QCD-aware Recursive Neural Networks for Jet Physics

arXiv:1702.00748

Gilles Louppe, Kyunghyun Cho, Cyril Becot, Kyle Cranmer

A machine learning perspective

Kyle's talks on QCD-aware recursive nets:

- Theory Colloquium, CERN, May 24, https://indico.cern.ch/event/640111/
- DS@HEP 2017, Fermilab, May 10, https://indico.fnal.gov/conferenceDisplay.py?confId=13497
- Jet substructure and jet-by-jet tagging, CERN, April 20, https://indico.cern.ch/event/633469/
- Statistics and ML forum, CERN, February 14, https://indico.cern.ch/event/613874/contributions/2476427/

Today: the inner mechanisms of recursive nets for jet physics.

Credits: Lecun et al, 2015

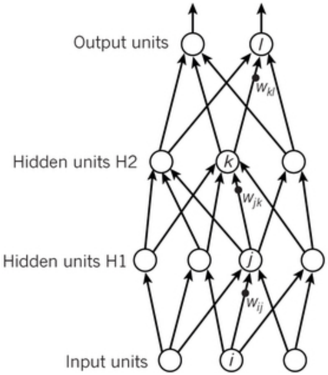
Neural networks 101

Goal = Function approximation

- Learn a map from x to y based solely on observed pairs
- Potentially non-linear map from x to y
- x and y are fixed dimensional vectors

Model = Multi-layer perceptron (MLP)

- Parameterized composition $f(\cdot;\theta)$ of non-linear transformations
- Stacking transformation layers allows to learn (almost any) arbitrary highly non-linear mapping



Learning

- Learning by optimization
- Cost function

$$J(\theta; D) = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, f(x_i; \theta))$$

- Stochastic gradient descent optimization

$$\theta_m := \theta_{m-1} - \eta \nabla_\theta J(\theta_{m-1}; B_m)$$

where $B_m \in D$ is a random subset of D.

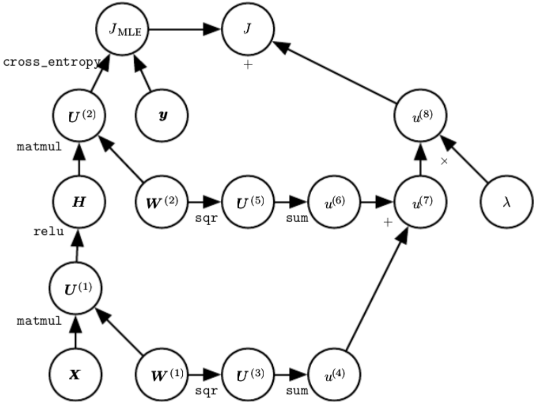
How does one derive $\nabla_\theta J(\theta)$?

Credits: Goodfellow et al, 2016. Section 6.5.

Computational graphs

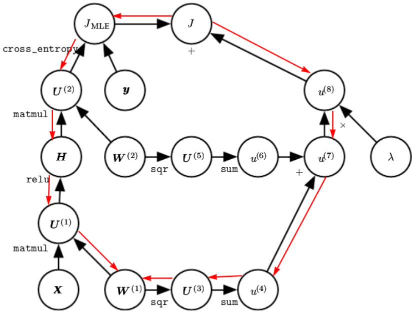
$$f(x;\theta=(W^{(1)},W^{(2)}))=W^{(2)}\texttt{relu}(W^{(1)}x) \quad \text{(simplified 1-layer MLP)}$$

$$J(\theta=(W^{(1)},W^{(2)}))=J_{MLE}+\lambda\left(\sum_{i,j}\left(W^{(1)}_{i,j}\right)^2+\left(W^{(2)}_{i,j}\right)^2\right)$$



Backpropagation

- Backpropagation = Efficient computation of $\nabla_\theta J(\theta)$
- Implementation of the chain rule for the (total) derivatives
- Applied recursively from backward by walking the computational graph from outputs to inputs



$$\frac{dJ}{dW^{(1)}} = \frac{\partial J}{\partial J_{MLE}} \frac{dJ_{MLE}}{dW^{(1)}} + \frac{\partial J}{\partial u^{(8)}} \frac{du^{(8)}}{dW^{(1)}}$$

$$\frac{dJ_{MLE}}{dW^{(1)}} = \ldots \quad \text{(recursive case)}$$

$$\frac{du^{(8)}}{dW^{(1)}} = \ldots \quad \text{(recursive case)}$$

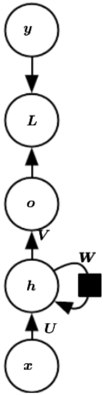
Recurrent networks

Setup

- Sequence $x = (x_1, x_2, ..., x_\tau)$
  - E.g., a sentence given as a chain of words
- The length of each sequence may vary

Model = Recurrent network

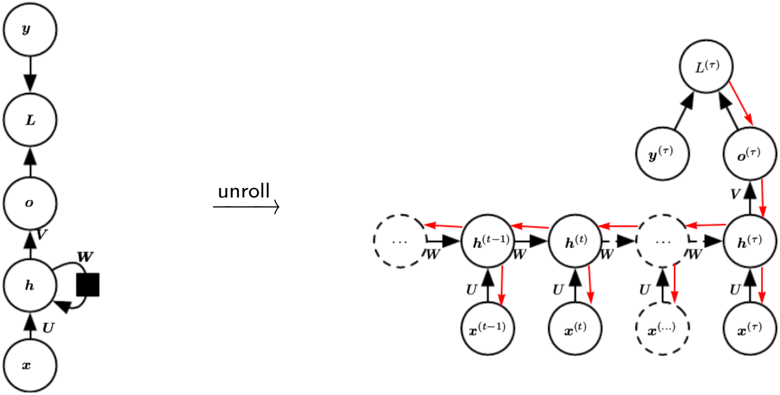
- Compress x into a single vector by recursively applying a MLP with shared weights on the sequence, then compute output.
- $h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$
- $o = g(h^{(\tau)}; \theta)$



How does one backpropagate through the cycle?

Backpropagation through time

- Unroll the recurrent computational graph through time
- Backprop through this graph to derive gradients



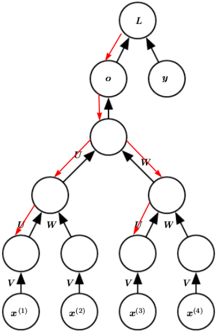
This principle generalizes to any kind of (recursive or iterative) computation that can be unrolled into a directed acyclic computational graph.

(That is, to any program!)

Recursive networks

Setup

- x is structured as a tree
  - E.g., a sentence and its parse tree
- The topology of each training input may vary



Model = Recursive networks

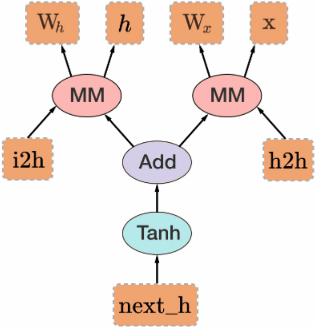
- Compress x into a single vector by recursively applying a MLP with shared weights on the tree, then compute output.
- $h^{(t)} = \begin{cases} v(x^{(t)}; \theta) & \text{if } t \text{ is a leaf} \\ f(h^{(t_{\text{left}})}, h^{(t_{\text{right}})}; \theta) & \text{otherwise} \end{cases}$
- $o = g(h^{(0)}; \theta)$

Credits: pytorch.org/about

Dynamic computational graphs

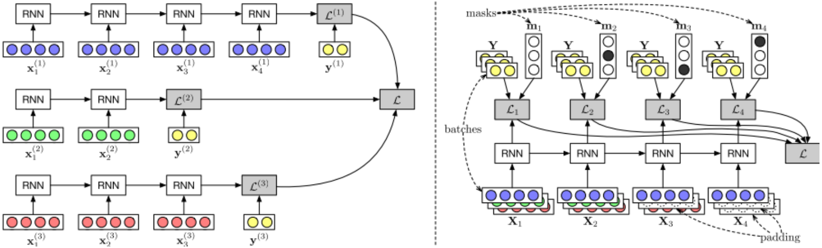
- Most frameworks (TensorFlow, Theano, Caffee or CNTK) assume a static computational graph.
- Reverse-mode auto-differentiation builds computational graphs dynamically on the fly, as code executes.
  - One can change how the network behaves (e.g. depending on the input topology) arbitrarily with zero lag or overhead.
  - Available in autograd, Chainer, PyTorch or DyNet.

```
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))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```



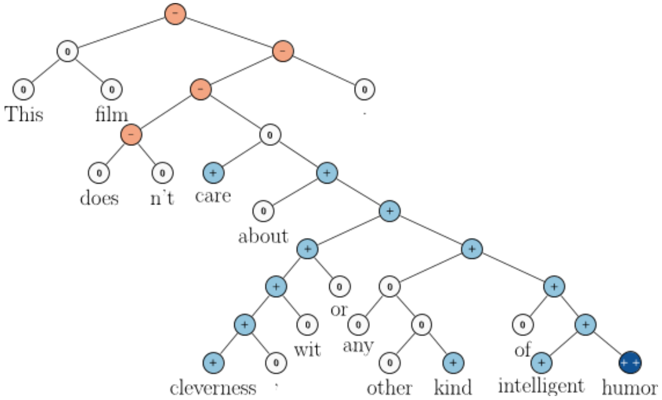
Operation batching

- Distinct per-sample topologies make it difficult to vectorize operations.
- However, in the case of trees, computations can be performed in batch level-wise, from bottom to top.



On-the-fly operation batching (in DyNet)

From sentences to jets

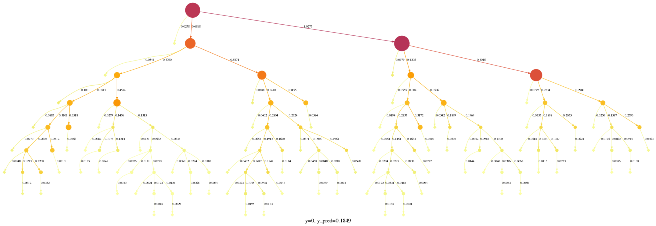


Analogy:

- word $\longrightarrow$ particle
- sentence $\longrightarrow$ jet
- parsing $\longrightarrow$ jet algorithm

Jet topology

- Use sequential recombination jet algorithms (k_T, anti-k_T, etc) to define computational graphs (on a per-jet basis).
- The root node in the graph provides a fixed-length embedding of a jet, which can then be fed to a classifier.
- Path towards ML models with good physics properties.



A jet structured as a tree by the k_T recombination algorithm

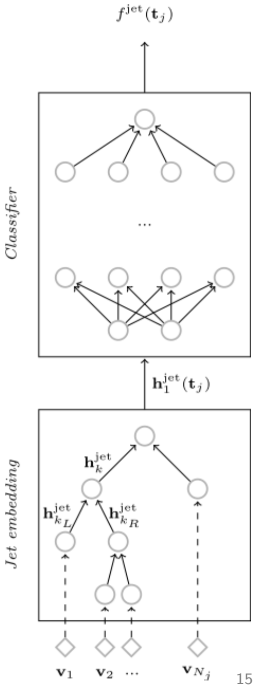
QCD-aware recursive neural networks

Simple recursive activation: Each node k combines a non-linear transformation u_k of the 4-momentum o_k with the left and right embeddings h_{k_L} and h_{k_R}.

$$\mathbf{h}_k^{\text{jet}} = \begin{cases} \mathbf{u}_k & \text{if } k \text{ is a leaf} \\ \sigma\left(W_h \begin{bmatrix} \mathbf{h}_{k_L}^{\text{jet}} \\ \mathbf{h}_{k_R}^{\text{jet}} \\ \mathbf{u}_k \end{bmatrix} + b_h \right) & \text{otherwise} \end{cases}$$

$$\mathbf{u}_k = \sigma\left(W_u g(\mathbf{o}_k) + b_u \right)$$

$$\mathbf{o}_k = \begin{cases} \mathbf{v}_{i(k)} & \text{if } k \text{ is a leaf} \\ \mathbf{o}_{k_L} + \mathbf{o}_{k_R} & \text{otherwise} \end{cases}$$



QCD-aware recursive neural networks

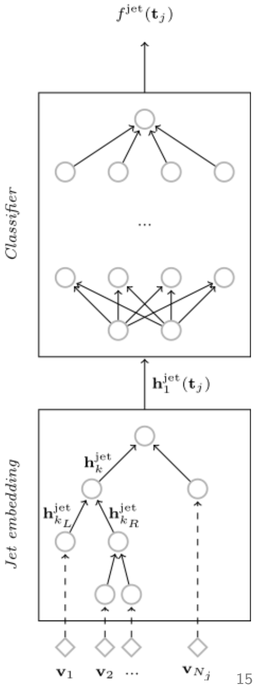
Gated recursive activation: Each node actively selects, merges or propagates up the left, right or local embeddings as enabled with reset and update gates $\mathbf{r}$ and $\mathbf{z}$. (Similar to a GRU.)

$$\mathbf{h}_k^{\text{jet}} = \begin{cases} \mathbf{u}_k & \text{if } k \text{ is a leaf} \\ \mathbf{z}_H \odot \tilde{\mathbf{h}}_k^{\text{jet}} + \mathbf{z}_L \odot \mathbf{h}_{k_L}^{\text{jet}} + & \text{otherwise} \\ \hookrightarrow \mathbf{z}_R \odot \mathbf{h}_{k_R}^{\text{jet}} + \mathbf{z}_N \odot \mathbf{u}_k & \end{cases}$$

$$\tilde{\mathbf{h}}_k^{\text{jet}} = \sigma\left(W_{\tilde{h}} \begin{bmatrix} \mathbf{r}_L \odot \mathbf{h}_{k_L}^{\text{jet}} \\ \mathbf{r}_R \odot \mathbf{h}_{k_R}^{\text{jet}} \\ \mathbf{r}_N \odot \mathbf{u}_k \end{bmatrix} + b_{\tilde{h}} \right)$$

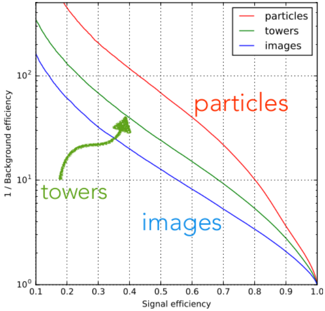
$$\begin{bmatrix} \mathbf{z}_H \\ \mathbf{z}_L \\ \mathbf{z}_R \\ \mathbf{z}_N \end{bmatrix} = \text{softmax}\left(W_z \begin{bmatrix} \tilde{\mathbf{h}}_k^{\text{jet}} \\ \mathbf{h}_{k_L}^{\text{jet}} \\ \mathbf{h}_{k_R}^{\text{jet}} \\ \mathbf{u}_k \end{bmatrix} + b_z \right)$$

$$\begin{bmatrix} \mathbf{r}_L \\ \mathbf{r}_R \\ \mathbf{r}_N \end{bmatrix} = \text{sigmoid}\left(W_r \begin{bmatrix} \mathbf{h}_{k_L}^{\text{jet}} \\ \mathbf{h}_{k_R}^{\text{jet}} \\ \mathbf{u}_k \end{bmatrix} + b_r \right)$$



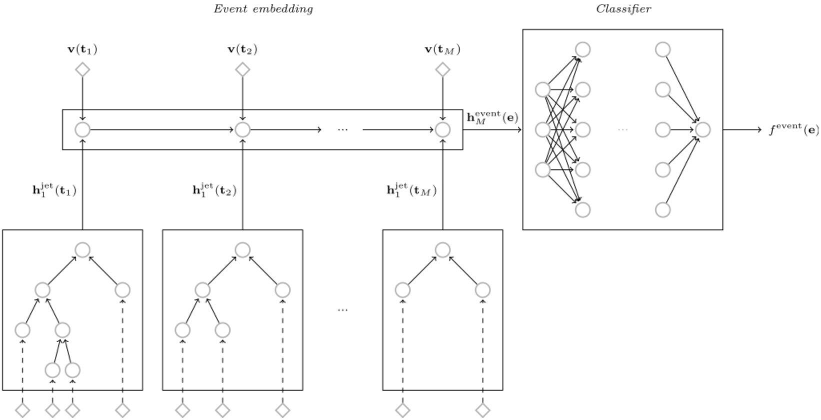
Jet-level classification results

- W-jet tagging example (data from 1609.00607)
- On images, RNN has similar performance to previous CNN-based approaches.
- Improved performance when working with calorimeter towers, without image pre-processing.
- Working on truth-level particles led to significant improvement.
- Choice of jet algorithm matters.



Input	Architecture	ROC AUC	$R_{\epsilon=50\%}$
	Projected into images		
towers	MaxOut	**0.8418**	–
towers	k_t	0.8321 ± 0.0025	**12.7 ± 0.4**
towers	k_t (gated)	0.8277 ± 0.0028	12.4 ± 0.3
	Without image preprocessing		
towers	τ_{21}	0.7644	6.79
towers	mass + τ_{21}	0.8212	11.31
towers	k_t	0.8807 ± 0.0010	24.1 ± 0.6
towers	C/A	0.8831 ± 0.0010	24.2 ± 0.7
towers	anti-k_t	0.8737 ± 0.0017	22.3 ± 0.8
towers	asc-p_T	0.8835 ± 0.0009	**26.2 ± 0.7**
towers	desc-p_T	**0.8838 ± 0.0010**	25.1 ± 0.6
towers	random	0.8704 ± 0.0011	20.4 ± 0.3
particles	k_t	0.9185 ± 0.0006	68.3 ± 1.8
particles	C/A	**0.9192 ± 0.0008**	68.3 ± 3.6
particles	anti-k_t	0.9096 ± 0.0013	51.7 ± 3.5
particles	asc-p_T	0.9130 ± 0.0031	52.5 ± 7.3
particles	desc-p_T	0.9189 ± 0.0009	**70.4 ± 3.6**
particles	random	0.9121 ± 0.0008	51.1 ± 2.0
	With gating (see Appendix A)		
towers	k_t	0.8822 ± 0.0006	25.4 ± 0.4
towers	C/A	0.8861 ± 0.0014	26.2 ± 0.8
towers	anti-k_t	0.8804 ± 0.0010	24.4 ± 0.4
towers	asc-p_T	0.8849 ± 0.0012	27.2 ± 0.8
towers	desc-p_T	**0.8864 ± 0.0007**	**27.5 ± 0.6**
towers	random	0.8751 ± 0.0029	22.8 ± 1.2
particles	k_t	0.9195 ± 0.0009	74.3 ± 2.4
particles	C/A	**0.9222 ± 0.0007**	81.8 ± 3.1
particles	anti-k_t	0.9156 ± 0.0012	68.3 ± 3.2
particles	asc-p_T	0.9137 ± 0.0046	54.8 ± 11.7
particles	desc-p_T	0.9212 ± 0.0005	**83.3 ± 3.1**
particles	random	0.9106 ± 0.0035	50.7 ± 6.7

From paragraphs to events



Analogy:

- word $\longrightarrow$ particle
- sentence $\longrightarrow$ jet
- parsing $\longrightarrow$ jet algorithm
- paragraph $\longrightarrow$ event

Joint learning of jet embedding, event embedding and classifier.

Event-level classification results

RNN on jet-level 4-momentum $v(t_j)$ only vs. adding jet-embeddings h_j:

- Adding jet embedding is much better (provides jet tagging information).

RNN on jet-level embeddings vs. RNN that simply processes all particles in the event:

- Jet clustering and jet embeddings help a lot!

Input	ROC AUC	$R_{\epsilon=80\%}$
Hardest jet		
$\mathbf{v}(\mathbf{t}_j)$	0.8909 ± 0.0007	5.6 ± 0.0
$\mathbf{v}(\mathbf{t}_j)$, $\mathbf{h}_j^{\text{jet}(k_t)}$	**0.9602 ± 0.0004**	**26.7 ± 0.7**
$\mathbf{v}(\mathbf{t}_j)$, $\mathbf{h}_j^{\text{jet}(\text{desc}-p_T)}$	0.9594 ± 0.0010	25.6 ± 1.4
2 hardest jets		
$\mathbf{v}(\mathbf{t}_j)$	0.9606 ± 0.0011	21.1 ± 1.1
$\mathbf{v}(\mathbf{t}_j)$, $\mathbf{h}_j^{\text{jet}(k_t)}$	0.9866 ± 0.0007	156.9 ± 14.8
$\mathbf{v}(\mathbf{t}_j)$, $\mathbf{h}_j^{\text{jet}(\text{desc}-p_T)}$	**0.9875 ± 0.0006**	**174.5 ± 14.0**
5 hardest jets		
$\mathbf{v}(\mathbf{t}_j)$	0.9576 ± 0.0019	20.3 ± 0.9
$\mathbf{v}(\mathbf{t}_j)$, $\mathbf{h}_j^{\text{jet}(k_t)}$	0.9867 ± 0.0004	152.8 ± 10.4
$\mathbf{v}(\mathbf{t}_j)$, $\mathbf{h}_j^{\text{jet}(\text{desc}-p_T)}$	**0.9872 ± 0.0003**	**167.8 ± 9.5**
No jet clustering, desc-p_T on $\mathbf{v}_i$		
$i = 1$	0.6501 ± 0.0023	1.7 ± 0.0
$i = 1, \ldots, 50$	**0.8925 ± 0.0079**	**5.6 ± 0.5**
$i = 1, \ldots, 100$	0.8781 ± 0.0180	4.9 ± 0.6
$i = 1, \ldots, 200$	0.8846 ± 0.0091	5.2 ± 0.5
$i = 1, \ldots, 400$	0.8780 ± 0.0132	4.9 ± 0.5

Summary

- Neural networks are computational graphs whose architecture can be molded on a per-sample basis to express and impose domain knowledge.
- Our QCD-aware recursive net operates on a variable length set of 4-momenta and use a computational graph determined by a jet algorithm.
  - Experiments show that topology matters.
  - Alternative to image-based approaches.
  - Requires much less data to train (10-100x less data).
- The approach directly extends to the embedding of full events. Intermediate jet representation helps.
- Many more ideas of hybrids of QCD and machine learning!