

# Pandora for developers

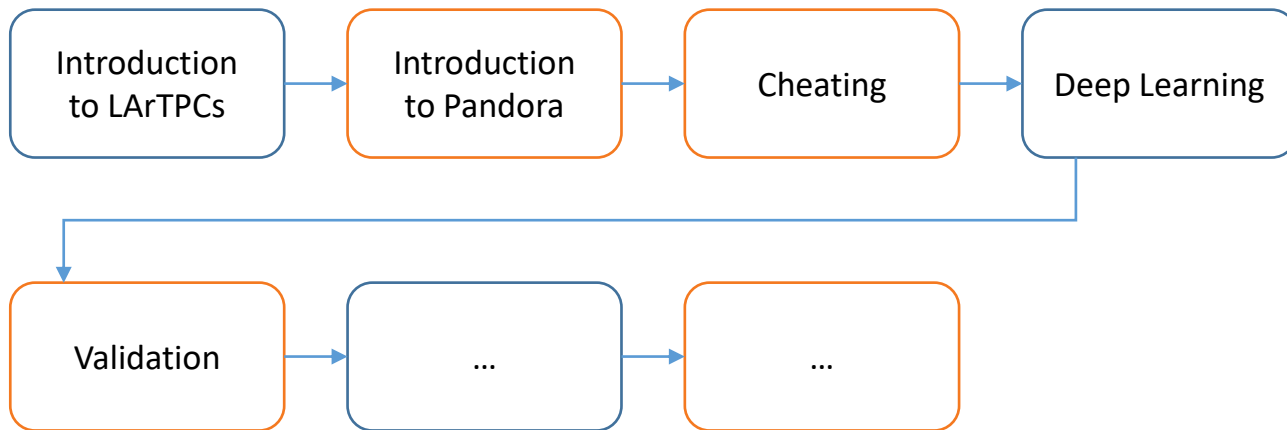
Andy Chappell for the Pandora team

---

24/03/2023

Pandora Workshop

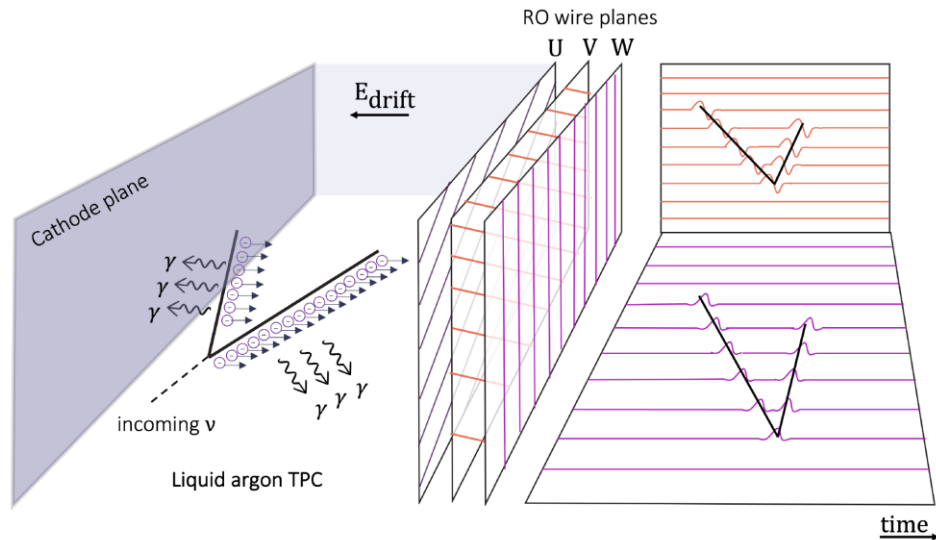
# Overview



Key references: [Pandora ProtoDUNE paper](#)  
[Pandora MicroBooNE paper](#)

# LArTPC detectors

- Pandora attempts to reconstruct 3D particle hierarchies from (processed) LArTPC detector outputs
- Charged particles deposit ionization trails in liquid argon
- Ionization electrons drift in an applied electric field
- Electrons are detected by a series of readout planes
  - Wires in horizontal drift (shown)
  - PCB strips in vertical drift
- LArTPC experiments using Pandora:
  - ICARUS, ProtoDUNE-SP/DP/HD/VD
  - MicroBooNE, SBND, DUNE HD, DUNE VD
  - DUNE NDLAr (not discussed here – native 3D readout)



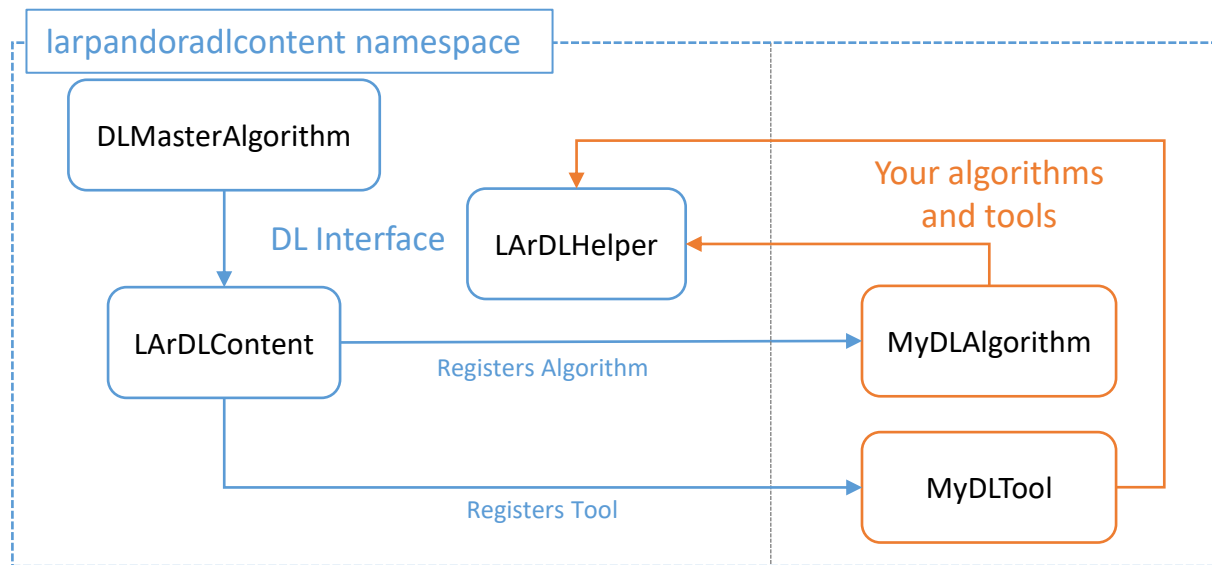
Credit: Maria Brigida Brunetti

# Pandora deep learning

- Pandora uses a multi-algorithm approach:
  - Build up events gradually
  - Each step is incremental - aim not to make mistakes (undoing mistakes is hard...)
  - Deploy more sophisticated algorithms as picture of event develops
  - Build physics and detector knowledge into algorithms
- Historically algorithms could be:
  - “Traditional”, hand-tuned by developers
  - Machine learning – Support Vector Machines, Boosted Decision Trees
- Since LArContent v03\_22\_00 we’ve added PyTorch deep neural network support
  - No need for either/or approach to algorithmic and deep learning – we can do both
  - Deep networks integrated into the algorithm workflow – still target specific problems anywhere in the reconstruction chain
  - Vertexing via deep network introduced to the horizontal drift accelerator neutrino workflow in v04\_00\_00
  - Expansion to other workflows since

# Pandora deep learning interface

- Access to LibTorch features provided by a lightweight interface in Pandora
  - Build up events gradually



## Main Steps


1. Create your algorithm/tool.
2. Use helper interface to initialize tensor inputs, load your model, and run inference.
3. Write code to prepare files for training (the single step that happens outside of Pandora).
4. Write code to populate your tensor inputs and read the inference tensor outputs.
5. Register your algorithm/tool with LArDLContent.

## Registering algorithms and tools

- As for non-deep learning algorithms (and tools) in LArContent, all algorithms must be registered with LArDLContent in order to be available for use
  - This is very simple to do, requiring two, single-line additions to LArDLContent
  - If you attempt to run a new algorithm and you see an error concerning algorithm registration, you probably forgot this step (the alternative is that you used LArMaster in your XML instead of LArDLMaster, or you haven't built Pandora with deep learning support enabled at all)
- So, how does this work?
  - Open LArDLContent.cc (in the larpandoradlcontent folder) and add the following two lines

```
#include "larpandoradlcontent/LArSuitableFolder/DlSuitableNameForAlgorithm.h"
...
#define LAR_DL_ALGORITHM_LIST(d) \
    d("LArDLMaster", DLMasterAlgorithm) \
    ...
    d("LArDLSuitableNameFor", DlSuitableNameForAlgorithm) \
    d("LArDLVertexing", DlVertexingAlgorithm)
```

These backslashes matter  
if you aren't inserting at  
the end



- You'll see some clear ordering of the various files based on location and alphabetic order, please maintain this for readability (clang format will enforce part of this for you, but is prohibited from formatting the macros)

## Using networks in Pandora

- We're going to start by jumping straight to the end of the process; using an already trained network in Pandora
- We first want you to get a sense of how things plumb into the Pandora workflow before we worry about the details of training a network
- We'll begin by introducing you to the DLHelper class and then show a few examples of the key bits of code you'll need to write to make use of your networks in Pandora
- We can't cover every use case, as the range of architectures, inputs and outputs is large and growing, but we can indicate the principles that we hope will translate across your various use cases
- Once we've covered that, we'll return to the topic of preparing training samples, training a network in Python (this will not be a tutorial on deep learning in Python, it will just note the workflow from Pandora to Python and back to Pandora) and converting the Python network to TorchScript format

# The DLHelper class

- The DLHelper class is composed of four type definitions and three functions
  - The typedefs act to provide a more convenient descriptor to a number of key torch types

```
typedef torch::jit::script::Module TorchModel;  
typedef torch::Tensor TorchInput;  
typedef std::vector<torch::jit::IValue> TorchInputVector;  
typedef at::Tensor TorchOutput;
```

- TorchModels hold network architectures and their associated weights
- TorchInputs describe the (arbitrary) tensor structure to be passed to a network at inference, while a TorchInputVector, as you might imagine, holds some number of these input tensors
- TorchOutputs describe the tensor structure returned by the network at inference



# The DLHelper class

- The [DLHelper](#) class is composed of four type definitions and three functions
  - The functions provide support for loading models, initialising inputs and running inference

```
static pandora::StatusCode LoadModel(const std::string &filename, TorchModel &model);
static void InitialiseInput(const at::IntArrayRef dimensions, TorchInput &tensor);
static void Forward(TorchModel &model, const TorchInputVector &input, TorchOutput &output);
```

- These functions are very simple
  - LoadModel accepts a filename (or full path\*) and returns (via the model reference argument) the loaded model, returning a StatusCode indicating whether or not the load was successful
  - InitialiseInput accepts a target dimensionality for an input tensor (more later) and outputs that tensor, filled with zeros via the tensor reference argument
  - Forward takes a TorchModel and vector of TorchInputs, performs the model inference step and returns the output via the output reference variable
- Why so lightweight?
  - The simplicity means **any** TorchScript compatible architecture is supported
  - However, this does mean you have some work to do to populate inputs and prepare training samples

\* LArFileHelper::FindFileInPath(modelName, "FW\_SEARCH\_PATH");

## Preparing inputs for inference

- All PyTorch networks accept a tensor as input, but the format of that tensor depends entirely on your architectural choice for the network
- As a result, when you populate a tensor prior to running the inference step, you need to specify its dimensionality and then fill its elements according to the needs of your network
  - This is what the `InitialiseInput` function is for
  - Let's assume you require a 4D tensor input, with dimensions representing batch number, channel number, image height, and image width
  - We'll assume a batch size of 1, a channel count of 1 and a 256 x 256 pixel image and a declared `TorchInput` variable named `networkInput`
  - This should be initialised via

```
LArDLHelper::InitialiseInput({1, 1, 256, 256}, networkInput); pandora::StatusCode
```

- At this point you have an input tensor of dimension (1, 1, 256, 256) filled with zeros, so now we need to fill it with meaningful input

## Preparing inputs for inference

- The details of what actually goes into your tensor depend entirely on your use case, but the process of filling the input tensor is simple enough

- First you need to get the accessor for your tensor, to get write access to the elements

```
auto accessor{networkInput.accessor<float, 4>()};
```

- You'll note you had to indicate the underlying type of your tensor and its dimensionality, in this case it's a float with four dimensions, but your requirements may be different
  - Then you need to fill those elements, which will look something like this

```
accessor[b][c][z][x] += adc;
```

- Here we assume the indices for batch, channel, z pixel index, and x pixel index are indicated by variables corresponding to their respective initial and that we are incrementing the element according to some ADC value. It is hopefully clear how you would modify this to your use case

## Running the model inference step

- In the previous slide we populated a single tensor, but the inference step expects a vector of such tensors, so we need to provide one (even if it only has one tensor in it)
  - We declare a suitable vector and then add our previously created tensor to it

```
LArDLHelper::TorchInputVector inputs;  
inputs.emplace_back(networkInput);
```

- Now we can declare our output tensor and run the inference step

```
LArDLHelper::TorchOutput output;  
LArDLHelper::Forward(model, inputs, output);
```

## Accessing the model output

- Now we need to access the outputs, which is conceptually similar to accessing input elements when populating the input tensor
  - Get the accessor for the output tensor (here we assume a one-dimensional float output, e.g. some value per batch element, like you might see with a simple classification network)

```
auto outputAccessor{output.accessor<float, 1>()};
```

- Now we can access an individual element, *i*, in that tensor via

```
const float value{outputAccessor[i]};
```

- Note that this just gets the raw network output
  - This might be all you need, but
  - You may have to do some additional post-processing to determine a class ID for example
  - It's not always as easy as you might hope to determine how to apply certain operations
  - It's very important that you understand what your network provides as output versus any interpretation (e.g. the argmax inset example) that might be required and whether you should apply this directly to the output tensor or the accessor derived from the output tensor

Sometimes you need to perform an operation directly on the output tensor. Here's an example involving argmax (the index over which you run, and the resultant tensor dimensionality may vary)

```
auto classes{torch::argmax(output, 1)};  
auto classesAccessor{classes.accessor<long, 3>()};
```

## Preparing training samples

- So far, we've considered how to prepare inputs, run inference and access outputs for an already trained model, but how do we train that model in the first place?
- The first thing to note is that there is currently no specific interface for this, though a template pattern is likely to be available in the future
- There is however a recommended approach
  - Preparation of training samples should be handled in the same class (algorithm) as inference
  - Whether the algorithm is to run in inference mode or training sample preparation mode should be handled by a simple flag that can be set via the XML parameters using the standard ReadSettings function
  - The standard Run function can then choose an appropriate execution path according to the mode

```
StatusCode DlVertexingAlgorithm::Run()  
{  
    if (m_trainingMode)  
        return this->PrepareTrainingSample();  
    else  
        return this->Infer();  
  
    return STATUS_CODE_SUCCESS;  
}
```

```
StatusCode DlVertexingAlgorithm::ReadSettings(  
    const TiXmlElement xmlHandle)  
{  
    PANDORA_RETURN_RESULT_IF_AND_IF(STATUS_CODE_SUCCESS,  
        STATUS_CODE_NOT_FOUND, !=, XmlHelper::ReadValue(  
            xmlHandle, "TrainingMode", m_trainingMode));  
  
    ...  
  
    return STATUS_CODE_SUCCESS;  
}
```

## Preparing training samples

- The details of how you produce your training output will depend upon what you need to provide to your network for training, but the default format for that output is a comma separated value (CSV) file (if you particularly want a different format you'll need to provide this yourself – but be aware that we try to minimize external dependencies in Pandora)
- A single training sample can be produced using the `MvaFeatureVector` class, where each feature constituting a sample can be added to a vector (of doubles), before this vector is **appended** to the specified output file

```
LArMvaHelper::MvaFeatureVector featureVector;  
featureVector.emplace_back(static_cast<double>(sampleValue1));  
featureVector.emplace_back(sampleValue2);  
...  
LArMvaHelper::ProduceTrainingExample(trainingFilename, true, featureVector);
```

Flag as signal



- The format of the output is a datetime stamp, then each of the features in the sample, followed by a signal/background flag (which may, or may not be relevant to your use case)

## Training in Python

- Once you have your CSV file(s) from the previous step, it's time to leave Pandora and switch to a Python environment to train your network
- This isn't intended to be a deep learning tutorial, so we'll assume you know how to design and/or load a network architecture and train it, so we've just cover those issues directly relevant to Pandora
- Typically, your network will want either images or torch tensors as input, so you'll likely need to convert the CSV inputs to the appropriate format – Python has numerous options supporting CSV file processing
- The key issue from a Pandora perspective is the need for a LibTorch 1.6 TorchScript network
  - In practice this means you'll need to train your network using nothing newer than PyTorch 1.6 (for now), or at least be able to convert a newer network into something compatible with PyTorch 1.6
  - At present the model to be used for inference must be a C++, CPU bound model
  - You can, and should train on a GPU, but you'll need to convert the output...



# Making a TorchScript model

- Conversion to a TorchScript model is quite simple and can be performed within your Python script
- There are, however, some basic requirements
  - No conditional execution paths in your network, it needs to be deterministic
  - (actually, conditional execution paths can be supported with tracing, but we won't discuss this here)
  - The model and tensors must be moved from GPU to CPU
- The code that handles these requirements is as follows:

```
torch.set_default_tensor_type(torch.FloatTensor)
Input_filename = f"my_trained_gpu_model.pkl"
output_filename = f"my_output_model.pt"
device = torch.device('cpu')
model.load_state_dict(torch.load(input_filename, map_location=device))
model.eval()
sm = torch.jit.script(model)
sm.save(output_filename)
```

Make tensors non-CUDA for CPU

Register a CPU-bound device

Load the model weights  
(Assumes the model is already declared)

Convert to TorchScript

## That's it

- The TorchScript network that you created in the previous slide can now be run inside a Pandora deep learning algorithm
- All that remains is to put the network in the right place
  - LArMachineLearningData/PandoraNetworkData in Pandora standalone – please don't try to commit model files like these to git
  - The suitable data path in LArSoft – e.g. \$DUNE\_PARDATA\_DIR/PandoraNetworkData for DUNE
  - In both cases you'll need to communicate with the core Pandora development team and the data repository manager in the relevant experiment if you want to release the network for general use, you likely won't have the necessary permissions to do this yourself
- If you're just trying to test in the LArSoft context pre-release, you can just create the PandoraNetworkData folder in the current working directory and ensure that the current working directory is part of the FW\_SEARCH\_PATH environment variable and Pandora will pick it up