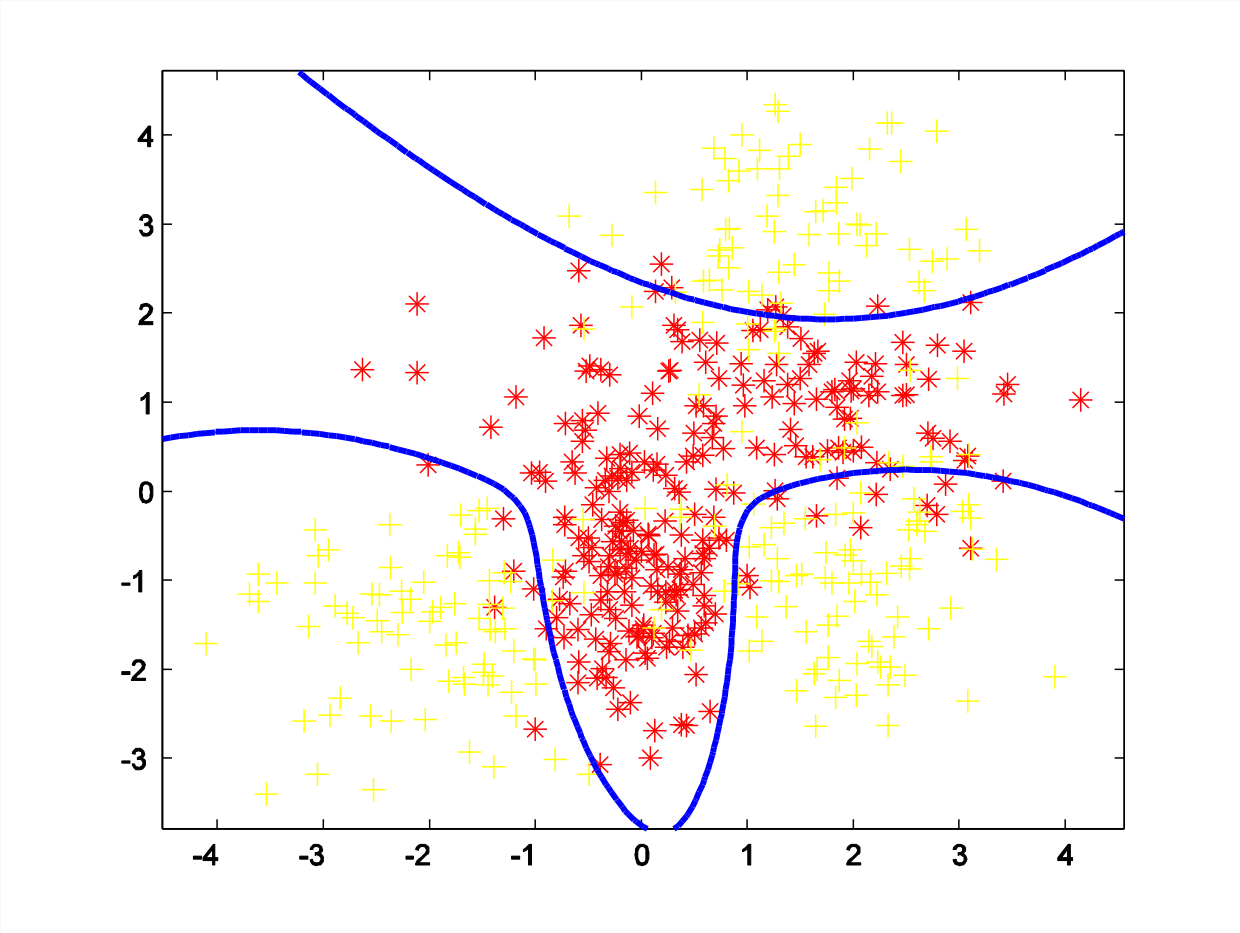
- Small network: possibility for undertraining.
- Large network: possibility for overtraining
- We seek for networks with ‘optimal’ complexity
- We need to have an estimate of the generalization error of a network.
  - Use of a test set (not used for training) (holdout)
  - K-fold Cross-validation
  - Leave-one-out (K=number of examples)

Cross-Validation

- **K-fold cross-validation (**K-CV):
  - Split dataset D into K *disjoint* subsets (folds) D_1 ,…, D_K (usually K=10).
  - For each subset D_i (i=1 ,…, K), train a model using D-D_i as training set and compute generalization error (ge_i) using D_i as test set.
  - Compute ge = average(ge_i)
  - Depends (somehow) on initial splitting
- **Leave-one-out (**K=N): deterministic, more reliable, but computationally expensive
- We use cross-validation to select the best model (e.g. network architecture) .
- The final solution is obtained by training the selected model on the whole dataset.

Example

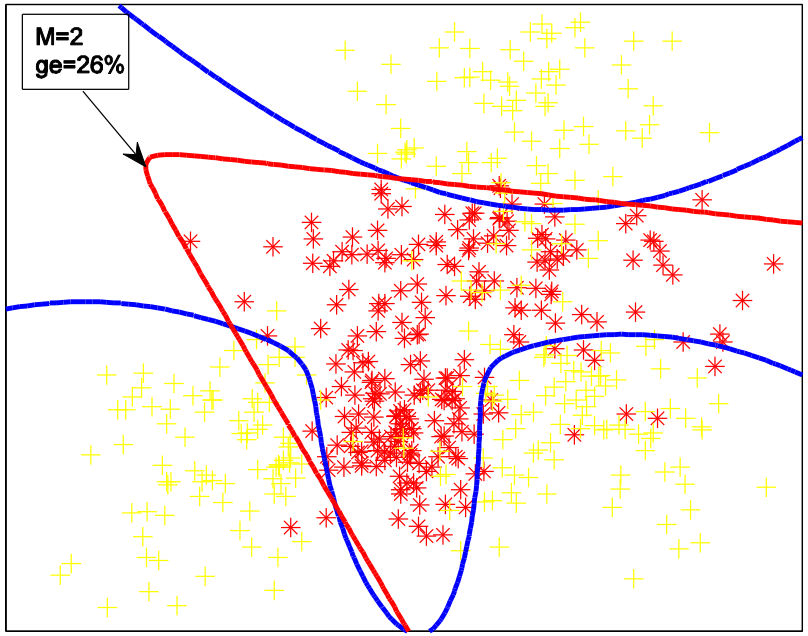
Dataset (theoretical generalization: 89%)

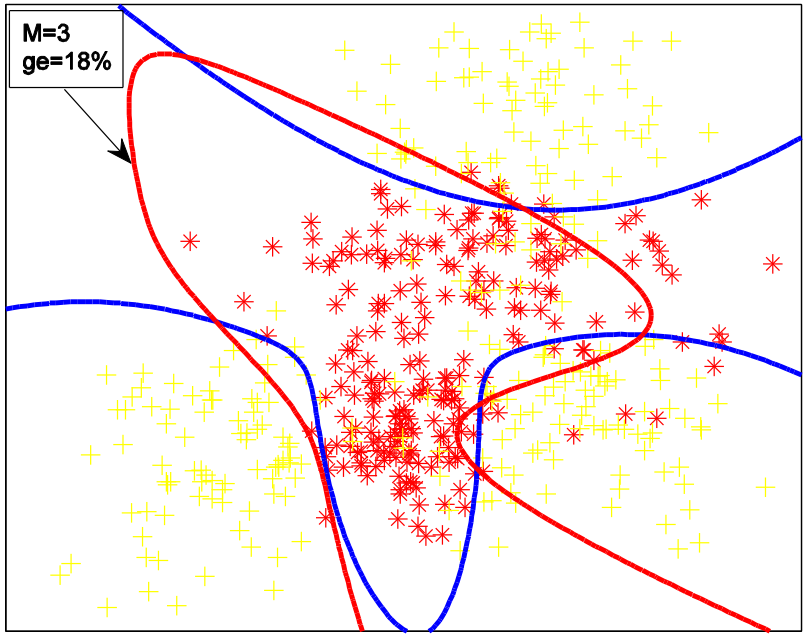


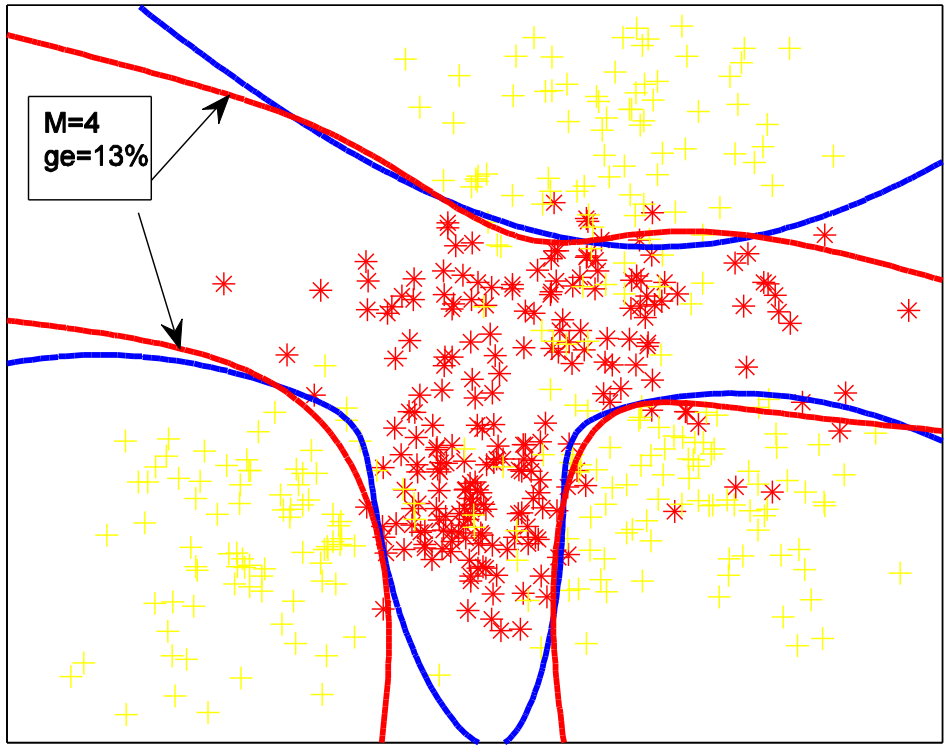
Example

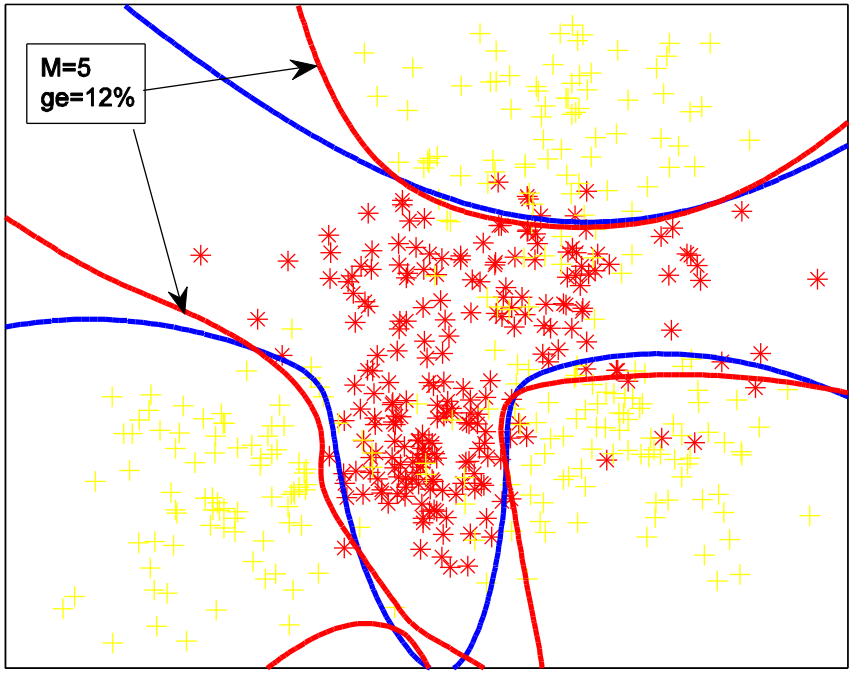
Optimum at M=5, gen=100-12=88%

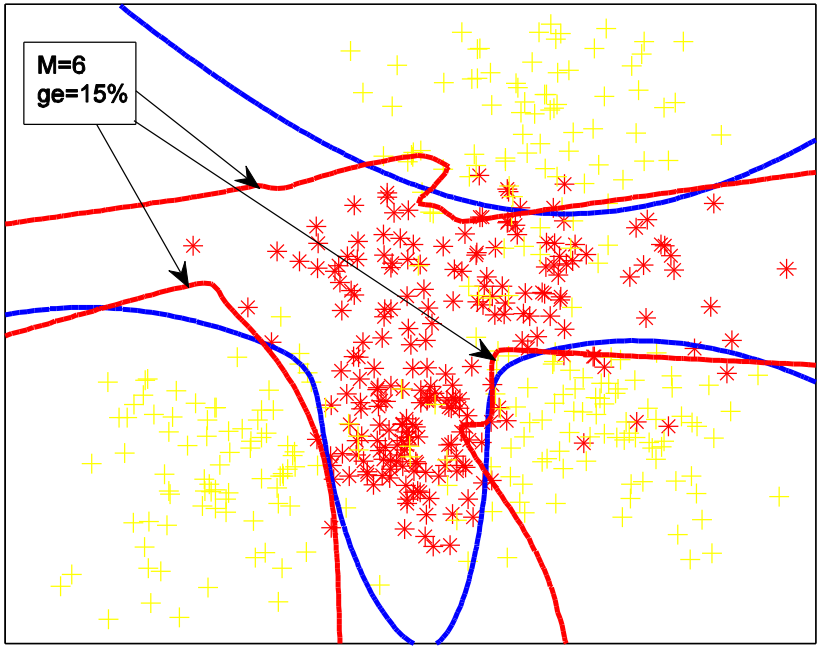
M	Generalization Error(10-CV)
2	28%
3	18%
4	13%
5	12%
6	15%
7	15%

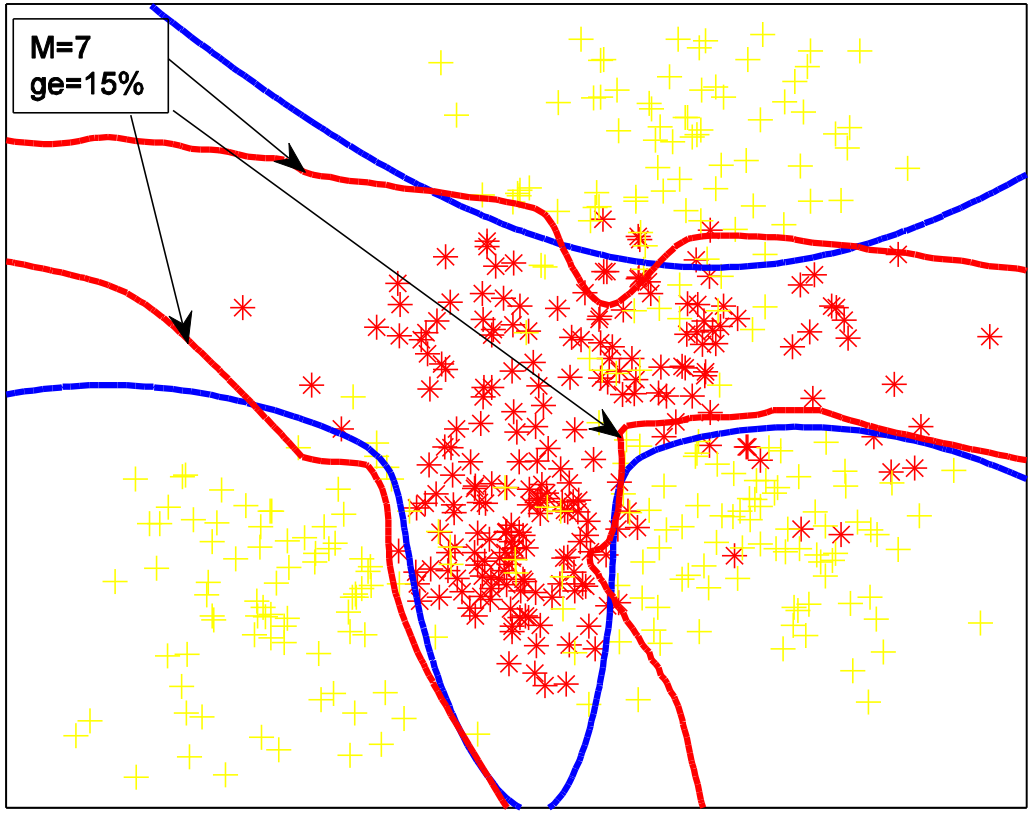

M=2
ge=26%


M=3
ge=18%


M=4
ge=13%


M=5
ge=12%


M=6
ge=15%


M=7
ge=15%

Overtraining

- In case we train a model **more flexible than necessary**, it is possible to achieve **overtraining**.
- **Overtraining: low error in training examples, large error on unseen examples (poor generalization)**

- There are several techniques to **avoid overtraining**:
  - Regularization (weight decay)
  - Early stopping
  - Weight sharing
  - Model Averaging
  - Dropout

Weight Decay

- Use a regularization (penalty term):

$$R(w)=\sum_{i=1}^{L} w_i^2$$

- Forces weights to obtain low values during training.
- The error function for training is:

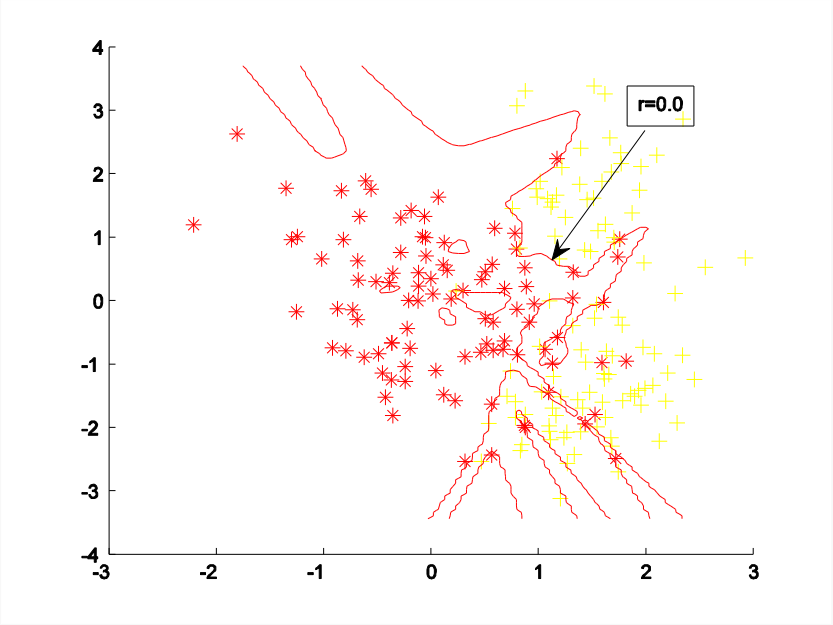
$$E_R(w)=E(w)+rR(w)=E(w)+r\sum_{i=1}^{L} w_i^2$$

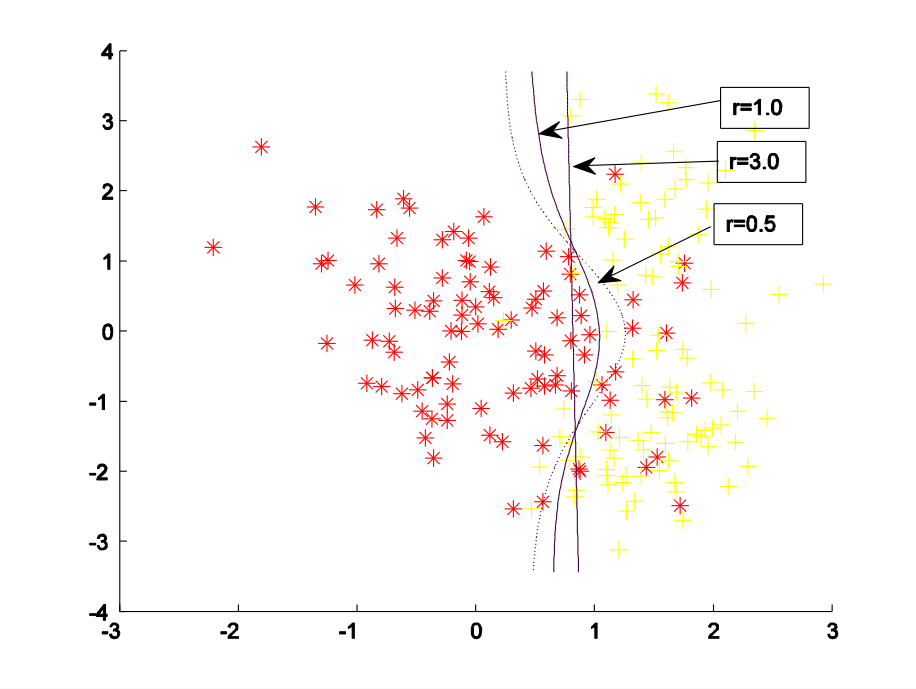
  E(w) is the usual error function.
- parameter r determines the regularization strength.

- Weight updates: $w_i(t+1)=w_i(t)-\eta\left(\frac{\partial E}{\partial w_i}+2rw_i(t)\right)$

Weight Decay

- MLP with 1 hidden layer (20 hidden neurons)

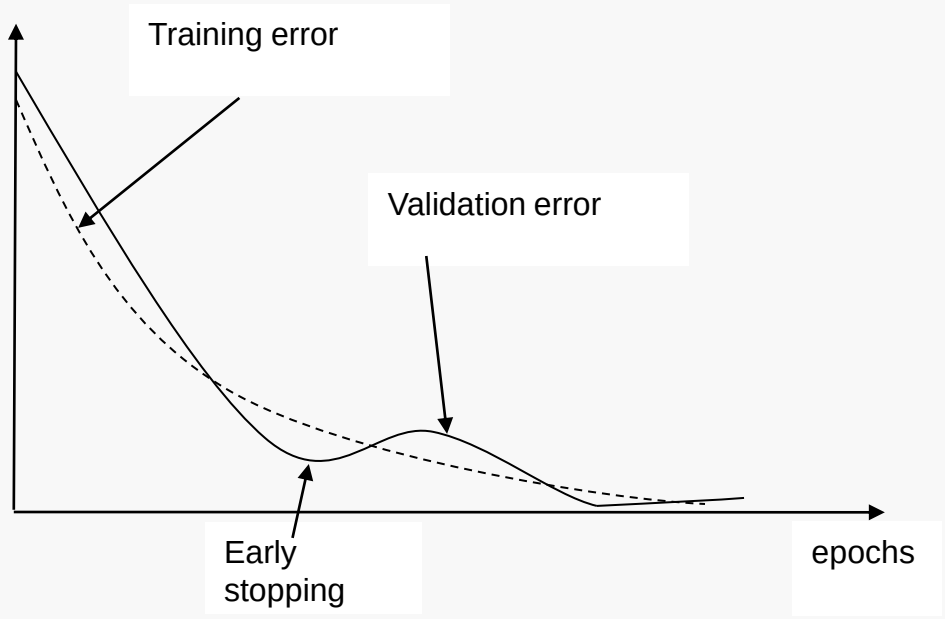




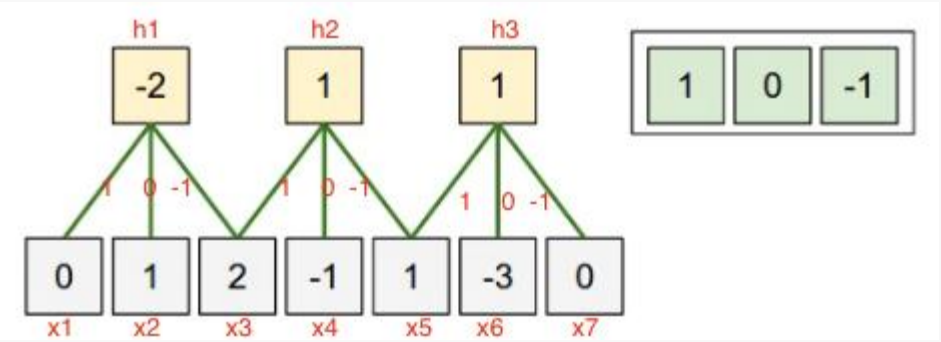
Early stopping

- Use of **validation set** (different from training and test set)
- We train a **(large)** MLP (weight updates) using the training set
- At regular intervals (e.g. 10 epochs) we **'freeze' training** and using the current weights we compute **the error in the validation set (validation error)**.
- In the beginning of training, validation error drops along with training error.
- There may be a time point when the validation error starts to increase. This is an indication of overtraining and a suggestion to stop training.
- Possible disadvantage: we must remove examples from the training set, to use them in the validation set.
- **Training**, **validation** and **test** sets should not contain common examples.

Early stopping



Weight Sharing

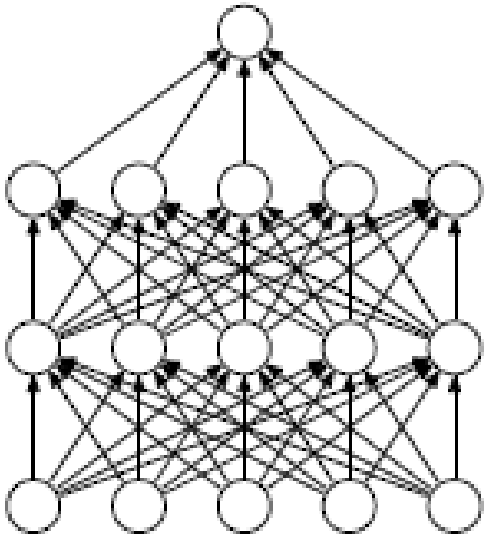


- One or more connections have (share) the same weight value w.
- The shared connections are usually in the same connection layer.
- In this way the **number of network parameters** (adjustable weights) is **reduced**, thus the model flexibility is reduced.
- The **gradient** of the error with respect to the **shared weight** parameter w is equal to the **sum of the gradients** with respect to the **individual connection weights** that share the parameter w.
- Successfully used in convolutional neural networks to achieve translation invariance in computer vision problems.

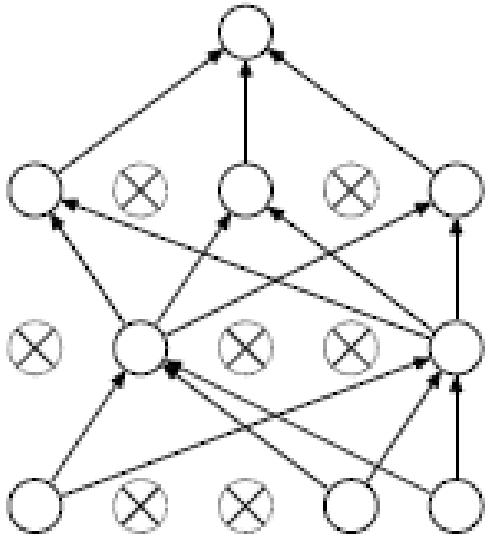
Model Averaging

- **Averaging reduces variance**
- Train several networks
  - Same or different architecture
  - Same dataset or random bootstraps of the dataset
- To predict for an example
  - Use the example as input to all trained models
  - Average the outputs (regression) or perform voting (classification)
  - Each model may also have weight for his decision
- **Bagging and boosting** are successful methodologies (same model trained on different datasets obtained by sampling with replacement from the original dataset).
- Model averaging is expensive for large models (e.g. deep neural networks)

Dropout



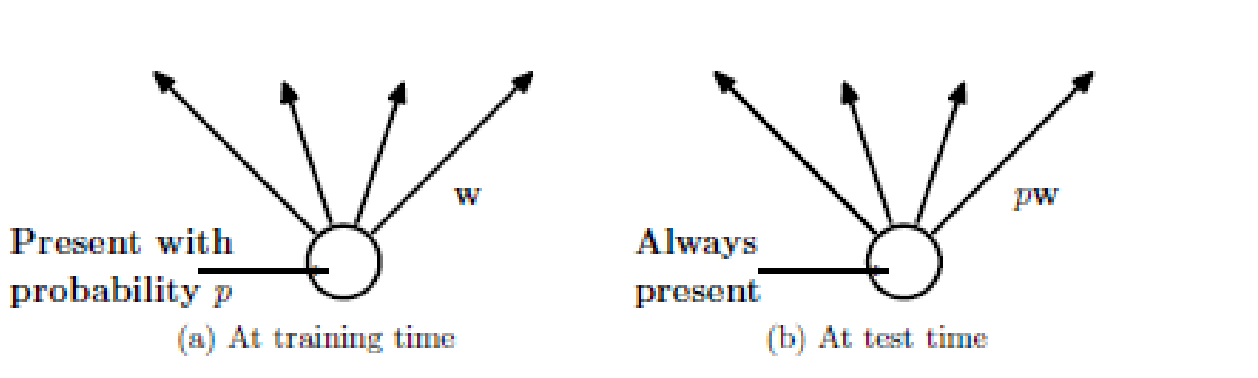
(a) Standard Neural Net



(b) After applying dropout.

- **Dropout** is a **regularization** technique for reducing overfitting in neural networks.
- It is a very efficient way of performing **model averaging** with neural networks.
- **Drop out a node: temporarily remove it from the network (along with its connections)**

Dropout

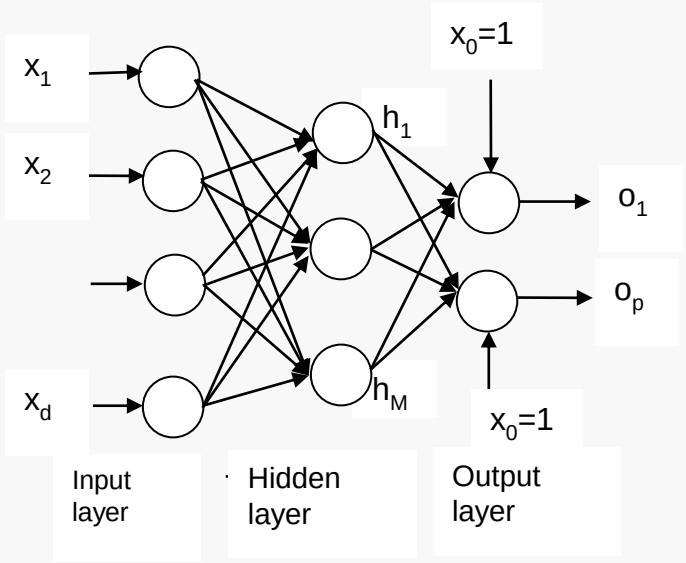


- Typically each node is removed with probability p (dropout probability) independent of the other nodes (e.g. p=0.5)
- Training using on-line gradient descent. For each training example, we sample a ‘thinned’ network by dropping out units and perform backpropagation training on the ‘thinned’ network.
- At test time all nodes are present. The weights of each node are p*w, instead of w (p is the dropout probability). In this way, it is like performing **averaging** over all ‘thinned’ networks.
- Possible to use dropout at connection level (instead of node level).

RBF Neural Network

RBF Network

- RBF (radial basis function) networks.
- **Feedforward** neural network
- MLP alternative.
- **Only one hidden layer with RBF hidden units**
- Output units similar to MLP

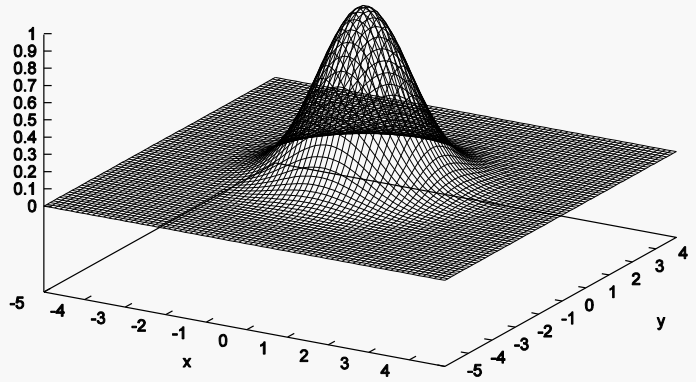


RBF functions

- **RBF hidden unit j**:

$$h_j(x)=\exp\left(-\frac{\|x-w_j\|^2}{2\sigma_j^2}\right)=\exp\left(-\frac{\sum_{l=1}^{d}(x_l-w_{jl})^2}{2\sigma_j^2}\right)$$

- $w_j=(w_{j1},\ldots, w_{jd})^T$ center, σ_j radius



RBF network

- RBF is a universal approximator (as MLP)
- Trained by minimizing a loss function (e.g. using gradient descent)
- Gradient computation is easy (only one hidden layer)
- Initialization of RBF centers is difficult (clustering may be used)
- Number of RBF hidden units may be determined using model selection techniques (e.g. cross-validation)