Lecture 9 – Perceptrons

Machine Learning
Queens College
Today

• Perceptrons
• Leading to
  – Neural Networks
  – aka Multilayer Perceptron Networks
  – But more accurately: Multilayer Logistic Regression Networks
Review: Fitting Polynomial Functions

- Fitting nonlinear 1-D functions

- Polynomial: 
  \[ f(x, \bar{w}) = \sum_{d=1}^{D} w_d x^d + w_0 \quad f(x, \theta) = \sum_{d=1}^{D} \theta_d x^d + \theta_0 \]

- Risk:
  \[ R_{\text{emp}}(\bar{w}) = \frac{1}{2N} \sum_{i=0}^{N-1} \left( t_i - \begin{bmatrix} 1 & x_i \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \right)^2 \]

  \[ R(\theta) = \frac{1}{2N} \left\| \begin{bmatrix} t_0 \\ t_1 \\ \vdots \\ t_{N-1} \end{bmatrix} - \begin{bmatrix} 1 & x_0 \\ 1 & x_1 \\ \vdots \\ 1 & x_{N-1} \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \end{bmatrix} \right\|^2 \]

  \[ = \frac{1}{2N} \left\| \tilde{t} - \tilde{X} \bar{w} \right\|^2 \]
Review: Fitting Polynomial Functions

- Order-D polynomial regression for 1D variables is the same as D-dimensional linear regression.
- Extend the feature vector from a scalar.
  \[ \vec{x}_i = [x_i^0 \ x_i^1 \ x_i^2 \ \ldots \ x_i^D]^T \]
- More generally
  \[ \vec{x}_i = [\phi_0(x_i) \ \phi_1(x_i) \ \phi_2(x_i) \ \ldots \ \phi_D(x_i)]^T \]
Neuron inspires Regression

• Graphical Representation of linear regression
  – McCullough-Pitts Neuron
  – Note: Not a graphical model
Neuron inspires Regression

• Edges multiply the signal \((x_i)\) by some weight \((\theta_i)\).
• Nodes sum inputs
• Equivalent to Linear Regression

\[
f(x, \theta) = \sum_{d=1}^{D} \theta_d x^d + \theta_0
\]
Introducing Basis functions

• Graphical representation of feature extraction
  \[ f(x, \theta) = \sum_{d=1}^{D} \theta_d \phi_d(x) + \theta_0 \]

• Edges multiply the signal by a weight.
• Nodes apply a function, \( \phi_d \).
Extension to more features

• Graphical representation of feature extraction

\[ f(\vec{x}, \theta) = \sum_{n=0}^{N-1} \sum_{d=1}^{D} \theta_d \phi_d(x_n) + \theta_0 \]

\[ f(\vec{x}, \theta) = \begin{array}{c}
    x_0 \\
    x_1 \\
    x_2 \\
    \vdots \\
    x_D \\
\end{array} \]

\[ \begin{array}{c}
    \phi_1 \\
    \phi_2 \\
    \phi_D \\
\end{array} \]

\[ \begin{array}{c}
    \theta_1 \\
    \theta_2 \\
    \theta_D \\
    \theta_0 \\
\end{array} \]

\[ \begin{array}{c}
    1 \\
\end{array} \]

\[ f(\vec{x}, \theta) \]
Combining function

• How do we construct the neural output

\[ f(\vec{x}, \theta) = \sum_{n=0}^{N-1} \sum_{d=1}^{D} \theta_d \phi_d(x_n) + \theta_0 \]

\[ f(\vec{x}, \theta) = \theta^T \vec{x} \]
Combining function

- Sigmoid function or Squashing function

\[ f(\vec{x}, \theta) = g(\theta^T \vec{x}) \quad g(z) = (1 + \exp(-z))^{-1} \]
Logistic Neuron optimization

• Minimizing $R(\theta)$ is more difficult

\[
R(\theta) = \frac{1}{2N} \sum_{i=0}^{N-1} (t_i - g(\theta^T x_i))^2
\]

\[
\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0
\]

\[
g(z) = (1 + \exp(-z))^{-1} \quad g'(z) = g(z)(1 - g(z))
\]

Bad News: There is no “closed-form” solution.
Gradient Descent

• The Gradient is defined (though we can’t solve directly)

\[ \nabla_{\theta} R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0 \]

• Points in the direction of fastest increase
Gradient Descent

• Gradient points in the direction of fastest increase

\[ \nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0 \]

• To minimize R, move in the opposite direction
Gradient Descent

• Gradient points in the direction of fastest increase
  \[ \nabla_{\theta} R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0 \]

• To minimize R, move in the opposite direction
Gradient Descent

- Initialize Randomly \( \theta_0 = \text{random} \)
- Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \)
- (nearly) guaranteed to converge to the minimum

\[

\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0

\]

\( -\nabla_\theta R \)
Gradient Descent

• Initialize Randomly $\theta_0 = \text{random}$
• Update with small steps $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
• (nearly) guaranteed to converge to the minimum

$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0$$
Gradient Descent

- Initialize Randomly \( \theta_0 = \text{random} \)
- Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \)
- (nearly) guaranteed to converge to the minimum

\[
\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0
\]
Gradient Descent

- Initialize Randomly  \( \theta_0 = \text{random} \)
- Update with small steps  \( \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \)
- (nearly) guaranteed to converge to the minimum

\[
\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i \Rightarrow 0
\]
Gradient Descent

• Initialize Randomly \( \theta_0 = \text{random} \)
• Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \)
• (nearly) guaranteed to converge to the minimum

\[
\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0
\]
Gradient Descent

- Initialize Randomly: $\theta_0 = \text{random}$
- Update with small steps: $\theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t}$
- (nearly) guaranteed to converge to the minimum

$$\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0$$
Gradient Descent

- Initialize Randomly
  \[ \theta_0 = \text{random} \]
- Update with small steps
  \[ \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \]
- (nearly) guaranteed to converge to the minimum

\[
\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i \approx 0
\]
Gradient Descent

- Initialize Randomly \( \theta_0 = \text{random} \)
- Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \)
- Can oscillate if \( \eta \) is too large

\[
\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i \geq 0
\]
Gradient Descent

- Initialize Randomly \( \theta_0 = \text{random} \)
- Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \)
- Can oscillate if \( \eta \) is too large

\[
\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'((\theta^T x_i)x_i) = 0
\]
Gradient Descent

- Initialize Randomly \( \theta_0 = \text{random} \)
- Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \)
- Can oscillate if \( \eta \) is too large

\[
\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i \xrightarrow{\eta} 0
\]
Gradient Descent

- Initialize Randomly \( \theta_0 = \text{random} \)
- Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_{\theta} R|_{\theta_t} \)
- Can oscillate if \( \eta \) is too large

\[
\nabla_{\theta} R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0
\]
Gradient Descent

• Initialize Randomly \( \theta_0 = \text{random} \)
• Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_{\theta} R|_{\theta_t} \)
• Can oscillate if \( \eta \) is too large

\[
\nabla_{\theta} R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i \geq 0
\]
Gradient Descent

- Initialize Randomly \( \theta_0 = \text{random} \)
- Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \)
- Can oscillate if \( \eta \) is too large

\[
\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i \\
\n\Rightarrow 0
\]
Gradient Descent

- Initialize Randomly \( \theta_0 = \text{random} \)
- Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_{\theta} R|_{\theta_t} \)
- Can oscillate if \( \eta \) is too large

\[
\nabla_{\theta} R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i \\
\text{where } x_i \geq 0
\]
Gradient Descent

- Initialize Randomly \( \theta_0 = \text{random} \)
- Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_{\theta} R|_{\theta_t} \)
- Can oscillate if \( \eta \) is too large

\[
\nabla_{\theta} R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i = 0
\]
Gradient Descent

• Initialize Randomly \( \theta_0 = \text{random} \)
• Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_\theta R|_{\theta_t} \)
• Can oscillate if \( \eta \) is too large

\[

\nabla_\theta R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i \Rightarrow 0

\]
Gradient Descent

• Initialize Randomly \( \theta_0 = \text{random} \)

• Update with small steps \( \theta_{t+1} = \theta_t - \eta \nabla_{\theta} R|_{\theta_t} \)

• Can stall if \(-\nabla_{\theta} R\) is ever 0 not at the minimum

\[
\nabla_{\theta} R = \frac{1}{2N} \sum_{i=0}^{N-1} 2(t_i - g(\theta^T x_i))(-1)g'(\theta^T x_i)x_i \Rightarrow 0
\]
Back to Neurons

Linear Neuron

Logistic Neuron

\[ f(\vec{x}, \theta) \]
Perceptron

• Classification squashing function

\[ g(z) = \begin{cases} 
-1 & \text{when } z < 0 \\
+1 & \text{when } z \geq 0
\end{cases} \]

• Strictly classification error
Classification Error

• Only count errors when a classification is incorrect, i.e., $t_i \neq g(\theta^T x_i)$.

$$R(\theta) = \frac{1}{2N} \sum_{i=0}^{N-1} (t_i - g(\theta^T x_i))^2$$

Sigmoid leads to greater than zero error on correct classifications.
Classification Error

• Only count errors when a classification is incorrect, i.e., $t_i \neq g(\theta^T x_i)$.

$$R(\theta) = \frac{1}{N} \sum_{i=0}^{N-1} \text{step}(-t_i \theta^T x_i)$$
Perceptron Error

• Can’t do gradient descent on this. $\nabla_\theta R = 0$

$R(\theta) = \frac{1}{N} \sum_{i=0}^{N-1} \text{step}(-t_i \theta^T x_i)$
Perceptron Loss

- With classification loss: \( \nabla_\theta R = 0 \)
- Define Perceptron Loss.
  - Loss calculated for each misclassified data point

\[
R(\theta) = -\frac{1}{N} \sum_{i \in \text{error}} t_i (\theta^T x_i)
\]

- Now piecewise linear risk rather than step function
Perceptron Loss

- Perceptron Loss. \( R(\theta) = -\frac{1}{N} \sum_{i \in \text{error}} t_i (\theta^T x_i) \)
  - Loss calculated for each misclassified data point
  \[
  \nabla_{\theta} R(\theta) = -\frac{1}{N} \sum_{i \in \text{error}} t_i x_i
  \]
  - Nice gradient descent
  \[
  \theta_{t+1} = \theta_t - \eta \nabla_{\theta} R(\theta) = \theta_t + \eta \frac{1}{N} \sum_{i \in \text{error}} t_i x_i
  \]
Online Perceptron Training

• Online Training
  – Update weights for each data point.

• Iterate over points $x_i$ point,
  – If $x_i$ is correctly classified $\theta_{t+1} = \theta_t$
  – Else $\theta_{t+1} = \theta_t + t_i x_i$

• Theorem: If $x_i$ in $X$ are linearly separable, then this process will converge to a $\theta^*$ which leads to zero error in a finite number of steps.
Linearly Separable

- Two classes of points are **linearly separable**, iff there exists a line such that all the points of one class fall on one side of the line, and all the points of the other class fall on the other side of the line.
Linearly Separable

• Two classes of points are **linearly separable**, iff there exists a line such that all the points of one class fall on one side of the line, and all the points of the other class fall on the other side of the line.
Perceptron as a Linear Separator

• Since perceptron uses linear threshold function, it is searching for a linear separator that discriminates the classes.

\[ \theta_1 o_2 + \theta_2 o_3 > T \]

\[ o_3 > -\frac{\theta_1}{\theta_2} o_2 + \frac{T}{\theta_2} \]

Or hyperplane in n-dimensional space
Concept Perceptron Cannot Learn

• Cannot learn **exclusive-or**, or parity function in general.
Next

• Multilayer Neural Networks