Backprop as Functor
Brendan Fong, with David Spivak, Rémy Tuyéras

SYCO 1
University of Birmingham
21 September 2018
Consider the function:

\[
\text{Cat?}: \text{Pictures} = \mathbb{R}^{100 \times 100 \times 3} \rightarrow \langle \text{cat, not\_cat} \rangle = \mathbb{R}^2
\]

\[\rightarrow 1.00|\text{cat}\rangle + 0.00|\text{not\_cat}\rangle\]

\[\rightarrow 0.12|\text{cat}\rangle + 0.95|\text{not\_cat}\rangle\]

\[\rightarrow 1.00|\text{cat}\rangle + 1.00|\text{not\_cat}\rangle\]

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 $f: A \rightarrow B$, use pairs $(a, f(a))$ to approximate $f$.

Method: use the following data

- Hypothesis set: $P$
- Implementation function: $I: P \times A \rightarrow B$
- Update function $U: P \times A \times B \rightarrow P$
- Request function $r: P \times A \times B \rightarrow A$

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

---

*actually an equivalence class.
Goal: learn a function from examples  
Fix sets $A$, $B$. For all $f : A \rightarrow B$, use pairs $(a, f(a))$ to approximate $f$.

Method: use the following data

Hypothesis set: $P \leadsto$ Strategies  
Implementation function: $I : P \times A \rightarrow B \leadsto$ Play  
Update function $U : P \times A \times B \rightarrow P \leadsto$ Equilibrium  
Request function $r : P \times A \times B \rightarrow A \leadsto$ Coutility

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

*actually an equivalence class.
Goal: learn a function from examples
Fix sets $A$, $B$. For all $f: A \rightarrow B$, use pairs $(a, f(a))$ to approximate $f$.

Method: use the following data

Hypothesis set: $P$
Implementation function: $I: P \times A \rightarrow B$
Update function $U: P \times A \times B \rightarrow P$
Request function $r: P \times A \times B \rightarrow A$

\[
\begin{array}{c}
\text{a} \quad \overline{I_p(-)} \quad \text{b}
\end{array}
\]

A learner $A \rightarrow B$ 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 \textit{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:

\[
I: P \times A \rightarrow B
\]

\[
(U,r): P \times A \times B \rightarrow P \times A
\]
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:

\[
(q, p, a) \mapsto J(q, I(p, a))
\]
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:

\[
(q,p,a,c) \rightarrow \left( V(q, I(p,a), c), U(p,a, s(q, I(p,a), c)), r(p,a, s(q, I(p,a), c)) \right).
\]
Key idea: composition creates local training data.
The **monoidal product** of \((P, I, U, r): A \to B\) and \((Q, J, V, s): C \to D\) 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.

\[
I : (\mathbb{R}^5 \times \mathbb{R}^3) \times \mathbb{R}^2 \longrightarrow \mathbb{R}; \\
(p, q, a) \longmapsto \sigma(q_1 \sigma(p_{11}a_1 + p_{12}a_2 + p_{1b}) + q_2 \sigma(p_{21}a_1 + p_{2b}) + q_b).
\]

where \( \sigma : \mathbb{R} \rightarrow \mathbb{R} \) 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 $\sigma: \mathbb{R} \to \mathbb{R}$ defines a prop functor

$I_\sigma: \text{NNet} \longrightarrow \text{Para}.$
Differentiable parametrised functions can also be constructed using string diagrams in Para.

The image of NNet under $I_\sigma$ 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.

$$m \xrightarrow{\mathbb{R}^k, I} n = m \xrightarrow{\mathbb{R}^k} n$$

Then copy.

$$m \xrightarrow{\mathbb{R}^k, 1_{\mathbb{R}^k}} n \xrightarrow{\mathbb{R}^0, I} (\mathbb{R}^0, J) \xrightarrow{t, u}$$
III. Backprop: Updates and Requests via Gradient Descent
Theorem
Fix $\epsilon > 0$, $e: \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ such that $\frac{\partial e}{\partial x}(x_0, -): \mathbb{R} \to \mathbb{R}$ has inverse $h_{x_0}$ for each $x_0$.

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

$$L_{\epsilon, e}: \text{Para} \longrightarrow \text{Learn}$$

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

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

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

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 $\text{quad}(x, y) = \frac{1}{2} (x - y)^2$.

**Corollary**

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

$$L_{\epsilon, \text{quad}}: \text{Para} \longrightarrow \text{Learn}$$

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

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

$$r_I(p, a, b)_i = a_i - \sum_j (I_j(p, a) - b_j) \frac{\partial I_j}{\partial a_i}.$$
neural architecture → NNet → Para → Learn

weights and biases
weight-tying
$\sigma$
activation function
convolutional
$\epsilon, \epsilon$
cost function
learning rate/step size
update parameters
training data
gradient descent
backpropagation
COMPOSITIONALITY


www.compositionality-journal.org

Steering Board
John Baez
Bob Coecke
Kathryn Hess
Steve Lack
Valeria de Paiva

Editors
Corina Cirstea
Ross Duncan
Andréě Ehresmann
Tobias Fritz
Neil Ghani

Dan Ghica
Jeremy Gibbons
Nick Gurski
Helle Hvid Hansen
Chris Heunen
Aleks Kissinger

Joachim Kock
Martha Lewis
Samuel Mimram
Simona Paoli
Dusko Pavlovic
Christian Retoré

Mehroosh Sadriadeh
Peter Selinger
Paweł Sobocinski
David Spivak
Jamie Vicary
Simon Willerton