Announcements

- **Project 5** due **Tuesday, Nov 29** at 11:59pm PT
- **Strike is underway**
- **Homework 9** deadline postponed
- **Homework 10** release and deadline postponed
- Online, emergency-only office hours
  - 6:30-8:00pm Tu/Th (Peyrin & Igor)
  - 5:00-6:30pm M/W (Peyrin)
- Everyone gets 1 discussion participation credit this week
- Lectures are online-only for duration of strike
CS 188: Artificial Intelligence

Optimization and Neural Networks

[These slides were created by Dan Klein, Pieter Abbeel, Anca Dragan for CS188 Intro to AI at UC Berkeley. All CS188 materials are available at http://ai.berkeley.edu.]
Reminder: Multiclass Logistic Regression

- Recall Multi-Class Perceptron:
  - A weight vector for each class: $w_y$
  - Score (activation) of a class $y$: $z_y = w_y \cdot f(x)$
  - Prediction highest score wins: $y = \arg \max_y w_y \cdot f(x)$

- How to make the scores into probabilities?

$$P(y \mid x ; w) = \frac{e^{z_y}}{\sum_{y'} e^{z_{y'}}} = \text{softmax}(z_1, ..., z_n)_y$$

$$\text{softmax}(z_1, ..., z_n)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$
Reminder: Best $w$ for Multi-Class Logistic Regression

- Given data pairs $x^{(i)}, y^{(i)}$ maximize log-likelihood:

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

- Use numerical optimization inspired by hill climbing
  - General problem: $\hat{w} = \arg\max_w g(w)$

- Derivative of the function $\frac{\partial g}{\partial w}$ tells us which direction to step into
1-D and 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:
\[
    w \leftarrow w + \alpha \cdot \nabla_w g(w)
\]

with: \( \nabla_w g(w) = \left[ \begin{array}{c} \frac{\partial g}{\partial w_1}(w) \\ \frac{\partial g}{\partial w_2}(w) \end{array} \right] = \text{gradient} \)
Idea:

Start somewhere
Repeat: Take a step in the gradient 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

- \( \alpha \): learning rate --- tweaking parameter that needs to be chosen carefully
- How? Try multiple choices
  - Crude rule of thumb: update changes \( \mathcal{W} \) about 0.1 – 1%

```
Init \( w \)
for iter = 1, 2, ...
\[
w \leftarrow w + \alpha \cdot \nabla g(w)
\]
```
Choice of learning rate $\alpha$ is a hyperparameter

Example: $\alpha=0.001$ (too small)

Source: https://distill.pub/2017/momentum/
Choice of step size $\alpha$ is a hyperparameter

Example: $\alpha=0.004$ (too large)

Source: https://distill.pub/2017/momentum/
Gradient Ascent with Momentum*

- Often use *momentum* to improve gradient ascent convergence

Gradient Ascent:

Init \( w \)

for iter = 1, 2, ...

\[
    w \leftarrow w + \alpha \cdot \nabla g(w)
\]

Gradient Ascent with momentum:

Init \( w \)

for iter = 1, 2, ...

\[
    z \leftarrow \beta \cdot z + \nabla g(w)
\]

\[
    w \leftarrow w + \alpha \cdot z
\]

- One interpretation: \( w \) moves like a particle with mass
- Another: exponential moving average on gradient
Gradient Ascent with Momentum*

Example: $\alpha=0.001$ and $\beta=0.0$

Source: https://distill.pub/2017/momentum/
Gradient Ascent with Momentum*

Example: $\alpha=0.001$ and $\beta=0.9$

Source: https://distill.pub/2017/momentum/
Batch Gradient Ascent on the Log Likelihood Objective

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

\[
g(w)
\]

\[\text{init } w\]
\[\text{for iter } = 1, 2, \ldots\]

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

\[
\text{init } w \\
\text{for iter } = 1, 2, \ldots \\
\quad \text{pick random } j \\
\quad w \leftarrow w + \alpha \cdot \nabla \log P(y^{(j)} | x^{(j)}; w)
\]
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

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

= special case of neural network

\[ z_i = \sum_j w_{i,j} \cdot f_i(x) = w_i \cdot f(x) \]
Deep Neural Network = Also learn the features!

\[ f_1(x) \rightarrow z_1 \rightarrow \text{softmax} \rightarrow P(y_1|x; w) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}} \]

\[ f_2(x) \rightarrow z_2 \rightarrow \text{softmax} \rightarrow P(y_2|x; w) = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}} \]

\[ f_3(x) \rightarrow z_3 \rightarrow \text{softmax} \rightarrow P(y_3|x; w) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}} \]

\[ f_4(x) \rightarrow \ldots \rightarrow z_3 \rightarrow \text{softmax} \]
Deep Neural Network = Also learn the features!

\[ x_1, x_2, x_3, \ldots, x_L, z_1^{(1)}, z_1^{(2)}, z_2^{(1)}, z_2^{(2)}, z_3^{(1)}, z_3^{(2)}, \ldots, z_K^{(1)}, z_K^{(2)} \]

\[ f_1(x), f_2(x), f_3(x), \ldots, f_L(x), z_1^{(OUT)}, z_2^{(OUT)}, z_3^{(OUT)}, \ldots, f_L(x) \]

\[ P(y_1|x; w), P(y_2|x; w), P(y_3|x; w) \]
Deep Neural Network

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

- Neural network with \( n \) layers
- \( z^{(k)} \): activations at layer \( k \)
- \( W^{(k-1,k)} \): weights taking activations from layer \( k-1 \) to layer \( k \)

\( g = \) nonlinear activation function
More compactly as matrix multiplication:

\[
\mathbf{z}^{(k)} = g\left( \mathbf{W}^{(k-1,k)} \mathbf{z}^{(k-1)} \right)
\]
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)**

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

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

Multiple outputs ("heads") possible

Can use learned features for classification (similar to logistic regression):

\[ P(y|x; w) \]

- \( P(y = +1|x; w) = \frac{1}{1 + e^{-z^{OUT}}} \)
- \( P(y = -1|x; w) = 1 - \frac{1}{1 + e^{-z^{OUT}}} \)
Deep Neural Network: Training

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!)
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 $f(x)$.

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”
Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko

Abstract. In this paper we demonstrate that finite linear combinations of compositions of sigmoidal activation functions and a net of affine functions can uniformly approximate any continuous function of a real variable with respect to the uniform metric. Only mild conditions are imposed on the sigmoid function. Our results settle as open questions about representability in the class of single hidden layer feedforward networks. In particular, we show that arbitrary discrete regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single hidden layer and any continuous sigmoidal activation function. The approximation error is measured in terms of uniform norm and the approximation can be obtained for very general conditions involving the sigmoidal activation function.

Key words. Neural networks, Approximation, Computation.

1. Introduction

A network of diverse application areas are concerned with the representation of general functions of an n-dimensional real variable, x ∈ R^n, by finite linear combinations of the form

\[ f(x) = \sum_{j=1}^{N} \theta_j \sigma_j(\sum_{i=1}^{m} \theta_{ji} \phi_i(x)), \]

where \( \theta_j \) and \( \theta_{ji} \) are fixed, \( \phi_i \) is the transpose of \( \phi \) so that \( \phi_j \) is the inner product of \( \phi \) and \( x_i \). Here the activation function \( \sigma \) depends heavily on the context of the application. Our major concern is with so-called sigmoidal \( \sigma \):

\[ \sigma(x) = \frac{1}{1 + e^{-x}}, \quad x \in \mathbb{R}. \]

Such functions arise naturally in neural network theory as the activation function of a neural node (or unit) in becoming the preferred term [L1, R1, H1]. The main result of this paper is a demonstration of the fact that some of the form (1) are dense in the space of continuous functions on the unit cube if \( x \) is any continuous sigmoidal.

* Data received: October 21, 1988. Revised: February 17, 1989. This research was supported in part by NSF Grant DCR-8619283, ONR Contract N00014-86-0022 and DOE Grant DE-FC02-82ER23067.

+ Center for Experimental Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, USA.

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

Shimon Schocken

Leonard N. Stern School of Business
New York University
New York, NY 10003

Center for Research on Information Systems Information Systems Department
Leonard N. Stern School of Business
New York University
Working Paper Series

STERNS-91-36

“Approximations by superpositions of sigmoidal functions”

Cybenko (1989) “Approximation with 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”
How about computing all the derivatives?

Derivatives tables:

<table>
<thead>
<tr>
<th>Derivative</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{d}{dx}(a) )</td>
<td>0</td>
</tr>
<tr>
<td>( \frac{d}{dx}(x) )</td>
<td>1</td>
</tr>
<tr>
<td>( \frac{d}{dx}(au) )</td>
<td>( a \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx}(u + v - w) )</td>
<td>( \frac{du}{dx} + \frac{dv}{dx} - \frac{dw}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx}(uv) )</td>
<td>( u \frac{dv}{dx} + v \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx}\left(\frac{u}{v}\right) )</td>
<td>( \frac{1}{v} \frac{du}{dx} - \frac{u}{v^2} \frac{dv}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx}(u^n) )</td>
<td>( nu^{n-1} \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx}(\sqrt{u}) )</td>
<td>( \frac{1}{2\sqrt{u}} \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx}\left(\frac{1}{u}\right) )</td>
<td>( -\frac{1}{u^2} \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx}\left(\frac{1}{u^n}\right) )</td>
<td>( -\frac{n}{u^{n+1}} \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx}[f(u)] )</td>
<td>( \frac{d}{du}[f(u)] \frac{du}{dx} )</td>
</tr>
</tbody>
</table>

[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:

  \[
  \text{If } \quad f(x) = g(h(x))
  \]

  \[
  \text{Then } \quad f'(x) = g'(h(x))h'(x)
  \]

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

Automatic differentiation software

- e.g. TensorFlow, PyTorch, Jax
- 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? Details outside of scope of CS188, but we’ll show a basic example
Example: Automatic Differentiation

- Build a *computation graph* and use chain rule
Example: Automatic Differentiation*

- Build a computation graph and use chain rule: \( f(x) = g(h(x)) \) \( f'(x) = g'(h(x))h'(x) \)
- Example: neural network with quadratic loss \( L(a_2, y^*) = \frac{1}{2} (a_2 - y^*)^2 \) and ReLU activations \( g(z) = \max(z, 0) \)
- \( a_2 = g_2(w_2 \cdot g_1(w_1 \cdot x)) \)

\[
\frac{\partial z_2}{\partial w_2} = \frac{\partial}{\partial w} (w_2 \cdot a_1) = a_1
\]

\[
\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial w_2} = 4 \cdot a_1 = 8
\]

\[
\frac{\partial L}{\partial a_2} = a_2 - y^* = 4
\]

\[
\frac{\partial L}{\partial a_2} = \frac{\partial L}{\partial z_2} \frac{\partial z_2}{\partial a_2} = 4 \cdot 1
\]

\[
\frac{\partial a_2}{\partial z_2} = \frac{\partial}{\partial z} \max(z_2, 0) = 1 \text{ (when } z_2 > 0) \]

\[
\frac{\partial L}{\partial y^*} = -(a_2 - y^*) = -4
\]
Fun Neural Net Demo Site

Demo-site:

http://playground.tensorflow.org/
Summary of Key Ideas

Optimize probability of label given input

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

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
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 early stopping!

Automatic differentiation gives the derivatives efficiently (how? = outside of scope of 188)