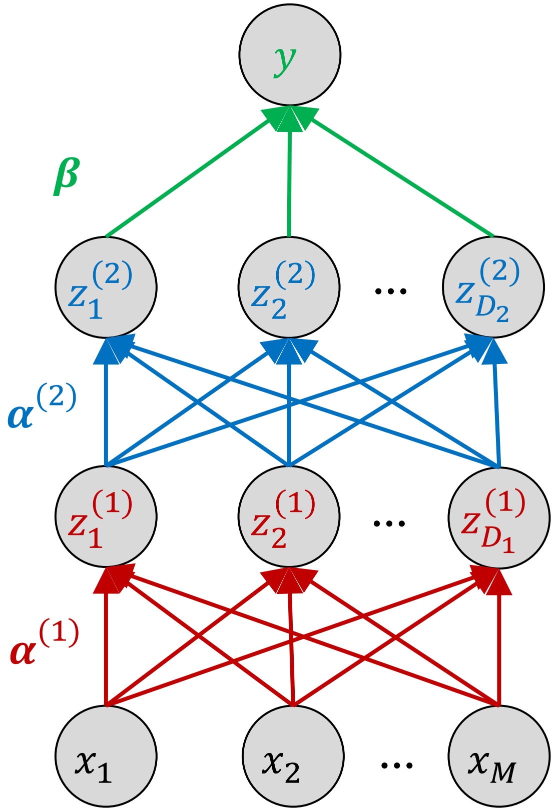
10-301/601: Introduction to Machine Learning Lecture 13 – Differentiation

Henry Chai & Matt Gormley & Hoda Heidari

2/26/24

Recall: Neural Networks (Matrix Form)



$\boldsymbol{\beta} \in \mathbb{R}^{D_2}$

$\beta_0 \in \mathbb{R}$

$\boldsymbol{\alpha}^{(2)} \in \mathbb{R}^{M \times D_2}$

$\boldsymbol{b}^{(2)} \in \mathbb{R}^{D_2}$

$\boldsymbol{\alpha}^{(1)} \in \mathbb{R}^{M \times D_1}$

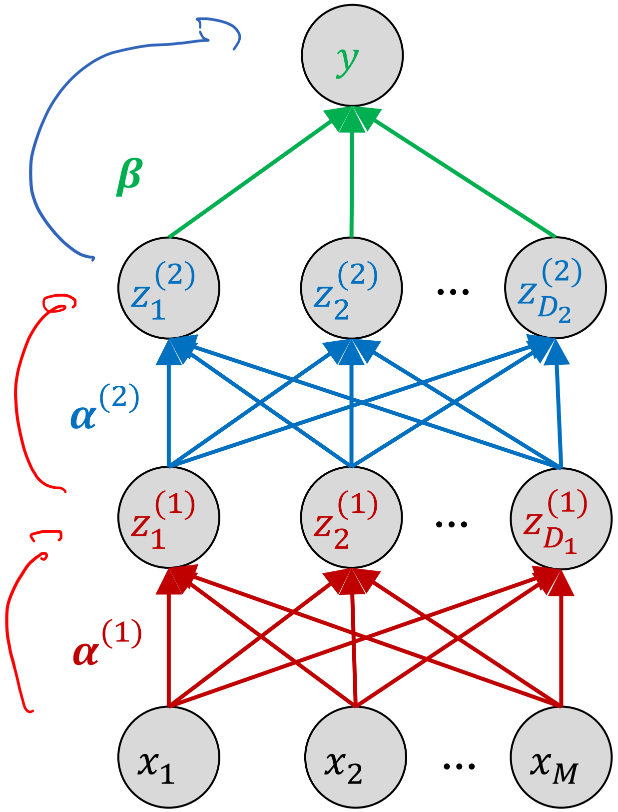
$\boldsymbol{b}^{(1)} \in \mathbb{R}^{D_1}$

$$y = \sigma((\boldsymbol{\beta})^T \boldsymbol{z}^{(2)} + \beta_0)$$

$$\boldsymbol{z}^{(2)} = \sigma((\boldsymbol{\alpha}^{(2)})^T \boldsymbol{z}^{(1)} + \boldsymbol{b}^{(2)})$$

$$\boldsymbol{z}^{(1)} = \sigma((\boldsymbol{\alpha}^{(1)})^T \boldsymbol{x} + \boldsymbol{b}^{(1)})$$

Recall: Neural Networks (Matrix Form)



$$y = \sigma\left(\boldsymbol{\beta}'^T \begin{bmatrix} 1 \\ \mathbf{z}^{(2)} \end{bmatrix}\right)$$

$$\boldsymbol{\beta}' = \begin{bmatrix} \beta_0 \\ \boldsymbol{\beta} \end{bmatrix} \in \mathbb{R}^{D_2+1}$$

$$\mathbf{z}^{(2)} = \sigma\left({\boldsymbol{\alpha}^{(2)}}'^T \begin{bmatrix} 1 \\ \mathbf{z}^{(1)} \end{bmatrix}\right)$$

$$\boldsymbol{\alpha}^{(2)\prime} = \begin{bmatrix} {\mathbf{b}^{(2)}}^T \\ \boldsymbol{\alpha}^{(2)} \end{bmatrix} \in \mathbb{R}^{(D_1+1)\times D_2}$$

$$\mathbf{z}^{(1)} = \sigma\left({\boldsymbol{\alpha}^{(1)}}'^T \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix}\right)$$

$$\boldsymbol{\alpha}^{(1)\prime} = \begin{bmatrix} {\mathbf{b}^{(1)}}^T \\ \boldsymbol{\alpha}^{(1)} \end{bmatrix} \in \mathbb{R}^{(M+1)\times D_1}$$

Forward Propagation for Making Predictions

- Inputs: weights $\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}$ and a query data point $\boldsymbol{x}'$
- Initialize $\boldsymbol{z}^{(0)} = \boldsymbol{x}'$
- For $l = 1, \dots, L$
  - $\boldsymbol{a}^{(l)} = \boldsymbol{\alpha}^{(l)^T} \boldsymbol{z}^{(l-1)}$
  - $\boldsymbol{z}^{(l)} = \sigma\left(\boldsymbol{a}^{(l)}\right)$
- $\hat{y} = \sigma\left(\boldsymbol{\beta}^T \boldsymbol{z}^{(L)}\right)$
- Output: the prediction $\hat{y}$

Stochastic Gradient Descent for Learning

- Input: $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}, \gamma$
- Initialize all weights $\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}$
- While TERMINATION CRITERION is not satisfied
  - For $i \in \text{shuffle}(\{1, \dots, N\})$
    - Compute $g_{\boldsymbol{\beta}} = \nabla_{\boldsymbol{\beta}} J^{(i)}(\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta})$
    - For $l = 1, \dots, L$
      - Compute $g_{\boldsymbol{\alpha}^{(l)}} = \nabla_{\boldsymbol{\alpha}^{(l)}} J^{(i)}(\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta})$
    - Update $\boldsymbol{\beta} = \boldsymbol{\beta} - \gamma g_{\boldsymbol{\beta}}$
    - For $l = 1, \dots, L$
      - Update $\boldsymbol{\alpha}^{(l)} = \boldsymbol{\alpha}^{(l)} - \gamma g_{\boldsymbol{\alpha}^{(l)}}$
- Output: $\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}$

Two questions:

1. What is this loss function $J^{(i)}$?

2. How on earth do we take these gradients?

- Input: $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}, \gamma$
- Initialize all weights $\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}$
- While TERMINATION CRITERION is not satisfied
  - For $i \in \text{shuffle}(\{1, \dots, N\})$
    - Compute $g_{\boldsymbol{\beta}} = \nabla_{\boldsymbol{\beta}} J^{(i)}(\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta})$
    - For $l = 1, \dots, L$
      - Compute $g_{\boldsymbol{\alpha}^{(l)}} = \nabla_{\boldsymbol{\alpha}^{(l)}} J^{(i)}(\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta})$
    - Update $\boldsymbol{\beta} = \boldsymbol{\beta} - \gamma g_{\boldsymbol{\beta}}$
    - For $l = 1, \dots, L$
      - Update $\boldsymbol{\alpha}^{(l)} = \boldsymbol{\alpha}^{(l)} - \gamma g_{\boldsymbol{\alpha}^{(l)}}$
- Output: $\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}$

Two questions:

1. What is this loss function $J^{(i)}$?

2. How on earth do we take these gradients?

- Input: $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}, \gamma$
- Initialize all weights $\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}$
- While TERMINATION CRITERION is not satisfied
  - For $i \in \text{shuffle}(\{1, \dots, N\})$
    - Compute $g_{\boldsymbol{\beta}} = \nabla_{\boldsymbol{\beta}} J^{(i)}(\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta})$
    - For $l = 1, \dots, L$
      - Compute $g_{\boldsymbol{\alpha}^{(l)}} = \nabla_{\boldsymbol{\alpha}^{(l)}} J^{(i)}(\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta})$
    - Update $\boldsymbol{\beta} = \boldsymbol{\beta} - \gamma g_{\boldsymbol{\beta}}$
    - For $l = 1, \dots, L$
      - Update $\boldsymbol{\alpha}^{(l)} = \boldsymbol{\alpha}^{(l)} - \gamma g_{\boldsymbol{\alpha}^{(l)}}$
- Output: $\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}$

Loss Functions for Neural Networks

- Let $\boldsymbol{\Theta} = \{\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}\}$ be the parameters of our neural network
- Regression - squared error (same as linear regression!)

$$J^{(i)}(\boldsymbol{\Theta}) = \left(\hat{y}_{\boldsymbol{\Theta}}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right)^2$$

- Binary classification - cross-entropy loss (same as logistic regression!)
  - Assume $Y \in \{0,1\}$ and $P(Y = 1|\boldsymbol{x}, \boldsymbol{\Theta}) = \hat{y}_{\boldsymbol{\Theta}}(\boldsymbol{x})$

$$J^{(i)}(\boldsymbol{\Theta}) = -\log P\left(y^{(i)}|\boldsymbol{x}^{(i)}, \boldsymbol{\Theta}\right)$$

Loss Functions for Neural Networks

- Let $\boldsymbol{\Theta} = \left\{\boldsymbol{\alpha}^{(1)}, \ldots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}\right\}$ be the parameters of our neural network
- Multi-class classification - cross-entropy loss again!
  - Express the label as a one-hot or one-of-C vector e.g.,

$$\boldsymbol{y} = [0 \quad 0 \quad 1 \quad 0 \quad \cdots \quad 0]$$

  - Assume the neural network output is also a vector of length C, $\hat{\boldsymbol{y}}_{\boldsymbol{\Theta}}$

$$P(\boldsymbol{y}[c] = 1 | \boldsymbol{x}, \boldsymbol{\Theta}) = \hat{\boldsymbol{y}}_{\boldsymbol{\Theta}}\left(\boldsymbol{x}^{(i)}\right)[c]$$

Okay but how do we get our network to output this vector?

- Let $\boldsymbol{\Theta} = \{\boldsymbol{\alpha}^{(1)}, \ldots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}\}$ be the parameters of our neural network
- Multi-class classification - cross-entropy loss
  - Express the label as a one-hot or one-of-C vector e.g.,

$$\boldsymbol{y} = [0 \quad 0 \quad 1 \quad 0 \quad \cdots \quad 0]$$

  - Assume the neural network output is also a vector of length C, $\hat{\boldsymbol{y}}_{\boldsymbol{\Theta}}$

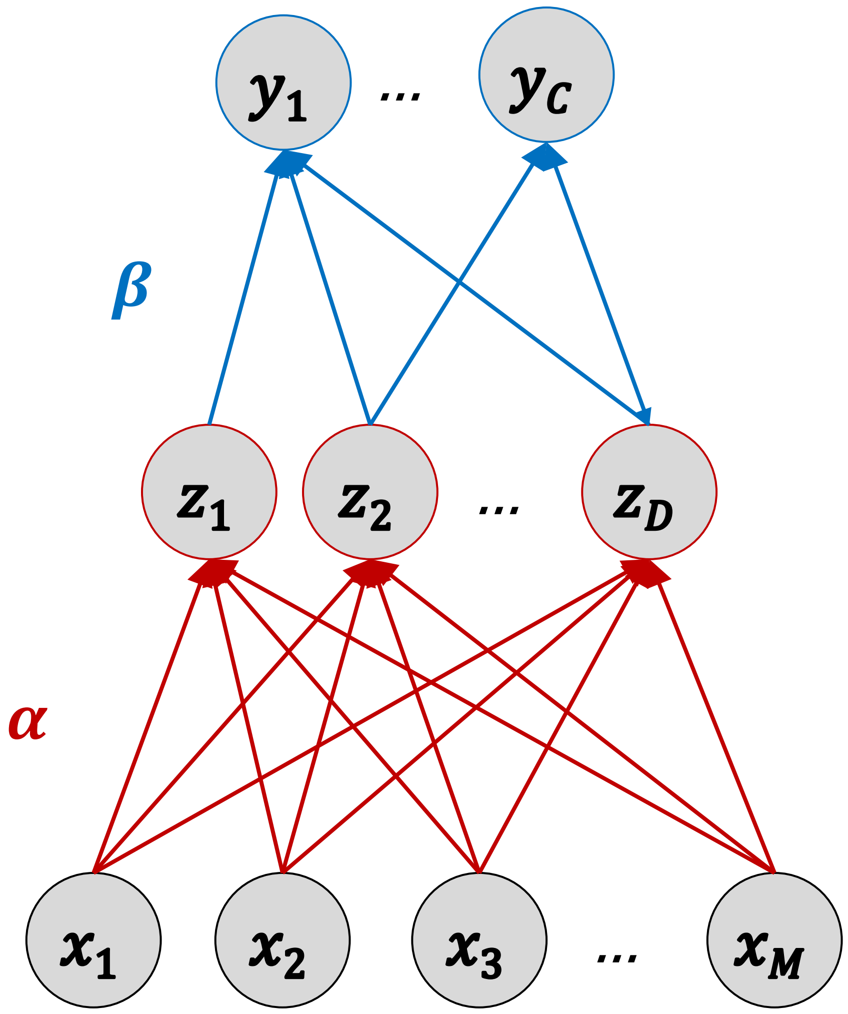
$$P(\boldsymbol{y}[c] = 1 | \boldsymbol{x}, \boldsymbol{\Theta}) = \hat{\boldsymbol{y}}_{\boldsymbol{\Theta}}\left(\boldsymbol{x}^{(i)}\right)[c]$$

  - Then the cross-entropy loss is

$$J^{(i)}(\boldsymbol{\Theta}) = -\log P\left(y^{(i)} | \boldsymbol{x}^{(i)}, \boldsymbol{\Theta}\right)$$

$$= -\sum_{c=1}^{C} \boldsymbol{y}^{(i)}[c] \log\left(\hat{\boldsymbol{y}}_{\boldsymbol{\Theta}}\left(\boldsymbol{x}^{(i)}\right)[c]\right)$$

Softmax



$$y_c = \frac{\exp b_c}{\sum_{k=1}^{C} \exp b_k}$$

$$b_c = \sum_{j=0}^{D} \beta_{c,j} z_j$$

$$z_d = \sigma(a_d)$$

$$a_d = \sum_{i=0}^{M} \alpha_{d,i} x_i$$

Two questions:

1. What is this loss function $J^{(i)}$?

2. How on earth do we take these gradients?

- Input: $\mathcal{D} = \{(\boldsymbol{x}^{(i)}, y^{(i)})\}_{i=1}^{N}, \gamma$
- Initialize all weights $\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}$
- While TERMINATION CRITERION is not satisfied
  - For $i \in \text{shuffle}(\{1, \dots, N\})$
    - Compute $g_{\boldsymbol{\beta}} = \nabla_{\boldsymbol{\beta}} J^{(i)}(\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta})$
    - For $l = 1, \dots, L$
      - Compute $g_{\boldsymbol{\alpha}^{(l)}} = \nabla_{\boldsymbol{\alpha}^{(l)}} J^{(i)}(\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta})$
    - Update $\boldsymbol{\beta} = \boldsymbol{\beta} - \gamma g_{\boldsymbol{\beta}}$
    - For $l = 1, \dots, L$
      - Update $\boldsymbol{\alpha}^{(l)} = \boldsymbol{\alpha}^{(l)} - \gamma g_{\boldsymbol{\alpha}^{(l)}}$
- Output: $\boldsymbol{\alpha}^{(1)}, \dots, \boldsymbol{\alpha}^{(L)}, \boldsymbol{\beta}$

Matrix Calculus

Numerator

Types of Derivatives	**scalar**	**vector**	**matrix**
scalar	$\frac{\partial y}{\partial x}$	$\frac{\partial \mathbf{y}}{\partial x}$	$\frac{\partial \mathbf{Y}}{\partial x}$
vector	$\frac{\partial y}{\partial \mathbf{x}}$	$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$	$\frac{\partial \mathbf{Y}}{\partial \mathbf{x}}$
matrix	$\frac{\partial y}{\partial \mathbf{X}}$	$\frac{\partial \mathbf{y}}{\partial \mathbf{X}}$	$\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}$

Denominator

Table courtesy of Matt Gormley

Matrix Calculus: Denominator Layout

- Derivatives of a scalar always have the *same shape* as the entity that the derivative is being taken with respect to.

Types of Derivatives	**scalar**
scalar	$\frac{\partial y}{\partial x} = \left[\frac{\partial y}{\partial x}\right]$
vector	$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_P} \end{bmatrix}$
matrix	$\frac{\partial y}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial y}{\partial X_{11}} & \frac{\partial y}{\partial X_{12}} & \cdots & \frac{\partial y}{\partial X_{1Q}} \\ \frac{\partial y}{\partial X_{21}} & \frac{\partial y}{\partial X_{22}} & \cdots & \frac{\partial y}{\partial X_{2Q}} \\ \vdots & & & \vdots \\ \frac{\partial y}{\partial X_{P1}} & \frac{\partial y}{\partial X_{P2}} & \cdots & \frac{\partial y}{\partial X_{PQ}} \end{bmatrix}$

Table courtesy of Matt Gormley

Matrix Calculus: Denominator Layout

Types of Derivatives	**scalar**	**vector**
scalar	$\frac{\partial y}{\partial x} = \left[\frac{\partial y}{\partial x}\right]$	$\frac{\partial \mathbf{y}}{\partial x} = \left[\frac{\partial y_1}{\partial x} \quad \frac{\partial y_2}{\partial x} \quad \cdots \quad \frac{\partial y_N}{\partial x}\right]$
vector	$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_P} \end{bmatrix}$	$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_N}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_N}{\partial x_2} \\ \vdots & & & \\ \frac{\partial y_1}{\partial x_P} & \frac{\partial y_2}{\partial x_P} & \cdots & \frac{\partial y_N}{\partial x_P} \end{bmatrix}$

Table courtesy of Matt Gormley

Three Approaches to Differentiation

- Given $f\colon \mathbb{R}^D \to \mathbb{R}$, compute $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = {}^{\partial f(\boldsymbol{x})}/_{\partial \boldsymbol{x}}$

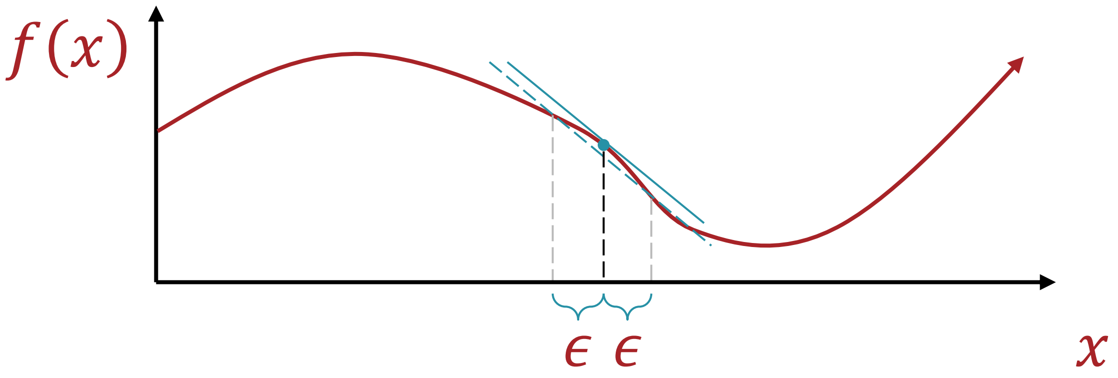
1. Finite difference method
2. Symbolic differentiation
3. Automatic differentiation (reverse mode)

Approach 1: Finite Difference Method

- Given $f\colon \mathbb{R}^D \to \mathbb{R}$, compute $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = {}^{\partial f(\boldsymbol{x})}/_{\partial \boldsymbol{x}}$

$$\frac{\partial f(\boldsymbol{x})}{\partial x_i} \approx \frac{f(\boldsymbol{x} + \epsilon \boldsymbol{d}_i) - f(\boldsymbol{x} - \epsilon \boldsymbol{d}_i)}{2\epsilon}$$

where $\boldsymbol{d}_i$ is a one-hot vector with a 1 in the i^{th} position



- We want ϵ to be small to get a good approximation but we run into floating point issues when ϵ is too small
- Getting the full gradient requires computing the above approximation for each dimension of the input

Approach 1: Finite Difference Method Example

- Given

$$y = f(x,z) = e^{xz} + \frac{xz}{\ln(x)} + \frac{\sin(\ln(x))}{xz}$$

what are ${}^{\partial y}/_{\partial x}$ and ${}^{\partial y}/_{\partial z}$ at $x = 2, z = 3$?

Example courtesy of Matt Gormley

Three Approaches to Differentiation

- Given $f\colon \mathbb{R}^D \to \mathbb{R}$, compute $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = {}^{\partial f(\boldsymbol{x})}/_{\partial \boldsymbol{x}}$

1. Finite difference method
   - Requires the ability to call $f(\boldsymbol{x})$
   - Great for checking accuracy of implementations of more complex differentiation methods
   - Computationally expensive for high-dimensional inputs
2. Symbolic differentiation
3. Automatic differentiation (reverse mode)

Approach 2: Symbolic Differentiation

- Given

$$y = f(x, z) = e^{xz} + \frac{xz}{\ln(x)} + \frac{\sin(\ln(x))}{xz}$$

what are ${}^{\partial y}/_{\partial x}$ and ${}^{\partial y}/_{\partial z}$ at $x = 2, z = 3$?

Example courtesy of Matt Gormley

The Chain Rule of Calculus

- If $y = f(z)$ and $z = g(x)$ then

the corresponding computation graph is

- If $y = f(z_1, z_2)$ and $z_1 = g_1(x), z_2 = g_2(x)$ then

- If $y = f(\boldsymbol{z})$ and $\boldsymbol{z} = g(x)$ then

Poll Question 1

- If $y = f(\boldsymbol{z})$, $\boldsymbol{z} = g(\boldsymbol{w})$ and $\boldsymbol{w} = h(x)$, does the equation

$$\frac{\partial y}{\partial x} = \sum_{d=1}^{D} \frac{\partial y}{\partial z_d} \frac{\partial z_d}{\partial x}$$

still hold?

A. Yes
B. No
C. Only on Fridays (TOXIC)

Approach 2: Symbolic Differentiation

- Given

$$y = f(x, z) = e^{xz} + \frac{xz}{\ln(x)} + \frac{\sin(\ln(x))}{xz}$$

what are ${}^{\partial y}/_{\partial x}$ and ${}^{\partial y}/_{\partial z}$ at $x = 2, z = 3$?

Three Approaches to Differentiation

- Given $f\colon \mathbb{R}^D \to \mathbb{R}$, compute $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = {}^{\partial f(\boldsymbol{x})}/_{\partial \boldsymbol{x}}$

1. Finite difference method
   - Requires the ability to call $f(\boldsymbol{x})$
   - Great for checking accuracy of implementations of more complex differentiation methods
   - Computationally expensive for high-dimensional inputs
2. Symbolic differentiation
   - Requires systematic knowledge of derivatives
   - Can be computationally expensive if poorly implemented
3. Automatic differentiation (reverse mode)

Approach 3: Automatic Differentiation (reverse mode)

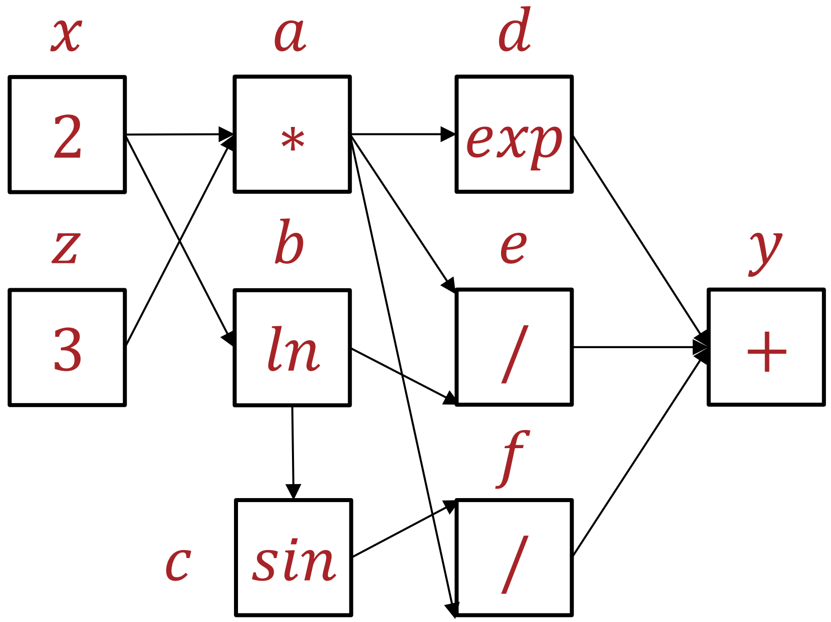
- Given

$$y = f(x,z) = e^{xz} + \frac{xz}{\ln(x)} + \frac{\sin(\ln(x))}{xz}$$

what are $^{\partial y}/_{\partial x}$ and $^{\partial y}/_{\partial z}$ at $x = 2, z = 3$?

- First define some intermediate quantities, draw the computation graph and run the "forward" computation

$$a = xz$$
$$b = \ln(x)$$
$$c = \sin(b)$$
$$d = e^a$$
$$e = {}^{a}/_{b}$$
$$f = {}^{c}/_{a}$$
$$y = d + e + f$$



Example courtesy of Matt Gormley

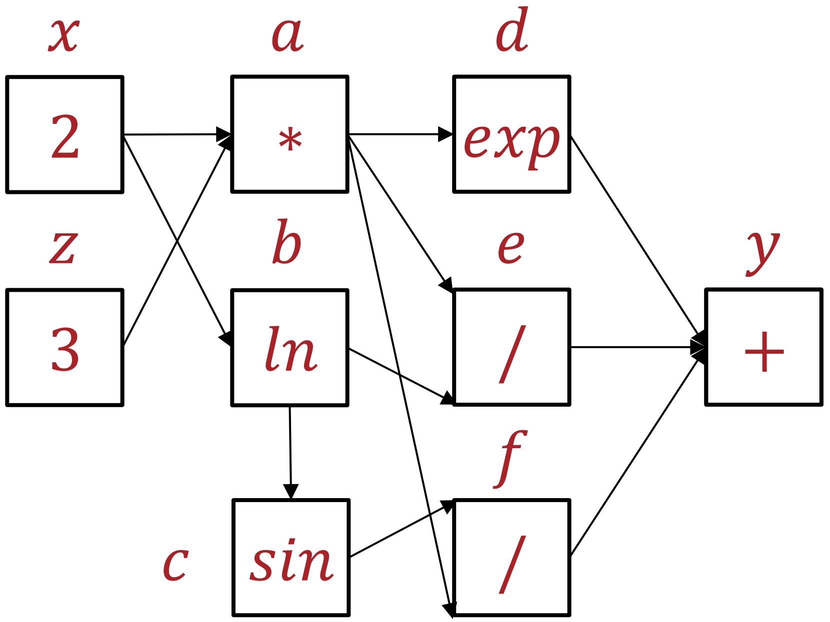
Approach 3: Automatic Differentiation (reverse mode)

- Given

$$y = f(x, z) = e^{xz} + \frac{xz}{\ln(x)} + \frac{\sin(\ln(x))}{xz}$$

what are ${\partial y}/{\partial x}$ and ${\partial y}/{\partial z}$ at $x = 2, z = 3$?

- Then compute partial derivatives, starting from y and working back



Three Approaches to Differentiation

- Given $f: \mathbb{R}^D \rightarrow \mathbb{R}$, compute $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = {}^{\partial f(\boldsymbol{x})}/_{\partial \boldsymbol{x}}$

1. Finite difference method
   - Requires the ability to call $f(\boldsymbol{x})$
   - Great for checking accuracy of implementations of more complex differentiation methods
   - Computationally expensive for high-dimensional inputs
2. Symbolic differentiation
   - Requires systematic knowledge of derivatives
   - Can be computationally expensive if poorly implemented
3. Automatic differentiation (reverse mode)
   - Requires systematic knowledge of derivatives *and* an algorithm for computing $f(\boldsymbol{x})$
   - Computational cost of computing ${}^{\partial f(\boldsymbol{x})}/_{\partial \boldsymbol{x}}$ is proportional to the cost of computing $f(\boldsymbol{x})$

Automatic Differentiation

- Given $f\colon \mathbb{R}^D \to \mathbb{R}^C$, compute $\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) = {}^{\partial f(\boldsymbol{x})}\!/_{\partial \boldsymbol{x}}$

3. Automatic differentiation (reverse mode)
   - Requires systematic knowledge of derivatives *and* an algorithm for computing $f(\boldsymbol{x})$
   - Computational cost of computing $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})_c = {}^{\partial f(\boldsymbol{x})_c}\!/_{\partial \boldsymbol{x}}$ is proportional to the cost of computing $f(\boldsymbol{x})$
   - Great for high-dimensional inputs and low-dimensional outputs ($D \gg C$)
4. Automatic differentiation (forward mode)
   - Requires systematic knowledge of derivatives *and* an algorithm for computing $f(\boldsymbol{x})$
   - Computational cost of computing ${}^{\partial f(\boldsymbol{x})}\!/_{\partial \boldsymbol{x}_d}$ is proportional to the cost of computing $f(\boldsymbol{x})$
   - Great for low-dimensional inputs and high-dimensional outputs ($D \ll C$)

Computation Graph: 10-301/601 Conventions

- The diagram represents *an algorithm*
- Nodes are rectangles with one node per intermediate variable in the algorithm
- Each node is labeled with the function that it computes (inside the box) and the variable name (outside the box)
- Edges are directed and do not have labels
- For neural networks:
  - Each weight, feature value, label and *bias term* appears as a node
  - We *can* include the loss function

Neural Network Diagram Conventions

- The diagram represents a *neural network*
- Nodes are circles with one node per hidden unit
- Each node is labeled with the variable corresponding to the hidden unit
- Edges are directed and each edge is labeled with its weight
- Following standard convention, the bias term is typically *not* shown as a node, but rather is assumed to be part of the activation function i.e., its weight does not appear in the picture anywhere.
- The diagram typically does *not* include any nodes related to the loss computation

Backprop Learning Objectives

You should be able to…

- Differentiate between a neural network diagram and a computation graph
- Construct a computation graph for a function as specified by an algorithm
- Carry out the backpropagation on an arbitrary computation graph
- Construct a computation graph for a neural network, identifying all the given and intermediate quantities that are relevant
- Instantiate the backpropagation algorithm for a neural network
- Instantiate an optimization method (e.g. SGD) and a regularizer (e.g. L2) when the parameters of a model are comprised of several matrices corresponding to different layers of a neural network
- Use the finite difference method to evaluate the gradient of a function
- Identify when the gradient of a function can be computed at all and when it can be computed efficiently
- Employ basic matrix calculus to compute vector/matrix/tensor derivatives.