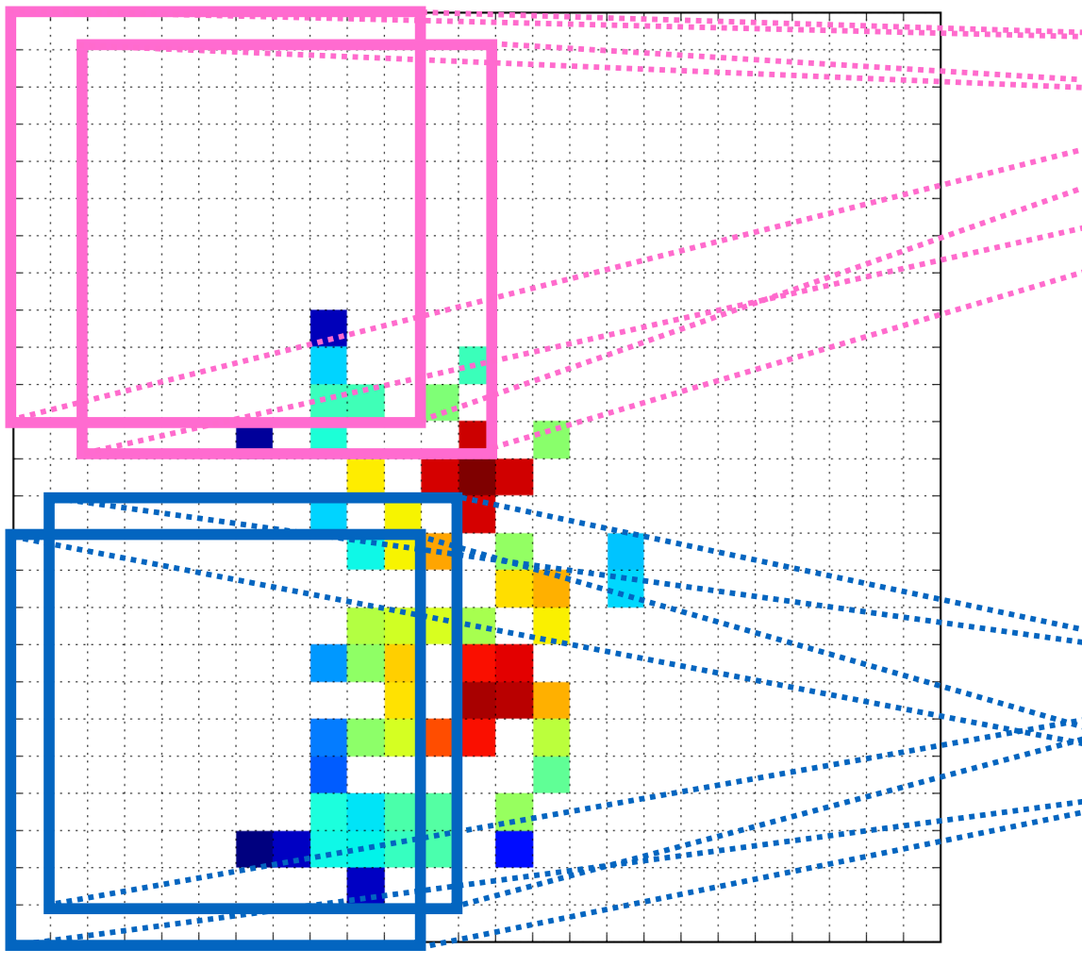
Introduction to Machine Learning for Physics

Benjamin Nachman

Lawrence Berkeley National Laboratory

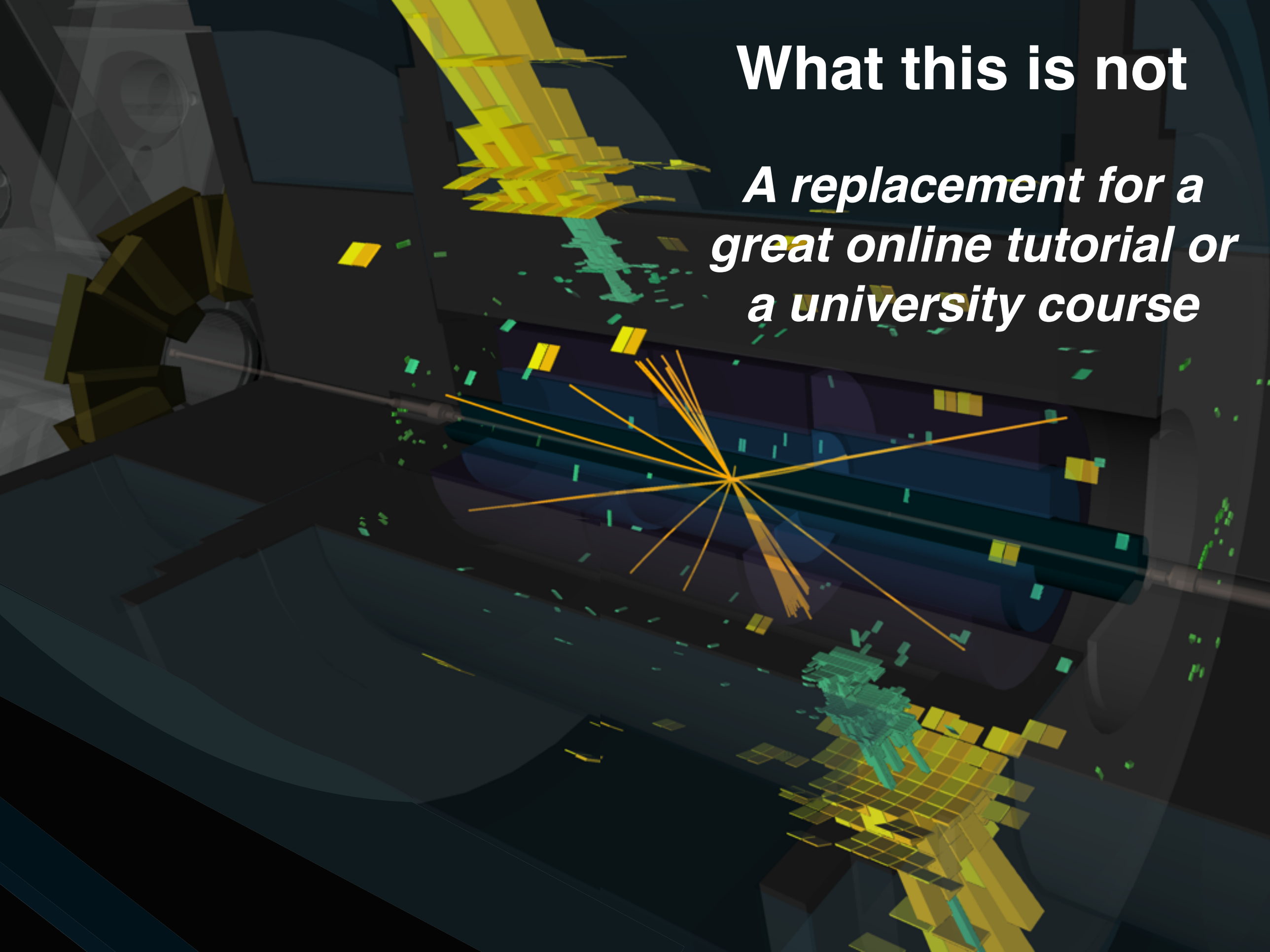
TRISEP School
August 1, 2019



What this is not

What this is not

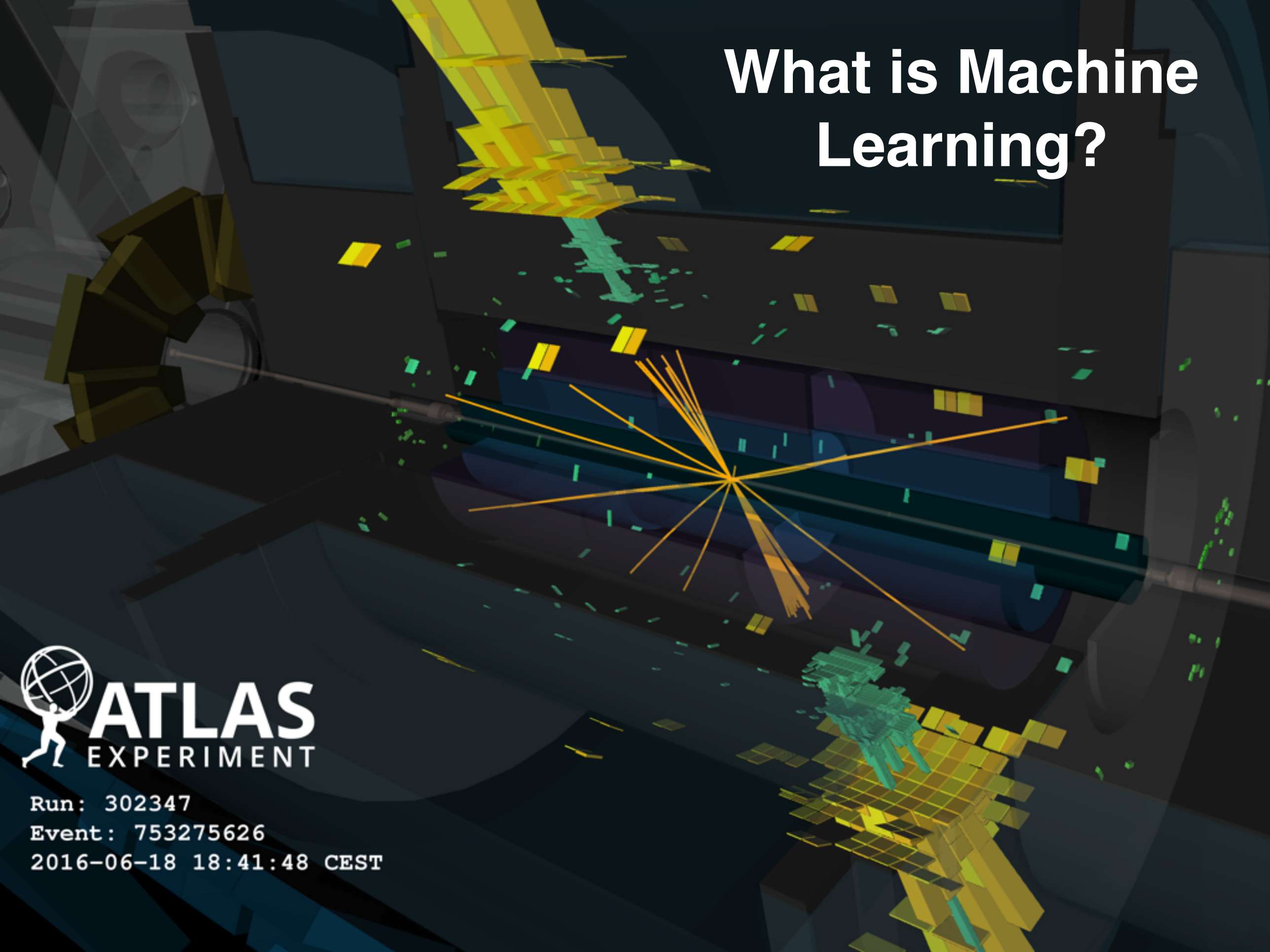
…the Higgs boson?



What this is not

A replacement for a great online tutorial or a university course

What is Machine Learning?



What is Machine Learning?

Answer: just about everything we do!

…algorithms for identifying and analyzing structure in data

What can we use machine learning for?

Supervised learning

Classification

Regression

Generation

the machine is presented examples of multiple classes and learns to differentiate

Unsupervised learning

Clustering

Anomaly detection

the machine is presented data and asked to give you multiple classes

What can we use machine learning for?

Supervised learning

Classification

Regression

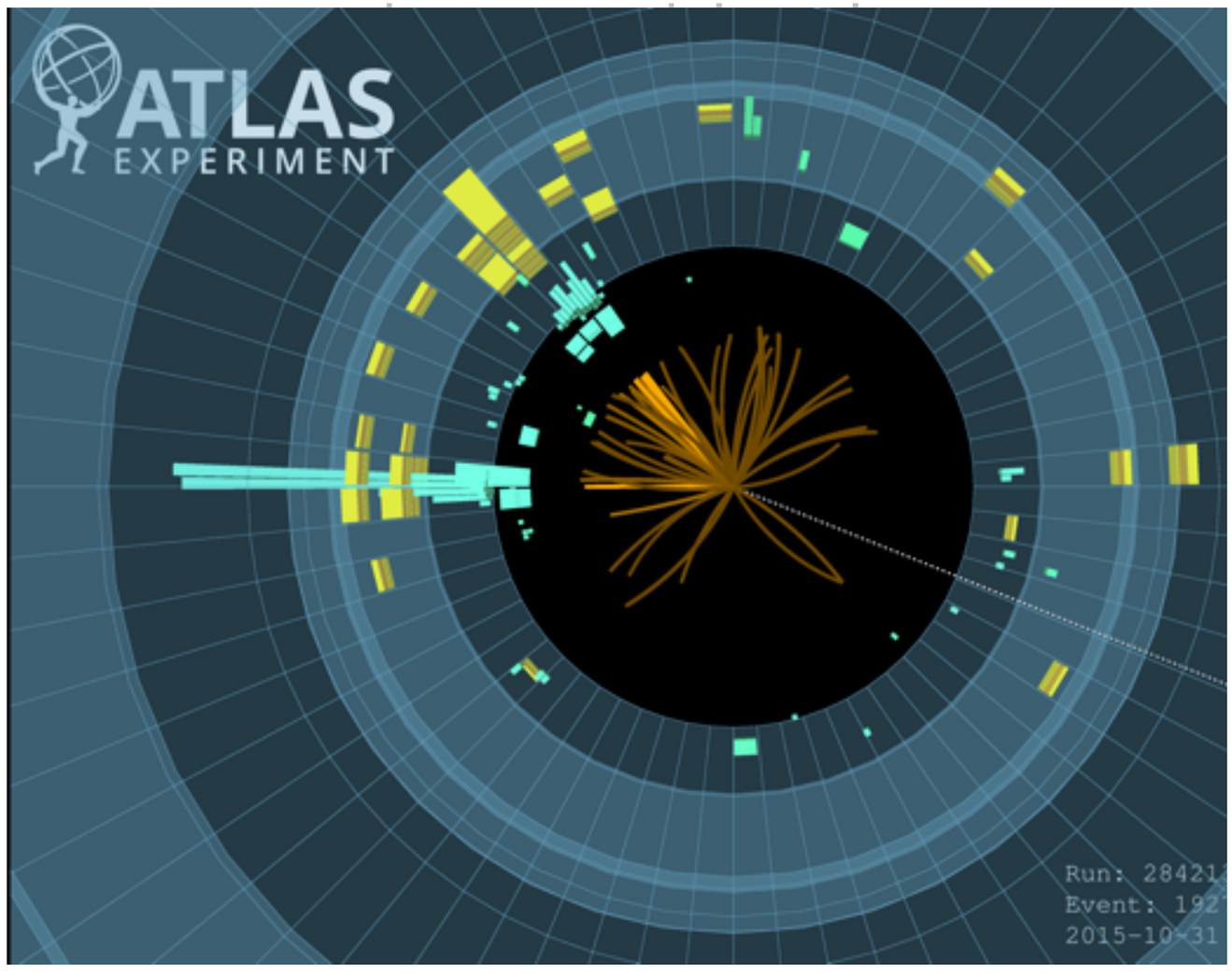
Generation

Unsupervised learning

Clustering

Anomaly detection

Higgs boson or gluon?



multiple classes

What can we use machine learning for?

Supervised learning

Classification

Regression

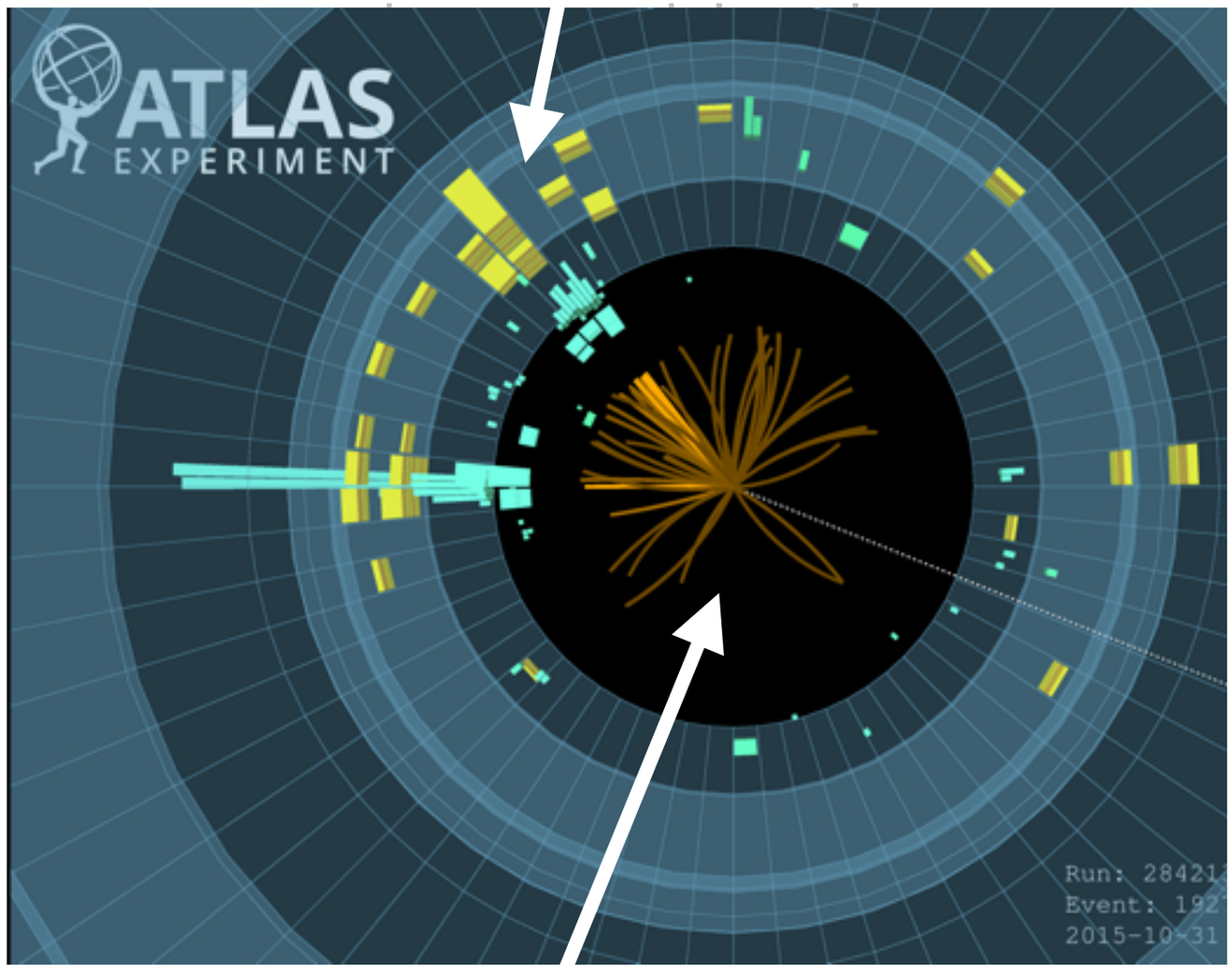
Generation

Unsupervised learning

Clustering

Anomaly detection

What is the energy of this spray of particles (jet)?



multiple classes

What are the momenta of these charged particles?

What can we use machine learning for?

Supervised learning

Classification

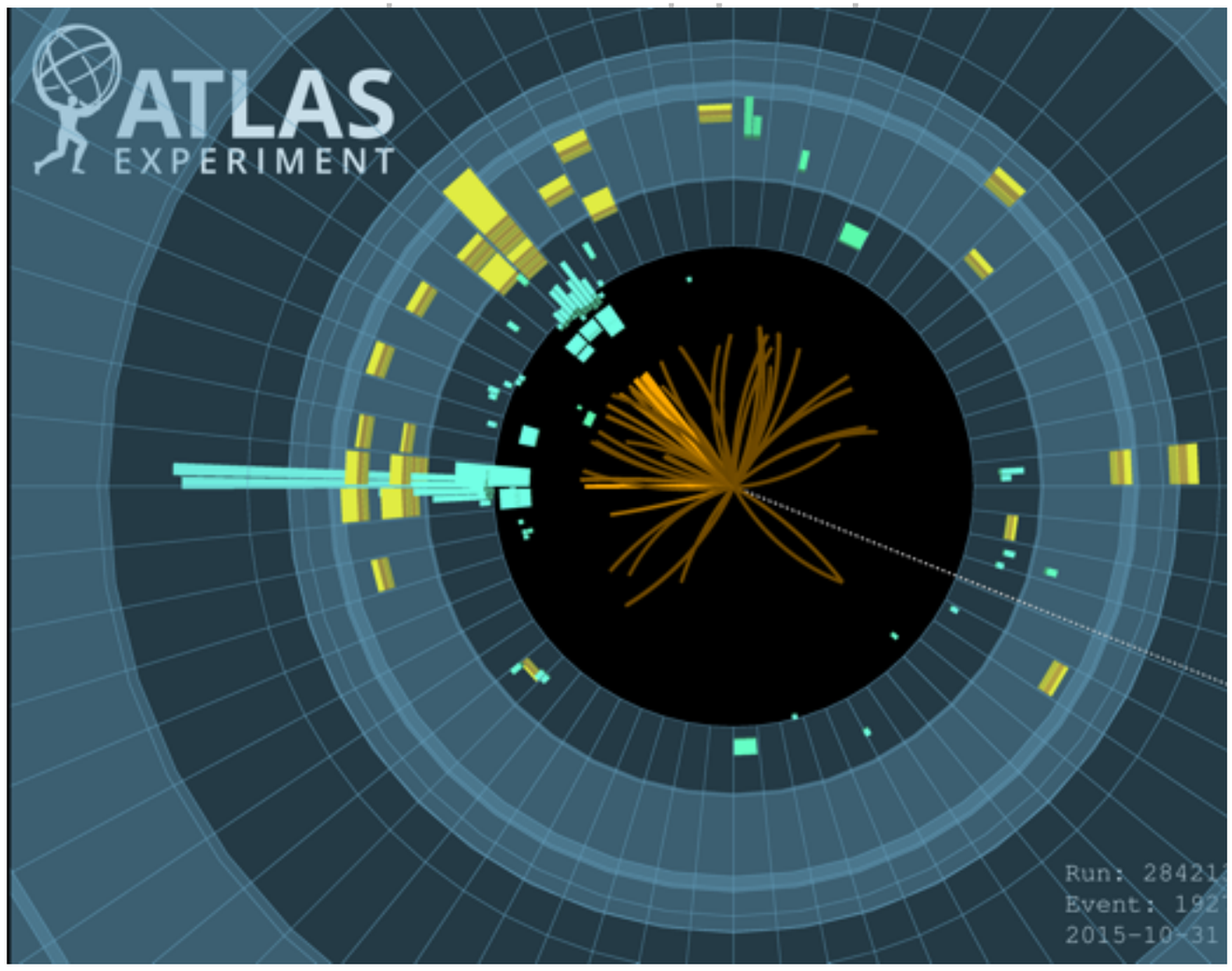
Regression

Generation

Unsupervised learning

Clustering

Anomaly detection



multiple classes

What would Higgs boson events look like with a different mass?

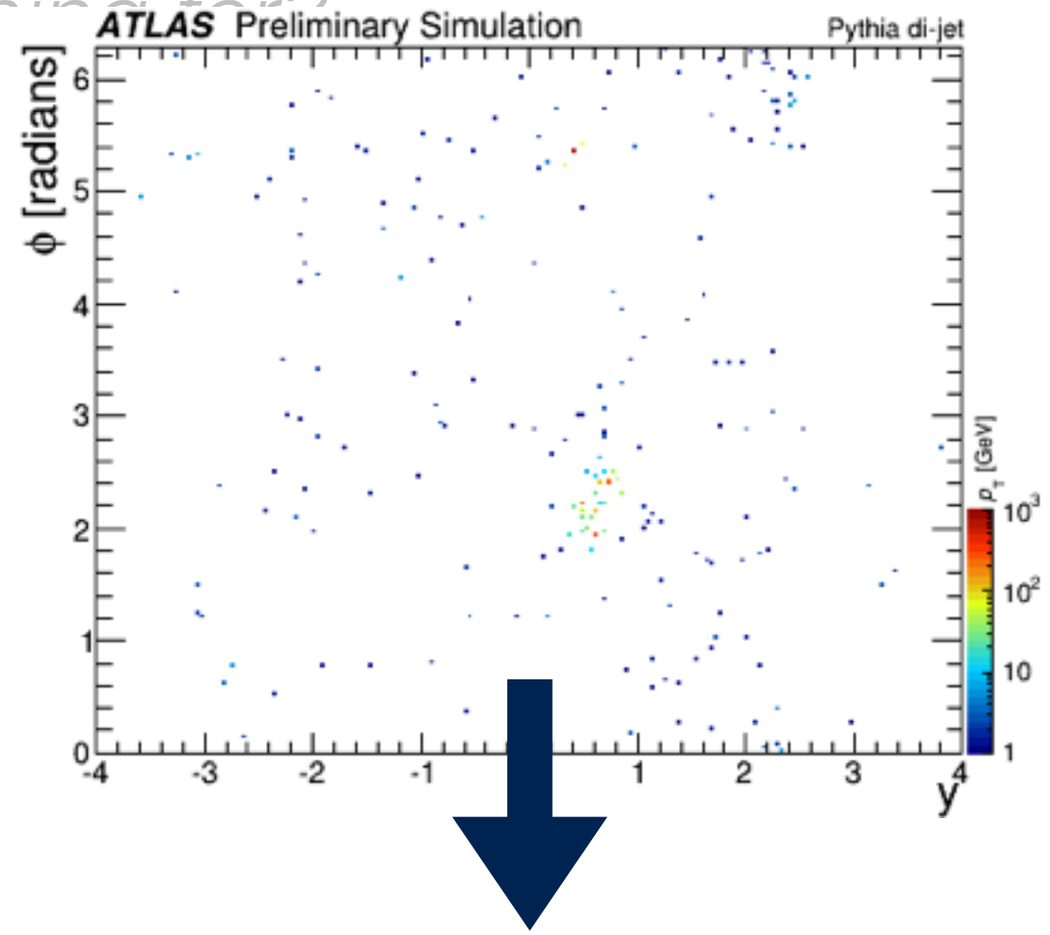
What can we use machine learning for?

Supervised learning

Classification

Regression

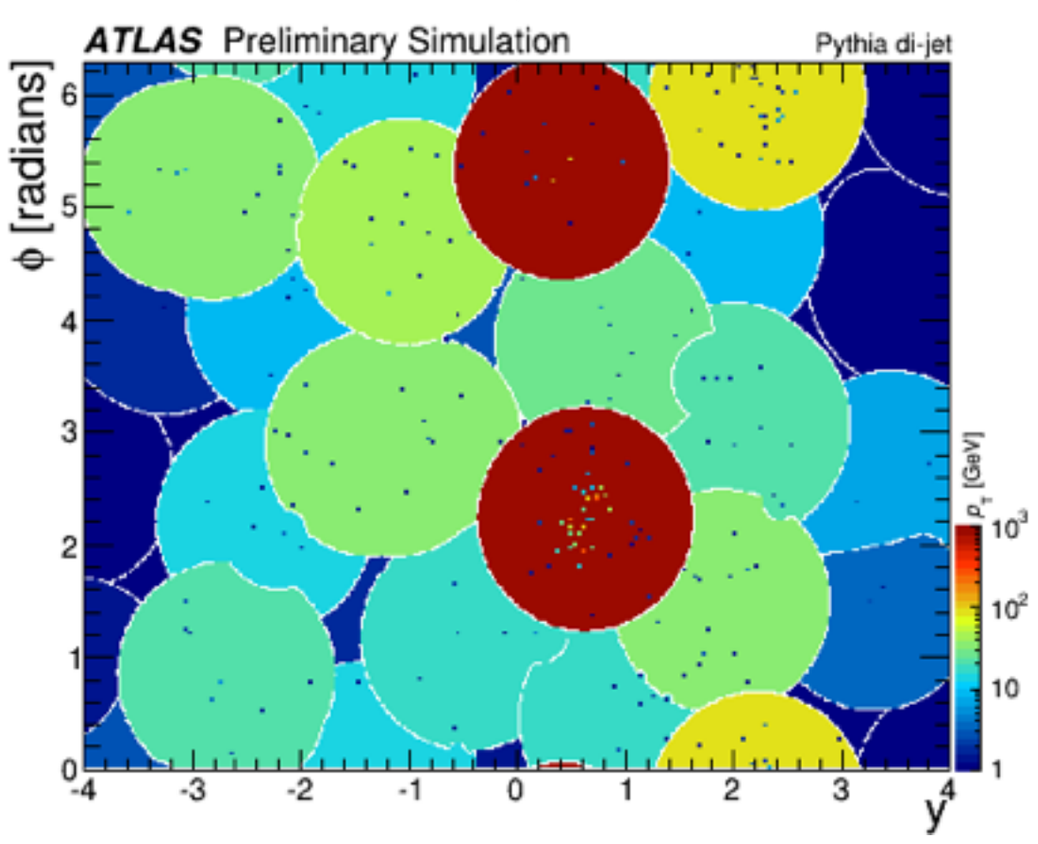
Generation



Unsupervised learning

Clustering

Anomaly detection



What can we use machine learning for?

Supervised learning

Classification

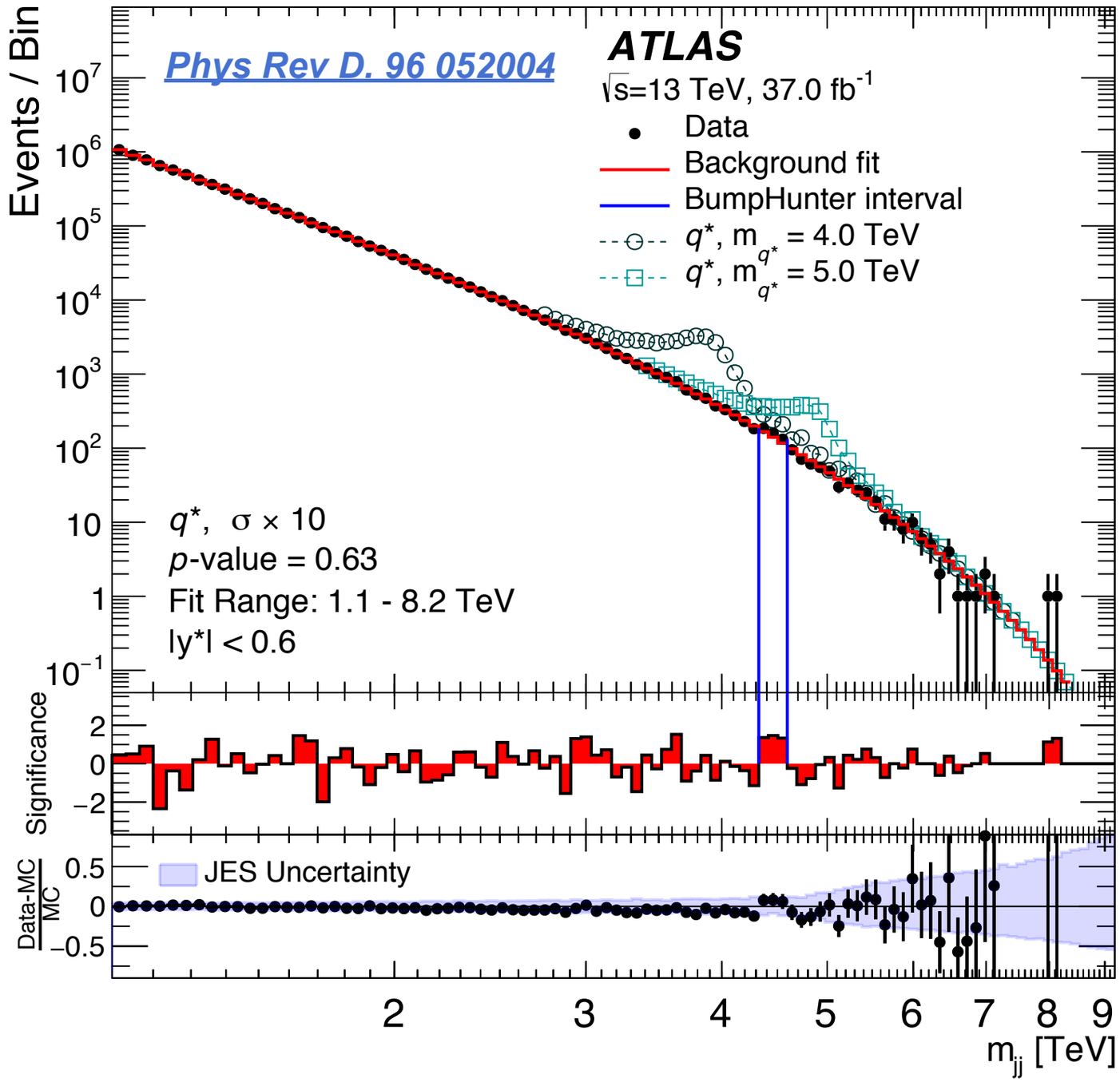
Regression

Generation

Unsupervised learning

Clustering

Anomaly detection



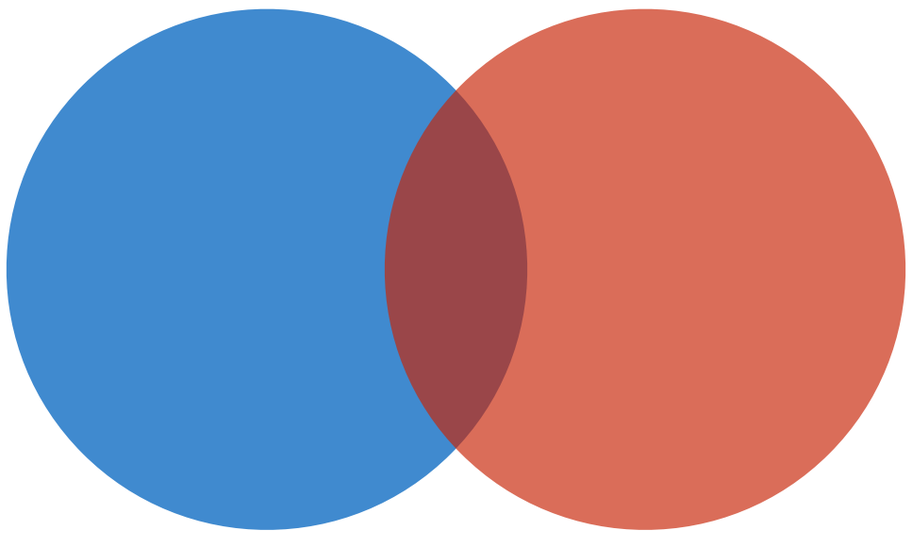
Classification

Goal: Given a *feature vector*, return an integer indexed by the set of possible *classes.*

In most cases, we care about *binary* classification in which there are only two classes (signal versus background)

There are some cases where we care about *multi-class classification*

Feature vector can be many-dimensional



Harder = more overlap between for **S** and **B**

Classification

Goal: Given a *feature vector*, return an integer indexed by the set of possible *classes.*

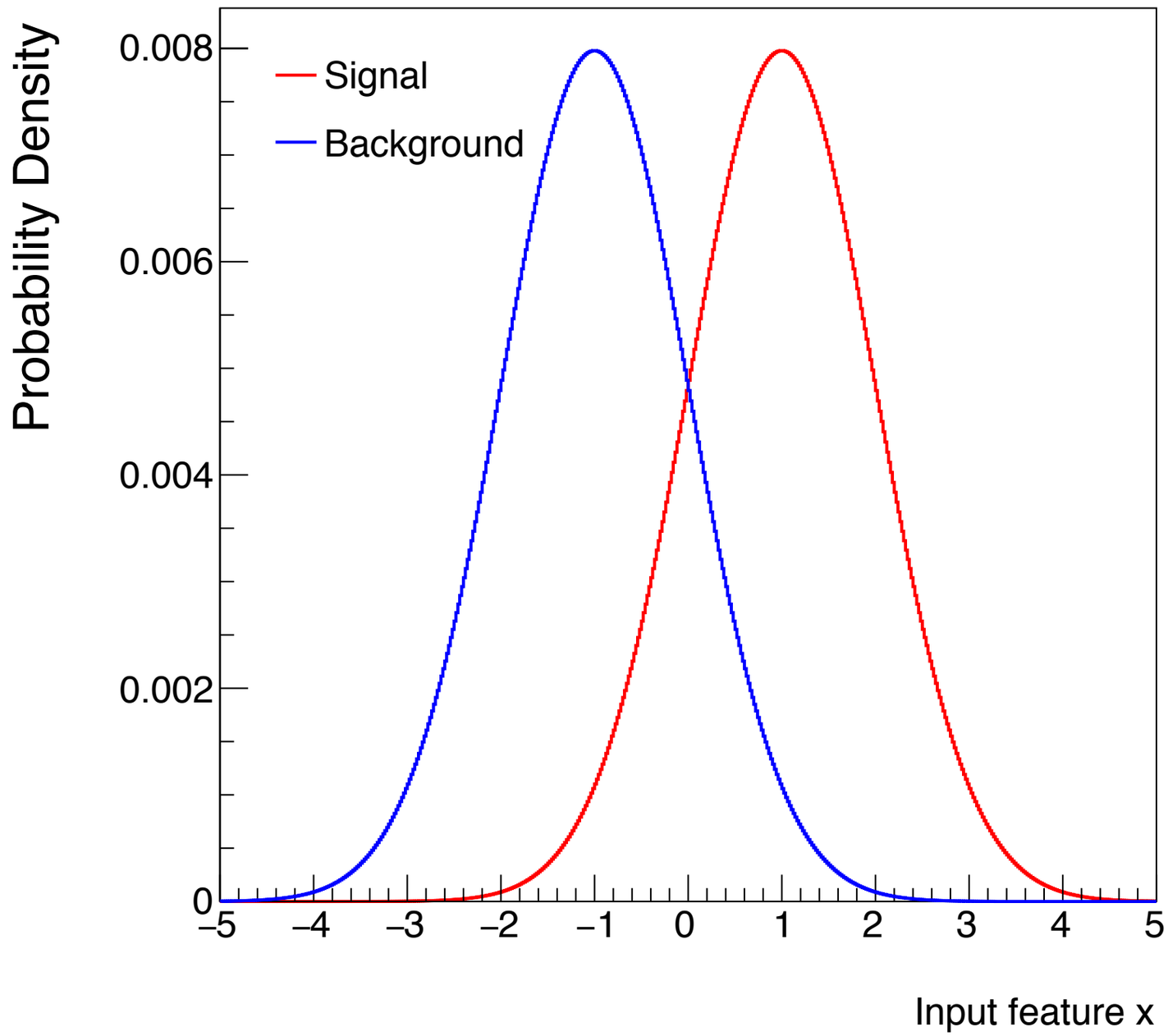
In practice, we don't just want one classifier, but an entire set of classifiers indexed by:

True Positive Rate = signal efficiency =
Pr(label signal | signal) = sensitivity

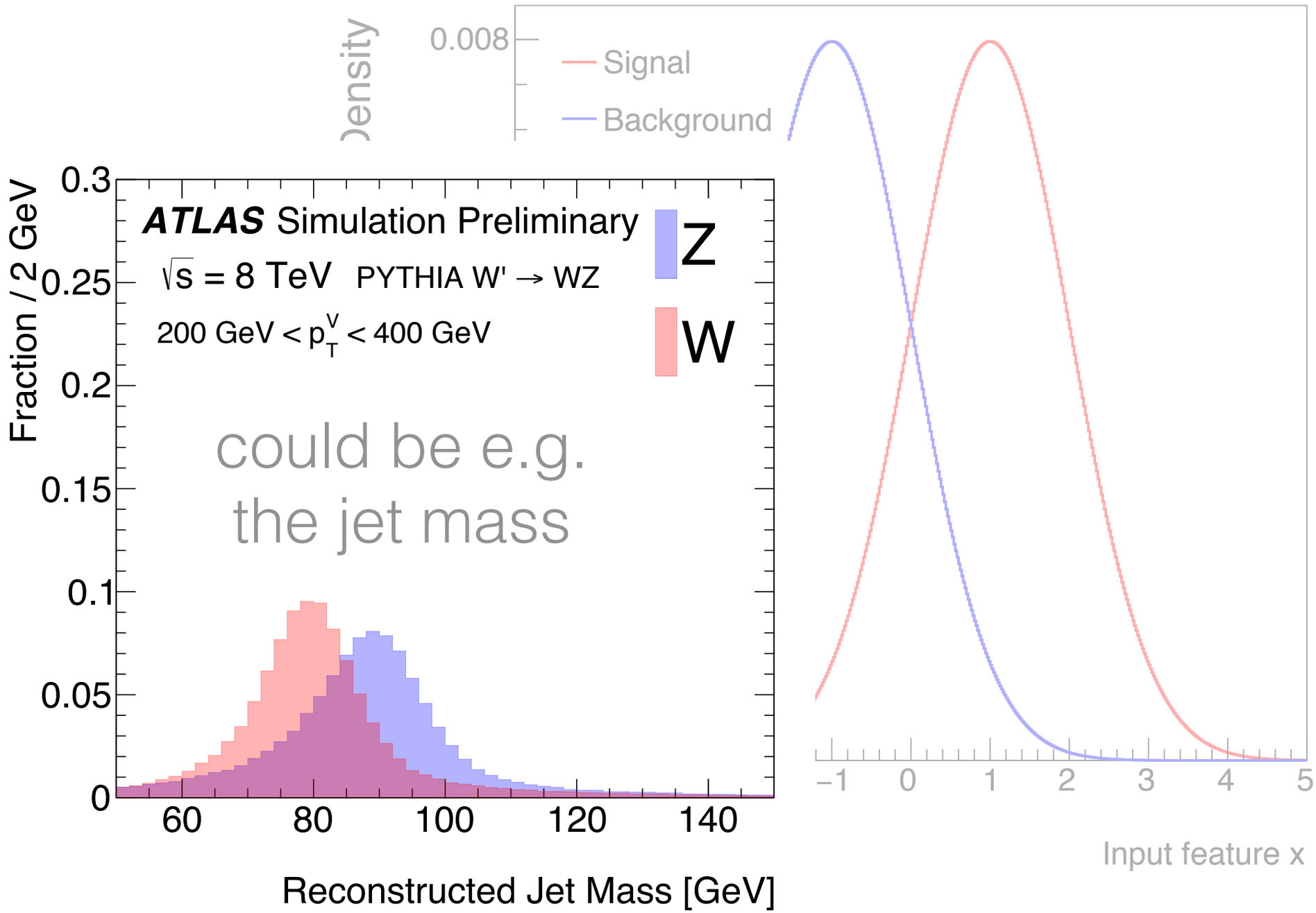
True Negative Rate = 1 - background efficiency
= 1 - false positive rate (FPR)
rejection = Pr(label background | background) = specificity

For a given TPR, we want the lowest possible TNR!

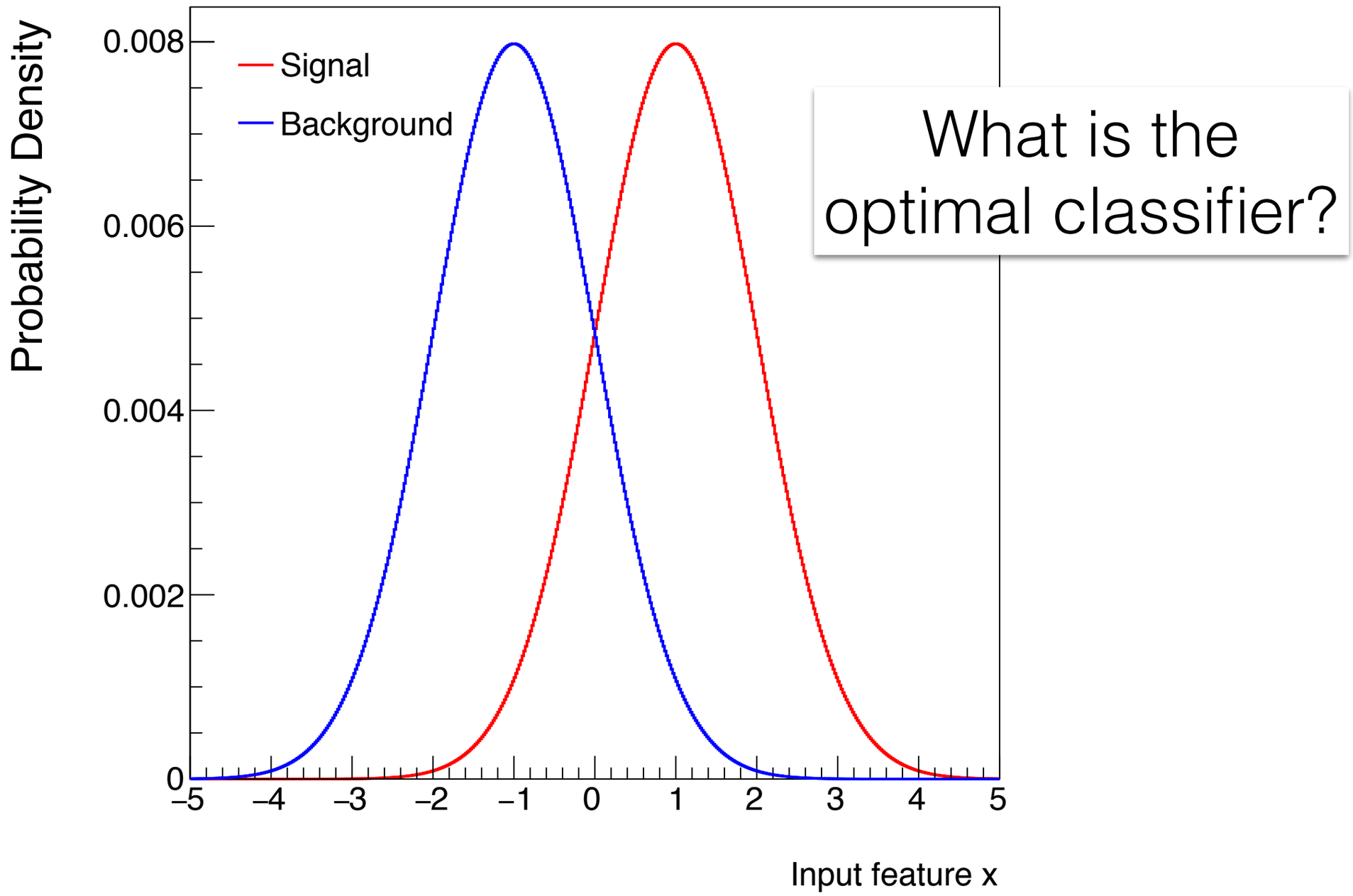
Let's consider an important special case: binary classification in 1D



Let's consider an important special case: binary classification in 1D

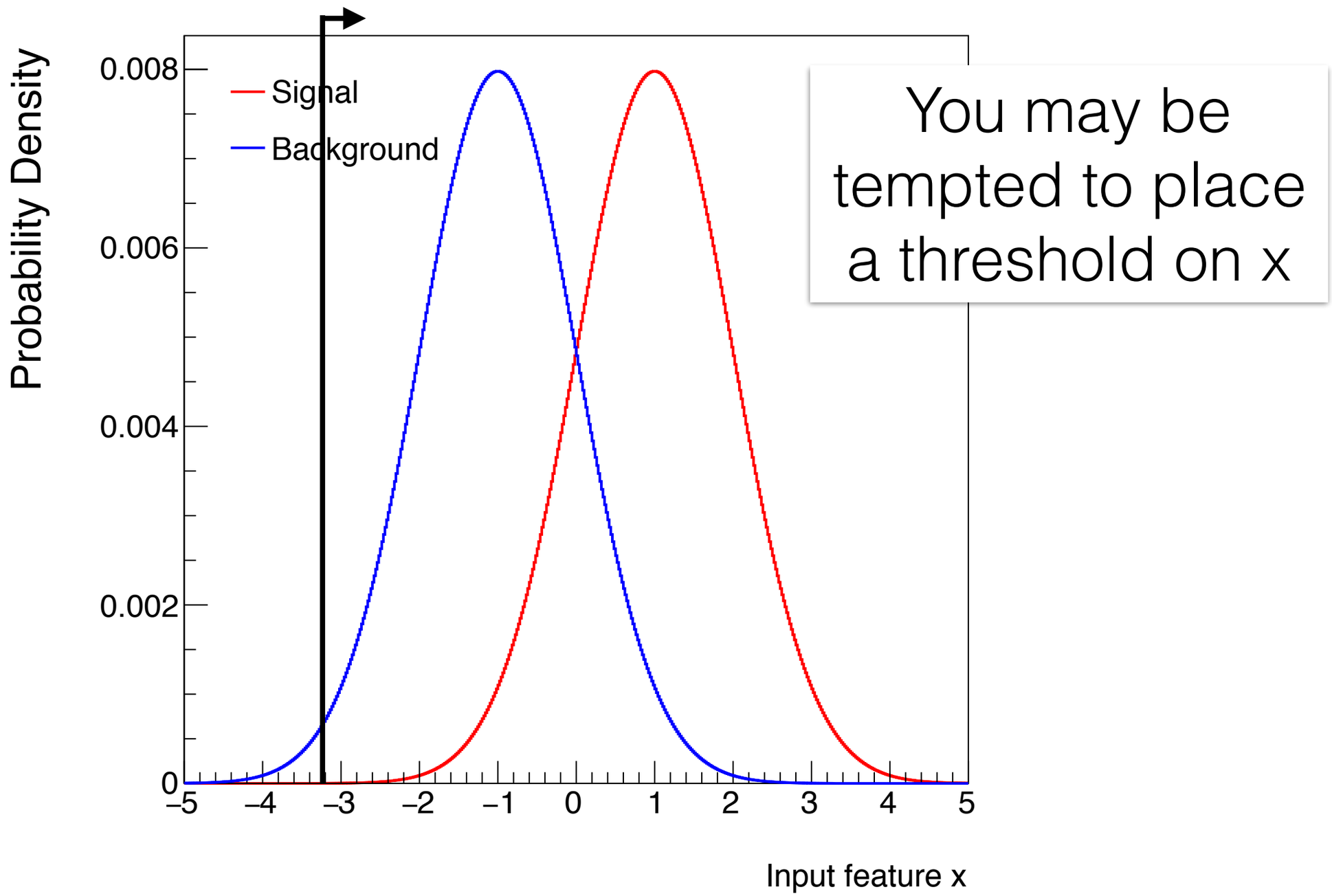


Let's consider an important special case: binary classification in 1D



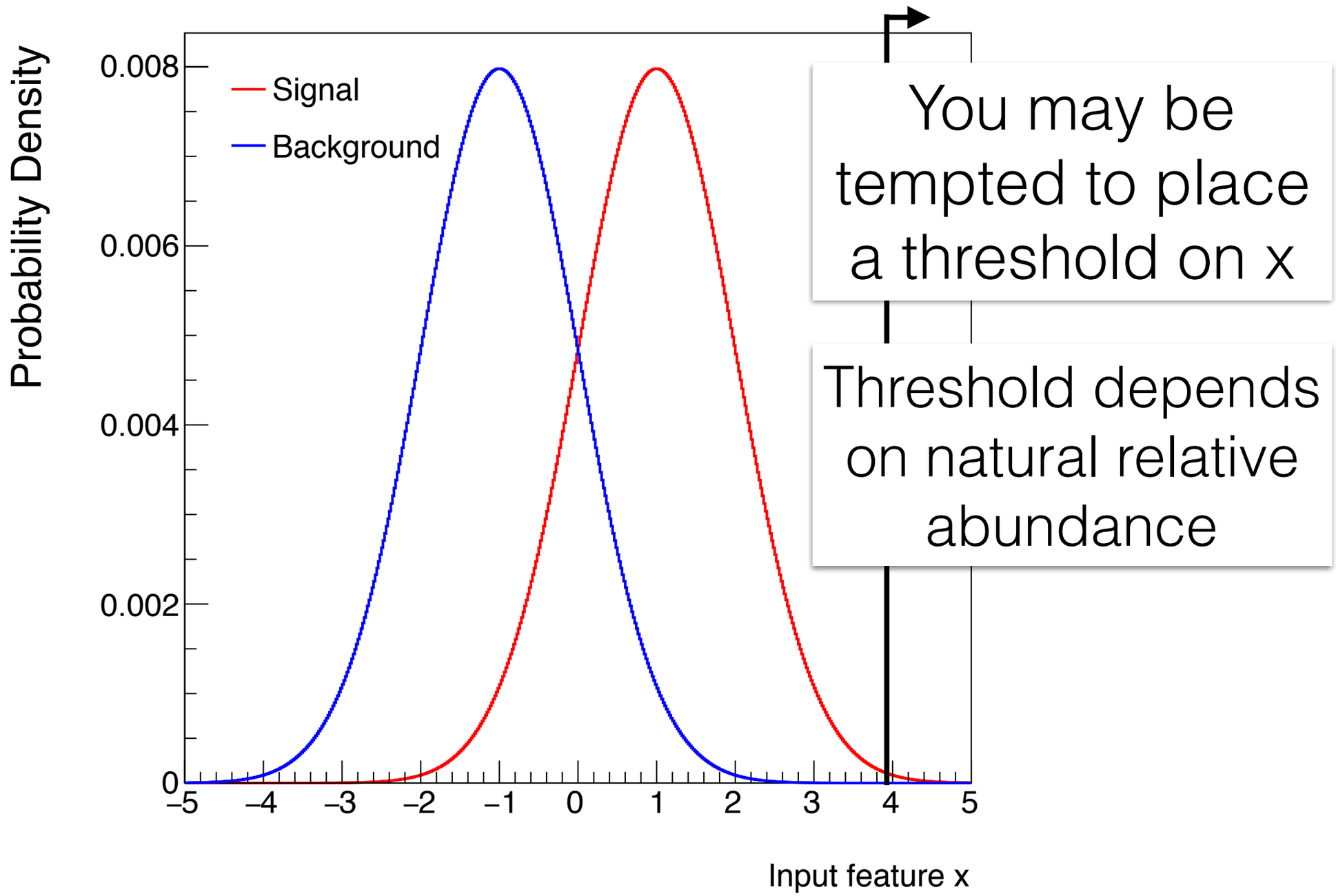
What is the optimal classifier?

Let's consider an important special case: binary classification in 1D



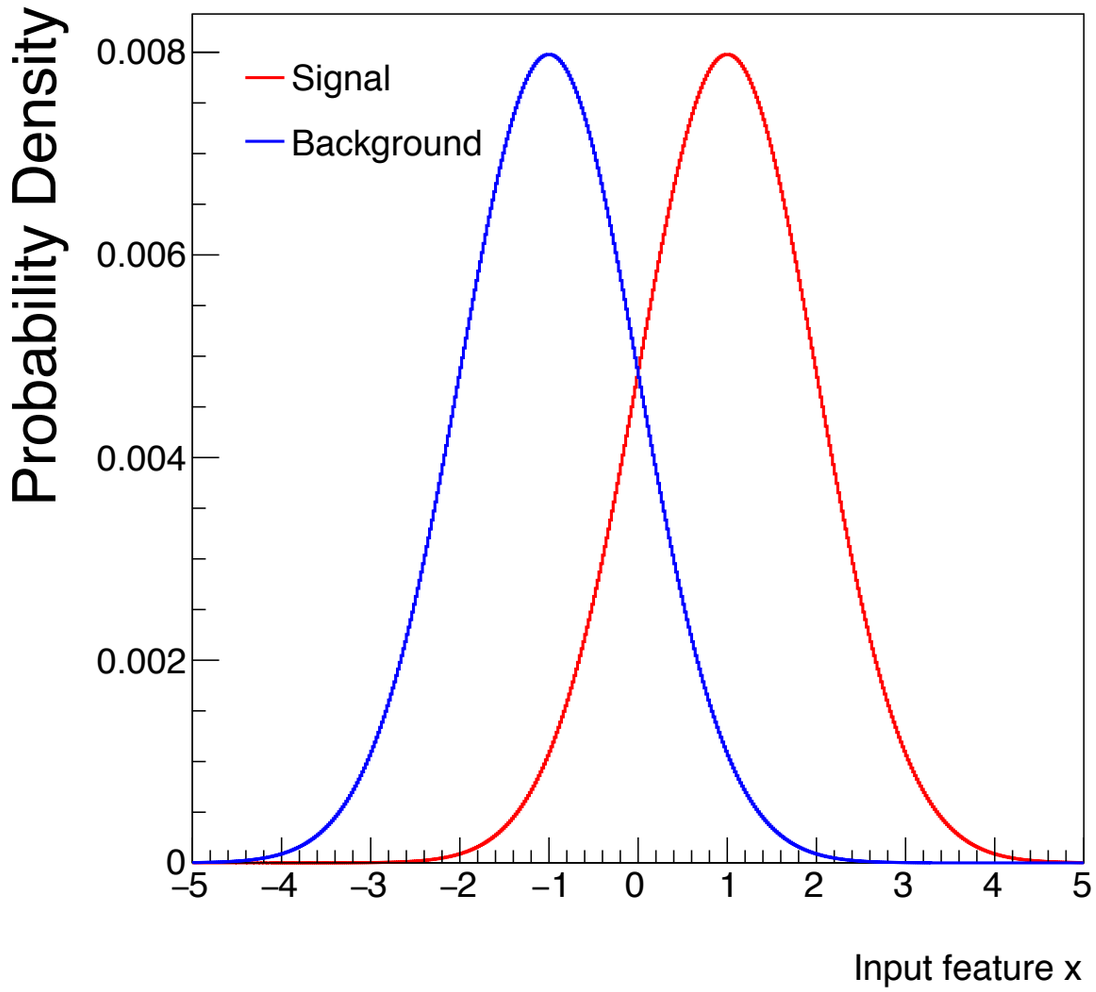
You may be tempted to place a threshold on x

Let's consider an important special case: binary classification in 1D



You may be tempted to place a threshold on x

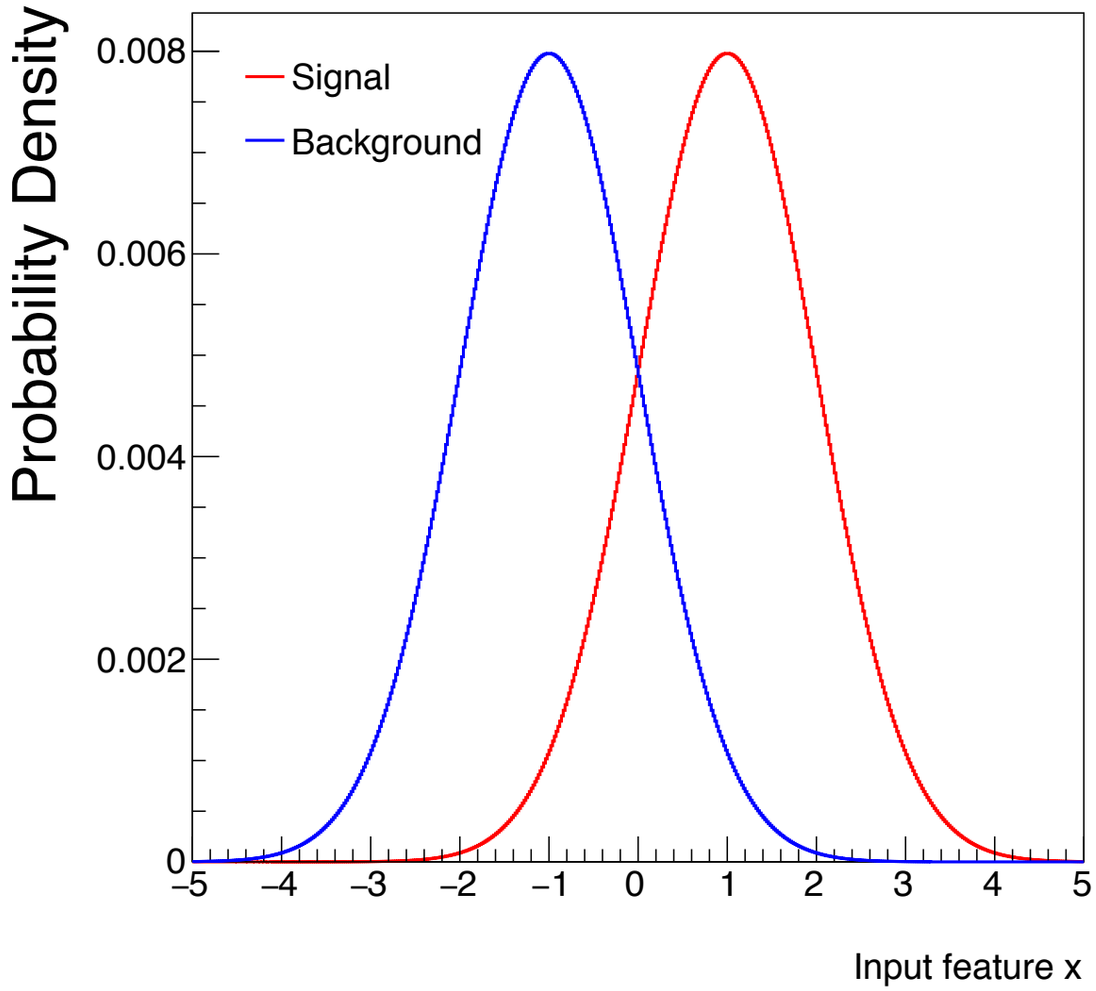
Threshold depends on natural relative abundance



Is the simple threshold cut **optimal**?

Fact 1: The classifier that results in the lowest FPR for a given TPR is a cut on the **likelihood ratio (LR)**.

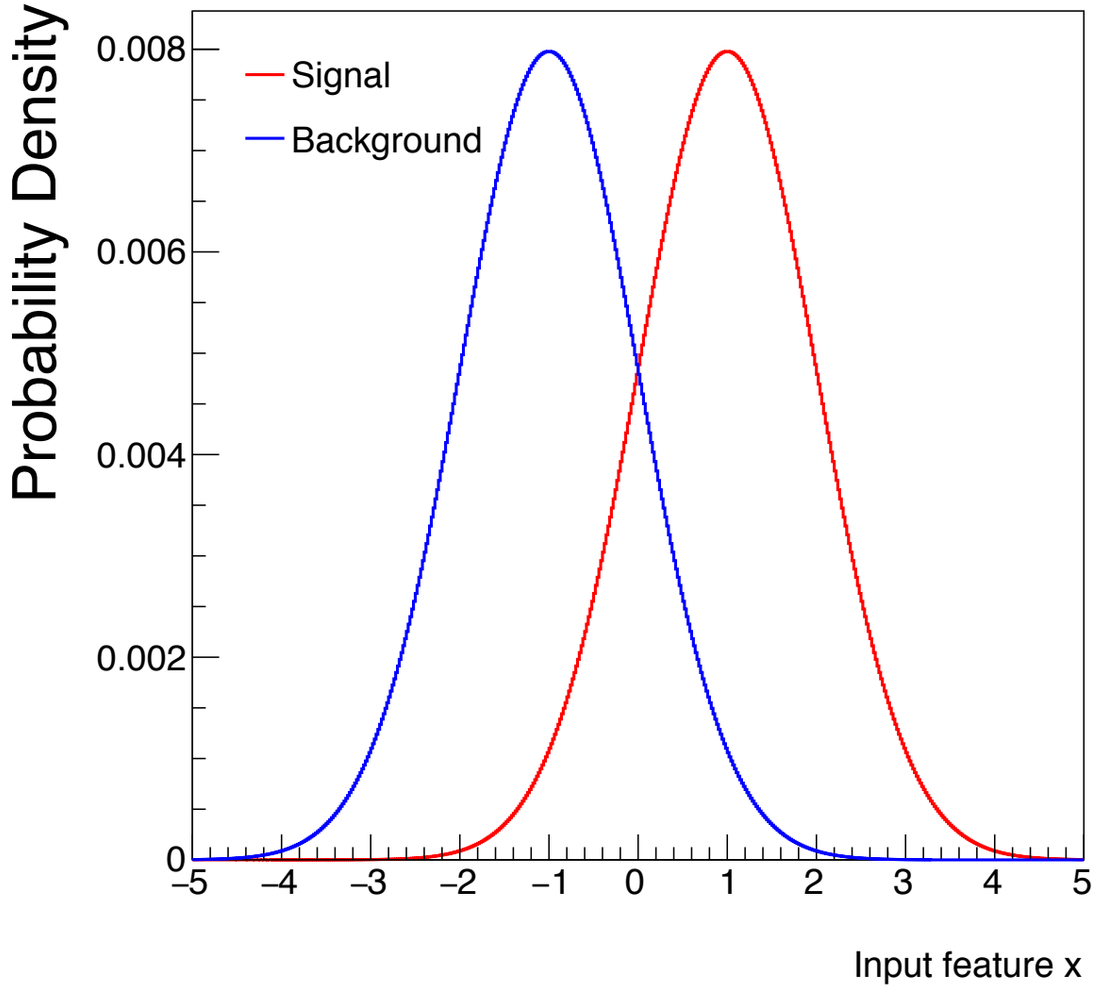
$$LR(x) > c,\ LR(x) = p(x|\text{signal}) \,/\, p(x|\text{background})$$



Is the simple threshold cut **optimal**?

Fact 1: The classifier that results in the lowest FPR for a given TPR is a cut on the **likelihood ratio (LR)**.

Fact 2: Two classifiers that are related by a **monotonic transformation** result in the same performance.

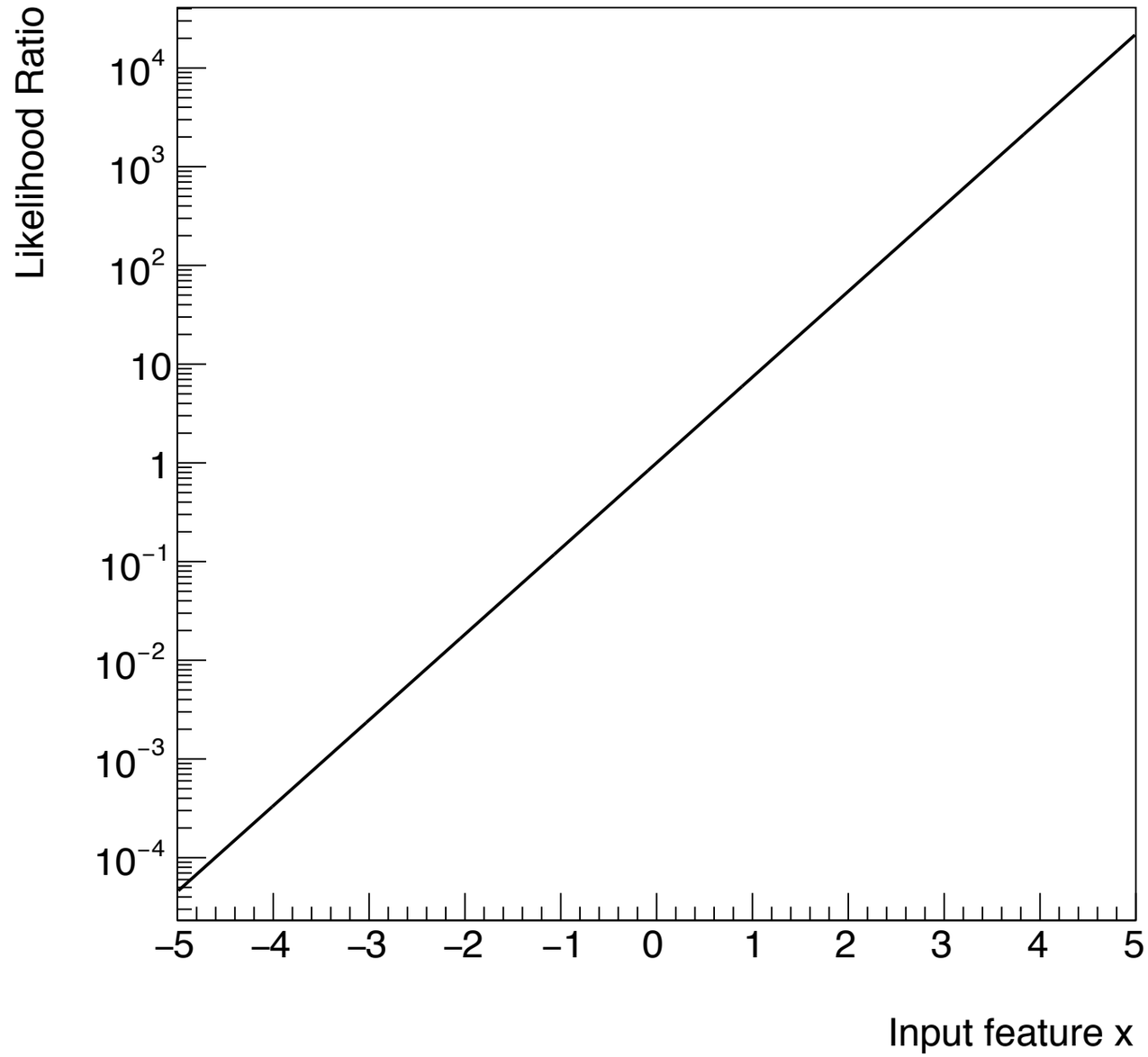


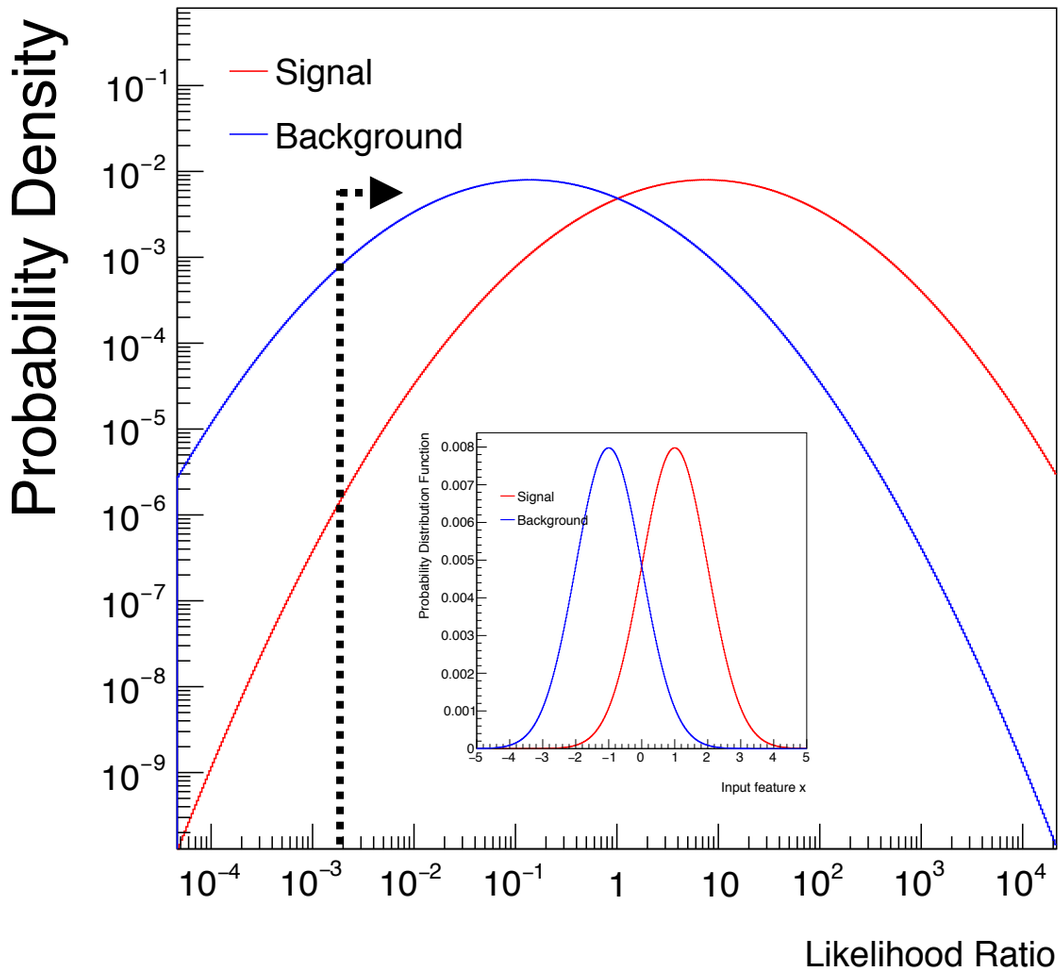
Is the simple threshold cut **optimal**?

In this simple case, the log LR is proportional to x:

no need for non-linearities!

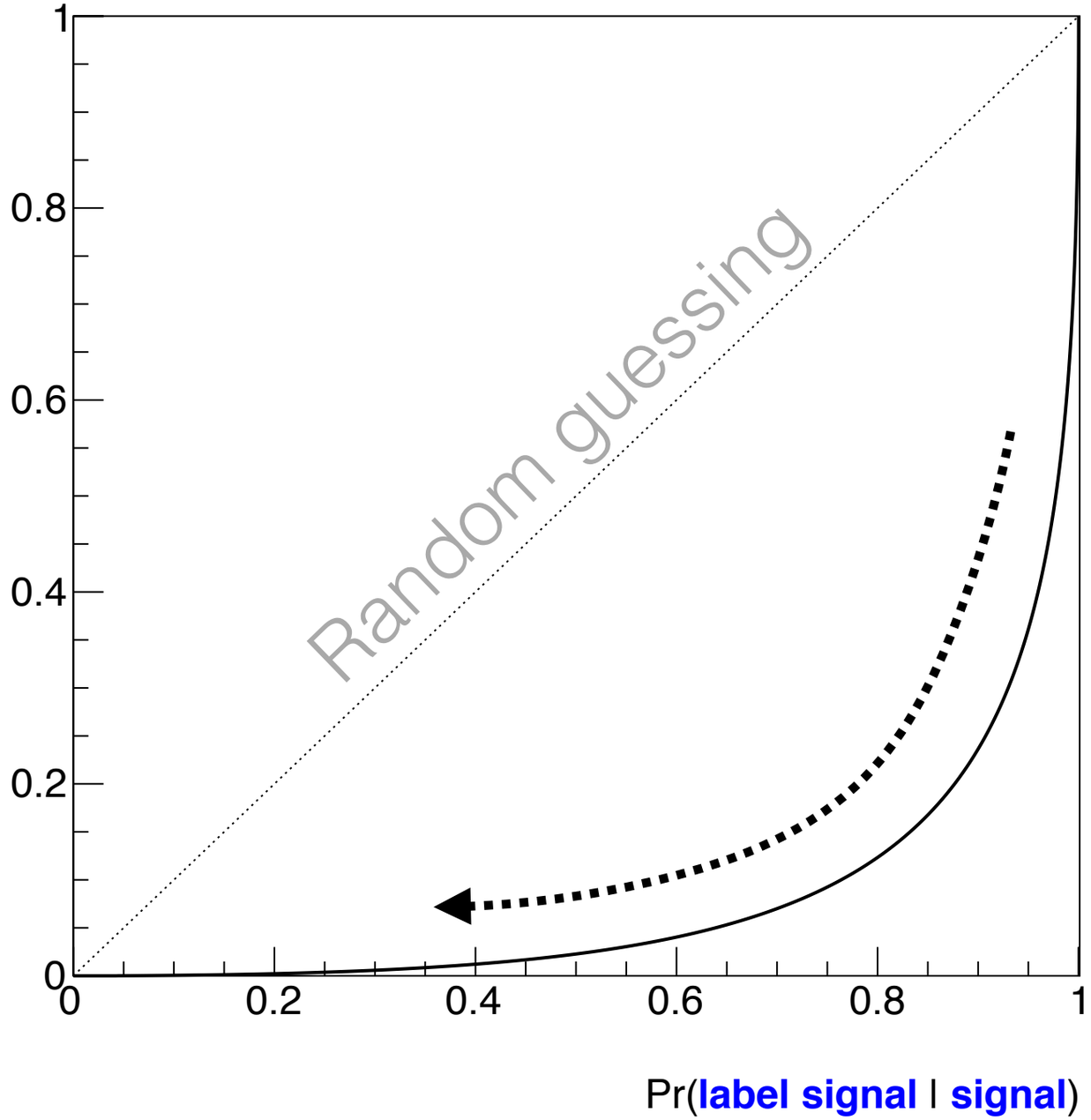
Threshold cut is optimal





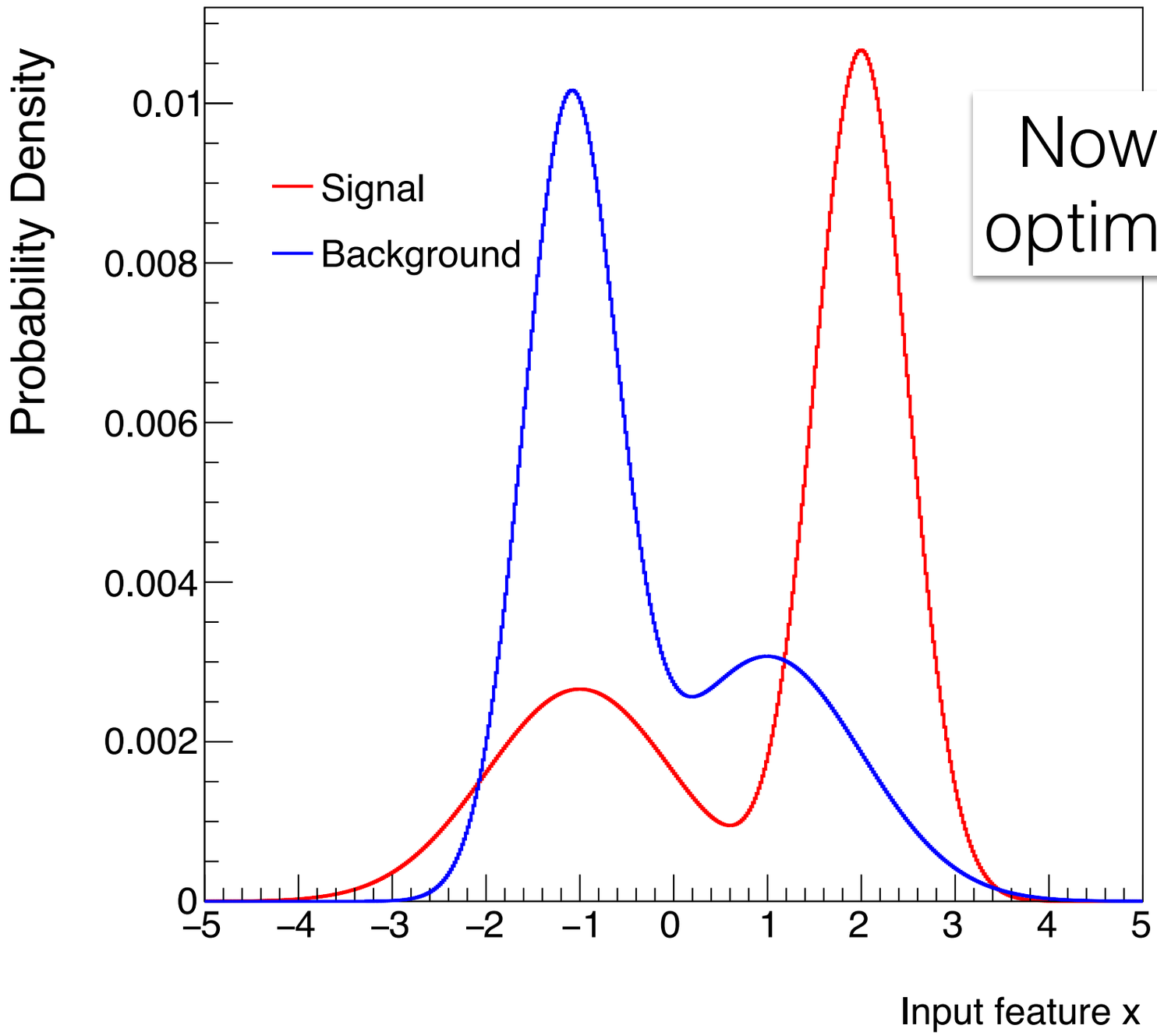
The optimal procedure is a threshold on the LR

"Receiver Operating Characteristic" (**ROC**) Curve

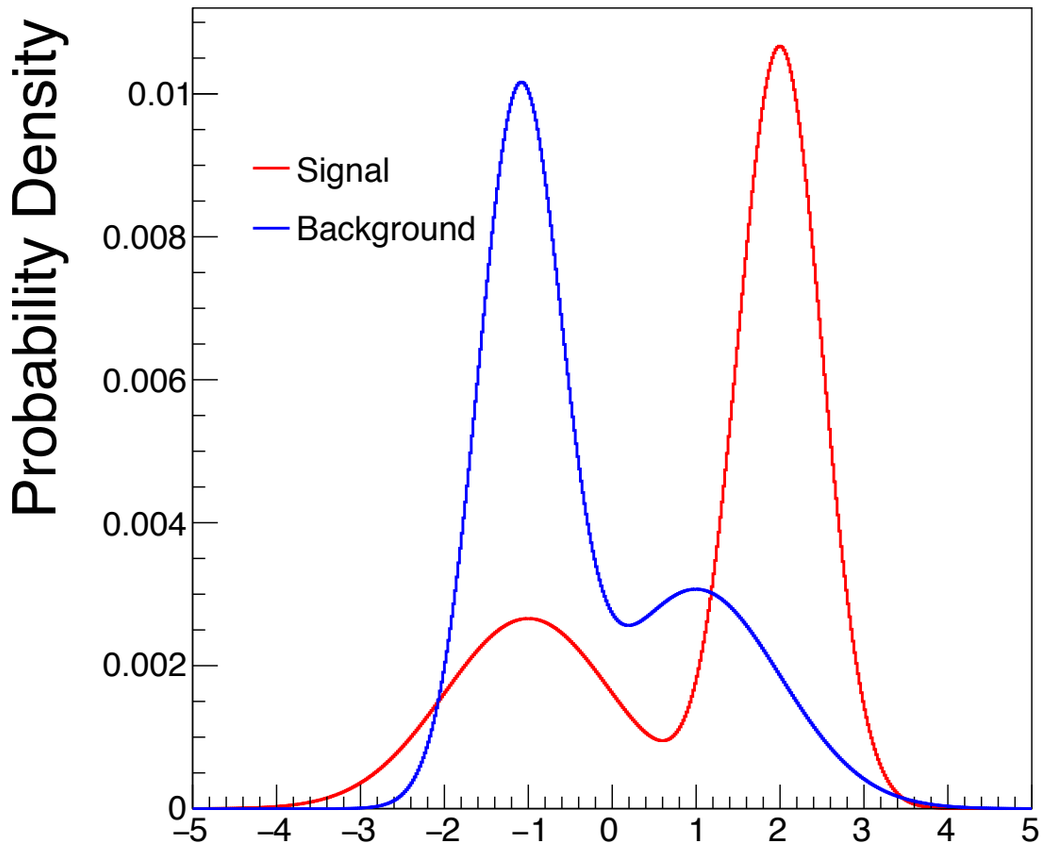


What if the distribution of x is complicated?

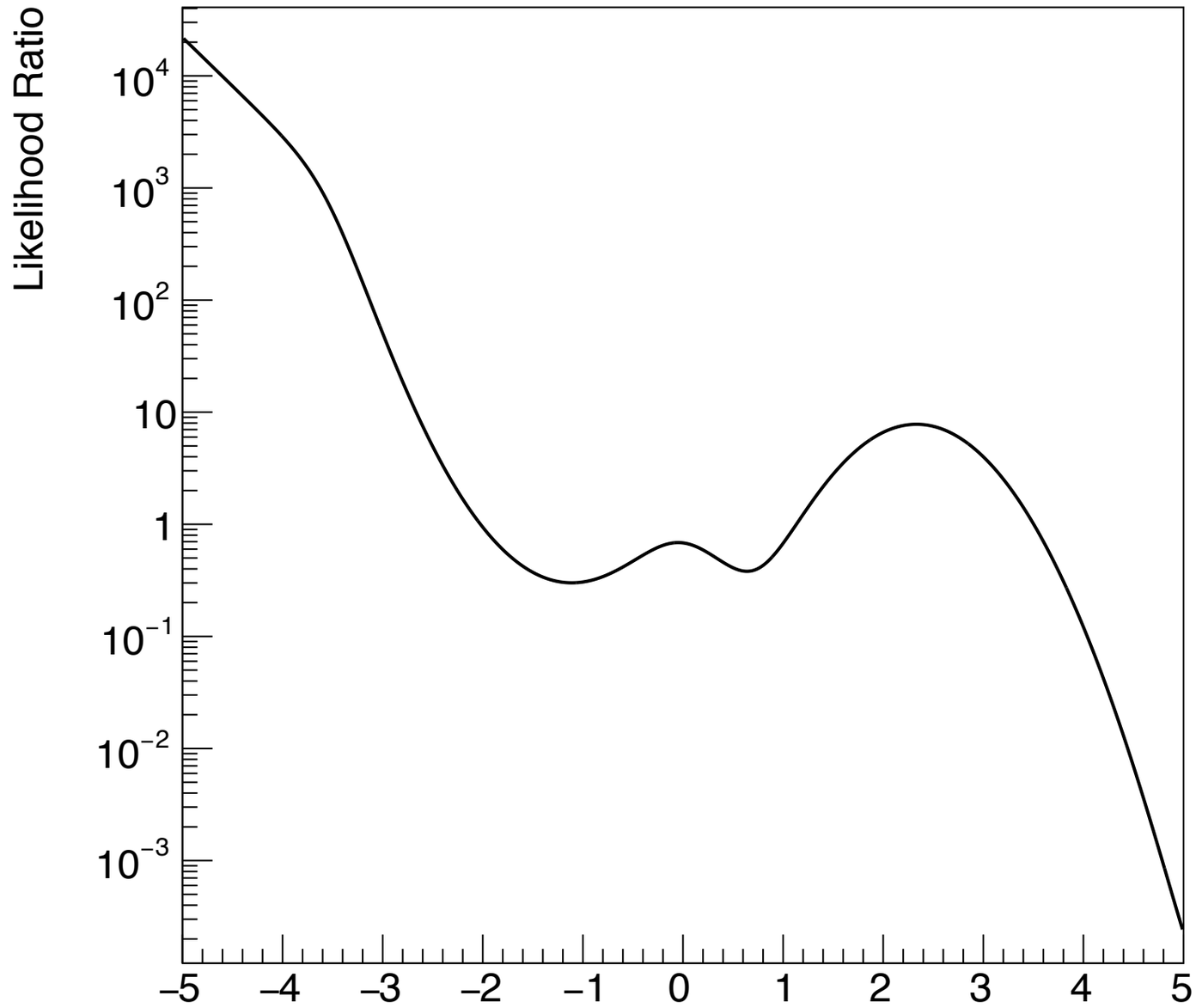
Real life is complicated!



Now what is the optimal classifier?

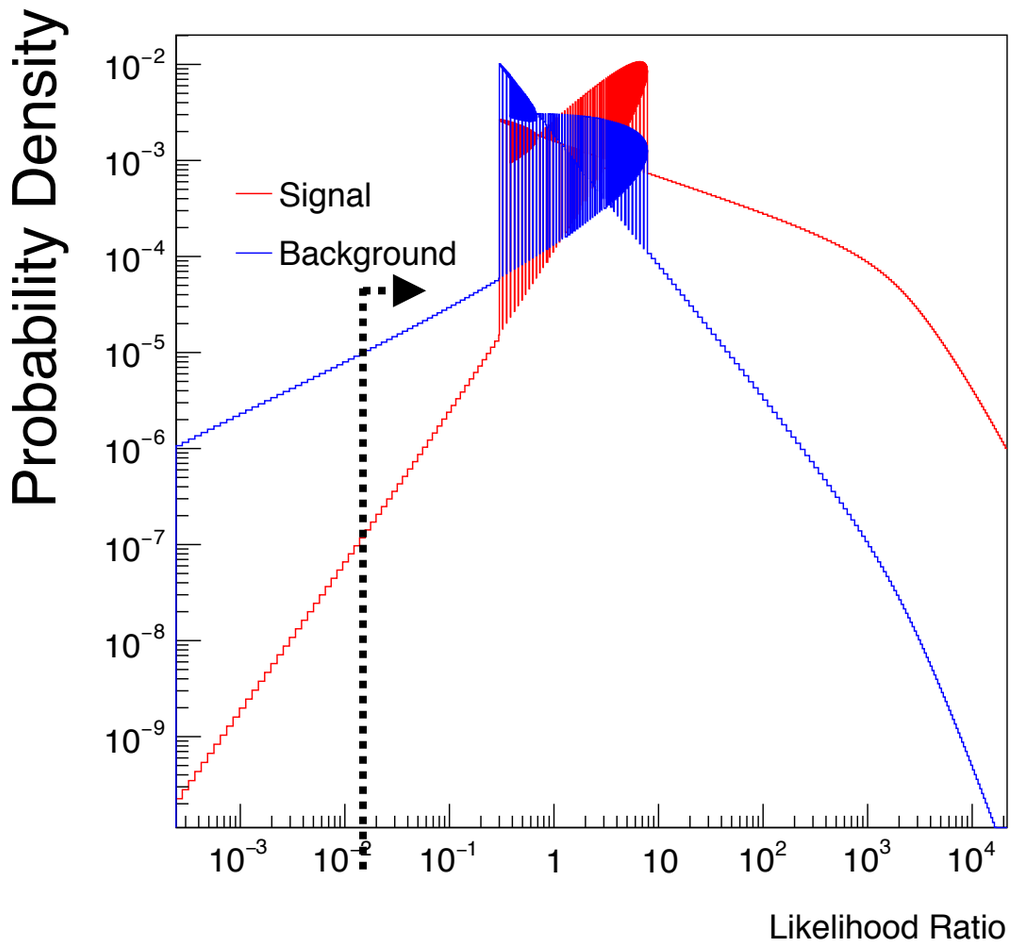


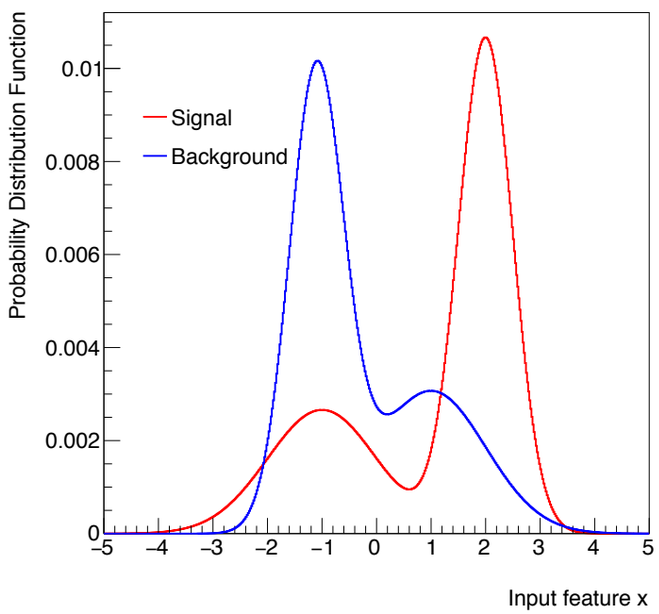
In this case, LR is highly non-linear (**non-monotonic**) function of x

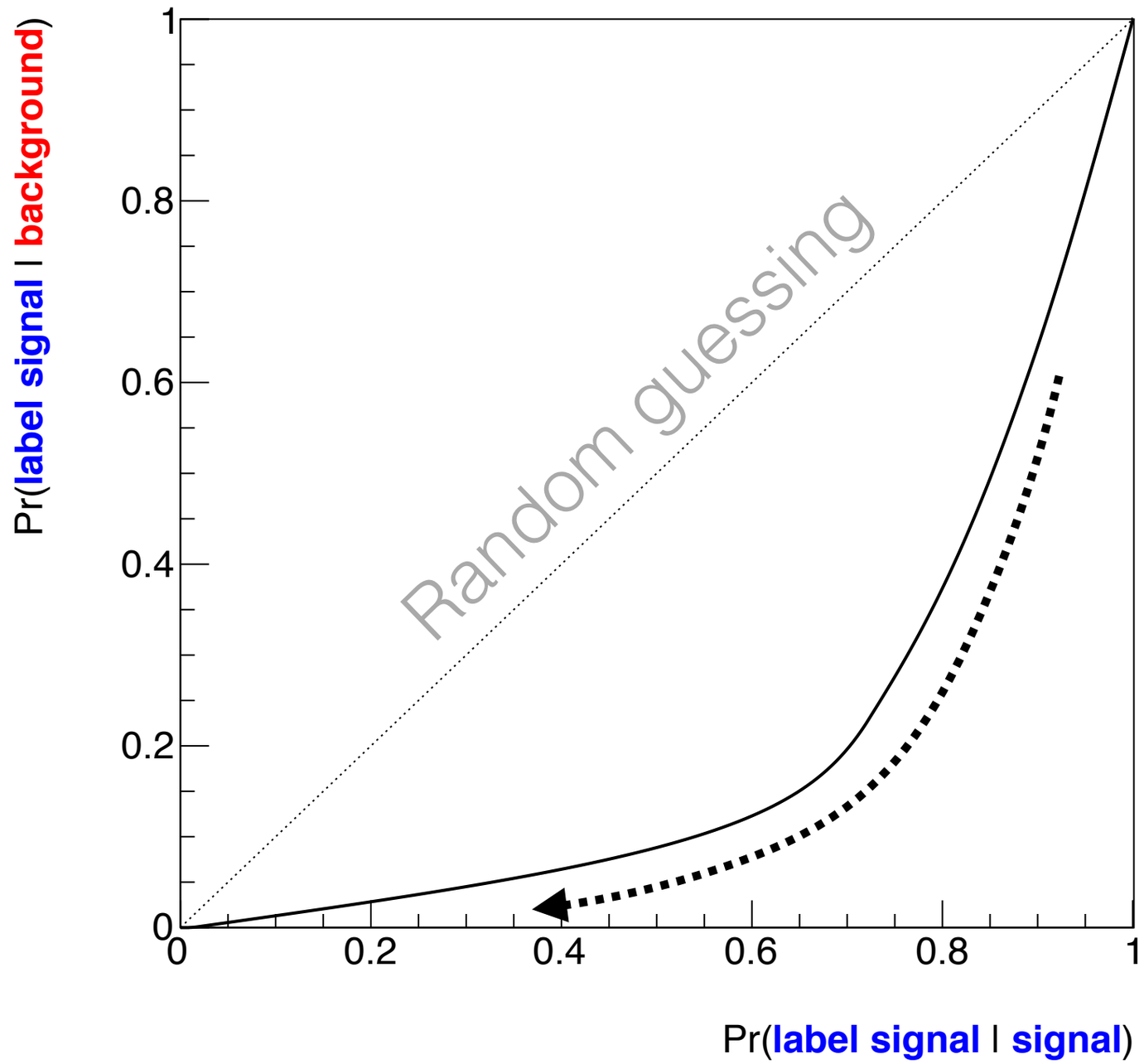


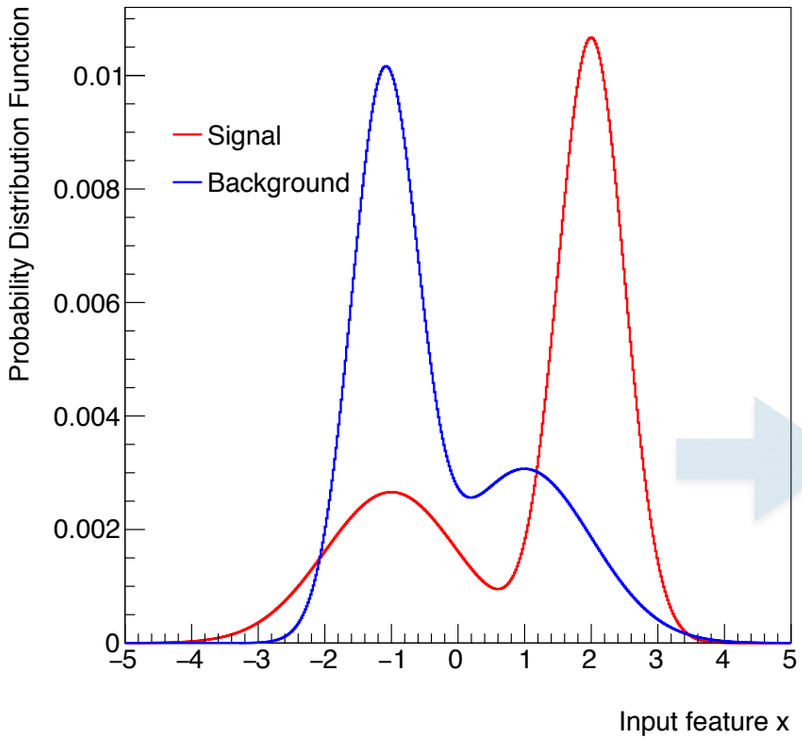
A threshold on x would be sub-optimal

ROC is worse than the Gaussians, but that is expected since the overlap in their PDFs is higher.





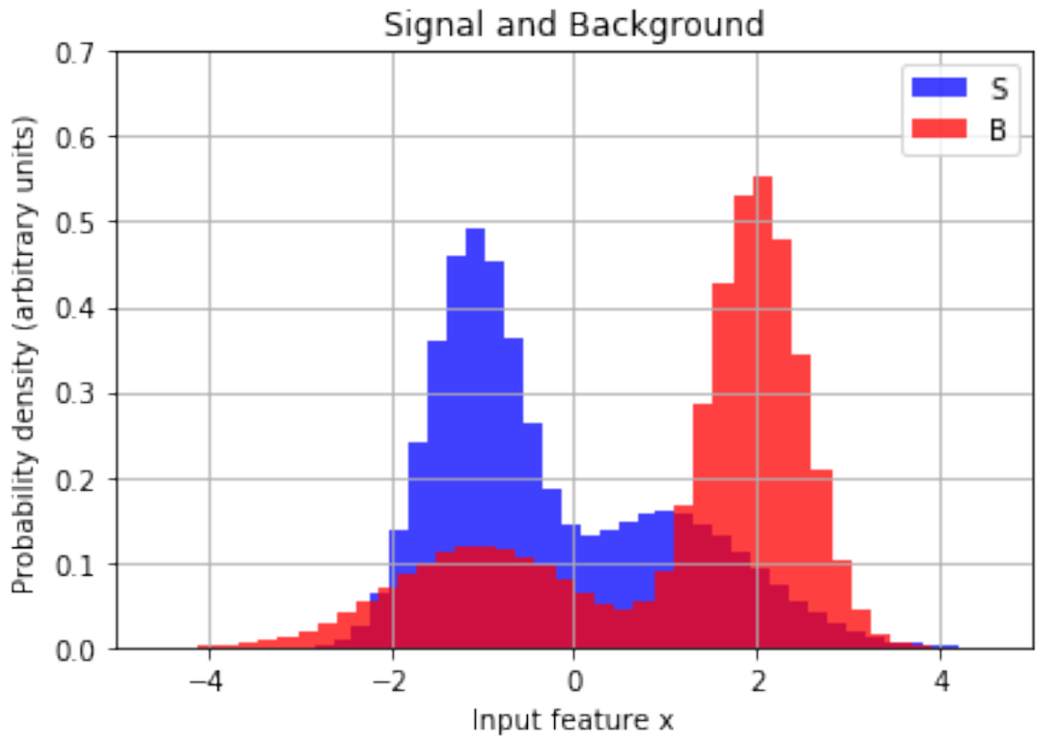


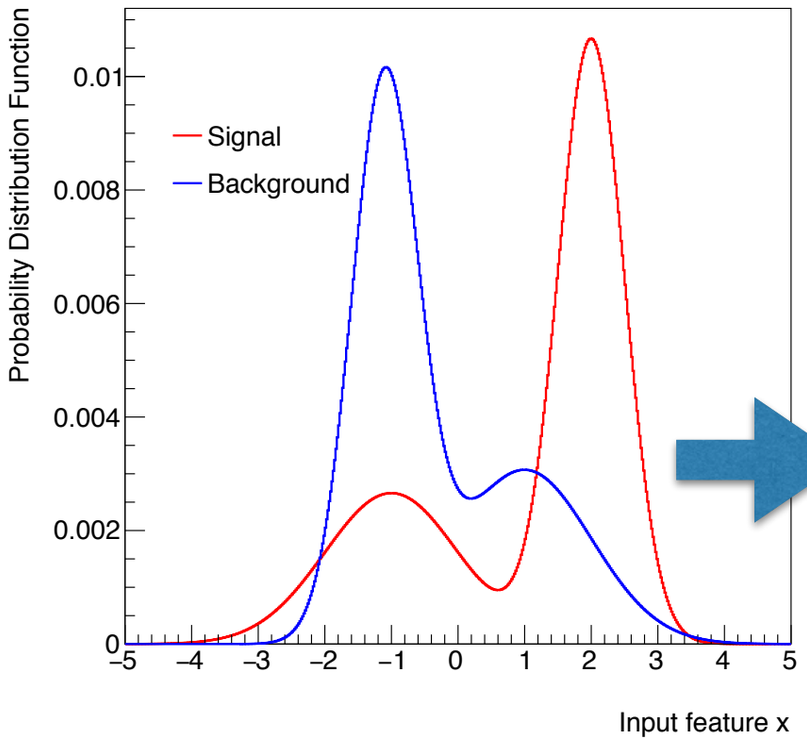


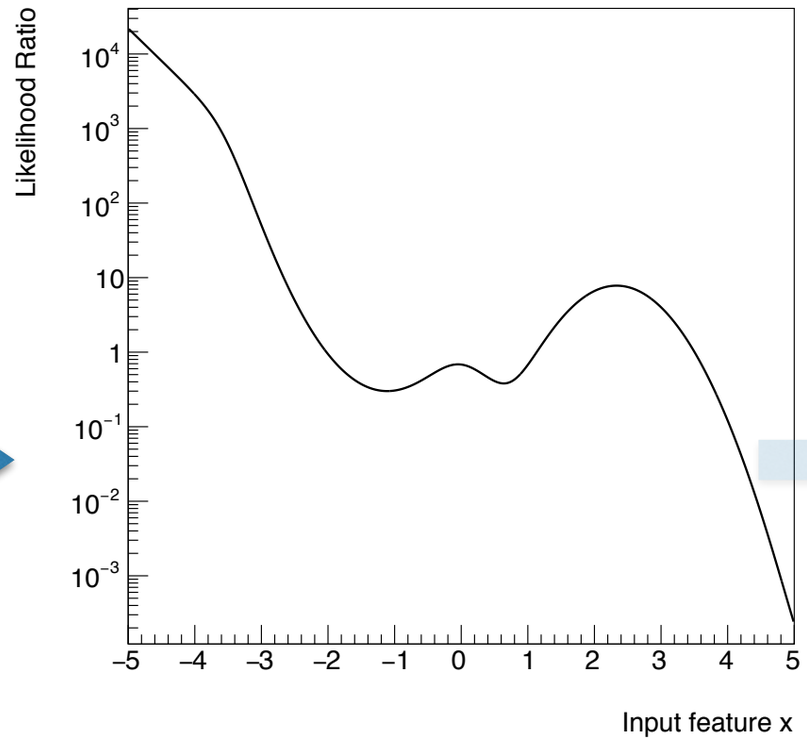

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
```

```
In [2]: N = 10000000
        mu1a, sigma1a = -1.1, 0.5
        x1a = np.random.normal(mu1a, sigma1a, int(0.6*N))
        mu1b, sigma1b = 1, 1
        x1b = np.random.normal(mu1b, sigma1b, int(0.4*N))

        mu2a, sigma2a = 2, 0.5
        x2a = np.random.normal(mu2a, sigma2a, int(0.7*N))
        mu2b, sigma2b = -1, 1
        x2b = np.random.normal(mu2b, sigma2b, int(0.3*N))

        signal = np.append(x1a,x1b)
        background = np.append(x2a,x2b)
```

```
In [3]: plt.hist(signal, 50, normed=1, facecolor='blue', alpha=0.75, label='S')
        plt.hist(background, 50, normed=1, facecolor='red', alpha=0.75, label='B')
        plt.xlabel('Input feature x')
        plt.ylabel('Probability density (arbitrary units)')
        plt.title(r'Signal and Background')
        plt.legend(loc='upper right')
        plt.axis([-5, 5, 0, 0.7])
        plt.grid(True)
        plt.show()
```

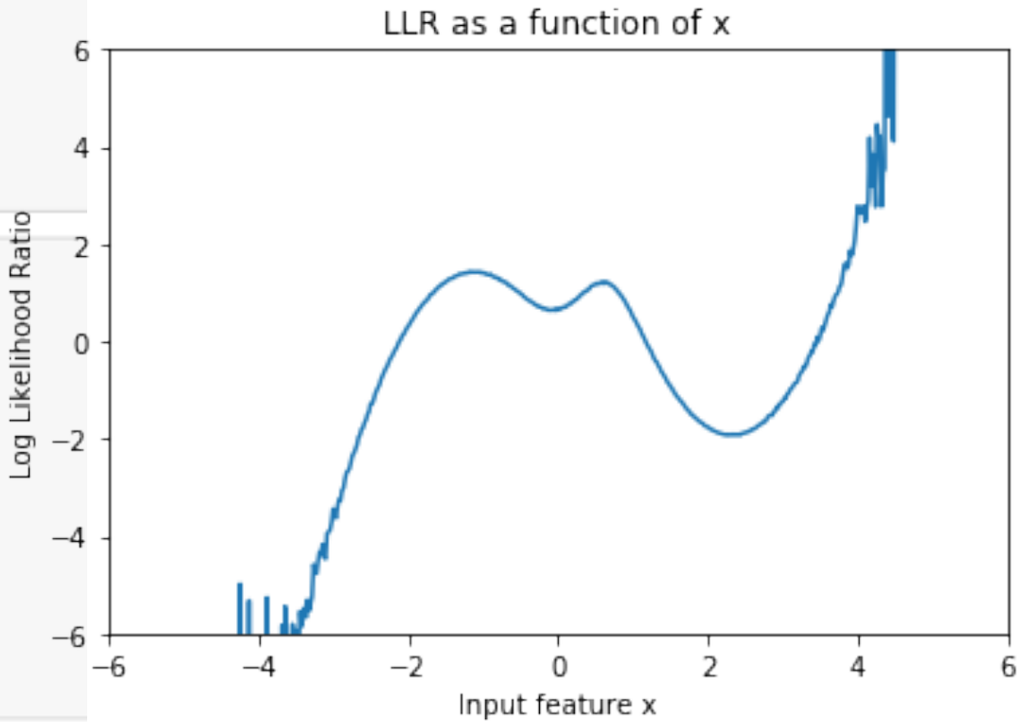


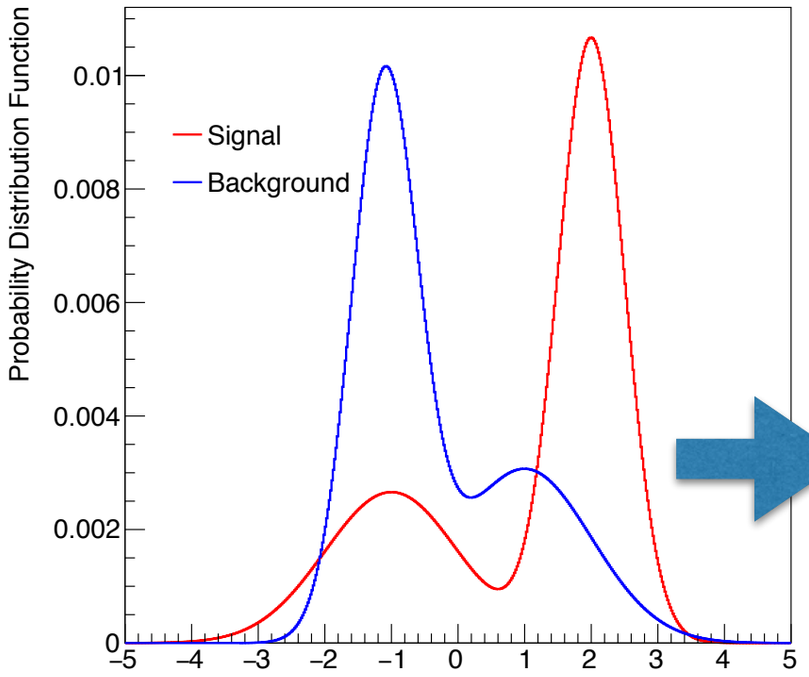


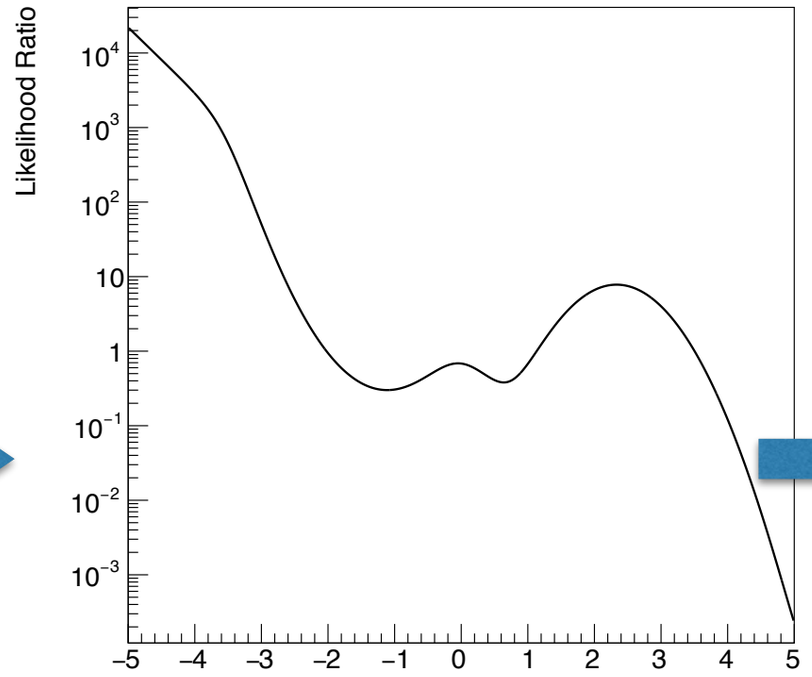


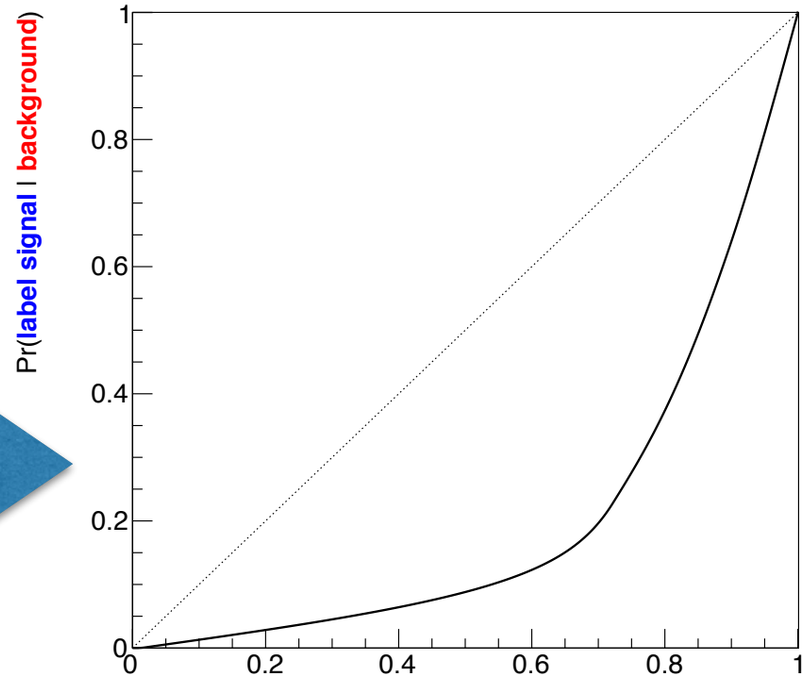
In [4]:
```
h_signal = np.histogram(signal, bins=500, range=(-5,5))
h_background = np.histogram(background, bins=500, range=(-5,5))

LL_dict = {} # used only for plotting
for i in range(len(h_signal[0])):
    if (h_background[0][i] > 0 and h_signal[0][i] > 0):
        LL_dict[h_background[1][i]] = np.log(1.*h_signal[0][i]/h_background[0][i])
    elif (h_signal[0][i] > 0): # in case background bin = 0
        LL_dict[h_background[1][i]] = np.log(100000.) #huge number
    elif (h_background[0][i] > 0): # in case signal bin = 0
        LL_dict[h_background[1][i]] = np.log(1./100000.) #very small number
    else:
        LL_dict[h_background[1][i]] = np.log(1.)
```

In [5]:
```
xvals = np.array([d for d in LL_dict])
yvals = np.array([LL_dict[d] for d in LL_dict])

index_sorted = xvals.argsort()
xvals = xvals[index_sorted[::-1]]
yvals = yvals[index_sorted[::-1]]

plt.plot(xvals,yvals)
plt.xlabel('Input feature x')
plt.ylabel('Log Likelihood Ratio')
plt.title(r'LLR as a function of x')
plt.axis([-6,6,-6,6])
plt.show()
```

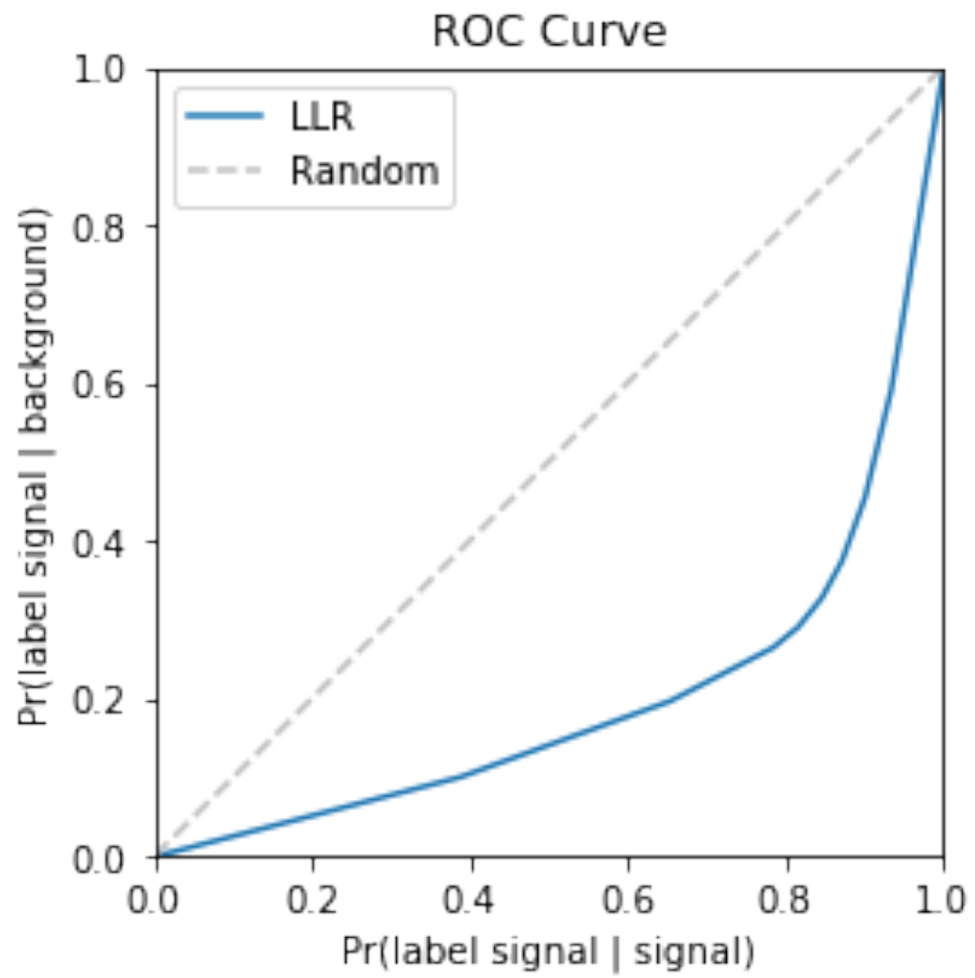









```
In [8]: nbins = 50
h_signal_yvals = np.histogram([], bins=nbins, range=(-10,10))
h_background_yvals = np.histogram([], bins=nbins, range=(-10,10))

for i in range(len(h_signal[0])):
    whichbin = np.digitize(LL_dict[h_signal[1][i]], h_signal_yvals[1])
    if (whichbin > 49):
        whichbin = 49
    h_signal_yvals[0][whichbin]+=h_signal[0][i]
    h_background_yvals[0][whichbin]+=h_background[0][i]

# Make the ROC curve
ROCx = np.zeros(nbins)
ROCy = np.zeros(nbins)
intx = 0.
inty = 0.

for i in range(nbins):
    intx+=h_signal_yvals[0][i]
    inty+=h_background_yvals[0][i]

for i in range(nbins):
    sum_signal = 0.
    sum_background = 0.
    for j in range(i,len(h_signal_yvals[1])-1):
        sum_signal+=h_signal_yvals[0][j]
        sum_background+=h_background_yvals[0][j]

    ROCx[i] = sum_signal/intx
    ROCy[i] = sum_background/inty

# Plot the ROC curve
plt.axes().set_aspect('equal')
plt.plot(ROCx,ROCy,label="LLR")
plt.plot([0,1],[0,1],linestyle='--',color="#C0C0C0",label="Random")
plt.xlabel('Pr(label signal | signal)')
plt.ylabel('Pr(label signal | background)')
plt.title(r'ROC Curve')
plt.axis([0, 1, 0, 1])
plt.legend(loc='upper left')
```



Why don't we always just
compute the optimal classifier?

In the last slides, we had to estimate the likelihood ratio - this required binning the PDF

binning works very well in 1D, but becomes quickly intractable as the feature vector dimension >> 1 ("curse of dimensionality")

machine learning for classification is simply
the art of estimating the likelihood ratio with limited training examples

Tools for Classification

= tools for likelihood ratio estimation

- “Histograming”
- Nearest Neighbors
- Support Vector Machines (SVM)
- (Boosted) Decision Trees
- (Deep) Neural Networks
- …

Not widely used; only useful if decision boundary is ‘simple’

has most things and ROOT-compatible but the community base is **much** smaller than the other ones

Software: TMVA, scikit-learn, keras, …

does “everything” exempt DNNs

python interface to DNN tools TensorFlow and Theano. Popular alternative is PyTorch

Data formats: .root, .npy, .hdf5

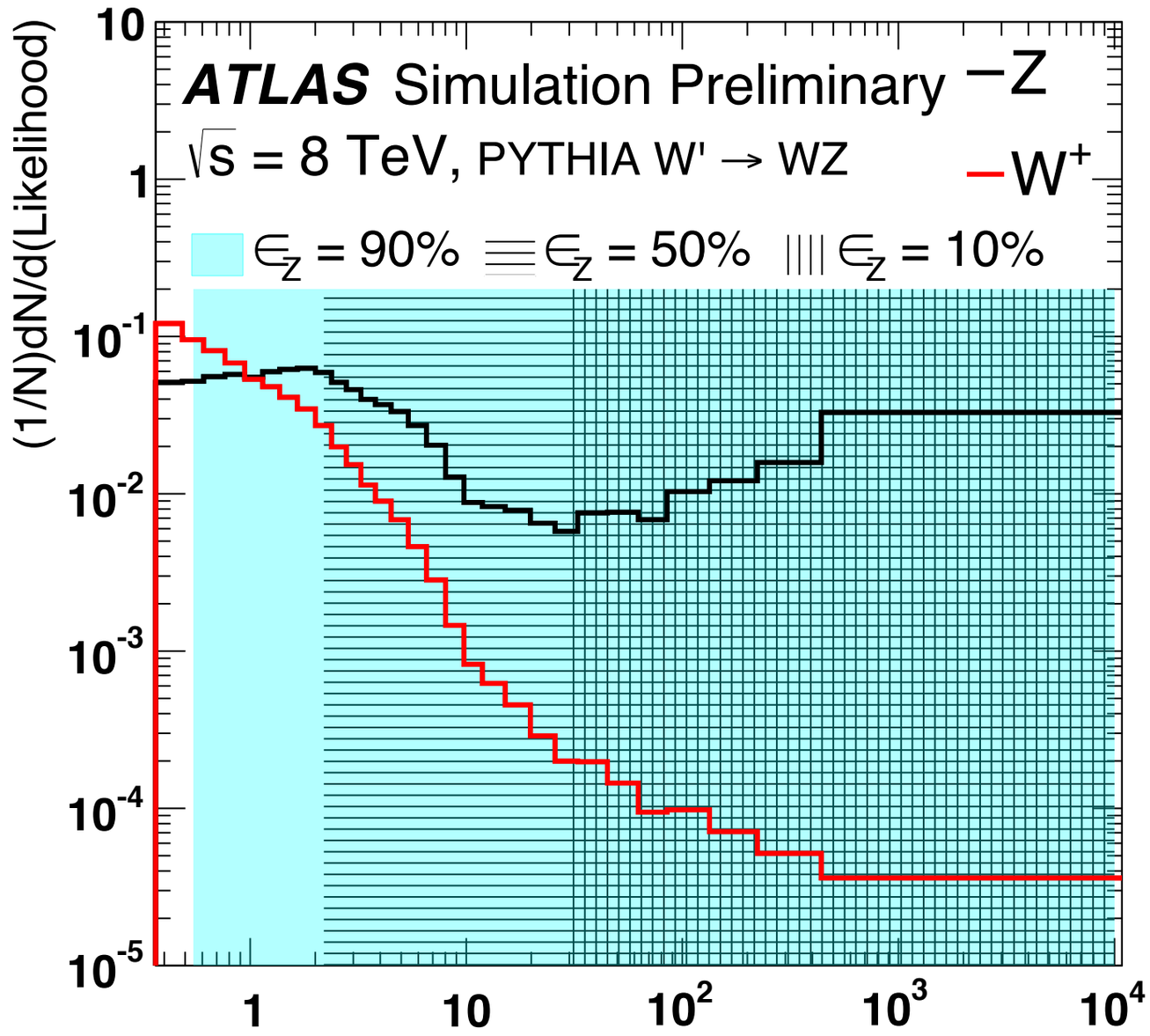
Histograming

If you have a 1D problem, look no further!

If your problem can be decomposed into a product/sum of 1D problems…look no further!

If these do not apply… look elsewhere.

Eur. Phys. J. C76 (2016) 238

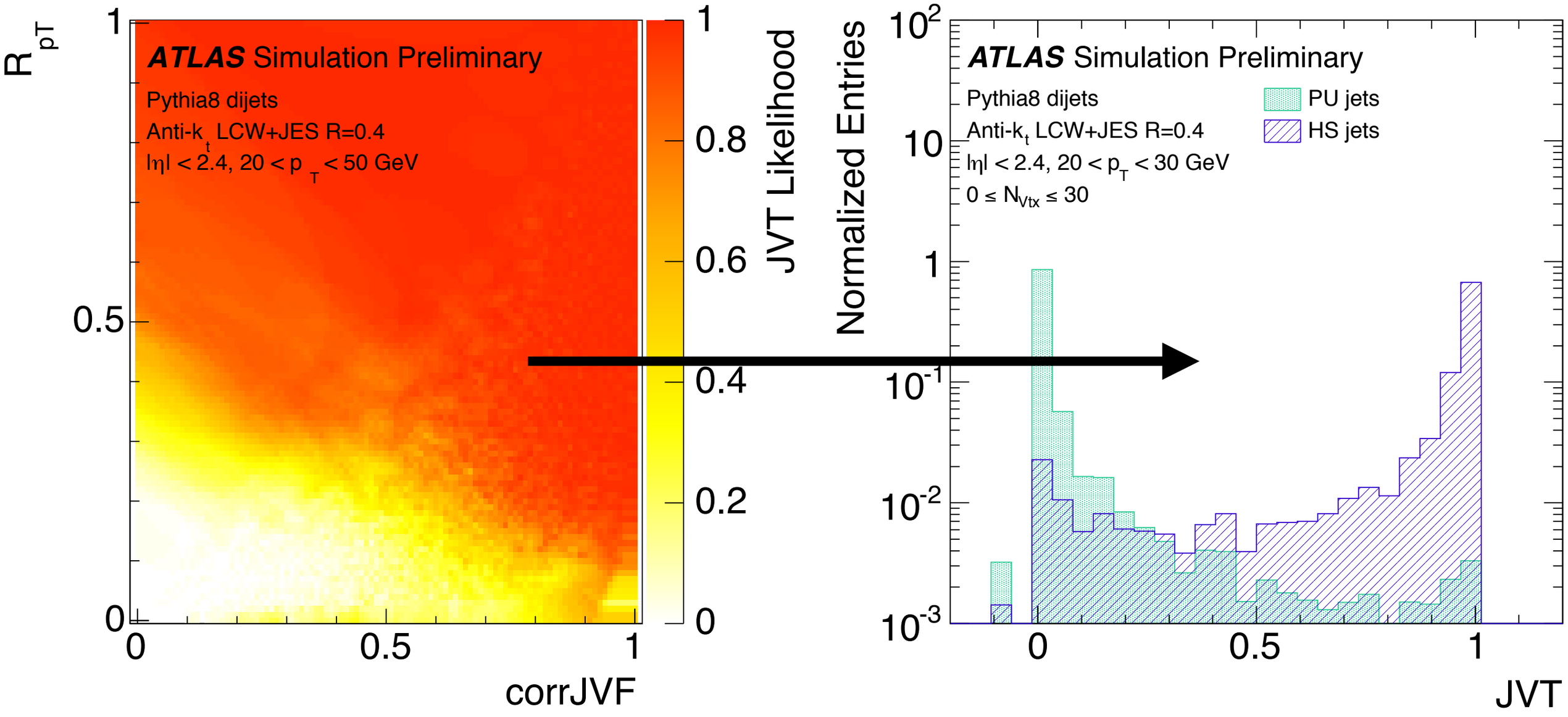


$$p(M, Q, B|V) = \sum_{\mathcal{F}} \Pr(\mathcal{F}|V)p(M|\mathcal{F}, V)p(Q|\mathcal{F}, V)\Pr(B|\mathcal{F}, V),$$

Nearest Neighbors

In 2D, a nice extension of histogramming is to estimate the likelihood ratio based on the number of S and B points nearby.

ATLAS-CONF-2014-018



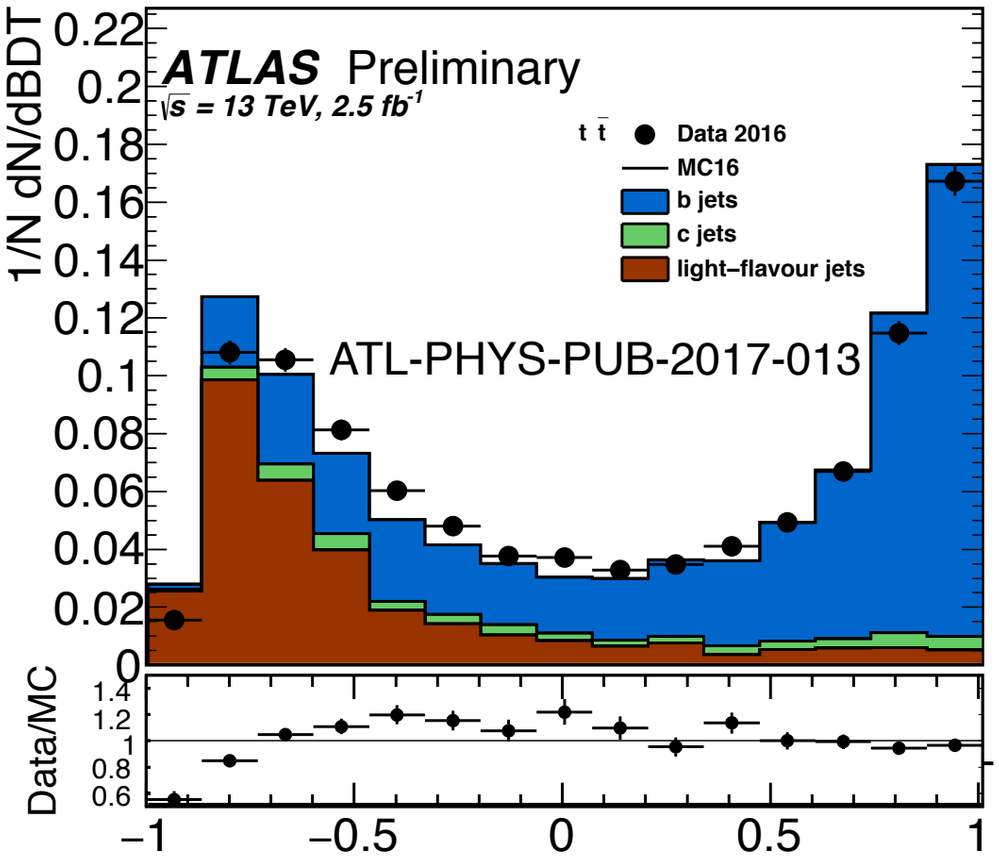
Boosted Decision Trees (BDTs)

A decision tree is a partition of the feature space.
One tree is a set of binary “cuts”.

Boosting makes an ensemble classifier. For example, a community favorite AdaBoost, applies weights to the misclassified events.

N.B. BDTs are not differentiable

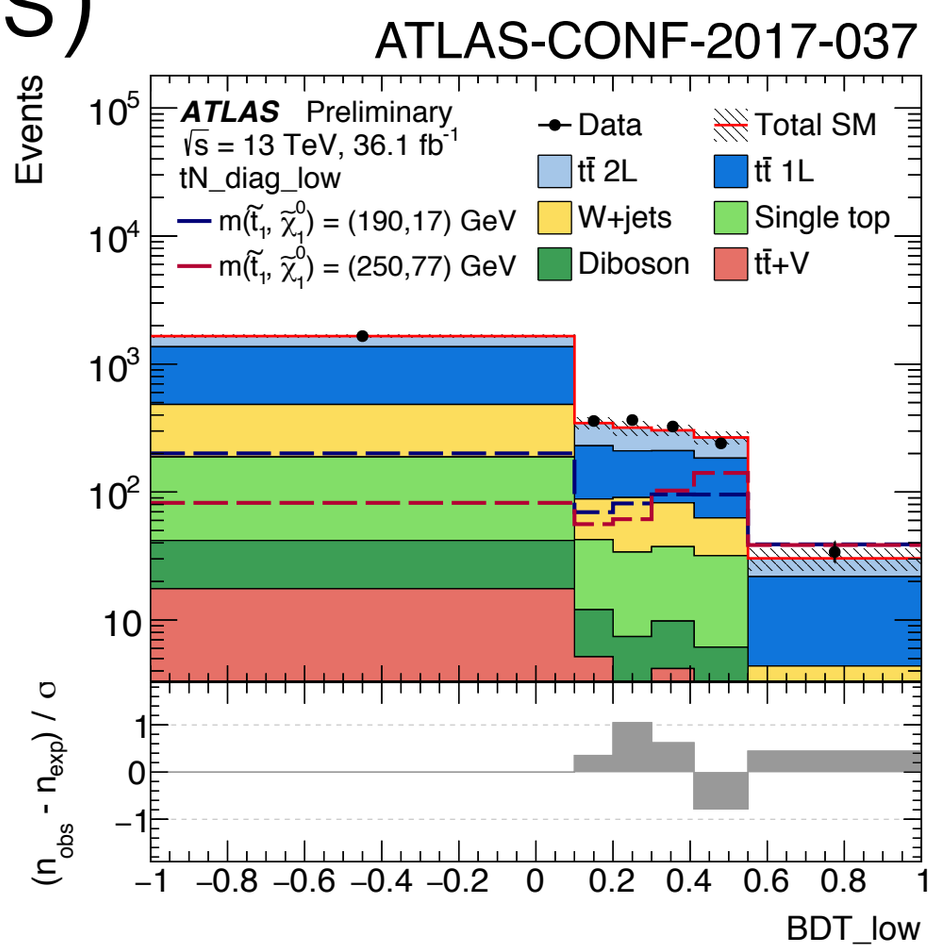
Boosted Decision Trees (BDTs)



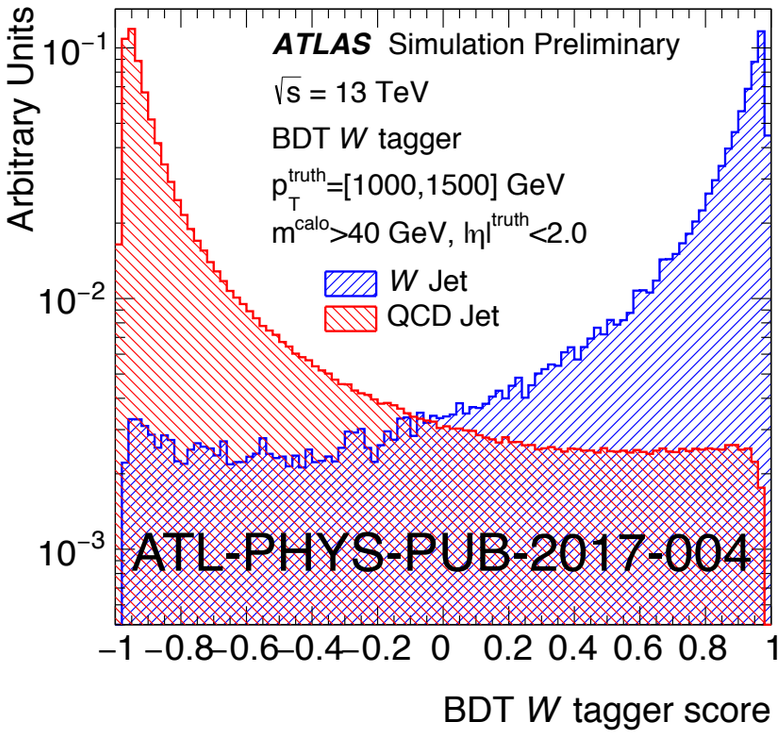
We love BDTs.

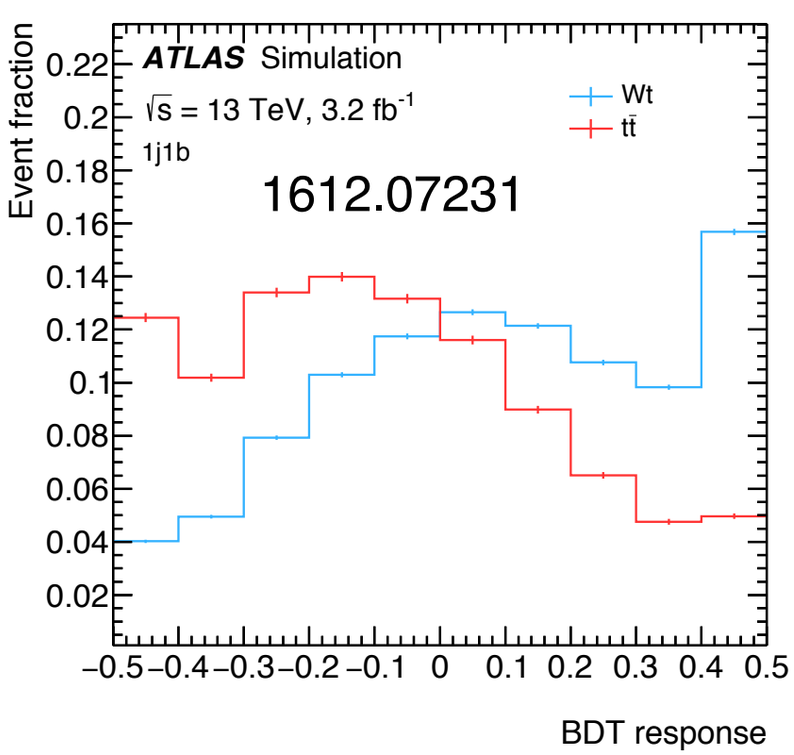
If 3 < dim(feature vector) < O(10)

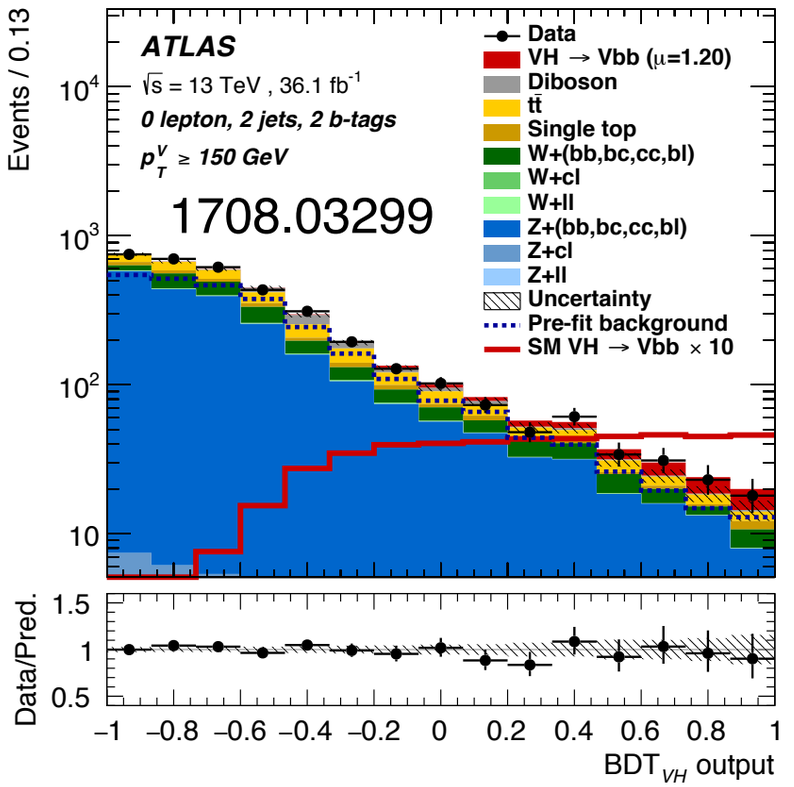
this is probably right for you!



SMT BDT Discriminant

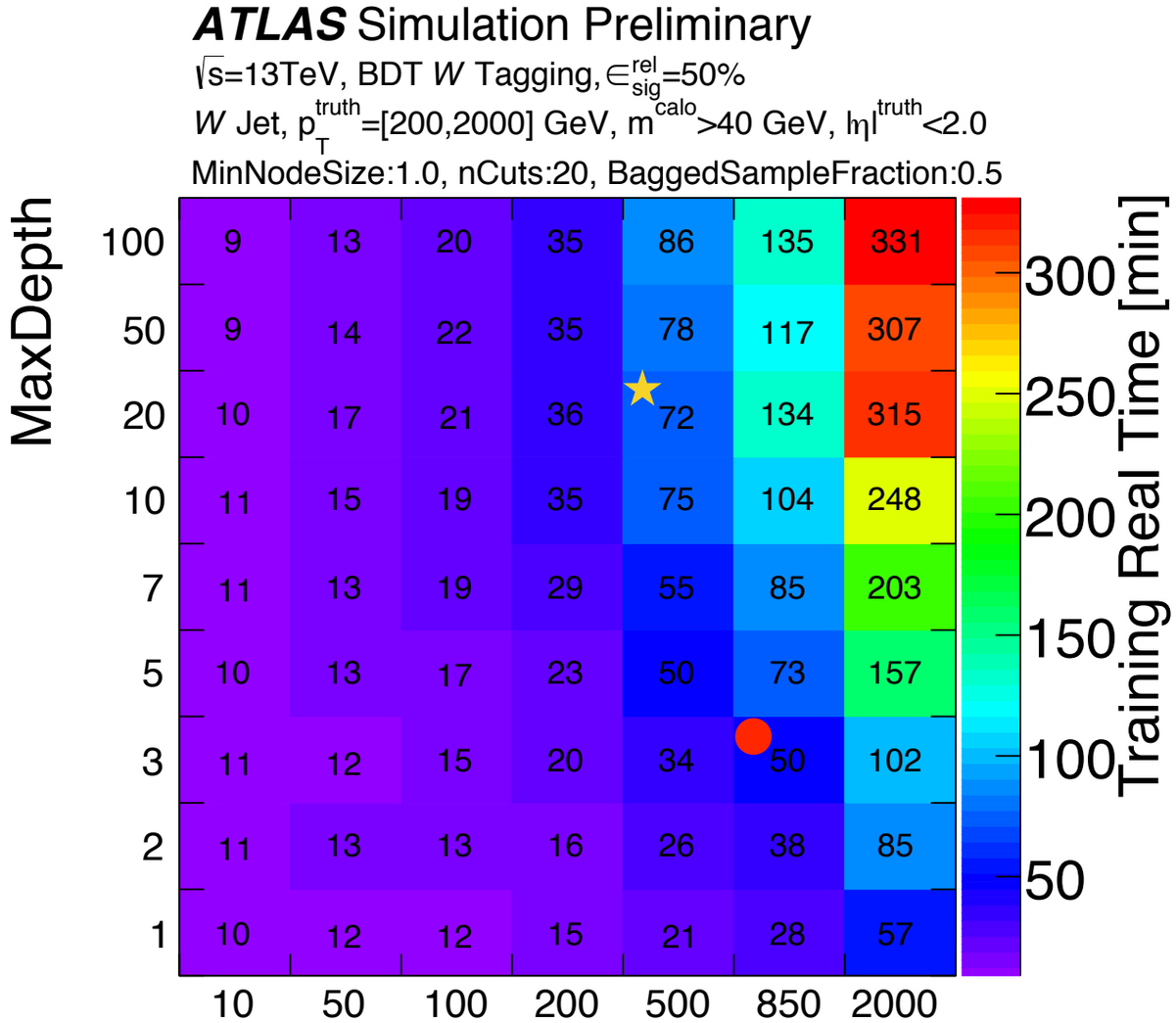


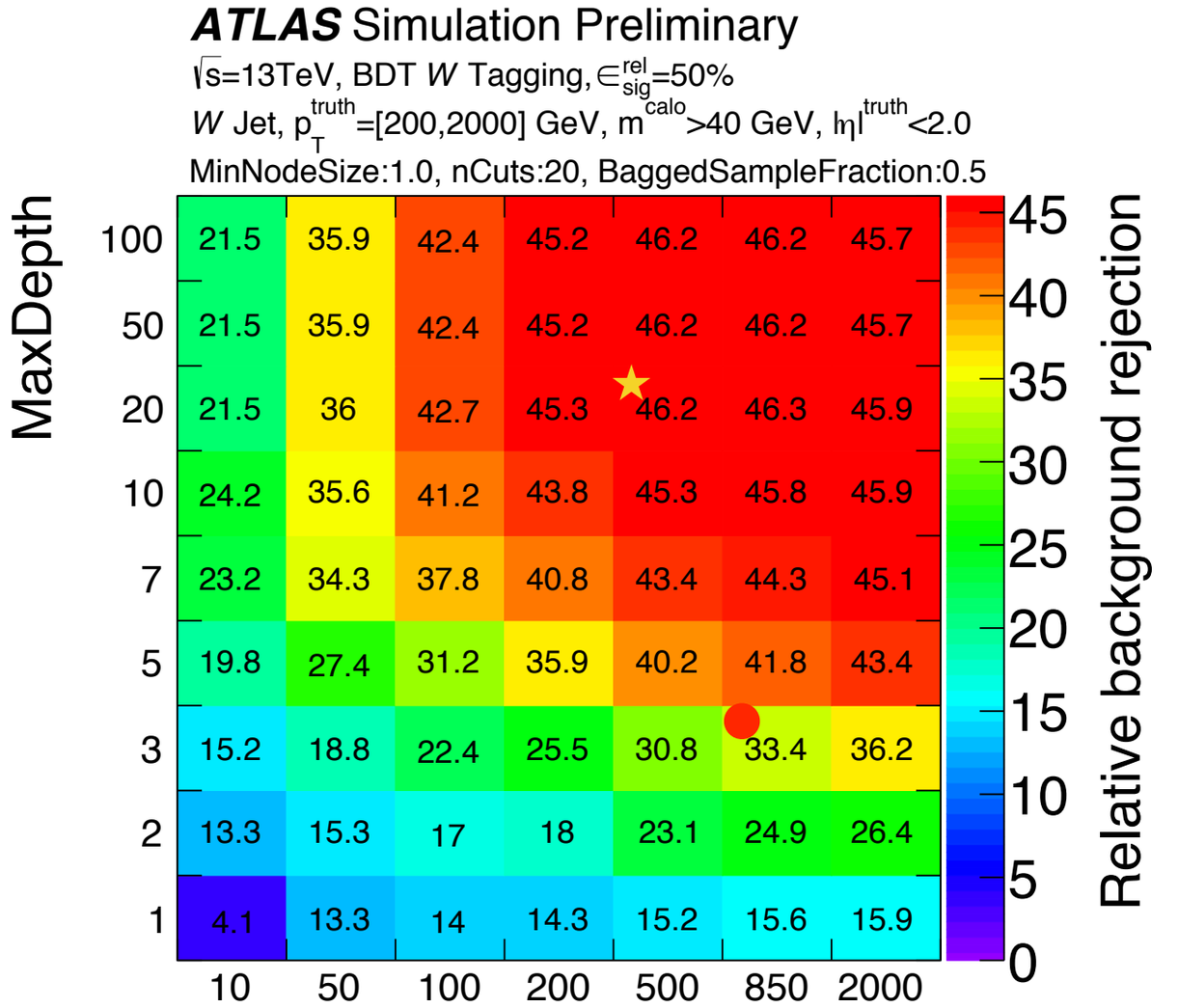




Boosted Decision Trees (BDTs)

We love BDTs because they are fast to train, are close to “cuts”, and do not have very many parameters. They are also rather robust to *overtraining*.

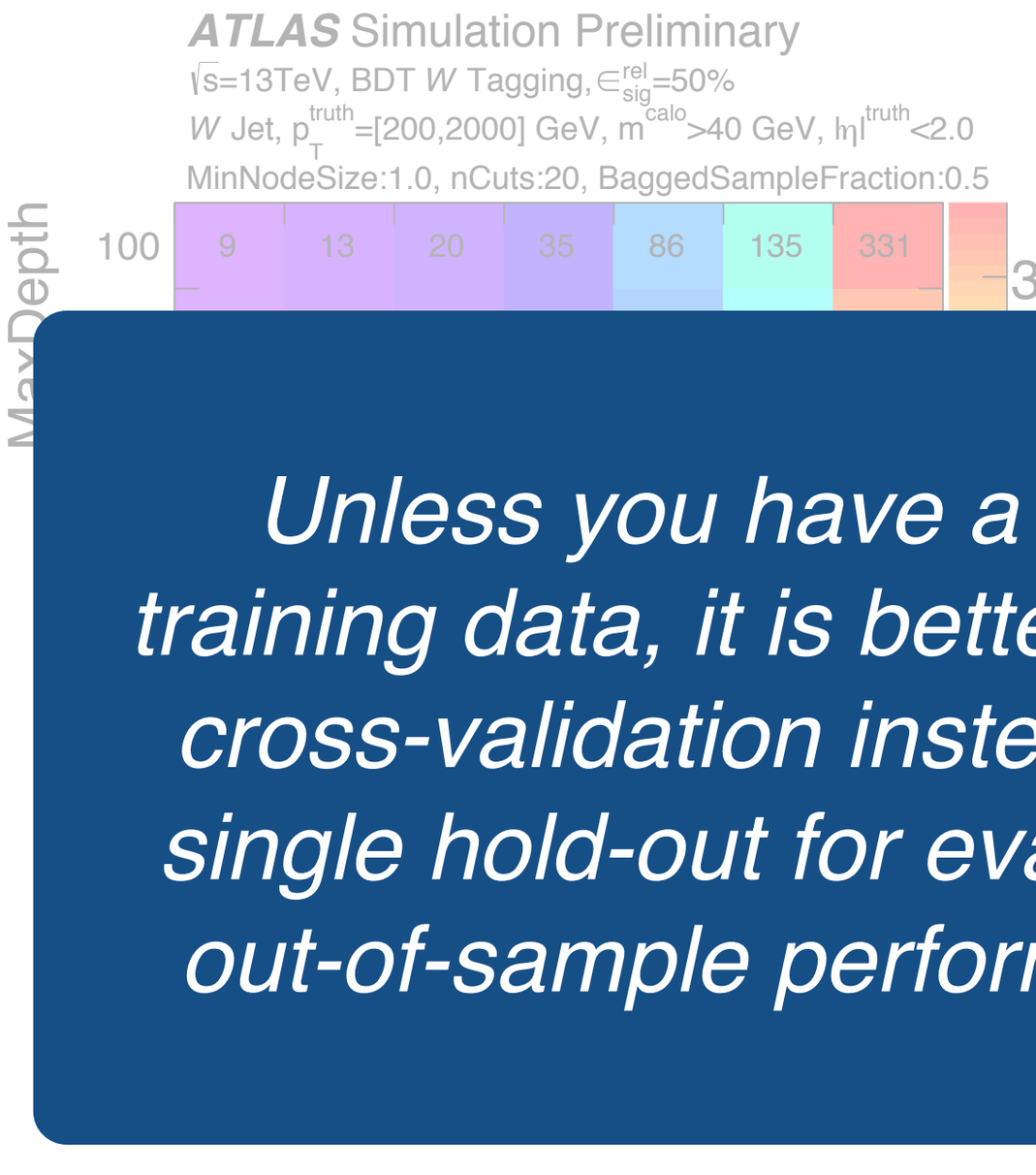


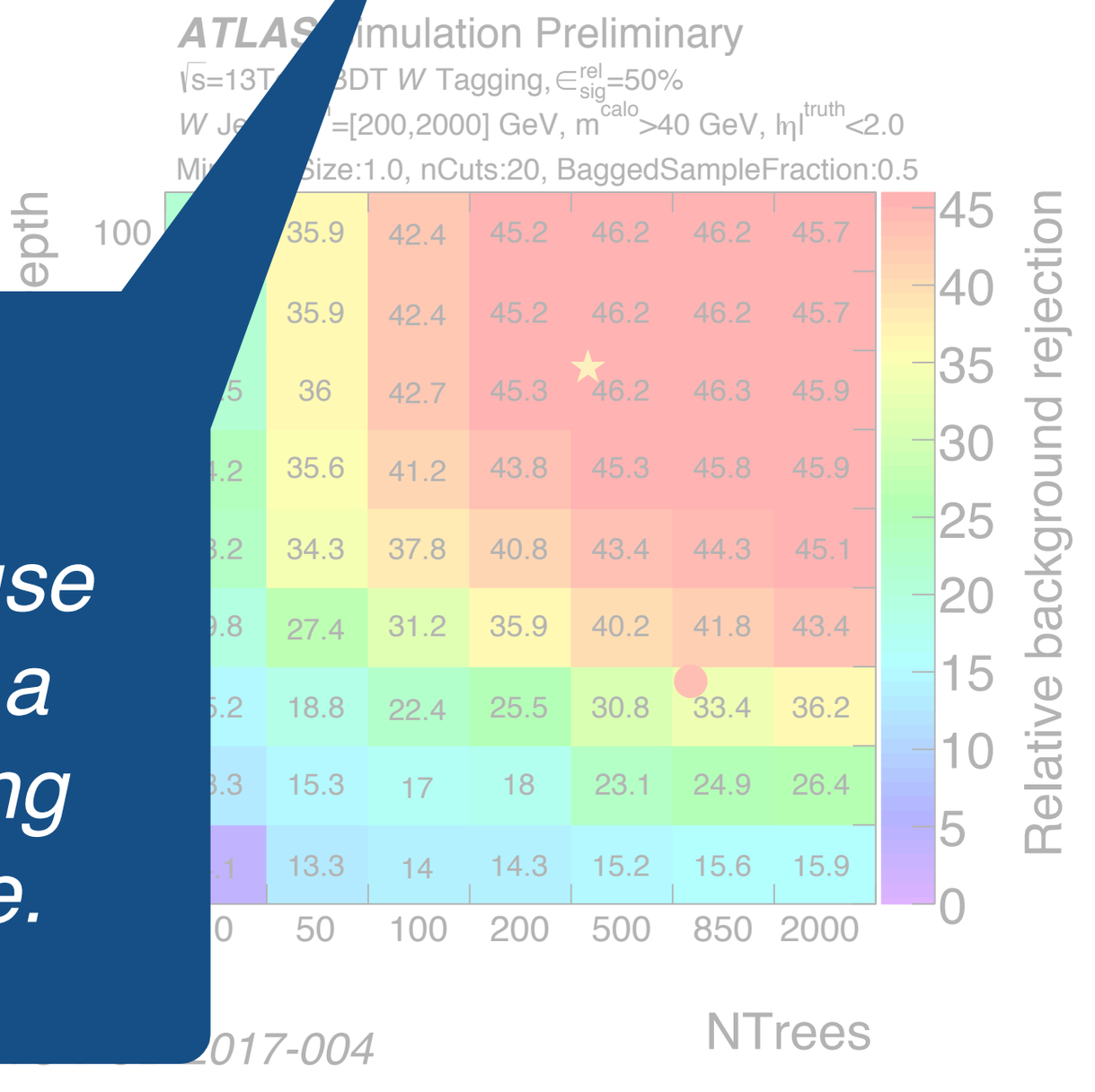


ATL-PHYS-PUB-2017-004

Boosted Decision Trees (BDTs)

We love BDTs because they are fast to train, are close to “cuts”, and do not have very many parameters. They are also rather robust to ***overtraining***.

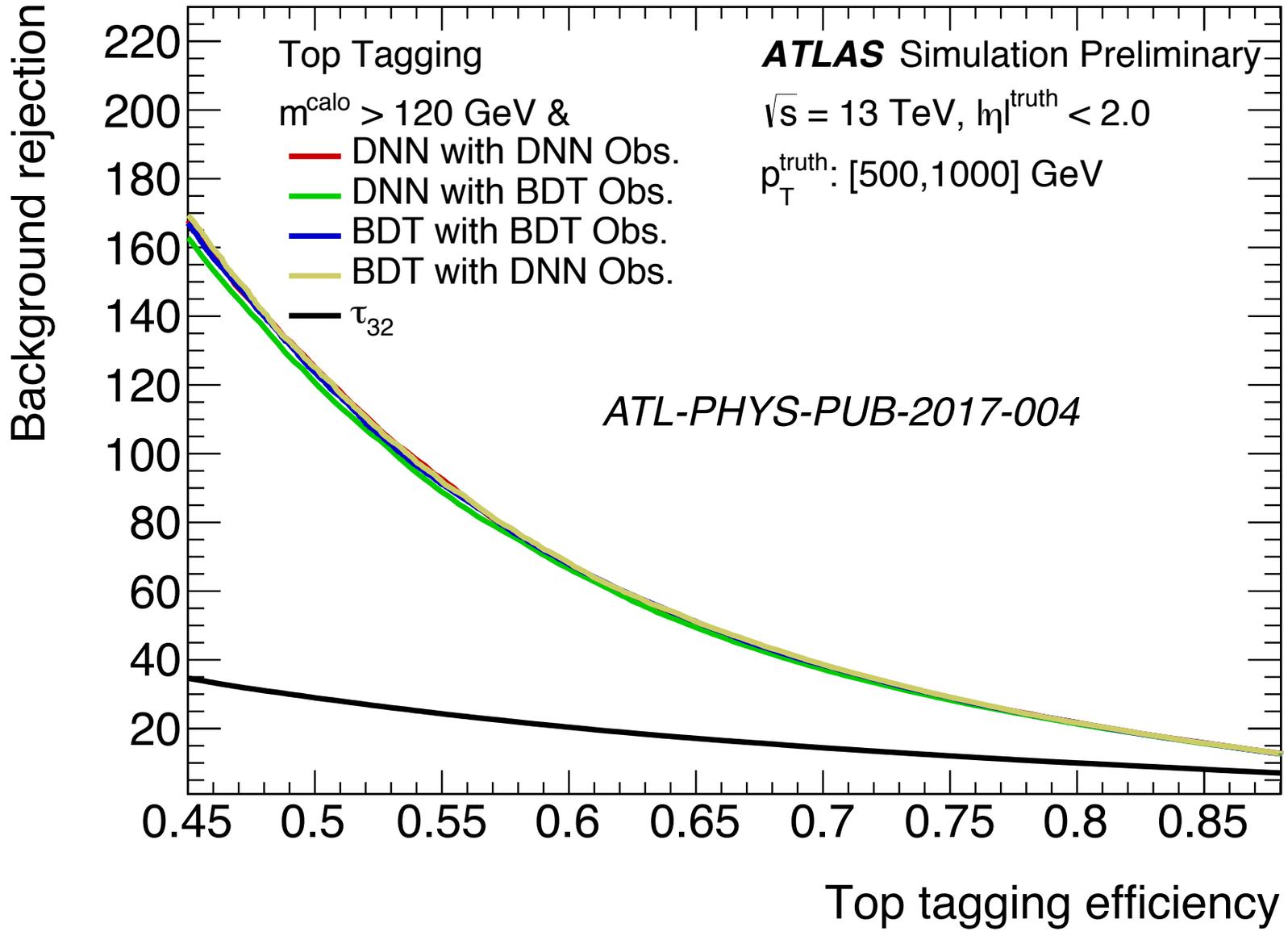




Unless you have a lot of training data, it is better to use cross-validation instead of a single hold-out for evaluating out-of-sample performance.

Boosted Decision Trees (BDTs)

There is really not a good reason to use a DNN with << O(100) dimensions.

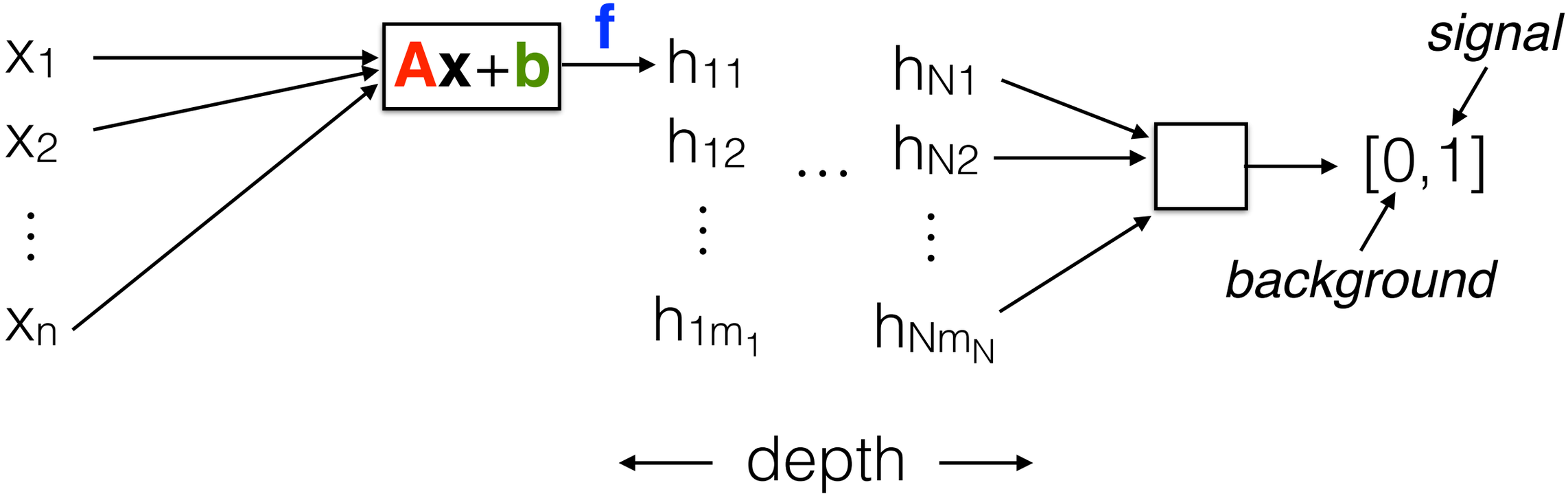


However, they are becoming increasingly easy to train …

Modern Deep NN's for Classification

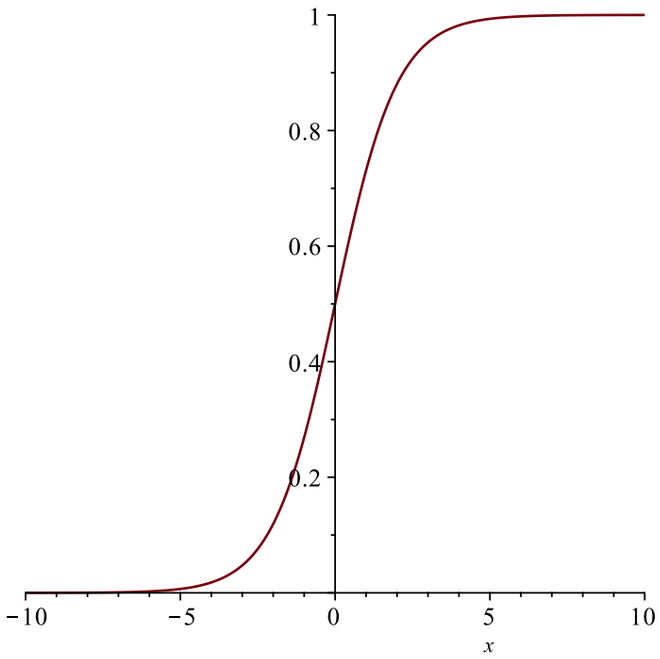
Neural Network: composition of functions **f**(**A**x+**b**) for inputs **x** (features) matrix **A** (weights), bias **b**, non-linearity **f**.

N.B. I'm not mentioning biology - there may be a vague resemblance to parts of the brain, but that is not what modern NN's are about.



Fact: NN's can approximate "any" function.

Choosing the non-linearity (activation function) **f**



Logistic (aka Sigmoid): one of the most widely-used functions in the past, now basically only used for the last layer.

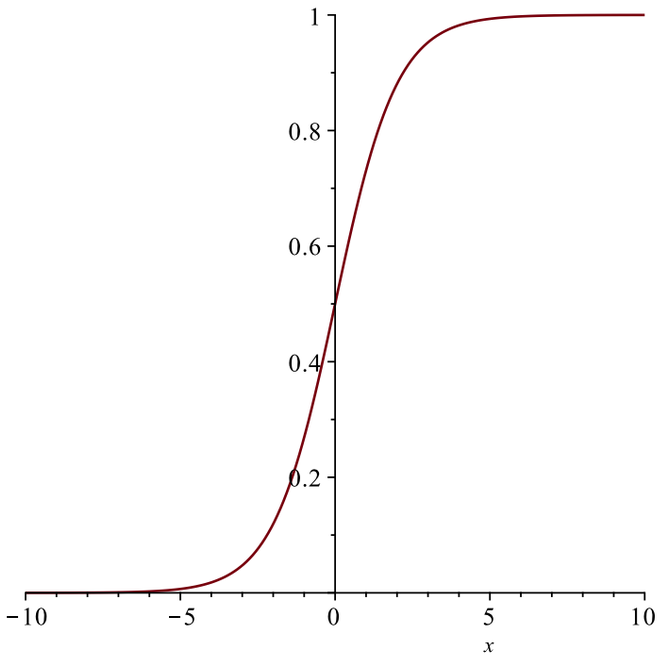
generalization to multi-dimensional input: softmax

$$\mathbf{f}(\vec{x}) = e^{x_i} / \sum_i e^{x_i}$$

N.B. without any hidden layers, this is logistic regression.

…and without any non-linearity, a NN is linear regression.

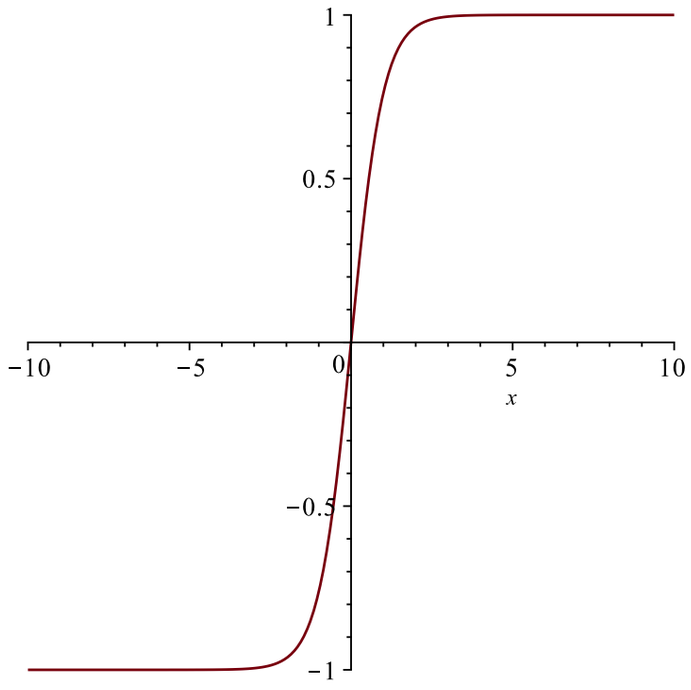
Choosing the non-linearity (activation function) **f**



Logistic (aka Sigmoid): one of the most widely-used functions in the past, now basically only used for the last layer.

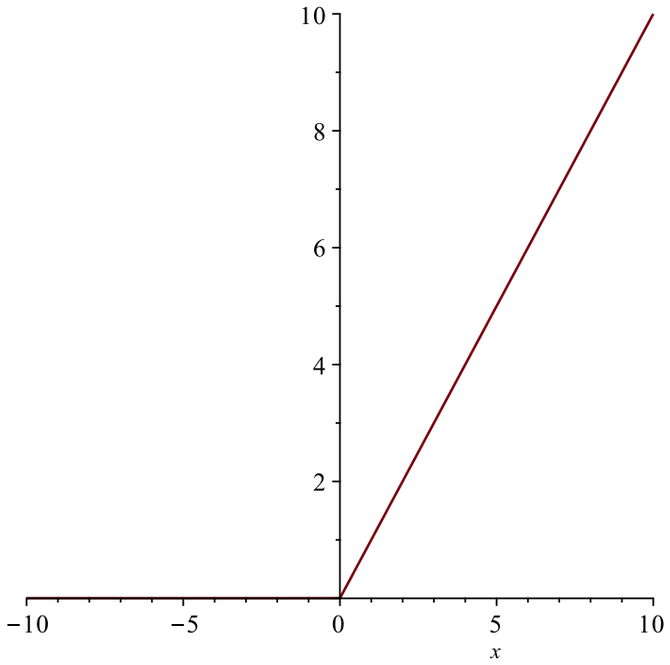
generalization to multi-dimensional input: softmax

$$\mathbf{f}(\vec{x}) = e^{x_i} / \sum_i e^{x_i}$$



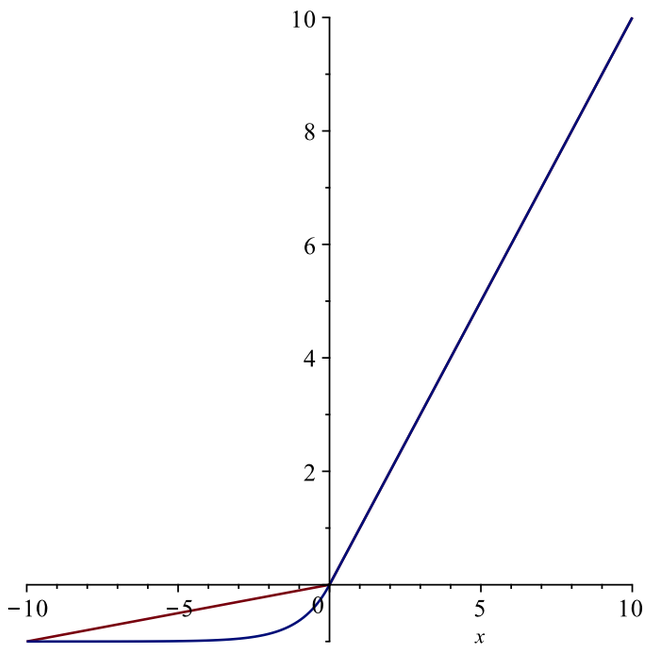
tanh: similar story to sigmoid.

Choosing the non-linearity (activation function) **f**



Rectified Linear Unit **ReLU**: one of the most widely-used functions now.

do not suffer from the vanishing gradient problem



Leaky ReLU / Exponential LU (ELU): variations on the ReLU that are popular.

Functions that act on multiple nodes in one layer

MaxOut: Take the maximum of multiple inputs

reduces the dimensionality of a hidden layer

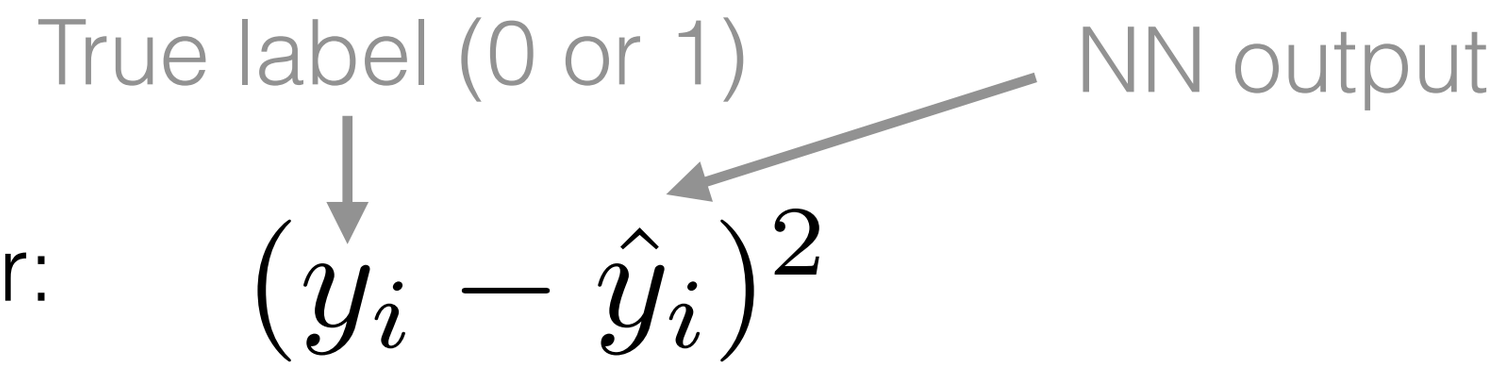
DropOut: Randomly remove (for one forward/backward pass) nodes from a layer.

helps with over-training

(D)NN Training

Training proceeds by minimizing a loss function.

Typical loss functions



Squared error: $(y_i - \hat{y}_i)^2$

Cross-entropy: $-y_i \log(\hat{y}_i) - (1 - y_i) \log(1 - \hat{y}_i)$

Categorical cross-entropy (multi-class):

$$-\sum_{i=1}^{n_{\text{categories}}} \hat{y}_i \log(\hat{y}_i(x))$$

Label & output are vectors

Reduces to cross-entropy when *n = 2* and soft max activation on last layer

Does it matter which loss function I use?

The NN optimization problem:

$$f = \text{argmin}_{f'} \mathbb{E}[\text{loss}(f'(X), Y)]$$

= expected value
= mean = average

$$Y \in \{0, 1\}$$

class = signal/ background

Loss functions from previous slide:

$$\text{loss}(f(X), Y) = (f(X) - Y)^2 \quad \textit{squared error}$$

$$\text{loss}(f(X), Y) = -Y \log(f(X)) - (1 - Y) \log(1 - f(X))$$

cross-entropy

Does it matter which loss function I use?

$$f = \text{argmin}_{f'} \mathbb{E}[\text{loss}(f'(X), Y)]$$

$$= \text{argmin}_{f'} \mathbb{E}[\mathbb{E}[\text{loss}(f'(X), Y)|X]$$

By the mean value theorem, it is sufficient to minimize the inner expectation for all *X*. This is a function (and not functional) optimization problem.

Let's work this out for the squared error and cross-entropy

What does the machine learn with the squared error?

$$
\begin{aligned}
g(x) &= \text{argmin}_{f'} \mathbb{E}[\text{loss}(f'(x), Y)|X = x] \\
&= \text{argmin}_{f'} \mathbb{E}[(f'(x) - Y)^2|X = x] \\
&= \text{argmin}_{f'} \mathbb{E}[(f'(x))^2 + Y^2 - 2f'(x)Y|X = x] \\
&= \text{argmin}_{f'} \Big((f'(x))^2 + \mathbb{E}[Y^2|X = x] - 2f'(x)\mathbb{E}[Y|X = x]\Big) \\
&= \text{argmin}_{f'} \Big((f'(x))^2 - 2f'(x)\mathbb{E}[Y|X = x]\Big) \\
&= \text{argmin}_{z} \Big(z^2 - 2z\mathbb{E}[Y|X = x]\Big)
\end{aligned}
$$

z = f'(x) is just a number

If you take the derivative w.r.t. *z*
and set it equal to zero, you find

$$z = \mathbb{E}[Y|X = x]$$

i.e. the NN **learns the average** value of the target given the features.

What does the machine learn with the cross-entropy?

$$
\begin{aligned}
g(x) &= -\text{argmin}_{f'}\mathbb{E}[Y\log(f'(x)) + (1-Y)\log(1-f'(x))|X=x] \\
&= -\text{argmin}_{f'}\left(\mathbb{E}[Y|X=x]\log(f'(x)) + (1-\mathbb{E}[Y|X=x])\log(1-f'(x))\right) \\
&= -\text{argmin}_{z}\left(\mathbb{E}[Y|X=x]\log(z) + (1-\mathbb{E}[Y|X=x])\log(1-z)\right)
\end{aligned}
$$

If you take the derivative w.r.t. z and set it equal to zero, you find

$$
\frac{\mathbb{E}[Y|X=x]}{z} - \frac{1-\mathbb{E}[Y|X=x]}{1-z} = 0 \implies z = \mathbb{E}[Y|X=x]
$$

…same answer as squared error!

Does it matter which loss function I use?

Some remarks:

- Squared error and cross-entropy both learn $\langle Y|X \rangle$
  - This is a coincidence. For fun, can you show what happens if you use the 4th power of the difference instead of the square?
  - The absolute error loss $|f(X)-Y|$ learns the median instead of the mean. The 0-1 loss learns the mode.
- For classification, Y is 0 or 1 so $\langle Y|X \rangle = Pr(Y=1|X)$
  - This is not the likelihood ratio! How is it related?

(D)NN training

Objective function is minimized using stochastic gradient descent (almost exclusively with the Adam algorithm)

Stochastic gradient decent: Using single (or multiple "mini-batches") examples, weights are updated:

$$A_{ij} \mapsto A_{ij} - \eta \nabla_{ij} \mathcal{L}$$

learning rate

back-propagation: weights updated backwards and gradients are recycled.

N.B. a NN can do better than random **before any training**!
For instance, if you initialize all the weights to 1 and the signal has generally higher values then the NN will beat random.

ng

ng stochastic gradient
he Adam algorithm)

g single (or multiple
ghts are updated:

This is one of the powerful features of NNs - you can iteratively apply the chain rule to compute the gradient of the loss with respect to any weights in the network.

$$A_{ij} \mapsto A_{ij} - \eta \nabla_{ij} \mathcal{L}$$

learning rate

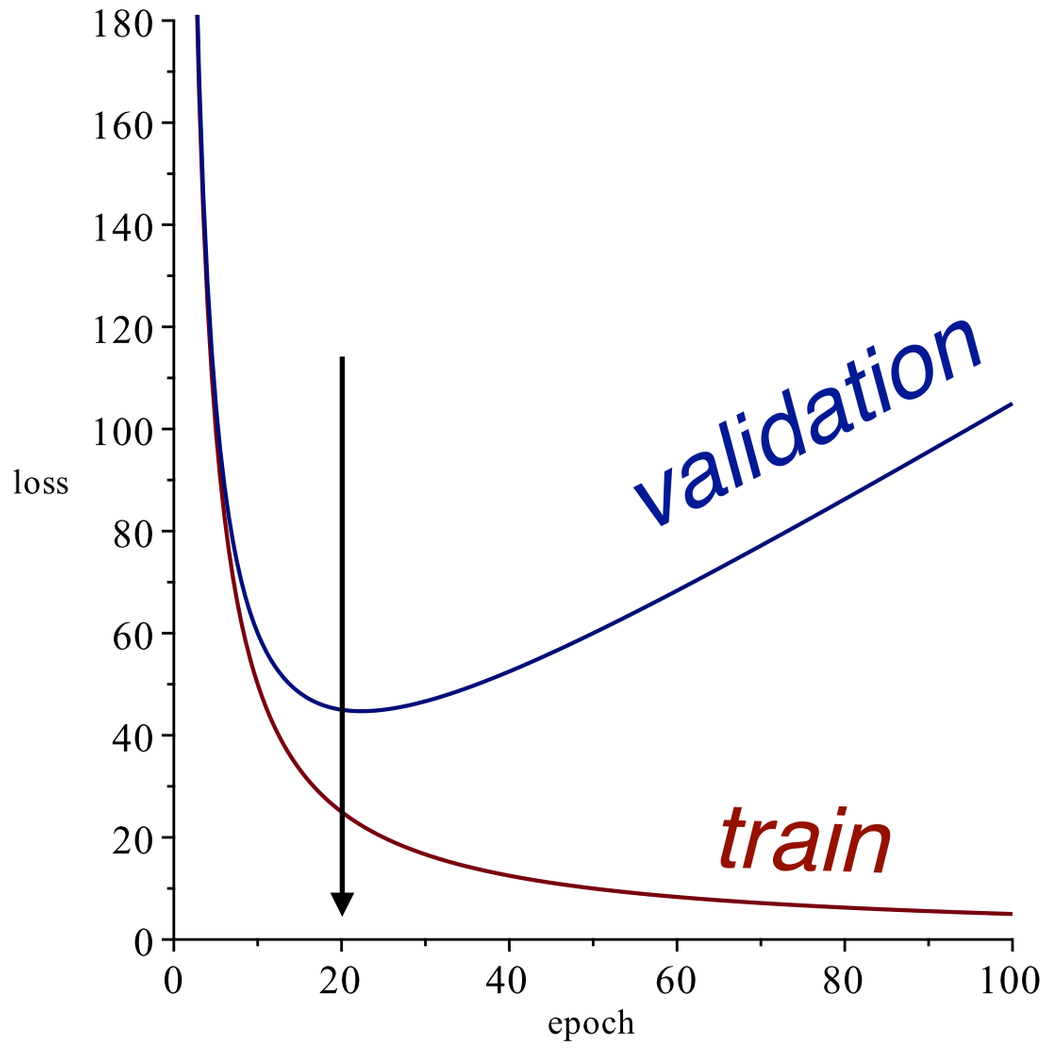
back-propagation: weights updated backwards and gradients are recycled.

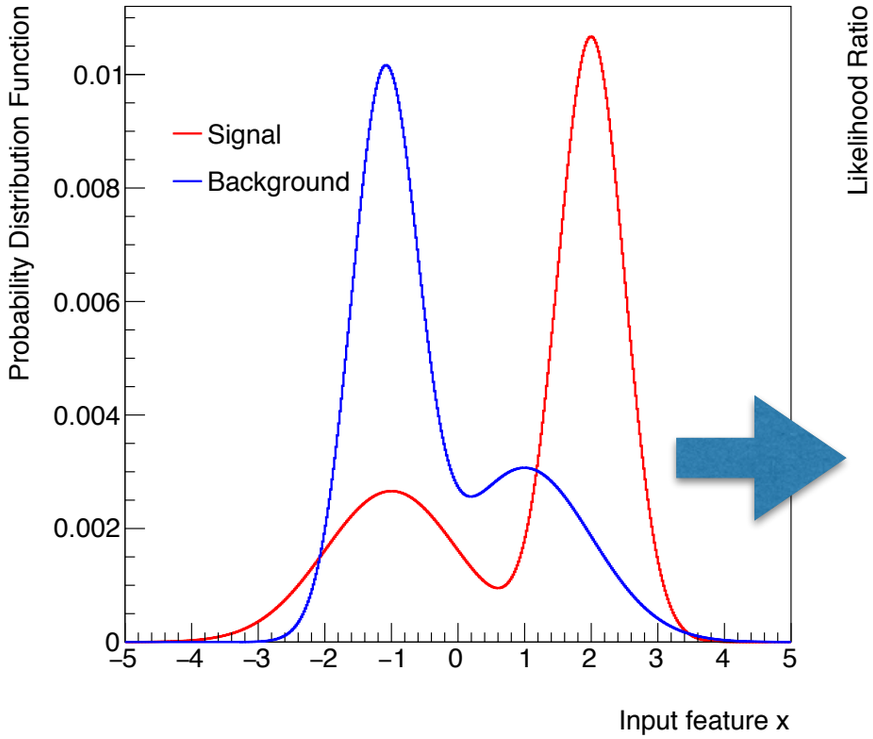
N.B. a NN can do better than random **before any training**!
For instance, if you initialize all the weights to 1 and the signal has generally higher values then the NN will beat random.

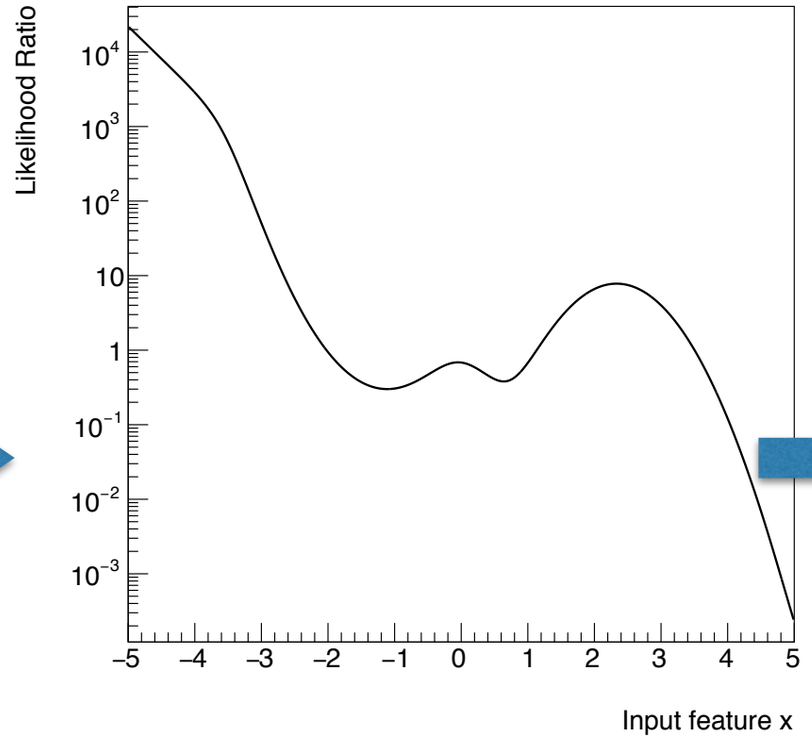
(D)NN training

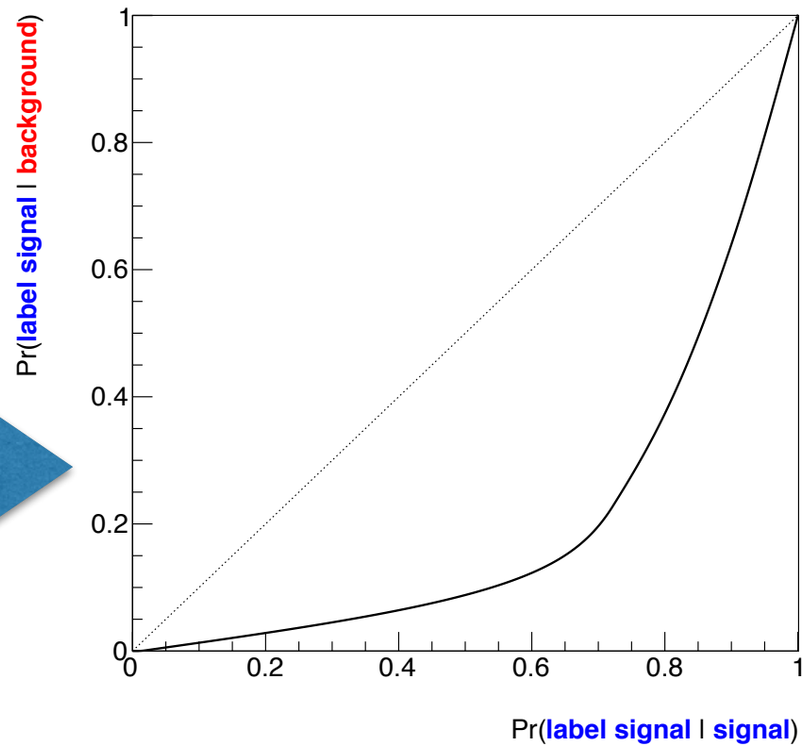
Training proceeds multiple times (epochs), reshuffling the data.

Early stopping: stop at the epoch where the validation error starts to increase

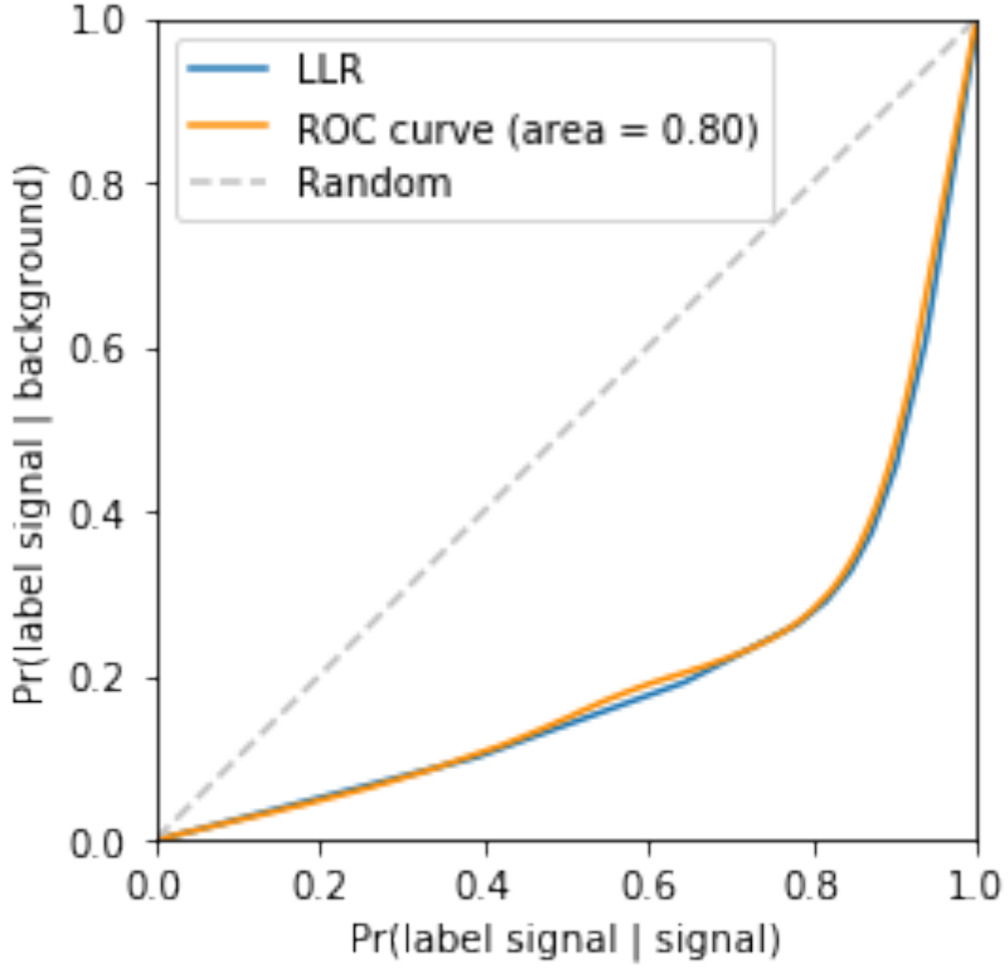









```
In [44]: from keras.models import Sequential
         from keras.layers import Dense, Dropout
         from sklearn.model_selection import train_test_split
         import random

In [45]: random.shuffle(signal)
         random.shuffle(background)

In [89]: model = Sequential()
         model.add(Dense(1, input_dim=1, activation='relu'))
         model.add(Dense(10, activation='relu'))
         model.add(Dense(1, activation='sigmoid'))
         model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
         model.summary()

         N_small = 1000000
         X = np.concatenate([signal[0:N_small],background[0:N_small]])
         Y =  np.concatenate([np.ones(N_small),np.zeros(N_small)])
         X_train, X_test, Y_train, Y_test, = train_test_split(X, Y, test_size=0.5)
         model.fit(X_train,Y_train, epochs=3,batch_size=2000)

In [91]: from sklearn.metrics import roc_curve, auc
         fpr, tpr, _ = roc_curve(Y_test, model.predict(X_test))
         roc_auc = auc(fpr, tpr)
         plt.axes().set_aspect('equal')
         plt.plot(ROCx,ROCy,label="LLR")
         plt.plot(tpr,fpr,color='darkorange',label='ROC curve (area = %0.2f)' % roc_auc)
         plt.plot([0,1],[0,1],linestyle='--',color="#C0C0C0",label="Random")
         plt.xlabel('Pr(label signal | signal)')
         plt.ylabel('Pr(label signal | background)')
         plt.axis([0, 1, 0, 1])
         plt.legend(loc='upper left')
         plt.show()
```

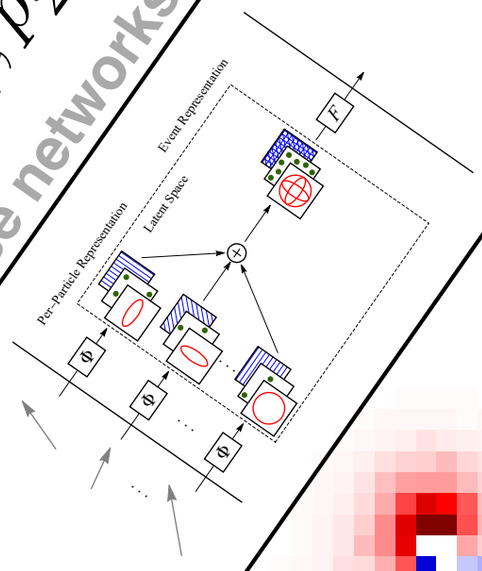


Beyond fully connected networks

Fixed sets

$J = \{p_1^\mu, p_2^\mu, \ldots, p_n^\mu\}$

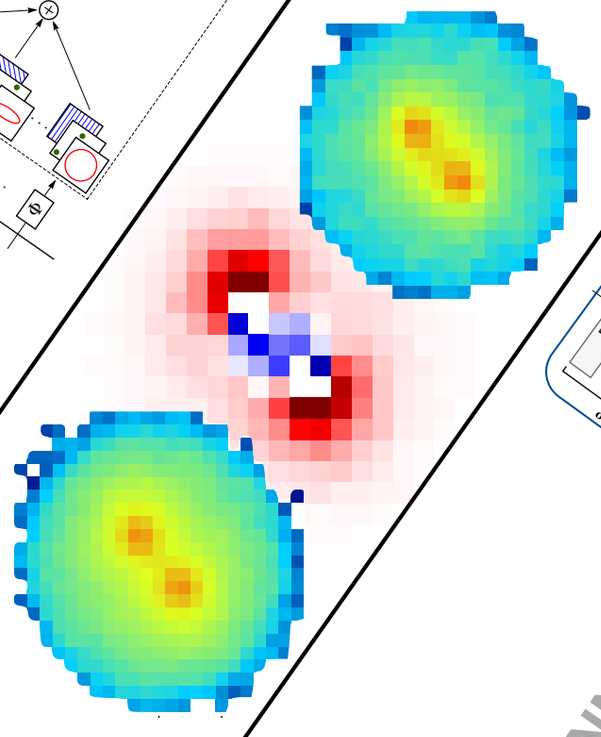
Dense networks



Variable sets

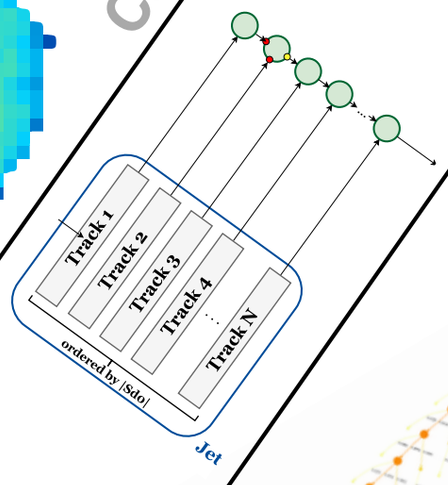
Deep sets

Images



Convolutional NNs (CNNs)

Sequences

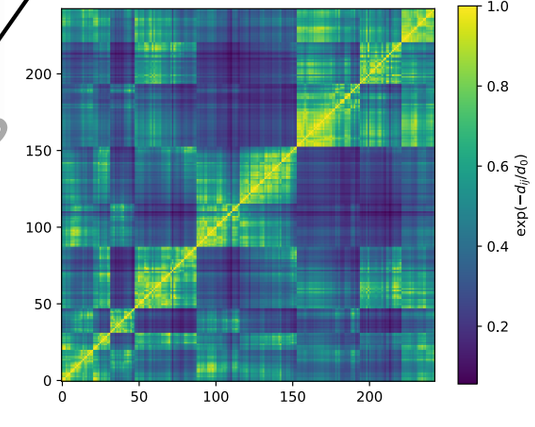


Recurrent NNs

Trees

Recursive NNs

Graphs

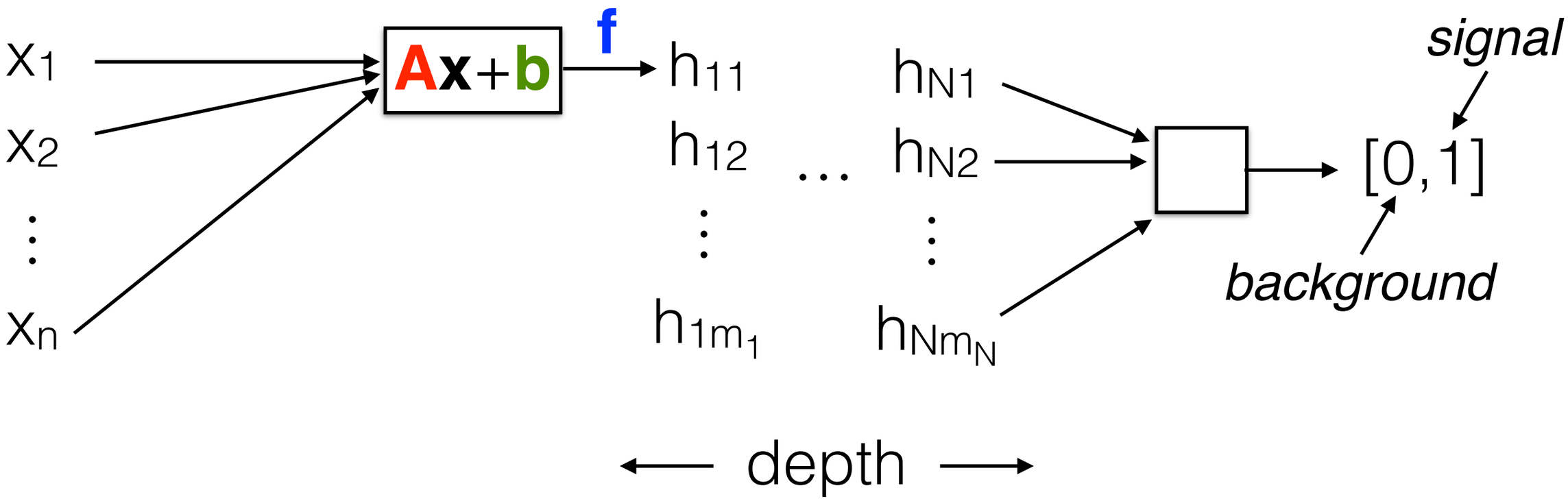


Graph CNNs

Beyond fully connected networks

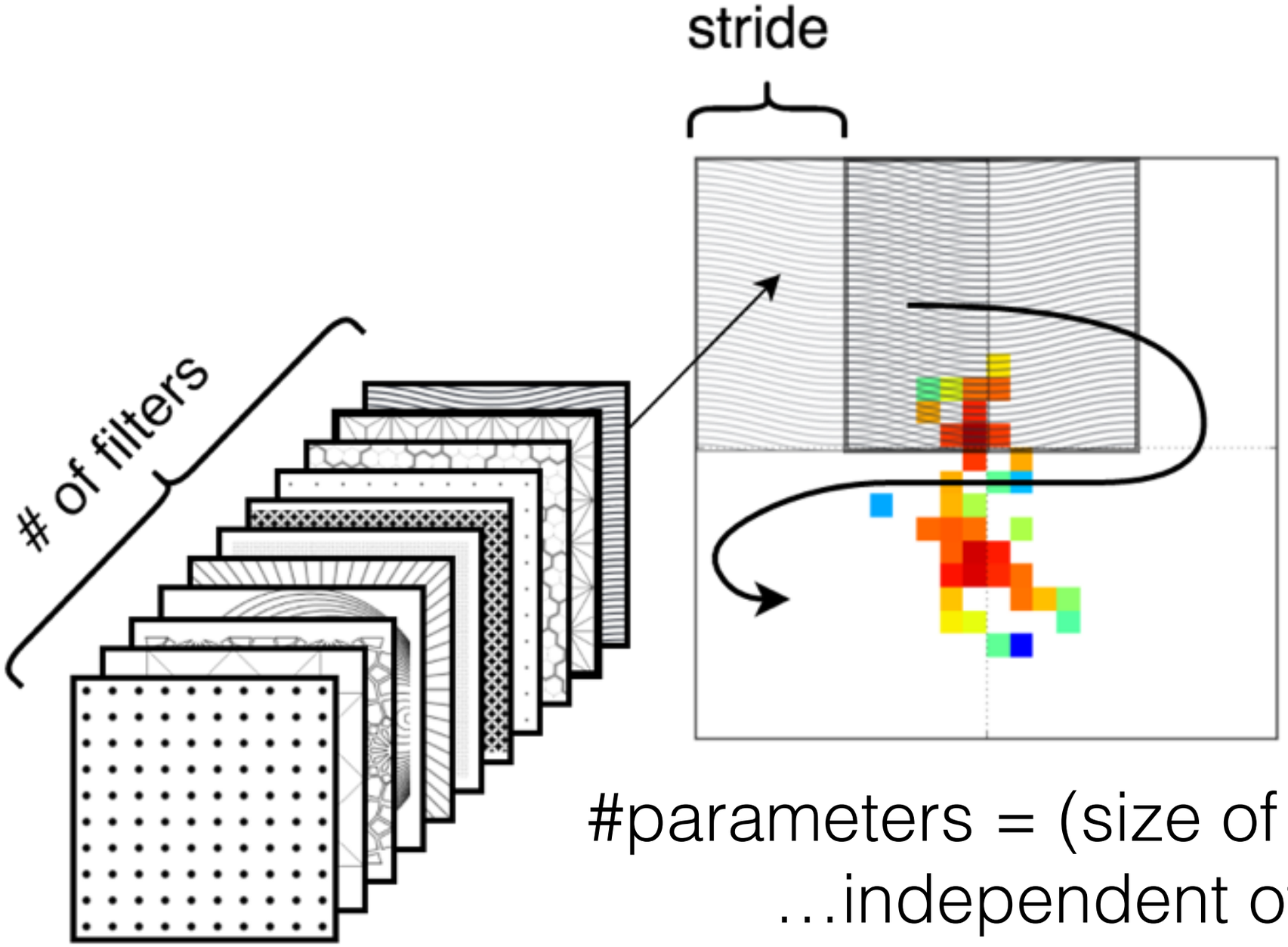
A fully connected ("dense" in keras) has **many parameters** and does **not know about the structure** of the data.

$$\#\text{ of parameters} \sim (n+1) \times m_1 + (m_1+1) \times m_2 + \ldots + (m_{N-1}+1) \times m_N$$



Convolutional neural networks

The community standard for **image data** is CNNs. Fixed-size filters are convolved with the images to produce additional images. This is an **automated feature extraction** stage.



Benefit: independent of where the features are in the image (translational invariance)

#parameters = (size of filter) x (# of filters)
…independent of image size!

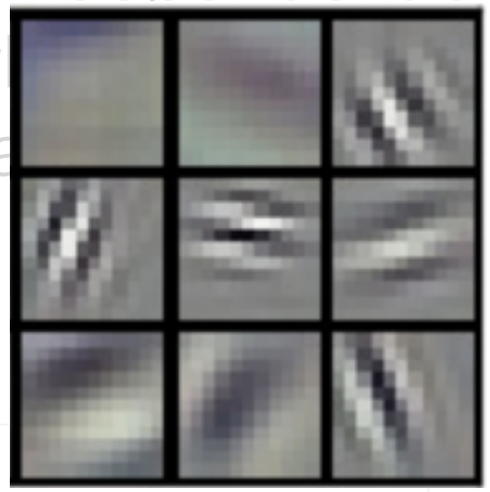
Convolutional neural networks

The community standard for **image data** is CNNs. Fixed-size filters are convolved with the images to produce additional images. This is an **automated feature extraction** stage.

stride

This means that the filters learn to do e.g. edge detection early in the network and then detect windows, buildings, etc. in later layers.

Benefit: independent of w… …es a… …e



Credit: AlexNet paper

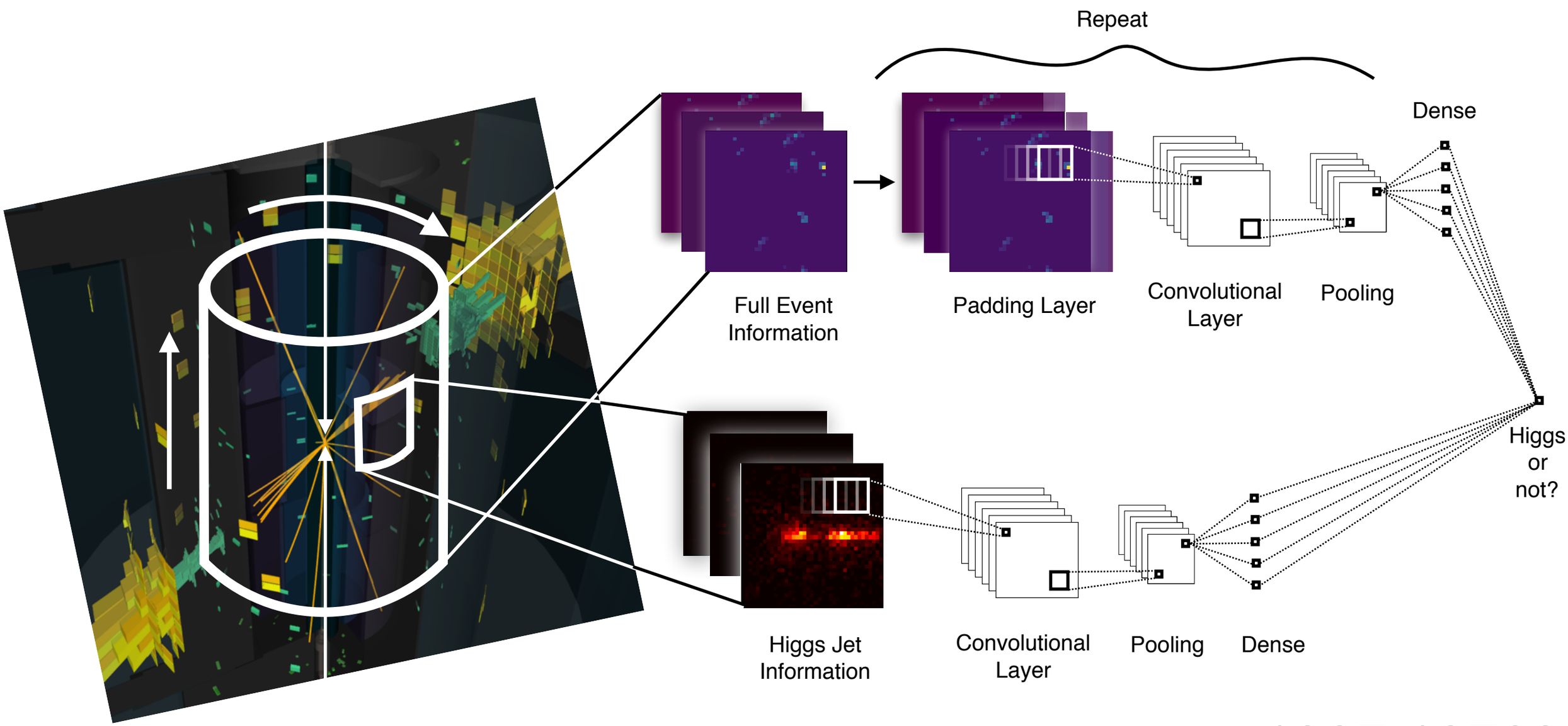
#parameters = (size of filter) x (# of filters)

…independent of image size!

Convolutional neural networks

Related topics: padding and (max) pooling.

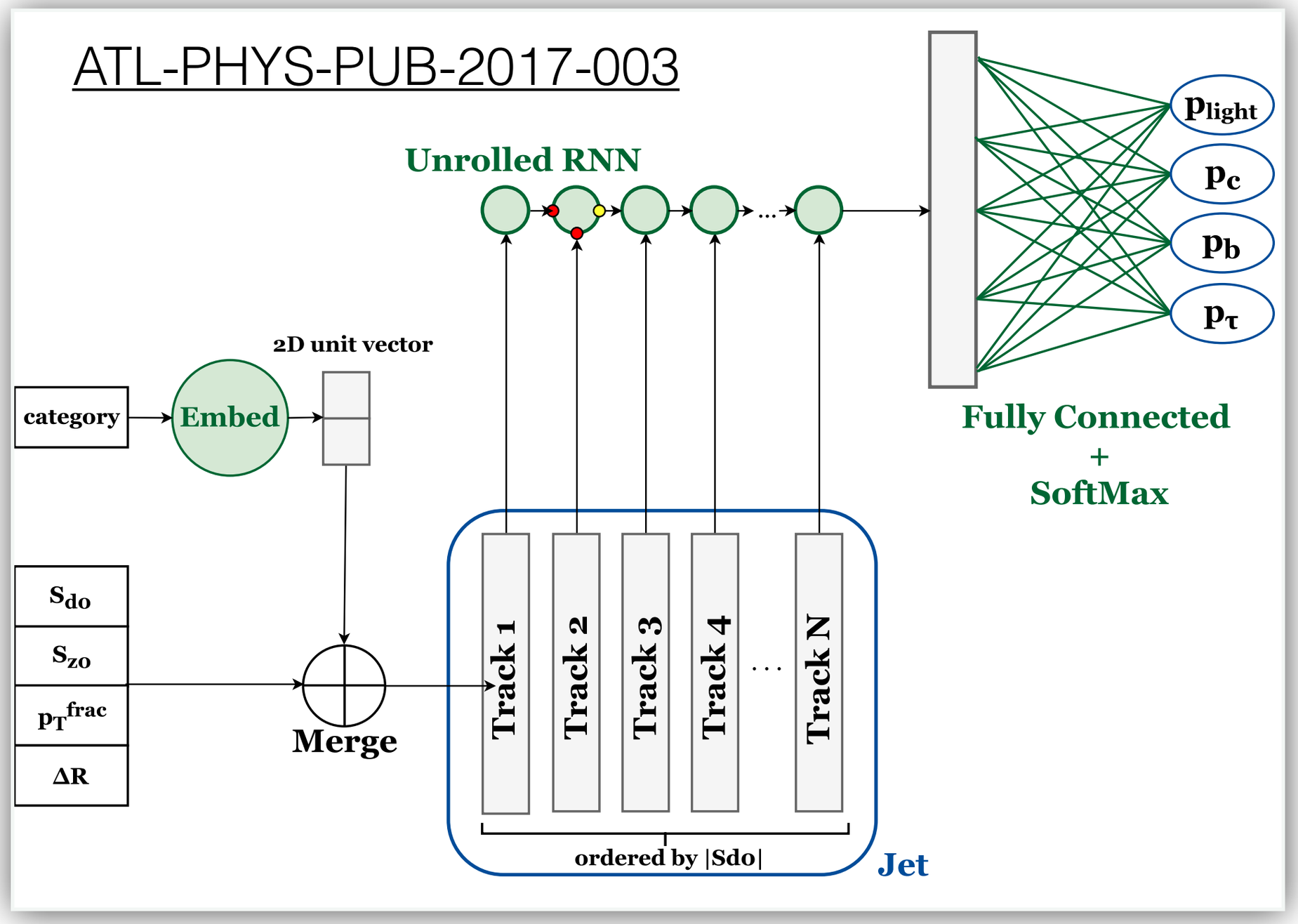
Usually a CNN ends with some fully connected layers.



1807.10768

Recurrent neural networks

The community standard for **text data** is RNNs. **Variable-length ordered inputs** are fed into a recurrent unit.



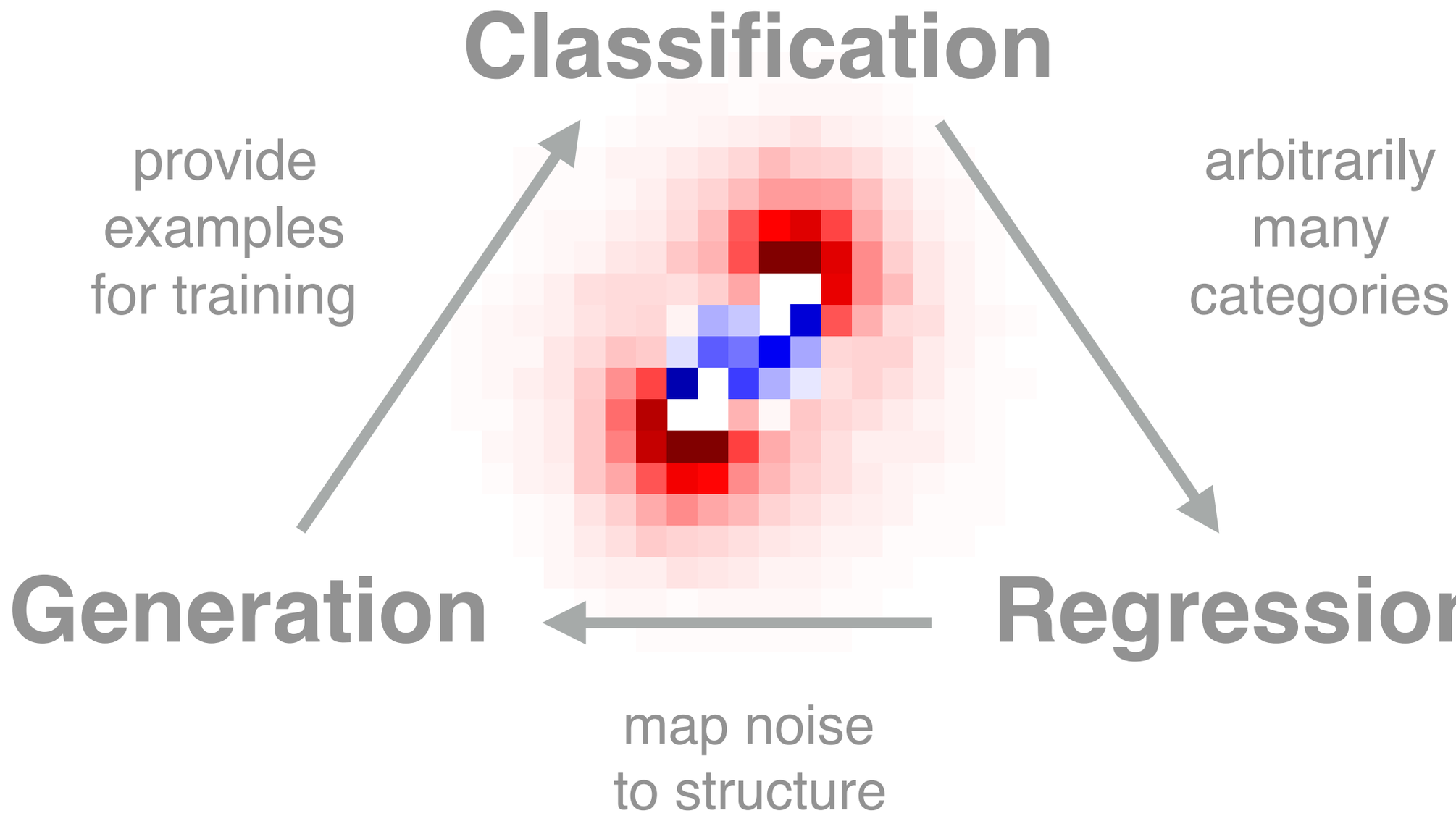
Other neural network architectures

There are many other architectures that can be useful for handling unordered variable length data, graphs, trees, etc.

Picking the “right” architecture can result more efficient and robust training and better performance.

It is also possible (and often not hard) to make new architectures that are specifically designed for your task!

Beyond classification



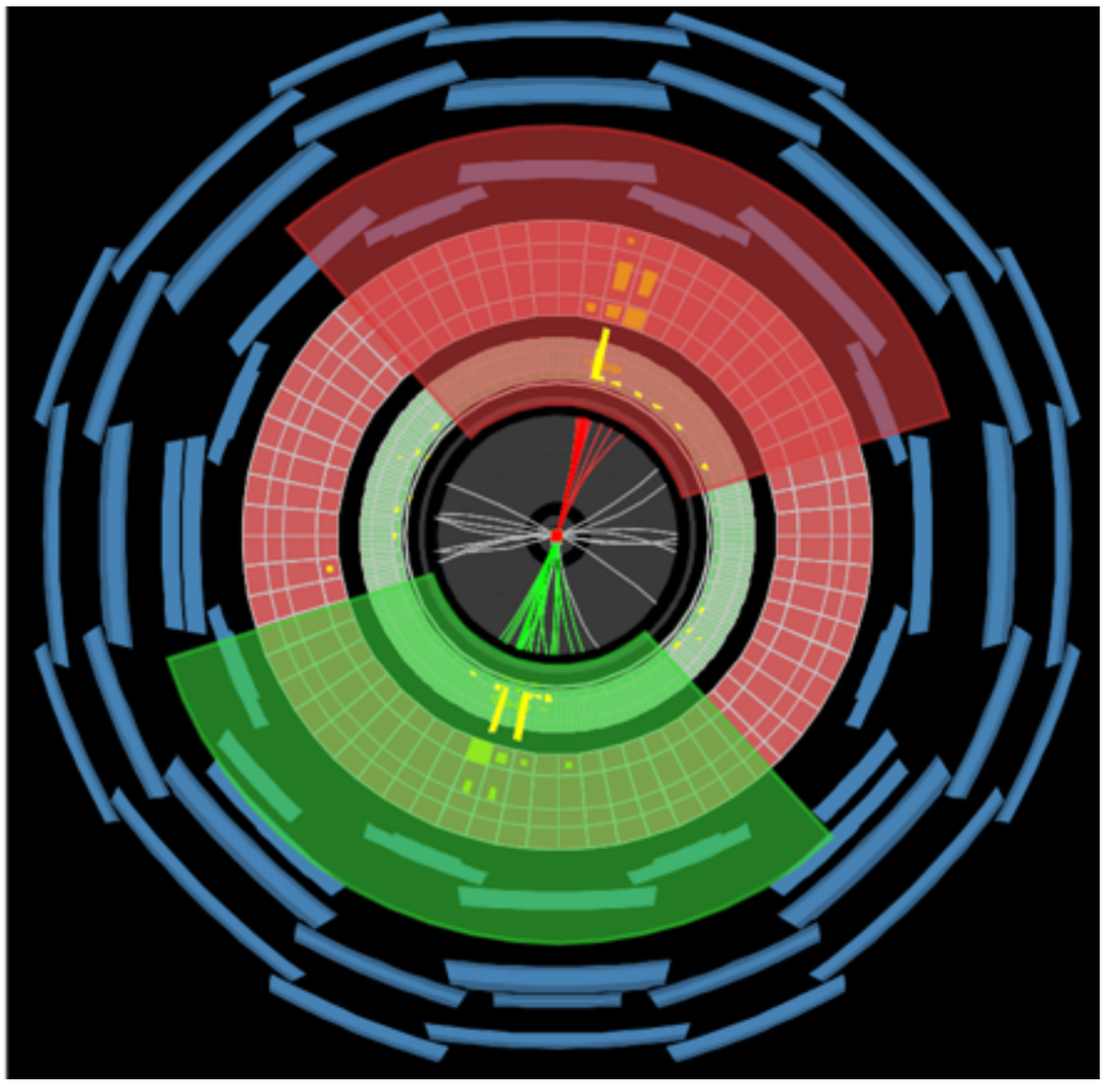
I'll give some concrete examples tomorrow, but I wanted to mention the breadth of possibilities already today.

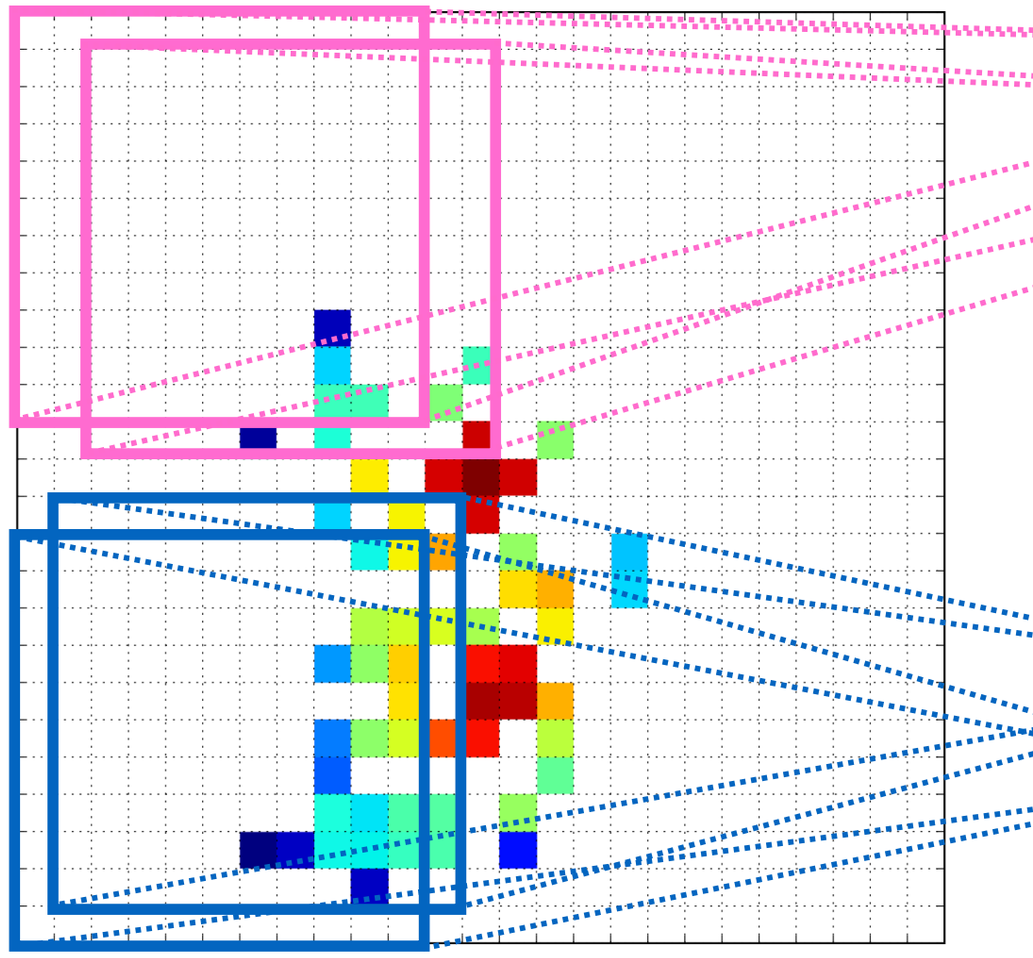
Before finishing: some practical matters

- Hardware
  - GPUs can be much faster for big networks. If you don't have access to GPUs, consider cloud resources!
- Preprocessing
  - While choosing the right architecture can improve performance, so can preprocessing. However, be warned that preprocessing can also remove information if not done carefully.
  - Standardization - when your inputs are not O(1) floats, it is useful to subtract the mean and divide by the standard deviation.
- Hyperparameter tuning
  - Be warned that this non-gradient-based part of training is very important! It is not fair to compare methods if you have not optimized each one.

The future

(D)NN's are powerful tools that will help us fully exploit the physics potential of our theory and experiments.





We must be cautious to apply the right tool for the right job.
The more you know, the less black the boxes will be…