

# Calculus

Arthur J. Redfern

[arthur.redfern@utdallas.edu](mailto:arthur.redfern@utdallas.edu)

Jan 28, 2019

Jan 30, 2019

# Outline

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

# Motivation

# Optimization And Approximation

- Neural network training is an optimization problem that we solve using calculus (math highlights listed, obviously others also present)
  - Forward propagation - linear algebra and algorithms\*
  - Error calculation - probability\*
  - Backward propagation - calculus
  - Weight update - calculus
- Neural networks are universal 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

\* Will be covered in a later lecture; however, the use here should be straightforward enough to understand

# Derivatives

# Scalar Functions Of A Single Variable

- Scalar functions of a single variable:  $y = f(x)$ 
  - $f: \mathbb{R} \rightarrow \mathbb{R}$
- 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  $x$

# Scalar Functions Of A Single Variable

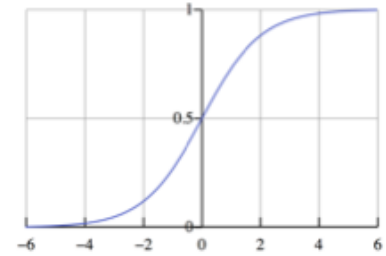
- Differentiable functions
  - At values of  $x$  where the limit exists  $f$  is differentiable
- Non differentiable functions
  - At values of  $x$  where the limit does not exist  $f$  is not differentiable
  - If there are a countable number of non differentiable values then it's potentially possible 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

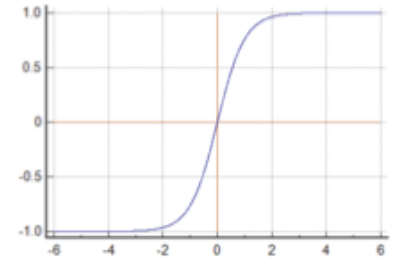
- Example: sigmoid

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



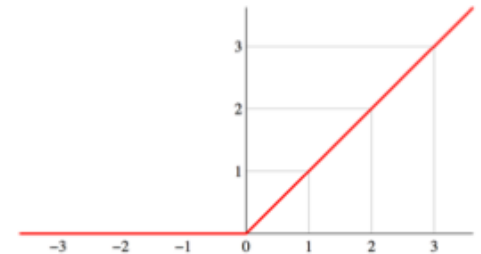
- Example: tanh

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



- Example: ReLU

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





# 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:  $y = f(\mathbf{x})$ 
  - $f: \mathbb{R}^K \rightarrow \mathbb{R}$
  - $\mathbf{x} = [x(0), x(1), \dots, x(K-1)]^T$
- 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
  - $K \times 1$  vector of partial derivatives with respect to each variable
  - $\nabla f(\mathbf{x}) = [(\partial f / \partial x(0)), (\partial f / \partial x(1)), \dots, (\partial f / \partial x(K - 1))]^T$
- 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 used in network error calculation,  $\mathbf{x}^*$  is used to denote the ideal value

- 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

- $f(\mathbf{x}^*, \mathbf{x}) = -\sum_k x^*(k) \log(x(k)),$  we'll talk about this in the probability lecture  
 $= -\log(x(k^*)),$  if  $\mathbf{x}^*$  has a 1 at position  $k^*$  and 0s elsewhere
- $\partial f / \partial \mathbf{x} = -1/(x(k^*)),$  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  $\partial f / \partial \mathbf{x}$  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:  $\mathbf{y} = f(\mathbf{x})$ 
  - $f: \mathbb{R}^K \rightarrow \mathbb{R}^M$
- Jacobian
  - Using numerator notation the Jacobian is a  $M \times K$  matrix with entry  $(m, k)$  corresponding to the partial derivative of output  $f(m)$  with respect to input variable  $x(k)$
  - $J_f(\mathbf{x}) = \begin{bmatrix} \partial f(0)/\partial x(0), & \dots, & \partial f(0)/\partial x(K-1) \\ \vdots, & \dots, & \vdots \\ \partial f(M-1)/\partial x(0), & \dots, & \partial f(M-1)/\partial x(K-1) \end{bmatrix}$
- 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  $\mathbf{a}$ 
    - $f(\mathbf{x}) \approx f(\mathbf{a}) + J_f(\mathbf{a})(\mathbf{x} - \mathbf{a})$

# Vector Functions Of Multiple Variables

These are common linear transforms that comprise key xNN layers

- Example: matrix vector multiplication
  - $f(\mathbf{A}, \mathbf{x}) = \mathbf{A} \mathbf{x}$
  - $\partial \mathbf{f} / \partial \mathbf{x} = \mathbf{A}^T$
  - $\partial \mathbf{f} / \partial \mathbf{A} = \mathbf{x}^T$ , note:  $(f(\mathbf{A}, \mathbf{x}))^T = \mathbf{x}^T \mathbf{A}^T$ ,  $\partial \mathbf{f}^T / \partial \mathbf{A}^T = \mathbf{x}$ ,  $\partial \mathbf{f} / \partial \mathbf{A} = \mathbf{x}^T$
- Example: matrix matrix multiplication
  - $f(\mathbf{A}, \mathbf{X}) = \mathbf{A} \mathbf{X}$
  - Via matrix vector multiplication with vectorized  $\mathbf{X}$  and block diagonal  $\mathbf{A}$
- Example: CNN style 2D convolution
  - $f(\mathbf{H}^{\text{No} \times \text{Ni} \times \text{Fr} \times \text{Fc}}, \mathbf{X}^{\text{Ni} \times \text{Lr} \times \text{Lc}}) = \mathbf{H}^{\text{No} \times \text{Ni} \times \text{Fr} \times \text{Fc}} \circledast \mathbf{X}^{\text{Ni} \times \text{Lr} \times \text{Lc}}$
  - Via CNN style 2D convolution with  $F_r \times F_c$  filters decomposed into a sum of 1x1 filters with corresponding feature map shifts leading to a sum of matrix matrix multiplications

# Vector Functions Of Multiple Variables

These are common elementwise operations

- Example: elementwise addition

- $f(\mathbf{A}, \mathbf{X}) = \mathbf{X} + \mathbf{A}$
- $\partial f / \partial \mathbf{X} = \mathbf{I}$
- $\partial f / \partial \mathbf{H} = \mathbf{I}$

- Example: elementwise multiplication

- $f(\mathbf{H}, \mathbf{X}) = \mathbf{H} \odot \mathbf{X}$
- $\partial f / \partial \mathbf{X} = \mathbf{H}^\top$
- $\partial f / \partial \mathbf{H} = \mathbf{X}^\top$

- Example: fixed scale

- $f(\mathbf{X}) = \alpha \mathbf{X}$
- $\partial f / \partial \mathbf{X} = \alpha \mathbf{I}$

- Example: sigmoid, tanh and ReLU

- Conceptually a pointwise application of scalar derivative rules forms a diagonal matrix
- Though in practice for efficiency only the diagonal values are generated



# Vector Functions Of Multiple Variables

These are common operations that can be considered from a structural perspective

- Example: average pooling
  - Divide the gradient at the output equally to form the gradient at all the inputs
- 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
- Example: split
  - Add all gradients at the output to form the gradient at the input

# 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

- $\mathbf{f}(\mathbf{x}) = (1 / (\sum_k e^{x^{(k)}})) [e^{x^{(0)}}, e^{x^{(1)}}, \dots, e^{x^{(K-1)}}]^T$

- $\partial \mathbf{f} / \partial \mathbf{x} = \begin{bmatrix} f(0)(1-f(0)), & -f(1)f(0), & \dots, & -f(K-1)f(0) \\ -f(0)f(1), & f(1)(1-f(1)), & & -f(K-1)f(1) \\ \vdots, & \vdots, & \ddots, & \vdots \\ -f(0)f(K-1), & -f(1)f(K-1), & & f(K-1)(1-f(K-1)) \end{bmatrix}$

# Vector Functions Of Multiple Variables

This is commonly used in classification network training

- Chain rule
  - $h(\mathbf{x}) = f(g(\mathbf{x}))$  and define  $\mathbf{u} = g(\mathbf{x})$
  - $\partial h / \partial \mathbf{x} = (\partial \mathbf{u} / \partial \mathbf{x}) (\partial f / \partial \mathbf{u})$ , 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  $\mathbf{u} = g(\mathbf{x})$
  - $f(\mathbf{p}) = H_{\text{CE}}(\mathbf{p}^*, \mathbf{p})$ , like  $f(\mathbf{u})$
  - $e(\mathbf{x}) = H_{\text{CE}}(\mathbf{p}^*, \text{softmax}(\mathbf{x}))$ , 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
  - $= [p(0), \dots, p(k^*-1), (p(k^*)-1), p(k^*+1), \dots, p(K-1)]^T$

# Optimization

# Setup

- Problem
  - Minimize  $f_0(\mathbf{x})$ , objective function
  - Subject to  $f_c(\mathbf{x}) \leq b_c$ ,  $c = 1, \dots, C$ , 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:  $f_i(\alpha \mathbf{x}_0 + \beta \mathbf{x}_1) \leq \alpha f_i(\mathbf{x}_0) + \beta f_i(\mathbf{x}_1)$ ,  $i = 0, 1, \dots, C$ ,  $\alpha + \beta = 1$ ,  $\alpha \geq 0$ ,  $\beta \geq 0$
  - Linear:  $f_i(\alpha \mathbf{x}_0 + \beta \mathbf{x}_1) = \alpha f_i(\mathbf{x}_0) + \beta f_i(\mathbf{x}_1)$ ,  $i = 0, 1, \dots, C$

# Critical Points

- Critical points
  - For a scalar function of multiple variables a critical point is a value  $\mathbf{x}$  at which all partial derivatives of  $f(\mathbf{x})$  are 0
  - Local extrema of  $f(\mathbf{x})$  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  $\mathbf{x}$
  - In xNNs we have inputs  $\mathbf{x}$  and optimize over parameters  $\mathbf{h}$  (or some other variable)
  - So in the optimization section our goal is to find the optimal  $\mathbf{x}$ , and in the xNN training section it's to find the optimal  $\mathbf{h}$  (or other parameter variable)

# Closed Form Methods

- Example: least squares solution to a linear systems of equations
  - Minimize  $f(\mathbf{x}) = (\mathbf{A} \mathbf{x} - \mathbf{y})^\top (\mathbf{A} \mathbf{x} - \mathbf{y})$
  - Multiply out, take partial derivatives, set to 0 and solve
  - $f(\mathbf{x}) = \mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x} - \mathbf{x}^\top \mathbf{A}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{A} \mathbf{x} + \mathbf{y}^\top \mathbf{y}$   
 $= \mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x} - 2 \mathbf{y}^\top \mathbf{A} \mathbf{x} + \mathbf{y}^\top \mathbf{y},$  transpose of scalar = scalar
  - $\partial f / \partial \mathbf{x} = 2 \mathbf{A}^\top \mathbf{A} \mathbf{x} - 2 \mathbf{A}^\top \mathbf{y},$   $\mathbf{A}^\top \mathbf{A}$  is symmetric  
 $= 0$
  - $\mathbf{x} = (\mathbf{A}^\top \mathbf{A})^{-1} \mathbf{A}^\top \mathbf{y},$  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  $\alpha_t$
    - 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  $f(\mathbf{x}) = (\mathbf{A} \mathbf{x} - \mathbf{y})^\top (\mathbf{A} \mathbf{x} - \mathbf{y})$
  - $f(\mathbf{x})$ 

$$= \mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x} - \mathbf{x}^\top \mathbf{A}^\top \mathbf{y} - \mathbf{y}^\top \mathbf{A} \mathbf{x} + \mathbf{y}^\top \mathbf{y}$$

$$= \mathbf{x}^\top \mathbf{A}^\top \mathbf{A} \mathbf{x} - 2 \mathbf{y}^\top \mathbf{A} \mathbf{x} + \mathbf{y}^\top \mathbf{y}$$
  - $\nabla f(\mathbf{x})$ 

$$= 2 \mathbf{A}^\top \mathbf{A} \mathbf{x} - 2 \mathbf{A}^\top \mathbf{y}$$

$$= 2 \mathbf{A}^\top (\mathbf{A} \mathbf{x} - \mathbf{y})$$
  - $\mathbf{x}_{t+1}$ 

$$= \mathbf{x}_t - \alpha_t \nabla f(\mathbf{x}_t)$$

$$= \mathbf{x}_t - \alpha_t 2 \mathbf{A}^\top (\mathbf{A} \mathbf{x}_t - \mathbf{y})$$

# Neural Network Training

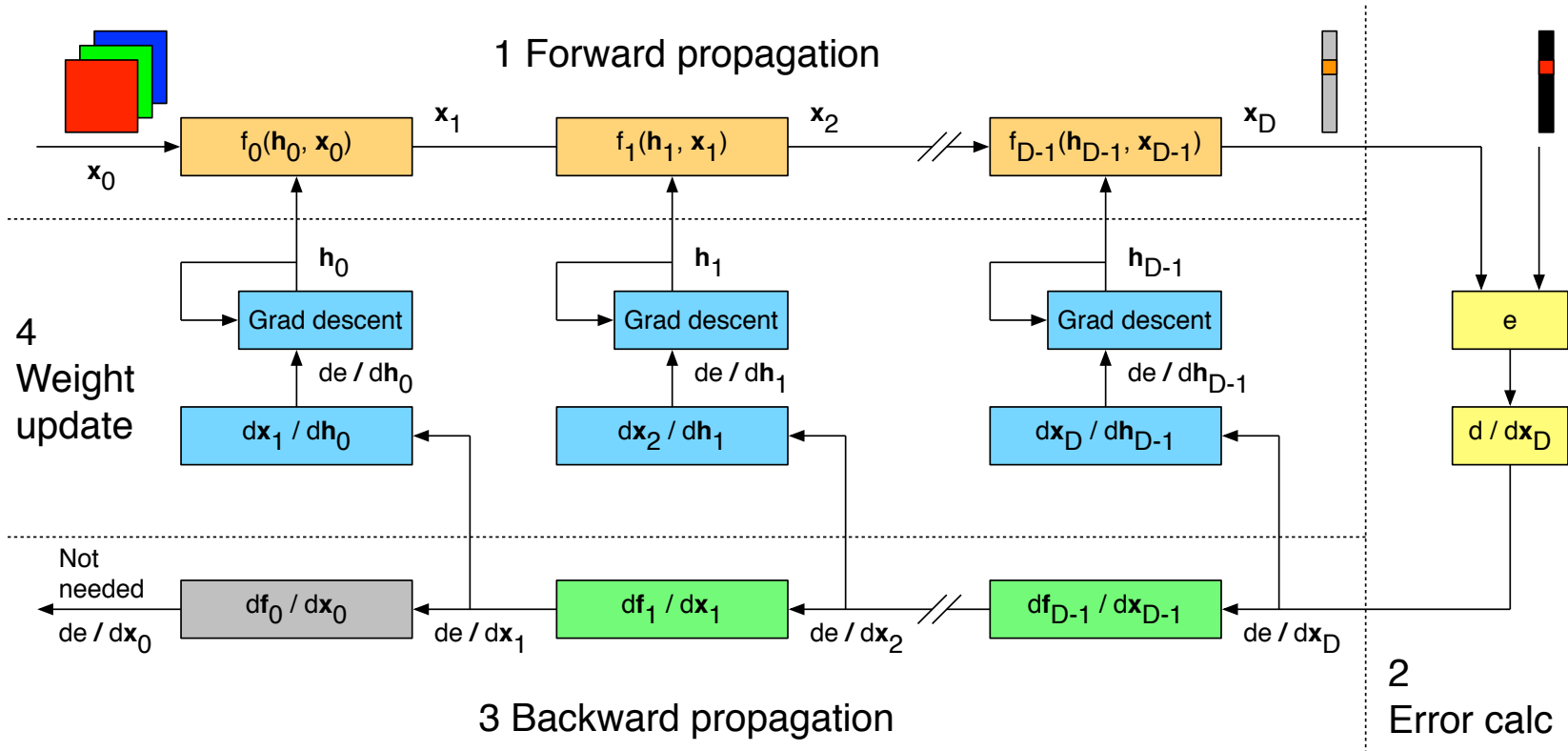
# Specifically

- 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

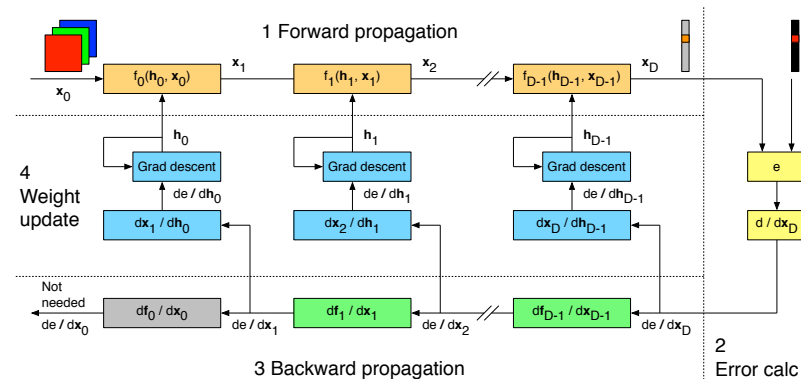


# Forward Propagation

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

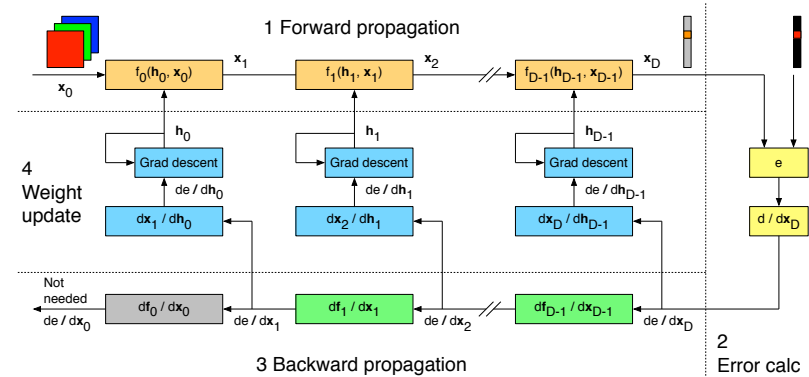
- Example

- 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  $K \times 1$  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  $k \neq k^*$



# 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  $K \times 1$  vector  $\mathbf{x}_D$
  - $\mathbf{p} = \text{softmax}(\mathbf{x}_D)$ 
    - $p(k)$  is in  $[0, 1]$  and  $\sum_k p(k) = 1$
    - 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
    - $e = -\sum_k p^*(k) \log(p(k)) = -\log(p(k^*))$
    - Interpretation of defining the error based on the converted probability for the correct class

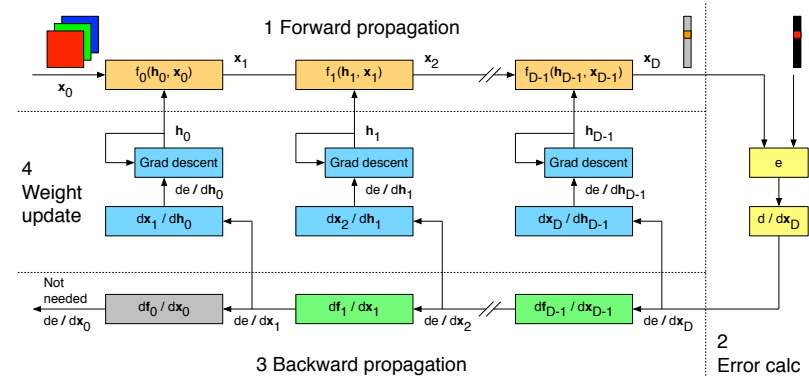


# Error Calculation

- 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  $K \times 1$  vector  $\mathbf{x}_D$
  - $e = 0.5 \text{ mse}(\mathbf{x}^*, \mathbf{x}_D)$ 
    - 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

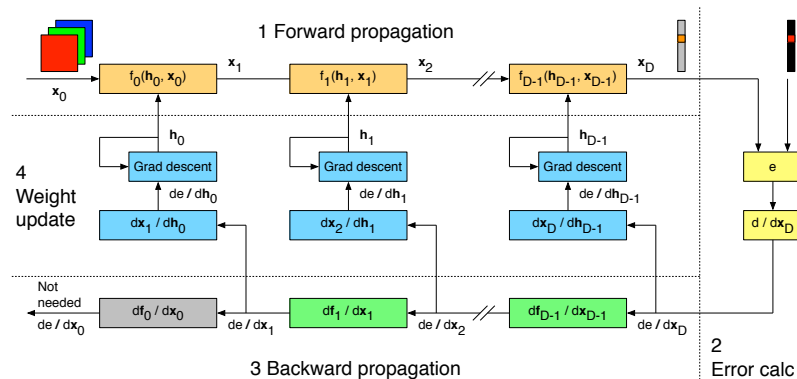
- Goal: determine sensitivity (gradient) of the error with respect to all feature maps
- Automatic differentiation with reverse mode accumulation
  - Decompose forward propagation into a set of building blocks (practically layers, theoretically any primitive operation with a known derivative)
  - Build 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

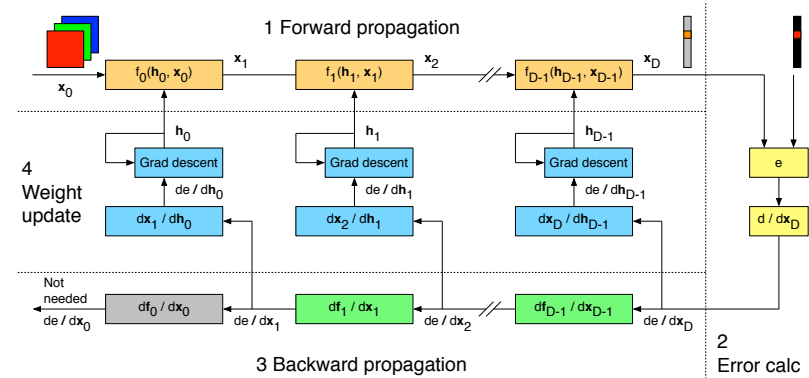
## • Cookbook

- 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)



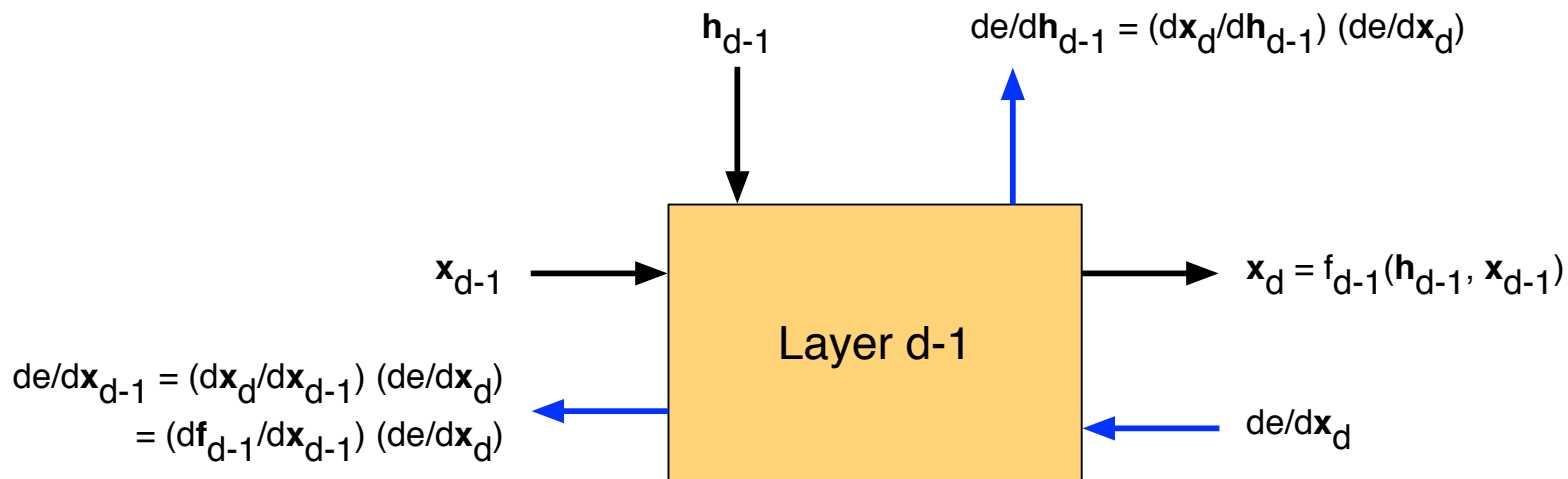
# Weight Update

- 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 (A)
  - Then use gradient descent to adapt the parameters to reduce the error (B)
- In math ( $d - 1$  is the layer,  $t$  is the time step)
  - A:  $\partial e / \partial \mathbf{h}_{d-1} = (\partial \mathbf{x}_d / \partial \mathbf{h}_{d-1}) (\partial e / \partial \mathbf{x}_d)$   
 $= (\partial \mathbf{f}_{d-1} / \partial \mathbf{h}_{d-1}) (\partial e / \partial \mathbf{x}_d)$
  - B:  $\mathbf{h}_{t+1,d-1} = \mathbf{h}_{t,d-1} - \alpha_t \partial e / \partial \mathbf{h}_{t,d-1}$



# Summary For 1 Layer

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



# Summary For 1 Layer

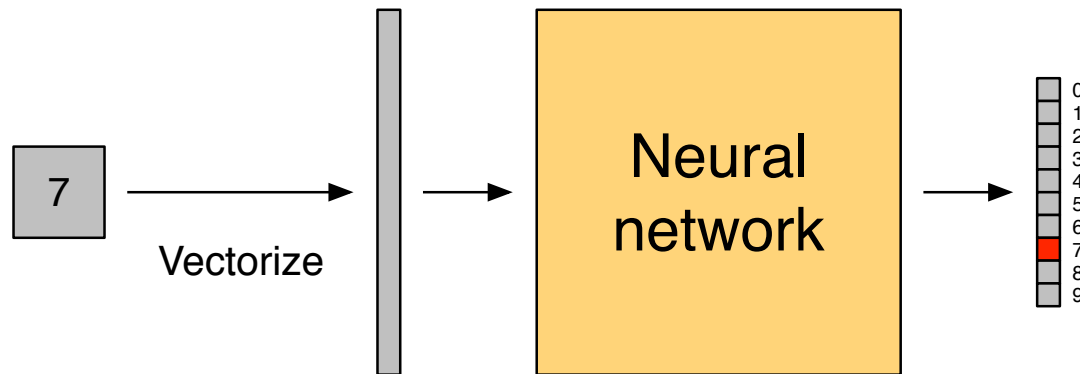
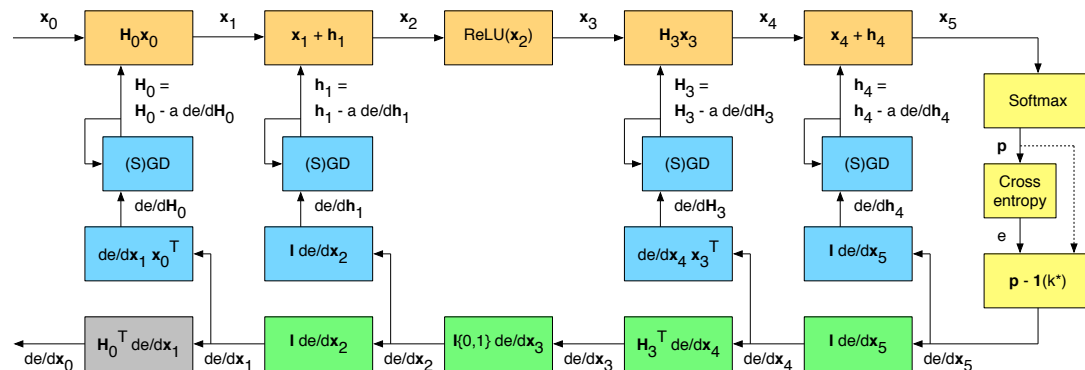
- Example: matrix vector multiplication (dimensions  $(M \times 1) = (M \times K) (K \times 1)$ )
  - $\mathbf{x}_d = f_{d-1}(\mathbf{H}_{d-1}, \mathbf{x}_{d-1}) = \mathbf{H}_{d-1} \mathbf{x}_{d-1}$
- Error gradient with respect to the feature map (dimensions  $(K \times 1) = (K \times M) (M \times 1)$ )
  - 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  $(M \times K) = (M \times 1) (1 \times K)$ )
  - Given:  $\partial e / \partial \mathbf{x}_d$
  - Compute:  $\partial e / \partial \mathbf{H}_{d-1} = (\partial \mathbf{x}_d / \partial \mathbf{H}_{d-1}) (\partial e / \partial \mathbf{x}_d) = (\partial e / \partial \mathbf{x}_d) (\partial \mathbf{f}_{d-1} / \partial \mathbf{H}_{d-1}) = (\partial e / \partial \mathbf{x}_d) (\mathbf{x}_{d-1})^T$ 
    - Note the reversing of order as a consequence of the transpose for this case (to see match terms with previous case)
    - 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

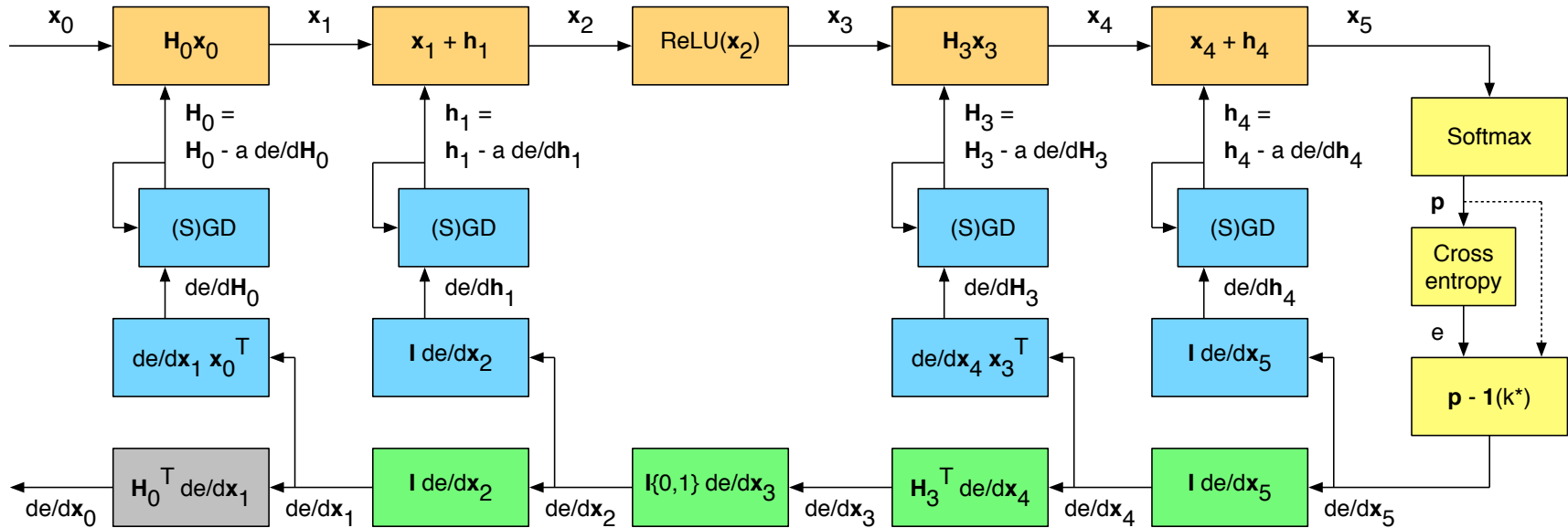
# Example

- MNIST digit recognition
  - The hello, world! of machine learning
- Simple network design to illustrate training (not optimal for accuracy)
  - $\mathbf{x}_5 = \mathbf{H}_3 \text{ReLU}_2(\mathbf{H}_0 \mathbf{x}_0 + \mathbf{h}_1) + \mathbf{h}_4$
- Goal
  - Make arg max error of testing small
  - Proxy of making training error  $e$  small
- What controls the proxy error  $e$ 
  - For this topology  $\mathbf{H}_0$ ,  $\mathbf{h}_1$ ,  $\mathbf{H}_3$  and  $\mathbf{h}_4$
- How to adapt  $\mathbf{H}_0$ ,  $\mathbf{h}_1$ ,  $\mathbf{H}_3$  and  $\mathbf{h}_4$  to make the proxy error  $e$  small
  - (Stochastic) gradient descent
- How to find the values (S)GD needs for the update
  - Back propagation





# Example



# Universal Approximation

# Definition

- A neural network with a single hidden layer, finite number of neurons and appropriate nonlinearity can approximate arbitrarily closely any continuous function on a compact subset of  $\mathbb{R}^k$
- 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 Approximation Matters

- Neural networks are a general tool to solve all sorts of problems
  - Theoretically we don't need to learn a 1000 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 neural network with a single hidden layer 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

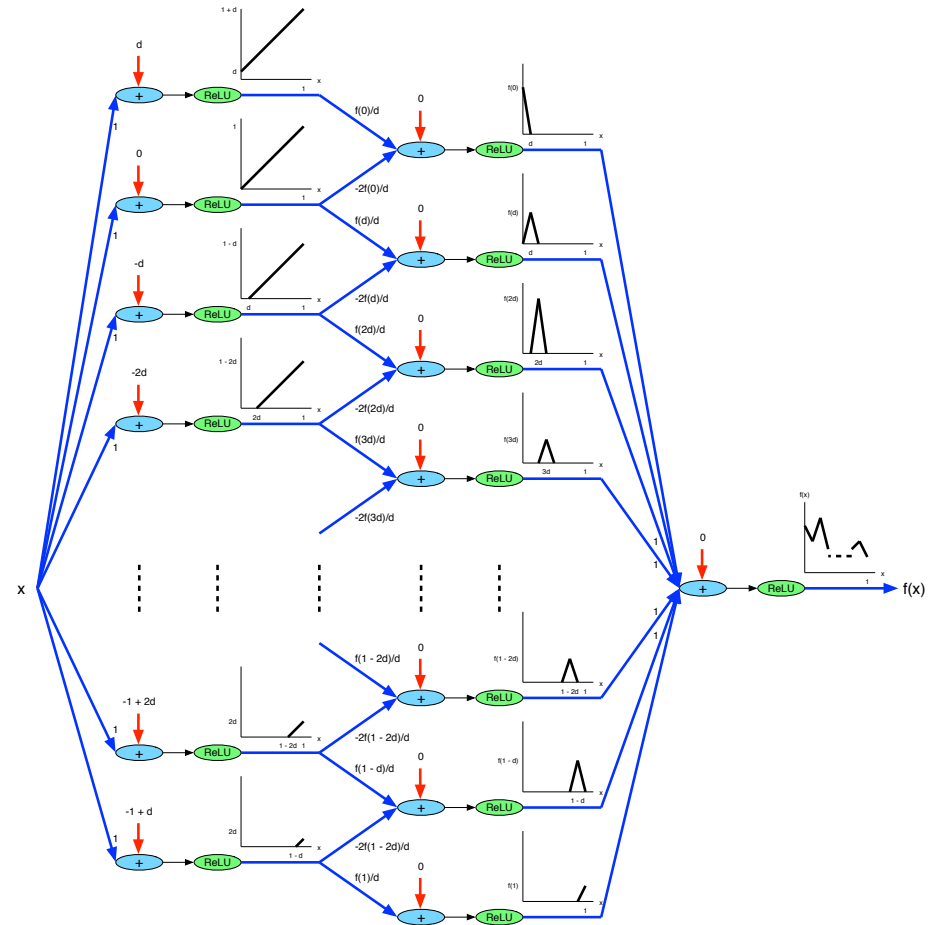
- It makes sense to discuss approximation within the context of calculus
  - Limits, bounds on derivatives, ...
  - And there's not a separate analysis lecture
- It's also nice to think about universal approximation before getting into network design

# 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

# Constructive Proof

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



# Constructive Proof

- Generalizing to additional cases
  - $x_{\min} \leq x \leq x_{\max}$  where  $x_{\min}$  and  $x_{\max}$  are finite and not limited to  $[0, 1]$ 
    - Select initial bias values as  $x_{\min} + d, x_{\min}, \dots, -x_{\max} + d$
    - Use associated locations for hidden value 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  $-f_{\min}$
    - Do not apply the last ReLU
  - Vector input  $\mathbf{x} \in \mathbb{R}^K$ 
    - Replicate the input layer for each input component  $x(k)$
    - Combine them in the hidden layer to produce  $K$  dimensional pyramids
  - Vector output  $\mathbf{y} = f(\mathbf{x}) \in \mathbb{R}^M$ 
    - Replicate the hidden layer and output layer for each output component  $y(m)$



# Constructive Proof

- Intuition
  - 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
  - More layers would allow more complex basis shapes to be created, potentially reducing the width requirements of the network
  - These benefits of depth can be shown in theory and are observed in practice

# References

# General

- 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](https://web.stanford.edu/~boyd/cvxbook/bv_cvxbook.pdf)

# xNN

- 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>
- 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>
- ResNet with one-neuron hidden layers is a universal approximator
  - <https://arxiv.org/abs/1806.10909>