Learning, Part 2

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear and Nonlinear</td>
<td>2</td>
</tr>
<tr>
<td>Zero Error Linear Decision Boundary</td>
<td>2</td>
</tr>
<tr>
<td>Low Error Linear Decision Boundary</td>
<td>3</td>
</tr>
<tr>
<td>Nonlinear Decision Boundary</td>
<td>4</td>
</tr>
<tr>
<td>Zoom on Nonlinear Decision Boundary</td>
<td>5</td>
</tr>
<tr>
<td>Neural Networks</td>
<td>6</td>
</tr>
<tr>
<td>Motivation for Neural Networks</td>
<td>6</td>
</tr>
<tr>
<td>Feedforward Neural Networks</td>
<td>7</td>
</tr>
<tr>
<td>Computing Outputs</td>
<td>8</td>
</tr>
<tr>
<td>Xor Network</td>
<td>9</td>
</tr>
<tr>
<td>Computing for Xor</td>
<td>10</td>
</tr>
<tr>
<td>Learning by Backpropagation</td>
<td>11</td>
</tr>
<tr>
<td>Comments on Backpropagation</td>
<td>12</td>
</tr>
<tr>
<td>Support Vector Machines</td>
<td>13</td>
</tr>
<tr>
<td>SVMs</td>
<td>13</td>
</tr>
<tr>
<td>Kernel Functions</td>
<td>14</td>
</tr>
<tr>
<td>Slow SVM Learning Algorithm</td>
<td>15</td>
</tr>
<tr>
<td>Comments on SVMs</td>
<td>16</td>
</tr>
<tr>
<td>The Nearest Neighbor Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>The Nearest Neighbor Algorithm</td>
<td>17</td>
</tr>
<tr>
<td>Ensemble Learning</td>
<td>18</td>
</tr>
<tr>
<td>Ensemble Learning</td>
<td>18</td>
</tr>
<tr>
<td>Boosting</td>
<td>19</td>
</tr>
<tr>
<td>Example Boosting Algorithm</td>
<td>20</td>
</tr>
<tr>
<td>Example Run of AdaBoost</td>
<td>21</td>
</tr>
<tr>
<td>Example Run of AdaBoost, Continued</td>
<td>22</td>
</tr>
<tr>
<td>Overfitting</td>
<td>23</td>
</tr>
<tr>
<td>Overfitting Example</td>
<td>24</td>
</tr>
<tr>
<td>Validation Set</td>
<td>25</td>
</tr>
<tr>
<td>k-Fold Cross-Validation</td>
<td>26</td>
</tr>
<tr>
<td>Regularization</td>
<td>27</td>
</tr>
<tr>
<td>Regularization Example</td>
<td>28</td>
</tr>
<tr>
<td>How to Use a Test Set</td>
<td>29</td>
</tr>
<tr>
<td>How to Use a Test Set</td>
<td>29</td>
</tr>
<tr>
<td>Can I Use the Test Set Again?</td>
<td>30</td>
</tr>
<tr>
<td>Error Rate and Comparing Algorithms</td>
<td>31</td>
</tr>
<tr>
<td>Paired-Difference Test</td>
<td>32</td>
</tr>
</tbody>
</table>
Neural Networks

Motivation for Neural Networks
- Neural networks are inspired by neurons and their connections in the brain.
- An artificial neuron, called a unit, has inputs and an output.
- The output can be connected to other units.
- Typically, the output of a unit is computed by a linear function of its inputs passed through an activation function.
- Learning is adjusting weights to reduce error.
- Advantage: can learn non-linear functions.
- Disadvantage: more parameters and no guarantee of optimality.

Feedforward Neural Networks
- A feed-forward neural network is the most common type. Its units are organized as a directed acyclic graph.
- Input units simply output the values of the input features.
- Hidden units input the values of other units and produce outputs for other units.
- Output units produce predictions of output features.
- One standard model is an initial “layer” of input units, which feed into a single layer of hidden units, which feed into the output units.

Computing Outputs
Procedure `NNOutput(e, H, O)`
Inputs $e = x$: the inputs of an example $e$
- $H$: hidden units with activation function $f_H$ and weights $w_{H_j}$ for each hidden unit $H_j$
- $O$: output unit with activation function $f_O$ and weights $w_O$
for each hidden unit $H_j$
- $h_j = f_H(w_{H_j} \cdot x)$
- $o = f_O(w_O \cdot h)$
return $o$
Xor Network

Input Units  Hidden Units  Output Unit
\(X_0\)  -2  \(H_0\)  -2  Prediction
\(X_1\)  4  \(H_1\)  4
\(X_2\)  4  \(H_2\)  4

Computing for Xor

Suppose \(f(z) = 1/(1 + e^{-z})\) is the activation function. \(X_0\) and \(H_0\) always output 1.

<table>
<thead>
<tr>
<th>(x)</th>
<th>(X_0, X_1, X_2)</th>
<th>(H_1)</th>
<th>(H_2)</th>
<th>(O)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1, 0, 0</td>
<td>(f(-2)\approx0.12)</td>
<td>(f(-6)\approx0.00)</td>
<td>(f(-1.53)\approx0.18)</td>
</tr>
<tr>
<td>1</td>
<td>1, 0, 1</td>
<td>(f(2)\approx0.88)</td>
<td>(f(-2)\approx0.12)</td>
<td>(f(1.05)\approx0.74)</td>
</tr>
<tr>
<td>1</td>
<td>1, 1, 0</td>
<td>(f(2)\approx0.88)</td>
<td>(f(-2)\approx0.12)</td>
<td>(f(1.05)\approx0.74)</td>
</tr>
<tr>
<td>0</td>
<td>1, 1, 1</td>
<td>(f(6)\approx1.00)</td>
<td>(f(2)\approx0.88)</td>
<td>(f(-1.53)\approx0.18)</td>
</tr>
</tbody>
</table>

For example for \(X_0, X_1, X_2 = 1, 0, 1\), then
\[\begin{align*}
H_1 &= f(-2 * 1 + 4 * 0 + 4 * 1) = f(2) \approx 0.88 \\
H_2 &= f(-6 * 1 + 4 * 0 + 4 * 1) = f(-2) \approx 0.12 \\
O &= f(-2 * 4 * 0.88 - 4 * 0.12) = f(-1.05) \approx 0.74
\end{align*}\]

0.74 is closer to 1 than 0, so predict 1, which = \(Y\).

---

Learning by Backpropagation

Procedure \(\text{NNLearn}(e, H, O, h, o, \eta)\)

Input: \(e = (x, y)\): inputs and output of example \(e\)

- \(H\): hidden units with activation function \(f_H\) and weights \(w_H\) for each hidden unit \(H_j\)
- \(O\): output unit with activation function \(f_O\) and weights \(w_O\)
- \(h, o\): outputs of hidden and output units
- \(\eta\): learning rate

\[
\delta_O \leftarrow \text{derivative of } w_O \cdot h \text{ wrt error} \\
w_O \leftarrow w_O - \eta \delta_O \cdot h
\]

for each hidden unit \(H_j\)

\[
\delta_{H_j} \leftarrow \delta_O \ast \text{derivative of } w_{H_j} \cdot x \text{ wrt } w_O \cdot h \\
w_{H_j} \leftarrow w_{H_j} - \eta \delta_{H_j} \cdot x
\]

Comments on Backpropagation

- If squared error and \(f(z) = 1/(1 + e^{-z})\)
  
  \[
  \delta_O \leftarrow \alpha (1-o)(o-y) \\
  \delta_{H_j} \leftarrow h_j (1-h_j)(w_{O,j}) \delta_O
  \]

- \(\text{NNLearn}\) needs to be applied for many epochs (passes over the examples).
- The weights need to be initialized to small random values.
- Multiple runs might be needed to find good weights (and hyperparameters such as number of hidden units).
- More sophisticated gradient descent algorithms are much faster than backpropagation.
Support Vector Machines

SVMs

- A support vector machine (SVM) assigns a weight \( \alpha \) to each training example \((x_i, y_i)\) (\(x_i\) is a vector of the values of the input features, and \(y_i\) is either \(-1\) or \(1\)).
- A [simplified] SVM computes a value on \(x\) by summing over all examples:
  \[
  h(x) = \sum_i \alpha_i y_i k(x, x_i)
  \]
  where \(k\) is a kernel function.
- Learning is by optimizing the error function:
  \[
  \text{minimize } \frac{\|h\|^2}{2} + C \sum_i \max(0, 1 - y_i h(x_i))
  \]
  subject to \(0 \leq \alpha_i \leq C\) where \(\|h\|\) is the size of \(h\) in kernel space.

Kernel Functions

- The effect of a kernel function is to expand the number of features for linear classification.
- E.g., the kernel function \(k(x_i, x_j) = (x_i \cdot x_j)^2\) corresponds to a dot product of all the quadratic combinations of input features. If \(x_i = (1, a, b)\) and \(x_j = (1, c, d)\), then
  \[
  (x_i \cdot x_j)^2 = (1 + ac + bd)^2
  = 1 + 2ac + 2bd + a^2c^2 + 2abcd + b^2d^2
  = (1, \sqrt{2a}, \sqrt{2b}, a^2, \sqrt{2ab}, b^2) \cdot (1, \sqrt{2c}, \sqrt{2d}, c^2, \sqrt{2cd}, d^2)
  \]
- The gaussian kernel \(k(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2)\) is a common choice. \([\gamma\] is an extra parameter to choose.\]

Slow SVM Learning Algorithm

Procedure \(\text{SVMLearner}(X, Y, E, k, C)\)

Inputs \(X, Y\): input features and target feature
\(E\): set of training examples
\(k\): kernel function
\(C\): “soft margin” constant
initialize all example weights \(\alpha_i\) to zero
repeat until termination
  for each example \(e = (x_i, y_i) \in E\)
    \(\hat{y} \leftarrow \sum_j \alpha_j y_j k(x_i, x_j)\)
    if \(y_i \hat{y} < 1\) and \(\alpha_i < C\) increase \(\alpha_i\)
    if \(y_i \hat{y} > 1\) and \(\alpha_i > 0\) decrease \(\alpha_i\)
return \(\alpha\)

Comments on SVMs

- Each example with \(\alpha_i > 0\) is called a support vector.
- The solution will have \(\alpha_i = 0\) if \(y_i \hat{y} > 1\) and \(\alpha_i = C\) if \(y_i \hat{y} < 1\) with inbetween values for \(y_i \hat{y} = 1\).
- An SVM has a global minimum with no local minima, though actual algorithms only get very close.
- Multiple runs might be needed to find the best kernel function and soft margin constant.
- More sophisticated SVM algorithms are much faster.
The Nearest Neighbor Algorithm

- The $k$-nearest neighbor algorithm classifies a test ex. by finding the $k$ closest training exs., returning the most common class.
- Suppose 10% noise (best possible test error is 10%). With enough training exs., a test ex. will agree with its nearest neighbor with prob. $(.9)^2 + (.1)^2 = .82$ (both not noisy or both noisy) and disagree with prob. $.9*.1 + .1*.9 = .18$.
- $1$-NN converges to less than twice optimal error ($3$-NN to less than 32% higher).
- Problems: distance measure, number of exs. needed, find NNs efficiently.

Ensemble Learning

- There are many algorithms for learning a single hypothesis.
- Ensemble learning will learn many hypotheses: run different algorithms or run the same algorithm on different training sets.
- Bagging runs a learning algorithm on repeated subsamples of the training set.
- If there are $n$ examples, then a subsample of $n$ examples is generated by sampling with replacement.
- On a test example, each hypothesis casts 1 vote for the class it predicts.

Boosting

- In boosting, the hypotheses are learned in sequence.
- Both hypotheses and examples have weights with different purposes.
- After each hypothesis is learned, its weight is based on its error rate, and the weights of the training examples (initially all equal) are also modified.
- On a test example, when each hypothesis predicts a class, its weight is the size of its vote. The ensemble predicts the class with the highest vote.

Example Boosting Algorithm

Procedure $AdaBoost(E, A, t)$
Inputs $E, A$: examples and learning algorithm
$n$: number of hypotheses to generate
initialize example weights $w$ to $1/\text{number of exs.}$
for $i$ from 1 to $n$
  $h[i] \leftarrow A(E, w)$
  $\epsilon \leftarrow \text{sum weights of exs. missed by } h[i]$
for $j$ from 1 to number of examples
  if $h[i]$ is correct on example $j$
    then $w[j] \leftarrow 0.5 \cdot w[j] / (1 - \epsilon)$
    else $w[j] \leftarrow 0.5 \cdot w[j] / \epsilon$
  weight of $h[i] \leftarrow \log((1 - \epsilon) / \epsilon)$
return $h[1 \ldots n]$ and their weights

Example Run of AdaBoost

Using the 14 examples as a training set:
- Example weights are initialized to $1/14$.
- The hypothesis windy = false ↔ class = pos is wrong on 5 of the 14 examples.
- The weights of the correctly classified examples are multiplied by $14/18$, and incorrectly classified by $14/10$. An example will have a weight of $1/18$ or $1/10$.
- This hypothesis has a weight of $\log(9/5)$.
- Note that after weight updating, the sum of the correctly classified examples equals the sum of the incorrectly classified examples.
**Example Run of AdaBoost, Continued**

- The next hypothesis must be different from the previous one to have error less than \( \frac{1}{2} \).
- Now the hypothesis outlook = overcast \( \leftrightarrow \) class = pos has an error rate of 29/90.
- The weights of the correctly classified examples are multiplied times 90/122, and the incorrectly classified times 90/58.
- This hypothesis has a weight of \( \log\left(\frac{61}{29}\right) \).

**Overfitting**

- **Overfitting** is when the learner uses regularities that appear in the training set, but do not appear in the test set.
- The learning algorithm might fit noise in the training data (outliers, random fluctuations).
- The learning algorithm might fit to unique values in the training data (id numbers).
- The learning algorithm might “memorize” the training examples (more parameters than examples).
- This can be observed when running the algorithm longer does better on the training set, but does worse on the test set.

**Validation Set**

- A validation set is one way to avoid overfitting.
- The original data is divided into a training set, a validation set and a test set (more about test set later).
- As the learning algorithm runs on the training set, keep track of its error on the validation set. Use the algorithm settings that minimize validation error.

**k-Fold Cross-Validation**

- In \( k \)-fold cross-validation, the data is divided into a training set and a test set.
- The training set is evenly divided into \( k \) folds.
- Then \( k \) times: Train the learning algorithm on \( k-1 \) folds and validate on the remaining fold.
- Average over the results.
Regularization

- Regularization is a way to avoid overfitting.
- A learning model might have many parameters.
  - A NN with 10 hidden units has over 10 times more parameters than a linear model.
  - In natural language tasks, a parameter for every word (millions of words) or every word-word combination.
- If a linear model has as many or more parameters than examples, then the examples can be “memorized”.
- Regularization modifies the error function so there is a penalty for larger parameters.

Regularization Example

- Suppose we want to minimize hinge loss over a set of examples. That is, find \( w \) minimizing:
  \[
  \sum_i \max(0, 1 - y_i (w \cdot x_i))
  \]
- Perhaps this formulation leads to overfitting. To regularize this, we can add a term that penalizes large weights.
  \[
  \lambda (w \cdot w)/2 + \sum_i \max(0, 1 - y_i (w \cdot x_i))
  \]
- The weight update in the LinearLearn algorithm becomes:
  \[
  w \leftarrow w (1 - \eta \lambda/m) + \eta \delta x
  \]
  where \( m \) is the number of examples.
- Use validation to find \( \lambda \).

How to Use a Test Set

- Recall: the original data is divided into a training set, a validation set and a test set.
- Do not look at the test set.
- Do all the processing and analysis you want using the training and validation sets.
- Do not look at the test set.
- Select a hypothesis (or a few hypotheses) by training on the training and validation sets combined, using the best settings on the best algorithm(s).
- Evaluate hypothesis (or hypotheses) on the test set.

Can I Use the Test Set Again?

- In theory, no. By reusing the test set, you are in effect using the test set for training (searching for an algorithm that does well on the test set).
- In practice, a test set can be reused once or twice without getting into too much trouble, probably, maybe.
- Using many different datasets (and so many different test sets) from a variety of domains can be used to empirically compare algorithms.
- Moral: Prepare a second test set (maybe a third or fourth) so you have one in reserve after a possible initial failure.

Error Rate and Comparing Algorithms

- What is the true error rate?
  - If \( \epsilon \) is the test error rate, and \( m \) examples in the test set (\( m > 100 \)), then a 95% confidence interval is:
  \[
  \epsilon \pm 1.96 \sqrt{\frac{\epsilon (1 - \epsilon)}{m - 1}}
  \]
- Is algorithm A better than algorithm B?
  - Use the paired-difference test (next page assuming more than 100 examples in test set).
Paired-Difference Test

Let $h_1$ and $h_2$ be the two hypotheses to compare.

For each $(x_i, y_i) \in E$:

let $d_i = \begin{cases} 
1 & \text{if } h_1(x_i) = y_i \neq h_2(x_i) \\
0 & \text{if } h_1(x_i) = h_2(x_i) \\
-1 & \text{if } h_1(x_i) \neq y_i = h_2(x_i) 
\end{cases}$

Calculate:

\[ u = \frac{\sum_{i=1}^{m} d_i}{m} \quad s^2 = \frac{\sum_{i=1}^{m} (d_i - u)^2}{m - 1} \quad z = \frac{u}{s/\sqrt{m}} \]

If $z \geq 1.96$, answer "$h_1$ is better than $h_2$".
If $z \leq -1.96$, answer "$h_1$ is worse than $h_2$".

The significance is 0.05.

If $|z| < 1.96$, then answer "I don't know".