Neural Networks
Module 2: learning with Gradient Descent

**Module 2: numerical optimization**

- formulate problem by model/parameters
- formulate error as mathematical objective
- optimize numerically the parameters for the given objective
- usually algebraic setup
  - involves matrices and calculus
- probabilistic setup (likelihoods) next module
Module 2 Objectives / Neural Networks

- perceptron rules
- neural network idea, philosophy, construction
- NN weights
- Backpropagation: training NN using gradient descent
- NN modes, autoencoders
- run NN-autoencoder on a simple problem
The perceptron

\[ h_w(x) = xw = \sum_{d=0}^{D} x^d w^d \]
The perceptron

• (like with regression) we are looking for a linear classifier

\[ h_\mathbf{w}(\mathbf{x}) = \mathbf{x} \mathbf{w} = \sum_{d=0}^{D} x^d w^d \]

• error different than regression: weighted sum over misclassified points set \( M \)

\[ J(\mathbf{w}) = \sum_{x \in M} -h_\mathbf{w}(\mathbf{x}) = \sum_{x \in M} -x \mathbf{w} \]
• perceptron is a linear (hyperplane) separator
• for simplicity, will transform data points with $y=-1$ (left) to $y=1$ (right) by reversing the sign
The perceptron

• To optimize for perceptron error, use gradient descent
  \[ \nabla_w J(w) = \sum_{x \in M} -x^T \]

• with update rule
  \[ w := w + \lambda \sum_{x \in M} x^T \]

• batch update:
  1. init \( w \)
  2. LOOP
  3. get \( M = \) set of misclassified data points
  4. \( w = w + \lambda \sum_{x \in M} x^T \)
  5. UNTIL \( |\lambda \sum_{x \in M} x| < \epsilon \)
• perceptron update: the plane (dotted red) normal $w$ (red arrow) moves in the direction of misclassified $p_1$ until $p_1$ is on the correct side.
Perceptron proof of convergence

- if data is indeed linearly separable, the perceptron will find the separator line.

**Proof of perceptron convergence** Assuming data is linearly separable, or there is a solution $\tilde{w}$ such that $x\tilde{w} > 0$ for all $x$. Let's call $w_k$ the $w$ obtained at the $k$-th iteration (update). Fix an $\alpha > 0$. Then

$$w_{k+1} - \alpha \tilde{w} = (w_k - \alpha \tilde{w}) + x_k^T$$

where $x_k$ is the datapoint that updated $w$ at iteration $k$. Then

$$||w_{k+1} - \alpha \tilde{w}||^2 = ||w_k - \alpha \tilde{w}||^2 + 2x_k(w_k - \alpha \tilde{w}) + ||x_k||^2 \leq ||w_k - \alpha \tilde{w}||^2 - 2x_k\alpha \tilde{w} + ||x_k||^2$$

Since $x_k\tilde{w} > 0$ all we need is an $\alpha$ sufficiently large to show that this update process cannot go on forever. When it stops, all datapoints must be classified correctly.
Multilayer perceptrons
• build/explain a 3-layer perceptron that give the same classification as the logical XOR function

\[ XOR(x, y) = OR(x, y) \ AND \ (NOT(AND(x, y))) \]

• your answer is required! Submit via dropbox.
Neural Networks

- NN is a stack of connected perceptrons
- bottom up:
  - input layer
  - hidden layer
  - output layer
- multilayer NN very very powerful in that they can approximate almost any function
  - with enough training data
Neural Networks

- Each unit performs first a linear combination of inputs
  \[ net_j = \sum_{i=1}^{d} x_i w_{ji} + w_{j0} = \sum_{i=0}^{d} x_i w_{ji} = \text{w}_j^t x \]

- Then applies a nonlinear (ex. logistic) function “f” before outputting a value
  \[ y_j = f(\text{net}_j) \]

- Three layer NN output can be expressed mathematically as
  \[ g_k(x) = z_k = f \left( \sum_j w_{kj} f \left( \sum_i w_{ij} x_i + w_{j0} \right) + w_{k0} \right) \]
Training the NN weights ($w$)

- one datapoint

$$J(w) = \frac{1}{2} \sum_k (t_k - z_k)^2$$

$$\Delta w_{pq} = -\lambda \frac{\partial J}{\partial w_{pq}},$$

- set of weights up (close to output):

$$\frac{\partial J}{\partial w_{kj}} = \frac{\partial J}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial w_{kj}} = -\delta_k \frac{\partial \text{net}_k}{\partial w_{kj}}$$

$$\delta_k = -\frac{\partial J}{\partial \text{net}_k} = -\frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial \text{net}_k} = (t_k - z_k)f'(\text{net}_k)$$

$$\frac{\partial \text{net}_k}{\partial w_{kj}} = y_j$$

- we obtain the hidden-output weight update rule

$$w_{kj} = w_{kj} + \lambda(t_k - z_k)f'(\text{net}_k)y_j$$
Training the NN weights \( w \)

- weight first set of weights (close to input)

\[
\frac{\partial J}{\partial w_{ji}} = \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ji}}
\]

\[
\frac{\partial J}{\partial y_j} = - \sum_k (t_k - z_k) \frac{\partial z_k}{\partial \text{net}_k} \frac{\partial \text{net}_k}{\partial y_j} - \frac{1}{2} \sum_k (t_k - z_k)^2
\]

\[
\frac{\partial h_j}{\partial \text{net}_j} = f'(\text{net}_j)
\]

\[
\frac{\partial \text{net}_j}{\partial w_{ji}} = x_i
\]

\[
w_{ji} \leftarrow w_{ji} - \lambda \left[ \sum_k (t_k - z_k) f'(\text{net}_k) w_{kj} f'(\text{net}_j) x_i \right]
\]
**NN training**

**STOCHASTIC TRAINING**
Select $x_t$ (randomly chosen)

$$w_{ij} = w_{ij} + \lambda \delta_j x_i$$
$$w_{jk} = w_{jk} + \lambda \delta_k y_j$$

until $| \nabla_w J | < \epsilon$

**BATCH TRAINING**

for each iteration:

for each $x_t$

$$\delta w_{ij} = \delta w_{ij} + \lambda \delta_j x_i$$
$$\delta w_{jk} = \delta w_{jk} + \lambda \delta_k y_j$$

$$w_{ij} \leftarrow w_{ij} + \delta w_{ij}$$
$$w_{jk} \leftarrow w_{jk} + \delta w_{jk}$$

until $|| \nabla_w J || < \epsilon$
Autoencoders

- network is “rotated”
  - from left to right: input-hidden-output
- input and output are the same values
  - hidden layer encodes the input and decodes back to itself

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden Values</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000000</td>
<td>→ .89 .04 .08</td>
<td>→ 100000000</td>
</tr>
<tr>
<td>010000000</td>
<td>→ .15 .99 .99</td>
<td>→ 010000000</td>
</tr>
<tr>
<td>001000000</td>
<td>→ .01 .97 .27</td>
<td>→ 001000000</td>
</tr>
<tr>
<td>000100000</td>
<td>→ .99 .97 .71</td>
<td>→ 000100000</td>
</tr>
<tr>
<td>000010000</td>
<td>→ .03 .05 .02</td>
<td>→ 000010000</td>
</tr>
<tr>
<td>000001000</td>
<td>→ .01 .11 .88</td>
<td>→ 000001000</td>
</tr>
<tr>
<td>000000100</td>
<td>→ .80 .01 .98</td>
<td>→ 000000100</td>
</tr>
<tr>
<td>000000010</td>
<td>→ .60 .94 .01</td>
<td>→ 000000010</td>
</tr>
</tbody>
</table>
BackPropagation (Tom Mitchell book)

\textbf{BACKPROPAGATION(} training\textunderscore examples, \eta, n_{in}, n_{out}, n_{hidden}\textbf{)}

Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$, where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.

$\eta$ is the learning rate (e.g., .05). $n_{in}$ is the number of network inputs, $n_{hidden}$ the number of units in the hidden layer, and $n_{out}$ the number of output units.

The input from unit $i$ into unit $j$ is denoted $x_{ji}$, and the weight from unit $i$ to unit $j$ is denoted $w_{ji}$.

- Create a feed-forward network with $n_{in}$ inputs, $n_{hidden}$ hidden units, and $n_{out}$ output units.
- Initialize all network weights to small random numbers (e.g., between $-.05$ and $.05$).
- Until the termination condition is met, Do
  - For each $\langle \vec{x}, \vec{t} \rangle$ in training\textunderscore examples, Do

\textit{Propagate the input forward through the network:}

1. Input the instance $\vec{x}$ to the network and compute the output $o_u$ of every unit $u$ in the network.

\textit{Propagate the errors backward through the network:}

2. For each network output unit $k$, calculate its error term $\delta_k$

   \[ \delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \]  \hspace{1cm} (T4.3)

3. For each hidden unit $h$, calculate its error term $\delta_h$

   \[ \delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{kh}\delta_k \]  \hspace{1cm} (T4.4)

4. Update each network weight $w_{ji}$

   \[ w_{ji} \leftarrow w_{ji} + \Delta w_{ji} \]

   where

   \[ \Delta w_{ji} = \eta \delta_j x_{ji} \]  \hspace{1cm} (T4.5)