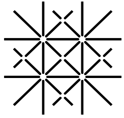


Universität Basel

Computer Science / 15731-01 / 2025

Multimedia Retrieval

Additional Material
Not part of the exam

Chapter 99: Machine Learning Methods

99.1 Machine Learning Process
99.2 Neural Networks
99.3 Clustering
99.4 Naive Bayes
99.5 Hidden Markov Models
99.6 Literature

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

99.1 Machine Learning Process

- In this chapter, we will explore a range of machine learning (ML) methods, starting from simple methods like Naïve Bayes to more advanced techniques like transformers for language processing. While the focus of this course is on retrieval techniques, understanding and correctly applying ML methods are crucial. Mastering the retrieval problem requires a comprehension of the strengths and limitations of these underlying methods. Therefore, we cover specific ML methods here to lay that foundation. The final chapter of this course provides a comprehensive description of these methods, rather than cluttering them throughout all chapters.
- In this introductory section, our emphasis is on the machine learning process in general. We introduce key learning concepts and discuss potential pitfalls such as underfitting and overfitting, which can hinder the successful application of ML methods. In modern data science, the concept of MLOps has gained significance aiming to structure and organize the data preparation, training, deployment, and operations of ML functions more effectively.
- In his 1997 book, "Machine Learning," Mitchell defined the machine learning problem as follows:

  > 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]

  - Mitchell considers the task to be anything that we want the system to perform, ranging from simple classification tasks to complex scenarios like self-driving cars, as we will explore with examples in the following pages.
  - To evaluate system performance for a given task, Mitchell introduces the notion of a performance measure. This measure not only helps in improving the system's task performance but also allows us to compare two systems. We can use measure such as accuracy or mean squared error to model performance.
  - Lastly, the term "experience" refers to the input provided to the system and its ability to utilize that input to enhance its performance. In supervised learning, we provide numerous examples to demonstrate how to perform a task correctly. In reinforcement learning, we offer feedback through a reward function, guiding the system towards better models. In unsupervised learning, we provide only data without labels or a reward function. The system must learn to describe the underlying distribution, either by clustering objects or detecting anomalies.

99.1.1 Tasks

The following provides a summary of the most common learning task. It's important to note that there are many more learning tasks in various domains where machine learning is applied.

- **Classification**: The task involves mapping input features to a set of K categories. Typically, this means finding a function f that maps an M-dimensional vector $\boldsymbol{x}$ to a category represented by a discrete value y. Another variant of classification involves assigning a probability distribution $P(y)$ over all classes y, which sum up to 1 over all classes y. Applications of classification include object recognition in images, text categorization, spam filtering, handwriting and speech recognition, credit scoring, pattern recognition, and more.

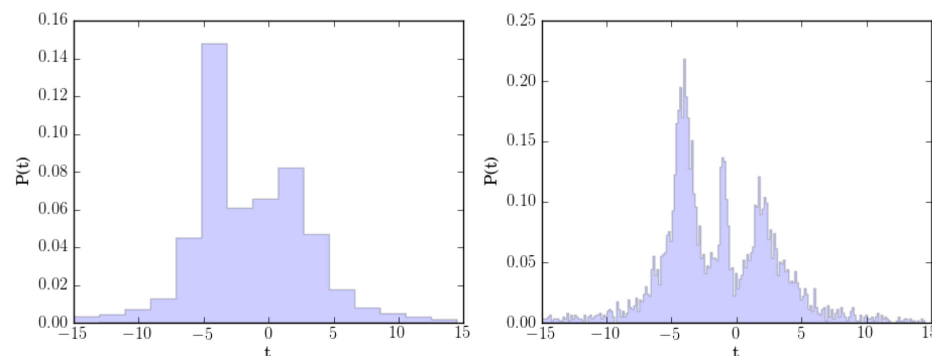
Sample	fixed acidity	volatile acidity	citric acid	pH	alcohol	quality
#1	8.5	0.28	0.56	3.3	10.5	7
#2	8.1	0.56	0.28	3.11	9.3	5
#3	7.4	0.59	0.08	3.38	9	4
#4	7.9	0.32	0.51	3.04	9.2	6
#5	8.9	0.22	0.48	3.39	9.4	6

- **Classification with missing input:** This task is similar to classification but allows for missing input values. Instead of a single function f, a set of functions is required to map different subsets of inputs to a category y (or a distribution $P(y)$). Alternatively, learning probability distributions over relevant features and marginalizing out the missing ones can be a better approach. All tasks have a generalization that accommodates missing inputs.

- **Regression**: The task involves predicting a numerical value based on the input features. The learning algorithm must find a function f that maps an M-dimensional vector $\boldsymbol{x}$ to a numeric value. Unlike classification, regression aims to produce a real number as the output and does not provide distribution functions over all possible values. Applications of regression include predictions/extrapolations to the future, statistical analysis, algorithmic trading, expected claim estimation in insurance, financial risk assessment, cost restrictions, budgeting, data mining, pricing (and its impact on sales), and correlation analysis.

- **Clustering** divides a set of inputs into groups. Unlike classification, the number of groups is not known in advance, and the machine learning algorithm must discover them. Since the output is unknown during training, this task is referred to as "unsupervised" while the previous tasks are labeled as "supervised" (we provided expected outputs). Applications of clustering include human genetic clustering, market segmentation (customer groups), social network analysis (communities), image segmentation, anomaly detection, and crime analysis.

- **Density estimation (probability mass function estimation)** entails constructing an estimate of an unknown probability density function based on the input features. In its simplest form, the algorithm learns a function $p\colon \mathbb{R}^M \rightarrow \mathbb{R}$ where $p(\boldsymbol{x})$ is interpreted as a probability density function (or a probability mass function for discrete $\boldsymbol{x}$). An example of basic density estimation is shown using histogram-based density estimation with different numbers of bins. Applications of density estimation include age estimation for countries, modeling complex patterns, feature extraction, and simplification of models.

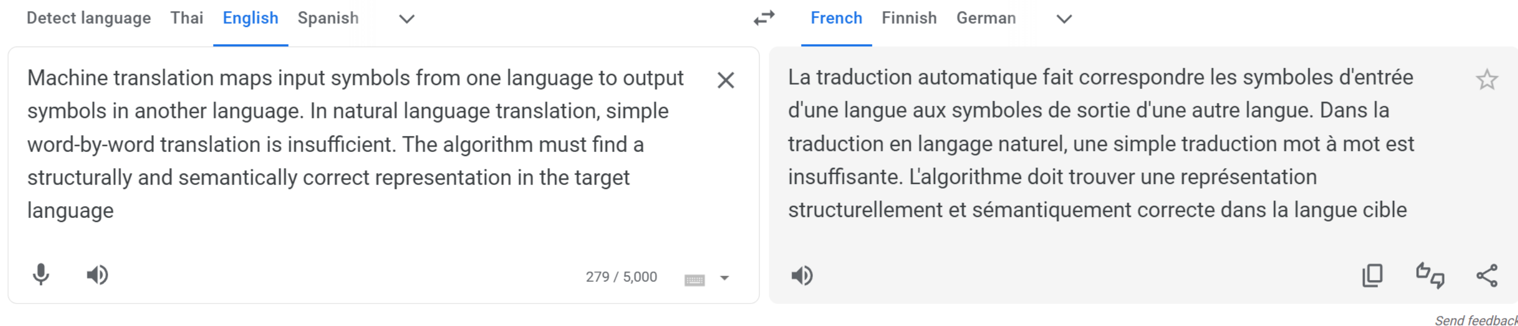


- **Imputation of missing** values involves replacing (estimating/guessing) missing data with substituted values. Given a new example $\boldsymbol{x} \in \mathbb{R}^M$ with some missing x_i, the algorithm must provide a prediction for the missing values. Applications of imputation of missing values include incomplete sensing data, demographics (incomplete personal data), medical analysis (incomplete or expensive test data), and signal restoration after data loss.

- **Anomaly detection** requires the algorithm to identify unusual, incorrect, or atypical events or data points. The output can be a simple binary flag (0 or 1, indicating an anomaly) or a probability of an anomaly. Supervised anomaly detection requires a training set with labels for "normal" (0) and "abnormal" (1) instances. Unsupervised anomaly detection means the algorithm must describe the "normal behavior" (e.g., using density estimation) and automatically detect outliers. Applications of anomaly detection include credit card fraud detection, intrusion detection (cybersecurity), elimination of outliers for statistical analysis, change detection, system health monitoring, event detection, and fault detection.

- **Generative AI** is the generation of new data instances that resemble the existing training data. It captures underlying patterns to create samples with similar characteristics, such as images, text, music, and other content. Generative AI is often a self-supervised learning task where algorithms generate output not tied to specific labeled inputs. Applications include image synthesis, text generation, artistic style transfer, data augmentation, chatbots, computer-assisted coding, artefact creation for games, and text summarization.

- **Machine translation** maps input symbols from one language to output symbols in another language. In natural language translation, simple word-by-word translation is insufficient. The algorithm must find a structurally and semantically correct representation in the target language.
  - Example with Google Translate

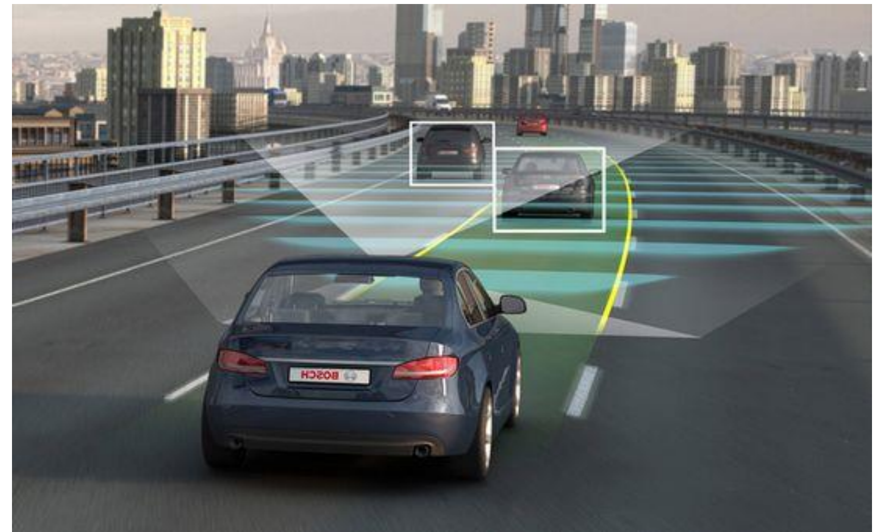


- **Transcription** involves converting unstructured data into a discrete, often textual form. Optical character recognition (OCR) and speech recognition are well-known transcription applications.

- **Dimensionality reduction** simplifies input vectors by transforming them into a lower-dimensional space. The output is interpreted as topics, concepts or embeddings, making it easier for the machine to find documents with similar topics. Dimensionality reduction is commonly used for data mining, latent semantic analysis, principal component analysis, statistical analysis, data reduction, and compression.

- **Reasoning** involves generating conclusions from knowledge using logical techniques like deduction and induction. Knowledge-based systems, including expert systems written in Prolog, have been used for the past 30 years. These systems used facts and rules to prove or disprove statements within a closed world. Modern approaches utilize machine learning for theorem proving or constraint solving. Cognitive reasoning and cognitive AI have recently improved the performance of chatbots and speech recognition.

- **Autonomous robots** employ reinforcement learning, where they adjust their behavior based on incentives and penalties from the environment. Autonomous driving has presented new challenges in reinforcement learning, particularly in machine ethics. Robots must make decisions in unforeseen scenarios where programmers cannot anticipate or hard-code the behavior. For instance, when faced with an inevitable collision with either an animal or a person, should the machine risk an evasive maneuver that endangers its passengers or accept the potential harm to the animal or person on the street?
  - While the field is relatively young, recent progress has been accelerated by deep learning techniques. Tesla claims that its autopilot is ten times safer than the average driver.
  - Laws and societal acceptance of robots are still in their early stages. Concerns regarding safety, privacy, and car hacking are raised, and insurance issues regarding who is liable for mistakes made by robots remain as further obstacles.



99.1.2 Performance

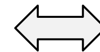
- Performance measures evaluate the effectiveness of a system in performing a task, with each task having its own interpretation of what "good" means. These measures can be categorized broadly whether they support supervised, unsupervised, and reinforcement learning.
- In **supervised learning**, the main tasks are regression and classification
  - In **regression tasks**, performance is measured using the mean squared error (MSE), which calculates the squared difference between the actual values (vector $\boldsymbol{y} \in \mathbb{R}^N$) and the predicted values (vector $\widehat{\boldsymbol{y}} \in \mathbb{R}^N$).

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\widehat{y}_i - y_i)^2 = \frac{1}{N}\|\widehat{\boldsymbol{y}} - \boldsymbol{y}\|_2^2$$

Note that the factor $1/N$ does not change the solution $\boldsymbol{\theta}^*$, hence we can omit it below

Regression models, represented by a function f with parameters $\boldsymbol{\theta}$, map M input values x_{i} to N output values y_{i}, where $f: \mathbb{R}^M \rightarrow \mathbb{R}^N$ and $\boldsymbol{\theta} \in \mathbb{R}^D$. The number of parameters, D, depends on the chosen function. The goal is to find the best solution $\boldsymbol{\theta}^*$ that minimizes the MSE, which involves finding the values of $\boldsymbol{\theta}$ where the gradient is zero.

$$\boldsymbol{\theta}^* = \mathrm{argmin}_{\boldsymbol{\theta}}\|\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}) - \boldsymbol{y}\|_2^2$$



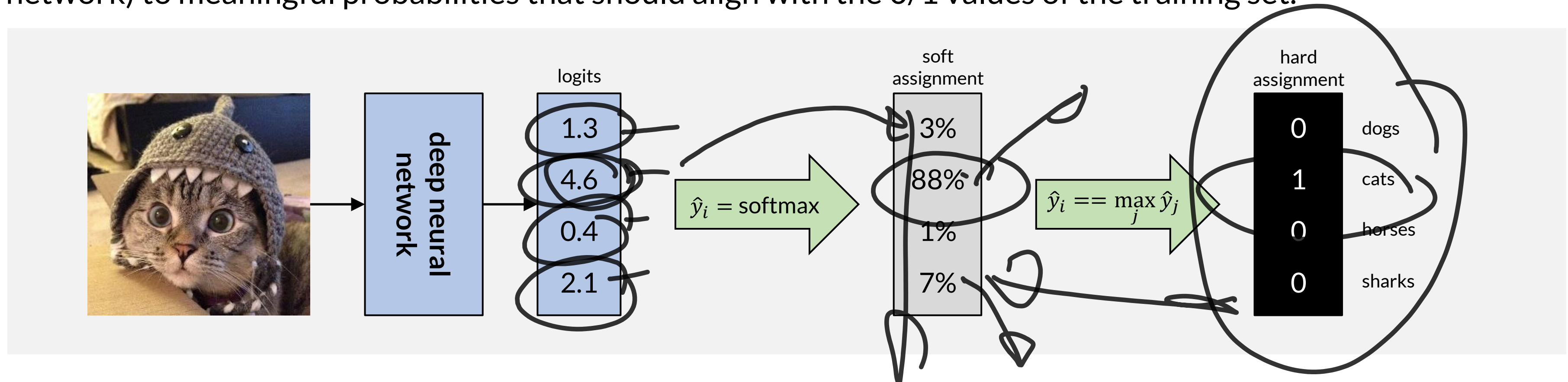
$$\nabla_{\boldsymbol{\theta}}\|\boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x}) - \boldsymbol{y}\|_2^2 = 0$$

For simple regression models, an exact solution can be found using calculus. In more complex cases, a numerical solution using gradient descent is often used, even if it only finds a local minimum (approximate result). The use of squared error simplifies the gradient calculations significantly. Backpropagation, employed in neural networks, utilizes a similar approach to train the weights in the network through stochastic gradient descent.

– **Classification tasks** can be categorized based on two dimensions:

1) **Binary vs. multi-class**: Binary classification distinguishes between two output values, while multi-class deals with multiple output values. The assessment of binary classification often includes precision, recall, accuracy, and other values from the confusion matrix. Tasks often involve classifying against hundreds of different labels, where the confusion matrix helps assess model accuracy.

2) **Hard vs. soft assignments**: To understand this dimension, let's consider an example of detecting different animals in pictures (dogs, cats, horses, sharks). With hard assignments, a neural network can use an output bin for each animal where the highest value represents the predicted class (1) and the other bins are set to 0. With soft assignments, the network outputs probabilities or likelihoods for each class, forming a probability distribution across the classes. Soft assignments require mapping the network's logits (unnormalized output of the neural network) to meaningful probabilities that should align with the 0/1 values of the training set.



With hard assignments, we can compare the output of a neural network directly with the labels in the training set and assess the accuracy of the predictions (see next chapter on "Evaluation" for more details). In contrast, with soft assignments, we first need an effective method to map the logits (network outputs) to meaningful probabilities for each class. Secondly, we require performance measures that evaluate how well these probabilities align with the 0/1 values of the training set.

Consider the above example: the logits are converted into probabilities (the next page will introduce the softmax function). For instance, the bin representing "cats" receives an 88% probability, indicating that the picture likely contains a cat (hard assignment). However, a system that generates only a 79% likelihood for "cats" performs comparatively worse, even though it still results in the same hard assignment ("it's a cat").

With soft assignments, machine learning models produce K output values $o_k \in \mathbb{R}$, known as logits, where K is the number of classes. The range of values for o_k can vary based on the model and activation functions used in deep neural networks, making it challenging to control or predict. We have to convert these values into probabilities p_k, maintaining the relative order among o_k values and ensuring $\sum p_k = 1$ for a valid probability distribution across the K classes. There are many options for such a mapping, but the "softmax" function is a widely used approach.

$$\hat{y}_k = \frac{\exp(o_k)}{\sum_j \exp(o_j)}$$

If we do not state otherwise exp/log always refers to the natural base e. However, for our purpose, the base is irrelevant as it only scales the result but does not change order

In information theory, **cross-entropy** measures the accuracy of a model distribution p in matching the true distribution q over a set of events ε. For classification, the true distribution is often represented as a 'one-hot' vector $\boldsymbol{y}$, with only one component with value 1, and all others with value 0. The cross-entropy loss is then:

$$J(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_k y_k \log \hat{y}_k$$

softmax

$$J(\boldsymbol{y}, \boldsymbol{o}) = -\sum_k y_k \log \frac{\exp(o_k)}{\sum_j \exp(o_j)}$$

By plugging in the softmax definition, we arrive at the formula on the right, which can be further simplified to:

$$J(\boldsymbol{y}, \boldsymbol{o}) = \sum_k y_k \log\left(\sum_j \exp(o_j)\right) - \sum_k y_k o_k \quad = \log\left(\sum_j \exp(o_j)\right) - \sum_j y_j o_j$$

Similar to regression earlier, our goal is to find a model that minimizes this loss function using gradient descent search. To do this, we need to compute the partial derivatives of the loss function for each logit o_k, which simplifies to the difference between the softmax value $\hat{y}_k$ and the true label y_k.

$$\frac{\partial J}{\partial o_k} = \frac{\partial}{\partial o_k}\left(\log\left(\sum\nolimits_j \exp(o_j)\right) - \sum_j y_j o_j\right) = \frac{\exp(o_k)}{\sum_j \exp(o_j)} - y_k \quad = \hat{y}_k - y_k$$

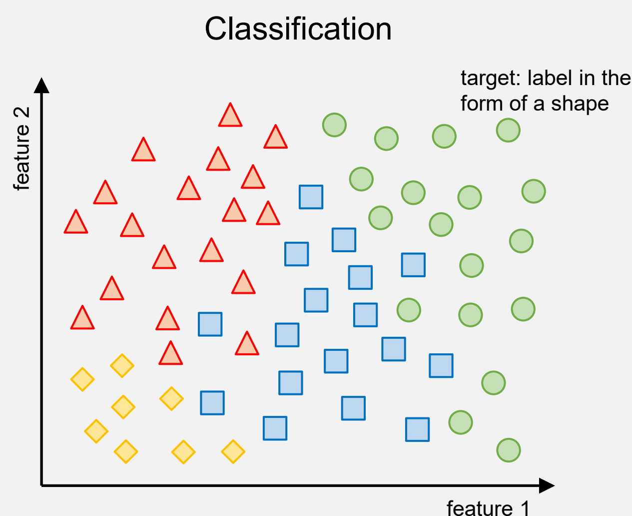
Recall the chain rule from calculus:
$F(x) = f(g(x))$
$F'(x) = f'(g(x)) \cdot g'(x)$

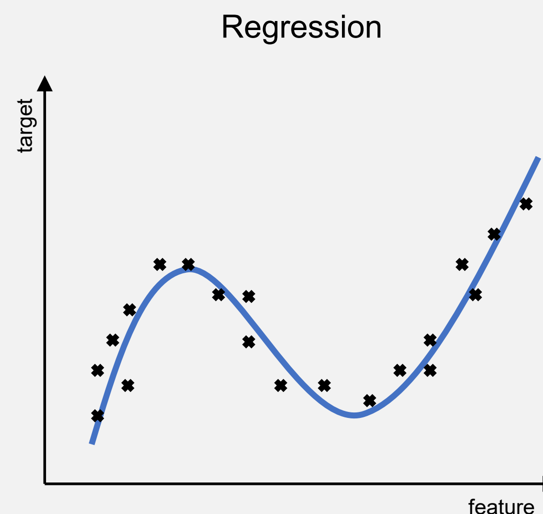
- **Unsupervised learning** encompasses tasks where no ground truth is available, making comparison methods from the previous pages inapplicable. Instead, we evaluate unsupervised tasks based on the model's effectiveness in tasks like identifying "good" clusters and detecting "true" outliers.
  - **Clustering** presents a key challenge in selecting the appropriate number of clusters and achieving well-defined cluster shapes. Having too many clusters may cause coherent regions to split, while too few clusters might result in insufficient distinction among data items, causing everything to blur together. Common methods to address this challenge include the elbow method, Silhouette Score, Davies-Bouldin Index, and Adjusted Rand Index (ARI).
  - **Dimensionality reduction** methods, like Principle Component Analysis (PCA) or auto-encoders, simplify and compress data representation. Performance measures evaluate the model's ability to reconstruct the original data from the reduced representation. Often, dimensionality reduction approximates results for main tasks, like vector search for text retrieval. This requires balancing faster execution with potential loss of quality due to approximation when compared to methods without compression.

- **Reinforcement learning** agents evaluate actions in an environment to maximize cumulative rewards. The tasks are broad and studied in various fields like game theory, control theory, autonomous driving, simulations, and genetic algorithms. Unlike supervised learning, reinforcement learning doesn't have known input/output correlations. The focus is on balancing exploration (of unknown situations) and exploitation (of current knowledge). The agent interacts with the environment in discrete time steps, observing potential rewards, choosing actions, and receiving rewards for transitions. The goal is to maximize cumulative rewards. A few examples:
  - Autonomous Driving: The reward function encourages safe and efficient driving. The agent (self-driving car) receives positive rewards for following traffic rules, staying on the road, and avoiding collisions. It receives negative rewards for breaking rules, driving too fast or erratically, and causing accidents.
  - Game Playing: The reward function focuses on gaining strategic advantages that lead to winning the game. In chess, this includes capturing opponent pieces, controlling strategic cells in the center of the board, and avoiding losing own pieces.
  - Financial Trading: The reward function balances trade profitability with risk exposure. If the agent (trade bot) focuses too much on highly profitable trades, it may deviate from the defined risk profile of the portfolio owner. Conversely, a low-risk trading strategy could result in missed opportunities. The trading strategy must withstand unpredicted market changes while aiming for profitable outcomes.

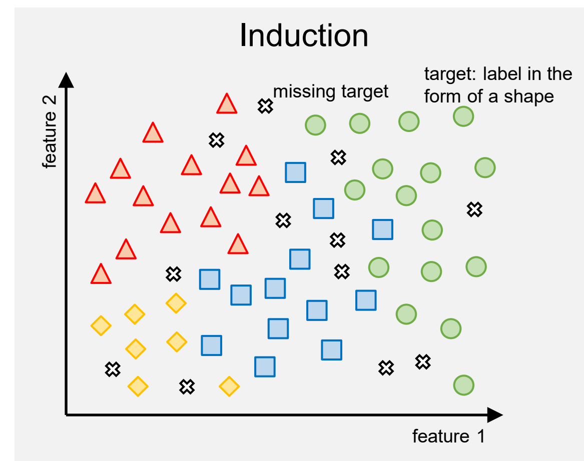
99.1.3 Experience

- **Supervised Learning** algorithms observe a dataset with features and corresponding target labels. The objective is to learn a generic rule that maps features to target labels, allowing the algorithm to predict outcomes for new data instances. The term "supervised" comes from the idea that the target labels are provided by an instructor or teacher. For example, in classification tasks, each data example consists of features and a corresponding label. The "teacher" provides instructions on how the features should be mapped to labels, and the algorithm learns this mapping rule.
  - In the next section, the task is not merely to "learn" the mappings in the training set, but to create a generic model that performs well for new data. The teacher typically provides both the labels and a performance measure, assessing how effectively the generalization worked compared to exact replication of the training data
  - Generating labels for training sets is a laborious and expensive task, similar to obtaining metadata for data objects. New approaches aim to avoid the need for explicit labeling while still gaining the advantages of supervised learning. An example is Generative Adversarial Networks (GANs), where a generator creates fake data (e.g., images) and a competing discriminator evaluates whether a given sample is real (drawn from a given data set) or generated. The generator attempts to deceive the discriminator by maximizing the number of “real” outcomes, while the discriminator optimizes its model to better distinguish between real drawn from the given set and generated data.





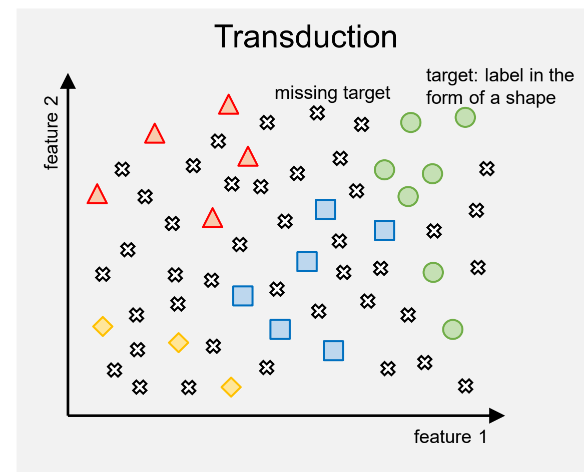
- **Semi-Supervised Learning** is a variation of supervised learning. The algorithm receives both features and targets, but some of the features or the label may be missing in the training data (incomplete observation). Depending on the task, the algorithm needs to either fill in the missing features or predict targets for new data sets.
  - **Missing targets** occur when some objects in the training set lack targets (or labels) due to the expensive or labor-intensive labeling process. For instance, in credit card fraud detection, only a small subset of transactions is labeled as "fraud" or "no fraud" based on investigations. The vast majority remains unlabeled. Algorithms dealing with such data make assumptions to learn effectively.
    1) Smoothness: points in close proximity share the same label. Hence, we assume that the distribution function is smooth and 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 the data is more likely to lie on a low dimensional manifold



    - **Induction**: When only a few labels are missing, a useful approach is to learn the distribution from the labeled data using supervised learning. We can then use this knowledge to predict the missing labels. However, this method becomes ineffective when a large number of objects lack labels, as the training set becomes inadequate to capture the true label distribution. Consequently, this approach disregards a significant portion of the data, leading to information loss.
    - **Transduction:** To utilize all data points, transductive algorithms identify clusters in the dataset and assign the same label to all objects within each cluster. One approach is the partitioning transduction method:

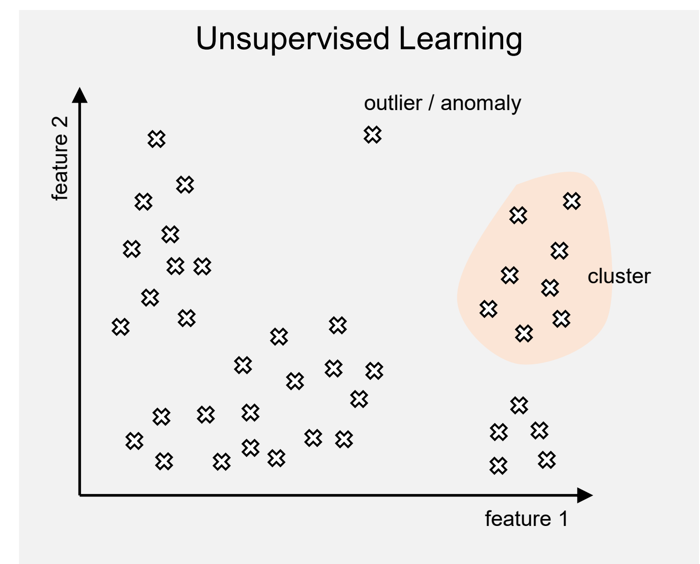
      1. Start with a single cluster containing all objects.
      2. While a cluster has two objects with different labels, partition the cluster to resolve the conflict.
      3. Assign the same label to all objects within each cluster.

      Various other variants exist for developing these clusters.



– **Missing features:** The training set has complete targets, but some items 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 and thus not all features (test results) are available.
  - Naïve Bayes is a simple technique for building classifiers using conditional probabilities. With K classes C_k and M features x_i, the best class k^* is determined 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). When predicting the class for a new object with missing features, we simply ignore those features in the Naïve Bayes optimization.
  - If we have learned the distribution function for all features, we can simply "integrate" or "average" over the missing features. This means assuming that the missing features follow the distribution of the training set and approximating them with an expected value.

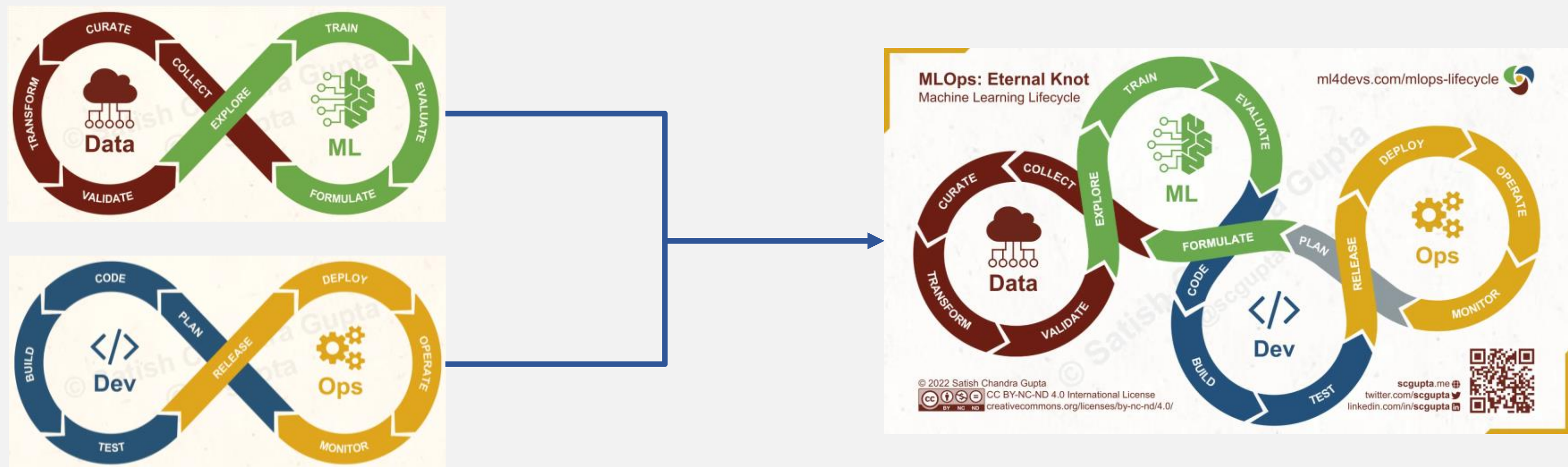
- **Unsupervised Learning** algorithms analyze a data set without targets and derive a function that captures the underlying structure or distribution of the data. The goal is to discover meaningful patterns and gain insights into the data's structure. Unlike supervised learning, there is no instructor or teacher providing targets or evaluating the model's performance. The algorithm must learn and make discoveries independently.
  – Clustering: Identifying groups of objects that are similar based on a distance function. The number of clusters is often unknown.
  – Outlier/Anomaly Detection: Learning the "normal" behavior and identifying outliers that significantly deviate from the rest. The training data may also contain outliers.
  – Density Function: Describing the data with an appropriate density function. Gaussian approximation is a method, while more complex ones optimize for the best fit among different distribution functions.
  – Dimensionality Reduction: Extracting core concepts from high-dimensional features using techniques like Principal Component Analysis for a simpler yet accurate view of the data.
  – Self-Organizing Maps (SOM): Mapping high-dimensional data to 2-dimensional presentation using a competitive learning approach.



99.1.4 Training

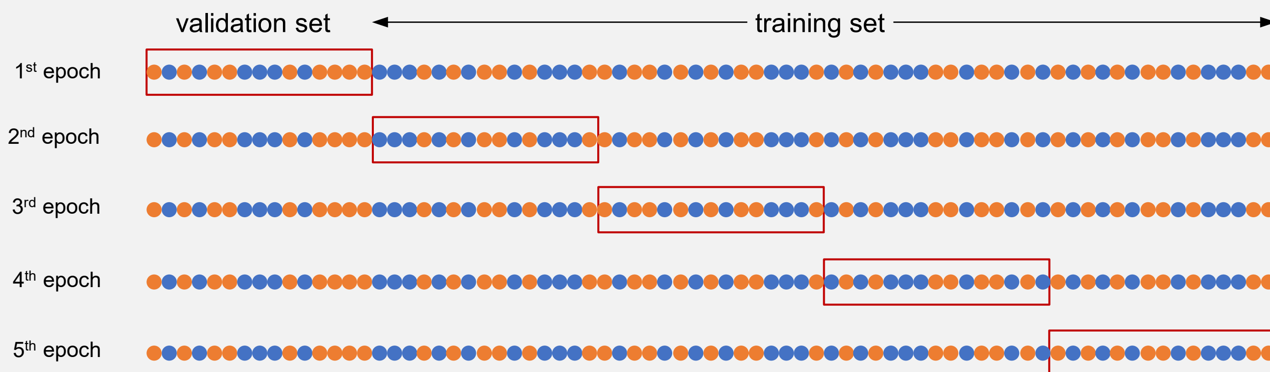
- Training is a vital step in the broader "Model Development" process, which is now integrated with DevOps to form a comprehensive machine learning lifecycle known as MLOps. This process encompasses four main cycles:
  - The **data** cycle collects, curates, and generates datasets for learning. For example, this involves gathering images, filtering, and labeling them for an object recognition task, and creating training, test, and validation subsets.
  - The **machine learning** cycle selects an appropriate model, optimizes hyperparameters, trains the model, and evaluates its performance to choose the best option. In the image recognition example, we might use a ResNet architecture, adjust the number of layers and other parameters to achieve the highest accuracy on the test set.
  - The **development** cycle integrates the model into the application architecture or business process and conducts end-to-end testing of the (business) use case. For the image example, this could mean embedding the trained model in a container with an API accessible by other parts of the architecture.
  - The **operations** cycle extends the application operations to monitor the performance of the machine learning model and trigger feature requests if its performance falls below the desired level. In the image example, this might include a feedback mechanism where end-users can report issues with objects not properly identified.

source: ml4devs.com, Satish Chandra Gupta



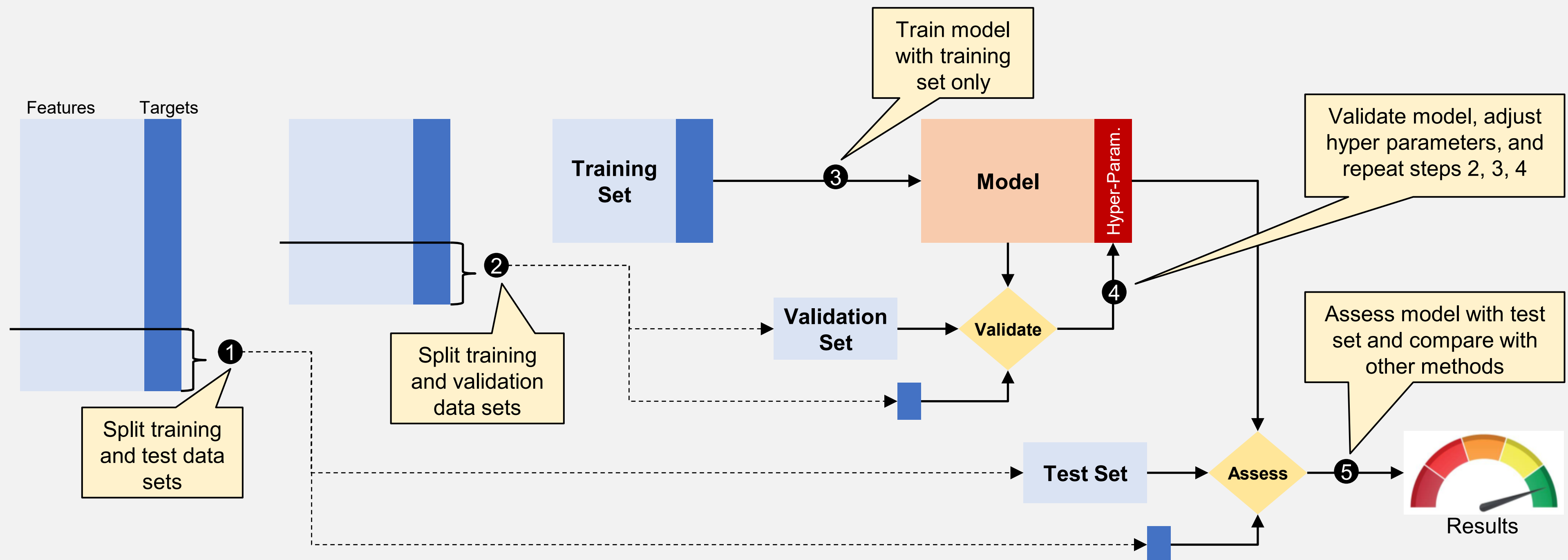
- While we aim for good performance on the training data, our primary goal is to ensure that the model excels with new, unseen data. Memorizing the training data is undesirable since it leads to poor performance on unfamiliar data. Instead, we seek a model that not only performs well on the training data but also generalizes its learnings, allowing it to achieve equally impressive results on new, previously unseen data.
- Earlier in this chapter, we introduced cross-entropy as a loss function for classification tasks. During model training, our goal is to minimize this loss. However, the best model is not solely determined by the lowest loss on the training data. It is the model that achieves the highest accuracy on new, unseen data. To estimate this accuracy, we split the data into two parts: roughly 80% for training, and the remaining for final evaluation without being used in training. In machine learning competitions, participants do not even have access to the test data. They can optimize their models with training data, but final evaluation is with unknown new data to identify the best-performing models.
- Most models have hyperparameters, like ResNet in PyTorch with varying number of layers. Self-developed models may have other hyperparameters like activation functions, optimization algorithms, regularization features, and different number representations (int16, fp16, fp32). This adds an extra optimization loop to the training process to find the best hyperparameter set.
  - Once more, our goal is to find the best hyperparameters that perform well on future predictions. While minimizing the training loss is essential, we do not want the model to memorize it; instead, it should learn to generalize based on the chosen hyperparameters. We select the hyperparameters that enables the model to generalize most effectively.
  - However, using the test data for this purpose would "leak" information into the training process that compromises the accuracy of future predictions. Even model developers should refrain from inspecting the test data to diagnose model failures, as it could lead to over-optimization for the training and test sets, rather than improving the model's ability to generalize.
  - Instead, we further divide the training data: approximately 80% for model training, and the remaining portion for validating the chosen hyperparameters. After finding a good set of hyperparameters, we can then use the test set to assess the final model performance.
- To avoid bias towards the training set, modify the validation set in each iteration when searching for optimal hyperparameters. Remember, we want to find models that generalize well and not models that memorize training and/or validation data

- In cases where we have limited training examples due to the double splitting of the data, we can use the **k-fold cross-validation** approach. This method not only ensures effective model training but also prevents overfitting by presenting different data to the model in each epoch.
  - As before, it is crucial to keep the test data separate from the training data. Never use any part of the test data during the training process, even if you are running low on examples.
  - However, we vary the validation set in each epoch. An epoch is a full iteration over all the training data, and during this iteration, we evaluate the model on the validation set. For each epoch, we split the training and validation data differently using the following approach illustrated below.
    - The training data, after splitting the test data, is divided into k folds. For smaller datasets, using a larger value for k can lead to better results, with typical values ranging between 5 and 10.
    - In each epoch, one fold is used for validation, while the other folds are used for training. As we iterate over epochs, different folds are used for training, reducing the risk of bias towards the training data. However, this increases the variance of our accuracy estimates.
    - k is a hyperparameter itself, and you have to optimize it for each learning process again. Try varying k to observe which value yields the best validation results (do not use the test data for this, as discussed before).
    - Sometimes datasets are imbalanced, with some labels being rare while others are frequent. To improve the stability of the learning process, we need to ensure that the folds contain a representative number of each class.
  - If more epochs are required, we can restart with the first fold and continue iterating until the model's performance has converged or we decide to stop the current run and begin again with new hyperparameters.



The training process involves the following steps (see picture below)

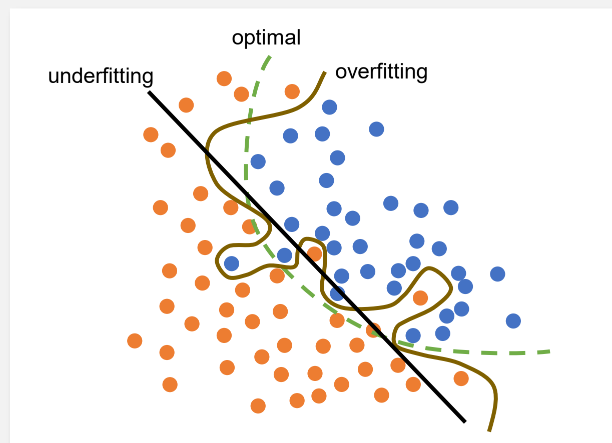
1. Split the data into training and test sets. Keep about 20% test data for final assessment, but avoid using or looking at the test data during training. In competitions, the test data is usually not shared with contestants
2. Split the training data again into training and validation sets. Allocate about 20% for validating hyperparameters. Consider k-fold cross-validation for limited data or to prevent bias towards the training data
3. Train the model with the training set and iterate over epochs until performance converges or it has become evident that the current hyperparameters are sub-optimal
4. Adjust hyperparameters based on validation results. Retrain the model with the optimal set of hyperparameters using more epochs if needed
5. Finally, evaluate the model's performance using the test data and compare it with other approaches. If you iterate the training process, use a different test data set to avoid bias. In competitions, the contestants only receive the final assessment results but cannot inspect where and why the model has failed

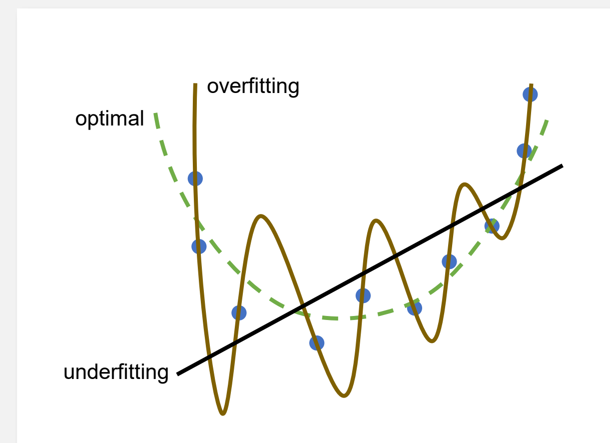


99.1.5 Over- and Underfitting

- During the training process, we mentioned the bias towards the training set without fully explaining its meaning and how to detect it. A fundamental concept in machine learning deals with the balance between two types of errors that can occur during the training process: bias and variance
  - Bias is the error introduced by using an overly simplistic model to approximate the data. A model with high bias is not able to capture the essential patterns and relationships in the data
  - Variance is the sensitivity of the model to small changes in the data, often caused by a too complex model. A model with high variance has good performance on the training data but poor performance on new, unseen data
  - A tradeoff is necessary because reducing bias can increase variance, and vice versa. The goal is to strike a balance by finding a model that generalizes well to unseen data. Thus, the goal is to minimize the sum of bias and variance
- Model complexity is a key factor influencing bias and variance. Different machine learning structures have different facets of complexity. For instance, a linear regression model is simpler with fewer parameters to adjust $y = a \cdot x + b$, while a polynomial model has more parameters, giving it greater adaptability to different datasets. Capacity, in this context, refers to a model's ability to adjust itself through structural changes and model parameters.
- But what is the right complexity: if the model is too simple (low complexity), we risk a high bias; if the model is too complex, we risk a high variance like memorizing the training data rather than generalizing the observed patterns
  - An example for a too simplistic model: “if the sun is out, it is warm”
  - An example for an overly complex model: “if the sun is out and it is a summer month and you are on the north side or it is a winter month and you are on the south side or you are equatorial or you are in a dessert and it is not an ice dessert and it is not cloudy or raining or snowing and there is not a strong wind and there is not a sun eclipse and there is not a volcano eruption and you are not in the water or in a cave or in the shadows or in a house with air conditioning or in a car with air conditioning or in a freezer … then it is warm”
- Our brain is excellent in finding the right level of abstraction. Consider the following examples:
  - “birds can fly” but wait, not all birds can fly → we us a simple model and learn the exceptions
  - “describe what makes a chair a chair” → no simple model, so we employ abstract concepts (“you can sit on it”)
  - “horse” → narrow variety of forms and what is accepted as a horse (e.g., donkey, zebra, giraffe)
  - “dog” → wide variety of forms that count as dogs yet we recognize them immediately

- **Overfitting and underfitting** are common issues in machine learning. Overfitting happens when the model becomes overly complex, trying to match the training data too closely. This often occurs when the model has too many parameters relative to the available training data, leading to poor predictive performance when applied to new data. Underfitting, on the other hand, occurs when the model is too simplistic to capture the underlying data trends. For example, fitting a linear model to a non-linear data distribution will result in high training error and inadequate predictive performance.
  - As shown below, **overfitting** occurs when the model is optimized for the training data using too many parameters. Such a model may display a small training loss, indicating good adaptation to the training data, but it fails to predict new data points effectively
  - On the other hand, **underfitting** exhibits large errors on the training data and poor prediction performance for new data points. It clearly fails to capture the true essence of the distribution
  - To control overfitting and underfitting, we can adjust the model's capacity. The optimal capacity is achieved when the model shows small errors on both the training set and the validation set
- To recognize overfitting and underfitting during training, we use a loss function like mean squared error or cross-entropy loss. Underfitting is indicated by high losses on both the training and validation sets, while overfitting is characterized by small losses on the training data but significantly higher losses on the validation set. Note that the loss on validation sets is typically higher, but a large gap is a key sign of overfitting.

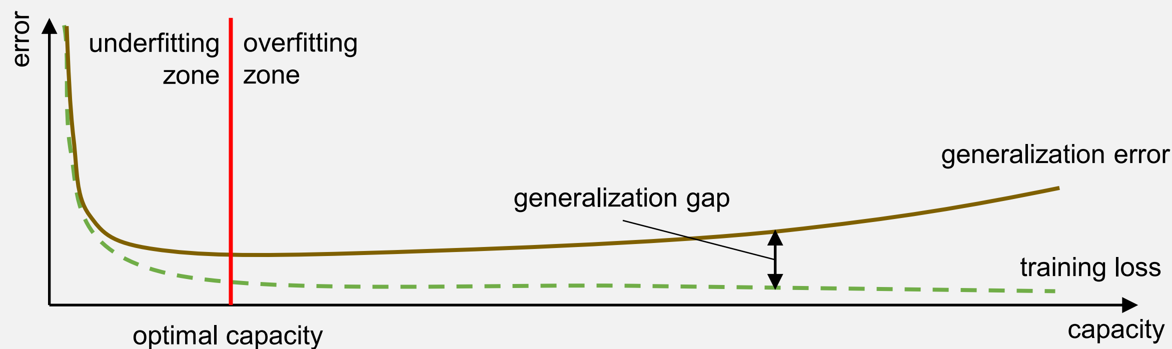




- How can we optimize the model's capacity to strike a balance between bias and variance?
  - **Occam's razor** is an intuitive heuristic, first stated by William of Ockham (c. 1287-1347). It has been further refined, especially in the 20th century, for statistical learning purposes.

Numquam ponenda est pluralitas sine necessitate (Plurality must never be posited without necessity

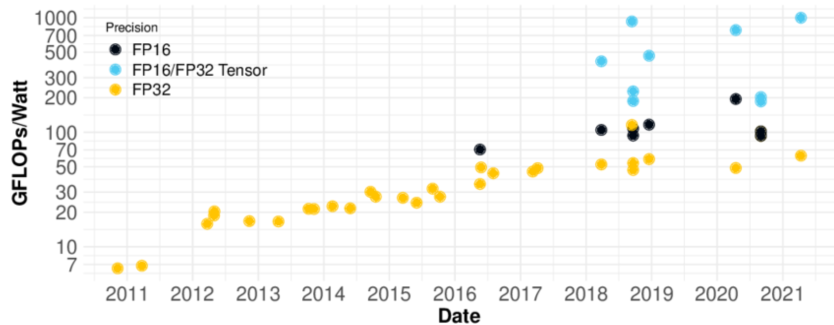
    In a modern interpretation, Occam's razor suggests that when multiple hypotheses explain observations equally well, we should prefer the simplest one. This means we aim for models with low complexity and only increase it if needed to perform well for the specific task. We can also find examples for Occam's razor in physics, such as Newton's laws, which offer a simplified yet effective approximation of the real-world in many situations.
  - Simpler models are better at generalizing, but we must avoid models that are overly simplistic and have high training loss. As we increase the model's capacity, the training loss decreases. However, if the capacity becomes too high, the model loses its ability to generalize (e.g., memorizing data), and the gap between training loss and validation loss widens. The illustration below shows the underfitting and overfitting zones, divided by the optimal model capacity that reduces training loss and enables good generalization at the same time.



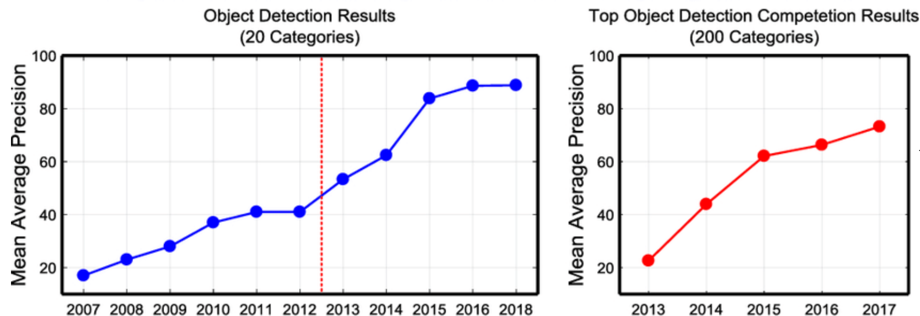
  - Modern language models use deep learning architectures with billions of parameters. Handling such a large number of parameters presents new challenges as we must prevent the model to memorize and instead encourage it to generalize. Later in this chapter, we will explore different regularization techniques, which aim to penalize complex models and strike a balance between bias and tradeoff. For instance, L2 regularization in deep learning introduces a penalty in the loss function to discourage the utilization of many high value weights. This motivates the model to use fewer connections in the network and prevents overfitting.

99.2 Neural Networks

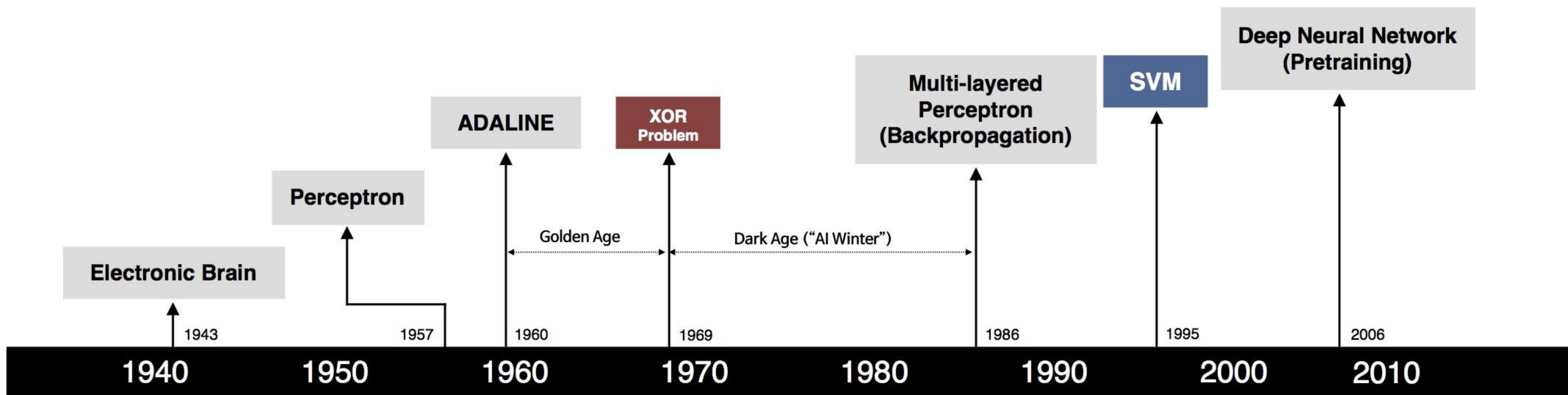
- Artificial neural networks are machine learning models inspired by the brain. Brain research has often influenced new approaches, such as using connections between non-adjacent neuron layers (multi-layer approach). Neural networks are commonly used to model the brain and its learning algorithms.
- The initial phase of neural network research began in the late 1950s, primarily centered on single perceptrons in hardware. Multiple perceptrons could work in parallel but were limited to input and output connections. The well-known issue with perceptrons was their inability to learn the basic XOR function. While it was demonstrated that a two-layer network could encode XOR, its limitations were evident, marking the onset of the first AI winter.
- The second wave began in the 1960s when hidden layers were introduced. Various researchers explored similar ideas, but major credit is often attributed to Rumelhart, Hinton, and Williams for their 1986 paper on backpropagation, which remains a foundational reference in textbooks. This revival led to convolutional networks, recurrent networks, belief networks, and many concepts seen in modern deep learning. Yet, the field grappled with calculation problems (vanishing and exploding gradients) and computational constraints in the 1980s and 1990s.
- In the early 2000s, research and funding for the field were scarce. However, a small research team led by Hinton, funded by the Canadian government, rebranded it as "Deep Learning" and published a groundbreaking paper in 2006, introducing a fast learning algorithm for deep belief nets. Simultaneously, computing power greatly expanded. Inspired by the Canadian team, the field resurged, realizing that GPUs were up to 100 times faster than CPUs. This enabled rapid training of deep networks in hours and days instead of weeks and months. In 2011, Google initiated the Google Brain project, connecting thousands of CPUs for a network with 1 billion weights.
- Over the past five years, transformers have seen significant advancements, including the growth of massive models, increased efficiency, progress in few-shot learning, and widespread applications across various domains. The launch of ChatGPT created a new “euphoria” around AI, especially, generative AI. Hinton and Hopfield received the nobel prize for Physics in 2024 for their foundational work on neural networks.



source: https://www.researchgate.net/figure/Theoretical-Nvidia-GPUs-GFLOPS-per-Watt-Data-in-Table-8-in-the-appendix_fig3_354573934



source: https://www.researchgate.net/figure/An-overview-of-recent-object-detection-performance-we-can-observe-a-significant_fig3_336934637



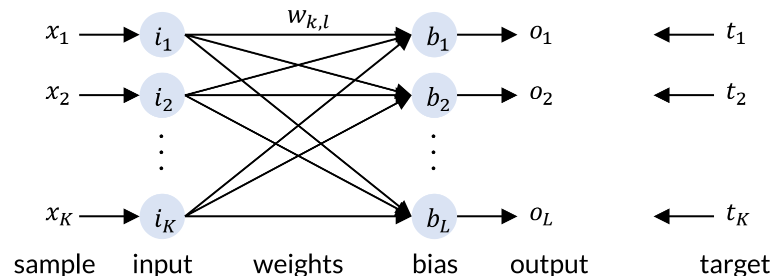
source: https://beamandrew.github.io/deeplearning/2017/02/23/deep_learning_101_part1.html

99.2.1 The Perceptron

- Let's start with the original perceptron concept: it's essentially a binary classifier that maps a real-valued input vector $\boldsymbol{x} \in \mathbb{R}^K$ to a binary output value $f(\boldsymbol{x})$:

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

Here, $\boldsymbol{w} \in \mathbb{R}^K$ represents the weights, and b is the bias. Based on this definition, the perceptron divides space with a hyperplane described by $\boldsymbol{w}^T \boldsymbol{x} + b$. In a broader context, L perceptrons with weights $\boldsymbol{w}_l$ and bias b_l connect to K input values i_k and generate L binary output values o_l. We can illustrate this general configuration as follows:



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

with the binary step function

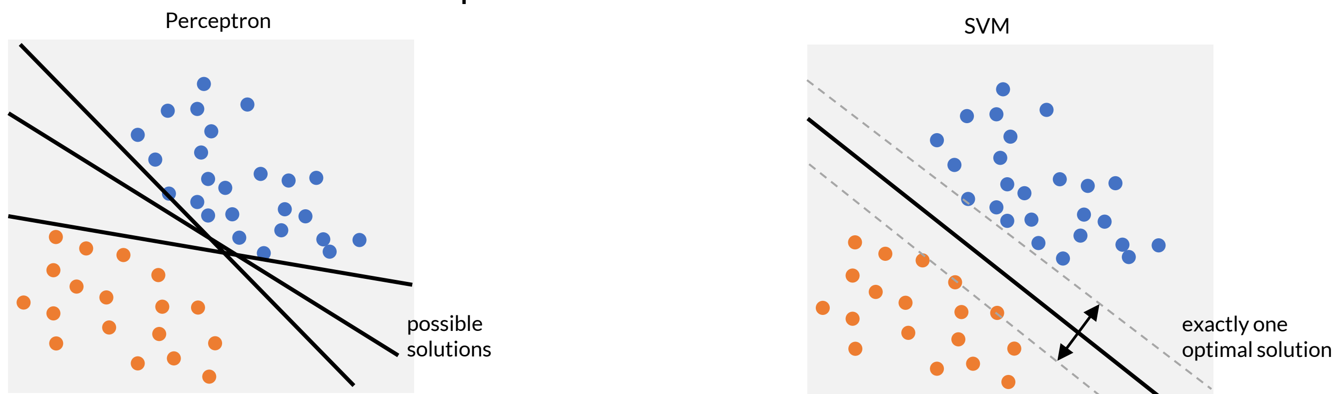
$$f(z) = \begin{cases} 1 & z > 0 \\ 0 & \text{otherwise} \end{cases}$$

The learning algorithm is then as follows: (demo: https://codepen.io/bagrounds/full/wdqypY)

1. Initialize the weights $w_{k,l}^{(0)}$ and the biases $b_L^{(0)}$ with small random values. Set a learning rate $0 \leq \alpha \leq 1$
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 occurs only when the dataset is linearly separable. Otherwise, the algorithm might fail. Various variants have been developed to tackle this challenge.

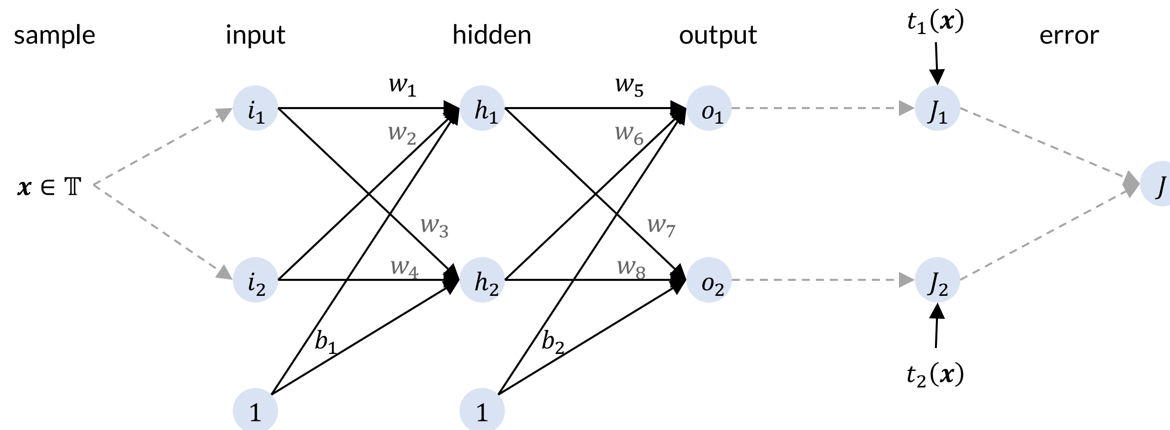
- In simple terms, the perceptron learning algorithm adjusts weights (and bias) only when the target differs from the output. If the output is 0 and the target is 1, weights and bias increase; otherwise, they decrease (assuming $x_l \geq 0$). It's important to note that the algorithm doesn't aim to optimize any objective function; it's a heuristic method for weight learning. When the data is separable, it converges to a binary space partition using a hyperplane (one of many possible partitions).
- In contrast, the **support vector machine (SVM)** finds the best hyperplane that separates the sets while maximizing the margin (distance from marginal points to the hyperplane). SVM can also handle non-separable data by minimizing the partitioning error. The details of SVM computation are not discussed here.



- A binary classifier can be adapted for learning multiclass outputs. The "one-vs-all" approach trains a binary classifier for each of the L classes to distinguish class C_l from the others. This means using L perceptrons, with the binary target vector $\boldsymbol{t}$ having $t_l = 1$ and all other components set to 0. During prediction, the class with the highest output value is considered the "winner". Alternatively, the "one-vs-one" strategy employs $L(L-1)/2$ perceptrons to differentiate pairs of classes, training these perceptrons individually. Again, the class with the highest output value during prediction is regarded as the "winner".
- SVM's linear classification might seem restrictive. However, SVM employs the "kernel trick", where data is mapped to a higher-dimensional space, enhancing separability. This mapping is typically nonlinear. The "kernel trick" means we don't explicitly compute the high-dimensional mapping; instead, we calculate the inner product required for SVM using kernels like $K(\boldsymbol{x},\boldsymbol{y}) = (1+\boldsymbol{x}^\top\boldsymbol{y})^2$ in a 6-dimensional space for $\boldsymbol{x},\boldsymbol{y} \in \mathbb{R}^2$. A Gaussian kernel, $K(\boldsymbol{x},\boldsymbol{y}) = \exp\left(-\gamma\|\boldsymbol{x}-\boldsymbol{y}\|^2\right)$, yields an infinite-dimensional mapping φ. The "kernel trick" is seen as human intervention into machine learning. SVM classification is efficient but requires an appropriate kernel function design for the problem.

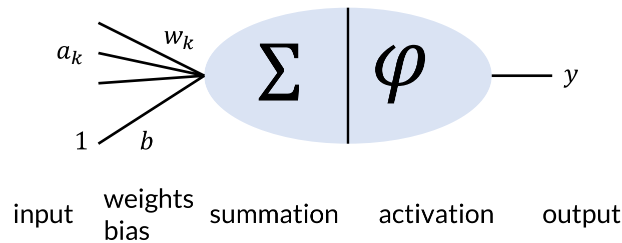
99.2.2 Multilayer networks

- Multilayer networks represent a significant evolution from the original perceptron. They introduce several key changes to the architecture: instead of a single layer, these networks feature multiple "hidden" layers situated between the input and output layers. These hidden layers allow for more complex computations. Additionally, the activation functions are not restricted to binary outputs, enabling more varied responses from the individual neurons. To optimize the performance, objective functions are used to define the optimal state for all the network parameters. This helps the network learn and adapt more effectively. Finally, a new learning algorithm, known as backpropagation, is employed to adjust the weights of the network, facilitating the training and fine-tuning.
- We begin by examining the basics using a straightforward two-layer network as a concrete example. Afterward, we extend these concepts to networks of arbitrary shapes.



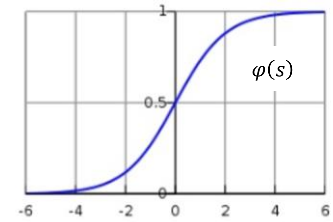
- The network comprises two input neurons i_1, i_2, two hidden neurons h_1, h_2, and two output neurons o_1, o_2. Shared biases include b_1 for the hidden neurons, and b_2 for the output neurons, with the bias modeled as a weight from a neuron always in the state 1. The connections are defined by weights w_1 to w_8, linking one layer to the next, without inter-layer connections or cycles, simplifying the topology. Additional nodes J_1 and J_2 are introduced to measure the training error, with J representing the overall training error.

- **Feed-Forward:** When provided with a data sample $\boldsymbol{x}$ from the training set $\mathbb{T}$, the network calculates the state of each neuron using a straightforward model.



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

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



- We denote the summation result as s and apply the logistic activation function φ, also known as the soft step function. Using this, we can determine the state of each neuron based on 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 remains 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 aim to assess the network's ability to predict targets for all data samples in the training set $\mathbb{T}$. To begin, we employ the mean square error (MSE).

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

  Here, $\boldsymbol{\theta}$ represents the network's parameters. In our example, $\boldsymbol{\theta} = (w_1, \dots, w_8, b_1, b_2)$. The process of learning the network involves finding the parameters θ^* that minimize the error function:

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

  - Because of the network's size and the volume of data, solving the equation directly is often impractical. Instead, we employ the gradient descent method to iteratively find a (local) optimum. The **gradient descent** method relies on the gradient $\boldsymbol{\nabla}J(\boldsymbol{\theta})$ for the network's parameters θ, defining the network's learning strategy:

    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: $\Delta^{(t)} = \eta \cdot \boldsymbol{\nabla}J\left(\boldsymbol{\theta}^{(t)}\right)$
       - Adjust parameters: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \Delta^{(t)}$

  - Gradient descent can be slow near the (local) minimum and may exhibit zigzag behavior, especially for poorly conditioned convex functions. Moreover, with large-scale datasets and networks, gradient descent demands significant computational resources and storage capacity to compute the gradient, which we can derive directly for the network as we will explore later.
  - Improved gradient descent methods, such as SGD, Adagrad, Adam, natural gradient descent, and Nesterov Accelerated Gradient, offer solutions to standard gradient descent's limitations. They enhance convergence speed, handle complex optimization landscapes, and adapt learning rates based on parameter-specific characteristics. These techniques are vital for efficiently training machine learning models and neural networks.

– Neural network algorithms commonly employ **stochastic gradient descent (SGD)** along with momentum to mitigate the zigzag problem. Instead of computing the gradient over all data samples, SGD approximates it using a sub-set of the data (so-called mini-batch), which minimizes storage requirements. However, SGD can exhibit slow convergence in later iterations. Momentum, on the other hand, accelerates descent by retaining the gradient from the previous iteration and applying a fraction γ of it in the current 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 $\mathbb{T}$ into K subsets $\mathbb{T}_\mathrm{k}$
   - For each $\mathbb{T}_\mathrm{k}$:
     - Compute gradient: $\Delta = \gamma \cdot \Delta + \eta / |\mathbb{T}_\mathrm{k}| \cdot \sum_{\boldsymbol{x} \in \mathbb{T}_\mathrm{k}} \boldsymbol{\nabla} J(\boldsymbol{x}; \boldsymbol{\theta}^{(t)})$
     - Adjust parameters: $\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \Delta$
   - Increase γ

The momentum γ determines how long a past gradient remains influential. Typically, we begin with $\gamma = 0.5$ and gradually increase it to $\gamma = 0.9$ or even higher once the initial learning stabilizes.

– The above algorithm outlines the learning strategy. In each epoch (step 2), the entire training set is processed, adjusting the network's weights and biases for each mini-batch. The remaining task is to calculate the gradient $\boldsymbol{\nabla} J(\boldsymbol{x}; \boldsymbol{\theta})$ for the current data sample and the current network parameters.

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

- **Gradient Calculation:** Before delving into the backpropagation algorithm, let's revisit our initial example network with two input nodes, two hidden nodes, and two output nodes. To perform stochastic gradient descent, we must compute the gradient. In our example, we have $\boldsymbol{\theta} = (w_1, \ldots, w_8, b_1, b_2)$. The gradient is represented by the partial derivatives with respect to $J(\boldsymbol{x}; \boldsymbol{\theta})$:

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

With the specified targets t_1 and t_2 for data sample $\boldsymbol{x}$, and using the previously defined functions o_1 and o_2 that depend on $\boldsymbol{x}$, as well as the weights $w_1, \ldots, w_8$ and biases b_1 and b_2.

– Let's start with w_5. It exclusively influences o_1, not o_2. Therefore, the partial derivative is as follows:

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

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

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

$$\frac{\partial o_1}{\partial w_5} = \frac{\partial}{\partial w_5}\left(\varphi(s_{o_1})\right) = \varphi(s_{o_1}) \cdot \left(1 - \varphi(s_{o_1})\right) \cdot \frac{\partial s_{o1}}{\partial w_5} = o_1 \cdot (1 - o_1) \cdot \frac{\partial s_{o1}}{\partial w_5}$$

$$\frac{\partial s_{o1}}{\partial w_5} = \frac{\partial}{\partial w_5}(w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) = h_1$$

all together:

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

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

– 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 can observe recurring patterns in the calculations: the error function derivatives are multiplied by the activation function derivatives and then multiplied by the summation derivatives. To calculate the gradients, we need the results (states) from the feed-forward step, allowing us to efficiently compute the gradients using the backpropagation method.

– Now, let's consider the remaining partial derivatives (refer to the next page for details on deriving them for w_1).

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

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

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

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

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

– Now, let's focus on w_1. It's worth noting that w_1 is present in h_1, which, in turn, contributes to both o_1 and o_2.

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

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

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

$$\frac{\partial o_1}{\partial w_1} = \frac{\partial}{\partial w_1}\left(\varphi\left(s_{o_1}\right)\right) = \varphi\left(s_{o_1}\right) \cdot \left(1 - \varphi\left(s_{o_1}\right)\right) \cdot \frac{\partial s_{o_1}}{\partial w_1} = o_1 \cdot (1 - o_1) \cdot \frac{\partial s_{o_1}}{\partial w_1} \qquad \frac{\partial o_2}{\partial w_1} = o_2 \cdot (1 - o_2) \cdot \frac{\partial s_{o_2}}{\partial w_1}$$

$$\frac{\partial s_{o_1}}{\partial w_1} = \frac{\partial}{\partial w_1}(w_5 \cdot h_1 + w_6 \cdot h_2 + b_2) = w_5 \cdot \frac{\partial h_1}{\partial w_1} \qquad \frac{\partial s_{o_2}}{\partial w_1} = w_7 \cdot \frac{\partial h_1}{\partial w_1}$$

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

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

all together:

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

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

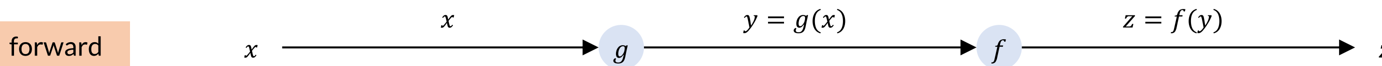
99.2.3 Backpropagation

- It is indeed feasible to compute all partial derivatives for the gradient, but this approach appears laborious and error-prone. Is there a simpler way? Yes, there is. Backpropagation offers a remarkably straightforward method to calculate the gradient, beginning at the error node and working backward toward the input nodes. Although it doesn't yield closed-form derivatives, it efficiently computes the gradient without redundant calculations.
  - Let's revisit the chain rule from calculus:

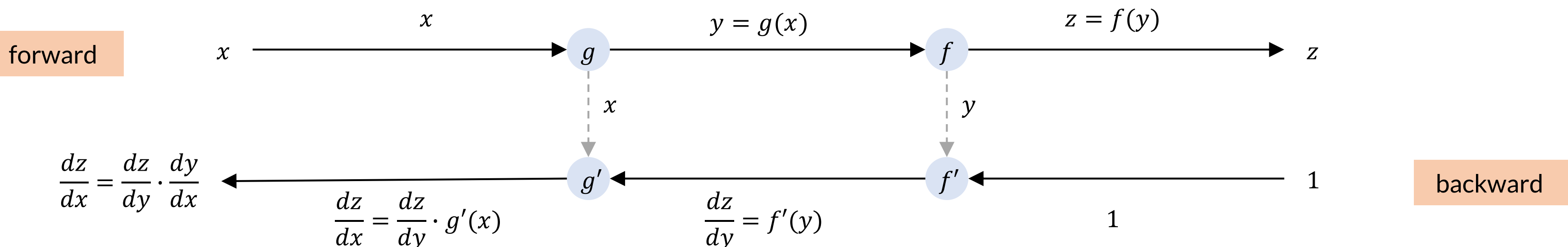
$$F(x) = f \circ g = f\big(g(x)\big) \qquad F'(x) = f'\big(g(x)\big) \cdot g'(x)$$

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:



To compute the derivative $\frac{dz}{dx}$ for x, we work backward. We start by calculating $f'(y)$ and then multiply it by $g'(x)$. This requires keeping track of intermediate results to use them on the backward path for derivative calculation



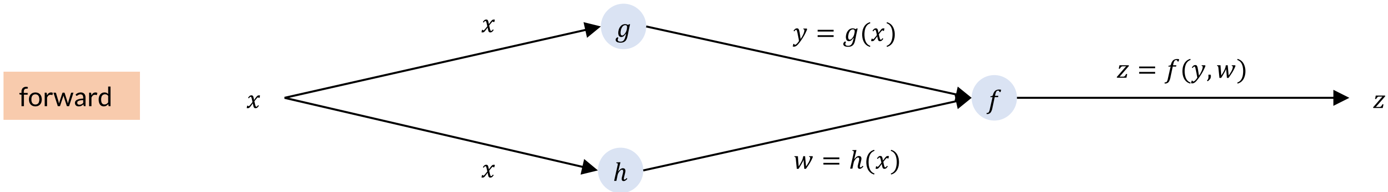
- Likewise, we can examine multivariable chain rules.

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

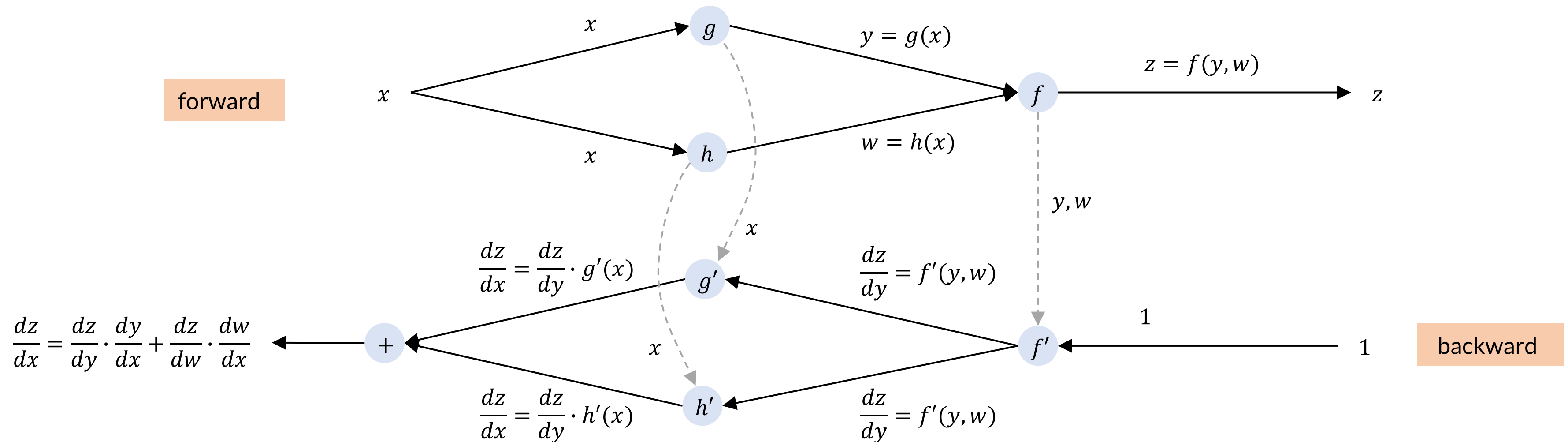
or in Leibniz notation with $z = f(y), y = g(x)$ and $w = h(x)$

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

In graphical notation, we establish the forward path for function computation.



To calculate the derivative $\frac{dz}{dx}$ for x, we move backward in a manner similar to what we've done before.



- Now, let's employ the chain rule within our neural network. We begin with the output neurons. To streamline the structure, we introduce a node a_0, which is always in the state 1, and the weight $w_0 = b$, representing the bias. This adjustment simplifies the formulas. The visual representation of the forward and backward paths is illustrated below.

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

forward

$a_0 = 1$ — $w_{k,l}, w_{0,l} = b_l$ — t_l

$s_l = \sum_k a_k \cdot w_{k,l}$ — $o_l = \varphi(s_l)$ — $J_l = \frac{1}{2}(t_l - o_l)^2$ — $J = \sum_l J_l$

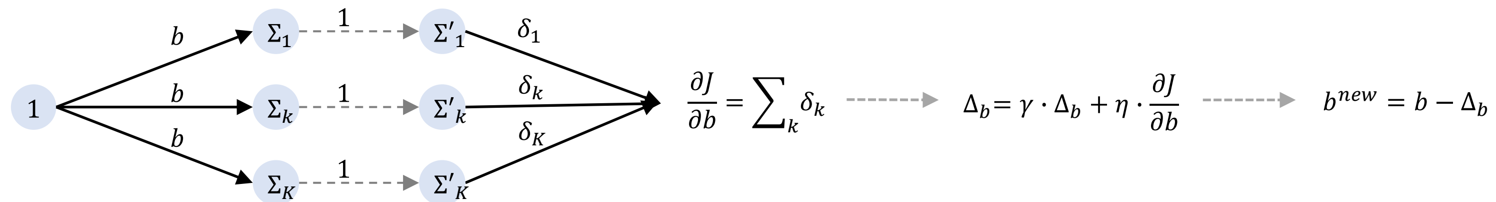
a_k — $o_l = \varphi(s_l)$ — t_l, o_l — J_l

backward

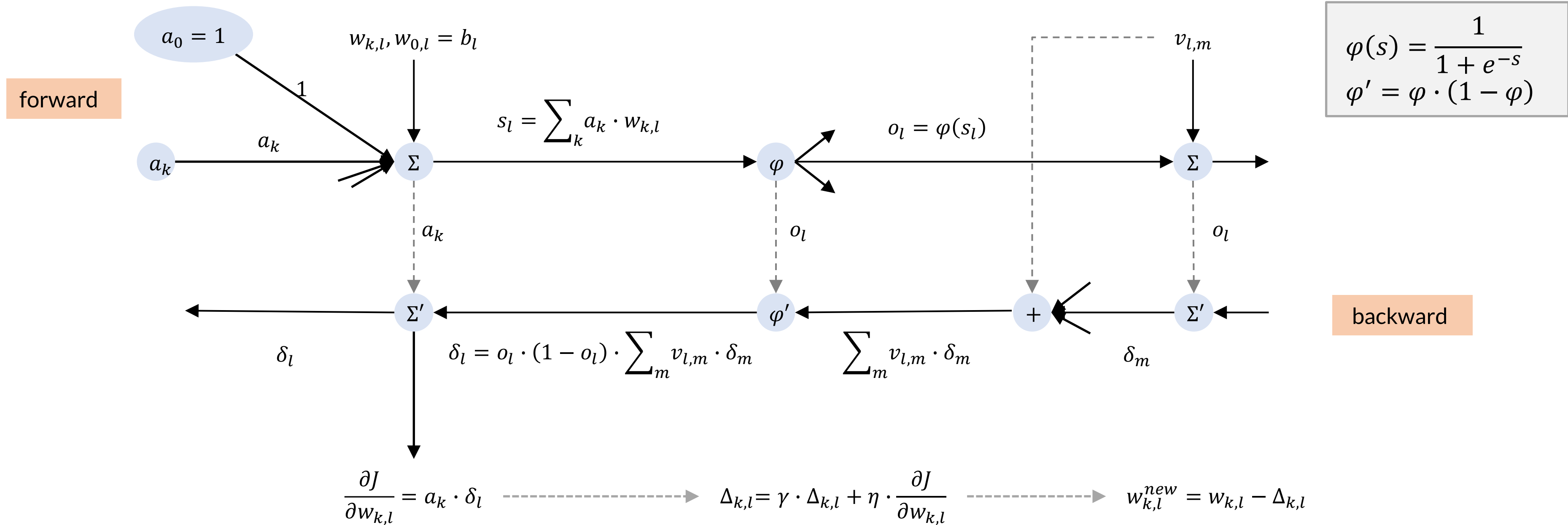
δ_l — $\delta_l = o_l \cdot (1-o_l) \cdot (t_l - o_l)$ — $(t_l - o_l)$ — 1

$$\frac{\partial J}{\partial w_{k,l}} = a_k \cdot \delta_l \quad \rightarrow \quad \Delta_{k,l} = \gamma \cdot \Delta_{k,l} + \eta \cdot \frac{\partial J}{\partial w_{k,l}} \quad \rightarrow \quad w_{k,l}^{new} = w_{k,l} - \Delta_{k,l}$$

  - Each layer produces δ-values, which are propagated back to the inputs to adjust the parameters in every layer. In the previous illustration, we employed individual biases b_l for each node. However, if we were to use a shared bias across the layer, as in the example, we would simply sum up the deltas for nodes using the same bias, i.e.,:

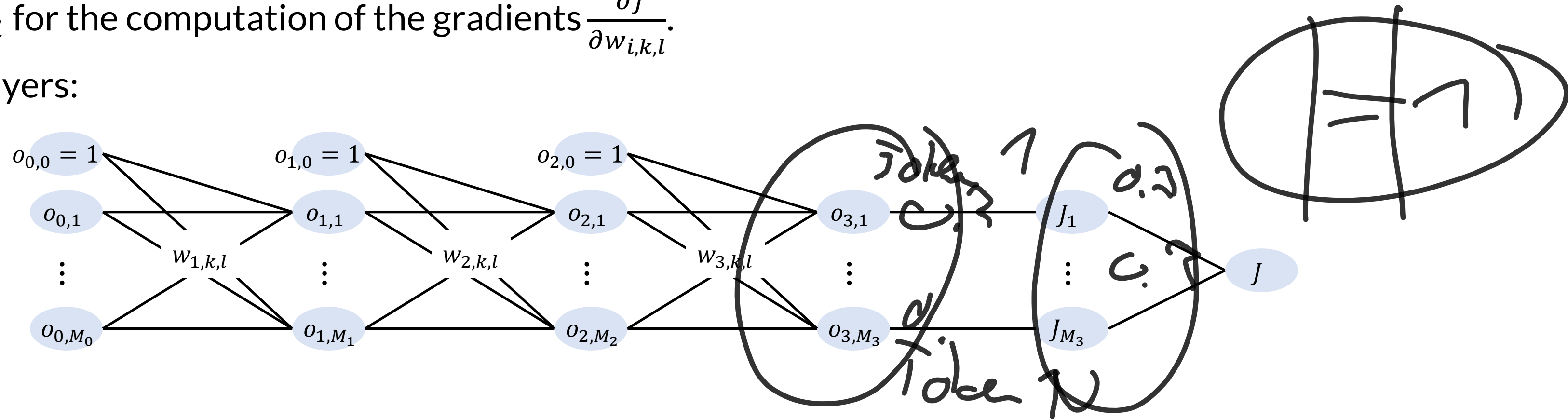


– Hidden layers are computed in a similar manner, but during backpropagation, there are L incoming edges from the subsequent layer. The visual representation of the forward and backward paths is as follows:



– In summary, the backpropagation algorithm is a crucial component of stochastic gradient descent, where we seek the optimal parameters (weights, biases, etc.) for the network. To compute the gradient of these parameters with respect to an error function J, we first use the network in a forward pass to predict the output with the current parameters. Simultaneously, we track intermediate values needed for the backward pass. We then calculate the error for a single sample and propagate the partial derivatives backward through the previous layers. At each layer, we compute Δ-values for the weights to update them. It's essential to note that the previous weights are still required for the preceding layer to compute its partial derivative (as seen in the figure above, the (+)-node relies on weights v_l, m from the subsequent layer).

- **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:



  - The Feed Forward process is defined as follows:

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 $E_k(o_{N,k}; t_k) = \frac{1}{2}\left(t_k - o_{N,k}\right)^2$. We will see further activation functions and error (or loss) functions in the deep learning section.

- Finally, backpropagation (e.g., with logistic activation function and mean square error) is implemented as follows:

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: While it may be tempting to update the weights within the inner loop (step 4), we must retain the old weights for the preceding layer in the next iteration of step 4 to compute $\delta_{i,k}$.

- Modern deep learning models still use fully connected multilayer networks. However, the original approaches from the 1980s and 1990s faced several challenges, which we will address in the deep learning section. The primary issues revolved around numerical problems during gradient computation (such as vanishing and exploding values) and the substantial computational power required for training moderate to large networks. Conversely, smaller networks did not perform effectively in typical classification tasks, and alternatives emerged such as SVM with kernel functions. Eventually, SVM replaced neural networks, resulting in a temporary decline in research on neural networks after the 1990s.

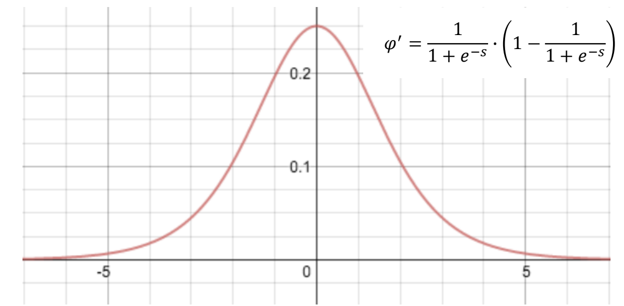
99.2.4 Vanishing and exploding Gradients

- The second wave of neural network research quickly dwindled due to fundamental issues in the learning algorithm. Despite the theoretical capacity of neural networks to learn any function, this often didn't translate into practical success. Adding more hidden layers didn't necessarily improve results, and larger networks became increasingly unstable. The challenges of vanishing and exploding gradients and the competition from support vector machines (SVM) with sophisticated kernels led the field into a deadlock. Only the Canadian government continued to fund neural network research, with Geoff Hinton and his team publishing a breakthrough paper in 2006 on deep belief networks that addressed early backpropagation issues. Simultaneously, the availability of large labeled datasets and the parallel processing power of GPUs significantly accelerated the success of what is now known as deep learning.
- First, let's address the vanishing gradient problem. In the network from the previous section, we had an input layer, a hidden layer, and an output layer. We optimized the network's parameters by minimizing a quadratic cost function. The backpropagation algorithm computes gradients and updates a weight on the first layer with the following formula:

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

  The gradient involves two multiplicative terms, both having factors of the form $x \cdot (1 - x)$ due to the use of the sigmoid activation function. Here, x represents the output of a neuron after the activation function, specifically $x = \varphi(s) = \frac{1}{1+e^{-s}}$. Additionally, these multiplications include the weights of the last layer. When we introduce more hidden layers to the network, we encounter more factors of the form $x \cdot (1 - x)$ and more weights from subsequent layers in the gradients of the first layer's weights and biases.

- The derivative of the sigmoid function $\varphi(s) = \frac{1}{1+e^{-s}}$ is depicted on the right-hand side. Notably, its maximum value is 1/4, and values rapidly diminish on both sides. When we initialize weights between 0 and 1, the gradient computation becomes a sequence of small value multiplications, resulting in very minor updates to weights and biases, even if they are significantly off target. This necessitates a large number of iterations to converge to optimal values, making the learning process slow and resource-intensive.



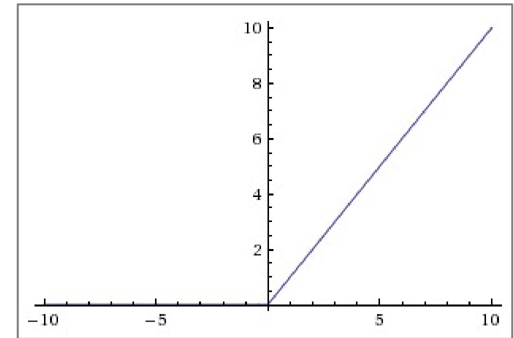
$$(t_1 - o_1) \cdot \underbrace{o_1 \cdot (1 - o_1)}_{\leq 1/4} \cdot \underbrace{w_5}_{\leq 1} \cdot \underbrace{h_1 \cdot (1 - h_1)}_{\leq 1/4} \cdot x_1 \leq 1/16$$

  This results in a quarter reduction of gradients for each layer during backpropagation, causing training of networks with many layers (such as GoogLeNet with around 20 layers) to become exceedingly slow.
- Conversely, when we extend the weights and input values beyond the usual range of -1 to 1, the gradients start to explode because we are now multiplying several values larger than 1. With just a few layers, gradients grow exponentially as they propagate backward, causing the weights and biases to increase in absolute values. This, in turn, leads to potentially even larger gradients in the subsequent iterations, ultimately resulting in unstable gradient computations and causing numerous attempts at deeper networks to fail.

• The breakthrough moment for deep learning was a result of several key factors and developments. It began with the availability of large labeled datasets like ImageNet, which allowed deep learning models to learn from extensive data. This was further empowered by the increased computational power, particularly the use of GPUs, which made training large neural networks efficient. Advanced architectures, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), greatly improved model performance, while innovative activation functions like ReLU helped mitigate the vanishing gradient problem. Regularization techniques, including dropout and L1/L2 regularization, enhanced model generalization, and optimization algorithms like Adam and RMSprop made training more efficient. Transfer learning, where pre-trained models are fine-tuned for specific tasks, accelerated model development. Pioneering research, industry investment, and remarkable success in diverse applications contributed to the resurgence of neural networks, marking a significant breakthrough in artificial intelligence.

- The **rectified linear unit (ReLU)** is a fundamental activation function, replacing the previous sigmoid function. Although various activation functions exist, ReLU played a pivotal role in ensuring more reliable gradient calculations. It is defined as:

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



The function is depicted on the right. What makes it unique? Firstly, it closely resembles the behavior of biological neurons, in contrast to the sigmoid function and hyperbolic tangent, which draw inspiration from probability theory. Secondly, its gradient is either 0 or 1:

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

Therefore, the gradients of the activation function don't accelerate the vanishing and exploding gradients problem. ReLU has become the prevailing activation function in deep learning, despite some associated challenges:

  - With output values no longer confined to the range [0, 1], a challenge arises when mapping the last layer's output to class labels. To address this, the softmax function is employed to transform output values into class probabilities. It is frequently used alongside the cross-entropy loss function to streamline gradient computations, as shown in the following equations. Let o_k represent the k-th output value, and y_k denote the target label. Then:

$$p_k = \frac{e^{o_k}}{\sum_j e^{o_j}} \qquad J(\boldsymbol{\theta}) = -\sum_k y_k \cdot \log p_k$$

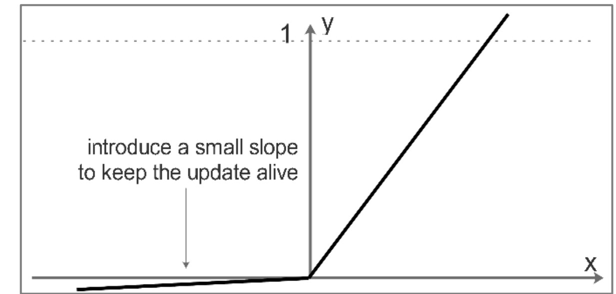
J is defined as the cross-entropy loss function. ***θ*** contains all parameters of the network, i.e., weights and biases.

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

that is simple!

  - Batch normalization (as discussed in the advanced image retrieval chapter) addresses the value range issues by normalizing the activations of each layer during training. This helps in mitigating internal covariate shift, making training more stable and efficient.

- The derivative of ReLU can be 0, stopping backpropagation at that unit, limiting weight adjustments in early layers. While some view this as a network regularization method, resembling the sparse connections in biological neurons, others raise concerns about randomly closed paths due to initial weight and bias selections, hindering the network's learning. An alternative activation function is the **leaky ReLU**, defined as follows (including its derivative, plotted on the right side):



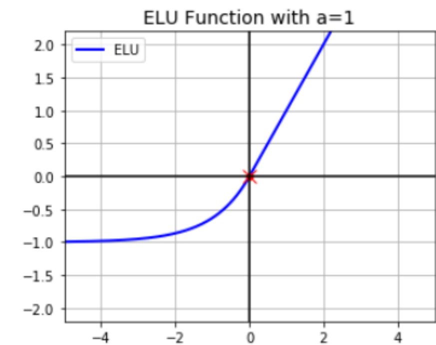
$$\varphi(s) = \begin{cases} 0.01 \cdot s, & s < 0 \\ s, & s \geq 0 \end{cases} \qquad \varphi'(s) = \begin{cases} 0.01, & s < 0 \\ 1, & s \geq 0 \end{cases}$$

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

  - The **parametric ReLU** is the generalization of a leaky ReLU with a learnable parameter α instead of the constant value used for the leaky ReLU above:

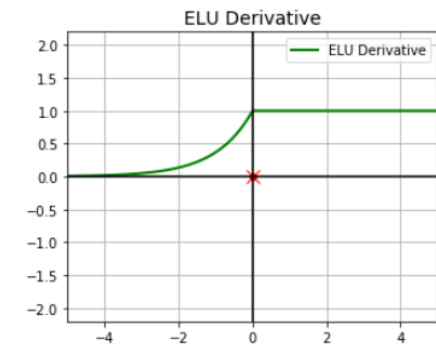
$$\varphi(s) = \begin{cases} \alpha \cdot s, & s < 0 \\ s, & s \geq 0 \end{cases} \qquad \varphi'(s) = \begin{cases} \alpha, & s < 0 \\ 1, & s \geq 0 \end{cases}$$

  - During training, the model learns the parameter for each activation function, which helps it adjust the slope for negative values to better fit the situation.



- ReLU and leaky ReLU have non-smooth derivatives that jump from 0 to 1. In contrast, the **Exponential Linear Unit (ELU)** is a smooth, continuous function with a derivative that allows for updates to negative values:

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

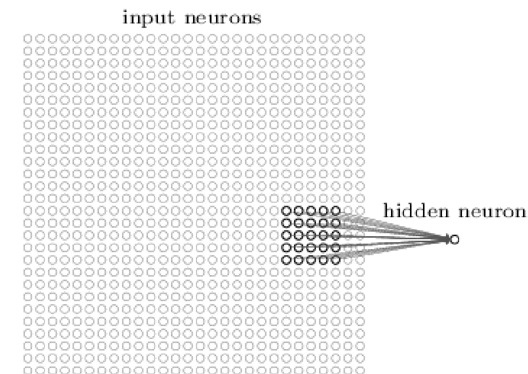


- Look at the right side for a graph of the function and its derivative. The derivative isn't smooth, but it doesn't suddenly change at 0. Instead, the function values keep decreasing gradually as the argument value becomes negative and smaller.

99.2.5 Deep Learning Architecture Improvements

- Deep learning has witnessed remarkable architectural enhancements aimed at tackling the vanishing and exploding gradient issues. Rather than relying on fully connected layers, modern deep networks have adopted a range of sophisticated techniques. These include convolutional layers, which employ only a few weights and biases to connect to numerous output neurons. This design allows for the aggregation of thousands of updates during backpropagation, significantly reducing the overall number of parameters. Additionally, regularization techniques, such as drop-out, have been employed to limit the number of active connections, thus improving training efficiency and curbing the risk of overfitting.
- In the case of recurrent neural networks (RNNs), which initially had issues with the vanishing gradient problem, innovative solutions like gated recurrent units (GRUs) and long short-term memory (LSTM) cells have been instrumental. These innovations have empowered RNNs to capture long-range dependencies within sequential data. Furthermore, the integration of bidirectional RNNs and attention mechanisms has significantly enhanced their contextual understanding.
- Concurrently, transformers have revolutionized the domain of natural language processing. They introduced self-attention mechanisms, allowing models to weigh the importance of different input elements dynamically. Exemplified by models like BERT and GPT-3, transformers have demonstrated their superior capabilities in pre-training and fine-tuning tasks, enabling them to better comprehend and generate human-like text. To make these advancements more accessible and versatile, both RNNs and transformers have embraced efficiency, parallelization, and scalability. These steps have ensured their practicality in a broad spectrum of real-world applications, from machine translation to image captioning. The ongoing evolution of deep learning architectures signifies substantial progress in the field, ushering in breakthroughs across multiple domains.
- We omit here a detailed description of all architectural improvements and refer to the chapter on advanced text retrieval and advanced image retrieval for selected examples of architectural improvements.

- **Changing the network structure** by cutting down on parameters may not always be the best option. It can make it harder for the network to handle complicated tasks, and smaller networks have not always worked well. But we can tweak the network structure to get more weight updates and add extra paths for updating the first layers.
  - So far, we have been looking at layers that are fully connected to the previous layer. Each connection had its own weight, and neurons had their own bias or a shared bias. However, when it comes to understanding images, our eyes use receptive fields to pick out details from nearby areas. These fields work the same way regardless of what we are looking at.



    - In the past, images were prepared for learning using various methods (like Gaussian, Sobel, HOG) that provide receptive field inputs into the network. However, this approach also placed limits on how much a network could learn and required human intervention that may not transfer from one classification task to the next.
    - Deep learning adds a new layer called the **convolutional layer**. This layer connects a small spatial neighborhood (see figure above on the right, 5x5 input neurons) to a hidden neuron. This happens for every location in the matrix, resulting in a hidden layer of the same size (padding is used at the boundaries). The output of the neuron takes the form of a 2-dimensional convolution with a trainable kernel:

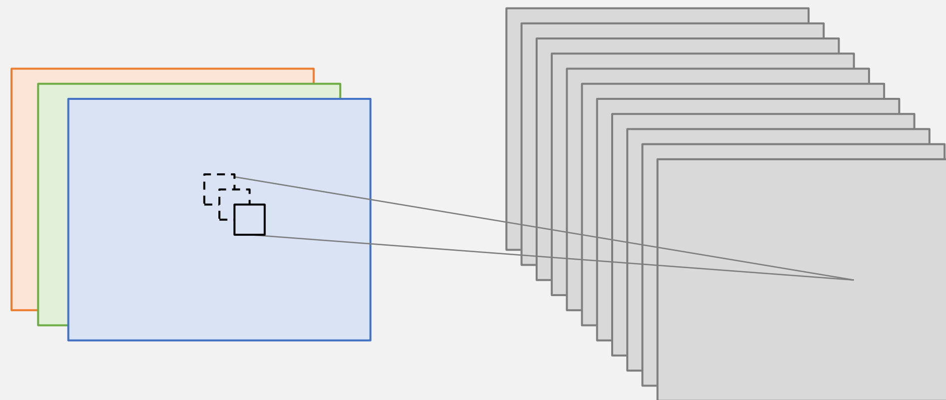
$$o_{i,j}(\boldsymbol{x}) = \varphi\left(b + \sum_{k,l} w_{k,l} \cdot x_{i+k,j+l}\right)$$

    - One interesting aspect is that the weights and bias are shared by all the neurons in the new layer. This means we can train any kernel that provides useful features for the classification task, and we also get a lot of updates on weights and biases through backpropagation. For example, in a 1024x1024 layer, we get more than 1 million updates for the weights and the bias, one for each pixel. The convolution takes all input channels into account and allows for any number of output features. So, the output at the hidden neuron is not just a single value, but a multi-dimensional vector that can be used as the input for the next layer. Another interesting point is that we can remove the human interaction mentioned earlier and create a more general structure that can adjust to various tasks. We will look at some sample structures later in the chapter.

- The convolutional layer typically takes a 2-dimensional input vector with M dimensions and produces a 2-dimensional output vector with N dimensions. For example, in image classification, we might begin with 3 channels ($M = 3$) and generate 20 features per pixel ($N = 20$). The convolution function is a mapping from an M-dimensional input vector to an N-dimensional output vector. At a pixel location (x, y), we get:

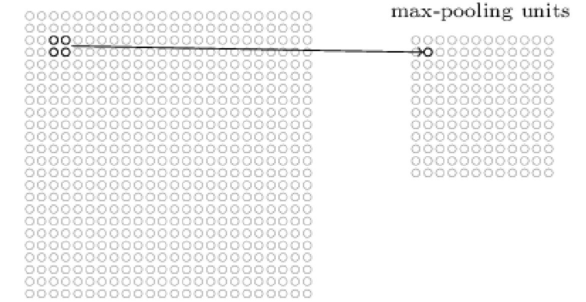
$$o_{i,j,n}(\boldsymbol{x}) = \varphi\left(b_n + \sum_{k,l,m} w_{k,l,m,n} \cdot x_{i+k,j+l,m}\right)$$

- Let's say we have a 5x5 convolution on $M = 3$ input channels, and we want $N = 20$ output features. The formula above has shared biases b_n for each output feature, and shared weights $w_{k,l,m,n}$ for each position in the 5x5 window, for each channel, and each output feature. So, we end up with 20 biases and 1500 weights. This is true no matter the size of the input images. On the other hand, a fully connected layer would need separate biases $b_{k,l,n}$ and weights $w_{k,l,n,i,j,m}$ for each output field, feature, each input field, and channel. With a 256x256 image, this would require a huge number of biases and weights, making it impractical for even small images. Convolution layers, on the other hand, can work with any image size without increasing the number of parameters (although the cost of running the program goes up as the number of pixels increases). This reduces the number of parameters on the one hand, and provides a general structure that can learn the best features for the classification task.



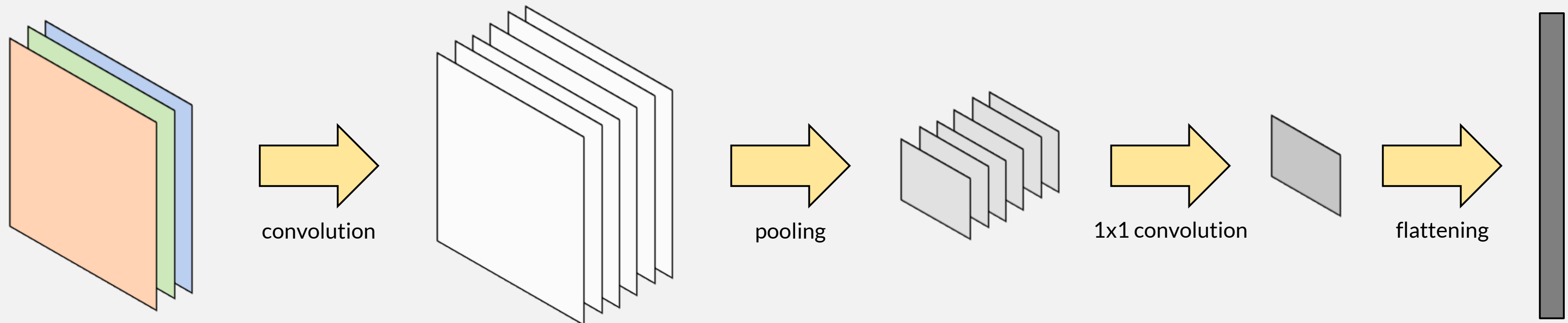
- **Strides:** Convolution uses a sliding window to calculate an output value at each location. It's possible to specify how far apart two consecutive windows should be. A stride of (2,2) means that only every other value in both dimensions is used as the starting location of the window. This results in only half as many rows and columns in the output. Strides can be used to decrease the initial size of the network. A (2,2) stride will result in 4 times fewer output neurons. This can be useful for scaling down the size of images and computing features at different scales.
- **Padding:** Convolutional layers usually use odd numbers for the height and width of the kernel. By using 0s for the values outside the input matrix, we can keep the same dimensions for the output layer as the input layer.
- **Sizing:** Convolution can handle images of any size, but the network architecture needs the final 2D layer to connect to a group of output neurons that generate the classification result (softmax). Strides and pooling layers gradually decrease the dimensionality until it's possible to use a fully connected network to map from a 2D layer to a 1D classification layer without the need of a large number of parameters.
- The **1x1 convolution** is used to decrease the size of the feature values and the number of parameters in the next layers. For example, if we want to learn a 5x5 convolution with 20 output features and 20 input features, we would need to learn 10000 weights and 20 biases (a total of 10020 parameters). Using a 1x1 convolution can reduce the number of parameters to learn.
  - To start, we can use a 1x1 convolution to create 3 output features from the 20 input features. This layer needs 60 weights and 3 biases, for a total of 63 parameters.
  - Next, we input the 3 features from the 1x1 convolution into a 5x5 convolution with 20 output features. This requires 1500 weights and 20 biases, for a total of 1520 parameters.
  - The new network structure has 1583 parameters, while the old one had 10020 with a simple mapping.

- Convolution layers are often followed by **Pooling Layers**. Pooling summarizes the values of a neighborhood and reduces the number of neurons for the next layer.
  - For instance, look at the picture on the right side. A 2x2 **max-pooling layer** maps the highest value in the 2x2 window to its output neuron. If we also use a stride of (2,2), this makes each dimension 4 times smaller. If the input has multiple channels, the pooling operator is used on each channel separately. In this case, we don't use an activation function.



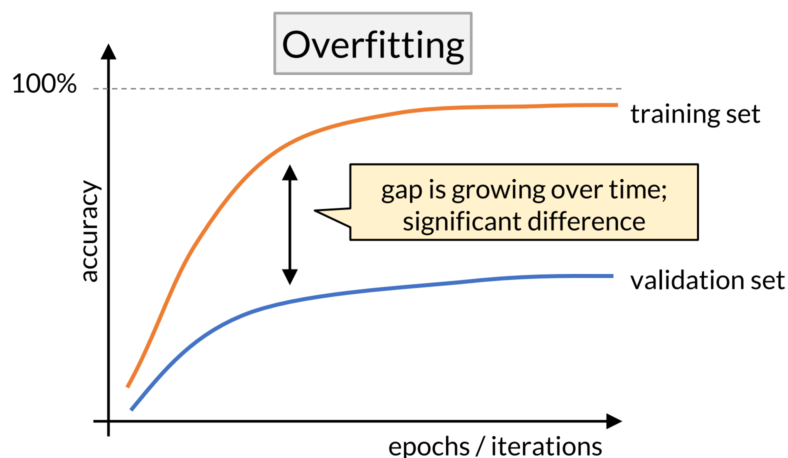
$$o_{i,j,n}(\boldsymbol{x}) = \max_{l,k} \ x_{i+k,j+l,n}$$

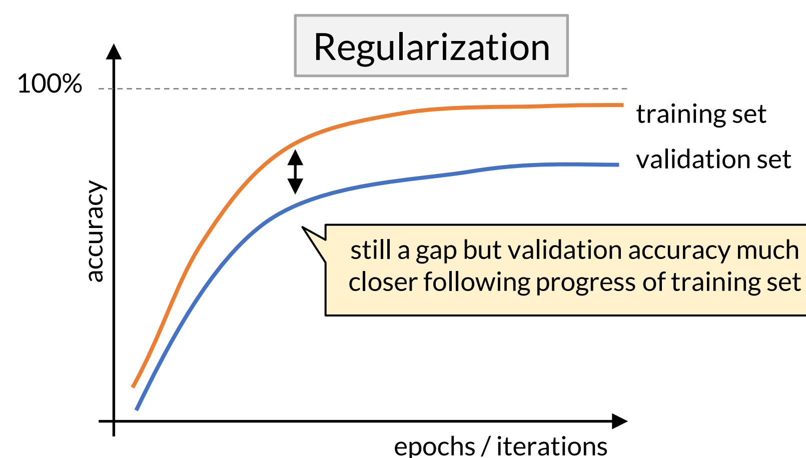
  - Other types of summarization functions can be used alongside max pooling. Some common examples are average pooling and 2-Norm pooling. Pooling layers are crucial for controlling the size and the number of parameters in the network model. This significantly cuts down on computation and the risk of overfitting. It's important to note that pooling only reduces spatial dimensions if the stride is larger than 1. However, it doesn't reduce the number of features. To do that, a 1x1 convolution is needed, as mentioned earlier.
  - **Global pooling** simplifies a whole feature map to one value, eliminating the need for a fully connected layer.
- A **flattening layer** is used to change the final 2D layer (usually the output of a pooling layer) into a single 1D vector for each feature value. To reduce the feature dimensions to 1, an extra 1x1 convolution is needed.
- In image classification, we use convolution, pooling, and flattening layers together. This allows us to handle large images with fewer parameters. The basic structure is as follows:



99.2.6 Regularization

- Regularization is a crucial aspect of deep learning to mitigate overfitting to the training data, especially when using millions or billions of parameters:
  - As discussed in the first section of this chapter, overfitting arises when the model contains an excessive number of parameters, allowing it to memorize data rather than deduce general patterns from it. To address this issue and identify overfitting problems, we need strategies to prevent the network from merely memorizing the input-to-target mapping.
  - Overfitting is a failure to generalize which becomes apparent when we apply a trained model to new data not included in the training process. We can utilize a validation set as follows to detect common signs of overfitting:
    - Nearly perfect accuracy on the training set during training.
    - A considerably lower accuracy on the validation set at the end of training.
    - A widening gap between training accuracy and validation accuracy as training progresses.
    - Sometimes: phase of decreasing validation accuracy after a phase of progress.





- We can choose from various regularization methods.
  - **Modifying the network structure** by reducing the parameters is not always a viable choice, as it limits the ability to learn complex tasks, and small networks have shown limited success.
  - **Increase the training set** by augmenting existing data, such as applying small rotations, adjusting brightness, adding noise, using Gaussian filters, and more. These modifications can significantly expand the training data, increasing the dataset without requiring additional labeling.
  - **Revised learning strategies** with enhanced learning algorithms, weight adjustment decay, and early stopping have yielded promising outcomes for large-scale networks.
  - **Modify the cost function** to favor simpler models. An effective approach is to introduce a penalty into the cost function for utilizing large weights. Smaller weights, ideally approaching zero, reduce model complexity. This allows us to strike a balance between training overfitting and penalizing more intricate models. Our cost function now appears as follows (L2 regularization):

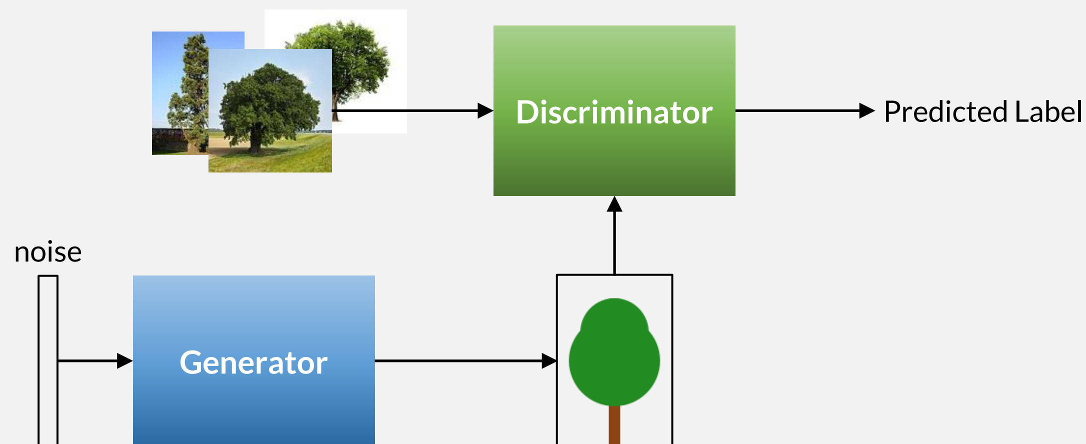
$$J_{reg}(\boldsymbol{\theta}) = J(\boldsymbol{\theta}) + \frac{\lambda}{2 \cdot |\mathbb{T}|} \sum_i w_i^2$$

With $|\mathbb{T}|$ representing the number of training samples and $\lambda > 0$ as the regularization parameter, it's important to note that we apply penalties to the weights, not the biases. This results in a modified update for w_i during backpropagation. Let Δ_i denote 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 introduces a weight decay factor of $\left(1 - \frac{\eta\lambda}{|\mathbb{T}|}\right)$ for each weight, causing them to decrease over time unless the gradient offsets this effect by increasing the weights during learning. This technique has proven effective in significantly reducing the risk of overfitting.

- **Increase the training set:**
  - In general, bigger models with more parameters need more data to avoid overfitting and to help the model learning generalization rules. For example, a large language model with 1 billion parameters (the biggest models are now trying out 1 trillion parameters) needs 5 to 10 times more data points than parameters, which is 5 to 10 billion labeled data points. To get this much data, model designers introduced self-supervised training: the model is trained with masked sequences from a text corpus and has to predict the masked terms. Another option is to predict the next term in a sequence of terms, which can also come from a large text corpus with self-supervision.
  - To classify images, we can make changes like cropping, rotating slightly, adjusting brightness, adding noise, using Gaussian filters, and more. These changes can greatly increase the training data, expanding the dataset without needing more labeling and make it more robust (reduce variance) against small changes in the images.
  - In NLP, the foundational models are trained on a general next / masked token task in a self-supervised manner. We can then use these models and apply fine-tuning with extra layers and task-specific data. Because the foundational model has a general understanding of language, the fine-tuning requires much less data to optimize.
  - In 2016, Ian Goodfellow introduced the idea of **Generative Adversarial Networks (GAN)**. When creating new data like images, you have two models competing with each other: 1) a generator that creates an image from noise, and 2) a discriminator that distinguishes between real and fake (i.e., generated) images. As the two models compete with each other, they become better at generating images and at telling fake from real ones over time. The discriminator tries to maximize the chance of giving the right label to both real and generated samples, while the generator tries to minimize the chance of the discriminator's right answer.



- **Modify the cost function: L1/L2 Regularization**
  - An effective approach is to introduce a penalty into the cost function for utilizing many weights and large weights. Smaller weights, ideally approaching zero, reduce model complexity. This allows us to strike a balance between training overfitting and penalizing more intricate models. Our cost function now appears as follows (**L2 Regularization**):

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

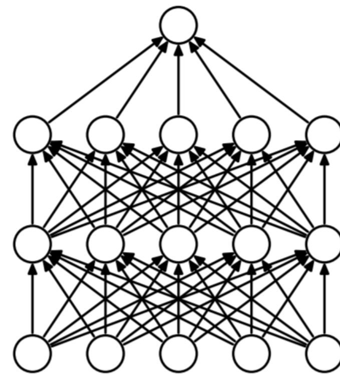
    With $|\mathbb{T}|$ representing the number of training samples and $\lambda > 0$ as the regularization parameter, it's important to note that we apply penalties to the weights, not the biases. This results in a modified update for w_i during backpropagation. Let Δ_i denote the update for w_i without regularization, and η be the learning rate, then:

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

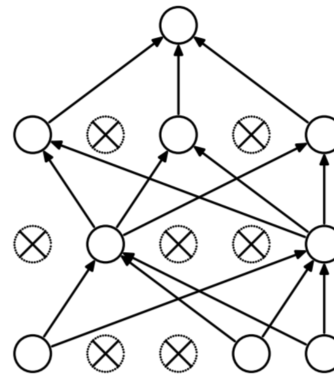
    Regularization introduces a weight decay factor of $\left(1 - \frac{\eta\lambda}{|\mathbb{T}|}\right)$ for each weight, causing them to decrease over time unless the gradient offsets this effect by increasing the weights during learning. This technique has proven effective in significantly reducing the risk of overfitting. The hyperparameter λ stays the same during training but can be changed through hyperparameter optimization. On the other hand, η can change over the epochs.
    - Fixed Schedule: learning rate η remains the same for all epochs
    - Learning Rate Decay: Gradually decrease the learning rate. You can do this by following a set schedule (for example, every few epochs) or by considering specific conditions (such as when your performance levels off).
    - Adaptive Methods: Utilize adaptive learning rate techniques like Adam, Adagrad, RMSprop, etc., to adjust the learning rates for each parameter based on past gradients or other factors.
  - L1 Regularization adds up the absolute weights, while L2 Regularization uses squared weights in the loss function.

- **Advanced Learning Strategies:**
  - New learning techniques improve Stochastic Gradient Descent (SGD) to make training neural networks more efficient and effective. Momentum, RMSprop, and Adam are methods that address issues with traditional SGD. Momentum uses past gradients to speed up convergence and handle shallow or noisy gradients. RMSprop adjusts learning rates for each parameter separately, giving better results in changing environments. Adam combines momentum and RMSprop, using adaptive moment estimates and per-parameter learning rates.
  - Batch Normalization (BN) and Layer Normalization (LN) are techniques used in deep learning to improve the training stability and accelerate the convergence of neural networks. They normalize the input to a layer, mitigating issues like vanishing or exploding gradients, and allowing for more stable and faster training.
    - The sigmoid activation function limits outputs to a range of 0 to 1, which then become inputs for the next layer. However, using ReLU allows for values to become very large or very small without any limits. This can cause values to explode and disrupt the training process. To address this issue, batch and layer normalization apply a Gaussian normalization before the values enter the next layer: $\bar{x} = (x - mean(x_i))/\sqrt{var(x_i) + \epsilon}$. This normalization changes the input values to a normal distribution with a mean of 0 and a standard deviation of 1.
    - **Batch normalization (BN)** uses the values of the mini-batch samples in the current training iteration to estimate the distribution and applies normalization to each input value separately. After training, the mean and variance values are kept as model parameters for inference. To accurately estimate the population mean and variance, larger batch sizes are needed. This can make it more challenging to train networks for tasks like object detection and semantic segmentation, which typically involve high input resolution.
    - **Layer normalization (LN)** uses the values from the entire layer and the mini-batch samples in the current training iteration to estimate the distribution and apply the same normalization for all input values. The mean and variance values are then kept as model parameters for inference after training. Group Normalization is similar to Layer Normalization in that it is applied along the feature direction. However, it divides the features into specific groups and normalizes each group separately. In practice, Group Normalization has been found to perform better than Layer Normalization, and its parameter "num_groups" is adjusted as a hyperparameter.
    - **Weight Standardization (WS)** is transforming the weights of any layer to have zero mean and unit variance. Often, this method is combined with either Batch or Layer normalization to achieve stable training conditions.
    - Generally, batch normalization works better in image classification with convolutions, while layer normalization is often used in recurrent neural networks (RNN) and transformer architectures.

- The **Dropout technique** involves heuristic adjustments to the network structure during learning. At any given time, only a portion of the network is active, with nodes randomly selected for activation. This selection can vary throughout the learning process.
  - During each training step, nodes are dropped out with a probability of $1 - p$. This results in different sets of active nodes learning from the training examples over time.
  - Feed forward: If a node is dropped out, its output value is set to 0, but weights and biases are retained, as the node may become active again in subsequent training steps.
  - Back propagation: When a node is dropped out, it no longer propagates changes, and the weights of connections to/from such a node do not receive updates.
  - The final prediction model utilizes all nodes but adjusts their weights by $(1 - p)$. The dropout technique can be seen as training multiple networks concurrently. These individual networks are then combined into a larger network. This approach helps mitigate overfitting, as each subset of the network adapts differently to the training data. By "averaging" the networks for prediction, the impact of overfitting in one subset is balanced by the other subsets, which may have overfitted other aspects of the training data.



(a) Standard Neural Net



(b) After applying dropout.

99.3 Clustering

- With unsupervised learning tasks, the machine learning algorithm observes a 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 finite 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
  - Vector search techniques based on product quantization often use k-means clustering to map several dimensions to a small number of bits.
  - 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.

- **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 $\boldsymbol{x}_1, \dots, \boldsymbol{x}_N$. We then want to partition the data items into K sets $\mathbb{S} = \{\mathbb{S}_1, \dots, \mathbb{S}_K\}$ 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 $\boldsymbol{\mu}_1^{(0)}, \dots, \boldsymbol{\mu}_K^{(0)}$ (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., $\left\|\boldsymbol{x} - \boldsymbol{\mu}_k^{(t)}\right\| \le \left\|\boldsymbol{x} - \boldsymbol{\mu}_l^{(t)}\right\| \ \forall l: 1 \le l \le K$ (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 optimum.

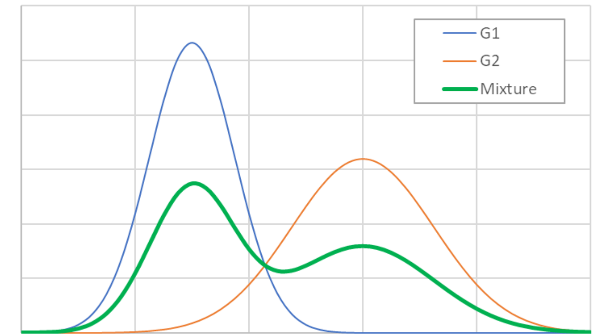
- Let us start with a simple one dimensional example with a mixture of two ($K = 2$) Gaussian distributions $\mathcal{N}(\mu_k, \sigma_k^2)$. The picture on the right shows the two Gaussian distributions and their mixture. With an infinite number of Gaussians, a mixture can model any distribution. Each 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 $\mathbb{T} = \{x_1, \dots, x_N\}$. 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:

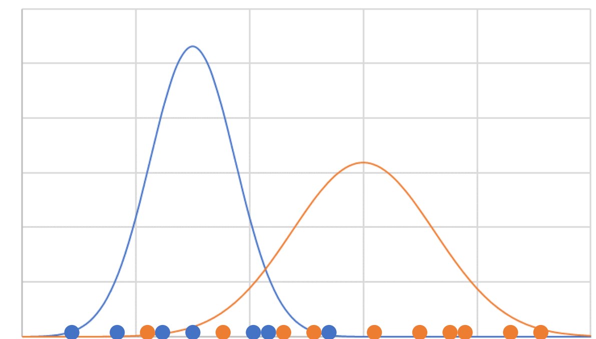
$$\mu_k = \frac{\sum_{x \in \mathbb{S}_k} x}{|\mathbb{S}_k|} \qquad \sigma_k^2 = \frac{\sum_{x \in \mathbb{S}_k} (x - \mu_k)^2}{|\mathbb{S}_k|} \qquad P(C_k) = \frac{|\mathbb{S}_k|}{N}$$

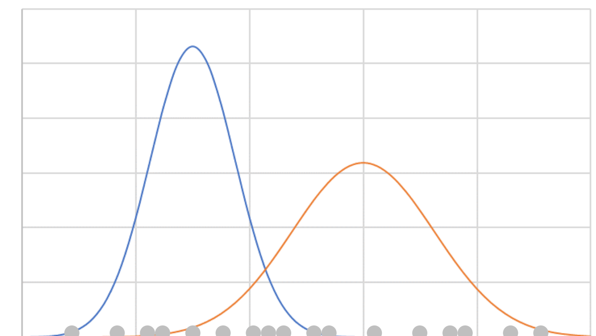
- On the other side, assume we know the parameters μ_k, σ_k^2 of the distributions and the priors $P(C_k)$, can we estimate the probability $P(C_k|x_i)$ that a point x_i is part of cluster C_k?

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

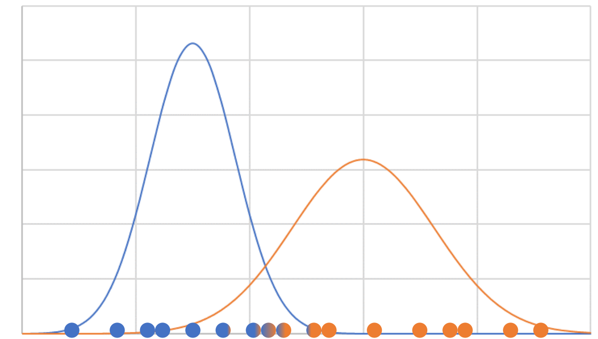
with $\quad P(x_i|C_k) = f(x_i;\ \mu_k, \sigma_k^2) = \frac{1}{\sqrt{2\pi\sigma_k^2}} \cdot \exp\left(-\frac{(x_i - \mu_k)^2}{2\sigma_k^2}\right)$







- 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 $1 \leq k \leq K$
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

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

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

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

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:

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

- Where does the name Expectation Maximization come from? Let $\mathbb{X} = \{\boldsymbol{x}_i\}$ be the set of data items and $\mathbb{Y} = \{w_1, \mu_1, \sigma_1, \dots, w_k, \mu_K, \sigma_K\}$ 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.

- 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

- 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:

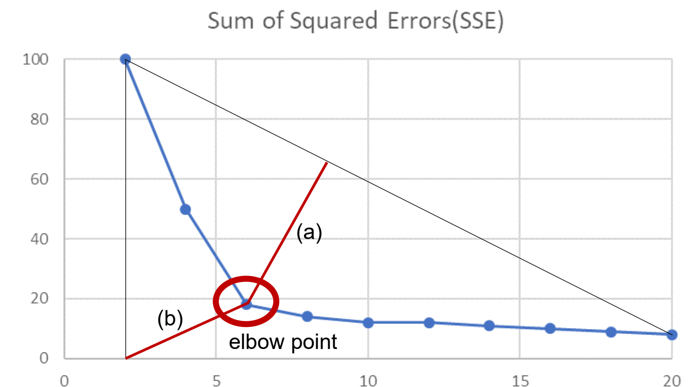
$$SSE(\text{k–means}) = \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in \mathbb{S}_k} \|\boldsymbol{x} - \boldsymbol{\mu}_k\|_2^2 \qquad SSE(\text{EM}) = \sum_{k=1}^{K} \sum_{\boldsymbol{x} \in \mathbb{S}_k} (\boldsymbol{x} - \boldsymbol{\mu}_k)^T \boldsymbol{\Sigma}_\text{k}^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_k)$$

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.



99.4 Naive Bayes

- Naive Bayes comes from a line of probabilistic thinking that begins with Thomas Bayes, the eighteenth century minister and mathematician whose work on inverse probability led to what is now Bayes theorem. Statisticians turned his idea into a general method for updating beliefs when new evidence appears. In the twentieth century this method became important in pattern recognition, where researchers needed models that handle uncertain observations yet stay computationally simple. The Naive Bayes classifier is a practical compromise: it uses Bayes rule but assumes the features describing an instance are conditionally independent given the class label. That assumption is rarely exactly true, but it simplifies the joint probability so the model is fast, flexible and easy to train.
- The model computes how likely the features are under each class and chooses the class with the highest posterior probability. Its appeal is that each feature provides a small piece of evidence, so the classifier can still work when data are sparse. In the early years of information retrieval research, this robustness drew attention. Text collections are high dimensional and dominated by rare terms. Many models struggle in that setting, but Naive Bayes handles it well because it breaks the likelihood into independent contributions from each term frequency or binary occurrence. Learning becomes counting how often terms appear in documents of each class and normalizing those counts. This estimation scales to large corpora and updates naturally when new documents arrive.
- In retrieval tasks that require classification, Naive Bayes sits between indexing and ranking. When a system must decide a document's category, the classifier can use term statistics the index already stores. Its probability-based structure also fits the retrieval tradition of treating scores as estimates of relevance. Some studies have linked Naive Bayes to probability-based retrieval models, letting its simple probability model guide ranking and relevance feedback. Even though modern systems often use neural methods, Naive Bayes remains a useful baseline because it is easy to interpret and its assumptions are clear. It also performs well in low resource situations, for example when only a few labeled examples exist for each class.
- Although it is simple, the model is elegant. By treating features as independent, it turns a potentially intractable problem into one that is solvable and practical. Its long use in classification and retrieval shows a lasting lesson in applied machine learning: a model does not have to mirror the worlds complexity to be useful. It must instead provide stable, scalable estimates that support decisions across domains. Naive Bayes meets that standard, so it remains in toolkits even after more advanced methods have appeared.

- 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's 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 $P(C_k)$ and the likelihood $P(x_j|C_k)$, 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.

- **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 in practice.
  - To find the probability distribution $P\left(x_j \middle| C_k\right)$ 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 $N_k\left(x_j = v\right)$ 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

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

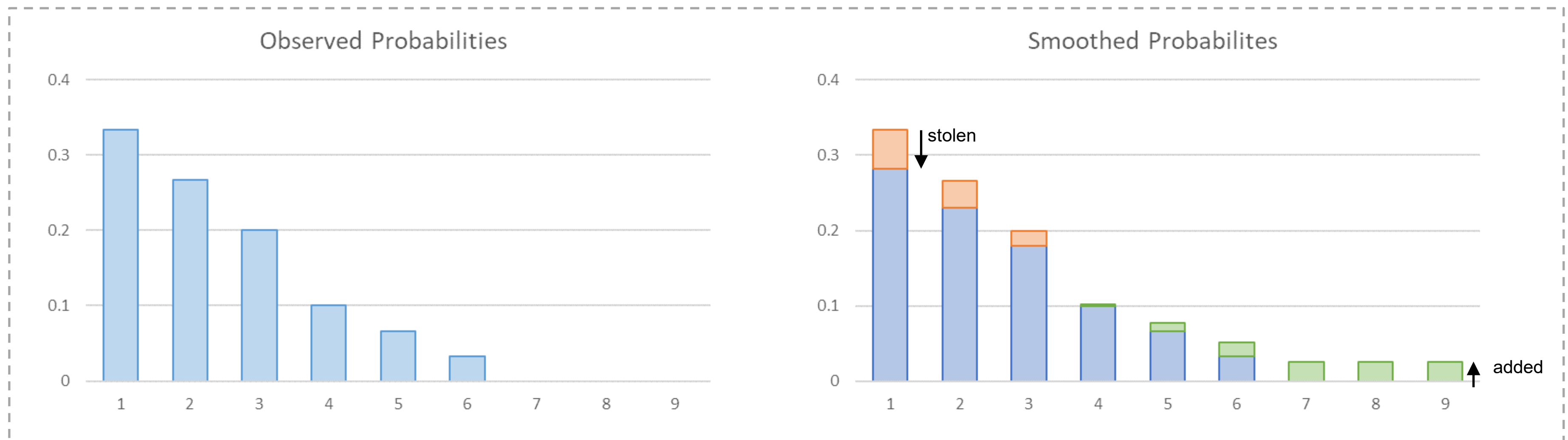
– What if a value v is never seen for x_j over a class C_k. Obviously, $P(x_j = v \mid C_k) = 0$ and with that:

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

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. An example from text classification is 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:

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

  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 $\mathcal{N}\left(\mu_{k,i,l}, \sigma_{k,i,l}^2\right)$ with weights w_l:

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

  To learn the parameters of the normal distributions, we can use the **Expectation Maximization** approach (see 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 well, but cannot generalize 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 impact the ordering:

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

  - 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.

99.5 Hidden Markov Models

- Sequential data appears whenever observations unfold over time, from acoustic signals in speech recognition to DNA sequences in bioinformatics. Early work on random processes introduced Markov chains to capture dependencies between successive events. These models assume the future depends only on the present state. Many real systems hide their true condition. Speech is produced by articulatory states that cannot be measured directly, and biological processes often involve hidden functional states that give rise to observable residues or signals. To describe these invisible dynamics while modeling their visible outcomes, researchers developed Hidden Markov Models (HMMs). HMMs extend Markov chains by pairing each hidden state with a probability distribution over possible observations. As speech recognition grew more complex in the late twentieth century, HMMs became a natural and mathematically grounded tool for modeling the temporal structure of spoken language.
- The classical Markov assumption says that for a sequence of states $X_1, X_2, \ldots, X_T$, the probability of the next state depends only on the current state. Formally, $P(X_t \mid X_{t-1} \ \ldots X_1) = P(X_t \mid X_{t-1})$. Such models are adequate when the states themselves can be observed. For example, a simple weather model with states sunny, cloudy, and rainy describes how weather changes if we directly see those states. Many systems provide only indirect evidence. A speech recognizer measures acoustic features rather than phonetic labels. This motivates hidden state models. In these models a hidden state sequence X_t evolves as a Markov chain and emits observable symbols O_t according to emission probabilities. The observations are noisy reflections of the hidden states. For example, sensors may produce signals influenced by the true weather while the true weather remains hidden.
- In speech recognition, Markov models treat hidden states as linguistic units like phonemes, which represent distinct speech sounds. Phonemes are not directly observable; the system captures acoustic features from the audio waveform, and those features serve as the observations. Hidden states follow a Markov process, so the probability of the current phoneme depends only on the previous one. The observed acoustic signals are produced in a probabilistic way by each hidden phoneme, because the sound of a phoneme can vary across speakers, background noise, and other factors.
- This setup lets a speech recognizer represent how speech sounds change over time as a sequence of hidden phoneme states, using the recorded sound features as indirect evidence. The Markov assumption simplifies the model by assuming each phoneme depends mainly on nearby phonemes, and the probabilistic link between hidden states and observed sounds accounts for the variability and uncertainty in real audio recordings.

- An HMM consists of
  - a finite set of hidden states x
  - a set of observable symbols o
  - a matrix of transition probabilities $a_{ij} = P(X_t = j \mid X_{t-1} = i)$
  - a set of emission probabilities $b_j(o) = P(O_t = o \mid X_t = j)$
  - and an initial distribution $\pi_i = P(X_1 = i)$
- Given a state sequence $x_1, \dots, x_T$ and an observation sequence $o_1, \dots, o_T$, the joint probability factors into the product of initial, transition and emission terms

$$P(x_1, \dots, x_T, o_1, \dots, o_T) = \pi_{x_1} \cdot b_{x_1}(o_1) \cdot \prod_{t=2}^{T} a_{x_{t-1}} \cdot b_{x_1}(o_t)$$

- This factorization shows that hidden states follow the Markov property and that observations are independent given those states. In speech recognition, the hidden states can represent phonemes, and each state generates acoustic feature vectors from a probability distribution, often modeled by a mixture of Gaussians or other parametric models.
- Working with an HMM involves three main tasks:
  - First, compute the probability of an observation sequence to measure how well the model explains the data.
  - Second, find the most likely sequence of hidden states that could have produced the observations. This is important for applications such as phoneme recognition, where the hidden states are the desired output.
  - Third, estimate the model parameters from data, whether the hidden states are known or must be inferred.

  These three tasks form the analytical core of HMMs. Without efficient algorithms for likelihood calculation, decoding and parameter estimation, HMMs would be impractical for real-world sequences.

- Before introducing decoding, review the ideas behind dynamic programming. Many optimization problems have optimal substructure: the best solution to a large problem can be built from best solutions to smaller subproblems. These subproblems also often overlap. Instead of listing all possible state sequences, dynamic programming stores intermediate results and reuses them, saving a lot of computation. For HMMs, finding the most probable path in a state space of size N for a sequence of length T would otherwise require evaluating N^T possibilities. Dynamic programming reduces this to a manageable time by using the recursive structure implied by the Markov property.
- The decoding problem is to find the hidden state sequence $x_1, \dots, x_T$ that maximizes the joint probability with the given observations. The Viterbi algorithm defines variables $\delta_t(j)$ that give the probability of the best state sequence ending in state j at time t. The Viterbi method is an efficient and intuitive dynamic programming procedure:
  - Initialization is

    $$\delta_1(j) = \pi_j \cdot b_j(o_1)$$

  - For each time $t > 1$, the recursion is

    $$\delta_t(j) = b_j(o_t) \cdot \max_i \left(\delta_{t-1}(i) \cdot a_{ij}\right)$$

    which picks the best previous state i. At the same time, backpointers $\psi_t(i)$

    $$\psi_t(i) = \operatorname*{argmax}_i \left(\delta_{t-1}(i) \cdot a_{ij}\right)$$

    record which previous state gave that maximum.
  - After reaching time T, the algorithm picks the state with the largest $\delta_T(j)$ and backtracks along the ψ_t pointers to recover the full optimal sequence.

- How well you can train a hidden Markov model depends on the data you have. With labeled data, where each observation sequence comes with its true hidden state sequence, estimating parameters is simple: count state transitions and emissions, then normalize to get transition and emission probabilities. In the unsupervised case only the observations are available. The Baum-Welch algorithm uses expectation maximization (EM): it computes expected state occupancies and transitions and then updates parameters to increase the data likelihood. Viterbi training is an alternative. It finds a single best state sequence with the Viterbi algorithm and updates parameters from that hard alignment. Its updates are simpler but the method can get stuck in local optima. In speech recognition both approaches have been used, although EM remains the standard for robust estimation.
- Using HMMs for long sequences creates numerical problems. Repeated multiplications of small probabilities cause underflow, so implementations use log probabilities to keep values stable. Initialization is also important in unsupervised training because poor starting parameters can make the algorithm settle in a suboptimal region of the parameter space. Despite these issues, HMMs work well in many fields. In speech tagging they map acoustic signals to phoneme or word labels. In gene prediction they represent hidden biological states that produce characteristic nucleotide patterns. For temporal segmentation they model transitions between phases of an event, for example activity cycles in sensor data. The Viterbi algorithm is the standard method for decoding in all of these cases.
- Classical HMMs assume first order Markov dynamics and independent emissions. Several extensions relax these assumptions. Higher order models capture dependencies that span multiple time steps. Hierarchical HMMs add structure at different levels of abstraction to reflect complex temporal patterns. Discriminative versions change the training goal to focus on classification accuracy rather than fitting a generative model. Contemporary sequence models, such as neural networks, often outperform HMMs on large scale tasks. Still, the clarity and mathematical elegance of HMMs make them useful for applications that require interpretability. The path from Markov chains to hidden state models shows how a simple idea becomes a versatile tool for understanding sequential data.

99.6 Literature

- Tom M. Mitchell, **Machine Learning**, 1997, McGraw-Hill Science/Engineering/Math, ISBN: 0070428077
- Ian Goodfellow, Yoshua Bengio, Aaron Courville, **Deep Learning**, 2016, MIT Press, online version: https://www.deeplearningbook.org/
- Various authors, **Dive into Deep Learning**, 2023, to be published at Cambridge University Press, online version: https://d2l.ai/
- **MLOps: Machine Learning Life Cycle**, 2022, https://www.ml4devs.com/articles/mlops-machine-learning-life-cycle/