

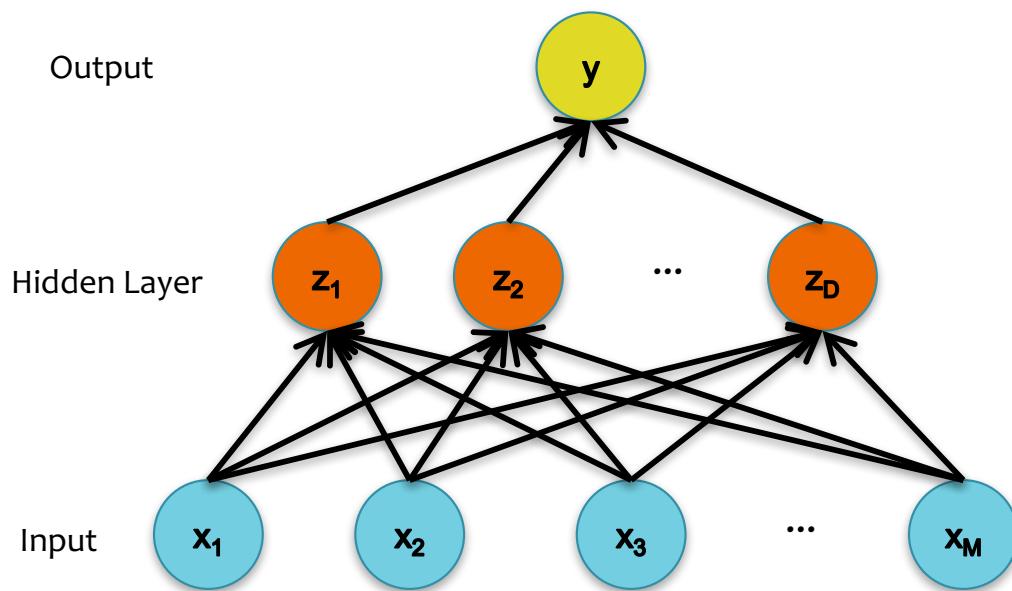
Neural Network Architectures

Even for a basic Neural Network, there are many design decisions to make:

1. # of hidden layers (depth)
2. # of units per hidden layer (width)
3. Type of activation function (nonlinearity)
4. Form of objective function

Activation Functions

Neural Network with sigmoid activation functions



(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)

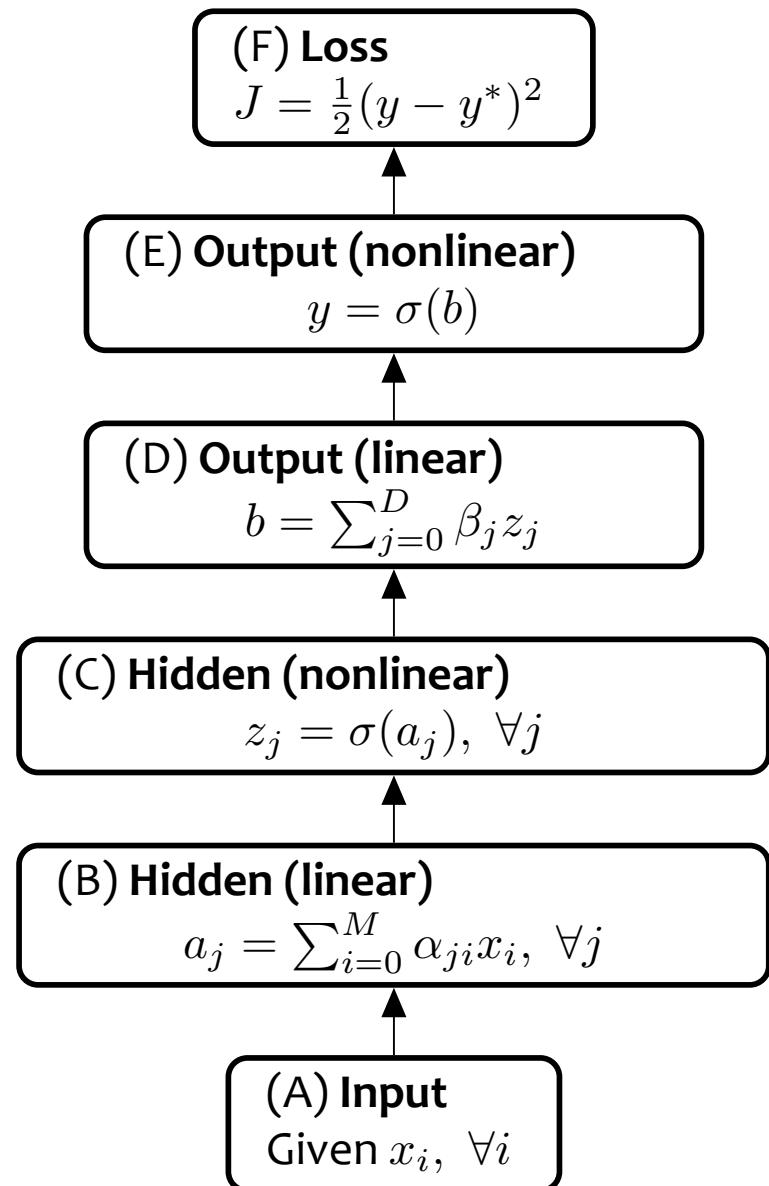
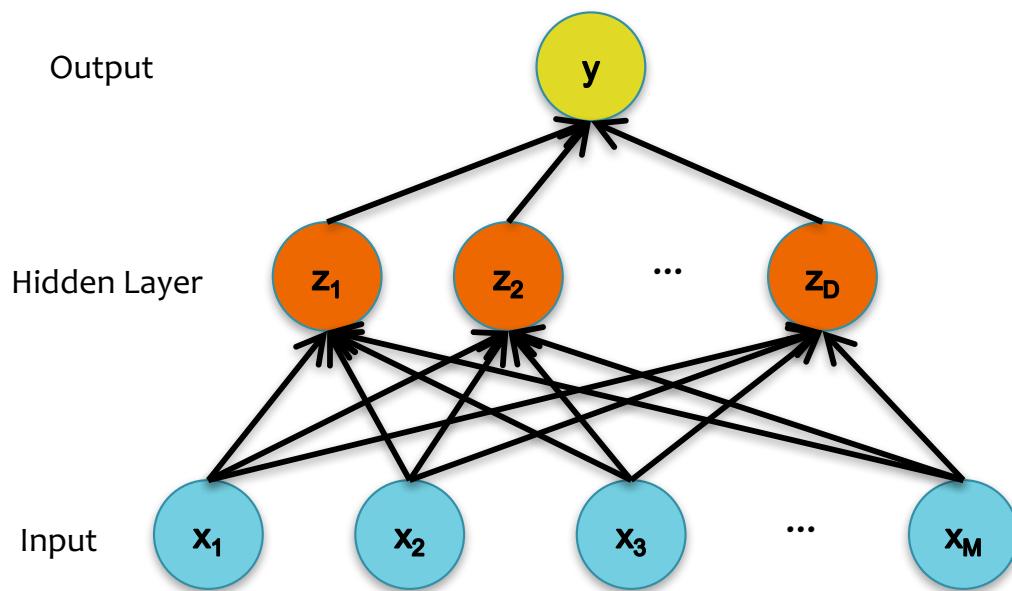
$$a_j = \sum_{i=0}^M \alpha_{ji} x_i, \forall j$$

(A) Input

Given $x_i, \forall i$

Activation Functions

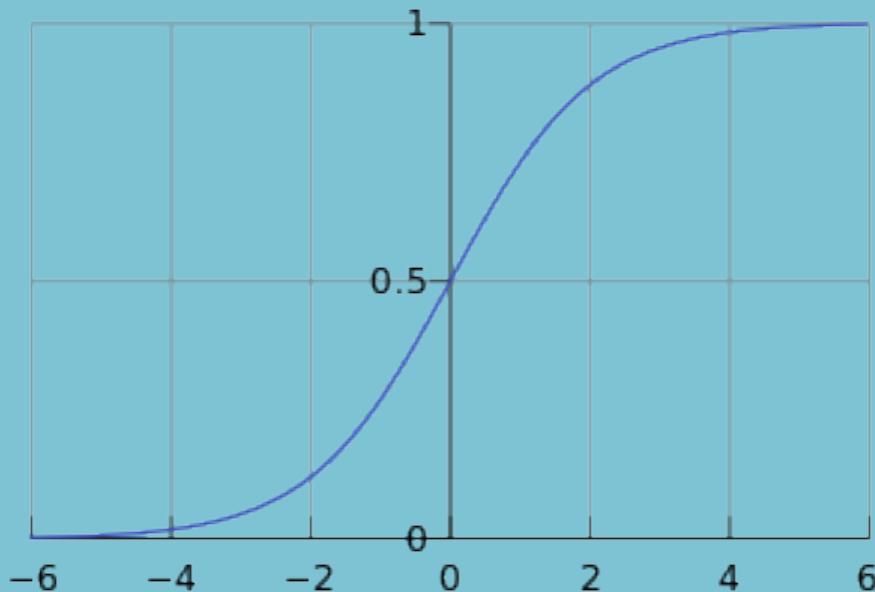
Neural Network with arbitrary nonlinear activation functions



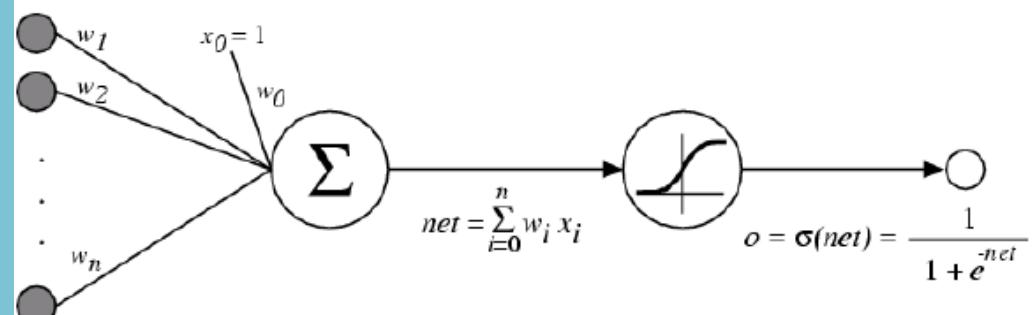
Activation Functions

Sigmoid / Logistic Function

$$\text{logistic}(u) \equiv \frac{1}{1 + e^{-u}}$$

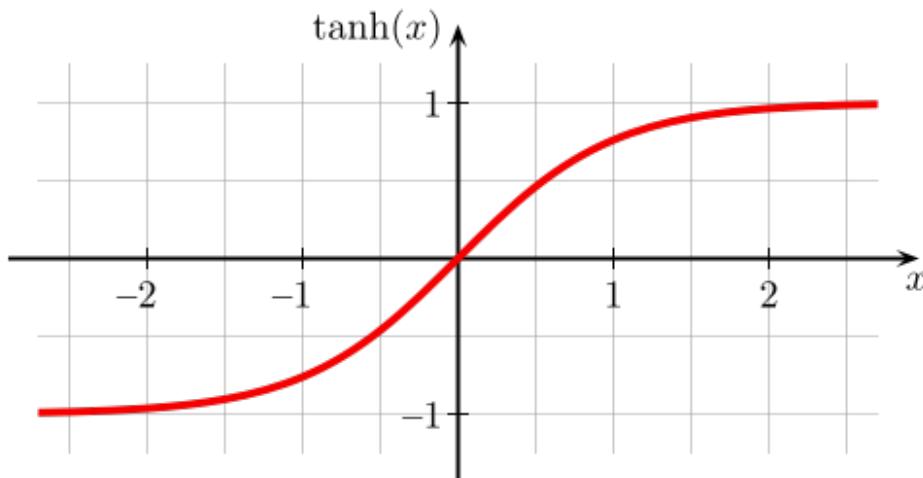


So far, we've assumed that the activation function (nonlinearity) is always the sigmoid function...



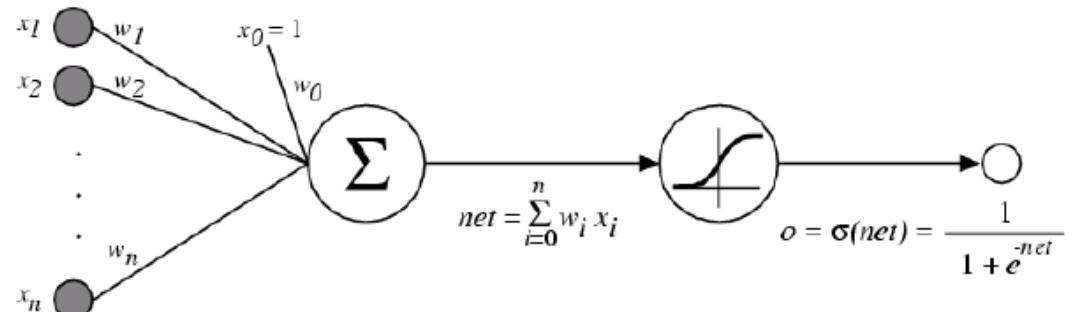
Activation Functions

- A new change: modifying the nonlinearity
 - The logistic is not widely used in modern ANNs



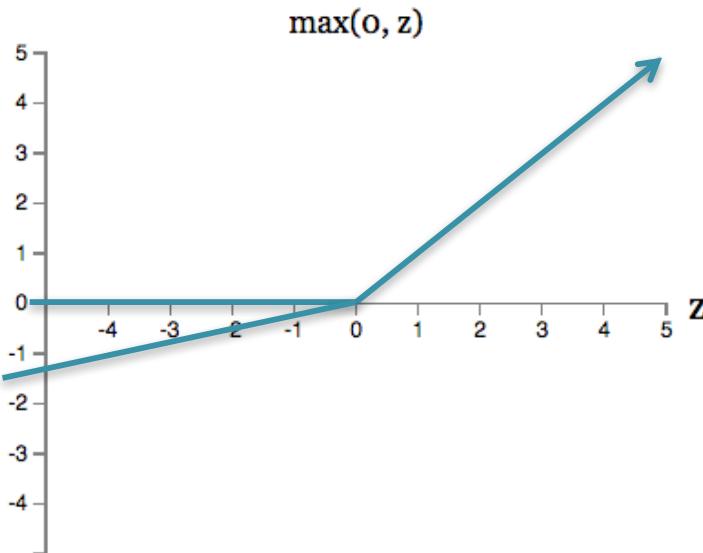
Alternate 1:
 \tanh

Like logistic function but
shifted to range $[-1, +1]$



Activation Functions

- A new change: modifying the nonlinearity
 - reLU often used in vision tasks

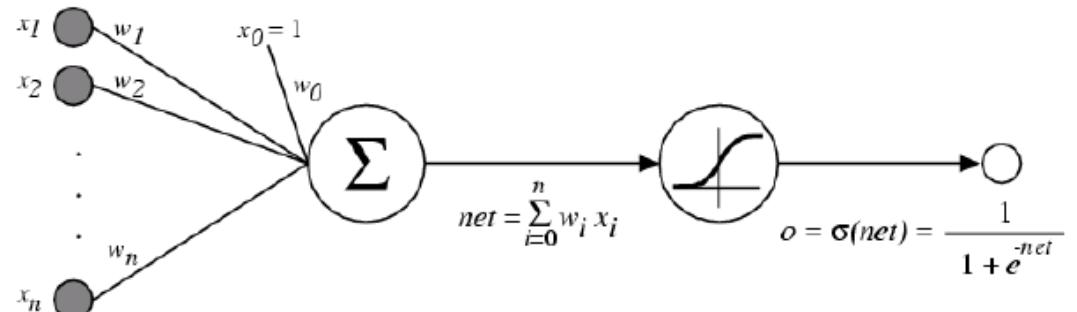


Alternate 2: rectified linear unit

Linear with a cutoff at zero

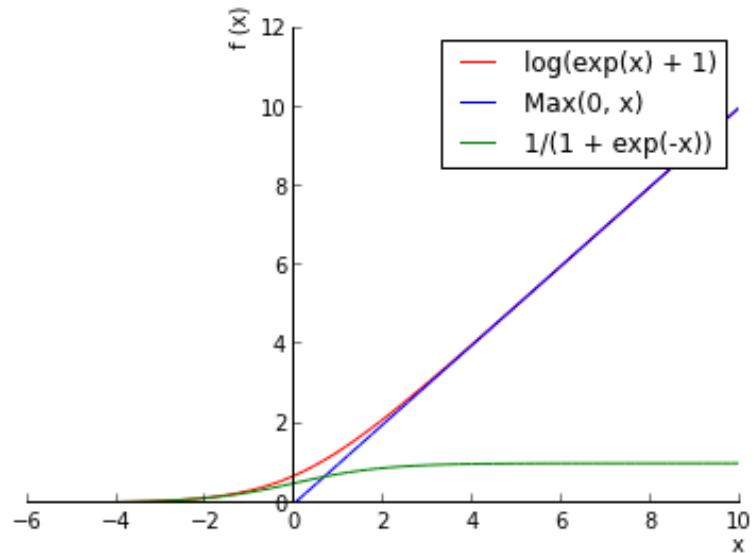
(Implementation: clip the gradient when you pass zero)

$$\max(0, w \cdot x + b).$$



Activation Functions

- A new change: modifying the nonlinearity
 - reLU often used in vision tasks



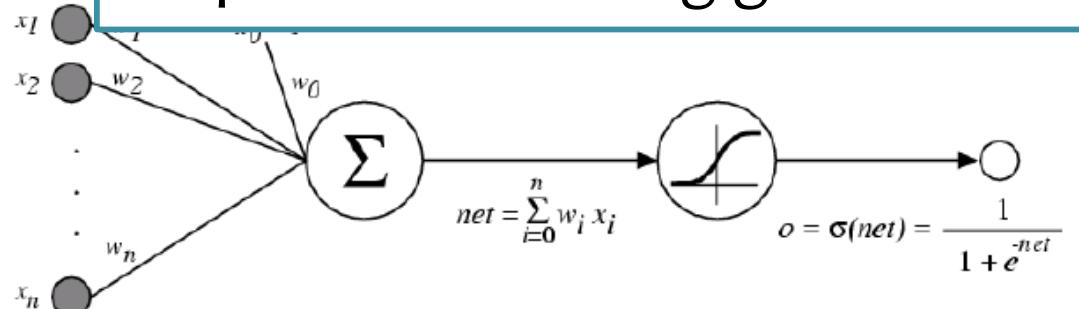
Alternate 2: rectified linear unit

Soft version: $\log(\exp(x)+1)$

Doesn't saturate (at one end)

Sparsifies outputs

Helps with vanishing gradient



Objective Functions for NNs

- Regression:
 - Use the same objective as Linear Regression
 - Quadratic loss (i.e. mean squared error)
- Classification:
 - Use the same objective as Logistic Regression
 - Cross-entropy (i.e. negative log likelihood)
 - This requires probabilities, so we add an additional “softmax” layer at the end of our network

Forward

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

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

Backward

$$\frac{dJ}{dy} = y - y^*$$

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

Cross-entropy vs. Quadratic loss

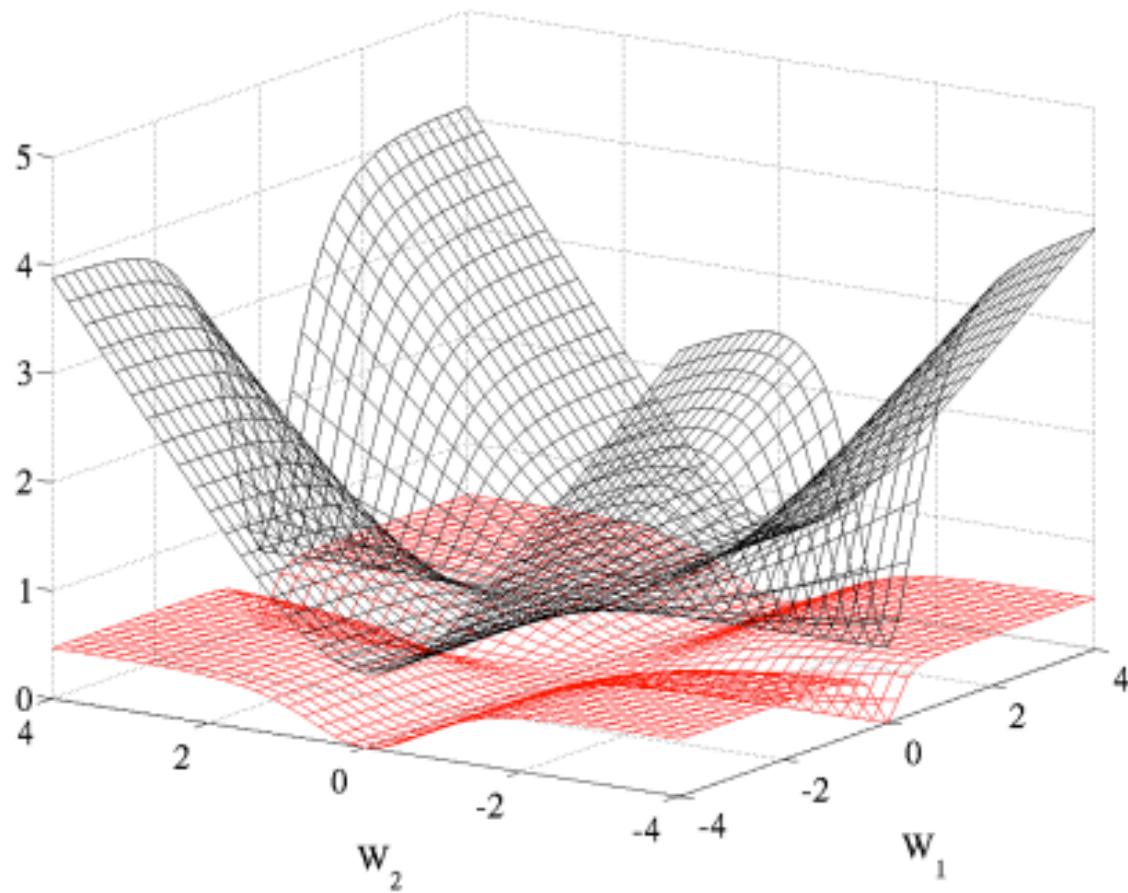


Figure 5: Cross entropy (black, surface on top) and quadratic (red, bottom surface) cost as a function of two weights (one at each layer) of a network with two layers, W_1 respectively on the first layer and W_2 on the second, output layer.

Background

A Recipe for Machine Learning

1. Given training data:

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

2. Choose each of these:

- Decision function

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

- Loss function

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

3. Define goal:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

4. Train with SGD:

(take small steps
opposite the gradient)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

Objective Functions

Matching Quiz: Suppose you are given a neural net with a single output, y , and one hidden layer.

1) Minimizing sum of squared errors...

2) Minimizing sum of squared errors plus squared Euclidean norm of weights...

3) Minimizing cross-entropy...

4) Minimizing hinge loss...

... gives...

5) ... MLE estimates of weights assuming target follows a Bernoulli with parameter given by the output value

6) ... MAP estimates of weights assuming weight priors are zero mean Gaussian

7) ... estimates with a large margin on the training data

8) ... MLE estimates of weights assuming zero mean Gaussian noise on the output value

- A. 1=5, 2=7, 3=6, 4=8
- B. 1=5, 2=7, 3=8, 4=6
- C. 1=7, 2=5, 3=5, 4=7
- D. 1=7, 2=5, 3=6, 4=8

- E. 1=8, 2=6, 3=5, 4=7
- F. 1=8, 2=6, 3=8, 4=6

BACKPROPAGATION

Background

A Recipe for Machine Learning

1. Given training data:

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

2. Choose each of these:

- Decision function

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

- Loss function

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

3. Define goal:

$$\boldsymbol{\theta}^* = \arg \min_{\boldsymbol{\theta}} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

4. Train with SGD:

(take small steps
opposite the gradient)

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\mathbf{x}_i), \mathbf{y}_i)$$

- **Question 1:**
When can we compute the gradients of the parameters of an arbitrary neural network?
- **Question 2:**
When can we make the gradient computation efficient?

Training

Approaches to Differentiation

1. Finite Difference Method
 - Pro: Great for testing implementations of backpropagation
 - Con: Slow for high dimensional inputs / outputs
 - Required: Ability to call the function $f(\mathbf{x})$ on any input \mathbf{x}
2. Symbolic Differentiation
 - Note: The method you learned in high-school
 - Note: Used by Mathematica / Wolfram Alpha / Maple
 - Pro: Yields easily interpretable derivatives
 - Con: Leads to exponential computation time if not carefully implemented
 - Required: Mathematical expression that defines $f(\mathbf{x})$
3. Automatic Differentiation - Reverse Mode
 - Note: Called Backpropagation when applied to Neural Nets
 - Pro: Computes partial derivatives of one output $f(\mathbf{x})_i$ with respect to all inputs x_j in time proportional to computation of $f(\mathbf{x})$
 - Con: Slow for high dimensional outputs (e.g. vector-valued functions)
 - Required: Algorithm for computing $f(\mathbf{x})$
4. Automatic Differentiation - Forward Mode
 - Note: Easy to implement. Uses dual numbers.
 - Pro: Computes partial derivatives of all outputs $f(\mathbf{x})_i$ with respect to one input x_j in time proportional to computation of $f(\mathbf{x})$
 - Con: Slow for high dimensional inputs (e.g. vector-valued \mathbf{x})
 - Required: Algorithm for computing $f(\mathbf{x})$

Given $f : \mathbb{R}^A \rightarrow \mathbb{R}^B, f(\mathbf{x})$
Compute $\frac{\partial f(\mathbf{x})_i}{\partial x_j} \forall i, j$

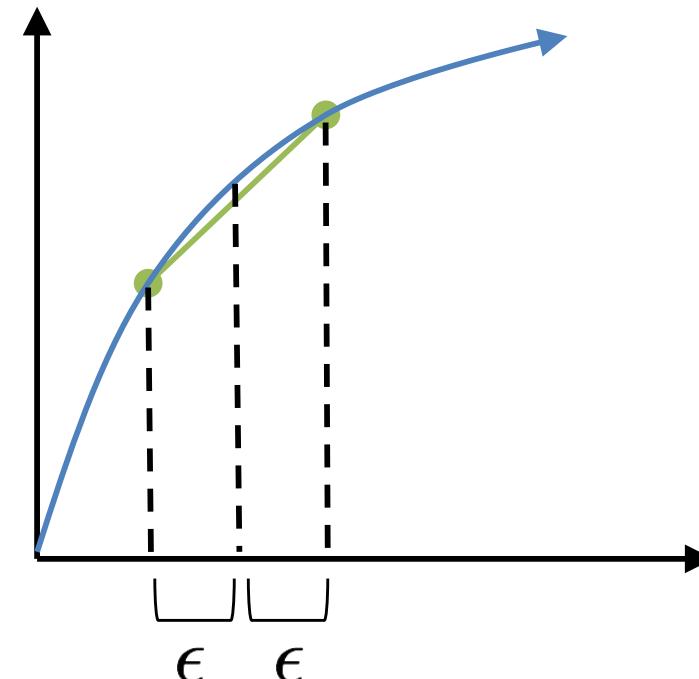
The centered finite difference approximation is:

$$\frac{\partial}{\partial \theta_i} J(\theta) \approx \frac{(J(\theta + \epsilon \cdot d_i) - J(\theta - \epsilon \cdot d_i))}{2\epsilon} \quad (1)$$

where d_i is a 1-hot vector consisting of all zeros except for the i th entry of d_i , which has value 1.

Notes:

- Suffers from issues of floating point precision, in practice
- Typically only appropriate to use on small examples with an appropriately chosen epsilon



Calculus Quiz #1:

Suppose $x = 2$ and $z = 3$, what are dy/dx and dy/dz for the function below?

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

Calculus Quiz #2:

A neural network with 2 hidden layers can be written as:

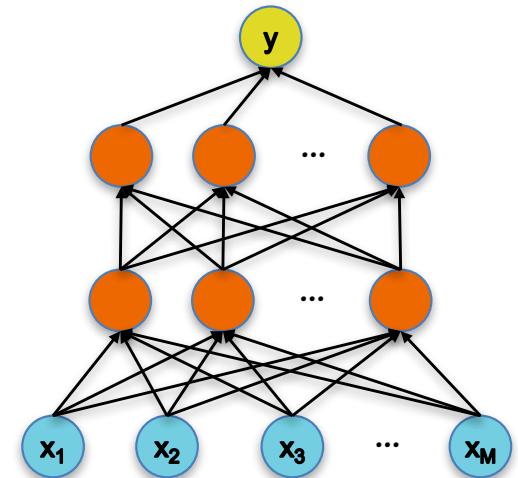
$$y = \sigma(\beta^T \sigma((\alpha^{(2)})^T \sigma((\alpha^{(1)})^T \mathbf{x})))$$

where $y \in \mathbb{R}$, $\mathbf{x} \in \mathbb{R}^{D^{(0)}}$, $\beta \in \mathbb{R}^{D^{(2)}}$ and $\alpha^{(i)}$ is a $D^{(i)} \times D^{(i-1)}$ matrix. Nonlinear functions are applied elementwise:

$$\sigma(\mathbf{a}) = [\sigma(a_1), \dots, \sigma(a_K)]^T$$

Let σ be sigmoid: $\sigma(a) = \frac{1}{1+exp-a}$

What is $\frac{\partial y}{\partial \beta_j}$ and $\frac{\partial y}{\partial \alpha_j^{(i)}}$ for all i, j .



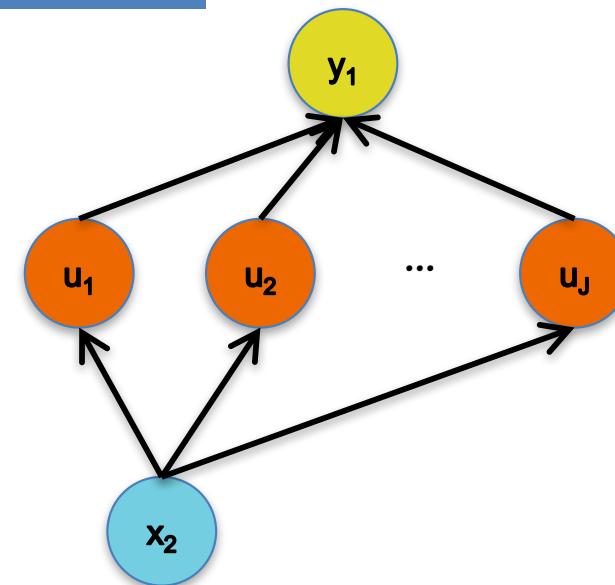
Whiteboard

– Chain Rule of Calculus

Given: $y = g(u)$ and $u = h(x)$.

Chain Rule:

$$\frac{dy_i}{dx_k} = \sum_{j=1}^J \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$



Training

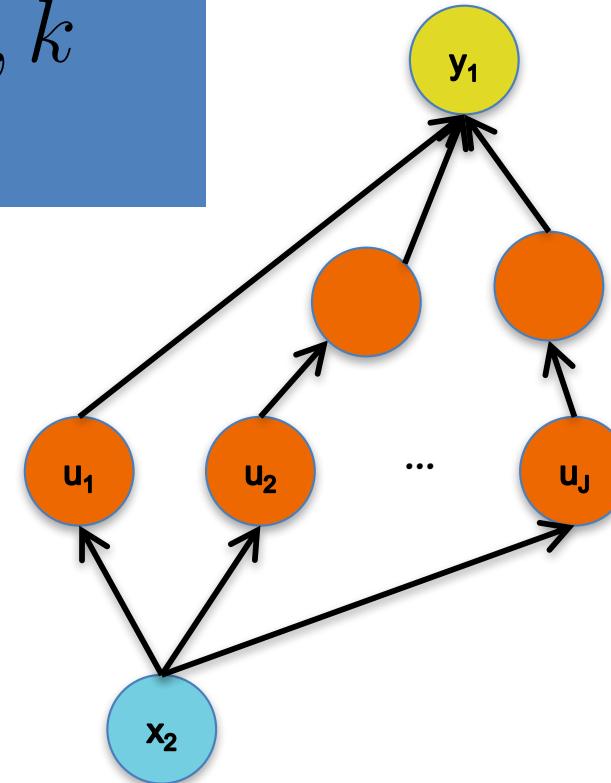
Chain Rule

Given: $y = g(u)$ and $u = h(x)$.

Chain Rule:

$$\frac{dy_i}{dx_k} = \sum_{j=1}^J \frac{dy_i}{du_j} \frac{du_j}{dx_k}, \quad \forall i, k$$

Backpropagation
is just repeated
application of the
chain rule from
Calculus 101.



Whiteboard

- Example: Backpropagation for Calculus Quiz #1

Calculus Quiz #1:

Suppose $x = 2$ and $z = 3$, what are dy/dx and dy/dz for the function below?

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

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, \dots, v_N
 - a. Compute $u_i = g_i(v_1, \dots, v_N)$
 - b. Store the result at the node

Backward Computation

1. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
2. Visit each node in **reverse topological order**.
For variable $u_i = g_i(v_1, \dots, v_N)$
 - 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

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

$$J = \cos(u)$$

$$u = u_1 + u_2$$

$$u_1 = \sin(t)$$

$$u_2 = 3t$$

$$t = x^2$$

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

$$J = \cos(u)$$

$$u = u_1 + u_2$$

$$u_1 = \sin(t)$$

$$u_2 = 3t$$

$$t = x^2$$

Backward

$$\frac{dJ}{du} += -\sin(u)$$

$$\frac{dJ}{du_1} += \frac{dJ}{du} \frac{du}{du_1}, \quad \frac{du}{du_1} = 1$$

$$\frac{dJ}{dt} += \frac{dJ}{du_1} \frac{du_1}{dt}, \quad \frac{du_1}{dt} = \cos(t)$$

$$\frac{dJ}{dt} += \frac{dJ}{du_2} \frac{du_2}{dt}, \quad \frac{du_2}{dt} = 3$$

$$\frac{dJ}{dx} += \frac{dJ}{dt} \frac{dt}{dx}, \quad \frac{dt}{dx} = 2x$$

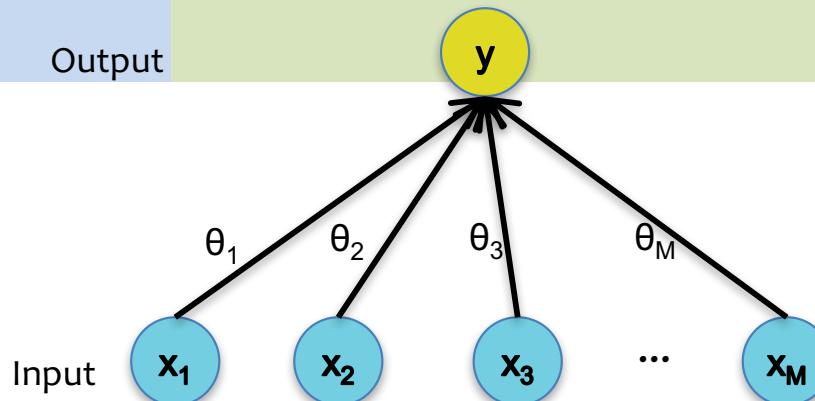
Whiteboard

- SGD for Neural Network
- Example: Backpropagation for Neural Network

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

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

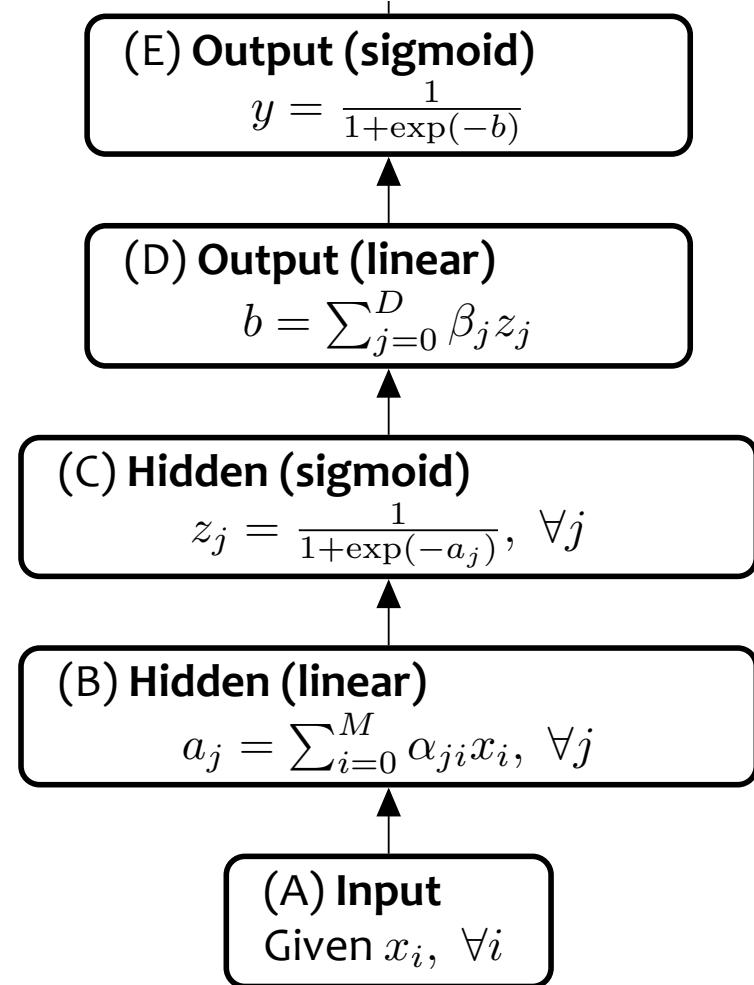
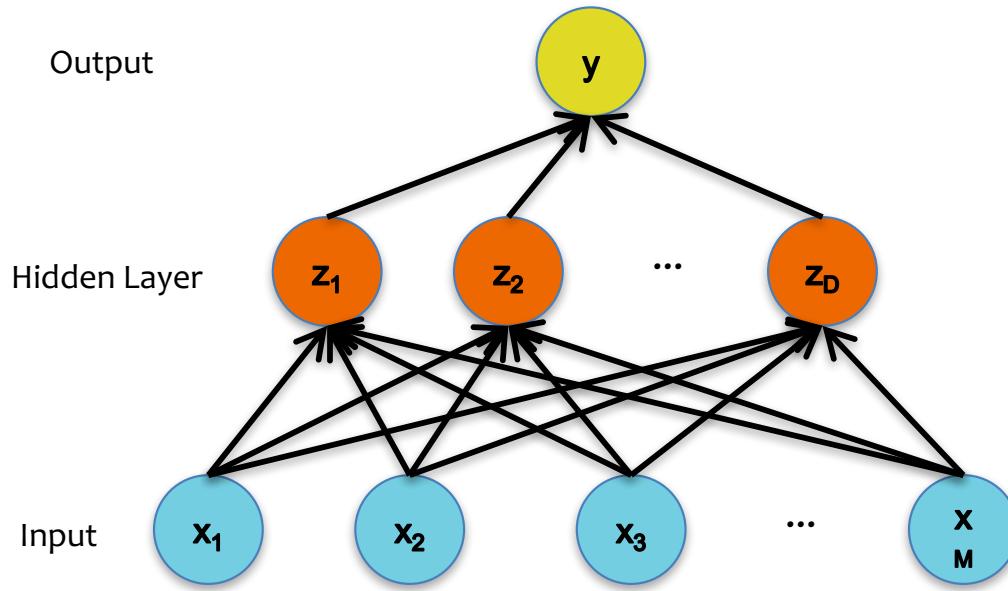
$$\frac{dJ}{da} = \frac{dJ}{dy} \frac{dy}{da}, \quad \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2}$$

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

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

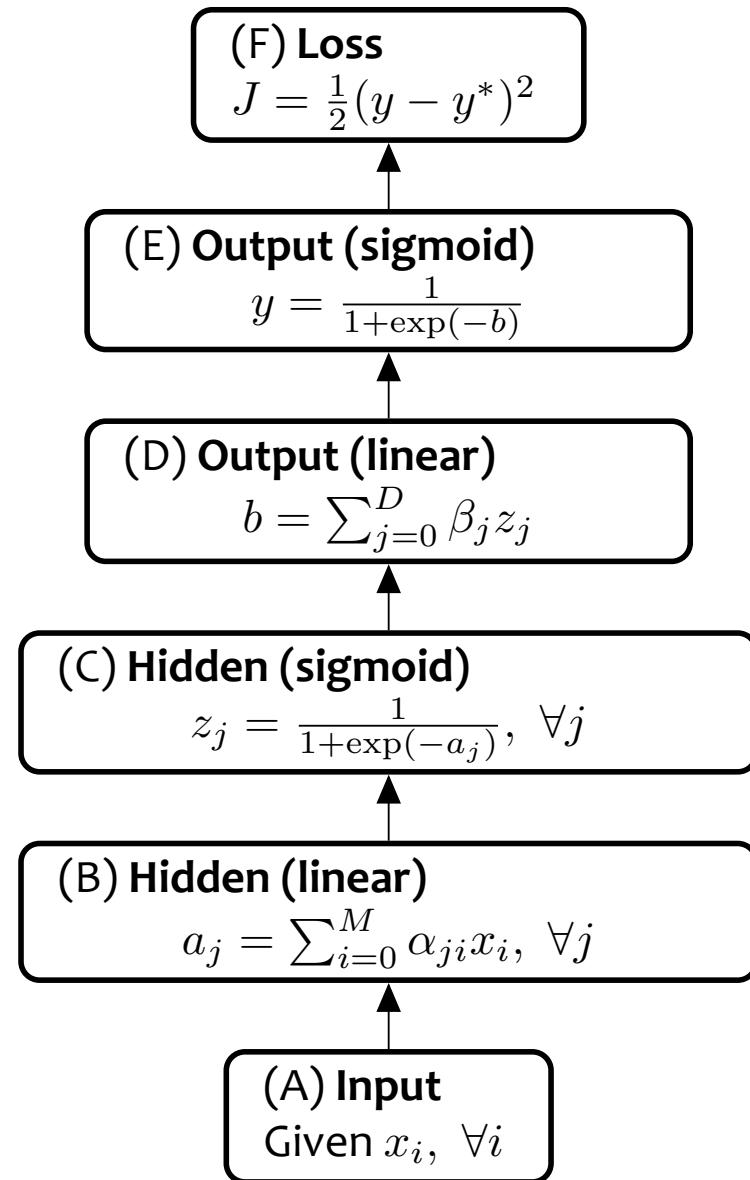
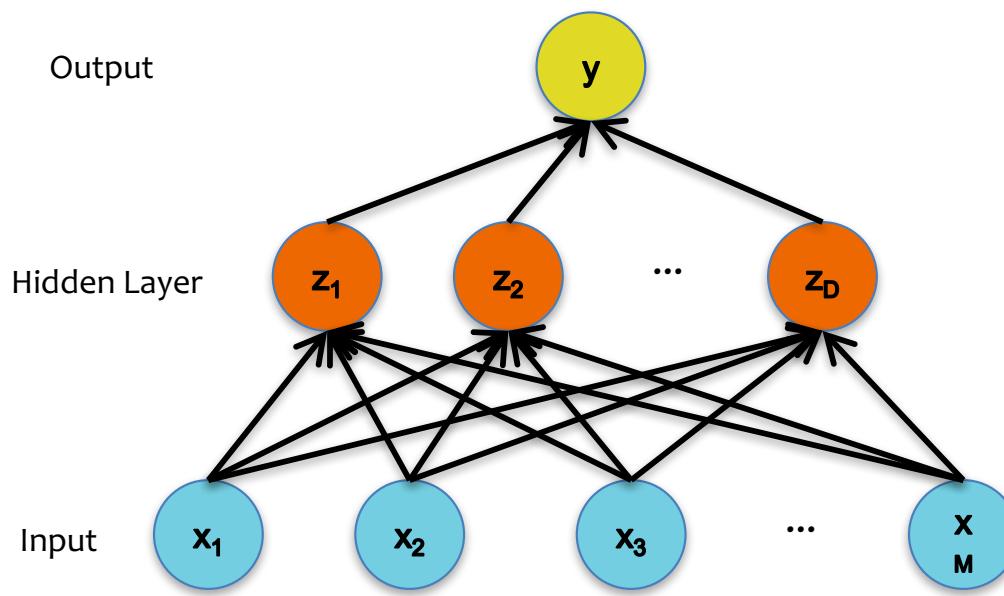
Training

Backpropagation



Training

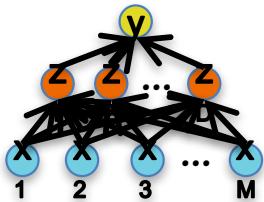
Backpropagation



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

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

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

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

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

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

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

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

Backpropagation (Auto.Dif. - Reverse Mode)

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**.
 - a. Compute the corresponding variable’s value
 - b. Store the result at the node

Backward Computation

3. **Initialize** all partial derivatives dy/du_j to 0 and $dy/dy = 1$.
4. Visit each node in **reverse topological order**.
For variable $u_i = g_i(v_1, \dots, v_N)$
 - 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

Case 2:

Module 5

Forward

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

Backward

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

Module 4

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

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

Module 3

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

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

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

Module 2

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

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

Module 1

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

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

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

Background

1. Given training data

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

2. Choose each of the

- Decision function

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

- Loss function

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

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!

(opposite the gradient)

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

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