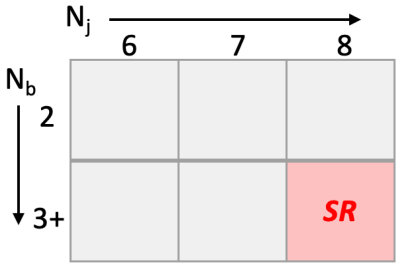
Instruction for ABCDnn

(Hayoung Oh from Korea Univ. haoh@cern.ch)

Example root files

You can test the ABCDnn code immediately using example root files in the **data** directory. 'source_ttbar.root' and 'source_qcd.root' are example of MC root files. 'target_data.root' is an example of real data. The example files are in flat ntuple format which consists of a tree name with 'Events'. In the files, there are nJet and nbJet branches which can be used as the control variables in this example and also other feature variables which can be used as input variables for ABCDnn. Also, a branch of xsecWeight is a kind of event weight. In this example, control variables are used to divide the following control regions (CRs) and signal region (SR):



The region with nJet = 8 and nbJet>=3 is signal region. Other 5 regions are CRs.

Implement using Google Colab

You can easily implement ABCDnn using a cloud based GPU through Google Colab. In Corlab, you can open the code written in the ipynb file, the Jupiter Notebook file format, to execute the code per cell and see the results immediately.
(Introduction for Colab: https://colab.research.google.com/)

First step.

Upload the 'Implement_ABCDnn.ipynb', example root files and other python codes to Google drive. Especially for example files, create a folder name with 'data' and upload it there. (If you want to upload them to another directory, specify the path in line2 of **Input Setting**.) Open the 'Implement_ABCDnn.ipynb' file through colab. The code is divided into several sections, most of them are modularized in *.py file and are called from the main code. Their role is described on the last page of this instruction. When you click 'Run all' at Runtime on the top bar or press the play button that appears when you place the cursor on each cell in order, the code will work using the data file already prepared. All input values are already set so that they can be operated immediately without any modifications. You can test the code by changing the inputs or by changing the data files. Most of the descriptions below are about the input setting.

In line6 of the second cell of Input Setting section, set the variables to be used as input. Since nJet and nbJet are used to divide CRs and SR, they are essential. In addition, enter other variables for which you want to predict the shape. At least one other variable is required. Edit the lists of *iscategorical* and *upperlimit* according to the variables. (line 6-8)

In lines40 and 65, the source and target files are opened in *prepdata()*. You can add some event selections for the input variables in *mcsel*, *datasel*. (line42-44, 67-69)

Set *mc_weight* according to how to consider the event weight of the source sample. You can set it as one of the inputs of function *train_and_validate()* in section **Run**. It should be set to None, “weight”, or “unweight”.
If *mc_weight* is None, the event weight of the result file is stored as 1.
If *mc_weight* is "weight", the event weight of the result file is stored the weight in xsecWeight in the source file.
If *mc_weight* is "unweight", the events of source sample are copied follow the ratio of weight stored in branch of xsecWeight to give weight during training. The weight of the resulting file is stored as 1. (If the cross section weight consists of many values or the proportion of the values is too large, it cause too many replicated samples and can be too slow.) You can test this option using ‘source_qcd.root’ which has different values of xsecWeight as source file.

In line15 of **Train and Validation** section, *inputdim* should be set by the number of input variables except for control variables. Each element should be a list consist of a label of plot, name of the input variable, the start point of the x-axis and endpoint of x-aixs.

In line 1-9 of **Run** section, set the hyperparameter used in training :

- NDense : number of nodes in hidden layers
- minibatch : size of minibatch
- lr : learning rate of *SawToothLearningSchedule()*
- gap : 1 period of *SawToothLearningSchedule()*
- beta1 : beta1 for Adam optimizer (Sould be greater than 0, lesser than 1)
- beta2 : beta2 for Adam optimzier (Sould be greater than 0, lesser than 1)
- Nafdim : number of nodes in hidden layers for updating the weight and bias of NAF
- Depth : number of iterations of permutations when construct NAF
- seed : random seed

Set *sdir* in line11 with the path where the results are stored. (The current setting automatically creates a folder called ABCDnn.)

For other inputs for *train_and_validate()*, set the *step* with the step size what you want. *retrain* and *train* decides whether retrain the existing checkpoint or not.
If retrain=True and train=True, existing checkpoints are ignored and retrained. (overwritten)
If retrain=False and train=True, the training continues from the existing checkpoint.
If retrain=Flase and train=False, the existing checkpoint is called and only validation is applied without training.

Set *mc_weight* with None, “weight”, or “unweight”. The resulting histogram is saved as a file name which is start with *title_step*.

Result files

The following is a description of the files generated as a result of code execution.

*_linear_ratio.pdf : Those plots shows the distributions of the morphed source and the target of the feature variables entered into the input. The orange colored histogram shows the distribution of the target, and the blue histogram is the distribution morphed through ABCDnn. The bottom is ratio plot shows predict(blue) / true(orange). If the number of the file name is 1,2,3, it represents the CRs with (Jet=7 & nbJet=2), (nJet=7 & nbJet>=3), (nJet=8 & nbJet=2) respectively. These are the distributions of the CRs used for training. (The CRs with nJet=6 were used for training, but did not plot). If the number of the file name is 4, it represents distribution of SR. Note that the SR was not used for training. If you want to draw those plot in log scale, change the *yscales* in line140 in Section **Train and Validation**.

result_*.root : This is the root file where the result of ABCDnn is saved in flat ntuple format. The name of tree is 'mytree' and the morphed variables are saved in branches. Note that the number of events before and after morphing is maintained because ABCDnn uses invertible transformation. In branch of nbjet, there are values of 2 or 3, but nbJet=3 actually means 3 or more bjet. For branch of weight, 1 or source's weight is stored according to *mc_weight*.

result_correl*.pdf : Those are the scattering plots consist of points with (x,y) = (before morphing, after morphing). The meaning of number in file's name:

1 : CRs with nJet=7 & nbJet=2
2 : CRs with nJet=7 & nbJet>=3
3 : CRs with nJet=8 & nbJet=2
4 : SR with nJet=8 & nbJet>=3

result_matrix_*_linear2.pdf : Those plot show distributions of feature variables before (green) and after morphing(orange). The target distribution is blue points.

trainingperf.pdf : This graph shows gloss and mmd loss as training progresses.

abcdnn.pkl : Monitor file

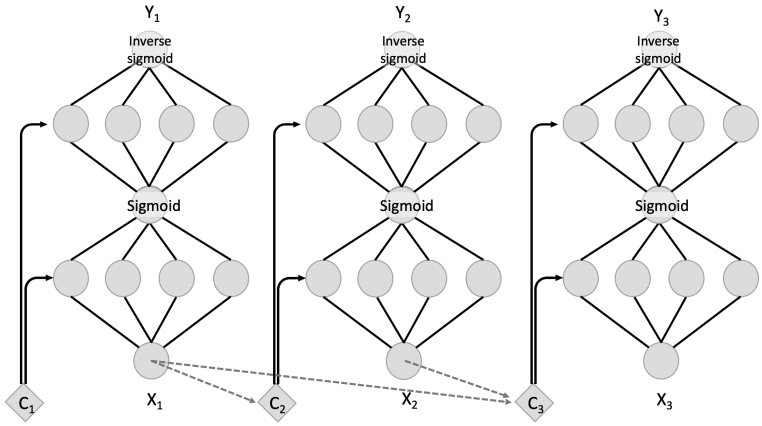
hyperparams.pkl : Saved hyperparameter in pkl file

checkpoint, ckpt*.data-*, ckpt*.index : Files for checkpoint. ckpt files are kept up to 10 according to line33 in section **ABCDnn**, and old ones are erased to conserve drive storage. The most recent checkpoint is the one with a large number.

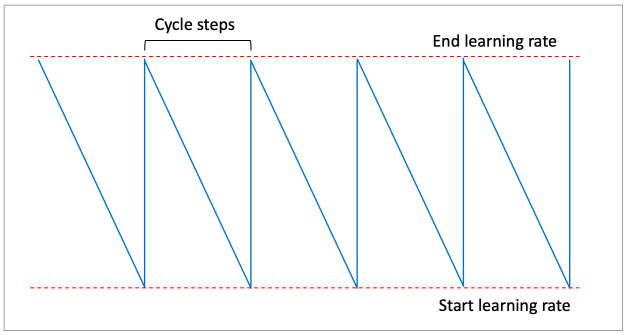
Tasks for each section

- Preparation
  - mount to google drive
  - import tensorflow libraries
  - install uproot, scikit-hep

- Construct NAF
  - construct NAF structure



- Learning Rate Scheduling
  - define SawtoothSchedule



Sawtooth learning rate scheduling

- MMD Loss
  - define the mmd loss
- ABCDnn
  - set real data, mc
  - store hyperparameters
  - share batch
  - training
  - mornitoring and save check point
- Onehot Enoding
  - onehot encoding, decoding
- Input Setting
  - read data and MC files
- Train and Validation
  - training, validation
  - save result, draw histograms.
- Run
  - set hyperparameters
  - running the *train_and_validate()*