

# Instruction level automatic differentiation on a reversible Turing machine

Jin-Guo Liu<sup>1,\*</sup> and Taine Zhao<sup>2</sup>

<sup>1</sup>*Institute of Physics, Chinese Academy of Sciences, Beijing 100190, China*

<sup>2</sup>*Department of Computer Science, University of Tsukuba*

This paper considers the instruction-level adjoint mode automatic differentiation (AD). Here, instruction-level means, given the backward rules of basic instructions like  $+$ ,  $-$ ,  $*$  and  $/$ , one can differentiate an arbitrary program efficiently. In this paper, we review why instruction-level AD is hard for traditional machine learning frameworks and propose a solution to these problems by back-propagating a reversible Turing machine. Instruction level AD is a powerful tool to generate backward rules. We demonstrate its power by differentiating over the reversible implementations of exp function, and linear algebra functions unitary matrix multiplication and QR decomposition. It is also a promising direction towards solving the notorious memory wall problem in machine learning. Also, we discuss the challenges that we face towards rigorous reversible programming from the instruction and hardware perspective.

## I. INTRODUCTION

Computing the gradients of a numeric model  $f : \mathbb{R}^m \rightarrow \mathbb{R}^n$  plays a crucial role in scientific computing. Automatic differentiation (AD) is the key technique to obtain gradients automatically by propagating gradients with the chain rule. The propagation of gradients can happen in two directions. Consider a program  $\vec{y} = f(\vec{x})$ , a tangent mode AD propagate the gradient forward and computes one column of its Jacobian  $\frac{\partial \vec{y}}{\partial x_i}$  efficiently, where  $x_i$  is one of the input variables, whereas an adjoint mode AD propagates the gradients backward and computes one row of Jacobian  $\frac{\partial y_i}{\partial \vec{x}}$  efficiently. [1] For this reason, the adjoint mode AD is preferred in variational applications, where the loss as output is always a scalar. However, implementing adjoint mode AD is harder than implementing its tangent mode counterpart, because it requires a program's intermediate information for back-propagation. These information includes

1. the computational graph,
2. and variables used in back propagation.

A computational graph is a directed acyclic graph (DAG) that records the relationship between data (edges) and functions (nodes). In Pytorch [2] and Flux [3], every variable has a tracker field that stores its parent information, i.e., the input data and function that generate this variable. TensorFlow [4] implements a static computational graph as a description of the program before actual computation happens. Source to source AD package Zygote [5, 6] uses an intermediate representation (IR) of a program, the static single assignment (SSA) form, as the computational graph. To cache intermediate states, it has to access the global storage.

Several limitations are observed in these AD implementations due to the recording and caching. First of all, these packages require a lot of primitive functions with programmer-defined backward rules. These backward rules are not necessary given the fact that, at the lowest level, these

primitive functions are compiled to a finite set of instructions including  $+$ ,  $-$ ,  $*$ ,  $/$ , and conditional jump. By defining backward rules for these basic instructions, AD should work without extra effort. These machine learning packages do not use instructions as the computational graph, because the overhead of memorizing the computational graph and caching intermediate states is significant. The memory overhead for caching inputs of every instruction is a linear to the computing time. In many deep learning models like recurrent neural network [7] and residual neural networks [8], the depth can reach several thousand. The memory is often the bottleneck of these programs, which is known as the memory wall problem. [9] Secondly, inplace functions are not supported in these packages. This limitation comes the requirement of caching intermediate states. If a data is allowed to change inplace, then the cached data would not be safe anymore. Even if Zygote uses SSA as the computational graph, it is not trivial to handle inplace functions properly. On the other side, most functions in BLAS and LAPACK are implemented as inplace functions. As a result, all AD packages that use BLAS and LAPACK functions have to define their own backward rules for their non-inplace wrappers due to the lack of AD support to inplace functions. Thirdly, obtaining higher-order gradients are not efficient in these packages. For example, in most machine learning packages, people back-propagate the whole program of obtaining first-order gradients to obtain second-order gradients. The repeated use of back-propagation algorithm causes an exponential overhead concerning the order of gradients. A better approach is using Taylor propagation like in JAX [10]. However, Taylor propagation requires tedious work of implementing higher order backward rules for all primitives.

Our solution to these issues is making a program reversible. It is not the same as the workarounds in the machine learning field that use information buffer [11] and reversible activation functions to reduce the memory allocations in recurrent neural networks [12] and residual neural networks [13]. Our approach is general purposed. We develop NiLang, an embedded domain-specific language (eDSL) in Julia language [14, 15] that implements a reversible Turing machine (RTM). [16, 17]. This eDSL contains a macro that generates reversible functions, which is completely com-

---

\* [cacate0129@iphy.ac.cn](mailto:cacate0129@iphy.ac.cn)

patible with Julia ecosystem. It provides reversible control flows, instructions and memory managements, meanwhile provides multiple-dispatch based abstraction. With these features and less than 100 lines of code, we implemented the AD engine that can differentiate any thing, including linear algebra functions.

NiLang is not the first implementation of RTM. In history, there have been some prototypes of reversible languages like Janus [18], R (not the popular one) [19], Erlang [20] and object-oriented ROOPL [21]. These languages have reversible control flow that allows user to input an additional postcondition in control flows to help programs run backward. In the past, the primary motivation for making a program time-reversible is to support reversible devices. Reversible devices do not erase information hence do not have a lower bound of energy consumption by Landauer principle [22]. However, people show less and less interest to reversible programming since 15 years ago, because the energy efficiency of CMOS devices are still two orders [17] above this lower bound, removing this lower bound is not an urgent problem yet. Our work breaks the information barrier between the machine learning community and the reversible programming community, and provides yet another strong motivation to develop reversible programming.

In this paper, we first introduce the language design of NiLang in Sec. II. In Sec. III, we explain the algorithm to back-propagation Jacobians and Hessians in this eDSL. In Sec. IV, we show several examples including Fobonacci number,  $\exp$  function, unitary matrix multiplication and QR decomposition. We show how to generate first order and second order backward rules for these functions. In Sec. V, we discuss several important issues, how time-space tradeoff works, reversible instructions and hardware, and finally, an outlook to some open problems to be solved. In the appendix, we show the grammar of NiLang and a gradient free self-consistent training strategy.

## II. LANGUAGE DESIGN

### A. Intruductions to reversible language design

In a modern programming language, functions are pushed to a global stack for scheduling. The memory layout of a function is consisted of input arguments, a function frame with information like the return address and saved memory segments, local variables, and working stack. After the call, the function clears these runtime information, only stores the return value. In the reversible programming style, this kind of design pattern is no longer the best practice. One can not discard input variables and local variables easily after a function call, since discarding information may ruin reversibility. Hence reversible functions are very differential from irreversible ones from multiple perspectives.

First of all, the memory management in a reversible language is special. A variable can not be assigned or discarded directly in a reversible language. There are two basic types of memory management in traditional reversible languages,

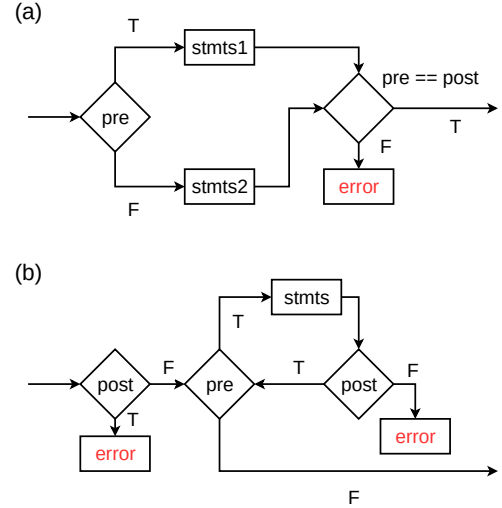


Figure 1. Flow chart for reversible (a) if statement and (b) while statement. “stmts”, “stmts1” and “stmts2” are statements, statements in true branch and statements in false branch respectively. “pre” and “post” are precondition and postconditions respectively.

the ancilla and the stack. In Janus, ancillas are handled by a statement pair of local and delocal. local initializes a variable to a specific value, while delocal deallocates that variable with a preassumed value. Stack operations includes push and pop statements, it is similar to a traditional stack except it zero-clears the variable after pushing and presupposes the variable being zero-cleared before popping. To summarize, reversible program deallocates a variable only when its state is known.

Secondly, the control flows are special. The if statement is shown in Fig. 1 (a), the program enters the branch specified by precondition. After executing this branch, the program checks the consistency of precondition and postcondition to make sure they are the same. In the reverse pass, the program enters the branch specified by the postcondition. For the while statement shown in Fig. 1 (b), before entering, the program preassumes the postcondition is false. After each iteration, the program asserts the postcondition to be true. The inverse function exchanges the precondition and postcondition. The definition of reversible for statement is similar to irreversible ones. Before entering the loop, the program stores variables start, step, and stop locally. After executing the loop, the program checks the values of these variables to make sure they are not changed. The reverse program exchanges start and stop and inverse the sign of step.

Lastly, the basic arithmetic and boolean instructions are also different. Every instruction has a unique inverse that can undo the changes. For example  $z += x * y$  is legal reversible instruction for integers, with its inverse  $z -= x * y$ , while  $x *= y$  is not. In the following discussion, we assume  $x += y$  is reversible for floating point numbers although it is not true considering rounding error. A not very well known fact is  $+$ ,  $-$ ,  $*$ ,  $/$  instructions can be implemented reversibly for both integers and floating point numbers, it is

only a matter of decision. We will come to this point later in Sec. VB

## B. NiLang’s Reversible IR

In our paper, we want use a reversible program as the computational graph. To obtain the gradient, we need to insert the code of obtaining gradients into the reversed program. The reversible language design should have related abstraction power to provide polymorphism on the same instruction sequence. This motivates us to design a new reversible language that can fit this task.

The grammar of NiLang is shown in Appendix A. Julia language’s meta programming and its package MLStyle [23] allow user to define an eDSL to explain the reversible statements conveniently. The type inference and just in time compiling can remove most overheads in our eDSL, providing a comparable performance with respect to its irreversible counterpart. Most importantly, the multiple dispatch in Julia provides the polymorphism that allowing us to obtain the gradients on the reversed program. The main feature of NiLang is contained in a single macro `@i`. The following example shows a minimal working example of compiling a NiLang function to native julia function.

```
julia> using NiLangCore, MacroTools

julia> macroexpand(Main, :(@i function f(x, y)
    SWAP(x, y)
end)) |> prettify
quote
  $(Expr(:meta, :doc))
  function $(Expr(:where, :(f(x, y))))
    gaur = SWAP(x, y)
    x = (NiLangCore.wrap_tuple(gaur))[1]
    y = (NiLangCore.wrap_tuple(gaur))[2]
    return (x, y)
  end
  if typeof(f) != typeof(~f)

    function $(Expr(:where, :(( # $ TODO: remove this comment
      mongoose::typeof(~f))(x, y))))
      mandrill = (~SWAP)(x, y)
      x = (NiLangCore.wrap_tuple(mandrill))[1]
      y = (NiLangCore.wrap_tuple(mandrill))[2]
      return (x, y)
    end
  end
  if ! (NiLangCore._hasmethod1(
    NiLangCore.isreversible, typeof(f)))
    NiLangCore.isreversible(::typeof(f)) = true
  end
end
```

Macro `@i` generates two functions `f` and `~f`, where `~f` is an callable of type `Inv{typeof(f)}`. `NiLangCore.wrap_tuple` is used to unify output data types, it will wrap any non-tuple variable to a tuple. The output of an instruction or a function call will be assigned back to input variables, this requires the length output

variables same as input variables. In NiLang, this is always true because input variables of a function will be returned automatically, i.e. a reversible function defined by NiLang is a mapping between the same set of variables. At last, `isreversible` is binded to `f` to mark it as reversible.

To design such an eDSL, we first introduce a reversible IR that plays a central role in NiLang. In this IR, a statement can be an instruction, a function call, a control flow, a macro call, or the inverse statement `~`. Any statement in this IR has a unique inverse as shown in Table I.

statement	inverse
<code>&lt;f&gt;(&lt;args&gt;...)</code>	<code>(~&lt;f&gt;)(&lt;args&gt;...)</code>
<code>&lt;y!&gt; += &lt;f&gt;(&lt;args&gt;...)</code>	<code>&lt;y!&gt; -= &lt;f&gt;(&lt;args&gt;...)</code>
<code>&lt;y!&gt; .+= &lt;f&gt;(&lt;args&gt;...)</code>	<code>&lt;y!&gt; .-= &lt;f&gt;(&lt;args&gt;...)</code>
<code>&lt;y!&gt; ∇= &lt;f&gt;(&lt;args&gt;...)</code>	<code>&lt;y!&gt; ∇= &lt;f&gt;(&lt;args&gt;...)</code>
<code>&lt;y!&gt; .∇= &lt;f&gt;(&lt;args&gt;...)</code>	<code>&lt;y!&gt; .∇= &lt;f&gt;(&lt;args&gt;...)</code>
<code>@anc &lt;a&gt; = &lt;expr&gt;</code>	<code>@deanc &lt;a&gt; = &lt;expr&gt;</code>
<code>begin</code> <code>&lt;stmts&gt;</code> <code>end</code>	<code>begin</code> <code>~(&lt;stmts&gt;)</code> <code>end</code>
<code>if (&lt;pre&gt;, &lt;post&gt;)</code> <code>&lt;stmts1&gt;</code> <code>else</code> <code>&lt;stmts2&gt;</code> <code>end</code>	<code>if (&lt;post&gt;, &lt;pre&gt;)</code> <code>~(&lt;stmts1&gt;)</code> <code>else</code> <code>~(&lt;stmts2&gt;)</code> <code>end</code>
<code>while (&lt;pre&gt;, &lt;post&gt;)</code> <code>&lt;stmts&gt;</code> <code>end</code>	<code>while (&lt;post&gt;, &lt;pre&gt;)</code> <code>~(&lt;stmts&gt;)</code> <code>end</code>
<code>for &lt;i&gt;=&lt;m&gt;:&lt;s&gt;:&lt;n&gt;</code> <code>&lt;stmts&gt;</code> <code>end</code>	<code>for &lt;i&gt;=&lt;m&gt;:-&lt;s&gt;:&lt;n&gt;</code> <code>~(&lt;stmts&gt;)</code> <code>end</code>
<code>@safe &lt;expr&gt; x</code>	<code>@safe &lt;expr&gt;</code>

Table I. A collection of reversible statements.

The conditional expression in a `if` or a `while` statements is a two-element tuple that consists of a precondition and a postcondition. Macro `@anc` and `@deanc` plays the same role as `local` and `delocal` in Janus. `@anc a = <expr>` binds variable `a` to an initial value specified by `<expr>`. Its inverse `@deanc a = <expr>` deallocates the variable `a`. Before deallocating the variable, the program checks that the its value is the same as the value of `<expr>`, otherwise throws an `InvertibilityError`. `@anc` and `@deanc` must appear in pairs inside a function call, a `while` statement, or a `for` statement. A function call in NiLang supports Julia’s broadcasting magic, a function defined on scalars can be broadcasted to vectors by adding a `“.”` after the function call. The `@safe` macro allows users to use external statements that do not break reversibility. For example, one can use `@safe @show var` for debugging. There is not stack operations in our language design because we haven’t find a use case that requires a stack yet.

## C. Compiling

The interpretation of a reversible function consists three stages.

The first stage preprocess human inputs to a reversible IR. The preprocessor expands the symbol `~` in the postcondition field of `if` statement to the precondition, adds missing `@deanc` to ensure `@anc` and `@deanc` statements appear in pairs and expands `@routine` macro. Here, `@routine r <stmt>` records a statement to symbol `r`. In preprocessing stage, `~@routine r` will be replaced to `~<stmt>` for uncomputing. We will use macro extensively in the examples in Sec. IV. The following example preprocess an `if` statement to the reversible version.

```
julia> prettify(@code_preprocess if (pre, ~)
      z += x * y
      end)
:(if (pre, pre)
  z += x * y
  else
  end)
```

The second stage generates the reversed IR according to table Table I.

```
julia> prettify(@code_reverse if (pre, post)
      z += x * y
      else
      z += x / y
      end)
:(if (post, pre)
  z -= x * y
  else
  z -= x / y
  end)
```

The third stage is translating this IR and its inverse to native Julia code. At the end of a function definition, it attaches a return statement that returns variables in the argument list. Now the function is ready to execute on the host language. The following example shows how to compile an `if` statement.

```
julia> prettify(@code_interpret if (pre, post)
      z += x * y
      else
      z += x / y
      end)
quote
  bat = pre
  if bat
    @assignback (PlusEq(*))(z, x, y)
  else
    @assignback (PlusEq(/))(z, x, y)
  end
  @invcheck post bat
end
```

The compiler translates the instruction according to Table II and adds `@assignback` before each instruction and function call statement. The macro `@assignback` assigns the output of a function to the argument list of a function, which will be explained in detail in the next subsection. At the end of the code `@invcheck post bat` is added to check the consistency between preconditions and postconditions to ensure reversibility. This statement will throw an error if target variables `bat` and `post` are not “equal” to each other up to the rounding error.

## D. Types and Dataviews

So far, the language design is not too different from a traditional reversible language. To implement the adjoint mode AD, we introduce types and dataviews.

The type that can be used in the reversible context is just a normal Julia type with extra requirements that its default constructor is not overwritten. The constructor of a reversible type should be a reversible function. Its inverse “destructor” unpacks data and deallocates derived fields. One can use `@iconstruct` to define such a reversible constructor. The following example defines a reversible generator that copies the input as a derived field.

```
using NiLangCore, Test

struct DVar{T}
  x::T
  g::T
end

@iconstruct function DVar(xx, gg=zeros(xx))
  gg += identity(xx)
end

@test (~DVar)(DVar(0.5)) == 0.5
```

The first parameter `xx` is the actual input of constructor. The assign statement in argument list `gg = zeros(xx)` initializes a new memory as a derived field. The function body is

a reversible program that transforms  $\mathbf{xx}$  and  $\mathbf{gg}$  reversibly. Finally, the compiler appends a default constructor  $\mathbf{DVar}(\mathbf{xx}, \mathbf{gg})$  at the end to instantiate a new object as the return value. The destructor that coverts an object of type  $\mathbf{DVar}\{T\}$  to type  $T$  can be defined by reversing the above statements.

A dataview of variable can be the variable itself, a field of its view, an array element of its view, or a bijective mapping of its view. Let us first consider the following example.

```
julia> using NiLangCore.ADCore

julia> arr = [GVar(3.0), GVar(1.0)]
2-element Array{GVar{Float64,Float64},1}:
 GVar(3.0, 0.0)
 GVar(1.0, 0.0)

julia> x, y = 1.0, 2.0
(1.0, 2.0)

julia> @instr -arr[2].g += x * y

julia> arr
2-element Array{GVar{Float64,Float64},1}:
 GVar(3.0, 0.0)
 GVar(1.0, -2.0)
```

`-arr[2].g`,  $\mathbf{x}$  and  $\mathbf{y}$  are all dataviews. In Julia language, `-grad(arr[2]) += x * y` statement will throw a syntax error because a function call `-(...)` can not be assigned, and `arr[3]` is an immutable type. In our eDSL, we wish it works because a memory cell is assumed to be modifiable in our eDSL. `@instr` translate the above statement to

```
1 res = (PlusEq{*})(-arr[2].g, x, y)
2 arr[2] = chfield(arr[2], Val{:g},
3   chfield(arr[2].g, -, res[1]))
4 x = res[2]
5 y = res[3]
```

The first line `PlusEq{*})(-arr[2].g, x, y)` computes the output, which is a tuple of length 3. At lines 2-3, `chfield(x, Val{:g}, val)` modifies the `g` field of  $\mathbf{x}$  and `chfield(x, -, res[1])` returns `-res[1]`. Here, modifying a field requires the default constructor of a type is not overwritten. The assignments at lines 4 and 5 are straightforward.

### III. AUTOMATIC DIFFERENTIATION

#### A. First order gradient

Given a node  $\vec{y} = f(\vec{x})$  in a computational graph, the adjoint mode AD propagates the Jacobians in the reversed direction

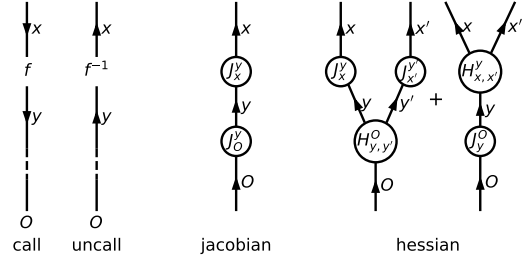


Figure 2. Adjoint rules for Jacobians and Hessians in tensor network language.

like

$$\begin{aligned} J_{O'}^O &= \delta_{O,O'}, \\ J_x^O &= J_y^O J_x^y, \end{aligned} \quad (1)$$

where  $O$  represents the outputs of the program,  $J_{x/y}^O$  is the gradient to be propagated, and  $J_x^y$  is the local Jacobian matrix. The Einstein's notation is used here so that duplicated indices are summed over. This back-propagation rule can be rewritten in the language of tensor networks [24], as shown in Fig. 2, where a circle is a tensor, dangling edges are unpaired labels and connected edges are paired labels.

In reversible programming with multiple-dispatch, we implement the adjoint mode AD as

---

#### Algorithm 1: Reversible programming AD

---

**Result:**  $\text{grad}(\vec{x}_g)$   
 let  $\text{iloss}$  be the index of loss variable in  $\vec{x}$   
 $\vec{y} = f(\vec{x})$   
 $\vec{y}_g = \mathbf{GVar}(\vec{y})$   
 $\text{grad}(\vec{y}_g[\text{iloss}]) += 1.0$   
 $\vec{x}_g = f^{-1}(\vec{y}_g)$

---

Here, `GVar` is a reversible type. The constructor attaches a zero gradient field to a variable. If the input is an array, `GVar` will be broadcasted to each array element. One can access the gradient field of a `GVar` instance through the `grad` dataview. Its inverse `~GVar` deallocates the gradient field safely and returns its value field. Here, "safely" means before deallocation, the program will check the gradient field to make sure its value is restored to 0. When an instruction meets a `GVar`, besides computing its value field `value(\vec{y}) = \text{instruct}(\text{value}(\vec{x}))`, it also updates the gradient field  $\text{grad}(\vec{y}) = [J_{\vec{x}}^{\vec{y}}]^{-1} \text{grad}(\vec{x})$ , where  $[J_{\vec{x}}^{\vec{y}}]^{-1}$  is the Jacobian of `instruct`<sup>-1</sup>. One can define this gradient function on either `instruct` or `instruct`<sup>-1</sup>. If one defines the backward rule on `instruct`, the compiler will generate the backward rule for its inverse `instruct`<sup>-1</sup> as the inverse function. This is doable because the inverse and adjoint operation commutes [25]. This implementation that changes the elementary data type is similar to the use of dual number in tangent mode AD [26]. In the following example, We bind the adjoint function of `ROT` to its reverse `IROT` by defining a new function that dispatch to `GVar`.



```

@i function IROT(a!::GVar, b!::GVar, θ::GVar)
    IROT(value(a!), value(b!), value(θ))
    NEG(value(θ))
    value(θ) -= identity(π/2)
    ROT(grad(a!), grad(b!), value(θ))
    grad(θ) += value(a!) * grad(a!)
    grad(θ) += value(b!) * grad(b!)
    value(θ) += identity(π/2)
    NEG(value(θ))
    ROT(grad(a!), grad(b!), π/2)
end

```

The definition of ROT instruction could be found in Sec. B. This backward rule has been included in NiLang, one can check the gradients by typing in a Julia REPL

```

julia> using NiLang, NiLang.AD

julia> x, y, θ = GVar(0.5), GVar(0.6), GVar(0.9)
              (GVar(0.5, 0.0), GVar(0.6, 0.0)), GVar(0.9, 0.0)

julia> @instr grad(x) += identity(1.0)

julia> @instr ROT(x, y, θ)

julia> x, y, θ
(GVar(-0.1591911616411577, 0.6216099682706646),
 GVar(0.7646294357761403, 0.7833269096274833),
 GVar(0.8999999999999999, 0.6))

```

The implementation of Algorithm 1 is so short that we present the function definition as follows.

```

@i function (g::Grad)(args...; kwargs...)
    @safe @assert count(x -> x isa Loss, args) == 1
    @anc iloss = 0
    @routine getiloss begin
        for i=1:length(args)
            if (tget(args,i) isa Loss, iloss==i)
                iloss += identity(i)
                (~Loss)(tget(args,i))
            end
        end
    end
    end

    g.f(args...; kwargs...)
    GVar.(args)
    grad(tget(args,iloss)) += identity(1.0)
    (~g.f)(args...; kwargs...)

    ~@routine getiloss
end

```

Here, we use Loss type to mark the loss variable. Input variables must contain exactly one Loss. This program first checks the input parameters and locates the loss variable as iloss. Then Loss unwraps the loss variable. After computing the forward pass and backward pass, @routine

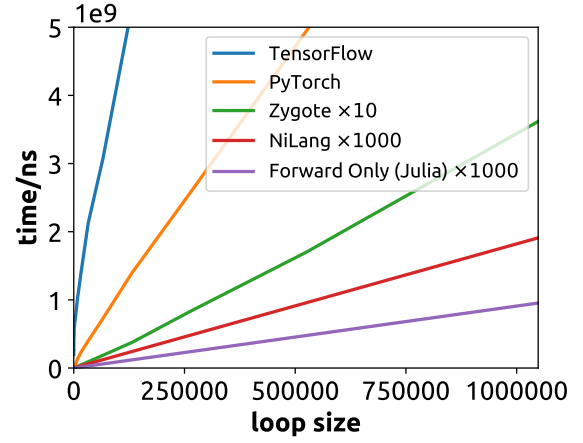


Figure 3. The time to obtain gradient as function of loop size.  $\times n$  in legend represents a rescaling of time.

getiloss uncomputes the ancilla iloss and returns the location information to the loss variable. tget(args, i) returns the  $i$ -th element of a tuple. Here, in order to avoid possible ambiguity in supporting array indexing, we forbid tuple indexing deliberately.

The overhead of using GVar type is negligible thanks to Julia’s multiple-dispatch and type inference. Let us consider a simple example that accumulates 1.0 to a target variable  $x$  for  $n$  times.

[JG: Grammaly here!]

```

julia> using NiLang, NiLang.AD, BenchmarkTools

julia> @i function prog(x, one, n::Int)
    for i=1:n
        x += identity(one)
    end
end

julia> @benchmark prog'(Loss(0.0), 1.0, 10000)
BenchmarkTools.Trial:
  memory estimate:  1.05 KiB
  allocs estimate:  39
  -----
  minimum time:     35.838 μs (0.00% GC)
  median time:      36.055 μs (0.00% GC)
  mean time:        36.483 μs (0.00% GC)
  maximum time:     185.973 μs (0.00% GC)
  -----
  samples:          10000
  evals/sample:     1

```

We implement the same function with TensorFlow, PyTorch and Zygote for comparison. The code could be found in our paper’s github repository [27]. Benchmark results on CPU Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz are shown in Fig. 3. One can see that the NiLang implementation is unreasonably fast, it is approximately two times the forward pass written in native Julia code. Reversible programming

is not always as fast as its irreversible counterparts. In practical applications, a reversible program may have memory or computation overhead. We will discuss the details of time and space trade off in Sec. V A.

## B. Second-order gradient

Second-order gradients can be obtained in two different approaches.

### 1. Back propagating first-order gradients

Back propagating the first-order gradients is the most widely used approach to obtain the second-order gradients. Suppose the function space is closed under gradient operation, one can obtain higher-order gradients by recursively differentiating lower order gradient functions without defining new backward rules.

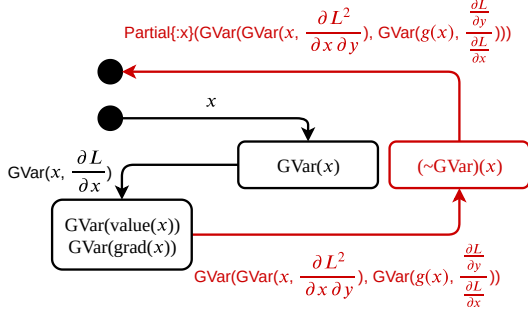


Figure 4. Obtaining the second-order gradient with the reversible differentiation approach. Black lines are computing gradients, red lines are back-propagating the process of obtaining the first-order gradients. Annotations on lines are data types used in the computation.

Fig. 4 shows the four passes in computing Hessian in this way. The first two passes (black lines) obtains the gradients. Before entering the third pass, the program wraps each field in GVar with another layer of GVar. Then we pick a variable  $x_i$  and add 1 to  $\text{grad}(\text{grad}(x_i))$  to compute the  $i$ -th row of Hessian. Before entering the final pass, the  $\sim\text{GVar}$  is called. It does not unwrap GVar directly because the second-order gradient fields may be nonzero in this case. Instead, we use  $\text{Partial}\{x\}(\cdot)$  to take the  $x$  field of an instance without deallocating memory. By repeating the above process for different  $x_i$ , one can obtains the Hessian matrix.

### 2. Taylor propagation

A probably more efficient approach is back-propagating Hessians directly [28] using the relation

$$\begin{aligned} H_{O',O''}^O &= \mathbf{0}, \\ H_{x,x'}^O &= J_x^y H_{y,y'}^O J_{x'}^{y'} + J_y^O H_{x,x'}^y. \end{aligned} \quad (2)$$

Here, the Hessian tensor  $H_{x,x'}^O$  is rank three, where the top index is often taken as a scalar and omitted. In tensor network language, the above equation can be represented as the right panel of Fig. 2. Hessian propagation is a special case of Taylor propagation. With respect to the order of gradients, Taylor propagation is exponentially more efficient in obtaining higher-order gradients than differentiating lower order gradients recursively. However, the exhausted support to Taylor propagation [10] requires much more effort than Jacobian propagation, this is why most AD packages choose the recursive approach. Instruction level automatic differentiation has the advantage of having very limited primitives. It is more flexible in obtaining higher-order gradients like Hessian. An example to obtain Hessians is provided in Sec. IV B.

## C. Gradient on ancilla problem

An ancilla can also carry gradient during computation. As a result, even if an ancilla can be uncomputed rigorously in the original program, its GVar version can not be safely uncomputed. In these case, we simply “drop” the gradient field instead of raising an error. In this subsection, we prove the correctness of the following theorem

**Theorem 1.** *Dropping the gradient field of an ancilla when deallocating does not make the gradient function irreversible.*

*Proof.* Consider a reversible function  $\vec{y}, b = f(\vec{x}, a)$ , where  $a$  and  $b$  are the input and output values of an ancilla. The reversibility requires  $b = a$  for any  $\vec{x}$ . So that

$$\frac{\partial b}{\partial \vec{x}} = \vec{0}. \quad (3)$$

Suppose in the backward pass, we discard the gradient field of  $b$ . Since the gradient fields are derived from the values of variables, they should not have any effect to the value fields. The rest is to show changing the value of  $\text{grad}(b)$  does not result in a different  $\text{grad}(\vec{x})$  in the backward pass. It can be seen from the back-propagation rule

$$\frac{\partial O}{\partial \vec{x}} = \frac{\partial O}{\partial \vec{y}} \frac{\partial \vec{y}}{\partial \vec{x}} + \frac{\partial O}{\partial b} \frac{\partial b}{\partial \vec{x}}, \quad (4)$$

where the second term with  $\frac{\partial O}{\partial b}$  vanishes naturally. Hence the gradient field of an ancilla can be any value when entering a function, it does not have to be the discarded value.  $\square$

This theorem is very important to the AD in NiLang. Without it, ancillas and GVar will not fit with each other. On the other side, we emphasis that although the initial value of the gradient field of an ancilla can be randomly chosen, not having a gradient field at the beginning is a different story.

## IV. EXAMPLES

### A. Computing Fibonacci Numbers

An example that everyone likes

## B. exp function

An exp function can be computed using Taylor expansion

$$y+ = \sum_n \frac{x^n}{\text{factorial}(n)} \quad (5)$$

One can compute the accumulated item  $s_n \equiv \frac{x^n}{\text{factorial}(n)}$  iteratively as  $s_n = \frac{x s_{n-1}}{n}$ . Considering the fact that product and division are considered as irreversible in NiLang, one can not deallocate  $s_{n-1}$  after computing  $s_n$ . This recursive computation mimics the famous pebble game [16]. However, there is no known constant memory and polynomial time solution to pebble game. Here the case is different. Notice  $*$  and  $/$  are arithmetically reversible to each other, we can “uncompute” previous state  $s_{n-1}$  by  $s_{n-1} = \frac{n s_n}{x}$  approximately, and use the dirty ancilla in next iteration. The implementation is

```
using NiLang

@i function rfib(out!, n::T) where T
    @anc n1 = zero(T)
    @anc n2 = zero(T)
    @routine init begin
        n1 += identity(n)
        n1 -= identity(1)
        n2 += identity(n)
        n2 -= identity(2)
    end
    if (value(n) <= 2, ~)
        out! += identity(1)
    else
        rfib(out!, n1)
        rfib(out!, n2)
    end
    ~@routine init
end
```

The following example shows how to construct a reversible while statement. It computes the first Fibonacci number that greater or equal to the second argument z.

```
@i function rfibn(n!, z)
    @safe @assert n! == 0
    @anc out = 0
    rfib(out, n!)
    while (out < z, n! != 0)
        ~rfib(out, n!)
        n! += identity(1)
        rfib(out, n!)
    end
    ~rfib(out, n!)
end
```

In this example, the postcondition  $n!=0$  is false before entering the loop, and becomes true in later iterations. In the reverse program, the while statement stops at  $n==0$ . If executed correctly, a user will see the following result.

```
julia> rfib(0, 10)
(55, 10)

julia> rfibn(0, 100)
(12, 100)

julia> (~rfibn)(rfibn(0, 100)...)
(0, 100)
```

```
using NiLang, NiLang.AD

@i function iexp(y!, x::T; atol::Float64=1e-14)
    where T
        @anc anc1 = zero(T)
        @anc anc2 = zero(T)
        @anc anc3 = zero(T)
        @anc iplus = 0
        @anc expout = zero(T)

        y! += identity(1.0)
        @routine r1 begin
            anc1 += identity(1.0)
            while (value(anc1) > atol, iplus != 0)
                iplus += identity(1)
                anc2 += anc1 * x
                anc3 += anc2 / iplus
                expout += identity(anc3)
                # arithmetic uncompute
                anc1 -= anc2 / x
                anc2 -= anc3 * iplus
                SWAP(anc1, anc3)
            end
        end

        y! += identity(expout)

        ~@routine r1
    end
```

Here, the definition of SWAP instruction can be found in Appendix B. The two lines bellow the comment “# arithmetic uncompute” uncompute variables anc1 and anc2 approximately, which is only arithmetically true. As a result, the final output is not exact due to the rounding error. On the other side, the reversibility is not affected since the inverse call at the last line of function uncomputes all ancilla bits rigorously. The while statement takes two conditions, the precondition and postcondition. Precondition  $\text{val}(\text{anc1}) > \text{atol}$  indicates when to break the forward pass and postcondition  $\text{iplus} \neq 0$  indicates when to break the backward pass.



To obtain gradients, one can wrap the variable `y!` with `Loss` type and feed it into `iexp'`

```
julia> y!, x = 0.0, 1.6
(0.0, 1.6)

julia> @instr iexp'(Loss(y!), x)

julia> grad(x)
4.9530324244260555
```

`iexp'` is a callable instance of type `Grad{typeof(iexp)}`, which wraps input variables with updates gradient fields as outputs. It is reversible and differentiable, hence one can use back-propagates this function to obtain Hessians as introduced in Sec. III B 1. It is implemented in NiLang as `simple_hessian`.

```
julia> y!, x = 0.0, 1.6
(0.0, 1.6)

julia> simple_hessian(iexp, (Loss(y!), x))
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  4.95303
```

A more efficient Taylor propagation approach introduced in Sec. III B 2 can be accessed by feeding variables into `iexp''`

```
julia> y!, x = 0.0, 1.6
(0.0, 1.6)

julia> @instr iexp''(Loss(y!), x)

julia> collect_hessian()
2×2 Array{Float64,2}:
 0.0  0.0
 0.0  4.95303
```

`iexp''` computes the second-order gradients by wrapping variables with type `BeijingRing` [29]. Whenever an  $n$ -th variable or ancilla is created, we push a ring of size  $2n - 1$  to a global tape. Whenever an ancilla is deallocated, we pop a ring from the top. The  $n$ -th ring stores  $H_{i \leq n, n}$  and  $H_{n, i < n}$ . We didn't use the symmetry relation  $H_{i,j} = H_{j,i}$  to save memory here in order to simplify the implementation of backward rules described in the right most panel of Fig. 2. The final result can be collected by calling `collect_hessian`, it will read out the Hessian stored in the global tape.

### C. QR decomposition

Let's consider a slightly non-trivial function, the QR decomposition

```
@i function iqr(Q, R, A::AbstractMatrix{T}) where T
    @anc anc_norm = zero(T)
    @anc anc_dot = zeros(T, size(A,2))
    @anc ri = zeros(T, size(A,1))
    for col = 1:size(A, 1)
        ri .= identity.(A[:,col])
        for precol = 1:col-1
            idot(anc_dot[precol], Q[:,precol], ri)
            R[precol,col] +=
                identity(anc_dot[precol])
            for row = 1:size(Q,1)
                ri[row] -= anc_dot[precol] *
                    Q[row, precol]
            end
        end
        inorm2(anc_norm, ri)

        R[col, col] += anc_norm^0.5
        for row = 1:size(Q,1)
            Q[row,col] += ri[row] / R[col, col]
        end

        ~(ri .= identity.(A[:,col]));
        for precol = 1:col-1
            idot(anc_dot[precol], Q[:,precol], ri)
            for row = 1:size(Q,1)
                ri[row] -= anc_dot[precol] *
                    Q[row, precol]
            end
        end
        inorm2(anc_norm, ri)
    end
end
```

This implementation of QR decomposition is very naive that does not consider reorthogonalization. `idot` and `inorm2` are functions to compute dot product and vector norm. They are implemented as

```

@i function idot(out, v1::AbstractVector{T}, v2)
    where T
    @anc anc1 = zero(T)
    for i = 1:length(v1)
        anc1 += identity(v1[i])
        CONJ(anc1)
        out += v1[i]*v2[i]
        CONJ(anc1)
        anc1 -= identity(v1[i])
    end
end

@i function inorm2(out, vec::AbstractVector{T})
    where T
    @anc anc1 = zero(T)
    for i = 1:length(vec)
        anc1 += identity(vec[i])
        CONJ(anc1)
        out += anc1*vec[i]
        CONJ(anc1)
        anc1 -= identity(vec[i])
    end
end

```

One can easily check the correctness of the gradient function

```

using Test
A = randn(4,4)
q = zero(A)
r = zero(A)

@i function test1(out, q, r, A)
    iqr(q, r, A)
    out += identity(q[1,2])
end

@i function test2(out, q, r, A)
    iqr(q, r, A)
    out += identity(r[1,2])
end

@test check_grad(test1, (Loss(0.0), q, r, A);
    atol=0.05, verbose=true)
@test check_grad(test2, (Loss(0.0), q, r, A);
    atol=0.05, verbose=true)

```

Here, the `check_grad` function is a gradient checker function defined in module `NiLangCore.ADCore`.

#### D. Unitary Matrices

Unitary matrices can be used to ease the gradient exploding and vanishing problem in recurrent networks [30–32]. One of the simplest way to parametrize a unitary matrix is representing a unitary matrix as a product of two-level unitary operations [32]. A real unitary matrix of size  $N$  can be parametrized

compactly by  $N(N - 1)/2$  rotation operations [33]

$$\text{ROT}(a!, b!, \theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} a! \\ b! \end{bmatrix}, \quad (6)$$

where  $\theta$  is the rotation angle,  $a!$  and  $b!$  are target registers.

```

using NiLang, NiLang.AD

@i function umm!(x!,  $\theta$ )
    @safe @assert length( $\theta$ ) ==
        length(x!)*(length(x!)-1)/2
    @anc k = 0
    for j=1:length(x!)
        for i=length(x!)-1:-1:j
            k += identity(1)
            ROT(x![i], x![i+1],  $\theta[k]$ )
        end
    end

    @deanc k = length( $\theta$ )
end

```

Here, the ancilla  $k$  is deallocated manually by specifying its value, because we know the loop size is  $N(N - 1)/2$ . We define the test functions in order to check gradients.

```

julia> @i function isum(out!, x::Vector)
    for i=1:length(x)
        out! += identity(x[i])
    end
end

julia> @i function test!(out!, x!::Vector,  $\theta$ ::Vector)
    umm!(x!,  $\theta$ )
    isum(out!, x!)
end

julia> out, x,  $\theta$  = Loss(0.0), randn(4), randn(6);

julia> @instr test!(out, x,  $\theta$ )

julia> x
4-element Array{GVar{Float64,Float64},1}:
 GVar(1.220182125326287, 0.14540743042341095)
 GVar(2.1288634811475937, -1.3749962375499805)
 GVar(1.2696579252569677, 1.42868739498625)
 GVar(0.1083891125379283, 0.2170123344615735)

julia> @instr (~test!')(out, x,  $\theta$ )

julia> x
4-element Array{Float64,1}:
 1.220182125326287
 2.1288634811475933
 1.2696579252569677
 0.10838911253792821

```

In the above testing code, `test'` attaches a gradient field to each element of  $x$ . `~test'` is the inverse program that erase the gradient fields. `GVar` is broadcasted to arrays, thanks to Julia's just in time compiling, using an array of `GVar` does not

have much overhead.

## V. DISCUSSION AND OUTLOOK

In this paper, we introduce a Julia eDSL NiLang that simulates a reversible Turing machine (RTM). We developed an instruction-level automatic differentiation tool on this eDSL for differential programming. It can differentiate over any program to any order reliably and efficiently without sophisticated designs to memorize computational graph and intermediate states. For those who are interested in gradient free trainings. In Appendix C, we derive a self-consistency relation for reversible program that can be used in training.

In the following, we discussed some practical issues reversible programming, and several future directions to go.

### A. Time Space Tradeoff

In history, there has been many other designs of reversible languages and instruction sets. One of the main reason why RTM is not so popular is it may have either a space overhead proportional to computing time  $T$  or a computational overhead that sometimes can be exponential. In the simplest g-segment trade off scheme [34, 35],

$$Time(T) = \frac{T^{1+\epsilon}}{S^\epsilon} \quad (7)$$

$$Space(T) = \epsilon 2^{1/\epsilon} (S + S \log \frac{T}{S}) \quad (8)$$

with  $T$  and  $S$  the time and space usage on a irreversible Turing machine,  $\epsilon$  is the control parameter. It is related to the g-segment trade off parameters by  $g = k^n$ ,  $\epsilon = \log_k(2k - 1)$  with  $n \geq 1$  and  $k \geq 1$ . This section, we try to convince the readers that the overhead of reversible computing is not as terrible as people thought.

First, at  $\epsilon \rightarrow 0$ , the resource used by a RTM is same as the caching strategy used in a traditional machine learning package that memorizing every inputs of primitives. Memorizing inputs always make a program reversible since it does not discard any information. For deep neural networks, people used checkpointing trick to trade time with space [36], which is also a widely used trick in reversible programming [16]. RTM just provides more alternatives to trade time and space.

Second, some computational overhead of running recursive algorithms with limited space resources can be mitigated by "pseudo uncomputing" without sacrificing reversibility like in the `iexp` example. With reversible floating point `*` and `/` operations [37], many primitives can be implemented with pure reversible functions, which may significant decrease the computation time and memory usage. We will review this point in Sec. VB.

Third, making reversible programming an eDSL rather than a independent language allows flexible choices between reversibility and computational overhead. For example, in order to deallocate the gradient memory in a reversible language one

has to uncompute the whole process of obtaining this gradient. As a reversible eDSL, we have the flexibility to deallocate the memory irreversibly, i.e. trade energy with time. To quantify the overhead of uncomputing, we introducing the concept

**Definition 1** (program granularity). The logarithm of the ratio between the execution time of a reversible program and its irreversible counter part.

$$\log_2 \frac{Time(T)}{T} \quad (9)$$

Whenever the program uncomputes the memory, the program granularity increases by approximately one. In instruction design, defining primitive functions like `iexp`, and deallocating gradients used for training, we need uncomputing ancilla bits and increase the granularity. The overhead increase exponentially as the granularity increase. The granularity can be decreased by cleverer compilation of a program, since the uncomputing of ancillas can be executed at any level of abstraction.

### B. Instructions and Hardwares

So far, our eDSL is not really compiled to instructions, instead, it runs on a irreversible host Julia. In the future, it can be compiled to low level instructions and is execute on a reversible devices. For example, the control flow defined in this NiLang can be compiled to reversible instructions like conditioned `goto` instruction. The reversibility requires the target instruction a `comefrom` instruction that specifying the postcondition. [38]

Arithmetic instructions should be redesigned to support better reversible programs. The major obstacle to exact reversibility programming is current floating point adders used in our computing devices are not exactly reversible. There are proposals of reversible floating point adders and multipliers [37, 39–41] that introduces garbage bits to ensure reversibility. With these infrastructure, a reversible program can be executed without suffering from the irreversibility from rounding error and reduce the computational and memory overhead significantly. Notably, in machine learning field, information buffer is used to make multiplication operations [11] reversible and reduce memory cost.

Reversible programming is not necessarily related to reversible hardwares. Reversible programs are a subset of irreversible programs, hence can be simulated efficiently on CMOS devices [38]. By Using reversible hardwares [], the computation may costs zero energy by Landauer's principle [22]. Reversible hardwares are not necessarily related to reversible gates such as Toffoli gate and Fredkin gate. Devices with the ability of recovering signal energy is able to save energy, which is known as generalized reversible computing. [42, 43] In the following, we comment briefly on a special type of reversible device Quantum computer.

### 1. Quantum Computers

One of the fundamental difficulty of building a quantum computer is, unlike a classical state, an unknown quantum state can not be copied. Quantum random access memory [44] is very hard to design and implement, which is known to have many caveats [45]. A quantum state in a environment will decoherence and can not be recovered, this underlines the simulation nature of quantum devices. Reversible computing does not enjoy the quantum advantage, nor the quantum disadvantages of non-cloning and decoherence. The reversibility of quantum computing comes from the fact that microscopic processes are unitary. On the other side, the irreversibility is rare, it can come from interacting with classical devices. Irreversible processes include decaying, qubit state resetting, measurements and classical feed backs to quantum devices. These are typically harder to implement on a quantum device.

Given the fundamental limitations of quantum decoherence and non-cloning and the reversible nature of microscopic world. It is reasonable to have a reversible computing device to bridge the gap between classical and universal quantum computing. By introducing entanglement little by little, we can accelerate some basic components. For example, quantum Fourier transformation provides an interesting alternative to the adders and multipliers by introducing one additional CPHASE gate even though it is a classical-in classical-out algorithm. [46] The compiling theory developed for reversible programming will have profounding effect to quantum computing.

### C. Outlook

NiLang can be used in solving many existing problems. First of all, it can be used to generate AD rules for existing machine learning packages. For example, in Zygote, a user sometimes need to define new primitives. If this primitive is written in NiLang, then the backward rules can be generated automatically. We can built reversible BLAS and LAPACK on top of NiLang. These functions are extensively used in physics [47, 48] and many other deciplines. However, manually derived backward rules for singular value decoposition and eigenvalue decomposition functions suffer from the gradient exploding problem [49–51].

Secondly, we can use it to break the memory wall problem in machine learning. NiLang provides a more systematic and elegant time-space trade off scheme through uncomputing. A successful related example is the memory efficient domain-specific AD engine in high performance quantum simulator

Yao [52]. This domain-specific AD engine is written in a reversible style and solved the memory bottleneck in variational quantum simulations. It also gives the best performance in differentiating parameters in a quantum circuit. We can use similar idea to differentiate reversible integrators [53, 54] to enjoy AD without intermediate state caching. With reversible integrators, it should be possible to rewrite the control system in robotics [55] in a reversible style. In robotics, the control parameters are often floating point numbers rather than tensors. Writing a control program reversibly should boost the training performance a lot.

Latstly, the reversible IR is a good starting point to study quantum compiling. Most quantum programming language assumes a classical coprocessor and use classical control flows [56]. We believe reversible control flows are also very important to a universal quantum computer.

To solve the above problems better, NiLang can be improved from multiple perspectives.

- Create a better compiling tool that decreases granularity and hence reduces overhead.
- Implement rigorous reversible floating point arithmetics on hardwares to let the reversibility free from rounding error.
- Show the advantage of NiLang in parallel computing in handle asynchronous computing [57] and debugging with bidirectional move [58]. It is interesting to see how NiLang combines with other parts of Julia ecosystem like CUDAnative [59] and Debugger.

These improvements needs the participation from multiple fields.

## VI. ACKNOWLEDGMENTS

Jin-Guo Liu thank Lei Wang for motivating the project with possible applications reversible integrator, normalizing flow and neural ODE. Xiu-Zhe Luo for discussion on the implementation details of source to source automatic differetiation, Shuo-Hui Li for helpful discussion on differential geometry. Damian Steiger for telling me the comefrom joke. Tong Liu and An-Qi Chen for helpful discussion on quantum adders and multipliers. The authors are supported by the National Natural Science Foundation of China under the Grant No. 11774398, the Strategic Priority Research Program of Chinese Academy of Sciences Grant No. XDB28000000.

---

[1] L. Hascoet and V. Pascual, *ACM Transactions on Mathematical Software (TOMS)* **39**, 20 (2013).  
 [2] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, in *NIPS Autodiff Workshop* (2017).

[3] M. Innes, E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah, “Fashionable modelling with flux,” (2018), [arXiv:1811.01457 \[cs.PL\]](https://arxiv.org/abs/1811.01457).  
 [4] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat,

- I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, “TensorFlow: Large-scale machine learning on heterogeneous systems,” (2015), software available from tensorflow.org.
- [5] M. Innes, “Don’t unroll adjoint: Differentiating ssa-form programs,” (2018), [arXiv:1810.07951 \[cs.PL\]](#).
- [6] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, *CoRR* **abs/1907.07587** (2019), [arXiv:1907.07587](#).
- [7] Z. C. Lipton, J. Berkowitz, and C. Elkan, “A critical review of recurrent neural networks for sequence learning,” (2015), [arXiv:1506.00019 \[cs.LG\]](#).
- [8] K. He, X. Zhang, S. Ren, and J. Sun, 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2016), 10.1109/cvpr.2016.90.
- [9] “Breaking the Memory Wall: The AI Bottleneck,” <https://blog.semi.org/semi-news/breaking-the-memory-wall-the-ai-bottleneck>.
- [10] J. Bettencourt, M. J. Johnson, and D. Duvenaud, (2019).
- [11] D. Maclaurin, D. Duvenaud, and R. Adams, in *Proceedings of the 32nd International Conference on Machine Learning*, Proceedings of Machine Learning Research, Vol. 37, edited by F. Bach and D. Blei (PMLR, Lille, France, 2015) pp. 2113–2122.
- [12] M. MacKay, P. Vicol, J. Ba, and R. B. Grosse, in *Advances in Neural Information Processing Systems 31*, edited by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett (Curran Associates, Inc., 2018) pp. 9029–9040.
- [13] J. Behrmann, D. Duvenaud, and J. Jacobsen, *CoRR* **abs/1811.00995** (2018), [arXiv:1811.00995](#).
- [14] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, *arXiv preprint arXiv:1209.5145* (2012).
- [15] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, *SIAM Review* **59**, 65–98 (2017).
- [16] K. S. Perumalla, *Introduction to reversible computing* (Chapman and Hall/CRC, 2013).
- [17] M. P. Frank, *IEEE Spectrum* **54**, 32–37 (2017).
- [18] C. Lutz, “Janus: a time-reversible language,” (1986), *Letter to R. Landauer*.
- [19] M. P. Frank, *The R programming language and compiler*, Tech. Rep. (MIT Reversible Computing Project Memo, 1997).
- [20] I. Lanese, N. Nishida, A. Palacios, and G. Vidal, *Journal of Logical and Algebraic Methods in Programming* **100**, 71–97 (2018).
- [21] T. Haulund, “Design and implementation of a reversible object-oriented programming language,” (2017), [arXiv:1707.07845 \[cs.PL\]](#).
- [22] R. Landauer, *IBM journal of research and development* **5**, 183 (1961).
- [23] “MLStyle.jl,” <https://github.com/thautwarm/MLStyle.jl>.
- [24] R. Orús, *Annals of Physics* **349**, 117–158 (2014).
- [25] A. McInerney, *First steps in differential geometry* (Springer, 2015).
- [26] J. Revels, M. Lubin, and T. Papamarkou, “Forward-mode automatic differentiation in julia,” (2016), [arXiv:1607.07892 \[cs.MS\]](#).
- [27] “Paper’s Github Repository,” <https://github.com/GiggleLiu/nilangpaper/tree/master/codes>.
- [28] J. Martens, I. Sutskever, and K. Swersky, “Estimating the hessian by back-propagating curvature,” (2012), [arXiv:1206.6464 \[cs.LG\]](#).
- [29] When people ask for the location in Beijing, they will start by asking which ring. We use the similar approach to locate the elements of Hessian matrix.
- [30] M. Arjovsky, A. Shah, and Y. Bengio, *CoRR* **abs/1511.06464** (2015), [arXiv:1511.06464](#).
- [31] S. Wisdom, T. Powers, J. R. Hershey, J. L. Roux, and L. Atlas, “Full-capacity unitary recurrent neural networks,” (2016), [arXiv:1611.00035 \[stat.ML\]](#).
- [32] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. A. Skirlo, M. Tegmark, and M. Soljacic, *CoRR* **abs/1612.05231** (2016), [arXiv:1612.05231](#).
- [33] C.-K. LI, R. ROBERTS, and X. YIN, *International Journal of Quantum Information* **11**, 1350015 (2013).
- [34] C. H. Bennett, *SIAM Journal on Computing* **18**, 766 (1989), <https://doi.org/10.1137/0218053>.
- [35] R. Y. Levine and A. T. Sherman, *SIAM Journal on Computing* **19**, 673 (1990).
- [36] T. Chen, B. Xu, C. Zhang, and C. Guestrin, *CoRR* **abs/1604.06174** (2016), [arXiv:1604.06174](#).
- [37] T. Häner, M. Soeken, M. Roetteler, and K. M. Svore, “Quantum circuits for floating-point arithmetic,” (2018), [arXiv:1807.02023 \[quant-ph\]](#).
- [38] C. J. Vieri, *Reversible Computer Engineering and Architecture*, Ph.D. thesis, Cambridge, MA, USA (1999), aAI0800892.
- [39] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *10th IEEE International Conference on Nanotechnology* (2010) pp. 233–237.
- [40] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *2011 11th IEEE International Conference on Nanotechnology* (2011) pp. 451–456.
- [41] T. D. Nguyen and R. V. Meter, “A space-efficient design for reversible floating point adder in quantum computing,” (2013), [arXiv:1306.3760 \[quant-ph\]](#).
- [42] M. P. Frank, in *35th International Symposium on Multiple-Valued Logic (ISMVL’05)* (2005) pp. 168–185.
- [43] M. P. Frank, in *Reversible Computation*, edited by I. Phillips and H. Rahaman (Springer International Publishing, Cham, 2017) pp. 19–34.
- [44] V. Giovannetti, S. Lloyd, and L. Maccone, *Physical Review Letters* **100** (2008), 10.1103/physrevlett.100.160501.
- [45] S. Aaronson, *Nature Physics* **11**, 291 (2015).
- [46] L. Ruiz-Perez and J. C. Garcia-Escartin, *Quantum Information Processing* **16** (2017), 10.1007/s11128-017-1603-1.
- [47] J.-G. L. Hao Xie and L. Wang, [arXiv:2001.04121](#).
- [48] H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang, *Physical Review X* **9** (2019), 10.1103/physrevx.9.031041.
- [49] M. Seeger, A. Hetzel, Z. Dai, E. Meissner, and N. D. Lawrence, “Auto-differentiating linear algebra,” (2017), [arXiv:1710.08717 \[cs.MS\]](#).
- [50] Z.-Q. Wan and S.-X. Zhang, “Automatic differentiation for complex valued svd,” (2019), [arXiv:1909.02659 \[math.NA\]](#).
- [51] C. Hubig, “Use and implementation of autodifferentiation in tensor network methods with complex scalars,” (2019), [arXiv:1907.13422 \[cond-mat.str-el\]](#).
- [52] X.-Z. Luo, J.-G. Liu, P. Zhang, and L. Wang, “Yao.jl: Extensible, efficient framework for quantum algorithm design,” (2019), [arXiv:1912.10877 \[quant-ph\]](#).
- [53] P. Hut, J. Makino, and S. McMillan, *The Astrophysical Journal* **443**, L93 (1995).
- [54] D. N. Laikov, *Theoretical Chemistry Accounts* **137** (2018), 10.1007/s00214-018-2344-7.



- [55] M. Gifftaler, M. Neunert, M. Stäuble, M. Frigerio, C. Semini, and J. Buchli, *Advanced Robotics* **31**, 1225–1237 (2017).
- [56] K. Svore, M. Roetteler, A. Geller, M. Troyer, J. Azariah, C. Granade, B. Heim, V. Kliuchnikov, M. Mykhailova, and A. Paz, *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018* (2018), 10.1145/3183895.3183901.
- [57] D. R. Jefferson, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **7**, 404 (1985).
- [58] B. Boothe, *ACM SIGPLAN Notices* **35**, 299 (2000).
- [59] T. Besard, C. Foket, and B. D. Sutter, *CoRR* **abs/1712.03112** (2017), [arXiv:1712.03112](https://arxiv.org/abs/1712.03112).
- [60] M. Bender, P.-H. Heenen, and P.-G. Reinhard, *Rev. Mod. Phys.* **75**, 121 (2003).

## Appendix A: NiLang Grammar

To define a reversible function one can use “@i” plus a normal function definition like bellow

```

"""
docstring...
"""
@i function f(args..., kwargs...) where {...}
    <stmts>
end

```

where the definition of “<stmts>” are shown in the grammar on the next page. The following is a list of terminologies used the definition

- *ident*, symbols
- *num*, numbers
- $\epsilon$ , empty statement
- *JuliaExpr*, native Julia expression
- [ ], zero or one repetitions.

Here, all *JuliaExpr* should be pure, otherwise the reversibility is not guaranteed. Dataview is a view of a data, it can be a bijective mapping of an object, an item of an array or a field of an object.

```

⟨Stmts⟩ ::=  $\epsilon$ 
          | ⟨Stmt⟩
          | ⟨Stmts⟩ ⟨Stmt⟩
⟨Stmt⟩ ::= ⟨BlockStmt⟩
          | ⟨IfStmt⟩
          | ⟨WhileStmt⟩
          | ⟨ForStmt⟩
          | ⟨InstrStmt⟩
          | ⟨RevStmt⟩
          | ⟨@anc⟩ ⟨Stmt⟩
          | ⟨@routine⟩ ⟨Stmt⟩
          | ⟨@safe⟩ JuliaExpr
          | ⟨CallStmt⟩
⟨BlockStmt⟩ ::= begin ⟨Stmts⟩ end
⟨RevCond⟩ ::= ( JuliaExpr , JuliaExpr )
⟨IfStmt⟩ ::= if ⟨RevCond⟩ ⟨Stmts⟩ [else ⟨Stmts⟩] end
⟨WhileStmt⟩ ::= while ⟨RevCond⟩ ⟨Stmts⟩ end
⟨Range⟩ ::= JuliaExpr : JuliaExpr [: JuliaExpr]
⟨ForStmt⟩ ::= for ident = ⟨Range⟩ ⟨Stmts⟩ end
⟨KwArg⟩ ::= ident = JuliaExpr
⟨KwArgs⟩ ::= [⟨KwArgs⟩ , ] ⟨KwArg⟩
⟨CallStmt⟩ ::= JuliaExpr ( [⟨DataViews⟩] [; ⟨KwArgs⟩] )
⟨Constant⟩ ::= num |  $\pi$ 
⟨InstrBinOp⟩ ::= += | -= |  $\forall$ =
⟨InstrTrailer⟩ ::= [ ] ( [⟨DataViews⟩] )
⟨InstrStmt⟩ ::= ⟨DataView⟩ ⟨InstrBinOp⟩ ident [⟨InstrTrailer⟩]
⟨RevStmt⟩ ::= ~ ⟨Stmt⟩
⟨@routine⟩ ::= @routine ident ⟨Stmt⟩
⟨AncArg⟩ ::= ident = JuliaExpr
⟨@anc⟩ ::= @anc ⟨AncArg⟩
           | @deanc ⟨AncArg⟩
⟨@safe⟩ ::= @safe JuliaExpr
⟨DataViews⟩ ::=  $\epsilon$ 
               | ⟨DataView⟩
               | ⟨DataViews⟩ , ⟨DataView⟩
⟨DataView⟩ ::= ⟨DataView⟩ [ JuliaExpr ]
               | ⟨DataView⟩ . ident
               | JuliaExpr ( ⟨DataView⟩ )
               | ⟨DataView⟩ '
               | - ⟨DataView⟩
               | ⟨Constant⟩
               | ident
               | ident ...

```

## Appendix B: Instruction Table

The translation of instructions to Julia functions  
The list of instructions implemented in NiLang

instruction	translated	symbol
$y += f(args...)$	PlusEq( $f$ ) ( $args...$ )	$\oplus$
$y -= f(args...)$	MinusEq( $f$ ) ( $args...$ )	$\ominus$
$y \forall = f(args...)$	XorEq( $f$ ) ( $args...$ )	$\odot$

Table II. Instructions and their interpretation in NiLang.

instruction	output
SWAP( $a, b$ )	$b, a$
ROT( $a, b, \theta$ )	$a \cos \theta - b \sin \theta, b \cos \theta + a \sin \theta, \theta$
IROT( $a, b, \theta$ )	$a \cos \theta + b \sin \theta, b \cos \theta - a \sin \theta, \theta$
$y += a^b$	$y + a^b, a, b$
$y += \exp(x)$	$y + e^x, x$
$y += \log(x)$	$y + \log x, x$
$y += \sin(x)$	$y + \sin x, x$
$y += \cos(x)$	$y + \cos x, x$
$y += \text{abs}(x)$	$y +  x , x$
NEG( $y$ )	$-y$
CONJ( $y$ )	$y'$

Table III. A collection of reversible instructions, “.” is the broadcast operations in Julia.

## Appendix C: Learn by consistency

Consider a training task that with input  $\vec{x}^*$  and output  $\vec{y}^*$ , find a set of parameters  $\vec{p}_x$  that satisfy  $\vec{y}^* = f(\vec{x}^*, \vec{p}_x)$ . In traditional machine learning, we define a loss  $\mathcal{L} = \text{dist}(\vec{y}^*, f(\vec{x}^*, \vec{p}_x))$  and minimize it with gradient  $\frac{\partial \mathcal{L}}{\partial \vec{p}_x}$ . This works only when the target function is locally differentiable.

Here we provide an alternative by making use of reversibility. We construct a reversible program  $\vec{y}, \vec{p}_y = f_r(\vec{x}, \vec{p}_x)$ , where  $\vec{p}_x$  and  $\vec{p}_y$  are “garbage” spaces which include trainable parameters and auxiliary parameters. The algorithm can be summarized as

### Algorithm 2: Learn by consistency

```

Result:  $\vec{p}_x$ 
Initialize  $\vec{x}$  to  $\vec{x}^*$ , garbage space  $\vec{p}_x$  to random.
if  $\vec{p}_y$  is null then
  |  $\vec{x}, \vec{p}_x = f_r^{-1}(\vec{y}^*)$ 
else
  |  $\vec{y}, \vec{p}_y = f_r(\vec{x}, \vec{p}_x)$ 
  | while  $\vec{y} \neq \vec{y}^*$  do
  |   |  $\vec{y} = \vec{y}^*$ 
  |   |  $\vec{x}, \vec{p}_x = f_r^{-1}(\vec{y}, \vec{p}_y)$ 
  |   |  $\vec{x} = \vec{x}^*$ 
  |   |  $\vec{y}, \vec{p}_y = f_r(\vec{x}, \vec{p}_x)$ 

```

Here,  $\text{garbage}(\cdot)$  is a function for taking the garbage space. This algorithm utilizes the self-consistency relation

$$\vec{p}_x^* = \text{garbage}(f_r^{-1}(\vec{y}^*, \text{garbage}(f_r(\vec{x}^*, \vec{p}_x^*)))), \quad (\text{C1})$$

Similar idea of training by consistency is used in self-

consistent meanfield theory [60] in physics. The difficult part of self-consistent training is to find a self-consistency relation, here the reversibility provides a natural self-consistency relation. Learn by consistency can be used to handle discrete optimization. However, it is not a silver bullet, and should be used with caution. Let's consider the following example

```
@i function f1(y!, x, p!)
    p! += identity(x)
    y! -= exp(x)
    y! += exp(p!)
end

@i function f2(y!, x!, p!)
    p! += identity(x!)
    y! -= exp(x!)
    x! -= log(-y!)
    y! += exp(p!)
end

function train(f)
    loss = Float64[]
    p = 1.6
    for i=1:100
        y!, x = 0.0, 0.3
        @instr f(y!, x, p)
        push!(loss, y!)
        y! = 1.0
        @instr (~f)(y!, x, p)
    end
    loss
end
```

Functions `f1` and `f2` computes  $f(x, p) = e^{(p+x)} - e^x$  and stores the output in a new memory `y!`. The only difference is `f2` “uncompute” `x` arithmetically. The task of training is to find a  $p$  that make the output value equal to target value 1. After 100 steps, `f2` runs into the fixed point with  $x$  equal to 1 upto machine precision. However, `f1` does not do any training. The training of `f2` fails because this function actually computes  $f1(y, x, p) = y + e^{(p+x)} - e^x, x, x + p$ , where the training parameter  $p$  is completely determined by the garbage space on the output side  $x \cup x + p$ . As a result, shifting  $y$  directly is the only approach to satisfy the consistency relation. On the other side,  $f2(y, x, p) = y + e^{(p+x)} - e^x, \tilde{0}, x + p$ , the garbage

space can not uniquely determine the input garbage space  $p$  and  $y$ . Here we use  $\tilde{0}$  to denote the zero with rounding error.

By viewing  $\vec{x}$  and parameters in  $\vec{p}_x$  as variables, we can study the trainability from the information perspective.

**Theorem 2.** Only if the conditional entropy  $S(\vec{y}|\vec{p}_y)$  is nonzero, algorithm 2 is trainable.

*Proof.* The above example reveals a fact that the training can not work when the output  $\vec{p}_x$  completely determines  $\vec{p}_x$ , that

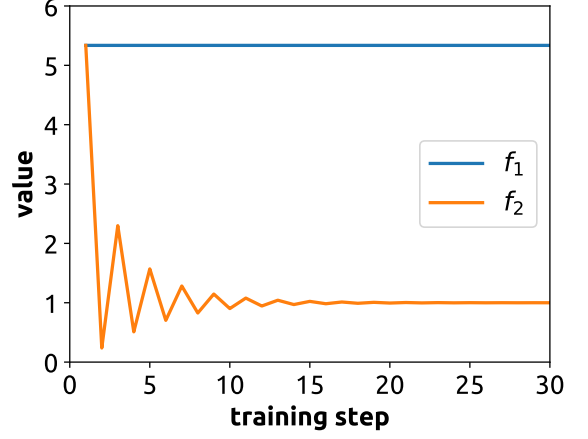


Figure 5. The value of  $x$  as a function of self-consistent training step.

is

$$\begin{aligned}
 S(\vec{p}_x|\vec{p}_y) &= S(\vec{p}_x \cup \vec{p}_y) - S(\vec{p}_y) \\
 &\leq S((\vec{p}_x \cup \vec{x}) \cup \vec{p}_y) - S(\vec{p}_y), \\
 &\leq S((\vec{p}_y \cup \vec{y}) \cup \vec{p}_y) - S(\vec{p}_y), \\
 &\leq S(\vec{y}|\vec{p}_y).
 \end{aligned} \tag{C2}$$

The third line uses the bijectivity  $S(\vec{x} \cup \vec{p}_x) = S(\vec{y} \cup \vec{p}_y)$ .  $\square$

This inequality shows that when the garbage space on the output side satisfies  $S(\vec{y}|\vec{p}_y) = 0$ , i.e. contains all information to determine the output field, the input parameters are also completely determined by this garbage space. In the above examples, it corresponds to the case  $S(e^{(x+y)-e^x}|x \cup x + y) = 0$  in `f1`. One should remove the redundancy of information by uncomputing to make training by consistency work properly.