**Exam notes**

**Machine learning**

A model M is called a machine learning model if its performance (P) at a specific class of tasks (T) improves with experience (E). It modifies/adapts its actions (for example: making predictions, controlling a robot) so that these actions get more accurate (thereby improving), where accuracy is measured by how well the chosen actions reflect the correct ones.

* The computational complexity of a machine learning algorithm is important because we might want to use them on very large datasets. Polynomial complexity in the size of the dataset will be a problem.
  + The complexity of training 🡪 Training doesn´t happen very often, and is not usually time critical, so it can take longer.
  + The complexity of applying the trained algorithm 🡪 We want a decision about a test point quickly, and there are potentially a lot of test points, so this needs to have low computational cost.
* Testing 🡪 The only way to know how successfully the algorithm has improved is to compare the predictions with known target labels, which is how training is done for supervised learning. We need a training set and a test set. We use the test set to examine how well the algorithm does on unseen data.
* Overfitting 🡪 When we (over)train our algorithm to fit too well to our specific data set, so that it doesn´t generalize as well as it should. The problem can happen if we define too strict decision boundaries.
* Underfitting 🡪 When we (under)train our algorithm so that it performs poorly on the test set. The problem can happen if we define too loose decision boundaries.
* Confusion matrix 🡪 In classification problems, we use a simple matrix/table to visualize the relationship between the predicted labels and the true labels for each class.
  + Accuracy metrics 🡪 To analyze the results of our training, we can use several accuracy measures. Calculating accuracy alone does not always give us enough information. Therefore we have two additional pairs of measurement strategies: a) Sensitivity and specificity and b) precision and recall. There is to some extent an inverse relationship between precision and recall. If the number of false positives increases, the number of false negatives decreases (meaning that the algorithm is using a broader definition of that class) Positives = class 1, Negatives = class 2. True positives = TP, False positives = FP, True negatives = TN, False negatives = FN. N = the size of our test data.
  + Accuracy = |TP| + |TN| / N
  + Sensitivity = |TP| / |TP| + |FN|  known as true positive rate, the number of correct positive examples.
  + Specificity = |TN| / |TN| + |FP|  known as true negative rate, the number of correct negative examples.
  + Precision = |TP| / |TP| + |FP|
  + Recall = |TP| / |TP| + |FN|
  + F1-measure 🡪 combine precision and recall: F1 = 2 \* ((precision x recall) / (precision + recall))
* Scaling 🡪 involves changing the range of the feature values in order to bring them to a similar scale within a specific range. For example, by using min-max scaling, where the feature values are transformed to a specified range (often [0,1]), or standardization / z-score normalization, where features are rescaled to have a mean of 0 and a standard deviation of 1. We want to use scaling if our feature values have different units or ranges, as this can lead to some features dominating others. Scaling the data will in that case improve the performance of the machine learning algorithm. Scaling can also help convergence speed and precision of optimization algorithms.

**Optimalization and search**

* Search landscape 🡪 The set of solutions we are searching through illustrated as a landscape.
* Optimalization problem 🡪 To find the best solution to a given problem.
* Optimization search 🡪 The method we use to solve the optimization problem. We search through the search landscape. A goal in optimization is to test as few solutions as possible, finding the solution in an efficient manner.
* We need a numerical representation of solutions. We represent a possible solution to a problem with the variable x. The solution can also be represented by an array, for example, the array [w, h] – containing the most optimal width and height of a flight wing.
  + A function f(x) tells us how good the solution (x) is. It can be a mathematical algorithm that helps us estimate the quality of a solution.
* Maximalization and minimalization problems are similar. We use the same algorithms. We want to find a solution x that maximizes or minimizes f(x).
* Continuous optimization
  + Optimizes something on a continuous spectrum. There is a function value for every possible combination of input.
  + Can be used to solve maximalization and minimalization problems.
  + An example of continuous optimization is optimizing the parameters of a machine learning algorithm.
  + Continuous optimization is also used to optimize the design of mechanical parts/shapes, in economy, process engineering and robotics.
  + Methods:
    - Greedy Search
    - Hill Climbing
    - Simulated Annealing
    - Gradient ascent/descent
* Discrete Optimization
  + Problems where we don´t have a continuum of solutions, we have a finite set of solutions that we can choose from, and we can´t necessarily interpolate with them.
  + We search through a set of solutions in order to find a solution with specific properties.
  + To interpolate with a solution means using an existing solution on a problem to estimate or try to calculate the function value for other input values. It´s common within numerical analysis and mathematics and is used to estimate the solution to a problem.
  + Can also be used to solve maximalization and minimalization problems, but it´s (often) not possible to find an intermediate solution.
  + We can translate a continuous problem to a discrete problem by dividing our data into categories/buckets, making them discrete.
  + Examples of discrete optimization problems:
    - To make a time schedule for courses at the university with as few collisions as possible.
    - Travelling Salesman Problem (TSP)
  + Methods:
    - Exhaustive Search
    - Greedy Search
    - Hill Climbing
    - Simulated Annealing
* Exhaustive Search 🡪 Brute-force search
  + Test all possible solutions and choose the best one.
  + The only one that guarantees an optimal solution.
  + Only works for extremely simple discrete problems, has time complexity n! 🡪 which is worse than exponential time
* Greedy Search
  + Generates and evaluates only one solution. Finds the first and best solution by continuously choosing the next most optimal choice. However, the overall result is not always the most optimal solution.
* Hill Climbing
  + Issues with this algorithm:
    - If we start near an optimal local solution, we will quickly reach a peak and not find any better solutions.
    - If there are lots of foothills near the global optimum, the algorithm will climb to a local optimum and get stuck there.
    - If there is a plateau, the changes the algorithm makes might not have any affect and it might get stuck and terminate because of the lack of increase in fitness
    - If there is a gently sloping ridge in the data, the changes that the algorithm makes might have such a small effect that it decides that it has reached the global optimum and terminate.
* Simulated Annealing 🡪 Annealing is a process within physics that starts after a solid material is gradually heated to its melting point. Annealing is when the material, that is now a liquid, gradually cools down. The particles organize themselves into a minimum energy state, which leads to the material obtaining the form of the container in which it was melted. As the material cools down, there is less random movement in the particles. The analogy to the algorithm is the temperature T, which – when high (as it is initially) gives more exploration, and then when low (decreases with each iteration) gives more exploitation.
* Gradient ascent/descent 🡪 the underlying mathematical framework for training of deep neural networks and therefore the most important algorithm within the field of machine learning today. A gradient is a vector that contains the partial derivatives of all the dimensions we have and describes the direction and length of our solution. For continuous problems gradient descent is an efficient way of finding the maximum/minimum value, but has the same issues as hill climbing.
* Exploitation 🡪 Taking advantage of an existing solution and finding ways to make it better.
* Exploration 🡪 Exploring completely new solutions, like with an exhaustive search.
* Balancing exploitation and exploration give us the best solution.
  + With the mentioned algorithms, for example, we can try hill climbing from an arbitrary starting point until we no longer find any better solutions, and then retry from a different starting point and compare the results. This combines exploitation (hill climbing) and exploration (trying different starting points).
  + We can also choose "worse" edges with an underlying probability that they will lead to a better overall solution in the long run.

**Evolutionary learning**

Inspired by biology, the algorithm creates a population of candidate solutions which correspond to potential solutions to a problem. It evaluates the fitness of each solution based on how well it performs in solving the problem. It is often used for optimization problems where the search space is large or complex, and a mathematical or analytical solution is difficult to obtain.

* To krefter innenfor evolusjonære algoritmer som sammen gir oss den beste globale løsningen
  + Å øke populasjonens mangfold gjennom mutasjon og rekombinasjon
    - Denne kraften fører til mer utforskning og nye løsninger
  + Å minke/begrense populasjonens mangfold gjennom seleksjon av foreldre og overlevende/videreførte løsninger.
    - Denne kraften fører til en bedre kvalitet
* Initialisering 🡪 vanligvis initialiserer vi søket ved å velge en mengde med tilfeldige individer. Vi må sørge for en god spredning og en variasjon av mulige alleler, og derfor en tilfeldighet et viktig element i initialiseringen. Vi kan også av og til inkludere eksisterende løsninger, eller benytte oss av problemspesifikk heuristikk, for å så populasjonen. Da utnytter vi kunnskap som vi eventuelt har ervervet tidligere om hva som kan være en god løsning på det aktuelle problemet for å effektivisere søket.
* Populasjon 🡪 Alle aktuelle løsninger som vi vurderer. Hver gang vi oppdaterer populasjonen (for hver iterasjon), danner vi en ny generasjon.
  + Vi jobber med en populasjon av løsninger, og vi prøver å holde denne populasjonen godt distribuert og mangfoldig, slik at vi kan finne den beste globale løsningen. Hver løsning kalles et individ.
  + Når vi initialiserer en populasjon, må vi evaluere hvert individ og gi det en form for skår som kan fortelle oss noe om hvor god den er. Denne delen av algoritmen er vanligvis den mest tidkrevende (90-99 % av prosesseringstiden). Vi må vanligvis gjøre ganske komplekse beregninger. Evaluering av individer 🡪 Å måle kvaliteten på løsningen vår, og gi den en skår
    - Evaluering gjør det mulig å selektere og sammenligne individer
    - Går ut på å tildele en verdi, kalt «fitness», til hvert individ.
* Seleksjon 🡪 Gjennom seleksjon ønsker vi å identifisere hvilke egenskaper som er nyttige å ha med videre i populasjonen vår, og velge disse. Dette gjør vi basert på fitness-verdi. Seleksjon opererer på populasjonsnivå. Vi må se på alle, eller mange av, individene og selektere noen av dem til å være med videre til neste generasjon. Vi selekterer på to tidspunkt i algoritmen:
  + Først selekterer vi foreldrene basert på vår initielle populasjon. Vi identifiserer foreldrene. Denne seleksjonen er vanligvis probabilistisk. Det er høyere sannsynlighet for å selektere individer av høyere kvalitet (høyere fitness-verdi), men ikke garantert. Vi ønsker også å velge noen suboptimale individer som på sikt kan føre oss til det beste individet. Dette for å unngå å sitte fast i et lokalt optimum.
  + Senere selekterer vi de individene vi ønsker å videreføre ved å sammenligne de nye individene vi har generert med individene i vår initielle populasjon. Vi identifiserer de individene vi ønsker å videreføre til neste generasjon.
  + Seleksjonsoperasjoner 🡪 Er uavhengige av representasjon fordi de baserer seg på fitness-verdi.
  + Seleksjonspress 🡪 Hvor mye press det er i algoritmen om å være det sterkeste individet. Ved lavt press kan vi generere mer utforskning, mens ved høyt press ønsker vi å utnytte i større grad. Vi starter derfor med et lavt press for å utforske søkslandskapet i starten, og øker etter hvert for å utnytte i større og større grad.
  + Fitness-proporsjonal seleksjon 🡪 Roulette-hjul-seleksjon gir en høyere sjanse for å velge individer med bedre fitness-verdi. Sannsynligheten for at individet blir selektert blir da proporsjonal med fitness-verdien.
    - Sannsynligheten for at et individ i blir selektert P(i) = fitness-verdien til i delt på summen av alle fitness-verdier.
    - Prematur konvergens 🡪 Ett veldig bra individ kan raskt ta over hvis resten av populasjonen har en mye lavere fitness-verdi. Dette individet ender opp med å bli selektert igjen og igjen, og de andre individene vil gjennom flere generasjoner ende opp med å konvergere mot denne ene løsningen. Vi har da alt for høyt seleksjonspress tidlig i søket.
    - Tap av seleksjonspress 🡪 Mot slutten av søket kan individene våre bli så like at vi ender opp med et lavt seleksjonspress.
  + Rang-seleksjon 🡪 Baserer sannsynligheten for seleksjon på relativ fitness-verdi snarere enn absolutt fitness-verdi. Vi rangerer populasjonen og sorterer individene deretter.
  + Tournament selection🡪 De ovennevnte metodene krever at vi kalkulerer en fitness-verdi for hvert individ, og sammenligner dem deretter. Noen ganger ønsker vi ikke å gjøre dette, for eksempel ved parallell-programmering, hvor noen individer får en fitness-verdi mens andre blir selektert. Noen ganger er det heller ikke hensiktsmessig å sammenligne absolutt alle individer. Tournament selection bruker lokale fitness-verdier. Vi velger k antall tilfeldige individer, og velger deretter den beste av disse. Dette repeteres for å velge flere. Sannsynligheten for å selektere et individ vil være avhengig av rangen til individet, og størrelsen på utvalget k. Et større utvalg øker seleksjonspresset. Vi kan videre velge om vi ønsker å tillate at et individ kan sammenlignes med seg selv, og hvorvidt vi alltid vil velge det beste individet, eller om dette skal foregå basert på en sannsynlighet p.
  + My (antall gamle løsninger) og Lambda (antall barn).
  + Elitisme 🡪 Beholde de n beste individene.
  + (My, Lambda)-seleksjon 🡪 Basert på My genererer vi Lambda, og så velger vi nye individer ut fra Lambda. Denne metoden kan føre til at vi mister de beste individene, men er bedre på å komme seg vekk fra lokale optima.
  + (My + Lambda)-seleksjon 🡪 Basert på My genererer vi Lambda, men så velger vi nye individer fra My og fra Lambda (som ved elitisme, hvor vi vil beholde de n beste individene).
* Mangfold 🡪 Vi ønsker å preservere mangfoldet fordi vi ikke vet hvor i søkslandskapet den beste løsningen ligger, og fordi søkslandskapet eller miljøet i den virkelige verden kan endre seg / være litt annerledes enn det vi så for oss.
  + Implisitt 🡪 Endrer strukturen til algoritmen ved å forsøke å beholde flere subpopulasjoner, med det målet om at de subpopulasjonene vil løse problemene våre.
    - Automatic speciation 🡪 Kun tillate kombinasjon mellom to like individer, eller gruppere individene inn i subpopulasjoner ved å gi dem en tag eller putte dem i en bøtte.
    - Geografical separation 🡪 Kjøre algoritmen fra flere ulike «øyer», for så å periodevis migrere individer fra én øy til en annen.
  + Eksplisitt 🡪 Går inn og endrer sannsynligheten for å velge ulike individer, slik at vi får en høyere sannsynlighet for å velge individer som ikke ligner på de vi allerede har sett. Det motiverer til utforskning.
    - Fitness-deling 🡪 Begrenser antall individer med en gitt egenskap / en gitt måte å løse problemet ved å dele deres fitness-verdi. Like individer må dele én fitness-verdi, mens nye individer får beholde sin. Vi må sette en Sigma-share, som sier noe om hvor stor del av genotypen eller fenotypen som kan være lik før de må dele fitness-verdi.
    - Crowding 🡪 Nye individer erstatter like individer. De sammenlignes med sin nærmeste forelder og vi måler avstanden mellom dem. Dette fører til at individene spres rundt i søkslandskapet.
* Variasjon opererer på individnivå 🡪 Målet er å generere nye individer. Når vi endrer et individ (en løsning), endrer vi vanligvis på et lite utvalg av individene, ikke hele populasjonen. Variasjon kan deles inn i flere typer, basert på ariteten til funksjonene:
  + Aritet 1 🡪 Mutasjon
  + Aritet > 1 🡪 Rekombinasjon
  + Aritet = 2 🡪 Krysning
  + Aritet > 2 🡪 Teoretisk mulig, men ikke ofte brukt innenfor evolusjonære algoritmer
* Mutasjon 🡪 Målet med mutasjon er å skape en liten, tilfeldig endring i en genotype. Tilfeldighet er essensielt, og differensierer mutasjon fra andre unære heuristiske operasjoner. Dette for å motivere til utforskning. Mutasjon er likevel en utnyttelsesstrategi, fordi vi kun varierer en liten del av forelderen og dermed kun tar et lite hopp til en side i søkslandskapet.
* Rekombinasjon 🡪 Målet med rekombinasjon er å skape et bedre avkom ved å kombinere elementer fra de to (eller flere) foreldrene. Valget om hvordan vi rekombinerer gjøres stokastisk, det vil si at vi ikke forsøker å bruke erfaringer eller kunnskap i prosessen, men velger tilfeldig. Dette for å motivere til utforskning. Vi kan rekombinere på flere måter, ett eksempel er å velge starten av den ene genotypen og slutten av den andre og kombinere disse til et nytt individ. Rekombinasjon/krysning tar i motsetning til mutasjon et stort hopp i søkslandskapet, og utforsker derfor i større grad helt nye løsninger.
* Termineringsbetingelser 🡪 Betingelsene for terminering av en evolusjonsalgoritme kan være følgende:
  + Når vi har nådd et kjent/ønsket resultat (en viss fitness-verdi)
  + Når vi har nådd et maksimalt antall mulige/tillatte generasjoner
  + Når vi har nådd et minimumsnivå hva gjelder mangfold
  + Når vi har nådd et spesifisert antall generasjoner uten forbedringer av fitness-verdien
* Representasjon 🡪 Representasjonen bestemmer hvilke variasjonsoperasjoner vi kan utføre, altså hvordan vi gjør mutasjon og rekombinasjon. Når vi velger/definerer en representasjon (enkoding), må vi også velge hvordan den skal dekodes.
* Memetic algorithms 🡪 Algoritmer som kombinerer evolusjonære algoritmer med operasjoner fra andre algoritmer, eller inkorporering av problemspesifikk kunnskap. Dette kan gjøres i ulike stadier av en evolusjonær algoritme. Memetiske algoritmer er betydelig raskere og mer nøyaktige enn evolusjonære algoritmer på noen problemer, og blir betraktet som «state of the art» med hensyn til mange problemer. Do a local search, for example a greedy ascent or steepest ascent for each iteration of the EA loop.
  + Hybridize in the initialization of the EA algorithm by including known solutions, constructive heuristics (domain-specific knowledge about the problem), selecting good individuals and doing a local search
  + Hybridize in the mating pool by using problem specific information in the crossover operator 🡪 for example using tailored operators from other algorithms which have been used on the same problem
  + Hybridize in the offspring by doing a local search and using problem specific information in the mutation operator
  + Hybridize in the replacement process by modifying the selection operator
* **Multi-objective problems** 🡪 Problems that can be categorized by the presence of a number of n possibly conflicting objectives. For example: energy efficiency vs. completeness (how thorough something is). In these cases, we want to identify a set of good solutions that will balance the objectives we have on hand, because we usually can´t have both be the best. With these problems we want to identify several peaks, and this means we want to preserve diversity, instead of converging to one peak. We can preserve diversity with implicit or explicit approaches. Two main approaches can be used to find the good solutions:
  + Weighted sum (scalarization) 🡪 transforms the different objectives into a single scalar value, making it a single objective problem. We can then optimize using traditional techniques. For example we can assign a weight to each objective, and then compute the weighted sum of the different objectives – which then can be used to represent the multiple objectives.
  + Pareto front 🡪 a set of solutions where no solution can be improved in one objective without sacrificing the performance in another objective. The solution is considered “pareto optimal” if it is not dominated by any other solution in the objective space. So the Pareto set or Pareto front contains all non-dominated solutions from the entire feasible solution space. The members of this set are Pareto-optimal solutions.
    - Dominance relation: A solution x dominates solution y if x is better than y in at least one objective AND x is not worse than y in all other objectives.
    - Pareto optimiality: A solution x is non-dominated among a set of solutions in Q if no solution from Q dominates x.
* Multi-objective optimizers 🡪 The goal is to find a set of non-dominated solutions (approximation set) following the criteria of convergence (as close as possible to the Pareto-optimal front) and diversity (spread out and well-distributed). We can use evolutionary algorithms to solve multi-objective problems:
  + Selection 🡪 We usually select individuals based on dominance, with ranking or depth-based selection that favors solutions that are non-dominated. The most non-dominated solutions get a higher ranking than the others. One such selection method is Non-dominated Sorting Genetic Algorithm (NSGA) or Fast Non-dominated Sorting Genetic Algorithm (NSGA-II)
  + Diversity 🡪 We preserve a diverse set of points the same way as we would to for multi-modal problems, which are problems that have multiple good solutions (peaks) in the solution landscape 🡪 see implicit and explicit approaches under the section “diversity”. We call these techniques “niching”.
  + Elitism 🡪 We want to preserve all the non-dominated points we have seen, usually by just using elitism or keeping an archive of all non-dominated solutions and using this archive as for example a second population in recombination etc.

**Supervised learning**

The algorithm is trained on a set of examples (training set) containing both input features and corresponding output labels. Based on this training set, the algorithm tries to identify patterns and relationships in the data set, and then generalize the patterns to make predictions on new, unseen data. The goal is to accurately predict the correct labels for new input data. There are two types of supervised learning: classification and regression.

* **Input** 🡪 is typically a vector that consists of several real numbers (floats), which means a series of real numbers xi (from 1 to i), for example (0.2,0.45,0.75,-0.3). The size of the vector, which is the amount of elements in the vector, is called the dimensionality of the input. This is because if we were to plot the vector as a point, we would need one dimension of space for each of the different elements of the vector. Notation: lowercase boldface letters for vectors and uppercase boldface letters for matrices. The number of input dimensions if referred to as m. The vector represents the feature we have chosen.
* **Features** (also called attributes) 🡪 Input variables that are used to make predictions. They are individual measurable properties/characteristics of a phenomenon (often represented by numbers). Features can be categorical or numerical. A numerical feature is a feature that can take or be represented by a numerical value. Example 🡪 The frequency of a word in a text (which would be the feature) can be counted and thus represented numerically. A categorical feature is one that can be represented by a category from a finite set of categories, for example “yes” or “no”. Categorical features can be made to fit a classifier that expects numerical features by assigning a numerical value n for each category. Similarly, numerical features can be made to fit a classifier that expects categorical features by organizing the numerical values into categories/buckets and using these categories as features. We can combine features with weights and a bias term to capture the relationship between the input features and the target labels.
  + **1D 🡪 Each sample has a single feature x, a real number.**
  + **2D 🡪 Each sample has two features x = (x1, x2), a vector.**
  + **3D 🡪 Each sample has three features x = (x1, x2, x3), a vector.**
  + **nD 🡪 Each sample has n features x = (x1, x2, x3, …, xn), a vector.**
* Identifying informative, discriminating, and independent features is a crucial element of effective algorithms in pattern recognition, classification, and regression. Choosing how to process the data is equally important.
  + Decide on a set of features that can be observed and quantified.
  + Decide on a set of possible values for each feature.
  + Extract the values of the feature for each object.
* **Labels** 🡪 Output variables that the model is trying to predict. Labels are also called classes and are a set of categories/buckets that form the possible output of our model.
* Data collection, preparation, and feature selection:
  + If the problem is completely new, we want to start by collecting only the required data. Typically done by assembling a reasonably small data set with all of the features that we believe to be useful, and experimenting with it before choosing the best features, and collecting and analyzing the full dataset. For supervised learning, target data (labels) is also needed, which can require involvement of experts in the relevant field and significant investment of time. Finally, the quantity of data needs to be large enough to be able to use for generalization, but not so large that it causes excessive computational overhead (use of time and memory). The perfect size is generally impossible to predict.
* **Training set** 🡪 We need a set of preprepared labels, and we need to tell the algorithm what features map to those labels.
* **Test set** 🡪 There is a chance that the algorithm adapts too well to the training data (overfitting) and might therefore do well on tests that we perform on the training data, but not on unseen data. This is because we experiment with different learning algorithms and various hyperparameter settings on the training data to see what works best. We therefore need a test set that we haven’t used during training.
* **Development set** 🡪 During the development phase, when we experience with different algorithms and settings, we need to be able to test them in order to evaluate how well they work – but without using our test set. This is what we can use the development set for.
* **N-fold cross validation** 🡪 We can split the training data into n equally sized buckets, and run n amounts of experiments on them. We can use one of them as a test set, and the rest as a training set, and then take the averages of the n many experiments as an evaluation measure. The n algorithms will be slightly different because they are trained on slightly different training sets. We can use n cross-validation if we don’t have access to a large data set, because it allows us to test on a larger test set.
* **Classification** 🡪 The classification problem consists of taking input vectors and deciding which of N classes they belong to (labeling), based on training from exemplars of each class. The classification problem is discrete 🡪 each example belongs to precisely one class, and the set of classes covers the whole possible output space. The methods of performing classification aim to find decision boundaries that can be used to separate out different classes. Given the features that are used as inputs to the classifier, we need to identify some values of those features that will enable us to decide which class the current input is in (discriminating them).
  + Decision boundary 🡪 The point, line or plane that separates the classes/labels from each other. The decision boundary depends on the type of problem.
* **Regression** 🡪 The regression problem in statistics consists of fitting a mathematical function describing a curve, so that the curve passes as close as possible to all the datapoints. It is a problem of function approximation or interpolation (estimating/predicting values), working out the value between values that we know. In machine learning it is a problem where the goal is to predict a continuous numerical value as the output (linear regression model) or classify (logistic regression models). The training data for a regression problem consists of input feature vectors and their corresponding target values. The algorithm learns from this data to create a regression model or function that can take new input features and make predictions on the expected output. The goal is to find the best-fitting regression model that minimizes the difference between the predicted values and the actual target values. We do that by choosing an appropriate type of regression algorithm, such as linear or logistic regression, and then during training we adjust the set of parameters (usually means updating the weights, number of epochs and learning rate) that help determine the shape and behavior of the function. The main objective in the training phase is to find the optimal values for the predefined function. This is often done by minimizing a cost function, such as mean squared error (MSE), which measures the average squared difference between the current function’s predictions and the actual values (labels). We can use gradient descent to iteratively update the parameters in order to minimize this cost function.
  + **Linear regression** 🡪 We should use a linear regression model when the goal is to predict a continuous or numerical outcome based on the input features. It aims to establish a linear relationship between the features and the target variables, making it suitable for tasks like forecasting, modeling trends and estimating numerical quantities. Therefore, the relationship between the input features and the target variable should ideally be linear. We train the linear regression model by minimizing the cost function MSE through gradient descent.
    - We don’t typically use this as a classifier, but we could by fitting the function to all the datapoints at the same time, creating a straight line, and then transform the numerical output (which is somewhere between -1 and 1) into a prediction of classes by using the Heaviside step function. If the output is > 0, then we predict the positive class. If it is < 0, we predict the negative class. The decision boundary is a line that separates the classes.
  + **Binary logistic regression** 🡪 Binary logistic regression models the probability of the feature belonging to one of two labels by using the logistic function (also known as the sigmoid function. The model can be used for problems where we can assume linearity between the input features and the logarithm of the odds ratio (log-odds/logit). The logistic function takes a real value and maps it to a value between 0 and 1, which represents the probability of the positive class. We can use logistic regression as a binary classifier. A logistic regression model predicts the probability of a binary outcome based on input features. We train the model by finding the best-fitting weights and bias term using the technique of minimizing a loss function with gradient descent. The weights then represent the importance or contribution of each feature in the prediction. By multiplying the feature values with their respective weights, the model assigns different level of significance to individual features when making the predictions. By iteratively updating the weights during the training phase, we essentially adjust the parameters that help us find the values that influence the target variable. The bias term, also known as the intercept, allows the function to capture the baseline value or starting point of the predictions, irrespective of the feature values. The bias term is estimated or learned during the training phase along with the weights of the input features. In logistic regression, the bias term determines the location of the decision boundary. If the bias term is positive, it shifts the decision boundary to the right, while a negative bias term shifts it to the left. The bias term accounts for the scenarios where the expected target variable may not be zero even when all the features are zero. Domain-specific knowledge about the starting point, custom initialization or regularization techniques can be used to guide the estimation of the bias term. In logistic regression the weights and bias term are learned simultaneously, and their effects are therefore inherently intertwined.
  + **Multinomial logistic regression** 🡪 Uses the same principles as a binary logistic regression model with the addition of multiple sets of weights and bias terms associated with each class. It calculates the probabilities of the instances belonging to each class using the softmax function, which ensures that the probabilities sum up to 1 across the classes.
* **The perceptron** 🡪 The perceptron is a simple binary classification algorithm based on the concept of a simplified model of a biological neuron. If the classes are linearly separable, the perceptron will converge and find a decision boundary that separates the data points. The algorithm adjusts the weights and bias term based on misclassifications until all data points are correctly classified or it reaches a predefined number of iterations. It's important to note that the perceptron is an algorithm for binary classification tasks and works best when the classes are linearly separable. It does not accommodate complex patterns or handle nonlinear problems well. To overcome this limitation, more advanced neural network architectures like multi-layer perceptrons (MLPs) or deep learning models can be used.
* **Nearest Neighbor Methods** 🡪 Simple machine learning algorithms used for classification and regression tasks. They work based on the principle that data points that are close to each other are likely to have similar properties. We look at similar data and choose to be in the same class as them.
* **Decision trees** 🡪 Trees are efficient data structures that can help us make the algorithms faster. DT-algorithms are used for both classification and regression tasks. They model decisions or predictions based on a tree-like flowchart structure, where each internal node represents a decision based on a feature, and each leaf node represents a final prediction or outcome.
  + Training 🡪 they take a labeled training data set consisting of input features and their corresponding class labels.
  + Feature selection 🡪 evaluate different features and select the one that best separates or classifies the data points based on certain criteria.
  + Splitting 🡪 the selected feature is used to split the data into separate branches, where each branch represents a different value or range of the chosen feature.
  + Recursive splitting 🡪 the process of feature selection and splitting is repeated recursively for each branch, creating a tree-like structure. At each step, the algorithm aims to split the data in a way that maximizes the separation or purity of the resulting subsets.
  + Leaf nodes and prediction 🡪 The splitting process continues until a stopping criterion is met, such as reaching a maximum depth, minimum number of samples or homogeneous subsets. Each leaf node represents a predicted class label for a new, unseen data point.
* **Random forest** 🡪 Combining several decision trees in order to make a better classifier. We can achieve this by training the trees on slightly different data, taking bootstrap samples from the dataset for each tree. A bootstrap sample is a subset of the training set which contains n items with replacement. We can achieve a larger variation between the trees by restricting the construction at each node to only consider a subset of all the features. After training, we will have a collection of many decision trees that make independent and somewhat different decisions. We classify by taking the majority vote between the trees.
* **Feed-Forward Neural Network (Multi-layer Perceptron)** 🡪 A feedforward network is a multilayer network in which the units are connected with no cycles; the outputs from the units in each layer are passed to units in the next higher layer, and no outputs are passed back to lower layers. The input layer x is a vector of simple scalar values. The hidden layer h is formed of multiple neural units taking a weighted sum of its inputs and then applying a non-linear activation function. A standard feed-forward neural network is fully-connected, which means that each unit takes as inputs the outputs from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers. For each hidden layer, we combine the input units with a weight vector and add a bias term. Then we apply an activation function, the sigmoid or the ReLu to this result. If we use ReLu, the result is a vector of real-valued numbers, and we can’t use it as the output of the final layer (which will be the output of the classifier). We have to return a vector of probabilities. To do that, we need to normalize the result. We do that by using the softmax function for the final layer. Another way to look at this, is that a neural network is like multinomial logistic regression, but (a) with many layers, since it is basically layers of logistic regression classifiers, and (b) with those intermediate (hidden) layers having many possible activation functions instead of just the sigmoid, and (c) rather than forming the features by feature templates, the prior layers of the network induce the feature representations themselves.
* **Things to consider when training a neural network model:**
  + Scaling the data is often helpful
  + The weights 🡪 should not be initialized to 0, but random numbers between -1 and 1. We could also multiply with 1/sqrt(m).
  + Local minima 🡪 The loss function of a multi-layered neural network model is not convex, so it can be caught in local minima. We should therefore make several runs with different initializations and compare the results by calculating the mean and standard deviation.
  + Early stopping 🡪 If we train for too long, there is a chance of overfitting, so we should use a validation set V and stop the training if the loss starts to increase.
  + Variations of gradient descent 🡪 Stochastic gradient descent and Mini-bacth training can be a way to avoid local minima
    - Batch training 🡪 Given a set of weights, one predicts the output values for all items in the training set, and then compues the loss and the gradient for the whole set and upgrade the weights accordingly. This process is repeated a number of times.
    - Stochastic gradient descent 🡪 One calculates the loss and the gradient for one randomly chosen item in the training set for each epoch, and repeats. This is different to normal gradient descent because the normal version calculates the gradients for the entire training data set for each epoch. SGD is therefore more efficient. Since we choose the item randomly, SGD also adds more exploration and helps us avoid getting stuck in local minima. This can in turn lead to better generalization on unseen data and reduce overfitting. This actually also adds a form of regularization.
    - Mini-batch training 🡪 An in-between of the two mentioned above. We first decide on a batch size, and then the whole data set is split into mini-batches of that size. In one epoch, we process all the mini-batches 🡪 calculate the output, loss and gradient and update the weights accordingly. We can also shuffle the data points between each epoch.
* **Recurrent neural networks** 🡪 A neural network which contains a cycle within its network connections. The cycle could be a reflexive relation or a cycle including several nodes. The RNN is processed at time steps, where the values of a node x at time t+1 is calculated from the values of all the nodes at time t and the direct connections into node x. This means that the state of a node x at time t+1 will be influenced by the state of nodes at time t (an all earlier stages) – which gives the network a form of memory. The RNN is particularly suited for sequence problems, for example sentences in natural language, where we want a model that can identify a word at a time, but also remember the preceding words in the sentence.
* **Convolutional neural networks 🡪** Originally used for image analysis, is a neural network that applies filters, known as convolutional kernels, to the image. These filters scan the image through a process called convolution 🡪 extracting different features such as edges, corners and textures. Each filter produces a feature map that highlights the areas of the input image where the feature is present. The output of the convolutional layer is then passed through an activation function, which introduces non-linearity into the network. It allows the CNN to capture complex relationships between the features. Following the convolutional layers, the CNN typically includes pooling layers – which reduce the spatial dimensions of the feature maps while retaining the most relevant information. Finally, the pooled feature maps are flattened and fed into fully connected layers, which act as a traditional neural network and perform classification or regression based on the learned features.

**Reinforcement learning**

The algorithm improves by exploring its environment and trying to find the right answer through trial and error. It receives feedback in the form of rewards or punishment (for example gets told the output is wrong), but no guidance as how to improve or correct the output. This type of machine learning is used in game playing, robotics, and autonomous systems, where the algorithm needs to learn to make sequential decisions in a dynamic environment.

* Policy 🡪 Are the rules that determine the mapping between the observations we make in the state we are in and the actions. Policies can be deterministic or stochastic, yielding more or less exploitation and exploration. By allowing the policy to change over time, the system can adapt to the environment 🡪 which is what we call learning. We can introduce randomness at each stage in the reinforcement learning algorithm cycle in order to make the policy stochastic. Exploration is essential in reinforcement learning, because we need to explore different actions to see where they lead in order to accumulate knowledge. As the training progresses, we can exploit more. The policy is commonly represented by the weights in a neural network or as a table.
  + Neural network implementation of policy 🡪 w is the network weights given by the parameter vector w.
  + Q-table 🡪 a table (numpy array) that stores all previous learning experience with one entry for each (state, action) combination. The entry contains the average reward obtained after taking the specified action a when in the state s. There are three different ways of using the Q-table to represent the policy:
    - Greedy 🡪 action = argmax Q[s,a]. This means 🡪 choose the action aj with the highest average reward in state si.
    - Epsilon-greedy 🡪 action = argmax Q[s,a] with probability 1 – epsilon OR a random action with probability epsilon. This means 🡪 draw a random number x in an interval. If x is larger or the same as epsilon, then choose the action aj with the highest average reward in state si. Else: select an action at random.
    - Softmax 🡪 Draw as many random numbers as we have actions, and use the softmax function.
* States 🡪 The environment in which we are working to solve a problem has, at any given time, a particular state. Based on our policy, we choose an action, and this action affects the state we are in. The result of the action can change the state of our environment.
* Typical problem in RL 🡪 Get from a start state to a destination state in the least number of actions/movements (shortest path problem), given the following constraints:
  + Not all state transitions are allowed
  + Some actions/movements are bad and some are good
* Rewards 🡪 instead of giving the model the correct output (by using a training set that costs resources), we tell the model if the result of the action is good or bad by giving positive rewards and negative rewards (penalties) (can also be neutral).
* Agent 🡪 An agent repeatedly adjusts the policy based on the previous state, the current state and the reward received.
* Learning 🡪 the learning part in the reinforcement algorithm cycle is the updating of the weights. How we update the weights depends on the implementation of our policy:
  + Policy-gradient methods (neural network implementation) 🡪 The neural network learns to optimize the policy by maximizing the expected cumulative reward. Follow these steps:
    - Execute the current policy
    - Collect the rewards
    - After several iterations, adjust the weights in the direction of increased expected reward (gradient ascent).
  + Actor-critic methods (neural network implementation) 🡪 We use two neural networks that interact and are trained together, the critic and the actor.
    - Using the actions generated by the actor network, the critic network learns the values of different (state, action) pairs) from the received rewards, which are the Q-values 🡪 average reward for different actions given a state. It then updates the actor.
    - The actor network 🡪 learns the correct actions using feedback (updates) from the critic network.
  + Q-learning and SARSA (Q-table implementation) 🡪 We use the information we have in the Q-table about the rewards for each action in each state to tell us which action we should take. During training, the Q-table is updated using the update rule. The update rule ensures that we let recent rewards have a higher impact than older rewards, and that the Q-table-values reflect the total accumulated reward from state s and all the way to the final destination. For this we use a learning rate and take the weighted sum of all the preceding rewards. We also want rewards that will be received later on the path to the destination to be reduced (discounted), because the environment can change, and we don’t want to put too much weight into something that may or may not happen in the long distant future. For this we can use a discount rate. This update rule is called the SARSA learning rule, but Q-learning uses an alternative method where it always assumes that we will choose the action with the highest reward in each state. This is called off-policy learning.

**Unsupervized learning**

The algorithm explores data to identify inherent patterns, relationships, or clusters in a data set without being trained beforehand. Unlike SL, there are no predefined labels or outputs in the unsupervised learning. The algorithm identifies similarities between the inputs, so that inputs that have something in common are grouped/categorized together. The statistical approach to unsupervised learning is known as density estimation. We use unsupervised learning for the following problems:

* Generate similar data 🡪 density estimation, generative models.
  + Generative models learn a data generation mechanism by using a training set, and then apply the data generation mechanism to generate new data. The model can be evaluated by comparing the distribution of the generated data with the distribution of the training data.
    - Discrete random variables 🡪 A probability distribution for a discrete random variable describes the probability of different values occurring.
    - Continuous random variables 🡪 The probability distribution is calculated by using a density function f(x) and define P(X in [a,b]) (the probability in an interval).
      * Histogram 🡪 A graphical representation of the distribution of a dataset. It consists of bars where the width represents the intervals where x can occur and the height represents the frequency for x being in that interval. It is used to understand the shape, central tendency and spread of the data.
      * Smooth density estimator 🡪 A statistical technique used to estimate the underlying probability density function of a continuous random variable from a finite set of observations. Provides a continuous estimation.
      * Generative Adversarial Networks (GANs) 🡪 Two neural networks that compete against each other in a game-like setting. The generator and the discriminator. The generator improves its ability to create realistic data to fool the discriminator, while the discriminator improves its ability to differentiate between real and generated data. This adversarial process continues until the generator produces data that is indistinguishable from real data, and the discriminator is no longer able to differentiate between the two. Used for generating images, for example.
        + Generator 🡪 Takes random noise as input and tries to generate data that is similar to the real data it was trained on.
        + Discriminator 🡪 Tries to distinguish between real data and the data generated by the generator
* **Clustering** 🡪 Learning how to group similar data points together based on inherent characteristics or similarities. The goal is to discover patterns and structures in the data without having any (or a lot of) prior knowledge or labels.
  + **Hierarchical Clustering 🡪** Starts by defining each data point as one cluster, and then repeatedly merging clusters that are close together to form larger clusters until all points are in a single cluster, or until a stopping criterion is met. This stopping criterion can be when we reach a certain amount of clusters. The result is a dendrogram, which shows the arrangement of the clusters and their relationships at different levels of similarity. Based on the levels of dissimilarity observed in the dendrogram, we can determine the optimal number of clusters.
    - **Agglomerative Hierarchical Clustering** 🡪 Starts with each data point as an individual cluster and recursively merges the closest pairs of clusters until all the data is in a single cluster.
    - **Divisive Hierarchical Clustering 🡪** Starts will all the data points in one single cluster and then splits the clusters recursively until each data point is in its own cluster.
    - **The dendrogram** 🡪
      * The vertical lines represent the individual data points or clusters.
      * The height at which two clusters are joined together or split apart represents their dissimilarity or distance at which the merge or split occurred.
      * The arrangement of branches and the clustering order provide insights into the relationships and similarities between the data points or clusters.
    - **Important aspects** 🡪
      * We have to keep track of the whole history of cluster merges in order to plot the tree at the end.
      * We have to compute the distance between each pair of the remaining clusters for each iteration.
      * We have to define what we mean by the distance between two clusters.
    - **Linkage methods** 🡪The choice of linkage method is important as different linkage methods can yield different clustering results. Before linking two clusters, we calculate the distance between each pair of clusters, and then decide which clusters should be merged based on our linkage method.
      * Single linkage 🡪 distance(C1, C2) is the smallest distance we can get between a sample in C1 and a sample in C2. Suitable for a sparse dataset and can handle irregularly shaped clusters and is robust to outliers. Effectively identifies sparse and separated clusters. Low computational complexity.
      * Complete linkage 🡪 distance(C1, C2) is choosing the smallest largest distance we can get between a sample in C1 and a sample in C2. Suitable for problems where we want all the data points to be close to each other. Will capture well-separated clusters, and not assign a lot of weight to outliers.
      * Average linkage 🡪 distance(C1, C2) is the smallest average distance between all the samples in C1 and C2. Suitable for a dense dataset, and to identify clusters where we have outliers. Low computational complexity.
      * Ward linkage 🡪 distance(C1, C2) is the smallest increase in within-cluster variance if the clusters are merged. Suitable for compact and spherical clusters. High computational complexity. It minimizes the sum of squared differences within each cluster. Ward's method aims to create clusters with small variances and is suitable for minimizing the overall within-cluster variance.
  + **K-Means Clustering** 🡪 Is an iterative method where we select k number of clusters, and then group the datapoints in the cluster center that is closest to them. It is suitable for datasets with well-defined clusters and for low data dimensions. Some issues with k-means:
    - It can take long to converge if the randomly initialized cluster centers are bad.
    - We need some knowledge about the problem in order to estimate the number k.
    - It is sensitive to outliers because we calculate the average between datapoints. We could avoid this by using the median instead.
    - The algorithm assumes that the clusters are spherical and equally sized, making it less effective in clustering non-linear or irregularly shaped data.
    - Useful for:
      * Identifying groups of customers based on purchasing behavior or demographic information.
      * Dividing an image into different regions or objects based on color or texture similarity.
      * Grouping documents or texts based in their content or semantic similarity.
      * Identifying outliers or abnormal datapoints in a dataset.
* Data compression 🡪 For some problems, we want to compress the data. Data compression gives us an encoded version of the data that is more compact, but that is also possible to decode back to the original format. We don’t use data compression for human interpretation purposes.
  + Autoencoders 🡪 Neural networks that learn to produce an output similar to the input and are used for data compression problems. If the hidden layer is smaller than the input/output layers, the network learns a compact representation of the data. There are several types of autoencoders:
    - Sparse autoencoders 🡪 Trained to obtain a sparse hidden layer representation. This means that most of the neurons have low activation values (close to zero), while only a few others are significantly activated (high values). This can in turn promote the emergence of meaningful and selective features in the data.
    - Denoising autoencoders 🡪 Trained to recapture a denoised version of the input vector. During training, the autoencoder corrupts the input data with noise (adding Gaussian noise or randomly setting some input values to zero), and then it learns to reconstruct the original data from the corrupted input. This encourages the autoencoder to capture underlying structures of the data and filter out the noise from the input.
    - Contractive autoencoders 🡪 Trained to obtain a hidden layer representation that is robust to small changes in the input.
    - Variational autoencoders 🡪 Trained to obtain a representation useful also for generating new data from the same distribution.
* Dimension reduction 🡪 For some data, we want to reduce the dimensions. This is because we may wish to plot high-dimensional data (humans can’t perceive more than three dimensions), visually explore the structural features of a data set, reduce the number of features before further analysis or to reduce noise. When we reduce the dimensions of the data, the encoded version we get should preserve only the important structures and be useful for human interpretation or as input for other analysis (this is because of two things: 1) the curse of dimensionality – the higher the dimensions, the more training data we need, and 2) dimensionality is an explicit factor for the computational cost of many algorithms). Decoding is not an aim. The following are methods used for dimension reduction:
  + Feature selection 🡪 Looking through the features and deciding whether or not they are useful/essential (i.e. correlated to the output values). In other words, we check for correlations.
  + Feature derivation 🡪 Deriving new features from the old ones, generally by applying transforms to the dataset that simply change the axes (coordinate system) of the graph by moving and rotating them (the axes). Practically, this means applying different mathematical operations to the dataset, for example by applying a matrix to the data and thereby combine the matrix and the data.
  + Clustering 🡪 Group similar datapoints and see if this allows fewer features to be used.
  + Principal Component Analysis (PCA) 🡪 is a dimensionality reduction technique that uses feature derivation and aims to transform a high-dimensional dataset into a lower-dimensional representation while retaining as much information as possible. The idea is that by finding particular sets of coordinate axes, it will become clear that some of the dimensions are not required.
    - Centering the data 🡪 Helps remove any biases or offsets present in the dataset. By subtracting the mean from each feature, the data is shifted so that its average or centroid is positioned at the origin (0,0). This ensures that the principal components represent the directions of maximum variance, rather than being influenced by any global or systematic shifts in the data. Centering preserves linear relationships and simplifies the interpretation of the principal components. The components will now represent variations around the mean, making it easier to understand their influence on the dataset.
    - Covariance matrix 🡪 Tells us something about the scatter within a dataset, which is the amount of spread that there is within the data. If our dataset is easy to separate into classes, then this within-class scatter should be small, so that each class is tightly clustered together. The covariance matrix thus provides information about the relationships and variances between pairs of features. But we also want the distance between the classes to be large, so the between-classes scatter should be a large number and is calculated by simply taking the difference in the means of the classes.
    - Eigenvectors 🡪 Represent the directions or components along which the data varies the most.
    - Eigenvalues 🡪 Indicate the magnitude of the variance explained by each eigenvector.
    - Principal component 🡪 The direction in the data with the largest variation. The eigenvectors with the highest eigenvalues are considered the principal components. The principal components thus capture the most significant features and explain the majority of the variance in the data. The number of principal components chosen determines the dimensionality of the transformed space. Each principal component represents a direction in the feature space that captures the maximum amount of variance in the data.
    - Projection 🡪 Transforming the data points from the original high-dimensional space to a lower-dimensional space. This is done by multiplying the centered datapoints with the principal components. The resulting values represent the coordinates of each data point along the corresponding principal component axes.

**Artificial Developmental Systems**

An artificial developmental system is a combination of a developmental representation and the rules that specify how the representation must be interpreted to build the artificial system. In this course the focus is on systems inspired by the early developmental processes of multicellular organisms – that is, the development of the embryo. Artificial developmental systems can operate on different levels of abstraction – from the (low-level) details of the chemistry and physics of gene regulation and cell interaction – to (high-level) cell duplication. They are important in machine learning for representing different problems and can be used to create the topology of for example a network (graph) that make it easier to train and optimize a neural network that solves the problem.

* Some vocabulary on the biological background:
  + Morphogens 🡪 Chemicals that influence development. Provides a powerful mapping from genotype to a phenotype.
  + Heterochrony 🡪 Timed expression of morphogens
  + Homologous evolution 🡪 Shared genetic ancestry
* **Developmental representation** 🡪A developmental representation is a method of representing and organizing information that allows for continuous learning and adaptation of a system over time. It involves encoding and storing various types of data in a way that allows for the system to build upon and modify its knowledge as it interacts with its surroundings. A developmental representation provides the possibility of defining a compact description of potentially very complex structures. Minimal changes in the developmental description can result in major changes in the size and structure of the outcome of the developmental process.
  + **Repetition** 🡪 The conciseness of the representation is partly due to the possibility of reuse of the developmental programs describing the structure of substructures.
  + **Modularity** 🡪 The system is divided into specific modules/components that perform specific functions. This encapsulates different functionality, which results in efficiency in information retrieval.
  + **Self-similarity** 🡪 The structure exhibits similar patterns or characteristics at different scales or levels of the abstraction. In practice, it involves applying the same function/algorithm at different stages or levels of learning.
  + **Symmetry 🡪**
  + **Scalability** 🡪 The structure can be found in different scales. This is related to the fact that the developmental process is self-organized and distributed.
* **Rewriting systems** 🡪 A high-level class of formal models/systems that work on strings of symbols called words. The system defines a set of rules that specify how a word transforms into another word. Starting with an initial word, we can apply the rules recursively in order to obtain a sequence of words.
  + **L-systems** 🡪 A rewriting system that operates on strings of symbols. It is defined by assigning an alphabet A of symbols, an initial string of symbols w called the axiom, and a set of rewriting or production rules that specify how each alphabet symbol is replaced/extended by a string of symbols at each rewriting step. L-systems are called parallel rewriting systems because the production rules are applied in parallel to all the symbols of the current word at each iteration (in contrast to a formal grammar, which is a sequential system). A stopping condition is usually specified in the definition of the L-system – the simplest one being a certain, pre-defined number of rewriting steps. For L-systems that have the property of self-limited growth, the stopping condition can be the observation that the structure has not changed between two rewriting steps (that means it will never change again if it is deterministic).
    - **Graph interpretation of L-systems** 🡪 An L-system where the alphabet A is divided into two subsets N (nodes, often a set of alphabetic characters) and L (links, often a subset of the set of integers – denoting the weight). The words generated by an L-system for graph interpretation must be structured as a sequence of blocks composed by an element of N followed by one or more elements of L.
    - **Bracketed L-systems** 🡪 An L-system whose alphabet includes two additional symbols used to save and restore a certain state (the position and orientation of the cell):
      * **[** 🡪 pushes the state before this bracket onto a stack.
      * **]** 🡪 pops the state from the stack (returns the last element that was pushed).
    - **Stochastic L-systems** 🡪 An L-system that can associate multiple production rules with a single symbol of the alphabet, adding some randomness to the developmental process. The different production rules can then be applied with different pre-defined probabilities.
    - **Parametric L-systems** 🡪 Adds the possibility of having numerical parameters within the strings processed by the L-system and the possibility of manipulating them during the rewriting process. The parameters can be formal parameters belonging to a set X, or actual numeric or symbolic values. We can add an additional degree of flexibility by admitting conditions with the production rules. The parameters and conditions not only add flexibility, but also allow us to model biological parameters such as the concentration of chemicals in a cell, the age of a cell, or the number of divisions that it has undergone.
    - **Context-Sensitive L-systems** 🡪
* **Compositional Pattern-Producing Network (CPPN)** 🡪 A neural network that evolves both the activation function and the topology of the network. Any node can connect with any other node through mutations. Similarly, the activation function and of course the values of the weights of the network evolve. Can for example take two coordinates (x, y) from an image and output a specific color for that pixel.

**Swarm intelligence**

Swarm intelligence refers to the study of large collections of relatively simple agents that 1) can collectively solve problems that are too complex for a single agent, or 2) can display robustness and adaptability to environmental variation displayed by biological agents. The agents can use stigmergy – a specific type of social communication through modification of the environment to affect the behavior of another individual. It is not intentional and is therefore a form of cue-based communication. The agents can also communicate through intentional signals, which in turn lead to more complex dynamics. Different types of collective behavior can emerge, and can be seen as an extension of the genotype:

* Aggregation 🡪 Behavior affected by positive and/or negative feedback.
* Clustering 🡪 Making clusters and sorting objects for different purposes.
* Foraging 🡪 Collecting food or other objects and storing them in one place.
* Nest construction 🡪 Building complex architectural structures that exceed the perceptual and cognitive abilities of the single individuals.