Learning Real-Valued Functions

- Learning to approximate real-valued functions
- Bayesian recipe for learning real-valued functions
- Brief digression – continuity, differentiability, Taylor series approximation of functions
- Learning linear functions using gradient descent in weight space
- Universal function approximation theorem
- Learning nonlinear functions using gradient descent in weight space
- Practical considerations and examples
Review: Bayesian Recipe for Learning

\[ P(h \mid D) = \frac{P(D \mid h)P(h)}{P(D)} \]

- \( P(h) \) = prior probability of hypothesis \( h \)
- \( P(D) \) = prior probability of training data \( D \)
- \( P(h \mid D) \) = probability of \( h \) given \( D \)
- \( P(D \mid h) \) = probability of \( D \) given \( h \)

Bayesian recipe for learning

Choose the most likely hypothesis given the data

\[ h_{\text{MAP}} = \arg \max_{h \in H} P(h \mid D) \quad \text{(Maximum a posteriori hypothesis)} \]
\[ = \arg \max_{h \in H} \frac{P(D \mid h)P(h)}{P(D)} \]
\[ = \arg \max_{h \in H} P(D \mid h)P(h) \]

If \( \forall h_i, h_j \in H \ P(h_i) = P(h_j) \),

\[ h_{\text{ML}} = \arg \max_{h \in H} P(D \mid h) \quad \text{(Maximum likelihood hypothesis)} \]
Learning a Real Valued Function

• Consider a real-valued target function $f$
• Training examples $\langle x_i, d_i \rangle$, where $d_i$ is noisy training value $d_i = f(x_i) + e_i$
• $e_i$ is random variable (noise) drawn independently for each $x_i$ according to Gaussian distribution
  with zero mean
• $\Rightarrow d_i$ has mean $f(x_i)$ and same variance

Then the maximum likelihood hypothesis $h_{ML}$ is one that minimizes the sum of squared error:

$$h_{ML} = \arg \min_{h \in H} \sum_{i=1}^{m} (d_i - h(x_i))^2$$

Maximize natural log of this instead...
Learning a Real Valued Function

\[ h_{ML} = \arg \max_{h \in H} \sum_{i=1}^{m} \ln \frac{1}{\sqrt{2\pi\sigma^2}} - \frac{1}{2} \left( \frac{d_i - h(x_i)}{\sigma} \right)^2 \]

\[ = \arg \max_{h \in H} \sum_{i=1}^{m} - \frac{1}{2} \left( \frac{d_i - h(x_i)}{\sigma} \right)^2 \]

\[ = \arg \max_{h \in H} \sum_{i=1}^{m} (d_i - h(x_i))^2 \]

\[ = \arg \min_{h \in H} \sum_{i=1}^{m} (d_i - h(x_i))^2 \]

Maximum Likelihood hypothesis is one that minimizes the mean squared error!

Approximating a linear function using a linear neuron

\[ y = \sum_{i=0}^{n} w_i x_i \]
Learning Task

\[ \mathbf{W} = [W_0, \ldots, W_n] \] is the weight vector
\[ \mathbf{X}_p = [X_{0p}, \ldots, X_{np}] \] is the \( p \)th training sample
\[ y_p = \sum_i W_i X_{ip} = \mathbf{W} \cdot \mathbf{X}_p \] is the output of the neuron for input \( \mathbf{X}_p \)
\[ \mathbf{X}_p = f(\mathbf{X}_p) \] is the desired output for input \( \mathbf{X}_p \)
\[ e_p = (d_p - y_p) \] is the error of the neuron on input \( \mathbf{X}_p \)
\[ S = \{\mathbf{X}_p, d_p\} \] is a (multi) set of training examples
\[ E_S(\mathbf{W}) = E_S(W_0, W_1, \ldots, W_n) = \frac{1}{2} \sum_p e_p^2 \] is the estimated error of \( \mathbf{W} \) on training set \( S \)

Goal: Find \( \mathbf{W}^* = \arg \min_w E_S(\mathbf{W}) \)

Learning linear functions

The error is a quadratic function of the weights in the case of a linear neuron.
Learning linear functions

\[ w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i} \]

\[
\frac{\partial E}{\partial w_i} = \frac{1}{2} \frac{\partial}{\partial w_i} \left( \sum_p e_p^2 \right) = \frac{1}{2} \left( \sum_p \frac{\partial}{\partial w_i} (e_p^2) \right) \\
= \frac{1}{2} \left( \sum_p \frac{1}{w_i} (\frac{\partial w_i}{\partial w_i} e_p^2) \frac{\partial}{\partial w_i} \right) = \frac{1}{2} \sum_p e_p \left( \frac{\partial}{\partial w_i} (\sum_j w_j x_{jp}) \right) \\
= - \sum_p (d_p - y_p) \left( \frac{\partial}{\partial w_i} (w_i x_{ip}) + \frac{\partial}{\partial w_i} (\sum_j w_j x_{jp}) \right) \\
= - \sum_p (d_p - y_p) x_{ip} \\

w_i \leftarrow w_i + \eta \sum_p (d_p - y_p) x_{ip} \]

Least Mean Square Error (LMSE) Learning Rule

\[ w_i \leftarrow w_i + \eta \sum_p (d_p - y_p) x_{ip} \]

Batch Update

Per sample Update
Choice of learning rate

In theory, infinitesimally small (why?)

Per sample Update

Batch Update

Where $\lambda_{\text{max}}$ is the largest Eigen Value of the Hessian of $E_S$ (matrix of second order partial derivatives of $E_S$ with respect to the weights)

Recall that Eigen values of a matrix $A$ are given by solutions of $|A-\lambda I|=0$

Choice of learning rate in practice – Momentum update

$$w_i(t+1) = w_i(t) + \Delta w_i(t)$$

$$\Delta w_i(t) = -\eta \left. \frac{\partial E}{\partial w_i} \right|_{w_j=w_i(t)} + \alpha \Delta w_i(t-1) \text{ where } 0 < \alpha < 1$$

$$= -\eta \sum_{t=0}^{t} \alpha^{t-t} \left. \frac{\partial E}{\partial w_i} \right|_{w_j=w_i(t)}$$

The momentum update allows effective learning rate to increase when feasible and decrease when necessary. Converges for $0 \leq \alpha < 1$
Learning approximations of nonlinear functions from data – the generalized delta rule

• Motivations
• Universal function approximation theorem (UFAT)
• Derivation of the generalized delta rule
• Back-propagation algorithm
• Practical considerations
• Applications

Motivations

• Psychology – Empirical inadequacy of behaviorist theories of learning – simple reward-punishment based learning models are incapable of learning functions (e.g., exclusive OR) which are readily learned by animals (e.g., monkeys)
• Artificial Intelligence – the need for learning highly nonlinear functions where the form of the nonlinear relationship is unknown a-priori
• Statistics – Limitations of linear regression in fitting data when the relationship is highly nonlinear and the form of the relationship is unknown
• Control – Need for nonlinear control methods
Kolmogorov’s theorem (Kolmogorov, 1940)

- Any continuous function from input to output can be expressed in the form

\[
g(x_1, x_2, \ldots, x_n) = \sum_{j=1}^{2n+1} g_j \left( \sum_{i} u_{ij} (x_i) \right)
\]

\[\forall (x_1, x_2, \ldots, x_n) \in I^n (I = [0,1]; n \geq 2)\]

by choosing proper nonlinearities \( g_j \) and the weights and \( u_{ij} \)

Universal function approximation theorem (Cybenko, 1989)

- Let \( \varphi : \mathbb{R} \rightarrow \mathbb{R} \) be a non-constant, bounded (hence non-linear), monotone, continuous function. Let \( I_N \) be the \( N \)-dimensional unit hypercube in \( \mathbb{R}^N \).

- Let \( C(I_N) = \{ f : I_N \rightarrow \mathbb{R} \} \) be the set of all continuous functions with domain \( I_N \) and range \( \mathbb{R} \). Then for any function \( f \in C(I_N) \) and any \( \varepsilon > 0 \), \( \exists \) an integer \( L \) and a sets of real values \( \theta, \alpha_j, \theta_j, w_{ji} \) (\( 1 \leq j \leq L; 1 \leq i \leq N \)) such that

\[
F(x_1, x_2, \ldots, x_n) = \sum_{j=1}^{L} \alpha_j \varphi \left( \sum_{i=1}^{N} w_{ji} x_i - \theta_j \right) - \theta
\]

is a uniform approximation of \( f \) – that is,

\[\forall (x_1, x_2, \ldots, x_n) \in I_N, \ \left| F(x_1, x_2, \ldots, x_n) - f(x_1, x_2, \ldots, x_n) \right| < \varepsilon\]
Universal function approximation theorem (UFAT)

\[ F(x_1, x_2, ..., x_n) = \sum_{j=1}^{\infty} \alpha_j \varphi \left( \sum_{i=1}^{N} w_{ij} x_i - \theta_j \right) - \theta \]

- Unlike Kolmogorov’s theorem, UFAT requires only one kind of nonlinearity to approximate any arbitrary nonlinear function to any desired accuracy
- The sigmoid function satisfies the UFAT requirements
  \[ \varphi(z) = \frac{1}{1 + e^{-\alpha z}}; \quad \alpha > 0 \]
  \[ \lim_{z \to -\infty} \varphi(z) = 0; \quad \lim_{z \to +\infty} \varphi(z) = 1 \]

Similar universal approximation properties can be guaranteed for other functions – e.g., radial basis functions

Universal function approximation theorem

- UFAT guarantees the existence of arbitrarily accurate approximations of continuous functions defined over bounded subsets of \( \mathbb{R}^N \)
- UFAT tells us the representational power a certain class of multi-layer networks relative to the set of continuous functions defined on bounded subsets of \( \mathbb{R}^N \)
- UFAT is not constructive – it does not tell us how to choose the parameters to construct a desired function
- To learn an unknown function from data, we need an algorithm to search the hypothesis space of multilayer networks
- Generalized delta rule allows nonlinear function to be learned from the training data
Alternatives

• Brute force – select a complete set of nonlinear basis functions (e.g., all polynomials of degree from 0 to \( N \)) to map the \( N \)-dimensional input into a very high dimensional feature space where a linear mapping to desired outputs exists – needs too many parameters to be determined from a limited number of training samples

\[
g \left( \sum_{i=1}^{N} g_i(x_i) + w_0 \right)
\]

• Additive models –
  • select some nonlinear functions and try to adjust the parameters of the chosen nonlinear functions to fit the data – it is hard to know a priori which nonlinear functions to choose

• Projection pursuit – closely related model but differing in algorithmic details

\[
\sum_{j=1}^{L} w_j f_j \left( \sum_{i=1}^{N} w_{ji} x_i + w_{j0} \right) + w_0
\]

different parameters are learned in groups – first \( w_{10} \ldots w_{1N} \) then \( w_{20} \ldots w_{2N} \) through \( w_{J0} \ldots w_{JN} \) followed by \( w_j \) (\( j = 0 \ldots J \))

iterating until some desired error criterion is met
Feed-forward neural networks

- A feed-forward 3-layer network consists of 3 layers of nodes
  - Input nodes
  - Hidden nodes
  - Output nodes
  - interconnected by modifiable weights from input nodes to the hidden nodes and the hidden nodes to the output nodes
- More general topologies (with more than 3 layers of nodes, or connections that skip layers – e.g., direct connections between input and output nodes) are also possible

A three layer network that approximates the exclusive or function
Three-layer feed-forward neural network

- A single bias unit is connected to each unit other than the input units.
- Net input

\[ n_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} \equiv W_j \cdot \mathbf{x}, \]

where the subscript \( i \) indexes units in the input layer, \( j \) in the hidden; \( w_{ji} \) denotes the input-to-hidden layer weights at the hidden unit \( j \).

- The output of a hidden unit is a nonlinear function of its net input. That is, \( y_j = f(n_j) \) e.g.,

\[ y_j = \frac{1}{1 + e^{-n_j}} \]

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

\[ n_k = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = W_k \cdot \mathbf{Y}, \]

where the subscript \( k \) indexes units in the output layer and \( n_H \) denotes the number of hidden units.

- The output can be a linear or nonlinear function of the net input e.g.,

\[ y_k = n_k \]
Computing nonlinear functions using a feed-forward neural network

Realizing non linearly separable class boundaries using a 3-layer feed-forward neural network
Learning nonlinear functions

- Given a training set determine
- Network structure – number of hidden nodes or more generally, network topology
  - Start small and grow the network
  - Start with a sufficiently large network and prune away the unnecessary connections
- For a given structure, determine the parameters (weights) that minimize the error on the training samples (e.g., the mean squared error)
- For now, we focus on the latter

Generalized delta rule – error back-propagation

- Challenge – we know the desired outputs for nodes in the output layer, but not the hidden layer
- Need to solve the credit assignment problem – dividing the credit or blame for the performance of the output nodes among hidden nodes
- Generalized delta rule offers an elegant solution to the credit assignment problem in feed-forward neural networks in which each neuron computes a differentiable function of its inputs
- Solution can be generalized to other kinds of networks, including networks with cycles

Copyright Vasant Honavar, 2006.
Feed-forward networks

- Forward operation (computing output for a given input based on the current weights)
- Learning – modification of the network parameters (weights) to minimize an appropriate error measure
- Because each neuron computes a differentiable function of its inputs if error is a differentiable function of the network outputs, the error is a differentiable function of the weights in the network – so we can perform gradient descent!

A fully connected 3-layer network
Generalized delta rule

Let $t_{kp}$ be the $k$-th target (or desired) output for input pattern $X_p$, and $z_{kp}$ be the output produced by $k$-th output node and let $W$ represent all the weights in the network.

Training error:

$$E_S(W) = \frac{1}{2} \sum_p \sum_{k=1}^M (t_{kp} - z_{kp})^2 = \sum_p E_p(W)$$

The weights are initialized with pseudo-random values and are changed in a direction that will reduce the error:

$$\Delta w_{ji} = -\eta \frac{\partial E_S}{\partial w_{ji}} \quad \Delta w_{kj} = -\eta \frac{\partial E_S}{\partial w_{kj}}$$

$\eta > 0$ is a suitable learning rate $W \leftarrow W + \Delta W$

Hidden–to-output weights

$$\frac{\partial E_p}{\partial w_{kj}} = \frac{\partial E_p}{\partial n_{kp}} \cdot \frac{\partial n_{kp}}{\partial w_{kj}} \quad \frac{\partial n_{kp}}{\partial w_{kj}} = y_{jp}$$

$$\frac{\partial E_p}{\partial n_{kp}} = \frac{\partial E_p}{\partial z_{kp}} \cdot \frac{\partial z_{kp}}{\partial n_{kp}} = -(t_{kp} - z_{kp})(1)$$

$$w_{kj} \leftarrow w_{kj} - \eta \frac{\partial E_p}{\partial w_{kj}} = w_{kj} + (t_{kp} - z_{kp}) y_{jp} = w_{kj} + \delta_{kp} y_{jp}$$
Generalized delta rule

Weights from input to hidden units

\[
\frac{\partial E_p}{\partial w_{ji}} = \sum_{k=1}^{M} \frac{\partial E_p}{\partial z_{kp}} \frac{\partial z_{kp}}{\partial w_{ji}} = \sum_{k=1}^{M} \frac{\partial E_p}{\partial z_{kp}} \frac{\partial z_{kp}}{\partial y_{jp}} \cdot \frac{\partial y_{jp}}{\partial n_{jp}} \cdot \frac{\partial n_{jp}}{\partial w_{ji}}
\]

\[
= \sum_{k=1}^{M} \frac{\partial}{\partial z_{kp}} \left[ \frac{1}{2} \sum_{l=1}^{M} (t_{lp} - y_{lp})^2 \right] \left( w_{kj} y_{jp} (1 - y_{jp}) x_{ip} \right)
\]

\[
= \sum_{k=1}^{M} (t_{kp} - y_{kp}) \left( w_{kj} y_{jp} (1 - y_{jp}) x_{ip} \right)
\]

\[
= \left( \sum_{l=1}^{M} \delta_{lp} (w_{kj} y_{jp} (1 - y_{jp}) x_{ip}) \right) \delta_{jp}
\]

\[
= -\delta_{jp} x_{ip}
\]

\[
w_{ji} \leftarrow w_{ji} + \eta \delta_{jp} x_{ip}
\]

Back propagation algorithm

- Start with small random initial weights
- Until desired stopping criterion is satisfied do
  - Select a training sample from \( S \)
  - Compute the outputs of all nodes based on current weights and the input sample
  - Compute the weight updates for output nodes
  - Compute the weight updates for hidden nodes
  - Update the weights
Using neural networks for classification

Network outputs are real valued.

How can we use the networks for classification?

\[ F(X_p) = \arg \max_k z_{kp} \]

Classify a pattern by assigning it to the class that corresponds to the index of the output node with the largest output for the pattern.

Training multi-layer networks – Some Useful Tricks

• Initializing weights to small random values that place the neurons in the linear portion of their operating range for most of the patterns in the training set improves speed of convergence e.g.,

\[
W_{ji} = \pm \frac{1}{2N} \sum_{i=1, \ldots, N} \frac{1}{|x_i|}
\]

For input to hidden layer weights with the sign of the weight chosen at random.

\[
W_{ij} = \pm \frac{1}{N} \sum_{i=1, \ldots, N} \left( \frac{1}{\sum w_{ji}} \right)
\]

For hidden to output layer weights with the sign of the weight chosen at random.
Some Useful Tricks

• **Use of momentum** term allows the effective learning rate for each weight to adapt as needed and helps speed up convergence – in a network with 2 layers of weights,

\[
\begin{align*}
    w_{ji}(t+1) &= w_{ji}(t) + \Delta w_{ji}(t) \\
    \Delta w_{ji}(t) &= -\eta \frac{\partial E}{\partial w_{ji}}_{w_{ji} \rightarrow w_{ji}(t)} + \alpha \Delta w_{ji}(t-1) \\
    w_{kj}(t+1) &= w_{kj}(t) + \Delta w_{kj}(t) \\
    \Delta w_{kj}(t) &= -\eta \frac{\partial E}{\partial w_{kj}}_{w_{kj} \rightarrow w_{kj}(t)} + \alpha \Delta w_{kj}(t-1)
\end{align*}
\]

where \(0 < \alpha, \eta < 1\) with typical values of \(\eta = 0.5\) to 0.6, \(\alpha = 0.8\) to 0.9

• Use sigmoid function which satisfies \(\phi(-z) = -\phi(z)\) helps speed up convergence

\[
\phi(z) = a \left( \frac{1 - e^{-bz}}{1 + e^{-bz}} \right)
\]

\(a = 1.716, \ b = \frac{2}{3} \Rightarrow \frac{\partial \phi}{\partial z} \bigg|_{z=0} \approx 1\)

and \(\phi(z)\) is linear in the range \(-1 < z < 1\)
Some Useful Tricks

- Randomize the order of presentation of training examples from one pass to the next helps avoid local minima.
- Introduce small amounts of noise in the weight updates (or into examples) during training helps improve generalization – minimizes over fitting, makes the learned approximation more robust to noise, and helps avoid local minima.
- If using the suggested sigmoid nodes in the output layer, set target output for output nodes to be 1 for target class and -1 for all others.

Some useful tricks

- **Regularization** helps avoid over fitting and improves generalization.

\[
R(W) = \lambda E(W) + (1 - \lambda) C(W); \quad 0 \leq \lambda \leq 1
\]

\[
C(W) = \frac{1}{2} \left( \sum_{ji} w_{ji}^2 + \sum_{kj} w_{kj}^2 \right)
\]

\[
- \frac{\partial C}{\partial w_{ji}} = -w_{ji} \quad \text{and} \quad - \frac{\partial C}{\partial w_{kj}} = -w_{kj}
\]

Start with $\lambda$ close to 1 and gradually lower it during training. When $\lambda < 1$, it tends to drive weights toward zero setting up a tension between error reduction and complexity minimization.
Some Useful Tricks

Input and output encodings

- **Do not eliminate natural proximity** in the input or output space
  - Do not normalize input patterns to be of unit length if the length is likely to be relevant for distinguishing between classes
- **Do not introduce unwarranted proximity** as an artifact
  - Do not use log₂ M outputs to encode M classes, use M outputs instead to avoid spurious proximity in the output space
- Use error correcting codes when feasible

Examples of a good code

- Binary thermometer codes for encoding real values
  - Suppose we can use 10 bits to represent a value between -1.0 and +1.0
  - We can quantize the interval [-1, 1] into 10 equal parts
  - 0.38 in thermometer code is 1111000000
  - 0.60 in thermometer code is 1111110000
  - Note values that are close along the real number line have thermometer codes that are close in Hamming distance

Example of a bad code

- Ordinary binary representations of integers
Some Useful Tricks

- Normalizing inputs – know when and when not to normalize
- Scale each component of the input separately to lie between -1 and 1 with mean of 0 and standard deviation of 1

\[
\mu_i = \frac{1}{P} \sum_{q=1}^{P} x_{iq}
\]

\[
\sigma_i^2 = \frac{1}{P} \sum_{q=1}^{P} x_{iq}^2 - \mu_i^2
\]

\[
x_{ip} = \left( x_{ip} - \mu_i \right) \frac{\mu_i}{\sigma_i}
\]

Initializing weights (revisited)

Suppose weights are uniformly distributed between \(-w\) and \(+w\)

Standardized input to a hidden neuron is distributed between \(-w\sqrt{N}\) and \(+w\sqrt{N}\)

We want this to fall between -1 and +1 \(\Rightarrow w = \frac{1}{\sqrt{N}}\)

\[
-\frac{1}{\sqrt{N}} < w_{ji} < \frac{1}{\sqrt{N}}
\]

\[
-\frac{1}{\sqrt{n_H}} < w_{kj} < \frac{1}{\sqrt{n_H}}
\]
Some Useful Tricks

- Normalizing inputs – know when and when not to normalize
- Normalizing each input pattern so that it is of unit length is commonplace, but often inappropriate

\[ X_p \leftarrow \frac{X_p}{\|X_p\|} \]

Some Useful Tricks

- **Use of problem specific information** (if known) speeds up convergence and improves generalization
- In networks designed for translation-invariant visual image classification, building in translation invariance as a constraint on the weights helps
- If we know the function to be approximated is smooth, we can build that in as part of the criterion to be minimized – minimize in addition to the error, the gradient of the error with respect to the inputs
Some Useful Tricks

- **Manufacture training data** – training networks with translated and rotated patterns if translation and rotation invariant recognition is desired
- **Incorporate hints** during training
- Hints are used as additional outputs during training to help shape the hidden layer representation

![Hint nodes (e.g., vowels versus consonants in training a phoneme recognizer)](image)

Some Useful Tricks

- Reducing the effective number of free parameters (degrees of freedom) helps improve generalization
- Regularization
- Preprocess the data to reduce the dimensionality of the input –
  - Train a neural network with output same as input, but with fewer hidden neurons than the number of inputs
  - Use the hidden layer outputs as inputs to a second network to do function approximation
Some Useful Tricks

- Choice of appropriate error function is critical – do not blindly minimize sum squared error – there are many cases where other criteria are appropriate
- Example

\[ E_S(W) = \sum_{p=1}^{P} \sum_{k=1}^{M} t_{kp} \ln \left( \frac{t_{kp}}{z_{kp}} \right) \]

is appropriate for minimizing the distance between the target probability distribution over the \( M \) output variables and the probability distribution represented by the network.

- Interpreting the outputs as class conditional probabilities
- Use exponential but not sigmoid output nodes
Bayes classification and Neural Networks

\[ P(\omega_k | X) = \frac{P(X | \omega_k)P(\omega_k)}{\sum_{l=1}^{M} P(X | \omega_l)P(\omega_l)} \]

\[ t_k(X_p) = \begin{cases} t_{kp} = 1 & \text{if } X_p \in \omega_k \\ t_{kp} = 0 & \text{if } X_p \notin \omega_k \end{cases} \]

\[ g_k(X_p; W) = \text{kth output for input } X_p \]

\[ E_s(W) = \sum_{p=1}^{P} \left( g_k(X_p; W) - t_{kp} \right)^2 \]

\[ = \sum_{X_p \in \omega_k} \left( g_k(X_p; W) - 1 \right)^2 + \sum_{X_p \notin \omega_k} \left( g_k(X_p; W) - 0 \right)^2 \]

Because generalized delta rule minimizes this quantity with respect to \( W \), we have

\[ g_k(X; W) \approx P(\omega_k | X) \]

Assuming that the network is expressive enough to represent

\[ P(\omega_k | X) \]
Radial-Basis Function Networks

- A function is approximated as a linear combination of radial basis functions (RBF). RBFs capture local behaviors of functions.
- RBFs represent local receptive fields

\[
x^2
\]

\[
x_m
\]

\[
y
\]

\[
w_H
\]

\[
w_I
\]

- Hidden layer applies a non-linear transformation from the input space to the hidden space.
- Output layer applies a linear transformation from the hidden space to the output space.
**φ-separability of patterns**

\[
\varphi(x) = \langle \varphi_1(x), \ldots, \varphi_H(x) \rangle \\
\varphi_i \\
\{\varphi_i(x)\}_{i=1}^{H}
\]

Hidden layer representation

A (binary) partition, also called dichotomy, \((C_1, C_2)\) of the training set \(C\) is **φ-separable** if there is a vector \(w\) of dimension \(H\) such that:

\[
\begin{align*}
W \cdot \varphi(X) &> 0 & X \in C_1 \\
W \cdot \varphi(X) &< 0 & X \in C_2
\end{align*}
\]

---

**Examples of φ-separability**

- Separating surface: \(A \cdot \varphi(X) = 0\)
- Examples of separable partitions \((C_1, C_2)\):

  - **Linearly separable:**
  - **Quadratically separable:**
    - Spherically separable:
Example of a radial basis function

- Hidden units: use a radial basis function

\[ \varphi_\sigma(||X - W||^2) \]

the output depends on the distance of the input \( x \) from the center \( t \)

\[ \varphi_\sigma(||X - W||^2) \]

\( W \) is called center
\( \sigma \) is called spread
center and spread are parameters

Radial basis function

- A hidden neuron is more sensitive to data points near its center. This sensitivity may be tuned by adjusting the spread \( \sigma \).

- Larger spread \( \Rightarrow \) less sensitivity

- Neurons in the visual cortex have locally tuned frequency responses.
Gaussian Radial Basis Function $\phi$

$\phi :$

$\sigma$ is a measure of how spread the curve is:

- Large $\sigma$
- Small $\sigma$

Types of $\phi$

- Multiquadrics
  \[ \varphi(r) = \left( r^2 + c^2 \right)^{1/2} \quad c > 0 \]

- Inverse multiquadrics
  \[ \varphi(r) = \frac{1}{\left( r^2 + c^2 \right)^{1/2}} \quad r = \| X - W \| \]

- Gaussian functions:
  \[ \varphi(r) = \exp \left( -\frac{r^2}{2\sigma^2} \right) \quad \sigma > 0 \]
Implementing Exclusive OR using an RBF network

- Input space:
  - Input space:
  - Output space:
  - Construct an RBF pattern classifier such that:
    - (0,0) and (1,1) are mapped to 0, class C1
    - (1,0) and (0,1) are mapped to 1, class C2

 exclusive OR revisited

In the feature (hidden) space:

\[
\begin{align*}
\phi_1(x_1, x_2) &= e^{-||x-W_1||^2} = z_1 \\
\phi_2(x_1, x_2) &= e^{-||x-W_2||^2} = z_2
\end{align*}
\]

\[
W_1 = [1, 1]^T \\
W_2 = [0, 0]^T
\]

When mapped into the feature space \(<z_1, z_2>\), C1 and C2 become linearly separable. So a linear classifier with \(\phi_1(x)\) and \(\phi_2(x)\) as inputs can be used to solve the XOR problem.
RBF Learning Algorithm

$$\Delta \sigma_j = -\eta_j \frac{\partial E_S}{\partial \sigma_j}$$

$$\Delta \alpha_j = -\eta_j \frac{\partial E_S}{\partial \alpha_j}$$

$$\Delta w_{ji} = -\eta_{ji} \frac{\partial E_S}{\partial w_{ji}}$$

Depending on the specific function can be computed using the chain rule of calculus.

RBF Learning Algorithm

$$z_{jp} = e^{-\frac{(x_p - w_j)^2}{2\sigma_j^2}}$$

$$y_p = \sum_{j=0}^{L} \alpha_j z_{jp}$$

$$E_p = \frac{1}{2} (t_p - y_p)^2$$

$$x_p = [x_{1p}, \ldots, x_{xp}]^T$$

$$w_j = [w_{1j}, \ldots, w_{jn}]^T$$
RBF Learning Algorithm

\[
\Delta \alpha_j = -\eta_j \frac{\partial E_p}{\partial \alpha_j} = \eta_j \left( t_p - y_p \right) z_{jp}
\]

\[
\alpha_j' = \alpha_j + \eta_j \left( t_p - y_p \right) z_{jp}
\]

\[
\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial y_p} \frac{\partial y_p}{\partial z_{jp}} \frac{\partial z_{jp}}{\partial w_{ji}}
\]

\[
= - \left( t_p - y_p \right) \alpha_j \left( \frac{z_{jp}}{\sigma^2_j} \right) \left( x_{jp} - w_{ji} \right)
\]

\[
w_{ji} = w_{ji}' + \eta_j \left( t_p - y_p \right) \alpha_j \left( \frac{z_{jp}}{\sigma^2_j} \right) \left( x_{jp} - w_{ji} \right)
\]

---

RBF Learning Algorithm

\[
\frac{\partial E_p}{\partial \sigma_j} = \frac{\partial E_p}{\partial y_p} \frac{\partial y_p}{\partial z_{jp}} \frac{\partial z_{jp}}{\partial \sigma_j}
\]

\[
= - \left( t_p - y_p \right) \alpha_j \left( -z_{jp} \right) \left( \frac{2}{\sigma_j} \right) \left( \ln z_{jp} \right)
\]

\[
\sigma_j' = \sigma_j - \eta_j \left( t_p - y_p \right) \alpha_j \left( z_{jp} \right) \left( \frac{2}{\sigma_j} \right) \left( \ln z_{jp} \right)
\]
RBF Learning Algorithm (continued)

Some useful facts

\[ \|V\|_2^2 = V^T V \quad \text{(norm)} \]
\[ \|V\|_C^2 = (CV)^T (CV) = V^T C^T CV \quad \text{(weighted norm)} \]
\[ \|V\|_C^2 = \|V\|_2^2 \quad \text{if} \ C^T C = \text{identity matrix} \]

\[ \frac{d}{dX} (AX) = A \]

\[ \frac{d}{dX} (X^T AX) = 2AX \quad \text{(when A is a symmetric matrix)} \]

\[ \frac{d}{dA} (X^T AX) = X^T X \]

RBF Learning Algorithm

\[ z_{jp} = e^{-\frac{1}{2}(x_p-w_j)^T \varepsilon_j (x_p-w_j)} \]
\[ y_p = \sum_{j=0}^{L} \alpha_j z_{jp} \]
\[ E_p = \frac{1}{2}(t_p - y_p)^2 \]
\[ X_p = [x_{1p}, \ldots, x_{Np}]^T \]
\[ W_j = [w_{j1}, \ldots, w_{jN}]^T \]

Exercise: Derive the weight update equations from first principles
RBF Learning Algorithm (continued)

More general form of radial basis function

\[ C_j^{-1} \] is the inverse of an \( N \times N \) covariance matrix

Note that the covariance matrix is symmetric

\[
  z_{jp} = e^{\left(-\left(\mathbf{w}_j - \mathbf{x}_p\right)^T C_j^{-1} \left(\mathbf{w}_j - \mathbf{x}_p\right)\right)}
\]

Exercise: derive a learning rule for an RBF network with such neurons in the hidden layer and linear neurons in the output layer
RBF Learning Algorithm (continued)

Some useful facts

\[ \|V\|_2^2 = V^TV \text{ (norm)} \]
\[ \|V\|_C^2 = (CV)^T(CV) = V^TC^TV \text{ (weighted norm)} \]
\[ \|V\|_C^2 = \|V\|_2^2 \text{ if } C^TC \text{ is identity matrix} \]

\[ \frac{d}{dX}(AX) = A \]
\[ \frac{d}{dX}(X^TAX) = 2AX \text{ (when } A \text{ is a symmetric matrix)} \]
\[ \frac{d}{dA}(X^TAX) = X^TX \]

RBF Learning Algorithm

- Initialize the parameters -- centers of the hidden neurons are typically initialized to coincide with a subset of the training set
- Use gradient descent to adjust the parameters using the training data until the desired performance criterion is satisfied
Instance Based Learning

- Read: Chapter 8, Machine Learning, Mitchell
- Outline
- Motivation
- K nearest neighbor classifiers
- Locally weighted k nearest neighbor classifiers
- Locally weighted regression
- Lazy versus eager learning
- Prototype learning
- Representative Applications

Instance Based Learning

- Nearest neighbor methods
- Are conceptually simple
- Asymptotically have error rates that are no worse than twice that of the optimum Bayes classifier
- Learn by simply memorizing training examples
- Construct a different approximation on the fly for each input instance (query instance) unlike the other learning algorithms we have considered so far which construct a single approximation to the target function during the learning phase and use it thereafter for generating the output for each query instance.
Instance Based Learning

- Nearest neighbor methods
- The computational effort of learning is low
- The storage requirements of learning is high – need to memorize the examples in the training set
- Cost of classifying new instances can be high
- A distance measure needs to be defined over the input space – e.g., Euclidean distance, Hamming distance, etc. as appropriate
- Performance degrades when there are many irrelevant attributes

K nearest neighbor method

- Instances are assumed to lie in an \( n \)-dimensional instance space – e.g., the Euclidean space
- An instance \( X \) is described by a feature vector

\[
X_p = [x_{1p}, \ldots, x_{Np}]
\]

Where \( x_{ip} \) denotes the value of the \( i \)th feature in \( X_p \)

\[
d(X_p, X_r) = \left( \sum_{i=1}^{N} (x_{ip} - x_{ir})^2 \right)
\]

Defines the Euclidean distance between two patterns in the Euclidean space – other distance measures can be used as needed
K nearest neighbor Classifier

Learning Phase
• For each training example \((X_i, f(X_i))\), store the example in memory

Classification phase
• Given a query instance \(X_q\), identify the \(k\) nearest neighbors \(X_1, \ldots, X_k\) of \(X_q\)

\[
g(X_q) = \arg\max_{X \in \Omega} \sum_{i=1}^{K} \delta(\alpha_i f(X_i))
\]

where

\[
\delta(a, b) = 1 \text{ iff } a = b \text{ and } \delta(a, b) = 1.
\]

K nearest neighbor Function Approximator

Learning Phase
• For each training example \((X_i, f(X_i))\), store the example in memory

Prediction phase
• Given a query instance \(X_q\), identify the \(k\) nearest neighbors \(X_1, \ldots, X_k\) of \(X_q\)

\[
g(X_q) \leftarrow \frac{\sum_{i=1}^{K} f(X_i)}{K}
\]
Distance weighted K nearest neighbor Classifier

Learning Phase
• For each training example \((X_i, f(X_i))\), store the example in memory

Classification phase
• Given a query instance \(X_q\), identify the \(k\) nearest neighbors of \(X_q\) - KNN \((X_q) = \{X_1...X_k\}\)

\[
g(X_q) = \arg\max_{i \in \Omega} \sum_{l=1}^{k} w_l \delta(\omega, f(X_i))
\]

where

\[
w_l = \frac{1}{d(X_l, X_q)^2}
\]

whenever \(X_q \not\in KNN(X_q)\)
and \(g(X_q) = f(X_q)\)

Prediction phase
• Given a query instance \(X_q\), identify the \(k\) nearest neighbors of \(X_q\) - KNN \((X_q) = \{X_1...X_k\}\)

\[
g(X_q) = \sum_{l=1}^{k} w_l f(X_i)
\]

where

\[
w_l = \frac{1}{d(X_l, X_q)^2}
\]

whenever \(X_q \not\in KNN(X_q)\)
and \(g(X_q) = f(X_q)\)
Locally weighted regression

- Locally weighted regression involves calculating an approximation of the function value for a given input based on its nearest neighbors when needed during the approximation phase as opposed to during the learning phase.

- Let the approximation be of the form

\[ g(X) = w_0 + \sum_{i=1}^{N} w_i x_i \]

in a small neighborhood around a query \( X_q \)
Locally weighted regression

\[ g(X) = w_0 + \sum_{i=1}^{N} w_i x_i \]

Minimize the error over the K nearest neighbors of \( X_q \)

\[ E(X_q) = \frac{1}{2} \sum_{X \in \text{KNN}(X_q)} (f(X) - g(X))^2 \]

\[ w_i \leftarrow w_i - \eta \frac{\partial E(X_q)}{\partial w_i} \]

\[ w_i \leftarrow w_i + \eta \sum_{X \in \text{KNN}(X_q)} (f(X) - g(X)) x_i \]

Locally weighted regression

Minimize the error over all the neighbors of \( X_q \) in the training set weighted by an inverse function of distance to the neighbors

\[ E_2(X_q) = \frac{1}{2} \sum_{X(X,i),j(X) \in D} (f(X) - g(X))^2 \phi(d(X_q, X)) \]

\[ w_i \leftarrow w_i - \eta \frac{\partial E_2(X_q)}{\partial w_i} \]

\[ w_i \leftarrow w_i + \eta \sum_{X(X,i),j(X) \in D} \phi(d(X_q, X))(f(X) - g(X)) x_i \]
Locally weighted regression

Minimize the error over the $K$ nearest neighbors of $X_q$ in the training set weighted by an inverse function of distance to the neighbors:

$$E_q(X_q) = \frac{1}{2} \sum_{X \in \text{KNN}(X_q)} (f(X) - g(X))^2 \phi(d(X_q, X))$$

$$w_i \leftarrow w_i - \eta \frac{\partial E_q(X_q)}{\partial w_i}$$

$$w_i \leftarrow w_i + \eta \sum_{X \in \text{KNN}(X_q)} \phi(d(X_q, X))(f(X) - g(X)) x_i$$