Neural networks

Motivation
- Humans are able to process complex tasks efficiently (perception, pattern recognition, reasoning, etc.)
- Ability to learn from examples
- Adaptability and fault tolerance

Engineering applications
- Nonlinear approximation and classification
- Learning (adaptation) from data: black-box modeling
- Very-Large-Scale Integration (VLSI) implementation
Biological neuron

- **Soma**: body of the neuron.
- **Dendrites**: receptors (inputs) of the neuron.
- **Axon**: output of neuron; connected to dendrites of other neurons via synapses.
- **Synapses**: transfer of information between neurons (electrochemical signals).

Neural networks

- Biological neural networks
  - Neuron switching time: 0.001 second
  - Number of neurons: $10^{11}$ (100 bilion)
  - Connections per neuron (synapses): $10^{14}$ (100 trillion)
  - Recognition time: $10^{-3}$ s (milliseconds)

- **parallel computation**

- Artificial neural networks
  - Weighted connections amongst units
  - Highly parallel, distributed process
  - Emphasis on tuning weights automatically
Use of neural networks

- Input is high-dimensional
- Output is multidimensional
- Mathematical form of system is unknown
- Interpretability of identified model is unimportant

Applications
- Pattern recognition
- Classification
- Prediction
- Modeling

<table>
<thead>
<tr>
<th>Biological neural network</th>
<th>Artificial neural network</th>
</tr>
</thead>
<tbody>
<tr>
<td>Soma</td>
<td>Neuron</td>
</tr>
<tr>
<td>Dendrite</td>
<td>Input</td>
</tr>
<tr>
<td>Axon</td>
<td>Output</td>
</tr>
<tr>
<td>Synapse</td>
<td>Weight</td>
</tr>
</tbody>
</table>

ANN: history

1943 Warren McCulloch & Walter Pitts

Definition of a neuron:
- The activity of a neuron is an all or none process
- The structure of the net does not change with time

Too simple structure, however:
- Proved that networks of their neurons could represent any finite logical expression
- Used a massively parallel architecture
- Provided important foundation for further development
1948 Donald Hebb

- Major contributions:
  - Recognized that information is stored in the weight of the synapses
  - Postulated a learning rate that is proportional to the product of neuron activation values
  - Postulated a cell assembly theory: repeated simultaneous activation of weakly-connected cell group results in a more strongly connected group.

1957 Frank Rosenblatt

- Defined first computer implementation: the perceptron
- Attracted attention of engineers and physicists, using model of biological vision
- Defined information storage as being “in connections or associations rather than topographic representations”
- Defined both self-organizing and supervised learning mechanisms
1959  Bernard Widrow & Marcian Hoff

- Engineers who simulated networks on computers and implemented designs in hardware (Adaline and Madaline).
- Formulated Least Mean Squares (LMS) algorithm that minimizes sum-squared error.
- LMS adapts weights even when classifier output is correct.

1977  David Rumelhart

- Introduced computer implementation of backpropagation learning and delta rule

1982  John Hopfield

- Implemented recurrent network
- Developed way to minimize “energy” of network, defined stable states
- First NNs on silicon chips built by AT&T using Hopfield net
ANN: history

1989  Cybenko (approximation theory)

1990  Jang et al. (neuro-fuzzy systems)

1993  Barron (complexity vs. accuracy)

ADAPTIVE NETWORKS
Adaptive (neural) networks

- Massively connected computational units inspired by the working of the human brain
- Provide a mathematical model for biological neural networks (brains)
- Characteristics:
  - learning from examples
  - adaptive and fault tolerant
  - robust for fulfilling complex tasks

Network classification

- Learning methods (supervised, unsupervised)
- Architectures (feedforward, recurrent)
- Output types (binary, continuous)
- Node types (uniform, hybrid)
- Implementations (software, hardware)
- Connection weights (adjustable, hard-wired)
- Inspirations (biological, psychological)
Adaptive network architecture

- Nodes are static (no dynamics) and parametric
- Network can consist of heterogeneous nodes
- Links do not have weights or parameters associated
- Node functions are differentiable except at a finite number of points

Adaptive networks categories

- Feedforward

- Recurrent
Adap. network representations

- Layered

- Topological ordering

Feedforward adaptive network

- Static mapping between input and output spaces
- Aim: construct network to obtain nonlinear mapping regulated by a data set (training data set) of desired input-output pairs of a target system to be modeled
- Procedures: learning rules or adaptation algorithms (parameter adjustment to improve network performance)
- Network performance: measured as the discrepancy between desired and network’s output for same input (error measure)
Examples of adaptive networks

- **Adaptive network with single linear node**

  \[ x_3 = f_3(x_1, x_2; a_1, a_2, a_3) = a_1 x_1 + a_2 x_2 + a_3 \]

- **Perceptron network (linear classifier)**

  \[ x_3 = f_3(x_1, x_2; a_1, a_2, a_3) = a_1 x_1 + a_2 x_2 + a_3 \]

  \[ x_4 = f_4(x_3) = \begin{cases} 
  1 & \text{if } x_3 \geq 0 \\
  0 & \text{if } x_3 < 0 
\end{cases} \]

---

Examples of adaptive networks

- **Multilayer perceptron** (3-3-2 neural network)

  \[ x_7 = \frac{1}{1 + \exp[-(w_{4,7} x_4 + w_{5,7} x_5 + w_{6,7} x_6 + t_7)]] \]

  - Parameter set of node 7: \( \{ w_{4,7}, w_{5,7}, w_{6,7}, t_7 \} \)
SUPERVISED LEARNING
NEURAL NETWORKS

Perceptron

- Early (and popular) attempt to build intelligent and self-learning systems by using simple components
- Derived from McCulloch-Pitts (1943) model of the biological neuron
- Models output by weighted combinations of selected features (feature classifier)
- Essentially a linear classifier
- Incremental learning roughly based on gradient descent
Training algorithm

- Perceptron (Rosenblatt, 1958). Can only learn **linearly separable** functions.

- **Training algorithm:**
  1. Select an input vector $\mathbf{x}$ from the training data set
  2. If the perceptron gives an incorrect response, modify all connection weights $w_j$
Training algorithm

- Weight training:
  \[ w_i(l+1) = w_i(l) + \Delta w_i(l) \]
- Weight correction is given by the delta rule:
  \[ \Delta w_i(l) = \alpha x_i(l)e(l) \]
  - \( \alpha \) - learning rate
  - \( e(l) = y_d(l) - y(l) \)

**Question:** Can we represent a simple exclusive-OR (XOR) function with a single-layer perceptron?

XOR problem

How to classify the patterns correctly?

Linear classification is not possible!
Example

- Linearly separable classifications
  - If classification is linearly separable, we can have any number of classes with a perceptron.
  - For example, consider classifying furniture according to height and width:

![Diagram of linear classification](image)

Example

- Each category can be separated from the other 2 by a straight line:
  - 3 straight lines
  - each output node fires if point is on right side of straight line:

![Diagram of classification with output nodes](image)

More than one output node could fire at same time!
Artificial neuron

- $x_i$: $i$-th input of the neuron
- $w_i$: synaptic strength (weight) for $x_i$
- $y = \sigma(\sum w_i x_i)$: output signal

Types of neurons

- Threshold $\theta$ (McCulloch and Pitts, 1943):
  $$y = \text{sign}\left(\sum_{i=1}^{n} w_i x_i - \theta\right)$$
- Other types of activation functions ($net = \sum w_i x_i$):
  - $y_{\text{step}} = \begin{cases} 1, & \text{if } net \geq 0 \\ 0, & \text{if } net < 0 \end{cases}$
  - $y_{\text{sigmoid}} = \frac{1}{1 + e^{-net}}$
  - $y_{\text{linear}} = net$
**Activation functions**

- **Logistic**
  \[ f(x) = \frac{1}{1 + e^{-x}} \]

- **Hyperbolic tangent**
  \[ f(x) = \tanh\left(\frac{x}{2}\right) = \frac{1 - e^{-x}}{1 + e^{-x}} \]

- **Identity (linear)**
  \[ f(x) = x \]

*sigmoidal or squashing functions

---

**Single-layer perceptron (SLP)**

- Single-layer perceptron can only classify linearly separable patterns, regardless of the activation function used.
- How to cope with problems which are not linearly separable?

**Using multilayer neural networks!**
Multi-Layer Perceptron for XOR

\[ w_1 \theta = -w_0 \]

Backpropagation MLP

- Most commonly used NN structures for applications in wide range of areas:
  - Pattern recognition, signal processing, data compression and automatic control
- Well-known applications:
  - NETtalk: trained an MLP to pronounce English text;
  - Carnegie Mellon University’s ALVINN (Autonomous Land Vehicle in a Neural Network) used an NN for steering an autonomous vehicle;
  - Optical Character Recognition (OCR).
Multi-Layer Perceptron

- Can learn functions that are not linearly separable.

Most common MLP
Most common MLP

- Output of neurons in the hidden-layer $h_j$:

$$h_j = \sigma\left(\sum_{i=1}^{n} w_{ij}^h x_i + b_j^h\right) = \sigma\left(\sum_{i=0}^{n} w_{ij}^h x_i\right)$$

$$= \tanh\left(\sum_{i=0}^{n} w_{ij}^h x_i\right) \quad \sigma \Rightarrow \text{sigmoid}$$

- Output of neurons in the output-layer $y_k$:

$$y_k = \sigma\left(\sum_{j=1}^{m} w_{jk}^o h_j + b_j^o\right) = \sigma\left(\sum_{j=0}^{m} w_{jk}^o h_k\right)$$

$$= \sum_{j=0}^{m} w_{jk}^o h_j \quad \sigma \Rightarrow \text{linear}$$

Learning in NN

- Biological neural networks:
  - Synaptic connections amongst neurons which simultaneously exhibit high activity are strengthen.

- Artificial neural networks:
  - Mathematical approximation of biological learning.
  - Error minimization (nonlinear optimization problem).
    - Error backpropagation (first-order gradient)
    - Newton methods (second-order gradient)
    - Levenberg-Marquardt (second-order gradient)
    - Conjugate gradients
    - ...
Supervised learning

Training data: \( X = \begin{bmatrix} x_1^T & x_2^T & \cdots & x_N^T \end{bmatrix}^T \)
\( Y = \begin{bmatrix} y_1^T & y_2^T & \cdots & y_N^T \end{bmatrix}^T \)

Error backpropagation

- Initialize all weights and thresholds to small random numbers

**Repeat**
1. Input training examples and compute network and hidden layer outputs
2. Adjust output weights using output error
3. Propagating output error backwards, adjust hidden-layer weights

**Until** satisfied with approximation
Backpropagation in MLP

- Compute the output of the output-layer, and compute error:
  \[ e_k = y_{d,k} - y_k, \quad k = 1, \ldots, l \]

- The cost function to be minimized is the following:
  \[ J(w) = \frac{1}{2} \sum_{k=1}^{l} \sum_{q=1}^{N} e_{kq}^2 \]

- \( N \) – number of data points

Learning using gradient

- Output weight learning for output \( y_k \):
  \[ w_{jk}^o (p + 1) = w_{jk}^o (p) - \alpha \nabla J (w_{jk}^o) \]
  \[ \nabla J (w_{jk}^o) = \left( \frac{\partial J}{\partial w_{1k}^o}, \frac{\partial J}{\partial w_{2k}^o}, \ldots, \frac{\partial J}{\partial w_{mk}^o} \right)^T \]
Output-layer weights

\[ y_k = \sum_{j=0}^{m} w_{jk}^o h_j, \quad e_k = y_{d,k} - y_k, \quad J(w_{jk}^o, w_{0j}^h) = \frac{1}{2} \sum_{k=1}^{f} e_k^2 \]

Applying the chain rule with then:

\[ \frac{\partial J}{\partial w_{jk}^o} = \frac{\partial J}{\partial e_k} \frac{\partial e_k}{\partial y_k} \frac{\partial y_k}{\partial w_{jk}^o} \]

with \( \frac{\partial J}{\partial e_k} = e_k, \quad \frac{\partial e_k}{\partial y_k} = -1, \quad \frac{\partial y_k}{\partial w_{jk}^o} = h_j \)

then \( \frac{\partial J}{\partial w_{jk}^o} = -h_j e_k \)

Thus: \( w_{jk}^o (p + 1) = w_{jk}^o (p) - \alpha \nabla J (w_{jk}^o) = w_{jk}^o (p) + \alpha h_j e_k \)

Recall that for SLP: \( \Delta w_j = \alpha x_j e \)
Hidden-layer weights

Partial derivatives:

\[ \frac{\partial J}{\partial h_j} = \sum_{k=1}^{k} -e_k w^o_{jk}, \quad \frac{\partial h_j}{\partial \text{net}_j} = \sigma'_j(h_j), \quad \frac{\partial \text{net}_j}{\partial w^h_{ij}} = x_i \]

then

\[ \frac{\partial J}{\partial w^h_{ij}} = -x_i \sigma'_j(h_j) \sum_{k=1}^{l} (-e_k w^o_{jk}) \]

and \( \Delta w^h_{ij}(p) = \alpha x_i \sigma'_j(h_j) \sum_{k=1}^{l} (-e_k w^o_{jk}) \)
Error backpropagation algorithm

- Initialize all weights to small random numbers

**Repeat:**

1. Input training example and compute network outputs.
2. Adjust output weights using gradients:
   \[
   w_{jk}^o(p+1) = w_{jk}^o(p) + \alpha h_j e_k
   \]
3. Adjust hidden-layer weights:
   \[
   w_{ij}^h(p+1) = w_{ij}^h(p) + \alpha x_i \sigma'_j(h_j) \sum_{k=1}^l (-e_k w_{jk}^o)
   \]

**Until** satisfied or fixed number of epochs \( p \)

---

First-order gradient methods

Diagram showing the optimization process with a function \( J(w) \) and steps to minimize it.
Second-order gradient methods

- **Update rule** for the weights:
  \[ w(p + 1) = w(p) - H(w(p))\nabla J(w(p)) \]
  \[ w(p) = w^h_i, w^o_{jk}, \ldots \]

- **H(w)** is the Hessian matrix of **w**

- Learning does not depend on a learning coefficient \( \alpha \)
- Much more efficient in general
Approximation power

- General function approximators
- “Feedforward neural network with one hidden layer and sigmoidal activation functions can approximate any continuous function arbitrarily well on a compact set” (Cybenko)
- Intuitive relation to localized receptive fields
- Little constructive results

Function approximation

\[ y = w_1^o \tanh(w_1^h x + b_1^h) + w_2^o \tanh(w_2^h x + b_2^h) \]

Activation (weighted summation)
Function approximation

Transformation through $\tanh$ of $z$

Summation of neuron outputs

RADIAL BASIS FUNCTION NETWORKS
Radial Basis Function Networks (RBFN)

- Feedforward neural networks where hidden units do not implement an activation function; they represent a radial basis function.
- Developed as an approach to improve accuracy and decrease training time complexity.

Radial Basis Function Networks

- Activation functions are radial basis functions
- Activation level of \( i \)th receptive field (hidden unit):
  \[
  R_i(x) = R_i\left(\frac{\|x - u_i\|}{\sigma_i}\right)
  \]
  
  - \( u_i \) - center of basis function
  - \( \sigma_i \) - spread of basis function
  - \( j = 1, 2, ..., n \)
  - No connection weights between input and hidden layers
Radial Basis Function Networks

- Localized activation functions. Gaussian and logistic:

\[
R_i(x) = \exp\left(-\frac{\|x - u_i\|^2}{2\sigma_i^2}\right)
\]

\[
R'_i(x) = \frac{1}{1 + \exp\left(\frac{\|x - u_i\|^2}{\sigma_i^2}\right)}
\]

- Weighted sum or average output:

\[
y(x) = \sum_{i=1}^{H} c_i w_i = \sum_{i=1}^{H} c_i R'_i(x)
\]

\[
y(x) = \frac{\sum_{i=1}^{H} c_i R_i(x)}{\sum_{i=1}^{H} R_i(x)}
\]

- \(c_i\) can be constants or functions of inputs: \(c_i = a_i^T x + b_i\)

RBFN architecture

- Weighted sum
- Weighted average

Localized activation functions in the hidden layer
RBFN learning

- Supervised learning to update all parameters (e.g. with Genetic Algorithms)
- Sequential training: fix basis functions and then adjust output weights by:
  - orthogonal least squares
  - data clustering
  - soft competition based on “maximum likelihood estimate”
- $\sigma_i$ sometimes estimated based on standard deviations
- Many other schemes also exist

Least-squares estimate of weights

- Given basis functions $R$ and a set of input-output data: $[x_k, y_k], k = 1, \ldots, N$, estimate optimal weights $c_{ij}$

1. Compute the output of the neurons:

   $$ R_i(x_k) = e^{-\frac{||s_i - u_i||^2}{2\sigma_i^2}} $$

   The output is linear in the weights: $y = R \cdot c$.

2. Least squares estimate:

   $$ c = [R^T R]^{-1} R^T y $$
RB FN and Sugeno systems

Equivalent if the following hold:

- Both RBFN and TS use same aggregation method for output (weighted sum or weighted average).
- Number of basis functions in RBFN equals number of rules in TS.
- TS uses Gaussian membership functions with same $\sigma$ (variance) as basis functions and rule firing is determined by product.
- RBFN response function ($c_i$) and TS rule consequents are equal.
Approximation properties of NN

- [Cybenko, 1989]: A feedforward NN with at least one hidden layer can approximate any continuous function $\mathbb{R}^p \rightarrow \mathbb{R}^n$ on a compact interval, if sufficient hidden neurons are available.

- [Barron, 1993]: A feedforward NN with one hidden layer and sigmoidal activation functions can achieve an integrated squared error of the order $J = O(1/h)$.
  - independently of the dimension of the input space $p$
  - $h$: number of hidden neurons (for smooth functions)

Approximation properties

- For a basis function expansion (polynomial, trigonometric, singleton fuzzy model, etc.) with $h$ terms, $J = O(1/h^{2/p})$, where $p$ is the dimension of the input.

Examples:
1. $p = 2$: polynomial $J = O(1/h^{2/2}) = O(1/h)$
   neural net $J = O(1/h)$
2. $p = 10$, $h = 21$: polynomial $J = O(1/21^{2/10}) = 0.54$
   neural net $J = O(1/21) = 0.048$
Example of approximation

- To achieve the same accuracy:

  - \( J = \mathcal{O}(1 / h_n) = \mathcal{O}(1 / h_b) \),
  - \( h_n = h_b^{2/\rho} \),
  - \( h_b = \sqrt[h_n]{h_b^\rho} = \sqrt{21^{10}} \approx 4 \times 10^6 \)

Hopfield network

- **Recurrent ANN.** Example (single-layer):

  - Learning capability is much higher.
  - Successive iterations may not necessarily converge; may lead to chaotic behavior (unstable network).
Feedforward backpropagation network

1. Input and target
   - \( P = [0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10]; \) %input
   - \( T = [0 \ 1 \ 2 \ 3 \ 4 \ 3 \ 2 \ 1 \ 2 \ 3 \ 4]; \) %target

2. Create net
   - help newff
   - net = newff(P,T,5);

3. Simulate and plot net
   - \( Yi = \text{sim}(\text{net},P); \)
   - plot(P,T,'rs-',P,Yi,'-o')
   - legend('T','Yi',0),xlabel('P')
4. Train the network for 50 epochs
   - `net.trainParam.epochs = 50;`
   - `net = train(net,P,T);`
   - `T = [0 1 2 3 4 3 2 1 2 3 4]; %target`

5. Simulate net and plot the results
   - `Y = sim(net,P);`
   - `figure;`
   - `plot(P,T,'rs-','P,Yi','bo','P,Y','g^');`
   - `legend('T','Yi','Y',0),xlabel('P')`
Compute the mean absolute and squared errors

- \( \text{ma\_error} = \text{mae}(T-Y) \)
  
  \( \text{ma\_error} = 0.1120 \)

- \( \text{ms\_error} = \text{mse}(T-Y) \)
  
  \( \text{ms\_error} = 0.0169 \)

Plot the network error

- \( \text{figure,plot}(P,T-Y,'o'), \text{grid} \)
- \( \text{ylabel('error'),xlabel('P')} \)
Check the parameters of the network

- `net`

Some important parameters

- inputs: `{1x1 cell}` of inputs
- layers: `{2x1 cell}` of layers
- outputs: `{1x2 cell}` containing 1 output
- targets: `{1x2 cell}` containing 1 target
- biases: `{2x1 cell}` containing 2 biases
- inputWeights: `{2x1 cell}` containing 1 input weight
- layerWeights: `{2x2 cell}` containing 1 layer weight
Feedforward backpropagation network

- `adaptFcn: 'trains'`
- `initFcn: 'initlay'`
- `performFcn: 'mse'`
- `trainFcn: 'trainlm'`
- `adaptParam: .passes`
- `trainParam: .epochs, .goal, .show, .time`
- `IW: {2x1 cell} containing 1 input weight matrix`
- `LW: {2x2 cell} containing 1 layer weight matrix`
- `b: {2x1 cell} containing 2 bias vectors`

Note that every time that a network is initialized, different random numbers are used for the weights.

Example in the following:
- Initialization and training of 10 networks
- Computation of mean absolute error
- Computation of mean squared error
Feedforward backpropagation network

- MA_error = []; MS_error = [];
- for i = 1:10
  - net = newff(P,T,5);
  - net.trainParam.epochs = 50;
  - net = train(net,P,T);
  - Y = sim(net,P);
  - MA_error = [MA_error mae(T-Y)];
  - MS_error = [MS_error mse(T-Y)];
- end

Feedforward backpropagation network

- figure,
- subplot(2,1,1),plot(MA_error,'o'),grid,
- title('Mean Absolute Error')
- subplot(2,1,2),plot(MS_error,'o'),grid
- title('Mean Squared Error')