Machine Learning inference using PYNQ environment in an AWS EC2 F1 Instance

Dr. Marco Lorusso

University of Bologna - Department of Physics and Astronomy

National Institute for Nuclear Physics - Bologna Division

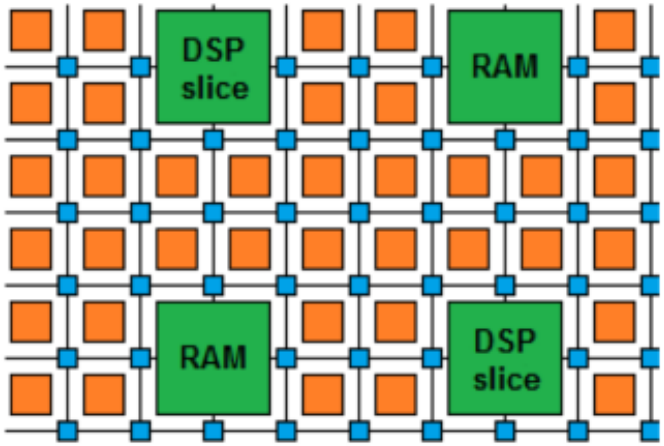
24th March 2022

Field Programmable Gate Array

Field Programmable Gate Arrays (FPGAs) → Middle ground between ASICs and multipurpose CPUs:

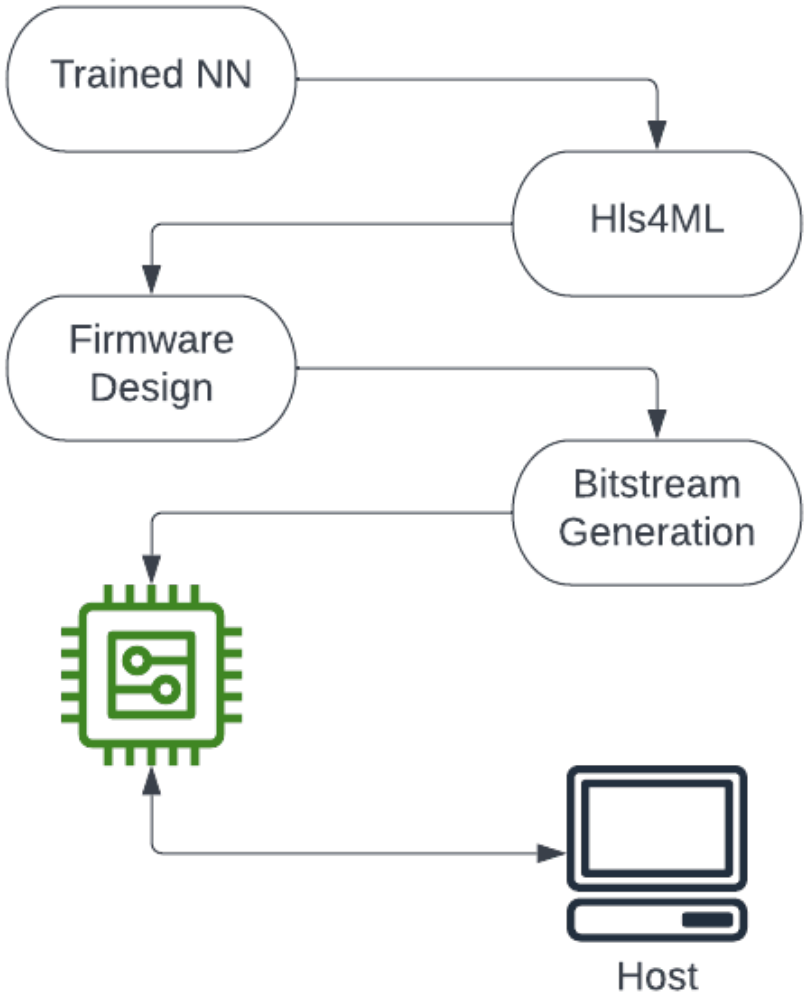
- **Programmables** to perform a wide range of tasks;
- **Low-level/Near-metal** implementation of algorithms → low latency;
- Blend the **benefits of both hardware and software**;
- Internal layout made up of *logic blocks* (**LUTs**, **flipflops**, **Digital Signal Processor** slices), embedded in a **general routing structure**.

FPGA diagram



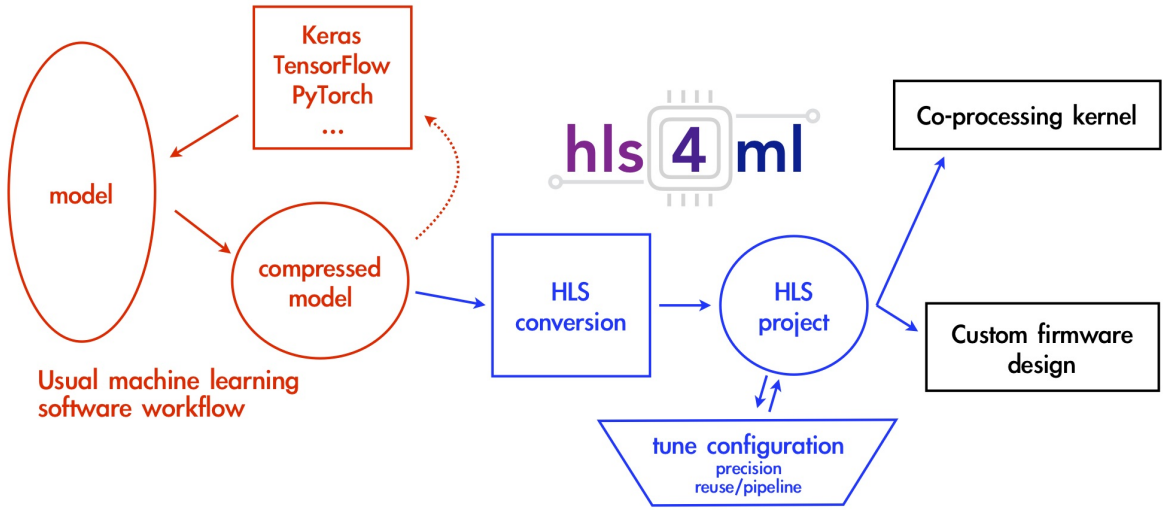
Implementing a Neural Network on an FPGA

- **NN Translation into HLS** (C++) using *hls4ml* (see next slide);
- **Firmware design** (I/O interfaces);
- **Synthesis and implementation** of the design;
- Production of the **bitstream and programming** of the FPGA;
- **Running** of the inference using an application on the **host** machine.



The hls4ml package

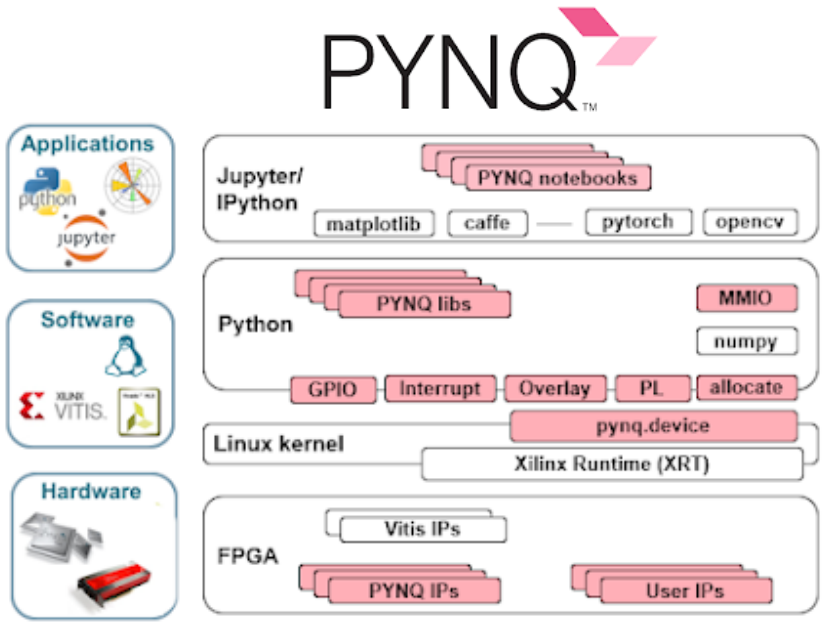
https://fastmachinelearning.org/hls4ml



- Developed by members of the HEP community to translate **ML algorithms** written in **Python** into **High Level Synthesis** code;
- HLS allows the **generation** of **hardware descriptive code** (HDL) from *behavioral descriptions* contained in C++ program;
- The translated Python objects can be injected in the automatic workflow of proprietary software like Vivado from Xilinx Inc.

The PYNQ project

- *PYNQ* is an **open-source** project from Xilinx®;
- It provides a **Jupyter-based framework** with Python APIs for using Xilinx platforms;
- The **Python language** opens up the **benefits of programmable logic** (PL) to people **without** in-depth knowledge of **low-level programming languages**.



https://pynq.readthedocs.io

An introduction to PYNQ

- The **overlay** class is the **core** of the library;
- An overlay object is built providing the **FPGA design** to run on the PL;
- FPGA is **programmed** and relevant **interface** is available through **PYNQ API function** calls;
- It is possible to **accelerate** a software **application**, or to customize the hardware platform for a particular application.

```
1 from pynq import Overlay
2
3 overlay = Overlay("designbitstream.xclbin") # or .awsxclbin
4 result = overlay.<function described in FPGA design>
```

The testing ground: AWS F1 Instances

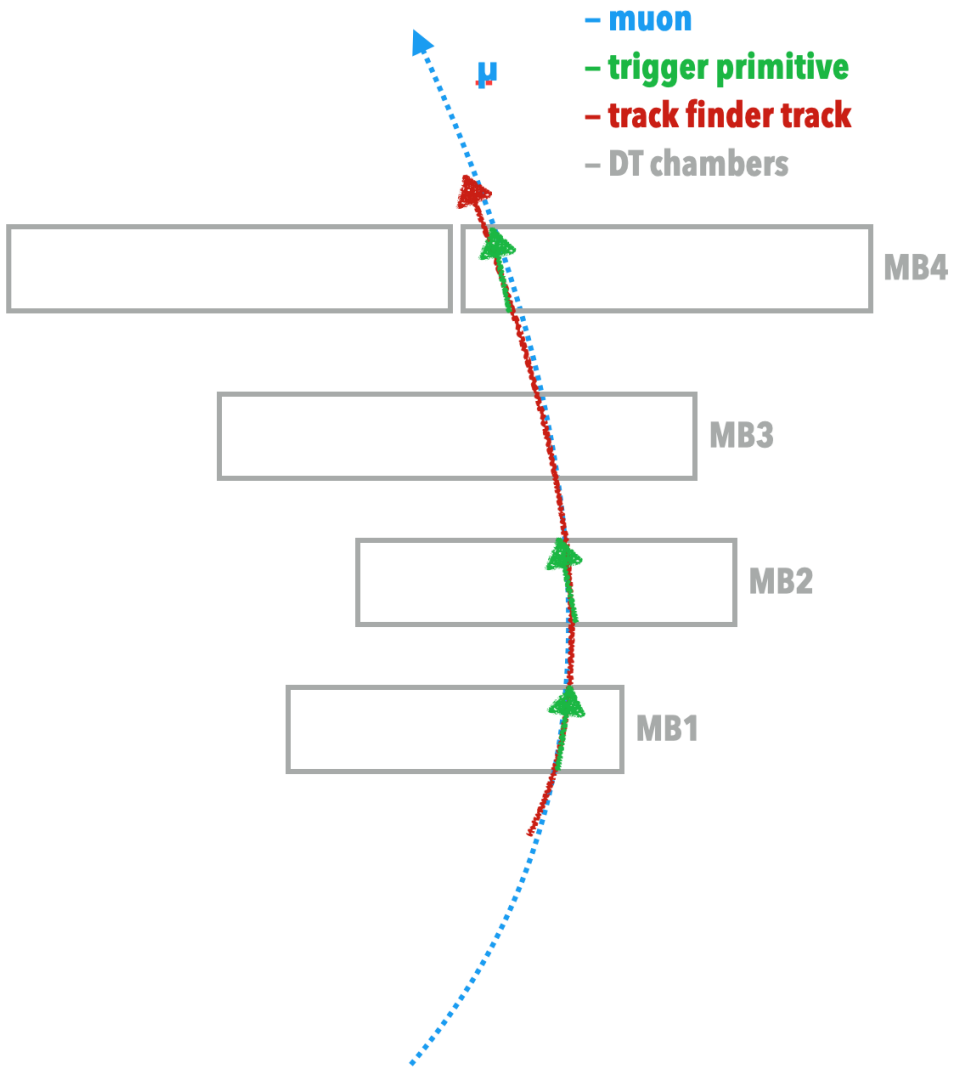
Cloud computing is used to test the capabilities of these tools in preparation for deployment of FPGA accelerator cards in a local server.

- Part of the **AWS Cloud Computing** catalogue;
- EC2 F1 instances use **FPGAs** to enable **delivery** of **custom hardware accelerations**;
- Packaged with **tools** to **develop**, simulate, debug, and **compile** a design;
- Once the FPGA design is complete, it can be registered as an **Amazon FPGA Image** (AFI) and be **reused** across **different F1 instances**.

The tested models

To **test** the **workflow** and the **performance**, **two Neural Networks** have been considered:

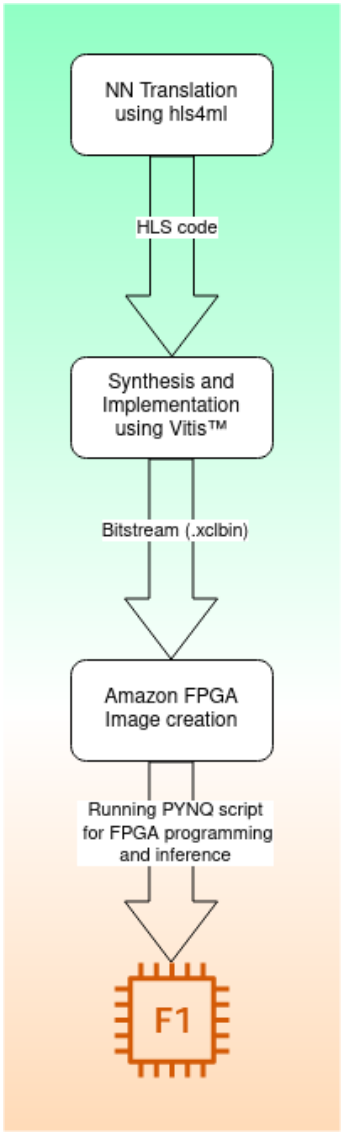
- **Classifier** (pattern recognition) using the IRIS database;
- **Regressor** in the context of **triggering** at the CMS experiment at CERN:
  - NN predicts **transverse momentum** of **muons** using their position and direction in the detector.



Deploying on F1

- Follow the *Application Acceleration development flow*, offered by Vitis™, targeting data center acceleration cards;
- **Upload** the **bitstream** to a S3 bucket and request the **creation** of an *Amazon FPGA Image* (AFI) accessible from all F1 instances;
- Write a **Pyhton script** using PYNQ APIs.

A "more traditional" approach is to use **OpenCL** to write the host application: both ways follow the **same** list of **basic instructions**.



OpenCL vs PYNQ

The first thing to do in both cases, is to **program the device and initialize** the software context.

```
auto devices = xcl::get_xil_devices();
auto fileBuf = xcl::read_binary_file(binaryFile);
cl::Program::Binaries bins{{fileBuf.data(),
    fileBuf.size()}};
OCL_CHECK(err, context = cl::Context({device}, NULL,
    NULL, NULL, &err));
OCL_CHECK(err, q = cl::CommandQueue(context, {device},
    CL_QUEUE_PROFILING_ENABLE, &err));
OCL_CHECK(err, cl::Program program(context, {device},
    bins, NULL, &err));
OCL_CHECK(err, krnl_vector_add = cl::Kernel(program,
    "vadd", &err));
```

```
import pynq
ov =
    pynq.Overlay("model_binary.awsxclbin")
nn = ov.myproject
```

In OpenCL host and FPGA **buffers** need to be handled separately and linked after creation; with PYNQ, the user is only presented with a single interface for both:

```
std::vector<int, aligned_allocator<int>>
    source_in1(DATA_SIZE);
OCL_CHECK(err, l::Buffer buffer_in1(context,
    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
    vector_size_bytes,
    source_in1.data(), &err))
```

```
inp = pynq.allocate(27, 'u2')
out = pynq.allocate(1, 'u2')
```

OpenCL vs PYNQ (cont'd)

To **initiate data transfers** the direction as a function parameter must be specified in OpenCL, while in PYNQ the same is done with a specific function:

```
OCL_CHECK(err, err =
↪ q.enqueueMigrateMemObjects({buffer_input}, 0 /*0 means from host*/,NULL,&eventinp));
```

```
inp.sync_to_device()
```

To **run the kernel** in OpenCL each kernel argument need to be set explicitly using the `setArgs()` function, before starting the execution with `enqueueTask()`; in PYNQ, the `.call()` function does everything in a single line.

```
std::vector<int, aligned_allocator<int>>
↪ source_in1(DATA_SIZE);
OCL_CHECK(err, l::Buffer buffer_in1(context,
    CL_MEM_USE_HOST_PTR | CL_MEM_READ_ONLY,
   ↪ vector_size_bytes,
    source_in1.data(), &err))
```

```
nn.call(inp,out)
```

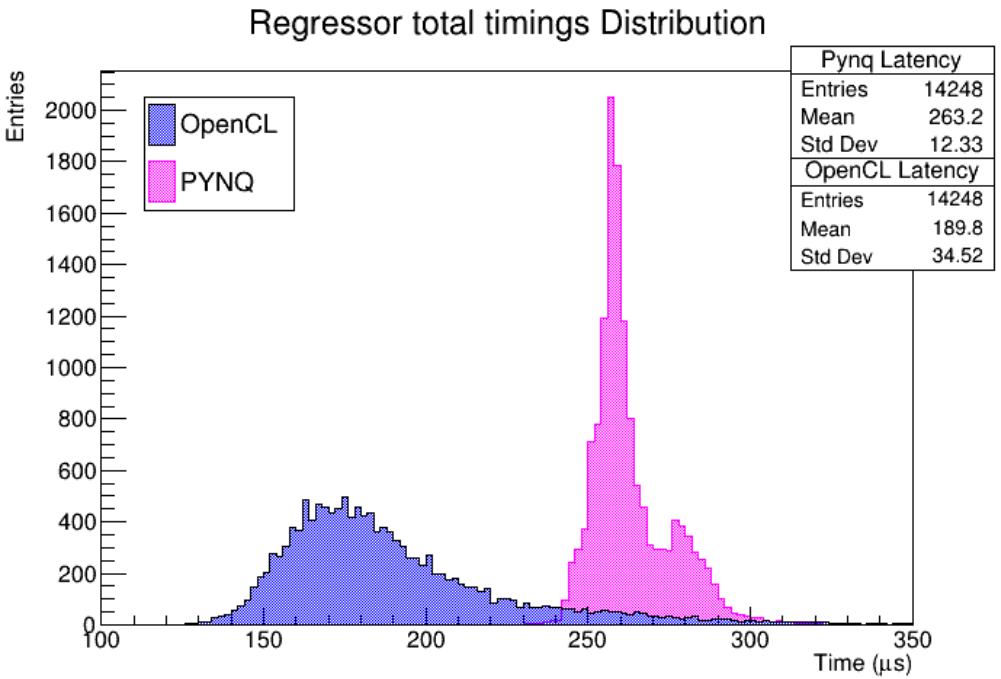
Finally, the **output is retrieved** in both cases similarly to the input transfer:

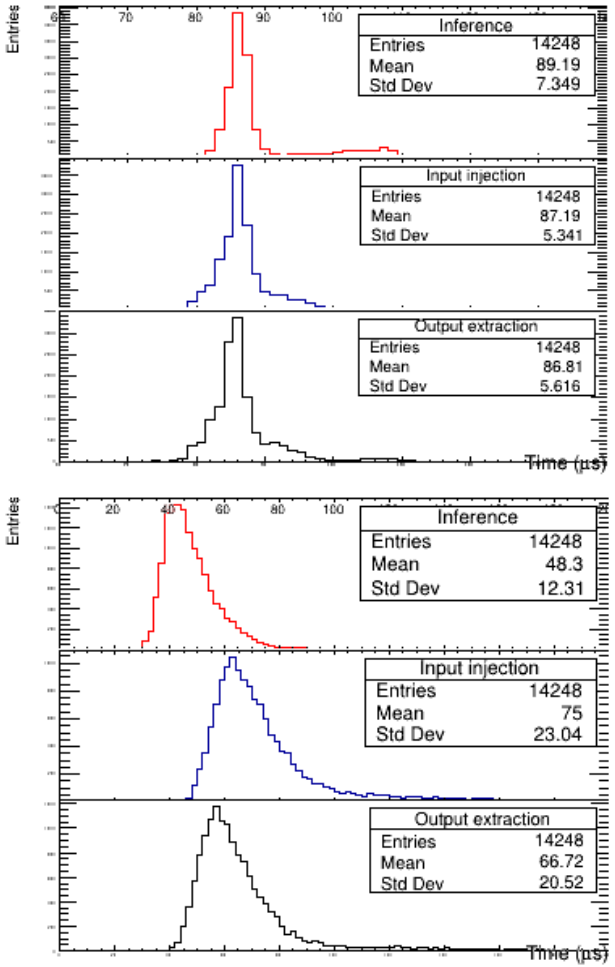
```
OCL_CHECK(err, err =
↪ q.enqueueMigrateMemObjects({buffer_output},
CL_MIGRATE_MEM_OBJECT_HOST));
```

```
out.synq_from_device()
```

Timing Comparison

A **difference** in **computation times** can be seen between the same algorithm deployed with PYNQ and OpenCL:

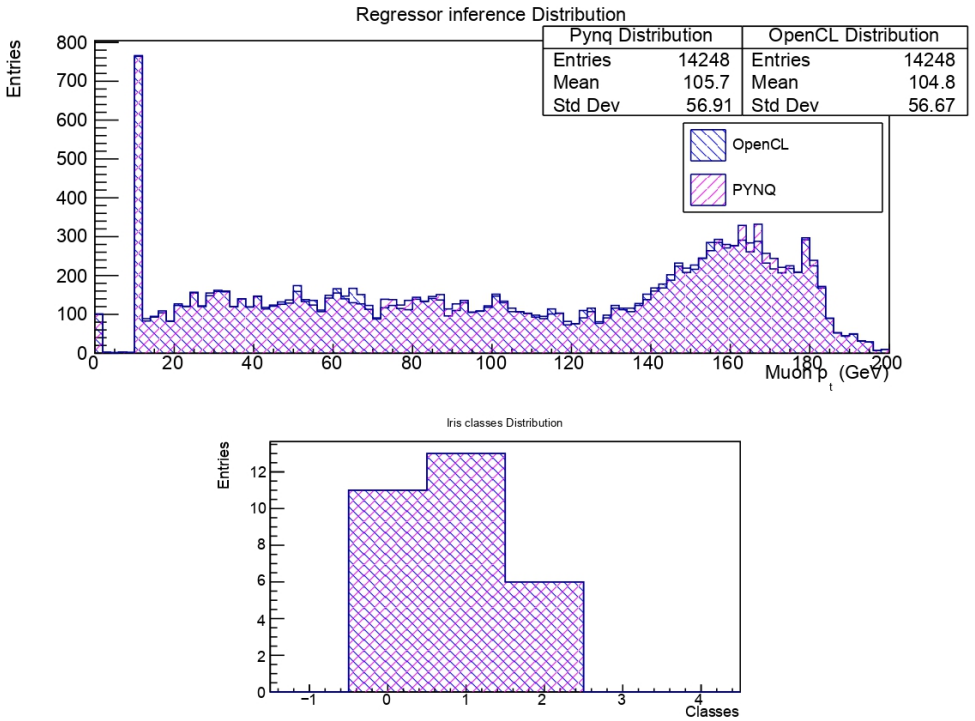




Inference comparison

In both the regressor and classifier case the **models' output** have been **validated against** the **OpenCL implementation**:

- Regressor: **small difference** traceable to **different** implementation of **floating** point to **fixed** point **conversion**;
- Classifier: **difference** in outputs **not enough** to **change class** assignement.



Summary and conclusions

- This work is still in **progress** (i.e. kernel optimization);
- The possibility of **deploying** a **Neural Network** on a **FPGA** inside an **AWS instance** has been explored;
- A **fast** and **easy-to-use** alternative to host applications written in **OpenCL** has been found in **PYNQ** using the **Python** programming language;
- There seems to be **no important drawbacks** from using this new approach.

Thank you!

Backup

Dataset to train and test the NN

The entire **dataset** contains about **300000 simulated muons** with a range in p_T **from 3 to 200 GeV/c**. Based on **trigger primitives**, a set of **information** is included in order to **predict** the muon p_T:

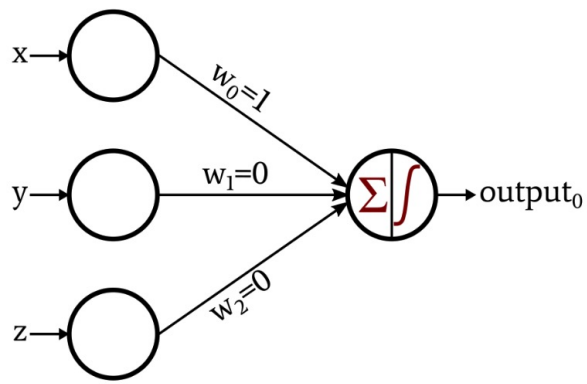
- **Primitives' position** (wheel, sector, ϕ) for each station crossed by the particle;
- **Direction** of the primitives in CMS global coordinates (ϕ_b).
- Trigger primitives' **quality** (i.e. number of hits used to build a TP).

To perform the analysis a **train/test split** of **80%/20%** was performed.

Artificial Neural Networks

The p_T assignment is currently carried out using **precompiled LUTs**. An alternative was explored using **Artificial Neural Network** (ANN):

- An ANN is a network designed to tackle **non-linear learning problems**;
- The **Fully Connected Multilayer Perceptrons** (MLPs) are made up of single units called *Perceptrons*;
- Perceptrons can be stacked together to **build arbitrarily deep custom networks**;
- The NN *learns* during the **training** process by receiving **input patterns** together with the corresponding true target variable and finding the **best set of weights**;
- The weights are used to **predict the output with unseen data**.



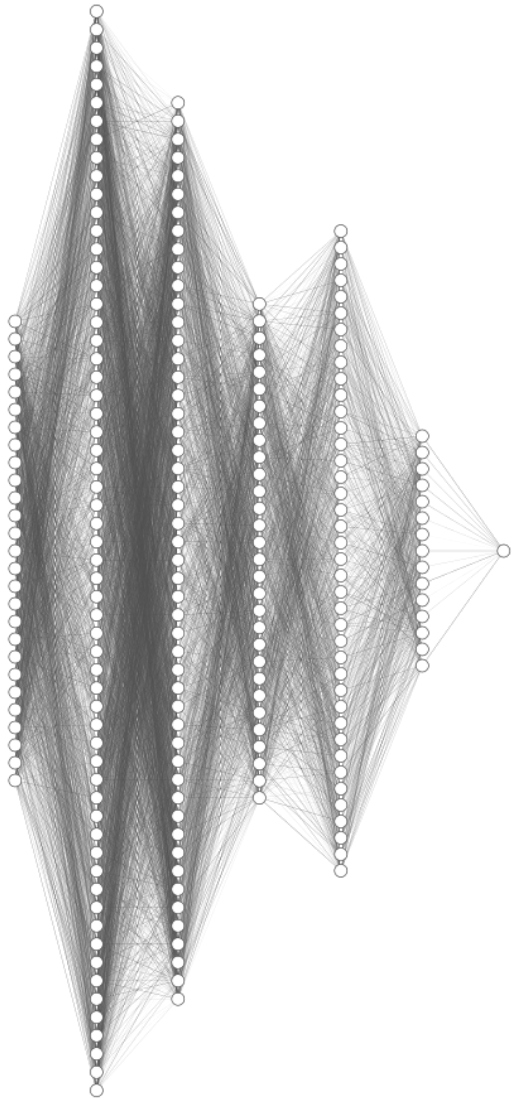
Graphical representation of a Perceptron.

Neural Network for regression

A **Fully Connected MLP** was built using QKeras with:

- **Input layer**: 27 features;
- **6 hidden layers**: 35, 20, 25, 40, 20, 15 nodes;
- **Output layer**: returns the p_T value.
- **Activation function**: Rectified Linear Unit;
- **Weight pruned**.

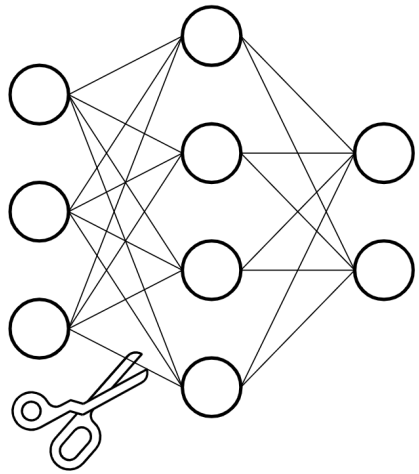
The model was **tested using a consumer CPU** before the hardware implementation.



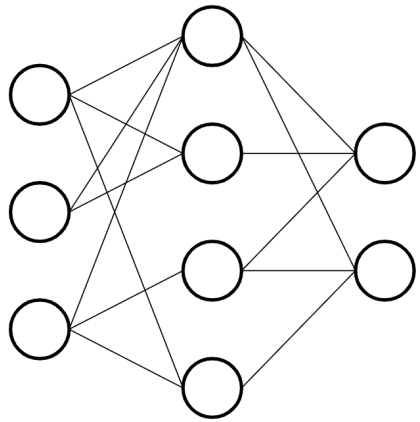
Optimization techniques

To produce an **optimized** **NN** for **implementation** on an FPGA:

- **Quantization**:
  the parameters were converted **from double precision floating-points to fixed points** to exploit the efficiency of DSPs;
- **Pruning**: **connections**
  between nodes with low influence were **cut** to **minimize** the number of **paramaters** and operations during inference and **reduce the resources** needed for implementation.

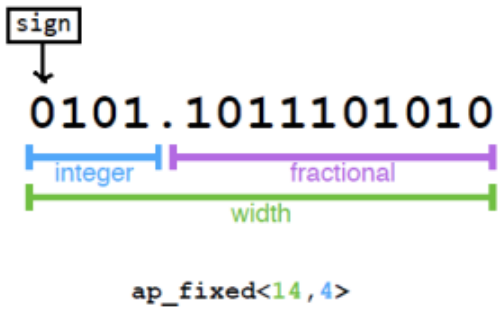


Before pruning



After pruning

Quantization



In order to produce an **optimized NN** for **implementation** on an FPGA, the models were *quantized*:

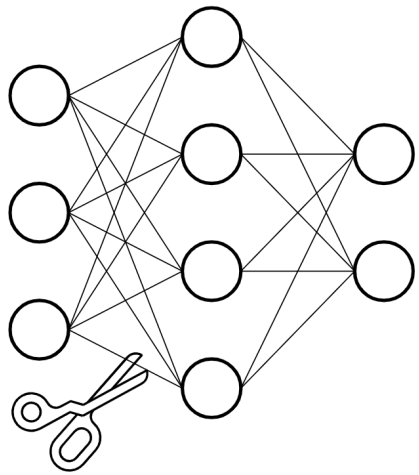
- *Quantization* is the conversion **from high-precision floating-points to normalized low-precision integers** (*fixed-point*) parameters;
- *QKeras* is a Python package developed as a collaboration between Google and HEP researchers to **build NN with quantized parameters**;
- It has an easy-to-use API: there are **drop-in replacements** for the most common layers used with Keras (e.g. Dense $\rightarrow$ QDense).

```
QDense(64, kernel_quantizer = quantized_bits(6,0),
          bias_quantizer = quantized_bits(6,0)(x))
QActivation('quantized_relu(6,0)')(x)
```

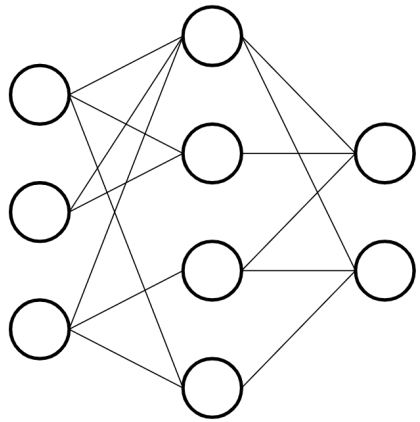
Slimming techniques - Weight Pruning

When building a NN model,
the final hardware platform where the inference computation will be run, has to be considered.

- *Weight Pruning* is the elimination of unnecessary values in the weight tensor;
- Connections between nodes with low influence are "cut" during the synthesis of the HLS design;
- This is aimed at minimizing the number of parameters and operations involved in the inference computation.



Before pruning



After pruning

Iris classifier