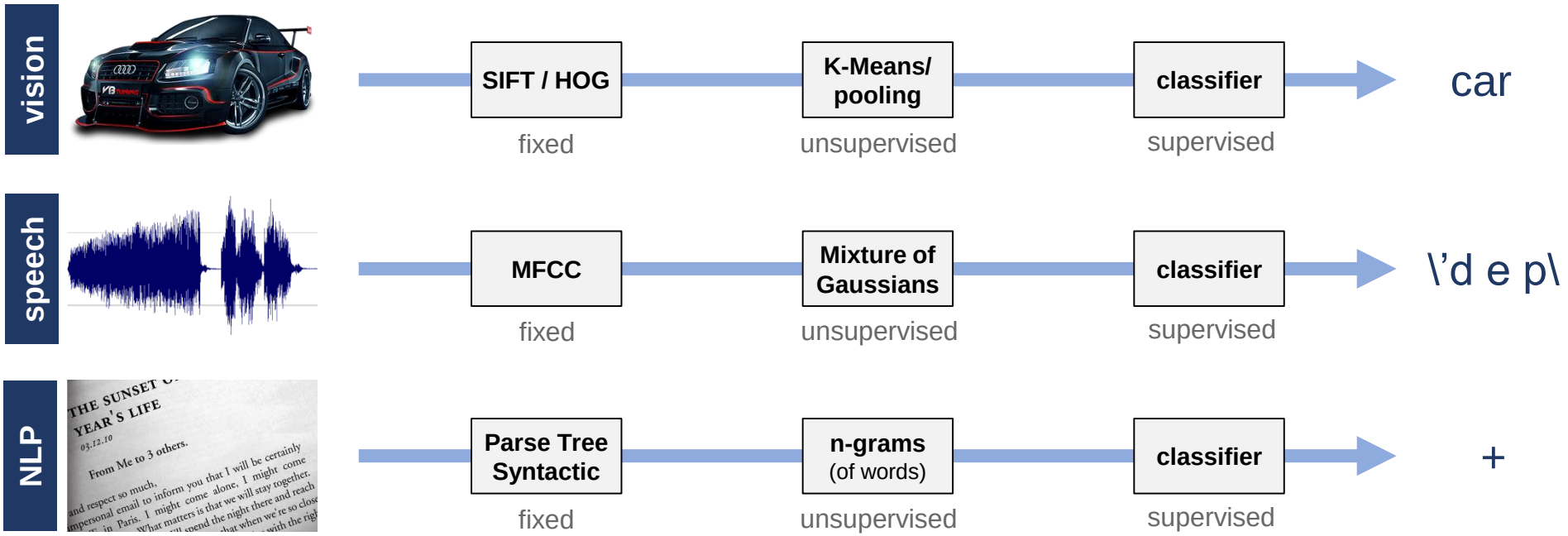
Multimedia Retrieval

Chapter 5: High-level Features with Machine Learning

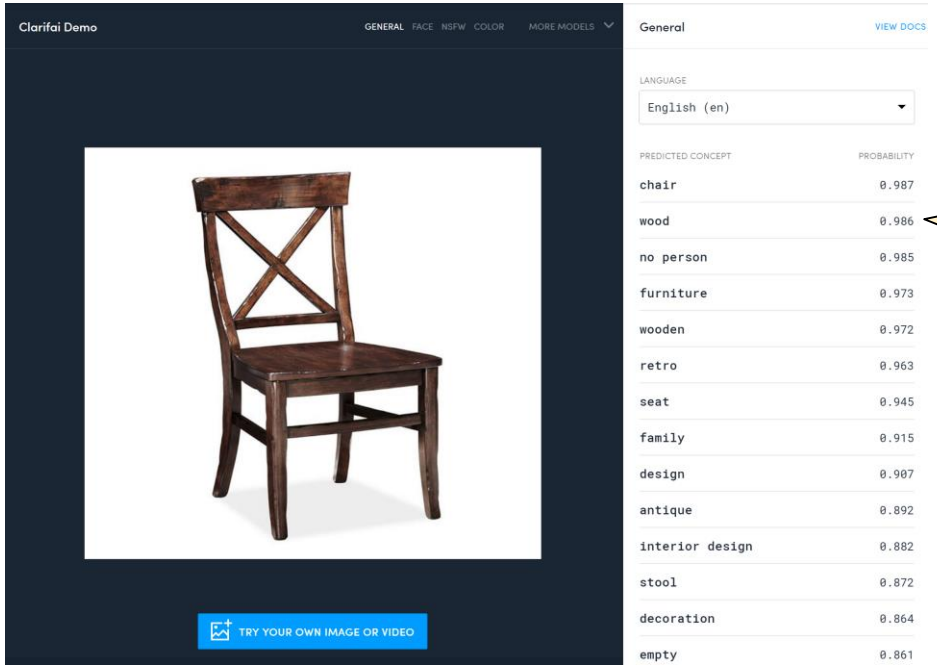
Dr. Roger Weber, roger.weber@ubs.com

5.1 Motivation

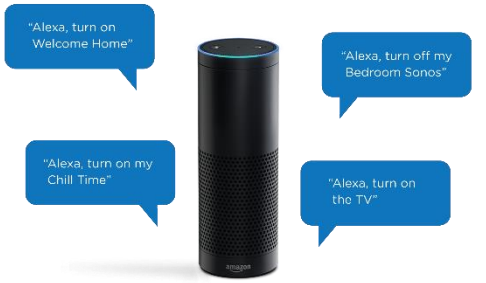
- Signal information is too low level and too noisy to allow for accurate recognition of higher-level features such as objects, genres, moods, or names. As an example, there are exceedingly many ways how a chair can be depicted in an image based on raw pixel information. Learning all combinations of pixels or pixel distributions is not a reasonable approach (also consider clipped chairs due to other objects in front of them).
- Feature extraction based on machine learning abstracts lower level signal information in a series of transformations and learning steps as depicted below. The key ingredient of a learning approach is to eliminate noise, scale, and distortion through robust intermediate features and then cascade one or many learning algorithms to obtain higher and higher levels of abstractions.



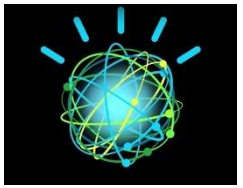
- Demo: clarifai
  - Clarifai provides APIs to recognize ‘models’ in images. Developers can use the APIs to retrieve tags from existing models or can add and train new models.
  - https://www.clarifai.com



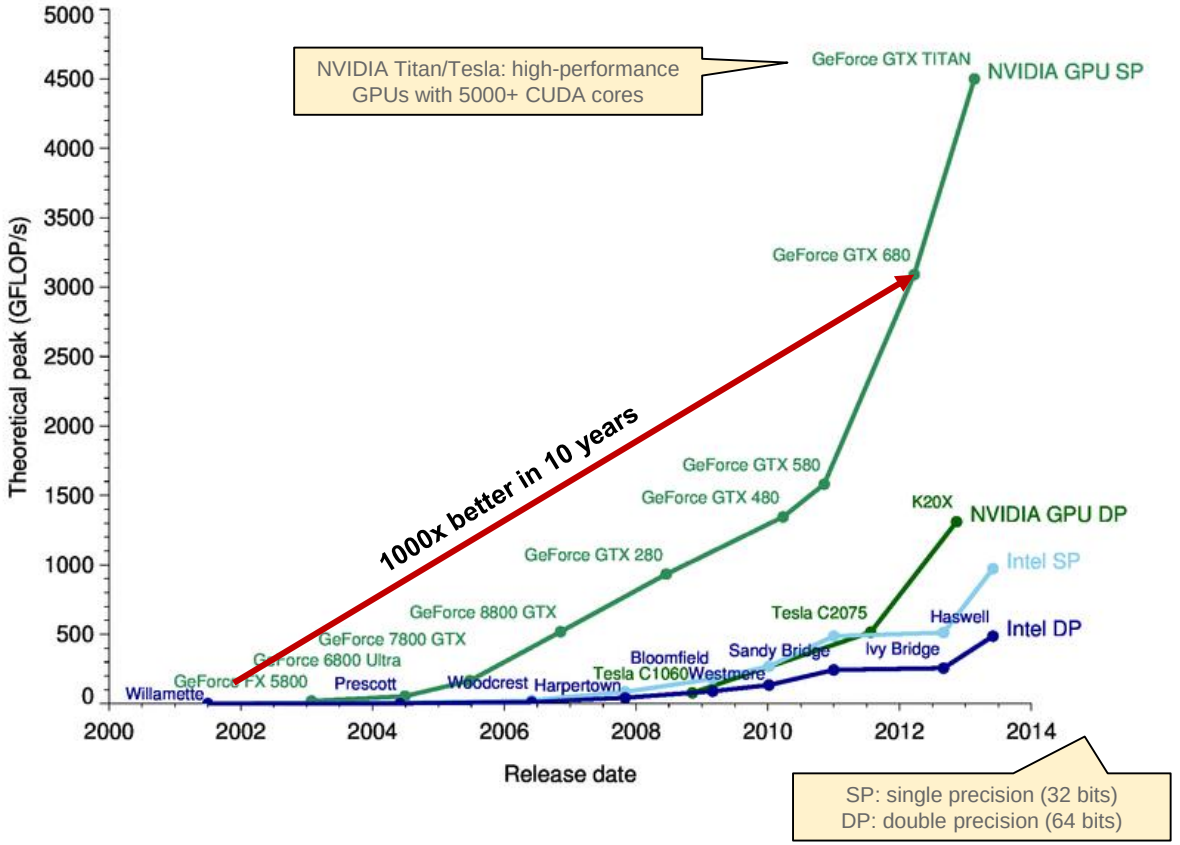
Probability that the model / concept is present in the picture



- Demo: Recognition of handwriting
- Demo: Speech Recognition

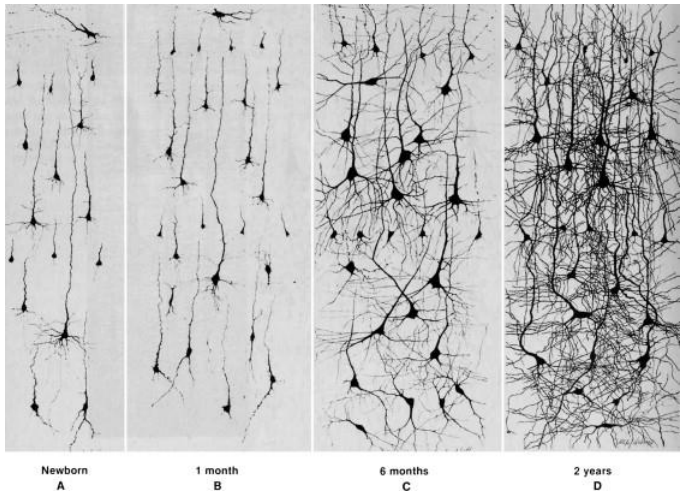


- Machine learning has greatly improved over the past years because of three factors:
  - Deep learning has introduced new layers and methods that removed the limitations of (linear) multi-layer networks.
  - CPUs and especially GPUs have allowed for much deeper and larger networks. What took months in the 90s can be computed within hours 20 years later
  - Availability of frameworks like Tensorflow makes it very simple to build a huge distributed network to compute large-scale neural nets.



The biggest improvement over the past ten years was the creation of CUDA, a extreme parallel computing platform created by Nvidia. In combination with new neural network algorithms and the advent of map/reduce as a generic distributed computing paradigm, enormous amounts of data became processable through the sheer brute force of 1000s of connected machines. Going forward, we will see highly specialized chips (like Google's TPUs) and cloud compute hardware (like HPEs 'The Machine') further accelerating the hunt in ever larger data lakes.

- Although not every aspect of the human brain is understood, there are a number of key insights that helped to further developed and refine deep learning. For instance:
  - It was believed that the brain adapts in the first months of a new born and does not change afterwards. This belief was disproved: next to short term and long term memory adjustments, the brain is also able to functionally change. Areas of the brain that are used more frequently become more excitable and become easier to activate. The brain can shift how and when such areas are getting activated and with that can provide more neurons for a task. It has been shown, with limitations, that different areas can take over functions after brain damages. For instance, somebody who loses eye sight with age is able to accentuate other senses and to use them as compensation of the visual information (no longer stimulating the visual cortex).



  - What does this mean? The brain is most likely working with a “universal algorithm” rather than task dedicated learning patterns. The way we learn a musical tune is similar to learn a complicated sequence of movements. Even more, it is believed that the algorithms are rather simple but given the dynamically built connections and the sizes allow for even very complicated tasks. But as you know, learning rates greatly vary between individually. While some learn patterns extremely fast, others require months and months of hard training. It is shown that we learn best with increasing difficulties and if we struggle in the practice. Every learning session will change your brain, but each one will adapt in different ways.
- Many researchers switch between neuroscience and artificial intelligence and have stimulated both areas with exchange of ideas.

5.2 Machine Learning Basics

- The Machine Learning Problem

> A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E [Mitchell 1997]

- There is a wide variety of machine learning problems as a combination of what the task is, what experience is provided and how performance is measured. Subsequently, we look at each individual component independently to categorize the different flavors of machine learning.
- Often, real-life examples employ a set of different approaches and combine them to achieve the overall objective of the problem. For instance, in credit card fraud, the first component is to learn fraudulent transaction based on past transactions and investigations. This knowledge is used to predict fraud in real-time for new transaction. A second component segments transactions to identify outliers or anomalies that may lead to new types of fraud that have not been identified/learned yet. While the first component is an example for supervised learning where the algorithms get labeled data to learn from, the second component is unsupervised, i.e., we don't know what we are looking for and the algorithm must identify the patterns without any human feedback.
- Other examples include cascading several methods: for instance, a first step reduce dimensionality and eliminates outliers (unsupervised learning), a second step learns that mapping of reduced features to a set of labels (supervised learning).
- Modern approaches in Deep Learning build excessively deep sequences with neuronal networks to apply multiple different approaches to extents that require vast amounts of compute power to train and then to use the network.

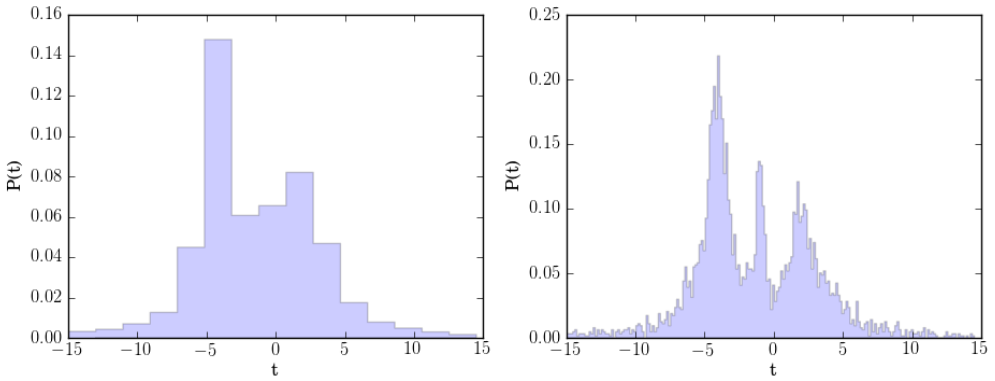
5.2.1 Tasks

- With task, we do not mean the learning process itself. Rather the ability that the machine is supposed to perform. For instance, if we want a car to drive autonomously, then driving is the task. Often, machine learning tasks involve a set of input features that the system needs to process into a "correct" set of output features.
- **Classification** is the task of mapping the input features to a set of K categories. Typically this means to find a function f that maps a M-dimensional vector $\boldsymbol{x}$ to a category represented by a numeric value y, i.e., $y = f(\boldsymbol{x})$ with $f\colon \mathbb{R}^M \rightarrow \{1, \ldots, K\}$. A variant of the classification task requires a probability distribution $P(y)$ over all classes y with $P(y) = 1$ denoting the class y is certain and $P(y) = 0$ denoting the class y is impossible, i.e., $P(y) = f(\boldsymbol{x})$ with $f\colon \mathbb{R}^M \rightarrow [0,1]^K$
  - Applications include object recognition in images, text categorization, spam filtering, handwriting and speech recognition, credit scoring, pattern recognition, and many more

Sample	fixed acidity	volatile acidity	citric acid	pH	alcohol	quality
#1	8.5	0.28	0.56	3.3	10.5	7
#2	8.1	0.56	0.28	3.11	9.3	5
#3	7.4	0.59	0.08	3.38	9	4
#4	7.9	0.32	0.51	3.04	9.2	6
#5	8.9	0.22	0.48	3.39	9.4	6

- **Classification with missing input** is similar to classification with the exception that some input values can be missing. Instead of a single function f, a set of functions is needed to map different subsets of inputs to a category y (or distribution $P(y)$), potentially 2^M functions. A better way is to learn the probability distributions over all relevant features and to marginalize out the missing ones. All tasks have a generalization with missing inputs.

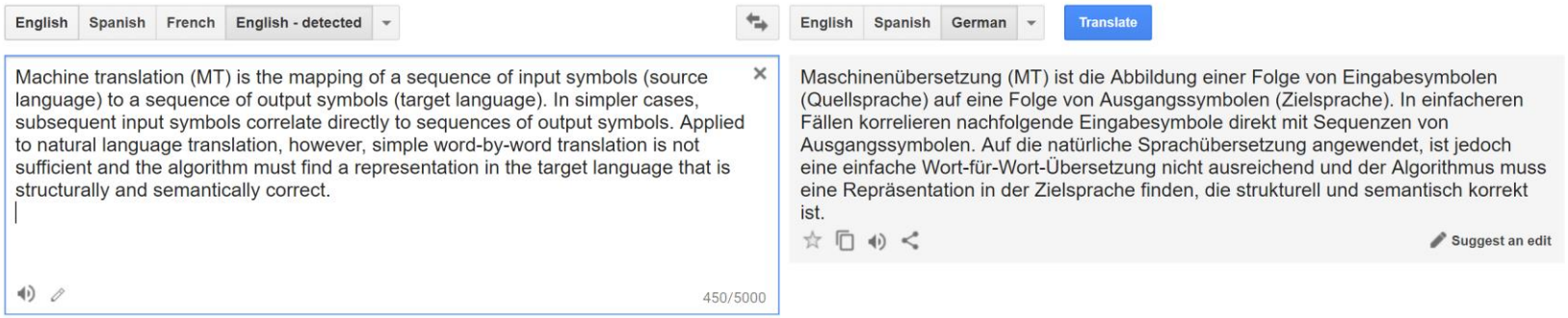
- **Regression** is the task of predicting a numerical value given the input features. The learning algorithm must find a function f that maps a M-dimensional vector $\boldsymbol{x}$ to a numeric value, i.e. $f: \mathbb{R}^M \rightarrow \mathbb{R}$. The difference to classification is the output: instead of a category, a real number is required. Also, regression does not deliver distribution functions over all possible values.
  - Applications: predictions / extrapolations to the future, statistical analysis, algorithmic trading, expected claim (insurance), risk assessment (financial), cost restrictions, budgeting, data mining, pricing (and impact on sales), correlation analysis
- **Clustering** divides a set of inputs into groups. Unlike in classification, the groups (and the number of groups) are not known beforehand and the machine learning algorithm must find them. As the output is not known at training time, this type of task is called “unsupervised” while the ones before are “supervised” (we tell the machine what outputs we expect).
  - Applications: human genetic clustering, market segmentation (groups of customers), social network analysis (communities), image segmentation, anomaly detection, crime analysis
- **Density estimation (probability mass function estimation)** is the construction of an estimate of an underlying, unknown probability density function given the input features. In the most simple case, the algorithm must learn a function p: $\mathbb{R}^M \rightarrow \mathbb{R}$ where $p(x)$ is interpreted as a probability density function (if $\boldsymbol{x}$ is discrete p is called probability mass function). The most basic form is shown in the example on the right with histogram based density estimation using two different numbers of bins.
  - Applications: age at death for countries, modelling of complex patterns, feature extraction, simplification of models



- **Imputation of missing values** requires an algorithm to replace (estimate / guess) missing data with substituted values. For a new example $\boldsymbol{x} \in \mathbb{R}^M$ with some missing x_j, the algorithm must provide a prediction for the missing values.
  - Applications: incomplete sensing data, demographics (incomplete data over person), medical analysis (incomplete or expensive test data), restoration of signal (after data loss)
- **Synthesis and sampling** is a type of task where the machine learning algorithm must generate new examples that are similar to the training data. In video games, for example, large portions of the immersive landscape are generated automatically instead of by hand. This also requires some sort of variance in the output to break "dull" patterns that are easily recognized as artificial landscape (see example on the right side). Other examples include speech synthesis where a written text is emitted as an audio waveform for the spoken version of the text. The challenge for the algorithm is the lack of a "correct answer" and the necessity to include large quantities of variation in the output.

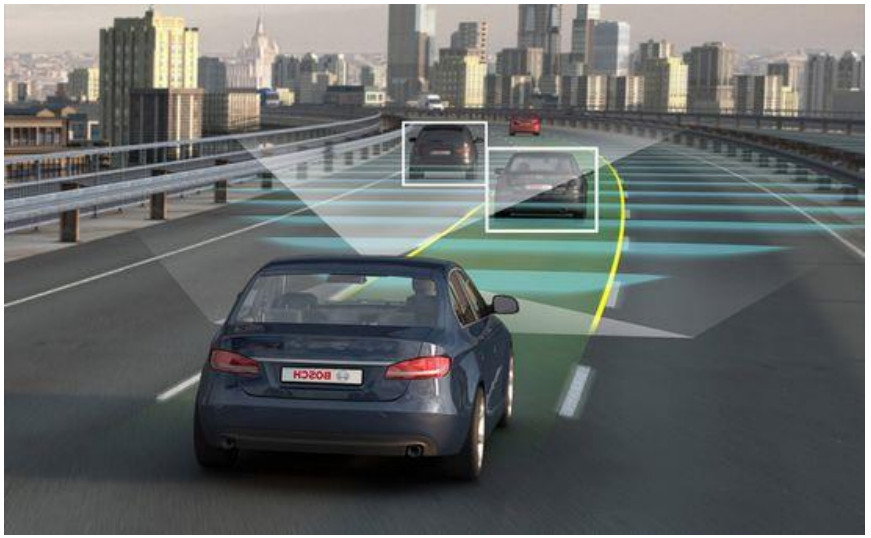
- **Anomaly detection** requires the algorithm to flag unusual, incorrect, or atypical events or data points. The output can be a simple $\{0,1\}$ flag (1 indicating an anomaly) or a probability for an anomaly. Supervised anomaly detection needs a training set with labels "normal (0)" and "abnormal (1)". Unsupervised anomaly detection requires the algorithm to describe the normal behavior (e.g., using density estimation) and to detect outliers automatically.
  - Applications: credit card fraud, intrusion detection (cyber security), outliers to improve statistics, change detection, system health monitoring, event detection, fault detection

- **Machine translation (MT)** is the mapping of a sequence of input symbols (source language) to a sequence of output symbols (target language). In simpler cases, subsequent input symbols correlate directly to sequences of output symbols. Applied to natural language translation, however, simple word-by-word translation is not sufficient and the algorithm must find a representation in the target language that is structurally and semantically correct.
  - Google Translate



- **Transcription** asks a machine learning algorithm to observe a unstructured representation of the data an to transcribe it into a discrete (often textual) form. The most widely known versions are optical character recognition (OCR) and speech recognition.
- **Dimensionality Reduction** simplifies the input vectors to a lower-dimensional space. In many cases, the output is interpreted as topics or concepts that are key to disseminate the input vectors as good as possible (topic modelling). This allows the machine to more easily find documents that cover similar topics, i.e., instead of considering hundred thousands of different terms (words), only a few topics are considered. Dimensionality reduction is often used to reduce the amount of input data but to keep as much of the core information as possible.
  - Application: data mining, latent semantic analysis, principal component analysis, statistical analysis, data reduction/compression

- **Reasoning** is the process of generating conclusions from knowledge using logical techniques such as deduction and induction. Knowledge-based systems have been used over the past 30 years including expert-system written in prolog. Facts and rules were used to prove (or disprove) a new statement within a closed world. Newer approaches use machine learning to prove theorems or constraint solvers. Cognitive reasoning and cognitive AI have recently boosted performance of chat bots and speech recognition.
- **Autonomous Robots** work with reinforcement learning, i.e., it is not possible to provide samples that connect input signals with correct or expected output signals. Rather, robots need to adjust their behavior based on incentives and penalties provided by the environment. The rise of autonomous driving has created an entire new set of challenges on reinforcement learning: machine ethics. While this sounds like science fiction, there are many scenarios where robots must make decisions that programmers cannot foresee or hard code. As an example, if the car is inevitably hitting an animal or a person on the street, should the machine try a risky evasive move endangering its passengers or accept the potential death of the animal or person (including potential damages)
  - While the field is relatively young, recent progress was accelerated by deep learning techniques. Tesla states that its autopilot is 10 times safer than the average driver.
  - Laws for and acceptance of robots in society are in its infancy. People are still worried about safety and mostly the fact of having the car hacked
  - Further obstacles are insurance issues (who pays for a mistake of a robot)

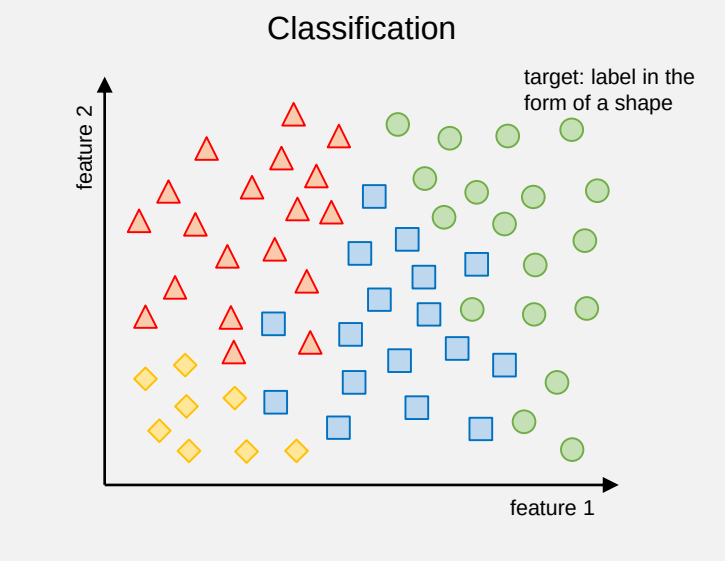


5.2.2 Performance

- To evaluate (and improve) a machine learning algorithm, we need to provide a quantitative measure for the "accuracy" of carrying out the task T. We already looked at different type of performance measures in Chapter 1 (Evaluation of Performance). A short summary:
  - **Binary classification** (0-1 decisions) uses a **confusion matrix** to assess the performance, and provides numeric summary values to optimize for a desired optimum for the task. Typical measures include precision, accuracy and so on.
  - **Multi-class classification** (one out of a set of classes) requires a generalized **confusion matrix** resulting in a table with pair-wise "confusion". Accuracy still works fine; in addition, we can summarize performance of a single class against all other classes.
  - **Binary classification with scores and thresholds** is a simple extension of the confusion matrix. With increasing threshold values, we obtain a method to optimize the threshold (adjustment of a hyper-parameter), and the Receiver Operating Characteristic Curve (ROC Curve). The area under the ROC curve is a simple method to assess performance.
  - **Multi-class Classification with Probabilities** measures the performance based on the probabilities on the class labels of an object. Typically, this is based on cross-entropy with the log-loss measure being a simpler version of it.
  - With **Regression** tasks, we measure the performance as the **mean squared error (MSE)** between the actual values and the predicted ones.
  - As we will see, machine learning algorithms not only use these measures to evaluate performance but also employ them to find an optimal set of parameters to minimize the error/loss function. In addition, it can also be used to control so-called hyper-parameters (as we see later).

5.2.3 Experience

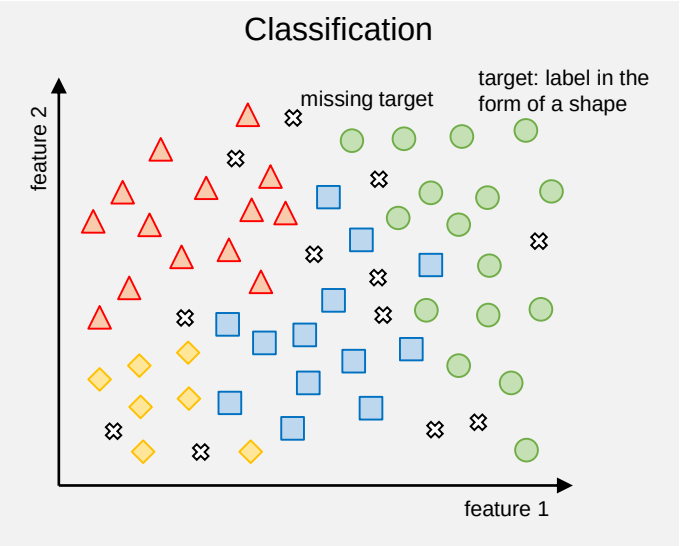
- **Supervised Learning** algorithms observe a data set with features and a target for each instance of the data set. The goal is to learn a general rule that maps features to targets and that can be applied to predict the outcome of newly presented data items. The term “supervised” originates from the view that the target is provided by an instructor or teacher. As an example, classification tasks presents for each example, described as a set of feature, a target in the form of a label (or set of labels). The “teacher” instructs the algorithm how sets of features are correctly mapped to labels and the algorithm should learn the mapping rule.



  - As discussed in the “Performance” section, the teacher also provides an error measure that enables the machine learning algorithm to assess accuracy during training sessions.
  - Even though targets are given, the algorithm must be able to deal with noise in the output values due to human errors (wrong labelling) or sensor errors (defects, distortion)

- **Semi-Supervised Learning** is a special case of supervised learning. The algorithm is presented with features and targets, however, some features or targets are missing (incomplete observation) in the training data. Depending on the task, the algorithm must either complete the missing features or predict targets for newly presented data sets.
  - **Missing targets:** The training set consists of complete features but some objects do not have targets (or labels). Incomplete targets often result if the labeling process is expensive or labor intensive. Consider a data set for credit card fraud detection with billions of transactions. Naturally, credit card firms investigate only a small subset of “suspicious” transactions and label them based on the outcome of an investigation (“fraud”, “no fraud”). The vast amount is not labeled. To learn from such data sets, algorithms make one of the following assumptions:
    1) Smoothness: points in close proximity share the same label, i.e., the distribution function is continuous
    2) Cluster: data tends to form clusters and all objects in the same cluster share the same label
    3) Manifold: often, features are high-dimensional but there are only a few labels. Hence, the data is more likely to lie on a low dimensional manifold.

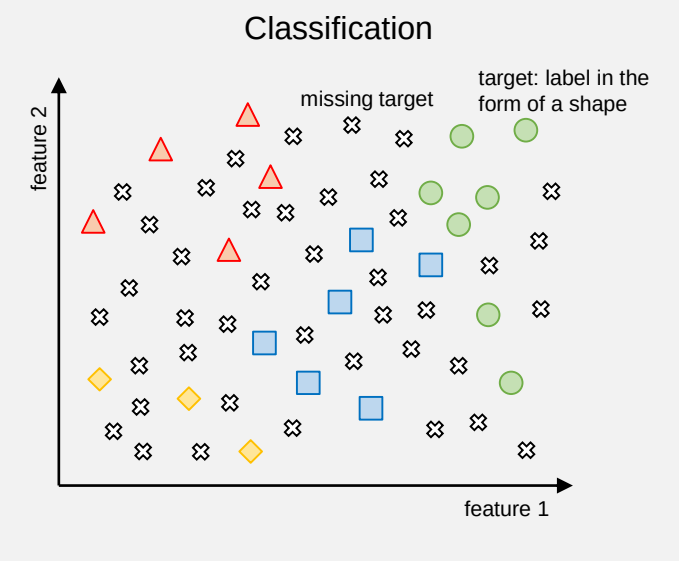
    Semi-supervised learning takes ideas both from supervised learning and from unsupervised learning.



- **Induction:** if only a few labels are missing, a good strategy is to learn the distribution from the labeled data items with a supervised learning method. We can then go back and predict the missing labels. However, this does not work well if most objects have no label as the training set is not sufficient to capture the true distribution of labels. Evidently, such training ignores most of the data (information loss).

- **Transduction:** to consider all data points, transductive algorithms identify clusters in the data set and apply the same label to all objects in the cluster. A simple approach is the partitioning transduction:
  1. Start with a single cluster with all objects
  2. While a cluster has two objects with different labels
     Partition the cluster to resolve the conflict
  3. For all clusters
     Assign the same label to all objects in the cluster

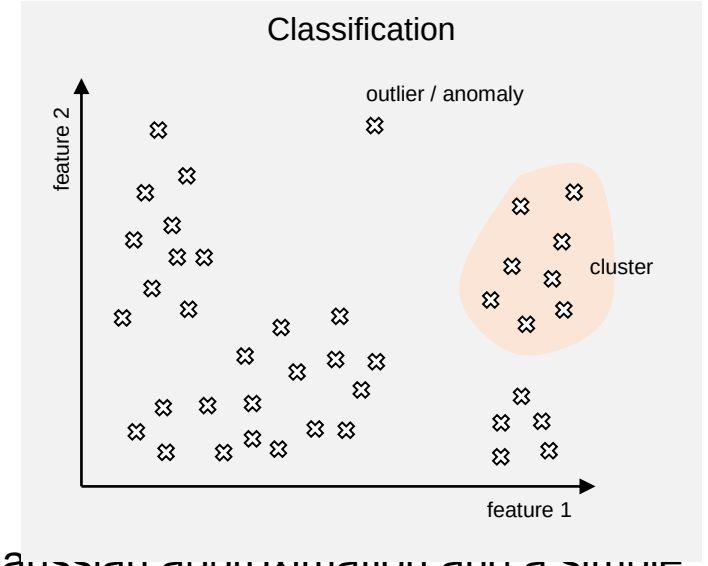
  There are other variants to develop the clusters.



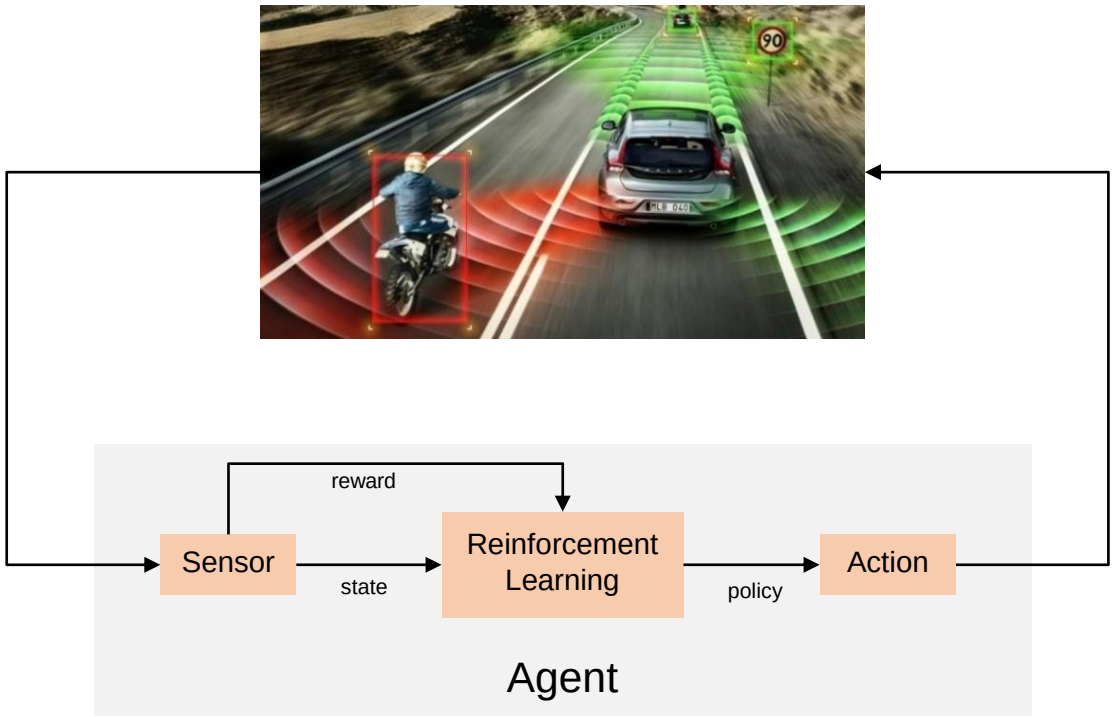
– **Missing features:** The training set has complete targets, but some objects lack some of the features. For newly presented data, potentially with missing features, the algorithm must predict the target. A good example is disease prediction where the target ("healthy", "has disease") must be predicted from a set of test results. Laboratory tests are expensive so naturally not all features are available. The approach to do so depends on the selected method:

- Naïve Bayes (more details later in the deck) is a simple technique for constructing classifiers based on conditional probabilities. Let there be K classes C_k and M features x_i. The best class k^* is then given by $k^* = \underset{k}{\operatorname{argmax}} P(C_k) \prod_i P(x_i|C_k)$. The probabilities $P(C_k)$ and $P(x_i|C_k)$ are learned from the training data (ignoring missing features x_i). To predict the class for a new object with missing features, we simply ignore them in the Naïve Bayes optimization.
- If we have learned the distribution function over all features, we can simply "integrate" or "average" over the missing features, i.e., we assume that the missing features follow the distribution of the training set and we approximate them with an expected value.

- **Unsupervised Learning** algorithms observe a data set without targets and infer a function that captures the inherent structure and/or distribution of the data. In other words, we want to identify interesting facts in the data and derive new knowledge about its structure. In contrast to supervised learning, there is no instructor or teacher that provides targets or assess the performance of the outcome. The algorithm must learn without any guidance.
  - Clustering: the most common task for unsupervised learning is to identify groups of objects that "belong" together (with regard to a distance function). The number of clusters is often not known and must be learned too.
  - Outlier/Anomaly detection: the algorithm must learn the "normal" behavior through any means and identify outliers that significantly differ from the other objects. Note that the training data may also contain outliers.
  - Density function: describe the data set through an "appropriate" density function. A simple case would be a Gaussian approximation and a simple learning of the mean value and the variance. More complex cases may choose from a set of different distribution functions and optimize to the "best fit"
  - Dimensionality reduction: high-dimensional features often disguise an inherent simple characteristic of the data. Principle component analysis extracts "core concepts" along principal directions in the feature space that provide a simpler (but still accurate) view on the data.
  - Self-organizing maps (SOM): a SOM produces a discrete (often 2-dimensional) presentation of the data in a mesh of nodes, thereby mapping high-dimensional data to a low-dimensional view. It uses a competitive learning approach.



- **Reinforcement learning** evaluates possible actions in an environment so as to maximize the cumulative award. The problem is very general and broad and studied in various fields such as game theory, control theory, operations research, simulations, and genetic algorithms. Reinforcement learning is different to supervised learning as correct input/output correlations are not known. The focus is on finding a balance between exploration (of unknown situations) and exploitation (of current knowledge).
  - A reinforcement agent typically interacts with its environment in discrete time steps. At each time t, the machine observes the environment including potential rewards. It then chooses and action from the set of available actions and performs it against the environment receiving rewards for the transition. The objective is to maximize the cumulative rewards.



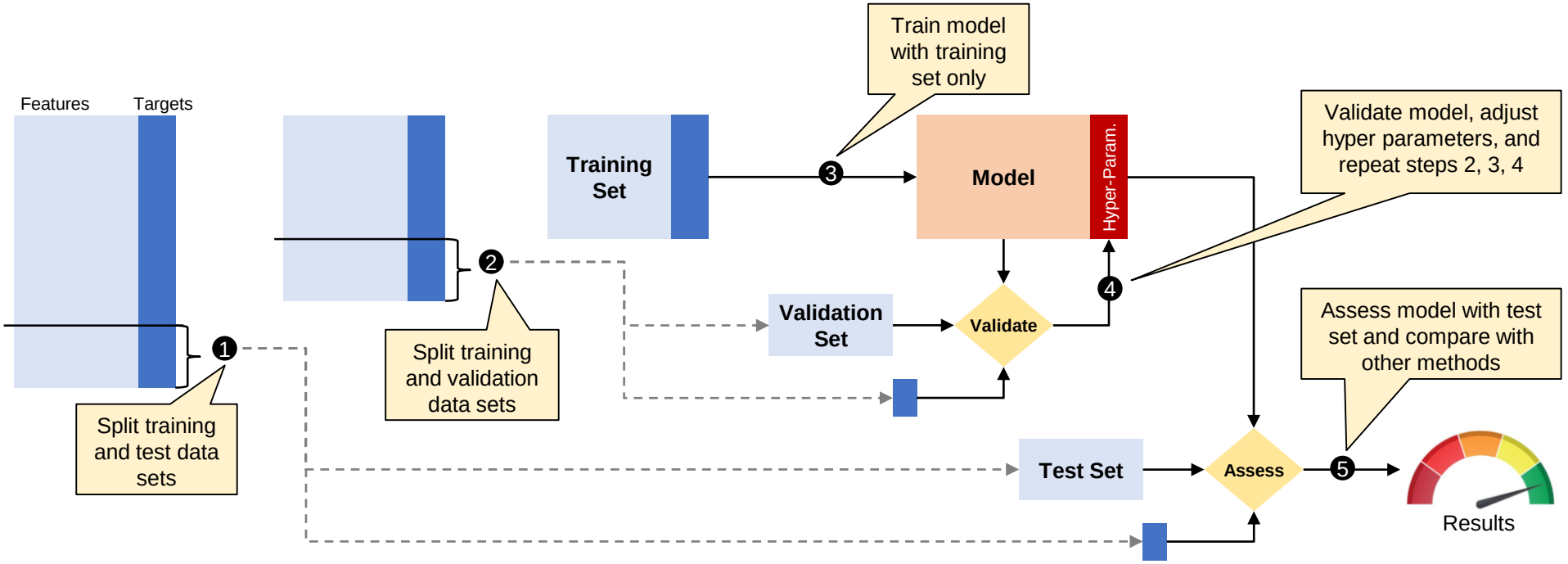
- A policy is a series of actions. Instead of optimizing for individual actions, reinforcement learning algorithms define policies and choose the best policy for immediate and cumulative rewards. Exploration is the process of developing (or composing) new policies, while exploitation is the application of the best known policy. Exploration can lead to algorithms that are no longer understood by the human developers. AlphaGo, Google's Go program that has beaten the world champion, can not be explained anymore, i.e., it is not clear how the computer decides and what the winning strategy is.
- Reinforcement learning is an efficient approach if the environment behaves non-deterministic or even chaotic due to incomplete or erroneous observations. It is the only viable option if we lack an accurate error (or success) measure. Driving autonomously in a city is a good example for the chaotic and non-deterministic nature of such tasks. Though it is possible to describe broadly what success means ("arrive safely at the target within n minutes"), it is not possible to provide accurate measures at every point in time.

5.3 The Learning Proces

- Machine learning algorithm learn from data. It is critical that we feed the “right” data into this process for the task that we want it to solve. “Right” is not only referring to good data quality, complete data, but also the extraction of meaningful features. A number of challenges arises in this context:
  - Feature selection, i.e., ability to capture essential information to learn a task
  - Data cleansing, i.e., ability to remove the negative impact of outliers or of noise
  - Normalization, i.e., ability to address correlation between features and to normalize scales
  - Curse of dimensionality, i.e., inability to learn underlying structure due to sparse data space
  - Overfitting, i.e., inability to generalize well from training data to new data sets
  - Underfitting, i.e., inability of the algorithm to capture the true data structure

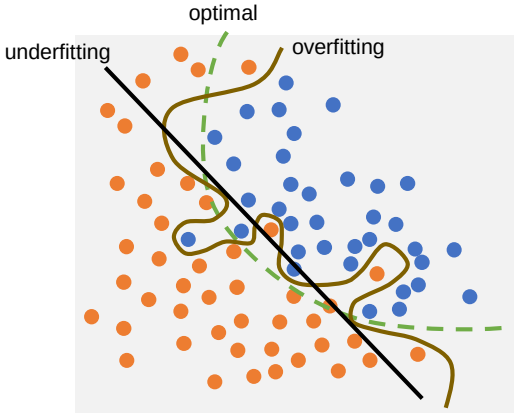
- Data preparation is a 3-step approach which we do not further discuss in this section. With data we always include features and targets (if they are available)
  1) Select Data
  2) Preprocess Data
  3) Transform Data

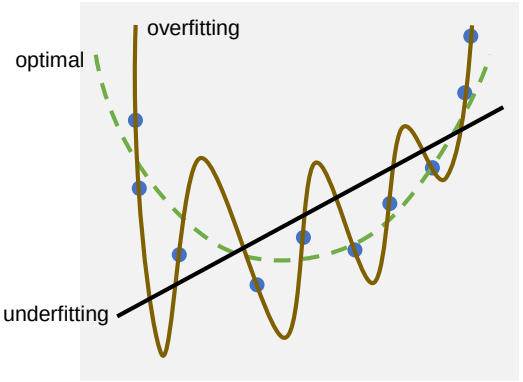
- We need to pay attention how we divide the data sets into training sets, validations sets, and test sets. The latter aspects is essential to adjust hyper-parameter of the algorithm including capacity and to measure its ability to correctly generalize. In the following, we focus on the overall learning process and address the above overfitting and underfitting issues.



- To understand how well a machine learning algorithms can generalize to new data sets, it is essential that training sets and test sets are distinct. Otherwise, we can construct a memorizing algorithm that simply stores all features and targets. Assessments of such an algorithm will produce the best possible results, but the algorithm will perform poorly on new data.
- Most algorithms have models with so-called hyper parameters that drive their inherent **capacity** or structure. For example, we can vary the degree of a polynomial regression model to adjust to a larger variety of functions. In a neural network, the capacity is provided by the number of neurons and connections. In a nutshell, models with small capacity struggle to fit the training data and to capture its distribution; models with high capacity tend to overfit the training data and poorly generalize to new data sets. The usage of validation sets (again, distinct from the training sets) allows algorithms to optimize their hyper-parameters.

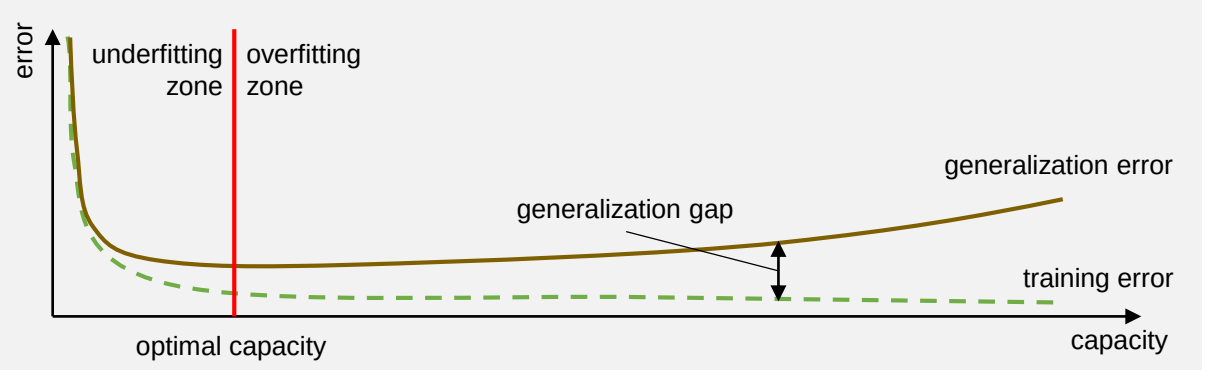
- **Overfitting** and **underfitting** are common problems in machine learning. Overfitting occurs when the model is excessively complex to match the training data as accurately as possible. Often, such a model has too many parameters relative to the number of training items. But even worse, the model is likely to overreact to minor changes leading to poor predictive performance (see figure on the right hand side as an example). Underfitting, on the other side, occurs when the model cannot capture the underlying trend of data and over-simplifies the distribution. For instance, fitting a linear model to a non-linear data distribution will result in a high training error and poor predictive performance.





  - As illustrated above, we can observe that overfitting is the result of optimizing for the training data with too many parameters. With MSE, an overfitting model shows small errors indicating its ability to adapt nicely to the training data, but it can not predict new data points well enough.
  - Underfitting, on the other side, shows both large errors on the training data and poor prediction performance for new data points; it obviously cannot capture the true essence of the distribution.
  - We can control overfitting and underfitting by altering the **capacity of the model**. Optimal capacity is reached if the model exhibits small errors on both the training set and the validation set. To work best, training set and validation set must be distinct; but we can run several iterations to adjust the capacity with different partitioning of training and validation set.

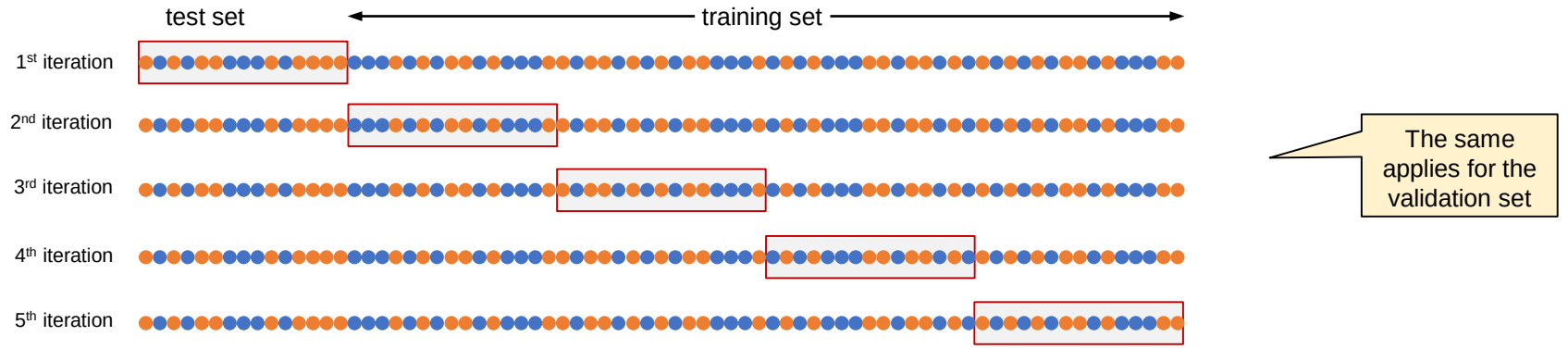
- When altering the capacity of the model, **Occam's razor** provides an intuitive heuristic. The principle was first stated by William of Ockham (c. 1287-1347) and has been made more precise over time, most notably in the 20th century for statistical learning. The principle states:
  - Numquam ponenda est pluralitas sine necessitate [Plurality must never be posited without necessity]
  - In a more modern language, the principle states that among competing hypothesis that explain observations equally well, one should choose the "simplest" one.
  - Indeed, simpler models are better able to generalize but we must choose a sufficiently complex model to achieve low training error. Typically, training error decreases gradually as capacity increases. The generalization error, however, has a U-shaped curve as a function of capacity:



- The **bias-variance tradeoff** (or **dilemma**) is the problem of simultaneously minimizing two sources of errors that prevent models to generalize well beyond their training data
  - The bias is the error of a model causing it to miss relevant relations in the data set (underfitting)
  - The variance is the error from sensitivity to small changes in the input. High variance can cause the model to adopt to noise in the training data rather than to the data (overfitting)

  The **bias-variance decomposition** is a way to analyze the expected generalization error. It uses the sum of the bias, variance, and irreducible error (noise) in the problem.

- To drive the learning process, we partition the original data set (and its targets) into a training set (70-80% of data) and test set (20-30% of data). If the model has need to optimize some hyper-parameters, we further partition the data to obtain the validation set (20-30% of data):
  - The **training set** is used for learning, i.e., to fit the parameters/weights minimizing training error
  - The **validation set** is used to tune hyperparameters (models, capacity) to prevent underfitting and overfitting issues. Validation data is not used for training and also not used for final testing
  - The **test set** is used to assess the performance, i.e., the ability of the model to generalize
- Ideally, the three data sets are large enough to represent the true distribution equally well. If the data set is too small, however, validation and testing lack statistical certainty on average errors making it difficult to assess and compare performance. **Cross-validation** uses rotation schemes an multiple iterations to improve the accuracy of validation and testing.
  - **k-fold cross validation** partitions the original data set into k equal sized subsamples. In each iteration, one subsample denotes the test set, and the remaining k-1 subsample form the training set. The k results are averaged to produce a single value. k=10 is a typical value. The same approach can be used for the validation set.



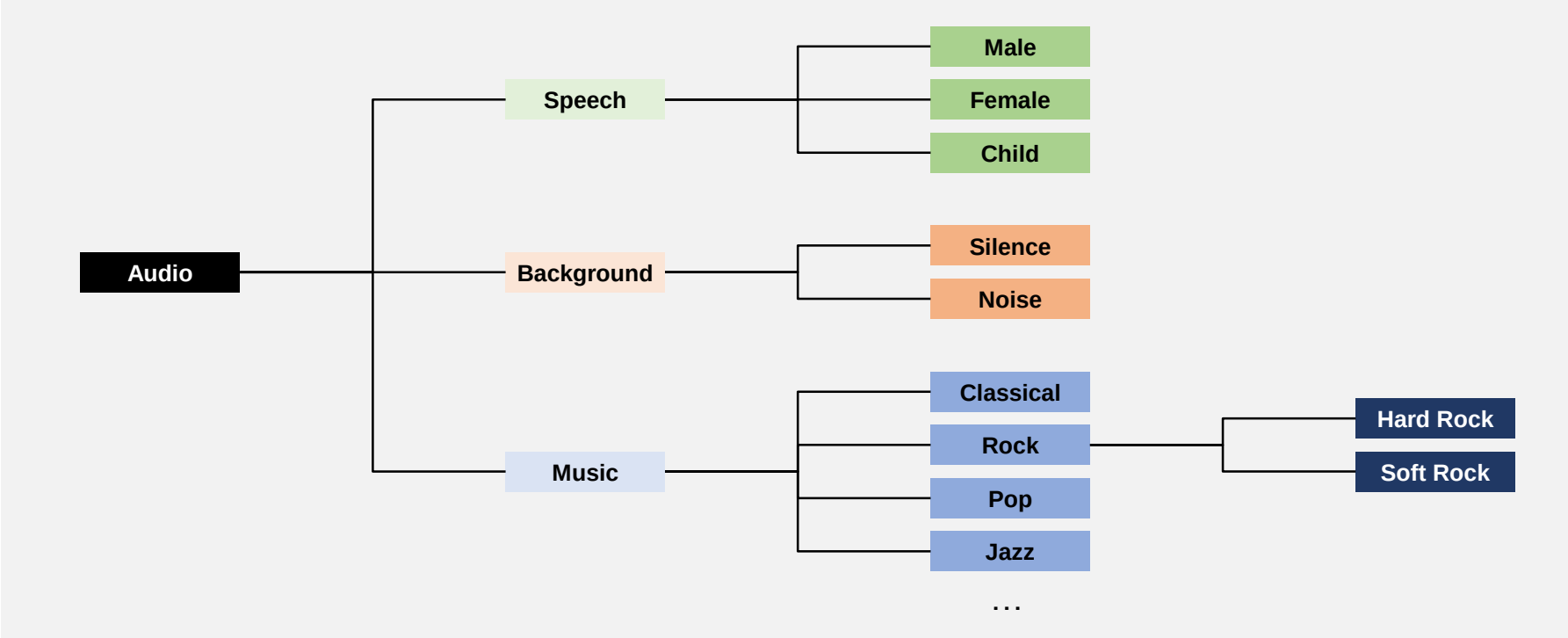
5.4 Methods

- Classification of Tasks (based on Input)
  - Unsupervised
  - Supervised
  - Semi-Supervised
  - Reinforcement Learning
- Classification of Tasks (based on Output)
  - Regression
  - Classification
  - Clustering
  - Density distribution of a distribution
  - Topic Modelling / Dimensionality reduction

- Approaches considered in the following
  - Decision Trees (ID3, C4.5)
  - Naïve Bayes
  - Unsupervised Clustering (k-means / Expectation Maximization)
  - Multi-layer Network
  - Deep Learning

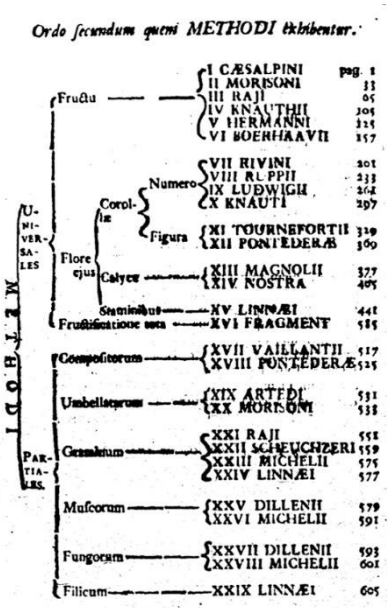
5.4.1 Decision Tree Learning

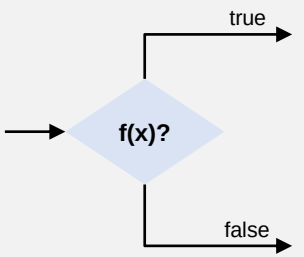
- Classification is a key concept to obtain higher-level features. The usual approach is to extract low-level features from the signal, normalize and transform the features, and deduce a mapping to pre-defined categories. Let us consider an audio database with a simple classification as follows:



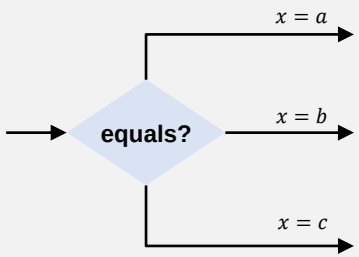
- Decision tree learning is a simple but effective classification approach. We start with a data set that has discrete and continuous features and given labels (targets for objects), and then create the “optimal” decision hierarchy to map the features with a series of tests to their labels. The resulting classification tree is easy understandable by humans and machines and can create efficient rules for classification, i.e., predicting the class with a minimal number of tests.

- The concept of classification trees is quite old. An early example is the classification scheme of Carl Linnaeus (1735) for plants (see right hand figure) and animals. Each node represents a test and each branch to the right denotes a possible outcome of the test. Leaf nodes, finally, contain the class labels. The tree does not have to be balanced and different numbers of tests may be required to reach a leaf node.
- A node in a classification tree usually tests for a single feature only. If the feature is discrete (a set of values), a node partitions the values into distinct sets (or just individual values) each with a separate branch out. The test in the node checks which partition includes the feature value. If the feature is continuous, the branches are given by distinct ranges in the feature domain. Features can be multi-dimensional but it is more common to treat each dimension as an individual ("orthogonal") feature achieved through dimensionality reduction. A special case is the binary test node which yields "true" if a condition on the feature is met and otherwise no. In many cases, nodes branch always into exactly two children (binary decision trees) but actually any number of branches is possible. Examples:

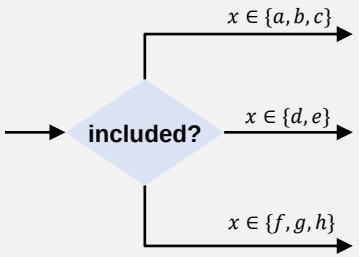




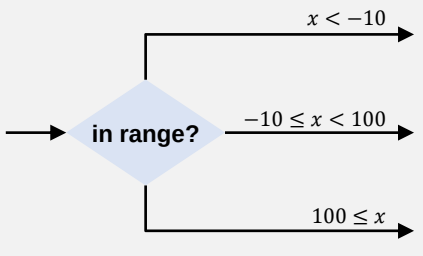
binary test
(f(x) is the condition)



discrete values

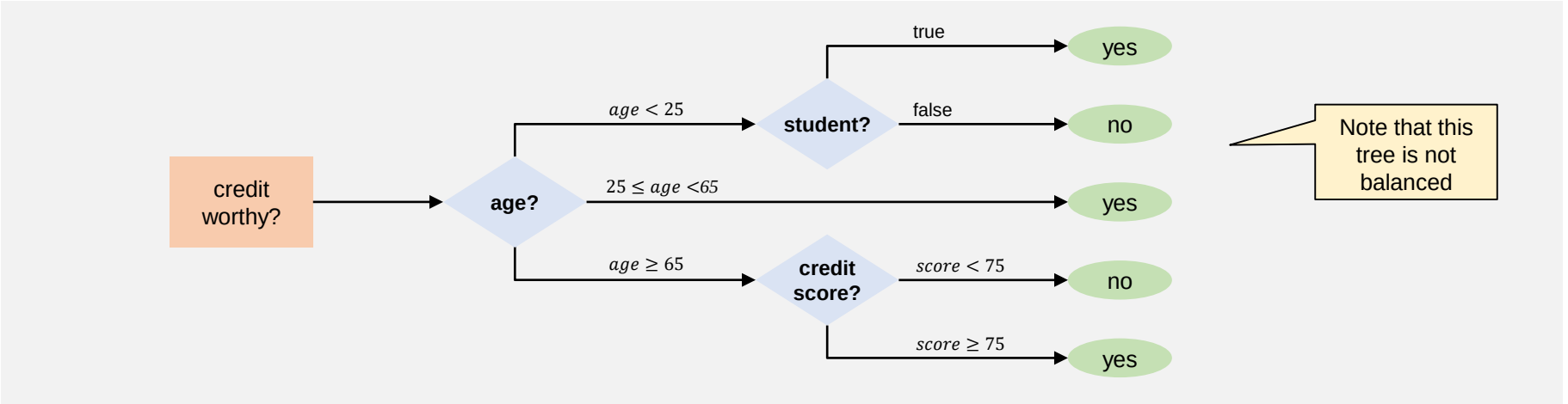


discrete sets



continuous values

- The leaf nodes denote the labels (or targets) associated with the objects. The series of test should deterministically lead to a leaf node and thus the label. Example:



- In order to create a decision tree, the machine learning approach must identify a set of tests against the features of the training data sets that lead to the observed labels with a minimal number of steps. Once the tree is learned, we can follow the decision hierarchy for a new data instance until a node is reached. The label in the node is our prediction for that new data instance.
  - Note: the condition “minimal number of steps” leads to the most simple tree that maps features to labels following Occam’s razor (i.e., prefer simple solutions over complex ones)

- To construct decision trees, we will use a fundamental concept from information theory: **information gain**. In a nutshell, the information gain is the reduction of entropy given the observation that a random variable has a given value. With this in mind, we build test nodes in the decision tree such that they maximize the information gain, i.e., choose a feature and conditions on it that reduces the uncertainty of the outcome (here: label) as much as possible.
  - Let $\mathbb{T}$ be the training set of the form $(\boldsymbol{x}, y) = (x_1, x_2, x_3, \ldots, x_M, y)$ where x_j is the j-th feature value with values from $\mathbb{V}_j$ and y the target label. The expected information gain is then a function of entropy H. Let $\mathbb{T}_{j,v} = \{\boldsymbol{x} \in \mathbb{T} \mid x_j = v\}$ be the subset of $\mathbb{T}$ such that all elements have $x_j = v$:

$$IG(\mathbb{T}, x_j) = H(\mathbb{T}) - \sum_{v \in \mathbb{V}_j} \frac{|\mathbb{T}_{j,v}|}{|\mathbb{T}|} H(\mathbb{T}_{j,v})$$

Entropy H is defined on the probabilities of the target labels y_i. $P(y_i)$ denotes the probability that a randomly selected item from $\mathbb{T}$ has the label $y = y_i$. We can estimate these probabilities through simple counting of labels in the training set.

$$H(\mathbb{T}) = -\sum_i P(y_i) \cdot \log_2(P(y_i))$$

$$H(\mathbb{T}_{j,v}) = -\sum_i P(y_i|x_j = v) \cdot \log_2\left(P(y_i|x_j = v)\right)$$

Entropy is usually based on $\log_2$ but for the purposes here, the basis of the logarithm is irrelevant

In summary, the idea of information gain is to measure whether the entropy (uncertainty about the distribution of the target labels) would decrease if we split the data set along the feature x_j

– Example: consider the table on the right hand side. There are four features x_j in the first columns and a target y ("can we play tennis?") in the last column. Let us compute the information gain if we choose $j = Windy$. The entropy $H(\mathbb{T})$ is obtained as:

Outlook	Temp.	Humidity	Windy	Play
Sunny	Hot	High	FALSE	No
Sunny	Hot	High	TRUE	No
Overcast	Hot	High	FALSE	Yes
Rainy	Mild	High	FALSE	Yes
Rainy	Cool	Normal	FALSE	Yes
Rainy	Cool	Normal	TRUE	No
Overcast	Cool	Normal	TRUE	Yes
Sunny	Mild	High	FALSE	No
Sunny	Cool	Normal	FALSE	Yes
Rainy	Mild	Normal	FALSE	Yes
Sunny	Mild	Normal	TRUE	Yes
Overcast	Mild	High	TRUE	Yes
Overcast	Hot	Normal	FALSE	Yes
Rainy	Mild	High	TRUE	No

$$H(\mathbb{T}) = -\sum_{y \in \{Yes,No\}} P(y) \cdot \log_2\big(P(y)\big) = -\frac{9}{14} \cdot \log_2\left(\frac{9}{14}\right) - \frac{5}{14} \cdot \log_2\left(\frac{5}{14}\right) = 0.9403$$

14 entries with 9 'Yes' and 5 'No'

The entropy given the observation of $x_j = v$ for $j = Windy$ with $\mathbb{V}_{Windy} = \{TRUE, FALSE\}$ is:

6 TRUE entries with 3 'Yes' and 3 'No'

$$H\big(\mathbb{T}_{j,TRUE}\big) = -\sum_{y \in \{Yes,No\}} P\big(y|x_j = TRUE\big) \cdot \log_2\Big(P\big(y|x_j = TRUE\big)\Big) = -\frac{3}{6} \cdot \log_2\left(\frac{3}{6}\right) - \frac{3}{6} \cdot \log_2\left(\frac{3}{6}\right) = 1$$

$$H\big(\mathbb{T}_{j,FALSE}\big) = -\sum_{y \in \{Yes,No\}} P\big(y|x_j = FALSE\big) \cdot \log_2\Big(P\big(y|x_j = FALSE\big)\Big) = -\frac{6}{8} \cdot \log_2\left(\frac{6}{8}\right) - \frac{2}{8} \cdot \log_2\left(\frac{2}{8}\right) = 0.8113$$

8 FALSE entries with 6 'Yes' and 2 'No'

Leading to an information gain $IG(T, x_j)$ of:

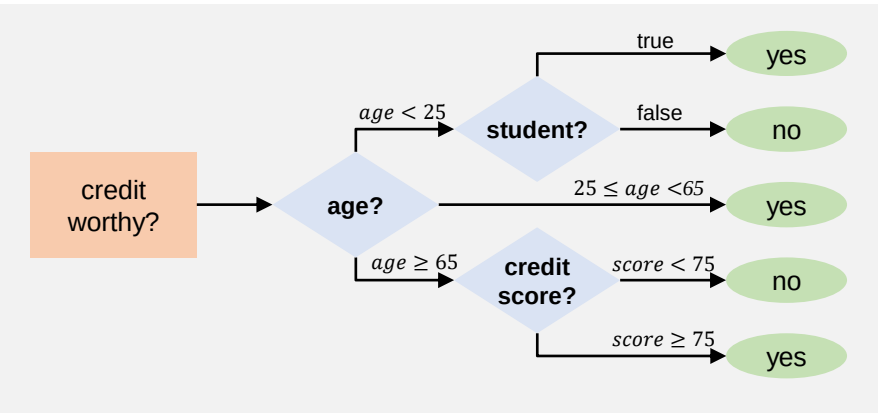
$$IG(T, x_j) = H(\mathbb{T}) - \sum_{v \in \{TRUE,FALSE\}} \frac{|\mathbb{T}_{j,v}|}{|\mathbb{T}|} H\big(\mathbb{T}_{j,v}\big) = 0.9403 - \frac{6}{14} \cdot 1 - \frac{8}{14} \cdot 0.8113 = 0.0481$$

14 entries with 6 'TRUE' and 8 'FALSE'

- A high-level pseudo code for constructing a decision tree is given as follows

```
Function DecisionTree(Features, Targets)
    TrainingSet, ValidationSet, Attributes ← CleanseData(Features, Targets)
    Root ← BuildTree(TrainingSet Attributes)
    Rules ← PruneTree(Root, ValidationSet)
    Return Rules
```

  - We can re-write a decision tree as a set of rules where each rule denotes a path from the root to a leaf with all tests along the path and the label of the leaf. Depending on the programming context, the algorithm can produce native implementations with the high computational efficiency (while traversing the decision tree is sub-optimal). For instance:



Note that 'true' and 'false' are labels and not Boolean values in the rules

```
Rule Set:
   yes ← (age<25) AND (student=='true')
   no  ← (age<25) AND (student=='false')
   yes ← (25<=age) AND (age<65)
   no  ← (age>=65) AND (score<75)
   yes ← (age>=65) AND (score>=75)
```

```
public boolean isCreditWorthy(Customer c) {
    if c.getAge()<25  && c.isStudent()           return true;
    if c.getAge()<25  && !c.isStudent()          return false;
    if 25<=c.getAge() && c.getAge()<65           return true;
    if c.getAge()>=65 && c.getCreditScore()<75   return false;
    if c.getAge()>=65 && c.getCreditScore()>=75  return true;
    return false; // default: false
}
```

rules compiler

Further optimizations of code generation possible

- **Cleanse data:** our running example from the previous page had unique rows, i.e., there are no two entries with the exact same feature values. In practice, however, there will be several observations with the same feature values, and more importantly, they may have conflicting labels. In addition, some feature values may be missing, or labels are not given. Not all features might be useful for classification. E.g., having a column “Date” in our running example would not help us to identify good rules for classification. Further transformations on the data are possible depending on the domain. This can include outlier and noise elimination, or dimensionality reduction:

```
Function CleanseData(Features, Targets)
    Features, Targets ← eliminate entries with missing Targets (=NULL) and outliers
    Features ← predict missing Features (=NULL) with domain knowledge
    Features ← transform and normalize Features with domain knowledge
    Attributes ← select set of useful Features with domain knowledge

    collapse entries that share the same Features
    assign the most frequent label from Targets to the collapsed entry
    keep Counts (=number of entries) for correct entropy calculations later on
    Data ← combine Features, Targets, and Counts into a structure

    TrainingSet, ValidationSet ← Split Data into distinct sets with given Ratio (e.g., 70:30)
    Return TrainingSet, ValidationSet, Attributes
```

  - Note: most of the data cleansing and feature selection is domain dependent. Although there are generic approaches for data preparation such as dimensionality reduction, clustering and outlier elimination, most of the manual work goes into a good feature design with the goal to have as few features as possible with minimal correlation between each other and the ability to separate target values.

Additional information — not part of the exams

- **Build tree:** Tree construction is recursive. At each iteration, a new node is inserted with a test on a selected attribute. The algorithm is called for each branch until the subset is empty or has one label

```
Function BuildTree(Data, Attributes)
    N ← new Node and associate most common label in Targets with node N
    If all Targets have same label Then Return N
    If Attributes is empty OR Data too small Then Return N
    A, Tests, Fitness ← SelectBestAttribute(Data, Attributes)
    If Fitness below Threshold Then Return N
    ForEach T in Tests Do
        B ← add new branch to node N for test T
        P ← get partition of Data which fulfills test T
        If P is empty Then add new (empty) node below branch B with same label as node N
        Else C ← BuildTree(P, Attributes – {A}); add node C below branch B
    End
    Return N

Function SelectBestAttribute(Data, Attributes)
    ForEach A in Attributes
        Tests[A], Partitions ← split feature values for attribute A and determine partitions
        Fitness[A] ← determine a fitness/score for attribute A (e.g., information gain)
    End
    Abest ← find A with Fitness[A]==max(Fitness)
    Return Abest, Tests[Abest], Fitness[Abest]
```

The typical approach is to use an attribute only once on each decision path in the tree. Hence, tree height is limited by the number of selected attributes.

  - We can observe that attributes are only used once during classification. We may relax this condition for continuous features to enable finer interval splits later in the tree.
  - The scoring function determines how well an attributes helps us to decide quickly along the paths in the decision tree. In ID3 this is the information gain as introduced before; extensions such as C4.5 balance this with the ability of the attribute to generalize.
  - We will discuss further details when we review concrete implementations like ID3 and C4.5.

Additional information — not part of the exams

- **Prune tree:** decision trees tend to overfit to the training set due to their recursive creation of nodes until no further attribute split is possible. As a consequence, generalization to new data sets may be poor. As we discussed earlier, a validation step allows a machine learning approach to compromise training errors with the ability to generalize. To do so, the pruning step eliminates tests that are not significantly improving performance against the validation set (remember Occam's razor). Pruning can also be done during building time: in `BuildTree()`, if the data set is too small or if the split along an attribute is not significant enough (fitness too small), the algorithm stops the recursion. We illustrate a few pruning techniques:
  - Elimination of branches: we assess the performance against the validation set, for instance, using accuracy (percentage of correct predictions). Then, we visit decision nodes and replace the subtree underneath them with leaf nodes if that improves overall accuracy

```
Function PruneTree(Root, ValidationSet)
    Repeat
        Accuracy ← get total accuracy for ValidationSet
        ForEach N underneath Root
            If N is leaf Then Accuracy[N]=Accuracy
            Else
                replace subtree at node N with leaf (keep label of N = most common target)
                Accuracy[N] ← get total accuracy for ValidationSet
                insert original N into the tree again
            End
        End
        N ← find node N with AccuracyNode[N]==max(AccuracyNode)
        If AccuracyNode[N]>Accuracy Then replace subtree at node N with leaf
    Until AccuracyNode[N]<=Accuracy

    Return (Rules ← create rule set given the tree underneath node Root)
```

This pseudo-code is obviously not optimized for speed but rather shows the steps that are necessary for pruning

Additional information — not part of the exams

– Pruning rules: Each rule contributes to the overall accuracy for the data items that pass through it. Initially, rules are not sorted because they are mutually exclusive (i.e., each data item can fulfill exactly one rule). The ‘pruning rules’ approach considers each condition in the rules and eliminates them if that improves overall accuracy. As a side effect, rules are no longer distinct and need to be sorted by their contribution to the overall accuracy.

This pseudo-code is obviously not optimized for speed but rather shows the steps that are necessary for pruning

```
Function PruneTree(Root, ValidationSet)
    Rules ← create rule set given the tree underneath node Root
    Repeat
        Accuracy ← sort Rules by accuracy; get total accuracy for ValidationSet
        ForEach R in Rules
            ForEach condition C in R
                remove condition C in R
                AccuracyRule[R][C] ← get total accuracy for ValidationSet
                insert condition C into R again
            End
        END
        R,C ← find rule R and condition C with AccuracyRule[R][C]==max(AccuracyRule)
        If AccuracyRule[R][C]>Accuracy Then remove condition C in R
    Until AccuracyRule[R][C]<=Accuracy

    Return (Rules ← sort Rules by accuracy)
```

Additional information — not part of the exams

- Implementations (selected examples):
  - The **ID3 algorithm** was invented by Ross Quinlan in 1986. It only worked for attributes with discrete values and used the information gain to select attributes. For each attribute x_j and each value in $\mathbb{V}_j$, the training set $\mathbb{T}$ is split into subsets $\mathbb{T}_{j,v}$ with $v \in \mathbb{V}_j$. The information gain is:

$$IG(\mathbb{T}, x_j) = H(\mathbb{T}) - \sum_{v \in \mathbb{V}_j} \frac{|\mathbb{T}_{j,v}|}{|\mathbb{T}|} H(\mathbb{T}_{j,v})$$

To compute the entropy $H(\mathbb{T})$ over the K labels y_k, we simply count the frequencies $f_k(\mathbb{T})$ of y_k in the set $\mathbb{T}$. Similarly, for the subsets $\mathbb{T}_{j,v}$, the frequencies are given by $f_k(\mathbb{T}_{j,v})$. This leads to:

$$IG(\mathbb{T}, x_j) = -\sum_{k=1}^{K} \frac{f_k(\mathbb{T})}{|\mathbb{T}|} \cdot \log_2\left(\frac{f_k(\mathbb{T})}{|\mathbb{T}|}\right) + \sum_{v \in \mathbb{V}_j} \frac{|\mathbb{T}_{j,v}|}{|\mathbb{T}|} \sum_{k=1}^{K} \frac{f_k(\mathbb{T}_{j,v})}{|\mathbb{T}_{j,v}|} \cdot \log_2\left(\frac{f_k(\mathbb{T}_{j,v})}{|\mathbb{T}_{j,v}|}\right)$$

The best attribute x_{j^*} maximizes the information gain, hence:

$$j^* = \underset{j}{\operatorname{argmax}}\, IG(\mathbb{T}, x_j) = \underset{j}{\operatorname{argmax}} \sum_{k=1}^{K} \sum_{v \in \mathbb{V}_j} f_k(\mathbb{T}_{j,v}) \cdot \log_2\left(\frac{f_k(\mathbb{T}_{j,v})}{|\mathbb{T}_{j,v}|}\right)$$

Since we are looking for the maximum value, the base of the logarithm is irrelevant.

If $\mathbb{T}_{j,v}$ is empty, the summand evaluates to $0 \cdot \log_2 \frac{0}{0} = 0$, i.e., empty partitions are simply ignored. Similarly, if $f_k(\mathbb{T}_{j,v}) = 0$, the summand evaluates to $0 \cdot \log_2 \frac{0}{|\mathbb{T}_{j,v}|} = 0$.

- Decision nodes only exists for discrete attributes. Partitioning is straightforward: for each possible value of the attribute, its partition contains all training items that have that value. Should a partition be empty (e.g., at that level of the tree no item has the value), prediction assume the most common label of the node.

– Ross Quinlan refined the ID3 algorithm and published the **C4.5 algorithm** in 1993. It got quite popular after ranking #1 in a Springer LNCS publication “10 top data mining algorithms (2008)”. C4.5 addresses many of the disadvantages of the original ID3 algorithm:
  - The information gain measure favors attributes with many values increasing the risk of overfitting the training data. Quinlan improved selection of attributes with the so-called split information. It is given as the entropy with respect to the attribute values rather than with respect to the target values as usually in this section. For each attribute x_j with discrete values $v \in \mathbb{V}_j$, the training set $\mathbb{T}$ is split into subsets $\mathbb{T}_{j,v}$. The split information $SI(\mathbb{T}, x_j)$ is:

$$SI(\mathbb{T}, x_j) = -\sum_{v \in \mathbb{V}_j} \frac{|\mathbb{T}_{j,v}|}{|\mathbb{T}|} \cdot \log_2 \frac{|\mathbb{T}_{j,v}|}{|\mathbb{T}|}$$

    The gain ratio is then the ratio between information gain and split information:

$$GR(\mathbb{T}, x_j) = \frac{IG(\mathbb{T}, x_j)}{SI(\mathbb{T}, x_j)}$$

    A practical issue, however, occurs if one $\mathbb{T}_{j,v}$ is almost as big a $\mathbb{T}$. This leads to a split information that is close to zero and hence a very large gain ratio. Clearly, such attributes are not interesting for decision nodes as most entries lie in the same branch. To counter this anomaly, the attribute selection process works as follows:

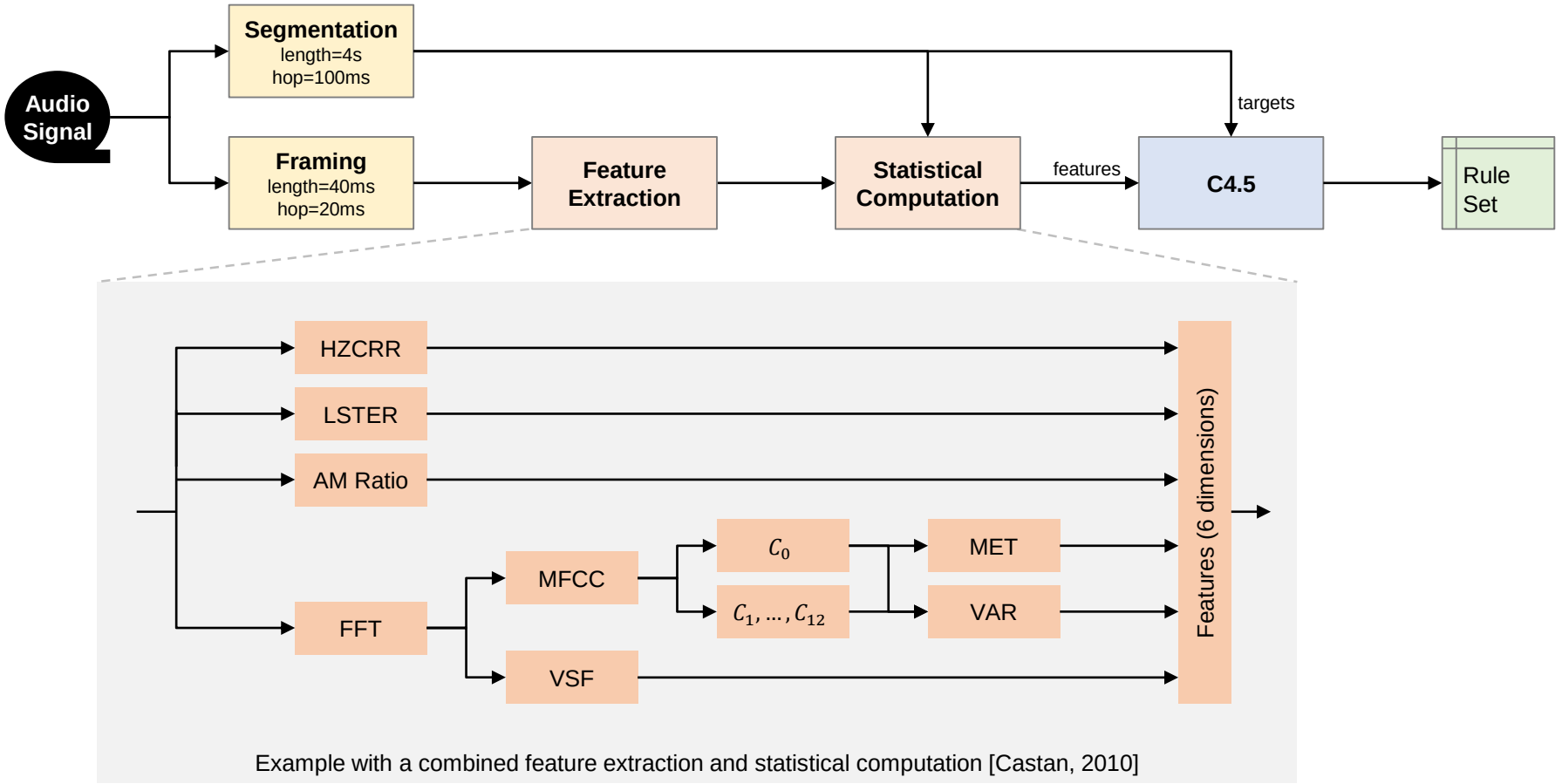
    - compute $IG(\mathbb{T}, x_j)$ for all x_j
    - select a threshold $IG_{Threshold}$, for example:
      - $IG_{Threshold} = \text{avg}\left(IG(\mathbb{T}, x_j)\right)$ (mean information gain)
      - $IG_{Threshold} = P_{50}\left(IG(\mathbb{T}, x_j)\right)$ (median information gain, 50-percentile)
    - $j^* = \underset{j;\, IG(\mathbb{T}, x_j) > IG_{Threshold}}{\text{argmax}} \left(\frac{IG(\mathbb{T}, x_j)}{SI(\mathbb{T}, x_j)}\right)$

- To allow for continuous values in decision trees, C4.5 maps a continuous attribute x_j to a Boolean attribute with a simple threshold value τ. If $x_j < \tau$, then the value is 'true', otherwise it is 'false'. So how can we select the best threshold value? Consider the example

Temperature	40	48	60	72	80	90
Play	No	No	Yes	Yes	Yes	No

  Obviously, we want to select a threshold value that maximizes the information gain. By sorting the training set according to the attribute values, we only need to identify adjacent data items with different targets, and create threshold candidates in between their values. For the example above, the first candidate is between 48 (No) and 60 (Yes), so we select $(48 + 60)/2 = 54$. The second candidate is between 80 (Yes) and 90 (No), so it is $(80 + 90)/2 = 85$. This produces two decision criteria $Temperature_{<54}$ and $Temperature_{<85}$ for which we compute the information gain and select the better one.
  There are extensions that map a continuous attribute to several intervals instead of just two as proposed by the C4.5 algorithm. In such cases, a balance is required to avoid overfitting to the intervals of the training set (e.g., using the validation set and an alternate scoring).
- There are several strategies to address missing values: the most simple one is to dismiss the object further down the branch if a test cannot be performed. This, however, disables also predictions for new data sets with missing values. A better strategy is to assign the most common value for the attribute at the current node (either for all training items or for only those that share the same target) and continue with this new value. A more complex approach is to compute distributions across all values, and use fractions for each value when following the branch. For instance, if there are two values, and if 40% of the data items have value 1 and 60% have value 0, a data item missing this attribute will be split and used in both branches but only counting for a fraction (0.4 in the branch for 1, 0.6 in the branch for 0).

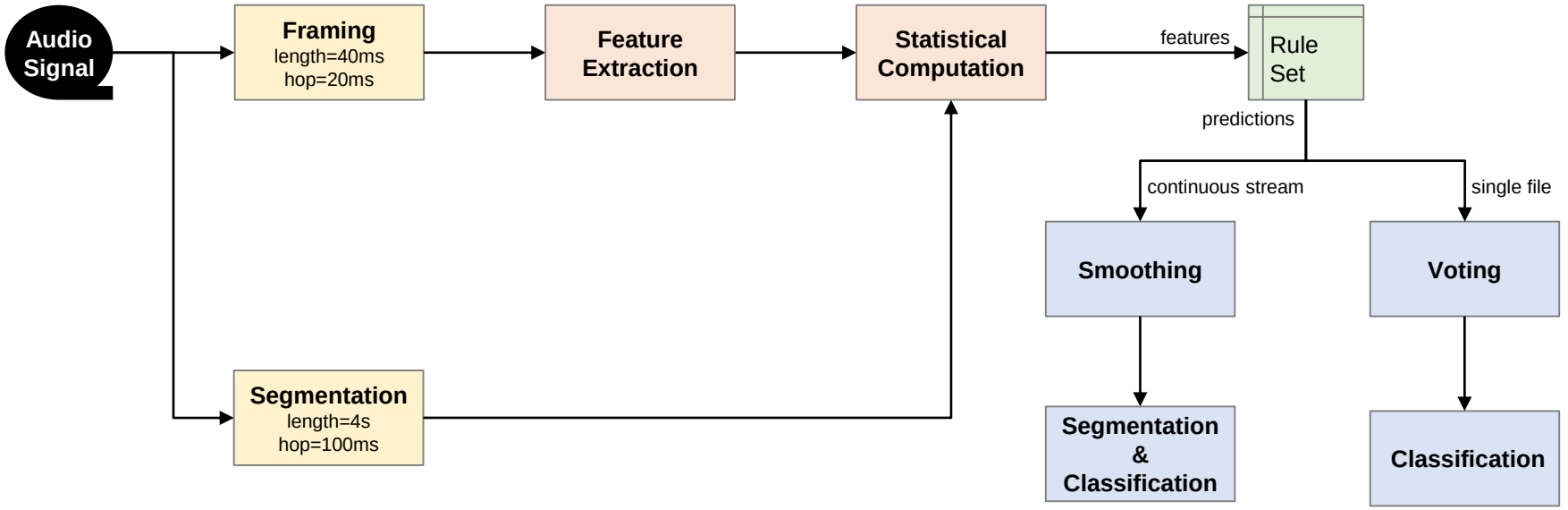
- **Example: audio classification**
  - Decision trees are very simple and produce efficient classifiers that are more than sufficient for many tasks. An example is discussed here: classify audio signals into speech and music.
  - In the learning phase, we need to pre-process the audio signal, extract features, gather statistical information about features and their mapping to output classes (music, speech), and select the best features for classification. In this example, we use C4.5 to select features and derive rules.



Example with a combined feature extraction and statistical computation [Castan, 2010]

- Framing and Segmentation: the audio signal is processed in overlapping frames and segments. Each frame and segment has the same length, and the hop distance specifies when the subsequent frame/segment starts. Typically, features are extracted per frame, and statistical measures are applied for the segment over its frame.
- Castan (2010) focused on a small number of characteristic features:
  - **HZCRR**: The Zero-Crossing Rates (ZCR) measures how often the amplitude of the signal passes the 0-value within a frame. The High Zero-Crossing Rate Ratio (HZCRR) measures, per segment, the ratio (percentage) of ZCR values of frames in the segment that are 1.5 times higher than the average ZCR value of frames in the segment.
  - **LSTER**: The Short Time Energy (STE) is simple the sum of squared amplitude of the signal within the frame (a measure of energy in the frame). The Low Short Time Energy Ratio measures, per segment, the ratio (percentage) of STE values of frames in the segment that are smaller than 50% of the average STE value of frames in the segment.
  - **AMR**: The Amplitude Modulation Ratio (AMR) measures the low-pass energy of a frame, i.e., the sum of squared amplitude after applying a low-pass filter with cut-off at 25Hz. It then measure the ratio of highest energy over lowest energy over all frames in the segment. Speech has a much higher ratio than music due to gaps between vowels and consonants.
  - **VSF**: The Spectral Flux (SF) is the Euclidean distance between subsequent frames over their fourier transformed signals (spectrum magnitudes). The Variation of Spectral Flux (VSF) measures the variance over the frames in the segment.
  - **MET & VAR**: For each frame, we extract 13 Mel-Frequency Cepstrum Coefficients (MFCC) denoted as $C_0, \ldots, C_{12}$. The Minimum-Energy Tracking (MET) measure how long C_0 is above a threshold. Pauses in speech will result in short lengths. VAR sums the variance of all MFCC over the frames in the segment. Small VAR values indicate music.

– In the prediction phase, we need to perform the same pre-processing, windowing, feature extraction, and statistical computations as in the learning phase. In addition, we want to smooth the results over the entire duration of the song (voting based approach) or to segment a continuous audio signal (e.g., radio broadcast) to detect changes from speech to music.



- Smoothing uses weighted sums over past predictions with exponentially smaller weighs to avoid fast alteration between targets. If enough support for a change is present, segmentation closes the current segment (not to be confused with the segments used for feature extraction) and labels it with the last class label. Then it marks the start of a new segment.
- Voting is rather simple: the single file is classified either by the label most frequently predicted for its segments, or classification returns probabilities for labels based on their frequencies in the predictions over all segments of the single file.

5.4.2 Naïve Bayes

- Bayesian classifiers go back to 1950. It has been applied in many areas, and still is competitive in text classification and medical diagnosis. Especially, Naïve Bayes scales very well to large feature dimensions where other methods, like decision trees, struggle from the curse of dimensionality:
- Naïve Bayes uses a conditional probability model based on Bayes theorem:

$$P(C_k|\boldsymbol{x}) = \frac{P(\boldsymbol{x}|C_k) \cdot P(C_k)}{P(\boldsymbol{x})} \qquad posterior = \frac{likelihood \cdot prior}{evidence}$$

  where $\boldsymbol{x}$ is a feature vector and C_k the class (=target). $P(C_k)$ is the so-called "**prior**", i.e., the knowledge (here a probability) about the distribution of classes C_k. $P(\boldsymbol{x}|C_k)$ is the **likelihood** to observe the feature $\boldsymbol{x}$ for a given class C_k, and $P(\boldsymbol{x})$ is the **evidence** to observe $\boldsymbol{x}$ (for any class). $P(C_k|\boldsymbol{x})$ is then the so-called "**posterior**", i.e., the knowledge we gain (or better: predict) given the observation of feature $\boldsymbol{x}$ to infer that it belongs to class C_k.
- Let $\boldsymbol{x}$ be a high-dimensional vector, for instance, from a huge term space for documents. Due to the high-dimensionality and the limited set of training data, it is difficult to accurately describe the probability distribution function in such a sparse space. To simplify matters, naïve Bayes assumes conditional independence of features. This immediately leads to the following simplification:

$$P(C_k|\boldsymbol{x}) = P(C_k|x_1, \dots, x_M) = \frac{1}{P(\boldsymbol{x})} \cdot P(C_k) \cdot \prod_{j=1}^{M} P(x_j|C_k)$$

Note that $P(\boldsymbol{x})$ is a constant over classes c_k and scales the probabilities. For our purposes, we do not need to know it.

- Given the probability model, we pick the hypothesis (here: class C_{k^*}) which is most probable. This selection rule is also known as the **maximum a posteriori (MAP)**:

$$k^* = \underset{k}{\operatorname{argmax}}\ P(C_k|\boldsymbol{x}) = \underset{k}{\operatorname{argmax}}\ P(C_k) \cdot \prod_{j=1}^{M} P(x_j|C_k)$$

That is it! The equation describes the decision rule of Naïve Bayes. The only thing left are the estimates for the probabilities on the right hand side

- To obtain the prior and the likelihood, we need to estimates the probability distributions based on the training set. And we need to address a number of practical issues such as numerical underflow due to the multiplication of many (small) probabilities, smoothing to address missing features, and feature selection. At the end, we apply the method to text classification
- **Learning process**
  - Estimating $P(C_k)$ is the easy part: let N_k bet the number of training items with label C_k and let N be the total number of training items. Then:

    $$P(C_k) = \frac{N_k}{N}$$

    If the exact numbers are not clear (for instance, spam classifier: what is the ratio between spam and normal email?), the probabilities can be approximated with $P(C_k) = 1/K$ with K denoting the number of classes, i.e., equiprobable classes. This is not accurate but works well.
  - To find the probability distribution $P(x_j|C_k)$ we first need to model the underlying distribution of values for x_j, and then learn the model parameters from the training set. The typical approach to learn estimators from training data is the **maximum likelihood estimation (MLE)**, i.e., choosing model parameters that maximize the likelihood of making the observations given the parameters.
  - Let x_j be discrete with values from $\mathbb{V}_j$. Let $N_k(x_j = v)$ with $v \in \mathbb{V}_j$ be the number of training items with label C_k that have $x_j = v$. In other words, it denotes how often $x_j = v$ is observed in the training set for items belonging to the class C_k. Naturally, we obtain

    $$P(x_j = v \mid C_k) = \frac{N_k(x_j = v)}{N_k}$$

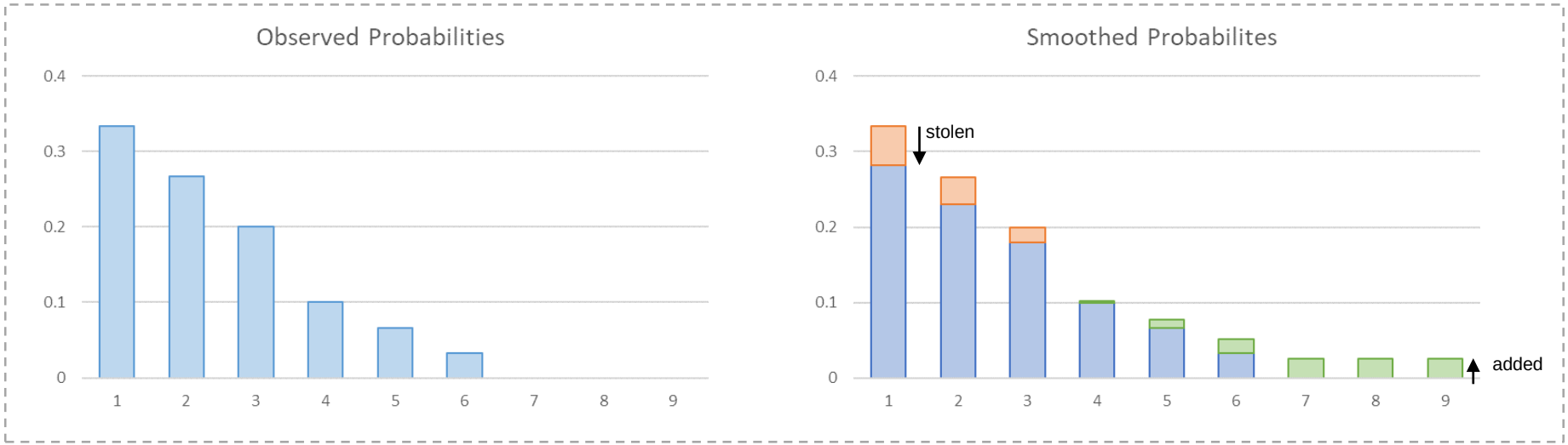
– What if a value v is never seen for x_j over a class C_k. Obviously, $P(x_j = v \mid C_k) = 0$ and with that:

$$P(C_k|\boldsymbol{x}) = P(C_k|x_1, \ldots, x_j = v, \ldots, x_M) = 0$$

In other words, if v was never observed for a class C_k, its presence in a new data item eliminates C_k as a prediction regardless how well the other features support C_k. To prevent 0-probabilities, we need to smooth the probability distribution, commonly using **Laplace smoothing (add-1)**. The idea is that we "steal" probability mass and distribute it to the values with 0-probabilities:

$$P(x_j = v \mid C_k) = \frac{N_k(x_j = v) + 1}{N_k + |\mathbb{V}_j|}$$

Note: the sum of $P(x_j = v \mid C_k)$ over all values $v \in \mathbb{V}_j$ is still 1. But we got rid of 0-probabilities.



Red indicates "stolen" probability mass and green denotes added probability mass.

– A special case is a discrete Boolean value $x_j \in \{0,1\}$ denoting the presence ($x_j = 1$) or absence ($x_j = 0$) of a feature in the training data. In this case, the distribution follows a Bernoulli event model (or a multivariate **Bernoulli event model** if several values are Boolean). As the probabilities sum up to 1, only one parameter is required:

$$P(x_j \mid C_k) = (p_{k,j})^{x_j} \cdot (1 - p_{k,j})^{(1-x_j)}$$

with $p_{k,j}$ representing the probability that the feature is present, i.e., how often $x_j = 1$ is observed in the training set for objects with label C_k. Hence:

$$p_{k,j} = \frac{N_k(x_j = 1)}{N_k} \qquad \text{or smoothed:} \qquad p_{k,j} = \frac{\min\left(N_k - 1, \max\left(1, N_k(x_j = 1)\right)\right)}{N_k}$$

Note that smoothing is done with stealing 1 only in the extreme case that all observations are the same (either all $x_j = 1$ or all $x_j = 0$).

– A final case for discrete values is the **multinomial event model** which is given by a feature vector $\boldsymbol{x} = (x_1, \ldots, x_M)$ representing a histogram with x_j counting the number of times a feature or event j was observed in the training set. We will see an example later on with x_j denoting the number of occurrences of a term t_j in a document. The probability distribution is given by:

$$P(\boldsymbol{x} \mid C_k) = \frac{(\sum_j x_j)!}{\prod_j x_j!} \cdot \prod_j (p_{k,j})^{x_j}$$

Note that the factor to the left of the product symbol is a constant when looking for the best class C_k and hence drops in the argmax equation

Let $n_{k,j}$ be the total number of occurrences of feature j in all training items with label C_k. Then:

$$p_{k,j} = \frac{n_{k,j}}{\sum_l n_{k,l}} \qquad \text{or smoothed:} \qquad p_{k,j} = \frac{n_{k,j} + 1}{\sum_l n_{k,l} + M}$$

– If feature values x_i are continuous, we need to choose a model for the probability distribution $p(x_i|C_k)$ and then learn the parameters of the model using the training set. A common approach is assuming a **Gaussian distribution** with the two parameters $\mu_{k,i}$ denoting the mean value, and $\sigma_{k,i}^2$ being the variance. The probability distribution is defined as:

$$p(x_i|C_k) = \frac{1}{\sqrt{2\pi\sigma_{k,i}^2}} \cdot e^{-\frac{(x_i-\mu_{k,i})^2}{2\sigma_{k,i}^2}}$$

To estimate the two parameters, we need to use the **unbiased** estimators based on the observations from the training set. Let $N_k = |C_k|$ be the number of training items with label C_k:

$$\mu_{k,i} = \frac{1}{N_k} \sum_{x \in C_k} x_i \qquad \sigma_{k,i} = \frac{1}{N_k - 1} \sum_{x \in C_k} (x_i - \mu_{k,i})^2$$

When estimating variance from samples, we must account for the error in the estimated mean value, that is, we underestimate the variance because differences between values and the estimated mean are too small.

– Using a **Gaussian mixture model,** we can adopt to arbitrarily shaped distribution function. We overlay L normal distributions $\mathcal{N}(\mu_{k,i,l}, \sigma_{k,i,l}^2)$ with weights w_l:

$$p(x_i|C_k) = \sum_{l=1}^{L} w_l \cdot \mathcal{N}(\mu_{k,i,l}, \sigma_{k,i,l}^2)$$

To learn the parameters of the normal distributions, we can use the **Expectation Maximization** approach (we will see this later for clustering methods). In addition, we should use a validation set to adjust the hyper-parameter L, i.e., if L is large, we may fit the probability distribution for the training set very well, but cannot generalize well to the validation set due to overfitting. Using least mean squared errors over the validation set provides an instrument to control L.

- **Prediction**
  - To predict the class C_{k^*} to which a new data item with features $\boldsymbol{x}$ belongs to, we apply the maximum a posteriori (MAP) selection:

$$k^* = \underset{k}{\operatorname{argmax}}\ P(C_k|\boldsymbol{x}) = \underset{k}{\operatorname{argmax}}\ P(C_k) \cdot \prod_{j=1}^{M} P\left(x_j \middle| C_k\right)$$

    With moderate to large numbers for M, we run into practical issues due to the multiplications of small probabilities (numerical underflow). To provide a stable calculation of the probabilities, naïve Bayes algorithms compute log-probabilities as the logarithm does not change the ordering:

$$k^* = \underset{k}{\operatorname{argmax}}\ \log\left(P(C_k|\boldsymbol{x})\right) = \underset{k}{\operatorname{argmax}} \left(\log P(C_k) + \sum_{j=1}^{M} \log P\left(x_j \middle| C_k\right) \right)$$

  - To reduce the noise of a large number of features, we can focus on a few features only that are sufficient to classify data items. In general terms, we want to identify features whose presence or absence is correlated with the data item having or not having a label. This leads to 4 tests for each of the combinations of {“feature present”, “feature not present”} and {“item in class”, “item not in class”}. If there is a strong correlation for any combination of events, then the feature is discriminative for classification. Literature provides several approaches with **Chi-square** and **mutual information** being the most prominent ones. A much simpler approach is to select the most discriminative features, much like we have seen in classical text retrieval.

- **Example: Text Classification –** Naïve Bayes is quite popular due to its simplicity, its speed, and accuracy. Common applications include spam detection, author identification, age/gender identification, language identification, and sentiment analysis. With sentiment analysis, for example, we want to distinguish positive from negative movie reviews.
  - There are two models for text classification: 1) set of words, and 2) bag of words. With the former, we consider only the presence of terms and apply a multivariate Bernoulli model. With the latter, we count term occurrences and use the multinomial model. Both approaches assume that the position of terms in the text does not matter and that terms are conditionally independent.
  - **Set of words and multivariate Bernoulli**: like with Boolean text retrieval models, a binary feature vector $\boldsymbol{x}$ denotes the presence of terms, taken from a defined vocabulary, in the given documents. The training documents have labels for classes C_k, and we use the training set to estimate the probabilities. Let N_k bet the number of training items with label C_k, then

$$P(C_k) = \frac{N_k}{N} \qquad \text{or if } N_k \text{ is not known:} \quad P(C_k) = \frac{1}{K}$$

Let $x_j = 1$ denote that term t_j is present in the document represented by $\boldsymbol{x}$. Then:

$$p_{k,j} = \frac{N_k(x_j = 1)}{N_k} \qquad \text{or smoothed:} \quad p_{k,j} = \frac{\min\big(N_k - 1, \max\big(1, N_k(x_j = 1)\big)\big)}{N_k}$$

Prediction means finding the class that maximizes $P(C_k|\boldsymbol{x})$ for a document with representation $\boldsymbol{x}$:

$$k^* = \underset{k}{\operatorname{argmax}}\ P(C_k|\boldsymbol{x}) = \underset{k}{\operatorname{argmax}} \left(\log P(C_k) + \sum_{j=1}^{M} \big(x_j \log p_{k,j} + \big(1 - x_j\big) \log\big(1 - p_{k,j}\big) \big) \right)$$

Instead of using all terms of the vocabulary, we can reduce the features (see feature selection) or only take the terms present in the document (i.e., we only consider $x_j = 1$).

– **Bag of words and multinomial**: like with vector space retrieval models, a feature vector $\boldsymbol{x}$ denotes the number of occurrences of terms, taken from a defined vocabulary, in the given documents. The training documents have labels for classes C_k, and we use the training set to estimate the probabilities. Let N_k bet the number of training items with label C_k, then

$$P(C_k) = \frac{N_k}{N} \qquad \text{or if } N_k \text{ is not known:} \quad P(C_k) = \frac{1}{K}$$

Let $n_{k,j}$ be the total number of occurrences of term t_j in all training documents with label C_k:

$$p_{k,j} = \frac{n_{k,j}}{\sum_l n_{k,l}} \qquad \text{or smoothed:} \quad p_{k,j} = \frac{n_{k,j} + 1}{\sum_l n_{k,l} + M}$$

Prediction means finding the class that maximizes $P(C_k|\boldsymbol{x})$ for a document with representation $\boldsymbol{x}$:

$$k^* = \underset{k}{\operatorname{argmax}}\ P(C_k|\boldsymbol{x}) = \underset{k}{\operatorname{argmax}} \left(\log P(C_k) + \sum_{x_j > 0} x_j \log p_{k,j} \right)$$

That is, we select the best class only with the terms that are present in the document.

- Summary: Naïve Bayes is not so naïve. Even though the strong assumption of independence does not always apply in practices, it excels due to high speed, low storage requirements, robustness to noise, and very good performance (accuracy). There are better methods but still naïve Bayes is an excellent baseline for text classification.

5.4.3 Unsupervised Clustering

- With unsupervised learning tasks, the machine learning algorithm observes data set without targets and infers a function that captures the inherent structure and/or distribution of the data. In a clustering scenario, that function is a set of clusters and the ability to assign new data items to one (or several) of the clusters. In this chapter, we study the **k-means clustering** and the **Expectation Maximization over a Gaussian mixture** to infer a mapping of features to clusters. In the context of multimedia data, typical applications are:
  - Feature quantization, i.e., reducing a multivariate feature to a small number of discrete values. The quantized value serve as an approximated or smoothed version of the original ones much like histograms approximates the distribution of data values
  - Cluster analysis, i.e., the validation of the cluster hypothesis and the extraction of clusters to infer labels for the clusters.
  - Image segmentation, i.e., the extraction of different areas in an image that “belong” to each other. In a first step, clustering reduces the number of features through quantization. In a second step, morphological operators build coherent regions for segmentation.
- As we do not know the number of clusters that are present in the data (we have no labels!), we need to guide clustering algorithms in the selection of the optimal number K of clusters. Again, poor choice for the number of clusters can lead to underfitting (extreme case is $K = 1$) and overfitting (extreme case is $K = N$ with N being the number of training items). As we have no targets, we cannot use a validation set to measure accuracy of prediction. Instead, we utilize a target function for the compactness of the clusters and the separation between clusters and must prevent, at the same time, an excessive number of clusters.
- We conclude this section with an example from image segmentation and a very early application called Blobworld.

- **k-means clustering** goes back to the 1960s as an approach to quantify vectors for signal processing. It subsequently became very popular in data mining for cluster analysis. k-means clusters the data set into k clusters in such a way that each data point belongs to the cluster with the nearest centroid (or prototype of the cluster). The centroids are the mean position over all points in the cluster. The centroids divide the space into Voronoi diagrams defining the cluster shapes.
  - Although the computation of the optimal K centroids is a NP-hard problem, there are very efficient heuristics that lead to a (local) optimum. We will first describe the classical approach using Lloyd's algorithm and then re-interpret the approach with Expectation Maximization.
  - Let N be the number of data items with the d-dimensional representations $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_N$. We then want to partition the data items into K sets $\mathbb{S} = \{\mathbb{S}_1, \ldots, \mathbb{S}_K\}$ such that the **within-cluster sum of squares** (**WCSS**, also called the variance) become minimal, i.e.:

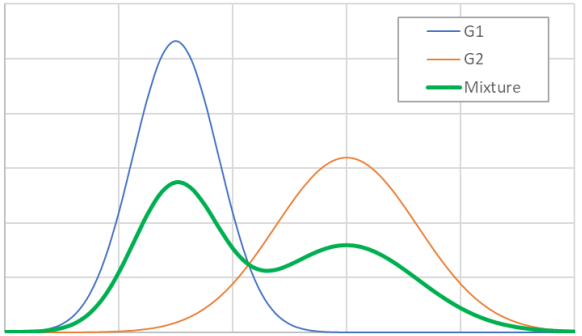
$$\mathbb{S}^* = \underset{\mathbb{S}}{\operatorname{argmin}} \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in \mathbb{S}_k} \|\boldsymbol{x} - \boldsymbol{\mu}_k\|_2^2 = \underset{\mathbb{S}}{\operatorname{argmin}} \sum_{k=1}^{K} |\mathbb{S}_k| \cdot \sigma_k^2$$

  with $\boldsymbol{\mu}_k$ denoting the mean of items in $\mathbb{S}_k$, and σ_k^2 being the variance of items in $\mathbb{S}_k$. With Lloyd's algorithm, we obtain a local optimum with a simple iterative algorithm:

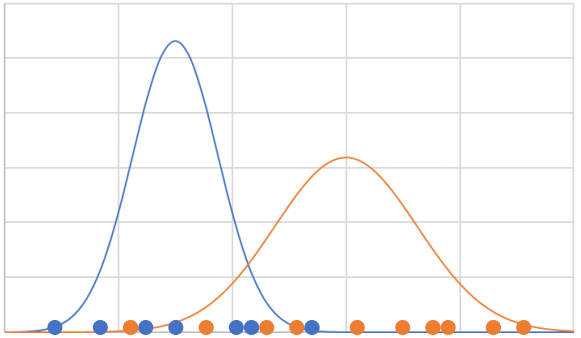
1. Select an initial set of centroids $\boldsymbol{\mu}_1^{(0)}, \ldots, \boldsymbol{\mu}_K^{(0)}$ (see later how to select)
2. Assign each data point $\boldsymbol{x}$ to the set $\mathbb{S}_k^{(t)}$ if it is closest to $\boldsymbol{\mu}_k$, i.e., $\left\|\boldsymbol{x} - \boldsymbol{\mu}_k^{(t)}\right\| \leq \left\|\boldsymbol{x} - \boldsymbol{\mu}_l^{(t)}\right\| \ \forall l: 1 \leq l \leq K$ (if several centroids are closest, pick one randomly)
3. Calculate the new centroids for the next iteration $(t+1)$:
$$\boldsymbol{\mu}_k^{(t+1)} = \frac{1}{\left|\mathbb{S}_k^{(t)}\right|} \sum_{\boldsymbol{x} \in \mathbb{S}_k^{(t)}} \boldsymbol{x}$$
4. Repeat steps 2 and 3 until algorithm has converged

  - Initial choice of centroids
    - Random points: pick K random items from the data set. This leads to a spread of centroids across the data space.
    - Random partition: assign each data item to a random cluster (1 to K) and compute centroids over these random clusters. These centroids tend to be closer together near the center of the data set.
    - k-means++: the first centroid is chosen randomly from the data set. Each subsequent centroid (up to K) is chosen from the remaining items with probabilities proportional to the their squared distance to closest centroid. Although more expensive, it leads to much smaller final errors and faster convergence during the iterative part.
- **Expectation Maximization (EM)** (and interpretation of k-means algorithm)
  - Expectation maximization is an iterative method to estimate parameters in a statistical model than cannot be solved in closed form. It assumes that the observations (here: the training set) are obtained from probability distribution, typically a mixture of several distributions with a soft assignment. In k-means, we used a hard assignment, that is, every data point is assigned to exactly one cluster. In EM, soft assignment denotes that cluster assignment of a point follows a conditional distribution. Finally, the objective is to find the soft assignment and the parameters of the distributions (e.g., with Gaussian, these are the means and variances) that best explain the observations (maximum likelihood).
  - Solving above objective function in closed form is not always possible. The EM algorithm consists of two steps: in the expectation step, the distribution parameters are constant and we compute the best soft assignment. In the maximization step, we keep the soft assignment constant and choose the parameters that maximize the objective function. With each step, the objective function increases and eventually converges, but not necessarily to a global maximum.

– Let us start with a simple one dimensional example with a mixture of two ($K = 2$) Gaussian distributions $\mathcal{N}(\mu_k, \sigma_k^2)$. The picture on the right shows the two Gaussian distributions and their mixture. With an infinite number of Gaussians, a mixture can model any distribution. Each Gaussian represent a sub-population (cluster) of the data items that follow its distribution. In addition, a prior $P(C_k)$ defines how likely data items come from k-the cluster with $\sum P(C_k) = 1$.

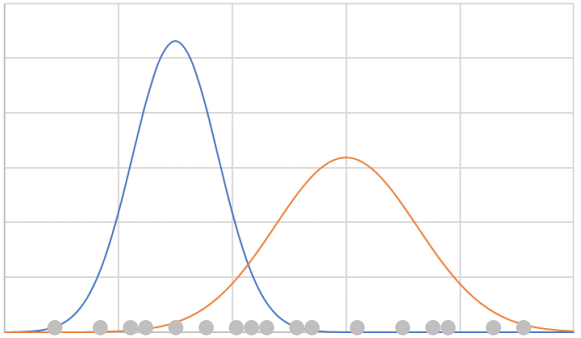


– Now, assume we make the observations $\mathbb{T} = \{x_1, \ldots, x_N\}$. Further assume, we know that all $x \in \mathbb{S}_1$ stem from the blue cluster C_1, and all $x \in \mathbb{S}_2 = \mathbb{T} \setminus \mathbb{S}_1$ stem from the red cluster C_2. We then can easily compute the parameters and the priors of the distributions using the (biased) estimators:



$$\mu_k = \frac{\sum_{x \in \mathbb{S}_k} x}{|\mathbb{S}_k|} \qquad \sigma_k^2 = \frac{\sum_{x \in \mathbb{S}_k} (x - \mu_k)^2}{|\mathbb{S}_k|} \qquad P(C_k) = \frac{|\mathbb{S}_k|}{N}$$

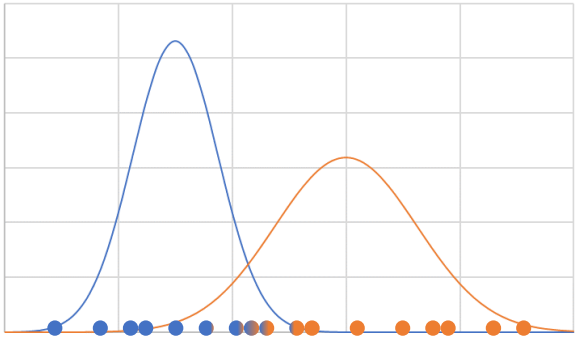
– On the other side, assume we know the parameters μ_k, σ_k^2 of the distributions and the priors $P(C_k)$, can we estimate the probability $P(C_k|x_i)$ that a point x_i is part of cluster C_k?



$$P(C_k|x_i) = \frac{P(x_i|C_k) \cdot P(C_k)}{P(x_i)} = \frac{P(x_i|C_k) \cdot P(C_k)}{\sum_k P(x_i|C_k) \cdot P(C_k)}$$

with $$P(x_i|C_k) = f(x_i;\, \mu_k, \sigma_k^2) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \cdot \exp\left(-\frac{(x_i - \mu_k)^2}{2\sigma_k^2}\right)$$

– Given the probabilities $P(C_k|x_i)$ that x_i belongs to cluster C_k we no longer have a hard assignment as above with $\mathbb{T} = \mathbb{S}_1 \cup \mathbb{S}_2$, and $\mathbb{S}_1 \cap \mathbb{S}_2 = \emptyset$, but utilize soft assignments. In other words, we are not entirely sure from which sub-population the points come from but have a fairly good understanding how likely they stem from each cluster. To estimate the parameters and the priors, we need to take the soft assignments into account:



$$\mu_k = \frac{\sum_i P(C_k|x_i) \cdot x_i}{\sum_i P(C_k|x_i)} \qquad \sigma_k^2 = \frac{\sum_i P(C_k|x_i) \cdot (x - \mu_k)^2}{\sum_i P(C_k|x_i)} \qquad P(C_k) = \frac{\sum_i P(C_k|x_i)}{N}$$

– Now we can summarize the EM algorithm: to this end, we introduce the **responsibility** $\gamma_{i,k} = P(C_k|x_i)$ denoting the soft assignment of data item x_i to cluster C_k, and the **weights** $w_k = P(C_k)$ representing the prior of cluster C_k. The algorithm runs as follows:

1. Select initial values for $\mu_k^{(0)}, \sigma_k^{2\,(0)}$ and $w_k^{(0)}$ for $1 \le k \le K$
2. E-step: evaluate new responsibilities $\gamma_{i,k}^{(t)}$ for $1 \le i \le N$ and $1 \le k \le K$ using current parameters

$$\gamma_{i,k}^{(t)} = \frac{\mathrm{w_k^{(t)}} \cdot f\left(x_i;\, \mu_k^{(t)}, \sigma_k^{2\,(t)}\right)}{\sum_k \mathrm{w_k^{(t)}} \cdot f\left(x_i;\, \mu_k^{(t)}, \sigma_k^{2\,(t)}\right)}$$

3. M-step: evaluate new parameters $\mu_k^{(t+1)}, \sigma_k^{2\,(t+1)}$ and $w_k^{(t+1)}$ for $1 \le k \le K$ using current responsibilities

$$\mu_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot x_i}{\sum_i \gamma_{i,k}^{(t)}} \qquad \sigma_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot \left(x_i - \mu_k^{(t+1)}\right)^2}{\sum_i \gamma_{i,k}^{(t)}} \qquad w_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)}}{N}$$

4. Repeat E-step and M-step until the parameters stop changing

– Once convergence of EM is reached after ϑ iterations, we can (hard) assign a data item x_i to its most likely cluster C_{k^*} by solving the following equation:

$$k^* = \underset{k}{\operatorname{argmax}} P(C_k|x_i) = \underset{k}{\operatorname{argmax}} \frac{P(x_i|C_k) \cdot P(C_k)}{P(x_i)} = \underset{k}{\operatorname{argmax}} \left(w_k^{(\vartheta)} \cdot f\left(x_i;\ \mu_k^{(\vartheta)}, \sigma_k^{2\,(\vartheta)}\right)\right)$$

– We can generalize this approach to d-dimensional spaces with $d = M$ being the number of features. We create a mixture of K multi-variate (or multi-dimensional) Gaussian distribution $\mathcal{N}(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ with $\boldsymbol{\mu}_k = \mathrm{E}[\boldsymbol{x} \in \mathbb{T}_k]$ denoting the centroid of items of cluster C_k, and $\boldsymbol{\Sigma}_k = \mathrm{E}_{\boldsymbol{x}\in\mathbb{T}_k}[(\boldsymbol{x} - \boldsymbol{\mu})(\boldsymbol{x} - \boldsymbol{\mu})^T]$ the covariance matrix of items in cluster C_k.

1. Select initial values for $\boldsymbol{\mu}_k^{(0)}, \boldsymbol{\Sigma}_k^{2\,(0)}$ and $w_k^{(0)}$ for $1 \le k \le K$
2. E-step: evaluate new responsibilities $\gamma_{i,k}^{(t)}$ for $1 \le i \le N$ and $1 \le k \le K$ using current parameters

$$\gamma_{i,k}^{(t)} = \frac{\mathrm{w_k^{(t)}} \cdot f\left(\boldsymbol{x}_i;\ \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{2\,(t)}\right)}{\sum_k \mathrm{w_k^{(t)}} \cdot f\left(\boldsymbol{x}_i;\ \boldsymbol{\mu}_k^{(t)}, \boldsymbol{\Sigma}_k^{2\,(t)}\right)}$$

3. M-step: evaluate new parameters $\boldsymbol{\mu}_k^{(t+1)}, \boldsymbol{\Sigma}_k^{2\,(t+1)}$ and $w_k^{(t+1)}$ for $1 \le k \le K$ using current responsibilities

$$\boldsymbol{\mu}_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot \boldsymbol{x}_i}{\sum_i \gamma_{i,k}^{(t)}} \qquad \boldsymbol{\Sigma}_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot \left(\boldsymbol{x}_i - \boldsymbol{\mu}_k^{(t+1)}\right)\left(\boldsymbol{x}_i - \boldsymbol{\mu}_k^{(t+1)}\right)^T}{\sum_i \gamma_{i,k}^{(t)}} \qquad w_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)}}{N}$$

4. Repeat E-step and M-step until the parameters stop changing

– Again, we obtain a hard assignment for a data item $\boldsymbol{x}_i$ to its most likely cluster C_{k^*} as follows:

$$k^* = \underset{k}{\operatorname{argmax}} \left(w_k^{(\vartheta)} \cdot f\left(\boldsymbol{x}_i;\ \boldsymbol{\mu}_k^{(\vartheta)}, \boldsymbol{\Sigma}_k^{2\,(\vartheta)}\right)\right) \qquad f\left(\boldsymbol{x}_i;\ \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k^2\right) = \frac{1}{\sqrt{(2\pi)^d |\boldsymbol{\Sigma}|}} \cdot \exp\left(-\frac{1}{2}(\boldsymbol{x}_i - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_{\mathrm{k}}^{-1} (\boldsymbol{x}_i - \boldsymbol{\mu}_k)\right)$$

– Where does the name Expectation Maximization come from? Let $\mathbb{X} = \{\boldsymbol{x}_i\}$ be the set of data items and $\mathbb{Y} = \{w_1, \mu_1, \sigma_1, \ldots, w_k, \mu_K, \sigma_K\}$ be the set of unknown parameters of the mixture of K Gaussian distributions. In addition, we have the latent unobserved data items $\mathbb{Z} = \{\gamma_{i,k}\}$ denoting the soft memberships of x_i to cluster C_k. Given, $\mathbb{X}$ we want to find the parameters $\mathbb{Y}$ that maximize the probability that the data items in $\mathbb{X}$ are observations from the mixture using these parameters. This is called the **maximum likelihood estimate (MLE)**:

$$\mathbb{Y}^* = \underset{\mathbb{Y}}{\operatorname{argmax}}\, p(\mathbb{X}|\mathbb{Y}) = \int_{\mathbb{Z}} p(\mathbb{X}, \mathbb{Z}|\mathbb{Y})\, d\mathbb{Z}$$

In other words, if $\mathbb{Y}$ is known, how likely is it that data items in $\mathbb{X}$ follow the mixture of the K Gaussian distributions. Adding the soft memberships $\mathbb{Z}$, $p(\mathbb{X}|\mathbb{Y})$ is given by the marginal probability of $p(\mathbb{X}, \mathbb{Z}|\mathbb{Y})$ over all possible sets of $\mathbb{Z}$. This equation, however, is often not solvable in closed forms. Instead, an iterative method is used, that improves $\log p(\mathbb{X}|\mathbb{Y})$ with each iteration. EM uses a so-called Q-function that indirectly improves $\log p(\mathbb{X}|\mathbb{Y})$ given current estimates $\mathbb{Y}^{(t)}$:

$$Q\left(\mathbb{Y}\middle|\mathbb{Y}^{(t)}\right) = E_{\mathbb{Z}|\mathbb{X},\mathbb{Y}^{(t)}}[\log p(\mathbb{X}, \mathbb{Z}|\mathbb{Y})]$$

The right hand side is the expectation function over $\log p(\mathbb{X}, \mathbb{Z}|\mathbb{Y})$ given the conditional distribution of $\mathbb{Z}$ given $\mathbb{X}$ and the current estimates $\mathbb{Y}^{(t)}$. Now, the E-step generates this expectation function by computing the probabilities $P(C_k|x_i)$ for $\mathbb{Z}$ (soft assignment) given $\mathbb{X}$ and the current estimates $\mathbb{Y}^{(t)}$ and uses Bayes' rule as we have done above. Then, given $\mathbb{Z}$, the M-step maximizes the Q-function over all possible $\mathbb{Y}$ to obtain a new estimate $\mathbb{Y}^{(t+1)}$. With log-probabilities and Gaussian distributions, we can cancel $\log$ and $\exp$ in the equation, and solutions are found by solving for the maximum (partial derivative is zero). We omit proof for solutions and convergence.

Additional information — not part of the exams

– Let us reconsider the k-means algorithm as an EM problem. We can re-write the objective function (within-cluster sum of squares, WCSS) as follows:

$$J = \sum_{i=1}^{N} \sum_{j=1}^{k} \gamma_{i,k} \|\boldsymbol{x}_i - \boldsymbol{\mu}_k\|_2^2$$

$\gamma_{i,k}$ are the hard assignments of x_i to C_k, i.e., for each $1 \leq i \leq N$ exactly one $\gamma_{i,k} = 1$ and all others are 0. We can transform k-means to an EM algorithm over a mixture of K Gaussian distributions with hard assignments as follows:

1. Select initial values for $\boldsymbol{\mu}_k^{(0)}$. Keep $\boldsymbol{\Sigma} = \mathbf{I}$ and $w_k = 1/k$ constant
2. E-step: evaluate new responsibilities $\gamma_{i,k}^{(t)}$ for $1 \leq i \leq N$ and $1 \leq k \leq K$ using current parameters

$$\gamma_{i,k}^{(t)} = \begin{cases} 1 & \text{if } k = \underset{l}{\operatorname{argmin}} \|\boldsymbol{x}_i - \boldsymbol{\mu}_l\|_2^2 \\ 0 & \text{otherwise} \end{cases}$$

3. M-step: evaluate new parameters $\boldsymbol{\mu}_k^{(t+1)}$ for $1 \leq k \leq K$ using current responsibilities

$$\boldsymbol{\mu}_k^{(t+1)} = \frac{\sum_i \gamma_{i,k}^{(t)} \cdot \boldsymbol{x}_i}{\sum_i \gamma_{i,k}^{(t)}}$$

4. Repeat E-step and M-step until the parameters stop changing

Additional information — not part of the exams

- For both k-means and EM, we need to control then number K of clusters. If the number is too small, the error value is high and the algorithms suffer from underfitting. If we select a large K, we can reduce the error but at risk of overfitting. Let $\mathbb{S}_k$ be the set of data items $\boldsymbol{x}$ that are assigned to cluster C_k. To control K, we determine the **sum of squared errors** SSE over all clusters:

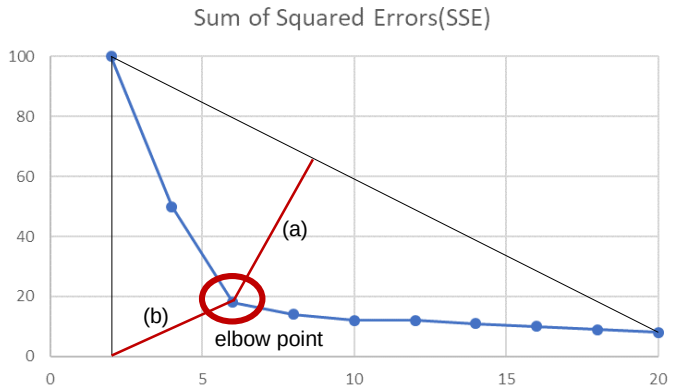
$$SSE(\text{k}-\text{means}) = \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in \mathbb{S}_k} \|\boldsymbol{x} - \boldsymbol{\mu}_k\|_2^2 \qquad SSE(\text{EM}) = \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in \mathbb{S}_k} (\boldsymbol{x} - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_{\text{k}}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_k)$$

If we plot this SSE as a function of K, we obtain a graph like on the right side below. As we increase the number K, the SSE decreases. However, we cannot simply solve for K that minimizes the SSE function as $K = N$ would have an $SSE = 0$ but clearly overfits the data. Rather, we look for the so-called elbow point as highlighted in the figure where the SSE-functions "abruptly" levels out as is decreasing much slower than before the elbow. We can obtain an optimal K in two ways:

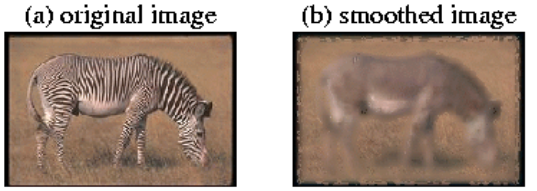
a) Vary K from 2 to an upper bound (here 20) and determine the point that lies farthest away from the line between the start and the end of the curve.

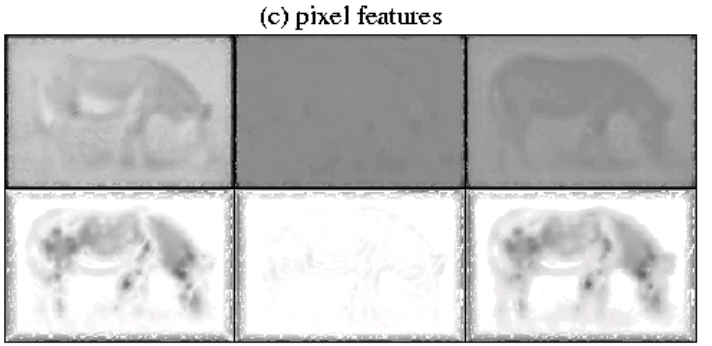
b) Start with $K = 2$ and determine the distance to the point (2,0). While increasing K observe the distance. Stop if the distance starts growing.

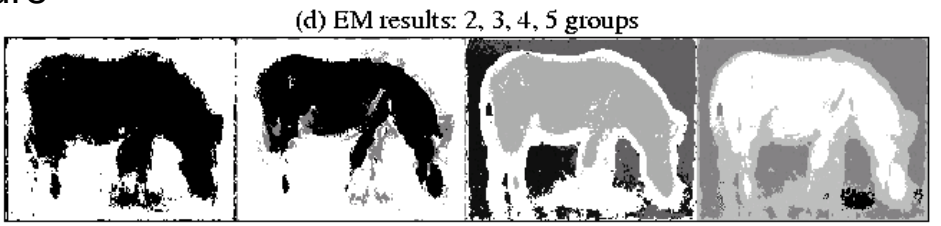
Method b) has the advantage of iterating less over K. For both variants to work, we need to normalize the two dimensions, for instance with a min/max scaling, to obtain a meaningful result.

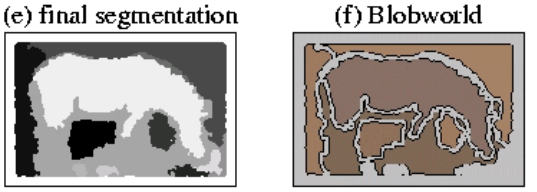


- **Example: Image Segmentation (Blobworld)**
  - Blobworld was a project at the University of Berkeley and published first in 1999. It was using segmentation to divide an image into distinct regions and used descriptors on these regions to retrieve objects embedded in images. The right hand side shows an example of the segmentation
    a) The original image contains too many edges and corners yielding a large number of potential regions
    b) A rough Gaussian filter smooths the image and eliminates finer structures
    c) Color is transformed into the L*a*b* space. For each pixel, Blobworld extract additional texture features describing the polarity (clear direction of edges in a neighborhood), edgeness, and texture contrast. The feature vector consists of the pixel position (x, y), the 3 color and the 3 texture values at that position.
    d) Apply the EM algorithm on a Gaussian mixture model over the 8 feature values. This is computed for 2, 3, 4, and 5 clusters.
    e) To steer the number of clusters, a special objective function based on the Minimum Description Length (MDL) was applied.
    f) Blobworld hard assigns pixels to a cluster and selects a unique color for each cluster.



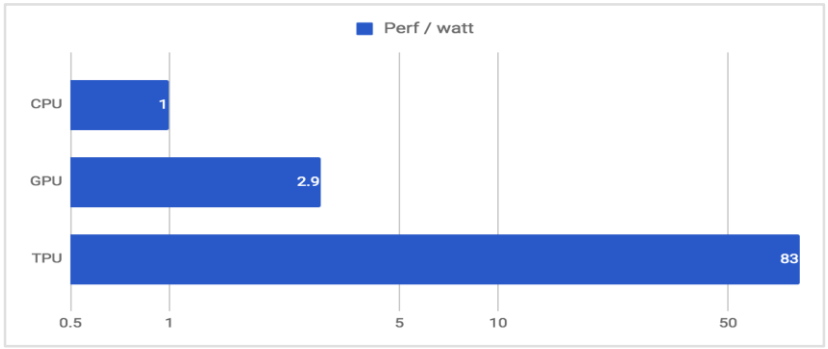




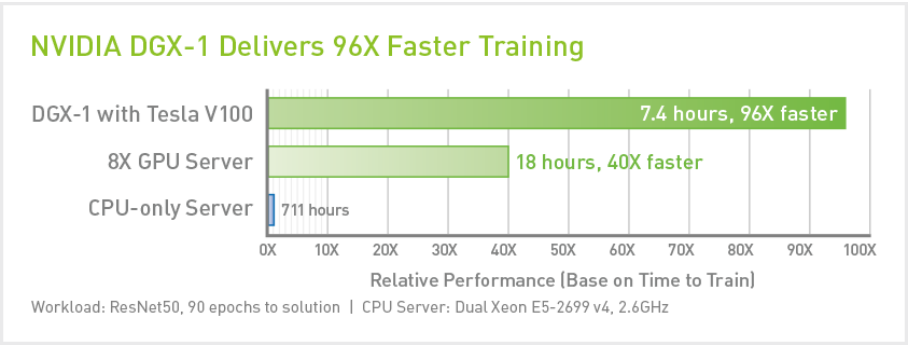


5.4.4 Multi-Layer Network

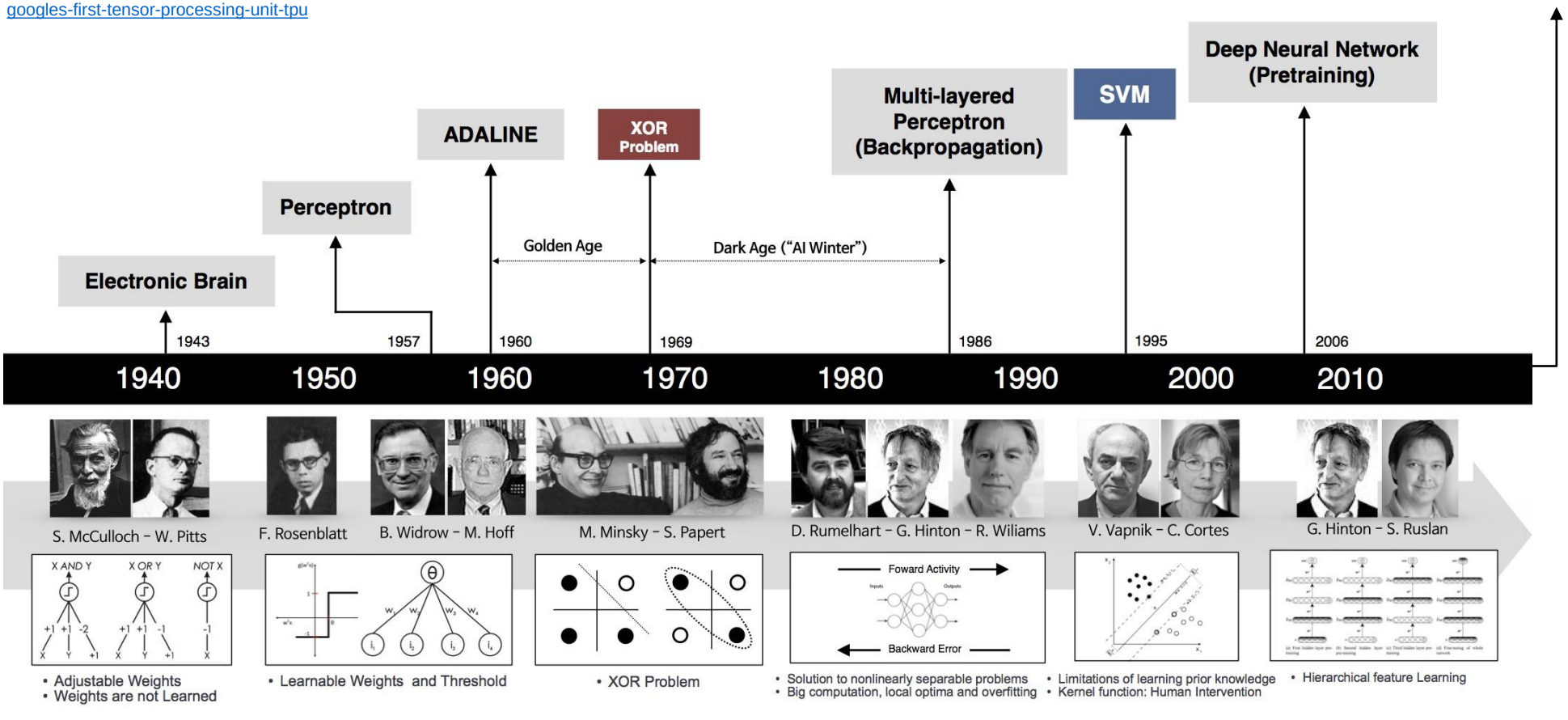
- Artificial neural networks are machine learning models that are inspired by how the brain works. Indeed, brain research has frequently led to new approaches like the use of connections between neurons of different layers rather than adjacent ones (multi-layer approach). Neural network, on the other hand, are often employed to model the brain and its learning algorithms.
- The first wave of neural network research started in the late 1950s and was focusing on a single perceptron (in hardware). It was possible to use multiple perceptrons in parallel, but they were only connected to input and output states. The problem of perceptrons was articulated in its famous inability to learn a simple XOR function. Even though it was shown that a two-layer network could indeed encode an XOR function, the limitations were obvious and a first AI winter began.
- The second wave started with research in the 1960s with the introduction of hidden layers. Several researchers were developing similar ideas but the credits usually go the Rumelhart, Hinton, and Williams and their 1986 paper on backpropagation which describes the approach with such clarity that it is still the basis for many descriptions in text books. The area revived quickly and lead to convolutional networks, recurrent networks, belief networks with many of the concepts found today in deep learning. However, the field suffered from calculation issues (vanishing and exploding gradients) and the computational limitations in the 1980s and 1990s.
- At the beginning of the 2000s, almost no research was published or cited and funding was very sparse. However, the Canadian government funded a small research team around Hinton that first rebranded the field into “Deep Learning” and then published in 2006 a break-through paper with a fast learning algorithm for deep belief nets. In parallel, compute power has significantly grown. Inspired by the Canadian research team, the field arose again and soon it was found that GPUs were up to 100 times faster than CPUs. This allowed the training of deep networks within hours and days rather than weeks and months. Google started in 2011 its Google Brain research project to connect thousand of CPUs for a network with 1 billion weights. Since then, research has generated an enormous amount of improvements and efficient learning frameworks leading to an overwhelming success story of AI with many applications.



source: https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu



source: https://www.nvidia.com/en-us/data-center/dgx-server/

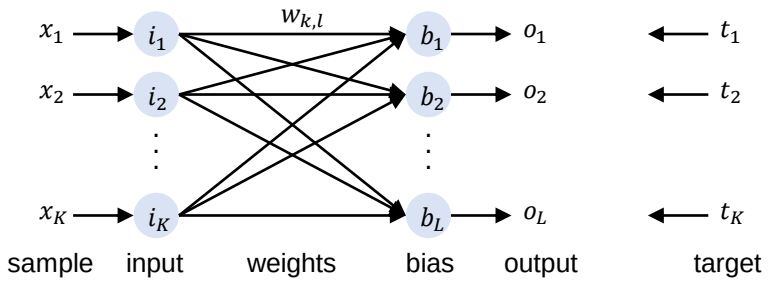


source: ttps://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html

- We first consider the original perceptron idea: in principle, it is a binary classifier mapping a real-valued input vector $\boldsymbol{x} \in \mathbb{R}^K$ to a binary output value $f(\boldsymbol{x})$:

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}^\top \boldsymbol{x} + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

where $\boldsymbol{w} \in \mathbb{R}^K$ are the weights and b is the bias. From this definition we derive that the perceptron is splitting the space with a hyperplane given by $\boldsymbol{w}^T \boldsymbol{x} + b$. In a more general setup, L perceptrons with weights $\boldsymbol{w}_l$ and bias b_l are connected to the K input value i_k and produce L binary output values o_l. We can visualize this general setup as follows:



$$\forall 1 \le l \le L: \; o_l = f\left(\sum_{k=1}^{K} i_k \cdot w_k + b\right)$$

with the binary step function

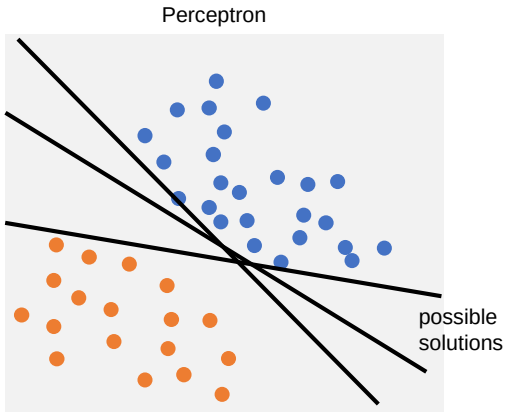
$$f(z) = \begin{cases} 1 & z > 0 \\ 0 & \text{otherwise} \end{cases}$$

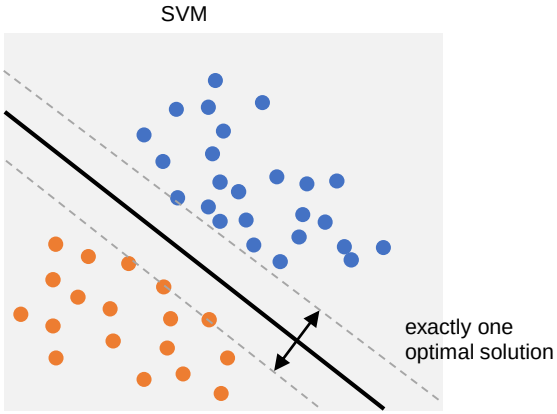
The learning algorithm is then as follows: (demo: https://www.cs.utexas.edu/~teammco/misc/perceptron/)

1. Initialize the weights $w_{k,l}^{(0)}$ and the biases $b_L^{(0)}$ with small random values. Set a learning rate $0 \le \alpha \le 1$
2. For each example $\boldsymbol{x} \in \mathbb{T}$, apply it to the perceptron, i.e., let $\boldsymbol{i} = \boldsymbol{x}$
   - Calculate that actual output: $o_l = f\left(\sum_{k=1}^{K} i_k \cdot w_{k,l} + b_l\right)$
   - Update the weights: $w_{k,l}^{(t+1)} = w_{k,l}^{(t)} + \alpha(t_l - o_l) \cdot i_{k,l}$ (i.e., only adjust if target≠output)
   - Update the bias: $b_l^{(t+1)} = b_l^{(t)} + \alpha(t_l - o_l)$ (i.e., only adjust if target≠output)

Convergence is only reached if the data set is linearly separable. Otherwise, the algorithm may fail completely. A number of variants address this later issue.

- Intuitively, the perceptron learning algorithm only adjust weights (and bias) if the target differs from the output. If the output is 0 but the target is 1, then weights and bias are incremented, otherwise they are decremented (assuming $x_l \geq 0$). We also note that the algorithm does not aim to optimize any objective function but merely is a heuristic approach to learn the weights. If data is separable, it converges to binary partition of the space with a hyperplane (one of many that partition the space).
- In contrast, the **support vector machine (SVM)** computes an optimal solution for the hyperplane that separates the sets and maximizes the margin (the distance of marginal points to the hyperplane). SVM even works if the data is not separable; it then finds a solution that minimizes the partitioning error. We are not considering here how SVMs are computed.

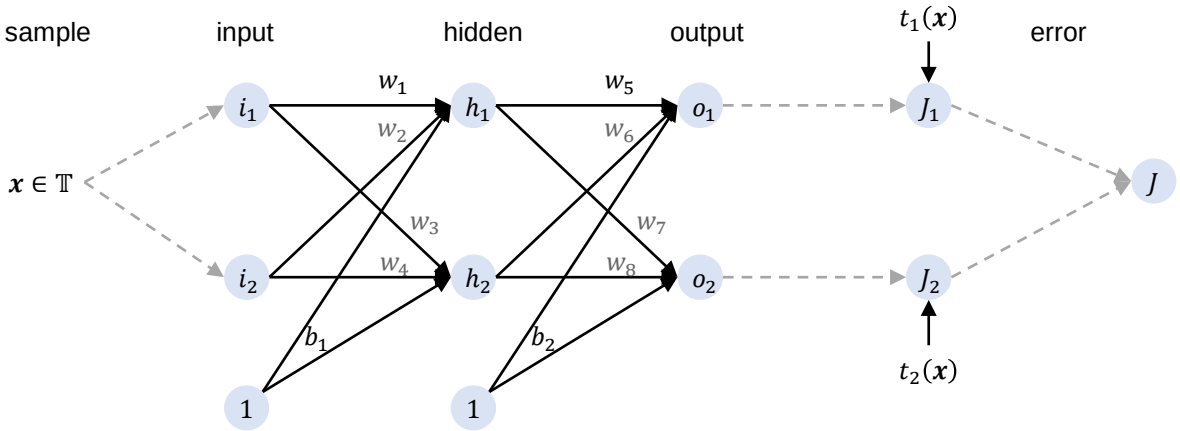




- In any case, a binary classifier can be used to learn multiclass outputs as well. The “one-vs-all” approach learns a binary classifier for each of the L classes to separate a class C_l from the rest. In other words, we use L perceptrons and the binary target vector $\boldsymbol{t}$ has $t_l = 1$ and all other components are 0. For prediction, the output with the highest value denotes the “winning” class. Alternatively, the “one-vs-one” strategy uses $L(L-1)/2$ perceptrons to separate two classes from each other learning the perceptrons individually. For prediction, the output with the highest value indicates the “winning” class.

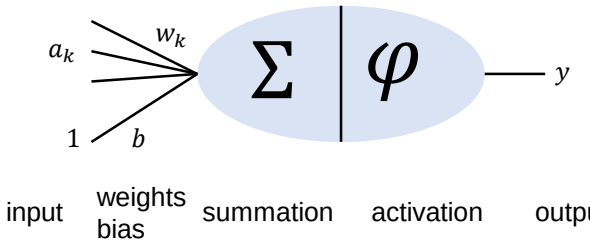
- The linear classification approach of SVM seems rather limiting (like for perceptron). However, SVM has the “kernel trick”: the idea is that data points are mapped to a higher dimensional space that enables better separability of the data by a hyperspace. The mapping to this higher dimensional space is typically non-linear. The “kernel-trick” now means that we do not explicitly compute the mapping to the high-dimensional space, but rather only compute the inner product between data points that is required for the SVM calculations. For instance, the kernel $K(\boldsymbol{x}, \boldsymbol{y}) = (1 + \boldsymbol{x}^{\mathrm{T}}\boldsymbol{y})^2$ with $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^2$ is an efficient way to compute the inner product of two mapped values $\varphi(\boldsymbol{x})$ and $\varphi(\boldsymbol{y})$ in a 6-dimensional space. With a Gaussian kernel $K(\boldsymbol{x}, \boldsymbol{y}) = \exp(-\gamma\|\boldsymbol{x} - \boldsymbol{y}\|^2)$ we obtain an infinite-dimensional mapping function φ.
- The “kernel trick” is often considered as a human intervention into the machine learning process. SVM classification works very well and is efficient but we need to design an appropriate kernel function for the problem at hand.

- **Multilayer networks** introduce a number of changes to the original perceptron
  - several “hidden” layers between input and output
  - different activation functions to “fire” a neuron, and not necessarily only binary output
  - objective functions to define an optimal state for all network parameters
  - a new algorithm to learn the weights (the so-called **backpropagation**)
- Let us start with a simple two-layer network to understand the fundamentals with a concrete example, and then we generalize the concepts to arbitrary shaped networks.



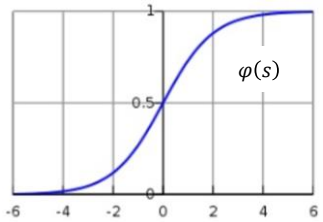
- The network consists of two input neurons i_1, i_2, two hidden neurons h_1, h_2 and two output neurons o_1, o_2. We have two (shared) biases, b_1 for the hidden neurons and b_2 for the output neurons. Note that we modeled the bias as a weight from a neuron that always has the state 1. $w_1, \ldots, w_8$ denote the weights on the connections. Even though we have 6 neurons, the connections are only from one layer to the next one and especially, there are no inter-layer connections or cycles. This is an important topological constraint that will simplify our learning algorithm. Finally, we added nodes to capture the training error: J_1 and J_2 measure the error between the first and the second target component $t_1(\boldsymbol{x})$ and the computed output of the network. J denotes the training error.

- **Feed-Forward:** given a data sample $\boldsymbol{x}$ from the training set $\mathbb{T}$, the network is computing the state of each neuron using a simple model:



$$s = \sum_k a_k \cdot w_k + b$$

$$y = \varphi(s) = \frac{1}{1 + e^{-s}}$$



We use s to indicate the result of the summation, and we employ the logistic activation function φ also known as soft step. With this, we can determine every state of a neuron, given the input $\boldsymbol{x} \in \mathbb{T}$:

$$\varphi(s) = \frac{1}{1 + e^{-s}}$$

$$i_1 = x_1 \quad \text{and} \quad i_2 = x_2$$

$$h_1 = \varphi(s_{h_1}) = \varphi(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1) \quad \text{and} \quad h_2 = \varphi(s_{h_2}) = \varphi(w_3 \cdot x_1 + w_4 \cdot x_2 + b_1)$$

$$o_1 = \varphi(s_{o_1}) = \varphi(w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) = \varphi(w_5 \cdot \varphi(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1) + w_6 \cdot \varphi(w_3 \cdot x_1 + w_4 \cdot x_2 + b_1) + b_2)$$

$$o_2 = \varphi(s_{o_2}) = \varphi(w_7 \cdot h_1 + w_8 \cdot h_2 + b_2) = \varphi(w_7 \cdot \varphi(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1) + w_8 \cdot \varphi(w_3 \cdot x_1 + w_4 \cdot x_2 + b_1) + b_2)$$

The calculations are straightforward. The term feed-forward denotes that we “feed” the data sample first into the input layer, and then forward the results from one layer to the next one. Each layer can be computed concurrently.

Later on, we will see different activation functions and also different approaches to connectivity and sharing of weights between subsequent layers. The principle model for neurons remain the same for most deep networks. We will also encounter special dropout neurons, that set input elements to zero with a certain probability to prevent overfitting of the network.

- **Error function:** we want to measure how well the network is able to predict the targets for all given data samples in the training set $\mathbb{T}$. As a starting point, we use the mean square error (MSE):

$$J(\boldsymbol{\theta}) = \frac{1}{|\mathbb{T}|}\sum_{\boldsymbol{x}\in\mathbb{T}} J(\boldsymbol{x};\boldsymbol{\theta}) = \frac{1}{2\cdot|\mathbb{T}|}\sum_{\boldsymbol{x}\in\mathbb{T}} \|t(\boldsymbol{x}) - o(\boldsymbol{x};\boldsymbol{\theta})\|_2^2$$

where $\boldsymbol{\theta}$ denotes the parameters of the network. In our example: $\boldsymbol{\theta} = (w_1, \ldots, w_8, b_1, b_2)$. Learning a network means finding parameters θ^* that minimizes the error function:

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} J(\boldsymbol{\theta}) = \frac{1}{2\cdot|\mathbb{T}|}\sum_{\boldsymbol{x}\in\mathbb{T}} \|t(\boldsymbol{x}) - o(\boldsymbol{x};\boldsymbol{\theta})\|_2^2$$

  - Due to the size of networks and the number of data items, it is generally not feasible to solve the equation in closed form. Instead, we use the gradient descent method to find a (local) optimum through an iterative approach. Let $\nabla J(\boldsymbol{\theta})$ be the gradient of $J(\theta)$ for the parameters θ of the network. The **gradient descent** method defines the learning strategy for the network:

    1. Choose an initial random vector for $\boldsymbol{\theta}^{(0)}$ and a learning rate $0 \leq \eta \leq 1$
    2. Repeat until $\left\|\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}\right\|_2^2 \leq \varepsilon$ or $t > t_{max}$
       - Compute gradient: $\Delta^{(t)} = \eta \cdot \nabla J\left(\boldsymbol{\theta}^{(t)}\right)$
       - Adjust parameters: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \Delta^{(t)}$

  - Gradient descent is relatively slow close to the minimum and often “zigzags” for poorly conditioned convex functions. In addition, for large-scale data sets and networks, gradient descent requires enormous computational and storage requirements to determine the gradient (which we can derive in closed form for the network as we will see later).

– Instead of gradient descent, neural network algorithms use the **stochastic gradient descent (SGD)** often in combination with a momentum method to prevent the afore mentioned zigzag issue. SGD approximates the true gradient of $J(\boldsymbol{\theta})$ with a single data sample (instead of over all data samples). As we will see with backpropagation, this allows us to quickly update the weights with minimal storage overhead. SGD still suffers from slow convergence especially towards the end of the iterations. Momentum is one method to accelerate the descent. We keep the gradient of the past iteration and re-apply some fraction γ of it in the descent:

1. Choose an initial random vector for $\boldsymbol{\theta}^{(0)}$, a learning rate $0 \leq \eta \leq 1$, and a momentum $0 \leq \gamma \leq 1$.
2. Repeat until $\left\|\boldsymbol{\theta}^{(t+1)} - \boldsymbol{\theta}^{(t)}\right\|_2^2 \leq \varepsilon$ or $t > t_{max}$
   - Randomly shuffle the training set $\mathbb{T}$
   - $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)}$
   - For each $\boldsymbol{x} \in \mathbb{T}$
     - Compute gradient: $\Delta = \gamma \cdot \Delta + \eta \cdot \nabla J(\boldsymbol{x}; \boldsymbol{\theta}^{(t+1)})$
     - Adjust parameters: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t+1)} - \Delta$
   - Increase γ

The momentum γ defines how long a previous gradient is still used. Generally, we start with $\gamma = 0.5$ and then increase it after the initial learning stabilizes to $\gamma = 0.9$ or even higher.

– The above algorithm defines the overall learning strategy. Each batch (step 2) runs against the entire training set and for each data samples, the weights and biases in the network are adjusted for each data sample. What remains to do is to compute the gradient $\nabla J(\boldsymbol{x}; \boldsymbol{\theta})$ for the current data sample and the current set of parameters of the network.

$$J(\boldsymbol{x}; \boldsymbol{\theta}) = \frac{1}{2}\|t(\boldsymbol{x}) - o(\boldsymbol{x}; \boldsymbol{\theta})\|_2^2 \qquad \nabla J(\boldsymbol{x}; \boldsymbol{\theta}) = ?$$

- **Gradient computation:** before we consider the backpropagation algorithm, let us re-consider our example network from the beginning with two input nodes, two hidden nodes, and two output nodes. For the stochastic gradient descent, we need to compute the gradient. Note that in our example, we have $\boldsymbol{\theta} = (w_1, \dots, w_8, b_1, b_2)$. The gradient is then given as the partial derivatives over $J(\boldsymbol{x}; \boldsymbol{\theta})$:

$$\nabla J(\boldsymbol{x}; \boldsymbol{\theta}) = \left(\frac{\partial J}{\partial w_1}, \dots, \frac{\partial J}{\partial w_8}, \frac{\partial J}{\partial b_1}, \frac{\partial J}{\partial b_2} \right) \qquad J(\boldsymbol{x}; \boldsymbol{\theta}) = J_1(\boldsymbol{x}; \boldsymbol{\theta}) + J_2(\boldsymbol{x}; \boldsymbol{\theta}) = \frac{1}{2} \cdot (t_1 - o_1)^2 + \frac{1}{2} \cdot (t_2 - o_2)^2$$

with given targets t_1 and t_2 for data sample $\boldsymbol{x}$, and o_1 and o_2 as given previously as a function of $\boldsymbol{x}$ and the weights $w_1, \dots, w_8$ and the biases b_1 and b_2.

– Let us start simple: consider w_5. It only occurs in o_1 but not in o_2. Thus the partial derivative is:

$$o_1 = \varphi(s_{o_1}) = \varphi(w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) \qquad o_2 = \varphi(s_{o_2}) = \varphi(w_7 \cdot h_1 + w_8 \cdot h_2 + b_2)$$

$$\frac{\partial J}{\partial w_5} = \frac{\partial}{\partial w_5} \left(\frac{1}{2} \cdot (t_1 - o_1)^2 + \frac{1}{2} \cdot (t_2 - o_2)^2 \right) = \frac{\partial}{\partial w_5} \left(\frac{1}{2} \cdot (t_1 - o_1)^2 \right) = (t_1 - o_1) \cdot \frac{\partial o_1}{\partial w_5}$$

$$\frac{\partial o_1}{\partial w_5} = \frac{\partial}{\partial w_5} \left(\varphi(s_{o_1}) \right) = \varphi(s_{o_1}) \cdot \left(1 - \varphi(s_{o_1}) \right) \cdot \frac{\partial s_{o_1}}{\partial w_5} = o_1 \cdot (1 - o_1) \cdot \frac{\partial s_{o_1}}{\partial w_5}$$

$$\frac{\partial s_{o_1}}{\partial w_5} = \frac{\partial}{\partial w_5} (w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) = h_1$$

all together:

$$\frac{\partial J}{\partial w_5} = (t_1 - o_1) \cdot o_1 (1 - o_1) \cdot h_1$$

$$\varphi(s) = \frac{1}{1 + e^{-s}}$$
$$\varphi' = \varphi \cdot (1 - \varphi)$$

Additional information — not part of the exams

– Similarly, we obtain the other partial derivatives $\frac{\partial J}{\partial w_6}$, $\frac{\partial J}{\partial w_7}$, $\frac{\partial J}{\partial w_8}$, and $\frac{\partial J}{\partial b_2}$. Altogether, we have:

$$\frac{\partial J}{\partial w_5} = (t_1 - o_1) \cdot o_1(1 - o_1) \cdot h_1 \qquad \frac{\partial J}{\partial w_6} = (t_1 - o_1) \cdot o_1(1 - o_1) \cdot h_2$$

$$\frac{\partial J}{\partial w_7} = (t_2 - o_2) \cdot o_2(1 - o_2) \cdot h_1 \qquad \frac{\partial J}{\partial w_8} = (t_2 - o_2) \cdot o_2(1 - o_2) \cdot h_2$$

$$\frac{\partial J}{\partial b_2} = (t_1 - o_1) \cdot o_1(1 - o_1) + (t_2 - o_2) \cdot o_2(1 - o_2)$$

We already note the recurring patterns in the calculations: the derivatives on the error function are multiplied by the derivative on the activation function and are multiplied by the derivative on the summation. For the gradients, we require the results (=states) from the feed-forward step and can the efficiently compute the gradients (see backpropagation).

– Now to the remaining partial derivatives (see next page how to derive for w_1):

$$\frac{\partial J}{\partial w_1} = h_1 \cdot (1 - h_1) \cdot x_1 \cdot \big((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7\big)$$

$$\frac{\partial J}{\partial w_2} = h_1 \cdot (1 - h_1) \cdot x_2 \cdot \big((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7\big)$$

$$\frac{\partial J}{\partial w_3} = h_2 \cdot (1 - h_2) \cdot x_1 \cdot \big((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_6 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_8\big)$$

$$\frac{\partial J}{\partial w_4} = h_2 \cdot (1 - h_2) \cdot x_2 \cdot \big((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_6 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_8\big)$$

$$\frac{\partial J}{\partial b_1} = h_1 \cdot (1 - h_1) \cdot \big((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7\big) +$$
$$h_2 \cdot (1 - h_2) \cdot \big((t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_6 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_8\big)$$

Additional information — not part of the exams

– Let us now consider w_1: we note that w_1 only occurs in h_1 which in turn is part of both o_1 and o_2.

$o_1 = \varphi(s_{o_1}) = \varphi(w_5 \cdot h_1 + w_6 \cdot h_2 + b_2)$ $\quad$ $o_2 = \varphi(s_{o_2}) = \varphi(w_7 \cdot h_1 + w_8 \cdot h_2 + b_2)$

$h_1 = \varphi(s_{h_1}) = \varphi(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1)$ $\quad$ $h_2 = \varphi(s_{h_2}) = \varphi(w_3 \cdot x_1 + w_4 \cdot x_2 + b_1)$

$$\frac{\partial J}{\partial w_1} = \frac{\partial}{\partial w_1}\left(\frac{1}{2} \cdot (t_1 - o_1)^2 + \frac{1}{2} \cdot (t_2 - o_2)^2\right) = (t_1 - o_1) \cdot \frac{\partial o_1}{\partial w_1} + (t_2 - o_2) \cdot \frac{\partial o_2}{\partial w_1}$$

$$\frac{\partial o_1}{\partial w_1} = \frac{\partial}{\partial w_1}\left(\varphi(s_{o_1})\right) = \varphi(s_{o_1}) \cdot \left(1 - \varphi(s_{o_1})\right) \cdot \frac{\partial s_{o_1}}{\partial w_1} = o_1 \cdot (1 - o_1) \cdot \frac{\partial s_{o_1}}{\partial w_1}$$

$$\frac{\partial o_2}{\partial w_1} = o_2 \cdot (1 - o_2) \cdot \frac{\partial s_{o_2}}{\partial w_1}$$

$$\frac{\partial s_{o_1}}{\partial w_1} = \frac{\partial}{\partial w_1}(w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) = w_5 \cdot \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial s_{o_2}}{\partial w_1} = w_7 \cdot \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial h_1}{\partial w_1} = \frac{\partial}{\partial w_1}\left(\varphi(s_{h_1})\right) = \varphi(s_{h_1}) \cdot \left(1 - \varphi(s_{h_1})\right) \cdot \frac{\partial s_{h_1}}{\partial w_1} = h_1 \cdot (1 - h_1) \cdot \frac{\partial s_{h_1}}{\partial w_1}$$

$$\frac{\partial s_{h_1}}{\partial w_1} = \frac{\partial}{\partial w_1}(w_1 \cdot x_1 + w_2 \cdot x_2 + b_1) = x_1$$

all together:

$$\frac{\partial J}{\partial w_1} = (t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 \cdot h_1 \cdot (1 - h_1) \cdot x_1 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7 \cdot h_1 \cdot (1 - h_1) \cdot x_1$$

$$\varphi(s) = \frac{1}{1 + e^{-s}}$$
$$\varphi' = \varphi \cdot (1 - \varphi)$$

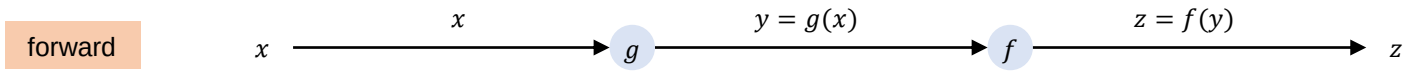
Additional information — not part of the exams

- Evidentially, it is possible to compute all partial derivatives for the gradient, but it seems tedious work to do so (and error prone). Can we do it simpler? Yes, we can. Backpropagation is an astonishingly simple scheme that computes the gradient starting at the error node and working back towards the input nodes. It does not provide us with the closed forms of the derivatives, but it computes the gradient avoiding multiple computations of the same sub-expressions.
  - Let us look again at the chain rule from calculus:

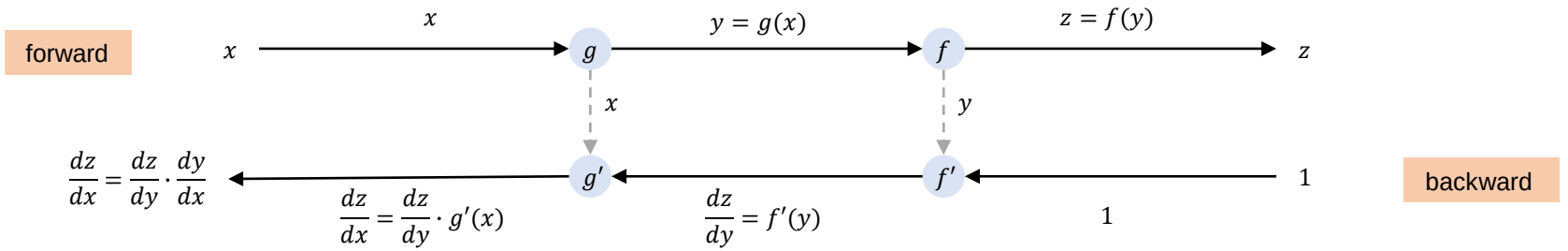
$$F(x) = f \circ g = f\big(g(x)\big) \qquad F'(x) = f'\big(g(x)\big) \cdot g'(x)$$

or in Leibniz notation with $z = f(y)$ and $y = g(x)$: $\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} = f'(y) \cdot g'(x)$

In graphical notation, we obtain the forward path to compute the composite function:



Now to compute the derivative $\frac{dz}{dx}$ for x we move backwards. We first compute $f'(y)$ and then multiply it with $g'(x)$. To this end, we need to keep track of intermediate results and use them on the back path to calculate the derivative:



Additional information — not part of the exams

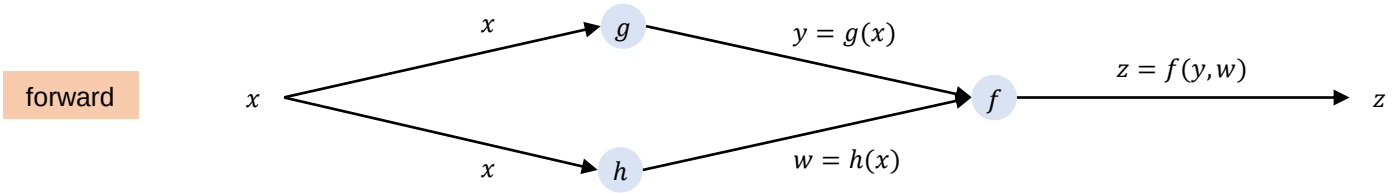
– Similarly, we can look at multivariable chain rules

$$F(x) = f(g(x), h(x)) \qquad F'(x) = f'(g(x), h(x)) \cdot g'(x) + f'\big(g(x), h(x)\big) \cdot h'(x)$$

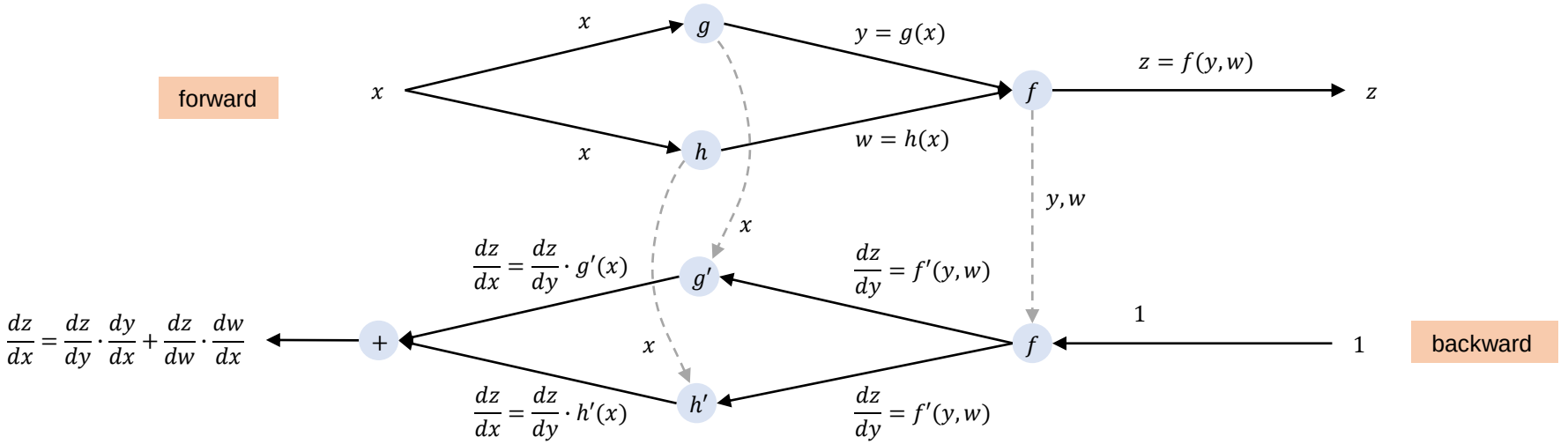
or in Leibniz notation with $z = f(y), y = g(x)$ and $w = h(x)$

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} + \frac{dz}{dw} \cdot \frac{dw}{dx} = f'(y, w) \cdot g'(x) + f'(y, w) \cdot h'(x)$$

In graphical notation, we obtain the forward path to compute the function:

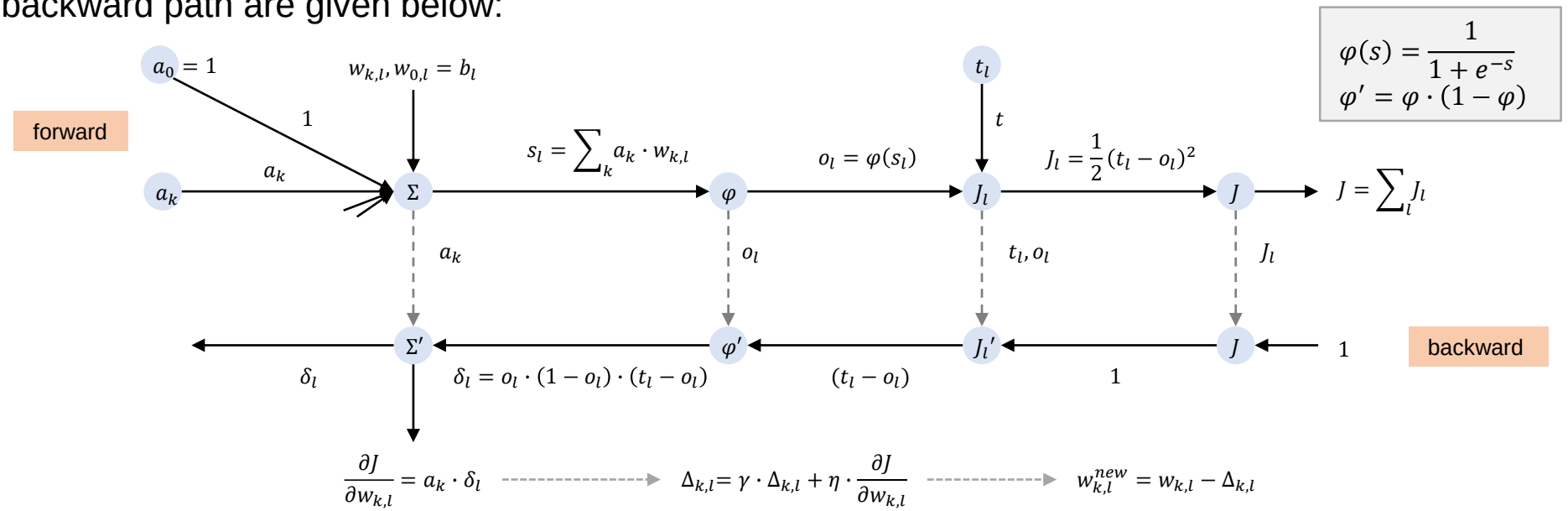


Now to compute the derivative $\frac{dz}{dx}$ for x we move backwards similarly as before:

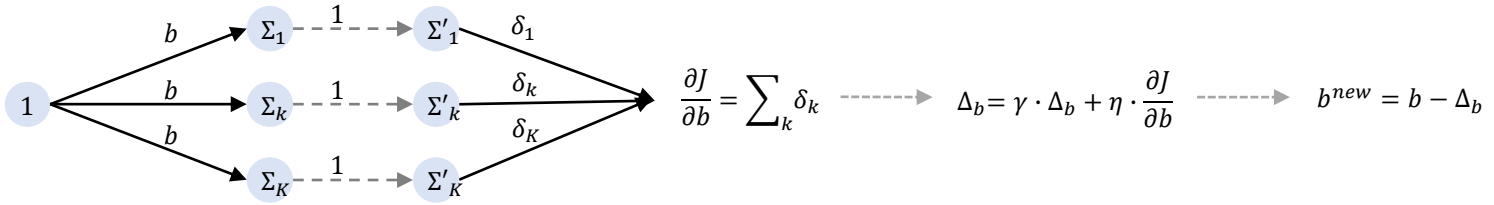


Additional information — not part of the exams

- Let us apply the chain rule to our neural network. Let start with the output neurons. To simplify the structure, we introduce a node a_0 which always has the state 1, and the weight $w_0 = b$ which represents the bias. All formulas become a bit simpler. The visualization for the forward and backward path are given below:

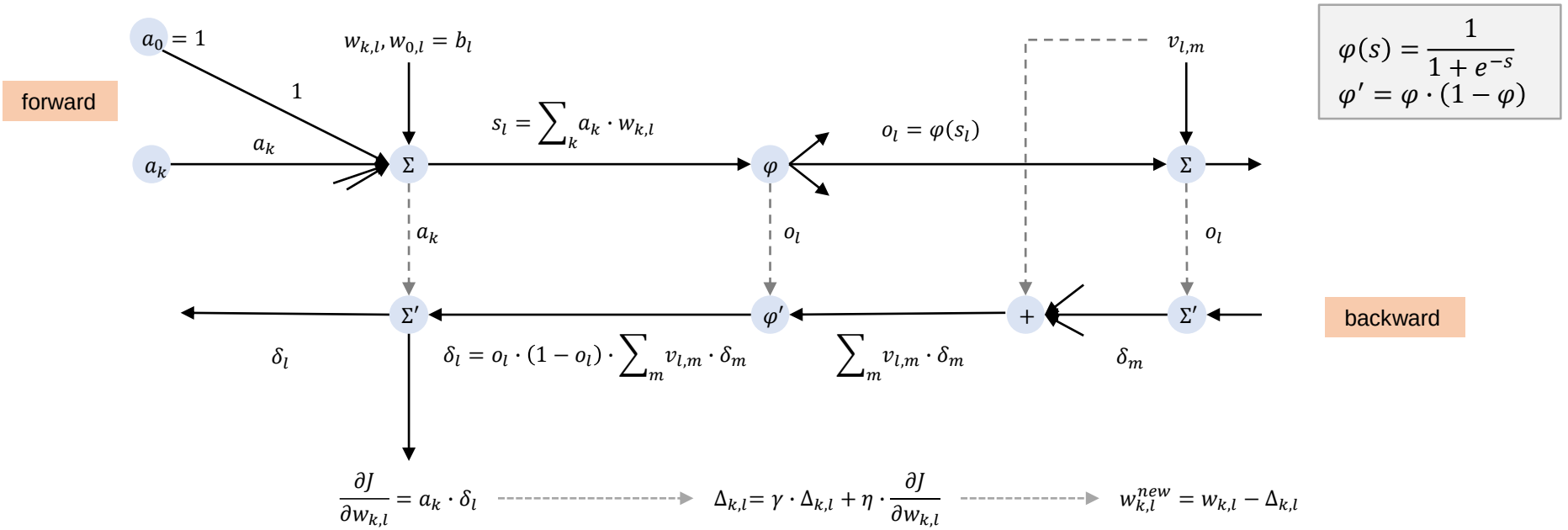


  - Every layer outputs the δ-values that are propagated back to the inputs and are used to adjust the parameters in every layer. Above, we used a separate bias b_l for each node. If we would share the bias across the layer like in the example, we need to simply sum up the deltas over the nodes using the same bias, i.e.:



Additional information — not part of the exams

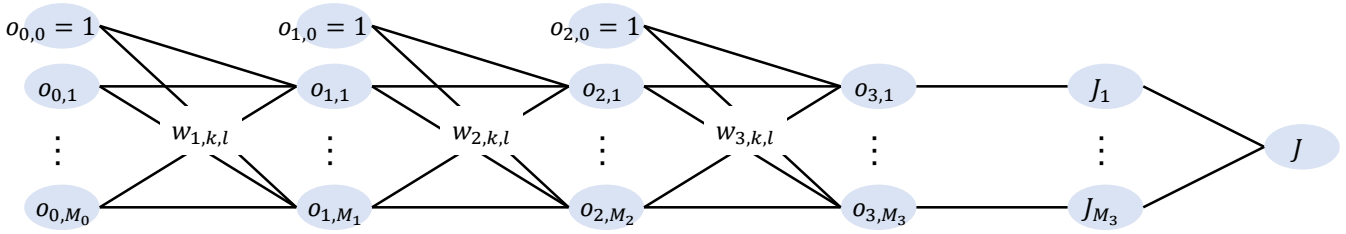
– Hidden layers are calculated similarly, however, there are L incoming edges from the subsequent layer during backpropagation. The visualization for the forward and backward path are as follows:



– Let us sum up the backpropagation algorithm: during the stochastic gradient descent, we search for the optimal parameters (weights, biases, etc.) of the network. To compute the gradient for these parameters with respect to an error function J, we first use the network in forward mode to predict the output with the current set of parameters. At the same time, we keep track of intermediate values that are required on the backward path. We then compute the error with regard to a single sample and propagate the partial derivatives backwards to the previous layers. At each layer, we compute the Δ-values for the weights to obtain new estimates for them. Note that the old weights are still required for the preceding layer to compute its partial derivative (see figure above, the (+)-node requires weights v_l, m from the subsequent layer).

Additional information — not part of the exams

- **Generic implementation of multilayer networks:** let us model a dense multilayer network. We assume N layers L_i and we denote L_0 to be the input layer and L_N to be the output layer. Each layer has M_i neurons with states $o_{i,k}$ with $0 \leq i \leq N$ and $0 \leq k \leq M_i$ whereby $o_{i,0} = 1$ (used for the bias). Further we use weights $w_{i,k,l}$ with $1 \leq i \leq N$, $0 \leq k \leq M_i$ and $1 \leq l \leq M_{i-1}$ to connect the l-th node of Layer L_{i-1} with the k-th node of Layer L_i. In addition, we keep track of the increments $\Delta_{i,k,l}$ for the computation of the gradients $\frac{\partial J}{\partial w_{i,k,l}}$.
  - Example with 3 layers:



  - Feed Forward is then given as:

1. Initialize $o_{o,k} = x_k$ from the current data sample $\boldsymbol{x} \in \mathbb{T} \subset \mathbb{R}^{M_0}$ with target $\boldsymbol{t} \in \mathbb{R}^{M_N}$
2. For each layer L_i with i iterating from 1 to N:
   - Compute $o_{i,k} = \varphi(\sum_l w_{i,k,l} \cdot o_{i-1,l})$ with a selected activation function φ for all $1 \leq k \leq M_i$
3. Compute $J_k = E_k(o_{N,k};\ t_k)$ with a selected error function E for all $1 \leq k \leq M_N$
4. Compute training error $J(x;\theta) = \sum_k J_k = E(o_{N,k}; t_k)$ for current sample

So far we have used the logistic activation function $\varphi(s) = \frac{1}{1+e^{-z}}$ and the mean square error (MSE) with $J(\theta) = \frac{1}{2 \cdot |\mathbb{T}|} \sum_{\boldsymbol{x} \in \mathbb{T}} \|t(\boldsymbol{x}) - o(\boldsymbol{x}; \boldsymbol{\theta})\|_2^2$ such that $E_k\left(o_{N,k}; t_k\right) = \frac{1}{2}\left(t_k - o_{N,k}\right)^2$. We will see further activation functions and error (or loss) functions in the deep learning section.

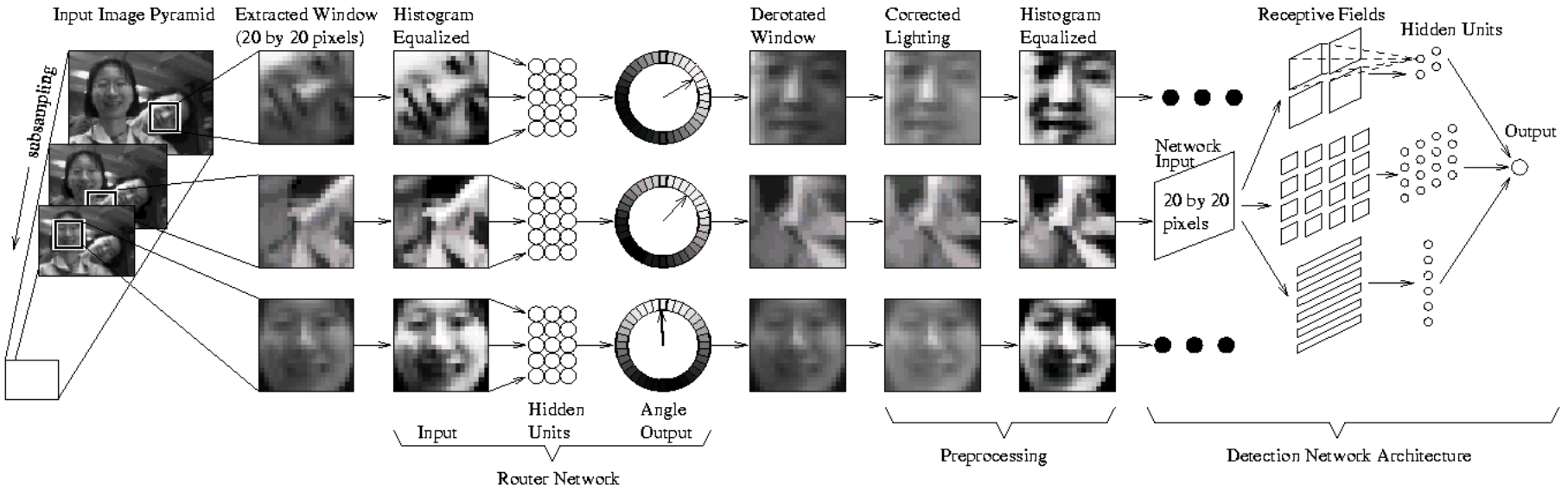
– Backpropagation is finally (e.g., with logistic activation function and mean square error):

1. Given target $\boldsymbol{t}$ and assume output $\boldsymbol{o}_N$ from feed forward step; assume learning rate η and momentum γ
2. Initialize $\Delta_{i,k,l} = 0$
3. Compute $\delta_{N,k} = \varphi'(o_{N,k}) \cdot E_k'(o_{N,k}; t_k) = o_{N,k} \cdot (1 - o_{N,k}) \cdot (t_k - o_{N,k})$ for all $1 \leq k \leq M_N$
4. For each layer L_i with i iterating from $N-1$ down to 1:
   - Compute $\delta_{i,k} = \varphi'(o_{i,k}) \cdot \sum_l w_{i+1,l,k} \cdot \delta_{i+1,l}$ for all $1 \leq k \leq M_\mathrm{i}$
   - Compute $\Delta_{i,k,l} = \gamma \cdot \Delta_{i,k,l} + \eta \cdot o_{i-1,l} \cdot \delta_{i,k}$ for all $1 \leq k \leq M_\mathrm{i}$
5. Update weights $w_{i,k,l} = w_{i,k,l} - \Delta_{i,k,l}$

Note: it is tempting to update the weights in the inner loop (step 4). However, we need the old weights in the preceding layer (next iteration in step 4) to compute $\delta_{i,k}$.

- While multilayer networks are still used in later layers in deep learning scenarios, the original approaches in 1980s and 1990s suffered from a number of issues (we will discuss them in the deep learning section). Essentially, the main issues involved numerical problems while computing the gradients (vanishing and exploding values) and the vast compute power necessary to learn moderate to large network. The smaller networks, on the other hand, did not work too well on typical classification scheme, and with SVM and kernel functions superior alternatives emerged.

- **Example: Face Detection**
  - Rowley, Baluja, Kanade [1998], Carnegie Mellon University, defined an elaborated algorithm for detecting faces at any scale and direction. To keep the neural network small, their approach was to first learn only normalized faces, and to then apply an exhaustive search for faces on images. The detection network is based on a 20x20 input network (preprocessed image window). In a first layer, 3 types of receptive fields are created: a) four 10x10 areas, b) 16 5x5 areas, and c) six overlapping 20x5 areas. Each area is fully connected to a hidden unit which is fully connected to an output. An output of 1 denotes a face, and an output of -1 denotes no face.
  - A second network (router network) was trained to estimate the direction of a face within a window. The 20x20 input network (preprocessed image window) is fully connected to hidden units which in turn are fully connected to 36 output values representing an angle of $i \cdot 36°$. The angle can be used in the predication phase to normalize the face before application of the detection network.



– Once trained, we can find faces in an image as follows: first, we build a pyramid of images by subsampling to smaller and smaller sizes. This allows us to find faces of different sizes. Then, a 20x20 windows is sliding across the image and for each location, the network tests whether the window contains a face. Due to the usage of normalized faces, the algorithm can return the location and direction of faces as well as estimating the position of the eyes.

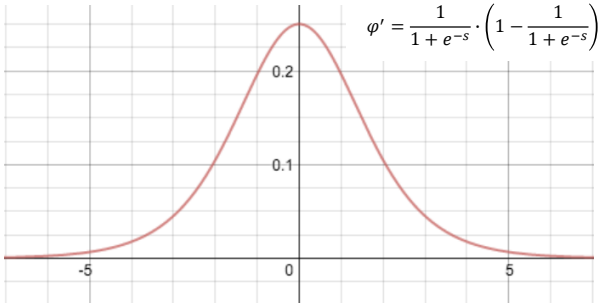
5.4.5 Deep Learning

- The second wave of neural network research died very quickly after discovering more structural issues with how the learning algorithm works. Even though it was proven that neural networks can learn any function, that theory often would not materialize in practice. Especially, it was observed that adding additional hidden layers does not lead to better results, and bigger networks were becoming increasingly instable to operate. The famous notion of **vanishing and exploding gradients** and the competition of support vector machine (SVM) with elaborated kernels drove a whole research field into a dead end. Only the Canadian government continued to fund neural network research: Geoff Hinton and team published in 2006 a paper on deep belief network where they showed how they could learn a network layer wise overcoming the issues of early backpropagation learning. In parallel, the massive amount of labeled data sets (a prerequisite to start learning) and the massive parallelism of GPUs greatly accelerated the success of what is know simply called deep learning (although the concepts are much older).
- Let us first consider the vanishing gradient problem. In the network of the previous section, we had a input layer, a hidden layer, and an output layer and were optimizing the networks parameters by minimizing a quadratic cost function. The backpropagation algorithm computes gradients and would update a weight on the first layer with:

$$\frac{\partial J}{\partial w_1} = (t_1 - o_1) \cdot o_1 \cdot (1 - o_1) \cdot w_5 \cdot h_1 \cdot (1 - h_1) \cdot x_1 + (t_2 - o_2) \cdot o_2 \cdot (1 - o_2) \cdot w_7 \cdot h_1 \cdot (1 - h_1) \cdot x_1$$

  The gradient is the sum of two multiplications, each with factors of the form $x \cdot (1 - x)$ due to the usage of the sigmoid activation function. Note that x stands for the outcome of a neuron after the activation function, hence $x = \varphi(s) = \frac{1}{1+e^{-s}}$. In addition, the multiplications include the weights of the last layer. If we add more hidden layers to the network, more factors of the form $x \cdot (1 - x)$ and more weights of later layers appear in the gradients of weights and bias of the first layer.

– The derivative of the sigmoid function $\varphi(s) = \frac{1}{1+e^{-s}}$ is plotted on the right hand side. We note that the maximum value is ¼ and that values quickly drop on both sides. If we initialize weights between 0 and 1, the gradient computation turns into a series of multiplications of small values yielding very small updates weights and biases even if they a significantly wrong. This requires a huge number of iterations to move weights and biases towards their optimal values, hence, learning is very slow and expensive.



$$(t_1 - o_1) \cdot \underbrace{o_1 \cdot (1 - o_1)}_{\leq 1/4} \cdot \underbrace{w_5}_{\leq 1} \cdot \underbrace{h_1 \cdot (1 - h_1)}_{\leq 1/4} \cdot x_1 \leq 1/16$$

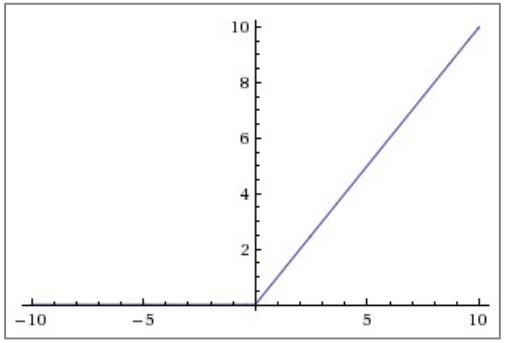
As a consequence, gradients are reduced to a fourth for each layer in the backpropagation making it very slow to train networks with lots of layers (GoogLeNet used ~20 layers).

– On the other hand, if we scale the weights and input values beyond the typical $[-1,1]$ range, the gradients will explode as we a now multiply several numbers larger than 1. With only a few layers, gradients become exponentially larger as we propagate back, and with that the weights and biases grow in absolute values, resulting in potentially even larger gradients in the next iteration. Several attempts for deeper networks failed due to instable gradient computations.

- Deep learning addressed these issues with backpropagation friendly activation functions (ReLu), improved architecture (convolution, pooling, inception modules, residual networks), and improved regularization techniques (dropout, ReLu, L1, L2). We consider some of these concepts subsequently.

- The **rectified linear unit (ReLU)** is a simple activation function replacing the sigmoid function used previously. There are now many alternative activation functions, but the ReLU marked an important step towards more stable gradient computations. It is defined as

$$\varphi(s) = \max(0, s)$$



The function is plotted on the right hand side. What is so special about this function? First, its is closer to the way biological neurons works while the sigmoid function (and its counterpart the hyperbolic tangent) were inspired by probability theory. Second, its gradient is either 0 or 1:

$$\varphi'(s) = \begin{cases} 0, & s < 0 \\ 1, & s \geq 0 \end{cases}$$

Hence, the gradients of the activation function do not accelerate the vanishing and exploding effects as described before. ReLU have become the standard activation function for deep learning despite some of the challenges that come with them:

– The output is no longer in the range $[0,1]$. If we train classifiers, how can we map the output of the last layer to class labels? The softmax function can be used to convert output values to class probabilities. It is often used together with the cross-entropy loss function to simplify gradient calculations as follows. Let o_k be the k-th output value, and y_k be the target label. Then:

$$p_k = \frac{e^{o_k}}{\sum_k e^{o_k}}$$

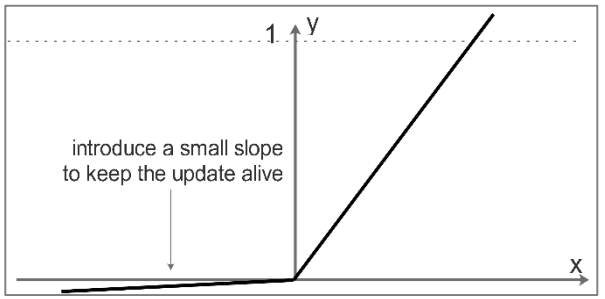
$$J(\boldsymbol{\theta}) = -\sum_k y_k \cdot \log p_k$$

J is defined as the cross-entropy loss function. $\boldsymbol{\theta}$ contains all parameters of the network, i.e., weights and biases.

$$\frac{\partial J}{\partial o_k} = p_k - y_k$$

that is simple!

– The derivative of the ReLU can become 0 which means that back propagation stops at this unit and predecessors are not adjusted. While some see this as a regularization of the network by thinning out the connections (much like neurons in the brain are also not fully connected), others are concerned that an initial selection of weights and biases may randomly close paths and the network can only slowly recover from that (if at all). Instead, a common extension is the **leaky ReLU** which is defined as (including its derivative):

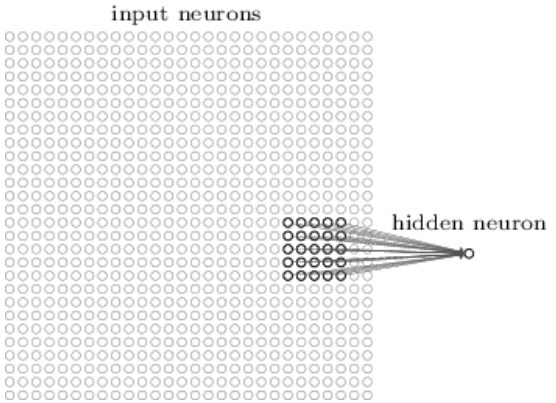


$$\varphi(s) = \begin{cases} 0.01 \cdot s, & s < 0 \\ s, & s \geq 0 \end{cases} \qquad \varphi'(s) = \begin{cases} 0.01, & s < 0 \\ 1, & s \geq 0 \end{cases}$$

The advantage is that the derivative is never becoming 0; it is small for negative values allowing a network to recover a closed path

- To overcome the vanishing and exploding gradient, deep learning improved the architecture of the network: instead of fully connected, cascading layers, deep networks uses convolution, pooling, inception, residuals, and regularizations to structure the network. Convolution, for instance, uses a few weights and biases that feed into several thousands output neurons. Hence, during backpropagation, even though the gradients may have become small, thousands of updates are summed up in one iteration. Regularizations, as another example, reduces the number of active connections. Similar to convolutions, this reduces the number of (active) parameters in the network making it more efficient to train and faster to learn. We look at these individual measure first in isolation and then put all together for a truly deep learning network.

- **Convolution**
  - So far, we considered layers that were fully connected with the previous layer. Each connection had its own weight, and neurons hat either their own bias or a shared bias.
  - In contrast, the visual perception of nature works with receptive fields that extract features from a spatial neighborhood. The fields work the same across the entire visual range. In the traditional learning, hence, images were pre-processed using different algorithms (Gaussian, Sobel, HOG). However, that also limited the ways a network can learn.



  - Deep learning introduced a new layer, the convolutional layer. As depicted above, it connects only a small spatial neighborhood (here 5x5 input neurons) to a hidden neuron. This occurs for all locations in the matrix, creating an identically sized hidden layer (using padding at the boundaries). The output of the neuron is given as:

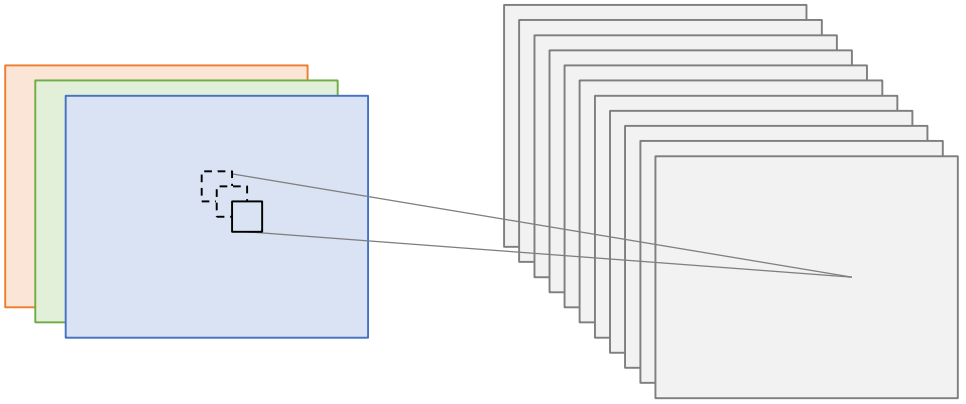
$$o_{i,j}(\boldsymbol{x}) = \varphi\left(b + \sum_{k,l} w_{k,l} \cdot x_{i+k,j+l}\right)$$

    An interesting aspect is that the weights $w_{k,l}$ and the bias b are shared across the neurons of the new layer. In fact, the above formula correspond to the convolution approach we have seen in the previous chapter (hence the name). Only, here we task the network to learn the best convolution for the task at hand.
  - In addition, we can define an arbitrary number of such filters within a single convolution layer. The output at the hidden neuron is then not only a single value, but a N-dimensional vector which can be used as the input for the next layer.

– As the output of a convolution can be N-dimensional, so can the input be an M-dimensional vector. In fact, when processing images, we typically start with three channels. These three channels can then be mapped through convolution to an arbitrary number N of output features (N is often called the depth of the output). The more general convolution functions is hence a mapping of an M-dimensional input vector $\boldsymbol{x}$ to an N-dimensional output vector $\boldsymbol{o}$. For a pixel location (i, j), we obtain:

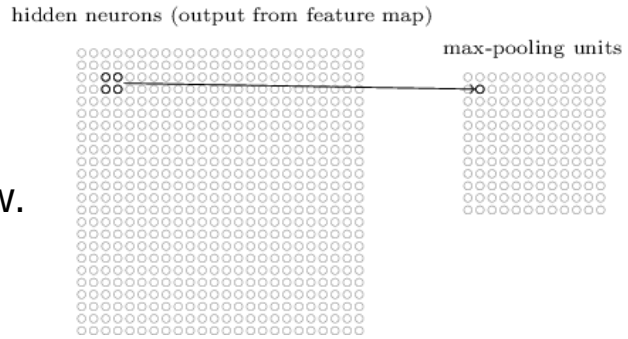
$$o_{i,j,n}(\boldsymbol{x}) = \varphi\left(b_n + \sum_{k,l,m} w_{k,l,m,n} \cdot x_{i+k,j+l,m}\right)$$

For example, let us assume a 5x5 convolution on three ($M = 3$) input channels, and we want to convolute to $N = 20$ output feature. The above formula contains shared biases b_n for each output feature $1 \leq n \leq N$, and shared weights $w_{k,l,m,n}$ for each of the 5x5 positions of the window, for each channel $1 \leq m \leq M$ and each output feature $1 \leq n \leq N$. Hence, we have 20 biases and 5x5x3x20=1500 weights. The shared parameters are then used for all pixel locations in the image. If we started with a 256x256 input image with 3 channels, the output of the convolution is now a 256x256x20 arrays. Interestingly, we do not need to map the color spaces as the network now can also learn the best linear combination of the channels.



- The special case of a **1x1 convolution** is often used to reduce the dimensionality of the input values. Assume we want to learn a 5x5 convolution with 20 output features and we have 20 input features: we would need to learn 5x5x20x20=10'000 weights and 20 biases (in total 10'020 parameters). A 1x1 convolution can reduce the number of parameters to learn as follows:
  - We can first apply a 1x1 convolution to generate 3 output features (from the 20 input features). We require 1x1x20x3=60 weights and 3 biases for this layer (63 parameters in total).
  - We then feed the 3 features from the 1x1 convolution into a 5x5 convolution with 20 output features. We require 5x5x3x20=1'500 weights and 20 biases (1'520 parameters in total)
  - Overall, the new network structure has 1'583 parameters compared to the 10'020 with the naïve, straightforward mapping.
- An interesting aspect of convolution is that its complexity (number of parameters) is independent of the input size of the network. However, computational complexity (forward and backward steps) depend on the number of input values. For instance, an input sizing for 256x256 is 4 times faster than for a 512x512 sizing. If images are the input, the typical approach is to scale them down to a reasonable size that can be fed into the network. We will see later techniques to deal with scale variance, e.g., recognizing objects at different scales.
- **Strides:** convolution uses a sliding window which is applied at each location to compute an output value. In addition, it is also possible to define how far apart two subsequent windows must lie. A stride of (2,2) means that only every other value in both dimensions is used as the starting location of the window. Thus, only half as many rows and columns are created in the output. Strides can be used to reduce the initial size of the network. A (2,2) stride will lead to 4 times less output neurons. For images, this allows to scale down the size and compute features at various scales.

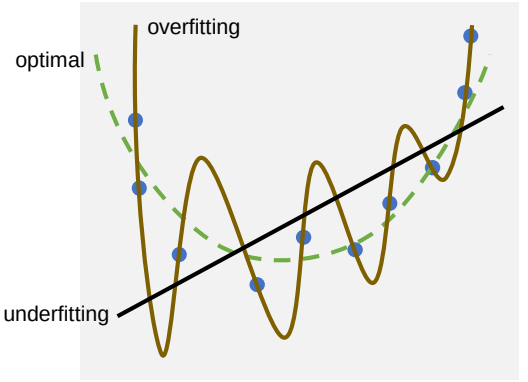
- Convolution layers are often followed by **Pooling Layers**. Pooling reduces the number of neurons and thus simplify the overall information.
  - A pooling layer is again a spatially organized structure. It summarizes the values of a window in the previous layer. For example consider the picture on the right hand side: a 2x2 max-pooling layer outputs the maximum value of the 2x2 window. If we additionally use a stride of (2,2), this reduces the "feature map" by 4 times. If the input consists of multiple channels, then the pooling operator is applied at each channel individually. Here, we do not apply an activation function:



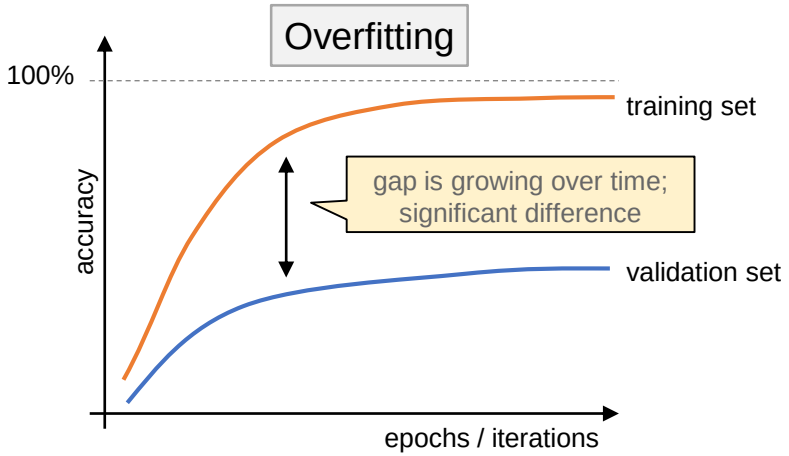
$$o_{i,j,n}(\boldsymbol{x}) = \max_{l,k} x_{i+k,j+l,n}$$

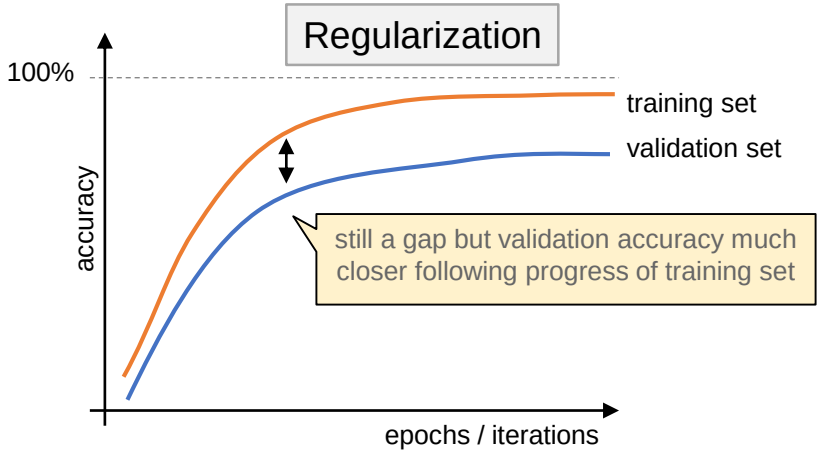
  - Next to max pooling, other summarization functions are possible. Typical examples include average pooling and L_2-Norm pooling.
  - In deep learning, for instance image object recognition, pooling layers are an important control mechanism to reduce the spatial size of the representation and with that the number of parameters in the network model. This not only greatly reduces the amount of computation but also reduces the risk of overfitting. Recall that the best model is the simplest one among equally good methods. Also note that pooling only reduce spatial dimensions if the stride is larger than 1. It does, however, not reduce the number of features (depth). For that, a 1x1 convolution is required as described before.

- **Regularization** is an important element in deep learning to prevent overfitting to the training data.
  - As we discussed earlier, overfitting occurs if the model has too many parameters and hence memorizes the data rather than generalizing rules from it. The picture on the right hand shows a simple example of what overfitting means. While the models on the right side may use dozens of parameters, a deep neural network can have several millions of parameters. Hence, how do we prevent the network from simply memorizing the input to target mapping, and how can we detect an overfitting problem.



  - Overfitting is the lack of generalization and will become evident if we apply a trained to new data items that were not used during training. The validation set can be used to detect overfitting. Overfitting can be recognized as follows:
    - Almost perfect accuracy for the training set at the end of the learning
    - Significant lower accuracy for the validation set at the end of the learning
    - The gap between training accuracy and validation accuracy is growing over the learning time





– We have several options for regularization
  - **Adjust the network structure** and reduce the number of parameters—not really an option given that we want to learn complex tasks. The success of small networks was rather limited.
  - **Expand the training set**—not always feasible, but we can modify and alter the existing data set. For instance, small rotations, varying brightness, adding noise, Gaussian filters, etc. With a few such modifications, we can create 10 to 100 times more training data without any additional labelling costs.
  - **Adjust the cost function** to prefer simpler models. A simple method is to add a penalty to the cost functions for the use of large weights. Smaller weights (preferably 0) reduce the complexity of the model. This way we can balance overfitting to the training with a penalty for more complex models. Our cost function looks now as follows (L2 regularization):

$$J_{reg}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \frac{\lambda}{2 \cdot |\mathbb{T}|} \sum_i w_i^2$$

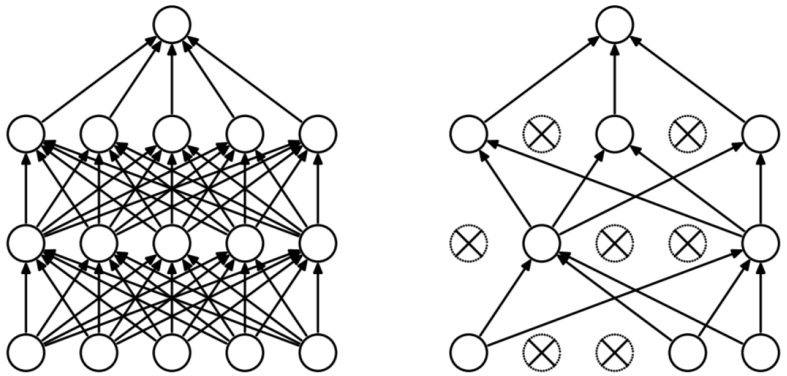
With $|\mathbb{T}|$ being the number of training samples and $\lambda > 0$ the regularization parameter. Note that we only add penalties for the weights but not for the biases. With this, we have a new update for w_i during back propagation. Let Δ_i be the update for w_i without regularization, then:

$$w_i^{(t+1)} = \left(1 - \frac{\eta\lambda}{|\mathbb{T}|}\right) \cdot w_i^{(t)} - \Delta_i$$

Regularization adds a weight decay factor $\left(1 - \frac{\eta\lambda}{|\mathbb{T}|}\right)$ for each weight, making them gradually smaller unless the gradient compensates enough to increase weights in the learning step. This was shown to greatly reduce the risk of overfitting.

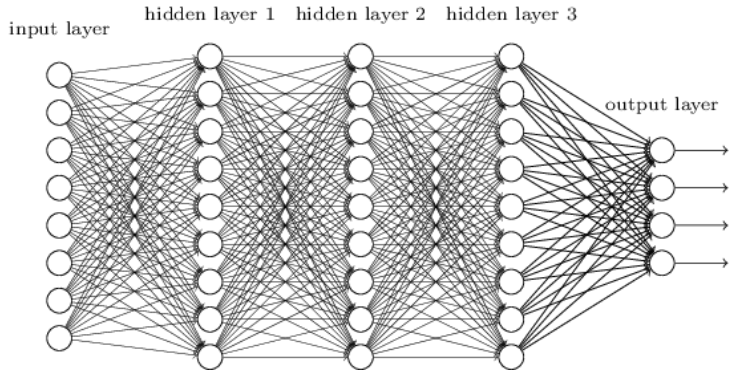
- The **Dropout technique** heuristically adjust the network structure during the learning phase. At any point in time during the learning phase, only parts of the network are active (with a random selection of nodes). This selection can change over time:
  - At each training step, nodes are dropped out with a probability of $1 - p$. Over the learning time, different sets of active nodes learn the training example
  - Feed forward: if a node is dropped out, its output value is set to 0. We keep weights and biases as the node may become active in a subsequent training step
  - Back propagation: if a node is dropped out, it does no longer propagate changes. The weights of connection to/from such a node do not receive an update.
  - The final model for prediction uses all nodes but compensates their weights with $(1 - p)$.

  We can interpret the dropout technique as learning many different networks at the same time. Finally, we combine all the individual networks into a single, bigger network. This helped with overfitting as each individual subset of the network has adapted differently to the training set. By "averaging" the networks for prediction, the impact of overfitting in one such sub-network is evened out the other sub-networks (which may have overfitted other aspects of the training set)

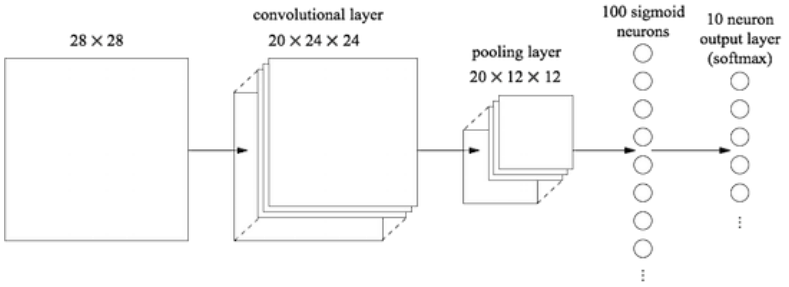


(a) Standard Neural Net (b) After applying dropout.

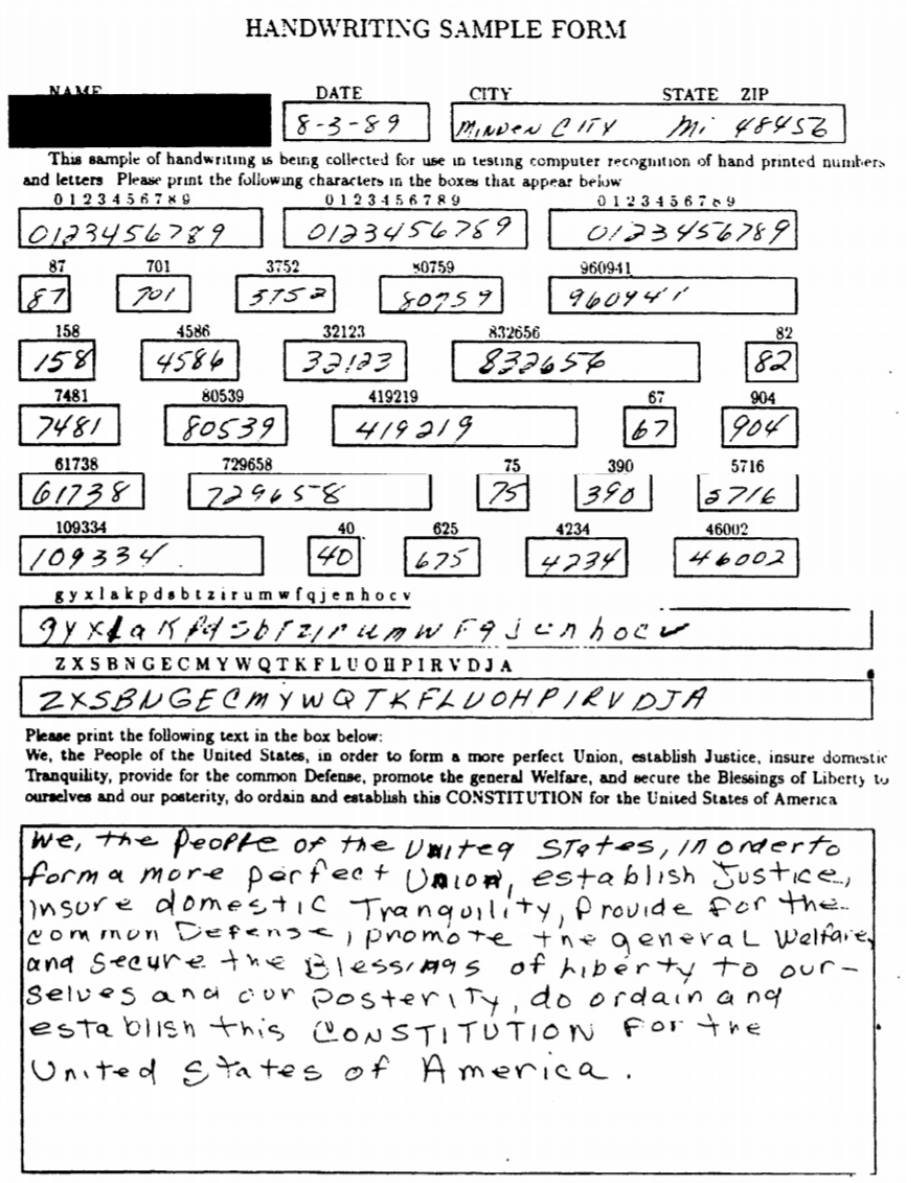
- **Putting all together**
  - Let us start with a simpler example: the MNIST database (see next page) consists of 28x28 images depicting hand written digits (0, 1, 2, …, 9)
  - The conventional approach with neural network used fully connected hidden layers like in the picture on the top right. Its performance was ok but methods like SVM and k-NN classification proved to be better.
  - The deep learning approach: use of convolution and pooling greatly improved performance. The picture on the bottom right show a possible architecture. The first 5x5 convolution produces 20 features with a ReLU activation (here, no padding is applied hence the size of the network reduces to 24x24). A subsequent 2x2 max-pooling layer reduces the spatial dimension to 12x12 (with 20 features). These 12x12x20=2880 elements are fully connected to 100 neurons. Finally, a softmax layer reduces the 100 neurons to 10 classes. The output neuron with the highest value denotes the class for prediction.



deep learning



- The original black and white images from NIST were size normalized to fit in a 20x20 pixel box while preserving their aspect ratio. The resulting images contain grey levels as a result of anti-aliasing. The images were centered in a 28x28 image by computing the center of mass of the pixels and moving the 20x20 image.
- The data set consists of 60'000 training items and 10'000 test items. The algorithms must learn a prediction method to map an image to one of the 10 classes 0, 1, 2, …, 9. The error rate is computed against the test data.
- The best method currently (a convolutional network) has an error rate of 0.23%. It is noteworthy to comment that some of the wrongly labelled images are also a challenge for humans to read correctly.


HANDWRITING SAMPLE FORM

NAME [redacted] DATE 8-3-89 CITY MINDEN CITY STATE MI ZIP 48456

This sample of handwriting is being collected for use in testing computer recognition of hand printed numbers and letters. Please print the following characters in the boxes that appear below

0 1 2 3 4 5 6 7 8 9

0123456789 0123456789 0123456789

87 701 3752 80759 960941

158 4586 32123 832656 82

7481 80539 419219 67 904

61738 729658 75 390 5716

109334 40 625 4234 46002

g y x l a k p d s b t z i r u m w f q j e n h o c v

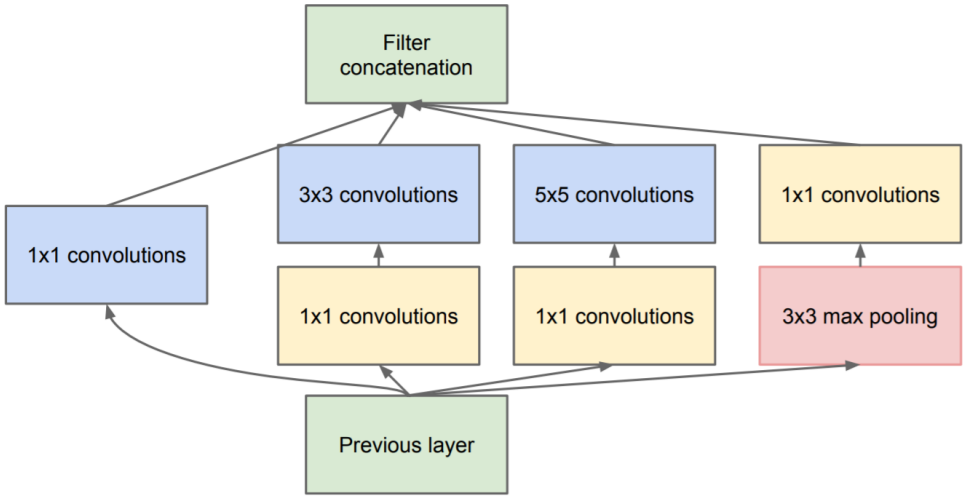
ZXSBNGECMYWQTKFLUOHPIRVDJA

Please print the following text in the box below:
We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America

We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of liberty to ourselves and our Posterity, do ordain and establish this CONSTITUTION for the United States of America.

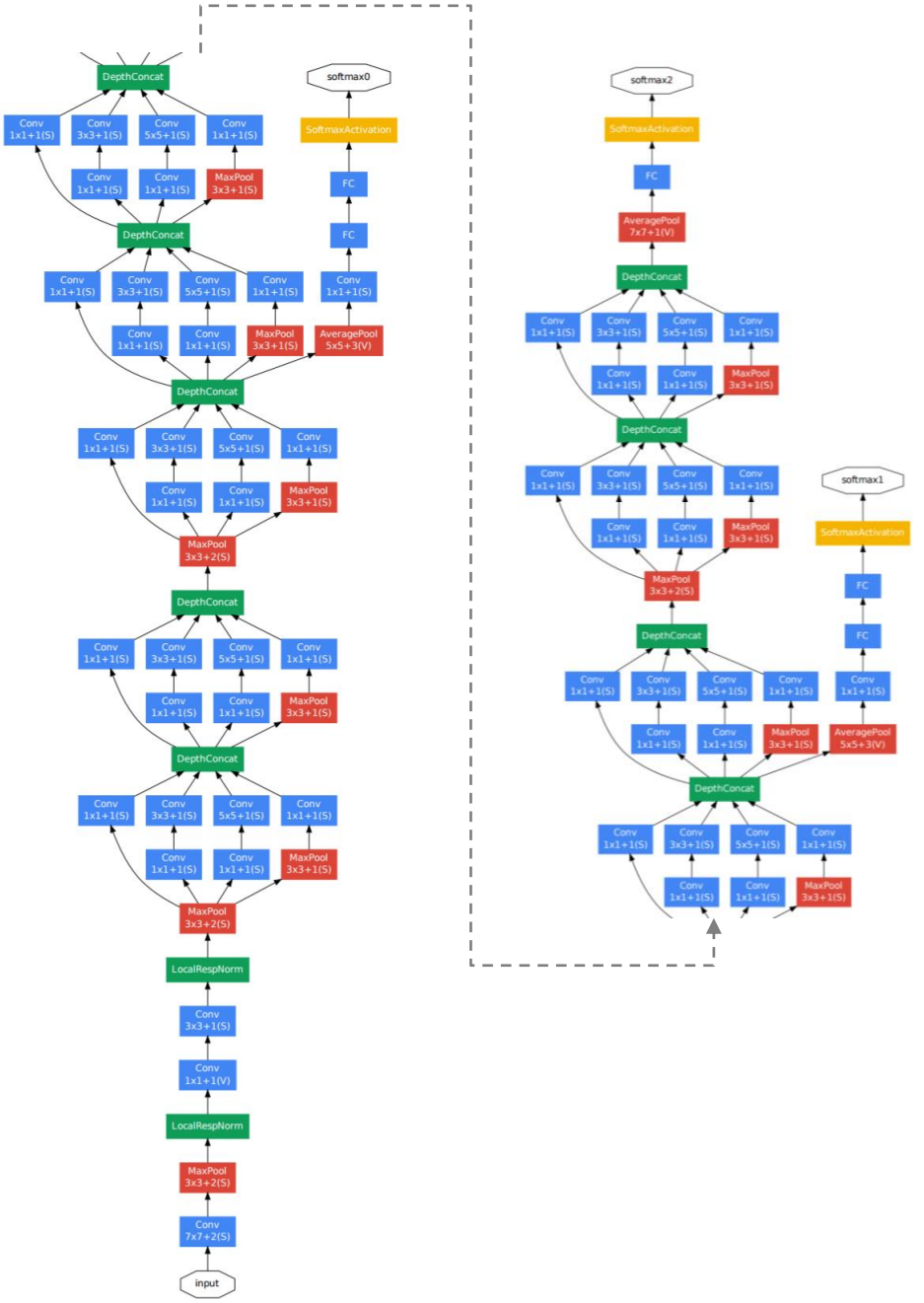
- List of further datasets for machine learning
  - https://en.wikipedia.org/wiki/List_of_datasets_for_machine_learning_research

- **GoogleLeNet for image classification**
  - GoolgLeNet was the winner of the ILSVRC 2014 Classification Challenge. The contest consisted of 500k images with object labeling in 200 classes.
  - A key ingredient of their network architecture included the use of inception modules which are building blocks for the network as shown below:
    - The inception module applies different operators on the output of a previous layer. In the example below, 1x1, 3x3, 5x5 convolutions and a 3x3 max pooling are all applied in parallel. Their output is then concatenated to produce the output features. The idea is that the network should learn itself, which of the operator works best for certain scenarios.
    - To control the complexity of the model, 1x1 convolutions (marked in yellow) are added to reduce the number of features. As previously discussed, this greatly helps to reduce the computational complexity of a 3x3 or 5x5 convolution.

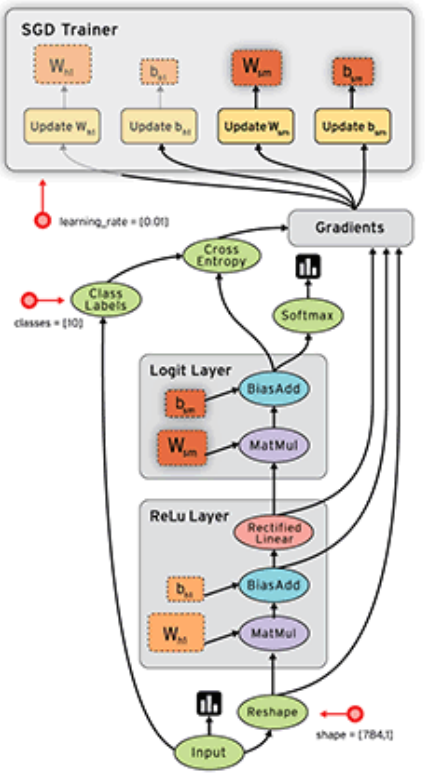


- The full architecture of GoogleLeNet for image classification
  - Input: 224x224 RGB images

Type	size/stride	output	#params	#ops
convolution	7×7/2	112×112×64	2.7K	34M
max pool	3×3/2	56×56×64		
convolution	3×3/1	56×56×192	112K	360M
max pool	3×3/2	28×28×192		
inception (3a)		28×28×256	159K	128M
inception (3b)		28×28×480	380K	304M
max pool	3×3/2	14×14×480		
inception (4a)		14×14×512	364K	73M
inception (4b)		14×14×512	437K	88M
inception (4c)		14×14×512	463K	100M
inception (4d)		14×14×528	580K	119M
inception (4e)		14×14×832	840K	170M
max pool	3×3/2	7×7×832		
inception (5a)		7×7×832	1072K	54M
inception (5b)		7×7×1024	1388K	71M
avg pool	7×7/1	1×1×1024		
dropout -40%		1×1×1024		
linear		1×1×1000	1000K	1M
softmax		1×1×1000		

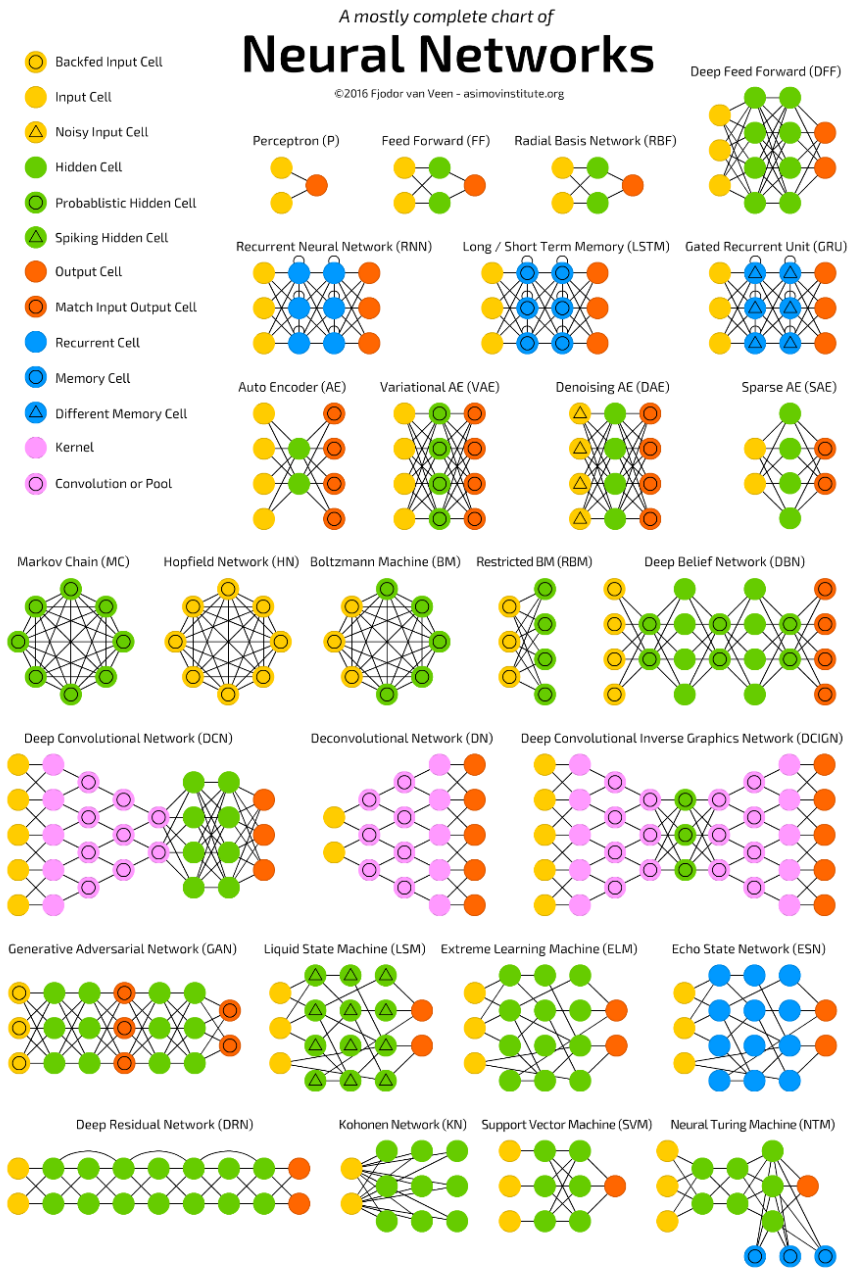


- **Tensorflow**
  - Tensorflow was developed by the Google Brain team, initially for Google internal use only. But meanwhile the framework is openly available under Apache 2.0 license and provides a simple to use Python programming front end to its core.
  - The term tensor stands for an arbitrary dimensional array holding the data values (often float32).
  - Tensorflow has two elements
    - Nodes are operators on input tensors and produce an output tensor
    - Data edges combine nodes and connect outputs with inputs
  - The Python front-end provides a simple way of building these graphs based on constants, variables and a rich set of defined operators. In the context of deep learning, most known methods have been implemented into tensorflow allowing for an efficient way of learning and applying a network
  - Another aspect of tensorflow is the distributed execution of the graph and the support for CUDA (GPU based operations) and parallel execution of operations. The largest networks can span hundreds of machines and can run against thousands of CUDA cores accelerating computations of large graphs. All this is transparent to the end-user, i.e., the user only must define the graph and tensorflow considers the fastest way to compute the graph.
  - For more information see: www.tensorflow.org



- In this chapter, we only looked at deep learning for spatial data sets (images, videos). But there is a great number of further architecture extensions to support, for instance, natural language processing, memorization of facts and data, and so on.
- The Asimov Institute published in 2016 a map outlining the neural network zoo

  http://www.asimovinstitute.org/neural-network-zoo/



5.5 References

- Papers
  - Quinlan, J. R. *Induction of Decision Trees*. Mach. Learn. 1, 1 (Mar. 1986), 81–106
  - Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.
  - X. Wu, V. Kumar, J.R. Quinlan, et al., *A survey paper on Top 10 Algorithms in Data Mining,* Knowledge and Information Systems, 14(1), 2008.
  - Lavner, Y and Ruinskiy, D, *A Decision-Tree-Based Algorithm for Speech/Music Classification and Segmentation*, EURASIP Journal on Audio, Speech, and Music Processing, 2009.
  - Diego Castan, Alfonso Ortega, Eduardo Lleida, *Speech/Music classification by using the C4.5 decision tree algorithm*, FALA 2010 VI Jornadas en Tecnología del Habla and II Iberian SLTech Workshop
  - C. Szegedy et al, Going Deeper with Convolutions, *Computer Vision and Pattern Recognition (CVPR), 2015.*
- Books
  - Mitchell, Tom M. *Machine Learning*. McGraw-Hill, 1997.
  - M. Nielsen, Neural Networks and Deep Learning, free online book, Dec 2017.
  - I. Goodfellow, Deep Learning (Adaptive Computation and Machine Learning series), 2016. Free online version available at: http://www.deeplearningbook.org
- Software
  - Tensorflow, Apache 2.0, https://www.tensorflow.org/
  - Scikit-learn, BSD, http://scikit-learn.org/
  - Online Neural Network: http://playground.tensorflow.org/