Why Differentiable Programming?

UCSD CSE 291
Tzu-Mao Li
Derivative-based optimization is becoming a central piece of computer science.

[Diagram showing parameters flowing through a differentiable model to an objective function.]
Neural networks are a popular differentiable model
How to implement a Convolutional Neural Network?
How to implement a Convolutional Neural Network?
How to implement a Convolutional Neural Network in 2012?
How did Alex Krizhevsky implement a Convolutional Neural Network in 2012?

Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440→186,624→64,896→64,896→43,264→4096→1000.
How did Alex Krizhevsky implement a Convolutional Neural Network in 2012?
Deep learning frameworks are fundamentally why deep learning is successful.

Before required expertise:
- program differentiation
- numerical computing
- GPU hacking
- optimization

After sliding courtesy of Caroline Lemieux
If you do well in this class, you will understand how deep learning frameworks are made.
Neural networks are a very limited form of differentiable models.

\[ \frac{\partial}{\partial} \text{parameters} \]
The blessing and curse of deep learning frameworks

- chicken & egg problem: the feedback loop reinforces the design to cater existing “layers”
The blessing and curse of deep learning frameworks

- chicken & egg problem: the feedback loop reinforces the design to cater existing “layers”

Machine Learning Systems are Stuck in a Rut

Paul Barham
Google Brain

Michael Isard
Google Brain

In this paper we argue that systems for numerical computing are stuck in a local basin of performance and programmability. Systems researchers are doing an excellent job improving the performance of 5-year-old benchmarks, but gradually making it harder to explore innovative machine learning research ideas.
Deep learning frameworks are limited

conv4d and higher dimension generalization #1661

Closed jerabaul29 opened this issue on Mar 26, 2016 17 comments
Deep learning frameworks are limited

conv4d and higher dimension generalization #1661

Closed  jerabaul29 opened this issue on **Mar 26, 2016**  17 comments

girving commented on **Jun 16, 2017**

*I think this is unlikely to happen in a way that makes anyone happy, so closing. All applications of 4D convs that I know of will almost certainly be forced to use special factorizations for efficiency.*

Closed  girving closed this on **Jun 16, 2017**
OK, Deep Learning has outlived its usefulness as a buzz-phrase. Deep Learning est mort. Vive Differentiable Programming!

Yeah, Differentiable Programming is little more than a rebranding of the modern collection of Deep Learning techniques, the same way Deep Learning was a rebranding of the modern incarnations of neural nets with more than two layers.

But the important point is that people are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization.

An increasingly large number of people are defining the networks procedurally in a data-dependent way (with loops and conditionals), allowing them to change dynamically as a function of the input data fed to them. It's really very much like a regular program, except it's parameterized, automatically differentiated, and trainable/optimizable. Dynamic networks have become increasingly popular (particularly for NLP), thanks to deep learning frameworks that can handle them such as PyTorch and Chainer (note: our old deep learning framework Lush could handle a particular kind of dynamic nets called Graph Transformer Networks, back in 1994. It was needed for text recognition).

People are now actively working on compilers for imperative differentiable programming languages. This is a very exciting avenue for the development of learning-based AI.

Important note: this won't be sufficient to take us to "true" AI. Other concepts will be needed for that, such as what I used to call predictive learning and now decided to call Imputative Learning. More on this later....

https://www.facebook.com/yann.lecun/posts/10155003011462143
We want to use general programs as our differentiable model.

```python
def f(x):
    for i:
        ...
        y = f(g(y))
    if ...:
        return ...
    else:
        return ...
```

Parameters

\[ \frac{\partial}{\partial} \text{objective} \]
Example: inverse rendering

3D scene:
- triangle positions
- camera pose
- materials

...
Example: inverse rendering
Example: inverse rendering

The Irishman @ ILM
Example: inverse rendering
Example: robotics

Xu, Kim, Chen, Rodriguez, Agrawal, Matusik, Sueda, 2022
Example: robotics
Example: Computer-Aided-Design

overheard: Dyson heavily used differentiable programming over fluid dynamics solvers to optimize their products

Simulation of airplane frame, directly on CAD representation.

Asymmetric wheel... unstable spin
Optimized wheel... spins stably

Inverse shape design under gravity: let there be light!

Hafner, Schumacher, Knoop, Auzinger, Bickel, Bacher, 2019
Example: climate modeling
Example: fundamental physics

astrophysics, high energy physics, molecular dynamics, …

Forecasting the power of Higher Order Weak Lensing Statistics with automatically differentiable simulations

Denise Lanzieri1,*, François Lanusse2, Chirag Modi3, Benjamin Horowitz4,5, Joachim Harnois-Déraps6, Jean-Luc Starck2, and The LSST Dark Energy Science Collaboration (LSST DESC)

New directions for surrogate models and differentiable programming for High Energy Physics detector simulation
Anything from you?
Deep learning systems are not designed for non-deep-learning workload.

Deep learning systems assume each “layer” has high arithmetic intensity and the memory cost between layers is negligible.
def ray_triangle_isect(rays, tris):
    e1 = tris[1:2, :] - tris[0:1, :]
    e2 = tris[2:3, :] - tris[0:1, :]
    s1 = torch.cross(rays[3:, :], e2)
    div = torch.dot(s1, e1)
    mask = torch.where(div != 0,
                       torch.ones_like(div),
                       torch.zeros_like(div))
    inv_div = 1 / div
    s = rays[:3, :]
    u = torch.dot(s, s1) * inv_div
    s2 = torch.cross(s, e1)
    v = torch.dot(rays[3:, :], s2) * inv_div
    t = torch.dot(e2, s2) * inv_div
    mask *= torch.where(t > 0 and 
                        u >= 0 and v >= 0 and u + v <= 1,
                        torch.ones_like(mask),
                        torch.zeros_like(mask))
    return mask, u, v, t
Deep learning systems are not designed for non-deep-learning workload.

```python
def ray_triangle_isect(rays, tris):
    e1 = tris[1:2, :] - tris[0:1, :]
    e2 = tris[2:3, :] - tris[0:1, :]
    s1 = torch.cross(rays[3:, :], e2)
    div = torch.dot(s1, e1)
    mask = torch.where(div != 0,
        torch.ones_like(div),
        torch.zeros_like(div))
    inv_div = 1 / div
    s = rays[:3, :]
    u = torch.dot(s, s1) * inv_div
    s2 = torch.cross(s, e1)
    v = torch.dot(rays[3:, :], s2) * inv_div
    t = torch.dot(e2, s2) * inv_div
    mask *= torch.where(t > 0 and 
        u >= 0 and v >= 0 and u + v <= 1,
        torch.ones_like(mask),
        torch.zeros_like(mask))
    return mask, u, v, t
```

A basic ray triangle intersection routine.

Many low arithmetic intensity operations, leading to very high memory cost (if we don’t fuse them)
Deep learning systems are not designed for non-deep-learning workload

```python
def ray_triangle_isect(rays, tris):
    e1 = tris[1:2, :] - tris[0:1, :]
    e2 = tris[2:3, :] - tris[0:1, :]
    s1 = torch.cross(rays[3:, :], e2)
    div = torch.dot(s1, e1)
    mask = torch.where(div != 0,
                       torch.ones_like(div),
                       torch.zeros_like(div))
    inv_div = 1 / div
    s = rays[:3, :]
    u = torch.dot(s, s1) * inv_div
    s2 = torch.cross(s, e1)
    v = torch.dot(rays[3:, :], s2) * inv_div
    t = torch.dot(e2, s2) * inv_div
    mask *= torch.where(t > 0 and \
                        u >= 0 and v >= 0 and u + v <= 1,
                        torch.ones_like(mask),
                        torch.zeros_like(mask))
    return mask, u, v, t
```

A basic ray triangle intersection routine

Branching is extremely tedious for the programmer
We need other differentiable programming systems for differentiating non-deep-learning programs using a classical technique: automatic differentiation.

def f(x):
    for i:
        ...
        y = f(g(y))
    if ...:
        return ...
    else:
        return ...
If you do well in this class, you will know how to build compilers that generate derivative code 100x faster than deep learning systems in certain cases.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Forward Time</th>
<th>Backward Time</th>
<th>Total Time</th>
<th># Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TensorFlow</td>
<td>13.20 ms</td>
<td>35.70 ms</td>
<td>48.90 ms (188×)</td>
<td>190</td>
</tr>
<tr>
<td>CUDA</td>
<td>0.10 ms</td>
<td>0.14 ms</td>
<td>0.24 ms (0.92×)</td>
<td>460</td>
</tr>
<tr>
<td>DiffTaichi</td>
<td>0.11 ms</td>
<td>0.15 ms</td>
<td>0.26 ms (1.00×)</td>
<td>110</td>
</tr>
</tbody>
</table>

Li, Gharbi, Adams, Durand, Ragan-Kelley, 2018
If you do well in this class, you will know how to formulate derivatives of seemingly non-differentiable things.

\[
\partial \text{sort}([2.3, 3.1, 8.9, 5.0, -7.1])
\]

\[
\partial (l_1 \lor l_2 \lor x_2) \land \\
(\neg x_2 \lor l_3 \lor x_3) \land \\
(\neg x_3 \lor l_4 \lor x_4) \land \ldots \land \\
(\neg x_{n-3} \lor l_{n-2} \lor x_{n-2}) \land \\
(\neg x_{n-2} \lor l_{n-1} \lor l_n)
\]

\[
\partial \text{if } \ldots : \\
\partial \text{else } \ldots : \\
\partial \text{argmin}_x f(x)
\]
Logistics

• three 20% homeworks (+1 not graded), one 40% final project

  • late penalty: score \* \left( 1 - \frac{\text{seconds passed since deadline midnight}}{86400} \right)

• TA: Trevor Hedstrom (tjhedstr@ucsd.edu)

• My office hour: Monday 11am-11:59am CSE 4116

• TA’s office hour: Friday 2pm-3pm B275

• Discussion & announcements: Piazza (https://piazza.com/ucsd/spring2024/cse291)
Prerequisites

• Python

• a little bit of parallel programming

• a little bit of compiler

• a little bit of calculus
Readings (optional)

- Evaluating Derivatives
  Principles and Techniques of Algorithmic Differentiation
  Second Edition

- The Art of Differentiating Computer Programs
  An Introduction to Algorithmic Differentiation

The Elements of Differentiable Programming

Mathieu Blondel
Google DeepMind
mblondel@google.com

Vincent Roulet
Google DeepMind
vroulet@google.com
The loma language & compiler

throughout the class, we will modify the loma compiler (written in Python, generate C/ISPC/OpenCL code) step-by-step to implement automatic differentiation

https://github.com/BachiLi/loma_public

def sum_array(arr : In[Array[float]], arr_size : In[int]) -> float:
    i : int = 0
    s : float = 0.0
    while (i < arr_size, max_iter := 1000):
        s = s + arr[i]
        i = i + 1
    s_relu : float = 0.0
    if s > 0:
        s_relu = s
    return s_relu

d_sum_array = rev_diff(sum_array)
Homework 0 (not graded)

UCSD CSE 291 (Differentiable Programming) Assignment 0:

Introduction to the loma Programming Language

In this course, we will be using a C-like parallel programming language loma (invented for this course!). Our ultimate goal is to make loma “differentiable” and capable of automatically generating the derivatives of its code, and then use loma to implement your final project. In this handout, we will introduce the language and its compiler.

The design principles of loma are:

- **Minimal.** To make implementing the compiler and the differentiation as simple as possible, we restrict as much language features as possible. As a result, the language lacks advanced object-oriented/functional features such as higher-order functions, subtyping or fancy type inference.

- **Differentiable.** Furthermore, to make differentiation possible/simpler/efficient, we impose restrictions to the language. For example, functions can only have one `return` statement in the end, all `while` loops need to have a static upper bound on the iteration size, recursion is not allowed, and there is no pointer. These restrictions will make more sense as you start to implement differentiation. As a result, loma is not Turing complete and can be seen as a domain-specific language – this is a good thing as we can specialize the compiler to generate efficient code.
Homework 1

UCSD CSE 291 (Differentiable Programming) Assignment 1:
Forward mode automatic differentiation

1 Total derivatives

In our first homework, we will implement what is known as the “forward mode” automatic differentiation in Julia for generating code that computes the derivatives. Specifically, given a (differentiable) function $f : \mathbb{R}^n \to \mathbb{R}^m$, we will automatically generate its total derivative $df : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m$. The total derivative $df(x, dx)$ (noticed that it takes two inputs $x$ and $dx$ instead of only one!) is defined as the best linear approximation at point $x$:

$$\lim_{dx \to 0} \frac{\|f(x + dx) - (f(x) + df(x, dx))\|}{\|dx\|} = 0, \quad (1)$$

where $df(x, dx)$ is linear over the $dx$ argument (but not necessarily linear over the $x$ argument).

We can get the partial or directional derivatives that we are more used to from the total derivative function $df$. For example, we can apply a “one-hot” vector $v_i$ for computing the partial derivative of the $i$-th component of the input:

$$\frac{\partial}{\partial x_i} f(x) = df(x, v_i), v_{i,j} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise}. \end{cases} \quad (2)$$

For notational convenience, we further define a function $Df : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^m \times \mathbb{R}^m$ that combines the outputs of function $f$ and its total derivative $df$:

$$Df(x, dx) = (f(x), df(x, dx)). \quad (3)$$
Homework 2

UCSD CSE 291 (Differentiable Programming) Assignment 2:
Reverse mode automatic differentiation

1 Getting gradients by running the function backwards.

In the second homework, we will implement what’s called the reverse-mode automatic differentiation on
straight-line code. It is equivalent (or can be seen as a generalization) of the popular backpropagation
algorithm in deep learning. Before we talk about reverse mode, let’s talk about why we need something
other than the forward mode we implemented last time.

Consider a function $f(x)$ that takes many numbers and outputs a scalar $f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$, where $n$ can
be a large number (say, a million). Suppose we want to compute the gradient of $f$ (say, for optimizing
or sampling $f$ using gradient-based methods). Applying forward mode differentiation to $f$ would give us a
function $Df(x, dx) : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R} \times \mathbb{R}$. Since $Df$ can only output one derivative at a time, we need to feed
in a different “one-hot” $dx$ (where one component is 1 and all others are 0) in order to get the full gradient vector – this is too inefficient. Instead, reverse mode gives us a function $D^T f(x, dy) : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R}^n$ that can
give us the full gradient vector at once if we set $dy = 1$.

How is this possible? Let’s use the same example we used for forward mode:

```python
def f(x: float, y: float) -> float:
    z0 = h(x, y)
    z1 = g(z0)
    return z1
```

Recall that the forward mode transform turns the function into their “$Df$” version.

```python
class _dfloat:
    val : float
dval : float

def Df(x : _dfloat, y : _dfloat) -> _dfloat:
    z0 : _dfloat = h(x, y)
    z1 : _dfloat = g(z0)
    return z1
```

In reverse mode, we want to turn $f$ into its $Df$ version. The function signature of $Df$ is:

```python
def Dff(x : float, _dx : Out[float], y : float, _dy : Out[float], dz1 : Out[float])
```
Homework 3

UCSD CSE 291 (Differentiable Programming) Assignment 3:
Handling control flow and function calls

Now we are going to extend our previous implementations of forward and reverse modes to handle if/else statements, general function calls, and while loops. We will dive into each case immediately! As usual, use hw_tests/hw3/test.py to test your implementation.

1 Handling if/else statements

Handling if/else statements in both forward and reverse modes are as simple as passing down the then and else statements and process them independently in the mutate_ifelse method.

**Forward mode.** For example, the following code

```python
def ifelse(x : In[float], y : In[float]) -> float:
    ret : float
    if y > 0.0:
        ret = 5.0 * x
    else:
        ret = 2.0 * x
    return ret

fwd_ifelse = fwd_diff(ifelse)
```

would be transformed into something like:

```python
def fwd_ifelse(x : In[_dfloat], y : In[_dfloat]):
    ret : _dfloat
    if y.val > 0.0:
        ret = make_dfloat(5.0 * x.val, 0.0 * x.val + 5.0 * x.dval
```
Final Project

- handout in preparation...

- You will differentiate some algorithms of your choice using loma. Extend the compiler if necessary.
Course overview/syllabus (very tentative)

https://cseweb.ucsd.edu/~tzli/cse291/sp2024/
If you do well in this class, you will understand how deep learning frameworks are made.
If you do well in this class, you will know how to build compilers that generate derivative code 100x faster than deep learning systems in certain cases.
If you do well in this class, you will know how to formulate derivatives of seemingly non-differentiable things.

\[ \partial \text{sort}([2.3, 3.1, 8.9, 5.0, -7.1]) \]

\[ \partial (l_1 \lor l_2 \lor x_2) \land 
(\neg x_2 \lor l_3 \lor x_3) \land 
(\neg x_3 \lor l_4 \lor x_4) \land \ldots \land 
(\neg x_{n-3} \lor l_{n-2} \lor x_{n-2}) \land 
(\neg x_{n-2} \lor l_{n-1} \lor l_n) \]

\[ \partial \text{if} \ldots : \text{else} \ldots : \partial \text{argmin}_x f(x) \]