# EXHAUSTIVE SEARCH - GENERAL ALGORITHM

# tests all possible solutions and is therefore guaranteed to return the best one

# can only be performed for small discrete problems with n <= 10

exhaustive\_search(start, solutions):

best\_solution = start

best\_value = evaluate\_solution(start), or infinity or -infinity # minimalization or maximalization problem

for each s in solutions:

value = evaluate\_solution(s)

if value < or > best\_value: # minimalization or maximalization problem

best\_solution = s

best\_value = value

return best\_soluton, best\_value

# GREEDY SEARCH - FOR TSP

# generates and evaluates one solution by picking the closest neighbour at each stage

greedy\_search(cities, distances):

solution = [|cities|]

solution[0] = arbitrary city from cities

distance = 0

dist = distances # copy map of distances so we can manipulate dist

for i in range(|cities|-1):

solution[i+1] = min(dist[solution[i], :]) # find the closest neighbour from here

distance += dist[solution[i], solution[i+1]]

dist[:, solution[i+1]] = infinity # make sure we don't visit this city again

distance += distances[solution[|solution|-1], 0]

return solution, distance

# HILL CLIMBING - FOR TSP

# performs a local search around the current solution by swapping two neighbours

# chooses any option that improves the result

hill\_climbing(cities, distances):

best\_soluton = arbitrary permutation of the array cities

p = best\_solution

best\_distance = evalueate\_distance(best\_solution)

# run for a pre-defined amount of swaps or until no swap improves the result for some pre-defined length of time

for i in range(1000): # pre-defined amount of swaps

start = random int between 0 and |best|-4

seq = best[start : start + 4]

seq[0], seq[1], seq[2], seq[3] = seq[0], seq[3], seq[2], seq[1] # swap two edges (1 and 3)

p[start : start + 4] = seq

new\_d = 0

for j in range(|p|-1):

new\_d += distances[p[j], p[j+1]]

new\_d = distances[p[|p|-1], p[0]] # distance between the last and first city

if new\_d < d:

d = new\_d

best = p

return [best, d]

# SIMULATED ANNEALING

# similar to hill\_climbing, but allows for exploration - which can help us avoid getting stuck in local optima

simulated\_annealing(cities, distances):

t = 20 # decide a temperature t, start at a high temperature for more exploration in the beginning

best\_solution = arbitrary permutation of the array cities

best\_distance = evaluate\_distance(best\_solution)

run for a pre-defined amount of iterations i.e evaluations

for i in range(1000):

new\_solution = generate\_random\_neighbour(best\_solution) # swaps two cities, like shown above

new\_distance = evaluate\_distance(new\_solution) # evaluates the new solution, like shown above

if new\_distance < best\_distance:

best\_solution = new\_solution

best\_distance = new\_distance

# if the new solution is worse, keep it with a probability p

# p depends on the difference in quality between the old and new solution and the temperature --> if t is high, we are open to choosing a new solution that is worse than the old one

else if SHOULD\_I\_EXPLORE(best\_solution, new\_solution, t):

best\_solution = new\_solution

best\_distance = new\_distance

t -= 0.1 # reduce the temperature to allow for less exploration and more exploitation

return best\_solution, best\_distance

# GRADIENT DESCENT/ASCENT

# choose a step size (gamma), this can be adjusted in the algorithm to obtain a higher efficiency

# start at a point (guess which one)

# calculate the derivative of the function in order to decide where to move next

# the derivative is calculated using n \* x^n-1 for each x

# the gradient is a vector that contains all the derivatives of all the dimensions we have

# it describes the length and the direction of the solution, and it points in the direction of the steepest ascent

# --> therefore we move towards it for gradient ascent and away from it for gradient descent

# stop when the gradient reaches (somewhere close to) 0 using a precision measure

# gradient descent moves in the direction where the solution quality changes the most

gradient\_descent(x, precision = 0.0001, gamma = 0.1):

dx = gamma \* df(x) # calculate the gradient using the step size

while the absolute value of dx > precision:

x = x + dx OR x - dx # descide the direction --> ascent vs. descent

dx = gamma \* df(x) # update the derivative of x

return x, f(x)

# EVOLUTIONARY ALGORITHM

# mimics the biological evolution to optimize solutions in a wide variety of complex problems

# in every new generation, a new set of solutions is created using the fittest parts of the old one

# termination conditions:

# --> We have reached a desired fitness value / a result we know is good enough

# --> After a certian amount of iterations

# --> We have reached a certain minimum amount of diversity in the population

# --> We have reached a certain amount of generations without obtaining a better fitness-value

evolutionary\_algorithm():

initial\_population = a set of random candidate solutions # or include some (good) existing solutions for problem-specific heuristics

evaluate each individual

repeat until termination condition is satisfied:

select parents # fitness\_proportional\_selection, rang\_selection or tournament\_selection

recombine the pairs of parents # operation depends on the representation

mutate the resulting offspring # operation depends on the representation

evaluate the new individuals

select individuals for the next generation

# THE PERCEPTRON

INITIALIZE a set of weights wij to small (positive and negative) random numbers

TRAIN:

for T iterations of until all the outputs are correct:

for each input vector:

compute the activation of each neuron j using activation function g (weighted sums):

sum of all weight \* x

update each of the weights individually using update rule 2

RECALL:

compute the activation function of each neuron j using the same function g

# K NEAREST NEIGHBOURS

# returns the majority label of the k nearest neighbours

# a small k makes the algorithm sensitive to noise, a large k reduces accuracy as points that are too far away are considered

# for classification we can count labels for each neighbour, for regression we can compute the average of the weighted sums

kNN():

distances = []

for each point in datapoints:

distance = calculate\_distance(point) # Euclidean distance

add distance to distances

sort distances in ascending order

k\_nearest = distances[:k] # identify nearest neighbours

class\_counts = {} # map with labels and a count of them

# count and add the labels of the nearest neighbours

for neighbour in k\_nearest:

label = label of the neighbour

class\_counts[label] += 1

# predict the majority class

predicted\_class = max(class\_counts, key = keys of class\_counts)

return predicted\_class

# RANDOM FOREST TRAINING

random\_forest():

for each of N trees:

create a new bootstrap sample of the training set

use this bootstrap sample to train a decision tree

at each node of the decision tree, randomly select m features, and compute the information gain (or Gini impurity) only on that set of features, selecting the optimal one

repeat until the tree is complete

# REINFORCEMENT LEARNING

reinforcement\_learning():

s[0] = initial state

w = initial weights or weights from a previous learning cycle (episode)

for i in range(N):

a = action(S[i], w) # implement the policy --> map from observation to action, and adjust through the parameter vector w

S[i+1] = newstate(S[i], a) # change the state based on the action we just made

r = reward(S[i], S[i+1], a) # calculate the reward based on the previous state, the current state, and the action

update the weights # update the weights (the policy) based on r

Q-learning():

initialize a Q-table where the actions in each state Q(s,a) have small random values

REPEAT until there are no more episodes:

initialise the state s

REPEAT for each step of the current episode:

select an action a using epsilon-greedy or another policy

take action a and recieve reward r

sample new state 's

update Q(s,a) with off-policy learning (update rule that assumes a greedy policy)

set s to be the s' # moving to the next state

SARSA():

initialize a Q-table where the actions in each state Q(s,a) have small random values

REPEAT until there are no more episodes:

initialise the state s

choose an action a using the current policy

REPEAT for each step of the current episode:

take action a and recieve reward r

sample new state 's

choose action a´ using the current policy

update Q(s,a) with on-policy learning (update rule that assumes the current policy)

set s to be the s' and a to be a' # moving to the next state

# UNSUPERVIZED LEARNING

k\_means\_clustering():

\*\*INITIALIZATION\*\*

# choose a value for k, often a hyperparameter

choose k random positions in the input space

assign the cluster centres to those positions

\*\*LEARNING\*\*

REPEAT until the cluster centres stop moving:

for each datapoint xi:

compute the distance to each cluster centre # euclidean is most used, but also manhatten and cosine

assign the datapoint to the nearest cluster centre

for each cluster centre:

move the position of the centre to the mean of the points in that cluster

# use update function --> average of all datapoints

\*\*USAGE\*\*

for each test point:

compute the distance to each cluster centre

assign the datapoint to the nearest cluster centre

k\_means\_network():

\*\*INITIALIZATION\*\*

choose a value for k, which corresponds to the number of output nodes

initialize the weights to have small random values

\*\*LEARNING\*\*

normalise the data so that all the points lie on the unit sphere

REPEAT until number of iterations is above a threshold:

for each datapoint:

compute the activations of all the nodes

pick the winner as the node with the highest activation

update the weights using the update function

\*\*USAGE\*\*

for each test point:

compute the activations of all the nodes

pick the winner as the node with the highest activation

PCA():

center the data by subtracting the mean of each feature (column) from each feature

compute the covariance matrix C

compute the eigenvectors V and eigenvalues D of the covariance matrix

sort the columns of D in a descending order

apply the same order to the columns of V

\*\*PC\*\* select the desired amount of learned features by choosing the first n eigenvectors # essentially choosing the principal components

project the centered data onto the principal components by:

\*\*PD\*\*

multiplying the centered datapoints with the selected eigenvectors

return the principal components (\*\*PC\*\*) and the projected data (\*\*PD\*\* --> the data in a lower dimensional space)