Machine Learning

Federico Matteoni

A.A. 2021/22

Index

0.1 Introduction

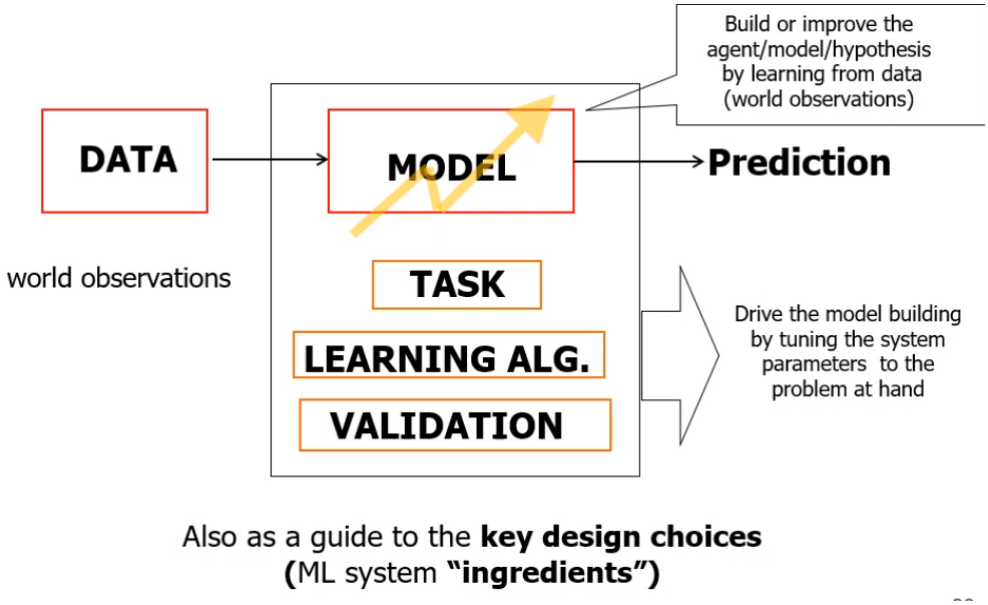
What is ML? Area of research combining aims of creating computers that could learn and powerful and adaptive statistical tools with rigorous foundation in computational science. Luxury or necessity? Growing availability and need for analysis of empirical data and difficult to provide intelligence and adaptivity by programming it. Change of paradigm.
Examples: spam classification, written text recognition... No or poor prior knowledge and rules for solving the problem, but easier to have a source of training experience.
ML is considered the latest general-purpose technology, capable of drastically affect pre-existing economic and social structures. And already has. The ultimate aim is to bring benefits to the people by solving big and small problems, accelerating human progress and empowering humans to add intelligence in any other science field.

Machine Learning We restrict to the computational framework: principles, methods and algorithms for learning and prediction, from experience. Building a model to be used for predictions. Common framework: infer a model or a **function** from a set of examples which allows the generalization (accurate response to new data).
When can we use ML? Be aware of the opportunity and awareness. ML is useful when there's no or poor theory surrounding the phenomenon, or uncertain, noisy or incomplete data which hinders formalization of solutions. The requests are: source of training experience (representative data) and a tolerance on the precision of results. The best examples are models to solve real-world problems that are difficult to be treated with traditional techniques: face and voice recognition (knowledge too difficult to formalize in an algorithm), predicting biding strength of molecules to proteins (not enough human knowledge) and personalized behavior, such as recommendation systems, scoring messages according to user preferences...

Definition The ML studies and proposes methods to build functions/hypothesis from examples of observed data that fits the known examples and able to generalize, with reasonable accuracy, for new data (according to verifiable results and under statistical and computational conditions and criteria.

Data Data **represents the available experience**. Representation problem: capturing the structure of the analyzed objects. Flat (attribute-value), structured..., categorical or continuous, missing data... **preprocessing**: variable scaling, encoding, selection...

Task The task defines the purpose of the application: knowledge that we want to achieve? which is the helpful nature of the result? what information are available?
Predictive task, classification and regression: function approximation
Descriptive task, cluster analysis and association rules: find subsets or groups of unclassified data.



Also as a guide to the **key design choices**
(ML system **"ingredients"**)

Supervised learning Given a set of training examples as ⟨input, output⟩ = $\langle x, d \rangle$ (**labeled examples**) for an unknown function f, find a *good approximation* of f, an hypothesis h that can be used for making predictions on unseen data x'.
The target d can be:

> Discrete value, for **classification tasks**.
> $f(x) \in \{1, 2, \ldots, k\}$
> Patterns, feature vectors, are seen as members of a class and the goal is to assign the new patterns observed to the correct class (or label)
> If the number of possible classes is two, then f is a *boolean function* and the task is called **binary classification** or **concept learning**: true or false, positive or negative, 0 or 1...
> If the number of classes is greater that two then is a **multi-class classification task**.
>
> Real continuous value, for **regression tasks**.
> The patterns are seen as sets of variables (real values), and the task is a curve fitting task. The process aims to estimate a real-value function based of a finite set of noisy samples $\langle x, f(x) +$ random noise$\rangle$

Unsupervised learning No teacher. The training set is a set of unlabeled data $\langle x \rangle$. Examples: clustering, finding natural groupings in a set of data.

Learning algorithm Basing on data, task and model: heuristic search through the hypothesis space H of the **best hypothesis**. I. e. the best approximation of the unknown target function, typically searching for the h with the minimum *error*. H may not coincide with the set of all possible function and the search cannot be exhaustive, we need to make **assumpitons** (**inductive bias**).

Learning Also called:

> Inference, in statistics
>
> Adapting, in biology and systems
>
> Optimizing, in mathematics
>
> Training, in neural networks
>
> Function approximations, in mathematics
>
> ...

After introducing data, task, model and learning algorithm we will focus on: inductive bias, loss and concepts of generalization and validation.

Inductive bias To set up a model we can make assumptions about the nature of the target function, concerning either:

> constraints in the model, **language bias** (in the hypothesis space H, due to the set of hypothesis that we can express or consider
>
> constraints or preferences in learning algorithm/search strategy, **search bias** which is preferred
>
> or both

Such assumptions are needed to obtain an useful model for the ML aims, i.e. a model with generalization capabilities. We can imagine learning a discrete function with discrete inputs assuming **conjunctive rules**, so using a **language bias** to work with a restricted hypothesis space.

Version Space An hypothesis h is consistent with the TR if $h(x) = d(x)$ for each training example $\langle x, d(x) \rangle$.
The **version space** $VS_{H,TR}$ is the subset of H of the hypothesis consistent with all the training examples $\langle x, d(x) \rangle$ in the TR.
It's possible to do an exhaustive search in an efficient way, using clever algorithms. This means finding the set of all the hypothesis h consistent with the TR set.

Unbiased Learner The language bias (ex: using only conjunctive rules, may be too restrictive: if the target concept is not in H it cannot be represented in H. We can use an H that expresses every teachable concept (among propositions), that means that H is the set of all possible subsets of X: the power set $P(X)$. If $n = 10$ binary inputs, then $|X| = 2^{10} = 1024$ and $|P(X)| = 2^{1024} = 10^{308}$ possible concepts, which is much more than the number of the atoms in the universe.
An unbiased learner is unable to generalize: the only examples that are unambiguously classified by an unbiased learner represented with the VS are the training examples themselves. Each unobserved instance will be classified positively by exactly half of the hypothesis in the VS and negative by the other half. Indeed: $\forall\, h$ consistent with x_i, $\exists\, h'$ identical to h except $h'(x_i) \neq h(x_i)$, $h \in \text{VS} \Rightarrow h' \in \text{VS}$ (because they are identical on the TR)

Why prefer the search bias? In ML we use flexible approaches (expressive hypothesis spaces with universal capability of the models, for example neural networks or decision trees. We avoid the language bias, so we do not exclude a priori the unknown target function, but we focus on the search bias (ruled by the learning algorithm).

Loss How to measure the quality of an approximation? We want to measure the distance between $h(x)$ and d, using a loss function/measure $L(h(x), d)$ for a pattern x which has high value in cases of bad approximation. The error (or risk or loss) is an expected value of this L, for example $E(w) = \frac{1}{l}\sum_{p=1}^{l} L(h(x_p), d_p)$. Different L for different tasks. Examples of loss functions:

Regression: $L(h(x_p), d_p) = (d_p - h(x_p))^2$, the squared error. MSE (mean squared error) over the data set

Classification: $L(h(x_p), d_p) = \begin{cases} 0 & h(x_p) = d_p \\ 1 & else \end{cases}$

Learning and generalization Learning: search for a **good function** in a function space from known data (typically minimizing an error/loss). **Good** with respect to generalization error: it measures how accurately the model predicts over novel samples of data (**measured over <u>new</u> data**).
Generalization is the crucial point of ML. Performance in ML is the generalization accuracy or *predictive accuracy* estimated by the error on the test set.

ML issues Inferring general functions from known data is an ill posed problem, which means that in general the solution is not unique because we can't expect the exact solution with finite data. What can we represent? And so, what can we learn?
Learning phase: building the model including training. The prediction phase is evaluating the learned function over new never-seen-before samples (generalization capability). Inductive learning hypothesis: any h that approximates f well on training examples will also approximate f well on new unseen instances x.
Overfitting: a learner overfits data if it outputs an hypothesis $h \in H$ having true/generalization error (risk) R and empirical (training) error E, but there's another $h' \in H$ with $E' > E$ and $R' < R$, which means that h' is the better one despite having a worse fitting.

Statistical Learning Theory Under what mathematical conditions is a model able to generalize? We want to investigate the generalization capability of a model, measured as a risk or test error, the role of the model complexity and the role of the number of data.
Formal Setting: approximate a function $f(x)$, with d target ($d = f(x)+$ noise), minimizing the **risk function**

$$R = \int L(d, h(x))\, dP(x, d)$$

which is the **true error over all the data**, given:

a value d from the teacher and the probability distribution $P(x, d)$

a loss function $L(h(x), d) = (d - h(x))^2$

We search for $h \in H \mid min\, R$, but we only have the finite data set $TR = (x_p, d_p)$ with $p = 1 \ldots l$. Looking for h means minimizing the empirical risk (the training error E), finding the best values for the model free parameters

$$R_{emp} = \frac{1}{l}\sum_{p=1}^{l}(d_p - h(x_p))^2$$

The inductive principle is the **ERM**, Empirical Risk Minimization: can we use R_{emp} to approximate R?

Vapnik-Chervoneniks dim and SLT Given the VC dimension (simply VC), a measure of complexity of H and by that we mean its flexibility to fit data.
The VC-bound states that it holds with probability $\frac{1}{\delta}$ that

$$R \leq R_{emp} + \epsilon\left(\frac{1}{l}, VC, \frac{1}{\delta}\right)$$

ϵ is a function called VC-confidence, that grows with VC and decreases with higher l and δ

R_{emp} decreases using complex models (with high VC)

δ is the confidence, and it rules the probability that the bound holds.
$\delta = 0.01 \Rightarrow$ the bound holds with probability 0.99

Intuitively:

Higher l (data) $\Rightarrow$ lower VC confidence and bound closer to R

A too simple model, meaning with low VC, can be not sufficient due to high R_{emp} (**underfitting**)

An higher VC with fix $l \Rightarrow$ lower R_{emp} but VC and hence R may increase (**overfitting**)

Structural risk minimization Minimize the bound! There are different bounds formulations according to different classes of f, of tasks...
In other words, we can make a good approximation of f from examples, provided that we have a good number of data and the complexity of the model is suitable for the task.

Complexity control The Statistical Learning Theory allows for a formal framing of the problem of generalization and overfitting, providing an analytic upper bound to the risk R for the prediction over all the data, regardless of the type of learning algorithm or the details of the model. So **the machine learning is well founded**, the learning risk can be analytically limited and only a few concepts are fundamental. This leads to new models (such as the Support Vector Machine) and other methods that directly consider the control of the complexity in the construction of the model.

Validation Central role for the applications and the project. Two aims:

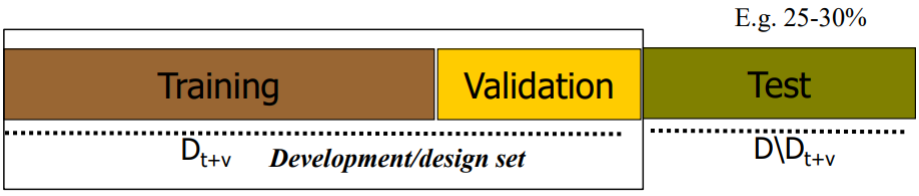
Model Selection: estimating the performance (**generalization error**) of different models in order to choose the best one. This includes searching for the best hyperparameters of the model.
It returns a model.

Model Assessment: with the final model, estimating/evaluating its prediction error/risk (**generalization error**) over new test data.
It returns an estimation.

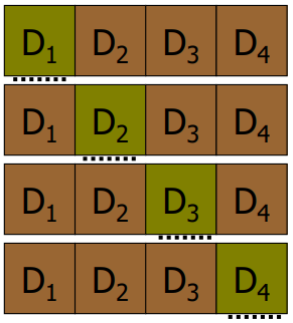
Golden rule: keep the two goals separated and use different datasets for each one.

In an ideal world, we'd have a large training set, a large validation set for model selection and a very large external unseen data test set. With finite and often small data sets we have just an estimation of the generalization performance. We have to use some techniques: hold-out and k-fold cross validation, for example.

Hold-Out: we partition the dataset D into **training set** TR, **validation/selection set** VL and **test set** TS. All three are disjoint: TR is used to run the training algorithm, VL can be used to select the best model (hyperparameters tuning) and the **TS is only used for model assessment**.



K-Fold: this technique can make use of insufficient data. We split the dataset D into k mutually esclusive subsets $D_1, \ldots, D_k$, we train on $D - D_i$ and test it on D_i.
This can be applied to both VL and TS splitting. Can be computationally very expensive and there's the issue of choosing the number of folds k.



Confusion Matrix

Actual/Predicted	**Positive**	**Negative**
Positive	TP	FN
Negative	FP	TN

Specificity $= \frac{TN}{FP+TN}$, and **true negative rate** $= 1 - FPR$
Sensitivity $= \frac{TP}{TP+FN}$, also known as **true positive rate** or **recall**
Precision $= \frac{TP}{TP+FP}$
Accuracy: % of correctly classified patterns $= \frac{TP + TN}{total}$. Note that, for example, a 50% accuracy on a binary classifier is equivalent to a random predictor.

ROC Curve We plot **specificity** on **x-axis** and **sensitivity** on the **y-axis**. The diagonal corresponds to the worst classifier, the random guesser. Better curves have greater Area Under the Curve (AUC)

Linear Models Mainstay of statistics.

Univariate Linear Regression Simple linear regression: we start with 1 input variable x and 1 output variable y. We assume a model $h_w(x)$ expressed as $out = w_1 x + w_0$ where w are real-valued coefficients or **free parameters**, also called **weights**.
Given that the ws are continuous valued, we have an infinite hypothesis space but a nice solution from classical math. We can learn with this basic tool and, although simple, it includes many relevant concepts of modern ML and many methods in the field are based on this.
Least Mean Square: learning means finding w such that it minimizes the error/empirical loss, with best data fitting on the training set with l examples.
So given a set of l training examples (x_p, y_p) with $p = 1, \ldots, l$, we have to find $h_w(x)$ in the form $w_1 x + w_0$ that minimizes the expected loss on the training data. For the loss, we use the square of errors: **least mean square**, find w to **minimize** the residual sum of squares.

$$Loss(h_w) = E(w) = \sum_{p=1}^{l} (y_p - h_w(x_p))^2 = \sum_{p=1}^{l} (y_p - (w_1 x_p + w_0))^2$$

To have the mean, divide by l. How to solve? Local minimum as stationary point, so the gradient $\frac{\partial E(w)}{\partial w_i} = 0$ with $i = 1, \ldots,$ dim_input $+1 = 1, \ldots, n+1$. For the simple linear regression (2 free parameters):

$$\frac{\partial E(w)}{\partial w_0} = 0 \quad \frac{\partial E(w)}{\partial w_1} = 0$$

$$\frac{\partial E(w)}{\partial w_0} = -2(y - h_w(x)) \quad \frac{\partial E(w)}{\partial w_1} = -2(y - h_w(x)) \cdot x$$

Classification The same models used for regression can be used for classification: **categorical targets** y or d, for example 0/1, -1/+1...
We use an hyperplane (wx) assuming negative or positive values. We exploit such models to decide if a point x belongs to the positive or the negative zone of the hyperplane to classify it. So we want to learn w such that we get a good classification accuracy. The decision boundary is $x^T w = w^T x = w_0 + w_1 x_1 + w_2 x_2 = 0$ and we can introduce a threshold function which can be written in many ways:

$$h(x) = \begin{cases} 1 & \text{if } wx + w_0 \geq 0 \\ 0 & \text{else} \end{cases}$$

$$h(x) = sign(wx + w_0)$$

...

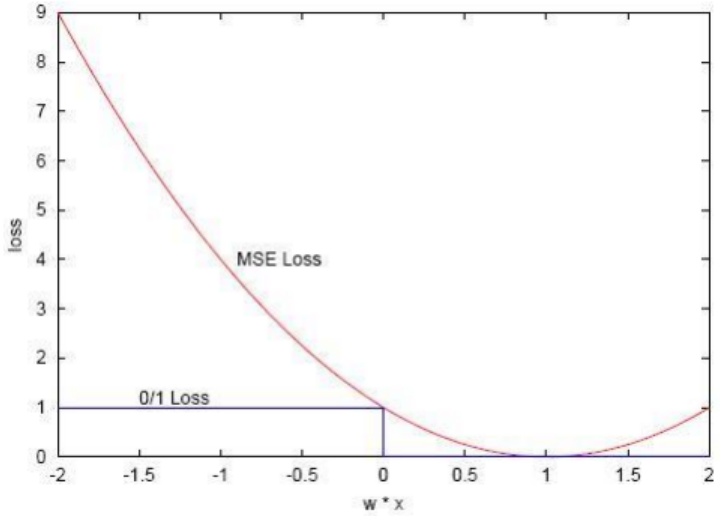
w_0 is called **threshold** or **bias**. $h(x) = w^T x + w_0 \geq 0 \Leftrightarrow w^T x \geq -w_0$

Learning Algorithms Introducing 2 learning algorithms, both based on LSM and used for the linear model on regression and classification tasks. We start redefining the learning problem and the loss for them (in the case of l data and multidimensional inputs).

Learning problem for classification tasks: given a set of l training examples (x_p, y_p) and a loss function L, with $y_p \in \{0, 1\}$ or $y_p \in \{-1, +1\}$, find the weight vector w that minimizes expected loss on the training data

$$R_{emp} = \frac{1}{l} \sum_{p=1}^{l} L(h(x_p), y_p)$$

The expected loss can be approximated by a smooth function. We can make the optimization problem easier by replacing the original objective function L (0/1 loss) with a smooth, differentiable function: for example, the MSE loss (mean squared error).



No classification error minimizing either 0/1 loss or MSE loss. Given l training examples (x_p, y_p), find w that minimizes the residual sum of squares

$$E(w) = \sum_{p=1}^{l} (y_p - x_p^T w)^2 = ||y - Xw||^2$$

We can't use $h(x)$ in $E(w)$, as for regression, because $h(x) = sign(w^T x)$ is non-differentiable. Also, this is a quadratic function so the minimum always exists (but may not be unique). X is a $l \times n$ matrix with a row for each x_p.

Direct Approach with a normal equation Differentiating $E(w)$ with respect to w to get the **normal equation**

$$(X^T X)w = X^T y$$

In the derivation we also find that

$$\frac{\partial E(w)}{\partial w_j} = -2 \sum_{p=1}^{l} (x_p)_j \cdot (y_p - x_p^T w)$$

If $X^T X$ is not singular, then the unique solution is given by

$$w = (X^T X)^{-1} X^T y = X^+ y$$

with X^+ being the Moor-Penrose pseudoinverse. Else the solutions are infinite, so we can choose the **min norm**(w) solution.
The **Singular Value Decomposition** can be used for computing the pseudoinverse of a matrix (X^+). With $X = U\Sigma V^T \Rightarrow X^+ = V\Sigma^+ U^T$ replacing every non-zero entry by its reciprocal. We can apply SVD directly to compute $w = X^+ y$, obtaining the minimal norm (on w) solution of least squares problem.

$$\frac{\partial E(w)}{\partial w_j} = \frac{\partial \sum_{p=1}^{l}(y_p - x_p^T w)^2}{\partial w_j} = \ldots = -2\sum_{p=1}^{l}(y_p - x_p^T w)(x_p)_j$$

Gradient Descent The derivation suggests an approach based on an iterative algorithm based on $\frac{\partial E(w)}{\partial w_j} = -2\sum_{p=1}^{l}(y_p - x_p^T w)(x_p)_j$. The **gradient** is the **ascent direction**. We can move toward the minimum with a gradient descent $\Delta w = -$ gradient of $E(w)$. **Local search**: we begin with a initial weight vector and modify it iteratively to minimize the error function. The gradient vector is

$$\Delta w = -\frac{\partial E(w)}{\partial w} = \begin{bmatrix} -\frac{\partial E(w)}{\partial w_1} \\ \vdots \\ -\frac{\partial E(w)}{\partial w_n} \end{bmatrix} = \begin{bmatrix} \Delta w_1 \\ \vdots \\ \Delta w_n \end{bmatrix}$$

Allowing us to work in a multi dimensional space without the need to visualize it. Hence, the iterative approach will move using a learning rule based on a "delta" of w proportional to the opposite of the local gradient. The movements will be made according to

$$w_{new} = w + \eta \cdot \Delta w$$

The simple algorithm is as follows:

1. Start with weight vector $w_{initial}$ and fix $0 < \eta < 1$
2. Compute $\Delta w = -$ gradient of $E(w) = -\frac{\partial E(w)}{\partial w}$ (or for each w_i)
3. Compute $w_{new} = w + \eta \cdot \Delta w$ (or for each w_i)
   η is the step size or **learning rate**
4. Repeat from 2 until convergence or $E(w)$ sufficiently small

Batch version The gradient is the sum over all the l patterns. Provides a more precise evaluation of the gradient over l data. We upgrade the weight after the sum

$$\frac{\partial E(w)}{\partial w_j} = -2\sum_{p=1}^{l}(y_p - x_p^T w)(x_p)_j$$

Online/Stochastic version We upgrade the weights with the error that is computed for each pattern. Hence, the second pattern output is based on weights already updated from the first and so on. In makes progress with each example it sees. Can be faster, but needs smaller η

$$\frac{\partial E_p(w)}{\partial w_j} = -2(y_p - x_p^T w)(x_p)_j = -\Delta_p w_j$$

Gradient Descent as error correction delta rule The error correction rule, also called Widrow-Hoff or delta rule, changes w_j proportionally to the error (target y − output)

$$\Delta w_j = 2\sum_{p=1}^{l}(x_p)_j(y_p - x_p^T w)$$

$$w_{new} = w + \eta \cdot \Delta w$$

We improve by learning on previous errors.

Gradient descent is a simple and effective local search approach to a LMS solution. It allows to search through an infinite hypothesis space, can be easily applied for continuous H and differentiable losses and isn't only for linear models (also neural networks and deep learning models).
Many possible improvements (Newton, quasi-Newton methods, conjugate gradients...)

Linear models

Language bias: H is a set of linear functions.

Search Bias: ordered search guided by the least squares minimization goal. For instance, we could prefer a different method to obtain a restriction on the values of parameters, achieving a different solution with other properties.

Shows that even for a simple model there are many possibilities. We still need a principled approach.

Limitations In geometry, two set of points are linearly separable in an n-dimensional space if they can be separated by a $(n-1)$-dimensional hyper-plane. In 2 dimensions, if they can be separated by a line, in 3 dimensions, by a plane...
The linear decision boundary can provide exact solutions only for linearly separable sets of points.

Extending the linear model We can use transformed inputs, such as $x, x^2, x^3 \ldots$ with a non-linear relationship between inputs and output, maintaining the learning machinery used so far.

$$h_w(x) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

Linear basis expansion (LBE)

$$h_w(x) = \sum_{k=0}^{K} w_k \phi_k(x)$$

Augments the input vector with additional variables which are transformations of x according to a function $\phi_k : R^n \to R$, so number of parameters $K > n$: linear in the parameters, so we can use the same learning algorithms as before.
Which ϕ? Towards the dictionary approaches. Pro: can model more complicated relationships than linear, so it's more expressive. Cons: with large basis of functions, we easily risk overfitting, hence we require controlling the complexity (as in flexibility of the model to fit the data). How to do that? Many approaches:

Ridge Regression (or **Tikhonov Regularization**): smoothed model.
Add constraints to the sum of value of $|w_j|$, penalizing models with high values of $|w|$ (so favoring sparse models, using less terms due to weights $w_j = 0$ or close)

$$Loss(w) = \sum_{p=1}^{l} (y_p - x_p^T w)^2 + \lambda ||w||^2$$

with λ being the regularization hyper-parameter. It implements the control of the model complexity, leading to a model with less VC-dim with a trade-off controlled through a single parameter, λ.
This uses $||\,||_2$

Lasso uses $||\,||_1$

Elastic nets uses both $||\,||_1$ and $||\,||_2$

Learning Timing

Eager: analyze data and construct an explicit hypothesis

Lazy: store tr data and wait test data point, then construct an ad hoc hypothesis.

k-NN The algorithm is simple: store the training data and given and input x find the k nearest training examples x_i, then output the mean label.

Voronoi Diagram Each cell consists of point closer to x than any other patterns. The segments are all points in plan equidistant to two patterns. It is implicitly used by K-NN.

K-NN vs linear Two extremes of the ML panorama:

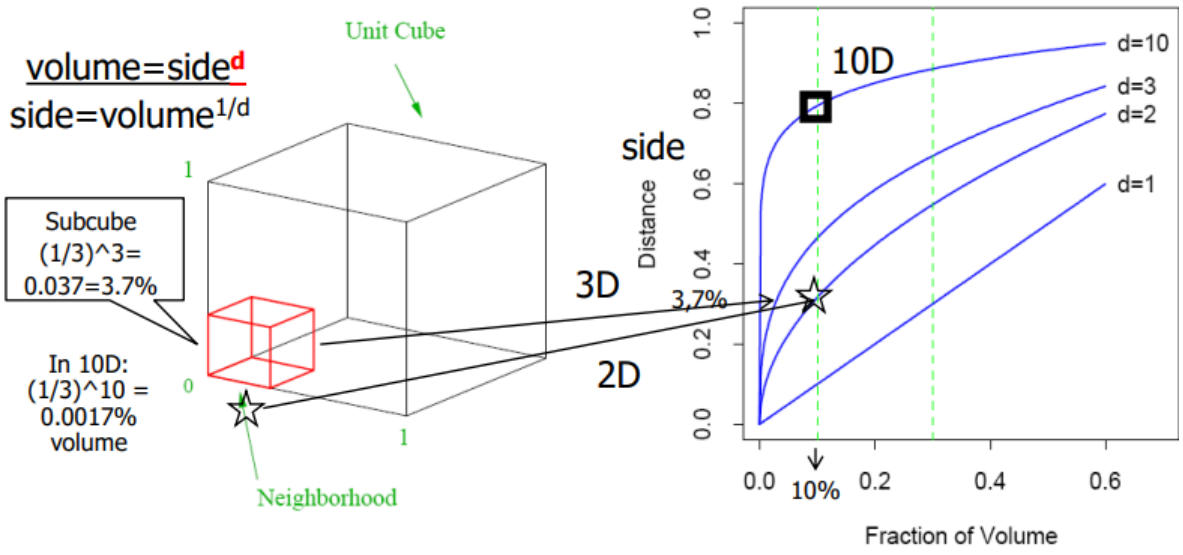
Linear	K-NN
Rigid (low variance)	flexible (high variance)
Eager	Lazy
Parametric	Instance-Based

Bayes Error Rate If we know the density $P(x, y)$, we classify the most probable class, using the conditional distribution as: output the class $v \mid$ is $max\, P(v \mid x)$.
The error rate of this classifier (called the Bayes classifier) is called the Bayes error rate: the minimum achievable error rate given the distribution of the data. K-NN directly approximates this solution (majority vote in a nearest neighborhood) except that conditional probability is relaxed to conditional probability withing a neighbordhood and probabilities are estimated by training sample proportions

Inductive bias of K-NN The assumed distance tells us which are the most similar examples. The classification is assumed similar to the classification of the neighbors according to the assumed metric.

Limitations The computational cost is deferred to the prediction phase: makes the local approximation to the target function for each new example to be predicted.
Moreover, high retrieval cost: computationally intensive, in time, for each new input because computes the distance from the test sample to all stored vectors, so high cost in space too (all training data).
It provides a good approximation if we can find a significant set of data close to any x. Can fail when we have a lot of input variables (high n, high dimensionality) due to the curse of dimensionality:

Hard to find nearby points in high dimensions
K-NN can fail in high dimensions because it becomes difficult to gather K observation close to a target point x_q: near neighborhoods tend to be spatially large and estimates are no longer local



Low sampling density for high-dim data
Sampling density is proportional to $l^{\frac{l}{d}}$ with l data and d data dim. If 100 points are needed to estimate a function in R (1 dim), then 100^{10} are needed in R^{10}

Irrelevant features: the **curse of noisy**
If the target depends only on a few features in x, we could retrieve a similar pattern with the similarity dominated by the large number of irrelevant features.
This grows with dimensionality.
We may weight features according to the relevance, or adopt feature selection approaches (eliminating some variables)

Neural Networks Models used to:

Study and model biological systems and learning processes (biological realism is essential)

Introduce effective machine learning systems and algorithms (often losing a strict biological realism, but machine learning, computational and algorithmic properties are essential)

For us: **Artificial Neural Networks** (ANN): a flexible machine learning tool in the sens of approximating functions (builds a mathematical function $h(x)$ with special properties). A neural network:

Can learn from examples

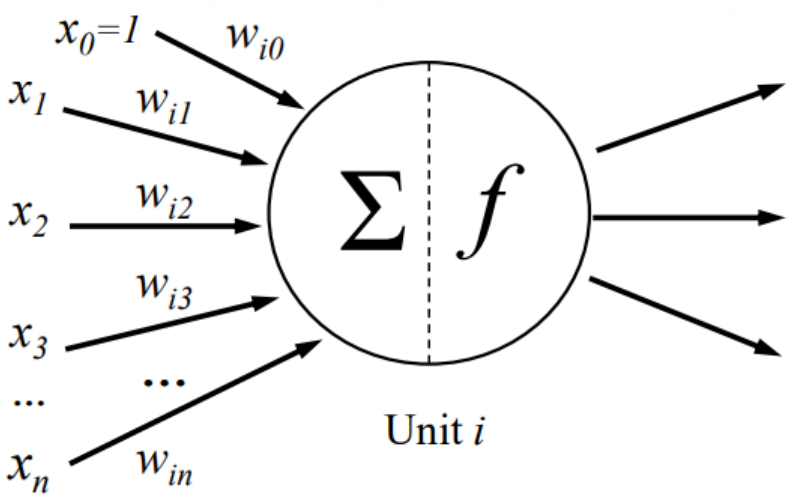
Are universal approximators (**Theorem of Cybenko**): flexible approaches for arbitrary functions (including non-linear)

Can deal with noisy and incomplete data, with a graceful degradation of performance

Can handle continuous real and discrete data for both regression and classification tasks

It's a **paradigm**: it encompasses a wide set of models.

Artificial Neuron Input from external source or other units, with weights w as free parameters: can be modified by the learning process. The unit i computes $f(\sum_j w_{ij}x_j)$ with w_{ij} the weight from input j to unit i. f is called **activation function**: linear, threshold or logistic (sigmoid). The weighted sum $(\sum_j w_{ij}x_j)$ is called net input to unit i, or net_i.



Three types of activation functions:

Linear, or identity: $f(x) = x$

Threshold, for the **perceptrons**: $f(x) = sign(x)$

Logistic: $f(x) = \frac{1}{1+e^{-\alpha x}}$

Perceptron A neuron that uses a threshold as activation function. Can be composed and connected to build a network: **MLP**, Multi Layer Perceptron

Xor $x_1 \oplus x_2 = x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2$. Let $h_1 = x_1 \cdot x_2, h_2 x_1 + x_2$ then $x_1 \oplus x_2 = \overline{h_1} \cdot h_2$ with $\wedge = \cdot$ and $\vee = +$.
So two layers are sufficient, but single layer cannot model all functions due to limits of single perceptron and the linear separable problems.

Learning for one unit model

1. **Adaline**, adaptive linear neuron: LMS direct solution and gradient descent solution
2. **Perceptron**, non linear: only classificiation
   Minimize number of misclassified patterns, find $w \mid sign(w^T x) = d$. Online algorithm, a step can be made for each input pattern.
   (a) Initialize weights
   (b) Pick learning rate η (between 0 and 1)
   (c) Until stopping condition (es weights don't change)
   For each training pattern (x, d) compute output activation $out = sign(w^T x)$.
   If $out = d$ don't change weights, if $out \neq d$ update weights $w_{new} = w + \eta \cdot d \cdot x$ adding $+\eta x$ if $wx \leq 0$ and $d = +1$ or $-\eta x$ if $wx > 0$ and $d = -1$.
   Different form $w_{new} = w + \frac{1}{2} \cdot \eta \cdot (d - out) \cdot x$

   Delta Rule: the form $w_{new} = w + \eta \cdot d \cdot x$ is the **Hebbian learning** form, while the form $w_{new} = w + \frac{1}{2} \cdot \eta \cdot (d - out) \cdot x$ is the **error correction learning** form. It's a recall from LMS: a error correction/delta/Widrow-Hoff/Adaline/LMS rule that changes the w proportionally to the error (target $d-$ output).
   In terms of neurons, the adjustment made to a synaptic weight is proportional to the product of error signal and the input signal that excite the synapse. Easy to compute when errors signal δ is directly measurable (meaning that we know the desired response for each unit).

Perceptron Convergence theorem A perceptron can represent linear decision boundaries, so it can solve linearly separable problems. Also, it can always learn the solution with the perceptron learning algorithm.
The **perceptron convergence theorem** is a milestone: a biologically inspired model with well-defined and proved computational capabilities and proved by a theorem. It states that **the perceptron is guarateed to converge** (classifying correctly all the input patterns) **in a finite number of steps if the problem is linearly separable**. This independently of the starting point, although the final solution is not unique and it depends on the starting point.

Preliminaries We focus on positive patterns, assuming $(x_i, d_i) \in$ TR set with $d_i = +1$ or -1. We also omit ^{T} for the dot product $w^T x$ (sometimes).
Linearly separable $\Rightarrow \exists\, w^*$ solution $\mid d_i(w^* x_i) \geq \alpha = \min_i d_i(w^* x_i) > 0$, henche $w^*(d_i x_i) \geq \alpha$.
With $x'_i = (d_i x_i)$ then w^* solution $\Leftrightarrow$ w^* solution of $(x'_i, +1)$
This because w^* solves $\Rightarrow d_i(w^* x_i) \geq \alpha \Rightarrow (w^* d_i x_i) \geq \alpha \Rightarrow (w^* x'_i) \geq \alpha \Rightarrow w^*$ solution of $(x'_i, +1)$.
And if w^* is a solution of $(x'_i, +1) \Rightarrow (w^* d_i x_i) \geq \alpha \Rightarrow d_i(w^* x_i) \geq \alpha \Rightarrow w^*$ solves for x_i

Also assuming $w(0) = 0$ (at step 0), $\eta = 1$ and $\beta = \max_i |x_i|^2$ where $|\,|$ is the euclidean norm.
After q errors (all false negatives), $w(q) = \sum_{j=1}^{q} x_{ij}$ with $_{ij}$ denoting the patterns belonging to the subset of misclassified patterns.

Proof The basic idea is that we can find lower and upper bound to $|w|$ as a function of q^2 steps (lower bound) and q steps (upper bound) $\Rightarrow$ we can find the number of steps $q \mid$ the algorithm converges.
Lower bound on $|w(q)|$ is $w^* w(q) = w^* \sum_{j=1}^{q} x_{ij} \geq q\alpha$ recalling that $\alpha = \min_i (w^* x_i)$. With Cauchy-Schwartz we know that $(wv)^2 \leq |w|^2 |v|^2$ where $|w|^2 = ||w||_2^2$.
$|w^*|^2 |w(q)|^2 \geq (w^* w(q))^2 \geq (q\alpha)^2 \Rightarrow |w(q)|^2 \geq \frac{(q\alpha)^2}{|w^*|^2}$. Also $|w(q)|^2 = |w(q-1) + x_{iq}|^2 = |w(q-1)|^2 + 2w(q-1)x_{iq} + |x_{iq}|^2$ because $|a + b|^2 = |a|^2 + 2ab + |b|^2$
For the q-th error, $2w(q-1)x_{iq} < 0$, so $|w(q)|^2 \leq |w(q-1)|^2 + |x_{iq}|^2$ and by iteration $w(0) = 0$ we have $|w(q)|^2 \leq \sum_{j=1}^{q} |x_{ij}|^2 \leq q\beta$ with $\beta = \max_i |x_i|^2$.

So we have an upper bound $q\beta$ and a lower bound $\frac{(q\alpha)^2}{|w^*|^2}$, so

$$q\beta \geq |w(q)|^2 \geq \frac{(q\alpha)^2}{|w^*|^2}$$

$$q\beta \geq q^2 \alpha'$$

$$\beta \geq q\alpha'$$

$$q \leq \frac{\beta}{\alpha'}$$

Differences

$$w_{new} = w + \eta(d - \text{out})x$$

Apparently similar but:

LMS algorithm

LSM rule derived without threshold activation functions

Hence for training $\delta = d - w^T x$

Can still be used for classification using $h(x) = sign(w^T x)$, LTU

Minimizes $E(w)$ with out $= w^T x$

Asymptotic convergence also for not linear separable problems

Not always zero classification errors

Can be extended to network of units (NN) using the gradient based approach

Perceptron learning algorithm

Perceptron uses out $= sign(w^T x)$

versus $\delta = d - sign(w^T x)$

Minimizes misclassifications (out $= sign(w^T x)$)

Always converges in a finite number of steps for a linear separable problem to a perfect classifier
Else it doesn't converge

Difficult to be extended to network of units (NN)

Activation functions

Linear, or identity: $f(x) = x$

Threshold, for the **perceptrons**: $f(x) = sign(x)$

Logistic: $f(x) = \frac{1}{1+e^{-\alpha x}}$
This is a non-linear squashing function like the sigmoidal logistic function: it assumes a continuous range of values in the bounded interval $[0, 1]$. It has the property of being a smoothed differentiable threshold function, with α being the slope parameter of the sigmoid function.

Radial basis: $f(x) = e^{-\alpha x^2}$

Softmax

Stochastic neurons, where the output is +1 with probability $P(net)$ or -1 with $1 - P(net) \Rightarrow$ Boltzmann machines and other models rooted in statistical mechanics.

For the derivatives, a step function has no derivative, which is exactly why it isn't used. The sigmoids have $\frac{df_\sigma(x)}{dx} = f_\sigma(x)(1 - f_\sigma(x))$ and $\frac{df_{tanh}(x)}{dx} = 1 - f_{tanh}(x)^2$ for $\alpha = 1$.
The sigmoid logistic function has the property of being a smoothed *differentiable* threshold function. Hence, we can derive a Least (Mean) Square algorithm, by computing the gradient of the mean square loss function as for the linear units (also for a classifier).
From $\sigma(x) = x^T w$ to $\sigma(x) = f_\sigma(x^T w)$ where f_σ is a logistic function. Find w that minimizes the residual sum of squares

$$E(w) = \sum_p (d_p - \sigma(x_p))^2 = \sum_p (d_p - f_\sigma(x_p^T w))^2$$

$$\frac{\partial E(w)}{\partial w_j} = -2 \sum_p^l (x_p)_j (d_p - f_\sigma(x_p^T w)) f'_\sigma(x_p^T w) =$$

$$= -2 \sum_p^l (x_p)_j \delta_p f'_\sigma(\text{net}(x_p)) \text{ for 1 unit and patterns } p = 1 \dots l$$

Gradient descent algorithm The same as for linear units using the new delta rule $w_{new} = w + \eta \cdot \delta_p \cdot x_p$ where $\delta_p = (d_p - f_\sigma(net(x_p)))f'_\sigma(net(x_p)) = (d_p - out(x_p))f'_\sigma$

Neural Network In an MLP architecture: units connected by weighted links, organized in layers:

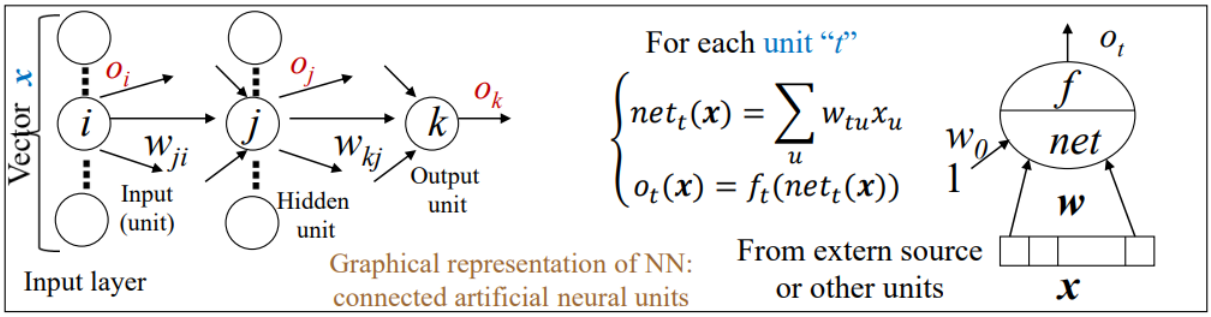
input layer, source of the input x. Copies the source external input patterns x, without computing net and f.

hidden layer, projects onto another hidden or an output layer, computing.

Can be viewed as network of units o flexible function:

$$h(x) = f_k\left(\sum_j w_{kj} f_j\left(\sum_i w_{ji} x_i\right)\right)$$

As for notation, given



We have that the index t denotes a generic unit, can be either k or j. u denotes a generic input component, either i or j.
x is a generic input from an external source (input vector) or from other units according to the position of the unit in the network. If we load the pattern x in the input layer, we can use the notation with o for both the inputs and the hidden units outputs. Hence, inside the network, the input to each unit t from any source u (through the connection w_{tu}) is typically denoted as o_u.

Architectures

Feedforward NN Standard architecture of the NNs. Direction: input $\longrightarrow$ output

Recurrent NN adds feedback loops connections to the network topology. These self-loop connections provide the network with dynamical properties, leaving a "memory" of the past computations inside the model. This allows us to extend the representation capability of the model to the processing of sequences and structured data.

Flexibility The hypothesis space is a **continuous space** of all the functions that can be represented by assigning the weight values of the given architecture. Depending on the class of values produced by the output units (discrete or continuous), the model can deal, respectively, with classification tasks (sigmoidal output f) or regression tasks (linear output f). Also multi-class and multi-regression, with multiple output units.

Universal approximation The flexibility is theoretically grounded (Cybenko 1989, Hornik et al. 1993...). In short, a single hidden-layer network with logistic activation functions can approximate (arbitrary well) every continuous function, provided enough units in the hidden layer.
A MLP network can approximate (arbitrarily well) every input-output mapping, provided enough units in the hidden layers.

Existence theorem: given ϵ, $\exists\, h(x) \mid |f(x) - h(x)| < \epsilon\ \forall\, x$ in the hypercube.

With this fundamental result (MLP can represent *any* function), two issues arise: how to learn by neural network and how to decide its architecture.
The **expressive power** of a NN is strongly influenced by the number of units and their configuration (architecture). The number of units can be related to the discussion of the VC-dim, specifically: the network capabilities are influenced by the number of parameters w^*, which is proportional to the number of units. Further studies also report the dependencies on their value sizes.

Es: weights = 0 ⇒ minimal VC-dim, small weights ⇒ linear part of the activation function,
high values weights ⇒ more complex model.
The universal approximation theorem is a fundamental contribution, showing that one hidden layer is sufficient in general, but it doesn't assure that a "small number" of units would do the work. For many function families it's possible to find boundaries on that number, but also cases for which a single hidden layer network would require an exponential number of units (in n input dimension).
More layers can help, but is it possible to efficiently train deep networks?

Learning Algorithm The learning algorithm allows adapting the weights w of the model in order to obtain the best approximation of the target function. Often realized in terms of minimization of and error/loss function on the training dataset. Same problem as other models: given a set of l training examples (x_p, d_p) and a loss measure L (ex. the MSE $L(h(x_p), d_p) = (d_p - h_w(x_p))^2$), find the weight vector w that minimizes the expected error on the training data

$$E(w) = R_{emp} = \frac{1}{l}\sum_{p=1}^{l} L(h(x_p), d_p)$$

Credit assignment problem: which credit to the hidden units? Not easy when the error signal is not directly measurable: we don't know the error (delta) or the desired response for the hidden units, which is useful for changing their weights. Supposed too difficult, but the backpropagation algorithm brought a renaissance of the NN field.
Loading problem, NP-complete: given a network and a set of examples, is there a set of weights so that the network will be consistent with the examples?

In practice, networks can be trained in a reasonable amount of time, although optimal solutions are not guaranteed. How to solve? Key steps:

> Credit assignment problem: how to change the hidden layer weights?
>
> Gradient descent approach can be extended to MLP (provided that loss and activations are differentiable functions, to find the delta for every unit in the network)

Backpropagation algorithm We need a differentiable loss, differentiable activation functions and a network to follow the information flow.
Find w by computing the gradient of the error function

$$E(w) = R_{emp} = \frac{1}{l}\sum_{p=1}^{l} (h(x_p) - d_p)^2$$

It has nice properties: easy because of the **compositional** form of the model, and keeps track only of the quantities local of each unit (local variables), so the **modularity** of the units is preserved.

```
def backprop():
  # 1. Initialize all the weights w in the network and eta
  # 2. Compute out and e_tot
  while e_tot < epsilon:  # with epsilon desired value or other criteria
    for w_i in w:
      d_w_i = - (gradient of e_tot respect to w_i)  # step (1)
      w_new = w + eta*d_w_i + ...  # step (2)
    # Compute out and e_tot
  end
```

Step (1)

$$\Delta w = -\frac{\partial E_{tot}}{\partial w} = -\sum_p \frac{\partial E_p}{\partial w} = \sum_p \Delta_p w$$

Issues in training neural networks Heuristic guidelines in setting the backward propagation (backprop) algorithm. Generally, the models are over-parametrized, the optimization problem is not convex and is potentially unstable. We will discuss few of the issues. A good interpretation is to see backprop as a path trough the loss/weight space. The path depends on: data, neural network, starting point (initial weight values), rate of convergence, final point (stopping rule). This defines a control for the search over the hypothesis space. The basic algorithm, once again, is:

1. Start with weight vector $w_{initial}$ and fix $0 < \eta < 1$
2. Compute $\Delta w = -$ gradient of $E(w) = -\frac{\partial E(w)}{\partial w}$ (or for each w_i)
3. Compute $w_{new} = w + \eta \cdot \Delta w$ (or for each w_i)
   η is the step size or **learning rate**
4. Repeat from 2 until convergence or $E(w)$ sufficiently small

But now the $\Delta w = -\frac{\partial E(w)}{w}$ were obtained through backprop derivation/algorithm for any weight in the network. At step 2, to compute the error, we first apply inputs to the network computing an output (**forward phase**), then we retro-propagate the deltas for the gradient (**backward phase**). So how to choose the $w_{initial}, \eta$ and the convergence?

Starting values In the basic algorithm, $w_{initial}$. The weights are initialized with random values close to zero. To be avoided: all zero, high values or all equal (symmetry), because this hampers the training. For standardized data, values $in[-0.7, +0.7]$. There are other heuristics: $range \cdot \frac{2}{fanin}$ with $fanin$ being the number of inputs to a hidden unit, or orthogonal matrices...

Multiple minima The loss is not convex, has local minima. This affects the results, which depends on the starting weight values, hence: try a number of random starting configurations (5-10 or more **training runs** or **trials**). Useful taking the mean results (mean of errors) and looking at the variance to evaluate the model and then, if only one response is needed: we can choose the solution giving the lowest (penalized) validation error or we can take advantage of different end points and outputting a mean of the outputs (**committee approach**).
A "good" local minima is often sufficient: in ML we don't need the global or local minimum on R_{emp}, as we are searching the minimum of R (which we can't compute). Often we stop early, in a point of non-zero gradient so being neither a local nor a global minima for the training error. The neural network builds a variable size hypothesis space, so VC-dim increases during training and the training error decreases toward zero (or global minimum) while the neural network becomes too complex. We **stop before this condition of overtraining**, avoiding overfitting.

Online/batch Batch version: sum all the gradients of each pattern over an epoch and then update the weights (Δw after each epoch of l patterns). Online/stochastic: upgrade w for each pattern p, making progress with each example it sees. Faster but need smaller η.

Batch

1. Start with weight vector $w_{initial}$ and fix $0 < \eta < 1$
2. Compute $\Delta w = -$ gradient of $E(w)$ on the entire training set (**epoch**)
3. Compute $w_{new} = w + \eta \cdot \Delta w$ (or for each w_i)
4. Repeat from 2 until convergence

More accurate estimation of gradient **Online**

1. Start with weight vector $w_{initial}$ and fix $0 < \eta < 1$
2. For each pattern p
   (a) Compute $\Delta_p w = -$ gradient of $E(w)$
   (b) Compute $w_{new} = w + \eta \cdot \Delta_p w$ η is the step size or **learning rate**
3. Repeat from 2 until convergence

Since the gradient of a single data point can be considered a noisy approximation to the overall gradient, this is also called stochastic gradient descent.

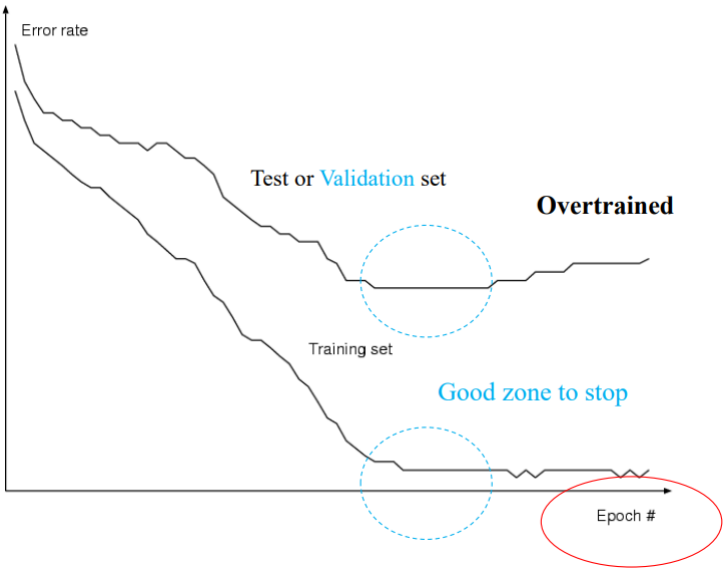
Many variations exists, for example stochastic gradient descent with minibatch.

Learning rate With batch training we have more accurate estimation of gradient, higher η. With online we have training faster but potentially unstable, lower η. So high vs low $\eta \Leftarrow$ fast but unstable vs slow but stable. Typically $\eta \in [0.01, 0.5]$.
The learning curve, plotting the errors during training, allows to check the behavior in the early phases of the model design. Of course the absolute value depends also on model capability and other hyperparameters, but η plays a big role in the curve quality. It's useful to have the mean of the gradients over the epoch: uniform approach (Least **Mean** Square). Some improvements: momentum (Nesterov), variable and adaptive learning rates, varying depending on the layers (in deep networks)...
With momentum it becomes $\Delta w_{new} = -\eta \frac{\partial E(w)}{w} + \alpha \Delta w_{old}$, saving Δw_{new} in Δw_{old} for the next step. Becomes faster in plateaus but damps in oscillations (inertia effect, allows higher η)
Can be used in online by considering the previous example, Δw_{p-1} as Δw_{old}.
It smooths the gradient over different examples. A variant is to evaluate the gradient after the momentum is applied

(so using $\overline{w} = w + \alpha \Delta w_{old}$), improves the rate of convergence for the batch mode (not online!).
The variable learning rate starts high and decays linearly for each step until iteration τ, using $\alpha = \frac{s}{\tau}$ with s current step so that $\eta_s = (1-\alpha)\eta_0 + \alpha\eta_\tau$, then stops and uses a fixed small η_τ. Set up as $\eta_\tau = 1\%$ of η_0 and τ as few hundred steps. η_0 same no instability/no stuck trade off.
With adaptive learning rate, it's automatically adapted during training possibly avoiding/reducing the fine tuning phase via hyperparameters selection.

Stopping criteria When to stop training? Basic: error. The best metric if we know the tolerance of data. Other: max tolerance instead of mean, number of misclassified, no more relevant weight changes or no more significant error decreasing.
In any case stop after too many epochs, but avoid stopping at an arbitrary fixed number of epochs, and not necessarily stop with very low training error.
Control of complexity is the main aim to achieve best generalization capability.

Overfitting and regularization Typically, stopping at the global minimum of $R_{emp}(w)$ is likely to be an overfitting solution. The control of complexity is our main aim to achieve the best generalization capability. For instance, we need to add some **regularization**: can be achieve directly (**penalty term**) or indirectly (**early stopping**). Model selection with cross validation on empirical data to find the trade-off.
In neural networks, we start learning with small random weights (breaking the symmetry!). As optimization proceeds, hidden units tend to saturate, increasing the effective number of free parameters (hence increasing the VC-dim). As we discussed, this is a variable-sized hypothesis space (changes during training).



How to act on the overtraining?

1. Early stopping: using a validation set to determine when to stop (vague indication: when the validation error increases, so use more than one epoch before estimating (**patience**))
   Since the effective number of parameters grows during the training, halting the process effectively limits the complexity.

2. Regularization on the loss: we can optimize the loss considering the weights values.
   Related to Tikhonov, so well principled approach: add a penalty term to the error function: $Loss(w) = \sum_p (d_p - f(x_p))^2 + \lambda ||w||^2$ with $||w||^2 = \sum_i w_i^2$
   The effect is a weight decay, basically $w_{new} = w + \eta \cdot \Delta w - 2\lambda w$.
   λ is the **regularization parameter**, generally very low (0.01) and selected in the model selection phase. Applied on the linear model it's the ridge regression.
   More sophisticated penalty terms have been developed (ex: weight elimination, Haykin). Misunderstandings:

   > Regularization is not a technique to control the stability of the training convergence but controls the **complexity of the model**, measured by VC-dim and related to the number of weights and values of the weights in the neural networks

Early stopping needs a validation set to decide when to stop, which sacrifices some data. The regularization is a principled approach, as it allows the VL curve to follow the TR curve so that early stopping is not needed.
But you can use both!

Typically the bias (w_0) is omitted because its inclusion causes the results to be not independent from target shift/scaling. May be included with its own regularization coefficient.
Typically it's applied in the batch version. So for online/mini batch we need to take into account possible effects over many steps, so it's better to use $\lambda \cdot \frac{mb}{l}$ with l being the number of total patterns.
Other techniques: **dropout**.

3. Pruning methods

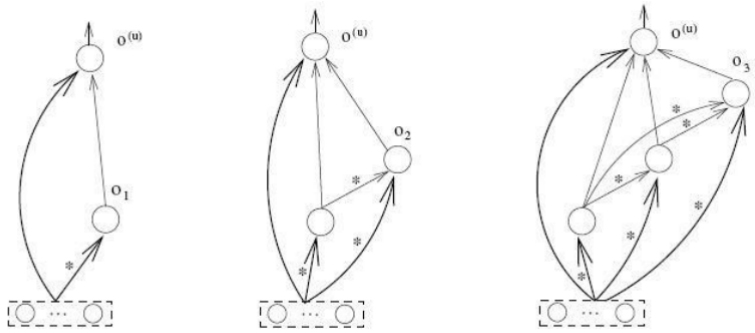
Number of units This is related to the control of complexity but also to the input dimension and the size of the TR set. In general, this is a model selection issue. The number of units, along with the regularization parameters, can be selected with the model selection phase (cross-validation).
Too few units ⇒ underfitting, viceversa too many units ⇒ overfitting. The number can be high with proper regularization.

Constructive approaches: the learning algorithm decides the number of hidden units, starting with small networks and adding new units.
Incremental approach: algorithms that build a network starting with a minimal configuration and add new units and connections during training. Examples: Tower, Tiling Upstart for classification and Cascade Correlation for both regression and classification.

The **Cascade Correlation** algorithm starts with N0, a network without hidden units, which is trained and evaluated. If N0 cannot solve the problem, go to N1: a hidden unit is added such that the correlation between the output of the unit and the residual error of N0 is maximized (by training its weights).
After training, the weights of the new unit are frozen and the remaining weights are restrained. If the obtained network N1 cannot solve the problem, new hidden units are progressively added, which are connected with all the inputs and previously installed units. The process continues until the residual errors of the output layer satisfy a specified stopping criteria.
This method dynamically builds up a neural network and terminates once a sufficient amount of hidden units has been found to solve the given problem. Specifically, it works by interleaving the minimization of the total error function (LMS) by simple backpropagation training of the output layer and the maximization of the (non-normalized) correlation (the covariance) of the new inserted hidden (candidate) units with the residual error. With $E_{p,k} = (o_{p,k} - d_{p,k})$ residual error, with $o_{p,k}$ output, p pattern and k output unit

$$S = \sum_k \left| \sum_p (o_p - mean_p(o_{p,k}))(E_{p,k} - mean_p(E_{p,k})) \right|$$

$$\frac{\partial S}{\partial w_j} = \sum_k sign(S_k) \sum_p (E_{p,k} - mean_p(E_{p,k})) f'(net_{p,h}) I_{p,j} \text{ with } h \text{ candidate index}$$



© A. Micheli 2003

* = weights frozen after candidate training

Pruning methods: start with a large network and progressively delete weights or units.