L13
NONLINEAR CLASSIFIERS: MULTILAYER PERCEPTRONS
Outline

- Combining Linear Classifiers
- Learning Parameters
Outline

- Combining Linear Classifiers
- Learning Parameters
AND and OR operations are linearly separable problems.
The XOR Problem

- XOR is not linearly separable.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>XOR</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>B</td>
</tr>
</tbody>
</table>

- How can we use linear classifiers to solve this problem?
Combining two linear classifiers

- Idea: use a logical combination of two linear classifiers.

\[ g_1(x) = x_1 + x_2 - \frac{1}{2} \]

\[ g_2(x) = x_1 + x_2 - \frac{3}{2} \]
Combining two linear classifiers

Let \( f(x) \) be the unit step activation function:
\[
f(x) = 0, \quad x < 0 \\
f(x) = 1, \quad x \geq 0
\]

Observe that the classification problem is then solved by
\[
f\left( y_1 - y_2 - \frac{1}{2} \right)
\]
where
\[
y_1 = f(g_1(x)) \quad \text{and} \quad y_2 = f(g_2(x))
\]

\[
g_1(x) = x_1 + x_2 - \frac{1}{2}
\]
\[
g_2(x) = x_1 + x_2 - \frac{3}{2}
\]
Combining two linear classifiers

- This calculation can be implemented sequentially:
  1. Compute $y_1$ and $y_2$ from $x_1$ and $x_2$.
  2. Compute the decision from $y_1$ and $y_2$.
- Each layer in the sequence consists of one or more linear classifications.
- This is therefore a two-layer perceptron.

\[
\begin{align*}
  f\left(y_1 - y_2 - \frac{1}{2}\right) \\
  \text{where} \\
  y_1 &= f\left(g_1(x)\right) \quad \text{and} \quad y_2 = f\left(g_2(x)\right) \\
  g_1(x) &= x_1 + x_2 - \frac{1}{2} \\
  g_2(x) &= x_1 + x_2 - \frac{3}{2}
\end{align*}
\]
The Two-Layer Perceptron

\[ y_1 = f\left( g_1(x) \right) \text{ and } y_2 = f\left( g_2(x) \right) \]

where

\[ f\left( y_1 - y_2 - \frac{1}{2} \right) \]

Layer 1

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( y_1 )</th>
<th>( y_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0(-)</td>
<td>0(-)</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1(+)</td>
<td>0(-)</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1(+)</td>
<td>0(-)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1(+)</td>
<td>1(+)</td>
</tr>
</tbody>
</table>

Layer 2

\[ g_2(x) = x_1 + x_2 - \frac{3}{2} \]

\[ g_1(x) = x_1 + x_2 - \frac{1}{2} \]
The Two-Layer Perceptron

The first layer performs a nonlinear mapping that makes the data linearly separable.

\[ y_1 = f(g_1(x)) \] and \[ y_2 = f(g_2(x)) \]

\[ g_1(x) = x_1 + x_2 - \frac{3}{2}x_2 \]

\[ g_2(x) = x_1 + x_2 - \frac{1}{2} \]
The Two-Layer Perceptron Architecture

Input Layer

Hidden Layer

Output Layer

\[ g_1(x) = x_1 + x_2 - \frac{1}{2} \]

\[ g_2(x) = x_1 + x_2 - \frac{3}{2} \]

\[ y_1 - y_2 - \frac{1}{2} \]
Note that the hidden layer maps the plane onto the vertices of a unit square.

\[ y_1 = f(g_1(x)) \text{ and } y_2 = f(g_2(x)) \]
Higher Dimensions

- Each hidden unit realizes a hyperplane discriminant function.
- The output of each hidden unit is 0 or 1 depending upon the location of the input vector relative to the hyperplane.

\[ x \in \mathbb{R}^l \quad \rightarrow \quad y = [y_1, \ldots, y_p]^T, \quad y_i \in \{0, 1\} \quad i = 1, 2, \ldots, p \]
Together, the hidden units map the input onto the vertices of a $p$-dimensional unit hypercube.

$x \in \mathbb{R}^l \rightarrow y = [y_1, \ldots y_p]^T$, $y_i \in \{0, 1\}$ $i = 1, 2, \ldots p$
Two-Layer Perceptron

- These $p$ hyperplanes partition the $l$-dimensional input space into polyhedral regions.
- Each region corresponds to a different vertex of the $p$-dimensional hypercube represented by the outputs of the hidden layer.
In this example, the vertex \((0, 0, 1)\) corresponds to the region of the input space where:

- \(g_1(x) < 0\)
- \(g_2(x) < 0\)
- \(g_3(x) > 0\)
Limitations of a Two-Layer Perceptron

- The output neuron realizes a hyperplane in the transformed space that partitions the \( p \) vertices into two sets.
- Thus, the two layer perceptron has the capability to classify vectors into classes that consist of unions of polyhedral regions.
- But **NOT ANY** union. It depends on the relative position of the corresponding vertices.
- How can we solve this problem?
Suppose that Class A consists of the union of $K$ polyhedra in the input space.

Use $K$ neurons in the 2nd hidden layer.

Train each to classify one Class A vertex as positive, the rest negative.

Now use an output neuron that implements the OR function.
Thus the three-layer perceptron can separate classes resulting from any union of polyhedral regions in the input space.
The Three-Layer Perceptron

- The first layer of the network forms the **hyperplanes** in the input space.
- The second layer of the network forms the **polyhedral regions** of the input space.
- The third layer forms the appropriate **unions of these regions** and maps each to the appropriate class.

![Diagram of a three-layer perceptron]

- **Input layer**
- **1st hidden layer**
- **2nd hidden layer**
- **Output layer**
Outline

- Combining Linear Classifiers
- Learning Parameters
The training data consist of $N$ input-output pairs:

$$\left( y(i), x(i) \right), \quad i \in 1, \ldots, N$$

where

$$y(i) = \begin{bmatrix} y_1(i), \ldots, y_{k_l}(i) \end{bmatrix}^t$$

and

$$x(i) = \begin{bmatrix} x_1(i), \ldots, x_{k_0}(i) \end{bmatrix}^t$$
Choosing an Activation Function

- The unit step activation function means that the error rate of the network is a discontinuous function of the weights.
- This makes it difficult to learn optimal weights by minimizing the error.
- To fix this problem, we need to use a smooth activation function.
- A popular choice is the sigmoid function we used for logistic regression:
Smooth Activation Function

\[ f(a) = \frac{1}{1 + \exp(-a)} \]
For a binary classification problem, there is a single output node with activation function given by

\[ f(a) = \frac{1}{1 + \exp(-a)} \]

Since the output is constrained to lie between 0 and 1, it can be interpreted as the probability of the input vector belonging to Class 1.
For a $K$-class problem, we use $K$ outputs, and the softmax function given by

$$y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)}$$

Since the outputs are constrained to lie between 0 and 1, and sum to 1, $y_k$ can be interpreted as the probability that the input vector belongs to Class $K$. 
Now each layer of our multi-layer perceptron is a logistic regressor.

Recall that optimizing the weights in logistic regression results in a convex optimization problem.

Unfortunately the cascading of logistic regressors in the multi-layer perceptron makes the problem non-convex.

This makes it difficult to determine an exact solution.

Instead, we typically use gradient descent to find a locally optimal solution to the weights.

The specific learning algorithm is called the backpropagation algorithm.
Nonlinear Classification and Regression: Outline

- Multi-Layer Perceptrons
  - The Back-Propagation Learning Algorithm
- Generalized Linear Models
  - Radial Basis Function Networks
  - Sparse Kernel Machines
    - Nonlinear SVMs and the Kernel Trick
    - Relevance Vector Machines
The Backpropagation Algorithm


Backpropagation: Summary of Algorithm

1. Initialization
   - Initialize all weights with small random values

2. Forward Pass
   - For each input vector, run the network in the forward direction, calculating:
     \[ v_j^r(i) = \left( w_j^r \right)^T \mathbf{y}^{r-1}(i); \quad y_j^r(i) = f \left( v_j^r(i) \right) \]
   and finally
     \[ \varepsilon(i) = \frac{1}{2} \sum_{m=1}^{k} \left( e_m(i) \right)^2 = \frac{1}{2} \sum_{m=1}^{k} \left( \hat{y}_m(i) - y_m(i) \right)^2 \]

3. Backward Pass
   - Starting with the output layer, use our inductive formula to compute the \( \delta_j^{r-1}(i) \):
     - Output Layer (Base Case): \( \delta_j^L(i) = e_j^L(i)f'(v_j^L(i)) \)
     - Hidden Layers (Inductive Case): \( \delta_j^{r-1}(i) = f'(v_j^{r-1}(i)) \sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r \)

4. Update Weights
   \[ w_j^r(\text{new}) = w_j^r(\text{old}) - \mu \sum_{i=1}^{N} \frac{\partial \varepsilon(i)}{\partial w_j^r} \quad \text{where} \quad \frac{\partial \varepsilon(i)}{\partial w_j^r} = \delta_j^r(i)y^{r-1}(i) \]
Batch vs Online Learning

- As described, on each iteration backprop updates the weights based upon all of the training data. This is called **batch learning**.

\[ w_j r^{(new)} = w_j r^{(old)} - \mu \sum_{i=1}^{N} \frac{\partial \epsilon(i)}{\partial w_j r} \text{ where } \frac{\partial \epsilon(i)}{\partial w_j r} = \delta_j r(i)y_{r-1}(i) \]

- An alternative is to update the weights after each training input has been processed by the network, based only upon the error for that input. This is called **online learning**.

\[ w_j r^{(new)} = w_j r^{(old)} - \mu \frac{\partial \epsilon(i)}{\partial w_j r} \text{ where } \frac{\partial \epsilon(i)}{\partial w_j r} = \delta_j r(i)y_{r-1}(i) \]
Batch vs Online Learning

- One advantage of batch learning is that averaging over all inputs when updating the weights should lead to smoother convergence.
- On the other hand, the randomness associated with online learning might help to prevent convergence toward a local minimum.
- Changing the order of presentation of the inputs from epoch to epoch may also improve results.
Remarks

- **Local Minima**
  - The objective function is in general non-convex, and so the solution may not be globally optimal.

- **Stopping Criterion**
  - Typically stop when the change in weights or the change in the error function falls below a threshold.

- **Learning Rate**
  - The speed and reliability of convergence depends on the learning rate $\mu$. 
Limiting network complexity

Number of hidden layers

- Given a sufficiently large number of hidden neurons, a two-layer MLP can approximate any continuous function arbitrarily well
  - The example below shows that a combination of four hidden neurons can produce a “bump” at the output space of a two-layered MLP
  - A large number of “bumps” can approximate any surface arbitrarily well
- Nonetheless, the addition of extra hidden layers may allow the MLP to approximate more efficiently, i.e., with fewer weights [Bishop, 1995]
Number of hidden neurons

- While the number of inputs and outputs are dictated by the problem, the number of hidden units $N_H$ is not related so explicitly to the application domain
  - $N_H$ determines the degrees of freedom or expressive power of the model
  - A small $N_H$ may not be sufficient to model complex I/O mappings
  - A large $N_H$ may overfit the training data and prevent the network from generalizing to new examples
- Despite a number of “rules of thumb” published in the literature, a-priori determination of an appropriate $N_H$ is an unsolved problem
  - The “optimal” $N_H$ depends on multiple factors, including number of examples, level of noise in the training set, complexity of the classification problem, number of inputs and outputs, activation functions and training algorithm.
  - In practice, several MLPs are trained and evaluated in order to determine an appropriate $N_H$
- A number of adaptive approaches have also been proposed
  - **Constructive** approaches start with a small network and incrementally add hidden neurons (e.g., cascade correlation),
  - **Pruning** approaches, which start with a relatively large network and incrementally remove weights (e.g., optimal brain damage)
Weight decay

– To prevent the weights from growing too large (a sign of over-training) it is convenient to add a decay term of the form

\[ w(n + 1) = (1 - \epsilon)w(n) \]

• Weights that are not needed eventually decay to zero, whereas necessary weights are continuously updated by back-prop

– Weight decay is a simple form of regularization, which encourages smoother network mappings [Bishop, 1995]

Early stopping

– Early stopping can be used to prevent the MLP from over-fitting the training set

– The stopping point may be determined by monitoring the sum-squared-error of the MLP on a validation set during training

Training with noise (jitter)

– Training with noise prevents the MLP from approximating the training set too closely, which leads to improved generalization
Tricks of the trade

**Activation function**
- An MLP trained with backprop will generally train faster if the activation function is anti-symmetric: \( f(-x) = -f(x) \) (e.g., the hyperbolic tangent)

**Target values**
- Obviously, it is important that the target values are within the dynamic range of the activation function, otherwise the neuron will be unable to produce them
- In addition, it is recommended that the target values are not the asymptotic values of the activation function; otherwise backprop will tend to drive the neurons into saturation, slowing down the learning process
  - The slope of the activation function, which is proportional to \( \Delta w \), becomes zero at \( \pm \infty \)

**Input normalization**
- Input variables should be preprocessed so that their mean value is zero or small compared to the variance
- Input variables should be uncorrelated (use PCA to accomplish this)
- Input variables should have the same variance (use Fukunaga’s whitening transform)
Initial weights

- Initial random weights should be small to avoid driving the neurons into saturation
  - HO weights should be made larger than IH weights since they carry the back propagated error
  - If the initial HO weights are very small, the weight changes at the IH layer will initially be very small, slowing the training process

Weight updates

- The derivation of backprop was based on one training example but, in practice, the data set contains a large number of examples
- There are two basic approaches for updating the weights during training
  - On-line training: weights are updated after presentation of each example
  - Batch training: weights are updated after presentation of all the examples (we store the $\Delta w$ for each example, and add them up to the weight after all the examples have been presented)
- Batch training is recommended
  - Batch training uses the TRUE steepest descent direction
  - On-line training achieves lower errors earlier in training but only because the weights are updated $n$ (# examples) times faster than in batch mode
  - On-line training is sensitive to the ordering of examples