Backprop as Functor

Brendan Fong, with David Spivak, Rémy Tuyéras

SYCO 1
University of Birmingham
21 September 2018

Consider the function:

$$\text{Cat?: Pictures} = \mathbb{R}^{100\times 100\times 3} \longrightarrow \langle \texttt{cat}, \texttt{not\_cat} \rangle = \mathbb{R}^2$$

$\longmapsto 1.00|\texttt{cat}\rangle + 0.00|\texttt{not\_cat}\rangle$

$\longmapsto 0.12|\texttt{cat}\rangle + 0.95|\texttt{not\_cat}\rangle$

$\longmapsto 1.00|\texttt{cat}\rangle + 1.00|\texttt{not\_cat}\rangle$

How do we program it?

Outline

I. Supervised Learning, Compositionally
II. Specifying Parametrised Functions
III. Backprop: Updates and Requests via Gradient Descent

I. Supervised Learning, Compositionally

Goal: learn a function from examples

Fix sets A, B. For all $f\colon A \to B$, use pairs $(a, f(a))$ to *approximate* f.

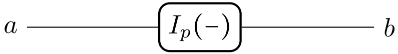
Method: use the following data

Hypothesis set: P

Implementation function: $I\colon P \times A \to B$

Update function $U\colon P \times A \times B \to P$

Request function $r\colon P \times A \times B \to A$



A **learner** $A \to B$ is a tuple* (P, I, U, r).

---

*actually an equivalence class.

Goal: learn a function from examples

Fix sets A, B. For all $f\colon A \to B$, use pairs $(a, f(a))$ to *approximate* f.

Method: use the following data

Hypothesis set: $P \leftsquigarrow$ Strategies

Implementation function: $I\colon P \times A \to B \leftsquigarrow$ Play

Update function $U\colon P \times A \times B \to P \leftsquigarrow$ Equilibrium

Request function $r\colon P \times A \times B \to A \leftsquigarrow$ Coutility

$$a \;\longrightarrow\; \boxed{I_p(-)} \;\longrightarrow\; b$$

A **learner** $A \to B$ is a tuple[*] (P, I, U, r).

[*]actually an equivalence class.

Goal: learn a function from examples

Fix sets A, B. For all $f\colon A \to B$, use pairs $(a, f(a))$ to *approximate* f.

Method: use the following data

Hypothesis set: P

Implementation function: $I\colon P \times A \to B$

Update function $U\colon P \times A \times B \to P$

Request function $r\colon P \times A \times B \to A$

$$a \longrightarrow \boxed{I_p(-)} \longrightarrow b$$

A **learner** $A \to B$ is a tuple[*] (P, I, U, r).

[*]actually an equivalence class.

The symmetric monoidal category Learn has

objects: sets

morphisms: learners (P, I, U, r).

*How does **composition** work? Suppose we have a pair of learners:*

$$A \xrightarrow{(P,I,U,r)} B \xrightarrow{(Q,J,V,s)} C.$$

How does **composition** *work? Suppose we have a pair of learners:*

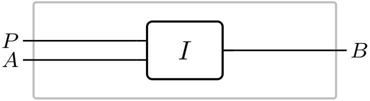
$$A \xrightarrow{(P,I,U,r)} B \xrightarrow{(Q,J,V,s)} C.$$

The new parameter space is just the product $Q \times P$.

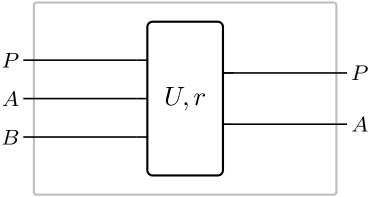
*How does **composition** work? Suppose we have a pair of learners:*

$$A \xrightarrow{(P,I,U,r)} B \xrightarrow{(Q,J,V,s)} C.$$

Let's represent our learners with string diagrams:



$$I\colon P \times A \longrightarrow B$$

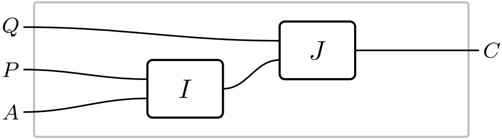


$$(U,r)\colon P \times A \times B \longrightarrow P \times A$$

*How does **composition** work? Suppose we have a pair of learners:*

$$A \xrightarrow{(P,I,U,r)} B \xrightarrow{(Q,J,V,s)} C.$$

Composing implementation functions is straightforward:

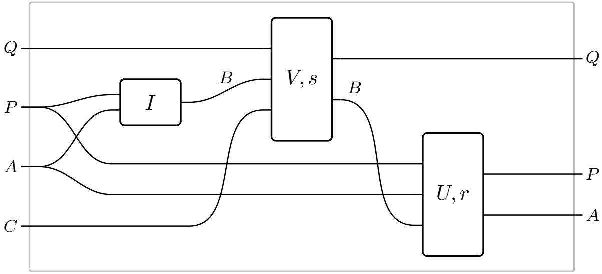


$$(q, p, a) \longmapsto J(q, I(p, a))$$

*How does **composition** work? Suppose we have a pair of learners:*

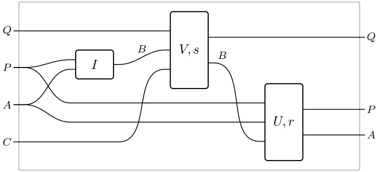
$$A \xrightarrow{(P,I,U,r)} B \xrightarrow{(Q,J,V,s)} C.$$

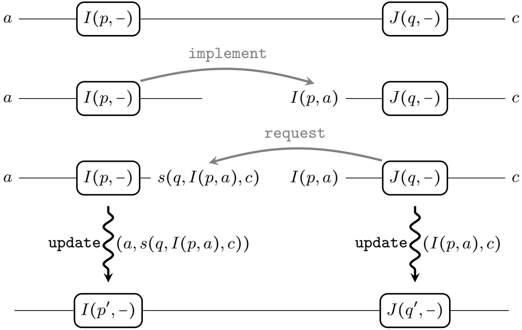
Composing update/request functions is more complicated:



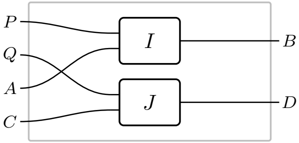
$$(q, p, a, c) \longmapsto \Big(V\big(q, I(p,a), c\big), U\big(p, a, s(q, I(p,a), c)\big), r\big(p, a, s(q, I(p,a), c)\big)\Big).$$

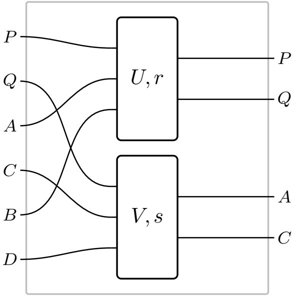
Key idea: composition creates local training data.





The **monoidal product** of $(P, I, U, r)\colon A \to B$ and $(Q, J, V, s)\colon C \to D$ is given by





A compositional framework for supervised learning:

- **Learning**: parameter updates.
- **Supervised**: training is by (input, output) pairs.
- **Compositional**: we can build new learners from old.

A compositional framework for supervised learning:

Learning: parameter updates.

Supervised: training is by (input, output) pairs.

Compositional: we can build new learners from old.

But how can we explicitly construct a learner?

II. Specifying Parametrised Functions

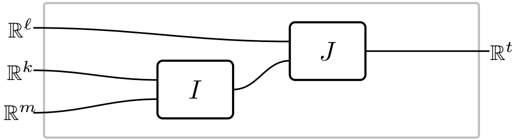
The prop Para has

objects: natural numbers

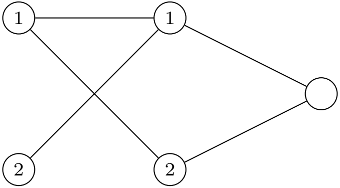
morphisms $m \to n$: *differentiable* functions

$$I : \mathbb{R}^k \times \mathbb{R}^m \to \mathbb{R}^n.$$

Composition is as for implementation functions in Learn:



Neural networks (sequences of bipartite graphs) are a compositional, combinatorial language for specifying differentiable parametrised functions.



$$
\begin{aligned}
I\colon (\mathbb{R}^5 \times \mathbb{R}^3) \times \mathbb{R}^2 &\longrightarrow \mathbb{R}; \\
(p, q, a) &\longmapsto \sigma\big(q_1\sigma(p_{11}a_1 + p_{12}a_2 + p_{1b}) + q_2\sigma(p_{21}a_1 + p_{2b}) + q_b\big).
\end{aligned}
$$

where $\sigma\colon \mathbb{R} \to \mathbb{R}$ is a differentiable function known as the activation.

The prop NNet has

objects: natural numbers.

morphisms $m \to n$: neural networks with m inputs and n outputs.

composition: concatenation of neural networks.

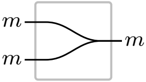
Theorem

A differentiable function $\sigma\colon \mathbb{R} \to \mathbb{R}$ defines a prop functor

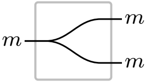
$$I_\sigma\colon \mathsf{NNet} \longrightarrow \mathsf{Para}.$$

Differentiable parametrised functions can also be constructed using string diagrams in Para.

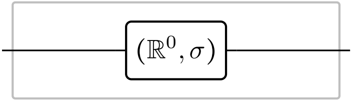
The image of NNet under I_σ is contained in the composite of:



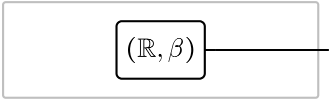
$$\mu(a_1, a_2) = a_1 + a_2$$



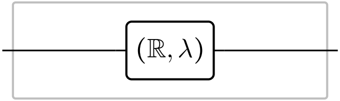
$$\delta(a) = (a, a)$$



$$\sigma(x) = \sigma(x)$$

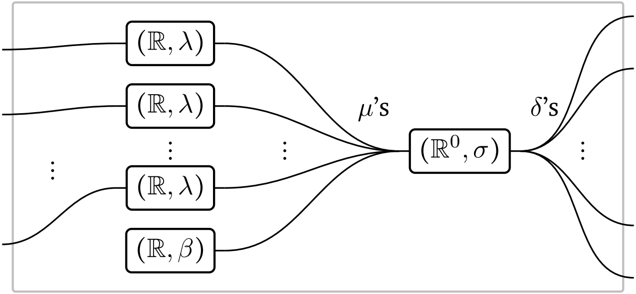


$$\beta(w) = w$$



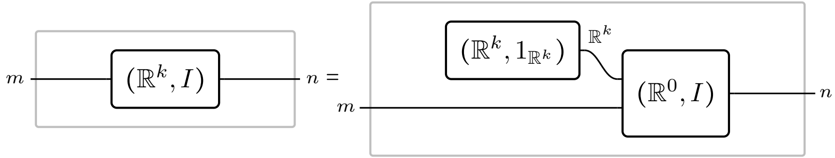
$$\lambda(w, x) = wx$$

Differentiable parametrised functions can also be constructed using string diagrams in Para.

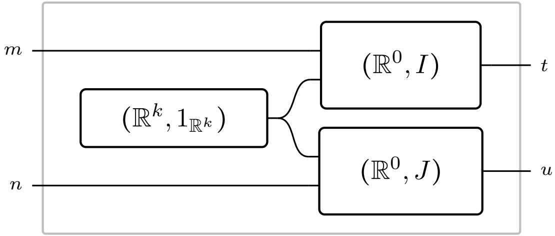


Weight-tying is a technique that identifies parameters that describe the same structure.

We factorise.



Then copy.



III. Backprop: Updates and Requests via Gradient Descent

Theorem

Fix $\epsilon > 0$, $e\colon \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ such that $\frac{\partial e}{\partial x}(x_0, -)\colon \mathbb{R} \to \mathbb{R}$ has inverse h_{x_0} for each x_0.

There is a faithful, injective-on-objects, strong symmetric monoidal functor

$$L_{\epsilon,e}\colon \mathsf{Para} \longrightarrow \mathsf{Learn}$$

sending each object m to $\mathbb{R}^m$, and each morphism $(\mathbb{R}^k, I)\colon m \to n$ to the learner $(\mathbb{R}^k, I, U_I, r_I)\colon \mathbb{R}^m \to \mathbb{R}^n$ defined by

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

$$r_I(p,a,b) = h_a\Big(\nabla_a E_I(p,a,b)\Big),$$

Here $E_I(p,a,b) = \sum_i e(I(p,a)_i, b_i)$ and h_a denotes component-wise application of h_{a_i}.

Let e be the *quadratic error* $\mathsf{quad}(x, y) = \frac{1}{2}(x - y)^2$.

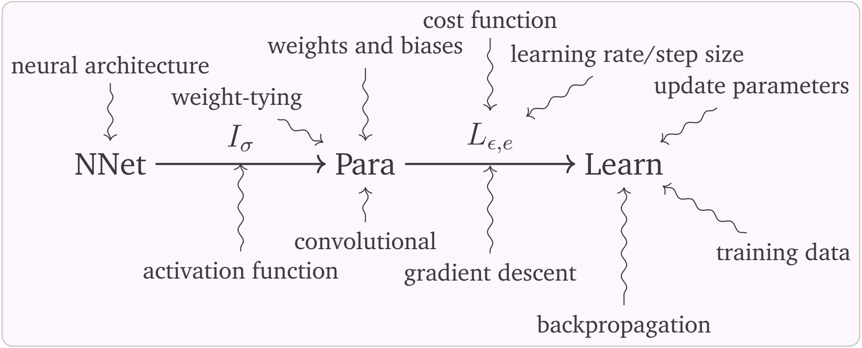
Corollary
For every $\epsilon > 0$, there is a strong symmetric monoidal functor

$$L_{\epsilon,\mathsf{quad}} \colon \mathsf{Para} \longrightarrow \mathsf{Learn}$$

sending $(\mathbb{R}^k, I) \colon m \to n$ to the learner $(\mathbb{R}^k, I, U_I, r_I) \colon \mathbb{R}^m \to \mathbb{R}^n$ defined by

$$U_I(p, a, b)_k = p_k - \epsilon \sum_j (I_j(p, a) - b_j) \frac{\partial I_j}{\partial p_k}$$

$$r_I(p, a, b)_i = a_i - \sum_j (I_j(p, a) - b_j) \frac{\partial I_j}{\partial a_i}.$$



cost function
weights and biases
learning rate/step size
neural architecture
update parameters
weight-tying
I_σ
$L_{\epsilon,e}$
NNet
Para
Learn
convolutional
activation function
gradient descent
training data
backpropagation