Machine Learning Department
School of Computer Science
Carnegie Mellon University

Backpropagation

+

Deep Learning

Matt Gormley
Lecture 13
Feb. 26, 2023

Reminders

- **Homework 5: Neural Networks**
  - Out: Sun, Feb 26
  - Due: Fri, Mar 17 at 11:59pm

Algorithm

BACKPROPAGATION FOR A SIMPLE COMPUTATION GRAPH

Training

Backpropagation

Whiteboard

- From equation to forward computation
- Representing a simple function as a computation graph
- Backprogation on a simple computation graph

Differentiation Quiz #1:

Suppose x = 2 and z = 3, what are dy/dx and dy/dz for the function below? **Round your answer to the nearest integer**.

$$y = \exp(xz) + \frac{xz}{\log(x)} + \frac{\sin(\log(x))}{xz}$$

Training

Backpropagation

Simple Example: The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

Forward

$J = cos(u)$

$u = u_1 + u_2$

$u_1 = sin(t)$

$u_2 = 3t$

$t = x^2$

Training

Backpropagation

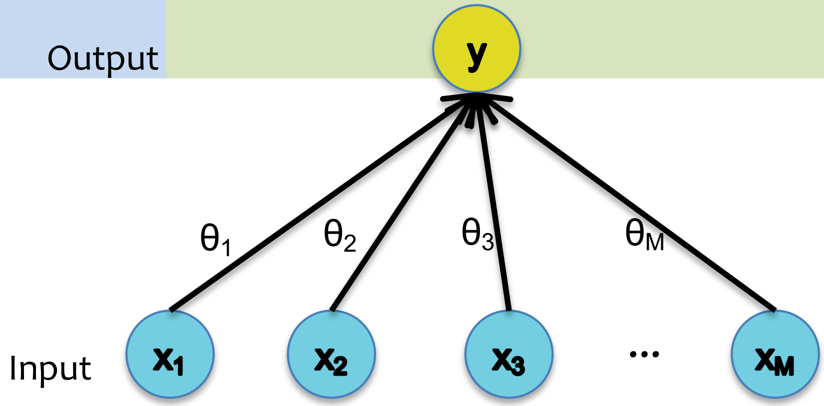
Simple Example: The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

Forward	Backward
$J = cos(u)$	$\frac{dJ}{du} \mathrel{+}= -sin(u)$
$u = u_1 + u_2$	$\frac{dJ}{du_1} \mathrel{+}= \frac{dJ}{du}\frac{du}{du_1}, \quad \frac{du}{du_1} = 1 \qquad \frac{dJ}{du_2} \mathrel{+}= \frac{dJ}{du}\frac{du}{du_2}, \quad \frac{du}{du_2} = 1$
$u_1 = sin(t)$	$\frac{dJ}{dt} \mathrel{+}= \frac{dJ}{du_1}\frac{du_1}{dt}, \quad \frac{du_1}{dt} = \cos(t)$
$u_2 = 3t$	$\frac{dJ}{dt} \mathrel{+}= \frac{dJ}{du_2}\frac{du_2}{dt}, \quad \frac{du_2}{dt} = 3$
$t = x^2$	$\frac{dJ}{dx} \mathrel{+}= \frac{dJ}{dt}\frac{dt}{dx}, \quad \frac{dt}{dx} = 2x$

Algorithm

BACKPROPAGATION FOR BINARY LOGISTIC REGRESSION

Training Backpropagation



Case 1: Logistic Regression

Forward

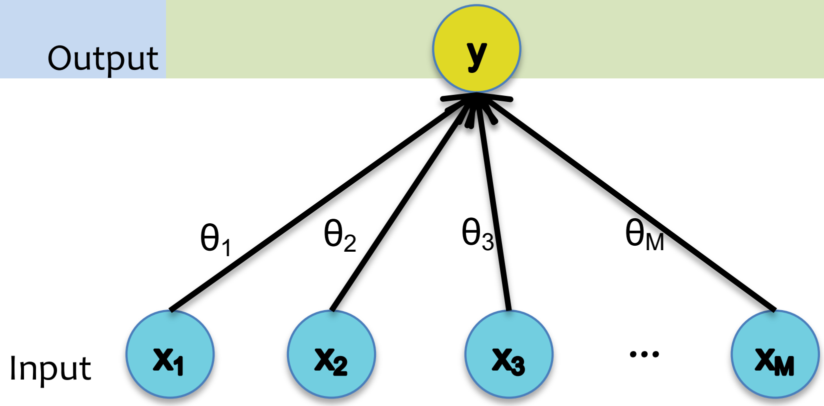
$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

Training Backpropagation

Case 1: Logistic Regression



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-a)}$$

$$a = \sum_{j=0}^{D} \theta_j x_j$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \quad \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

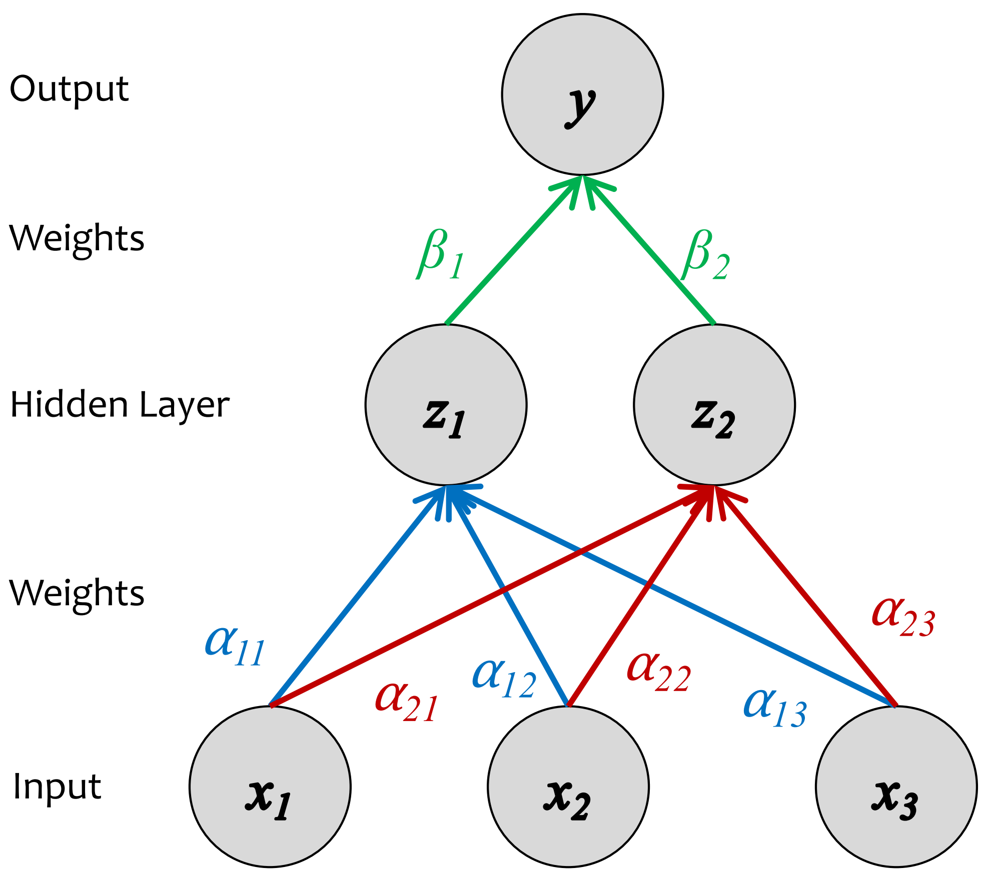
$$\frac{dJ}{d\theta_j} = \frac{dJ}{da}\frac{da}{d\theta_j}, \quad \frac{da}{d\theta_j} = x_j$$

$$\frac{dJ}{dx_j} = \frac{dJ}{da}\frac{da}{dx_j}, \quad \frac{da}{dx_j} = \theta_j$$

A 1-Hidden Layer Neural Network

TRAINING / FORWARD COMPUTATION / BACKWARD COMPUTATION

Training — Forward-Computation



(E) **Output (sigmoid)**
$y = \frac{1}{1+\exp(-b)}$

↑

(D) **Output (linear)**
$b = \sum_{j=0}^{D} \beta_j z_j$

↑

(C) **Hidden (sigmoid)**
$z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$

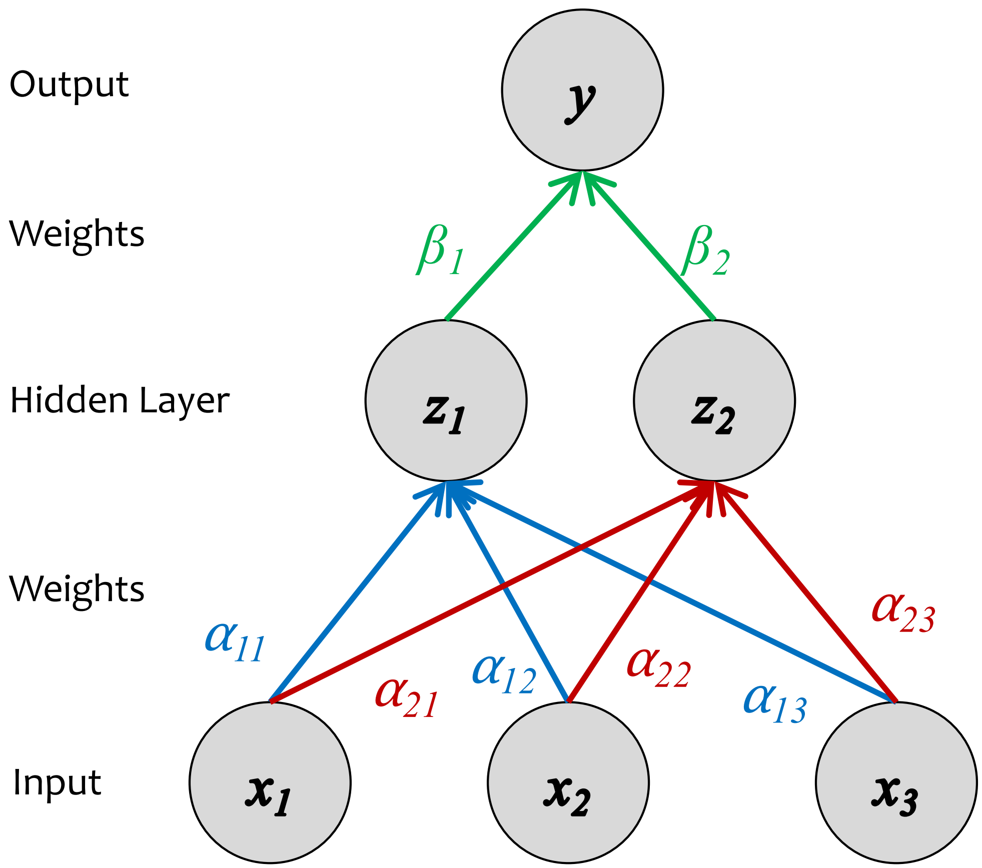
↑

(B) **Hidden (linear)**
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$

↑

(A) **Input**
Given $x_i, \ \forall i$

Training Forward-Computation



(F) **Loss**
$J = \frac{1}{2}(y - y^*)^2$

(E) **Output (sigmoid)**
$y = \frac{1}{1+\exp(-b)}$

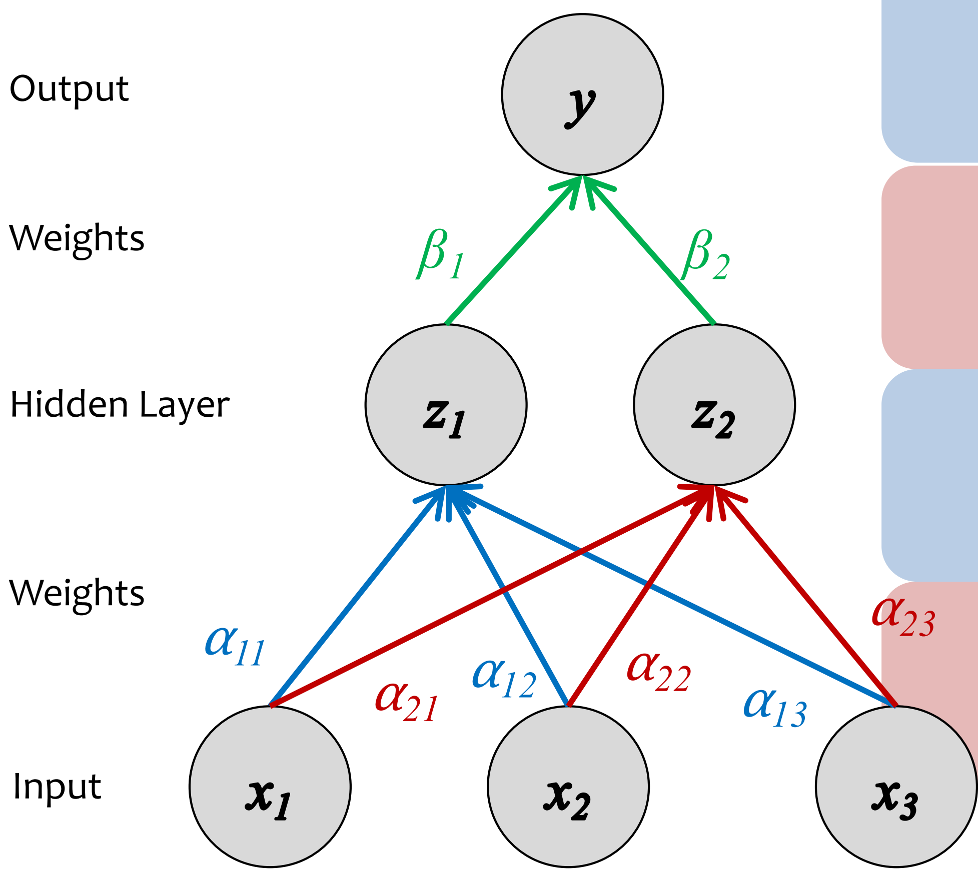
(D) **Output (linear)**
$b = \sum_{j=0}^{D} \beta_j z_j$

(C) **Hidden (sigmoid)**
$z_j = \frac{1}{1+\exp(-a_j)}, \;\; \forall j$

(B) **Hidden (linear)**
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \;\; \forall j$

(A) **Input**
Given $x_i, \;\; \forall i$

Training Forward-Computation



(F) **Loss**
$J = \frac{1}{2}(y - y^*)^2$

(E) **Output (sigmoid)**
$y = \frac{1}{1+\exp(-b)}$

(D) **Output (linear)**
$b = \sum_{j=0}^{D} \beta_j z_j$

(C) **Hidden (sigmoid)**
$z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$

(B) **Hidden (linear)**
$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \ \forall j$

(A) **Input**
Given $x_i, \ \forall i$

Training — SGD with Backprop

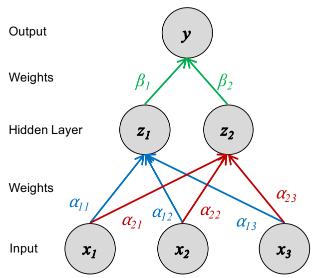
Example: 1-Hidden Layer Neural Network

Algorithm 1 Stochastic Gradient Descent (SGD)

procedure SGD(Training data $\mathcal{D}$, test data $\mathcal{D}_t$)
    Initialize parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$
    for $e \in \{1, 2, \ldots, E\}$ **do**
        for $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ **do**
            Compute neural network layers:
            $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J) = \text{NNFORWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
            Compute gradients via backprop:
            $$\left.\begin{array}{l} \mathbf{g}_{\boldsymbol{\alpha}} = \nabla_{\boldsymbol{\alpha}} J \\ \mathbf{g}_{\boldsymbol{\beta}} = \nabla_{\boldsymbol{\beta}} J \end{array}\right\} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{o})$$
            Update parameters:
            $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \gamma \mathbf{g}_{\boldsymbol{\alpha}}$
            $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \gamma \mathbf{g}_{\boldsymbol{\beta}}$
        Evaluate training mean cross-entropy $J_{\mathcal{D}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
        Evaluate test mean cross-entropy $J_{\mathcal{D}_t}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
        validation
    return parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$

Training — Backpropagation

Case 2: Neural Network



Forward

$$J = y^* \log y + (1 - y^*) \log(1 - y)$$

$$y = \frac{1}{1 + \exp(-b)}$$

$$b = \sum_{j=0}^{D} \beta_j z_j$$

$$z_j = \frac{1}{1 + \exp(-a_j)}$$

$$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$$

Backward

$$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$$

$$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \quad \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$$

$$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \quad \frac{db}{d\beta_j} = z_j$$

$$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \quad \frac{db}{dz_j} = \beta_j$$

$$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \quad \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$$

$$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \quad \frac{da_j}{d\alpha_{ji}} = x_i$$

$$\frac{dJ}{dx_i} = \sum_{j=0}^{D} \frac{dJ}{da_j}\frac{da_j}{dx_i}, \quad \frac{da_j}{dx_i} = \alpha_{ji}$$

Training

Backpropagation

$\frac{\partial J}{\partial J} = 1$

Case 2:

	Forward	Backward
Loss	$J = y^* \log y + (1 - y^*) \log(1 - y)$	$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$
Sigmoid	$y = \frac{1}{1 + \exp(-b)}$	$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \quad \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$
Linear	$b = \sum_{j=0}^{D} \beta_j z_j$	$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \quad \frac{db}{d\beta_j} = z_j$ $\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \quad \frac{db}{dz_j} = \beta_j$
Sigmoid	$z_j = \frac{1}{1 + \exp(-a_j)}$	$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \quad \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$
Linear	$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \quad \frac{da_j}{d\alpha_{ji}} = x_i$ $\frac{dJ}{dx_i} = \sum_{j=0}^{D} \frac{dJ}{da_j}\frac{da_j}{dx_i}, \quad \frac{da_j}{dx_i} = \alpha_{ji}$

Derivative of a Sigmoid

First suppose that $s = \sigma(b)$

$$s = \frac{1}{1 + \exp(-b)} \tag{1}$$

To obtain the simplified form of the derivative of a sigmoid.

$$
\begin{aligned}
\frac{ds}{db} &= \frac{\exp(-b)}{(\exp(-b)+1)^2} && (2)\\
&= \frac{\exp(-b)+1-1}{(\exp(-b)+1+1-1)^2} && (3)\\
&= \frac{\exp(-b)+1-1}{(\exp(-b)+1)^2} && (4)\\
&= \frac{\exp(-b)+1}{(\exp(-b)+1)^2} - \frac{1}{(\exp(-b)+1)^2} && (5)\\
&= \frac{1}{(\exp(-b)+1)} - \frac{1}{(\exp(-b)+1)^2} && (6)\\
&= \frac{1}{(\exp(-b)+1)} - \left(\frac{1}{(\exp(-b)+1)}\frac{1}{(\exp(-b)+1)}\right) && (7)\\
&= \frac{1}{(\exp(-b)+1)}\left(1 - \frac{1}{(\exp(-b)+1)}\right) && (8)\\
&= s(1-s) && (9)
\end{aligned}
$$

Training Backpropagation

Case 2:

	Forward	Backward
Loss	$J = y^* \log y + (1 - y^*) \log(1 - y)$	$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$
Sigmoid	$y = \frac{1}{1 + \exp(-b)}$	$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \boxed{\frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}}$
Linear	$b = \sum_{j=0}^{D} \beta_j z_j$	$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$ $\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$
Sigmoid	$z_j = \frac{1}{1 + \exp(-a_j)}$	$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \boxed{\frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}}$
Linear	$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$ $\frac{dJ}{dx_i} = \sum_{j=0}^{D} \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \alpha_{ji}$

Training

Backpropagation

cause of the vanishing gradient problem!

Case 2:

	Forward	Backward
Loss	$J = y^* \log y + (1 - y^*) \log(1 - y)$	$\frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$
Sigmoid	$y = \frac{1}{1 + \exp(-b)}$	$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \boxed{\frac{dy}{db} = y(1 - y)}$
Linear	$b = \sum_{j=0}^{D} \beta_j z_j$	$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$ $\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$
Sigmoid	$z_j = \frac{1}{1 + \exp(-a_j)}$	$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \boxed{\frac{dz_j}{da_j} = z_j(1 - z_j)}$
Linear	$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$ $\frac{dJ}{dx_i} = \sum_{j=0}^{D} \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \alpha_{ji}$

SGD with Backprop

Example: 1-Hidden Layer Neural Network

Algorithm 1 Stochastic Gradient Descent (SGD)

procedure SGD(Training data $\mathcal{D}$, test data $\mathcal{D}_t$)
    Initialize parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$
    for $e \in \{1, 2, \ldots, E\}$ **do**
        for $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ **do**
            Compute neural network layers:
            $\mathbf{o} = \texttt{object}(\mathbf{x}, \mathbf{a}, \mathbf{b}, \mathbf{z}, \hat{\mathbf{y}}, J) = \text{NNF}\textsc{orward}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta})$
            Compute gradients via backprop:
            $\left.\begin{array}{l} \mathbf{g}_{\boldsymbol{\alpha}} = \nabla_{\boldsymbol{\alpha}} J \\ \mathbf{g}_{\boldsymbol{\beta}} = \nabla_{\boldsymbol{\beta}} J \end{array}\right\} = \text{NNB}\textsc{ackward}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{o})$
            Update parameters:
            $\boldsymbol{\alpha} \leftarrow \boldsymbol{\alpha} - \gamma \mathbf{g}_{\boldsymbol{\alpha}}$
            $\boldsymbol{\beta} \leftarrow \boldsymbol{\beta} - \gamma \mathbf{g}_{\boldsymbol{\beta}}$
        Evaluate training mean cross-entropy $J_{\mathcal{D}}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
        Evaluate test mean cross-entropy $J_{\mathcal{D}_t}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
    return parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$

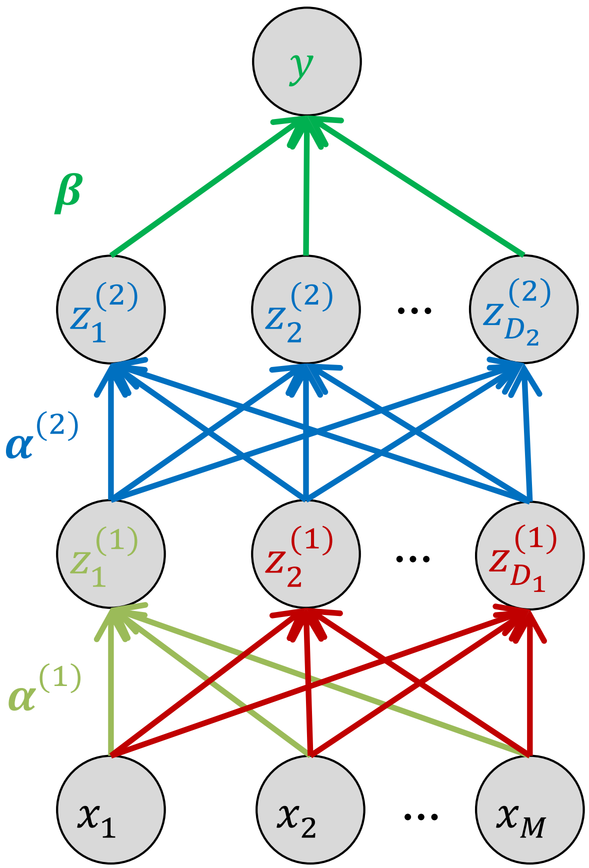
A 2-Hidden Layer Neural Network

TRAINING / FORWARD COMPUTATION / BACKWARD COMPUTATION

Backpropagation

Recall: Our 2-Hidden Layer Neural Network

Question: How do we train this model?



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

$\beta_0 \in \mathbb{R}$

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

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

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

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

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

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

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

Training

Backpropagation

Whiteboard

- Example: Backpropagation for Neural Network with 2-Hidden Layers
  - SGD Training
  - Forward Computation
  - Computation Graph
  - Backward Computation

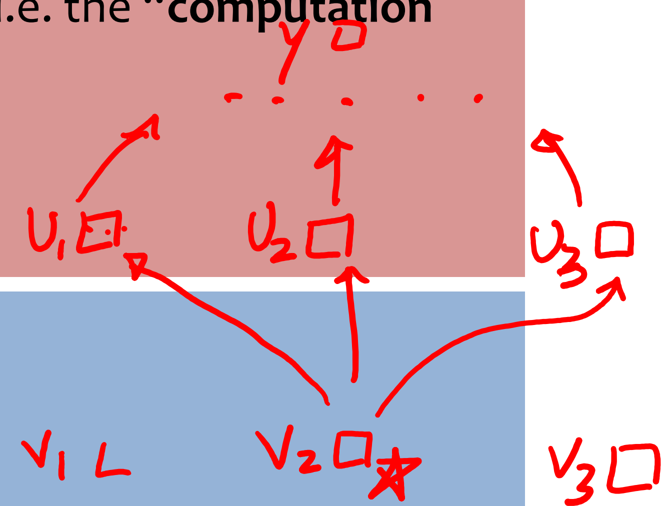
THE BACKPROPAGATION ALGORITHM

Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation

1. Write an **algorithm** for evaluating the function y = f(**x**). The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the "**computation graph"**)
2. Visit each node in **topological order**.
   For variable u_i with inputs $v_1,\ldots, v_N$
   a. Compute $u_i = g_i(v_1,\ldots, v_N)$
   b. Store the result at the node



Backward Computation (Version A)

1. **Initialize** dy/dy = 1.
2. Visit each node v_j in **reverse topological order**.
   Let $u_1,\ldots, u_M$ denote all the nodes with v_j as an input
   Assuming that y = h(**u**) = h($u_1,\ldots, u_M$)
   and **u** = g(**v**) or equivalently $u_i = g_i(v_1,\ldots, v_j,\ldots, v_N)$ for all i
   a. We already know dy/du_i for all i
   b. Compute dy/dv_j as below (Choice of algorithm ensures computing (du_i/dv_j) is easy)

$$\frac{dy}{dv_j} = \sum_{i=1}^{M} \frac{dy}{du_i}\frac{du_i}{dv_j}$$

Return partial derivatives dy/du_i for all variables

Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

<u>Forward Computation</u>

1. Write an **algorithm** for evaluating the function $y = f(\mathbf{x})$. The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the "**computation graph**")
2. Visit each node in **topological order**.
   For variable u_i with inputs $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   b. Store the result at the node

<u>Backward Computation (Version B)</u>

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
   For variable $u_i = g_i(v_1, \ldots, v_N)$
   a. We already know dy/du_i
   b. Increment dy/dv_j by $(dy/du_i)(du_i/dv_j)$
   (Choice of algorithm ensures computing (du_i/dv_j) is easy)

Return partial derivatives dy/du_i for all variables

Training

Backpropagation

Why is the backpropagation algorithm efficient?

1. Reuses **computation from the forward pass** in the backward pass
2. Reuses **partial derivatives** throughout the backward pass (*but only if the algorithm reuses shared computation in the forward pass*)

   (Key idea: partial derivatives in the backward pass should be thought of as variables stored for reuse)

Background

A Recipe for

1. Given training dat

$$\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^{N}$$

2. Choose each of th

– Decision function

$$\hat{y} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i)$$

– Loss function

$$\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}_i) \in \mathbb{R}$$

Gradients

Backpropagation can compute this gradient!

And it's a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!

opposite the gradient)

$$\boldsymbol{\theta}^{(t} \quad {}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)$$

MATRIX CALCULUS

Q&A

Q: Do I need to know **matrix calculus** to derive the backprop algorithms used in this class?

A: Well, we've carefully constructed our assignments so that you do **not** need to know matrix calculus.

That said, it's pretty handy. So we *added matrix calculus to our learning objectives* for backprop.

Matrix Calculus

Let $y, x \in \mathbb{R}$ be scalars, $\mathbf{y} \in \mathbb{R}^M$ and $\mathbf{x} \in \mathbb{R}^P$ be vectors, and $\mathbf{Y} \in \mathbb{R}^{M \times N}$ and $\mathbf{X} \in \mathbb{R}^{P \times Q}$ be matrices

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

Matrix Calculus

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	$\frac{\partial y}{\partial \mathbf{X}} = \begin{bmatrix} \frac{\partial y}{\partial X_{11}} & \frac{\partial y}{\partial X_{12}} & \cdots & \frac{\partial y}{\partial X_{1Q}} \\ \frac{\partial y}{\partial X_{21}} & \frac{\partial y}{\partial X_{22}} & \cdots & \frac{\partial y}{\partial X_{2Q}} \\ \vdots & & & \vdots \\ \frac{\partial y}{\partial X_{P1}} & \frac{\partial y}{\partial X_{P2}} & \cdots & \frac{\partial y}{\partial X_{PQ}} \end{bmatrix}$

$x^{(i)}$

Matrix Calculus

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	$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_P} \end{bmatrix}$	$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_N}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_N}{\partial x_2} \\ \vdots & & & \\ \frac{\partial y_1}{\partial x_P} & \frac{\partial y_2}{\partial x_P} & \cdots & \frac{\partial y_N}{\partial x_P} \end{bmatrix}$

Matrix Calculus

Whenever you read about matrix calculus, you'll be confronted with two layout conventions:

Let $y, x \in \mathbb{R}$ be scalars, $\mathbf{y} \in \mathbb{R}^M$ and $\mathbf{x} \in \mathbb{R}^P$ be vectors.

1. In numerator layout:

$$\frac{\partial y}{\partial \mathbf{x}} \text{ is a } 1 \times P \text{ matrix, i.e. a row vector}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \text{ is an } M \times P \text{ matrix}$$

2. In denominator layout:

$$\frac{\partial y}{\partial \mathbf{x}} \text{ is a } P \times 1 \text{ matrix, i.e. a column vector}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \text{ is an } P \times M \text{ matrix}$$

In this course, **we use denominator layout.**

Why? This ensures that our gradients of the objective function with respect to some subset of parameters are the same shape as those parameters.

Matrix Calculus

Common Vector Derivatives

Let $\frac{\partial f(\vec{x})}{\partial \vec{x}} = \nabla_x f(\vec{x})$ be the vector derivative of f, $B \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^m$

Scalar Derivative	
$f(x) \to \frac{\partial f}{\partial x}$	
$bx \to b$	
$xb \to b$	
$x^2 \to 2x$	
$bx^2 \to 2bx$	

Vector Derivative	
$f(x) \to \frac{\partial f}{\partial \vec{x}}$	
$x^T B \to B$	
$x^T b \to b$	
$x^T x \to 2x$	
$x^T B x \to 2Bx$	

B symmetric

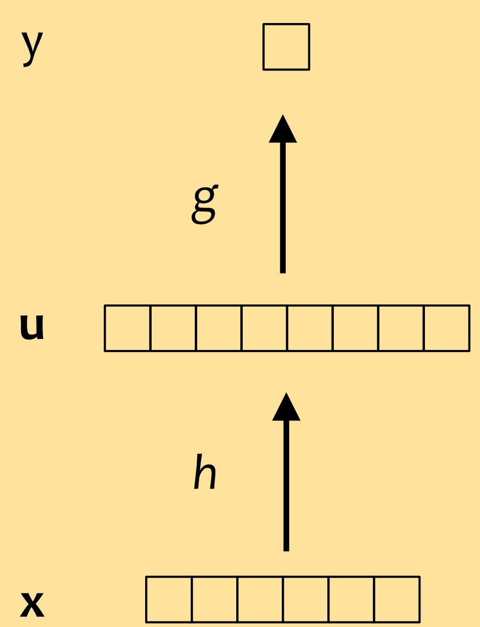
Matrix Calculus

Recall:

$$\frac{\partial y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \\ \frac{\partial y}{\partial x_2} \\ \vdots \\ \frac{\partial y}{\partial x_P} \end{bmatrix} \qquad \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_N}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_N}{\partial x_2} \\ \vdots & & & \\ \frac{\partial y_1}{\partial x_P} & \frac{\partial y_2}{\partial x_P} & \cdots & \frac{\partial y_N}{\partial x_P} \end{bmatrix}$$

Question: Q1

Suppose y = g(**u**) and **u** = h(**x**)



Which of the following is the correct definition of the chain rule?

Answer:

$$\frac{\partial y}{\partial \mathbf{x}} = \ldots$$

A. $\frac{\partial y}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{x}}$ 50%

B. $\frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial \mathbf{x}}$

C. $\frac{\partial y}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{x}}^T$

D. $\frac{\partial y}{\partial \mathbf{u}}^T \frac{\partial \mathbf{u}}{\partial \mathbf{x}}^T$

E. $(\frac{\partial y}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{x}})^T$

$\boxed{\text{F.}}$ None of the above 1%

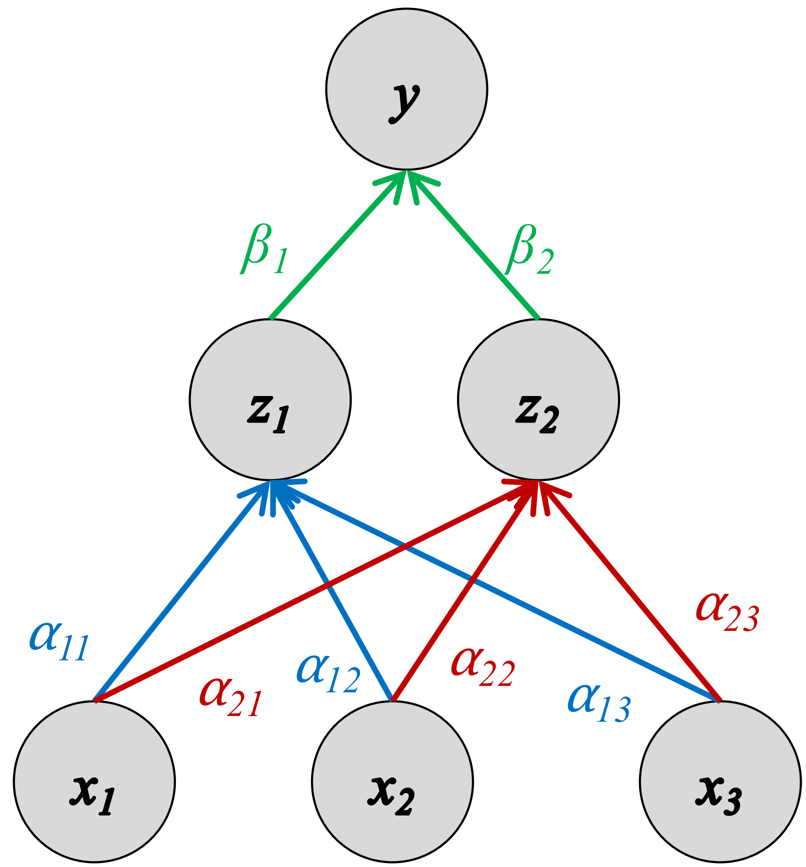
G = toxic

DRAWING A NEURAL NETWORK

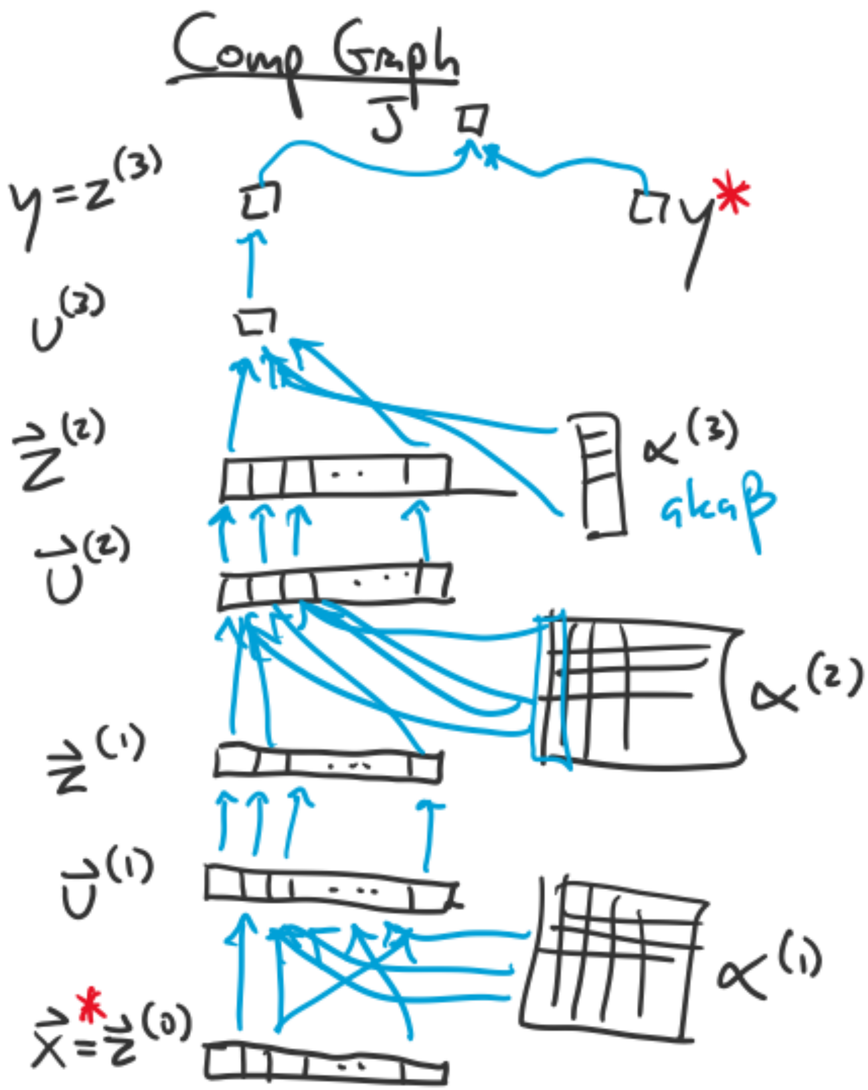
Ways of Drawing Neural Networks

Neural Network Diagram

- The diagram represents a neural network
- Nodes are **circles**
- One node per hidden unit
- Node is labeled with the variable corresponding to the hidden unit
- For a fully connected feed-forward neural network, a hidden unit is a nonlinear function of nodes in the previous layer
- *Edges are directed*
- Each edge is labeled with its weight (side note: we should be careful about ascribing how a matrix can be used to indicate the labels of the edges and pitfalls there)
- Other details:
  - Following standard convention, the intercept term is NOT shown as a node, but rather is assumed to be part of the non-linear function that yields a hidden unit. (i.e. its weight does NOT appear in the picture anywhere)
  - The diagram does NOT include any nodes related to the loss computation



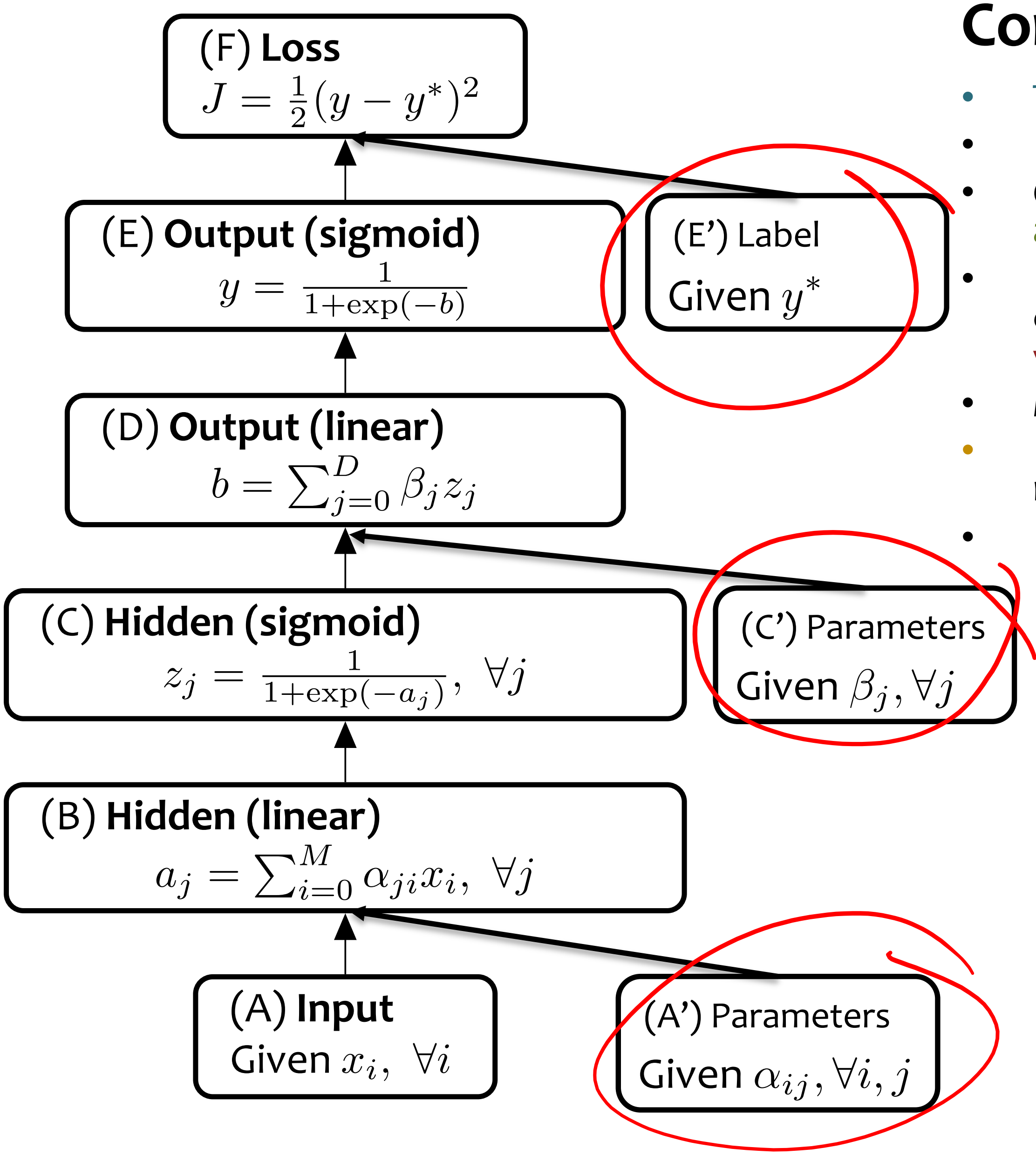
Ways of Drawing Neural Networks



Computation Graph

- The diagram represents an algorithm
- Nodes are **rectangles**
- One node per intermediate variable in the algorithm
- Node is labeled with the function that it computes (inside the box) and also the variable name (outside the box)
- *Edges are directed*
- Edges do not have labels (since they don't need them)
- For neural networks:
  - Each intercept term should appear as a node (if it's not folded in somewhere)
  - Each parameter should appear as a node
  - Each constant, e.g. a true label or a feature vector should appear in the graph
  - It's perfectly fine to include the loss

Ways of Drawing Neural Networks



Computation Graph

- The diagram represents an algorithm
- Nodes are **rectangles**
- One node per intermediate variable in the algorithm
- Node is labeled with the function that it computes (inside the box) and also the variable name (outside the box)
- *Edges are directed*
- Edges do not have labels (since they don't need them)
- For neural networks:
  - Each intercept term should appear as a node (if it's not folded in somewhere)
  - Each parameter should appear as a node
  - Each constant, e.g. a true label or a feature vector should appear in the graph
  - It's perfectly fine to include the loss

Ways of Drawing Neural Networks

Neural Network Diagram

- The diagram represents a neural network
- Nodes are **circles**
- One node per hidden unit
- Node is labeled with the variable corresponding to the hidden unit
- For a fully connected feed-forward neural network, a hidden unit is a nonlinear function of nodes in the previous layer
- *Edges are directed*
- Each edge is labeled with its weight (side note: we should be careful about ascribing how a matrix can be used to indicate the labels of the edges and pitfalls there)
- Other details:
  - Following standard convention, the intercept term is NOT shown as a node, but rather is assumed to be part of the non-linear function that yields a hidden unit. (i.e. its weight does NOT appear in the picture anywhere)
  - The diagram does NOT include any nodes related to the loss computation

Computation Graph

- The diagram represents an algorithm
- Nodes are **rectangles**
- One node per intermediate variable in the algorithm
- Node is labeled with the function that it computes (inside the box) and also the variable name (outside the box)
- *Edges are directed*
- Edges do not have labels (since they don't need them)
- For neural networks:
  - Each intercept term should appear as a node (if it's not folded in somewhere)
  - Each parameter should appear as a node
  - Each constant, e.g. a true label or a feature vector should appear in the graph
  - It's perfectly fine to include the loss

Important!

Some of these conventions are specific to 10-301/601. The literature abounds with varations on these conventions, but it's helpful to have *some* distinction nonetheless.

Summary

1. **Neural Networks…**
   – provide a way of learning features
   – are highly nonlinear prediction functions
   – (can be) a highly parallel network of logistic regression classifiers
   – discover useful hidden representations of the input
2. **Backpropagation…**
   – provides an efficient way to compute gradients
   – is a special case of reverse-mode automatic differentiation

Backprop Objectives

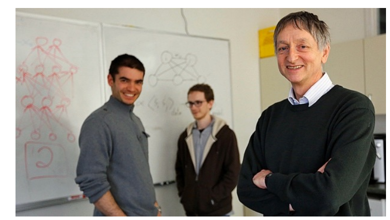
You should be able to…

- 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
- Apply the empirical risk minimization framework to learn 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.

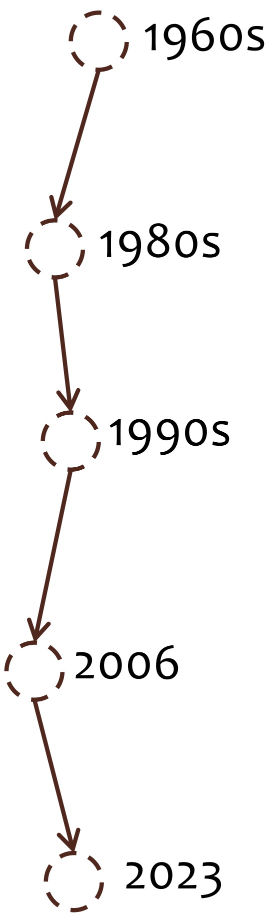
DEEP LEARNING

Why is everyone talking about Deep Learning?

- Because a lot of money is invested in it…
  - DeepMind: Acquired by Google for **$400 million**
  - Deep Learning startups command **millions of VC dollars**
  - Demand for deep learning engineers continually outpaces supply
- Because it made the **front page** of the New York Times



Why is everyone talking about Deep Learning?



Deep learning:

- Has won numerous pattern recognition competitions
- Does so with minimal feature engineering

This wasn't always the case!

Since 1980s: Form of models hasn't changed much, but lots of new tricks…

- More hidden units
- Better (online) optimization
- New nonlinear functions (ReLUs)
- Faster computers (CPUs and GPUs)

Backpropagation and Deep Learning

Convolutional neural networks (CNNs) and **recurrent neural networks** (RNNs) are simply fancy computation graphs (aka. hypotheses or decision functions).

Our recipe also applies to these models and (again) relies on the **backpropagation algorithm** to compute the necessary gradients.