

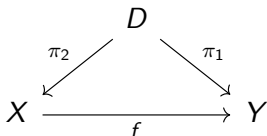
Deep Learning for the Working Category Theorist

Jade Master
University of California Riverside
jmast003@ucr.edu

February 12, 2019

- 1 Deep Learning With Commutative Diagrams
- 2 Compositional Learning Algorithms

A good intro to deep learning from a categorical perspective can be found in Jake Bian's *Deep Learning on C^∞ Manifolds* [1]. Deep learning solves the following problem.



Where $D \subseteq X \times Y$ and π_1 and π_2 are the canonical projections.

- Commuting approximately is enough. We don't want to overfit.
- any old function won't do! f should *extrapolate* the data from D .

To get more of this extrapolating flavor we equip Y with a metric (maybe without some axioms)

$$\mathcal{L}: Y \times Y \rightarrow \mathbb{R}_+$$

and measure the error of f via

$$E(f) = \sum_{(x,y) \in D} \mathcal{L}(f(x), y)$$

We also require f to be within some restricted class of functions between X and Y (e.g. linear, smooth). This allows us to impose some external assumptions about the way the relationship between X and Y should behave.

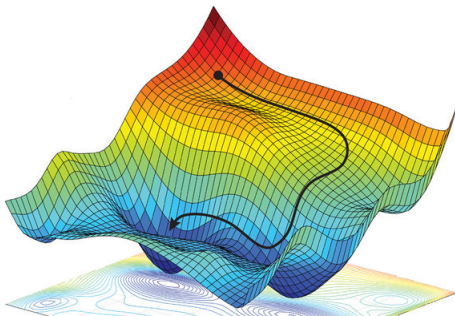
For example we can require f to be smooth, so that f represents some physical, differentiable relationship. In this case, minimizing E can be framed as a problem in variational calculus. Physicists and differential geometers know how to solve this using the Euler-Lagrange equation.

Unfortunately we are interested in the case when D and the dimension of X are very large. So this would require solving a massive partial differential equation. This is not practical.

Let's stick with "smooth" as our class of functions. To make things more manageable let's consider a family of functions parameterized by some manifold Σ . Our class of functions can be summarized by a map

$$\alpha: \Sigma \times X \rightarrow Y$$

which is smooth in each variable. The key idea of deep learning is minimize the error E on element of D at a time obtaining a sequence of functions which converges to the global minimum.



That is, find a path on Σ ,

$$\gamma: [0, 1] \rightarrow \Sigma$$

such that the induced 1-parameter family of functions

$$f_t = \alpha(\gamma(t), -): X \rightarrow Y$$

converges to a minimum of the functional E . One way to obtain such a curve is to

- Evaluate the derivatives of E , giving a vector field dE on Σ
- Numerically integrate $-dE$ starting from an initial parameter λ_0 .
Because we're doing this numerically we'll use a finite sequence of functions rather than a smooth family.

Let $\mu: T\Sigma \rightarrow \Sigma$ be the integrator. For a fixed step size this function advances a parameter in the direction of a given tangent vector.

$$\begin{array}{ccccccc}
& & \{x_0\} & & a & & \{y_0\} \\
& & \downarrow & & & & \downarrow \\
\{*\} & \xrightarrow{\lambda_0} & \Sigma & \xhookrightarrow{i_\Sigma} & \Sigma \times X & \xrightarrow{\alpha} & Y \xhookrightarrow{i_Y} Y \times Y \xrightarrow{\mathcal{L}} \mathbb{R}^+ \\
& & & & & & \downarrow d\mathcal{L} \\
T^*\Sigma & \xleftarrow{(i_Y \circ \alpha \circ i_\Sigma)^*} & T^*(Y \times Y) & & & & \\
\downarrow \mu & & \{x_1\} & & & & \{y_1\} \\
& & \downarrow & & & & \downarrow \\
\Sigma & \xhookrightarrow{i_\Sigma} & \Sigma \times X & \xrightarrow{\alpha} & Y & \xhookrightarrow{i_Y} & Y \times Y \xrightarrow{\mathcal{L}} \mathbb{R}^+ \\
& & & & & & \downarrow d\mathcal{L} \\
T^*\Sigma & \xleftarrow{(i_Y \circ \alpha \circ i_\Sigma)^*} & T^*(Y \times Y) & & & & \\
\downarrow \mu & & \{x_2\} & & & & \{y_2\} \\
& & \downarrow & & & & \downarrow \\
\Sigma & \xhookrightarrow{i_\Sigma} & \Sigma \times X & \xrightarrow{\alpha} & Y & \xhookrightarrow{i_Y} & Y \times Y \xrightarrow{\mathcal{L}} \mathbb{R}^+ \\
& & & & & & \downarrow d\mathcal{L} \\
& & & & & & \dots
\end{array}$$

Neural networks are a special case of this. Let X and Y be vector spaces with dimension m and n respectively. It's natural to choose linear maps as our class of functions. This gives "the line of best fit" which is a relatively crude tool. Instead we choose an intermediate vector space V with dimension k and a nonlinear map $\sigma: V \rightarrow V$. The class of maps we will consider will be of the form

$$X \xrightarrow{a_0} V \xrightarrow{\sigma} V \xrightarrow{a_1} Y$$

where f and g are linear maps. In this case

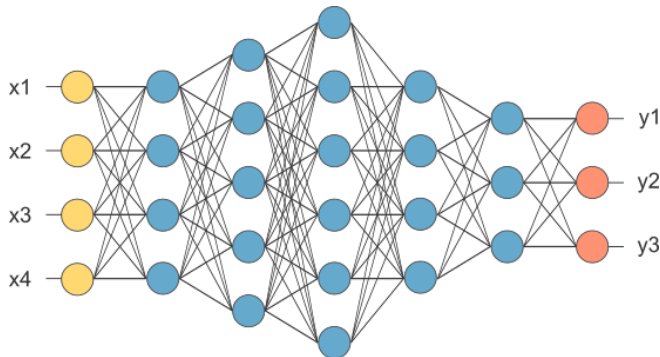
$$\Sigma = GL(m, k) \times GL(k, n)$$

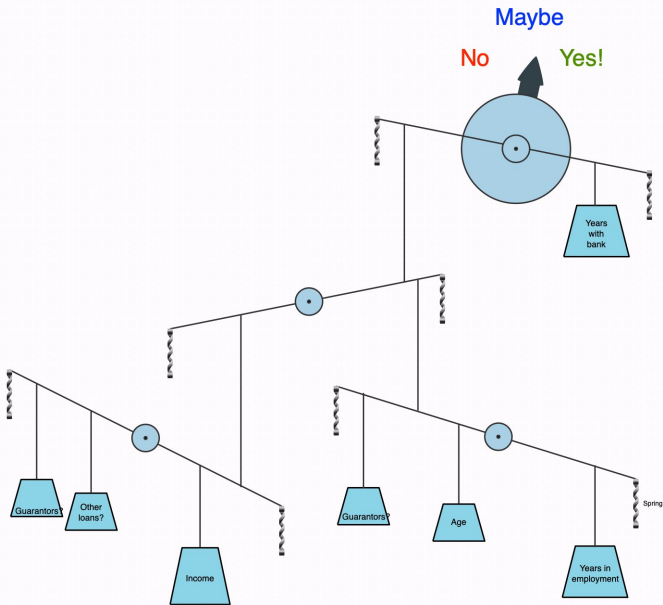
This is called a **one layer neural network**.

This can be generalized to multiple layers. Indeed, choosing a finite sequence of vector spaces and nonlinear maps $\{V_i, \sigma_i\}$ you can consider functions of the form

$$X \xrightarrow{a_0} V_0 \xrightarrow{\sigma_0} V_0 \xrightarrow{a_1} V_1 \dots V_n \xrightarrow{\sigma_n} V_n \xrightarrow{a_n} Y$$

for your training algorithm. This is a **multiple layer neural network**.





Backprop as a Functor introduces a categorical framework for building and training large neural networks as a sequence of smaller networks. First we need to abstractify the above discussion.

Definition

Let A and B be sets. A **supervised learning algorithm** from A to B is tuple (P, I, U, r) where P is a set and I , U and r are functions of the following form.

$$I: P \times A \rightarrow B$$

$$U: P \times A \times B \rightarrow P$$

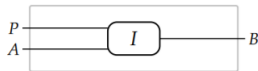
$$r: P \times A \times B \rightarrow A$$

The request function seems mysterious at first. It allows you to compose learning algorithms. Given training data $(x, y) \in X \times Y$ and $(y, z) \in Y \times Z$ we need to synthesize this into a pair $(x, z) \in X \times Z$. The request function does this by allowing the data to flow backwards. Given a comparison between the desired input and output and the request function tells you what the input *should have been* to get a better result.

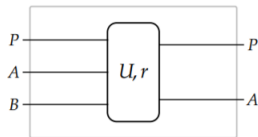
Composition of Learning Algorithms

Given learners $(P, I, U, r): A \rightarrow B$ and $(Q, J, V, s): B \rightarrow C$ we construct a new learner from A to C with parameter set $P \times Q$. Let's use string diagrams!

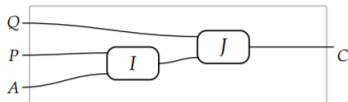
We can draw implement functions like



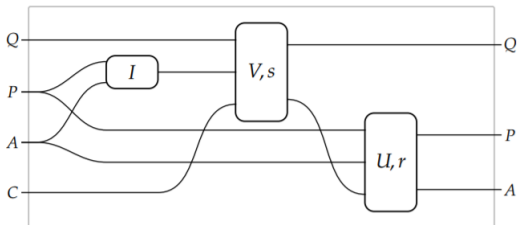
and the update request functions as



Composition of implementations is straightforward



but composition of update and request is a little more complicated.



Two learners are **equivalent** if there is a bijection between their sets of parameters commuting with the implement, update and request functions

Definition

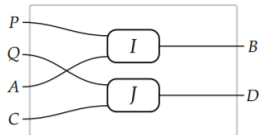
Let Learn be the category where the objects are sets, the morphisms are equivalence classes of learners, and composition is defined as above.

Theorem

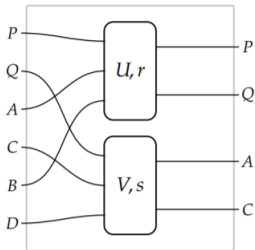
Learn is a symmetric monoidal category.

On objects the monoidal product is given by cartesian product. The monoidal product is given on morphisms as follows.

For implememt functions



and for update and request



So far these learning algorithms are *too* general and abstract. We can impose extra structure on them via symmetric monoidal functors.

Definition

Let Para be the category where

- objects are Euclidean spaces and,
- a morphism from \mathbb{R}^n to \mathbb{R}^m is a differentiable map $f: P \times \mathbb{R}^n \rightarrow \mathbb{R}^m$ up to equivalence.

The following theorem gives a symmetric monoidal functor which imposes the structure of a gradient descent learning algorithm on every morphism in Para .

Theorem

Fix a real number $h > 0$ and a differentiable error function

$$e: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$$

such that $\frac{\partial e}{\partial x}(z, -)$ is invertible for all z in \mathbb{R} . Then there is a symmetric monoidal functor

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

which is the identity on objects and sends a parameterized function $I: P \times A \rightarrow B$ the learner (P, I, U_I, r_I) defined by

$$U_I(p, a, b) := p - h \nabla_p E_I(p, a, b)$$

and

$$r_I := f_a(\nabla_a E_I(p, a, b))$$

where $E_I(p, a, b) := \sum_j e(I(p, a)_j, b_j)$ and f_a is the component-wise application of the inverse to $\frac{\partial e}{\partial x}(a_i, -)$ for each i . Here i ranges over the dimension of A and j ranges over the dimension of B .

Remark: Let's compare to what we had before: $E(p, a, b) = \mathcal{L}(I(a, p), b)$, U_I is the composite up to the second Σ , and the request function is not present.

$$\begin{array}{ccccccc}
 & & \{x_0\} & & a & & \{y_0\} \\
 & & \downarrow & & & & \downarrow \\
 \{*\} & \xrightarrow{\lambda_0} & \Sigma & \xleftarrow{i_\Sigma} & \Sigma \times X & \xrightarrow{\alpha} & Y & \xleftarrow{i_Y} & Y \times Y & \xrightarrow{\mathcal{L}} & \mathbb{R}^+ \\
 & & & & & & & & \downarrow d\mathcal{L} & & \\
 T^*\Sigma & \xleftarrow{(i_Y \circ \alpha \circ i_\Sigma)^*} & T^*(Y \times Y) & & & & & & & & \\
 \downarrow \mu & & \{x_1\} & & & & \{y_1\} & & & & \\
 & & \downarrow & & & & \downarrow & & & & \\
 \Sigma & \xleftarrow{i_\Sigma} & \Sigma \times X & \xrightarrow{\alpha} & Y & \xleftarrow{i_Y} & Y \times Y & \xrightarrow{\mathcal{L}} & \mathbb{R}^+ \\
 & & & & & & & & \downarrow d\mathcal{L} & & \\
 T^*\Sigma & \xleftarrow{(i_Y \circ \alpha \circ i_\Sigma)^*} & T^*(Y \times Y) & & & & & & & & \\
 \downarrow \mu & & \{x_2\} & & & & \{y_2\} & & & & \\
 & & \downarrow & & & & \downarrow & & & & \\
 \Sigma & \xleftarrow{i_\Sigma} & \Sigma \times X & \xrightarrow{\alpha} & Y & \xleftarrow{i_Y} & Y \times Y & \xrightarrow{\mathcal{L}} & \mathbb{R}^+ \\
 & & & & & & & & \downarrow d\mathcal{L} & & \\
 & & & & & & & & \dots & &
 \end{array}$$

We define a category where the morphisms gives the shape of a neural network.

Definition

Let List be a category where

- the objects are natural numbers and,
- a morphism $m \rightarrow n$ is a list of natural numbers (a_0, a_1, \dots, a_k) with $a_0 = m$ and $a_k = n$

composition is given by concatenation and this category is symmetric monoidal using the $+$ operation in \mathbb{N} .

The idea is that a morphism (a_0, a_1, \dots, a_k) represents a neural network with $k - 1$ hidden layers and number of neurons given by the a_i .

Theorem

Let $\sigma: \mathbb{R} \rightarrow \mathbb{R}$ be a differentiable function. Then there is a symmetric monoidal functor

$$F: \text{List} \rightarrow \text{Para}$$

which

- sends a natural number n to vector space \mathbb{R}^n and,
- sends a list (a_0, a_1, \dots, a_k) to the parameterized map

$$\mathbb{R}^m \times GL(a_1, a_2) \times \dots \times GL(a_{k-2}, a_{k-1}) \rightarrow \mathbb{R}^n$$

which is the alternating composite of the linear maps given by the parameters and the extension of σ to the corresponding vector spaces.

Note that this is the same description described in *Deep Learning on C^∞ Manifolds*

What's all this good for?

Now we have a machine for building neural networks compositionally.

$$\text{List} \xrightarrow{F} \text{Para} \xrightarrow{L} \text{Learn}$$





This is known...but now we've generalized it and described it with category theory!

If you're one of those folks that loves string diagrams then you're in luck. This theorem gives a framework for building neural networks, neuron by neuron, using string diagrams.

First, fixing a step size and an error function we can use the passage

$$\mathbf{FVect} \rightarrow \mathbf{Para} \rightarrow \mathbf{Learn}$$

to induce a bimonoid structure on \mathbf{Learn} . For example using quadratic error gives the following.

	Implementation	Request
<p>Multiplication, μ</p> <p>$(1, I_\mu, !, r_\mu)$</p> 	$I_\mu: A \times A \longrightarrow A;$ $(a_1, a_2) \longmapsto a_1 + a_2$	$r_\mu: (A \times A) \times A \longrightarrow A \times A;$ $(a_1, a_2, a_3) \longmapsto (a_3 - a_2, a_3 - a_2)$
<p>Unit, η</p> <p>$(1, I_\eta, !, r_\eta)$</p> 	$I_\eta: \mathbb{R}^0 \longrightarrow A;$ $0 \longmapsto 0$	$r_\eta: A \longrightarrow \mathbb{R}^0;$ $a \longmapsto 0$
<p>Comultiplication, δ</p> <p>$(1, I_\delta, !, r_\delta)$</p> 	$I_\delta: A \longrightarrow A \times A;$ $a \longmapsto (a, a)$	$r_\delta: A \times (A \times A) \longrightarrow A;$ $(a_1, a_2, a_3) \longmapsto a_2 + a_3 - a_1$
<p>Counit, ϵ</p> <p>$(1, I_\epsilon, !, r_\epsilon)$</p> 	$I_\epsilon: A \longrightarrow \mathbb{R}^0;$ $a \longmapsto 0$	$r_\epsilon: A \longrightarrow A;$ $a \longmapsto 0$

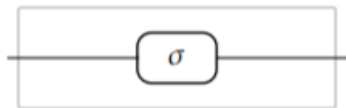
There's a scalar multiplication learner $\lambda: \mathbb{R} \rightarrow \mathbb{R}$



a bias learner $\beta: 1 \rightarrow \mathbb{R}$,

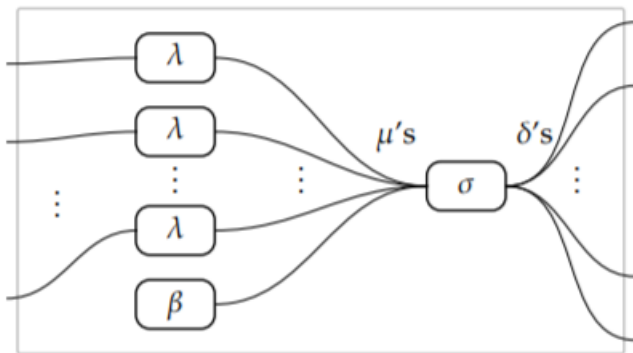


and an activation learner $\sigma: \mathbb{R} \rightarrow \mathbb{R}$



I didn't describe biases in the previous theorem. But they can be added and adds a parameter of constant weights to your approximation function.

Layers of your neural network can be built as the following:



Conclusion

- Some learning algorithms like regression analysis don't learn one data point at a time. Regression takes the whole data-set into account at each successive approximation. It would be nice to have a model which takes this into account.
- Can this be made more general?
- The data of a neural network is a representation of a linear Quiver. Can we use representation theorems to classify them?

References



Jake Bian (2017)

Deep Learning on C^∞ Manifolds. Available as
<http://jake.run/pdf/geom-dl.pdf>



Brendan Fong, David I. Spivak, and Rémy Tuyéras (2017)

Backprop as a Functor: A compositional perspective on supervised learning.
Available as <https://arxiv.org/abs/1711.10455>