Why we should develop reversible programming - the machine learning perspective

Jin-Guo Liu,[1] Tai-Ne Zhao,[1] and Lei Wang[1,2,3,*]

[1]*Institute of Physics, Chinese Academy of Sciences, Beijing 100190, China*
[2]*CAS Center for Excellence in Topological Quantum Computation,*
University of Chinese Academy of Sciences, Beijing 100190, China
[3]*Songshan Lake Materials Laboratory, Dongguan, Guangdong 523808, China*

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 LANGUAGE DESIGN

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.

[*] wanglei@iphy.ac.cn

A. Memory, Instructions

A reversible instruction/function call in NiLang is a mapping between a same set of symbols.

```
julia> using NiLang

julia> @anc x = 0.5
0.5

julia> @anc y = 0.6
0.6

julia> @anc out! = 0.6
0.6

julia> @instr out! += x * y

julia> out!, x, y
(0.8999999999999999, 0.5, 0.6)
```

Here, the macro `@anc` binds symbols to a initialized memory space. 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 instruction is $\oplus(*)$, 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`

```
julia> @deanc x = 0.5

julia> @deanc out! = 0.6
ERROR: InvertibilityError("x (=0.8999999999999999) val
     (=0.6)")
/sample: 986tacktrace:
 [1] deanc(::Float64, ::Float64) at
      /home/leo/.julia/dev/NiLangCore/src/Core.jl:51
 [2] top-level scope at REPL[13]:1
```

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

B. 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 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.

We forbit `return` statement 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.

C. Control flow

instruction	**dual**
`a += b`	`a -= b`
`a = b`	`a = b`

`if (precond, postcond)`
`...`
`else`
`...`
`end`
	`if (postcond, precond)`
	`...`
	`else`
	`...`
	`end`

`while (precond, postcond)`
`...`
`end`
	`while (postcond, precond)`
	`...`
	`end`

`for i=start:step:stop`
`...`
`end`
	`for i=stop:-step:start`
	`...`
	`end`

Table I. Packages in the benchmark.

D. Types

E. π calculus et. al.

III. 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.

IV. SPEUDO INVERSE

In many cases, for example the $\exp(x)$ operation, due to the lack of reversible multiplication instruction in a classical computer instruction set, we have to either use a list to bookmark intermediate results, or using uncomputing technic that cost polynomial or even exponential more times to limit the memory usage to a constant.

For these arithmetics, here we introduce a trick that computes these functions in constant memory with a constant uncomputing overhead.

Let's see the following Taylor serie implementation of $\exp(x)$ in a reversible fashion

```
using NiLang, NiLang.AD


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

    out! += 1.0
    anc1 += 1.0
    while (val(anc1) > atol, !isapprox(iplus, 0.0))
        iplus += 1.0
        anc2 += anc1 * x
        anc3 += anc2 / iplus
        out! += anc3
        # speudo inverse
        anc1 -= anc2 / x
        anc2 -= anc3 * iplus
        SWAP(anc1, anc3)
    end

    ~(while (val(anc1) > atol, !isapprox(iplus, 0.0))
        iplus += 1.0
        anc2 += anc1 * x
        anc3 += anc2 / iplus
        # speudo inverse
        anc1 -= anc2 / x
        anc2 -= anc3 * iplus
        SWAP(anc1, anc3)
    end)
    anc1 -= 1.0
end
```

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.

A. Computing Jacobians, Hessians

B. First order gradient

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

$$J^{\vec{O}}_{x_i} = \sum_j J^{\vec{O}}_{y_j} J^{y_j}_{x_i} \tag{1}$$

and the adjoint mode

$$J^{y_j}_{\vec{I}} = \sum_j J^{y_j}_{x_i} J^{x_i}_{\vec{I}} \tag{2}$$

Here, $\vec{I}$ is the inputs and $\vec{O}$ is the outputs. This result can be obtained diagrammatically.

With reversible programming, f^{-1} is used for uncomputing intermediate results, $J^{\vec{y}}_{\vec{x}} \equiv (J^{\vec{x}}_{\vec{y}})^{-1}$ can also be obtained during uncomputing.

C. Second order gradient

The second order gradient can also be 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.

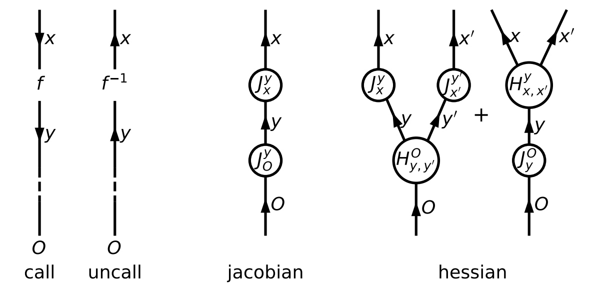

Figure 1. Adjoint rules for Jacobians and Hessians in tensor network language.

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.

D. 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. 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.

V. A NEW TRAINING SCHEME

We compute the output as

$$y = f(x), \tag{5}$$

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 folloing example

```
function f(x::Ref, y::Ref)
    add!(y, x)
    exp!(x)
    neg!(x)
    exp!(y)
    add!(x, y)
    return x, y
end

function invf(x::Ref, y::Ref)
    sub!(x, y)
    log!(y)
    neg!(x)
    log!(x)
    sub!(y, x)
    return x, y
end

x, y = Ref(0.3), Ref(1.6)
for i=1:100
    x = Ref(0.3)
    f(x, y)
    x[] = 1.0
    invf(x, y)
end
```

$f(x, y)$ computes $e^{(y+x)} - e^x$ and stores the output in x, 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 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"

$$@asserty == 0 \tag{6}$$
$$\texttt{fl}(y) + = exp(\texttt{fl}(x)) \tag{7}$$
$$\texttt{fl}(x) - = log(y) \tag{8}$$

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 $(x, x_a) \rightarrow (y, y_a)$

mutual information is

$$I(x, y) = S(x, y) - S(x) - S(y) \tag{9}$$
$$S(x, x_a) = S(y, y_a) \tag{10}$$
$$I(x, x_a) = 0 \tag{11}$$
$$I(y, y_a) = S(y, y_a) - S(y_a) \tag{12}$$

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.

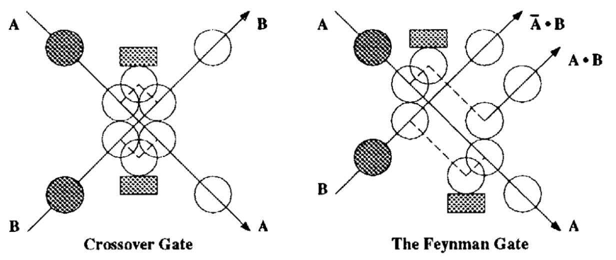

Figure 2. Two examples of BBM logic gates.

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.

B. Quantum computers

Fully entangled quantum systems are not easy to prepair. For a quantum system, finding a reliable dissipation source is also not easy, this is why the time for resetting a qubit to 0 (or erasing a bit information) can be the bottleneck of a computing.

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.

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 [10].

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] T. Chen, B. Xu, C. Zhang, and C. Guestrin, CoRR **abs/1604.06174** (2016), arXiv:1604.06174.
[11] M. Arjovsky, A. Shah, and Y. Bengio, CoRR **abs/1511.06464** (2015), arXiv:1511.06464.
[12] 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].
[13] L. Jing, Y. Shen, T. Dubcek, J. Peurifoy, S. A. Skirlo, M. Tegmark, and M. Soljacic, CoRR **abs/1612.05231** (2016), arXiv:1612.05231.
[14] C.-K. LI, R. ROBERTS, and X. YIN, International Journal of Quantum Information **11**, 1350015 (2013).

1. NiLang Grammar

Appendix A: NiLang

- Terminals in quotes or lower case.
  - ident, symbols
  - num, numbers
  - null, empty statement
  - expr, native julia expression.
  - expr::Int, native julia expression with integer return value.
  - expr::Bool, native julia expression with boolean return value.
- { }* indiceates zero or more repetitions.
- [ ] indicates zero or one repetitions.

```
Function ::= 'function' Fname '(' Fargs [',' ident
     '...']; Fargs ')' 'where' '{' Symbols '}'
                 Statements
              'end'

Fname ::= ident
          | '(' ident '::' ident ')'

Fargs ::= ident, Fargs
          | ident '::' ident, Fargs
          | ident '=' ident, Fargs
          | ident '::' ident '=' expr, Fargs
          | null

Symbols ::= ident, Symbols
          | ident


Statements ::= Ifstmt Statements
          | Whilestmt Statements
          | Forstmt Statements
          | Callstmt Statements
          | Instrstmt Statements
          | Revstmt Statements
          | @anc Statements
          | @routine Statements
          | @safe Statements
          | null

Ifstmt ::= 'if' '(' expr::Bool ',' expr::Bool ')'
                 Statements
              ['else'
                 Statements]
              'end'

Whilestmt ::= 'while' '(' expr::Bool ',' expr::Bool
     ')'
                 Statements
              'end'

Forstmt ::= 'for' ident '=' expr::Int[':' expr::Int]
     ':' expr::Int
                 Statements
              'end'

Callstmt ::= ident '(' Dataviews ')'

Instrstmt ::= ident '+=' ident ['(' Dataviews ')']
          | ident '-=' ident ['(' Dataviews ')']
          | ident '⊻=' ident ['(' Dataviews ')']
          | ident '.+=' ident ['.' '(' Dataviews ')']
          | ident '.-=' ident ['.' '(' Dataviews ')']
          | ident '.⊻=' ident ['.' '(' Dataviews
               ')']

Revstmt ::= '~' '(' Statements ')'

@routine ::= '@routine' ident 'begin'
                 Statements
              'end'
          | '@routine' ident

@anc ::= '@anc' ident '=' expr
          | '@deanc' ident '=' expr

@safe ::= '@safe' expr

Dataviews ::= Dataview, Dataviews
          | null

Dataview ::= Dataview '[' expr::Int ']'
          | ident {'.' ident}*
          | ident '(' Dataview ')'
          | Constant

Constant ::= num | 'π'
```

Dataview is a special surjective mapping of parent data, e.g. a field 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: 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.

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.

Appendix C: Unitary Matrices

Recurrent networks with a unitary parametrized network ease the gradient exploding and vanishing problem [11–13]. Among different parametrization schemes, the most elegant one is Ref. 13, 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 [14]. 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.

```
@assert j==0, i==0, k==0, ip==0
loop N-1
   j + 1
   loop j
      i + 1
      k + 1
      ip + i + 1
      rot(x[i], x[ip], z[k])
      ip - 1 - i
   end
   i - j
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]`$*\pi$.

$$R(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \tag{C1}$$

a. The gradient

Its backward rule is

$$\begin{aligned} \overline{\theta} &= \sum \frac{\partial R(\theta)}{\partial \theta} \odot (\overline{y} x^T) \\ &= \mathrm{Tr}\left[\frac{\partial R(\theta)}{\partial \theta}^T \overline{y} x^T\right] \\ &= \mathrm{Tr}\left[R\left(\frac{\pi}{2} - \theta\right) \overline{y} x^T\right] \end{aligned} \tag{C2}$$

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.

b. Rounding errors

$$a = a\cos(k\pi) - b\sin(k\pi) \tag{C3}$$

$$b = a\sin(k\pi) + b\cos(k\pi) \tag{C4}$$

$$\begin{aligned} \epsilon_a &= -a\sin(\theta)\theta\epsilon - b\cos(\theta)\theta\epsilon + a\epsilon\cos(\theta) - b\epsilon\sin(\theta) \\ &\sim \max(a, b)\epsilon \end{aligned} \tag{C5}$$

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.

c. Two level unitary for rectangular matrices

1. Invertible random numbers