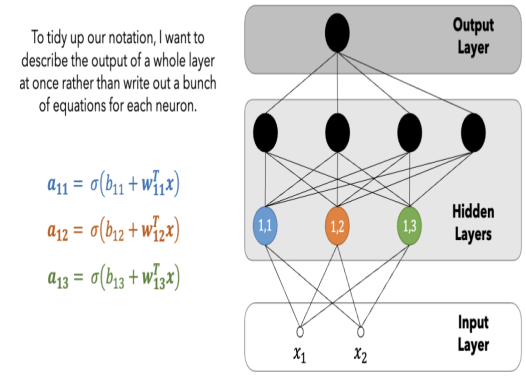
Week 7.2 Neural Network

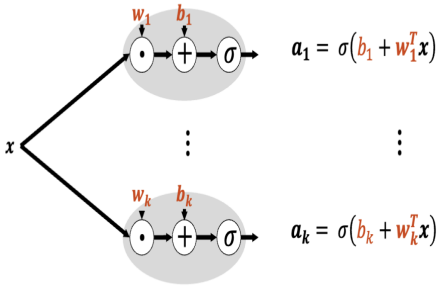
Objectives:

- What is a fully- connected layer?
- What is a loss function?
- A quick review of vector calculus
- What is forward/reverse mode differentiation?
- How does backpropagation work?
- What is a computation graph?

Working on Notation for Feed-Forward Neural Networks



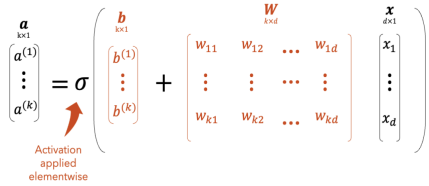
Fully-Connected Layer of Neurons



CS 434 © Stefan Lee 16 CS 434 17

Fully-Connected Layer of Neurons Vectorized

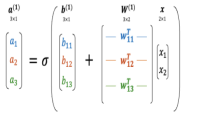
$$a = \sigma(b + Wx)$$



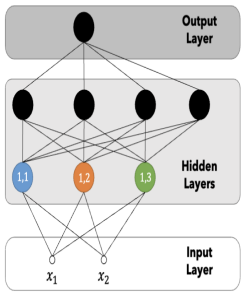
CS 434 © Stefan Lee 18

Working on Notation for Feed-Forward Neural Networks

$a_{11} = \sigma(b_{11} + w_{11}^T x)$
$a_{12} = \sigma(b_{12} + w_{12}^T x)$
$a_{13} = \sigma(b_{13} + w_{13}^T x)$

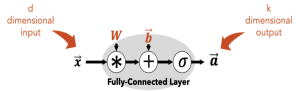


$a^{(1)} = \sigma(b^{(1)} + W^{(1)} x)$



CS 434 © Stefan Lee 19

Fully-Connected Layer



Hyperparameters : width of layer (k) and choice of activation functions (σ)

Learnable Parameters: $W \in \mathbb{R}^{k \times d}$ and $\vec{b} \in \mathbb{R}^k$

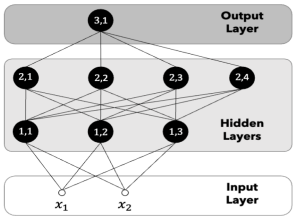
Working on Notation for Feed-Forward Neural Networks

Suppose a network has L layers:

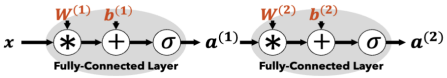
Let $W^{(1)}, W^{(2)}, \ldots, W^{(L)}$ be **weight matrices** for layers 1,2,…,L.

Let $b^{(1)}, b^{(2)}, \ldots, b^{(L)}$ be **bias vectors** for layers 1,2,…,L.

Let $a^{(1)}, a^{(2)}, \ldots, a^{(L)}$ be **vectors of activations** for layers 1,2,…,L.



Multi-layer Networks



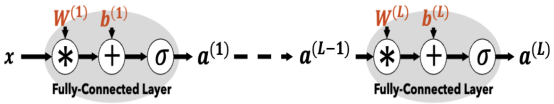
$$a^{(1)} = \sigma(b^{(1)} + W^{(1)}x)$$

$$a^{(2)} = \sigma(b^{(2)} + W^{(2)}a^{(1)})$$

Multi-layer Networks

$$a^{(2)} = \sigma(b^{(2)} + W^{(2)}\sigma(b^{(1)} + W^{(1)}x))$$

Multi-layer Networks



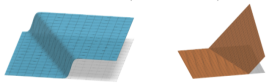
Hyperparameters : depth in layers (L), width of each layer ($k_1, \ldots, k_L$), and choice of activation functions for each layer ($\sigma_1, \ldots, \sigma_L$)

Learnable Parameters: $W_1 \in \mathbb{R}^{k_1 \times d}, W_{i>1} \in \mathbb{R}^{k_i \times k_{i-1}}$ and $b_i \in \mathbb{R}^{k_i}$

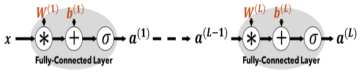
Let's Review Where We are

I introduced a neuron last class and said they weren't very expressive on their own.

- Sigmoid neuron and ReLU neuron outputs below (2d inputs):



Had the great idea to combine them together to make them more powerful - called this a neural network. One common configuration is a feed-forward neural network.



Question remaining: How do I actually get this complicated looking thing to do something useful?

What are Neural Networks good for?

Non-linear Function Learning: As we saw in the demo yesterday, neural networks can learn non-linear decision boundaries. But unlike the kernel methods / basis functions we used before, we don't need to specify the exact from of this non-linear function.

Theoretically Universal Approximators: For any continuous function F over a bounded subspace of D-dimensional space, there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximates F arbitrarily well. That is to say, for all x in the domain of F, $|F(x) - \hat{F}(x)| < \epsilon$

Incredibly Flexible: Can be used for classification or regression. Or just any problem with a differentiable loss function.

What is a loss function?

Loss Functions

Loss Function *L*- A function measuring "how bad" a network's output is, usually relative to some gold-standard for what the output should be.

Loss Functions - Regression

Squared Error

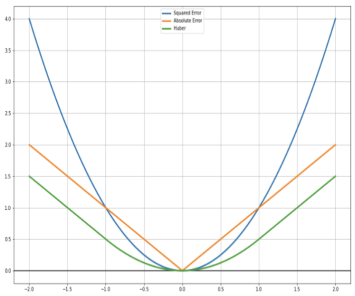
$$L(a,b) = (a-b)^2$$

Absolute Error

$$L(a,b) = |a-b|$$

Huber

$$L(a,b) = \begin{cases} \frac{1}{2}(a-b)^2 & \text{if } |a-b| < \delta \\ \delta|a-b| - \frac{1}{2}\delta^2 & \text{else} \end{cases}$$



Loss Functions - Classification

Cross Entropy ($\vec{a} \in \Delta^C$ $\vec{b} \in \Delta^C$)

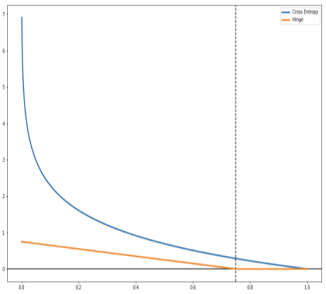
$$L(\vec{a},\vec{b}) = -E_b[\log(a)] = -\sum_{i=0}^{C} b_i \log(a_i)$$

➡ *if* $\vec{b} = \vec{1}_c \rightarrow -log(a_c)$

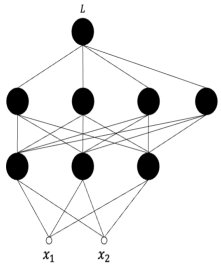
Hinge ($\vec{a} \in \mathbb{R}^C$ $\vec{b} = \vec{1}_c$)

$$L(\vec{a},\vec{b}) = \sum_{i=0}^{C} \max(0, \delta - b_i a_i)$$

➡ *if* $\vec{b} = \vec{1}_c \rightarrow \max(0, \delta - b_c a_c)$



Intuition For Neural Network Training



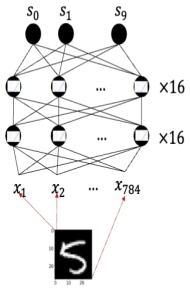
Neural Network Training In Words

Forward Pass
1) For each training example:
- Compute and store all activations
- Compute loss

Backward Pass
2) Compute gradient of the loss with respect to all network parameters
- Will do this efficiently with an algorithm called **backpropagation**

Update
Take a step of gradient descent to minimize the loss

An Example from the Homework



Input: an image represented as a 784 dimensional vector of real values

Model: 2 hidden layers with ReLU activations and widths of 16 neurons

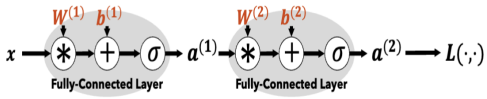
Output: scores for each digit class. Can compute probability for each as:

$$p_{0|x} = \frac{e^{s_0}}{\sum_{c=0}^{9} e^{s_c}}$$

Loss: cross entropy, for this example:

$$\mathcal{L} = -log\ p_{5|x}$$

Time to Pay the Math Tax



$$\frac{\delta L}{\delta W^{(1)}} = ?$$

Stochastic Gradient Descent

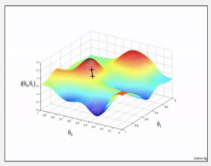
Recall from Machine Learning / Calculus / Optimization

Our loss function $L: \theta \rightarrow \mathbb{R}$ defines a "loss surface".
Optimization is about getting as low as possible on this surface.

Recall: Gradient (vector of partial derivatives) point in the direction of steepest ascent.

Stochastic Gradient Descent Algorithm

$\theta \leftarrow random(\ )$
Sample batch B
while $|\nabla_\theta L| > \epsilon$ *or iters remain*
$\theta = \theta - \alpha \nabla_\theta L(B; \theta)$



Recall from Calculus: Scalar Partial Derivatives

Derivative of f with respect to x:

$$\frac{d}{dx} f(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Partial Derivatives of f with respect to x and y:

$$\frac{\delta}{\delta x} f(x,y) = \lim_{h \to 0} \frac{f(x+h,y) - f(x,y)}{h}$$

$$\frac{\delta}{\delta y} f(x,y) = \lim_{h \to 0} \frac{f(x,y+h) - f(x,y)}{h}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$\frac{\delta \textbf{vector}}{\delta \textbf{scalar}}$

$$\frac{\delta \vec{y}}{\delta x} = \begin{bmatrix} \frac{\delta y_1}{\delta x} \\ \frac{\delta y_2}{\delta x} \\ \vdots \\ \frac{\delta y_d}{\delta x} \end{bmatrix}_{d \times 1}$$

$\frac{\delta \textbf{scalar}}{\delta \textbf{vector}}$

$$\frac{\delta y}{\delta \vec{x}} = \begin{bmatrix} \frac{\delta y}{\delta x_1} & \frac{\delta y}{\delta x_2} & \cdots & \frac{\delta y}{\delta x_c} \end{bmatrix}_{1 \times c}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$\frac{\delta \textbf{vector}}{\delta \textbf{scalar}}$

$$\frac{\delta \vec{y}}{\delta x} = \begin{bmatrix} \frac{\delta y_1}{\delta x} \\ \frac{\delta y_2}{\delta x} \\ \vdots \\ \frac{\delta y_d}{\delta x} \end{bmatrix}_{d \times 1}$$

$$\vec{y}(x) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -5x \\ x^2 \end{bmatrix}_{2 \times 1}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$\frac{\delta \textbf{vector}}{\delta \textbf{scalar}}$

$$\frac{\delta \vec{y}}{\delta x} = \begin{bmatrix} \frac{\delta y_1}{\delta x} \\ \frac{\delta y_2}{\delta x} \\ \vdots \\ \frac{\delta y_d}{\delta x} \end{bmatrix}_{d \times 1}$$

$$\vec{y}(x) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -5x \\ x^2 \end{bmatrix}_{2 \times 1}$$

$$\frac{\delta \vec{y}}{\delta x} = \begin{bmatrix} -5 \\ 2x \end{bmatrix}_{2 \times 1}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$\frac{\delta \textbf{scalar}}{\delta \textbf{vector}}$

$$\frac{\delta y}{\delta \vec{x}} = \begin{bmatrix} \frac{\delta y}{\delta x_1} & \frac{\delta y}{\delta x_2} & \cdots & \frac{\delta y}{\delta x_c} \end{bmatrix}_{1 \times c}$$

$$y(\vec{x}) = \vec{w}^T \vec{x} = \sum_i w_i x_i \qquad \vec{w} \in \mathbb{R}^c$$

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$\frac{\delta \textbf{scalar}}{\delta \textbf{vector}}$

$$\frac{\delta y}{\delta \vec{x}} = \begin{bmatrix} \frac{\delta y}{\delta x_1} & \frac{\delta y}{\delta x_2} & \cdots & \frac{\delta y}{\delta x_c} \end{bmatrix}_{1 \times c}$$

$$y(\vec{x}) = \vec{w}^T \vec{x} = \sum_i w_i x_i \qquad \vec{w} \in \mathbb{R}^c$$

$$\frac{\delta y}{\delta \vec{x}} = \begin{bmatrix} w_1 & w_2 & \cdots & w_c \end{bmatrix}_{1 \times c} = \vec{w}^T$$

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$\frac{\delta \textbf{vector}}{\delta \textbf{vector}}$

$$\frac{\delta \vec{y}}{\delta \vec{x}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \cdots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \cdots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}_{d \times c}$$

Called the Jacobian

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\delta \vec{y}}{\delta \vec{x}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \ldots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \ldots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}_{d \times c}$$

Called the Jacobian

$$\vec{y}(\vec{x}) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -5x_1 + x_2 \\ x_1^2 \end{bmatrix}$$

Notation for Vector Calculus

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\delta \vec{y}}{\delta \vec{x}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \ldots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \ldots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}_{d \times c}$$

Called the Jacobian

$$\vec{y}(\vec{x}) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} -5x_1 + x_2 \\ x_1^2 \end{bmatrix}$$

$$\frac{\delta \vec{y}}{\delta \vec{x}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} \end{bmatrix} = \begin{bmatrix} -5 & 1 \\ 2x_1 & 0 \end{bmatrix}$$

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

$$\frac{\delta \text{vector}}{\delta \text{matrix}}$$

$$\frac{\delta \vec{y}}{\delta \mathbf{X}} = \begin{bmatrix} \frac{\delta y_1}{\delta x_{11}} & \frac{\delta y_1}{\delta x_{12}} & \cdots & \frac{\delta y_1}{\delta x_{mn}} \\ \frac{\delta y_2}{\delta x_{11}} & \frac{\delta y_2}{\delta x_{12}} & \cdots & \frac{\delta y_2}{\delta x_{mn}} \\ \vdots & \vdots & \ldots & \vdots \\ \frac{\delta y_d}{\delta x_{11}} & \frac{\delta y_d}{\delta x_{12}} & \ldots & \frac{\delta y_d}{\delta x_{mn}} \end{bmatrix}_{d \times mn}$$

Vectorize the matrix and treat like vector-vector derivative

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

Chain Rule

$$L(\vec{x}) = L\left(\vec{y}(\vec{x})\right)$$

$$\underset{1 \times c}{\frac{\delta L}{\delta \vec{x}}} = \underset{1 \times d}{\frac{\delta L}{\delta \vec{y}}} * \underset{d \times c}{\frac{\delta \vec{y}}{\delta \vec{x}}}$$

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

Chain Rule

$$\underset{1 \times c}{\begin{bmatrix} \frac{\delta L}{\delta x_1} & \frac{\delta L}{\delta x_2} & \cdots & \frac{\delta L}{\delta x_c} \end{bmatrix}} = \underset{1 \times d}{\begin{bmatrix} \frac{\delta L}{\delta y_1} & \frac{\delta L}{\delta y_2} & \cdots & \frac{\delta L}{\delta y_d} \end{bmatrix}} \underset{d \times c}{\begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \ldots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \ldots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}}$$

Scalars: $y, x \in \mathbb{R}$ Vectors: $\vec{y} \in \mathbb{R}^d, \vec{x} \in \mathbb{R}^c$ Matrices: $\mathbf{X} \in \mathbb{R}^{m \times n}, \mathbf{Y} \in \mathbb{R}^{o \times p}$

Our convention: Numerator dim → rows, Denominator dim → cols

Chain Rule

$$\begin{bmatrix} \frac{\delta L}{\delta x_1} & \frac{\delta L}{\delta x_2} & \cdots & \frac{\delta L}{\delta x_c} \end{bmatrix} = \begin{bmatrix} \frac{\delta L}{\delta y_1} & \frac{\delta L}{\delta y_2} & \cdots & \frac{\delta L}{\delta y_d} \end{bmatrix} \begin{bmatrix} \frac{\delta y_1}{\delta x_1} & \frac{\delta y_1}{\delta x_2} & \cdots & \frac{\delta y_1}{\delta x_c} \\ \frac{\delta y_2}{\delta x_1} & \frac{\delta y_2}{\delta x_2} & \cdots & \frac{\delta y_2}{\delta x_c} \\ \vdots & \vdots & \ldots & \vdots \\ \frac{\delta y_d}{\delta x_1} & \frac{\delta y_d}{\delta x_2} & \cdots & \frac{\delta y_d}{\delta x_c} \end{bmatrix}$$

$$\frac{\delta L}{\delta x_2} = \sum_{i=1}^{d} \frac{\delta L}{\delta y_i} \frac{\delta y_i}{\delta x_2}$$

Backpropagation

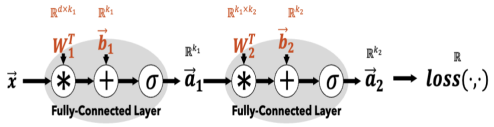
Backpropagation - A reverse-mode automatic differentiation algorithm commonly used to efficiently compute parameter gradients when training neural networks via gradient descent.

Builds off two simple observations / ideas:

1) Neural networks tend to have lower dimensional outputs than inputs.

2) We shouldn't recompute something we already know.

Gradients of Our Parameters



$$\frac{\delta loss}{\delta W^{(1)}} = \frac{\delta loss}{\delta a^{(2)}} * \frac{\delta a^{(2)}}{\delta a^{(1)}} * \frac{\delta a^{(1)}}{\delta W^{(1)}}$$

Dim: $1\times(dk_1)$ $1\times k_2$ $k_2\times k_1$ $k_1\times(dk_1)$

Questions Break!

Should I multiply these matrices left-to-right or right-to-left?

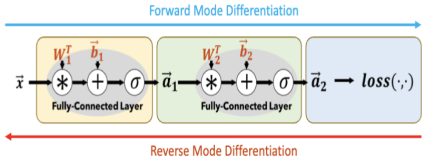
Hint: The computational complexity of multiplying an $a\times b$ and $b\times c$ matrix is $O(abc)$

$$\frac{\delta loss}{\delta W^{(1)}} = \frac{\delta loss}{\delta a^{(2)}} * \frac{\delta a^{(2)}}{\delta a^{(1)}} * \frac{\delta a^{(1)}}{\delta W^{(1)}}$$

$O(k_2dk_1)$ $O(k_2k_1dk_1)$ $O(k_2k_1)$ $O(k_1dk_1)$

Dim: $1\times k_2$ $k_2\times k_1$ $k_1\times(dk_1)$

Seed 1) Forward vs. Reverse Mode Differentiation

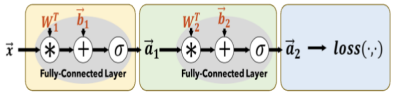


Forward Mode Differentiation
(input to output, dim input << dim output)

$$\frac{\delta loss}{\delta W_1} = \frac{\delta loss}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta \vec{a}_1} * \frac{\delta \vec{a}_1}{\delta W_1}$$

Reverse Mode Differentiation
(output to input, dim output << dim input)

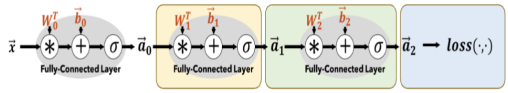
Gradients of Our Parameters / The Seeds of Backprop



$$\frac{\delta L}{\delta W_2} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta W_2} \qquad \frac{\delta L}{\delta \vec{b}_2} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta \vec{b}_2}$$

$$\frac{\delta L}{\delta W_1} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta \vec{a}_1} * \frac{\delta \vec{a}_1}{\delta W_1} \qquad \frac{\delta L}{\delta \vec{b}_1} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta \vec{a}_1} * \frac{\delta \vec{a}_1}{\delta \vec{b}_1}$$

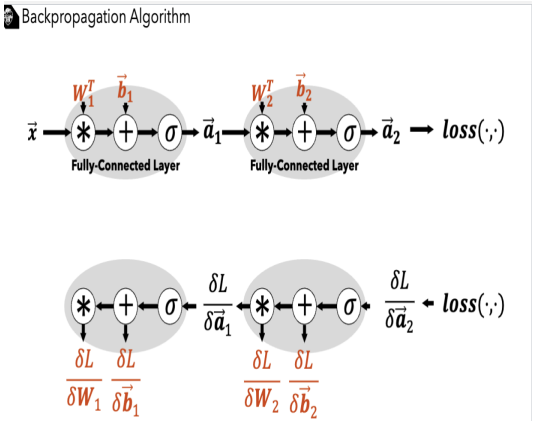
Gradients of Our Parameters / The Seeds of Backprop

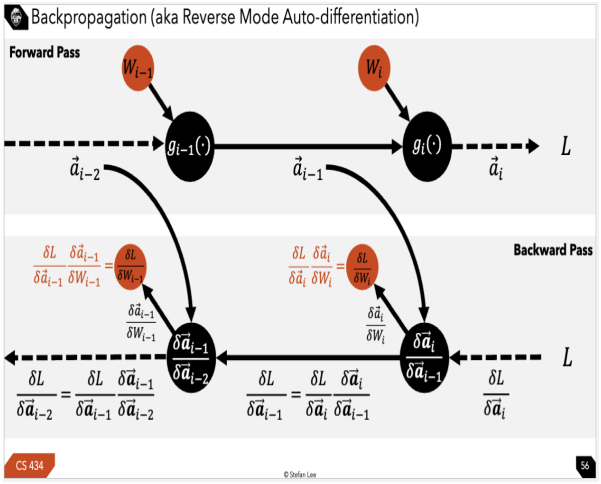


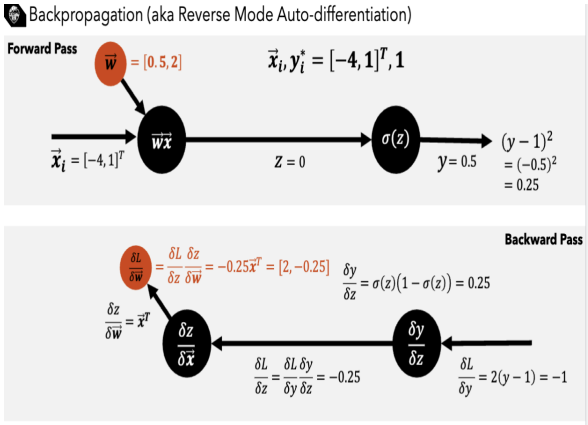
$$\frac{\delta L}{\delta W_2} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta W_2} \qquad \frac{\delta L}{\delta \vec{b}_2} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta \vec{b}_2}$$

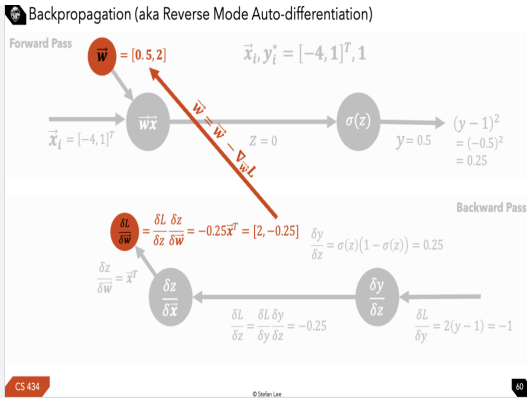
$$\frac{\delta L}{\delta W_1} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta \vec{a}_1} * \frac{\delta \vec{a}_1}{\delta W_1} \qquad \frac{\delta L}{\delta \vec{b}_1} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta \vec{a}_1} * \frac{\delta \vec{a}_1}{\delta \vec{b}_1}$$

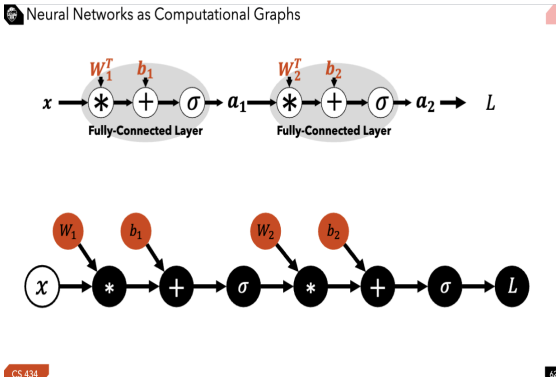
$$\frac{\delta L}{\delta W_0} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta \vec{a}_1} * \frac{\delta \vec{a}_1}{\delta \vec{a}_0} * \frac{\delta \vec{a}_1}{\delta W_0} \qquad \frac{\delta L}{\delta \vec{b}_0} = \frac{\delta L}{\delta \vec{a}_2} * \frac{\delta \vec{a}_2}{\delta \vec{a}_1} * \frac{\delta \vec{a}_1}{\delta \vec{a}_0} * \frac{\delta \vec{a}_0}{\delta \vec{b}_0}$$

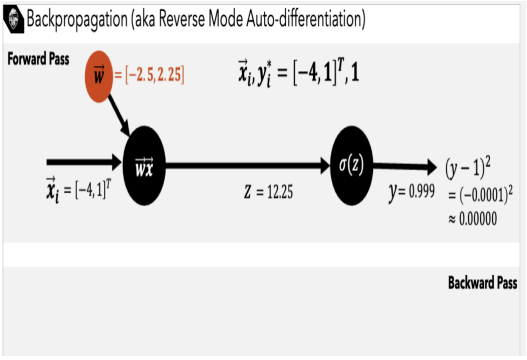






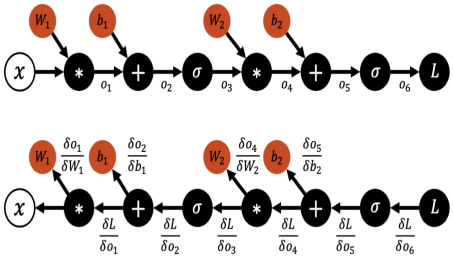






Computational graph:

Neural Networks as Computational Graphs

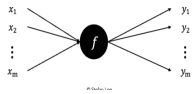


Backpropagation on Computational Graphs

Computational Graph - A directed acyclic graph (DAG) with vertices corresponding to computation and edges to intermediate results of the computation.

For backprop to work, each node needs to define:

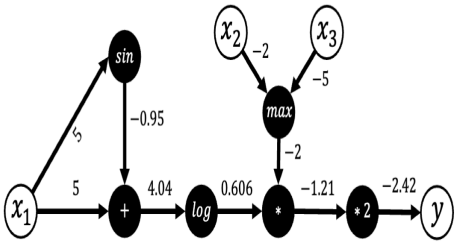
- Its **forward** computation $y_1, \ldots, y_k = f(x_1, \ldots, x_m)$
- Its **backward** computation: $\frac{\delta y_i}{\delta x_j}\ \forall i, j$



CS 434 65

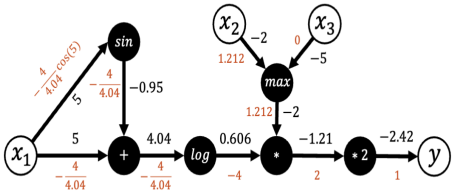
Backpropagation on Computational Graphs

$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$



Backpropagation on Computational Graphs

$$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$$

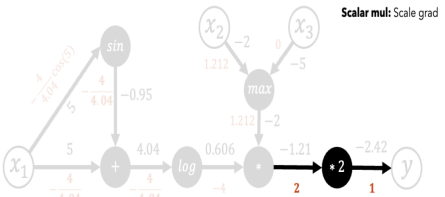


$$\frac{\delta_y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta_y}{\delta x_2} = 1.212 \qquad \frac{\delta_y}{\delta x_3} = 0$$

Backpropagation on Computational Graphs

$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$

Scalar mul: Scale grad



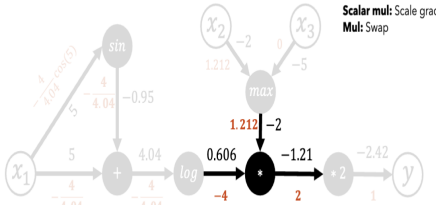
$$\frac{\delta_y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta_y}{\delta x_2} = 1.212 \qquad \frac{\delta_y}{\delta x_3} = 0$$

S 434

Backpropagation on Computational Graphs

$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$

Scalar mul: Scale grad
Mul: Swap



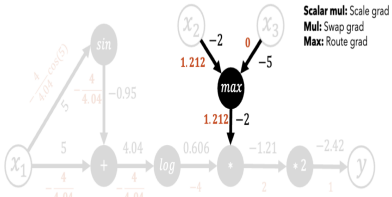
$$\frac{\delta_y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta_y}{\delta x_2} = 1.212 \qquad \frac{\delta_y}{\delta x_3} = 0$$

69

Backpropagation on Computational Graphs

$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$

Scalar mul: Scale grad
Mul: Swap grad
Max: Route grad

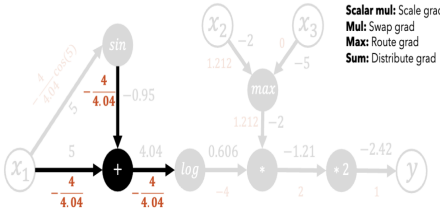


$$\frac{\delta_y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta_y}{\delta x_2} = 1.212 \qquad \frac{\delta_y}{\delta x_3} = 0$$

Backpropagation on Computational Graphs

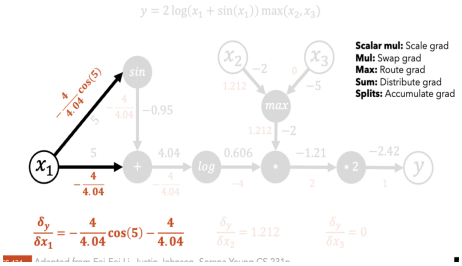
$y = 2\log(x_1 + \sin(x_1))\max(x_2, x_3)$

Scalar mul: Scale grad
Mul: Swap grad
Max: Route grad
Sum: Distribute grad

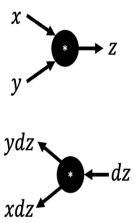


$$\frac{\delta_y}{\delta x_1} = -\frac{4}{4.04}\cos(5) - \frac{4}{4.04} \qquad \frac{\delta_y}{\delta x_2} = 1.212 \qquad \frac{\delta_y}{\delta x_3} = 0$$

Backpropagation on Computational Graphs



Modularized Implementation: Forward / Backward API




```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

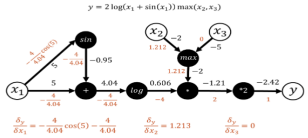
Modern Autograd Frameworks Make This Easy

Implement special variable types (e.g., Tensor in PyTorch) that:

- Override most operations with corresponding forward/backward APIs
- Dynamically build a computation graph as the variables are manipulated

Implement engine that can run backpropagation over the computation graph

Modern Autograd Frameworks -- PyTorch




```
import torch
import numpy as np

x = torch.tensor([5.0, -2.0, -5.0], requires_grad=True)
y = 2*torch.log( x[0] + torch.sin(x[0]))/np.log(10)*torch.max( x[1:])

y.backward()
print(x.grad)
```

```
tensor([-0.5518,  1.2130,  0.0000])
```

Modern Autograd Frameworks -- PyTorch

```
at::Tensor norm_backward(const at::Tensor & grad, const at::Tensor & self, const optional<at::Scalar> & p_, const at::Tensor & norm);
at::Tensor norm_backward(at::Tensor grad, const at::Tensor & self, const optional<at::Scalar> & p_, at::Tensor norm, at::IntArrayRef dim, bool keepdim);
at::Tensor pow_backward(at::Tensor grad, const at::Tensor & self, const at::Scalar & exponent_);
at::Tensor pow_backward_self(at::Tensor grad, const at::Tensor & self, const at::Tensor & exponent);
at::Tensor pow_backward_exponent(at::Tensor grad, const at::Tensor& self, const at::Tensor& exponent, at::Tensor result);
at::Tensor pow_backward_exponent(at::Tensor grad, const at::Scalar & base, const at::Tensor& exponent, at::Tensor result);
at::Tensor angle_backward(at::Tensor grad, const at::Tensor& self);
at::Tensor mul_tensor_backward(Tensor grad, Tensor other, ScalarType self_st);
at::Tensor div_tensor_self_backward(Tensor grad, Tensor other, ScalarType self_st);
at::Tensor div_tensor_other_backward(Tensor grad, Tensor self, Tensor other);
at::Tensor mvlgamma_backward(at::Tensor grad, const at::Tensor & self, int64_t p);
at::Tensor permute_backwards(const at::Tensor & grad, at::IntArrayRef fwd_dims);
at::Tensor rad2deg_backward(const at::Tensor& grad);
at::Tensor deg2rad_backward(const at::Tensor& grad);
at::Tensor unsqueeze_multiple(const at::Tensor & t, at::IntArrayRef dim, size_t n_dims);
at::Tensor sum_backward(const at::Tensor & grad, at::IntArrayRef sizes, at::IntArrayRef dims, bool keepdim);
at::Tensor nansum_backward(const at::Tensor & grad, const at::Tensor & self, at::IntArrayRef dims, bool keepdim);
std::vector<int64_t> reverse_list(const at::IntArrayRef list);
at::Tensor reverse_dim(const at::Tensor& t, int64_t dim);
at::Tensor prod_safe_zeros_backward(const at::Tensor &grad, const at::Tensor& inp, int64_t dim);
at::Tensor prod_backward(const at::Tensor& grad, const at::Tensor& input, const at::Tensor& result);
at::Tensor prod_backward(at::Tensor grad, const at::Tensor& input, at::Tensor result, int64_t dim, bool keepdim);
at::Tensor solve_backward_self(const at::Tensor & grad, const at::Tensor & self, const at::Tensor & A);
at::Tensor solve_backward_A(const at::Tensor & grad, const at::Tensor & self, const at::Tensor & A, const at::Tensor & solution);
at::Tensor cumsum_backward(const at::Tensor & x, int64_t dim);
at::Tensor logsumexp_backward(at::Tensor grad, const at::Tensor & self, at::Tensor result, at::IntArrayRef dim, bool keepdim);
at::Tensor logcumsumexp_backward(at::Tensor grad, const at::Tensor & self, at::Tensor result, int64_t dim);
```

Modern Autograd Frameworks -- PyTorch

```
Tensor sum_backward(const Tensor & grad, IntArrayRef sizes, IntArrayRef dims, bool keepdim) {
  if (!keepdim && sizes.size() > 0) {
    if (dims.size()==1) {
      return grad.unsqueeze(dims[0]).expand(sizes);
    } else {
      Tensor res = unsqueeze_multiple(grad, dims, sizes.size());
      return res.expand(sizes);
    }
  } else {
    return grad.expand(sizes);
  }
}
```