LEGO® Deep Learning
Medium Brick Box (23 pcs)

ML4Jets 2020, January 15, New York University
Gilles Louppe, g.louppe@uliege.be
The toolbox
People are now building a new kind of software by assembling networks of parameterized functional blocks and by training them from examples using some form of gradient-based optimization.

An increasingly large number of people are defining the networks procedurally in a data-dependent way (with loops and conditionals), allowing them to change dynamically as a function of the input data fed to them.

Yann LeCun, 2018.
DL as an architectural language
Inputs and outputs  Architectures  Losses
Inputs/Outputs

- Vectorized attributes, tabular data
- Images
- Sequences (words, speech, images, videos)
- Graphs
- Structured data

All may serve as both inputs and outputs.
Logistic regression

- I/O: vectorized inputs → class labels
- Architecture: logistic unit
- Loss: cross-entropy

\[ y = \sigma(w^T x + b) \]
Multi-layer perceptron

Stacking logistic units in parallel results in a layer.

Composing layers in series results in the MLP (or fully connected feedforward nets).
Convolutional networks

- I/O: images, audio, text → class labels or regression targets
- Architecture: convolutions, pooling
- Loss: cross-entropy, negative log-likelihood

Convolutional networks extend fully connected nets with **convolutional** and **pooling** layers.
Locality and translation invariance

- Locality: objects tend to have a local spatial support.
- Translation invariance: object appearance is independent of location.

Convolution

Training deep networks leads to **vanishing gradients**.

Skip connections provide a brick to make them **trainable**.
ResNet-34

DenseNet

Unet

(a) without skip connections

(b) with skip connections

Credits: Li et al, 2017.
Recurrent and attention networks

- I/O: sequences → class labels, regression targets, or sequences
- Architecture: recurrent networks, attention
- Loss: cross-entropy, negative log-likelihood

RNNs are equipped with an **internal state**, which acts as memory cell. They are equivalent to feedforward nets but with **shared** layers.
The **seq2seq** model is an encoder-decoder architecture composed of:

- an encoder RNN that processes the input sequence and compresses the information into a **context vector** $\mathbf{c}$ of fixed length;

- a decoder RNN initialized with the context vector that emits an output sequence.
Attention

For inferring the next state $s_t$, the attention layer builds the dynamic context $c_t$ which estimates how strongly the current state is correlated with (or attends to) other elements.

$$c_t = \sum_i \alpha_{t,i} h_i \quad \alpha_{t,i} = \text{softmax}(\text{score}(s_{t-1}, h_i))$$

Credits: Bahdanau et al, 2015.
Transformer networks compose **multi-head self-attention** mechanisms for sequence-to-sequence modeling, **without** recurrent units.

\[
\text{softmax}(\frac{QK^T}{\sqrt{n}})V
\]

**Credits:** Vaswani et al., 2017.
Neural computers

Networks can be coupled with memory storage to produce neural computers:

- The controller processes the input sequence and interacts with the memory to generate the output.
- The read and write operations attend to all the memory addresses.
Programs as neural nets

- I/O: structured data (one or more sources) → class labels, regression targets or structured data
- Architecture: graph networks, dynamic graphs of computation
- Loss: cross-entropy, negative log-likelihood

A graph is created on the fly

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```
Recursive nets for jet physics

- Use sequential recombination jet algorithms (kt, anti-kt, etc) to define computational graphs (on a per-jet basis).
- The root node in the graph provides a fixed-length embedding of a jet.

Credits: Louppe et al., 2017.
Neural message passing

Even though the graph topology is dynamic, the unrolled computation is fully differentiable. The program is trainable.

Algorithm 1 Message passing neural network

Require: $N \times D$ nodes $x$, adjacency matrix $A$
  $h \leftarrow \text{Embed}(x)$
  for $t = 1, \ldots, T$ do
    $m \leftarrow \text{Message}(A, h)$
    $h \leftarrow \text{VertexUpdate}(h, m)$
  end for
  $r = \text{Readout}(h)$
return $\text{Classify}(r)$

Further examples of differentiable programs

Differentiable Ranks and Sorting using Optimal Transport

Marc R. Cuturi · Olivier Teboul · Jean-Philippe Vert

Google Research, Brain Team

cuturi, olivier, jvert@google.com

Abstract

Sorting is used pervasively in machine learning, either to define elementary algorithms, such as k-nearest neighbors (k-NN) rules, or to define test-time metrics, such as top-k classification accuracy or ranking losses. Sorting is however a poor match for the end-to-end, automatically differentiable pipelines of deep learning. Indeed, sorting procedures output two vectors, neither of which is differentiable: the vector of sorted values is piecewise linear, while the sorting permutation itself (or its inverse, the vector of ranks) has no differentiable properties to speak of, since it is integer-valued. We propose in this paper to replace the usual sort procedure with a differentiable proxy. Our proxy builds upon the fact that sorting can be seen as an optimal assignment problem, one in which the u values to be sorted are matched to an auxiliary probability measure supported on any increasing family of target values. From this observation, we propose extended rank and sort operators by considering optimal transport (OT) problems (the natural relaxation for assignments) where the auxiliary measure can be any weighted measure supported on an increasing family of target values, where u = w. We recover differentiable operators by regularizing these OT problems with an entropic penalty, and solve them by applying Sinkhorn iterations. Using these smoothed rank and sort operators, we propose differentiable proxies for the classification 0/1 loss as well as for the quantile regression loss.

OptNet: Differentiable Optimization as a Layer in Neural Networks

Brandon Amos · J. Zico Kolter

Abstract

This paper presents OptNet, a network architecture that integrates optimization problems (formally and in the form of quadratic programs) as individual layers in larger end-to-end trainable deep networks. These layers encode constraints and complex dependences between the hidden states that traditional convolutional and fully-connected layers often cannot capture. In this paper, we explore the foundations for such an architecture. We show how techniques from sensitivity analysis, biased optimization, and implicit differentiation can be used to exactly differentiate through these layers and with respect to layer parameters; we develop a highly efficient solver for these layers that exploits fast GPU-based batch solvers within a primal-dual interior point method, and which provides backpropagation gradients with virtually no additional cost on top of the solve; and we highlight the applicability of these approaches in several problems. In one notable example, we show that the method is capable of learning to play mini-Sudoku (4x4) given just input and output games, with no a priori information about the rules of the game; this highlights the ability of our architecture to learn top-down can reduce the overall depth of the network while preserving contexts of representation. Specifically, we build a framework where the output of the i-th layer in a network is the solution to a constrained optimization problem based upon previous layers. This framework naturally encompasses a wide variety of inference problems expressed within a neural network, allowing for the potential of much richer end-to-end training for complex tasks that require such inference procedures.

Concretely, in this paper we specifically consider the task of solving small quadratic programs as individual layers. These optimization problems are well-suited to capturing interesting behavior and can be efficiently solved with GPUs. Specifically, we consider layers of the form

\[ z_{i+1} = \arg\min_z \frac{1}{2} \mathbb{E}_x \left[ Q(z_{i}, x) + \eta(z_{i})^T z \right] \]

subject to \( A(z_{i}) \leq b(z_{i}) \)

\[
Q(z_{i}, x) = \mathbb{E}_y \left[ C(z_{i}, y, x) \right] 
\]

where z is the optimization variable, \( Q(z_{i}, x) \), \( A(z_{i}) \), \( b(z_{i}) \), \( C(z_{i}, y, x) \), and \( h(z_{i}) \) are parameters of the optimization problem. As the notation suggests, these parameters can depend in any differentiable way on the previous layer \( z_{i} \), and which can eventually be optimized just like any other variable in a neural network. These layers can be learned.
LEGO® Creator Expert

Credits: Vinyals et al., 2019.
Conclusions

- Deep Learning is more than feedforward networks.
- It is a methodology:
  - assemble networks of parameterized functional blocks
  - train them from examples using some form of gradient-based optimisation.
- Bricks are simple, but their nested composition can be arbitrarily complicated.
- The Large Creative Brick Box includes many more bricks!

What it is is beautiful.
Credits

This talk is strongly inspired from "Deep Learning: Practice and Trends" (NIPS 2017 Tutorial) by Scott Reed, Nando de Freitas and Oriol Vinyals.

References

- Gilles Louppe, "INFO8010 Deep Learning, ULiège", 2018-2019. [url]
- Jay Alammar, "The Illustrated Transformer", 2018. [url]