Instruction level automatic differentiation on a reversible Turing machine

Jin-Guo Liu[1, *] and Taine Zhao[2]

[1]*Institute of Physics, Chinese Academy of Sciences, Beijing 100190, China*
[2]*Department of Computer Science, University of Tsukuba*

This paper considers differentiating a general program. We review why instruction-level AD is hard for traditional machine learning frameworks and propose a solution to these problems by writing a program reversiblly. We developed an reversible eDSL NiLang in Julia that can be used 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. We also 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 $f : \mathbb{R}^m \to \mathbb{R}^n$ plays a crutial role in scientific computing. Consider a computing process

$$\begin{aligned}
\mathbf{x}^1 &= f_1(\mathbf{x}^0)\\
\mathbf{x}^2 &= f_2(\mathbf{x}^1)\\
&\ldots\\
\mathbf{x}^L &= f_L(\mathbf{x}^{L-1})
\end{aligned}$$

where $x^0 \in R^m$, $x^L \in R^n$, L is the depth of computing. The Jacobian of this program is a $n \times m$ matrix $J_{ij} \equiv \frac{\partial x_i^L}{\partial x_j^0}$, where x_j^0 and x_i^L are single elements of inputs and outputs. Computing part of the Jacobian automatically is what we called automatic differentiation (AD). It can be classified into three classes, the tangent mode AD, the adjoint mode AD and the mixed mode AD. [1] The tangent mode AD computes the Jacobian matrix elements that related to a single input using the chain rule $\frac{\partial \mathbf{x}^k}{\partial x_j^0} = \frac{\partial \mathbf{x}^k}{\partial \mathbf{x}^{k-1}}\frac{\partial \mathbf{x}^{k-1}}{\partial x_j^0}$, while a tangent mode AD computes Jacobian matrix elements that related to a single output using $\frac{\partial \mathbf{x}^k}{\partial x_j^0} = \frac{\partial \mathbf{x}^k}{\partial \mathbf{x}^{k-1}}\frac{\partial \mathbf{x}^{k-1}}{\partial x_j^0}$. Mixed mode AD is a mixture of both. In variational applications where the loss function always outputs a scalar, the adjoint mode AD is perfered,. However, implementing adjoint mode AD is harder than implementing its tangent mode counterpart, because it requires propagating the gradients in the inverse direction of computing the loss. The back propagation of gradients requires intermediate information of a program that includes

1. the computational process,
2. and variables used in computing gradients.

The computational process is often stored in a computational graph, a directed acyclic graph (DAG) that represents the relationship between data and functions. In Pytorch [2] and Flux [3], every variable has a tracker field that stores its parent information, i.e., the input data and function generating this variable. TensorFlow [4] implements a static computational graph as a description of the program before actual computation happens. The required variables are also recorded in this graph. For source to source AD package, Tapenade [1] uses source code as the computational graph and 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, they use a global stack.

Several limitations are observed in these AD implementations due to the recording and caching. First of all, most packages require a lot of primitive functions with programmer-defined backward rules. For example, the backward rule of `exp` should be provided although it is composed of basic instructions '+', '-', '*', '/', and conditional jumps. Defining backward rules for these basic instructions in the computational graph scheme suffers from the overhead of memorizing the computational graph and caching intermediate states. Even in Tapenade, the program has to remember the control at each place where the flow merges in the forward sweep. Secondly, the memory consumption is significant, also known as the memory wall problem. [7]. The overhead of naive caching every input of instructions is linear to the computing time. In many deep learning models like recurrent neural network [8] and residual neural networks [9], the depth can reach several thousand, where the memory is often the bottleneck of these programs. Another important source of memory overhead is from the fact that inplace functions are forbidden delibrately in a computational graph based AD scheme in order to protect the cached data. 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] and beautiful differentiation [11]. However, Taylor propagation in the adjoint mode AD requires tedious implemention of higher order backward rules for primitives.

We tackle these issues by making a program reversible. In the machine learning field, reversibility has been used in reduce the memory allocations in recurrent neural networks [12] and residual neural networks [13]. These techinics include information buffer [14] and reversible activation functions [15, 16]. Our approach is general purposed. We develop

* cacate0129@iphy.ac.cn

an embedded domain-specific language (eDSL) NiLang in Julia language [17, 18] that implements reversible programming. [19, 20]. This eDSL provides a macro to generate reversible functions, and is completely compatible with Julia ecosystem. One can write reversible control flows, instructions and memory managements in this macro. Combining it with Julia's type system, we implement the AD engine within 100 lines that differentiate any program written in this eDSL, including linear algebra functions.

In history, there have been some prototypes of reversible languages like Janus [21], R (not the popular one) [22], Erlang [23] and object-oriented ROOPL [24]. In the past, the primary motivation of making a program reversible is to support reversible computing devices [25] like adiabatic energy efficiency of complementary metal–oxide–semiconductor (CMOS) [26], molecular mechanical computing system [27] and superconducting system [28, 29]. These devices either implements reversible logical gates or being able to recover signal energy. They do not erase information hence do not have a lower bound of energy consumption by Landauer principle [30]. However, investigators show less interest to reversible programming since 15 years ago, because the energy efficiency of traditional CMOS devices is still several orders [20, 31] above this lower bound, removing this lower bound is not an urgent problem yet. Our work aims to 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 back-propagation algorithm of Jacobians and Hessians in this eDSL. In Sec. IV, we show several examples including Fobonacci number, `exp` function, unitary matrix multiplication and QR decomposition [32]. 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 consists 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. For this reason, reversible functions are very

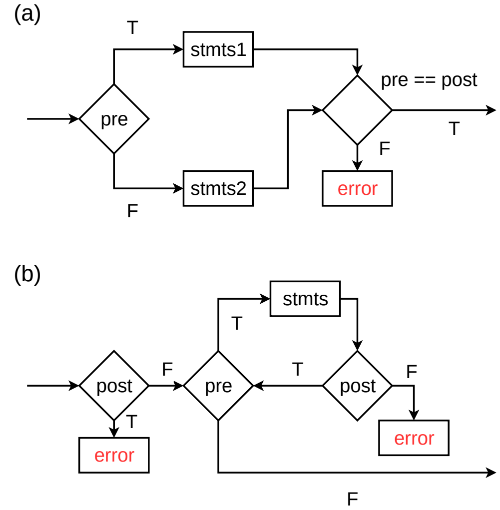

Figure 1. Flow chart for reversible (a) `if` statement and (b) `while` statement. "stmts", "stmts1" and "stmts2" are statements, statements in true branch and statements in false branch respectively. "pre" and "post" are precondition and postconditions respectively.

different from irreversible ones from multiple perspectives.

First of all, the memory management in a reversible language is different. The key difference is when a variable in a reversible program is discarded, its contents should be known. We denote the allocation of a zero emptied memory to variable `x` as `x ← 0`, and deallocate a zero emptied variable `x` as `x → 0`. A variable allocated and deallocated in a local scope is called an ancilla, it does not occupy the memory for long period. A variable can also be pushed to a stack and used later with a pop statement. This is similar to a traditional stack operation except it zero-clears the variable after pushing and presupposes the variable being zero-cleared before poping.

Secondly, a reversible control flow is sepcially designed. The reversible `if` statement is shown in Fig. 1 (a), the program enters the branch specified by precondition. After executing that 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 reversible `while` statement shown in Fig. 1 (b), before executing the condition expressions, the program preassumes the postcondition is false. After each iteration, the program asserts the postcondition to be true. In the reverse pass, we exchange the precondition and postcondition. The reversible `for` statement is similar to irreversible ones except after executing the loop, the program checks the values of these variables to make sure they are not changed. In the reverse pass, we exchange start and stop and inverse the sign of step.

Lastly, the reversible arithmetic and boolean instructions are also different. Every instruction has a unique inverse that can undo the changes. For logical expressions, we have `y ⊻= f(args...)` self reversible. In the following discussion, we assume `y += f(args...)` and `y -= f(args...)` reverse to each other although it is not true for floating point numbers considering rounding error. Here f can be identity, *, / and ^ et. al. We will discuss the number system in detail later in Sec. V B

B. NiLang's Reversible IR

In the last subsection, we have reviewed basic building blocks of a typical reversible language. In order to insert the code of obtaining gradients into the reversed program, the reversible language design should have related abstraction power. This motivates us to design a new reversible language NiLang to fit this task. NiLang is an eDSL in Julia. We choose Julia as the host language for multipile purposes. Julia's meta programming and its package for pattern matching MLStyle.jl [33] allow us to define an eDSL conveniently. Meanwhile, the type inference and just in time compiling can remove most overheads introduced in our eDSL, providing a reasonable performance. Most importantly, the multiple dispatch provides the polymorphism that will be used in our autodiff engine.

The main feature of NiLang is contained in a single macro @i that compiles a reversible function. The allowed statements in this eDSL are shown in Appendix A. The following is a minimal 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)) |> MacroTools.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 three functions f, ~f and NiLangCore.isreversible. f and ~f are a pair of functions that reverse to each other, where ~f is an callable of type Inv{typeof(f)}. In the body of f, NiLangCore.wrap_tuple is used to unify output data types, it will wrap any non-tuple variable to a tuple. The outputs of SWAP are assigned back to its input variables, in other words, a function modifies inputs inplace. At the end this function, this macro attaches a return statement that returns all input variables. NiLangCore.isreversible is a function to mark the reversibility trait of f.

To understand the design of reversibility, 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 memory allocation/deallocation, or the inverse statement "~". Any statement is this IR has a unique inverse as shown in Table I.

statement	**inverse**
<f>(<args>...)	(~<f>)(<args>...)
<y> += <f>(<args>...)	<y> -= <f>(<args>...)
<y> .+= <f>.(<args>...)	<y> .-= <f>.(<args>...)
<y> ⊻= <f>(<args>...)	<y> ⊻= <f>(<args>...)
<y> .⊻= <f>(<args>...)	<y> .⊻= <f>(<args>...)
<a> ← <expr>	<a> → <expr>
(<T1> => <T2>)(<x>)	(<T2> => <T1>)(<x>)
begin <stmts> end	begin ~(<stmts>) end

if (<pre>, <post>)
<stmts1>
else
<stmts2>
end
	if (<post>, <pre>)
	~(<stmts1>)
	else
	~(<stmts2>)
	end

while (<pre>, <post>)
<stmts>
end
	while (<post>, <pre>)
	~(<stmts>)
	end

for <i>=<m>:<s>:<n> <stmts> end	for <i>=<m>:-<s>:<n> ~(<stmts>) end
@safe <expr>	@safe <expr>

Table I. A collection of reversible statements. "." is the symbol for broadcasting magic in Julia, "~" is the symbol for reversing a statement or a function. <...> represents a non-keyword, where <pre> stands for precondition, <post> stands for postcondition, <args>... stands for the argument list of a function, <stmts> stands for statement, <exprs> stands for expression, <T1> and <T2> stand for types and the reset are variables.

,

"←" and "→" are symbols for memory allocation and deallocation, one can input them by typing "\leftarrow" and "\rightarrow" respectively followed by a Tab key in a Julia editor or REPL. "begin <stmts> end" is the block statement in Julia, it represents a code block. It can be inverted by reversing the order of <stmts> as well as each element in it. The conditional expression in if or while statements is a tuple of precondition and postcondition. Finally, the special macro @safe allows users to use external statements that do not break reversibility. For example, one can use @safe @show <var> for debugging.

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 ancilla deallocation operations to ensure ← and → statements appear in pairs and expands `@routine` macro. Here, `@routine <stmt>` records a statement. In preprocessing stage, `~@routine` will be replaced to `~<stmt>` for uncomputing. We will use macro extensively in the "compmute-copy-uncompute" design pattern of reversible programming. The following example preprocesses an `if` statement to the reversible version.

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

Here, the symbol ~ means *same as precondition*, it will be expanded at this stage.

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

```
julia>
    MacroTools.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>
    MacroTools.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 used in the reversible context is just a normal Julia type with extra requirements that its constructor is reversible. The inverse of a constructor is called a “destructor”, which unpacks data and deallocates derived fields. Data packing and unpacking are implemented in following ancilla statements.

```
    x ← new{TX, TG}(x, g)
    x → new{TX, TG}(x, g)
```

Ancilla operations with `new` function is treated specially, because `new` keeps all information in input argument list. The first statement is equivalent to says, do `x = newTX, TG(x, g)` just like a normal ancilla allocation, but do not deallocate `g` because the information of `g` is transferred to `x`. Its inverse the second statement allocates a new ancilla `g`. The following example shows how to define as reversible type `GVar`.

```
julia> using NiLangCore

julia> @i struct GVar{T,GT} <: IWrapper{T}
           x::T
           g::GT

    function GVar{T,GT}(x::T, g::GT) where {T,GT}
        new{T,GT}(x, g)
           end
           function GVar(x::T, g::GT) where {T,GT}
        new{T,GT}(x, g)
           end
           @i function GVar(x::T) where T
        g ← zero(x)
        x ← new{T,T}(x, g)
           end
           @i function GVar(x::AbstractArray)
        GVar.(x)
           end
       end

julia> GVar(0.5)
GVar{Float64,Float64}(0.5, 0.0)

julia> (~GVar)(GVar(0.5))
0.5

julia> (~GVar)(GVar([0.5, 0.6]))
2-element Array{Float64,1}:
 0.5
 0.6
```

GVar is a reversible type that copied from the AD submodule of NiLang. It is used to store gradient information of a variable. Here, we only pick two of its constructors. We put `@i` macro before both `struct` and `function` statements. The ones before functions mark reversible functions, while the one before `struct` keyword moves the definitions of `GVar` functions to the outside of this type definition.

The cast between two types can be defined conveniently with the macro `@icast`.

```
julia> @pure_wrapper A

julia> @icast A(x) => GVar(x, g) begin
           g ← zero(x)
           g += identity(1)
       end

julia> x = A(0.5)
A(0.5)

julia> @instr (A=>GVar)(x)
GVar{Float64,Float64}(0.5, 1.0)

julia> @instr (GVar=>A)(x)
A(0.5)
```

Here, we first define a simple reversible wrapper type `A` with macro `@pure_wrapper`, then define the cast rule between `A` type and `GVar` type. The function body is a reversible program that transforms `x` to `(x, g)`. The compiler appends a default constructor `DVar(xx, gg)` at the end of program to instantiate a new object as the return value. Its inverse that coverts an object of type `GVar` to type `A` is automatically generated by reversing the above statements.

To access and manipulate the fields of a data type, especially an immutable type, we introduce the dataview. 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> arr = [GVar(3.0), GVar(1.0)]
2-element Array{GVar{Float64,Float64},1}:
 GVar{Float64,Float64}(3.0, 0.0)
 GVar{Float64,Float64}(1.0, 0.0)

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

julia> @instr -arr[2].g += x * y
2.0

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

`-arr[2].g`, `x` and `y` are all dataviews. In Julia language, `-grad(arr[2]) += x * y` statement will throw a syntax error because the function call "-" can not be assigned, and `GVar` is an immutable type. In our eDSL, we wish it works because a memory cell is assumed to be modifiable in our eDSL. The secret of how it works lies in the macro `@instr`, it translates 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[3].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 `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

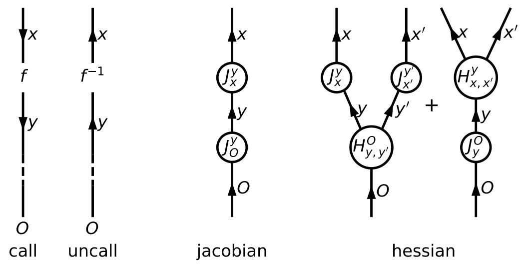

Figure 2. Computional processes in the tensor network diagram. From left to right, the diagrams represent computing, uncomputing, back propagating Jacobians and back propagating Hessians.

like

$$J^{O}_{O'} = \delta_{O,O'},$$
$$J^{O}_{x} = J^{O}_{y} J^{y}_{x}, \tag{1}$$

where O represents the outputs of the program, $J^{O}_{x/y}$ is the gradient to be propagated, and J^{y}_{x} 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 diagram of tensor networks [34], 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: grad.($\mathbf{x}_g$)
let `iloss` be the index of loss variable in $\mathbf{x}$
$\mathbf{y} = f(\mathbf{x})$
$\mathbf{y}_g$ = GVar.($\mathbf{y}$)
grad($\mathbf{y}_g$[iloss]) += 1.0
$\mathbf{x}_g = f^{-1}(\mathbf{y}_g)$

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 `instruct` meets a `GVar`, besides computing its value field `value`($\mathbf{y}$) = `instruct`(`value`($\mathbf{x}$)), it also updates the gradient field `grad`($\mathbf{y}$) = $\left[J^{\mathbf{y}}_{\mathbf{x}}\right]^{-1}$ `grad`($\mathbf{x}$), where $\left[J^{\mathbf{y}}_{\mathbf{x}}\right]^{-1}$ is the Jacobian of `instruct`$^{-1}$. One can define this gradient function on either `instruct` or `instruct`$^{-1}$. If one defines the backward rule on `instruct`, the compiler will generate the backward rule for its inverse `instruct`$^{-1}$ as the inverse function. This is doable because the inverse and adjoint operation commutes [35]. This implementation that changes the elementary data type is similar to the use of dual number in tangent mode AD [36]. 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
    iloss ← 0
    @routine for i=1:length(args)
        if (tget(args,i) isa Loss, iloss==i)
            iloss += identity(i)
            (~Loss)(tget(args,i))
        end
    end

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

    ~@routine
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` uncomputes the ancilla `iloss` and returns the location infor-

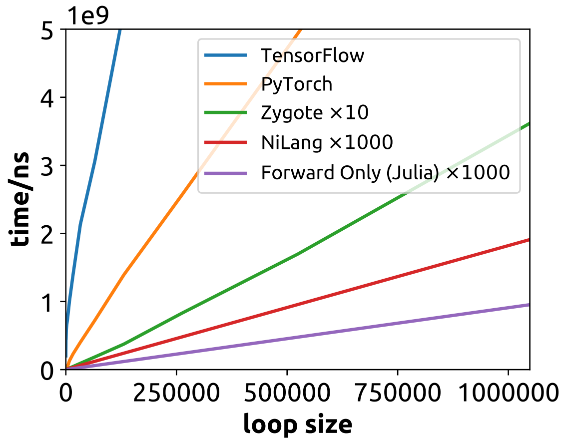

Figure 3. The time to obtain gradient as function of loop size. $\times n$ in lengend represents a rescaling of time.

mation 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: Grammarly 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 [37]. 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.

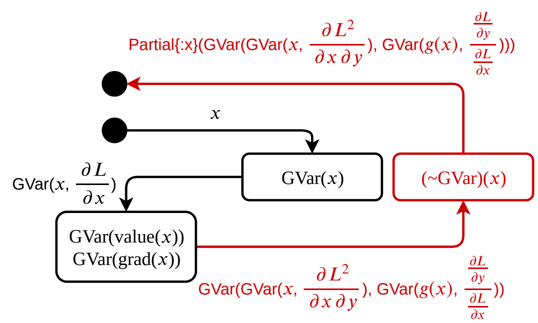

Figure 4. Obtaining the second-order gradient with the recursive differentiation approach. Black lines are computing gradients, red lines are back-propagating the process of obtaining the first-order gradients. Annotations on lines are data types used in the computation.

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 `grad(grad(`x_i`))` to compute the i-th row of Hessian. Before entering the final pass, the `~GVar` is called. It does not unwrap `GVar` directly because the second-order gradient fields may be nonzero in this case. Instead, we use `Partial{:x}(·)` 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 [38] using the relation

$$H^{O}_{O',O''} = \mathbf{0},$$
$$H^{O}_{x,x'} = J^{y}_{x} H^{O}_{y,y'} J^{y'}_{x'} + J^{O}_{y} H^{y}_{x,x'}. \tag{2}$$

Here, the Hessian tensor $H^{O}_{x,x'}$ is rank three, where the top index is often taken as a scalar and omitted. In tensor network diagram, 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 exhaused 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 $\mathbf{y}, b = f(\mathbf{x}, a)$, where a and b are the input and output values of an ancilla. The reversibility requires $b = a$ for any $\mathbf{x}$. So that

$$\frac{\partial b}{\partial \mathbf{x}} = \vec{0}. \tag{3}$$

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 `grad(b)` does not result in a different `grad.(x)` in the backward pass. It can be seen from the back-propagation rule

$$\frac{\partial O}{\partial \mathbf{x}} = \frac{\partial O}{\partial \mathbf{y}}\frac{\partial \mathbf{y}}{\partial \mathbf{x}} + \frac{\partial O}{\partial b}\frac{\partial b}{\partial \mathbf{x}}, \tag{4}$$

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, i.e. no need to cache the discarded value. □

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 begining is a different story.

IV. EXAMPLES

A. Computing Fibonacci Numbers

The following is an example that everyone likes, computing Fibonacci number recursively.

```julia
using NiLang

@i function rfib(out!, n::T) where T
    n1 ← zero(T)
    n2 ← zero(T)
    @routine 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
end
```

Here, we use `x += identity(y)` to represent `x += y`. The later is forbidden deliberately to avoid possible ambiguity between a function and a dataview. The time complexity of this recursive algorithm is exponential to input `n`. It is also possible to write a reversible linear time for loop algorithm.

A slightly non-trivial task is computing the first Fibonacci number that greater or equal to the second argument z, where a `while` statement is required.

```julia
@i function rfibn(n!, z)
    @safe @assert n! == 0
    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` in the `while` statement 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
julia> rfib(0, 10)
(55, 10)

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

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

B. exp function

An exp function can be computed using Taylor expansion

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

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 [19]. 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, NiLang.AD

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

    y! += identity(1.0)
    @routine 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
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 ancillas rigorously.

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)}`. It wraps input variables with a gradient fields as outputs. This function itself is reversible and differentiable, one can back-propagate this function to obtain Hessians as introduced in Sec. III B 1. In NiLang, it is implemented 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
```

To obtain Hessians, we can also use the Taylor propagation approach as introduced in Sec. III B 2.

```
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. It wraps variables with type `BeijingRing` [39] in the backward pass. `BeijingRing` records Jacobians and Hessians for a variable, where Hessians are stored in a global storage. 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 storage.

C. QR decomposition

Let's consider a linear algebra function, the QR decomposition

```julia
using NiLang, NiLang.AD

@i function qr(Q, R, A::AbstractMatrix{T}) where T
    anc_norm ← zero(T)
    anc_dot ← zeros(T, size(A,2))
    ri ← zeros(T, size(A,1))
    for col = 1:size(A, 1)
        ri .+= identity.(A[:,col])
        for precol = 1:col-1
            dot(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
        norm2(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

        ~begin
            ri .+= identity.(A[:,col])
            for precol = 1:col-1
                dot(anc_dot[precol], Q[:,precol], ri)
                for row = 1:size(Q,1)

     ri[row] -= anc_dot[precol] * Q[row, precol]
                end
            end
            norm2(anc_norm, ri)
        end
    end
end
```

Here, in order to avoid frequent uncomputing, we allocate ancillas `ri` and `anc_dot` as vectors. the expression in ~ is used to uncompute `ri`, `anc_dot` and `anc_norm`. `R[col, col] += anc_norm^0.5` is a ternary instruction, whose backward rule is defined in NiLang. The algorithm used to compute QR decomposition is very naive. It does not consider reorthogonalization. `idot` and `inorm2` are functions to compute dot product and vector norm. They are implemented as

```julia
@i function idot(out, v1::AbstractVector{T},
        v2) where T
    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
    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

```julia
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 loss function `test1` and `test2` are defined as single elements in the output matrices. The `check_grad` function is a gradient checker function defined in module NiLang.AD. In this example, `GVar` is broadcasted to arrays. Thanks to Julia's just in time compiling, using `GVar` in an array does not have much overhead. It should be possible to define other reversible linear algebra functions too. We leave this future projects.

D. Unitary Matrices

Unitary matrices features uniform eigenvalues and reversibility. It is widely used as an approach to ease the gradient exploding and vanishing problem [40–42] and the memory wall problem [43]. One of the simplest way to parametrize a unitary matrix is representing a unitary matrix as a product of two-level unitary operations [42]. A real unitary matrix of size N can be parametrized compactly by $N(N-1)/2$ rotation operations [44]

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

where θ is the rotation angle, `a!` and `b!` are target registers.

```
using NiLang, NiLang.AD

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

    k → length(θ)
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, θ::Vector)
           umm!(x!, θ)
           isum(out!, x!)
       end

julia> out, x, θ = Loss(0.0), randn(4), randn(6);

julia> @instr test!'(out, x, θ)

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, θ)

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. Notablly, this reversible implementation is inplace with zero memory allocation. In traditional automatic differentiation framework, one needs to implement special designs [15, 16] to describe this property.

V. DISCUSSION AND OUTLOOK

In this paper, we introduce automatic differentiation on a reversible Turing machine (RTM) and a Julia eDSL NiLang that simulates an RTM. We show a program on an RTM can be differentiated to any order reliably and efficiently without sophisticated designs to memorize computational graph and intermediate states.

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

A. Time Space Tradeoff

In history, there has been many other interesting designs of reversible languages and instruction sets. However, current popular programming languages are all irreversible. Comparing with a irreversible Turing machine, an RTM has either a space overhead that proportional to computing time T or a computational overhead that sometimes can be exponential. The tradeoff between space and time is one of the most important issue in the theory of RTM. In the simplest g-segment trade off scheme [45, 46], we have

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

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

Here, T and S are the time and space usage on a irreversible Turing machine. ϵ 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 $k \geq 1$. In this section, we try to convince the readers that the overhead of reversible computing is not as terrible as people thought.

First, let $\epsilon \to 0$, there is not overhead in time. The space used by a RTM is bounded the caching strategy used in a traditional machine learning package that memorizes every inputs of primitives. Memorizing the inputs always make a primitive reversible since it does not discard any information. For deep neural networks, people used checkpointing trick to trade time with space [47]. This trick is also widely used in reversible programming [19]. Reversible programming just provides more alternatives to trade time and space.

Second, many computational overheads come from of the irreversibility of `/=` and `*=` operations. This part is not fundamental because reversible floating point instructions have already been designed [48, 49]. Using reversible floating point instructions may significant decrease the computation time and memory usage of a RTM. Even in current stage, the overhead can be mitigated by "arithematic uncomputing" without sacrificing reversibility as shown in the `iexp` example. We will review this point in Sec. V B.

Thrid, clever compiling can remove most overheads. Often, when we define a new reversible function, we allocate some ancillas at the beginning of the function and deallocate them through uncomputing at the end. The overhead comes from the uncomputing, in the worst case, the time used for uncomputing can be the same as the forward pass. In a hierarchical design, uncomputing can appear in every layer. 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

$$\log_2 \frac{Time(T)}{T}. \tag{9}$$

The computing time increases exponentially as the granularity increases. A cleverer compilation of a program can reduce the granularity. Given plenty of space, all uncomputing can be merged to avoid repeated works since the uncomputing of ancillas can be executed in any level of hierarchy.

At last, making reversible programming an eDSL rather than an 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. In our eDSL, we can just deallocate the memory irreversibly, i.e. trade energy with time.

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 reversible `goto` instruction, where the target instruction can be a `comefrom` instruction that specifing the postcondition. [50]

Arithmetic instructions should be redesigned to support better reversible programs. The major obstacle to exact reversibility programming is current floating point adders and multipliers used in our computing devices are not exactly reversible. There are proposals of reversible floating point adders and multipliers [48, 49, 51, 52] that introduces garbage bits to ensure reversibility, however this design with garbage qubits allocates new bits in each operaton, which is not too different from the information buffer approach [14]. With floating point numbers, rigorous reversible arithematic designs without using information buffer or garbage qubits is nearly impossible. Alternatives include fixed point numbers [53] and logarithmic numbers [54, 55], where logarithmic number system is reversible under $*=$ and $/=$. With these infrastructures, a reversible program can be executed without suffering from the rounding error.

Reversible programming is not nessesarily related to reversible hardwares. Reversible programs is a subset of irreversible programs, hence can be simulated efficiently on traditional CMOS devices [50]. Reversible programming just provides an alternative to execute on an energy efficient reversible hardwares. Reversible hardwares are not necessarily related to reversible gates such as the Toffoli gate and the Fredkin gate. Devices with the ability to recover signal energy can also be used to run reversible programs energy efficiently, which is known as the generalized reversible computing. [56, 57] In the following, we comment briefly on a special type of reversible device Quantum computer.

1. *Quantum Computers*

Building a universal quantum computer is difficult. The difficulty lies in the fact that, unlike a classical state, an unknown quantum state can not be copied. A quantum state in a environment suffers from decoherence, this underlines the simulation nature of quantum devices. Although there are proposals about quantum random access memory [58], they are difficult to implement, and are known to have many caveats [59]. 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.

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 devices and universal quantum computing devices. By introducing entanglement little by little, we can accelerate some basic components in reversible computing. For example, quantum Fourier transformation provides an interesting alternative to the reversible adders and multipliers by introducing one additional CPHASE gate, even though adders and multipliers are classical functions. [60] The development of reversible compiling theory will also have significant effect to quantum compiling.

C. Outlook

The reversible eDSL NiLang can be used to solve many existing scientific computing 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 [61, 62] and many other deciplines. However, manually derived backward rules for singular value decoposition and eigenvalue decomposition functions suffer from the gradient exploding problem [63–65]. Hopefully, the automatically generated backward rules do not have such problems.

Secondly, we can use it to break the memory wall problem. NiLang provides a systematic time-space trade off scheme through uncomputing. A successful related example is the memory efficient domain-specific AD engine in quantum simulator Yao [43]. This domain-specific AD engine is written in a reversible style and solved the memory bottleneck in variational quantum simulations. It also gives hitherto the best performance in differentiating quantum circuit parameters. Similarily, we can write memory efficient normalizing flow [66] with NiLang. Normalizing flow is a successful class of generative model in both computer vision [67] and quantum physics [68, 69]. Its building block bijector is required to be reversible and differentiable. We can use similar idea to differentiate reversible integrators [70, 71]. With reversible integrators, it should be possible to rewrite the control system in robotics [72] 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.

Thirdly, reversibility is a resource for training. For those who are interested in non-gradient based training. In Appendix C, we provide a self-consistency training strategy for reversible programs.

Latstly, the reversible IR is a good starting point to study quantum compiling. Most quantum programming language preassumes a classical coprocessor and use classical control flows [73] in universal quantum computing. However, 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.

- Add a compiling stage that decreases granularity and reduces uncomputing overheads.
- Better type inference. Current type inference assumes the variable types not changed by a function, which is not true. The type cast rule of a reversible function should be recorded somewhere to help type inference.
- Implement rigorous reversible floating point arithematics on hardwares to let the reversibility free from rounding error.
- Show the advantage of NiLang in parallel computing in handle asynchronious computing [74] and debugging with bidirectional move [75]. It is interesting to see how NiLang combines with other parts of Julia ecosystem like CUDAnative [76] and Debugger.

These improvements needs participation of people 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. Chris Rackauckas for helpful discussion on reversible integrators. Mike Innes for reviewing the comments about Zygote. Simon Byrne and Chen Zhao for helpful discussion on floating point and logarithmic numbers. 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].

[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] "Breaking the Memory Wall: The AI Bottleneck," `https://blog.semi.org/semi-news/breaking-the-memory-wall-the-ai-bottleneck`.
[8] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," (2015), arXiv:1506.00019 [cs.LG].
[9] 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.
[10] J. Bettencourt, M. J. Johnson, and D. Duvenaud, (2019).
[11] C. M. Elliott, ACM Sigplan Notices **44**, 191 (2009).
[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] 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.
[15] A. N. Gomez, M. Ren, R. Urtasun, and R. B. Grosse, in *Advances in Neural Information Processing Systems 30*, edited by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Curran Associates, Inc., 2017) pp. 2214–2224.
[16] J.-H. Jacobsen, A. W. Smeulders, and E. Oyallon, in *International Conference on Learning Representations* (2018).
[17] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, arXiv preprint arXiv:1209.5145 (2012).
[18] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, SIAM Review **59**, 65–98 (2017).
[19] K. S. Perumalla, *Introduction to reversible computing* (Chapman and Hall/CRC, 2013).
[20] M. P. Frank, IEEE Spectrum **54**, 32–37 (2017).
[21] C. Lutz, "Janus: a time-reversible language," (1986), *Letter to R. Landauer*.
[22] M. P. Frank, *The R programming language and compiler*, Tech. Rep. (MIT Reversible Computing Project Memo, 1997).
[23] I. Lanese, N. Nishida, A. Palacios, and G. Vidal, Journal of Logical and Algebraic Methods in Programming **100**, 71–97 (2018).
[24] T. Haulund, "Design and implementation of a reversible object-oriented programming language," (2017), arXiv:1707.07845 [cs.PL].
[25] M. P. Frank and T. F. Knight Jr, *Reversibility for efficient computing*, Ph.D. thesis, Massachusetts Institute of Technology, Dept. of Electrical Engineering and . . . (1999).
[26] J. G. Koller and W. C. Athas, in *Workshop on Physics and Computation* (1992) pp. 267–270.
[27] R. C. Merkle, R. A. Freitas, T. Hogg, T. E. Moore, M. S. Moses, and J. Ryley, Journal of Mechanisms and Robotics **10** (2018), 10.1115/1.4041209.
[28] K. Likharev, IEEE Transactions on Magnetics **13**, 242 (1977).
[29] V. K. Semenov, G. V. Danilov, and D. V. Averin, IEEE Transactions on Applied Superconductivity **13**, 938 (2003).
[30] R. Landauer, IBM journal of research and development **5**, 183 (1961).
[31] E. P. DeBenedictis, J. K. Mee, and M. P. Frank, Computer **50**, 76 (2017).
[32] G. H. Golub and C. F. Van Loan, *Matrix computations*, Vol. 3 (JHU press, 2012).
[33] "MLStyle.jl," `https://github.com/thautwarm/MLStyle.jl`.
[34] R. Orús, Annals of Physics **349**, 117–158 (2014).
[35] A. McInerney, *First steps in differential geometry* (Springer, 2015).
[36] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in julia," (2016), arXiv:1607.07892 [cs.MS].
[37] "Paper's Github Repository," `https://github.com/GiggleLiu/nilangpaper/tree/master/codes`.
[38] J. Martens, I. Sutskever, and K. Swersky, "Estimating the hessian by back-propagating curvature," (2012), arXiv:1206.6464 [cs.LG].
[39] 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.
[40] M. Arjovsky, A. Shah, and Y. Bengio, CoRR **abs/1511.06464** (2015), arXiv:1511.06464.
[41] 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].
[42] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. A. Skirlo, M. Tegmark, and M. Soljacic, CoRR **abs/1612.05231** (2016), arXiv:1612.05231.
[43] 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].
[44] C.-K. LI, R. ROBERTS, and X. YIN, International Journal of Quantum Information **11**, 1350015 (2013).
[45] C. H. Bennett, SIAM Journal on Computing **18**, 766 (1989).
[46] R. Y. Levine and A. T. Sherman, SIAM Journal on Computing **19**, 673 (1990).
[47] T. Chen, B. Xu, C. Zhang, and C. Guestrin, CoRR **abs/1604.06174** (2016), arXiv:1604.06174.
[48] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *10th IEEE International Conference on Nanotechnology* (2010) pp. 233–237.
[49] M. Nachtigal, H. Thapliyal, and N. Ranganathan, in *2011 11th IEEE International Conference on Nanotechnology* (2011) pp. 451–456.
[50] C. J. Vieri, *Reversible Computer Engineering and Architecture*, Ph.D. thesis, Cambridge, MA, USA (1999), aAI0800892.
[51] 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].
[52] T. Häner, M. Soeken, M. Roetteler, and K. M. Svore, "Quantum circuits for floating-point arithmetic," (2018), arXiv:1807.02023 [quant-ph].
[53] "FixedPointNumbers.jl," `https://github.com/JuliaMath/FixedPointNumbers.jl`.
[54] F. J. Taylor, R. Gill, J. Joseph, and J. Radke, IEEE Transactions on Computers **37**, 190 (1988).
[55] "LogarithmicNumbers.jl," `https://github.com/cjdoris/LogarithmicNumbers.jl`.
[56] M. P. Frank, in *35th International Symposium on Multiple-Valued Logic (ISMVL'05)* (2005) pp. 168–185.
[57] M. P. Frank, in *Reversible Computation*, edited by I. Phillips and H. Rahaman (Springer International Publishing, Cham,

2017) pp. 19–34.
[58] V. Giovannetti, S. Lloyd, and L. Maccone, Physical Review Letters **100** (2008), 10.1103/physrevlett.100.160501.
[59] S. Aaronson, Nature Physics **11**, 291 (2015).
[60] L. Ruiz-Perez and J. C. Garcia-Escartin, Quantum Information Processing **16** (2017), 10.1007/s11128-017-1603-1.
[61] J.-G. L. Hao Xie and L. Wang, arXiv:2001.04121.
[62] H.-J. Liao, J.-G. Liu, L. Wang, and T. Xiang, Physical Review X **9** (2019), 10.1103/physrevx.9.031041.
[63] M. Seeger, A. Hetzel, Z. Dai, E. Meissner, and N. D. Lawrence, "Auto-differentiating linear algebra," (2017), arXiv:1710.08717 [cs.MS].
[64] Z.-Q. Wan and S.-X. Zhang, "Automatic differentiation for complex valued svd," (2019), arXiv:1909.02659 [math.NA].
[65] C. Hubig, "Use and implementation of autodifferentiation in tensor network methods with complex scalars," (2019), arXiv:1907.13422 [cond-mat.str-el].
[66] I. Kobyzev, S. Prince, and M. A. Brubaker, "Normalizing flows: An introduction and review of current methods," (2019), arXiv:1908.09257 [stat.ML].
[67] D. P. Kingma and P. Dhariwal, 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. 10215–10224.
[68] L. Dinh, J. Sohl-Dickstein, and S. Bengio, "Density estimation using real nvp," (2016), arXiv:1605.08803 [cs.LG].
[69] S.-H. Li and L. Wang, Physical Review Letters **121** (2018), 10.1103/physrevlett.121.260601.
[70] P. Hut, J. Makino, and S. McMillan, The Astrophysical Journal **443**, L93 (1995).
[71] D. N. Laikov, Theoretical Chemistry Accounts **137** (2018), 10.1007/s00214-018-2344-7.
[72] M. Giftthaler, M. Neunert, M. Stäuble, M. Frigerio, C. Semini, and J. Buchli, Advanced Robotics **31**, 1225–1237 (2017).
[73] 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.
[74] D. R. Jefferson, ACM Transactions on Programming Languages and Systems (TOPLAS) **7**, 404 (1985).
[75] B. Boothe, ACM SIGPLAN Notices **35**, 299 (2000).
[76] T. Besard, C. Foket, and B. D. Sutter, CoRR **abs/1712.03112** (2017), arXiv:1712.03112.
[77] 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 column. The following is a list of terminologies used in the definition of grammar

- *ident*, symbols
- *num*, numbers
- ϵ, 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⟩ ::= ϵ
          | ⟨Stmt⟩
          | ⟨Stmts⟩ ⟨Stmt⟩
⟨Stmt⟩ ::= ⟨BlockStmt⟩
          | ⟨IfStmt⟩
          | ⟨WhileStmt⟩
          | ⟨ForStmt⟩
          | ⟨InstrStmt⟩
          | ⟨RevStmt⟩
          | ⟨AncillaStmt⟩
          | ⟨TypecastStmt⟩
          | ⟨@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 | π
⟨InstrBinOp⟩ ::= += | -= | ⊻=
⟨InstrTrailer⟩ ::= [.] ( [⟨DataViews⟩] )
⟨InstrStmt⟩ ::= ⟨DataView⟩ ⟨InstrBinOp⟩ ident [⟨InstrTrailer⟩]
⟨RevStmt⟩ ::= ~ ⟨Stmt⟩
⟨AncillaStmt⟩ ::= ident ← JuliaExpr
⟨TypecastStmt⟩ ::= ( JuliaExpr => JuliaExpr ) ( ident )
⟨@routine⟩ ::= @routine ident ⟨Stmt⟩
⟨@safe⟩ ::= @safe JuliaExpr
⟨DataViews⟩ ::= ϵ
          | ⟨DataView⟩
          | ⟨DataViews⟩ , ⟨DataView⟩
          | ⟨DataViews⟩ , ⟨DataView⟩ ...
⟨DataView⟩ ::= ⟨DataView⟩ [ JuliaExpr ]
          | ⟨DataView⟩ . ident
          | JuliaExpr ( ⟨DataView⟩ )
          | ⟨DataView⟩ '
          | - ⟨DataView⟩
          | ⟨Constant⟩
          | ident
```

Appendix B: Instruction Table

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

instruction	translated	symbol
$y \mathrel{+}= f(args...)$	`PlusEq(f)(args...)`	⊕
$y \mathrel{-}= f(args...)$	`MinusEq(f)(args...)`	⊖
$y \veebar= f(args...)$	`XorEq(f)(args...)`	⊙

Table II. Instructions and their interpretation in NiLang.

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

Table III. A collection of reversible instructions, "." is the broadcasting operations in Julia.

Appendix C: Learn by consistency

Consider a training that with input $\mathbf{x}^*$ and output $\mathbf{y}^*$, find a set of parameters $\mathbf{p}_x$ that satisfy $\mathbf{y}^* = f(\mathbf{x}^*, \mathbf{p}_x)$. In traditional machine learning, we define a loss $\mathcal{L} = \mathrm{dist}(\mathbf{y}^*, f(\mathbf{x}^*, \mathbf{p}_x))$ and minimize it with gradient $\frac{\partial L}{\partial \mathbf{p}_x}$. 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 $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$, where $\mathbf{p}_x$ and $\mathbf{p}_y$ are "parameter" spaces on the input side and output side. The algorithm can be summarized as

Algorithm 2: Learn by consistency

Result: $\mathbf{p}_x$
Initialize $\mathbf{x}$ to $\mathbf{x}^*$, parameter space $\mathbf{p}_x$ to random.
if $\mathbf{p}_y$ *is* `null` **then**
  $\mathbf{x}, \mathbf{p}_x = f_r^{-1}(\mathbf{y}^*)$
else
  $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$
  while $\mathbf{y} \not\approx \mathbf{y}^*$ **do**
    $\mathbf{y} = \mathbf{y}^*$
    $\mathbf{x}, \mathbf{p}_x = f_r^{-1}(\mathbf{y}, \mathbf{p}_y)$.
    $\mathbf{x} = \mathbf{x}^*$
    $\mathbf{y}, \mathbf{p}_y = f_r(\mathbf{x}, \mathbf{p}_x)$

Here, `parameter`$(\cdot)$ is a function for taking the parameter space. This algorithm utilizes the self-consistency relation

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

Similar idea of training by consistency is used in self-consistent meanfield theory [77] in physics. Finding the self-consistent relation is crucial to a self-consistency based training. Here, the reversibility provides a natural self-consistency relation. However, it is not a silver bullet, 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` uncomputes 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, parameters in `f1` does change at all. The training of `f1` fails because this function actually computes $\texttt{f1}(y, x, p) = y + e^{(p+x)} - e^x, x, x + p$, where the training parameter p is completely determined by the parameter 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, $\texttt{f2}(y, x, p) = y + e^{(p+x)} - e^x, \tilde{0}, x + p$, the output parameters $\tilde{0} \cup x + p$ can not uniquely determine input parameters p and x. Here, we use $\tilde{0}$ to denote the zero with rounding error.

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

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

Proof. The above example reveals a fact that the training can not work when output parameters completely determines input parameters. In other words, if $S(\mathbf{p}_x|\mathbf{p}_y) = 0$, the traning can not work.

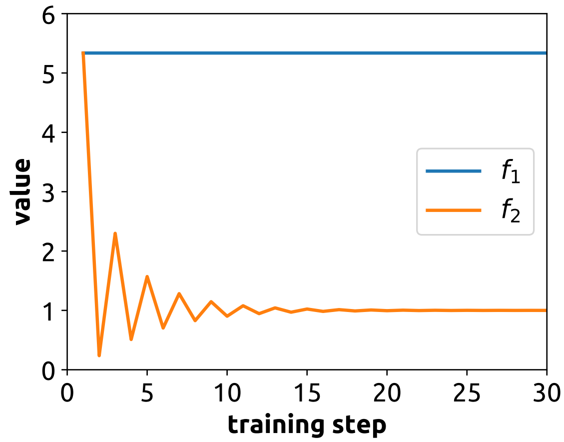

Figure 5. The output value `y!` as a function of self-consistent training step.

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

The third line uses the bijectivity $S(\mathbf{x} \cup \mathbf{p}_x) = S(\mathbf{y} \cup \mathbf{p}_y)$. This inequality shows that when the parameter space on the output side satisfies $S(\mathbf{y}|\mathbf{p}_y) = 0$, i.e. contains all information to determine the output field, the input parameters are also completely determined by this parameter space, hence traning can not work. □

In the above example, it corresponds to the case $S\left(e^{(x+y)-e^x}|x \cup x + y\right) = 0$ in $f1$. The solution is to remove the information redundancy in output parameter space through uncomputing as shown in $f2$.