Computing Derivatives
Part II Roadmap

• Part I – Linear Algebra (units 1-12) \( Ac = b \)
  - linearize
  - line search

• Part II – Optimization (units 13-20)
  - (units 13-16) Optimization -> Nonlinear Equations -> 1D roots/minima
  - (units 17-18) Computing/Avoiding Derivatives
  - (unit 19) Hack 1.0: “I give up” \( H = I \) and \( J \) is mostly 0 (descent methods)
  - (unit 20) Hack 2.0: ”It’s an ODE!?“ (adaptive learning rate and momentum)
Smoothness

• Discontinuous functions cannot be differentiated
  • Even methods that don’t require derivatives struggle when functions are discontinuous

• Continuous functions may have kinks (discontinuities in derivatives)
  • Discontinuous derivatives cause methods that depend on derivatives to fail, since function behavior cannot be adequately predicted from one side of the kink to the other

• Typically, functions need to be “smooth enough”, which has varying meaning depending on the approach

• Specialty approaches for special classes of functions, e.g. linear algebra, linear programming, convex optimization, second order cone program, etc.
  • Nonlinear Systems/Optimization are so difficult that they often receive less scrutiny/care, as best practices/techniques often do not exist
Biological Neurons (towards real AI)

• Aim to mimic human biological neural networks and learning
• Biological neurons are “all or none”, which motivates similar strategies in artificial neural networks
  • This leads to a discontinuous function with an identically zero derivative everywhere else
  • Disastrous for optimization!
• Biological neurons fire with increased frequency for stronger signals
  • This leads to a piecewise constant and discontinuous derivative
  • Problematic for optimization!
• Smoothing allows optimization to work, i.e. to minimize the loss to find the parameters/coefficients for the network architecture
Heaviside Function

- \( H(x) = 1 \) for \( x \geq 0 \), and \( H(x) = 0 \) for \( x < 0 \)
- Motivated by biological neurons being “all or none”, but has a discontinuity at 0 and derivative identically zero elsewhere
(Inequality) Constrained Optimization

• Minimize $\hat{f}(c)$ subject to $\hat{g}(c) \geq 0$ (or strictly $\hat{g}(c) > 0$)

• Heaviside function can be used to create a penalty term $-H(-\hat{g}_i(c))\hat{g}_i(c)$ which is only nonzero when $\hat{g}_i(c) < 0$
  • This penalty term is minimized by forcing negative $\hat{g}_i(c)$ towards zero (as desired)

• Given diagonal matrix $D$ of (positive) weights indicating the relative importance of various constraints, unconstrained optimization can be used to minimize

$$\hat{f}(c) - \sum_i H \left( -\hat{e}_i^T D \hat{g}(c) \right) \hat{e}_i^T D \hat{g}(c)$$

• However, this requires differentiating the non-smooth Heaviside function
  • Smoothing the Heaviside function makes the cost function differentiable
Sigmoid Function (an example)

- Any smoothed Heaviside function, e.g. \( S(x) = \frac{1}{1+e^{-x}} \) (there are many options)
- Continuous and monotonically increasing, although the derivative is close to zero away from \( x = 0 \)
Example: Binary Classification

- Training data \((x_i, y_i)\) where the \(y_i = \pm 1\) are binary class labels
- Find hyperplane \(n^T (x - x_o) = 0\) that separates the data between the two class labels (\(n\) is the unit normal and \(x_o\) is a point on the plane)
- The closest \(x_i\) on each side of this hyperplane are called the support vectors
- If the hyperplane is equidistant between the support vectors, then they lie on parallel planes: \(n^T (x - x_o) = \pm \epsilon\) (where \(\epsilon\) is the margin)
- Dividing by \(\epsilon\) to normalize gives \(c^T (x - x_o) = \pm 1\) where \(c\) is in the normal direction (but not unit length), and then maximizing the margin \(\epsilon\) is equivalent to minimizing \(\|c\|_2\)
- That is, minimize \(\hat{f}(c) = \frac{1}{2} c^T c\) while still fitting the data
Example: Binary Classification

• Minimize $\hat{f}(c) = \frac{1}{2} c^T c$ subject to inequality constraints

• $c^T (x_i - x_o) \geq 1$ when $y_i = 1$ and $c^T (x_i - x_o) \leq -1$ when $y_i = -1$ can be combined into $y_i c^T (x_i - x_o) \geq 1$ for every data point

• Alternatively, $y_i (c^T x_i - b) \geq 1$ where the new scalar unknown is $b = c^T x_o$

• New data will be inferred/classified based on the sign of $c^T x_{new} - b$

• When approached via unconstrained optimization, the Heaviside function incorporates constraints into the cost function (and subsequently smoothing the Heaviside is called soft-margin)
Rectifier Functions (an example)

- $R(x) = \max(x, 0)$ or similar functions which are continuous and have increasing values
- Motivated by biological neurons firing with increased frequency for stronger signals
- Piecewise constant and discontinuous derivative causes issues with optimization
Softplus Function

- **Softplus function** $SP(x) = \log(1 + e^x)$ smooths the discontinuous derivative typical of rectifier functions
Leaky Rectifier Function

• Modifies the negative part of a rectifier function to also have a positive slope instead of being set to zero
• Can be smoothed (as well)
Arg/Soft Max

• **Arg Max** returns 1 for the largest argument and 0 for the other arguments
• E.g. (.99,1) → (0,1), (1,.99) → (1,0), etc.
• Highly discontinuous!

• **Soft Max** is a smoothed out version, e.g. \((x_1, x_2) \rightarrow \left(\frac{e^{x_1}}{e^{x_1}+e^{x_2}}, \frac{e^{x_2}}{e^{x_1}+e^{x_2}}\right)\)
• This is a smooth function of the arguments, differentiable, etc.
• Variants/weightings exist to make it closer/further from Arg Max (while preserving differentiability)
Symbolic Differentiation

• When a function is known in closed form, it can be differentiated by hand
• Software packages such as Mathematica can aid in symbolic differentiation (and subsequent simplification)
• Some benefits of knowing the closed form derivative:
  • Provides a better understanding of the underlying problem
  • Enables well thought out smoothing/regularization
  • Allows one to implement highly efficient code
  • Subsequently allows access to more accurate higher derivatives
  • Some of the aforementioned benefits enable the use of better solvers
  • Helps to write/maintain code with less bugs
  • Etc.
Symbolic Differentiation of Code

- Sometimes a function is not analytically known and/or merely represents the output of some source code.
- Often parts of code have known derivatives, and those known derivatives can be utilized/leveraged via the mathematical rules for differentiation.
- Moreover, when parts of the code are always used consecutively, they can be merged; subsequently, merged code with known derivatives in each part can often have the derivative treatment simplified for robustness/efficiency.
Differentiate the Right Thing

• Consider an iterative solver, e.g. CG, that solves $Ac = b$ to find $c$ given $b$

• Furthermore, suppose that the code is enormous, complicated, confusing, a black box, etc. (basically impenetrable)

• It is tempting to consider some of the code bases that claim to differentiate such chunks of code
  • In fact, many times these approaches work, and the answers are reasonable
  • Though it is always hard to know whether computational inaccuracies (as discussed in this class) are having an adverse effect in this black box approach

• On the other hand, when invertible, $c = A^{-1}b$ and $\frac{\partial c_k}{\partial b_i} = \tilde{a}_{ik}$ where $\tilde{a}_{ik}$ is an entry in $A^{-1}$ (a similar approach can be taken for $A^+$)

• That is, the derivative is independent of the iterative solver and the errors that might accumulate within it due to poor conditioning
Beware of the claim that it is good to be able to use something without understanding it.

The claim is true, and many of us enjoy driving our cars without knowledge or care of what is under the hood.

However, those who design cars, manufacture cars, repair cars, etc. benefit greatly from understanding as much as possible about them, and we too benefit enormously from their expertise.

Though, admittedly, there are those in the car business, such as used car salesmen, that authentically do not require any real knowledge/expertise.

The question is: what kind of computer scientist do you want to be?
Oversimplified Thinking

• Beware of claims that drastically oversimplify

• E.g., some say that code is very simple and merely consists of simple operations like add/subtract/multiply/divide that are easily differentiated

• However, in reality, even the simple $z = x + y$ has subtleties that can matter
• The computer actually executes $z = \text{round}(x + y)$

• Too many claim that issues they have not carefully considered don’t matter in practice; meanwhile, many of the practices are not well understood (leaving one to question the first claim)
Finite Differences

• Derivatives can be approximated by various formulas, which (recall) is how the Secant method was derived from Newton’s method

• Given a small perturbation \( h \), Taylor expansions can be manipulated to write:
  
  • **Forward Difference**: \( g'(t) = \frac{g(t+h) - g(t)}{h} + O(h), 1^{st} \text{ order accurate} \)
  
  • **Backward Difference**: \( g'(t) = \frac{g(t) - g(t-h)}{h} + O(h), 1^{st} \text{ order accurate} \)
  
  • **Central Difference**: \( g'(t) = \frac{g(t+h) - g(t-h)}{2h} + O(h^2), 2^{nd} \text{ order accurate} \)
  
  • **Second Derivative**: \( g''(t) = \frac{g(t+h) - 2g(t) + g(t-h)}{h^2} + O(h^2), 2^{nd} \text{ order accurate} \)

• These approximations can be evaluated even when \( g(t) \) is not known precisely but merely represents the output of some code with input \( t \)
Finite Differences (Drawbacks)

• Finite Differences only give an approximation to the derivative, and contain truncation errors related to the perturbation size \( h \)

• One has to reason about the effects that truncation error and the size of \( h \) have on other aspects of the code

• If the code is very long and complex, the overall effects of truncation errors may be unclear

• Still, finite difference methods have had a broad positive impact in computational science!
Automatic Differentiation

- In machine learning, this is often referred to as **Back Propagation**
- For every (potentially vector valued) function \( F(c_{input}) \) written into the code, an analytically correct companion function for the Jacobian matrix \( \frac{\partial F}{\partial c} (c_{input}) \) is also written into the code
- Then when evaluating \( F(c_{input}) \), one can also evaluate \( \frac{\partial F}{\partial c} (c_{input}) \)
  - Of course, \( \frac{\partial F}{\partial c} (c_{input}) \) contains roundoff errors based on machine precision (and conditioning, etc.)
  - But it does not contain the much larger truncation errors present in finite differencing
- Code chunks combine together various functions via arithmetic/compositional rules
- Analytic differentiation has its own set of rules (linearity, product rule, quotient rule, chain rule, etc.) that are used to assemble the derivative (evaluated at \( c_{input} \)) for the code chunk
  - Roundoff errors will accumulate, of course, and the resulting error has the potential to be catastrophic
  - Similar (potentially worse) sentiments hold for the much larger truncation errors
Second Derivatives

• If $c_{input}$ is size $n$ and $F(c_{input})$ is size $m$, then the Jacobian matrix $\frac{\partial F}{\partial c}(c_{input})$ is size $mxn$

• The Hessian of second derivatives is a size $mxnxn$
  • Note that $m = 1$ for optimization, i.e. for $\hat{f}(c_{input})$

• Writing automatic differentiating functions for all second derivatives can be difficult/tedious

• Storing Hessians for all second derivatives can be unwieldy/intractable

• Roundoff error accumulation is an even bigger problem for second derivatives, and the resulting errors are even more likely to have adverse effects

• Additional smoothness is required for second derivatives as well
  • This is a problem for any method that considers a second derivatives, and is not specific to automatic differentiation approaches
One way to combat overfitting is to train several different network architectures on the same data, inference them all, and average the result.

- This is costly, especially if there are many networks.

Dropout is a “hacky” approach to achieving a network function averaged over multiple network architectures.
- Though Google did patent it!

The idea is to simply ignore parts of the code with some probability when training the network, mimicking a perturbed network architecture.

- Although this can be seen as computing correct derivatives on perturbed functions, it can also equivalently be seen as adding uncertainty to the derivative computation.
- That is, instead of regularization via model averaging, it can be seen as creating a network robust to errors in derivative estimation.
Function Layers

- More complex processes work in a pipeline with many complex layers
- Each layer completes a task on its inputs $X_j$ to create outputs $X_{j+1}$
- Each layer may depend on parameters $C_j$
- There may be a known/desired output $X_{target}$ to compare the final result to

$$
\hat{f}(X_4) = \|X_4 - X_{target}\|
$$
Function Layers (Example)

**Layer 1**
- **Input**: animation controls
- **Function**: linear blend shapes, nonlinear skinning, quasistatic physics simulation, etc. to deform a face
- **Parameters**: lots of hand tuned or known parameters including shape libraries, etc.
- **Output**: vertex positions of a triangle mesh
Function Layers (Example)

LAYER 2

• **Input**: vertex positions of a triangle mesh
• **Function**: scanline renderer or ray tracer
• **Parameters**: lots of hand tuned or known parameters for material models, lighting and shading, textures, etc.
• **Output**: RGB colors for pixels (an image)
Function Layers (Example)

LAYER 3

- **Input**: RGB colors for pixels (an image)
- **Function**: facial landmark detector
- **Parameters**: parameters for the network architecture determined by training the network to match hand labeled data
- **Output**: 2D locations of landmarks on the image
Function Layers (Example)

TARGET OUTPUT

• Run a landmark detector on a photograph of the individual to obtain 2D landmark positions

• The goal is to have the 2D landmarks output from the complex multi-layered function match the 2D landmarks on the photograph
Function Layers (Example)

- Modifying animation controls changes the triangulated surface which changes the rendered pixels in the image which changes the network’s determination of landmarks.
- When the two sets of landmarks agree, the animation controls indicate what the person in the photograph was doing.
Classical Optimization

- Find the input $X_1$ that minimizes $\hat{f}(X_4)$

$$\frac{\partial \hat{f}(X_4)}{\partial x_1} = \frac{\partial \hat{f}(X_4)}{\partial x_4} \frac{\partial x_4}{\partial x_3} \frac{\partial x_3}{\partial x_2} \frac{\partial x_2}{\partial x_1} = \frac{\partial f(X_4)}{\partial x_4} \frac{\partial f_3(x_3, c_3)}{\partial x_3} \frac{\partial f_2(x_2, c_2)}{\partial x_2} \frac{\partial f_1(x_1, c_1)}{\partial x_1}$$

- Parameters are considered fixed/constant

$$\hat{f}(X_4) = \|X_4 - X_{target}\|$$
Network Training

- Train network $f_2$ by finding parameters $C_2$ that minimize $\hat{f}(X_4)$
- Chain rule:
  \[
  \frac{\partial \hat{f}(X_4)}{\partial C_2} = \frac{\partial \hat{f}(X_4)}{\partial X_4} \frac{\partial X_4}{\partial X_3} \frac{\partial X_3}{\partial C_2} = \frac{\partial \hat{f}(X_4)}{\partial X_4} \frac{\partial f_3(X_3, C_3)}{\partial X_3} \frac{\partial f_2(X_2, C_2)}{\partial C_2}
  \]

\[
\begin{align*}
X_1 & \xrightarrow{\text{in}} f_1(X_1, C_1) & \xrightarrow{\text{out}} X_2 & \xrightarrow{\text{in}} f_2(X_2, C_2) & \xrightarrow{\text{out}} X_3 & \xrightarrow{\text{in}} f_3(X_3, C_3) & \xrightarrow{\text{out}} X_4 \\
& \overset{\text{params}}{\downarrow} & \overset{\text{params}}{\downarrow} & \overset{\text{params}}{\downarrow} & \overset{\text{params}}{\downarrow} & \overset{\text{params}}{\downarrow} & \\
\hat{f}(X_4) &= \|X_4 - X_{\text{target}}\|}
\]
Network Training

• Any preprocess to the network does not require differentiability
• The network itself only requires differentiability in terms of its parameters
• Any postprocess to the network requires input/output differentiability, but does not require differentiability in terms of its parameters

\[ \hat{f}(X_4) = \|X_4 - X_{target}\| \]