UNIVERSITY OF BRISTOL

MASTERS THESIS

Modelling Neural Networks with Recursion Schemes

Author:
Minh NGUYEN

Supervisor:
Dr. Nicolas WU

A thesis submitted in fulfillment of the requirements for the degree of Computer Science

in the

Department Of Engineering
University Of Bristol

June 1st 2019

Declaration of Authorship

I, Minh NGUYEN, declare that this thesis titled, "Modelling Neural Networks with Recursion Schemes" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Contents

Executive Summary

I postulate that the structure and training of neural networks can be modelled using recursion schemes to represent forward and back propagation as the act of folding and unfolding. During the project:

- I learned
  - The fundamental algorithmic process of forward and back propagation
  - The intrinsic recursive properties presented in neural networks
  - How to unify and implement the relationship of neural networks and recursion schemes
- I implemented three neural networks using a recursion scheme system of catamorphisms and anamorphisms as the core components. Specifically, I:
  - Modelled fully connected neural networks as a metamorphism (a catamorphism composed with an anamorphism)
  - Modelled convolutional neural networks as a metamorphism
  - Modelled deep LSTM neural networks as a metamorphism, where its catamorphism calls a another catamorphism, and its anamorphism calls a hylomorphism (an anamorphism composed with a catamorphism)
- I have demonstrated:
  - The ability of all implementations to successfully learn from training data
  - A common pattern of metamorphic behaviour displayed by the investigated neural networks
  - The advantages and potential which this approach offers in comparison to existing technologies.

Supporting Technologies

- I used the *Haskell* language (GHC 7.10.3, Stack 1.9.3), and a minimal number of libraries such as *lens* and *vector*
- I used the *Python 2.7* language and *SciPy* library to construct graphs to demonstrate my implementation's functionality.

Acknowledgements

Many thanks to my supervisor, Dr. Nicolas Wu, for his invaluable insight and helping me feel smarter than I actually am, and to my best friend Alessio Zakaria, who helps me feel less smart than I actually am.

Chapter 1

Introduction

The field of deep learning continually advances at an astronomical rate and is responsible for a huge amount of our economical and technological growth, with neural networks becoming arguable the most significant contributors to machine learning [11]. Data acts as the life force of all companies with their success primarily reliant on data-driven decisions; a study conducted finds that 49% of organizations report exploring into deploying machine learning, while a slight majority of 51% claimed to be early adopters or sophisticated users [1]. This all means that we are becoming increasingly dependent on neural network systems to act as the basis on which vital decisions are being made. To sustain this, we need to know that the foundations on which neural networks are constructed and trained can enable the most effective and reliable approaches possible - but how true is this currently?

Neural networks present very inspiring ways of designing more purer and elegant portrayals of their structure, and methods which represent the flow of the training process better. This project intends to capture these ideas through demonstrating through implementation the potential relationship between recursion schemes and deep learning, in hope of unifying these disjoint fields.

1.1 Neural Networks

A neural network is a series of algorithms that endeavors to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. Their structure is hence a set of neurons and connections, capable of taking input data and applying transformations to output more meaningful information and learn from any errors incurred. Neural networks can adapt to changing input; so the network generates the best possible result without needing to redesign the output criteria.

The process describing input data being transformed to classifiable output data is referred to as *forward propagation*, and the process describing the error being used to update and adapt the neural network's hyperparameters is referred to as *back propagation*. The general simplicity and effectiveness of neural networks as compared to other machine learning approaches have led them to become dominant in the machine learning field. As neural networks have been implemented using various programming languages into frameworks and libraries, the various approaches are questionable.

1.2 Objectives: Current State Of Deep Learning

In the current state of deep learning development, Python is overwhelmingly the most used language in this industry [9]. Alongside this, the python libraries Keras [10] and Tensorflow [13] strongly overshadow any framework alternatives [8]. However Keras is little more than a high-level API that works as a wrapper around Tensorflow. Given the informal consensus that these are the best available tools to the deep learning community, on what grounds does this new approach to architecting neural networks contest its relevancy?

- **Representational Power:** Python is multi-paradigm and supports OO, procedural, and functional programming styles - this can result in very inconsistent and indecisive design. At the same time, as a language becomes more accessible and high level, it reduces the amount of fine-grained control a user has over their program. As a result, the Tensorflow and Keras libraries are implemented with a very unusual methodology which fails to grasp the underlying mechanism of neural networks. Tensorflow's approach of describing very primitive computations by abstracting them into computational graphs harmfully diverges from the neural network's natural system.

- **Transparency:** Libraries built on Python, e.g. Tensorflow and Keras, are built on a purely declarative paradigm which is problematic for deep learning developers. Declarative languages focus on describing what should be computed and avoid mentioning how that computation should be performed. This is practical for situations where the mechanics of data flow and state propagation makes sense to be hidden from the developer. In deep learning, we should care very much care about the computation's logistics if we are to achieve fine control. Unfortunately this leads to a strong lack of transparency in the flow of a Tensorflow program, due to the necessity of calling sub-processes to handle fundamentally important tasks without any power or confident understanding of the mechanism.
- **Reliability:** The narrow focus of the deep learning industry on achieving fast results and a quality final performance is harmful. This completely ignores the need for correctness and sustainable design. The idea that Python's dynamic type system allows superior productivity is an illusion; it is fundamentally unreliable and near impossible to formally verify. There exist situations where a Python program can terminate successfully yet have unintended types inferred. Immediate type verification is invaluable when handling the complexity that arises in designing large scale networks.

It should additionally be noted that *performance* is an area which this project does *not* intend to address or discuss.

1.3 Proposal: Recursion Schemes Within Neural Networks

There are multiple narratives to which deep learning can be understood. Neuroscience draws analogies to biology. A representative narrative focuses on the transformations of data and the manifold hypothesis. The probabilistic narrative interprets neural networks as finding latent variables. Although these perspectives aren't mutually exclusive, they present very different ways of expressing deep learning. As a proposed solution, this project will entail researching an uninvestigated narrative - the relationship between neural networks to recursion schemes and functional programming, and the impacts which this design paradigm can have on future of deep learning. The system which this project implements and discusses is called **Catana**.

1.4 Technical Aims

The primary goals of this project are to:

- **Demonstrate the ability of neural networks to be represented with recursion schemes.** I will implement three neural networks using a recursion scheme system in Haskell from the ground-up, using only core libraries from the Haskell platform
- **Illustrate the relationship between neural networks and recursion schemes.** This will occur through making various comparisons which demonstrate the inherent recursive nature in neural nets, and discussing the reoccurring patterns that present themselves during implementation.
- **Present results demonstrating that all implementations can learn correctly.** This will show that our implementation's accuracy in classifying data improves as more samples are provided.
- **Evaluate Catana in terms of significance and implementation quality.** I will discuss what the potential and benefits that this approach brings to the representation of neural networks, and assess Catana's strengths and weaknesses in comparison to existing implementations.

Outline In Chapter 2 we will discuss any existing research related to this project's proposal. Chapter 3 outlines the general structure and mechanism of neural networks, and the fundamental recursion scheme concepts. The next three chapters will then dive into particular examples of neural networks; each will begin by giving a more specific technical background of the named network, followed by a walk-through implementation using recursion schemes.

In Chapter 4 we introduce one of the most basic types of models, a fully connected network, and how this can modelled as a metamorphic recursion scheme. Chapter 5 moves onto a more difficult architecture, a convolutional neural network, and demonstrates its ability to be modelled as a metamorphism. Lastly, Chapter 6

introduces the most complex model of this project, deep long-short-term-memory networks; this presents a more elaborate recursive pattern than the last two examples. The final implementation combining all three chapters completes Catana.

In Chapter 7 we give a critical analysis of Catana, exploring whether the technical goals stated above have been achieved, and demonstrating results which prove that all three implementations can train from data. Finally, Chapter 8 summarises our findings, discussing if any impacts have been made towards achieving a more reliable, transparent and representational deep learning framework.

Chapter 2

Existing Research

Prior to diving into the background material which will lead us onto investigating the relationship between recursion schemes and specific neural network models, it would be useful to gain insight into any existing approaches to representing neural networks. I will discuss Tensorflow, the most popular deep learning framework, as well as Grenade, what I believe is the most accomplished deep learning functional programming library. I will then address the existing research relevant to both recursion schemes and neural networks.

2.1 Tensorflow

Tensorflow represents neural networks with data flow graphs [14]. These are a series of processing nodes that describe how data moves through a graph. Each node represents a mathematical operation, and each connection between nodes is a matrix representing the data consumed or produced by a computation. This allows the user to represent computations in terms of the dependencies between individual operations. The justification of using data flow graphs are benefits such as parallelism, distributed execution, and fast compilation. It is fairly clear that these benefits prioritise performance.

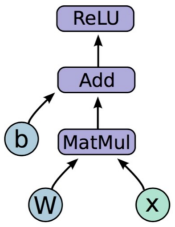


FIGURE 2.1: Tensorflow Data Flow Graph

However for researchers, Tensorflow is hard to learn and use. Research requires flexibility, and lack of flexibility is baked into Tensorflow at a deep level. For example, if a user were to extract the values of intermediate layers of a neural net, they would need to: define a graph, execute it with the data passed in as a dictionary, and add the intermediate layers as outputs of the graph. Another observation is that Tensorflow represents the layers of neural networks themselves as functions which perform forward propagation, and back propagation as an external function. I believe that a neural network should exist first as a data structure for algorithms to operate on, before existing as an algorithm itself; this is something which Catana engages with.

2.2 Grenade

Grenade is a Haskell library [4] which represents networks with heterogeneous lists of layers (lists containing different layer types), where their type includes not only the layers of the network, but also the shapes of data that are passed between the layers. This means that networks and layers are easily composed at the type level. One can use a trained Network as a small component in a larger network easily.

The definition of a network looks like so:

```
class UpdateLayer x ⇒ Layer x (i :: Shape) (o :: Shape) where
  -- Back propagation variables required for this layer
  type Tape x i o :: Type
  -- Take the input from the previous layer and gives the output
  -- from the current layer .
  runForwards     :: x → S i → (Tape x i o, S o)
  -- Takes the current layer , and back propagation variables to return
```

```
-- the gradient layer and derivatives .
runBackwards :: x → Tape x i o → S o → (Gradient x, S i)
```

Data types for specific layers are defined as instances of the class `Layer` which contains the functions `runForwards` and `runBackwards` to perform forward and back propagation. Creating an instance of `Layer x i o` says that the layer `x` can sensibly perform a transformation between the input and output shapes `i` and `o`. The strengths of Grenade's approach include the exploitation of composition and dependent types, leading to practical and concise specifications of complex networks in Haskell. The relationship between between layers and networks is exceptionally well designed.

Whilst this approach boasts various advantages, some of which Catana also offers, the process in how networks are trained do not employ/focus on recursive methodologies, let alone recursion schemes. Although the explicitness and propriety of recursion as a design choice varies is very situational, if used productively in combination with a functional programming language, the benefits seen can typically be: a reduction in boilerplate code, better expressiveness and elegance, being easier to reason about, and a more maintainable codebase. In certain situations, one of which I believe are neural networks, recursion just makes sense.

2.3 Recursion Schemes In Neural Networks

On seeking coverage of my proposal, I found that the concept of integrating recursion schemes into neural nets is one that has very little, if at all any, relevant formal research.

With regards to research publications, there exists the paper "Making Neural Programming Architectures Generalize via Recursion" [3], but it is unrelated to the functional programming realm and does not discuss recursion schemes; it focuses entirely on variations of assembly code to compare partially, fully, and non-recursive code. A slightly more relevant paper, "Understanding The Principles Of Recursive Neural Networks" [5], approaches the principles of recursive neural nets - but similarly to the previous paper, it solely investigates the idea of basic recursion.

The only two found mentions related to recursion schemes are limited to simple propositions. The first is from a short paper, "Applications Of Structured Recursion Schemes" [2], which reviews the concept of algebras and coalgebras in the context of category theory and how it gives rise to the formulation of structured recursion schemes over recursive structures. It briefly illustrates several examples of potential applications of recursion schemes, including neural networks. A comparison is made to compare a RNN (recursive neural network) to a catamorphism, and states "the connection to category theory has only recently been recognized, and detailed analysis in this context has not yet been carried out to the best of our knowledge."

The second mention is found in a blog post, "Neural Networks, Types, and Functional Programming" [12]. This is the most pertinent to my intended research. A number of neural nets are taken and compared to various functional programming concepts – e.g. it discusses the narratives of viewing 'Generating RNNs' as unfolds. Simultaneously, the writer states "I expect this idea is wrong, because most untested ideas are wrong. But it could be right, and I think it's worth talking about." This highlights the current state of the relationship between the areas of functional programming/recursion schemes and deep learning – it remains a virtually untouched discussion.

The most significant part of the dialogue the writer creates, is the momentary but inspiring resemblance made between 'Tree Nets' and 'Inverse Tree Nets' to the recursion schemes 'catamorphisms' and 'anamorphisms' - this recognises the presence of recursion schemes in the mechanism of these networks. Tree structures however, are a well known example for giving rise to the formulation of structured recursion schemes. Representing the training process of a Tree Net would not be too structurally dissimilar to a standard tree data type. I believe recursion schemes are not limited to such specific examples of neural nets which unmistakably exhibit a recursive pattern, but can potentially be applicable to all types of networks. Although it is not viable to exhaust every type of neural network, we will engage with three different notable models which do not explicitly display recursive behaviours, and use this as the basis that the mathematical structures which neural networks present can be exploited with recursion schemes.

Chapter 3

Background

We will first provide a very general outline of the structure and mechanisms of neural networks, which the following chapters will later provide detailed background into the specific types of models we encounter. Afterwards, we will gradually lead up to how recursion schemes work and the different components necessary to form a system which abstracts recursive behaviour away from a recursive data type and recursive function. Finally, we will consider different types of recursion schemes and discuss their relation to folds and unfolds, as well as methods of composing recursion schemes. This will provide us the tools to proceed with applying this to an example network in Chapter 4.

3.1 Neural Networks

Neural networks use different layers of mathematical processing to make sense of a specific type of information it is fed. A neural network is a set of connected neurons organized in layers, which can be structurally represented by a graph of nodes and edges. The types of layers consist of the input layer, the hidden layers, and the output layer. This can all can be structurally represented by a graph of nodes and edges. Given neurons in adjacent hidden layers, the connections between them each have their own hyperparameters, referred to as the weights and biases. The bias is the same for all connections with the same target neuron.

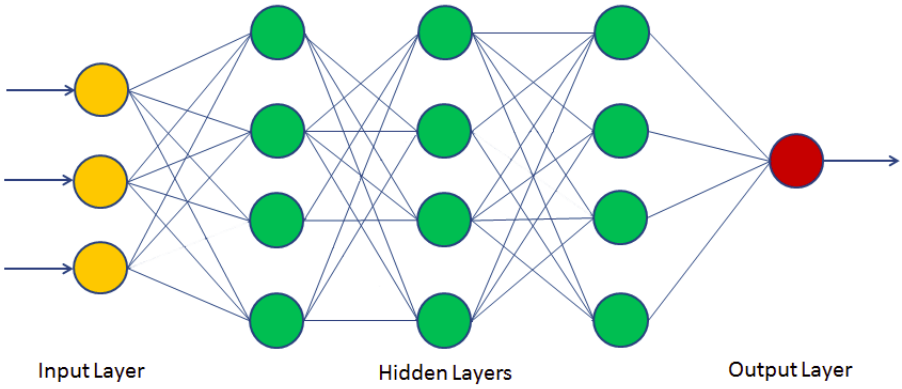


FIGURE 3.1: Basic Neural Network

The process of training a neural network aims to fine-tune the hyperparameters of the connections by repeatedly: taking an input sample, using the network to produce an output value, and then comparing the output to the desired output in order to update the weights and biases. This system can be deconstructed into two processes known as forward propagation and back propagation.

The process of forward propagation aims to apply a series of transformations to the provided input data, each of which corresponds to evaluating a certain feature of the data. This is generally described as the following: The input layer receives external information provided by a user; this is the data that the network aims to process and learn about. From the input layer, the data goes through one or more hidden layers - their purpose is to apply a series of transformations which the output layer can use. The output layer then produces the

final output values - by now, the transformations applied should have extracted certain features allowing the output data to be in a format which can be classified.

The process of back propagation aims to use the final output values to improve the neural network's hyperparameters. This is generally described as the following: During the training period, output is compared to the user provided desired output of what the input data should be classified as. The difference between the actual output and desired output is used to determine how the hyperparameters of the network should be updated. We process the network's layers in the backward direction and adjust its hyperparameters according to the ideal gradients which would cause the input to produce a result closer to the desired output.

Once the neural network has been trained with the significant amount of data, it will try to classify future data based on what it has observed previously.

3.2 Functors

Functors are useful concept which revolve around the idea that if we can prove that certain properties of a structure hold, then we are able to map a function over it. The function `fmap` is something that takes a function of type `a -> b` and a data structure of type `f a`, and applies this function to modify the data structure's internal values without affecting its underlying shape. There exists a typeclass (which is much like an interface in object oriented programming) called Functor. A data structure can be considered a functor if it has an instance defined for the Functor typeclass - this requires defining how `fmap` will operate on the data structure.

```
class Functor f where
    fmap :: (a → b) → f a → f b
```

All functors must be parameterized data types which means that the they must take another type as a parameter to exist as a concrete data type. For example, a list is a functor; however a list cannot exist just as a list, it needs to be a list of integers, chars or bools etc. This property can be observed by the parameter `a` on LHS of list.

```
data List a = Nil | Cons a (List a)
```

To define `List` as an instance of `Functor`, we define `fmap` over lists - this operates by: given a function, apply the function to the head element of the list, and then recurse by applying `fmap` over the tail of the list. The empty list acts as a base case for this recursion, where mapping a function over the empty list is equivalent to applying the identity function `id` which returns the original input untouched.

```
instance Functor List where
    fmap :: (a → b) → [a] → [b]
    fmap g (x:xs) = ((g x):(fmap g xs))
  f map g []     = []
```

To demonstrate, the below example maps a function which adds the integer one to every element in the provided list.

```
fmap (\x → x + 1) [0,1,2,3,4]  -- this returns  [1,2,3,4,5]
```

3.3 Folding and Unfolding

Given a list of numbers, consider the vast amount of things we can do with it: find its length, map a function over it, sum over it. Wouldn't it be great if there were a single function that generalizes what we can do with a list? In Haskell, there are higher-order functions called fold and unfold which encapsulate a pattern of structural recursion over recursive structures, and provide a generalization over functions on a specific data type. In other words, they allow us to work with recursive data structures without having to write recursive functions. In this section, we will describe how folds and unfolds operate which leads us on to how recursion schemes build on top of this.

Folds and unfolds are typically associated with lists, where a list of values can be evaluated down to a value, or a value can be taken to construct a list. However this of course can extend to further structures such as trees and vectors. A data type which is considered foldable or unfoldable must have its own instance defined for the type-class Foldable or Unfoldable respectively; this states that any recursive functions applied to the data type can also be equivalently represented by folds or unfolds. Below, we go into further detail into how folds and unfolds work.

3.3.1 Folds

Folds refer to evaluating a value from a structure by collapsing it. To perform a fold, we need a recursive data type, a seed value, and a binary function. A fold works by starting with the seed value as an accumulating value, and then traversing through the elements of the recursive data type - at each iteration, we apply our binary function to the current element we are on and the accumulated value to produce a new accumulated value. At the end, we are left with a final value which is the result of evaluating the provided data structure.

Below we define the function `foldr` for the data type list. This takes arguments: a binary function `g`, the accumulated value `seed`, and a list `list`. It states that we first pattern match on the list to check if it is non-empty `(x:xs)` or empty `[]`. If non empty, we apply the function `g` to both the head element of the list and the accumulated value to produce a new value. This is used along with the tail of the list to recurse by calling `foldr`. If empty, we simply return the final evaluated value.

```
foldr :: (a → b → b) → b → [a] → b
foldr g seed list =
    case list of (x:xs) → foldr g (g x seed) xs
                 []     → seed
```

For example, if we were to fold over the following list using the binary function addition `(+)` with a seed value of `0`, this would result in summing over the list.

```
foldr (+) 0 [1, 2, 3, 4, 5] = 15
```

3.3.2 Unfolds

Unfolds refer to generating a structure from a seed value. To perform an unfold, we need a seed value and a unary function which defines how to take an input to an output. An unfold works by starting with the seed value and applying the unary function to it to produce a new element which is added to an accumulating recursive data structure - this is repeated with each new element being used as the next value as an argument to the unary function until meeting a certain specified base case which terminates the process. At the end, we are left with a final construction of a data type.

Below we define the function `unfoldr` for the data type list. This takes arguments: a unary function (`h`) and the seed value (`seed`). It states that we first apply the function `h` to the seed to produce an output of type `Maybe (b, a)` - a value of type `Maybe (b, a)` can either be `Nothing` or `Just (b, a)`. We pattern match on the output; if it is of `Just (b, a)`, then we create a list where `b` is the head element and the tail is a recursive call of `unfoldr` on the new seed value `a`. If it is of `Nothing`, then the process terminates by returning an empty list.

```
unfoldr :: (a → Maybe (b, a)) → a → [b]
unfoldr h x =
    case (h x) of Just (b, a) → (b:(unfoldr h a))
                  Nothing     → []
```

For example, if we were to unfold using a seed value of 5 with the function `h` which given `x` returns `Just (x, x - 1)` if `x` is greater than 0, and `Nothing` otherwise, this results in the following list being constructed.

```
h :: Int → Just (Int, Int)
```

```
h x
  x > 0 = Just (x - 1, x) otherwise = Nothing

unfoldr h 5 = [5, 4, 3, 2, 1]
```

3.4 Components Of A Recursion Scheme

What fold and unfold have achieved is, by encapsulating the recursive behaviour inside its definition, it abstracts away the recursion from the core function we provide. This allows us to define our evaluative or generative function with non-recursive and clearer logic. However we are still left with the following issues:

1. The data structures we use to fold or unfold over are still defined with explicit recursion.
   `data List a = Cons x (List a) | Nil`
2. Defining a data structure to be foldable requires us to define a new instance of the typeclass Foldable every time. This means each fold and unfold definition is type-specific to a given data type.

Recursion schemes resolve both of these problems. They are composable combinators that automate the process of traversing and recursing through nested data structures. Before we introduce different examples of these, we will need to go through the necessary components which work in combination with recursion schemes. These consist of:

- A parameterized data type `f`. (Covered in 3.4.1)
- The fixpoint type `Fix`. (Covered in 3.4.2)
- An algebra and a coalgebra. (Covered in 3.4.3)

In the next few sections, we will cover what these entail.

3.4.1 Parameterized Data Types

This addresses the issue of having data structures which are explicitly recursive. Take the following recursive data type as an example which represents mathematical expressions. The multiply and add constructors, `Mul` and `Add`, are recursive, and the constant constructor `Const` behaves as the fix point.

```
data Expr = Mul Expr Expr  Add Expr Expr  Const Int
```

We can abstract away recursion from a recursive data type by rewriting it as a functor. This consists of taking any recursive parameters in the data constructors and changing them to an arbitrary parameter. We also need to add this same arbitrary parameter to the data type itself. During this, we also need to make sure that the data type is an instance of the functor type class, and also that it has a constructor which acts as a base case. This guarantees that mapping a function over the base case constructor should be equivalent to applying the identity function.

To demonstrate with the above example, let's create a functor equivalent called `ExprF`. Observe that all occurrences of `Expr` as a constructor parameter have been replaced with some type parameter `k`, and this `k` has been added to `ExprF` itself. This now states that `ExprF` can be an expression of any arbitrary type `k`. The power of this is that it allows us to express both a recursive expression like before, but also an non-recursive expression containing a value of some other type. We refer to the parameter `k` as the *carrier type*.

```
data ExprF k = MulF k k  AddF k k  ConstF Int
```

We now need to define its functor instance, which means defining how the mapping function `fmap` behaves over `ExprF`. Mapping a function `f` over any constructors containing the type parameter `k` will result in the `f` being applied to its parameters `k`. Mapping `f` over any constructors containing only concrete types, i.e. `ConstF Int`, should return the original expression untouched.

```
instance Num a ⇒ Functor ExprF where
  fmap f (MulF x y) = MulF (f x) (f y)
  fmap f (AddF x y) = AddF (f x) (f y)
  fmap f (ConstF i) = ConstF i
```

3.4.2 Fix

Least fixed points of functions refer to when the input to a function causes it to simply behave as the identity function, or put differently, a fixed point is a solution to the equation $x = f(x)$. In Haskell, we define this as the type `Fix` which takes a functor `f` as its type parameter, where a functor is any parameterized data type which can have a function mapped over it. It has one constructor *Fx* which lets us create a structure of type `Fix f` from type `f (Fix f)`. Additionally we define the function `unFix` which behaves as a deconstructor, unwrapping the constructor `Fx`.

```
newtype Fix f = Fx (f (Fix f))

unFix :: Functor f ⇒ Fix f → f a
unFix (Fx f) = f
```

`Fix` behaves as a wrapper around the functor, allowing an unknown depth of unravelling of `Fix f` with the predictability of outputting `f (Fix f)`. In recursion the relevance becomes clear as the least fixed point represents the point of convergence – i.e. an infinite number of compositions of the function to the fix point, will always return the original input value.

$$f(x) = x$$
$$f(f(f(x))) = x$$

At a first glance, this looks quite pointlessly recursive to no end, but what it allows us to do is to take a data type with a parameter, `f a`, and hide the parameter `a` at the type level. Removing the requirement to specify the type of `a` is incredibly useful as it is unviable to need to give a type definition of an arbitrarily nested structure.

```
Fix :: f a → Fix f
```

The appeal of `Fix` can be observed when using parameterized data types. To compare, we will first discuss examples of trying to define types of different possible values of `ExprF` without using `Fix`.

- Take a look at the following lengthy nested type definition for an instance of an `ExprF`. This is unpleasant to the eye and inviable to work with, especially during recursion as we cannot statically define a type definition which satisfies each level of the data structure we will traverse to.

  ```
  example :: ExprF (ExprF (ExprF (ExprF Int)))
  example = AddF (MulF (MulF (ConstF 5) (ConstF 3))
                       (AddF (ConstF 4) (ConstF 7)) )
                 (AddF (MulF (ConstF 15) (ConstF 9))
                       (AddF (ConstF 8) (ConstF 12)) )
  ```

- To give another example, how would we attempt to define the type for the following value?

  ```
  example :: ???
  example = AddF (MulF (ConstF 5) (ConstF 3)) (ConstF 2)
  ```

  The answer is that we simply cannot. According to the definition of `ExprF`, the constructor `AddF` should take two parameters of the same type. However in this case, we're providing a type `ExprF (ExprF Int)` on the LHS and a type `ExprF Int` on the RHS. This is clearly undesirable, as it should make sense for it to be interpreted as a correct expression.

Now let us introduce Fix into the equation.

- The usage of Fix as a wrapper is integral to recursion schemes, as it allows our functor to no longer require to state its type parameter. This prevents encountering 'infinite type' errors during the endless struggle to recursively satisfy Haskell's type system. Any possible instance of `ExprF` wrapped with `Fix` will always result in having the same type `Fix ExprF`.

```
example  = Fix ExprF
exampleF = Fx (AddF (Fx $ MulF (Fx $ ConstF 5) (Fx $ ConstF 3) ) (Fx $ ConstF 2) )
```

3.4.3 Algebras and Coalgebras

Remember when using folds and unfolds, we always had to provide a extra function which defines either how we evaluate or generate a data structure? Algebras and coalgebras are very much like these extra functions, except we use them with recursion schemes instead. The specifics of how we do this will be detailed later, but for now we will discuss what it means to be an algebra or coalgebra.

Algebras When concerned with folding or evaluating over a recursive data structure using recursion schemes, we use algebras. These are functions satisfying the type `f a → a` where `f` represents our functor and `a` represents our carrier type.

$$algebra :: f\ a \to a$$

Lets create an algebra for evaluating expressions of `ExprF Int` to an integer.

```
alg :: ExprF Int → Int
alg (ConstF x) = x
alg (AddF a b) = a + b
alg (MulF a b) = a * b
```

The fundamental part to notice about this is that our evaluating function is non-recursive and behaves as though the expression we evaluate `ExprF Int` is a fixed point where the parameter of `ExprF` is only an integer and thus contains no further expressions. This is in contrast to the below function `eval` which has to recursively evaluate expressions of the recursive data type `Expr`.

```
eval :: Expr → Int
eval (Const x) = x
eval (Add a b) = (eval a) + (eval b)
eval (Mul a b) = (eval a) * (eval b)
```

Coalgebras When concerned with unfolding over a seed value to generate a data structure using recursion schemes, we use coalgebras. It makes sense that coalgebras are functions which satisfy the reverse type definition of algebras: the type `a → f a`.

$$coalgebra :: a \to f\ a$$

Lets create a coalgebra for generating an expression of `ExprF Int` from an integer.

```
coalg :: Int → ExprF Int
coalg x
  x > 0 = AddF 1 (x - 1) x == 0        = ConstF 0
  x `mod` 2 = 0 = MulF 2 (x/2)
```

Of course there are many ways we could define a set a rules of how to generate an expression from an integer. The point of interest is that, like algebras, our coalgebra is a non-recursive function where the expression uses an integer as its carrier type. This is contrast to the following function which recursively generates expressions of the recursive data type `Expr`.

```
generate :: Int → Expr
generate x
  x > 0 = Add (Const 1) (generate (x - 1)) x  == 0       = Const 0
  x `mod` 2 = 0 = Mul (Const 2) (generate (x/2))
```

Overview It may seem odd that our algebra and coalgebra functions simply return a result immediately without recursing. Firstly, when using our algebra on a greatly nested instance of `ExprF`, the carrier of `ExprF` would not be an integer which gives us the type `ExprF Int`, but rather further expressions. Secondly, given an integer, applying our coalgebra would trivially generate the simplest expression possible containing only integers without producing further nested expressions. However, when combining these with specific recursion schemes, we are able to unveil an entire recursive process. In the next section we will bring examples of recursion schemes into the discussion and piece together all of the components that have led up to now.

3.5 Catamorphisms

As mentioned, fold allows us to evaluate recursive data structures without having to write recursive functions. If fold provides this generalization over recursive functions on a specific data type, then a catamorphism is a generalization over folds which lets us fold over any recursive data structure. Catamorphisms work with algebras to do this.

Below is the definition of a catamorphism in Haskell.

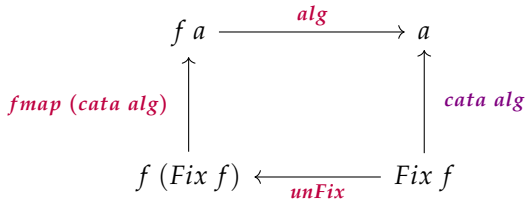
```
cata :: Functor f ⇒ (f a → a) → Fix f → a
cata alg = alg . fmap (cata alg) . unFix
```

This can seem a bit perplexing on a first encounter - lets look at the type definition. This says that the function `cata` takes as arguments a function of type `f a → a` and a value of type `Fix f`, to return us the type `a`. Conveniently, algebras satisfy the type `f a → a`, and the fix of a functor `f` satisfies the type `Fix f`. This means that given an algebra and the fix of a functor, a catamorphism will recursively evaluate the data structure `Fix f` to produce a value of type `a`.

Now lets look at how the function operates. First, observe that if only one argument has been provided to `cata`, namely the algebra, this is equivalent to returning a function of type `Fix f → a`. Secondly, `cata` is defined by the composition of three functions working from right to left. When `cata` is provided an algebra and a value of `Fix f`, we perform the following process:

1. Apply the function `unFix` to the data structure of type `Fix f`. This unwraps the `Fx` constructor from the current level of recursion of the data structure.
   This gives us the type `f (Fix f)`.
2. Map the function `cata alg` over the data structure of type `f (Fix f)` just produced. This performs the recursive behaviour, which applies the catamorphism process to the next level of the data structure
   This gives us the type `f a`.
3. Apply the algebra to the data structure of type `f a` just produced. This evaluates the current level of the data structure to a value.
   This gives us the type `a`.

Catamorphisms can actually be represented by the following diagram. Observe that there exist two paths from $Fix\ f$ to a; the purple path given by function `cata alg`, and the pink path given by a composition of functions `alg . fmap (cata alg) . unFix`. This exactly corresponds to the definition of function `cata`.



Example:

To illustrate how this would work, take the following algebra and example value of type `Fix ExprF`.

```
example :: Fix ExprF
example = Fx (AddF (Fx (ConstF 5)) (Fx (ConstF 3)))

alg :: ExprF Int → Int
alg (ConstF x) = x
alg (AddF a b) = a + b
alg (MulF a b) = a * b
```

When applying a catamorphism to evaluate this expression by calling `cata alg example`, a breakdown of how the data structure would look like at each step would look as follows:

Depth 1: We apply `cata alg` to `example`
i) `unFix` removes the `Fx` constructor
AddF (Fx (ConstF 5)) (Fx (ConstF 3))

Depth 2: `cata alg` is mapped onto `AddF`'s parameters.
i) unFix removes the Fx constructor
AddF (ConstF 5) (ConstF 3)

ii) `cata alg` is mapped onto `ConstF`'s parameters.
This is equivalent to the identity function, which returns the original input.
AddF (ConstF 5) (ConstF 3)

iii) `alg` is applied to `ConstF 5` and `ConstF 3`
AddF 5 3

Depth 1: We traverse to the previous layer.
i) `alg` is applied to `AddF 5 3`
8

Intuitively, the process of catamorphism can be viewed as a process of continually unwrapping each `Fx` constructor of an expression whilst diving into the next nested expression, until reaching a base case at which mapping a function is equivalent to applying the identity function. At this point, we then continually apply the function `alg` to each current expression whilst traversing back out of the nesting. This works because if every inner expression of the current expression has been evaluated, then this allows the current layer to be evaluated.

3.6 Anamorphisms

As described before, unfold allows us to generate recursive data structures from a seed value without having to write recursive functions. If unfold provides this generalization over recursive functions on a specific data type, then an anamorphism is a generalization over unfolds which lets us unfold over any recursive data structure. Anamorphisms work with coalgebras to do this.

Below is the definition of an anamorphism in Haskell.

```
ana :: Functor f ⇒ (a → f a) → a → Fix f
ana coalg = Fx . fmap (ana coalg) . coalg
```

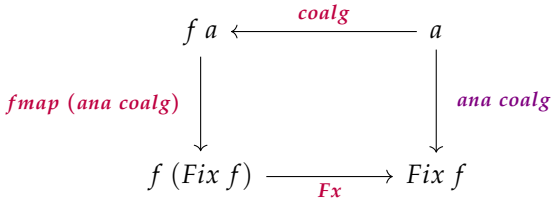
We can view the definition to be somewhat be the reverse behaviour of a catamorphism. Looking at the type level definition, we take a coalgebra of type `a → f a` and a seed value `a` to produce a data structure `Fix f`. Ana consists of a composition of three functions, which is described by the following process:

1. Apply the coalgebra first to the value of type `a` to create a data structure.
   This gives us the type `f a`.

2. Map the function `ana coalg` over the data structure `f a` just produced. This performs the recursive behaviour which will take the value `a` inside the data structure and apply the anamorphism process to it again.
   This gives us the type `f (Fix f)`.

3. Apply the constructor `Fx` to the data structure `f (Fix f)` just produced, wrapping it inside `Fx`.
   This gives us the type `Fix f`.

Anamorphisms can actually be represented by the following diagram. Observe that there exist two paths from a to $Fix\ f$; the purple path given by function `ana coalg`, and the pink path given by a composition of functions `Fx . fmap (ana coalg) . coalg`. This exactly corresponds to the definition of function `ana`.



Example:

To illustrate how this would work, take the following coalgebra and example integer.

```
example :: Int
example = 3

coalg :: Int → ExprF Int
coalg x
  x `mod` 2 = 1 = AddF 1 (x - 1) _          = ConstF x
```

When applying this coalgebra to generate an expression by calling `ana coalg example`, a breakdown of how the data structure would look like at each step would look as follows:

Depth 1: We apply `ana coalg` to `example`.
i) `coalg` is applied to `example`
AddF 1 2

Depth 2: `ana coalg` is mapped onto `AddF`'s parameters.
i) `coalg` is applied to each parameter of AddF
AddF (ConstF 1) (AddF 1 1)

Depth 3: `ana coalg` is mapped onto both `ConstF 1` and `AddF 1 1`.
i) `coalg` is applied to each parameter of `AddF 1 1`
AddF (ConstF 1) (AddF (ConstF 1) (ConstF 1))
ii) `Fx` is applied to each parameter of `AddF (ConstF 1) (ConstF 1)`
AddF (ConstF 1) (AddF (Fx (ConstF 1)) (Fx (ConstF 1)))

Depth 2: We traverse to the previous layer.
i) `Fx` is applied to `(ConstF 1) and (AddF (Fx (ConstF 1)) (Fx (ConstF 1)))`
AddF (Fx (ConstF 1)) (AddF (Fx (ConstF 1)) (Fx (ConstF 1)))

Depth 1: We traverse to the previous layer.
i) `Fx` is applied to the whole data structure
Fx (AddF (Fx (ConstF 1)) (AddF (Fx (ConstF 1)) (Fx (ConstF 1))))

Intuitively, the process of an anamorphism can be viewed as a process of continually generating a data structure from a seed value, and then mapping this same process over the parameters of the data structure until we reach a base case; at this point, mapping a function is equivalent to applying the identity function. We then continually apply the type constructor `Fx` to each structure whilst traversing back out of the nesting.

3.7 Metamorphisms

Catamorphisms and anamorphisms can actually compose. A metamorphism is a recursion scheme which refers to a catamorphism followed by an anamorphism - this means evaluating a data structure to a value, and

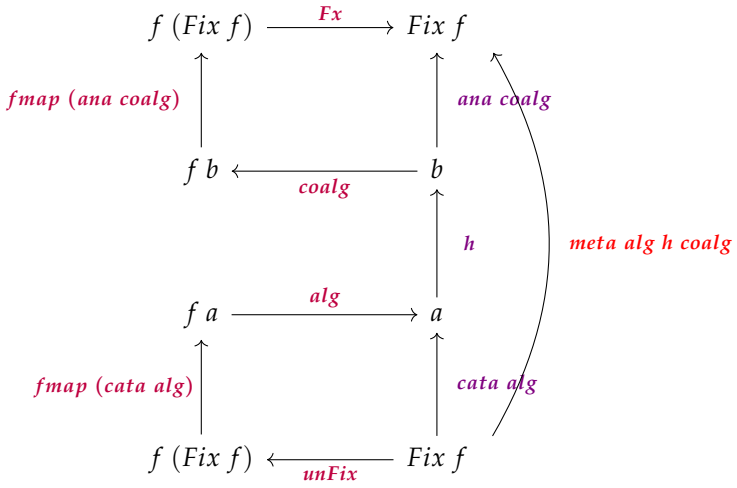
then taking this value to generate a new data structure.

The definition of metamorphism is given by:

```
meta :: Functor f ⇒ (f a → a) → (a → b) → (b → f b) → Fix f → Fix f
meta alg h coalg = ana coalg . h . cata alg
```

The definition may be slightly different than expected, as rather than just simply composing `ana` and `cata` to produce `ana coalg . cata alg`, we also have an intermediary function `h` of type `(a -> b)` in the middle of the composition. This changes the carrier type between each recursion scheme, which allows the carrier type `a` of the catamorphism to be different from the carrier type `b` of the anamorphism.

Metamorphisms can actually be represented by the following diagram. Observe that we have composed our diagrams for catamorphisms and anamorphisms with the intermediary function `h`. All possible paths which take us from the bottom right `Fix f` to the top right `Fix f` are equivalent in functionality. We say that this diagram is therefore commutative, such that all directed paths in the diagram with the same start and endpoints lead to the same result.



The concept of metamorphisms is essential to integrating recursion schemes with neural networks, which will be discussed in the next chapter.

3.8 Hylomorphisms

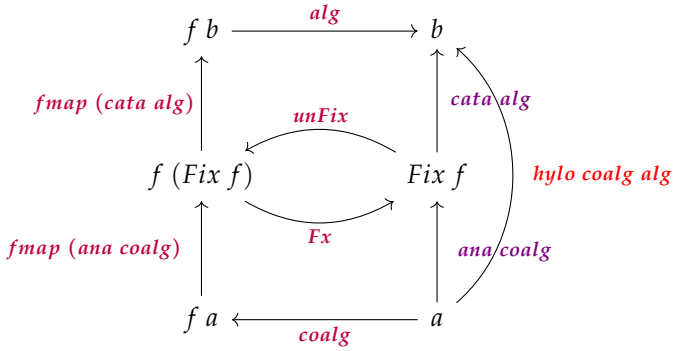
Catamorphisms and anamorphisms can also compose in the opposite direction. A hylomorphism is a recursion scheme which refers to a anamorphism followed by an catamorphism - this means generating a data structure from a seed value and then evaluating this data structure to a value.

The definition of hylomorphism is given by:

```
hylo :: Functor f ⇒ (a → f a) → (f b → b) → a → b
hylo coalg alg = cata alg . ana coalg
```

The definition depicts the composition of an anamorphism followed by a catamorphism, where each recursion scheme can use a different carrier type.

Hylomorphisms can be represented by the following diagram. Unsurprisingly, this is the reverse composition of the diagrams for catamorphisms and anamorphisms seen in the metamorphism diagram, except this time with no intermediary function `h`.



3.9 Summary

We have now explored all the fundamental components which allow us to construct recursion schemes, as well as discussed the recursion schemes which will be later used in this project: catamorphisms, anamorphisms, metamorphisms, and hylomorphisms. In the following chapters, we will investigate specific neural networks and see how these concepts find their way into the representation of neural nets.

Chapter 4

Defining Metamorphisms For Fully Connected Neural Networks

One of the most structurally simple types of neural nets are called 'feed-forward' networks, which have the property of being non-recursive in their process; i.e. there are no loops, meaning that data forward propagates in one direction. Although these types of models cannot be used to represent all other kinds of networks, the processes of forward and back propagation are a component shared amongst all networks, and feed-forward networks are an effective way of demonstrating the natural recursive patterns which any type of network is capable of displaying. In this chapter, we will define a recursion scheme which learns a fully connected network - the first network implementation for Catana. This will provide strong foundations to progress onto more difficult models and perhaps different recursive patterns.

We will start by introducing a background on how fully connected networks are structured and how forward and back propagation works when learning them. This will follow by attempting to find the best data type to represent the network which is compatible with using recursion schemes, and also facilitates the demands of the network learning process. From then, we will discuss how forward propagation can be implemented as a catamorphism (fold), and back propagation as an anamorphism (unfold). Finally we will link all of these components together by using metamorphisms to represent learning and also examine potential improvements to the implementation. During the entire development we will see that many problems are encountered along the way, and this requires our model to continually evolve to adapt.

4.1 Fully Connected Neural Networks

4.1.1 Structure

Fully connected neural networks are the amongst the most simple neural net models under the feed-forward category, after perceptrons. Their name refers to the property that each node in a layer is connected to all to nodes in the previous layer and all the nodes in the next layer.

As per convention, they are represented by graphs of nodes as neurons and edges as connections, grouped into belonging to a certain layers. These layers are stacked side by side and each connection between nodes in hidden layers have their own specified weight and bias as hyperparameters.

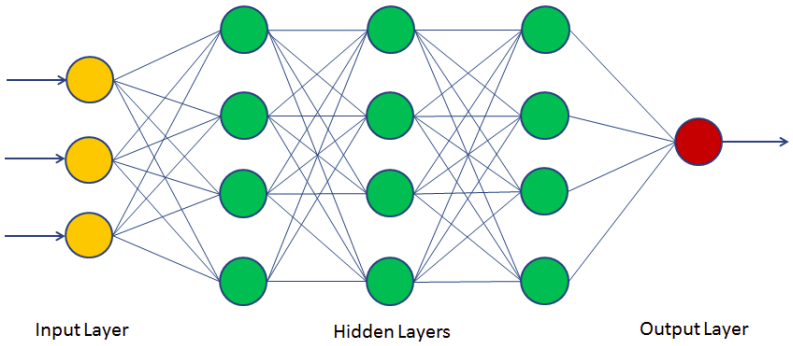


FIGURE 4.1: Fully Connected Neural Network

4.1.2 Forward Propagation

Forward propagation consists of first a set of input sample variables being provided to the input layer. This is then followed by the given iterated process:

1. The nodes in the current layer receive their input variables a^{L-1} from its previous layer.
2. These are then sent along each connected edge and is multiplied by the edge's weight w^L and then added on a bias b^L to produce the 'weighted output' z^{L+1}.
3. Lastly, an activation function is applied to all values. We will assume every layer uses the sigmoid function σ as its activation function. This yields the final output of current layer a^L.

These steps can be described by the following equations.

Weighted Output Of Layer L
$$z^{L+1} = w^L a^{L-1} + b^L \tag{1}$$

Sigmoid Function
$$\sigma(z) = \frac{1.0}{1.0 + exp(-z)} \tag{2}$$

Activated Output Of Layer L
$$a^L = \sigma(z^{L+1}) \tag{3}$$

4.1.3 Back Propagation

Back propagation is about finding the optimal change in weights and biases in a network using the cost produced from our error to reduce the cost the next time forward propagation is performed. Ultimately, this means computing the partial derivatives of the cost with respect to the weights and biases, but to compute those we must find an intermediate quantity delta δ^L which denotes the error from the lth layer - this leads to the reason why we traverse backwards when updating our hyperparameters. Computing the delta values of a given layer requires us to know the delta values of the next layer along, and this unravels into depending on knowing every subsequent layer's delta. This means that the required approach is to update each layer's hyperparameters in a productive direction from the output to the input layer, while passing back the current layer's delta.

We will now list the necessary equations used and then walk through the process of back propagation.

Let $\odot$ denote element-wise multiplication, and $\otimes$ denote the outer product. We choose the quadratic loss function as our means of calculating the error.

Equation for error in the output layer	$\delta^L = \partial C/\partial a^L \odot \sigma'(z^L)$	
which when using quadratic loss function is:	$= (a^L - y) \odot \sigma'(z^L)$	(1)
Equation for error in terms of the error in the next layer	$\delta^L = ((w^{L+1})^T \delta^{L+1}) \odot \sigma'(z^L)$	(2)
Equation for rate of change of cost with respect to the bias.	$\partial C/\partial b^L = \delta^L$	(3)
Equation for rate of change of cost with respect to the weight.	$\partial C/\partial w^L = a^{L-1}\delta^L$	(4)
Equation for weight update	$w^L = w^L - \eta \odot (a^{L-1} \otimes \delta^L)$	(5)
Equation for bias update	$b^L = b^L - \eta \odot \delta^L$	(6)

- **For from the output layer**, we're left with the vector of activation output values a^L and our desired output values y. We can compute the δ^L using (1), where z is the weighted input the layer takes prior to applying an activation function, and σ' is the inverse sigmoid function. This first δ value is then passed backwards to the previous layer.
- **For every other layer**, we follow an iterative process of computing each layer's δ, updating its weights and biases, and then passing back its δ to the previous layer which repeats the same process.

  Each layer's δ^L can be computed using the subsequent layer's δ^{L+1} (2), and then the partial derivatives for the cost with respect to the weights (4) and biases (3) are computed using this δ^L.

  To update the weights and biases, we use the calculated partial derivatives when next performing equations (5) and (6) respectively - here, η denotes the learning rate we choose which is a small constant. Finally, we pass δ^L to the previous layer.

4.2 Finding A Data Structure

The first component to address before the process of learning a network can be modelled, is a valid data structure which displays a potential recursive behaviour in our neural net, and adheres to the requirements of a data type previously discussed in 3.4.1. We intend this data type to be as representative of a neural network as possible; clearly this is something that will evolve over time as new requirements are encountered. We first discuss using neurons as a data type and then look at how using layers differ. During this we examine the recursive processes which each data structure exhibits and reason why trying to represent a neural network with neurons is much less desirable than when using layers.

4.2.1 Using Neurons As Recursive Structures

An initial choice may be to choose the neurons of the network to be our main data structure, using the concept of modelling the network as a tree. A tree is a well known foldable structure, and the similarities in the structures of a fully connected network and a tree can be observed from the fact that both are fundamentally a set of nodes belonging to distinct layers or depths, with connections occurring between nodes in adjacent layers. The leaves of the tree can be seen as the input nodes of the first layer, and the root of the tree as the output node in the last layer.

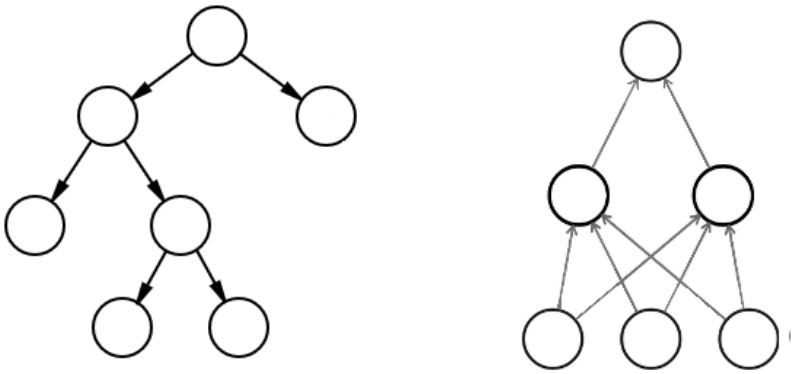


FIGURE 4.2: Binary Tree vs. Fully Connected Neural Net

A problem exists however, as each neuron in a neural net has to somehow contain references to both its connected set of input and output nodes in order to perform forward and back propagation. Unlike a binary tree where a node can either be a leaf or a node with two definite connections, a neuron has an unknown number of connections.

```
data Tree a = Node (Tree a) (Tree a)  Leaf a
```

One solution to this could be to instead have each neuron contain a two lists as parameters, one representing its set of input nodes and one representing its set of output nodes.

```
data Neuron k = Neuron [k] [k]
```

An basic proposal could thus be the following; let our data type be `Neuron` which takes a type parameter `k`. Its constructor `Neuron` then takes a list of output connections to neurons in the next layer, with each connection represented by a 3-tuple containing a weight, bias and a value of type `k`.

As with every other recursive data structure, we require a constructor to act as a base-case so that mapping a function over it is equivalent to applying the identity function, allowing the recursion to terminate. We therefore need to consider whether to represent this with an input neuron or an output neuron.

Let's first consider using an output neuron as a base case. Our forward propagation would then be defined to work on a structure where the output neurons are nested inside a stack of previous hidden neurons. We intend to use catamorphisms to perform forward propagation. From how catamorphisms function, it will recursively `fmap` itself deeper inside the structure until hitting a terminating point – only then will the algebra we provide it actually be used. This means that the algebra will first be applied to the output neuron, but this is completely useless - the output neuron has nothing passed to it yet from its previous neurons. In other words the evaluation would be performed the wrong way round. We instead need to apply the algebra to the input neuron first. Therefore choosing the input neurons as a base case makes sense; this also follows how trees are folded over as their leaf nodes act as their base case.

The input neuron will also take one of the input values of the network as a parameter in order to have access to them in the network. For clarity, we additionally create type synonyms to define the weights, biases and inputs/outputs as doubles.

```
type Weight   = Double
type Bias     = Double
type Input    = Double
data Neuron k = Neuron [(Weight, Bias, k)]  InputNeuron Inputderiving Functor
```

However this introduces a number of problems.

1. The number of neurons realistically used in a network would result in an example network being incomprehensible and awkward – this is illustrated by the fact that any two neurons which are connected to the same neuron in the next layer means that both would need to contain the same copy of that neuron. This is an approach which entails unnecessary repetitions of values and updates.

2. Trying to correctly handle the path of traversal through a neural net as a non-binary tree is complex and the nesting of the data structure will grow to become disorganized.

4.2.2 Using Layers As Recursive Structures

This all points to the idea that an entirely different data type is needed – one which represents a layer of the network. A layer would only require to contain another single layer, meaning no copies of layers are shared as parameters - this would allow for a perfect nesting in the data structure, and completely removes the previous dilemma had with using neurons. Given all the layers are stacked side-by-side and processed consecutively, using layers as a data type in a recursion scheme is comparable to using a list. If we imagine a list where its elements are layers, folding over a neural network's layers is essentially the same structural process as folding over a list.

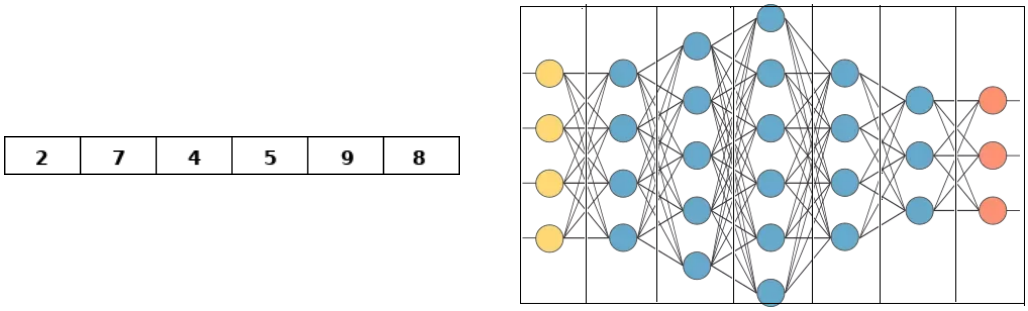


FIGURE 4.3: Visualizing A Neural Network As A List of Layers

The next question, how can a single layer represent every neuron and its connective properties? If we take a step back, viewing neurons as concrete data types when visualizing a neural network is ultimately unimportant, as we can abstract away from the idea of nodes and edges entirely and ignore them as types. The neurons and connections between each layer can be implicitly captured and represented using matrices containing the weights and biases. The weights of the connections between two layers can be stored in a 2D matrix, with each internal list representing all the neurons' weights of the first layer which are associated with one of the nodes in the second layer. The biases and also the inputs of a layer can then be depicted as a vector.

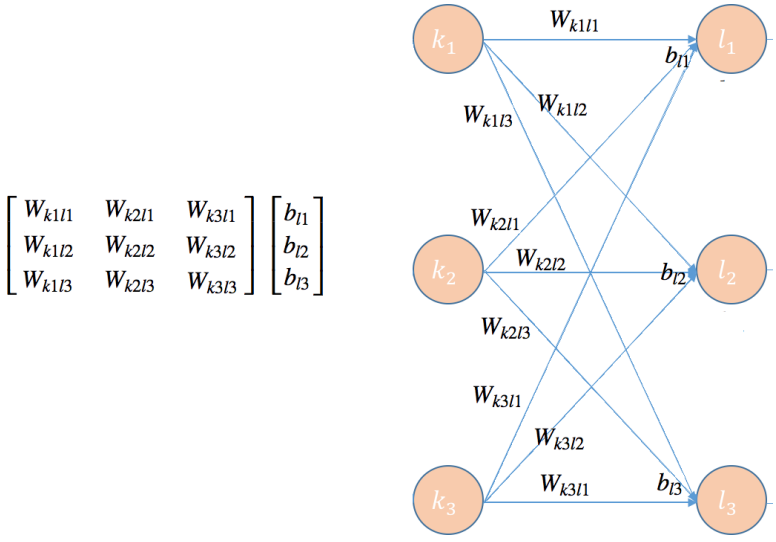


FIGURE 4.4: A Layer's Hyperparameters As Matrices

We shall redefine the types of weights to be a 2D list, and the biases and input to be a list. The data type of a Layer is similar to that of Neuron, except that the Layer constructor no longer require a list to represent

multiple neurons.

```
type Weights = [[Double]]
type Biases  = [Double]
type Input   = [Double]
type Output  = [Double]
data Layer k = Layer Weights Biases k  InputLayer Inputderiving Functor
```

This looks very promising and simultaneously elegant – it exhibits an example of how the labels or components of a structure can be completely abstracted over and generalized, leaving a data type which is little more than a functor carrying a matrix and vector. It is now possible to visualize an example of the network; this naturally requires wrapping each Layer inside a Fix type.

```
type Fix f  = Fx (f (Fix f))

example :: Fix Layer
example = Fx (Layer [[2.5,  3.0],[0.1,  3.9]]  [1.0,  1.0]
               (Fx $ Layer  [[1.0,  0.0],  [8.3,  2.0]]  [1.0,  1.0]
                      (Fx $ InputLayer  []) ) )
```

We now have a data type which allows us to experiment with how we can forward propagate over it using catamorphisms, and later on back propagate using anamorphisms.

4.3 Implementing Forward Propagation

We aim to model forward propagation as a catamorphism such that given an input, a neural network can be evaluated to produce an output. Given the previous description of how forward propagation works, we will begin defining an algebra which represents it.

4.3.1 Defining An Algebra

To model forward propagation as a catamorphism, we need to begin with implementing an algebra which represents the logic which occurs when forward propagation occurs over a single layer. This means defining a function satisfying the type `f a → a`. What needs to be decided is what the carrier type `a` a should be - however this is not so straightforward. As an initial idea, if each layer takes a list of inputs to produce a list of outputs, it makes sense that our carrier type a will be a list of doubles. We are therefore looking to define a function using the below template.

```
alg :: Layer Input → Input
alg (Layer weights biases k) = ...
alg (InputLayer inputs)      = ...
```

1. Input Layer

Pattern matching on the input layer constructor is very straightforward; the algebra should forward propagate the input data give to the input layer by simply outputting the input data to the next layer. The input layer being the fixed point means we should simply return its `inputs` parameters.

```
alg (InputLayer input)      = input
```

2. Layer

Lets now create complete a pattern match for the layer constructor; the algebra should forward propagate the input data given to the layer by: multiplying it by the weights, adding the biases, and then applying the activation function to give an output to the next layer.

First we will focus on computing the weighted output of a layer.

Weighted Output Of Layer L

$$z^{L+1} = w^L a^{L-1} + b^L \tag{1}$$

Recall that each of the inner lists (rows) in the type `Weights` denotes the weights of all the connections towards a specific output node. The function `zipWith` allows us to take two lists and apply a binary operation between their corresponding elements to produce a new list. For each of the weight matrix rows, we use `zipWith` between the row and the input array using the multiplication operator, followed by a summation over the resulting list.

$w^L * a^L$ = map (**sum** . (**zipWith** (*) input)) weights

Afterwards we are left a single list. We again apply `zipWith` between it and the biases but this time using the addition operator.

$w^L * a^L + b^L$ = **zipWith** (+) biases (map (**sum** . (**zipWith** (*) input)) weights)

We finish off by applying the sigmoid function as an activation function to each value; this gives the final output of the layer.

Sigmoid Function
$$\sigma(z) = \frac{1.0}{1.0 + exp(-z)} \tag{2}$$

Activated Output Of Layer L
$$a^L = \sigma(z^{L+1}) \tag{3}$$

This can be done by defining a sigmoid function and mapping it over the previous result. This leaves us with the following code, where we define the entire process as the function `forward`.

```
sigmoid :: Double → Double
sigmoid x = 1.0 / (1.0 + exp (negate x))

forward :: Weights → Biases → Input → Input
forward weights biases inputs
    = map sigmoid $ zipWith (+) biases (map ( sum . (zipWith (*) input)) weights)
```

Our definition of a complete basic algebra for forward propagation between layers can be seen below, where applying `alg` to the pattern match `Layer` consists of calling the function `forward` on the layer's parameters. The fundamental achievement here is that the core behaviour of forward propagation has been represented incredibly simply and is pleasingly coherent.

```
alg :: Layer Input → Input
alg (Layer weights biases k) = forward weights biases k
alg (InputLayer input)       = input
```

4.3.2 Introducing Statefulness To Our Algebra

Although we are currently focused on modelling forward propagation as catamorphisms, when planning ahead for back propagation it can be realised that a significant complication occurs. After evaluating the entire network to a single list of output values, how do we remember what the network structure was in order to rebuild it during back propagation? We cannot define a general list of rules for the coalgebra to follow on how to construct a network unless we have access to the original network. The issue now becomes finding a way of making our algebra stateful.

One idea is to try and give each layer a reference to its previous layer, so that when we arrive at the final layer, we can recursively find our way back to the first layer. However it would be unwise to do this by trying to modify the data type of Layer to have an additional parameter to store the previous layer, as this runs into complications of replicating layers and difficult nesting. We want the layer data type to be truly representative of a layer in a neural network, which means not resorting to short-cuts which are antagonistic to what we intend to achieve using recursion schemes and functional programming - a minimalistic and coherent system. So what if we instead change the carrier type of `Layer` to store extra information in addition to an input data? If our algebra returned a value that also holds the current layer we are evaluating, this would suffice as being a 'pointer' to the previous layer.

Let our new carrier type therefore be the tuple `(Fix Layer, Input)`. Note that we need to store this layer as the type `Fix Layer` rather than the type `Layer`, as we need to hide its carrier type to avoid complications at the type-level. The algebra can be be defined as below.

```
alg :: Layer (Fix Layer, Input) → (Fix Layer, Input)
alg (Layer weights biases (innerLayer, input))
    = let output = forward weights biases input
      in (Fx (Layer weights biases innerLayer), output)
alg (InputLayer input)
    = (Fx (InputLayer input), input)
```

Here we have renamed the carrier as `(innerLayer, input)`. We refer to the new variable as `innerLayer` which is also synonymous to being the 'previous layer'.

This new algebra successfully returns both the fix of the layer being evaluated, as well as the value evaluated from it. When dealing with the pattern match for `Layer`, we additionally change the carrier from being `(innerLayer, input)` to be just `innerLayer`. This change in carrier type is necessary because when later working on a type of `Fix Layer`, we cannot infer that its carrier is a tuple due to the definition of Fix enforcing that `Fix f = f (Fix f)` holds. This algebra now represents forward propagation in a fully connected layer; we can now consider this suitable enough to begin implementing back propagation.

4.4 Implementing Back Propagation

Now working in the opposite direction, we look to model back propagation as an anamorphism which constructs an updated neural network from the output of the catamorphism. Using the back propagation algorithm, we can begin recognizing the requirements which the algorithm imposes on our model and resolve this. Afterwards we will start defining a suitable coalgebra for our anamorphism.

4.4.1 Resolving Access Issues To Missing Back Propagation Data

When writing a coalgebra, recall that we intend to define a function of type `(a → f a)` which in this specific context might look like `(Fix Layer, Input) → Layer (Fix Layer, Input)` if we directly use the output of the catamorphism as the seed value for `a`. However, when inspecting what variables we need access to in the back propagation algorithm, there are evident issues in this choice of carrier type which prevent us creating a coalgebra.

Firstly, we don't have access to each layer's input a^{L-1} and output a^L as required by the algorithm, due to our algebra currently only allowing us access to the previous layer and the very final output.

Error in the output layer
$$\delta^L = (a^L - y) \odot \sigma'(z^L) \tag{1}$$

Error in all other layers
$$\delta^L = ((w^{L+1})^T \delta^{L+1}) \odot \sigma'(z^L) \tag{2}$$

This demands that we backtrack back to our algebra to come up with a suitable modification which enables us to progress with our coalgebra. There's an elegant way to deal with this - what if we instead changed the carrier type so that we forward propagate a list of all outputs/inputs produced? At each level of recursion during the catamorphism, we would append a new output to the list, resulting in a stack of all the outputs/inputs after evaluating the final layer. This change would result in the following algebra:

```
alg :: Layer (Fix Layer, [Input]) → (Fix Layer, [Input])
alg (Layer weights biases (innerLayer, inputStack))
    = let input = head inputStack
          output = forward weights biases input
      in (Fx (Layer weights biases innerLayer), (output:inputStack))
alg (InputLayer input)
    = (Fx (InputLayer input), [input])
```

Secondly, we lack a way of back propagating the next layer's delta δ^{L+1} and its weights w^{L+1}. We'll therefore define a friendly data type which represents the back propagation variables.

```
type Deltas    = [Double]
data BackProp = BackProp { outerWeights :: Weights,
                           outerDeltas  :: Deltas,
                           desiredOutput :: Output}
```

At last, the new data type for back propagation and the modified algebra suffice to allow us to progress with defining a coalgebra.

4.4.2 Defining A Coalgebra

We begin by defining a function adhering to the coalgebra type `a → f a`. However the coalgebra does not need to use the same carrier as our algebra did - we can conveniently use an intermediary function after the algebra to change the carrier type. We need this new carrier to hold three things: the back propagation data type, the accumulated inputs during forward propagation, and the stored reference to the previous layer. This leaves us with the following type definition for the coalgebra.

```
coalg :: (Fix Layer, [Input], BackProp) → Layer (Fix Layer, [Input], BackProp)
```

We can break down the coalgebra into three main steps. At every layer we need to find:

1. Computing the delta error δ^L
2. Computing the updated weights w^L and biases b^L
3. Returning the layer with updated hyperparameters and carrier

The new weights and biases computed will replace the old ones in the layer, and the old weights and the computed delta will need to be stored in the `BackProp` data type to be passed backwards to the previous layer. It is also important to remember that when traversing through layers, we need to correctly recognize which of the values in our list of inputs corresponds to the current layer's inputs and outputs. Hence when back propagating through each layer, we will only return the tail of list of inputs. This ensures that the first element will always be the current layer's output a^L and the second element will always be the current layer's input a^{L-1}.

The template of the coalgebra will therefore take the following structure, illustrating the necessary steps to be performed when back propagating.

```
coalg :: (Fix Layer, [Input], BackProp) → Layer (Fix Layer, [Input], BackProp)
coalg (Fx (Layer weights biases innerLayer), inputStack, backPropData)
    = let (output:input:xs) = inputs
          delta             = ...
          updatedWeights    = ...
          updatedBiases     = ...
          updatedBackProp   = backPropData {outerWeights = updatedWeights,
                                            outerDeltas  = delta}
      in Layer updatedWeights updatedBiases (Fx innerLayer, tail inputStack, updatedBackProp)
```

We will now go through each of the three steps to implement and integrate the necessary equations into the coalgebra.

1. Computing The Delta Error δ^L

We start by implementing the function which computes delta error.

Error in the output layer
$$\delta^L = (a^L - y) \circ \sigma'(z^L) \tag{1}$$

Error in all other layers
$$\delta^L = ((w^{L+1})^T \circ \delta^{L+1}) \circ \sigma'(z^L) \tag{2}$$

First note that the equation to compute delta does not use the layer's input a^L, but rather it takes the value z^L which is the weighted input *before* an activation function has been applied. This can be found by defining the inverse sigmoid function σ^{-1}, such that $\sigma^{-1}(a^L) = z^L$. This is given by $\sigma^{-1}(x) = log(\frac{x}{1-x})$.

Second, we need to define the function σ' in the above equations - this denotes the differential sigmoid function. This is given by $\sigma'(x) = \sigma(x) * (1 - \sigma(x))$.

```
invSigmoid :: Double → Double
invSigmoid x = log(x/(1.0 − x))

sigmoid' :: Double → Double
sigmoid' x = let y = sigmoid x
             in  y * (1.0 − y)
```

Third, observe that the equation to compute delta is different for the output layer (1) than it is for all other layers (2). This means we need a way to detect if we're at the output layer or not, seeing as the data type `Layer` does not allow us to distinguish the output layer from the other hidden layers. Therefore, let's state that the value of `BackProp` that the output layer receives will have an empty list initialised as its `outerDelta` field.

We define this function as `computeDelta` where we have used some helper functions for matrix linear algebra defined elsewhere to reduce boilerplate code - these are for example, element-wise vector operations and matrix-vector multiplication.

```
computeDelta :: [Input] → BackProp → Deltas
computeDelta inputStack (BackProp outerWeights outerDeltas desiredOutput)
    = let (output:input:xs) = inputStack
          sigmoid'_z = map (sigmoid' . invSigmoid) input
      in  case outerDeltas of [] → elemul (elesub output desiredOutput) sigmoid'_z
                              ys → elemul (mvmul (transpose outerWeights) outerDeltas) sigmoid'_z
```

Given the list of inputs `inputStack` and the back propagation variables, we first extract the layer's output and input data from `inputStack`. We then acquire $\sigma'(z^L)$ by mapping the composition of the differential sigmoid function and inverse sigmoid function over the input - this is denoted as `sigmoid'_z`.

We can then pattern match on the `outerDeltas` value; if this identifies as the empty list, then we recognise that we should use equation (1) for the output layer by subtracting the desired output from the output. Otherwise, we use equation (2) for the other layers by multiplying the transpose of the next layer's weights by the next layer's delta. In both pattern matches, we end by multiplying the result with `sigmoid_z` to return delta.

2. Updating Weights w^L And Biases b^L

To update the weights and biases, we must define the functions which give us the gradients of the cost with respect to the weights and biases. The gradient with respect to the bias (3) is simply the delta value we just computed. The gradient with respect to the weights (4) is the outer product between the inputs and the delta, which produces a matrix.

Rate of change of cost w.r.t the bias. $$\partial C / \partial b^L = \delta^L \tag{3}$$

Rate of change of cost w.r.t the weight. $$\partial C / \partial w^L = a^{L-1} \otimes \delta^L \tag{4}$$

We can then use these gradients to update our hyperparameters, by multiplying the gradients with a chosen learning rate η and subtracting them from the original respective hyperparameters.

Equation for weight update $$w^L = w^L - \eta \odot (a^{L-1} \otimes \delta^L) \tag{5}$$

Equation for bias update $$b^L = b^L - \eta \odot \delta^L \tag{6}$$

We will define a function `backward` which computes these new hyperparameters, and also calls `computeDelta` to return δ^L.

```
backward :: Weights → Biases → [Input] → BackProp → (Weights, Biases, Deltas)
backward weights biases inputStack backPropData
    = let learning_rate      = 0.2
          (output:input:xs)  = inputStack
          delta              = computeDelta inputStack backPropData
```

```
        biasGradient     = delta                                          -- (3)
        weightGradient   = outerproduct input delta                       -- (4)
        updatedWeights   = elesubm weights (map2 (learningRate *) weightGradient)  -- (5)
        updatedBiases    = elesub biases (map (learning_rate *) delta)    -- (6)
    in  (updatedWeights, updatedBiases, delta)
```

This takes the existing weights and biases, the input stack, and the back propagation variables. We initialise our learning rate η to `0.2` and acquire the delta error by calling `computeDelta`. The next few lines translate the four above equations directly, as can be seen labelled in the code comments. This returns a tuple containing the current layer's new weights and biases as well as the delta to be passed on to the previous layer.

3. Returning The Layer With Updated Hyperparameters And Carrier

Finally we bring everything together to complete the coalgebra.

```
coalg :: (Fix Layer, [Input], BackProp) → Layer (Fix Layer, [Input], BackProp)
coalg (Fx (Layer weights biases innerLayer), inputStack, backPropData)
    = let (updatedWeights, updatedBiases, delta) = backward weights biases inputStack backPropData
          updatedBackProp = backPropData {outerWeights = weights,
                                          outerDeltas  = delta}
      in Layer updatedWeights updatedBiases (Fx innerLayer, tail inputStack, updatedBackProp)
```

Calling the function `backward` gives us the new weights and biases and the computed delta. Using this, we update the back propagation data with the computed delta and the layer's old weights, and update the layer with new weights and biases. This coalgebra now represents back propagation in a fully connected layer.

4.5 Training A Fully Connected Neural Network

We have now successfully implemented forward and back propagation as a catamorphism and anamorphism respectively. What's left to do is connect the two processes to model the training of a neural network as a metamorphism. Afterwards, we will examine potential improvements to the model in how to be better representative of a neural net and its behaviour.

4.5.1 Constructing The Metamorphism

We give the definition of a metamorphism below, which shows the composition of a catamorphism, an intermediary function, and an anamorphism.

```
meta :: Functor f ⇒ (b → f b) → (a → b) → (f a → a) → a → Fix f
meta coalg h alg = ana coalg . h . cata alg
```

First recall that the carrier of the catamorphism uses type `(Fix Layer, [Input])` whereas the anamorphism's carrier type is `(Fix Layer, [Input], BackProp)`. In order for the composition of the catamorphism and anamorphism to occur, we need to define the intermediary function `h` which changes the carrier type between the execution of these two processes. The entire metamorphism and the intermediary function `h` will be defined inside a wrapper function `run_sample` which takes a neural network, the input data, and the desired output data.

```
run_sample :: Fix Layer → Output → Fix Layer
run_sample neural_net desired_output
    = meta alg h coalg $ neural_net
      where h :: (Fix Layer, [Input]) → (Fix Layer, [Input], BackProp)
            h (nn, inputStack) = (nn, inputStack, BackProp [[]] [] desiredOutput)
```

The intermediary function `h` takes the output of the catamorphism, creates an initial `BackProp` value containing empty list values for the outerWeights and outerDeltas fields, and then combines all of these values to form a tuple matching the carrier type of the anamorphism. This can now be used inside the metamorphism.

The metamorphism represents a function which when given a neural network (containing the input data inside the input layer constructor) and the desired output, it performs forward propagation and back propagation to update the network's hyperparameters according to the error between the actual output and the desired

output. On the bright side, it's functional. There's still some clumsiness in the whole process that needs to be refined though before we can be satisfied with the code.

4.5.2 Improving The Model

We intend to achieve a true representation of the neural net through the data type for layers. However, the way the initial inputs and the desired outputs are treated is very illogical; it doesn't really make any sense to pass the desired output as a parameter to `run_sample` but embed the input variables inside the `InputLayer` constructor. At the moment the definitions infer that there is little or no relationship between two components, whereas they should instead be inextricably linked.

It's a bit odd for `InputLayer` to have to contain the actual input we give it. The neural network should be able to exist independently of a provided input, rather it should wait to be given an input. In other words, it is more appropriate for the neural net to behave as a function which takes an input to an output. Although one could argue that we could create a function which inserts a new input to the parameter for every sample we give it, this isn't the most advantageous usage of functional programming and is laborious to have to reach inside the data structure every time.

Therefore, let us first change the data type for `Layer` by removing the input parameter from the constructor `InputLayer`.

```
data Layer k = Layer Weights Biases k | InputLayer
```

We'll then change the carrier type for the algebra to hold a function which allows us to isolate the input variables from the data structure. At the moment the carrier type we use during forward propagation is `(Fix Layer, [Input])`. To make this behave as a function, we change `[Input]` to be of type `[Input] → [Input]`. This means that the catamorphism can be run on a neural net without any input variables actually being provided. If each layer is evaluated to return a function of kind `(* → *)`, this would lead to a composition of all of these functions starting from the input layer.

The algebra would take a function of type `[Input] → [Input]` to compose with a new function which when given the output of the previous function, produces a new list of outputs. This results in a chaining of forward propagation functions, each of which uses the previous function's result to define its output. What we end up with from evaluating a neural network, is a single function of type `[Input] → [Input]`; this can take the initial input data and return us the list of outputs produced by each layer.

So lets again edit the carrier type to be `(Fix Layer, [Input] → [Input])` and redefine our algebra. We refer to this new function as `forwardPass`.

```
alg :: Layer (Fix Layer, [Input] → [Input]) → (Fix Layer, [Input] → [Input])
alg (Layer weights biases (innerLayer, forwardPass))
    = let newForwardPass = (\inputStack →
                                let input = head inputStack
                                in ((forward weights biases input):inputStack)) . forwardPass
      in (Fx (Layer weights biases innerLayer), newForwardPass)
alg InputLayer
    = (Fx InputLayer, id)
```

When pattern matching on the constructor `Layer`, we compose the existing forward pass function that has been returned to us from the inner layer, with a new function which awaits an input stack so that it can return a updated one. When pattern matching on the constructor `InputLayer`, this should return the identity function so that we can pass it an input sample and expect it returned unmodified.

All we need to do now is redefine how the catamorphism and anamorphism are connected in the function `run_sample`.

```
run_sample :: Fix Layer → Input → Output → Fix Layer
run_sample neural_net sample desired_output
    = meta alg h coalg $ neural_net
```

```
    where h :: (Fix Layer, [Input] → [Input]) → (Fix Layer, [Input], BackProp)
          h (nn, forwardPass) = (nn, (forwardPass [sample]), BackPropData [[]] [] desired_output)
```

We can see that the forward pass function is extracted from running the catamorphism, and is then applied to a sample we give it in order to receive the input stack.

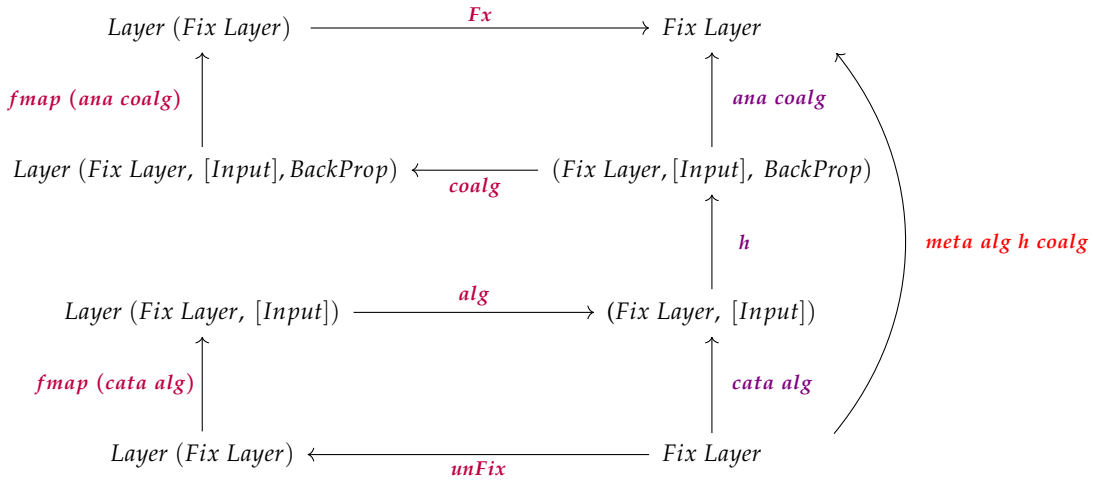
To finish off, realistically we should be able to run multiple samples through the network, rather than just a single sample which `run_sample` performs. To do this, we will define the function `train`.

```
train :: Fix Layer → [Input] → [Output] → Fix Layer
train neural_net samples desired_outputs
  = foldr (\(sample, desired_output) nn →
                  train nn sample desired_output) neural_net training_data
    where training_data = zip samples desired_outputs
```

This takes the lists of samples and desired outputs, and evaluates the network iteratively through folding over the lists using the function `run_sample`.

• Summary

We have now completed our implementation of the first neural network, given by a metamorphism whose catamorphism evaluates the network to represent it as a function which performs forward propagation, and an anamorphism which takes the results of forward propagation to reconstruct an updated neural network. This can be illustrated by taking the metamorphism diagram and tailoring its types to our fully connected network implementation.



Chapter 5

Defining Metamorphisms For Convolutional Neural Networks

Having successfully modelled the learning of a fully connected network as a metamorphism, we will now progress onto a complex type of network in order to see how the end implementation compares. A convolutional neural network is still a feed-forward network in that data propagates in a single direction with no loops, however the structure is slightly less straight-forward than fully connected networks, and it can be difficult to visually compare them to a graph of neurons and edges. They let us handle higher dimensional data and are therefore well known for analysing image, audio and video data.

We begin by describing the background of convolutional neural networks in how they are structured and how propagation differs from the previous example. As before, we then find an adequate data type to represent the network layer structure. Afterwards, we discuss how forward propagation is implemented as a catamorphism and back propagation as an anamorphism, illustrating a common pattern of recursion schemes which seems to exist within the process of learning a network. Finally we connect all of our components to model a metamorphism and evaluate what our progress and what it has demonstrated.

5.1 Convolutional Neural Networks

5.1.1 Structure

As convolutional neural networks are feed forward networks, their layers are stacked side by side with the input flowing strictly from the input layer to the output layer. However there are 5 different layers we need to become accustomed to:

- **Convolutional Layer**
  This contains a number of filters which it uses to perform convolution over the input data given to recognize patterns.
- **ReLu Layer**
  This is an application of the function $f(x) = max(0, x)$ which speeds up training.
- **Pooling Layer**
  This groups up elements in the input data to extract the max value, to help the network be invariant to small perturbations of the data.
- **Fully Connected Layer**
  This reshapes the input data into a vector of values corresponding to the probabilities of belonging to a certain class.
- **Input Layer**
  This simply passes forward the input data to the next layer.

Every convolutional neural network always begins with the input layer and ends with the fully connected layer. Between these two layers, any combination of the convolutional layer, relu layer, and pooling layer, can

be used.

With regards to the format of data and the layers themselves, rather than dealing with individual neurons in layers and a number of input variables, it is easier to visualize the input as an image in the form of a 3D matrix (although can be extended to higher dimensions), and each layer as being associated with applying a different operation on the input image. This will become clearer when later describing forward propagation.

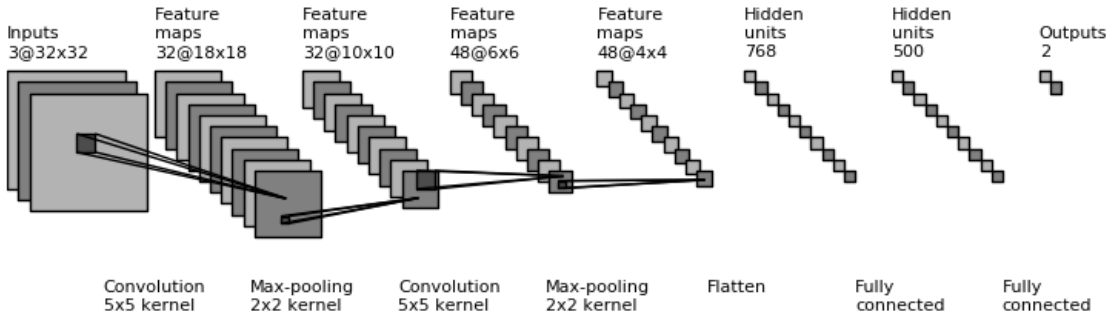


FIGURE 5.1: Convolutional Neural Network [7]

5.1.2 Forward Propagation

1. Pooling Layer

The **pooling layer** exists to reduce the 2D dimensionality of an input image. Using the 'max-pooling' operation, given a spatial extent (size of a square matrix) and a stride length (number of cells we slide along between matrix operations), we shift a matrix over an image to return only the maximum value at each region.

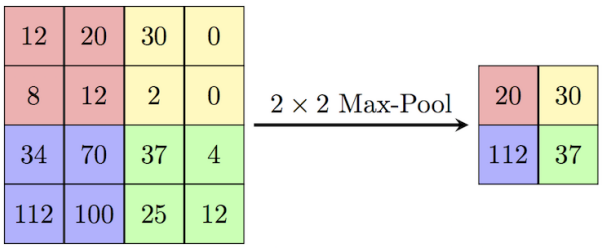


FIGURE 5.2: Max-Pooling using a pool size of 2x2 and stride length of 2

2. Convolutional Layer

The **convolutional layer** owns a number of weights which are called filters - these are 2D matrices with a smaller 2D dimensionality than the input it receives. Each filter shares the same 2D dimensionality as all others, and have the same volume as the input image.

The forward pass consists unsurprisingly of performing convolution between the filters and input images. We slide (more precisely, convolve) each filter across the width and height of the input volume and apply the convolutional operation between the entries of the filter and the input at any position. At every position the operation is performed, all of the values at each depth produced from applying a filter to its corresponding 2D region in the input volume, are summed up to output a single value. This then has a bias added on. Each filter convolved over the input produces a single 2D image, and an activation function is then mapped over every value.

$$z_{i,j}^{L} = \sum_{M}\sum_{N} w_{m,n}^{L} a_{i+m,j+n}^{L+1} + b_{i,j}^{L}$$
$$a^{L} = \sigma(z_{i,j}^{L})$$

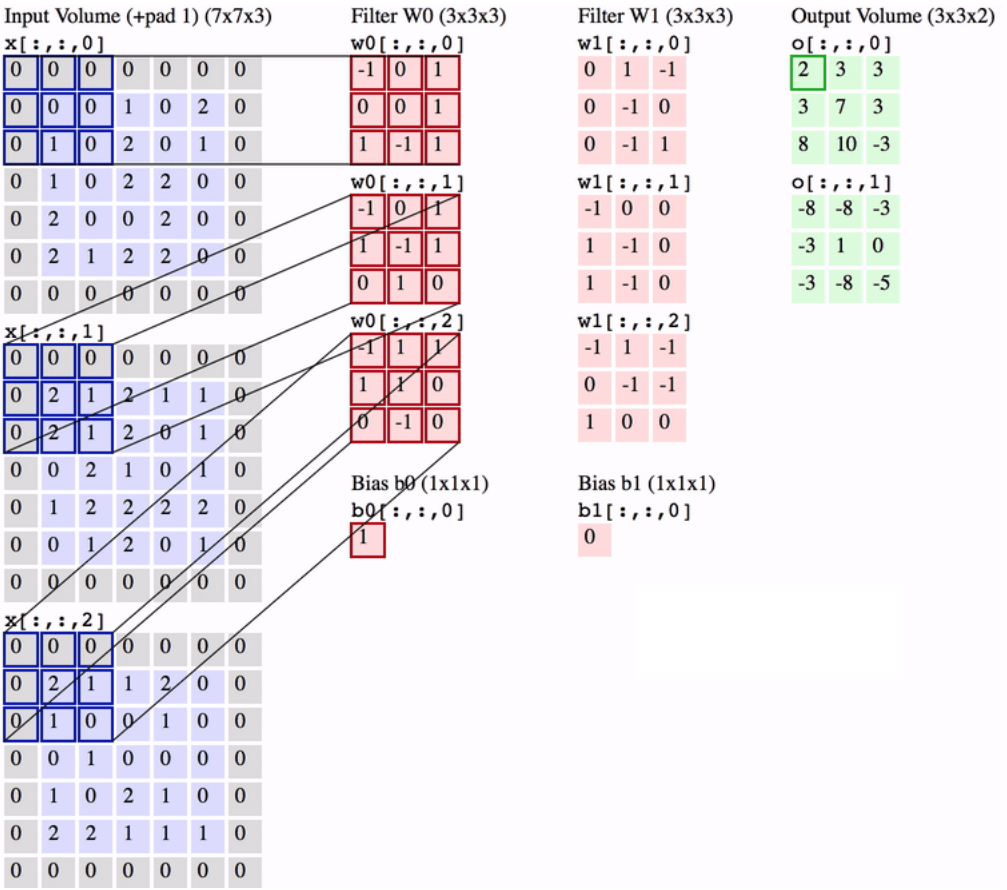


FIGURE 5.3: Convolutional Layer: Forward Propagation [6]

3. ReLu Layer

The **relu layer** simply takes all negative values in the input image and changes these to 0.

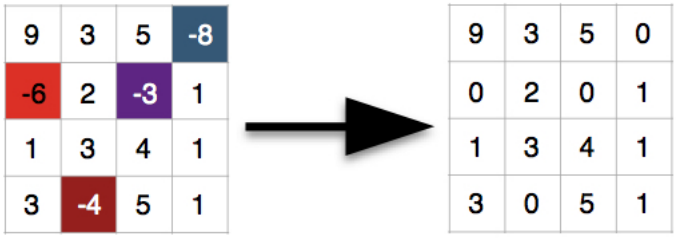


FIGURE 5.4: ReLu Layer

4. Fully Connected Layer

The **fully connected layer** is our final layer producing the output of the neural network. It entails taking a set of images and flattening them out to be an array of values such that each value corresponds to the probability of the original sample belonging to a certain class.

5.1.3 Back Propagation

1. Fully Connected Layer

Beginning from the output layer, i.e. the fully connected layer, we are interested in finding its delta value δ^L as the cost given the actual output and desired output; this is acquired using the following equation, where a denotes the actual output and $\overline{a}$ denotes the desired output:

$$\delta^L = \frac{1}{2}(a - \overline{a})$$

This is then passed back to the previous layer.

2. Convolutional Layer

In the convolutional layer, back propagation intends to do 3 things:

- Find the delta error value δ^L
- Find the gradient of the loss with respect to the filters $\frac{\partial E}{\partial W}$
- Update the layer's filters

To acquire the delta error δ^L, we carry out a *full convolution* $\circledast$ between the next layer's delta error δ^{L+1} and the transpose of the filter w^T. This is then multiplied with the differential activation function σ' applied to the input x^L.

$$\delta^L = (\delta^{L+1} \circledast w^T)\sigma'(z^L)$$

Next we will compute the gradient of the filters, $\frac{\partial E}{\partial W}$. To acquire the gradient of the loss with respect to the filters $\frac{\partial E}{\partial W}$, we perform convolution between the next layer's delta δ^{L+1} and the input x^L.

$$\frac{\partial E}{\partial W} = \delta^{L+1} * z^L$$

Finally, we update the filters by multiplying a chosen learning rate η with the gradient of our filters, and then subtracting this from our original filters.

$$w^{new} = w^{old} - \eta \odot \frac{\partial E}{\partial W}$$

3. Pooling Layer

In the pooling layer, back propagation only intends to find the delta error value δ^L. As mentioned previously when implementing forward propagation, this consists of 'unpooling' the output it produced. This means taking the pooling layer's input matrices, and setting all the max values we extracted when max-pooling to 1, and setting all other values to 0.

4. ReLu Layer

Similarly, the ReLu layer only requires us to find the delta error value δ^L. This is actually the same process as forward propagation - the delta is computed by taking the input matrices and setting all negative input values to be set to 0.

5.2 Finding A Data Structure

To start off, let's define type aliases for the variables we are going to dealing with.

```
type Filter        = [[[Double]]]
type Biases        = [Double]
type Image         = [[[Double]]]
type Stride        = Int
type SpatialExtent = Int
```

The data type for layers will have to account for all of the previously mentioned types of layers as well as the input layer. We will create a separate constructor for each variation.

- The convolutional layer will need a list of its filters and their corresponding biases, as well as the stride length during convolution.
- The relu layer requires no parameters as it is a simple application of zeroing negative numbers
- The fully connected layer requires no parameters as it performs no operations on its provided inputs.
- The pooling layer needs both the stride and spatial extent parameters to know the frequency and region-size of max-pooling.

```
data Layer k = InputLayer
             | ReluLayer k
             | FullyConnectedLayer k
             | PoolingLayer Stride SpatialExtent k
             | ConvolutionalLayer Stride [Filter] [Biases] k
```

5.3 Implementing Forward Propagation

Similarly to fully connected neural networks, during forward propagation we can simply pass forward a function which takes a list of images to a list of images as our carrier type. We will hence begin with the idea of using the carrier type `(Fix Layer, [Image] -> [Image])`.

5.3.1 Pooling Layer

Recall: *The pooling layer uses the 'max-pooling' operation, given a spatial extent (size of a square matrix) and a stride length (number of cells we slide along between matrix operations), we shift a matrix over an image to return only the maximum value at each region.*

First we need to acknowledge that during back propagation we have to be able to 'unpool' our output - this means reconstructing the original matrix by placing the max values at their positions before being max pooled. We need to store the positions of the values' at the previous layer; the easiest way to do this is to change the carrier type we are forward propagating - let's create a new data type `ForwardProp` to store both the image output as well as the pool layer positions, hence our carrier type will now be `(Fix Layer, [ForwardProp] -> [ForwardProp])`.

```
data ForwardProp = ForwardProp { _image     :: Image,
                                 _positions :: [[(Int, Int)]] }
```

To implement max pooling, we must define how to extract the regions of which the matrix is shifted over from the image. It is done so in the same way as convolution. One method of performing this, is to take the input image, the stride length, and the spatial extent of the filter, to extract a chronological list of regions in the image which the filter will be applied over.

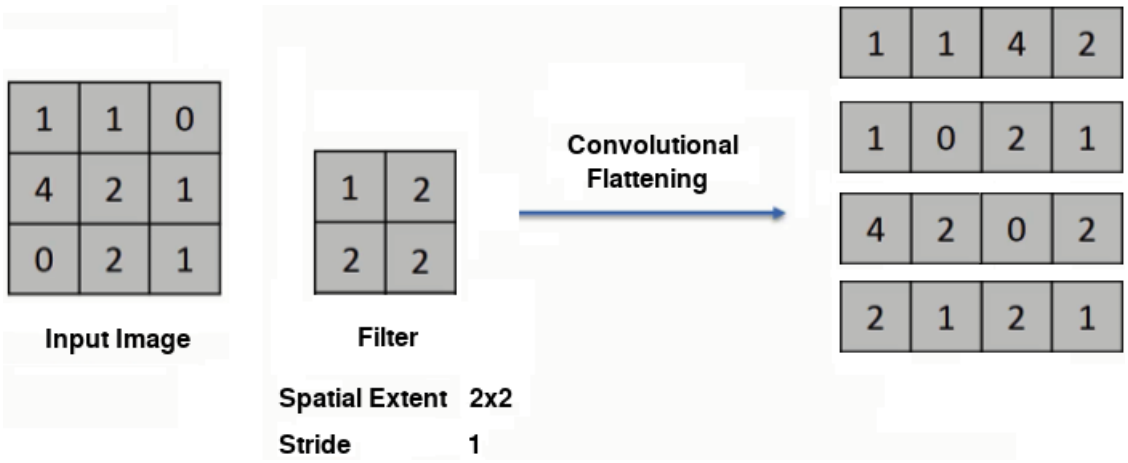


FIGURE 5.5: Reshaping An Image Into Convolution Segments

We will define this as `convoluteFlatten`. Given an (n x n) matrix, it operates by `splitVert` taking the first n rows from the image and then calling `splitHori`. The function `splitHori` shifts across the provided rows by taking the first n elements from each of the arrays to form a single array which it adds to its list. Both inner functions shift down or across by respectively either taking the stride length number of rows off of the image, or taking the stride length number of elements off of each of the original rows, followed by recursively calling itself.

```
convoluteFlatten :: [[a]] → SpatialExtent → Stride → [[a]]
convoluteFlatten image spatialExtent stride =

    let splitVert imageV stackArray = case () of
            _ | length imageV < spatialExtent → then stackArray
            _ | otherwise → else ( splitHori image' (take spatialExtent imageV) stackArray)

        splitHori imageH imageChunk stack = case () of
            _ | length (head imageChunk) < spatialExtent → ( splitVert (drop stride imageH) stack)
            _ | otherwise → let newStack = stack ++ (concat $ map (take spatialExtent) $
                                                        take spatialExtent imageChunk)
                            in splitH imageH (map (drop stride) imageChunk) newStack

    in chunksOf (sqr spatialExtent ) ( splitVert image [])
```

Another function we will find useful, is one which gives us the dimensions of an output image after we perform convolution over it. Given the spatial extent, stride length, and image, this is given by the equation $((i - m)/stride + 1, (j - n)/stride + 1)$ where (m, n) are the max-pool/filter's dimensions and (i, j) are the image's dimensions.

```
convoluteDims :: [[a]] → SpatialExtent → Stride → (Int, Int)
convoluteDims image spatialExtent stride =
    let (m0, n0, i0, j0) = ( spatialExtent, spatialExtent,
                             length image, length $ head image )
    in  (quot (i0 − m0) stride + 1 , quot (j0 − n0) stride + 1 )
```

Using our `convoluteFlatten` function, we acquire the regions in the image. For each region, we extract the max value and their corresponding positions. In order to do this, some list comprehensions and zipping is required beforehand.

```
pool :: Int → Int → [[Double]] → ([[( Int, Int )]], [[Double]])
pool stride spatialExtent image2D =
    let flatImage = convoluteFlatten image spatialExtent stride

        (h, w)    = (length $ image, length $ head image)
        (m, n)    = convoluteDims image2D spatialExtent stride
```

```
    (positions, image2d) = unzip $ max_pool flatImage 0
  in (positions, chunksOf n maxImage)
  where
    max_pool :: [[Double]] → [((Int, Int), Double)]
    max_pool (x:xs) i  = let max_x     = (maximum x)
                             index     = fromJust $ elemIndex max_x x
                             (m', n')  = (quot ind spatialExtent, ind `mod` spatialExtent)
                             (row, col) = (quot i n, (stride * i) `mod` n)
                         in  (((row + m', col + n'), max_x):(f xs (i + 1)))
    max_pool [] i = []
```

The above function returns us a tuple, containing firstly a matrix of positions represented as (row, column) and secondly the max-pooled matrix. This leaves us to define the algebra for pooling layers, which is given below.

```
alg :: Layer (Fix Layer, [ForwardProp] → [ForwardProp]) → (Fix Layer, [ForwardProp] → [ForwardProp])
alg (PoolingLayer stride spatialExtent (innerLayer, forwardPass))
      = (Fx (PoolingLayer stride spatialExtent innerLayer),
            (\fps →
                let input       = head fps
                    inputImage  = input ^. image
                    pooledinput = (map (pool stride spatialExtent) inputImage)
                    (originalPositions, outputImage) = unzip . map unzip $ pooledinput

                    output = (head fps) & image     .~ outputImage
                                        & positions .~ originalPositions
                in  (output:fps)) . forwardPass)
```

We update our existing forward propagation function `forwardPass` of type `[ForwardProp] -> [ForwardProp]`, by composing it with a new forward propagation function that we define.

This new function operates by first taking the list of `ForwardProp` values the previous layer produced, and extracts the input image from the head element. It then maps the function `pool` over it to apply the max-pool operation to each 2D image slice - this returns us a list of tuples of type `[([[(Int, Int)]], [[Double]])]` which contains the original positions and the max-pooled matrix. We can then use a composition of `map unzip` and `unzip` to reformat this data to give a single tuple of type `([[[(Int, Int)]]], [[[Double]]])`, containing firstly a list of all positions and secondly a list of all 2D max-pooled images. Finally, we create a new `ForwardProp` value called `output` to contain these results, and add it to the list of forward propagation values which we return.

5.3.2 Convolutional Layer

Recall: *When forward propagating in the convolutional layer, for each filter we perform the convolutional operation between the filter and the input - each output then has its 2D matrices at each depths summed up element-wise. Afterwards, every value then has a bias added on and an activation function is applied. Each filter should produce a single 2D output matrix.*

Let's start off by defining the algebra for the convolutional layer. The core part of this is implementing convolution - as far as neural networks are concerned, cross-correlation and convolution are identical, so we will actually be defining cross-correlation.

It makes sense to first define 2D convolution before 3D. Fortunately from the previous functions we defined for the pooling layer, 2D convolution becomes a straightforward process of calling `convoluteFlatten` to get the regions which will be convolved over, and then mapping over these to perform element-wise multiplication between the filter and region followed by a summation of the result. The computed dimensions from `convoluteDims` allows us to then correctly reconstruct the shape of the output matrix.

```
convolute2D :: [[Double]] → [[Double]] → Stride → [[Double]]
convolute2D filter image stride
    =  let spatialExtent   = (length filter)
```

```
        flat_image      = convoluteFlatten image spatialExtent stride
        (m, n)          = convoluteDims image spatialExtent stride
    in chunksOf n $ map (sum . zipWith (*) (concat filter )) flat_image
```

Taking convolution to 3D then becomes a simple task, where we use a list comprehension to apply 2D convolution between each 2D filter and 2D image slice.

```
convolute3D :: [[[Double]]] → [[[Double]]] → Stride → [[[Double]]]
convolute3D filter image stride
    = [ convolute2D filter2D image2D stride (image2D, filter2D) <- zip image filter]
```

We can then use this to define a function `forwardConvolution` which performs forward propagation for one of the filters.

```
forwardConvolutional :: Filter → Image → Stride → Image2D
forwardConvolutional filter image stride
    = let (m, n)              = convoluteDims (length $ head filter ) image stride
          bias = 1.0
      in map2 (sigmoid . (bias +)) $ foldr eleaddm ( fillMatrix m n 0.0) (convolute3D filter image stride )
```

This works by first using `convoluteDims` to identify the resulting dimensions after convolution, and creating an zero-matrix of this size. It then applies convolution between the filter and the input image to produce a 3D matrix. Afterwards, we use `foldr` with the zero-matrix as a base-case to element-wise sum up each of the 2D matrices at each depth - this returns a single 2D matrix. Finally, we map over every value by adding on a bias and applying the sigmoid activation function.

Let's now use this to define a pattern match for convolutional layers in our algebra.

```
alg :: Layer (Fix Layer, [ForwardProp] → [ForwardProp]) → (Fix Layer, [ForwardProp] → [ForwardProp])
alg (ConvolutionalLayer filters biases stride (innerLayer, forwardPass))
    = (Fx (ConvolutionalLayer filters biases innerLayer),
       (\fps →
            let input      = head fps
                inputImage = input ^. image
                stride     = 1
                outputImage = [(forwardConvolutional filter inputImage 1) | filter <- filters]
                output     = (head fps) & image .~ outputImage
            in (output:fps)) . forwardPass)
```

This takes a similar structure to our algebra for the pooling layer in the sense that we compose the existing forward propagation function `forwardPass` with a new forward propagation function. In this new function, we first extract the input image and declare the stride length as being one. Then we create a list comprehension which applies `forwardConvolution` between each filter and the input image to produce the output image. This finishes by constructing a new `ForwardProp` value to add on to the returned list of `ForwardProp` values.

5.3.3 ReLu Layer

Recall: *The ReLu layer simply takes all negative values in the input image and changes these to 0.*

This is fortunately simple enough to directly define inside the algebra for ReLu layers.

```
alg :: Layer (Fix Layer, [ForwardProp] → [ForwardProp]) → (Fix Layer, [ForwardProp] → [ForwardProp])
alg (ReluLayer (innerLayer, forwardPass))
    = (Fx (ReluLayer innerLayer),
       (\fps →
            let input      = head fps
                inputImage = input ^. image
                outputImage = (map3 (\x → if x < 0 then 0 else x) inputImage)
                output = (head fps) & image .~ outputImage
            in (output:fps) ) . forwardPass)
```

In here, we map over all input matrices with a function which changes a value to 0 upon it being less than 0 - this gives us the output image which we use to add a new `ForwardProp` value to the `ForwardProp` list.

5.3.4 Fully Connected Layer

Recall: *The fully connected layer is the final layer which takes a set of images and flattens them out to be an array of values such that each value corresponds to the probability of the original sample belonging to a certain class.*

To define this, we first need to create a function `flattenImage`; this can simply work by applying `concat` twice. However, when flattening the input remember that we still need this output to be of type `Image` which is a 3D array. Therefore, after flattening to produce a single array of doubles, we will have to wrap each individual value inside its own 2D array.

```
flattenImage :: Image → Image
flattenImage image = [ [[x]] | x <- (concat $ concat $ image) ]

alg :: Layer (Fix Layer, [ForwardProp] → [ForwardProp]) → (Fix Layer, [ForwardProp] → [ForwardProp])
alg (FullyConnectedLayer (innerLayer, forwardPass))
    = (Fx (FullyConnectedLayer innerLayer),
            (\fps →
                let input       = head fps
                    inputImage  = input ^. image
                    outputImage = (flattenImage $ inputImage)
                    output      = (head fps) & image .~ outputImage
                in  (output:fps) ) . forwardPass )
```

The algebra is incredibly straightforward - it applies `flattenImage` to the input image and adds this to the `ForwardProp` list.

5.3.5 Input Layer

Finally, we finish off by defining the algebra for the input layer.

```
alg :: Layer (Fix Layer, [ForwardProp] → [ForwardProp]) → (Fix Layer, [ForwardProp] → [ForwardProp])
alg (InputLayer)
    = (Fx InputLayer, id)
```

Exactly as in our implementation of fully connected networks, the input layer returns the identity function so that we can pass in an initial input and have it outputted to the next function in the chain of function compositions we're creating.

5.4 Implementing Back Propagation

We now move on to modelling back propagation as an anamorphism. Let us start by defining the carrier type we will use; as with our implementation of fully connected networks, we create a data type `BackProp` to represent the back propagated variables.

```
data BackProp = BackProp { nextDeltas    :: Deltas
                         , desiredOutput :: Image   }
```

This will contain the next layer's computed delta value, and the desired output. We also need to have access to the list of forward propagated values; this means our resulting carrier type will be `(Fix Layer, [ForwardProp], BackProp)`. From here we will define the coalgebra for each layer.

5.4.1 Fully Connected Layer

Recall: *In the fully connected layer, we require to compute the delta error δ^L found by the error using the actual output and the desired output.*

We begin with implementing the coalgebra where back propagation begins - the fully connected layer. To find its delta value, we create the function `computeCost`.

```
computeCost :: Image → [[[Double]]] → (Int, Int) → Deltas
computeCost actualOutput desiredOutput (m_dim, n_dim) =
    let     (actualOutput', desiredOutput') = mapTuple2 (concat . concat) (actualOutput, desiredOutput)
    in      unflatten  (zipWith f actualOutput' desiredOutput')
    where
            f :: Double → Double → Double
            f = (\actOutput desOutput → 0.5 * ((snd actOutput) − desOutput))

            unflatten :: [Double] → Deltas
            unflatten flattened_deltas
                                = map (chunksOf m_dim) (chunksOf (m_dim * n_dim) flattened_deltas)
```

In this, we define the function `f` which represents the following equation, where a denotes the actual output and $\overline{a}$ denotes the desired output:

$$\delta = \frac{1}{2}(a - \overline{a})$$

However recall that the layer's outputs have been flattened such that every element in the array is a 2D array containing a single value. The delta we compute must be of the same dimensionality as the layer's original input. We thus define the function `unflatten` which reshapes a list of doubles into a 3D matrix.

Using this, we can compute delta by using `zipWith` with the function `f` between the flattened actual output and flattened desired output, followed by applying `unflatten` to return a 3D matrix with the layer's input dimensions.

Now we can define the coalgebra.

```
coalg :: (Fix Layer, [ForwardProp], BackProp) → Layer (Fix Layer, [ForwardProp], BackProp)
coalg (Fx (FullyConnectedLayer innerLayer), fps, BackProp outerDeltas desiredOutput)
    =   let (output:input:_) = fps
            inputImage       = input ^. image
            outputImage      = output ^. image
            (m_dim, n_dim) = (length (head $ head inputImage), length (head inputImage))

            deltas           = compDeltaFullyConnected outputImage desiredOutput (m_dim, n_dim)

        in  FullyConnectedLayer (innerLayer, (tail fps), BackPropData deltas desiredOutput)
```

This will first require getting the output and input image from the list of `ForwardProp` values, getting the input image dimensions, and calling `computeCost` to acquire the delta value. We then return a fully connected layer with a new carrier value containing: the inner (previous) layer, the tail of the `ForwardProp` list, and the back propagation data where its delta value has been updated to our recently computed delta.

5.4.2 Convolutional Layer

Recall: *In the convolutional layer, back propagation requires us to do three things: 1) find the delta error value δ^L, 2) find the gradient of the loss with respect to the filters $\frac{\partial E}{\partial W}$, and 3) update the layer's filters w.*

Implementing the back propagation process for convolutional layers is very fiddly due to the heavy amount of matrix operations and dimension handling we perform.

1) Computing The Delta Error δ^L

We will start off by looking at computing the delta error δ^L, given by the following equation. This says that we use δ^{L+1} to convolve over the filters w^T, before multiplying it with the differential activation function of the layer's input.

$$\delta^L = (\delta^{L+1} \circledast w^T)\sigma'(z^L)$$

This requires us to first define the full convolution operation $\circledast$. Full convolution means that the matrix we use to convolve over a subject matrix, performs convolution at every position where at least one of its elements is in contact with an element in the subject matrix. This is in contrast to normal convolution where all of its elements must be in contact with the subject matrix. To do this, we take advantage of the fact that normal convolution on a zero-padded matrix is equivalent to full convolution on an unpadded matrix. This means all we need to do is correctly zero-pad the filter and apply normal convolution between the delta and the padded filter.

```
fullConvolution :: [[Double]] → [[[Double]]] → Stride → [[[Double]]]
fullConvolution delta filter stride
    = let (w, h)        = (length (head filter) − 1, length filter − 1)
          padding_w     = if w == 0 then []
                          else [ 0 | x <− [1 .. w]]
          padding_h     = if h == 0 then []
                          else [[ 0 | x <− [1 .. (length (head filter) + w + w)]] | y <− [1 .. h] ]
          padded_filter = map (\mat → if h == 0 then mat else padding_h ++ mat ++ padding_h)
                          . map2 (\row → padding_w ++ row ++ padding_w) $ filter
      in [ convolute2D delta filter2d stride | filter2d <− padded_filter ]
```

The function `fullConvolution` uses delta to extract its dimensions in order to construct the correctly sized vertical and horizontal zero paddings. These are then concatenated onto the filter to give the padded filter. Finally, we can use a list comprehension to apply normal 2D convolution between the delta and each 2D slice of the padded filter.

Using this, we can create a function `convDeltaX` which computes delta for a convolutional layer.

```
convDeltaX :: Deltas → [ Filter ] → Image → Deltas
convDeltaX outerDeltas filters inputImage
    = let wTdelta = [ fullConvolution outerDelta (transpose3D filter) 1
                    (outerDelta, filter) <- zip outerDeltas filters]wTdelta' = foldr sumMat3D (head wTdelta) (tail
wTdelta)in mmmul3d wTdelta' (map3 sigmoid' inputImage)
```

From the below implementation, the first step applies full convolution between each δ^{L+1} slice and its respective filter, returning a list of 3D matrices `wTdelta`. The second step then sums over this list to give us `wTdelta'` which represents the component $(\delta^{L+1} \circledast w^T)$. Finally, the last step multiplies this with $\sigma'(x^L)$ to produce δ^L.

2) Computing The Cost Gradients For The Filters $\frac{\partial E}{\partial W}$

Next we will compute the gradient of the loss with respect to the filters, $\frac{\partial E}{\partial W}$. This consists of performing convolution between the next layer's delta δ^{L+1} and the input z^L, as described in the below equation.

$$\frac{\partial E}{\partial W} = \delta^{L+1} * z^L$$

We create a function `convoluteDeltaW` to represent this equation.

```
convoluteDeltaW :: Deltas → Image → Stride → Deltas
convoluteDeltaW deltas image stride
    = let convolve2Don3D x y = [ convolute2D x y stride y2d <- y ]in [ convolve2Don3D delta image stride delta <− deltas]
```

Given a single 2D slice of delta and the 3D input image, convolving them will produce a gradient for a single filter - we name this function as `convolve2Don3D`. This means that doing this for all deltas gives us all of the filters' gradients - this is seen in the final list comprehension returned.

3) Updating The Filters $\frac{\partial E}{\partial W}$

Finally, let's now use both of our delta functions to create the coalgebra for convolutional layers.

```
coalg :: (Fix Layer, [ForwardProp], BackPropData) |$\rightarrow$| Layer (Fix Layer, [ForwardProp], BackPropData)
coalg (Fx (ConvolutionalLayer filters biases innerLayer), fps, BackPropData outerDeltas desiredOutput)
      = let (output:input:_) = fps
            inputImage       = input ^. image
```

```
        outputImage     = output ^. image
        learningRate    = 0.1

        deltaX          = convDeltaX outerDeltas filters inputImage
        deltaW          = convDeltaW outerDeltas filters inputImage

        newFilters      = [ zipWith elesubm filter (map3 (learningRate *) delta_w)
                                | (filter, delta_w) <- (zip filters deltaW) ]

    in  ConvolutionalLayer newFilters biases (innerLayer, (tail fps),
                                BackPropData deltaX filters desiredOutput)
```

We first use `convDeltaX` and `convDeltaW` to retrieve δ^L and $\frac{\partial E}{\partial W}$. Afterwards, we update the filters at the end by multiplying a chosen learning rate η with the gradient of our filters, and then subtracting this from our original filters. This is given by the following equation.

$$w^{new} = w^{old} - \eta \odot \frac{\partial E}{\partial W}$$

Finally, we return the layer with its new filters as well as the passing back the new δ^L value into the `BackProp` data type.

5.4.3 Pooling Layer

Recall: *Back propagating through the pooling layer requires only finding the delta error value δ^L. This consists of taking all the max values we extracted when max-pooling during forward propagation, and placing them at their original positions in the input matrix; we then set all of these to 1 and all other values to 0.*

Lets decompose the problem to performing unpooling for a single 2D matrix, given by the function `unpool`. All we need are the original width and height of the input, and the original input matrix positions of the extracted output matrix's max values.

```
replaceElement :: [a] → Int → a → [a]
replaceElement xs i x =
    fore ++ (x : aft)
    where fore = take i xs
          aft = drop (i+1) xs

unpool :: (Int, Int) → [[ Position ]] → [[Double]]
unpool (orig_w, orig_h) positions =
    let zeros           = replicate orig_h $ replicate orig_w 0.0
        set ls (y:ys)   = let (m, n) = y
                          in set (replaceElement ls m (replaceElement (ls !! m) n 1.0)) ys
        set ls []       = ls
    in  set zeros (concat positions)
```

We create a 2D matrix of zeros using the width and height of the input, and then insert the integer one at every position in the list of matrix positions.

This makes defining the coalgebra fairly simple.

```
coalg :: (Fix Layer, [ForwardProp], BackPropData) → Layer (Fix Layer, [ForwardProp], BackPropData)
coalg (Fx (PoolingLayer stride spatialExtent innerLayer),
                                fps, BackProp outerDeltas outerFilters desiredOutput)
  = let (output:input:_) = fps
        inputImage = input ^. image
        (input_width, input_height) = (length $ head $ head inputImage, length $ head $ inputImage)
        deltas = [unpool (input_width, input_height) positions2d | positions2d <- (output ^. positions) ]
    in (PoolingLayer stride spatialExtent (innerLayer,
                                (tail fps), BackProp deltas outerFilters desiredOutput) )
```

The can compute the deltas by defining a list comprehension where we call unpool for each 2D matrix in the 3D matrix of positions. This gives us the final delta error δ^L which we pass backwards inside the `BackProp` data type.

5.4.4 ReLu Layer

Recall: *The ReLu layer requires us to compute only the delta error value δ^L by taking the input matrix and setting all negative values to 0.*

This is very easy to define.

```
coalg :: (Fix Layer, [ForwardProp], BackPropData) → Layer (Fix Layer, [ForwardProp], BackPropData)
coalg (Fx (ReluLayer innerLayer), fps, BackPropData outerDeltas desiredOutput)
        = let inputImage = head (tail fps) ^. image
              deltas     = map3 (\x → if x < 0 then 0 else x) inputImage
          in (ReluLayer (innerLayer, (tail fps), BackPropData deltas desiredOutput) )
```

To find δ^L, we simply use the same method as in forward propagation, by mapping over the input image with a function which sets all negative values to 0.

5.4.5 Input Layer

Finally, we finish off with the coalgebra of the base case - the input layer.

```
coalg :: (Fix Layer, [ForwardProp], BackPropData) → Layer (Fix Layer, [ForwardProp], BackPropData)
coalg (Fx InputLayer, fps, backPropData)
       = InputLayer
```

5.5 Training A Convolutional Neural Network

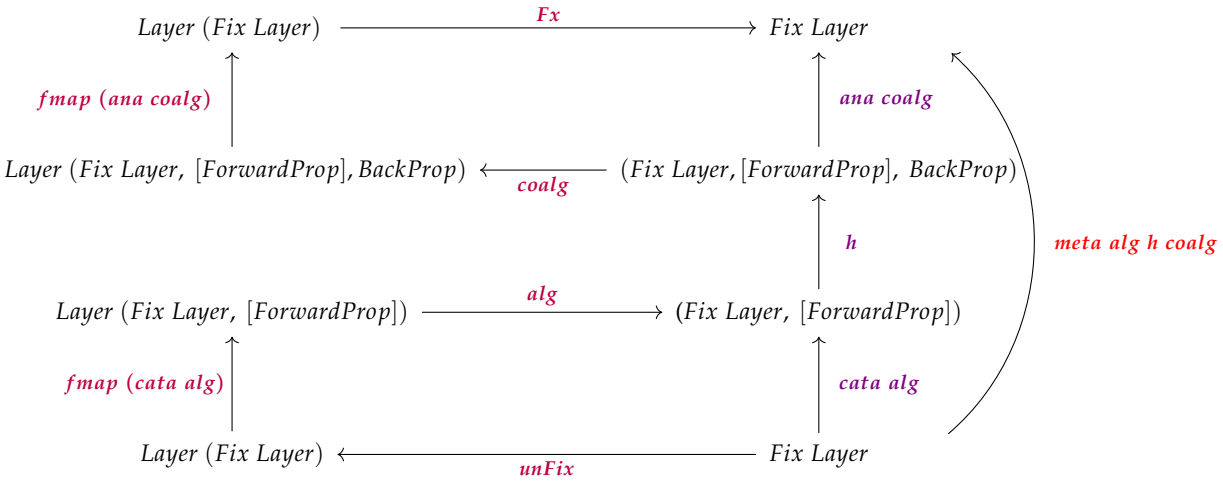
Just like fully connected networks, the entire process for convolutional networks is a metamorphism. In fact, we can define this almost exactly the same way. Below we give the function `run_sample` for running a single sample through a convolutional network, and the function `train` for running a set of samples through a convolutional network.

```
run_sample :: Fix Layer → Image → DesiredOutput → Fix Layer
run_sample neural_net sample desired_output
  = meta alg h coalg $ neural_net
    where h :: (Fix Layer, [ForwardProp] → [ForwardProp]) → (Fix Layer, [ForwardProp], BackProp)
          h = \(neural_net', forwardPass) →
                  (neural_net', (forwardPass [(ForwardProp sample [[[]]])]),
                      BackProp [[[]]] [[[[]]]] desired_output))

train :: Fix Layer → [Image] → [DesiredOutput] → Fix Layer
train neural_net samples desired_outputs
  = foldr (\(sample, desired_output) nn →
                run_sample nn sample desired_output) neural_net training_data
    where training_data = zip samples desired_outputs
```

The function `run_sample` performs the metamorphism of the network using a single sample, and the function `train` uses `foldr` to execute the process over a list of samples. The intermediary function `h` inside `run_sample` changes the carrier type between the catamorphism and anamorphism. It first takes the forward propagation function `forwardPass` the catamorphism produces and applying this to a list containing the initial `ForwardProp` value and the input data. Then it constructs an initial `BackProp` data type containing the desired output for the anamorphism to start off with. The parameters containing empty lists are simply the variables which will later be initialised during the process.

- **Summary**

We have now completed our implementation of the second neural network, given by a metamorphism whose catamorphism evaluates the convolutional network to represent it as a function which performs forward propagation, and an anamorphism which takes the results of forward propagation to reconstruct an updated convolutional network. Although it may be too early to tell, a certain recursion pattern seems to be reoccurring when modelling neural networks with recursion schemes. Our implementation can be illustrated by taking the metamorphism diagram and tailoring its types to our convolutional network implementation.



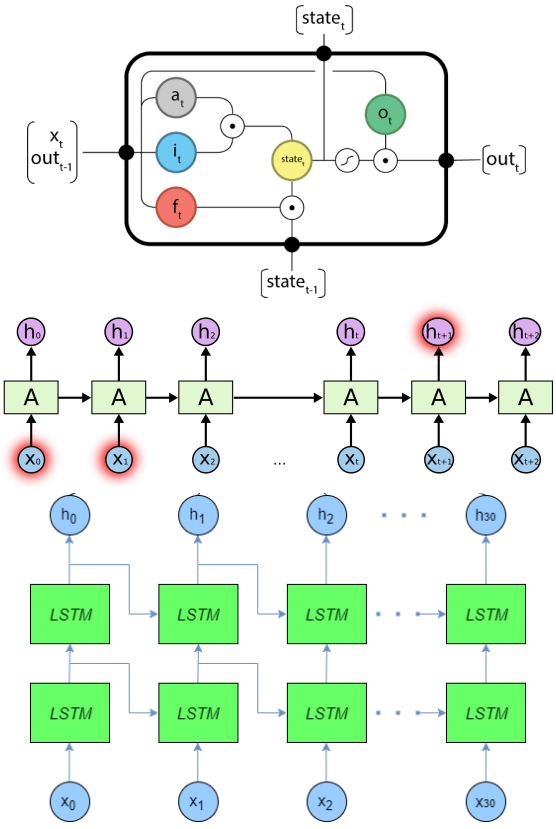
Chapter 6

Defining Metamorphisms For Deep LSTMs

It's becoming more evident that metamorphisms and neural network training seem to fit well together. However both of our previous examples are feed-forward networks. In this section I aim to further emphasize the presence of recursion schemes in models considered as less obvious scenarios which are capable of demonstrating a recursive pattern. As such, we tackle an area of neural nets known as recurrent networks, which encapsulate an entirely different learning process. Specifically, I have chosen to use Deep LSTM (Long Short Term Memory) recurrent networks.

The main difference between recurrent and feed-forward networks, is that recurrent networks have understand the concept of time to an extent. This means that unlike previous examples, recurrent networks are not limited to processing single data points (such as images), but also entire sequences of data. For example, an LSTM can be used for time-series recognition, handwriting recognition or speech recognition.

In this chapter, we will first implement a *non-deep LSTM network*. LSTMs do not just use layers, but also revolve around a concept called cells. A cell encompasses the idea of updating and caching states and also calculating other variables; each single cell performs multiple computations to achieve this, as can be seen from the below complex work flow. A non-deep LSTM network consists of a single layer of cells stacked side by side. A main property which distinguishes the flow from being feed forward is that each cell takes two inputs - an input variable from the sample as well as the output variable from the previous cell.



Afterwards, we will extend this implementation to model a *deep LSTM network*. A deep LSTM can be considered to be a vertical stacking of multiple layers of cells. Just by looking at the network's circulation, the large potential for recursive patterns is obvious. We could not only employ a recursion scheme on the cellular level of the network, but also encapsulate this within another recursion scheme which operates on the layer layer of the network. This can potentially describe some sort of double metamorphism, where one is nested inside another.

As always, we start off by describing the structure and training process of each network, followed by a discussion of how these translate into implementations.

6.1 LSTMs

6.1.1 Structure

1. Cells

Cells as data types are comparatively more complex than the layers we have seen and implemented so far, this stems from the multitude of new variables we must introduce and maintain in each cell. However, they can be roughly thought of as a more powerful type of layer which can encapsulate statefulness.

A cell's structure contains a specific workflow as can be seen from the diagram. It takes as input three variables:

- The input x given from below the cell
- The output out_{t-1} from the cell on its left hand side
- The cell's current state $state_{t-1}$

It has four main computational components known as gates; given an input, each gate performs a specific function to produce an output variable which is then cached inside the cell. These are:

- Input Gate i
- Output Gate o
- Activation Gate a
- Forget Gate f

This network of computations in a cell results in producing an output variable out_t and updating the current state to $state_t$.

2. LSTM Network

An LSTM network can be visualized as a layer consisting of a single row of cells connected side by side. To elaborate on the network's workflow, each cell requires an input variable x_n from the input data below, as well as the previous cell's output variable h_{n-1} from the left, in order to produce its own output variable h_n. As can be observed, every output h_n produced by a cell is outputted twice; one copy is sent to the next cell on the right, and the other copy is sent above as a final output variable used for classification.

Additionally, all cells in a row share the same weight matrices and bias matrices. Specifically, there are 8 weight matrices for a whole row (2 weights assigned for each gate type), and 4 bias vectors (1 assigned for each gate type). These are namely:

- The set of W weights, consisting of W_a, W_i, W_f, W_o
- The set of U weights, consisting of U_a, U_i, U_f, U_o
- The set of B bias vectors b_a, b_i, b_f and b_o

3. Dimensionalities

The dimensions of our variables are important to know that all matrix and vector operations fit together nicely. Given H which is the length of the state vector we choose, and D which is the length of an input vector:

- Input Vector $x_t \in \mathbb{R}^D$
- Gate Output Vectors $f_t, i_t, a_t, o_t \in \mathbb{R}^H$
- State Vector $state_t \in \mathbb{R}^H$
- W Weight Matrices $W_a, W_i, W_f, W_o \in \mathbb{R}^{HxD}$
- U Weight Matrices $U_a, U_i, U_f, U_o \in \mathbb{R}^{HxH}$
- B Bias Vectors $b_a, b_i, b_f, b_o \in \mathbb{R}^H$

6.1.2 Forward Propagation

For an LSTM network consisting of a single row of cells, forward propagation occurs horizontally from the left to right hand side due to a cell's dependency on its previous cell. The input data x consists of a different input vector x_t for each cell.

1. Understanding The Initial Variables

Starting from the first LHS cell, recall the set of variables we require to forward propagate through a cell:

- The cell's input: x_t
- The cell's previous state: $state_{t-1}$
- The previous cell's output: out_{t-1}

We will always have immediate access to x_t from the input data and the cell's state $state_{t-1}$, but have to wait for the previous cell to finish forward propagating in order to acquire its output out_{t-1}. In the case of there existing no previous state or previous cell, variables $state_{t-1}$ or out_{t-1} will be initialized as a vector of zeros.

2. Computing The Gate Variables

The variables described in Step 1 are fed these into a series of gates, where each gate performs a computation to produce a vector. The following equations describe the computation which each gate applies to its given input.

Activation Gate Variable
$$a_t = tanh(W_a \circ x_t + U_a \circ out_{t-1} + b_a) \quad (1)$$

Input Gate Variable
$$i_t = \sigma(W_i \circ x_t + U_i \circ out_{t-1} + b_i) \quad (2)$$

Forget Gate Variable
$$f_t = \sigma(W_f \circ x_t + U_f \circ out_{t-1} + b_f) \quad (3)$$

Output Gate Variable
$$o_t = \sigma(W_o \circ x_t + U_f \circ out_{t-1} + b_f) \quad (4)$$

3. Computing The Output And Updated State

The results of the previous computations allow us to then compute the cell's output out_t and its updated state $state_t$, given by the equations below:

State
$$state_t = a_t \odot i_t + f_t \circ state_{t-1} \quad (5)$$

Output
$$out_t = tanh(state_t) \odot o_t \quad (6)$$

The cell's output is sent upwards as a final output variable, and also sent to the cell on its right hand side to continue forward propagation. The next cell can then repeat this entire process.

The series of computations described by Step 1, 2 & 3 is performed by every cell in the row, one-by-one until the last cell is reached. This results in a series of output vectors produced, which represent the probabilities for data classification.

6.1.3 Back Propagation

Our aim during back propagation in an LSTM network is to update the sets of weights W and U and the biases B which are shared amongst the entire row, based on the error of the outputs produced during forward propagation. As always, back propagation of a single row of cells will unsurprisingly begin at the last cell, propagating to the left to gradually finish at the first cell - during this, we calculate a set of delta gradients to know the rate of change that should be applied to the weights and biases.

Lets describe and work through how to carry out back propagation in a single cell. Let $\circ$ denote multiplication, $\odot$ denote element-wise multiplication, and $\otimes$ denote the outer product.

1. Understanding The Initial Variables

We first state our input variables we have been given and the initial set of output variables we aim to produce.

Each cell has access to the following input variables:

The variables below have been computed and passed back from the next cell on the RHS.

- Delta Output Δout_T
- Delta State $\delta state_{T+1}$

Additionally, we have access to some pre-existing variables from the next and previous cells.

- Forget Gate Variable f_{T+1} - from the next cell
- State $state_{T-1}$ - from the previous cell

And of course, we have access to the variables which each cell has produced during forward propagation.

- The cell's input: x_t
- The cell's state: $state_{t-1}$
- The cell's output: out_t

2. Computing The Current Cell's Delta Variables

Given the above variables, we first need to find the following delta variables:

Output Difference	$\Delta_T = out_T - desiredout_T$	(1)
Delta State	$\delta state_T = (\Delta_T + \Delta out_T) \circ o_T \circ (1 - tanh(state_T)^2) + \delta state_{T+1} \circ f_{T+1}$	(2)
Delta Activation Gate Variable	$\delta a_T = \delta state_T \circ i_T \circ (1 - {a_T}^2)$	(3)
Delta Input Gate Variable	$\delta i_T = \delta state_T \circ a_T \circ i_T \circ (1 - i_T)$	(4)
Delta Forget Gate Variable	$\delta f_T = \delta state_T \circ state_{T-1} \circ f_T \circ (1 - f_T)$	(5)
Delta Output Gate Variable	$\delta o_T = \delta state_T \circ tanh(state_T \circ o_T \circ (1 - o_T)$	(6)

Although the number of variables we deal with may seem overwhelming, the equations themselves are straightforward to follow.

2. Computing The Previous Cell's Delta Output Δout_{t-1}

Moving on, we now have all the necessary components to compute the previous cell's delta output Δout_{T-1} - this is what will be back propagated to the previous cell. First we will need to do some preparation where we concatenate our weights and gate variables to achieve the required dimensions to work with.

Let the variable $\delta gates_t$ of a cell be the concatenation of all delta gate variables found in the cell. As each of these delta gate variables are vectors of length H, this results in a vector of length 4H.

- $\delta gates_t = \delta a_t +\!\!+ \ \delta i_t +\!\!+ \ \delta f_t +\!\!+ \ \delta o_t \in \mathbb{R}^{4H}$

Let the weight matrix variables W and U each be the concatenation of their corresponding four weight matrices transposed. As every matrix $w \in W$ is of dimension DxH, this will result in a 4DxH matrix. Every matrix $u \in U$ being of dimension HxH, means this will become a 4HxH matrix.

- $W = W_a +\!\!+ \ W_i +\!\!+ \ W_f +\!\!+ \ W_o \in \mathbb{R}^{4DxH}$
- $U = U_a +\!\!+ \ U_i +\!\!+ \ U_f +\!\!+ \ U_o \in \mathbb{R}^{4HxH}$

Finally, we can use (7) below to compute the Δout_{T-1}.

Delta Output Of Previous Cell
$$\Delta out_{T-1} = U^T \bullet \delta gates_t \tag{7}$$

To complete the process of back propagation in a single cell, the cell has to pass back Δout_{T-1} along with $\delta state_t$ to the previous cell, which will carry out **Step 1** and **Step 2**. This continues until reaching the first cell.

3. Updating The Network's Hyperparameters

After all cells have completed back propagating, the only thing left to do is update the entire row's weights and biases. This first requires computing the deltas of each of these; this consists of summing over the certain properties of every cell. Here

Delta W
$$\Delta W = \sum_{t=0}^{T} \delta gates_t \otimes x_t \tag{9}$$

Delta U
$$\Delta U = \sum_{t=0}^{T-1} \delta gates_{t+1} \otimes out_t \tag{10}$$

Delta B
$$\Delta B = \sum_{t=0}^{T} \delta gates_{t+1} \tag{11}$$

Finally, we update our hyperparameters by multiplying each of the above deltas by a learning rate η and then subtracting them from the original hyperparameters.

W Weights Update
$$W^{new} = W^{old} - \eta \circ \delta W^{old} \tag{12}$$

U Weights Update
$$U^{new} = U^{old} - \eta \circ \delta U^{old} \tag{13}$$

B Biases Update
$$B^{new} = B^{old} - \eta \circ \delta B^{old} \tag{14}$$

6.2 Modelling An LSTM Network

6.2.1 Finding A Data Structure

1. Variable/Hyperparameter Types

Let's start with defining the variables and hyperparameters we will be dealing with. In total there are two weight sets W and U and one bias set B which are shared among all cells in a row. Each weight holds four matrices and similarly the bias holds four vectors.

- The set of W weights, consisting of W_a, W_i, W_f, W_o
- The set of U weights, consisting of U_a, U_i, U_f, U_o
- The set of B bias vectors b_a, b_i, b_f and b_o

We will later need to have access to any of the individual four variables of each hyperparameter when required. One solution is to create a record data type with fields associated with each variable. However, this unnecessarily shifts it away from being a mathematical structure to something more object oriented. Looking at the equations in which these are used, it to make sense to use 3D matrices instead. However, if we already know that all weights and biases will hold exactly four variables, it is a lot more type-safe to make them fixed sized vectors of 2D or 1D arrays respectively. This of course means we need to maintain a strict ordering of their variable subscripts - we use the order a, i, f, o. We also make a convenient type for hyperparameters, containing the weights W, weights U, and biases B in that order.

```
type Weights = V.Vec4 [[Double]]
type Biases  = V.Vec4 [Double]
type HyperParameters = (Weights, Weights, Biases)
```

The same reasoning used for weights and biases also applies to the gate variables of each cell. We can represent the gates of a cell as a fixed-sized four length vector of arrays, where each array corresponds to a_t, i_t, f_t or o_t.

```
type Gates    = V.Vec4 [Double]
```

The state is a single 1D array.

```
type State    = [Double]
```

Finally, each input x_t for a cell is a 1D array and each output out_t is a 1D array.

```
type Input    = [Double]
type Output   = [Double]
type DesiredOutput = [Double]
```

2. Cell Data Type

For the cell structure, it makes sense to begin with adopting our previous approach to modelling layers when creating data types for cells, and see where this leads us.

```
data Cell k = Cell { _cellState :: [Double],
                     _innerCell :: k }
            | InputCell
```

We therefore define the data type `Cell` with the constructors `Cell` which contains only a state and the next cell, and base case `InputCell`.

3. Layer Data Type

A layer is a row of cells. Although we won't perform any recursion on layers right now, we need a suitable data type to hold the cells along with the shared weights and biases (hyperparameters) - otherwise when updating the weights and biases during the anamorphic reconstruction of the network structure, we will have no appropriate way to store them.

```
data Layer = Layer { _hyperparams :: HyperParameters,
                     _cells       :: Fix Cell
                   }
```

As we are currently only dealing with a single layer of cells, we can define this as a non-recursive data type for now, given by `Layer`.

4. Forward Propagation Data Type

We now discuss how we should define our carrier type during forward propagation. Due to the large amount of variables being handled it is necessary to create a data type `ForwardProp` to hold all of these.

Let's list what fields will need to exist in this data type to enable forward propagation:

- The variables which are forward propagated are the previous cell's **state** $state_t$ and **output** out_t.
- The list of all **input variables** x_t and their **desired output** $desiredout_t$. We will adopt our previous approach of passing a list of inputs which each subsequent cell uses the head element and then passes forward the tail, except this time we use a list of tuples containing the input and their desired output.
- We need to access the **weights and biases** when working with cells, however these hyperparameters are stored in the layer and thus out of scope. To resolve this, they will also be included in the forward propagation data type.

Remember that it is also important to think about what other variables computed during forward propagation need to be accessed later during back propagation. These are:

- The cell's **input** x_t and **previous cell's output** out_{t-1}
- The cell's **output** out_t and **desired output** $desiredout_t$

- The cell's **gate output variables** $gates_t$

This leaves us with the following data type for forward propagation.

```
data ForwardProp = ForwardProp { _gates       :: Gates,
                                 _input       :: Input,
                                 _des_out     :: DesiredOutput,
                                 _output      :: Output,
                                 _state       :: State,
                                 _params      :: HyperParameters,
                                 _inputStack  :: [(Input, DesiredOutput)]
                               } deriving Show
```

5. Back Propagation Data Type

Finally, we need a data type for back propagation for the same purpose. The variables which are back propagated from a cell include the following:

- The cell's delta state: $\delta state_t$
- The cell's delta output: Δout_t
- The cell's delta gates: $\delta gates_t$
- The cell's forget gate variable: f_t

We denote this data type as `BackProp` like usual.

```
data BackProp = BackProp { _nextDState    :: [Double],
                           _nextDOut      :: [Double],
                           _nextDGates    :: Gates,
                           _nextF         :: [Double] }
```

6.2.2 Implementing Forward Propagation

We begin modelling forward propagation in an LSTM network as a catamorphism by defining an algebra for it. Similar to the previous chapter, we choose the carrier type to be a 2-tuple contain a type `Fix Cell` and a function of type `[ForwardProp] -> [ForwardProp]`. This results in the algebra having the following type definition.

```
algCell :: Cell (Fix Cell, [ForwardProp] → [ForwardProp]) → (Fix Cell, [ForwardProp] → [ForwardProp])
```

We can break down the algebra into four main steps:

1. Compute the gate variables
2. Compute the state
3. Compute the output
4. Finishing the algebra: Update the state and forward propagation function

We will now work through each of them one-by-one.

1. Computing The Gate Variables

The equations which describe the computation each gate applies to its input is restated below.

Activation Gate Variable $$a_t = tanh(W_a \circ x_t + U_a \circ out_{t-1} + b_a) \quad (1)$$

Input Gate Variable $$i_t = \sigma(W_i \circ x_t + U_i \circ out_{t-1} + b_i) \quad (2)$$

Forget Gate Variable $$f_t = \sigma(W_f \circ x_t + U_f \circ out_{t-1} + b_f) \quad (3)$$

Output Gate Variable $$o_t = \sigma(W_o \circ x_t + U_f \circ out_{t-1} + b_f) \quad (4)$$

It can be observed that each equation shares an pattern of containing $(W_\gamma \circ x_\gamma + U_\gamma \circ out_{t-1} + b_\gamma)$.

The best approach would be to create a function `compGates`, which uses this pattern to generalize over all gates before applying their individual activation functions.

```
compGates :: HyperParameters → [Double] → [Double] → Gates
compGates (weightW, weightU, bias) input prev_out
    = V.zipWith ($) activations preactivated_gates
      where mvmul' = flip mvmul
            activations = (V.fromList [map sigmoid, map sigmoid, map tanh, map sigmoid])
            preactivated_gates = eleadd3v (V.map (mvmul' input) weightsW)
                                          (V.map (mvmul' prev_out) weightsU)
                                          (biases)
```

We can achieve a shared computation by mapping over the weights with a function which performs a matrix-vector multiplication with either the input or the previous cell's output, followed by adding all three components $W_\gamma \circ x_\gamma$, $U_\gamma \circ out_{t-1}$, and b_γ. This will compute the described pattern for a single gate variable. Doing this for each gate gives us `preactivated_gates`.

Afterwards, the activation functions can be applied; given the gate variables, we create a list of activation functions in correct order and then apply `zipWith` between them using function application `($)`.

2. Computing The State

The state is trivial to compute, consisting of simple element-wise vector multiplication and addition using three of the recently acquired gate variables and the previous cell's state.

State $$state_t = a_t \odot i_t + f_t \circ state_{t-1} \tag{5}$$

The define this function as `compState`.

```
compState :: Gates → [Double]
compState gates prev_cell_state
    = eleadd (elemul a i) (elemul f prev_cell_state')
      where prev_cell_state' = case prev_cell_state of [] → replicate (length gate_a) 0
                                                       xs → xs
            gate_f = gates ! 0
            gate_i = gates ! 1
            gate_a = gates ! 2
```

In this function, we extract the variables f_t, i_t, and a_t from the gates. When using the previous cell's state, it is important to account for the situation where there is no previous cell - in this case, we work with the assumption that we have instead been provided an empty list, allowing us to pattern match on this and use an array of zeros in its place. The computation can be seen to be translated directly from the equation (5) in the returned result.

3. Computing The Output

The output is also straightforward to define, seen by the below equation.

Output $$out_t = tanh(state_t) \odot o_t \tag{6}$$

We call this function `compOutput`.

```
compOutput :: [Double] → Gates → [Double]
compOutput state gates = elemul (map tanh state) gate_o
    where gate_o = (gates ! 3)
```

This translates equation (6) directly, performing element-wise vector multiplication between the tanh of the state we have just computed and the output gate variable o_t.

4. Finishing The Algebra: Update The State And Forward Propagation Function

We finish writing the algebra by constructing an updated forward propagation function which composes with the old function `forwardPropfunction`.

```
algCell :: Cell (Fix Cell, [ForwardProp] → [ForwardProp]) → (Fix Cell, [ForwardProp] → [ForwardProp])
algCell (Cell state (nextCell, forwardPropFunction)) =
    let forwardPropFunction' = (\forwardPropStack →
        let = fp                  = head forwardPropStack
              (input', des_out')  = head (fp ^. inputStack)
              gates'              = compGates (fp ^. params) x (fp ^. output)
              state'              = compState gates (fp ^. state)
              output'             = compOutput gates state'
              fp'                 = fp & state .~ state'
                                       & input     .~ input'
                                       & prev_out  .~ (fp ^. output)
                                       & output    .~ output'
                                       & des_out   .~ des_out'
                                       & gates     .~ gates'
                                       & inputStack %~ tail
        in (fp':forwardPropStack)) . forwardPropFunction
    in (Fix (Cell state' nextCell), forwardPropFunction')
```

In this function, we acquire all of the necessary variables using the previous functions `compGates`, `compState` and `compOutput`, and use these to create an updated `ForwardProp` value to append this to the existing list. Finally, we return this new function alongside the cell with an updated state.

However, notice that this won't compile because the updated state `state'` is computed inside the scope of the new forward propagation function, and is therefore not in reach. To resolve this, we will proceed with an unupdated state during the algebra and instead update it during back propagation in the coalgebra; this is possible due to the updated states being stored in the list of `ForwardProp` values we accumulate.

Therefore, we edit the last line to be:

```
    in (Fix (Cell state nextCell), forwardPropFunction')
```

As a final note, let's not forget the algebra for the input cell.

```
algCell :: Cell (Fix Cell, [ForwardProp] → [ForwardProp]) → (Fix Cell, [ForwardProp] → [ForwardProp])
algCell InputCell = (Fix InputCell, id)
```

As always, this will return the fix of the input cell and the identity function.

6.2.3 Implementing Back Propagation

Next we shall model backward propagation in an LSTM network as an anamorphism by defining a coalgebra for cells. We use the usual convention for the back propagation carrier type, which is a tuple containing: the fix of the cell, the list of forward propagation values produced from the previous algebra, and the back propagation data type.

```
coalgCell :: (Fix Cell, [ForwardProp], BackProp) → Cell (Fix Cell, [ForwardProp], BackProp)
```

We can break down the cell's coalgebra into five main steps:

1. Compute the output difference and delta state
2. Compute the deltas of the gate variables
3. Compute the delta output for the previous cell
4. Compute the deltas of the hyperparameters for one cell
5. Completing the coalgebra

Afterwards, we need to take care of back propagation on the layer level, which can be done in two more steps:

6. Computing the total hyperparameter deltas over all cells
7. Update the layer hyperparameters

We will now work through each step one by one.

1. Compute The Output Difference (Δ_t) And Delta State ($\delta state_t$)

Let us restate the equations needed to acquire these variables.

Output Difference $$\Delta_T = out_T - desiredout_T \tag{1}$$

Delta State $$\delta state_T = (\Delta_T + \Delta out_T) \circ o_T \circ (1 - tanh(state_T)^2) + \delta state_{T+1} \circ f_{T+1} \tag{2}$$

The function `compOutputDiff` first computes the output difference Δ_t by subtracting the desired output from the output of a cell as seen in equation (1).

```
compOutputDiff :: [Double] → [Double] → [Double]
compOutputDiff output desired_output = elesub output desired_output
```

We then create `compdState` which finds the delta state ($\delta state_t$) using (2).

```
compdState :: [Double] → [Double] → [Double] → [Double] → [Double] → [Double] → [Double]
compdState o_gate state next_cell_dState next_cell_f_gate next_cell_dOut output_diff =
    let g = case next_cell_dOut
                of ([])                → error
                   (next_cell_dOut')   → eleadd next_cell_dOut' error
    in  eleadd (elemul3 g o_gate (map (\x → x − 1) . sqr. tanh) (state))
               (elemul next_cell_dState next_cell_f_gate)
```

This takes the variables o_t, $state_t$, $\delta state_{t+1}$, f_{t+1}, Δout_t, and Δ_T, to perform a computation seen in the final line which directly represents equation (2). However, remember that the variable Δout_T is something which is computed and passed back by the next cell. If we are already at the last cell, we work under the assumption that pattern matching on the value Δout_T will identify as an empty list - in this scenario, we can ignore its presence in the equation.

2. Compute The Delta Gate Variables: (δa_t), (δi_t), (δf_t), (δo_t)

The deltas of each gate are given by the following equations.

Delta Activation Gate Variable $$\delta a_T = \delta state_T \circ i_T \circ (1 - {a_T}^2) \tag{4}$$

Delta Input Gate Variable $$\delta i_T = \delta state_T \circ a_T \circ i_T \circ (1 - i_T) \tag{5}$$

Delta Forget Gate Variable $$\delta f_T = \delta state_T \circ state_{T-1} \circ f_T \circ (1 - f_T) \tag{6}$$

Delta Output Gate Variable $$\delta o_T = \delta state_T \circ tanh(state_T \circ o_T \circ (1 - o_T) \tag{7}$$

These are very straightforward to translate, where each of the four equations can be seen to correspond to a line of code in `compdGates`.

```
compdGates :: Gates → [Double] → [Double] → [Double] → [Double]
compdGates gate dState state prev_cell_state
    = let   d_f       = elemul4 dState (gate ! 1) prev_cell_state  (map sub1 (gate ! 1))
            d_i       = elemul4 dState (gate ! 2) (gate ! 3)  (map sub1 (gate ! 2))
            d_a       = elemul3 dState (gate ! 2) (map (sub1 . sqr) (gate ! 3))
            d_o       = elemul4 dOut  (gate ! 4) (map tanh state)  (map sub1 (gate ! 4))
      in    d_f ++ d_i ++ d_a ++ d_o
```

The last line of `compdGates` denotes the concatenation of all delta gate variables to form $\delta gates_t$.

3. Compute The Delta Output For The Previous Cell (Δout_{t-1})

Next, we shall define the computation for the delta output Δout_{t-1} to be back propagated to the previous cell.

Delta Output Of Previous Cell
$$\Delta out_{T-1} = U^T \bullet \delta gates_t \tag{8}$$

We call this function `compDOut`.

```
compDOut :: V.Vec4 [[Double]] → [Double] → [Double]
compDOut weightsU deltaGates =
    let weightsU' = (V.foldr (++) [[]]) weightsU
    in  mvmul (transpose weightsU) deltaGates
```

We first need to change the weights U from the type `V.Vec4 [[Double]]` to the type `[[[Double]]]`; this allows us to transpose it and multiply it with the delta gates to give Δout_{t-1}.

4. Compute The Hyperparameter Deltas (ΔW), (ΔU), And (ΔB)

Below we give the equations to compute each of these hyperparameter deltas. In general, they work by summing over certain variables produced by each cell.

Delta W
$$\Delta W = \sum_{t=0}^{T} \delta gates_t \otimes x_t \tag{9}$$

Delta U
$$\Delta U = \sum_{t=0}^{T-1} \delta gates_{t+1} \otimes out_t \tag{10}$$

Delta B
$$\Delta B = \sum_{t=0}^{T} \delta gates_{t+1} \tag{11}$$

This step leaves a slight issue due to how an anamorphism works by constructing the network from the last cell to the first cell, i.e. it starts from the outside of the structure and works its way in. If we try to compute the delta hyperparameters during our back propagation, this means the final value would have to be computed and stored in the inner most cell. To access this, we would have to travel to the most inner cell after the anamorphism is finished - this seems like a bit of a strange process. A more natural way would be to instead compute and store these deltas in each corresponding cell during the anamorphism. Afterwards, we can perform another catamorphism which sums up all of the delta values.

Therefore, let's create a new data type `Deltas` which holds the delta hyperparameters for a single cell. We also update the data type for cells to contain a new field `cellDeltas` to store these deltas.

```
data Deltas = Delta { deltaW :: [[Double]],
                      deltaU :: [[Double]],
                      deltaB :: [Double]   }

data Cell k = Cell { _cellState  :: [Double],
                     _cellDeltas :: Deltas,
                     _innerCell  :: k       }
              Inputcell
```

Let δW_t, δU_t, and δB_t of a single cell be one of the components of the summations which make up ΔW, ΔU and ΔB. We'll now make a function `compdWUB` to compute this for a single cell.

```
compdWUB :: [Double] → [Double] → [Double] → [Double] → Deltas
compdWUB dGates dGates_next input output =
    Delta dW dU dB
    where dW = outerProduct dGates input
          dU = case dGates_next of [] → fillMatrix (length dGates) (quot (length dGates 4) 0.0
                                   _  → outerProduct dGates_next output
          dB = dGates
```

The variables δW_t and δB_t are simple to define. When computing δU_t however, notice that it uses $\delta gates_{t+1}$ in its equation - this means that the last cell in a row does not contribute to the summation. Thus we need to

check whether we are dealing with the last cell in the row; this can be verified by seeing if the next cell's delta gates is an empty list of not.

5. Finishing The Coalgebra

This lets us complete the coalgebra for the cell.

```
coalgCell :: (Fix Cell, [ForwardProp], BackProp) → Cell (Fix Cell, [ForwardProp], BackProp)
coalgCell (Fx InputCell, forwardProps, backProp)
  = InputCell
coalgCell (Fx cell, forwardProps, backProp)
  = let fp = head forwardProps
        lastState                = (head (tail forwardProps)) ^. state
        (gate, updatedState)     = (fp ^. gates, fp ^. state)

        BackProp dState_next dOut_next dGates_next f_next _ _ = backProp

        -- Part 1.
        outputDiff = compOutputDiff (fp . output) (fp ^. des_out)
        dState     = compdState (gate ! 4) updatedState dState_next gate_f_next dOut_next outputDiff
        -- Part 2.
        dGates     = compdGates gate dState updatedState lastState
        -- Part 3.
        dOut       = compDOut weightsU dGates
        -- Part 4.
        dWUB       = compdWUB dGates dGates_next (fp ^. input) (fp ^. output)

        backProp'  = backProp & nextDState .~ dState
                              & nextDOut  .~ dOut
                              & nextDGates .~ dGates
                              & nextF      .~ (gate ! 1)


    in  (cell & cellState .~ updatedState
              & cellDeltas .~ dWUB
              & innerCell .~ (fromJust (cell ^? innerCell), tail forwardProps, backProp'))
```

We first combine all of the previous four steps to acquire the desired variables, as annotated in the code comments. We use all of these to then update the back propagation data type to be used by the previous cell. Finally, we return a newly constructed cell with an updated state and deltas, as well as the tail of the forward propagation value list and the new back propagation value.

6. Computing Total Hyperparameter Deltas Over All Cells

We are now at the stage where every cell in the network has stored its own (δW_t), (δU_t), and (δB_t) values. The next step is to perform a catamorphism over the network which performs a summation over these deltas.

Lets therefore create a second algebra for the cells. We choose the carrier type to be a tuple containing the fix cell, and a function which takes delta hyperparameters as input to delta hyperparameters as output.

```
algCell2 :: Cell (Fix Cell, Deltas → Deltas) → (Fix Cell, Deltas → Deltas)
```

In this algebra, we need to implement the following set of equations.

Delta W
$$\Delta W = \sum_{t=0}^{T} \delta gates_t \otimes x_t \tag{9}$$

Delta U
$$\Delta U = \sum_{t=0}^{T-1} \delta gates_{t+1} \otimes out_t \tag{10}$$

Delta B
$$\Delta B = \sum_{t=0}^{T} \delta gates_{t+1} \tag{11}$$

As usual, the input cell will return the fix of itself and the identity function. All other cells will create a new function of type `Deltas -> Deltas` which composes with the existing function `deltaTotalFunc`.

```
algCell2 :: Cell (Fix Cell, Deltas → Deltas) → (Fix Cell, Deltas → Deltas)
algCell2 InputCell
  = (Fx InputCell, id)
algCell2 (Cell state deltas (innerCell, deltaTotalFunc))
  = let Deltas dW dU dB = deltas
        deltaTotalFunc' = (\deltaTotal →
            let Deltas deltaW deltaU deltaB = deltaTotal
                deltaW_total = eleaddM dW deltaW
                deltaU_total = eleaddM dU deltaU
                deltaB_total = eleadd dB deltaB
            in Deltas deltaW_total deltaU_total deltaB_total) . deltaTotalFunc
    in (Fx (Cell state deltas innerCell), deltaTotalFunc')
```

This new function works by taking the accumulated summation of delta hyperparameters and adding on the current cell's delta hyperparameters onto it. At the end, this will return the total summation of delta hyperparameters.

7. Updating The Layer Hyperparameters

Finally, we must update the weights and biases of the layer, given by the following equations.

W Weights Update
$$W^{new} = W^{old} - \eta \circ \delta W^{old} \tag{12}$$

U Weights Update
$$U^{new} = U^{old} - \eta \circ \delta U^{old} \tag{13}$$

B Biases Update
$$B^{new} = B^{old} - \eta \circ \delta B^{old} \tag{14}$$

We define a function `updateHyperparameters`.

```
updateHyperparameters :: Layer k → Deltas → Layer k
updateHyperparameters layer delta_total
    = let Deltas deltaW_total deltaU_total deltaB_total = delta_total
          (weights_w,weights_u,biases) = fromJust $ layer ^? hparams
          w = concat $ V.toList weights_w
          u = concat $ V.toList weights_u
          b = concat $ V.toList biases
          w'      = V.fromList $ map cons $ elesubm w (map2 (0.1 *) deltaW_total)
          u'      = V.fromList $ map cons $ elesubm u (map2 (0.1 *) deltaU_total)
          b'      = V.fromList $ map cons $ elesub  b (map (0.1 *)  deltaB_total)

      in layer & hparams .~ (w', u', b')
```

Given the total deltas of the hyperparameters previously computed, we must first convert the weights and biases from a vector of 2D arrays into a 3D array so that we can perform matrix operations easily. This gives us `w`, `u` and `b`. Afterwards, we use `map` to multiply the learning rate η (which is 0.1) with each delta hyperparameter, and then subtract this result from our current hyperparameters. Finally, we convert the weights and biases back to a vector of 2D arrays, giving us `w'`, `u'` and `b'`, and return the layer with its updated hyperparameters.

6.2.4 Training An LSTM Network

At last, we can put all of this together and connect the forward and back propagation stages.We begin by first creating functions which can give us initial values for the data types of `ForwardProp`, `BackProp`, and `Deltas`.

```
initForwardPropValue :: Int → Int → HyperParameters → Input → ForwardProp
initForwardPropValue h d params sample
    = ForwardProp (V.fromList ( replicate 4 [])) [] [] [] ( replicate h 0.0) ( replicate h 0.0) params sample

initBackPropValue :: Int → Int → BackProp
initBackPropValue h d
    = BackProp (replicate h 0) ( replicate h 0) ( replicate (h*d) 0) ( replicate h 0) []

initDeltaValue :: Int → Int → Deltas
initDeltaValue h d
    = Deltas ( fillMatrix (4 * h) (d) 0.0) ( fillMatrix (4 * h) (h) 0.0) ( replicate (4 * h) 0.0) [[]]
```

Given the dimensions H and D we are working with, we construct these initial values by either providing existing variables we have access to, such as the hyperparameters and input data, or instead using fields of empty matrices and zero-filled matrices for variables which do not yet exist. Note that we can get these dimensions from the weight matrices of W which will be of dimensions HxD.

Next we summarise the training process into five main steps.

1. **We initialise values for data types `ForwardProp`, `BackProp` and `Deltas`.**
2. **We perform a catamorphism to execute forward propagation.** This returns us a new set of cells as well as a forward propagation function which we apply to the initial `ForwardProp` list in order to get a new resulting list of `ForwardProp` values.
3. **We perform an anamorphism to reconstruct an updated network of cells.** This computes the delta hyperparameters for each cell in stores in the corresponding cell.
4. **We then run another catamorphism using the second algebra we defined.** This gives us a function which lets us evaluate the total summation of the delta hyperparameters stored in each cell.
5. **Finally, we update the hyperparameters and cells of the layer.** We call `updateHyperParameters` to return the updated layer with new weights and biases.

Note that the anamorphism from step 3 and the catamorphism from step 4 can actually be composed to form a **hylomorphism** as discussed in 3.8. The definition for hylomorphisms is given below.

```
hylo :: Functor f ⇒ (a → f a) → (f b → b) → a → b
hylo coalg alg = cata alg . ana coalg
```

The training process of an LSTM network using a single sample can be viewed below in `runLayer`.

```
runLayer :: Layer → Input → Layer
runLayer (Layer params cells) sample =
  let -- Step 1
      w_weight = (params ^. _1)
      (h_dim, d_dim) = (length (w ! 1), length (head (w ! 1)))
      initialFPValue = initForwardPropValue h_dim d_dim params sample
      initialBPValue = initBackPropValue h_dim d_dim
      initialDeltaValue = initDeltaValue h_dim d_dim
      -- Step 2
      ( cells', fpFunc) = cata algCell cells
      fpStack = fpFunc [initialFPValue]
      -- Step 3 & 4
      ( cells'', deltaFunc) = hylo algCell2 coalgCell ( cells', fpStack, initialBPValue)
      deltaTotal = deltaFunc initialDeltaValue
      -- Step 5
      updatedLayer = updateParameters (Layer params cells'') deltaTotal
  in updatedLayer
```

Each of the five steps described can be seen clearly annotated by the code comments. Exploring recurrent networks has interestingly uncovered a new unexpected recursive behaviour - the recursion scheme used to updated a single layer of cells is no longer a metamorphism, but rather can be seen as composing a cata with a hylo. This is illustrated by the diagram belows, where we use the diagrams for catamorphisms and hylomorphisms, and tailor them to our implementation.

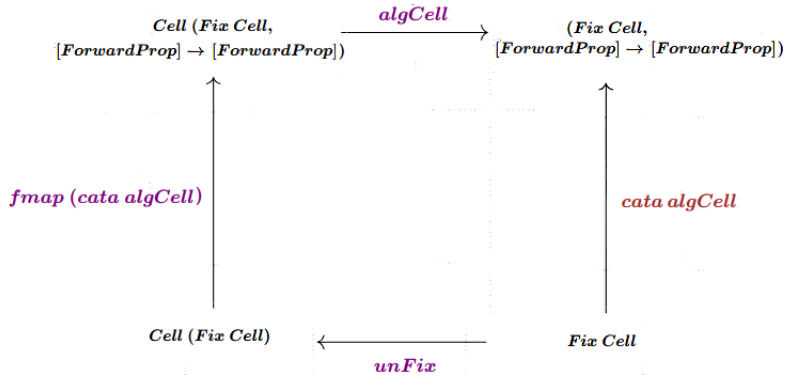


FIGURE 6.1: Cell Catamorphism

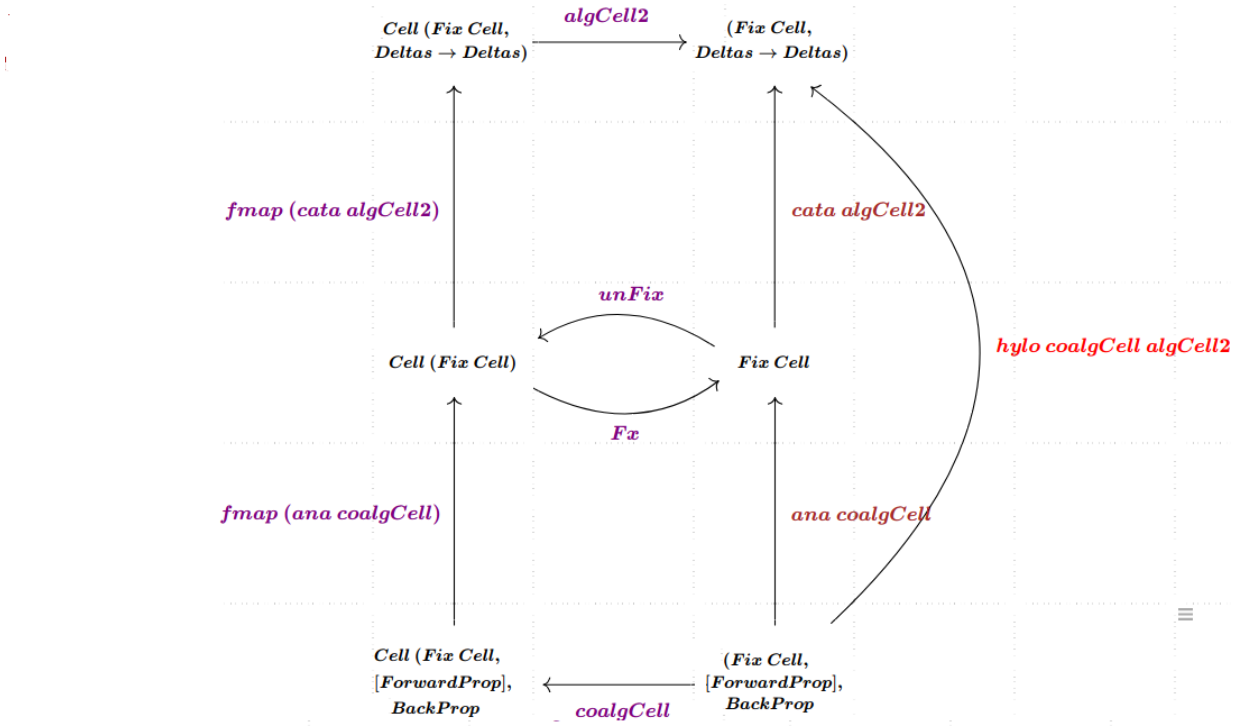


FIGURE 6.2: Cell Hylomorphism

We can take this even further in the next section by implementing forward and back propagation at the layer level.

6.3 Deep LSTMs

We will now start looking at how at extend our system to a deep LSTM network. As per the previous section, let's first outline their structure, and how forward and back propagation differ from an LSTM network. This will give us an idea of how to implement a deep LSTM network using a recursion scheme system.

6.3.1 Structure

The structure of a deep LSTM network consists of multiple vertically stacked layers of cells - in other words, we can take numerous instances of our previous LSTM network and simply connect them up such that their cells align. This means the outputs produced by one LSTM network are used as inputs to another LSTM network. Each of these layers has its own set of hyperparameters which are shared between its cells. Although no new components or variables are introduced, we now encounter a new level of recursion in the layer data type.

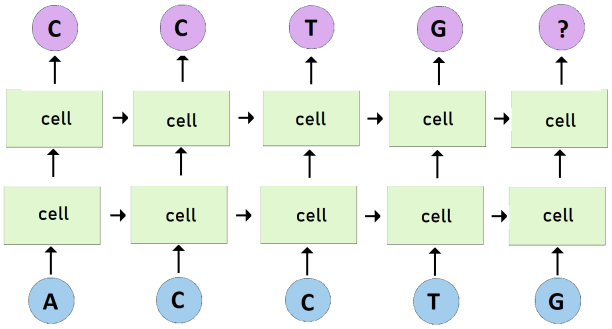


FIGURE 6.3: Deep LSTM

6.3.2 Forward Propagation

Forward propagation in deep LSTM networks works as it can seen in the above figure. After one layer of cells has finished forward propagation on the cellular level, each of these outputs are simply fed as inputs to the aligned cells in the next layer to be processed the same way.

6.3.3 Back Propagation

When back propagating, the process changes slightly more. Back propagation intends to compute the relevant gradients needed to tune the hyperparameters according to the cost incurred by the output error. However this time we are back propagating in two directions - vertically through the layers, and horizontally through the cells. This means we must also compute the cost gradient with respect to the layer's input and involve this variable when updating the layer's hyperparameters, as well as passing this variable back to the previous layer.

To do this, we need to compute a new variable δx_t for each cell when back propagating through cells. This is acquired by multiplying the transpose of the W weight matrix W^T with the delta gates $\delta gates_t$.

Delta Input
$$\delta x_t = W^T \bullet \delta gates_t \tag{1}$$

This then affects the computation of the output difference Δ_t; if we are not at the last layer, we use the δx_t computed by the next layer's adjacent cell.

Output Difference At Output Layer
$$\Delta_T = out_t - desiredout_t \tag{2}$$

Output Difference At Other Layers
$$\Delta_T = out_t - out_t \odot (-\delta x_t)^{L+1} \tag{3}$$

We then back propagate each cell's δx to be used as the error Δ_T to the previous row of cells.

6.4 Modelling A Deep LSTM Network

6.4.1 Creating A Data Type

We already have a data type for layers, but now we are going to extend it to become a functor parameterized by some carrier type `k`.

```
data Layer k = Layer { _hyperparams :: HyperParameters,
                       _cells       :: Fix Cell,
                       _innerLayer  :: k}
             InputLayer deriving Functor
```

This involves adding a new field `innerLayer` to the `Layer` constructor to store the carrier type, as well as introducing an additional constructor `InputLayer` as the base case. This means we can now nest layers inside each other.

With regards to the forward and back propagation data types for layers, we will not be introducing any new data types to represent these. We can instead conveniently use the existing definitions which cells use. How we actually do this will become clear in the upcoming subsections.

6.4.2 Implementing Forward Propagation

We begin modelling forward propagation in a deep LSTM network as a catamorphism by defining an algebra for it. Let's first decide the carrier type for the algebra. The algebra for cells uses a function which produces a type `[ForwardProp]` in its carrier, which represents the list of forward propagated variables in a layer's cells. Hence, it makes sense that the algebra for layers should use a function which outputs the type `[[ForwardProp]]`, representing the list of all layers' list of forward propagated variables. This leaves the algebra's type definition to be:

```
algLayer :: Layer (Fix Layer, [[ForwardProp]] → [[ForwardProp]])
          → (Fix Layer, [[ForwardProp]] → [[ForwardProp]])
```

When implementing the pattern matching for `InputLayer`, this will simply return the input layer and the identity function as usual.

```
algLayer :: Layer (Fix Layer, [[ForwardProp]] → [[ForwardProp]])
          → (Fix Layer, [[ForwardProp]] → [[ForwardProp]])
algLayer InputLayer = (Fx InputLayer, id)
```

When implementing the pattern match for `Layer`, the main part of this is constructing an updated forward propagation function which composes with `forwardPropFunction`. This is essentially copy and pasting the first half of the function `runLayer` defined for non-deep LSTMS. We can break the implementation down into two steps:

1. **Extract the previous layer's outputs and corresponding labels.** We use these to initialise a `ForwardProp` value for the current layer.
2. **Perform a catamorphism on the current layer's cells to carry out forward propagation on the cell level.** This returns a function which lets us acquire the layer's `[ForwardProp]` value which we append to the old list of type `[[ForwardProp]]`.

These steps can be seen labelled in the comments of the function `algLayer` below.

```
algLayer :: Layer (Fix Layer, [[ForwardProp]] → [[ForwardProp]])
          → (Fix Layer, [[ForwardProp]] → [[ForwardProp]])
algLayer (Layer params cells (innerLayer, forwardPropFunction))
  = let forwardPropFunction' = (\layerForwardPropStack →
      let -- Step 1
          fp                = head layerForwardPropStack
          inputs_labels     = map toTuple2 $ chunksOf 2 $ fp <**> [(^.output),(^.label)]
          (h_dim, d_dim)    = let (w, u, b) = params in (length $ w ! 1), length $ head $ w ! 1)
          initialForwardProp = initForwardProp h_dim d_dim params inputs_labels

          -- Step 2
          (cells', fpFunc)  = cata algCell cells
          layerForwardProp  = fpFunc [initialForwardProp]
      in (layerForwardProp:layerForwardPropStack)) . forwardPropFunction
    in (Fx (Layer params cells innerLayer), forwardPropFunction')
```

How these steps are carried out is further detailed below.

Step 1 The only information we are interested in from the previous layer are the cell's outputs and labels; we can extract these from the list of forward propagated values the previous layer produced. We do this by using the infix function `<**>` - in this scenario, this lets us take the list of `ForwardProp` values and a list of lens getter functions, and apply each getter function to each `ForwardProp` value, resulting in a sequence of outputs and desired outputs. We then pair up each corresponding output and desired output using `map toTuple 2 $ chunksOf 2` to create a list of tuples. We then use this to create an initial `ForwardProp` value to pass to the cells.

Step 2 Finally, we can call a catamorphism on the layer's cells using the cell algebra. This returns us a function of type `[ForwardProp] -> [ForwardProp]`; applying this to the initial `ForwardProp` value (wrapped in a list) lets us acquire the list of `ForwardProp` values for the entire layer's cells. We now have the final definition for the layer's algebra.

6.4.3 Implementing Back Propagation

We now move onto creating a coalgebra for the back propagation of layers. When considering the carrier type, we can actually use a similar approach to how the carrier of the cell coalgebra is defined: we of course need the type `[[ForwardProp]]` which holds every layer's forward propagated variables, as well as the type `BackProp`. This leaves us with the type definition below.

```
coalgLayer :: (Fix Layer, [[ForwardProp]], BackProp) → Layer (Fix Layer, [[ForwardProp]], BackProp)
```

• Preparation

Before we can start defining the coalgebra, we will need to account for the effect the new variable δx_t has on the existing implementation for cells. What we ultimately want to achieve, is a way of computing δx_t for every cell belonging to the current layer, so that we can store them in a list somewhere.

Updating Data Types

First, we need to modify the data type `Deltas`: we add a new field `deltaXs` which represents the list of all δx_t values computed for every cell in the current layer.

```
data Deltas = Delta { deltaW :: [[Double]],
                      deltaU :: [[Double]],
                      deltaB :: [Double],
                      deltaXs :: [[Double]]}
```

Second, although I said that we can use the existing `BackProp` data type for the carrier, we do need to make a minor change; every layer (apart from the output layer) needs to access the next layer's computed δx's as seen in equation (2). Hence we will add a new field `nextLayerDXs`; during cell back propagation, this field will be treated similarly to the `[[ForwardProp]]` carrier - by systematically returning the tail of this list, we can recognise that the head element will always correspond to the δx_t belonging to the above cell in the next layer.

```
data BackProp = BackProp { _nextDState :: [Double],
                           _nextDOut   :: [Double],
                           _nextF      :: [Double],
                           _nextDGates :: [Double],
                           _fpStack    :: [ForwardProp],
                           _nextLayerDXs :: [[Double]]    }
```

Moving on, we need to tweak a couple of functions which the cell coalgebra uses. Assume that the cell coalgebra has been updated as necessary to satisfy the arguments taken by the new function changes.

Computing Delta X

Here we include the computation of delta x in the old function `compdWUB` to give a new function `compdWUBX`.

Delta Input $$\delta x_t = W^T \bullet \delta gates_t \tag{1}$$

```
compdWUBX :: [[Double]] → [Double] → [Double] → [Double] → Deltas
compdWUBX weightsW dGates dGates_next input output =
    Delta dW dU dB [dX]
    where dW = outerProduct dGates input
          dU = case dGates_next of [] → fillMatrix (length dGates) (quot (length dGates 4) 0.0
                                   _  → outerProduct dGates_next output
          dB = dGates
          dX = mvmul (transpose weightsW) dGates
```

This is very simply the multiplication of the transpose of weights W with the delta gates.

Using Delta X

Here we see how we make use the next layer's delta x's, as described in the below equations when computing the output difference.

Output Difference At Output Layer $$\Delta_T = out_t - desiredout_t \tag{2}$$

Output Difference At Other Layers $$\Delta_T = out_t - out_t \odot (-\delta x_t)^{L+1} \tag{3}$$

We update the function `compOutputDiff` by adding a new argument `nextLayerDXs` which can be retrieved from the `BackProp` data type; this is the list of the next layer's δx_t values.

```
compOutputDiff :: [[Double]] → [Double] → [Double] → [Double]
compOutputDiff nextLayerDXs output desired_output =
      case nextLayerDxs of []          → elesub (fp^.output) (fp^.des_out)
                           nextLayerDxs' → elesub (fp^.output) (map ((−1) * ) (head nextLayerDxs'))
```

By pattern matching on `nextLayerDXs`, we can recognise if we are at the last layer through this field being the empty list - this tells us whether to use equation (2) or (3).

Collecting Delta X Values

After computing all δx_t's, we need to change the *second* cell algebra such that in addition to summing over all the cell delta hyperparameter values, it also accumulates a list of all cell δx_t values.

```
algCell2 :: Cell (Fix Cell, Deltas → Deltas) → (Fix Cell, Deltas → Deltas)
algCell2 InputCell
  = (Fx InputCell, id)
algCell2 (Cell state deltas (innerCell, deltaTotalFunc))
  = let Deltas dW dU dB dXs = deltas
        deltaTotalFunc' = (\deltaTotal →
            let Deltas deltaW deltaU deltaB deltaXs = deltaTotal
                deltaW_total = eleaddM dW deltaW
                deltaU_total = eleaddM dU deltaU
                deltaB_total = eleadd dB deltaB
                deltaXs'     = dXs ++ deltaXs
            in Deltas deltaW_total deltaU_total deltaB_total deltaXs') . deltaTotalFunc
    in (Fx (Cell state deltas innerCell), deltaTotalFunc')
```

This can be seen in the line `deltaXs' = dXs ++ deltaXs`.

• Creating The Layer Coalgebra

At last, we can begin defining the coalgebra for the layer back propagation - this is fortunately quite effortless to do. When considering the carrier type, we can actually use a similar approach to how the carrier of the

cell coalgebra is defined: we of course need the type `[[[ForwardProp]]]` which holds every layer's forward propagated variables, as well as the recently updated type `BackProp`.

```
coalgLayer :: (Fix Layer, [[[ForwardProp]]], BackProp) → Layer (Fix Layer, [[[ForwardProp]]], BackProp)
```

When implementing the pattern match for `InputLayer`, this will simply return the input layer itself.

```
coalgLayer :: (Fix Layer, [[[ForwardProp]]], BackProp) → Layer (Fix Layer, [[[ForwardProp]]], BackProp)
coalgLayer (Fx InputLayer, fp, bp)
    = InputLayer
```

When implementing the pattern match for `Layer`, this is essentially copy and pasting the second half of the function `runLayer` defined for non-deep LSTM networks. We can break down the implementation down into two steps:

1. **Execute the cell-level hylomorphism which performs back propagation along all of the layer's cells.** This returns us the updated cells as well as a function, which if applied to some initial empty `Deltas` value we create, will output the `Deltas` value computed by the entire layer.
2. **Update the hyperparameters of the layer using the recently computed delta values.** We then initialise and return a `BackProp` value which contains the list of δx_t values of the current layer, so that the previous layer can access them.

These steps can be seen labelled in the comments of the function `coalgLayer` below.

```
coalgLayer (Fx (Layer params cells innerLayer), fps, backProp)
    = let -- Step 1
          (cell, deltaFunc)   = hylo algCell2 coalgCell (cells, head fps, backProp)
          weight_w            = params^._1
          (hDim, dDim)        = (length $ weight_w ! 1, length $ head $ weight_w ! 1)
          deltaTotal          = deltaFunc (initDelta hDim dDim)

          -- Step 2
          layerBackProp       = initBackProp hDim dDim (Just $ deltaXs deltaTotal) (Just $ w)
      in (updateParameters (Layer params cell (innerLayer, tail fps, backProp')) deltaTotal)
```

This completes the back propagation for deep LSTM networks.

6.4.4 Training A Deep LSTM Network

We can now wrap the entire chapter up by connecting our layer algebra and coalgebra, creating the functions `run_sample` which runs a single sample, and `train` which runs multiple samples through a deep LSTM network.

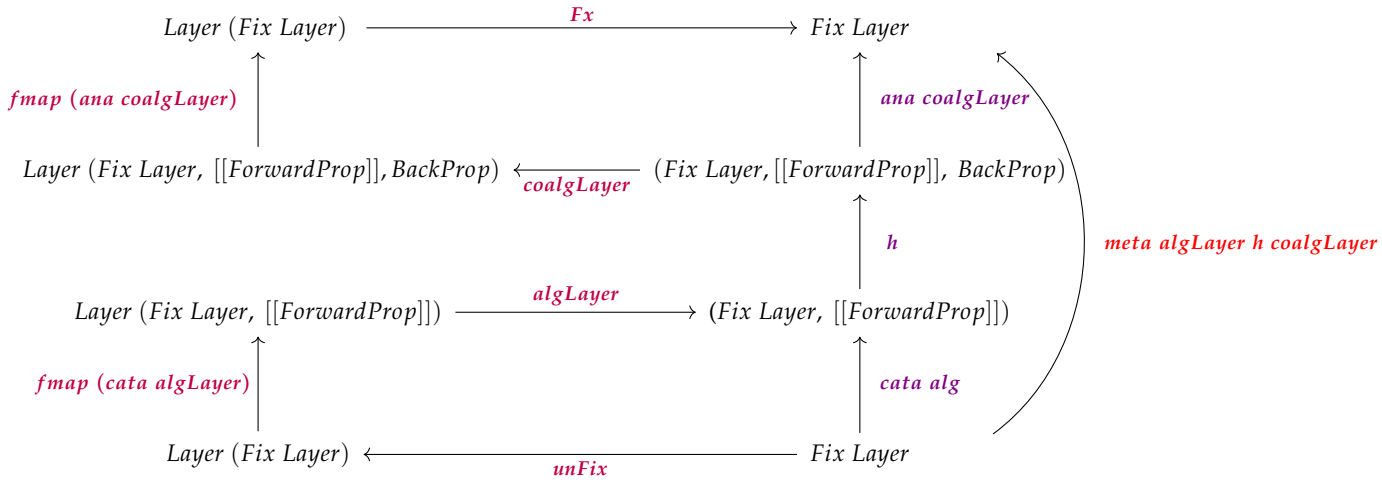
```
run_sample :: Fix Layer → Image → DesiredOutput → Fix Layer
run_sample neural_net sample desired_output
  = meta alg h coalg $ neural_net
    where h :: (Fix Layer, [ForwardProp] → [ForwardProp]) → (Fix Layer, [ForwardProp], BackProp)
          h = \(neural_net', forwardPass) →
                  (neural_net', (forwardPass [(ForwardProp sample [[[]]])]),
                      BackProp [[[]]] [[[[]]]] desired_output))

train :: Fix Layer → [Image] → [DesiredOutput] → Fix Layer
train neural_net samples desired_outputs
  = foldr (\(sample, desired_output) nn →
              run_sample nn sample desired_output) neural_net training_data
    where training_data = zip samples desired_outputs
```

These take almost exactly the same structure as the functions `run_sample` and `train` we have defined for our other two networks; `run_sample` performs the metamorphism of the network using a single sample, and `train` uses `foldr` to execute the process over a list of samples. As before, the function `h` inside `run_sample` changes

the carrier type between the catamorphism and anamorphism.

• Summary

We have now completed our final implementation of a neural network using recursion schemes. The entire system of the deep LSTM is modelled by a metamorphism, where the layer-level algebra calls a cell-level algebra, and the layer-level coalgebra calls a cell-level hylomorphism. This is a fascinating recursive pattern; although the introduction of the hylomorphism is new, this is only necessary due to how we need a catamorphism to perform the summation over the deltas in each cell. The entire forward and back propagation mechanism still remains fundamentally metamorphic. The layer-level metamorphism is illustrated in the diagram below.



Chapter 7

Evaluation

Now that we have explored the implementations of three different types of neural networks using recursion schemes, I will evaluate the success of Catana. First, we will see actual evidence of the implemented neural nets training on various types of data sets and examples of how to run the program. Afterwards, we go through what Catana means in terms of research. Finally I will then discuss the initial problems that this project has intended to engage with, and assess how well these have been addressed whilst comparing to existing alternatives.

7.1 Results

I will demonstrate my networks' ability to learn through training various models with data sets appropriate to their architecture, and displaying the resulting graphs of error over the number of samples trained. In addition, I will present the correlation coefficients, which are used to measure the strength of the relationship between two variables. What we should ideally see from a functioning neural network, is a negative correlation coefficient and a graph with a generally negative curvature, showing a decrease in error over run samples. This means that the accuracy in which the network classifies data improves as we provide more training data. This gradient should gradually become less steep as the network's hyperparameters converge to an optimal value. It should be noted that the architectures we use are very basic and use no modern training optimization tools, as well as the size of the data sets being significantly smaller than realistic examples of training data. This may affect the distinctiveness of learning behaviours.

7.1.1 Training A Fully Connected Neural Network

To demonstrate the behaviour of the fully connected networks previously implemented, I have chosen to use training data consisting of random decimal numbers and their corresponding desired outputs to be the sine function of those numbers. The constructed network can be seen below, where the input value is sent to three input nodes, and then propagated through 3 hidden layers to the final output layer consisting of a single node.

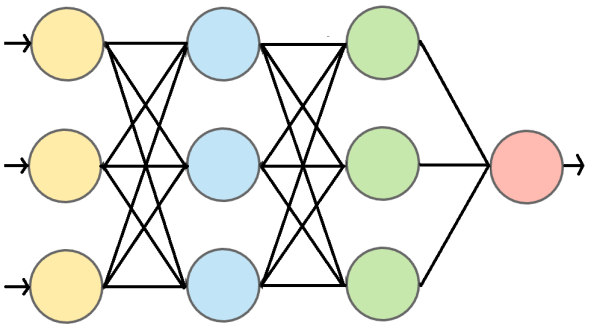


FIGURE 7.1: Fully Connected Network

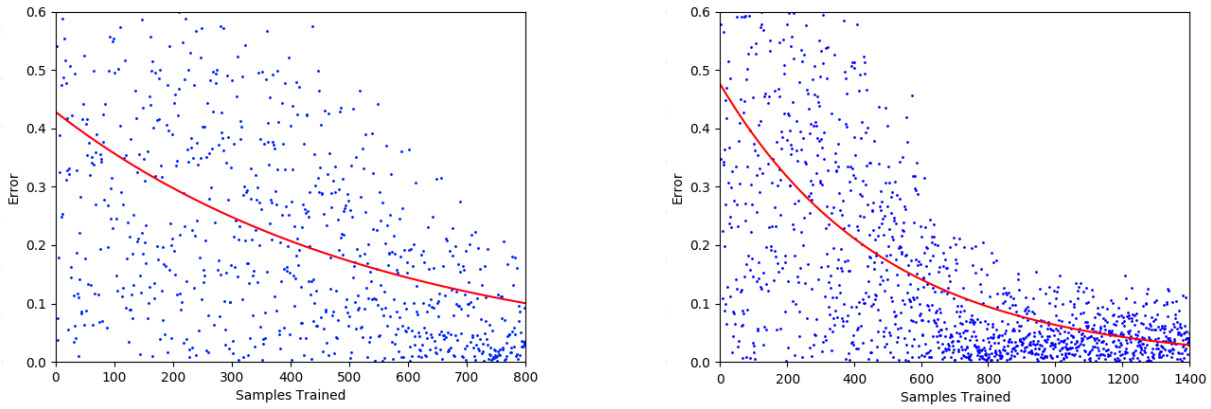
This network can be represented in Catana with the following code. We initialise the weights of each layer using the function `randMat2D` which creates a 2D matrix containing random doubles, and initialise the biases to an array of zeros. These are then used in the construction of the fully connected network `fc_network`.

```
exampleFCNetwork :: IO (Fix Layer)
exampleFCNetwork = do
    weights_a <- randMat2D 3 3
    weights_b <- randMat2D 3 3
    weights_c <- randMat2D 1 3
    let biases = replicate 3 0
        fc_network = (Fx (Layer weights_c biases
                         (Fx (Layer weights_b biases
                             (Fx (Layer weights_a biases
                                 (Fx InputLayer )))))))
    return fc_network
```

An example piece of code to run this, is given below. We read local files containing the input data and their desired outputs, and format them to the correct types so that they can be passed to the function `train` along with the example network above.

```
main = do
    inputFile  <- readFile "fullyconnected_results/sine_data_1400"
    labelsFile <- readFile "fullyconnected_results/sine_labels_1400"
    fc_network <- exampleFCNetwork
    let inputlines       = lines inputFile
        labelslines      = lines labelsFile
        input            = map read inputlines  :: [Double]
        labels           = map read labelslines :: [Double]
        samples          = (map (\x -> replicate 3 x) input)
        desired_outputs = (map (\x -> [x]) labels)
        nn = train fc_network samples desired_outputs
    print $ show nn
```

Here we see the change in cost as the amount of run samples increases, using sample sizes of 800 and 1400.



(A) Sample Size Of 800
Correlation Coefficient: -0.541

(B) Sample Size Of 1400
Correlation Coefficient: -0.649

FIGURE 7.2: Fully Connected Network: Error Over Sine Samples Trained

It can be observed that the error produced by samples gradually converges to give an attractive negative curvature, demonstrating the network's ability to progressively produce more accurate values. Additionally, the graphs show that using a larger sample size clearly improves the resulting accuracy of the network; this is also indicated by the higher magnitude in negative correlation coefficient.

7.1.2 Training A Convolutional Neural Network

Convolutional neural networks are used primarily to classify images. To demonstrate the behaviour of the convolutional neural networks, I have chosen to classify square matrices as being either an image displaying

the symbol 'X' or an image displaying the symbol 'O'.

A diagram of the neural network used can be seen below, where dimensions are given as *(width* x *height* x *depth)* x *amount*. An input image of dimensions (7x7x1) is provided to be passed through four hidden layers, and a final output is acquired through the fully connected layer. The output, being a vector of length two, has each value corresponding to the probability of the image being classified as either an 'X' or an 'O' symbol.

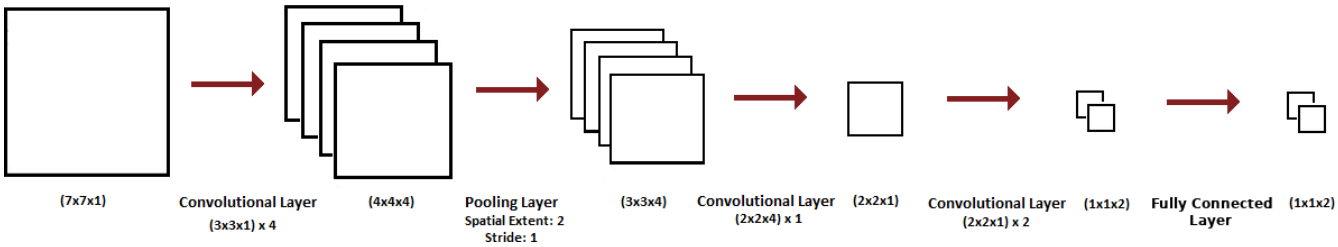


FIGURE 7.3: Convolutional Network

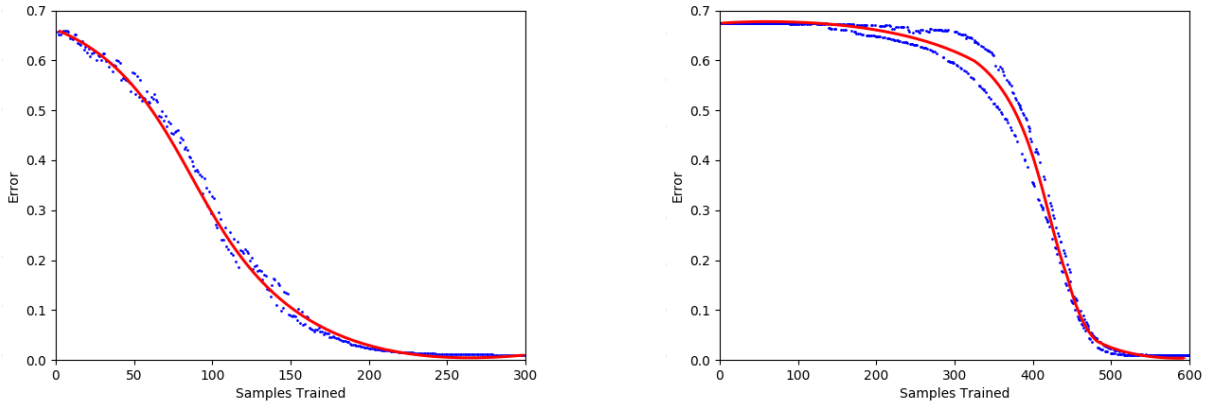
This network can be represented in Catana with the following code, where we generate random weights using `randMat4D` and provide matrices of zeros for the biases.

```
exampleConvNetwork :: IO (Fix Layer)
exampleConvNetwork = do
    weights_a <- randMat4D 3 3 4 1
    weights_b <- randMat4D 2 2 4 1
    weights_c <- randMat4D 2 2 1 2
    let nn = Fx (FullyConnectedLayer
                (Fx $ ConvolutionalLayer weights_c  [[0.0],[0.0]]
                    (Fx $ ConvolutionalLayer weights_b  [[0.0]]
                        (Fx $ PoolingLayer 1 2
                            (Fx $ ConvolutionalLayer weights_a  [[0.0]]
                                (Fx $ InputLayer )))))
    return nn
```

An example piece of code to run this, is given below. After reading in the local files containing the input data and desired outputs, we format the input data into matrices of the correct input image dimension, and format the desired outputs into the shape of the network's outputs. This allows us to then run `train` on the neural network.

```
main = do
     inputFile   <- readFile "conv_results/oz_data"
     labelsFile  <- readFile "conv_results/oz_labels"
     cnn         <- exampleConvNetwork
     let   inputlines       = lines  inputFile
           labelslines      = lines  labelsFile
           image_width      = 7
           samples          = map (\x -> [x]) $ map (chunksOf image_width)
                                    $ (map2 read (map (splitOn ",") inputlines )) ::    [[Double]]
           desired_outputs = map ((\x -> [[[x]]])  . read)   outputlines
           nn               = train  cnn samples desired_outputs
     print $ show nn
```

Here we see the change in error as the amount of trained samples increases, using sample sizes of 300 and 600.



(A) Sample Size Of 300
Correlation Coefficient: -0.747

(B) Sample Size Of 600
Correlation Coefficient: -0.876

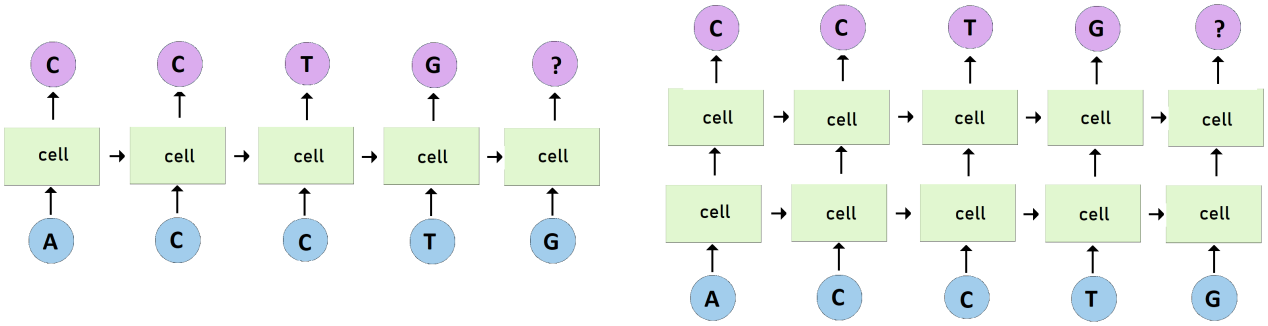
FIGURE 7.4: Convolutional Network: Error Over X/O Samples Trained

This shows a pleasing negative exponential curvature, demonstrating that our convolutional neural network is successful in learning to classify our provided images. Two distinctive streams of blue dots can also be observed in each graph, which represent the different paths of error induced by both of the sample types. The negative correlation coefficient is stronger in magnitude when using a sample size of 600, showing that the resulting accuracy of the network improves with a larger data set.

7.1.3 Training An LSTM And A Deep LSTM Network

LSTM networks are primarily used for classifying sequential time-series data in tasks such as text prediction, hand writing recognition or speech recognition. To demonstrate the functionality of our LSTM and deep LSTM network implementations, I have chosen to use DNA strands as data - these can be represented using the four characters 'A', 'T', 'C', 'G'. Given a strand of five DNA characters, our networks will attempt to learn the next character in the sequence.

Diagrams of the used LSTM and deep LSTM networks can be seen below, consisting of a single layer of five cells and two layers of five cells, respectively.



(A) LSTM

(B) Deep LSTM

FIGURE 7.5: LSTM & Deep LSTM Network

These networks can be represented in Catana with the following code. We initialise the weights with random values using `randMat3D` and set the biases to be an array of zeros. These are passed in as the hyperparameters to the network's layer(s), as well as the cells which have their state initialised to a zero array and deltas to an empty array.

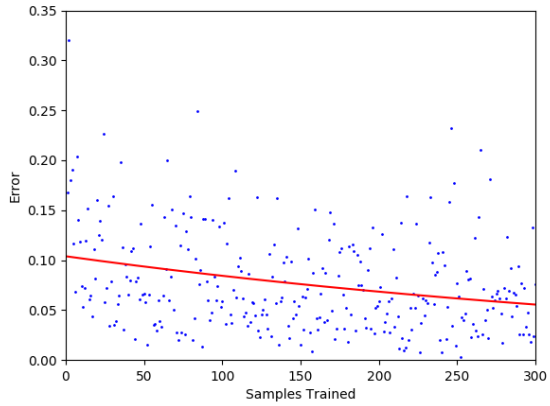
```
exampleLSTM = do
```

```
    weights_w <- randMat3D 2 1 4
    weights_u <- randMat3D 1 1 4
    let biases = replicate 4 [0.0]
    return (Fx (Layer (V.fromList weights_w, V.fromList weights_u, V.fromList biases)
                (Fx (EndCell [0.0] NoDeltas
                        (Fx (Cell [0] NoDeltas
                            (Fx (Cell [0] NoDeltas
                                (Fx (Cell [0] NoDeltas
                                    (Fx (Cell [0] NoDeltas
                                        (Fx InputCell ))))))))))
                (Fx InputLayer)))
exampleDeepLSTM = do
    weights_w <- randMat3D 1 1 4
    weights_u <- randMat3D 1 1 4
    input_layer <- lstm
    let biases = replicate 4 [0.0]
    return $ Fx (Layer (V.fromList weights_w, V.fromList weights_u, V.fromList biases)
                    (Fx (EndCell [0] NoDeltas
                        (Fx (Cell [0] NoDeltas
                            (Fx (Cell [0] NoDeltas
                                (Fx (Cell [0] NoDeltas
                                    (Fx (Cell [0] NoDeltas
                                        (Fx InputCell ))))))))))) (input_layer)
```

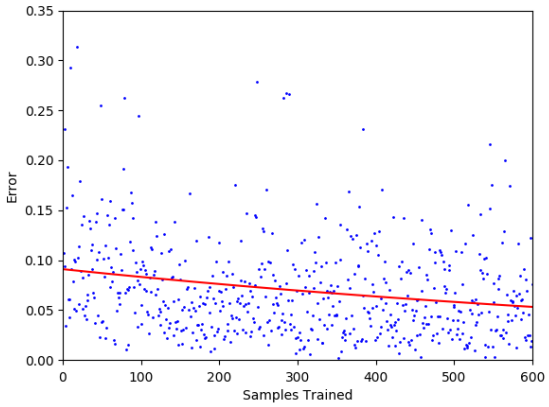
An example piece of code to run this is given below, where we read the DNA strands of six characters long from a local file. We map each of the characters of the strands to be represented by a specific value, and then take the first five values and last five values to give our input data and desired output respectively. This can then be used along with our example networks in the function `train`.

```
mapDNA :: [String] -> [[([Double], [Double])]]
mapDNA dna_strand =
     map f dna_strand
     where f :: String -> [([Double], [Double])]
           f dna = let dna_strand = map (\z -> case z of    'a' -> 0.2
                                                            'c' -> 0.4
                                                            'g' -> 0.6
                                                            't' -> 0.8) dna
                       input = init dna_strand
                       desired_output = tail dna_strand
                       res = map (mapT2 (\x -> [x])) $ zip input desired_output
                   in  res
main = do
    deeplstm <- exampleDeepLSTM
    inputFile <- readFile "rnn_results/dna_data_300"
    desoutputFile <- readFile "rnn_results/dna_labels_300"
    let dna_data = mapDNA inputFile
        dna = mapDNA linesOfFile deeplstm
        nn = train deeplstm dna
     print $ show nn
```

Below, we see the change in error as the amount of trained samples increases, using sample sizes of 300 and 600 for both LSTM and deep LSTM networks.

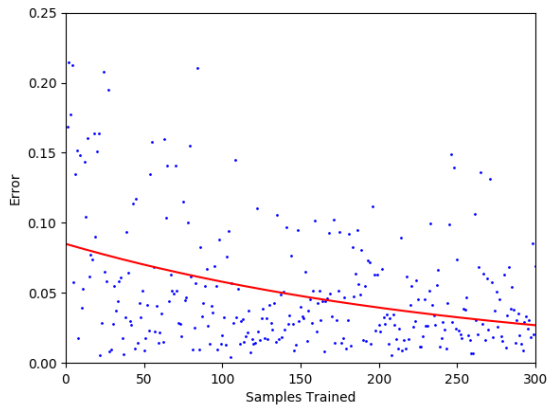


(A) Sample Size Of 300
Correlation Coefficient: -0.314

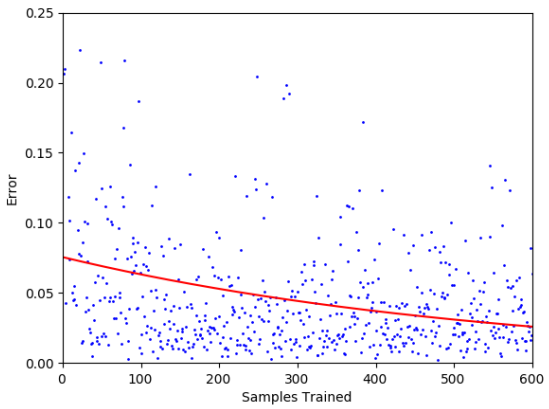


(B) Sample Size Of 600
Correlation Coefficient: -0.254

FIGURE 7.6: LSTM Network: Error Over DNA Samples Trained



(A) Sample Size Of 300
Correlation Coefficient: -0.335



(B) Sample Size Of 600
Correlation Coefficient: -0.317

FIGURE 7.7: Deep LSTM Network: Error Over DNA Samples Trained

The negative curvature shown in these graphs is less noticeable than the tests performed on previous networks, but still clearly present - this is fundamentally successful in demonstrating the ability of our networks to learn. The deep LSTM network also performs better than the non-deep LSTM; although increased complexity in network does not necessarily mean better performance, these networks are both very simple and I would have expected the deep LSTM to demonstrate a stronger ability to learn. As a final note, in contrast to the previous results, the correlation coefficient for large sample sizes are lower in magnitude. I hypothesis that this, in addition to the subtlety of the negative curvatures, are due to the high complexity of LSTMs - achieving more distinctive results would require an informed approach to architecture and usage of modern deep learning tools.

7.2 What Ideas Do Our Findings Signify?

We have demonstrated the underlying recursive nature exhibited by various neural networks, regardless of the construction and behaviour of the chosen models not immediately displaying obvious signs of recursive properties, unlike Tree Neural Networks for example.

Fully Connected Neural Networks In this implementation, we initially observed that the pattern of data propagation is a single-direction evaluation or construction. Hence by abstracting the neurons and connections of a network into a layer, training over the network as a series of layers becomes analogous to folding a list

using the input data as a base value, and unfolding a list using the output data as a seed value. In this, we learned that the system of a neural network can be perfectly modelled by a metamorphism.

Convolutional Neural Networks During this chapter, we further explored our previous findings. Convolutional networks exhibit a far more complex set of mechanisms and structures; however, just like fully connected networks, we observed that they could also be represented with metamorphisms. Both of these models being feed-forward networks emphasizes the idea that the pattern of data propagation in feed forward networks could allow all of them to exhibit this same recursion scheme.

LSTM Neural Networks When progressing to LSTMs, i.e. a single row of cells, we recognized that the pattern of recursion became slightly different. This stems from the property of an LSTM network having shared weights and biases between all cells. Back propagation entails 1) computing the deltas of each cell with respect to the hyperparameters, whilst also 2) accumulating and summing all of these values to provide a total delta. If we were to instead perform both of these actions during the anamorphism, the recursive behaviour of anamorphisms would mean that the end result would only be accessible in the deepest layer - the input layer. Regardless of either approach, another traversal through the entire structure would be necessary to acquire this value. Hence we observed that back propagation required a hylomorphism, whilst forward propagation could remain a catamorphism as usual.

Deep LSTM Neural Networks We then extended LSTMs by vertically stacking different layers of cells. The process of propagating through multiple rows, during which each row needing to perform propagation through all of its cells, introduces an extra layer of recursion on top of the previous implementation. Propagation in two directions has resulted in forward propagation being represented by a catamorphism in which its algebra calls a catamorphism, and back propagation being an anamorphism in which its coalgebra calls a hylomorphism. This development was as expected; aside from the occurrence of the hylomorphism, this adds further confidence in the notion that many other neural networks are also intrinsically metamorphic.

Formulating neural networks in terms of recursion schemes has enabled us to discover particular similarities in different network structures which tend to be seen as disparate problems. The knowledge that in general, forward propagation can be modelled as a catamorphism and back propagation as an anamorphism, is significant in realising the shared ability of networks to be modelled in a concise and compact paradigm.

7.3 Comparison To Alternatives: Achievements & Criticisms

In this section, I discuss certain strengths and shortcomings of Catana while comparing with Keras and Tensorflow, the most popular deep learning frameworks in the machine learning community. This has a primary focus on the original objectives that this project has set out to engage with.

7.3.1 Achievements

1) Conciseness And Elegance In The Definition Of Neural Networks

Let us compare our implementation of a fully connected layer with the Tensorflow implementation which consists of a function performs forward propagation. Specifically, the documentation states it is a function which "*creates a variable called weights, representing a fully connected weight matrix, which is multiplied by the inputs to produce a Tensor of hidden units*" [15].

An immediate observation of Listing 7.8b is the overwhelming number of parameters the function takes; whilst our implementation has not accounted for extensions such as initializers or regularizers, one would argue that these variables along with the majority of others are not true properties of the mathematical structure of a layer. The lack of modularity in Python forces the most convenient approach of architecting a layer to entail passing all utilised processes or flags as parameters. If this were to be designed in our Haskell implementation, it would be possible to isolate these processes as external functions and associating them to layers through either a new single data type acting as an indicator, or applying some sort of function composition on the layer data type. Additionally, the method of representing weights and biases by passing an initialiser

```
data Layer k  = Layer Weights Biases k
              | InputLayer
            deriving Functor
```

(A) Catana

```
def fully_connected(inputs, num_outputs, activation_fn=nn.relu,
                    normalizer_fn=None, normalizer_params=None,
                     weights_initializer = initializers . xavier_initializer (), weights_regularizer=None,
                     biases_initializer =init_ops. zeros_initializer (), biases_regularizer=None,
                    reuse=None, variables_collections=None, outputs_collections=None,
                    trainable=True, scope=None):
```

(B) Tensorflow [16]

FIGURE 7.8: Fully Connected Layer Implementation

to have Tensorflow generate weights and biases for you is very declarative. Although convenience is nice, it detracts from how layers are represented and also decreases the amount of fine-grained control the user has. In our implementation, Listing 7.8a, we have managed to condense the entire representation of the network down to a layer data type with two constructors. Each constructor contains only the absolutely necessary parameters which are true characteristic of layers; the hidden layer `Layer` which holds the weights and biases (as well as the next layer), and the input layer `InputLayer` which has no parameters at all. This is minimalistic and immediately straightforward to grasp. Additionally, as seen from the example constructions of neural networks, there is a means of conveniently generating weights and biases without compromising the representation of neural networks.

2) Representability In The Structure Of Neural Networks

Another remark regarding Listing 7.8b, is that Tensorflow chooses to represent a layer as a function which performs forward propagation, meaning back propagation is defined elsewhere as a separate function. Whilst it is not incorrect to view a layer as a forward propagation function, it is a personal disagreement that layers should solely be perceived this way. I believe that a neural network exists first as a data structure for different machine learning algorithms to operate on, before existing as an algorithm itself. Hence, one should be able to construct a neural network and have it exist as a lone data structure which can later be propagated over with provided input data, rather than a function which immediately requires inputs and returns a output.

A legitimate argument to this is that, a neural network can be viewed as simple mathematical models defining a function $f(x)$ which is a composition of other functions $g_i(x)$ that can further be decomposed. However in my implementation, this same composition of functions can be created by applying a catamorphism over the network, and the neural network forward propagation function can then be found in the second field of the tuple returned. This approach achieves essentially the same idea whilst also enabling a neural network to be constructed as data type.

```
forward_propagate :: Fix Layer → (Fix Layer, [Input] → [Input])
forward_propagate neural_net = cata alg neural_net
```

3) Representability In The Training Of Neural Networks

When an individual learns about how training a network occurs, the first concept understood is that forward propagation is applied to produce a value, and back propagation is applied to update the network using this value. In the declarative paradigm of Python, attempting the understand the background mechanism is extremely painful. Having the entire program's execution hidden behind opaque processes such as `sess.run()` encourages blind trust and discourages attempts to understand the underlying algorithms.

```
def run(self, fetches, feed_dict=None, options=None, run_metadata=None):
    raise NotImplementedError('run')
```

In our implementation, training is captured entirely in the function `run_sample` which has the exact same structure for all network implementations. Ignoring the intermediary function `h`, we always observe the composition of forward propagation followed by back propagation. Extending this to training multiple samples holds essentially the same idea.

```
run_sample :: Fix Layer → Input → Output → Fix Layer
run_sample neural_net sample desired_output
  = meta alg h coalg $ neural_net
    where h :: (Fix Layer, ?) → (Fix Layer, ?)
          h = ...
```

Likewise, by using data types which hold only essential fields, the user knows exactly what variables are involved in the actual forward and back propagation algorithms.

```
data ForwardProp      = ForwardProp {
                            input :: Input
                        } deriving Show
data BackProp         = BackProp {
                            outerWeights  :: Weights,
                            outerDeltas   :: Deltas,
                            desiredOutput :: Output,
                            inputStack    :: [Input]
                        }
```

4) Maintainability And Transparency In The Mechanism Of Neural Network Training

We use an arbitrary example to compare against, the Tensorflow function `LaunchConv2DBackpropInputOp` [17], written in C++. This function oddly returns a `struct` which is essentially an object. This is a primary reason for the size of programs found in Tensorflow source code; object oriented code tends to lead to less reusable code due to encapsulation and numerous initialised variable and functions needing to be accessed via objects. This motivates similar functions to be replicated but tailored to their own class. This creates further complications when attempting to follow the program's flow - various object types are continually created and have their own functions called, causing the call tree to be massive in terms of width. The amount of inheritance used in a large scale framework does not help this issue either, adding further obscurity to the core behaviour of the functions intent. Another downside is the abundance of sequenced code consisting of very primitive instructions; for the developer, when writing the program it can feel more straightforward and immediately logical to write, however this can make the task of grasping an implementation laborious. Imperative languages being based upon the notion of sequencing means that the only way to increase the level of abstraction is to introduce more keywords or standard functions, thus cluttering up the language and program. Aside from the use of inheritance, it is difficult to distinguish shared concepts between similar processes and neural network types.

In Catana, through being able to modularise the sub-processes of data propagation into non-recursive functions, each function successfully presents its meaningful behaviour without requiring a recursive mindset to comprehend. This is possible primarily due to the power of function composition, which is the act of pipelining the result of one function to the input of another, to create an entirely new function. The ability to easily compose functions encourages factoring functions for maintainability and code reuse, also making it realistically scalable towards a large scale framework. Hence, a single function exists for both the mechanisms of forward propagation and back propagation - although these decompose into calling external utility functions, this does not detract from the function's main intent. In combination with having abstracted away recursion from the design, the entire flow becomes immediately transparent in the program and the lines of code needed for a neural network implementation becomes minimal. Training a network over multiple samples boils down to folding the same recursion scheme over a data set.

7.3.2 Criticisms

1) End-developer Usability

There is a lot to be desired in the current state of the implementation, most of which is due to it being intended as a research project rather than a development project. With regards to handling input data and pipelining it into the training process, there is no general function defined for this due to there being no specified data format to adhere to. Hence it requires a very manual and specific approach from the user to ensure the data is arranged appropriately and the correct types are met. This inconvenience diminishes its viability to be used in a realistic, large-scale deep learning task.

The act of constructing a neural network from a series of layers is additionally quite unattractive. As we have seen from previous examples, our current method of architecting a network is by creating a large nesting of layers. This, along with the size of the weights and biases we have to define, causes the resulting network to be rather unintelligible and awkward to write. Keras, in contrast, does this job extremely well; the user can simply sequentially define layers to append to the network.

```
model.add(Conv2D(64, kernel_size=3, activation='relu', input_shape=(28,28,1)))
model.add(Conv2D(32, kernel_size=3, activation='relu'))
model.add(Flatten())
model.add(Dense(10, activation='softmax'))
```

2) Reliability

Although working in Haskell automatically provides us immediate benefits of strong typing and no side-effects, these properties could have been taken further advantage of. There is no assurance that the dimensions of the weights and biases of adjacently placed layers are compatible, hence it is possible to create an invalid neural network. Additionally, there is a lot of unsafe access and lack of exception handling throughout the program; the heavy amount of list management makes the program hypothetically vulnerable to run time errors, or worse, unknown incorrect computations.

3) Implementation Quality

Due to having implemented all algorithms virtually from scratch, the definitions for the various computations can sometimes be slightly disorganized or monolithic. In these cases, it is not immediately obvious what a computation performs which reduces the coherency of the program. To resolve this, a certain number of line extracts from functions would have to be rewritten and modularised further by recognising and isolating shared computations into a mathematical utility library.

4) Relationship Between Different Neural Networks

All three of our networks share a very similar structure with respect to the representation of neural networks and training; however these exist in very isolated environments from each other and there is no means of combining different layers due to them all being defined for their own specific network. For example, a convolutional neural network uses a fully connected layer and should logically be able to reuse the layer implementation from the fully connected network, rather than having to redefine it.

In Tensorflow and Keras, inheritance establishes relationships between different layer types easily. The base class `Layer` [18] acts as the root binding together all different variants of layers, and hence creates a shared environment where variants of layers can be combined.

• Summary

This section has demonstrated the success of Catana in its ability to learn from data. Furthermore, it has discussed the underlying significance that its approach brings and gives a critical analysis using comparisons to other existing technologies.

Chapter 8

Conclusion

We conclude this paper by first assessing how well the technical aims set specified in 1.4 have been achieved, followed by an exploration into potential further areas for work in Catana. Lastly, we will discuss the original objectives specified in 1.2 and the quality of which this project has addressed them.

8.1 Success Of Project

We can compare Catana with its original technical aims:

Model the training of three neural networks using recursion schemes This has been successful; I have managed to use recursion schemes as the core foundation of which all three types of neural networks are implemented around.

Illustrate the relationship between neural networks and recursion schemes I feel this has been achieved through repetitively demonstrating how the common process of forward and back propagation can be represented as a folding and unfolding, regardless of the network architecture. It is acknowledged that this does not strictly imply that all other network variations share this property, but hopefully acts as inspiration as to what could be further discovered.

Present proof of their ability to learn correctly This has been successful - there is a consistent negative curvature in all training tests demonstrating the expected relationship between error over the number of samples run. If we were to employ modern deep learning techniques such as optimisers, regularizers, or gradient descent algorithms, it is expected that this behaviour would become more distinct.

Evaluate Catana in terms of significance and implementation quality In this chapter, I have discussed the achievements and shortcomings of Catana with a primary focus on the original goals of the project. During this, I compare different aspects of my approach with the current most dominating software tools in deep learning - Keras and Tensorflow.

8.2 Future Work

Network Construction When addressing the unattractive style of constructing neural networks, free monads act as a perfect solution to this problem. Implementing a free monadic API would provide the user a way of elegantly declaring layers without nesting them, similar to how Keras is used. Free monads are just a general way of turning functors into monads. This would help shift the implementation closer to the ideals of the functional programming paradigm and improve its viability as a deep learning tool.

Correctness Dependent types offer a means of improving type-safety when confirming the compatibility of adjacently placed layers with respect to the dimensions of the weights and biases. This issue is very similar to verifying that the dimensions of matrices in a matrix multiplication chain are valid. This could likely be solved by incorporating fixed length vectors and natural numbers. In general, further refactoring of the code

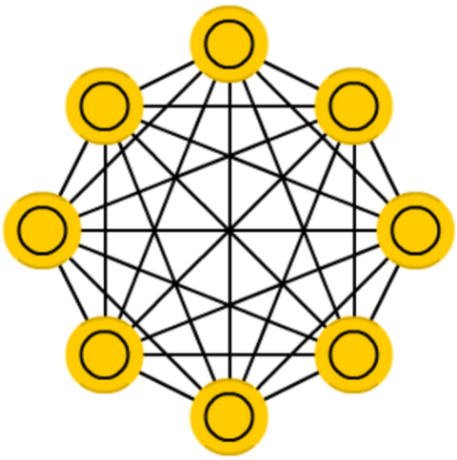
to increase safety and maintainability as a system is necessary - such as incorporating the Maybe type, and introducing a systematic method of accessing variables in lists and tuples.

Expressing Relationships It is fundamental to establish a sense of relation between various types of layers. This could be by further modularising the data structure of layers such that they are no longer considered inextricably linked to a specific network, or use type-classes to provide a common feature set over layers. An attractive option is to adopt data-families/type-families which allow abstract data types to be parameterized by their content type such that the data structure implementing the abstract type varies in a desired way; for example, this would potentially let us define different behaviours of a fully connected layer depending on the neural network we specify in the content type.

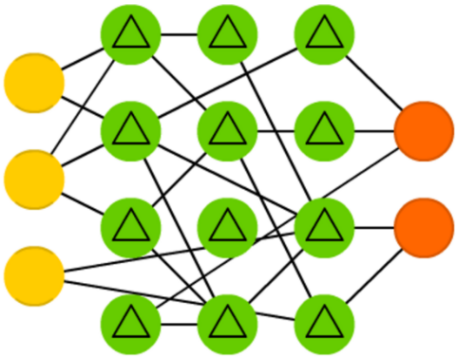
Optimizations Catana presents great potential for fold fusion to be incorporated. Training over a realistic data set means that thousands of intermediate data structures are being constructed from the compositions of various recursion schemes that could possibly be eliminated. Investigation into whether certain neural network implementations can satisfy the fusion laws can lead to significant optimizations.

Alternative Recursion Schemes Catana uses solely catamorphisms and anamorphisms, however other unexplored recursion schemes also exist as variants of folding and unfolding. Paramorphisms and apomorphisms would actually be the first obvious choice due to their ability to gain information about the original structure being traversed over - this is useful during forward propagation. Similarly, histomorphisms and futumorphisms are also potential alternatives, with the properties of being able to access particular computations of the recursion either performed in the past or in the future.

Further Neural Networks It is important to investigate more complex neural networks to recognize a spectrum of recursive patterns. This could hopefully uncover a system which helps us corresponds recursion scheme compositions with neural network architectures. Examples of very interesting choices are Hopfield Networks and Liquid State Machines (LSMs). Hopfield networks are composed of only one layer of nodes, where each node is connected to all other nodes. LSMs consist of a collection of randomly connected nodes where circuits are not hard coded to perform a specific task.



(A) Hopfield Neural Network



(B) Liquid State Machine

8.3 Summary

The core aims of this project were to demonstrate via implementation the underlying recursive properties of neural networks that allows us to represent their learning process using recursion schemes. Although Catana remains experimental, I have been successful to a significant degree in achieving the research goals of the project. This has demonstrated that using a system of catamorphisms and anamorphisms, it is possible to

construct and train fully-connected networks, convolutional networks, and deep LSTM networks, which are proven to be fully capable of learning from data. This has also illustrated a common theme around how neural networks in general can be expressed with recursion schemes. Finally, I have discussed the potential benefits to be reaped from this approach relative to existing technology, and also the areas of research that remain before a Catana-like approach can be considered suitable for production.

- **Representability** is achieved in this project in the sense that the implementation attempts to strictly adhere to how neural networks and data propagation are mathematically depicted, and takes a very caution approach to design in order to not over-saturate definitions.
- **Transparency** is achieved from a high level perspective - this is primarily assisted by the strengths which function composition and recursion schemes bring to the table. A separation of concerns between recursion and the actual meaningful behaviour means Catana can be consistently decomposed into three distinctive, core concepts: layers as a functor, forward propagation as an algebra, and back propagation as a coalgebra. However, the lower level mechanisms can be much further refined to make computations less ambiguous - this is expected due to the implementation being very self-contained and unreliant on external libraries.
- **Reliability** is a strength which naturally comes programming in a pure, functional programming language. Relative to mainstream deep learning tools written in imperative or object oriented languages, Catana can be considered in some ways more reliable due to an enforced strong type system and absence of side-effects. A primary drawback is that Catana does not exploit enough of the many approaches to safe design which exist in Haskell. Incorporating this is simply a matter of putting in further man-hours time.

Despite Catana not being immediately viable as an industry adopted deep learning framework, it is an exciting step into a promising paradigm, achieving strengths in new areas which other frameworks have failed to engage with. A number of concerns remain to be addressed before this approach can be put forward into real world usage. These are primarily: the exploration of further neural networks and recursion schemes; attention to practicality as a framework for end-users; and incorporation of further advanced functional programming concepts.

At a more abstract level, I postulated that the general data structure and shared mechanism of forward and back propagation amongst neural networks could promote a means of modelling any network as a form of recursion scheme. Having demonstrated this for three variants of neural networks, I hope that this can inspire further endeavors into what potential Catana can bring to the real world.

Bibliography

[1] Randy Beans. *The State of Machine Learning in Business Today*. September 2018. URL: https://www.forbes.com/sites/ciocentral/2018/09/17/the-state-of-machine-learning-in-business-today/#23155cbf3b1d.

[2] Dániel Berényi et al. *Applications Of Structured Recursion Schemes*. November 2017. URL: https://pdfs.semanticscholar.org/fd70/cec4a88f5292003578c5b7352c75bb0d9f0a.pdf.

[3] Jonathon Cai et al. *Making Neural Programming Architectures Generalize Via Recursion*. 2017. URL: https://people.eecs.berkeley.edu/~dawnsong/papers/iclr_2017_recursion.pdf.

[4] Huw Campbell. *Grenade*. May 2019. URL: https://github.com/HuwCampbell/grenade.

[5] Alejandro Chinea and Department of Fundamental Physics. *Understanding The Principles Of Recursive Neural Networks*. November 2009. URL: https://arxiv.org/pdf/0911.3298.pdf.

[6] Daphne Cornelisse. *An intuitive guide to Convolutional Neural Networks*. April 2018. URL: https://medium.freecodecamp.org/an-intuitive-guide-to-convolutional-neural-networks-260c2de0a050.

[7] Gavin Weiguang Ding. *Draw Convnet*. July 2018. URL: https://github.com/gwding/draw_convnet.

[8] Jeff Hale. *Deep Learning Framework Power Scores 2018*. September 2016. URL: https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a.

[9] IBM Jean-Francois Puget. *The Most Popular Language For Machine Learning and Data Science Is ..* December 2016. URL: https://www.kdnuggets.com/2017/01/most-popular-language-machine-learning-data-science.html.

[10] Keras. May 2019. URL: https://keras.io/.

[11] Netscribes. *Global Machine Learning Market (2018-2023)*. January 2018. URL: https://www.marketresearchstore.com/report/global-machine-learning-market-2018-2023-369608.

[12] Christopher Olah. *Neural Networks, Types, and Functional Programming*. September 2015. URL: https://colah.github.io/posts/2015-09-NN-Types-FP.

[13] Tensorflow. May 2019. URL: https://www.tensorflow.org/.

[14] Tensorflow. *Graphs and Sessions*. May 2019. URL: https://www.tensorflow.org/guide/graphs.

[15] Tensorflow. *Tensorflow Core 1.13 Documentation - Fully Connected Layer*. November 2018. URL: https://www.tensorflow.org/api_docs/python/tf/contrib/layers/fully_connected.

[16] Tensorflow. *Tensorflow Source Code*. November 2018. URL: https://github.com/tensorflow/tensorflow/blob/r1.13/tensorflow/contrib/layers/python/layers/layers.py.

[17] Tensorflow. *Tensorflow Source Code*. November 2018. URL: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/kernels/conv_grad_input_ops.cc.

[18] Tensorflow. *Tensorflow Source Code*. November 2018. URL: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/layers/python/layers/layers.py.