Exam in3050 – spring 2020

Candidate number: 15398

Task 1: Search/optimization

a)

I-dimensional search landscape

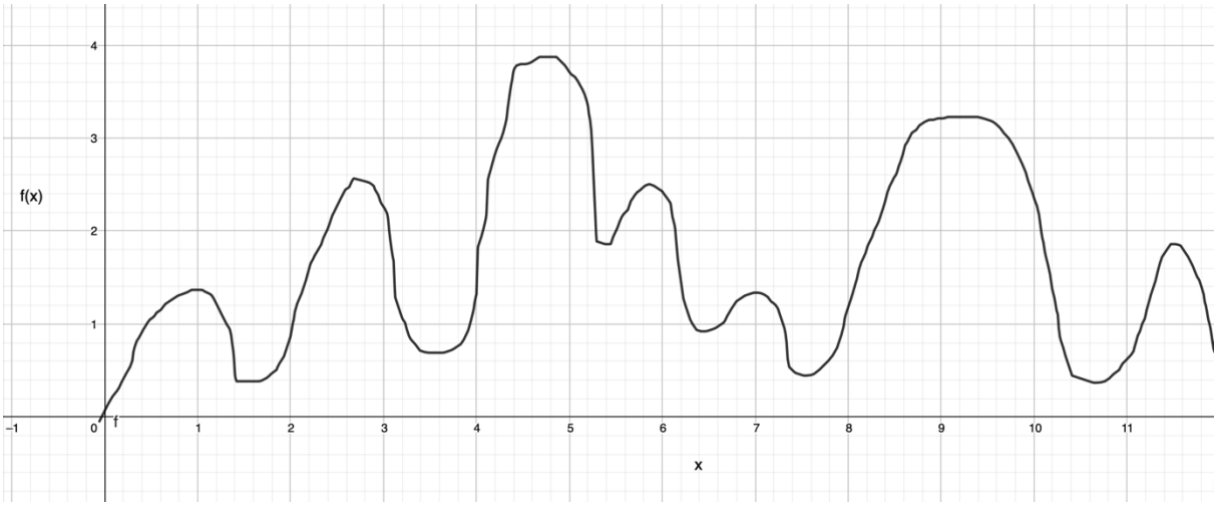

Image 1: search landscape

b) Local optima

In a search space like this, there are multiple local optima. The local optima points are more the maximum or minimum (depends on the problem) within a set of neighbor solutions. Once the solution got no better neighbor solutions, it will stop at the local optima.

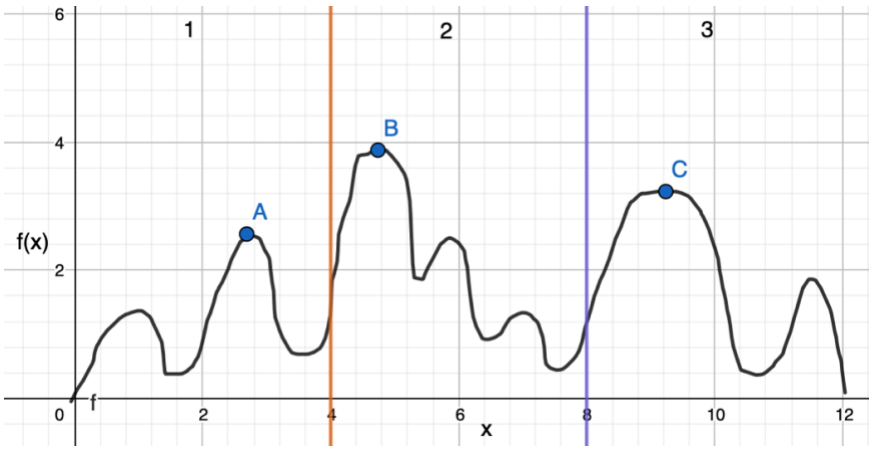

Image 2: local and global optimum for optimizing problem

In image 2, I have separated 3 different sets of local points. All three point are local optima for their set of solutions.

c) The global optimum

The global optimum is the most optimum local optima for all the solutions. It could also be maximum or minimum depending on the problem. In image 2, B is the global optimum for this particular optimization problem.

d) Exploration

Exploration is the action of choosing completely new solutions. This method is used to prevent the search from being stuck in a local optimum and contributes with exploring more sets of solutions. This is used in search algorithms like exhaustive search. In image 2, even if the search finds the local maximum A in set 1, it will also make some random searches so that it may find better solution in other part of the search landscape like point C.

e) Exploitation

Exploitation is the action of searching for better solution than the ones we already have. We get better solution, but we might also end up stuck in a local optimum, therefore it is better to use a combination of exploration and exploitation. Exploitation is used in search algorithms like hill climbing algorithm.

f)

Pure-exploitation algorithms is very much likely to find a local optimum, because it will look for better solution until there is no better neighbor solution. The algorithm might be lucky and find the global optimum if it starts searching close to it, but it will be very unlikely in a big search landscape.

Task 2: Evolutionary Algorithm

a)

Genotype: since we are trying to optimize the social distancing in the squared room, I have chosen to represent the genotype as permutation bit strings. The corresponding genotype to the figure could be: 1001000001000010

phenotypes: The phenotype in GA is an individual of a candidate solution and, therefore I have chosen the phenotype as the square room of NxN meters.

Fitness function: The fitness function in this case is the maximum total distance from the people in the room divided by total distance from every person to another person in the current room. There are N people in the room, and the maximum distance from one person to another is also N. Therefore, the maximum total distance from the people to another person is NxN. The total distance from every person in the current phenotype is x. The fitness function will then be $f(x) = x / (N * N)$. Let us take the example phenotype as a demonstration:

$x = 3 + 2 + 2 + 3 = 10$

$f(10) = 10 / (4 * 4) = 0.625$ (1 is max fitness)

Mutation operator: for mutation operator I have chosen bitwise mutation with a little twist. If a bit with value 1 change, another random bit with value 0 will also change and vice versa. This is like a swap mutation, and the reason for using this is to maintain the correct number of people in the room.

Crossover operator: I need to have an operator that takes care of the permutation bits. Therefore, I have chosen cycle crossover.

Initialization criterion: random initialization to ensure spread.

Termination criterion: we should have a specified number of generations since there is no limit on how big the room can be. Therefore, I have specified N * N number of generations without improvement.

Selection operator: roulette wheel selection for exploitation and some exploration as parent selection. $(\mu+\lambda)$-selection for large selection pressure.

b)

There is a requirement that says it has to be exactly N people in the room, so we have to make sure that mutation and crossover does not remove a 1 bit without adding a 1 bit or add a 1 bit without removing a 1 bit. It is important that the initialization also makes sure that there are N people even though it is random.

c)

We can use fitness sharing in order to preserve diversity and avoid premature convergence in the population. Fitness sharing restricts groups of individuals that are similar and gives them shared fitness value. In our problem, we have a lot of similar solution with the same minimal distance. The way fitness sharing can be implemented for this problem is by using the equation for fitness sharing for every fitness of the individual:

$$f'(x) = \frac{f(x)}{\sum_{y=1}^{\mu} sh(d(x,y))}$$

where f(x) is the old fitness and sh(d) is a value between 0 and 1, indicating how much of the fitness should be shared between the individuals. I would have to make a distance metric in order to calculate the distance d. Since the maximum total distance in a room ranges from 8 to 16, there is going to be a lot of solution for each shared fitness.

d)

1)

A way to hybridize the EA is to combine it with local search operators. Local search is a way of looking for improved versions of a current solution by doing small local changes. It only keeps a solution if it is better than the one before. An example in this problem is to swap two bits in a genotype and check whether that solution is better than it was. If it was not better, then swap other bits.

2)

The problem is that local search will always look for better solutions until it finds it, and always focus on the fittest individual. This can lead to premature convergence and the search could end up in a local optimum. On the other hand, we have the fitness sharing that tries to

avoid premature convergence by grouping similar solutions together. This will lead to a conflict.

3)

Fitness sharing and hybridization wants the opposite of each other. Fitness sharing is exploring different peaks of the fitness landscape with the different niches, while hybridization wants to exploit the exploit and improve the already known solutions.

Task 3: Classification

a)

The classifier is deciding which class C and D belong to, based on the features and training data. The classifier finds the decision boundary based on the training data, and that can be to separate the different classes. The decision boundary is on the form of $y = w * x + b$ (slope + intercept), and we can clearly see that the decision boundary line in this specific problem is $y = 2x + 2$.

If $2x + 2 > 0$ means that the value is on the positive side.

If $2x + 2 < 0$ means that the value is on the negative side.

Plot the values in the formula:

$2 * (-2) + 2 = -2$ (left - negative side)

$2 * (2) + 2 = 6$ (Right – positive side)

Note(The classifier will need to find the decision boundary by calculating the slope and intercept).

b)

Assume bias is -1 and learning rate is 0.2.

x0 (bias)	x1	x2
-1	1	2
-1	1	1

Setting weights to small random numbers:

w0 = -0.3

w1 = 0.5

w2 = 0.2

weights after considering point E:

w0 = -0.3 + 0.2 * (0-1) * -1 = -0.1

w1 = 0.5 + 0.2 * (0-1) * 1 = 0.3

w2 = 0.2 + 0.2 * (0-1) * 2 = -0.2

weight after considering F:

w0 = -0.1 + 0.2 * (1-1) * -1 = -0.1

w1 = 0.3 + 0.2 * (1-1) * 1 = 0.3

w2 = -0.2 + 0.2 * (1-1) * 1 = -0.2

c)

The learning rate decides how fast the algorithm shall learn. By having a small learning rate value in the perceptron algorithm, the weights need to have a lot of input in order to change significantly. That will result in slow, but also more stable, learning. When having a big learning rate value, the weights will change a lot even when the answer is wrong. The best choice is to use a moderate learning rate in order to balance the precision and speed of the learning.

d)

The learning rate in gradient descent for linear regression has the same concept as the learning rate in perceptron. When the learning rate is high, the gradient descent will take longer steps and could end up overshooting the local minimum. Low learning rate will result in smaller steps and slower learning. Because of this, moderate learning is also best in gradient descent.

Task 4: Evaluation

a)

Confusion matrix

N = 30	Predicted: red	Predicted: blue
Actual: red	2	0
Actual: blue	6	22

Accuracy: (true positive + true negative) / total N

(22 + 2) / 30 = 0.8

The accuracy score for this classifier is 0.8.

b)

precision: true positive / (true positive + false positive). How often the predicted class is correct.

precision blue: 22 / 22 = 1

precision red: 6 / 8 = 0.75

recall: true positive / (true positive + false negative). How often a predicted class is the actual class.

Recall blue: 22 / 28 = 0.79

Recall red: 2 / 2 = 1

c)

We should rely the on also precision and recall when we need to measure the relevance of the accuracy. A real-life problem could be when a doctor diagnoses a set of patients if they have a dangerous virus. Because of the importance of predicting the right patient to the right class, then it is important to see how often they get the class that they get and how often the predicted class is the actual class.

Task 5: Neural networks

a)

I assume that the weights and bias are the same as in the figure.

m = 2

n = 3

input = [1, 2]

weights_input :[0.1, 0.2, 0.3
0.4, 0.5, 0.6
0.7, 0.8, 0.9]

Weights_hidden:[0.1,
0.2,
0.3,
0.4]

Calculate hidden layer:

h1 = 1 * 0.4 + 2 * 0.7 + 0.1 = 1.9

h2 = 1 * 0.5 + 2 * 0.8 + 0.2 = 2.3

h3 = 1 * 0.6 + 2 * 0.9 + 0.3 = 2.7

Activation function on hidden layer:

h1 = 1 / (1 + e^-1.9) = 0.87

h2 = 1 / (1 + e^-2.3) = 0.91

h3 = 1 / (1 + e^-2.7) = 0.94

Calculate output:

Output = 0.87 * 0.2 + 0.91 * 0.3 + 0.94 * 0.4 + 0.1 = 0.92

b)

output: t = 10

learning rate = 0.1

Error at the output node:

$\delta_O = y_j(1 - y_j)(t_j - Y_j)$

$\delta_O = 0.92(1 - 0.92)(10 - 0.92) = 0.67$

Error at the hidden layer:

$\delta_h = a_h(1 - a_h) \sum_{j=1}^{m} w_{j,h}\, \delta_j$

$\delta_{h1} = 0.87(1 - 0.87) * 0.67 * 0.2 = 0.015$

$\delta_{h2} = 0.91(1 - 0.91) * 0.67 * 0.3 = 0.016$

$\delta_{h3} = 0.94(1 - 0.94) * 0.67 * 0.4 = 0.015$

Update output weights:

$w_{h,o} = w_{h,o} - learning\ rate * output\ error * hidden\ activation$

$w_{h1,o} = 0.2 - 0.1 * 0.67 * 0.87 = 0.14$

$w_{h2,o} = 0.3 - 0.1 * 0.67 * 0.91 = 0.24$

$w_{h3,o} = 0.4 - 0.1 * 0.67 * 0.94 = 0.34$

Update hidden weights:

$w_{i,h} = w_{i,h} - learning\ rate * hidden\ error * x_i$

$w_{i1,h1} = 0.4 - 0.1 * 0.015 * 1 = 0.4$

$w_{i1,h2} = 0.5 - 0.1 * 0.016 * 1 = 0.5$

$w_{i1,h3} = 0.6 - 0.1 * 0.015 * 1 = 0.6$

$w_{i2,h1} = 0.7 - 0.1 * 0.015 * 2 = 0.7$

$w_{i2,h2} = 0.8 - 0.1 * 0.016 * 2 = 0.8$

$w_{i2,h3} = 0.9 - 0.1 * 0.015 * 2 = 0.9$

Network after backwards step

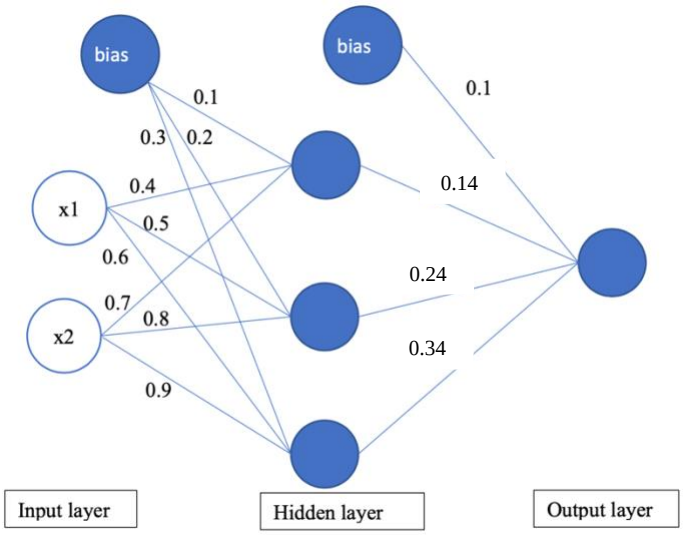

Image 3: updated weights in networ

Task 6: Multi-class classification

a)

Multi-class classification: a way of classifying when there are multiple classes in the problem. To approach this problem, we have used “one hot encoding” to make both the target value and predicted value into vectors, and we used one vs. rest strategy to train one classifier for each class.

b)

The reason why representing each label as numerical values is not a good idea is that the feed forward neural network wants to know the connection and similarities between the labels in order to find the right one for each observation.

c)

One hot encoding is a binary representation of classes. These binary representations are vectors in order to make the classifier train classifier for each class.

One-hot-encoding for the 6 classes:
apple = (0, 0, 0, 0, 0, 1)
banana = (0, 0, 0, 0, 1, 0)
grapes = (0, 0, 0, 1, 0, 0)
orange = (0, 0, 1, 0, 0, 0)
pear = (0, 1, 0, 0, 0, 0)
plum = (1, 0, 0, 0, 0, 0)

d)

One vs rest works by splitting the multi-class dataset and training a single classifier per class. An example could be:
Classification problem 1: apple vs [banana, grapes, orange, pear, plum]
Classification problem 2: banana vs [apple, grapes, orange, pear, plum]
Classification problem 3: grapes vs [apple, banana, orange, pear, plum]
And so on…

Task 7: Data scaling

a)

The min-max scaler is rescaling the data so that the features are mapped to number in the interval [0, 1].

Two observations:

A: (4.0, 2.0, 140) and B: (3.0, 0.1, 115)

Scale(xi) = xi – min(x) / max(x) – min(x)

Scale(A1) = (4 – 2.5) / (10.5 – 2.5) = 0.19

Scale(A2) = (2 – 0.2) / (3 – 0.2) = 0.64

Scale(A3) = (140 – 120) / (150 – 120) = 0.67

Scale(B1) = (3 – 2.5) / (10.5 – 2.5) = 0.06

Scale(B2) = (0.1 – 0.1) / (3 – 0.1) = 0

Scale(B3) = (115 – 115) / (150 – 115) = 0

A and B scaled:

A: (0.19, 0.64, 0.67) and B: (0.06, 0, 0)

b)

Using standard scaler on the same observations. The standard scaler tells us how far from the mean we are in terms of standard deviations, and it also got a higher variance than the min max scaler.

Scale(xi) = (xi - mean) / stdev

Scale(A1) = (4 - 6) / 2 = -1

Scale(A2) = (2 – 1.3) / 0.7 = 1

Scale(A3) = (140 - 140) / 5 = 0

Scale(B1) = (2 - 6) / 2 = -2

Scale(B2) = (0.1 – 1.3) / 0.7 = -1.7

Scale(B3) = (115 - 140) / 5 = -5

A and B scaled:

A: (-1, 1, 0) and B: (-2, -1.7, -5)

c)

Scaling in K nearest neighbors (KNN): KNN is sensitive to scaling, because it looks at the distance between the data points. The scaling will help to normalize the data within a particular range, and that will decide the result. When be scale the data, the axis will be stretched in order for the nearest neighbors to get better distances between the points. The point is that when the data is normalized, the accuracy and set of k neighbors will be different than without scaling.

Scaling in logistic regression: I believe scaling will help the speed of training in logistic regression. Scaling is not required and will not affect the model in most cases, but it will help the classifier if the value difference of the features is big.

Scaling in feed-forward neural network (MLP): In feed-forward neural network we should scale the the data close to 0. The prediction (product of weight and feature) should not be too large because if it is, not much learning can go on because the gradient will be close to 0 and that will result in slow learning.

Task 8: Q-learning

a)

discount factor = 0.9

learning rate = 0.2

Actions

Q(2, down) <- 0 + 0.2 * (5 + 0.9 * 0 - 0) = 1

Q(5, right) <- 0 + 0.2 * (-0.5 + 0.9 * 0 - 0) = -0.1

Q(6, up) <- 0 + 0.2 * (-0.5 + 0.9 * 0 - 0) = -0.1

Q(3, right) <- 0 + 0.2 * (-1 + 0.9 * 0 - 0) = -0.2

Q(3, left) <- 0 + 0.2 * (-0.5 + 0.9 * 1 - 0) = 0.08

Q(2, down) <- 1 + 0.2 * (5 + 0.9 * 0 - 1) = 1.8

Q(5, right) <- -0.1 + 0.2 * (-0.5 + 0.9 * 0 – (-0.1)) = -0.18

Q(6, down) <- 0 + 0.2 * (-0.5 + 0.9 * 0 - 0) = -0.1

Updated values after actions

Image 4: q-values av actions

b)

The reason the Q-value changes after the second time of performing “down” is that the calculation of Q-value is using the previous Q-value to calculate the next. This way, the algorithm learns which actions are better than others. If the “down” action is performed many times, the Q-value will increase, and it will become clearly that this is a good way to go. I believe that the Q-value will eventually converge towards optimal values if each action is executed in each state infinite time.

c)

Rank of how much the policies explore:

1. A soft-max policy with a temperature of 1
2. An epsilon-greedy policy with epsilon equal to 0.9
3. A greedy policy

The probability of soft-mac policy to go “down” from state 2 = 25% chance, since the temperature is 1. That means it is purely explorational, and every action will have a 25% chance off happening. That is also why it got 1st place of exploration.

The probability of epsilon-greedy policy to go “down” from state 2 = 10% chance, since the epsilon is set to be 0.9 the probability of selecting random actions is 0.9. That is why it got 2nd place of exploration.

The greedy policy will always choose the best action and in this case, it will go “down” 100%. That is why it got 3rd place in exploration.

d)

Discount factor has a role of deciding how much the algorithm will care about future rewards and is important for the algorithm to choose a better path. If we reduced the discount factor to 0.1, the algorithm would have cared more about immediate rewards. The q-values would have been higher because of the downward adjustment of the discount.

e)

Calculating the Q-value based on the actions decided by the cat’s policy.

Q(2, down) <- 0 + 0.2 * (5 + 0.9 * 0 - 0) = 1

Q(5, right) <- 0 + 0.2 * (-0.5 + 0.9 * 0 - 0) = -0.1

Q(6, up) <- 0 + 0.2 * (-0.5 + 0.9 * 0 - 0) = -0.1

Q(3, right) <- 0 + 0.2 * (-1 + 0.9 * 0 - 0) = -0.2

Q(3, left) <- 0 + 0.2 * (-0.5 + 0.9 * 1 - 0) = 0.08

Q(2, down) <- 1 + 0.2 * (5 + 0.9 * (-0.1) - 1) = 1.782

Q(5, right) <- -0.1 + 0.2 * (-0.5 + 0.9 * 0 – (-0.1)) = -0.18

The different Q-value is Q(2, down).

Off-policy (Q-learning) gives higher Q-values than on-policy(SARSA). The reason for this is that Q-learning is calculating the Q-value based on an assumption of choosing the next action with the highest amount of reward, while SARSA is estimating the future value based on the action made by the policy. In this case, the Q-value from Q-learning is after action “Down” in

state two the second time is 1.8 because it used the maximum future value. In SARSA, the Q-value after that action is 1.78. Here we are not using the maximum future value, but instead we are using the future value of the next action based on the policy. This is done because SARSA wants to explore more than Q-learning.

f)

The cat should have had a penalty for going back and forth from two tiles. With the q-values as in image 3, the cat will gradually get higher reward as long as it keeps going from 2 to 3 and 3 to 2.

Task 9: Particle swarm 0ptimization

a)

$$x_{i,d}(it+1) = x_{i,d}(it) + v_{i,d}(it+1) \qquad (1)$$

The particles' position is represented as "x" and is updated by the velocity that is represented as "v". In the new iteration, the new position of the particle is the sum of the old position and the new velocity.

Best found solution

$$\begin{aligned} v_{i,d}(it+1) &= v_{i,d}(it) \\ &+ C_1 * Rnd(0,1) * [pb_{i,d}(it) - x_{i,d}(it)] \\ &+ C_2 * Rnd(0,1) * [gb_d(it) - x_{i,d}(it)] \end{aligned} \qquad (2)$$

The updating of the velocity is in two parts:

- Adjust the velocity of a particle based on its best-found solution represented as "pb".
- Adjust the velocity of the particle based on the best-found solution out of all the particles represented as "gb".

b)

The global best solution is required for updating the velocity, but it might still work anyways. Without the global best solution, we would only update the velocity based on the particle's

best solution, and because of that the search for solution could end up trapped at a local optimum. It will become less exploration.

c)

1. The particles correspond to the individuals in a population. They have kind of the same characteristics and they both have population-based metaheuristic.
2. The positional parameters correspond to the genes of each individual. Both of them are representative adjustable/mutable parameter of an individual.
3. The positional update rule corresponds to the genotype of an individual and is where the individual is relative to the outer conditions.