

Statistical Methods in Particle Physics

10. Machine Learning

Heidelberg University, WS 2023/24

Klaus Reygers, Martin Völkl (lectures)
Ulrich Schmidt (tutorials)



You

Which topics should be covered in a introductory lecture on machine learning?



ChatGPT

An introductory lecture on machine learning should cover fundamental concepts and provide a broad overview of the field. Here are key topics that you might consider covering:

1. Introduction to Machine Learning:

- Definition of machine learning.
- Distinction between traditional programming and machine learning.
- Overview of real-world applications.

2. Types of Machine Learning:

- Supervised learning, unsupervised learning, and reinforcement learning.
- Explanation of each type with examples.

3. Supervised Learning:

- Definition and examples.
- Key components: input features, labels, and the learning process.
- Common algorithms: linear regression, logistic regression, decision trees, and support vector machines.

4. Unsupervised Learning:

- Definition and examples.
- Key components: clustering and dimensionality reduction.
- Common algorithms: k-means clustering, hierarchical clustering, and principal component analysis (PCA).

5. Reinforcement Learning:

- Definition and examples.
- Key components: agents, environments, and rewards.
- Common algorithms: Q-learning and deep reinforcement learning.

6. Introduction to Neural Networks:

- Basic structure of artificial neural networks.
- Neurons, layers, and activation functions.
- Feedforward and backpropagation concepts.

7. Deep Learning:

- Explanation of deep neural networks.
- Convolutional Neural Networks (CNNs) for image processing.
- Recurrent Neural Networks (RNNs) for sequential data.

8. Evaluation Metrics:

- Accuracy, precision, recall, F1 score, and confusion matrix.
- Choosing appropriate metrics based on the problem.

9. Feature Engineering:

- Importance of selecting relevant features.
- Techniques for feature selection and extraction.

10. Model Training and Evaluation:

- Splitting data into training and testing sets.
- Cross-validation.
- Overfitting and underfitting.

11. Challenges and Considerations:

- Bias and fairness in machine learning.
- Ethical considerations.
- Limitations and challenges in real-world applications.

12. Practical Examples and Demos:

- Showcase simple examples or demos to illustrate key concepts.
- Demonstrate how to implement a basic machine learning model using a popular framework.

13. Resources for Further Learning:

- Point to relevant books, online courses, and resources for those interested in diving deeper into machine learning.

Multivariate analysis: An early example from particle physics

G. Cowan, Lecture on Statistical data analysis

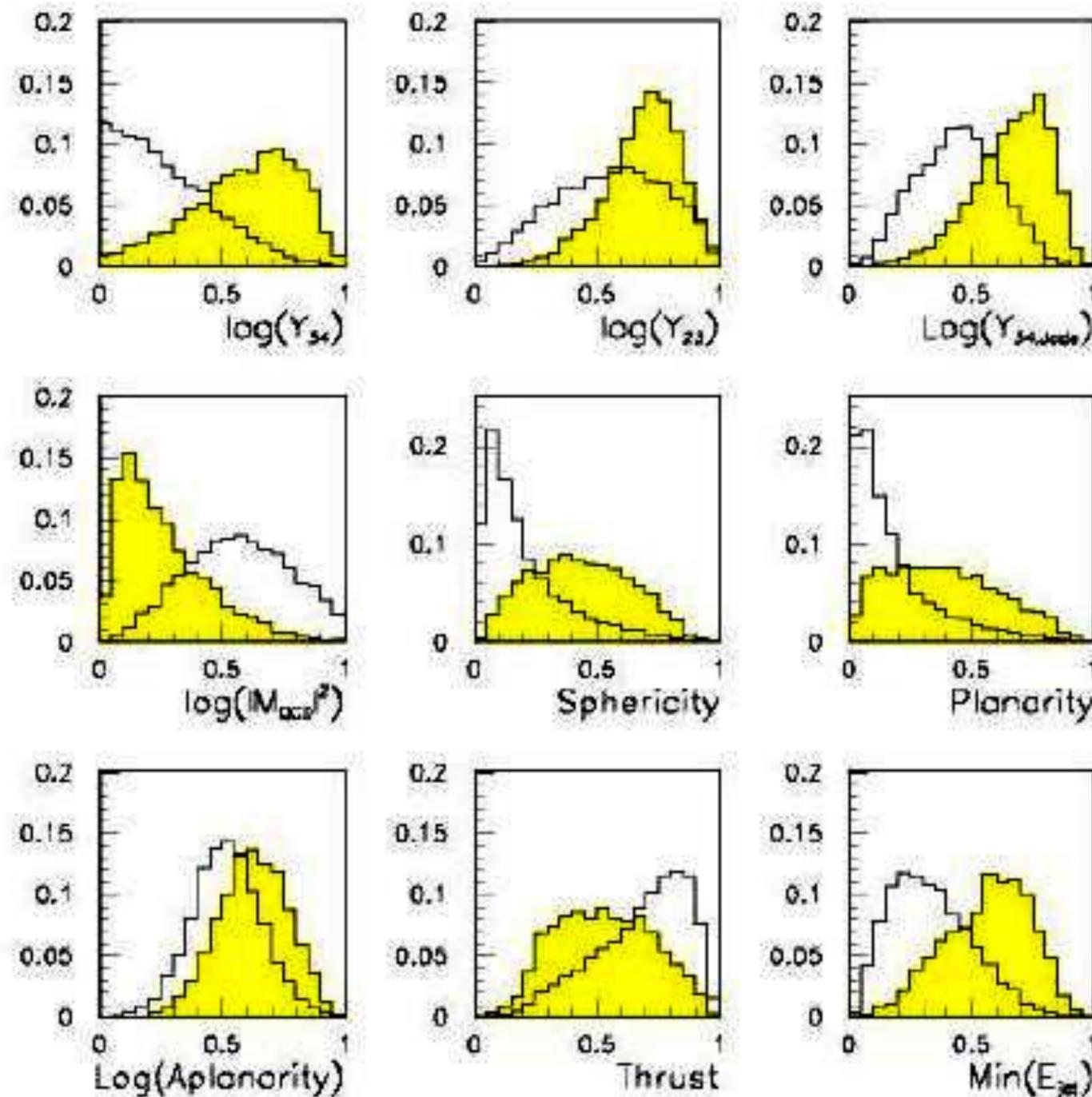
Signal: $e^+e^- \rightarrow W^+W^-$

often 4 well separated hadron jets

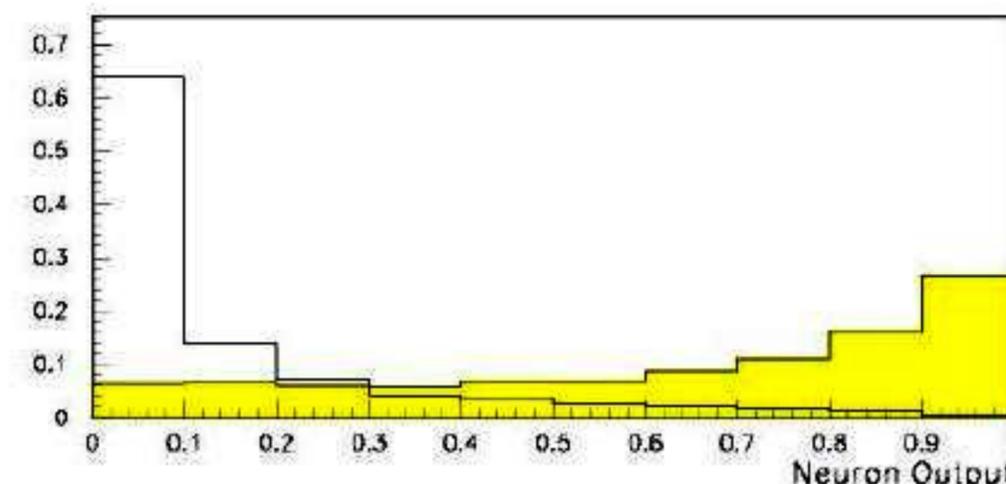
Background: $e^+e^- \rightarrow q\bar{q}g\bar{g}$

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)

Machine learning

"Machine learning is the subfield of computer science that gives computers the ability to learn without being explicitly programmed" – Wikipedia

Example: spam detection

J. Mayes, Machine learning 101

<p>Write a computer program with explicit rules to follow</p> <pre>if email contains V!agrå then mark is-spam; if email contains ... if email contains ...</pre>	<p>Write a computer program to learn from examples</p> <pre>try to classify some emails; change self to reduce errors; repeat;</pre>
Traditional Programming	Machine Learning Programs

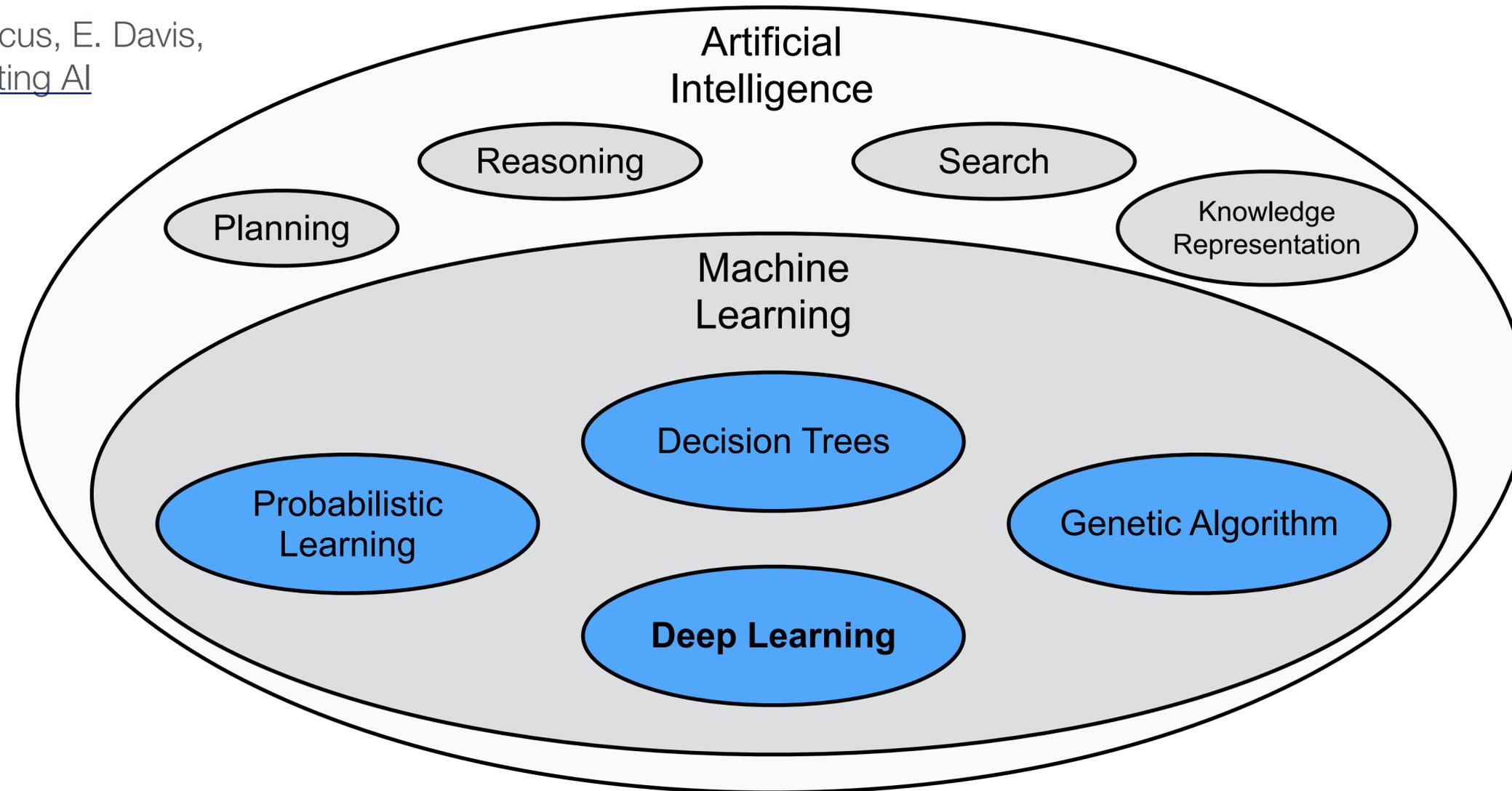
Manual feature engineering vs. automatic feature detection

AI, ML, and DL

"AI is the study of how to make computers perform things that, at the moment, people do better."

Elaine Rich, Artificial intelligence, McGraw-Hill 1983

G. Marcus, E. Davis,
[Rebooting AI](#)



"deep" in deep learning: artificial neural nets with many neurons and multiple layers of nonlinear processing units for feature extraction

Some successes and unsolved problems in AI

Arithmetic (1945)	}	Easy
Sorting lists of numbers (1959)		
Playing simple board games (1959)	}	Solved, after a lot of effort
Playing chess (1997)		
Recognizing faces in pictures (2008)		
Usable automated translation (2010)		
Playing Go (2016)	}	Real progress
Usable real-time translation of spoken words (2016)		
Driverless cars	}	Nowhere near solved
Automatically providing captions for pictures		
Understanding a story & answering questions about it	}	
Human-level automated translation		
Interpreting what is going on in a photograph		
Writing interesting stories		
Interpreting a work of art	}	
Human-level general intelligence		

M. Woolridge,
[The Road to Conscious Machines](#)

Impressive progress in certain fields:

- ▶ Image recognition
- ▶ Speech recognition
- ▶ Recommendation systems
- ▶ Automated translation
- ▶ Chatbots based on Large Language Models (LLMs)
- ▶ Analysis of medical data

How can we profit from these developments in physics?

Different modeling approaches

- Simple mathematical representation like linear regression. Favored by statisticians.
- Complex deterministic models based on scientific understanding of the physical process. Favored by physicists.
- Complex algorithms to make predictions that are derived from a huge number of past examples (“machine learning” as developed in the field of computer science). These are often black boxes.
- Regression models that claim to reach causal conclusions. Used by economists.

D. Spiegelhalter, The Art of Statistics – Learning from data

Application of machine learning in experimental particle physics

- Event reconstruction and particle identification
- Monte Carlo simulation
 - ▶ use generative models for faster MC event generation
- Data acquisition / trigger
 - ▶ faster algorithms
- Offline data analysis
 - ▶ better algorithms
- Detector monitoring
 - ▶ anomaly detection

“Machine Learning in High Energy Physics Community White Paper”,
[arXiv:1807.02876](https://arxiv.org/abs/1807.02876)

Machine learning: The "hello world" problem

Recognition of handwritten digits

- ▶ MNIST database
(Modified National Institute of Standards and Technology database)
- ▶ 60,000 training images and 10,000 testing images labeled with correct answer
- ▶ 28 pixel x 28 pixel
- ▶ Algorithms have reached "near-human performance"
- ▶ Smallest error rate (2018): 0.18%



https://en.wikipedia.org/wiki/MNIST_database

Play with MNIST data set and Keras (Stefan Wunsch, CERN IML Workshop):

https://github.com/stwunsch/iml_tensorflow_keras_workshop

Machine learning: Image recognition

ImageNet database

- ▶ 14 million images, 22,000 categories
- ▶ Since 2010, the annual ImageNet Large Scale Visual Recognition Challenge (ILSVRC):
1.4 million images, 1000 categories
- ▶ In 2017, 29 of 38 competing teams got less than 5% wrong

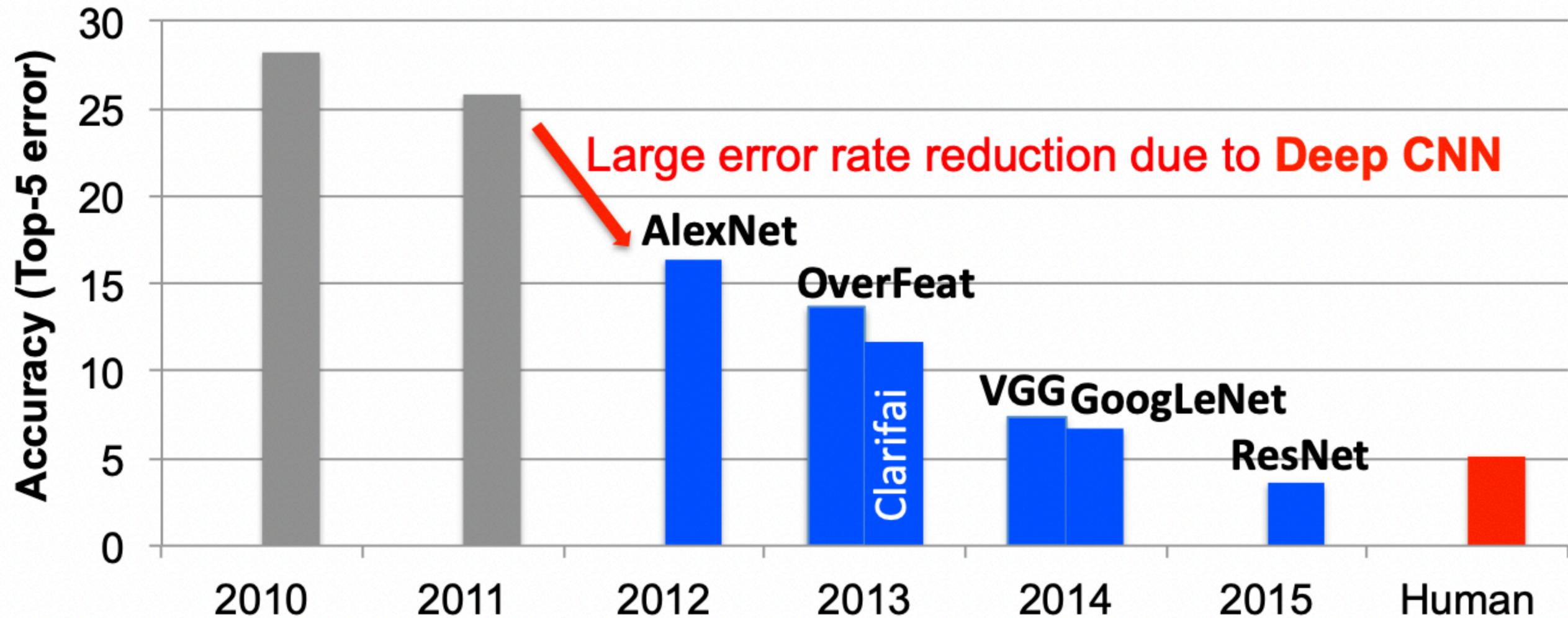
<https://en.wikipedia.org/wiki/ImageNet>



https://www.tensorflow.org/tutorials/image_recognition

ImageNet: Large Scale Visual Recognition Challenge

Error rate in percent:



O. Russakovsky et al, arXiv:1409.0575
figure from <https://arxiv.org/pdf/1703.09039.pdf>

Adversarial examples

Ian J. Goodfellow, Jonathon Shlens, Christian Szegedy, arXiv:1412.6572v1



\mathbf{x}

“panda”

57.7% confidence

+ .007 ×

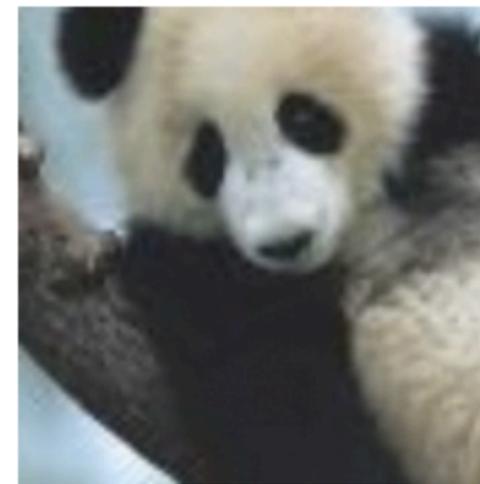


$\text{sign}(\nabla_{\mathbf{x}} J(\boldsymbol{\theta}, \mathbf{x}, y))$

“nematode”

8.2% confidence

=



$\mathbf{x} + \epsilon \text{sign}(\nabla_{\mathbf{x}} J(\boldsymbol{\theta}, \mathbf{x}, y))$

“gibbon”

99.3 % confidence

Three types of learning

LeCun 2018, Power And Limits of Deep Learning, <https://www.youtube.com/watch?v=0tEhw5t6rhc>

Reinforcement learning

- ▶ The machine ("the agent") predicts a scalar reward given once in a while
- ▶ Weak feedback



arXiv:1312.5602

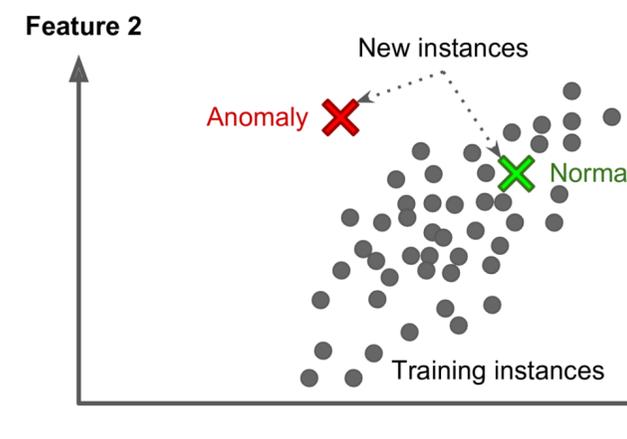
Supervised learning

- ▶ The machine predicts a category based on labeled training data
- ▶ Medium feedback



Unsupervised learning

- ▶ Describe/find hidden structure from "unlabeled" data
- ▶ Cluster data in different sub-groups with similar properties



Aurélien Géron,
Hands-On Machine
Learning with Scikit-
Learn and TensorFlow

Example:
anomaly detection

Learning resources

- Ian Goodfellow and Yoshua Bengio and Aaron Courville, ***Deep Learning***, free online <http://www.deeplearningbook.org/>
- Andreas Lindholm, Niklas Wahlström, Fredrik Lindsten, and Thomas B. Schön, ***Machine Learning - A First Course for Engineers and Scientists***, draft version available: <http://smlbook.org/>
- Simon J.D. Prince, ***Understanding Deep Learning***, draft version available: <https://udlbook.github.io/udlbook/>
- Aurélien Géron, ***Hands-On Machine Learning with Scikit-Learn and TensorFlow***
- Kevin Patrick Murphy, ***Probabilistic Machine Learning***, <https://probml.github.io/pml-book/>
- ***A Living Review of Machine Learning for Particle Physics***, <https://github.com/iml-wg/HEPML-LivingReview>
- Tilman Plehn, Anja Butter, Barry Dillon, Claudius Krause, ***Modern Machine Learning for LHC Physicists***, <https://arxiv.org/abs/2211.01421>

Useful libraries

- scikit-learn, <https://scikit-learn.org/>
- PyTorch, <https://pytorch.org/>
- TensorFlow, <https://www.tensorflow.org/>
- XGBoost, <https://xgboost.ai/>



Multivariate classification

Consider events which can be either signal or background events.

Each event is characterized by n observables:

$$\vec{x} = (x_1, \dots, x_n) \quad \text{"feature vector"}$$

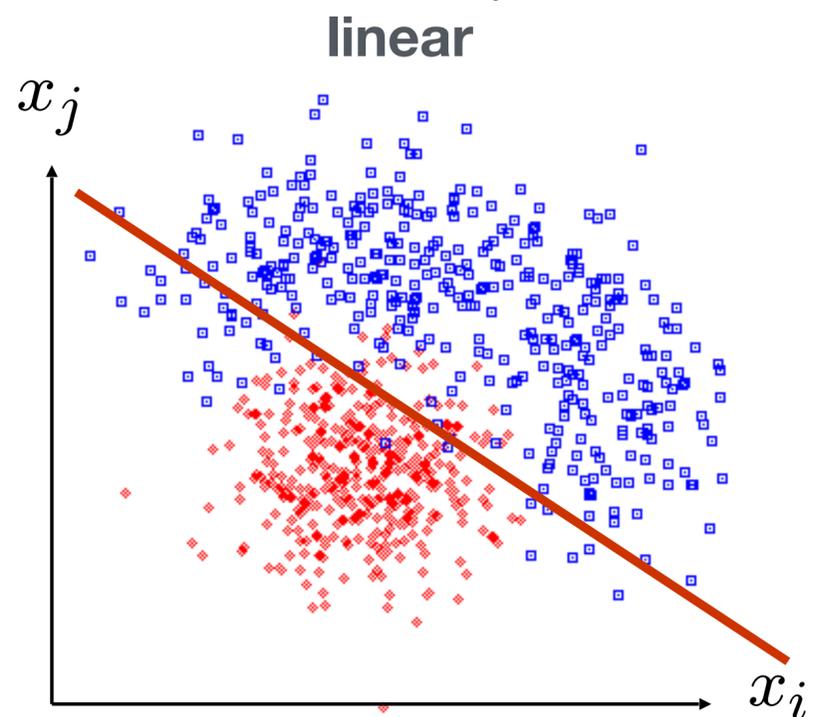
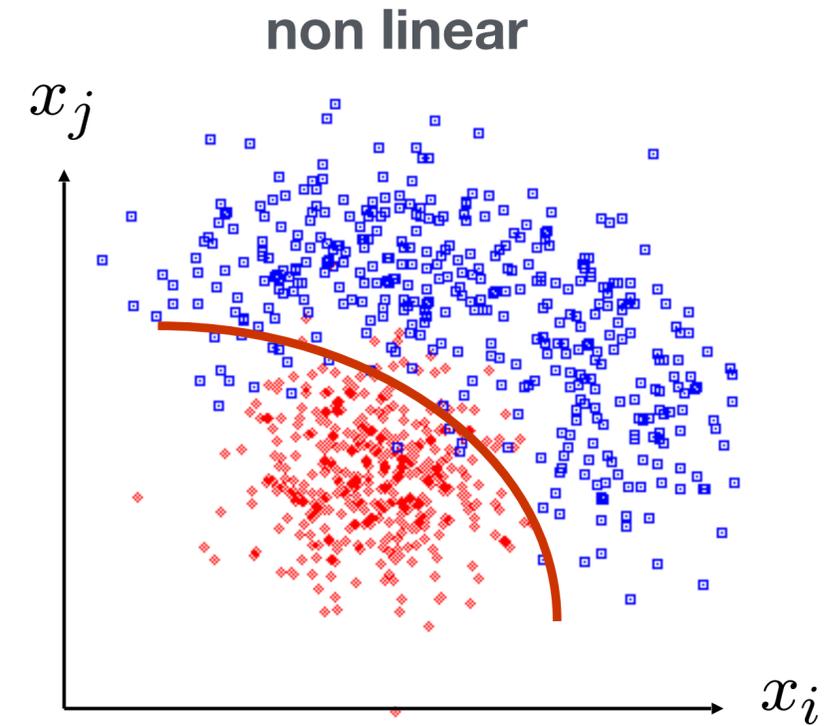
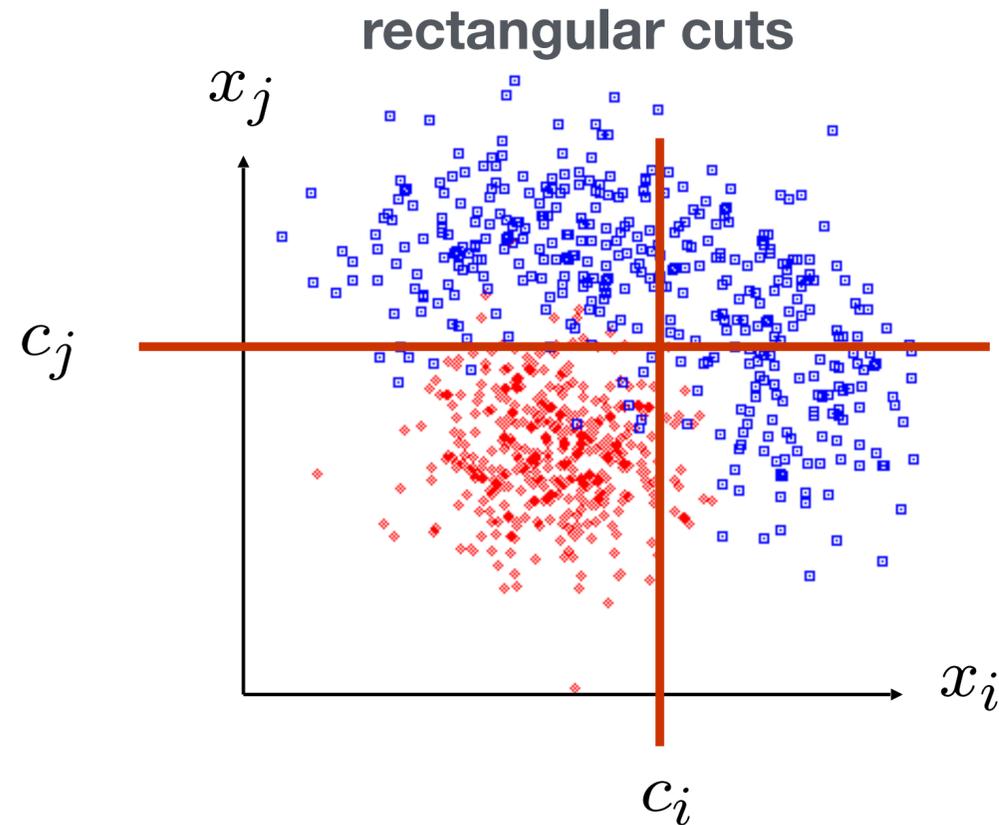
Goal: classify events as signal or background in an optimal way.

This is usually done by mapping the feature vector to a single variable, i.e., to scalar "test statistic":

$$\mathbb{R}^n \rightarrow \mathbb{R} : y(\vec{x})$$

A cut $y > c$ to classify events as signal corresponds to selecting a potentially complicated hyper-surface in feature space. In general superior to classical "rectangular" cuts on the x_i .

Classification: Learning decision boundaries



k -Nearest-Neighbor,
Boosted Decision Trees,
Multi-Layer Perceptrons,
Support Vector Machines

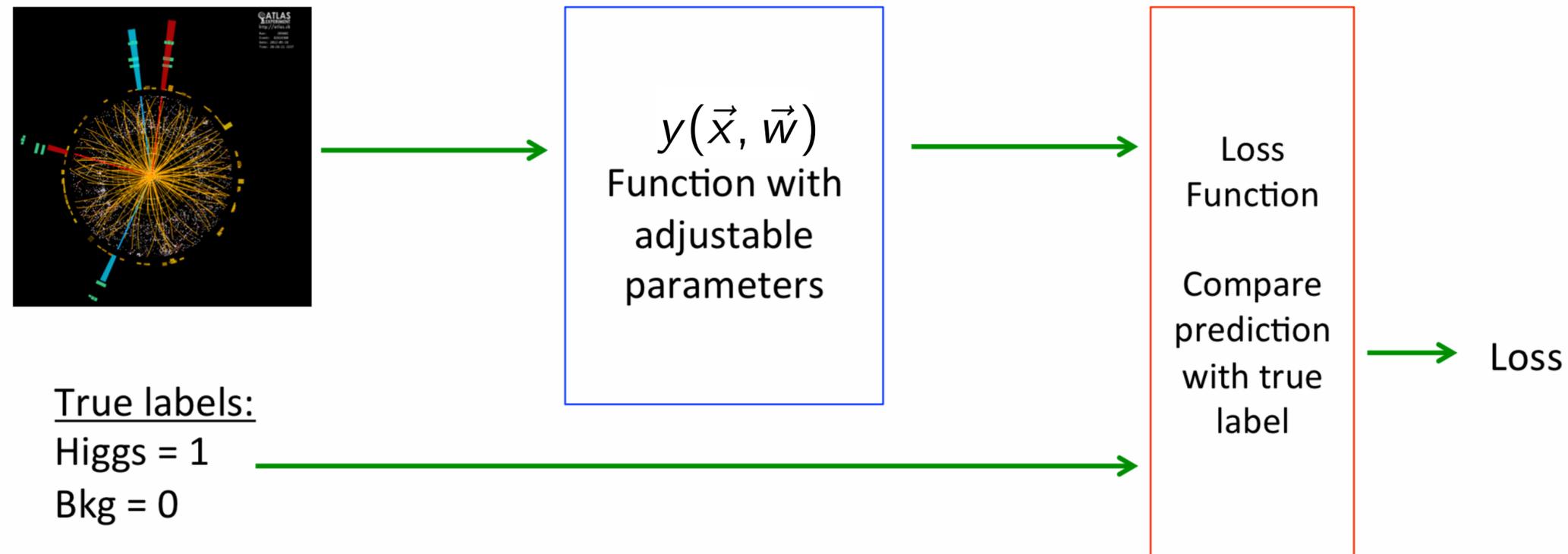
...

G. Cowan:

https://www.pp.rhul.ac.uk/~cowan/stat_course.html

Supervised learning in a nutshell

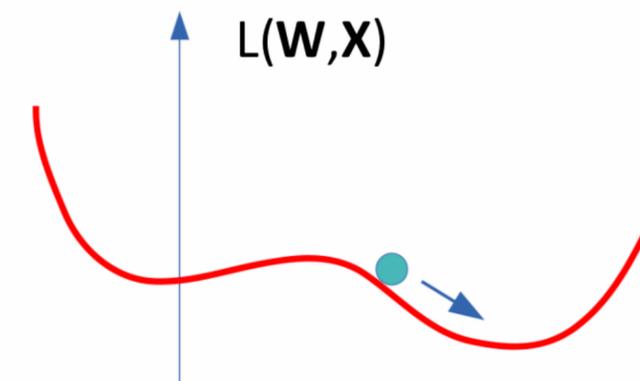
Supervised Machine Learning requires *labeled training data*, i.e., a training sample where for each event it is known whether it is a signal or background event



Design function $y(\vec{x}, \vec{w})$ with adjustable parameters \vec{w}

Design a loss function

Find best parameters which minimize loss



Supervised learning: classification and regression

The codomain Y of the function $y: X \rightarrow Y$ can be a set of labels or classes or a continuous domain, e.g., \mathbb{R}

Binary classification: $Y = \{0, 1\}$ e.g., signal or background

Multi-class classification: $Y = \{c_1, c_2, \dots, c_n\}$

Labels sometimes represented as "**one-hot vector**"
(no ordering btw. labels):

$$t_a = \{0, 0, \dots, 1, \dots, 0\}$$

$Y =$ finite set of labels \rightarrow classification

$Y =$ real numbers \rightarrow regression

"All the impressive achievements of deep learning amount to just curve fitting"

J. Pearl, Turing Award Winner 2011,

<https://www.quantamagazine.org/to-build-truly-intelligent-machines-teach-them-cause-and-effect-20180515/>

Supervised learning: Training, validation, and test sample

- Decision boundary fixed with training sample
- Performance on training sample becomes better with more iterations
- Danger of overtraining:
Statistical fluctuations of the training sample will be learnt
- **Validation sample** = independent labeled data set not used for training
→ check for overtraining
- Sign of overtraining: performance on validation sample becomes worse
→ Stop training when signs of overtraining are observed ("early stopping")
- Determine performance using **performance metric** (such as accuracy, precision, recall, etc.):
apply classifier to independent **test sample**
- Often: test sample = validation sample (only small bias)

Supervised learning: Cross validation

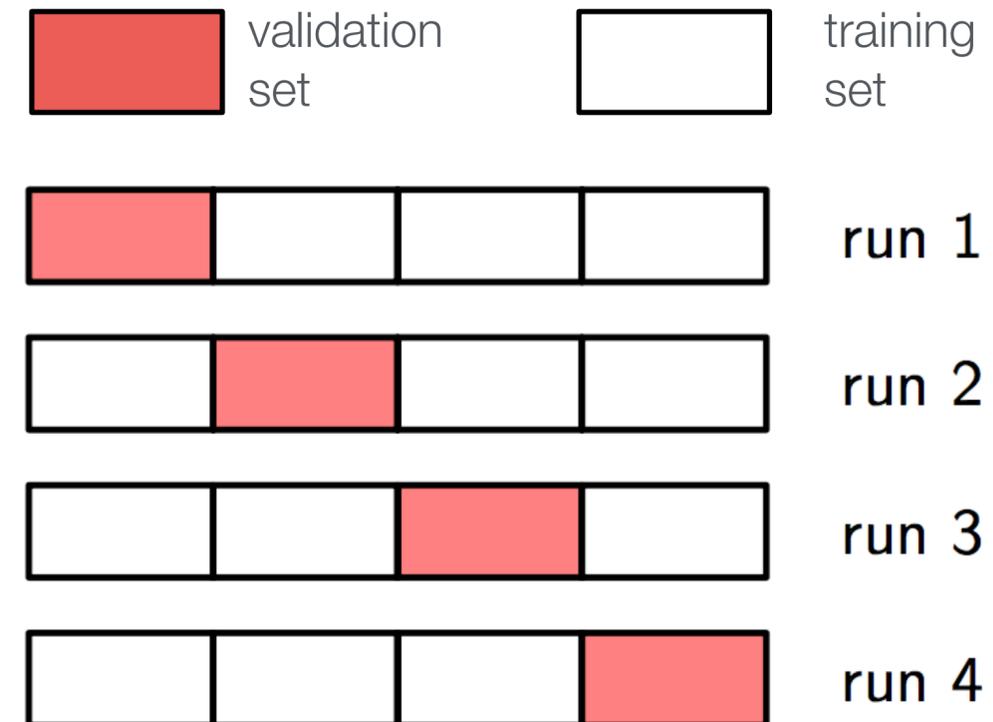
Rule of thumb if training data not expensive

- ▶ Training sample: 50%
- ▶ Validation sample: 25%
- ▶ Test sample: 25%

} often test sample = validation sample,
i.e., training : validation/test = 50:50

Cross validation (efficient use of scarce training data)

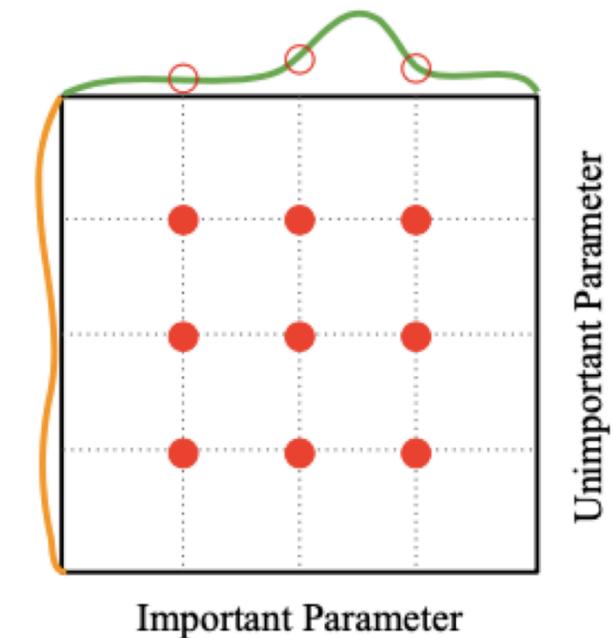
- ▶ Split training sample in k independent subset T_k of the full sample T
- ▶ Train on $T \setminus T_k$ resulting in k different classifiers
- ▶ For each training event there is one classifier that didn't use this event for training
- ▶ Validation results are then combined



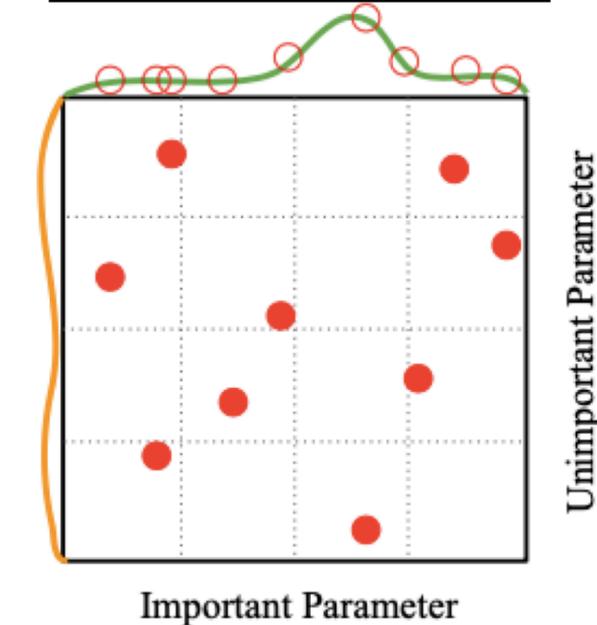
Hyperparameter optimization

- Example of hyperparameters
 - ▶ Number of layers/nodes in an ANN, learning rate, number of leaves in a decision tree, ...
- These parameters are fixed during model training, i.e., they are **not optimized with gradient descent**.
- Various approaches to hyperparameter optimization
 - ▶ Grid search
 - ▶ Random search
 - ▶ Bayesian optimization
- Tools
 - ▶ Optuna, <https://optuna.readthedocs.io>
 - ▶ scikit-optimize, <https://scikit-optimize.github.io/stable/>
 - ▶ ...

Grid Search



Random Search



More precise sampling of the important parameter in the case of random search.

Often used loss functions

Squared error loss:

- often used in regression

$$J(\vec{w}) = \frac{1}{n} \sum_{i=1}^n (y(\vec{x}_i, \vec{w}) - t_i)^2$$

value predicted by the model

true value

Cross entropy:

- consider binary classification
- $t \in \{0, 1\}$
- Often used in classification

$$J(\vec{w}) = \frac{1}{n} \sum_{i=1}^n -t_i \log y(\vec{x}_i, \vec{w}) - (1 - t_i) \log(1 - y(\vec{x}_i, \vec{w}))$$

binary cross entropy

predicted "probability" for outcome $t = 1$

$$= \frac{1}{n} \sum_{i=1}^n \begin{cases} -\log y(\vec{x}_i, \vec{w}), & \text{if } t_i = 1 \\ -\log(1 - y(\vec{x}_i, \vec{w})), & \text{if } t_i = 0 \end{cases}$$

More on entropy

Self-information of an event x : $I(x) = -\log p(x)$

- ▶ Measure of the “amount of surprise” of an outcome value x .
- ▶ If $\log \equiv \log_2$: **Shannon information measured in bits.**

Example: 32 equally likely outcomes. A given observed event contains $-\log_2(1/2^5)$ bits = 5 bits of information.

Shannon entropy: $H(P) = -\sum p_i \log p_i$

- ▶ Expected amount of information of an event drawn from a distribution P .
- ▶ Measure of the minimum of amount of bits needed on average to encode symbols drawn from a distribution
- ▶ In the above (trivial) example: $H = -\sum 1/2^5 \log_2(1/2^5) = 5$ bits

Cross entropy: $H(P, Q) = -E[\log q_i] = -\sum p_i \log q_i$

- ▶ Can be interpreted as a measure of the amount of bits needed when a wrong distribution Q is assumed while the data actually follows a distribution P
- ▶ Measure of dissimilarity between distributions P and Q (i.e, a measure of how well the model Q describes the true distribution P)

Logistic Regression for Binary Classification (1)

Logistic regression can be viewed as a modification of linear regression. The output of the model is number in $[0, 1]$ which is usual interpreted as a probability.

First, a score z is calculated from the feature vector \vec{x} :

$$z = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_p x_p = \vec{w}^T \vec{x}$$

The score z is mapped to the interval $[0, 1]$ using the *logistic function* $h(z) = \frac{e^z}{1 + e^z}$:

$$y(\vec{x}; \vec{w}) = \frac{e^{\vec{w}^T \vec{x}}}{1 + e^{\vec{w}^T \vec{x}}}$$

We can then find the optimal parameters \vec{w} with the aid of the **maximum likelihood method**. Consider a training data set $\{\vec{x}_i, y_i\}_{i=1}^n$ where $y_i \in \{0, 1\}$. The optimal parameters are then given by:

$$\hat{\vec{w}} = \operatorname{argmax}_{\vec{w}} \sum_{i=1}^n \log p(y_i | \vec{x}_i; \vec{w})$$

Logistic Regression for Binary Classification (2)

In our model $p(y = 1 | \vec{x}, \vec{w}) \equiv y(\vec{x}, \vec{w})$ and accordingly $p(y = 0 | \vec{x}, \vec{w}) = 1 - y(\vec{x}, \vec{w})$.

The log-likelihood therefore reads:

$$\log p(y_i | \vec{x}_i; \vec{w}) = \begin{cases} \log y(\vec{x}_i, \vec{w}) & \text{if } y_i = 1 \\ \log(1 - y(\vec{x}_i, \vec{w})) & \text{if } y_i = 0 \end{cases}$$

So we need to maximize

$$\frac{1}{n} \sum_{i=1}^n \begin{cases} \log y(\vec{x}_i, \vec{w}), & \text{if } y_i = 1 \\ \log(1 - y(\vec{x}_i, \vec{w})), & \text{if } y_i = 0 \end{cases}$$

Maximizing the log-likelihood therefore is nothing else but minimizing the cross entropy loss function.

This optimization problem has no closed-form solution and needs to be done numerically.

Multinomial logistic regression: Softmax function

In the previous example we considered two classes, 0 and 1. For multi-class classification, the logistic function can be generalized to the softmax function.

Consider K classes and let z_i be the score for class i , $\vec{z} = (z_1, \dots, z_K)$

A probability for class i can be predicted with the softmax function:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \quad \text{for } i = 1, \dots, K$$

The softmax function is often used as the last activation function of a neural network in order to predict probabilities in a classification task.

Multinomial logistic regression is also known as softmax regression.

Simple example of logistic regression with scikit-learn (1)

Read data

<https://scikit-learn.org>

Data are from the [wikipedia article on logistic regression](#)



```
# data: 1. hours studied, 2. passed (0/1)
filename = "data/exam.txt"
df = pd.read_csv(filename, engine='python', sep='\s+')
```

```
x_tmp = df['hours_studied'].values
x = np.reshape(x_tmp, (-1, 1))
y = df['passed'].values
```

Fit the model

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(penalty='none', fit_intercept=True)
clf.fit(x, y);
```

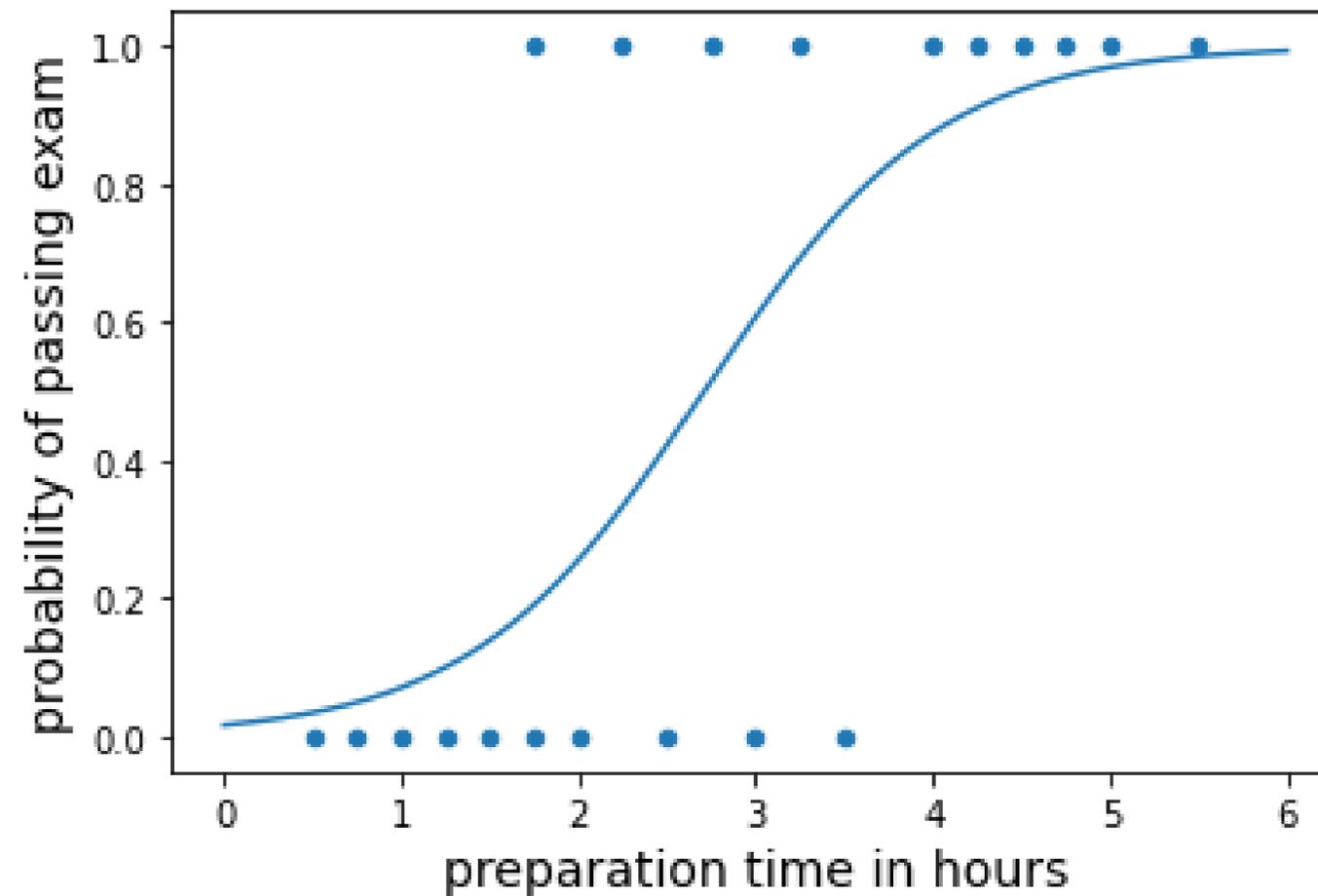
Calculate predictions

```
hours_studied_tmp = np.linspace(0., 6., 1000)
hours_studied = np.reshape(hours_studied_tmp, (-1, 1))
y_pred = clf.predict_proba(hours_studied)
```

Simple example of logistic regression with scikit-learn (2)

Plot result

```
df.plot.scatter(x='hours_studied', y='passed')  
plt.plot(hours_studied, y_pred[:,1])  
plt.xlabel("preparation time in hours", fontsize=14)  
plt.ylabel("probability of passing exam", fontsize=14)  
plt.savefig("logistic_regression.pdf")
```



Reminder: Neyman–Pearson lemma

The likelihood ratio

$$t(\vec{x}) = \frac{f(\vec{x}|H_1)}{f(\vec{x}|H_0)} \quad \begin{array}{l} H_1 : \text{signal hypothesis} \\ H_0 : \text{background hypothesis} \end{array}$$

is an optimal test statistic, i.e., it provides highest "signal efficiency" $1 - \beta$ for a given "background efficiency" α .

Accept hypothesis if

$$t(\vec{x}) = \frac{f(\vec{x}|H_1)}{f(\vec{x}|H_0)} > c$$

Problem: the underlying pdf's are almost never known explicitly.

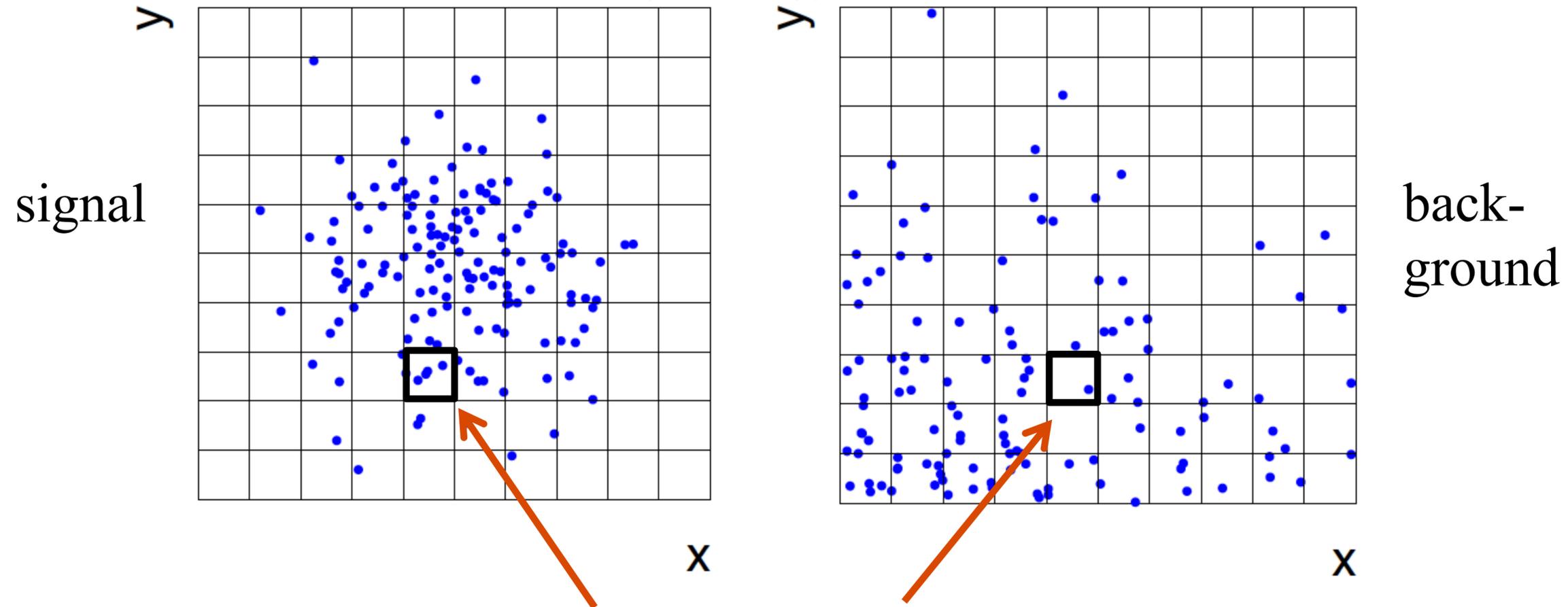
Two approaches:

- 1.** Estimate signal and background pdf's and construct test statistic based on Neyman-Pearson lemma
- 2.** Decision boundaries determined directly without approximating the pdf's (linear discriminants, decision trees, neural networks, ...)

Estimating PDFs from histograms?

Consider 2d example:

G. Cowan: https://www.pp.rhul.ac.uk/~cowan/stat_course.html



approximate PDF by $N(x, y|S)$ and $N(x, y|B)$

M bins per variable in d dimensions: M^d cells

→ hard to generate enough training data (often not practical for $d > 1$)

In general in machine learning, problems related to a large number of dimensions of the feature space are referred to as the "**curse of dimensionality**"

Naïve Bayesian classifier (also called "projected likelihood classification")

Application of the Neyman-Pearson lemma (ignoring correlations between the x_i):

$$f(x_1, x_2, \dots, x_n) \text{ approximated as } L = f_1(x_1) \cdot f_2(x_2) \cdot \dots \cdot f_n(x_n)$$

$$\text{where } f_1(x_1) = \int dx_2 dx_3 \dots dx_n f(x_1, x_2, \dots, x_n)$$

$$f_2(x_2) = \int dx_1 dx_3 \dots dx_n f(x_1, x_2, \dots, x_n)$$

⋮

In practice, the $f_k(x_k)$ **are often obtained from the 1d histograms** of the feature k of the feature training feature vectors. Some smoothing is typically applied by fitting a suitable function to the 1d histogram.

Classification of feature vector \vec{x} :

$$y(\vec{x}) = \frac{L_s(\vec{x})}{L_s(\vec{x}) + L_b(\vec{x})} = \frac{1}{1 + L_b(\vec{x})/L_s(\vec{x})}$$

Performance not optimal if true PDF does not factorize

k -nearest neighbor method (1)

k -NN classifier

- ▶ Estimates probability density around the input vector
- ▶ $p(\vec{x} | \mathcal{S})$ and $p(\vec{x} | \mathcal{B})$ are approximated by the number of signal and background events in the training sample that lie in a small volume around the point \vec{x}

Algorithm finds k nearest neighbors:

$$k = k_s + k_b$$

Probability for the event to be of signal type:

$$p_s(\vec{x}) = \frac{k_s(\vec{x})}{k_s(\vec{x}) + k_b(\vec{x})}$$

k -NN is an example of a **nonparametric model** (method only uses training data to make predictions).

By contrast, parametric models learn parameters from data.

k-nearest neighbor method (2)

Simplest choice for distance measure in feature space is the Euclidean distance:

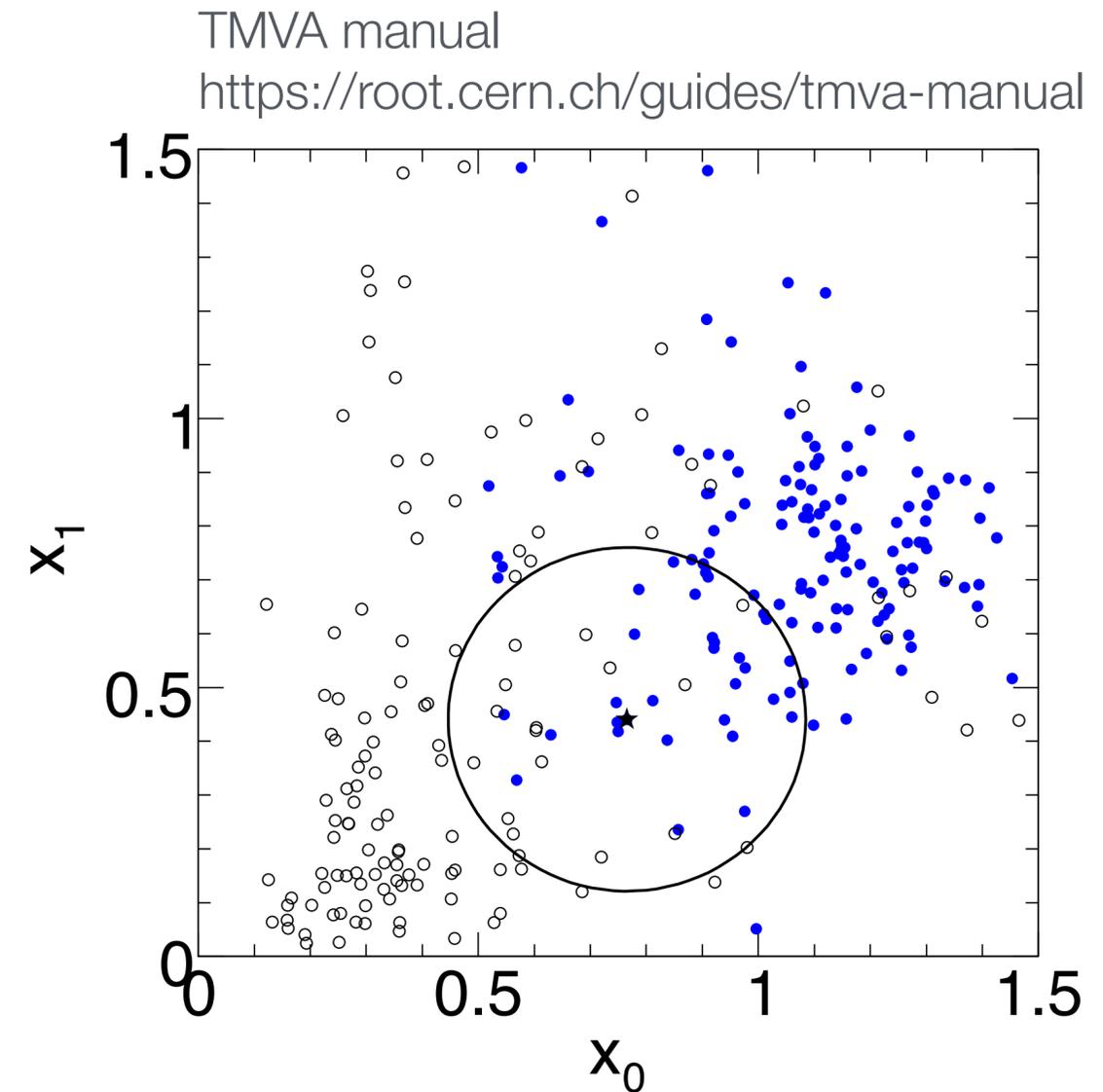
$$R = |\vec{x} - \vec{y}|$$

Better: take correlations between variables into account:

$$R = \sqrt{(\vec{x} - \vec{y})^T V^{-1} (\vec{x} - \vec{y})}$$

V = covariance matrix

"Mahalanobis distance"



The k -NN classifier has best performance when the boundary that separates signal and background events has irregular features that cannot be easily approximated by parametric learning methods.

Fisher linear discriminant

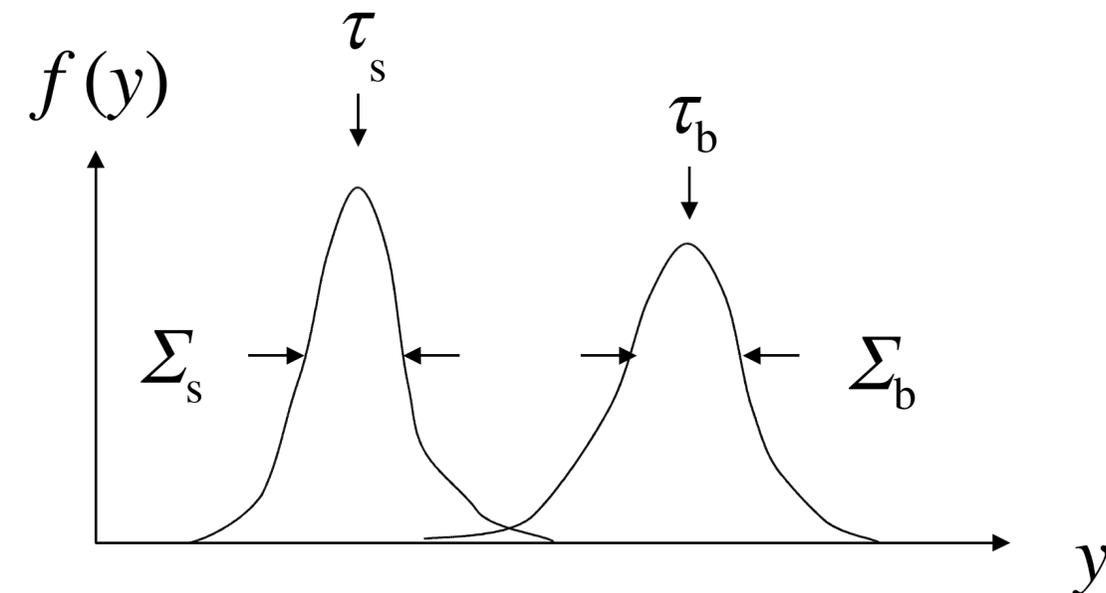
Linear discriminant is simple. Can still be optimal if amount of training data is limited.

Ansatz for test statistic: $y(\vec{x}) = \sum_{i=1}^n w_i x_i = \vec{w}^T \vec{x}$

Choose parameters w_i so that separation between signal and background distribution is maximum.

Need to define "separation".

Fisher: maximize $J(\vec{w}) = \frac{(\tau_s - \tau_b)^2}{\Sigma_s^2 + \Sigma_b^2}$



G. Cowan:

https://www.pp.rhul.ac.uk/~cowan/stat_course.html

Fisher linear discriminant: Variable definitions

Mean and covariance for signal and background:

$$\mu_i^{s,b} = \int x_i f(\vec{x}|H_{s,b}) d\vec{x}$$

$$V_{ij}^{s,b} = \int (x_i - \mu_i^{s,b})(x_j - \mu_j^{s,b}) f(\vec{x}|H_{s,b}) d\vec{x}$$

Mean and variance of $y(\vec{x})$ for signal and background:

$$\tau_{s,b} = \int y(\vec{x}) f(\vec{x}|H_{s,b}) d\vec{x} = \vec{w}^T \vec{\mu}_{s,b}$$

$$\Sigma_{s,b}^2 = \int (y(\vec{x}) - \tau_{s,b})^2 f(\vec{x}|H_{s,b}) d\vec{x} = \vec{w}^T V_{s,b} \vec{w}$$

G. Cowan':

https://www.pp.rhul.ac.uk/~cowan/stat_course.html

Fisher linear discriminant: Determining the coefficients w_i

Numerator of $J(\vec{w})$:

$$\begin{aligned}(\tau_s - \tau_b)^2 &= \left(\sum_{i=1}^n w_i (\mu_i^s - \mu_i^b) \right)^2 = \sum_{i,j=1}^n w_i w_j (\mu_i^s - \mu_i^b) (\mu_j^s - \mu_j^b) \\ &\equiv \sum_{i,j=1}^n w_i w_j B_{ij} = \vec{w}^T B \vec{w}\end{aligned}$$

Denominator of $J(\vec{w})$:

$$\Sigma_s^2 + \Sigma_b^2 = \sum_{i,j=1}^n w_i w_j (V^s + V^b)_{ij} \equiv \vec{w}^T W \vec{w}$$

Maximize:

$$J(\vec{w}) = \frac{\vec{w}^T B \vec{w}}{\vec{w}^T W \vec{w}} = \frac{\text{separation between classes}}{\text{separation within classes}}$$

G. Cowan':

https://www.pp.rhul.ac.uk/~cowan/stat_course.html

Fisher linear discriminant: Determining the coefficients w_i

Setting $\frac{\partial J}{\partial w_i} = 0$ gives:

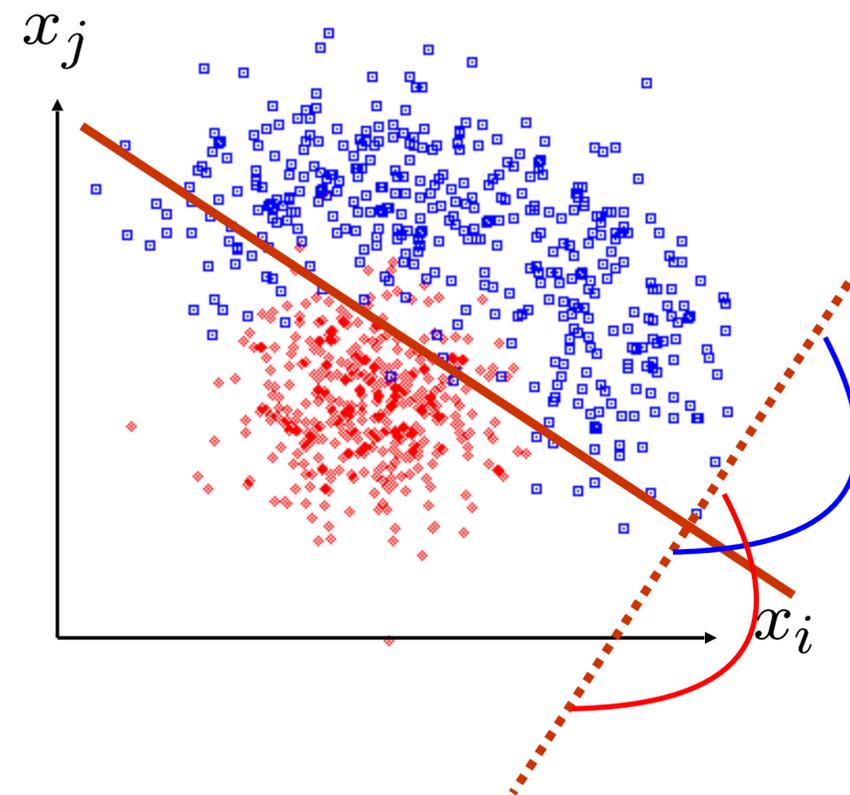
$$y(\vec{x}) = \vec{w}^T \vec{x} \quad \text{with} \quad \vec{w} \propto W^{-1}(\vec{\mu}_s - \vec{\mu}_b)$$

We obtain linear decision boundaries.

Weight vector \vec{w} can be interpreted as a direction in feature space on which the events are projected.

G. Cowan':
https://www.pp.rhul.ac.uk/~cowan/stat_course.html

linear decision boundary



Fisher linear discriminant: Remarks

In case the signal and background pdfs $f(\vec{x}|H_s)$ and $f(\vec{x}|H_b)$ are both multivariate Gaussian with the same covariance but different means, the Fisher discriminant is

$$y(\vec{x}) \propto \ln \frac{f(\vec{x}|H_s)}{f(\vec{x}|H_b)}$$

That is, in this case the Fisher discriminant is an optimal classifier according to the Neyman-Pearson lemma (as $y(\vec{x})$ is a monotonic function of the likelihood ratio)

Test statistic can be written as

$$y(\vec{x}) = w_0 + \sum_{i=1}^n w_i x_i$$

where events with $y > 0$ are classified as signal. Same functional form as for the **perceptron** (prototype of neural networks).

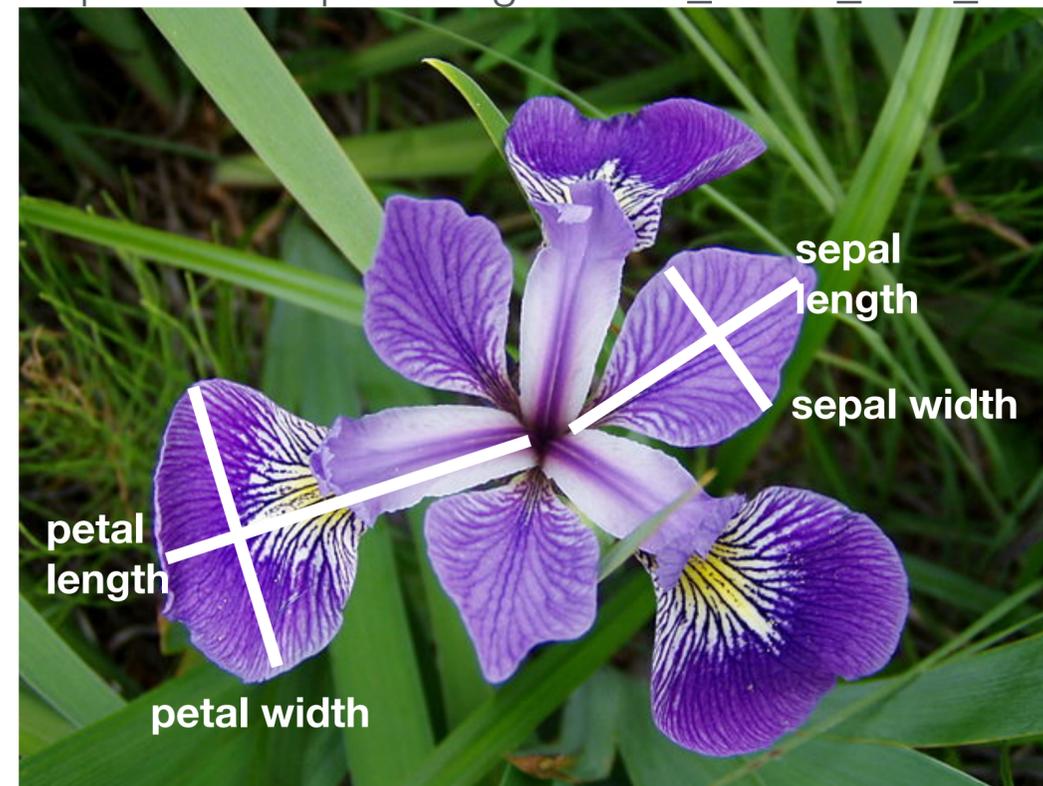
Example: Classification with scikit-learn (1)

Iris flower data set

- ▶ Introduced 1936 in a paper by Ronald Fisher
- ▶ Task: classify flowers
- ▶ Three species: iris setosa, iris virginica and iris versicolor
- ▶ Four features: petal width and length, sepal width/length, in centimeters

<https://archive.ics.uci.edu/ml/datasets/Iris>

https://en.wikipedia.org/wiki/Iris_flower_data_set



Example: Classification with scikit-learn (2)



```
# import some data to play with
# columns: Sepal Length, Sepal Width, Petal Length and Petal Width
iris = datasets.load_iris()
X = iris.data
y = iris.target
```

```
: # just to create a nice table
df = pd.DataFrame({"Sepal Length (cm)": X[:,0], "Sepal Width (cm)": X[:,1],
                  "Petal Length (cm)": X[:,2], "Petal Width (cm)": X[:,3],
                  "category": y})
df.head()
```

```
:
```

	Sepal Length (cm)	Sepal Width (cm)	Petal Length (cm)	Petal Width (cm)	category
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

```
list(iris.target_names)
```

```
['setosa', 'versicolor', 'virginica']
```

```
# split data into training and test data sets
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=42)
```

Example: Classification with scikit-learn (3)



Softmax regression

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression(multi_class='multinomial', penalty='none')
log_reg.fit(x_train, y_train);
```

k-nearest neighbor

```
from sklearn.neighbors import KNeighborsClassifier
kn_neigh = KNeighborsClassifier(n_neighbors=5)
kn_neigh.fit(x_train, y_train);
```

Fisher linear discriminant

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
fisher_ld = LinearDiscriminantAnalysis()
fisher_ld.fit(x_train, y_train);
```

Classification accuracy

```
for clf in [log_reg, kn_neigh, fisher_ld]:
    y_pred = clf.predict(x_test)
    acc = accuracy_score(y_test, y_pred)
    print(type(clf).__name__)
    print(f"accuracy: {acc:0.2f}")

# confusion matrix: columns: true class, row: predicted class
print(confusion_matrix(y_test, y_pred), "\n")
```

Output:

```
LogisticRegression
accuracy: 0.96
[[29  0  0]
 [ 0 23  0]
 [ 0  3 20]]
```

```
KNeighborsClassifier
accuracy: 0.95
[[29  0  0]
 [ 0 23  0]
 [ 0  4 19]]
```

```
LinearDiscriminantAnalysis
accuracy: 0.99
[[29  0  0]
 [ 0 23  0]
 [ 0  1 22]]
```

With scikit-learn it is extremely simple to test and apply different classification methods

Examples of metrics: Accuracy, precision and recall

Precision:

Fraction of correctly classified instances among all instances that obtain a certain class label:

$$\text{precision} = \frac{TP}{TP + FP}$$

“purity”

Accuracy:

Fraction of correct predictions:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

Iris classification example:
precision and recall for softmax classification
see [sklearn.metrics.classification_report](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.classification_report.html)

Recall:

Fraction of positive instances that are correctly classified.

$$\text{recall} = \frac{TP}{TP + FN}$$

“efficiency”

TP: true positives
FP: false positives
FN: false negatives

$$F_1 = \frac{2 \cdot \text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$$

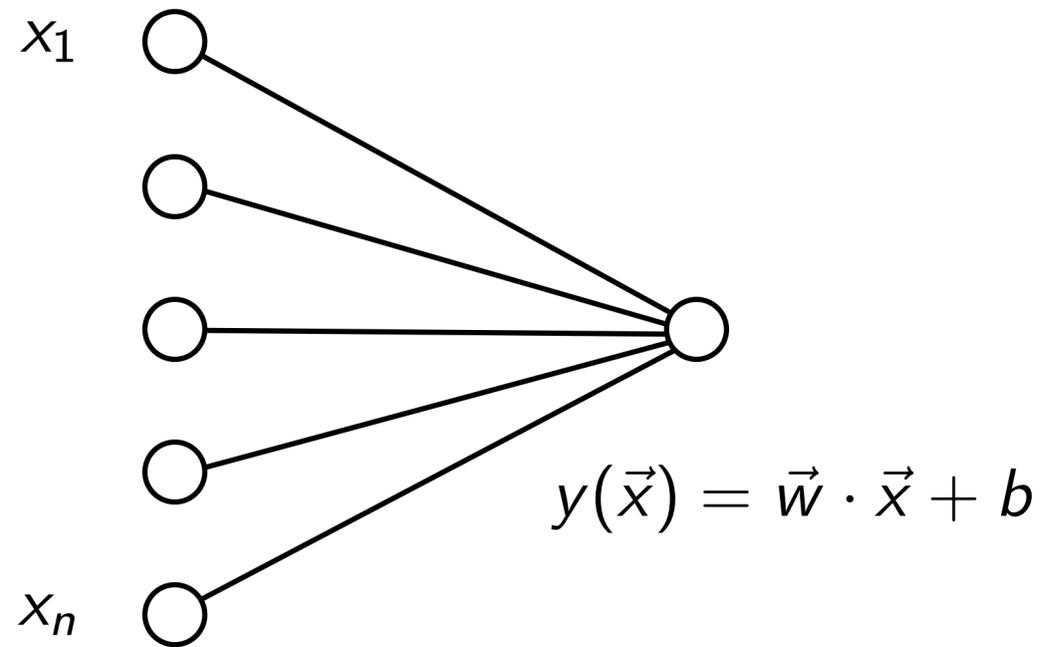
```
y_pred = log_reg.predict(x_test)
print(classification_report(y_test, y_pred))
```

	precision	recall	f1-score	support
0	1.00	1.00	1.00	29
1	0.88	1.00	0.94	23
2	1.00	0.87	0.93	23
accuracy			0.96	75
macro avg	0.96	0.96	0.96	75
weighted avg	0.96	0.96	0.96	75

https://en.wikipedia.org/wiki/Precision_and_recall

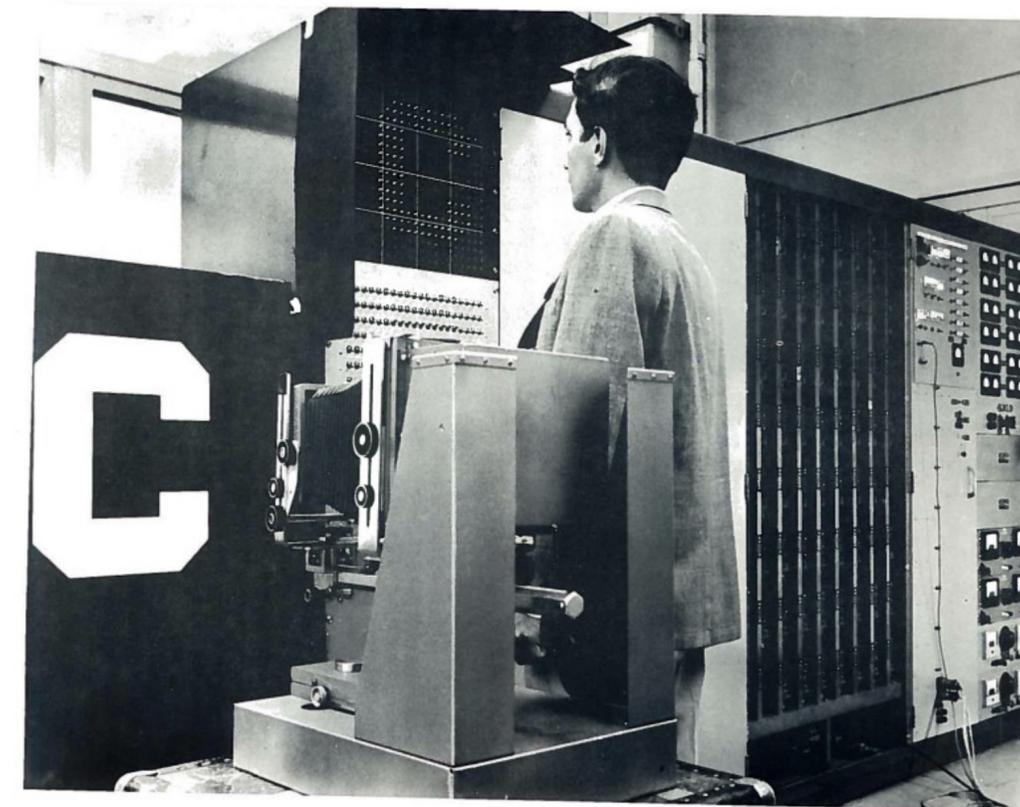
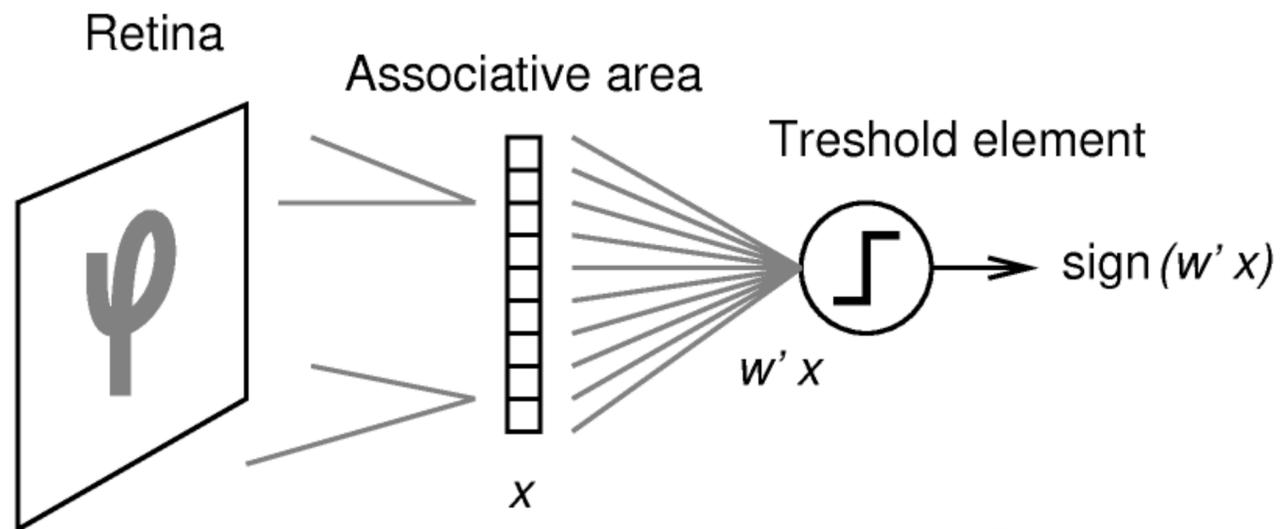
Perceptron (1)

Rosenblatt, 1957



Output: “binary classifier”

$$h(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$



THE MARK I PERCEPTRON

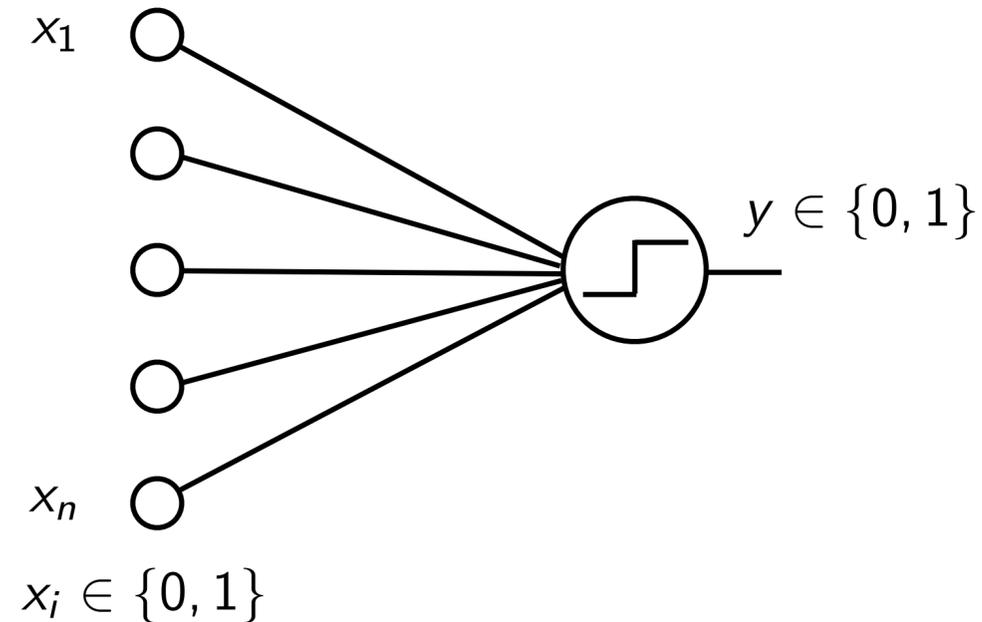
Mark 1 Perceptron. Source: Rosenblatt, Frank (1961) Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms

The perceptron was designed for image recognition. It was first implemented in hardware (400 photocells, weights = potentiometer settings).

Perceptron (2)

McCulloch–Pitts (MCP) neuron (1943)

- ▶ First mathematical model of a biological neuron
- ▶ Boolean input
- ▶ Equal weights for all inputs
- ▶ Threshold hardcoded

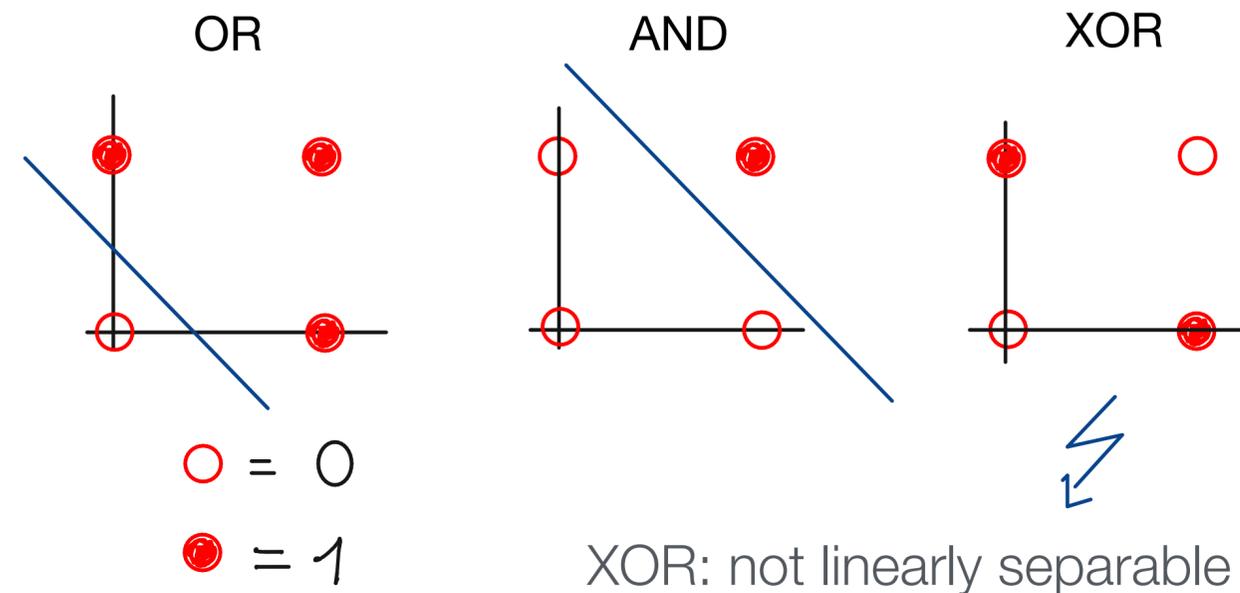


Improvements by Rosenblatt:

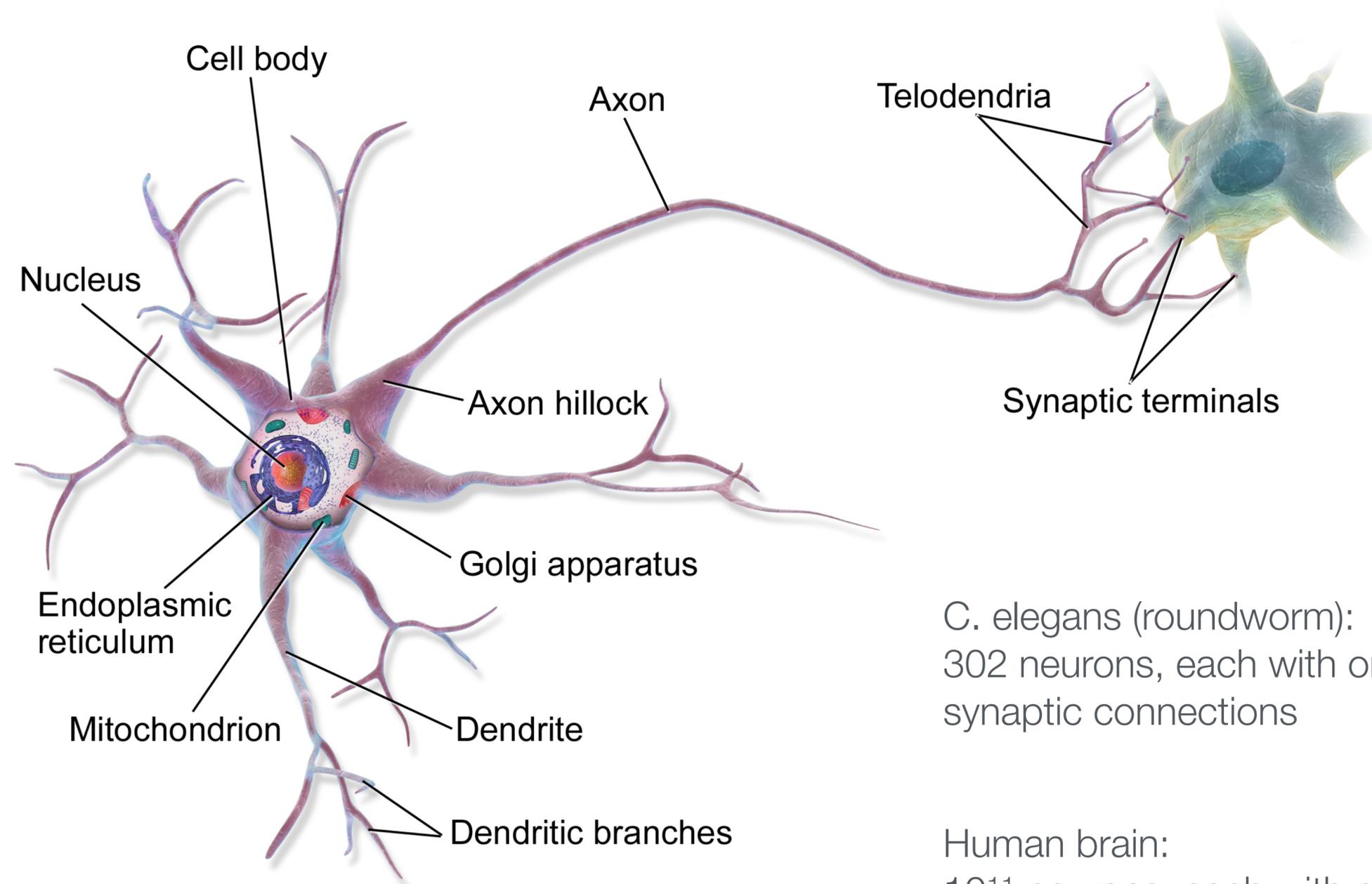
- ▶ Different weights for inputs
- ▶ Algorithm to update weights and threshold given labeled training data

Shortcoming of the perceptron:
it cannot learn the XOR function

Minsky, Papert, 1969



The biological inspiration: the neuron



C. elegans (roundworm):
302 neurons, each with on average 25
synaptic connections

Human brain:
 10^{11} neurons, each with on average
7000 synaptic connections

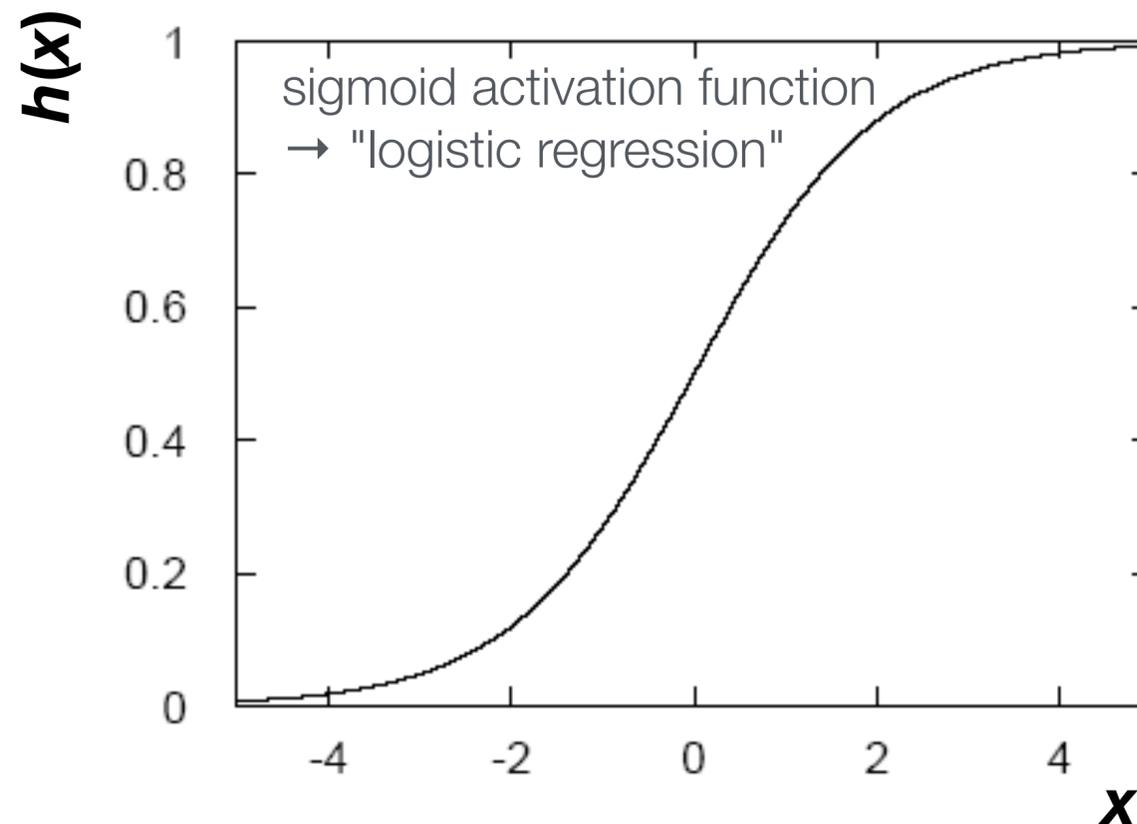
<https://en.wikipedia.org/wiki/Neuron>

https://en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons

Non-linear transfer / activation function

Discriminant:
$$y(\vec{x}) = h \left(w_0 + \sum_{i=1}^n w_i x_i \right)$$

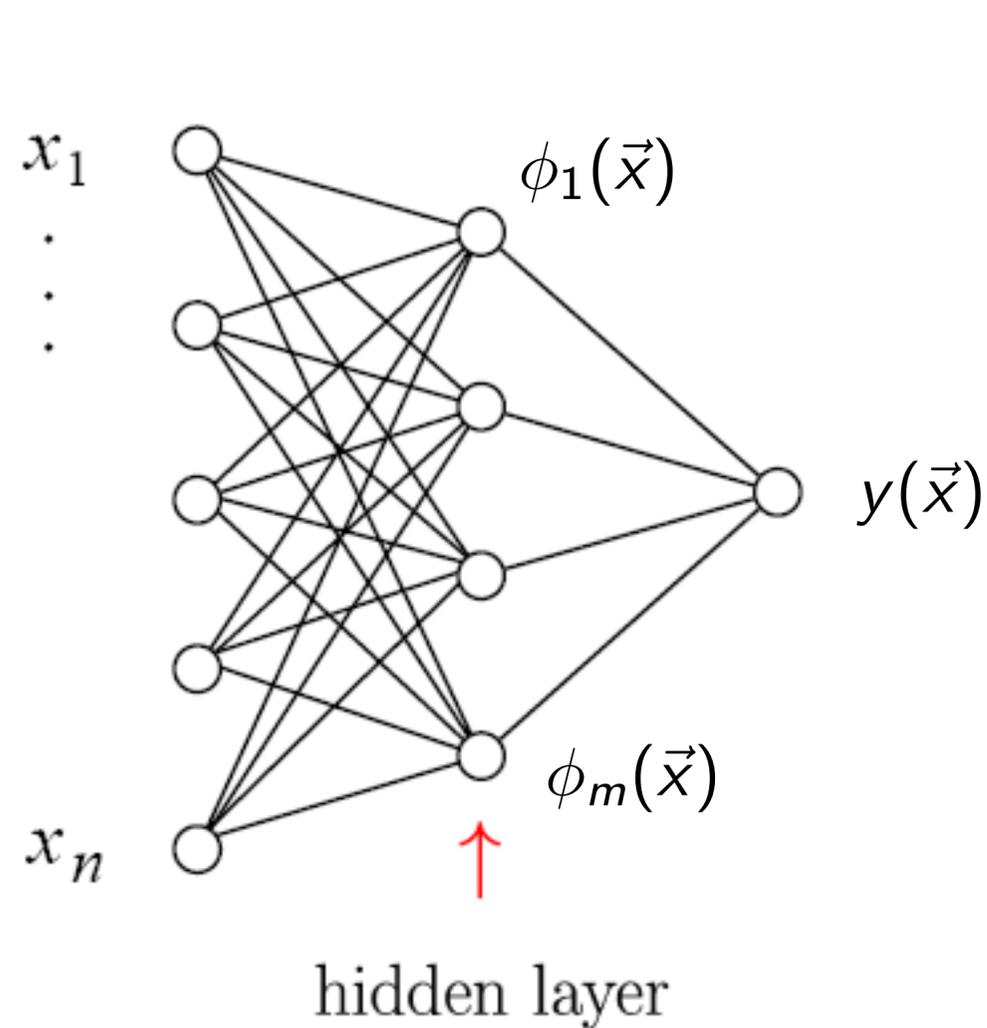
Examples for h : $\frac{1}{1 + e^{-x}}$ ("sigmoid" or "logistic" function), $\tanh x$



Non-linear activation function needed in neural networks when feature space is not linearly separable

Neural net with linear activation functions is just a perceptron

Feedforward neural network with one hidden layer



superscripts indicates layer number

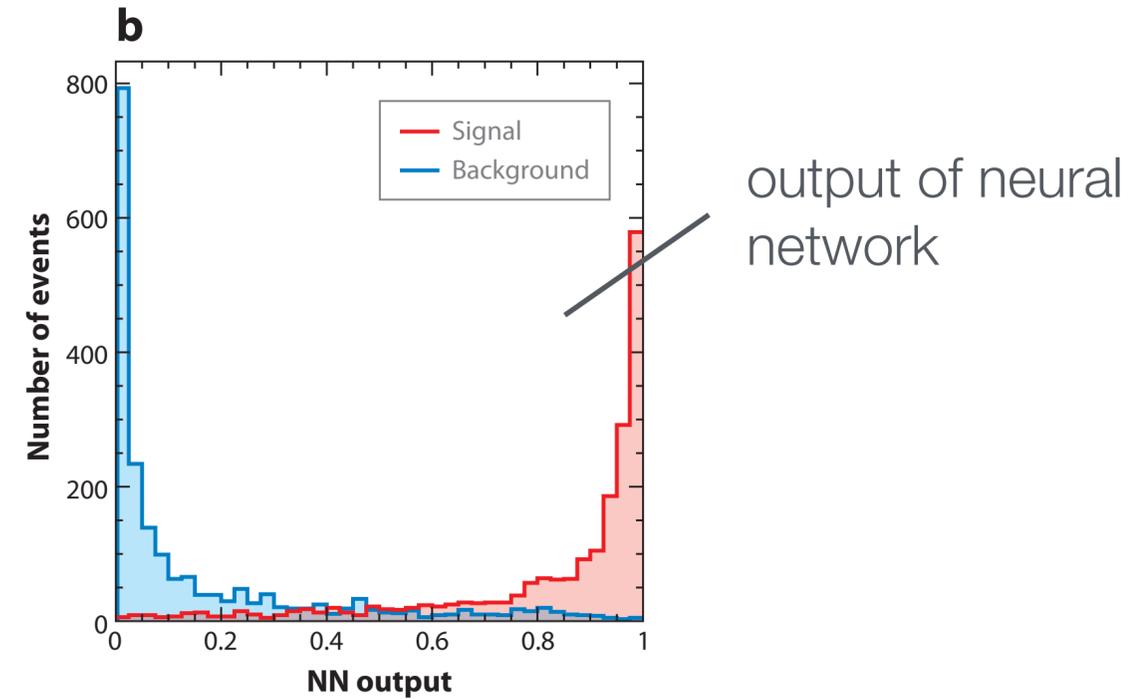
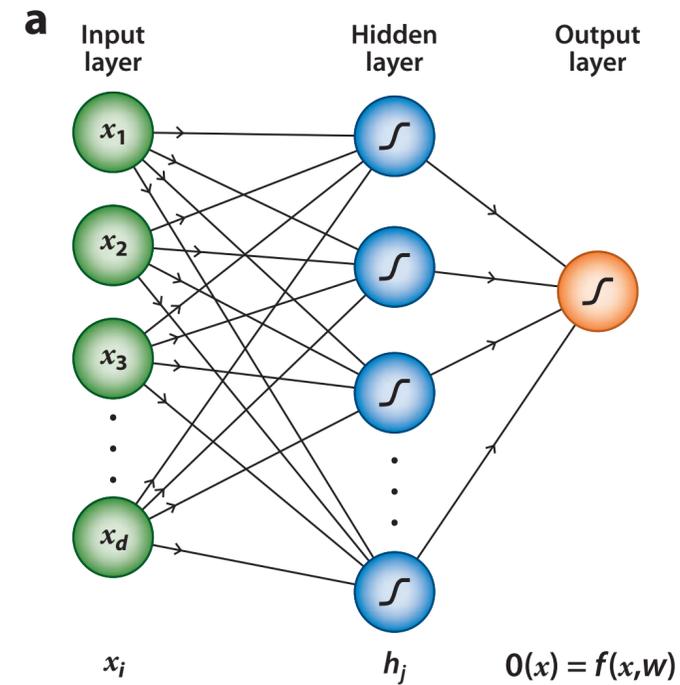
$$\phi_i(\vec{x}) = h \left(w_{i0}^{(1)} + \sum_{j=1}^n w_{ij}^{(1)} x_j \right)$$

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

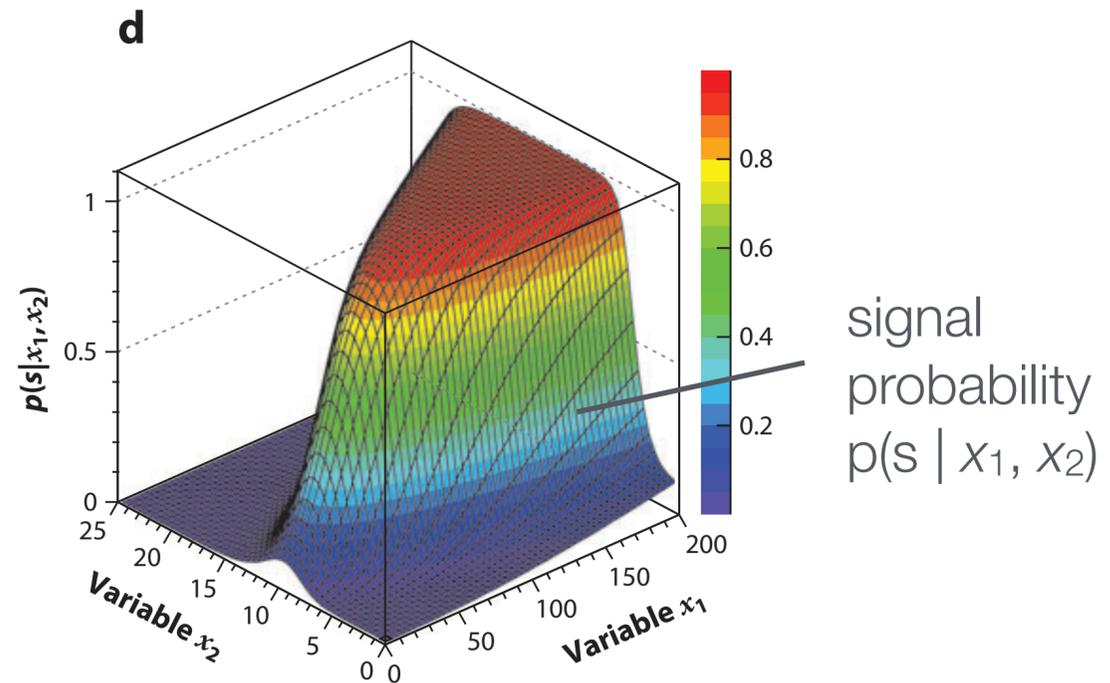
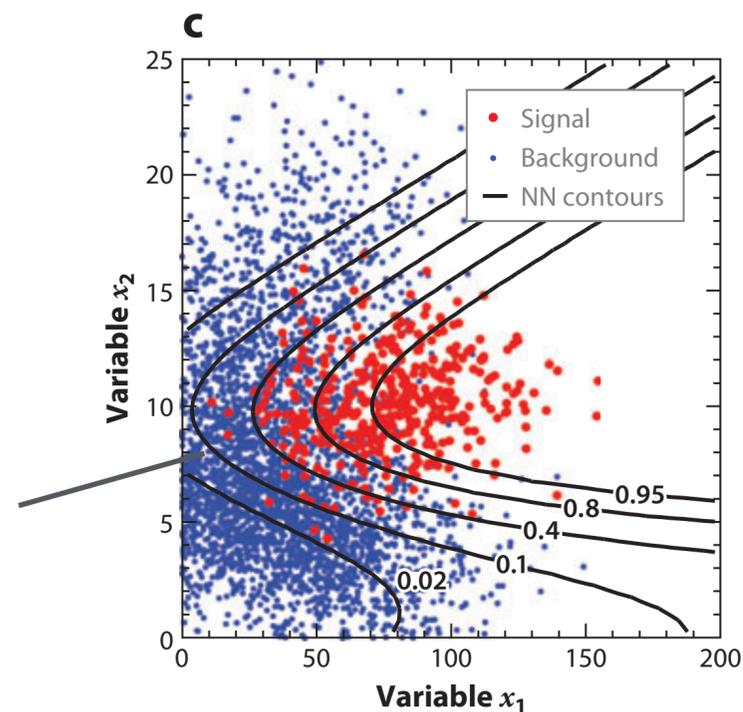
Straightforward to generalize to multiple hidden layers

Neural network output and decision boundaries

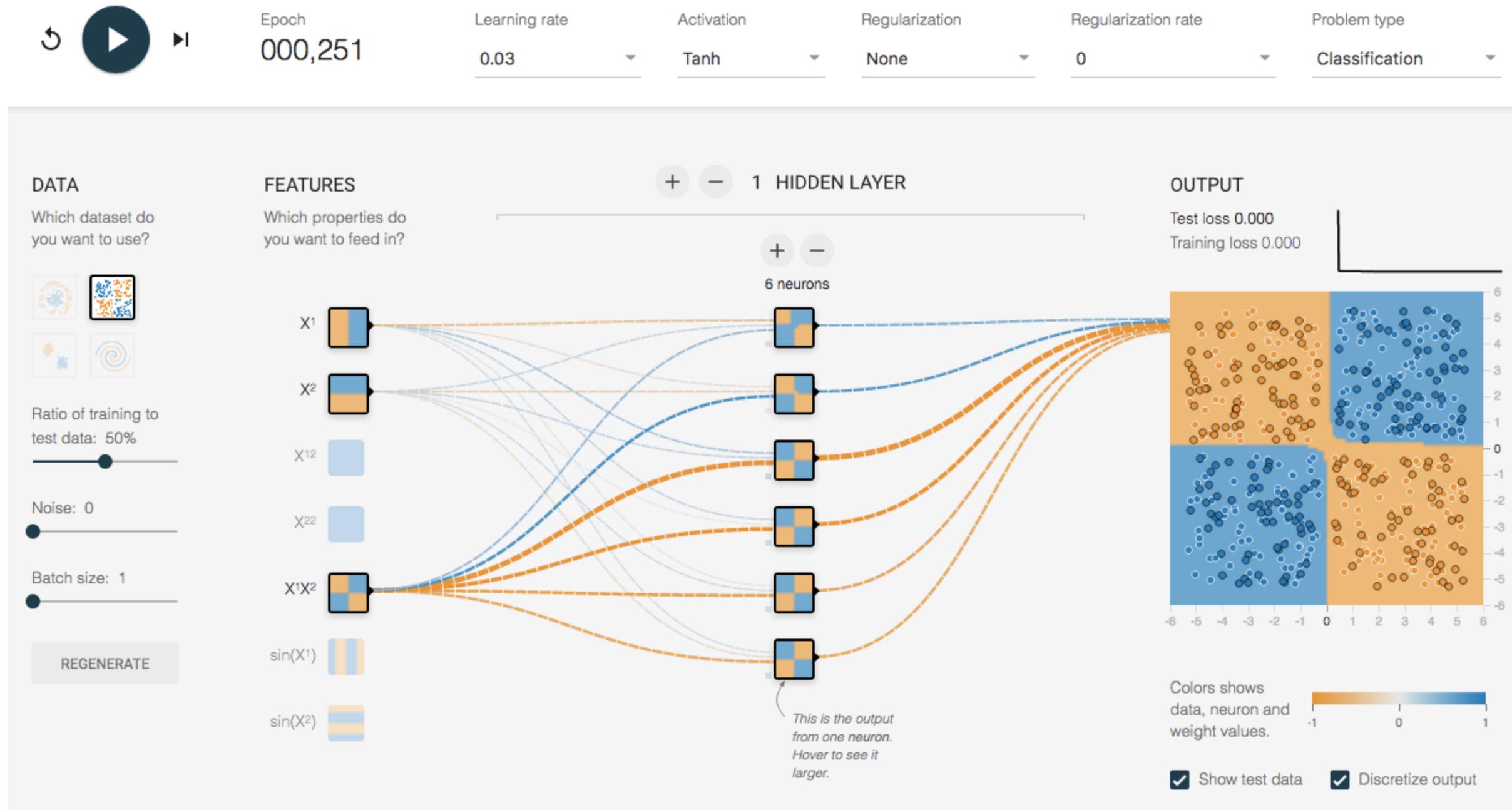
P. Bhat, Multivariate Analysis Methods in Particle Physics, inspirehep.net/record/879273



decision boundaries for different cuts on NN output



Fun with neural nets in the browser



<http://playground.tensorflow.org>

Network training

\vec{x}_a : training event, $a = 1, \dots, N$

t_a : correct label for training event a

↙ e.g., $t_a = 1, 0$ for signal and background, respectively

\vec{w} : vector containing all weights

Loss function (example):

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

Weights are determined by minimizing the loss function (also called error function)

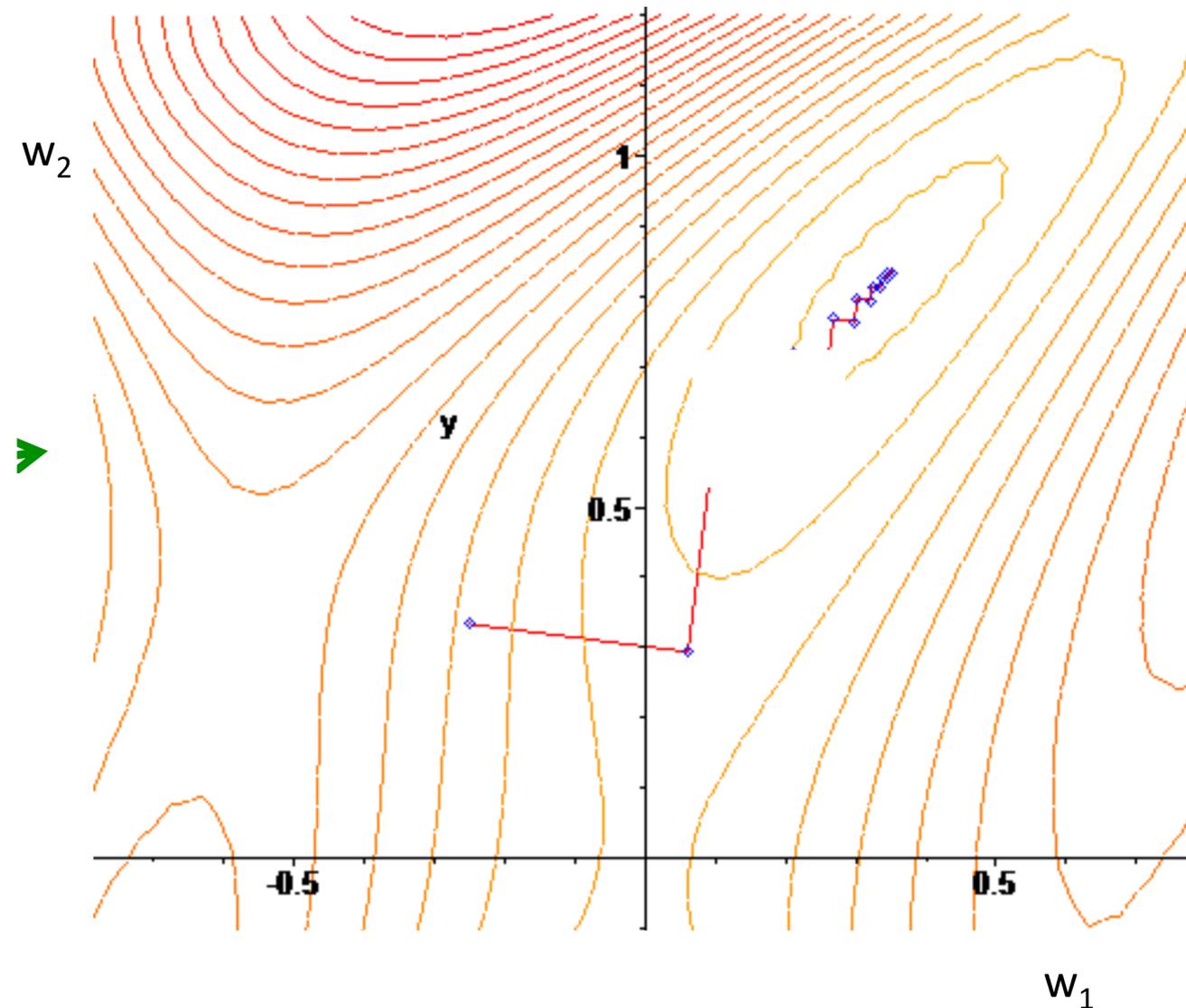
Back-propagation (1)

Start with an initial guess $\vec{w}^{(0)}$ for the weights and then update weights after each training event:

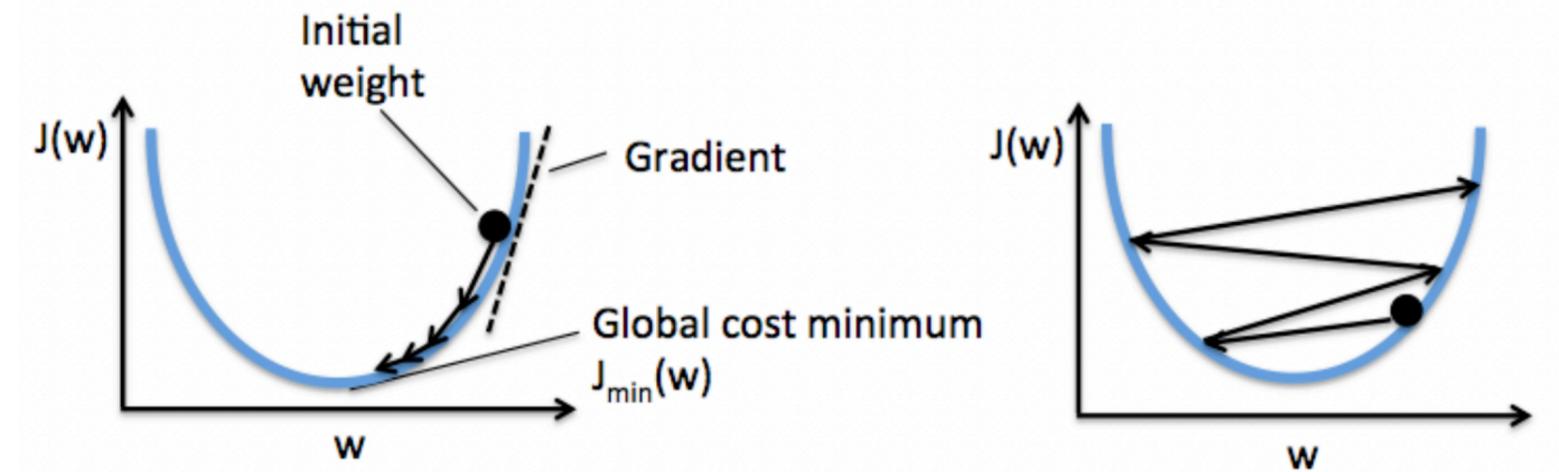
$$\vec{w}^{(\tau+1)} = \vec{w}^{(\tau)} - \eta \nabla E_a(\vec{w}^{(\tau)})$$

└── learning rate

Gradient descent:



Choosing a good learning rate can be essential:



<https://github.com/rasbt/python-machine-learning-book/blob/master/code/ch02/ch02.ipynb>

Back-propagation (2)

Let's write network output as follows:

$$y(\vec{x}) = h(u(\vec{x})) \text{ with } u(\vec{x}) = \sum_{j=0}^m w_{1j}^{(2)} \phi_j(\vec{x}), \quad \phi_j(\vec{x}) = h\left(\sum_{k=0}^n w_{jk}^{(1)} x_k\right) \equiv h(v_j(\vec{x}))$$

Here we defined $\phi_0 = x_0 = 1$ and the sums start from 0 to include the offsets.

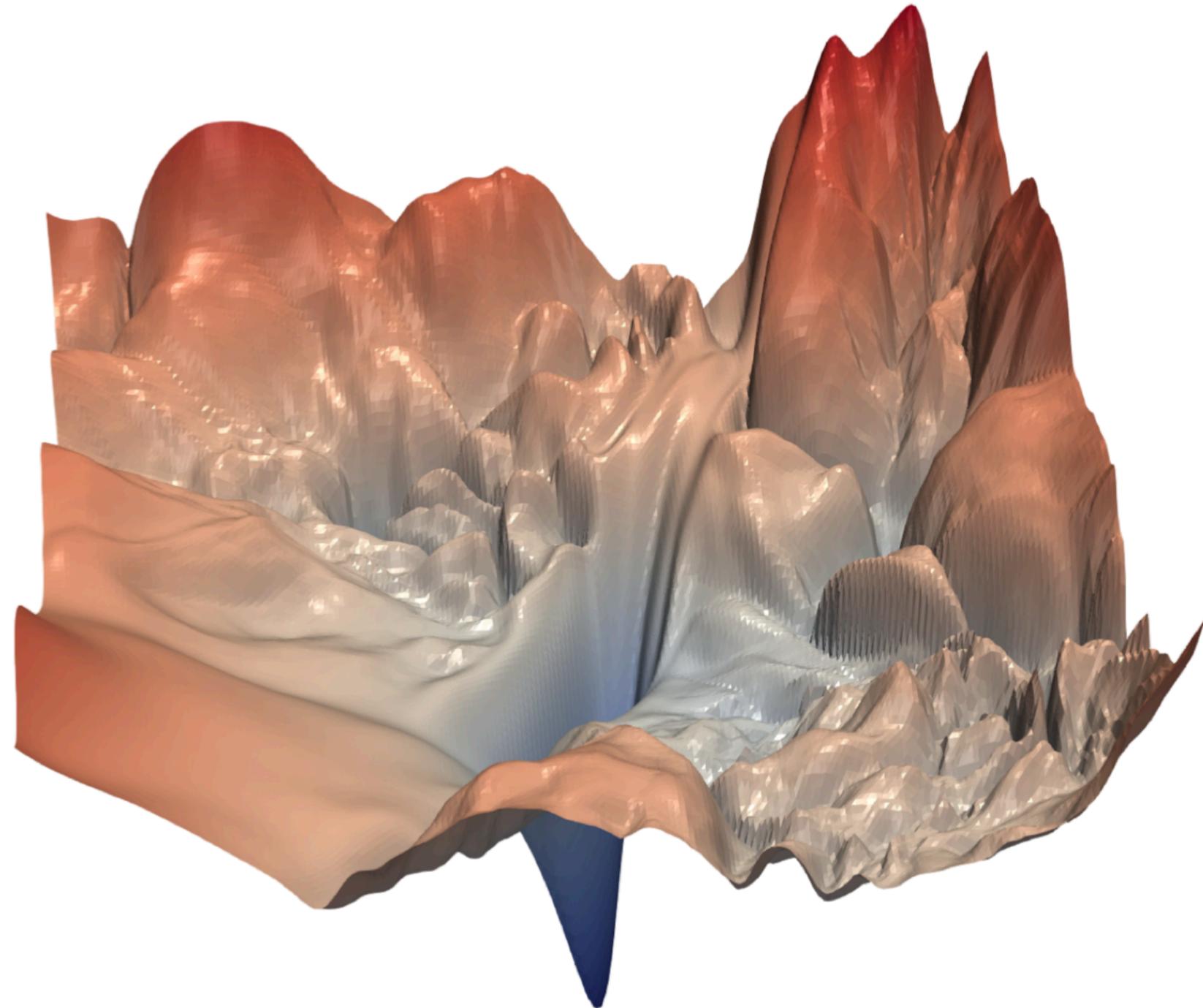
Weights from hidden layer to output:

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

Further application of the **chain rule** gives weights from input to hidden layer.

"Learning representations by back-propagating errors.",
Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams

Example of a loss landscape of a modern artificial neural network



Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, Tom Goldstein

Visualizing the Loss Landscape of Neural Nets

https://papers.nips.cc/paper_files/paper/2018/hash/a41b3bb3e6b050b6c9067c67f663b915-Abstract.html

More on gradient descent

■ Stochastic gradient descent

- ▶ just uses one training event at a time
- ▶ fast, but quite irregular approach to the minimum
- ▶ can help escape local minima
- ▶ one can decrease learning rate to settle at the minimum ("simulated annealing")

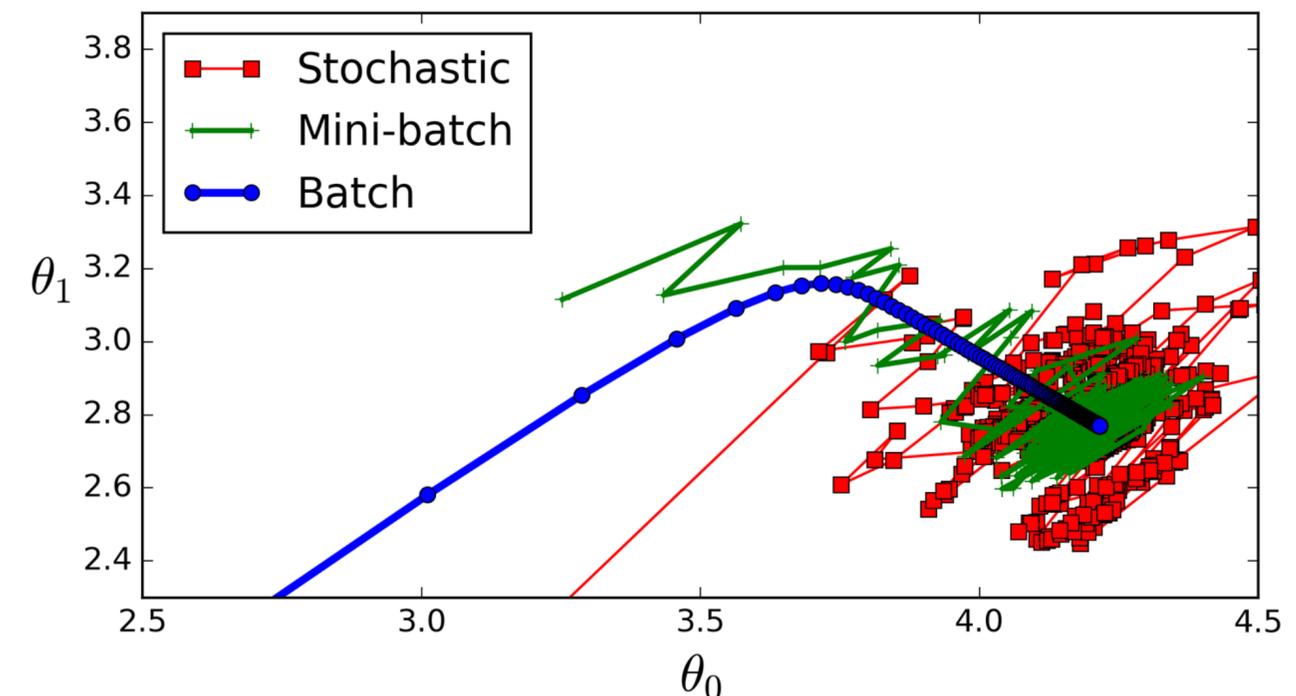
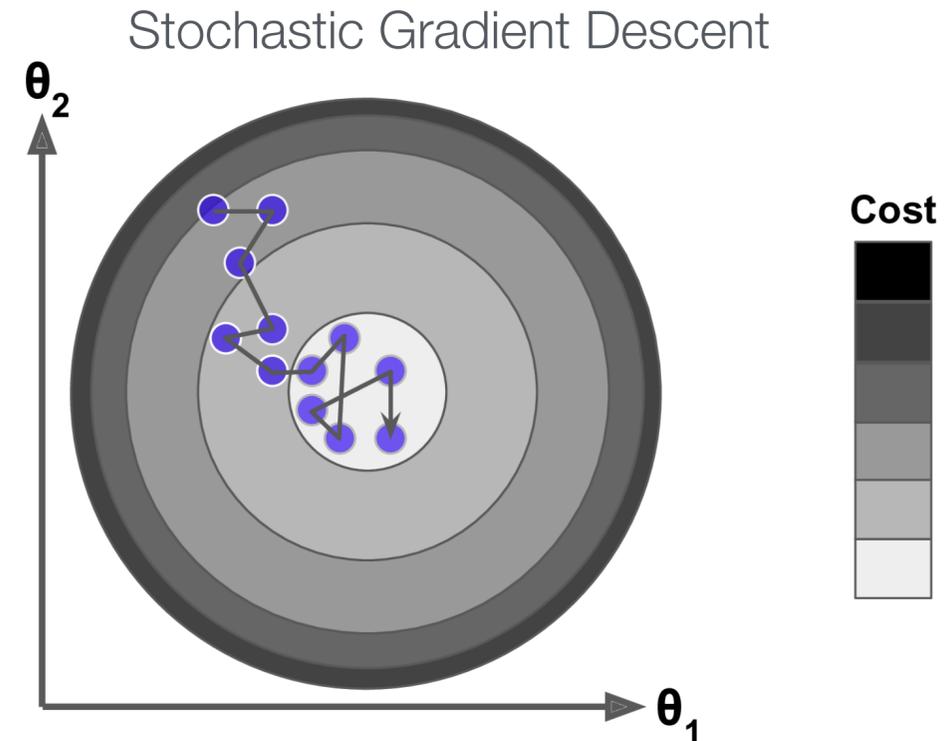
■ Batch gradient descent

- ▶ use entire training sample to calculate gradient of loss function
- ▶ computationally expensive

■ Mini-batch gradient descent

- ▶ calculate gradient for a random sub-sample of the training set

Aurélien Géron, Hands-On Machine Learning with Scikit-Learn and TensorFlow



Universal approximation theorem

https://en.wikipedia.org/wiki/Universal_approximation_theorem

"A feed-forward network with a single hidden layer containing a finite number of neurons (i.e., a multilayer perceptron), can approximate continuous functions on compact subsets of \mathbb{R}^n ."

One of the first versions of the theorem was proved by George Cybenko in 1989 for sigmoid activation functions

The theorem does not touch upon the algorithmic learnability of those parameters

Math. Control Signals Systems (1989) 2: 303–314

**Mathematics of Control,
Signals, and Systems**

© 1989 Springer-Verlag New York Inc.

Approximation by Superpositions of a Sigmoidal Function*

G. Cybenko†

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, univariate function and a set of affine functionals can uniformly approximate any continuous function of n real variables with support in the unit hypercube; only mild conditions are imposed on the univariate function. Our results settle an open question about representability in the class of single hidden layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single internal, hidden layer and any continuous sigmoidal nonlinearity. The paper discusses approximation properties of other possible types of nonlinearities that might be implemented by artificial neural networks.

Key words. Neural networks, Approximation, Completeness.

<https://link.springer.com/article/10.1007/BF02551274>

Deep neural networks

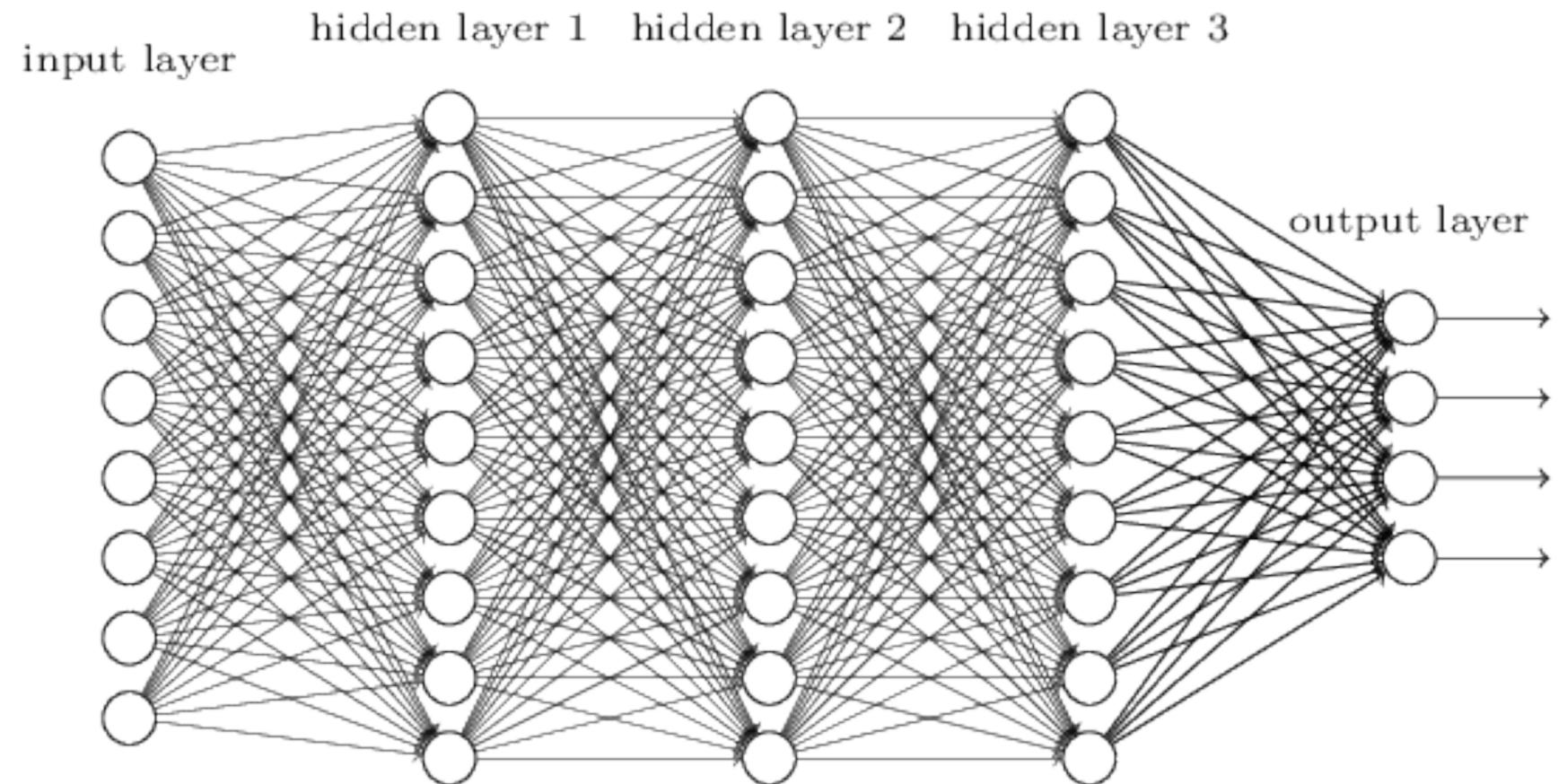
Deep networks: many hidden layers with large number of neurons

Challenges

- ▶ Hard to train ("vanishing gradient problem")
- ▶ Training slow
- ▶ Risk of overtraining

Big progress in recent years

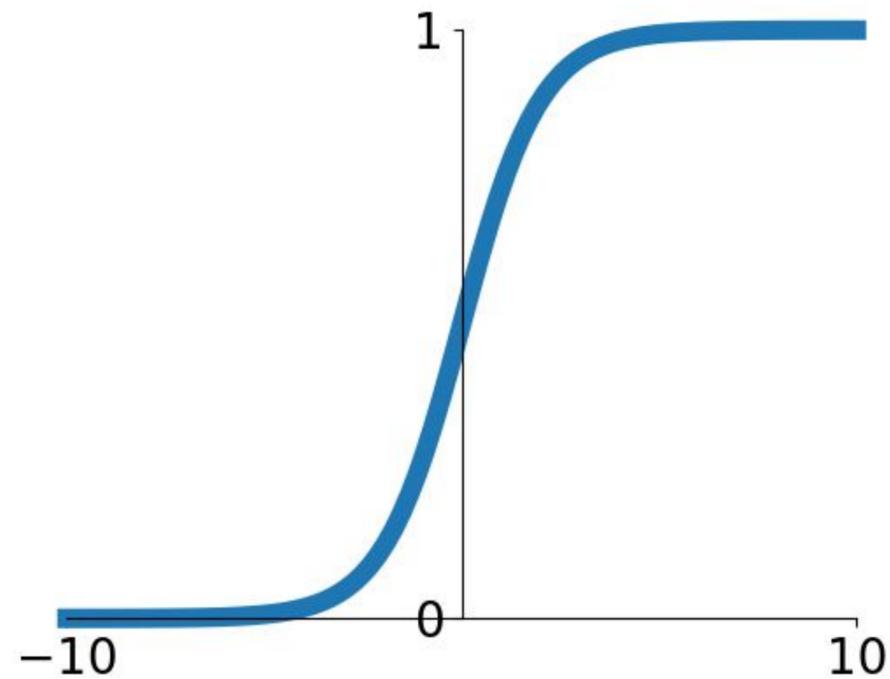
- ▶ Interest in NN waned before ca. 2006
- ▶ Milestone: paper by G. Hinton (2006): "learning for deep belief nets"
- ▶ Image recognition, AlphaGo, ...
- ▶ Soon: self-driving cars, ...



<http://neuralnetworksanddeeplearning.com>

Drawbacks of the sigmoid activation function

<http://cs231n.stanford.edu/slides>



Sigmoid

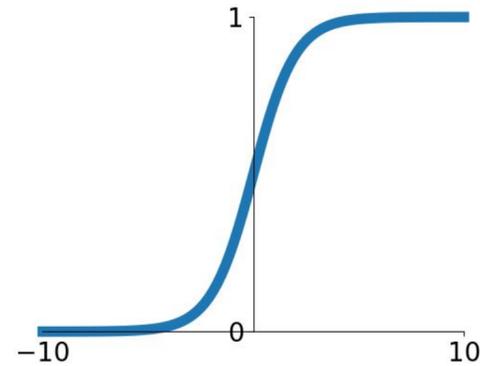
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- Saturated neurons “kill” the gradients
- Sigmoid outputs are not zero-centered
- $\exp()$ is a bit compute expensive

Activation functions

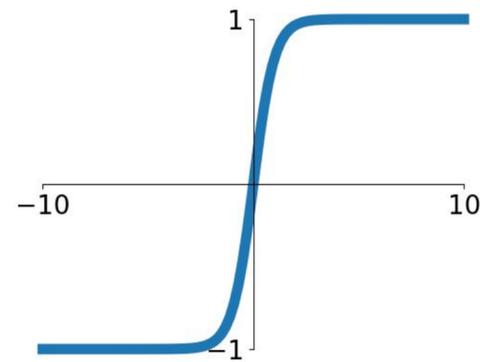
Sigmoid

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



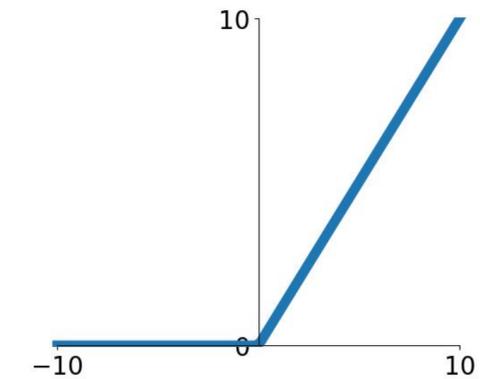
tanh

$$\tanh(x)$$



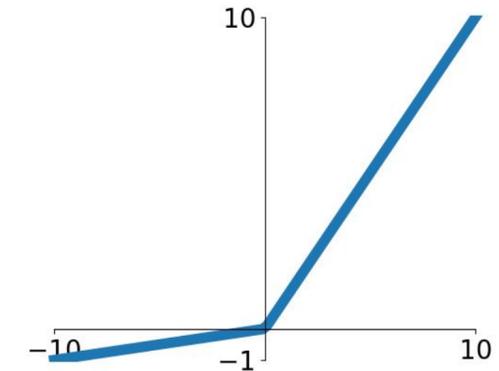
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

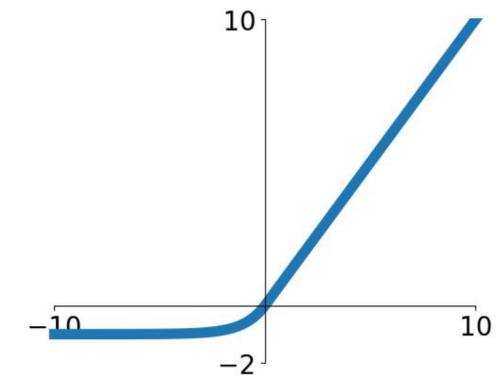


Maxout

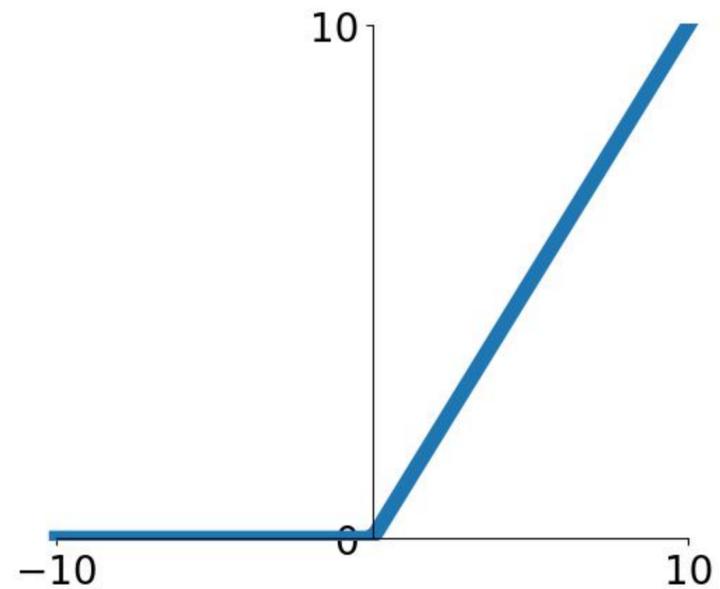
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



<http://cs231n.stanford.edu/slides>



ReLU (Rectified Linear Unit)

$$f(x) = \max(0, x)$$

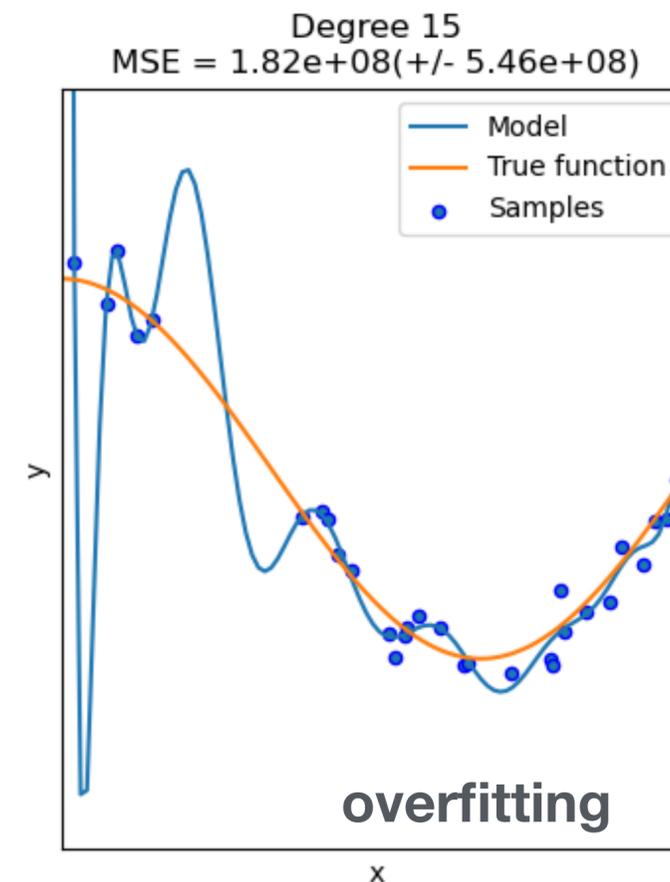
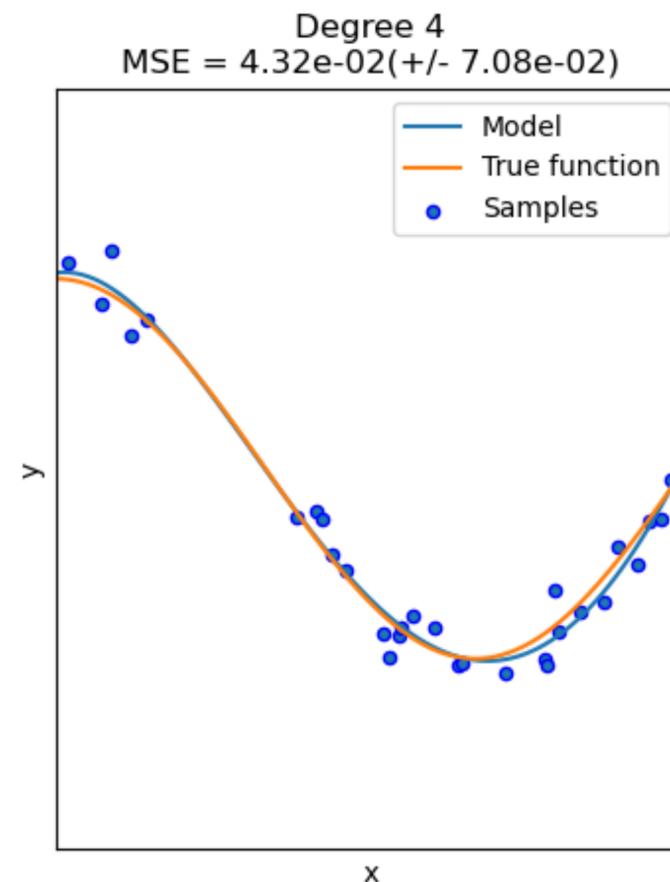
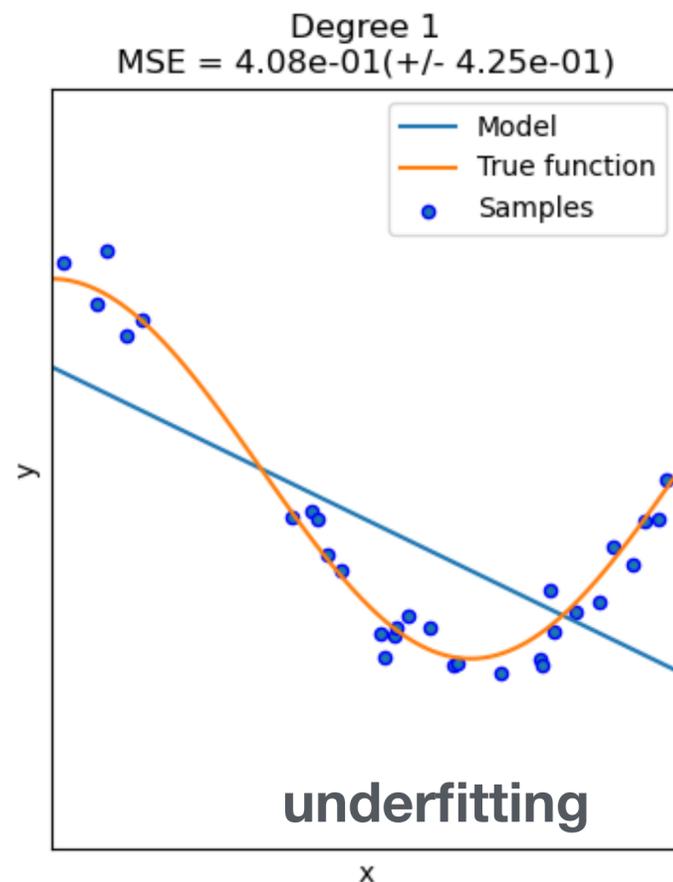
- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

But: gradient vanishes for $x < 0$

Bias-variance tradeoff

Goal: generalization of training data

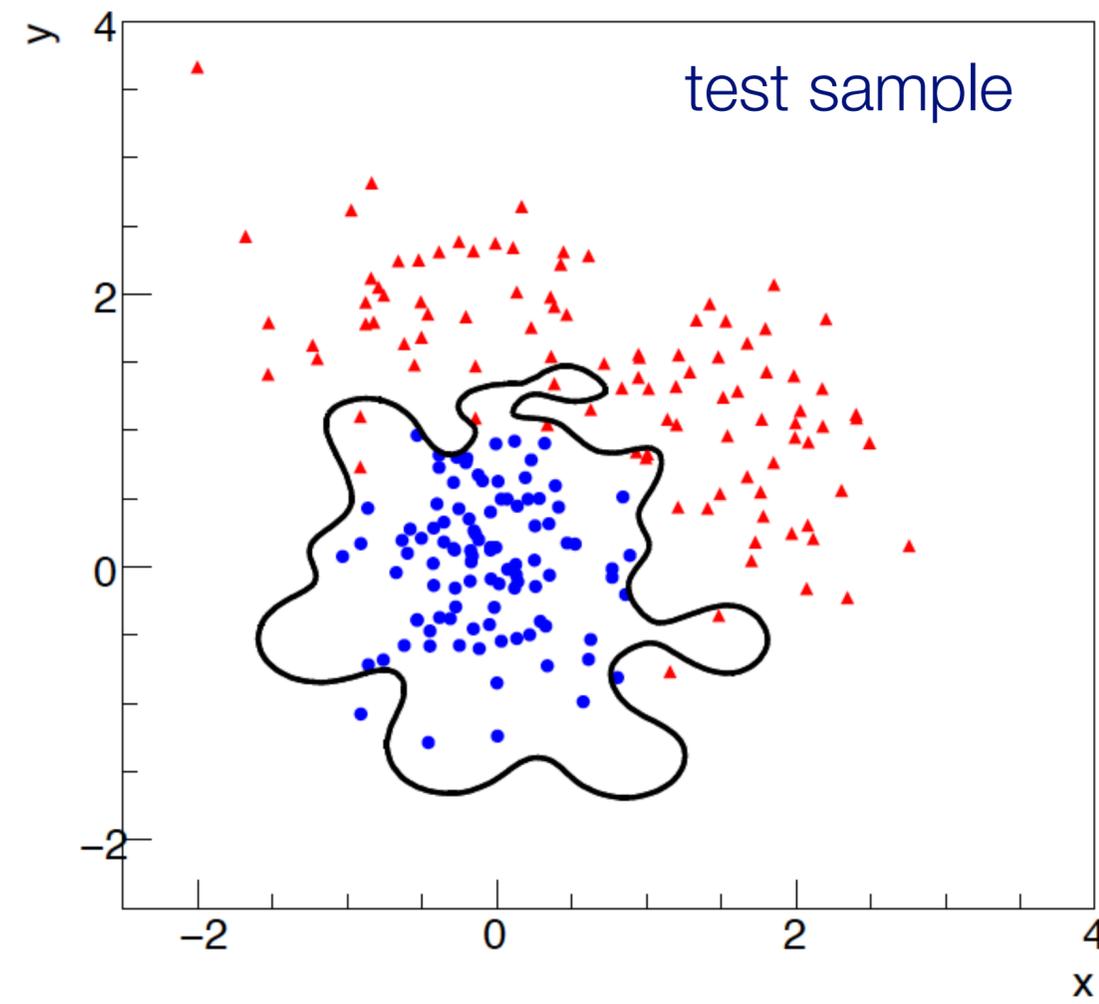
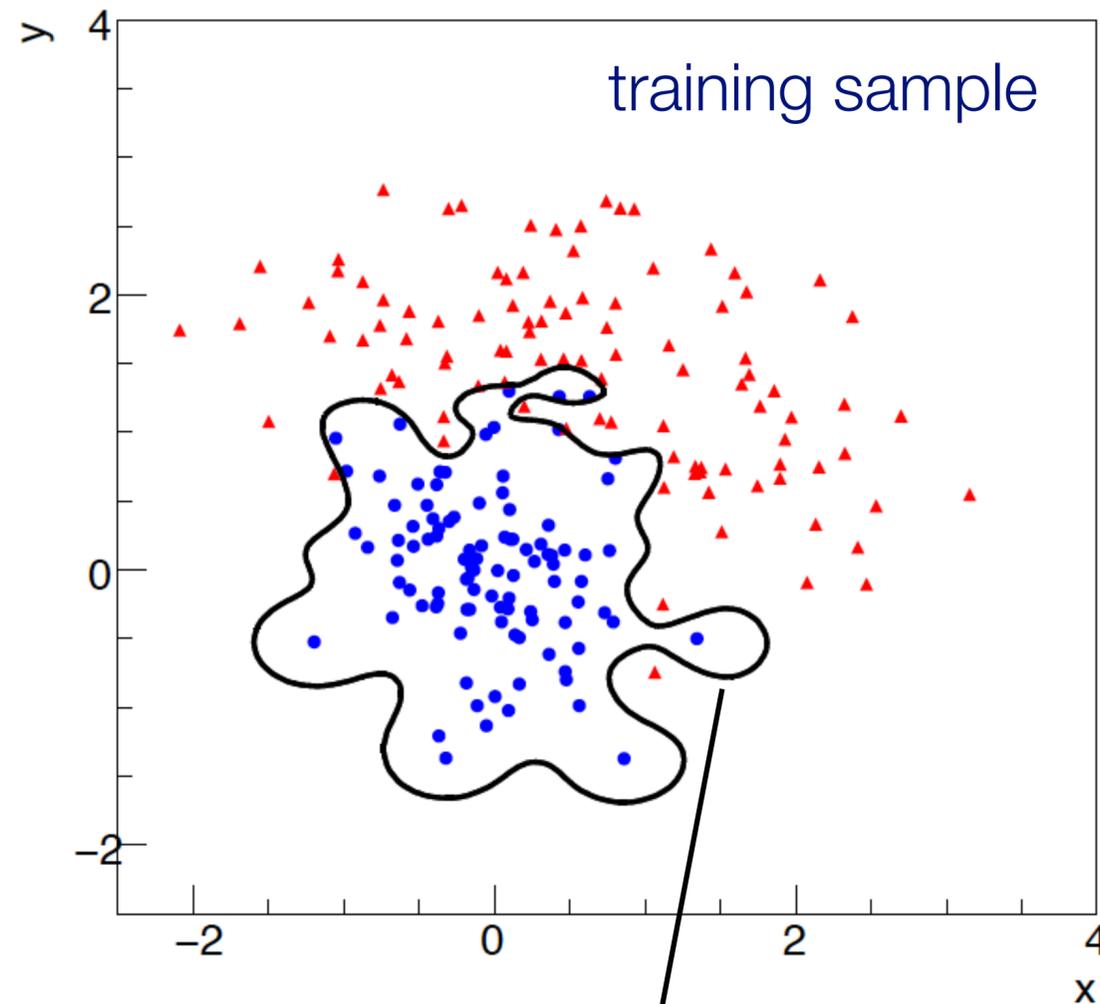
- Simple models (few parameters): danger of bias
 - ▶ Classifiers with a small number of degrees of freedom are less prone to statistical fluctuations: different training samples would result in similar classification boundaries ("small variance")
- Complex models (many parameters): danger of overfitting
 - ▶ large variance of decision boundaries for different training samples



https://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html

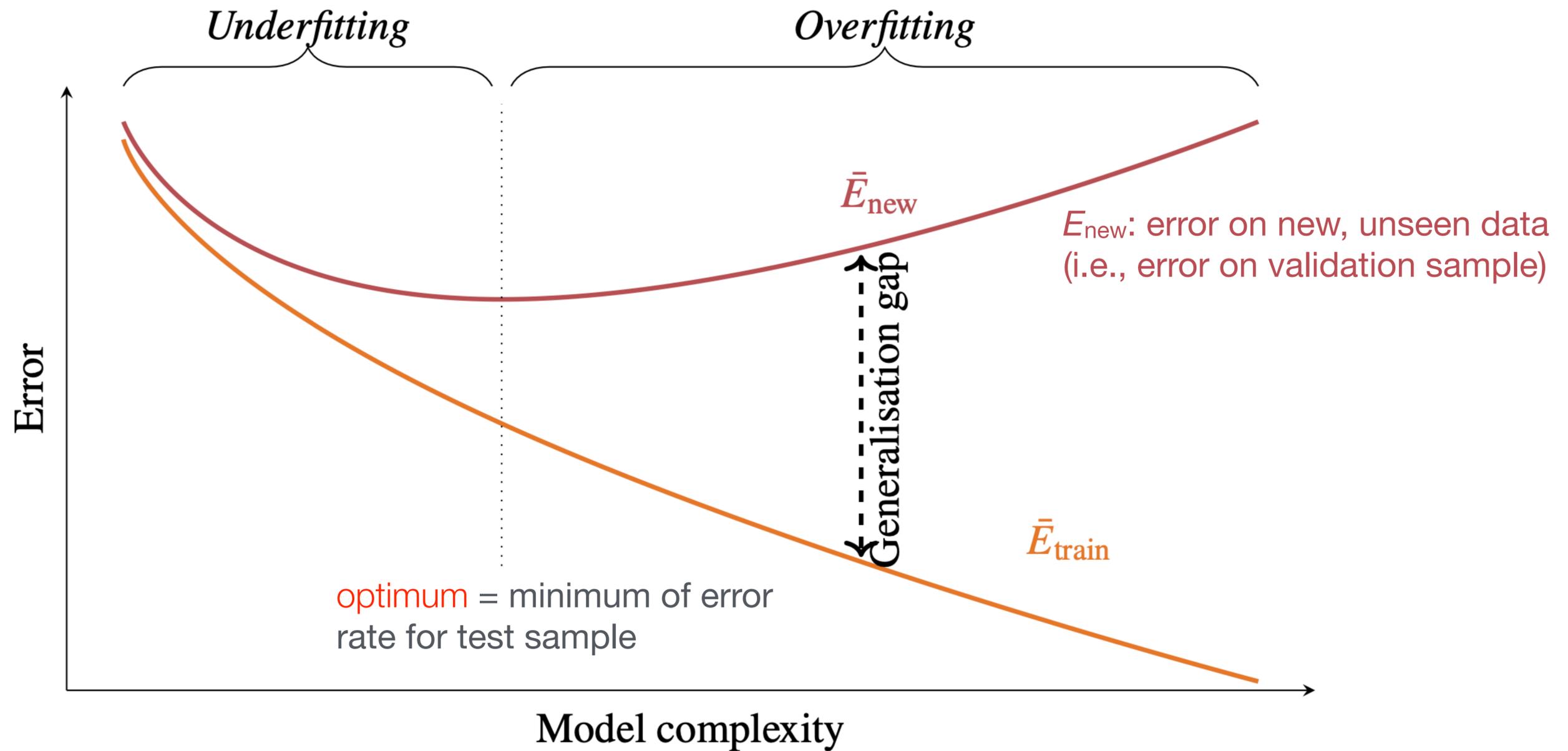
Example of overtraining

Too many neurons/layers make a neural network too flexible → overtraining



Network "learns" features that are merely statistical fluctuations in the training sample

Error on training and validation sample vs. model complexity

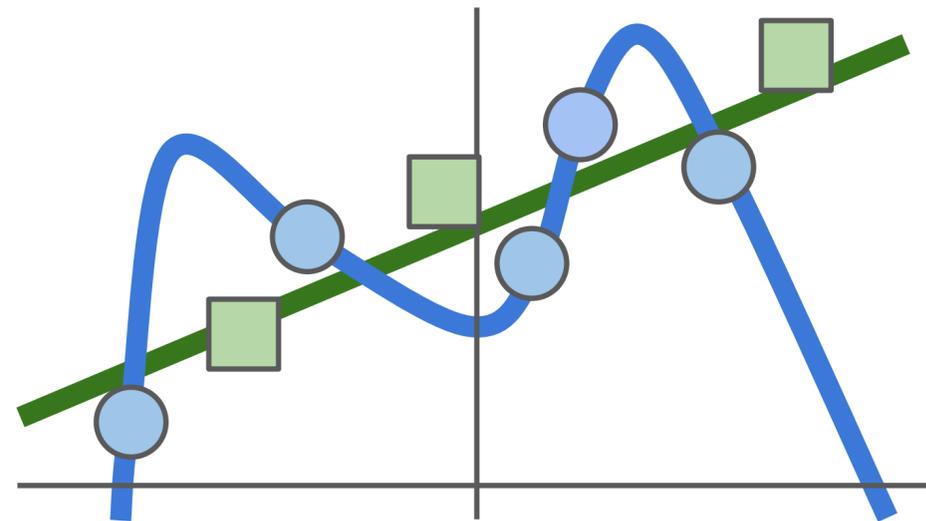


Regularization: Avoid overfitting

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \underbrace{\lambda R(W)}_{\text{Regularization}}$$

Data loss: Model predictions should match training data

Regularization: Model should be “simple”, so it works on test data



Occam's Razor:

“Among competing hypotheses, the simplest is the best”

William of Ockham, 1285 - 1347

In common use:

L2 regularization

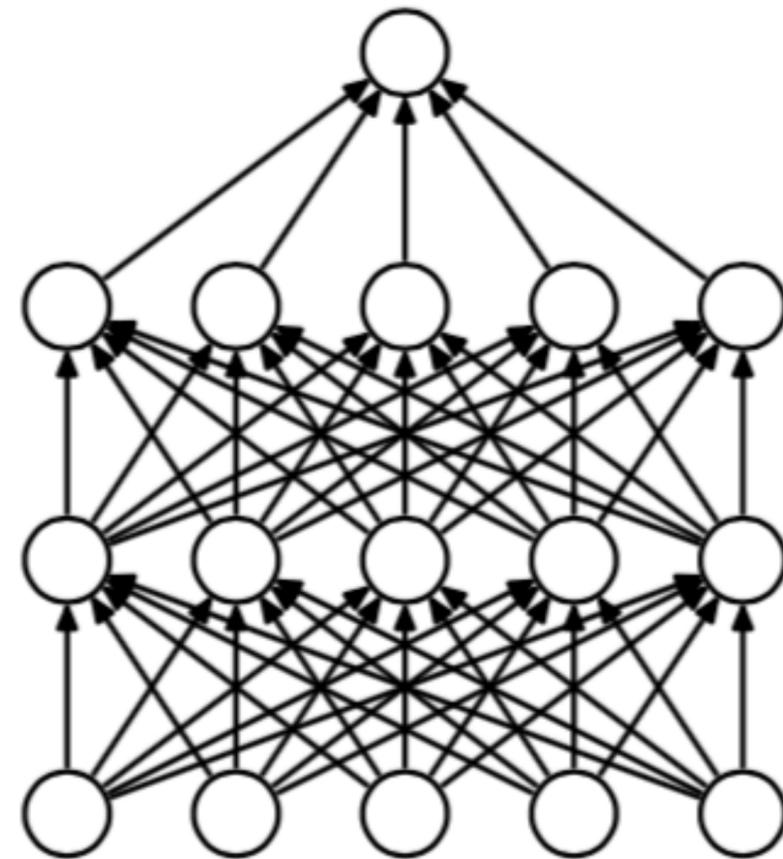
$$R(W) = \sum_k \sum_l W_{k,l}^2$$

L1 regularization

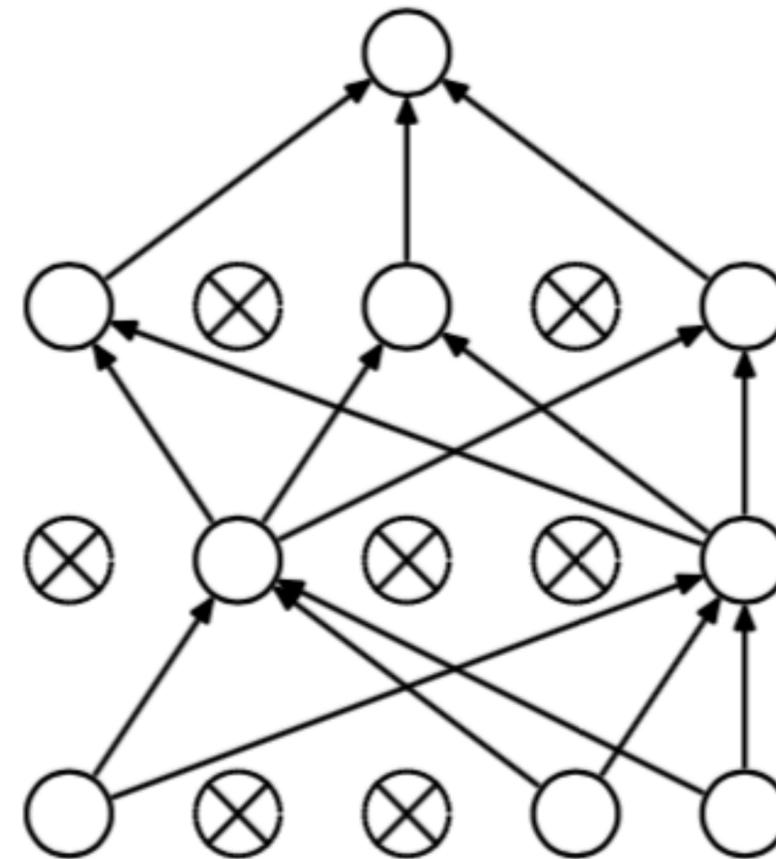
$$R(W) = \sum_k \sum_l |W_{k,l}|$$

Another approach to prevent overfitting: Dropout

- Randomly remove nodes during training
- Avoid co-adaptation of nodes



(a) Standard Neural Net

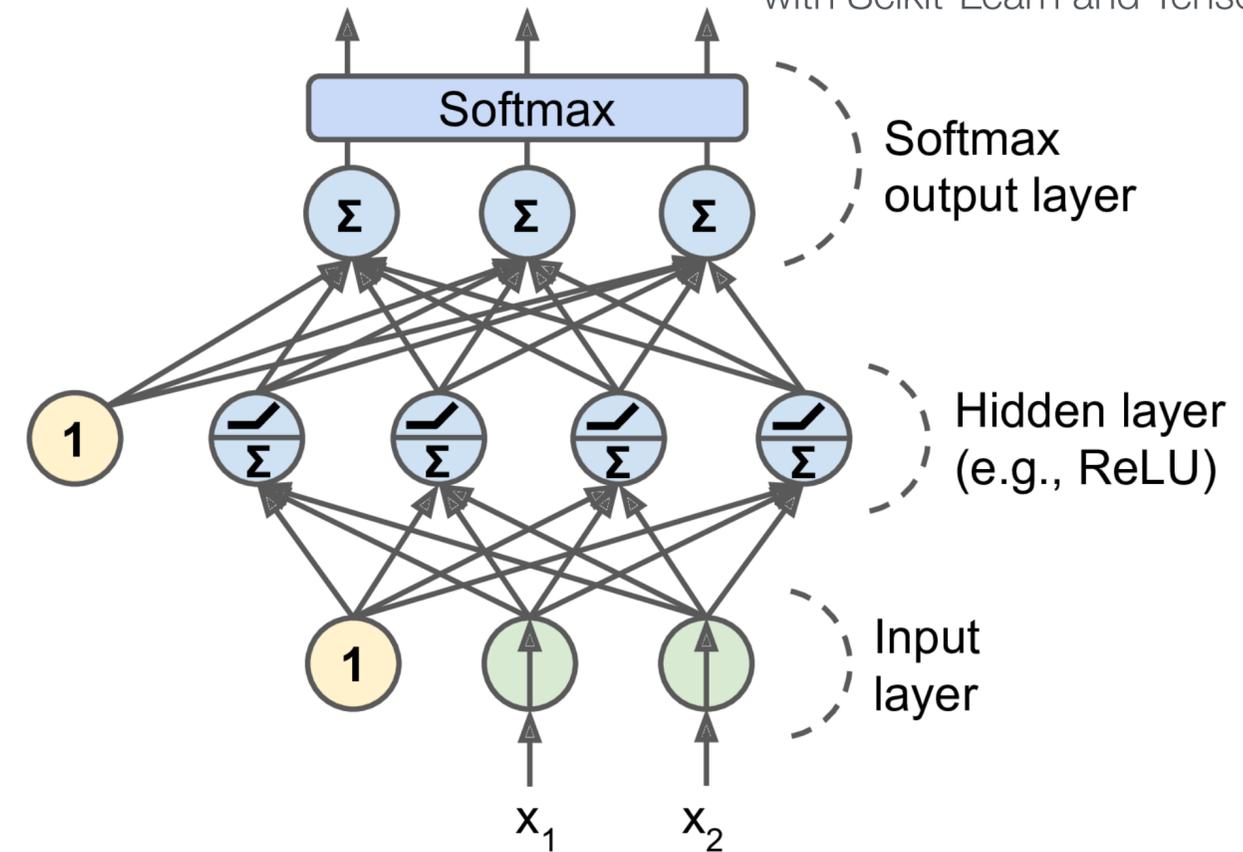


(b) After applying dropout.

Srivastava et al., ["Dropout: A Simple Way to Prevent Neural Networks from Overfitting"](#)

Xavier and He initialization

- Initial weights determine speed of convergence and whether algorithm converges at all
- Xavier Glorot and Yoshua Bengio
 - Paper "[Understanding the Difficulty of Training Deep Feedforward Neural Networks](#)"
 - Idea: Variance of the outputs of each layer to be equal to the variance of its inputs



Layer with n_{in} inputs connected to n_{out} neurons in the next layer

Activation function	Uniform distribution $[-r, r]$	Normal distribution ($\mu = 0$)
Logistic	$r = \sqrt{\frac{6}{n_{in} + n_{out}}}$	$\sigma = \sqrt{\frac{2}{n_{in} + n_{out}}}$
tanh	$r = 4\sqrt{\frac{6}{n_{in} + n_{out}}}$	$\sigma = 4\sqrt{\frac{2}{n_{in} + n_{out}}}$
ReLU (and variants)	$r = \sqrt{2}\sqrt{\frac{6}{n_{in} + n_{out}}}$	$\sigma = \sqrt{2}\sqrt{\frac{2}{n_{in} + n_{out}}}$

Pros and cons of multi-layer perceptrons

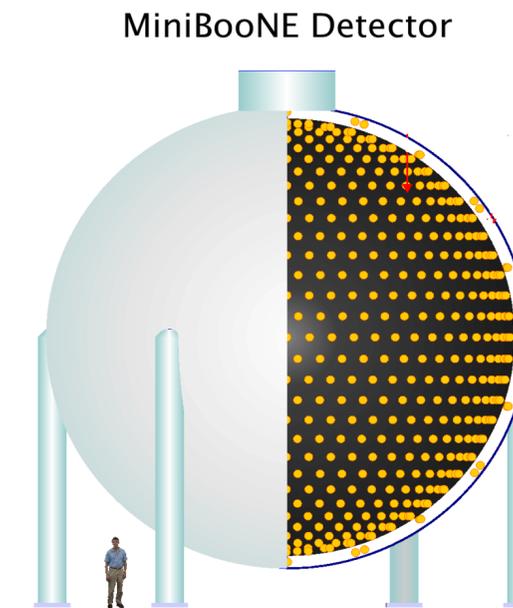
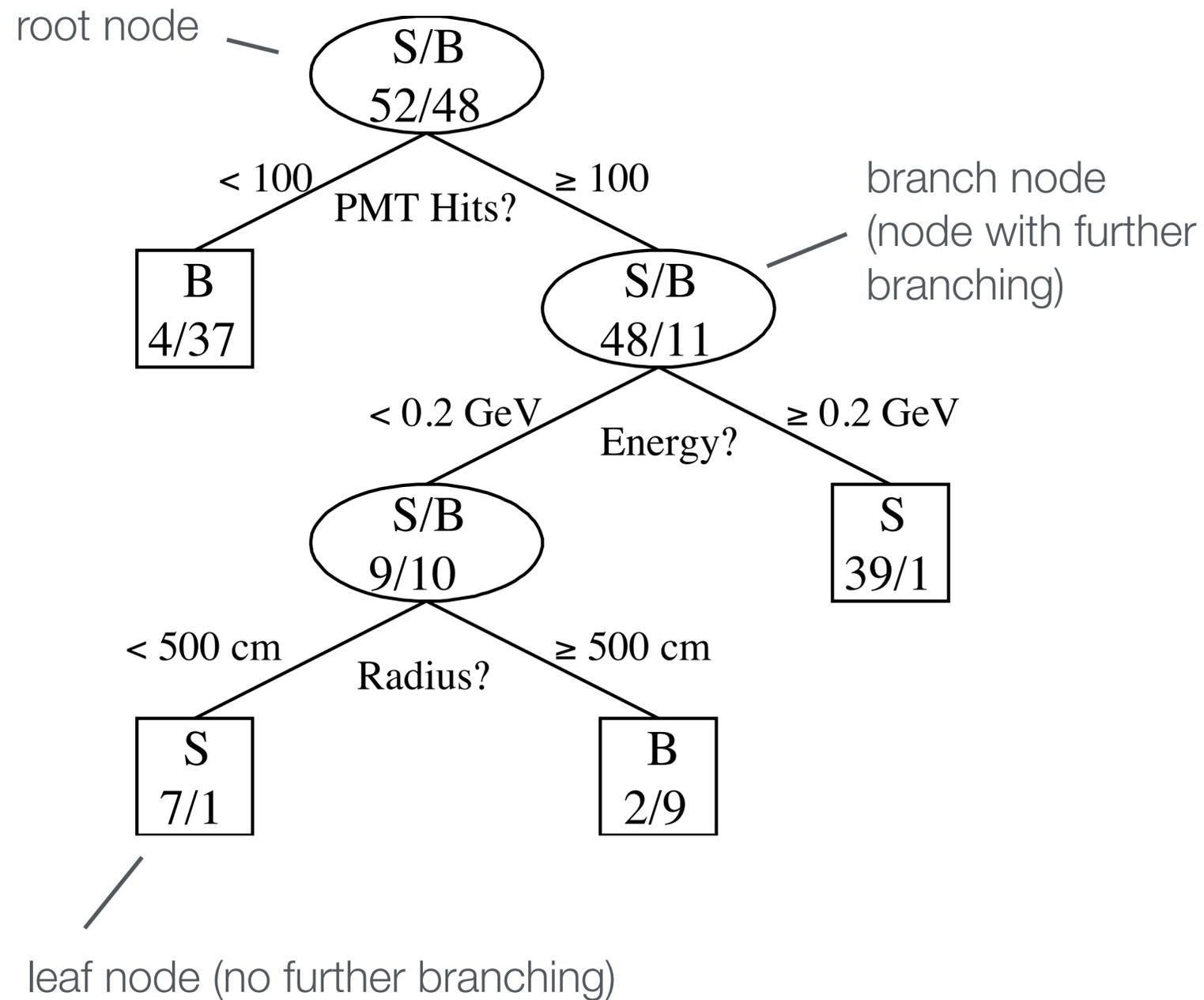
Pros:

- Capability to learn non-linear models

Cons:

- Loss function can have several local minima
- Hyperparameters need to be tuned
 - ▶ number of layers, neurons per layer, and training iterations
- Sensitive to feature scaling
 - ▶ preprocessing needed (e.g., scaling of all feature to range $[0,1]$)

Decision trees

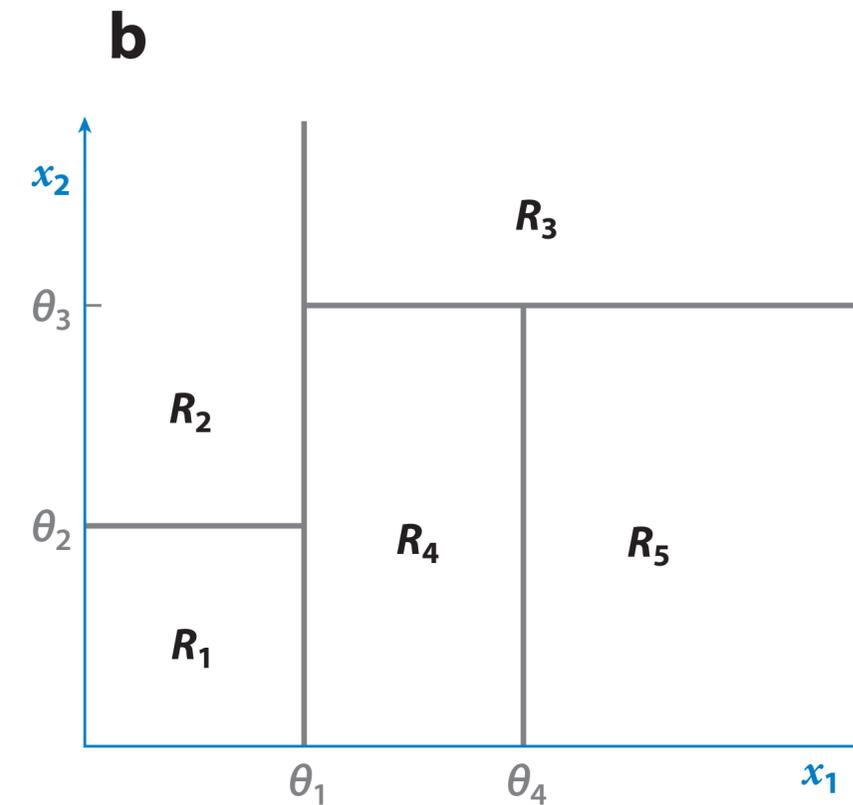
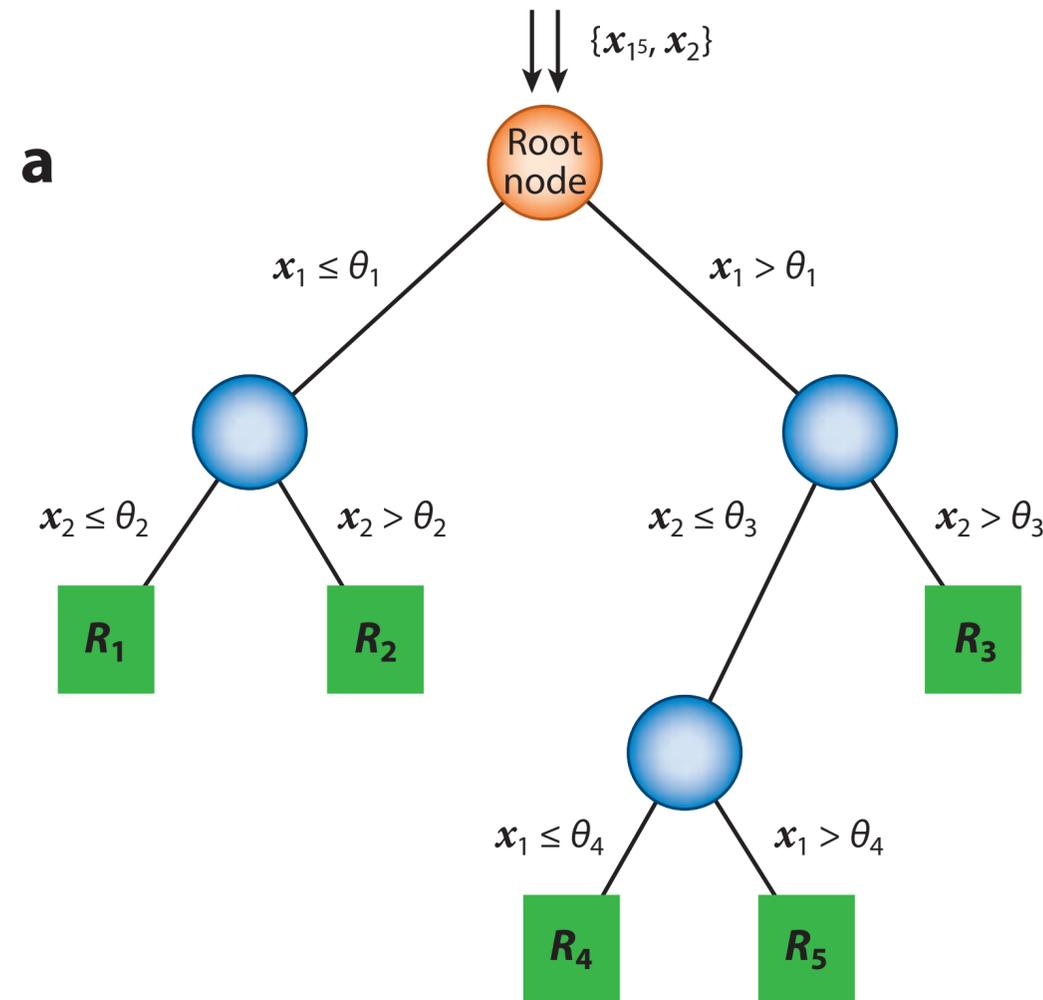


MiniBooNE: 1520 photomultiplier signals, goal: separation of ν_e from ν_μ events

arXiv:physics/0508045v1

Leaf nodes classify events as either signal or background

Decision trees



Easy to interpret and visualize:
Space of feature vectors split up into rectangular volumes
(attributed to either signal or background)

How to build a decision tree in an optimal way?

Finding optimal cuts

Separation btw. signal and background is often measured with the *Gini index* (or *Gini impurity*):

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

Here p is the purity:

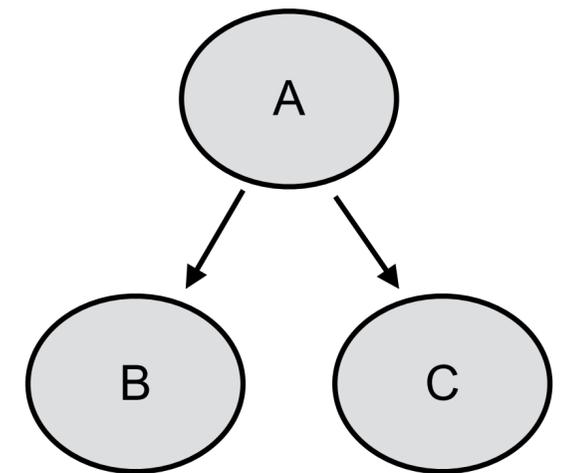
$$p = \frac{\sum_{\text{signal}} w_i}{\sum_{\text{signal}} w_i + \sum_{\text{background}} w_i}$$

$w_i =$ weight of event i

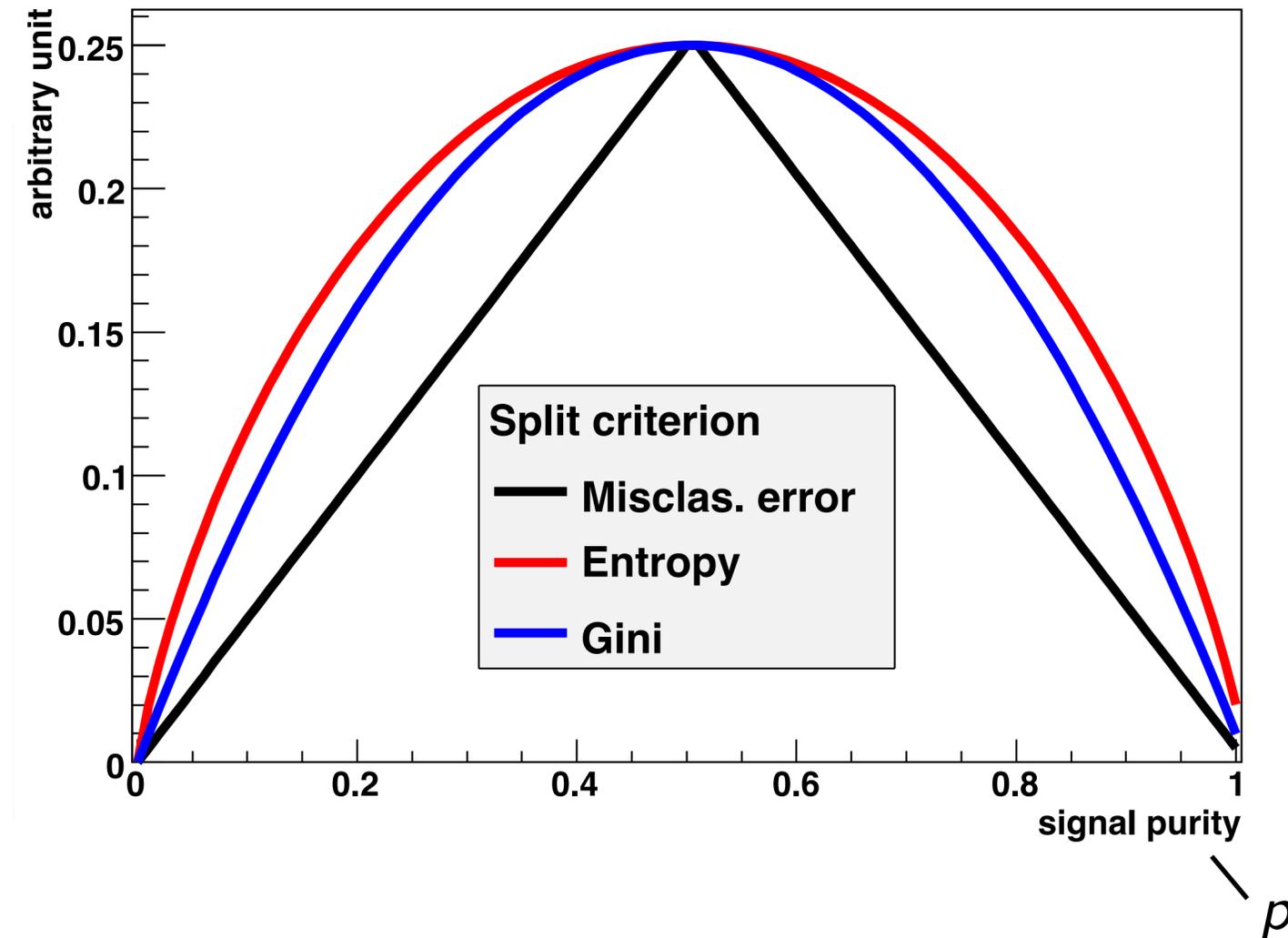
[usefulness of weights will become apparent soon]

Improvement in signal/background separation after splitting a set A into two sets B and C:

$$\Delta = W_A G_A - W_B G_B - W_C G_C \quad \text{where} \quad W_X = \sum_X w_i$$



Separation measures



Entropy:

$$-p \ln p - (1 - p) \ln(1 - p)$$

Gini index:

$$p(1 - p)$$

[after Corrado Gini, used to measure income and wealth inequalities, 1912]

Misclassification rate:

$$1 - \max(p, 1 - p)$$

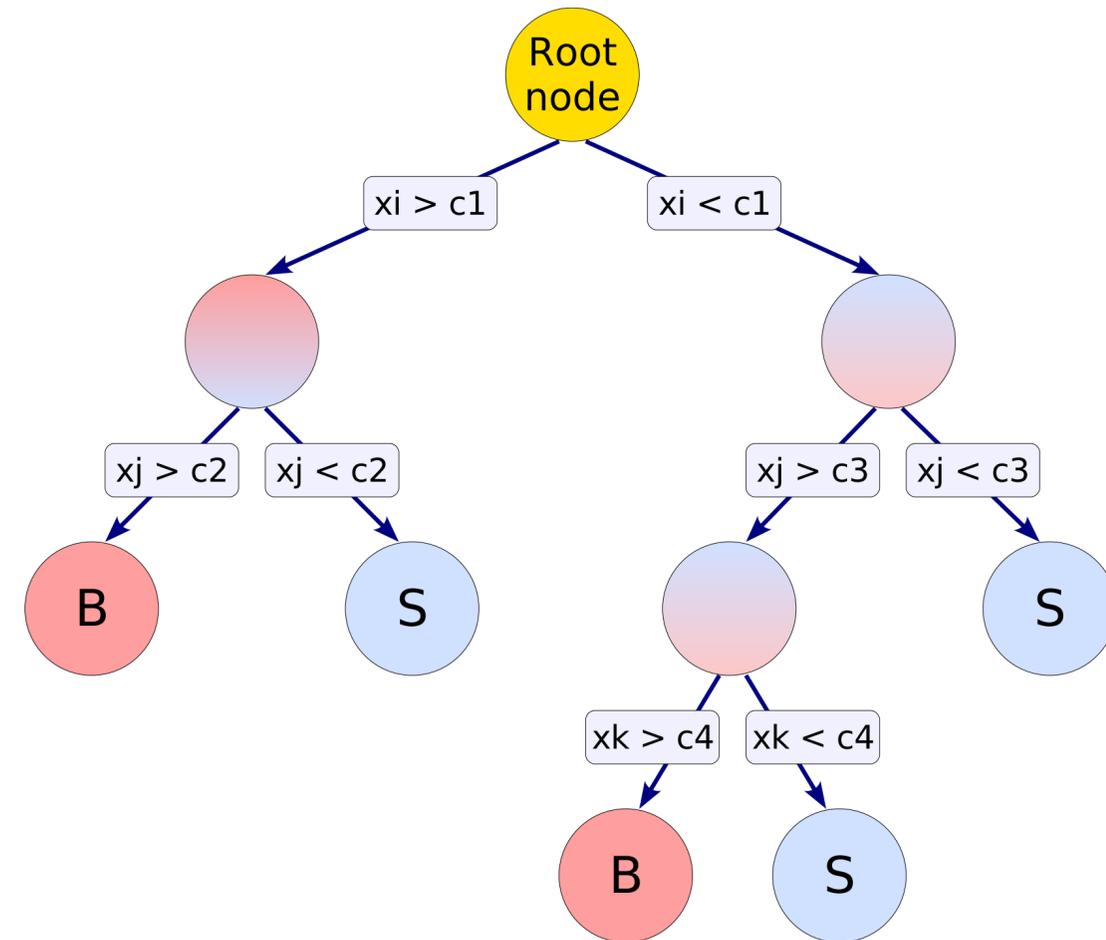
Decision tree pruning

When to stop growing a tree?

- ▶ When all nodes are essentially pure?
- ▶ Well, that's overfitting!

Pruning

- ▶ Cut back fully grown tree to avoid overtraining, i.e., replace nodes and subtrees by leaves



Single decision trees: Pros and cons

Pros:

- Requires little data preparation
- Can use continuous and categorical inputs

Cons:

- Danger of overfitting training data
- Sensitive to fluctuations in the training data
- Hard to find global optimum
- When to stop splitting?

Ensemble methods: Combine weak learners

- **Bootstrap Aggregating (Bagging)**

- ▶ Sample training data (with replacement) and train a separate model on each of the derived training sets
- ▶ Classify example with majority vote, or compute average output from each tree as model output

$$y(\vec{x}) = \frac{1}{N_{\text{trees}}} \sum_{i=1}^{N_{\text{trees}}} y_i(\vec{x})$$

- **Boosting**

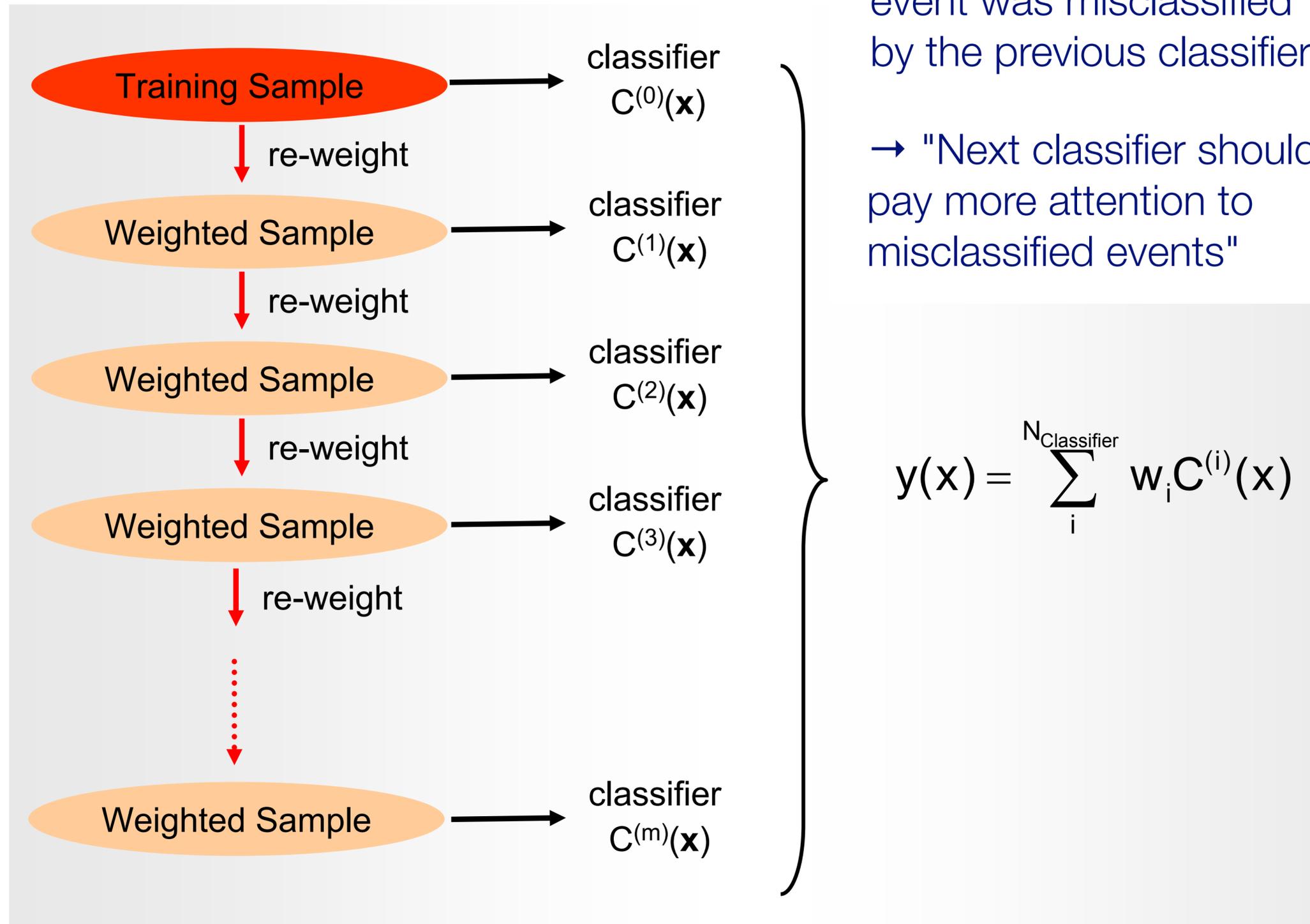
- ▶ Train N models in sequence, giving more weight to examples not correctly classified by previous model
- ▶ Take weighted average to classify examples

$$y(\vec{x}) = \frac{\sum_{i=1}^{N_{\text{trees}}} \alpha_i y_i(\vec{x})}{\sum_{i=1}^{N_{\text{trees}}} \alpha_i}$$

Random forests

- “One of the most widely used and versatile algorithms in data science and machine learning” (arXiv:1803.08823v3)
- Use bagging to select random example subset
- Train a tree, but only use random subset of features at each split
 - ▶ this reduces the correlation between different trees
 - ▶ makes the decision more robust to missing data

Boosted decision trees: Idea



H. Voss, Lecture: Graduierten-Kolleg, <http://tmva.sourceforge.net/talks.shtml>

AdaBoost (short for *Adaptive Boosting*)

Initial training sample

$\vec{x}_1, \dots, \vec{x}_n$: multivariate event data
 y_1, \dots, y_n : true class labels, +1 or -1
 $w_1^{(1)}, \dots, w_n^{(1)}$: event weights

with equal weights normalized as

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

Train first classifier f_1 :

$f_1(\vec{x}_i) > 0$ classify as signal
 $f_1(\vec{x}_i) < 0$ classify as background

AdaBoost: Updating event weights

Define training sample $k+1$ from training sample k by updating weights:

$$w_i^{(k+1)} = w_i^{(k)} \frac{e^{-\alpha_k f_k(\vec{x}_i) y_i / 2}}{Z_k}$$

$i = \text{event index}$ normalization factor so that $\sum_{i=1}^n w_i^{(k)} = 1$

Weight is increased if event was misclassified by the previous classifier

→ "Next classifier should pay more attention to misclassified events"

At each step the classifier f_k minimizes error rate

$$\varepsilon_k = \sum_{i=1}^n w_i^{(k)} I(y_i f_k(\vec{x}_i) \leq 0), \quad I(X) = 1 \text{ if } X \text{ is true, } 0 \text{ otherwise}$$

AdaBoost: Assigning the classifier score

Assign score to each classifier according to its error rate:

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

Combined classifier (weighted average):

$$f(\vec{x}) = \sum_{k=1}^K \alpha_k f_k(\vec{x})$$

It can be shown that the error rate of the combined classifier satisfies

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

Gradient boosting

- Like in AdaBoost, decision trees are iteratively added to an ensemble
- Can be applied to classification and regression
- Basic idea
 - ▶ Train a first decision tree
 - ▶ Then train a second one on the residual errors made by the first tree
 - ▶ And so on ...

Labeled training data: $\{\vec{x}_i, y_i\}$

Model prediction at iteration m : $F_m(\vec{x}_i)$

New model: $F_{m+1}(\vec{x}) = F_m(\vec{x}) + h_m(\vec{x})$

Find $h_m(\vec{x})$ by fitting it to

$$\{(\vec{x}_1, y_1 - F_m(\vec{x}_1)), (\vec{x}_2, y_2 - F_m(\vec{x}_2)), \dots, (\vec{x}_n, y_n - F_m(\vec{x}_n))\}$$

General remarks on multi-variate analyses (MVAs)

MVA Methods

- ▶ More effective than classic cut-based analyses
- ▶ Take correlations of input variables into account

Important: find good input variables for MVA methods

- ▶ Good separation power between S and B
- ▶ No strong correlation among variables
- ▶ No correlation with the parameters you try to measure in your signal sample!

Pre-processing

- ▶ Apply obvious variable transformations and let MVA method do the rest
- ▶ Make use of obvious symmetries: if e.g. a particle production process is symmetric in polar angle θ use $|\cos \theta|$ and not $\cos \theta$ as input variable
- ▶ It is generally useful to bring all input variables to a similar numerical range

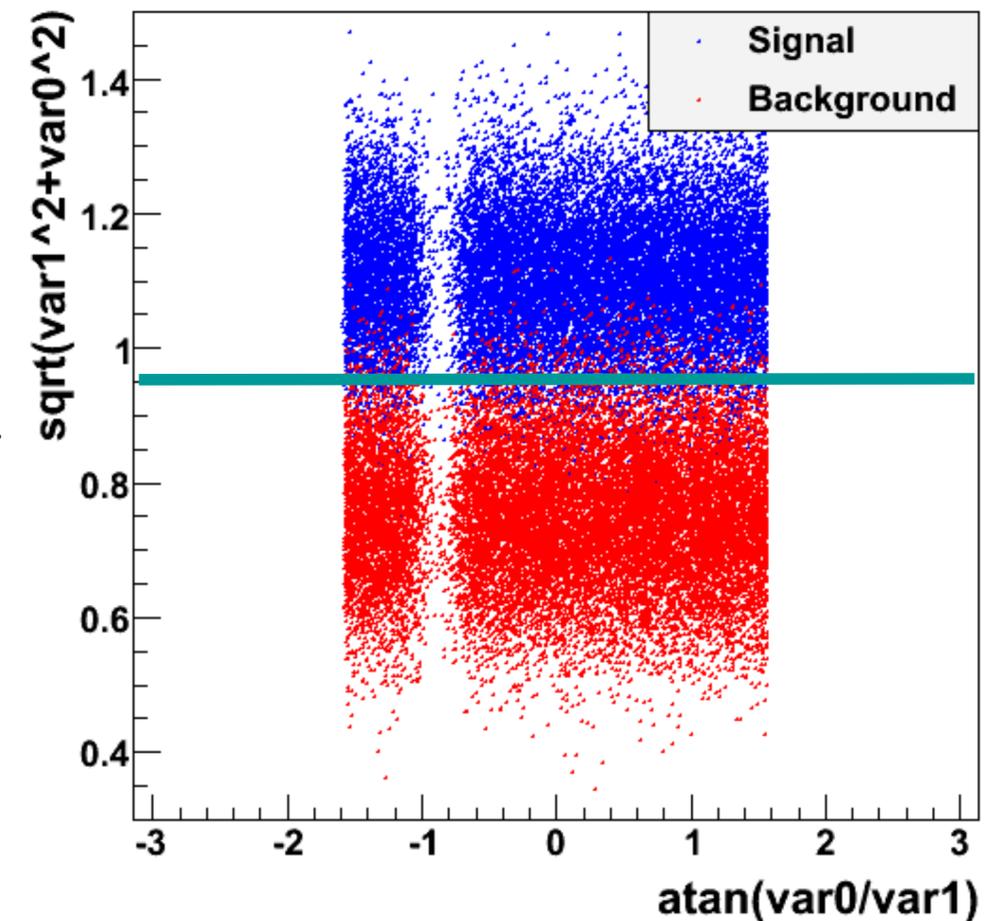
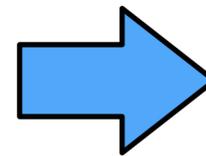
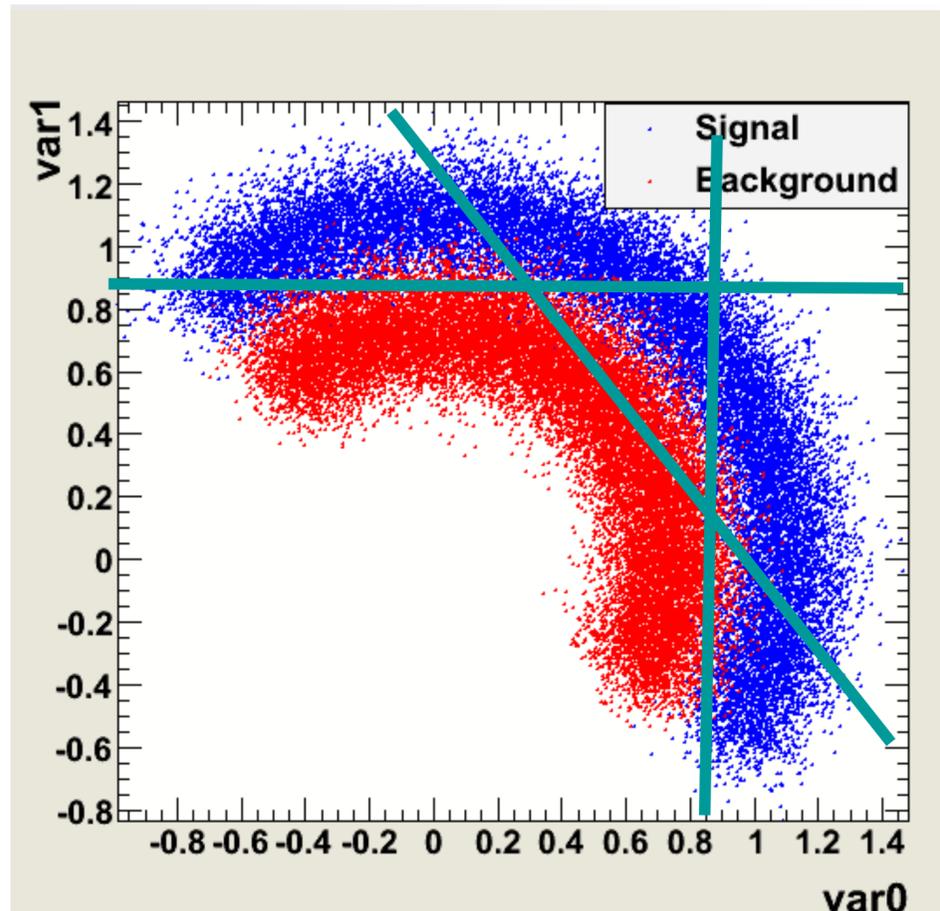
H. Voss, Multivariate Data Analysis and Machine Learning in High Energy Physics
<http://tmva.sourceforge.net/talks.shtml>

Example of a feature transformation

$$\text{var}0^l = \sqrt{\text{var}0^2 + \text{var}1^2}$$

$$\text{var}1^l = \text{atan}\left(\frac{\text{var}0}{\text{var}1}\right)$$

In this case a linear classifier works well after feature transformation



H. Voss, Multivariate Data Analysis and Machine Learning in High Energy Physics
<http://tmva.sourceforge.net/talks.shtml>

Which method to use?

M. Kagan,
<https://indico.cern.ch/event/619370/>

- Linear model
- Nearest Neighbors
- (Deep?) Neural network
- Decision tree ensemble
- ...

No free lunch theorem

David Wolpert, William Macready, 1997

<https://de.wikipedia.org/wiki/No-free-Lunch-Theoreme>

"Folkloric" version:

Any two optimization algorithms are equivalent when their performance is averaged across all possible problems

In other words:

If an algorithm performs well on a certain class of problems then it necessarily pays for that with degraded performance on the set of all remaining problems

How do we pay for our lunch?

Domain knowledge and/or biases in the choice of the algorithms
([link](#))

Relevance for practical problem?

Practical advice – Which algorithm to choose?

M. Kagan, <https://indico.cern.ch/event/619370/>

From Kaggle competitions:

Structured data: "High level" features that have meaning

- ▶ feature engineering + decision trees
- ▶ Random forests
- ▶ XGBoost

Unstructured data: "Low level" features, no individual meaning

- ▶ deep neural networks
- ▶ e.g. image classification: convolutional NN