Datatype led Deep Learning

Aloïs Brunel
Deepomatic
alois@deepomatic.com

When differential programming did not exist, the world was a scary place

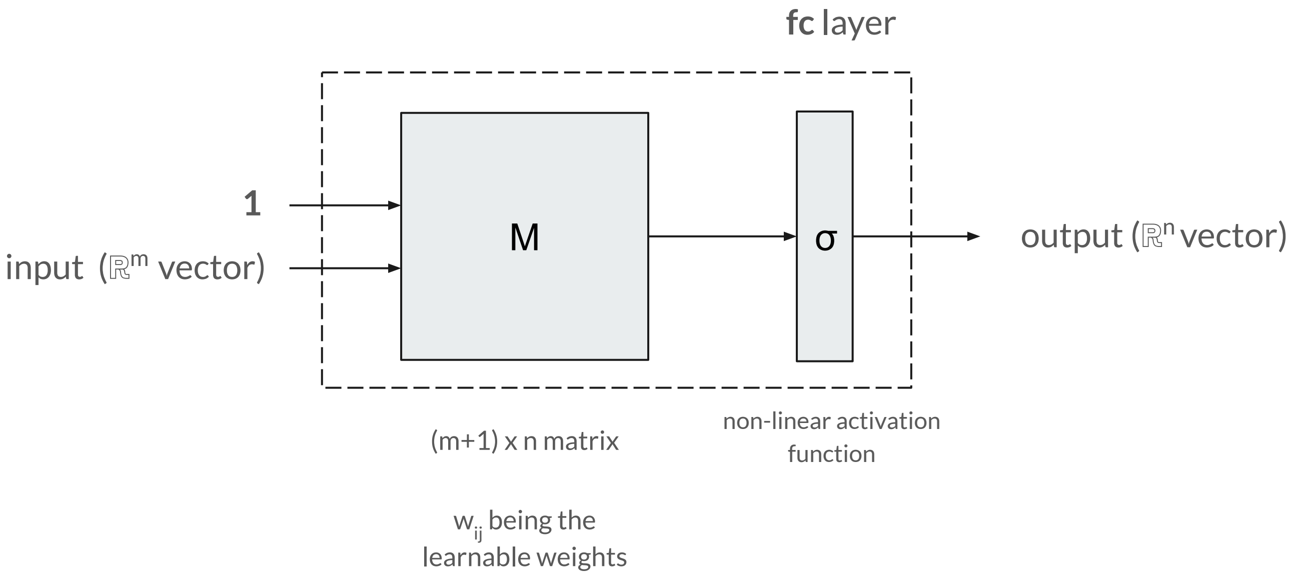
What is a neural network?

A composition tree of differentiable operators on finite-dimensional tensors.

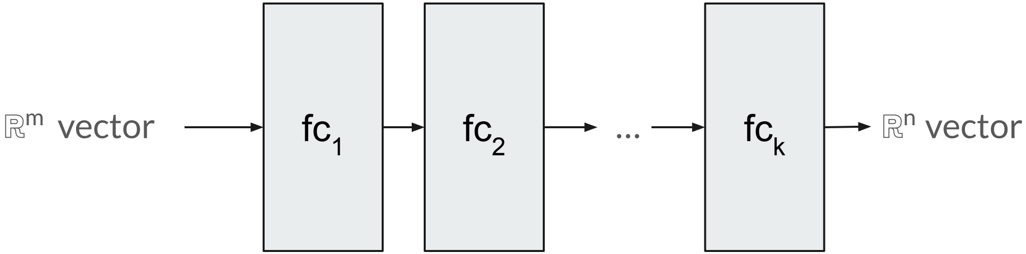
$$A : (\underbrace{\mathbb{R}^N}_{\text{Learnable parameters}} \otimes \underbrace{(\mathbb{R}^{n1} \otimes \ldots \otimes \mathbb{R}^{nk})}_{\text{Input}}) \longrightarrow \underbrace{\mathbb{R}^{m1} \otimes \ldots \otimes \mathbb{R}^{mk}}_{\text{output}}$$

A program that implements a differentiable function, whose learning weights can be optimized through (typically) back-propagation. Learnable parameters are “free variables” substituted by values that are tuned over time as the model is trained.

Good Old Perceptron

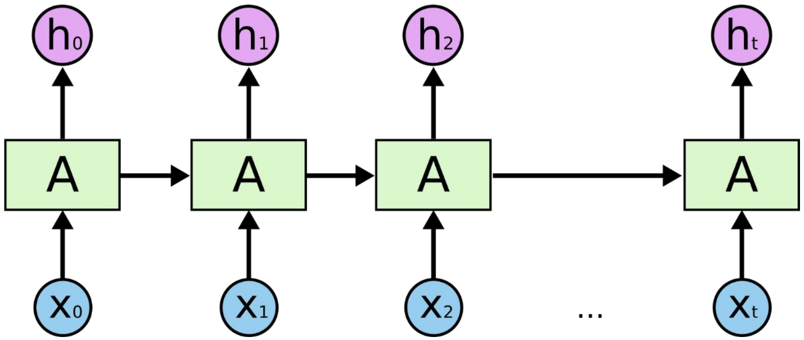


Feed forward neural network



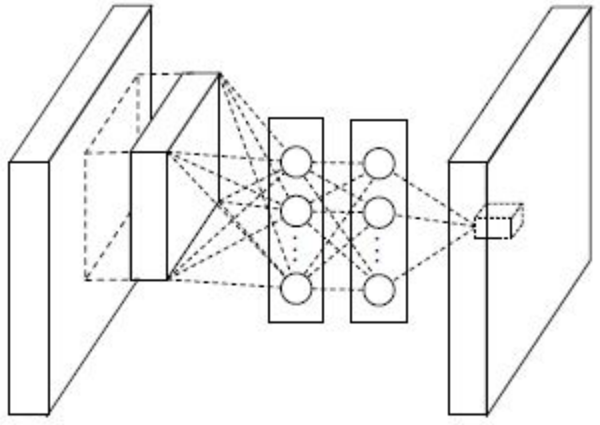
There is a functional flavor to NN architectures

The functional flavor of NN architectures
Example #1: RNN

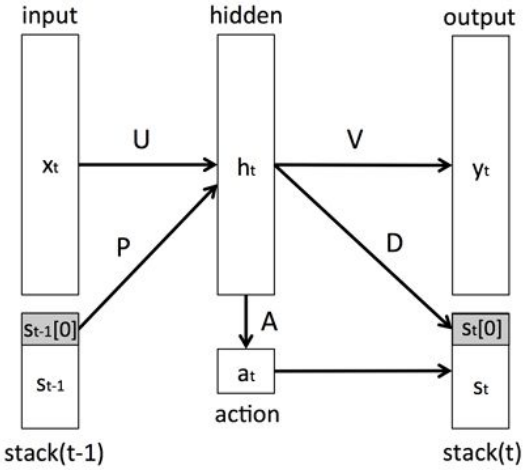



```
let (h0, s1) = A x0 s0 in
 let (h1, s2) = A x1 s1 in
  let (h2, s3) = A x2 s2 in
    ...
      let (ht, _) = A x(t-1) s(t-1) in
      ht
```

The functional flavor of NN architectures
Example #2: Net-in-net



The functional flavor of NN architectures
Example #3: Stack augmented NNs



- A RNN, where each layer perform a linear combination of push and pop operations
- This is state-passing-style
- H: $A \otimes S \rightarrow A \otimes S$

NNs are more functional than imperative

OVERVIEW

Quick introduction to inductive datatypes

Algebraic Data Types

- ADTs allow the programmer to create new datatypes from existing ones

Algebraic Data Types

- ADTs allow the programmer to create new datatypes from existing ones

Examples - Sum types

data ***Bool*** = ***True*** | ***False***

Algebraic Data Types

- ADTs allow the programmer to create new datatypes from existing ones

Examples - Product types

data ***Point*** *a* = ***P*** *a a*

P *1.0 2.0 ::* ***Point*** *Float*

Algebraic Data Types

- ADTs allow the programmer to create new datatypes from existing ones

Examples - The type List of lists of elements of type *a*

data ***List a*** = ***Nil*** | ***Cons*** *a* (***List*** *a*)

Cons *1* (***Cons*** *2* ***Nil***) :: ***List*** *Int*

Algebraic Data Types

- ADTs come with a battery of useful programming patterns and tool
- An example: **pattern matching**
  - match b :: **Bool** with
    | **True** → # Do something
    | **False** → # Do something else
  - match l :: **List a** with
    | **Nil** → # Do something
    | **Cons** x q → # Do something else with x and q

Inductive datatypes

- Inductive datatypes are defined as fixed points
- Integers
  - data *Int* = *Zero* | *Succ Int*
  - 3 = Succ (Succ (Succ Zero)) :: Int
- Lists
  - data *List a* = *Nil* | *Cons a* (*List a*)
  - [1, 2, 3] = Cons 1 (Cons 2 (Cons 3 Nil)) :: List Int
- Binary Trees
  - data *BinTree a* = *Nil* | *Node a* (*BinTree a*) (*BinTree a*)
  - Node 1 (Node 2 Nil Nil) (Node 3 Nil Nil) :: BinTree Int
- Domain Specific Languages & ASTs

Data structure determines program structure

- *"There are certain close analogies between the methods used for structuring data and the methods for structuring a program which processes that data."*
  —— Hoare

- Inductive datatypes come with associated recursive schemes used to build programs that process that data.

Fold

- Consider the following function on lists
  - **size** (**Nil**) = 0
  - **size** (**Cons** x q) = **size** q + 1
- This function collapses a list into a value (here, an integer)
- **fold** generalizes this type of function to any list
  - **fold** :: b → (a → b → b) → **List** a → b
  - **fold** z f (**Cons** x_1 (**Cons** x_2 … (**Cons** x_n y)…)) → f x_1 (f x_2 … (f x_n z) …)
  - **size** = **fold** 0 (\x -> \k -> 1 + k)

Fold

- The same exists for **Int** or **BinTree**
- **Int**
  - **data** *Int* = ***Zero*** | *Succ Int*
  - **fold** :: b ⟶ (b ⟶ b) ⟶ **Int** ⟶ b
  - **fold** z f **n** ⟶ $f^n(z)$ = f … f z
- **BinTree**
  - **data** *BinTree a* = ***Leaf*** | *Node a (BinTree a) (BinTree a)*
  - **fold** :: b ⟶ (a ⟶ b ⟶ b ⟶ b) ⟶ **BinTree a** ⟶ b
  - **fold** z f (**Node a t1 t2**) ⟶ f a (I fold z f t1) (I fold z f t2)

Fold

- **fold** can be generalized to any ADT
- Useful to build programs that follow closely their input
- It corresponds to the categorical notion of **catamorphism**
- **Spoiler: there is a whole world beyond catamorphisms**

A PROPOSAL

Datatypes as a central component to deep learning architectures

Catamorphisms everywhere

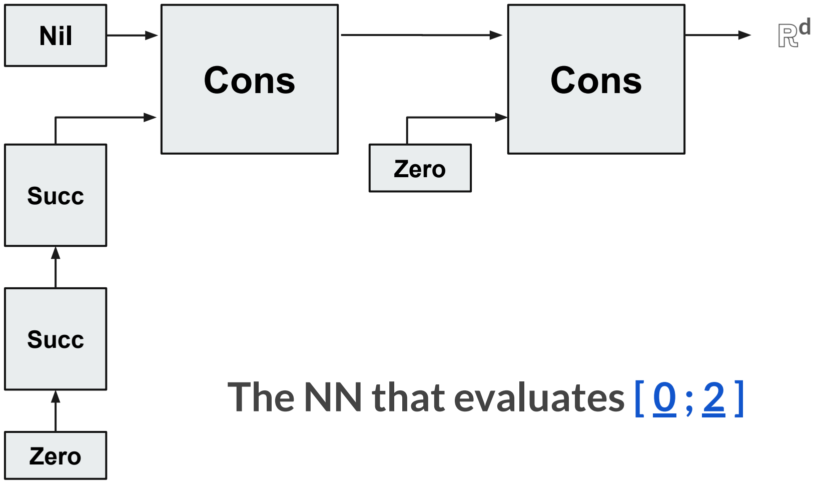
- I propose a re-decomposition of neural networks according to two components:
  - Datatype
  - Layers
- The choice of datatype is responsible for the overall “shape” of the NN
  - A datatype AND a recursion scheme
- The layers are the individual operators that flow around the data, that contain both algorithmic and numerical components
  - Ex: Encoder block

Datatype at the forefront

- A choice of inductive datatype (used to encode the input) and an output dimension **d**
- For each constructor C, a program A_C with the right type.
  - ex: for Succ :: Int → Int , we have $A_{\mathbf{Succ}}$ :: $R^d \to R^d$
- Each A_C has a certain number N_C of parameters, represented by free variables of type $\mathbb{R}$
- The NN is defined by **fold $\mathbf{C_1\ C_2 \dots C_n\ x}$** : $\mathbb{R}^{\mathbf{d}}$
- Actually, there is a catch: if your ADT is actually using other ADTs as parameters, you have to apply the various fold functions recursively
  - example for List(Int): fold z (Cons n l) = C_{Cons} (fold$_{int}$ C_{Zero} C_{Succ} n) (fold C_{Nil} C_{Cons} l)

RNN - The List(Int) example

- For Int:
  - A_{Zero} : $\mathbb{R}^k$
  - A_{Succ} : $\mathbb{R}^k \longrightarrow \mathbb{R}^k$
- For List(Int)
  - A_{Nil} : $\mathbb{R}^d$
  - A_{Cons}: $\mathbb{R}^k \longrightarrow \mathbb{R}^d \longrightarrow \mathbb{R}^d$



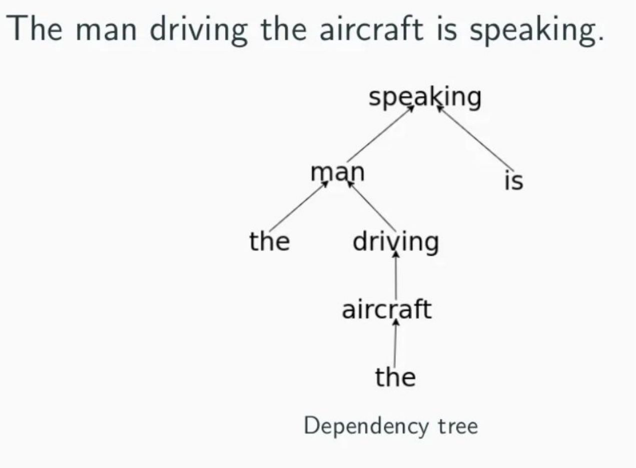
The NN that evaluates [0 ; 2]

Token embedding

- In NLP, we typically process an input that is a list of tokens
- Provided to the NN through a token embedding
- Given a vocabulary of size d, each token is represented by a vector R^k
- In ADT words:
  - data Token = Tok_1 | Tok_2 | … | Tok_d
  - C_{tok1} is a R^k free variable

Tree-LSTMs

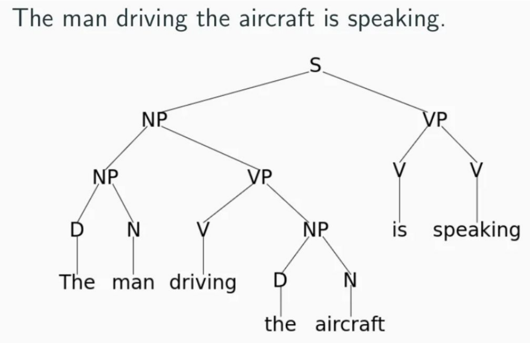
- Tree-LSTMs are similar to LSTMs (which are variants of RNNs) but take trees as input



- **data** Token = man | is | speaking | …
  - **all possible words in the vocabulary**
- **data** Tree a = Leaf | Node a (List Tree)
- **data** DepTree = Tree Token
- A_{Cons} x y = x + y
  - Node does sum pooling on its children
- A_{Node} is basically the LSTM block

Tree-LSTMs

- Tree-LSTMs are similar to LSTMs (which are variants of RNNs) but take trees as input

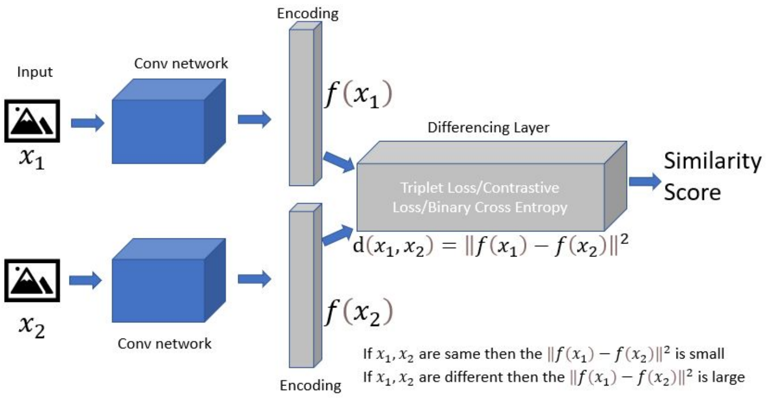


Constituency tree

data ConsTree a = S (Tree a) (Tree a) | NP (Tree a) (Tree a) | VP (Tree a) (Tree a) | D a | N a | V a

Siamese networks

- Siamese networks are about comparing two inputs (e.g, image similarity)



- data **Siamese a** = **Siam a a**
- If we take **Siamese Int**
- S :: $R^d \to R^d \to R$
- $\text{fold}_{\text{Siamese}}\ C_{\text{Siam}}\ (\text{Siam}\ x\ y) =$

  $C_{\text{Siam}}\ (\text{fold}_{\text{Int}}\ C_{\text{Zero}}\ C_{\text{Succ}}\ x)\ (\text{fold}_{\text{Int}}\ C_{\text{Zero}}\ C_{\text{Succ}}\ y)$
- S is the comparison block (e.g, distance)

By given having datatype as a first-class citizen...

- You can easily test various data representations and see which one yields the best results
- Induce interesting biases and properties intentionally, by choosing the data structure that possesses the traits you are looking for.
- You can actually achieve true end-to-end learning
- Explore the world of FP data structures and see what kind of NN they induce

Examples of interest: the zipper

- A way to represent the notion of “structure with a hole” (a context)
- A way to consider a location in a structure and provide navigation methods
- data ListZip a = Zip (List a) a (List a)

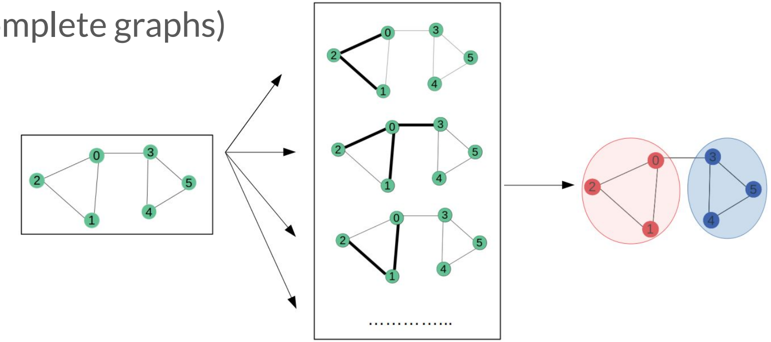
$$[\, a_1 ; a_2 ; \boxed{a_3} ; a_4 ; \ldots ; a_n \,] = \text{Zip}\,[\, a_2; a_1 \,]\ \ a_3\ \ [a_4; \ldots ; a_n]$$

$$\Big\downarrow \text{ Go Right}$$

$$\text{Zip}\,[\, a_3 ; a_2; a_1 \,]\ \ a_4\ \ [a_5; \ldots ; a_n]$$

Examples of interest: graphs

- Convolutional Graph Neural Networks work with graphs as inputs
- Conventional ADTs can't represent cycles, but generalizations exist [2001; Turbak, Wells], where CyFold is computed as a fixed point in an iterated manner
- Graph Neural Networks correspond to some bounded version of CyFold
- Transformers can be seen as particular GNNs (complete graphs)



Beyond ADTs and catamorphisms

- Other recursion schemes
  - Paramorphisms: Residual Link
  - Histomorphisms: dynamic programming
- GADT and indexed catamorphisms
- Cyclic structures and cyfold

Conclusion

- Inductive datatypes and fold-like functions provide a nice way to recover classical neural networks
- Composing datatypes allow for modularity of input representation
- Allow to explore the space of neural networks through the prism of existing FP concepts

Thank you.