Machine Learning

Classification - Part II - Beyond Decision Trees

Claudio Sartori

DISI
Department of Computer Science and Engineering – University of Bologna, Italy
claudio.sartori@unibo.it

Naive Bayes Classifier [2]

Main issues

- Based on statistics
  - In particular, on the Bayes' theorem
- Consider the contribution of all the attributes
- Assume that each attribute is independent from the others, given the class[1]
  - This is a very strong assumption, rarely verified, but, nevertheless, the method works!
- Estimate the probabilities with the frequencies, as usual

1 This means $\mathbf{Pr}\left(d_1 = v_1, d_2 = v_2 \mid c = c_x\right) = \mathbf{Pr}\left(d_1 = v_1 \mid c = c_x\right) \cdot \mathbf{Pr}\left(d_2 = v_2 \mid c = c_x\right)$

2 Description based on [Witten et al.(2011)Witten, Frank, and Hall]

The weather/play data

Numbers of cases and fractions of the Weather/Play dataset

Outlook			Temperature			Humidity			Windy			Play	
	yes	no		yes	no		yes	no		yes	no	yes	no
sunny	2	3	hot	2	2	high	3	4	false	6	2	9	5
overcast	4	0	mild	4	2	normal	6	1	true	3	3		
rainy	3	2	cool	3	1								
sunny	2/9	3/5	hot	2/9	2/5	high	3/9	4/5	false	6/9	2/5	9/14	5/14
overcast	4/9	0/5	mild	4/9	2/5	normal	6/9	1/5	true	3/9	3/5		
rainy	3/9	2/5	cool	3/9	1/5								

A new sample needs classification

Outlook: sunny, Temperature: cool, Humidity: high, Windy: true, Play: ?

- Treat the five features and the overall likelihood that play is yes or no as equally important
  - they are independent pieces of evidence, the overall likelihood is obtained by multiplying the probabilities (i.e. the frequencies)

$$\text{likelyhood of yes} = 2/9 \times 3/9 \times 3/9 \times 3/9 \times 9/14 = 0.0053$$
$$\text{likelyhood of no} = 3/5 \times 1/5 \times 4/5 \times 3/5 \times 5/14 = 0.0206$$

- Normalize to 1

$$\mathbf{Pr}\left(yes\right) = \frac{0.0053}{0.0053 + 0.0206} = 20.5\% \qquad \mathbf{Pr}\left(no\right) = \frac{0.0206}{0.0053 + 0.0206} = 79.5\%$$

- no is more likely than yes, about four times

The Bayes' theorem

Given a hypothesis H and an evidence E that bears on that hypothesis

$$\mathbf{Pr}(H \mid E) = \frac{\mathbf{Pr}(E \mid H)\,\mathbf{Pr}(H)}{\mathbf{Pr}(E)}$$

- The hypothesis is the class, say c, the evidence is the tuple of values of the element to be classified
- We can split the evidence into pieces, one per attribute, and, if the attributes are independent inside each class,

$$\mathbf{Pr}(c \mid E) = \frac{\mathbf{Pr}(E_1 \mid c) \times \mathbf{Pr}(E_2 \mid c) \times \mathbf{Pr}(E_3 \mid c) \times \mathbf{Pr}(E_4 \mid c) \times \mathbf{Pr}(c)}{\mathbf{Pr}(E)}$$

The Naive Bayes method

- Compute the conditional probabilities from examples
- Apply the theorem
- The denominator is the same for all the classes, and is eliminated by the normalization step
- It is called naive since the assumption of independence between attributes is quite simplistic
  - Nevertheless it works quite well in many cases

Problem

What if value v of attribute d never appears in the elements of class c?

- In this case $\mathbf{Pr}\left(d = v \,|\, c\right) = 0$
- This makes the probability of the class for that evidence drop to zero
- In practice, this case is quite common, in particular in a domain with many attributes and many distinct values
- An alternative solution is needed

Values not represented in a class – Laplace smoothing

Optional

α – Smoothing parameter, typical value is 1

$af_{d=v_i,c}$ – Absolute frequency of value v_i in attribute d over class c

V – number of distinct values in attribute d over the dataset

af_c – Absolute frequency of class c in the dataset

$$\text{Smoothed frequency } sf_{d=v_i,c} = \frac{af_{d=v_i,c} + \alpha}{af_c + \alpha V}$$

- With $\alpha = 0$ we obtain the standard, unsmoothed formula
- Higher values of α give more importance to the prior probabilities for the values of d w.r.t. the evidence given by the examples

Missing values

Optional

They do not affect the model, it is not necessary to discard an instance with missing value(s)

- Test instance:
  - The calculation of the likelyhood simply omits this attribute
  - The likelihood will be higher for all the classes, but this is compensated by the normalization
- Train instance:
  - The record is simply not included in the frequency counts for that attribute
  - The descriptive statistics are based on the number of values that occur, rather than on the number of instances

Numeric values

- The method based on frequencies is inapplicable
- Additional assumption: the values have a Gaussian distribution
- Instead of the fractions of counts, we compute, from the examples the mean μ and the variance σ of the values of each numeric attribute inside each class
- For a given attribute and a given class, the distribution is supposed to be

$$f(x) = \frac{1}{\sqrt{2\pi\sigma}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

The weather/play data with numeric values

Optional

Numbers of cases, fractions and descriptive statistics of the Weather/Play dataset

Outlook			Temperature			Humidity			Windy			Play	
	yes	no		yes	no		yes	no		yes	no	yes	no
sunny	2	3		83	85		86	85	false	6	2	9	5
overcast	4	0		70	80		96	90	true	3	3		
rainy	3	2		68	65		80	70					
				64	72		65	95					
				69	71		70	91					
				75			80						
				75			70						
				72			90						
				81			75						
sunny	2/9	3/5	mean	73	74.6	mean	79.1	86.2	false	6/9	2/5	9/14	5/14
overcast	4/9	0/5	stdev	6.2	7.9	stdev	10.2	9.7	true	3/9	3/5		
rainy	3/9	2/5											

Using numeric values in Naive Bayes

OPTIONAL

- We are considering a yes outcome when the temperature is 66
- Plug the the value under consideration, the mean and the stdev in the gaussian probability density formula

$$f(\text{temperature} = 66|\text{yes}) = \frac{1}{\sqrt{2\pi} \cdot 6.2} e^{-\frac{(66-73)^2}{2 \cdot 6.2^2}} = 0.0340$$

$$f(\text{humidity} = 90|\text{yes}) = 0.0221$$

Using numeric values in Naive Bayes

OPTIONAL

Probability and probability density are closely related, but are not the same thing

- On a continuous domain, the probability of a variable assuming exactly a single real value is zero
- A value of the density function is the probability that the variable lies in a small interval around that value
- The value we use are, of course, rounded at some precision factor
- That precision factor is the same for all the classes, then we can disregard it
- If numeric values are missing, mean and standard deviation are based only on the values that are present

Classification of a sample with numeric values Optional

Outlook: sunny, Temperature: 66, Humidity: 90, Windy: true, Play: ?

$$\text{likelyhood of yes} = 2/9 \times 0.0340 \times 0.0221 \times 3/9 \times 9/14 = 0.000036$$

$$\text{likelyhood of no} = 3/5 \times 0.0221 \times 0.0381 \times 3/5 \times 5/14 = 0.000108$$

$$\mathbf{Pr}\left(yes\right) = \frac{0.000036}{0.000036 + 0.000108} = 25.0\%$$

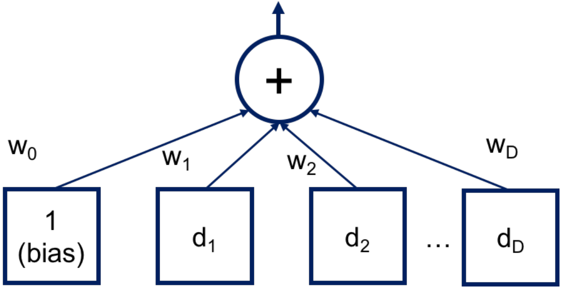
$$\mathbf{Pr}\left(no\right) = \frac{0.000108}{0.000036 + 0.000108} = 75.0\%$$

Summary

- Clear semantics for learning probabilistic knowledge
- Excellent results in many cases
- Dramatic degradation when the simplistic conditions are not met
  - Violation of independence – for instance, if an attribute is simply a copy of another (or a linear transformation), the weight of that particular feature is enforced (something like squaring the probability)
  - Violation of gaussian distribution – use the standard probability estimation for the appropriate distribution, if known, or use estimation procedures, such as Kernel Density Estimation

The linear perceptron [3]

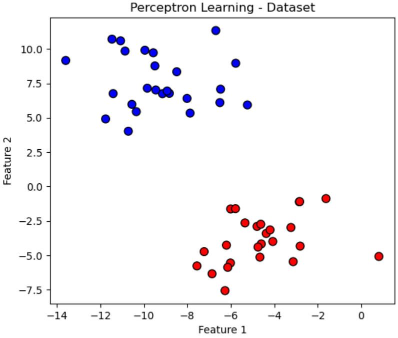
- Often called also artificial neuron
- In practice, a linear combination of weighted inputs



3 Description based on [Witten et al.(2011)Witten, Frank, and Hall]

Separate examples of two classes

- For a dataset with numeric attributes
- Learn a hyperplane such that all the positives lay on one side and all the negatives on the other



The hyperplane

- The hyperplane is described by a set of weights $w_0, \ldots, w_D$ in a linear equation on the data attributes $x_0, \ldots, x_D$
  - the fictitious attribute $x_0 = 1$ is added to allow a hyperplane that does not pass through the origin
- There are either none or infinite such hyperplanes

$$w_0 * x_0 + w_1 * x_1 + \ldots + w_D * x_D \quad \begin{cases} > 0 \Rightarrow \text{positive} \\ < 0 \Rightarrow \text{negative} \end{cases}$$

Learning the hyperplane

Algorithm 1: Perceptron learning of a separating hyperplane

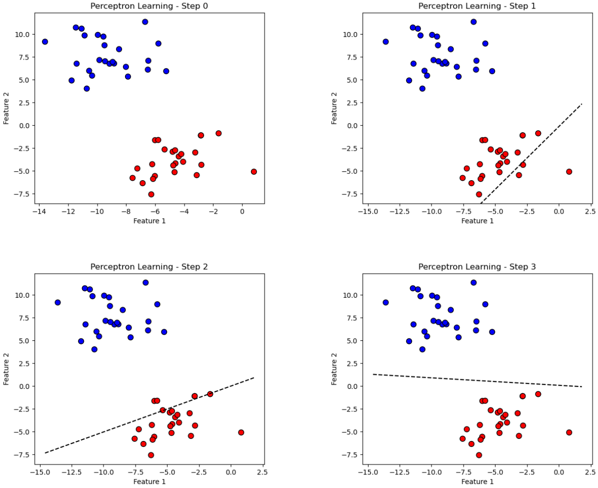
Data: Training set $\mathcal{X} = \{(x_i, y_i)\}$
Result: Weight vector **w** defining the hyperplane

```
Initialize all weights w ← 0;
while there are examples incorrectly classified do
    foreach training instance (x_i, y(x)_i) ∈ D do
        if (x_i, y(x)_i) is incorrectly classified then
            if y(x)_i is positive then
                w ← w + x_i;
            else
                w ← w − x_i;
            end
        end
    end
end
```

Linear perceptron convergence

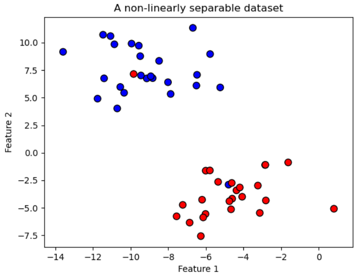
- Each change of weights moves the hyperplane towards the misclassified instance, consider the equation after the weight change for a positive instance x_i which was classified as negative $(w_0 + x_{i0}) * x_{i0} + (w_1 + x_{i1}) * x_{i1} + \ldots + (w_D + x_{iD}) * x_{iD}$
- The result of the equation is increased by a positive amount $x_{i0}^2 + \ldots + x_{iD}^2$
- Therefore the result will be less negative or, possibly, even positive
- Analogously for a negative instance which was classified as positive

Perceptron learning



Linear perceptron algorithm termination

- The corrections are incremental and can interfere with previous updates
- The algorithm converges if the dataset is linearly separable, otherwise it does not terminate
- For practical applicability it is necessary to set an upper bound to the iterations



Support Vector Machines (SVM) for binary classification I

Limitations of the linear models

Description of SVM based on [Witten et al.(2011)Witten, Frank, and Hall]

- What can we do if the data are not linearly separable?
  - This means that the boundary between classes is some hyper–surface more complex than a hyperplane
- One possibility would be to give up the linearity, e.g.:

$$w_1 * x_1^3 + w_2 * x_1^2 * x_2 + w_3 * x_1 * x_2^2 + w_4 * x_3^3$$

Support Vector Machines (SVM) for binary classification II

Limitations of the linear models

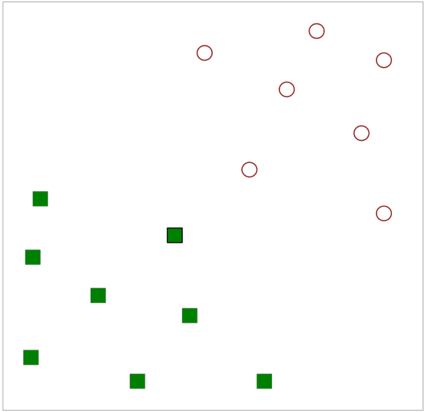
Description of SVM based on [Witten et al.(2011)Witten, Frank, and Hall]

- The method would become soon intractable for any reasonable number of variables
  - with 10 variables and limiting to factors with maximum order 5 we would need something like 2000 coefficient
- The method would be extremely prone to overfitting, if the number of parameters approaches the number of examples

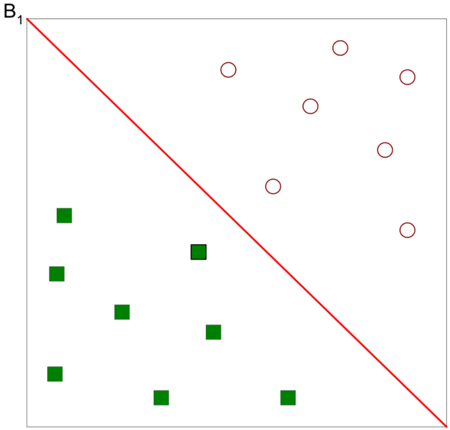
Key ideas

- Computational learning theory
- New efficient separability of non–linear functions that use kernel functions
- Optimization rather than greedy search
- Statistical learning
  - The search of a predition function is modeled as a function estimation problem

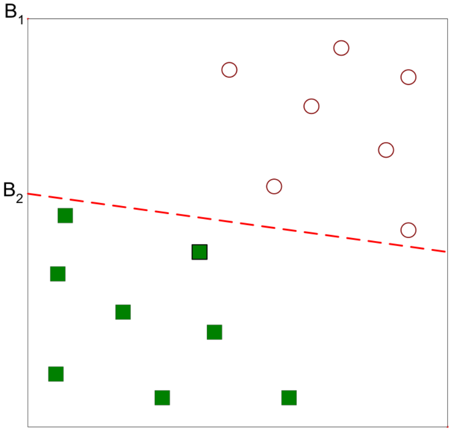
The Maximum Margin hyperplane



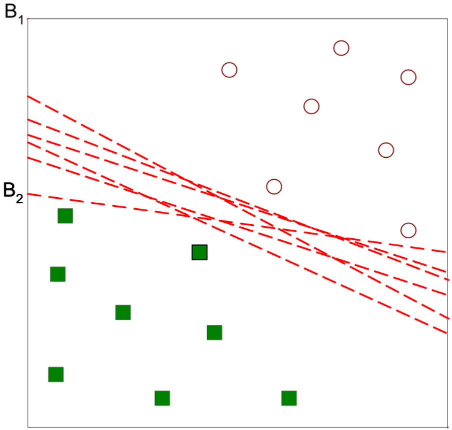
The Maximum Margin hyperplane



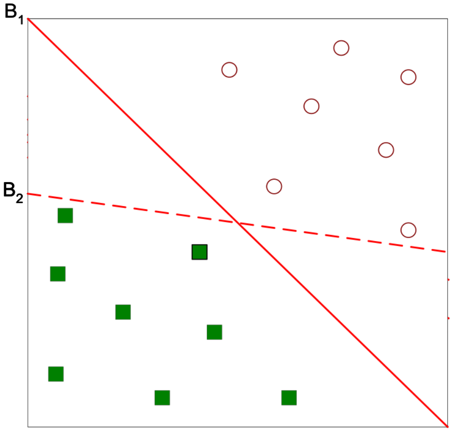
The Maximum Margin hyperplane



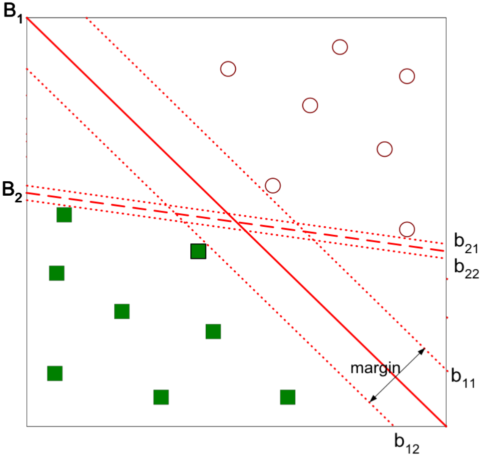
The Maximum Margin hyperplane



The Maximum Margin hyperplane

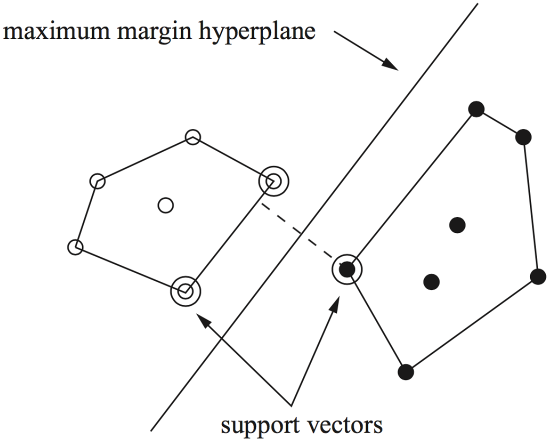


The Maximum Margin hyperplane



Maximum Margin Hyperplane

- The linear perceptron accepts any hyperplane able to separate the classes of the training examples
  - It is conceivable that some hyperplanes are better than others for the classification of new items
- The maximum margin hyperplane gives the greatest separation between the classes



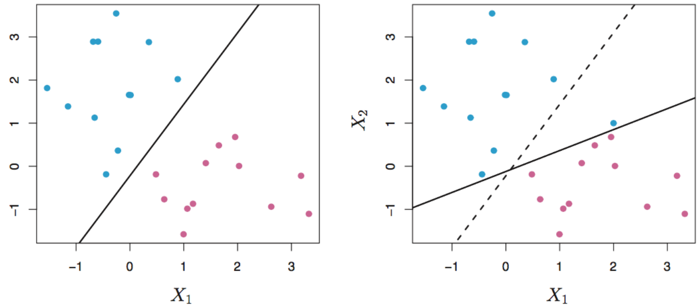
Maximum Margin Hyperplane (continued)

OPTIONAL

- The convex hull of a set of points is the tightest enclosing convex polygon
  - if the dataset is linearly separable the convex hulls of the classes do not intersect
- The maximum margin hyperplane is as far as possible from both the hulls
  - it is the perpendicular bisector of the shortest line connecting the hulls
- In general a subset of the points is sufficient to define the hull
  - those are the support vectors (circled in the figure of the previous page)
- Support vectors are the elements of the training set which would change
- Finding the support vectors and the maximum margin hyperplane belongs to the well known class of constrained quadratic

Soft margin

- It is quite common that a separating hyperplane does not exists
  - see figure in slide24 in the perceptron section
- $\Rightarrow$ find an hyperplane which almost separate the classes
- $\Rightarrow$ disregard examples which generate a very narrow margin



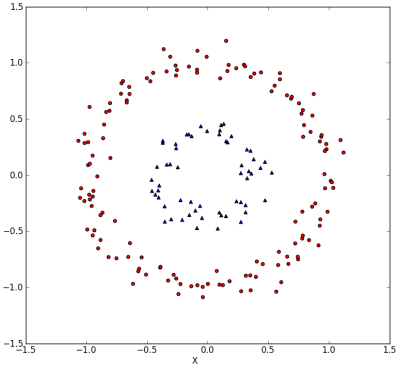
Soft margin Support Vector classifier

- Greater robustness to individual observations
- Better classification of most of the training observations
- Obtained by adding a constraint to the optimization problem expressed by a single numeric parameter, usually called C in the literature
  - C, the penalty parameter of the error term, controls the amount of overfitting
  - C tuning is critical

Non–linear class boundaries I

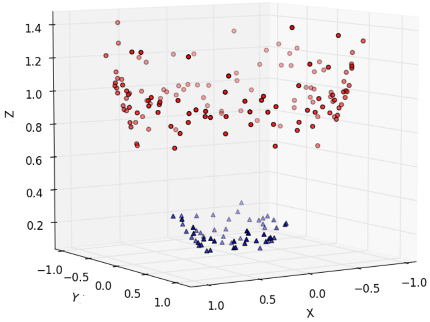
- The support vector method avoids the problem of overfitting
- The nonlinearity of boundaries can be overcome with a non–linear mapping
  - the data are mapped into a new space, usually called feature space, such that a linear boundary in the feature space can correspond to a non–linear boundary in the original space
  - the feature space can have a number of dimensions higher than the original one

Non–linear class boundaries II



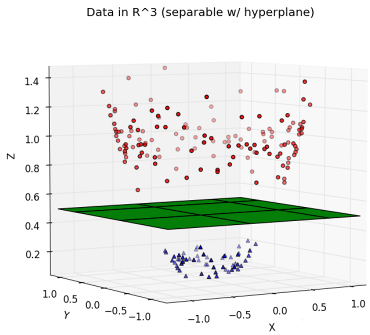
Input space

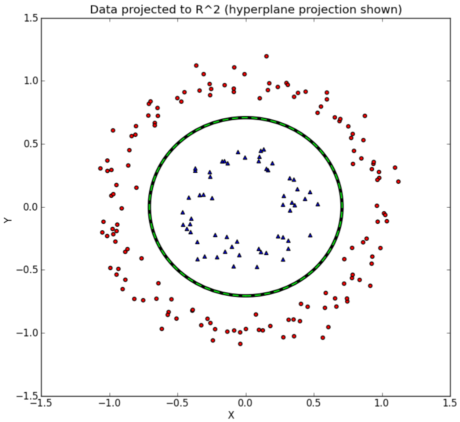
$\varphi[X,Y] = [X,Y,X^2+Y^2]$



Feature space

Non–linear class boundaries III





Now linearly separable

The kernel trick

OPTIONAL

- The separating hyperplane computation requires a series of dot product computations among the training data vectors
- Defining the mapping on the basis of a particular family of functions, called kernel functions, or simply kernels, the mapping does not need to be explicitly computed, and the computation is done in the input space
- This avoids an increase in the complexity
- Some kernel functions:

$$
\begin{array}{rl}
\text{linear} & \langle x, x' \rangle \\
\text{polynomial} & (\gamma \langle x, x' \rangle + r)^{dg} \\
\text{rbf} & \exp(-\gamma \|x - x'\|^2) \\
\text{sigmoid} & \tanh(\langle x, x' \rangle + r)
\end{array}
$$

- γ, dg, r are parameters specified in the documentation of the learning tools
- Rule of thumb: start with the simpler and then try with the more complex if necessary

Choosing Activation Functions: Rules of Thumb I

- ReLU (Rectified Linear Unit)
  - Default choice for most deep networks.
  - Fast convergence due to linear, non-saturating behavior.
  - May suffer from "dying ReLU" (neurons stuck at 0).
  - Use LeakyReLU or ELU as alternatives if neurons die.
- Sigmoid
  - Maps input to (0, 1), good for probabilities.
  - Causes vanishing gradients for large $|x|$.
  - Use mainly in the output layer for binary classification.

Choosing Activation Functions: Rules of Thumb II

- Tanh (Hyperbolic Tangent)
  - Zero-centered output in (-1, 1).
  - Often better than sigmoid for hidden layers in shallow nets.
  - Still suffers from vanishing gradients in deep nets.
- Arctan
  - Smooth and bounded, similar to tanh but slower to saturate.
  - Sometimes used in experimental or small networks.
- Softmax
  - Use in the output layer for multi-class classification.
  - Converts raw scores into normalized probabilities.

Activation Function Comparison Table

Activation	Output Range	Derivative Range	Zero-Centered?	Common Use
Sigmoid	(0, 1)	(0, 0.25)	No	Binary output, probability modeling
Tanh	(-1, 1)	(0, 1)	Yes	Hidden layers (small networks)
ReLU	[0, +inf)	0, 1	No	Default for deep hidden layers
LeakyReLU	(-inf, +inf)	alpha, 1	Partially	Fix for dying ReLU
Arctan	(-pi/2, pi/2)	(0, 1)	Yes	Smooth alternative to tanh
Softmax	(0, 1), sum=1	Complex	No	Multi-class output layers

Quick Guidelines:

- Start with ReLU for hidden layers.
- Use Sigmoid or Softmax in output layers (classification).
- Try Tanh or LeakyReLU if gradients vanish or neurons die.

Choosing Output Activations Depending on the Task I

- Regression Tasks
  - Predict continuous values (e.g., temperature, price).
  - Common output activation: None (linear).
  - Example: $\hat{y} = Wx + b$.
- Binary Classification
  - Output: probability of positive class.
  - Common output activation: Sigmoid.
  - Example: $\hat{y} = \sigma(Wx + b)$, with $\sigma(x) = 1/(1 + e^{-x})$.

Choosing Output Activations Depending on the Task II

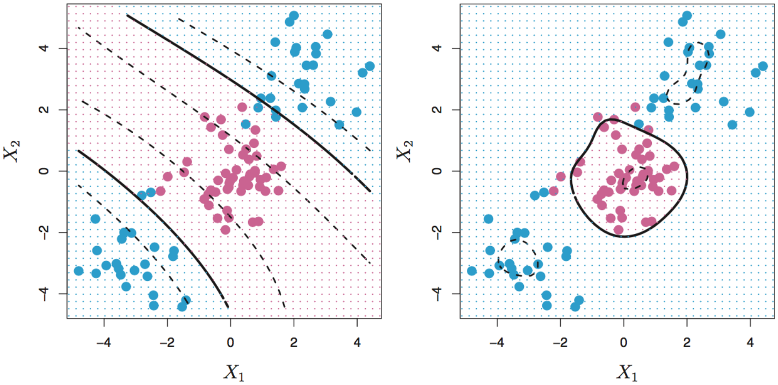
- **Multi-Class Classification**
  - Output: probability distribution over classes.
  - Common output activation: Softmax.
  - Example: $\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$.
- **Multi-Label Classification**
  - Each output node is independent (can be 1 or 0).
  - Common output activation: Sigmoid (per node).
  - Loss: Binary cross-entropy per output unit.

Choosing Output Activations Depending on the Task III

Summary: Use ReLU (or variant) for hidden layers, choose the output activation according to the prediction type: linear for regression, sigmoid for binary, softmax for multi-class.

Examples of decision boundaries with kernels

Optional



Decision boundaries for polynomial kernel of degree (left) and radial based kernel (RBF, right)
In this case they seem to be both appropriate, but the generalization capabilities change, depending on where new data could be expected

SVM Complexity

OPTIONAL

- The time complexity is mainly influenced by the efficiency of the optimization library
- The popular `libSVM` library scales from $\mathcal{O}(D * N^2)$ to $\mathcal{O}(D * N^3)$, depending on the effectiveness of data caching in the library, which is data dependent
  - in case of sparse data it is reduced

Final remarks on SVM

- Learning is in general slower than simpler methods, such as decision trees
- Tuning is necessary to set the parameters (not discussed here)
- The results can be very accurate, because subtle and complex decision boundaries can be obtained
- Explicitely based on a theoretical model of learning
- Are not affected by local minima
- Do not suffer from the curse of dimensionality: do not use any notion of distance
- SVMs do not directly provide probability estimates, these can be calculated using rather expensive estimation techniques (see `scikit-learn` documentation in References below)
  - nevertheless, SVM can produce a confidence score related to the distance of an example from the separation hyperplane

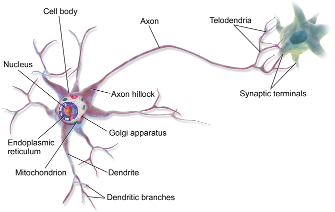
SVM - References

Optional

1. See reference [James et al.(2015)James, Witten, Hastie, and Tibshirani] for a more detailed explanation of the maths behind SVM (unfortunately it is not publicly available, ask the teacher for a temporary loan)
2. Jordan, Michael I., and Romain Thibaux. "The Kernel Trick." Lecture Notes. 2004. Web. 5 Jan. 2013. https://people.eecs.berkeley.edu/j̃ordan/courses/281B-spring04/lectures/lec3.pdf
3. Berwick, Robert. "An Idiot's Guide to Support Vector Machines (SVMs)". Lecture Slides. 2003. Web. 5 Jan. 2013. http://www.cs.ucf.edu/courses/cap6412/fall2009/papers/Berwick2003.pdf
4. "Scikit-learn: sklearn.svm.SVC Documentation", Pedregosa et al. http://scikit-learn.org/dev/modules/generated/sklearn.svm.SVC.html
5. Rifkin, Ryan. "Multiclass Classification". Lecture Slides. February 2008. Web. 6 Jan. 2013. http://www.mit.edu/ 9.520/spring09/Classes/multiclass.pdf
6. Hofmann, Martin. "Support Vector Machines – Kernels and the Kernel Trick". Notes. 26 June 2006. Web. 7 Jan. 2013. http://www.cogsys.wiai.uni-bamberg.de/teaching/ss06/hs_svm/slides/SVM_Seminarbericht_Hofmann.pdf

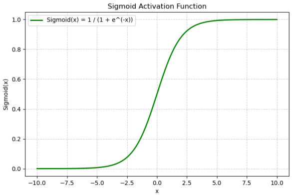
Neural Networks

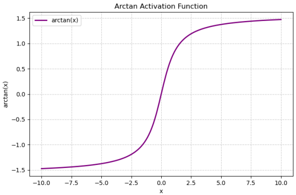
- Arrange many perceptron–like elements in a hierarchical structure
  - Another way to overcome the limit of linear decision boundary
- Inspired to the complex interconnections of neurons in animal brains
- A neuron is a signal processor with threshold
- Signal transmission from one neuron to another is weighted
  - weights change over time, also due to learning

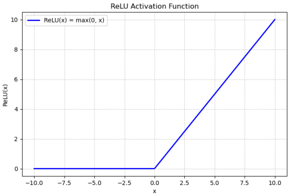


Multi–layer perceptron

- The signals transmitted are modeled as real numbers
- The threshold of the biological system is modeled as a mathematic function
    - continuous and differentiable, superiorly and inferiorly limited
    - the derivative can be expressed in terms of the function itself
        - this simplifies the mathematics
- Several functions available, here some of them



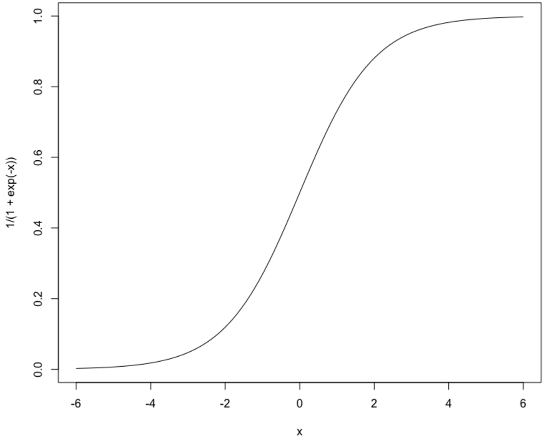




Sigmoid

- Also called squashing function
- Maps reals into $]0, 1[$
- Is continuous, differentiable, non–linear

$$\frac{1}{1 + e^{-x}}$$



Importance of non–linearity

Optional

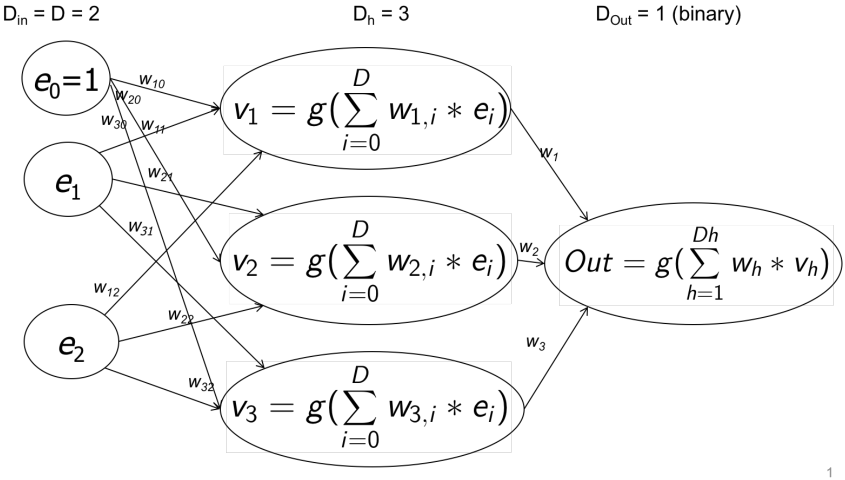
- Results with the linear perceptron were non satisfactory because of linearity
  - in addition to the problem of separability
- In a linear system $f(x1 + x2) = f(x1) + f(x2)$
  - if $x2$ is generated by noise, it is completely transferred to the output
- In a non–linear system, in general, $f(x1 + x2) \neq f(x1) + f(x2)$
- The shape of the function can influence the learning speed

Feed–forward multi–layered network

- Inputs feed an input layer
  - one input node for each dimension in the training set
- Input layer feeds (with weights) a hidden layer
- Hidden layer feeds (with weights) an output layer
  - the number of nodes in the hidden layer are a parameter of the network
  - the number of nodes in the output layer is related to number of different classes in the domain
    - one node if there are two classes
    - one node per class in the other cases[4]

4 As an alternative, it is possible to adopt the OVO or OVA architecture, see page 103

Feed–forward multi–layered network – Example



Details

- $g(.)$ is the transfer function of the node, e.g. the sigmoid
- The unitary input x_0 is added for dealing with the bias, as in the case of the linear perceptron
- The weights are for each edge connecting two nodes
- Feed–forward defines which oriented edges are present
  - edges connect only a node in a layer to a node in the following layer
    - input to hidden and hidden to output
  - each node of one layer is connected to all the nodes of the following layer
- In this way the signal flows from the input to the output, without loops

Training the Neural Network I

Algorithm 2: Training a neural network (backpropagation procedure)

Data: Training set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$
Result: Trained neural network weights

Initialize all weights to random values;
while termination condition is not satisfied **do**
    foreach training instance $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{D}$ **do**
        Feed the network with $\mathbf{x}_i$ and compute the output $nn(\mathbf{x}_i)$;
        Compute the weight corrections for the error $nn(\mathbf{x}_i) - \mathbf{y}_i$;
        Propagate back the weight corrections;
    end
end

Training the Neural Network II

- In analogy with learning in the animal domain, the examples must repeatedly feed the network
- The weights encode the knowledge given by the supervised examples
- The encoding is not easily understandable: it looks like a structured set of real numbers
- Convergence is not guaranteed
- Important issues:
  - computing the weight corrections
  - preparation of the training examples
    - standardize the attributes to have zero mean and unit variance
  - termination condition

Computing the error

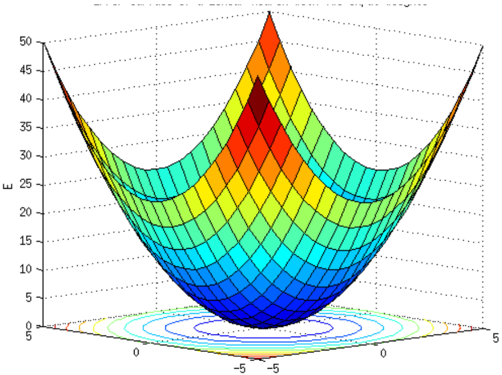
Optional

- Let $\mathbf{x}$ and y be the input vector and the desired output of a node, respectively
- Let $\mathbf{w}$ be the input weight vector of a node
- The error is:

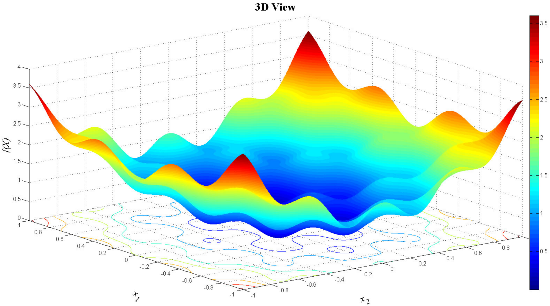
$$E(\mathbf{w}) = \frac{1}{2}(y - \text{Transfer}(\mathbf{w}, \mathbf{x}))^2$$

Error functions

Optional



Convex error function



Non convex error function

Computing the gradient I

OPTIONAL

- Move towards a (local) minimum of the error
  - follow the gradient
  - compute the partial derivatives of the error as a function of the weights

$$\mathrm{sgm}(x) = \frac{1}{1+e^{-x}}$$
$$\frac{d}{dx}\mathrm{sgm}(x) = \frac{e^{-x}}{(1+e^{-x})^2} = (1-\mathrm{sgm}(x))\mathrm{sgm}(x)$$

Computing the gradient II

OPTIONAL

- The weight is changed subtracting the partial derivative multiplied by a learning rate constant
  - the learning rate influences the convercence speed and can be adjusted as a tradeoff between speed and precision
- The subtraction moves towards smaller errors
- The derivatives of the input weights of the nodes of a layer can be computed if the derivatives for the following layer are known
- The actual derivatives are omitted here

$$w_{ij} \leftarrow w_{ij} - \lambda \frac{\partial E(\mathbf{w})}{\partial w_{ij}}$$

Training Algorithm – Revised

Algorithm 3: Revised training algorithm for neural networks

Data: Training set $\mathcal{D} = \{(\mathbf{x}_i, \mathbf{y}_i)\}$
Result: Updated neural network weights

Initialize all weights to random values;
while termination condition is not satisfied **do**
  foreach training instance $(\mathbf{x}_i, \mathbf{y}_i) \in \mathcal{D}$ **do**
    `// Forward phase`
    1 – Feed the network with $\mathbf{x}_i$ and compute the output $nn(\mathbf{x}_i)$;
    2 – Compute error prediction at output layer: $nn(\mathbf{x}_i) - \mathbf{y}_i$;
    `// Backward phase`
    3 – Compute derivatives and weight corrections for the output layer;
    4 – Compute derivatives and weight corrections for the hidden layer;
  end
end

Steps 1 and 2 are forward; steps 3 and 4 are backward.

Learning modes

Optional

Stochastic
- each forward propagation is immediately followed by a weight update (as in the algorithm of previous slide)
- introduces some noise in the gradient descent process, since the gradient is computed from a single data point
- reduces the chance of getting stucked in a local minimum
- good for online learning

Batch
- many propagations occur before updating the weights, accumulating errors over the samples within a batch
- generally yelds faster and stable descent towards the local minimum, since the update is performed in the direction of the average error

Repetitions

- A learning round over all the samples of the network is called epoch
- In general, after each epoch the network classification capability will be improved
- Several epochs will be necessary
- After each epoch the starting weights will be different

Design choices

- The structure of input and output layers is determined by the domain (the training set)
- The number of nodes in the hidden layer can be changed
- The learing rate can be changed in different epochs
  - in the begining a higher learning rate can push faster towards the desired direction
  - in later epochs a lower learning rate can push more precisely towards a minimum

Stop criteria

- All the weight updates in the epoch have been small
- The classification error rate goes below a predefined target
- A timeout condition is reached

Risks

- Local minima are possible, as usual in gradient tracking methods
- Overfitting is possible, if the network is too complex w.r.t. the complexity of the decision problem

Regularization

- Technique used in many machine learning functions to improve the generalisation capabilities of a model
- Modify the performance function, which is normally chosen to be the sum of squares of the network errors on the training set
- In essence:
  - improvement of performance is obtained by reducing a loss function (i.e. the sum of squared errors)
  - regularisation corrects the loss function in order to smooth the fitting to the data
  - the amount of regularisation must be tuned

Choosing Activation Functions: Rules of Thumb I

- ReLU (Rectified Linear Unit)
  - Default choice for most deep networks.
  - Fast convergence due to linear, non-saturating behavior.
  - May suffer from "dying ReLU" (neurons stuck at 0).
  - Use LeakyReLU or ELU as alternatives if neurons die.
- Sigmoid
  - Maps input to (0, 1), good for probabilities.
  - Causes vanishing gradients for large $|x|$.
  - Use mainly in the output layer for binary classification.

Choosing Activation Functions: Rules of Thumb II

- Tanh (Hyperbolic Tangent)
  - Zero-centered output in (-1, 1).
  - Often better than sigmoid for hidden layers in shallow nets.
  - Still suffers from vanishing gradients in deep nets.
- Arctan
  - Smooth and bounded, similar to tanh but slower to saturate.
  - Sometimes used in experimental or small networks.
- Softmax
  - Use in the output layer for multi-class classification.
  - Converts raw scores into normalized probabilities.

Activation Function Comparison Table

Activation	Output Range	Derivative Range	Zero-Centered?	Common Use
Sigmoid	(0, 1)	(0, 0.25)	No	Binary output, probability modeling
Tanh	(-1, 1)	(0, 1)	Yes	Hidden layers (small networks)
ReLU	[0, +inf)	0, 1	No	Default for deep hidden layers
LeakyReLU	(-inf, +inf)	alpha, 1	Partially	Fix for dying ReLU
Arctan	(-pi/2, pi/2)	(0, 1)	Yes	Smooth alternative to tanh
Softmax	(0, 1), sum=1	Complex	No	Multi-class output layers

Quick Guidelines:

- Start with ReLU for hidden layers.
- Use Sigmoid or Softmax in output layers (classification).
- Try Tanh or LeakyReLU if gradients vanish or neurons die.

Choosing Output Activations Depending on the Task I

- **Regression Tasks**
  - Predict continuous values (e.g., temperature, price).
  - Common output activation: None (linear).
  - Example: $\hat{y} = Wx + b$.
- **Binary Classification**
  - Output: probability of positive class.
  - Common output activation: Sigmoid.
  - Example: $\hat{y} = \sigma(Wx + b)$, with $\sigma(x) = 1/(1 + e^{-x})$.

Choosing Output Activations Depending on the Task II

- Multi-Class Classification
  - Output: probability distribution over classes.
  - Common output activation: Softmax.
  - Example: $\hat{y}_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$.
- Multi-Label Classification
  - Each output node is independent (can be 1 or 0).
  - Common output activation: Sigmoid (per node).
  - Loss: Binary cross-entropy per output unit.

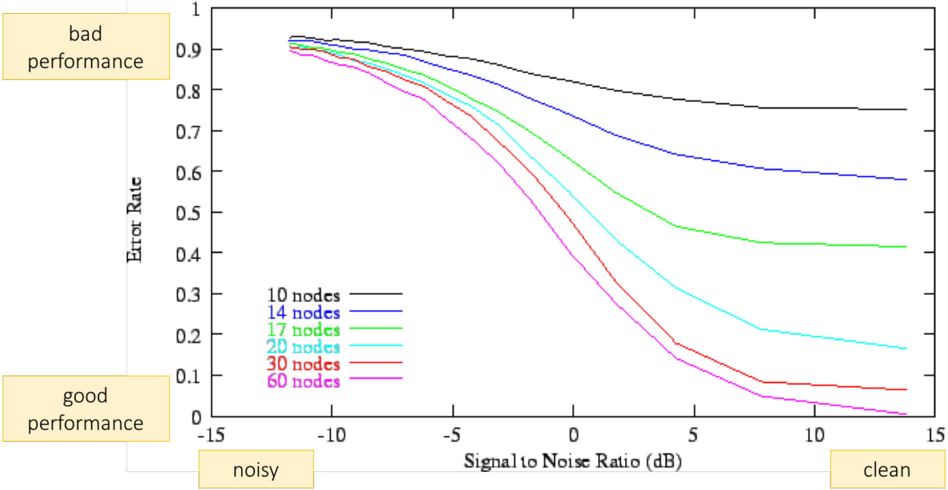
Choosing Output Activations Depending on the Task III

Summary: Use ReLU (or variant) for hidden layers, choose the output activation according to the prediction type: linear for regression, sigmoid for binary, softmax for multi-class.

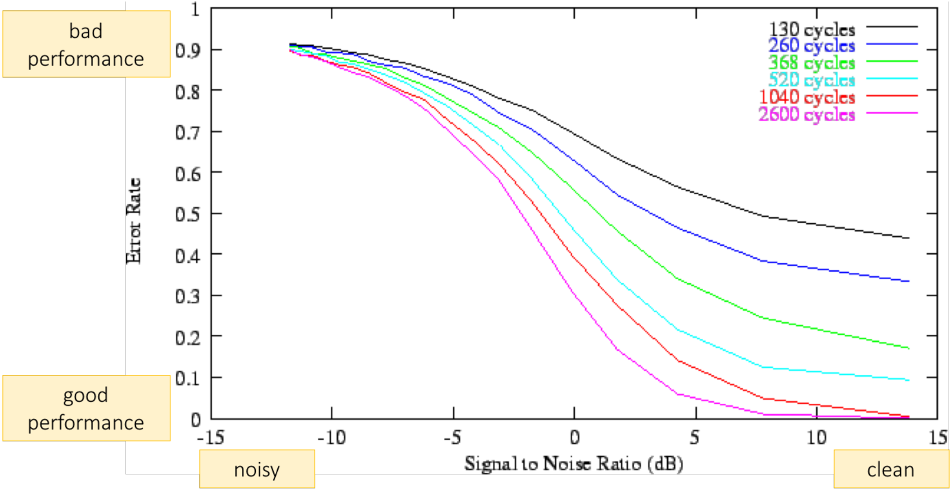
Example

- Recognition of characters with noise
- The images are 7x5 arrays of floating points
  - before noise -1 for black, 1 for white
- 35 input nodes
- up to 60 hidden nodes
- 26 output nodes
  - only one at a time should give a value near 1, all the others should be near 0
  - a good alternative could be a 5 bit coding

Error rate – Varying signal/noise and hidden nodes



Error rate – Varying signal/noise and number of epochs



Universal Approximation Theorem

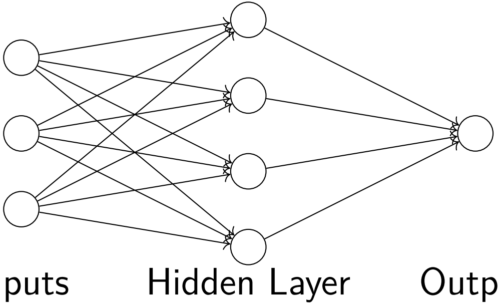
A feedforward neural network with:

- at least one hidden layer,
- a non-linear activation function

can approximate any continuous function on a bounded domain

Neural networks are general-purpose function approximators

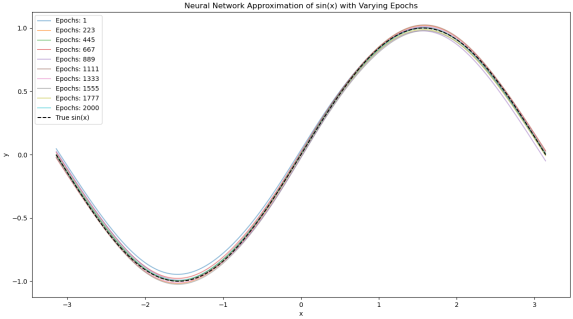
Neural Network as Function Approximator



The hidden layer allows the network to construct non-linear mappings.

Example: Approximating $sin(x)$

The network learns $sin(x)$ by observing sampled input-output pairs



The approximation closely follows the true function

Even a small neural network can learn to approximate $sin(x)$

The network does not know trigonometry; it just learns patterns in the data

Approximating $sin(x)$ with a neural network

```python
import numpy as np
import tensorflow as tf
from tensorflow import keras
import math
import matplotlib.pyplot as plt
# Generate data
xs = np.linspace(-math.pi, math.pi, 200).reshape(-1, 1)
ys = np.sin(xs)
# Model
model = keras.Sequential([
    keras.layers.Dense(16, activation='tanh', input_shape
          =(1,)),
    keras.layers.Dense(1)
])
model.compile(optimizer=keras.optimizers.Adam(0.01)
           , loss='mse')
history = model.fit(xs, ys, epochs=2000, verbose=0)
```

```python
# Predictions
ys_pred = model.predict(xs)

# Plot sin(x) vs approximation
plt.figure()
plt.plot(xs, ys)
plt.plot(xs, ys_pred)
plt.savefig("sin_approx.png")

# Plot loss
plt.figure()
plt.plot(history.history['loss'])
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.savefig("../fig/sin_loss.png")
```

Model Definition in Keras

The model is defined as:

```
model = keras.Sequential([
    keras.layers.Dense(16, activation='tanh', input_shape=(1,)),
    keras.layers.Dense(1)
])
```

- `Sequential([...])` constructs a network where each layer feeds into the next.
- This creates a 1-hidden-layer neural network for regression.
- Input shape `(1,)` means each input example is a single scalar x.
- The first layer has 16 neurons with a tanh activation, providing non-linearity.
- The final layer has 1 neuron and no activation, producing a real-valued output $\hat{y}$.

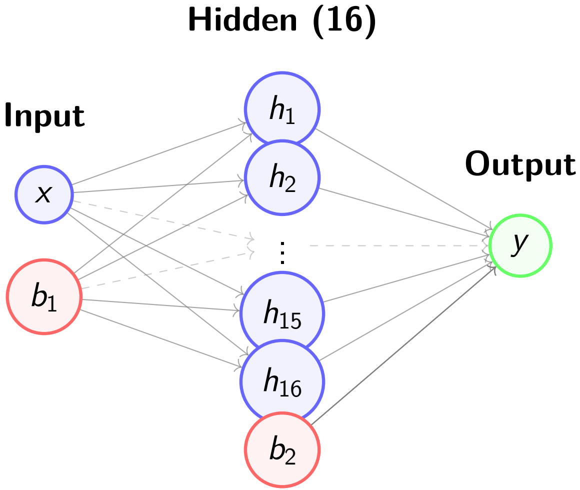
How the Model Computes the Output

Let x be the input

1. First dense layer: $z_1 = xW_1 + b_1$ where W_1 has shape $(1, 16)$ and b_1 has shape (16) $h = \tanh(z_1)$
2. Output layer: $\hat{y} = hW_2 + b_2$ with W_2 of shape $(16, 1)$ and b_2 scalar

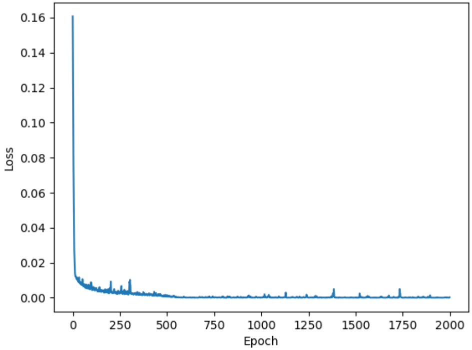
Total trainable parameters: $(1 \cdot 16 + 16) + (16 \cdot 1 + 1) = 49$
Key idea: The hidden layer $\tanh()$ allows the model to approximate non-linear functions like $\sin(x)$

Architecture of the neural network



Training Loss Over Time

During training, the loss decreases as the network improves its approximation.



Takeaways

- Only one hidden layer is needed (in theory)
- Non-linearity is essential
- The theorem guarantees existence, not that training will find the best solution

Neural networks succeed because they compose simple functions into complex ones

K Nearest Neighbours Classifier

- keeps all the training data
  - i.e. the model is the entire training set
- future predictions by computing the similarity between the new sample and each training instance
  - can use any similarity function
- picks the K entries in the database which are closest to the new data point
- majority vote
- main parameters
  - the number of neighbours to check
  - the metric used to compute the distances
    - the Mahalanobis distance has good performance

Definition of Loss

A loss function (or cost function) is a mathematical function that measures how well a machine learning model's predictions match the actual target values.

Formally, for a single example:

$$L(y, \hat{y})$$

where:

- y is the true value (ground truth)
- $\hat{y}$ is the predicted value from the model
- L quantifies the discrepancy between prediction and reality

Key Concepts

Purpose:

- The loss function provides a single numerical score indicating how wrong the model is
- Lower loss means better predictions
- The model learns by minimizing this loss during training

Total Loss:

For a dataset with n examples, the total loss is typically the average:

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} L(y_i, \hat{y}_i)$$

Common Loss Functions

Regression (continuous outputs):

- Mean Squared Error (MSE): $L(y, \hat{y}) = (y - \hat{y})^2$
- Mean Absolute Error (MAE): $L(y, \hat{y}) = |y - \hat{y}|$

Classification (discrete outputs):

- Cross-Entropy Loss: $L(y, \hat{y}) = -\sum_c y_c \log(\hat{y}_c)$
- Hinge Loss: Used in SVMs

The choice of loss function depends on the type of problem and the desired properties of the model.

Training Process

The learning algorithm aims to find model parameters θ that minimize the loss:

$$\theta^* = \arg\min_{\theta} \mathcal{L}(\theta)$$

This is typically done through:

- Gradient descent or its variants
- Computing $\nabla_\theta \mathcal{L}$ (gradient of loss with respect to parameters)
- Iteratively updating: $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$

The loss function is thus the objective that guides the entire learning process.

Loss Function vs. Metric

Loss Function

- Used during training to guide optimization.
- Must be differentiable (or have a usable gradient/subgradient) so methods like gradient descent can update parameters.
- Examples: cross-entropy loss, hinge loss.

Metric (like accuracy)

- Used after training (or during validation) to measure model performance in an interpretable way.
- Does not need to be differentiable.
- Accuracy is simply the proportion of correctly classified samples:

Why Accuracy is Not Used as a Loss

- It is non-differentiable (a step function).
- Small parameter changes often do not change accuracy at all.
- Gradient-based optimizers cannot use it.
- Instead, we use smooth functions like cross-entropy loss, which correlates well with accuracy but is differentiable.

Loss vs Accuracy

- Cross-entropy loss $\rightarrow$ drives the training.
- Accuracy $\rightarrow$ checks how well the model is doing.

Cross-entropy – Intuitive Idea

It measures how "surprised" your model is by the true labels.

- If the model assigns high probability to the correct class → low surprise → low cross-entropy.
- If it assigns low probability to the correct class → high surprise → high cross-entropy.

Cross-entropy penalizes wrong or overconfident predictions — it compares two probability distributions:

- The true distribution p (which puts probability 1 on the correct class).
- The predicted distribution q (the model's probabilities).

Formula

For one sample with true class y and predicted probabilities q:

$$H(p, q) = -\sum_i p_i \log q_i$$

Since $p_i = 1$ only for the true class, this simplifies to:

$$H(p, q) = -\log(q_y)$$

The loss is small if q_y (the predicted probability of the correct class) is close to 1, and large if it is close to 0.

Example: Cat vs. Dog Classifier

True label: "cat"
Model outputs:

$$P(\text{cat}) = 0.9, \quad P(\text{dog}) = 0.1$$

Then:

$$\text{Cross-entropy} = -\log(0.9) = 0.105$$

If instead:

$$P(\text{cat}) = 0.1, \quad P(\text{dog}) = 0.9$$

Then:

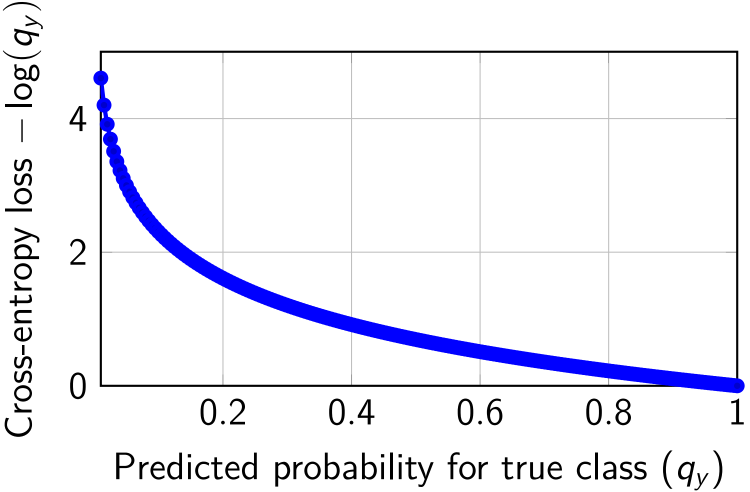
$$\text{Cross-entropy} = -\log(0.1) = 2.302$$

Much higher! The model is "very surprised" and gets a large penalty.

Intuitive Summary

- Cross-entropy = "How bad is my probability guess?"
- It encourages models to be both correct and confident (well-calibrated).
- It is the standard loss for classification because it aligns with maximum likelihood estimation.

Visual: Cross-Entropy vs. Predicted Probability



From a binary classifier to multi–class classification[5]

- Several classifiers (e.g. SVM and linear perceptron), generate a binary classification model
- two ways to deal with multi–class classification
  - transform the training algorithm and the model
    - sometimes at the expenses of an increased size of the problem
  - use a set of binary classifiers and combine the results
    - at the expenses of an increased number of problems to solve
    - one-vs-one and one-vs-rest strategies
- most implementations of "intrinsically binary classifiers" include one of these transformations when dealing with multi-class problems

5 [James et al.(2015)James, Witten, Hastie, and Tibshirani] chapter 9

One-vs-one strategy (OVO)

- consider all the possible pairs of classes and generate a binary classifier for each pair
  - $C * (C - 1)/2$ pairs
- each binary problem considers only the examples of the two selected classes
- at prediction time apply a voting scheme
  - an unseen example is submitted to the $C * (C - 1)/2$ binary classifiers, each winning class receives a $+1$
  - the class with the highest number of $+1$ wins

One-vs-rest strategy (OVR)

- consider C binary problems where class c is a positive example, and all the others are negatives
- build C binary classifiers
- at prediction time apply a voting scheme
  - an unseen example is submitted to the C binary classifiers, obtaining a confidence score
  - the confidences are combined and the class with the highest global score is chosen

OVO vs OVR

- OVO requires solving a higher number of problems, even if they are of smaller size
- OVR tends to be intrinsically unbalanced
  - if the classes are evenly distributed in the examples, each classifier has a proportion positive to negative 1 to $C - 1$

Ensemble methods

aka Classifier combination

- train a set of base classifiers
- the final prediction is obtained taking the votes of the base classifiers
- ensemble methods tend to perform better than a single classifier

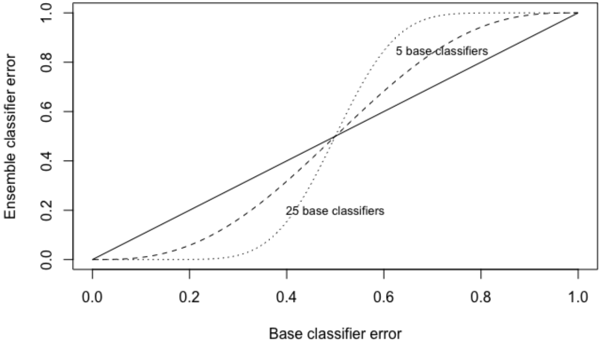
Why should this be better?

A hypothetical, extreme case

- let us consider 25 binary classifiers, each of which has error rate $\epsilon = 0.35$
- the ensemble classifier output is the majority of the predictions (13 or more)
- if the base classifiers are equal, the ensemble error rate is still ϵ
- if the base classifiers have all error rate ϵ, but they are independent, i.e. their errors are uncorrelated, then the ensemble will be wrong only when the majority of the base classifier is wrong

$$\epsilon_{ensemble} = \sum_{i=13}^{25} \binom{25}{i} \epsilon^i (1-\epsilon)^{25-i} = 0.06$$

Ensemble error – a hypothetical extreme case



Comparison between error of the base classifiers and error of the ensemble classifier

Rationale for ensemble method

Ensemble methods are useful if:

1. the base classifiers are independent
2. the performance of the base classifier is better than random choice

Methods for ensemble classifiers I

By manipulating the training set -
Data are resampled according to some sampling strategy

- **Bagging** repeatedly samples with replacement according to a uniform probability distribution
- **Boosting** iteratively changes the distribution of training examples so that the base classifier focus on examples which are hard to classify
- **Adaboost** the importance of each base classifier depends on its error rate

Methods for ensemble classifiers II

By manipulating the input features -

Subset of input features can be chosen either random or according to suggestions from domain experts

Random forest uses decision trees as base classifiers

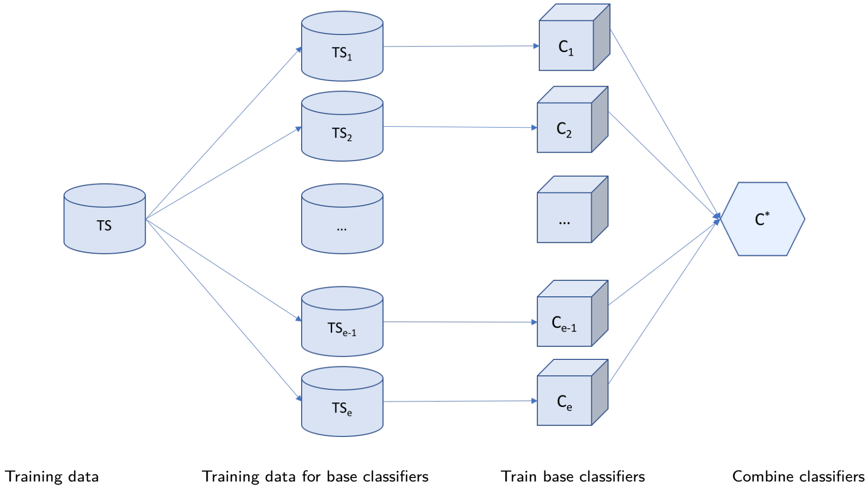
- frequently produces very good results

Methods for ensemble classifiers III

By manipulating the class labels -

- Useful when the number of classes is high
- For each base classifier, randomly partition class labels into two subsets, say A_1, A_2 and re-label the dataset
- Train binary classifiers with respect to the two classes
- At testing time, when a subset is selected all the classes it includes receive a vote
- The class with the top score will win

General schema for ensemble methods



Forest of randomised trees

- perturb-and-combine techniques specifically designed for trees
- a diverse set of classifiers is created by introducing randomness in the classifier construction
  - the classifiers must be independent
  - each tree in the ensemble is built from a sample drawn with replacement (i.e., a bootstrap sample) from the training set
  - furthermore, when splitting each node during the construction of a tree, the best split is found either from all input features or a random subset of size `max_features`
- the prediction of the ensemble is given as the combination of the predictions of the individual classifiers (e.g. voting)

Bias–vs–Variance tradeoff[6]

- Bias is the simplifying assumptions made by the model to make the target function easier to approximate
- Variance is the amount that the estimate of the target function will change, given different training data
- Bias-variance trade-off is the sweet spot where our machine model performs between the errors introduced by the bias and the variance.

If Bias vs Variance was the act of reading, it could be like
Skimming a Text vs Memorizing a Text

6 https://www.kdnuggets.com/2020/09/understanding-bias-variance-trade-off-3-minutes.html

Random Forest

the purpose of the two sources of randomness is to decrease the variance of the forest estimator

- individual decision trees typically exhibit high variance and tend to overfit
- the injected randomness in forests yield decision trees with somewhat decoupled prediction errors
- by taking an average of those predictions, some errors can cancel out
- random forests achieve a reduced variance by combining diverse trees, sometimes at the cost of a slight increase in bias.
- In practice the variance reduction is often significant hence yielding an overall better model.

Ensemble learning with boosting

- a weight is associated to each training instance
- different classifiers are trained with those weights
- the weights are modified iteratively according to classifier performance

AdaBoost I

- fit a sequence of weak learners on repeatedly modified versions of the data
- the predictions from all of them are then combined through a weighted majority vote (or sum) to produce the final prediction
- the data modifications at each so-called boosting iteration consist of applying and modifying weights to each of the training samples

AdaBoost II

- initially, the weights $w_1, w_2, \ldots, w_N$ are all set to $1/N$, so that the first step simply trains a weak learner on the original data
- for each successive iteration, the sample weights are individually modified and the learning algorithm is reapplied to the reweighted data
  - at a given iteration, the training examples that were incorrectly predicted by the boosted model induced at the previous step have their weights increased, whereas the weights are decreased for those that were predicted correctly

AdaBoost III

- as iterations proceed, examples that are difficult to predict receive ever-increasing influence; each subsequent weak learner is thereby forced to concentrate on the examples that are missed by the previous ones in the sequence

Other algorithms

- **XGBoost** (Extreme Gradient Boosting) stands for eXtreme Gradient Boosting, is a popular and powerful machine learning algorithm used for both classification and regression tasks. It is an implementation of gradient boosted decision trees designed for speed and performance
- **LightGBM** gradient boosting framework developed by Microsoft that is designed for distributed and efficient training. It is known for its high performance and supports large datasets.
- **CatBoost** is a gradient boosting library developed by Yandex that is designed to handle categorical features efficiently. It is known for its ease of use and ability to perform well with default hyperparameters.
- **Gradient Boosting Machines** general term that refers to the class of algorithms that build a model in a stage-wise fashion by optimising a differentiable loss function.
- **H2O.ai** is an open-source software for data analysis that includes an implementation of gradient boosting. It is designed for distributed computing and is known for its speed and scalability.
- **TensorFlow Decision Forests** is a library developed by Google that provides an implementation of decision forests, which includes both Random Forests and Gradient Boosted Decision Trees (GBDT).
- **Sklearn Gradient Boosting** or Gradient Boosted Decision Trees (GBDT) is a generalization of boosting to arbitrary differentiable loss functions, see the seminal work of [Friedman2001]. GBDT is an excellent model for both regression and classification, in particular for tabular data.

What is XGBoost?

eXtreme Gradient Boosting.
It's an ensemble learning method that combines many weak learners (typically decision trees) to create a strong predictor.
Think of it as a committee of experts where:

- Each expert (tree) is relatively simple
- Each new expert focuses on correcting the mistakes of previous experts
- Their combined opinion is surprisingly accurate

The Core Idea: Learning from Mistakes

XGBoost builds trees sequentially, not independently:

1. Start with a simple prediction (e.g., mean of training labels)
2. Build a tree to predict the errors (residuals) of the current model
3. Add this tree to the model with a small weight
4. Repeat: each new tree focuses on what previous trees got wrong

The final prediction is the sum of all tree predictions:

$$\hat{y} = \sum_{t=1}^{T} f_t(x)$$

where f_t is the t-th tree and T is the total number of trees.

Why "Gradient" Boosting?

The algorithm uses gradient descent in function space:

- Define a loss function $L(y, \hat{y})$ measuring prediction error
- Each new tree approximates the negative gradient of the loss
- This is like taking a small step toward minimizing the error
- Mathematically: fit tree to $-\frac{\partial L}{\partial \hat{y}}$

For classification with log-loss:

- Gradients represent how much each prediction is "off"
- Trees learn to push probabilities in the right direction

Classification with XGBoost

For binary classification:

1. Trees output real-valued scores, not probabilities
2. Final score: $F(x) = \sum_{t=1}^{T} f_t(x)$
3. Convert to probability using sigmoid function:

$$P(y = 1|x) = \frac{1}{1 + e^{-F(x)}}$$

4. Predict class 1 if $P(y = 1|x) > 0.5$

For multi-class classification:

- Build one set of trees per class
- Use softmax to convert scores to probabilities
- Predict the class with highest probability

What Makes XGBoost "eXtreme"? I

Beyond standard gradient boosting, XGBoost adds:

Regularization:

- Penalizes complex trees to prevent overfitting
- Controls tree depth, number of leaves, and leaf weights

Smart tree building:

- Uses second-order gradients (Newton's method) for better optimization
- Approximate split finding for faster training on large datasets

What Makes XGBoost "eXtreme"? II

Systems optimization:

- Parallel processing for building trees
- Cache-aware computation
- Handles missing values automatically

Key Hyperparameters I

Understanding these helps you control the learning process:

Number of trees (T):

- More trees = more complex model
- Too many $\rightarrow$ overfitting; too few $\rightarrow$ underfitting

Learning rate (η):

- Shrinks each tree's contribution
- Smaller values require more trees but often generalize better

Key Hyperparameters II

Tree depth:

- Controls complexity of individual trees
- Deeper trees capture more interactions

Regularization parameters:

- Control penalty on tree complexity

Intuitive Analogy

Think of XGBoost like collaborative problem-solving:

- **First attempt:** You make a rough guess (baseline prediction)
- **Expert 1:** Identifies patterns in your mistakes and suggests corrections
- **Expert 2:** Looks at remaining mistakes and suggests further refinements
- **Expert 3, 4, ...:** Each focuses on residual errors
- **Final decision:** Combine all suggestions with appropriate weights

The learning rate is like asking each expert to be modest in their suggestions, preventing any single voice from dominating.

Why XGBoost Wins Competitions

XGBoost frequently dominates on structured/tabular data:

Strengths:

- Highly accurate on medium-sized datasets
- Handles non-linear relationships and interactions well
- Robust to outliers and irrelevant features
- Fast training compared to other ensemble methods
- Works well out-of-the-box with minimal tuning

When to use it:

- Tabular data with mixed feature types
- Need for high accuracy with reasonable training time
- Situations where deep learning would be overkill

Practical Tips I

Training strategy:

1. Start with default parameters
2. Use cross-validation to tune hyperparameters
3. Common tuning sequence: number of trees $\rightarrow$ learning rate $\rightarrow$ tree depth $\rightarrow$ regularization
4. Monitor validation loss to detect overfitting

Practical Tips II

Watch out for:

- Overfitting with too many deep trees
- Class imbalance: use `scale_pos_weight` parameter
- Memory usage on very large datasets

Interpretation:

- Use feature importance to understand what matters
- SHAP values for detailed explanations

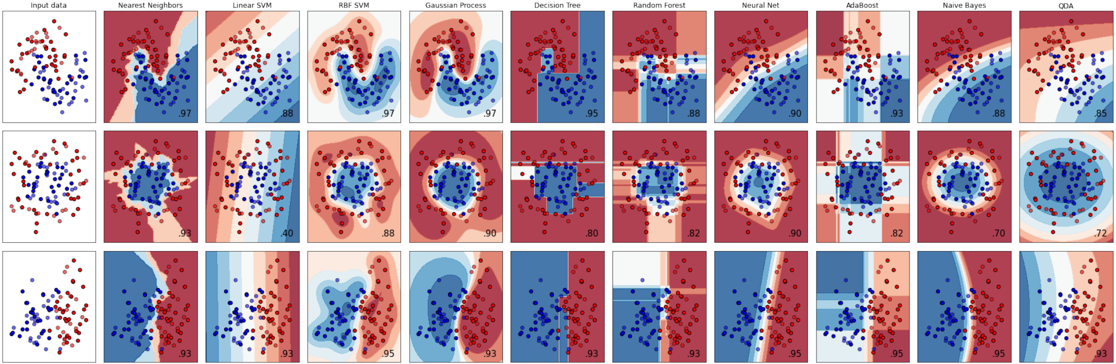
Summary

XGBoost is a powerful ensemble method that:

- Builds trees sequentially, each correcting previous errors
- Uses gradient descent to optimize a loss function
- Adds regularization and system optimizations for performance
- Excels on structured data for classification and regression

It's often the first choice for tabular data problems where interpretability isn't the primary concern and you have enough data to train effectively.

Classifiers comparison



Bibliography

- Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani.
  An Introduction to Statistical Learning.
  Springer, 2015.
- Ian H. Witten, Eibe Frank, and Mark Hall.
  Data Mining – Practical Machine Learning Tools and Techniques.
  Morgan Kaufman, 2011.
  ISBN 978-0-12-374856-0.