Unit 17A – Computing Derivatives

Smoothness
- **Discontinuous functions** cannot be differentiated, and even the methods that don’t require derivatives struggle when functions are discontinuous
- Even continuous functions may have **kinks** that represent discontinuities in their derivatives
- A discontinuous derivative can cause methods that depend on derivatives to fail, as they cannot adequately predict derivative behavior from one side of the kink to the other
- Typically, functions need to be what is called “smooth enough”, which varies depending on the approach
- Nonlinear Optimization and Nonlinear Systems are quite difficult, so many researchers focus on slightly easier problems at the boundary between linear algebra and nonlinear phenomena
  - Examples include convex optimization, linear programming problems, etc. (we will not cover these methods in class, but you should be aware of them)

Activation Functions
- In spite of the fact that machine/deep learning training data works off of the simple idea of function interpolation (admittedly, this statement contains my own bias), the greater community working on artificial intelligence often aims to mimic human biological neural networks
- Biological neurons have a “all or none” property meaning they are either firing or not, which leads those building artificial neural networks to utilize similar concepts in their network design
- **Heaviside function**: $H(x)=1$ for $x \geq 0$, and $H(x)=0$ for $x < 0$
  - The Heaviside function is not continuous at 0, and has a derivative of zero everywhere making it a disaster for everything we’ve previously discussed in optimization
- Any smoothed out Heaviside function is loosely referred to as a **Sigmoid function**, e.g. consider $S(x) = \frac{1}{1+e^{-x}}$ (but there are lots of popular options)
  - The Sigmoid function is continuous and has a monotonically increasing derivative everywhere, but the derivative is close to zero everywhere except near $x = 0$
  - Although biological neurons either fire or they don’t, their firing frequency increases with stronger signals, and this has led to the use of **Rectifier functions** $R(x) = \max(x,0)$, which are continuous and have increasing values
  - The piecewise constant and discontinuous derivative can be fixed with a smoothed out version called the **Softplus function**: $SP(x) = \log(1 + e^x)$
  - The **Leaky Rectifier** modifies the negative part of the rectifier to also have a negative slope instead of being set to zero (and this too can be smoothed)
  - Generally speaking, smoothing so that the optimization works better to minimize the loss and find the parameters/coefficients of the network is quite common
  - For example, **Soft Max** is a smoothed out version of **Arg Max** where instead of choosing the largest argument one uses a differentiable function that depends on all the arguments and tends towards the largest one
Symbolic Differentiation
- If \( g(t) \) is known is closed form, one can differentiate it by hand
- Alternatively, one can use code such as Mathematica to aid in the process of differentiation and subsequent simplification
- The benefit here is that the derivates are known as functions themselves, whereas other approaches can only calculate/evaluate the numerical value of derivatives at a given point
- This often allows for much more efficient/optimized code
- However, when \( g(t) \) is unknown analytically, and merely represents the output of some code with input \( t \), one cannot use this approach

Finite Differences
- Can manipulate the Taylor Expansion to show that:
  - Forward Difference: \( g'(t) = \frac{g(t+h) - g(t)}{h} + O(h), \ 1^{st} \) order accurate
  - Backward Difference: \( g'(t) = \frac{g(t) - g(t-h)}{h} + O(h), \ 1^{st} \) order accurate
  - Central Difference: \( g'(t) = \frac{g(t+h) - 2g(t) + g(t-h)}{2h} + O(h^2), \ 2^{nd} \) order accurate
  - Second Derivative: \( g''(t) = \frac{g(t+h) - 2g(t) + g(t-h)}{h^2} + O(h^2), \ 2^{nd} \) order accurate
- These and similar approximations can be used to replace derivates and partial derivatives wherever they appear, noting that the perturbation \( h \) needs to be small compared to the variation in the functions in order to have any reasonable accuracy
- Note that the finite difference approximations can be applied even when \( g(t) \) is not known precisely but merely represents the output of some code with input \( t \)
- These are similar to aforementioned secant methods, but rely on sampling the function at nearby (i.e. small \( h \)) locations to approximate derivatives, instead of using prior iterates

Automatic Differentiation
- Finite Differences have truncation errors related to the perturbation size \( h \), and it is hard to know how small \( h \) needs to be especially when \( g(t) \) is a chunk of code with input \( t \)
- Automatic Differentiation aims to avoid truncation error entirely producing an estimate of derivatives precise to roundoff error levels
- It does this by considering the code as a sequence of operations and functions that can be differentiated using the chain rule
- Often, when someone writes a function, they provide a corresponding implementation of the derivative of that function for use in automatic differentiation
- There are various options including both forward and reverse accumulation, but it all uses the same standard chain rule
- Machine Learning folks like to refer to this approach as Back Propagation

Mike’s Network Cost Function Stuff (Unit 17B)