Reminder: Linear Classifiers

- Inputs are feature values
- Each feature has a weight
- Sum is the activation

activation(w, x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x)

- If the activation is:
  - Positive, output 1
  - Negative, output -1

How to get probabilistic decisions?

- Activation:  \( z = w \cdot f(x) \)
- If \( z = w \cdot f(x) \) very positive → want probability going to 1
- If \( z = w \cdot f(x) \) very negative → want probability going to 0

- Sigmoid function

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

Best w?

- Maximum likelihood estimation:

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

with:

\[
P(y^{(i)} = +1 | x^{(i)}; w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}
\]

\[
P(y^{(i)} = -1 | x^{(i)}; w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}
\]

= Logistic Regression
Multiclass Logistic Regression

- Multi-class linear classification
  - A weight vector for each class: \( W_y \)
  - Score (activation) of a class \( y \): \( w_y \cdot f(x) \)
  - Prediction w/ highest score wins: \( y = \arg \max_y w_y \cdot f(x) \)

- How to make the scores into probabilities?

  \[
  z_1, z_2, z_3 \rightarrow \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}
  \]

  (original activations) \rightarrow \textbf{softmax activations}

Best \( w \)?

- Maximum likelihood estimation:

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

  with:

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

This Lecture

- Optimization
  - i.e., how do we solve:

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

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

### 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:
    - $w_1 \leftarrow w_1 + \alpha \cdot \frac{\partial g}{\partial w_1}(w_1, w_2)$
    - $w_2 \leftarrow w_2 + \alpha \cdot \frac{\partial g}{\partial w_2}(w_1, w_2)$
- Updates in vector notation:
  - $w \leftarrow w + \alpha \cdot \nabla_w g(w)$
  - with: $\nabla_w g(w) = \begin{bmatrix} \frac{\partial g}{\partial w_1}(w) \\ \frac{\partial g}{\partial w_2}(w) \end{bmatrix}$
  - gradient

### 2-D Optimization

Source: offconvex.org

### Gradient Ascent

- Idea:
  - Start somewhere
  - Repeat: Take a step in the gradient direction

Figure source: Mathworks
What is the Steepest Direction?

\[ \max_{\Delta: \Delta_1^2 + \Delta_2^2 \leq \varepsilon} g(w + \Delta) \]

- First-Order Taylor Expansion:
  \[ g(w + \Delta) = g(w) + \frac{\partial g}{\partial w_1} \Delta_1 + \frac{\partial g}{\partial w_2} \Delta_2 \]
- Steepest Descent Direction:
  \[ \Delta = \frac{\nabla g}{\|\nabla g\|} \]
- Recall:
  \[ \max_{\Delta: \Delta_1^2 + \Delta_2^2 \leq \varepsilon} \rightarrow \Delta = \frac{\nabla g}{\|\nabla g\|} \]
- Hence, solution:
  \[ \Delta = \frac{\nabla g}{\|\nabla g\|} \]

Gradient direction = steepest direction!

Gradient in n dimensions

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

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
  - Grade rule of thumb: update changes \( w \) about 0.1 – 1%

Batch Gradient Ascent on the Log Likelihood Objective

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

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

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

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 \(j\)
      \[ w \leftarrow w + \alpha \nabla \log P(y^{(j)}|x^{(j)}; w) \]

**Mini-Batch Gradient Ascent on the Log Likelihood Objective**

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

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 \(J\)
      \[ w \leftarrow w + \alpha \sum_{j \in J} \nabla \log P(y^{(j)}|x^{(j)}; w) \]

**How about computing all the derivatives?**

- We’ll talk about that once we covered neural networks, which are a generalization of logistic regression

**Neural Networks**
Deep Neural Network = Also learn the features!

\[ f(x) = g(\sum_{j} W^{(k-1,k)}_{i,j} z^{(k-1)}_j) \]

\( g = \) nonlinear activation function

\[ z^{(k)}_i = g(\sum_{j} W^{(k-1,k)}_{i,j} z^{(k-1)}_j) \]
Common Activation Functions

- **Sigmoid Function**
  \[ g(x) = \frac{1}{1 + e^{-x}} \]
  \[ g'(x) = g(x)(1 - g(x)) \]
- **Hyperbolic Tangent**
  \[ h(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \]
  \[ h'(x) = 1 - h(x)^2 \]
- **Rectified Linear Unit (ReLU)**
  \[ g(x) = \max(0, x) \]
  \[ g'(x) = \begin{cases} 1, & x > 0 \\ 0, & \text{otherwise} \end{cases} \]

Deep Neural Network: Also Learn the Features!

- Training the deep neural network is just like logistic regression:
  \[
  \max_w \quad \ell(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)
  \]
  
  - just \( w \) tends to be a much, much larger vector \( \Theta \)
  - 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!]

Universal Function Approximation Theorem*

- **In words:** Given any continuous function \( f(x) \), if a 2-layer neural network has enough hidden units, then there is a choice of weights that allow it to closely approximate \( f(x) \).

Universal Function Approximation Theorem*

Cybenko (1989) “Approximations by superpositions of sigmoidal functions”

Hornik (1991) “Approximation Capabilities of Multilayer Feedforward Networks”

Leshno and Schocken (1991) “Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function”

Fun Neural Net Demo Site

How about computing all the derivatives?

- Derivatives tables:

How about computing all the derivatives?

- Demo-site:
  - http://playground.tensorflow.org/

- But neural net $f$ is never one of those?

- No problem: CHAIN RULE:

  $f(x) = g(h(x))$

  $f'(x) = g'(h(x))h'(x)$

  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
  \[ \max_w \ell(w) = \max_w \sum \log \mathbb{P} \left( y \mid x; w \right) \]

- Continuous optimization
  - Gradient ascent
    - Compute direction of steepest uphill direction = gradient (i.e., 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
    - \( \rightarrow \) computing the features
      - \( \rightarrow \) features are learned rather than hand-designed
    - Universal function approximation theorem
      - If neural net is large enough
        - \( \Rightarrow \) 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)

How well does it work?  

Computer Vision
Backpropagation: applications

- Perhaps the most successful and widely used learning algorithm for NNs;
- Used in a variety of domains:
  - clinical diagnosis,
  - predicting protein structure,
  - character recognition,
  - fingerprint recognition,
  - modeling residual chlorine decay in water,
  - weather forecast,
  - waveform recognition,
  - backgammon, etc.