CMSC 838B & 498Z: Differentiable Programming

Tues/Thur 12:30pm – 1:45pm
IRB 4105 (T) & IRB 5105 (R)
http://www.cs.umd.edu/class/fall2021/cmsc838b

Ming C. Lin
IRB 5162
lin@cs.umd.edu
http://www.cs.umd.edu/~lin
Office Hours: After Class or By Appointment

Why Automatic Differentiation (AD)?

To solve optimization problems using gradient methods we need to compute the gradients (derivatives) of the objective w.r.t. the parameters

- In neural nets we’re talking about the gradients of the loss function w.r.t. the parameters \( \theta \): \( \nabla L = \frac{\partial L}{\partial \theta} \)

- AD is important - it’s been suggested that “Differentiable programming” could be the term that ultimately replaces deep learning

M. C. Lin
Computing Derivatives

Three ways to compute derivatives –

● Symbolically differentiate the function w.r.t. its parameters
  – Problem: Static - can’t “differentiate algorithms”

● Make estimates using finite differences
  – Problems: Numerical errors - will compound in deep nets

● Use Automatic Differentiation

What is Automatic Differentiation (AD)

● A method to get exact derivatives efficiently, by storing information as you go forward that you can reuse as you go backwards
  – Takes code that computes a function and uses that to compute the derivative of that function
  – The goal isn't to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.
Differentiation and Programming

Example (Math)

\[ x = ? \]
\[ y = ? \]
\[ a = xy \]
\[ b = \sin(x) \]
\[ z = a + b \]

Example (Code)

\[ x = ? \]
\[ y = ? \]
\[ a = x * y \]
\[ b = \sin(x) \]
\[ z = a + b \]

The Chain Rule of Differentiation

Recall the chain rule for a variable/function \( z \) that depends on \( y \) which depends on \( x \):

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

- In general, the chain rule can be expressed as:

\[
\frac{\partial w}{\partial t} = \sum_{i=1}^{N} \frac{\partial w}{\partial u_i} \frac{\partial u_i}{\partial t} = \frac{\partial w}{\partial u_1} \frac{\partial u_1}{\partial t} + \frac{\partial w}{\partial u_2} \frac{\partial u_2}{\partial t} + \cdots + \frac{\partial w}{\partial u_N} \frac{\partial u_N}{\partial t}
\]

where \( w \) is some output variable, and \( u_i \) denotes each input variable \( w \) depends on.

M. C. Lin
Applying the Chain Rule

- Let's differentiate the previous expression w.r.t. some yet to be given variable t:

<table>
<thead>
<tr>
<th>Expression</th>
<th>( \frac{\partial x}{\partial t} = ? )</th>
<th>( \frac{\partial y}{\partial t} = ? )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = ? )</td>
<td>( a = x y )</td>
<td>( b = \sin(x) )</td>
</tr>
<tr>
<td>( y = ? )</td>
<td>( z = a + b )</td>
<td></td>
</tr>
</tbody>
</table>

- If we substitute \( t = x \) in the above we'll have an algorithm for computing \( dz/dx \). To get \( dz/dy \) we'd just substitute \( t = y \).

Translating to code I

We could translate the previous expressions back into a program involving differential variables \( \{dx, dy, \ldots\} \) which represent \( dx/dt, dy/dt, \ldots \) respectively:

- \( dx = ? \)
- \( dy = ? \)
- \( da = y * dx + x * dy \)
- \( db = \cos(x) * dx \)
- \( dz = da + db \)

What happens to this program if we substitute \( t = x \) into the math expression?
Translating to code II

\[
\begin{align*}
\text{dx} &= 1 \\
\text{dy} &= 0 \\
\text{da} &= y \times \text{dx} + x \times \text{dy} \\
\text{db} &= \cos(x) \times \text{dx} \\
\text{dz} &= \text{da} + \text{db}
\end{align*}
\]

The effect is remarkably simple: to compute \( \frac{dz}{dx} \) we just seed the algorithm with \( \text{dx}=1 \) and \( \text{dy}=0 \).

Translating to code III

\[
\begin{align*}
\text{dx} &= 0 \\
\text{dy} &= 1 \\
\text{da} &= y \times \text{dx} + x \times \text{dy} \\
\text{db} &= \cos(x) \times \text{dx} \\
\text{dz} &= \text{da} + \text{db}
\end{align*}
\]

To compute \( \frac{dz}{dy} \) we just seed the algorithm with \( \text{dx}=0 \) and \( \text{dy}=1 \).
Making Rules

- We’ve successfully computed the gradients for a specific function, but the process was far from automatic.
- We need to formalize a set of rules for translating a program that evaluates an expression into a program that evaluates its derivatives.
- We have actually already discovered 3 of these rules:

\[ \begin{align*}
  c &= a + b & \Rightarrow & & dc = da + db \\
  c &= a \times b & \Rightarrow & & dc = b \times da + a \times db \\
  c &= \sin(a) & \Rightarrow & & dc = \cos(a) \times da \\
\end{align*} \]

More Rules

These initial rules:

- \[ c = a + b \quad \Rightarrow \quad dc = da + db \]
- \[ c = a \times b \quad \Rightarrow \quad dc = b \times da + a \times db \]
- \[ c = \sin(a) \quad \Rightarrow \quad dc = \cos(a) \times da \]

Can easily be extended further using multivariable calculus:

- \[ c = a - b \quad \Rightarrow \quad dc = da - db \]
- \[ c = a/b \quad \Rightarrow \quad dc = da/b - a \times db/b^2 \]
- \[ c = \cos(a) \quad \Rightarrow \quad dc = -\sin(a) \times da \]
- \[ c = \tan(a) \quad \Rightarrow \quad dc = da/\cos(a)^2 \]
Forward Mode AD

- To translate using the rules we simply replace each primitive operation in the original program by its differential analogue.

- The order of computation remains unchanged: if a statement K is evaluated before another statement L, then the differential analogue of K is evaluated before the analogue statement of L.

- This is **Forward-mode Automatic Differentiation**

Reversing the Chain Rule

- The chain rule is symmetric — this means we can turn the derivatives upside-down:

\[
\frac{\partial s}{\partial u} = \sum_{i}^{N} \frac{\partial w_i}{\partial u} \frac{\partial s}{\partial w_i} = \frac{\partial w_1}{\partial u} \frac{\partial s}{\partial w_1} + \frac{\partial w_2}{\partial u} \frac{\partial s}{\partial w_2} + \cdots + \frac{\partial w_N}{\partial u} \frac{\partial s}{\partial w_N}
\]

- In doing so, we have inverted the input-output role of the variables: \( u \) is some input variable, the \( w_i \)'s are the output variables that depend on \( u \). \( s \) is the yet-to-be-given variable.

- In this form, the chain rule can be applied repeatedly to every input variable \( u \) (akin to how in forward mode we repeatedly applied it to every \( w \)). Therefore, given some \( s \) we expect this form of the rule to give us a program to compute both \( ds/dx \) and \( ds/dy \) in one go.

M. C. Lin
Reversing the Chain Rule: Example

\[ \frac{\partial s}{\partial u} = \sum \frac{\partial w_i}{\partial u} \frac{\partial s}{\partial w_i} \]

\[ \frac{\partial s}{\partial z} = ? \]

\[ \frac{\partial s}{\partial b} = \frac{\partial z}{\partial b} \frac{\partial s}{\partial z} = \frac{\partial s}{\partial z} \]

\[ \frac{\partial s}{\partial a} = \frac{\partial z}{\partial a} \frac{\partial s}{\partial z} = \frac{\partial s}{\partial z} \]

\[ x = ? \]
\[ y = ? \]
\[ a = xy \]
\[ b = \sin(x) \]
\[ z = a + b \]

Visualizing Dependencies

- Differentiating in reverse can be quite mind-bending: instead of asking what input variables an output depends on, we have to ask what output variables a given input variable can affect.

- We can see this visually by drawing a dependency graph of the expression:

```
   x
   / \    y
  /   \   .
 sin  \   
 a   \   
 b +   
   z
```
Translating to Code

- Let’s now translate our derivatives into code. As before we replace the derivatives (ds/dz, ds/db, . . .) with variables (gz, gb, ...) which we call adjoint variables:
  
  
  \[
  \begin{align*}
  gz &= \text{?} \\
  gb &= gz \\
  ga &= gz \\
  gy &= x \cdot ga \\
  gx &= y \cdot ga + \cos(x) \cdot gb
  \end{align*}
  \]

- If we go back to the equations and substitute s = z we would obtain the gradient in the last two equations. In the above program, this is equivalent to setting gz = 1.

- This means to get the both gradients dz/dx and dz/dy we only need to run the program once!

Limitations of Reverse Mode AD

- If we have multiple output variables, we’d have to run the program for each one (with different seeds on the output variables). For example:

  \[
  \begin{align*}
  z &= 2x + \sin x \\
  v &= 4x + \cos x
  \end{align*}
  \]

- We can’t just interleave the derivative calculations (since they all appear to be in reverse). . . How can we make this automatic?
Implementing Reverse Mode AD

There are two ways to implement Reverse AD:

1. We can parse the original program and generate the adjoint program that calculates the derivatives:
   - Potentially hard to do.
   - Static, so can only be used to differentiate algorithms that have parameters predefined.
   - But, efficient (lots of opportunities for optimisation)

2. We can make a dynamic implementation by constructing a graph that represents the original expression as the program runs.

Constructing an Expression Graph

The "roots" of the graph are the independent variables x and y. Constructing these nodes is as simple as creating an object:

```python
class Var:
    def __init__(self, value):
        self.value = value
        self.children = []

x = Var(0.5)
y = Var(4.2)
```

Each Var node can have children which are the nodes that depend directly on that node. The children allow nodes to link together in a Directed Acyclic Graph.
Building Expressions

By default, nodes do not have any children. As expressions are created, each expression \( u \) registers itself as a child of each of its dependencies \( w \) together with its weight \( \frac{\partial w}{\partial u} \) which will be used to compute gradients:

```python
class Var :
    ...
    def __mul__( self , other ) :
        z = Var ( self . value * other . value )
        # weight = dz/ dself = other . value
        self . children . append (( other . value , z))
        # weight = dz/ dother = self . value
        other . children . append (( self . value , z))
        return z
    ...  

# " a" is a new Var that is a child of both x and y
a = x * y
```

Computing Gradients

Finally, to get the gradients we need to propagate the derivatives. To avoid unnecessarily traversing the tree multiple times we will cache the derivative of a node in an attribute `grad_value`:

```python
class Var :
    def __init__ ( self ) :
        ...
        self . grad_value = None

    def grad ( self ) :
        if self . grad_value is None :
            # calculate derivative using chain rule
            self . grad_value = sum ( weight * var . grad () for weight , var in self . children )
            return self . grad_value
        ...

a . grad_value = 1 . 0
print ("da / dx u= u {} "). format ( x . grad () )
```
### AD in the PyTorch Autograd Package

- PyTorch's AD is remarkably similar to the one we've just built:
  - it eschews the use of a tape
  - it builds the computation graph as it runs (recording explicit `Function` objects as the children of Tensors rather than grouping everything into `Var` objects)
  - it caches the gradients in the same way we do (in the `grad` attribute) - hence the need to call `zero_grad()` when recomputing the gradients of the same graph after a round of backprop.
- PyTorch does some clever memory management to work well in a reference-counted regime and aggressively frees values that are no longer needed.
- The backend is actually mostly written in C++, so it's fast, and can be multi-threaded (avoids problems of the GIL)
- It allows easy "turning off" of gradient computations through `requires_grad`
- In-place operations which invalidate data needed to compute derivatives...