# Machine Learning 23-24

Notes

University of Pisa

M.Sc. in Computer Science

# Contents

# Chapter 1

# Predictive Models

The most common framework in ML is to construct **predictive models** capable of making predictions by looking at a set of data which may be uncertain, noisy, or incomplete. Predictive models are also useful when the theory behind a certain phenomenon is poor or completely absent. ML studies and proposes methods to infer **functions** from **examples** of observed data. Such function must:

- **fit** the example data;
- **generalize** with reasonable accuracy.

A simple example might be recognizing handwritten numbers. The model is presented with some real-life data (pictures of numbers written on paper), and then approximates a function capable of recognizing numbers out of pictures never seen before.

## 1.1 Components of a Predictive System

A predictive system can be broken down into the following key components:

- Data;
- Task;
- Model;
- Learning algorithm;
- Validation;
- Prediction.

### 1.1.1 Data

The data represents the available facts. Can be either flat (attribute-value language, fixed-size vector), or structured (more complex data structures).

**Flat data** can be represented as a table with multiple rows and columns.

- Rows: **examples**, patterns, instances, samples...
- Columns: **features**, attributes, elements, components...
- The number of examples is the dimension of the dataset;
- The number of features per example is the dimension of the input;
- Examples and attributes can be indexed: e.g., $x_i$ is the $i^{th}$ feature of element $x$; $x_i$ is the $i^{th}$ element; $x_{p,i}$ is the $i^{th}$ feature of the $p^{th}$ example.

It is often useful to **encode** the data. Structured data is a complex data structure, such as a list, tree, graph, multi-relational data, etc.

**Noise**

Noise is the addition of external factors to the stream of information caused by randomness in measurements (which are never always 100% accurate).

**Outliers**

Outliers are unusual data values, inconsistent with most observations (may be caused by incorrect measurements). They can be removed by appropriate preprocessing (outlier detection).

**Feature selection**

The selection of a subset of features deemed informative. Can provide an optimal input representation for the data.

### 1.1.2 Task

The task defines the purpose of the application: what information do we want to achieve? What can the result be used for? What data is available?

**Supervised Learning**

- **Given**: a **training set (TR)** consisting of training examples $\boldsymbol{< input, output > = < x, d >}$ (where $d$ is the **target value**, a categorical or numerical **label**) for an unknown function $\boldsymbol{f}$ (called **target function**).
- **Find**: a "good" approximation of $f$, called **hypothesis $\boldsymbol{h}$**, that can be used on unseen data to predict its value according to $f$ (= is able to **generalize**).

**Unsupervised learning** is defined the same as the above, but the input data is a TR of unlabeled data.

**Types of tasks**

- **Classification**: input vectors are seen as members of two or more classes; the goal is to assign a class to any input $x$. The function produces discrete valued outputs (corresponding to the different classes found).

  If the number of classes is 2, the task is binary classification. Otherwise, it's a multi-class problem.
- **Regression**: estimate a real-value function on the basis of a finite set of (noisy) examples.

These are the two most common supervised tasks. Some other tasks will be seen later.

### 1.1.3 Model

A model describes the relationships among the data by using a ”language” (numerical, symbolic,...). The model also defines the set of function the learning machine can implement: the **hypotheses space H**.

Many models exist as of today: linear models, symbolic rules, probabilistic models, etc. Why so many? Because *there is no free lunch*: each model is effective and efficient for some problems and not others, has different levels of flexibility, and has different ways to control its complexity.

### 1.1.4 Learning Algorithm

Given data, task, and model, the learning algorithm executes an **heuristic search** in the hypotheses space H, in order to find the “best” hypothesis that approximates the target function. By “best”, we mean the function with the **minimum error**. In order to setup a model, we must make assumptions about the target function, regarding either/both:

- Constraints in the model: **language bias**;
- Constraints in the learning algorithm: **search bias**.

To explain why such biases are needed, an example is presented. We want to approximate a boolean function $f$, and we are given a certain number of examples, each of $n$ features, and paired to a label (the output, 0/1). If we want to search for the best approximation of $f$, we would have to search throughout all the boolean functions in the hypotheses space H; the dimension of H is $2^{2^n}$ ($2 =$ number of possible outputs, $2^n =$ number of possible inputs). Our function would be essentially **incapable of generalizing**: if the learner receives an input it already saw, it can classify it, otherwise it can't produce an output (it's a simple lookup table).

$$\boxed{\textbf{NO INDUCTIVE BIAS} \rightarrow \textbf{NO GENERALIZATION}}$$

We can introduce a language bias to limit the number of hypotheses to search through. For example: let us use conjunctive rules to express the hypotheses (functions can only be propositions connected by the AND operator). The number of total functions that can be expressed in this notation lowers to $3^n + 1$ (total of $n$ possible literals per rule, each literal can be either true, false, or not appear at all; the extra 1 is the hypothesis that always evaluates to false).

### Consistency and version space

An hypothesis $h$ is **consistent** with a training set TR if $h(x) = d$ for each training example $< x, d >$ in TR. The set of all hypotheses in H consistent with a TR is called **version space (VS)** (can also be referred to as $\boldsymbol{VS_{H,TR}}$).

Many algorithms exist that can perform a complete search in the hypotheses space H, as in, that are capable of identifying the version space with respect to a TR. However, when using language biases, the restriction of the hypotheses space may be excessive, with the risk of excluding the target function: e.g., referring to the previous example, any rule that uses disjunction (OR operator) would be excluded from the search altogether.

The other approach is to instead use a language capable of expressing all possible hypotheses, and introduce a search bias, using an incomplete search strategy.

### Loss and error

In order to measure the "goodness" of an hypothesis $h$, an inner measure is used, called the **loss function (or loss measure)**: $\boldsymbol{L(h(x), d)}$. The expected value of $L$ is called the error relative to $h$.

The loss function and the error can be calculated in any different ways depending on tasks. The following list presents some commonly used loss and error functions ("$w$" in the error function refers to the vector of parameters that identify that specific $h$).

- **Regression**: predicting a numerical value.

  **H**: set of real-valued funcs.

  **L**: $L(h(x_p), d_p) = (d_p - h(x_p))^2$ (squared error)

  **E**: $E(h_w) = \frac{1}{l} \sum_{p=1}^{l} (d_p - h_w(x_p))^2$ (mean squared error)

- **Classification**: classifying data into two or more classes.

  **H**: a set of indicator functions (discrete-valued).

  **L**: $L(h(x_p), d_p) = \begin{cases} 0 & \text{if } h(x_p) = d_p \\ 1 & \text{otherwise} \end{cases}$

  **E**: $E(h_w)$ = mean loss over the data set providing % of misclassified patterns

- **Clustering**: partitioning data in clusters, each approximated by a centroid/ prototype.

  **H**: set of vector quantizers.

  **L**: $L(h(x_p), d_p) = (x_p - h(x_p)) * (x_p - h(x_p))$ (squared error distortion)

  **E**: TBD

- **Density estimation**: estimating density from an assumed class.

  **H**: set of density functions.

  **L**: $L(h(x_p), d_p) = -\ln h(x_p)$

  **E**: TBD

Error is calculated by measuring loss on new samples of data (a set of labeled examples).

### 1.1.5 Validation

Validation is the evaluation of performances in ML systems. Any $h$ that approximates the target function $f$ correctly on training examples should also approximate $f$ well on unseen examples.

**Overfitting**

A learner **overfits** the data if it outputs an hypothesis $h \in H$ with true/generalization error $R$ and empirical (training) error $E$, but there's a $h' \in H$ with error $R' < R$ and $E' > E$.

**OVERFITTING DOES NOT DEPEND ON NOISE**

Overfitting is caused by choosing a model with a complexity such that it adapts too much to the data (noisy data as well), but the presence of noise does not automatically equate to an overfit model.

Underfitting

A learner **underfits** the data if it outputs an hypothesis with both very high values of generalization error $R$ and empirical error $E$; i.e., the model is too simple to accurately approximate the target function.

## 1.2 Statistical Learning Theory

Statistical Learning Theory (SLT) studies the properties of models to determine their generalization capability, with respect to the training error and underfitting/overfitting zones.

### 1.2.1 VC-bound

To approximate a target function $f$, the learner must find the hypothesis $h$ with the minimum **risk function**: $\boldsymbol{R = \int L(d, h(x)) dP(x, d)}$, where $P(x, d)$ is a probability distribution. However, when constructing a model, we can only calculate what is the value of the **empirical risk** $\boldsymbol{R_{emp}}$ (also called training error E). We can use it to approximate $R$ by indicating an upper bound for it.

Given the **VC-dim / VC** (Vapnik-Chervonenkis-dim), a measure of the complexity of $H$, then it holds with probability $1 - \delta$ the following inequality:

$$\boldsymbol{R \leq R_{emp} + \epsilon(1/l, VC, 1/\delta)}$$

Function $\epsilon$ is called **VC-confidence** and is defined so that it is directly proportional to $VC$, and inversely proportional to $l$ (the dimension of the TR) and $\delta$ (the chosen confidence). With this bound, we can explain undefitting and overfitting in a more formal way:

- **If l is big**: value of $\epsilon$ is low, bound mostly depends of $R_{emp}$ → **enough data to approximate target function**;
- **If l is low**: value of $\epsilon$ is high causing the upper bound to increase, regardless of $R_{emp}$ → **too little data to approximate target function**;
- **If VC-dim is high**: $R_{emp}$ is low but $\epsilon$ is high, model may be too complex → **overfitting**;

- **If VC-dim is low**: $\epsilon$ is low but $R_{emp}$ is high, model may be too simple → **underfitting**.

The goal now is to minimize this bound by finding the best trade-off between model complexity and fitting of training data.

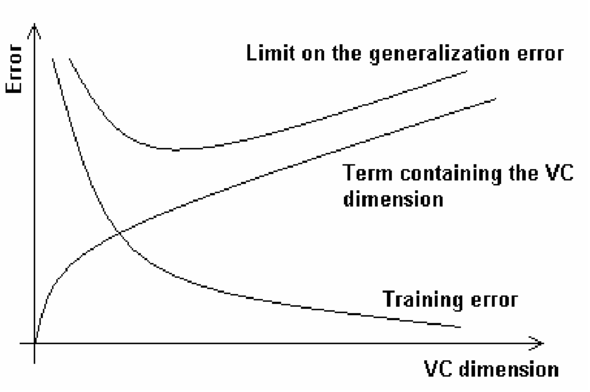


Figure 1.1: Graphical visualization of how generalization error and training error vary as the VC-dim of the model changes.

### 1.2.2 Validation

Validation takes place after model training, which determines the weight parameters $w_0, w_1, \dots w_n$ to use for the $h$. The construction of a model can be divided into:

- **Model selection**: estimating the generalization error of different models and choosing the best one (this includes searching for the best hyper-parameters of the model) → it returns a model
- **Model assessment**: once a model has been chosen and both parameters and hyper-parameters have been adjusted, evaluating its generalization error on new test data as a way to measure its "quality" → it returns an estimation

Among the different techniques, two simple ones are hold-out cross validation and k-fold cross validation.

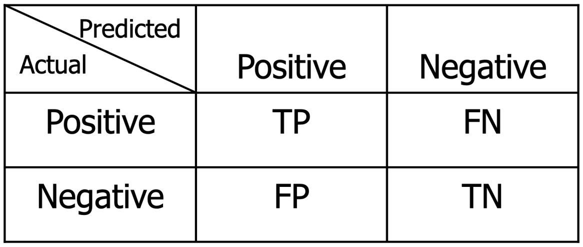
With hold-out cross validation, the entire available dataset is divided into three partitions: the training set TR, the validation set VL, and the test set TS. TR is used to run the training algorithm; VL is used to select the best model; TS is used to measure the quality of the final model.

Simple hold-out cross validation can make insufficient use of data. An alternative method is called k-fold cross validation. The dataset $D$ is split into $k$ partitions,

$D_1, D_2, ..., D_k$. The training algorithm uses $D/D_i$, and then the validation/test phase is done by using $D_i$. This way all the data is used for both training and validation/testing. The main issues in using this method are choosing the value of $k$, and the fact that's pretty computationally expensive.

**Confusion Matrix** The performance of a classifier can be represented using four counts (True Positives, True Negatives, False Positives, False Negatives), and calculating evaluation measures that summarize the information provided by these counts. The most common measures are:

- **Specificity**: $\dfrac{TN}{FP+TN}$
- **Sensitivity**: $\dfrac{TP}{TP+FN}$
- **Precision**: $\dfrac{TP}{TP+FP}$
- **Accuracy**: $\dfrac{TP+TN}{total}$



| Predicted<br>Actual | Positive | Negative |
|---|---|---|
| Positive | TP | FN |
| Negative | FP | TN |

Figure 1.2: Confusion matrix.

# Chapter 2

# Linear and K-NN Models

## 2.1 Linear models

### 2.1.1 Univariate Linear Regression

**Unidimensional Input**

The simplest form of regression, with one input variable $x$, and one output variable $y$. We assume a model expressed as:

$$h_w(x) = w_1 x + w_0$$

where $w_0$ and $w_1$ are real-valued coefficients (or parameters/weights). This hypothesis corresponds to a simple line. Learning such a function with LMS (least mean squares) means finding the $w$ that minimize the residual sum of squares; i.e., find $\arg\min_w E(w)$ in $L_2$ (where $E(w) = \frac{1}{l}\sum_{p=1}^{l}(y_p - h_w(x_p))^2$).

In order to find such value of $w$, we must find the local minimum of the error function, corresponding to the point where the gradient is equal to 0:

$$\frac{\partial E(w)}{\partial w_i} = 0,\ i = 0, \dots, n$$

For the univariate case, $w$ has dimension = 2, so we must search the value of $w$ such that:

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

Below is how the partial derivative of $E(w)$ is calculated for a generic $w_i$:

$$\frac{\partial E(w)}{\partial w_i} = 2\sum_{p=1}^{l}(y_p - h_w(x_p)) \cdot \frac{\partial(y_p - h_w(x_p)}{\partial w_i} = \boxed{2\sum_{p=1}^{l}(y_p - h_w(x_p))\frac{\partial(y_p - w_0 - w_1 x)}{\partial w_i}}$$

By calculating for $w_0$ and $w_1$, we get:

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

$$\frac{\partial E(w)}{\partial w_1} = -2\sum_{p=1}^{l}(y_p - h_w(x_p)) \cdot x.$$

**Multidimensional Input**

When the input has dimension $n > 1$, the function $h_w(x)$ can be expressed as:

$$h_w(x) = w_0 + \sum_{i=1}^{n} w_i x = w_0 + w_1x_1 + w_2x_2 + ... + w_nx_n = \boxed{w^T x + w_0}$$

$w_0$ is often called the **intercept/threshold/bias/offset...**, since $h_w(x) = 0$ is equivalent to $w^T x = -w_0$.

### 2.1.2 Classification

The same model(s) used for regression can be used for classification as well. Given a set of $l$ points, equally distributed in two classes, we want to find an **hyperplane**, represented by $w^T x + w_0$, that can help decide whether a new point belongs to one class or the other. The function to approximate will be in the form:

$$h_w(x) = sign(w^T x + w_0) \qquad \text{or, alternatively, } h_w(x) = \begin{cases} 1 & \text{if } w^T x \geq -w_0 \\ 0 & \text{otherwise} \end{cases}$$

both of whom classify a point depending on where it sits with respect to the defined decision boundary.

### 2.1.3 Learning Algortihms

The following section will present two learning algorithms, both using a linear model and based on LMS. The first algorithm uses a direct approach, based on the normal equation solution. The second algorithm offers an iterative solution based on gradient descent.

The first step is redefining the learning problem and the loss. Assuming a classification problem:

- **Given**: a set of $l$ examples $(x_p, y_p)$, and a loss function $L$

- **Find**: The weight vector $w$ that minimizes the expected loss on the training data $R_{emp} = \dfrac{\sum_{p=1}^{l} L(h_w(x_p), y_p)}{l}$

Let's assume error is being calculated by using least squares (as for the regression), so $E(w) \propto \sum_{p=1}^{l} (y_p - w^T x_p)^2$.

The issue with trying to minimize the expected loss is that using a piecewise constant function to calculate it's value can make it a difficult problem, since the function is not differentiable. The solution is to ”soften” the loss function by using a smooth, differentiable function instead.

The problem can be reformulated as:

- **Given**: as above
- **Find**: find $w$ to minimize the residual sum of squares:
  $E(w) = \sum_{p=1}^{l} (y_p - x_p^T w)^2 = ||y - Xw||^2$,
  where X is a matrix $l \times n$ where each row is an input vector $x_p$.

Since this (differentiable) error function is quadratic, the minimum error always exists (but may not be unique). Let's differentiate the function:

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

$$2\sum_{p=1}^{l} (y_p - x_p^T w) \cdot (\frac{\partial y_p}{\partial w_j} - \frac{\partial x_{p,1} w_1}{\partial w_j} - \frac{\partial x_{p,2} w_2}{\partial w_j} - \cdots - \frac{\partial x_{p,n} w_n}{\partial w_j}) =$$

$$\boxed{-2\sum_{p=1}^{l} (y_p - x_p^T w)^2 \cdot x_{p,j}}$$

The gradient can be also expressed as:

$$\frac{\partial E(w)}{\partial w_j} = -2\sum_{p=1}^{l} \delta_p \cdot s_{p,j}$$

This form will be used later.

### Singular Value Decomposition (direct approach)

We can get the **normal equation** (the point with gradient of $E$ equal to 0):

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

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

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

where $X^+$ is the **Moore-Penrose pseudoinverse** of X.

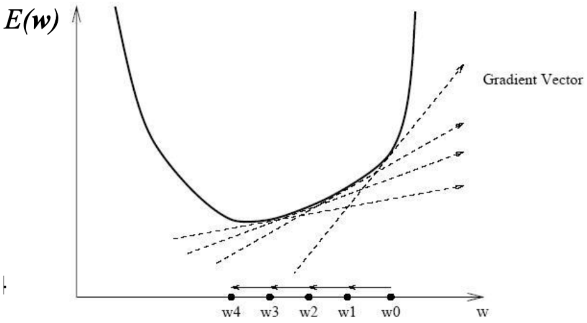
If $X^TX$ is singular, then the solutions are infinite. The **singular value decomposition (SVD)** can be used to compute this pseudoinverse:

$$X = U\Sigma V^T \implies X^+ = V\Sigma^+U^T$$

where $\Sigma$ is the diagonal, and $\Sigma^+$ is the diagonal where all nonzero entries are replaced by their reciprocal. Moreover, we can apply SVD directly to calculate $w = X^+y$.

### Gradient Descent (iterative approach)

The **gradient** is the ascent direction; it expresses in which direction the error function grows. So, by moving with a **gradient descent**, we can gradually move towards the minimum of the function. The search begins with some initial values given to the weight vector, then the values are modified iteratively to minimize the error function (local search).



So, for each iteration, the amount $w$ should be changed with is calculated as:

$$\Delta w = -\frac{\partial E(w)}{\partial w} = \begin{bmatrix} -\dfrac{\partial E(w)}{\partial w_1} \\ -\dfrac{\partial E(w)}{\partial w_2} \\ \dots \\ -\dfrac{\partial E(w)}{\partial w_n} \end{bmatrix} = \begin{bmatrix} \Delta w_1 \\ \Delta w_2 \\ \dots \\ \Delta w_n \end{bmatrix}$$

Each new value of $w$ will be calculated as: $w_{new} = w + \eta * \Delta w$, where $\eta$ is the **learning rate**: the "step size" that controls the speed of the gradient descent. The algorithm is the following:

1. Start with $w_{initial}$ (small), fix value of $\eta \in [0, 1]$
2. Compute $\Delta w = \dfrac{\partial E(w)}{\partial w}$
3. Compute $w_{new} = w + \eta * \Delta w$
4. Repeat (2) and (3) until convergence, or $E(w)$ is sufficiently small.

To use least mean squares, divide $\Delta w$ by $l$.

This way of calculating $w$ is called an **error correction rule** (also called **delta rule** since it refers to $\delta_p = y_p - x_p^T w$), since it changes the value of $w$ proportionally to the error to "correct" it:

- $\boldsymbol{(input_j > 0, error > 0)}$ → $\Delta w$ is $+$ → $w_j$ must be increased → increment output → **less error**
- $\boldsymbol{(input_j > 0, error < 0)}$ → $\Delta w$ is **-** → $w_j$ must be decreased → reduce output → **less error**
- $\boldsymbol{(input_j < 0, error > 0)}$ → $\Delta w$ is **-** → $w_j$ must be decreased → decrement output → **less error**
- $\boldsymbol{(input_j < 0, error < 0)}$ → $\Delta w$ is $+$ → $w_j$ must be increased → increase output → **less error**

This is a simple and effective local search approach to LMS, since it allows us to search through an infinite hypothesis space, and can be applied to models other than linear ones; however, it's not terribly efficient.

### Batch and On-line (stochastic) Gradient Descent

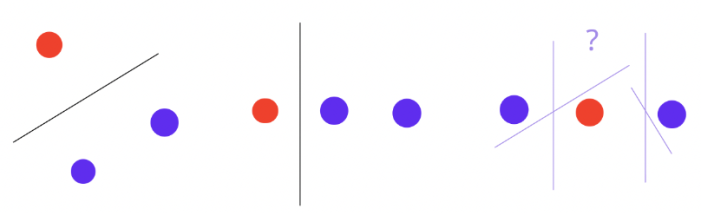
The **batch** version of the algorithm, the gradient is calculated as the sum over all $l$ patterns, which provide a more "precise" evaluation off the gradient over the data-set. After the sum, the weights are updated.

The **on-line** version upgrades the weights with the error computed for each pattern; so, first the weights are updated calculating the error for the $1^{st}$ pattern; those weights are used to calculate the error on the second pattern, updating the weights again; the new weights are then used to calculate the error on the third pattern, and so on. This method can be faster, but needs a smaller $\eta$.

### 2.1.4 Extending the Linear Model

A problem in $n$ dimensions is **linearly separable** if it's possible to construct a $n-1$-dimensional hyperplane that separates the points into two groups. The linear decision boundary can provide exact solutions only for these problems.

For example, given 3 points, we can always find a separation plane for every assignment of $f(x)$ (class), save for when all three points are aligned on a straight line and the point in the middle belongs to a different class than the two at the edges:



So, instead of using a linear model, we can use transformed inputs with non-linear relationships with the output, so, for example, we can use a function like the following:

$$h_w(x) = w_0 + w_1 x + w_2 x^2 + \cdots + w_m x^m = \sum_{i=0}^{m} w_i x^i$$

#### Linear Basis Expansion

This transformation is done by using the **linear basis expansion (LBE)**:

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

where each $\phi_k()$ is a function that ”transforms” the input; in the previous example, $\phi_0(x) = x^0$, $\phi_1(x) = x^1$, $\phi_2(x) = x^2$, and so on.

Typically, the number of parameters $K$ is greater than $n$. The model is still linear in the parameters (in $\phi$, not in $x$), so we can still use the same learning algorithm as before, both for regression and classification (by applying the sign function).

#### Choosing Phi

This type of approach is called a ”dictionary” approach, since it constructs the linear base expansion out of the function. Using this method can model more complicated relationships (non-linear) since it's more expressive; on the other hand, there's a high risk of overfitting, so we need methods to control the complexity of the model.

### Tikhonov Regularization

Controlling the complexity can be done in many different ways. One of them is called **ridge regression (or Tikhonov regularization)**. This method involves adding constraints to the sum of $|w_j|$, penalizing models with high values of $|w|$; i.e., favoring models that use less terms by setting most of the weights to 0 (which results in a simpler model). The loss is calculated as:

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

where $\lambda$ is the regularization hyper-parameter, a small positive value chosen during the model selection phase. Also note that here the name Loss (used for model training) is used to distinguish from Error E (used to evaluate the model error, and corresponding to the data term in this Loss function).

Reformulating the two approaches seen before (direct and iterative), they become as follows:

- **SVD (direct)**: $w = (X^T X + \lambda I)^{-1} X^T y$
- **Gradient descent (iterative)**: $w_{new} = w + \eta \Delta w - 2\lambda w$ (this is called a weight decay technique; even when the gradient is 0, this addition still reduces the weight by a fraction of it)

A balancing can be noted between the value of $\lambda$ and the complexity of the model:

- **Small $\lambda$ value**: the focus is obtaining a small data error term (first term) with a too complex model → overfitting;
- **Large $\lambda$ value**: the focus is on the second term, so the data error could grow too much → underfitting.

The goal is to find the ideal value of $\lambda$ that's neither too small nor too big. We can make a connection between Tikhonov regularization and statistical learning theory; controlling the value of $\lambda$ means controlling the $VC - dim$ of the model, after all: high $\lambda$ means a smaller $VC - dim$ and vice-versa.

### Other Regularization Techniques

- Ridge regression: $\|\|_2$
- Lasso: $\|\|_1$

- Elastic nets: both $\|\|_1$ and $\|\|_2$

The L2 norm penalizes the square value of the weights, and tends to drive all weights to smaller values. L1 norm, on the other hand, penalizes the absolute value of the weights, and tends to drive some weights to exactly 0, but it also introduces a non differentiable loss.

### 2.1.5 Multi-class Task

- **OVA (one versus all)**: a discriminant function for each class, built on top of real-valued binary classifiers. We train $k$ classifiers, each to distinguish the examples in a single class from the examples in any other remaining class; to classify a new example, all $k$ classifiers are run, and the classifier that outputs the largest value is chosen.

- **AVA (all versus all)**: each classifier separates a pair of classes; one classifier for each possible pair. to classify a new example, all the classifiers are run, and the winner is the one with the max sum of outputs or the class with the most votes

## 2.2 K-Nearest Neighbor

The linear model employs an **eager** learning algorithm: it analyzes the data and constructs an explicit hypothesis capable of generalizing. The final model does not store the training data.

K-Nearest Neighbors instead uses a **lazy** approach, in which the training data is stored as is; once a new input is presented, it constructs an *ad hoc* hypothesis to classify only that one input.

Its inductive bias is assuming that we can use the distance between points to estimate the output value of a new input.

### 2.2.1 1-Nearest Neighbor

The simplest possible algorithm. All the training data $TR =< x_p, y_p >$ is stored. Once an input $x$ is received, it finds the nearest neighbor, i.e., the point $x_i$ such that $\arg\min_i d(x, x_i)$. The distance can be calculated in many different ways (typically euclidean distance). The output for $x$ will be $y_i$, the target value of $x_i$.

This learning algorithm produces a model with no training error, producing decision boundaries that are not linear. Obviously, this leads to overfitting, so the solution is to somewhat increase the training error in order to lessen the generalization error.

### 2.2.2 K-Nearest Neighbors

Instead of only looking at one neighbor, the output value is calculated by averaging the value of a point's $k$ nearest neighbors. The algorithm first calculates $N_k(x)$, a neighborhood of $x$ that contains $k$ neighbors. Then, the output for regression is calculated as the average target value across that neighborhood:

$$h(x) = avg_k(x) = \frac{\sum_{x_i \in N_k(x)} y_i}{k}$$

The output for classification is calculated by majority voting. For a binary classifier it's simply:

$$h(x) = \begin{cases} 1 & avg_k(x) > 0.5 \\ 0 & else \end{cases} ,$$

or, in the case of multiple classes:

$$h(x) = \arg\max_v \sum_{x_i \in N_k(x)} 1_{v,y_i} ,$$

$$1_{v,y_i} = \begin{cases} 1 & v = y_i \\ 0 & else \end{cases}$$

The model implicitly uses a **Voronoi diagram**.

A variant can be used with **weighted distances**: we may want to add weights to the contribution given by neighbors (i.e. closer neighbors influence the output more than farther ones). So the output for regression is calculated as:

$$h(x) = \frac{1}{k} \sum_{x_i \in N_k(x)} \frac{y_i}{d(x, x_i)^2} ,$$

while the one for classification as:

$$h(x) = \arg\max_v \sum_{x_i \in N_k(x)} \frac{1_{v,y_i}}{d(x, x_i)^2}$$

The main issue in constructing such a model lies in choosing the value of $k$. As seen before, if $k$ is too small, there's overfitting. If $k$ is too big, then both training and generalization errors increase, leading to underfitting.

**Bayes Error Rate**

A Bayes classifier is a learner that uses Bayes' rule to evaluate the most probable output. To find the optimal solution for a classifier, since we know the density $P(x, y)$ (unrealistic situation), we classify to the most probable class: $h(x) = \arg\max_v P(v|x)$. The error rate of the optimal classifier is called the **Bayes rate**. It corresponds to the minimum achievable error rate given the distribution of the data.

The K-NN classifier approximates this solution, except that the conditional probability looking at all the data is relaxed to a conditional probability that only refers to a neighborhood of the point, and the actual probability density is unknown.

### 2.2.3 Cons of K-NN

1. **Scale changes affect the model**: the algorithm does not work as intended is the data is scaled (and thus distances are distorted); the data must be normalized first.

2. **High computational cost**: K-NN does not construct a single approximation function; instead, it constructs a new approximation for each input it receives, so the computational cost is completely deferred to the prediction phase. Other than being costly in terms of time (having to calculate distances every time a new input is presented), it's also costly in terms of space, since all the training data must be ready to be used for the prediction.

3. **Little space for interpretation**: it's heavily dependent on the metric used.

4. **Curse of dimensionality**: as the dimensionality of the data increases, the sparser it becomes, and in turn, the harder it becomes to find the neighbors of a point; and even if found, they would be so far away that the resulting estimate would be no longer local. This would push towards lowering the value of $k$ (so that the estimate depends on closer points), but at the risk of running into overfitting.

   Another relevant issue is that the sampling density ($\frac{l}{volume}$) is proportional to $l^{\frac{1}{n}}$; so, as the dimension grows, the number of points needed to estimate the same function exponentially increases as well.

   Finally, if the target depends only on few of the many features, the output could be influenced by the irrelevant features. This can be solved by either weighting features depending on their relevance, or do feature selection.

## 2.3 Linear Model vs. K-NN Model

The two models can be considered two extremes:

| Linear | K-NN |
|---|---|
| Rigid | Flexible |
| Eager | Lazy |
| Parametric | Instance-based |

In conclusion, we can say that low variance/ rigidity is poor, but too high variance/ flexibility is dangerous.

# Chapter 3

# Neural Networks

(Artificial) neural networks are a flexible machine learning tool capable of approximating functions with special properties. They can learn from examples, are universal approximators (Cybenko's/Universal approximation Theorem), can deal with noisy and/or incomplete data, and can handle both continuous real and discrete data. Neural networks encompass a wide set of models.

## 3.1 Artificial Neurons

Artificial neural networks are made up of several **nodes** (also called artificial neurons or units) connected in a net capable of solving artificial intelligence problems. They are heavily inspired by biological neural networks, down to how the single units work and communicate with each other to learn a certain function.

The main ideas that these models are based on are:

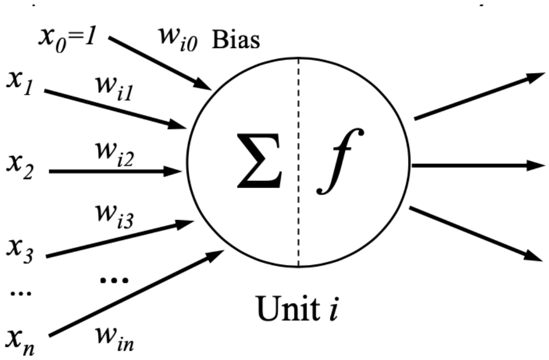
- **Strength reinforcement**: stimuli "reinforce" the weights;
- **Plasticity**: the nervous system is highly capable of adapting.

Each neuron $i$ has a number of **inputs** coming from external sources or other units, and corresponding **weights**, the free parameters that can be modified during the learning phase. Each neuron calculates its **net input** (weighted sum of inputs) and its output as follows:

$$\begin{cases} net_i(x) = \sum_j w_{ij} x_j \\ o_i(x) = f(net_i(x)), \end{cases}$$

where $f$ is the unit's **activator function**.

Note that $w_{ij}$ is the weight of the input coming from the node $j$ and going into the node $i$; some books/libraries/simulators, etc. may use the opposite notation.



Some examples of activator functions include:

- the **linear activation function**;
- the **perceptron/threshold activation function**;
- the **sigmoid/logistic function**.

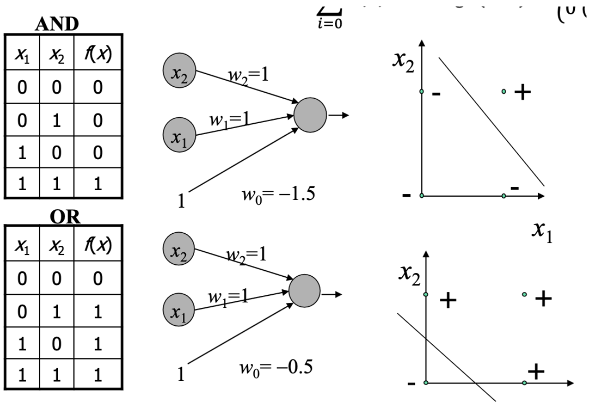
### 3.1.1 Perceptrons

The perceptron was proposed and implemented in 1958 by Frank Rosenblatt. The first perceptron was an actual physical machine designed for image recognition, and not a program. It had a few hundreds of photocells, randomly connected to a layer of neurons.

Multiple perceptrons can be composed and connected to build a network. This is called a **multi-layer perceptron neural network (MLP NN)**.

McCulloch and Pitts proposed a neural network model in 1943. In this model, each neuron is in one of two possible states: firing (1), or not firing (0). All synapses (connections) are equivalent and characterized by a weight $w_i$ which is positive for "excitatory" connections, and negative for "inhibitory" connections. A neuron $i$ becomes active when the sum of the connections coming from other active neurons and the bias is larger than 0. Both inputs and outputs are binary, so it can implement binary classification tasks.

The following pictures illustrate how this model can be used to represent boolean functions AND and OR by using a single neuron. Each neuron has two inputs, plus a bias ($w_0$). Both problems are linearly separable, so if the inputs are represented graphically on a plane, we can draw a single line (described by the net function) that separates all inputs that produce a 0 from all the ones that produce a 1.



AND

| $x_1$ | $x_2$ | $f(x)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

OR

| $x_1$ | $x_2$ | $f(x)$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

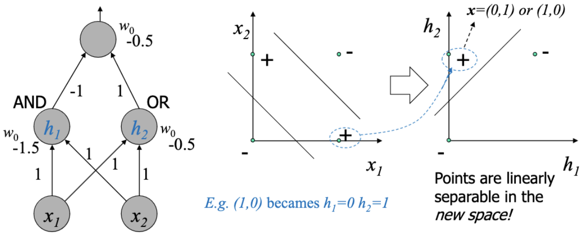
If we instead consider the XOR operator, it corresponds to a non-linearly separable problem, so we can't use a single neuron. The solution is to use a two layer network. The operation can be rewritten as follows:

$$x_1 \oplus x_2 = x_1 \cdot \bar{x_2} + \bar{x_1} \cdot x_2$$

then we have:

$$x_1 \oplus x_2 = \bar{h_1} \cdot h_2$$

So the XOR operation is moved to a new space that represents the problem as linearly separable:



This type of composition can be used to perform more complex tasks, extending the network to many layers of abstraction. In NN, this internal representation can be learned.

## 3.2 Learning Algorithms For One Unit Models

There's two kinds of learning algorithms:

- **Adaline (Adaptive Linear Neuron)**: linear unit during training, can use LMS with either SVD or gradient descent algorithm. This approach an be generalized to multi-level perceptron NNs.

- **Perceptron**: non-linear unit during training, with hard limit or threshold activation function. Can only be used for classification.

### 3.2.1 Perceptron Learning Algorithm

The goal of the algorithm is to minimize the number of misclassified patterns; so it must find $w$ such that $sign(w^T x) = d$. This is an on-line algorithm, so one step can be done for each input pattern. The algorithm can be summarized as follows:

1. Initialize the weights (either to 0 or a small random value);

2. Pick a learning rate $\eta$;

3. For each training pattern $< x, d >$, where $d$ is either $+1$ or $-1$, compute $out = sign(w^T x)$; if $out = d$, don't change the weights, otherwise modify the weights as:

$$w_{new} = w + \eta d x$$

or, in a different form,

$$w_{new} = w + \frac{1}{2}\eta(d - out)x$$

Looking at the problem from a geometrical point of view, it's as if we modified the weight vector $w$ by summing the vector $\eta d x$, where $d$ indicates the direction of the vector with respect to that $x$: if positive, the addition will move $w$ "towards" the point, if negative, it will move it "away" from it.

The form $w_{new} = w + \eta d x$ is in the form of Hebbian learning, while $w_{new} = w + \eta(d - out)x = w + \eta\delta x$ is in the form of error-correcting learning.

### 3.2.2 Perceptron Convergence Theorem

"The perceptron is guaranteed to converge (= classify correctly all input patterns) in a finite number of steps if the problem is linearly separable." Note that a regressor using LMS does not guarantee convergence.

Let us assume $(x_i, d_i)$ in the TR set, where $d_i \in \{-1, +1\}$. If the problem is linearly separable, here exists $w^*$, solution, such that

$$d_i(w^{*T} x_i) \geq \alpha\,, \quad \alpha = \min_i d_i(w^{*T} x_i),\ \alpha > 0$$

hence

$$w^{*T}(d_i x_i) \geq \alpha$$

Let us define $x'_i = (d_i x_i)$; then $w^*$ is a solution iff $w^*$ is a solution of $(x'_i, +1)$ (it classifies all as positive). Assuming $w(0) = 0$ at step 0, $\eta = 1$, and $\beta = \max_i \|x_i\|^2$, after $q$ errors (all false negative):

$$w(q) = \sum_{j=1}^{q} x_{(i_j)} \,,$$

because $w(q) = w(q-1) + x_{i_q}$ according to the update rule used, where $x_{i_j}$ contains the pattern misclassified at the step $j$. Then, we can define the upper and lower bounds for $\|w(q)\|^2$:

- the lower bound is given by:

$$\begin{aligned} &w(q) = x_{i_0} + x_{i_1} + \cdots + x_{i_q} \\ &w^{*T} w(q) = w^{*T} x_{i_0} + w^{*T} x_{i_1} + \cdots + w^{*T} x_{i_q} \geq q\alpha \,, \end{aligned}$$

  since $w^*$ is the optimal solution. With Cauchy-Swartz, it holds that:

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

  so

$$\|w(q)\|^2 \geq (q\alpha)^2 / \|w^*\|^2$$

- the upper bound is given by:

$$\|w(q)\|^2 = \|w(q-1) + x_{i_q}\|^2 = \|w(q-1)\|^2 + 2w(q-1)x_{i_q} + \|x_{i_q}\|^2$$

  where $w(q-1)x_{i_q} < 0$ since it produces a misclassification; therefore

$$\|w(q)\|^2 \leq \|w(q-1)\|^2 + \|x_{i_q}\|^2$$

  and, by iteration, since $w(0) = 0$:

$$\|w(q)\|^2 \leq \sum_{j=1}^{q} \|x_{i_j}\|^2 \leq q\beta$$

Putting it all together:

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

Therefore:

$$q \leq \frac{\beta}{\alpha^2 / \|w^*\|^2}.$$

### 3.2.3 Differences Between LMS and Perceptron Learning Algorithm

They are apparently very similar; they both calculate the new value of $w$ by adding a $\delta$ multiplied by the learning rate $\eta$, along with the input $x$ for the Perceptron Learning Algorithm. The $\delta$ for LMS is calculated as $(d - w^T x)$, while for perceptron learning algorithm it's $(d - sign(w^T x))$. However similar they are, there a few important differences:

- LMS does not necessarily minimize the number of training examples misclassified by the LTU, since it changes the weights for both misclassified and correctly classified ones;
- The perceptron learning algorithm always converges for a linear separable problem to a perfect classifier, while LMS has asymptotic convergence (also for non linearly separable problems);
- The perceptron learning algorithm is difficult to extend to a NN, while the LMS can be extended to a NN by using the gradient based approach.

## 3.3 Activation Functions

As seen before, the possible activation functions can be a linear function, a threshold function (perceptron/LTU), or a non-linear function such as the **sigmoidal logistic function**. The latter is a function that assumes a continuous range of values in the bounded interval $[0, 1]$. It has the important property of being a smoothed differentiable function. The slope of the sigmoid function is defined by the parameter $a$. Some common examples of functions are:

- **Logistic function**: $f_\sigma(x) = \dfrac{1}{1 + e^{-ax}}$ (with output in the range $[0, 1]$)
- **tanh function**: $f_{tanh}(\frac{x}{2}) = 2f_\sigma(x) - 1$ (with output in the range $[-1, +1]$)
- **Radial basis function**: $f(x) = e^{-ax^2}$
- **Rectified linear unit (ReLU)**: $f(x) = \begin{cases} 0 & x < 0 \\ x & x \geq 0 \end{cases}$
- **Softplus function**: $f(x) = ln(1 + e^x)$

The derivative of the identity function is 1. The derivative of the threshold function is not defined, which is why it's not used in LMS. As for sigmoid functions:

$$\frac{df_\sigma(x)}{dx} = f_\sigma(x)(1 - f_\sigma(x))$$

$$\frac{df_{tanh}(x)}{dx} = 1 - f_{tanh}(x)^2$$

## 3.4 LMS With Sigmoidal Function

Since the sigmoidal logistic function is differentiable, we can derive a LMS algorithm by computing the gradient of the mean square loss function as for the linear units. The output of a neuron is calculated as:

$$out(x) = f_\sigma(x^T w)$$

The objective of the algorithm is to find $w = argmin_w E(w) = \sum_p (d_p - out(x_p))^2 = \sum_p (d_p - f_\sigma(x_p^T w))^2$, so the weights that minimize the residual sum of squares.

The gradient descent algorithm is the same, except it uses the new delta rule: $w_{new} = w + \eta \delta_p x_p$, where $\delta_p = (d_p - out(x_p)) f'_\sigma$. Additionally, the parameter $a$, the slope of the function $f_\sigma$, can affect the step of the gradient descent.

Overall: the max of $f'$ corresponds to net inputs close to 0, while the minimum of $f'$ corresponds to **saturated cases**, as in where the function $f$ goes to either 0 or 1 asymptotically.

## 3.5 Multi Layer Perceptrons

A MLP can be seen in two possible ways: either as a network of units, or as a flexible function. As a network, a MLP contains a number of units connected by **weighted links**. The units are organized in layers: the first layer that loads the input is called the **input layer**; the layer that produces the final output is called the **output layer**; all other inbetween layers are called **hidden layers**. The notation used to refer to networks will be the following:

- the index $t$ denotes a generic unit, while $k$ denotes an output unit;
- the index $u$ denotes a generic input component;
- $x$ is a generic input from an external source (if it's an input vector) or from another unit;

- if the pattern $x$ is loaded in the input layer, the notation $o$ can be used for both inputs and hidden layer inputs, so, inside the network, the input to each unit $t$ from any source $u$ is simply denoted as $o_u$.

As a flexible function, the MLP can be written as a function in the following form:

$$h(x) = f_k(\sum_j w_{kj} f_j(\sum_i w_{ji} f_i(\ldots))) \, ,$$

where $f_k$ is a sigmoid activation function.

### 3.5.1 Architecture

The architecture of a NN defines the topology of the connections between units. A **feedforward architecture** describes a network that operates as follows. For each input pattern $x$, do:

1. load the input in the input layer;
2. compute the output of all the units in the first hidden layer;
3. compute the output of all the units in the second hidden layer;
4. ...
5. compute the output of all the units in the output layer;
6. compute the error (delta) at the output level.

A **recurrent architecture** describes a network with feedback loops, which allow the output of units to go "back" in the network towards units in previous layers.

## 3.6 Flexibility of NNs

The hypothesis space of a NN is the 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 as output, the model can deal with both regression and classification tasks, by using, respectively, a linear function or a sigmoidal one. It can also implement multi-regression and multi-class classifiers by defining multiple output units.

When we consider a NN as a function, so in the form

$$h(x) = f_k(\sum_j w_{kj} f_j(\sum_i w_{ji} f_i(\dots))) \, ,$$

each $f_j$ can be seen as computed by an independent unit, or a special kind of $\phi$ of Linear Basis Expansion. Additionally, $h(x)$ is non-linear in the parameters $w$. So, we can reformulate the function as follows:

$$h(x) = f_k(\sum_j w_{kj} f_j(\sum_i w_{ji} x_i)) = f(\sum_j w_j \phi_j(x, w))$$

The main difference with LBE is that the $\phi$ are adapted to data by fitting the $w$.

### 3.6.1 Theoretical Grounds

The universal approximation theorem declares that single hidden-layer network (with logistic activation functions) can approximate (arbitrarily well) every continuous function, given enough units; a MLP network can approximate (arbitrarily well) every input-output mapping (provided enough units in the hidden layers), as well. Note that this theorem does not say which learning algorithm to use, nor the number of units needed.

Two issues arise: how can we learn with a NN?, and, how can we decide what NN architecture to use? The answers to these two questions will be provided later. The expressive power of NN is strongly influenced by two aspects: the number of units and the architecture. More specifically, the number of parameters $w$ (which is proportional to the number of units) influences the network capabilities; additionally, even their value can influence the VC-dim of the network: the lower the weights, the lower the VC-dim.

Another important aspect is the number of layers to use in the network. The universal approximation theorem tells us that 1 layer is sufficient to approximate any function, yet it gives no indication on the number of units to use, which may be incredibly high. So, instead of using only one layer, the network is split into multiple layers, each with a limited number of units.

## 3.7 Backpropagation Learning Algorithm

The learning algorithm used for NN follows the LMS approach. The goal is to adapt the free parameters $w$ in order to obtain the best approximation of the target function. This is normally done by checking the value of the error (or loss) function calculated on the training set. In the case of NNs, however, this is not as straightforward: the network has multiple units across layers, so the learning algorithm must first decide how much credit to give to the hidden units in causing the error to increase.

The so called **loading problem** is formulated as follows: given a network and a set of examples, is there a set of weights so that the network will be consistent with the examples? The problem is NP-complete, and while networks can in practice be trained in a reasonable amount of time, an optimal solution in not guaranteed.

The idea behind the backpropagation algorithm solution is to extend the gradient descent approach, so it can be used with MLP networks as well. The things we need are:

- differentiable loss;
- differentiable activation functions;
- a network to follow the information flow.

What we want to find is $w$ obtained by computing the gradient of the error function. The advantages of the backpropagation algorithm are:

- it's easy because of the compositional form of the model;
- it keeps track of the quantities local to each unit;
- it's efficient, since it's $O(\#W)$ rather than $O(\#W^2)$
- supposedly, the brain's learning "algorithm" is a local sub optimal approximation of backpropagation (although this debate remains controversial).

### 3.7.1 Issues in Training NNs

The resulting model is often over-parameterized; additionally, the optimization problem is no longer convex, and is potentially unstable.

Some problems in training NNs involve:

- **Hyperparameters**: starting values, choosing between on-line/batch, learning rate, number of hidden units in the network;
- **Multiple minima**: the loss is no longer a simple convex function, so finding a minimum needs specific techniques;
- **Stopping criteria**;
- **Overfitting and regularization**;
- **Input scaling/output representation**.

### Hyperparameters

**Starting values** The weights are normally initialized with random values near 0; we want to avoid a completely null weight vector, too high values, or all components equal to each other, since these can hamper training. We can also consider the **fan-in**, i.e., the number of inputs to a hidden unit: $range * 2/fan - in$. This should be avoided if the fan-in is too large, or if the unit in question is an output unit, since the $\delta$ would be too close to 0 (since we're using backpropagation, those $\delta$ values would be propagated to previous layers, and cause all others to decrease as well).

There's other heuristics, such as a relatively recent one proposed by Glorot and Bengio.

**On-line or Batch gradient descent** The batch version of the algorithm calculates all the gradients of each pattern over an epoch, then it updates the weights. The stochastic version instead updates the weights after calculating the gradient on one pattern at a time; it also needs a smaller learning rate $\eta$ compared to the batch version, since a value too high may produce a training that's too "chaotic".

In terms of MLP with backpropagation, the way the gradient is calculated is:

$$-\frac{\partial E(w)}{\partial w_{tu}} = -\sum_{p=1}^{l} \frac{\partial E_p(w)}{\partial w_{tu}} = \sum_{p=1}^{l} \delta_{p,t} o_{p,u}$$

for the batch version, and:

$$-\frac{\partial E(w)}{\partial w_{tu}} = \delta_{p,t} o_{p,u}$$

for the on-line version.

The batch version produces a more accurate estimation of the gradient, but the on-line version can help avoid local minima while not being as accurate. Additionally, when using the on-line version, a random shuffling of the patterns should be done before each epoch, in order to avoid any bias in the gradient descending.

A variant of the on-line version is the **minibatch (MB)**. In this version, the epochs are divided in parts; the gradient is summed up to $mb$ patterns ($mb =$ the size of a part) before updating the weights, instead of updating them after every single calculation. Indeed, the batch an on-line versions can be seen as extreme cases of the minibatch: the first uses a part that's exactly the same size as the dataset, the other uses parts of size $= 1$. A commonly used value of $mb$ is 100, since it's the ideal partitioning of the dataset that exploits GPU memory parallelism the best.

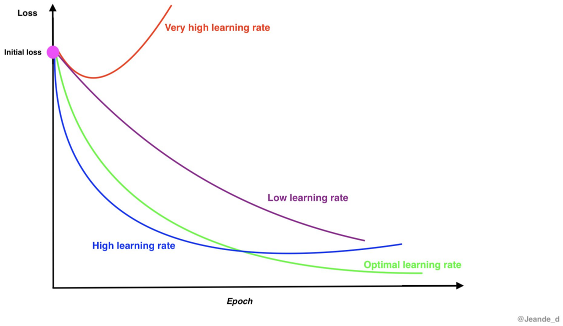
Basic SGD is generally efficient even with NNs, but depends on the number of training data and the number of epochs. There's other heuristic approaches, such as

Quick-prop, in which we go to the minima of the local estimated convex function, or R-prop, which does not use the value of the gradient and instead only its' sign.

Some commonly used approaches are **second order methods**, which calculate more information about the curvature of the objective function in order to plan a better descent. Some methods include approximation of Newton's methods, and Hessian free approaches (conjugate gradients and quasi-Newton methods). These methods, however, have never replaced gradient descent for NN training, because they are susceptible to saddle points (which grow exponentially in high dimensional spaces); gradient descent, instead, shows how to escape saddle points. The length of the training path with gradient descent may still be high for reasons unrelated to local minima or saddle points, however; the reason is still subject of studies.

In practice, the main basic approaches are more that enough if applied correctly. After all, this optimization is done on the training set, but our focus should be on validation.

**Learning rate** An higher value of $\eta$ causes the descent to be faster but more unstable; a lower value of $\eta$ causes the descent to be slower but more stable. A way to monitor the behavior of the model (and adjust the learning rate as desired) is to plot the **learning curve**, which is obtained by plotting the error against the number of epochs. Depending on the value of $\eta$, the curve may have a different appearance.



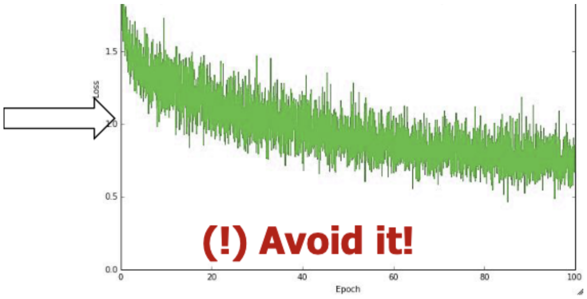
@Jeande_d

The green line may appear smooth, but it's actually irregular if we zoom in:

In a practical approach, it's useful to consider the mean of the gradients over the epoch, in order to have a uniform approach with respect to the number of input data. When using LMS, dividing by $l$ is equivalent to using $\eta/l$ as our learning rate. As a way to improve the choice of learning rate, the following approaches may be considered:

- Using **momentum**: by adding momentum, $\Delta w$ is now calculated as:

$$\Delta w = -\eta \frac{\partial E(w)}{\partial w} + \alpha \Delta w_{old}\,,$$



so we introduce a new term that depends on the previous value of $\Delta w$. $\alpha$ is a value between 0 and 1.

This is called the "heavy ball method". It's faster in plateaus, because the same sign of gradient causes the momentum to increase the delta, and also compensates oscillations in the training, allowing us to use a higher value of $\eta$. It's commonly assumed to help with batch mode more, but can also be used with on-line; in this case, $\Delta w_{old}$ is the $\Delta w_{p-1}$ (as in, of the previous example).

A variant of momentum is the **Nesterov momentum**, where the gradient is evaluated after the momentum is applied; so we firs calculate $w = w + \alpha \Delta w_{old}$, the we evaluate the gradient on this new $w$. This variant has been shown to improve the rate of convergence for the batch mode, but not for the stochastic mode.

- Variable learning rate (start high, then decrease): using minibatch, the gradient does not reach 0 even when close to a minimum (as exact gradient can do), hence a fixed learning rate should be avoided. We can linearly decay $\eta$ for each step until iteration $\tau$, using $\alpha = step/\tau$:

$$\eta_s = (1 - \alpha)\eta_0 + \alpha\eta_\tau$$

  and then stop for some iteration, from which we can use a fixed small value of $\eta$. Ideally, the final value of $\eta$ should be $\sim 1\%$ of $\eta_0$, so it takes a few hundred steps to reach it.

- Adaptive learning rates (changed during training and for each $w$): automatically adapt the learning rate during training, avoiding or reducing the fine tuning phase via hyperparameter selection. Some popular ones include AdaGrad, RMSProp, Adam. Can be combined with momentum.

- Varying in NNs with many layers (higher for deep layers, or higher for units with few input connections).

**Number of units** The number of units controls the complexity of the NN. This choice is in general a model selection issue, so it's selected by a cross-validation. Few units typically lead to underfitting, while too many lead to overfitting, but we can have NNs with many units that don't overfit if regularization is used. We can follow two approaches; either a **constructive**, incremental, one, in which the learning algorithm decides the (small) starting number of units and then adds more as needed, or a **pruning** one, where we start with a large network and progressively eliminate weights or units.

A common constructive approach is the **Cascade Correlation (CC)** learning algorithm. It starts with a minimal network, and adds units until the error is low. It learns both the network weights and topology.

**Algorithm 1** Cascade Correlation algorithm.

```
Start with N0, a network with no hidden units, train it and calculate the error.
if N0 does not solve the problem then
    Set i = 1.
    repeat
        Create a new unit, and add to the network (creating Ni).
        Train the unit so that the correlation between the output of the unit and residual error of network Ni-1 is maximized.
        Freeze the weights of the unit, and retrain all other units.
        Calculate error of the new network.
        Increase i by 1.
    until Error is $\leq$ a given threshold.
end if
```

Specifically, this algorithm works by interleaving the minimization of the error function with the maximization of the correlation, which is the covariance of the output of the new unit with the residual error. The role of hidden units is to reduce the residual output error.

### Multiple Minima

Loss in no longer convex. The function may have multiple minima, as well as maxima. The final result depends a lot on the starting weight values. Ideally, we should try a number of random starting configurations (trials), and then take the mean result (as in, the mean of errors) and check the variance in order to evaluate the model. Then, we can either pick the solution that produces the lowest/median validation error, or we can consider the mean of the outputs.

It's worth noting that finding a local minima with too high error that stops training is not a big issue, since we can always check the final training error (and restart if needed). In general, we don't need to find the global minimum, a "good" local minima is sufficient. This is because the minimum we find is of $R_{emp}$, not $R$ (which is what we want to approximate). Also, instead of finding a point corresponding to the null gradient, we may want to stop at a point that has sufficiently small gradient.

Another aspects to consider is that the NN builds a variable size hypothesis space, and tends to increase the VC-dim during training. As the VC-dim increases, the $R_{emp}$ decreases towards the global minimum, but we'll likely incur into overfitting, so stopping before we find the minimum might actually produce a better approximation of the target function.

### Stopping Criteria

The basic stopping criteria is to check the error (mean or max error $E$), however, we may not always have enough information to set a tolerance threshold for the error. We may want to use an internal criterion: we stop if the improvement of the error (e.g., less than 0.1%), or the changes to the weights (e.g. norm of gradient $< \eta$) are negligible.

We must not stop at an arbitrary number of steps; if it's too small, then it may be too early and cause underfitting, while if it is too large, we may incur into overfitting. Also, if we use K-Fold cross validation, the number of ideal steps for each fold may vary: how do we choose it for the final model, trained over all data? We can consider the number of epochs as a hyperparameter, and select its value as the mean across the folds; however, changing the data size at the end, adding all the records of validation and test set, the stop point may be different, leading to underfitting.

### Overfitting and Regularization

As said before, we typically don't want the global minimizer of $R_{emp}$, since it would be an overfitting solution. The control of complexity requires some form of regularization (as seen for LBE). This can be achieved by introducing a penalty term, or indirectly by early stopping. Another important step is to perform cross-validation on empirical data to find the best trade-off.

In NNs, learning normally starts by setting the weights to small, random values, and the VC-dim is low. As optimization proceeds, hidden units tend to saturate, increasing the number of free parameters, hence increasing the VC-dim. So, how do we choose when to stop this optimization?

- **Early stopping**: using a validation set to determine when to stop. We ideally want to consider multiple epochs to estimate the error. Since the effective number of parameters grows during the course of training, stopping means limiting the complexity.

- **Regularization**: we can use a regularization related to Tikhonov theory, applied to the loss; a penalty term is added, such that the loss will be calculated as:

$$Loss(w) = \sum_p (d_p - o(x_p))^2 + \lambda \|w\|^2$$

  This is a form of weight decay, since the new values of the weights during gradient descent will be calculated by adding the term $-\lambda w$, which causes it to decrease even when $\Delta w$ is 0. The regularization parameter $\lambda$ is generally a low value, and is selected in the model selection phase.

- **Pruning methods**: will see later.

Remember that, when using regularization, the loss is used during model training, while the error (or risk) for the “data term” is used during model evaluation, since it only measures how different the output of the hypothesis is from the correct label. A common misconception is that regularization helps convergence stability, but it does not. It only controls the complexity.

Also, early stopping and regularization can be used together. The difference is that early stopping is an **empirical approach**, that requires a VL set to decide a stopping point, while regularization is a **principled approach**, and allows the VL curve to follow the TR curve.

Note, also, that the bias $w_0$ is often omitted from the regularizer, since its inclusion causes the results to not be independent from target shift or scaling. If it’s included, it has its own regularization term.

### Putting Momentum and Weight Decay Together

By putting the two together, the $\Delta w$ can be calculated as:

$$\Delta w = -\eta \frac{\partial Loss(w)}{\partial w_{tu}} + \alpha \Delta w_{old_{tu}} \doteq \eta \delta_t o_u - \lambda w_{tu} + \alpha \Delta w_{old_{tu}}$$

Here, the index $p$ is omitted and we consider the Loss as the sum of Error + Penalty and using $\eta$ only for the Error term. This form, where $\Delta w$ includes $\lambda$, $\eta$ and $\alpha$, is often used by major tools/libraries for NNs. However, we can define a form that keeps the hyperparameters separated, b y writing:

$$\Delta w = \eta \delta_t o_u + \alpha \Delta w_{old_{tu}} \ ,$$

$$w_{new_{tu}} = w_{tu} + \Delta w_{tu} - \lambda w_{tu}$$

In this form, eta and alpha are independent.

### Input Scaling/Output Representation

Preprocessing of the data can have a large effect on the result of training. Data should be normalized via either standardization (each feature is modified so that mean is 0 and standard deviation is 1), or rescaling (the range of values is restricted to [0,1]).

For regression, there's one or more output linear units; for classification, there's either a singular binary output unit, or there's multiple binary units. In the latter case, we can use a sigmoid to choose the threshold to assign the class, a rejection zone, or 1-to-K encoding to choose the "winner" class. Often, the symmetric logistic function learns faster. To avoid asymptotic convergence, 0.9 and 0.1 can be used instead of 1 an 0 as a label value (for the logistic function, for the tanh function -0.9 is used instead of -1). In any case, the target range must be in the output range of units. If targets are 0/1, it's common to use the Softmax function:

$$o_k(x) = \frac{e^{(-net_k)}}{\sum_{j=1}^{K} e^{(-net_j)}}$$

We can also use an alternative error/cost function, where we minimize the cross entropy:

$$E(w) = -\sum_{i \in TR} \{ d_i \log(out(x_i)) + (1 - d_i) \log(1 - out(x_i)) \}$$

## 3.8 When To Consider NNs

- The input is high dimensional discrete or real-valued, both for regression and classification tasks;
- The dataset possibly contains noise;
- The form of the target function is unknown;
- Human readability of the output is not critical;
- Training time is not critical;
- The computation of the output itself has to be fast.

# Chapter 4

# Validation and SLT

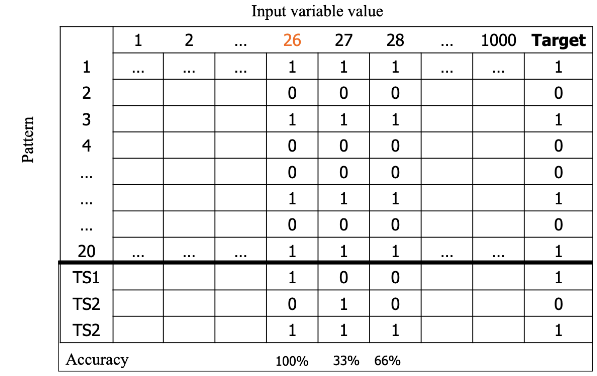
When a model is being trained, we're looking at the best solution, i.e., the best balancing between fitting of the training data and model complexity, as not to incur into over- or underfitting. Obviously, the training set cannot be used to estimate the test error, as low training error leads to overfitting.

Assume we have only one hyperparameter $\theta$ that varies the model complexity, and we want to find the "best" value of it. The estimation of the error can be approximated, either analytically (e.g., AIC, BIC for linear models, Minimum Description Length, Structural Risk Minimization and so on), or by resampling, so via **cross-validation** or bootstrap.
Remember the distinction between model selection and model assessment:

- **Model selection** estimates the performance of different models in order to select the one the generalizes the best; this phase returns a model;
- **Model assessment**: having chosen a final model, it estimates its prediction error/risk by using unseen data (the test set), and returns an estimation value.

The gold rule is to keep separation between the goals, and for each goal to use a different set: one for training, one for validation, and one for testing.

Let's see an example. Assume we have 20-30 examples, with 1000 input variables. The target (random) is 0/1. We select a model with a single input variable that guesses 100% on any splitting of the data in TR/VL/TS. This means that we produced a model that apparently always finds the correct output... or does it? In reality, the true error estimation is 50%; this is because the chosen variable just so happens to match the target values. But in a real application, the model is basically coin guessing the output. In summary, the error estimated on the training set (and validation set) is not a good estimation of the generalization error; we need a new set of unseen data to do so.



Input variable value

| Pattern | 1 | 2 | … | 26 | 27 | 28 | … | 1000 | Target |
|---|---|---|---|---|---|---|---|---|---|
| 1 | … | … | … | 1 | 1 | 1 | … | … | 1 |
| 2 |  |  |  | 0 | 0 | 0 |  |  | 0 |
| 3 |  |  |  | 1 | 1 | 1 |  |  | 1 |
| 4 |  |  |  | 0 | 0 | 0 |  |  | 0 |
| … |  |  |  | 0 | 0 | 0 |  |  | 0 |
| … |  |  |  | 1 | 1 | 1 |  |  | 1 |
| … |  |  |  | 0 | 0 | 0 |  |  | 0 |
| 20 | … | … | … | 1 | 1 | 1 | … | … | 1 |
| TS1 |  |  |  | 1 | 0 | 0 |  |  | 1 |
| TS2 |  |  |  | 0 | 1 | 0 |  |  | 0 |
| TS2 |  |  |  | 1 | 1 | 1 |  |  | 1 |
| Accuracy |  |  |  | 100% | 33% | 66% |  |  |  |

## 4.1 Grid Search

Since hyperparameters are not learned by the learning algorithm, it's our task to find the best values for them. Choosing them is done by calculating the error on the validation set (VL). One way to search for hyperparameter values is to perform a grid search; candidates are generated from a grid of parameter values and tested on the validation set. This technique can have an high cost, since the total number of candidates is equal to $n^m$, where $n$ is the number of values in the range and $m$ is the number of hyperparameters. For this reason, grid search is typically used to fix some values in preliminary phases.

Additionally, we can perform two or more levels of nested grid search. First we apply a coarse search to a table with the combinations of all possible values set at intervals (e.g., growing exponentially) in order to find the best interval. Then we apply a second grid search by considering only the values withing that interval, and we can repeat the grid search until we perform a search fine enough for our needs.

There are alternatives to grid search, such as random search, automatized search, and many more offered by popular libraries.

## 4.2 K-fold Cross Validation

The most basic way of setting up data for model selection and model assessment is to split all of our data into three partitions (hold out cross validation). One of them will be the training set, one the validation set, and one the test set. However, it can be difficult to decide how to perform this split; it can depend on the complexity of the model, and signal-to-noise ratio. Also, how can we split the data so that each part is sufficiently sized? And can we avoid to perform a split that influences the results?

K-fold cross validation is another technique that can be used to split the data. Typically, the data is split into TR set and TS set. Then, k-fold cross validation is used over the TR set to obtain new partitions of the data to form different TR sets and VL sets. After finding the best hyperparameters, the model is retrained on the whole original TR set, and is assessed on the external TS set. Again, if the estimation is good enough, the model can be retrained on the test set data as well to produce the final model.

K-fold cross validation can still present some problems. It may be influenced by biases in the sample of data used. Cross validation can be repeated, by performing different splits with different random sampling; the estimation can be obtained by averaging the results. Another issue is having a sample too small to accurately represent the general population.

## 4.3 Statistical Learning Theory

### 4.3.1 Vapnik-Cervonenkis Dimension

The VC-dim (Vapnik-Cervonenkis dimension) is a measure of the complexity of a class of hypotheses $H$. It does not depend on the actual cardinality of the space, but on the number of distinct instances that can be completely discriminated (i.e., without error) using $H$.

Given $X$ an input space containing $N$ examples, and $H$ the hypothesis space, assume a binary classification task; there are $2^N$ possible **dichotomies** (i.e., partitions of the $N$ points either -1 or 1). A particular dichotomy is **represented** in $H$ if there exists an hypothesis $h$ in $H$ that realizes the dichotomy (it isolates all the points belonging to one class from the others).

**Shattering**

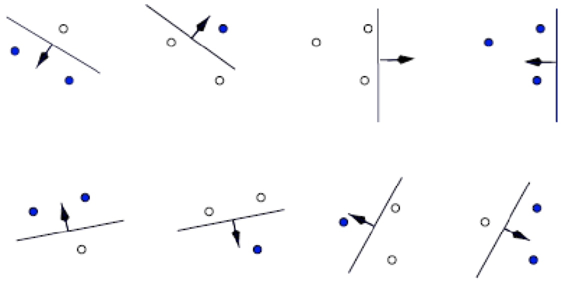
The hypothesis space $H$ **shatters** $X$ if and only if $H$ can represent all the possible dichotomies on $X$.

**VC-dimension**

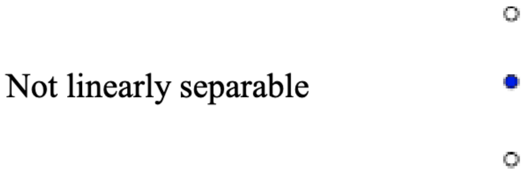
The VC-dimension of a class of functions $H$ is the maximum cardinality of a configuration of points in $X$ that can be shattered by $H$.

If $VC(H) = p$, it means that there's at least an hypothesis in $H$ that shatters a configuration of size $p$, but cannot shatter any configuration of size $p + 1$. If arbitrarily large (but finite) sets of $X$ can be shattered by $H$, then $VC$ is infinite. Note that there needs to be just one configuration of $p$ points that is shattered by an $h$; $h$ does not need to shatter all configurations of size $p$.

To make the concept clearer, consider an hypothesis space $H$ that is a set of lines $(h(x) = sign(wx + w_0)$, $x \in IR^2)$. Each dichotomy is a different labeling of points as -1 or 1. This hypothesis space shatters some configurations of 3 points, such as the one shown in the figure below:



but cannot shatter this one:



Still, its VC-dim is at least 3, since it can shatter at least the configuration on top. If we consider any configuration of 4 points, a single line cannot always separate them. This means that for any configuration of 4 points, there's at least a dichotomy that is not linearly separable, therefore the VC-dim is less than 4. In general, the VC-dim of a class of linear hyperplanes (LTU) in an $n$-dimensional space is $n + 1$.

VC-dim is not the number of free parameters, although it is related. There exists models with only one free parameter and infinite VC-dim, such as K-NN. However, for many reasonable hypothesis classes the VC-dimension is linear in the number of free parameters of the model. Once we determine the VC-dim, we can then estimate the risk by defining an upper bound:

$$R[h] \leq R_{emp}[h] + \epsilon(VC, N, \delta)$$

The function $\epsilon()$ can be calculated in different ways depending on the task, class of functions, etc. This bound gives us a way to estimate the generalization error based only on training error and VC-dimensionality of $H$.

### 4.3.2 Structural Risk Minimization

SRM uses the VC-dimension as a control parameter to realize the best estimation of the generalization bound. Assume a nested structure of $n$ models/hypotheses space, such that:

$$\begin{gathered} H_1 \subseteq H_2 \subseteq \cdots \subseteq H_n \\ VC(H_1) \leq VC(H_2) \leq \cdots \leq VC(H_n) \end{gathered}$$

An example could be a nested structure of neural networks, where each $H_i$ corresponds to a network containing a different number of units; another could be polynomials with increasing degree.

The more the VC-dim increases, the more the VC-confidence and therefore the bound increase as well. The goal of SRM is to find the model corresponding to the trade-off on the bound between VC-dim and VC-conf. Once this bound has been determined, can it be used as an estimate of prediction errors? Yes, but rarely. It can be useful during model selection, but it's an overly pessimistic estimate for model assessment.

# Chapter 5

# Support Vector Machines

Support Vector Machines (SVMs) are a type of model used for both classification and regression. They used to be the most popular approach for supervised learning when there's little to no domain knowledge for the data, but they've been mostly replaced by neural networks and random forests.

## 5.1 Hard Margin SVM

Assume a classification task. Given a training set $TR =< x_i, d_i >$, $i = 1 \dots N$, we want to find an hyperplane of equation $w^T x + b = 0$ to separate the examples; specifically, we want:

$$w^T x_i + b \geq 0 \text{ for } d_i = +1$$
$$w^T x_i + b < 0 \text{ for } d_i = -1$$

$g(x) = w^T x + b$ is called the **discriminant function**, and $h(x) = sign(g(x))$ is the hypothesis. Note that here the bias is referred to as $b$ instead of $w_0$. To find the hyperplane that separates all the points, instead of simply minimizing the empirical risk, the SVM minimizes the expected generalization error by finding the separator that is farthest away from all the examples in the dataset. It also establishes a margin ($\rho$) around it, whose width is exactly twice the distance between the hyperplane and the closest data point(s). The optimal hyperplane is the one that maximizes this margin $\rho$: $w_o^T x + b_o = 0$, where $\rho = 2/\|w_o\|$.

The two terms $w$ and $b$ can be rescaled so that the closest points to the separating hyperplane satisfy $\|g(x)\| = 1$, so we can write:

$$w^T x_i + b \geq 1 \text{ for } d_i = +1$$
$$w^T x_i + b \leq -1 \text{ for } d_i = -1 \,,$$

or, in a compact form,

$$d_i(w^T x_i + b) \geq 1 \,.$$

Any $x_i$ that satisfies this equation is called a **support vector**, and is referred to as $x^{(s)}$. Let's denote the distance between the optimal hyperplane and a point $x$ as $r$, such that $x = x_p + r\frac{w_o}{\|w_o\|}$ (where $x_p$ is a point on the hyperplane). Evaluating $g(x)$ we obtain:

$$\begin{aligned} g(x) = g(x_p + r\frac{w_o}{\|w_o\|}) &= \\ = w_o^T x_p + b_o + w_o^T r\frac{w_o}{\|w_o\|} &= \\ = g(x_p) + w_o^T r\frac{w_o}{\|w_o\|} &= \\ = 0 + r\frac{\|w_o\|^2}{\|w_o\|} &= r\|w_o\| \,, \end{aligned}$$

thus, $r = \dfrac{g(x)}{\|w_o\|}$.

Consider the distance between the hyperplane and a positive support vector $x^{(s)}$, then $r$ is calculated as:

$$r = \frac{g(x^{(s)})}{\|w_o\|} = \frac{1}{\|w_o\|} = \frac{\rho}{2} \,,$$

therefore, $\rho = \dfrac{2}{\|w_o\|}$.

### 5.1.1 Primal Problem

The optimum hyperplane will maximize $\rho$ and minimize $\|w\|$. One approach to find it is to perform a gradient descent to find the $w$ and $b$ that minimize/maximizes them, but there's another approach, that is solving a **quadratic optimization problem**.

**Quadratic optimization problem (primal form)**

Given the training examples $TR =< x_i, d_i >$, find the optimal values of $w$ and $b$ which minimize

$$\Psi(w) = \frac{1}{2}w^T w$$

satisfying the constraints

$$d_i(w^T x_i + b) \geq 1\,.$$

The objective function $\Psi(w)$ is quadratic and convex in $w$. The constraints are linear in $w$, and solving the problem scales with the size of the input space $m$.

To solve this problem, the **Lagrangian multipliers method** is used. The Lagrangian function corresponding to the quadratic optimization problem is constructed:

$$J(w, b, \alpha) = \frac{1}{2}w^T w - \sum_{i=1}^{N} \alpha_i(d_i(w^T x_i + b) - 1)\,,$$

where $\alpha_i \geq 0$ are the **Lagrangian multipliers**. Each term in the sum corresponds to a constraint of the primal problem; $J$ must be minimized with respect to $w$ and $b$ and maximized with respect to $\alpha$. The solution will correspond to a saddle point of $J$.

If we minimize $J$ with respect to $w$, then:

$$\frac{\partial J}{\partial w} = \frac{2}{2}w - \sum_{i=1}^{N} \alpha_i(d_i(1 * x_i + 0) + 0) =$$

$$= w - \sum_{i=1}^{N} \alpha_i(d_i(x_i)) = 0$$

so

$$w = \sum_{i=1}^{N} \alpha_i(d_i(x_i))\,.$$

Thus the optimal hyperplane is expressed as:

$$g(x) = w_o^T x + b_o = 0$$

$$\Longleftrightarrow$$

$$\sum_{i=1}^{N} \alpha_{o,i} d_i x_i^T x + b_o = 0\,.$$

If instead we minimize it with respect to $b$:

$$\frac{\partial J}{\partial b} = 0 - \sum_{i=1}^{N} \alpha_i d_i = 0\,.$$

These can be substituted in $J$ to study the dual form of the problem.

From the **Kuhn-Tucker Conditions**, it follows that

$$\alpha_i(d_i(w^T x_i + b) - 1) = 0\,, \forall i = 1, \dots, N$$

in the saddle point of $J$. If $\alpha_i > 0$, then $(d_i(w^T x_i + b) = 1$, and $x_i$ is a support vector. If $x_i$ is not a support vector, then $\alpha_i = 0$. Hence we can restrict the computation to $N_s : w_o = \sum_{i=1}^{N_s} \alpha_{o,i} d_i x_i$. The hyperplane depends only on support vectors.

### 5.1.2 Dual Problem

To obtain the Lagrangian multipliers $\alpha_i$, we solve the problem in its dual form:

**Quadratic optimization problem (dual form)**

Given the training examples $TR =< x_i, d_i >$, find the optimal values of $\alpha_i$ which maximize

$$Q(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j d_i d_j x_i^T x_j$$

satisfying the constraints

$$\alpha_i \geq 0\,, \forall i = 1, \dots, N\,,$$
$$\sum_{i=1}^{N} \alpha_i d_i = 0\,.$$

The value of $\alpha_i$ can be found by solving the quadratic programming (QP) problem, or by more recent and efficient approaches (such as sequential minimal optimization (SMO)). Solving this problem scales with the number of training examples, less with the dimensionality. To find $w_o$ and $b_o$, we proceed as follows:

$$w_o = \sum_{i=1}^{N} \alpha_{o,i} d_i x_i$$
$$b_o = 1 - w_o^T x^{(s)} = 1 - \sum_{i=1}^{N} \alpha_{o,i} d_i x_i^T x^{(s)}\,.$$

How do we use all of this? We don't actually need to explicitly know $w_o$. All we need are calculating the Lagrangian multipliers by solving the dual problem, and then calculating

$b_o$. So, given the input pattern $x$, we compute $g(x) = \sum_{i=1}^{N} \alpha_{o,i} d_i x_i^T x + b_o$, and classify it as $h(x) = sign(g(x))$. Minimizing the norm of $w$ is equivalent to minimizing the VC-dim and VC-confidence of the model.

**Theorem - Vapnik**

Let $D$ be the diameter of the smallest ball around the data points $x_1, \ldots, x_N$. For the class of separating hyperplanes described by the equation $w^T x + b = 0$, the upper bound to the VC-dim is

$$VC \leq min(\lceil \frac{D^2}{\rho^2} \rceil, m_0) + 1\,,$$

where $m_0$ indicates the dimensionality of the input space.

Since $D^2/\rho^2 = Radios^2 \|w\|^2$, VC can be less than $m_0 + 1$ computed for general hyperplanes by constraining the search to the "regularized" one with maximum margin.

This approach of finding optimal separating hyperplane maximizing the margin also provides:

- An unique solution with zero errors for the binary classifier;
- An automatized approach to Structural Risk Minimization that minimizes the VC-confidence without having to deal with hyperparameters;
- The use of a solver in the class of constrained quadratic programming (instead of gradient descent) with a nice dual form;
- A solution focused on a selection of training data points: the support vectors.

But what about noisy or non linearly separable data?

## 5.2 Soft Margin SVM

In realistic datasets, the training set is not going to be perfect. It will contain noisy points and outliers that make the problem not linearly separable. The solution is to find a separator with a soft margin, that is, a margin that allows some errors within it. Because the margin can allow points to fall inside of it, the support vectors are no longer going to be the closest points to the margin.

We introduce what are called **slack variables**, which are non negative scalar variables:

$$\xi_i \geq 0 \,, \forall i = 1, \dots, N$$
$$d_i(w^T x_i + b) \geq 1 - \xi_i \,, \forall i = 1, \dots, N$$

If $x^{(s)}$ is a support vector, it will satisfy the equation:

$$d_i(w^T x^{(s)} + b) = 1 - \xi_i$$

Here, Vapnik's theorem does not hold anymore. The problem can be then rewritten to admit points in the margin.

### Quadratic optimization problem with Soft Margin (primal form)

Given the training examples $TR =< x_i, d_i >$, find the optimal values of $w$ and $b$ which minimize

$$\Psi(w, \xi) = \frac{1}{2} w^T w + C \sum_{i=1}^{N} \xi_i$$

satisfying the constraints

$$d_i(w^T x_i + b) \geq 1 - \xi_i \,, \forall i = 1, \dots N$$
$$\xi_i \geq 0 \,, \forall i = 1, \dots N$$

The term $C$ is a user-defined regularization hyperparameter; so this version of SVM is no longer "automatic" like hard margin SVM was. $C$ is found as the trade-off between empirical risk minimization and capacity term (VC-confidence). If $C$ is too low, we allow many training errors, leading to underfitting. If $C$ is too high, we don't let any training error, leading to overfitting.

**Quadratic optimization problem with Soft Margin (dual form)**

Given the training examples $TR =< x_i, d_i >$, find the optimal values of $\alpha_i$ which maximize

$$Q(\alpha) = \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j d_i d_j x_i^T x_j$$

satisfying the constraints

$$0 \leq \alpha_i \leq C \, , \forall i = 1, \ldots, N$$
$$\sum_{i=1}^{N} \alpha_i d_i = 0$$

The Kuhn-Tucker conditions can be redefined as:

$$\alpha_i (d_i (w^T x_i + b) + \xi_i - 1) = 0 \, , \forall i = 1, \ldots, N$$
$$\mu_i \xi_i = 0 \, , \forall i = 1, \ldots, N \, ,$$

where $\mu_i$ are Lagrange multipliers introduced to enforce non-negativity of the slack variables in the primal function. If $0 < \alpha_i < C$, then $\xi_i = 0$. If $\alpha = C$, then $\xi_i \geq 0$. To solve the problem, we again solve the dual problem with respect to $\alpha_i$, and calculate $w_o$ and $b_o$ exactly as before.

## 5.3 Mapping To a High-dimensional Space

If the TR set represents a non-linearly separable problem, the data points can be mapped from the input space to a high-dimensional **feature space**, where they are linearly separable. The approach we follow is analogous to the LBE for linear models. We define some function $\phi : \mathbb{R}^{m_0} \to \mathbb{R}^{m_1}$. Finding this function, however, is not always easy, unless we have some prior knowledge allowing us to select the proper feature space. Also, using high dimensional feature spaces can lead to overfitting.

Given this function, we map all points to the new feature space: $x \mapsto \phi(x)$. The problem is formulated as before, with a new training set $TR =< \phi(x_i), d_i >$, and a new hyperplane $g(x) = w^T \phi(x) + b = 0$. The notation used here incorporates the bias in the weight vector, with $w_0 = b$ and $\phi_0(x) = 1$:

$$\phi(x) = (\phi_0(x) = 1, \phi_1(x), \ldots, \phi_{m_1}(x))^T$$

The weight vector is now a linear combination of the feature vectors:

$$w = \sum_{i=1}^{N} \alpha_i d_i \phi(x_i)$$

and the hyperplane equation can be written as:

$$g(x) = \sum_{i=1}^{N} \alpha_i d_i \phi^T(x_i)\phi(x) = 0$$

Evaluating $\phi(x)$ may be intractable. Fortunately, under certain conditions we do not need to evaluate it directly, or even know the feature space itself. This is possible with a so-called **kernel trick**, i.e. by using a function $k$ to directly compute the dot products $\phi^T(x_i)\phi(x)$ in the feature space:

$$k : \mathbb{R}^{m_0} \times \mathbb{R}^{m_0} \to \mathbb{R}$$

This function is known as the **inner product kernel function**. It's also a symmetric function, so $k(x_i, x) = k(x, x_i)$. Consider the function $\phi(x) = \phi((x_1, x_2)^T) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)^T$. Given $x = (x_1, x_2)^T$ and $y = (y_1, y_2)^T$ in $\mathbb{R}^2$, we compute $\phi^T(x)\phi(y)$ in this

$$\begin{gathered}
\phi^T(x)\phi(y) = (x_1^2, \sqrt{2}x_1x_2, x_2^2)(y_1^2, \sqrt{2}y_1x_2, y_2^2)^T = \\
x_1^2y_1^2 + 2x_1y_1x_2y_2 + x_2^2y_2^2 = (x_1y_1 + x_2y_2)^2 = \\
((x_1, x_2)(y_1, y_2)^T)^2 = (x^Ty)^2 = k(x, y)
\end{gathered}$$

We can arrange the dot products in the feature space between the img of the input training patterns in a $N$ by $N$ matrix, called **kernel matrix**:

$$K = \begin{pmatrix} k(x_1, x_1) & \dots & k(x_1, x_N) \\ k(x_2, x_1) & \dots & k(x_2, x_N) \\ \vdots & & \vdots \\ k(x_N, x_1) & \dots & k(x_N, x_N) \end{pmatrix}, \; K = \{k(x_i, x_j)\}_{(i,j)=1)}^{N}$$

The kernel matrix is symmetrical, as the inner product kernel is symmetrical. Not every kernel function computes the inner product in a feature space; this property holds only for kernels gaining positive semi-definite kernel matrices. This is related to the matrix having non negative Eigenvalues.

Given $k_1$ and $k_2$ both kernels over $\mathbb{R}^{m_0} \times \mathbb{R}^{m_0}$. The following are also kernel functions:

- $k_1(x, y) + k_2(x, y)$;

- $\alpha k_1(x,y)\,\forall \alpha \in \mathbb{R}_+$;
- $k_1(x,y)k_2(x,y)$.

The problem is reformulated in both forms as follows.

**Quadratic optimization problem in feature space (primal form)**

Given the training examples $TR =< \phi(x_i), d_i >$, find the optimal values of $w$ which minimizes

$$\Psi(w,\xi) = \frac{1}{2}w^T w + C\sum_{i=1}^{N}\xi_i$$

satisfying the constraints

$$d_i(w^T\phi(x_i)) \geq 1 - \xi_i\,, \forall i = 1, \dots N$$
$$\xi_i \geq 0\,, \forall i = 1, \dots N$$

**Quadratic optimization problem in feature space (dual form)**

Given the training examples $TR =< \phi(x_i), d_i >$, find the optimal values of $\alpha_i$ which maximize

$$Q(\alpha) = \sum_{i=1}^{N}\alpha_i - \frac{1}{2}\sum_{i,j=1}^{N}\alpha_i\alpha_j d_i d_j k(x_i,x_j)$$

satisfying the constraints

$$\sum_{i=1}^{N}\alpha_i d_i = 0$$
$$0 \leq \alpha_i \leq C\,, \forall i = 1, \dots N$$

Remember that C is a user specified non-negative regularization hyperparameter.

To classify an unseen input pattern $x$ using the machine we trained, we compute $\sum_i \alpha_i d_i k(x,x_i)$, and then classify $x$ as $h(x) = sign(\sum_{i=1}^{n}\alpha_i d_i k(x,x_i))$. Note: we have to memorize the $x_i$ in the training set for the test phase.

Some examples of commonly used kernels include:

- **Polynomial Learning Machine**: $k(x, x_i) = (x^T x_i + 1)^p$ (where $p$ is a user-specified parameter);

- **Radial Basis Function Net**: $k(x, x_i) = e^{-\frac{\|x-x_i\|^2}{2\sigma^2}}$ (where $\sigma$ is a user-specified parameter);

- **Two-layer Perceptron**: $k(x, x_i) = tanh(\beta_0 x^T x_i + \beta_1)$ (where $\beta_0 > 0$ and $\beta_1 < 0$ are user-specified parameters).

For the first two kernels, we always get an inner product kernel, but for a two-layer perceptron the values of $\beta_0$ and $\beta_1$ must be chosen carefully to ensure we have the appropriate kernel. Also, using a kernel function of type RBF always leads to a feature space with an infinite number of dimensions.

## 5.4 SVMs For Nonlinear Regression

Earlier, we defined a task by specifying that the TR set contains a number of inputs and corresponding labels. Each label $d_i = f(x) + v$, i.e., is equal to the output value of the (unknown) target function plus a certain noise disturbance. This noise term $v$ is statistically independent from $x$.

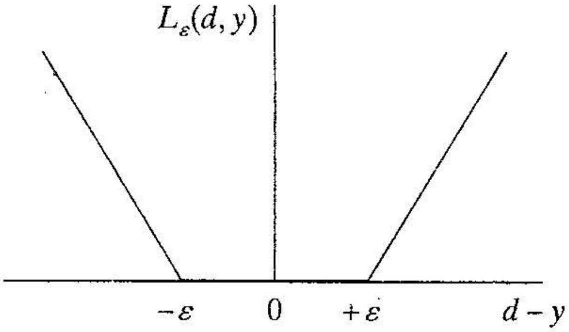
To estimate the regression output $y$ using a linear expansion of non-linear functions:

$$y = h(x) = w^T \phi(x)\,,$$

where $w = (w(0) = 1, w(1), \dots, w(m_1))^T$ and $\phi(x) = (\phi_0(x) = 1, \phi_1(x), \dots, \phi_{m_1}(x))^T$.

For SVMs, the loss function used is called **$\epsilon$-insensitive loss function**, and is defined as:

$$L_\epsilon(d, y) = \begin{cases} |d - y| - \epsilon & \text{if } |d - y| \geq \epsilon \\ 0 & \text{otherwise} \end{cases}$$



By plotting the function and the resulting margin, the $\epsilon$-tube is constructed. (?)

To formulate the optimization problem for regression, we must first introduce two sets of slack variables, $\xi'_i$ and $\xi_i$, such that:

$$-\xi'_i - \epsilon \leq d_i - w^T\phi(x_i) \leq \epsilon + \xi_i \,, \forall i = 1, \dots, N$$

We can now formulate the primal and dual problems for regression.

**Quadratic optimization problem for regression (primal form)**

Given the training set TR $= < x_i, d_i >$, find the optimal values of $w$ which minimize

$$\Psi(w, \xi, \xi') = \frac{1}{2}w^T w + C\sum_{i=1}^{N}(\xi_i + \xi'_i)$$

satisfying the following constraints

$$\begin{gathered} -\epsilon - \xi'_i \leq d_i - w^T\phi(x_i) \leq \epsilon + \xi_i \\ \xi_i \geq 0 \\ \xi'_i \leq 0 \end{gathered}$$

**Quadratic optimization problem fr regression (dual form)**

Given the training set TR $= < x_i, d_i >$, find the optimal values of $\alpha_i$ and $\alpha'$ which maximize

$$Q(\alpha, \alpha') = \sum_{i=1}^{N} d_i(\alpha_i - \alpha'_i) - \epsilon\sum_{i=1}^{N}\alpha_i + \alpha'_i - \frac{1}{2}\sum_{i,j=1}^{N}(\alpha_i - \alpha'_i)(\alpha_j - \alpha'_j)k(x, x_i)$$

satisfying the following constraints

$$\begin{gathered} \sum_{i=1}^{N}\alpha_i - \alpha'_i = 0 \\ 0 \leq \alpha_i \leq C \\ 0 \leq \alpha'_i \leq C \end{gathered}$$

Again, solving the dual problem gives us the best values of $\alpha_i$ and $\alpha'_i$, and then we compute the weight vector:

$$w = \sum_{i=1}^{N}(\alpha_i - \alpha'_i)\phi(x_i) \iff w = \sum_{i=1}^{N}\gamma_i\phi(x_i)$$

In this case, the support vectors correspond to non-zero values of $\gamma_i$.

So, the estimate function can be expanded accordingly:

$$\begin{aligned} h(x) = y = w^T \phi(x) = \\ = \sum_{i=1}^{N} \gamma_i \phi(x_i) \phi(x) = \\ = \sum_{i=1}^{N} \gamma_i k(x_i, x) \,. \end{aligned}$$

## 5.5 Pros and Cons of SVM

**Pros:**

- The regularization is embedded within the optimization problem, so it does not need any external hyperparameter for it;
- It automatically approximates SRM by finding the hypothesis with the best possible VC-dim;
- It's a convex problem, so the global minimum can always be found;
- Features are implicitly transformed through kernels;
- Linear model with bound of the complexity that depends on the margin (which is optimized);
- Rich set of non-linear decision functions in the input space via kernels.

**Cons:**

- Kernel and kernel parameters must be chosen explicitly;
- It uses a batch algorithm to update the weights, so no chance to parallelize;
- Very large problems were computationally intractable (although nowadays many efficient solutions have been proposed, including gradient descent ones);
- For soft margin SVM, since we have to select the $C$ parameter and the kernel function, we have no guarantee that the final model will have high accuracy.

Note that for the last point about soft margin, the VC-dim of the model is controlled by the width of the margin. If we choose the smallest possible margin (with just one support vector), we can classify an arbitrarily large number of training points correctly, thus the VC-dim will be infinite. If instead we have a very large width, we end up in a situation where all points in the TR set are used as support vectors, therefore leading to a low VC-dim. This is analogous to using a k-NN classifier with $k = 1$ and $k = l$, respectively. Also, the hyperparameter $C$ used for regularization must be chosen appropriately for the kernel hyperparameters, so cross-validation is needed.

## 5.6 Kernel Methods

In general, **kernelization** of an approach is done by substituting a dot product or similarity measure in a model with a kernel function that maps to an high-dimensional space.

$K(s, t)$ can be related to a similarity measure, comparing $s$ and $t$; e.g.:

$$d_k(s, t) = \sqrt{k(s, s) - 2k(s, t) + k(t, t)}$$

If the objects are similar (so the distance is small) the kernel will have high value. The focus is on the idea that in some situations, it's easier to focus on simplifying pairwise comparison by doing it in a different space rather than the representation of the single object.

The best choice of kernel for a given problem is still a research issue.

# Chapter 6

# Bias, Variance, and Ensembling

A model is trained on the basis of a specific training dataset containing a finite number of instances. Each training set is only one of the possible realizations of the universe of data obtained through empirical measurements, so different training sets can provide different estimates. When determining the error of a model, noise should be taken into consideration. This chapter will focus on the different aspects that influence the quality of the final model.

## 6.1 Bias, Variance, Noise

The expected error of the model at a point $x$ can be decomposed as:

- **Bias**: it quantifies the inability of the model to accurately approximate the target function. It can be seen as an approximation error.

- **Variance**: it quantifies the variability of the response of the model for different realizations of the training data. It can be seen as an estimation error.

- **Noise**: it's a random error included in a label in the training set, caused by imperfect measurements. It's irreducible, since it does not depend on the model.

Assume we have a regression task, where the target variable is $y$ and we use the squared error loss $L_2$. Our training set is TR $= < x, y >$, where $y = f(x) + \epsilon$, and $\epsilon$ is Gaussian noise with 0 mean and standard deviation $\sigma$. We fit a linear hypothesis $h(x) = w_1 x + w_0$ to minimize the SSE over the training data. We will have a systematic prediction error, since the model is not flexible enough to approximate certain target functions (e.g. second degree functions). Also, depending on the dataset, the $w$ parameters will be calculated differently.

So, given a new data point $x$, what is the expected prediction error? Assume that all the data points are drawn from a set of independent and identically distributed variables with the same probability distribution $P$. The goal is to compute, for any new $x$:

$$E_p[(y - h(x))^2]\,,$$

where $y$ is the value of $x$ that can be found in a dataset, and the expectation is over all training sets drawn according to $P$.

Let $Z$ be a random variable with possible values $z_i$, $i = 1 \dots l$ and probability distribution $P(Z)$. Its **expected value (mean)** is:

$$\bar{Z} = E_p[Z] = \sum_{i=1}^{l} z_i P(z_i)$$

If $Z$ is continuous, the sum is replaced with an integral, and the distribution function by a density one. The **variance** of $Z$ is:

$$Var[Z] = E[(Z - \bar{Z})^2] = E[Z^2] - \bar{Z}^2\,.$$

The proof of this last equality is the following:

$$\begin{gathered}
Var[Z] = E[(Z - \bar{Z})^2] = \sum_{i=1}^{l} (z_i - \bar{Z})^2 P(z_i) = \\
= \sum_{i=1}^{l} (z_i^2 - 2z_i\bar{Z} + \bar{Z}^2) P(z_i) = \sum_{i=1}^{l} z_i^2 P(z_i) - \sum_{i=1}^{l} 2z_i\bar{Z}P(z_i) + \sum_{i=1}^{l} \bar{Z}^2 P(z_i) = \\
= E[Z^2] - 2\bar{Z}\sum_{i=1}^{l} z_i P(z_i) + \bar{Z}^2 \sum_{i=1}^{l} P(z_i) = E[Z^2] - 2\bar{Z}\bar{Z} + \bar{Z}^2 \cdot 1 = \\
= E[Z^2] - 2\bar{Z}^2 + \bar{Z}^2 = E[Z^2] - \bar{Z}^2 = Var[Z]
\end{gathered}$$

We will use the form $E[Z^2] = \bar{Z}^2 + Var[Z]$ (variance lemma).

The error can be rewritten as:

$$E_p[(y - h(x))^2] = E_p[h(x)^2 - 2yh(x) + y^2] = E_p[h(x)^2] + E_p[y^2] - 2E_p[y]E_p[h(x)]$$

Let $\bar{h}(x) = E_p[h(x)]$ be the mean prediction of the hypothesis at $x$ when $h$ is trained on data drawn from $P$. Using the variance lemma defined above, we can rewrite the squared error as:

$$
\begin{aligned}
E_p[(y-h(x))^2] = \bar{h}(x)^2 + E_p[(h(x)-\bar{h}(x))^2] + f(x)^2 + E_p[(y-f(x))^2] - 2f(x)h\bar{(x}) = \\
= E_p[(h(x)-\bar{h}(x))^2] + \bar{h}(x)^2 - 2f(x)E_p[h(x)] + f(x)^2 + E_p[(y-f(x))^2] = \\
= E_p[(h(x)-\bar{h}(x))^2] + \textbf{(variance)} \\
+(h\bar{(x}) - f(x))^2 + \textbf{(bias}^2\textbf{)} \\
+E_p[(y-f(x))^2] = \textbf{(noise}^2\textbf{)} \\
= Var[h(x)] + Bias[h(x)]^2 + E_p[\epsilon^2]
\end{aligned}
$$

(the mean of $y$ is $f(x)$).

### 6.1.1 Regularization

Recall how we implemented regularization to calculate the loss function:

$$Loss(w) = \sum_p (d_p - o(x_p))^2 + \lambda \|w\|^2$$

By varying the value of the regularization hyperparameter $\lambda$, we ca obtain more or less complex solutions. Consider the following graphical examples to understand how it affects bias and variance of the model; the training sets here are taken from a sinusoidal distribution.

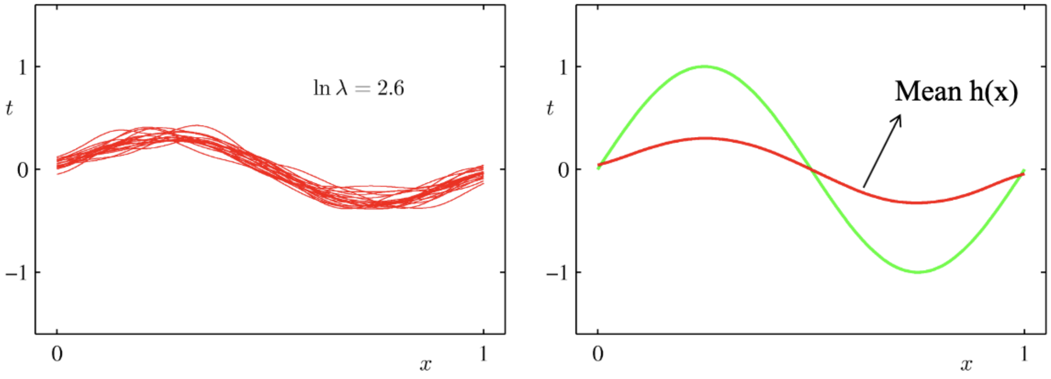


Figure 6.1: High lambda, high bias, low variance. Not too dependent on data, but the model is too rigid. The model underfits the data.

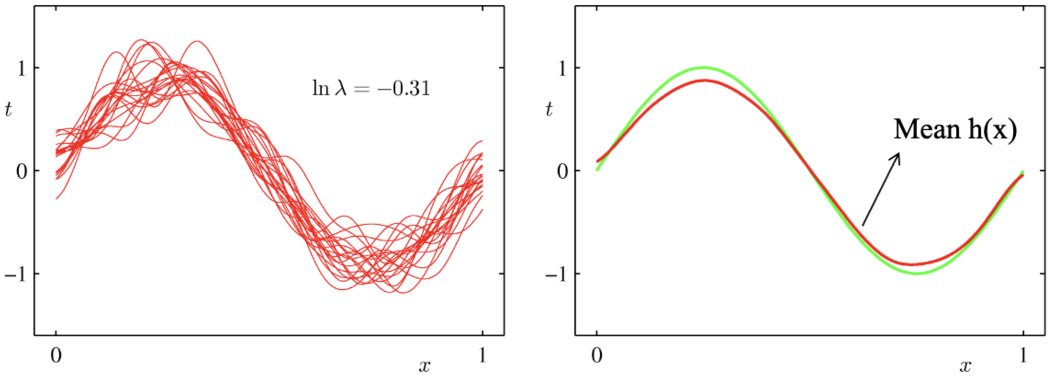


Figure 6.2: Low lambda, low bias, high variance. The model is more dependent on training data, but approximates better.

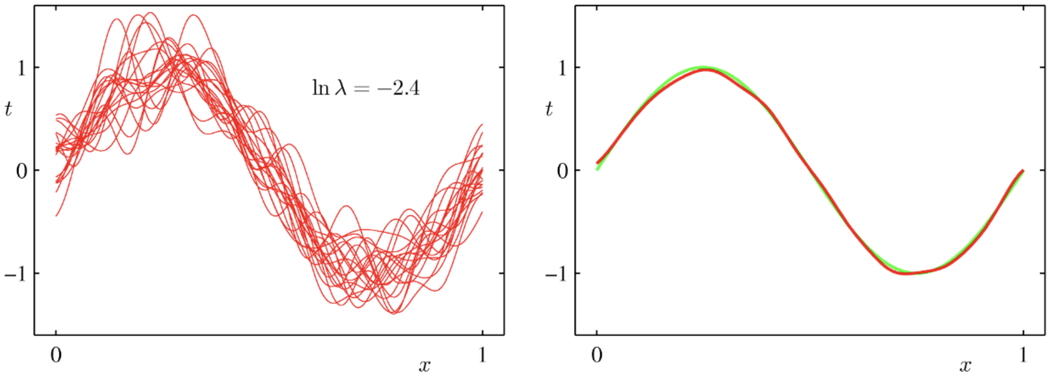


Figure 6.3: Very low lambda, very low bias, very high variance. Final hypothesis may produce an high test error. The model overfits the data.

This highlights once again how the goal of model selection is to find the best trade-off between model complexity (bias) and training error (variance).

## 6.2 Ensemble learning

Ensemble learning estimates the output of an instance by considering the prediction of multiple models trained differently instead of just one.

The simplest case is **voting**. All models are trained on the same training set. For regression, the simple average committee of the models is calculated:

$$o(x) = \frac{1}{K}\sum_{k=1}^{K} h_k(x)$$

The square error of a committee is less or equal than the mean of the square errors. For classification, we use different classifiers and take the majority vote (or the instance is classified after calculating the mean of the continuous outputs).

Another scheme is **bagging** (also called **bootstrap aggregating**). $K$ models are trained on different subsets of the training set, and each training is differentiating using bootstrap (so resampling with replacement). For regression, the final prediction will be the mean, while for classification we consider the majority vote.

If models have the same errors, however, we will not have any advantage in using ensembling. A solution is **boosting**: we first differentiate each training in order concentrating on errors, so we give more weight to "difficult" instances, such as ones that get misclassified by the previous classifier. The results are then combined in the output weights. If not stopped, boosting will learn to classify correctly all training instances, even when given a set of weak learners that are barely better than a random guesser. Even if bias is reduced, variance does not increase, so it resists well to overfitting (although it's not immune). However, it's sensitive to noisy data, since it will be deemed "difficult" and therefore receive an heavier weight.

# Chapter 7

# Deep Learning

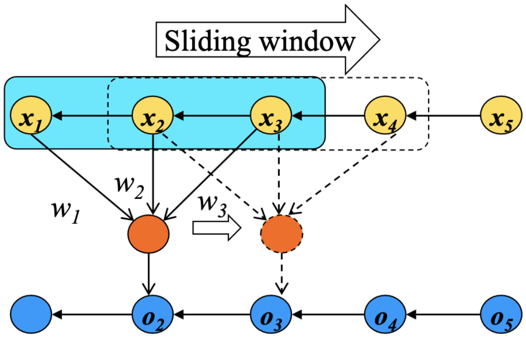
## 7.1 Convolutional Neural Networks

Convolutional Neural Networks are a specialized kind of neural network for processing data with a known, grid-like topology, such as 2D images. The name "convolutional" refers to the usage of the mathematical operation called **convolution**, indicated by an asterisk ($*$). This is an operation of two functions of a real valued argument, defined as follows:

$$s(t) = (f * g)(t) \overset{def}{=} \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \,,$$

The idea behind this operator is that we want to calculate the average of $f$ weighted by another function $g$ moved over time ("sliding"), calculated for a certain $t$. In convolutional network terminology, the first argument (here, $f$) is referred to as the **input**, and the second argument ($g$) as the **kernel**. The output is called **feature map**.

This operator can be applied to neural networks as well. Consider a simple network with one hidden layer:



Here, the output of each node in the hidden layer is calculated as $out_t = \sum_{i=1}^{3} w_i x_{t+i-2}$. In other words, the weights assigned to the inputs "slide" across the hidden layer.

Weights are tuned as usual by learning.

### 7.1.1 2D Convolution

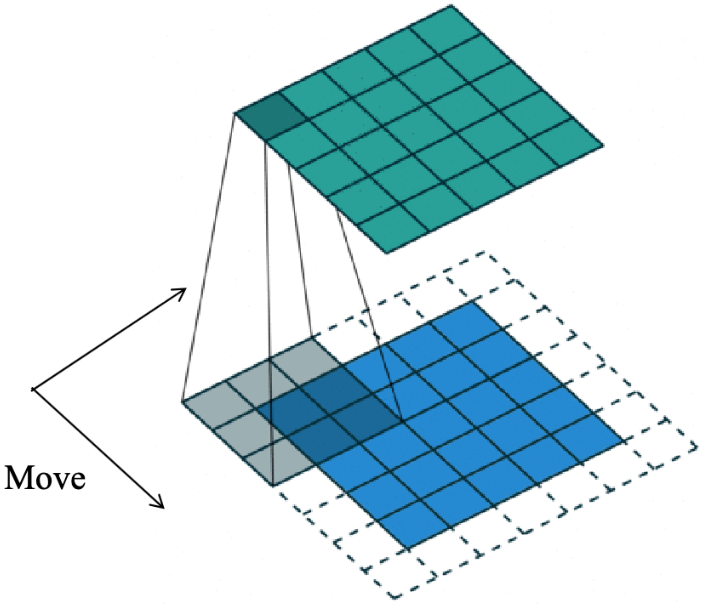
Discrete convolution can be seen as a multiplication by a matrix, with several entries constrained to be equal to other entries. The convolution over a 2D image $I$ with a kernel $K$ can be expressed as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)\,,$$

or, as expressed by many libraries, as the **cross-correlation function**:

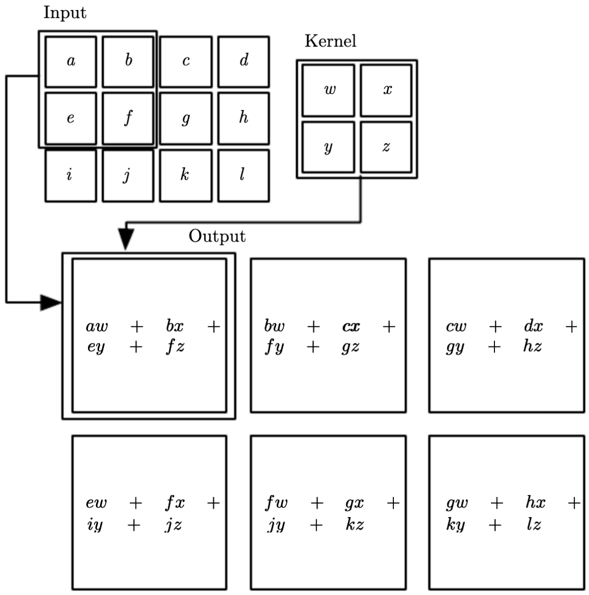
$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)\,.$$

The example below shows a 2D image with 25 pixels, and a 3x3 kernel (unit local receptive field) with a stride equal to 1 (i.e., the kernel moves across the image 1 pixel at a time; by choosing the stride we choose the size of the feature map). The image also has padding added to its edge.



Once the kernel reaches the end of the first "row" of pixels, it restarts from the position it started from shifted one pixel below. The full movement of the kernel over the image produces the feature map.

The image below better shows the matrix multiplication interpretation of convolution; the image is 4x3 and the kernel is 2x2. The stride is again equal to 1.



Each unit's weights are a **filter** trained to detect some specific feature or pattern in the image. Each filter produces the strongest response to a spatially local input pattern, and then the filters obtained by training are applied to the whole (global) image, so that features and patterns can be identified regardless of where they are positioned on the image.

The size of the feature map can be reduced via pooling. Some examples of pooling are:

- Subsampling, using a stride greater than 1;
- Calculating a mean (average or weighted average);
- Using the max pool operation (most common option).

This way, instead of producing a value for every single pixel of the original image, we get a smaller set of pixels where each value is obtained by considering the values of neighboring ones (calculating the mean or max value). Pooling also helps to make the representation approximately invariant to small translations of the input, since the mean or max value of a neighborhood of points is unlikely to be affected by a small translation of the pixels in the image.

CNN exploits **weight sharing**, where the number of connections in the network is kept the same while reducing the number of actual free parameters. The produced sliding window of units is applied over a segment of the input, and reapplied multiple times to produce various layers of feature maps. Training of the weights is usually done via backpropagation. Since these networks tend to be big and deal with large amount of data, many hyperparameters are fixed by experience or by suggestions of experts, since it would be too expensive to run cross-validation.

## 7.2 Deep Learning

The Deep Learning framework includes many different models, such as:

- Deep Neural Networks;
- Convolutional Neural Networks;
- Deep belief Networks;
- Recurrent and Recursive Neural Networks.

They differ from "shallow" models in that they have a big amount of layers.

There's many different approaches to build a deep learning model. All these approaches have some aspects in common:

- Multiple layers of nonlinear processing units;
- Supervised or unsupervised learning of feature representation in each layer, with the layers forming some hierarchy from low to high level features;
- Hierarchical sparse/distributed representation of the input data.

The core concept at the base of deep learning is increasing the level of abstraction of the data through the use of several layers; for example, an image can be gradually abstracted on each layer, first as a vector of intensity values per pixel, then a set of edges, then regions of a particular shape, and so on. We've already mentioned this idea with CNNs: the original complicated mapping is broken down into its simple elements, by gradually calculating simpler mapping at each layer. A series of hidden layers extracts increasingly abstract features from a set of example images. Additionally, these abstract features, once learned by the units, can also be combined together to generalize on examples that were never seen during training.

In general, deeper networks are often able to use less units per layer, thus less free parameters as well and less training data required to achieve a good generalization. Still, many layers may be harder to be trained, so there's a need to improve the techniques we know regarding gradient descent, regularization, and data exploitation.

### 7.2.1 Insights

**Why So Many Layers?**

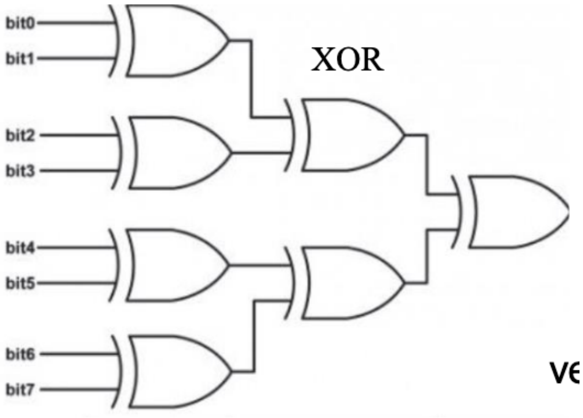
Imagine a two-layer circuit of logic gates, which can represent any Boolean function. Any Boolean function can indeed be written as a sum of products (i.e., in disjunctive normal form). With logical circuits of depth two, the number of logic gates required to represent most Boolean functions is exponential w.r.t. input size.

An example of such function is the parity function: it returns 1 if there is an odd number of 1s over $N$ binary inputs (i.e., $N$ bits), 0 otherwise. If we were to implement this function with logic gates, assuming $N$ inputs, we would need:

$$\frac{2^N}{2} + 1 = 2^{N-1} + 1$$

gates, since we have to perform 1 OR and exactly $2^{N-1}$ ANDs.

We can propose an alternative solution with a polynomial number of gates, by increasing the number of layers to $log(N)$. The solution for $N = 8$ is shown below (in a simplified view where each XOR corresponds to 2 AND gates and 1 OR gate). In this solution, the number of gates is greatly reduced from $2^7 + 1 = 129$ to only $7 * 3 = 21$.



The universal approximation theorem states that even with 1 hidden layer, a NN can approximate any possible function, however it does not specify the number of units needed. For some families of functions a boundary on this number can be found, but as seen in the example above, the bound may be exponential w.r.t. the dimension of the input. Also, there exist families of functions which can be approximated efficiently by a NN with depth greater than some value $d$, but which require a much larger model if depth is restricted to be less or equal than $d$.

This theorem also implies that regardless of what function we are trying to learn, a large MLP is able to represent the function, but there's no guarantee that that the training algorithm will be able to learn that function, either because it can't find the value of the parameters that correspond to the desired function, or because it might choose the wrong function due to overfitting. So another advantage to using multiple layers is ensuring that the learning algorithm can actually properly learn.

The inductive bias is: choosing a deep model encodes the (very general) belief that the function we want to learn should involve composition of several simple functions. If our task actually matches our bias, then the deep shape of the learner is suitable, and those deeper models also perform better than shallow ones. Typically, these tasks are the ones that involve images or language, but for other tasks that deal with different data this deep structure may not be appropriate.

Other challenges that motivate the usage of deep neural networks are related to the curse of dimensionality and manifold learning.

Recall that the curse of dimensionality is the phenomenon in which as the number of dimensions in the data increases, the less dense it is, thus becoming a problem for distance based machine learning algorithms. One challenge posed by this phenomenon is a statistical one: this challenge arises when the number of possible configurations of an input $x$ is much larger than the number of actual training examples. Let's say we want to estimate the probability density for some point $x$ by returning the number of training examples in the same unit volume cell as $x$ divided by the size of the training set. But if we want to estimate this density for a point that has no close neighbors within the unit volume cell (as is often the case when the dimensionality of data is high), there's no immediate way to calculate it. Additionally, even if present, neighbors are likely going to be very few. But deep learning is the solution: it can learn the single features it identifies in the training data, and those features can then be combined to produce an output even for instances that were never seen before.

A manifold is a connected region. Mathematically, it's a set of points associated with a neighborhood around each point. From any given point, the manifold locally appears to be a Euclidean space. As an example, in the real world, we experience the surface of the Earth as 2D, but in reality it's a sphere in 3D space. In the case of Machine Learning, "manifold" is a term used to broadly refer to a connected set of points that can be approximated well by considering only a small number of dimensions. Different points can also have a different number of dimensions, typically when the manifold intersects itself. **Manifold learning** assumes that the only interesting inputs occur only along a collection of manifolds containing a small subset of points, with interesting variations in the output occurring only across directions that lie on the manifold, or only when moving from one manifold to the other. This assumption is often used for data types such as images, text, or audio that occur in real life, since uniform, artificial noise never resembles the inputs from those domains, so the probability distribution of the only valid data is highly concentrated. *?

### Representation Learning

With Deep Learning, other concepts are emerging, such as representation learning, which is the set of methods that automatically discover the best representation of raw data for some other task. It's especially useful for more complex data types like images, text, or audio data. Deep Learning methods are also representation learning methods with multiple levels of representation.

The reason behind representation learning is that many information processing tasks can have varying difficulty based on the way the information is represented. In Machine Learning, a good representation is one that makes a subsequent learning task easier. Supervised learning in MLPs is an example that leads to an automatic representation at every hidden layer taking on properties that make the output layer task easier (think of the XOR example). Another example is CNNs, since they take as input a raw image and automatically learn to identify the main features of the image, decomposing it into its parts.

The hidden representation of data can be obtained via **pretraining approaches**, semi-supervised learning, which means we can learn a representation for the unlabeled data and use it to solve supervised tasks. The representation can be then exploited with **transfer learning** approaches.

**Pretraining** **Greedy Layer-Wise Unsupervised Pretraining** was the first approach to make it possible to train a deep supervised network. It makes the training of the whole network much easier, since there's no need for an end-to-end training from the output layer to the previous hidden layers with a gradient descent, which used to be difficult for deep models. The adjective "greedy" refers to the fact that each layer is optimized independently in an unsupervised way. This methods works not only as a good initialization strategy, but also as regularization regarding the complexity of the original data, since it extracts the features that simplify the unsupervised process. It also reduces the variance of the estimation training process, since pretrained models tend to gather in a smaller region after training).

**Autoencoders** An autoencoder is a neural network that is trained to attempt to copy its input to its output. It has a hidden layer $h$ that describes a **code** used to represent the input. The network can be seen as two components: the encoder function $h = f(x)$, and the decoder $r = g(h)$ that produces a reconstruction. Autoencoders are specifically trained so that they are unable to copy the input perfectly, but instead only copy parts of the data or copy it only approximately. Since the model is forced to prioritize only some aspects of the input, it often learns useful properties of the data.

Autoencoders can be:

- **Undercomplete**, if the code dimension is less than the input dimension. This autoencoder is forced to learn the salient features of the training data by the constraints in the architecture itself.

- **Overcomplete**, if the code dimension is greater than the input dimension. By itself the autoencoder would learn to copy the input as is, so some form of regularization is needed to introduce constraints in the sparsity of the representation, robustness in the noise, and other properties of interest.

A commonly used type of autoencoders are denoising autoencoders. They are trained on a set of images, finding the best representation of them (the one they can obtain even from corrupted/noisy data). Once they are presented with an image with added noise, they are capable of reconstructing the original.

Stacked autoencoders are multiple autoencoders stacked on top of each other. The algorithm used to train a stacked autoencoder is the following:

1. Train the first layer as an autoassociator (i.e., an autoecoder whose input and output are the same), in order to minimize the reconstruction error of the raw input;

2. The hidden units' outputs in the autoassociator are now used as input for another layer, also trained to be an autoassociator. Repeat this step until the desired number of layers is reached.

3. Take the last hidden layer output as input to a supervised layer, and initialize only its parameters (randomly or by supervised learning).

4. Fine-tune all the parameters of the architecture with respect to the supervised criterion.

The idea is that the unsupervised greedy layer-wise initialization put all the parameters of all the layers in a region of parameter space from which a local optimum can be reached via local descent.

In the past, pretraining acted as the start of Deep Learning. It can yield improvements for some tasks, especially NLP, but not in others. The general role of pretraining is nowadays under critical revision by researchers.

**Transfer Learning** **Transfer learning** refer to the situation where what has been learned in one setting is used to improve generalization in another setting. In this case, we use the best representation of the data found by pretraining to improve the generalization capabilities of some model. In **domain adaptation**, the task remains the same between each setting, but the input distribution is different. With **multi-task learning**, a trained model can be used for a task that's different from the one it was trained for (it will receive the same inputs, but the target variable(s) change).

### Distributed Representation

As opposed to **symbolic representation**, in which each input corresponds to an unique symbol, in **distributed representation** each input is represented by a set of elements that can be set separately from each other. As an example, consider a vector of $n$ binary features, which can take $2^n$ possible configurations, each corresponding to a different region in input space. This is a type of distributed representation. The same data can be represented via one-hot encoding: $n$ feature detectors, each corresponding to the detection of a certain input. Only $n$ configurations of the input space are possible. This is a type of symbolic representation. The two representations are visualized below.

| Input | barks | hasFur | hasLegs | isFruit |
|---|---|---|---|---|
| Dog | 1 | 1 | 1 | 0 |
| Cat | 0 | 1 | 1 | 0 |
| Apple | 0 | 0 | 0 | 1 |
| Kiwi | 0 | 1 | 0 | 1 |

| Input | isDog | isCat | isApple | isKiwi |
|---|---|---|---|---|
| Dog | 1 | 0 | 0 | 0 |
| Cat | 0 | 1 | 0 | 0 |
| Apple | 0 | 0 | 1 | 0 |
| Kiwi | 0 | 0 | 0 | 1 |

Table 7.1: Distributed representation on the top, symbolic representation on the bottom.

In distributed representation, values assigned to attributes can also be real-valued. By using this representation, we can calculate similarity between two different concepts (inputs); in symbolic representation, the distance between two inputs is always $\sqrt{2}$.

In general, distributed representation is used in input only if we have background knowledge (one-hot encoding is used instead), while it is commonly used automatically during learning. A big advantage is that there's no need to characterize all the possible configurations of the input, but only the distinction across specific concepts (e.g., in the example above, we learn the characteristics "barks", "hasFur", "isFruit" instead of learning exactly what a dog or an apple are).

Sharing attributes among inputs allows the model to both treat them similarly, and automatically learn such similarity. As per the example above, "dog" and "cat" share some features. If we are presented with textual data containing several different words, and by learning the model knows the similarity between "cat" and "dog", a sentence that contains the word "cat" can inform the predictions made for a sentence containing the word "dog", and vice-versa. Since they're represented in a distributed way, the distance between the two is actually measurable and comparable with respect to other inputs (e.g., "dog" is closer to "cat" but both are further away from "apple", while with one-hot-encoding their distance is $\sqrt{2}$ regardless of how similar the concepts represented by those words are).

One drawback of distributed representation is that it's more difficult to interpret.

### 7.2.2 Techniques

When implementing a deep neural network, there's many technical aspects to consider. As already stated before, the deeper the network, the less units are needed for each layer, and therefore there will be less parameters to train, but some layers may be difficult to train. This section will focus on the methods that need improvement and their issued when used with deep NNs.

The first issue is related to the gradient. With deep NNs, there's now a lot of layers to backpropagate the gradient through. If the weights are too small, they will shrink exponentially (gradient vanish), while if they are too big, they will grow exponentially. This happens because the repetition of multiplication through many layers can introduce cliffs in the cost functions, that when traversed will drastically increase or decrease the weights. One solution is using **gradient clipping**: if the norm of the gradient $||g||$ is greater than some threshold $v$, then $g = v * \frac{g}{||g||}$, so the movement towards the gradient direction still happens, but the weight updates are bounded.

For gradient vanish, traditional activation functions have gradients in the range $[-1, 1]$, and backpropagation computes gradients via chain rule repeated through layers. This means that for each layer, the gradient of the layer in the front will be multiplied by a value $d$, $|d| \geq 1$, so the layers closer to the input will be trained much more slowly. There are many approaches to deal with it, one of which is using the **ReLU (Rectified**

**Linear Unit)** activation function:

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

This function makes it so that the derivative of a unit remains large as long as the unit is active. *?

**Batch normalization** is a method that normalizes each batch by calculating each individual batch statistics such as mean and variance for each layer. Each matrix (batch x activation of units) is normalized with mean and variance, shifting the values to zero-mean and unit variance. This technique helps to keep the normalization of the input across all layers of the network. It also achieves a faster learning and higher accuracy for Deep Learning.

**Dropout** is a method that makes bagging more practical for large neural networks. Bagging involves training multiple models on different subsets of the data, and then evaluating multiple models on the same test example. Dropout trains the ensemble consisting of all sub-networks that can be formed by removing non-output units from an underlying base network. In most modern neural networks, we can effectively remove a unit from a network by multiplying its output value by 0. Then, a minibatch-based learning algorithm is used to train one working sub-network at a time. The sub-network is selected at random, with each binary mask to apply to the original network having a different probability set as an hyperparameter (e.g., 0.8 for input units and 0.5 for hidden units). The sub-networks inevitably share weights, since they're obtained from the same base network; this causes the training of the sub-networks to find good settings for the parameters.

Dropout has a regularization effect as well: it avoids to train all units on all training data and reduces unit interactions. It also reduces variance without affecting bias just like bagging does. It even regularizes singular hidden units to be not just good features, but features that can be good in different contexts (different sub-networks). It can be used for any model that uses distributed representation and SGD training.

# Chapter 8

# Randomized Machine Learning

Randomness can be seen as an effective part of Machine Learning approaches. It can enhance different aspects, such as data processing, learning, hyper-parameter selection, and so on. It can both enhance predictive performance and alleviate difficulties of classical ML methodologies.

Randomness can also be exploited to reduce the training needed for a MLP/RNN, by implementing **randomized neural networks** (or **random weights neural network**). The network is constructed with randomly connected hidden layers with fixed weights. The training involves only the output weights, overcoming the problems associated with the typical training algorithms used for neural networks. The general architecture of a randomized NN is defined by two components:

- The hidden layer (untrained, randomly initialized). It linearly embeds the input into a high-dimensional feature space where the problem is more likely to be solved linearly.
- The readout layer (trained). It combines the features in the hidden space for output computation.

**Cover's theorem** states: "a complex pattern-classification problem, cast in a high-dimensional space non-linearly, is more likely to be linearly separable than in a low dimensional space, provided that the space is not densely populated".

## 8.1 Pros and Cons

A large set of hidden units can provide a sufficient "basis expansion" to project the non-linearly separable data into an higher dimension, and is also extremely efficient at doing so.

On the other hand, focused models with a smaller number of adaptive units trained on the specific task at hand will appear more “elegant”.

# Chapter 9

# Self-Organizing Maps

Unsupervised Learning includes all tasks in which there is no teacher to assign labels to examples. The training set is a set of unlabeled data $< x_i >$. Typical unsupervised learning tasks include clustering, dimensionality reduction, and modeling data density. One advantage of unlabeled data is that it's cheaper to obtain compared to labeled data. In this chapter, we will see an historical NN model called **Self-Organizing Maps** (or **Kohonen Maps**) capable of solving unsupervised learning tasks.

## 9.1 Clustering

The goal of clustering is to find subsets of data points that are connected to each other through some form of similarity. Points within a cluster are more similar to each other than any point belonging to a different cluster. Each cluster has a **centroid**, which corresponds to the mean of all points in that cluster.

There's different ways to define similarity among points. One popular measure is Euclidean distance, but it's just one of many possible measures.

**Vector quantization** is a set of techniques aimed at encoding a data manifold $V \subseteq \mathbb{R}^n$ using only a finite set of vectors $w = (w_1, \ldots, w_k), w_i \in \mathbb{R}^n$, called reference (or codebook) vectors. A data vector $x \in V$ is described by the best-matching ("winning") reference vector $w_{i^*(x)}$ for which the distortion error $d(x, w_{i^*(x)})$ is minimal. The manifold is therefore divided into a number of sub-regions:

$$V_i = \{x \in V : \|x - w_i\| \leq \|x - w_j\| \forall j\}$$

called **Voronoi polyhedra**, out of which each data vector $x$ is described by the corresponding reference vector $w_{i^*(x)}$. Computing the optimal Voronoi diagram is an NP-complete problem.

The idea is to use vector quantization for clustering tasks. A clustering can be seen as an optimal partitioning of the data into regions (clusters) approximated by a reference (centroid). The distortion error is calculated by the chosen similarity measure. The average value of the distortion error over the distribution of inputs is the average quantization error; this will be the loss function. Below is the loss function both in the continuous and the discrete version. The symbol $w$ is used for reference vectors.

$$E = \int f(d(x, w_{i^*(x)}))p(x)dx = \int \|x - w_{i^*(x)}\|^2 p(x)dx$$

$$E = \sum_{i=1}^{l} \sum_{j=1}^{k} \|x_i - w_j\|^2 \delta_{win}(i,j)$$

$\delta_{win}(i,j)$ is equal to 1 if $j$ is the winner for $i$, 0 otherwise.

### 9.1.1 K-means

K-means is one learning algorithm used for clustering. This section will present the on-line version of the algorithm; the following is its pseudocode:

1. Choose $k$ cluster centers to coincide with $k$ randomly selected points in the hypervolume containing the pattern set.
2. Assign each pattern to the winner cluster center (centroids).
3. Recompute the centroids using the current cluster members.
4. If the convergence criterion is not met, repeat from step 2.

Given the cluster centers $w_1, \dots w_k$, for each $x$ the winner is the nearest prototype:

$$i^*(x) = \arg\min_i \|x - w_i\|^2 ,$$

then, for each cluster $i$, the new centroid is calculated as:

$$w_i = \frac{1}{|cluster_i|} \sum_{j:x_j \in cluster} x_j .$$

This algorithm is popular because it's easy to implement and is generally efficient. One drawback is that $k$, the number of clusters to partition the data into, must be provided by the user. K-means is good for finding globular clusters, but cannot separate irregularly shaped clusters.

The local minima of the loss function depend on the initialization. To avoid confinement to local minima, a common approach is to introduce a **soft-max** adaptation rule: not only the winning centroid is updated, but all other centroids as well, in an amount that's inversely proportional to their proximity to the current $x$. An instance of this strategy in neural networks is **Kohonen self-organizing maps**.

### 9.1.2 Self-Organizing Maps

Self-organizing map neural networks consist of $N$ neurons located on a regular low-dimensional (usually 2D) lattice. Each unit corresponds to a coordinate on the map, each unit receives the same input $x$, and has a weight $w$. SOMs learn a topology preserving map from input space to a lattice of neural units. Topology preserving means that neighboring units will respond to similar input patterns, and data points close in the input space are mapped onto the same (or to different but close) map units. SOMs can be used for many different tasks, such as clustering, pattern recognition and vector quantization, data compression, projection and exploration, and so on.

**Competitive learning** is an adaptive process in which neurons gradually become specialized to different sets of samples. This competition makes it so that the winner is allowed to learn more than the other neurons.

The pseudocode for the SOM learning algorithm is the following:

1. Randomly initialize the map (i.e., randomly initialize the weights).
2. Draw a sample input $x$.
3. Competitive stage: the winner is the unit with the $w$ most similar to $x$.
4. Cooperative stage: upgrade the weight of all the units with topological relationships (on the map) with the winner unit using the soft-max adaptation rule.

Note that the soft-max rule here updates neurons close on the map, not in the data space.

In the **competitive stage**, the current input vector $x$ is compared with all the unit weights using Euclidean distance. The winner is chosen as

$$i^*(x) = \arg\min_i \|x - w_i\|^2 \, .$$

In the **cooperative stage**, the weight of the winning unit and the weight of all units in its neighborhood are moved closer to the input vector:

$$w_i(t+1) = w_i(t) + \eta(t) h_{i,i^*(x)}(t)[x - w_i(t)] \, ,$$

where $h(t)$ is the neighborhood function/kernel, i.e., it decreases monotonically for increasing $\|i - j\|$:

$$h_{i,i^*(x)}(t) = exp(-\frac{\|r_i - r_{i^*}\|^2}{2\sigma_{nh}^2(t)}) \, ,$$

where $r_i$ and $r_{i^*(x))}$ are the coordinates of the units on the lattice.

The learning rate $\eta$ and the radius value $\sigma$ are decreased as a function of iteration $t$; the neighborhood is wide at the beginning and gradually shrinks during learning. In practice, the shape of the neighborhood is chosen from a standard geometrical shape, to include a finite set of the map units.

The update rule of the cooperative stage is fundamental to the formation of topologically ordered maps. The weights are not modified independently of each other but as topologically related subsets. At each update, the subset is selected on the basis of the neighborhood of the winning unit.

After training, the map can be used to visualize different features of the SOM and the data represented on the map, such as the density of the reference vectors and the distance between reference vectors of neighboring units.

# Chapter 10

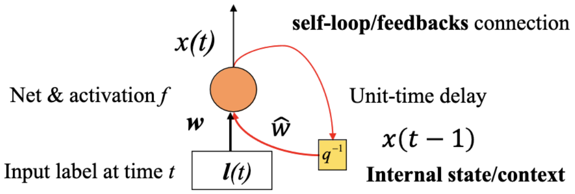
# Recurrent Neural Networks

Up until now, we considered feedforward neural networks. The input is read from the first layer, then traverses a number of hidden layers, and the prediction is finally produced by the output layer. Recurrent neural networks are a different category of architecture, based on the addition of feedback loops to the network topology. These self loops provide the network with dynamical properties, introducing the ability of holding a memory (state) of past computations of the model.

RNNs have been the reference approach for sequence processing, especially for speech and text recognition, processing, and generation. The type of data handled by these models is structured; it is usually an ordered set of sequences of vectors.

## 10.1 Memory

The introduction of feedback loops allows the network to hold a memory about past computations. This memory is needed because the output of a certain input depends on previous outputs produced by the same network.



The output of the node is calculated for a time $t$ recursively, as:

$$x(t) = \begin{cases} 0 & t = 0 \\ \tau(i(t), x(t-1)) = f(w^T i(t) + \hat{w}x(t-1) + \theta) & t \geq 0 \end{cases}$$

Here, $f$ is the activation function of the unit, $i(t)$ is the input label at time $t$, $\hat{w}$ is the recurrent weight (the one that's coming from a feedback), and $\theta$ is the bias. The internal state summarizes the past information, and changes each time the unit produces a new output. The encoding of the past memory is also adaptive. $\tau$ is the state transition function realized by the NN. $x(t)$ here refers only to one state, but it can also include a set of states.

## 10.2 Properties

Many RNN architectures are possible, but even a simple one with a few nodes each with its own feedback loop is already incredibly powerful. They are universal approximators of non-linear dynamic systems, and are Turing equivalent (they can simulate any automata).

RNN models are based on the following assumptions:

- **Causality**: a system is causal if the output at time $t_0$ only depends on inputs at time $t < t_0$ (necessary and sufficient for internal state);
- **Stationarity**: time invariance after model training. The state transition function $\tau$ is independent on node $v$ of the sequence.
- **Adaptivity**: transition functions are realized by NN with free parameters, so they are learned from data.

RNNs can also be **unfolded**. Unfolding a RNN means representing it as a graph with a repetitive structure corresponding to a chain of events (so it represents how the same model behaves through time). Unfolding is associated with weight sharing between unfolded layers. The learning algorithms used for RNNs must account the set of encode transitions developed by the model for each step of the inputs. **Backpropagation through time (BPTT)** and **real time recurrent learning (RTRL)** are two supervised learning algorithms designed for RNNs that compute gradient values of the output errors across an unfolded network.

An approach related to RNNs are **transformers**. A transformer is a deep learning architecture that relies on a mechanism called **attention**, which allows the model to access any preceding point along the sequence. The attention layer weighs all previous states according to some learned measure of relevance. This method is especially useful for language translation, where the context of the whole sentence (given by all the previous words) is important to define the meaning of a word.

## 10.3 Echo State Networks

Echo State Networks are an emerging paradigm for efficiently modeling RNNs. An ESN consists of:

- A **reservoir** of sparsely connected recurrent units, untrained after a **random initialization**;
- Simple feedforward readout of linear units, trained by efficient linear methods.

They are efficient in the sense that they don't need any training of the recurrent units, and yet they're capable of easily solving certain tasks (under specific conditions). The reservoir acts as a projection of the input sequences into values belonging to a state space. Different input sequences are discriminated on a suffix-based fashion with no training needed: the closeness between states is proportional to the length of their common suffix.

RNNs can be extended to rooted trees, as **Recursive Neural Networks**.

# Chapter 11

# Structured Domain Learning

Structured Domain Learning concerns all learning tasks that involve structured data, i.e., data represented with complex structures such as sequences, trees, graphs, multi-relational data, and so on. Feature based and adjacent matrix representations are incomplete for structured data; so instead of forcing a specific representation, the goal is to learn a mapping between a Structured Domain of information and a (discrete or continuous) space that is equivalent to performing a regression or classification task.

## 11.1 Transductions

The mapping that matches each information object to a point in a discrete/continuous space is called **transduction**. Imagine we're given a training set of graphs (for example, representations of molecules), each paired with a corresponding regression value of class label, in the form $(graph_i, target_i)$. The goal is to find the transduction $T$ such that $T(graph)$ returns the point in the space that corresponds to the "correct" output.

A transduction on a graph can be either **Structure-to-Structure** (for node based tasks), **Structure-to-Scalar** (for graph based tasks), or **Non-isomorphic**.

The main properties of models for graphs are:

- Support of different graph structures, as well as flexible representations of global, node, and edge attributes that can be customized according to the specific task at hand;
- Handling of non-euclidean geometry;
- Modularity and compositionality, since they can learn independent mechanisms that can be reused in several parts of the graph;

- Cross-modality, as they learn how to combine structured and unstructured data sources;
- Multiscale integration of the data, using different levels of complexity for granular data.

TODO

## 11.2 Deep Graph Networks

Deep Graph Networks are based on the concept of **message passing**. Each node in the graph computes a “message” for each of its neighbors, obtained as a function of the node label, the neighbors, and the edge between them. Each node then aggregates the nodes it receives, using a permutation-invariant function (i.e., it doesn’t matter in which order the messages are received). This function is usually a sum or average. Finally, the nodes update their attributes as a function of their current attribute and the aggregated messages, and combine them with the free parameters. After this message passing phase, an output is emitted for either each node or the whole graph, again using a permutation-invariant function that aggregates node representation, called **global pooling**. The embeddings $h_v^{(l)}$ of each node $v$ at layer (iteration) $l$ are calculated as:

$$h_v^{(l)} = AGG_{W^{(l)}}(x_v, h_v^{(l-1)}, \{h_u^{(l-1):u \in N_v}), l = 1, \ldots, L$$

where $N$ is the neighborhood of $v$, calculated according to the adjacency matrix of the graph. If this computation is applied to each node of the graph, we obtain the equivalent of a parallel, unordered visit of the graph at each iteration $l$.

A convolutional technique used for Deep Learning on graphs is **layering**. The idea is that mutual dependencies between nodes are managed architecturally through different layers; instead of iterating at the same layer, each vertex/node can take the context of other vertices/nodes computed in the previous layers, progressively accessing the whole graph/network. CNNs cannot be immediately applied to graphs. In data that can be represented by a 2D grid (such as images), the CNN kernel can be applied to the grid, calculating the average of the cell values in the neighboring window; the neighbors of a vertex are ordered, and have fixed size. With graphs, the neighbors are unordered and variable in size, so the receptive field must be constrained to the neighbors, exploiting weight sharing as permutation-invariant so that it is not affected by the ordering of the nodes.

Deep Graph Networks are inherently deep: the process is not local, because each node receives and spreads information across the whole graph. Each message passing iteration $l$ is a different layer of the network, and the context of a given node is obtained by composing the information at all layers. The receptive field is incrementally extended when the number of layers is increased, so the depth of the network is functional to the context growth. By training the model, we learn how to represent the graph.

For the recursive approaches (Graph Neural Networks and Graph Echo State Networks), the diffusion process is similar, but $l$ is the number of message passing iterations on the same layer (or, seen from a different perspective, using different layers with shared weights among layers). In GNN and GESN cycles are allowed in state computation, and the state is computed iterating the state transition function until it converges.

Some issues that affect GNNs are:

- **Over-smoothing**: a phenomenon in which interacting nodes converge to indistinguishable representations as the number of GNN layers increase. This is typically caused by having a number of layers much higher than the problem radius (as in, the furthest distance a message has to travel in the graph);

- **Over-squashing**: similar to over-smoothing, a phenomenon in which an exponentially growing amount of information is squashed into a fixed size vector (the aggregations computed at each node). It's caused by the layer composition of the network itself.

- **Heterophilic graphs**: as opposed to homophilic graphs, where nodes in the same neighborhood share the same class, heterophilic graphs contain nodes with the same class that are further apart. Predictions that rely on a node's neighbors can be misleading. The heterophily of a graph is calculated as:

$$h_G = \frac{\#intraclass\ edges}{\#edges}$$

# Bibliography

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[2] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1999.

[3] Tom M. Mitchell. *Machine learning*. McGraw-Hill, 1997.