CS 188: Artificial Intelligence
Optimization and Neural Nets

Instructors: Brijen Thananjeyan and Aditya Baradwaj --- University of California, Berkeley

[These slides were created by Dan Klein, Pieter Abbeel, Sergey Levine. All CS188 materials are at http://ai.berkeley.edu.]
Logistic Regression: How to Learn?

- Maximum likelihood estimation

$$\theta_{ML} = \arg \max \theta P(X|\theta)$$

$$= \arg \max \theta \prod_i P_\theta(X_i)$$

- Maximum conditional likelihood estimation

$$\theta^* = \arg \max \theta P(Y|X, \theta)$$

$$= \arg \max \theta \prod_i P_\theta(y_i|x_i)$$

$$\ell(w) = \prod_i e^{w_{y_i} \cdot x_i}$$

$$\ell(w) = \sum_i \log P_w(y_i|x_i)$$

$$= \sum_i w_{y_i} \cdot x_i - \log \sum_y e^{w_y \cdot x_i}$$
Best $w$?

- Maximum likelihood estimation:

$$\max_w \ ll(w) = \max_w \ \sum_i \ \log P(y^{(i)}|x^{(i)}; w)$$

with:

$$P(y^{(i)}|x^{(i)}; w) = \frac{e^{w_{y^{(i)}} \cdot f(x^{(i)})}}{\sum_y e^{w_y \cdot f(x^{(i)})}}$$

= Multi-Class Logistic Regression
Hill Climbing

- Recall from CSPs lecture: simple, general idea
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

- What’s particularly tricky when hill-climbing for multiclass logistic regression?
  - Optimization over a continuous space
    - Infinitely many neighbors!
  - How to do this efficiently?
1-D Optimization

- Could evaluate $g(w_0 + h)$ and $g(w_0 - h)$
  - Then step in best direction

- Or, evaluate derivative:
  \[
  \frac{\partial g(w_0)}{\partial w} = \lim_{h \to 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}
  \]
  - Tells which direction to step into
2-D Optimization

Source: offconvex.org
Gradient Ascent

- Perform update in uphill direction for each coordinate
- The steeper the slope (i.e. the higher the derivative) the bigger the step for that coordinate
- E.g., consider: 

\[ g(w_1, w_2) \]

- Updates:

\[
\begin{align*}
  w_1 &\leftarrow w_1 + \alpha \cdot \frac{\partial g}{\partial w_1}(w_1, w_2) \\
  w_2 &\leftarrow w_2 + \alpha \cdot \frac{\partial g}{\partial w_2}(w_1, w_2)
\end{align*}
\]

- Updates in vector notation:

\[
\begin{align*}
  w &\leftarrow w + \alpha \cdot \nabla_w g(w) \\
  \text{with: } \nabla_w g(w) &= \begin{bmatrix} \frac{\partial g}{\partial w_1}(w) \\ \frac{\partial g}{\partial w_2}(w) \end{bmatrix} = \text{gradient}
\end{align*}
\]
Gradient Ascent

- Idea:
  - Start somewhere
  - Repeat: Take a step in the gradient direction
What is the Steepest Direction?

First-Order Taylor Expansion:

Steepest Descent Direction:

Recall:

Hence, solution:

Gradient direction = steepest direction!
\[
\n\begin{align*}
\frac{\partial g}{\partial w_1} &= w_2 \\
\frac{\partial g}{\partial w_2} &= w_1 \\
\frac{\partial g}{\partial w_3} &= 1
\end{align*}
\]

\[\nabla g = \begin{bmatrix}
\frac{\partial g}{\partial w_1} \\
\frac{\partial g}{\partial w_2} \\
\vdots \\
\frac{\partial g}{\partial w_n}
\end{bmatrix}\]

\[g(w_1, w_2, w_3) = w_1w_2 + w_3\]

\[\nabla g(w) = \begin{bmatrix} w_2 \\ w_1 \\ 1 \end{bmatrix}\]
Optimization Procedure: Gradient Ascent

- **init** $w$
- **for** iter = 1, 2, ...

$$w \leftarrow w + \alpha \cdot \nabla g(w)$$

- $\alpha$: learning rate --- tweaking parameter that needs to be chosen carefully
- How? Try multiple choices
  - Crude rule of thumb: update changes $w$ about 0.1 – 1%
Batch Gradient Ascent on the Log Likelihood Objective

\[
\max_w \ ll(w) = \max_w \ \sum_i \log P(y^{(i)}|x^{(i)}; w)
\]

\[
g(w)
\]

- init \( w \)
- for \( \text{iter} = 1, 2, ... \)
  \[
w \leftarrow w + \alpha \sum_i \nabla \log P(y^{(i)}|x^{(i)}; w)
\]
Stochastic Gradient Ascent on the Log Likelihood Objective

\[
\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)
\]

**Observation:** once gradient on one training example has been computed, might as well incorporate before computing next one

- **init** \( w \)
- **for** iter = 1, 2, ...
  - **pick** random \( j \)

\[
w \leftarrow w + \alpha \ast \nabla \log P(y^{(j)} | x^{(j)}; w)
\]
Mini-Batch Gradient Ascent on the Log Likelihood Objective

\[
\max_w ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)
\]

**Observation:** gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one.

- **init** \( w \)
- **for** iter = 1, 2, ...
  - pick random subset of training examples \( J \)
  \[
  w \leftarrow w + \alpha \sum_{j \in J} \nabla \log P(y^{(j)}|x^{(j)}; w)
  \]
Gradient for Logistic Regression

\[ \ell(w) = \sum_i \log P_w(y_i|x_i) \]
\[ = \sum_i w_{y_i} \cdot f(x_i) - \log \sum_y e^{w_y \cdot f(x_i)} \]

\[ \frac{d}{dw_y} \log P_w(y_i|x_i) = \begin{cases} 
  f(x_i) - f(x_i) \frac{e^{w_{y'_i} \cdot f(x_i)}}{\sum_{y'} e^{w_{y'_i} \cdot f(x_i)}} & \text{if } y = y_i \\
  -f(x_i) \frac{e^{w_{y'_i} \cdot f(x_i)}}{\sum_{y'} e^{w_{y'_i} \cdot f(x_i)}} & \text{otherwise} 
\end{cases} \]

\[ = f(x_i)(I(y = y_i) - P(y|x_i)) \]

\[ 1(y = y_i) = \begin{cases} 
  1 & \text{if } y = y_i \\
  0 & \text{otherwise} 
\end{cases} \]

- Recall perceptron:
  - Classify with current weights
  - Exercise

\[ y = \begin{cases} 
  +1 & \text{if } w \cdot f(x) \geq 0 \\
  -1 & \text{if } w \cdot f(x) < 0 
\end{cases} \]

- If correct (i.e., \( y = y^* \)), no change!
- If wrong: adjust the weight vector by adding or subtracting the feature vector. Subtract if \( y^* \) is -1.

\[ w = w + y^* \cdot f \]
Neural Networks
Multi-class Logistic Regression

- = special case of neural network

\[ z_i = \omega_i^T f(x) \]

\[ P(y_1|x; w) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}} \]

\[ P(y_2|x; w) = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}} \]

\[ P(y_3|x; w) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}} \]
Deep Neural Network = Also learn the features!

will learn these features

\[
\begin{align*}
P(y_1 | x; w) &= \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}} \\
P(y_2 | x; w) &= \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}} \\
P(y_3 | x; w) &= \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}
\end{align*}
\]
Deep Neural Network = Also learn the features!

\[ f_1(x) \]
\[ f_2(x) \]
\[ f_3(x) \]
\[ \cdots \]

\[ g(\mathbf{w}_i^T \mathbf{x}) \]

\[ \text{softmax} \]

\[ P(y_1 | x; w) \]
\[ P(y_2 | x; w) \]
\[ P(y_3 | x; w) \]

\[ g = \text{nonlinear activation function} \]
Deep Neural Network = Also learn the features!

\[
z_i^{(k)} = g\left(\sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)}\right)
\]

\(g = \text{nonlinear activation function}\)
Common Activation Functions

Sigmoid Function

\[ g(z) = \frac{1}{1 + e^{-z}} \]

\[ g'(z) = g(z)(1 - g(z)) \]

Hyperbolic Tangent

\[ g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \]

\[ g'(z) = 1 - g(z)^2 \]

Rectified Linear Unit (ReLU)

Most popular

\[ g(z) = \max(0, z) \]

\[ g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases} \]

\[ y(z) = \begin{cases} z, & z > 0 \\ 0, & \text{otherwise} \end{cases} \]

Training the deep neural network is just like logistic regression:

$$\max_w \ ll(w) = \max_w \ \sum_i \ \log P(y^{(i)} | x^{(i)}; w)$$

just \( w \) tends to be a much, much larger vector 😊

→ just run gradient ascent
+ stop when log likelihood of hold-out data starts to decrease
Neural Networks Properties

- **Theorem (Universal Function Approximators).** A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

- **Practical considerations**
  - Can be seen as learning the features
  - Large number of neurons
    - Danger for overfitting
    - (hence early stopping!)
Neural Net Demo!

https://playground.tensorflow.org/
How about computing all the derivatives?

- Derivatives tables:

\[
\begin{align*}
\frac{d}{dx}(a) &= 0 \\
\frac{d}{dx}(x) &= 1 \\
\frac{d}{dx}(au) &= a \frac{du}{dx} \\
\frac{d}{dx}(u + v - w) &= \frac{du}{dx} + \frac{dv}{dx} - \frac{dw}{dx} \\
\frac{d}{dx}(uv) &= u \frac{dv}{dx} + v \frac{du}{dx} \\
\frac{d}{dx} \left( \frac{u}{v} \right) &= \frac{v \frac{du}{dx} - u \frac{dv}{dx}}{v^2} \\
\frac{d}{dx} \left( u^n \right) &= nu^{n-1} \frac{du}{dx} \\
\frac{d}{dx} \left( \sqrt{u} \right) &= \frac{1}{2\sqrt{u}} \frac{du}{dx} \\
\frac{d}{dx} \left( \frac{1}{u} \right) &= -\frac{1}{u^2} \frac{du}{dx} \\
\frac{d}{dx} \left( \frac{1}{u^2} \right) &= -\frac{n}{u^{n+1}} \frac{du}{dx} \\
\frac{d}{dx} \left( f(u) \right) &= \frac{df(u)}{du} \frac{du}{dx} \\
\frac{d}{dx} \left( \ln u \right) &= \frac{1}{u} \frac{du}{dx} \\
\frac{d}{dx} \left( \log_a u \right) &= \frac{1}{u} \frac{du}{dx} \\
\frac{d}{dx} \left( e^u \right) &= e^u \frac{du}{dx} \\
\frac{d}{dx} \left( a^u \right) &= a^u \ln a \frac{du}{dx} \\
\frac{d}{dx} \left( u^v \right) &= u^{v-1} \frac{dv}{dx} + v \frac{du}{dx} \\
\frac{d}{dx} \left( \sin u \right) &= \cos u \frac{du}{dx} \\
\frac{d}{dx} \left( \cos u \right) &= -\sin u \frac{du}{dx} \\
\frac{d}{dx} \left( \tan u \right) &= \sec^2 u \frac{du}{dx} \\
\frac{d}{dx} \left( \cot u \right) &= -\csc^2 u \frac{du}{dx} \\
\frac{d}{dx} \left( \sec u \right) &= \sec u \tan u \frac{du}{dx} \\
\frac{d}{dx} \left( \csc u \right) &= -\csc u \cot u \frac{du}{dx}
\end{align*}
\]

[source: http://hyperphysics.phy-astr.gsu.edu/hbase/Math/derfunc.html]
How about computing all the derivatives?

- But neural net f is never one of those?
  - No problem: CHAIN RULE:

\[
\begin{align*}
\text{If} & \quad f(x) = g(h(x)) \\
\text{Then} & \quad f'(x) = g'(h(x))h'(x)
\end{align*}
\]

→ Derivatives can be computed by following well-defined procedures
Automatic Differentiation

- **Automatic differentiation software**
  - e.g. Theano, TensorFlow, PyTorch, Chainer
  - Only need to program the function \( g(x,y,w) \)
  - Can automatically compute all derivatives w.r.t. all entries in \( w \)
  - This is typically done by caching info during forward computation pass of \( f \), and then doing a backward pass = “backpropagation”
  - Autodiff / Backpropagation can often be done at computational cost comparable to the forward pass

- Need to know this exists
- How this is done? -- outside of scope of CS188
Summary of Key Ideas

- Optimize probability of label given input
  \[ \max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w) \]

- Continuous optimization
  - Gradient ascent:
    - Compute steepest uphill direction = gradient (= just vector of partial derivatives)
    - Take step in the gradient direction
    - Repeat (until held-out data accuracy starts to drop = “early stopping”)

- Deep neural nets
  - Last layer = still logistic regression
  - Now also many more layers before this last layer
    - = computing the features
    - \( \rightarrow \) the features are learned rather than hand-designed
  - Universal function approximation theorem
    - If neural net is large enough
    - Then neural net can represent any continuous mapping from input to output with arbitrary accuracy
    - But remember: need to avoid overfitting / memorizing the training data \( \rightarrow \) early stopping!
    - Automatic differentiation gives the derivatives efficiently (how? = outside of scope of 188)