Reversible Checkpointing

UCSD CSE 291 Differentiable Programming

Tzu-Mao Li

Recall: checkpointing trades between memory and compute

```
for i in range(n):
  stack.push(x)
  x = sin(x)

for i in range(n):
  x = stack.pop()
  dx = dx * cos(x)
```

with some heuristics, for this case we can achieve

```
for i in range(n):
  x = x0
  for j in range(i):
    x = sin(x)
  dx = dx * cos(x)
```

time complexity: O(n)
space complexity: O(n)

time complexity: O(nlog(n))
space complexity: O(log(n))

time complexity: O(n^2)
space complexity: O(1)

Recall: checkpointing trades between memory and compute

can we have the best of both worlds?

```
for i in range(n):
  stack.push(x)
  x = sin(x)

for i in range(n):
  x = stack.pop()
  dx = dx * cos(x)
```

time complexity: O(n)
space complexity: O(1)


```
for i in range(n):
  x = x0
  for j in range(i):
    x = sin(x)
  dx = dx * cos(x)
```

with some heuristics, for this case we can achieve

time complexity: O(n)
space complexity: O(n)

time complexity: O(nlog(n))
space complexity: O(log(n))

time complexity: O(n^2)
space complexity: O(1)

Let's differentiate the following code

```
i = 0
while True:
   x = i * x * x
   i = i + 1
```

Let's differentiate the following code

```
i = 0
while True:
  x = i * x * x
  i = i + 1
```

```
i = 0
loop_cnt = 0
while True:
  stack.push(x)
  x = i * x * x
  stack.push(i)
  i = i + 1
  loop_cnt += 1

while loop_cnt > 0:
  i = stack.pop()
  x = stack.pop()
  dx = 2 * i * dx * x
  loop_cnt -= 1
```

Let's differentiate the following code

can we do better?

```
i = 0
while True:
  x = i * x * x
  i = i + 1
```

```
i = 0
loop_cnt = 0
while True:
  stack.push(x)
  x = i * x * x
  stack.push(i)
  i = i + 1
  loop_cnt += 1

while loop_cnt > 0:
  i = stack.pop()
  x = stack.pop()
  dx = 2 * i * dx * x
  loop_cnt -= 1
```

Let's differentiate the following code

```
i = 0
while True:
   x = i * x * x
   i = i + 1
```

observation 1:
we can reconstruct i
without storing it!

```
i = 0
loop_cnt = 0
while True:
   stack.push(x)
   x = i * x * x
   # stack.push(i)
   i = i + 1
   loop_cnt += 1

while loop_cnt > 0:
   # i = stack.pop()
   i = i - 1
   x = stack.pop()
   dx = 2 * i * dx * x
   loop_cnt -= 1
```

Let's differentiate the following code

```
i = 0
while True:
   x = i * x * x
   i = i + 1
```

```
i = 0
loop_cnt = 0
while True:
   stack.push(x)
   x = i * x * x
   # stack.push(i)
   i = i + 1
   loop_cnt += 1

while loop_cnt > 0:
   # i = stack.pop()
   i = i - 1
   x = stack.pop()
   dx = 2 * i * dx * x
   loop_cnt -= 1
```

can we reconstruct x too?

observation 1:
we can reconstruct i
without storing it!

Let's differentiate the following code

```
i = 0
while True:
  x = i * x * x
  i = i + 1
```

observation 2:
we can reconstruct x too!
(for simplification
we assume x is positive here,
but same trick works for
negative x)

```
i = 0
loop_cnt = 0
while True:
  # stack.push(x)
  x = i * x * x
  # stack.push(i)
  i = i + 1
  loop_cnt += 1

while loop_cnt > 0:
  # i = stack.pop()
  i = i - 1
  # x = stack.pop()
  x = sqrt(x / i)
  dx = 2 * i * dx * x
  loop_cnt -= 1
```

Let's differentiate the following code

time complexity: O(n)
space complexity: O(1)

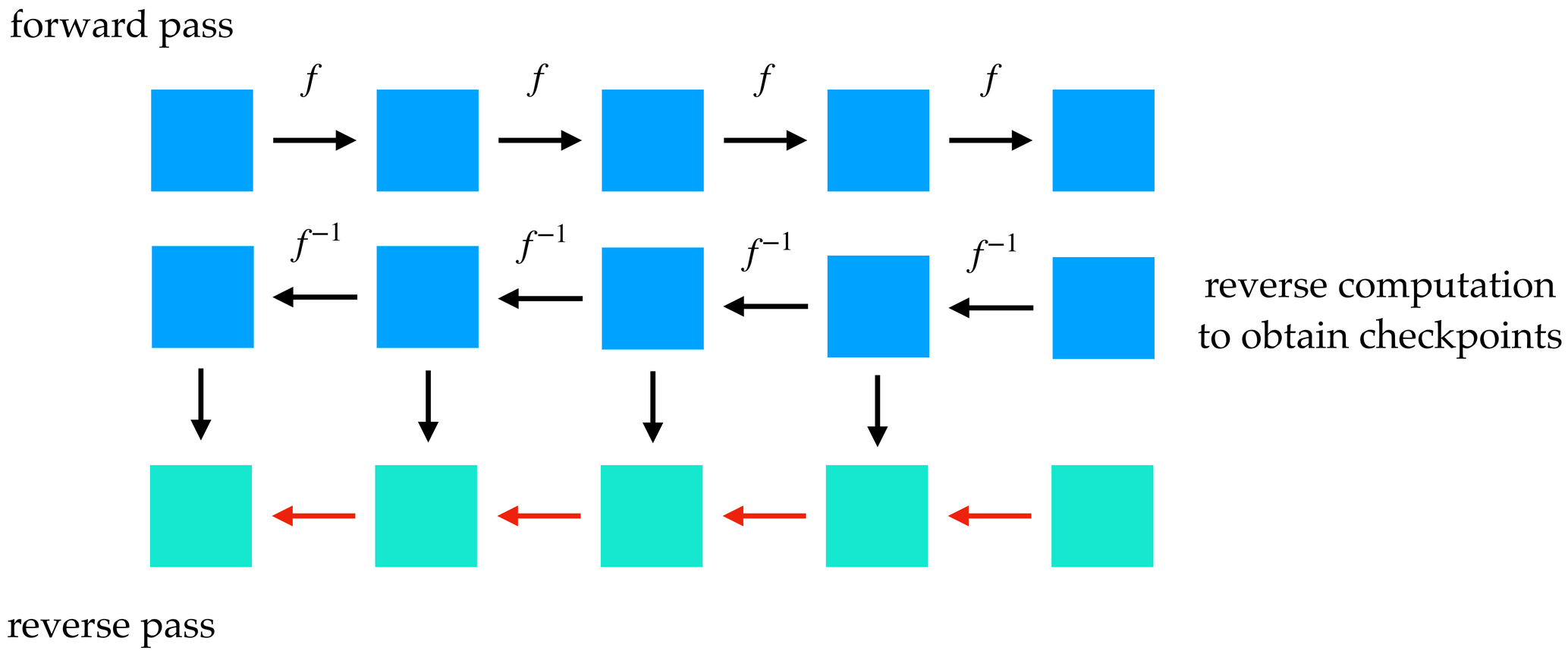
```
i = 0
while True:
    x = i * x * x
    i = i + 1
```

observation 2:
we can reconstruct x too!
(for simplification
we assume x is positive here,
but same trick works for
negative x)

```
i = 0
loop_cnt = 0
while True:
    # stack.push(x)
    x = i * x * x
    # stack.push(i)
    i = i + 1
    loop_cnt += 1

while loop_cnt > 0:
    # i = stack.pop()
    i = i - 1
    # x = stack.pop()
    x = sqrt(x / i)
    dx = 2 * i * dx * x
    loop_cnt -= 1
```

Reversible checkpointing: reconstruct overwritten values by reversing the computation



We can make non-reversible functions reversible by adding extra information

```
z = x + y
```

not reversible

We can make non-reversible functions reversible by adding extra information

```
z = x + y
```

not reversible

```
z = x + y
z' = x - y
```

reversible

We can make non-reversible functions reversible by adding extra information

```
z = x + y
```

not reversible

```
z = x + y
z' = x - y
```

reversible

```
x = (z + z')/2
y = (z - z')/2
```

Ensuring reversibility at the language level: reversible programming languages

JANUS: A TIME-REVERSIBLE LANGUAGE

By Christopher Lutz and Howard Derby, circa 1982

JANUS is a compiler and interpreter for the time-reversible language JANUS. The JANUS compiler is written in SLIMEULA, and compiles the code into an internal SLIMEULA Class structure which can be interpreted directly. 'SLIMEULA' means SIMULA running on a DECSYSTEM-20.

JANUS is considered to be a throw-away piece of code. It will not be maintained and is not purported to be robust.

The compiler consists of four major parts: A lexical analyzer which tokenizes the input stream and generates a symbol table; a recursive descent parser made of the Init Code of SLIMEULA Classes; an interpreter which consists of the procedure 'exec' common to all of the Classes created in the parsing; and the runtime command scanner.

(not invented for AD,
it was a course project at Caltech —
they just thought it would be cool)

forward Janus program

```
procedure fib
  from i = n
  do
    x1 += x2
    x1 <=> x2 # swap(x1, x2)
    i -= 1
  until i = 2
```

reverse Janus program

```
procedure fib
  from i = 2
  do
    i += 1
    x1 <=> x2
    x1 -= x2
  loop
  until i = n
```

https://en.wikipedia.org/wiki/Janus_(time-reversible_computing_programming_language)

Reversible languages can be made Turing complete

i.e., we don't lose any computing power even if we restrict our programming constructs to be reversible

Semantics for a Turing-complete Reversible Programming Language with Inductive Types

Kostia Chardonnet
Department of Computer Science and Engineering, University of Bologna, Italy

Louis Lemonnier[1]
Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France

Benoît Valiron
Université Paris-Saclay, CNRS, CentraleSupélec, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France

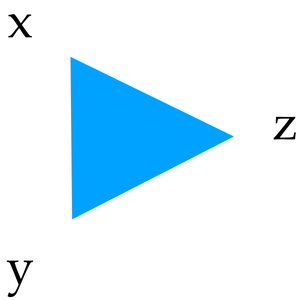
Abstract

This paper is concerned with the expressivity and denotational semantics of a functional higher-order reversible programming language based on Theseus. In this language, pattern-matching is used to ensure the reversibility of functions. We show how one can encode any Reversible Turing Machine in said language. We then build a sound and adequate categorical semantics based on join inverse categories, with additional structures to capture pattern-matching. We then derive a full completeness result, stating that any computable, partial injective function is the image of a term in the language.

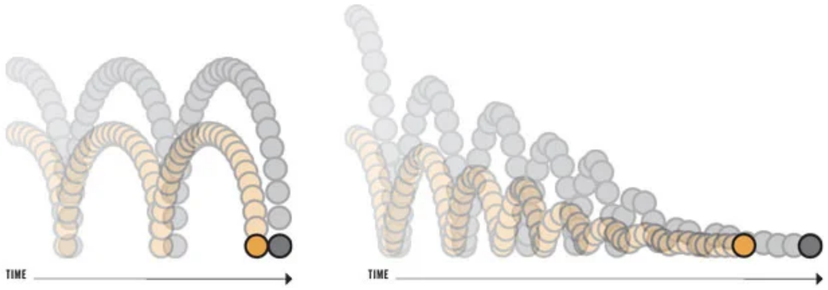
Related idea: reversible computing

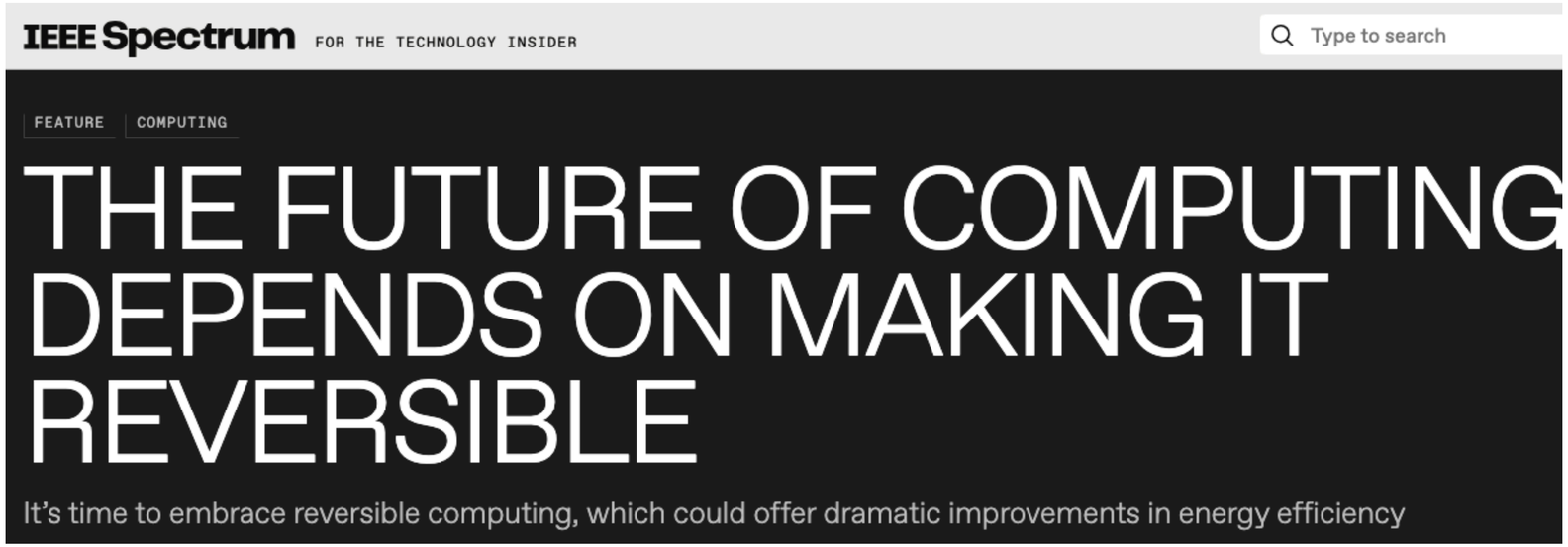
if we make our circuits reversible,
it should save energy!!

an adder



the reduction of bits
actually turns into heat/increase entropy,
wasting energy





Related idea: quantum computing

in quantum computing, every single operation also needs to be reversible because of … physics

a qubit is a superposition of two states

$$\left|\Psi\right\rangle = a\left|0\right\rangle + b\left|1\right\rangle$$

where $a^2 + b^2 = 1$

to ensure $a^2 + b^2 = 1$, any linear operations A imposed on it needs to be unitary, i.e., $A^T A = I$, this means that A is invertible

a cool language that ensures reversibility for quantum computing

Tower: Data Structures in Quantum Superposition

CHARLES YUAN, MIT CSAIL, USA
MICHAEL CARBIN, MIT CSAIL, USA

Emerging quantum algorithms for problems such as element distinctness, subset sum, and closest pair demonstrate computational advantages by relying on *abstract data structures*. Practically realizing such an algorithm as a program for a quantum computer requires an efficient implementation of the data structure whose operations correspond to *unitary operators* that manipulate quantum *superpositions* of data.

To correctly operate in superposition, an implementation must satisfy three properties — *reversibility*, *history independence*, and *bounded-time execution*. Standard implementations, such as the representation of an abstract set as a hash table, fail these properties, calling for tools to develop specialized implementations.

In this work, we present Core Tower, the first language for quantum programming with random-access memory. Core Tower enables the developer to implement data structures as pointer-based, linked data. It features a reversible semantics enabling every valid program to be translated to a unitary quantum circuit.

We present Boson, the first memory allocator that supports reversible, history-independent, and constant-time dynamic memory allocation in quantum superposition. We also present Tower, a language for quantum programming with recursively defined data structures. Tower features a type system that bounds all recursion using classical parameters as is necessary for a program to execute on a quantum computer.

Using Tower, we implement Ground, the first quantum library of data structures, including lists, stacks, queues, strings, and sets. We provide the first executable implementation of sets that satisfies all three mandated properties of reversibility, history independence, and bounded-time execution.

Reversible language applied to AD

NiLang

NiLang.jl (逆lang), is a reversible domain-specific language (DSL) that allow a program to go back to the past.

- Requires Julia version >= 1.3,

NiLang features:

- any program written in NiLang is differentiable,
- a reversible language with abstraction and arrays,
- complex values
- reversible logarithmic number system

https://github.com/GiggleLiu/NiLang.jl

Differentiate Everything with a Reversible Embeded Domain-Specific Language

Jin-Guo Liu
Institute of Physics, Chinese Academy of Sciences,
Beijing 100190, China
cacate0129@iphy.ac.cn

Taine Zhao
Department of Computer Science, University of Tsukuba
thaut@logic.cs.tsukuba.ac.jp

A very general reversible model

physical dynamics that do not lose energy (aka Hamiltonian dynamics) is always reversible!

(this is called a “conservative force”)

$$v_{t+1} = v_t + f(x_t)$$

$$x_{t+1} = x_t + v_{t+1} \cdot \Delta_t$$

(this is called a “symplectic integrator”)

A very general reversible model

physical dynamics that do not lose energy (aka Hamiltonian dynamics) is always reversible!

(this is called a “conservative force”)

$$v_{t+1} = v_t + f(x_t)$$

$$x_{t+1} = x_t + v_{t+1} \cdot \Delta_t$$

(this is called a “symplectic integrator”)

$$x_t = x_{t+1} - v_{t+1} \cdot \Delta_t$$

$$v_t = v_{t+1} - f(x_t)$$

Gradient descent + momentum
= energy preserving physical dynamics

this means that we can differentiate gradient descent + momentum
with millions of iterations, almost without storing anything across iterations!!

$$v_{t+1} = (1-\alpha)v_t + \alpha \nabla L(x_t)$$

$$x_{t+1} = x_t + v_{t+1} \cdot \Delta_t$$

$$x_t = x_{t+1} - v_{t+1} \cdot \Delta_t$$

$$v_t = \frac{v_{t+1} - \alpha \nabla L(x_t)}{1-\alpha}$$

Gradient descent + momentum = energy preserving physical dynamics

Gradient-based Hyperparameter Optimization through Reversible Learning

Dougal Maclaurin[†] MACLAURIN@PHYSICS.HARVARD.EDU
David Duvenaud[†] DDUVENAUD@SEAS.HARVARD.EDU
Ryan P. Adams RPA@SEAS.HARVARD.EDU

Abstract

Tuning hyperparameters of learning algorithms is hard because gradients are usually unavailable. We compute exact gradients of cross-validation performance with respect to all hyperparameters by chaining derivatives backwards through the *entire training procedure*. These gradients allow us to optimize thousands of hyperparameters, including step-size and momentum schedules, weight initialization distributions, richly parameterized regularization schemes, and neural network architectures. We compute hyperparameter gradients by exactly reversing the dynamics of stochastic gradient descent with momentum.

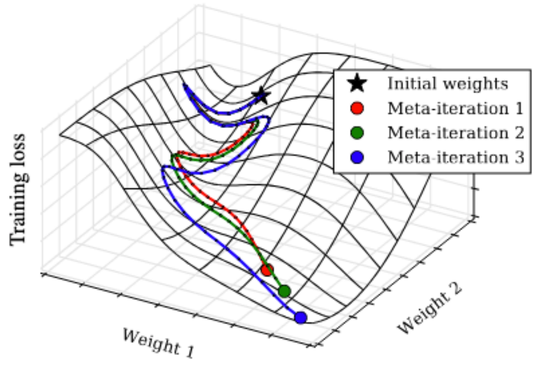


Figure 1. Hyperparameter optimization by gradient descent. Each meta-iteration runs an entire training run of stochastic gradient descent to optimize elementary parameters (weights 1 and 2). Gradients of the validation loss with respect to hyperparameters are then computed by propagating gradients back through the elementary training iterations. Hyperparameters (in this case, learning rate and momentum schedules) are then updated in the direction of this hypergradient.

1. Introduction

Machine learning systems abound with hyperparameters. These can be parameters that control model complexity, such as L_1 and L_2 penalties, or parameters that specify the

A slightly modified ResNet
= energy preserving physical dynamics

this means we can differentiate ResNet with lots of layers without storing intermediate!!

original ResNet

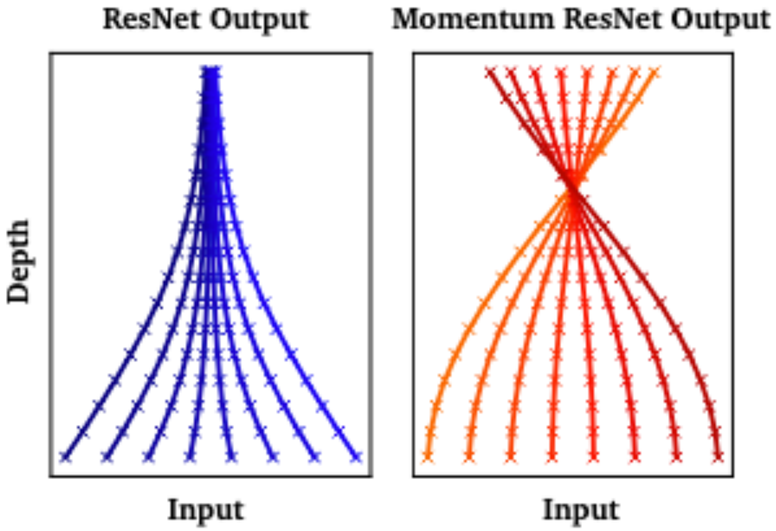
$$x_{t+1} = x_t + f(x_t, \theta_t)$$

usually a neural net

momentum ResNet

$$v_{t+1} = (1 - \alpha)v_t + \alpha f(x_t, \theta_t)$$

$$x_{t+1} = x_t + v_{t+1}$$



Momentum Residual Neural Networks

Michael E. Sander [1 2] Pierre Ablin [1 2] Mathieu Blondel [3] Gabriel Peyré [1 2]

RealNVP
= energy preserving physical dynamics

$$\begin{cases} y_{1:d} & = x_{1:d} \\ y_{d+1:D} & = x_{d+1:D} \odot \exp\big(s(x_{1:d})\big) + t(x_{1:d}) \end{cases}$$

$$\Leftrightarrow \begin{cases} x_{1:d} & = y_{1:d} \\ x_{d+1:D} & = \big(y_{d+1:D} - t(y_{1:d})\big) \odot \exp\big(-s(y_{1:d})\big), \end{cases}$$

DENSITY ESTIMATION USING REAL NVP

Laurent Dinh*
Montreal Institute for Learning Algorithms
University of Montreal
Montreal, QC H3T1J4

Jascha Sohl-Dickstein
Google Brain

Samy Bengio
Google Brain

ABSTRACT

Unsupervised learning of probabilistic models is a central yet challenging problem in machine learning. Specifically, designing models with tractable learning, sampling, inference and evaluation is crucial in solving this task. We extend the space of such models using real-valued non-volume preserving (real NVP) transformations, a set of powerful, stably invertible, and learnable transformations, resulting in an unsupervised learning algorithm with exact log-likelihood computation, exact and efficient sampling, exact and efficient inference of latent variables, and an interpretable latent space. We demonstrate its ability to model natural images on four datasets through sampling, log-likelihood evaluation, and latent variable manipulations.

A very general reversible model

physical dynamics that do not lose energy (aka Hamiltonian dynamics) is always reversible!

(this is called a “conservative force”)

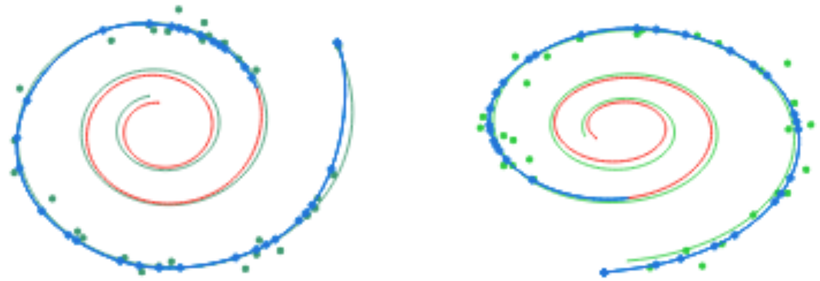
$$v_{t+1} = v_t + f(x_t)$$

$$x_{t+1} = x_t + v_{t+1} \cdot \Delta_t$$

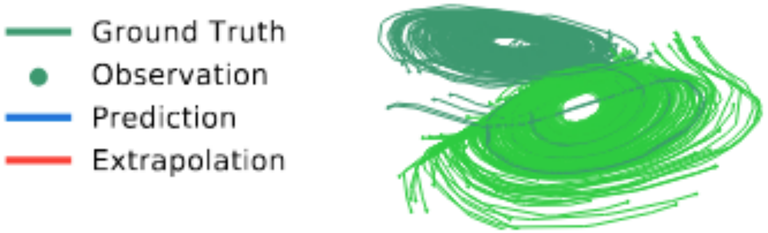
(this is called a “symplectic integrator”)

the key is this f can be arbitrarily complicated,
adding a lot to the modeling power

Application: modeling time-series data



(b) Latent Neural Ordinary Differential Equation

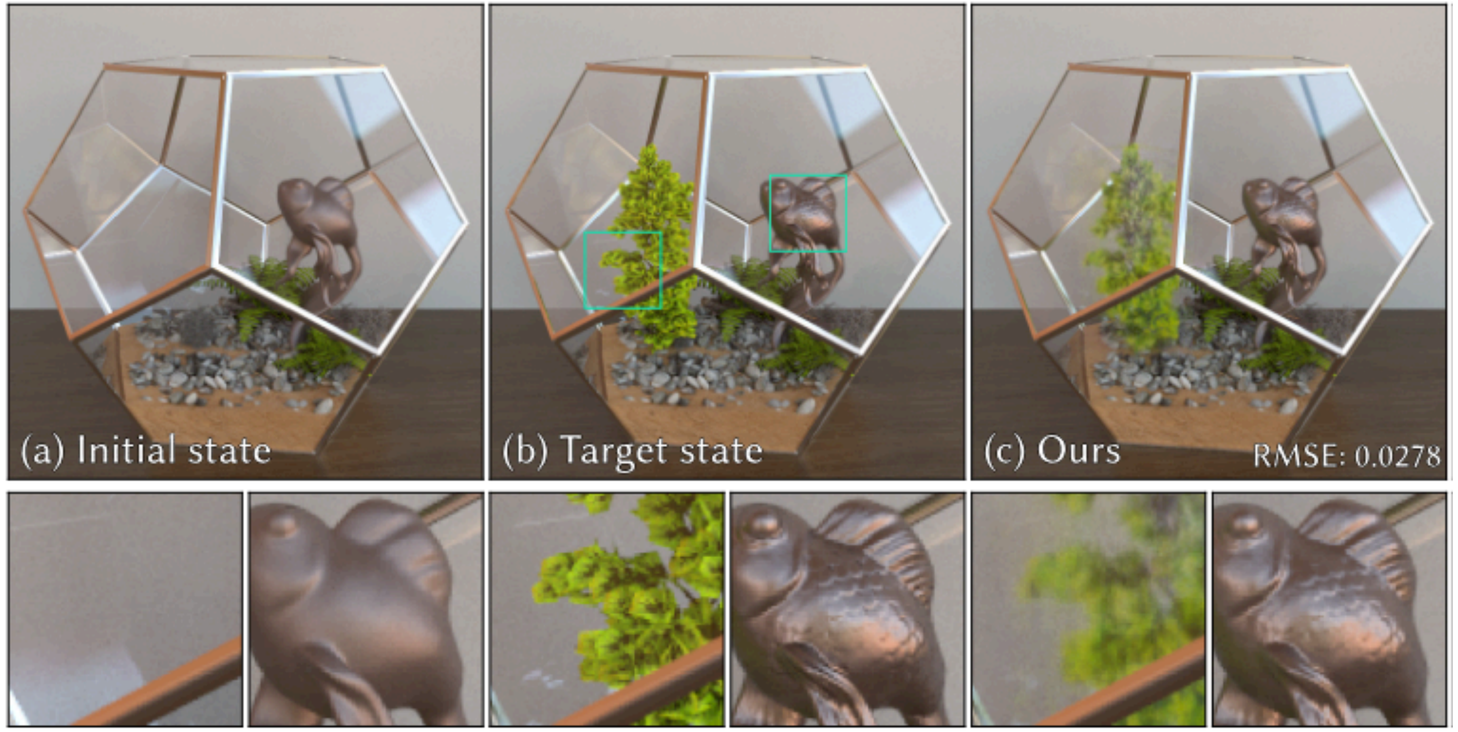


(c) Latent Trajectories

Neural Ordinary Differential Equations

Ricky T. Q. Chen*, Yulia Rubanova*, Jesse Bettencourt*, David Duvenaud
University of Toronto, Vector Institute
{rtqichen, rubanova, jessebett, duvenaud}@cs.toronto.edu

Application: inverse rendering

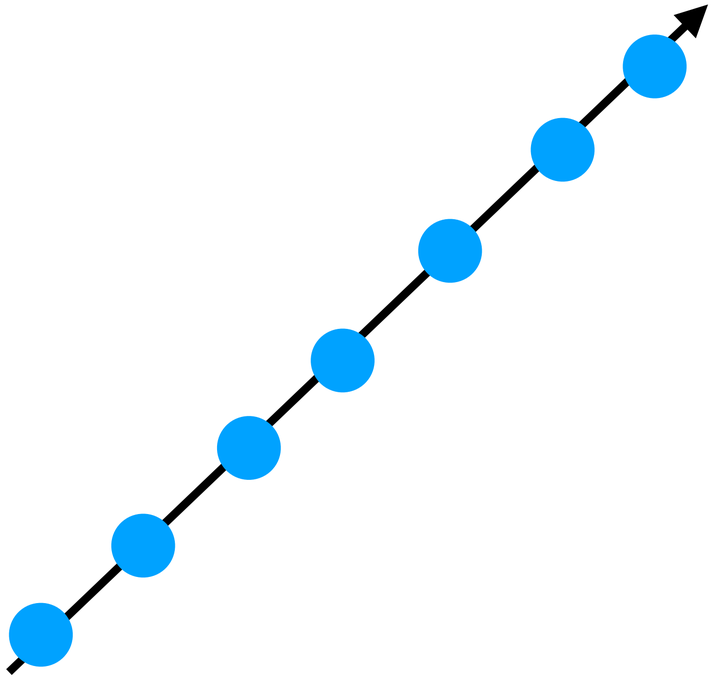


Path Replay Backpropagation: Differentiating Light Paths using Constant Memory and Linear Time

DELIO VICINI, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
SÉBASTIEN SPEIERER, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland
WENZEL JAKOB, École Polytechnique Fédérale de Lausanne (EPFL), Switzerland

Application: NeRF

alpha blending in NeRF is reversible! (I never see this mentioned in papers, not sure if people do it)



$$T_{t+1} = T_t \cdot \alpha_t$$

$$L_{t+1} = L_t + T_t \cdot (1 - \alpha_t) C_t$$

$$\downarrow$$

$$T_t = \frac{T_t}{\alpha_t}$$

$$L_t = L_{t+1} - T_t \cdot (1 - \alpha_t) C_t$$

Application: meta learning

Gradient-based Hyperparameter Optimization through Reversible Learning

Dougal Maclaurin[†] MACLAURIN@PHYSICS.HARVARD.EDU
David Duvenaud[†] DDUVENAUD@SEAS.HARVARD.EDU
Ryan P. Adams RPA@SEAS.HARVARD.EDU

Abstract

Tuning hyperparameters of learning algorithms is hard because gradients are usually unavailable. We compute exact gradients of cross-validation performance with respect to all hyperparameters by chaining derivatives backwards through the *entire training procedure*. These gradients allow us to optimize thousands of hyperparameters, including step-size and momentum schedules, weight initialization distributions, richly parameterized regularization schemes, and neural network architectures. We compute hyperparameter gradients by exactly reversing the dynamics of stochastic gradient descent with momentum.

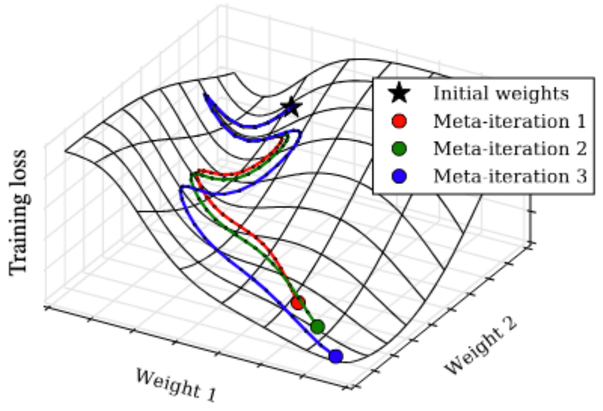


Figure 1. Hyperparameter optimization by gradient descent. Each meta-iteration runs an entire training run of stochastic gradient descent to optimize elementary parameters (weights 1 and 2). Gradients of the validation loss with respect to hyperparameters are then computed by propagating gradients back through the elementary training iterations. Hyperparameters (in this case, learning rate and momentum schedules) are then updated in the direction of this hypergradient.

1. Introduction

Machine learning systems abound with hyperparameters. These can be parameters that control model complexity, such as L_1 and L_2 penalties, or parameters that specify the

More applications?