Report on Automatic Differentiation and Implementation of its reverse mode in Haskell

Heng Wei

Introduction

The fields of machine learning and automatic differentiation (AD) have only recently converged. One of the main ways neural networks "learn" is by updating its numerous parameters with a procedure called Stochastic Gradient Descent. SGD requires the computation of gradients, hence the relevance of AD in machine learning. This review revisits the mathematical and technical foundations of AD, different ways to implement it in functional programming, and lastly explores some interesting areas of active research (policy gradients, randomized automatic differentiation, etc).

1 Background [1]

Other than automatic differentiation, there are 3 methods of computing the derivative in computer programs:

- Manually working out the derivative and coding them
- Numerical differentiation using finite difference approximations
- Symbolic differentiation using expression manipulation

Manual differentiation involves manually deriving and coding the derivative of functions. This method is impractical for complex models and is prone to human error. Numerical differentiation is inexact, and research has shown that even the most advanced numerical techniques face a tradeoff between complexity and approximation error. In deep learning, where n, the number of weights, is often astronomically large, the best-case complexity of $O(n)$ for numerical methods makes them impractical. Symbolic differentiation, on the other hand, suffers

from the issue of "expression swell." Additionally, both manual and symbolic differentiation require the function to be in a closed form, limiting the algorithmic control flow and expressiveness of the functions we aim to differentiate. These issues have significantly hindered research in machine learning until the discovery of automatic differentiation (AD) by the machine learning community.

2 Automatic Differentiation (AD) [1]

2.1 Overview

Mathematically, we are looking to compute the partial derivative of a function w.r.t. a certain parameter, by accumulating values of the partial derivatives at each step (computed via symbolic differentiation) to form the value of the final partial derivative we are looking for. The problem of "expression swell" is mitigated because the derivative of each sub-functions is evaluated and passed on. In other words, we do not output the expression of the derivative, but rather its exact value. By interleaving as much as possible our computations (reverse mode AD), the technique becomes computationally feasible even for mind-boggling big models (AD has ideal asymptotic efficiency and only a small constant overhead). As a technical term, AD refers to a specific family of techniques to help us achieve this.
Note: the sub-functions are often elementary functions with known derivatives (hence why we use symbolic differentiation). Furthermore we know that all numerical computations are ultimately composed of a finite set of elementary operations for which the derivative is known (Verma, 2000).

2.2 Forward Mode AD

Let a model $f : \mathbb{R}^n \to \mathbb{R}^m$ be a composition of many elementary operations/nodes. In forward mode AD, we define a forward pass as feeding the model an input $\mathbf{x} = \mathbf{a}$, and then computing and storing the two things below as we follow the computation graph starting from the input all the way to the output:

1. The function evaluation at each node, v_i. This is called the forward primal trace. Each node is an elementary operation for which the derivative is known.

2. The derivative evaluation at each node, v_i'. This is called the corresponding tangent/derivative trace. The derivative evaluation can be done w.r.t. any x_i, the i-th component of $\mathbf{x}$, or any intermediary variables v_i.

Together, they form a list of tuples (v, v'), called a dual number.

Mathematically, we express a dual number as $v + (v')\epsilon$, where ϵ is a nilpotent number ($\epsilon^2 = 0, \epsilon \neq 0$). By setting up a regime where $h(v + v') = h(v) + h(v')v'\epsilon$, with h being any node in our computation graph, we obtain iteratively the corresponding dual number. In more technical terms, we can set up this regime by taking the source code of a certain model E, and change it so that it handles the dual operation of simultaneously computing v and v'. Another way is through operator overloading.

In general, forward AD can be applied to functions $f : \mathbb{R}^n \to \mathbb{R}^m$. By feeding it an input $\mathbf{x} = \mathbf{a}$, we have $\mathbf{x}' = \mathbf{e_i}$, therefore one forward pass gives us the i-th column of the Jacobian of f:

$$\begin{bmatrix} \frac{\partial y_1}{\partial x_i} \\ \vdots \\ \frac{\partial y_m}{\partial x_i} \end{bmatrix}$$

Even more generally, if we initialize $\mathbf{x}' = \mathbf{r}$, one forward pass computes efficiently the Jacobian-vector product (JVP),

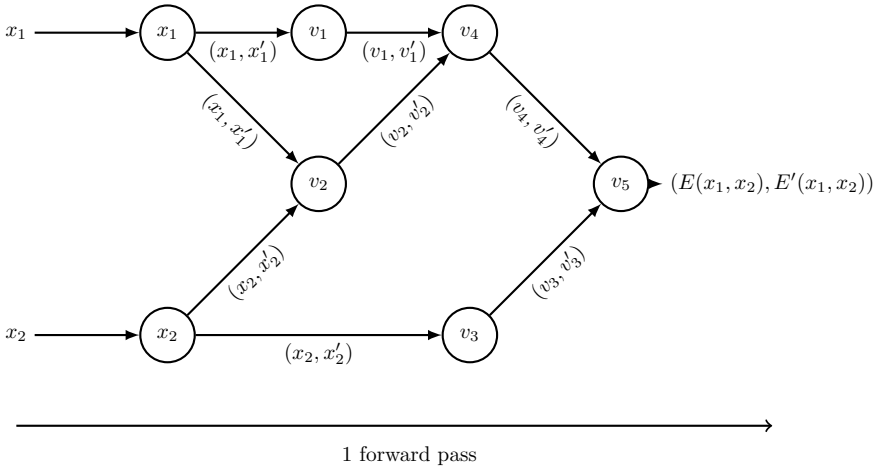
$$J_f \mathbf{r}\Big|_{\mathbf{x}=\mathbf{a}}$$

without explicitly constructing the Jacobian nor performing any matrix operations.

$$J_f \mathbf{r} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} r_1 \\ \vdots \\ r_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{n} \frac{\partial y_1}{\partial x_j} r_j \\ \vdots \\ \sum_{j=1}^{n} \frac{\partial y_m}{\partial x_j} r_j \end{bmatrix}$$

Important note: In the context of deep learning, after feeding the input, the entire neural network is defined as $E : \mathbb{R}^n \to \mathbb{R}$, where n is the number of intermediary variables or weights, not the dimension of the input. Therefore, after one pass we would have $\frac{\partial E}{\partial v_i}$. The issue here is that we must compute n passes to obtain all our partial derivatives ($\nabla E = J_E = \begin{bmatrix} \frac{\partial E}{\partial v_1} & \frac{\partial E}{\partial v_2} & \cdots & \frac{\partial E}{\partial v_n} \end{bmatrix}^T$). In deep learning, n is often astronomically big.

For example, a forward pass in a neural network with 4 intermediary variables, no matter which v_i we are deriving w.r.t., would look like this:



Note: for any node v_i that serves as input to a node v_j, we consider $\frac{\partial v_i}{\partial v_j} = 0$, and not $\frac{\partial v_i}{\partial v_j} = \frac{\partial v_j}{\partial v_i}^{-1}$, because updating a node in our graph does not affect the nodes that serves as its input. In other words, if $v_j = h_i(v_i)$, we do not interpret $v_i = h_i^{-1}(v_j)$.

2.3 Reverse Mode AD

Reverse mode AD, also known as backpropagation, is particularly useful for neural networks and large-scale models. A round in reverse mode is composed of two passes. The first is a forward pass that builds only the forward primal trace, i.e. populates all intermediary variables v_i. The second is a backward pass, also known as backpropagation, starting from the output and going back towards the input, that builds the reverse adjoint trace. The ajoint is defined as the partial derivative of the final output y_i by an intermediary variable:

$$\bar{v}_i = \frac{\partial y_j}{\partial v_i}$$

In more technical terms, the forward pass builds a computational graph where each node represents an elementary operation and edges represent dependencies/weights between these operations. Inputs flow forward through the graph to compute the final output. Each node typically stores both the value of the operation and references to any nodes that contribute inputs to the operation.

During the backward pass, the gradients are computed with respect to each operation in reverse order of execution (from output back to inputs). As the computation moves backwards, gradients of the output with respect to intermediate variables and inputs are accumulated. For each node, the gradient of the node's output value is used to compute the gradients of the inputs to that node. A key technique in reverse mode AD is the use of a "tape" data structure. During the forward pass, all the operations performed (along with necessary values and contextual information) are recorded (or "taped") sequentially. During the backward pass, this tape is traversed in reverse order to apply the chain rule and calculate derivatives. This is why reverse mode AD is sometimes referred to as "tape-based" AD.

In general, reverse mode AD can be applied to functions $f : \mathbb{R}^n \to \mathbb{R}^m$. By initializing the backpropagation step with $\bar{\mathbf{y}} = \mathbf{e_i}$, a reverse round (composed of a forward pass and backpropagation) gives us i-th row of the Jacobian of f:

$$\begin{bmatrix} \frac{\partial y_i}{\partial x_1} & \frac{\partial y_i}{\partial x_2} & \cdots & \frac{\partial y_i}{\partial x_n} \end{bmatrix}$$

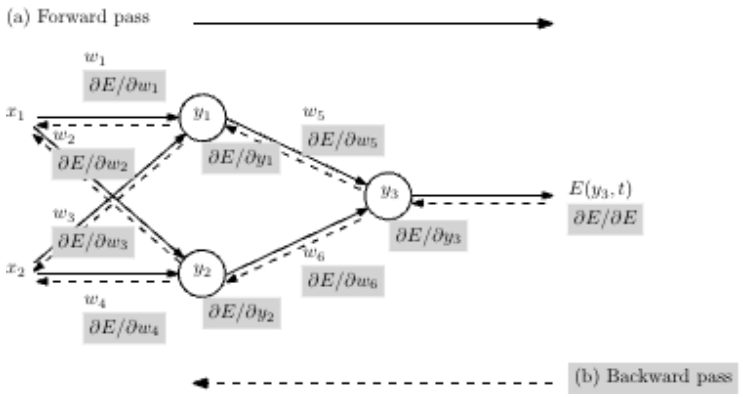
Even more generally, if we initialize the backpropagation step with $\bar{\mathbf{y}} = \mathbf{r}$, one reverse pass computes efficiently the vector-Jacobian product (VJP),

$$\mathbf{r}^T J_f \Big|_{\mathbf{x}=\mathbf{a}}$$

without explicitly constructing the Jacobian nor performing any matrix operations.

$$\mathbf{r}^T J_f = \begin{bmatrix} r_1 & \cdots & r_m \end{bmatrix} \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^m \frac{\partial y_j}{\partial x_1} r_j \\ \vdots \\ \sum_{j=1}^m \frac{\partial y_j}{\partial x_n} r_j \end{bmatrix}$$

Important note: In the context of deep learning, contrary to forward mode that requires n forward passes to compute the gradient (assuming $E : \mathbb{R}^n \to \mathbb{R}$), reverse mode does it in a single reverse round. After the forward pass the entire neural network is defined as $E : \mathbb{R}^k \to \mathbb{R}$, where k is the number of weights. To obtain all our partial derivatives ($\nabla E = J_E = \begin{bmatrix} \frac{\partial E}{\partial w_1} & \frac{\partial E}{\partial w_2} & \cdots & \frac{\partial E}{\partial w_k} \end{bmatrix}^T$), we start the backpropagation step with $\frac{\partial E}{\partial E} = 1$. With the chain rule of total derivatives, we store our partial derivatives and accumulate them. For example, a reverse pass in a neural network with 6 weights would look like this:



3 Implementation of Reverse Mode AD in Haskell

Attempt 1: Starting with curried functions

Suppose the user defines a function f in Haskell like this:

```
g :: Double -> Double
g x = x**2
f :: Double -> Double -> Double
f x y = g x + y
```

In order to differentiate a function f using reverse-mode automatic differentiation (AD), we need a program that can generate a computation graph from the function declaration itself. This graph is crucial for computing the gradient during backpropagation. However, Haskell presents a unique challenge: it does not support introspection, meaning you cannot inspect the implementation or structure of a function after it has been declared. This limitation is consistent with the abstraction principle in functional programming, where the details of *how* something is done are abstracted away from *what* is being done. In other words, in Haskell, you cannot directly observe or manipulate the code of a function to understand its computational steps. This poses a problem for differentiating the function, as we typically need to analyze the operations performed within the function to generate the necessary computation graph.

To overcome this challenge, we leverage Haskell's powerful type system and type classes to "reify" the function. Reification is the process of transforming the function into a form that is easier to manipulate and analyze for differentiation purposes. Specifically, we reify each argument of the function into a structure that keeps track of three key pieces of information:

1. **Value**: The actual value of the argument.
2. **Differentiation Context (Phantom Type s)**: A type that represents the context or scope of differentiation
3. **Accumulated Gradient**: The gradient that accumulates as we traverse the computation graph during backpropagation.

Here's a high-level overview of how we implement this strategy:

- **Custom Data Type (`BVar`)**: We define a data type `BVar` that encapsulates the value, gradient, and backpropagation function. This type plays a crucial role in tracking the necessary information as we reify the function.
- **Type Classes (`Reify`)**: We use type classes to generalize the reification process across different types, such as basic types like `Double` and functions with any number of scalar arguments (e.g., `Double -> ... -> Double`, but not `[Double] -> Double`). The `Reify` type class allows us to convert standard Haskell functions into their reified forms, which can be differentiated. `BVar` types instead of simple values. This transformation allows us to build the computation graph implicitly as the function is applied to its arguments.
- **Backpropagation**: Once the function is reified and evaluated, we perform backpropagation through the computation graph by leveraging the accumulated gradients stored in the `BVar` type.

At the time of writing, I have successfully implemented the reification process of functions with any number of scalar arguments (Double), but I have not successfully implemented the gradient accumulation process that works on any Haskell function type yet. I plan on finishing it before the presentation of my project. Below is a version of my code (to be updated).

```
{-# LANGUAGE GADTs #-}
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE TypeFamilies #-}
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE ScopedTypeVariables #-}
{-# LANGUAGE UndecidableInstances #-}

import Data.Proxy

-- Define the BVar type to hold both value and gradient, and a backpropagation f
data BVar s a = BVar
  { value :: a
```

```
  , grad :: a
  , backpropFn :: a -> a
  }

-- Add a Show instance for BVar
instance Show a => Show (BVar s a) where
  show (BVar val grad _) = "BVar " ++ show val ++ " " ++ show grad

-- Define the Reify type class
class Reify s a where
  type Reified s a :: *
  reify :: Proxy s -> a -> Reified s a
  unreify :: Proxy s -> Reified s a -> a

-- Reify instance for simple types (e.g., Double)
instance Reify s Double where
  type Reified s Double = BVar s Double
  reify _ x = BVar x 0 id
  unreify _ (BVar x _ _) = x

-- Reify instance for BVar itself
instance Reify s (BVar s a) where
  type Reified s (BVar s a) = BVar s a
  reify _ bvar = bvar
  unreify _ bvar = bvar

-- Reify instance for functions
instance (Reify s a, Reify s b) => Reify s (a -> b) where
  type Reified s (a -> b) = Reified s a -> Reified s b
  reify p f = reify p . f . unreify p
  unreify p f = unreify p . f . reify p

-- Backpropagate function
backprop :: (Reify s (a -> b), Reify s b) => Proxy s -> (a -> b) -> a -> Reified
backprop p f x =
  let reifiedF = reify p f
      reifiedX = reify p x
      result = reifiedF reifiedX
      gradX = backpropFn result 1
  in reifiedX { grad = gradX }

-- Test functions
g :: Double -> Double
g x = x ** 2

f :: Double -> Double
```

```
f x = x ** 2

f2 :: Double -> Double -> Double
f2 x y = x + y

h :: Double -> Double -> Double -> Double
h x y z = x * y + z

h2 :: Double -> Double -> Double -> Double
h2 x y z = x * y + g z

h3 :: Double -> Double -> Double
h3 x y = g (f2 x y)

main :: IO ()
main = do
--Tests
  let p = Proxy :: Proxy ()

  let bvar = backprop p f 2
  print $ grad bvar

  let reifiedG = reify p g
  print $ reifiedG (BVar 3 0 id)

  let reifiedF2 = reify p f2
  print $ reifiedF2 (BVar 2 0 id) (BVar 3 0 id)

  let reifiedH = reify p h
  print $ reifiedH (BVar 2 0 id) (BVar 3 0 id) (BVar 4 0 id)

  let reifiedH2 = reify p h2
  print $ reifiedH2 (BVar 2 0 id) (BVar 3 0 id) (BVar 4 0 id)

  let reifiedH3 = reify p h3
  print $ reifiedH3 (BVar 2 0 id) (BVar 3 0 id)
```

Attempt 2: Starting with functions of type `Num a => [a] → [a]`

In my second attempt at implementing Reverse Mode AD, I opted for a more generalized and flexible approach by focusing on functions of type `Num a => [a] -> [a]`. This approach allows me to leverage Haskell's powerful typeclass system (`Num`, `Fractional`, and `Floating`) to overload basic arithmetic

and mathematical operators, enabling them to operate on a custom data type `Dvar a`.

Key Concepts

The `Dvar` Data Structure: The `Dvar` data type encapsulates both the value of a variable and a `backprop` function that represents how changes in this variable propagate through the computational graph. The `backprop` function is crucial for accumulating gradients during backpropagation.

```
data Dvar a = Dvar { value :: a, backprop :: (a, Int) -> IO a }
```

Overloading Operators: By creating instances of the `Num`, `Fractional`, and `Floating` typeclasses for `Dvar a`, I allow mathematical operations to be automatically tracked and differentiated. Each operation (e.g., addition, multiplication, sine, cosine) is redefined to handle `Dvar` types, ensuring that the computational graph is built implicitly as the operations are performed.

```
instance Num a => Num (Dvar a) where
    (+) (Dvar x dx) (Dvar y dy) = Dvar (x + y) (\(dz, j) -> do
        dx_result <- dx (dz, j)
        dy_result <- dy (dz, j)
        return (dx_result + dy_result))
    ...
```

Implicit Computational Graph Construction: The computational graph is implicitly constructed through the chaining of `backprop` functions. When a `Dvar` is involved in an operation, its `backprop` function captures how that operation influences the gradient flow. When I eventually call `backprop` on the result `Dvar`, it triggers a series of backward passes through the graph, computing the gradients for each variable.

The `jac` Function: The `jac` function computes the Jacobian matrix of a function `f :: [Dvar a] -> [Dvar a]` by applying the function to a list of `Dvar`s initialized with the input values. The gradients are accumulated in an `IORef` and read out after backpropagation.

```
jac :: (Num a, ToList b a) => ([Dvar a] -> b) -> [a] -> IO [[a]]
jac f inputs = do
    let n = length inputs
```

```
        m = length (toList (f (map (\x -> Dvar x (\(_, _) -> return 0)) inputs)))
    jacobianRef <- newIORef (replicate m (replicate n 0))
    let dvars = zipWith (lift jacobianRef) [0..] inputs
        results = toList (f dvars)
    mapM_ (\(j, result) -> backprop result (1, j)) (zip [0..] results)
    readIORef jacobianRef
```

Gradient Accumulation: In this approach, the `backprop` function for each `Dvar` accumulates the gradient contributions from various paths in the computational graph. This design allows the Jacobian to be calculated efficiently for a wide variety of functions.

Advantages of This Approach

- **Flexibility**: By working with lists of `Dvar`s, this implementation is more general and can handle a broader range of functions compared to the initial curried function approach.
- **Scalability**: The method scales well to functions with any number of inputs and outputs.
- **Efficiency**: The implicit construction of the computational graph reduces the overhead of manually managing graph structures, leading to cleaner and potentially faster code.

Example Functions and Usage

```
main :: IO ()
main = do
    gradient <- jac f [2.0, 3.0, 1.0]
    print gradient
    gradient <- jac f2 [2.0]
    print gradient
    gradient <- jac f4 [5,1]
    print gradient
    gradient <- jac g2 [2,3]
    print gradient
    gradient <- jac h2 [2,3]
    print gradient
    gradient <- jac m2 [5,2]
    print gradient

    jacobian <- jac n [2.0, 3.0]
```

```
    print jacobian
```

The complete implementation:

```
{-# LANGUAGE FlexibleInstances #-}
{-# LANGUAGE FunctionalDependencies #-}
{-# LANGUAGE UndecidableInstances #-}

import Data.IORef
data Dvar a = Dvar { value :: a, backprop :: (a, Int) -> IO a }

instance Num a => Num (Dvar a) where
    (+) (Dvar x dx) (Dvar y dy) = Dvar (x + y) (\(dz, j) -> do
        dx_result <- dx (dz, j)
        dy_result <- dy (dz, j)
        return (dx_result + dy_result))
    (-) (Dvar x dx) (Dvar y dy) = Dvar (x - y) (\(dz, j) -> do
        dx_result <- dx (dz, j)
        dy_result <- dy (-dz, j)
        return (dx_result + dy_result))
    (*) (Dvar x dx) (Dvar y dy) = Dvar (x * y) (\(dz, j) -> do
        dx_result <- dx (y * dz, j)
        dy_result <- dy (x * dz, j)
        return (dx_result + dy_result))
    negate (Dvar x dx) = Dvar (negate x) (\(dz, j) -> do
        result <- dx (negate dz, j)
        return result)
    abs (Dvar x dx) = Dvar (abs x) (\(dz, j) -> dx (signum x * dz, j))
    signum (Dvar x dx) = Dvar (signum x) (\(_, _) -> return 0)
    fromInteger n = Dvar (fromInteger n) (\(_, _) -> return 0)

instance Fractional a => Fractional (Dvar a) where
    (/) (Dvar x dx) (Dvar y dy) = Dvar (x / y) (\(dz, j) -> do
        dx_result <- dx (dz / y, j)
        dy_result <- dy ((-x / (y * y)) * dz, j)
        return (dx_result + dy_result))
    fromRational r = Dvar (fromRational r) (\(_, _) -> return 0)

instance Floating a => Floating (Dvar a) where
    (**) (Dvar x dx) (Dvar y dy) = Dvar (x ** y) (\(dz, j) -> do
        dx_result <- dx (y * (x ** (y - 1)) * dz, j)
        dy_result <- dy ((x ** y) * log x * dz, j)
        return (dx_result + dy_result))
    pi = Dvar pi (\(_, _) -> return 0)
    exp (Dvar x dx) = Dvar (exp x) (\(dz, j) -> do
```

```
        result <- dx (exp x * dz, j)
        return result)
    log (Dvar x dx) = Dvar (log x) (\(dz, j) -> do
        result <- dx (dz / x, j)
        return result)
    sin (Dvar x dx) = Dvar (sin x) (\(dz, j) -> do
        result <- dx (cos x * dz, j)
        return result)
    cos (Dvar x dx) = Dvar (cos x) (\(dz, j) -> do
        result <- dx (-(sin x) * dz, j)
        return result)
    asin (Dvar x dx) = Dvar (asin x) (\(dz, j) -> do
        result <- dx (dz / sqrt(1 - x*x), j)
        return result)
    acos (Dvar x dx) = Dvar (acos x) (\(dz, j) -> do
        result <- dx (-dz / sqrt(1 - x*x), j)
        return result)
    atan (Dvar x dx) = Dvar (atan x) (\(dz, j) -> do
        result <- dx (dz / (1 + x*x), j)
        return result)
    sinh (Dvar x dx) = Dvar (sinh x) (\(dz, j) -> do
        result <- dx (cosh x * dz, j)
        return result)
    cosh (Dvar x dx) = Dvar (cosh x) (\(dz, j) -> do
        result <- dx (sinh x * dz, j)
        return result)
    asinh (Dvar x dx) = Dvar (asinh x) (\(dz, j) -> do
        result <- dx (dz / sqrt(x*x + 1), j)
        return result)
    acosh (Dvar x dx) = Dvar (acosh x) (\(dz, j) -> do
        result <- dx (dz / sqrt(x*x - 1), j)
        return result)
    atanh (Dvar x dx) = Dvar (atanh x) (\(dz, j) -> do
        result <- dx (dz / (1 - x*x), j)
        return result)

jac :: (Num a, ToList b a) => ([Dvar a] -> b) -> [a] -> IO [[a]]
jac f inputs = do
    let n = length inputs
        m = length (toList (f (map (\x -> Dvar x (\(_, _) -> return 0)) inputs)))
    jacobianRef <- newIORef (replicate m (replicate n 0))
    let dvars = zipWith (lift jacobianRef) [0..] inputs
        results = toList (f dvars)
    mapM_ (\(j, result) -> backprop result (1, j)) (zip [0..] results)
    readIORef jacobianRef
```

```
lift :: Num a => IORef [[a]] -> Int -> a -> Dvar a
lift jacobianRef i x = Dvar x (\(dz, j) -> do
    modifyIORef jacobianRef (\jac ->
        take j jac ++
        [take i (jac !! j) ++ [dz + ((jac !! j) !! i)] ++ drop (i+1) (jac !! j)] ++
        drop (j+1) jac)
    return dz)

class ToList a b | a -> b where
    toList :: a -> [Dvar b]

instance ToList (Dvar a) a where
    toList x = [x]

instance ToList [Dvar a] a where
    toList = id

f :: Floating a => [a] -> a
f [x, y, z] = x*x + x*y + sin z

f2 [x] = 2*x**3
f3 [x,y] = x+y
f4 [x,y] = f3 [x,y] + f2 [x]

c x = 2*x**3
c1 x y = x+y
c2 x y = c1 x y + c x

g [x,y] = y*sin x
g1 [x,y] = (g [x,y])**2
g2 [x,y] = (g1 [x,y]) + y

h [x] = sin x
h1 [x,y] = h [x] + y
h2 [x,y] = (h1 [x,y]) * y

m x y = x*y
m1 x y = (m x y) + cos y
m2 [x,y] = x*(m1 x y)

n :: Floating a => [a] -> [a]
n [x, y] = [x*x + y, sin x + y*y]


main :: IO ()
main = do
```

```
gradient <- jac f [2.0, 3.0, 1.0]
print gradient
gradient <- jac f2 [2.0]
print gradient
gradient <- jac f4 [5,1]
print gradient
gradient <- jac g2 [2,3]
print gradient
gradient <- jac h2 [2,3]
print gradient
gradient <- jac m2 [5,2]
print gradient

jacobian <- jac n [2.0, 3.0]
print jacobian
```

References

[1] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *The Journal of Machine Learning Research*, 18(153):1–43, 2018.