Name:

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

M. C. Lin

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 $\boldsymbol{\theta}$: $\nabla L = \frac{\partial L}{\partial \theta}$
- 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 Problems: Numerical errors - will compound in deep nets
- **Use Automatic Differentiation**

M. C. Lin

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.

M. C. Lin

Name:

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:

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

where w is some output variable, and u_i denotes each input variable w depends on

M. C. Lin

Name:

Applying the Chain Rule

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

Expression
$x = ?$
$y = ?$
$a = xy$
$b = \sin(x)$
$z = a + b$

$$\frac{\partial x}{\partial t} = ?$$
$$\frac{\partial y}{\partial t} = ?$$
$$\frac{\partial a}{\partial t} = x\frac{\partial y}{\partial t} + y\frac{\partial x}{\partial t}$$
$$\frac{\partial b}{\partial t} = \cos(x)\frac{\partial x}{\partial t}$$
$$\frac{\partial z}{\partial t} = \frac{\partial a}{\partial t} + \frac{\partial b}{\partial t}$$

- 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

M. C. Lin

Translating to code I

We could translate the previous expressions back into a program involving differential variables {dx, dy, ...} which represent dx/dt, dy/dt, . . . 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?

M. C. Lin

Name:

Translating to code II

```
dx = 1
dy = 0
da = y * dx + x * dy
db = cos (x) * dx
dz = da + db
```

The effect is remarkably simple: to compute dz/dx we just seed the algorithm with dx=1 and dy=0.

M. C. Lin

Translating to code III

```
dx = 0
dy = 1
da = y * dx + x * dy
db = cos (x) * dx
dz = da + db
```

To compute dz/dy we just seed the algorithm with dx=0 and dy=1

M. C. Lin

Name:

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:

```
c = a + b    =>  dc = da + db
c = a * b    =>  dc = b * da + a * db
c = sin(a)   =>  dc = cos(a) * da
```

M. C. Lin

More Rules

These initial rules:

```
c=a+b       =>   dc=da+db
c=a*b       =>   dc=b*da+a*db
c=sin(a)    =>   dc=cos(a)*da
```

can easily be extended further using multivariable calculus:

```
c=a-b       =>   dc=da -db
c=a/b       =>   dc=da/b-a*db/b**2
c=cos(a)    =>   dc=-sin(a)*da
c=tan(a)    =>   dc=da/cos(a)**2
```

M. C. Lin

Name:

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**

M. C. Lin

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 wi '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

Name:

Reversing the Chain Rule: Example

$$\frac{\partial s}{\partial u} = \sum_{i}^{N} \frac{\partial w_i}{\partial u} \frac{\partial s}{\partial w_i}$$

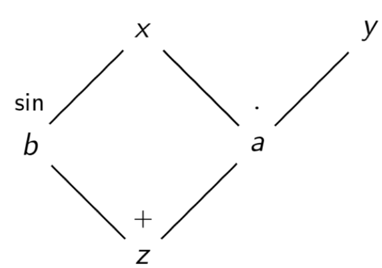
$$\begin{aligned} x &= ? \\ y &= ? \\ a &= x\,y \\ b &= \sin(x) \\ z &= a + b \end{aligned}$$

$$\begin{aligned} \frac{\partial s}{\partial z} &= ? \\ \frac{\partial s}{\partial b} &= \frac{\partial z}{\partial b}\frac{\partial s}{\partial z} = \frac{\partial s}{\partial z} \\ \frac{\partial s}{\partial a} &= \frac{\partial z}{\partial a}\frac{\partial s}{\partial z} = \frac{\partial s}{\partial z} \\ \frac{\partial s}{\partial y} &= \frac{\partial a}{\partial y}\frac{\partial s}{\partial a} = x\,\frac{\partial s}{\partial a} \\ \frac{\partial s}{\partial x} &= \frac{\partial a}{\partial x}\frac{\partial s}{\partial a} + \frac{\partial b}{\partial x}\frac{\partial s}{\partial b} \\ &= y\,\frac{\partial s}{\partial a} + \cos(x)\,\frac{\partial s}{\partial b} \\ &= (y + \cos(x))\,\frac{\partial s}{\partial z} \end{aligned}$$

M. C. Lin

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:



M. C. Lin

Name:

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:

```
gz = ?
gb = gz
ga = gz
gy = x * ga
gx = y * ga + cos (x) * gb
```

- 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!

M. C. Lin

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:

$$\begin{cases} z = 2x + \sin x \\ v = 4x + \cos x \end{cases}$$

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

M. C. Lin

Name:

Implementing Reverse Mode AD

There are two ways to implement Reverse AD:

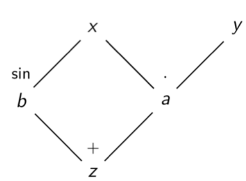
1. We can parse the original program and generate the adjoint program that calculates the derivatives:
   - Potentially hard to do.
   - Static, so can only be used to differentiate algorithms that have parameters predefined.
   - But, efficient (lots of opportunities for optimisation)

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

M. C. Lin

Constructing an Expression Graph

The goal is to get something akin to the graph we saw earlier:



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

```
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.**

M. C. Lin

Name:

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_i together with its weight $\partial w_i/\partial u$ which will be used to compute gradients:

```
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
```

M. C. Lin

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`:

```
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 () )
```

M. C. Lin

Name:

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 its 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

M. C. Lin