```
#Loading the packages for running the networks
import os
import keras
import math
import sys
from keras.models import Sequential, Model
from keras.layers import Dense, Input, BatchNormalization, Dropout
from keras import metrics
from keras.optimizers import SGD
from keras.losses import binary_crossentropy
import sklearn as skl
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_curve, auc
from sklearn.preprocessing import StandardScaler
#Loading the packages for handling the data
import uproot as ur
import pandas
import numpy as np
#Loading packages needed for plottting
import matplotlib.pyplot as plt
#Defining colours for the plots
#The colours were chosen using the xkcd guice
#color_tW = '#66FFFF'
color_tW = '#0066ff'
#color_tt = '#FF3333'
color_tt = '#990000'
color_sys = '#009900'
color_tW2 = '#02590f'
color_tt2 = '#FF6600'

#-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
---------------------


#Setting up the output directories
output_path = './Piet/'
array_path = output_path + 'arrays/'
if not os.path.exists(output_path):
    os.makedirs(output_path)
if not os.path.exists(array_path):
    os.makedirs(array_path)


plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0))
plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0))

#-----------------------------------------------------------------------------------
-----------------------------------------------------------------------------------
---------------------
#This is the main class for the adversarial neural network setup
class neuralNetworkEnvironment(object):

    def __init__(self):
        #At the moment not may variables are passed to the class. You might want to
change this
        #A list of more general settings
        self.variables = np.array(["mass_lep1jet2", "pTsys_lep1lep2met",
"pTsys_jet1jet2", "mass_lep1jet1", "deltapT_lep1_jet1", "deltaR_lep1_jet2",
"deltaR_lep1lep2_jet2", "mass_lep2jet1", "pT_jet2", "deltaR_lep1_jet1",
"deltaR_lep1lep2_jet1jet2met", "deltaR_lep2_jet2", "cent_lep2jet2",
"deltaR_lep2_jet1"])
        #The seed is used to make sure that both the events and the labels are
shuffeled the same way because they are not inherently connected.
        self.seed = 193
        #All information necessary for the input
        #The exact data and targets are set later
        self.input_path = "/cephfs/user/s6chkirf/work/area/run/test_ANNinput.root"
        self.signal_sample = "wt_nominal"
        self.background_sample = "tt_nominal"
        self.signal_tree = ur.open(self.input_path)[self.signal_sample]
        self.background_tree = ur.open(self.input_path)[self.background_sample]
        self.sample_training = None
        self.sample_validation = None
        self.target_training = None
        self.target_validation = None
        #Dimension of the variable input used to define the size of the first layer
        self.input_dimension = self.variables.shape
        #These arrays are used to save loss and accuracy of the two networks
        #That is also important to later be able to use the plotting software
desired. matplotlib is not the best tool at all times
        self.discriminator_history_array = []
        self.model_history_array = []
        self.discriminator_history = None
        #Here are the definitions for the two models
        #All information for the length of the training. Beware that epochs might
only come into the pretraining
        #Iterations are used for the adversarial part of the training
        #If you want to make the training longer you want to change these numbers,
there is no early stopping atm, feel free to add it
        self.discriminator_epochs = 10
        self.batchSize = 512
        #Setup of the networks, nodes and layers
        self.discriminator_layers = 4
        self.discriminator_nodes = 128
        #Setup of the networks, loss and optimisation
        self.discriminator_optimizer = SGD(lr = 0.1, momentum = 0.5)
        self.discriminator_dropout = 0.1
        self.discriminator_loss = binary_crossentropy

        self.validation_fraction = 0.4

        #The following set of variables is used to evaluate the result
        #fpr = false positive rate, tpr = true positive rate
        self.tpr = 0.  #true positive rate
        self.fpr = 0.  #false positive rate
        self.threshold = 0.
        self.auc = 0.  #Area under the curve


#Initializing the data and target samples
#The split function cuts into a training sample and a test sample
#Important note: Have to use the same random seed so that event and target stay in
the same order as we shuffle
    def initialize_sample(self):
        #Signal and background are needed for the classification task, signal and
systematic for the adversarial part
        #In this first step the events are retrieved from the tree, using the
```

```
chosen set of variables
        #The numpy conversion is redundant
        self.events_signal = self.signal_tree.pandas.df(self.variables).to_numpy()
        self.events_background =
self.background_tree.pandas.df(self.variables).to_numpy()
        #Setting up the weights. The weights for each tree are stored in
'weight_nominal'
        self.weight_signal =
self.signal_tree.pandas.df('weight_nominal').to_numpy()
        self.weight_background =
self.background_tree.pandas.df('weight_nominal').to_numpy()
        #Rehsaping the weights
        self.weight_signal = np.reshape(self.weight_signal,
(len(self.events_signal), 1))
        self.weight_background = np.reshape(self.weight_background,
(len(self.events_background), 1))
        #Normalisation to the eventcount can be used instead of weights, especially
if using data
        self.norm_signal = np.reshape([1./float(len(self.events_signal)) for x in
range(len(self.events_signal))], (len(self.events_signal), 1))
        self.norm_background = np.reshape([1./float(len(self.events_background))
for x in range(len(self.events_background))], (len(self.events_background), 1))
        #Calculating the weight ratio to scale the signal weight up. This tries to
take the high amount of background into account
        self.weight_ratio = ( self.weight_signal.sum())/
self.weight_background.sum()
        self.weight_signal = self.weight_signal / self.weight_ratio

        #Setting up the targets
        #target combined is used to make sure the systematics are seen as signal
for the first net in the combined training
        self.target_signal = np.reshape([1 for x in
range(len(self.events_signal))], (len(self.events_signal), 1))
        self.target_background = np.reshape([0 for x in
range(len(self.events_background))], (len(self.events_background), 1))
        #The samples and corresponding targets are now split into a sample for
training and a sample for testing. Keep in mind that the same random seed should be
used for both splits
        self.sample_training, self.sample_validation =
train_test_split(np.concatenate((self.events_signal, self.events_background)),
test_size = self.validation_fraction, random_state = self.seed)
        self.target_training, self.target_validation =
train_test_split(np.concatenate((self.target_signal, self.target_background)),
test_size = self.validation_fraction, random_state = self.seed)
        #Splitting the weights
        self.weight_training, self.weight_validation =
train_test_split(np.concatenate((self.weight_signal, self.weight_background)),
test_size = self.validation_fraction, random_state = self.seed)
        self.norm_training, self.norm_validation =
train_test_split(np.concatenate((self.norm_signal, self.norm_background)),
test_size = self.validation_fraction, random_state = self.seed)

        #Setting up a scaler
        #A scaler makes sure that all variables are normalised to 1 and have the
same order of magnitude for that reason
        scaler = StandardScaler()
        self.sample_training = scaler.fit_transform(self.sample_training)
        self.sample_validation = scaler.fit_transform(self.sample_validation)
```

```
#----------------------------------------------------------------------------------
----------------------------------------------------------------------------------
--------------------------------------
#Here the discriminator is built
#It has an input layer fit to the shape of the variables
#A loop creates the desired amount of deep layers
#It ends in a single sigmoid layer
#Additionally the last layer is saved to be an optional input to the adversary
    def buildDiscriminator(self):
        #The discriminator aims to separate signal and background
        #There is an input layer after which the desired amount of hidden layers is
added in a loop
        #In the loop normalisation and dropout are added too

        self.network_input = Input( shape = (self.input_dimension) )
        self.layer_discriminator = Dense( self.discriminator_nodes, activation =
"elu")(self.network_input)
        self.layer_discriminator = BatchNormalization()(self.layer_discriminator)
        self.layer_discriminator = Dropout(self.discriminator_dropout)
(self.layer_discriminator)
        for layercount in range(self.discriminator_layers -1):
            self.layer_discriminator = Dense(self.discriminator_nodes, activation =
"elu")(self.layer_discriminator)
            self.layer_discriminator = BatchNormalization()
(self.layer_discriminator)
            self.layer_discriminator = Dropout(self.discriminator_dropout)
(self.layer_discriminator)
        self.layer_discriminator = Dense( 1, activation = "sigmoid")
(self.layer_discriminator)

        self.model_discriminator = Model(inputs = [self.network_input], outputs =
[self.layer_discriminator])
        self.model_discriminator.compile(loss = "binary_crossentropy",
weighted_metrics = [metrics.binary_accuracy], optimizer =
self.discriminator_optimizer)
        self.model_discriminator.summary()

    def trainDiscriminator(self):

        #print(self.target_training[12:500])
        #print(self.target_training[-1:-100])

        self.model_discriminator.summary()

        self.discriminator_history =
self.model_discriminator.fit(self.sample_training, self.target_training.ravel(),
epochs=self.discriminator_epochs, batch_size = self.batchSize, sample_weight =
self.weight_training.ravel(), validation_data = (self.sample_validation,
self.target_validation, self.weight_validation.ravel()))
        self.discriminator_history_array.append(self.discriminator_history)
        print(self.discriminator_history.history.keys())


    def predictModel(self):

        self.model_prediction =
self.model_discriminator.predict(self.sample_validation).ravel()
        self.fpr, self.tpr, self.threshold = roc_curve(self.target_validation,
self.model_prediction)
```

```
        self.auc = auc(self.fpr, self.tpr)

        print('Discriminator AUC:', self.auc)

    def plotLosses(self):
        ax = plt.subplot(111)
        plt.plot(self.discriminator_history.history['loss'])
        plt.plot(self.discriminator_history.history['val_loss'])
        plt.title('Discriminator Losses')
        plt.ylabel('Loss')
        plt.xlabel('Epoch')
        plt.legend(['train', 'test'], loc='upper left')
#        plt.legend(loc="upper right", prop={'size' : 7})
        ax.ticklabel_format(style='sci', axis ='both', scilimits=(0,0))
        plt.gcf().savefig(output_path + 'losses.png')




    def plotRoc(self):
        plt.title('Receiver Operating Characteristic')
        plt.plot(self.fpr, self.tpr, 'g--', label='$AUC_{train}$ = %0.2f'%
self.auc)
        plt.legend(loc='lower right')
        plt.plot([0,1],[0,1],'r--')
        plt.xlim([-0.,1.])
        plt.ylim([-0.,1.])
        plt.ylabel('True Positive Rate', fontsize='large')
        plt.xlabel('False Positive Rate', fontsize='large')
        plt.legend(frameon=False)
        #plt.show()
        plt.gcf().savefig(output_path + 'roc.png')
        #plt.gcf().savefig(output_path + 'simple_ROC_' + file_extension + '.eps')
        plt.gcf().clear()

    def plotSeparation(self):
        self.signal_histo = []
        self.background_histo = []
        for i in range(len(self.sample_validation)):
            if self.target_validation[i] == 1:
                self.signal_histo.append(self.model_prediction[i])
            if self.target_validation[i] == 0:
                self.background_histo.append(self.model_prediction[i])

        plt.hist(self.signal_histo, range=[0., 1.], linewidth = 2, bins=30,
histtype="step", density = True, color=color_tW, label = "Signal")
        plt.hist(self.background_histo, range=[0., 1.], linewidth = 2, bins=30,
histtype="step", density = True, color=color_tt, label = "Background")
        plt.legend()
        plt.xlabel('Network response', horizontalalignment='left',
fontsize='large')
        plt.ylabel('Event fraction', fontsize='large')
        plt.legend(frameon=False)
        plt.gcf().savefig(output_path + 'separation_discriminator.png')
        plt.gcf().clear()
    def plotAccuracy(self):
        plt.plot(self.discriminator_history.history['weighted_binary_accuracy'])

plt.plot(self.discriminator_history.history['val_weighted_binary_accuracy'])
        plt.title('model accuracy')
        plt.ylabel('accuracy')
        plt.xlabel('epoch')
        plt.legend(['train', 'test'], loc='upper left')
        plt.gcf().savefig(output_path + 'acc.png')
        plt.gcf().clear()

#In the following options and variables are read in
#This is done to keep the most important features clearly represented

#You need to find a nice set of variables to use and add them as a text file
#with open('/cephfs/user/s6chkirf/whk_ANN_variables.txt','r') as varfile:
  #  variableList = varfile.read().splitlines()

#print(variableList)
#def ReadOptions(region):
#    with variables =
variableListopen('/cephfs/user/s6chkirf/config_whk_ANN.txt','r') as infile:
#        optionsList = infile.read().splitlines()
#    OptionDict = dict()
#    for options in optionsList:
#           # definition of a comment
#        if options.startswith('#'): continue
#        templist = options.split(' ')
#        if len(templist) == 2:
#            OptionDict[templist[0]] = templist[1]
#        else:
#            OptionDict[templist[0]] = templist[1:]
#    return OptionDict
#    # a dictionary of options is returned


first_training = neuralNetworkEnvironment()
first_training.initialize_sample()
first_training.buildDiscriminator()
first_training.trainDiscriminator()
first_training.predictModel()
first_training.plotRoc()
first_training.plotSeparation()
first_training.plotAccuracy()
first_training.plotLosses()
```