

Statistical Data Analysis 2024/25

Lecture Week 5



London Postgraduate Lectures on Particle Physics
University of London MSc/MSci course PH4515



Glen Cowan
Physics Department
Royal Holloway, University of London
`g.cowan@rhul.ac.uk`
`www.pp.rhul.ac.uk/~cowan`

Course web page via RHUL moodle (PH4515) and also
`www.pp.rhul.ac.uk/~cowan/stat_course.html`

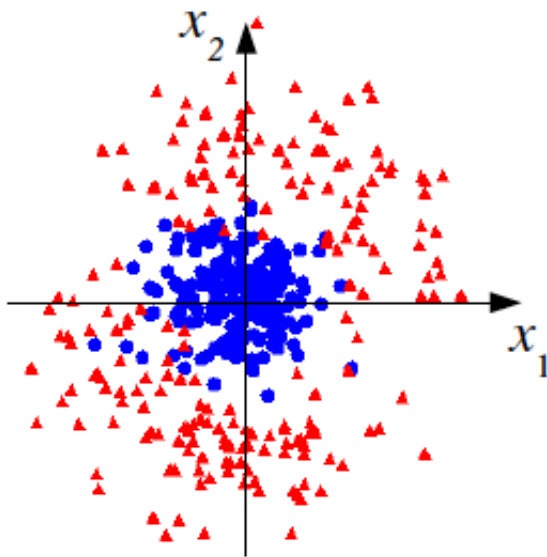
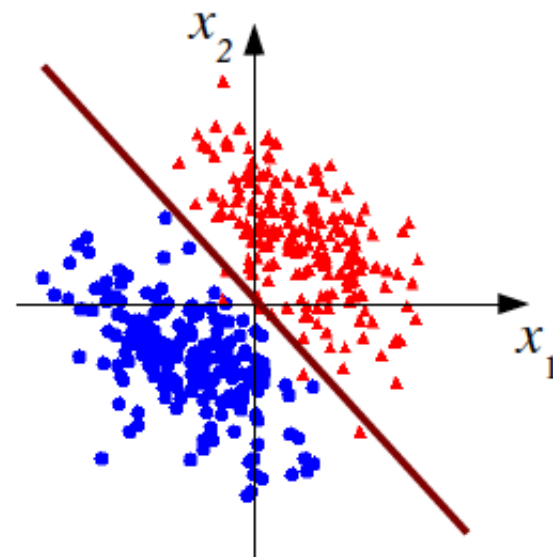
Statistical Data Analysis

Lecture 5-1

- Beyond linear classifiers
- Neural networks

Linear decision boundaries

A linear decision boundary is only optimal when both classes follow multivariate Gaussians with equal covariances and different means.



For some other cases a linear boundary is almost useless.

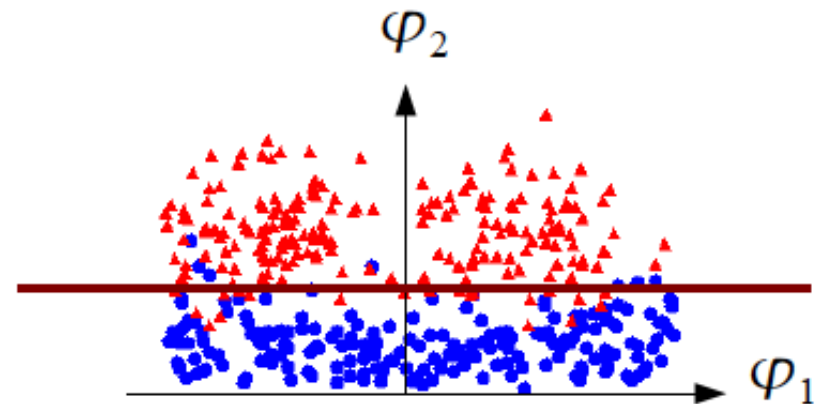
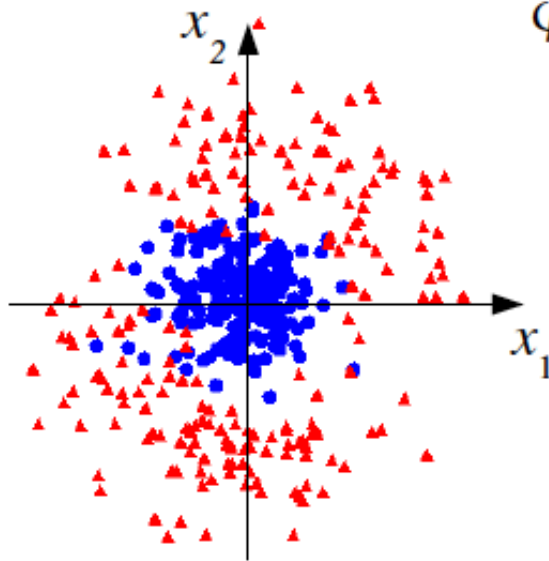
Nonlinear transformation of inputs

We can try to find a transformation, $x_1, \dots, x_n \rightarrow \varphi_1(\vec{x}), \dots, \varphi_m(\vec{x})$ so that the transformed “feature space” variables can be separated better by a linear boundary:

$$\varphi_1 = \tan^{-1}(x_2/x_1)$$

$$\varphi_2 = \sqrt{x_1^2 + x_2^2}$$

Here, guess fixed
basis functions
(no free parameters)



Neural networks

Neural networks originate from attempts to model neural processes (McCulloch and Pitts, 1943; Rosenblatt, 1962).

Widely used in many fields, and for many years the only “advanced” multivariate method popular in HEP.

We can view a neural network as a specific way of parametrizing the basis functions used to define the feature space transformation.

The training data are then used to adjust the parameters so that the resulting discriminant function has the best performance.

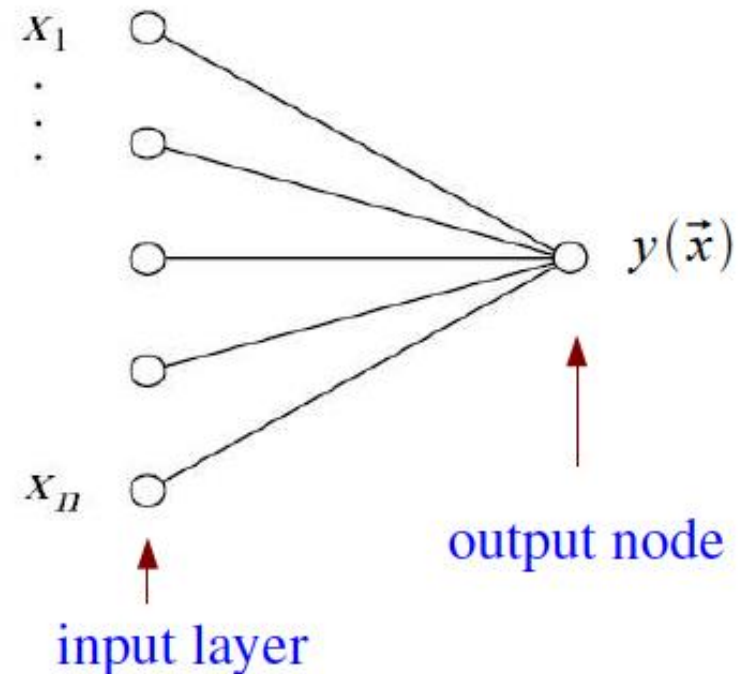
The single layer perceptron

Define the discriminant using $y(\vec{x}) = h\left(w_0 + \sum_{i=1}^n w_i x_i\right)$

where h is a nonlinear, monotonic **activation function**; we can use e.g. the logistic sigmoid $h(x) = (1 + e^{-x})^{-1}$.

If the activation function is monotonic, the resulting $y(\mathbf{x})$ is equivalent to the original linear discriminant.

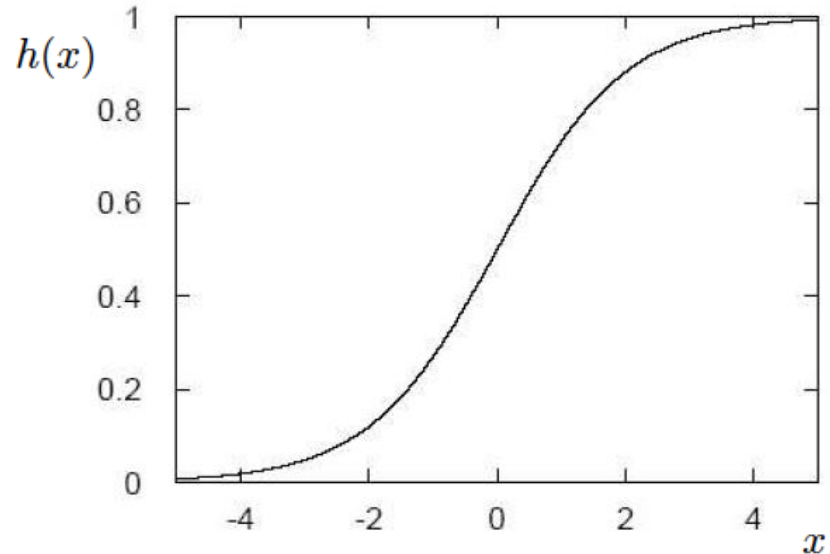
This is called the **single layer perceptron**:



The activation function

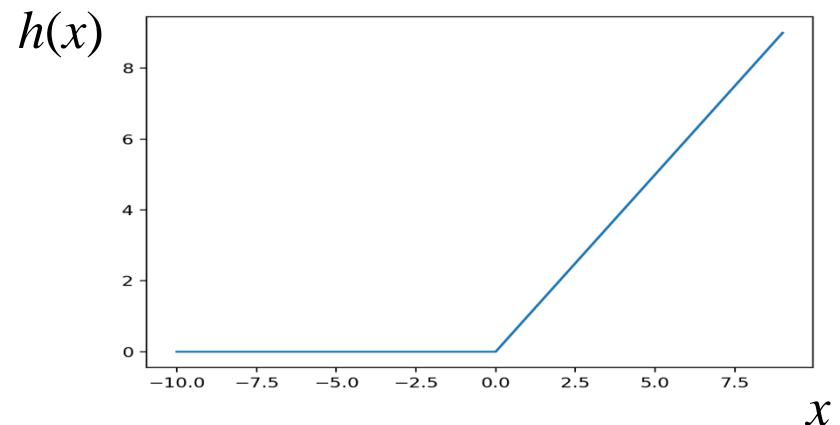
Can use e.g. the “logistic sigmoid”:

$$h(x) = \frac{1}{1 + e^{-x}}$$



or (esp. with deep neural networks) the “Rectified Linear Unit” (ReLU) function:

$$h(x) = \begin{cases} 0 & x \leq 0, \\ x & x > 0. \end{cases}$$



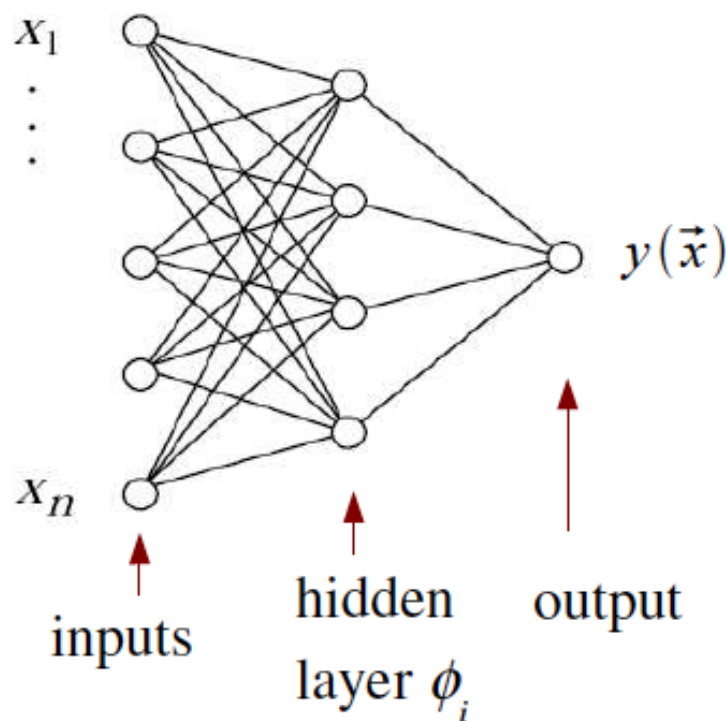
The multilayer perceptron

Now use this idea to define not only the output $y(\mathbf{x})$, but also the set of transformed inputs $\varphi_1(\vec{x}), \dots, \varphi_m(\vec{x})$ that form a “hidden layer”:

Superscript for weights indicates layer number

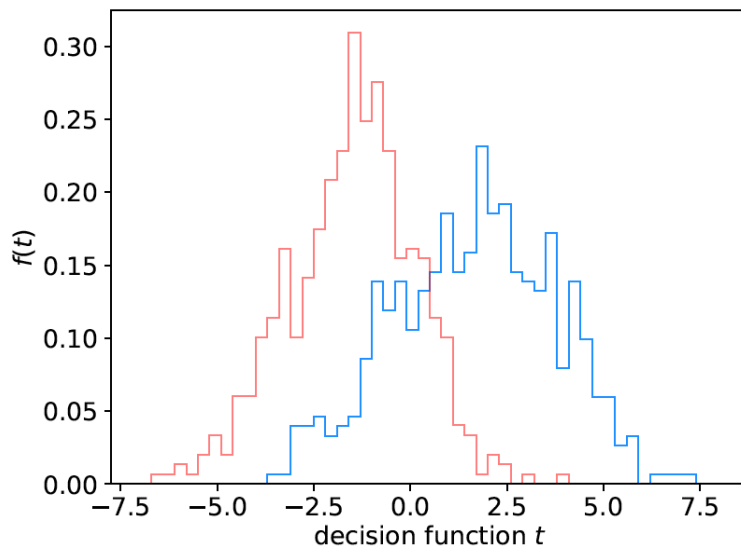
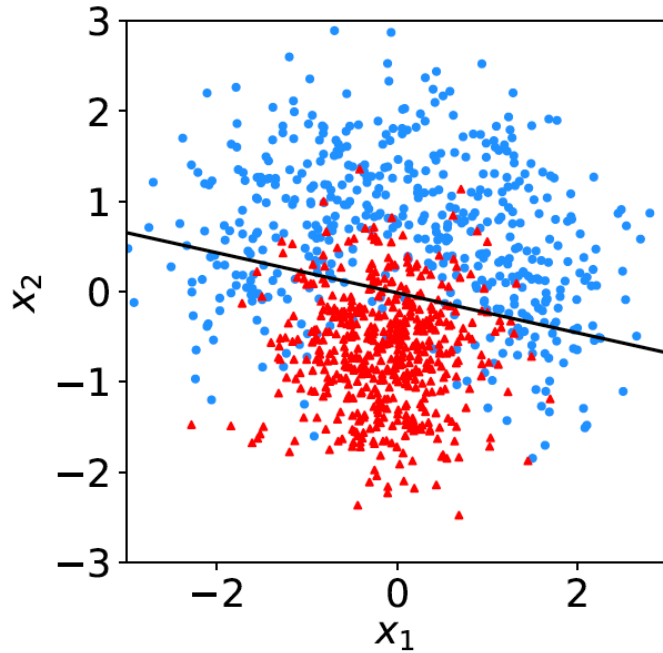
$$\varphi_i(\vec{x}) = h\left(w_{i0}^{(1)} + \sum_{j=1}^n w_{ij}^{(1)} x_j\right)$$

$$y(\vec{x}) = h\left(w_{10}^{(2)} + \sum_{j=1}^m w_{1j}^{(2)} \varphi_j(\vec{x})\right)$$

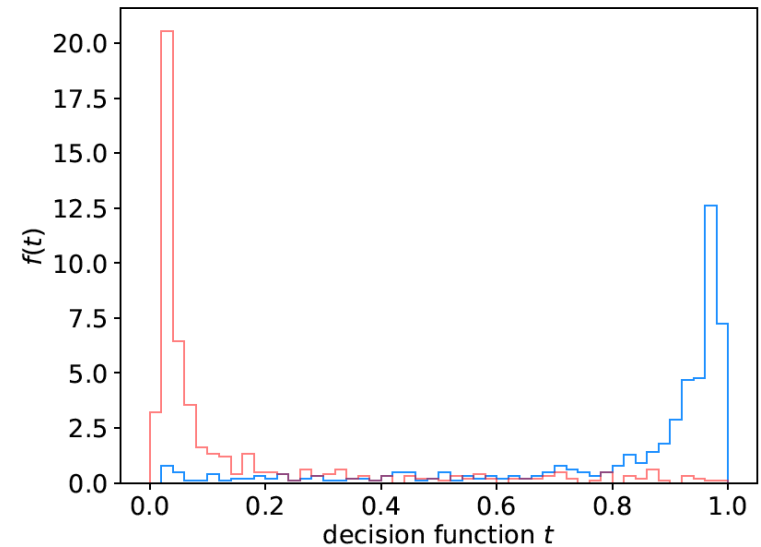
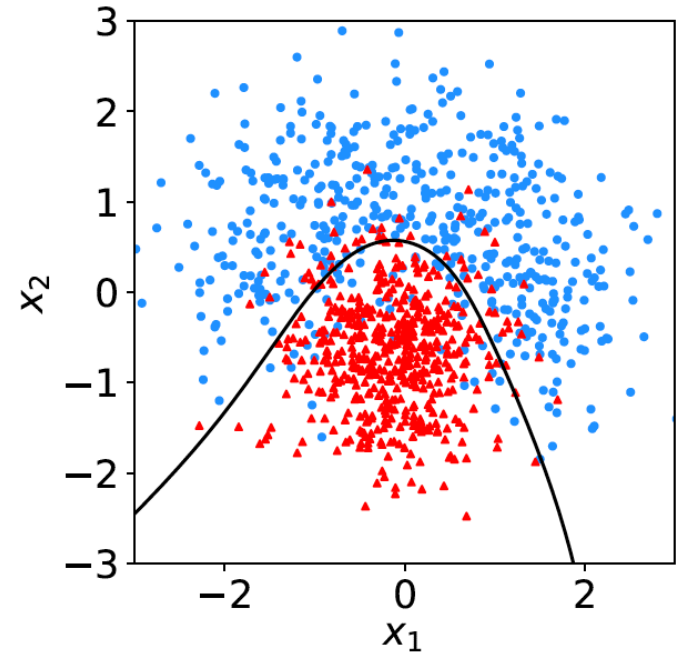


This is the **multilayer perceptron**, our basic neural network model; straightforward to generalize to multiple hidden layers.

Fisher (linear):



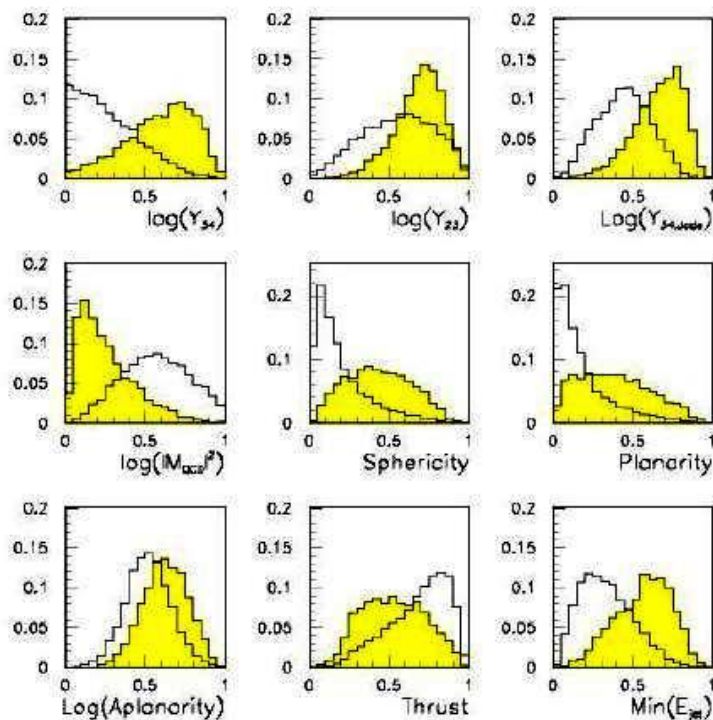
Neural network:



Neural network example from LEP II

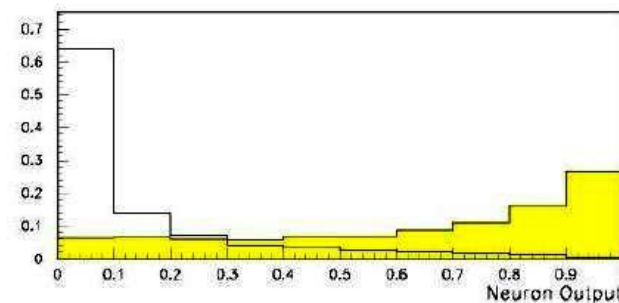
Signal: $e^+e^- \rightarrow W^+W^-$ (often 4 well separated hadron jets)

Background: $e^+e^- \rightarrow q\bar{q}g\bar{g}$ (4 less well separated hadron jets)



← input variables based on jet structure, event shape, ...
none by itself gives much separation.

Neural network output:



(Garrido, Juste and Martinez, ALEPH 96-144)

Statistical Data Analysis

Lecture 5-2

- Network architecture
- Training neural networks
- Overtraining

Network architecture

Theorem: An MLP with a single hidden layer having a sufficiently large number of nodes can approximate arbitrarily well the optimal decision boundary.

Holds for any continuous non-polynomial activation function

Leshno, Lin, Pinkus and Schocken (1993) *Neural Networks* 6, 861-867

However, the number of required nodes may be very large; cannot train well with finite samples of training data.

Recent advances in *Deep Neural Networks* have shown important advantages in having multiple hidden layers.

For particle physics applications of Deep Learning, see e.g.

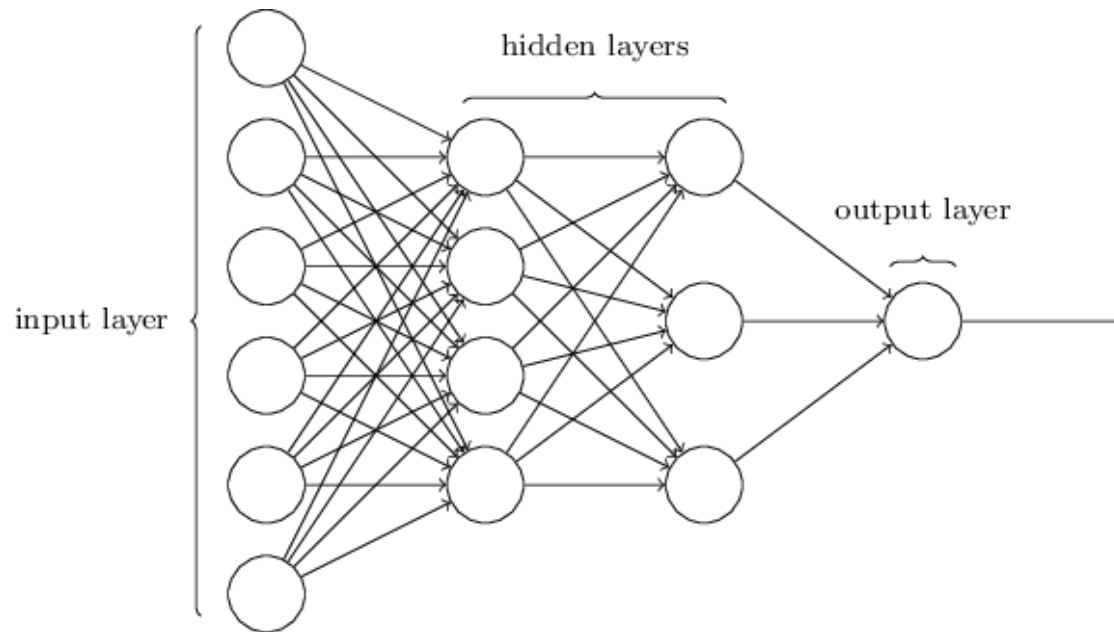
Baldi, Sadowski and Whiteson, *Nature Communications* 5 (2014); arXiv:1402.4735.

Dan Guest, Kyle Cranmer, Daniel Whiteson, *Deep Learning and its Application to LHC Physics*, *Annu. Rev. Nucl. Part. Sci.* 2018. 68:1–22, arXiv:1806.11484.

Deep Neural Networks

The multilayer perceptron can be generalized to have an arbitrary number of hidden layers, with an arbitrary number of nodes in each (= “network architecture”).

A “deep” network has several (or many) hidden layers:



“Deep Learning” is a very recent and active field of research.

Network training

The type of each training event is known, i.e., for event a we have:

$$\begin{aligned}\vec{x}_a &= (x_1, \dots, x_n) && \text{the input variables, and} \\ t_a &= 0, 1 && \text{a numerical label for event type ("target value")}\end{aligned}$$

Let \mathbf{w} denote the set of all of the weights of the network. We can determine their optimal values by minimizing a sum-of-squares “error function” (or “loss function”)

$$E(\mathbf{w}) = \frac{1}{2} \sum_{a=1}^N |y(\vec{x}_a, \mathbf{w}) - t_a|^2 = \sum_{a=1}^N E_a(\mathbf{w})$$



Contribution to error function
from each event

Numerical minimization of $E(\mathbf{w})$

Consider gradient descent method: from an initial guess in weight space $\mathbf{w}^{(1)}$ take a small step in the direction of maximum decrease.

I.e. for the step τ to $\tau+1$,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$



learning rate ($\eta > 0$)

If we do this with the full error function $E(\mathbf{w})$, gradient descent does surprisingly poorly; better to use “conjugate gradients”.

But gradient descent turns out to be useful with an online (sequential) method, i.e., update \mathbf{w} for randomly chosen event a , (or “mini-batch”)

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_a(\mathbf{w}^{(\tau)})$$

(“stochastic
gradient descent”)

Error backpropagation

Error backpropagation (“backprop”) is an algorithm for finding the derivatives required for gradient descent minimization.

The network output can be written $y(\mathbf{x}) = h(u(\mathbf{x}))$ where

$$u(\vec{x}) = \sum_{j=0} w_{1j}^{(2)} \varphi_j(\vec{x}), \quad \varphi_j(\vec{x}) = h\left(\sum_{k=0} w_{jk}^{(1)} x_k\right)$$


where we defined $\phi_0 = x_0 = 1$ and wrote the sums over the nodes in the preceding layers starting from 0 to include the offsets.

So e.g. for event a we have

$$\frac{\partial E_a}{\partial w_{1j}^{(2)}} = (y_a - t_a) h'(u(\vec{x})) \varphi_j(\vec{x})$$

Chain rule gives all the needed derivatives.

derivative of
activation function



Comments on network training

The algorithms for adjusting the network parameters have become a very active field of research (and beyond the scope of this course).

Recent ideas include:

- “Deep” neural nets, use of ReLU activation function

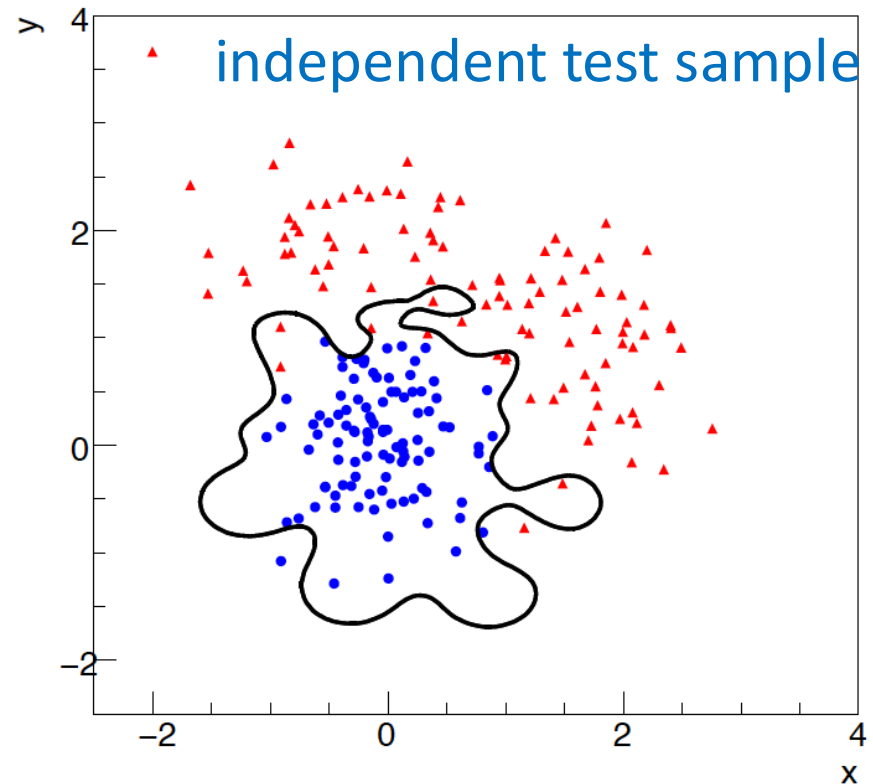
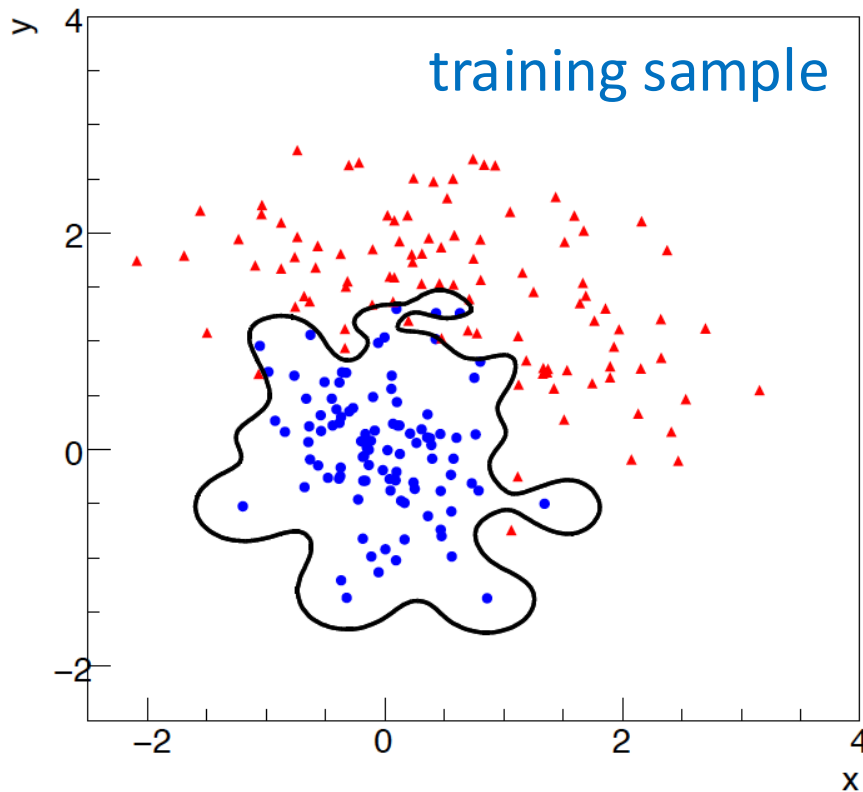
- Stochastic gradient descent: estimate of gradient approximated by a randomly selected subset of the data.

- Dropout: randomly exclude nodes during training (prevents “overtraining”)

Overtraining

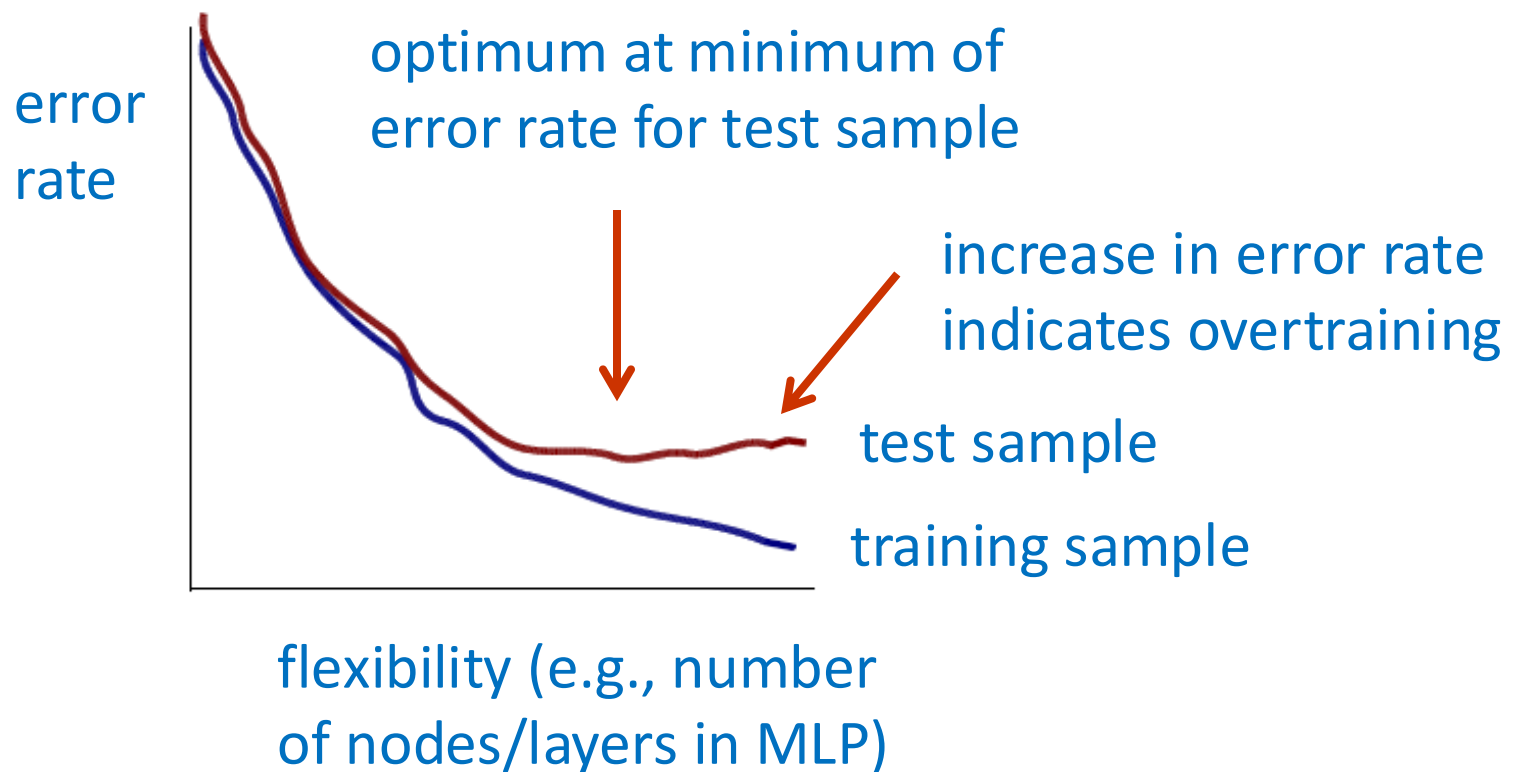
Including more parameters in a classifier makes its decision boundary increasingly flexible, e.g., more nodes/layers for a neural network.

A “flexible” classifier may conform too closely to the training points; the same boundary will not perform well on an independent test data sample (→ “overtraining”).



Monitoring overtraining

If we monitor the fraction of misclassified events (or similar, e.g., error function $E(\mathbf{w})$) for test and training samples, it will usually decrease for both as the boundary is made more flexible:



Statistical Data Analysis

Lecture 5-3

- Non-parametric probability density estimation
- Kernel density estimator

Probability Density Estimation (PDE)

A possible way to approximate the likelihood ratio for two hypotheses $f(\mathbf{x}|H_0)$ and $f(\mathbf{x}|H_1)$ is to first find a *non-parametric estimator* for the corresponding pdfs and then use these to define the test statistic (decision function),

$$y(\mathbf{x}) = \frac{\hat{f}(\mathbf{x}|H_1)}{\hat{f}(\mathbf{x}|H_0)} \quad \text{(hats denote estimators)}$$

Here the term “non-parametric” means the the estimate will be very general, not necessarily from a specific pdf family, and have a “local” character reflecting training data values.

The n -dimensional histogram was a brute-force example of this; there are better ways.

Non-parametric pdf estimates are useful in many ways; here for obtaining a test statistic but one example.

Correlation vs. independence

In general a multivariate pdf $f(\mathbf{x})$ for $\mathbf{x} = (x_1, \dots, x_n)$ does not factorize into a product of the marginal pdfs for the individual variables, and

$$f(\mathbf{x}) = \prod_{i=1}^n f_i(x_i)$$

only holds if the components of \mathbf{x} are independent.

In particular, the components of \mathbf{x} may in general have nonzero covariances, i.e., they are correlated:

$$V_{ij} = \text{cov}[x_i, x_j] = E[x_i x_j] - E[x_i]E[x_j] \neq 0$$

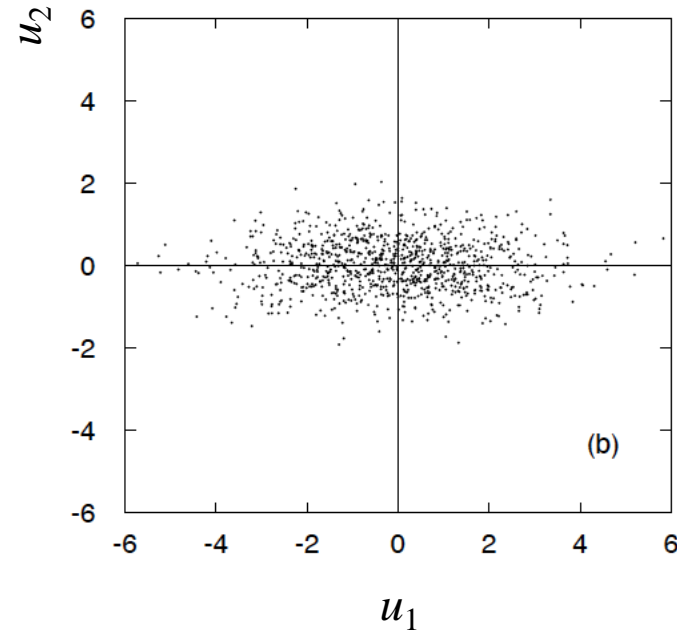
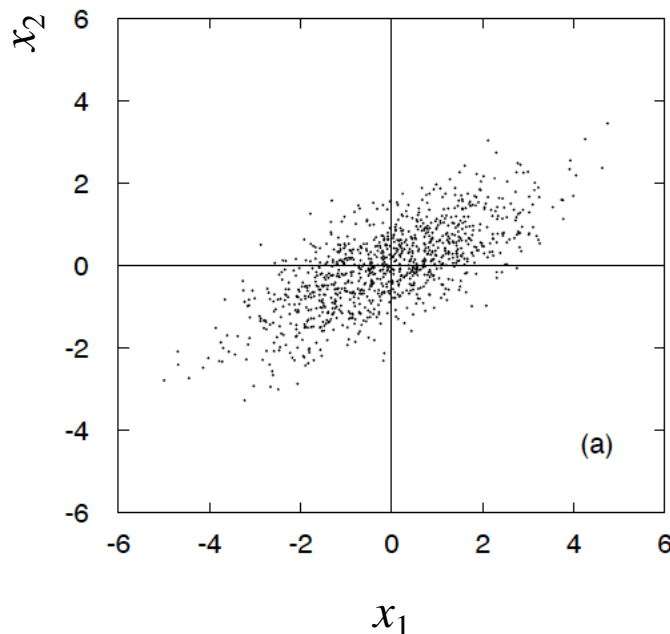
Decorrelation of input variables

We can always define new input variables by an orthogonal transformation such that the transformed variables are uncorrelated:

$$\mathbf{u} = A\mathbf{x}$$

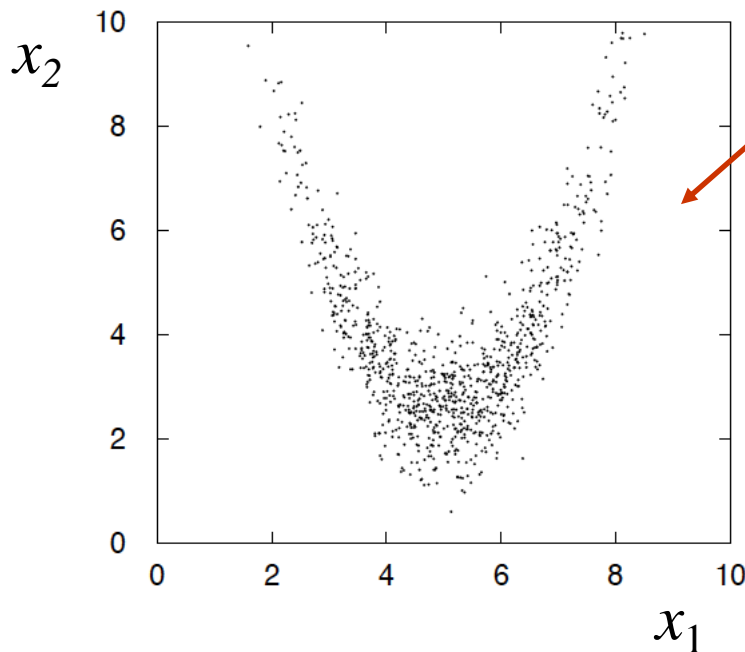
$$\text{cov}[u_i, u_j] = V[u_i]\delta_{ij}$$

One can show that this is achieved when the rows of the matrix A are the eigenvectors of $V_{ij} = \text{cov}[x_i, x_j]$ (cf. SDA Sec. 1.7).



Decorrelation is only first step

But even with zero correlation, a multivariate pdf $f(\mathbf{x})$ will in general have features such that the components are not independent.



pdf with zero covariance (no tilt)
but components still not
independent, since clearly

$$f(x_2|x_1) \equiv \frac{f(x_1, x_2)}{f_1(x_1)} \neq f_2(x_2)$$

and therefore

$$f(x_1, x_2) \neq f_1(x_1)f_2(x_2)$$

Naive Bayes method

First “decorrelate” \mathbf{x} , i.e., find $\mathbf{u} = A\mathbf{x}$, with $\text{cov}[u_i, u_j] = V[u_i] \delta_{ij}$.

Pdfs of \mathbf{x} and \mathbf{u} are then related by

$$f(\mathbf{x}) = |J|g(\mathbf{u}(\mathbf{x})) \quad \text{where} \quad J = \det(A)$$

Suppose that the “decorrelated” $g(\mathbf{u})$ can be approximated by product of marginal pdfs

$$g(\mathbf{u}) \approx \prod_{i=1}^n g_i(u_i)$$

and take as an *estimator* for $f(\mathbf{x})$

$$\hat{f}(\mathbf{x}) = |J|g(\mathbf{u}(\mathbf{x})) = |J| \prod_{i=1}^n g_i(u_i(\mathbf{x})) = |\det(A)| \prod_{i=1}^n g_i((A\mathbf{x})_i)$$

Naive Bayes method (2)

Approximate the pdfs separately for the two hypotheses H_0 and H_1 (separate matrices A_0 and A_1 and marginal pdfs $g_{0,i}$, $g_{1,i}$). Then define test statistic as

$$y(\mathbf{x}) = \frac{\hat{f}(\mathbf{x}|H_1)}{\hat{f}(\mathbf{x}|H_0)}$$

Gives “Naive Bayes” classifier.

Reduces problem of estimating an n -dimensional pdf to finding n one-dimensional marginal pdfs $g_i(u_i)$.

Kernel-based PDE (KDE)

Consider d dimensions, N training events, $\mathbf{x}_1, \dots, \mathbf{x}_N$,
estimate $f(\mathbf{x})$ with

$$\hat{f}(\vec{x}) = \frac{1}{Nh^d} \sum_{i=1}^N K\left(\frac{\vec{x} - \vec{x}_i}{h}\right)$$

\vec{x} where we want to know pdf

\vec{x} of i^{th} training event

kernel

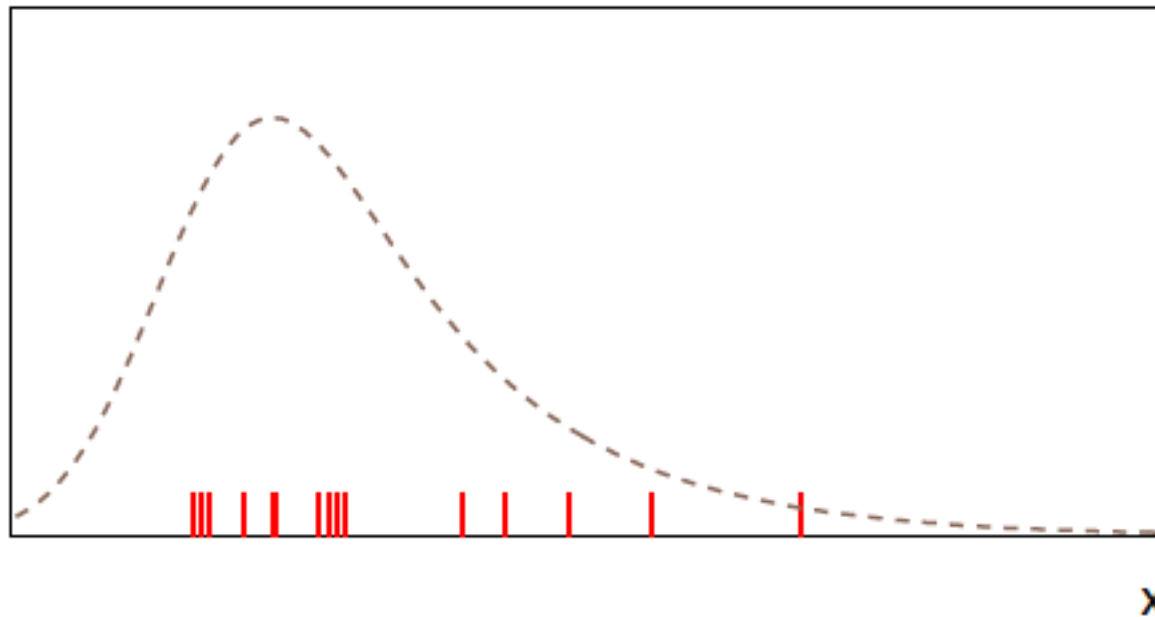
bandwidth (smoothing parameter)

Use e.g. Gaussian kernel: $K(\vec{x}) = \frac{1}{(2\pi)^{d/2}} e^{-|\vec{x}|^2/2}$

and do individually for each component (i.e. $f(\mathbf{x}) \rightarrow g_i(u_i)$).

Gaussian KDE in 1-dimension

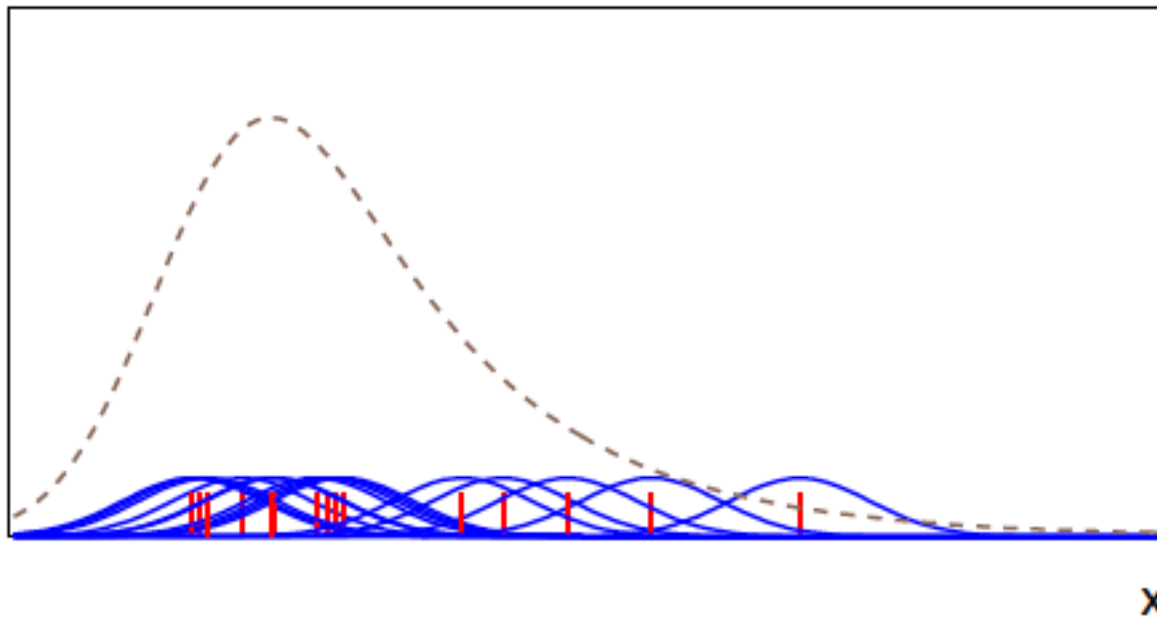
Suppose the pdf (dashed line) below is not known, but we can generate or observe values that follow it (the red tick marks):



Goal is to find an approximation to the pdf using the data values.

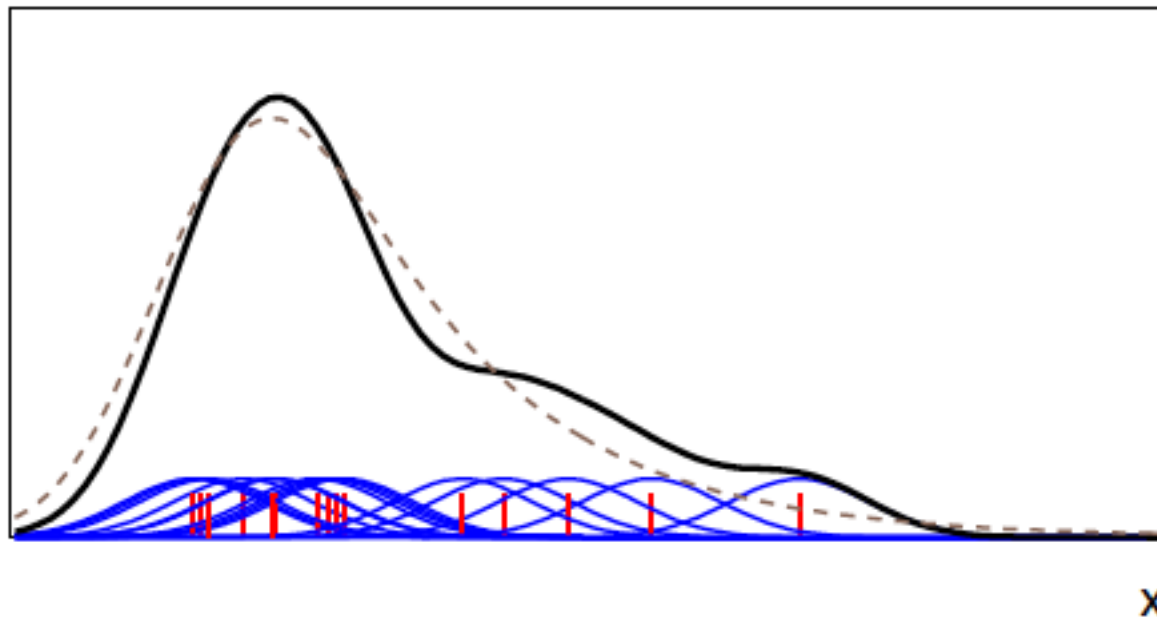
Gaussian KDE in 1-dimension (cont.)

Place a kernel pdf (here a Gaussian) centred around each generated event weighted by $1/N_{\text{event}}$:



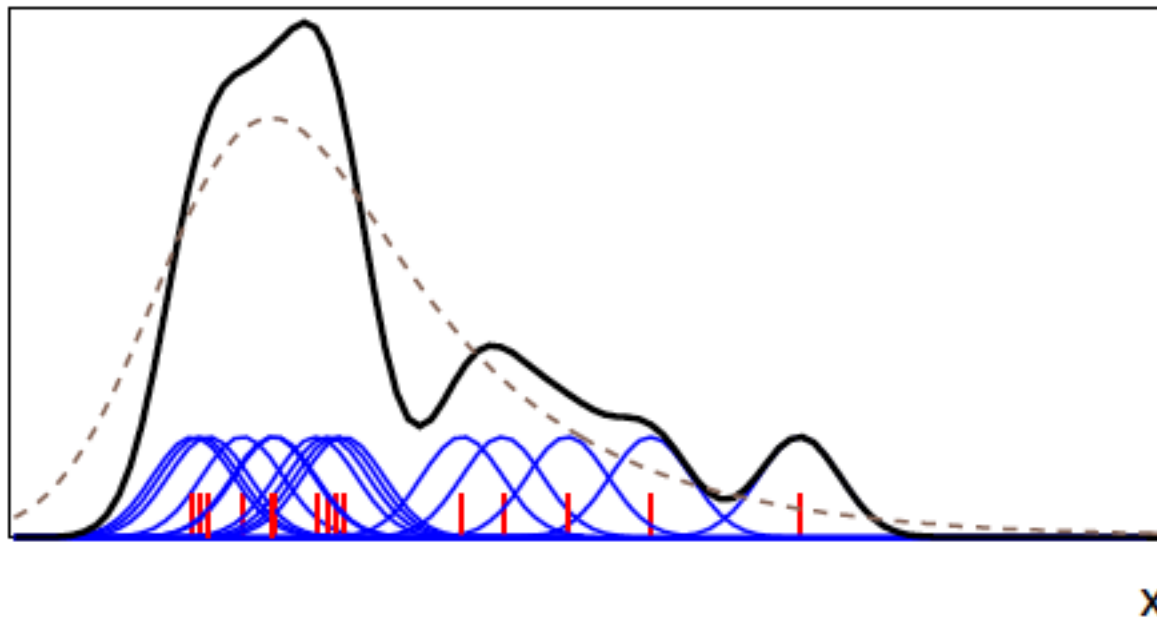
Gaussian KDE in 1-dimension (cont.)

The KDE estimate the pdf is given by the sum of all of the Gaussians:



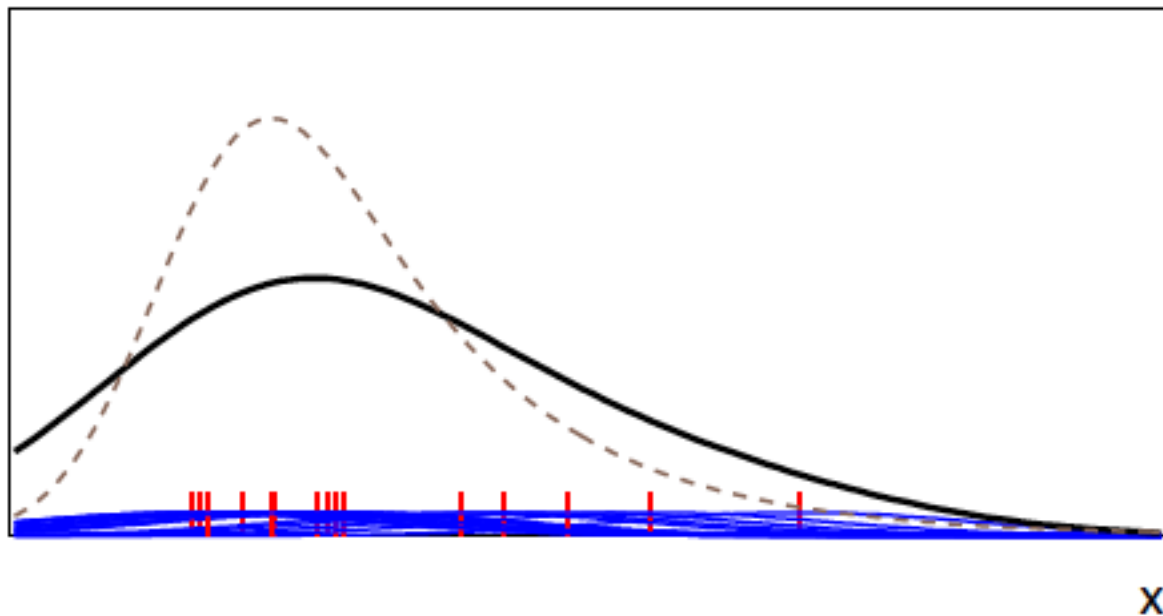
Choice of kernel width

The width h of the Gaussians is analogous to the bin width of a histogram. If it is too small, the estimator has noise:



Choice of kernel width (cont.)

If width of Gaussian kernels too large, structure is washed out:



KDE discussion

Various strategies can be applied to choose width h of kernel based trade-off between bias and variance (noise).

Adaptive KDE allows width of kernel to vary, e.g., wide where target pdf is low (few events); narrow where pdf is high.

Advantage of KDE: no training!

Disadvantage of KDE: to evaluate we need to sum N_{event} terms, so if we have many events this can be slow.

Special treatment required if kernel extends beyond range where pdf defined. Can e.g., renormalize the kernels to unity inside the allowed range; alternatively “mirror” the events about the boundary (contribution from the mirrored events exactly compensates the amount lost outside the boundary).

Software in ROOT: RooKeysPdf (K. Cranmer, CPC 136:198,2001),
or in python: `sklearn.neighbors.KernelDensity`

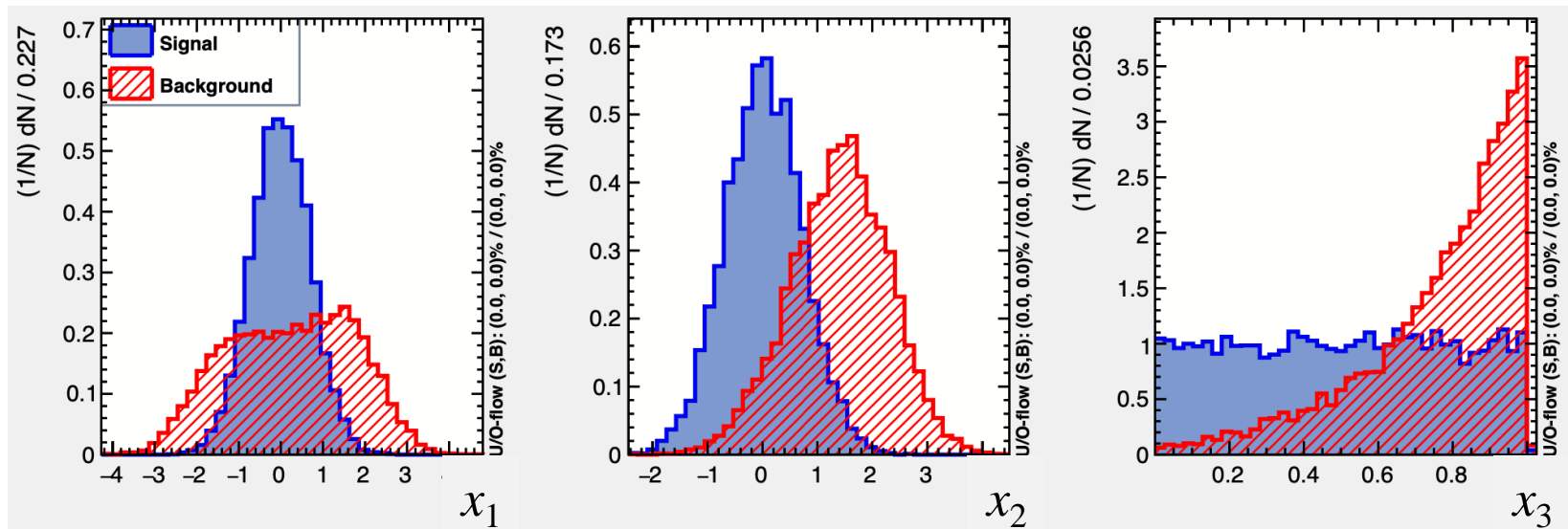
Statistical Data Analysis

Lecture 5-4

- Examples of classifiers
- Boosted decision trees

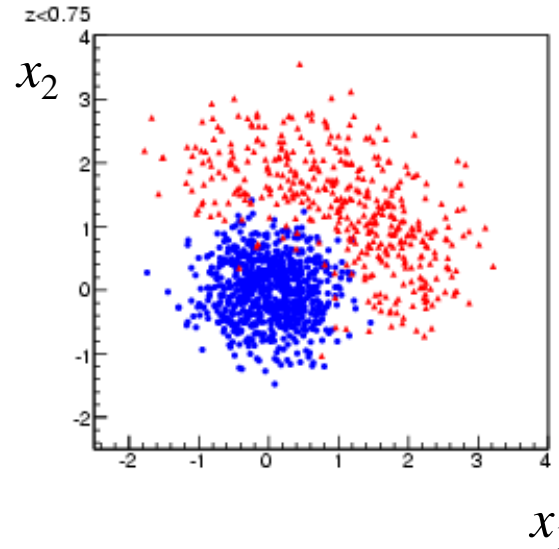
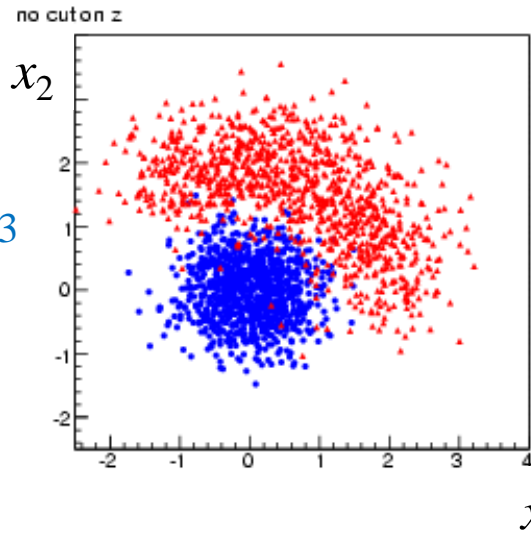
Test example with TMVA

Each event characterized by 3 variables, $\mathbf{x} = (x_1, x_2, x_3)$:



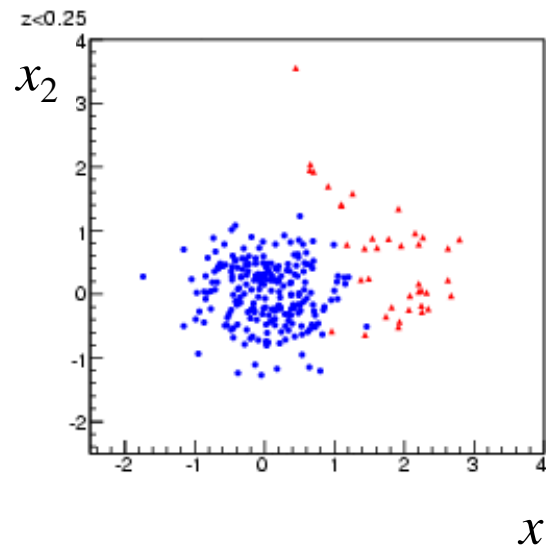
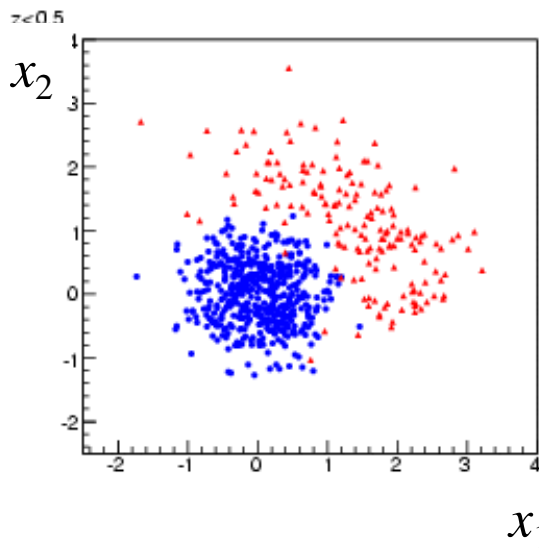
Test example (x_1, x_2, x_3)

no cut on x_3



$x_3 < 0.75$

$x_3 < 0.5$



$x_3 < 0.25$

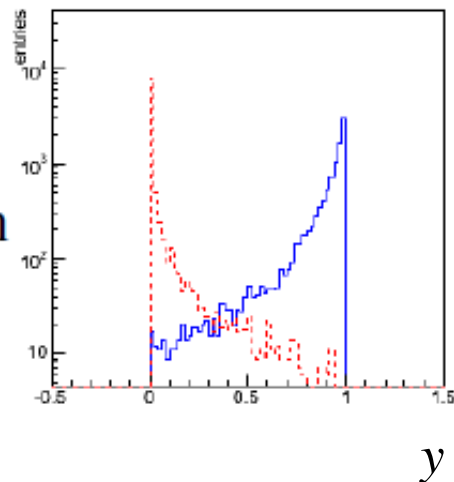
Test example results

Fisher
discriminant

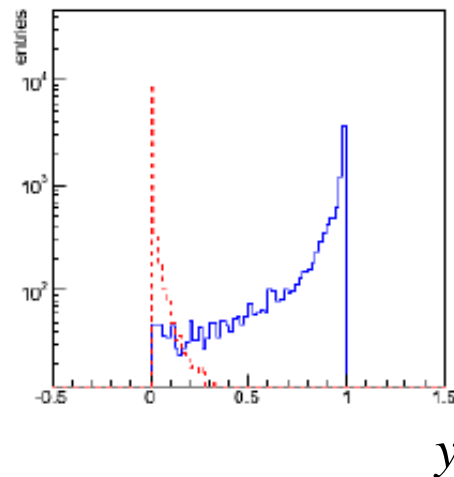
Multilayer
perceptron

Find these on next homework assignment.

Naive Bayes,
no decorrelation

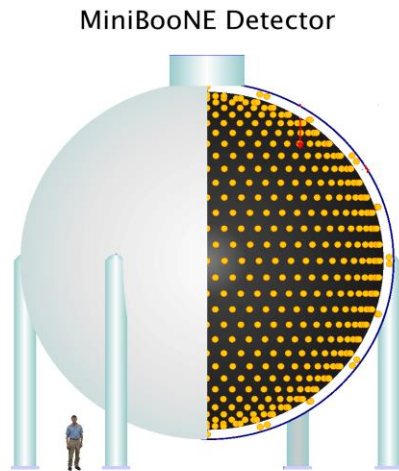


Naive Bayes with
decorrelation

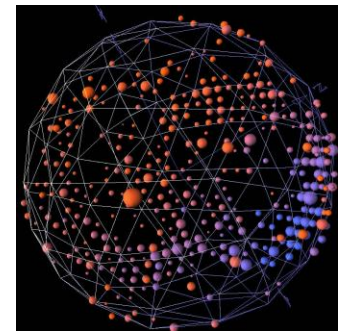
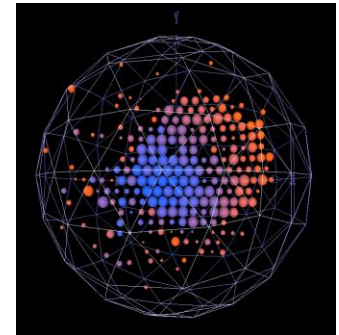
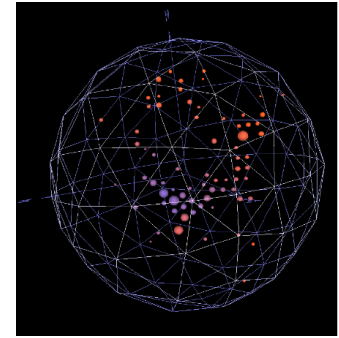
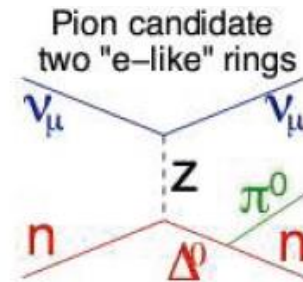
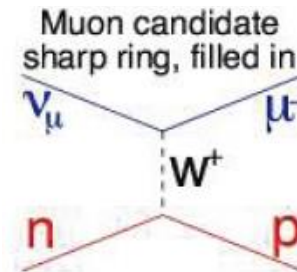
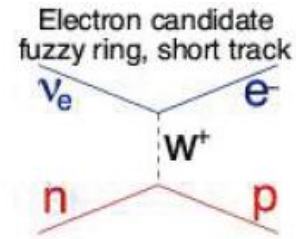


Particle i.d. in MiniBooNE

Detector is a 12-m diameter tank of mineral oil exposed to a beam of neutrinos and viewed by 1520 photomultiplier tubes:



Search for ν_μ to ν_e oscillations required particle i.d. using information from the PMTs.



H.J. Yang, MiniBooNE PID, DNP06

Decision trees

Out of all the input variables, find the one for which with a single “cut” (require e.g. $x < x_c$) gives best improvement in signal purity:

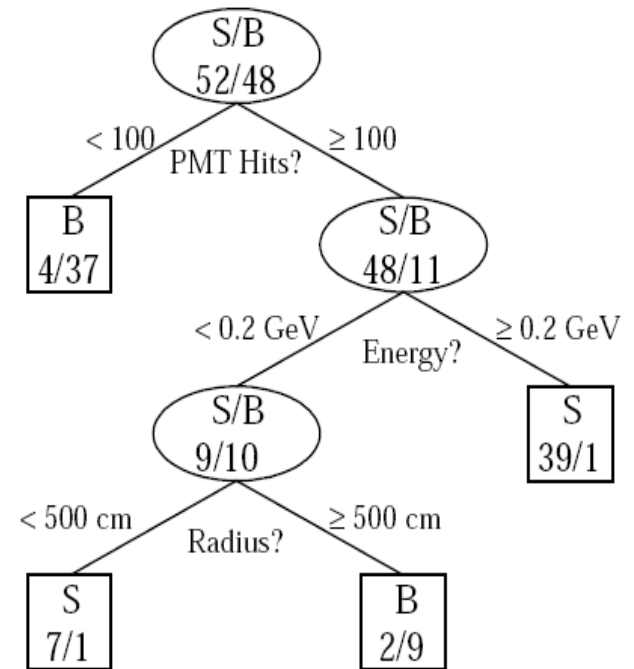
$$p = \frac{\sum_{\text{signal}} w_i}{\sum_{\text{signal}} w_i + \sum_{\text{background}} w_i}$$

where w_i is the weight of the i th event.

Resulting nodes classified as either signal/background.

Iterate until stop criterion reached based on e.g. purity or minimum number of events in a node.

The set of cuts defines the decision boundary.



Example by MiniBooNE experiment,
B. Roe et al., NIM 543 (2005) 577

Finding the best single cut

The level of separation within a node can, e.g., be quantified by the *Gini coefficient*, calculated from the (s or b) purity p as:

$$G = p(1 - p)$$

$p = 0$ or 1 gives $\min G = 0$,

$p = 1/2$ gives $\max G = 1/4$.

For a cut that splits a set of events a into subsets b and c , one can quantify the improvement in separation by the change in weighted Gini coefficients:

$$\Delta = W_a G_a - W_b G_b - W_c G_c \quad \text{where, e.g.,} \quad W_a = \sum_{i \in a} w_i$$

Choose e.g. the cut to the maximize Δ .

Decision trees (2)

The terminal nodes (leaves) are classified a signal or background depending on majority vote (or e.g. signal fraction greater than a specified threshold).

This classifies every point in input-variable space as either signal or background, a decision tree classifier, with discriminant function

$$f(\mathbf{x}) = 1 \text{ if } \mathbf{x} \text{ in signal region, } -1 \text{ otherwise}$$

Decision trees tend to be very sensitive to statistical fluctuations in the training sample.

Ensemble methods such as boosting can be used to reduce these fluctuations.

Boosting

Boosting is a general method for creating a set of classifiers that can be combined to achieve a new one that is better than any individual one (an example of "ensemble learning").

Often applied to decision trees but can be applied to any classifier.

Suppose we have a training sample T consisting of N events with

$\mathbf{x}_1, \dots, \mathbf{x}_N$	event data vectors
y_1, \dots, y_N	true class labels (+1 or -1)
w_1, \dots, w_N	event weights

Define a rule to create from this an ensemble of training samples T_1, T_2, \dots , derive a classifier from each and average them.

Trick is to create modifications in the training samples to give classifiers with smaller error rates than the preceding ones.

A successful example is **AdaBoost** (Freund and Schapire, 1997).

AdaBoost

First initialize the training sample T_1 using the original

$\mathbf{x}_1, \dots, \mathbf{x}_N$	event data vectors
y_1, \dots, y_N	true class labels (+1 or -1)
$w_1^{(1)}, \dots, w_N^{(1)}$	event weights

with the weights equal and normalized such that

$$\sum_{i=1}^N w_i^{(1)} = 1$$

Then train the classifier $f_1(\mathbf{x})$ (e.g., a decision tree) with a method that uses the event weights. Recall for an event at point \mathbf{x} ,

$f_1(\mathbf{x}) = +1$ for \mathbf{x} in signal region, -1 in background region

We will define an iterative procedure that gives a series of classifiers $f_1(\mathbf{x}), f_2(\mathbf{x}), \dots$

Error rate of the k th classifier

At the k th iteration the classifier $f_k(\mathbf{x})$ has an error rate

$$\varepsilon_k = \sum_{i=1}^N w_i^{(k)} I(y_i f_k(\mathbf{x}_i) \leq 0)$$

where $I(X) = 1$ if X is true and is zero otherwise.

Next assign a score to the k th classifier based on its error rate,

$$\alpha_k = \frac{1}{2} \ln \frac{1 - \varepsilon_k}{\varepsilon_k}$$

Updating the event weights

The classifier at each iterative step is found from an updated training sample, in which the weight of event i is modified from step k to step $k+1$ according to

$$w_i^{(k+1)} = w_i^{(k)} \frac{e^{-\alpha_k f_k(\mathbf{x}_i) y_i}}{Z_k}$$

Here Z_k is a normalization factor defined such that the sum of the weights over all events is equal to one.

That is, the weight for event i is increased in the $k+1$ training sample if it was classified incorrectly in step k .

Idea is that next time around the classifier should pay more attention to this event and try to get it right.

Defining the decision function

After K boosting iterations, the final decision function is defined as a weighted linear combination of the $f_k(\mathbf{x})$,

$$t(\mathbf{x}) = \sum_{k=1}^K \alpha_k f_k(\mathbf{x})$$

One can show that the error rate on the training data of the final classifier satisfies the bound

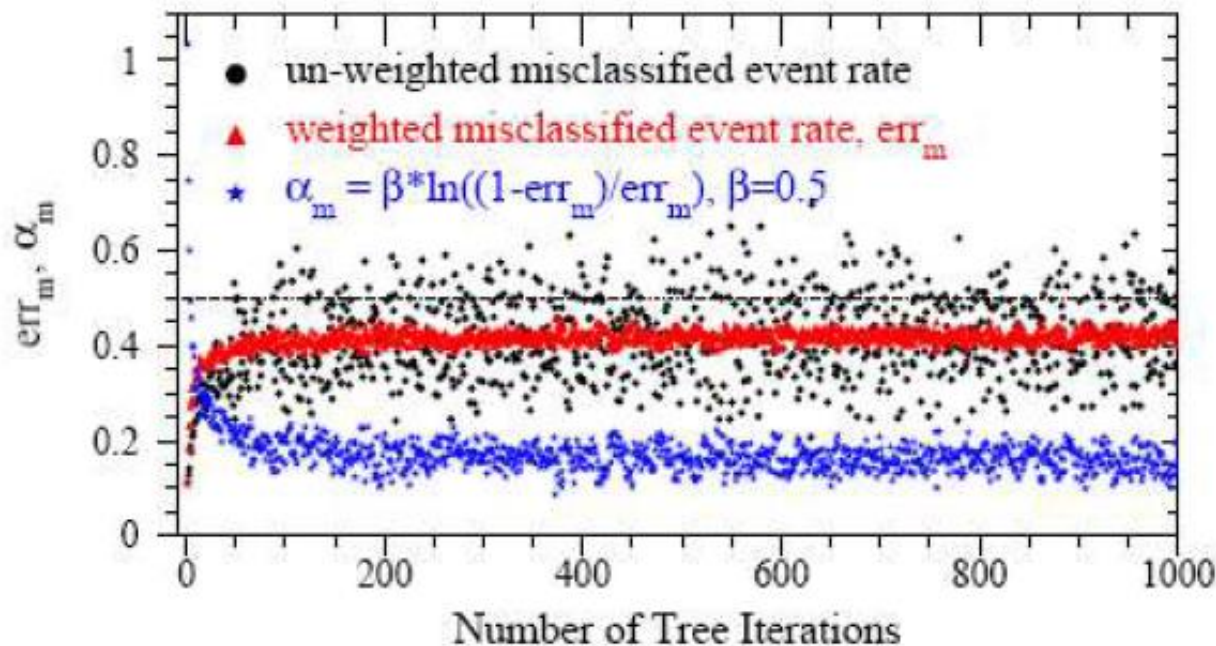
$$\varepsilon \leq \prod_{k=1}^K 2\sqrt{\varepsilon_k(1 - \varepsilon_k)}$$

i.e. as long as the $\varepsilon_k < 1/2$ (better than random guessing), with enough boosting iterations every event in the training sample will be classified correctly.

BDT example from MiniBooNE

~200 input variables for each event (ν interaction producing e , μ or π).

Each individual tree is relatively weak, with a misclassification error rate $\sim 0.4 - 0.45$



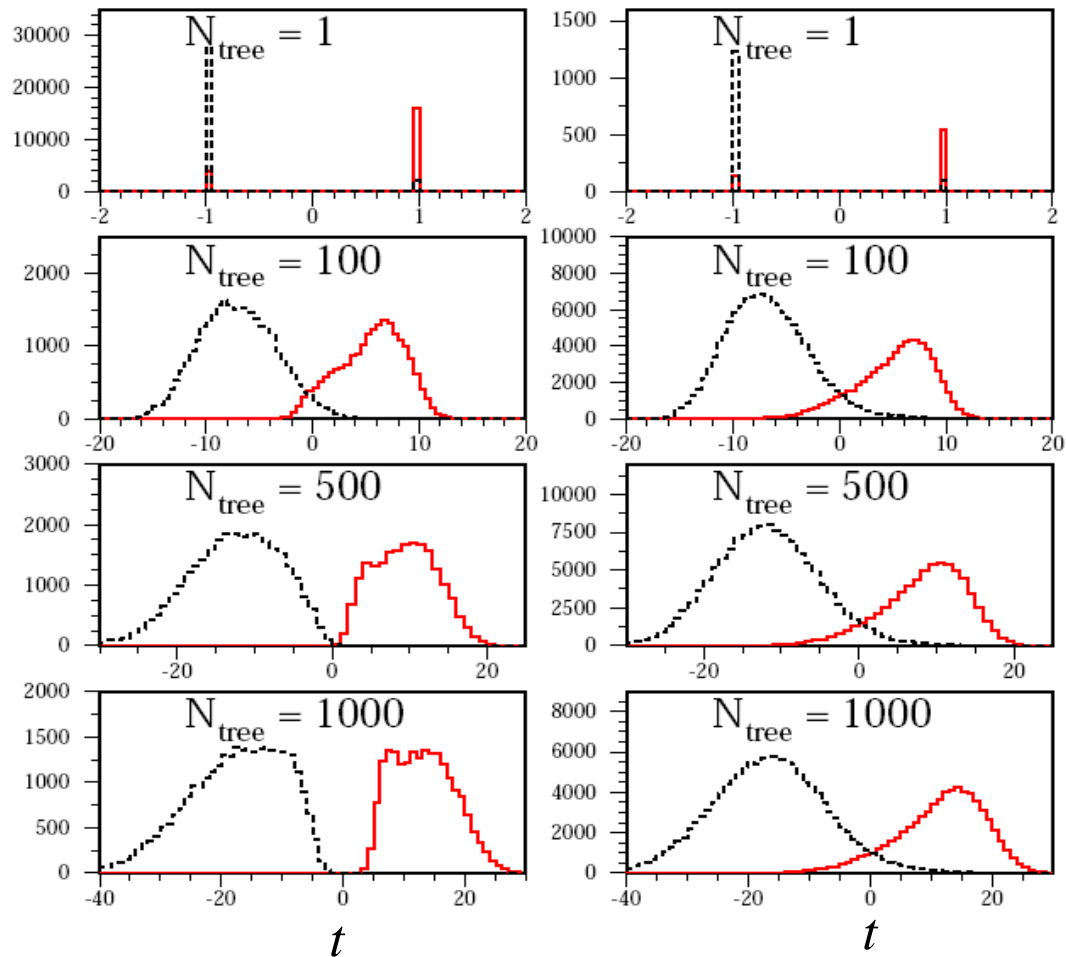
B. Roe et al., NIM 543 (2005) 577

Monitoring overtraining

From MiniBooNE
example:

Performance stable
after a few hundred
trees.

Training MC Samples .VS. Testing MC Samples



Ensemble methods

Boosting is an example of “ensemble learning”; the original training sample is “boosted” into an ensemble of samples.

Other related methods include:

Bagging (bootstrap aggregating) : the training samples are created by sampling events randomly from the original sample with replacement. In a given sample, an event might occur zero, one or multiple times.

Random forest: a type of bagging where features are randomly dropped.

More in Ch. 8 of *An Introduction to Statistical Learning with Applications in R* by James, Witten, Hastie and Tibshirani;
<https://www.statlearning.com/> see also the videos by Hastie and Tibshirani: https://youtube.com/playlist?list=PL0g0ngHtcqbPTIZzRHA2ocQZqB1D_qZ5V

Extra slides

Network training

For each of the training events we have the feature vector and true event type (class label):

$$x_a = (x_1, \dots, x_n), \quad y_a = 0, 1, \quad a = 0, \dots, N$$

We have a functional form for the decision function $t(\mathbf{x}; \mathbf{w})$ that depends on a vector of weights \mathbf{w} .

Use the training data to determine the weights by minimizing a “loss function”. Various possibilities, e.g.,

$$E(\mathbf{w}) = \frac{1}{2} \sum_{a=1}^N |t(\mathbf{x}_a, \mathbf{w}) - y_a|^2$$

quadratic loss function

$$L_{\text{CE}}(\mathbf{w}) = - \sum_{a=1}^N [y_a \log t(\mathbf{x}_a; \mathbf{w}) + (1 - y_a) \log(1 - t(\mathbf{x}_a; \mathbf{w}))]$$

cross
entropy

A simple example (2D)

Consider two variables, x_1 and x_2 , and suppose we have formulas for the joint pdfs for both signal (s) and background (b) events (in real problems the formulas are usually not available).

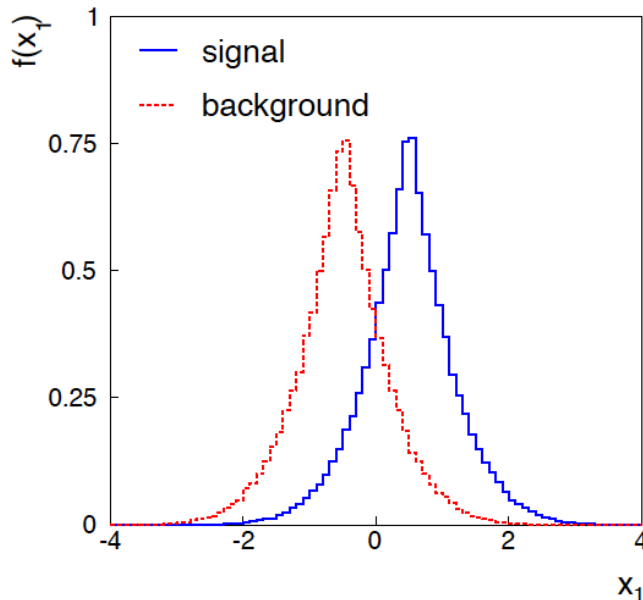
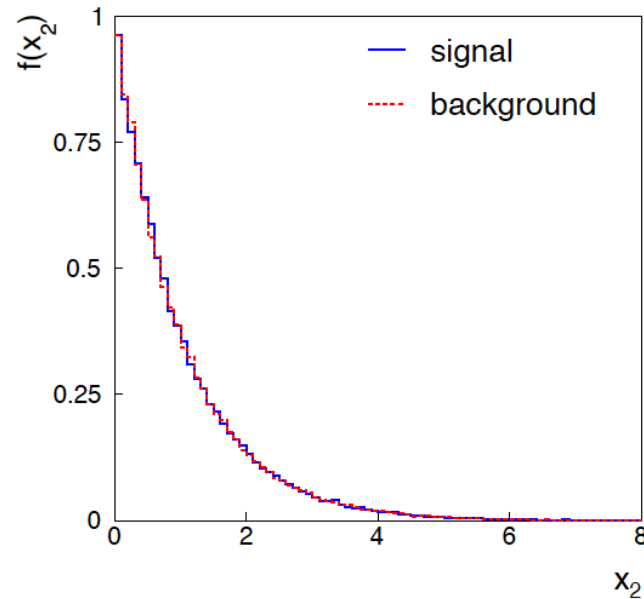
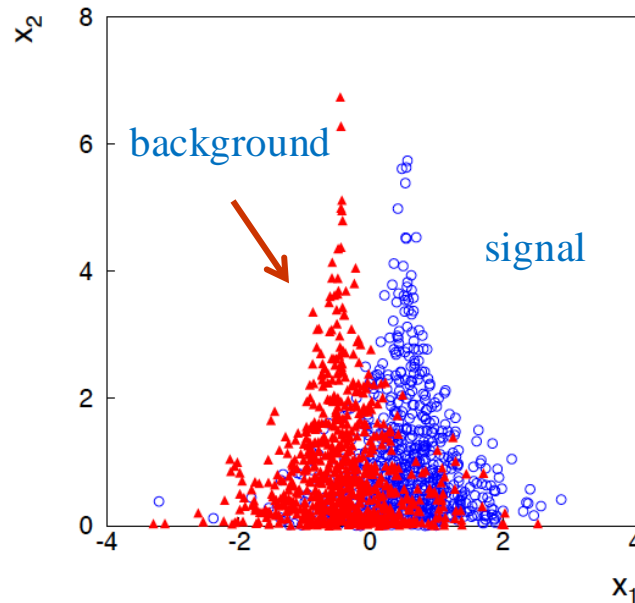
$f(x_1|x_2) \sim$ Gaussian, different means for s/b,
Gaussians have same σ , which depends on x_2 ,
 $f(x_2) \sim$ exponential, same for both s and b,
 $f(x_1, x_2) = f(x_1|x_2) f(x_2)$:

$$f(x_1, x_2|s) = \frac{1}{\sqrt{2\pi}\sigma(x_2)} e^{-(x_1 - \mu_s)^2 / 2\sigma^2(x_2)} \frac{1}{\lambda} e^{-x_2/\lambda}$$

$$f(x_1, x_2|b) = \frac{1}{\sqrt{2\pi}\sigma(x_2)} e^{-(x_1 - \mu_b)^2 / 2\sigma^2(x_2)} \frac{1}{\lambda} e^{-x_2/\lambda}$$

$$\sigma(x_2) = \sigma_0 e^{-x_2/\xi}$$

Joint and marginal distributions of x_1, x_2



Distribution $f(x_2)$ same for s, b.

So does x_2 help discriminate between the two event types?

Likelihood ratio for 2D example

Neyman-Pearson lemma says best critical region is determined by the likelihood ratio:

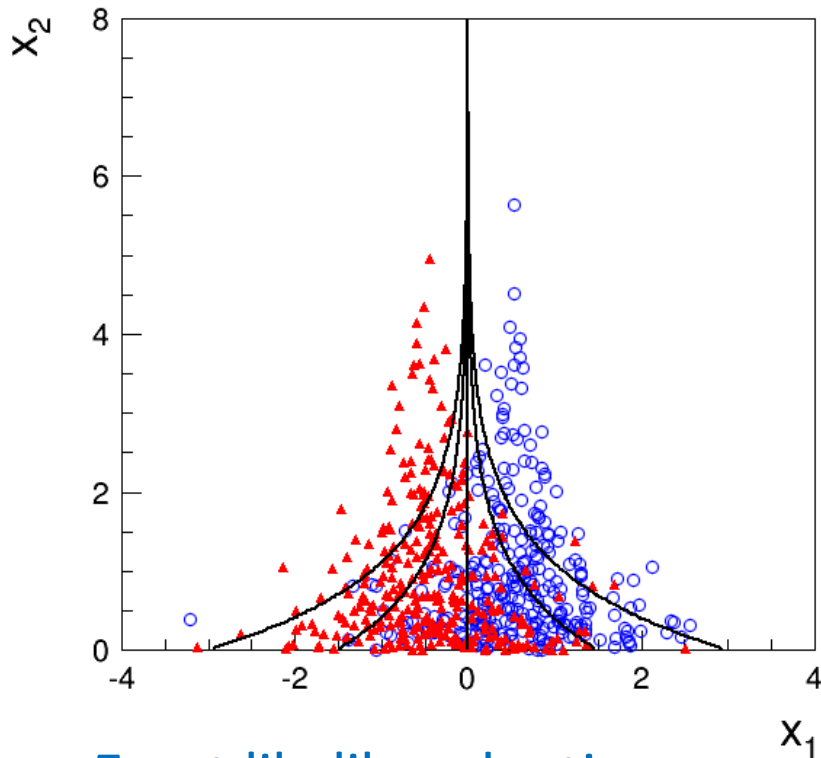
$$t(x_1, x_2) = \frac{f(x_1, x_2 | s)}{f(x_1, x_2 | b)}$$

Equivalently we can use any monotonic function of this as a test statistic, e.g.,

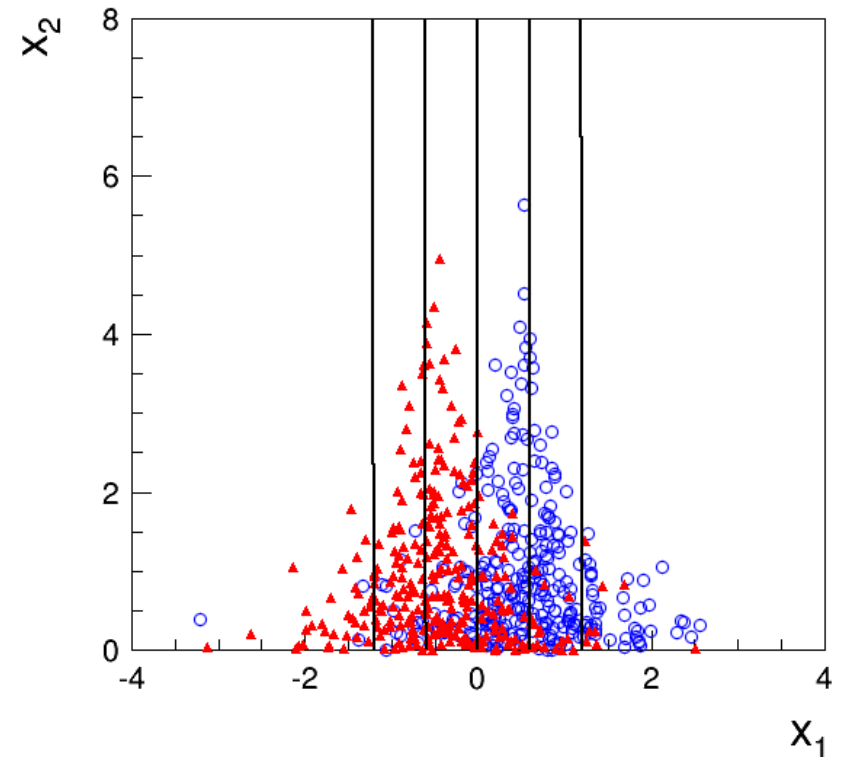
$$\ln t = \frac{\frac{1}{2}(\mu_b^2 - \mu_s^2) + (\mu_s - \mu_b)x_1}{\sigma_0^2 e^{-2x_2/\xi}}$$

Boundary of optimal critical region will be curve of constant $\ln t$, and this depends on x_2 !

Contours of constant MVA output

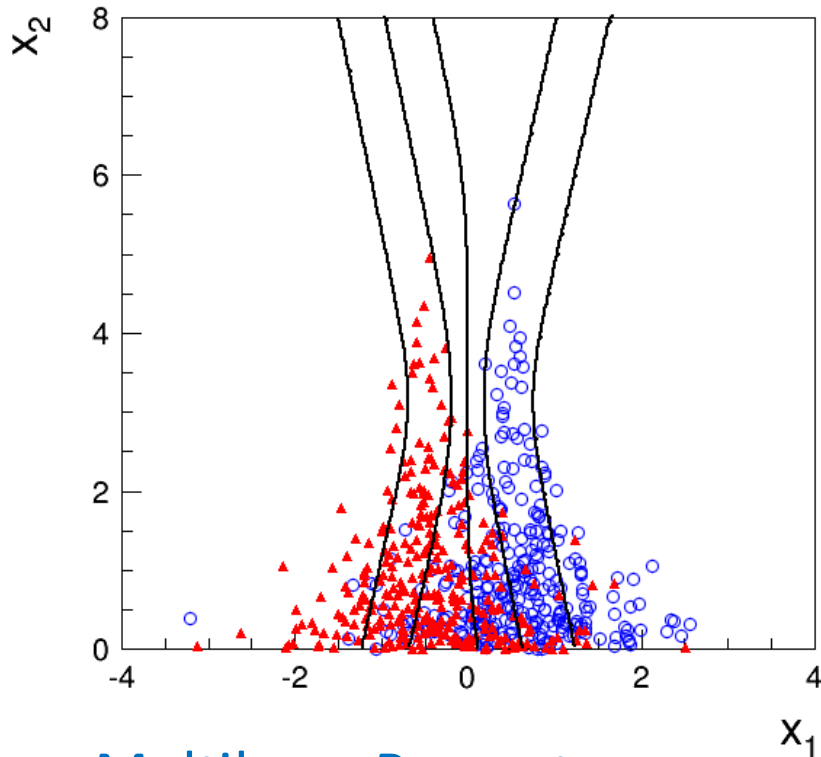


Exact likelihood ratio

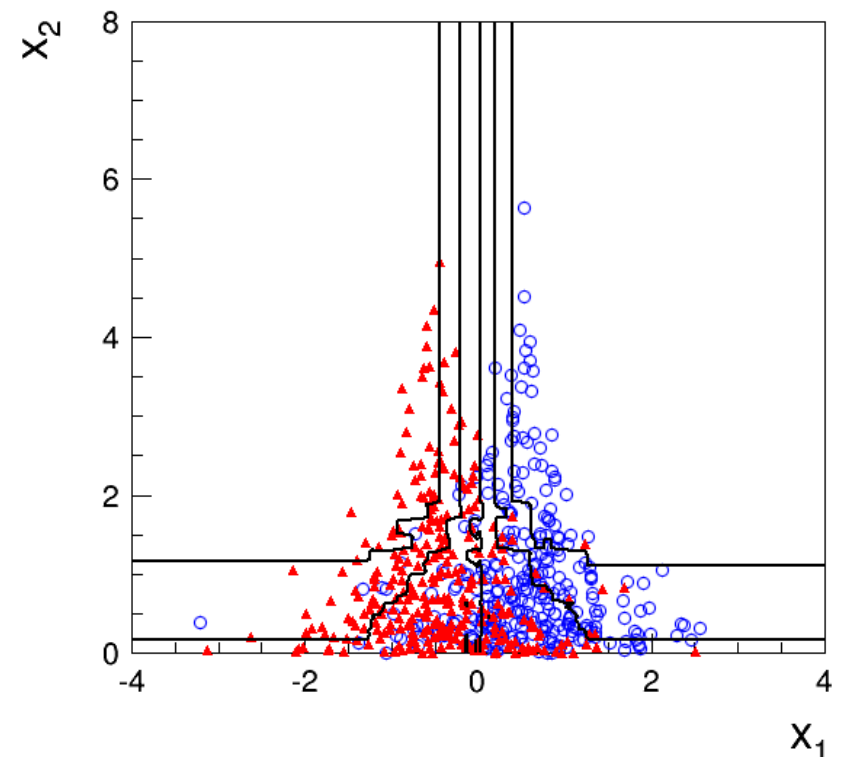


Fisher discriminant

Contours of constant MVA output



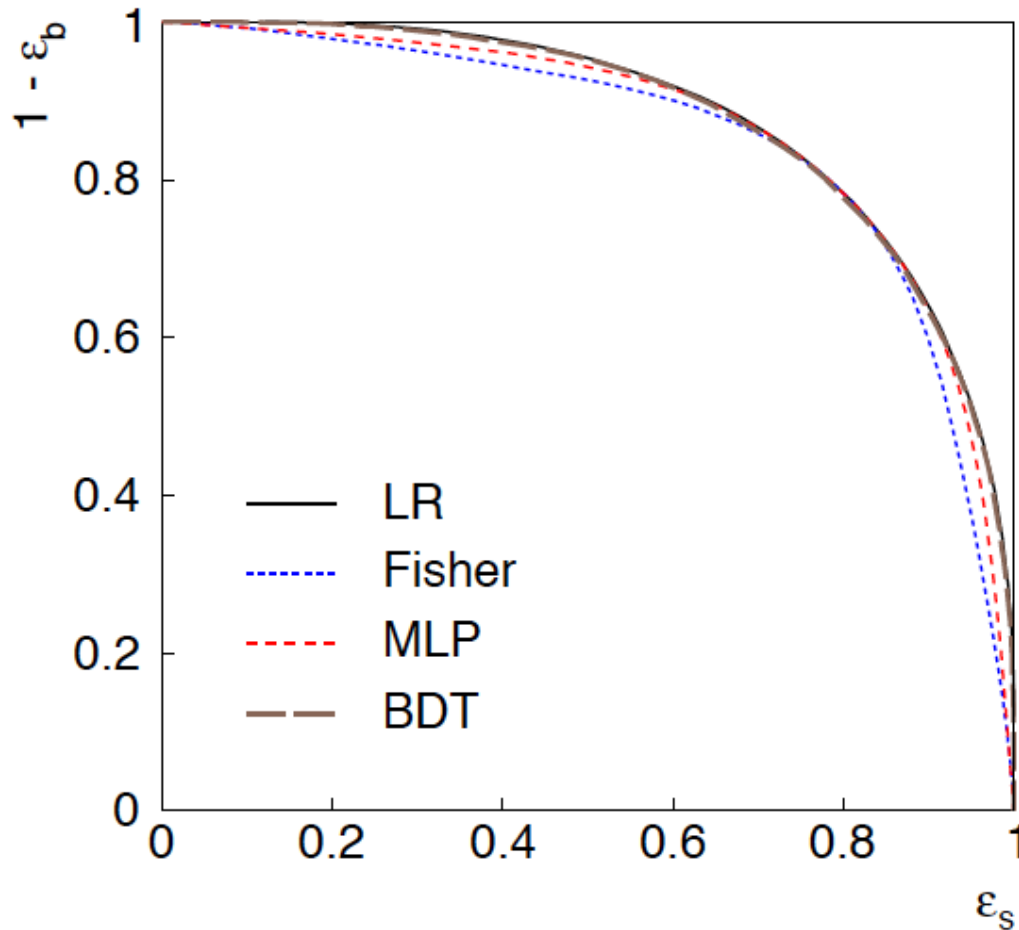
Multilayer Perceptron
1 hidden layer with 2 nodes



Boosted Decision Tree
200 iterations (AdaBoost)

Training samples: 10^5 signal and 10^5 background events

ROC curve



ROC = “receiver operating characteristic” (term from signal processing).

Shows (usually) background rejection ($1 - \epsilon_b$) versus signal efficiency ϵ_s .

Higher curve is better; usually analysis focused on a small part of the curve.

2D Example: discussion

Even though the distribution of x_2 is same for signal and background, x_1 and x_2 are not independent, so using x_2 as an input variable helps.

Here we can understand why: high values of x_2 correspond to a smaller σ for the Gaussian of x_1 . So high x_2 means that the value of x_1 was well measured.

If we don't consider x_2 , then all of the x_1 measurements are lumped together. Those with large σ (low x_2) “pollute” the well measured events with low σ (high x_2).

Often in HEP there may be variables that are characteristic of how well measured an event is (region of detector, number of pile-up vertices,...). Including these variables in a multivariate analysis preserves the information carried by the well-measured events, leading to improved performance.

Summary on multivariate methods

Particle physics has used several multivariate methods for many years:

- linear (Fisher) discriminant
- neural networks
- naive Bayes

and has in recent years started to use a few more:

- boosted decision trees
- support vector machines
- kernel density estimation
- k -nearest neighbour

The emphasis is often on controlling systematic uncertainties between the modeled training data and Nature to avoid false discovery.

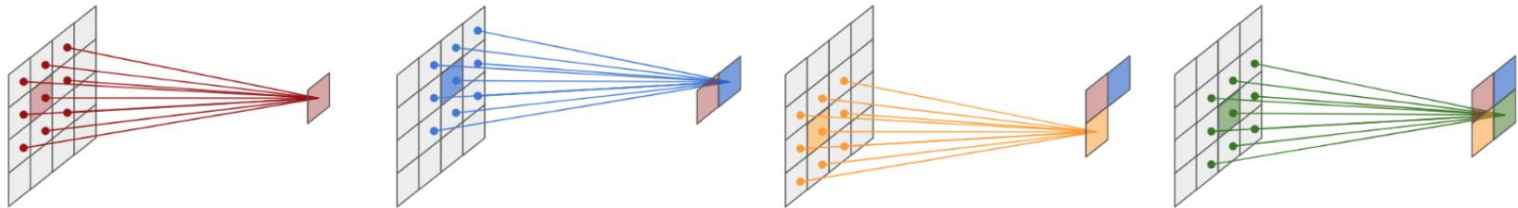
Although many classifier outputs are "black boxes", a discovery at 5σ significance with a sophisticated (opaque) method will win the competition if backed up by, say, 4σ evidence from a cut-based method.

Convolutional Neural Networks

Designed for image data (pixels) \rightarrow number of input variables $\gtrsim 10^6$.

Intermediate layers include “convolutions” of an area in previous layer, i.e., transformed pixel is a linear combination of pixels in local neighborhood in previous layer

\rightarrow far fewer connections than a fully connected MLP.



CNNs widely used for image classification.

Recurrent Neural Networks

Designed for sequential data (time series).

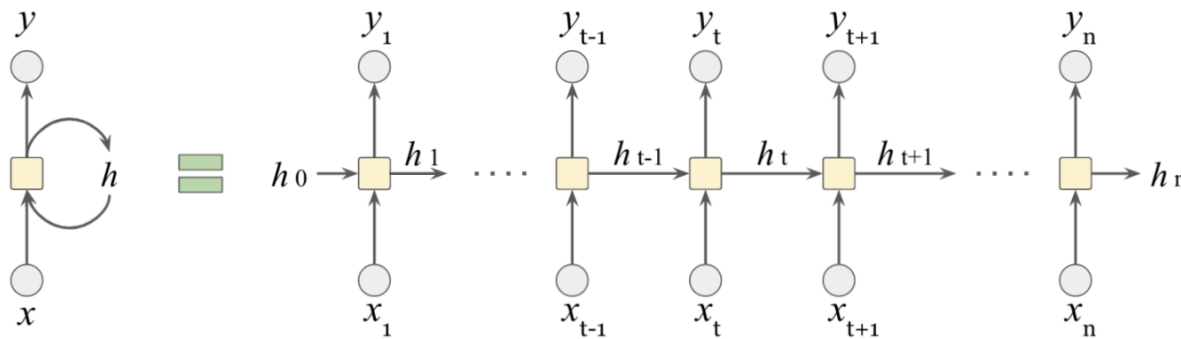


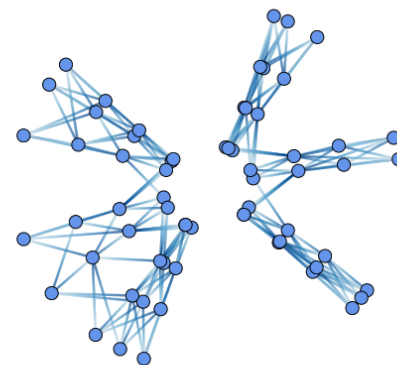
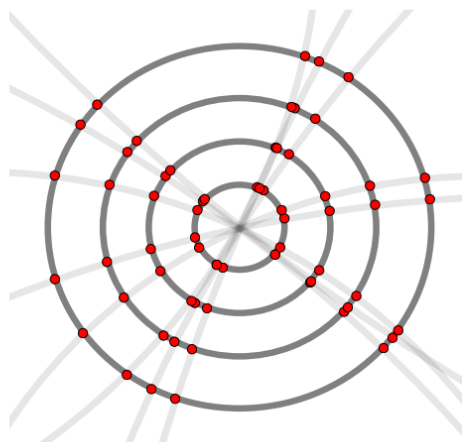
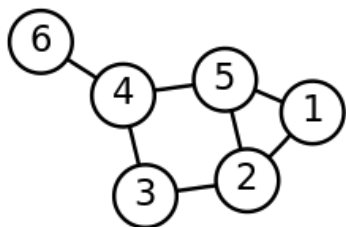
Figure 41.6: Pictorial description of a RNN (on the left) which takes an input and produces an output at every step with a hidden-to-hidden connection. The right diagram is unrolled over discrete steps. The yellow box represents a cell: a set of operations unique to each architecture.

RNNs used, e.g., in natural language processing.

Graph Neural Networks

GNNs work with graph-structured input data, e.g., signals from particles in tracking detector:

Graph = set of nodes plus set of edges:



Part of a larger field called “geometric deep learning”:

CNN is a type of GNN, graph relates pixel to its neighbors.

Transformer is a GNN that uses a mechanism called “attention”, used in natural language processing (T of ChatGPT).