Calculus

Arthur J. Redfern
arthur.redfern@utdallas.edu

Outline

- Motivation
- Derivatives
- Optimization
- Neural network training
- Universal function approximation
- References

Not here; we'll only talk about derivatives

Disclaimer

- This set of slides is more accurately titled “A brief refresher of a subset of calculus for people already somewhat familiar with the topic followed by it’s specific application to xNN related items needed by the rest of the course”

- However, that’s not very catchy so we’ll just stick with “Calculus”

- In all seriousness, recognize that calculus is a very broad and deep topic that has and will continue to occupy many lifetimes of work; if interested in learning more, please consult the references to open a window into a much larger world

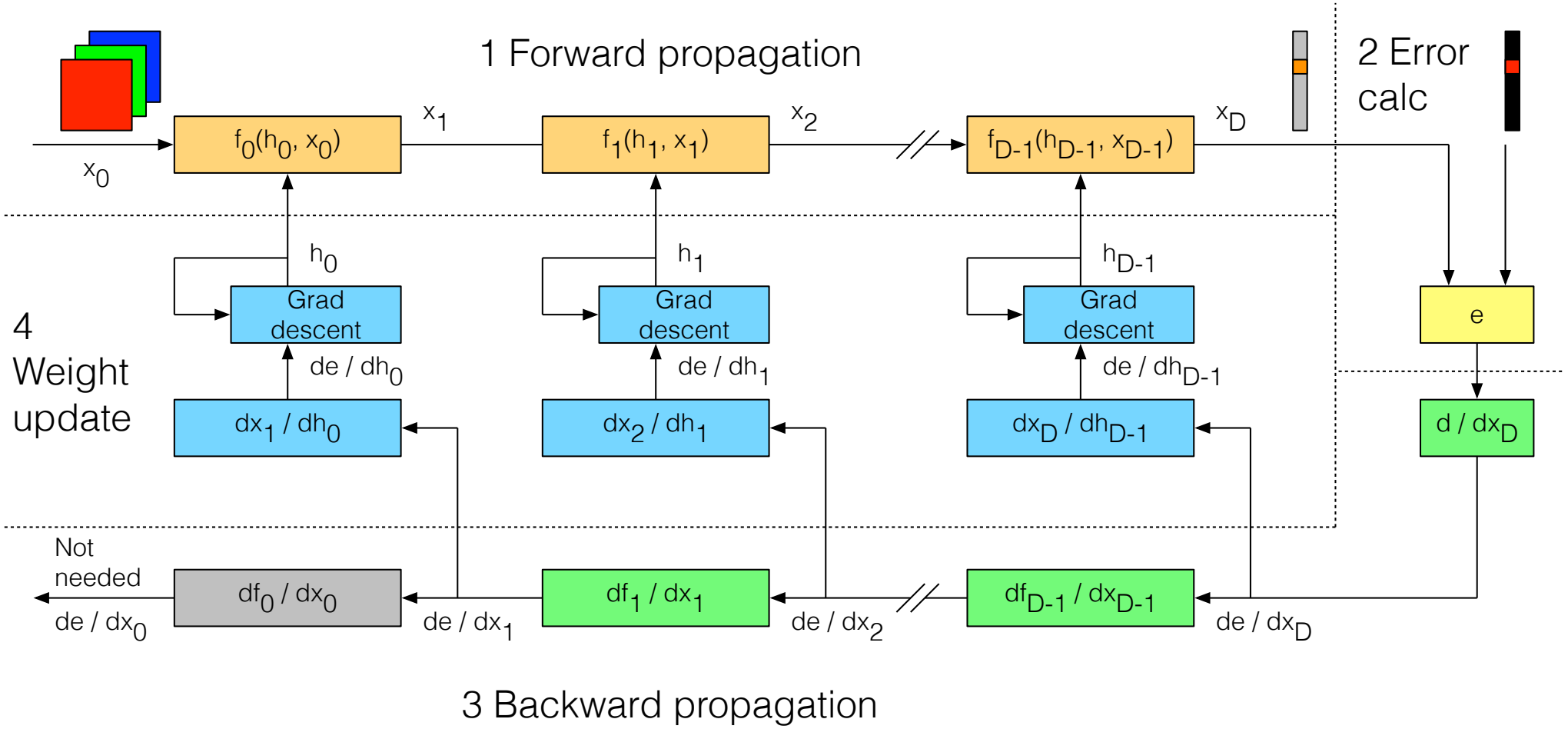
Motivation

Optimization And Approximation

- Neural network training is an optimization problem that we solve using calculus (math highlights listed, obviously others also present); the key is that all layers in forward propagation and error calculation are differentiable
  - Forward propagation - linear algebra, algorithms and probability
  - Error calculation - probability and info theory
  - Backward propagation - calculus
  - Weight update - calculus

- Neural networks are universal function approximators
  - A general tool for solving feature extraction and prediction problems
  - A constructive proof provides some feel for how problem complexity and network design are related

Big Picture Reminder



Derivatives

A Cookbook For Every (Every) xNN Layer

- For every layer we've seen in the linear algebra, algorithms and probability lectures
  - We already have a forward function
    - output = forward(input, params)
  - This section will give us the backward function
    - sensitivity of the error wrt the input and params = backward(sensitivity of the error wrt the output, input, params)

- It's a little tedious to go through every case, but once it's done it's done and we just use it like a cookbook (a forward function has a given backward function that we derived once and never have to derive again)
  - Scalar functions of a single variable
  - Scalar functions of multiple variables
  - Vector functions of multiple variables

- Then in the next section we'll use the sensitivity of the error with respect to the params to update the params to reduce the error

Scalar Functions Of A Single Variable

- Scalar functions of a single variable: $y = f(x)$
  - $f: R \rightarrow R$
- Definition of the derivative
  - $df/dx = \lim_{\Delta x \rightarrow 0} (f(x + \Delta x) - f(x)) / \Delta x$
- Interpretation
  - The derivative is a linear operator that maps functions to function
  - Intuition of sensitivity of function f to changes in input around x

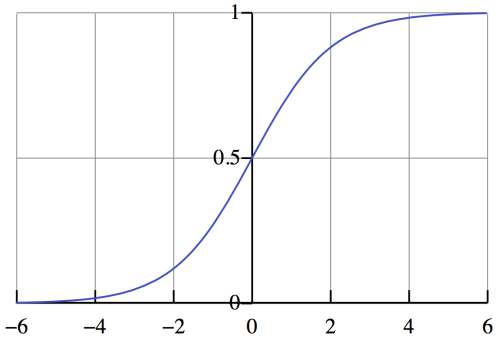
Scalar Functions Of A Single Variable

- Differentiable functions
  - At values of x where the limit exists f is differentiable
- Non differentiable functions
  - At values of x where the limit does not exist f is not differentiable
  - If there are a countable number of non differentiable values then it's potentially possible to approximate the derivative of f at a value where it's not differentiable via the derivative just to the left or right (or any derivative in this range)
  - This will be useful in taking the derivative of the ReLU and other functions

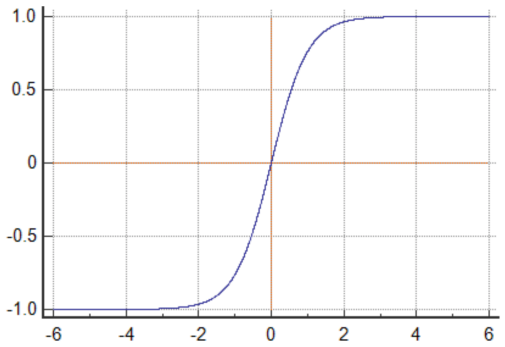
Scalar Functions Of A Single Variable

These are common pointwise nonlinearities applied after linear transformation + bias addition

- Example: sigmoid
  - f(x) = $1 / (1 + e^{-x})$
  - df/dx = f(x)(1 – f(x))

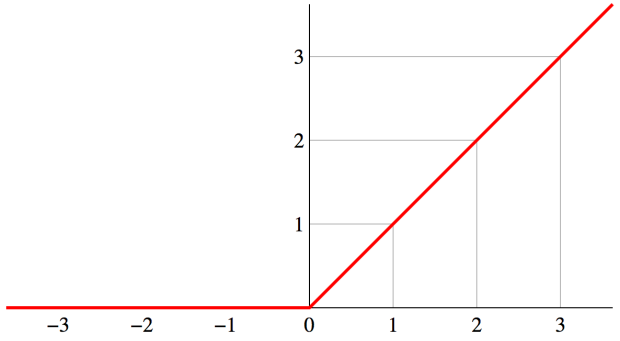


- Example: tanh
  - f(x) = $(e^{x} – e^{-x}) / (e^{x} + e^{-x})$
  - df/dx = $1 – f^{2}(x)$



Even though sigmoid, tanh and ReLU layers typically take a tensor in and generate a tensor out, the mapping is done on a per element basis making the derivative calculation that of a scalar function of a single variable, repeated for every element of the tensor

- Example: ReLU
  - f(x) = max(0, x)
  - df/dx = 0, $x < 0$
    = 0, $x = 0$, though any df/dx $\in [0, 1]$ is ok
    = 1, $x > 0$



Figures from https://en.wikipedia.org/wiki/Sigmoid_function, https://www.medcalc.org/manual/tanh_function.php and https://en.wikipedia.org/wiki/Ramp_function

Scalar Functions Of A Single Variable

These are common elementwise operations; in the case of a tensor input output mapping through 1 of these layers these rules are applied individually to each element of the tensor

- Example: elementwise addition
  - $f(a, x) = a + x$
  - $\partial f/\partial x = 1$
  - $\partial f/\partial a = 1$

- Example: elementwise multiplication
  - $f(a, x) = a\ x$
  - $\partial f/\partial x = a$
  - $\partial f/\partial a = x$

- Example: fixed scale
  - $f(x) = \alpha\ x$
  - $\partial f/\partial x = \alpha$

Scalar Functions Of A Single Variable

- Product rule
  - h(x) = f(x) g(x)
  - dh/dx = (df/dx) g(x) + f(x) (dg/dx)

- Quotient rule
  - h(x) = f(x) / g(x)
  - dh/dx = ((df/dx) g(x) – f(x) (dg/dx)) / g^2(x), general case
    = – (dg/dx) / g^2(x), f(x) = 1 case

- Chain rule
  - h(x) = f(g(x)) and define u = g(x)
  - dh/dx = (df/du) (du/dx)

Scalar Functions Of Multiple Variables

- Scalar function of multiple variables: $y = f(\mathbf{x})$
  - $f: R^K \rightarrow R$
  - $\mathbf{x} = [x(0), x(1), ..., X(K-1)]^T$

- Partial derivative
  - Let $\mathbf{e}_k$ be coordinate direction k
  - $\partial f / \partial x_k = \lim_{\Delta \rightarrow 0} (f(\mathbf{x} + \Delta \mathbf{e}_k) - f(\mathbf{x})) / \Delta$

- Interpretation
  - View all variables x(i) where $i \neq k$ as constants
  - Take derivative of f at $\mathbf{x}$ with respect to x(k)
  - Result is slope of the function f at point $\mathbf{x}$ in coordinate direction $\mathbf{e}_k$

Scalar Functions Of Multiple Variables

- Gradient
  - K x 1 vector of partial derivatives with respect to each variable
  - $\nabla f(\mathbf{x}) = [(\partial f/\partial x(0)), (\partial f/\partial x(1)), ..., (\partial f/\partial x(K-1))]^T$

- Interpretation
  - Same shape and dimension as $\mathbf{x}$ (note this is choosing denominator layout)
  - Points in direction of maximum change of f at $\mathbf{x}$
  - When doing iterative error minimization we're typically going to figure out the gradient of the error with respect to the parameters (i.e., the sensitivity of the error to changes in the parameters) and adjust the parameters in the opposite direction of the gradient to reduce the error
  - Directional derivative in unit vector direction $\mathbf{u}$ can be found from $(\nabla f(\mathbf{x}))^T \mathbf{u}$

Scalar Functions Of Multiple Variables

These are common examples from linear algebra; derive all by writing out in summation notation, differentiating with respect to 1 variable while freezing the rest and repeating for all variables while arranging results to match format

- Example: vector vector multiplication
  - $f(\mathbf{x}^T, \mathbf{a}) = \mathbf{x}^T \mathbf{a}$, $(1 \times 1) = (1 \times K)(K \times 1)$
  - $\partial f / \partial \mathbf{x}^T = \mathbf{a}^T$, $(1 \times K) = (1 \times K)$
  - $\partial f / \partial \mathbf{a} = \mathbf{x}$, $(K \times 1) = (K \times 1)$

- Example: vector vector multiplication (same as above just flipping variables)
  - $f(\mathbf{a}^T, \mathbf{x}) = \mathbf{a}^T \mathbf{x}$, $(1 \times 1) = (1 \times K)(K \times 1)$
  - $\partial f / \partial \mathbf{a}^T = \mathbf{x}^T$, $(1 \times K) = (1 \times K)$
  - $\partial f / \partial \mathbf{x} = \mathbf{a}$, $(1 \times K) = (1 \times K)$

- Example: vector matrix vector multiplication
  - $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x}$
  - $\partial f / \partial \mathbf{x} = (\mathbf{A} + \mathbf{A}^T) \mathbf{x}$, general case
    $= 2 \mathbf{A} \mathbf{x}$, symmetric $\mathbf{A}$ case

Scalar Functions Of Multiple Variables

These are common pooling operations that can be considered from this perspective

- Example: average pooling
  - Divide the gradient at the output equally to form the gradient at all the inputs
  - To see this consider each output of average pooling as the result of vector vector multiplication $y = \mathbf{a}^T \mathbf{x}$ where all elements of $\mathbf{a}$ are 1/M where M is the vector length; then apply the rule from the previous page and repeat for each output

- Example: max pooling
  - Send gradient at output to the max input that created it
  - Requires saving the max index from the forward path for each output
  - To see this consider max pooling as an identity scale that connects the max element of the input with the output and 0 scales for all of the other input elements and repeat for each output

Scalar Functions Of Multiple Variables

These are common examples used in network error calculation, $\mathbf{x}^*$, $\mathbf{p}^*$ and k* are used to denote the ideal values

- Example: mean square error
  - $f(\mathbf{x}^*, \mathbf{x}) = (1/K)\ (\mathbf{x} - \mathbf{x}^*)^T\ (\mathbf{x} - \mathbf{x}^*)$
  - $\partial f/\partial \mathbf{x} = (2/K)\ (\mathbf{x} - \mathbf{x}^*)$
  - Commonly used in regression

- Example: cross entropy
  - $f(\mathbf{p}^*, \mathbf{p}) = -\Sigma_k\ p^*(k)\ \log(p(k))$,
    $= -\log(p(k^*))$, if $\mathbf{p}^*$ has a 1 at position k* and 0s elsewhere
  - $\partial f/\partial \mathbf{p} = -1/(p(k^*))$, at $k = k^*$
    $= 0$, at $k \neq k^*$
  - Commonly used in classification
  - Log base e is used here for convenience so no extra scale values
  - The $\partial f/\partial \mathbf{p}$ vector only having 1 non 0 element will allow the product of the derivative of the softmax function and cross entropy function to have a very nice form

Vector Functions Of Multiple Variables

- Vector functions of multiple variables: $\mathbf{y} = f(\mathbf{x})$
  - $f: R^K \rightarrow R^M$

- Jacobian
  - Using denominator notation the Jacobian is a K x M matrix with entry (k, m) corresponding to the partial derivative of output f(m) with respect to input variable x(k)
  - $J_f(\mathbf{x}) = \begin{bmatrix} \partial f(0)/\partial x(0), & \ldots, & \partial f(M-1)/\partial x(0) \\ \vdots, & \ldots, & \vdots \\ \partial f(0)/\partial x(K-1), & \ldots, & \partial f(M-1)/\partial x(K-1) \end{bmatrix}$

- Comments
  - Will use the Jacobian to transform error gradients through layers during back propagation
  - The Jacobian is the optimal linear approximation of a vector valued function near a point **a**
    - $f(\mathbf{x}) \approx f(\mathbf{a}) + J_f^T(\mathbf{a})(\mathbf{x} - \mathbf{a})$

Vector Functions Of Multiple Variables

These are common linear transforms that comprise key xNN layers

- Example: vector vector multiplication (scalar function, repeating here for convenience)
  - $f(\mathbf{x}^T, \mathbf{a}) = \mathbf{x}^T \mathbf{a}$, $(1 \times 1) = (1 \times K)(K \times 1)$
  - $\partial f/\partial \mathbf{x}^T = \mathbf{a}^T$, $(1 \times K) = (1 \times K)$
  - $\partial f/\partial \mathbf{a} = \mathbf{x}$, $(K \times 1) = (K \times 1)$

- Example: vector matrix multiplication
  - $\mathbf{f}^T(\mathbf{x}^T, \mathbf{A}) = \mathbf{x}^T \mathbf{A} = \mathbf{x}^T [\mathbf{a}_0, \ldots, \mathbf{a}_{N-1}]$, $(1 \times N) = (1 \times K)(K \times N)$, derivation: repeated vector vector
  - $\partial \mathbf{f}^T/\partial \mathbf{x}^T = \mathbf{A}^T$, $(N \times K) = (N \times K)$, use: $\partial e/\partial \mathbf{x}^T = (\partial e/\partial \mathbf{y}^T)(\partial \mathbf{y}^T/\partial \mathbf{x}^T)$
  - $\partial \mathbf{f}^T/\partial \mathbf{A} = \mathbf{x}$, $(K \times 1) = (K \times 1)$, use: $\partial e/\partial \mathbf{A} = (\partial \mathbf{y}^T/\partial \mathbf{A})(\partial e/\partial \mathbf{y}^T)$

- Example: matrix matrix multiplication
  - $\mathbf{F}^T(\mathbf{X}^T, \mathbf{A}) = \mathbf{X}^T \mathbf{A} = [\mathbf{x}_0^T; \ldots; \mathbf{x}_{M-1}^T] \mathbf{A}$, $(M \times N) = (M \times K)(K \times N)$, derivation: repeated vector matrix
  - $\partial \mathbf{F}^T/\partial \mathbf{X}^T = \mathbf{A}^T$, $(N \times K) = (N \times K)$, use: $\partial e/\partial \mathbf{X}^T = (\partial e/\partial \mathbf{Y}^T)(\partial \mathbf{Y}^T/\partial \mathbf{X}^T)$
  - $\partial \mathbf{F}^T/\partial \mathbf{A} = \mathbf{X}$, $(K \times M) = (K \times M)$, use: $\partial e/\partial \mathbf{A} = (\partial \mathbf{Y}^T/\partial \mathbf{A})(\partial e/\partial \mathbf{Y}^T)$

Vector Functions Of Multiple Variables

These are common linear transforms that comprise key xNN layers; same as the previous page with **f** (**F**), **x** (**X**) and **a** (**A**) transposed

- Example: vector vector multiplication (scalar function, repeating here for convenience)
  - $f(\mathbf{a}^T, \mathbf{x}) = \mathbf{a}^T \mathbf{x}$, $(1 \times 1) = (1 \times K)(K \times 1)$
  - $\partial f / \partial \mathbf{a}^T = \mathbf{x}^T$, $(1 \times K) = (1 \times K)$
  - $\partial f / \partial \mathbf{x} = \mathbf{a}$, $(1 \times K) = (1 \times K)$

- Example: matrix vector multiplication
  - $\mathbf{f}(\mathbf{A}^T, \mathbf{x}) = \mathbf{A}^T \mathbf{x} = [\mathbf{a}_0^T; \ldots; \mathbf{a}_{M-1}^T] \mathbf{x}$, $(M \times 1) = (M \times K)(K \times 1)$, derivation: repeated vector vector
  - $\partial \mathbf{f} / \partial \mathbf{A}^T = \mathbf{x}^T$, $(1 \times K) = (1 \times K)$, use: $\partial e / \partial \mathbf{A}^T = (\partial e / \partial \mathbf{y})(\partial \mathbf{y} / \partial \mathbf{A}^T)$
  - $\partial \mathbf{f} / \partial \mathbf{x} = \mathbf{A}$, $(K \times M) = (K \times M)$, use: $\partial e / \partial \mathbf{x} = (\partial \mathbf{y} / \partial \mathbf{x})(\partial e / \partial \mathbf{y})$

- Example: matrix matrix multiplication
  - $\mathbf{F}(\mathbf{A}^T, \mathbf{X}) = \mathbf{A}^T \mathbf{X} = \mathbf{A}^T [\mathbf{x}_0, \ldots, \mathbf{x}_{N-1}]$, $(M \times N) = (M \times K)(K \times N)$, derivation: repeated matrix vector
  - $\partial \mathbf{F} / \partial \mathbf{A}^T = \mathbf{X}^T$, $(N \times K) = (N \times K)$, use: $\partial e / \partial \mathbf{A}^T = (\partial e / \partial \mathbf{Y})(\partial \mathbf{Y} / \partial \mathbf{A}^T)$
  - $\partial \mathbf{F} / \partial \mathbf{X} = \mathbf{A}$, $(K \times M) = (K \times M)$, use: $\partial e / \partial \mathbf{X} = (\partial \mathbf{Y} / \partial \mathbf{X})(\partial e / \partial \mathbf{Y})$

Vector Functions Of Multiple Variables

These are common linear transforms that comprise key xNN layers

- Example: split
  - Add all gradients at the output to form the gradient at the input

- Example: CNN style 2D convolution
  - $f(\mathbf{H}^{No \times Ni \times Fr \times Fc}, \mathbf{X}^{Ni \times Lr \times Lc}) = \mathbf{H}^{No \times Ni \times Fr \times Fc} \circledast \mathbf{X}^{Ni \times Lr \times Lc}$
  - Derivative computation strategy
    - Lower CNN style 2D convolution into matrix matrix multiplication: $\mathbf{Y}^{2D} = \mathbf{H}^{2D}\ \mathbf{X}^{2D}$ where $\mathbf{Y}^{2D}$ is a 1 to 1 rearrangement of the 3D output tensor $\mathbf{Y}$, $\mathbf{H}^{2D}$ is a 1 to 1 rearrangement of the 4D weight tensor $\mathbf{H}$ and $\mathbf{X}^{2D}$ is a Toeplitz style filtering matrix where each element of the 3D input tensor $\mathbf{X}$ appears ~ Fr*Fc times
    - As such, in the forward direction CNN style 2D convolution can be viewed as a split operation on X followed by matrix matrix multiplication
    - For the error gradient mapping in the backwards direction apply the derivative rule for matrix multiplication followed by the derivative rule for split

Vector Functions Of Multiple Variables

The softmax function conceptually maps a vector input to a pmf and is commonly used before the error calculation in classification problems

- Softmax
  - No parameters so only $\partial \mathbf{f}/\partial \mathbf{x}$ needs to be computed
  - $f(\mathbf{x}) = (1/(\Sigma_k\, e^{x(k)}))\, [e^{x(0)}, e^{x(1)}, ..., e^{x(K-1)}]^T$
  - $\partial \mathbf{f}/\partial \mathbf{x} =$

$$\begin{bmatrix} f(0)(1-f(0)), & -f(1)f(0), & \cdots, & -f(K-1)f(0) \\ -f(0)f(1), & f(1)(1-f(1)), & & -f(K-1)f(1) \\ \vdots & \vdots & \ddots & \vdots \\ -f(0)f(K-1), & -f(1)f(K-1), & & f(K-1)(1-f(K-1)) \end{bmatrix}$$

Vector Functions Of Multiple Variables

This is commonly used in classification network training

- Chain rule
  - $h(\mathbf{x})$ = $f(g(\mathbf{x}))$ and define $\mathbf{u} = g(\mathbf{x})$
  - $\partial \mathbf{h}/\partial \mathbf{x}$ = $(\partial \mathbf{u}/\partial \mathbf{x})\ (\partial \mathbf{f}/\partial \mathbf{u})$, note denominator layout ordering

- Example: softmax cross entropy error gradient for classification
  - Consider a xNN that outputs a vector, converted to a pmf via softmax and an error via cross entropy
  - It's much more stable to take the partial derivative with respect to both the softmax and cross entropy together than separately because of this clean result
  - $\mathbf{p}^*$ has a 1 at position k* and 0s elsewhere
  - $\mathbf{p}$ = $\text{softmax}(\mathbf{x})$, like $\mathbf{u} = g(\mathbf{x})$
  - $f(\mathbf{p})$ = $H_{CE}(\mathbf{p}^*, \mathbf{p})$, like $f(\mathbf{u})$
  - $e(\mathbf{x})$ = $H_{CE}(\mathbf{p}^*, \text{softmax}(\mathbf{x}))$, like $h(\mathbf{x}) = f(g(\mathbf{x}))$
  - $\partial e/\partial \mathbf{x}$ = $(\partial \mathbf{p}/\partial \mathbf{x})\ (\partial f/\partial \mathbf{p})$, both components shown on earlier slides
    = $[p(0), \ \ldots, \ p(k^*-1), \ (p(k^*)-1), \ p(k^*+1), \ \ldots, \ p(K-1)]^T$

Optimization

Setup

- Problem
  - Minimize $f_0(\mathbf{x})$, objective function
  - Subject to $f_c(\mathbf{x}) \leq b_c$, $c = 1, ..., C$, constraint functions
- Solution
  - $\mathbf{x}^*$ is the optimal solution if it has the smallest objective value out of all vectors that satisfy the constraints
- A few special classes of optimization problems
  - Convex: $f_i(\alpha\mathbf{x}_0 + \beta\mathbf{x}_1) \leq \alpha f_i(\mathbf{x}_0) + \beta f_i(\mathbf{x}_1)$, $i = 0, 1, ..., C$, $\alpha + \beta = 1$, $\alpha \geq 0$, $\beta \geq 0$
  - Linear: $f_i(\alpha\mathbf{x}_0 + \beta\mathbf{x}_1) = \alpha f_i(\mathbf{x}_0) + \beta f_i(\mathbf{x}_1)$, $i = 0, 1, ..., C$

Critical Points

- Critical points
  - For a scalar function of multiple variables a critical point is a value **x** at which all partial derivatives of f(**x**) are 0
  - Local extrema of f(**x**) occur at critical points (Fermat's theorem)
    - Minimum (in all directions)
    - Maximum (in all directions)
    - Saddle (minimum in some directions, maximum in some directions)

- Potential notational confusion
  - In optimization literature it's common to optimize over parameters **x**
  - In xNNs we have inputs **x** and optimize over parameters **h** (or some other variable)
  - So in the optimization section our goal is to find the optimal **x**, and in the xNN training section it's to find the optimal **h** (or other parameter variable)

Closed Form Methods

- Example: least squares solution to a linear systems of equations
  - Minimize $f(\mathbf{x}) = (\mathbf{A}\,\mathbf{x} - \mathbf{y})^T (\mathbf{A}\,\mathbf{x} - \mathbf{y})$
  - Multiply out, take partial derivatives, set to 0 and solve
  - $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A}^T \mathbf{A}\,\mathbf{x} - \mathbf{x}^T \mathbf{A}^T \mathbf{y} - \mathbf{y}^T \mathbf{A}\,\mathbf{x} + \mathbf{y}^T \mathbf{y}$
    $= \mathbf{x}^T \mathbf{A}^T \mathbf{A}\,\mathbf{x} - 2\,\mathbf{y}^T \mathbf{A}\,\mathbf{x} + \mathbf{y}^T \mathbf{y}$, transpose of scalar = scalar
  - $\partial f/\partial \mathbf{x} = 2\,\mathbf{A}^T \mathbf{A}\,\mathbf{x} - 2\,\mathbf{A}^T \mathbf{y}$, $\mathbf{A}^T \mathbf{A}$ is symmetric
    $= 0$
  - $\mathbf{x} = (\mathbf{A}^T \mathbf{A})^{-1} \mathbf{A}^T \mathbf{y}$, over determined case with linearly indep cols of $\mathbf{A}$
    $= \mathbf{A}^{-1} \mathbf{y}$, uniquely determined case with invertible $\mathbf{A}$

- For xNN training we usually don't have a simple closed form solution for finding optimal parameters (sadness)

Iterative Methods

- Gradient descent
  - Find $\mathbf{x}$ that minimizes $f(\mathbf{x})$ via moving in the opposite direction of the gradient

- Basic strategy (many many variations exist)
  - Initialization
    - Let $t = 0$
    - Initial guess $\mathbf{x}_0$
  - Iteration
    - Compute the gradient $\nabla f(\mathbf{x}_t)$
    - Select a step size α_t
    - Take a step in the opposite direction of the gradient: $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha_t \nabla f(\mathbf{x}_t)$
    - Let $t = t + 1$

Iterative Methods

- Repeating the closed form example using gradient descent for ease of understanding

- Example: least squares solution to a linear systems of equations
  - Minimize $f(\mathbf{x}) = (\mathbf{A}\,\mathbf{x} - \mathbf{y})^T (\mathbf{A}\,\mathbf{x} - \mathbf{y})$
  - $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A}^T \mathbf{A}\,\mathbf{x} - \mathbf{x}^T \mathbf{A}^T \mathbf{y} - \mathbf{y}^T \mathbf{A}\,\mathbf{x} + \mathbf{y}^T \mathbf{y}$
    $= \mathbf{x}^T \mathbf{A}^T \mathbf{A}\,\mathbf{x} - 2\,\mathbf{y}^T \mathbf{A}\,\mathbf{x} + \mathbf{y}^T \mathbf{y}$
  - $\nabla f(\mathbf{x}) = 2\,\mathbf{A}^T \mathbf{A}\,\mathbf{x} - 2\,\mathbf{A}^T \mathbf{y}$
    $= 2\,\mathbf{A}^T (\mathbf{A}\,\mathbf{x} - \mathbf{y})$
  - $\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha_t \nabla f(\mathbf{x}_t)$
    $= \mathbf{x}_t - \alpha_t\, 2\,\mathbf{A}^T (\mathbf{A}\,\mathbf{x}_t - \mathbf{y})$

Neural Network Training

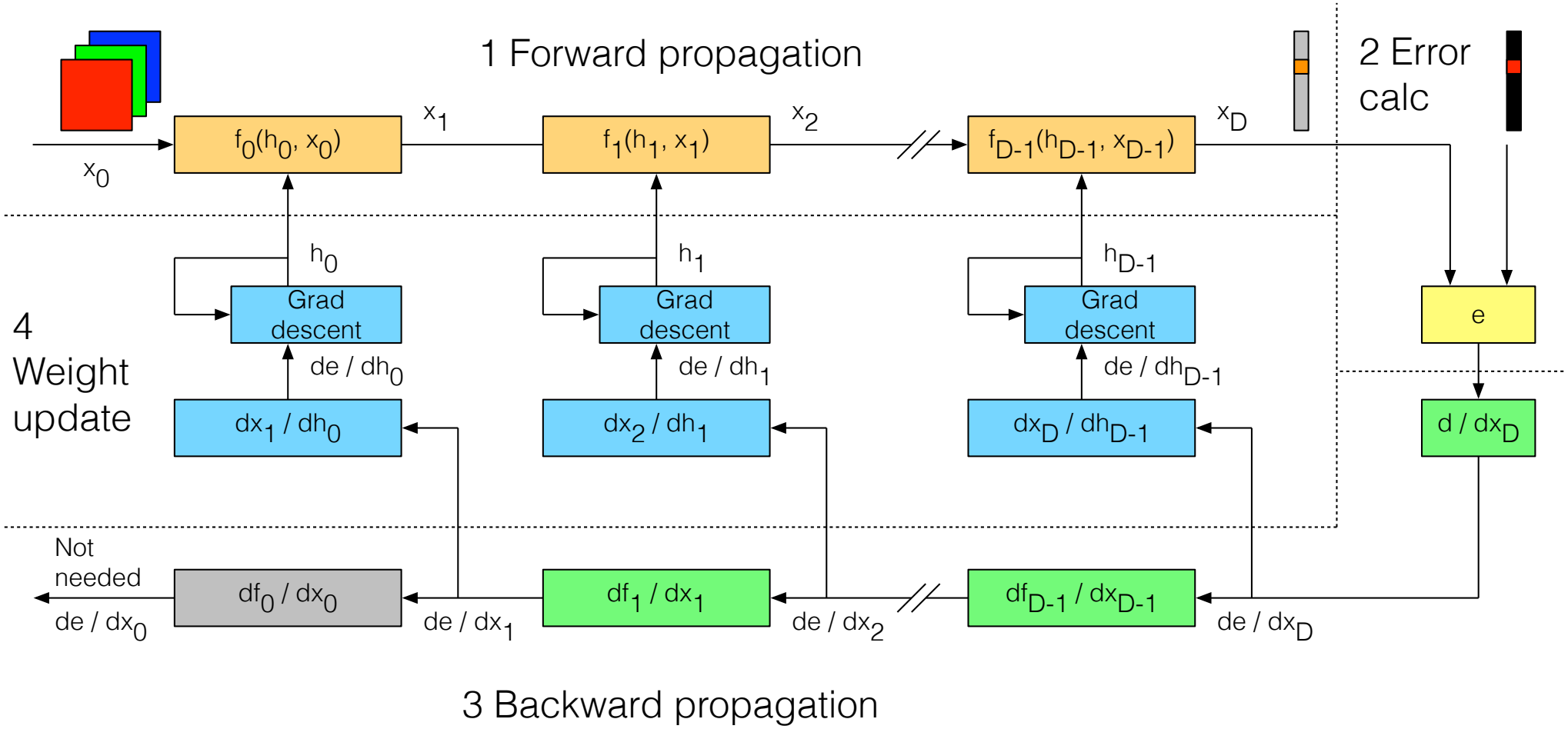
Specifically

- The basics of xNN training are described here
- Many variations to improve training will be introduced in the training lecture
  - It’s ok to start a long trip with a 1st step

Strategy

- Initialization
  - Assume for now, will discuss later
- Forward propagation
- Error calculation
- Backward propagation
  - Partial derivatives and the chain rule
- Weight update
  - Gradient descent (actually the stochastic variant that just uses a batch of inputs at a time)

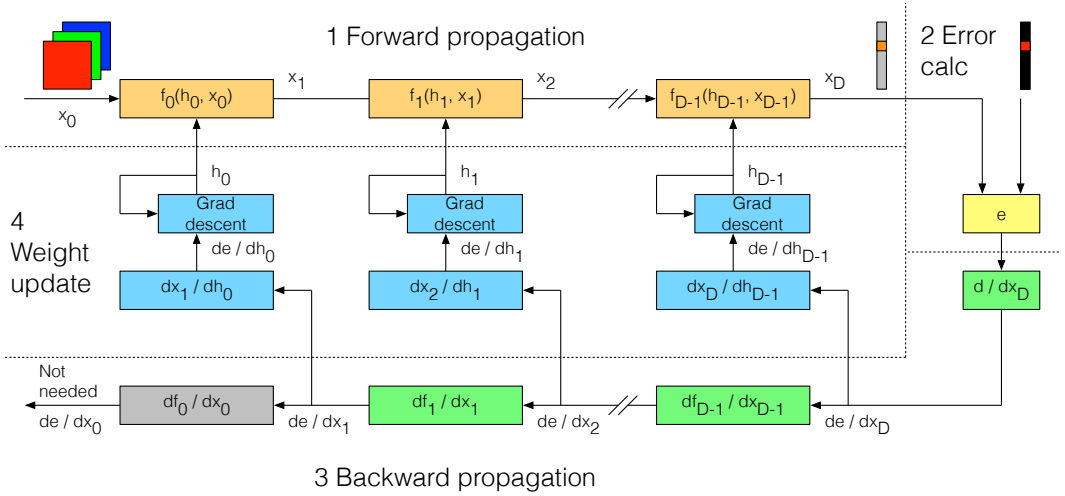
Strategy


1 Forward propagation
2 Error calc
x_0
$f_0(h_0, x_0)$
x_1
$f_1(h_1, x_1)$
x_2
$f_{D-1}(h_{D-1}, x_{D-1})$
x_D
h_0
h_1
h_{D-1}
Grad descent
Grad descent
Grad descent
e
4 Weight update
de / dh_0
de / dh_1
de / dh_{D-1}
dx_1 / dh_0
dx_2 / dh_1
dx_D / dh_{D-1}
d / dx_D
Not needed
de / dx_0
df_0 / dx_0
de / dx_1
df_1 / dx_1
de / dx_2
df_{D-1} / dx_{D-1}
de / dx_D
3 Backward propagation

Forward Propagation

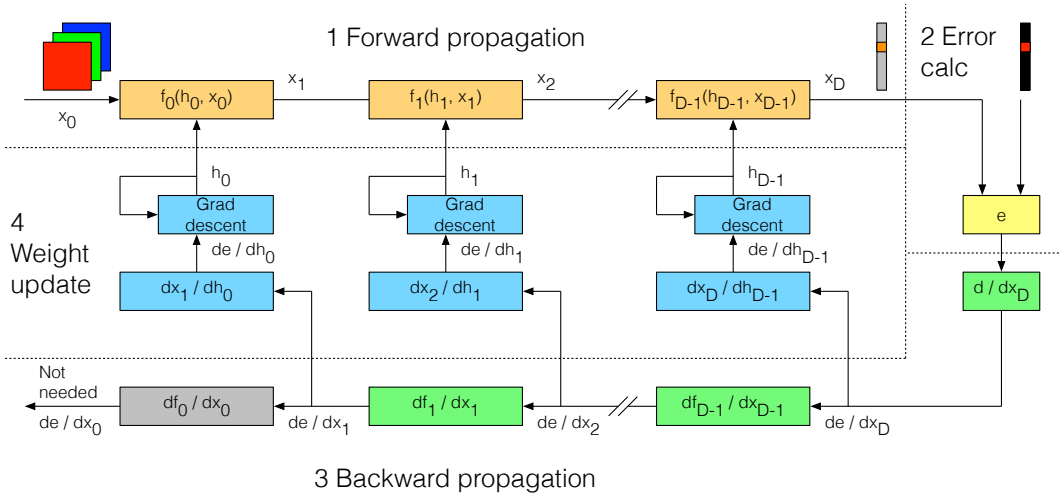
- Use the network to generate a batch of outputs from a batch of inputs (batch SGD)

- Example
  - Image classification to 1 of K classes
  - Network input is an image
  - Network performs some calculations
    - Multiple layers, some layers controlled by parameters that training needs to optimize
  - Network output is a K x 1 vector
    - Each element x(k) corresponds to a class
    - k* is the correct class for the image
    - Network attempts to make x(k) large for k = k* and small for k ≠ k*



Error Calculation

- How well did the network do?
  - Error calculation quantifies an answer to this question computing a single scalar error value for a training input (yes, 1 number)
  - The goal of back propagation and weight update will be to adapt all the parameters in the network to minimize this error
- For the error calculation we'll assume that the ideal output is known
  - Supervised learning
- Consider errors optimized for 2 types of predictions: classification and regression



Error Calculation

- For classification we'll commonly use softmax cross entropy
  - Note that this is really 2 layers but for implementation reasons (the previously mentioned gradient calculation) we'll treat together

- Error calculation
  - Network output is a K x 1 vector $\mathbf{x}_D$
  - $\mathbf{p} = \text{softmax}(\mathbf{x}_D)$
    - $p(k)$ is in $[0, 1]$ and $\Sigma_k\, p(k) = 1$
    - Interpretation of converting the network output to a probability mass function
  - $e = H_{CE}(\mathbf{p}^*, \mathbf{p})$
    - Ideal output $\mathbf{p}^*$ has a 1 at position k* and 0s elsewhere
    - $e = -\Sigma_k\, p^*(k) \log(p(k)) = -\log(p(k^*))$
    - Interpretation of defining the error based on the converted probability for the correct class

Error Calculation

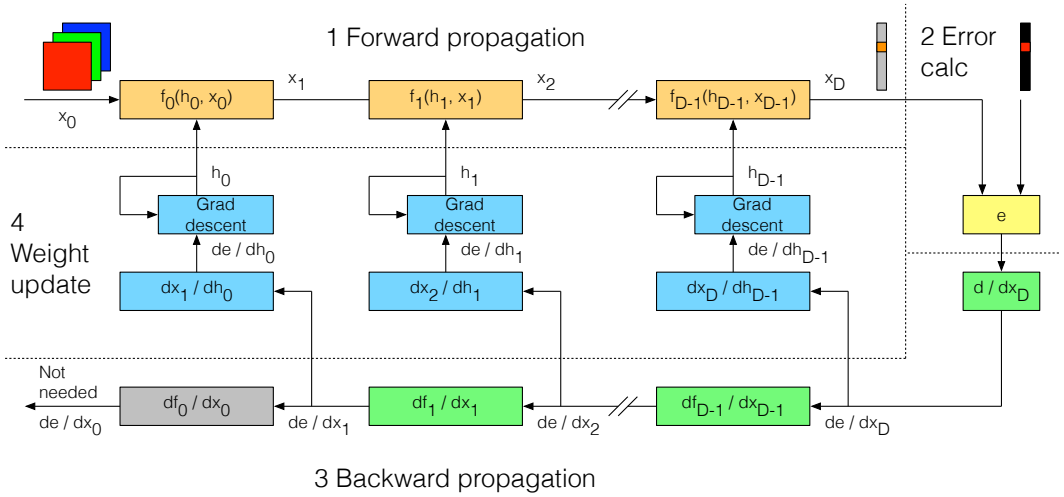
- For regression we'll commonly use 0.5 MSE
  - 0.5 really isn't needed, it just makes equations look more beautiful

- Error calculation
  - Network output is a K x 1 vector $\mathbf{x}_D$
  - e = 0.5 mse($\mathbf{x}^*$, $\mathbf{x}_D$)
    - Ideal output $\mathbf{x}^*$
    - $e = (0.5/K) (\mathbf{x}^* - \mathbf{x}_D)^T (\mathbf{x}^* - \mathbf{x}_D)$
    - Interpretation of defining the error based on the 2 norm distance between the network output and the ideal value

Backward Propagation

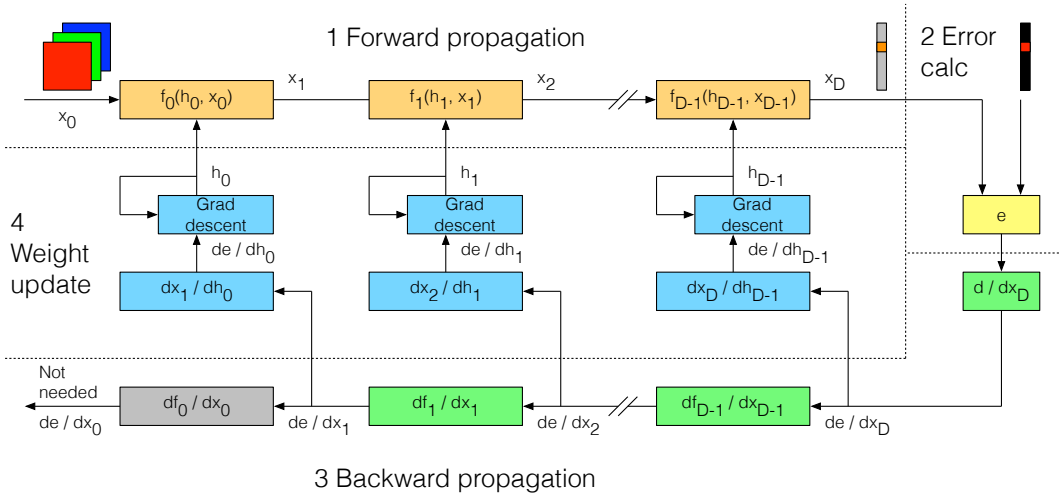
- Goal: determine sensitivity (gradient) of the error with respect to all feature maps

- Automatic differentiation with reverse mode accumulation
  - Decompose forward propagation into a set of building blocks (practically layers, theoretically any primitive operation with a known derivative)
  - Build associated graph for back propagation ~ via reversing arrows and replacing forward propagation nodes representing layers with back propagation nodes representing layer derivatives
  - Back propagation nodes map from $\partial e/\partial \mathbf{x}_d$ to $\partial e/\partial \mathbf{x}_{d-1}$ via the chain rule



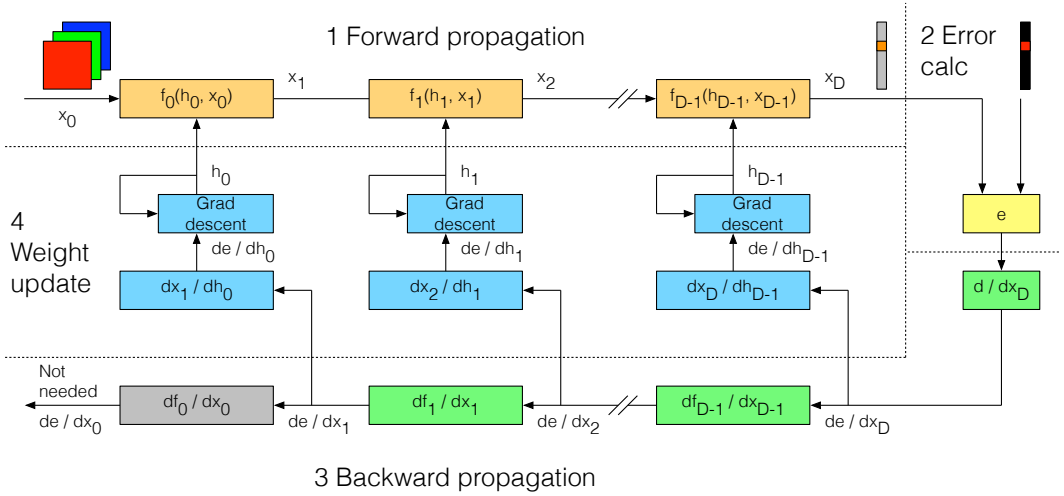
Backward Propagation

- Cookbook
  - Build the backward propagation graph
  - Start with $\partial e/\partial \mathbf{x}_D$, the gradient of the error with respect to the output of the last layer
    - How to calculate this was shown on earlier slides for softmax cross entropy and MSE
  - Compute the gradient of the error at the input of the last layer (the output of the next to last layer)
    - This is found using the chain rule
    - $\partial e/\partial \mathbf{x}_{D-1} = (\partial \mathbf{x}_D/\partial \mathbf{x}_{D-1})\,(\partial e/\partial \mathbf{x}_D)$
      $= (\partial \mathbf{f}_{D-1}/\partial \mathbf{x}_{D-1})\,(\partial e/\partial \mathbf{x}_D)$
    - $\partial e/\partial \mathbf{x}_D$ is given
    - $\partial \mathbf{f}_{D-1}/\partial \mathbf{x}_{D-1}$ for many common layers was shown on an earlier slide
  - Compute the gradient of the error at the input of the next to last layer ... (and repeat)



Weight Update

- Goal: update parameters to reduce the error
- Strategy
  - Use the error gradient with respect to the feature map output for the current layer provided via back prop with the derivative of the forward node output with respect to the params to determine the error gradient with respect to the params (1)
  - Then use gradient descent to adapt the parameters to reduce the error (2)
- In math (d – 1 is the layer, t is the time step)
  - 1: $\partial e/\partial \mathbf{h}_{d-1} = (\partial e/\partial \mathbf{x}_d)(\partial \mathbf{x}_d/\partial \mathbf{h}_{d-1}) = (\partial e/\partial \mathbf{x}_d)(\partial \mathbf{f}_{d-1}/\partial \mathbf{h}_{d-1})$
  - 2: $\mathbf{h}_{t+1,d-1} = \mathbf{h}_{t,d-1} - \alpha_t \partial e/\partial \mathbf{h}_{t,d-1}$



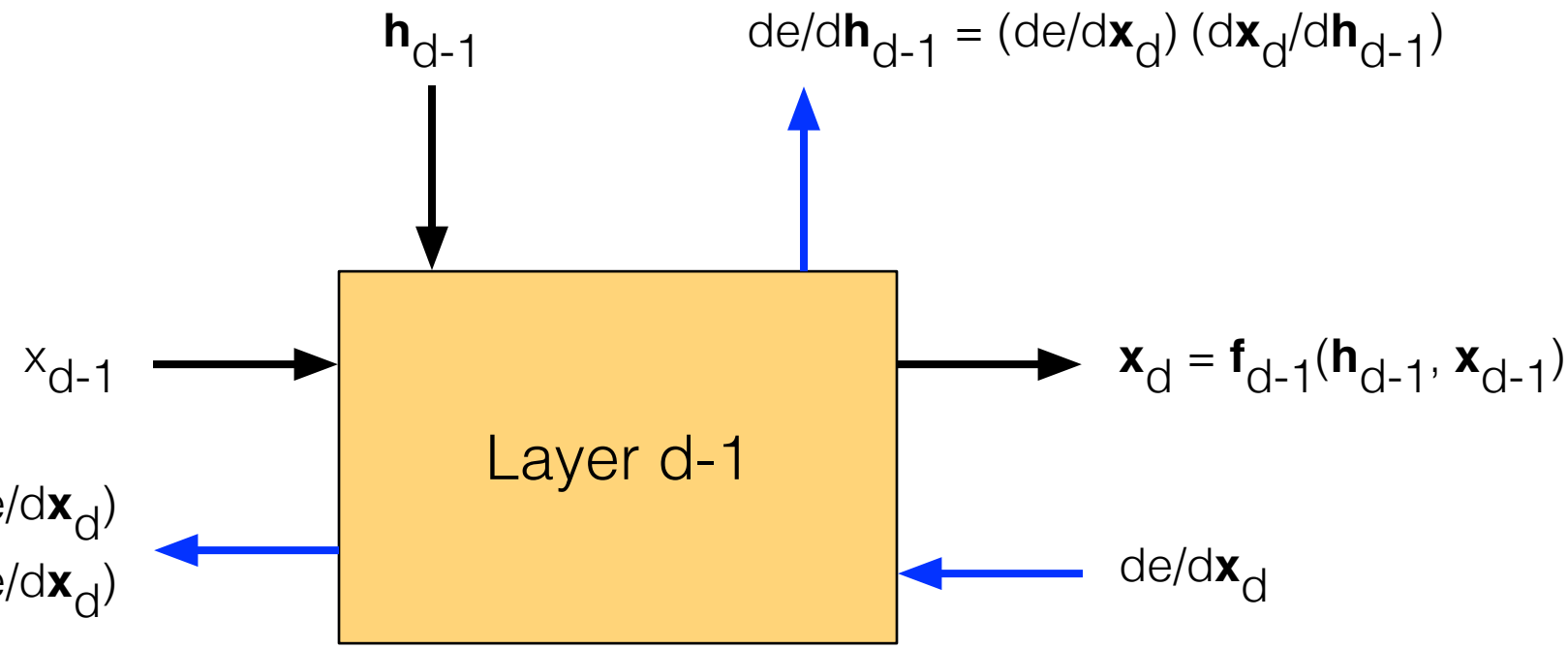
Summary For 1 Layer

The differentials in this figure should be partial differentials – a limitation wrt my abilities in figure creation application I use ...

This is important

If a layer in the forward path of a xNN is differentiable with respect to inputs and parameters, then error gradients can be back propagated through the layer and weights can be updated

If every layer in the forward path satisfies this property then the network can be trained end to end



$$de/d\mathbf{x}_{d-1} = (d\mathbf{x}_d/d\mathbf{x}_{d-1})\,(de/d\mathbf{x}_d)$$
$$= (d\mathbf{f}_{d-1}/d\mathbf{x}_{d-1})\,(de/d\mathbf{x}_d)$$

Summary For 1 Layer

- Example: matrix vector multiplication (dimensions (M x 1) = (M x K) (K x 1))
  - $\mathbf{x}_d = f_{d-1}(\mathbf{H}_{d-1}, \mathbf{x}_{d-1}) = \mathbf{H}_{d-1}\,\mathbf{x}_{d-1}$
- Error gradient with respect to the feature map (dimensions (K x 1) = (K x M) (M x 1))
  - Given: $\partial e/\partial \mathbf{x}_d$
  - Compute: $\partial e/\partial \mathbf{x}_{d-1} = (\partial \mathbf{x}_d/\partial \mathbf{x}_{d-1})\,(\partial e/\partial \mathbf{x}_d) = (\partial \mathbf{f}_{d-1}/\partial \mathbf{x}_{d-1})\,(\partial e/\partial \mathbf{x}_d) = (\mathbf{H}_{d-1})^T\,(\partial e/\partial \mathbf{x}_d)$
- Error gradient with respect to the filter coefficients (dimensions (M x K) = (M x 1) (1 x K))
  - Given: $\partial e/\partial \mathbf{x}_d$
  - Compute: $\partial e/\partial \mathbf{H}_{d-1} = (\partial e/\partial \mathbf{x}_d)\,(\partial \mathbf{x}_d/\partial \mathbf{H}_{d-1}) = (\partial e/\partial \mathbf{x}_d)\,(\partial \mathbf{f}_{d-1}/\partial \mathbf{H}_{d-1}) = (\partial e/\partial \mathbf{x}_d)\,(\mathbf{x}_{d-1})^T$
    - Note the reversing of order of terms for this case (can show from bookkeeping)
    - Note that this implies saving $\mathbf{x}_{d-1}$ from the forward pass; this has consequences with respect to memory
- Weight update (time t to time t + 1)
  - Compute: $\mathbf{H}_{t+1,d-1} = \mathbf{H}_{t,d-1} - \alpha_t\,\partial e/\partial \mathbf{H}_{t,d-1}$

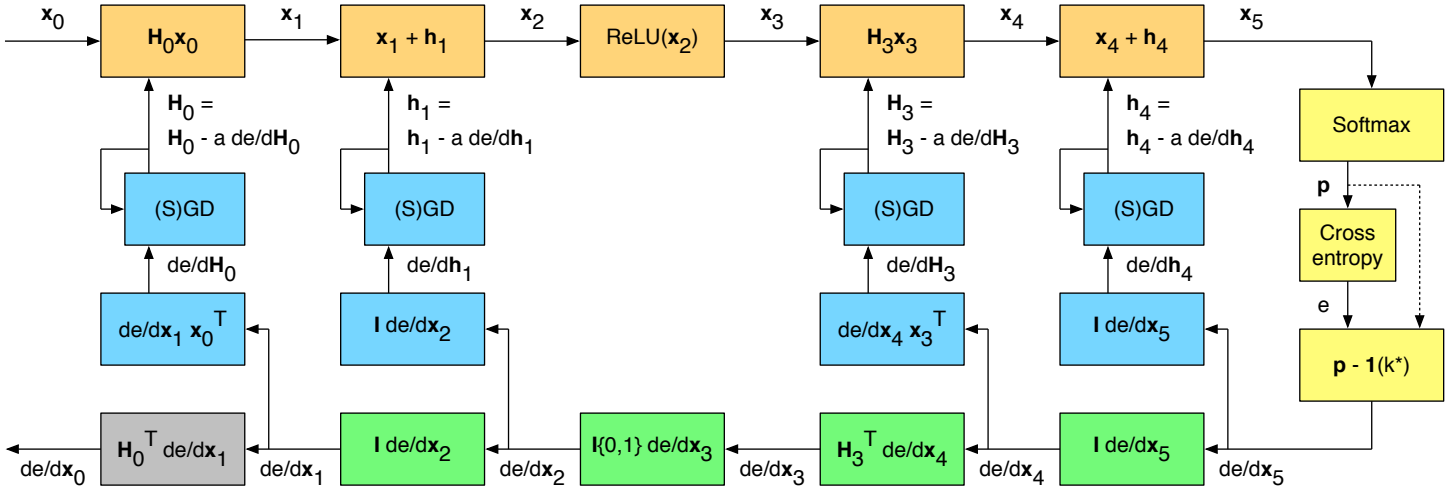
Training Vs Standard Optimization

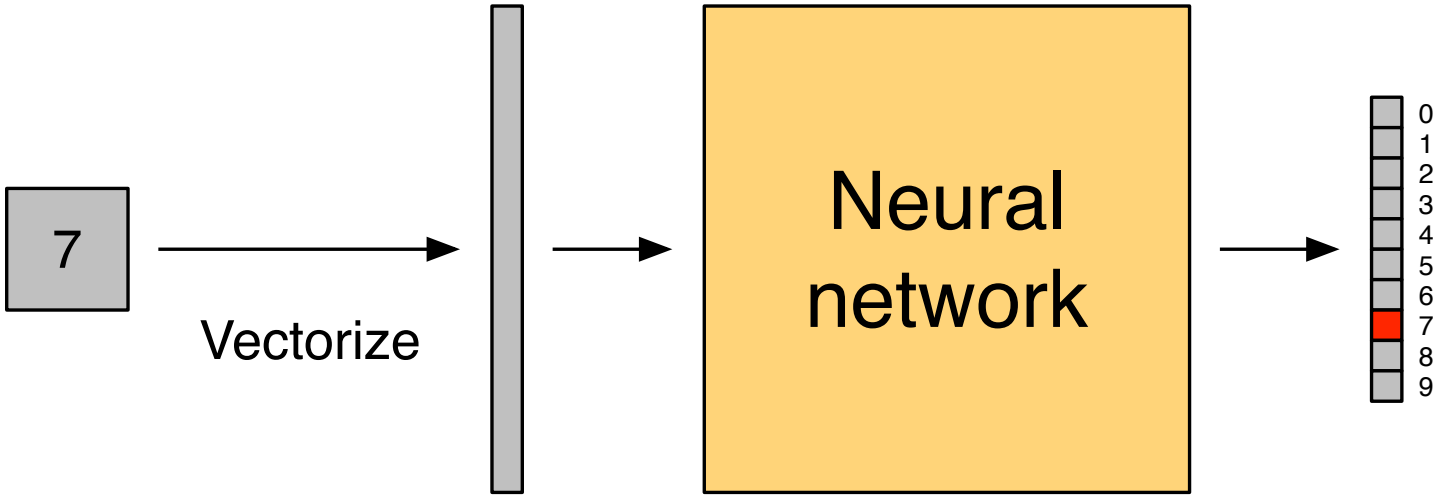
- There are a number of differences between xNN training and optimization
  - Optimize an error function with the training data to perform well on testing data
    - Typically thought of as optimizing for empirical vs true distribution
    - The concept of an input distribution is used as there's typically not a practical deterministic model for the input
  - Use a surrogate error function vs actual error function
    - E.g., softmax + cross entropy vs arg max
    - Also don't use the actual top 1 for all training data at 1x
      - Only use a fraction or batch of it at a time

- These differences highlight the need for generalization
  - Want to train xNNs on training data with a surrogate error function such that they will generalize well to new data with high levels of accuracy
  - The training lecture will look at various regularization methods to improve generalization

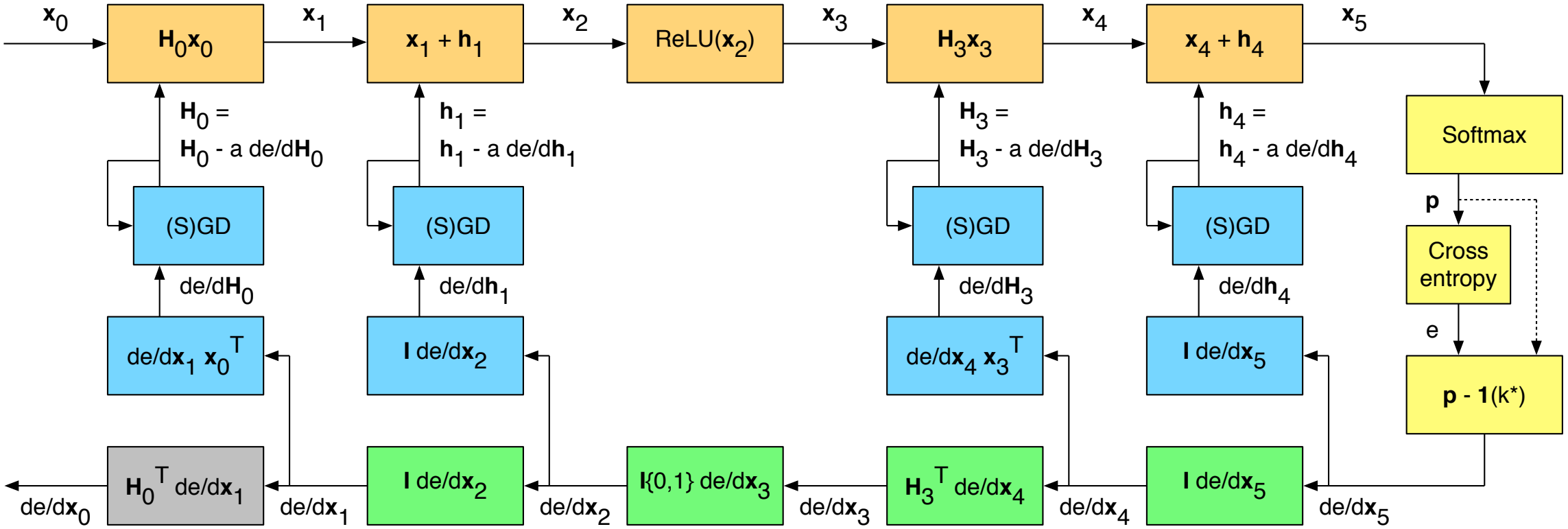
Example

- MNIST digit recognition
  - The hello, world! of machine learning
- Simple network design to illustrate training (not optimal for accuracy)
  - $x_5 = H_3\ \mathrm{ReLU}_2(H_0\, x_0 + h_1) + h_4$
- Goal
  - Make arg max error of testing small
  - Proxy of making training error e small
- What controls the proxy error e
  - For this topology H_0, h_1, H_3 and h_4
- How to adapt H_0, h_1, H_3 and h_4 to make the proxy error e small
  - (Stochastic) gradient descent
- How to find the values (S)GD needs for the update
  - Back propagation





Example


x0
H0x0
x1
x1 + h1
x2
ReLU(x2)
x3
H3x3
x4
x4 + h4
x5
Softmax
H0 = H0 - a de/dH0
h1 = h1 - a de/dh1
H3 = H3 - a de/dH3
h4 = h4 - a de/dh4
(S)GD
(S)GD
(S)GD
(S)GD
de/dH0
de/dh1
de/dH3
de/dh4
p
Cross entropy
e
p - 1(k*)
de/dx1 x0T
I de/dx2
de/dx4 x3T
I de/dx5
H0T de/dx1
I de/dx2
I{0,1} de/dx3
H3T de/dx4
I de/dx5
de/dx0
de/dx1
de/dx2
de/dx3
de/dx4
de/dx5

Universal Function Approximation

Definition

- A 3 layer neural network with ReLU nonlinearity can approximate arbitrarily closely any continuous function on a compact subset of R^K

- Example: classification
  - The goal is to create a function that maps input images to a 1 hot vector at the output indicating class membership
  - If a function exists then it can be approximated via a neural network
  - Lots of other problems can be cast as finding an appropriate function to map from inputs to outputs

Why Universal Function Approx Matters

- Neural networks are a general tool to solve all sorts of problems
  - Theoretically we don't need to learn a million (slight exaggeration of the number of techniques taught in an intro ML course) different methods for information extraction and data generation
  - Theoretically we can apply neural networks and if a solution (with described constraints) exists then a neural network can approximate it
  - This is important

- Things not said
  - That a function exists that maps inputs to outputs
  - That a 3 layer neural network is the best way to approximate the function
  - How to train the neural network from training data to perform well on testing data

Why Discuss Universal Approximation Here

- It makes sense to discuss approximation within the context of calculus
  - Limits, bounds on derivatives, ...
  - And there's not a separate analysis lecture

- It's also nice to think about universal approximation before getting into network design

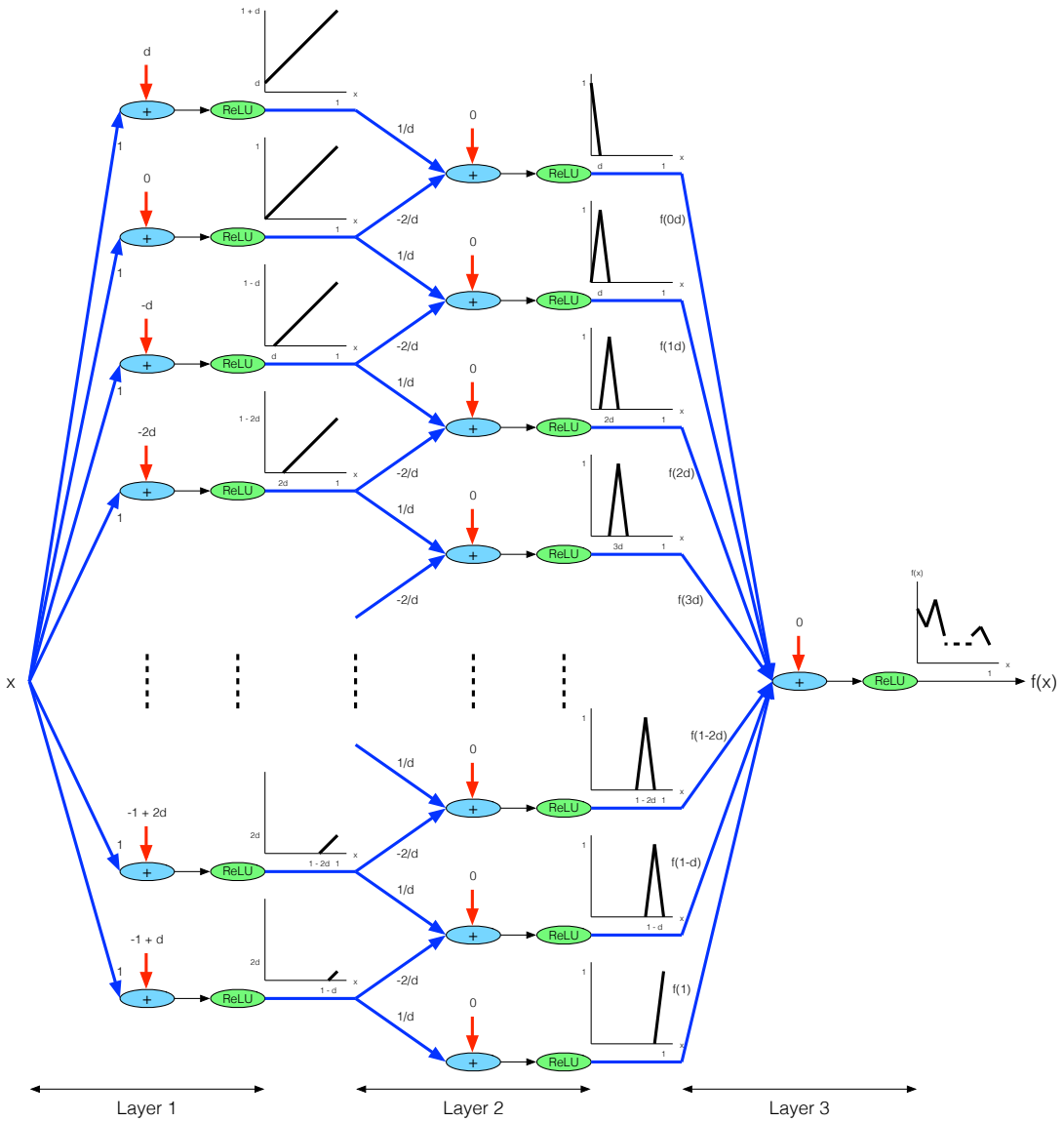
A Simple Constructive Proof

- Starting point
  - $x \in [0, 1]$
  - $f(x) \geq 0$
  - $f(x) < \infty$
  - f(x) continuous

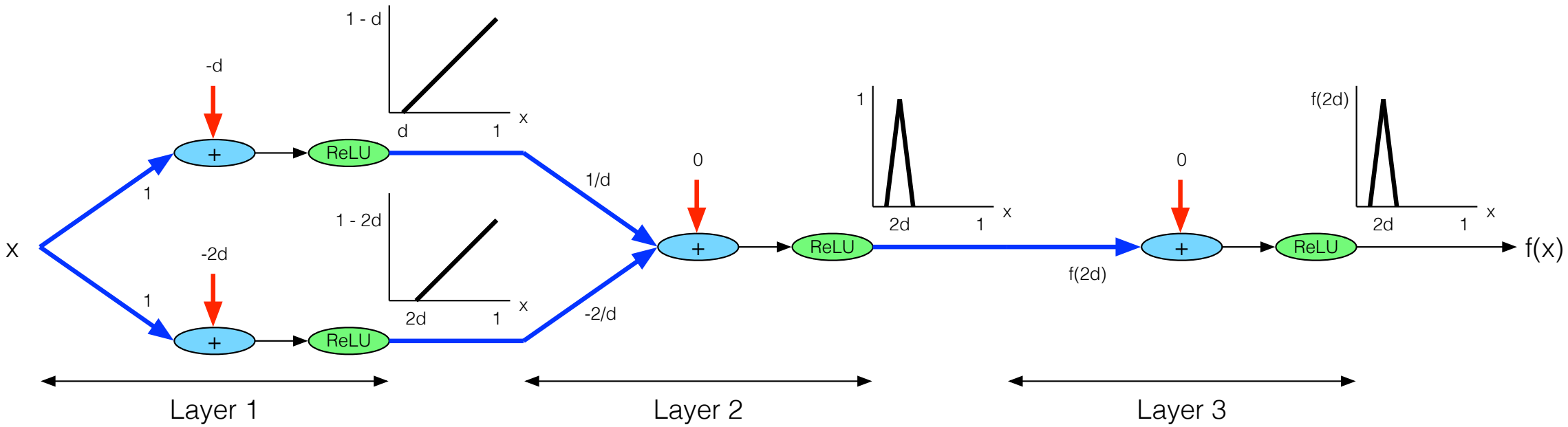
- A constructive “proof” (exceedingly loosely speaking) that a countably infinitely wide 3 layer (input, hidden, output) neural network can implement the above f(x) is shown on the next slide

Proof Outline

- Layer 1 identity weight matrix (blue), shifting bias (red) and ReLU (green) create 1/d + 1 building block segments
- Layer 2 creates overlapped triangles of height 1 and width 2d centered at multiples of d
- Layer 3 scales the triangle at nd by f(nd) and adds all of the triangles together to realize f(x) for $x \in [0, 1]$
- Let $d \rightarrow 0$



Zooming In On 1 Triangle For Clarity



Removing Simplifications

- $x_{min} \leq x \leq x_{max}$ where x_{min} and x_{max} are finite and not limited to [0, 1]
  - Select initial bias values as $x_{min} + d$, x_{min}, …, $-x_{max} + d$
  - Use associated locations for layer 3 weights
- f(x) has a countably infinite number of discontinuities
  - Include a countably infinite number of additional branches each centered at the discontinuity
- $f_{min} \leq f(x) \leq f_{max}$ where f_{min} and f_{max} are finite and can each be negative, 0 or positive
  - Approximate $f(x) + f_{min}$
  - At the last bias add $-f_{min}$ and do not apply the last ReLU
- Vector input $\mathbf{x} \in R^K$
  - Replicate the 1st layer for each input component x(k)
  - Combine them in the 2nd layer to produce K dimensional pyramids
- Vector output $\mathbf{y} = f(\mathbf{x}) \in R^M$
  - Replicate the 3rd layer for each output component y(m)

Comments

- Complexity
  - The more complex the function is with respect to changes the wider this network needs to be to approximate the function at a given level of accuracy
  - Complexity is multiplicative in input and output dimension

- Depth
  - More layers allow more complex basis shapes to be created as the combination of simple basis shapes
  - This typically greatly reduces the width requirements of the network
  - These benefits of depth can be shown in theory and are observed in practice
  - Ex: 20 nodes in 1 layer provides 20 paths from input to output, 20 nodes in 2 layers each with 10 nodes allows 100 paths from input to output

Alternative Views

- Different proofs of the same result
  - Different structures
  - Different activations
  - Different basis
  - More rigor ...
- Proofs of different results
  - xNNs are universal Boolean functions
  - xNNs are universal classifiers
- Lots of similar conclusions
  - Universality and depth matters
- See the references for additional information

References

Derivatives And Optimization

- Personal communication with T. Lahou
- The matrix cookbook
  - https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf
- Calculus
  - http://mathonline.wikidot.com/calculus
- Essence of calculus
  - https://www.youtube.com/playlist?list=PLZHQObOWTQDMsr9K-rj53DwVRMYO3t5Yr
- Convex optimization
  - https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf

Neural Network Training

- The softmax function and its derivative
  - https://eli.thegreenplace.net/2016/the-softmax-function-and-its-derivative/
- Learning representations by back-propagating errors
  - https://www.nature.com/articles/323533a0
- Automatic differentiation in machine learning: a survey
  - https://arxiv.org/abs/1502.05767

Universal Function Approximation

- The complexity of learning
  - https://page.mi.fu-berlin.de/rojas/neural/chapter/K10.pdf
- Neural networks: what can a network represent
  - http://deeplearning.cs.cmu.edu/slides/lec2.universal.pdf
- Approximation by superpositions of a sigmoidal function
  - http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.441.7873&rep=rep1&type=pdf

Universal Function Approximation

- Nearly-tight VC-dimension and pseudodimension bounds for piecewise linear neural networks
  - https://arxiv.org/abs/1703.02930
- A capacity scaling law for artificial neural networks
  - https://arxiv.org/abs/1708.06019
- The expressive power of neural networks: a view from the width
  - https://arxiv.org/abs/1709.02540
- Approximating continuous functions by ReLU nets of minimal width
  - https://arxiv.org/abs/1710.11278
- Neural networks should be wide enough to learn disconnected decision regions
  - https://arxiv.org/abs/1803.00094
- ResNet with one-neuron hidden layers is a universal approximator
  - https://arxiv.org/abs/1806.10909
- Small ReLU networks are powerful memorizers: a tight analysis of memorization capacity
  - https://arxiv.org/abs/1810.07770
- On exact computation with an infinitely wide neural net
  - https://arxiv.org/abs/1904.11955