CS 7643: Deep Learning

Topics:
- Computational Graphs
  - Notation + example
- Computing Gradients
  - Forward mode vs Reverse mode AD

Dhruv Batra
Georgia Tech
Administrativia

- HW1 Released
  - Due: 09/22

- PS1 Solutions
  - Coming soon
Project

• **Goal**
  – Chance to try Deep Learning
  – **Combine with other classes / research / credits / anything**
    • You have our blanket permission
    • Extra credit for shooting for a publication
  – Encouraged to apply to your research (computer vision, NLP, robotics,…)
  – Must be done this semester.

• **Main categories**
  – **Application/Survey**
    • Compare a bunch of existing algorithms on a new application domain of your interest
  – **Formulation/Development**
    • Formulate a new model or algorithm for a new or old problem
  – **Theory**
    • Theoretically analyze an existing algorithm
Administrativia

• Project Teams Google Doc
  – [https://docs.google.com/spreadsheets/d/1AaXY0JE4lAbHvoDaWlc9zsmfKMyuGS39JAn9dpeXhhQ/edit#gid=0](https://docs.google.com/spreadsheets/d/1AaXY0JE4lAbHvoDaWlc9zsmfKMyuGS39JAn9dpeXhhQ/edit#gid=0)
  – Project Title
  – 1-3 sentence project summary TL;DR
  – Team member names + GT IDs
Recap of last time
How do we compute gradients?

• Manual Differentiation

• Symbolic Differentiation

• Numerical Differentiation

• Automatic Differentiation
  – Forward mode AD
  – Reverse mode AD
    • aka “backprop”
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
Directed Acyclic Graphs (DAGs)

• Concept
  – Topological Ordering

\[ \sigma : V \rightarrow [n] = \{1, \ldots, n\} \]

\[ s.t \ (v_i, v_j) \in E \quad \sigma(v_i) < \sigma(v_j) \]
Directed Acyclic Graphs (DAGs)

\[ a_{ij} = 1 \quad (i,j) \in F \]

\[ A = \]

\[ n \times n \]
Computational Graphs

- Notation #1

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

• Notation #2

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

\[ f(x_1, x_2) = x_1 x_2 + \sin(x_1) \]
Logistic Regression as a Cascade

Given a library of simple functions

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

Compose into a complicate function

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

Slide Credit: Marc'Aurelio Ranzato, Yann LeCun
Forward mode vs Reverse Mode

• Key Computations
Forward mode AD

layer $g(\cdot)$

$h^e = g(h^{e-1})$

Jacobian
Reverse mode AD
Example: Forward mode AD

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

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

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

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

\[ a = x_1 \]

\[ x_1 \]
\[ x_2 \]

\[ \sin(\cdot) \]
\[ + \]

\[ \times \]
Example: Forward 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

\[ f(x_1, x_2) = x_1x_2 + \sin(x_1) \]
Forward Pass vs Forward mode AD vs Reverse Mode AD

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

\[ \begin{align*}
\dot{x}_1 &= \dot{x}_2 = x_1 \dot{x}_2 + x_1 \dot{x}_2 \\
\dot{x}_1 &= \cos(x_1) \dot{x}_1 \\
\dot{x}_2 &= \sin(x_1) \dot{x}_2 \\
\dot{x}_3 &= \dot{x}_1 + \dot{x}_2 \\
\dot{w}_1 &= w_1 \cos(x_1) \dot{x}_1 \\
\dot{w}_2 &= \dot{x}_2 x_2 + x_1 \dot{x}_2 \\
\dot{w}_3 &= 1 \\
\end{align*} \]
Forward mode vs Reverse Mode

• What are the differences?

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

• What are the differences?

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

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

• (Finish) Computing Gradients
  – Forward mode vs Reverse mode AD
  – Patterns in backprop
  – Backprop in FC+ReLU NNs

• Convolutional Neural Networks
Patterns in backward flow

\[ f(\ldots) = 2 \left( zy + \max(z, w) \right) \]
Patterns in backward flow

\[ w_3 = w_1 + w_2 \]

\[ \frac{\partial w_3}{\partial w_1} = 1 \]

\[ w_1 = x y \]

\[ \frac{\partial w_1}{\partial x} = y \]

\[ \frac{\partial w_1}{\partial w_3} = \frac{2}{w_3} \]

\[ f = 2 \cdot w_3 \]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Patterns in backward flow

**add gate: gradient distributor**

\[ w_3 = w_1 + w_2 \]

\[ \frac{\partial w_3}{\partial w_1} = 1 \]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
add gate: gradient distributor

Q: What is a max gate?
Patterns in backward flow

**add** gate: gradient distributor

**max** gate: gradient router

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
**Patterns in backward flow**

**add gate:** gradient distributor

**max gate:** gradient router

**Q:** What is a **mul** gate?
Patterns in backward flow

**add** gate: gradient distributor

**max** gate: gradient router

**mul** gate: gradient switcher

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Gradients add at branches
Duality in Fprop and Bprop

- **FPROP**
  - SUM
  - COPY

- **BPROP**
  - \( \frac{df}{dw} \)
  - \( \frac{df}{dx} \)
  - \( \frac{df}{\theta} \)
Modularized implementation: forward / backward API

Graph (or Net) object  \textit{(rough pseudo code)}

```python
class ComputationalGraph(object):
    
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss

    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Modularized implementation: forward / backward API

(x, y, z are scalars)

```python
class MultiplyGate(object):
    def forward(x, y):
        z = x * y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Modularized implementation: forward / backward API

\[
(x, y, z \text{ are scalars})
\]

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

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Example: Caffe layers

Caffe is licensed under BSD 2-Clause
Caffe Sigmoid Layer

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

\[(1 - \sigma(x)) \sigma(x) \]

* top_diff (chain rule)
$\frac{\partial L}{\partial W_l}$
Key Computation in DL: Forward-Prop
Key Computation in DL: Back-Prop

\[ \frac{\partial L}{\partial X} = \{ \frac{\partial Z}{\partial X}, \frac{\partial Z}{\partial \theta} \} \]

\[ X^{(t+1)} \leftarrow X^{(t)} - \eta \frac{\partial L}{\partial X} \]

\[ \theta^{t+1} = \theta^{t} - \eta \frac{\partial L}{\partial \theta} \]
Jacobian of ReLU

$$f(x) = \max(0, x)$$ (elementwise)

4096-d input vector

4096-d output vector

$$\frac{\partial h_i^l}{\partial h_j^l} = 0 \quad \text{if} \quad i \neq j$$

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
The function $f(x) = \max(0, x)$ is defined elementwise.

**Q:** What is the size of the Jacobian matrix?

**Slide Credit:** Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Jacobian of ReLU

Q: what is the size of the Jacobian matrix? [4096 x 4096!]

f(x) = max(0, x) (elementwise)

4096-d input vector  

4096-d output vector

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
i.e. Jacobian would technically be a \(4096 \times 4096\) matrix:

\[ f(x) = \max(0, x) \quad \text{(elementwise)} \]

4096-d input vector \quad 4096-d output vector

Q: what is the size of the Jacobian matrix? [4096 x 4096!]

In practice we process an entire minibatch (e.g. 100) of examples at one time:

i.e. Jacobian would technically be a \([409,600 \times 409,600]\) matrix.\"
Jacobian of ReLU

4096-d input vector \[ f(x) = \max(0,x) \text{ (elementwise)} \]

4096-d output vector

Q: what is the size of the Jacobian matrix? \([4096 \times 4096]!\)

Q2: what does it look like?

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Jacobians of FC-Layer

\[
\begin{align*}
\frac{\partial L}{\partial h} &= \text{(Expression)} \\
\frac{\partial L}{\partial h_i} &= \text{(Expression)} \\
\end{align*}
\]
Jacobians of FC-Layer

\[ h^l \xrightarrow{\text{Layer}} h^l \in \mathbb{R}^2 \]

\[ \frac{\partial h^l}{\partial w} \]

\( \mathbb{C}_2 \times \mathbb{C}_1 \)

\[ h^l = W_i h^l_{i-1} \]

\( \frac{\partial h^l}{\partial w} = (h^l)^T \)
Jacobians of FC-Layer
Convolutional Neural Networks
(without the brain stuff)
- Spatial correlation is local
- Waste of resources + we have not enough training samples anyway..

Example: 200x200 image
40K hidden units
~2B parameters!!!
Locally Connected Layer

Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

Note: This parameterization is good when input image is registered (e.g., face recognition).
Locally Connected Layer

**STATIONARITY?** Statistics is similar at different locations

Example: 200x200 image
40K hidden units
Filter size: 10x10
4M parameters

**Note:** This parameterization is good when input image is registered (e.g., face recognition).
Share the same parameters across different locations (assuming input is stationary):
Convolutions with learned kernels
Convolutions for mathematicians

\[ x(t) \ast y(t) = w(t) \]

\[ y(t) = (x \ast w)(t) = \int_{-\infty}^{\infty} x(t-a) w(a) \, da \]

\[ w(a) \rightarrow \mathcal{F} w(-a) \]

\[ w(-a) \rightarrow w(tf(t-a)) \]
\[ y(t_1, t_2) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} x(a-t_1, b-t_2) w(a,b) \, da \, db \]
Convolutions for computer scientists

\[ y[a, c] = \sum_{a=-\frac{N-1}{2}}^{\frac{N-1}{2}} \sum_{b=-\frac{N-1}{2}}^{\frac{N-1}{2}} x[a-a, c-b] w[a, b] \]
Convolutions for programmers

\[ y[a, c] = \sum_{a=0}^{N_1-1} \sum_{b=0}^{N_2-1} x[a+a', c+b'] n[a, b] \]
Convolution Explained

• http://setosa.io/ev/image-kernels/

• https://github.com/bruckner/deepViz
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Convolutional Layer
Mathieu et al. “Fast training of CNNs through FFTs” ICLR 2014
Convolutional Layer

\[
\begin{bmatrix}
-1 & 0 & 1 \\
-1 & 0 & 1 \\
-1 & 0 & 1 \\
\end{bmatrix}
\ast
\begin{bmatrix}
\end{bmatrix}
= 
\begin{bmatrix}
\end{bmatrix}
\]
Learn multiple filters.

E.g.: 200x200 image
100 Filters
Filter size: 10x10
10K parameters
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Fully Connected Layer

32x32x3 image -> stretch to 3072 x 1

input

\[ Wx \]

10 x 3072 weights

activation

1 number:
the result of taking a dot product between a row of W and the input (a 3072-dimensional dot product)

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Convolutional Layer
Convolutional Layer
Convolution Layer

32x32x3 image -> preserve spatial structure

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Convolution Layer

32x32x3 image

5x5x3 filter

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”
Convolution Layer

32x32x3 image

5x5x3 filter

Filters always extend the full depth of the input volume

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Convolution Layer

32x32x3 image

5x5x3 filter \( w \)

1 number: the result of taking a dot product between the filter and a small 5x5x3 chunk of the image (i.e. 5*5*3 = 75-dimensional dot product + bias)

\[ w^T x + b \]
Convolution Layer

32x32x3 image
5x5x3 filter
convolve (slide) over all spatial locations
Convolution Layer

- 32x32x3 image
- 5x5x3 filter

Convolve (slide) over all spatial locations

Consider a second, green filter

Activation maps

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
For example, if we had 6 5x5 filters, we’ll get 6 separate activation maps:

We stack these up to get a “new image” of size 28x28x6!