10-301/10-601 Introduction to Machine Learning

Machine Learning Department
School of Computer Science
Carnegie Mellon University

Backpropagation

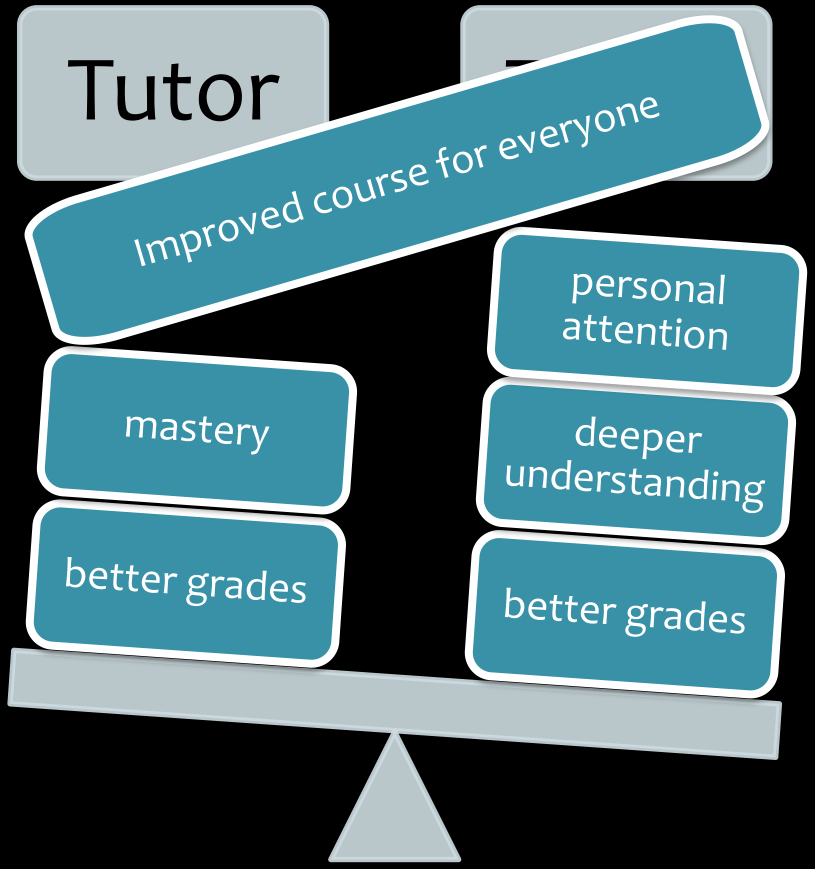
Matt Gormley & Henry Chai

Lecture 13
Oct. 7, 2024

Reminders

- Exam viewings
- Homework 4: Logistic Regression
  - Out: Mon, Sep 30
  - Due: Wed, Oct 9 at 11:59pm
- Homework 5: Neural Networks
  - Out: Wed, Oct 9
  - Due: Sun, Oct 27 at 11:59pm

Peer Tutoring



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!

A Recipe for

1. Given training dat

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

2. Choose each of t

– Decision functio

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

– Loss function

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

opposite the gradient)

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

Algorithm

BACKPROPAGATION FOR BINARY LOGISTIC REGRESSION

Training

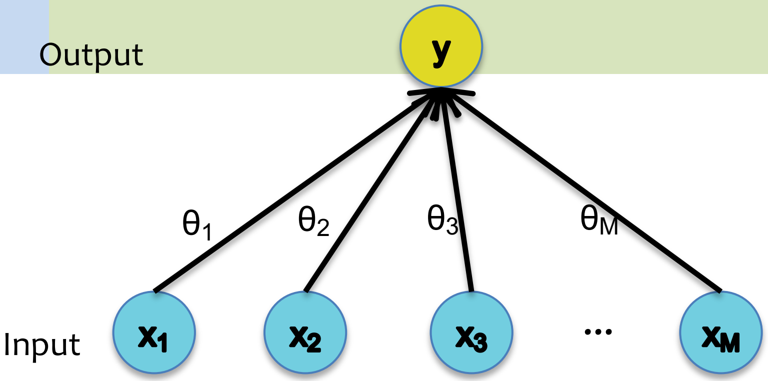
Backpropagation

Question: How do we compute this?

Answer: $g_{\theta_j} = \frac{\partial J}{\partial \theta_j} = \left(\frac{\partial J}{\partial a}\right)\left(\frac{\partial a}{\partial \theta_j}\right)$

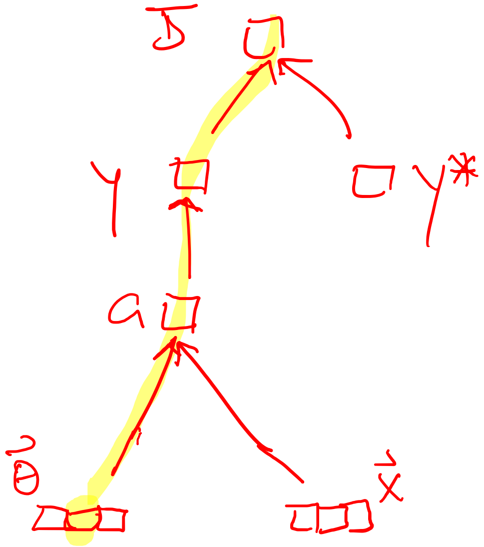
$= (g_a)(x_j)$

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



Case 1: Logistic Regression

Computation Graph



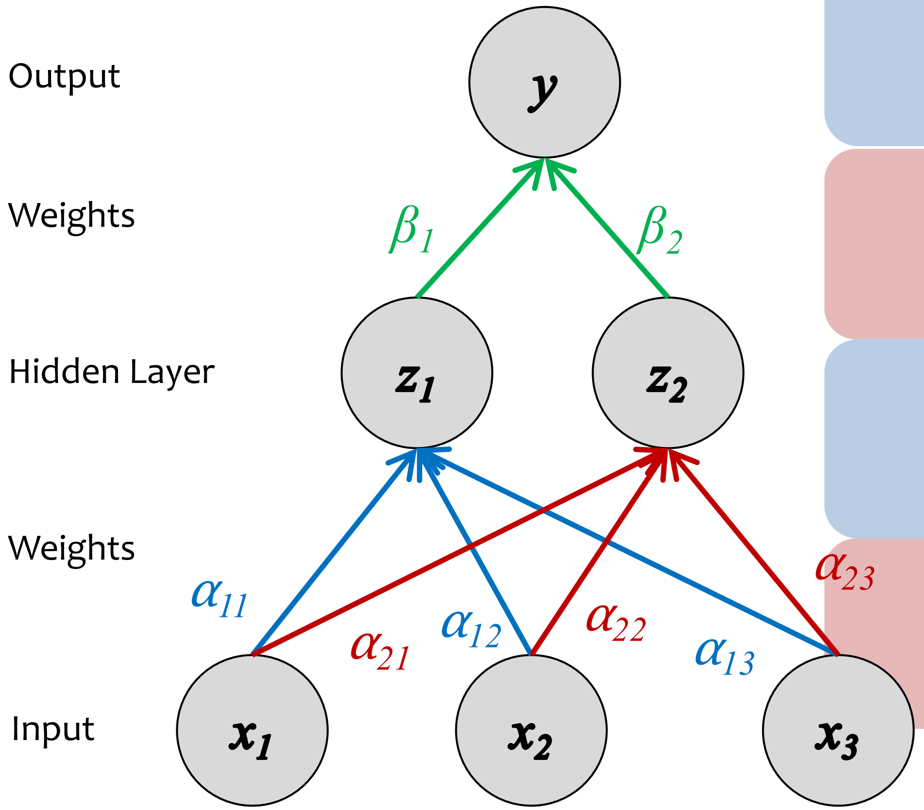
Forward	Backward
$J = y^* \log y + (1 - y^*) \log(1 - y)$	$g_y = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$
$y = \frac{1}{1 + \exp(-a)}$	$g_a = g_y \frac{\partial y}{\partial a}, \quad \frac{\partial y}{\partial a} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$
$a = \sum_{j=0}^{D} \theta_j x_j$	$g_{\theta_j} = \boxed{}$
	$g_{x_j} =$

$\frac{\partial J}{\partial a} = \frac{\partial J}{\partial y}\frac{\partial y}{\partial a}$

A 1-Hidden Layer Neural Network

TRAINING / FORWARD COMPUTATION / BACKWARD COMPUTATION

Training — Forward-Computation



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

quadratic (cross ent)

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

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

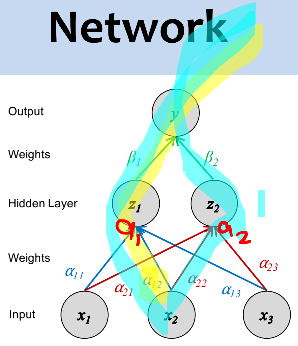
(C) Hidden (sigmoid)
$z_j = \frac{1}{1+\exp(-a_j)}, \ \forall j$

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

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

Case 2:
Neural
Network

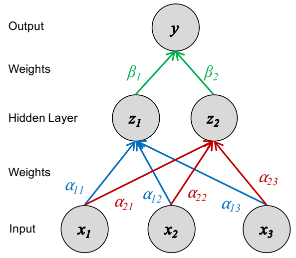
Backpropagation



	Forward	Backward	
Loss	$J = y^* \log y + (1 - y^*) \log(1 - y)$	$g_y = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$	1
Sigmoid	$y = \frac{1}{1 + \exp(-b)}$	$g_b = g_y \frac{\partial y}{\partial b}, \frac{\partial y}{\partial b} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}$	2
Linear	$b = \sum_{j=0}^{D} \beta_j z_j$	$g_{\beta_j} = g_b \frac{\partial b}{\partial \beta_j}, \frac{\partial b}{\partial \beta_j} = z_j$	3
		$g_{z_j} = g_b \frac{\partial b}{\partial z_j}, \frac{\partial b}{\partial z_j} = \beta_j$	4
Sigmoid	$z_j = \frac{1}{1 + \exp(-a_j)}$	$g_{a_j} = g_{z_j} \frac{\partial z_j}{\partial a_j}, \frac{\partial z_j}{\partial a_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}$	5
Linear	$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$g_{\alpha_{ji}} = g_{a_j} \frac{\partial a_j}{\partial \alpha_{ji}}, \frac{\partial a_j}{\partial \alpha_{ji}} = x_i$	6
		$g_{x_i} = \sum_{j=0}^{D} g_{a_j} \frac{\partial a_j}{\partial x_i}, \frac{\partial a_j}{\partial x_i} = \alpha_{ji}$	7

Case 2:
Neural
Network

Backpropagation



	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

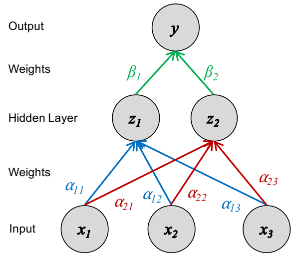
$$s = \frac{1}{1 + \exp(-b)} \tag{1}$$

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

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

Case 2: Neural Network

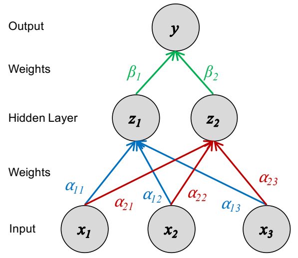
Backpropagation



	Forward	Backward
Loss	$J = y^* \log y + (1 - y^*) \log(1 - y)$	$g_y = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$
Sigmoid	$y = \frac{1}{1 + \exp(-b)}$	$g_b = g_y \frac{\partial y}{\partial b}, \boxed{\frac{\partial y}{\partial b} = \frac{\exp(-b)}{(\exp(-b) + 1)^2}}$
Linear	$b = \sum_{j=0}^{D} \beta_j z_j$	$g_{\beta_j} = g_b \frac{\partial b}{\partial \beta_j}, \frac{\partial b}{\partial \beta_j} = z_j$ $g_{z_j} = g_b \frac{\partial b}{\partial z_j}, \frac{\partial b}{\partial z_j} = \beta_j$
Sigmoid	$z_j = \frac{1}{1 + \exp(-a_j)}$	$g_{a_j} = g_{z_j} \frac{\partial z_j}{\partial a_j}, \boxed{\frac{\partial z_j}{\partial a_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2}}$
Linear	$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$g_{\alpha_{ji}} = g_{a_j} \frac{\partial a_j}{\partial \alpha_{ji}}, \frac{\partial a_j}{\partial \alpha_{ji}} = x_i$ $g_{x_i} = \sum_{j=0}^{D} g_{a_j} \frac{\partial a_j}{\partial x_i}, \frac{\partial a_j}{\partial x_i} = \alpha_{ji}$

Case 2:
Neural
Network

Backpropagation



This product of two small probabilities is what causes the vanishing gradient problem!

lines 5-6 lines 7-8

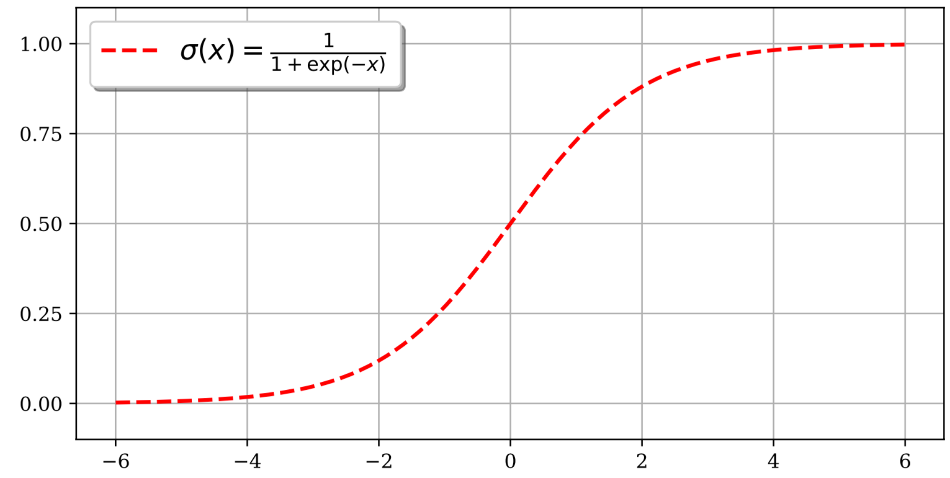
	Forward	Backward
Loss	$J = y^* \log y + (1 - y^*) \log(1 - y)$	$g_y = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1}$
Sigmoid	$y = \frac{1}{1 + \exp(-b)}$	$g_b = g_y \frac{\partial y}{\partial b}, \boxed{\frac{\partial y}{\partial b} = y(1 - y)}$
Linear	$b = \sum_{j=0}^{D} \beta_j z_j$	$g_{\beta_j} = g_b \frac{\partial b}{\partial \beta_j}, \frac{\partial b}{\partial \beta_j} = z_j$
		$g_{z_j} = g_b \frac{\partial b}{\partial z_j}, \frac{\partial b}{\partial z_j} = \beta_j$
Sigmoid	$z_j = \frac{1}{1 + \exp(-a_j)}$	$g_{a_j} = g_{z_j} \frac{\partial z_j}{\partial a_j}, \boxed{\frac{\partial z_j}{\partial a_j} = z_j(1 - z_j)}$
Linear	$a_j = \sum_{i=0}^{M} \alpha_{ji} x_i$	$g_{\alpha_{ji}} = g_{a_j} \frac{\partial a_j}{\partial \alpha_{ji}}, \frac{\partial a_j}{\partial \alpha_{ji}} = x_i$
		$g_{x_i} = \sum_{j=0}^{D} g_{a_j} \frac{\partial a_j}{\partial x_i}, \frac{\partial a_j}{\partial x_i} = \alpha_{ji}$

this causes the vanishing gradient problem

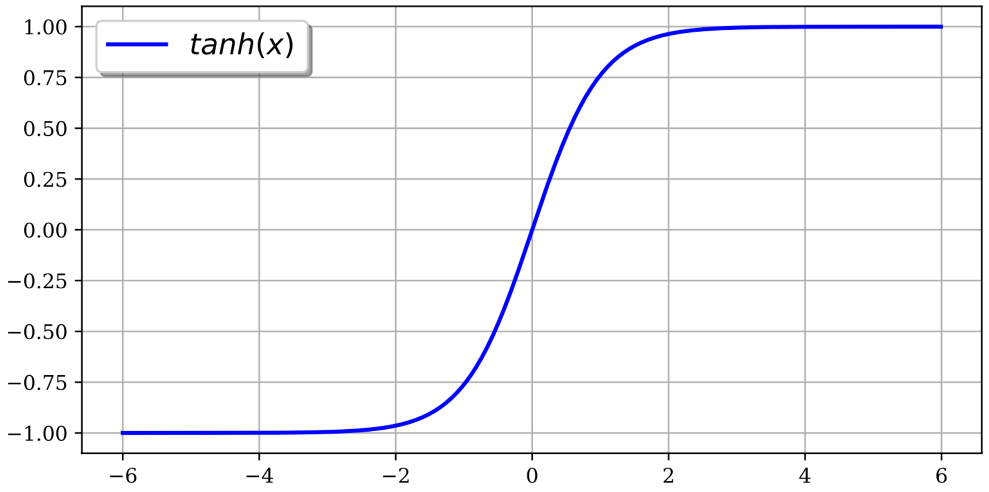
Activation Functions

- sigmoid, $\sigma(x)$
  - output in range (0,1)
  - good for probabilistic outputs
- hyperbolic tangent, tanh(x)
  - similar shape to sigmoid, but output in range (-1,+1)

Sigmoid (aka. logistic) function

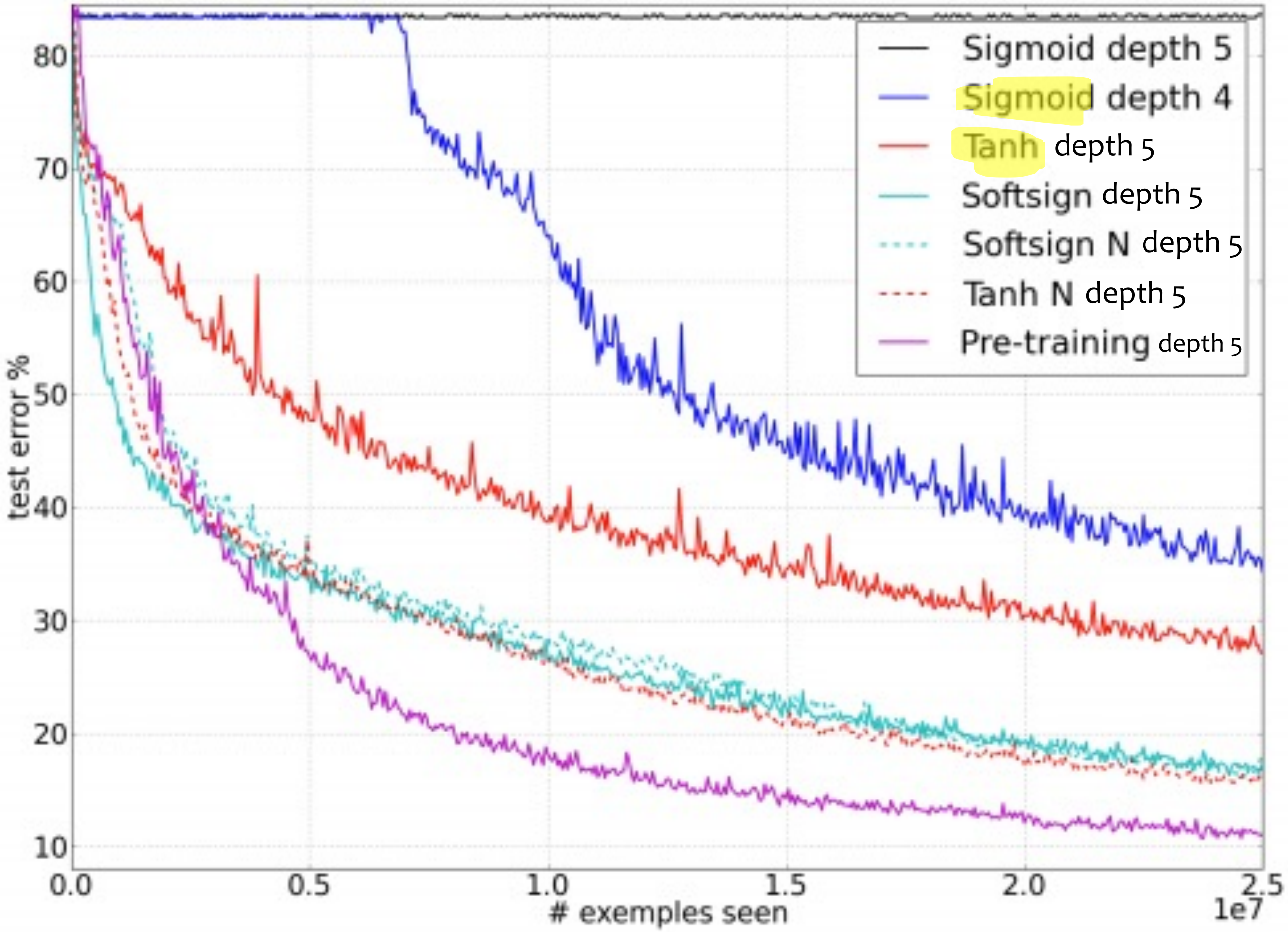


Hyperbolic tangent function



Understanding the difficulty of training deep feedforward neural networks

AI Stats 2010



sigmoid
vs.
tanh

Figure from Glorot & Bentio (2010)

Training

SGD with Backprop

Example: 1-Hidden Layer Neural Network

$\vec{\theta} = \text{concatenate}(\text{flatten}(\alpha), \beta)$

$\vec{g}_{\theta} = \text{concatenate}(\text{flatten}(g_\alpha), g_\beta)$

$\vec{\theta} \leftarrow \vec{\theta} - \gamma \vec{g}_{\vec{\theta}}$

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**   epochs
        for $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ **do**   examples
            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:
$$\left.\begin{array}{l} \boxed{\mathbf{g}_{\boldsymbol{\alpha}}} = \nabla_{\boldsymbol{\alpha}} J \\ \boxed{\mathbf{g}_{\boldsymbol{\beta}}} = \nabla_{\boldsymbol{\beta}} J \end{array}\right\} = \text{NNBACKWARD}(\mathbf{x}, \mathbf{y}, \boldsymbol{\alpha}, \boldsymbol{\beta}, \mathbf{o})$$
            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 $J_{\mathcal{D}_t}(\boldsymbol{\alpha}, \boldsymbol{\beta})$
    return parameters $\boldsymbol{\alpha}, \boldsymbol{\beta}$

Backprop

SGD

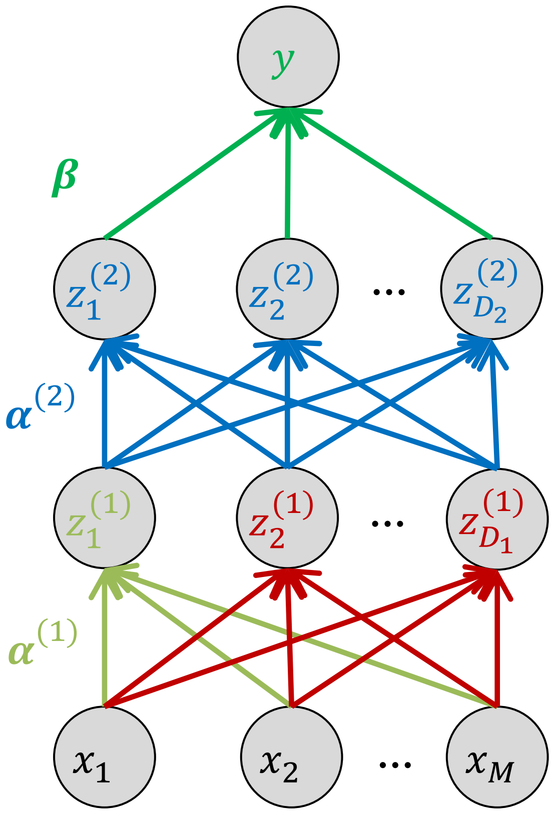
A 2-Hidden Layer Neural Network

TRAINING / FORWARD COMPUTATION / BACKWARD COMPUTATION

Backpropagation

Recall: Our 2-Hidden Layer Neural Network

Question: How do we train this model?



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

$\beta_0 \in \mathbb{R}$

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

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

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

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

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

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

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

Example: Neural Net Training (2-Hidden Layers)

- Consider a 2 hidden-layer NN
- Params are $\Theta = [\alpha^{(1)}, \alpha^{(2)}, \beta]$
- SGD Training:
  1. Initialize params $\alpha^{(1)}, \alpha^{(2)}, \beta$
  2. Iterate until convergence:
     a. Sample $i \sim \text{Uniform}(1, \ldots, N)$
     b. Compute gradient via Backprop

$$g_{\alpha^{(1)}} = \nabla_{\alpha^{(1)}} J^{(i)}(\Theta)$$
$$g_{\alpha^{(2)}} = \nabla_{\alpha^{(2)}} J^{(i)}(\Theta)$$
$$g_{\alpha^{(3)}} = \nabla_{\beta} J^{(i)}(\Theta)$$

$J^{(i)}(\Theta) = \ell(h_\Theta(\vec{x}^{(i)}), y^{(i)})$

     c. Update parameters

$$\alpha^{(1)} \leftarrow \alpha^{(1)} - \gamma g_{\alpha^{(1)}}$$
$$\alpha^{(2)} \leftarrow \alpha^{(2)} - \gamma g_{\alpha^{(2)}}$$
$$\beta \leftarrow \beta - \gamma g_{\beta}$$

Background:

$$\nabla J(\vec{a}, \vec{b}) = \nabla_{\vec{a},\vec{b}} J(\vec{a}, \vec{b}) = \begin{bmatrix} \partial J / \partial a_1 \\ \vdots \\ \partial J / \partial a_A \\ \partial J / \partial b_1 \\ \vdots \\ \partial J / \partial b_B \end{bmatrix}$$

$$\nabla_{\vec{a}} J(\vec{a}, \vec{b}) = \begin{bmatrix} \partial J / \partial a_1 \\ \vdots \\ \partial J / \partial a_A \end{bmatrix}$$

Example: Backpropagation (2-Hidden Layers)

without intercept terms

Given:
① Dec. fn. $\hat{y} = h(\vec{x}) = \sigma\Big(\underbrace{(\alpha^{(3)})^T}_{B}\, \sigma\big(\underbrace{(\alpha^{(2)})^T\, \underbrace{\sigma((\alpha^{(1)})^T \vec{x})}_{\vec{z}^{(1)}}}_{\vec{z}^{(2)}}\big)\Big)$

② Loss fn. $J = \ell(\hat{y}, y^*) = -\big(y^* \log(\hat{y}) + (1-y^*)\log(1-\hat{y})\big)$

③ Train ex. $(\vec{x}, y^*)$

④ Parameters. $\alpha^{(1)}, \alpha^{(2)}, \alpha^{(3)}$

Forward Comp.

$\vec{z}^{(0)} = \vec{x}$

For $i = 1,2,3$:

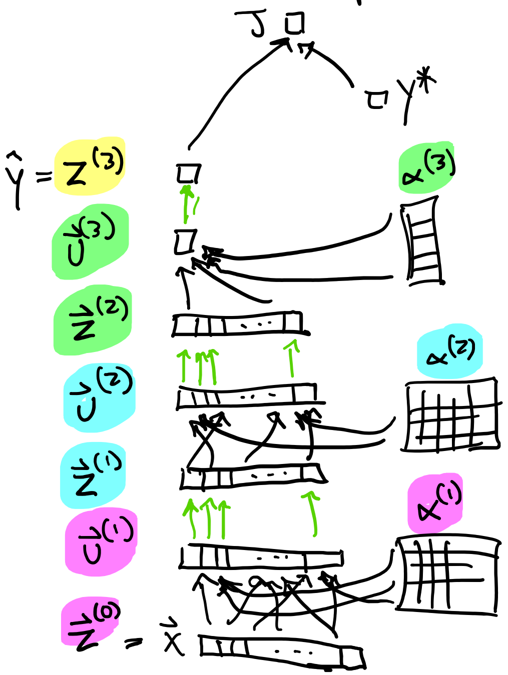
$\quad \vec{u}^{(i)} = (\alpha^{(i)})^T \vec{z}^{(i-1)}$

$\quad \vec{z}^{(i)} = \sigma(\vec{u}^{(i)})$

$\hat{y} = z^{(3)}$

$J = \ell(\hat{y}, y^*)$

Computation Graph



Backward Comp.

$g_J = [1]$

$g_{z^{(3)}} = g_{\hat{y}} = -\left(\frac{y^*}{\hat{y}} - \frac{(1-y^*)}{(1-\hat{y})}\right)$

for $i = 3,2,1$:

$\quad g_{u^{(i)}} = \ldots$

$\quad g_{\alpha^{(i)}} = \ldots$

$\quad g_{z^{(i-1)}} = \ldots$

HW5

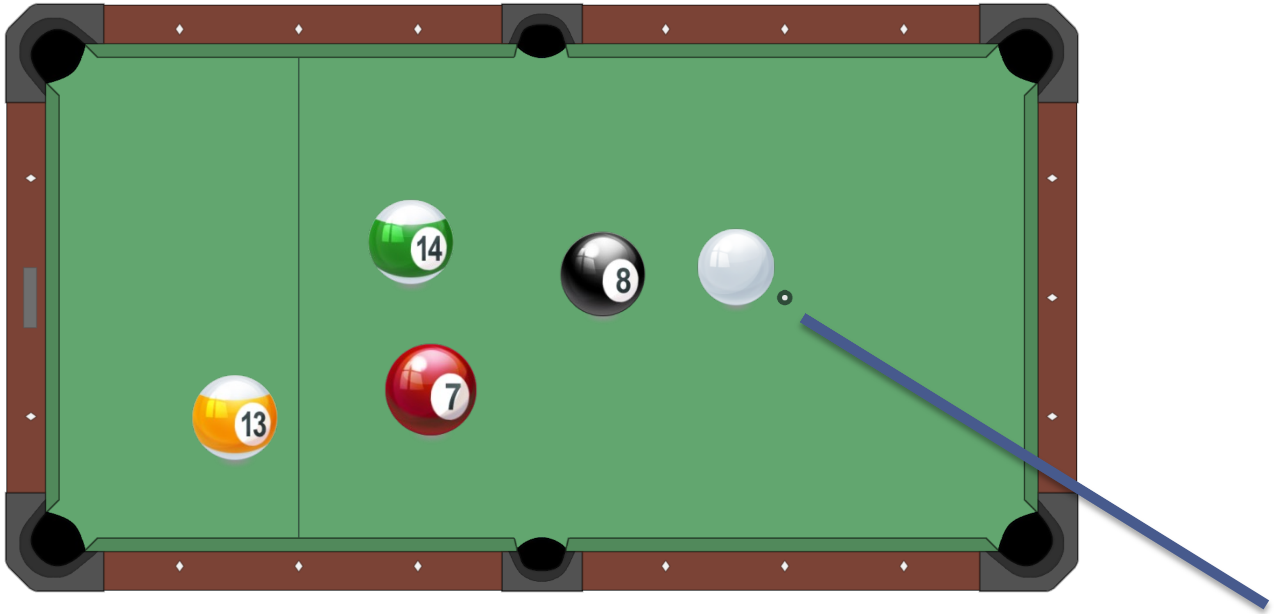
$g_{\vec{x}} = g_{\vec{z}^{(0)}}$

Example: Backpropagation (2-Hidden Layers)

Intuitions

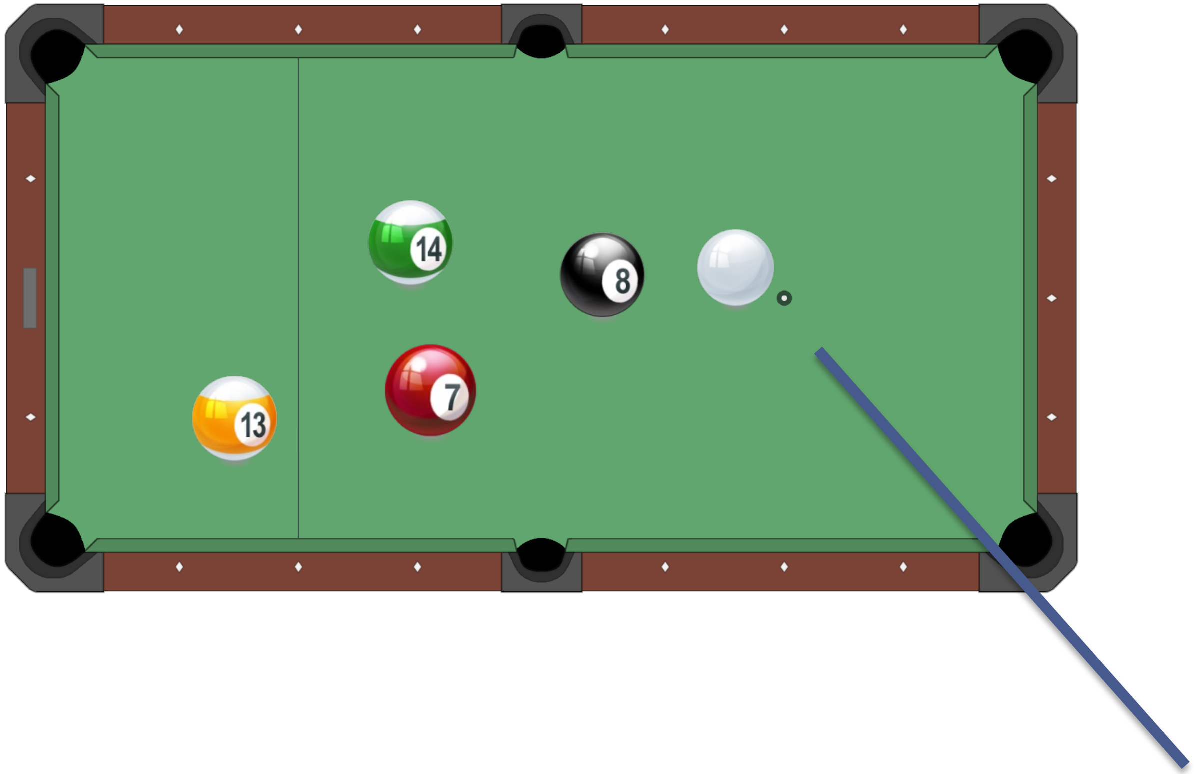
BACKPROPAGATION OF ERRORS

Error Back-Propagation



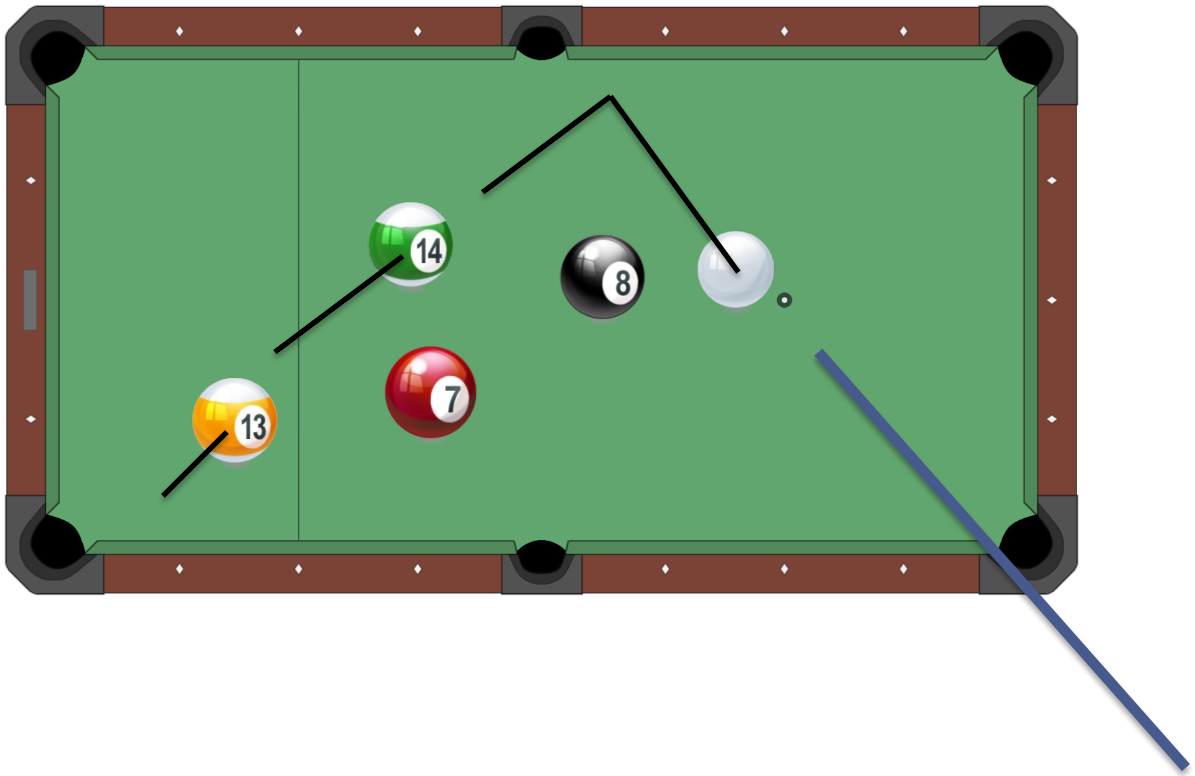
Slide from (Stoyanov & Eisner, 2012)

Error Back-Propagation



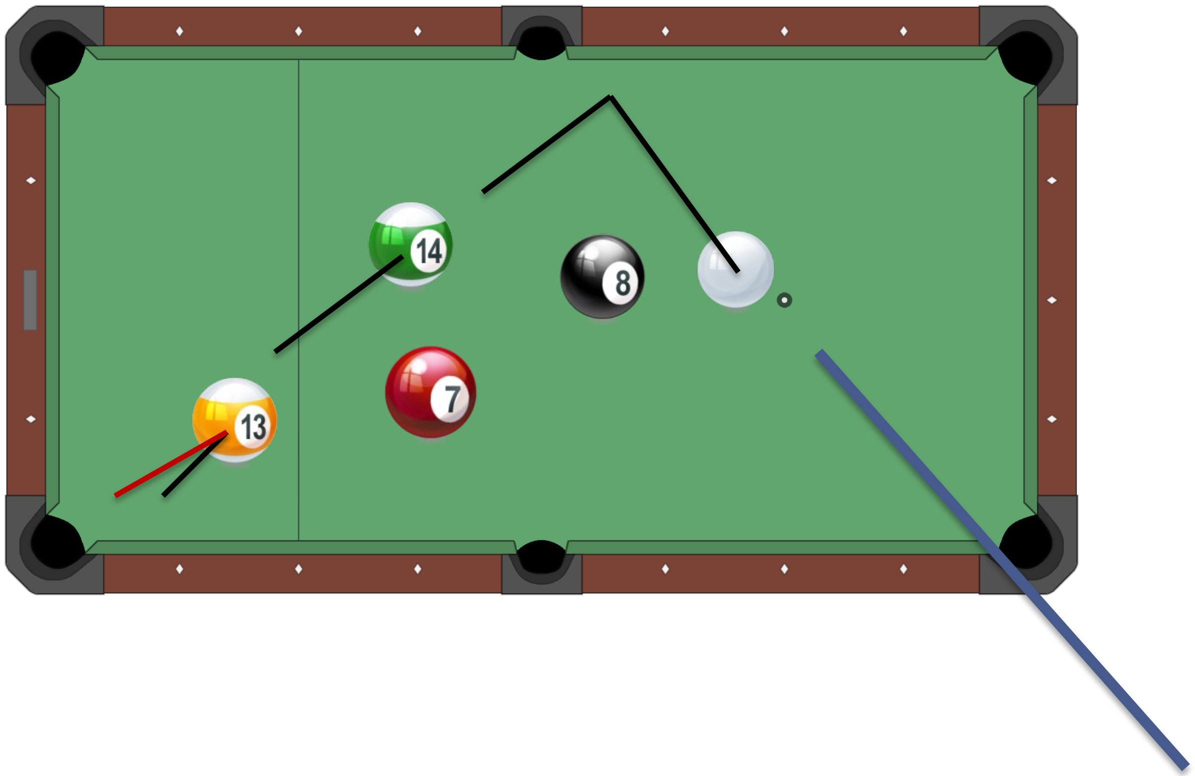
Slide from (Stoyanov & Eisner, 2012)

Error Back-Propagation



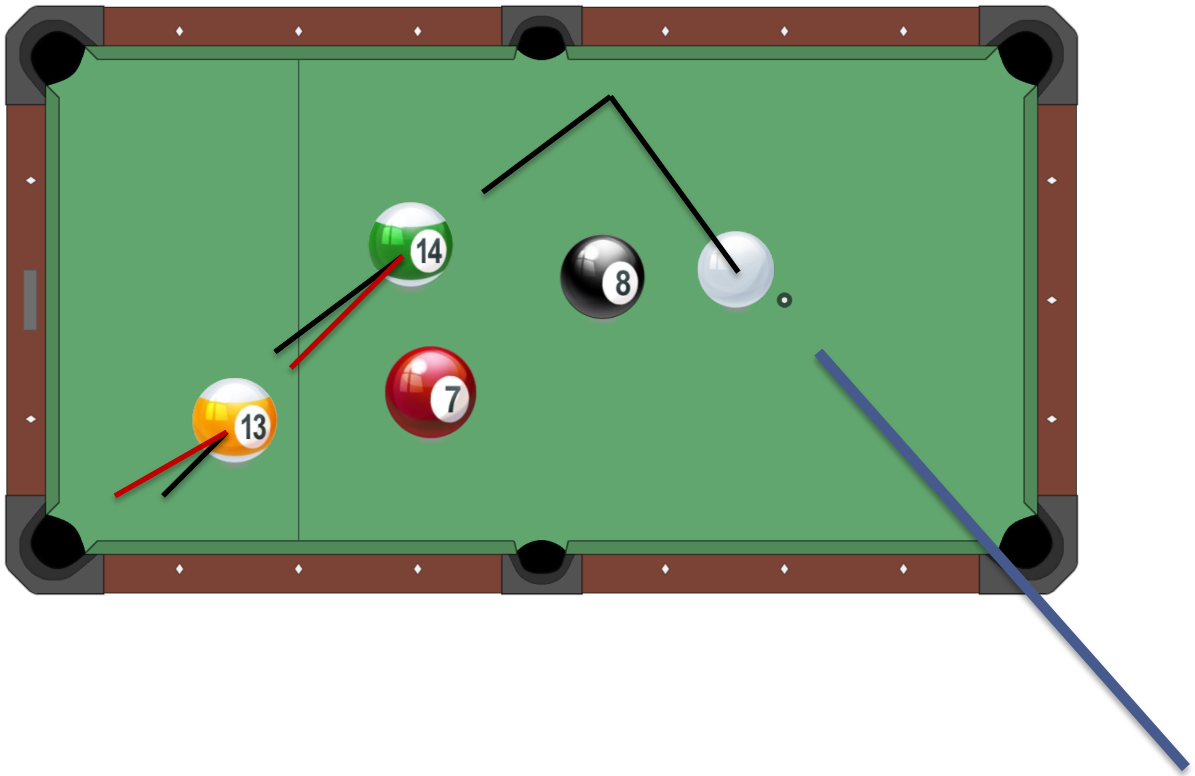
Slide from (Stoyanov & Eisner, 2012)

Error Back-Propagation



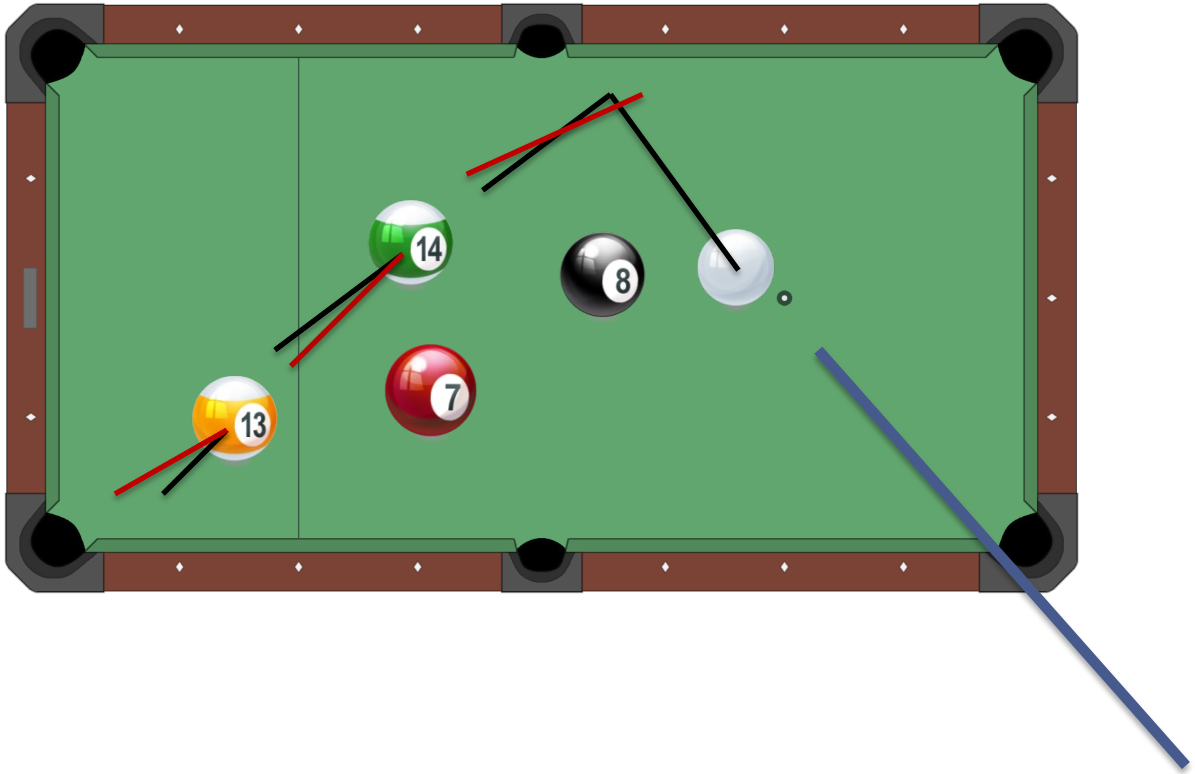
Slide from (Stoyanov & Eisner, 2012)

Error Back-Propagation



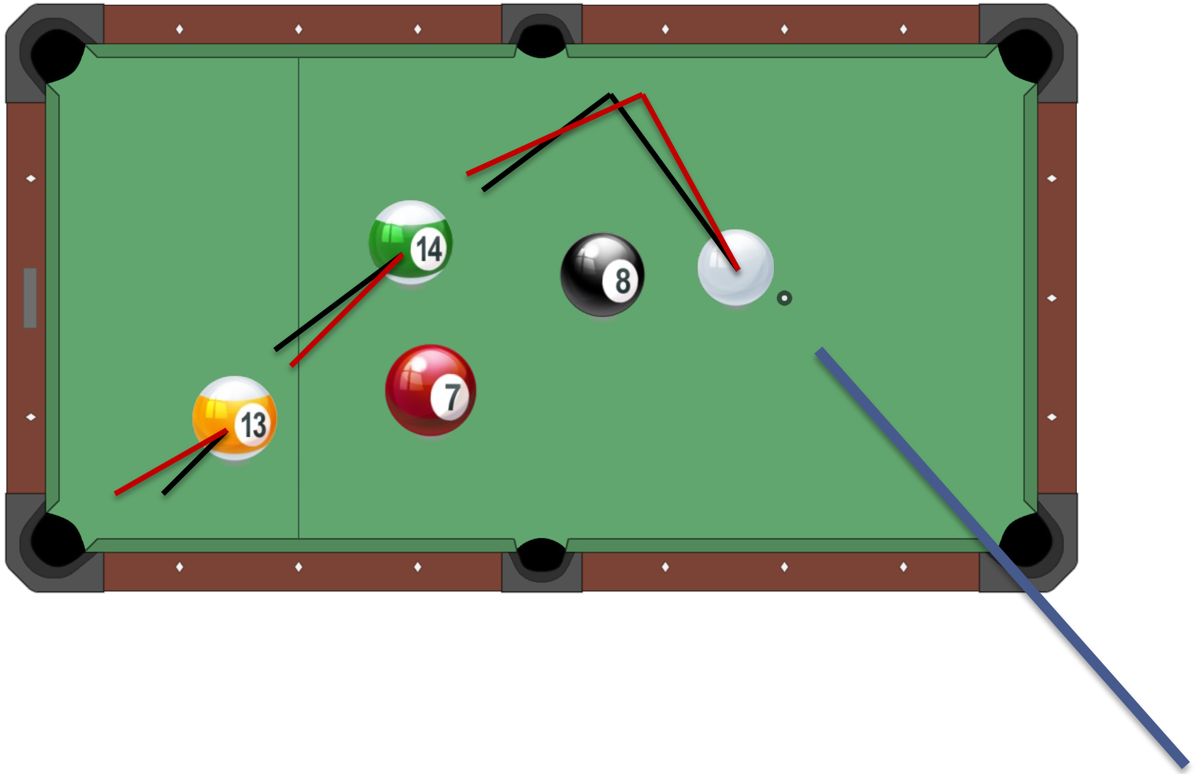
Slide from (Stoyanov & Eisner, 2012)

Error Back-Propagation



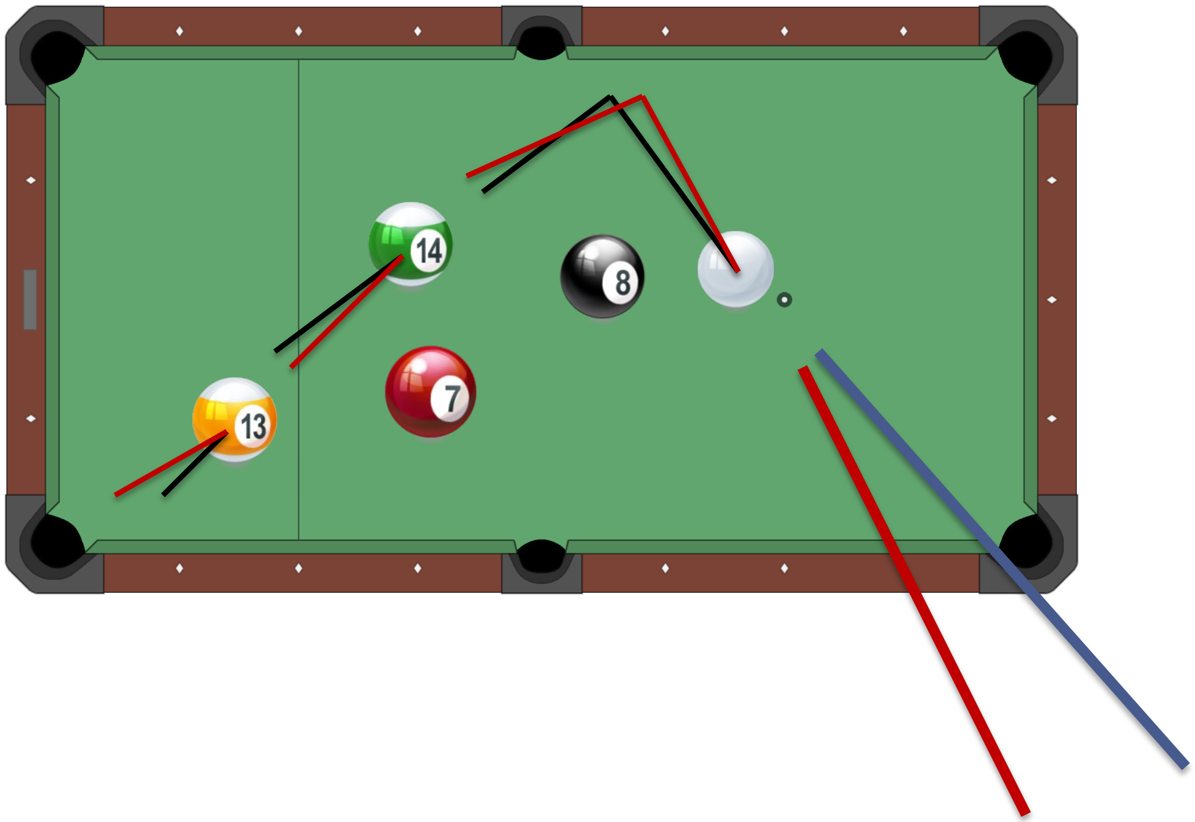
Slide from (Stoyanov & Eisner, 2012)

Error Back-Propagation



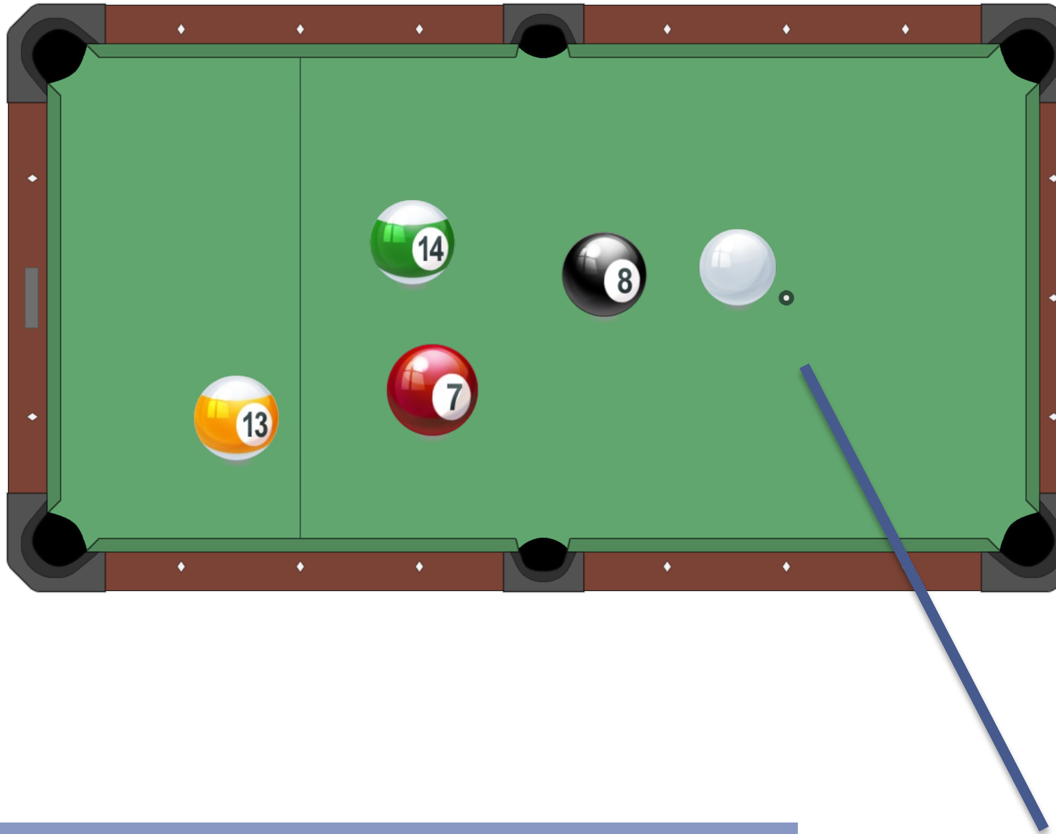
Slide from (Stoyanov & Eisner, 2012)

Error Back-Propagation



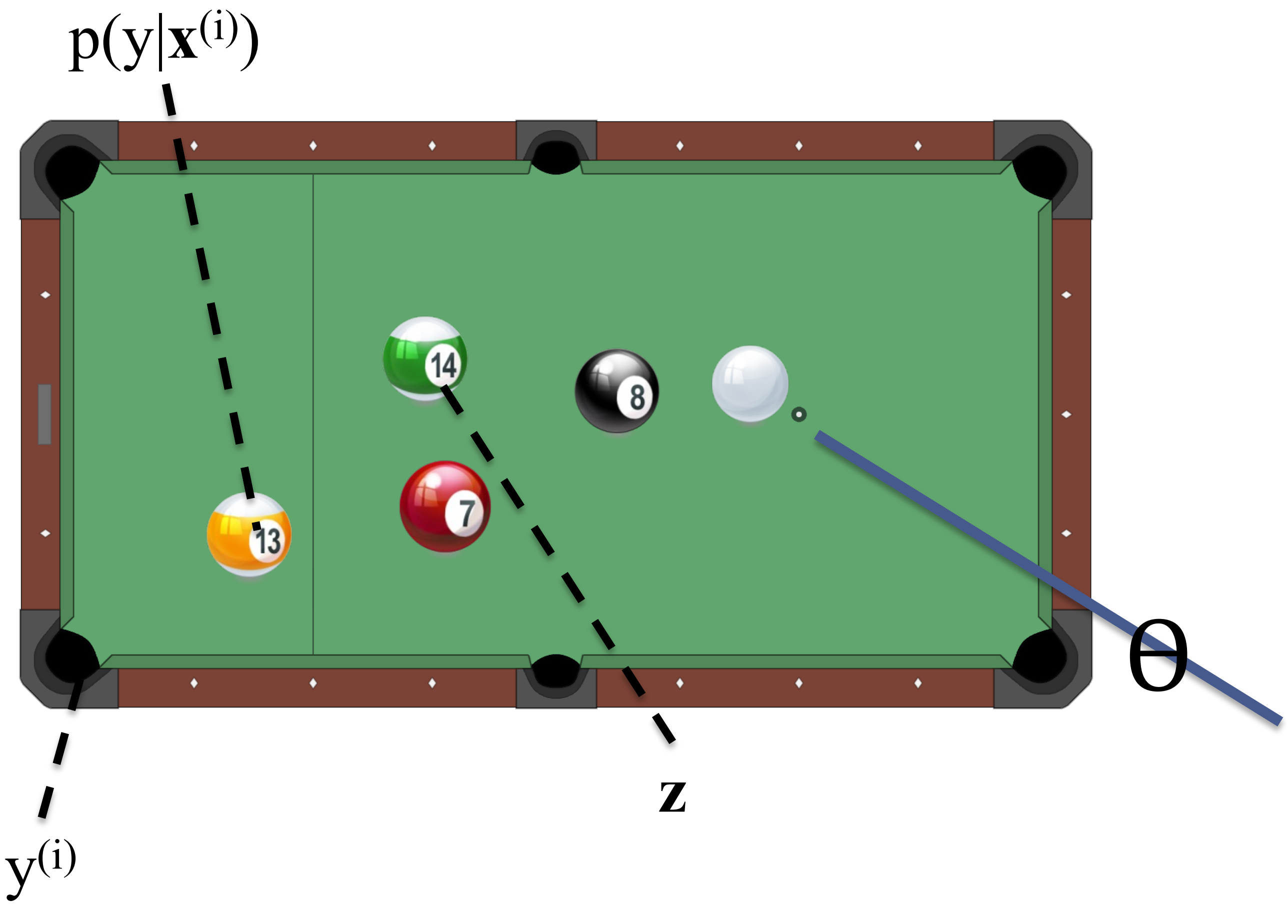
Slide from (Stoyanov & Eisner, 2012)

Error Back-Propagation



Slide from (Stoyanov & Eisner, 2012)

Error Back-Propagation



Slide from (Stoyanov & Eisner, 2012)

THE BACKPROPAGATION ALGORITHM

Backpropagation

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

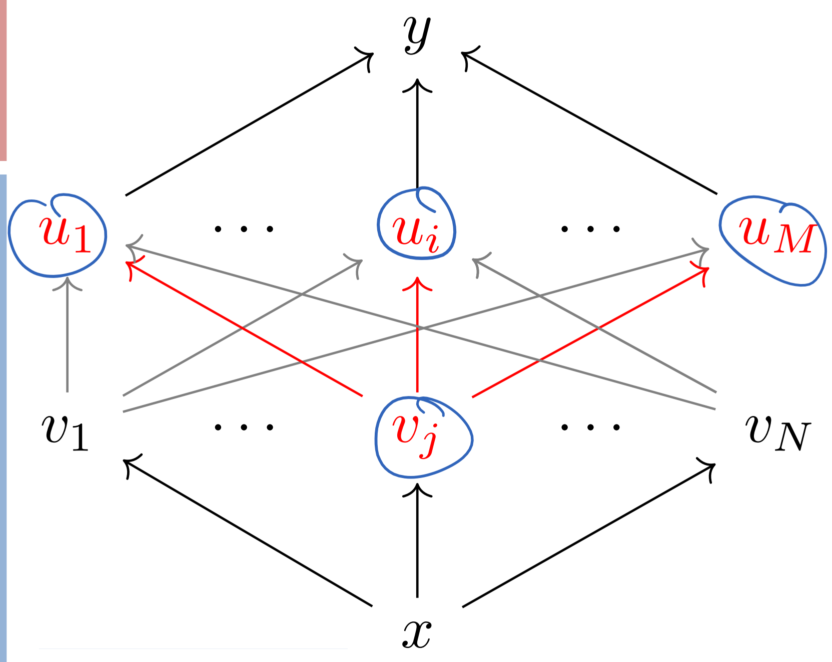
<u>Forward Computation</u>

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

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

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 $y = h(\mathbf{u}) = h(u_1, \ldots, u_M)$
   and $\mathbf{u} = g(\mathbf{v})$ 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

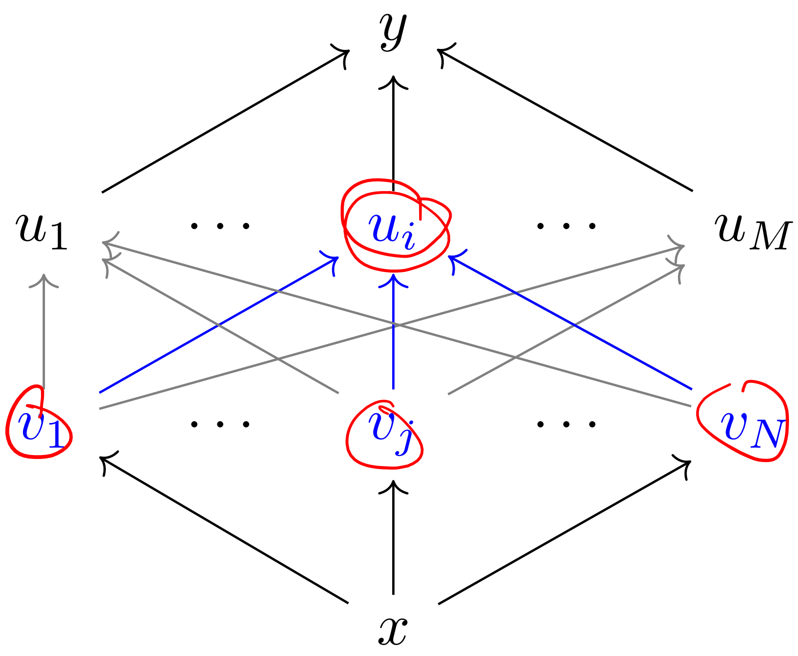
Automatic Differentiation – Reverse Mode (aka. Backpropagation)

<u>Forward Computation</u>

1. Write an **algorithm** for evaluating the function y = f(**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

Backpropagation (Version B)

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

$J = \cos(u)$

$u = u_1 + u_2$

$u_1 = \sin(t)$

$u_2 = 3t$

$t = x^2$

Backpropagation (Version B)

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.

$$g_u = 0, g_{u_1} = 0, g_{u_2} = 0, g_t = 0, g_x = 0$$

Initialize all the adjoints to zero

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

Notice that we increment the partial derivative for $\frac{dJ}{dt}$ in two places!

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)

A Recipe for

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!

1. Given training dat

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

2. Choose each of t

– Decision functio

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

– Loss function

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

$\boldsymbol{y}_i)$

opp site 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

Denominator

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	$\frac{\partial y}{\partial \mathbf{X}}$	$\frac{\partial \mathbf{y}}{\partial \mathbf{X}}$	$\frac{\partial \mathbf{Y}}{\partial \mathbf{X}}$

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

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.

Vector Derivatives

Scalar Derivatives

Suppose $x \in \mathbb{R}$
and $f : \mathbb{R} \to \mathbb{R}$

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

Vector Derivatives

Suppose $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{b} \in \mathbb{R}^m$,
$\mathbf{B} \in \mathbb{R}^{m \times n}$, $\mathbf{Q} \in \mathbb{R}^{m \times m}$
and $\mathbf{Q}$ is symmetric.

$f(\mathbf{x})$	$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$	type of f
$\mathbf{b}^T\mathbf{x}$	$\mathbf{b}$	$f : \mathbb{R}^m \to \mathbb{R}$
$\mathbf{x}^T\mathbf{b}$	$\mathbf{b}$	$f : \mathbb{R}^m \to \mathbb{R}$
$\mathbf{x}^T\mathbf{B}$	$\mathbf{B}$	$f : \mathbb{R}^m \to \mathbb{R}^n$
$\mathbf{B}^T\mathbf{x}$	$\mathbf{B}^T$	$f : \mathbb{R}^m \to \mathbb{R}^n$
$\mathbf{x}^T\mathbf{x}$	$2\mathbf{x}$	$f : \mathbb{R}^m \to \mathbb{R}$
$\mathbf{x}^T\mathbf{Q}\mathbf{x}$	$2\mathbf{Q}\mathbf{x}$	$f : \mathbb{R}^m \to \mathbb{R}$

Vector Derivatives

Scalar Derivatives

Suppose $\mathbf{x} \in \mathbb{R}^m$ and we have constants $a \in \mathbb{R}$, $b \in \mathbb{R}$

$f(x)$	$\frac{\partial f(x)}{\partial x}$
$g(x) + h(x)$	$\frac{\partial g(x)}{\partial x} + \frac{\partial h(x)}{\partial x}$
$ag(x)$	$a\frac{\partial g(x)}{\partial x}$
$g(x)b$	$\frac{\partial g(x)}{\partial x}b$

Vector Derivatives

Suppose $\mathbf{x} \in \mathbb{R}^m$ and we have constants $a \in \mathbb{R}$, $\mathbf{b} \in \mathbb{R}^n$

$f(\mathbf{x})$	$\frac{\partial f(\mathbf{x})}{\partial \mathbf{x}}$
$g(\mathbf{x}) + h(\mathbf{x})$	$\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}} + \frac{\partial h(\mathbf{x})}{\partial \mathbf{x}}$
$ag(\mathbf{x})$	$a\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}$
$g(\mathbf{x})\mathbf{b}$	$\frac{\partial g(\mathbf{x})}{\partial \mathbf{x}}\mathbf{b}^T$

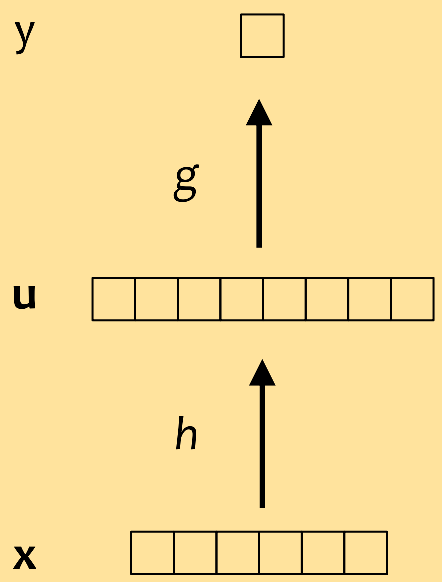
Matrix Calculus

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:

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. $\frac{\partial y}{\partial \mathbf{u}} \frac{\partial \mathbf{u}}{\partial \mathbf{x}}$

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

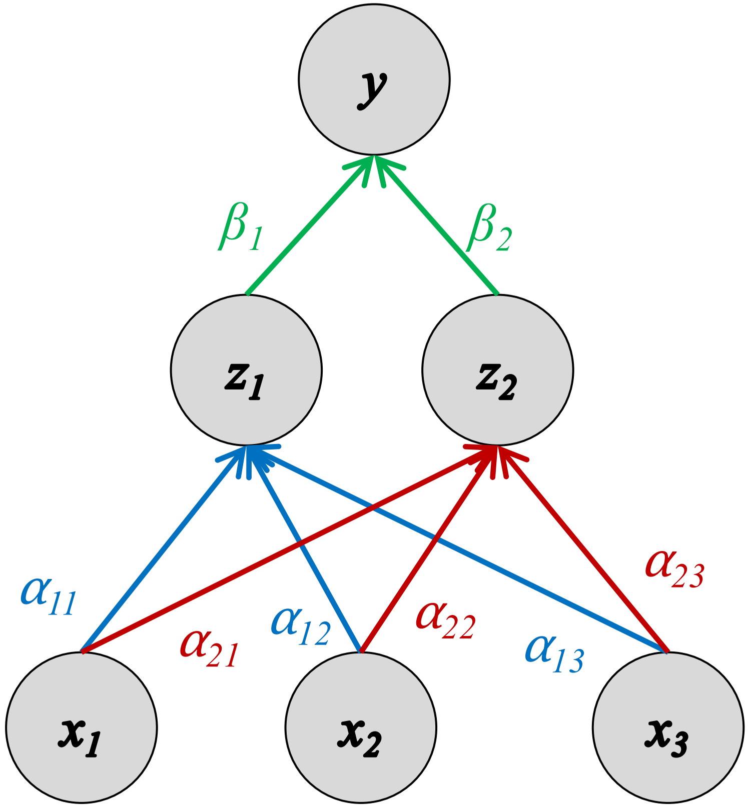
F. None of the above

DRAWING A NEURAL NETWORK

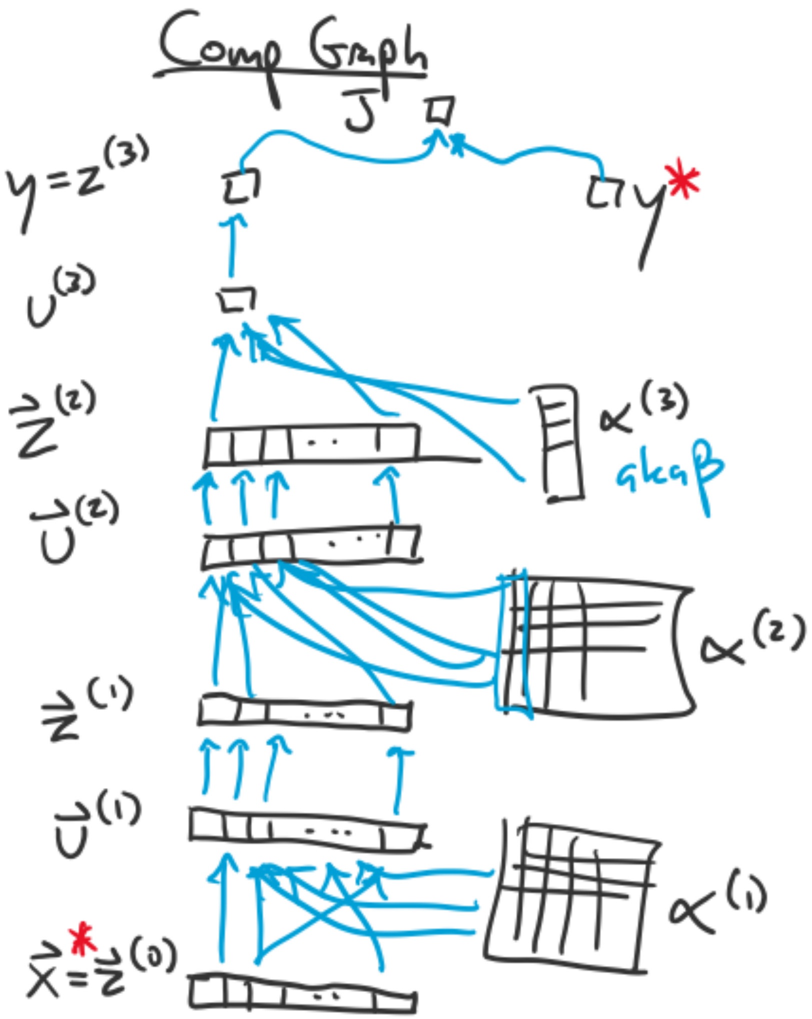
Ways of Drawing Neural Networks

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



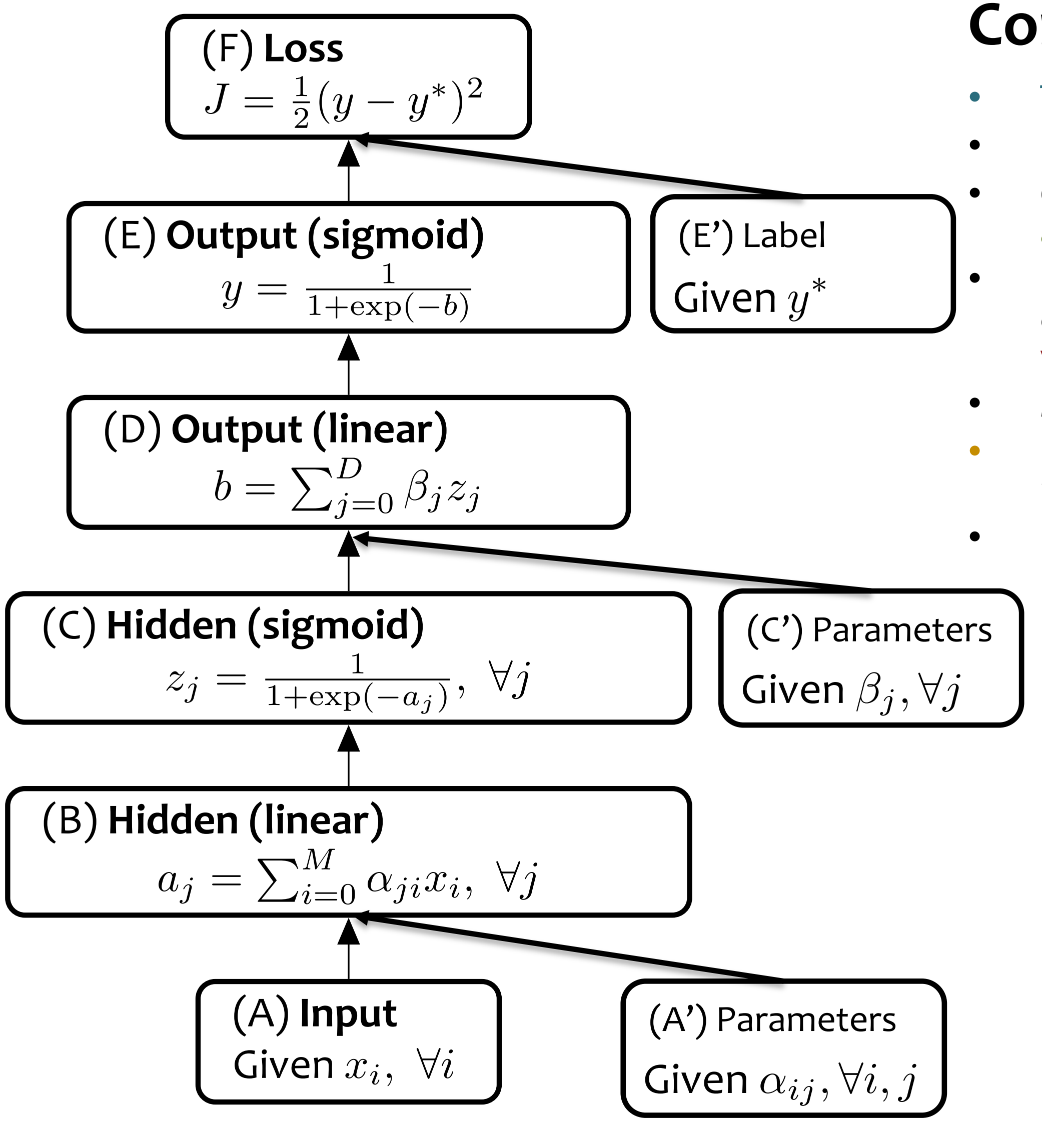
Ways of Drawing Neural Networks



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.