NATIONAL AND KAPODISTRIAN UNIVERSITY OF ATHENS

SCHOOL OF SCIENCES
DEPARTMENT OF INFORMATICS AND TELECOMMUNICATIONS

BSc THESIS

An Automatic Differentiation Algorithm for Recursive Functions and its Implementation in TensorFlow

Georgios P. Vasilakopoulos

Supervisors: **Calliope Kostopoulou,** PhD Candidate, Columbia University
Angelos Charalambidis, Assistant Professor, HUA
Panagiotis Rondogiannis, Professor, NKUA

ATHENS

JUNE 2024

ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αλγόριθμος για Αυτόματη Παραγώγιση Αναδρομικών Συναρτήσεων και η Υλοποίηση του στο TensorFlow

Γεώργιος Π. Βασιλακόπουλος

Επιβλέποντες: **Καλλιόπη Κωστοπούλου,** Υποψ. Διδάκτωρ, Πανεπιστήμιο Columbia
Άγγελος Χαραλαμπίδης, Επίκουρος Καθηγητής, ΧΠΑ
Παναγιώτης Ροντογιάννης, Καθηγητής, ΕΚΠΑ

ΑΘΗΝΑ

ΙΟΥΝΙΟΣ 2024

BSc THESIS

An Automatic Differentiation Algorithm for Recursive Functions and its Implementation in TensorFlow

Georgios P. Vasilakopoulos

S.N.: 1115202000018

SUPERVISORS: **Calliope Kostopoulou,** PhD Candidate, Columbia University
Angelos Charalambidis, Assistant Professor, HUA
Panagiotis Rondogiannis, Professor, NKUA

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Αλγόριθμος για Αυτόματη Παραγώγιση Αναδρομικών Συναρτήσεων και η Υλοποίηση του στο TensorFlow

Γεώργιος Π. Βασιλακόπουλος

Α.Μ.: 1115202000018

ΕΠΙΒΛΕΠΟΝΤΕΣ: **Καλλιόπη Κωστοπούλου,** Υποψ. Διδάκτωρ, Πανεπιστήμιο Columbia
Άγγελος Χαραλαμπίδης, Επίκουρος Καθηγητής, ΧΠΑ
Παναγιώτης Ροντογιάννης, Καθηγητής, ΕΚΠΑ

ABSTRACT

In recent years, due to significant advancements in computational power, the machine learning community is experimenting with increasingly complex algorithms that are designed to detect patterns in data. Under these circumstances, modern machine learning frameworks are expected to be *expressive*, so as to provide tools for defining arbitrarily complicated neural architectures. At the same time, in order for such models to be trainable, each underlying framework needs to implement certain optimization algorithms that rely on computing the derivatives of functions, through a process called Automatic Differentiation. What's more, the majority of popular frameworks is exploiting parallel data processing, in order to accelerate the training and inference procedures of machine learning algorithms.

TensorFlow, in particular, is a framework that operates based on the dataflow computational model, which is an alternative programming paradigm that is designed to parallelize computations, so that they can run efficiently in distributed execution environments. However, this inherent design pattern differs fundamentally from traditional, imperative style programming. As a result, efficiency has come at the expense of ease of use, and the framework has attempted to bridge this gap by introducing various features that aim to adapt dataflow constructs into imperative execution. In spite of these efforts to provide expressiveness, as of version 2.16, the framework officially [14] lacks support for recursive function definitions. Recursion is generally regarded as a powerful programming abstraction and is supported in most high level programming languages. For that reason, we believe that TensorFlow, as a machine learning framework, could benefit from supporting such a common feature that programmers are familiar with.

This thesis aims to propose a systematic way of handling recursive function definitions in TensorFlow, in a manner that is compatible and consistent with the dataflow computational model, upon which the framework is based. A significant portion of this work addresses the problem of implementing Automatic Differentiation on such functions, so that they can be used in practical situations, for machine learning.

SUBJECT AREA: Dataflow Computational Model

KEYWORDS: dataflow, tensorflow, recursive functions, automatic differentiation, static dataflow graphs

ΠΕΡΙΛΗΨΗ

Τα τελευταία έτη, λόγω αξιοσημείωτης προόδου σε υπολογιστικές δυνατότητες, η κοινότητα της μηχανικής μάθησης πειραματίζεται με ολοένα και πιο περίπλοκους αλγορίθμους που έχουν σκοπό την αναγνώριση μοτίβων σε δεδομένα. Υπο αυτές τις συνθήκες, οι σύγχρονες βιβλιοθήκες μηχανικής μάθησης οφείλουν να παρέχουν *εκφραστικότητα*, προσφέροντας εργαλεία τα οποία καθιστούν εφικτή την υλοποίηση οποιασδήποτε πολύπλοκης νευρωνικής αρχιτεκτονικής. Ταυτόχρονα, προκειμένου η οποιαδήποτε δομή να είναι εκπαιδεύσιμη, η εκάστοτε βιβλιοθήκη πρέπει να υλοποιεί ορισμένους αλγορίθμους βελτιστοποίησης, οι οποίοι, κατά κανόνα, βασίζονται στον υπολογισμό παραγώγων, μέσω μίας διαδικασίας που ονομάζεται Αυτόματη Παραγώγιση. Επιπλέον, οι περισσότερες βιβλιοθήκες αξιοποιούν την παράλληλη επεξεργασία δεδομένων, με σκοπό να επιταχύνουν τις διαδικασίες εκπαίδευσης και συμπερασμού στη μηχανική μάθηση.

Το TensorFlow, συγκεκριμένα, είναι μια βιβλιοθήκη η οποία λειτουργεί βάσει του υπολογιστικού μοντέλου dataflow, που είναι ένα εναλλακτικό μοντέλο, το οποίο διευκολύνει την παραλληλοποίηση υπολογισμών, με σκοπό την ταχύτερη διεκπεραίωση σε κατανεμημένα υπολογιστικά συστήματα. Παρ'όλ'αυτά, η εγγενής φύση αυτού του μοντέλου διαφέρει ριζικά απο το παραδοσιακό μοντέλο του προστακτικού προγραμματισμού. Ως εκ τούτου, η ευκολία χρήσης θυσιάζεται χάριν της απόδοσης, ενώ έχουν γίνει σημαντικές προσπάθειες κάλυψης αυτού του χάσματος, με την εισαγωγή μηχανισμών που επιχειρούν την ενσωμάτωση των δομών dataflow σε προστακτική εκτέλεση. Όμως, παρά τις διάφορες προσπάθειες βελτίωσης της εκφραστικότητας, η βιβλιοθήκη επίσημα δεν υποστηρίζει αναδρομικούς ορισμούς συναρτήσεων, εως και στην έκδοση 2.16. Η αναδρομή, γενικώς, θεωρείται μια εξαιρετικά εύελικτη προγραμματιστική τεχνική, ενω οι περισσότερες γλώσσες προγραμματισμού υψηλού επιπέδου την υποστηρίζουν. Για τον λόγο αυτό πιστεύουμε οτι το TensorFlow, μπορεί να επωφεληθεί εισάγοντας μια τόσο διαδεδομένη λειτουργικότητα στο μοντέλο της.

Η παρούσα πτυχιακή έχει στόχο να παρουσιάσει ένα συστηματικό τρόπο διαχείρισης αναδρομικών ορισμών συναρτήσεων στο TensorFlow, με ένα τρόπο συμβατό αλλά και συνεπή με το υπολογιστικό μοντέλο dataflow, επι του οποίου το TensorFlow βασίζεται. Ένα μεγάλο μέρος της εργασίας αναφέρεται στο πρόβλημα υλοποίησης αλγορίθμου Αυτόματης Παραγώγισης για τέτοιες συναρτήσεις, ώστε αυτές να μπορούν να εφαρμοστούν σε πρακτικές περιστάσεις, στα πλαίσια μηχανικής μάθησης.

ΘΕΜΑΤΙΚΗ ΠΕΡΙΟΧΗ: Μοντέλο Ροής Δεδομένων

ΛΕΞΕΙΣ ΚΛΕΙΔΙΑ: dataflow, tensorflow, αναδρομικές συναρτήσεις, αυτόματη παραγώγιση, στατικοί dataflow γράφοι

ACKNOWLEDGEMENTS

I would like to express my gratitude towards my advisors, Angelos Charalambidis and Calliope Kostopoulou for giving me the opportunity to continue their work on this challenging, yet fascinating topic.

Additionally, I would like to thank my academic advisor, Panos Rondogiannis, for inspiring me to continue the pursuit of knowledge, and for his consistent support throughout my academic journey.

CONTENTS

1. Introduction **11**

1.1 Background . . . 11

1.2 Motivation . . . 12

1.2.1 Improved Expressiveness . . . 12

1.2.2 Improved Performance . . . 12

1.3 Structure of Thesis . . . 12

2. Computations in TensorFlow **14**

2.1 Staged Execution . . . 14

2.1.1 Graph Execution Model . . . 14

2.1.2 Control Flow Operators . . . 15

2.1.3 Tagging & Context Switching Operators . . . 16

2.1.4 Functions . . . 17

2.1.5 Automatic Differentiation . . . 18

2.2 Tensorflow Eager . . . 19

2.2.1 Basic Features . . . 19

2.2.2 Functionalized Control Flow . . . 20

2.2.3 Gradient Tape . . . 20

3. Recursion in TensorFlow **21**

3.1 Tagging in Recursion . . . 21

3.2 Inline Placement . . . 22

3.3 Transforming the Fibonacci Function . . . 22

3.4 Functions with Multiple Arguments/Outputs . . . 24

4. Automatic Differentiation of Recursive Functions **25**

4.1 Intuition . . . 25

4.2 Algorithm . . . 25

4.3 Transforming the Exponent Function . . . 26

5. Implementation **30**

5.1 Forward Function Declaration . . . 30

5.2 Optimizer function_transformation . . . 31

5.3 Gradient Function Inference . . . 31

6. Evaluation **32**

6.1 Machine Learning Task . . . 32

6.2 Recursion Depth Test 32

7. Future Work & Conclusions **35**

7.1 Graph Load Balancing 35

7.2 Demand Driven Model 35

7.3 Adapting to Other Frameworks 35

7.4 Conclusions 35

ABBREVIATIONS - ACRONYMS **37**

REFERENCES **38**

LIST OF FIGURES

2.1 The Pythagorean Equation $\sqrt{a^2 + b^2}$ expressed in TensorFlow code 14
2.2 Graph of the Pythagorean Equation 15
2.3 The Switch Operator 15
2.4 The Merge Operator 16
2.5 A generic loop in graph mode 16
2.6 Body of function $Pyth(a, b) = \sqrt{a^2 + b^2}$ 17
2.7 Graph of Computation $Pyth(3, 4) - 1$ 17
2.8 After inlining function $Pyth$ 18
2.9 A generic function graph 18
2.10 Extended Gradient Graph 19

3.1 Tag changes between Call and Return nodes 22
3.2 Main Graph after inlining Fib 23
3.3 Transformed Graph 23
3.4 A function with 3 arguments and 4 outputs 24

4.1 A pair of function/gradient calls 26
4.2 After transforming 26
4.3 Segment of Main Graph at each iteration 27
4.4 Exp's Subgraph 27
4.5 Exp's Extended Subgraph (simplified) 27
4.6 Transformed Graph 28

6.1 Minimizing the Exponent Function - 100 iterations 33
6.2 Python Test Code: Static Unrolling 33
6.3 Python Test Code: Recursion 34

1. INTRODUCTION

1.1 Background

The majority of traditional, imperative programming languages, implement the notion of a function, under some form. Functions introduce a highly convenient abstraction layer, as they allow programmers to define local computations that can be composed and reused in a black-box manner. Even though this feature is elegant from a programmer's perspective, its implementation is not as simple. In order to localize each function call, most high level languages utilize a *call stack*: During runtime, whenever a function call is encountered, a new *stack frame* is pushed onto the call stack. Each of these frames introduces a unique *context* of local variables, so that they can be distinguished between previous and subsequent calls. Conversely, when a function call exits, its frame is removed from the top of the stack, and the next topmost frame corresponds to the context from which the call was made. An important benefit of this classical implementation is that it allows for function definitions to be recursive, which is particularly useful in situations where some computation can be expressed as a combination of subcomputations, that have similar structure.

In contrast to imperative languages, modern machine learning frameworks, such as TensorFlow, operate in a completely different way. Because of the fact that performance is critical in machine learning applications, such frameworks follow the *dataflow* programming paradigm [2]. In dataflow, each computation is represented as a directed graph, where nodes represent primitive operations and edges indicate the flow of data between operations. Each individual node/operation will execute the moment it has received data from all of its in-edges. Naturally, the output of the operation will be forwarded along the out-edges, onto the next nodes, until a certain 'sink' node is reached, indicating that the computation has finished. The advantage of this model is that it enables operations to be executed out of order and in parallel, which can greatly reduce the total execution time.

In the context of graph execution, functions can be thought of as independent subgraphs that can be appropriately connected with each other, in order to form a larger computation. In order to support user-defined functions, an important decision to be made is whether the invocation of functions will be *dynamic* or *static* [9]. The dynamic approach suggests that whenever a node that represents a function call is about to execute, it gets replaced with the actual function subgraph, which effectively causes the graph to expand at runtime. The static approach, however, proposes that each function body subgraph shall exist only once, but may execute several times, under different contexts. In the latter case, the independence of contexts is achieved by wrapping the flowing data with special tags, that specify the context in each case.

One crucial feature involving functions in machine learning, is *Automatic Differentiation* [8], which is a mechanism for calculating the derivatives of functions. The process of training any kind of model, essentially, comes down to minimizing a mathematical function with respect to its input variables. As a result, the calculation of derivatives is necessary, as it determines how each variable needs to be changed in order for the function to be minimized. This mechanism works by creating a 'backward' graph, which composes the partial gradients, using the chain rule of differentiation [7]. For each operation in the original graph, a corresponding gradient operation is added on the backward graph, which takes as input not only the partial gradients, computed along the backward path, but also, optionally, the inputs and outputs of the original operation.

The subject of this thesis, was to extend TensorFlow [1] with an algorithm for automatic differentiation which is applicable in static dataflow graphs and works for recursive function definitions. This work is heavily based upon previous work, made by my supervisor, Calliope Kostopoulou, as part of her undergraduate thesis [5].

1.2 Motivation

1.2.1 Improved Expressiveness

After the release of TF 2.0, the framework is promoting the function as the main abstraction at the user level, because of its intuitive nature. Users are no longer required to explicitly define computational graphs. Instead, they may define functions, using a special decorator that traces the execution of a Python function and automatically creates an appropriate dataflow graph. In spite of that transition, however, recursive function definitions are still not supported by TensorFlow. While, in older versions, recursion was supported through dynamic graph expansion at runtime, as of version 2.16, the protocol is to trace function bodies repeatedly, until no other function call remains. Under this policy, recursive functions cause the tracing procedures to crash, by falling into endless loops. In general, recursion is regarded as a highly powerful programming technique and several deep learning architectures utilize it, especially in cases of hierarchically structured data [6]. For that reason, it could be valuable to provide a mechanism that can handle recursion and automatic differentiation efficiently.

1.2.2 Improved Performance

With the introduction of TensorFlow Eager as the default mode of operation, the framework gradually shifts away from (staged) graph execution, and adopts imperative features. While it is true that staged execution is more difficult to program and to debug, it has significant benefits in terms of parallelization. Dataflow's declarative nature allows for the main graph to be partitioned into multiple subgraphs that can be deployed in distributed, heterogenous systems and execute in parallel without any conflicts. Therefore, we believe that an implementation of recursive functions in the staged context can leverage these benefits and outperform a potential implementation in imperative mode, where there is significant overhead in going back and forth into the Python interpreter. Additionally, our approach on recursion using static dataflow graphs guarantees that the graph structure will be known prior to the execution, which allows for better partitioning of the graph, when compared to the dynamic approach. Furthermore, our approach on automatic differentiation provides an efficient way to calculate gradients of user defined functions, without having to recompute function values, during the gradient evaluation. Note that this applies to all user-defined functions and not just recursive ones.

1.3 Structure of Thesis

The rest of the thesis is organized as follows:

Chapter 2 describes the framework's modes of operation in detail. Both staged and imperative modes are examined, as well as the mechanisms for automatic differentiation in

both cases.

Chapter 3 describes our approach in transforming graphs in order to support static recursion.

Chapter 4 describes the main algorithm for transforming gradient calls for recursive functions, in order to implement automatic differentiation.

Chapter 5 contains technical details related to the implementation of our approach. These details mostly concern TensorFlow's core and infrastructure.

Chapter 6 presents several performance results regarding our implementation

Chapter 7 proposes ideas that could possibly extend our current work and provides some final conclusions.

2. COMPUTATIONS IN TENSORFLOW

TensorFlow is an open-source framework, designed for implementing large-scale machine learning applications. It is highly flexible and it can be used to express a wide variety of training and inference algorithms for deep learning models. It was first released by Google in 2015 and its modes of operation have changed significantly over the course of its lifespan.

2.1 Staged Execution

Originally, TensorFlow [1] was based completely upon the dataflow computational model. It provided a high level API which allowed users to define their computations as dataflow graphs and make several configurations, such as graph optimizations and partitioning on multiple devices. This purely declarative mode of operation is commonly referred to as *staged execution* mode and it is still supported today.

2.1.1 Graph Execution Model

In staged execution, each computation is expressed as a directed graph. Each node may have zero or more inputs and zero or more outputs and it represents the instantiation of an *operation*. Each possible operation has a name and represents an abstract computation, such as 'matrix addition' or 'matrix multiplication'. The values that flow along the edges of the graph are *tensors*, meaning, arrays of arbitrary dimensionality.

```
a = tf.constant(3.0)
b = tf.constant(4.0)

a_squared = tf.multiply(a,a)
b_squared = tf.multiply(b,b)

sum_of_squares = tf.add(a_squared, b_squared)

result = tf.sqrt(sum_of_mults)
```

Figure 2.1: The Pythagorean Equation $\sqrt{a^2 + b^2}$ expressed in TensorFlow code

There also exists a special type of edges called *control dependencies*. Such edges do not carry any data, but simply impose a strict order of execution between the connected operations. In graphs, control dependencies are represented using dashed arrows.

After a graph is constructed, it undergoes several optimization procedures and it eventually gets partitioned so that it can be deployed on multiple, possibly heterogenous devices, including CPUs, GPUs and TPUs. Each operation's implementation on a particular type of device is called a *kernel* and it is the actual algorithm that computes the operation, by leveraging potential device-specific properties. In distributed runtime, there is a main process called *master process* that handles the graph's preprocessing, prior to execution. It applies a certain node-placement algorithm and, eventually, it sends the generated subgraphs to several *worker* processes, that may run on different devices. Each worker process is responsible for executing nodes/operations that belong in their own subgraph.

Nodes that have no pending input dependencies are placed in a *ready queue*, meaning that they are available for execution. Every other node is placed in a *waiting queue*. Whenever a node receives a new input, it is reevaluated for placement in the ready queue. The ready queue itself is processed in some order, depending on the underlying device capabilities.

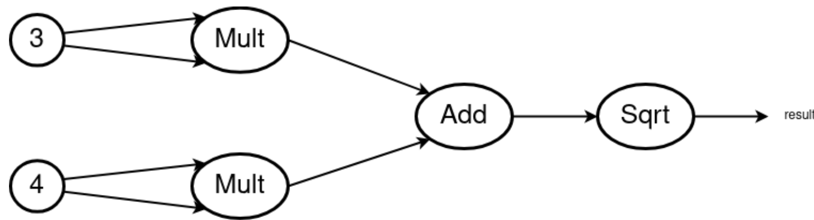


Figure 2.2: Graph of the Pythagorean Equation

2.1.2 Control Flow Operators

So far, operators have been described as simple computational abstractions that perform tensor operations. However, in order to provide support for complex dynamic control flow constructs [18], such as conditional statements, several operators with special semantics [17] have been introduced:

- **Switch:** The Switch operator is used to branch the execution into different subgraphs, according to the truth value of a certain boolean input. More specifically, it expects two inputs: The first input contains data that needs to be propagated to the appropriate subgraph and the second input expects a predicate that will determine the branch to be taken. Naturally, the Switch operator has two outputs, one corresponding to a predicate value of 'True', and another of 'False'.

  To be more precise, the taken branch of the switch operator will forward actual tensor data, while the other, non taken branch will forward 'dummy' data that will be propagated instantly across the non-taken subgraph. This mechanism is generally known as *deadness propagation* and its primary purpose is to signal the operations of the non-taken branch that they should not expect any incoming inputs.

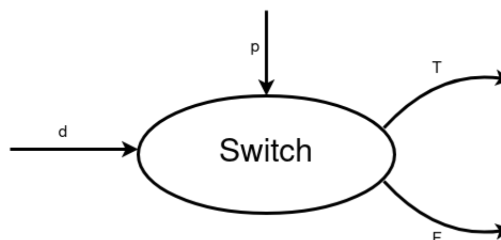


Figure 2.3: The Switch Operator

- **Merge:** The Merge operator usually goes in pair with the Switch operator and its purpose is to reconnect the two branches back into a single branch. In general, the Merge operator may expect one or more inputs, out of which only one will propagate actual data, while the others will forward a dead signal.

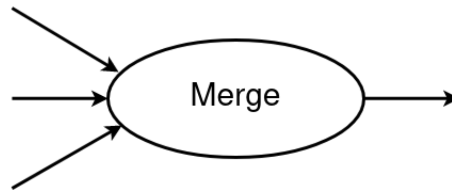


Figure 2.4: The Merge Operator

2.1.3 Tagging & Context Switching Operators

TensorFlow also supports the definition of iterative constructs (i.e. while loops), at a graph level. This poses a significant challenge, regarding the handling of flowing data: by allowing certain parts of the graph to execute multiple times, possibly simultaneously, it is necessary to enforce a mechanism that differentiates data between iterations. Otherwise, a node in the loop body subgraph could yield incorrect results, by executing its kernel on data that belong to different iterations.

TensorFlow addresses this challenge by introducing data *frames* (otherwise known as *tags* [9]). Frames can be thought of as special metadata that uniquely identify the context under which the accompanied data correspond to. Under this regime, in order to enforce consistency, each node is expected to perform operations on data that belong in the same context. The actual value of a frame is just an integer that represents the loop iteration number, as it suffices in order to specify the context. Also, frames can be nested, in order to support nested loops[1].

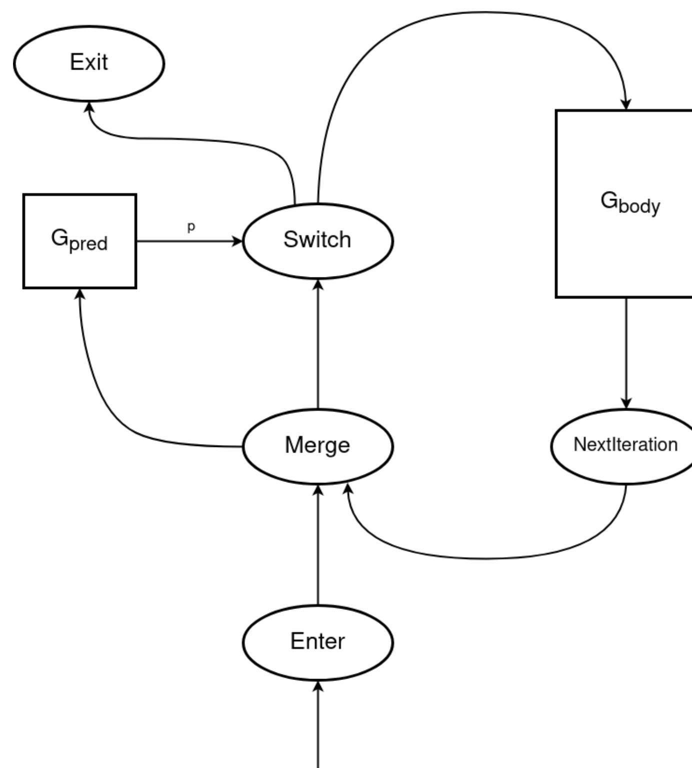


Figure 2.5: A generic loop in graph mode

[1] Two frames that are nested into each other are said to have a 'parent-child' relationship.

The tagging semantics described above are enforced through the following context switching operators [17]:

- **Enter:** The Enter operator is used in iterative constructs in order to create new contexts of operation. Essentially, it wraps received data inside of new frames and it propagates the framed data. The newly added frame is the current frame, while the previous frame, that corresponds to the previous loop context, is the parent frame.
- **NextIteration:** Similarly, the NextIteration operator also forwards its input into a new, uniquely identifiable context. Its purpose is to update the value of the current frame, by effectively incrementing the loop iteration number.
- **Exit:** Finally, the Exit operator is responsible for revoking the operations of Enter: It pops the current frame and forwards the data under the previous (parent) context.

2.1.4 Functions

In staged execution mode, a defined function has its own *graph definition*[2], which is, essentially, the function body. The main difference between a function's graph and the main graph of execution is that the former contains special *Argument* and *Return Value* nodes which are placeholders for the function's input/output. In addition, the function's name is registered as an operation, which means that an invocation of the function from the main graph will be initially represented by a single node. Consequently, it is not allowed to define a function with the name of an existing operation, such as 'Merge', for example.

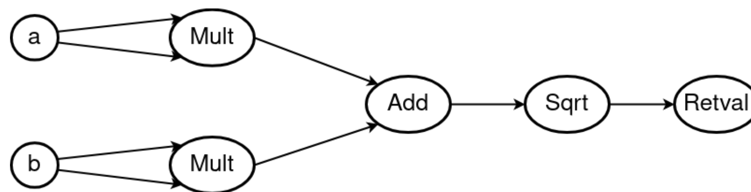


Figure 2.6: Body of function $Pyth(a, b) = \sqrt{a^2 + b^2}$

Before the execution of the graph, a certain graph optimizer, known as *function optimizer* inlines each function call operator until all of them are replaced. Of course, this means that a recursive function would lead to infinite inlining. Although this optimizer can be disabled by the user, the existence of function call operators during graph execution will lead to the expansion of the graph at runtime (dynamic invocation) and this is considered deprecated behaviour.

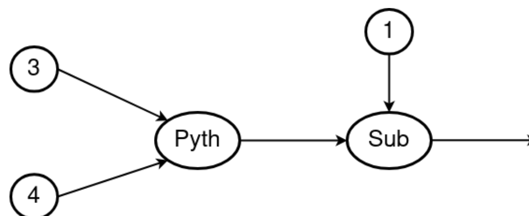


Figure 2.7: Graph of Computation $Pyth(3, 4) - 1$

[2] The difference between a graph and a graph definition is subtle: A graph definition can be thought of as a recipe to build graphs. Surprisingly, though, converting a graph back into a graph definition happens very often in the TensorFlow core

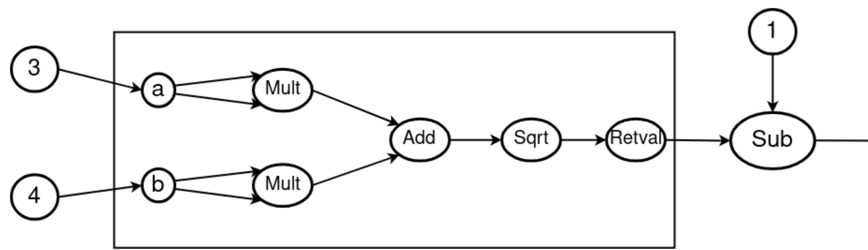


Figure 2.8: After inlining function $Pyth$

2.1.5 Automatic Differentiation

Many optimization algorithms in machine learning, such as stochastic gradient descent, rely on computing the gradient of a cost function with respect to its inputs. For that reason, TensorFlow provides built-in support for automatic gradient computation. According to the original TensorFlow paper [1], this mechanism works by extending the TensorFlow graph as follows:

- When TensorFlow needs to compute the gradient of a tensor C with respect to some tensor I on which C depends, it first detects the path between the node whose output is C and the node whose input is I.
- It then traverses this path in a backward manner, adding nodes to the graph that compose partial derivatives along the backward path, according to the chain rule.
- Each newly added node computes the ”gradient function” for the corresponding operation in the forward path.

An important detail to consider is that this algorithm assumes that, for each operation, the corresponding gradient operation is also registered internally. While the gradients of primitive operators are registered by default, in the case of function operators, the user has to manually register a gradient function. Additionally, each operation is assumed to be a black-box abstraction that is completely independent from its gradient counterpart. This is somewhat problematic for user-defined functions, as it means that the function subgraph may not forward intermediate results to the gradient subgraph. Consequently, this necessitates the recomputation of the function output, from within the gradient function subgraph.

TensorFlow also provides an API [15] that derives the gradient graph of a function, by *extending* its original graph with backward paths that compose gradients. This API could be used in order to automatically define the gradient of a function and then register it.

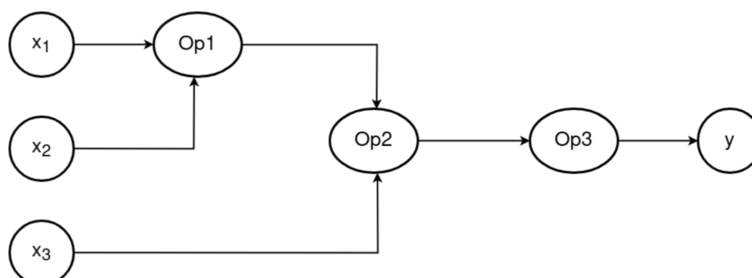


Figure 2.9: A generic function graph

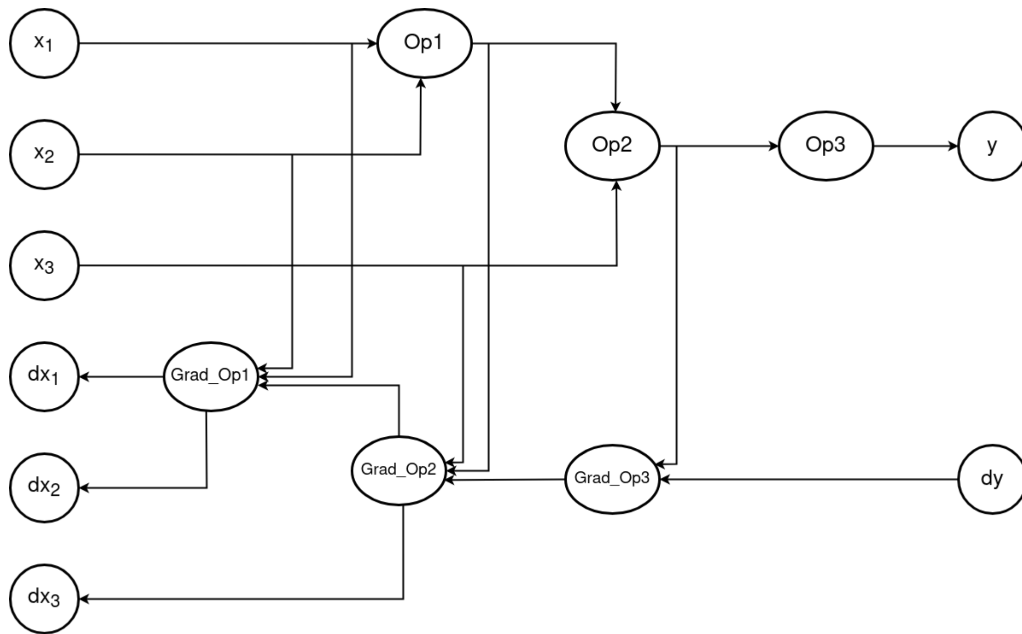


Figure 2.10: Extended Gradient Graph

In Figure 2.10, the function's original graph has been augmented with a backward path that composes gradients, using the provided API. Naturally, the original graph that calculates the value of the function is referred to as the *forward* path, while the subgraph that calculates the gradients is called the *backward* path. The input node labeled as dy represents the initial gradient weight, that is given as input from previous layers. If there are no previous layers (i.e. y is the terminal output), then a value of 1 is given to dy [15]. Each node in the forward path has a corresponding node in the backward path that calculates it gradient. It is evident that the Extended Gradient Graph calculates the value of the function internally, through the forward path.

It is worth noting how the number of inputs and outputs changes when the graph is extended: if a function has n inputs and m outputs, after its graph is extended, it will possess n + m inputs, out of which n of them will be the original arguments and m of them (one for each output) will correspond to gradient inputs from previous layers. Similarly, the outputs will be n + m, out of which m will be the original outputs and the other n will be the gradient outputs for each argument.

2.2 Tensorflow Eager

In spite of its significant benefits regarding parallelization, the staged execution mode proved to be difficult to use and counter-intuitive to programmers. For that reason, TensorFlow Eager [3] was introduced. TensorFlow Eager introduces imperative execution, which is the main mode of operation in other popular machine learning frameworks, such as PyTorch [10] or Chainer [16]

2.2.1 Basic Features

TensorFlow Eager is characterized as a multi-stage domain specific language, as it features both *imperative execution* and staged execution, although imperative execution is

enabled by default. The main difference between the two modes lies in the way that the kernels of operations are dispatched. In imperative mode, library functions construct operations and execute their kernels immediately, whereas in staged mode, the functions would return a symbolic representation of values to be computed later, instead of actual values.

Additionally, in order to make it easier to define dataflow graphs, a special decorator called `function` is introduced. This decorator is, essentially, a Just In Time compiler that traces the execution of Python functions and records all tensor operations into a dataflow graph. It is worth noting that this tracing process fully unrolls Python control flow constructs such as if-else statements and while loops, potentially creating large graphs. In order to introduce control flow at the graph level, users have to resort to functions such as `tf.cond`, which are purely symbolic. This behaviour is similar for function calls as well: all calls are expected to fully unroll during the tracing process, and, as a result, recursive functions cause the tracing process to unfold indefinitely.

2.2.2 Functionalized Control Flow

Upon the release of TensorFlow version 2.0, the framework changed the way it handles control flow constructs in staged execution. Instead of building statements using the original control flow operators (Switch, Merge, Enter, etc), it uses functional statements. For if-else statements, this means that both branches of the statement are treated as functions that should be called according to the predicate input. The behaviour for while statements is similar. In spite of this change, however, the framework still supports the original control flow operators and users may optionally disable the use of functional statements.

2.2.3 Gradient Tape

TensorFlow Eager supports automatic differentiation in imperative mode, through an abstraction called a *tape*. Each defined tape may *watch* a particular value by recording operations that take this value as input. Upon the first forward pass, every intermediate value that is related to the watched value is registered within the tape for later reference, during the backward pass. If a function, defined for staged execution (though the `function` decorator) is found, then this poses no problem, as long as the corresponding gradient function is also registered. Therefore, the tape is compatible in cases where imperative code and staged code are mixed up. Finally, the process of calculating a gradient using a tape is also a differentiable operation, meaning that tapes can be nested in order to compute higher order derivatives:

```
x = tf.constant(3.0)
with tf.GradientTape() as t1:
    with tf.GradientTape() as t2:
        t1.watch(x)
        t2.watch(x)
        y = x * x
    dy_dx = t2.gradient(y, x) # 6.0
d2y_dx2 = t1.gradient(dy_dx, x) # 2.0
```

3. RECURSION IN TENSORFLOW

This chapter presents our approach on implementing recursive function definitions in TensorFlow graphs. This approach is based on transforming the graph prior to execution in a way such that each function subgraph may run several times, under the different contexts of recursive calls. Under these circumstances, each output is redirected to the appropriate call site.

3.1 Tagging in Recursion

In order to allow for multiple executions of certain parts of a graph, possibly simultaneously, it is necessary to introduce data *tags* [9], similar to the ones that are used in iterative graph constructs. As mentioned in Chapter 2, the three existing context switching operators (Enter, NextIteration, Exit) are responsible for updating frames of incoming data, so that they are distinguishable between different iterations. Furthermore, it is essential to enforce that every other operator in the graph may only execute its kernel using data of the same context/tag.

The tagging mechanism that is described below is based upon an algorithm for transforming first-order functional programs into intentional ones, which are easily deployable in dataflow architectures. It was originally proposed by A. Yaghi as a subject for his PhD dissertation in 1984 and a formal description of it can be found in [12]

Whereas in iteration, the tags are expressed using a single integer, which represents the iteration number, in recursion, the tags are expressed as integer lists. In order to handle the management of these new tags, we introduce two special control flow operators:

- **Call:** The Call operator is placed as a guard to all locations where a function is called. It receives as input the function's argument, as supplied from the call site and it forwards it to the function body subgraph, while updating the tag. Each Call node in the graph also maintains two attributes: a reference to the function that is called and an identifier that is unique between all Call nodes of the graph. At runtime, tags of incoming data are updated by prepending the Call node's unique identifier into the integer list. This ensures that data derived by different, overlapping calls to the same function won't get mixed up during the execution.
- **Return:** The Return operator is placed in the exit points of a function. It collects the function's output value and it forwards it back to the 'caller graph', all while updating the tags appropriately. The Return operator maintains as an attribute the unique ID of the corresponding Call node, so that the Call and the Return nodes are matched with each other. The tags of incoming data are updated by removing the first element of the integer list, provided that it matches the internal ID of the Return node. If these two values don't match, then the incoming data is dropped completely.

Notice how the integer list, behaves similarly to the call stack of imperative languages: The size of the list increases as the function call depth increases and, if we traverse the list from start to end, we iterate from the most recent (inner) call site to the outmost one. This means that a given list corresponds to a specific sequence of call sites and, therefore, identifies a unique context.

Figure 3.1: Tag changes between Call and Return nodes

3.2 Inline Placement

Now that we have a mechanism for creating unique contexts, we can describe the algorithm for transforming the graph. First of all, it is necessary to identify all of the functions that are called through the main graph, either directly, or indirectly, through other functions. Since these functions will need to be computed statically, it is required to insert a copy of their subgraphs within the main graph. Once the body of every function has been placed, then all of the function call operators within the main graph can be replaced with pairs of Call/Return nodes that have distinct (incremental) identifier values, as inscribed by the Call operator's semantics.

All that remains is to make appropriate connections so that:

- The Call nodes forward their input to the respective function bodies.
- The function bodies forward their output to the Return operators.

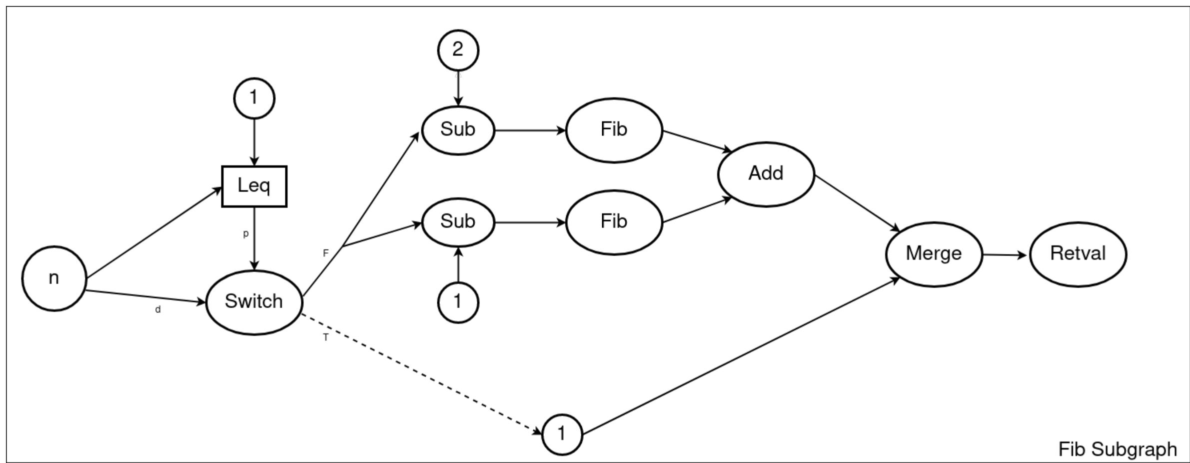
Every function body that has been newly placed in the main graph contains a special argument-placeholder node that represents the function's inputs. In order to connect multiple Call nodes with a single function body, it suffices to convert the argument placeholders into Merge nodes. The semantics of the Merge operator are convenient in this case, as they provide precisely the desired functionality: When a single Call node is executed, the function argument is forwarded to the Merge node, which will execute immediately, effectively forwarding the argument to the function body.

The function bodies also contain special placeholder nodes that represent the output values. These nodes also need to be transformed in order to forward their outputs to multiple Return nodes. According to Return node semantics, each Return node receives outputs from all call sites of that function, but only keeps the one that matches its internal identifier. Therefore, the output-placeholder nodes need to be transformed into *identity nodes*, meaning nodes that do not perform any actual computation, but simply forward their input to multiple nodes. By transforming such nodes into identity nodes and connecting their outputs to the Return nodes, the transformation is complete.

3.3 Transforming the Fibonacci Function

To demonstrate this transformation in practice, consider the computation `Fib(4) + Fib(7)`, where `Fib` is the Fibonacci sequence, defined recursively as:

```
Fib(n) = if (n <= 1) then 1 else Fib(n-1) + Fib(n-2)
```



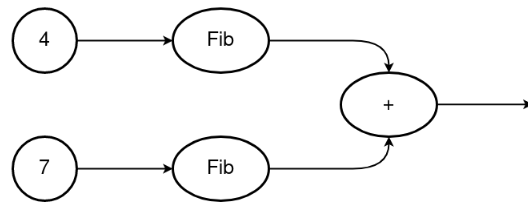


Figure 3.2: Main Graph after inlining Fib

As shown in Figure 3.2, the subgraph of the Fib function is defined using control flow operators. The nodes labeled as ’n’ and ’Retval’ are placeholder nodes that correspond to input and output respectively, while nodes labeled as ’Fib’ are function call operators that refer to the function itself.

In the transformed graph, as shown in Figure 3.3, all four ’Fib’ function calls have been converted to pairs of Call/Return nodes. All Call nodes forward their inputs to the newly created Merge node, so that for each call of Fib, the input is fed into the function subgraph. Additionally, the Identity node at the end of the function body forwards its input to all Return nodes, so that in each case, the data will return to the call site of origin, according to the identifier.

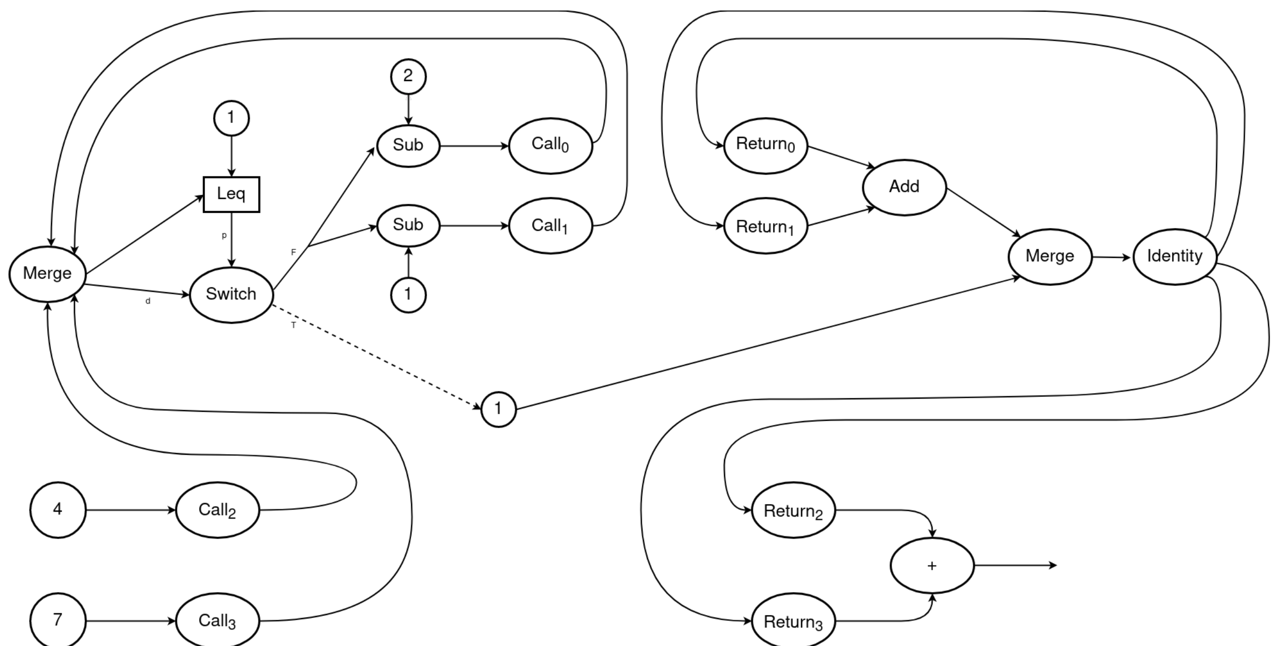


Figure 3.3: Transformed Graph

3.4 Functions with Multiple Arguments/Outputs

When handling functions that take more than one input and/or produce more than one output, there is a small technicality that needs to be addressed, during the transformation process:

Consider a function that takes x inputs and produces y outputs. In the main graph, every call operator of this function will take x inputs and will forward y distinct outputs. At the same time, the function subgraph will have x argument placeholders and y output placeholders. This means that each of the x inputs needs to be redirected into the appropriate argument placeholder node (to be converted into a Merge node) and each of the y output placeholders (to be converted into Identity nodes) needs to forward its output back to the call site.

The solution is to replace every call operator with x Call nodes and y Return nodes. As such, each function argument will be forwarded, through a Call operator, into a Merge node that is dedicated for that argument, throughout all call sites of that function. Similarly, each distinct function output will be connected with a dedicated Identity node. As for the internal IDs, all of the x Call and y Return nodes must share the same internal ID, in order to handle context management correctly. An interesting benefit of this approach is that it enables *partial function invocation*, by supplying arguments asynchronously. Furthermore, if a certain output depends only on a small subset of the inputs, then this output will be returned even if the rest of the inputs are not supplied.

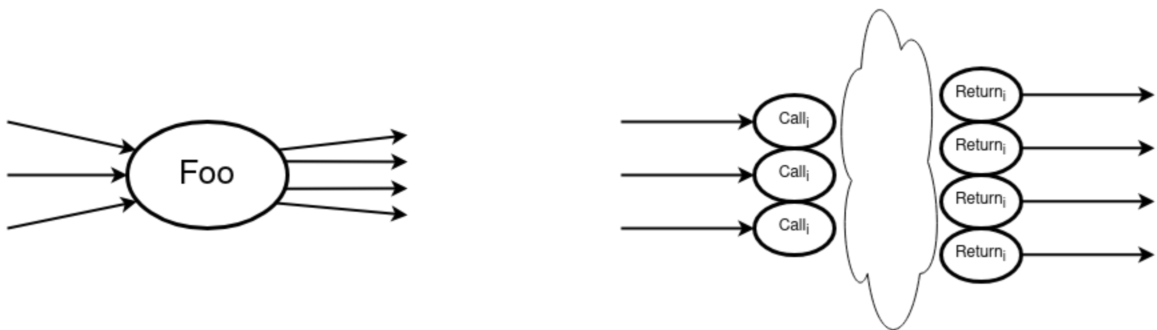


Figure 3.4: A function with 3 arguments and 4 outputs

4. AUTOMATIC DIFFERENTIATION OF RECURSIVE FUNCTIONS

Our approach on automatic differentiation demonstrates that in order to compute outputs and gradients of possibly recursive functions, it suffices to include and reuse a single instance of the Extended Gradient Graph within the main graph. A collateral benefit of this proposed fusion is that intermediate results produced during the forward phase can be shared with the backward phase, effectively eliminating the need to reevaluate the forward phase during gradient calculation.

4.1 Intuition

By examining how a function graph is extended into a gradient graph, we can make two important observations:

- If a function is recursive, then its extended graph will also be recursive, as the backward path will contain calls to the gradient function itself.
- If we disregard the original function and treat the extended graph as a new, atomic function, then all calls to the original function can be represented as *partial calls* to the new function, where only the original input arguments are provided.

Based on this, it is possible to exploit the static transformation, described in the previous section, in order to redirect every function and gradient call into a single instance of the Extended Graph. Furthermore, in cases where the gradient is evaluated right after the function value, these two calls may be combined, in order to avoid recomputing values.

4.2 Algorithm

The mechanism for transforming gradient calls is, essentially, an extension of the static graph transformation, described in Section 3.1.

Initially, all functions that will be called during execution are identified and are placed within the graph. If the gradient of a certain function is also identified, then only the Extended Graph of the function is placed, instead of the function subgraph.

For each function, the function call operators and the gradient call operators are organized into pairs, so that both the function value and the gradient are computed in a single pass of the extended graph. If certain function calls cannot be matched with another gradient call, then this is not a problem, as they can be computed by *partially invoking* the extended graph, by only supplying the function's original arguments. That is because the forward path of the Extended Graph can be fully executed and produce outputs, even if the rest of the arguments, that concern the backward pass, are not supplied (Section 3.4).

Afterwards, each pair of function/gradient calls is converted using Call and Return operators. Specifically, each argument of the function call is redirected into the corresponding argument-placeholder of the extended graph and each gradient input of the gradient call is redirected into the appropriate placeholders of the extended graph[1].

[1] Note that the placeholders will eventually be converted into Merge/Identity nodes, as the graph transformation dictates

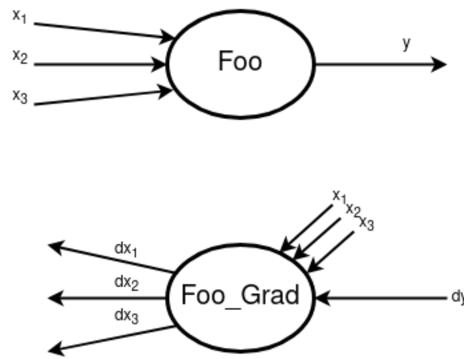


Figure 4.1: A pair of function/gradient calls

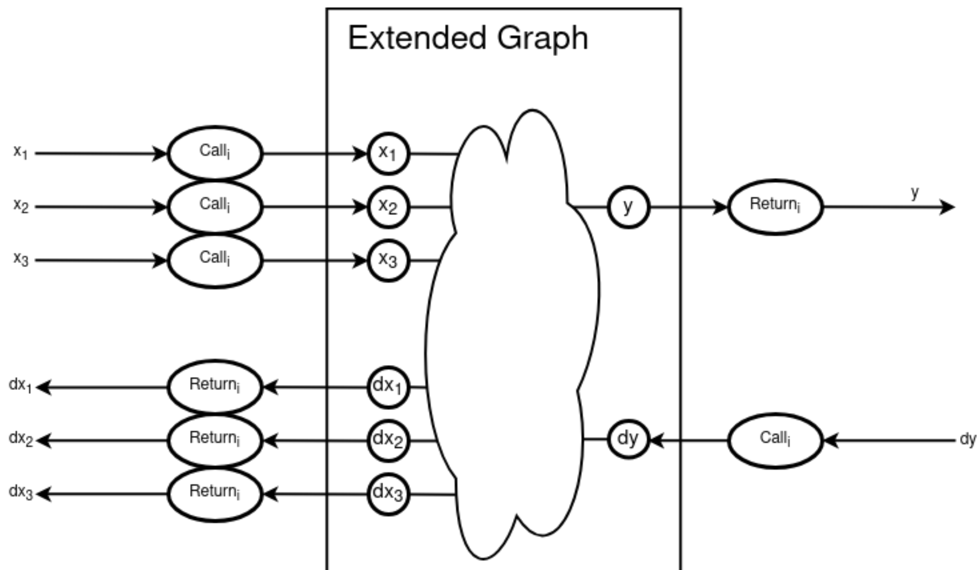


Figure 4.2: After transforming

An important detail to note is that when converting gradient calls into Call/Return nodes, the original arguments of the function are intentionally neglected, as they are redundant. Notice how in the example displayed in Figures 4.1 and 4.2, the inputs x_1, x_2, x_3 of Foo_Grad are not converted into Call nodes, since their values will be transported accordingly through the Extended Graph.

4.3 Transforming the Exponent Function

In order to better comprehend this procedure, we will closely examine the transformation of the function x^n, defined recursively as follows:

```
Exp(x,n) = if(n == 0) then 1 else x * Exp(x,n-1)
```

In most use cases of Automatic Differentiation, especially in algorithms such as Stochastic Gradient Descent, the program maintains a set of trainable parameters that are iteratively updated, until some stopping condition is satisfied. In each of these iterations, both the objective function and its gradient are evaluated upon the same set of trainable parameters. Therefore, in our example, if we assume that the objective function is $Exp(x, n_0)$, for

some fixed n_0, then, for each iteration, the computational graph will include the following set of nodes:

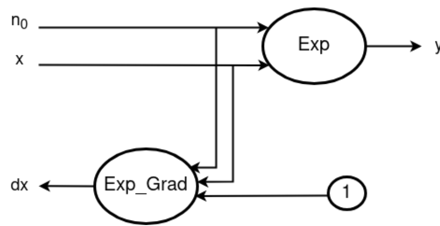


Figure 4.3: Segment of Main Graph at each iteration

In the above figure it is worth mentioning that the gradient of Exp returns output only for the x parameter, since n cannot be differentiated upon, as it is a discrete variable.

According to the definition of the Exp function, its computational graph uses a Switch/Merge conditional construct and has the following (recursive) structure:

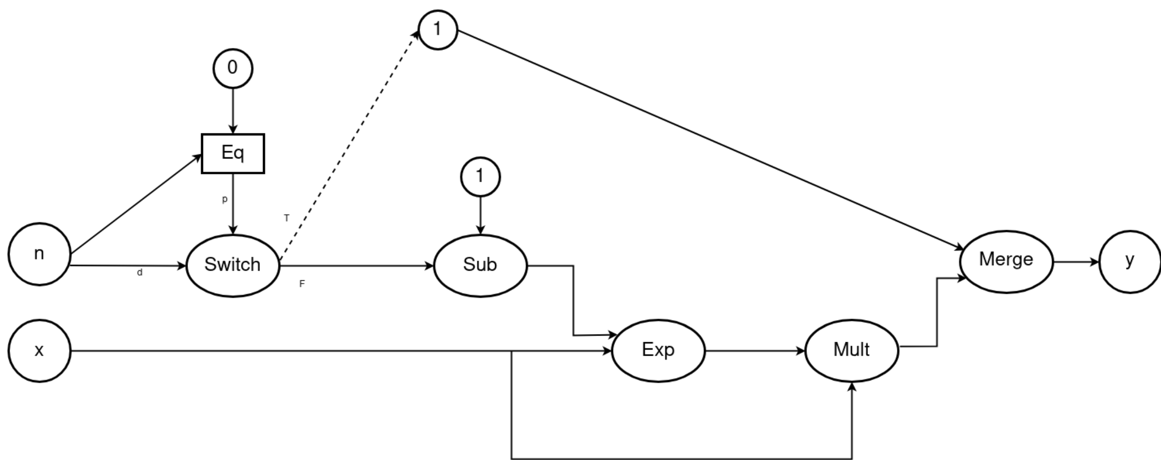


Figure 4.4: Exp's Subgraph

By applying the algorithm for extending the subgraph of Exp with backpropagation paths, we get the following graph:

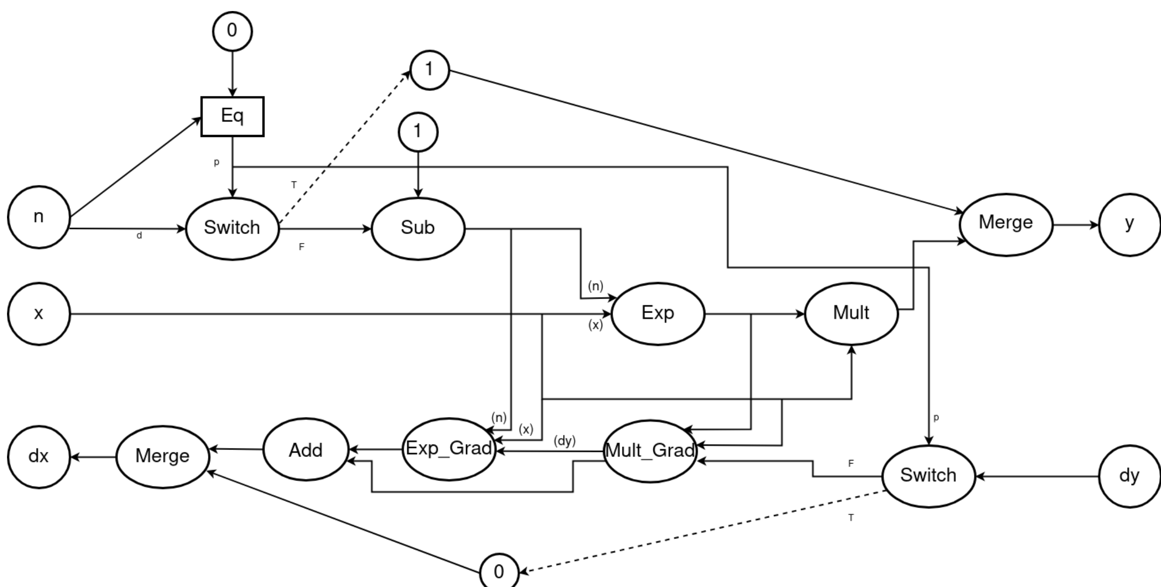


Figure 4.5: Exp's Extended Subgraph (simplified)

In order to get a better understanding of Exp's Extended Subgraph, we can make the following observations:

- The graph has been augmented with a backpropagation path that starts at node dy and ends at node dx. This backpropagation path contains a Switch-Merge conditional construct that is identical to the one that is present in the forward subgraph. Its purpose is to direct the gradient calculation according to the predicate value of `n == 0`, hence the input from the Eq operator.
- Each branch of the newly inserted Switch operator represents the inverse path of the branch taken during the forward pass. Notice how the two Switch operators are in sync with each other, meaning that if `n == 0`, then the 'True' branches get executed, effectively returning y = 1 and dx = 0. Conversely, if `n != 0`, then the 'False' branches are taken.
- The Extended Subgraph of Exp is recursive, since the call to Exp_Grad refers to the subgraph itself.
- Both calls to Exp and Exp_Grad receive the same input arguments (n) and (x), just like in the Main Graph.

The next step of the transformation procedure is to emplace the Extended Graph within the Main Graph, replace all Exp and Exp_Grad nodes with Call/Return operators and connect them appropriately with input/output placeholders. Finally, the last step is to convert input placeholders into Merge nodes and output placeholders into Identity nodes.

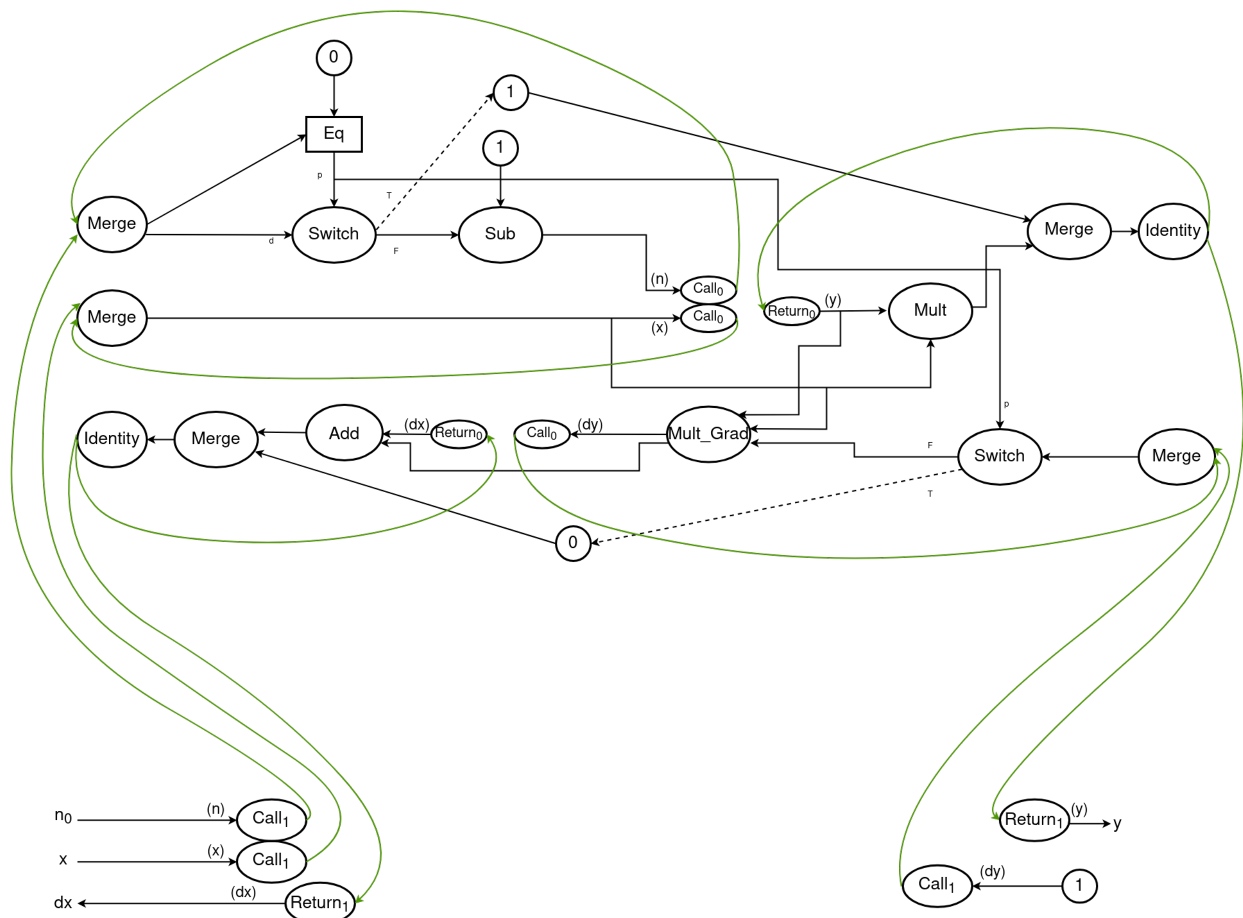


Figure 4.6: Transformed Graph

In Figure 4.6, the final version of the Main Graph is displayed. The edges colored in green are newly added edges that connect Call/Return nodes with Input/Output placeholders respectively. As explained in the previous section, the arguments (x) and (n) of Exp_Grad are dropped, and only the (dy) argument is supplied in the place of Exp_Grad.

Notice how the Extended Graph can be considered as a function that takes 3 input arguments: (n), (x), (dy) and returns two: (dx) and (y). Also, there are exactly two call sites of this recursive function, throughout the entire Main Graph, as indicated by the internal IDs of Call/Return nodes. The ID of 0 corresponds to the recursive call site, while the ID of 1 corresponds to the outer call site.

In order to observe the asynchronous nature of this structure, let's see what happens if we supply $x = 3$ and $n_0 = 1$:

- The call `{x = 3, n = 1, dy = 1}` is made, through the Call_1 nodes of the Main Graph.
- In the Extended Graph, the (x) and (n) data flow towards the corresponding Call_0 nodes.
- A *partial*, recursive function call is made, with arguments `{x = 3, n = 0}`, through the corresponding Call_0 nodes.
- Eventually, the (y) output is returned through Return_0, but since it was a partial call, the input for (dy) is still pending, for the inner call.
- The newly computed (y) output is further processed in order to compute the (y) output of the outer call, which is to be returned in the outer call site, in Return_1
- At the same time, the (y) output of the inner call is also propagated through Mult_Grad, towards Call_0 for (dy).
- Once it is reached, the (dy) input of the previous partial call is no longer pending, and, eventually, (dx) is returned in Return_0.
- Finally, the inner (dx) value is used to compute the (dx) output of the outer call, which is returned in the corresponding Return_1 node.

The big picture is that, through this conversion, instead of having two independent functions, Exp and Exp_Grad, that are actually subgraphs of each other, and perform duplicate computations, we create a more general function that solves both problems asynchronously. At its core, this mechanism is based on the fact that it is possible to make partial, overlapping function calls. The ability to perform partial calls is guaranteed by the independence of Call nodes between arguments (Section 3.4). On the other hand, overlapping calls are guaranteed through the tagging mechanism (Section 3.1).

5. IMPLEMENTATION

This chapter discusses several technical details regarding the implementation of the previous ideas in the TensorFlow core. In order to provide support for recursive functions in TensorFlow, it is required to overcome several issues that concern the Eager mode of execution. The implementation of our work is integrated as an additional feature to the framework, meaning that no changes were performed to the existing code.

The implementation itself is based on TensorFlow 2.16, although several legacy features are reintroduced in order to provide support for recursive function definitions.

Our code can be found on: https://github.com/GeorgeVasilakopoulos/tensorflow

5.1 Forward Function Declaration

When trying to define a recursive function in TensorFlow 2.16, one of the main problems is that the `tf.function` decorator traces the recursive function calls indefinitely, as it is designed to unfold control flow constructs and function calls. For example, the following segment of code causes a maximum recursion error:

```
@tf.function
def recursive_fn(n):
  if n > 0:
    return recursive_fn(n - 1)
  else:
    return 1

with assert_raises(Exception):
  recursive_fn(tf.constant(5))  # maximum recursion error.
```

For that reason, it is necessary to introduce an alternative way of defining recursive functions, so that the tracing process does not fall into an endless loop. The solution came through reintroducing a legacy class named `tf.function.Declare`, which allows users to declare a named function, along with its inputs and then use the Python object as a callable:

```
fac = tf.function.Declare("Fac", [("n", tf.float32)], [("ret", tf.float32)])

@function.Defun(tf.float32, func_name="Fac", out_names=["ret"])
def FacImpl(n):
  return tf.cond(tf.equal(n, 1),
    lambda: tf.constant(1.0),
    lambda: n * fac(n - 1))
```

Additionally, it is required to provide a mechanism that registers a function definition within the graph. Otherwise, the tracer will not recognize the function that is referred by the `tf.function.Declare` object and will return an error. In order to address this, we implemented a C API call named `TF_GraphAddFunctionDef` which can be used in order to register a function that has not yet been defined, as an operation in the graph.

5.2 Optimizer function_transformation

In TensorFlow, after a graph is constructed, several optimization procedures are applied to it, before it gets partitioned and executed. Each individual optimization that is applied on the graph is defined programmatically as a class with appropriate attributes and methods, and it can be manually disabled by the users.

Our approach on transforming function calls is implemented as an additional *Optimizer*, named 'function_transformation'. This optimizer works by placing in-line a single instance of all user-defined functions and transforming each call, according to the mechanism described in the previous sections. Also, because of the fact that our transformation creates graph cycles that are unaccounted for, it is necessary to loosen the conditions of cycles in the algorithm for topological ordering.

In order for the transformation to be applied properly, users also must disable the optimizer named function_optimizer, due to discrepancies with function_transformation. More specifically, function_optimizer performs recurring inline placements to all function calls, in order to enable other inter-procedural optimizations. Therefore, this specific optimizer cannot work along with our own optimizer, as it will remove all function calls from the main graph. Furthermore, function_transformation works with graphs that utilize the original control flow operators, which means that functionalized control flow (Section 2.2.2) needs to be manually disabled.

5.3 Gradient Function Inference

By default, in order to perform gradient computations for custom TensorFlow functions, users are required to manually register the gradients of functions as operations. The standard method to infer the gradient function subgraph is using an API which, as mentioned in Section 2.1.4, extends the original function graph with backpropagation paths.

In order to automate the process of extending the function graph, we introduce a special flag that can be supplied during the function definition. This flag is named `create_grad_func` and, if enabled, then the extended graph will be inferred and registered in place of the original function, so that the gradient call transformation will work as described in Section 4.2

```
@function.Defun(tf.float32, tf.float32, func_name="EXPONENT",
                create_grad_func = True, out_names=["ret"])
def ExpImpl(x, n):
    return tf.cond(tf.equal(n,0),
                lambda: tf.constant(1.0),
                lambda: x*exp(x,n-1))
```

6. EVALUATION

In order to evaluate our implementation, we performed experiments on some basic use cases of recursion and automatic differentiation and compared the performance against existing mechanisms provided by TensorFlow. Because of the fact that TensorFlow does not support data-dependent recursion, we considered use cases where the recursion depth is fixed and known in advance. Therefore, this type of recursion can be emulated by either unrolling the graph prior to execution, or by adapting iterative constructs such as `tf.while_loop`, were data-dependent looping conditions are actually possible. All tests were performed on a TensorFlow CPU-only build.

6.1 Machine Learning Task

In this test, we experimented with a relatively simple model in the family of Recursive Neural Networks, the TreeRNN [13]. More specifically, the task is to perform sentiment classification in data from the Stanford Sentiment Treebank dataset, where every instance is a fully labeled parse tree of a certain sentence, meant to represent the semantic structure of its individual words.

In the TreeRNN model, a set of trainable parameters is reused hierarchically, in each layer of the tree, in a bottom-up manner, in order to yield a prediction at the root of the tree. Since each training instance is an individual tree, the model needs to adapt to the unique structure of each data instance. This is where recursion can show its potential.

We compared our approach (labeled as Recursion) against an implementation that uses iteration, using the `tf.while_loop` operation. Both models were trained for 4 epochs, on a dataset containing 700 examples, and performed inference on a dataset containing 200 examples. The training and inference times listed in Table 6.1 indicate a significant speedup in favour of our recursive approach. This speed up can be attributed to the difference in managing gradient calls [18], hence the larger percentage difference in training. Although, to be fair, it is possible that an overhead is also introduced by 'flattening' recursion into iteration.

Table 6.1: TreeRNN Training & Inference Times in seconds

	Iteration	Recursion	Speedup
Training Time (s)	268.91	179.57	33.22%
Inference Time (s)	8.24	7.41	10.07%

6.2 Recursion Depth Test

In this test, we demonstrate how our approach scales, depending on the recursion depth. Specifically, for incremental values of n (which corresponds to the maximum recursion depth), we perform 100 gradient descent iterations on the (tail) recursively defined Exponent function x^n. Our approach is compared with the purely imperative one and the one that statically unrolls the graph prior to runtime, by using the `tf.function` decorator.

The results, displayed in Figure 6.1 imply that our transformation scales better for larger recursion depths. This can be attributed to the fact that the other methods 'expand' their

structure as the size of the input n increases, whereas, in our method, the computational graph remains the same. The aspect that expands in our case, is the amount of data that flows within the graph, originating from different contexts.

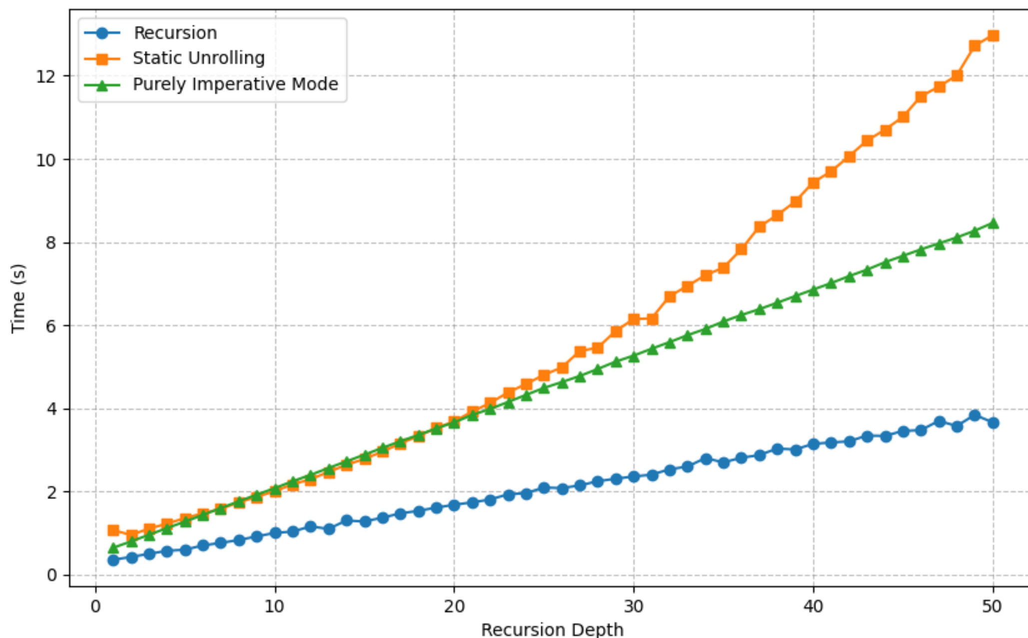


Figure 6.1: Minimizing the Exponent Function - 100 iterations

```
import tensorflow as tf
import time

@tf.function # Comment out for purely imperative mode
def f(n,x):
    if n == 0:
        return tf.constant(1.0)
    else:
        return x*f(n-1,x)

x = tf.Variable(1.0, dtype=tf.float32)
optimizer = tf.optimizers.SGD(learning_rate=1e-10)

def make_test(n):
    start_time = time.time()
    for step in range(100):
        with tf.GradientTape() as tape:
            loss = f(n,x)

        grad = tape.gradient(loss, x)
        optimizer.apply_gradients([(grad, x)])

    total_time = time.time() - start_time
    print(f"Final result: x = {x.numpy()}")
    return total_time
```

Figure 6.2: Python Test Code: Static Unrolling

```
import os
import time
import tensorflow as tf
from tensorflow.python.framework import function


tf.compat.v1.disable_eager_execution()
tf.compat.v1.disable_control_flow_v2()


exp = function.Declare("EXPONENT", [("x", tf.float32), ("n", tf.int32)],
                        [("ret", tf.float32)])


@function.Defun(tf.float32, tf.int32, func_name="EXPONENT",
                create_grad_func = True, out_names=["ret"])
def ExpImpl(x, n):
    return tf.cond(tf.equal(n,0),
                lambda: tf.constant(1.0),
                lambda: x*exp(x,n-1)
            )

def make_test(n):
    tf.compat.v1.reset_default_graph()
    x = tf.Variable(initial_value=1.0, dtype=tf.float32)
    n = tf.constant(n)
    y = ExpImpl(x,n)

    optimizer = \
        tf.compat.v1.train.GradientDescentOptimizer(learning_rate=1e-10)

    train_op = optimizer.minimize(y)

    sess = tf.compat.v1.Session()
    sess.run(tf.compat.v1.global_variables_initializer())

    start_time = time.time()

    for step in range(100):
        sess.run(train_op)

    total_time = time.time() - start_time

    final_x, final_f_x = sess.run([x, y])
    print(f"Final result: x = {final_x}, f(x) = {final_f_x}")

    return total_time
```

Figure 6.3: Python Test Code: Recursion

7. FUTURE WORK & CONCLUSIONS

7.1 Graph Load Balancing

In our approach we have demonstrated that in TensorFlow's graph execution mode, it suffices to include a single copy of each used function within the main graph. However, even though this approach allows for recursive function definitions and efficient gradient calculation in small graphs, in larger graphs, performance issues could arise. If a certain function were to be called through many different call sites in a large graph, then all of the data would have to pass through a single instance of the function body, given that there is only one throughout the entire graph. A possible improvement could be to add more than one instances of each function body, under certain circumstances, so that the load is balanced between multiple parts of the graph. This would allow for better parelleliza-tion amongst available devices, and it would yield more favourable results in distributed runtime.

7.2 Demand Driven Model

As of right now, TensorFlow's computational model is based on *data driven* dataflow, meaning that each node/operation is executed if and only if all of its arguments are available. An alternative approach would be to incorporate a *demand driven* [4] model, where each node awaits for a *demand* to be received from succeeding nodes. Naturally, this implies that, in order to initiate a computation, users would have to signal a demand that starts at the end of the graph and propagates backwards, towards the source nodes. This approach could be beneficial, as it avoids making redundant computations that do not contribute to the final result, potentially improving the execution time. It would be interesting to examine the behaviour of a demand driven architecture, or even a 'hybrid' mode of operation, that alternates between the two methods [11].

7.3 Adapting to Other Frameworks

Another interesting future direction would be to examine whether this transformation can benefit other popular machine learning frameworks, such as PyTorch [10]. It is worth examining whether the core idea that is proposed in this transformation can also be adapted into automatic differentiation mechanisms of imperative execution, such as the gradient tape.

7.4 Conclusions

The main goal of this project was to reintroduce recursive function definitions in TensorFlow and to present a suitable mechanism for computing the gradients of such functions. We believe that recursion is a powerful technique that should be exploitable by machine learning architectures, as it enables the definition of complicated, hierarchical structures in a convenient manner. According to our experimentation, our approach yields promising

results and it has the potential to perform even better in distributed execution environments.

For reasons unknown to us, TensorFlow's support for recursion has diminished after the release of TensorFlow Eager. We hope that this work could provide some insight in ways to establish a mechanism that allows for recursion in TensorFlow and is compatible with the existing features of the framework.

ABBREVIATIONS - ACRONYMS

NKUA	National and Kapodistrian University of Athens
TF	TensorFlow
HUA	Harokopio University of Athens
CPU	Central Processing Unit
GPU	Graphics Processing Unit
TPU	Tensor Processing Unit
API	Application Programming Interface

BIBLIOGRAPHY

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning, 2016.

[2] Ackerman. Data flow languages. *Computer*, 15(2):15–25, 1982.

[3] Akshay Agrawal, Akshay Naresh Modi, Alexandre Passos, Allen Lavoie, Ashish Agarwal, Asim Shankar, Igor Ganichev, Josh Levenberg, Mingsheng Hong, Rajat Monga, and Shanqing Cai. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning, 2019.

[4] E. A. Ashcroft. *Dataflow and education: Data-driven and demand-driven distributed computation*, pages 1–50. Springer Berlin Heidelberg, Berlin, Heidelberg, 1986.

[5] Kostopoulou Calliope. Recursive function definitions in static dataflow graphs and their implementation in tensorflow, 2018.

[6] Jishnu Ray Chowdhury and Cornelia Caragea. Modeling hierarchical structures with continuous recursive neural networks. In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 1975–1988. PMLR, 18–24 Jul 2021.

[7] Andreas Griewank. A mathematical view of automatic differentiation. *Acta Numerica*, 12:321–398, 2003.

[8] Andreas Griewank and Andrea Walther. *Evaluating derivatives: principles and techniques of algorithmic differentiation*. SIAM, 2008.

[9] Rishiyur S. Nikhil. Executing a program on the mit tagged-token dataflow architecture. In Mark D. Hill, Norman P. Jouppi, and Gurindar S. Sohi, editors, *Readings in Computer Architecture*, pages 323–341. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000.

[10] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[11] Panagiotis Repouskos. The dataflow computational model and its evolution, 2017.

[12] P. Rondogiannis and W. W. Wadge. First-order functional languages and intensional logic. *J. Funct. Program.*, 7(1):73–101, January 1997.

[13] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[14] TensorFlow. Recursive tf.functions are not supported, 2023. Accessed: 2024-07-25.

[15] TensorFlow. tf.gradients api documentation, 2024. Accessed: 2024-07-26.

[16] Seiya Tokui, Ryosuke Okuta, Takuya Akiba, Yusuke Niitani, Toru Ogawa, Shunta Saito, Shuji Suzuki, Kota Uenishi, Brian Vogel, and Hiroyuki Yamazaki Vincent. Chainer: A deep learning framework for accelerating the research cycle. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD ’19, page 2002–2011, New York, NY, USA, 2019. Association for Computing Machinery.

[17] TensorFlow Whitepaper. Implementation of control flow in tensorflow, 2017. Available at: `http://download.tensorflow.org/paper/white_paper_tf_control_flow_implementation_2017_11_1.pdf`.

[18] Yuan Yu, Martín Abadi, Paul Barham, Eugene Brevdo, Mike Burrows, Andy Davis, Jeff Dean, Sanjay Ghemawat, Tim Harley, Peter Hawkins, Michael Isard, Manjunath Kudlur, Rajat Monga, Derek Murray, and Xiaoqiang Zheng. Dynamic control flow in large-scale machine learning. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys ’18. ACM, April 2018.