Classification
Artificial Neural Network (ANN)

Huiping Cao
Outline

1. Perceptron Model
2. Multilayer ANN
3. Discussions
Motivation

- Inspired by biological neural systems
  - We are interested in replicating the **biological function**
  - The first step is to replicate the **biological structure**

- From the bird to plane
- **Neurons**: nerve cells
- **Axons**: strands of fibers, for linking neurons
- **Dendrites**: extensions from the cell body of the neuron, connects neurons and axons
- **Synapse**: the contact point between a dendrite and an axon
## Perceptron Model – Example

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

### Diagram

[Diagram of a perceptron model with input nodes $x_1$, $x_2$, $x_3$, and output nodes showing the summation and threshold $t=0.4$.]
**Perceptron Model – Concepts**

- **A perceptron** is a single processing unit of a neural net.
- **Nodes** in a neural network architecture are commonly known as **neurons** or units.
  - Input nodes: represent the input attributes
  - Output node: represent the model output
- **Weighted links**: emulate the strength of synaptic connection between neurons.

\[
\hat{y} = \begin{cases} 
1, & \text{if } 0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4 > 0 \\
-1, & \text{if } 0.3x_1 + 0.3x_2 + 0.3x_3 - 0.4 < 0 
\end{cases}
\]

- **Activation functions**

\[
\hat{y} = \text{sign}[w_d x_d + w_{d-1} x_{d-1} + \cdots + w_1 x_1 + w_0 x_0] = \text{sign}[w \cdot x]
\]

where \(w_0 = -t\) and \(x_0 = 1\).
A perceptron takes \( n \) inputs \( x_1 \) to \( x_n \). Each input \( x_j \) has an associated weight \( w_j \).

The output of the perceptron is produced by a two-stage process.

- The first stage computes a quantity which is the weighted sum of the inputs \( in = \sum_{j=1}^{n} w_j x_j \).
- The second stage applies an activation function \( g \) to \( in \). For perceptrons, the activation function used is the threshold activation function

\[
\hat{y} = g(in) = \begin{cases} 
1 & \text{if } in > \text{threshold } \sigma \\
-1 & \text{otherwise}
\end{cases}
\]
The threshold $\sigma$ is a parameter of the activation function.

$$
\hat{y} = g(in) = \begin{cases} 
1 & \text{if } in > \text{threshold } \sigma \\
-1 & \text{otherwise}
\end{cases}
$$

An alternative way of encoding a threshold activation function is to create a special input $x_0$.

- This input will always be fixed at -1 for every instance.
- The weight $w_0$ associated with this input can then be used in space of the threshold $\sigma$ (i.e., $w_0 = \sigma$).

Thus,

$$in = \sum_{j=0}^{n} w_j x_j$$

where $x_0 = -1$ and $w_0 = \sigma$

$$
\hat{y} = g(in) = \begin{cases} 
1 & \text{if } in > 0 \\
-1 & \text{otherwise}
\end{cases}
$$
Activation functions

- Sigmoid function \( \text{sigmoid}(s) = \theta(s) = \frac{e^s}{1+e^s} \). Its output is between 0 and 1. Its shape looks like a flattened out ‘s’. It is also called a \textit{logistic function}.

- Hyperbolic tangent function: \( \tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}} \)
Learning Perceptron Model

Perceptron Learning Algorithm (PLA)

- **Training a perceptron model**: adapting the weights of the links until they fit the input-output relationships of the underlying data.
- **Input**: \( D = \{(x_i, y_i)|i = 1, 2, \cdots, N\} \) be the set of training examples
- **Initialize the weight vector with random values** \( \mathbf{w}^{(0)} \).
- **repeat** (iteration no is \( k \))
  - **for** each training point \( (x_i, y_i) \in D \) do
    - Compute the predicted output \( \hat{y}_i^{(k)} \)
    - **for** each weight \( w_j \)
      - Update \( w_j^{(k+1)} = w_j^{(k)} + \lambda(y_i - \hat{y}_i^{(k)})x_{ij} \)
  - **until** stopping condition is met
Perceptron Learning Algorithm (PLA)

- **Weight update formula**: new weight is a combination of the old weight \( w_j^{(k)} \) and a term proportional to the prediction error \( (y_i - \hat{y}_i^{(k)}) \).

\[
    w_j^{(k+1)} = w_j^{(k)} + \lambda (y_i - \hat{y}_i^{(k)})x_{ij}
\]

- \( \lambda \): learning rate, \( \in [0, 1] \); it can be fixed or adaptive.

- The model is **linear** in its parameters \( \mathbf{w} \) and attributes \( \mathbf{x} \).
Perceptron Model – Linearly Separable

- A perceptron is a classifier, which takes $n$ continuous inputs, and returns a Boolean classification (+1 or -1).

- All the points s.t. $in(x) = 0$ defines a hyperplane in the space of inputs.
  - On one side of this hyperplane, all points are classified as +1.
  - On the other side, they are classified as -1.

- **Decision boundary**: A surface that divides the input space into regions of different classes.

- Perceptrons always have a linear decision boundary.

- **Linearly separable**: a classification function with a linear decision boundary.
Perceptron examples – Linearly Separable

- Perceptrons can represent many (but not ALL) Boolean functions on two inputs.

- A perceptron can represent the AND function by setting the weights to be $+1$ and the threefold to 1.5.
Perceptron examples – Non-linearly Separable

- XOR function on two inputs?

- Intuitively: we cannot find a line that can separate the four points with the points on one side having the same class labels.

Formally:

- Suppose that we can represent XOR using weights \( w = (w_0, w_1, w_2) \).
- (1) \( (x_1 = -1) \land (x_2 = -1) \), output is -1: \(-w_0 - w_1 - w_2 \leq 0\)
- (2) \( (x_1 = -1) \land (x_2 = 1) \), output is 1: \(-w_0 - w_1 + w_2 > 0\)
- So, \( w_2 \) is positive.

- (3) \( (x_1 = 1) \land (x_2 = 1) \), output is -1: \(-w_0 + w_1 + w_2 \leq 0\)
- (4) \( (x_1 = 1) \land (x_2 = -1) \), output is 1: \(-w_0 + w_1 - w_2 > 0\)
- So, \( w_2 \) is negative.
- Contradiction!
Perceptron Properties

- If the problem is **linearly separable**, PLA is guaranteed to **converge** to an optimal solution.

- If the problem is **not linearly separable**, the algorithm **fails to converge**.
The neural network

- **Hidden layer, hidden nodes**
- **Feed-forward NN**: the nodes in one layer are connected only to the nodes in the next layer.
- **Recurrent NN**: the links may connect nodes within the same layer or nodes from one layer to the previous layers.
- **Activation functions**: sign, linear, sigmoid (logistic), hyperbolic tangent
Learning the ANN model

- **Target function**: the goal of the ANN learning algorithm is to determine a set of weights $\mathbf{w}$ that minimize the total sum of squared errors.

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

- When $\hat{y}$ is a linear function of its parameters, we can replace $\hat{y} = \mathbf{w} \cdot \mathbf{x}$ into the above equation, then the error function becomes quadratic in its parameters.
In most cases, the output of an ANN is a **nonlinear function** of its parameters because of the choice of its activation functions (e.g., *sigmoid* or *tanh* function).

It is **no longer straightforward** to derive a solution for $w$ that is guaranteed to be globally optimal.

**Greedy algorithm** (e.g., based on **gradient descent methods**) are typically developed.

**Update formula**

$$w_j = w_j - \lambda \frac{\partial E(w)}{\partial w_j}$$
Gradient descent

- For hidden nodes, the computation is not trivial because it is difficult to assess their error term $\frac{\partial E(w)}{\partial w_j}$ without knowing what their output values should be.

- **Back-propagation** technique: two phases in each iteration
  - **forward** phase: at the $i$th iteration, $w^{(i-1)}$ are used to calculate the output value of each neuron in the network. 
    *Outputs of neurons* at level $k$ are computed before computing the outputs at level $k + 1$. I.e., $k \rightarrow k + 1$
  - **backward** phase: update of $w^{(i)}$ is applied in the reverse direction. 
    *Weights* at level $k + 1$ are updated before the weights at level $k$ are updated. I.e., $k + 1 \rightarrow k
Neural network R examples

```r
> install.packages("neuralnet")
> require(neuralnet)

> XOR <- c(0,1,1,0)
> xor.data <- data.frame(expand.grid(c(0,1), c(0,1)), XOR)
> xor.data
   Var1 Var2 XOR
 1   0   0   0
 2   1   0   1
 3   0   1   1
 4   1   1   0

> net.xor <- neuralnet(XOR~Var1+Var2, xor.data, hidden=2, rep=5)
> print(net.xor)
> plot(net.xor, rep="best")
```

- **Usage and more examples** see https://cran.r-project.org/web/packages/neuralnet/neuralnet.pdf

- **Function** `expand.grid {base}`: Create a data frame from all combinations of the supplied vectors or factors.

Huiping Cao, Slide 19/21
Design issues in ANN learning

- Assign an input node to each numerical or binary input variable.

- **Output nodes**: for two-class problem, one output node; k-class problem, k output nodes.

- **Target function representation** factors: (1) weights of the links, (2) the number of hidden nodes and hidden layers, (3) biases in the nodes, and (4) type of activation function

- Finding the right **topology** is not easy.

- The **initial weights and biases** can come from random assignments.

- Fix the **missing values** first.
Characteristics of ANN

- Universal **approximators**: they can be used to approximate any target functions.

- Can handle **redundant features**.

- Sensitive to the presence of **noise**.

- The gradient descent method often converges to **local minimum**. One way: add a momentum term to the weight update formula.

- **Training** is time consuming.

- **Testing** is rapidly.