# Folding Over Neural Networks

Anonymous Author(s)

## Abstract

Neural networks are a critical component of deep learning techniques, and are usually implemented using unstructured recursion. This paper outlines a deeply embedded DSL which demonstrates the ability to encode the training of neural networks using recursion schemes and free monads. This produces an implementation which promotes values such as elegance and modularity while expressing neural network structures in a succinct and clear style.

***Keywords*** neural networks, recursion schemes

## 1 Introduction

The need to express neural networks in a succinct and modular way has brought about a number of dedicated libraries and languages for the task. Python is overwhelmingly the most used language in deep learning [Hale 2018; Puget 2018], giving rise to tools such as Tensorflow [Abadi 2015], Keras [Chollet 2015] and PyTorch [Paszke et al. 2019]. In this paper we intend to use Haskell to promote values such as transparency, elegance, and modularity in neural network implementation by taking an uninvestigated functional perspective: the understanding of deep learning in the context of recursion schemes [Bird and Paterson 1999; Hinze et al. 2013] and free monads [Kiselyov and Ishii 2015].

Network training is widely implemented using directed graphs to represent computations [Scarselli et al. 2008]. Recursion is rarely found, but when needed, this is typically only for implementing recursive networks, done by embedding control flow into directed graphs [Jeong et al. 2018]. This paper argues the possibility that not only can recursion be used to represent training in general, but that there is still a lot of structure that could be abstracted.

The process of neural networks can be described by three fundamental components:

- The network structure, consisting of nodes and connections organized into layers.
- Forward propagation, where data is sent forwards through the layers of a neural network to be transformed into a form of probability or classification.
- Back propagation, where data is sent backwards through the layers of a neural network in order to train and update the network's parameters appropriately.

This paper builds upon the idea that the network structure can be described recursively. Furthermore, forward and back propagation can be viewed analogously to folding and unfolding, and thus training a network could possibly be represented by the composition of a fold and unfold. This can all be better implemented through using recursion schemes.

*Even if this is possible, what would it achieve?*

This offers a narrative where neural networks are understood as recursive algebraic data structures and network training as patterns of recursion over these structures. The idea that we can concisely describe forward and back propagation as a fold and unfold gives a clear high-level intuition of how these processes work.

Further, recursion is inherent in a neural network's structure, hence training is a recursive function which traverses their structure. Folds/unfolds allow us to factor out recursion from these semantics of training [Hutton 1999] so that we can focus entirely on the meaningful behaviour of forward/back propagation over an isolated component of a network.

Neural networks are mathematical models and encapsulate the notions of composition, higher-order functions, types, and recursion [Olah 2015] — the functional approach we take with recursion schemes successfully expresses all of these ideas accurately and minimally. This is in contrast to using abstractions within procedural/object-oriented paradigms which are incongruous when describing neural networks.

*Why use recursion schemes instead of folds and unfolds?*

This style of thinking allows us to write a much simpler free monadic DSL interface. Free monads give us nicer DSLs that are closer to what we're abstracting to — other options for embedding DSLs don't, as there is often junk in the representation [Swierstra 2008].

Also, using folds/unfolds requires our data type for neural networks to be defined with explicit recursion — this makes them harder to reason about. Instead, recursion schemes let us generalize folds/unfolds over arbitrary recursive data structures in the form of non-recursive functors [Hinze et al. 2013]. This lets us decouple how layers are connected from what a layer actually is.

By taking this approach, this paper aims towards an ideal like the following. A representation for neural networks is given by *Fix Layer*, which creates a tree whose nodes are shaped by the data type *Layer*. Forward and back propagation are then simply typed functions defined using fold and unfold.

**type** *Network* = *Fix Layer*

*forwardProp* :: *Network* → *Output*
*forwardProp* = *fold* ...

*backProp* :: *Output* → *Network*
*backProp* = *unfold* ...

Although it is not viable to exhaust every type of neural network, this paper intentionally engages with an extremely popular network architecture: fully connected networks (Section 3). These ideas have been further applied towards two

other very popular network types in the appendix: convolutional (Section A) and recurrent (Section B) neural networks. Through this, we aim to provide insight into how neural network architectures in general can be visualized as recursion scheme systems. This also aspires to pose a question as to the extent of which the ideas discussed have potential to be translated to more elaborate networks.

After giving a brief background on recursion schemes and neural networks (Section 2), this paper makes the following contributions:

- We demonstrate how it is possible to use composition of a fold and unfold, called a metamorphism, to encode the training of a fully connected network (Section 3).
- We improve on our implementation (from Section 3) by showing a way of constructing neural networks in a more elegant and modular manner using free monads and coproducts. During this, we show how metamorphisms can be tailored to incorporate these concepts (Section 4).
- We show how neural network training can be implemented as a single catamorphism (fold) rather than as a metamorphism (Section 5).

Finally, related work is dicussed (Section 6) and the paper concludes (Section 7).

What this paper is not concerned with, is the efficiency gained (or lost!) by this encoding or its potential to be used industrially. Nor does it unveil a significant new algebraic structure. Rather, we aim to illustrate how neural networks can be implemented functionally, and provide insight as to how concepts surrounding deep learning relate very naturally those in a functional design paradigm, thus offering an interesting angle to how neural networks can be understood.

## 2 Background

Before being able to flesh out our motivation concretely in the main text of the paper, we introduce the standard material for defining recursion schemes and also give a general overview of how neural networks work.

### 2.1 Recursion Schemes

Recursion schemes are composable combinators that generalize over folds or unfolds by automating recursive procedures over arbitrary nested data structures [Bird and Paterson 1999; Hinze et al. 2013; Meijer et al. 1991]. The main recursion schemes of interest are 'catamorphisms' (folds) and 'anamorphisms' (unfolds) which will be introduced later. A recursion scheme system consists of three components:

- A non-recursive functor $f$.
- The fixpoint type *Fix*.
- An *algebra* (when using a catamorphism) or *coalgebra* (when using an anamorphism).

***Non-recursive Functors*** Because recursion schemes operate over arbitrary algebraic data structures, we need to use non-recursive functors which are data types where their recursion has been factored away.

For example, the standard data type for lists:

$$\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)$$

can be converted into the non-recursive definition:

$$\textbf{data}\ List\ a\ k = Nil \mid Cons\ a\ k$$

if the new type parameter $k$ represents the previous recursive occurrence of $List\ a$.

***Fix*** When working with non-recursive type constructors, we need to introduce a generic recursive type to be able to crank out a recursive construction. This is given by type *Fix*.

$$\textbf{newtype}\ Fix\ f = In\ (f\ (Fix\ f))$$
$$out :: Functor\ f \Rightarrow Fix\ f \to f\ a$$
$$out\ (In\ f) = f$$

*Fix* takes the fixed point of a given functor $f$ to create a tree whose nodes are shaped by $f$. The function *out* then deconstructs this tree to attain a value of $f\ a$.

***Catamorphisms*** A *catamorphism* is a recursion scheme which generalizes over folds of lists to arbitrary algebraic data types, given below by the function *cata*. This takes as arguments a function of type $f\ a \to a$, called an *algebra*, and a data structure of type $Fix\ f$. It then uses this algebra to recursively evaluate the data structure to a value of type $a$.

$$cata :: Functor\ f \Rightarrow (f\ a \to a) \to Fix\ f \to a$$
$$cata\ alg = alg \circ fmap\ (cata\ alg) \circ out$$

*Algebras* go hand-in-hand with catamorphisms. We refer to type $a$ as the algebra's *carrier* type; algebras hence evaluate data structures of type $f\ a$ to a value of carrier type $a$.

***Anamorphisms*** An *anamorphism* is a recursion scheme which generalizes over unfolds of lists to arbitrary algebraic data types, given below by the function *ana*. This takes as arguments a function of type $b \to f\ b$, called a *coalgebra*, and a seed value of type $b$. It then uses this coalgebra on the seed value to recursively generate a data structure of the type $Fix\ f$.

$$ana :: Functor\ f \Rightarrow (b \to f\ b) \to b \to Fix\ f$$
$$ana\ coalg = In \circ fmap\ (ana\ coalg) \circ coalg$$

*Coalgebras* go hand-in-hand with anamorphisms. We refer to type $b$ as the coalgebra's *carrier* type; coalgebras hence use a seed value of carrier type $b$ to construct data structures of type $f\ b$.

***Metamorphisms*** A *metamorphism* is the composition of an anamorphism, an intermediary function $h$, and a catamorphism. It can be viewed simply as the composition of an unfold and fold. We write this as the function *meta* which takes as arguments: a coalgebra, intermediary function $h$,

and algebra. This can then be applied to a value of type $Fix\ f$ and return a new value of type $Fix\ f$.

$$meta :: Functor\ f \Rightarrow (b \to f\ b) \to (a \to b) \to (f\ a \to a) \to Fix\ f \to Fix\ f$$
$$meta\ coalg\ h\ alg = ana\ coalg \circ h \circ cata\ alg$$

The intermediary function $h$ of type $a \to b$ is used to change the value of the catamorphism's carrier type $a$ to a value of the anamorphism's carrier type $b$. This allows one to compose an anamorphism and catamorphism whilst being able to use different carriers for each.

### 2.2 Neural Networks

Neural networks [Anderson 1995; McCulloch and Pitts 1943] propagate input data through a series of layers of mathematical processing; this results in an outputting meaningful information which can be used to recognise underlying relationships in data. Their structure is a set of neurons and connections organized in layers, represented by a graph of nodes and edges as seen in Figure 1. A network's components can also contain learnable parameters — these are variables which determine the accuracy of the network's output, so training a network means optimizing these parameters.

The act of learning any neural network consists of two stages: *forward propagation* and *back propagation*, in which data is sent and processed in the corresponding direction of the given propagation.

***Forward propagation*** aims to apply a series of transformations to some provided input data as it passes through each layer of the network, resulting in an output which can be interpreted as a classification or probability etc. This process works from the input layer towards the output layer, and can be broken down into the following general stages:

1. The input layer receives some input data and sends it forward unmodified to the next layer.
2. Any following layer receives its input from its previous layer, and applies some (layer-specific) transformation to it. If the layer is an intermediate layer, the output is passed to the next layer; otherwise the layer is an output layer which will yield the final output of the entire network.

Note how the act of evaluating a neural network as a data structure (given an input) to acquire an output is what a fold can essentially achieve. If we can fold over a neural network to produce its output vector, then this can be modelled as a *catamorphism*. Given that a network's layers are stacked side-by-side and processed consecutively, a useful analogy is to compare it to a list. Folding over a neural network's layers is essentially the same structural process as folding over a list where its elements are layers.

***Back propagation*** aims to use the actual final output and the desired output to update each layer's parameters such that the next forward propagation will achieve results closer to the desired output. This process works from the output layer towards the input layer, and can be broken down into the following general stages:

1. The output layer uses its actual output and the desired output to update its parameters and compute a set of back propagation variables to be sent to the previous layer.
2. An intermediate layer will receive a set of back propagation variables from the next layer. It uses these to appropriately update its parameters and compute a new set of back propagated variables to be sent to the previous layer.
3. The input layer undergoes no updates as it contains no parameters.

Note how the act of using the output as a seed value to construct an updated neural network as a data structure is what an unfold can essentially achieve. If we can perform an unfold with an output to produce a new neural network, then this can be modelled as an *anamorphism*.

***Summary*** We have covered a basic understanding of recursion schemes and neural networks; during this, observations were briefly made with respect to the relationship between recursion schemes and neural network training. In the rest of this paper, we develop these ideas more concretely through implementation using fully connected networks.

## 3 Fully Connected Networks

In this section, we intend to model fully connected networks as a recursion scheme system. We first give a general background on the structure of fully connected networks. Afterwards, we find a data type which can be used to represent fully connected networks. Our ultimate goal is to implement forward propagation as a catamorphism (fold), and then back propagation as an anamorphism (unfold) — these components can then be combined together, resulting in an encoding of network training as a metamorphism (composition of a fold and unfold).

### 3.1 Background: Fully Connected Networks

The simplest type of neural networks are *fully connected networks*, first introduced by Ivakhnenko and Lapa [1967]; Schmidhuber [2015]. These come under the family of feed-forward networks, which means data propagates along one direction with no loops. The layers of a fully connected network are stacked side-by-side, with any given node only having connections to all nodes in the previous layer and next layer, as seen in Figure 1. Each connection has a learnable parameter referred to as a *weight*, and each node has a learnable parameter referred to as a *bias*.

Aside from the input layer, all layers in a fully connected network apply the same transformation during forward propagation. We can therefore say that there exist two types of layers:

- **Input Layer** This is the first layer in the network. It takes in some initial input data, and simply passes this forward to the next layer.
- **Dense Layer** This is any layer after the input layer. It takes an input from the previous layer, and then: multiplies this by its weights, adds on its biases, and applies a normalization function.

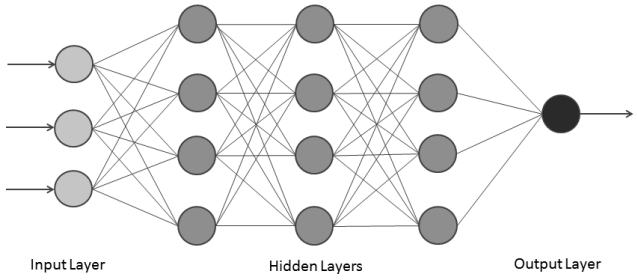


**Figure 1.** Fully Connected Neural Network

### 3.2 Finding A Data Type

The first component to address before we can model how such a network is learned, is defining a data type for a fully connected network which takes the form of a non-recursive functor. We intend this data type to be as representative of a fully connected network as possible; clearly this is something that will evolve over time as we encounter new requirements.

We will choose to use layers as our core data type to represent a neural network. "*But how can a single layer represent every neuron and connection it contains and their respective parameters?*". If we take a step back, we can abstract away from the idea of nodes and edges being their own data structure in the graph as shown in Figure 1, and instead capture them implicitly by using matrices. The weights of the connections between two layers can be represented as a 2D matrix, where an element of index $(i, j)$ represents the weight of the connection from the $i^{th}$ node on the LHS to the $j^{th}$ node on the RHS. A layer's biases can be represented as a vector, as each bias is associated with a node rather than a connection.

Hence, we will define type synonyms for our weights to be a 2D list of doubles, and for our biases and inputs/outputs to be a list of doubles. It should be noted that values can very often be both inputs and outputs — what is considered an output of one layer can be considered an input to another layer. Throughout this paper we use the type synonym *Input* to interchangeably mean either an input or output.

```
type Weights = [[Double]]
type Biases  = [Double]
type Input   = [Double]
```

Our core data type is *Layer* which takes a type parameter $k$. Its base case constructor is given by *InputLayer* which contains the network's initial input *Input* as a parameter. The constructor *DenseLayer* then contains parameters *Weights* and *Biases* as well as a value of type $k$ — this can be used to represent any layer after the input layer.

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

This is a pleasingly minimal and elegant encoding of the network structure. An instance of a network would then be represented by a recursive nesting of *Layer*s. However as *Layer* is a non-recursive type constructor, we need to introduce the type *Fix* to crank out a recursive construction; the type of a fully connected network would therefore be *Fix Layer*.

```
type FCNetwork = Fix Layer
```

Creating an example of such a network would then mean wrapping each occurence of a *Layer* inside a *Fix*, as demonstrated by *network*.

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

We now have a data type which allows us to experiment with how we can encode neural network training (forward and back propagation) as a recursion scheme system.

### 3.3 Forward Propagation as a Catamorphism

We aim to encode forward propagation to later be used as a catamorphism (fold) such that given an input, a neural network can be evaluated to an output. To do so, we only need to implement an algebra.

***Defining An Algebra*** This means defining a function satisfying the type of an algebra, $f\ a \to a$. This will represent the logic which occurs when forward propagating over a single layer. What needs to be decided is what the algebra's carrier type $a$ should be, however this is not so straightforward. As an initial idea, if each layer takes an input to produce an output, it makes sense to define our algebra as the function *forwardAlg* with the following type definition.

```
forwardAlg :: Layer Input → Input
```

Defining *forwardAlg* for *InputLayer* is trivial; the algebra should forward pass the input to the next layer, unmodified.

```
forwardAlg (InputLayer input) = input
```

Defining *forwardAlg* for *DenseLayer* is where the forward propagation computation comes in. First, let there exist a function *forward* which performs this computation.

```
forward :: Weights → Biases → Input → Input
forward wᴸ bᴸ aᴸ⁻¹ = aᴸ
  where aᴸ = σ(wᴸ ∗ aᴸ⁻¹ + bᴸ)
```

The output $a^L$ of layer $L$ is computed by multiplying its input $a^{L-1}$ with its weights $w^L$, adding on its biases $b^L$, and applying an normalization function $\sigma$.

We can now define the algebra for forward propagation on *DenseLayer*. This simply consists of calling *forward* on the layer's parameters to produce the layer's output.

$$forwardAlg\ (DenseLayer\ w\ b\ input) = forward\ w\ b\ input$$

By implementing forward propagation as an algebra, we achieve a concise description of how a layer takes input data and produces an output which classifies the input. The absence of recursion lets us focus on the meaningful behaviour of forward propagation on a single layer without worrying about the rest of the network.

If we were to use this with a catamorphism to perform forward propagation over an entire neural network, this would simply consist of calling *cata forwardAlg* on a value of type *Fix Layer*. Rather than illustrate this in detail however, we can skip this stage as we instead intend to use our algebra with a *metamorphism* later on.

***Improving Our Algebra*** What we have created so far would work perfectly fine if we were interested solely in using a neural network to classify data. However, it is insufficient if we then wanted to use these results to update the network afterwards i.e. *back propagation*. In order to initialise back propagation, we require certain information from the result of forward propagation that our implementation currently lacks. Therefore, we will next specify and engage with the two problems which hinder further progress.

*Firstly*, if we only evaluate our network to a single output, we no longer have information of how our network was structured. In order to update our network during back propagation, we need to know what the original network was.

We can solve this by changing the carrier type to additionally store 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 $(Fix\ Layer, Input)$. The algebra is then defined as below.

$$\begin{aligned}
&forwardAlg :: Layer\ (Fix\ Layer, Input) \rightarrow (Fix\ Layer, Input)\\
&forwardAlg\ (DenseLayer\ w\ b\ (prevLayer, input))\\
&\quad = (In\ (DenseLayer\ w\ b\ prevLayer), output)\\
&\quad \textbf{where}\ output = forward\ w\ b\ input\\
&forwardAlg\ (InputLayer\ input)\\
&\quad = (In\ (InputLayer\ input), input)
\end{aligned}$$

Here we refer to the carrier as $(prevLayer, input)$. Our algebra now successfully returns both the part of the network which has been evaluated so far, as well as the value evaluated from it.

*Secondly*, when back propagating over each layer of a network, we are required to know what the layer's input and output data were. However, our current algebra of type $Layer\ (Fix\ Layer, Input) \rightarrow (Fix\ Layer, Input)$ only lets us know the very final layer's output.

We can solve this by changing the carrier type so that we forward propagate a list of all outputs/inputs produced so far. This change is represented by the following algebra with the carrier type $(Fix\ Layer, [Input])$.

$$\begin{aligned}
&forwardAlg :: Layer\ (Fix\ Layer, [Input]) \rightarrow\\
&\qquad\qquad (Fix\ Layer, [Input])\\
&forwardAlg\ (DenseLayer\ w\ b\ (prevLayer, inputStack))\\
&\quad = (In\ (DenseLayer\ w\ b\ prevLayer), (output : inputStack))\\
&\quad \textbf{where}\ input\ \ = head\ inputStack\\
&\qquad\quad output\ = forward\ w\ b\ input\\
&forwardAlg\ (InputLayer\ input)\\
&\quad = (In\ (InputLayer\ input), [input])
\end{aligned}$$

At each level of recursion during the catamorphism, we append a new output to the list - this results in having a stack of all the outputs/inputs after evaluating the final layer. This algebra now sufficiently represents forward propagation over the layers in a fully connected network; using it with a catamorphism will hence represent forward propagation across an entire network.

### 3.4 Back Propagation as an Anamorphism

We next aim to implement back propagation as an anamorphism (unfold) which constructs an updated neural network using the output given to us by the catamorphism. To do so, we only need to implement a coalgebra.

***Defining A Coalgebra*** This means defining a function satisfying the type of a coalgebra, $b \rightarrow f\ b$. This will represent the logic which occurs when back propagating over a layer. The first step is choosing what the carrier type $b$ is, which can be viewed as the type of seed value we use to generate a data structure from.

We won't go into the specifics of how back propagation is performed; what is important is knowing what information is needed in order to carry it out. To be specific:

- We need to know the desired final output (associated with the initial input) to be able to initiate the process of back propagation.
- We need a way of back propagating the delta variable $\delta^L$ (to be computed by layer $L$) to the previous layer $L-1$. The variable $\delta^L$ is used as a term to refer to the derivative of layer $L$'s parameters with respect to its output error.
- We need a way of back propagating the original weights $w^L$ of the current layer $L$ to the previous layer $L-1$.

Given these requirements, we cannot directly use a value of type $(Fix\ Layer, [Input])$ from the catamorphism as our anamorphism's carrier, as this information is insufficient to perform back propagation. This is not a problem however — even though we wish to eventually compose an anamorphism with a catamorphism, the type of an anamorphism's

input $b$ does not necessarily have to be same as the type of the catamorphism's output $a$, due to the intermediary function $h :: a \to b$ (as discussed in Section 2.1). This conveniently provides us some flexibility.

We'll therefore define a friendly data type *BackProp*; this represents all the information necessary for back propagation, and thus stores both the list of inputs/outputs acquired from the catamorphism and any required back propagation variables.

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

Using *BackProp*, we can define a suitable carrier type for our coalgebra. We need this to hold two things: the *BackProp* data type and a reference to the previous layer. We will therefore let this be the tuple (*Fix Layer*, *BackProp*); this means we can define the coalgebra as the function *backwardCoalg* with the following type definition:

```
backwardCoalg :: (Fix Layer, BackProp) →
                 Layer (Fix Layer, BackProp)
```

Defining *backwardCoalg* in the case of *In InputLayer* should simply return *InputLayer*, as the input layer of the network is not processed during back propagation.

```
backwardCoalg (In InputLayer, backPropData) = InputLayer
```

Defining *backwardCoalg* in the case of *In* (*DenseLayer*...) can be broken down into into two main steps. At every layer we need to:

1. Compute the delta error $\delta^L$ and the updated weights $w^L_{new}$ and biases $b^L_{new}$.
2. Return the layer with its updated parameters and updated carrier.

Let there be a function *backward* which takes our layer's parameters and a *BackProp* value to compute and return a tuple containing the layer's delta error and its updated weights and biases:

```
backward :: Weights → Biases → BackProp →
            (Deltas, Weights, Biases)
```

We then use this to define a coalgebra for *In* (*DenseLayer* . .).

```
backwardCoalg ( In (DenseLayer w b prevLayer), backProp)
  = DenseLayer w' b' (In prevLayer, backProp')
  where
```

First, we call the function *backward* to compute the new weights and biases and the delta error. These are used to update the layer's weights and biases.

```
    (w', b', delta) = backward w b backProp
```

Additionally, we need to update the back propagation data to store: the newly computed delta, the layer's old weights, and last but not least, the tail of its original *inputStack* field (so that the first element will always be the output associated with the current layer being processed.).

```
    inputStack' = tail (inputStack backPropData)
    backProp'   = backPropData { inputStack    = inputStack'
                               , outerWeights  = w
                               , outerDeltas   = delta      }
```

We have now successfully implemented a coalgebra that represents back propagation in the layers of a fully connected network, allowing us to reconstruct an updated layer from a seed value. Using this coalgebra with an anamorphism will hence represent back propagation across an entire network. This can be done simply by calling *ana backwardCoalg* on a value of type (*Fix Layer*, *BackProp*). Rather than illustrate this in detail however, we can skip this stage and instead combine our coalgebra with our algebra to implement the whole training process as a *metamorphism*.

### 3.5 Encoding Training As A Metamorphism

So far, we have implemented forward and back propagation as a catamorphism and anamorphism respectively. Given that training a neural network consists of the composition of these two processes, we can use a *metamorphism* — this is the composition of a catamorphism, an intermediary function $h$, and an anamorphism. The intermediary function $h$ of type $a \to b$ is used to change the carrier type from the catamorphism's to the anamorphism's.

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

To implement training, we create a function *train* which acts as a wrapper around our metamorphism. This takes as arguments the neural network and desired output. Using these, it executes a metamorphism with our algebra *forwardAlg* and coalgebra *backwardCoalg* to train and update our network.

```
train :: Fix Layer → Input → Fix Layer
train neuralNet desiredOutput
  = (meta forwardAlg h backwardCoalg) neuralNet
  where
```

When defining the intermediary function $h$, this should take the output of the catamorphism, of carrier type (*Fix Layer*, [*Input*]), to the input of the anamorphism, of carrier type (*Fix Layer*, *BackProp*).

```
    h :: (Fix Layer, [Input]) → (Fix Layer, BackProp)
    h (nn, inputStack)
        = (nn, BackProp inputStack [[]] [] desiredOutput)
```

First, $h$ uses the *inputStack* produced by the catamorphism to initialise a *BackProp* value; this contains *inputStack*, the

provided desired output, and initialises the *outerWeights* and *outerDeltas* fields as empty lists. Using the provided neural network *nn*, it returns a tuple of type (*Fix Layer*, *BackProp*).

The metamorphism now successfully represents a function which when given a neural network (where the initial input is stored in the *InputLayer* constructor) and the desired output, it composes forward and back propagation to update the network's parameters according to the error between the actual output and the desired output. There's still some clumsiness in the design that needs to be refined though before we can be satisfied with the code.

### 3.6 Improving Our Design

Currently, the way the initial input and the desired output are treated is very illogical; it doesn't really make sense to pass the desired output as a parameter to *train*, yet have to embed the initial input inside the *InputLayer* constructor. A neural network should be able to exist independently of a provided input, i.e. it should wait to be given an input. It is hence more appropriate for a neural network to behave more as a function which takes an input to an output.

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

$$\textbf{data}\ \mathit{Layer}\ k = \mathit{DenseLayer}\ \mathit{Weights}\ \mathit{Biases}\ k \mid \mathit{InputLayer}$$

We'll then change the carrier type for the algebra so that it allows us to isolate the input data from the data structure. At the moment the carrier type we use during forward propagation is (*Fix Layer*, [*Input*]). If we change [*Input*] to behave as a function of type [*Input*] → [*Input*], this means we delay needing an initial input and the catamorphism can be run on a neural net without any input variables being provided.

These functions can be composed in the algebra. What we end up with from evaluating a neural network, is a single function representing the composition of each layer's forward propagation — this can take an initial input and return a list containing every layer's inputs/outputs. So let's redefine our algebra using the carrier (*Fix Layer*, [*Input*] → [*Input*]).

$$\begin{aligned}&\mathit{forwardAlg} :: \mathit{Layer}\ (\mathit{Fix}\ \mathit{Layer}, [\mathit{Input}] \to [\mathit{Input}]) \to \\ &\qquad (\mathit{Fix}\ \mathit{Layer}, [\mathit{Input}] \to [\mathit{Input}])\end{aligned}$$

We refer to functions of type [*Input*] → [*Input*] as *forwardPass*. When defining *forwardAlg* for *DenseLayer*, we compose the existing function *forwardPass* that was acquired from forward propagating over the previous layer, with a new function which describes how to use the previous layer's output and perform forward propagation on the current layer.

$$\begin{aligned}&\mathit{forwardAlg}\ (\mathit{DenseLayer}\ w\ b\ (\mathit{prevLayer}, \mathit{forwardPass})) \\ &\quad = \textbf{let}\ \mathit{newForwardPass} = (\lambda\ \mathit{inputStack} \to \\ &\qquad \textbf{let}\ \mathit{input} = \mathit{head}\ \mathit{inputStack} \\ &\qquad\quad \mathit{output} = \mathit{forward}\ w\ b\ \mathit{input} \\ &\qquad \textbf{in}\ (\mathit{output} : \mathit{inputStack})) \circ \mathit{forwardPass} \\ &\quad \textbf{in}\ (\mathit{In}\ (\mathit{DenseLayer}\ w\ b\ \mathit{prevLayer}), \mathit{newForwardPass})\end{aligned}$$

When defining *forwardAlg* for *InputLayer*, this should return the identity function so that we can pass it an input sample and expect it returned unmodified.

$$\mathit{forwardAlg}\ \mathit{InputLayer} = (\mathit{In}\ \mathit{InputLayer}, \mathit{id})$$

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

$$\begin{aligned}&\mathit{train} :: \mathit{Fix}\ \mathit{Layer} \to \mathit{Input} \to \mathit{Input} \to \mathit{Fix}\ \mathit{Layer} \\ &\mathit{train}\ \mathit{neuralNet}\ \mathit{initialInput}\ \mathit{desiredOutput} \\ &\quad = (\mathit{meta}\ \mathit{forwardAlg}\ h\ \mathit{backwardCoalg})\ \mathit{neuralNet} \\ &\quad \textbf{where} \\ &\quad h :: (\mathit{Fix}\ \mathit{Layer}, [\mathit{Input}] \to [\mathit{Input}]) \to (\mathit{Fix}\ \mathit{Layer}, \mathit{BackProp}) \\ &\quad h\ (\mathit{nn}, \mathit{forwardPass}) = \\ &\qquad \textbf{let}\ \mathit{inputStack} = \mathit{forwardPass}\ [\mathit{initialInput}] \\ &\qquad \textbf{in}\ (\mathit{nn}, \mathit{BackProp}\ \mathit{inputStack}\ [[\,]]\ [\,]\ \mathit{desiredOutput})\end{aligned}$$

This function now takes an extra argument — the initial input sample. We can see that after running the catamorphism, *h* uses the forward propagation function *forwardPass* found in the result and applies it to the initial input; this returns the inputs/outputs from forward propagating over every layer.

***Summary*** This completes our implementation of a fully connected network. This is given by a metamorphism, whose catamorphism evaluates a network to a forward propagation function, and whose anamorphism takes the results of forward propagation to reconstruct an updated network.

It should be noted that the approach of encoding neural network training as a metamorphism is an incomplete story. There is nothing strictly wrong with representing training as a composition of a fold followed by an unfold, however, this can be reduced to an even more self-contained recursion scheme system, which is later illustrated in Section 5.

## 4 Neural Networks à la Carte

Below, we compare two alternative ways of constructing the same neural network consisting of an input layer and two dense layers (where *w1* and *w2* are weight values).

```
fixNetwork =                          freeNetwork = do
  In (DenseLayer w2                     denselayer w2
    (In (DenseLayer w1                  denselayer w1
      (In (InputLayer)))))              inputlayer
                                        return ()
```

Our current method is shown by *fixNetwork* on the LHS. A better approach we could aim for is given by *freeNetwork* on the RHS. It's not difficult to see why *freeNetwork* is much more ideal, especially when dealing with significantly larger and more complex network architectures.

Using *Fix* to create networks, as seen in *fixNetwork*, is rather problematic — why this is, can be summed up in the following points:

**P.1** All the types of layers are currently defined to exclusively belong to one type of network. In reality, some types of layers can be used by different networks, and we shouldn't need to duplicate them for each network type i.e. this approach lacks modularity.

**P.2** Defining an instance of a neural network using *Fix f* requires that we must declare all of our layers at the same time — we cannot define them independently of each other. This is another lack of modularity in our *Fix f* approach.

**P.3** Instances of *Fix f* tend to be monolithic and incoherent due to an unattractive nesting of recursive data types.

A benefit of using recursion schemes is the ability to use free monads. This section introduces free monads and coproducts to our recursion scheme system in order to engage with these issues. As a result, we can achieve exactly what is shown in *freeNetwork*, which is a network defined using free monads, constructed using monadic do-notation. This takes influence from the 'data types à la carte' approach [Swierstra 2008].

### 4.1 Background: Free Monads and Coproducts

An issue we need to engage with is how *Fix* currently forces us to define entire networks in one go. *Free monads* enable us to use monadic combinators for elegant, composable constructions of functors; having free monads lets us separately define and connect segments of networks.

```
data Free f a = Var a | Op (f (Free f a))
```

Sometimes it makes more sense for the constructors of a data type to exist as their own type constructor. *Coproducts* allow us to create a data type as a coproduct of two type constructors *f* and *g*. They hence allow us to create independent data types for each type of layer — we can then define any neural network as the coproduct of its layers.

```
data (f :+: g) e = Inl (f e) | Inr (g e)
```

Data types à la carte [Swierstra 2008] provides a neat approach of constructing free monads and coproducts which this paper takes advantage of. To be able to conveniently construct coproducts, we use the concept of injections and smart constructors. The constraint *sub* :<: *sup* states that there must be some injection from *sub a* to *sup a*, where *sup* is any type signature that supports *sub*.

```
class (Functor sub, Functor sup) ⇒ sub :<: sup where
  inj :: sub a → sup a
instance Functor f ⇒ f :<: f where
  inj = id
instance (Functor f, Functor g) ⇒ f :<: (f :+: g) where
  inj = Inl
instance (Functor f, Functor g, Functor h, f :<: g) ⇒
    f :<: (h :+: g) where
  inj = Inr ∘ inj
```

The first instance states that (:<:) is reflexive. The second instance explains how to inject any value of type *f a* to a value of type (*f* :+: *g*) *a*, regardless of *g*. The third instance asserts that provided we can inject a value of type *f a* into one of type *g a*, we can also inject *f a* into a larger type (*h* :+: *g*) *a* by composing the first injection with an additional *Inr*. From this, we can then define the function *inject*.

```
inject :: (g :<: f) ⇒ g (Free f a) → Free f a
inject = Op ∘ inj
```

The function *inject* can be used in smart constructors to form values of type *Free f* () where *f* can be a coproduct.

### 4.2 Building Networks

We begin by decomposing our data type for fully connected networks into the individual data types *DenseLayer* and *InputLayer*. For simplicity, we ignore biases and define the parameters of the layers to consist only of weights.

```
data DenseLayer e = DenseLayer Weights e deriving Functor
data InputLayer e  = InputLayer deriving Functor
```

A fully connected network using free monads and coproducts would then be represented by the following type definition:

```
type FCNetwork = Free (InputLayer :+: DenseLayer) ()
```

Being able to represent neural networks as the coproduct of their types of layers promotes modular, reusable code. Layers are thus no longer seen as exclusively associated with a type of network, but rather a network is built up of independently existing layer types. This resolves problem **P.1**.

Recall the function *inject* from Section 4.1. Using this, we implement smart constructors for *DenseLayer* and *InputLayer*. These will inject the given layer into a free monad where its functor represents a neural network containing that layer. These smart constructors mean we can now declare layers on their own, without needing to declare them in one go inside a definition of a neural network — this resolves problem **P.2**.

```
denselayer  :: (DenseLayer :<: f) ⇒ Weights → Free f ()
denselayer w = inject (DenseLayer w (Var ()))
inputlayer  :: (InputLayer :<: f) ⇒ Free f ()
inputlayer  = inject (InputLayer)
```

With these smart constructors, we can successfully construct a fully connected neural network of type *FCNetwork* using the approach seen in *freeNetwork* given earlier at the start of Section 4. If we compare *freeNetwork* to our previous method *fixNetwork*, this is a vast improvement in terms of practicality. The do-notation and absence of nested data structures means our definitions of network become much more coherent and modular. This resolves problem **P.3**.

### 4.3 Training Networks With Free Monads

We next look at reimplementing our recursion scheme system to accomodate the usage of free monads and coproducts — this involves rethinking how our algebra and coalgebra for forward and back propagation should be defined.

***Choosing New Carrier Types*** We first consider what carrier types should be used for the free monadic versions of our algebra and coalgebra. Recall that we previously used $(Fix\ Layer, [Input] \to [Input])$ and $(Fix\ Layer, BackProp)$ as carriers for our algebra and coalgebra respectively. The problem we encounter is that we can no longer use *Fix Layer* in the carrier if we are to progress with using free monads instead. Originally, the purpose of including *Fix Layer* was so that we could store the structure of the original neural network during forward propagation, and use this information to generate an updated network during back propagation.

We can solve this by instead storing the properties of each layer in some new data type during forward propagation — this information can be reused later during back propagation. However, rather than create another data type (e.g. *ForwardProp*) to go alongside *BackProp*, it is more convenient to define a more general data type, *PropData*. This would store any information needed during either forward and back propagation; it combines the types $[Input] \to [Input]$ and *BackProp* from our original carriers, but also introduces a new field *weightStack* which is a list we append each layer's weights onto during forward propagation.

```
data PropData = PropData {
                  forwardPass   :: [Input] → [Input]
                , inputStack    :: [Input]
                , weightStack   :: [Weights]
                , desiredOutput :: [Double]
                , outerDeltas   :: [Double]
                , outerWeights  :: Weights  }
```

Our carrier type for both the algebra and coalgebra will now simply be *PropData*.

### 4.4 Forward Propagation: Catamorphism

Next we look at implementing forward propagation as a free monadic catamorphism. Instead of using *cata* to represent catamorphisms, we now use the free monadic version which is referred to as *eval*. Note the introduction of a new parameter *gen*, a generator that allows us to interpret values of $a$ into the desired carrier $b$.

```
eval :: Functor f ⇒ (f b → b) → (a → b) → Free f a → b
eval alg gen (Var x) = gen x
eval alg gen (Op op) = alg (fmap (eval alg gen) op)
```

Given that we use coproducts and each type of layer has its own data type, in order to make the algebra function *forwardAlg* work for each layer type during a catamorphism, we must create a class *ForwardAlg* — layers can then derive from this to state that they support and implement forward propagation *forwardAlg* as an algebra.

```
class (Functor h) ⇒ ForwardAlg h where
  forwardAlg :: h PropData → PropData
```

Creating an instance of *forwardAlg* for *DenseLayer* is almost exactly the same as our previous implementation of fully connected networks, except now we must deal with accessing and modifying values from the data type *PropData*.

```
instance ForwardAlg DenseLayer where
 forwardAlg (DenseLayer w propData)
   = let forwardPass' = (λ inputStack →
         let input  = head inputStack
             output = forward w input
         in  (output : inputStack)) ∘ (forwardPass propData)
```

Also, we are required to push the current layer's weights onto the list of weights *weightStack*, as a means of storing the structure of the original network to later allow back propagation to correctly reconstruct an updated network.

```
         weightStack' = w : (weightStack propData)
     in propData { forwardPass = forwardPass'
                 , weightStack = weightStack' }
```

Creating an instance of *forwardAlg* for *InputLayer* should just return the most basic possible *PropData* value.

```
instance ForwardAlg InputLayer where
  forwardAlg InputLayer = PropData id [[]] [] [] [] [[]]
```

Before we can be done with forward propagation, a new function we are required to design is the generator; this is the second argument of *eval* and is a function of type $(a \to b)$. It is responsible for changing the type () in *Free* (*InputLayer* :+: *DenseLayer*) () to our desired carrier type *PropData*.

```
gen :: () → (PropData)
gen () = PropData id [[]] [] [] [] [[]]
```

We define a function *gen* as our generator — given any input, this returns a value of *PropData*. What value this takes is not important, we only need to make sure that the required type is satisfied. Hence we make this function insert the most basic possible value of *PropData*.

### 4.5 Back Propagation: Anamorphism

Next we begin looking at implementing back propagation as an free monadic anamorphism. Instead of using *ana* to represent anamorphisms, we now use the free monadic version which is referred to as *build*.

```
build :: Functor f ⇒ (b → Either a (f b)) → b → Free f a
build f = either Var (Op ∘ fmap (build f)) ∘ f
```

This means that our coalgebra will actually need to be of type $b \to Either\ a\ (f\ b)$. Just like with our algebra, we will create a class *BackwardCoalg* for our coalgebra from which derived

instances state that they support and implement back propagation *backwardCoalg* as a coalgebra. However, the only instance which needs to be derived for this is the entire fully connected neural network, i.e. (*InputLayer* :+: *DenseLayer*).

```
class (Functor h) ⇒ BackwardCoalg h where
  backwardCoalg :: PropData → Either () (h PropData)
```

We will now create an instance for (*InputLayer* :+: *DenseLayer*). In order to know whether to create an *InputLayer* or *DenseLayer*, we take a look at the list of weights *weightStack* that we stored during forward propagation inside our *PropData* value.

```
instance BackwardCoalg (InputLayer :+: DenseLayer) where
  backwardCoalg propData
  = case (weightStack propData) of
```

If *weightStack* is not empty, then we back propagate as usual to generate an updated *DenseLayer*. This takes almost exactly the same approach as our previous coalebra implementation for *DenseLayer* — the only difference is we update the *PropData* value to contain the *tail* of *weightStack*, so that the first element will always be the weights associated with the current layer being processed.

```
    (w : ws) →
      let (updatedWeights, delta) = backward propData
          inputStack'  = tail (inputStack propData)
          weightStack' = tail (weightStack propData)
          propData' = propData { outerDeltas  = delta
                               , inputStack   = inputStack'
                               , weightStack  = weightStack'
                               , outerWeights = w    }
      in  Right (inj (DenseLayer updatedWeights propData'))
```

If *weightStack* is empty, then we know that all of the weights for the layers have been taken off, so all that is left is to return an *InputLayer*.

```
    []       → Right (inj InputLayer)
```

Regardless of what kind of layer we return, note that our result is always injected into the coproduct (*InputLayer* :+: *DenseLayer*) and then wrapped in the constructor *Right* to satisfy the required type:

```
Either () ((InputLayer :+: DenseLayer) PropData).
```

### 4.6 Training A Neural Network: Metamorphism

We are now ready to combine both our catamorphism and anamorphism to train a fully connected network as a metamorphism using free monads and coproducts. We create the function *train* which updates a neural network using a metamorphism given an initial input and a desired output.

```
train :: FCNetwork → Input → Input → FCNetwork
train nn initInput desOutput =
  (build backwardCoalg ∘ h ∘ eval forwardAlg gen) nn
  where
```

The intermediary function *h* is defined to process the catamorphism's output to be used by the anamorphism.

```
  h :: PropData → PropData
  h pd = let outputs = (forwardPass pd) [initInput]
         in pd { inputStack   = outputs
               , desiredOutput = desOutput }
```

Applying this to a free monadic neural network *nn* will successfully train it and return an updated network.

## 5 Training with a Single Fold

Having introduced free monads into the equation, there now exists a new problem in our current choice of implementation. Unfolding produces a coinductive data structure and hence may generate an infinite term that cannot be folded, since folds only work only over inductive structures. Free monads are an inductive type, thus unfolding with free monads is therefore not necessarily guaranteed to terminate. To resolve this, we therefore need to redefine back propagation as a free monadic catamorphism — this means coming up with an algebra for back propagation.

### 5.1 Back Propagation Revisited

Let's first discuss why we used anamorphisms for back propagation in the first place. Back propagation needs to start from the last layer and move backwards towards the input layer — this is because in order to back propagate over a layer, we need information from having back propagated over the layer after it. Anamorphisms let us process layers in this required order, whereas catamorphisms begin evaluating at the first layer and work towards to the last layer. However, this doesn't prevent catamorphisms from doing jobs which anamorphisms can do, just as unfold can be defined in terms of fold. Our problem is thus, "*how do we back propagate over a layer in a forward direction along the network, if we are missing required information from the next layer?*".

Now that the context of our issue is laid out, we can propose a solution towards designing a back propagation algebra. If we incorporate the idea of continuations in the algebra's carrier type and choose to return a function rather than a value, this lets us delay returning an updated layer and instead return a function that says, "*given some back propagation data from the next layer, use this to perform back propagation over the current layer and all previous layers*".

Our back propagation algebra's carrier will therefore be type *PropData* → *FCNetwork*. Let's create a new class for this, *BackwardAlg*, which contains the function *backwardAlg*. This class takes two parameters: parameter *h* represents the layer we want to back propagate over, and parameter *g* represents the entire network in the form of a coproduct of all layer data types.

```
class (Functor h, Functor g) ⇒ BackwardAlg h g where
  backwardAlg :: h (PropData → Free g ())
              →  (PropData → Free g ())
```

Defining *backwardAlg* for *InputLayer* consists of simply returning a function which always returns an *InputLayer* injected into a free monad.

```
instance BackwardAlg InputLayer (InputLayer :+: DenseLayer)
  where
  backwardAlg InputLayer = (λ _ → inject InputLayer)
```

Defining *backwardAlg* for *DenseLayer* consists of returning a function *backPass′* such that when given the propagation data from the next layer, will perform back propagation.

```
instance BackwardAlg DenseLayer (InputLayer :+: DenseLayer)
  where
  backwardAlg (DenseLayer w backPass) = backPass′
    where
    backPass′ :: PropData → FCNetwork
```

First, this computes the new propagation data and updated weights for the current layer.

```
    backPass′ propData =
      let (w′, delta) = backward propData
          propData′ = propData { outerDeltas  = delta
                               , ...          = ...    }
```

The updated weights can be used to replace the original weights of the current layer. In addition, having computed the new propagation data means that we have sufficient information to allow the previous layer to perform back propagation. We hence use the function *backPass* (evaluated from the previous layer) and provide it the new propagation data as an argument — this returns the updated version of the previous layer. The entire process is a chaining of promises between layers to update their previous layer.

```
      in inject (DenseLayer w′ (backPass propData′))
```

We now have everything we need to construct a catamorphism for back propagation. Defining a function *train* which encodes the neural network training as the composition of two catamorphisms is very straightforward to do. However, rather than demonstrating this idea, we can actually skip this and go one step further — we can encode neural network training as a *single* catamorphism.

### 5.2 Training as a Single Fold

The way we have designed our back propagation algebra's carrier to delay the construction of an updated network means it is possible to actually perform both of our algebras in a single catamorphism. By taking advantage of the 'banana split' property of folds [Meijer et al. 1991], any pair of applications of fold to the same list can always be combined into a single application of fold that generates a pair. This can also be applied to our network data structure. To demonstrate, we will define an algebra which simultaneously computes the forward pass and a function which computes the backwards pass.

***Defining An Algebra For Training*** As always, we first choose a carrier type; let this be the tuple (*PropData*, *PropData* → *Free g* ()), containing both of the original carrier types for the forward and backward propagation algebras.

Next, we define a class *TrainAlg* which describes layers that can perform the function *trainAlg* — an algebra using the carrier we have just chosen. This can be understood as the combination of classes *ForwardAlg* and *BackwardAlg*. Similar to before, the class *TrainAlg* takes two parameters: parameter *h* represents the layer data type we are training, and parameter *g* represents the coproduct of all layer data types in the neural network.

```
class (Functor h, Functor g) ⇒ TrainAlg h g where
  trainAlg :: h (PropData, PropData → Free g ())
           →  (PropData, PropData → Free g ())
```

Defining instances of *trainAlg* for *DenseLayer* and *InputLayer* is straightforward to do and takes the same approach for both.

For *DenseLayer*, the function *trainAlg* can be broken down into two halves where each half consists of essentially copy and pasting the previous algebra implementations for forward propagation and back propagation.

```
instance TrainAlg DenseLayer (InputLayer :+: DenseLayer)
  where
  trainAlg (DenseLayer weights (propData, backPass))
    = (propData′, backPass′)
    where
```

For forward propagation, we can refer to how *forwardAlg* is defined in Section 4.4.

```
    propData′ :: PropData
    propData′ = ....
```

For back propagation, we can refer to how *backwardAlg* is defined in Section 5.1.

```
    backPass′ :: PropData → FCNetwork
    backPass′ propData = ...
```

Above, *propData′* represents the result of our forward propagation algebra, i.e. a *PropData* value, and *backPass′* represents the result of our back propagation algebra, i.e. a function which takes this *PropData* value to return us an updated neural network. Both of these results are returned in a tuple.

For *InputLayer*, its algebra implementations for forward and back propagation can simply be placed side by side in a tuple.

```
instance TrainAlg InputLayer (InputLayer :+: DenseLayer)
  where
```

```
trainAlg InputLayer = ( PropData id [[]] [] [] [] [[]]
                      , λ _ → inject InputLayer)
```

***Training*** What's left to do is show use this to encode the training of a network as a single catamorphism — so let's define a function *train* to do this.

```
train :: FCNetwork → Input → Input → FCNetwork
train network initInput desOutput =
```

The first thing we need to do is change the generator function *gen* to accomodate the new carrier type. As always, we simply aim to return the most basic value possible which is of the algebra's carrier type. This is incidentally the same value which *trainAlg* on *InputLayer* returns.

```
  let gen :: () → (PropData, PropData → FCNetwork)
      gen () = (PropData id [[]] [] [] [] [[]]
               , λ _ → inject InputLayer)
```

Secondly, although we no longer use a metamorphism, we still require its intermediary function *h* to take the *PropData* result from forward propagation and correctly process it for back propagation.

```
      h :: PropData → PropData
      h pd = let outputs = (forwardPass pd) [initInput]
             in pd {inputStack   = outputs
                   , desiredOutput = desOutput}
```

We then come to the actual catamorphism *eval* which uses *trainAlg* to perform both forward and back propagation over a network in a single process. This returns a tuple containing the value *propData* acquired from forward propagation, and a function *updateNetwork* of type *PropData* → *FCNetwork* acquired from back propagation.

```
      (propData, updateNetwork) = eval trainAlg gen network
```

We can wrap this up by applying *updateNetwork* to the processed version of *propData*; this returns an updated neural network as the result of training with a single input sample.

```
  in updateNetwork (h propData)
```

***Summary*** We have demonstrated that the entire system of a fully connected network can be modelled by a single catamorphism, with some minimal extra processing on its results - this displays a certain beauty in the simplicity in how we can capture a neural network's mechanism. This is a significant realisation in building new understanding of the relationship between recursion schemes and neural networks. It also illustrates how despite the simplicity in the recursive pattern of fold, the introduction of tuples and functions as first-class values provided by Haskell allows folds to have greater expressive power than one might assume.

This finishes the story for fully connected networks. In Section A and Section B, we show how this approach can be extended to more complicated types of neural networks.

## 6 Related Work

The story of the relationship behind neural networks and recursion schemes has very little, if at all any, detailed analysis. The only two found mentions linking recursion schemes to neural networks [Berényi et al. 2017; Olah 2015] are limited to simple hypothetical suggestions. The most relevant discussion is presented by Olah [2015] who corresponds how we represent neural networks to type theory in functional programming. We believe that their perspective is an extremely healthy one. They makes a number of insights; a network as a chain of composed functions, recurrent networks as folds/unfolds. Tree nets are recognised as a potentially suitable network type with relevance to recursion schemes. However, this is expected as tree structures are a well known example for giving rise to the formulation of structured recursion schemes. The writer emphasises that these interpretations are entirely speculative; our paper not only fully fleshes out these vague, untested ideas, but demonstrates a strong relationship between recursion schemes and neural networks, furthermore exploring how we can exploit the advantages which recursion schemes bring.

To the best of our knowledge, there exists no formal work on the application of free monads or coproducts towards deep learning. One article [Panarin 2017] discusses using free monads to perform gradient descent, in contrast to our work which uses them for network construction.

More generally, there exists excellent research on realising the parallels between deep learning and functional programming: de Buitléir et al. [2013] discuss how a functional paradigm compliments the mathematical nature of neural networks; Gavranović [2019] builds a categorical formalism around a class of neural networks closed under composition (CycleGAN); Elliott [2018] develops a generalized automatic differentiation algorithm; Wang et al. [2019] investigate automatic differentiation techniques and show a connection between reverse-mode AD and delimited continuations.

## 7 Conclusion

This paper has discussed an approach of encoding the structure and training of neural networks as a recursion scheme system. Although only fully connected networks have been considered, the choice of architecture is pertinant such that the ideas demonstrated can be transferred to more complex network types; we specifically go through two other extremely popular network types, recurrent (Section A) and convolutional (Section B) networks, in the appendix.

Furthermore, we show how to integrate free monads and coproducts with recursion schemes — taking influence from *data types à la carte*, we use these to improve the modularity and practicality in both the construction and representation of neural networks. During this, it was shown that a single fold can represent the entire training process of a (feed-forward) neural network.

## References


Martin Abadi. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/

James A Anderson. 1995. *An introduction to neural networks*. MIT press.

Dániel Berényi, András Leitereg, and Gábor Lehel. 2017. Applications of structured recursion schemes. (2017).

Richard Bird and Ross Paterson. 1999. Generalised folds for nested datatypes. *Formal Aspects of Computing* 11, 2 (1999), 200–222.

F Chollet. 2015. Keras. https://github.com/fchollet/keras

Amy de Buitléir, Michael Russell, Mark Daly, Felipe Zapata, and Angel J Alvarez. 2013. The Monad. Reader Issue 21. (2013).

Conal Elliott. 2018. The simple essence of automatic differentiation. *Proceedings of the ACM on Programming Languages* 2, ICFP (2018), 1–29.

Bruno Gavranović. 2019. Learning Functors using Gradient Descent. (2019).

Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. 2008. A novel connectionist system for unconstrained handwriting recognition. *IEEE transactions on pattern analysis and machine intelligence* 31, 5 (2008), 855–868.

Jeff Hale. 2018. Deep Learning Framework Power Scores - Towards Data Science. https://towardsdatascience.com/deep-learning-framework-power-scores-2018

Ralf Hinze, Nicolas Wu, and Jeremy Gibbons. 2013. Unifying Structured Recursion Schemes. *SIGPLAN Not.* 48, 9 (Sept. 2013), 209–220. https://doi.org/10.1145/2544174.2500578

Graham Hutton. 1999. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming* 9, 4 (1999), 355–372.

Aleksei Grigorevich Ivakhnenko and Valentin Grigorevich Lapa. 1967. Cybernetics and forecasting techniques. (1967).

Eunji Jeong, Joo Seong Jeong, Soojeong Kim, Gyeong-In Yu, and Byung-Gon Chun. 2018. Improving the expressiveness of deep learning frameworks with recursion. In *Proceedings of the Thirteenth EuroSys Conference*. 1–13.

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. *ACM SIGPLAN Notices* 50, 12 (2015), 94–105.

Warren S McCulloch and Walter Pitts. 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5, 4 (1943), 115–133.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. 1991. Functional programming with bananas, lenses, envelopes and barbed wire. In *Conference on Functional Programming Languages and Computer Architecture*. Springer, 124–144.

Christopher Olah. 2015. Neural Networks, Types, and Functional Programmings.

K. Panarin. 2017. Gradient descent with free monads. https://towardsdatascience.com/gradient-descend-with-free-monads-ebf9a23bece5

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

J. Puget. 2018. The Most Popular Language For Machine Learning and Data Science is. https://www.kdnuggets.com/2017/01/most-popular-language-machine-learning-data-science.html

N Shobha Rani, Pramod N Rao, and Paul Clinton. 2018. Visual recognition and classification of videos using deep convolutional neural networks. *International Journal of Engineering & Technology* 7, 2.31 (2018), 85–88.

Hasim Sak, Andrew W Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. (2014).

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE Transactions on Neural Networks* 20, 1 (2008), 61–80.

Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.

Jürgen Schmidhuber, Daan Wierstra, and Faustino J Gomez. 2005. Evolino: Hybrid neuroevolution/optimal linear search for sequence prediction. In *Proceedings of the 19th International Joint Conferenceon Artificial Intelligence (IJCAI)*.

Wouter Swierstra. 2008. Data types à la carte. *Journal of functional programming* 18, 4 (2008), 423–436.

Aaron Van den Oord, Sander Dieleman, and Benjamin Schrauwen. 2013. Deep content-based music recommendation. In *Advances in neural information processing systems*. 2643–2651.

Fei Wang, Daniel Zheng, James Decker, Xilun Wu, Grégory M Essertel, and Tiark Rompf. 2019. Demystifying differentiable programming: Shift/reset the penultimate backpropagator. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 1–31.

Junho Yim, Jeongwoo Ju, Heechul Jung, and Junmo Kim. 2015. Image classification using convolutional neural networks with multi-stage feature. In *Robot Intelligence Technology and Applications 3*. Springer, 587–594.

## A Convolutional Neural Networks

A *convolutional network* handles higher dimensional data and is well known for analysing image [Yim et al. 2015], audio [Van den Oord et al. 2013] and video data [Rani et al. 2018]. As a result, they are more structurally and computationally complex than fully connected networks. Similar to fully connected networks, convolutional networks are from the family of feed-forward networks meaning data propagates along a single direction. However whereas fully connected networks have two types of layers, convolutional networks have five, each of which performs a different operation on the input data they are passed:

- **Convolutional Layer** This uses a set of filters as weights to perform convolution over the input data. The 'stride length' is the distance the filter is shifted across the image between iterations of convolution.
- **ReLu Layer** This applies the function $f(x) = max(0, x)$ to the image.
- **Pooling Layer** Given a spatial extent (region size) and a stride length (distance shifted between iterations), regions of the images are shifted over and only the maximum value at each region is returned.
- **Dense Layer** This reshapes the 3D 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.

The structure of any convolutional network is such that the first layer is an input layer and the last layer is a dense layer. All intermediate layers can be any combination and ordering of layer types excluding the input layer. Given the number of different layers involved, this illustrates the benefits of introducing free monads and coproducts; without this, our definition and construction of convolutional networks would be rather impractical and monolithic.

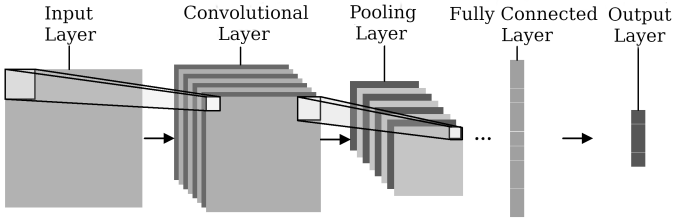


**Figure 2.** Convolutional Neural Network

What this section emphasises is that regardless of the complexity of our computations increasing, how data flows within a network is ultimately what affects the recursion scheme used. Both fully connected and convolutional networks are *feed-forward networks* and hence share the same pattern of data flow. Consequently, we should be able to model training as a single fold.

There are only three modifications that are worth talking about — this refers to the main components which a recursion scheme system comprises of. Every further detail which unravels from this is conceptually unchallenging.

1. Define a data type which represents a convolutional neural network and the different layers that it can contain. This is simply the act of creating a suitable data type for each of the five types of layers.
2. Define a carrier type for an algebra which can represent forward and back propagation in a convolutional network. This consists of ensuring that any new information needed to propagate through a convolutional network is accommodated by the carrier.
3. For each of the five layers, implement an algebra such that they represent their associated computations performed during forward and back propagation. This is little more than translating a set of formula equations into code. These computations are self-contained and do not affect our underlying recursion scheme model.

Having carried these steps out, the training of a convolutional network can then be encoded as a catamorphism in a similar manner as fully connected networks.

## B Recurrent Neural Networks

This section aims to further demonstrate the presence of recursion schemes in a neural network which uses a different data propagation flow — a recurrent neural network, or more specifically, a long-short-term-memory (LSTM) recurrent network. Both of our previous examples are feed-forward networks; the main difference between recurrent and feed-forward networks, is that recurrent networks 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, they can be used for time-series recognition [Schmidhuber et al. 2005], handwriting recognition [Graves et al. 2008], and speech recognition [Sak et al. 2014]. Figure 3 depicts a recurrent network with two layers.

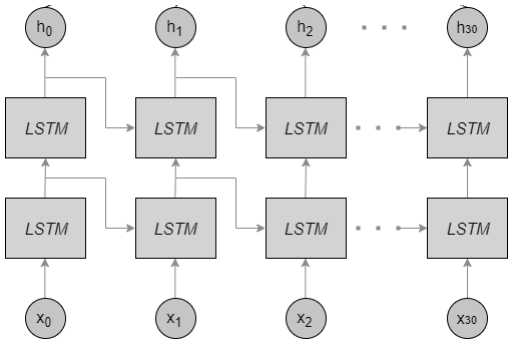


**Figure 3.** Recurrent Network (Depth Of 2 Layers)

A recurrent network consists of a vertical stacking of layers, where each layer is composed of a row of 'cells' stacked horizontally. Cells encompass the idea of being able to update and cache states whilst also computing multiple other variables. Just by looking at the circulation pattern, the potential for recursive patterns is exciting. In each layer, data can be seen to forward propagate horizontally across the row of cells, whilst also upwards into the next layer. Likewise, back propagation occurs in the opposite direction in both the cell-level and layer-level. This means that we could employ a recursion scheme system on both the cell-level and layer-level of the network — this potentially describes a kind of double catamorphism, where one is nested inside another.

**Finding A Data Type** First we will find a data type for cells. When considering a single row of cells, we can treat this row as its own 'neural network'. We define type constructors for cells, which consists of *Cell* and *InputCell*, similarly to how layers were implemented. Whatever fields a cell should contain are just the necessary parameters belonging to the cell.

```
data Cell k = Cell CellParams k deriving Functor
data InputCell k = InputCell deriving Functor
```

If we now define type constructors for the layers of cells, this consists of *Layer* and *InputLayer*. A layer would also have to contain its own parameters and the cells associated with it.

```
data Layer k = Layer LayerParams
                     (Free (InputCell :+: Cell) ()) k
  deriving Functor
data InputLayer k = InputLayer
  deriving Functor
```

**Training A Recurrent Network** This consists of running a catamorphism which folds over the network's layers. We take a similar approach to fully connected networks in the sense that we want to evaluate to the 2-tuple:

(*PropData*, *PropData* → *Free* (*InputLayer* :+: *Layer*) ())

These respectively represent forward and back propagation over layers.

The main difference is that within the algebra of the layer-level catamorphism, we must also run a *cell-level* catamorphism which evaluates to the 2-tuple:

(*PropData*, *PropData* → *Free* (*InputCell* :+: *Cell*) ())

These respectively represent forward and back propagation over cells.

This new pattern of data propagation seen in an recurrent network gives rise to a recursion scheme system consisting of a layer-level catamorphism whose algebra calls a cell-level catamorphism.

# C Training: Examples And Results

## C.1 Training A Fully Connected Network

We now demonstrate that our fully connected network is capable of learning. We aim to model the sine function; we use input data consisting of random decimal numbers and their corresponding desired outputs to be the sine function of those numbers.

The network we will construct can be seen in Figure 4, where the input value is sent to three input nodes, and then propagated through three intermediate layers to the final output layer consisting of a single node. This network is represented by *fcNetwork*; the function *randMat2D* is used to initalise weights, where *randMat2D m n* generates a randomly valued matrix of dimension $m \times n$.

```
fcNetwork :: FCNetwork
fcNetwork = do
  denselayer (randMat2D 3 3)
  denselayer (randMat2D 3 3)
  denselayer (randMat2D 1 3)
  inputlayer
  return ()
```

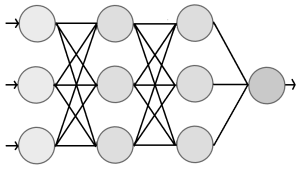

**Figure 4.** Fully Connected Network

Next, we create a function *trains* which lets us run multiple samples through the network rather than just a single sample which *train* performs. This simply folds over a list of samples and desired outputs whilst calling the function *train*.

```
trains :: Free (InputLayer :+: DenseLayer) ()
  → [Input]
  → [Input]
  → Free (InputLayer :+: DenseLayer) ()
trains nn initInputs desOutputs
  = foldr (λ (initInput, desOutput) nn →
                train nn initInput desOutput) nn samples
    where samples = zip initInputs desOutputs
```

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 *trains* along with the example network above.

```
main = do
  inputFile ← readFile "sine_data"
  labelsFile ← readFile "sine_labels"
  let initInputs  = map read (lines inputFile) :: [Input]
      desOutputs = map read (lines labelsFile) :: [Input]
      nn          = trains fcNetwork initInputs desOutputs
  print (show nn)
```

Below we see the change in error as the amount of run samples increases, using sample sizes of 800 (in Figure 5) and 1400 (in Figure 6).

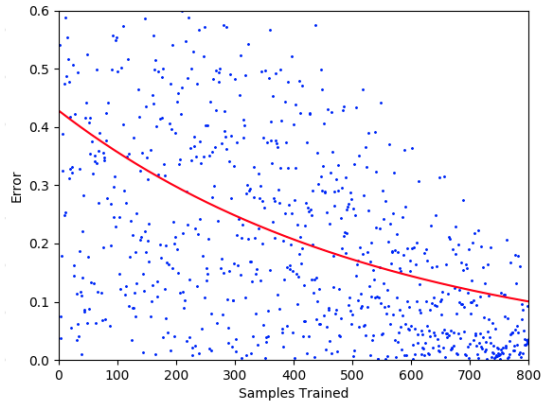


**Figure 5.** Sample Size 800, Correlation Coefficient: -0.54

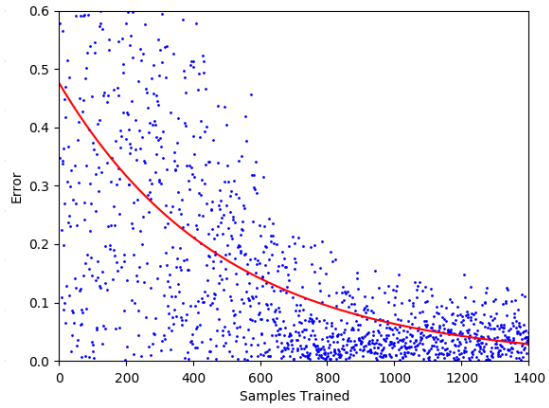


**Figure 6.** Sample Size 1400, Correlation Coefficient: -0.65

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 improves the resulting accuracy of the network; this is also indicated by the higher magnitude in negative correlation coefficient (-0.65 for a sample size of 1400 in contrast to -0.54 for a sample size of 800).

### C.2 Training A Convolutional Neural Network

Convolutional neural networks are used primarily to classify images. To demonstrate the ability of a full implementation of convolutional networks to learn, we choose 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 in Figure 7 where dimensions are given in the form *(width* × *height* × *depth)* × *number of filters*. An input image of dimensions $(7 \times 7 \times 1)$ is provided to the network, resulting in an output vector of length two where each value corresponds to the probability of the image being classified as either an 'X' or an 'O' symbol.

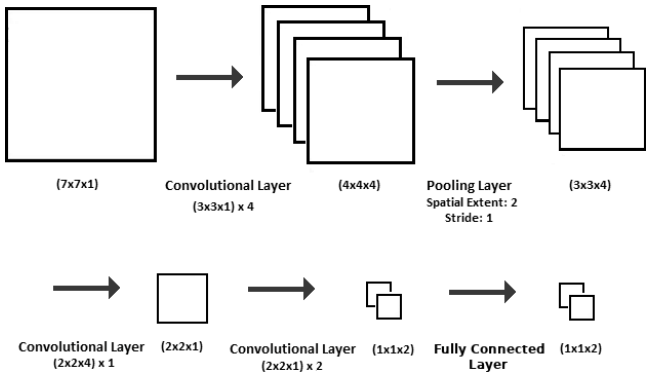


**Figure 7.** Convolutional Network

This network is constructed with *convNetwork*; we use smart constructors to inject each layer type into a free monad *Free ConvNetwork* (), where *ConvNetwork* is the coproduct of all five layer data types found in a convolutional network. We generate random weights using *randMat4D* and provide matrices of zeros for the biases using *zeroMat2D*.

```
type ConvNetwork = (InputLayer : + : ConvLayer : + :
                    PoolLayer : + : ReLuLayer : + : DenseLayer)
convNetwork :: Free ConvNetwork ()
convNetwork = do
  denselayer
  convlayer (randMat4D 2 2 1 2) (zeroMat2D 2 1)
  convlayer (randMat4D 2 2 4 1) (zeroMat2D 1 1)
  poollayer 1 2
  convlayer (randMat4D 3 3 4 1) (zeroMat2D 1 1)
  inputlayer
  return ()
```

Here we see the change in error as the amount of trained samples increases, using sample sizes of 300 (in Figure 8) and 600 (in Figure 9).

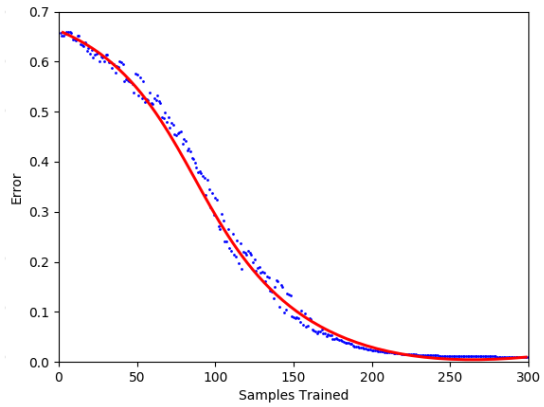


**Figure 8.** Sample Size Of 300, Correction Coefficient: -0.747

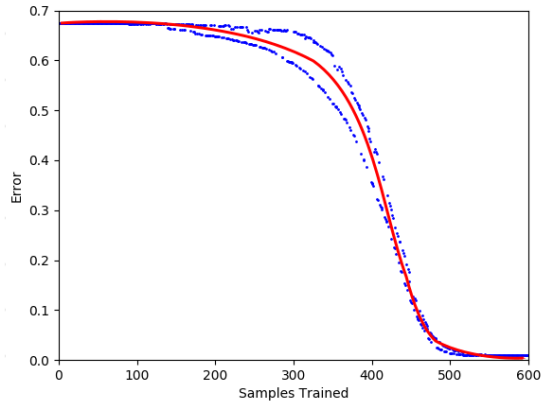


**Figure 9.** Sample Size Of 600, Correction Coefficient: -0.876

This shows a pleasing negative 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.

### C.3 Training A Recurrent Neural Network

Recurrent 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 recurrent network implementation, we 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.

A diagram of the recurrent network used is shown in Figure 10, consisting of two layers of five cells.

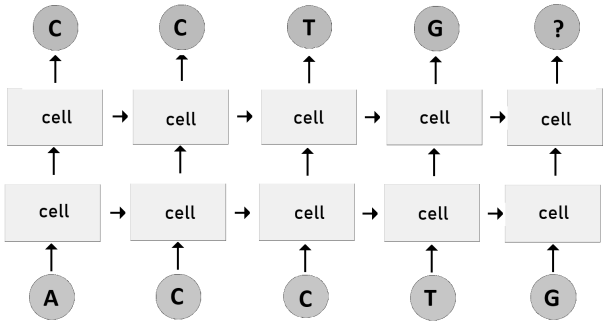


**Figure 10.** Recurrent Neural Network

Below, we see the change in error as the amount of trained samples increases, using sample sizes of 300 (in Figure 11) and 600 (in Figure 12).

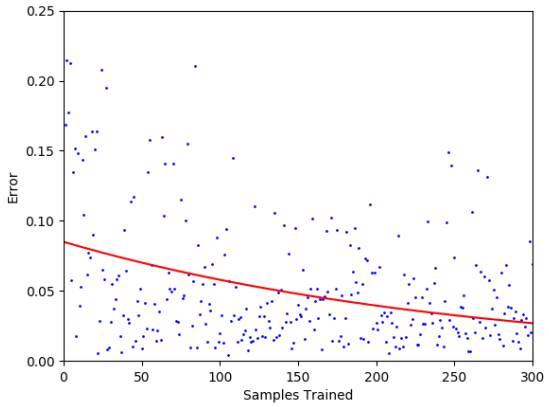


**Figure 11.** Sample Size 300, Correction Coefficient: -0.314

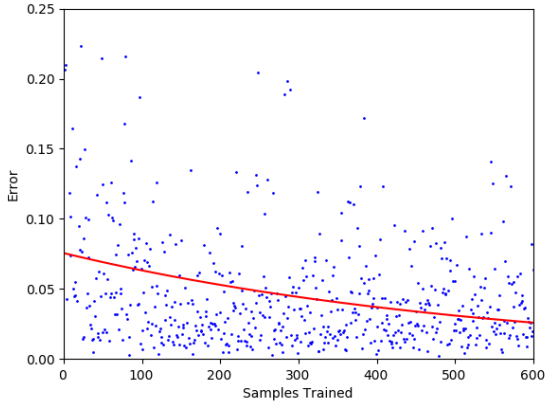


**Figure 12.** Sample Size 600, Correction Coefficient: -0.254

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 recurrent networks to learn. In contrast to the previous results, the correlation coefficient for the larger sample size of 600 is lower in magnitude than for the sample size of 300. We hypothesis that this, in addition to the subtlety of the negative curvatures, is due to the high complexity of recurrent networks — achieving more distinctive results would require an informed approach to the architecture and usage of modern deep learning tools applicable to recurrent networks.