Instruction level automatic differentiation on a reversible Turing machine

Jin-Guo Liu[1,*] and Hong-Xuan Zhao-Wang[2]

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

This paper considers instruction level differential programming, i.e. knowing only the backward rule of basic instructions like +, -, * and /, differentiate a program with proper performance. We will review briefly why instruction level automatic differentiation is hard for current machine learning package even for a source to source automatic differentiation package. Then we propose a reversible Turing machine implementation to achieve instruction level automatic differentiation.

I. AUTOMATIC DIFFERENTIATION

There are two basic modes of automatic differentiation [**?**], the tangent mode[**?**] and the adjoint mode. Consider a multi-in ($\vec{x}$) multi-out ($\vec{y}$) function f, the tangent mode computes a column of its Jacobian $\frac{\partial \vec{y}}{\partial x_i}$ efficiently, where x_i is single input variable and $\vec{y}$ is multiple output variables. Whereas the adjoint mode computes a row of Jacobian $\frac{\partial y_i}{\partial \vec{x}}$ efficiently.

Most popular automatic differentiation package implements the adjoint mode differentation, because they are computational more efficient in most optimization applications, where the output loss is always a scalar. Implementing adjoint mode AD requires tracing a program and its intermediate state backward, which requires storing extra information

1. computational graph,
2. and intermediate result caching.

The computational graph is a DAG that stores function calls from inputs to results. Intermediate results are usually the input variable of a function, it is nessesary to compute the adjoint of the function. In Pytorch [1] and Flux [**?**], every variable (tensor) has a tracker field that stores its parent (data and function that generate this variable) information in computational graph and intermediate state. TensorFlow [2] implements a static computational graph before actual computing happens. Source to source automatic differentiation package Zygote [3] use a intermediate representation SSA as the computational graph, so that it can back propagate over a native julia code. Still, intermediate caching is nessesary.

Since every computational process is compiled to instructs, these instructions is a natural sequential computational graphs. These instructions are from a finite set of '+', '-', '*', '/', conditional jump statements et. al. With instruction level computational graph, we do not need to define primitives like `exp`, or even linear algebras functions like singular value decomposition [] and eigenvalue decomposition. where the manually derived backwards rule still faces the degenerate spectrum problem (gradients explodes), instruction level AD will return reasonable gradients. With instruction level AD, people don't worry about inplace functions, which may be a huge problem in traditional approaches. We can back propagate over a quantum simulator, where all instructions are reversible two level unitaries (i.e. Jacobian rotation). We don't need extra effort to learn meta parameters. [] Neural ODE is much easier to design [4].

However, people don't use instructions computational graph for practical reasons. The cost of memorizing the computational graph and intermediate caching kills the performance for more than two orders (as we will show latter). A even more serious problem is the memory consumption for caching intermediate results increases linearly as time. Even in many traditional deep network like recurrent neural network and residual neural networks, where the depth is only several thousand, this memory cost can be a nightmare.

In this paper, we introduce a high performance instruction level AD by making a program time reversible. Making use of reversibility like information buffer [5] can reduce the memory allocations in recurrent neural network [6] and residual neural networks [7]. However, the use of reversiblity in these cases are not general purposed. We develop a domain specific language (DSL) in `Julia` that implements reversible Turing machine. In the past, the reversible Turing machine is not a widely used computational model for having either polynomial overhead in computational time or additional memory cost that propotional to computational time. There has been some prototypes of reversible languages like `Janus` [8], where reversible control flows are introduced. Our DSL borrows the design of reversible control flow, meanwhile provides abstraction and memory management. With these additional features, differentiating over a general program requires less than 100 lines. In this paper, we show that in many useful applications, the computational overhead is just a constant factor. Only in some worst cases, it is equivalent to a traditional machine learning framework that cache every input.

II. REVERSIBLE FUNCTIONS

In a mordern 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 informations like return address and saved memory

* cacate0129@iphy.ac.cn

segments, local variables and working stack. After each call, the function clears the input arguments, function frame, local variables and working stack and only stores the return value. In the invertible programming style, this kind of design pattern is nolonger the best practise, the local information of a function can not be easily emptied immediately after a function call. Sometimes discarding local information may ruin reversibility, and sometimes, active deallocation of memory may incurr bad performance. Hence, a reversible instruction/function call in NiLang is a mapping between a same set of symbols.

NiLang is a reversible DSL in Julia that simulates reversible Turing machine without actual hardware or even instruction level support. The grammar is shown in Appendix A. A function definition is composed of function head and statements, the interprecation of a reversible function to native Julia language consists three stages. The first stage preprocess human input to a reversible IR. It checks human input and removes redundancy in grammar. To be specific

1. adds missing `@deanc` to make sure `@anc` and `@deanc` statements appear in pairs,
2. expand `@routine` macro,
3. expand the symbol in postcondition field as precondition.

Here, the macro `@anc <a> = <expr>` binds `<a>` to an initial value specified by `<expr>`, while `@deanc <a> = <expr>` deallocates the symbol, before doing that, it checks `<a>` is restored to its initial value. This underlines the difference between irreversible assign statements and reversible ancilla statements. `@routine <name> [(<Stmts>)]` is a statements recorder, when it is followed by statements, it record the statements to variable . When

The macro `@instr` assign the output of a function to the argument list of a function. Hence, the values are changed while the symbol table is not changed. Here, the statement `out! += x * y` calls instruction $\oplus(*)(out!, x, y)$, which means accumulate a product of two variables to target symbol. To drop a symbol, we can use the inverse process of `@anc`, the `@deanc`

The second stage transforms this reversible IR to its reversed version.

It did nothing but errors on nonreversible memory deallocation. Only if the compiler knows the value deterministically, the value can be deallocated safely.

In a function definition, `@anc` and `@deanc` always appear in pairs. In a function definition, `@deanc` is often added automatically. We don't have a concept of stack in our design, ancilla data plays a similar role. Every variable has finite life cycle, a function is a natural manager of an ancilla's life cycle. When a function is called, it borns, when the function is ended, it dies. The program appears in a hyrachical pattern. The lowest level is instructions like $+, -, *, /$, second lowest level we have primitive functions like $\exp, \sin$. The highest level may be an application with a lot global variables. The lower the level, the shorter an ancilla's life. We will revisit this point in Sec. III A.

statement	**reverse**
`<f>(<args>)`	`(~<f>)(<args>)`
`<out!> += <f>(<args>)`	`<out!> -= <f>(<args>)`
`<out!> .+= <f>.(<args>)`	`<out!> .-= <f>.(<args>)`
`<out!> ⊻= <f>(<args>)`	`<out!> ⊻= <f>(<args>)`
`<out!> .⊻= <f>(<args>)`	`<out!> .⊻= <f>(<args>)`
`@anc <a> = <expr>`	`@deanc <a> = <expr>`

`begin`
`  <Stmts>`
`end`
	`begin`
	`  ~(<Stmts>)`
	`end`

`if (<pre>, <post>)`
`  <Stmts>`
`else`
`  <Stmts>`
`end`
	`if (<post>, <pre>)`
	`  ~(<Stmts>)`
	`else`
	`  ~(<Stmts>)`
	`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.

The third stage is translating this reversible IR to native Julia code. It adds `@instr` before each instruction and function call statement, attach a return statement after the function definition which returns the modified input variables as the output. It also adds statements to check the consistency between preconditions and postconditions to ensure reversibility.

Notice reversible Turing machine is a subset of irreversible Turing machine, reversible statements has less allowed statements. we forbit `return` and assign statements in our language design, instead, return the inputs as outputs. Combine it with `@instr` that assigns each output to each input, we simulate a mutable operations. Besides putting restrictions, it also allows user putting additional informations in control flows to help reverse the program. A post condition is a boolean expression that evaluated after the controlled body being executed. For example, to get the first Fibonacci number that is greater or equal to 0, the traditional approach is

```
function fib100()
    n = 0.0
    while fib(n) < 100
        n += 1.0
    end
    return n
end
```

The reversible version of this function is

```
@i function rfib100(n)
    @safe @assert n == 0
    while (fib(n) < 100, n != 0)
        n += identity(1.0)
    end
end
```

Here, `fib` and `rfib` are defined in Appendix C. The reversible `while` statement contains two statements, the precondition and postcondition. Before entering the while statement, the program check the postcondition to make sure it is false. After each iteration, postcondition returns true. The inverse function exchanges the precondition and postcondition so that the repeatition of loop body is not changed. The `@safe` macro can be followed by an arbituary statement, it allows user to use external statements that does not break reversibility. `@safe @show var` is often used for debugging.

A. Types and Views

Functional programming style fits well with a reversible DSL. A constructor is also a reversible function, it packs one or more data into one. The reverse function is a deconstructor, it does not deallocate memory directly but unpacks data. In current version of NiLang, constructor ususally attach a new field to an existing data (or the kernel), the initial value of this new field can be computed or uncomputed by its kernel so that guarantes reversibility.

To access a field of a user defined type, we introduce the concept of data view. A data view of a data can be data itself, a field of its view, an array element of its view, or a bijective mapping of its view.

```
julia> using NiLang, NiLang.AD

julia> x = 0.5
0.5

julia> @instr GVar(x)

julia> @instr x.x += identity(0.4)

julia> x
GVar(0.9, 0.0)

julia> @instr (~GVar)(x)

julia> x
0.9
```

Here `GVar` is a immutable type, this is why '@instr' is nessesary to change a field of a immutable type.

For arrays, we

```
julia> x = randn(3)
3-element Array{Float64,1}:
 -0.14307008145820874
  0.40510133517552077
  0.8444959804268539

julia> @instr GVar.(x)

julia> @instr x[2].x += identity(0.4)

julia> x
3-element Array{GVar{Float64,Float64},1}:
 GVar(-0.14307008145820874, 0.0)
 GVar(0.8051013351755207, 0.0)
 GVar(0.8444959804268539, 0.0)
```

Broadcasting is supported. To avoid possible confusing, tuple indexing is forbidden delebrately, one can use `tget(tuple, 2)` to get the second element of a tuple.

B. Instructions

Not only invertibility, but also the stability of gradient itself, requires reversible instruction support, otherwise invertibility can be easily ruined by rounding errors. Using information buffer in multiplication operations [5] in an approach to enforce invertibility in a memory efficient way. Invertibility has been studies in the cross field of computer science and physics a lot between 1980 and 2010. The motivation is saving energy. Carlin deviced SRCL logic family from Pendulum instruction set architecture (PISA) [9] for a invertible programming device. Notablely, the control flow defined in this NiLang can be compiled to reversible instructions, where the conditional `goto` instruction is designed in such a way that the target instruction is a `comefrom` statement which specifies the post condition.

III. TAYLOR PROPAGATION ON A REVERSIBLE TURING MACHINE

Taylor propagation is exponentially (as the order) more efficient in obtaining higher order gradients than differentiating lower order gradients recursively. The later requires traversing the computational graph repeatedly. In JAX, in order to support Taylor propagation, the propagation rules for part of primitives manually defined. The exhaused support requires much more effort than the first order gradient propagation. Instruction level automatic differentiation is more flexible in obtaining higher order gradients like Hessian.

A. First order gradient

Given a node $\vec{y} = f(\vec{x})$ in a computational graph, we can propagate the Jacobians in tangent mode like

$$J^O_{x_i} = J^O_{y_j} J^{y_j}_{x_i} \tag{1}$$

and the adjoint mode

$$J^{y_j}_I = J^{y_j}_{x_i} J^{x_i}_I \tag{2}$$

Here, I is the inputs and O is the outputs. Einstein's notation is used so that duplicated indices are summed over. The computational process can be described in tensor network language as shown in Fig. 1.

In reversible programming, we have the following implementation

1. The program runs forward and computes outputs,
2. call constructor `GVar` that transfer a floating point number type to a `GVar` type.
3. Add 1 to the gradient field of the loss variable,
4. The program uncomputes outputs and recover all inputs, gradients are the `grad` fields of input variables.

Here, `GVar` is a "reversible type". `GVar(x)` wraps a variable into a GVar, which attaches a zero gradient field to a variable just like the dual number in tangent mode automatic differentiation. Its inverse *GVar* deallocate the gradient field safely and returns its value field. Here, "safely" means it will check the gradient field to make sure it is in 0 state. When a `instruct` meets a `GVar`, besides computing its value field $value(x) \leftarrow f^{-1}(value(y))$, it also updates the gradient field $grad(x) = f^{-1}(value(x), grad(y))$. Since f^{-1} is bijective, J^y_x can be easily obtained by inspecting its inverse function f. The final output is stored in the gradient field, when then gradient is not used anymore, the faithful reversible programming style to compute gradients would be uncomputing the whole process to obtain gradient, which increases the hyrachy by 1. Whenever the hyrachy increase by 1, the computational overhead doubles comparing with its irreversible counter part.

B. Second order gradient

The second order gradient can also be back propagated

$$\begin{aligned} H^f_{y_L, y'_L} &= 0 \\ H^f_{y_{i-1}, y'_{i-1}} &= J^{y_i}_{y_{i-1}} H^f_{y_i, y'_i} J^{y'_i}_{y'_{i-1}} + J^f_{y_i} H^{y_i}_{y_{i-1}, y'_{i-1}} \end{aligned} \tag{3}$$

In tensor network language, it can be represented as in Fig. 1.

This approach can be easily extended to higher orders, or taylor propagation. However, this is not the widely adopted approach to compute higher order gradients. Althought back-propagating higher order gradients directly is exponentially faster than back propagating the computational graph for computing lower order gradients for computing higher order gradients, one has to extending the backward rules for each primitive rather than reusing existing ones. Here, we emphasis that with instruction level AD, rewritting backward rules for primitives turns out to be not so difficult.

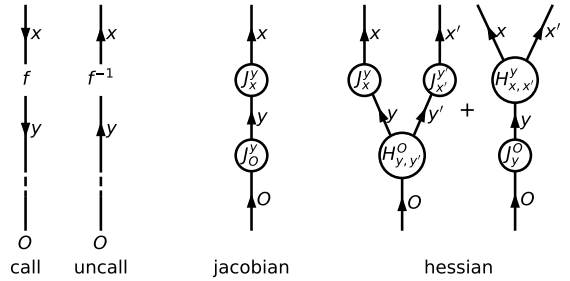

Figure 1. Adjoint rules for Jacobians and Hessians in tensor network language.

C. Gradient on ancilla problem

Ancilla can also carry gradients during computation, sometimes these gradients can not be uncomputed even if their parents can be uncomputed regoriously. In these case, we simply "drop" the gradient field instead of raising an error. In this subsection, we prove doing this is safe, i.e. does not have effect on rest parts of program.

Ancilla is the fixed point of a function, which means

$$\begin{aligned} &b, y \leftarrow f(x, a) \text{ , where } b == a \\ &\frac{\partial b}{\partial x} = 0 \end{aligned} \tag{4}$$

During the computation, the gradient field does not have any effect to the `value` field of variables. The key question is will the loss of gradient part in ancilla affect the reversibility of the gradient part of argument variables. The gradient of argument variable is defined as $\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y}\frac{\partial y}{\partial x} + \frac{\partial L}{\partial b}\frac{\partial b}{\partial x}$, where the second term vanish naturally.

IV. EXAMPLES

A. exp function

The following codes implements the exp function in a reversible way

```
using NiLang, NiLang.AD

@i function iexp(out!, 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)

    out! += 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)
            # speudo inverse
            anc1 -= anc2 / x
            anc2 -= anc3 * iplus
            SWAP(anc1, anc3)
        end
    end

    out! += identity(expout)

    ~@routine r1
end
```

The definition of SWAP instruction can be found in Appendix B. The two lines bellow the comment `# pseudo inverse` "uncompute" variables `anc1` and `anc2` to avalue very close to zero. Notice * and / are not exactly dual to each other. It is reasonable to assume this inexact uncomputing will cause negligible error on final output, but harms reversibility. In the latter case, error accumulates in the whole program. In the second for loop inside the inverse notation ~, we uncompute all ancilla bits rigorously. The `while` statement takes two conditions, the precondition and postconditoin. Precondition `val(anc1) > atol` indicates when to break the forward pass and post condition `!isapprox(iplus, 0.0)` indicates when to break the backward pass.

The appendix gives another example of QR decomposition.

To obtain the gradient, one can wrap the loss with `Loss` type.

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

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

julia> grad(x)
4.9530324244260555

julia> out!, x = 0.0, 1.6
(0.0, 1.6)

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

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

`iexp'` returns an object of type `Gradtypeof(exp)`, it is a reversible function. It is so short that we present the function defintion 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

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

    ~@routine getiloss
end
```

The program first checks the input parameters and locate the loss variable. Then `Loss` unwraps the loss variable, the location of loss variable is transferred to the ancilla `iloss` of integer type.

Due to the non-locality of Hessians, we use a global tape for Hessian propagation. Whenever a new variable is created, the tape allocates a larger ring. It is global to ease the memory allocations of ancillas, the Hessian in ancilla is important to reach correct result.

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

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

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

The final result can be obtained by calling a global function `collect_hessian`.

B. QR decomposition

Not only simple functions, linear algebra functions

```
@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
```

where `idot` and `inorm2` 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 gradient of this naive implementation of QR decomposition is correct

```
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 `NiLangCore.ADCore`.

C. Unitary Matrices

Recurrent networks with a unitary parametrized network ease the gradient exploding and vanishing problem [10–12]. Among different parametrization schemes, the most elegant one is [12], which parametrized the unitary matrix with two-level unitary operations, any unitary matrix of size $N \times N$ can be parametrized by $k = N(N-1)/2$ two level unitary matrices [13]. All these two-level unitary matrices can be applied in $O(1)$ time as a two register instruction. Hence a real unitary matrix can be parametrized compactly by k rotation angles, each represents a rotation operations between datas in

two target parameters.

```
@i function umm!(x, , Nin::Int, Nout::Int)
    @anc k = 0
    for j=1:Nout
        for i=Nin-1:-1:j
            k += identity(1)
            ROT(x[i], x[i+1], [k])
        end
    end

    # uncompute k
    for j=1:Nout
        for i=Nin-1:-1:j
            k -= identity(1)
        end
    end
end
```

Here, `loop j` means excuting the loop body for `j` times, it gurantes invertibility. Loop and branching can be implemented in a more regorious way [9] from instruction level, however it is still an open question how to implelement loops with instruction level invertibility without compiling technic. In the rotation instruction, `z[k]` is the rotation angle, which represents rotating data in target registers by an angle of $\theta =$ `z[k]`$*\pi$.

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{5}$$

1. *The gradient*

Its backward rule is

$$\begin{aligned} \bar{\theta} &= \sum \frac{\partial R(\theta)}{\partial \theta} \odot (\bar{y} x^T) \\ &= \mathrm{Tr}\left[\frac{\partial R(\theta)}{\partial \theta}^T \bar{y} x^T \right] \\ &= \mathrm{Tr}\left[R\left(\frac{\pi}{2} - \theta\right) \bar{y} x^T \right] \end{aligned} \tag{6}$$

The resulting instruction is

```
-z[k]
z[k] + pi/2
rot(gx[i], gx[ip], theta[k])
gtheta + gx[i]*x[i]
gtheta + gx[ip]*x[ip]
theta[k] - pi/2
-theta[k]

rot(gx[i], gx[ip], -pi/2)
```

In this way, the gradient is defined in a closure form.

V. TRAINING BY CONSISTENCY

We compute the output as

$$y = f(x), \tag{7}$$

and we have an expected output $\hat{y}$. In traditional machine learning, we define a loss and minimize it. However, this is not how human brain works. [**?**] Since the $\hat{y}$ is different with y, the network start to "think", if the output is $\hat{y}$, what should the input (including network parameters) be? So, we feed $\hat{y}$ back to the output end of network, and inverse the tape, with the runtime information generated by the input image. The network will compute a different value to network parameters, then the network feel "confused", and chose a value between old and new values.

Let's look at the following example

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

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

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

Loss function `f1` and `f2` computes $f(x, y) = e^{(y+x)} - e^x$ and stores the output in a new memory `out!`, the target is to find a y that make the output x equal to the target value 1. After 200 steps training, y runs into the fixed point and x is equal to 1 upto machine precision.

This training is similar the recursive convergence method that widely used in mathematics and physics. However, in programming language but it has a loophole, it is vulnarable to loss of injectivity. For example, if the result is accumulated to another variable $z+ = x$, and take z as the loss, then the loss can not accumulated to the target value correctly since the inverse of above operation is $z- = x$ and x is unchanged in the inverse run. This is because the redundancy introduced in operation $z = z + e^x$, which keeps both x and

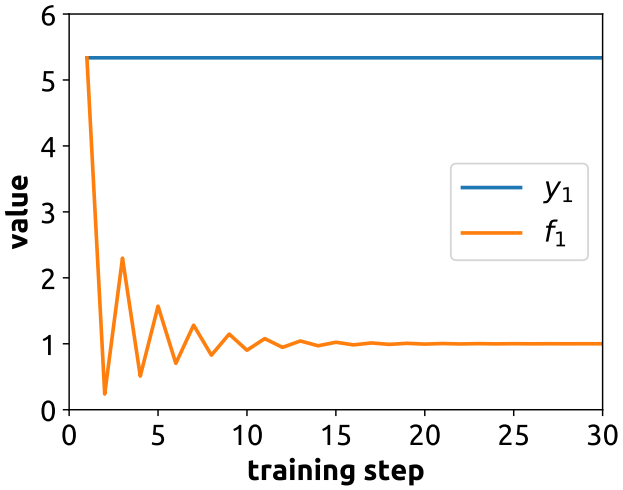

Figure 2. Target function value as a function of self-consistent training step.

z, the change of z shifting z directly is apparently a lazier approach to fit z with target value 1, to figure out the correct causality, corresponding change in output side to x and y are required. This redundancy can be detected by computing the entanglement entropy (or mutual information) between two output datas. We call this type of invertibility inference, which should be avoided in this type of training. Then how shall we calculate $f(x) = e^x$ like functions, which are not partly invertible from the numeric perspective? Here we introduce the notion of "weak invertibility"

$$@asserty == 0 \tag{8}$$
$$\texttt{fl}(y)+ = exp(\texttt{fl}(x)) \tag{9}$$
$$\texttt{fl}(x)- = log(y) \tag{10}$$

We can keep x to ensure numerical invertibility, meanwhile erase most informations inside it. Since x is effectively 0, thus no redundancy (or entanglement entropy) in output, in this way, the above training is still valid.

A. Compactness of data

Measure of compactness by entanglement entropy. In invertible programming, we define the entanglement between output space garbage space as the compactness of information.

If the output information is compact, then we have $(x, x_a) \to (y, y_a)$

mutual information is

$$I(x,y) = S(x,y) - S(x) - S(y) \tag{11}$$
$$S(x,x_a) = S(y,y_a) \tag{12}$$
$$I(x,x_a) = 0 \tag{13}$$
$$I(y,y_a) = S(y,y_a) - S(y_a) \tag{14}$$

VI. TIME SPACE TRADEOFF

Comparing with irreversible programming, reversible programming consumes more memory. Data have different life cycles, some are more persistent, like database, some are transiant like an ancilla variable in register that live only in several clock cycle.

VII. A INVERTIBLE PROGRAMMING STYLE

VIII. HARDWARES

A. Traditional Invertible Computing Devices

A reversible computer, by no means refer to a computing device that every instruction or program is reversible. A better definition would be, a reversible computing device reserves the right to retract energy through uncomputing.

Like quantum computer, it should be able to reset to qubits 0. The reset operation, sometimes can be expensive. In a quantum device, this reset operation can be done by dissipation and dicoherence (i.e. interact with environment and get thermalized). Alternatively, a immediate feedback loop can be introduced to a quantum device, where a classical computer is regarded as the dissipation source.

1. *Pendulum Design*

copy pasted begin

It is possible to design and fabricate a fully reversible processor using resources which are comparable to a conventional microprocessor. Existing CAD tools and silicon technology are sufficient for reversible computer design. Pendulum author demonstrated this by designing such a processor and fabricating and testing it in a commercially available CMOS process.

copy pasted end

2. *Margolus's Billiard Ball Model Cellular Automaton (BBMCA)*

The chip is known as Flattop, not very convenient to program, but is simple, universal, reversible, and scalable.

IX. DISCUSSION

One should notice the memory advantage of reversible programming to machine learning does comes from reversibility itself, but from a better data tracking strategy inspired from invertible programming. Normally, a reversible program is not as memory efficient as its irreversible couterpart due to the additional requirement of no information loss. A naive approach that keeping track of all informations will cost an additional space $O(T)$, where T stands for the excution time

in a irreversible TM, the longer the program runs, the larger the memory usage is. This is exactly the approach to keeping reversibility in most machine learning packages in the market. The point it, an reversible Turing Machine is able to trade space with time. In some cases, it may cause polynomial overhead than its irreversible counterpart.

A. Rounding errors

$$a = a\cos(k\pi) - b\sin(k\pi) \tag{15}$$
$$b = a\sin(k\pi) + b\cos(k\pi) \tag{16}$$

$$\begin{aligned}\epsilon_a &= -a\sin(\theta)\theta\epsilon - b\cos(\theta)\theta\epsilon + a\epsilon\cos(\theta) - b\epsilon\sin(\theta) \\ &\sim \max(a,b)\epsilon\end{aligned} \tag{17}$$

The error accumulates linearly as the number of floating point operations, for a $N\times N$ unitary matrix, each number is operated N times. In a double precision computation, the rounding errors in unitary matrix multiplication will probably not have a substaintial effect on resersibility.

B. Performance

Todays CPU are starving, that is, the memory access is more time consuming than actual computational time. How the extra uncomputing operations affect the performance can not be easily estimated by counting the number of instructions.

In the simplest g-segment trade off scheme [], it takes $Time(T) = T^{\log_g(4g-2)}$ and $Space(T) = (g-1)S\log_g T$. In practise, there are more practical trading off schemes that works much better in practise. [] Checkpointing [14]. In our work, especially in the `iexp` example, we show the the irreversibility of $*$ and / can be mitigated.

C. Reversibility and renormalization level

For an algorihms, the renormalization level of a program is the logarithm of the ratio between length of reversible instructions and irreversible instruction.

D. Quantum Computing

One of the fundamental difficulty of building a quantum computer is, unlike a classical state, an unknown quantum state can not be copied. A quantum state in a environment will decoherence and can not be recovered, this underlines the simulation nature of a quantum device. In the era of noisy intermediate sized quantum devices, more and more people are switching to classical-quantum hybrid devices, where a quantum device plays the role of a programmable simulator. Reversible computing does not enjoy the supremacy from quantum entanglement, nor the quantum limitations of non-cloning. Only the limitation of reversibility is retained, reversibility comes from the fact that micro scopic processes are all unitary. Irreversibility can only come from the interaction with classical devices, like environment induces decaying, qubit state resetting, measurements and classical feedbacks to quantum devices. These are rare resources in microscopic world as well as one of the most difficult part to implement in a practical device.

Quantum gates can possiblely simplify reversible instructions design, e.g. the quantum fourier transformation based adder. A classical in, classical out algorithm can find a quantum shortcut. With respect to this fact, it is reasonable to believe there would be a classical-quantum intermediate statge of computing, the reversible computing that bridge the gap between classical bits and fully entangled qubits. Although weaker than both, but with a killer application of instruction level automatic differentiation. By introducing quantum entanglement little by little, we will have faster instructions like addition and multiplication by utilizing quantum fourier transformation, faster algorithms and finally have an application that has genuine quantum advantage.

Reversible Turing machine and rotation gates $Ry(\theta)$ and $Rz(\theta)$ is equivalent to a quantum universal computer. In NiLang, we implement a path-integral based universal quantum simulator as shown in Appendix **??**. It is different from the classical-quantum hybrid mode compiler that using classical control flows, its control flow is reversible, and will be runable on a quantum computer in the future. The compiling theory developed for reversible programming will have profounding effect to quantum computers.

X. ACKNOWLEDGMENTS

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 and the research funding from Huawei Technologies under the Grant No. YBN2018095185.

[1] 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).

[2] 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.

[3] M. Innes, A. Edelman, K. Fischer, C. Rackauckas, E. Saba, V. B. Shah, and W. Tebbutt, CoRR **abs/1907.07587** (2019), arXiv:1907.07587.

[4] T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, CoRR **abs/1806.07366** (2018), arXiv:1806.07366.

[5] 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.

[6] 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.

[7] J. Behrmann, D. Duvenaud, and J. Jacobsen, CoRR **abs/1811.00995** (2018), arXiv:1811.00995.

[8] C. Lutz, "Janus: a time-reversible language," (1986), *Letter to R. Landauer*.

[9] C. J. Vieri, *Reversible Computer Engineering and Architecture*, Ph.D. thesis, Cambridge, MA, USA (1999), aAI0800892.

[10] M. Arjovsky, A. Shah, and Y. Bengio, CoRR **abs/1511.06464** (2015), arXiv:1511.06464.

[11] 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].

[12] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. A. Skirlo, M. Tegmark, and M. Soljacic, CoRR **abs/1612.05231** (2016), arXiv:1612.05231.

[13] C.-K. LI, R. ROBERTS, and X. YIN, International Journal of Quantum Information **11**, 1350015 (2013).

[14] T. Chen, B. Xu, C. Zhang, and C. Guestrin, CoRR **abs/1604.06174** (2016), arXiv:1604.06174.

Appendix A: NiLang Grammar

Terminologies

- *ident*, symbols
- *num*, numbers
- ϵ, empty statement
- *JuliaExpr*, native julia expression
- [ ], zero or one repetitions.

⟨Stmts⟩ ::= ϵ
| ⟨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*

⟨CallStmt⟩ ::= *JuliaExpr (* [⟨DataViews⟩] *)*

⟨Constant⟩ ::= *num* | π

⟨InstrBinOp⟩ ::= += | -= | ⩒=

⟨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⟩ ::= ϵ
| ⟨DataView⟩
| ⟨DataViews⟩ , ⟨DataView⟩

⟨DataView⟩ ::= ⟨DataView⟩ *[JuliaExpr]*
| ⟨DataView⟩ . *ident*
| *JuliaExpr (* ⟨DataView⟩ *)*
| ⟨Constant⟩
| *ident*

Dataview is a special bijective mapping of an object or a field (or item) of an object. The dataview can feedback to parent data with the `chfield` method, so that the modified object can generate desired dataview.

Appendix B: Instruction Table

Even though $\oplus(identity)$, $\oplus(*)$, $\oplus(/)$ and their reverse together with control flows are sufficient to write an arbituary differentiable program. For convinience we provide more,

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

Table II. A collection of reversible instructions.

Appendix C: Computing Fibonacci Sequence

```
function fib(n)
    if n > 2
        fib(n-1) + fib(n-2)
    else
        one(n)
    end
end
```

```
@i function rfib(out, n::T) where T
    @anc n1 = zero(T)
    @anc n2 = zero(T)
    @routine init begin
        n1 += identity(n)
        n1 -= identity(1.0)
        n2 += identity(n)
        n2 -= identity(2.0)
    end
    if (value(n) <= 2, ~)
        out += identity(1.0)
    else
        rfib(out, n1)
        rfib(out, n2)
    end
    ~@routine init
end
```

Appendix D: Julia based DSL implementation details

Macro ‘invfunc‘ defines a invertible function, ancillas are binded to a function, since ancillas are umcomputed to 0 at the end of call, so that it can be used repeatedly in a function, it is like a class variable in a class, with no side effects. When the *JuliaExpr* is a function all, it must be pure.

Some variables can be uncomputed to 0, but we choose not to for performance reason. For example `infer!(argmax, i, x)` which computes the location of maximum value in x and store the output to i, if we uncompute it, it doubles additional computational time. Here, we trade off the memory with computation time. As a result, we must feed `imax` to the function, so that this variable can be manipulated in outer scopes.