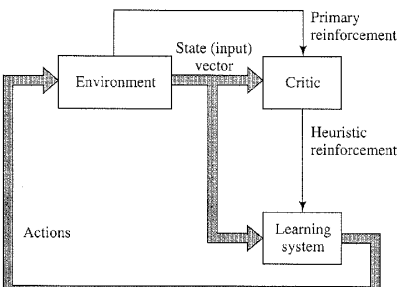
2.9 Learning without a Teacher

- No labeled training examples are now available.
- Two subclasses of learning without a teacher.

1. Reinforcement learning and Neurodynamic programming

- Block diagram of a reinforcement learning system.



- In reinforcement learning, a scalar index of performance is minimized.
- Here a *critic* is used.
- It converts a *primary reinforcement signal* received from the environment to a higher-quality *heuristic reinforcement signal*.

- Both of these signals are scalars.
- The system learns under *delayed reinforcement*.
- This means that the system observes a temporal sequence of state vectors.
- This eventually results in the generation of the heuristic reinforcement signal.
- The goal of the learning is to minimize a *cost-to-go-function*.
- This is defined as the expectation of the cumulative cost of *actions* taken over a sequence of steps.
- Another component of the reinforcement learning system is called the *learning machine*.
- Its task is to discover the actions determining the best overall behavior of the system.

- Delayed reinforcement learning is difficult to perform for two basic reasons:
  1. There is no teacher;
  2. A temporal credit assignment problem must be solved because of the time delay used.
- On the other hand, delayed reinforcement learning is very appealing.
- Reason: it learns to perform a task using only its own experiences.
- Reinforcement learning is closely related to *dynamic programming*.
- This was developed in optimal control theory (Bellman, 1957).
- Dynamic programming provides the mathematical formalism for sequential decision making.
- An introduction to dynamic programming and its relationship to reinforcement learning are presented in Chapter 12.

2. Unsupervised Learning

- In *unsupervised* (or *self-organized*) learning there is no external teacher or critic available for learning.
- Instead, the neural network tries to learn some meaningful internal statistical representation of the data.
- For example, a suitable linear model fitted to the input data.
- Some *task-independent measure* is used for the quality of the representation.
- The free parameters of the neural network are optimized with respect to this measure.
- After learning, the network can encode features of the input data.
- Competitive learning (winner-take-all networks) can be used for unsupervised learning.
- Hebbian learning is also often employed.

- Various forms of unsupervised learning are discussed in Chapters 8-11.

2.10 Learning Tasks

Pattern Association

- *Associative memory* is a brainlike distributed memory that learns by *association*.
- Association is an important property of the human memory.
- It is also a basic operation in all models of cognition.
- Two basic forms: *autoassociation* or *heteroassociation*.
- In **autoassociation**, a neural network is first required to *store* a set of input patterns (vectors).
- This is done by repeatedly presenting them to the network.
- After learning, a partial or noisy version of an original input pattern is shown to the network.
- The task is now to *retrieve (recall)* the original pattern.

- **Heteroassociation** differs from autoassociation in that an arbitrary set of input patterns is *paired* with another arbitrary set of output patterns.
- Autoassociation uses unsupervised learning, while heteroassociation requires supervised learning.
- The operation of an associative memory consists of two phases:
  - *storage phase* (training with input patterns);
  - *recall phase* (retrieval of a memorized pattern).
- If an associative memory cannot retrieve a stored input pattern perfectly, it is said to have made an error in recall.
- The number q of patterns stored in an associative memory measures the storage capacity of the network.
- A design goal: make the storage capacity as large as possible compared to the number of neurons in the network.
- On the other hand, a large fraction of memorized patterns should be recalled correctly. — Conflicting goals!

Pattern Recognition

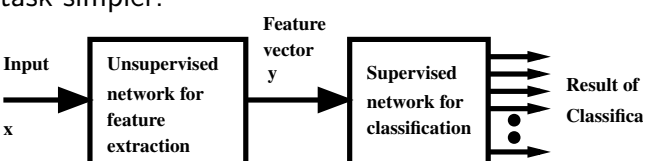
- Humans are good at pattern recognition.
- Examples: recognizing speech, a familiar face, objects in a scene.
- This ability is achieved through a learning process, or with neural networks.
- A formal definition of *pattern recognition*:
  A process whereby a received pattern or signal is assigned to one of a prescribed number of *classes (categories)*.
- In pattern recognition tasks, a neural network is first trained using input patterns having known categories.
- After training, the network should be able to classify (recognize) new patterns belonging to the learned classes.
- Formally, each pattern corresponds to a point in a multidimensional *decision space*.

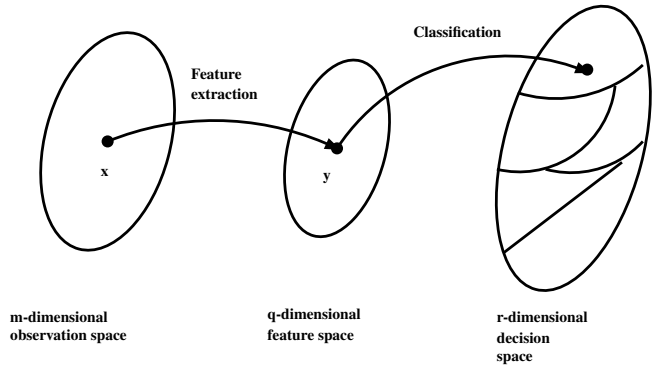
- The decision space is divided into regions corresponding to each pattern class.
- During the training process, the boundaries between the class regions are determined.
- Learning is statistical because the input patterns have inherent variability (coming from some statistical distributions).

- Neural pattern classifiers can have two general forms:

  1. The overall neural recognition system is divided into:
     - An unsupervised network for *feature extraction*.
     - A supervised network for subsequent *classification*.

     - In feature extraction, the dimensionality of input patterns is reduced.
     - This data compression makes the subsequent classification task simpler.





2. Single multilayer feedforward network.
   - Trained using supervised learning.
   - Neurons in the hidden layer(s) perform feature extraction.

Function Approximation

- Consider a nonlinear input-output mapping

$$\mathbf{d} = \mathbf{f}(\mathbf{x})$$

  Here $\mathbf{x}$ is the input and $\mathbf{d}$ the output vector.

- The vector-valued mapping function $\mathbf{f}(.)$ is assumed to be unknown.

- However, a set $\mathcal{F}$ of N labeled examples is known:

$$\mathcal{F} = \{(\mathbf{x}_1, \mathbf{d}_1), (\mathbf{x}_2, \mathbf{d}_2), \ldots, (\mathbf{x}_N, \mathbf{d}_N)\}$$

- The task is to design a neural network whose actual input-output mapping $\mathbf{F}(.)$ approximates the unknown mapping $\mathbf{f}(.)$ well enough.

- Requirement: the Euclidean norm

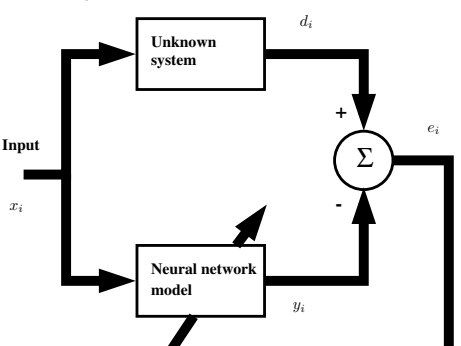
$$\| \mathbf{F}(\mathbf{x}) - \mathbf{f}(\mathbf{x}) \| < \varepsilon \text{ for all } \mathbf{x}$$

  where ε is a small positive number.

- If there are enough training samples as well as free parameters in the network, the goal can be achieved.
- This approximation problem is a perfect candidate for supervised learning.
- In fact, supervised learning may be understood as an approximation problem.
- Two important ways of exploiting the approximation ability of a neural network:

1. **System identification.**

   - Neural network learns the input-output mapping $\mathbf{f}(.)$ of an unknown system.
   - The system is a time-invariant, multiple-input multiple-output (MIMO) system.
   - The error signal $\mathbf{e}_i$ between the true output $\mathbf{y}_i$ of the network and the desired output $\mathbf{d}_i$ corresponding to the input vector $\mathbf{x}_i$ is used for training.

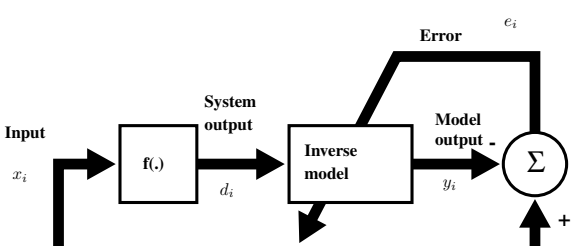


2. **Inverse system.**

- In this case, we know the MIMO system mapping $\mathbf{d} = \mathbf{f}(\mathbf{x})$.
- The task is now to learn the inverse mapping

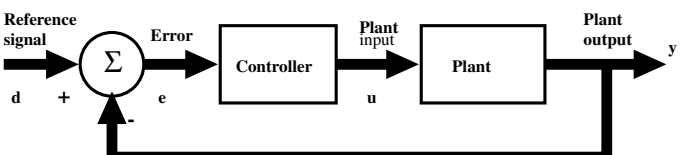
$$\mathbf{x} = \mathbf{f}^{-1}(\mathbf{d}).$$

- A straigthforward inversion of $\mathbf{f}$ is often impossible because $\mathbf{f}$ is far too complex.
- The same approach may be applied to the inverse problem as to standard system identification.
- The roles of $\mathbf{x}$ and $\mathbf{d}$ are now interchanged.



Control

- An often encountered task: control of a process or a critical part of a system.
- Neural networks may be applied instead of classical control theory.



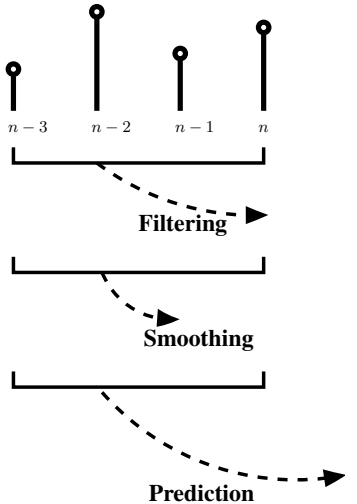
- A schematic diagram of a feedback control system.
- It is again assumed that an external reference signal $\mathbf{d}$ is available.
- The error signal $\mathbf{e}$ between $\mathbf{d}$ and the output $\mathbf{y}$ of a neural network controller is used for adjusting the free parameters of the network.
- However, now the error signal $\mathbf{e}$ propagates through the neural controller

- Therefore, for applying error-correction learning algorithm the Jacobian matrix $\mathbf{J}$ is needed.
- The elements of $\mathbf{J}$ are the partial derivatives $\partial y_k / \partial u_j$.
- Here y_k is an element of the plant output vector $\mathbf{y}$ and u_j is an element of the plant input vector $\mathbf{u}$.
- These partial derivatives are unknown.
- They can be learned either indirectly or directly.
- These methods are discussed briefly on a general level in the book.

Filtering

- In filtering, some interesting quantity is extracted from noisy observations using a suitable algorithm or device.
- Assume that we have measurement data up to a discrete time point n.
- A filter can be used for three basic tasks:
  1. *Filtering*. A quantity is estimated at time n.
  2. *Smoothing*. A quantity is estimated at time $n - d$, where d is a positive time delay.

     - Thus we have some future measurements from later than the estimation time available.
  3. *Prediction*. The estimation is performed at some future time $n+d$, $d > 0$.

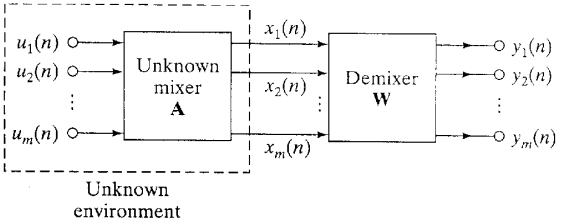


- **First example** of filtering: *cocktail party problem*.
- Separation of individual voices from their mixtures.
- More generally, this type of filtering problem arises in *blind signal separation*.

- Several efficient neural methods have been developed for blind signal separation recently.
- The data model used in blind signal separation:

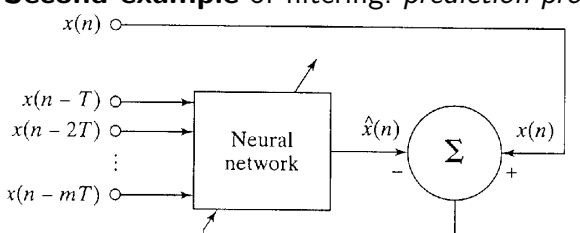
$$\mathbf{x}(n) = \mathbf{A}\mathbf{u}(n)$$

  Here $\mathbf{u}(n)$ is a vector consisting of m unknown but mutually statistically independent source signals at time n.
- $\mathbf{A}$ is an unknown nonsingular $m \times m$ mixing matrix.
- The task is to recover the source signals from the observed mixtures $\mathbf{x}(n)$ in an unsupervised manner.
- This can be done by using the strong but often plausible independence assumption.



- A blind separation system

- **Second example** of filtering: *prediction problem*.



- Now the task is to predict the present value $x(n)$ of a scalar process (time series).

- It is assumed that m previous values $x(n-T)$, $x(n-2T), \ldots, x(n-mT)$ of the time series are known.
- Here m is the order of prediction.
- This problem may be solved using error-correction learning in an unsupervised manner.
- $x(n)$ serves the purpose of desired response.
- Neural network provides a method for predicting *nonlinear* processes.

Beamforming

- Still one application of neural networks.
- This is needed in processing radar and sonar signal in direction of arrival estimation.
- Not so important application.

2.12 Adaptation

- When a neural network operates in a stationary environment, it can first learn the weight values (parameters).
- After learning, the weight values are frozen and the network is applied without further learning to new input data.
- However, in practice the environment (data) is often more or less *nonstationary*
- This means that the statistical properties of the data change with time.
- This requires that the neural network should track the statistical variations in the data by adapting its parameters.
- The theory of linear adaptation (linear adaptive filters) is well understood.
- However, in neural networks nonlinear filters are typically used.

- In practice, a neural network can be adapted by *retraining* it at suitable intervals.
- On short intervals, the data is usually roughly stationary even though it is generally nonstationary.
- Another adaptation method: continual training with time-ordered examples.
- In this case, the neural network becomes a nonlinear adaptive filter.
- In practice, successful adaptation requires that the time changes in the data are slow enough.

3. Single-Layer Perceptrons

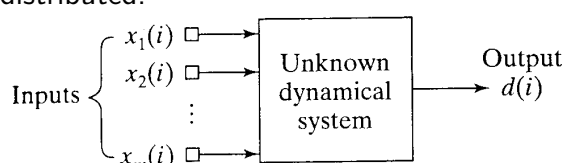
3.1 Introduction

- *Perceptron* is the first neural network model proposed for supervised learning (Rosenblatt, 1958).
- Simplest form of a neural network used for classification.
- Consists basically of a single neuron with adjustable synaptic weights and bias.
- Perceptron can be used for classifying *linearly separable* patterns belonging to two classes.
- The learning algorithm converges to a separating hyperplane in this case.
- Perceptron is closely related to the *least-mean-square (LMS)* algorithm or *Widrow-Hoff delta rule* (1960).

- The LMS algorithm is applied widely to *linear adaptive filtering* in signal processing.
- In this chapter, we first briefly discuss adaptive filtering and the LMS algorithm in sections 3.2-3.7.
- Rosenblatt's perceptron is then discussed at the end of the chapter in sections 3.8-3.10.

3.2 Adaptive filtering problem

- Consider an unknown dynamical system.
- We know the scalar outputs $d(i)$ (responses) corresponding to the input vectors $\mathbf{x}(i)$.
- The input vectors $\mathbf{x}(i)$ are assumed to be m-dimensional and identically distributed.



- The input vectors $\mathbf{x}(i)$ can arise either spatially or temporally:

  1. The m elements of $\mathbf{x}(i)$ originate at different points in space.
     - $\mathbf{x}(i)$ is then called a *snapshot* of data.
     - Example: an uniformly spaced line array of sensors.

2. The elements of $\mathbf{x}(i)$ are the m last values of a scalar time series (signal).
   - Samples are uniformly spaced in time.

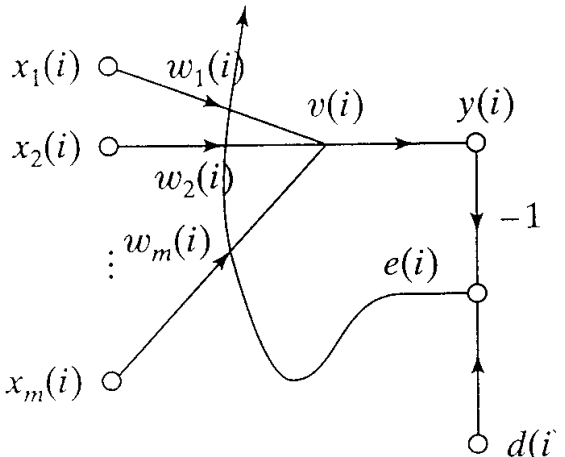
- The described multiple-input single-output system is modeled using a single linear neuron.

- The learning algorithm modifying the synaptic weights has the following properties:

  1. Starts from *arbitrary* initial values of the synaptic weights.
  2. The weights are updated (adjusted) continuously.
  3. Each update of synaptic weights is made during one single sampling interval.

- This kind of neuronal model is called an *adaptive filter*.

- The operation of the adaptive filter consist of two stages



1. *Filtering*. Here the output $y(i)$ and the error signal $e(i)$ are computed.
2. *Adaptation* of the weights using the error signal $e(i)$.

- These two processes form a *feedback loop* around the neuron.

- Since the neuron is linear, its output

$$y(i) = \mathbf{x}^T(i)\mathbf{w}(i) = \sum_{k=1}^{m} w_k(i)x_k(i)$$

where the weight vector

$$\mathbf{w}(i) = [w_1(i), w_2(i), \ldots, w_m(i)]^T$$

and the input vector

$$\mathbf{x}(i) = [x_1(i), x_2(i), \ldots, x_m(i)]^T.$$

The error signal

$$e(i) = d(i) - y(i)$$

is used to adapt the filter by minimizing some appropriate cost function.

- Before deriving the adaptive algorithm, some unconstrained optimization methods are reviewed in the next section.

- They are generally applicable to suitable cost functions used in neural networks and other areas.

3.3 Unconstrained optimization techniques

- Assume that $\mathcal{E}(\mathbf{w})$ is a *continuosly differentiable* cost function of the weight (parameter) vector $\mathbf{w}$.
- *Unconstrained optimization problem:* minimize the cost function $\mathcal{E}(\mathbf{w})$ with respect to $\mathbf{w}$.
- An optimal solution $\mathbf{w}^*$ satisfies the condition $\mathcal{E}(\mathbf{w}^*) \leq \mathcal{E}(\mathbf{w})$.
- The necessary condition for optimality is

$$\nabla \mathcal{E}(\mathbf{w}^*) = \mathbf{0}.$$

- Here ∇ is the *gradient operator*

$$\nabla = \left[\frac{\partial}{\partial w_1}, \frac{\partial}{\partial w_2}, \ldots, \frac{\partial}{\partial w_m} \right]^T$$

and $\nabla \mathcal{E}(\mathbf{w})$ is the *gradient vector* of the cost function:

$$\nabla \mathcal{E}(\mathbf{w}) = \left[\frac{\partial \mathcal{E}}{\partial w_1}, \frac{\partial \mathcal{E}}{\partial w_2}, \ldots, \frac{\partial \mathcal{E}}{\partial w_m} \right]^T.$$

- *Iterative descent* type algorithms are well suited for learning the weights of an adaptive filter or a neural network.

- In these algorithms, one tries to reduce the value of the cost function at each iteration:
$$\mathcal{E}(\mathbf{w}(n+1)) < \mathcal{E}(\mathbf{w}(n))$$

- The iteration starts from some initial guess $\mathbf{w}(0)$.

- It is hoped that the local iterative algorithm will converge to the optimal solution $\mathbf{w}^*$.

- This may not always happen because the algorithm:
  - gets stuck in a local minimum.
  - becomes unstable because of too large corrections.

- In the following, three general unconstrained optimization methods relying on iterative descent are described.

Method of Steepest Descent

- For brevity, denote the gradient vector by $\mathbf{g} = \nabla \mathcal{E}(\mathbf{w})$.
- The negative gradient vector $-\mathbf{g}(n)$ shows the direction of steepest descent of the cost function $\mathcal{E}(\mathbf{w})$ at point $\mathbf{w}(n)$.
- This leads directly to the steepest descent algorithm

$$\mathbf{w}(n+1) = \mathbf{w}(n) - \eta \mathbf{g}(n)$$

  where η is a positive constant called stepsize or learning-rate parameter.

- The correction (update) at step n is thus

$$\Delta \mathbf{w}(n) = \mathbf{w}(n+1) - \mathbf{w}(n) = -\eta \mathbf{g}(n).$$

- The new value $\mathcal{E}(\mathbf{w}(n+1))$ of the cost function can be approximated by using a first-order Taylor series expansion around $\mathbf{w}(n)$:

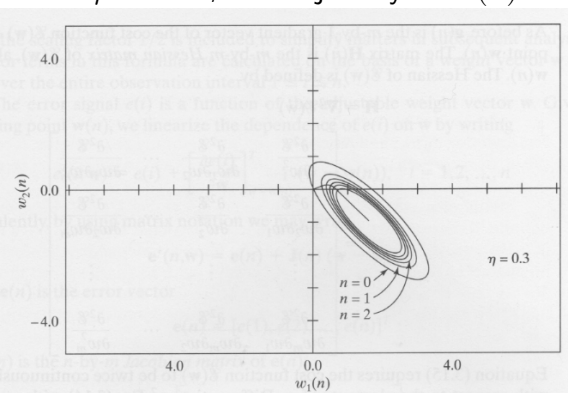
$$\mathcal{E}(\mathbf{w}(n+1)) \approx \mathcal{E}(\mathbf{w}(n)) + \mathbf{g}^T(n) \Delta \mathbf{w}(n)$$

- This approximation is justified for small η.
- Inserting $\Delta\mathbf{w}(n)$ in this formula yields

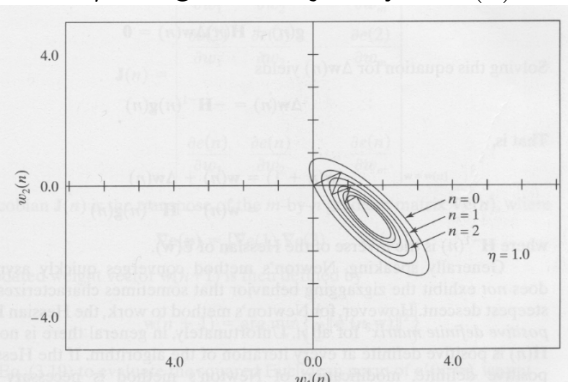
$$\mathcal{E}(\mathbf{w}(n+1)) \approx \mathcal{E}(\mathbf{w}(n)) - \eta \parallel \mathbf{g}(n) \parallel^2 .$$

- This shows that the value of the cost function decreases at each iteration for small learning rates η.
- The method of steepest descent converges to the optimal solution $\mathbf{w}^*$ slowly.

- The learning parameter η has a profound effect on the convergence behavior:
  - When η is small, the trajectory of $\mathbf{w}(n)$ follows a smooth path



– When η is large, the trajectory of $\mathbf{w}(n)$ oscillates



– If η exceeds a critical value, the steepest descent algorithm becomes unstable (diverges).

Newton's Method

- *Basic idea of Newton's method*: minimize the quadratic approximation of the cost function $\mathcal{E}(\mathbf{w}(n))$ around the current point $\mathbf{w}(n)$.
- This is done at each iteration using a second-order Taylor series expansion of the cost function.
- Recall first the Taylor series expansion for a function $f(x)$ of a single scalar variable x:

$$f(x + \Delta x) = f(x) + \frac{df(x)}{dx}\Delta x + \frac{1}{2}\frac{d^2 f(x)}{dx^2}(\Delta x)^2 + \cdots$$

- This can be generalized for a scalar function $\mathcal{E}(\mathbf{w})$ of several variables (components of the vector $\mathbf{w}$) as follows:

$$\mathcal{E}(\mathbf{w} + \Delta\mathbf{w}) = \mathcal{E}(\mathbf{w}) + \mathbf{g}^T\Delta\mathbf{w} + \frac{1}{2}(\Delta\mathbf{w})^T\mathbf{H}\Delta\mathbf{w} + \cdots$$

- Here $\mathbf{g} = \nabla \mathcal{E}(\mathbf{w})$ is the gradient vector of $\mathcal{E}(\mathbf{w})$, and $\mathbf{H}$ its Hessian matrix $\nabla^2 \mathcal{E}(\mathbf{w})$:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 \mathcal{E}}{\partial w_1^2} & \frac{\partial^2 \mathcal{E}}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 \mathcal{E}}{\partial w_1 \partial w_m} \\ \frac{\partial^2 \mathcal{E}}{\partial w_2 \partial w_1} & \frac{\partial^2 \mathcal{E}}{\partial w_2^2} & \cdots & \frac{\partial^2 \mathcal{E}}{\partial w_2 \partial w_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{E}}{\partial w_m \partial w_1} & \frac{\partial^2 \mathcal{E}}{\partial w_m \partial w_2} & \cdots & \frac{\partial^2 \mathcal{E}}{\partial w_m^2} \end{bmatrix}$$

- The Hessian $\mathbf{H}$ is a symmetric $m \times m$ matrix containing the second derivatives of $\mathcal{E}(\mathbf{w})$ with respect to the components of the weight vector $\mathbf{w}$.

- Requirement for using the Hessian: the cost function $\mathcal{E}(\mathbf{w})$ must be twice continuously differentiable with respect to the vector $\mathbf{w}$.

- The second-order approximation of the Taylor series yields

$$\Delta \mathcal{E}(\mathbf{w}(n)) = \mathbf{g}^T(n) \Delta \mathbf{w}(n) + \frac{1}{2} (\Delta \mathbf{w})^T(n) \mathbf{H}(n) \Delta \mathbf{w}(n).$$

- Differentiating this equation with respect to $\Delta\mathbf{w}$ leads to the condition

$$\mathbf{g}(n) + \mathbf{H}(n)\Delta\mathbf{w}(n) = \mathbf{0}.$$

  for minimizing the change $\Delta\mathcal{E}(\mathbf{w}) = \mathcal{E}(\mathbf{w} + \Delta\mathbf{w}) - \mathcal{E}(\mathbf{w})$.

- Solving this equation for $\Delta\mathbf{w}(n)$ yields for the update at step n

$$\Delta\mathbf{w}(n) = -\mathbf{H}^{-1}(n)\mathbf{g}(n).$$

  where $\mathbf{H}^{-1}(n)$ is the inverse of the Hessian of $\mathcal{E}(\mathbf{w})$.

- Thus iterations of Newton's method have the form

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \Delta\mathbf{w}(n) = \mathbf{w}(n) - \mathbf{H}^{-1}(n)\mathbf{g}(n).$$

- Newton's method converges quickly asymptotically.
- It does not exhibit a zigzagging behavior.
- However, $\mathbf{H}(n)$ must be a positive definite matrix for all n.

- This is not always true in practice; then the method must be modified.
- Other drawbacks of Newton's method:
  - Convergence may be slow in the beginning (far from optimum).
  - Requires knowledge of second derivatives.

Gauss-Newton Method

- The Gauss-Newton method is applicable to a cost function that is expressed as the sum of error squares. Let

$$\mathcal{E}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n} e^2(i)$$

- The error terms here are computed by keeping the weight vector $\mathbf{w}$ fixed over the entire observation interval $1 \leq i \leq n$.

- In this method, the dependence of the error $e(i)$ on $\mathbf{w}$ is linearized around the operating point $\mathbf{w}(n)$:

$$e'(i, \mathbf{w}) = e(i) + \left[\frac{\partial e(i)}{\partial \mathbf{w}}\right]^T (\mathbf{w} - \mathbf{w}(n)), \quad i = 1, 2, \ldots, n.$$

- The gradient is evaluated at the operating point $\mathbf{w}(n)$.

- These n equations can be written in matrix-vector form compactly as follows:

$$\mathbf{e}'(n, \mathbf{w}) = \mathbf{e}(n) + \mathbf{J}(n)(\mathbf{w} - \mathbf{w}(n))$$

- Here $\mathbf{e}(n)$ is the error vector

$$\mathbf{e}(n) = [e(1), e(2), \ldots, e(n)]^T$$

- $\mathbf{J}(n)$ is the $n \times m$ *Jacobian matrix* of $\mathbf{e}$(n):

$$\mathbf{J}(n) = \begin{bmatrix} \frac{\partial e(1)}{\partial w_1} & \frac{\partial e(1)}{\partial w_2} & \cdots & \frac{\partial e(1)}{\partial w_m} \\ \frac{\partial e(2)}{\partial w_1} & \frac{\partial e(2)}{\partial w_2} & \cdots & \frac{\partial e(2)}{\partial w_m} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e(n)}{\partial w_1} & \frac{\partial e(n)}{\partial w_2} & \cdots & \frac{\partial e(n)}{\partial w_m} \end{bmatrix}$$

- The Jacobian $\mathbf{J}(n)$ is the transpose of the $m \times n$ gradient matrix

$$\nabla \mathbf{e}(n) = [\nabla e(1), \nabla e(2), \ldots, \nabla e(n)].$$

- The updated weight vector $\mathbf{w}(n+1)$ is obtained by minimizing the squared norm

$$\frac{1}{2} \parallel \mathbf{e}'(n, \mathbf{w}) \parallel^2$$

with respect to $\mathbf{w}$.

- The squared norm is first evaluated and then differentiated with respect to $\mathbf{w}$.

- Setting the result equal to zero and solving leads to the *Gauss-Newton iteration*

$$\mathbf{w}(n+1) = \mathbf{w}(n) - [\mathbf{J}^T(n)\mathbf{J}(n)]^{-1}\mathbf{J}^T(n)\mathbf{e}(n).$$

- A more detailed derivation is given in Haykin's book.

- Advantage: second derivatives are not needed.

- Gauss-Newton method is often implemented in a slightly modified form

$$\mathbf{w}(n+1) = \mathbf{w}(n) - [\mathbf{J}^T(n)\mathbf{J}(n) + \delta\mathbf{I}]^{-1}\mathbf{J}^T(n)\mathbf{e}(n)$$

  where δ is a small positive constant and $\mathbf{I}$ the unit matrix.

- This guarantees that the inverse matrix always exists.