

A new generation of PDFs with deep learning models

Operator implementation in TensorFlow

Jesus Urtasun Elizari, University of Milan

Cambridge, June 2019



UNIVERSITÀ  
DEGLI STUDI  
DI MILANO



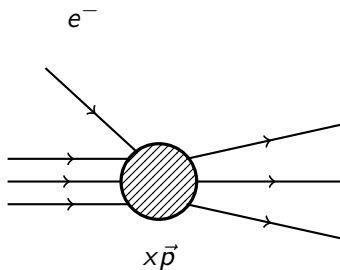
European  
Research  
Council

# Outline

1. Introduction. General structure of a process.
2. Structure of **n3fit**. Motivation for operator implementation.
3. Operator implementation in TensorFlow.
4. Results & Conclusions.

# General structure of a process

## Deep Inelastic Scattering



Convolute the partonic  $\hat{\sigma}_{ij}$  with the PDF  $\rightarrow$  Observable  $\sigma$ .

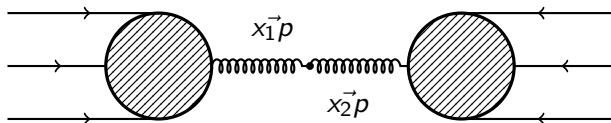
$$\sigma^{DIS} = \int_0^{\infty} dx f_{\alpha}(x) * \hat{\sigma}_{ij}(\mu_F, \mu_R(\alpha_s)) . \quad (1)$$

In our language  $\rightarrow$  vector of observables from a grid of  $x_i$ :

$$y_N^{DIS} = \sum_{i,\alpha} f_{\alpha}(x_i) FK_{Ni\alpha} . \quad (2)$$

# General structure of a process

Hadronic (Example: Drell-Yan)



Convolute the partonic  $\hat{\sigma}_{ij}$  with the PDF  $\rightarrow$  Observable  $\sigma$ .

$$\sigma^{DY} = \int_0^{\infty} dx_1 dx_2 f_{\alpha}(x_1) * f_{\beta}(x_2) * \hat{\sigma}_{ij}(\mu_F, \mu_R(\alpha_s)) . \quad (3)$$

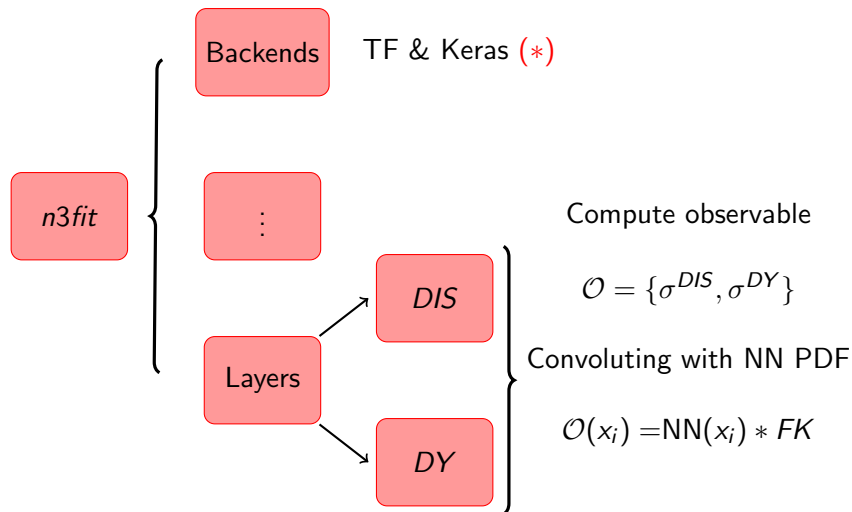
In our language  $\rightarrow$  vector of observables from a grid of  $x_i$ :

$$y_N^{DY} = \sum_{i,j,\alpha,\beta} f_{\alpha}(x_i) f_{\beta}(x_j) FK_{Nij\alpha\beta} . \quad (4)$$

Note: We will use DY to refer hadronic events

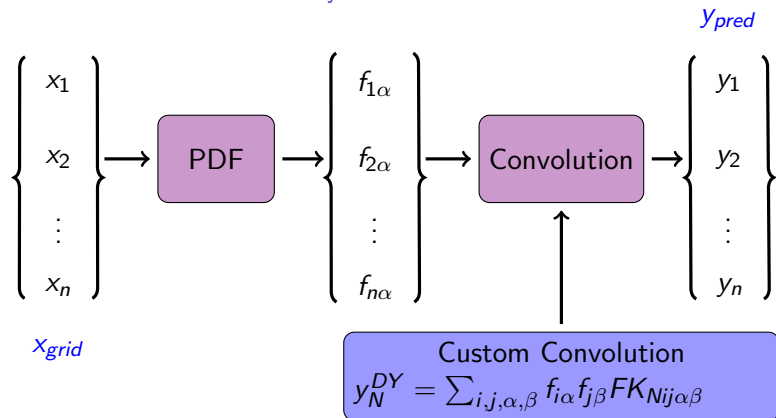
# General structure of the model

## General structure of n3fit



# Operator implementation

## i) General structure of observable layers



1. Build model to compute  $y_{pred}$  from  $x_{grid}$
2. Compute  $\chi^2$  loss by comparing with data  
$$\chi^2 = \sum_{i=1}^N y_i^{pred} - y_i^{data}$$
3. Compute gradient  $\nabla \chi^2$  and update values of PDF  $\rightarrow$  **Fit**

# Operator implementation

## ii) TF computation of the gradient

1. Computing gradient of the  $\chi^2$  with respect to all parameters of the network.

$$\nabla \chi^2 \longrightarrow \frac{\partial \chi^2}{\partial x_i}$$

2. TF requires the gradient with respect to any operation in the model.

$$\frac{\partial \chi^2}{\partial x_i} = \frac{\partial \chi^2}{\partial \mathbf{Op}} \frac{\partial \mathbf{Op}}{\partial f_{\mu\nu}} \cdots \frac{\partial f_{\mu\nu}}{\partial x_i} \quad (5)$$

3. TF does not know the structure of **Op**. Compute manually the gradient  $\frac{\partial \mathbf{Op}}{\partial f_{\mu\nu}}$  and implement it in the TF framework.

# Operator implementation

## iii) Manual computation of the gradient

1. From the expression of the output:

$$\mathbf{Op} \equiv y_N = \sum_{i,j,\alpha,\beta} f_\alpha(x_i) f_\beta(x_j) FK_{Nij\alpha\beta}$$

2. Gradient of the output with respect to the PDFs:

$$\begin{aligned} \frac{\partial y_N}{\partial f_{\mu\nu}} &= \frac{\partial}{\partial f_{\mu\nu}} \sum_{i,j,\alpha,\beta} f_{i\alpha} f_{j\beta} FK_{Nij\alpha\beta} \\ &= \sum_{i,j,\alpha,\beta} (\delta_{\mu i} \delta_{\nu \alpha} f_{j\beta} + \delta_{\mu j} \delta_{\nu \beta} f_{i\alpha}) FK_{Nij\alpha\beta} \\ &= \sum_{j,\beta} f_{j\beta} FK_{N\mu j \nu \beta} + \sum_{i,\alpha} f_{i\alpha} FK_{Ni \mu \alpha \nu} \end{aligned} \quad (6)$$



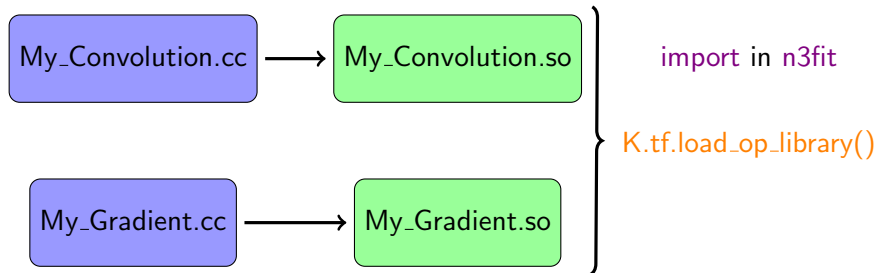
# Through technical details

## i) Generating libraries

1. Build `.cc` files computing the convolution and the gradient.
2. Compile them to build `.so` libraries.

```
g++ -std=c++11 -shared My_Convolution.cc -o  
My_Convolution.so $TF_FLAGS
```

3. load the libraries in DIS/DY layers of `n3fit` with  
`load_op_library()` function of TensorFlow.



# Through technical details

## ii) Writing the convolution in c++

1. Input PDF:  $f_{i\alpha}$  and FK table:  $FK_{Nij\alpha\beta}$
2. Basis flavors:  $basis = \{(1, 2), (3, 4), \dots, (3, 1)\}$

```
// Define the op's interface, Register the op in the tensorflow system by specify inputs and outputs and required attributes
// Create an op that takes two tensors of float and outputs a the tensor product
REGISTER_OP("MyConvolutionBy")
  .Input("pdf: float")
  .Input("fk table: float")
  .Attr("basis flav 1: list(int)")
  .Attr("basis flav 2: list(int)")
  .Output("y_pred: float")
  // Define the shape of the output tensor
  .SetShapeFn([](::tensorflow::shape_inference::InferenceContext* c) {
    // Give to the output 0 the next shape (the last one of the fk table (input 1))
    c->set_output(0, c->MakeShape( { c->Dim(c->input(1), 0) } ));
    return Status::OK();
  });
```

3. Compute convolution  $y_N^{DY} = \sum_{i,j,\alpha,\beta} f_{i\alpha} f_{j\beta} FK_{Nijc} \delta_{\alpha\beta}^c$

```
// Perform the convolution of the PDFs with the FK table
for (int n = 0; n < fk_shape.dim_size(0); n++) {
  y_pred(n) = 0.0;
  for (int i = 0; i < pdf_shape.dim_size(0); i++) {
    for (int k = 0; k < pdf_shape.dim_size(0); k++) {
      for (int c = 0; c < basis_flav_.size(); c++) {
        // PDFs and the FK table are summed only up to the flavour index, given by the flavour basis
        y_pred(n) += pdf(i, basis_flav_1_[c]) * pdf(k, basis_flav_2_[c]) * fk_table(n, c, i, k);
      }
    }
  }
}
```

# Through technical details

## iii) Writing the observable layer

A DY class written in TF would need at least 3 operations

1. Generate luminosity tensor  $\mathcal{L}_{ij\alpha\beta} = f_{i\alpha} * f_{j\beta}$
2. Apply mask to eliminate the non-active flavors
3. Perform the convolution  $y_N^{DY} = \sum_{i,j,\alpha,\beta} \mathcal{L}_{ij\alpha\beta} FK_{Nijc} \delta_{\alpha\beta}^c$

```
class DY_mask(Observable):  
    def call(self, pdf):  
        # Generate luminosity tensor  
        tensor_luminosity = self.tensor_product(pdf, pdf, axes = 0)  
        # Permute luminosity tensor for doing the convolution  
        permuted_luminosity = self.permute_dimensions(tensor_luminosity, (3,1,2,0))  
        # Eliminate the non-active flavours by applying the mask  
        masked_luminosity = self.boolean_mask(permuted_luminosity, self.basis, axis = 0)  
        # Compute the actual convolution  
        result = self.tensor_product(self.fktable, masked_luminosity, axes = 3)  
        return result
```

Custom DY class just calls the [MyConvolutionDY.cc](#)

```
class DY(Observable):  
    def call(self, pdf):  
        # Convolution computed by Custom_Convolution_DY.cc  
        my_result = conv_out_module.my_convolution_dy( pdf, self.fktable, basis_flavs_1 = self.basis_flavs_1, basis_flavs_2 = self.basis_flavs_2 )  
        return my_result
```

# Results

## Checking computation

DIS only:

	TensorFlow	Custom	Ratio
Convolution	1.9207904	1.9207904	1.0000000
	2.4611666	2.4611664	0.9999999
	1.3516952	1.3516952	1.0000000
Gradient	1.8794115	1.8794115	1.0000000
	1.505316	1.505316	1.0000000
	2.866085	2.866085	1.0000000

# Results

## Checking computation

DY-like only:

	TensorFlow	Custom	Ratio
Convolution	8.142365	8.142366	1.0000001
	8.947762	8.947762	1.0000000
	7.4513326	7.4513316	0.9999999
Gradient	18.525095	18.525095	1.0000000
	19.182995	19.182993	0.9999999
	19.551006	19.551004	0.9999999

# Results

## Checking memory

Global:

	TensorFlow	Custom Convolution	Diff
Virtual	22.3 GB	19.7 GB	2.6 GB
RES	16.7 GB	14.1 GB	2.6 GB

DY-like only:

	TensorFlow	Custom Convolution	Diff
Virtual	17.7 GB	13.8 GB	3.9 GB
RES	12.1 GB	8.39 GB	3.2 GB

# Conclusions & Next steps

1. Custom convolution takes 3 GB less of memory than TensorFlow.
2. Take full control on the computation of the observables.
3. Load FK tables in GPU: new possibilities.

Thank you!



This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 740006.