Efficient differentiable programming

November 27, 2019

Outline

intro

abstract

- an AD system for higher order functional array-processing language.
- supports both source-to-source forward mode AD and global optimizations
- gradient computation with forward AD can be as efficient as reverse mode
- Jacobian matrices needed for practical numerical schemes can be efficiently computed

The key contribution is in making traditional forward mode AD efficient through a number of compile time optimizing transformations, e.g. fusion

AD in general

Two main mode for AD:

- forward mode computes the derivative of the original computation while making a forward pass (across the computational graph)
- Reverse mode does a forward pass and then a backwards one to compute the adjoint derivatives.

For reverse mode we associate to every variable

$$v_i$$

its adjoint

$$\frac{\partial y}{\partial v_i}$$

AD in ML

We want a constant time computation of differentiation as an important assumption in algorithms is often that the gradient of a function has the same asymptotic computational cost as the original function.
Most algorithms in optimization have their computational complexity defined in the number of calls to an *oracle*. In first order methods the oracle is usually either gradient, a subgradient, or maybe some funky proximal operator.

ML

supervised learning setup

AI is pretty simple nowadays, it's all machine learning, which means given a dataset of N samples of the form

$$\{(x_1, y_1), \ldots, (x_N, y_N)\}$$

where the x's are the *feature vectors* and the y's the result of the prediction, we wish to find a good predictor, i.e. a function

$$g : X \to Y$$

.

To define the goodness of a predictor we define a *loss function*

$$f : Y \times Y \to \mathbb{R}$$

Define the empirical risk minimization has the average of all the losses over the sample. Then the best predictor is the one that minimizes the empirical risk.

jacobian

- In general if

$$f : \mathbb{R}^n \to \mathbb{R}^m$$

we define the Jacobian the following way:

$$J_f = \frac{\partial f}{\partial x} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{1}$$

Forward mode computes the Jacobian through columns whereas reverse mode computes rows.
Forward mode is efficient when there are many more rows than columns and vice versa for reverse mode. Reverse mode is thus good for very large dimensions in the input. Choosing the optimal mix between the two is NP complete.

remarks on the jacobian

- When computing the matrix there are many loop invariant computations, this calls for *loop-invariant code motion*
- Even if the asymptotic computational complexity of a gradient call is the same as one for the original function the hidden constant can be quite big in practice. Many intermediate vectors can be removed through deforestation aka loop fusion.

Pipeline

$$\begin{tikzcd}
\widetilde{\mathrm{M}} \text{ / MATLAB / NumPy} \arrow[r, "\text{ADiMat [Bischof et al. 2002] /}", "\text{Autograd [Maclaurin et al. 2015]}"'] \arrow[d, "\text{Lowering (Section 3.2)}"] & \text{Diff. } \widetilde{\mathrm{M}} \text{ / MATLAB / NumPy} \arrow[d, "\text{Lowering}"'] \\
\widetilde{\mathrm{F}} \text{ (Section 3.1) / F\#} \arrow[r, "d\widetilde{\mathrm{F}} \text{ (Section 4.2) /}", "\text{DiffSharp [Baydin et al. 2015a]}"'] \arrow[d, "\text{DPS [Shaikhha et al. 2017]}"] & \text{Diff. } \widetilde{\mathrm{F}} \text{/ F\#} \arrow[d, "\text{DPS [Shaikhha et al. 2017]}"'] \\
\text{C / C++} \arrow[r, "\text{Tapenade [Hascoet and Pascual 2013] /}", "\text{ADIC [Narayanan et al. 2010]}"'] & \text{Diff. C / C++}
\end{tikzcd}$$

d f-smooth and M-smooth

- The highest level /api/ mimicks the conventional interfaces to high level linear algebra libraries and languages such as MatLab and Numpy
- The program is then lowered into its representation in f smooth which computes the actual derivatives. After this computation many optimizations can be applied before...
- Compiling once again to C code, where memory is managed in a stack like manner by using the destination passing style technique, which is just a fancy way to say that the caller is responsible for the callees memory and passes to it the memory location where it can store the results.

differentiation in df-smooth

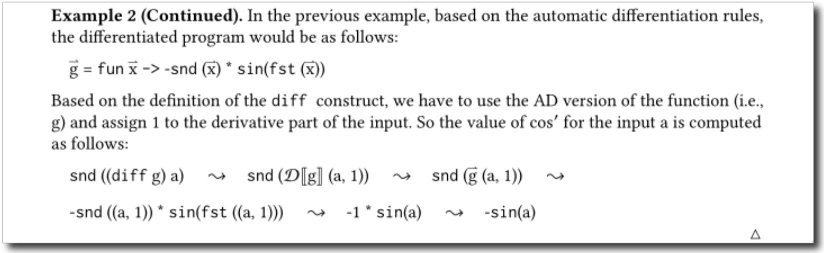
- source to source transformation based on the `deriv` construct
- everything done at compile time, macro-ish
- we can call `deriv` multiple times to get the derivative with respect to a list of free variables.

1. first deriv constructs a lambda with the free variables as input parameters
2. the produced lambda is passed to the source-to-source differentiation operator $\mathcal{D}$

example

The derivative of the program `cos a` is represented as $\texttt{snd}((\mathcal{D}[[\texttt{fun a -> cos a}]])(a, 1))$ which expands to

$$\texttt{snd}((\mathcal{D}[[\texttt{fun a -> cos a}]](a, 1)))$$



Example 2 (Continued). In the previous example, based on the automatic differentiation rules, the differentiated program would be as follows:

$\vec{\text{g}}$ = fun $\vec{\text{x}}$ -> -snd ($\vec{\text{x}}$) * sin(fst ($\vec{\text{x}}$))

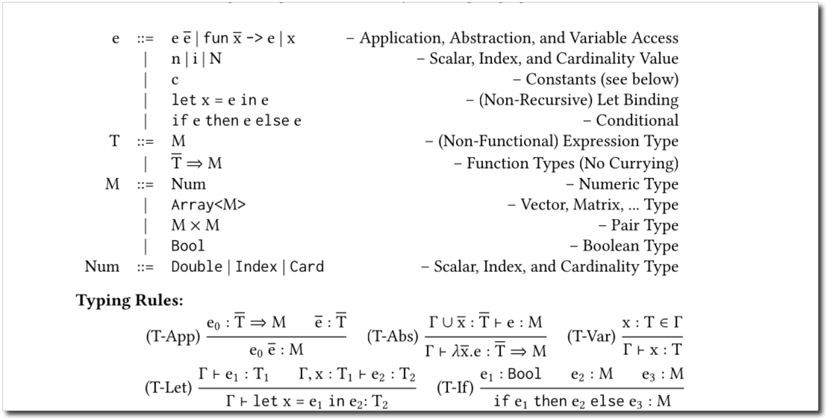
Based on the definition of the `diff` construct, we have to use the AD version of the function (i.e., $\vec{\text{g}}$) and assign 1 to the derivative part of the input. So the value of cos′ for the input a is computed as follows:

snd ((diff g) a) $\rightsquigarrow$ snd ($\mathcal{D}$[[g]] (a, 1)) $\rightsquigarrow$ snd ($\vec{\text{g}}$ (a, 1)) $\rightsquigarrow$

-snd ((a, 1)) * sin(fst ((a, 1))) $\rightsquigarrow$ -1 * sin(a) $\rightsquigarrow$ -sin(a)

△

f smooth



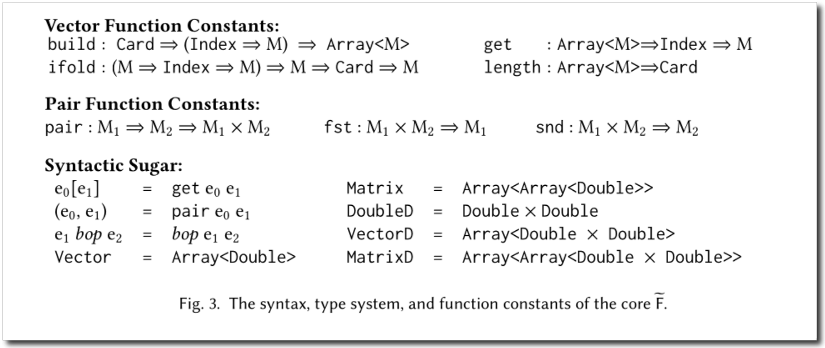
e	::=	$e\ \overline{e}$ | fun $\overline{x}$ -> e | x	– Application, Abstraction, and Variable Access
	|	n | i | N	– Scalar, Index, and Cardinality Value
	|	c	– Constants (see below)
	|	let x = e in e	– (Non-Recursive) Let Binding
	|	if e then e else e	– Conditional
T	::=	M	– (Non-Functional) Expression Type
	|	$\overline{T} \Rightarrow M$	– Function Types (No Currying)
M	::=	Num	– Numeric Type
	|	Array<M>	– Vector, Matrix, ... Type
	|	$M \times M$	– Pair Type
	|	Bool	– Boolean Type
Num	::=	Double | Index | Card	– Scalar, Index, and Cardinality Type

Typing Rules:

$$\text{(T-App)}\ \frac{e_0 : \overline{T} \Rightarrow M \qquad \overline{e} : \overline{T}}{e_0\ \overline{e} : M} \qquad \text{(T-Abs)}\ \frac{\Gamma \cup \overline{x} : \overline{T} \vdash e : M}{\Gamma \vdash \lambda\overline{x}.e : \overline{T} \Rightarrow M} \qquad \text{(T-Var)}\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

$$\text{(T-Let)}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : T_2} \qquad \text{(T-If)}\ \frac{e_1 : \texttt{Bool} \qquad e_2 : M \qquad e_3 : M}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : M}$$

f smooth 2



Vector Function Constants:

build : Card $\Rightarrow$ (Index $\Rightarrow$ M) $\Rightarrow$ Array<M>   get : Array<M>$\Rightarrow$Index $\Rightarrow$ M

ifold : (M $\Rightarrow$ Index $\Rightarrow$ M) $\Rightarrow$ M $\Rightarrow$ Card $\Rightarrow$ M   length : Array<M>$\Rightarrow$Card

Pair Function Constants:

pair : $M_1 \Rightarrow M_2 \Rightarrow M_1 \times M_2$ fst : $M_1 \times M_2 \Rightarrow M_1$ snd : $M_1 \times M_2 \Rightarrow M_2$

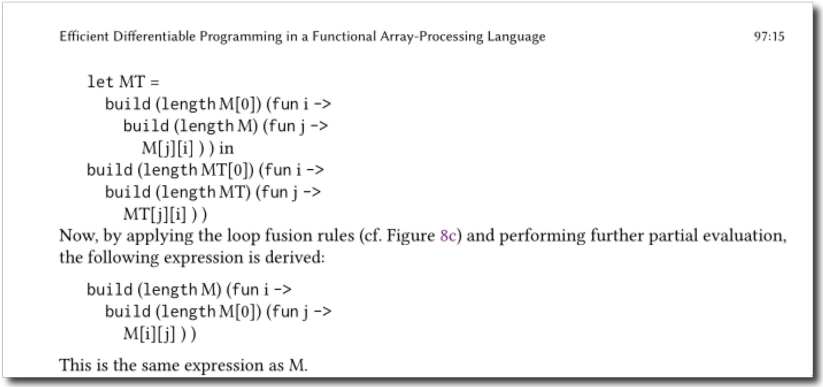
Syntactic Sugar:

$e_0[e_1]$	=	get e_0 e_1	Matrix	=	Array<Array<Double>>
(e_0, e_1)	=	pair e_0 e_1	DoubleD	=	Double × Double
e_1 *bop* e_2	=	*bop* e_1 e_2	VectorD	=	Array<Double × Double>
Vector	=	Array<Double>	MatrixD	=	Array<Array<Double × Double>>

Fig. 3. The syntax, type system, and function constants of the core $\widetilde{\text{F}}$.

built in functions for array programming : build ifold and length

loop fusion example



```
let MT =
  build (length M[0]) (fun i ->
    build (length M) (fun j ->
      M[j][i] ) ) in
build (length MT[0]) (fun i ->
  build (length MT) (fun j ->
    MT[j][i] ) )
```

Now, by applying the loop fusion rules (cf. Figure 8c) and performing further partial evaluation, the following expression is derived:

```
build (length M) (fun i ->
  build (length M[0]) (fun j ->
    M[i][j] ) )
```

This is the same expression as M.