COMP9444: Neural Networks

Review
Topics Covered (Weeks 1-5)

- Neuroanatomy (not examinable)
- Perceptrons and Backpropagation
- Geometry and Applications
- VC-dimension, PAC-learning and SRM
- Support Vector Machines
- Committee Machines (Boosting, Bagging)
Rosenblatt Perceptron

\[
x_1
\]
\[
x_2
\]
\[
\Sigma
\]
\[
\] \rightarrow \quad s \quad \rightarrow \quad g
\]
\[
g(s)
\]

\[
1
\]

\[
\begin{align*}
  x_1, x_2 & \text{ are inputs} \\
  w_1, w_2 & \text{ are synaptic weights} \\
  w_0 & \text{ is a bias weight} \\
  \text{th} & \text{ is a threshold} \\
  g & \text{ is transfer function}
\end{align*}
\]

\[
s = w_1x_1 + w_2x_2 - \text{th}
\]
\[
= w_1x_1 + w_2x_2 + w_0
\]
Perceptron Learning Rule

Adjust the weights as each input is presented.

recall: \( s = w_1 x_1 + w_2 x_2 + w_0 \),

\( \eta > 0 \) is called the **learning rate**

if \( g(s) = 0 \) but should be 1, \[ w_k \leftarrow w_k + \eta x_k \]
\[ w_0 \leftarrow w_0 + \eta \]

so \( s \leftarrow s + \eta (1 + \sum_k x_k^2) \)

if \( g(s) = 1 \) but should be 0, \[ w_k \leftarrow w_k - \eta x_k \]
\[ w_0 \leftarrow w_0 - \eta \]

so \( s \leftarrow s - \eta (1 + \sum_k x_k^2) \)

otherwise, weights are unchanged.

**Theorem:** This will learn to classify the data correctly, as long as they are **linearly separable**.
Linear Separability

Q: what kind of functions can a perceptron compute?

A: linearly separable functions

Examples include:

AND \quad w_1 = w_2 = 1.0, \quad w_0 = -1.5

OR \quad w_1 = w_2 = 1.0, \quad w_0 = -0.5

NOR \quad w_1 = w_2 = -1.0, \quad w_0 = 0.5

Q: How do we train it to learn a new function?
Multi-Layer Neural Networks

Problem: How do we train it to learn a new function? (credit assignment)
Weight Decay

Assume that small weights are more likely to occur than large weights, i.e.

\[ P(w) = \frac{1}{Z} e^{-\frac{\lambda}{2} \sum_j w_j^2} \]

where \( Z \) is a normalizing constant. Then the cost function becomes:

\[ E = \frac{1}{2} \sum_i (z_i - t_i)^2 + \frac{\lambda}{2} \sum_j w_j^2 \]

This can prevent the weights from “saturating” to very high values.

Problem: need to determine \( \lambda \) from experience, or empirically.
Momentum

If landscape is shaped like a “rain gutter”, weights will tend to oscillate without much improvement.

Solution: add a momentum factor

\[ \delta w \leftarrow \alpha \delta w + (1 - \alpha) \frac{\partial E}{\partial w} \]

\[ w \leftarrow w - \eta \delta w \]

Hopefully, this will dampen sideways oscillations but amplify downhill motion by \( \frac{1}{1-\alpha} \).
Case Studies

- Encoder networks
- Twin Spirals
- Face Recognition
- ALVINN
- TD-Gammon
VC-dimension and PAC-Learning

**Theorem** Upper Bound (Blumer et al., 1989)

Let \( L \) be a learning algorithm that uses \( F \) consistently, i.e. that finds an \( f \in F \) that is consistent with all the data. For any \( 0 < \varepsilon, \delta < 1 \) given

\[
\frac{(4\log\left(\frac{2}{\delta}\right) + 8\text{VC-dim}(F)\log\left(\frac{13}{\varepsilon}\right))}{\varepsilon}
\]

random examples, \( L \) will with probability of at least \( 1 - \delta \)

either produce a function \( f \in F \) with error \( \leq \varepsilon \)

or indicate correctly, that the target function is not in \( F \).
PAC Learning

What assumptions are made?

- data come from a fixed distribution and are correctly classified
- number of training samples is at least $f(\varepsilon, \delta, h)$
  where $h$ is the VC-dimension.

How is the error measured?

- with probability at least $(1 - \delta)$, error must be $\leq \varepsilon$. 
The Vapnik-Chervonenkis dimension

The VC-dimension is a useful combinatorial parameter on sets of subsets, e.g. on concept classes or hypothesis classes.

**Definition**

We say a set $S \subseteq X$ is **shattered** by $C$ if $\{S \cap c | c \in C\} = 2^S$.

The Vapnik-Chervonenkis dimension of $C$, $\text{VC-dim}(C)$, is the cardinality of the greatest set $S \subseteq X$ shattered by $C$, i.e.

$$\text{VC-dim}(C) = \max_{S \in \{S | S \subseteq X \land \{S \cap c | c \in C\} = 2^S\}} |S|$$
VC Dimension

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Optimal Hyperplane

Suppose the training data

\[(\vec{x}_1, d_1), \ldots, (\vec{x}_N, d_N), \vec{x} \in \mathbb{R}^m, \ d \in \{-1, +1\}\]

can be separated by a hyperplane

\[\vec{w}^T \vec{x} + b = 0.\]

The hyperplane which separates the training data without error and has maximal distance to the closest training vector is called the Optimal Hyperplane.
Support Vectors

The Optimal separating hyperplane has maximal margin to the training examples. Those lying on the boundary are called the Support Vectors.
Support Vector Machines

A Support Vector Machine maps the input space (nonlinearly) into a high-dimensional feature space and then constructs an Optimal Hyperplane in the feature space.
The Kernel Trick

The features \((\phi_1(\vec{x}), \ldots, \phi_m(\vec{x}))\) will play the same role that was previously played by the co-ordinates of \(\vec{x}\). By convention, we add an additional “bias” feature \(\phi_0()\) with \(\phi_0(\vec{x}) = 1\) for all \(\vec{x}\). Recall that we only need to be able to compute the dot products

\[
K(\vec{x}_i, \vec{x}_j) = \vec{\phi}_i^T \vec{\phi}_j = \sum_{k=0}^{m} \phi_k(\vec{x}_i)\phi_k(\vec{x}_j).
\]

Fortunately \(K(\vec{x}_i, \vec{x}_j)\), which is known as a **Kernel function**, can often be computed directly without having to compute the individual \(\phi_k\)'s. This is the key idea which makes Support Vector Machines computable.
AdaBoost

- given: \( N \) training items \((\vec{x}_1, d_1) \ldots (\vec{x}_N, d_N)\)
- train a series of learners \( C_1 \ldots C_T \) producing hypotheses \( F_1 \ldots F_T \)
- training items for \( C_n \) chosen using distribution \( D_n \)
- initialize \( D_1(i) = \frac{1}{N} \) for \( i = 1 \ldots N \)
- set
  \[
  \beta_n = \frac{\varepsilon_n}{1 - \varepsilon_n}, \quad \text{where} \quad \varepsilon_n = \text{training error of } C_n
  \]
- update
  \[
  D_{n+1}(i) = \frac{D_n(i)}{Z_n} \times \left\{ \begin{array}{ll}
  \beta_n, & \text{if } F_n(\vec{x}_i) = d_i \\
  1, & \text{otherwise}
  \end{array} \right.
  \]

where \( Z_n \) is a normalizing constant.
AdaBoost Generalization

- the **base learner** for AdaBoost could be any kind of learner (neural networks, decision trees, stumps ... )

- with AdaBoost, as with SVM’s, the test error often continues to decrease even after the training error has already reached zero

- this goes against the traditional conception of bias-variance tradeoff, Ockham’s Razor and overfitting

- although the number of “free parameters” is enormous, each additional degree of freedom is highly costrained
Good Luck!