Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Sondre Johannes Elgvin Rodahl

A Julia Implementation of the Differential Neural Computer

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
April 2020

NTNU
Norwegian University of
Science and Technology

Sondre Johannes Elgvin Rodahl

A Julia Implementation of the Differential Neural Computer

Master's thesis in Computer Science
Supervisor: Magnus Lie Hetland
April 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Acknowledgment

This master thesis, done in spring of 2020, is a part of the Master's degree at NTNU, Trondheim, in computer science with a spezialization in algorithms. First, I wish to express my gratefulness to Associate Professor Magnus Lie Hetland for accepting me as his master student and providing invaluable guidance both profesionally and personally through the process. This work would not be possible without his support. I also wish to thank Professor Anne Elster for guidance on high performance computing and for the use of hardware.
Finally, a huge thanks to my partner Ingvild Lervaag for never losing faith in me, even when I'm at my worst. Her emotional and practical support carried this thesis through to the end. My family has always supported me and believed, and I owe them a great thanks for arming me with the tools I need to achieve my goals.

Trondheim, Norway

Sondre Johannes Elgvin Rodahl

Abstract

In the paradigm of differentiable programming, programs are modeled as a parameterized solution prototype, which is trained by gradient-based optimization. DeepMind's Differential Neural Computer (DNC) is an example of the transition from traditional deep learning models to differential programs. In this thesis, a Julia implementation of the DNC is presented. An argument is made for the applicability of the Julia programming language and its machine learning ecosystem, with Flux at its core and Zygote as the algorithmic differentiation engine, on differentiable programs. The focus of the thesis is on the role of algorithmic differentiation in machine learning toolkits. The reimplementation is benchmarked, achieving 10% speedup on a simple training task with a simpler implementation, with some internal methods running up to seven times faster.

Sammendrag

I differensierbar programmering-paradigmet modelleres programmer som parametriserte løsninger på et problem som deretter trenes med gradientbaserte optimeringsmetoder. DeepMinds Differentiable Neural Computer (DNC) er et eksempel på overgangen fra tradisjonell dyp læring til differensierbar programmering. I denne oppgaven argumenteres det for at Julias maskinlæringsrammeverk, med Flux i sentrum og med Zygote som algoritmisk differensiator, er et mer passende verktøy for implementasjon av programmer som DNC enn TensorFlow. Implementasjonen settes i kontekst av algoritmisk differensiering sin rolle i differensierbar programmering. DNC reimplementers i Julia og dens interne metoder måles opp mot den originale TensorFlow-implementasjonen, der den oppnår lik treningstid med enkelte interne metoder som kjører opp mot syv ganger så raskt.

Contents

List of Figures **5**

List of Tables **6**

1 Introduction **1**

2 Algorithmic Differentiation **5**

2.1 Algorithmic differentiation 5

2.2 AD in Practice 9

3 DNC **19**

3.1 DNC Description 20

3.2 Julia Implementation of the DNC 24

4 Results **31**

4.1 Benchmarks 31

5 Discussion **35**

6 Conclusion **39**

Bibliography **41**

List of Figures

2.1 Forward mode AD 6

3.1 A overview of the DNC architecture. Dashed lines show recursive connections. 20
3.2 Dataflow graph for the memory writing operation. Circles correspond to functions in the Julia implementation. Red variables are state variables from the previous iteration, blue are input variables from the controller and green intermediate variables. See table 3.1 for a complete overview of the variables used. . . . 26
3.3 Dataflow graph for read vector calculation. Circles correspond to functions in the Julia implementation. Red variables are state variables from the previous iteration, blue are input variables from the controller and green intermediate variables. See table 3.1 for a complete overview of the variables used. 27
3.4 The controller is extended with a layer of independent linear transformations to obtain the memory access parameters. Variable names refer to table 3.1. 28

List of Tables

2.1 A calculation of $y = sin(a*b)+b$ and the corresponding $\frac{\partial y}{\partial a}$ using forward mode AD. 8
2.2 A calculation of $y = sin(a * b) + b$ and the partials $\frac{\partial y}{\partial a}$ and $\frac{\partial y}{\partial b}$ using reverse mode AD. 9
2.3 Adjoint SSA program compared with reverse mode AD in mathematical notation. Underlined variables reference the primal program in equation 2.9. 17

3.1 An overview of variables in the DNC. 22

4.1 Benchmarks of forward and gradient calculations of methods in the DNC. "G/F" column is the slowdown of gradient calculation compared with the forward pass only. Parameters: X=6, Y=5, N=16, W=64, R=4, batchsize=16. 32
4.2 Timing of gradient calculations of methods in both Julia and TensorFlow implementation of DNC. `BenchmarkTools.@btime` is used in the Julia implementation and the `timeit` module for the TensorFlow implementation. Parameters: X=6, Y=5, N=16, W=64, R=4, batchsize=16. 33

Chapter 1

Introduction

Deep learning has proven to be a useful tool yielding impressive results in areas including computer vision[1], language processing[2] and many more. While deep learning models achieve great results in their specific task, they rarely translate well to other problems, and new models need to be designed and trained with appropriate data. As machine learning models evolve, researchers are looking to generalize them so they can be applied to a wider range of problems. This leads to the evolution of a new type of software: programs that prototype a parameterized solution to a problem and apply gradient based optimization to solve it. In this sense, neural network is merely one of many possible prototypes. This new paradigm, called differentiable programming (δP), impact the way machine learning progresses today. Machine learning researchers are acknowledging the fact that the backpropagation algorithm is a special case of reverse mode algorithmic differentiation (AD), and frameworks for machine learning are transitioning from course-grained block based gradient calculation to general-purpose AD. Yann LeCun, Chief AI Scientist of Facebook, phrased it like this:[3] "Deep Learning est mort. Vive Differential Programming!"

The motivation for artificial neural nets is twofold. On one hand, they can be approached mathematically as highly parameterised functions that are able to approximate unknown relations. On the other hand, drawing inspiration from the world of psychology, they can be viewed as an attempt to model human intelligence, mimicking the functionality of the human brain. The human neural net consists of synapses that can trigger and propagate signals to other neurons. Similarly, artificial neurons will receive an input, perform calculations and propagate an output to the next layer of neurons. A natural

extension of this is to simulate other parts of the brain as well, such as the hippocampus. In psychology, concepts and limitations of our working memory is studied. The working memory involves an attention mechanism to focus on relevant information. Hippocampus, the memory bank, stores information while a central entity retrieve and manipulate the data.

The differential neural computer (DNC)[4], developed by DeepMind, draws inspiration from both the world of psychology and information science. Being an extension of the Neural Turing Machine (NTM)[5], the DNC mimics a computer in its architecture: A neural network, usually recurrent, acts as controller, similar to a CPU, and is able to access an external memory, the model's hippocampus. The entities accessing the memory are called read and write heads after the notation of the Turing machine.[6] In a traditional machine, memory locations are accessed explicitly. However, in the DNC, the memory access system needs to be wholly differentiable to enable learning. This is solved by using blurry access operations that interact with all memory slots to a varying degree. Being able to learn general problem solving is one of the main attractions of the DNC. As a combination of a deep learning network and a traditionally implemented memory access module, the DNC fits nicely as an illustrator of the transition from deep learning to δP.

The DNC is implemented in TensorFlow version 1.15[7], a machine learning toolkit developed by Google. TensorFlow provides high performance models using a static computational graph allowing thorough compiling optimizations and job planning for heterogeneous systems.

The goal of this thesis is to study next-generation tools for differential programming through a reimplementation of the DNC. The framework of choice is the Julia programming language[8, 9]. For many years the norm for scientific frameworks has been to provide a dynamic front end that is simple to write and understand, while the heavy lifting is done with C or Fortran in the back end to provide high performance. Languages such as MATLAB, Octave[10], and R[11] are all examples of this approach. This is called the two-language problem, and Julia aims to solve it [8]. Introduced in 2012, Julia achieves C-like performance in a dynamic language through modern optimization techniques and a carefully designed language. The language is aiming primarily at scientific and numerical computing and use an extensive type system with a multiple dispatch system to specialize against run-time types. Through Julia's machine learning framework Flux and its diverse ecosystem, the language is establishing itself as a hotbed for next generation δP applications, such as a differentiable ray tracer[12] and neural ODEs[13]. The core actors in play in the Julia implementation of the DNC are Flux,

providing high level abstractions of machine learning models, and Zygote, Flux' algorithmic differentiator. As gradient propagation is at the heart of any machine learning model, a high performance AD engine is key for a high performance ML framework. Zygote's approach to AD is different than most other dynamic AD engines, and is discussed in detail in section 2.2.
Through the implementation of the DNC, the aim is to discuss benefits and drawbacks of a dynamic approach compared with the static approach used in the original TensorFlow implementation. Performance metrics are easily compared, but their are not the only factor of a program's quality. Simplicity is a feature which unveils itself in multiple aspects of program development. High level languages are designed to empower the programmer through simplicity. Abstracting away the gritty details of the low level language, the programmer is allowed time to focus on creative and efficient design. However, the abstraction often comes at the price of performance. In the world of machine learning, a multitude of programs and libraries provides high level abstractions of complex data structures, networks, and mathematical routines. They allow the data scientist to focus on building the desired model correctly. Simplicity should also allow for fast debugging, inspection, testing and benchmarking of the program. A program that is simple to inspect is less likely to introduce bugs. Although not as easily quantified, simplicity is also a desirable feature of a program.
This thesis compares TensorFlow with Zygote through a concrete example: the DNC. Chapter 2 introduces the concept of algorithmic differentiation, especially considering its applicability in machine learning frameworks. AD in TensorFlow and Zygote is presented and compared. Chapter 3 presents the differentiable neural computer in detail and discuss the implementation process for the Julia version. Finally, benchmarking results are presented and discussed.

Chapter 2

Algorithmic Differentiation

Algorithmic differentiation is the process of automatically deriving gradients for a computer program. Commonly referred to as automatic differentiation, it differs from numerical and symbolic differentiation in several ways. AD is derived from the observation that any numerical algorithm is eventually a combination of elementary numerical operations, and given elementary differentiation rules, derivatives can propagate numerically through the program using the chain rule. AD gives true, not approximated, gradient values due to the implementation of basic derivative rules, and it avoids expression swell by (exact) numerical intermediate computation. A mathematical background for AD and its two modes is presented: forward, computing derivatives in parallel with the primal program, and reverse, propagating derivatives back through intermediate variables after completion of the primal program. The role of AD in modern machine learning framework is discussed with focus on TensorFlow and Zygote. Eventually, Zygote's approach to AD is discussed in detail.

Algorithmic differentiation

Although AD propagates numerical values, it is distinctively different from numerical differentiation. Numerical differentiation use algorithms to numerically estimate the derivative of a function based on some function values or other information about it. The algorithms are usually based on the difference quotient $\frac{f(x+h)-f(x)}{h}$ which when evaluated at the limit as $h \to 0$

gives the derivative of the function f at the point x. However, as setting $h = 0$ directly gives division by zero, an estimation error is expected, and for small values of h, floating point rounding errors can become an issue.
The difference between algorithmic and symbolic differentiation is less distinct. Symbolic differentiation, also called computer algebra, manipulates the function expression to generate an exact expression for the derivative function. Fundamental rules of differentiation such as the product and chain rule are applied and the result is simplified to an acceptable level. Symbolic differentiation is exact but is slow when applied to a function of many input variables, as is the case with all deep learning applications. Also, symbolic differentiation would require the programmer to express the program in closed form, as control flow and loops are not directly supported. The aim of algorithmic differentiation is to find the derivative of an arbitrary block of computer code. Symbolic differentiation would have to face the complications of expressing the entire program in a single mathematical expression.
At the core of AD is the chain rule of differentiation. Contrary to symbolic differentiation, AD aims not to manipulate the expressions symbolically with the chain rule but rather evaluate the intermediary results numerically and propagate the gradient through the program. With elementary operations having defined differentiation rules, AD is able to break a block of code down to its basic operations, find their gradient, and then propagate the gradient values with the chain rule.

Forward mode AD

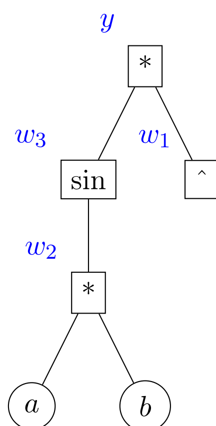


Figure 2.1: Forward mode AD

Allowing the differential values to propagate in parallel with program execution results in forward mode AD. Forward mode AD is simple to implement as it only requires an extra bookkeeping variable for the derivative of each variable which gets updated when an operation is performed on the variable. Interestingly, this is equivalent to replacing each variable in the program with a dual number $x+\dot{x}\epsilon$ (section 2.2). This is the implementation strategy followed by i.e. ForwardDiff.jl [14].

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_{n-1}} \frac{\partial w_{n-1}}{\partial x} = \frac{\partial y}{\partial w_{n-1}} (\frac{\partial w_{n-1}}{\partial w_{n-2}} \frac{\partial w_{n-2}}{\partial x}) = ... \tag{2.1}$$

Forward mode AD is using the chain rule to substitute each inner expression iteratively as in equation 2.1. For each sub-expression w_i, we are computing its derivative $\dot{w}_i = \frac{\partial w_i}{\partial x}$.
As a simple example, consider the function $y = sin(a * b) + b$. The two independent variables a and b determines y. The transformation is a combination of elementary operations:

$$\begin{aligned} w_1 &= a \\ w_2 &= b \\ w_3 &= w_1 * w_2 \\ w_4 &= \sin(w_3) \\ w_5 &= w_4 + w_2 \end{aligned} \tag{2.2}$$

A program evaluation trace such as this is often referred to as a tape or Wengert list[15, 16] and is the basis for both forward and reverse mode AD. By applying the chain rule, we can find the sensitivity of each independent and intermediate variable. Table 2.1 shows an iteration of a forward mode AD pass calculating $\frac{\partial y}{\partial a}$. The left column shows the program evaluation trace and the right column the accompanying derivative program. Calculations on the derivative values can be viewed as an independent program determined by the values $\dot{w}_1$ and $\dot{w}_2$. These seed values are determined by the choice of variable on which derivation is applied. In the illustrated example, the seed values are $\dot{w}_1 = \frac{\partial a}{\partial a} = 1$ and $\dot{w}_2 = \frac{\partial b}{\partial a} = 0$. As the differential calculations are done in parallel with the original program, the time complexity is only a constant factor to that of the original program. If $\frac{\partial y}{\partial b}$, the sensitivity on y with respect to b, is needed, the same procedure needs to be repeated but with different seeds, namely $\frac{\partial a}{\partial b} = 0$ and $\frac{\partial b}{\partial b} = 1$. This illustrates the fact that forward mode AD performance deteriorates when the number of input variables increases, motivating the need for reverse mode AD. While forward mode can calculate the sensitivity of an input variable for all the dependent

variables in one pass, reverse mode is able to calculate the sensitivities of all independent variables for a single dependent variable.

Table 2.1: A calculation of $y = sin(a*b)+b$ and the corresponding $\frac{\partial y}{\partial a}$ using forward mode AD.

Original program	Forward mode AD
$w_1 = \pi$	$\dot{w}_1 = 1$ (seed)
$w_2 = 2$	$\dot{w}_2 = 0$ (seed)
$w_3 = w_1 * w_2 = 2\pi$	$\dot{w}_3 = w_1 * \dot{w}_2 + w_2 * \dot{w}_1 = 2$
$w_4 = \sin(w_3) = 0$	$\dot{w}_4 = \cos(w_3) * \dot{w}_3 = 2$
$w_5 = w_4 + w_2 = 2$	$\dot{w}_5 = \dot{w}_4 + \dot{w}_2 = 2$

Reverse Mode AD

In case of $f : \mathbb{R}^m \to \mathbb{R}^n$, $m >> n$, we want to avoid passing through the program m times to calculate the derivative with respect to each independent variable. Reverse mode AD fixates a *dependent* variable, and computes the sensitivity of every sub-expression. In the extreme case of a function $f : \mathbb{R}^m \to \mathbb{R}$, only one pass using reverse mode is sufficient to calculate the complete gradient $\nabla f = (\frac{\partial y}{\partial x_1}, ..., \frac{\partial y}{\partial x_m})$. In deep learning applications, the number of independent variables can be very large.
Equivalently to forward mode, reverse mode AD is based on the chain rule. Instead of substituting the inner expression using the chain rule as in forward mode, now the *outer* expression is substituted

$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial w_1}\frac{\partial w_1}{\partial x} = (\frac{\partial y}{\partial w_2}\frac{\partial w_2}{\partial w_1})\frac{\partial w_1}{\partial x}... \tag{2.3}$$

For each sub-expression, we are interested in calculating the *adjoint*, denoted $\bar{w}_i$ defined in equation 2.4.

$$\bar{w}_i = \frac{\partial y_j}{\partial w_i} \tag{2.4}$$

$\bar{w}_i$ represents the sensitivity of an output variable y_j to changes in w_i. Contrary to forward mode, reverse mode AD can not be calculated in parallel with the original program as we start from the end and work our way backwards in the computational graph. Therefore, while requiring only one pass to calculate adjoints for each sub-expression w_i, it also requires the storage of intermediate variables w_i together with information on what operations were used to achieve them. Table 2.2 shows the evaluation of reverse mode AD through the same expression as in table 2.1. It should be noted that

both $\bar{a}$ and $\bar{b}$ are calculated in the same pass. As w_2 contributes to two expressions, their adjoint values need to added in the calculation of $\bar{w}_2$.

Table 2.2: A calculation of $y = sin(a * b) + b$ and the partials $\frac{\partial y}{\partial a}$ and $\frac{\partial y}{\partial b}$ using reverse mode AD.

Original program	Reverse mode AD
$w_1 = a = \pi$	$\bar{w}_5 = 1$ (seed)
$w_2 = b = 2$	$\bar{w}_4 = \bar{w}_5 \frac{\partial w_5}{\partial w_4} = 1$
$w_3 = w_1 * w_2 = 2\pi$	$\bar{w}_3 = \bar{w}_4 \frac{\partial w_4}{\partial w_3} = cos(w_3) = 1$
$w_4 = \sin(w_3) = 0$	$\bar{w}_2 = \bar{w}_5 \frac{\partial w_5}{\partial w_2} + \bar{w}_3 \frac{\partial w_3}{\partial w_2} = 1 + w_1 = 1 + \pi$
$w_5 = w_4 + w_2 = 2$	$\bar{w}_1 = \bar{w}_3 \frac{\partial w_3}{\partial w_1} = w_2 = 2$

AD in Practice

Forward Mode

Most forward mode AD implementations relies on the mathematical notion of dual numbers. A dual number is an element of the set $\mathbb{D}$ defined in equation 2.5.[17]

$$\mathbb{D} = \{\tilde{a} = a + \dot{a}\epsilon : a, \dot{a} \in \mathbb{R}, \epsilon^2 = 0, \epsilon \neq 0\} \tag{2.5}$$

For notational brevity, we define $a + \dot{a}\epsilon = (a, \dot{a})$, and call a the primal The notation is similar to that of complex numbers with the critical difference that $\epsilon^2 = 0$ and not -1. Addition and subtraction is defined identically as for complex numbers:

$$(a, \dot{a}) + (b, \dot{b}) = (a + b, \dot{a} + \dot{b}) \tag{2.6}$$

Inspecting dual multiplication, an interesting observation can be made.

$$(a + \dot{a}\epsilon)(b + \dot{b}\epsilon) = ab + a\dot{b}\epsilon + \dot{a}b\epsilon + \dot{a}\dot{b}\epsilon^2 = (ab) + (a\dot{b} + \dot{a}b)\epsilon \tag{2.7}$$

The coefficient in front of ϵ mirrors basic symbolic differentiation rules. We can exploit this by extending a function to operate on a dual number such that

$$f(a + \dot{a}) = f(a) + f'(a)\dot{a}\epsilon \tag{2.8}$$

In practice, this is usually implemented by overloading operators to accept dual numbers storing both a and $\dot{a}$. A bare-boned and extremely simple

example can be implemented in the Julia programming language using only a couple lines of code, as shown in listing 2.1. This simple script is enough to differentiate any expression consisting of addition, multiplication, and basic trigonometrical functions, and can easily be extended to include e.g. the exponential and logarithmic function.

Listing 2.1: Simple example of forward mode

```
struct Dual
    primal
    dual
end

import Base: +, *, sin, cos

+(a::Dual, b::Dual) = Dual(a.primal+b.primal, a.dual+b.dual)
*(a::Dual, b::Dual) = Dual(a.primal*b.primal, a.primal*b.dual+a.dual*b.primal)
sin(a::Dual) = Dual(sin(a.primal), a.dual*cos(a.primal))
cos(a::Dual) = Dual(cos(a.primal),-sin(a.primal)*a.dual)

function gradient(f, inputs...)
    res = zeros(length(inputs))
    # We need to iterate through the function once for each input
    for i in 1:length(inputs)
        # Seed the input derivatives
        # input `i` is seeded to 1, the rest 0
        duals = [j==i ? Dual(inputs[j], 1.0) : Dual(inputs[j], 0.0)
                 for j in 1:length(inputs)]
        # Calculate derivative with respect to input `i`
        res[i] = f(duals...).dual
    end
    res
end

julia> f(a, b) = sin(a*b) + b
f (generic function with 1 method)

julia> gradient(f, π, 2.0)
2-element Array{Float64,1}:
  2.0
  4.141592653589793
```

An advantage of operator overloading is that the user does not have to be aware of the internal dual representation. Although minimal, inefficient and error prone, this example illustrates the approach of e.g. the Julia package ForwardDiff.jl[14].

Reverse Mode and Machine Learning

Gradient based optimization is at the very core of modern machine learning. In its simplest form, gradient descent minimizes a function $f : \mathbb{R}^n \to \mathbb{R}$ by adjusting the function parameters $\boldsymbol{w}$ in the opposite direction of their derivative: $\boldsymbol{w}_t = \boldsymbol{w}_{t-1} - \gamma \nabla f$ with learning rate γ. The gradient is calculated using backpropagation. The backpropagation algorithm is in fact a special case of reverse AD; derivatives are propagated backwards from the calculated scalar loss value through all network layers. Gradient-based optimization techniques are used in a multitude of machine learning systems, so the need for a high performance gradient calculation is evident. Typically, machine learning systems use a tracing approach similar to the one introduced in section 2.1. Using operator overloading, methods now not only produce forward output, they also record the operations and the inputs, essentially producing a graph of basic operations equivalent to a Wengert list. Differentiating the Wengert list is straightforward - the same procedure as exemplified in section 2.1 can be utilized. For large models, a challenge arises as the number of intermediate values that needs to be stored become too large for the memory to handle. This can be solved by for example checkpointing, a scheme where only a subset of variables are stored, and sections of the forward run is recomputed as intermediate variables are needed. Other algorithms compromising between time and space complexity also exist[18]. Machine learning toolkits typically follow one of two paradigms: static or dynamic declaration, depending on whether they compile the tape or interpret it respectively. Programs following the static declaration paradigm are also called *define-and-run* programs, as they use a two step process.

1. Construct a computational graph

2. Execute the graph with different inputs

This approach is used by popular toolkits such as TensorFlow[7], Theano[19], and Caffe[20]. A number of benefits arise from static declaration: The computational graph can be subject to optimizations ensuring the following executions are as fast as possible. Since the second step often include thousands or millions of iterations, there is a lot to gain, and the developer can allow less efficient optimization routines as their one time cost will be amortized across a long training session. Additionally, the graph can be used to schedule computation across multiple, possibly heterogeneous, devices. The drawback however is the models' inflexibility. The static computational graph can only calculate the program it was designed for, and the graph may

only consist of specifically designed components the optimizers and work distributors can understand. The graph is usually expressed through a separate mini-language encapsulated in calls to library functions. Any necessary feature needs to be reimplemented in the library, as e.g. control flow has to be incorporated into the static graph. This results in TensorFlow having constructs such as `tf.cond` representing an if-statement and `tf.while_loop` for iterations. Semantically and syntactically, the sub-language will necessarily be more restricted than the host language. Many modern networks challenge the limits imposed by static graphs. In some cases, the behavior of the model depends on the input data provided to it. It could be required to handle variably sized inputs as is the case for natural language processors operating on sentences of different lengths. Tree-[21] and graph-structured[22, 23] networks allow variably structured inputs, requiring a significant complexity in the static architecture as each input may require a different model procedure.

Define-and-run toolkits overcome these issues by allowing unspecified input size at compile time and through the introduction of special control flow handling graph nodes such as `tf.cond`, being a building block for the implementation of non-trivial control flow. Indeed, one of TensorFlow's initial goals was to enable researchers to explore a wider variety of models[7]. Increased graph complexity decreases the opportunity space for optimizers however, thus degrading the main benefit of static graphs.

There are also some practical difficulties in using statically declared graphs for intricate model design. Complex program logic, which in itself may be difficult to implement in the host language, is even harder to code correctly through an indirect API and require a deep understanding of the library. While many errors are detectable at compile time, some will emerge at run time only, especially for programs allowing unspecified input size. Debugging is harder in statically declared programs because the distance between the bug and the error caused by it is greater both temporally and spatially. The value of simplicity should not be underrated when complex models are at subject, as simple debugging and inspection allow the programmer more time to experiment and understand complex issues, and less time inspecting bugs.

Programs in the dynamic declaration paradigm, including among others PyTorch[24], Chainer[25], and DyNet[26], and Zygote[27] merge the two steps described above. They are often labeled *define-by-run* frameworks as a computation graph is constructed on-the-fly following the execution of the loss function. A new graph is created for each training example, leaving advanced features such as variable input sized data, control flow, or recursion

to be handled by the host language. Gradient tracing is typically interleaved with model execution. Dynamic execution enables fast prototyping and simple debugging because the models are expressed in the host language. They can also draw benefit from all features of the host language as long as the operations are differentiable, allowing complex model design out of the box. The immediate drawback is of course the introduced overhead of reconstructing a graph for each example. The opportunity for whole-model optimization is lost. The graphs can however be kept lightweight as they are strictly defined (input size and structure is known) and use only basic nodes as complex features are handled by the host language. While this somewhat mitigates the drawback, graph creation and maintaining still represent a heavy cost for dynamic approaches.

TensorFlow

Introduced in late 2015 by Google Brain, TensorFlow aimed to be a tool for large scale machine learning. For Google, having the probably most vast amount of data available in the world, a machine learning interface had to be able to handle Google's massive production workload while also allowing flexibility for researchers and other users to experiment with model design. TensorFlow uses a dataflow-based programming abstraction. A model is represented by a directed graph. The user design the graph through an API, and then execute the graph with data. In a typical machine learning use case, the graph is defined once and then executed thousands or millions of times with some data. Each node in the graph represent an operation. The edges of the graph are tensors: n-dimensional arrays of some specified element type. Each operation has zero or more tensors as input and outputs zero or more tensors.

To understand the design choices of TensorFlow, a look at its predecessor is needed. DistBelief [28] was Google's previous system for large scale machine learning models. While DistBelief's programming model is similar to that of TensorFlow, the nodes in DistBelief are predefined layers, a composition of mathematical operators with adjoints defined at block level. By instead using primitive operations as graph nodes, TensorFlow enables programmers to write and experiment with their own layers without having to alter the core library. Another need TensorFlow addresses is the ability to refine existing training algorithm or even define new algorithms. Since DistBelief's origin in 2011, training algorithms have been an active area of research, and more flexibility in the implementation of the training algorithms was an important reason for the shift from DistBelief to TensorFlow. Nodes of

primitive operations are easy to define gradients for and allows users more fine grained control of the flow of the program, both in the backward and forward pass. TensorFlow thus originated partly as a result of more fine grained algorithmic differentiation. This development continues today with TensorFlow 2.0 enabling eager execution, i.e. dynamic graphs, by default with the option of converting the model to a static graph for training and deployment. Swift for TensforFlow is working on making AD a first class feature of the Swift language[29].

DistBelief uses a Python scripting interface to interact with layers. The layers were defined in C++ for performance reasons. For machine learning researchers seeking to experiment with new layer architectures, the use of a separate, less familiar language was seen as a barrier [7]. This is an illustrating example of the two language problem: expressive and flexible languages such as Python, R, or Matlab are used as scripting language for fast prototyping and development, but when high performance is needed, the program needs to be expressed in C or C++. For performance and portability, the core library of TensorFlow is also written in C++, however the scripting interface allows for much more fine-grained control of the operation composition.

TensorFlow use a tracing approach to automatic differentiation. Gradients are calculated by extending the TensorFlow graph in the following procedure: To compute, $\frac{\partial Y}{\partial I}$, the gradient of some output variable Y with respect to some variable X, first find the path in the graph from I to Y. Then, backtracking the path from Y to I, for each node in the path, add a new node in the TensorFlow graph computing the gradient function for that node. This is an implementation of reverse mode AD, a natural choice for an application aimed at large scale machine learning. Here, the number of input variables can be large and the number of output variables small, as for example in an image classification model. Each node in the gradient function sub-graph can take values from the forward path as input, in addition to previously computed partial derivatives. A simple example of this is a multiplication node. If the forward node defines the operation $f(u, v) = u * v$, the gradient function node computing $f'(u, v) = u * v' + u' * v$ is a function of both derivative values and primal values. This complicates memory management. Disregarding gradient computation, intermediate values in a typical neural net can be immediately disposed after being passed on to the next layer. However, values calculated early in the forward pass is often necessary late in the backward pass. These values hold on to memory resources, a resource that can be scarce, especially when operating on a GPU.

TensorFlow first define the model, i.e. the TensorFlow graph, and then ex-

ecute it. This deferred execution has the advantage that is allows graph optimizations before run time. Knowing the entire graph before execution, optimization routines such as common sub-expression elimination dead code elimination can be applied to the graph. Since the operations in the graph are typically repeated thousands or millions of times, TensorFlow may allow expensive optimization routines. Additionally, scheduling of operations increase performance because you are able to control memory usage and necessary data communication between devices.
With the TensorFlow graph effectively acting as a dataflow-based abstract syntax tree, TensorFlow is resembling its own separate programming language [30]. Instead of the usual Python syntax, the client is interacting with an underlying data structure through an API. This is especially apparent for control flow: conditional and iterative control flow is implemented through the `tf.cond` and `tf.while_loop` functions. The language duality can be a barrier for users as there are now two separate semantics to learn. It is thus compelling to question the necessity of this complexity in a machine learning library. Abadi argue that as machine learning applications benefit from programming language tools, machine learning appears more and more relevant to the design of new programming languages [30].

Zygote

The Julia programming language is designed to solve the two-language problem discussed in section 2.2 for technical computing. It aims to provide performance on the level of statically compiled languages as C and Fortran with the interactive and dynamic behavior of Python or LISP. This is done through careful language design with the main ingredient being a rich type system and multiple dispatch: aggressively customizing implementation based on the input types. Through its machine learning framework Flux[31], Julia is entering the machine learning stage promising high performance in a dynamic language. Flux developers argue that machine learning engineering is turning into a programming language problem. As researchers experiment with advanced stacks utilizing control flow, recursion, and data structures, the requirements for modeling tools are merging with requirements for programming languages. Flux thus becomes for not only machine learning, but for the more general paradigm of differentiable programming. Viewing machine learning models as simply differentiable algorithms, they should be expressed by a programming language, just like other algorithms are. Flux aims to be a simple and hackable ML framework by leveraging the fast Julia compiler. It is a dynamic define-by-run interface, and thus

differentiate itself from static graph building such as TensorFlow. Machine learning model are expressed not as a separate graph but rather directly in the Julia AST. Thus, all of the language features of Julia, such as control flow, custom data types, and macros, are supported. Flux, as the rest of base Julia, is written in Julia itself. This blurs the line between user and developer as anyone can read, understand and extend the library.
Zygote [32, 27] is the differentiation engine used by Flux, providing source-to-source AD for the ML framework. Claiming that the limitations of both static and dynamic approaches to AD are not inherent to AD itself, but rather to the Wengert list, Zygote employ a different approach to AD. Instead of working on improving differentiation of Wengert lists, they rather aim to differentiate code in Static Single Assignment form (SSA). SSA, first introduced in 1988[33], is a representation where each variable is assigned exactly once, and all variables are defined before they are used. It is used in many compilers as it enables efficient implementations of many compiler optimizations such as constant propagation and dead code elimination. Whereas the Wengert list is a linear record of elemental operations, SSA include control flow through a special joining function called the ϕ-function, generalizing the Wengert list. If the value to be used depend on the path of the program the theoretical ϕ-function will choose the correct value. In practice, compilers implement the ϕ-function with `goto`-blocks.
Consider the example explored in tables 2.1 and 2.1 where $f(a, b) = sin(a * b) + b$. Let $\mathcal{J}$ be a differentiation function such that for a function $y = f(x_1, x_2, ...)$, $\mathcal{J}(f, x_1, x_2, ...) = y, \mathcal{B}_y$ where $\mathcal{B}_y$ is a *pullback* function: it accepts the adjoint of y and outputs the adjoints of the inputs. The pullback function propagates the adjoint further back in the same way as each line in table 2.2: $\mathcal{B}_y(\bar{y}) = \bar{y}\frac{\partial y}{\partial x_1}, \bar{y}\frac{\partial y}{\partial x_2}...$ In other words, $\mathcal{J}$ calculate a step of the forward pass while also recording the necessary operations for the backwards pass symbolically. The pullback is not evaluated until it is called in the backwards pass.
Transforming the SSA form to use include $\mathcal{J}$, we end up with the following primal (forward) program:

$$\begin{aligned} \%1, \%2 &\leftarrow \mathcal{J}(*, a, b) \\ \%3, \%4 &\leftarrow \mathcal{J}(sin, \%1) \\ \%5, \%6 &\leftarrow \mathcal{J}(+, \%3, b) \end{aligned} \tag{2.9}$$

%1, %3, and %5 are values in the forward pass, while %2, %4, and %6 are pullback functions. Since there is no control flow, the representation is equal to that of a Wengert list, and AD can be performed as normal:

Table 2.3: Adjoint SSA program compared with reverse mode AD in mathematical notation. Underlined variables reference the primal program in equation 2.9.

SSA			Mathematical notation
$\%1, \%2$	$\leftarrow$	$\underline{\%6}(1)$	$\mathcal{B}_{w_5}(1) = (\frac{\partial w_5}{\partial w_2}, \frac{\partial w_5}{\partial w_4} = \bar{w}_4)$
$\%3$	$\leftarrow$	$\underline{\%4}(\%2)$	$\mathcal{B}_{w_4}(\bar{w}_4) = \bar{w}_4 \frac{\partial w_4}{\partial w_3} = \bar{w}_3$
$\%4, \%5$	$\leftarrow$	$\underline{\%2}(\%3)$	$\mathcal{B}_{w_3}(\bar{w}_3) = (\bar{w}_3 \frac{\partial w_3}{\partial w_1} = \bar{w}_1, \bar{w}_3 \frac{\partial w_3}{\partial w_2})$
$\%6$	$\leftarrow$	$\%1 + \%5$	$\bar{w}_2 = \frac{\partial w_5}{\partial w_2} + \bar{w}_3 \frac{\partial w_3}{\partial w_2}$
return %4, %6			

To understand the difference from differentiating Wengert lists, a branching example is useful. Consider the composite function

$$g(a, b) = \begin{cases} f(a, b) & b \geq 0 \\ a & otherwise \end{cases}$$

The forward pass require control flow based on the value of b. This is implemented with `goto` logic as in block 2.11. Note that block #2 is identical to block 2.9 and internally, this block is differentiated like table 2.3. Register %8 is used to record the actual path of the program so that the backwards traversal know which blocks it should pass through using a statement like "**goto** #x **if** %8", as shown in the right column of equation 2.11. Again, underlined references in the adjoint refer to the primal.

Primal program

block #1 :

$\%1 \leftarrow b < 0$

goto #3 **if** %1

block #2 :

$\%2,\%3 \leftarrow \mathcal{J}(*, a, b)$

$\%4,\%5 \leftarrow \mathcal{J}(sin, \%1)$

$\%6,\%7 \leftarrow \mathcal{J}(+, \%3, b)$

block #3 :

$\%8 \leftarrow \phi(\#1 \rightarrow false,\ \#2 \rightarrow true)$

$\%9 \leftarrow \phi(\#1 \rightarrow a,\ \#2 \rightarrow \%6)$

Adjoint program

block #1

goto #3 **if not** $\underline{\%8}$

block #2

equivalent to table 2.3

block #3

$\%7,\%8 \leftarrow \phi(\#1 \rightarrow (1, 0),\ \#2 \rightarrow (\%4, \%6))$

(2.11)

Julia's existing compiler use SSA form as part of the compilation process [9]. Since the output of Zygote is handed to an existing compiler, it it subject to its readily implemented optimization routines. As an example, consider the output in listing 2.2 for a simple function $f(x) = 3x + 2$. Here, Julia resolves the pullback functions for the $+$ and $*$ operators, and LLVM runs constant propagation on the SSA form and is able to reduce the gradient program to a single command, as the derivative is independent of the input variable.

Listing 2.2: Compiler optimization for gradient calculation at work. The interpreted program simply returns 3 for any input.

```
julia> @code_llvm gradient(x -> 3x + 2, 1)

define [1 x i64] @julia_gradient_18785(i64) {
    ret [1 x i64] [i64 3]
}
```

At the user end, the difference between Zygote and e.g. PyTorch is small. They will both play like a dynamic framework, allowing simple prototyping and fast development due to their interactive abilities. The goal of Zygote however is that the user should experience a significant speedup, especially when gradients are applied to non standard functions that are hard to optimize for a machine learning framework. A benefit of Zygote is that it can work with any external Julia package as Julia is a self-implemented language and most packages are also written in Julia itself. Thus, it will in the end break down to elementary Julia operations that, if differentiable, Zygote can handle. However, like Julia itself, Zygote is still young, and the developers ship it with a fair warning: Zygote is still under development and users should expect some hiccups. At the moment, array mutation is not supported.

Chapter 3

DNC

In 2016, Google's DeepMind proposed the idea of a Differentiable Neural Computer (DNC)[4]. The DNC was a continuation of the Neural Turing Machine (NTM) published in 2014[5]. Both the NTM and the DNC are recurrent neural networks; they contain a state transferring information across time steps and maps an input at time t to an output at time t. The complexity of the network state is what makes these networks stand out. They combine a neural network with an external memory section which the network will learn to utilize. Combining a neural net with a separate traditional style program allows these networks to enter the more generalized realm of differentiable programming (δP). In these cases, a system resembling a computer memory is controlled by the network through its output parameters. Memory operations are performed in a fuzzy manner by calculating weighted averages over the memory rows. The reason for this is differentiability. In a traditional program, a block of memory would simply be fetched by a read operation. This procedure is however not differentiable, hence the need for a fuzzy variant. The differentiability of the fuzzy memory operations allow for learning. With loss functions that depend on how well the network utilize its memory, the error gradient will be propagated through the entire program, from memory operations all the way through the neural net controller.

In this chapter, a detailed overview of the DNC is provided. The notation and equations are taken from the original paper on DNC[4]. Lastly, a comparison of implementation differences when using Julia compared to TensorFlow is discussed.

DNC Description

An overview of the DNC architecture is provided in figure 3.1.

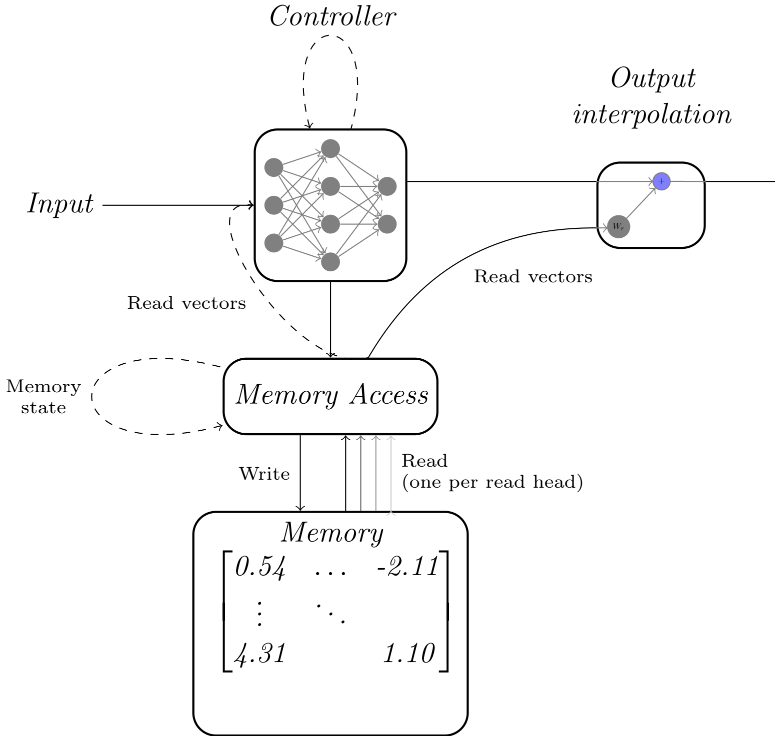


Figure 3.1: A overview of the DNC architecture. Dashed lines show recursive connections.
1

In each time step, the output of the controller is used as parameters for the reading and writing of memory. With an analogy to the Turing machine, the modules responsible for memory actions are called read and write heads. Each read head returns a read vector at each time step. The read vector is defined as a linear combination of each memory row [5]:

$$\mathbf{r}_t = \sum_i w_t^r(i)\mathbf{M}_t(i), \tag{3.1}$$

where $w_t^r(i)$ are the N elements of the read weightings $\mathbf{w}_t^r$ emitted by the read head. Since equation 3.1 is differentiable, the reading operation is

trainable; the sensitivity of the parameters determining $\mathbf{w}_t^r$ can be found and thereby the parameters can be tuned.
Memory writes are performed similarly although the procedure is slightly more complicated. The write head emits a write weighting $\mathbf{w}_t^w$ while the controller emits an erase vector $\mathbf{e}_t$ and a write vector $\mathbf{a}_t$. The erase vector describes by how much each memory word element should be erased. The add vector contains new values which should be written to the memory. The write weighting are weights over memory locations determining which words will be written to. Together, these vectors determine the write operation by the following equation:

$$\mathbf{M}_t = \mathbf{M}_{t-i} \circ (\mathbf{E} - \mathbf{w}_t^w \mathbf{e}^\top) + \mathbf{w}_t^w \mathbf{v}_t^\top \tag{3.2}$$

Here, $\circ$ denotes element wise multiplication and $\mathbf{E}$ a matrix of ones ($\mathbf{E}[i,j] = 1, \forall i,j$). The calculation of read and write weightings is described in the sections 3.1 and 3.1.
For a reference of all variables used in the DNC, see table 3.1 The domains $\mathcal{S}_N$ and Δ_N are defined as N-dimensional vectors where the elements respectively sum to 1 and sum to at most 1, as specified by equations 3.3 and 3.4.

$$\mathcal{S}_N = \{\boldsymbol{\alpha} \in \mathbb{R}^N : \boldsymbol{\alpha}_i \in [0,1], \sum_{i=1}^{N} \boldsymbol{\alpha}_i = 1\} \tag{3.3}$$

$$\Delta_N = \{\boldsymbol{\alpha} \in \mathbb{R}^N : \boldsymbol{\alpha}_i \in [0,1], \sum_{i=1}^{N} \boldsymbol{\alpha}_i \leq 1\} \tag{3.4}$$

Content based addressing

Content based addressing is calculated with the following equation:

$$\mathcal{C}(\boldsymbol{M}, \boldsymbol{k}, \beta) = \frac{\exp\{\mathcal{K}(\boldsymbol{M}[i,\cdot], \boldsymbol{k})\beta\}}{\Sigma_j \exp\{\mathcal{K}(\boldsymbol{M}[j,\cdot], \boldsymbol{k})\beta\}} \tag{3.5}$$

A key k that is compared with each row in memory using a similarity measure $\mathcal{K}$. A strength parameter β sharpens the similarity measure: a high strength value amplifies the values of the most similar rows. Cosine similarity is used as the similarity measure:

$$\mathcal{K}(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}||\mathbf{v}|} \tag{3.6}$$

Table 3.1: An overview of variables in the DNC.

Variable	Description	Domain
t	time step	$\mathbb{N}$
N	rows of memory	$\mathbb{N}$
W	memory word size	$\mathbb{N}$
R	number of read heads	$\mathbb{N}$
$\boldsymbol{x}_t$	input vector	$\mathbb{R}^X$
$\boldsymbol{y}_t$	output vector	$\mathbb{R}^Y$
z_t	target vector	$\mathbb{R}^Y$
$\boldsymbol{M}_t$	memory matrix	$\mathbb{R}^{N \times W}$
$\boldsymbol{k}_t^{r,i}$	read key $i (1 \leq i \leq R)$	$\mathbb{R}^W$
$\beta_t^{r,i}$	read strength i	$[1, \infty)$
$\boldsymbol{k}_t^w$	write key	$\mathbb{R}^W$
β_t^w	write strength	$[1, \infty)$
$\boldsymbol{e}_t$	erase vector	$[0, 1]^N$
$\boldsymbol{v}_t$	write vector	$\mathbb{R}^W$
f_t^i	free gate i	$[0, 1]$
g_t^a	allocation gate	$[0, 1]$
g_t^w	write gate	$[0, 1]$
$\boldsymbol{\psi}_t$	memory retention vector	$\mathbb{R}^N$
$\boldsymbol{u}_t$	memory usage vector	$\mathbb{R}^N$
$\boldsymbol{\phi}_t$	slot indices sorted by usage	$\mathbb{N}^N$
$\boldsymbol{a}_t$	allocation weighting	Δ_N
$\boldsymbol{c}_t^w$	write content weighting	$\mathcal{S}_N$
$\boldsymbol{w}_t^w$	write weighting	Δ_N
$\boldsymbol{p}_t$	precedence weighting	Δ_N
$\boldsymbol{L}_t$	link matrix	$\mathbb{R}^{N \times N}$
$\boldsymbol{f}_t^i$	forward weighting i	Δ_N
$\boldsymbol{b}_t^i$	backward weighting i	Δ_N
$\boldsymbol{c}_t^{r,i}$	read content weighting i	$\mathcal{S}_N$
$\boldsymbol{w}_t^{r,i}$	read weighting i	Δ_N
$\boldsymbol{\pi}_t^i$	read mode i	$\mathcal{S}_3$
W_r	read key weights	$\mathbb{R}^{(RW) \times Y}$
$\boldsymbol{\theta}$	controller weights	$\mathbb{R}^\Theta$
$\boldsymbol{\xi}_t$	interface vector	$\mathbb{R}^{(W \times R) + 3W + 5R + 3}$
$\boldsymbol{\chi}_t$	controller input vector	$\mathbb{R}^{(W \times R) + X}$

Writing memory

Writing to memory is done through an interpolation of the content based addressing scheme and *dynamic memory allocation* as per equation 3.7. The interpolation is controlled by the gate parameters $\boldsymbol{g}^a \in [0,1]^W$ and $\boldsymbol{g}^w \in [0,1]^W]$ emitted by the controller. The address weighting obtained by the content based addressing scheme is denoted $\boldsymbol{c}_t^w$, while $\boldsymbol{a}_t$ is the allocation based address weighting, detailed in this section.

$$\boldsymbol{w}_t^w = g_t^w[g_t^a \boldsymbol{a}_t + (1 - g_t^a)\boldsymbol{c}_t^w] \tag{3.7}$$

The dynamic memory allocation allows the machine to free memory and allocate new space. This is done through the maintenance of a usage vector $\boldsymbol{u}_t \in [0,1]^N$. In each iteration, the controller emits free gates f_t^i representing the degree to which the location last read by read head i can be freed. Thus, the *memory retention vector*, representing the degree each memory location will *not* be freed, is computed as follows:

$$\boldsymbol{\psi}_t = \prod_{i=1}^{R}(\mathbf{1} - f_t^i \boldsymbol{w}_{t-1}^{r,i}) \tag{3.8}$$

Then, denoting element-wise multiplication with $\circ$, the usage vector can be defined as:

$$\boldsymbol{u}_t = (\boldsymbol{u}_{t-1} + \boldsymbol{w}_{t-1}^w - \boldsymbol{u}_{t-1} \circ \boldsymbol{w}_{t-1}^w) \circ \boldsymbol{\psi}_t \tag{3.9}$$

Every write to a location increases its usage. It is decreased only through the free gates. Let $\boldsymbol{\phi}_t \in \mathbb{Z}^N$ be the indices of memory locations sorted by usage in ascending order. Then, $\boldsymbol{\phi}_t[1]$ is the index of the least used memory location. The allocation weighting is finally defined as follows:

$$\boldsymbol{a}_t[\boldsymbol{\phi}_t[j]] = (1 - \boldsymbol{u}_t[\boldsymbol{\phi}_t[j]]) \prod_{i=1}^{j-1} \boldsymbol{u}_t[\boldsymbol{\phi}_t[i]] \tag{3.10}$$

If all usages are 1, then $\boldsymbol{a}_t = \mathbf{0}$ and no new memory is allocated. Memory can still be written to depending on the value of the write gates g_t^w and g_t^a.

Reading memory

The read head have the capability of three different read modes: content mode, backward mode, and forward mode. Read mode weighting is emitted by the controller. $\boldsymbol{\pi}_t^i \in \mathcal{S}_3$ interpolates content, backward, and forward weighting. Backward and forward mode define read locations based on temporal linkage, allowing the read heads to iterate over memory locations in

the same (or, in backward mode, reverse) order they were written to. To do this, the system maintains a temporal link matrix $L \in \mathbb{R}^{N \times N}$. The link matrix L is defined such that location $L[i,j]$ represents the degree of which location i was written to after location j. Each row and each column sum to at most 1, so that they define a weighting over he possible locations. To calculate L, a precedence weighting describing the degree to which each location was last written to. Let p_t be such a precedence weighting at time step t. p is defined recursively as follows:

$$\begin{aligned} \boldsymbol{p}_0 &= \boldsymbol{0} \\ \boldsymbol{p}_t &= (1 - \sum_i \boldsymbol{w}_t^w[i])\boldsymbol{p}_{t-1} + \boldsymbol{w}_t^w \end{aligned} \tag{3.11}$$

$\boldsymbol{w}_t^w$ is the write weighting defined in equation 3.7.
With the precedence weighting, the link matrix can be defined:

$$\begin{aligned} L_0[i,j] &= 0 \qquad \forall i,j \\ L_t[i,i] &= 0 \qquad \forall i \\ L_t[i,j] &= (1 - \boldsymbol{w}_t^w[i])L_{t-1}[i,j] + \boldsymbol{w}_t^w[i]\boldsymbol{p}_{t-1}[j] \end{aligned} \tag{3.12}$$

Using the link matrix, forward and backward weighting can easily be calculated:

$$\begin{aligned} \boldsymbol{f}_t^i &= L_t \boldsymbol{w}_{t-1}^{r,i} \\ \boldsymbol{b}_t^i &= L_t^\top \boldsymbol{w}_{t-1}^{r,i} \end{aligned} \tag{3.13}$$

Finally, the read weighting can be calculated through an interpolation between the content, backward, and forward weightings:

$$\boldsymbol{w}_t^{r,i} = \boldsymbol{\pi}_t^i[1]\boldsymbol{b}_t^i + \boldsymbol{\pi}_t^i[2]\boldsymbol{c}_t^{r,i} + \boldsymbol{\pi}_t^i[3]\boldsymbol{f}_t^i \tag{3.14}$$

Julia Implementation of the DNC

As a hybrid between machine learning and traditional computing, the DNC is an excellent case study for algorithmic differentiation systems. The derivative needs to be propagated not only through well defined networks of simple algebraic structures, but also custom functions, which could contain complicated data manipulation. Ideally, the programmer should be able to program without worrying about adjoint calculation. As discussed in section 2.2, Zygote is in attempt at solving this for the Julia language, making algorithmic differentiation a first class feature of the language. In the implementation of

the DNC in Julia, strengths and weaknesses of Zygote were exposed. A big advantage is that used code does not need to be aware of Zygote in order to be differentiable. In practice, this means the programmer can use libraries that were written independently of Zygote.

Program architecture

Following DeepMind's original implementation, the methods are split into three groups:

- Addressing - handling the low level memory addressing methods. These include content based addressing (eq. 3.5), allocation weighting (eq. 3.10, and forward and backward weights.
- Access - implementing a memory access interface that writes and reads memory, returning read vectors
- Computer - implementing the module interface.

The original implementation is object oriented while this is functional. Julia is a functional language. While allowing custom types to be implemented as class-resembling structures, they do not have methods associated to them. Instead Julia use dynamic dispatch extensively, and users are encouraged to write multiple function definitions specialized for the input parameter types. It is argued that functions this way is associated with all of its input types, not just the type of its first argument. The module interface is the `Dnc` method, which is implemented as a recurrent cell to handle inputting previous iterations read vectors to the controller. The implementation follows Flux' standard implementation of recurrent cells and is therefore easily incorporated in the Flux ecosystem.
Figure 3.3 and 3.2 illustrate the data flow of the memory access module. Blue variables refer to variables from the controller as per figure 3.4 while red are variables stored from the last iteration. The write operation is performed first, allowing the read head to read data from this iteration if needed. The figures illustrates the operations nicely: the write weights are a linear combination of content based addressing weights and weights depending on memory allocation. The gates control whether to write at all (g^w) and the factor at which the write should use allocation weights (g^a). The read operation is similar as it is a linear combination of content weights and temporal weights. The read mode parameter π sets the weights for the linear combination. Since temporal weights are decided by the previous read weights (as e.g. forward read means to read the location that was written to after

the last read), only content key and strength need to be provided by the controller.

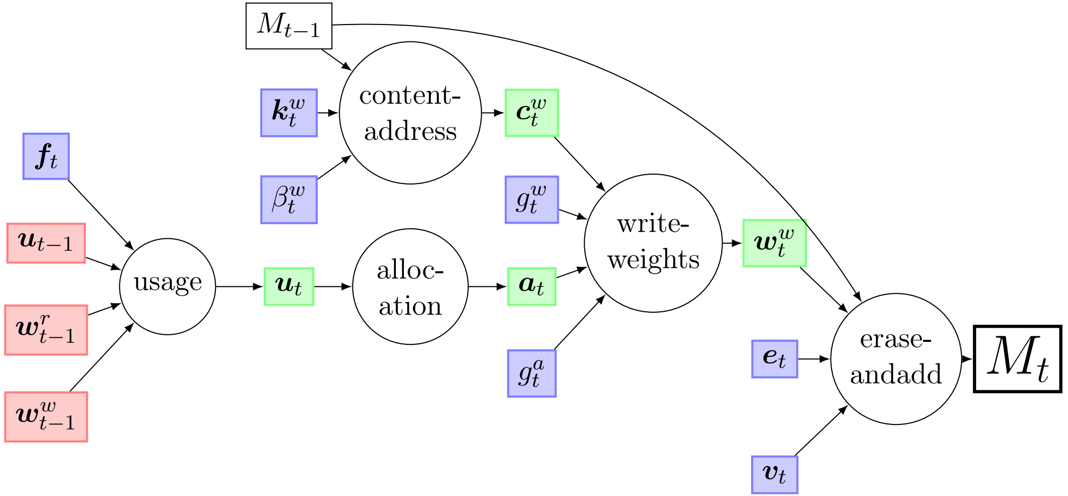


Figure 3.2: Dataflow graph for the memory writing operation. Circles correspond to functions in the Julia implementation. Red variables are state variables from the previous iteration, blue are input variables from the controller and green intermediate variables. See table 3.1 for a complete overview of the variables used.

Method Implementations

Splitting Controller Output

The output from the controller is to control the behavior of the memory access. The number of parameters needed grows with memory size and the number of read heads. As an example, the read key $\boldsymbol{k}_t^r$ is of size (W, R), where W is memory word size and R the number of read heads, requiring WR parameters from the controller alone. Using LSTM as the model controller, it is desirable to limit its number of output parameters for performance. For this reason, the DNC is initialized with an upper limit on controller size. Following the original implementation, an extra layer is added to transform the fixed-sized output to the necessary parameters. This is implemented with one linear transformation for each parameter, as shown in figure 3.4. The weights of the linear transformations are added to the set of trainable variables.

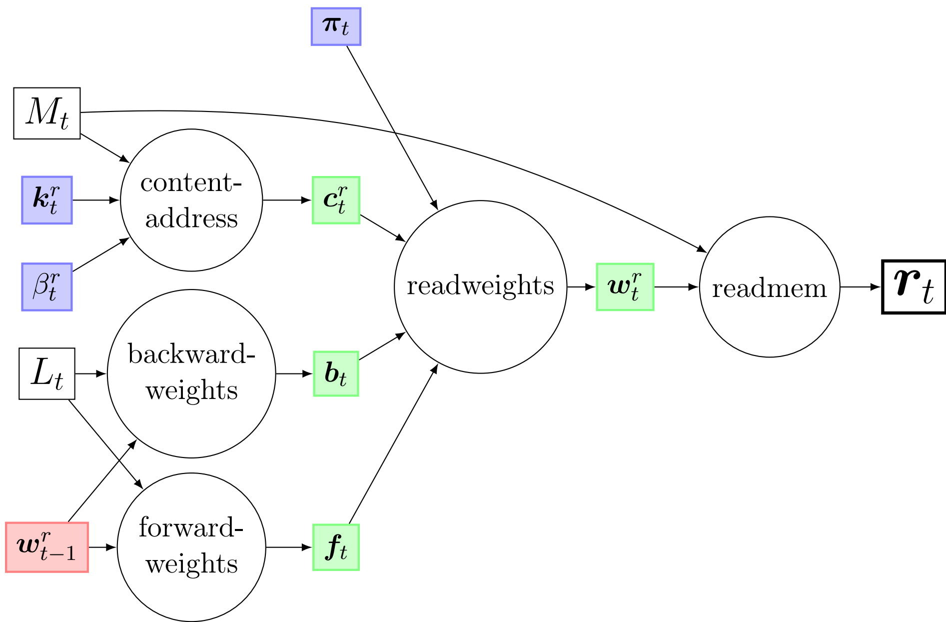


Figure 3.3: Dataflow graph for read vector calculation. Circles correspond to functions in the Julia implementation. Red variables are state variables from the previous iteration, blue are input variables from the controller and green intermediate variables. See table 3.1 for a complete overview of the variables used.

Content Based Addressing

Content based addressing is the most complex operation performed by the read and write heads. It requires the comparison of a key with each row of memory, followed by a normalisation calculation using the `softmax` function. An initial, intuitive implementation of content address is shown in listing 3.1

Listing 3.1: Naïve content address implementation.

```
cosinesim(u, v) = dot(u, v)/(norm(u)*norm(v))
weighted_softmax(xs, weight) = softmax(xs.*weight)
function contentaddress(key, M, β, K=cosinesim)
xs = [K(key, row) for row in eachrow(M)]
weighted_softmax(xs, β)
end
```

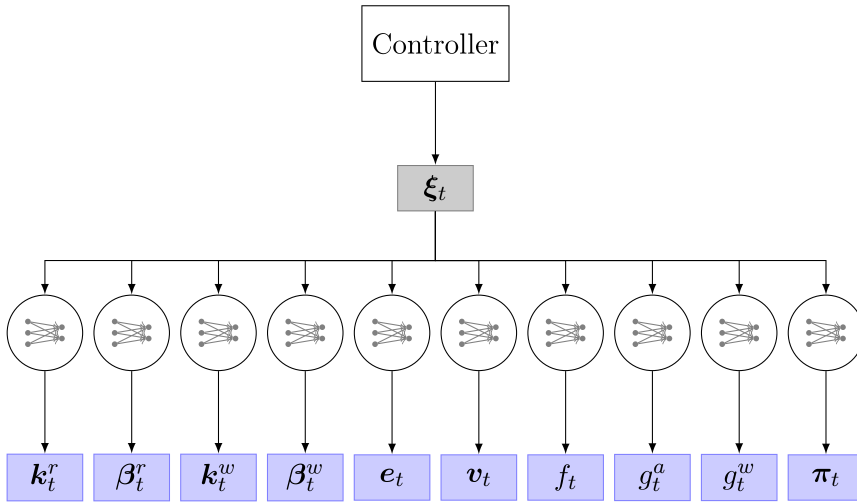


Figure 3.4: The controller is extended with a layer of independent linear transformations to obtain the memory access parameters. Variable names refer to table 3.1.

This naïve implementation reads practically as equation 3.5. While simple, the implementation has some drawbacks. The main issue nonetheless is that the `eachrow` function is not automatically differentiable with Zygote. Although this is not necessarily a problem here as the above solution is far from optimal anyway, it signals that the programmer needs to be trained to know how to write differentiable code. When the design philosophy is for AD to work out of the box on arbitrary Julia code, this could be troublesome. We repeat however that Zygote is a work in progress, and expanding the code base which allows differentiation is an active are of work. Secondly, by assuming that key is a one dimensional array, it is unable to handle both multiple keys, and batched inputs, i.e. a matrix or three dimensional tensor of keys. While this could be dealt with by broadcasting the function across a list of key-memory-pairs, multiple accesses to the same memory will hurt performance. Julia users are generally recommended to write loops explicitly, as loops are fast and efficient, contrasting other tools like R and Matlab. In this use case, however, looping will miss out on possible matrix multiplications which are implemented much more efficiently than the equivalent sequence of matrix-vector-multiplications. Changing `xs = [K(key, row) for row in eachrow(M)]` in listing 3.1 to an appropriate call to the `pairwise!` function in listing 3.2, allows for the handling of batched input

with multiple keys.

Listing 3.2: Pairwise comparison of columns of k and rows of M.

```
function _pairwise!(r::Zygote.Buffer,
                    col::AbstractArray{T, 3},
                    row::AbstractArray{T, 3},
                    β::AbstractArray{T, 2}) where T
    nrow = size(row, 1)
    ncol = size(col, 2)
    batchsize = size(col, 3)
    @inbounds for k = 1:batchsize
        @inbounds for j = 1:ncol
            colj = view(col, :, j, k)
            for i = 1:nrow
                rowi = view(row, i, :, k)
                r[i, j, k] = cosinesim(rowi, colj, β[j, k])
            end
        end
    end
    r
end
```

The function simply extracts the correct column and row from the keys `k` and the memory `M` respectively and applies the cosine similarity. Note the use of type `Buffer` for the return value: this is a workaround implemented to overcome Zygote's inability to handle array mutation. While solving the need to support three dimensional keys and memory, we still do not utilize matrix multiplication, and performance suffers. On simple examples using realistic memory and batch size, this implementation of content based addressing allocates 144 MiB memory in the backwards pass and takes $66ms$ to complete. Such a memory leak is unacceptable in production. Eventually, this was reduced to $286\mu s$ and 624 KiB, a 23000% improvement. To solve this, a closer look on the cosine similarity metric is useful. The nominator is a dot product, which for a matrix $\boldsymbol{k}$ with multiple read keys is simply a matrix multiplication $\boldsymbol{Mk}$. The denominator is the product of the normalized vectors, which generalizes to the matrix product of the reduced tensors. Listing 3.3 shows this far superior implementation; gradient calculation is approximately 70 times faster than the pairwise implementation and doesn't suffer from a memory leak. The reason for this is the underlying BLAS function calls that the batched multiplication function from `NNlib.jl` utilizes.

Listing 3.3: Julia implementation of content based addressing

```
function contentaddress(k, M, β)
```

```
    dot = batched_mul(M, k)
    n_k = sum(k.^2, dims=1)
    n_M = sum(M.^2, dims=2)
    np = batched_mul(n_M, n_k))
    norm = sqrt.(normprod)
    β = reshape(β, 1, size(β)...)
    weightedsim = dot.* β./ norm
    softmax(weightedsim; dims=1)
end
```

For comparison, the original implementation from DeepMind's DNC is included in listing 3.4.

Listing 3.4: Tensorflow implemention of content based addressing.

```
def _vector_norms(m):
squared_norms = tf.reduce_sum(m * m, axis=2, keepdims=True)
return tf.sqrt(squared_norms + _EPSILON)

def weighted_softmax(activations, strengths, strengths_op):
transformed_strengths = tf.expand_dims(strengths_op(strengths), -1)
sharp_activations = activations * transformed_strengths
softmax = snt.BatchApply(module_or_op=tf.nn.softmax)
return softmax(sharp_activations)

def _build(self, memory, keys, strengths):
# Calculates the inner product between the query vector and words in memory.
dot = tf.matmul(keys, memory, adjoint_b=True)

# Outer product to compute denominator (euclidean norm of query and memory).
memory_norms = _vector_norms(memory)
key_norms = _vector_norms(keys)
norm = tf.matmul(key_norms, memory_norms, adjoint_b=True)

# Calculates cosine similarity between the query vector and words in memory.
similarity = dot / (norm + _EPSILON)

return weighted_softmax(similarity, strengths, self._strength_op)
```

Both implementations use batched matrix multiplication to compute the dot products of key and memory rows. The denominator $|u||v|$ is also computed using batched matrix multiplication. The main difference is that the Julia implementation calls native Julia methods such as 'sum' and 'sqrt'. In the python implementation, calls to TensorFlow methods are needed for these basic operations so that they become part of the TensorFlow graph.

Chapter 4

Results

The Julia DNC implementation is a define-and-run version of the DNC with an easy-to-understand yet high performing code. Extending the machine learning library Flux, `DNC.jl` is readily incorporated into other models. It could for example be passed as any other layer in a Flux `Chain`, or be used in a neural differential equation using `DiffEqFlux.jl`[13]. An attempt at training the model on the repeated copying task from the original DNC paper[4] was made, yet no satisfying results were achieved. Rather, the model quickly converged to a trivial state of not reading the memory at all. The reasons for this is unknown. Through rigorous testing, the correctness of each module method is ensured, so the problem could be with the repeat-copy-implementation, and not the model itself. The performance metrics are therefore still considered valid.

Benchmarks

Table 4.1 presents the computation of a selection of methods used in the Julia implementation of DNC. All becnhmarks are done using eight Intel Core i7-4720HQ CPUs operating at 2.60GHz. The `@btime` macro from `BenchmarkTools.jl` was used to benchmark the methods. The method names refer to the Julia implementation and may differ from DeepMind's original implementation. The "Addressing"-methods are the most low level: they are called by methods in the "Access"-module. For example, `readweights` will call `contentaddress`, `forwardweight`, `backwardweight` and then interpolate these weights based on the readmode-paramater emitted by the controller. The gradient calculation time range between 2 and 4 times the equivalent forward calculation. The

relatively fastest gradient calculation is the `writeweights` method, which compute content based write address using `contentaddress` and allocation based weighting using `allocationweighting`. Since `allocationweighting` include a sorting operation, it is not differentiable and skipped in the backwards pass, hence the good performance. `contentaddress` is by far the most costly addressing operation, as it requires a pairwise comparison of all memory rows with all keys in addition to a weighted softmax operation along each column of the result. The achieved speed was only possible by converting the similarity measure to a matrix multiplication which have a fast adjoint implementation in `NNlib.jl`.

Table 4.1: Benchmarks of forward and gradient calculations of methods in the DNC. "G/F" column is the slowdown of gradient calculation compared with the forward pass only. Parameters: X=6, Y=5, N=16, W=64, R=4, batchsize=16.

Method	**Time**			**# alloc.**		**Alloc. memory**	
	Forw	Grad	G/F	Forw	Grad	Forw	Grad
ADDRESSING							
contentaddress	89.8μs	286μs	3.18	45	3254	189 KiB	624 KiB
memoryretention	8.35μs	20.8μs	2.49	37	91	10.6 KiB	32.6 KiB
usage	11.3μs	36.2μs	3.20	56	149	18.1 KiB	49.4 KiB
precedenceweight	5.06μs	12.4μs	2.45	24	71	5.27 KiB	11.3 KiB
forwardweight	13.6μs	57.5μs	4.23	8	165	24.6 KiB	52.9 KiB
backwardweight	35.6μs	85.3μs	2.40	128	299	31.6 KiB	61 KiB
ACCESS							
eraseandadd	76.5μs	230μs	3.00	29	252	267 KiB	609 KiB
readmem	118μs	283μs	2.40	161	365	93.5 KiB	172 KiB
writeweights	125μs	263μs	2.10	932	1913	126 KiB	442 KiB
readweights	107μs	399μs	3.73	163	3571	128 KiB	674 KiB
MemoryAccess	585μs	2.18ms	3.73	1658	7552	570 KiB	2.20 MiB
DNC							
DNC	1.20ms	4.20ms	3.50	1682	7829	693 KiB	2.93 MiB

Benchmarks for the Julia implementation are compared to DeepMind's implementation of the DNC in TensorFlow in table 4.2. TensorFlow benchmarks are achieved by recreating the necessary sections of the complete computational graph and running `tf.gradients` 1000 times using the `timeit` module, averaging elapsed time. Zygote beats TensorFlow in every internal method, yet for the complete graph calculation performance is equal.

Table 4.2: Timing of gradient calculations of methods in both Julia and TensorFlow implementation of DNC. `BenchmarkTools.@btime` is used in the Julia implementation and the `timeit` module for the TensorFlow implementation. Parameters: X=6, Y=5, N=16, W=64, R=4, batchsize=16.

Method	**Zygote**	**TensorFlow**	**TF/Zygote**
ADDRESSING			
contentaddress	286μs	734μs	2.57
usage	36.2μs	449μs	12.4
forwardweight	57.5μs	443μs	7.70
backwardweight	85.3μs	435μs	5.10
ACCESS			
eraseandadd	230μs	1475μs	6.41
readmem	283μs	306μs	1.08
writeweights	263μs	811μs	3.08
readweights	399μs	1015μs	2.54
MemoryAccess	2.18ms	3.22ms	1.48
DNC			
DNC	4.20ms	4.28ms	1.02

This could be a testament to the optimization features of TensorFlow. The method `readmem` is simply a batched matrix multiplication of memory and read weights. In the TensorFlow version, this is done with a single call to `tf.matmul` with the memory transposed, while the Julia implementation use `batched_mul` from `NNlib.jl` achieving equal results. Another interesting thing to note is `forwardweight` and `backwardweight`. From section 3.1, we have that $\boldsymbol{f}_t = L_t \boldsymbol{w}_{t-1}^r$ and $\boldsymbol{r}_t = L_t^\top \boldsymbol{w}_{t-1}^r$. TensorFlow appear to handle the matrix transpose very well, while the Julia implementation pay a small penalty to do the matrix transposition. The usage operation heavily outperforms TensorFlow, but this is with good reason. The Julia code assumes single write head only, meaning that the usage can operate on two dimensional matrices only. In the original TensorFlow, multiple write heads are allowed, although the benchmark is performed with only one.

Chapter 5

Discussion

When the Neural Turing Machine was introduced in 2014, it represented a new way of thinking about machine learning. Chasing generalizability, the idea was that a program designed to mimic the behavior of a computer while still being fully end-to-end trainable could learn to solve complex problems based on data alone. DNC, the descendant of NTM, improves its memory access methods and show intriguing results on a number of complex problem solving tasks. DNC nicely represents the transition from the deep learning paradigm to the differentiable programming paradigm. In this new paradigm, deep learning networks are merely a single case of differential equations modelling the solution to a given problem. They could be extended or replaced by any other differentiable construct. This transition require powerful and flexible differentiation tools. Problems requiring a full extent of language features will also require the differentiation engine to handle the full language. In the DNC, a neural network work nicely together with a traditional program manipulating and reading data in a matrix. Although initially a TensorFlow program, the DNC is considered a good fit for the Julia language for a number of reasons. First of all, it is simple to prototype and test due do Zygote providing efficient reverse mode AD on-the-fly. Julia's expressiveness combined with its C-like performance gives it an edge when handling complex models. As Zygote play effortlessly with most Julia packages, the developer is free to explore a wide range of libraries so minimize time spent reinventing the wheel. Lastly, Julia's growing ecosystem of machine learning and other scientific programming tools makes it interesting to have a DNC written in Julia available to use as a component of other applications.

Section 3.2 present the implementation process of DNC.jl. Some pitfalls were identified, and it was shown that with an inaccurate implementation performance could severely degrade and sometimes minor changes can lead to great improvements in runtime. To understand why, knowledge of the intermediate code representation that Zygote transforms is required. This can not and should not be expected from most users, but is not necessarily an implementation problem. More extensive documentation could guide the programmer to the correct solutions.

The benchmarking results show that a high performance model is possible in a dynamic framework, beating the TensorFlow implementation by a factor of up to seven on comparable methods. It is however interesting to note that the runtime on training the entire model is practically equal. This could be a testament to TensorFlow's powerful optimization and job scheduling, but it could also signal room for improvement in the Julia implementation. It seems to include extra overhead when incorporating the low level methods in a complete module call. The benchmarks show execution time on a multicore CPU. Given TensorFlow's benefit of being able the graph for optimal job scheduling on a heterogeneous system, is may be expected that it would outperform the Julia implementation for large training datasets on GPUs or TPUs. This is not tested as part of this project, but would be an interesting question for further work.

Working with benchmark creation of both implementations expose the difference in experience working with a static and dynamic framework. The Julia implementation, composed of eagerly executed functions, could easily be measured by simply running the functions with random input. In TensorFlow, manual recreation of the correct sub-graphs was needed in order to isolate the correct gradient calculation. One might argue that this is a matter of personal preference, however with TensorFlow 2 moving towards eager execution as default, it seems to be commonly accepted that dynamic declaration is faster and easier to prototype and develop.

The DNC suits as an illustrating example of the transition from deep learning to differentiable programming. While its generality and applicability to diverse problems is interesting enough in its own, it also opens the door to explore new opportunities with differentiable programs. Julia, with Flux and Zygote at the forefront, has a strong appeal for programmers in the δP paradigm as it offers an intuitive high-level interface while providing state-of-the-art performance. The adoption of DNC into the Julia ecosystem acts as an example of its ability to handle complex models with ease. The Julia implementation show that dynamic languages have the capability to compete with statically declared graphs if programmed correctly. However,

inaccuracies can cause significant bottlenecks. Model development benefits from the ability to get quick feedback and inspect data iteratively. While TensorFlow produce efficient model training at compile time, it is more to difficult to inspect, debug and test. In the perspective of δP, it can be useful to have an efficient implementation of the DNC in Julia as its growing ecosystem of tools for differentiable programming may be able to utilize in new applications and combinations. It should also be noted that TensorFlow version 2 is providing eager execution by default, so porting the original implementation to the new version is also a valid option.

Chapter 6

Conclusion

The implementation of algorithmic differentiation has been discussed and compared with a special focus on machine learning frameworks. In broad strokes, they can be split in two groups based on whether they compile a static computational graph and the execute it on input data or define the computation graph dynamically on-the-fly. Among the static frameworks, TensorFlow is probably the largest and most popular toolkit. It has been successfully deployed in a wide range of applications and power Google's many machine learning programs. The original implementation of DNC used TensorFlow, but an argument is here made for using the Julia programming language with Zygote as its differentiator. Zygote use dynamic declarations, but with a slightly different approach than other dynamic frameworks: it differentiates code in SSA from using source code transformation instead of the traditional method of tracing the program execution using a Wengert list. Zygote is looking to be a pillar of what Julia is trying to achieve: becoming a platform for research, prototyping, and deployment of scientific computing models, with algorithmic differentiation as a first-class feature of the language.

Through the reimplementation of the Differentiable Neural Computer, a thorough comparison of TensorFlow and Flux, Julia machine learning framework, was possible. Differences between static and dynamic programs become abundantly clear when working on both in parallel. The benefits of fast feedback and simple debugging is highlighted in the discussion. Although reaching a high performance Julia version, the implementation process revealed some pitfalls and quirks when working with Zygote. Some were due to the fact that Zygote is young and under development. Nevertheless, it

was apparent that choosing suboptimal method implementations could result in huge performance penalties. Additionally, extra care was needed when dealing with arrays, as mutation of arrays is not supported by Zygote. This lead to some unexpected errors both on self-written code, but more importantly, on imported library code. While workarounds were possible, these errors undermine the goal of allowing language-complete model complexity. The final model outperforms the original implementation gradient calculation in every internal method, and trains equally fast on the CPU. Training on GPUs or TPUs was not measured. Due to TensorFlow's job scheduling algorithms enabled by the static graph, it is assumed that its training times on heterogeneous systems will be difficult to beat. However, Flux offer first-class GPU support and can even run on TPUs, so this would an interesting research question for further work.
A benefit of the Julia DNC is that it is easily extendible and adjustable. The codebase is small, as is the codebase for Flux. This allows anyone to adjust the cod for their needs. Having a DNC available in the Julia ecosystem can be helpful for researchers wishing to quickly prototype extensions or modifications of the model, or even combine it with other models or programs.

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[2] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Advances in neural information processing systems*, pp. 3104–3112, 2014.

[3] Y. LeCun. Facebook post, 2018.

[4] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, pp. 471–476, 2016.

[5] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[6] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *J. of Math*, vol. 58, no. 345-363, p. 5, 1936.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)a*, pp. 265–283, 2016.

[8] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[9] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017.

[10] M. Murphy, "Octave: A free, high-level language for mathematics," *Linux journal*, vol. 1997, no. 39es, p. 8, 1997.

[11] R. Ihaka and R. Gentleman, "R: a language for data analysis and graphics," *Journal of computational and graphical statistics*, vol. 5, no. 3, pp. 299–314, 1996.

[12] A. Pal, "RayTracer.jl: A Differentiable Renderer that supports Parameter Optimization for Scene Reconstruction," 2019.

[13] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit, "Diffeqflux.jl - A julia library for neural differential equations," *CoRR*, vol. abs/1902.02376, 2019.

[14] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *arXiv:1607.07892 [cs.MS]*, 2016.

[15] R. E. Wengert, "A simple automatic derivative evaluation program," *Communications of the ACM*, vol. 7, no. 8, pp. 463–464, 1964.

[16] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon, "Automatic differentiation of algorithms," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1-2, pp. 171–190, 2000.

[17] W. K. Clifford, "Preliminary sketch of biquaternions," *Proceedings of the London Mathematical Society*, vol. s1-4, no. 1, pp. 381–395, 1871.

[18] L. Hascoet and V. Pascual, "The tapenade automatic differentiation tool: Principles, model, and specification," *ACM Transactions on Mathematical Software (TOMS)*, vol. 39, no. 3, pp. 1–43, 2013.

[19] F. Bastien, P. Lamblin, R. Pascanu, J. Bergstra, I. Goodfellow, A. Bergeron, N. Bouchard, D. Warde-Farley, and Y. Bengio, "Theano: new features and speed improvements," *arXiv preprint arXiv:1211.5590*, 2012.

[20] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678, 2014.

[21] K. S. Tai, R. Socher, and C. D. Manning, "Improved semantic representations from tree-structured long short-term memory networks," *arXiv preprint arXiv:1503.00075*, 2015.

[22] X. Liang, X. Shen, J. Feng, L. Lin, and S. Yan, "Semantic object parsing with graph lstm," in *European Conference on Computer Vision*, pp. 125–143, Springer, 2016.

[23] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *arXiv preprint arXiv:1812.08434*, 2018.

[24] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[25] S. Tokui, K. Oono, S. Hido, and J. Clayton, "Chainer: a next-generation open source framework for deep learning," in *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, vol. 5, pp. 1–6, 2015.

[26] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, *et al.*, "Dynet: The dynamic neural network toolkit," *arXiv preprint arXiv:1701.03980*, 2017.

[27] M. Innes, A. Edelman, K. Fischer, C. Rackauckus, E. Saba, V. B. Shah, and W. Tebbutt, "Zygote: A differentiable programming system to bridge machine learning and scientific computing," *arXiv preprint arXiv:1907.07587*, 2019.

[28] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, pp. 1223–1231, 2012.

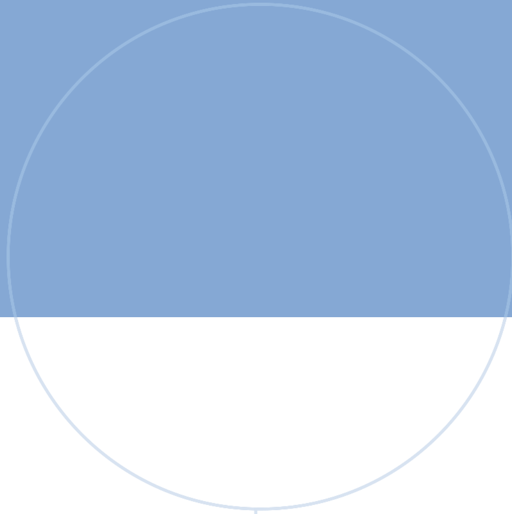
[29]

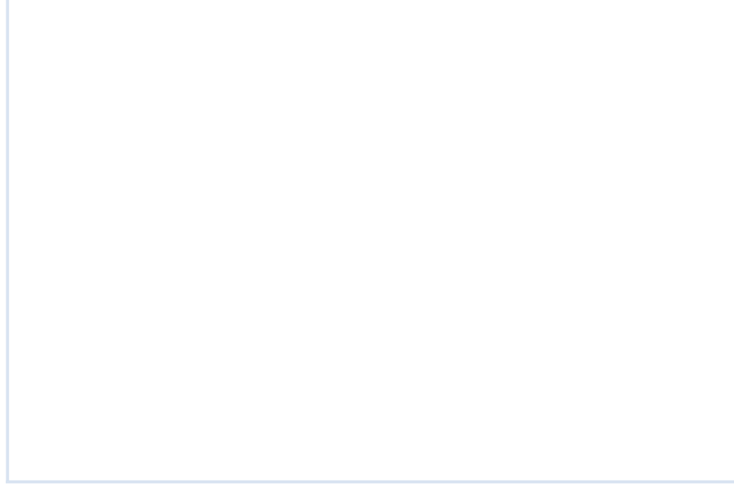
[30] M. Abadi, M. Isard, and D. G. Murray, "A computational model for tensorflow: an introduction," in *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, pp. 1–7, 2017.

[31] M. Innes, "Flux: Elegant machine learning with julia," *Journal of Open Source Software*, 2018.

[32] M. Innes, "Don't unroll adjoint: differentiating ssa-form programs," *arXiv preprint arXiv:1810.07951*, 2018.

[33] B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Global value numbers and redundant computations," in *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 12–27, 1988.





NTNU
Norwegian University of
Science and Technology