CS 4803 / 7643: Deep Learning

Topics:

- (Finish) Analytical Gradients
- Automatic Differentiation
  - Computational Graphs
  - Forward mode vs Reverse mode AD

Dhruv Batra
Georgia Tech
Administrativia

- HW1 Reminder
  - Due: 09/26, 11:55pm

- Fuller schedule + future reading posted
  - [https://www.cc.gatech.edu/classes/AY2020/cs7643_fall/](https://www.cc.gatech.edu/classes/AY2020/cs7643_fall/)
  - Caveat: subject to change; please don’t make irreversible decisions based on this.
Recap from last time
Strategy: **Follow the slope**

\[
\min_{\tilde{w}} \quad L(\tilde{w}, D) = \frac{1}{N} \sum_{i} L_i(\tilde{w})
\]
Gradient Descent

```
# Vanilla Gradient Descent

while True:
    weights_grad = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * weights_grad  # perform parameter update
```

\[ w^{(0)} = \text{init} \]

for \( t = 1 \ldots \text{tired} \)

\[ w^{(t+1)} = w^t - \eta \nabla \]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Stochastic Gradient Descent (SGD)

\[ L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W) + \lambda R(W) \]

\[ \nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W) + \lambda \nabla_W R(W) \]

Full sum expensive when N is large!

Approximate sum using a minibatch of examples 32 / 64 / 128 common

# Vanilla Minibatch Gradient Descent

```python
while True:
    data_batch = sample_training_data(data, 256)  # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad  # perform parameter update
```
How do we compute gradients?

- Analytic or “Manual” Differentiation
- Symbolic Differentiation
- Numerical Differentiation ❌
- Automatic Differentiation
  - Forward mode AD
  - Reverse mode AD
    - aka “backprop”
\[ l_1 = x \]
\[ l_{n+1} = 4l_n(1 - l_n) \]
\[ f(x) = l_4 = 64x(1 - x)(1 - 2x)^2(1 - 8x + 8x^2)^2 \]

\[ f'(x) = \frac{256x(1 - x)(1 - 8x + 8x^2)^2 - 64x(1 - 2x)^2(1 - 8x + 8x^2)^2}{(1 - 2x)(1 - 8x + 8x^2)^2} \]

**Coding**

\[ f(x): \]
\[ v = x \]
\[ \text{for } i = 1 \text{ to } 3 \]
\[ v = 4*v*(1 - v) \]
\[ \text{return } v \]

or, in closed-form,

\[ f(x): \]
\[ \text{return } 64*x*(1-x)*((1-2*x)^2)*((1-8*x+8*x*x)^2) \]

**Symbolic Differentiation of the Closed-form**

\[ f'(x): \]
\[ \text{return } 128*x*(1-x)*(-8+16*x)*(-8+16*x)*(1-2*x)*((1-8*x+8*x*x)^2)*((1-8*x+8*x*x-x^2)^2)^2 - 64*x*(1-2*x)^2(1-8*x+8*x^2)^2 - 256*x*(1-x)*(1-2*x)*(1-8*x+8*x^2)^2 \]

\[ f'(x_0) = f'(x_0) \]

**Exact**

**Automatic Differentiation**

\[ f'(x): \]
\[ (v,dv) = (x,1) \]
\[ \text{for } i = 1 \text{ to } 3 \]
\[ (v,dv) = (4*v*(1-v), 4*dv-8*v*dv) \]
\[ \text{return } (v,dv) \]

\[ f'(x_0) = f'(x_0) \]

**Exact**

**Numerical Differentiation**

\[ f'(x): \]
\[ h = 0.000001 \]
\[ \text{return } (f(x + h) - f(x)) / h \]

\[ f'(x_0) \approx f'(x_0) \]

**Approximate**
How do we compute gradients?

• Analytic or “Manual” Differentiation

• Symbolic Differentiation

• Numerical Differentiation

• Automatic Differentiation
  – Forward mode AD
  – Reverse mode AD
    • aka “backprop”
current W:

\[
\begin{bmatrix}
0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, \ldots
\end{bmatrix}
\]

loss 1.25347

\[
W + h \text{ (first dim)}:
\begin{bmatrix}
0.34 + 0.0001, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, \ldots
\end{bmatrix}
\]

loss 1.25322

gradient dW:

\[
\begin{bmatrix}
-2.5, \\
?, \\
?, \\
(1.25322 - 1.25347)/0.0001 \\
= -2.5 \\
\end{bmatrix}
\]

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]

\[
?, \\
?, \ldots
\]
<table>
<thead>
<tr>
<th>current W:</th>
<th>$\mathbf{W} + h$ (second dim):</th>
<th>gradient $d\mathbf{W}$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]</td>
<td>[0.34, -1.11 + 0.0001, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]</td>
<td>[-2.5, 0.6, ?, ?, ?]</td>
</tr>
<tr>
<td>loss 1.25347</td>
<td>loss 1.25353</td>
<td>$(1.25353 - 1.25347)/0.0001 = 0.6$</td>
</tr>
</tbody>
</table>

$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}$
Numerical vs Analytic Gradients

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]

**Numerical gradient**: slow :, approximate :, easy to write :

**Analytic gradient**: fast :), exact :, error-prone :

In practice: Derive analytic gradient, check your implementation with numerical gradient. This is called a **gradient check**.

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
How do we compute gradients?

• Analytic or “Manual” Differentiation

• Symbolic Differentiation

• Numerical Differentiation

• Automatic Differentiation
  – Forward mode AD
  – Reverse mode AD
    • aka “backprop”
Matrix/Vector Derivatives Notation

\[
\frac{\partial y}{\partial x} = \begin{bmatrix}
\frac{\partial y}{\partial x_1} \\
\frac{\partial y}{\partial x_2} \\
\vdots \\
\frac{\partial y}{\partial x_n}
\end{bmatrix}
\]

\[
\frac{\partial y}{\partial x} = \frac{\partial y}{\partial x} \cdot \frac{\partial x}{\partial x} = \frac{\partial y}{\partial x}
\]

\[
x, y \in \mathbb{R}
\]
\[
x \in \mathbb{R}^d, y \in \mathbb{R}^c
\]
\[
X, Y \in \mathbb{R}^{m \times n}
\]

\[
um = \dim 1
\]

\[
den = \dim 2
\]
Vector/Matrix Derivatives Notation

\[ \frac{d\mathbf{y}}{d\mathbf{x}} = \begin{bmatrix} \frac{dy_1}{dx_1} & \cdots & \frac{dy_1}{dx_m} \\ \vdots & \ddots & \vdots \\ \frac{dy_n}{dx_1} & \cdots & \frac{dy_n}{dx_m} \end{bmatrix} \]
Vector Derivative Example

\[
\begin{align*}
\dot{y} &= \begin{bmatrix} y_1 \\
\vdots \\
y_n \end{bmatrix} = \begin{bmatrix} x \\
x^2 \end{bmatrix} \\
\frac{\partial \dot{y}}{\partial x} &= \begin{bmatrix} 1 \\
2x \end{bmatrix}
\end{align*}
\]

\[
y = \begin{bmatrix} \mathbf{w}^T \\
\end{bmatrix} \begin{bmatrix} x \end{bmatrix}
\]

\[
\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \cdots & \frac{\partial y}{\partial x_n} \end{bmatrix}
\]

\[
\frac{\partial (\sum w_i x_i)}{\partial x_i} = w_i
\]

\[
\begin{bmatrix} w_1 & \cdots & w_n \end{bmatrix}
\]
Vector Derivative Example

\[ \frac{\partial (w^T A x^T)}{\partial (w)} = 2 w^T A \]

\[ y_i = Ax \]

\[ \frac{\partial y_i}{\partial x_j} a_{ij} \]
Extension to Tensors

\[ X \in \mathbb{R}^{d_1 \ldots d_m} \]

\[ Y \in \mathbb{R}^{C_1 \times C_2 \times \ldots \times C_n} \]

\[ y_{vec} = Y(:, :) \]

\[ x_{vec} = X(:, :) \]

\[ \frac{\partial Y[i_1, \ldots, i_n]}{\partial x[j_1, \ldots, j_m]} \]

\[ \frac{\partial y_{vec}}{\partial x_{vec}} = \begin{bmatrix} \vdots \end{bmatrix} \]
Plan for Today

• (Finish) Analytical Gradients

• Automatic Differentiation
  – Computational Graphs
  – Forward mode vs Reverse mode AD
  – Patterns in backprop
Chain Rule: Composite Functions

\[ L(x) = f(g(x)) = (f \circ g)(x) \]

\[ f(x) = \underbrace{g_{e}(g_{e-1} \cdots g_{1}(x))}_{L(w)} = (g_{e} \circ g_{e-1} \cdots g_{1})(x) \]
Chain Rule: Scalar Case

\[ x \xrightarrow{g(\cdot)} z \xrightarrow{f(\cdot)} y \]

\[ = f\left( g(x) \right) \]

\[ \frac{dy}{dx} = \frac{dy}{dz} \cdot \frac{dz}{dx} \]

Scalar prod.
Chain Rule: Vector Case

\[ \begin{align*}
\vec{x} &\in \mathbb{R}^d \quad &\xrightarrow{g(\cdot)}\quad \vec{z} &\in \mathbb{R}^m \\
\vec{z} &\in \mathbb{R}^m \quad &\xrightarrow{f(\cdot)}\quad \vec{y} &\in \mathbb{R}^c
\end{align*} \]

\[
\frac{\partial \vec{y}}{\partial \vec{x}} = \left( \frac{\partial \vec{y}}{\partial \vec{z}} \right) \cdot \left( \frac{\partial \vec{z}}{\partial \vec{x}} \right)
\]

\[ J_{f \circ g} = J_f \text{ Matrix Mult} \]

(C) Dhruv Batra
Chain Rule: Jacobian view

\[ i \begin{bmatrix} \frac{\partial y_i}{\partial x_j} \\ \vdots \\ \frac{\partial y_i}{\partial x_j} \end{bmatrix}_{c \times d} = i \begin{bmatrix} \frac{\partial y_i}{\partial z_k} \\ \vdots \\ \frac{\partial y_i}{\partial z_k} \end{bmatrix}_{c \times m} \begin{bmatrix} \frac{\partial z_k}{\partial x_j} \\ \vdots \\ \frac{\partial z_k}{\partial x_j} \end{bmatrix}_{m \times d} \]
Chain Rule: Graphical view

\[ \frac{\partial y_i}{\partial x_j} = \sum \text{paths} \]

\[ \frac{\partial^2 y_i}{\partial x_j \partial x_k} \text{ where } k \text{ is on path} \]
Linear Classifier: Logistic Regression

Input: $x \in \mathbb{R}^D$

Binary label: $y \in \{-1, +1\}$

Parameters: $w \in \mathbb{R}^D$

Output prediction: $p(y=1|x) = \frac{1}{1 + e^{-w^T x}}$

Loss: $L = \frac{1}{2} \left\| w \right\|^2 - \lambda \log(p(y|x))$

Log Loss

\[ \frac{\partial L}{\partial w} \]
Logistic Regression Derivatives

\[ L_i = -\log \left( \frac{1}{1 + e^{-w^T x}} \right) = \log \left( 1 + e^{-w^T x} \right) \]

\[ \frac{\partial L_i}{\partial w} = \frac{\partial L_i}{\partial p} \cdot \frac{\partial p}{\partial u} \cdot \frac{\partial u}{\partial w} \]

\[ = \left( \frac{1}{1 + e^{-w^T x}} \right) \left( e^{-w^T x} \right) \left( -w X^T \right) \]
Logistic Regression Derivatives
Chain Rule: Cascaded
Chain Rule: How should we multiply?
Convolutional network (AlexNet)
Neural Turing Machine

Figure reproduced with permission from a Twitter post by Andrej Karpathy.

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
How do we compute gradients?

• Analytic or “Manual” Differentiation

• Symbolic Differentiation

• Numerical Differentiation

• Automatic Differentiation
  – Forward mode AD
  – Reverse mode AD
    • aka “backprop”
Plan for Today

• (Finish) Analytical Gradients

• Automatic Differentiation
  – Computational Graphs
  – Forward mode vs Reverse mode AD
Deep Learning = Differentiable Programming

- Computation = Graph
  - Input = Data + Parameters
  - Output = Loss
  - Scheduling = Topological ordering

- Auto-Diff
  - A family of algorithms for implementing chain-rule on computation graphs
Computational Graph

\[ f = Wx \]

\[ L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Any DAG of differentiable modules is allowed!
Directed Acyclic Graphs (DAGs)

• Exactly what the name suggests
  – Directed edges
  – No (directed) cycles
  – Underlying undirected cycles okay

\[ G = \left( \mathcal{V}, \mathcal{E} \right) \]
\[ \mathcal{E} = \left\{ (v_i, v_j) \mid v_i, v_j \in \mathcal{V} \right\} \]
Directed Acyclic Graphs (DAGs)

- Concept
  - Topological Ordering

\[
\exists \text{ bijection } \sigma : V \rightarrow \{1, \ldots, n\} \text{ such that } \forall (v_i, v_j) \in E \text{ implies } \sigma(v_i) < \sigma(v_j)
\]
Directed Acyclic Graphs (DAGs)
Computation Graphs

- Notation

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]
Example

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]
\[ f(x) = \sigma \left( \log \left( 5 \left( \max\{x_1, x_2\} \cdot \frac{x_3}{x_4} - (x_5 + x_6) \right) \right) + \frac{1}{2} \right) \]
\[ f(x) = \sigma \left( \log \left( 5 \left( \max\{x_1, x_2\} \cdot \frac{x_3}{x_4} - (x_5 + x_6) \right) \right) + \frac{1}{2} \right) \]
Logistic Regression as a Cascade

Given a library of simple functions

\[
\begin{align*}
\sin(x) & \\
\cos(x) & \\
x^3 & \\
\exp(x) & \\
\log(x) & 
\end{align*}
\]

Compose into a complicate function

\[
- \log \left( \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}}} \right)
\]

(C) Dhruv Batra

Slide Credit: Marc'Aurelio Ranzato, Yann LeCun
Deep Learning = Differentiable Programming

- **Computation = Graph**
  - Input = Data + Parameters
  - Output = Loss
  - Scheduling = Topological ordering

- **Auto-Diff**
  - A family of algorithms for implementing chain-rule on computation graphs

(C) Dhruv Batra
Forward mode vs Reverse Mode

- Key Computations
Forward mode AD

Layer $l$

Input: \[
\frac{\partial h}{\partial x} \]

Goal: \[
\frac{\partial L}{\partial x}
\]

Calculation:
\[
\frac{\partial h}{\partial x} = \frac{\partial h^l}{\partial h^{l-1}} \cdot \frac{\partial h^{l-1}}{\partial x}
\]

Jacobian of $g$ Input
Reverse mode AD

\[ g = Ax \]
\[ \frac{\partial g}{\partial x} = A \]
\[ \frac{\partial^2 g}{\partial x^2} = WA \]
\[ \frac{\partial^2 g}{\partial x \partial e} = W \]

Goal: \[ \frac{\partial L}{\partial x} \]

\[ \hat{h}^{l-1} \]

\[ h = g(\hat{h}^{l-1}) \]

\[ \frac{\partial L}{\partial \hat{h}^l} = \frac{\partial L}{\partial \hat{h}} \frac{\partial \hat{h}}{\partial \hat{h}^l} \]

Input

Output
Example: Forward mode AD

\[
f(x_1, x_2) = x_1 x_2 + \sin(x_1)
\]

\[
\frac{df}{dx} = \frac{\partial f}{\partial x_1} \frac{dx_1}{dx} + \frac{\partial f}{\partial x_2} \frac{dx_2}{dx}
\]

\[
\frac{\partial f}{\partial x_1} = x_2 \\
\frac{\partial f}{\partial x_2} = x_1
\]

\[
\frac{\partial w_2}{\partial a} = x_i \frac{\partial x_2}{\partial a} + x_2 \frac{\partial x_i}{\partial a} \\
= x_1 x_2 + x_2 x_1
\]

\[
\alpha \in \{x_1, x_2\}
\]
Example: Forward mode AD

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]
Example: Forward mode AD

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]
Example: Forward mode AD

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]

\[
\dot{\omega}_3 = \dot{\omega}_1 + \dot{\omega}_2 = \cos(x_1) \dot{x}_1 + x_1 \dot{x}_2 + \dot{x}_1 x_2
\]

\[ \frac{df}{dx_1} \quad \frac{df}{dx_2} \]
Example: Forward mode AD

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]

\[ \dot{w}_3 = \dot{w}_1 + \dot{w}_2 \]

\[ \dot{w}_1 = \cos(x_1) \dot{x}_1 \quad \dot{w}_2 = \dot{x}_1 x_2 + x_1 \dot{x}_2 \]
Example: Reverse mode AD

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]
Example: Reverse mode AD

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]
Example: Reverse mode AD

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]
Forward Pass vs Forward mode AD vs Reverse Mode AD

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]
Forward mode vs Reverse Mode

• What are the differences?

\[
\dot{w}_3 = \dot{w}_1 + \dot{w}_2
\]

\[
\dot{w}_1 = \cos(x_1) \dot{x}_1 \quad \dot{w}_2 = \dot{x}_1 x_2 + x_1 \dot{x}_2
\]

\[
\bar{x}_1 = \bar{w}_1 \cos(x_1) \quad \bar{x}_1 = \bar{w}_2 x_2 \quad \bar{x}_2 = \bar{w}_2 x_1
\]
Forward mode vs Reverse Mode

• What are the differences?

• Which one is faster to compute?
  – Forward or backward?
Forward mode vs Reverse Mode

• What are the differences?

• Which one is faster to compute?
  – Forward or backward?

• Which one is more memory efficient (less storage)?
  – Forward or backward?