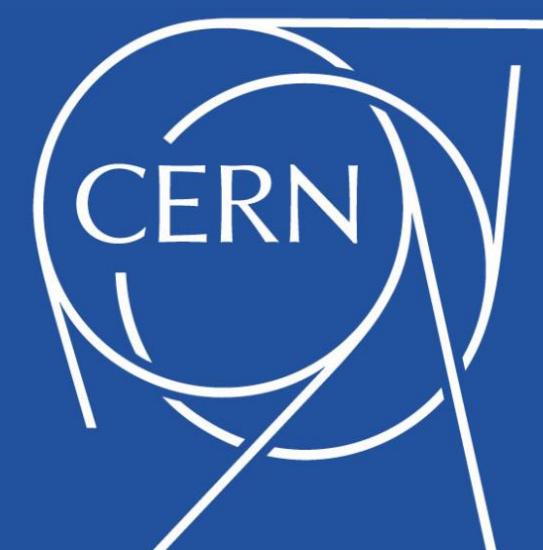


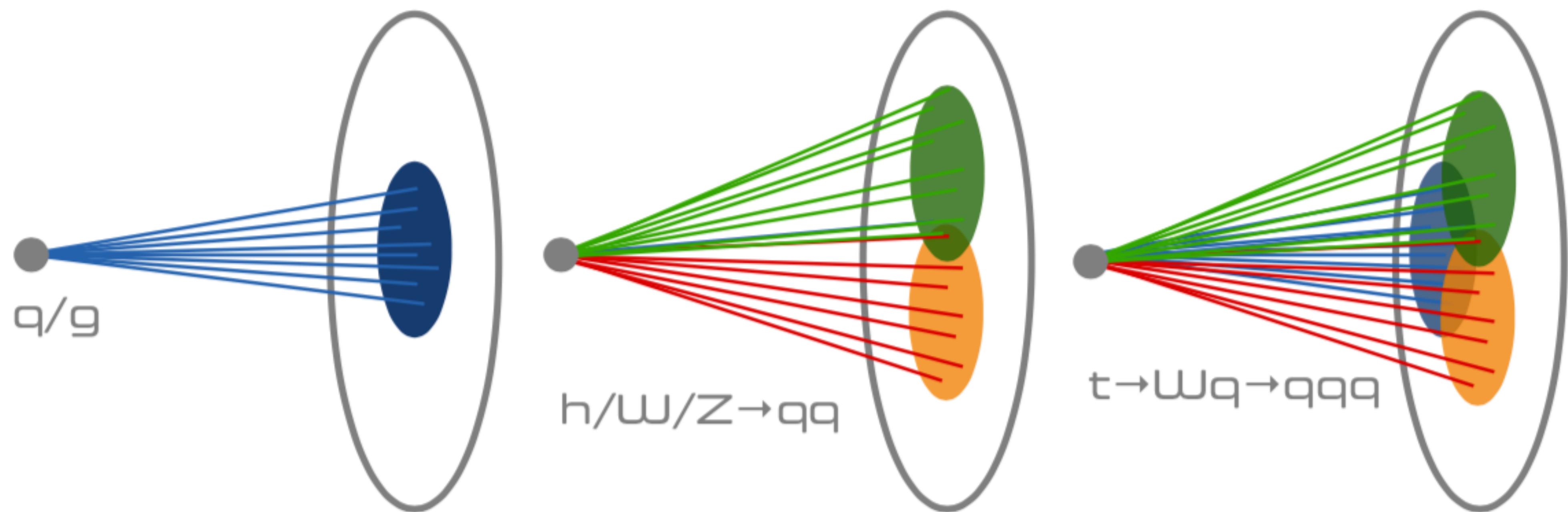
# Graph Neural Networks: hands-on session

Maurizio Pierini



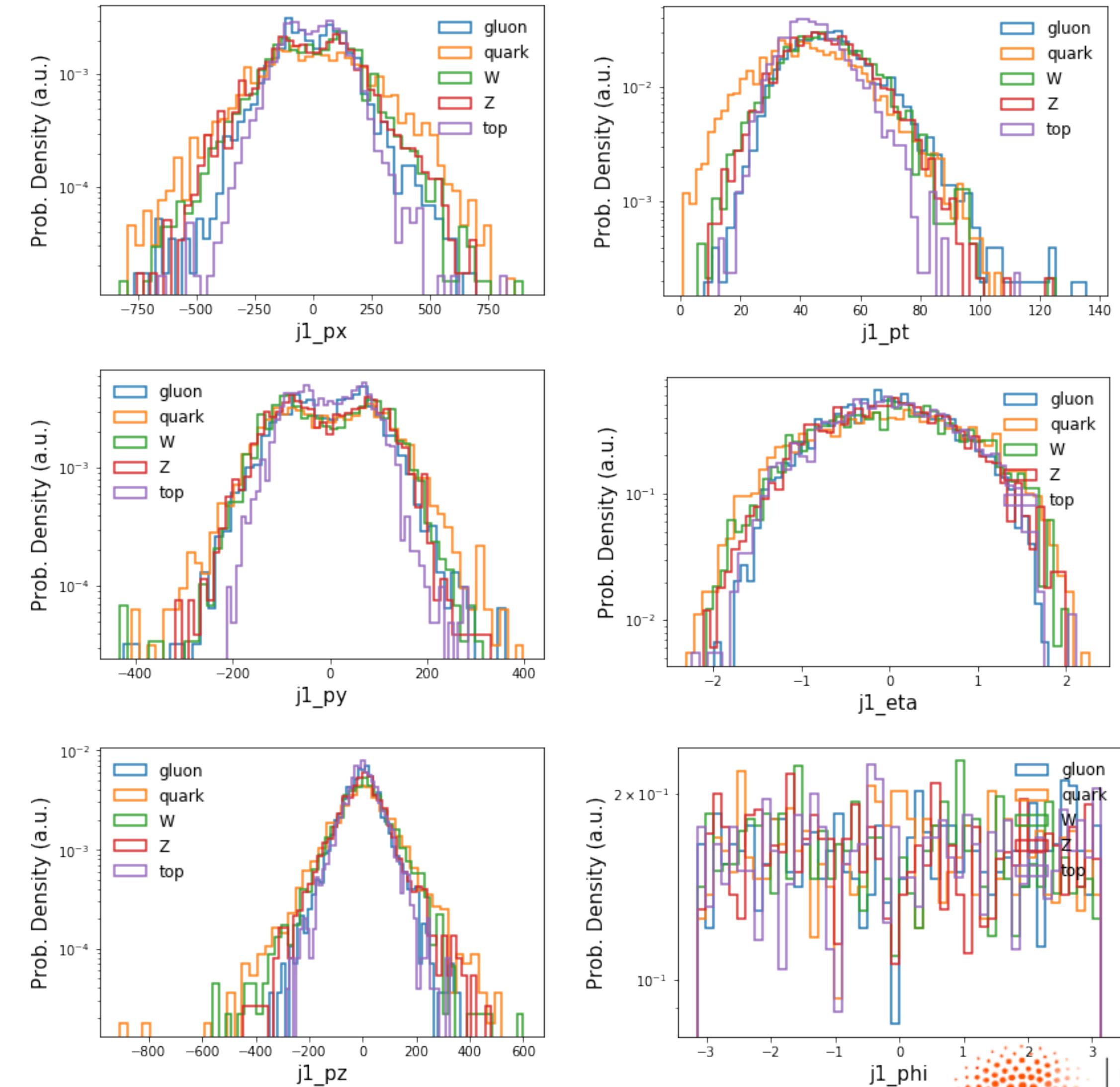
# The problem: jet tagging

- At the LHC, a jet is a spray of hadrons coming from a “shower” initiated by a quark or a gluon
- When heavy particles ( $W$ ,  $Z$ , top quark) decaying to quarks are produced at large momentum, multiple jets overlap in a single jet



# The data representation

- Each jet is represented as a list of up to 100 particles, ordered by decreasing transverse momentum
- We will cut the list at 30, to make the training faster
- Each particle is represented as a list of 16 features
- We will use only 3 coordinates:  $(px, py, pz)$  as a default, but you can switch to  $(p_T, \eta, \varphi)$  if you want to
- See [this notebook](#) for more info



# Step1: we read the data

- *Nothing particularly enlightening. Just push Shift+Enter*

## Import needed libraries

```
import os
import h5py
import glob
import numpy as np
import matplotlib.pyplot as plt
```

```
import torch
import torch.nn as nn
from torch.autograd.variable import *
import torch.optim as optim
```

```
%matplotlib inline
```

## Download dataset

```
! git clone https://github.com/pierinim/tutorials.git
! ls tutorials/Data/JetDataset/
```

# Step1: we read the data

- Nothing particularly enlightening. Just push Shift+Enter

## Append ground-truth and jet data in 2 numpy arrays

```
target = np.array([])
jetList = np.array([])
# we cannot load all data on Colab. So we just take a few files
datafiles = ['tutorials/Data/JetDataset/jetImage_7_100p_30000_40000.h5',
             'tutorials/Data/JetDataset/jetImage_7_100p_60000_70000.h5',
             'tutorials/Data/JetDataset/jetImage_7_100p_50000_60000.h5',
             'tutorials/Data/JetDataset/jetImage_7_100p_10000_20000.h5',
             'tutorials/Data/JetDataset/jetImage_7_100p_0_10000.h5']
# if you are running locally, you can use the full dataset doing
# for fileIN in glob.glob("tutorials/HiggsSchool/data/*h5"):
for fileIN in datafiles:
    print("Appending %s" %fileIN)
    f = h5py.File(fileIN)
    # for pt, eta, phi
    myJetList = np.array(f.get("jetConstituentList")[:, :, [5, 8, 11]])
    # for px, py, pz
    #myJetList = np.array(f.get("jetConstituentList")[:, :, [0, 1, 2]])
    mytarget = np.array(f.get('jets')[0:-6:-1])
    jetList = np.concatenate([jetList, myJetList], axis=0) if jetList.size else myJetList
    target = np.concatenate([target, mytarget], axis=0) if target.size else mytarget
    del myJetList, mytarget
    f.close()
print(target.shape, jetList.shape)
```

# Step1: we read the data

- Nothing particularly enlightening. Just push Shift+Enter

## Data preparation: transpose, encoding, train/val/test split

```
# pytorch Cross Entropy doesn't support one-hot encoding
target = np.argmax(target, axis=1)
# the dataset is N_jets x N_particles x N_features
# the IN wants N_jets x N_features x N_particles
jetList = np.swapaxes(jetList, 1, 2)
```

```
nParticle = 30
jetList = jetList[:, :, :nParticle]
print(jetList.shape)
```

```
from sklearn.model_selection import train_test_split
X_train, X_val, y_train, y_val = train_test_split(jetList, target, test_size=0.33)
print(X_train.shape, X_val.shape, y_train.shape, y_val.shape)
del jetList, target
```

```
# check if a GPU is available. Otherwise run on CPU
device = 'cpu'
args_cuda = torch.cuda.is_available()
if args_cuda: device = "cuda"
```

```
# Convert dataset to pytorch
X_train = Variable(torch.FloatTensor(X_train)).to(device)
X_val = Variable(torch.FloatTensor(X_val)).to(device)
y_train = Variable(torch.LongTensor(y_train).long()).to(device)
y_val = Variable(torch.LongTensor(y_val).long()).to(device)
```

# Step2: setup the device

- *Nothing particularly enlightening. Just push Shift+Enter*

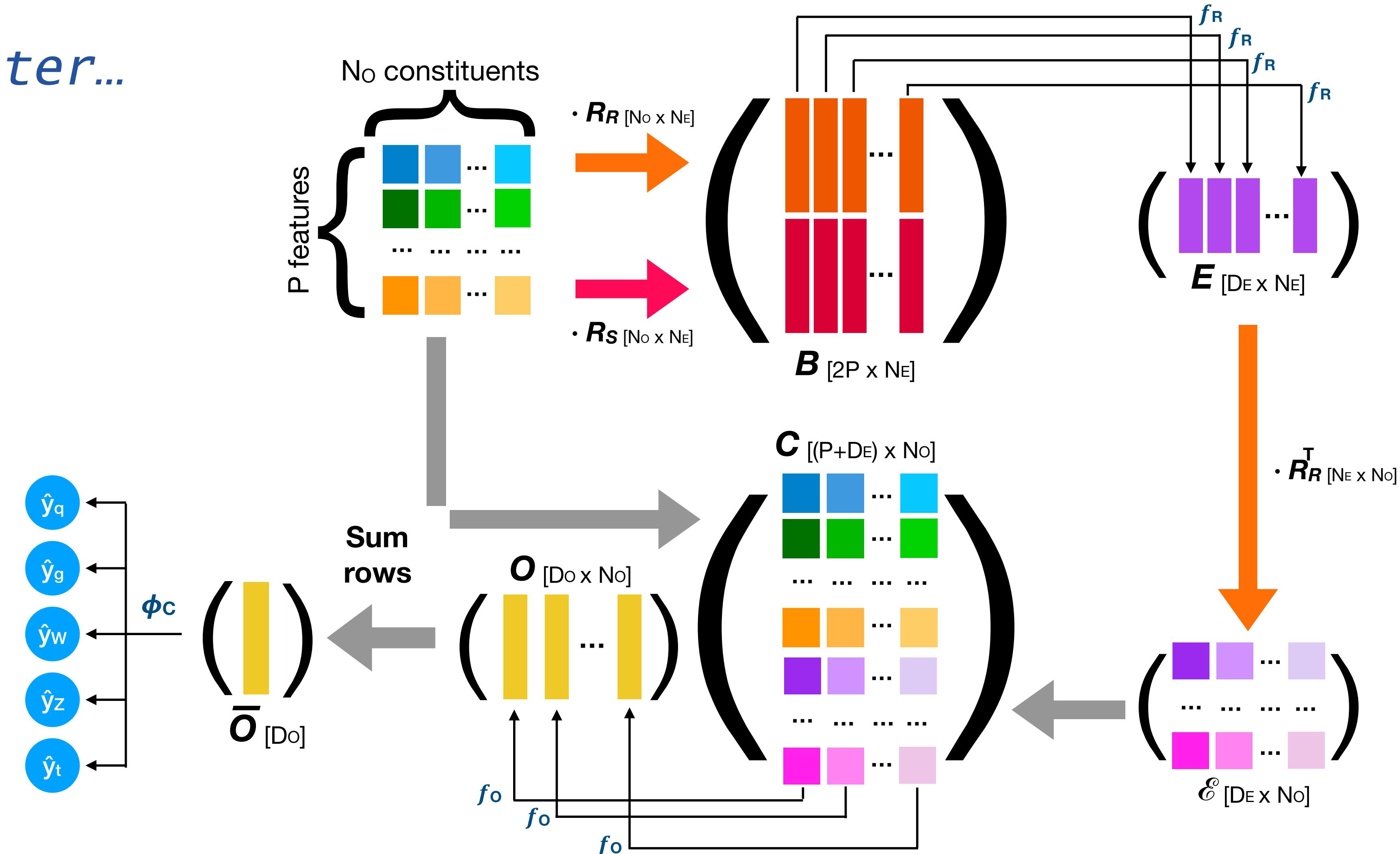
**If GPU available (it should) we move data there**

```
# check if a GPU is available. Otherwise run on CPU
device = 'cpu'
args_cuda = torch.cuda.is_available()
if args_cuda: device = "cuda"
```

```
# Convert dataset to pytorch
X_train = Variable(torch.FloatTensor(X_train)).to(device)
X_val = Variable(torch.FloatTensor(X_val)).to(device)
y_train = Variable(torch.LongTensor(y_train).long()).to(device)
y_val = Variable(torch.LongTensor(y_val).long()).to(device)
```

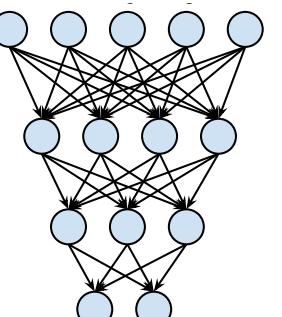
# Step3: we build the model

More on this later...



No: # of constituents  
 P: # of features  
 $N_E = \text{No}(\text{No}-1)$ : # of edges  
 $D_E$ : size of internal representations  
 $D_O$ : size of post-interaction internal representation

$\phi_C$ ,  $f_O$ ,  $f_R$   
parameterized as  
neural networks



# Step4: we train

- Nothing particularly enlightening. Just push Shift+Enter

```
n_epochs = 300
batch_size = 100
patience = 10

import util
# instantiate the model & setup training
# relu
gnn = GraphNet()
gnn.to(device)
loss = nn.CrossEntropyLoss()
optimizer = optim.Adam(gnn.parameters(), lr = 0.0001)

loss_train = np.zeros(n_epochs)
acc_train = np.zeros(n_epochs)
loss_val = np.zeros(n_epochs)
acc_val = np.zeros(n_epochs)
for i in range(n_epochs):
    print("Epoch %s" % i)
    for j in range(0, X_train.size()[0], batch_size):
        optimizer.zero_grad()
        out = gnn(X_train[j:j + batch_size,:,:])
        target = y_train[j:j + batch_size]
        l = loss(out, target)
        l.backward()
        optimizer.step()
        loss_train[i] += l.cpu().data.numpy()*batch_size
    loss_train[i] = loss_train[i]/X_train.shape[0]
```

# Step4: we train

- Nothing particularly enlightening. Just push Shift+Enter

```
#### val loss & accuracy
for j in range(0, X_val.size()[0], batch_size):
    out_val = gnn(X_val[j:j + batch_size])
    target_val = y_val[j:j + batch_size]

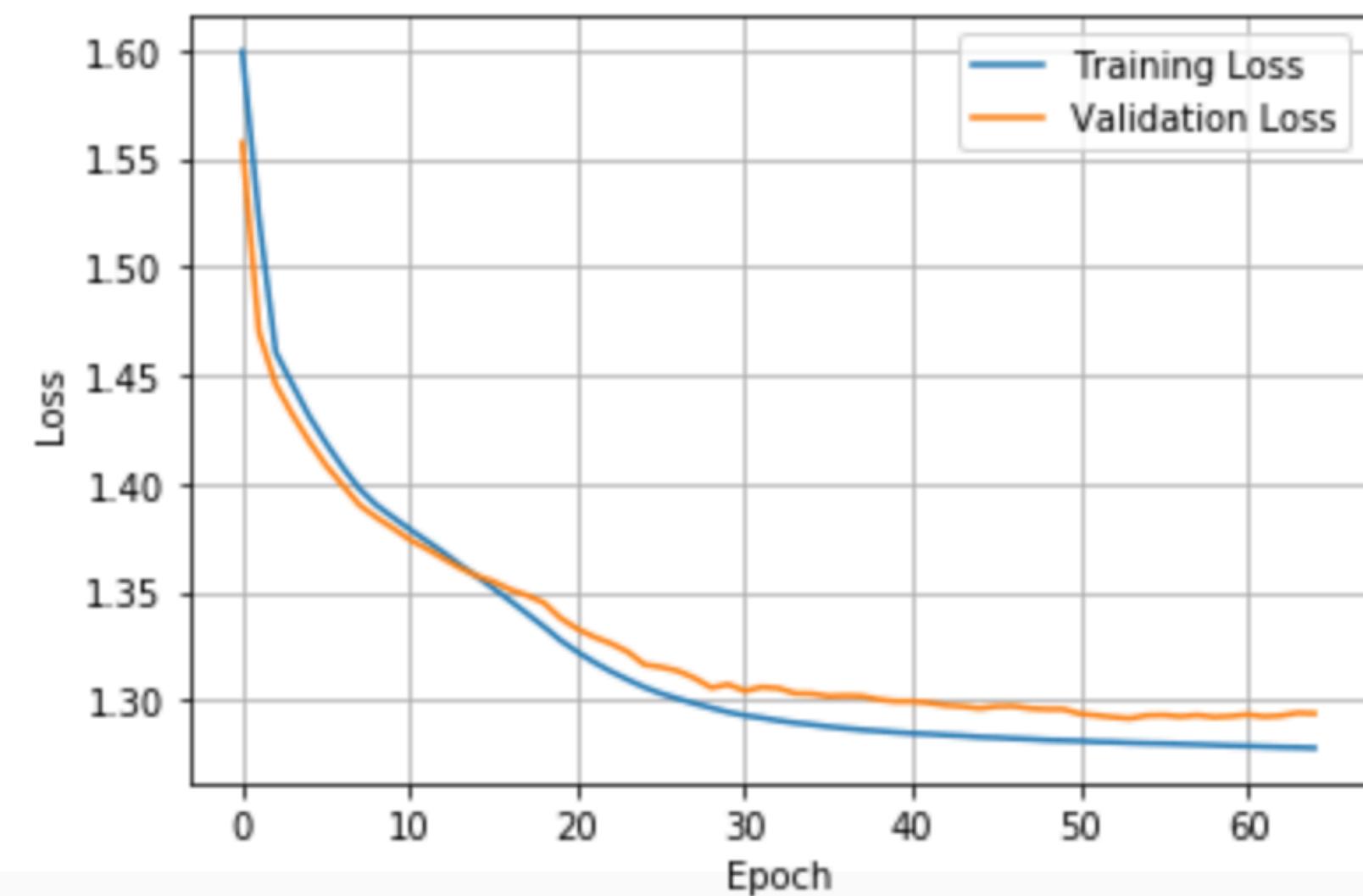
    l_val = loss(out_val,target_val)
    loss_val[i] += l_val.cpu().data.numpy()*batch_size
loss_val[i] = loss_val[i]/X_val.shape[0]
print("Training Loss: %f" %l.cpu().data.numpy())
print("Validation Loss: %f" %l_val.cpu().data.numpy())
if all(loss_val[max(0, i - patience):i] > min(np.append(loss_val[0:max(0, i - patience)], 200))) and i > patience:
    print("Early Stopping")
    break
print
```

# Step5: we validate

- Nothing particularly enlightening. Just push Shift+Enter

## Training history

```
: epoch_number = list(range((loss_train > 0.).sum()))
plt.figure()
plt.plot(epoch_number, loss_train[loss_train>0.],label='Training Loss')
plt.plot(epoch_number, loss_val[loss_train>0.],label='Validation Loss')
plt.ylabel("Loss")
plt.xlabel("Epoch")
plt.grid(True)
plt.legend(loc='upper right')
# plt.savefig('%s/ROC.pdf'%(options.outputDir))
plt.show()
```



# Step5: we validate

- Nothing particularly enlightening. Just push Shift+Enter

## Compute TPR and FPR for each class

```
labels = ['gluon', 'quark', 'W', 'Z', 'top']

lst = []
n_batches_val = int(X_val.size()[0]/batch_size)
if args_cuda:
    for j in torch.split(X_val, n_batches_val).cuda():
        a = gnn(j).cpu().data.numpy()
        lst.append(a)
else:
    for j in torch.split(X_val, n_batches_val):
        a = gnn(j).cpu().data.numpy()
        lst.append(a)
predicted = Variable(torch.FloatTensor(np.concatenate(lst)))

# there is no softmax in the output layer. We have to put it by
predicted = torch.nn.functional.softmax(predicted, dim=1)

predict_val = predicted.data.numpy()
true_val = y_val.data.numpy()
```

Pytorch puts the softmax in activation function

- Not at the output layer
- - We need to include it when we run the model

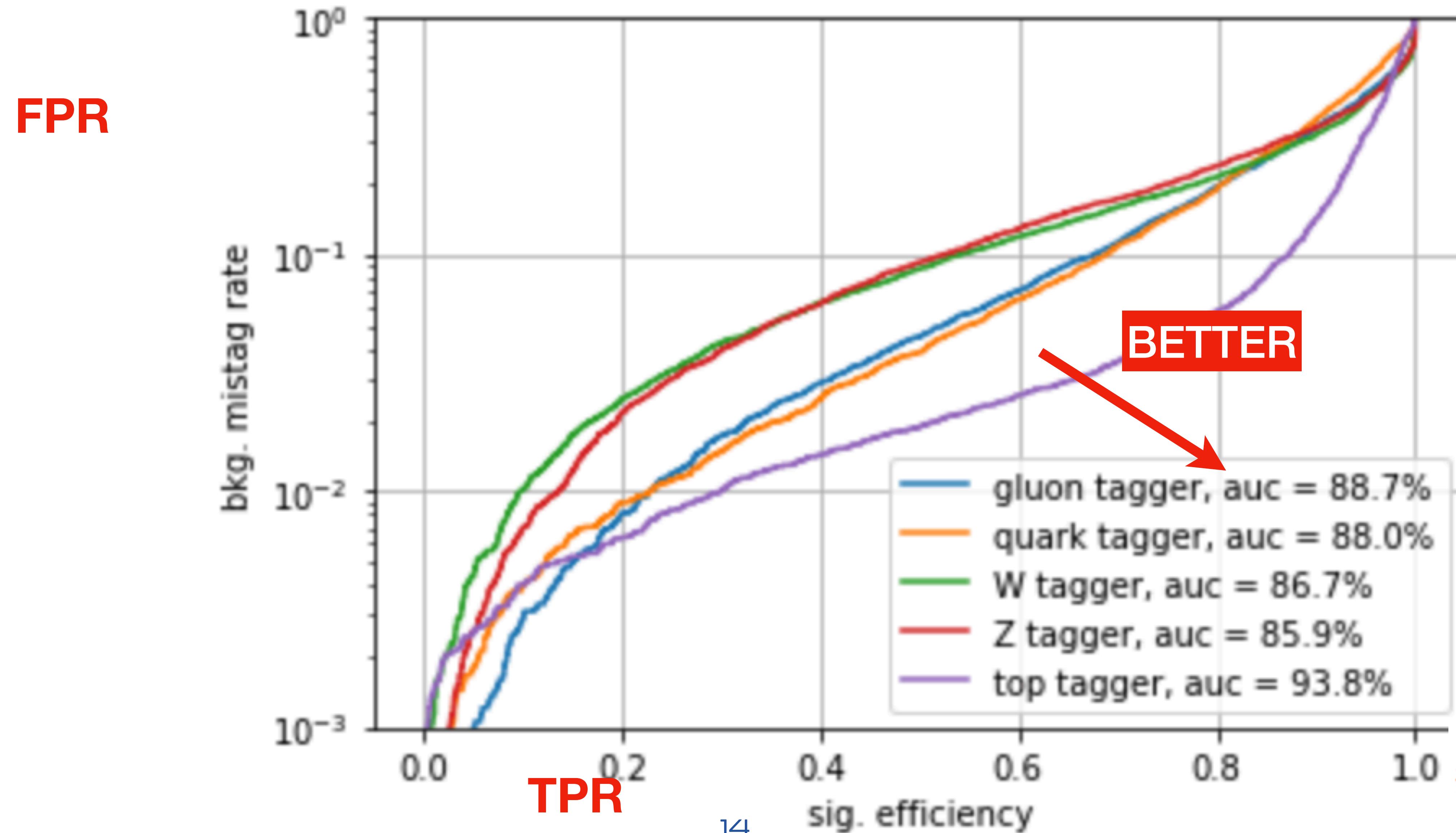
# Step5: we validate

- Nothing particularly enlightening. Just push Shift+Enter

## Build the ROC curve

```
: from sklearn.metrics import roc_curve, auc
#### get the ROC curves
fpr = {}
tpr = {}
auc1 = {}
plt.figure()
for i, label in enumerate(labels):
    fpr[label], tpr[label], threshold = roc_curve((true_val== i), predict_val[:,i])
    auc1[label] = auc(fpr[label], tpr[label])
    plt.plot(tpr[label],fpr[label],label='%s tagger, auc = %.1f%%'%(label, auc1[label]*100.))
plt.semilogy()
plt.xlabel("sig. efficiency")
plt.ylabel("bkg. mistag rate")
plt.ylim(0.001,1)
plt.grid(True)
plt.legend(loc='lower right')
# plt.savefig('%s/ROC.pdf'%(options.outputDir))
plt.show()
```

# Step5: we validate



# Building the model

- A *model* is allocated as a *class*
- All *networks* are allocated in `__init`
- The *forward* class defines the *network flow*
  - This is what you need to write
- A few auxiliary functions are provided (see next slide)

# Building the model

## ○ Auxiliary functions

```
def assign_matrices(self):
    self.Rr = torch.zeros(self.N, self.Nr)
    self.Rs = torch.zeros(self.N, self.Nr)
    receiver_sender_list = [i for i in itertools.product(range(self.N), range(self.N)) if i[0] != i[1]]
    for i, (r, s) in enumerate(receiver_sender_list):
        self.Rr[r, i] = 1
        self.Rs[s, i] = 1
    self.Rr = Variable(self.Rr)
    self.Rs = Variable(self.Rs)
```

## ○ Matrix multiplication

```
def tmul(self, x, y): #Takes (I * J * K)(K * L) -> I * J * L
    x_shape = x.size()
    y_shape = y.size()
    return torch.mm(x.view(-1, x_shape[2]), y).view(-1, x_shape[1], y_shape[1])
```

## ○ Load samples

```
def get_sample(training, target, choice):
    target_vals = np.argmax(target, axis = 1)
    ind, = np.where(target_vals == choice)
    chosen_ind = np.random.choice(ind, 50000)
    return training[chosen_ind], target[chosen_ind]
```

# Building the model

## You need to allocate the dense networks

### Building the IN model

```
In [ ]: import itertools

class GraphNet(nn.Module):
    def __init__(self):
        super(GraphNet, self).__init__()

        self.P = 3 # number of features
        self.N = nParticle # number of particles
        self.Nr = self.N * (self.N - 1)
        self.De = 8 # dimensionality of De learned representation
        self.Do = 8 # number of engineered features
        self.n_targets = 5 # number of target classes
        self.assign_matrices() # build Rr and Rs

        # a dense layer is allocated doing
        # layer = nn.Linear(nodes_in, nodes_out)
        # This computes y = w*x + b
        # activation functions come later
        self.fr1 = # FILL THIS LINE making sure that the input dimension is correct
        self.fr2 = # FILL THIS LINE
        self.fr3 = # FILL THIS LINE making sure that the output dimension is correct

        self.fo1 = # FILL THIS LINE making sure that the input dimension is correct
        self.fo2 = # FILL THIS LINE
        self.fo3 = # FILL THIS LINE making sure that the output dimension is correct

        self.fc1 = # FILL THIS LINE making sure that the input dimension is correct
        self.fc2 = # FILL THIS LINE
        self.fc3 = # FILL THIS LINE making sure that the output dimension is correct

    def assign_matrices(self):
        self.Rr = torch.zeros(self.N, self.Nr)
```

# Building the model

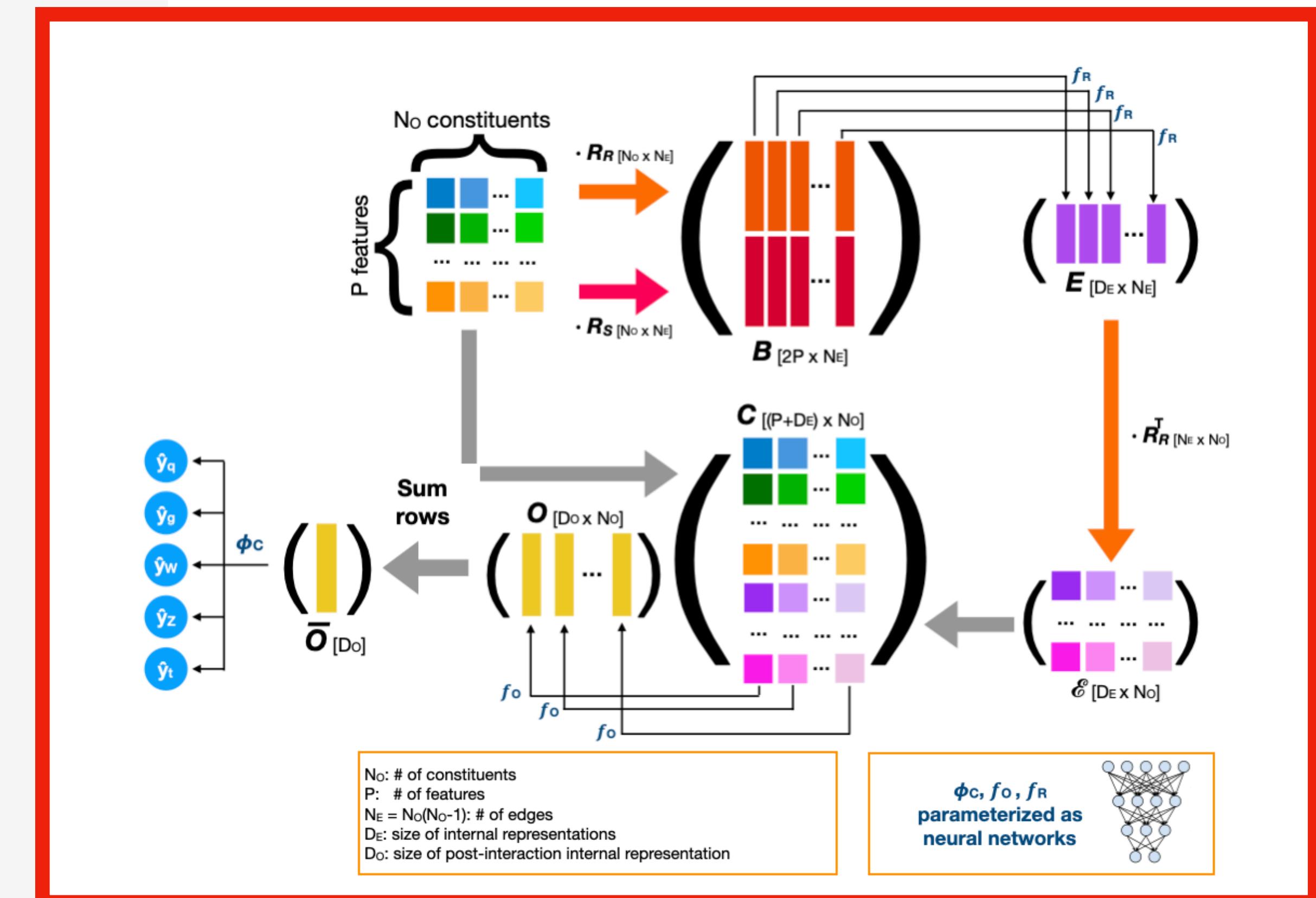
## And set the tensor computation to reflect the IN structure

```

def forward(self, x):
    # use Rr and Rs to create the B matrix
    Orr = self.tmul(x, self.Rr)
    Ors = self.tmul(x, self.Rs)
    # This is how you append two tensors
    # the 1 means that Rs is appended to Rr along the the second - vertical - axis
    B = torch.cat([Orr, Ors], 1)

    ### First MLP ####
    # this is how you transpose
    # we transpose a lot. This is because we represent the input such
    # that each column is an interation (or a message) between a sender and a receiver
    # We want to pass *columns* to the dense NNs
    B = torch.transpose(B, 1, 2).contiguous()
    # the view command is a reshape function
    B = nn.functional.relu(self.fr1(B.view(-1, 2 * self.P)))
    B = # FILL THIS LINE
    E = nn.functional.relu(self.fr3(B).view(-1, self.Nr, self.De))
    del B
    # now we transpose E back, so that interation are columns
    E = torch.transpose(E, 1, 2).contiguous()
    Ebar = self.tmul(E, torch.transpose(self.Rr, 0, 1).contiguous())
    del E
    # we build C appending Ebar to x
    C = # FILL THIS LINE
    del Ebar
    # again, we want to pass columns to the next networks.
    # so we transpose
    C = # FILL THIS LINE
    ### Second MLP ####
    C = nn.functional.relu(self.fo1(C.view(-1, self.P + self.De)))
    C = # FILL THIS LINE
    O = nn.functional.relu(self.fo3(C).view(-1, self.N, self.Do))
    del C
    # to build features, we apply an aggregation function
    # you can use .sum, .max, etc (see https://pytorch.org/docs/stable/torch.html)
    # the operation goes on the second axis, so you want to do torch.SOMETHING(0, 1)
    O = torch.sum(O, 1)
    ### Classification MLP ####
    N = # FILL THIS LINE
    N = # FILL THIS LINE
    del O
    N = # FILL THIS LINE: remember that the loss function in pytorch applies the softmax function
    return N

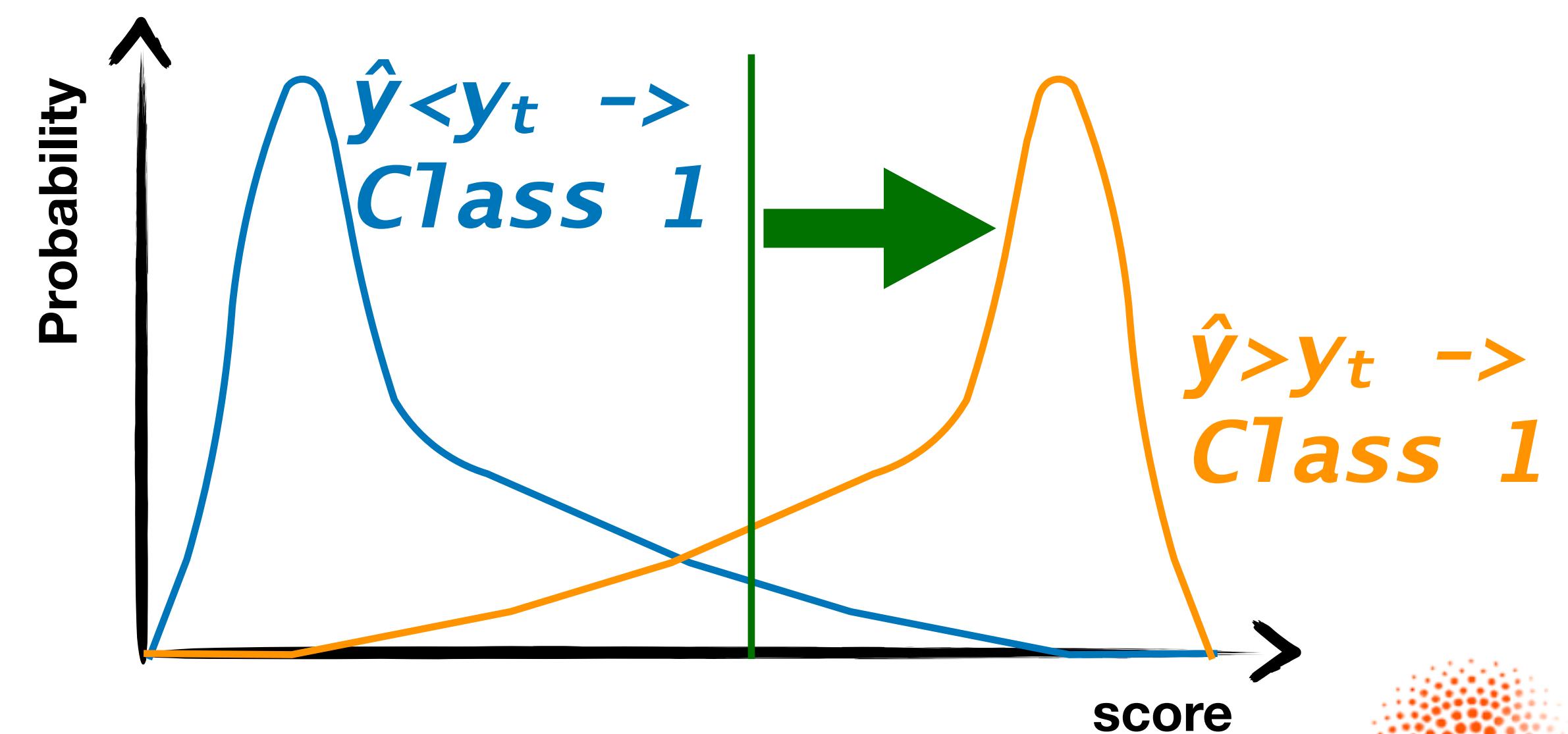
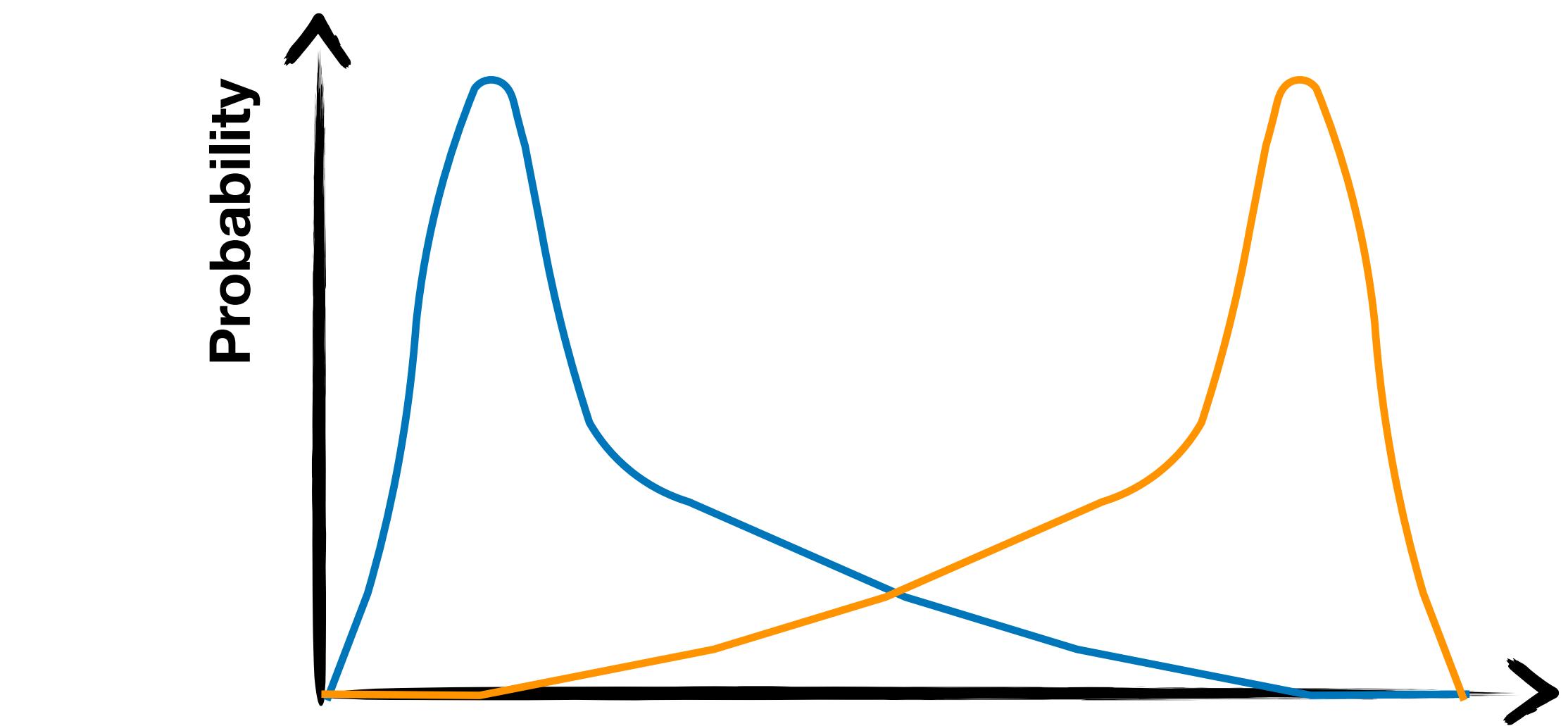
```



# Backup

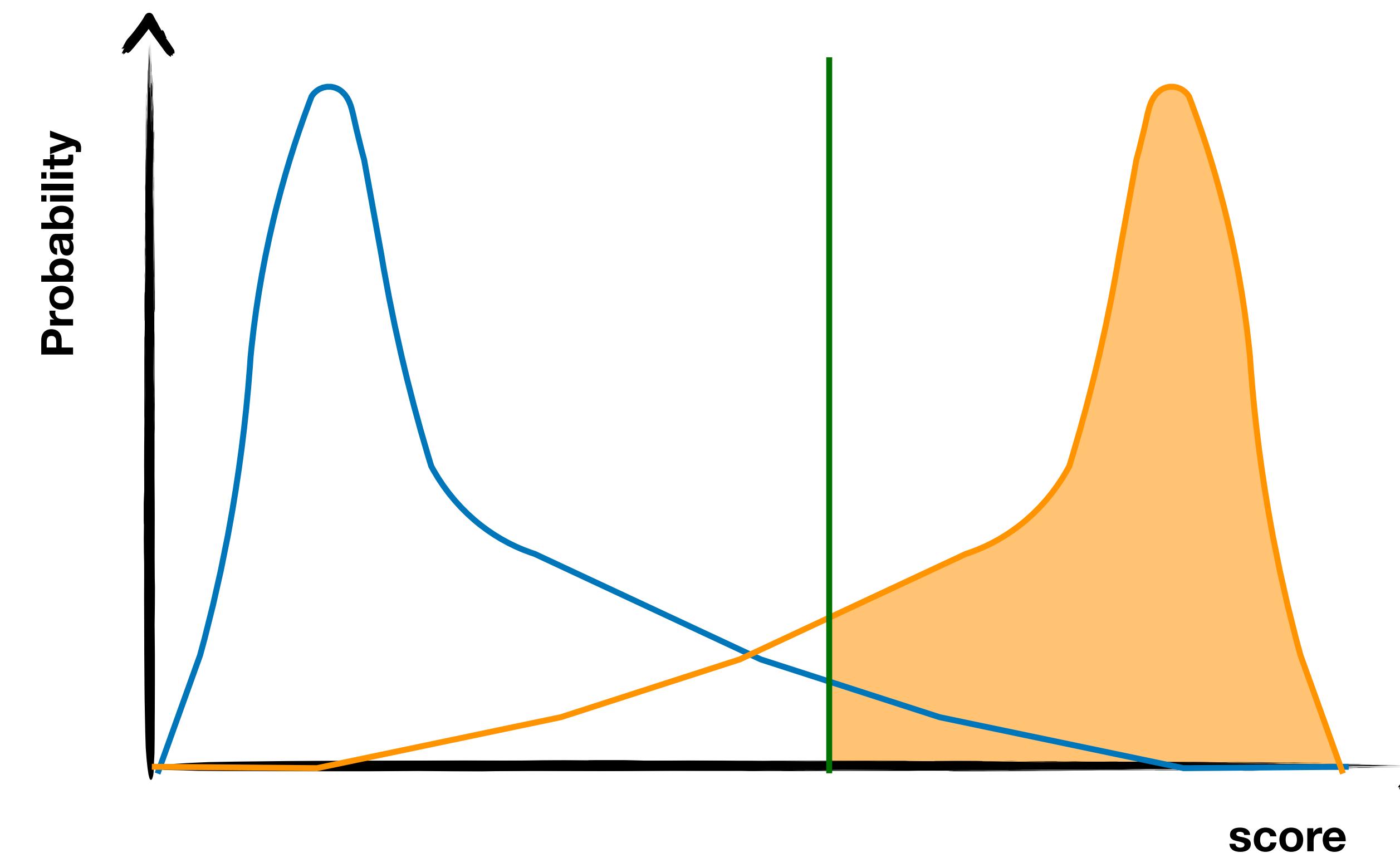
# What is a ROC curve?

- Consider a binary classifier
- Its output  $\hat{y}$  is a number in  $[0, 1]$
- If well trained, value should be close to 0 (1) for class-0 (class-1) examples
- One usually defines a threshold  $y_t$  such that:
  - $\hat{y} > y_t \rightarrow \text{Class 1}$
  - $\hat{y} < y_t \rightarrow \text{Class 0}$



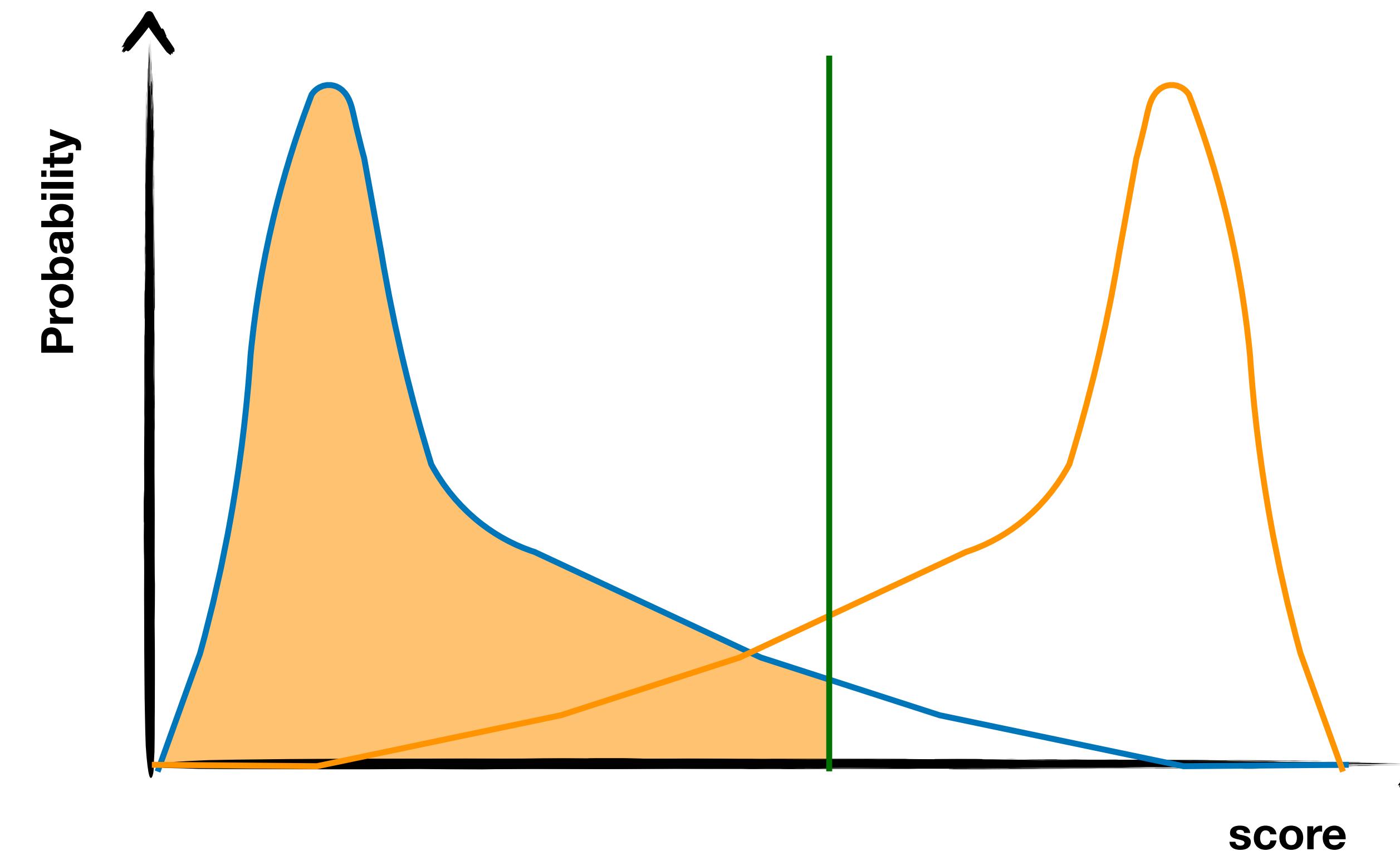
# What is a ROC curve?

- A given threshold defines the following qualities
  - True-positives: Class-1 events above the threshold
  - True-negatives: Class-0 events below the threshold
  - False-positives: Class-0 events above the threshold
  - False-negatives: Class-1 events below the threshold



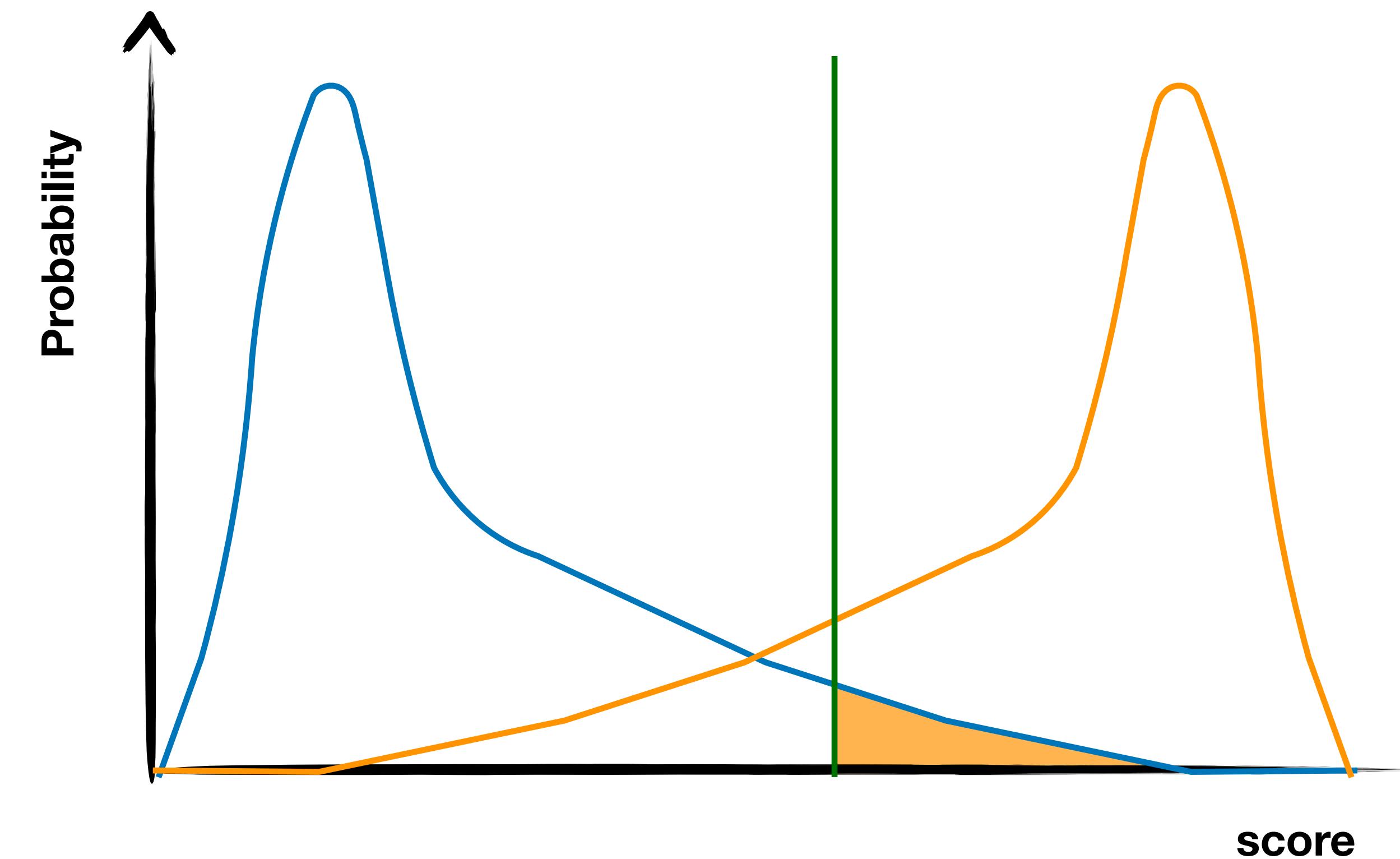
# What is a ROC curve?

- A given threshold defines the following qualities
  - True-positives: Class-1 events above the threshold
  - True-negatives: Class-0 events below the threshold
  - False-positives: Class-0 events above the threshold
  - False-negatives: Class-1 events below the threshold



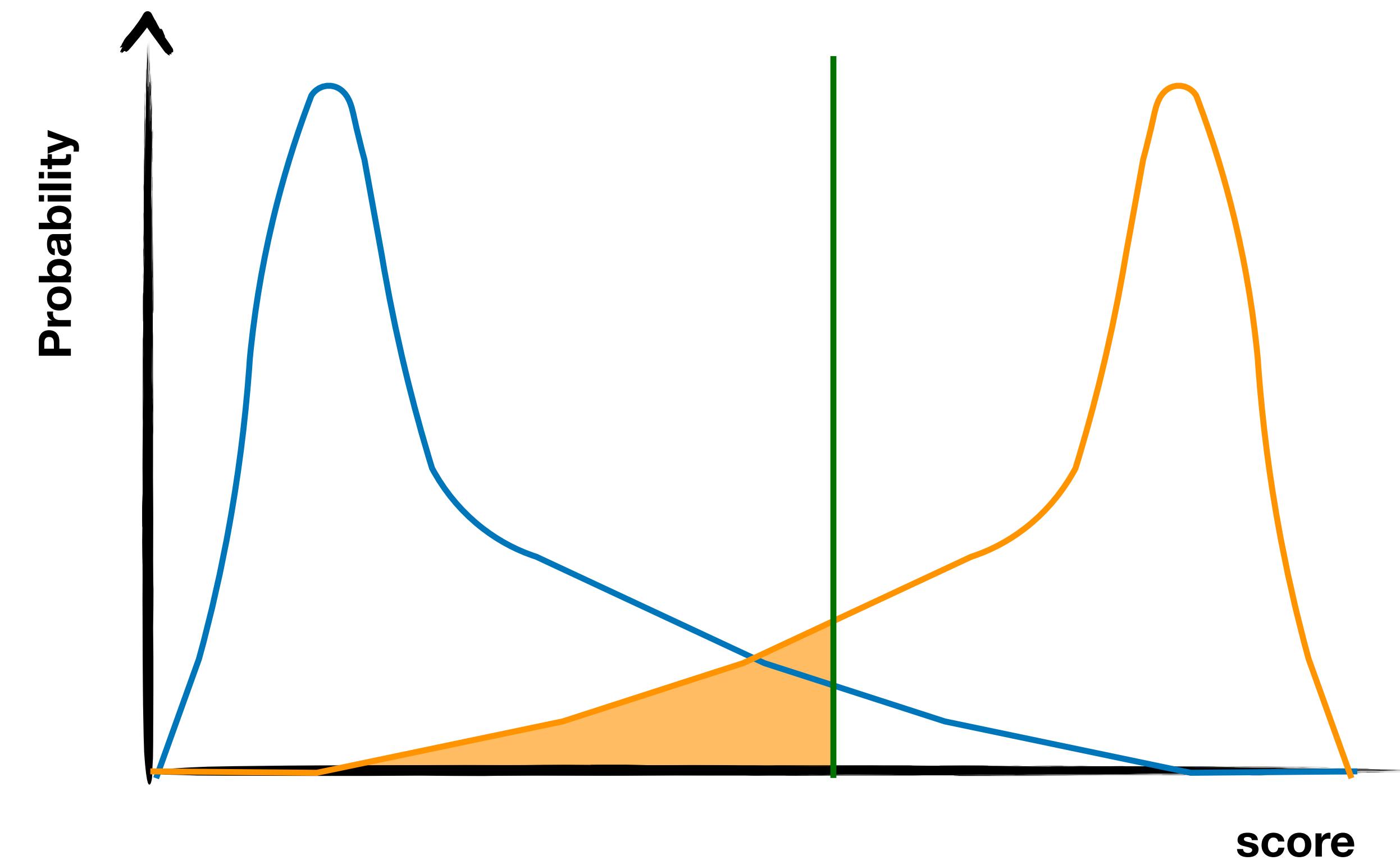
# What is a ROC curve?

- A given threshold defines the following qualities
  - True-positives: Class-1 events above the threshold
  - True-negatives: Class-0 events below the threshold
  - False-positives: Class-0 events above the threshold
  - False-negatives: Class-1 events below the threshold



# What is a ROC curve?

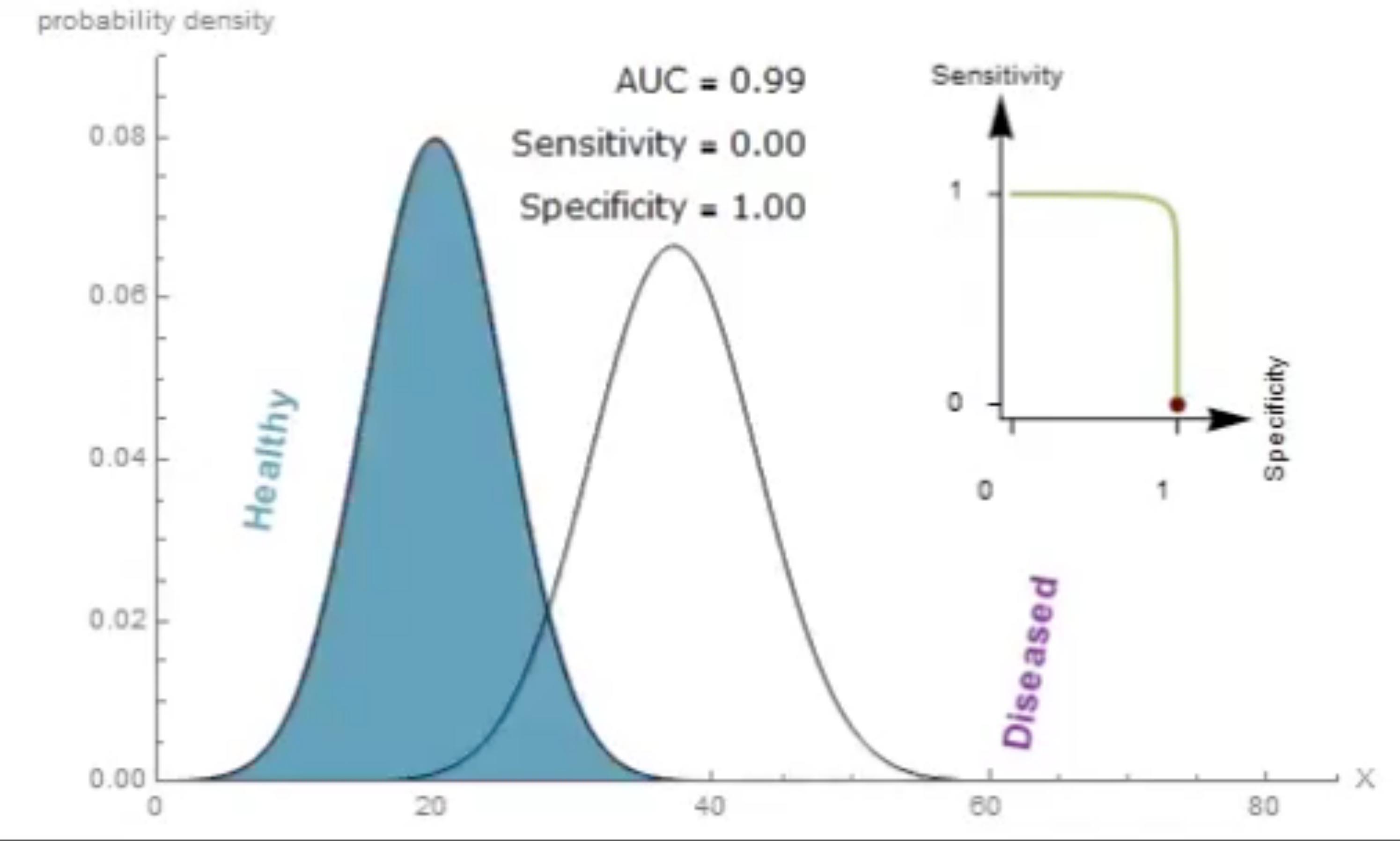
- A given threshold defines the following qualities
  - True-positives: Class-1 events above the threshold
  - True-negatives: Class-0 events below the threshold
  - False-positives: Class-0 events above the threshold
  - False-negatives: Class-1 events below the threshold



# Classifier metrics

- *Accuracy:  $(TP+TN)/Total$*
- *The fraction of events correctly classified*
- *Sensitivity:  $TP/(Total \ positive)$*
- *We call it signal efficiency*
- *Specificity:  $TN/(Total \ negative)$*
- *We call it 1-mistag*
- *These metrics depend on the chosen threshold*
- *To build a threshold-independent metric, we need a ROC curve*

# What is a ROC curve?



# A comparison

