



SJÄLVSTÄNDIGA ARBETEN I MATEMATIK

MATEMATISKA INSTITUTIONEN, STOCKHOLMS UNIVERSITET

A category-theoretic analysis of backpropagation in neural networks

av

Daniel Collin

2018 - No K32

A category-theoretic analysis of backpropagation in neural networks

Daniel Collin

Självständigt arbete i matematik 15 högskolepoäng, grundnivå

Handledare: Erik Palmgren

2018

A category-theoretic analysis of backpropagation in neural networks

Daniel Collin

August 29, 2018

Abstract

In this thesis we try to establish a compositional framework for learning algorithms based on category theory, in particular the theory of monoidal categories. By showing how to construct neural networks with string diagrams in the category \mathbf{Para} , the category of Euclidean spaces and parametrized differentiable functions, we gain insight in how learning algorithms can be constructed by gluing together smaller learning algorithms to form larger ones.

We then analyze gradient descent and backpropagation, a combined technique common for training neural networks, through the lens of category theory in order to show how our composed learning algorithms can be trained on the category \mathbf{Learn} , the category of sets and learning algorithms.

Additionally we find that recurrent neural networks give rise to a general construction on \mathbf{Learn} that allows us to define learning algorithms defined over sequences of objects.

Acknowledgments

I would like to thank my supervisor Erik Palmgren for his invaluable feedback as well as nudging me in the direction that I ended up taking. I would also like to thank Ben Ward for his time and very helpful feedback. Finally I would like to thank Christopher Olah for allowing and encouraging me with great enthusiasm to use his diagrams in my text.

Contents

1	Introduction	5
1.1	Background	5
1.2	Disposition	5
2	Categories	7
2.1	Functors	8
2.2	Isomorphism	8
2.3	Natural transformation	8
2.4	Natural isomorphism	8
3	Monoidal categories	8
3.1	Set is a monoidal category with its Cartesian product	9
3.2	String diagram	10
3.3	Braided category	10
4	Neural networks	11
5	Parametrized differentiable functions	14
5.1	Bimonoid	16
5.2	Building neural networks in <i>Para</i>	18
6	Learning algorithms and training	19
6.1	Gradient descent and backpropagation	20
6.1.1	Gradient descent	20
6.1.2	Backpropagation	20
6.1.3	Training	21
6.2	The category Learn	21
6.3	Functor	23
6.4	RNN	26
6.5	LSTM	28
7	Discussion	33
7.1	Further directions	33
8	References	34

1 Introduction

Our aim with this thesis is to provide a sketch of how category theory and especially the theory behind monoidal categories provide a suitable framework for discussing machine learning algorithms, in particular those pertaining to the field of deep learning. A inspirational and foundational source to our undertaking which spurred this direction is to be found in Fong *et al.* *Backprop as a Functor: A compositional perspective on supervised learning* [2].

1.1 Background

Machine learning, or more precisely the machine learning subfield deep learning, has in recent years seen quite a renaissance. From models beating the world champion in Go to discovering early stage cancer at a higher-than-average accuracy there are many reasons to pay attention to its development. In particular the construction of differentiable learning models have seen the most success; initially stemming from neural networks these wholly differentiable models have taken a life of their own, with proper abstract computing machines being defined only using these differentiable structures inherited from artificial neural networks.

As deep learning diverges from its roots in neural networks new terms have been suggested such as differentiable programming [7]. While this might seem as a simple attempt of re-branding and semantics it might hold a more substantial difference: the way in which these models are being built is similar to the way functional programming is done [6]. Considering the links between functional programming and category theory through lambda calculus on one hand [4] as well as work done by John Baez *et al.* [5] showing that the diagrams and networks often used in engineering and applied sciences belong to monoidal categories on the other hand, we find that category theory and in particular the theory behind monoidal categories prove to be a fitting scope for further investigating the properties of these learning algorithms.

A hope is that while shedding this surface-level category-theoretic light upon learning algorithms may not be the end-all of the odd marriage between machine learning and category theory it might serve as a slight paving of the path to understanding the way these models are being constructed in a high-level and foundational view without concerning ourselves too much the details of implementation.

1.2 Disposition

We will begin by giving the reader a reminder of the required theory both in neural networks and category theory in Sections 2,3 and 4. This includes basic definitions of monoidal categories as well as string diagrams which we will see become a natural way of portraying learning algorithms and neural networks in particular. On the deep learning end we will give a brief overview of neural networks and the way they are trained.

In Sections 5 and 6 we will begin developing the machinery described in Fong *et al.* [2] for the compositional learning algorithms, starting with constructing examples

of known networks in *Para* the category of parametrized differentiable functions and then showing how we can train these in the monoidal category *Learn*.

Further on in Section 6 we will find that *Learn* contains structure beyond that of traditional neural networks, in fact it is general enough to imagine many different sorts of learning algorithms. In particular we will see that recurrent neural networks exist as a more general recurrent learner on *Learn*.

2 Categories

This section of fundamental definitions is from Awodey's *Category Theory* [1].

Definition 1. A category \mathcal{C} consists of objects and morphisms (also known as arrows). For every morphism f in \mathcal{C} there are objects $\text{dom}(f)$ and $\text{cod}(f)$ known as the domain respectively the codomain of f . We write $f : A \rightarrow B$ to denote A as the domain and B as the codomain of f .

Given morphisms $f : A \rightarrow B$ and $g : B \rightarrow C$ in \mathcal{C} there is a morphism $g \circ f : A \rightarrow C$ known as the composite of f and g .

For each object A in \mathcal{C} there is a morphism $1_A : A \rightarrow A$ known as the identity morphism.

Such that associativity of morphism composition holds

$$h \circ (g \circ f) = (h \circ g) \circ f$$

and that the identity morphism acts as a unit for morphisms with respect to composition:

$$f \circ 1_A = f = 1_B \circ f$$

Remark 1. In Fong *et al.* [2] another definition is used. It is essentially the same however the order of composition is switched and the $;$ operator is used instead so that for $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ we have that $f;g : X \rightarrow Z$. It can be read as “first do f , then do x ”. This notation is quite useful when dealing with diagrams as they are also read left to right, however we will not be using it as it may prove somewhat confusing to deviate from standard notation.

Throughout this section we will be running with the simplest of example of a category, in some sense the starting point and canonical example of a category, the category *Set* of sets as objects and morphism as functions between them. To see that *Set* indeed fulfills our prior definition is quite trivial as composition coincides with ordinary function composition, which is associative, and the identity morphism coincides with the identity function.

Set is a convenient category as its properties are familiar from ordinary mathematics and set theory, however it is worth mentioning that *Set* is far from the only category there is.

Example

There is a category *FinVect* of finite-dimensional vector spaces and linear maps between these. To see that *FinVect* is a category note that linear maps respect composition and associativity, and that the identity function applied to a vector space is a linear map.

Example

There is a category \mathbb{E} of Euclidean spaces as objects and differentiable maps as morphisms. That \mathbb{E} is indeed a category can be derived from Set as \mathbb{E} is Set with all other sets than Euclidean spaces and all morphisms other than differentiable maps removed. Since the identity function is a differentiable map and composition of differentiable maps yields a differentiable map the categorical structure of Set is preserved.

2.1 Functors

Definition 2. A functor

$$F : \mathbb{C} \rightarrow \mathbb{D}$$

is a mapping from the category \mathbb{C} to the category \mathbb{D} , or more precisely from the objects of \mathbb{C} to the objects of \mathbb{D} and from the morphisms of \mathbb{C} to the morphisms of \mathbb{D} such that:

1. $F(f : A \rightarrow B) = F(f) : F(A) \rightarrow F(B)$,
2. $F(g \circ f) = F(g) \circ F(f)$,
3. $F(id_A) = id_{F(A)}$.

Remark 2. These requirements are, as many of these fundamental definitions, very intuitive. We simply demand that a functor works in the same spirit as a homomorphism, in other words it preserves the categorical structure such that composition and unit are respected.

2.2 Isomorphism

Definition 3. We say that a morphism f in some category \mathbb{C} is an isomorphism if it has an inverse, in other words that there exists some other morphism $f^{-1} : Y \rightarrow X$ in \mathbb{C} such that $f \circ f^{-1} = 1_Y$ and $f^{-1} \circ f = 1_X$.

2.3 Natural transformation

Definition 4. Given two functors $F, F' : \mathbb{C} \rightarrow \mathbb{D}$ a natural transformation $\alpha : F \Rightarrow F'$ assigns for every object X in \mathbb{C} a morphism $\alpha_X : F(X) \rightarrow F'(X)$ such that given any morphism $f : X \rightarrow Y$ in \mathbb{C} it holds that $\alpha_Y \circ F(f) = F'(f) \circ \alpha_X$.

2.4 Natural isomorphism

Definition 5. A natural isomorphism is a natural transformation such that α_X is an isomorphism for every object X in \mathbb{C} .

3 Monoidal categories

We will mostly be concerned with so called monoidal categories from here on as they provide a natural framework for interpreting networks that describe flow of information known as string diagrams. This section of definitions regarding monoidal

categories is from John Baez *et al. Physics, topology, Logic and Computation: A Rosetta Stone* [4].

Definition 6. A monoidal category \mathcal{C} is a category equipped with a functor $\otimes : \mathcal{C} \times \mathcal{C} \rightarrow \mathcal{C}$ known as the monoidal product, a unit object I , a natural isomorphism α called the associator such that for every triplet X, Y, Z of objects in \mathcal{C} we have an isomorphism $\alpha_{X,Y,Z} : (X \otimes Y) \otimes Z \rightarrow X \otimes (Y \otimes Z)$ and finally a couple of natural isomorphisms called the unitors assigning to each object in X in \mathcal{C} a couple of isomorphisms $l_X : I \otimes X \rightarrow X$ and $r_X : X \otimes I \rightarrow X$ such that:

$$\begin{array}{ccc} (A \otimes I) \otimes B & \xrightarrow{\alpha_{A,I,B}} & A \otimes (I \otimes B) \\ & \searrow r_A \otimes 1_B & \swarrow 1_A \otimes l_B \\ & A \otimes B & \end{array}$$

commutes for all $A, B \in \mathcal{C}$, and

$$\begin{array}{ccccc} & & ((A \otimes B) \otimes C) \otimes D & & \\ & \swarrow \alpha_{A \otimes B, C, D} & & \searrow \alpha_{A, B, C} \otimes 1_D & \\ (A \otimes B) \otimes (C \otimes D) & & & & (A \otimes (B \otimes C)) \otimes D \\ & \searrow \alpha_{A, B, C \otimes D} & & \downarrow \alpha_{A, B \otimes C, D} & \\ & & A \otimes (B \otimes (C \otimes D)) & \swarrow 1_A \otimes \alpha_{A, B, C} & \\ & & & & A \otimes ((B \otimes C) \otimes D) \end{array}$$

commutes for all $A, B, C, D \in \mathcal{C}$.

Remark 3. These diagrams describe some quite intuitive properties about the monoidal category in relation to its product; the first diagram demands that uniting from the right should be the same as uniting from the left and the second diagram demands we should be able to associate in any order we wish. Should the associator and unitors be identities we say that the monoidal category is strict.

3.1 Set is a monoidal category with its Cartesian product

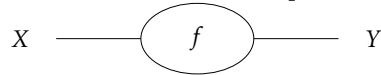
We ask ourselves if our faithful servant *Set* can be equipped with a monoidal product to make it into a monoidal category? The answer is, of course, yes and the most natural candidate for this product is the Cartesian product with an arbitrary singleton set

as the unit. However, there is a small caveat: Cartesian product is not strictly associative. To see this consider the product $A \times (B \times C)$, in other words the set $\{(a, (b, c)) \mid a \in A, b \in B, c \in C\}$ and compare it to $(A \times B) \times C$, the set $\{((a, b), c) \mid a \in A, b \in B, c \in C\}$. These two sets are obviously not equal since $(a, (b, c)) \neq ((a, b), c)$ and as such we need to use as our associator the natural isomorphism $A \times (B \times C) \cong (A \times B) \times C$.

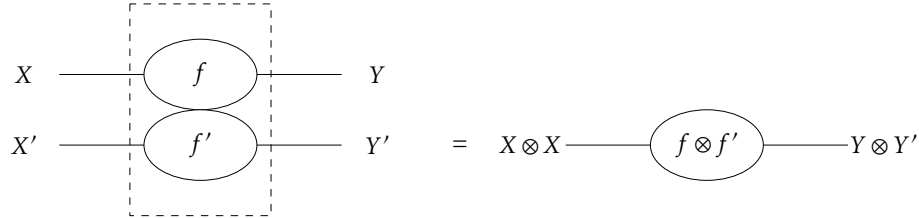
It is important to note that we can create different monoidal categories by our choice of product, for instance *Set* equipped with the disjoint union as its product and empty set as the unit is also an example of a monoidal category [4].

3.2 String diagram

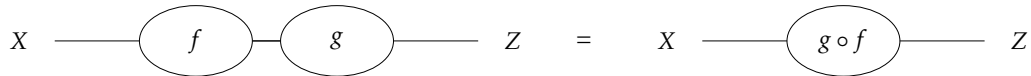
A string diagram is a way of representing monoidal categories by changing our standard way of depicting stationary objects and morphisms between them to stationary morphisms and objects between them. An intuitive way of describing string diagrams is to think of the morphisms as machines or black boxes and the objects as wires going into and from them as their inputs and outputs.



Tensoring is done simply by parallel placement:



While composition is represented by connecting one morphism to another:

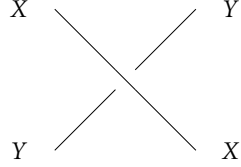


In general we can deform these diagrams quite heavily without losing any semantic content. We will not give a formal treatment of exactly how and why this is the case and refer the reader to Joyal and Street [8].

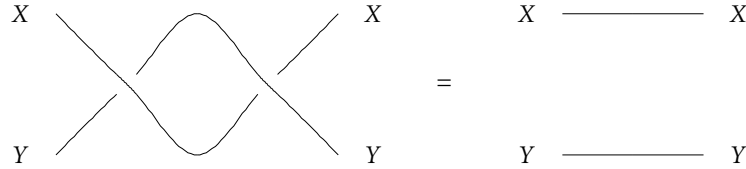
3.3 Braided category

Definition 7. A braided category \mathcal{C} is a monoidal category equipped with a natural isomorphism known as the braiding $b_{X,Y} : X \otimes Y \rightarrow Y \otimes X$. We can visualize this in

a string diagram as a crossing of strings:



Naturally Set, \times is a braided category with $b_{X,Y}$ being the isomorphism $b((x, y)) = (y, x)$. In fact Set, \times is what is known as a symmetric monoidal category [4] where applying a braiding twice is the same as doing nothing at all, illustrated by the following relation:



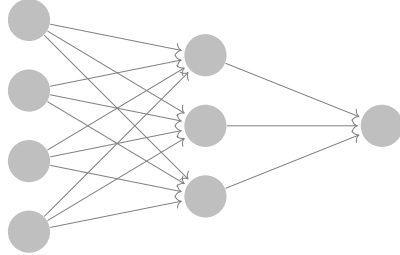
Which indeed is the case in Set, \times since:

$$b_{Y,X} \circ b_{X,Y}(x, y) = b_{Y,X}(y, x) = (x, y)$$

$$b_{Y,X} \circ b_{X,Y} = id_{X \times Y}$$

4 Neural networks

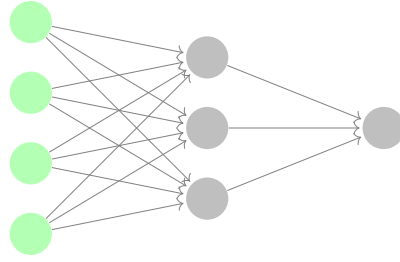
In this section we will remind ourselves of what neural networks are. This short introduction owes a lot to the excellent in-depth overview of deep learning in Goodfellow *et al.* [3].



A neural network, or artificial neural network to distinguish it from a biological neural network, is a directed graph that describes how to compose a series of functions or alternatively a flowchart of how input flows to output.

We can divide a neural network into so called layers, vertical segments of vertices. the layers themselves can be divided into three classes: input layer, hidden layers and output layer.

Input
layer



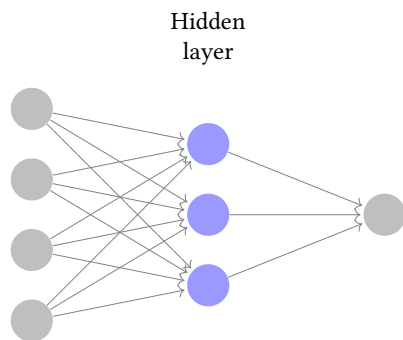
The input layer is where the input flows from, edges from the input layer dictates how the input will be distributed across the next layer. Any edge between vertices will have an associated weight that the input is multiplied with.

At each vertex other than the input layer the sum of the incoming inputs is run through a nonlinear function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$, commonly known as an activation function. Formally we can now define a neural network as a graph in the following manner:

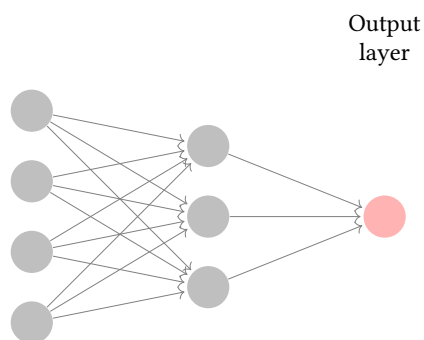
Definition 8. A neural network is a directed graph $\{V, E\}$ along with a real number w_i labeling each edge e_i and function $\sigma_j : \mathbb{R} \rightarrow \mathbb{R}$ labeling each vertex v_j not in the input layer. The real numbers labeling the edges are known as the weights of the edges and the functions labeling the vertices are known as the activation functions.

The activation function is the magic sauce of the neural network since without it the entire network is incapable of describing anything but linear relationships between its input and output and the only requirement of the activation function is that it is differentiable such that the whole network is differentiable.

The most commonly used activation function in modern deep learning is the ReLU or rectified linear unit $f(x) := \max(0, x)$ while perhaps the most canonical choice of activation function is the sigmoid function $f(x) := \frac{1}{1+e^{-x}}$ which stems from neuroscience [3]. ReLU however is not differentiable for $x = 0$, so in practice there are some techniques to circumvent this.

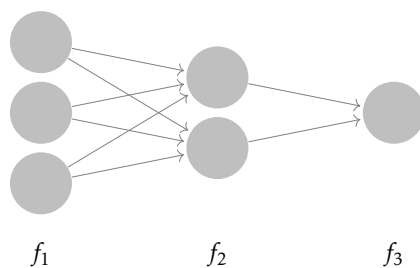


The hidden layers are all layers between the first and the final layer. A larger amount of hidden layers or size of hidden layers gives the neural network more “adjustable knobs” so that convergence during training might be achieved quicker.



The output layer is the output of the whole network, hence it has no outgoing edges only incoming ones.

Generally neural networks can be seen as blueprints for defining differentiable functions parametrized by their weights. Viewed in this light one usually takes every layer of the network to be a function composed with the rest of the network such that the whole network defines a composite function.



Since at every layer the only operations that are done is summing and applying

the activation functions the entirety of the composite function is differentiable.

5 Parametrized differentiable functions

In this section we will be looking at *Para*, the category of equivalence classes of differentiable parametrized functions in which we will find that we can reconstruct neural networks as composite morphisms.

The definition from Fong *et al.* [2]:

Definition 9. A Euclidean space is some real valued space \mathbb{R}^n for some natural number n .

Remark 4. \mathbb{R}^0 is the space containing only a single point, and as a set it is merely a singleton set.

Definition 10. A differentiable parametrized function $(P, I) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ consists of a Euclidean parameter space P and a differentiable function $I : P \times \mathbb{R}^n \rightarrow \mathbb{R}^m$.

Consider for example the differentiable function $g(p, x) := x + p$. Then there is a parametrized differentiable function $(\mathbb{R}, g) : \mathbb{R} \rightarrow \mathbb{R}$ such that $(\mathbb{R}, g)(x) = x + p$ for some p in \mathbb{R} . We say that g is parametrized by the parameter p and that \mathbb{R} is the parameter space of (\mathbb{R}, g) .

In order to have associativity in the category *Para* we need some way of considering two parametrized differentiable functions as equivalent if their parameter spaces can easily be transformed into the other.

Definition 11. We say that two parametrized differentiable functions $(P, I), (Q, J)$ are equivalent if there exists some invertible function $f : P \rightarrow Q$ such that f, f^{-1} are differentiable and $I(p, x) = J(f(p), x)$ for all $p \in P$. We will abuse notation when clear from context and write (P, I) to be equivalence classes of parametrized differentiable functions rather than the individual functions themselves.

If we continue with our parametrized differentiable function (\mathbb{R}, g) we can illustrate Definition 10 by considering an *equivalent* parametrized differentiable function (\mathbb{R}, g') where $g'(p, x) := x + \frac{p}{2}$. These two are equivalent by the definition since there is a differentiable function $f(x) := 2x$ that has a inverse differentiable inverse $f^{-1}(x) := \frac{x}{2}$ and $g(p, x) = x + p = g'(f(p), x)$ for all $p \in \mathbb{R}$.

Now that we have Definitions 10 and 11 we have the necessary prerequisites to formulate the theorem from Fong *et al.* [2]:

Proposition 1. The category *Para* of equivalence classes of differentiable parametrized functions together with monoidal product \otimes is a strict symmetric monoidal category such that:

$$\begin{aligned} (Q, J) \circ (P, I) &= (P \times Q, J \circ I) \\ J \circ I(q, p, a) &= J(q, I(p, a)) \end{aligned}$$

and

$$(P, I) \otimes (Q, J) = (P \times Q, (I, J))$$

with identity for some object X in $Para$ given by $(\mathbf{1}, id_X)$ and unit $\mathbf{1} = \mathbb{R}^0$.

Remark 5. Note that the morphisms of $Para$ are the equivalence classes of parametrized differentiable functions, hence our example functions from Definitions 10 and 11, (\mathbb{R}, g) and (\mathbb{R}, g') , are in fact the *same* morphism in $Para$.

Remark 6. We should be wary of that the monoidal product on $Para$ only acts as a Cartesian product for objects while for morphisms it is not quite a Cartesian product.

We must take care to note the difference between a morphism of $Para$ and its implementation function. A morphism $(P, I) : \mathbb{R}^n \rightarrow \mathbb{R}^m$ has an implementation function $I : P \times \mathbb{R}^n \rightarrow \mathbb{R}^m$, hence the morphism takes an object \mathbb{R}^n to an object \mathbb{R}^m . We will sometime abuse notation and simply write $I : \mathbb{R}^n \rightarrow \mathbb{R}^m$ to mean the morphism when the meaning is clear from context as to not deviate too much from the notation in Fong *et al.* [2].

Proof. We will show that $Para$ fulfills the definition of a strict monoidal category by verifying the properties one by one.

Associativity

$$(P_3, I_3) \circ ((P_2, I_2) \circ (P_1, I_1)) = (P_3 \times (P_2 \times P_1), I_3 \circ (I_2 \circ I_1))$$

reduces down to proving that $I_3 \circ (I_2 \circ I_1) = (I_3 \circ I_2) \circ I_1$:

$$\begin{aligned} I_3 \circ (I_2 \circ I_1)(p_3, p_2, p_1, a) &= I_3(p_3, I_2 \circ I_1(p_2, p_1, a)) \\ &= I_3(p_3, I_2(p_2, I_1(p, a))) \\ &= I_3 \circ I_2(p_3, p_2, I_1(p, a)) \\ &= (I_3 \circ I_2) \circ I_1(p_3, p_2, p_1, a) \end{aligned}$$

Identity

Identity is given by an arbitrary singleton set that we will denote $\mathbf{1}$ and the identity function, hence we have that for some $(P, I) : X \rightarrow Y$:

$$(\mathbf{1}, id_Y) \circ (P, I) = (\mathbf{1} \times P, id_Y \circ I)$$

Since $id_Y \circ I = I$ and $\mathbb{R}^0 \times \mathbb{R}^n = \mathbb{R}^n$ we have that $(\mathbf{1} \times P, id_Y \circ I) = (P, I)$. For id_X it is shown in an essentially identical manner.

Hence $Para$ is a category. We now show that it is strict symmetric monoidal: The monoidal product is defined as

$$(P, I) \otimes (Q, J) = (P \otimes Q, I \otimes J)$$

where

$$I \otimes J(q, p, a, c) = (I(p, a), J(q, c))$$

and for objects, equivalently parameter spaces, it is simply the Cartesian product such that $A \otimes B = A \times B$.

Identity

Braiding is given by the same isomorphism as in *Set* but trivially parametrized, $(\mathbf{1}, b_{X,Y})(x, y) = (y, x)$ which clearly is its own inverse.

Monoidality

Associator and unitors are all identities since $\mathbb{R}^0 \times \mathbb{R}^n = \mathbb{R}^{n+0}$ and $(\mathbb{R}^n \times \mathbb{R}^m) \times \mathbb{R}^k = \mathbb{R}^{n+m+k} = \mathbb{R}^n \times (\mathbb{R}^m \times \mathbb{R}^k)$, hence the category is strict symmetric monoidal. \square

5.1 Bimonoid

Since linear maps between vector spaces are differentiable we have a functor from *FinVect*, the category of finite vector spaces and maps between them, to *Para*. The functor $F : \text{FinVect} \rightarrow \text{Para}$ takes sets to sets and any linear map $\phi : \mathbb{R}^n \rightarrow \mathbb{R}^m$ to the trivially parametrized differential function $\phi' : 1 \times \mathbb{R}^n \rightarrow \mathbb{R}^m$. This equips every object in *Para* with the following morphisms, making every object into what is known as a bimonoid [2]:


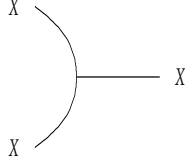
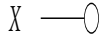
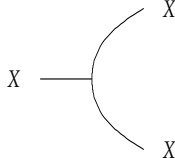
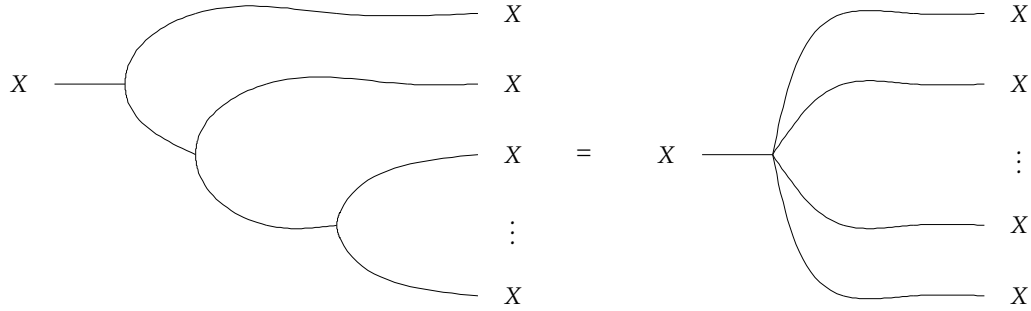
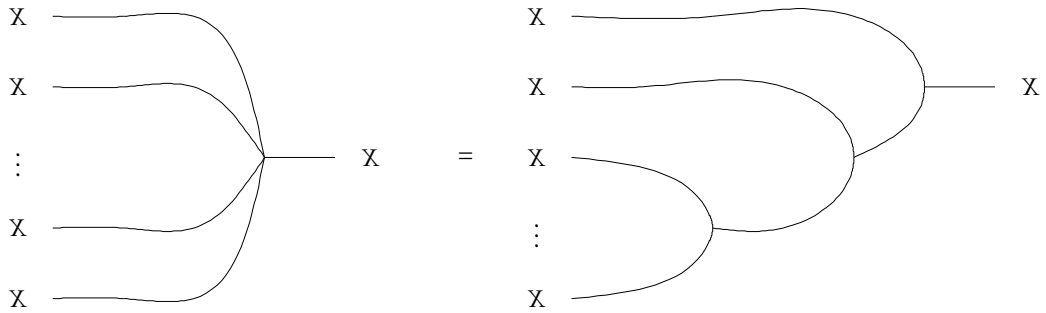
	$\eta : \mathbb{R}^0 \rightarrow X$	$I_\eta(p, 0) := 0$
	$\mu : X \times X \rightarrow X$	$I_\mu(p, x, y) := x + y$
	$\epsilon : X \rightarrow \mathbb{R}^0$	$I_\epsilon(p, x) := 0$
	$\delta : X \rightarrow X \times X$	$I_\delta(p, x) := (x, x)$

Figure 1: Table over bimonoid morphisms

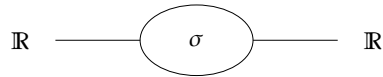
We will be using the following simplified notation utilizing the coassociativity of δ to make the string diagrams less cluttered:



Equivalently for μ , which is associative, we will depict multiple sequential applications as:



With these bimonoid morphisms under our belt we are almost able to generate any neural network as a composed morphism in *Para*. There are, however, a few building blocks missing:



This is the activation morphism, that given a choice of nonlinear differentiable activation function σ has implementation function $\sigma : \mathbb{R} \rightarrow \mathbb{R}$ and trivial parameter space 1.

$$\lambda : \mathbb{R} \rightarrow \mathbb{R}$$

This is the scalar multiplication morphism that has parameter space \mathbb{R} . Note that this is parametrized by a scalar that it multiplies the argument by. However we can create a similar but binary function that multiplies its two scalar arguments:

$$\lambda' : (\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R}$$

This is a trivially parametrized morphism where we instead have the “parameter” supplied by an external argument. We can in fact do this with any function in *Para* by moving its parameter to be an externally supplied input and have it trivially parametrized.

5.2 Building neural networks in *Para*

Given these functions above we can in fact reconstruct neural networks in *Para* not only graphically but as composite morphism.

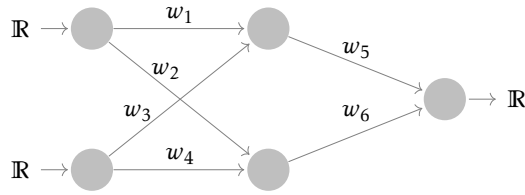


Figure 2: Three-layered neural network

Figure 2 is a simple three-layered feed-forward neural network consisting of an input layer, hidden layer and output layer. We wish to reconstruct this neural network in a *Para* string diagram:

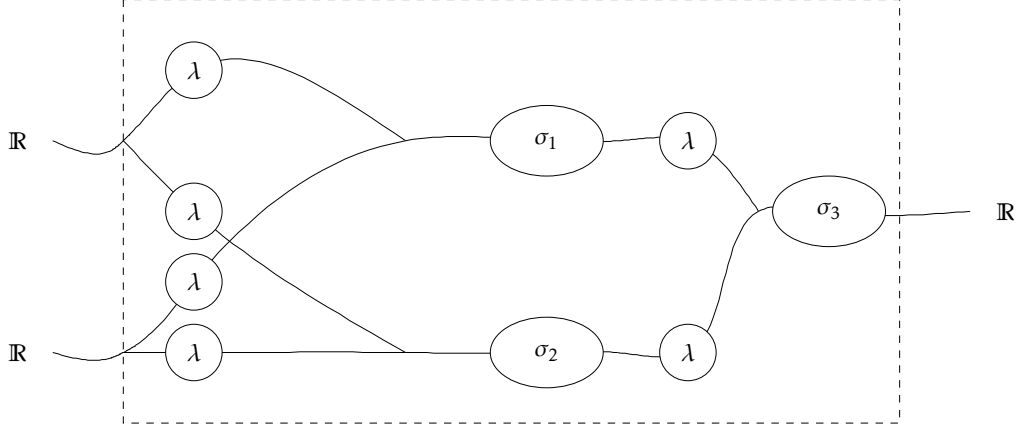


Figure 3: String diagram of the neural network in Figure 2.

The string diagram over *Para* in Figure 3 is not merely a graphical representation. It actually tells us that Figure 3 is the morphism

$$(\mathbb{R}^6, \sigma \circ \mu \circ (\lambda \otimes \lambda) \circ (\sigma \otimes \sigma) \circ (\mu \otimes \mu) \circ (id_{\mathbb{R}} \otimes b_{\mathbb{R}, \mathbb{R}} id_{\mathbb{R}}) \circ (\lambda \otimes \lambda \otimes \lambda \otimes \lambda) \circ (\delta \otimes \delta)) : \mathbb{R}^2 \rightarrow \mathbb{R}$$

with the implementation function

$$I : \mathbb{R}^6 \times \mathbb{R}^2 \rightarrow \mathbb{R}$$

$$I(p_6, p_5, p_4, p_3, p_2, p_1, x, y) := \sigma_3(p_5 \sigma_1(p_1 x + p_3 y) + w_6 \sigma_2(p_2 x + p_4 y))$$

This implementation function I corresponds the function defined by Figure 2 up to matching the correct weights. Generally when we compose morphisms (P, I) and (Q, J) we get the implementation function $J \circ I(q, p, a)$ hence the first parameters are the parameters belonging to last function to be composed.

6 Learning algorithms and training

So far we have done enough groundwork to categorically construct our neural networks in a manner we consider equivalent to a standard neural network, but the most essential property is still missing: the ability to train or update our networks.

When we train a network f with parameters p we wish to approximate some function by feeding the network training pairs a, b , consisting of input and output

examples, to update the parameters p such that $f(a)$ given the updated parameters is in some sense closer to b . What we then need in order to train our networks is some way of taking a composed function in *Para* and adding to its “interface” the ability to consume a training pair to update its parameters.

We will add these mechanisms to our constructions by the ways of a functor that takes a morphism in *Para* to a so called “learner”.

6.1 Gradient descent and backpropagation

A few words have to be said about how one trains parametrized differentiable functions. The main reason why we are interested in *differentiable* functions is that having wholly differentiable functions allow us to use the techniques of backpropagation and gradient descent. By using correct examples of input/output we improve our “approximation” of the function we wish to target by using the derivative to readjust parameters so that our function achieves results that, in terms of the error between our output and the correct output, are closer to the training examples.

6.1.1 Gradient descent

Gradient descent, as an optimization algorithm, works by minimizing some function by taking a step in the opposite direction of the gradient itself. This is the technique we use to train neural networks with a slight difference: instead of minimizing the function we wish to approximate we minimize the error function that describes the error between the output of the approximation and that of a correct training example.

Computing a gradient numerically can become very computationally expensive as these functions grow in size. The technique used, instead, to compute the gradient of the composed function is something called backwards autodifferentiation or backpropagation.

6.1.2 Backpropagation

Backpropagation, also known as reverse autodifferentiation, is a technique for efficiently calculating the gradient of a composite function using the well-known chain rule. More precisely when computing the gradient expression we utilize the fact that the composite function will be built from building blocks of primitive functions with prior known derivatives.

To illustrate the backpropagation technique consider the composite function $f_2 \circ f_1 : \mathbb{R} \rightarrow \mathbb{R}$ depicted as a graph in Figure 4:

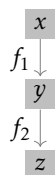


Figure 4: Function graph

Say that we wish to know the derivative of $f_2 \circ f_1$ with respect to x , in other words how changes in the input x affects the output of the composite function. The chain rule tells us that $(f_2 \circ f_1)' = (f_2' \circ f_1)f_1'$. Note now that the problem of knowing the derivative of the composed function has been reduced to knowing the derivatives of the components making up the composed function. Hence we find that as long as we know the derivatives of the primitive parts of the composed functions we can compute an entire expression for the gradient or derivative of the composite function without much computation. Backpropagation generalizes to composite functions of higher dimensions as well, we only need to consider the multidimensional chain rule instead [3].

So why do we call it backpropagation and not simply the chain rule? There is an additional aspect to this computation technique which is that any composed function will have repeated expressions with parts using the same derivatives hence we can store these expressions and reuse them for reduced computational complexity.

6.1.3 Training

With this brief introduction to gradient descent and backpropagation we can now sketch the updating algorithm that gives our parametrized function a new set of parameters: given an input/output training example a, b we propagate a through the network to get b' , the value given by our parametrized function with the current set of parameters p , and we get the error or loss between b , the training example, and b' . We then compute the gradient with respect to this error by the backpropagation technique and to get our new set of parameters we simply take an ϵ -step for some ϵ , also known as the *learning rate*, in the direction that minimizes the error. This is our new parameter p' .

6.2 The category Learn

We want to stress that neural networks are simply one form of learning algorithm, in fact they are simply one form of parametrized differentiable functions. As mentioned in the introduction neural networks loosely derive their structure from the biological brain and while this analogy has proven very fruitful at first there is much reason to relax this connection. As differentiable programming becomes a field of its own it is within our interest to put emphasis on this fact, that the differentiable functions we construct are a superset of the neural networks. This superset is not in any strict rigorous mathematical way but rather in the way they are being thought of and constructed.

With this said what then is a learning algorithm? If we only consider the supervised form of learning algorithms, meaning we supply it with input and output examples, then a learning algorithm consists of a parametrized function we wish to approximate, its parameters and finally some way of updating these parameters based on training examples. Furthermore we wish these to be composable in some sense.

We will be using the definition given by Fong *et al.* [2]:

Definition 12. A learning algorithm, or learner, consists of a tuple (P, I, U, r) , $I : P \times X \rightarrow Y$, $r : P \times X \times Y \rightarrow X$, and finally $U : P \times X \times Y \rightarrow P$ where P, X, Y are sets.

Remark 7. I , or the implementation function, corresponds to the function that is being approximated, P to its parameters and U to the function that updates our parameters. However the odd one out, r , is a bit technical; it is included to facilitate composition of morphisms. To understand r we first need to show the categorical structure of *Learn*, the symmetric monoidal category of sets as objects and equivalence classes of learners as morphisms.

Proposition 2. There exists a symmetric monoidal category *Learn* with sets as objects, equivalence classes of learners as morphisms with monoidal product \otimes such that given two learners (P, I, U_I, r_I) and (Q, J, U_J, r_J) we have that:

$$(Q, J, U_J, r_J) \circ (P, I, U_I, r_I) = (P \times Q, J \circ I, U_J \circ U_I, r_J \circ r_I)$$

where

$$\begin{aligned} J \circ I(q, p, a) &= J(q, I(p, a)) \\ U_J \circ U_I(p, q, a, c) &= (U_I(p, a, r_J(q, I(p, a), c)), U_J(q, I(p, a), c)) \\ r_J \circ r_I(p, q, a, c) &= r_I(p, a, r_J(q, I(p, a), c)) \end{aligned}$$

Identity morphism for an object X is given by $(\mathbf{1}, id_X, !, \pi_2)$, in other words a tuple of a singleton set, identity function $id_X : \mathbf{1} \times X \rightarrow X$ trivially parametrized by $\mathbf{1}$, $!$ the unique function from a set to the singleton set and finally $\pi_2 : \mathbf{1} \times X \times X \rightarrow X$ is the trivially parametrized second projection of the Cartesian product.

Monoidal product of objects is Cartesian product while monoidal product for morphism is:

$$(P, I, U_I, r_I) \otimes (Q, J, U_J, r_J) = (P \times Q, (I, J), (U_I, U_J), (r_I, r_J))$$

The symmetric braiding for some object $X \times Y$ is given by $(\mathbf{1}, b_{X,Y}, !, b_{X,Y} \circ \pi_2) : A \times B \rightarrow B \times A$ where $b_{X,Y}(x, y) = (y, x)$ is the braiding from *Set* in Section 4.3.

Remark 8. Now while P , I and U might be fairly self explanatory r does require some additional motivation. This motivation can be seen in the composition rule of update functions:

$$U_J \circ U_I(q, p, a, c) = (U_I(p, a, r_J(q, I(p, a), c)), U_J(q, I(p, a), c))$$

At the time of composition the scope of a single update function is only as far as the function it belongs to, when it is composed with another function we need some way for the composed update function of a learner $L_2 \circ L_1 : X \rightarrow Z$ only to require training pairs of the form $(x, z) \in X \times Z$ and not the intermediate example X, Y . The connection between the first function and the second is where this y normally would serve as input. This is exactly what r provides, it passes to the prior function the corrected or improved input that can be used in the update algorithm for the first function. It can be further illustrated in the composition rule for request functions:

$$r_J \circ r_I(q, p, a, c) = r_I(p, a, r_J(q, I(p, a), c))$$

Here we can see that the last learner in the pipeline, (Q, J, U_J, r_J) , first relays a corrected input for the first learner (P, I, U_I, r_I) to take in consideration when relaying a corrected input for the entirety of the function.

Proof. For a proof that this category is well-defined see Appendix A on p.26 in Fong *et al.* [2]. The proof is essentially the same as the proof given for Proposition 1. \square

6.3 Functor

As we have now made acquaintances with *Learn* we ask ourselves the following: is there some way of moving a morphism from *Para* to *Learn* such that we can train it with gradient descent and backpropagation? The answer is yes, by the means of a functor!

The functor is given by following theorem from Fong *et al.* [2].

Theorem 1. *Given learning rate $\epsilon \in \mathbb{R}_{>0}$, $\alpha : \mathbb{N} \rightarrow \mathbb{R}_{>0}$, and error function $e : \mathbb{R} \rightarrow \mathbb{R}$ such that $\frac{\partial e}{\partial x}(a, -)$ is invertible for each $a \in \mathbb{R}$ we can define a injective-on-objects symmetric monoidal functor*

$$L : \text{Para} \rightarrow \text{Learn}$$

*that sends objects to objects and sends a morphism $(P, I) : A \rightarrow B$ in *Para* to the morphism $(P, I, U_I, r_I) : A \rightarrow B$ in *Learn* defined in the following way:*

$$\begin{aligned} U_I(p, a, b) &:= p - \epsilon \nabla_p E_I(p, a, b) \\ r_I(p, a, b) &:= f_a \left(\frac{1}{\alpha_B} \nabla_a E_I(p, a, b) \right) \\ E_I(p, a, b) &:= \alpha_B \sum_k e(I_k(p, a), b_k) \end{aligned}$$

where α_B is $\alpha(n)$ such that n is the dimension of the codomain B and f_a is the component-wise application of $\left(\frac{\partial e}{\partial x}(a_i, -) \right)^{-1}$ where i indexes over the components of a .

To fully understand the functor it helps to see the proof which is why we will reproduce the proof from Appendix B in Fong *et al.* [2].

We will be using slightly different notations than in the original proof, one reason for this is found in Remark 1 regarding the use of $;$ instead of \circ in [2]. Another difference is that we will use slightly different index variables than in the original proof solely for visibility.

Proof. The functor is by definition injective-on-objects since it sends every object to itself. We will prove that the functor preserves compositionality by showing it for each component of a learner separately. Let $(P, I), (Q, J)$ be morphisms of *Para*. Then $F(J \circ I) = F(J) \circ F(I)$:

Update

$$\begin{aligned} (U_J \circ U_I)(q, p, a, c) &= (U_I(p, a, r_J(q, I(p, a), c)), U_J(q, I(p, a), c)) \\ &= (p - \epsilon \nabla_p E_I(p, a, r_J(q, I(p, a), c))), q - \epsilon \nabla_q E_J(q, I(p, a), c)) \end{aligned}$$

Whereas for the composition prior we get that:

$$U_{J \circ I}(p, q, a, c) = (p - \epsilon \nabla_p E_{J \circ I}(p, q, a, c), q - \epsilon \nabla_q E_{J \circ I}(p, q, a, c))$$

Hence the update functions are equal if:

$$\nabla_p E_I(p, a, r_J(q, I(p, a), c)) = \nabla_p E_{J \circ I}(p, q, a, c) \quad (1)$$

$$\nabla_q E_J(q, I(p, a), c) = \nabla_q E_{J \circ I}(p, q, a, c) \quad (2)$$

We start by showing the first equality:

$$\begin{aligned} &\nabla_p E_I(p, a, r_J(q, I(p, a), c)) \\ &= \alpha_B \nabla_p \sum_k e(I_k(p, a), r_{J_k}(q, I(p, a), c)) && \text{Definition of } E_I \\ &= \left(\alpha_B \sum_k \frac{\partial e}{\partial x}(I_k(p, a), r_{J_k}(q, I(p, a), c)) \frac{\partial I_k(p, a)}{p_l} \right)_l && \text{Let } l \text{ index over the components of } \nabla_p \\ &= \left(\alpha_B \sum_k \frac{\partial e}{\partial x}(I_k(p, a), f_{I(p, a)}(\frac{1}{\alpha_B} \nabla_{I(p, a)} E_J(q, I(p, a), c))) \frac{\partial I_k(p, a)}{p_l} \right)_l && \text{Definition of } r_J \\ &= \left(\sum_k \nabla_{I(p, a)} E_J(q, I(p, a), c) \frac{\partial I_k(p, a)}{p_l} \right)_l && f_x \text{ is the inverse of } \frac{\partial e}{\partial x}(a, -) \\ &= \left(\alpha_C \sum_k \nabla_{I(p, a)} \sum_m e(J_m(q, I(p, a), c), c_m) \frac{\partial I_k(p, a)}{p_l} \right)_l && \text{Definition of } E_J, \text{ let } m \text{ index over } J(q, I(p, a)) \\ &= \left(\alpha_C \sum_m \frac{\partial e}{\partial x}(J_m(q, I(p, a), c), c_m) \sum_k \frac{\partial J_m}{\partial I_k(p, a)} \frac{\partial I_k(p, a)}{p_l} \right)_l && \text{Definition of } \nabla_{I(p, a)} \\ &= \left(\alpha_C \sum_m \frac{\partial e}{\partial x}(J_m(q, I(p, a), c), c_m) \frac{\partial J_m}{p_l} \right)_l && \text{Multivariable chain rule backwards} \\ &= \alpha_C \nabla_p \sum_m e(J_m(q, I(p, a), c), c_m) && \text{Definition of } \nabla_p \\ &= \nabla_p E_{J \circ I}(p, q, a, c) && \text{Definition of } E_{J \circ I} \end{aligned}$$

For the second equality we only need to expand the expression for the error function:

$$\begin{aligned}
& \nabla_q E_J(q, I(p, a), c) \\
&= \nabla_q \alpha_C \sum_k e(J_k(q, I(p, a)), c_k) \\
&= \nabla_q \alpha_C \sum_k e((J \circ I)_k(p, q, a), c_k) \\
&= \nabla_q E_{J \circ I}(p, q, a, c)
\end{aligned}$$

Thus the functor preserves composition of update functions.

Request

$$r_I(p, a, r_J(q, I(p, a), c)) = r_{J \circ I}(p, q, a, c)$$

From the definition of the request function r_I we get:

$$r_I(p, a, r_J(q, I(p, a), c)) = f_a\left(\frac{1}{\alpha_B} \nabla_a E_I(p, a, r_J(q, I(p, a), c))\right)$$

Notice the expression $\nabla_a E_I(p, a, r_J(q, I(p, a), c))$ is the same expression as on the left hand side of the first equation we proved in the above section concerning the update functions, hence changing a for p in the same proof will yield that:

$$\begin{aligned}
& r_I(p, a, r_J(q, I(p, a), c)) \\
&= f_a\left(\frac{1}{\alpha_B} \nabla_a E_I(p, a, r_J(q, I(p, a), c))\right) && \text{Equation (1) with } a \text{ instead of } p \\
&= f_a\left(\frac{1}{\alpha_B} \nabla_a E_{J \circ I}(p, q, a, c)\right) && \text{By the definition of } r_J \\
&= r_{J \circ I}(p, q, a, c)
\end{aligned}$$

Identity

The identity morphism in *Para* for some object A is the trivially parametrized function $id_A : 1 \times A \rightarrow A$. Since id_A does not have any parameters the update function of $F(id_A)$ is also trivial and takes the singleton set to the singleton set, hence it coincides

with the unique function $!$. As for the request function r_{id_A} we have:

$$\begin{aligned}
r_{id_A}(\mathbf{1}, a, b) &= f_a\left(\frac{1}{\alpha_B} \nabla_a E_I(\mathbf{1}, a, b)\right) \\
&= f_a\left(\frac{1}{\alpha_B} \nabla_a \alpha_B \sum_k e(id_{A_k}(\mathbf{1}, a), b_k)\right) && \text{Definition of } E_I \\
&= f_a\left(\nabla_a \sum_k e(a_k, b_k)\right) \\
&= \left(\left(\frac{\partial e}{\partial a_i}(a_i, -)\right)^{-1} \circ \left(\frac{\partial}{\partial a_i} \left(\sum_k e(a_k, b_k)\right)\right)\right)_i && \text{Definition of } \nabla_a \text{ and } f_a \\
&= \left(\left(\frac{\partial e}{\partial a_i}(a_i, -)\right)^{-1} \circ \left(\frac{\partial e}{\partial a_i}(a_i, b_i)\right)\right)_i && \text{Partial derivative of sum is 0 for every term except when } i = k \\
&= b
\end{aligned}$$

We can thus conclude that r_{id_A} is the trivially parametrized second projection $\pi_2 : 1 \times A \times B \rightarrow B$, hence $F(id_A) = (\mathbf{1}, id_A, !, \pi_2)$. \square

6.4 RNN

It is important to note that while the gradient descent/backpropagation functors remain our canonical way of defining traditional learning algorithms there is plenty of structure to imagine other kinds of learning algorithms in *Learn*. In fact we will show that the recurrent neural network, a type of sequence-to-sequence network, generalizes to something called a recurrent learner on *Learn* without necessarily involving a neural network to begin with.

A recurrent neural network is a neural network defined on sequences. It consists of an input x , a state h and an output y . However, x and y are sequences and for each step in the sequence a state h is carried over from the previous step.

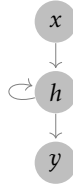


Figure 5: Recurrent neural network

In particular for a step t in the sequence h is conveying to the network at state t what the previous $t - 1$ states looked like. This state rarely captures the entire history of the previous part of the network, usually it is run through some activation function to achieve a sort of “lossy compression” of the states at previous time-steps. By this we

mean that while the network is working on an input sequence of arbitrary length the length of the state h remains fixed throughout the process hence we are embedding a sequence of vectors into a vector of fixed length.

Often when one wishes to formalize this notion of recurrence one speaks of unfolding the graph of a recurrent neural network. Given that the input and output sequences are of finite length we can instead view the graph of the network as a acyclic graph that is unfolded through the sequence where each repeated iteration of the network is known as a “cell” [3]:

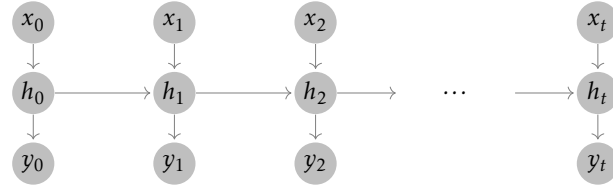


Figure 6: Unfolded recurrent neural network

The key point of a recurrent neural network is that each cell carries over a state, of some form, to the next cell. This can be viewed as a form of memory.

We can abstract the recurrent neural network to *Learn* and talk about a recurrent learner that takes sequences of objects to sequences of objects. First we need to establish what we mean by a sequence of objects in *Learn*:

Definition 13. A sequence of the object X of length n in *Learn* is the object $X^n := \prod_n X$, the n -fold Cartesian product of X with itself.

Remark 9. Note that the empty sequence of any object is $\mathbf{1}$.

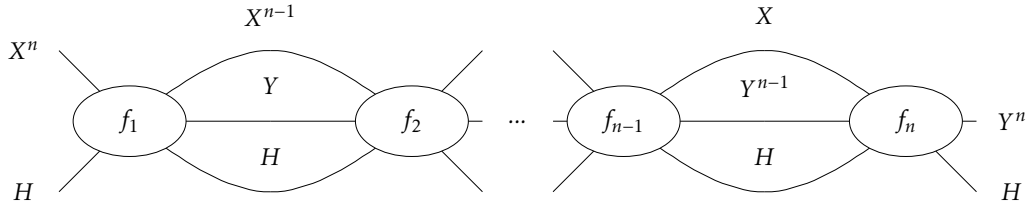
Definition 14. A recurrent learner cell is a morphism $X \times H \rightarrow Y \times H$ in *Learn*. X is known as the input, Y the output and H the state.

We can now state what a recurrent learner morphism looks like in *Learn*.

Definition 15. For each recurrent learner cell $f : X \times H \rightarrow Y \times H$ in *Learn* and $n \in \mathbb{N}$ there exists a recurrent learner $R_f^n : X^n \times H \rightarrow Y^n \times H$ defined as

$$R_f^n := f_n \circ f_{n-1} \circ \cdots \circ f_2 \circ f_1$$

or equivalently as the string diagram



where

$$\begin{aligned}
f_1 &: X^n \times H \rightarrow X^{n-1} \times Y \times H \\
f_1 &:= id_{X^{n-1}} \otimes f \\
f_i &: X^{n-i+1} \times Y^{i-1} \times H \rightarrow X^{n-i} \times Y^i \times H \\
f_i &:= (id_{X^{n-i}} \otimes b_{X, Y^{i-1}} \otimes id_H) \circ (id_{X^{n-i}} \otimes id_{Y^{i-1}} \otimes f) \\
f_n &: X \times Y^{n-1} \times H \rightarrow Y^n \times H \\
f_n &:= (id_{Y^{n-1}} \otimes f) \circ (b_{X, Y^{n-1}} \otimes id_H)
\end{aligned}$$

Remark 10. While the definition of R_f^n might look a tad complicated a closer look reveals that we are simply applying f to each component of its input in sequence while passing the state and accumulating the results.

6.5 LSTM

A commonly used learner is the LSTM, or long-short term memory learner [3]. Viewed as a neural network it is a recurrent neural network with each cell depicted as in Figure 7.

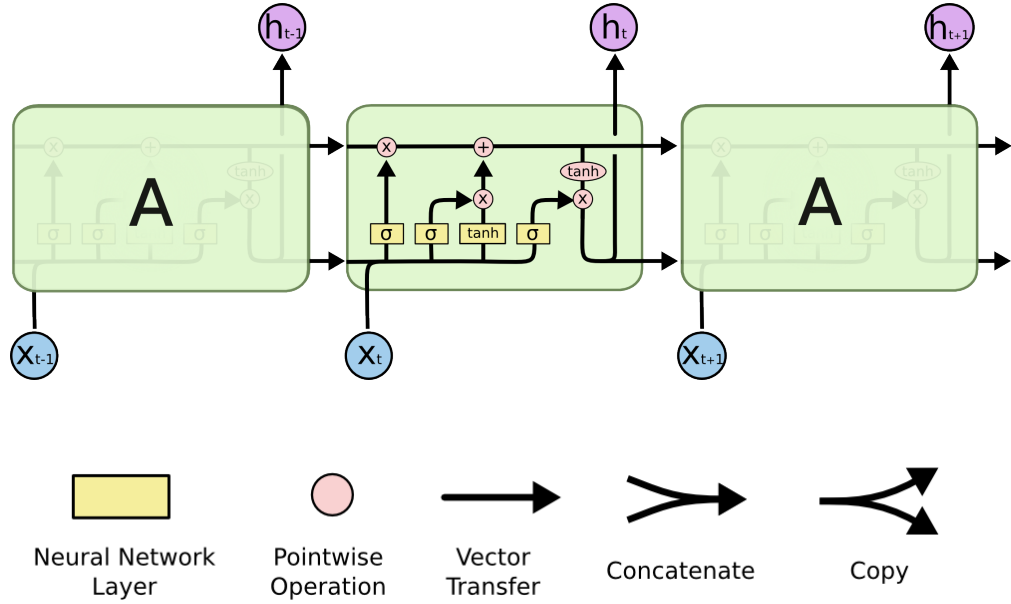


Figure 7: Diagram over LSTM from *colah's blog* [6] with permission from Christopher Olah.

With our previous endeavor of defining recurrent learners we can now reconstruct the LSTM in *Learn* as a composite morphism. We do not need to bother ourselves too

by defining a LSTM-cell $l : \mathbb{R}^a \times (\mathbb{R}^b \times \mathbb{R}^b) \rightarrow \mathbb{R}^b \times (\mathbb{R}^b \times \mathbb{R}^b)$ in *Para*. For brevity we will only present it as a string diagram:

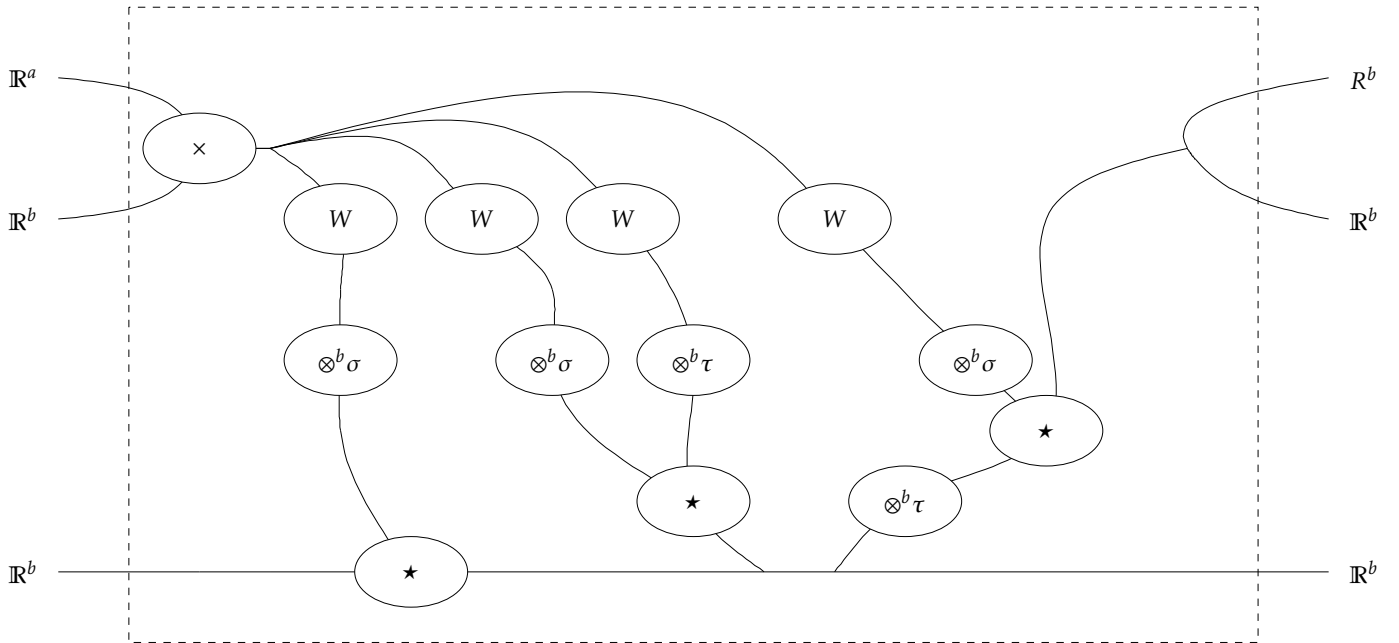


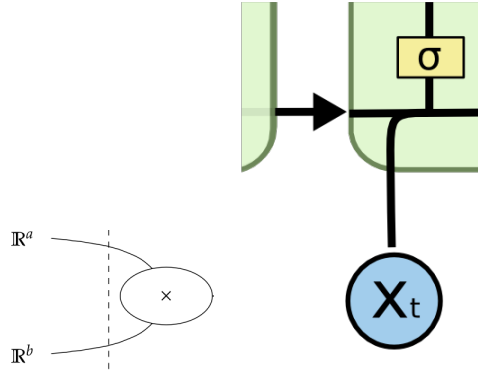
Figure 8: String diagram over an LSTM-cell

First of all there is some new notation that needs explaining. The notation $\otimes^n f$ means the morphism f tensored with itself n times. This notation applied to the activation functions σ and τ together with the morphism W makes up a *neural network layer* in Figure 7.

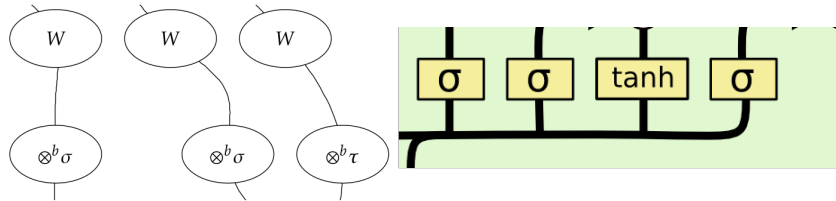
W is a morphism that takes a vector of length $a + b$, makes b copies of it and multiplies each component in each copy by a weight and sums up every copy, hence it is a morphism $W : \mathbb{R}^{a+b} \rightarrow \mathbb{R}^b$ with parameter space \mathbb{R}^{ab+b^2} . If W looks familiar it is because it is basic matrix multiplication with what is known as a weight matrix in [3] with b rows and $a + b$ columns multiplied with a vector of length $a + b$.

The morphism $\star: \mathbb{R}^b \times \mathbb{R}^b \rightarrow \mathbb{R}^b$ with trivial parameter space is pointwise multiplication of two vectors so that the i th component in the first vector is multiplied with the i th component in the second vector.

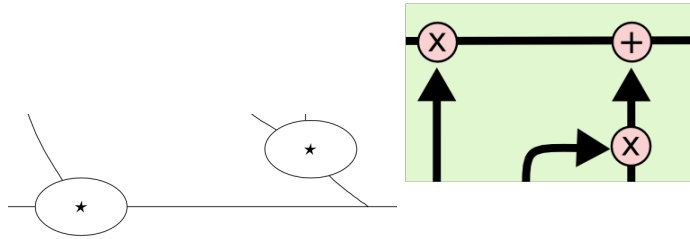
Note how the entry point of the input x_t in Figure 7. is concatenated with the entry of the state h_{t-1} , this is in our string diagram in Figure 8 represented by the monoidal product of objects which on *Para* which coincides with the Cartesian product to give us an object \mathbb{R}^{a+b} .



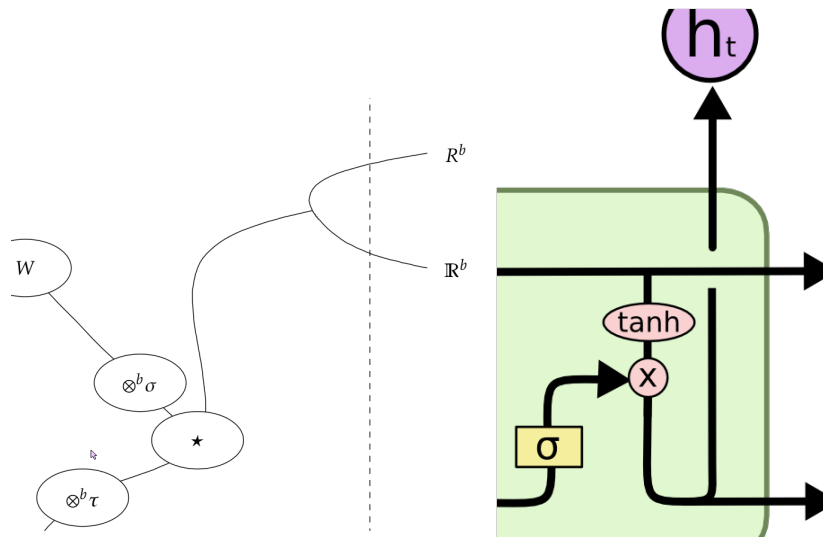
This concatenated object is then distributed across the different layers, represented by the yellow rectangles in Figure 7 and W composed with $\otimes^b \sigma$ (or $\otimes^b \tau$) in Figure 8.



After passing through the activation functions the vector of length $a + b$ is multiplied and added in various ways depicted by the multiplication and additions operations in Figure 7. eventually resulting in the state output C , a vector of length b .



Each component of C is then run through another activation function before being pointwise multiplied with another vector ultimately being duplicated into both the output and the next state.



Now our functor $L : Para \rightarrow Learn$, given some choice of learning rate ϵ , error function $e(x, y)$ and α will take this cell to a recurrent learner cell in $Learn$. Then there is, by the construction given in Definition 15, a recurrent learner $R_{L(l)}^n : \mathbb{R}^{an} \times (\mathbb{R}^b \times \mathbb{R}^b) \rightarrow \mathbb{R}^{bn} \times (\mathbb{R}^b \times \mathbb{R}^b)$ for some $n \in \mathbb{N}$.

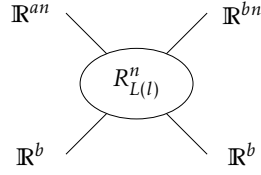


Figure 9: Recurrent learner LSTM on $Learn$

The LSTM recurrent learner demonstrates the toolbox that we have accumulated so far and the ability to reconstruct high-level learning algorithms in $Learn$ using only composable morphisms. The final composite morphism itself has the property that its parameters can be updated using gradient descent by the update function belonging to the morphism.

7 Discussion

In this thesis we have in Section 2 established what a category is in a brief overview of the required theory as well as spent some extra attention in Section 3 on the monoidal categories and how they can be represented as string diagrams.

We have then proceeded in Section 5 and 6 by following Fong *et al.* [2] in showing how the category *Para* is populated by equivalence classes of parametrized differential functions as morphisms and from there on extended into *Learn* by means of functors, determined by learning rate ϵ and error function $e(x, y)$, that take morphisms in *Para* to morphisms, also called *learners*, in *Learn*.

In the final part of Section 6 we have added to the results of Fong *et al.* [2] by re-viewing the role of the recurrent neural network and its corresponding constructions on *Learn*, the recurrent learner morphisms.

7.1 Further directions

The recurrent learner construction in Section 6 is not the most elegant construction and mostly serves as a proof-of-concept. One undertaking could be to see if there are some other, more intuitive way, of constructing a recurrent learner on *Learn* perhaps by considering a monad.

As *Learn* is a category with sets as objects there are exponential objects containing, say, all equivalence classes of learners from X to Y . A possible lane for further analysis then is to investigate whether there are morphisms between these objects perhaps to be used in applications that intend to optimize the architectures of the learners. This “meta-learning” is known as *parameter hyper-optimization* in Goodfellow *et al.* [3].

Another interesting direction to take is the relation between lambda calculus and category theory that ended up being outside the scope of this thesis. Could there possibly be some type of logic like linear logic or typed lambda calculus that *Learn* could serve as a model for? Closed Cartesian categories are models for lambda calculus and symmetric monoidal categories are models for linear logic as succinctly summarized by John Baez *et al.* in [4].

8 References

- [1] Steve Awodey, *Category Theory*, Oxford University Press Inc., 2nd Edition, 2010.
- [2] B. Fong, D. Spivak and R. Tuyeras, *Backprop as a Functor: A compositional perspective on supervised learning*, 2017, arXiv:1711.10455v2
- [3] Ian Goodfellow, Yoshua Bengio and Aaron Courville, *Deep Learning Book*, MIT Press, <http://www.deeplearningbook.org>, 2016
- [4] John C. Baez and Mike Stay, *Physics, Topology, Logic and Computation: A Rosetta Stone*, 2009, arXiv:0903.0340v3
- [5] John C. Baez, John Foley, Joseph Moeller, Blake S. Pollard, *Network Models*, 2017, arXiv:1711.00037
- [6] Christopher Olah, *Functional programming neural networks*, <http://colah.github.io/posts/2015-09-NN-Types-FP/>, Accessed 2018-07-31.
- [7] Fei Wang, Xilun Wu, Gregory Essertel, James Decker and Tiark Rompf, *Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator*, 2018, arXiv:1803.10228v1
- [8] Andre Joyal and Ross Street, The geometry of tensor calculus I, *Advances in Mathematics*, Volume 88, Issue 1, July 1991, Pages 55-112