Universidade do Minho
Escola de Engenharia
Departamento de Informática

Gonçalo José Azevedo Esteves

Functional Programming for Explainable AI

July 2023

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Gonçalo José Azevedo Esteves

Functional Programming for Explainable AI

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
José N. Oliveira (U.Minho)

July 2023

COPYRIGHT AND TERMS OF USE FOR THIRD PARTY WORK

ACKNOWLEDGEMENTS

As a first note, I would like to thank my parents for everything they have done for me. I am certain that all that I was able to achieve was because of their sacrifices in order to let me dream. I hope that this is one more step that I take that will make them proud.

After, I feel the urge to thank my supervisor, Professor José Nuno Oliveira. I am sure that, without him, nothing of this would be possible. His help and guidance throughout this process were precious and undispensable.

I also need to thank all my family and friends for all of the support and love that they have always given me. On a more particular note, I do need to express my feelings towards every single soul that I have met during my academic journey and that I now have the privilege to call a friend. I am sure that I have learned a lot from everyone, but more importantly, nothing would have been the same without you by my side since the very beginning.

Last but not least, I need to thank my grandfather, Valdemar Jorge, for always being my biggest fan. I am sure that you will continue to protect me, wherever you are.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

ABSTRACT

Neural Networks, increasingly used in Artificial Intelligence, are computational devices inspired by existing biological neural systems, seeking to be capable of learning how to perform tasks and recognize complex patterns. Internally, Neural Network programs are made up of several small structures called neurons (by analogy with Biology) which are responsible for handling input values in order to determine what the output values should be. The fact that these programs are organized hierarchically makes it plausible applying compositional patterns, often associated with functional programming, in order to obtain more refined neuronal networks, and whose understanding would be easier.

This dissertation intends to focus more on the possibility of using the high compositionality presented in functional languages, namely Haskell, in order to make Neural Network programming better structured and elegant, facilitating not only the creation but also the understanding of what goes on inside these systems, which are sometimes seen as black boxes, due to the great lack of knowledge about how they work.

KEYWORDS Artificial Intelligence, Neural Networks, Compositionality, Functional Programming, Haskell

RESUMO

No estudo da Inteligência Artificial é comum serem várias vezes referenciadas um conjunto de estruturas designadas de Redes Neuronais. Estas nada mais são do que representações computacionais inspiradas nos sistemas nervosos biológicos existentes, e que procuram tornar-se capazes de aprender não só a executar tarefas como também a reconhecer diferentes padrões, por exemplo. Internamente, estes programas são constituídos por diversas pequenas estruturas, chamadas de neurónios, tendo em conta a analogia estabelecida com a Natureza, e que são responsáveis por tratar valores de entrada de modo a determinar quais os valores de saída. O facto destes programas possuírem uma organização tão hierarquizada torna plausível a possibilidade de aplicar padrões composicionais, muitas vezes associados a programação funcional, de modo a obter Redes Neuronais mais refinadas, e cuja compreensão seja mais fácil.

Apresenta-se esta dissertação com a intenção de debruçar mais sobre a possibilidade da utilização da elevada composicionalidade presente em linguagens funcionais, nomeadamente Haskell, de forma a tornar a programação de Redes Neuronais num processo mais simples e elegante, facilitando não só a criação mas também o entendimento daquilo que se passa dentro destes sistemas, que por vezes são vistos como caixas negras, devido ao grande desconhecimento sobre a forma como funcionam.

PALAVRAS-CHAVE Dissertação de Mestrado, Programação Funcional, Composicionalidade, Inteligência Artificial, Redes Neuronais, Haskell

CONTENTS

Contents

LIST OF FIGURES

ACRONYMS

AI	Artificial Intelligence. 9, 21, 57
BPTT	Backpropagation through time. 36, 38, 48
BRNN	Bidirectional Recurrent Neural Network. 41–43, 46–48, 58
CNN	Convolutional Neural Networks. 17, 18
DL	Deep Learning. 4, 5, 10, 11, 16, 19, 36, 47–49
FM	Formal Methods. 19, 20, 47, 56, 57
FP	Functional Programming. 4–6, 10–12, 14, 19, 23, 27, 28, 48, 50, 56–58
GRU	Gated Recurrent Unit. 49, 51, 55
LSTM	Long Short Term Memory. 48, 49, 58
ML	Machine Learning. 3–6, 8, 10, 14, 17–21, 27, 36, 48, 56–58
NN	Neural Networks. 4–6, 10–19, 21, 22, 25–31, 33–38, 41, 47, 49, 50, 57, 58
RL	Reinforcement Learning. 8, 18
RNN	Recurrent Neural Network. 36–38, 40–43, 45–49, 51, 58
SL	Supervised Learning. 6–8, 18, 25–27, 29
UL	Unsupervised Learning. 7, 18
XAI	Explainable Artificial Intelligence. 4, 6, 9, 19, 57

1

INTRODUCTION

> *"(...) ML systems have been highly resistant to compositional description and analysis. However, there is a major push in current research to achieve this, in order to have explainable, verifiable and accountable AI."*
>
> Samson Abramsky (2022)

As explained by Zhou (2021), *Machine Learning (ML)* is a collection of Artificial Inteligence techniques that make some software systems capable of improving their own performance by learning from experience, via computational methods. The overall idea is heavily inspired by humans' way of learning from their every day experience and to make decisions based on such experience. When talking about computer systems, experience outcomes are stored as data and the programs will have to learn from such data, through so-called *learning algorithms* that enable them to create models of natural language or of particular situations in real life like interpreting images or videos.

The amazing evolution of hardware capabilities in recent times has made ML systems to improve and evolve significantly, while humans keep on finding more applications and uses for them, from the "simpler" and more common ones like *Google Translate*, that are now used by virtually anyone to perform the most varied tasks, to the more complex ones like virtual assistants as Apple's *Siri* and Amazon's *Alexa*. These systems turn the search for information an easier job, since they are capable of interpreting voice commands and performing the required tasks. Applications for these systems seem to be limitless, and the more they evolve, the more doors they will open.

With the emergence of ChatGPT[1], available online since November 2022, public opinion has become effervescent around the world on the subject of artificial intelligence. ChatGPT, an OpenAI creation, is a large language model trained to interact with the user in a conversational way, providing information on the most varied subjects, while being able to answer follow-up questions, admitting its mistakes, challenging incorrect premises and rejecting inappropriate requests (OpenAI, 2022). The views on this program differ greatly, with many seeing it as a great tool with a lot of potential, as shown by Twelverays (2023), while others start to ask if the problems that appear with its use justify it, like the undermining of journalism, the difficulty for universities to properly evaluate their students or the possibility of the extinction of some professions, as explained by Ranger (2023). Some

[1] https://chat.openai.com/

countries, like Italy, went as fair as banning the program because of privacy concerns (McCallum, 2023a), having only removed the ban after extra security measures were implemented (McCallum, 2023b).

Despite all recent advances in ML and *Neural Networks (NN)*, the truth is that there is still a long path to follow before these disciplines are *scientifically explainable*. Concerns about this state of affairs have already reached public opinion, as newspaper articles such as the one published recently by Spinney (2022) in The Guardian show. People are concerned about programs that are used in public services and seem to work, but no one knows exactly how.

ML techniques are more and more used in critical applications, whereby software development shifts from traditional coding to example-based training. This raises the need for achieving confidence in ML modeling through new forms of verification and validation and through *Explainable Artificial Intelligence (XAI)* techniques.

Several safety standards (e.g., ISO 26262, IEC 62304, EN50128, DO-178C) and security standards (e.g., Common Criteria, ISO/IEC 15408, NIST Publication 800-53), which have matured over the years, are thus facing new challenges. The work to be carried out in this project is triggered by the following research questions:

1. What "is" Machine Learning?
2. How explainable is it?
3. How compositional is it?

In particular, this project proposal arises from an interesting conjecture by Christopher Olah (2015a) in his research blog:

> *Strongly typed* Functional Programming (FP) *can play an important role in* XAI.

According to Olah, at present three narratives are competing with each other to become *the easy way* for one to understand *Deep Learning (DL)*:

- the neuroscience narrative,
- the probabilistic narrative and
- the representations narrative.

Data transformation is formalizable in a rigorous and calculational way (Oliveira, 2008) and Olah's conjecture suggests that the representation narrative can be easily associated with FP's type theory. Besides this, many models in DL, if not all of them, can be depicted as optimization problems related to function composition.

Considering that both type theory and the idea of function composition are some of the founding stones of FP (Bird and de Moor, 1997) and that such notions can be intuitively associated to other DL notions, we are led to conjecture that these two areas can effectively help each other, in particular leading to a better understanding of the NN body of knowledge.

1.1 MAIN GOALS

The lack of explainability in ML techniques is somewhat in contradiction with its heavy use of maths (linear algebra, in particular), which should shed some light on how ML programs work.

If we take into consideration the fact that this sort of technology is being increasingly more used in new applications, and many of them require high assurance (like those used in banks or medical necessities) we can easily understand why this sort of problems must be overcome, since they can put human lives at risk.

Considering our starting research questions, we propose to focus on compositionality of NN, in order to prove the third question, that ML can actually be compositional (in part, at least). With this, we also hope to help in the understanding of the second one (how explainable can ML be). The first one we understand that is something that one cannot explain easily, since after all these years no one was capable of answering it.

In summary, the overall goal of this study is to understand how far DL and FP can go together, in order to (possibly) surpass some of the problems that NN presents to us, namely lack of confidence, using the techniques and calculi that FP provides us with (Bird and de Moor, 1997).

Put in other words, validating Olah's conjecture is the main aim of this dissertation. The experimental part of the project will be carried out in Haskell.

2

STATE OF THE ART

This chapter is devoted to a state-of-the-art review of the literature on ML, while analyzing what is being currently researched in its intersection with FP.

The discussion will start by reviewing *supervised*, *unsupervised* and *reinforcement learning*, indicating the pros and cons of each approach and their applications. This is followed by dissecting why XAI is a hot topic and the reasons that make us regard it as an interesting subject for research in applied FP. Last but not least, we will present some previous work on how ML and FP can be combined. Before all that, let us just make a quick review of the three narratives that were referred to previously.

- **Neuroscience Narrative** - draws analogies with Biology, comparing the functioning of networks with how biological neurons work;
- **Probabilistic Narrative** - relates NN to finding latent variables;
- **Representations Narrative** - is based on data transformations and the manifold hypothesis.

As stated above, we will center our work in the third narrative, since it is the one that offers more correlation to FP (Olah, 2015a).

2.1 MACHINE LEARNING

2.1.1 *Supervised Learning*

Supervised Learning (SL) is an approach to ML that is characterized by its use of labeled training datasets (training sets that have inputs associated to their expected outputs) in order to train a NN to better classify data or determine the desired outputs (IBM Cloud Education, 2020a). With this, the model is able to adjust its weights so it can make correct predictions, using a loss function to determine if the error has been sufficiently minimized or if there are more changes needed in order to improve its accuracy.

The problems addressed by this strategy can be separated into two different types: *classification* problems and *regression* problems. The former involves the recognition of entries, trying to decide how they should be labeled or classified, using for that algorithms that assign such entries to specific categories. This can be applied, for example, to the automatic association of messages received in an inbox to their respective folders (important, spam, ...). The latter are related to understanding how dependent and independent variables are connected. Projections, such as sales revenue for a given business, are among possible applications that can be approached.

Despite all advantages that SL offers, like deep data insights and improved automation, challenges arise in order to build sustainable models. These can require high levels of expertise and knowledge to structure accurately, since their complex nature and sometimes difficult understanding can limit the capabilities of the programmer to build extensive and efficient models. Not only this, but model training can be very time consuming, since the training sets are pretty large in order to make the programs as precise as possible. Lastly, it is possible for the datasets to contain errors, since they are sometimes man-made, which can lead to algorithms learning incorrectly.

2.1.2 *Unsupervised Learning*

Unsupervised Learning (UL) is a strategy that implies the use of specific algorithms that are capable of interpreting unlabeled datasets in order to find common characteristics or patterns among data and aggregate them into information clusters (IBM Cloud Education, 2020b). Thanks to this, the strategy is able to group information into different sets without human intervention. This approach is ideal for some types of task, namely (a) *clustering*, which is the process of grouping unlabeled data according to a given characteristic (in which data can be similar or differ), structure or pattern found in the information; (b) *association*, that allows to find and establish relationships between variables in a dataset using for that a rule-based technique, being example of this the market basket analysis that allows companies to observe connections between products; and (c) *dimensionality reduction*, which is a method applied when the dataset has a high number of features, or dimensions, making it possible to reduce the number of data inputs, in order to optimize the performance of the algorithm and facilitate the interpretation of datasets without risking their integrity.

In spite of all of the pros regarding the use of UL, there are some barriers that occur when models are allowed to run without human intervention. For example, thanks to the high volumes of the training sets that are used, two distinct phenomena occur, one being a stronger need to use more computationally complex models (since the datasets are harder to interpret) and the other being long training times, even longer than those observed with other methods. There is also a higher probability of obtaining inaccurate and unwanted results, since there are no comparable outputs in the training data and, despite the search of results being completely autonomous, human analysis at the end is still needed, in order to validate outputs. Another problem that sometimes happens is the lack of transparency regarding the reason why data are clustered the way they are, with programs being unable to explain the reasons behind the distribution that is achieved.

2.1.3 *Reinforcement Learning*

Reinforcement Learning (RL) approaches other side of ML, where one tries to create an autonomous agent that is able to observe and analyze the environment in which it is inserted in order to make optimal decisions allowing it to achieve its goals (Mitchell, 1997). The main focus of the agent is to achieve the maximum reward possible on a sequence of actions, rewards being sums of numerical values that are given by reward functions, as an evaluation of each action performed. (The reward function can, for example, assign a positive value for a good action taken by the agent and a negative value for a bad action.) With this, it has to be capable of analyzing the results of performing different sequences of actions, in order to learn a control policy that enables it to maximize the final reward, independently of the initial state. This strategy can be applied in many different areas, such as robot learning, *bots* (software programs that are able to perform a series of automated tasks) and sequential schedulling problems.

Learning a control policy can be similar to a function approximation problem, but they are not the same, since there are many differences that distinguish them. For starters, the final reward value we want to maximize is not previously known, because we will only determine it after performing all of the required actions (and even the reward value of each action is known only after performing it); this contrasts with to what happens in SL, where we already know the output for every input when teaching the program with a training set. Next, instead of having fixed training sets, RL agents influence the distribution of them, thanks to the different actions they can take (which arises to the learner the question of what experimentation strategy is the best in a determined case: to keep on exploiting the known states and actions or to explore new ones). Besides this, it is possible for the agent to not have full knowledge on the actual state of the environment at a given time, which limitates decision making and may trigger the need of using previous observations too, in order to find a pattern. Finally, in many RL applications, the agent is obliged to learn a variety of different tasks related to the same environment in which it is inserted, enabling the possiblity of using the information gathered about a past task to facilitate the learning process of a new task.

2.2 EXPLAINABLE ARTIFICIAL INTELLIGENCE

Thanks to the constant increase of accumulated data and the consequent need for programs capable of interpreting them, ML techniques have been in the vanguard of application development, with some even arguing that the new programs that are developed using these concepts are now better than humans when it comes to find relationships among data, exposing man-made theories as oversimplifications of reality. Individuals even hypothesized that it could be the end of the scientific method, with theories being put into a corner and people embracing the simple correlations that were given to them (Spinney, 2022).

There is some truth associated to this, since the huge amount of information and complexity that are currently provided to us make it impossible for one to be able of writting theories useful for describing the presented problems, since experts do not even know how such theories could be formalized. Despite all this, theory refuses to die and there may be some reasons that lead to this. First of all, there is still the possibility of unknown "traditional theories" that are yet to be exposed. Second, *Artificial Intelligence (AI)* is incredible yes but it is fallible. Last but not least, humans themselves do not feel comfortable with something when they do not know how it works. And this is where XAI enters the scene.

XAI intends to create systems capable of providing more insight to humans on how they effectively work, giving for that more explanations not only on their capabilities and understandings, but also on what happened until a certain point, what will happen at that moment and what will happen in the future; they should also allow observers to know the important information that they use in order to obtain the results (Gunning et al., 2019).

Despite all this, it is relevant to refer that all explanations are context dependent, since the intended result may vary regarding who asks for them and for what they are using them. For starters, models can be fully interpretable (if the given explanations are transparent and provide a complete view of the system) or partially interpretable (if the given explanations only provide information on important pieces of their reasoning process). The determination of what is the best technique to use and what kind of information will be given always depends on the final use of the given explanation, and this is many times related to the type of user that seeks that information. Some users may search for explanations regarding the reasons why given results are the ones observed, while others may want to know how the results were obtained; there is even the possibility of users wanting answers for both of these aspects. Being the ultimate goal of XAI to provide effective explanations to humans, it should always take into consideration the target user group, in order to better suit the given answers to their background and expectations. When all this is taken into consideration, there are challenges and issues that arise, some of which to be presented next.

First, it is hard to decide if programs should take into account the knowledge that users have or do not have, as well the characteristics of the users that may obtain the explanation; also, can we get to a point where humans provide feedback of the given explanations, in order to generate an interactive system? It can also be difficult to find a “sweet spot” between accuracy (all the precise information that allowed the program to generate the result) and interpretability (what users can effectively understand), so one can obtain a understandable yet useful explanation. Besides this, there is always the challenge of finding abstractions that allow for latter explaining phenomena that occur, like high-level patterns that allow a simpler explanation of big plans in big steps. As a last example, we can refer the need to distinguish among decisions that were taken from the competencies that the system has, so that end users can understand if some result was not the intended one because of the path that was taken to find the solution or if the program is not capable of solving the problem.

2.3 MACHINE LEARNING AND FUNCTIONAL PROGRAMMING

As some researchers have already shown and hypothesized the ML and FP areas of knowledge are closer to each other more than one might think of. In this section, we will review previous work done carried out in this direction.

2.3.1 *Christopher Olah's View*

The state of Deep Learning and predictions about the future

Olah (2015a) defends that DL is still in its "ad-hoc state" meaning that because it is a relatively young field, it is still built in a very necessity based way, originating solutions that work and can solve problems without being completely understood by humans. Also, he argues that the whole DL concept is stuck together with the help of a tool that enables one being capable of creating solutions, despite having no way to create a general agreement on how DL should be seen and in what foundations should it be supported. In this context, he theorizes that DL will certainly be very different in the future and speculates about in how one will see DL in the future, despite knowing it is impossible to predict what is going to happen.

As previously stated, there are three narratives used currently to understand how someone can see DL. Not being mutually exclusive, they certainly represent different views: the *neuroscience narrative*, the *probabilistic narrative* and the *representations narrative*. Christopher Olah leans over the last one in this case, providing the proposition:

> *DL studies a connection between optimization and FP.*

The idea is to formalize a theory that establishes the parallelism between the representations narrative in DL and type theory in FP, seeing DL as the junction of two rather diverse areas.

The parallels between Deep Learning/Function Composition and Representations/Types

Anyone can effectively affirm that one of the characteristics of DL is embracing the study of deep neural networks, which are multilayered NNs that transform data across each layer in order to solve a given task. Despite the details of the layers being constantly evolving, the notion of "sequence of layers" stays the same, with each layer being a function that transforms the output of a previous layer. One can undoubtedly see this as a chain of composed functions that must be optimized so that it can solve the problem that was presented, and it is this statement that Olah regards as the heart of what we are studying.

Every layer of a NN transforms data in order to better fit the purpose of the task that is being performed. These transformed versions of data can be seen as "representations", which correspond to types in computer science, i.e. ways of embedding information in an arbitrary amount of bits. We can establish the parallel between types and representations if we look into the latter as a way of embedding information in an arbitrary number of dimensions. The similarities continue if we take into account that the same way two functions can only be composed if their types agree, two layers can only be composed when their representations agree.

In basic NN architectures with linear sequences of layers this becomes easy since there is only the need of making the output of a layer match the input of the next one, but the same can not be said when more complex architectures are built. When there are multiple input layers converging into one representation and multiple outputs diverging from the same representation, things get trickier. Despite this, it is plausible to think that one can formulate things in a way that makes it possible for all of the pieces to stick together in a harmonious way.

Considering that representations and types are the building blocks for DL and FP respectively, that one of the narratives of DL (representations narrative) is based on the fact that NNs transform data into new representations, and that there are many known relations between maths and FP, the author argues that the connections between representations and types are maybe very relevant.

Deep Learning and Functional Programming

When it comes to programming, function abstraction is an amazing technique that allows for one to reduce not only the amount of code written (since a piece of code is written once and used whenever it is needed, instead of repeating it) but also the probability of introducing bugs in programs, while making it easier to detect errors. Once again, we can start to establish a parallel between this and another DL technique, weight tying, which enables the use of many copies of the same neuron in a NN, making models learn more quickly since there is less to learn. The reuse of a neuron must be done with criteria though and one must understand where it could be placed, taking advantage of the structure in data so it will not harm the well functioning of the model.

There are a bunch of patterns based on this, such as recurrent and convolutional layers, that are widely used in NN and work as higher-order functions, which are functions that take other functions as arguments and are deeply studied in FP. Many of these patterns work similarly to well known functions, being the main difference the fact that instead of receiving normal functions as arguments, they receive chunks of a neural network. Some examples are:

- **Encoding Recurrent Neural Networks** - These work as *folds* and are often used in order to enable the NN to receive lists of data with variable lengths, like strings.

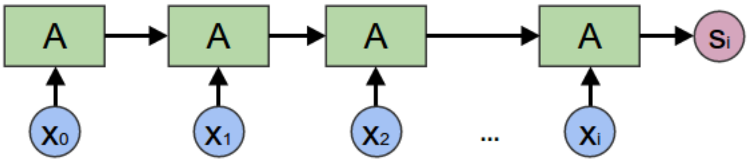

Figure 1.: Representation of an Encoding Recurrent Neural Network

- **Generating Recurrent Neural Networks** - These resemble *unfolds* and their intent is to allow the NN to create a list of outputs, such as words in a sentence.

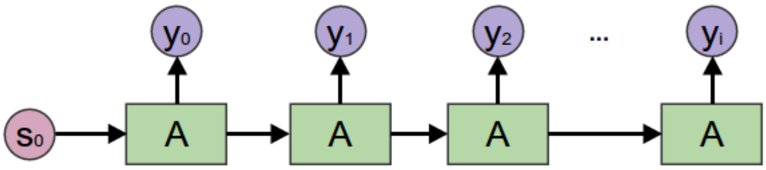

Figure 2.: Representation of a Generating Recurrent Neural Network

- **General Recurrent Neural Networks** - These *accumulating maps* are often used so systems can try to make predictions in a sequence. An example can be the prediction of a phenome in every time step in an audio segment, based on past context, when developing a voice recognition program.

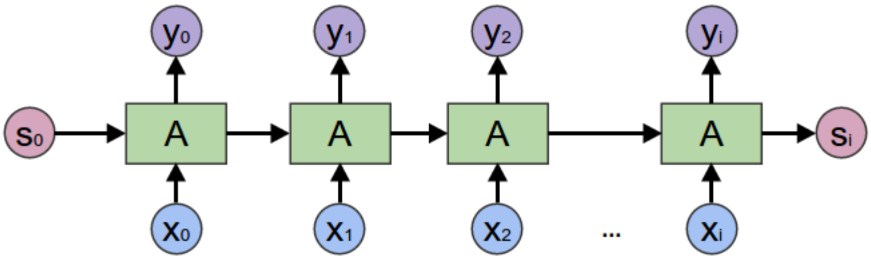

Figure 3.: Representation of a General Recurrent Neural Network

- **Bidirectional Recursive Neural Networks** - These can be seen as a *left and right accumulating map* zipped together and it is used to predict information from a sequence using both past and future context.

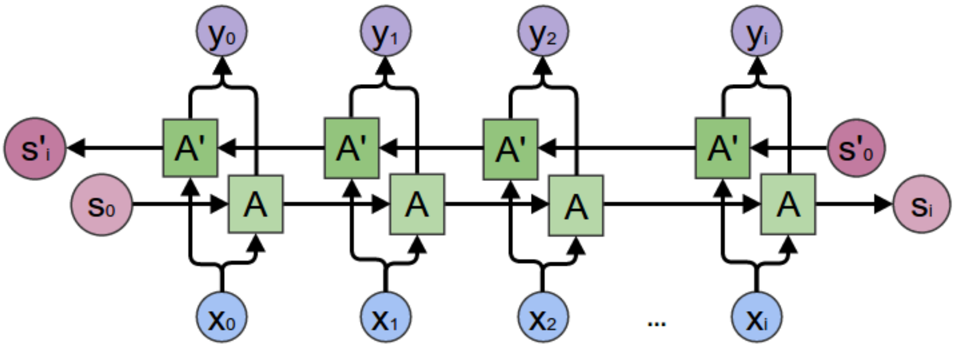

Figure 4.: Representation of a Bidirectional Recursive Neural Network

- **Convolutional Neural Networks** - These are similar to *maps* but not the same, since maps apply a function to every element while convolutional NN apply it also at neighboring elements.

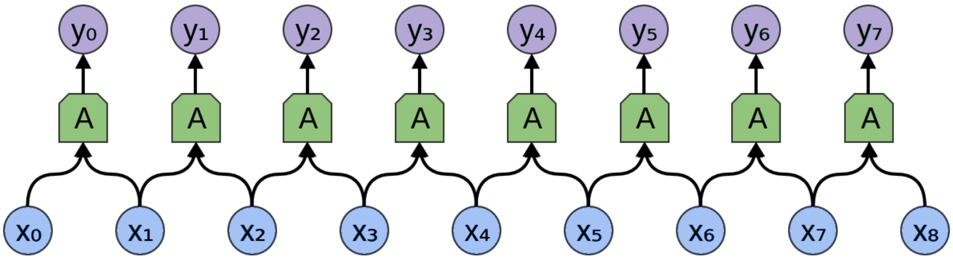

Figure 5.: Representation of a Convolutional Neural Network

- **Recursive Neural Networks** - These are also known as "TreeNets", they are a generalization of *catamorphisms* (Oliveira, 2019) that consume a data structure in a bottom-up pattern and are mainly used in natural language processing, so that NN can operate on parse trees.

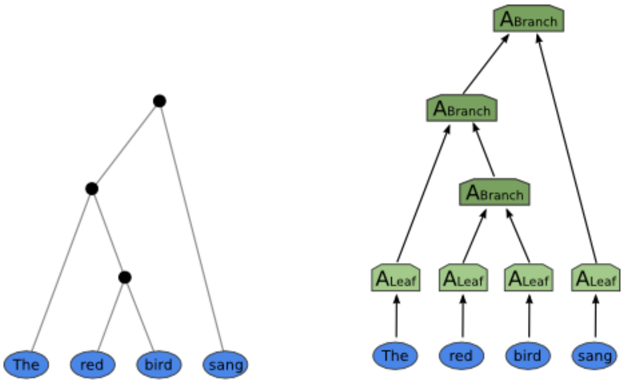

Figure 6.: Representation of a Recursive Neural Network

One can effectively see such a network of different patterns combined together as large functional programs with chunks of NN throughout. Being a functional program, it provides a high level structure, while still allowing for the chunks to be flexible pieces that actually learn to do the requested task inside the framework provided by the program. This is the intersection that Olah idealizes in order to somehow create a new kind of programming, capable of constructing programs that are able of doing things no one can create directly from scratch.

2.3.2 *Bogdan Penkovsky's View*

Learning Neural Networks The Hard Way

After a brief introduction on how Penkovsky (2020) was presented to ML, more specifically to NN and its very basics, he starts by referring some concepts about Haskell and FP, and why he believes these are two tools one can use in order to further develop NN. First of all, he believes that Haskell is a particularly good language for NN since these are very "functional objects": a network can be seen as a large composition of functions, which is a very common concept in any functional language. He also enumerates some advantages of Haskell as a language, like the ease on reasoning about what a program is doing or how one can refactor its base code, the ability to improve someone's capability on thinking towards a solution for a problem in a more pragmatic way and the fact that Haskell based programs run fast. One can easily understand why all of these features are very appealing for anyone who is developing NN, since NN systems are very hard to interpret (in what regards comprehending what is happening), need to be altered many times before obtaining the final version and must run as fast as possible.

The author then explains what gradient descent is and its importance in NN, letting one understand how by using this method, the error on a network, given by the error function, can be systematically decreased until a minimum error value is achieved. He then continues by giving a demonstration on how this can be implemented in Haskell, and how the written code works.

With this presented, he leans over a concrete example of a network, in this case a program to classify a flower into its specific classes, using for that some distinctive characteristics that distinguish each other. For starters, this example is useful to explain why natural numbers must not be used when referring to a class, but vectors with a number of dimensions equal to the number of existing classes, so that the distances between all of them are the same. After this, and considering what was previously written about gradient descent, he puts the pieces together in order to create the intended network, explaining step-by-step how it must be done, demonstrating not only the calculus that has to be made, but also how to convert it to a Haskell program and how to interpret the given results.

What Do Hidden Layers Do?

In the second part of his blog, Penkovsky begins with a brief explanation on why multi-layer NN can be so useful, sheding some light on their ability to enable a system to learn non-linear relations between inputs and outputs, providing a way to dodge the linear classifier trap.

With this in mind, he creates an example of a program that receives patterns made with dots and tries to associate a class to any new dot inserted based on its coordinates. This example is continuously used in order to better illustrate the observed phenomena, since only multi-layer architectures are capable of solving such tasks (single-layer architectures are only capable of drawing straight lines in the input space).

Using this example as a basis, several important aspects of the system's modeling are taken into account, namely:

- the need for NN weight random initialization in order to create neurons capable of learning distinct features;
- the need for using not only an adequate NN architecture for the task that we are solving, but also an effective training method so one can prevent the program from getting stuck with a suboptimal setting;
- being careful with all activation functions, ensuring all remain nonlinear in order to avoid that a multi-layer NN "collapses" into a single-layer one;
- the fact the ideal number of neurons and layers in a NN may depend on the problem that is being solved (in many cases, more neurons on a single layer do not represent a faster capacity to learn thanks to the fact that the composition of simple transformations made by a fewer number of neurons into more complex ones - what happens in multi-layer architectures - may benefit the resolution of non-linear tasks, since an architecture with more layers can better represent more complex relationships).

A curious fact is that, in order to better justify this last explanation, Penkovsky cites another post from Olah's blog (Olah, 2014), regarding the topological transformations that a NN does in the input space.

The author then proceeds demonstrating how all this can be implemented in Haskell, starting by recalling backpropagation, used in the previous demonstration, and how recursion can be used in order to make it more intuitive, while also constructing the basic data structures and functions that work as the founding stones of the program. He also creates the bridge between the *for* loops used in imperative programming languages, that are very useful for NN development, and Haskell's higher order functions, like *map* and *zip*. With all this, one is capable of starting the construction of the model itself, implementing the gradient descent function and the backprop algorithm. Then both the training and the model validation datasets are built, the architecture is defined and the weights are properly initialized. He also explains how an alternative to the gradient descent can be implemented, in order to avoid getting stuck in suboptimal settings in more complex tasks. In the end, a series of tests are carried out, giving evidence of the benefits of using the techniques referred previously.

Haskell Guide to Neural Networks

In order to continue to make one better understand the doors that Haskell opens on matters of DL studies, Penkovsky writes a new post on the importance of automatic differentiation and how this technique can be applied using this programming language.

He begins by explaining how random search works, giving for that an example that later also helps one to understand why it is not the most effective way to maximize the sum of two values, for example, since the inputs are changed in random directions, something that sometimes neither allows to optimize the previous value nor takes into account all of the work done until that point. It is here that automatic differentiation makes a step in, being capable of changing the input in such a way that the output is always improved. This is possible because the algorithm defines specific gradients only for elementary operators, combining them according to the chain rule so that the necessary gradients will be inferred to the strategy itself. All of this can be implemented in Haskell, as the author shows, and be adapted so that it fits the needs made up by the NN. In addition to this, one is also presented to a Haskell library that facilitates the construction of differentiable programs, composed by many tools that prove it to be useful when programming based on this principles.

The Importance of Batch Normalization

As Penkovsky states, there is one main challenge relative to NN that can be divided in two parts. First, one must know how to train the millions of parameters that compose the net and second, there is a need to know how to interpret them. In recent years, a new technique as appeared that makes training more efficient, reducing in a very significant way the number of training epochs and making possible the training of certain architectures. This method is batch normalization and consists on the division of a huge dataset into smaller mini-batches that will be processed one at a time during a forward or a backward pass. The precise way this is done varies depending on the phase of the learning process that is running at the moment, the existing differences between the application of the procedure at the training and the inference phases.

But how efficient can this method really be? When developing a program that is capable of recognizing human-written digits, tests show that a NN with batch normalization can reach higher accuracy values with way less epochs than a NN without batchnorm. However, and despite all of the good results that are presented, this technique presents some pitfalls that can sometimes hinder program construction. The fact that the methods used during the training phase are different from those used in the inference phase makes the implementation more complicated, while there is also a need to guarantee that all the training data has the same origin, which means one must ensure that every batch represents the whole dataset, having data that comes from the same distribution as the ML task being performed. Also, the truth is that there is still not a consensual explanation on why batchnorm is effective (similarly to what happens in many other matters related to ML). Initially, it was hypothesized that it reduced the internal covariate shift, while recent studies show that this may not be necessarily the case and that the true explanation may reside in the fact that it makes landscape smoother, providing a more efficient way to train gradient descent and allowing to use higher learning rates.

The author then proceeds to show how to modify the previously written Haskell code, so that the program becomes capable of supporting batch normalization, by explaining how to alter the data structures and the functions that were already made and how to initialize all the parameters needed.

Convolutional Neural Networks Tutorial

In this post, Penkovsky talks about *Convolutional Neural Networks (CNN)*, labeling them as the "master algorithm" in computer vision and explaining how they work and can be applied. This type of architecture also has a neuron as its building block and is differentiable, allowing a proper backpropagation training, but its distinctive feature resides in the connection topology that results in sparsely connected convolutional layers, where neurons share their weights, defined by a kernel, resulting in a reduced number of trainable parameters when compared to fully-connected layers. They were especially designed in order to address two specific issues, translation simmetry and image locality, and their capacity to solve them is associated to the convolutional application of an image kernel.

There are many different convolution types, namely Computer Vision-Style Convolution, dedicated to low-level computer vision that operates on one, three or four channels, in which a individual kernel is applied to each channel; LeNet-like Convolution, where a single convolution operates simultaneously on all input channels, producing a single channel, while also allowing for any number of output channels to be obtained depending on the number of kernels; and Depthwise Separable Convolution, where there are not only individual kernels applied to each individual channel but also another convolution in-between channels after that, with an optional activation before that.

Following up on all this, the author builds up a Haskell example based on a LeNet-5, but first gives an explanation on how one can obtain the convolutional layer gradients algebraically. After that, he assembles all the needed pieces so the NN can be made, constructing the data structures and the base functions. The use of stencils[1] and how to implement them is also covered, since these are very important tools in the creation of a CNN. In the end, tests show that an error twice as low as the one obtained with a simple fully-connected can be achieved, while also requiring a drastically lower number of learnable parameters.

[1] As explained by Penkovsky, stencils are patterns used in CNN that are capable of processing array elements. They are implemented on convolutions in order to process given points, making this approach also able of performing subsampling operations while allowing for efficient data parallelism

2.4 SUMMARY

In ML, distinct strategies are better suited for distinct problems. SL algorithms require the existence of labeled data, which they use in order to determine final outcomes or to distribute inputs throughout their possible categories. Despite tending to be more accurate, they require human intervention in the creation of training sets with correctly labeled information. However, labeled datasets enable the programs to avoid computational complexity, since they will not need training sets as big in order to produce the intended outcomes (IBM Cloud Education, 2020a). On the other hand, UL algorithms only require unlabeled data, from which they find patterns that enable the creation of groups and allow solving association or clustering problems. This can be useful in matters in which experts are not sure about the characteristics associated to a data set. There is still semi-supervised learning, which is an approach that involves giving only a partially labeled training set (IBM Cloud Education, 2020b), mixing these two previous ideas together. By adding RL algorithms to the mixture, even more different problems can be covered, since this strategy allows the creation of programs that are able to interpret information relative to its surroundings in order to perform a series of actions that enable it to achieve its goal. With this, one can make autonomous agents, that can learn from previous decisions and actions taken, in order to continually evolve and make better decisions (Mitchell, 1997).

Despite the existence of many different types and applications of ML, most of them still lack one very important aspect, *explanations*, and this is where XAI kicks in. With it, one can hope to build programs that can be understandable by humans, surpassing the "black-box phase" in which many of them are, in order to shed some light in all of the magic that happens behind the curtains. This technology can potentially be limitless and mark the start of a new era, where XAI systems can interact with humans in many intricate ways, in order to sky-rocket our society to the next level (Gunning et al., 2019).

One starts to see some studies and predictions about how FP and *Formal Methods (FM)* can help in the development of ML techniques, like the ones presented in the blogs of Olah (2015a) and Penkovsky (2020), but despite them we still lack knowledge on how these areas can be effectively combined. Olah theorizes on how one is capable of creating NN using as building blocks the similarities between DL and FP, giving for that examples on how these two areas converge, while Penkovsky shows that it is possible to create many different NN using FP, namely Haskell, but there is still a long way to go until someone can assert with certainty that these areas can cooperate. The main intent of this project is to help carving that path.

3

FORMAL METHODS MEET MACHINE LEARNING

Life-cycles for program design in formal methods (FM) follow a general, golden principle: it all starts with a formal (i.e. mathematical) *specification*, seen as an inviolable description of the customer's expectations, from which a running program is developed by stepwise *refinement*. This means that, in every stage n of the design process, a refinement S_n is a valid implementation of the original specification S — written $S_n \vdash S$ —, refinement consisting of improving the current version of the implementation until it becomes a runnable artifact, i.e. a program (Oliveira and Rodrigues, 2006).

Every time S_n is replaced by S_{n+1}, what is required of S_{n+1} is that:

$$\langle \forall\, i, o, o' \;:\; o\; S_n\; i :\; o'\; S_{n+1}\; i \wedge o = o' \rangle$$

(assuming deterministic, possibly partial) S_i. That is, their outputs are the same for the same inputs.

It turns out that (unfortunately) this method is not general practice in the software industry. Instead of completely formally specifying customers' intents, the starting step S is usually a test suite consisting of expected input-output pairs. Such text suite works as an imperfect, incomplete specification, since it will only describe the behaviour of the program for a given set of inputs, leaving the program to behave randomly for any other input data.

3.1 WHAT ABOUT MACHINE LEARNING?

In ML, the starting point is similar to what is observed in the rest of the software industry: it all starts with a *training set* S, which is just another name for *test suite*. As already mentioned, this makes the "specification" an imperfect one, since it is defined by a set of tests and not by mathematical rules. It is therefore impossible to create a perfect implementation, given that the tests do not include all possible cases. Consequently, the program will be prone to errors, being an acceptable implementation modulo the probability of such errors surfacing at runtime.

With this in mind, achieving our objective becomes even harder, since now the creation of a model S_n implies making a program that is capable of producing outputs for any given input, even with no previous information on it, while simultaneously ensuring the lowest output error margin possible, since equality between the produced output and the expected one will be impossible to achieve in most cases. All this can be translated into the following rule:

$$S_n \vdash_\epsilon S \quad \equiv \quad \langle \forall\, i, o, o' \,:\, o\, S_n\, i :\, o'\, S\, i \wedge |\delta\ (o, o')| \leqslant \epsilon \rangle$$

In other words, a program S_n will be considered a good implementation of the given "specification" S if the modular difference between the produced output and the expected output, calculated by a δ function, is not greater than a predetermined error threshold ϵ.

Still on this parallel between ML and program formalization, one can view the iterations of models S_n produced when the network is trained by a training set (in the training phase) as the ML counterpart to the improvement of programs when errors are spotted by programmers in the testing (vulg. debugging) phase. The main goal of training the network is to learn with it by adjusting its internal parameters, so one can effectively affirm that all of these intermediate iterations can be considered different models of the program, which will ultimately lead to an optimized final version. This final version should be, in theory, the one where it is possible to achieve the lowest error margin, so the program gets closer to a perfect implementation.

3.2 IT'S ALL FUNCTIONAL

When starting to analyze the problem, it is easy to understand that the relation described in a ML specification must be a function: it has to be entire, since all inputs must produce an output, and it has to be simple, since all inputs must produce no more than one output. Thanks to this, one can rewrite the previously shown expression by using function notation,

$$s_n \vdash_\epsilon s \quad \equiv \quad \langle \forall\, i, o, o' \,:\, o = s_n\, i \wedge o' = s\, i :\, |\delta\ (o, o')| \leqslant \epsilon \rangle$$

which is nothing but:

$$s_n \vdash_\epsilon s \quad \equiv \quad \langle \forall\, i \,::\, |\delta\ (s_n\, i, s\, i)| \leqslant \epsilon \rangle$$

This endorses the purpose of this study: to uncover the functional nature that lays beneath ML and NN and explain how it may be the key to create new and better methods for developing AI programs.

3.3 TOWARDS AN IMPLEMENTATION

Moving deeper into the proposed functional implementation, one can start to hypothesize how this could work. According to the introduction to NN of the previous chapter, it is easy to understand that, in most cases, the number of neurons of the input layer should be the number of input variables or columns, while the output layer is advised to have a number of neurons similar to the number of different possible outputs or a single neuron that determines the final output by itself. For example, if one tries to develop a program that is able to recognize a digit $(0\,..\,9)$ from an image, the number of neurons in the input layer should equal the number of pixels in the image, while the output layer should have ten neurons, one for each possible digit. Of course this example does not encompass the whole set of imaginable problems, since some are more complex and may require not so direct approaches, but it suffices for introducing the process.

Hidden layers are a lot trickier, and there is lack of general consensus on how to determine not only the number of hidden layers, but also the number of neurons that such layers need. This largely depends on a great number of factors, for example the number of input/output values and training cases, the amount of noise in the targets, the complexity of the task to be carried out, the architecture or the regularization — just to mention some (Sarle, 2002). Many defend that the best way to determine these parameters is by trial and error, until one achieves an acceptable combination. For many tasks, one or two hidden layers are more than enough to get the job done, and there are even cases where none are needed, since they will only bring more complexity to a problem that can get good results with a simpler approach.

When the matter is the number of hidden units, there are some "rules of thumb", depending on which considerations are made. For example, concerning input and output sizes, some say that the size of the hidden layer should be somewhere between them, others try to define more rules that calculate a precise value based on the number of input and output units, while others state that a hidden layer should be at most twice as big as the input layer; on the other hand, concerning the number of training cases, there is a recommendation that says that the number of weights should not be greater than a predetermined percentage of the training cases. If we take into consideration even more variables, like the different techniques that can be used to perform and optimize the program, the difficulty in determining the structure of the hidden layers just rises, giving more strength to the hypothesis of trial and error being the best shot at getting an accurate program. This turns the formalization of the problem into something even more complex, since now one is not only unable to rely on a perfect specification but also on a rule that always ensures the creation of the best possible architecture.

3.4 FUNCTIONAL PROGRAMMING BASICS

Before anything else, it is important to shed some light into concepts that will be extensively used in the sequel because of their relevance in FP.[1] We start by explaining the arrow notation that will be used in the sequel. The generic arrow

$$A \xrightarrow{f} B$$

represents a function f with input type A and output type B. That is, f will produce objects of type B wherever objects of type A are passed as inputs. This simple and elegant way to describe information processing is key to help reasoning on how a function, or a program, behaves, while producing a precise way to depict what is expected of it. On the other hand, it also helps explaining another very relevant feature of FP, which is function

[1] Most of the material in this section is inspired on the work developed by Oliveira (2019).

composition. One can say that two functions can be composed together when the output type of one of them is the input type of the other. For example, and using the following diagram,

$$\begin{tikzcd} A \arrow[r, "f"] \arrow[rr, bend right, "f \cdot g"'] & B \arrow[r, "g"] & C \end{tikzcd}$$

f and g can be composed together since the output type of the first (B) is the input type of the second. With this, the function $f \cdot g$ will be such that objects of type C are produced from objects of type A, abstracting from the intermediate type B.

Another very important concept is that of a higher order function. These functions are characterized by the fact that they can accept other functions as inputs and yield a function as output. For example, the map function,

$$A^* \xrightarrow{map\, f} B^*$$

receives not only a function $A \xrightarrow{f} B$, but also a list of objects of type A, committing itself to producing a list of objects of type B. Practically speaking, the map function will be responsible for applying the f function to each one of the elements present in the input list of A's, in order to produce a new list of B's. This is achievable since f maps objects of type A into objects of type B. Note that this example of higher-order function is not chosen randomly: the map function will be extremely important later in this dissertaion.

Up next, one needs to clarify the future usage of two very important functions, which are in and out.

$$\begin{tikzcd} A \arrow[r, bend left, "\mathsf{out}"] & F\,A \arrow[l, bend left, "\mathsf{in}"] \end{tikzcd}$$

Functions in and out are each other inverses: they work in opposite ways and their composition is equivalent to the id function. Technically, they are said to be *isomorphisms*, i.e. functions that do not lose information.

The out isomorphism is responsible for exposing the internal organization of a given, complex, usually recursive data structure, while its inverse in works on the opposite way, packaging the given information into the original data structure. These functions are of the utmost importance, since their mathematical properties allow for reasoning and proving useful properties about constructs that involve them (Oliveira, 2019), as is the case of the so-called *catamorphism* and *anamorphisms* combinators, explained next.

Catamorphisms and anamorphisms are higher-order recursive program combinators. This means that they *generate* recursive programs from given generating functions, referred to as *genes*. The following diagram describes the catamorphism generated by gene f:

$$\begin{tikzcd}
A^* \arrow[d, "\llparenthesis f \rrparenthesis"'] \arrow[r, bend right, "\mathsf{out}"'] & 1 + A \times A^* \arrow[l, bend right, "\mathsf{in}"'] \arrow[d, "id + id \times \llparenthesis f \rrparenthesis"] \\
B & 1 + A \times B \arrow[l, "f"]
\end{tikzcd}$$

Note how the catamorphism notation $\llparenthesis f \rrparenthesis$ captures the idea of f being its generator function, i.e. the catamorphism's gene.

Catamorphisms are known for their capacity of structurally transforming complex data structures into other (normally simpler) data types. In the example given above we see the diagram representation of a list catamorphism, where a given list A^* is used to produce a new object of type B. The list's structure is initially exposed thanks to the use of the $\mathtt{out}$ function, expressing the fact that a list is either empty (1) or composed by a head and a tail ($A \times A*$). Next, the catamorphism is applied to the tail of the list, in case it exists, while keeping the rest of the information as it is. Lastly, the obtained data are converted into an object of type B by applying the f function. Similar reasoning can be applied to other complex data structures, since the process is the same: exposing the structural way the data is stored, applying the catamorphism in the place where the information is kept "intact", summing up everything into a new object.

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d, "{[\!(f)\!]}"'] & 1 + B \times A \arrow[d, "id + id \times {[\!(g)\!]}"] \\
B^* \arrow[r, bend right, "\mathsf{out}"'] & 1 + B \times B^* \arrow[l, bend right, "\mathsf{in}"']
\end{tikzcd}$$

Anamorphisms, on the other hand, work in the opposite way. Represented by the $[\!(g)\!]$ notation, parametric on the anamorphism's gene g, they are usually responsible for creating complex data structures from other (normally simpler) objects. In the diagram above, which depicts a list anamorphism, the g function is applied to an object of type A to create a data structure where the information is stored in a similar way to what is expected in the final list structure. This is then followed by the (recursive) application of the anamorphism in the piece of data that keeps its s original form, creating a list of B's. After that, the information is covered with the usage of the $\mathtt{in}$ function, creating the intended B^*. Once again, this line of thought can be extended to any anamorphism, where, initially, a new, more complex, data structure is created, followed by the conversion of the remaining original information into the new structure and the final wrap up of all data involved.

This completes the brief summary of the notation and concepts that are used in the sequel to dscribe abstract models of neural networks. Readers interested in a deeper understanding of the rich theory behind the approach are referred to (Oliveira, 2019) and the literature mentioned there.

3.5 FROM WORDS TO DIAGRAMS

Resuming to section 3.3, the main goal below is to try to formalize the networks as much as possible, in order to better understand how all notions connect and what the best path towards achieving the best possible implementation is. All tasks that are expected to be accomplished by the program are done in order to solve at least one of two main problems: training the network and/or determining the output for an input data.

Let us start from the "big picture". From a given neural network NN one expects to extract a function *feed*

$$\begin{tikzcd} NN \arrow[r, bend right, "feed"'] & I \to O \end{tikzcd}$$

which, based on the given NN, is capable to produce an output (O) from any possible input (I). Suppose one just generates a random function *feed* and that its behaviour is poor, meaning that it needs to be improved. In SL, this is achieved by going in the opposite direction through a function *learn*

$$\begin{tikzcd} NN \arrow[r, bend right, "feed"'] & I \to O \arrow[l, bend right, "learn"'] \end{tikzcd}$$

which, equipped with an element of a training set (of type $I \to O$, now playing the role of supervisory data) generates an (as good as possible) network. The "training" process will then have its basis on a function *train*, which will be nothing but the composition of *feed* and *learn*, performed as many times as needed to go throughout all of the elements of the training set.

$$\begin{tikzcd} NN \arrow[r, "feed"] \arrow[rr, bend right, "train"'] & I \to O \arrow[r, "learn"] & NN \end{tikzcd}$$

Because the NN is a sequence of layers, the two (higher-order) functions *feed* and *learn* are dual in nature, once expressed in the functional programming jargon: while *feed* is a *fold* (or *catamorphism*), *learn* is an *unfold* (also known as *anamorphism*) (Oliveira, 2019).

This big picture is useful to set up the main functional ingredients of SL, but it is an oversimplification. Below we give a more detailed view of what needs to be set up for the overall scheme to work.

To begin with, let us explain how outputs are calculated in more detail by *feed*

$$I \xrightarrow{feed\ nn} O$$

that will be responsible for, provided a network and an input, producing an output. How?

The "feeding" process will be characterized by going through all layers of the NN, in a sequential way, while performing algebraic operations, namely matrix-vectorial multiplications. These operations will modify the input received by each layer and produce an output that will be passed onto the next layer, which means that the output of one layer is the input of the one that follows. Once the last layer is reached, its output is the final output of the whole program.

Because the semantics of each layer is a function, the parallel between this and function composition is obvious: the “feeding” process is nothing but the composition of n functions, n being the number of transitions between consecutive layers of the network:

$$\begin{tikzcd}
I \arrow[r, "f_1"] \arrow[rrr, bend right, "feed\ nn"'] & A \arrow[r, "f_2"] & ... \arrow[r, "f_n"] & O
\end{tikzcd}$$

Each function f_i performs the linear transformations that are associated to each transition (given by a pair of weights and biases), in the same order that they appear in the NN. This also gives a first hint on what training means — the improvement of $feed$, which will be made by improving the functions it composes. But how is this achieved?

When looking into the “training” process, things get a little trickier. In a first step, the network receives an input and produces an output, similarly to what happens in the “feeding” process, but after that, things may differ regarding the learning technique that is being used. For example, when approaching SL, the network then proceeds to compare the output that was created with the one in the training set, since labeled training sets will be used. The difference between the produced and expected outputs triggers a process of refining the parameters of each layer (weights and biases) so that the next $feed$ works better.

This second process, or function, will be $learn$, and will have a different semantics depending on the different learning process that is used. Once again, when considering SL, its output will be a new NN, and it will be generated from the current one together with the expected and produced outputs:

$$(NN \times O) \times O \xrightarrow{learn} NN$$

The inputs that are passed on are a little different from the ones stated above, but this “learning” function will be responsible for passing through all layers and update all of the networks’ internal values, a layer at a time, starting with the most external one. It will compare the output values that were expected to be produced with the ones that were obtained and determine the variation, use this to create the layer’s δ values, and then use these values to update the weights and biases of that layer. It then proceeds to the next layer and to determine the δ values of the new layer it uses the ones from the layer above. After this, the internal parameters are updated and the program continues in the next layer. This process now repeats itself until it reaches the initial layer, where, after updating its values, it will stop, having created a new and updated NN.

Now, in order to have the full training process, the *train* function is obtained by composing *feed* and the *learn* functions, so the value produced by the first one is used in the second one. With this in mind, it gets easy to reach the following idea:

$$\begin{tikzcd}
I \arrow[r, "feed\ nn"] \arrow[rr, bend right, "train\ (nn{,}o)"'] & O \arrow[r, "learn\ (nn{,}o)"] & NN
\end{tikzcd}$$

This is the main (overall) idea behind the NN implementation (in Haskell) that will be given in the next chapter.

3.6 SUMMARY

The structural way how the “feeding” and “learning” processes in ML resemble folds and unfolds in FP, respectively, and how the “training” process is nothing more then the composition of the previous two, makes one understand that, effectively, ML’s more complex parts can be translated into FP combinators that help us better understand how they work! When approaching deep NN, the obvious similarities between the layers’ operability and function composition are the living evidence that these two areas have a lot in common. Exploring this relationship, the chapter which follows will give practical evidence of how to use FP “schematology” to implement NNs.

4

NEURAL NETWORKS IN HASKELL

This chapter describes our own experiment in building a NN in Haskell from scratch. The inspiration comes from the attempt by Lynn (2015) to create a simple yet effective program in Haskell that is capable of recognizing handwritten digits. This case study proves to be adequate since it will be based on SL techniques (thanks to the given training set), that were sketched in the diagrams of the previous chapters.

4.1 BUILDING A SIMPLE NEURAL NETWORK

In a first approach, the case study was analyzed and some small changes were made to the code, so as to make it easier to understand how it works. This was followed by more profound changes using other studies as a solid base for the modifications.

To make the usage of linear algebra in all of the operations more apparent, the first step revolved around switching to the Matrix library[1] of Haskell, which abstracts from its *lists of lists* implementation. This change not only enables more accurate data structuring but also facilitates future changes, if needed. To make the needed operations easier, not only the information depicted in the list-of-list format (i.e. matrices) was converted into proper matrices, but also the one stored in lists (i.e. vectors), since these are nothing but matrices with only one column/row. After all this, some major changes were implemented inspired on Nguyen and Wu (2022) work, whereby new data structures and strategies were built and applied.

First of all, abstractions were created in order to help better understand the code, making it easier to distinguish the different matrix uses (such as e.g. which information a matrix supports, if it is a weight matrix or a bias vector, and so on). Besides this, a data structure that keeps all of the information within a layer was created, in order to better organize the information, since any layer's internal structure can be easily described using its weights and biases. Such a $Layer$ structure (please check code on page 59) is also built in such a way that recursion can be represented abstractly in its type parameter. After this, a new structure was implemented, to make possible the use of fix points, allowing for a relocation of all explicit recursion.

The idea behind such modifications is to help better defining and using the recursion schemes that will be present in the program, namely folds, which instantiate catamorphisms, and unfolds, that can be described as anamorphisms. If now one remembers Olah (2014)'s blog, we can start to establish the parallels between his

[1] https://hackage.haskell.org/package/matrix-0.3.6.1/docs/Data-Matrix.html

hypothesis and the work carried out by Nguyen and Wu (2022) now present in this project too: folds and unfolds can be found in many of the NN's patterns.

FEEDING Starting with "feeding", one has to consider two different aspects of the process. First, that it receives two arguments, being one nn, which is the NN of type $Fix\ Layer$ (abbreviated to FL in the diagram below), where the usage of fix points enables to sustain all network layers, and the other the input data of type $Values$ (abbreviated to V in the same diagram). Second, that the produced output will be a vector with a probabilistic distribution of the input to be any of the desired outputs, stored in a $Values$ structure. With this, the definition of the $feed$ function can be translated into the following diagram:

$$\begin{tikzcd}
V \arrow[d, "(,)\ nn"] \arrow[ddd, bend right=60, "feed\ nn"'] \\
FL \times V \arrow[d, "feed'"] \\
V^* \arrow[d, "head"] \\
V
\end{tikzcd}$$

where the two input arguments are combined into a pair sent to auxiliary function $feed'$, while $head$ is responsible for extracting the final output produced by the network, since $feed'$ will create a list of $Values$ with all of the outputs produced by each layer of the NN, being the first entry in the list the last one created. With this being said, the auxiliar function $feed'$ is represented as it follows.

$$\begin{tikzcd}
FL \times V \arrow[d, "(\!| alg |\!) \times id"] \arrow[dd, bend right=60, "feed'"'] \\
(V \rightarrow V^*) \times V \arrow[d, "applyR"] \\
V^*
\end{tikzcd}$$

Auxiliar function $applyR$ evaluates the first argument of a pair (which must be a function) to the second one, while the core of all the "feeding" process is developed by a FL catamorphism.

Catamorphisms receive an *algebra function* (alg), of type $f\ a \rightarrow a$, that describes how a data structure is recursively evaluated to a final value (the type of this value is called the function's carrier type). So alg can be regarded as the step-function that is applied through the whole recursive process. In our case, the catamorphism will recursively evaluate the structure of type $Fix\ Layer$, that represents the NN, down to an output of type $Values \rightarrow Values^*$, unwrapping for that the constructor of Fix (F) and then interpreting the constructors of

Layer (abbrev. L) with in the algebra. The output function that is created will be able to receive data of type *Values*, which will be the input of the NN, and produce a list with the outputs made by each layer in the network.

$$\begin{tikzcd}
FL \arrow[d, "(\!|alg|\!)"'] \arrow[rr, bend right, "outF"] & & L\ (FL) \arrow[ll, bend right, "inF"'] \arrow[d, "fmap\ (\!|alg|\!)"] \\
(V \to V^*) & & L\ (V \to V^*) \arrow[ll, "alg"]
\end{tikzcd}$$

The algebra function alg will be responsible for creating a function capable of originating the output made by a new layer as well as the outputs made by all of the previous layers. For that it must receive a data structure of type $Layer\ (Values \to Values^*)$, where the information of the current layer (biases and weights) is stored as well as the function that can construct the list with the outputs of the previous layers.[2] The architecture of this algebra is given in the following diagram:

$$\begin{tikzcd}
L\ (V \to V^*) \arrow[d, "outL"] \arrow[ddd, bend right=60, "alg"'] \\
1 + ((W \times B) \times (V \to V^*)) \arrow[d, "id+f\times id"] \\
1 + ((V^* \to V^*) \times (V \to V^*)) \arrow[d, "{[\underline{singl}, \widehat{(\cdot)}]}"] \\
(V \to V^*)
\end{tikzcd}$$

So, alg is a higher-order function whose details are as follows: $outL$ will expose the data hidden by the layer's structure[3], while f — the main component of alg — is responsible for receiving the *Weights* (W) and *Biases* (B) of the current layer and transform them into a function capable of producing an output of type $Values^*$ when receiving an input of the same type. What this means is that the function issued by f will receive a list with all the previous layers' outputs and produce a new one that contains not only those outputs but also the output of the current layer, created when applying the layer's weights and biases to the head of the received list (which is the last output produced).

The final step $[\underline{singl}, \widehat{(\cdot)}]$ will then be used to return the function $singl$[4] when receiving an empty layer (that can only be the input layer) or to compose the function created by f and the one already stored in the layer's structure.

[2] Some brief explanations concerning notation: $A + B$ and $A \times B$ denote the disjoint union of types A and B, respectively. Notation $\underline{k}$ denotes the everywhere-k constant function, that is, $\underline{k}\ x = k$. Currying and uncurrying of binary functions are denoted by $\overline{f}\ a\ b = f\ (a, b)$ and $\widehat{g}\ (a, b) = g\ a\ b$, respectively.

[3] The abstract structure $1 + X$ means a "pointer to X", i.e. it is either a null (1) or contains data of type X.

[4] Function $singl$ returns singleton lists, $singl\ a = [a]$.

LEARNING Moving on to the "learning" process, more specifically the $learn$ function, one can interpret it as an anamorphism.

Anamorphisms receive a *coalgebra function* ($coalg$), of type $a \rightarrow f \ a$, that describes how a data structure is constructed from an initial value (with its type being the carrier type of the coalgebra). In our case, the anamorphism will generate a $Fix\ Layer$ structure (abbreviates FL, as usual) from a seed of type $Fix\ Layer \times BackProp$, using the coalgebra to replace the occurences of the carrier type with constructors of $Layer$, wrapping the result at the end with the Fix constructor.

$$\begin{tikzcd}
FL \times BP \arrow[r, "coalg"] \arrow[d, "learn"'] & L\ (FL \times BP) \arrow[d, "fmap\ [\!(coalg)\!]"] \\
FL \arrow[r, bend right, "outF"'] & L\ (FL) \arrow[l, bend right, "inF"']
\end{tikzcd}$$

$BackProp$ (abbrev. BP) will be an auxiliary structure designed to sustain all the needed information to perform backpropagation: when modifying the i^{th} layer, it will contain the $deltas$ and previous weights (weights used before updating) of the $i+1^{th}$ layer, a list with the outputs produced by all layers and the output that was expected to be produced by the program. The $coalg$ will then be responsible for transforming the received pair into a structure of type $Layer$, i.e. a new layer for the NN, where the the weights and biases are updated and the structure's parameter contains the information needed to do the next backpropagation.

$$\begin{tikzcd}
FL \times BP \arrow[d, "(outL \cdot outF) \times id"] \arrow[dddd, bend right=60, "coalg"'] \\
(1 + ((W \times B) \times FL)) \times BP \arrow[d, "distl"] \\
(1 \times BP) + (((W \times B) \times FL) \times BP) \arrow[d, "! + g"] \\
1 + ((W \times B) \times (FL \times BP)) \arrow[d, "inL"] \\
L\ (FL \times BP)
\end{tikzcd}$$

In detail: $outL$ and $outF$ combine efforts to expose the information hidden within the received $Fix\ Layer$, the first element of the input pair, while $distl$ will be responsible for rearranging the pair. The final inL will combine the created data into a new $Layer$. Before this, either there is nothing to do ($!x = ()$) or there is a lot of work to be performed by the g auxiliary function. In it, not only the weights and biases will be updated, but also the information needed for the next backpropagation will be created. Function g is the composition of several steps, as follows:

$$\begin{tikzcd}
((W \times B) \times FL) \times BP \arrow[d, "\langle \pi_1 \times id, \langle (\pi_1 \cdot \pi_1) \times id, \pi_2 \cdot \pi_1 \rangle \rangle"] \arrow[ddddd, bend right=60, "g"'] \\
((W \times B) \times BP) \times ((W \times BP) \times FL) \arrow[d, "\widehat{backward} \times id"] \\
((W \times B) \times D) \times ((W \times BP) \times FL) \arrow[d, "assocr"] \\
(W \times B) \times (D \times ((W \times BP) \times FL)) \arrow[d, "id \times assocl"] \\
(W \times B) \times ((D \times (W \times BP)) \times FL) \arrow[d, "id \times (swap \cdot (bp \times id))"] \\
(W \times B) \times (FL \times BP)
\end{tikzcd}$$

The first *split*[5] is responsible for reorganizing the data so that the application of the following functions gets easier, like the function *backward*, that will be the engine of this process. In a first stance, it will be responsible for determining the *Deltas* of the current layer, needing for that either the data that comes from the previous backpropagation, or the expected output, when working on the ouput layer. After that, those values will be used to update the weights and biases of the layer, that will be returned in the end along with the Delta values.

```
backward :: (Weights, Biases) → BackProp → ((Weights, Biases), Deltas)
backward (w1, b1) (BackProp w2 d2 out (a1 : a0 : as)) = ((w1′, b1′), Just d1)
  where d1 = case d2 of
    Nothing → pointMult (elementwise dCost a1 out) (fmap relu′ a1)
    Just d2′ → pointMult (d2′ ∗ w2) (fmap relu′ a1)
    w1′ = descend w1 (outProd a0 d1)
    b1′ = descend b1 d1
```

Isomorphism $assocr\ ((a,b),c) = (a,(b,c))$ reorganizes the information, giving the pair yet another structure, and after that new changes will be applied to the second element of the pair by $assocl\ (a,(b,c)) = ((a,b),c)$. Primarily, internal reorganization will be made, using *assocl* and *swap* but a new *BackProp* will also be made, using *bp* for that, so that the information needed for the next backpropagation is made available.

```
bp (d, (w, (BackProp _ _ out a))) = BackProp w d out (tail a)
```

[5] Splits of functions produce pairs: $\langle f, g \rangle\ a = (f\ a, g\ b)$. Pairs are decomposed by so-called projection functions, $\pi_1\ (a,b) = a \wedge \pi_2\ (a,b) = b$. Pairs can be swapped, cf. $swap\ (a,b) = (b,a)$.

TRAINING Now working on the “training” process, since it will need to go through a list of input/output pairs in order to create a new NN, one can idealize it functioning as a catamorphism too, but in this case it will be a catamorphism of lists. Having this in mind, the following diagram captures the *training* function:

$$\begin{tikzcd}
(V \times V)^* \arrow[d, "training\ nn"'] \arrow[r, bend right, "outList"'] & 1 + (V \times V) \times (V \times V)^* \arrow[l, bend right, "inList"'] \arrow[d, "id + id \times (training\ nn)"] \\
FL & 1 + (V \times V) \times FL \arrow[l, "{[\underline{nn}, h]}"]
\end{tikzcd}$$

In detail: *training nn* is a catamorphism that receives a list of pairs of values and produces a new *Fix Layer*. In this case, the argument *nn* is the initial NN, that will be refined by training in order to create a new one. The algebra of the catamorphism is an alternative computation that will return the initial network when the list of pairs is empty or, when the list is not empty, apply the auxiliary function h to the next input/output pair, as well as the most recently updated network. This function will be responsible to train the network and can be represented by the following scheme:

$$\begin{tikzcd}
(V \times V) \times FL \arrow[d, "assocr"] \arrow[ddd, bend right=80, "h"'] \\
V \times (V \times FL) \arrow[d, "id \times swap"] \\
V \times (FL \times V) \arrow[d, "train"] \\
FL
\end{tikzcd}$$

where the first two steps simply reorganize the received pair, and the *train* function is structured as shown below:

$$\begin{tikzcd}
V \times (FL \times V) \arrow[d, "{\langle \pi_1 \cdot \pi_2, id \times feed' \rangle}"] \arrow[ddd, bend right=80, "train"'] \\
FL \times (V \times V^*) \arrow[d, "id \times \widehat{bp2}"] \\
FL \times BP \arrow[d, "learn"] \\
FL
\end{tikzcd}$$

As one can see, *train* is basically a “composition” of $feed'$ and *learn*, where some adaptations have to be made in order to apply directly the referred functions. The initial $\langle \cdot, \cdot \rangle$ preserves a copy of the initial NN, while the expected output is passed on directly to the next step, since it will not be used immediately. This output will be paired up with the produced list of $Values$, that are nothing more than all of the outputs produced by all of the layers, when the function $feed'$ is applied to the pair $Values \times Fix\ Layer$, which are input and network, respectively. The created pair is then passed on to $bp2$ to yield a $BackProp$ structure.

$$bp2 = BackProp\ (zero\ 1\ 1)\ Nothing$$

Having now a pair $Fix\ Layer \times BackProp$, where the first element is the current network and the second one the information needed to make a first backpropagation and consequently to start the learning process, one can apply $learn$ directly in order to create a updated version of the current NN.

All the Haskell code was created from the diagrams shown above and can be found in Appendix B.

4.2 FROM NNS TO RNNS

After approaching a multi-layered NN, one can start to lean over other kinds of complex ML structures, like, for example, *Recurrent Neural Network (RNN)*'s. This kind of networks are characterized by the sharing of parameters between all of the layers of the program, as well as their usage in the analyses of sequential or time series data (IBM Cloud Education, 2023). These DL algorithms are especially used in matters related to natural language processing, be it written or recorded, like translators and voice assistants. The main cause for allowing the programs to work in such a way that the treatment of sequential data is enabled is related to the intern use of a hidden state, which is nothing more than a vector that stores information related to previous inputs in order to use it when calculating the current output. With this, the network is built to allow the processing of related data, since the previously received inputs will influence the outputs created in the future.

The "learning" algorithm that allows the NN to learn, this is, to determine the gradients, is called *Backpropagation through time (BPTT)*. It is similar in some ways to traditional backpropagation, since the gradients are determined thanks to the computation of errors from its output layer to its input layer, allowing the tuning of the model's parameters. The main differences reside in the fact that BPTT sums the calculated errors at each time step, while normal feedforward networks do not need to do so, since their layers do not share parameters.

This parameter sharing may seem like an inconvenience at first glance but one quickly understands that, besides all of the already referred advantages, there is also one more that can be pretty helpful. Since all layers share the same parameters, it is possible to define one single layer, with all of the needed parameters, and use it repeatedly. With this in mind, a new data structure was created in order to store all of the information needed for a general RNN's layer, called $RLayer$ (see page 59 in the appendix). This structure is composed by two normal layers: the first one will be responsible for processing the received information (input and previous hidden state) and use it to create the new hidden state; the second will be used for transforming the created hidden state into an output. The final result of each iteration of the RNN is a pair with the new hidden state, that will be used in the next computation, as well as the created output, that will be returned in the end. In this case, normal layers are used since neither multi-layered structures are considered, nor the usage of more advanced techniques common to RNN's. The data structure can be seen at Appendix A. The implementation, namely the way how gradients are calculated during the BPTT, is inspired by the work of Krishna (2022), as this author provides an in-depth explanation of backpropagation in RNN that is very useful.

FEEDING Once again, and following the same schema used previously, one will start by explaining how the "feeding" process works for this specific kind of NN using diagrams:

$$\begin{tikzcd}
V^* \arrow[d, "feedR'\ (nn{,}s)"] \arrow[dd, bend right=60, "feedR\ nn\ s"'] \\
V \times BPR^* \arrow[d, "id \times map\ (\pi_1 \cdot outBR)"] \\
V \times V^*
\end{tikzcd}$$

The *feedR* function will receive three arguments, being the first one the RNN, the second one the initial state that will be used by the program and the third one the input sequence. The produced result will then be a pair where the first element is the final state produced, and the second element is the list of outputs.

feedR relies on *feedR′* as main engine of the "feeding" process, producing not only the output values, but also all the information that will be needed in the future in order to execute the backpropagations. This information will be stored in a new structure called *BackPropR* which is composed by nothing more then four *Values* vectors: produced output, produced hidden state, received hidden state and received input. The definition of the *feedR′* in Haskell is as follow:

$$\begin{aligned}
&feedR' :: (RLayer\ (Values \to Values), Values) \to [Values] \to (Values, [BackPropR]) \\
&feedR'\ (nn, s) = mapAccumL\ \overline{(algR\ nn)}\ s
\end{aligned}$$

As Christopher Olah theorizes, the forwarding process can be elegantly described by the Haskell combinator *mapAccumLeft*[6], where a *algR nn* function will be applied sequentially to all of the input values, while a hidden state, that starts with the provided initial state and is constantly being updated, is also passed in order to produce the needed information. The *algR* function will then be responsible for receiving a RNN, an initial hidden state and an input, keeping the last two, generating a new hidden state and use it to create the new output. A diagram representing the function can be constructed like this, based on two auxiliary functions f and g that will be described shortly:

$$\begin{tikzcd}
V \times V \arrow[d, "\langle f{,}id \rangle"] \arrow[dd, bend right=60, "algR\ (RL\ lx\ ly)"'] \\
V \times (V \times V) \arrow[d, "\langle \pi_1{,}g \rangle"] \\
V \times BPR
\end{tikzcd}$$

$$\begin{aligned}
&f = useL\ lx \cdot concatM \cdot swap \\
&g = inBR \cdot \langle useL\ ly \cdot \pi_1, id \rangle
\end{aligned}$$

[6] *mapAccum* combinators can be regarded as the glueing of a *catamorphism* with a *map* on the same type

The f function is responsible for combining the hidden state and the input value into one single vector and then apply the transformations associated to the lx layer of the RNN, while g produces the $BackPropR$ structure by storing the received information along with the new output, created thanks to the application of ly to the new hidden state. The final split involving g is needed since the new hidden state must be passed to the next input value.

LEARNING Regarding the “learning” process, this one is more complicated than the one presented for normal NN. As stated in Nguyen and Wu, in order to perform a backward pass, one will compute the different gradients, starting from the end of the input and processing the sequence one step at a time, in such a way that that gradients will essentially flow backward across time steps, giving BPTT its name. The gradient regarding the final output is calculated, being then used to compute the gradients of the weight matrix and the hidden state. The last one is then copied, added to the gradient from the previous time step and modified according to the activation functions that are being used. This final gradient is then propagated in order to be used by the initial matrices as well as the next time step, to determine both of their gradients.

With this in mind, and following Krishna (2022), one can define the $learnR$ function as follows:

$$\begin{tikzcd}
(RL\ (V \to V)) \times (V \times BPR)^* \arrow[d, "\langle \pi_2 \cdot \widehat{f}, \pi_1 \rangle"] \arrow[dd, bend right=60, "learnR\ (fx, fy)"'] \\
(RL\ (V \to V)) \times (RL\ (V \to V)) \arrow[d, "\widehat{backwardR}"] \\
RL\ (V \to V)
\end{tikzcd}$$

Here, fx and fy are the derivative functions that are going to be used in order to compute the gradients of the first and second layers, respectively, of the received $RLayer$, while $backwardR$ is a function that receives two $RLayer$'s, where the first one contains the final derivative values and the other one the initial RNN (more details about this will follow shortly). The core of the algorithm is the f function that is depicted in the diagram below.

$$\begin{tikzcd}
(V \times BPR)^* \arrow[d, "f\ nn"'] \arrow[r, bend right, "outList"'] & 1 + (V \times BPR) \times (V \times BPR)^* \arrow[l, bend right, "inList"'] \arrow[d, "id + id \times (f\ nn)"] \\
V \times RL\ (V \to V) & 1 + (V \times BPR) \times (V \times RL\ (V \to V)) \arrow[l, "{[g \cdot \underline{nn}, \widehat{coalgR\ nn}]}"]
\end{tikzcd}$$

$$\begin{aligned}
g &= \langle [\underline{zero\ 1\ 1}, \pi_2 \cdot \pi_1] \cdot outL \cdot \pi_1, inRL \rangle \cdot (h\ fx \times h\ fy) \cdot outRL \\
h\ fl &= inL \cdot (id + ((i \times i) \times \underline{fl})) \cdot outL \\
i &= \widehat{zero} \cdot \langle nrows, ncols \rangle
\end{aligned}$$

As one can see, f is a list catamorphism that will generate a pair of $Values$ and $RLayer$. The first is used to store the gradient that needs to be passed to the next backpropagation, while the second will store the sums of all of the gradients that have been already calculated. When the received list is empty, the g function will be used to produce “empty” structures, while in the opposite case, $coalgR$ will be applied to the received element, as well as the previously computed derivative values, in order to calculate the updated ones. The definition of $coalgR$ follows.

$$
\begin{array}{l}
coalgR :: RLayer\ (Values \to Values) \to (Values, BackPropR) \\
\quad \to (Values, RLayer\ (Values \to Values)) \\
coalgR\ (RL\ lx\ ly)\ (o, (BPR\ y\ a'\ a\ x))\ (danext, (RL\ dlx\ dly)) = \\
\quad (danext', (RL\ dlx'\ dly')) \\
\qquad \mathbf{where}\ danext' = daraw * (f\ daraw\ lx) \\
\qquad\quad dlx' = g\ daraw\ (concatM\ (x, a))\ dlx \\
\qquad\quad dly' = g\ dy\ a'\ dly \\
\qquad\quad daraw = pointMult\ ((i\ lx)\ a')\ da \\
\qquad\quad da = (dy * (f\ dy\ ly)) + danext \\
\qquad\quad dy = pointMult\ (elementwise\ dCost\ y\ o)\ ((i\ ly)\ y) \\
\qquad\quad f\ d = [identity \cdot \underline{ncols\ d}, transpose \cdot \pi_1 \cdot \pi_1] \cdot outL \\
\qquad\quad g\ d\ v = inL \cdot (id + (h\ d\ v)) \cdot outL \\
\qquad\quad h\ d\ v = (((+(outProd\ d\ v)) \times (+d)) \times id) \\
\qquad\quad i = [\underline{id}, \pi_2] \cdot outL
\end{array}
$$

Now, regarding the previously referred $backwardR$ function, one can depict it as shown:

$$\begin{tikzcd}
RL\ (V \to V) \arrow[d, "outRL"] \arrow[ddd, bend right=60, "backwardR\ (RL\ dlx\ dly)"'] \\
(L\ (V \to V)) \times (L\ (V \to V)) \arrow[d, "(f\ (outL\ dlx) \times f\ (outL\ dly))"] \\
(L\ (V \to V)) \times (L\ (V \to V)) \arrow[d, "inRL"] \\
RL\ (V \to V)
\end{tikzcd}$$

In the diagram, dlx and dly will be the computed derivative values that are received, while the f function here defined is responsible for applying the received values into the received $RLayer$ in order to compute the updated RNN as follows:

$$
\begin{array}{l}
f\ (Right\ (d, \_)) = inL \cdot (id + (g\ d)) \cdot outL \\
f\ \_ = id \\
g\ (dw, db) = ((h\ dw \times h\ db) \times id) \\
h\ m = (m+) \cdot fmap\ ((-0.1)*) \cdot \widehat{elementwise}\ (/) \cdot \langle id, fmap\ (sqrt \cdot (+1\mathrm{e}{-8}) \cdot (\uparrow 2))\rangle
\end{array}
$$

TRAINING Regarding the "training" process, this will be rather simple, with the *trainingR* function structured as follows:

$$\begin{tikzcd}
(V^* \times V^*)^* \arrow[d, "trainingR\ nn\ fs\ s"'] \arrow[r, bend right, "outList"'] & 1 + (V^* \times V^*) \times (V^* \times V^*)^* \arrow[l, bend right, "inList"'] \arrow[d, "id+id \times (trainingR\ nn\ fs\ s)"] \\
RL\ (V \to V) & 1 + (V^* \times V^*) \times (RL\ (V \to V)) \arrow[l, "{[\underline{nn}, h]}"]
\end{tikzcd}$$

In this case, nn is the initial RNN, fs are the derivative functions that will be applied and s is the initial hidden state. Function h is responsible for updating the network whenever the received list of output/input pairs is not empty.

$$\begin{tikzcd}
(V^* \times V^*) \times (RL\ (V \to V)) \arrow[d, "\langle hr, \pi_1 \cdot \pi_1 \rangle"] \arrow[dd, bend right=60, "h"'] \\
(((RL\ (V \to V) \times V) \times V^*) \times V^*) \arrow[d, "trainR\ fs"] \\
RL\ (V \to V)
\end{tikzcd}$$

$$hr = \langle h' \cdot \pi_2, \pi_2 \cdot \pi_1 \rangle$$
$$h' = \langle id, \underline{s} \rangle$$

hr will be simply responsible for reorganizing the received information, while $trainR$ is the main mechanism of the "training" process, being responsible for using the $learnR$ and $feedR'$ functions in order to update the received network:

$$\begin{tikzcd}
(((RL\ (V \to V) \times V) \times V^*) \times V^*) \arrow[d, "\langle (\pi_1 \cdot \pi_1) \times id, \pi_2 \cdot \widehat{feedR'} \cdot \pi_1 \rangle"] \arrow[dddd, bend right=60, "trainR\ fs"'] \\
((RL\ (V \to V) \times V) \times V^*) \times BPR^* \arrow[d, "assocr"] \\
(RL\ (V \to V) \times V) \times (V^* \times BPR^*) \arrow[d, "(id \times \widehat{zip})"] \\
(RL\ (V \to V) \times V) \times (V \times BPR)^* \arrow[d, "learnR\ fs"] \\
RL\ (V \to V)
\end{tikzcd}$$

4.3 UPGRADING RNNS: BRNNS ENTER THE SCENE

RNN's are great to deal with sequential information but only consider past context when calculating an output. What if one wants to include both past and future context when determining it? In that case, the wisest choice is to use a *Bidirectional Recurrent Neural Network (BRNN)*.

BRNN's are a RNN variant that allows the processing of the information regarding past and future inputs when calculating a new output (IBM Cloud Education, 2023). This happens since the NN will generate an hidden state, at each time step, for both the "normal" order of inputs (from first given to last) as well as for the "inverted" order (from last given to first). Both these hidden states will then be used in order to create the final output, which will then be better predicted thanks to all of the received information, comprising both past and future knowledge.

In order to produce programs that use BRNN's technologies, and since they are nothing more then a RNN variation, a new data structure was created, $BRLayer$, that is really similar to $RLayer$ (see page 59). The only difference that separates these two is the existence of a third layer. In this case, the first layer will be responsible for processing the received inputs from left to right, associating to each one of them the previous hidden state created by that layer, while the second one will also process the inputs, but from right to left, associating simultaneously the layer's previously produced hidden state. The third and final layer will then be responsible for processing the two new hidden states, that were created by the other two layers, in order to create the final output. In this case, the result of each iteration will be a triple with the two hidden states and the final output. Once again, normal layers are used while defining $BRLayer$ since it is supposed to be a simple BRNN, and the created code is given in Appendix A.

FEEDING Since BRNN's are a particularization of RNN's, it is expected that the processes associated to the network's functioning are similiar. In case of the "feeding" process, only some small changes will be observed, regarding the way how information is processed. Thanks to that, $feedBR$ will be very similar to its RNN counterpart, as shown in the following diagram:

$$\begin{tikzcd}
V^* \arrow[d, "feedBR'\ (nn{,}s)"] \arrow[dd, bend right=60, "feedBR\ nn\ s"'] \\
V \times BPBR^* \arrow[d, "(id \times map\ (fst \cdot outBBR))"] \\
V \times V^*
\end{tikzcd}$$

As already stated, the only differences between $feedBR$ and $feedR$ are the auxiliary functions that are used, since the main structure is exactly the same. In this case, $feedBR'$ will continue to be responsible for creating all of the required information for future backpropagations, but this will now be stored in a new kind of data structure, $BackPropBR$ (page 59). This data structure is composed of seven different $Values$ vectors, in order to store not only the created output, but all of the intermediate information that is created by both layers responsible for processing the received inputs and hidden layers. For each layer, there will be a new hidden state, a received one and a received input. The definition of the $feedBR'$ is as follows.

$$
\begin{aligned}
&feedBR' :: (BRLayer\ (Values \to Values), Values) \to [Values]\\
&\quad \to ((Values, Values), [BackPropBR])\\
&feedBR'\ (nn, s) = f \cdot \langle l, r\rangle\\
&\quad \textbf{where}\ l = mapAccumL\ \overline{algBR\ lxl}\ s\\
&\qquad r = mapAccumR\ \overline{algBR\ lxr}\ s\\
&\qquad f = \langle \pi_1 \times \pi_1, \widehat{zipWith}\ \overline{g} \cdot (\pi_2 \times \pi_2)\rangle\\
&\qquad g = inBBR \cdot \langle useL\ ly \cdot concatM \cdot (\pi_1 \times \pi_1), id\rangle\\
&\qquad ((lxl, lxr), ly) = outBRL\ nn
\end{aligned}
$$

Once again, Olah's predictions seem to be correct, as the main engine of the forwarding process on BRNN's can be easily constructed when zipping the results obtained by applying both left and right accumulating maps. The f function will behave as a more complex zip, in order to preserve the final hidden states while using the information in both lists to create the final output for each input (by applying the final layer) and generate the respective instances of $backPropBR$ structures. The function used by the accumulating maps will be the same, $algBR$, with the only difference being the layer that is applied since there will be a specialized layer for each one of the crossings. This function is fairly simple, being responsible for producing the new hidden state for the respective crossing, while storing the information needed in the future to create the $BackPropBR$'s. There is a need to produce two copies of the new hidden state in order to preserve one of them along side the rest of the information, while enabling the other to be passed for the next iteration.

$$\begin{tikzcd}
V \times V \arrow[d, "\langle f{,}id\rangle"] \arrow[dd, bend right, "algBR\ l"'] \\
V \times (V \times V) \arrow[d, "\langle \pi_1{,}id\rangle"] \\
V \times (V \times (V \times V))
\end{tikzcd}$$

$$f = useL\ l \cdot concatM \cdot swap$$

LEARNING When approaching the "learning" process, one easily understands that it will have to be more intricate than the one used in general RNN's, since the existence of an additional layer (that works in parallel with another one) will create the need for splitting the computation of the derivatives. Despite this, the general structure of the *learnBR* function will be similar to *learnR*, cf.:

$$\begin{tikzcd}
(BRL\ (V \to V)) \times (V \times BPBR)^* \arrow[d, "\langle f, \pi_1 \rangle"] \arrow[dd, bend right=60, "learnBR\ (fx,fy)"'] \\
(BRL\ (V \to V)) \times (BRL\ (V \to V)) \arrow[d, "\widehat{backwardBR}"] \\
BRL\ (V \to V)
\end{tikzcd}$$

Similarly to its RNN's counterpart, fx is the derivative function used in order to compute the gradients of the input layers, while fy will be responsible for processing the generated hidden states, in order to compute the final output. $backwardR$ receives two $BRLayer$'s, the first one being the final derivative values and the other one the initial BRNN. The main engine of the algorithm is once again the f function, depicted below.

$$\begin{aligned}
f &= inBRL \cdot \langle l2, r2 \rangle \cdot \langle l, r \rangle \cdot (outBRL \times map\ (id \times outBBR)) \\
l &= \pi_1 \times map\ (\pi_2 \cdot \pi_2) \\
r &= \widehat{r'} \cdot assocr \cdot (((\langle id, g\ fy \rangle \cdot \pi_2) \times map\ (id \times (id \times (\pi_1 \times \pi_1)))) \\
r'\ l &= \widehat{mapAccumL\ (}coalgBR\ l) \\
l2 &= \langle l3, r3 \rangle \cdot (id \times \pi_2) \\
r2 &= \pi_1 \cdot \pi_2 \\
l3 &= i \cdot ((\pi_1 \times (map\ \pi_1)) \times (map\ \pi_1)) \\
r3 &= i \cdot ((\pi_2 \times (reverse \cdot map\ \pi_2)) \times (reverse \cdot map\ \pi_2)) \\
g\ fl &= inL \cdot (id + ((h \times h) \times \underline{fl})) \cdot outL \\
h &= \widehat{zero} \cdot \langle nrows, ncols \rangle \\
i &= \pi_2 \cdot \widehat{i'} \cdot (id \times \widehat{zip}) \cdot assocr \\
i'\ l &= (\!|\, [\langle i'', id \rangle \cdot g\ fx \cdot \underline{l}, \widehat{coalgBR'\ l}] \,|\!) \\
i'' &= [\underline{zero\ 1\ 1}, \pi_2 \cdot \pi_1] \cdot outL
\end{aligned}$$

Despite the great extension of the function's definition, which occurs mainly because of the large amount of functions used to restructure the information in the intermediate steps, f is still responsible for calculating the derivative values. This time, it will use two auxiliary functions, $coalgBR$ and $coalgBR'$, in order to achieve its goal, the first one being responsible for computing the derivative values associated with the output layer (the layer that originates the final outputs from the created hidden states), while the second one determines the derivatives associated to the other two layers (this function needs to be used twice, one for each layer, what originates a total of three different computations). These computations are done separately, since the last two are independent but the results from the first one are needed for both. The definitions of $coalgBR$ and $coalgBR'$ follows:

$$\begin{aligned}
&coalgBR :: Layer\ (Values \to Values) \to Layer\ (Values \to Values) \\
&\quad \to (Values, (Values, (Values, Values)))
\end{aligned}$$

```
    → (Layer (Values → Values), (Values, Values))
coalgBR ly dly (o, (y, a)) = (dly′, da)
  where dly′ = f dy (concatM a) dly
    da = g (dy ∗ (h dy ly))
    dy = pointMult (elementwise dCost y o) ((i ly) y)
    f d v = inL · (id + (f′ d v)) · outL
    f′ d v = (((+(outProd d v)) × (+d)) × id)
    g = ⟨g′ take, g′ drop⟩ · toList
    g′ fg = fromLists · singl · f̂g · ⟨(flip div 2) · length, id⟩
    h d = [identity · ncols d, transpose · π1 · π1] · outL
    i = [id, π2] · outL
coalgBR′ :: Layer (Values → Values) → ((Values, (Values, Values)), Values)
    → (Values, Layer (Values → Values))
    → (Values, Layer (Values → Values))
coalgBR′ l ((a′, (a, x)), da) (danext, dl) = (danext′, dl′)
  where danext′ = daraw ∗ (f daraw l)
    dl′ = g daraw (concatM (x, a)) dl
    daraw = pointMult ((i l) a′) (da + danext)
    f d = [identity · ncols d, transpose · π1 · π1] · outL
    g d v = inL · (id + (h d v)) · outL
    h d v = (((+(outProd d v)) × (+d)) × id)
    i = [id, π2] · outL
```

coalgBR receives the output layer, the derivatives already calculated for that layer, and a pair where the first element is the expected output and the second one is another pair with the produced output and the hidden state that was created in that step. This yields originating a new pair composed by the updated derivatives, that will be used in the next step, as well as the derivatives associated with the hidden state, that will be used later to compute the derivatives of the input layers. *coalgBR′* receives an input layer and two pairs: the first one contains all information produced in a forward propagation step, as well as the derivative computed by *coalgBR* for that same step; while the second one is composed by the derivatives that were already calculated. Altogether, *coalgBR′* will return the updated derivative values.

The *backwardBR* function is pretty similar to its RNN counterpart, the only difference being related to the fact that three layers are used instead of one.

$$\begin{tikzcd}
BRL\ (V \to V) \arrow[d, "outBRL"] \arrow[ddd, bend right=60, "backwardBR\ (BRL\ dlxl\ dlxr\ dly)"'] \\
((L\ (V \to V)) \times (L\ (V \to V))) \times (L\ (V \to V)) \arrow[d, "(f\ (outL\ dlxl) \times f\ (outL\ dlxr)) \times f\ (outL\ dly)"] \\
((L\ (V \to V)) \times (L\ (V \to V))) \times (L\ (V \to V)) \arrow[d, "inBRL"] \\
BRL\ (V \to V)
\end{tikzcd}$$

dlxl, *dlxr* and *dly* are the computed derivative values that are received, while the defined *f* function is exactly the same used in RNN's.

$$\begin{aligned}
&f\ (Right\ (d, \_)) = inL \cdot (id + (g\ d)) \cdot outL \\
&f\ \_ = id \\
&g\ (dw, db) = ((h\ dw \times h\ db) \times id) \\
&h\ m = (m+) \cdot fmap\ ((-0.1)*) \cdot \widehat{elementwise}\ (/) \cdot \langle id, fmap\ (sqrt \cdot (+1\mathrm{e}{-8}) \cdot (\uparrow 2)) \rangle
\end{aligned}$$

TRAINING Let us finally address the "training" process, which will be exactly the same as the one presented for the general RNN's, the only differences being the auxiliary functions that are used. Thus, *trainingBR* is defined as follows:

$$\begin{tikzcd}
(V^* \times V^*)^* \arrow[d, "trainingBR\ nn\ fs\ s"'] \arrow[r, bend right, "outList"] & 1 + (V^* \times V^*) \times (V^* \times V^*)^* \arrow[l, bend right, "inList"'] \arrow[d, "id + id \times (trainingBR\ nn\ fs\ s)"] \\
RL\ (V \to V) & 1 + (V^* \times V^*) \times (RL\ (V \to V)) \arrow[l, "{[nn,h]}"]
\end{tikzcd}$$

Once again, *nn* will be the initial BRNN, *fs* are the derivative functions that will be applied and *s* is the initial hidden state. *h* is the function responsible for updating the network whenever the received list of pairs output/input in not empty.

$$\begin{tikzcd}
(V^* \times V^*) \times (BRL\ (V \to V)) \arrow[d, "\langle hr, \pi_1 \cdot \pi_1 \rangle"] \arrow[dd, bend right=60, "h"'] \\
((BRL\ (V \to V) \times V) \times V^*) \times V^* \arrow[d, "trainBR\ fs"] \\
BRL\ (V \to V)
\end{tikzcd}$$

$$hr = \langle \langle id, \underline{s} \rangle \cdot \pi_2, \pi_2 \cdot \pi_1 \rangle$$

The function effectively responsible for the "training" process is $trainBR$, using for that $learnBR$ and $feedBR'$ functions in order to update the received network, similarly to what happens in $trainR$.

$$\begin{tikzcd}
(((BRL\ (V \to V) \times V) \times V^*) \times V^*) \arrow[d, "\langle (\pi_1 \cdot \pi_1) \times id, \pi_2 \cdot \widehat{feedBR'} \cdot \pi_1 \rangle"] \arrow[dddd, bend right=60, "trainBR\ fs"'] \\
((BRL\ (V \to V) \times V) \times V^*) \times BPBR^* \arrow[d, "assocr"] \\
(BRL\ (V \to V) \times V) \times (V^* \times BPBR^*) \arrow[d, "(id \times \widehat{zip})"] \\
(BRL\ (V \to V) \times V) \times (V \times BPBR)^* \arrow[d, "learnBR\ fs"] \\
BRL\ (V \to V)
\end{tikzcd}$$

4.4 TESTING

Some tests were defined in order to understand if the created NN work and are efficient. In a first approach, the general NN was tested and after it the RNN and BRNN structures.

GENERAL NN The tests for this network are based on the work given in Lynn (2015), which in turn has been inspired by (Nielsen, 2019). The latter has created an online book dedicated to teaching about NN and DL, where many practical examples are provided to help understand the concepts. One such example is a NN that is able to recognize handwritten digits, and it is this one that is going to be used below, following the footsteps of the first referred author. The generated network will be responsible for interpreting datasets with digit representations, provided by the MNIST database of handwritten digits[7] and map them to the corresponding digit. It should be mentioned that the training and testing datasets are different, so that the liability of the network for any given input can be tested.

To make for a larger variety of tests, three different functions were created (please see Appendix F, page 71): the first, $getABrain$, mimics the test provided by the blog that is being used, allowing the tester to select a digit by its position, so that the program guesses what digit is represented, followed by a try to guess all of the digits stored in the dataset; the second, $allDigits$, simply tries to guess all of the digits in the dataset; the third and last one, $specificDigit$, allows the tester to simply select one representation to be predicted.

In order to better understand the results that were obtained, the ones provided by the second testing function will be more under focus, since these approach more significant values, thanks to the large number of guesses that were made.

[7] Cf. https://yann.lecun.com/exdb/mnist/.

It turns out that the percentage of correct guesses revolves around the same presented in Lynn (2015), which is between 75% and 90%. Despite not being as high as in many programs currently available, one still tends to believe that it is an honorable value, thanks to the simplicity of the network in matters of optimization and modernization by using more advanced techniques. Recall that the added bonus is a much more structured solution, thanks to the usage of FM and algebraic rules, leading one to believe that future enhancement of this program will be easier thanks to its better structured architecture.

RNN'S AND BRNN'S A different approach was taken regarding the testing of the RNN and BRNN implementations. In this case, the base example that was used is that provided by Patil (2022): a program capable of predicting the mean temperature in a city for the next day when provided with information regarding the climate conditions there in past days. In this case, the used dataset comes from Sumanthvrao (2019) and it provides the information about mean temperature, humidity, wind speed and mean pressure in the city of Delhi, India, between 1st January 2013 and 24th April 2017.

The example that was created using the constructed RNN and BRNN structures follows the ideas shown in Patil (2022). However, the results that were obtained are not satisfatory. The truth is that thanks to the simplicity presented on the created networks, this kind of results were predictable. In fact, and as discussed by IBM Cloud Education (2023), there are some different possibilities to why a RNN do not work as expected. A most common problems is related to vanishing or exploding gradients, that occur when running the BPTT process. Since in this process the errors are summed up in each time step, given the fact that parameters are shared across each layer, it is normal for the gradient, which is the slope of the loss function along the error curve, to be heavily affected. If the gradient becomes too small, it continues to decrease, updating the weight parameters until they become insignificant, and the program ceases to learn. On the other hand, when the gradient gets too large, one has to deal with the exploding gradient problem, since the model weights will grow enormously, eventually reaching NaN.

Another issue that should be mentioned is the long-term dependency problem, which questions how to deal with situations where a previous state that has influence in the current prediction is not present in the recent past context. In this case, the network is unlikely to be capable of correctly predicting the current state.

New tools have been developed to help solve problems of this kind, like the *Long Short Term Memory (LSTM)*'s, which are a special kind of RNN that have "cells" inside the hidden layers with special gates in them that allow for better control of the information flow, contributing for the optimization of the predictions that are done — please see the next chapter.

As Christopher Olah concludes, many of the remarkable results that are being obtained in this field were achieved thanks to LSTM study, so it is safe to assume that the poor results that were obtained when testing both the implementations of the RNN and the BRNN (recall that the latter is nothing more is than a variation of the first) may be related to the much too simple architecture of both models.

4.5 SUMMARY

The main conclusion of this chapter is that Olah's hypothesis is spot on regarding the construction of the algorithms in which DL is based upon, at least regarding those approached in this study. The constant presence of higher order functions, as well as the capability of depicting algorithm structures in simple mathematical diagramatic representations are the living proof that FP can effectively be an added value in the study of ML, making it possibly easier to explain some of the processes that reside underneath this complex paradigm.

5

TOWARDS ENCODING AND DECODING

As happens in all areas of knowledge, DL continues to grow and evolve. The appearance of new technologies is not only a constant, but even a necessity in order to continue to push the limits of what it is able to do. For example, regarding RNN's, new techniques are being constantly explored so that the execution of these programs becomes as efficient as possible, since, as we have seen before, the basic algorithms that are used will not always produce the expected results.

One such technique is called LSTM and, as Olah (2015b) explains, it consists on a special kind of RNN that is able to learn long-term dependencies, having been designed to avoid the problems created by them since they normally remember information for long periods of time. Their internal structure has a variable number of neural network layers, depending on the implementation that is being adopted, but the essential points are all the same: work in such a way that a cell state exists and is able to store only the most important information, selected by the gates, which are structures that filter the received information in order to decide what must go through. A particularly relevant type of LSTM is the *Gated Recurrent Unit (GRU)*, that merges two of the gates that are used in a traditional LSTM, as well as the cell and hidden states, so that the resultant model is simpler than the standard one.

Thanks to this progress, RNN's can be applied into increasingly difficult problems, like encoding and decoding, for example. As was already mentioned in this work, Olah sees an Encoding RNN, or Encoder, as a fold, while Generating RNN's, or Decoder, are seen as an unfold, so these structures do have the potential of further exploiting the hypothesis that one defends in this thesis. But what is an encoding-decoding system?

5.1 ENCODERS AND DECODERS

Following the work developed by Bahdanau et al. (2015), the authors define an encoder as an NN that reads and encodes a source sentence into a fixed-length vector, while the decoder will be able to output a translation from the produced encoded vector. If one applies this theory into a language translator, the encoder will receive a sentence, written in a given language, while the decoder will output the sentence translated into another language. For the system to work with many languages, an encoder and a decoder for each language will be needed, or a language-specific encoder that is applied to each sentence in order to compare the produced outputs. Each encoder-decoder system, which is nothing but an encoder-decoder for a language pair, is trained so that the probability of a correct translation is maximized.

Despite the effectiveness of this approach, the authors claim that it may have a potential issue regarding the capability of compressing all the needed information into a fixed-length vector, for example when receiving too long sentences, namely those that are longer than the ones present in the training set that is used to initialize the system. To address this issue, they propose a new extension of the encoder-decoder model that learns to align and translate jointly, since whenever the model generates a word in a translation it searches for a set of positions in the input sentence where the most relevant information is. The target word is then predicted based on the context vectors that are associated with these positions and all of the previously generated words. Also, this approach is able to encode the input sentence into a sequence of vectors, instead of a fixed-length vector, and will only use a subset of those that it sees as fit for a given time at the decoding process, disabling the need for compressing all the information into only one vector, and allowing to obtain better results when processing longer sentences.

Another very important point that is referred in the article is relative to facing translation from a probabilistic point-of-view. In this case, translation will be similar to finding an output sentence that maximizes the conditional probability that relates it to the input sentence. In the particular case of translation using NN, a paramenterized model will be trained with a training set, so that the conditional distribution is learned by it, making the model able to generate an output sentence for any input sentence in such a way that the conditional probability is maximized. With this in mind, if one wants to further explore the connections between NN and FP, another concept can be added to this theory, monads. To be more precise, the distribution monad (Erwig and Kollmannsberger, 2006; Oliveira and Miraldo, 2016). One can start to hypothesize that, given the probabilistic nature of encoder-decoder systems, the usage of the distribution monad can make things easier when it comes to deal with all of the possible values, since one will have to create a general way to process each value, while letting the monad "do it's magic" and transforming the entire structure. To be more precise, applying this monad will be useful on the decoder side of the program, since it is here that the conditional distribution is created.

While reflecting about all of this, a pseudo-prototype of the decoder that is presented to us in the paper mentioned above was created, in order to prove not only that the concepts that were approached in the previous sections continue to be valid, but also that the use of the distribution monad can bring great benefits to the cause. It is important to refer that only the forward passing on the decoder part of the system will be analyzed.

5.2 PROOF OF CONCEPT

As presented by Bahdanau et al. (2015), in the Encoder-Decoder framework, the encoder is responsible for processing a received sentence, which is composed by a sequence of vectors $x = (x_1, ..., x_{T_x})$, into a vector c. The commonly used approach revolves around the usage of a RNN to create a hidden state at a given time t ($h_t \in \mathbb{R}^n$), later using it along with some f and q nonlinear functions to create c as follows:

$$h_t = f(x_t, h_{t-1}) c = q(h_1, ..., h_{T_x})$$

The decoder, on the other hand, is trained to predict the next word $y_{t'}$ given the context vector c as well as the previously calculated words $\{y_1, ..., y_{t-1}\}$, which means that it will define a probability over the translation **y** (where **y** = $(y_1, ..., y_{T_y})$) by decomposing the joint probability into the ordered conditionals:

$$p(\mathbf{y}) = \prod_{t=1}^{T} p(y_t \mid \{y_1, ..., y_{t-1}\}, c)$$

If a RNN is used, then each conditional probability is defined like this:

$$p(y_t \mid \{y_1, ..., y_{t-1}\}, c) = g(y_{t-1}, s_t, c)$$

being g a nonlinear, potentially multilayered, function used to determine the probability of y_t, and s_t the hidden state of the RNN.

Now, following the approach presented by the authors, and considering that the output vector produced by the encoder is such that it contains all the annotations created for each input word ($\{h_1, ..., h_{T_x}\}$), which will later be used to create the context vector in each timestamp, one can start to define how the decoder will be structured. For that, the following definition was made.

$$\textbf{data}\ \mathit{CGRU}\ a = \mathit{CGRU}\ (\mathit{Layer}\ a)\ (\mathit{Layer}\ a)\ (\mathit{Layer}\ a)$$
$$\textbf{data}\ \mathit{Decoder}\ a = \mathit{Dec}\ (\mathit{Weights}, \mathit{Layer}\ a)\ (\mathit{CGRU}\ a)\ (\mathit{Layer}\ a)\ (\mathit{Layer}\ a)$$

The first data type will be used to store the information needed for a GRU that deals with a context vector to work: the first layer is the update gate layer, the second one is the intermediate gate layer and the third one is the final gate layer.

The second data type will be the decoder itself, essentially composed by a simple RNN that uses GRU technology capable of dealing with a context vector. The first parameter is a pair that is applied in order to transform the received inputs and hidden state into a context vector; the second one is the needed information for the GRU operations, that process the created context vector and the received hidden state, in order to originate a new hidden state; the third one is a layer that processes the context vector and the hidden state, using them to originate an intermediate state, needed for the creation of the output; and the fourth one is a layer responsible for processing the intermediate state in order to create a new output.

With all of this, one can start to define how the forwarding process (or the "feeding" process, as was previously written) can be defined. Thanks to Olah's hypothesis, that states that decoders are nothing more than unfolds, as well as the possibility of using the distribution monad, one can start to theorize that the *feed* function will be easily defined by resorting to the monadic unfold function. With this in mind, one can define *feedDec* as follows:

$$\mathit{feedDec} :: \mathit{Decoder}\ (\mathit{Values} \to \mathit{Values}) \to [\mathit{Values}] \to (\mathit{Int}, (\mathit{Values}, \mathit{Values})) \to \mathit{Dist}\ [\mathit{Values}]$$
$$\mathit{feedDec}\ nn\ x = \mathit{unfoldM}\ (\mathit{algDec}\ nn\ x)$$

Here, nn will be the decoder, while x is the list of annotations produced by the encoder. The initial state that will be used by the unfold will then be a triple composed by the maximum number of iterations to be executed (present to ensure that the program does not run indefinitely), the initial hidden state and the "initial output" (this last parameter is needed since every iteration will take into account the previously produced output, but will normally be zero, given that no other value makes real sense in this context). The *algDec* function will then be responsible for performing each iteration of the *unfoldM*, creating a new output on each one of them that will be added to the final probability distribution. With this in mind, and taking into account the fact that *unfoldM* requires for the performed function to return a *Maybe* inside the monadic wrapper, in order to determine when to stop the execution of the program, *algDec* can be defined like this:

$$
\begin{array}{l}
algDec :: Decoder\ (Values \rightarrow Values) \rightarrow [Values] \rightarrow (Int, (Values, Values)) \\
\quad \rightarrow Dist\ (Maybe\ (Values, (Int, (Values, Values)))) \\
algDec\ nn\ x = Cp.cond\ (\widehat{\vee} \cdot ((\leqslant 0) \times h)) \\
\quad (return \cdot \underline{Nothing}) \\
\quad (\widehat{g} \cdot ((flip\ (-)\ 1) \times (algDec'\ nn \cdot \langle id, f \rangle))) \\
\quad \mathbf{where}\, f = context\ la \cdot flip\ (,)\ x \cdot \pi_1 \\
\qquad g\ n = fmap\ (Just \cdot (id \times ((,)\ n))) \\
\qquad h = (>0) \cdot \widehat{getElem}\ 1 \cdot \langle ncols, id \rangle \cdot \pi_2 \\
\qquad la = \pi_1\ (outDec\ nn)
\end{array}
$$

This function will, in its core, be a conditional, since in every execution it will need to determine if it is time to stop, or if it is possible to run another iteration. For that it will analyze the received state, to see if either the number of iterations left has reached zero or the previously produced output was an "end-of-file" (one will consider that, in the *Values* matrix that stores the outputs, the last position is reserved for this special case). In case either of these conditions is verified, the function will return a *Nothing* value, that will be interpreted by the *unfoldM* function as a sign to stop running the iterations. If negative, then the function will proceed to produce a new output value. First of all, the context vector has to be determined, and for this the *context* function is used.

$$
\begin{tikzcd}
V \times V^* \arrow[d, "\langle g \cdot \widehat{f}\ i{,}\pi_2 \rangle"] \arrow[dddd, bend right, "context\ (w{,}l)"'] \\
Float^* \times V^* \arrow[d, "\widehat{zipWith}\ h"] \\
V^* \arrow[d, "\langle head{,}tail \rangle"] \\
V \times V^* \arrow[d, "\widehat{foldl\ (+)}"] \\
V
\end{tikzcd}
$$

$$
\begin{array}{l}
f\, f'\ x = map\ (f' \cdot (,)\ x) \\
g = \widehat{f}\ j \cdot \langle sum, id \rangle
\end{array}
$$

$$
\begin{aligned}
&h\ x = \mathit{fmap}\ (x*)\\
&i = \mathit{exp} \cdot \mathit{getElem}\ 1\ 1 \cdot (*w) \cdot \mathit{useL}\ l \cdot \mathit{concatM}\\
&j = \widehat{/} \cdot \mathit{swap}
\end{aligned}
$$

This function executes the operations referring to the creation of the context vector that are shown in Bahdanau et al. (2015). It receives all the annotations provided by the encoder, as well as the hidden state that was created in the previous iteration by the decoder, and combines them using the received layer l, along with the weights matrix w (which is, in fact, a column vector, so that when it is multiplied by the result of the application of l, the obtained product is a matrix with only one element, that can be seen as a scalar value) and other mathematical operations.

Now, returning to the definition of $algDec$, one needs to lean over the structure of the auxiliar function $algDec'$, since it will be there that the main operations of the "feeding" process will happen.

$$\begin{tikzcd}
(V \times V) \times V \arrow[d, "\langle \mathit{algDecl}\ \mathit{cgru}, \mathit{algDecr}\ \mathit{lt}\ \mathit{ly} \rangle"] \arrow[dddd, bend right=60, "\mathit{algDec'}\ (\mathit{Dec}\ \_\ \mathit{cgru}\ \mathit{lt}\ \mathit{ly})"'] \\
V \times (V \times \mathit{Float})^* \arrow[d, "\widehat{f}"] \\
((V \times V) \times \mathit{Float})^* \arrow[d, "\mathit{map}\ (\langle \pi_2, \mathit{id} \rangle \times \mathit{id})"] \\
((V \times (V \times V)) \times \mathit{Float})^* \arrow[d, "\mathit{mkD}"] \\
\mathit{Dist}\ (V \times (V \times V))
\end{tikzcd}$$

$$f\ s = \mathit{map}\ ((,)\ s \times \mathit{id})$$

As one can see, the main engines of this function will be its auxiliary functions *algDecl* and *algDecr*, since the rest of the definition is mainly responsible for restructuring the data structure in order to originate a probability distribution in the end[1]. Thanks to this, one will start to analyze how the *algDecl* function works.

$$\begin{tikzcd}
(V \times V) \times V \arrow[d, "\langle concatM \cdot (concatM \times id), id \rangle"] \arrow[ddd, bend right=60, "algDecl\ gru"'] \\
V \times ((V \times V) \times V) \arrow[d, "\langle id \times (\pi_1 \cdot \pi_1), id \rangle"] \\
(V \times V) \times (V \times ((V \times V) \times V)) \arrow[d, "f\ (outCGRU\ gru)"] \\
V
\end{tikzcd}$$

$$\begin{aligned}
&f\ (lu, (li, lf)) = i \cdot ((useL\ lu \times id) \times (useL\ lf \cdot concatM \cdot (h \times id) \cdot assocl \cdot (useL\ li \times id))) \\
&h = concatM \cdot (upm \times id) \cdot assocl \\
&i = \widehat{+} \cdot (upm \times upm) \cdot \langle (fmap\ (1-) \times id) \cdot \pi_1, \pi_1 \times id \rangle \\
&upm = \widehat{pointMult}
\end{aligned}$$

The *algDecl* function is responsible for determining the new hidden state. This can be done separately from the computation of the final distribution values since it will not be needed to define them, as we will see next. So, in order to compute the new hidden state (s_i), the function will receive a GRU as well as a triple with the previously produced hidden state (s_{i-1}) and output (y_{i-1}), along with the context vector (c_i) for the current timestamp. This information will then be reorganized in order to apply the transformation associated with the GRU usage, described in both Olah (2014) and Bahdanau et al. (2015):

$$\begin{aligned}
s_i &= (1 - z_i) \times s_{i-1} + z_i \times s'_i \\
s'_i &= tanh(WEy_{i-1} + U[r_i \times s_{i-1}] + Cc_i) \\
z_i &= \sigma(W_z E y_{i-1} + U_z s_{i-1} + C_z c_i) \\
r_i &= \sigma(W_r E y_{i-1} + U_r s_{i-1} + C_r c_i)
\end{aligned}$$

The f function will be responsible for applying these transformations, with each one of the layers provided by *gru* being used in order to achieve this: li will be used to determine r_i, followed by the application of lf, since s'_i depends on r_i; lu can be applied in parallel since it will be used to compute z_i and this one is not dependent on the first two. Once all these variables are determined, the function i is applied to determine the new hidden state s_i.

[1] The main reason for this restructuring lays on the nature of the monadic unfold, that requires that each element present in the next iterations receives the produced hidden state; thanks to this, the list of pairs output/probability needs to be mapped, in order to contain all the required information, before being converted into a distribution.

Next, let us see how *algDecr* works.

$$\begin{tikzcd}
(V \times V) \times V \arrow[d, "concatM \cdot (concatM \times id)"] \arrow[dddd, bend right=60, "algDecr\ lt\ ly"'] \\
V \arrow[d, "toList \cdot useL\ lt"] \\
Float^{*} \arrow[d, "g \cdot f"] \\
Float^{*} \arrow[d, "rearrange"] \\
(V \times Float)^{*}
\end{tikzcd}$$

$$\begin{array}{l}
f\ (x : y : xs) = (max\ x\ y) : (f\ xs) \\
f\ \_ = [\,] \\
g = toList \cdot useL\ ly \cdot fromLists \cdot singl \\
rearrange = filter\ ((>0) \cdot snd) \cdot g \cdot f \\
\quad \textbf{where}\ f\ l = map\ \langle \underline{length\ l}, /\,(sum\ l)\rangle\ l \\
\qquad g = map\ ((\widehat{h} \times id) \cdot assocl) \cdot zip\ [0\,..] \\
\qquad h\ p = fromLists \cdot singl \cdot \widehat{+\!\!+} \cdot (flip\ (+\!\!+)\ [1] \times tail) \cdot splitAt\ p \cdot flip\ replicate\ 0
\end{array}$$

Function *algDecr* is responsible for creating a list of pairs containing all the possible outputs as well as their probability (the probability of the designed value being the correct output for that timestep). As previously mentioned, this function will not need to use the hidden state that was produced in this iteration, instead it will receive two layers, lt and ly, along with the same triple that *algDecl* receives, the one containing (s_{i-1}), (y_{i-1}) and (c_i). Once again, let us take a look on the calculations that reside in this code, as proposed in Bahdanau et al. (2015).

$$\begin{aligned}
p(y_i \mid s_{i-1}, y_{i-1}, c_i) &\propto exp(y_i^{\top} W_o t_i) \\
t_i &= [max\{t'_{i,2j-1}, t'_{i,2j}\}]^{\top}_{j=1,\dots,l} \\
t'_i &= U_o s_{i-1} + V_o E y_{i-1} + C_o c_i
\end{aligned}$$

With this, it is easier to understand that lt will be used to calculate t'_i, while f is used to determine t_i. After that, ly is then applied to compute the value of $exp(W_o t_i)$ (the multiplication with the tranposed of y_i is only there to extract the probability value for that specific output, but since we want the full distribution now, we do not have to perform it). After all of this, *rearrange* will be applied in order to transform the list of probabilities into a list of pairs which associates the normalized probability to the position in which it was stored, now displayed as a *Value*.

5.3 SUMMARY

The main aim of this chapter was to show how the approach developed in the previous chapters evolves towards explaining the important field of *encoding* and *decoding* information in neural networks. If it were somewhat to be expected that the small scale approach of the previous chapters should expand to more sophisticated work, this chapter comes to prove it by setting up a basis for what a formal approach to a large scale encode-decode model could be.

The main new ingredient of such a basis is the use of monads to capture the probabilistic side of such models. Monads are indeed a *must* in any FM portfolio nowadays (Oliveira and Miraldo, 2016) and in modern functional programmimg. Altogether, the starting proposition that ML will benefit greatly from FM and FP research gets even clearer.

6

CONCLUSIONS AND FUTURE WORK

As shown throughout this dissertation, ML is inherently functional despite all of its complexity. The truth is that the FP paradigm is in fact adequate for taming such complexity, thanks to all of the simplifications that it can produce, allowing for a better understanding of what is being done by the models themselves. However, bulky functional code alone does not unveil the compositional structure of the various processes that, altogether, perform the overall task. Something else was needed in order to achieve the AI explainability that was presented in this dissertation, like expressing ML algorithmic processes by higher-order functional combinators, inspired initially by the research that has stemmed from Christopher Olah's pioneering blog, while later deepening it thanks to Nguyen and Wu's paper. Moreover, diagrams were used to depict the flow of information that is so essential to NN's, allowing for a better understanding of how data are captured and modified inside the models' layers.

This work is expected to open the doors to reasoning, proofs and calculational development of NN, allowing for more formal approaches in the field. The results obtained give evidence of Olah's hypothesis that strongly typed FP can play an important role in XAI.

Other hypotheses can be thought of, related to some elements that were present on the creation of this dissertation. For example, the work promoted by Nguyen and Wu on generic fixpoints, that was used to generate the data structures responsible for dealing with general NN let us believe that it may be possible to adapt other data structures in order to create more efficient programs. Thanks to fix point adaptative nature, one can believe that it may be possible to use them to model other more complex structures, like trees, that may promote the creation of a more efficient program.

On a personal register, I can but state how much I learned about ML and all of the branches of knowledge that come from it. Being it a topic that is hotter and hotter by the day (cf. articles like the one written by Spinney (2022), it witnessed an even bigger growth in the past months thanks to the creation of ChatGPT. Thanks to all of this, it is more and more important to understand these concepts, to know how they work and to keep updated on the advances that are promoted in this field.

Besides all of this, the knowledge that I obtained regarding the applicability of FM and algebraic techniques into real life problems is of the utmost importance, since it is not only an example of the usability and usefulness of these subjects, but also plain evidence that they can be effectively applied to the most varied fields of computer science. The capacity of developing a critical way of thinking that enables the need for searching for the most correct and valid solutions at all times is not only relevant, but necessary, specially when working in fields that are still under construction, like the case of ML and NN.

FUTURE WORK It is safe to say that this dissertation revolved around creating the basis for a more specific research to be done in the future. A lot can still be done in the fields of ML and FP, so one can only imagine what they can originate together.

At first glance, completing the experimental testing that was developed in this project seems to be the first task to be undertaken. The tests that were done on the created general NN seem to be very superficial, so it is possible that heavier and more complex ones need to be made in order to verify the true capabilities of the proposed architecture. On the other hand, thanks to the unsatisfying results that were obtained when testing the RNN and BRNN models, it is logical to say that these structures need some refactoring. As it was already stated, many new models have emerged that help obtaining better results, like the LSTM that was approached in Chapter 5 of this thesis, so implementing and testing them seems to be the way to go next.

Other more complex tasks need to be done, like creating more intricate models that better resemble real life problems. The work developed in Chapter 5 is still in an embrionary phase and it would be very interesting to be deepened up, since it relates NN and FP by using a varied set of tools, like monads, further embracing the idea that these two areas have more to win than to lose if they work together in a pragmatic and structured way.

BIBLIOGRAPHY

Samson Abramsky. Nothing will come of everything: Software towers and quantum towers, 2022. Department of Computer Science, UCL.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1409.0473.

R.S. Bird and O. de Moor. *Algebra of programming*. Prentice Hall International series in computer science. Prentice Hall, 1997. ISBN 978-0-13-507245-5.

M. Erwig and S. Kollmannsberger. Functional pearls: Probabilistic functional programming in Haskell. *J. Funct. Program.*, 16:21–34, January 2006.

David Gunning, Mark Stefik, Jaesik Choi, Timothy Miller, Simone Stumpf, and Guang-Zhong Yang. XAI - explainable artificial intelligence. *Sci. Robotics*, 4(37), 2019. doi: 10.1126/scirobotics.aay7120. URL https://doi.org/10.1126/scirobotics.aay7120.

IBM Cloud Education. What is supervised learning?, 2020a. Site: https://www.ibm.com/cloud/learn/supervised-learning, last read: July 13, 2023.

IBM Cloud Education. What is unsupervised learning?, 2020b. Site: https://www.ibm.com/cloud/learn/unsupervised-learning, last read: July 13, 2023.

IBM Cloud Education. What is recurrent neural networks?, 2023. Site: https://www.ibm.com/topics/recurrent-neural-networks, last read: July 13, 2023.

Neeraj Krishna. Backpropagation in rnn explained, 2022. Site: https://towardsdatascience.com/backpropagation-in-rnn-explained-bdf853b4e1c2, last read: July 13, 2023.

Ben Lynn. Get a brain, 2015. Site: https://crypto.stanford.edu/~blynn/haskell/brain.html, last read: July 13, 2023.

Shiona McCallum. Chatgpt banned in italy over privacy concerns, 2023a. Site: https://www.bbc.com/news/technology-65139406, last read: July 13, 2023.

Shiona McCallum. Chatgpt accessible again in italy, 2023b. Site: https://www.bbc.com/news/technology-65431914, last read: July 13, 2023.

Bibliography

Tom Mitchell. *Machine Learning*. McGraw Hill, 1997. ISBN 0-07-042807-7. OCLC 36417892.

Minh Nguyen and Nicolas Wu. Folding over neural networks. *CoRR*, abs/2207.01090, 2022. doi: 10.48550/arXiv.2207.01090. URL https://doi.org/10.48550/arXiv.2207.01090.

Michael Nielsen. Neural networks and deep learning, 2019. Site: http://neuralnetworksanddeeplearning.com/, last read: July 13, 2023.

C. Olah. Neural networks, manifolds, and topology, 2014. Blog: http://colah.github.io/posts/2014-03-NN-Manifolds-Topology/, last read: July 13, 2023.

C. Olah. Neural networks, types, and functional programming, 2015a. Blog: http://colah.github.io/posts/2015-09-NN-Types-FP/, last read: July 13, 2023.

C. Olah. Understanding lstm network, 2015b. Blog: http://colah.github.io/posts/2015-08-Understanding-LSTMs/, last read: July 13, 2023.

J.N. Oliveira. Transforming Data by Calculation. In *GTTSE'07*, volume 5235 of *LNCS*, pages 134–195. Springer-Verlag, 2008.

J.N. Oliveira. Program Design by Calculation, 2019. Draft of textbook in preparation, current version: October 2019. Informatics Department, University of Minho (PDF).

J.N. Oliveira and V.C. Miraldo. "Keep definition, change category" — a practical approach to state-based system calculi. *JLAMP*, 85(4):449–474, 2016. (DOI).

J.N. Oliveira and C.J. Rodrigues. Pointfree Factorization of Operation Refinement. In *FM'06*, volume 4085 of *Lecture Notes in Computer Science*, pages 236–251. Springer-Verlag, 2006. .

OpenAI. Introducing chatgpt, 2022. Site: https://openai.com/blog/chatgpt, last read: July 13, 2023.

Sanmit Patil. Delhi mean temperature lstm, 2022. Site: https://www.kaggle.com/code/sanmitpatil/delhi-mean-temperature-lstm, last read: July 13, 2023.

B. Penkovsky. 10 days of grad: Deep learning from the first principles, 2020. Blog: https://penkovsky.com/neural-networks/, last read: July 13, 2023.

Jamie Ranger. Who is afraid of chatgpt? the political problem of ai, 2023. Site: https://blog.politics.ox.ac.uk/who-is-afraid-of-chatgpt-the-political-problem-of-ai/, last read: July 13, 2023.

Warren S. Sarle. Ai faq/neural nets, 2002. Site: http://www.faqs.org/faqs/ai-faq/neural-nets/, last read: July 13, 2023.

Bibliography

Laura Spinney. Are we witnessing the dawn of post-theory science?, 2022. The Guardian, Sun 9 Jan 2022 09.00 GMT. Last read:July 13, 2023.

Sumanthvrao. Daily climate time series data, 2019. Site: https://www.kaggle.com/datasets/sumanthvrao/daily-climate-time-series-data, last read: July 13, 2023.

Twelverays. Benefits of chatgpt, 2023. Site: https://twelverays.agency/blog/benefits-of-chatgpt, last read: July 13, 2023.

Zhi-Hua Zhou. *Machine Learning*. Springer, 2021. ISBN 978-981-15-1966-6. doi: 10.1007/978-981-15-1967-3. URL https://doi.org/10.1007/978-981-15-1967-3.

Part I

APPENDICES

A

DATA STRUCTURES

Basic types:

```
data Layer a = IL
             | DL Weights Biases a
             deriving Functor
data RLayer a = RL (Layer a) (Layer a)
data BRLayer a = BRL (Layer a) (Layer a) (Layer a)
data BackProp = BP Weights Deltas Values [Values]
data BackPropR = BPR Values Values Values Values
data BackPropBR = BPBR Values Values Values Values Values Values Values
type Biases  = Matrix Float
type Deltas  = Maybe (Matrix Float)
type Values  = Matrix Float
type Weights = Matrix Float
```

Basic Functions for Layer usage:

```
inL :: Either () ((Weights, Biases), a) → Layer a
inL (Left _) = IL
inL (Right ((w, b), a)) = DL w b a
outL :: Layer a → Either () ((Weights, Biases), a)
outL (IL) = Left ()
outL (DL w b a) = Right ((w, b), a)
inRL :: (Layer a, Layer a) → RLayer a
inRL (lx, ly) = RL lx ly
outRL :: RLayer a → (Layer a, Layer a)
outRL (RL lx ly) = (lx, ly)
inBRL :: ((Layer a, Layer a), Layer a) → BRLayer a
inBRL ((lxl, lxr), ly) = BRL lxl lxr ly
```

```
outBRL :: BRLayer a → ((Layer a, Layer a), Layer a)
outBRL (BRL lxl lxr ly) = ((lxl, lxr), ly)

inBR :: (Values, (Values, (Values, Values))) → BackPropR
inBR (y, (a', (a, x))) = BPR y a' a x

outBR :: BackPropR → (Values, (Values, (Values, Values)))
outBR (BPR y a' a x) = (y, (a', (a, x)))

inBBR :: (Values, ((Values, (Values, Values)), (Values, (Values, Values)))) → BackPropBR
inBBR (y, ((al', (al, xl)), (ar', (ar, xr)))) = BPBR y al' al xl ar' ar xr

outBBR :: BackPropBR → (Values, ((Values, (Values, Values)), (Values, (Values, Values))))
outBBR (BPBR y al' al xl ar' ar xr) = (y, ((al', (al, xl)), (ar', (ar, xr))))
```

B

NEURAL NETWORK CODE

Initializing a NN:

$$
\begin{array}{l}
\mathit{newBrain} :: [\mathit{Int}] \rightarrow \mathit{IO}\ (\mathit{Fix\ Layer}) \\
\mathit{newBrain} = \mathit{newBrainAux}\ (\mathit{inF}\ (\mathit{inL}\ (i_1\ ()))) \\
\mathit{newBrainAux} :: \mathit{Fix\ Layer} \rightarrow [\mathit{Int}] \rightarrow \mathit{IO}\ (\mathit{Fix\ Layer}) \\
\mathit{newBrainAux}\ l\ (x : y : \mathit{xs}) = \\
\quad \mathbf{do}\ w \leftarrow \mathit{fmap\ fromLists}\ (\mathit{replicateM}\ y\ (\mathit{replicateM}\ x\ (\mathit{gauss}\ 0.01))) \\
\qquad \mathit{newBrainAux}\ (\mathit{inF}\ (\mathit{inL}\ (i_2\ ((w,(f\ y)),l))))\ (y : \mathit{xs}) \\
\qquad\quad \mathbf{where}\ f = \mathit{fromLists} \cdot \mathit{singl} \cdot \mathit{flip\ replicate}\ 1 \\
\mathit{newBrainAux}\ l\ \_ = \mathit{return}\ l
\end{array}
$$

Feeding process:

$$
\begin{array}{l}
\mathit{feed} :: \mathit{Fix\ Layer} \rightarrow \mathit{Values} \rightarrow \mathit{Values} \\
\mathit{feed}\ \mathit{nn} = \mathit{head} \cdot \mathit{feed}' \cdot (,)\ \mathit{nn} \\
\mathit{feed}' :: (\mathit{Fix\ Layer}, \mathit{Values}) \rightarrow [\mathit{Values}] \\
\mathit{feed}' = \mathit{applyR} \cdot ((\!|\mathit{alg}|\!) \times \mathit{id}) \\
\mathit{alg} :: \mathit{Layer}\ (\mathit{Values} \rightarrow [\mathit{Values}]) \rightarrow (\mathit{Values} \rightarrow [\mathit{Values}]) \\
\mathit{alg} = [\underline{\mathit{singl},}\widehat{\cdot} \cdot (f \times \mathit{id})] \cdot \mathit{outL} \\
\quad \mathbf{where}\ f\ (w,b) = (\lambda \mathit{vs} \rightarrow \mathbf{let}\ v = g\ ((w,\mathit{vs}),b)\ \mathbf{in}\ (v : \mathit{vs})) \\
\qquad g = \mathit{fmap\ relu} \cdot \widehat{+} \cdot (h \times \mathit{id}) \\
\qquad h = \mathit{transpose} \cdot \widehat{*} \cdot (\mathit{id} \times \mathit{transpose} \cdot \mathit{head})
\end{array}
$$

Learning process:

$$
\begin{array}{l}
\mathit{learn} :: (\mathit{Fix\ Layer}, \mathit{BackProp}) \rightarrow \mathit{Fix\ Layer} \\
\mathit{learn} = [\!(\mathit{coalg})\!] \\
\mathit{coalg} :: (\mathit{Fix\ Layer}, \mathit{BackProp}) \rightarrow \mathit{Layer}\ (\mathit{Fix\ Layer}, \mathit{BackProp}) \\
\mathit{coalg} = \mathit{inL} \cdot ((!) + g) \cdot \mathit{distl} \cdot ((\mathit{outL} \cdot \mathit{outF}) \times \mathit{id}) \\
\quad \mathbf{where}\ g = k \cdot j \cdot i \cdot h
\end{array}
$$

$$\begin{array}{l}
\qquad\quad h = \langle \pi_1 \times id, \langle (\pi_1 \cdot \pi_1) \times id, \pi_2 \cdot \pi_1 \rangle \rangle \\
\qquad\quad i = \widehat{backward} \times id \\
\qquad\quad j = assocr \\
\qquad\quad k = id \times (swap \cdot (bp \times id) \cdot assocl) \\
\qquad\quad bp\ (d, (w, (BP\ \_\ \_\ \mathsf{out}\ a))) = BP\ w\ d\ \mathsf{out}\ (tail\ a) \\
backward :: (Weights, Biases) \rightarrow BackProp \rightarrow ((Weights, Biases), Deltas) \\
backward\ (w1, b1)\ (BP\ w2\ d2\ \mathsf{out}\ (a1 : a0 : as)) = ((w1', b1'), Just\ d1) \\
\quad \mathbf{where}\ d1 = \mathbf{case}\ d2\ \mathbf{of} \\
\qquad\quad Nothing \rightarrow pointMult\ (elementwise\ dCost\ a1\ \mathsf{out})\ (fmap\ relu'\ a1) \\
\qquad\quad Just\ d2' \rightarrow pointMult\ (d2' * w2)\ (fmap\ relu'\ a1) \\
\qquad\quad w1' = descend\ w1\ (outProd\ a0\ d1) \\
\qquad\quad b1' = descend\ b1\ d1
\end{array}$$

Training process:

$$\begin{array}{l}
training :: Fix\ Layer \rightarrow [(Values, Values)] \rightarrow Fix\ Layer \\
training\ nn = \llparenthesis [\underline{nn}, h] \rrparenthesis \\
\quad \mathbf{where}\ h = train \cdot (id \times swap) \cdot assocr \\
train :: (Values, (Fix\ Layer, Values)) \rightarrow Fix\ Layer \\
train = learn \cdot \langle \pi_1 \cdot \pi_2, \widehat{bp2} \cdot (id \times feed') \rangle \\
\quad \mathbf{where}\ bp2 = BP\ (zero\ 1\ 1)\ Nothing
\end{array}$$

C

RECURRENT NEURAL NETWORK (RNN) CODE

Initializing a RNN:

$$\mathit{newBrainR} :: (((\mathit{Int},\mathit{Int}),a),(\mathit{Int},a)) \rightarrow \mathit{IO}\ (\mathit{RLayer}\ a)$$
$$\mathit{newBrainR}\ (((x,h),\mathit{ax}),(y,\mathit{ay})) = \mathbf{do}\ \mathit{lx} \leftarrow \mathit{generateLayer}\ ((x+h,h),\mathit{ax})$$
$$\quad \mathit{ly} \leftarrow \mathit{generateLayer}\ ((h,y),\mathit{ay})$$
$$\quad \mathit{return}\ (\mathit{inRL}\ (\mathit{lx},\mathit{ly}))$$

Feeding process:

$$\mathit{feedR} :: \mathit{RLayer}\ (\mathit{Values} \rightarrow \mathit{Values}) \rightarrow \mathit{Values} \rightarrow [\mathit{Values}] \rightarrow (\mathit{Values},[\mathit{Values}])$$
$$\mathit{feedR}\ \mathit{nn}\ s = (\mathit{id} \times \mathit{map}\ (\pi_1 \cdot \mathit{outBR})) \cdot \mathit{feedR}'\ (\mathit{nn},s)$$
$$\mathit{feedR}' :: (\mathit{RLayer}\ (\mathit{Values} \rightarrow \mathit{Values}),\mathit{Values}) \rightarrow [\mathit{Values}] \rightarrow (\mathit{Values},[\mathit{BackPropR}])$$
$$\mathit{feedR}'\ (\mathit{nn},s) = \mathit{mapAccumL}\ \overline{\mathit{algR}\ \mathit{nn}}\ s$$
$$\mathit{algR} :: \mathit{RLayer}\ (\mathit{Values} \rightarrow \mathit{Values}) \rightarrow (\mathit{Values},\mathit{Values}) \rightarrow (\mathit{Values},\mathit{BackPropR})$$
$$\mathit{algR}\ (\mathit{RL}\ \mathit{lx}\ \mathit{ly}) = \langle \pi_1, g\rangle \cdot \langle f,\mathit{id}\rangle$$
$$\quad \mathbf{where}\ f = \mathit{useL}\ \mathit{lx} \cdot \mathit{concatM} \cdot \mathit{swap}$$
$$\qquad g = \mathit{inBR} \cdot \langle \mathit{useL}\ \mathit{ly} \cdot \pi_1,\mathit{id}\rangle$$

Learning process:

$$\mathit{learnR} :: ((\mathit{Values} \rightarrow \mathit{Values}),(\mathit{Values} \rightarrow \mathit{Values}))$$
$$\quad \rightarrow (\mathit{RLayer}\ (\mathit{Values} \rightarrow \mathit{Values}),[(\mathit{Values},\mathit{BackPropR})])$$
$$\quad \rightarrow \mathit{RLayer}\ (\mathit{Values} \rightarrow \mathit{Values})$$
$$\mathit{learnR}\ (\mathit{fx},\mathit{fy}) = \widehat{\mathit{backwardR}} \cdot \langle \pi_2 \cdot \widehat{f},\pi_1\rangle$$
$$\quad \mathbf{where}\ f\ \mathit{nn} = \llparenthesis [g \cdot \underline{\mathit{nn}},\widehat{\mathit{coalgR}\ \mathit{nn}}] \rrparenthesis$$
$$\qquad g = \langle [\underline{\mathit{zero}\ 1\ 1},\pi_2 \cdot \pi_1] \cdot \mathit{outL} \cdot \pi_1,\mathit{inRL}\rangle \cdot (h\ \mathit{fx} \times h\ \mathit{fy}) \cdot \mathit{outRL}$$
$$\qquad h\ \mathit{fl} = \mathit{inL} \cdot (\mathit{id} + ((i \times i) \times \underline{\mathit{fl}})) \cdot \mathit{outL}$$
$$\qquad i = \widehat{\mathit{zero}} \cdot \langle \mathit{nrows},\mathit{ncols}\rangle$$
$$\mathit{coalgR} :: \mathit{RLayer}\ (\mathit{Values} \rightarrow \mathit{Values})$$
$$\quad \rightarrow (\mathit{Values},\mathit{BackPropR})$$

$$
\begin{aligned}
&\quad \to (\mathit{Values}, \mathit{RLayer}\ (\mathit{Values} \to \mathit{Values}))\\
&\quad \to (\mathit{Values}, \mathit{RLayer}\ (\mathit{Values} \to \mathit{Values}))\\
&\mathit{coalgR}\ (\mathit{RL}\ \mathit{lx}\ \mathit{ly})\ (o, (\mathit{BPR}\ y\ a'\ a\ x))\ (\mathit{danext}, (\mathit{RL}\ \mathit{dlx}\ \mathit{dly})) = (\mathit{danext}', (\mathit{RL}\ \mathit{dlx}'\ \mathit{dly}'))\\
&\quad \mathbf{where}\ \mathit{danext}' = \mathit{daraw} * (f\ \mathit{daraw}\ \mathit{lx})\\
&\qquad \mathit{dlx}' = g\ \mathit{daraw}\ (\mathit{concatM}\ (x, a))\ \mathit{dlx}\\
&\qquad \mathit{dly}' = g\ \mathit{dy}\ a'\ \mathit{dly}\\
&\qquad \mathit{daraw} = \mathit{pointMult}\ ((i\ \mathit{lx})\ a')\ \mathit{da}\\
&\qquad \mathit{da} = (\mathit{dy} * (f\ \mathit{dy}\ \mathit{ly})) + \mathit{danext}\\
&\qquad \mathit{dy} = \mathit{pointMult}\ (\mathit{elementwise}\ \mathit{dCost}\ y\ o)\ ((i\ \mathit{ly})\ y)\\
&\qquad f\ d = [\mathit{identity} \cdot \underline{\mathit{ncols}\ d}, \mathit{transpose} \cdot \pi_1 \cdot \pi_1] \cdot \mathit{outL}\\
&\qquad g\ d\ v = \mathit{inL} \cdot (\mathit{id} + (h\ d\ v)) \cdot \mathit{outL}\\
&\qquad h\ d\ v = (((+(\mathit{outProd}\ d\ v)) \times (+d)) \times \mathit{id})\\
&\qquad i = [\underline{\mathit{id}}, \pi_2] \cdot \mathit{outL}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{backwardR} :: \mathit{RLayer}\ (\mathit{Values} \to \mathit{Values})\\
&\qquad \to \mathit{RLayer}\ (\mathit{Values} \to \mathit{Values})\\
&\qquad \to \mathit{RLayer}\ (\mathit{Values} \to \mathit{Values})\\
&\mathit{backwardR}\ (\mathit{RL}\ \mathit{dlx}\ \mathit{dly}) = \mathit{inRL} \cdot (f\ (\mathit{outL}\ \mathit{dlx}) \times f\ (\mathit{outL}\ \mathit{dly})) \cdot \mathit{outRL}\\
&\quad \mathbf{where}\ f\ (\mathit{Right}\ (d, \_)) = \mathit{inL} \cdot (\mathit{id} + (g\ d)) \cdot \mathit{outL}\\
&\qquad f\ \_ = \mathit{id}\\
&\qquad g\ (\mathit{dw}, \mathit{db}) = ((h\ \mathit{dw} \times h\ \mathit{db}) \times \mathit{id})\\
&\qquad h\ m = (m+) \cdot \mathit{fmap}\ ((-0.1)*) \cdot \widehat{\mathit{elementwise}}\ (/) \cdot \langle \mathit{id}, \mathit{fmap}\ (\mathit{sqrt} \cdot (+1\mathrm{e}{-8}) \cdot (\uparrow 2))\rangle
\end{aligned}
$$

Training process:

$$
\begin{aligned}
&\mathit{trainingR} :: \mathit{RLayer}\ (\mathit{Values} \to \mathit{Values})\\
&\qquad \to ((\mathit{Values} \to \mathit{Values}), (\mathit{Values} \to \mathit{Values}))\\
&\qquad \to \mathit{Values}\\
&\qquad \to [([\mathit{Values}], [\mathit{Values}])]\\
&\qquad \to \mathit{RLayer}\ (\mathit{Values} \to \mathit{Values})\\
&\mathit{trainingR}\ \mathit{nn}\ \mathit{fs}\ s = \llparenthesis\, [\underline{\mathit{nn}}, h]\, \rrparenthesis\\
&\quad \mathbf{where}\ h = (\mathit{trainR}\ \mathit{fs}) \cdot \langle \mathit{hr}, \pi_1 \cdot \pi_1 \rangle\\
&\qquad \mathit{hr} = \langle h' \cdot \pi_2, \pi_2 \cdot \pi_1 \rangle\\
&\qquad h' = \langle \mathit{id}, \underline{s} \rangle
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{trainR} :: ((\mathit{Values} \to \mathit{Values}), (\mathit{Values} \to \mathit{Values}))\\
&\quad \to (((\mathit{RLayer}\ (\mathit{Values} \to \mathit{Values}), \mathit{Values}), [\mathit{Values}]), [\mathit{Values}])\\
&\quad \to \mathit{RLayer}\ (\mathit{Values} \to \mathit{Values})\\
&\mathit{trainR}\ \mathit{fs} = (\mathit{learnR}\ \mathit{fs}) \cdot (\mathit{id} \times \widehat{\mathit{zip}}) \cdot \mathit{assocr} \cdot \langle f, g \rangle\\
&\quad \mathbf{where}\ f = (\pi_1 \cdot \pi_1) \times \mathit{id}\\
&\qquad g = \pi_2 \cdot \widehat{\mathit{feedR}'} \cdot \pi_1
\end{aligned}
$$

D

BIDIRECTIONAL RECURRENT NEURAL NETWORK'S CODE

Initializing a BRNN:

$$\begin{array}{l}
\mathit{newBrainBR} :: (((\mathit{Int},\mathit{Int}),a),(\mathit{Int},a)) \\
\quad \rightarrow \mathit{IO}\ (\mathit{BRLayer}\ a) \\
\mathit{newBrainBR}\ (((x,h),\mathit{ax}),(y,\mathit{ay})) = \mathbf{do}\ \mathit{lxl} \leftarrow \mathit{generateLayer}\ ((x+h,h),\mathit{ax}) \\
\quad \mathit{lxr} \leftarrow \mathit{generateLayer}\ ((x+h,h),\mathit{ax}) \\
\quad \mathit{ly} \leftarrow \mathit{generateLayer}\ ((h+h,y),\mathit{ay}) \\
\quad \mathit{return}\ (\mathit{inBRL}\ ((\mathit{lxl},\mathit{lxr}),\mathit{ly}))
\end{array}$$

Feeding process:

$$\begin{array}{l}
\mathit{feedBR} :: \mathit{BRLayer}\ (\mathit{Values} \rightarrow \mathit{Values}) \rightarrow \mathit{Values} \rightarrow [\mathit{Values}] \\
\quad \rightarrow ((\mathit{Values},\mathit{Values}),[\mathit{Values}]) \\
\mathit{feedBR}\ \mathit{nn}\ s = (\mathit{id} \times \mathit{map}\ (\mathit{fst} \cdot \mathit{outBBR})) \cdot \mathit{feedBR}'\ (\mathit{nn},s)
\end{array}$$

$$\begin{array}{l}
\mathit{feedBR}' :: (\mathit{BRLayer}\ (\mathit{Values} \rightarrow \mathit{Values}),\mathit{Values}) \rightarrow [\mathit{Values}] \\
\quad \rightarrow ((\mathit{Values},\mathit{Values}),[\mathit{BackPropBR}]) \\
\mathit{feedBR}'\ (\mathit{nn},s) = f \cdot \langle l,r \rangle \\
\quad \mathbf{where}\ l = \mathit{mapAccumL}\ \overline{\mathit{algBR}\ \mathit{lxl}}\ s \\
\qquad r = \mathit{mapAccumR}\ \overline{\mathit{algBR}\ \mathit{lxr}}\ s \\
\qquad f = \langle \pi_1 \times \pi_1, \widehat{\mathit{zipWith}}\ \overline{g} \cdot (\pi_2 \times \pi_2) \rangle \\
\qquad g = \mathit{inBBR} \cdot \langle \mathit{useL}\ \mathit{ly} \cdot \mathit{concatM} \cdot (\pi_1 \times \pi_1),\mathit{id} \rangle \\
\qquad ((\mathit{lxl},\mathit{lxr}),\mathit{ly}) = \mathit{outBRL}\ \mathit{nn}
\end{array}$$

$$\begin{array}{l}
\mathit{algBR} :: \mathit{Layer}\ (\mathit{Values} \rightarrow \mathit{Values}) \rightarrow (\mathit{Values},\mathit{Values}) \\
\quad \rightarrow (\mathit{Values},(\mathit{Values},(\mathit{Values},\mathit{Values}))) \\
\mathit{algBR}\ l = \langle \pi_1,\mathit{id} \rangle \cdot \langle f,\mathit{id} \rangle \\
\quad \mathbf{where}\ f = \mathit{useL}\ l \cdot \mathit{concatM} \cdot \mathit{swap}
\end{array}$$

Learning Process:

$$\begin{array}{l}
\mathit{learnBR} :: ((\mathit{Values} \rightarrow \mathit{Values}),(\mathit{Values} \rightarrow \mathit{Values})) \\
\quad \rightarrow (\mathit{BRLayer}\ (\mathit{Values} \rightarrow \mathit{Values}),[(\mathit{Values},\mathit{BackPropBR})])
\end{array}$$

$$
\begin{array}{l}
\quad \rightarrow \mathit{BRLayer}\ (\mathit{Values} \rightarrow \mathit{Values}) \\
\mathit{learnBR}\ (\mathit{fx},\mathit{fy}) = \widehat{\mathit{backwardBR}} \cdot \langle f, \pi_1 \rangle \\
\quad \mathbf{where}\ f = \mathit{inBRL} \cdot \langle \mathit{l2}, \mathit{r2} \rangle \cdot \langle l, r \rangle \cdot (\mathit{outBRL} \times \mathit{map}\ (\mathit{id} \times \mathit{outBBR})) \\
\qquad l = \pi_1 \times \mathit{map}\ (\pi_2 \cdot \pi_2) \\
\qquad r = \widehat{r'} \cdot \mathit{assocr} \cdot (((\langle \mathit{id}, g\ \mathit{fy} \rangle \cdot \pi_2) \times \mathit{map}\ (\mathit{id} \times (\mathit{id} \times (\pi_1 \times \pi_1)))) \\
\qquad r'\ l = \widehat{\mathit{mapAccumL}\ (\mathit{coalgBR}\ l)} \\
\qquad \mathit{l2} = \langle \mathit{l3}, \mathit{r3} \rangle \cdot (\mathit{id} \times \pi_2) \\
\qquad \mathit{r2} = \pi_1 \cdot \pi_2 \\
\qquad \mathit{l3} = i \cdot ((\pi_1 \times (\mathit{map}\ \pi_1)) \times (\mathit{map}\ \pi_1)) \\
\qquad \mathit{r3} = i \cdot ((\pi_2 \times (\mathit{reverse} \cdot \mathit{map}\ \pi_2)) \times (\mathit{reverse} \cdot \mathit{map}\ \pi_2)) \\
\qquad g\ \mathit{fl} = \mathit{inL} \cdot (\mathit{id} + ((h \times h) \times \underline{\mathit{fl}})) \cdot \mathit{outL} \\
\qquad h = \widehat{\mathit{zero}} \cdot \langle \mathit{nrows}, \mathit{ncols} \rangle \\
\qquad i = \pi_2 \cdot \widehat{i'} \cdot (\mathit{id} \times \widehat{\mathit{zip}}) \cdot \mathit{assocr} \\
\qquad i'\ l = \llparenthesis\, [\langle i'', \mathit{id} \rangle \cdot g\ \mathit{fx} \cdot \underline{l}, \widehat{\mathit{coalgBR}'\ l}]\, \rrparenthesis \\
\qquad i'' = [\underline{\mathit{zero}\ 1\ 1}, \pi_2 \cdot \pi_1] \cdot \mathit{outL}
\end{array}
$$

$$
\begin{array}{l}
\mathit{coalgBR} :: \mathit{Layer}\ (\mathit{Values} \rightarrow \mathit{Values}) \\
\quad \rightarrow \mathit{Layer}\ (\mathit{Values} \rightarrow \mathit{Values}) \\
\quad \rightarrow (\mathit{Values}, (\mathit{Values}, (\mathit{Values}, \mathit{Values}))) \\
\quad \rightarrow (\mathit{Layer}\ (\mathit{Values} \rightarrow \mathit{Values}), (\mathit{Values}, \mathit{Values})) \\
\mathit{coalgBR}\ \mathit{ly}\ \mathit{dly}\ (o, (y, a)) = (\mathit{dly}', \mathit{da}) \\
\quad \mathbf{where}\ \mathit{dly}' = f\ \mathit{dy}\ (\mathit{concatM}\ a)\ \mathit{dly} \\
\qquad \mathit{da} = g\ (\mathit{dy} * (h\ \mathit{dy}\ \mathit{ly})) \\
\qquad \mathit{dy} = \mathit{pointMult}\ (\mathit{elementwise}\ \mathit{dCost}\ y\ o)\ ((i\ \mathit{ly})\ y) \\
\qquad f\ d\ v = \mathit{inL} \cdot (\mathit{id} + (f'\ d\ v)) \cdot \mathit{outL} \\
\qquad f'\ d\ v = (((+(\mathit{outProd}\ d\ v)) \times (+d)) \times \mathit{id}) \\
\qquad g = \langle g'\ \mathit{take}, g'\ \mathit{drop} \rangle \cdot \mathit{toList} \\
\qquad g'\ \mathit{fg} = \mathit{fromLists} \cdot \mathit{singl} \cdot \widehat{\mathit{fg}} \cdot \langle (\mathit{flip}\ \mathit{div}\ 2) \cdot \mathit{length}, \mathit{id} \rangle \\
\qquad h\ d = [\mathit{identity} \cdot \underline{\mathit{ncols}\ d}, \mathit{transpose} \cdot \pi_1 \cdot \pi_1] \cdot \mathit{outL} \\
\qquad i = [\underline{\mathit{id}}, \pi_2] \cdot \mathit{outL}
\end{array}
$$

$$
\begin{array}{l}
\mathit{coalgBR}' :: \mathit{Layer}\ (\mathit{Values} \rightarrow \mathit{Values}) \\
\quad \rightarrow ((\mathit{Values}, (\mathit{Values}, \mathit{Values})), \mathit{Values}) \\
\quad \rightarrow (\mathit{Values}, \mathit{Layer}\ (\mathit{Values} \rightarrow \mathit{Values})) \\
\quad \rightarrow (\mathit{Values}, \mathit{Layer}\ (\mathit{Values} \rightarrow \mathit{Values})) \\
\mathit{coalgBR}'\ l\ ((a', (a, x)), \mathit{da})\ (\mathit{danext}, \mathit{dl}) = (\mathit{danext}', \mathit{dl}') \\
\quad \mathbf{where}\ \mathit{danext}' = \mathit{daraw} * (f\ \mathit{daraw}\ l) \\
\qquad \mathit{dl}' = g\ \mathit{daraw}\ (\mathit{concatM}\ (x, a))\ \mathit{dl} \\
\qquad \mathit{daraw} = \mathit{pointMult}\ ((i\ l)\ a')\ (\mathit{da} + \mathit{danext}) \\
\qquad f\ d = [\mathit{identity} \cdot \underline{\mathit{ncols}\ d}, \mathit{transpose} \cdot \pi_1 \cdot \pi_1] \cdot \mathit{outL}
\end{array}
$$

$$\begin{array}{l}
\qquad g\ d\ v = inL \cdot (id + (h\ d\ v)) \cdot outL \\
\qquad h\ d\ v = (((+(outProd\ d\ v)) \times (+d)) \times id) \\
\qquad i = [\underline{id}, \pi_2] \cdot outL \\
backwardBR :: BRLayer\ (Values \rightarrow Values) \rightarrow BRLayer\ (Values \rightarrow Values) \\
\quad \rightarrow BRLayer\ (Values \rightarrow Values) \\
backwardBR\ (BRL\ dlxl\ dlxr\ dly) = \\
\quad inBRL \cdot ((f\ (outL\ dlxl) \times f\ (outL\ dlxr)) \times f\ (outL\ dly)) \cdot outBRL \\
\quad \textbf{where}\ f\ (Right\ (d, \_)) = inL \cdot (id + (g\ d)) \cdot outL \\
\qquad f\ \_ = id \\
\qquad g\ (dw, db) = ((h\ dw \times h\ db) \times id) \\
\qquad h\ m = (m+) \cdot fmap\ ((-0.1)*) \cdot \widehat{elementwise}\ (/) \cdot \langle id, fmap\ (sqrt \cdot (+1\mathrm{e}{-8}) \cdot (\uparrow 2))\rangle
\end{array}$$

Training process:

$$\begin{array}{l}
trainingBR :: BRLayer\ (Values \rightarrow Values) \\
\quad \rightarrow ((Values \rightarrow Values), (Values \rightarrow Values)) \\
\quad \rightarrow Values \\
\quad \rightarrow [([Values], [Values])] \\
\quad \rightarrow BRLayer\ (Values \rightarrow Values) \\
trainingBR\ nn\ fs\ s = (\!|\, [\underline{nn}, h]\, |\!) \\
\quad \textbf{where}\ h = (trainBR\ fs) \cdot \langle hr, \pi_1 \cdot \pi_1 \rangle \\
\qquad hr = \langle hr' \cdot \pi_2, \pi_2 \cdot \pi_1 \rangle \\
\qquad hr' = \langle id, \underline{s} \rangle \\
trainBR :: ((Values \rightarrow Values), (Values \rightarrow Values)) \\
\quad \rightarrow (((BRLayer\ (Values \rightarrow Values), Values), [Values]), [Values]) \\
\quad \rightarrow BRLayer\ (Values \rightarrow Values) \\
trainBR\ fs = (learnBR\ fs) \cdot (id \times \widehat{zip}) \cdot assocr \cdot \langle f, g \rangle \\
\quad \textbf{where}\ f = (\pi_1 \cdot \pi_1) \times id \\
\qquad g = \pi_2 \cdot \widehat{feedBR'} \cdot \pi_1
\end{array}$$

E

AUXILIAR FUNCTIONS

Basic Functions for Fixpoint Manipulation:

$$\mathbf{data}\ \mathit{Fix}\ a = \mathit{In}\ (a\ (\mathit{Fix}\ a))$$

$$\mathit{inF} :: a\ (\mathit{Fix}\ a) \to \mathit{Fix}\ a$$

$$\mathit{inF} = \mathit{In}$$

$$\mathit{outF} :: \mathit{Functor}\ a \Rightarrow \mathit{Fix}\ a \to a\ (\mathit{Fix}\ a)$$

$$\mathit{outF}\ (\mathit{In}\ f) = f$$

$$[\!(\cdot)\!] :: \mathit{Functor}\ f \Rightarrow (a \to f\ a) \to a \to \mathit{Fix}\ f$$

$$[\!(f)\!] = \mathit{inF} \cdot \mathit{fmap}\ [\!(f)\!] \cdot f$$

$$(\!| \cdot |\!) :: \mathit{Functor}\ f \Rightarrow (f\ a \to a) \to \mathit{Fix}\ f \to a$$

$$(\!| f |\!) = f \cdot \mathit{fmap}\ (\!| f |\!) \cdot \mathit{outF}$$

Functions for matrix manipulation:

$$\mathit{concatM} :: (\mathit{Matrix}\ a, \mathit{Matrix}\ a) \to \mathit{Matrix}\ a$$

$$\mathit{concatM} = \widehat{<|>}$$

$$\mathit{descend} :: \mathit{Matrix\ Float} \to \mathit{Matrix\ Float} \to \mathit{Matrix\ Float}$$

$$\mathit{descend}\ \mathit{iv} = (-)\ \mathit{iv} \cdot \mathit{fmap}\ (\eta *)$$

$$\mathit{expand} :: \mathit{Int} \to \mathit{Matrix\ Float} \to \mathit{Matrix\ Float}$$

$$\mathit{expand}\ n = \mathit{fromLists} \cdot \mathit{replicate}\ n \cdot \mathit{toList}$$

$$\mathit{outProd} :: \mathit{Matrix\ Float} \to \mathit{Matrix\ Float} \to \mathit{Matrix\ Float}$$

$$\mathit{outProd}\ x\ y = \mathit{pointMult}\ (\mathit{transpose}\ (\mathit{expand}\ (\mathit{ncols}\ x)\ y))\ (\mathit{expand}\ (\mathit{ncols}\ y)\ x)$$

$$\mathit{pointMult} :: \mathit{Matrix\ Float} \to \mathit{Matrix\ Float} \to \mathit{Matrix\ Float}$$

$$\mathit{pointMult} = \mathit{elementwise}\ (*)$$

Generating functions:

$$\mathit{generateLayer} :: ((\mathit{Int}, \mathit{Int}), a) \to \mathit{IO}\ (\mathit{Layer}\ a)$$

$$\mathit{generateLayer}\ ((x, y), a) = \mathbf{do}\ w \leftarrow \mathit{generateMatrix}\ (x, y)$$

$\quad b \leftarrow \mathit{generateMatrix}\ (1, y)$

$\quad \mathit{return}\ (\mathit{inL}\ (i_2\ ((w, b), a)))$

$\mathit{generateMatrix} :: (\mathit{Int}, \mathit{Int}) \rightarrow \mathit{IO}\ (\mathit{Matrix\ Float})$

$\mathit{generateMatrix} = (\mathit{fmap\ fromLists}) \cdot \widehat{\mathit{replicateM}} \cdot (\mathit{id} \times (\mathit{flip\ replicateM}\ (\mathit{gauss}\ 0.01)))$

Other small auxiliar functions:

$\mathit{applyR} :: (a \rightarrow b, a) \rightarrow b$

$\mathit{applyR}\ (f, x) = f\ x$

$\mathit{dCost} :: \mathit{Float} \rightarrow \mathit{Float} \rightarrow \mathit{Float}$

$\mathit{dCost}\ a\ y \mid y \equiv 1 \wedge a \geqslant y = 0$

$\quad \mid \mathit{otherwise} = a - y$

$\eta = 0.002$

$\mathit{gauss} :: \mathit{Float} \rightarrow \mathit{IO\ Float}$

$\mathit{gauss\ scale} = \mathbf{do}\ x_1 \leftarrow \mathit{randomIO}$

$\quad x_2 \leftarrow \mathit{randomIO}$

$\quad \mathit{return}\ \$\ \mathit{scale} * \mathit{sqrt}\ (-2 * \mathit{log}\ x_1) * \mathit{cos}\ (2 * \mathit{pi} * x_2)$

$\mathit{relu} :: \mathit{Float} \rightarrow \mathit{Float}$

$\mathit{relu} = \mathit{max}\ 0$

$\mathit{relu}' :: \mathit{Float} \rightarrow \mathit{Float}$

$\mathit{relu}'\ x \mid x \leqslant 0 = 0$

$\quad \mid \mathit{otherwise} = 1$

$\mathit{sigmoid} :: \mathit{Float} \rightarrow \mathit{Float}$

$\mathit{sigmoid} = (1/) \cdot (1+) \cdot \mathit{exp} \cdot ((-1)*)$

$\mathit{useL} :: \mathit{Layer}\ (\mathit{Values} \rightarrow \mathit{Values}) \rightarrow \mathit{Values} \rightarrow \mathit{Values}$

$\mathit{useL}\ (\mathit{IL}) = \mathit{id}$

$\mathit{useL}\ (\mathit{DL}\ w\ b\ \mathit{fl}) = \mathit{fl} \cdot (+b) \cdot (*w)$

F

NEURAL NETWORK TESTING

Datasets to be used:

```
dataSets = mapM (fmap decompress · BS.readFile) [
  "../Imports_And_Test_Files/train-images-idx3-ubyte.gz",
  "../Imports_And_Test_Files/train-labels-idx1-ubyte.gz",
  "../Imports_And_Test_Files/t10k-images-idx3-ubyte.gz",
  "../Imports_And_Test_Files/t10k-labels-idx1-ubyte.gz"
  ]
```

Data processing functions:

```
getImage s n = fmap (fromIntegral · BS.index s · (n ∗ 28 ↑ 2 + 16+)) [0 . . 28 ↑ 2 − 1]
getLabel s = fromIntegral · BS.index s · (8+)
getPairs s1 s2 ns = map (Cp.split (getY s1) (getX s2)) ns
getX s n = fromLists (singl ((fmap (/256) (getImage s n))))
getY s n = fromLists (singl (fmap f [0 . . 9]))
  where f = fromIntegral · fromEnum · (getLabel s n ≡)
```

Digit rendering functions:

```
drawDigit :: [Int] → IO ()
drawDigit = putStr · unlines · take 28 · fmap (take 28) · iterate (drop 28) · fmap render
render :: Integral a ⇒ a → Char
render n = let s = "  .:oO@"
  in s !! (fromIntegral n ∗ length s ‘div‘ 256)
```

Information representation functions:

```
bestOf :: Matrix Float → Int
bestOf = fst · maximumBy (comparing snd) · zip [0 . .] · toList
```

```
cute :: Int → Float → String
cute d score = show d ⧺ ": " ⧺ replicate (round $ 100 ∗ min 1 score) '+'
```

Testing functions:

```
getABrain = do
  [trainI, trainL, testI, testL] ← dataSets
  b ← newBrain [784, 30, 10]
  putStrLn "Select the test that must be used (0-9999):"
  n ← readLn
  drawDigit (getImage testI n)
  let
    example = getX testI n
    label = getLabel testL n
    bs = scanl training b (map (getPairs trainL trainI) [[0 . . 999],
      [1000 . . 2999],
      [3000 . . 5999],
      [6000 . . 9999]])
    smart = last bs
  putStrLn $ "Expected value: " ⧺ show label
  forM_ bs $ putStrLn · unlines · zipWith cute [0 . . 9] · toList · flip feed example
  putStrLn "Running all tests now..."
  let
    guesses = fmap (bestOf · (λn → feed smart (getX testI n))) [0 . . 9999]
    answers = fmap (getLabel testL) [0 . . 9999]
  putStrLn $ show (sum $ fmap fromEnum (zipWith (≡) guesses answers)) ⧺
    " right guesses in 10000 tests."


allDigits = do
  [trainI, trainL, testI, testL] ← dataSets
  b ← newBrain [784, 30, 10]
  let
    smart = training b (getPairs trainL trainI [0 . . 9999])
    guesses = fmap (bestOf · (λn → feed smart (getX testI n))) [0 . . 9999]
    answers = fmap (getLabel testL) [0 . . 9999]
  putStrLn $ show (sum $ fmap fromEnum (zipWith (≡) guesses answers)) ⧺
    " right guesses in 10000 tests."
specificDigit = do
```

```
[trainI, trainL, testI, testL] ← dataSets
b ← newBrain [784, 30, 10]
putStrLn "Select the test that must be used (0-9999):"
n ← readLn
drawDigit (getImage testI n)
let
  example = getX testI n
  label = getLabel testL n
  smart = training b (getPairs trainL trainI [0 . . 9999])
putStrLn $ "Expected value: " ++ show label
putStrLn $ "Best Guess: " ++ show (bestOf $ feed smart example)
```