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)

Theory

Methods

Smoothness

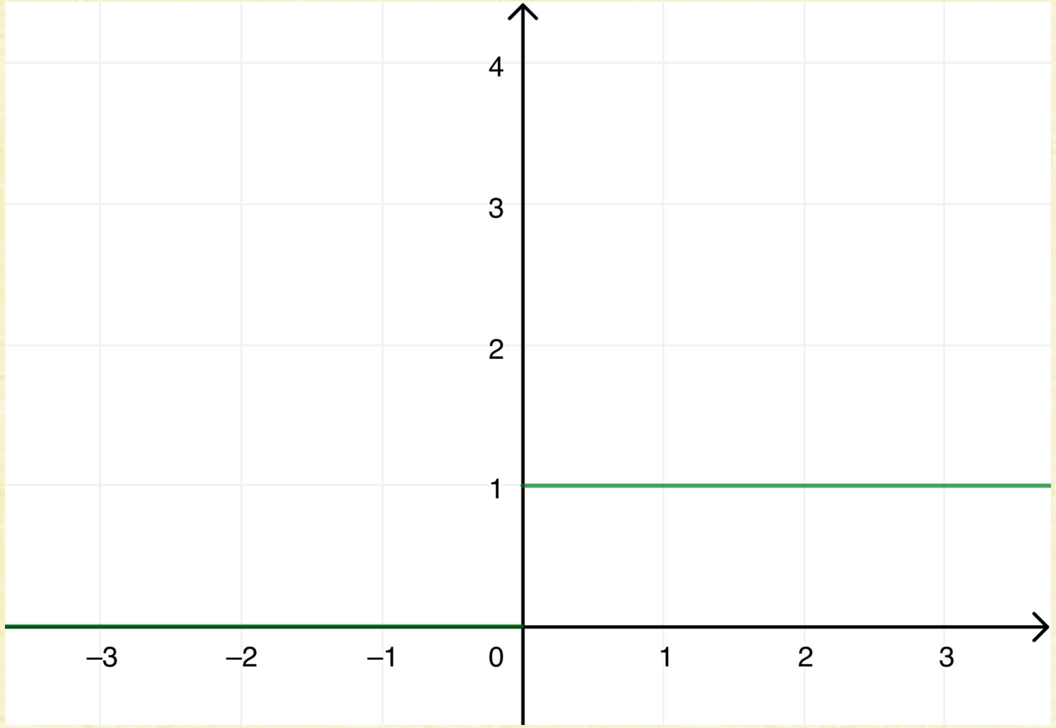
- Discontinuous functions cannot be differentiated
  - Even methods that don’t require derivatives struggle when functions are discontinuous
- Continuous functions may have kinks (discontinuities in derivatives)
  - Discontinuous derivatives can cause methods that depend on derivatives to fail, since 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 exist for special classes of functions, e.g. linear algebra, linear programming, convex optimization, second order cone program (SOCP), etc.
  - Nonlinear Systems/Optimization are more difficult, and best practices/techniques often do not exist

Biological Neurons (towards “real” AI)

- The aim is to mimic biological (typically human) 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. allows one to minimize the loss to find the parameters/coefficients for the network architecture

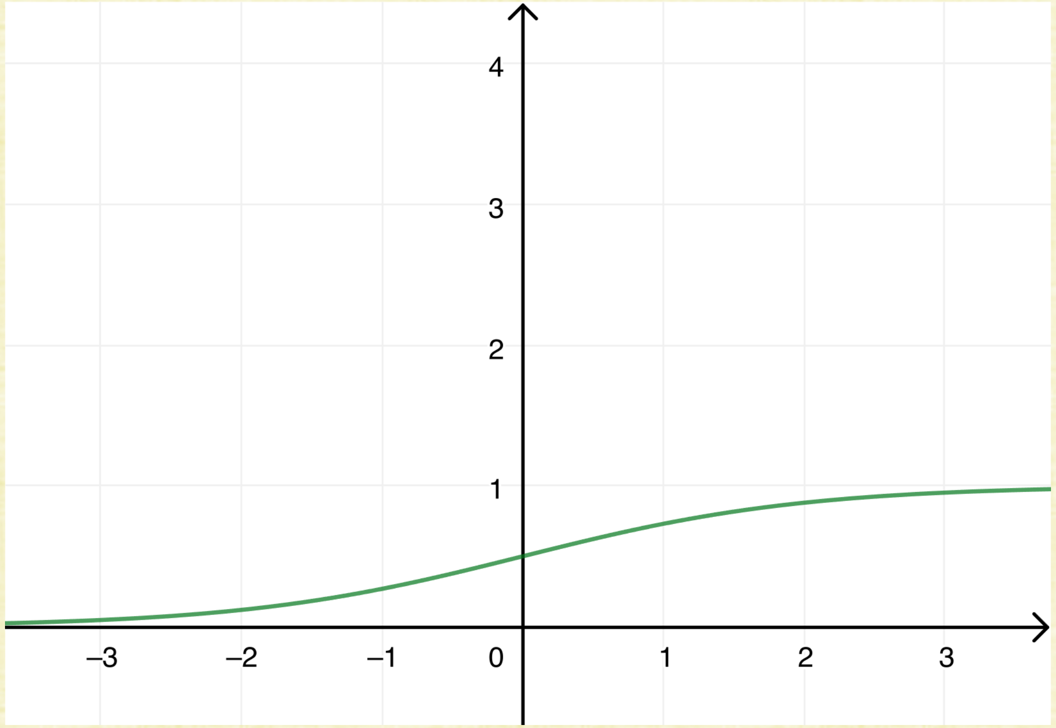
Heaviside Function

- $H(x) = 1$ for $x \geq 0$, and $H(x) = 0$ for $x < 0$
- Motivated by biological neurons being "all or none"
- Has a discontinuity at 0 and an identically zero derivative everywhere else



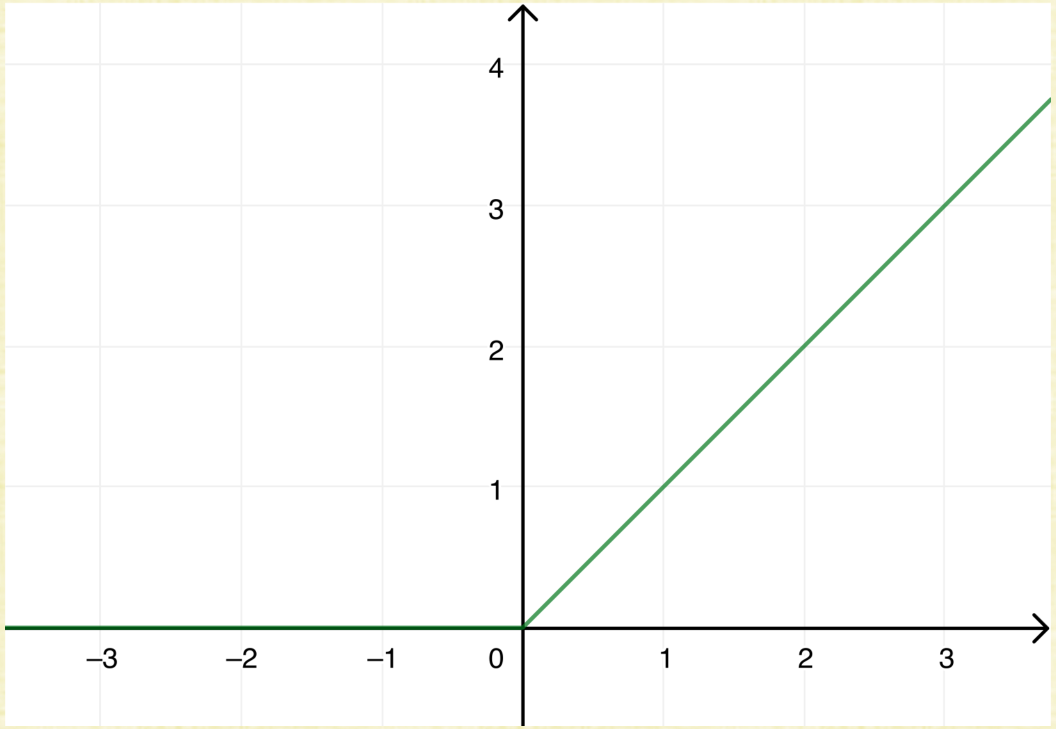
Sigmoid Function

- 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 further away from $x = 0$



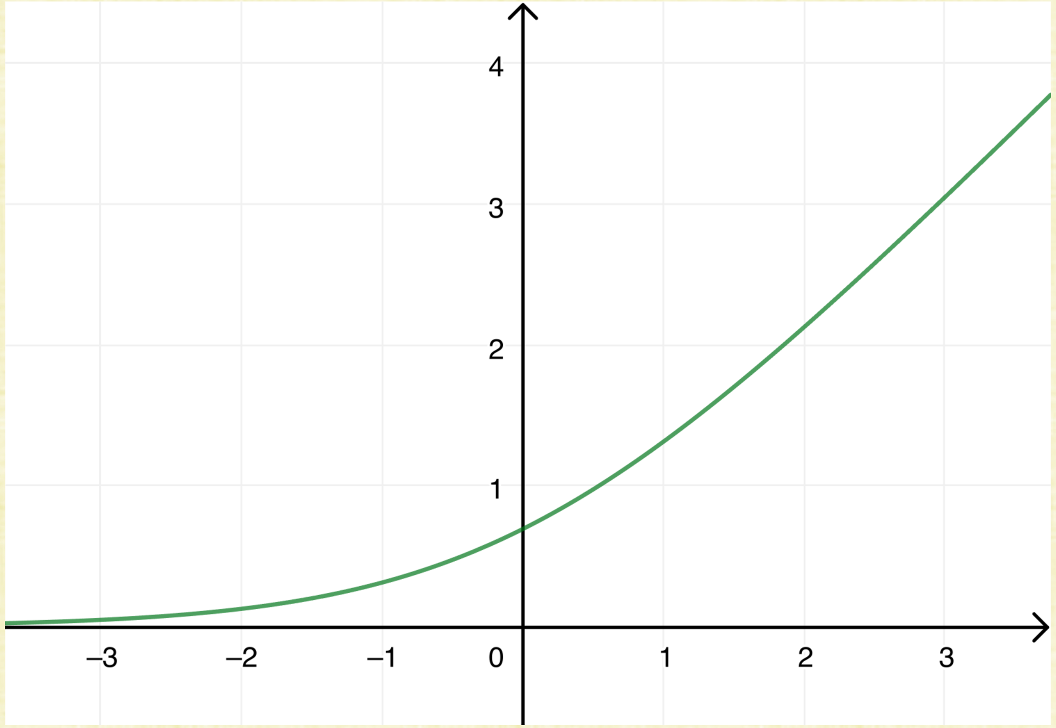
Rectifier Functions

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



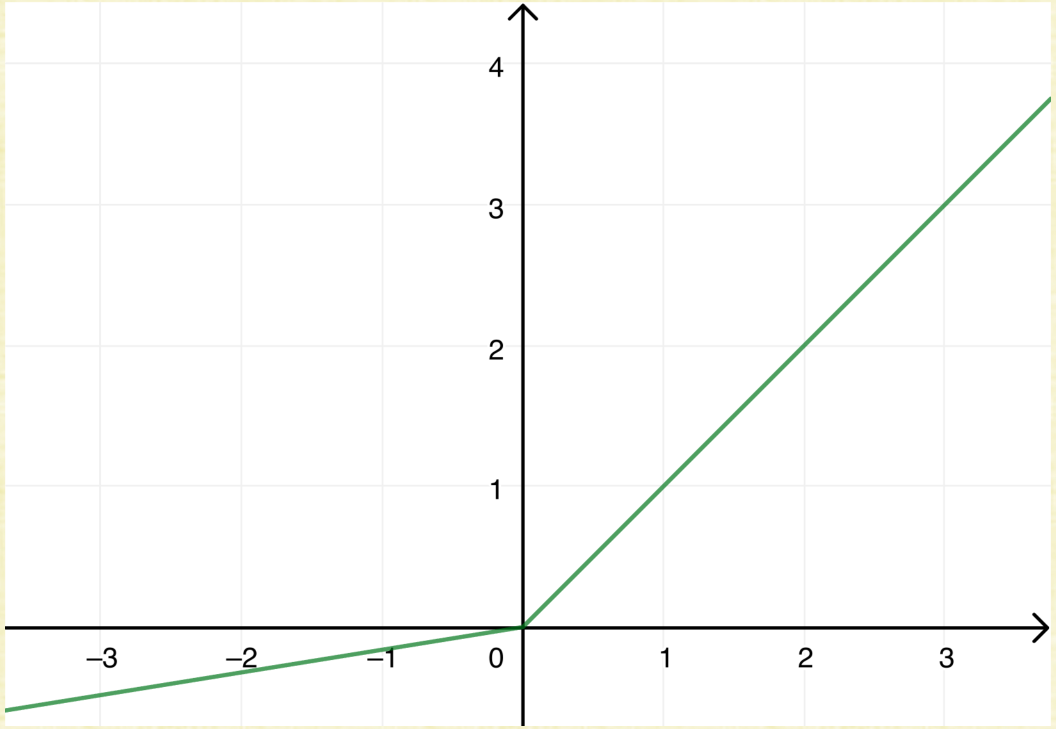
Softplus Function

- 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. $(.99,1) \rightarrow (0,1), (1,.99) \rightarrow (1,0)$, etc.
- Highly discontinuous!

- Soft Max is a smoothed 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)

Binary Classification

- Training data (x_i, y_i) where the $y_i = \pm 1$ are binary class labels

- Find plane $\hat{n}^T(x - x_o) = 0$ that separates the data between the two class labels ($\hat{n}$ is the unit normal and x_o is a point on the plane)
- The closest x_i on each side of the plane are called <u>support vectors</u>
- If the separating plane is equidistant between the support vectors, then they lie on parallel planes: $\hat{n}^T(x - x_o) = \pm\epsilon$ (where ϵ is the <u>margin</u>)
- Dividing by ϵ to normalize gives $c^T(x - x_o) = \pm 1$ where c points in the normal direction (but is not unit length); then, maximizing the margin ϵ is equivalent to minimizing $\|c\|_2$

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 combine these into $y_i c^T(x_i - x_o) \geq 1$ for every data point
  - Alternatively, $y_i(c^T x_i - b) \geq 1$ with a scalar unknown $b = c^T x_o$
- When approached via unconstrained optimization, Heaviside functions can be used to incorporate the constraints into the cost function
  - Subsequently smoothing those Heaviside functions is called soft-margin
- Note: new data is classified (via inference) based on the sign of $c^T x_{new} - b$

(Inequality) Constrained Optimization

- Minimize $\hat{f}(c)$ subject to $\hat{g}(c) \geq 0$ (or $\hat{g}(c) \ > 0$)
- Create a penalty term $-H\big(-\hat{g}_i(c)\big)\hat{g}_i(c)$, which is nonzero only when $\hat{g}_i(c) < 0$
    - This penalty term is minimized by forcing negative $\hat{g}_i(c)$ towards zero (as desired)
- Given a diagonal matrix D of (positive) weights indicating the relative importance of various constraints, <u>unconstrained</u> optimization can be used to minimize $\hat{f}(c) - \sum_i H\left(-\hat{e}_i^T D \hat{g}(c)\right) \hat{e}_i^T D \hat{g}(c)$
    - This requires differentiating the non-smooth Heaviside function
    - Smoothing the Heaviside function makes the modified cost function differentiable

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 more efficient code
  - Subsequently allows access to higher derivatives
  - Some of the aforementioned benefits enable the use of better solvers
  - Helps to write/maintain code with less bugs
  - Etc.

Example

- Suppose a code has the following functons:
  - $f(t) = t^2 - 4$ with $f'(t) = 2t$, and $g(t) = t - 2$ with $g'(t) = 1$
- Suppose another part of the code combines these functions:
  - $h(t) = \frac{f(t)}{g(t)}$ with $h'(t) = \frac{g(t)f'(t) - f(t)g'(t)}{\left(g(t)\right)^2}$
- Then $h(2) = \frac{f(2)}{g(2)} = \frac{0}{0}$ and $h'(2) = \frac{g(2)f'(2) - f(2)g'(2)}{\left(g(2)\right)^2} = \frac{0 \cdot 4 - 0 \cdot 1}{0^2}$
  - Adding a small $\epsilon > 0$ to the denominators (to avoid division by zero) gives $h(2) = 0$ and $h'(2) = 0$
  - Adding a small $\epsilon > 0$ to denominators is often done whenever the denominators are small, making $h(t) \approx 0$ and $h'(t) \approx 0$ for $t \approx 2$ as well
- Of course, $h(t) = t + 2$ is a straight line with $h(2) = 4$ and $h'(t) = 1$ everywhere

Symbolic Differentiation of Code

- Sometimes a function is not analytically known and/or merely represents the output of some source code
- But, parts of the code may 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 accuracy/robustness/efficiency

Differentiate the Right Thing

- Consider an iterative solver (e.g. CG, Minres, etc.) that solves $Ac = b$ to find c given b
- Sometimes 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
  - Sometimes these approaches work, and the answers are reasonable
  - But, it is often difficult to know whether or not computational inaccuracies (as discussed in this class) are having an adverse effect on such a black box approach
- Alternatively, when invertible: $c = A^{-1}b$ and $\frac{\partial c_k}{\partial b_i} = \tilde{a}_{ki}$ where $\tilde{a}_{ki}$ is an entry in A^{-1}
  - A similar approach can be taken for A^+, which can be estimated robustly via PCA, the Power Method, etc.
- The derivative is independent of the iterative solver (CG, Minres, etc.) and the errors that might accumulate within the iterative solver due to poor conditioning
  - More recently, this sort of approach is being referred to as an implicit layer

The Used Car Salesman

- Beware of the claim: it is good to be able to use something without understanding it
- The claim is often true, and many of us enjoy driving our cars without understanding much 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 the rest of us benefit enormously from their expertise)
- Though, admittedly, there are those in the car business, such as those who sell used cars, who legitimately don't 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
  - E.g. the computer actually executes $z = round(x + y)$

- Too many claim that issues they have not carefully considered don't matter in practice; meanwhile, many state-of-the-art practices in ML/DL are not well understood in the first place (leaving one to question these sorts of claims)

Finite Differences

- Derivatives can be approximated by various formulas, similar to how the Secant method was derived from Newton's method
- Given a small perturbation $h > 0$, Taylor expansions can be manipulated to write:
  - Forward Difference: $g'(t) = \frac{g(t+h)-g(t)}{h} + O(h)$, 1st order accurate
  - Backward Difference: $g'(t) = \frac{g(t)-g(t-h)}{h} + O(h)$, 1st order accurate
  - Central Difference: $g'(t) = \frac{g(t+h)-g(t-h)}{2h} + O(h^2)$, 2nd order accurate
  - Second Derivative: $g''(t) = \frac{g(t+h)-2g(t)+g(t-h)}{h^2} + O(h^2)$, 2nd 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
- 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 can be considered in chunks, which 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 can be 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 (this is typically even worse for the much larger truncation errors)

Second Derivatives

- If c_{input} is size n and $F(c_{input})$ is size m, the Jacobian matrix $\frac{\partial F}{\partial c}(c_{input})$ is size mxn
- The Hessian of second derivatives is size $mxnxn$
  - Recall: $m = 1$ for optimization, i.e. for $\hat{f}(c_{input})$
- Writing automatic differentiation functions for all possible second derivatives can be difficult/tedious
- Storing Hessians for all second derivatives can be unwieldy/intractable
- Roundoff error accumulation can be an even bigger problem for second derivatives, and the resulting errors are typically even more likely to lead to adverse effects
- Additional smoothness is required for second derivatives
- Some of these issues are problems for any method that considers second derivatives (not specific to an automatic differentiation approach)

Dropout

- One idea for combating overfitting is to train several different network architectures on the same data, inference them all, and average the result (model averaging)
  - This can be costly, especially if there are many networks
- Dropout is a "hacky" approach to achieving a 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 <u>correct derivatives on perturbed functions/architectures</u>, it can also equivalently be seen as <u>adding uncertainty to the derivative computation</u>
- That is, instead of <u>regularization via model averaging</u>, it can be seen as <u>creating a network robust to errors in the derivatives</u>

*Bard did so poorly, they renamed it Gemini; how is Gemini doing?

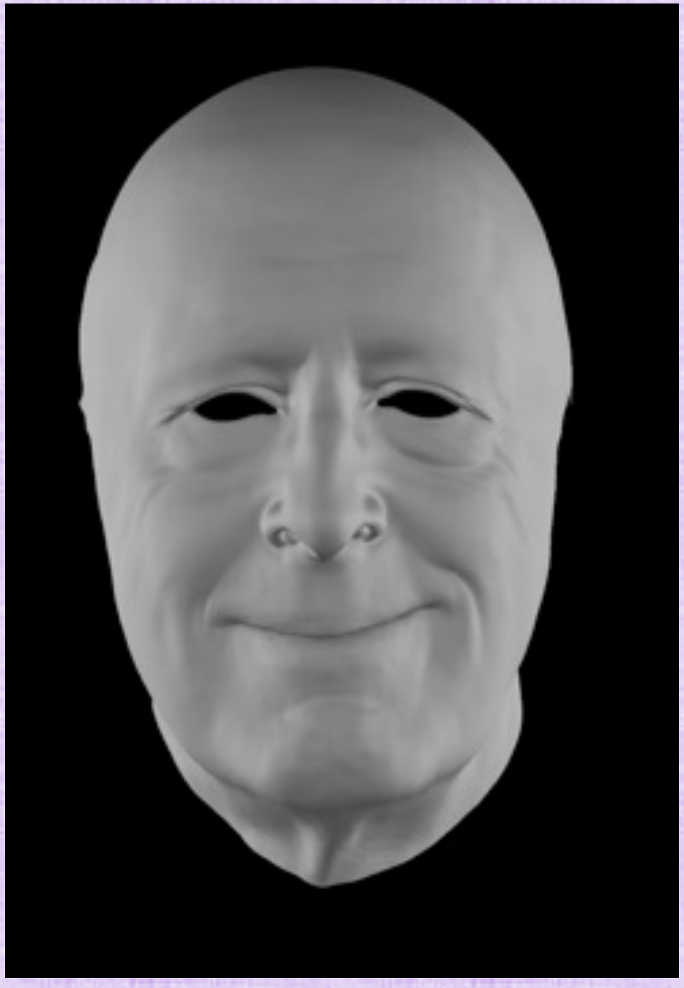
Function Layers

- Many complex processes work in a pipeline with many function layers
- Each layer completes a tasks 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

$$\begin{tikzcd}
 & C_1 \arrow[d, "\text{params}"] & & C_2 \arrow[d, "\text{params}"] & & C_3 \arrow[d, "\text{params}"] & \\
X_1 \arrow[r, "\text{in}"] & f_1(X_1; C_1) \arrow[r, "\text{out}"] & X_2 \arrow[r, "\text{in}"] & f_2(X_2; C_2) \arrow[r, "\text{out}"] & X_3 \arrow[r, "\text{in}"] & f_3(X_3; C_3) \arrow[r, "\text{out}"] & X_4
\end{tikzcd}$$

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

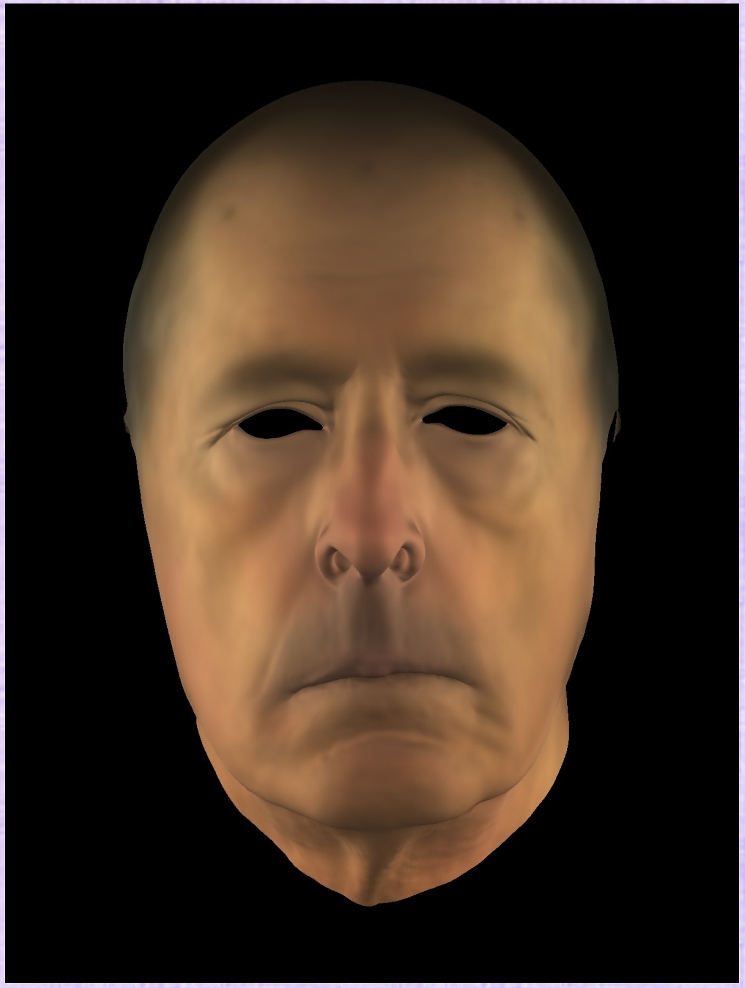
Function Layers (an example)



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: 3D vertex positions of a triangle mesh

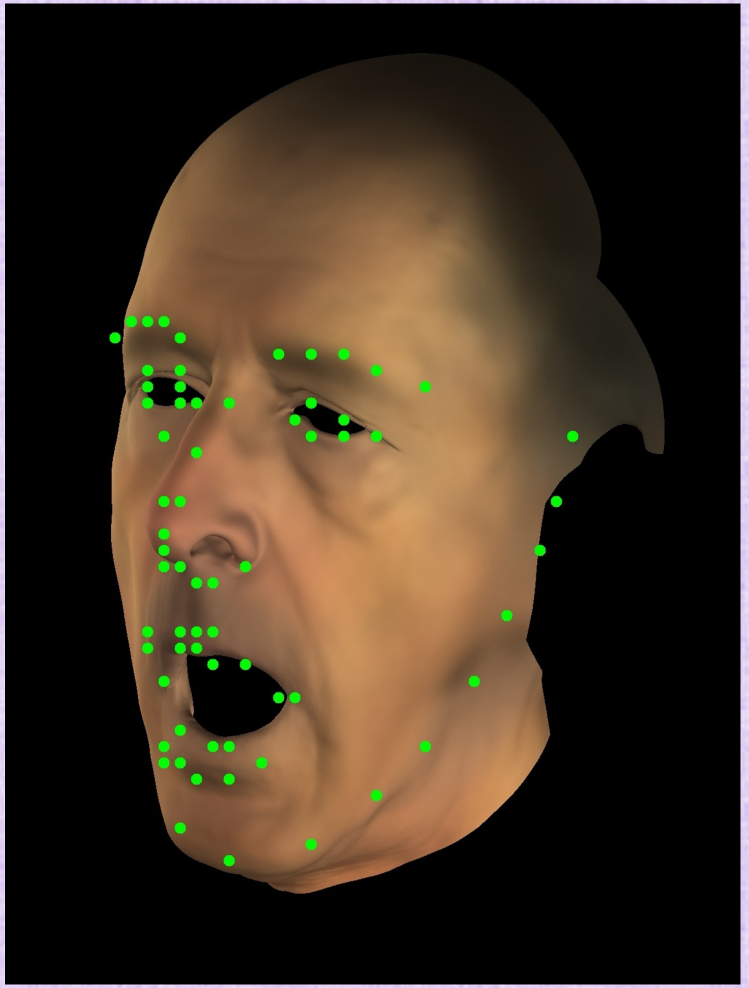
Function Layers (an example)



LAYER 2

- <u>Input</u>: 3D vertex positions of a triangle mesh
- <u>Function</u>: scanline renderer or ray tracer
- <u>Parameters</u>: lots of hand tuned or known parameters for material models, lighting and shading, textures, etc.
- <u>Output</u>: RGB colors for pixels (a 2D image)

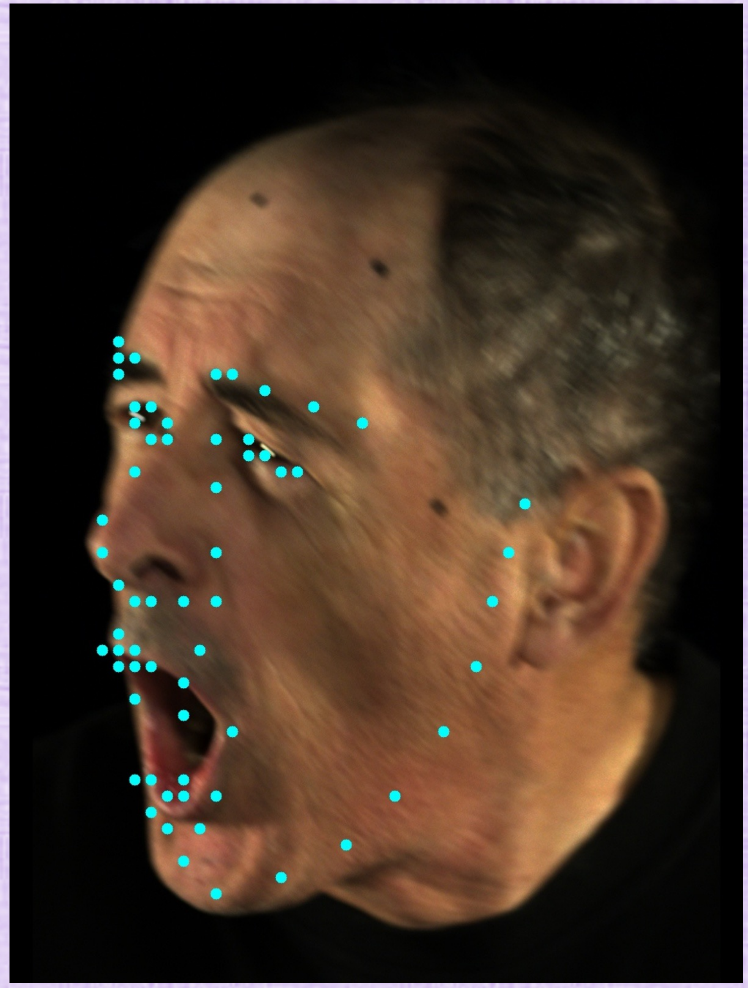
Function Layers (an example)



LAYER 3

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

Function Layers (an example)

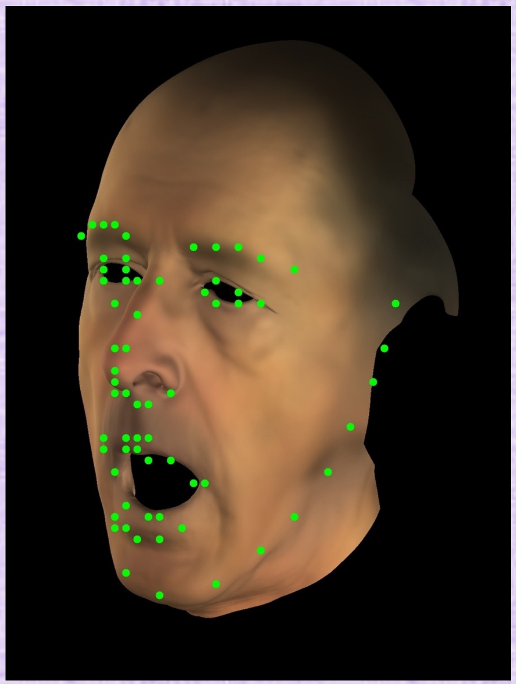


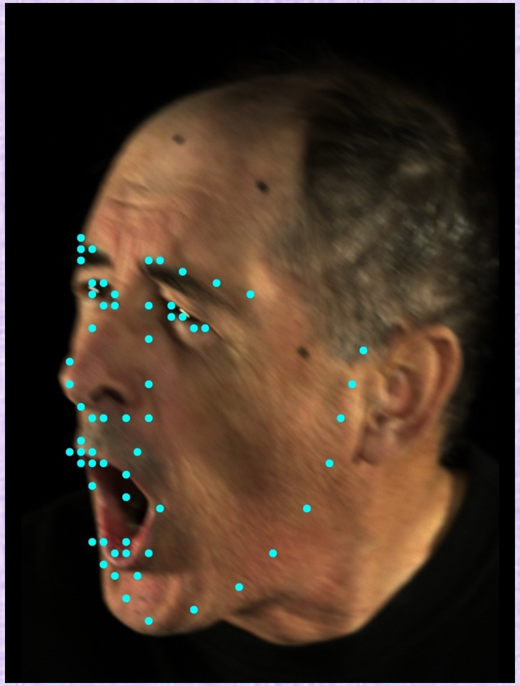
TARGET

- Run a landmark detector on a photograph of the individual to obtain 2D landmark locations (alternatively, can label by hand)
- The goal is to have the 2D landmarks output from the complex multi-layered function (on the prior three slides) match the 2D landmarks on the photograph

Function Layers (Example)

- Modifying animation controls changes the triangulated surface which changes the rendered pixels in the 2D image which changes the network's determination of the landmarks locations
- When the two sets of landmarks agree, the animation controls give some indication of what the person in the photograph was doing





Classical Optimization

- Find the input X_1 that minimizes $\hat{f}(X_4)$
- Chain rule: $\frac{\partial \hat{f}(X_4)}{\partial X_1} = \frac{\partial \hat{f}(X_4)}{\partial X_4}\frac{\partial X_4}{\partial X_3}\frac{\partial X_3}{\partial X_2}\frac{\partial X_2}{\partial X_1} = \frac{\partial \hat{f}(X_4)}{\partial X_4}\frac{\partial f_3(X_3,C_3)}{\partial X_3}\frac{\partial f_2(X_2,C_2)}{\partial X_2}\frac{\partial f_1(X_1,C_1)}{\partial X_1}$
- Parameters are considered fixed/constant

$$
\begin{array}{ccccccccccccc}
 & & C_1 & & & & C_2 & & & & C_3 & & \\
 & & \downarrow{\scriptstyle \text{params}} & & & & \downarrow{\scriptstyle \text{params}} & & & & \downarrow{\scriptstyle \text{params}} & & \\
X_1 & \xrightarrow{\text{in}} & f_1(X_1; C_1) & \xrightarrow{\text{out}} & X_2 & \xrightarrow{\text{in}} & f_2(X_2; C_2) & \xrightarrow{\text{out}} & X_3 & \xrightarrow{\text{in}} & f_3(X_3; C_3) & \xrightarrow{\text{out}} & X_4
\end{array}
$$

$$\hat{f}(X_4) = \left\| X_4 - X_{target} \right\|$$

Network Training

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

$$\begin{tikzcd} & C_1 \arrow[d, "\text{params}"] & & C_2 \arrow[d, "\text{params}"] & & C_3 \arrow[d, "\text{params}"] & \\ X_1 \arrow[r, "\text{in}"] & f_1(X_1; C_1) \arrow[r, "\text{out}"] & X_2 \arrow[r, "\text{in}"] & f_2(X_2; C_2) \arrow[r, "\text{out}"] & X_3 \arrow[r, "\text{in}"] & f_3(X_3; C_3) \arrow[r, "\text{out}"] & X_4 \end{tikzcd}$$

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

Network Training

- Any preprocess to the network does not require differentiability
- The network itself only requires differentiability with respect to its parameters
- Any postprocess to the network requires input/output differentiability, but does not require differentiability with respect to its parameters

$$\begin{tikzcd} & C_1 \arrow[d, "\text{params}"] & & C_2 \arrow[d, "\text{params}"] & & \cancel{C_3} \arrow[d, "\text{params}"] & \\ X_1 \arrow[r, "\text{in}"] & f_1(X_1, C_1) \arrow[r, "\text{out}"] & X_2 \arrow[r, "\text{in}"] & f_2(\cancel{X_2}, C_2) \arrow[r, "\text{out}"] & X_3 \arrow[r, "\text{in}"] & f_3(X_3, \cancel{C_3}) \arrow[r, "\text{out}"] & X_4 \end{tikzcd}$$

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