Table of Contents

Introduction

Sorry for typo errors in the copy-paste code, please look at the full code on the git repository:

- https://gitlab.cern.ch/angianni/mltool

paths to ntuples:

- ATLAS UI Napoli:
  - `/data3/agiannini/DBLCode_19nov19/TreeAnalysis_MLUpdate_05mar19/TreeAnalysis_InputJet`
  - `/data3/agiannini/DBLCode_19nov19/TruthDAOD/xAODReader/run/xMLTutorial_20jul19/`
- lxplus:
  - /afs/cern.ch/work/a/angianni/public/MLTutorial_22jul19/Trees
  - /afs/cern.ch/work/a/angianni/public/MLTutorial_22jul19/TruthTrees

Presequisites

- python

- https://cernbox.cern.ch/index.php/s/28dmy3XhLfDMifl

- https://cernbox.cern.ch/index.php/s/C0ZZaUVkx7CfD7w

- numpy
- https://www.datacamp.com/community/tutorials/python-numpy-tutorial?utm_source=adwords_ppc&utm_cam
- https://docs.scipy.org/doc/numpy/user/quickstart.html

- pandas
- https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html
- https://www.datacamp.com/community/tutorials/pandas-tutorial-dataframe-python

- uproot
- https://github.com/scikit-hep/uproot
- https://indico.cern.ch/event/686641/contributions/2894906/attachments/1606247/2548596/pivarski-uproot.pdf
- https://indico.cern.ch/event/686137/contributions/2814517/attachments/1575846/2488415/pivarski-1.pdf
- https://indico.cern.ch/event/694818/contributions/2986392/attachments/1683028/2704704/pyhep-pres.pdf

- miniconda
- https://docs.conda.io/en/latest/miniconda.html

How to install miniconda and keras

wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh

`bash Miniconda3-latest-Linux-x86_64.sh=====`

logout and back in, now conda should work

conda create -n AIenv

conda update conda

```
conda config --set auto_activate_base False
```

conda activate AIenv (if you have problem try only: activate AIenv)

=conda install keras scikit-learn pandas numpy theano matplotlib seaborn pydot colorama

=pip install scikit-hep (if you have problem with pip version please update the python version using the command: scl enable rh-python36 bash)

=pip install uproot

Usefull links

Machine Learning Forum suggestions:

- https://atlasml.web.cern.ch/atlasml/

Dinos @ Exotics Workshop 2018 Roma

- https://gitlab.cern.ch/kbachas/RomeExoticsWorkshopHandsOnML
- https://vidyoreplay.cern.ch/replay/showRecordingExternal.html?key=pm22wcZIWQTcJhr

RNN lecture

- slazebni.cs.illinois.edu/spring17/lec20_rnn.pdf

Michela Paganini tutorial

- https://github.com/YaleATLAS/CERNDeepLearningTutorial/blob/master/deeplearning_intro.ipynb

ML chat with Silvia/Claudio

- https://cernbox.cern.ch/index.php/s/V4zham0BgxQMWV0

lwtnn: how to use keras-model into C++ based framework

- https://github.com/lwtnn/lwtnn
- https://zenodo.org/record/3249317#.XTWXlFBS-qA

- ♦ Set arr = ->
  ♦ Set l = <
  ♦ Set r = >

Session1: Deep Neural Network (DNN)

Read Trees in ROOT files

Let's start to read the Trees stored into ROOT files we have. It is enought to know the path to ROOT files, the name of the files and the name of the tree we would like to read.

```
#!/bin/python

import os, re, pickle

import uproot

import pandas as pd<br /><br />

path = '/data3/agiannini/DBLCode_19nov19/TreeAnalysis_MLUpdate_05mar19/TreeAnalysis_InputJetsOR_2

files = ['FlatTree_ggFH1000_spin0.root', 'FlatTree_Diboson_spin0.root']

for f in files:

print(f)

file_name = path + '/' + f

print(file_name)

  if os.path.isfile( file_name ):

theFile = uproot.open(file_name)

theKeys = theFile.keys()

print(theKeys)

tree = theFile['nominal']

DF = tree.pandas.df()

print(DF)
```

Manage Panda

Given the dataframe from the tree, we can simply add new informations we need, for example, add a new columns where we can fix if the sample will be used for category 0 or 1 of the binary classification.

```
### Get Pandas DataFrame from tree

DF = tree.pandas.df()

if 'ggFH' in f : DF['Category'] = 1

else: DF['Category'] = 0

print(DF)
```

if you need you can do a pre-selection on the events you really need for the traning of the model:

```
### Do Pre-Selection

print ('{:<20} {:<15}'.format('Events before preselection:', Fore.BLUE+str(DF.shape[0])))
```

```
presel = (DF.isGGFMergedPresel==1 ) #& ()

DF = DF[ presel ]

print ('{:<20} {:<15}'.format('Events after preselection:', Fore.BLUE+str(DF.shape[0])))
```

we need to include a python library in order to use the nice color style 🙂

```
from colorama import init, Fore, Back, Style

init(autoreset=True)
```

It could be VERY usefull to write this dataframe to a pickle file or a numpy array file, for example in order to avoid to re-read the original file several times.

```
### Save DF as numpy or pickle file

file_name_simple = f

file_name_simple = file_name_simple.replace('FlatTree_', '')

file_name_simple = file_name_simple.replace('_spin0.root', '')

DF.to_pickle( path_output + file_name_simple + '_FullNoRandom.pkl')<br /><br />
```

Train/Test splitting

Given the dataframes directly from the tree, it could be usefull to shuffle the events there before start to work with them (think at how the different Z+jets slices are distributed into the trees...); of course it is NOT really needed here. We need the scikit-learn library to perform the shuffling.

```
import sklearn.utils

...

### Shuffle the events

X = sklearn.utils.shuffle(DF, random_state=123) #'123' is the random seed
```

Once we shuffled the events, we can start to split the samples in Test/Train sub-datasets.

```
### Split in Train/Test samples

print ('{:><hr />20} {:<15} {:<15}'.format('Number of events',Fore.GREEN+str(X.shape[0])," Splitt

Ntrain_stop = int(round(X.shape[0] * 0.7))

X_Train = X[:Ntrain_stop]

X_Test = X[Ntrain_stop:]

print ('{:><hr />20} {:<15}'.format('Train events',Fore.GREEN+str(X_Train.shape[0])))

print ('{:><hr />20} {:<15}'.format('Test events',Fore.GREEN+str(X_Test.shape[0])+'\n' ))
```

Now, let's think that we have more trees that we would like to use as samples of the Category 0 of our binary problems and more trees for the Category 1. The simpler way to work with them is just to put together all the dataframes of the Category0 and of the Category1.

```
### Manage Train/Test DFs already created

samples_categ0 = ['Diboson']

samples_categ1 = ['ggFH1000']

X_train_categ0 = []

X_test_categ0 = []

X_train_categ1 = []

X_test_categ1 = []<br /><br />

for f in files:

   for i in samples_categ0:

           if i in f:

                 print ('{:><hr />20} {:<15}'.format('Reading DataFrame ', Fore.GREEN+str(path_ou

                 X_train = pd.read_pickle( path_output + i + '_Train.pkl')

                 X_train_categ0 += [X_train]

                 print ('{:><hr />20} {:<15}'.format('Reading DataFrame ', Fore.GREEN+str(path_ou

                 X_test = pd.read_pickle( path_output + i + '_Test.pkl')

                 X_test_categ0 += [X_test]

for i in samples_categ1:

   if i in f:

        print ('{:><hr />20} {:<15}'.format('Reading DataFrame ', Fore.GREEN+str(path_output + i

        X_train = pd.read_pickle( path_output + i + '_Train.pkl')

        X_train_categ1 += [X_train]

        print ('{:><hr />20} {:<15}'.format('Reading DataFrame ', Fore.GREEN+str(path_output + i

        X_test = pd.read_pickle( path_output + i + '_Test.pkl')

        X_test_categ1 += [X_test]

DF_train_categ0 = pd.concat(X_train_categ0)

DF_train_categ1 = pd.concat(X_train_categ1)

DF_test_categ0 = pd.concat(X_test_categ0)

DF_test_categ1 = pd.concat(X_test_categ1)

DF_train_categ0.to_pickle( path_output + 'DF_Train_categ0.pkl')

DF_train_categ1.to_pickle( path_output + 'DF_Train_categ1.pkl')

DF_test_categ0.to_pickle( path_output + 'DF_Test_categ0.pkl')

DF_test_categ1.to_pickle( path_output + 'DF_Test_categ1.pkl')
```

now, put togheter and shuflle the events of the 2 categories:

```
### Mix together the 2 categories

DF_train = X_train_categ0 + X_train_categ1

DF_train = pd.concat(DF_train)

DF_train = sklearn.utils.shuffle(DF_train, random_state=123) #'123' is the random seed

DF_test = X_test_categ0 + X_test_categ1

DF_test = pd.concat(DF_test)

#DF_test = sklearn.utils.shuffle(DF_test, random_state=123) # it is not really needed shuffle the
```

finally, we can bulid the X and y arrays we need in order to perform the training of out model:

```
### build X and y for the training

VariablesToModel = [

'fat_jet_pt', 'fat_jet_eta', 'fat_jet_phi', 'fat_jet_E', 'l1_pt', 'l1_eta', 'l1_phi', 'l1_e', 'l2

X_train = DF_train[VariablesToModel].as_matrix()

y_train = DF_train['Category'].as_matrix()

X_test = DF_test[VariablesToModel].as_matrix()

y_test = DF_test['Category'].as_matrix()

DF_train[VariablesToModel].to_pickle( path_output + 'X_Train.pkl')

DF_train['Category'].to_pickle( path_output + 'y_Train.pkl')

DF_test[VariablesToModel].to_pickle( path_output + 'X_Test.pkl')

DF_test['Category'].to_pickle( path_output + 'y_Test.pkl')
```

Look at input features and features scaling

The last step before training the model is to perform the scaling of the input features (and why not give a look to them if not done already).

For make quick plots of the input feature use:

```
variable = 'fat_jet_pt'

bins = 50

a = 0

b = 1000

plt.hist(DF_train_categ0[variable], bins=bins, range=[a,b], histtype='step', lw=2, alpha=0.5, lab

plt.hist(DF_train_categ1[variable], bins=bins, range=[a,b], histtype='step', lw=2, alpha=0.5, lab

plt.hist(DF_test_categ0[variable], bins=bins, range=[a,b], histtype='stepfilled', lw=2, alpha=0.5

plt.hist(DF_test_categ1[variable], bins=bins, range=[a,b], histtype='stepfilled', lw=2, alpha=0.5

plt.ylabel('Norm. Entries')

plt.xlabel(variable)
```

```
plt.legend(loc="upper right")
plt.savefig(path_output+"/plot_" + variable + ".pdf")
plt.clf()
```

Include the sklear library:

```
from sklearn import preprocessing
from sklearn.preprocessing import StandardScaler, LabelEncoder
```

and do:

```
scaler = preprocessing.StandardScaler().fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

Build model: dense layers and dropout

We are now ready to build our architecture and try the training of the model; first include:

```
from keras.models import Sequential,Model
from keras.layers.core import Dense, Activation
from keras.layers import BatchNormalization, Dropout, concatenate
from keras.callbacks import ModelCheckpoint, EarlyStopping
from keras.optimizers import Adam
```

and then build the model:

```
print (Fore.BLUE+"--------------------------------")
print (Back.BLUE+"      BUILDING DNN MODEL      ")
print (Fore.BLUE+"--------------------------------")
N_input = 8 # number of input variables
width = 24 # number of neurons for layer
dropout = 0.3
depth = 3 # nuber of hidden layers
model = Sequential()
model.add(Dense(units=width, input_dim=N_input))
model.add(Activation('relu'))
model.add(Dropout(dropout))
for i in range(0, depth):
   model.add(Dense(width))
```

```
    model.add(Activation('relu'))
    model.add(Dropout(dropout))
model.add(Dense(1, activation='sigmoid'))<br /><br />
```

and perform the traning:

```
print (Fore.BLUE+"------------------------------")
print (Back.BLUE+"      TRAIN DNN MODEL      ")
print (Fore.BLUE+"------------------------------")
model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=['accuracy'])
from collections import Counter
cls_ytrain_count = Counter(y_train)
print(cls_ytrain_count)
Nclass = len(cls_ytrain_count)
print ('{:<25}'.format(Fore.BLUE+'Training with class_weights because of unbalance classes !!'))
wCateg0 = (cls_ytrain_count[1] / cls_ytrain_count[0])
wCateg1 = 1.0
print(Fore.GREEN+'Weights to apply:')
print('{:<15}{:<15}'.format('Category0',round(wCateg0, 3)))
print('{:<15}{:<15}'.format('Category1',wCateg1))
callbacks = [
    # if we don't have a decrease of the loss for 4 epochs, terminate training.
    EarlyStopping (verbose=True, patience=7, monitor='val_loss'),
    # Always make sure that we're saving the model weights with the best val loss.
    ModelCheckpoint ( path_output+'/model.h5', monitor='val_loss', verbose=True, save_best_only=Tr
modelMetricsHistory = model.fit(
    X_train,
    y_train,
class_weight={
0 : wCateg0,
1 : wCateg1},
    epochs=100,
    batch_size=2048,
    validation_split=0.2,
    callbacks=callbacks,
```

```
    verbose=1
)<br /><br />
```

Output score and ROC curve

```
### Evaluate the model on Train/Test
X_train_categ0 = X_train[y_train==0]
X_train_categ1 = X_train[y_train==1]
print ('Running model prediction on Xtrain_categ0')
yhat_train_categ0 = model.predict(X_train_categ0, batch_size=2048)
print ('Running model prediction on Xtrain_categ1')
yhat_train_categ1 = model.predict(X_train_categ1, batch_size=2048)
print ('Running model prediction on Xtrain')
yhat_train = model.predict(X_train, batch_size=2048)<br /><br />
X_test_categ0 = X_test[y_test==0]
X_test_categ1 = X_test[y_test==1]
print ('Running model prediction on Xtest_categ0')
yhat_test_categ0 = model.predict(X_test_categ0, batch_size=2048)
print ('Running model prediction on Xtest_categ1')
yhat_test_categ1 = model.predict(X_test_categ1, batch_size=2048)
print ('Running model prediction on Xtest')
yhat_test = model.predict(X_test, batch_size=2048)<br /><br />
bins=np.linspace(0,1, 50)
plt.hist(yhat_train_categ0,    bins=bins, histtype='step',      lw=2, alpha=0.5, label=[r'Categor
plt.hist(yhat_train_categ1,    bins=bins, histtype='step',      lw=2, alpha=0.5, label=[r'Categor
plt.hist(yhat_test_categ0,    bins=bins, histtype='stepfilled',      lw=2, alpha=0.5, label=[r'Ca
plt.hist(yhat_test_categ1,    bins=bins, histtype='stepfilled',      lw=2, alpha=0.5, label=[r'Ca
plt.ylabel('Norm. Entries')
plt.xlabel('DNN score')
plt.legend(loc="upper center")
plt.savefig(path_output+"/MC_Data_TrainTest_Score.pdf")
plt.clf()
```

Now, let's plot the ROC curve.

```
### Make ROC Curves
w_test = DF_test['weight']
```

```
w_train = DF_train['weight']

fpr_w, tpr_w, thresholds_w = roc_curve(y_test, yhat_test, sample_weight=w_test)

roc_auc_w = auc(fpr_w, tpr_w, reorder=True)

print ('{:<35} {:<25.3f}'.format(Fore.GREEN+'ROC AUC weighted',roc_auc_w))

fpr_train_w, tpr_train_w, thresholds_train_w = roc_curve(y_train, yhat_train, sample_weight=w_tra

roc_auc_train_w = auc(fpr_train_w, tpr_train_w, reorder=True)

print ('{:<35} {:<25.3f}'.format(Fore.GREEN+'ROC AUC weighted',roc_auc_train_w))

plt.plot(fpr_w, tpr_w, color='darkorange', lw=2, label='Full curve Test (area = %0.2f)' % roc_auc

plt.plot(fpr_train_w, tpr_train_w, color='c', lw=2, label='Full curve Train (area = %0.2f)' % roc

plt.plot([0, 0], [1, 1], color='navy', lw=2, linestyle='--')

plt.xlim([-0.05, 1.0])

plt.ylim([0.0, 1.05])

plt.ylabel('True Positive Rate (weighted)')

plt.xlabel('False Positive Rate (weighted)')

plt.title('ROC curve for Category1 vs Category0')

plt.legend(loc="lower right")

plt.savefig(path_output + "/ROC_weighted.png")

plt.clf()
```

Reorganise the code in a (simple) class-code

main --> https://gitlab.cern.ch/angianni/mltool/blob/master/CodeForSession2/runDNN.py

helper tool --> https://gitlab.cern.ch/angianni/mltool/blob/master/CodeForSession2/Helpers.py

Session2: Deep Neural Network (DNN)

Accuracy and Loss functions

The Accuracy is defined as:

The Loss function is defined as:

Add this very simple class (or just the code if you prefer):

```
def plotTrainPerformance(path_output, modelMetricsHistory):

plt.plot(modelMetricsHistory.history['acc'])

plt.plot(modelMetricsHistory.history['val_acc'])

plt.title('Model accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend(['Train', 'Val'], loc='lower right')

plt.savefig(path_output + "/Accuracy.png")

plt.clf()

# summarize history for loss

plt.plot(modelMetricsHistory.history['loss'])

plt.plot(modelMetricsHistory.history['val_loss'])

plt.title('Model loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend(['Train', 'Val'], loc='upper right')

plt.savefig(path_output + "/Loss.png")

plt.clf()
```

Features ranking: permutation importance method

Look at the Eli5 libray:

https://eli5.readthedocs.io/en/latest/index.html

https://eli5.readthedocs.io/en/latest/blackbox/permutation_importance.html?highlight=permutator

add it if there is time.

First, install the library:

conda install -c conda-forge eli5

Use the methods:

```
from eli5.permutation_importance import get_score_importances

def MyNNScore (X, y):
    modelpath = 'run_test/'
    model = load_model(modelpath+'/model.h5')
    yhat = model.predict(X, verbose = True, batch_size=2048)
    return yhat
def DoFeaturesRanking (path_output, X, y, variables):
    base_score, score_decreases = get_score_importances(MyNNScore, X, y)
    feature_importances = np.mean(score_decreases, axis=0)
    fpr0, tpr0, thresholds0 = roc_curve(y, base_score)
    AUC0 = auc(fpr0, tpr0)
    score = base_score - score_decreases<br />
    #print(score.shape)
    nRandom = score.shape[0]
    nVars = score.shape[1]
    nEvents = score.shape[2]<br /><br />   AUC = np.zeros( (nRandom, nVars) )
    for i in range(0, nRandom):
        for j in range(0, nVars):
            score_all = score[i][j][:]
            score_1 = score_all[y==1]
            score_0 = score_all[y==0]
            fpr, tpr, thresholds = roc_curve(y, score_all)
            AUC[i][j] = auc(fpr, tpr)
    AUCs = np.mean(AUC, axis=0)
    AUCs_min = np.min(AUC, axis=0)
    AUCs_max = np.max(AUC, axis=0)
    AUCs_error = ( AUCs_max-AUCs_min) / 2
    plt.figure()
    AUC0_vec = np.zeros( ( len(variables) ) )
    AUC0_vec[:] = AUC0
    plt.plot(variables, AUC0_vec, label='AUC0')
    plt.errorbar(variables, AUCs, yerr=AUCs_error, label='AUC variables')
```

```
plt.legend(loc="upper right")

plt.xlabel('')

plt.ylabel('AUC')

plt.savefig(path_output + 'plot_FR_AUC.pdf')

AUC_mean = np.zeros( (nVars) )

plt.figure()

for j in range(0, nVars):

    score_all = np.mean( score, axis=0 )

    score_all = score_all[j][:]

    print(score_all.shape)

    score_1 = score_all[y==1]

    score_0 = score_all[y==0]

    label = ', Var' + str(j)

    label = 'Removing ' + variables[j]

    plt.hist(score_1, range=[0,1], bins=50, label=label, histtype='step', normed=False)

    fpr, tpr, thresholds = roc_curve(y, score_all)

   AUC_mean[j] = auc(fpr, tpr)

  #   plt.plot(fpr, tpr, label=label)

plt.legend(loc="upper right")

plt.xlabel('DNN Score')

plt.ylabel('Events')

plt.savefig(path_output + 'plot_FR_scores.pdf')

plt.figure()

plt.plot(variables, AUC0_vec, label='AUC0')

#plt.plot(variables, 1-(AUC_mean/AUC0_vec), label='Features Weight')

plt.plot(variables, (1-AUC_mean)/(1-AUC0_vec), label='Model Inefficiency')

plt.legend(loc="upper right")

plt.xlabel('')

plt.ylabel('Rank')

plt.savefig(path_output + 'plot_FR_Rank.pdf')

return base_score, score_decreases, feature_importances<br /><br />
```

Example DNN1: Test a trained model over new datasets

The idea is that now the new dataset will be only-test events and we would like load the model we have and get the model prediction on these new samples.

We need to load the model and the scaler we use during the training phase:

```
from keras.models import load_model

...

def LoadDNNModel (modelpath):

   if not os.path.isfile(modelpath + '/model.h5'):

        print(Fore.RED + 'Model path does not exist')

        quit()

   print(Back.BLUE + "Loading model at path " + modelpath)

model = load_model(modelpath+'/model.h5')

   scaler = joblib.load(modelpath + '/scaler.save')

return model, scaler
```

do not forget to include the saving of the scaler during the training phase:

```
=from sklearn.externals import joblib=

=...=

=### Save the scaler=

=scaler_file = "scaler.save"=

=joblib.dump(scaler, scaler_filename)=
```

now, we should just convert the ROOT files, manage the dataframes (this time only features selection and scaling) and load the model. Use the function:

```
def DNNResultsOnlyTest (path_output, model, scaler, samples, VariablesToModel):

### read the DFs

for s in samples:

print ('{:><hr />20} {:<15}'.format('Reading DataFrame ', Fore.GREEN+str(path_output + s + '_Full

DF = pd.read_pickle( path_output + s + '_FullNoRandom.pkl')

X = DF[VariablesToModel].as_matrix()

y = DF['Category'].as_matrix()

        ### Features scaling

X = scaler.transform(X)

        ### Evaluate the model on Test

print ('Running model prediction on Xtest')

yhat = model.predict(X, batch_size=2048)
```

```
        bins=np.linspace(0,1, 50)
plt.hist(yhat,    bins=bins, histtype='step',      lw=2, alpha=0.5, label=[r'' + s ],    normed=T
### add the score to the DF
DF['DNNScore'] = yhat
DF.to_pickle( path_output + s + '_WithScore.pkl')
plt.ylabel('Norm. Entries')
plt.xlabel('DNN score')
plt.legend(loc="upper center")
plt.savefig(path_output+"/MC_Data_NewSamples_Score.pdf")
plt.clf()
```

so, we can use these method in the main:

```
#!/bin/python
import os, re, pickle
import Helpers as h
path = '/data3/agiannini/DBLCode_19nov19/TreeAnalysis_MLUpdate_05mar19/TreeAnalysis_InputJetsOR_2
files = ['FlatTree_ggFH1000_spin0.root',
         'FlatTree_ttBar_spin0.root', 'FlatTree_Diboson_spin0.root', 'FlatTree_Zjets_spin0.root',
         'FlatTree_Data15_spin0.root', 'FlatTree_Data16_spin0.root'
]
path_output = 'run_test/'
if not os.path.exists(path_output):
    os.makedirs(path_output)
###
BranchesToRead = [
   'weight', 'NJETS', 'isGGFMergedPresel',
   'fat_jet_pt', 'fat_jet_eta', 'fat_jet_phi', 'fat_jet_E',
   'l1_pt', 'l1_eta', 'l1_phi', 'l1_e',
   'l2_pt', 'l2_eta', 'l2_phi', 'l2_e',
   ]
h.ConvertROOTFiles(path, files, path_output, BranchesToRead)
###
samples_categ0 = ['Diboson']
samples_categ1 = ['ggFH1000']
```

```
#DF_train, DF_test = h.PrepareDF(path, files, path_output, samples_categ0, samples_categ1)<br /><
VariablesToModel = [
   'fat_jet_pt', 'fat_jet_eta', 'fat_jet_phi', 'fat_jet_E',
   'l1_pt', 'l1_eta', 'l1_phi', 'l1_e',
   'l2_pt', 'l2_eta', 'l2_phi', 'l2_e',
   ]
#X_train, X_test, y_train, y_test = h.PrepareDFForTraining(path_output, VariablesToModel, DF_trai
###
#model = h.DNNBuilder()
#model, modelMetricsHistory = h.DNNTrainer(path_output, model, X_train, y_train)
###
#h.DNNResults(path_output, model, X_train, X_test, y_train, y_test, DF_train, DF_test)
#h.plotTrainPerformance(path_output, modelMetricsHistory)
### Load model and Test
model, scaler = h.LoadDNNModel(path_output)
samples = ['Zjets', 'Diboson', 'ttBar', 'Data15', 'Data16', 'ggFH1000']
h.DNNResultsOnlyTest(path_output, model, scaler, samples, VariablesToModel)
```

it could be needed (and faster) read only a sub-set of branches from the ROOT files:

```
def ConvertROOTFiles (path, files, path_output, branchesToRead):
...
            DF = tree.pandas.df(branchesToRead)
```

Read output with scores and make plots (stack)

Use the very simple function:

```
def MakeStackPlot (path_output):
   Zjets = pd.read_pickle( path_output + 'Zjets_WithScore.pkl')
   Diboson = pd.read_pickle( path_output + 'Diboson_WithScore.pkl')
   ttBar = pd.read_pickle( path_output + 'ttBar_WithScore.pkl')
   Data15 = pd.read_pickle( path_output + 'Data15_WithScore.pkl')
   Data16 = pd.read_pickle( path_output + 'Data16_WithScore.pkl')
   d = [Data15, Data16]
   Data = pd.concat( d )
bins=np.linspace(0, 1, 50)
```

```
plt.hist( Zjets['DNNScore'],      bins=bins, stacked=True,      lw=2, alpha=0.5, label='Zjets'  ,

plt.hist( Diboson['DNNScore'],    bins=bins, stacked=True,      lw=2, alpha=0.5, label='Diboson'

plt.hist( ttBar['DNNScore'],      bins=bins, stacked=True,      lw=2, alpha=0.5, label='ttBar'  ,

_ = skh_plt.hist(Data['DNNScore'], bins=bins, errorbars=True, histtype='marker',label='Data', col

plt.yscale('log')

plt.ylabel('Entries')

plt.xlabel('DNN Score')

plt.legend(loc="upper right")

plt.savefig(path_output+"/StackPlot_Score.pdf")

plt.clf()<br />
```

Example DNN2: Parametrised-DNN

The architecture of the pDNN is the same of the DNN, there are just more input variables as the number of the parameters of the problem are. In our case, let's try to parametrised the truth mass of the signals. What we need is to use an extra variables to the DF and to the DNN inputs; let use:

```
def PrepareDFForTrainingPDNN (path_output, VariablesToModel, DF_train, DF_test):

   DF_train_0 = DF_train[ DF_train['Category']==0 ]

   DF_train_1 = DF_train[ DF_train['Category']==1 ]

   DF_train_1['TruthMass'] = DF_train_1['Sample']

   DF_train_1['TruthMass'] = DF_train_1.TruthMass.replace({'ggFH': ''}, regex=True)

   DF_train_1['TruthMass'] = pd.to_numeric(DF_train_1['TruthMass'])

   DF_train_0['TruthMass'] = np.random.choice( a=DF_train_1['TruthMass'], size=DF_train_0.shape[0

   DF_train = pd.concat( [DF_train_0, DF_train_1] )

   #print(DF_train_0['TruthMass'])

   #plt.figure()

   #plt.hist(DF_train_1['TruthMass'])

   #plt.hist(DF_train_0['TruthMass'])

   #plt.show()

   DF_test_0 = DF_test[ DF_test['Category']==0 ]

   DF_test_1 = DF_test[ DF_test['Category']==1 ]

   DF_test_1['TruthMass'] = DF_test_1['Sample']

   DF_test_1['TruthMass'] = DF_test_1.TruthMass.replace({'ggFH': ''}, regex=True)

   DF_test_1['TruthMass'] = pd.to_numeric(DF_test_1['TruthMass'])

   DF_test_0['TruthMass'] = np.random.choice( a=DF_test_1['TruthMass'], size=DF_test_0.shape[0] )

   DF_test = pd.concat( [DF_test_0, DF_test_1] )
```

```
    ### build X and y for the training
    X_train = DF_train[VariablesToModel].as_matrix()
    y_train = DF_train['Category'].as_matrix()
    X_test = DF_test[VariablesToModel].as_matrix()
    y_test = DF_test['Category'].as_matrix()
    DF_train[VariablesToModel].to_pickle( path_output + 'X_Train.pkl')
    DF_train['Category'].to_pickle( path_output + 'y_Train.pkl')
    DF_test[VariablesToModel].to_pickle( path_output + 'X_Test.pkl')
    DF_test['Category'].to_pickle( path_output + 'y_Test.pkl')
    ### Features scaling
    scaler = preprocessing.StandardScaler().fit(X_train)
    X_train = scaler.transform(X_train)
    X_test = scaler.transform(X_test)
    ### Save the scaler
    scaler_file = "scaler.save"
    joblib.dump(scaler, path_output + '/' + scaler_file)
    return X_train, X_test, y_train, y_test<br /><br />
```

once the model has been trained (the procedure is exactly the same of before, so just re-use the prefious methods) you can also test the model on all samples you like and in particular re-evaluate the model for each "truth value" of the parameter:

```
def PDNNResultsOnlyTest (path_output, model, scaler, samples, VariablesToModel):
    ### read the DFs
    for s in samples:
        print ('{:><hr />20} {:<15}'.format('Reading DataFrame ', Fore.GREEN+str(path_output + s +
        DF = pd.read_pickle( path_output + s + '_Train.pkl')
        masses = [300, 700, 1000, 2000, 3000]
        for i in masses:
            DF['TruthMass'] = i
            X = DF[VariablesToModel].as_matrix()
            y = DF['Category'].as_matrix()
            ### Features scaling
            X = scaler.transform(X)
            ### Evaluate the model on Test
            print ('Running model prediction on Xtest')
```

```
            yhat = model.predict(X, batch_size=2048)
            bins=np.linspace(0,1, 50)
            plt.hist(yhat,    bins=bins, histtype='step',      lw=2, alpha=0.5, label=[r'' + s + '
            ### add the score to the DF
            DF['DNNScore_' + str(i)] = yhat
    DF.to_pickle( path_output + s + '_WithScore.pkl')
    plt.ylabel('Norm. Entries')
    plt.xlabel('DNN score')
    plt.legend(loc="upper center")
    plt.savefig(path_output+"/MC_Data_NewSamples_Score.pdf")
    plt.clf()
```

make the function MakeStackPlot parametrised according the variable to plot:

```
def MakeStackPlot (path_output, variable):
  Zjets = pd.read_pickle( path_output + 'Zjets_WithScore.pkl')
  Diboson = pd.read_pickle( path_output + 'Diboson_WithScore.pkl')
  ttBar = pd.read_pickle( path_output + 'ttBar_WithScore.pkl')
  Data15 = pd.read_pickle( path_output + 'Data15_WithScore.pkl')
  Data16 = pd.read_pickle( path_output + 'Data16_WithScore.pkl')
  d = [Data15, Data16]
  Data = pd.concat( d )
  bins=np.linspace(0, 1, 50)
  plt.hist( Zjets[variable],      bins=bins, stacked=True,      lw=2, alpha=0.5, label='Zjets'   ,
  plt.hist( Diboson[variable],    bins=bins, stacked=True,      lw=2, alpha=0.5, label='Diboson'
  plt.hist( ttBar[variable],      bins=bins, stacked=True,      lw=2, alpha=0.5, label='ttBar'   ,
  _ = skh_plt.hist(Data[variable], bins=bins, errorbars=True, histtype='marker',label='Data', col
  plt.yscale('log')
  plt.ylabel('Entries')
  plt.xlabel('DNN Score')
  plt.legend(loc="upper right")
  plt.savefig(path_output+"/StackPlot_" + variable + ".pdf")
  plt.clf()
```

you can quickly plot all the scores in the different mass hypothesis you have:

```
h.MakeStackPlot(path_output, 'DNNScore_300')
```

```
h.MakeStackPlot(path_output, 'DNNScore_700')

h.MakeStackPlot(path_output, 'DNNScore_1000')

h.MakeStackPlot(path_output, 'DNNScore_2000')

h.MakeStackPlot(path_output, 'DNNScore_3000')
```

Session3: Recurrent Neural Network (RNN)

Recurrent input and variable-lenght input problem

Let's think to have the following problem. Measurements of the temperature T are collected at different time steps and the temperature is function of the time. The problem is if we have the first 9 time measurements is it possible to predict the 10th measurement? A (simple) RNN architecture is able to do it.

We can do a simulation of the temperatures datasets; let's consider that the temperature at a given timestep t is distributed as a gaussian with mean t*t and sigma 3% of the mean; for semplicity normalize the timestep to be in the range [0,1] (in this way we are guaranteed that the T values are normalised in the range [0,1]).

```
#!/bin/python

import os, re, pickle

import numpy as np

np.random.seed(123)

import sklearn.utils

import pandas as pd

from sklearn import preprocessing

from sklearn.preprocessing import StandardScaler, LabelEncoder

from keras.models import Sequential,Model,load_model

from keras.layers.core import Dense, Activation

from keras.layers import BatchNormalization, Dropout, GRU, concatenate, SimpleRNN, Masking, RNN,

from keras.callbacks import ModelCheckpoint, EarlyStopping

from keras.optimizers import Adam

from matplotlib import pyplot as plt

from sklearn.metrics import roc_curve, auc, roc_auc_score, classification_report, confusion_matri

from sklearn.externals import joblib

from skhep.visual import MplPlotter as skh_plt

from random import uniform, gauss, randint

nTimeSteps = 9

nEvents = 1000

dataset  = np.zeros( (nEvents, nTimeSteps, 1) )

dataset_t = np.zeros( (nEvents, nTimeSteps, 1) )

target_t = np.zeros( (nEvents, 1) )

target  = np.zeros( (nEvents, 1) )

for j in range(0, nEvents):

   Index = randint(0, nTimeSteps)
```

```
    for i in range(0, nTimeSteps):
        t = i/10 + 0.05
        mean = t*t
        sigma = 0.03 * t
        dataset_t[j][i][0] = t
        dataset[j][i][0]  = gauss(mean, sigma)
#       if i ><hr /> Index: dataset[j][i][0] = -99.
for j in range(0, nEvents):
    print( uniform(nTimeSteps, nTimeSteps+1) )
    t = nTimeSteps/10 + 0.05
    mean = t*t
    sigma = 0.03 * t
    target_t[j] = t
    target[j]  = gauss(mean, sigma)
dataset_draw = np.ma.array(dataset, mask=dataset == -99.)
print(dataset_draw.shape)
plt.figure()
plt.xlim([0., 1.])
logbins = np.logspace(-3, 0, 100)
plt.hist(dataset_draw[:, :, 0], bins=50, label='Temperature')
plt.xlabel('T')
plt.ylabel('Events')
plt.legend(loc="upper left")
plt.figure()
plt.ylim([0, 1])
plt.plot(dataset_t[:, :, 0], dataset_draw[:, :, 0], label='T measurements')
plt.plot(target_t , target, label='T target')
t = np.arange(0., 1., 0.01)
plt.plot(t, t**2, 'r--', label='T(t) = t*t')
plt.xlabel('time')
plt.ylabel('T(t)')
plt.legend(loc="upper left")
plt.show()
```

now, build a simple RNN LSTM-based model and perform the training on the dataset:

```
input = Input( (9, 1) )
mask = Masking(-99., name='jet_masking')(input)
net = LSTM(output_dim=1, name='net_gru', return_sequences=False, unroll=True)(mask)
model = Model( inputs=[input], outputs=net )
model.summary()
model.compile(loss='binary_crossentropy', optimizer='Adam', metrics=['mean_squared_error'])
callbacks = [
   # if we don't have a decrease of the loss for 4 epochs, terminate training.
   EarlyStopping (verbose=True, patience=7, monitor='val_loss'),
   # Always make sure that we're saving the model weights with the best val loss.
   #ModelCheckpoint( path_output+'/model.h5', monitor='val_loss', verbose=True, save_best_only=Tr
   ]
modelMetricsHistory = model.fit(
   dataset,
   target,
   epochs=20,
   batch_size=1,
   validation_split=0.2,
   callbacks=callbacks,
   verbose=1
   )
```

now, we can see how the predictions of the model are distributed both on the training dataset and on a new independent test dataset:

```
yhat = model.predict(dataset, batch_size=1)
print(yhat)
plt.figure()
plt.ylim([0, 1])
plt.plot(dataset_t[:, :, 0], dataset_draw[:, :, 0], label='T measurements')
plt.plot(target_t , yhat, label='T prediction')
t = np.arange(0., 1., 0.01)
plt.plot(t, t**2, 'r--', label='T(t) = t*t')
plt.xlabel('time')
plt.ylabel('T(t)')
```

```
plt.legend(loc="upper left")

test  = np.zeros( (nEvents, nTimeSteps, 1) )

test_t = np.zeros( (nEvents, nTimeSteps, 1) )

for j in range(0, nEvents):

   Index = randint(0, nTimeSteps)

   for i in range(0, nTimeSteps):

   print( uniform(i, i+1) )

   t = i/10 + 0.05

   mean = t*t

   sigma = 0.03 * t

       test_t[j][i][0] = t

       test[j][i][0]  = gauss(mean, sigma)

#       if i ><hr /> Index: test[j][i][0] = -99.

#       if i ><hr /> 7   : test[j][i][0] = -99.

yhat_test = model.predict(test, batch_size=1)

print(yhat_test)

test_draw = np.ma.array(test, mask=test == -99.)

plt.figure()

plt.ylim([0, 1])

plt.plot(test_t[:, :, 0], test_draw[:, :, 0], label='T measurements')

plt.plot(target_t , yhat_test, label='T prediction')

t = np.arange(0., 1., 0.01)

plt.plot(t, t**2, 'r--', label='T(t) = t*t')

plt.xlabel('time')

plt.ylabel('T(t)')

plt.legend(loc="upper left")

plt.show()
```

first, we can see that the model is able to predict the nTimeSteps+1 values (with also a smaller spread of the values), than we can play with:

- in the test dataset, remove some time-steps measurements, the model is still able to predict the 10th values?

- we can generalise the model training using a training dataset with some timesteps missing (use the line if i>Index: dataset[j][i][0] = -99.)

Manage input dataframes for jets as RNN inputs

The RNN model expects to have ad numpy array input a 3-dimensional array with the shape:

(nEventa, nTimeSteps, nFeatures)

so, we need to reshape our flat dataframe first; we can use the function:

```
def ReshapeForRNN (X, nTimeStepsRNN, nVariablesRNN):
    new_X = np.zeros( (X.shape[0], nTimeStepsRNN, nVariablesRNN) )
    for i in range(0, X.shape[0]):
        for j in range(0, X.shape[1]):
             if i%1000 == 0: print(i, j)
            new_X[i, int(j/nVariablesRNN), j%nVariablesRNN] = X.iloc[i, j]
        X = new_X
    return X
```

also the scaling is different for the RNN; do not forget that now will have some -99. values in our input dataframe

```
def ScalerForRNN (data, MaskValue, var_names, savevars, path_output, VAR_FILE_NAME='scaling'):
    '''
    Args:
    -----
        data: a numpy array of shape (nb_events, nb_particles, n_variables)
        var_names: list of keys to be used for the model
        savevars: bool -- True for training, False for testing
                  it decides whether we want to fit on data to find mean and std
                  or if we want to use those stored in the json file
    Returns:
    --------
        modifies data in place, writes out scaling dictionary
    '''
    scale = {}
    if savevars:
        for v, name in enumerate(var_names):
            print ('Scaling feature %s of %s (%s).' % (v + 1, len(var_names), name))
            f = data[:, :, v]
            # print(f)
```

```
            slc = f[f != MaskValue ]
            m, s = slc.mean(), slc.std()
            slc -= m
            slc /= s
            data[:, :, v][f != MaskValue ] = slc.astype('float32')
            scale[name] = {'mean' : float(m), 'sd' : float(s)}
    with open(path_output + '/' + VAR_FILE_NAME, 'wb') as varfile:
            pickle.dump(scale, varfile)
            varfile.close()
    else:
        f = open(path_output + '/' + VAR_FILE_NAME, "rb")
        varinfo = pickle.load(f)
        for v, name in enumerate(var_names):
            #print 'Scaling feature %s of %s (%s).' % (v + 1, len(var_names), name)
            f = data[:, :, v]
            slc = f[f != MaskValue ]
            m = varinfo[name]['mean']
            s = varinfo[name]['sd']
            slc -= m
            slc /= s
            data[:, :, v][f != MaskValue ] = slc.astype('float32')
```

so, we can build a function for preparing the DFs for the RNN training:

```
def PrepareDFForTrainingRNN (path_output, VariablesToModel, DF_train, DF_test, VariablesRNN, Mask
### build X and y for the training
y_train = DF_train['Category'].as_matrix()
y_test = DF_test['Category'].as_matrix()
### for RNN do extra operation on the DFs before convert to numpy
X_train = DF_train[VariablesToModel]
X_test = DF_test[VariablesToModel]
DF_train[VariablesToModel].to_pickle( path_output + 'X_Train.pkl')
DF_train['Category'].to_pickle( path_output + 'y_Train.pkl')
DF_test[VariablesToModel].to_pickle( path_output + 'X_Test.pkl')
DF_test['Category'].to_pickle( path_output + 'y_Test.pkl')
```

```
### Reshape the X dataframe
X_train = ReshapeForRNN (X_train, nTimeStepsRNN, nVariablesRNN)
X_test = ReshapeForRNN (X_test , nTimeStepsRNN, nVariablesRNN)
### Features scaling
ScalerForRNN (X_train, MaskValue, VariablesRNN, True , path_output, VAR_FILE_NAME='scaling')
ScalerForRNN (X_test , MaskValue, VariablesRNN, False, path_output, VAR_FILE_NAME='scaling')
np.save( os.path.join(path_output, "XReshaped_Train.npy") , X_train )
np.save( os.path.join(path_output, "XReshaped_Test.npy") , X_test )
np.save( os.path.join(path_output, "y_Train.npy") , y_train )
np.save( os.path.join(path_output, "y_Test.npy") , y_test )
return X_train, X_test, y_train, y_test
```

Example RNN: VBF to ggF classification

Now, it is time to build the RNN model, for example, the one we use in the VV semi-leptonic analysis:

```
def RNNBuilder(nTimeStepsRNN, nVariablesRNN, MaskValue):
print ('Building Simple RNN model')
JET_SHAPE = (nTimeStepsRNN, nVariablesRNN)
nRNNLayers = 1
DimRetSeqLayers = 25
DimNORetSeqLayers = 25
dropout = 0.3
jet_input = Input(JET_SHAPE)
jet_channel = Masking(MaskValue, name='jet_masking')(jet_input)
for i in range(0, nRNNLayers):
  LayerCounter1 = 1 + i
  if i==0: jet_channel = LSTM(input_dim=nVariablesRNN,  output_dim=DimRetSeqLayers, name='jet_lst
   $ else: jet_channel = LSTM(input_dim=DimRetSeqLayers, output_dim=DimRetSeqLayers, name='jet_ls
   jet_channel = Dropout(dropout, name='jet_dropout' + str(LayerCounter1) )(jet_channel)
   LayerCounter2 = LayerCounter1 + 1 + i
   jet_channel = LSTM(output_dim=DimNORetSeqLayers, name='jet_lstm' + str(LayerCounter2), return_
   jet_channel = Dropout(dropout, name='jet_dropout')(jet_channel)
   jet_channel = Dense(output_dim=1)(jet_channel)
   jet_channel_outputs = Activation('sigmoid')(jet_channel)
```

```
    combined_rnn = Model(inputs=[jet_input], outputs=jet_channel_outputs)
    combined_rnn.summary()
    return combined_rnn
```

and we can use the Builder function as for the DNN case, we can just use a new one for simplicity (if we need to adjust some parameters):

```
def RNNTrainer(path_output, model, X_train, y_train):
    ### Train the model
    print (Fore.BLUE+"-----------------------------")
    print (Back.BLUE+"      TRAIN RNN MODEL      ")
    print (Fore.BLUE+"-----------------------------")
    model.compile(optimizer=Adam(lr=0.001), metrics=['accuracy'], loss='binary_crossentropy' )
    from collections import Counter
    cls_ytrain_count = Counter(y_train)
    print(cls_ytrain_count)
    Nclass = len(cls_ytrain_count)
    print ('{:<25}'.format(Fore.BLUE+'Training with class_weights because of unbalance classes !!'
    wCateg0 = (cls_ytrain_count[1] / cls_ytrain_count[0])
    wCateg1 = 1.0
    print(Fore.GREEN+'Weights to apply:')
    print('{:<15}{:<15}'.format('Category0',round(wCateg0, 3)))
    print('{:<15}{:<15}'.format('Category1',wCateg1))
    callbacks = [
        # if we don't have a decrease of the loss for 4 epochs, terminate training.
        EarlyStopping (verbose=True, patience=7, monitor='val_loss'),
        # Always make sure that we're saving the model weights with the best val loss.
        ModelCheckpoint ( path_output+'/model.h5', monitor='val_loss', verbose=True, save_best_onl
    ]
    modelMetricsHistory = model.fit(
    [X_train],
    y_train,
    class_weight={
            0 : wCateg0,
            1 : wCateg1},
    epochs=30,
```

```
    batch_size=512,
    validation_split=0.2,
    callbacks=callbacks,
    verbose=1
    )
    return model, modelMetricsHistory
```

remember to change the preselection criteria in the ConvetROOTFiles function:

```
### Do Pre-Selection
print ('{:<20} {:<15}'.format('Events before preselection:', Fore.BLUE+str(DF.shape[0])))
#presel = (DF.isGGFMergedPresel==1 ) #& ()
presel = (DF.Topology!=2 ) & (DF.Jet1_pt><hr />0)
DF = DF[ presel ]
print ('{:<20} {:<15}'.format('Events after preselection:', Fore.BLUE+str(DF.shape[0])))
```

for the rest of the ML workflow we can basically re-use the functions we already did for the DNN standard and so use:

```
#!/bin/python
import os, re, pickle
import numpy as np<br /><br />
import Helpers as h<br /><br />
path = '/data3/agiannini/DBLCode_19nov19/TreeAnalysis_MLUpdate_05mar19/TreeAnalysis_InputJetsOR_2
files = ['FlatTree_VBFH1000_spin0.root', 'FlatTree_ggFH1000_spin0.root']
path_output = 'run_test_RNN/'
if not os.path.exists(path_output):
    os.makedirs(path_output)
###
BranchesToRead = [
    'weight', 'NJETS', 'Topology',
    'Jet1_pt', 'Jet1_eta', 'Jet1_phi', 'Jet1_E',
    'Jet2_pt', 'Jet2_eta', 'Jet2_phi', 'Jet2_E',
    'Jet3_pt', 'Jet3_eta', 'Jet3_phi', 'Jet3_E',
    'Jet4_pt', 'Jet4_eta', 'Jet4_phi', 'Jet4_E',
    'Jet5_pt', 'Jet5_eta', 'Jet5_phi', 'Jet5_E',
    'Jet6_pt', 'Jet6_eta', 'Jet6_phi', 'Jet6_E',
```

```
   ]
h.ConvertROOTFiles(path, files, path_output, BranchesToRead)
###
samples_categ0 = ['ggFH1000']
samples_categ1 = ['VBFH1000']
DF_train, DF_test = h.PrepareDF(path, files, path_output, samples_categ0, samples_categ1)
###
VariablesToModel = [
   'Jet1_pt', 'Jet1_eta', 'Jet1_phi', 'Jet1_E',
   'Jet2_pt', 'Jet2_eta', 'Jet2_phi', 'Jet2_E',
   'Jet3_pt', 'Jet3_eta', 'Jet3_phi', 'Jet3_E',
   'Jet4_pt', 'Jet4_eta', 'Jet4_phi', 'Jet4_E',
   'Jet5_pt', 'Jet5_eta', 'Jet5_phi', 'Jet5_E',
   'Jet6_pt', 'Jet6_eta', 'Jet6_phi', 'Jet6_E',
   ]
nTimeStepsRNN = 6
nVariablesRNN = 4
VariablesRNN = ['pt', 'eta', 'phi', 'E']
X_train, X_test, y_train, y_test = h.PrepareDFForTrainingRNN(path_output, VariablesToModel, DF_tr
### you can simply load the DF already ReSahped and Scaled the second time you run, withou runnin
X_train = np.load( path_output + '/XReshaped_Train.npy' )
X_test = np.load( path_output + '/XReshaped_Test.npy' )
y_train = np.load( path_output + '/y_Train.npy' )
y_test = np.load( path_output + '/y_Test.npy' )
###
model = h.RNNBuilder(nTimeStepsRNN, nVariablesRNN, -99.)
model, modelMetricsHistory = h.RNNTrainer(path_output, model, X_train, y_train)
###
h.DNNResults(path_output, model, X_train, X_test, y_train, y_test, DF_train, DF_test)
h.plotTrainPerformance(path_output, modelMetricsHistory)
```

Session 4: Beyond RNN and others architecutes

Multiclassification problem: spins classification

A very interesting multiclassification problem in the diboson resonant searches could be the classification of the spins of the resonance. In order to build this exercise let consider truth samples and consider the 4-momentum of the resonance X and of the two dauthers bosons particles VV.

Let use the main runMultiClassDNN.py:

```
#!/bin/python

import os, re, pickle

import Helpers as h

path = '/data3/agiannini/DBLCode_19nov19/TruthDAOD/xAODReader/run/xMLTutorial_20jul19/'

files = ['VBFH1000.root', 'VBFHVT1000.root', 'VBFRSG1000.root']

path_output = 'run_test_MultiClassDNN/'

if not os.path.exists(path_output):

   os.makedirs(path_output)

###

BranchesToRead = [

   'weight',

   'X_pT', 'X_eta', 'X_phi', 'X_E',

   'ZLL_pT', 'ZLL_eta', 'ZLL_phi', 'ZLL_E',

   'ZQQ_pT', 'ZQQ_eta', 'ZQQ_phi', 'ZQQ_E',

   ]

h.ConvertROOTFiles(path, files, path_output, BranchesToRead)

###

samples_categ0 = ['VBFH1000']

samples_categ1 = ['VBFHVT1000']

samples_categ2 = ['VBFRSG1000']

DF_train, DF_test = h.PrepareDF3Class(path, files, path_output, samples_categ0, samples_categ1, s

h.MakePlot3Class(path_output, DF_train, DF_test, 'X_pT', 50, 0, 1.e+6)

h.MakePlot3Class(path_output, DF_train, DF_test, 'X_E' , 50, 5.e+5, 5.e+6)

###

VariablesToModel = [

   'X_pT', 'X_eta', 'X_phi', 'X_E',

   'ZLL_pT', 'ZLL_eta', 'ZLL_phi', 'ZLL_E',
```

```
    'ZQQ_pT', 'ZQQ_eta', 'ZQQ_phi', 'ZQQ_E',
    ]
X_train, X_test, y_train, y_test = h.PrepareDFForTraining3Class(path_output, VariablesToModel, DF
###
model = h.DNNBuilder3Class()
model, modelMetricsHistory = h.DNNTrainer3Class(path_output, model, X_train, y_train)
###
h.DNNResults3Class(path_output, model, X_train, X_test, y_train, y_test, DF_train, DF_test)
h.plotTrainPerformance(path_output, modelMetricsHistory)
```

please, look at the dedicated "3Class" functions directly on the git repository 🙂.

parametrised-RNN and DNN-RNN convolution: future VBF-ggF classification?

In keras is possible to concatenate different layers using the concatenate layers. For example, it could be interesting to concatenate DNN and RNN.

Use the builder:

```
def ComboDNNRNNBuilder (nInputs, nTimeStepsRNN, nVariablesRNN, MaskValue):
    print (Fore.BLUE+"----------------------------------------")
    print (Back.BLUE+"      BUILDING COMBO DNN-RNN MODEL      ")
    print (Fore.BLUE+"----------------------------------------")
    ### Build RNN
    print ('Building Simple RNN model')
    #JET_SHAPE = Xjet_train.shape[1:]
    JET_SHAPE = (nTimeStepsRNN, nVariablesRNN)
    nRNNLayers = 1
    DimRetSeqLayers = 25
    DimNORetSeqLayers = 25
    dropout = 0.3
    jet_input = Input(JET_SHAPE)
    jet_channel = Masking(MaskValue, name='jet_masking')(jet_input)
<br />    for i in range(0, nRNNLayers):
        LayerCounter1 = 1 + i
        if i==0: jet_channel = LSTM(input_dim=nVariablesRNN,  output_dim=DimRetSeqLayers, name='je
```

```
        else:   jet_channel = LSTM(input_dim=DimRetSeqLayers, output_dim=DimRetSeqLayers, name='je
            jet_channel = Dropout(dropout, name='jet_dropout' + str(LayerCounter1) )(jet_channel)
    LayerCounter2 = LayerCounter1 + 1 + i
    jet_channel = LSTM(output_dim=DimNORetSeqLayers, name='jet_lstm' + str(LayerCounter2), return_
    jet_channel = Dropout(dropout, name='jet_dropout')(jet_channel)
    ### Build DNN
    #N_input = 13 # number of input variables
    width = 24 # number of neurons for layer
    dropout = 0.3
    depth = 3 # number of hidden layers
    lep_input  = Input( (nInputs,) )
    lep_channel = Dense(width, activation = 'relu')(lep_input)
    lep_channel = Dropout(dropout)(lep_channel)
    for i in range(0, depth):
        lep_channel = Dense(width, activation = 'relu')(lep_input)
        lep_channel = Dropout(dropout)(lep_channel)
    combined = concatenate([jet_channel, lep_channel])
    combined_outputs = Dense(1, activation='sigmoid')(combined)
    combined_net = Model(inputs=[lep_input, jet_input], outputs=combined_outputs)
    combined_net.summary()
    return combined_net
```

and use the main runComboDNNRNN.py:

```
#!/bin/python

<pre><br />import os, re, pickle
import numpy as np<br />import Helpers as h

<br />#path = '/data3/agiannini/DBLCode_19nov19/TreeAnalysis_MLUpdate_05mar19/TreeAnalysis_InputJ
path = '/data3/agiannini/DBLCode_19nov19/TreeAnalysis_MLUpdate_05mar19/TreeAnalysis_VTReaderRNN_F
files = ['FlatTree_VBFH1000_spin0.root', 'FlatTree_ggFH1000_spin0.root']<br /><br />path_output =
if not os.path.exists(path_output):
    os.makedirs(path_output)
<br />
###
BranchesToRead = [
    'weight', 'NJETS', 'Topology',
    'l1_pt', 'l1_eta', 'l1_phi', 'l1_e',
    'l2_pt', 'l2_eta', 'l2_phi', 'l2_e',
    'Jet1_pt', 'Jet1_eta', 'Jet1_phi', 'Jet1_E',
    'Jet2_pt', 'Jet2_eta', 'Jet2_phi', 'Jet2_E',
    'Jet3_pt', 'Jet3_eta', 'Jet3_phi', 'Jet3_E',
    'Jet4_pt', 'Jet4_eta', 'Jet4_phi', 'Jet4_E',
    'Jet5_pt', 'Jet5_eta', 'Jet5_phi', 'Jet5_E',
    'Jet6_pt', 'Jet6_eta', 'Jet6_phi', 'Jet6_E',
```

```
    ]
h.ConvertROOTFiles(path, files, path_output, BranchesToRead)

###
samples_categ0 = ['ggFH1000']
samples_categ1 = ['VBFH1000']
DF_train, DF_test = h.PrepareDF(path, files, path_output, samples_categ0, samples_categ1)

### Prepare DFs for recurrent inputs
VariablesToModel = [
    'Jet1_pt', 'Jet1_eta', 'Jet1_phi', 'Jet1_E',
    'Jet2_pt', 'Jet2_eta', 'Jet2_phi', 'Jet2_E',
    'Jet3_pt', 'Jet3_eta', 'Jet3_phi', 'Jet3_E',
    'Jet4_pt', 'Jet4_eta', 'Jet4_phi', 'Jet4_E',
    'Jet5_pt', 'Jet5_eta', 'Jet5_phi', 'Jet5_E',
    'Jet6_pt', 'Jet6_eta', 'Jet6_phi', 'Jet6_E',
    ]
nTimeStepsRNN = 6
nVariablesRNN = 4
VariablesRNN = ['pt', 'eta', 'phi', 'E']
X_train_rnn, X_test_rnn, y_train_rnn, y_test_rnn = h.PrepareDFForTrainingRNN(path_output, Variabl

### Prepare DFs for flat inputs
VariablesToModel = [
    'l1_pt', 'l1_eta', 'l1_phi', 'l1_e',
    'l2_pt', 'l2_eta', 'l2_phi', 'l2_e',
    ]
nInputs = 8
X_train_dnn, X_test_dnn, y_train_dnn, y_test_dnn = h.PrepareDFForTraining(path_output, VariablesT


#X_train = np.load( path_output + '/XReshaped_Train.npy' )
#X_test  = np.load( path_output + '/XReshaped_Test.npy' )
#y_train = np.load( path_output + '/y_Train.npy' )
#y_test  = np.load( path_output + '/y_Test.npy' )

###
model = h.ComboDNNRNNBuilder(nInputs, nTimeStepsRNN, nVariablesRNN, -99.)
model, modelMetricsHistory = h.RNNTrainer(path_output, model, [X_train_dnn, X_train_rnn], y_train

###
h.ComboDNNRNNResults(path_output, model, [X_train_dnn, X_train_rnn], [X_test_dnn, X_test_rnn], y_
h.plotTrainPerformance(path_output, modelMetricsHistory)
</pre>
```

look at the git page for all the dedicated methods .

Convolutional Neural Network (CNN): Jet Images

ask directly to me...

This topic: Sandbox > MachineLearningTutorialDeepNeuralNetworks
Topic revision: r27 - 2019-11-07 - AntonioGiannini