ΤΜΗΜΑ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
& ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΑΡΙΣΤΟΤΕΛΕΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΟΝΙΚΗΣ

Computational Intelligence

Παναγιώτης Καρβουνάρης

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών
Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης

Contents

1 Reinforcement Learning

1.1 Introduction to the RL problem

For the reinforcement learning project we chose to solve the maze problem. So the goal is to train an agent to solve a maze. The features of this game are simple:

- **Maze:** is the environment of the game. It has specific dimensions and have for kind of tiles, the player tile, the treasure tile, the wall tile and the empty tile. Both treasure and player can be only on empty tiles.
- **Player:** starting in a random position in the maze and his goal is to find the treasure with the least amount of steps.
- **Treasure:** is located in a random position inside the maze and if the player find the treasure the game ends.

1.2 RL Environment

In our problem we have created mazes with different dimensions (6x6, 8x8, 10x10). The environment provides the state, the information about whether the game is done or not and the reward for each action. Also, in our case it gives a visual representation of the current state of the maze. Let's analyze one by one each of these features:

- **State:** is practically the whole maze in form of a 2D grid. On this grid there is information about the stationary objects of the maze like the walls of the empty tiles, but also there is the position of the of the player which change every step and the position of the treasure.
- **Action:** the layer can do 4 moves, he can go up, down, left or right. If a specific move goes to a wall tile the player stays in the same tile, he can't move.
- **Reward:** is a value (float in our case) that provides a feedback on how good or bad was a certain action.
- **Done:** is a boolean type variable that provides the information about if the game is finished or not.

1.3 RL Agent

We used three different CNN models, the first one is simple and is custom model by us, the second one is based on dueling DQN-CNN architecture and the third one is based on inception DQN-CNN architecture.

Custom DQN-CNN

- **Model Overview:** The DQNCNN class, is tailored for reinforcement learning applications in maze-solving tasks. It uses convolutional layers, batch and layer normalization to effectively process spatial grid data representing the maze.
- **Input Specification:**
  - The network is designed to handle a grid input that depicts the maze structure, where the grid contains specific symbols representing walls, empty spaces, the player, and the treasure.
  - Input to the network is a tensor of shape $(1, H, W)$, representing a single-channel image where H and W denote the height and width of the grid, respectively. Each cell in the grid encodes specific elements like walls or paths.
- **Layers and Functions:**
  - **Convolutional and Normalization Layers:**
    - **Conv1 and BN1:** First layer applies a 3x3 convolution with 16 filters, stride 1, and padding 1, followed by batch normalization. This setup extracts basic spatial features such as the presence of walls and pathways.
    - **Conv2 and BN2:** A second convolution layer with 32 filters of the same size further processes features, preparing them for high-level abstraction.
  - **Activation Function:** ReLU is utilized after each normalization to introduce non-linearity, facilitating the learning of complex maze structures.
  - **Flattening:** The convolutional output is converted into a 1D vector to feed into fully connected layers.
  - **Fully Connected and Normalization Layers:**
    - **FC1 and LN1:** Transforms the flattened vector into a 128-dimensional space, with layer normalization stabilizing the learning process by normalizing the activations.
    - **FC2 and LN2:** Reduces to a 64-dimensional space, also normalized, preparing for decision output.
    - **FC3:** Outputs four values, each representing the Q-value for moving in one of four possible directions in the maze (up, down, left, right).
- **Output Specification:** The network outputs Q-values which estimate the utility of taking each action given the current state (grid configuration). These values guide the player's movements towards the goal.

Dueling DQN-CNN

- **Model Overview:** The DuelingDQNCNN class, features a dueling network architecture, designed to separately estimate state values and action advantages in a maze-solving task. This model is beneficial in reinforcement learning where such a distinction can lead to more robust policies.
- **Input Specification:**
  - The network accepts input tensors representing the state of the maze, typically encoded as a single-channel (grayscale) image of the grid, with dimensions $(1, H, W)$. This grid includes representations for walls, paths, the player, and the treasure.
- **Layers and Functions:**
  - **Convolutional and Normalization Layers:**
    - **Conv1 and BN1:** A 3x3 convolutional layer with 16 filters, stride 1, and padding 1, followed by batch normalization, processes basic spatial features.
    - **Conv2 and BN2:** Another 3x3 convolutional layer with 32 filters, enhancing the feature extraction, also followed by batch normalization.
  - **Activation Function:** ReLU is used to introduce non-linearity, enabling the network to learn complex patterns within the maze.
  - **Flattening:** Outputs from the convolutional layers are flattened into a one-dimensional vector to facilitate fully connected processing.
  - **Dueling Fully Connected and Normalization Layers:**
    - **Value Stream:**
      - **Value_FC1 and LN1:** Maps the flattened vector to a 128-dimensional space, followed by layer normalization, to compute a single state value.
      - **Value_FC2:** Outputs the scalar state value representing the overall value of the current state.
    - **Advantage Stream:**
      - **Advantage_FC1 and LN2:** Also processes the flattened vector through a 128-dimensional space, followed by layer normalization.
      - **Advantage_FC2:** Outputs a vector of four values, each representing the advantage of taking one of the possible actions (up, down, left, right).
    - **Combining Values:** The final Q-values are computed by combining the state value with the advantages and adjusting by subtracting the mean of the advantages, enhancing relative decision-making capabilities.

- **Output Specification:** The network outputs Q-values for each possible action from the current state, which are used by the agent to make informed decisions about the next move in the maze.

Inception DQN-CNN

- **Model Overview:** The InceptionDQNCNN class, employs an inception-style architecture, using multiple convolutional filters of different sizes at the same level. This structure allows it to capture information at various scales and contexts, useful in complex environments like mazes where local and broader spatial features are relevant for decision-making.
- **Input Specification:**
  - The model processes input tensors that represent the maze's grid state. These tensors are single-channel images with dimensions $(1, H, W)$, encoding elements like walls, paths, the player, and the treasure within the maze.
- **Layers and Functions:**
  - **Inception Modules:**
    - **Branch1x1:** A 1x1 convolution with batch normalization captures local features without altering spatial dimensions.
    - **Branch3x3:** A 3x3 convolution with padding maintains the spatial dimensions while capturing medium-scale features.
    - **Branch5x5:** A 5x5 convolution, again padded, to capture larger spatial features.
    - **Branch_pool:** A max pooling layer followed by a 1x1 convolution captures the most prominent features while maintaining dimensionality.
  - **Combining Features:** The outputs of all branches are concatenated along the channel dimension, combining different feature scales effectively.
  - **Further Convolution and Normalization:**
    - After concatenation, a 3x3 convolutional layer followed by batch normalization processes the combined features for further abstraction.
  - **Activation Function:** ReLU is employed extensively to add non-linearity, facilitating the learning of complex patterns in the maze.
  - **Flattening and Fully Connected Layers:**
    - The output is flattened into a vector to transition to dense layers.
    - Dense layers with layer normalization reduce the feature dimensionality, preparing the network to output action values.

  - **Output Layer:** The final fully connected layer outputs four values, representing the estimated Q-values for the four possible actions (up, down, left, right).

- **Output Specification:** The network outputs a vector of Q-values, guiding the RL agent's decision-making process by indicating the potential reward of each available move from the current state.

Usage in Reinforcement Learning

- In a maze environment, the agent uses the Q-values to decide the optimal path to the treasure, learning over time to navigate through the maze efficiently.
- The network enables the agent to adapt its strategy based on the encountered configurations, optimizing its path to the goal with minimal encounters with obstacles.
- The model is trained to update weights based on the reward outcomes, i.e., reaching the treasure or hitting walls, using algorithms like Q-learning with experience replay to improve the decision-making process.

1.4 Reinforcement Learning Architecture

The RL system comprises several components:

- An environment that represents the maze, which includes obstacles, a player position, and a target position.
- A neural network model that approximates the optimal action-value function.
- An optimizer that minimizes the loss between predicted Q-values and the target Q-values, enhancing the learning process.
- A memory buffer that stores experience to enable experience replay, a critical method for efficient learning.

1.5 Mathematical Foundation

The primary goal of the agent is to learn the action-value function, which provides the expected utility of taking an action in a given state and following a particular policy thereafter. The action-value function, $Q(s,a)$, is defined using the Bellman equation as follows:

$$Q(s,a) = r + \gamma \max_{a'} Q(s',a') \tag{1}$$

where s is the current state, a is the action taken, r is the reward received, s' is the new state, a' is the next action, and γ is the discount factor, which models the difference in importance between future rewards and immediate rewards.

1.6 Learning Process

- **Experience Replay:** The agent's experiences at each time step, $e_t = (s_t, a_t, r_t, s_{t+1})$, are stored in a data set D, from which mini-batches are sampled randomly. This breaks the correlation between consecutive samples, stabilizing the learning algorithms.

- **Q-Learning Updates:** For each sample in the mini-batch, a loss function is calculated between the current estimated Q-values and the target Q-value, which is computed as:

  $$L = \left(r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta)\right)^2 \tag{2}$$

  Here, θ denotes the parameters of the current network, and θ^- denotes the parameters of a target network, which is a slightly outdated version of θ and is used to stabilize training.

1.7 Epsilon-Greedy Strategy

To balance exploration and exploitation, an epsilon-greedy strategy is employed, where the agent selects a random action with probability ϵ and the best-known action with probability $1 - \epsilon$. Over time, ϵ is decayed exponentially towards a minimum value to ensure the agent explores less as it becomes more confident in its value estimates.

1.8 Training and Testing

The agent is trained over multiple episodes, with each episode involving the agent navigating from a start position to a goal within the maze. Performance metrics such as total reward per episode and average loss per training episode are tracked and visualized. Post training, the agent is tested on new maze configurations to evaluate its generalization ability.

1.9 Double DQN Algorithm

Standard DQN uses a single network to both select and evaluate an action, which can lead to significant overestimations of Q-values. Double DQN addresses this issue by decoupling the selection of actions from their evaluation:

- An *online network* predicts the next action.

- A *target network* evaluates the action chosen by the online network.

The key steps in the Double DQN algorithm are outlined below:

1. Initialize the online network with random weights and duplicate these weights to the target network, which is then kept static for a fixed number of steps.

2. Collect experiences in the replay buffer from the environment using the current policy derived from the online network, following an epsilon-greedy strategy for exploration.
3. Sample mini-batches from the replay buffer and for each batch:
    - Use the online network to select the best action for the next state.
    - Use the target network to calculate the Q-value for these actions (action evaluation is decoupled).
4. Calculate the target Q-value as:
$$y_t = r + \gamma Q(s', \arg\max_a Q(s', a; \theta); \theta^-) \tag{3}$$
where θ are the parameters of the online network, θ^- are the parameters of the target network, s' is the next state, r is the reward, and γ is the discount factor.
5. Update the online network by minimizing the loss:
$$L(\theta) = (y_t - Q(s, a; \theta))^2 \tag{4}$$
6. Periodically update the weights of the target network with the weights of the online network to stabilize learning.

1.10 Plotting

For the implementation of the RL environment, agents and logic we used python and the packages pytorch, numpy and matplotlib. The hyperparameters used are below (these are the default hyperparameters if any plot has something different then the default option does not apply to this prol):

- **Episodes:** 120
- **Discount Factor:** 0.95
- **Learning Rate:** 0.005
- **Batch Size:** 128
- **Memory Size:** 10000
- **Epsilon Start:** 1.0
- **Epsilon End:** 0.005
- **Epsilon Decay:** 0.98
- **Minimum Replay Size:** 100
- **Target Update Frequency:** 5
- **Train Frequency:** 50 (for the double DQN)

We can see the plots below

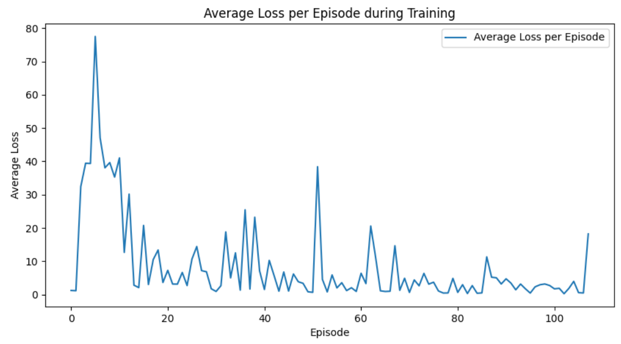

Figure 1: Average loss per episode using the custom DQN with normalization to the convolutional and the linear layers

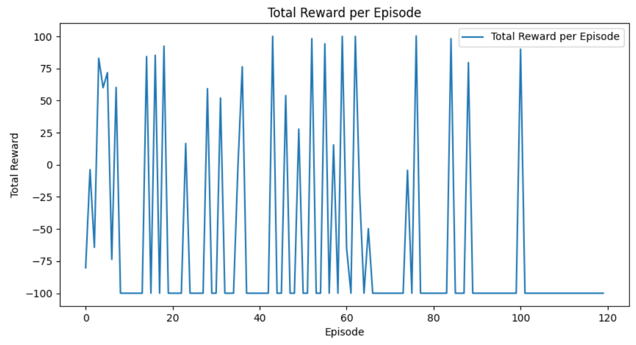

Figure 2: Total reward per episode using the custom DQN with normalization to the convolutional and the linear layers

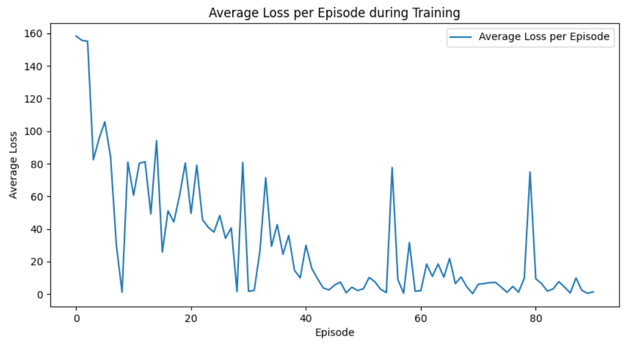

Figure 3: Average loss per episode using the double dueling DQN with normalization to the convolutional and the linear layers

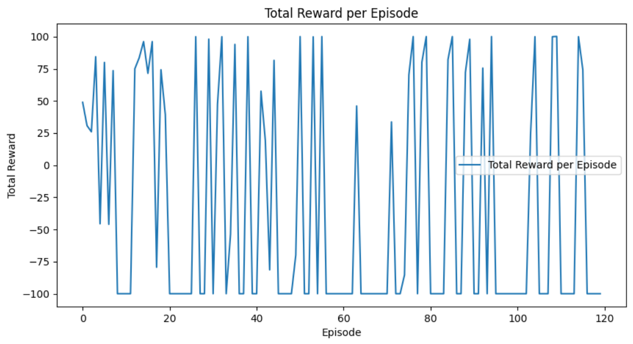

Figure 4: Total reward per episode using the double dueling DQN with normalization to the convolutional and the linear layers

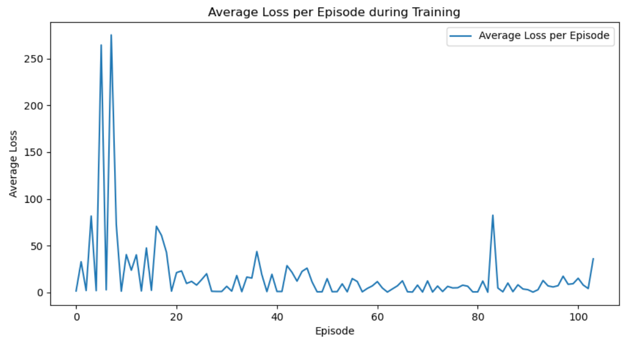

Figure 5: Average loss per episode using the double inception DQN with normalization to the convolutional and the linear layers

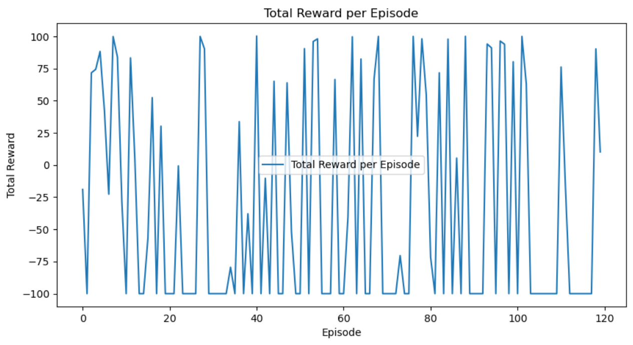

Figure 6: Total reward per episode using the double inception DQN with normalization to the convolutional and the linear layers

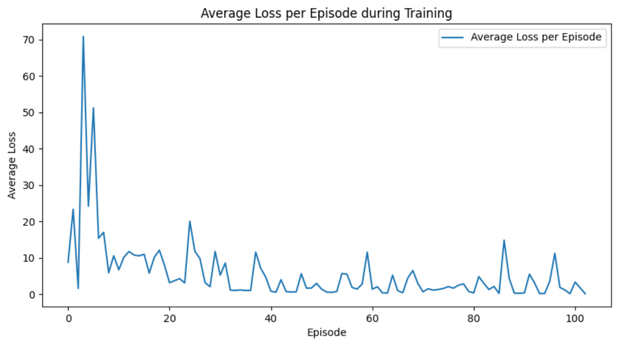

Figure 7: Average loss per episode using the dueling DQN with normalization to the convolutional only

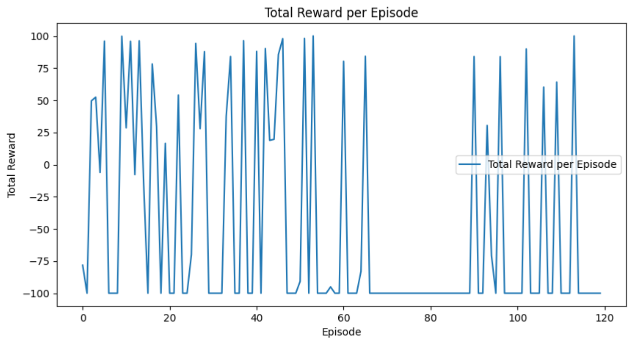

Figure 8: Total reward per episode using the dueling DQN with normalization to the convolutional only

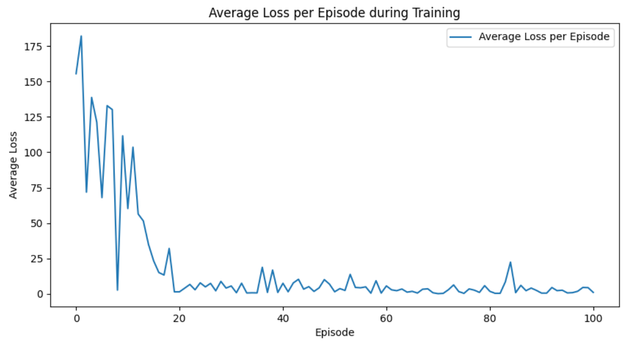

Figure 9: Average loss per episode using the dueling DQN with normalization to the convolutional and the linear layers

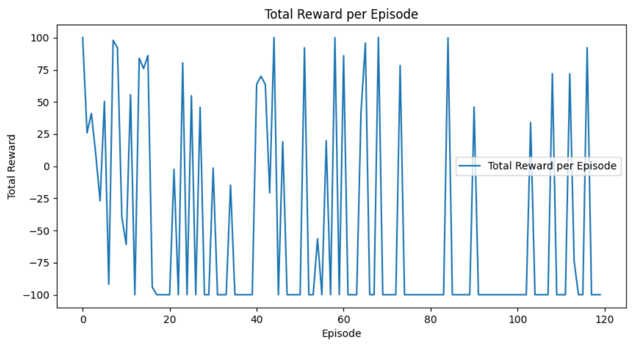

Figure 10: Total reward per episode using the dueling DQN with normalization to the convolutional and the linear layers

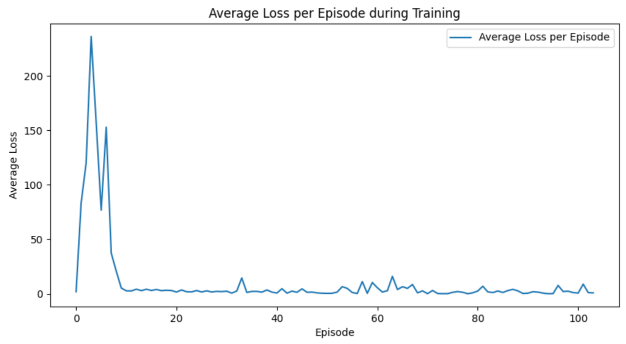

Figure 11: Average loss per episode using the dueling DQN with no normalization

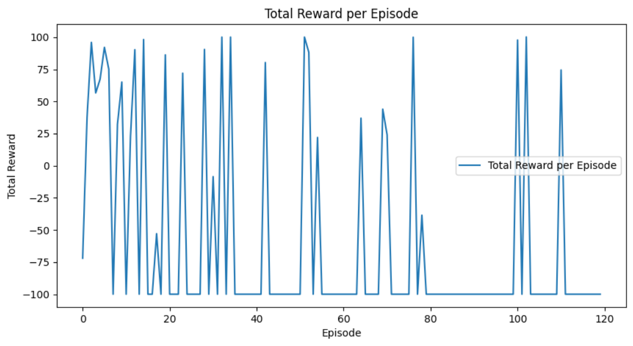

Figure 12: Total reward episode using the dueling DQN with no normalization

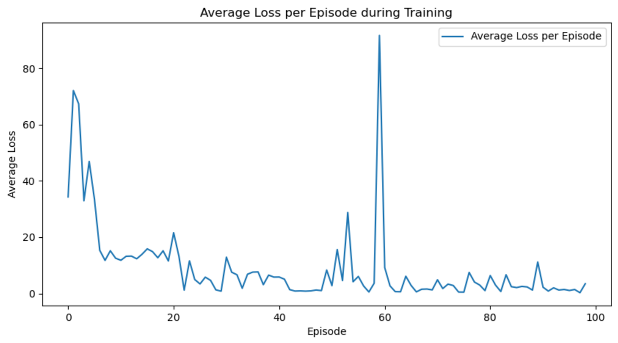

Figure 13: Average loss per episode using the inception DQN with normalization to the convolutional only

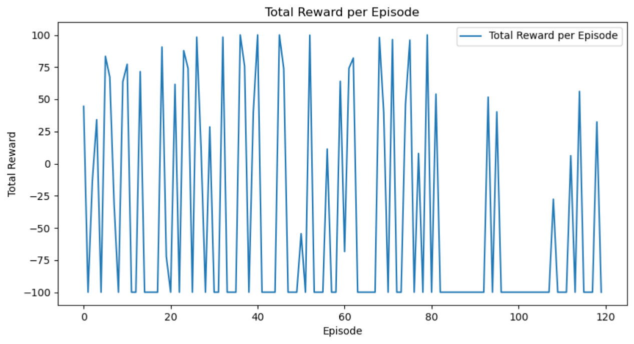

Figure 14: Total reward per episode using the inception DQN with normalization to the convolutional only

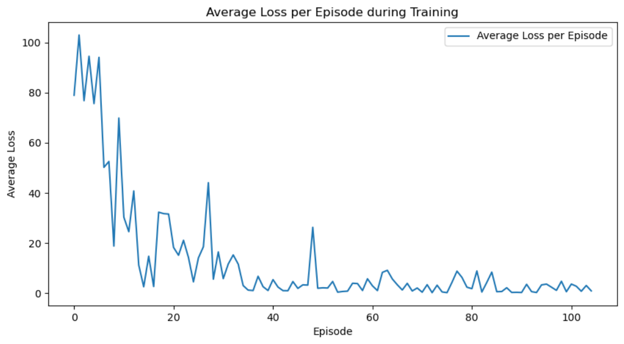

Figure 15: Average loss per episode using the inception DQN with normalization to the convolutional and the linear layers

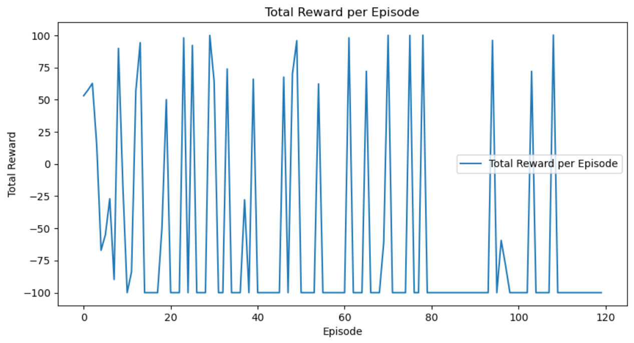

Figure 16: Total reward per episode using the inception DQN with normalization to the convolutional and the linear layers

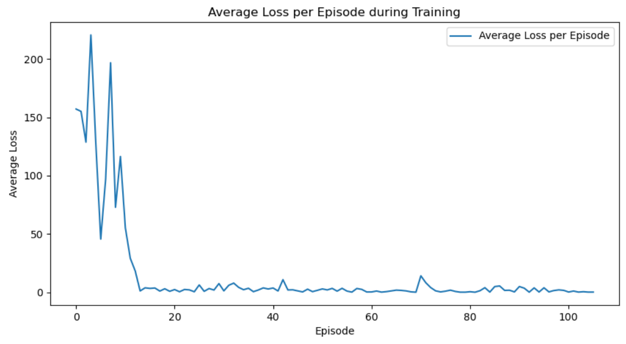

Figure 17: Average loss per episode using the inception DQN with no normalization

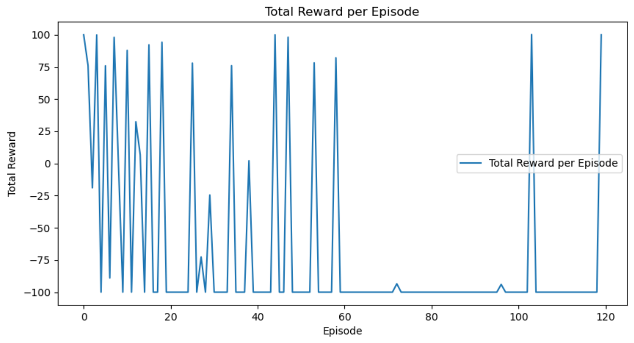

Figure 18: Total reward per episode using the inception DQN with no normalization

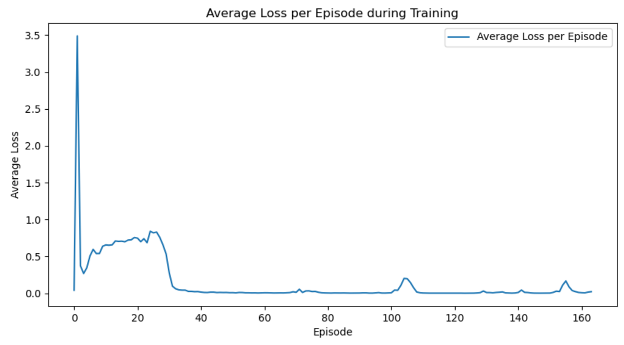

Figure 19: Average loss per episode using the custom DQN with no normalization. This case actual solve the maze but the player and the treasure have to be to the same starting positions always and after some experiments with different starting points it is obvious that it just learn one route do that every time

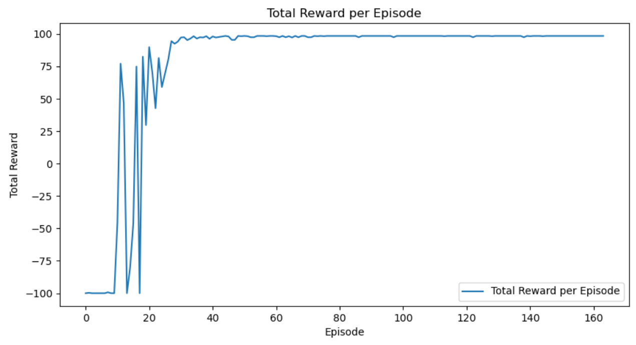

Figure 20: Total reward per episode using the custom DQN with no normalization. This case actual solve the maze but the player and the treasure have to be to the same starting positions always and after some experiments with different starting points it is obvious that it just learn one route do that every time

1.11 Conclusion

- The RL agents I used seems to learn the route only when the starting positions of the player and the treasure are not changing.
- The code seems to run fine, but maybe we need to try different hyperparameters in order to achieve the desired result.
- In general the average loss per episode converges to around zero, but the implementation seems to still not work.
- The agent can learn the maze if everything is set and nothing changes, this experiment happened to a 8x8 maze and a 10x10 maze and it completed successfully, but everything is set in these kinds of environments.

2 Fuzzy Logic

In this section we will give the answers to five different exercises from chapters 2 and 3.

2.1 Exercise 2.2

Given two fuzzy sets defined by their membership functions $\mu_A(x)$ and $\mu_B(x)$:

$$\mu_A(x) = \begin{cases} \frac{1}{1+(x-15)^{-2}} & \text{if } x > 15 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

$$\mu_B(x) = \frac{1}{1+(x-17)^4} \tag{6}$$

Question (a): Find and express the membership function of the fuzzy set

$C =$ "x greater than 15 AND x around 17" using various AND operators.

Minimum Operator

$$C_{\min}(x) = \min(\mu_A(x), \mu_B(x)) \tag{7}$$

Algebraic Product

$$C_{\text{prod}}(x) = \mu_A(x) \cdot \mu_B(x) \tag{8}$$

Bounded Difference

$$C_{\text{bounded}}(x) = \max(0, \mu_A(x) + \mu_B(x) - 1) \tag{9}$$

Question (b): Find and express the membership function of the fuzzy set

$D =$ "x greater than 15 OR x around 17" using various OR operators.

Maximum Operator

$$D_{\max}(x) = \max(\mu_A(x), \mu_B(x)) \tag{10}$$

Probabilistic Sum

$$D_{\text{prob}}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x) \cdot \mu_B(x) \tag{11}$$

Bounded Sum

$$D_{\text{bounded}}(x) = \min(1, \mu_A(x) + \mu_B(x)) \tag{12}$$

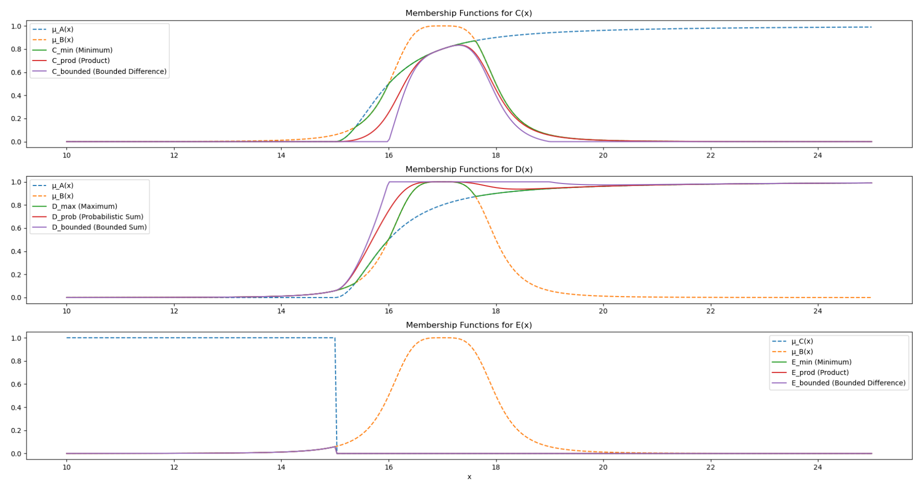

Figure 21: Membership functions

Question (c): Find and express the membership function of the fuzzy set

$E =$ "x not greater than 15 AND x around 17" with all AND operators.

Given the conditions, this set should theoretically be empty since $x \leq 15$ and $x \approx 17$ are mutually exclusive. However, using standard fuzzy logic operators, we still perform the calculations:

Minimum Operator

$$E_{\text{min}}(x) = \min(\mu_C(x), \mu_B(x)) \tag{13}$$

Algebraic Product

$$E_{\text{prod}}(x) = \mu_C(x) \cdot \mu_B(x) \tag{14}$$

Bounded Difference

$$E_{\text{bounded}}(x) = \max(0, \mu_C(x) + \mu_B(x) - 1) \tag{15}$$

where

$$\mu_C(x) = \begin{cases} 1 & \text{if } x < 15 \\ 0 & \text{otherwise} \end{cases} \tag{16}$$

2.2 Exercise 2.4

Given the fuzzy sets:

$$A = \{(6, 0.33), (7, 0.67), (8, 1), (9, 0.67), (10, 0.33)\}$$

$$B = \{(3, 0.20), (4, 0.60), (5, 1), (6, 0.60), (7, 0.20)\}$$

Complements

The complements of fuzzy sets A and B are calculated as follows:

$$\mu_{A^c}(x) = 1 - \mu_A(x), \quad \mu_{B^c}(x) = 1 - \mu_B(x) \tag{17}$$

Resulting in:

$$A^c = \{(6, 0.67), (7, 0.33), (8, 0), (9, 0.33), (10, 0.67)\} \tag{18}$$

$$B^c = \{(3, 0.80), (4, 0.40), (5, 0), (6, 0.40), (7, 0.80)\} \tag{19}$$

Union

The union of A and B is computed using the maximum membership values:

$$\mu_{A\cup B}(x) = \max(\mu_A(x), \mu_B(x)) \tag{20}$$

Thus:

$$A \cup B = \{(3, 0.20), (4, 0.60), (5, 1), (6, 0.60), (7, 0.67), (8, 1), (9, 0.67), (10, 0.33)\} \tag{21}$$

Intersection

The intersection of A and B using minimum values:

$$\mu_{A\cap B}(x) = \min(\mu_A(x), \mu_B(x)) \tag{22}$$

Thus:

$$A \cap B = \{(6, 0.33), (7, 0.20)\} \tag{23}$$

Max-Product Intersection

The intersection of A and B using the product of membership values:

$$\mu_{A\cap B}(x) = \mu_A(x) \cdot \mu_B(x) \tag{24}$$

Thus:

$$A \cap B = \{(6, 0.198), (7, 0.134)\} \tag{25}$$

Union with Bounded Difference

The union of A and B using Bounded Difference is:

$$\mu_{A\cup B}(x) = \min(1, \mu_A(x) + \mu_B(x)) \tag{26}$$

Thus:

$$A \cup B = \{(3, 0.20), (4, 0.60), (5, 1), (6, 0.93), (7, 0.87), (8, 1), (9, 0.67), (10, 0.33)\} \tag{27}$$

Intersection with Bounded Difference

The intersection of A and B using Bounded Difference is:

$$\mu_{A\cap B}(x) = \max(0, \mu_A(x) + \mu_B(x) - 1) \tag{28}$$

Thus:

$$A \cap B = \{(6, 0.198), (7, 0.134)\} \tag{29}$$

2.3 Exercise 2.11

Find the "max-min", "max-product", and "max-average" compositions of the following fuzzy relations R_1 and R_2.

$$R_1 = \begin{array}{c|ccccc} & y_1 & y_2 & y_3 & y_4 & y_5 \\ \hline x_1 & 0.1 & 0.2 & 0.0 & 1.0 & 0.7 \\ x_2 & 0.3 & 0.5 & 0.0 & 0.2 & 1.0 \\ x_3 & 0.8 & 0.0 & 1.0 & 0.4 & 0.3 \end{array}$$

$$R_2 = \begin{array}{c|cccc} & z_1 & z_2 & z_3 & z_4 \\ \hline y_1 & 0.9 & 0.0 & 0.3 & 0.4 \\ y_2 & 0.2 & 1.0 & 0.8 & 0.0 \\ y_3 & 0.8 & 0.0 & 0.7 & 1.0 \\ y_4 & 0.4 & 0.2 & 0.3 & 0.4 \\ y_5 & 0.0 & 1.0 & 0.0 & 0.8 \end{array}$$

Max-min Composition

The "max-min" composition of R_1 and R_2 is defined as:

$$\mu_{R_1 \circ R_2}(x, z) = \max_y \min(\mu_{R_1}(x, y), \mu_{R_2}(y, z)) \tag{30}$$

For example, for x_1 and z_1:

$$\mu_{R_1 \circ R_2}(x_1, z_1) = \max(\min(0.1, 0.9), \min(0.2, 0.2), \min(0, 0.8), \min(1.0, 0.4), \min(0.7, 0.0)) = 0.4 \tag{31}$$

	z_1	z_2	z_3	z_4
x_1	0.4	0.7	0.3	0.7
x_2	0.3	1.0	0.5	0.8
x_3	0.8	0.3	0.7	1.0

Max-product Composition

The "max-product" composition of R_1 and R_2 is defined as:

$$\mu_{R_1 \circ R_2}(x, z) = \max_y(\mu_{R_1}(x, y) \cdot \mu_{R_2}(y, z)) \tag{32}$$

For example, for x_1 and z_1:

$$\mu_{R_1 \circ R_2}(x_1, z_1) = \max(0.1 \cdot 0.9, 0.2 \cdot 0.2, 0 \cdot 0.8, 1.0 \cdot 0.4, 0.7 \cdot 0.0) = 0.4 \tag{33}$$

	z_1	z_2	z_3	z_4
x_1	0.4	0.7	0.3	0.56
x_2	0.27	1.0	0.4	0.8
x_3	0.8	0.3	0.7	1.0

Max-average Composition

The "max-average" composition of R_1 and R_2 is defined as:

$$\mu_{R_1 \oplus R_2}(x, z) = \max_y \frac{1}{2}(\mu_{R_1}(x, y) + \mu_{R_2}(y, z)) \tag{34}$$

For example, for x_1 and z_1:

$$\mu_{R_1 \oplus R_2}(x_1, z_1) = \max(\frac{1}{2}(0.1+0.9), \frac{1}{2}(0.2+0.2), \frac{1}{2}(0+0.8), \frac{1}{2}(1.0+0.4), \frac{1}{2}(0.7+0.0)) = 0.7 \tag{35}$$

	z_1	z_2	z_3	z_4
x_1	0.7	0.85	0.65	0.75
x_2	0.6	1.0	0.65	0.9
x_3	0.9	0.65	0.85	1.0

2.4 Exercise 3.4

A Takagi-Sugeno fuzzy model with four rules is described as follows:

- **K1:** If x is small and y is small, then $z = -x + y + 1$.
- **K2:** If x is small and y is large, then $z = -y + 3$.
- **K3:** If x is large and y is small, then $z = -x + 3$.
- **K4:** If x is large and y is large, then $z = x + y + 2$.

The membership functions for ‘small‘ and ‘large‘ conditions for variables x and y follow a sigmoidal form. The system’s output, z, depends on the aggregation of these rules.

The rules provided are:

1. If x is small and y is small, then $z = -x + y + 1$.
2. If x is small and y is large, then $z = -y + 3$.
3. If x is large and y is small, then $z = -x + 3$.

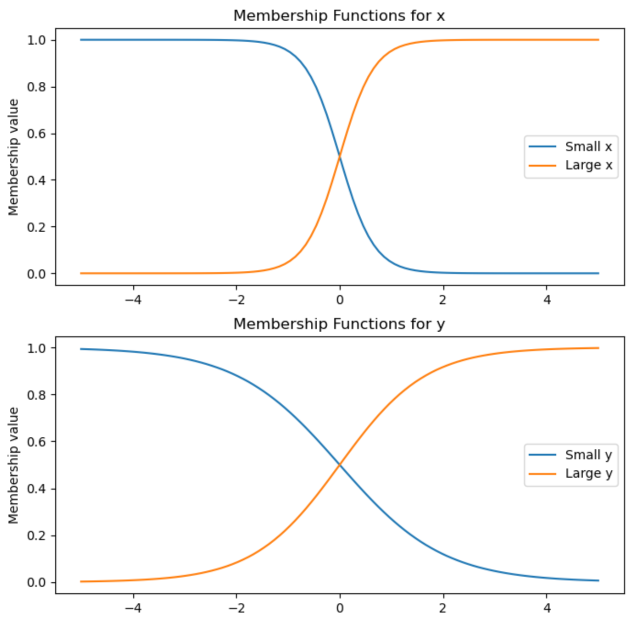

Figure 22: Membership Functions for x and y

4. If x is large and y is large, then $z = x + y + 2$.

The membership functions for "small" and "large" values of x and y are typically sigmoid or Gaussian functions. In this case, it looks like sigmoid functions based on the graph:

- For x and y, "small" has an S-shaped curve increasing as values move away from zero negatively, and "large" has an S-shaped curve increasing as values move away from zero positively.

Rule Application

When applying the rules:

$$z = \frac{\sum_{i=1}^{4} w_i \cdot z_i(x, y)}{\sum_{i=1}^{4} w_i} \tag{36}$$

Where w_i is the weight from the membership functions for each rule i, and $z_i(x, y)$ is the output from the ith rule.

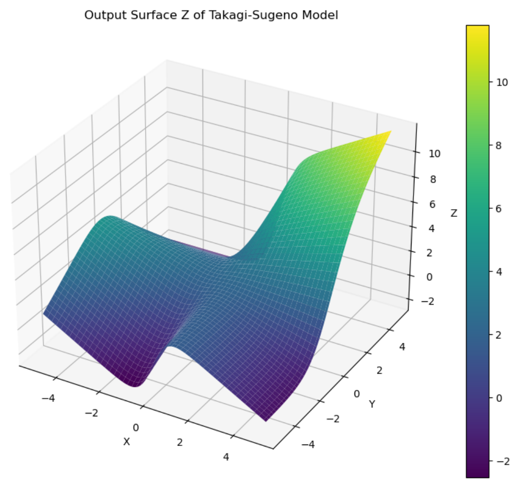

Figure 23: Output surface z

Exercise 3.5

Update the exercise 3.4 with the following membership function specifications:

$$\mu_{\text{small}}(x) = \text{sigmoid}(x, [-5, 0]) \tag{37}$$

$$\mu_{\text{large}}(x) = \text{sigmoid}(x, [5, 0]) \tag{38}$$

$$\mu_{\text{small}}(y) = \text{sigmoid}(y, [-2, 0]) \tag{39}$$

$$\mu_{\text{large}}(y) = \text{sigmoid}(y, [2, 0]) \tag{40}$$

These functions modify how inputs x and y are fuzzified, influencing the output z.

Mathematical Representation of the Sigmoid Function

The sigmoid function generally used in these contexts is defined as:

$$\sigma(v; a, b) = \frac{1}{1 + e^{-b(v-a)}} \tag{41}$$

where a is the center of the sigmoid, shifting it along the variable axis, and b controls the steepness or spread of the curve.

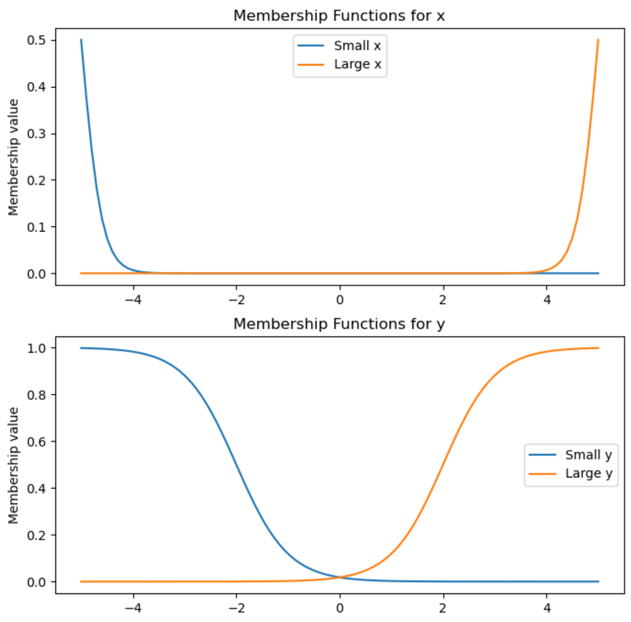

Figure 24: Membership Functions for x and y

Rule Application

When applying the rules:

$$z = \frac{\sum_{i=1}^{4} w_i \cdot z_i(x, y)}{\sum_{i=1}^{4} w_i} \tag{42}$$

Where w_i is the weight from the membership functions for each rule i, and $z_i(x, y)$ is the output from the ith rule.

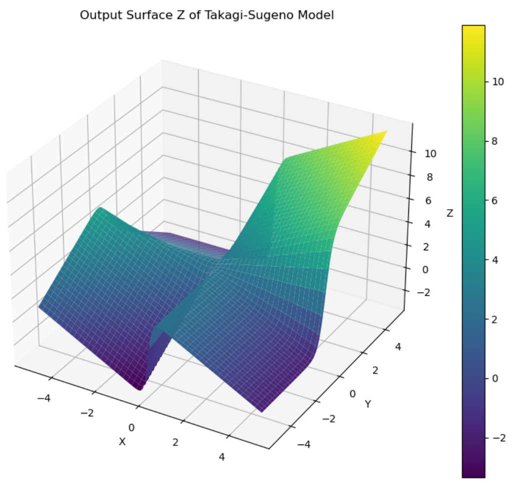

Figure 25: Output surface z

3 Genetic Algorithms

We will see the results of two genetic algorithms from the exercises of chapter 9.

3.1 Exercise 9.1

Implement a basic genetic algorithm using Python with the following characteristics:

- Parent selection proportional to fitness, using roulette wheel sampling.
- Population size $n = 100$.
- Single-point crossover with a probability of $P_c = 0.7$.
- Bit mutation with a probability of $P_m = 0.001$.
- Fitness function $f(x)$ equals the count of '1's in x, where x is a chromosome of length $l = 100$.

Execute 20 runs and measure the average generation in which the sequence consisting entirely of '1's is found. Repeat the same with zero crossover ($P_c = 0$). Execute similar runs varying the crossover rate to determine how these variables affect the average time needed by the genetic algorithm to find the

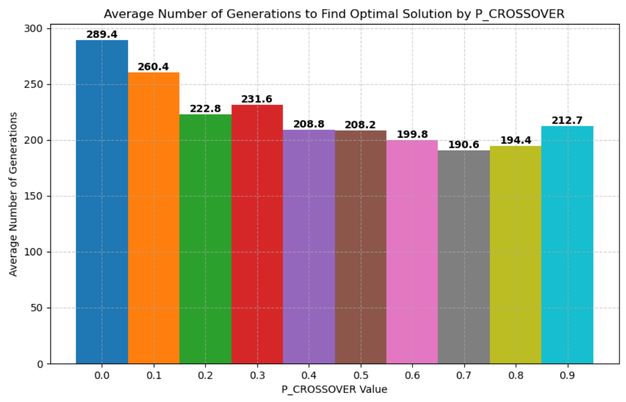

Figure 26: Average Number of Generations to Find Optimal Solution by P_CROSSOVER with changing mutation rate

optimal sequence. If the conclusion is that mutation with crossover performs better than mutation alone, explain why this happens.

We implement the genetic algorithm using python and we did two implementations, one we had the mutation rate constant and the other implementation is with changing mutation rate. The value of the mutation rate depends on the fitness of the chromosome. Low fitness means high mutation rate and high fitness means low mutation rate per chromosome.

It is important to mention that the genetic algorithm as it was given from the exercise need too many generations to converge to a good solution, so we use elitism. Practically, using elitism we take the four best chromosomes base on fitness score and we make sure that these four with have probability to be parents for the next generation equal to one. By doing this we have improved the genetic algorithm and the time it needs to complete its task.

By running the python code we get the plots from Figure 26 to Figure 29. We can see that when we use the crossover the genetic algorithm performs better and we can see that the best value for the probability of a crossover is between 0.7 and 0.8.

- **Exploration and Exploitation Balance:** Crossover is primarily responsible for exploitation, combining bits of parent solutions in new ways, potentially finding better solutions by exploring the space around the existing good solutions. Mutation, on the other hand, introduces random changes to individuals, helping to explore new areas of the solution space that may not be reachable by crossover alone.

- **Diversity Maintenance:** Crossover helps maintain and introduce diversity in the population by creating new individuals that are combinations of two parents, thus preserving and mixing genetic information across gen-

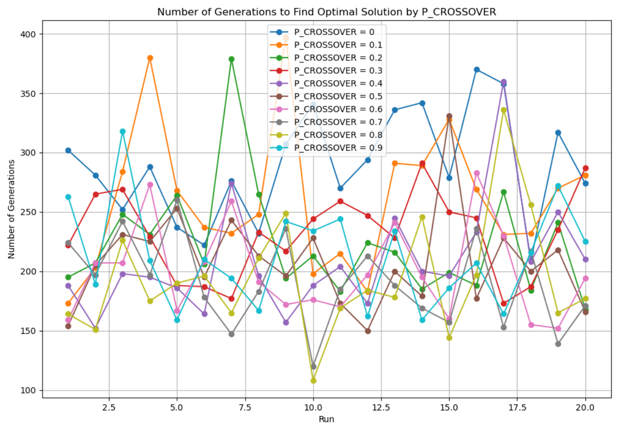

Figure 27: Number of Generations to Find Optimal Solution by P_CROSSOVER with changing mutation rate

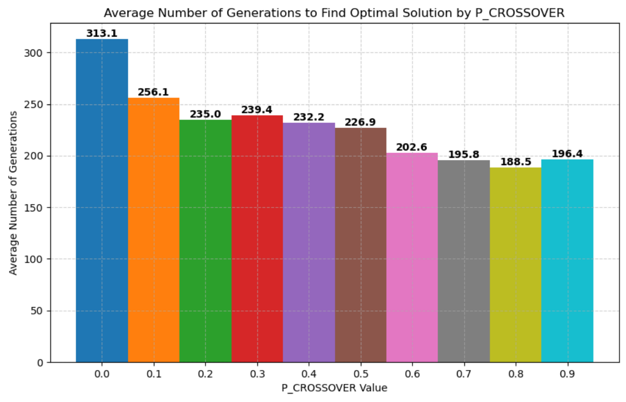

Figure 28: Average Number of Generations to Find Optimal Solution by P_CROSSOVER with constant mutation rate

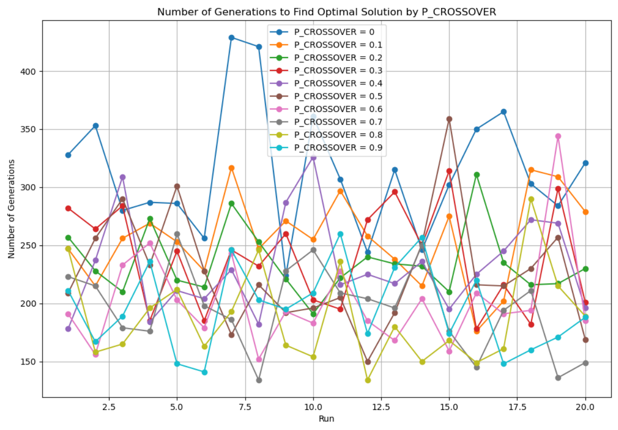

Figure 29: Number of Generations to Find Optimal Solution by P_CROSSOVER with constant mutation rate

erations. Mutation also contributes to diversity but in a more random and less structured way by altering individual genes.

- **Faster Convergence:** Crossover can quickly combine beneficial traits from different individuals to create potentially superior offspring. Mutation can then fine-tune these individuals, adjusting small parts of the solution that might not be otherwise optimized through crossover alone.

3.2 Exercise 9.2

Implement a basic genetic algorithm (GA) with the following characteristics:

- Selection - Reproduction mechanism: Discrete selection with a crossover probability of $P_c = 0.7$.
- Mutation of genes with $P_m = 0.001$.
- The adaptation function is $f(x) = a-$number of 1's in the binary sequence x (in the natural numbers, for example, 16-8-4-2-1) where x is a chromosome of length $l = 100$.

Run the genetic algorithm for 100 generations and identify the best sequence found in each generation. Furthermore, track how the chosen sequences evolve with the change of generations through selection and mutation rate. What happens if mutation is used alone ($P_c = 0$)?

We implement a genetic algorithm almost like the one we have described in exercise 9.1, but this time the main goal was to run many experiments and watch the change in the performance based on the change of the of some og

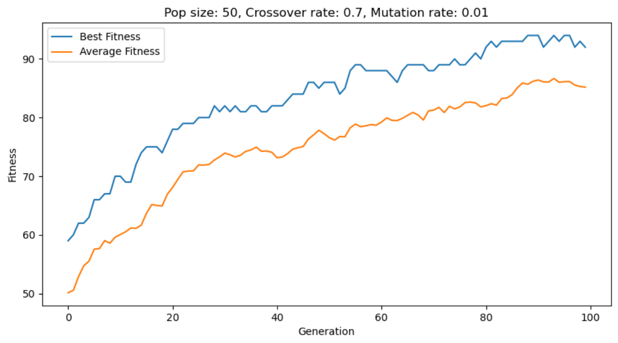

Figure 30: Best and average fitness per generation for specific algorithm parameters

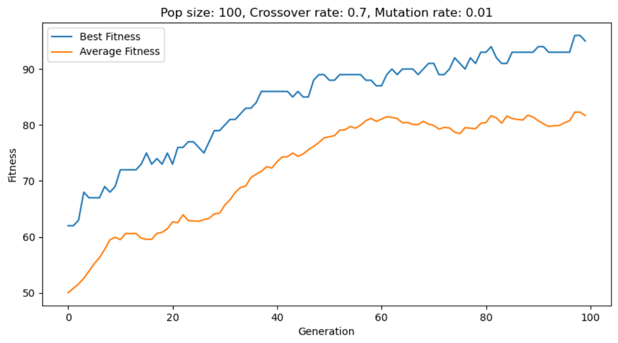

Figure 31: Best and average fitness per generation for specific algorithm parameters

the algorithm's parameters. In this implementation we continue using elitism of 4 chromosomes per generation, which seems to improve the quality of the results. We plot one figure per combination of parameters and the plots have the best fitness of every generation and the average of every generation for 100 generations. All the plots produced is available in the project zip file, here we will see some best case figures.

Population size plots

We provide the best cases for everyone of the different number of population size we used, from Figure 30 to Figure 34.

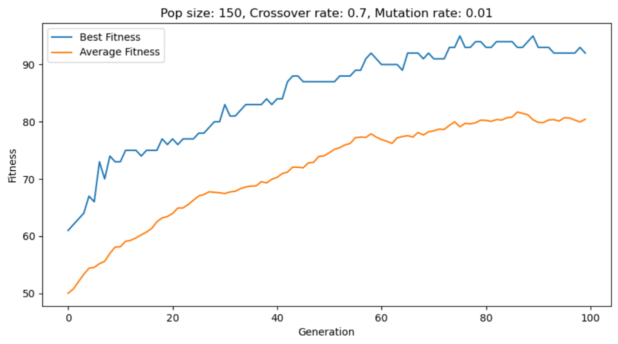

Figure 32: Best and average fitness per generation for specific algorithm parameters

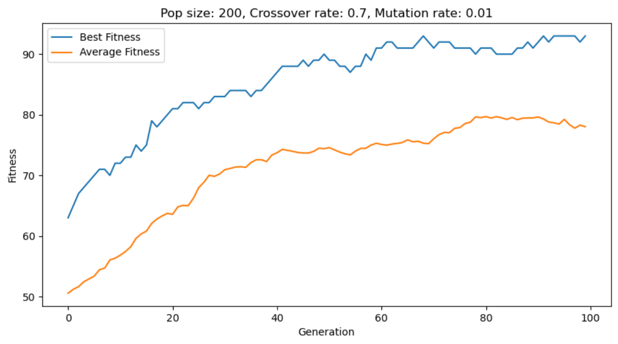

Figure 33: Best and average fitness per generation for specific algorithm parameters

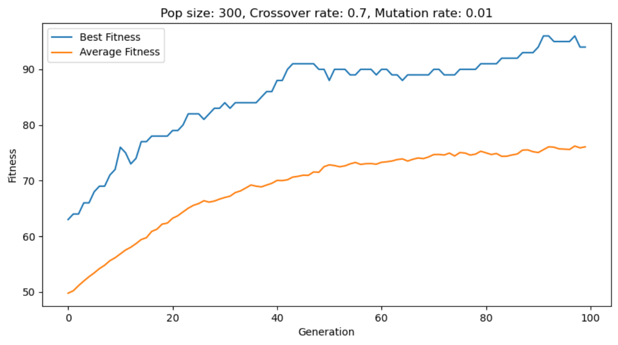

Figure 34: Best and average fitness per generation for specific algorithm parameters

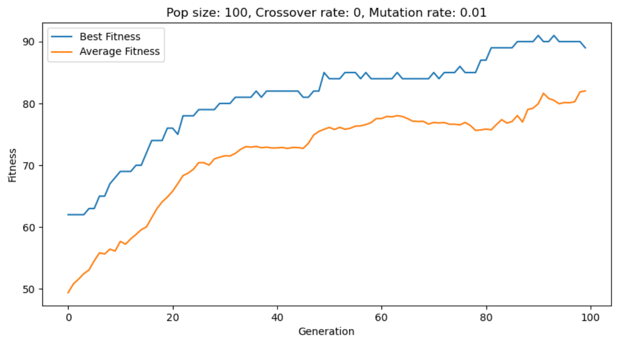

Figure 35: Best and average fitness per generation for specific algorithm parameters

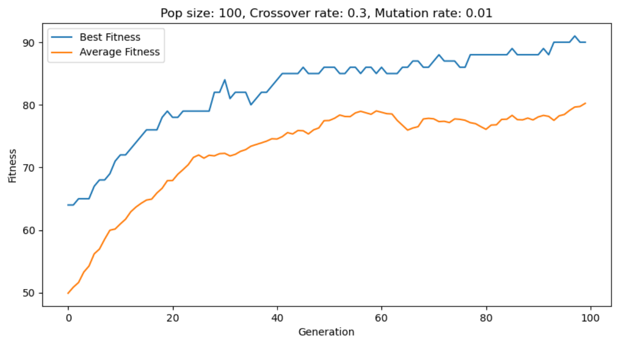

Figure 36: Best and average fitness per generation for specific algorithm parameters

Crossover rate plots

Below we have the best cases for everyone of the different crossover rates we used, from Figure 35 to Figure 39.

Mutation rate plots

Below we have the best cases for everyone of the different crossover rates we used, from Figure 40 to Figure 43.

Results

Let's sum up the results of these plots

- **Population size:** we can see that the higher the population goes the best results we get but after a threshold of 150 chromosomes in a population the

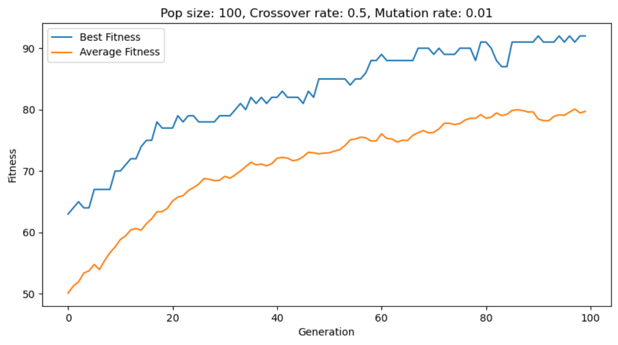

Figure 37: Best and average fitness per generation for specific algorithm parameters

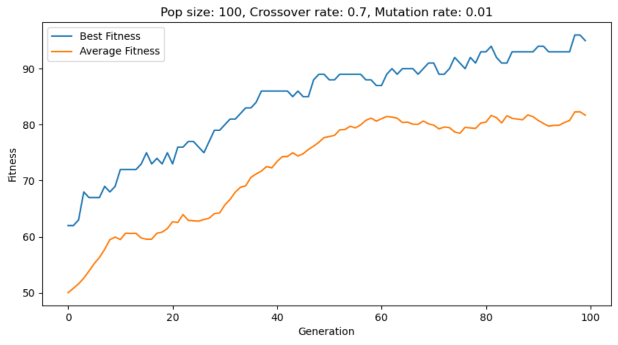

Figure 38: Best and average fitness per generation for specific algorithm parameters

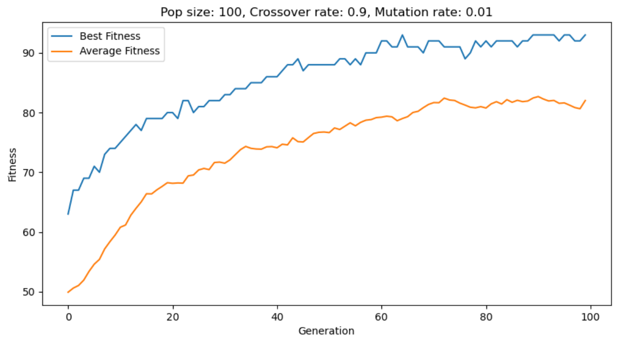

Figure 39: Best and average fitness per generation for specific algorithm parameters

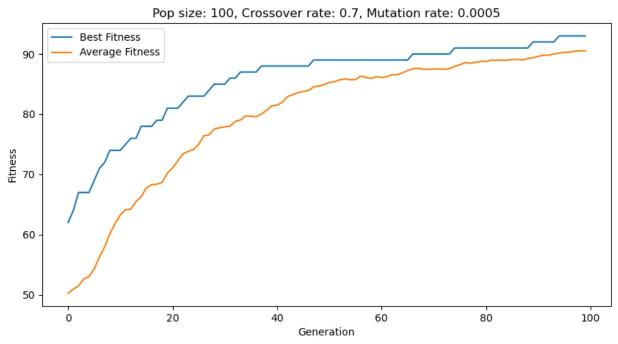

Figure 40: Best and average fitness per generation for specific algorithm parameters

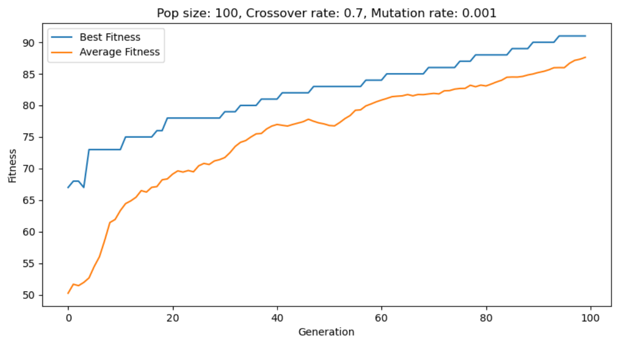

Figure 41: Best and average fitness per generation for specific algorithm parameters

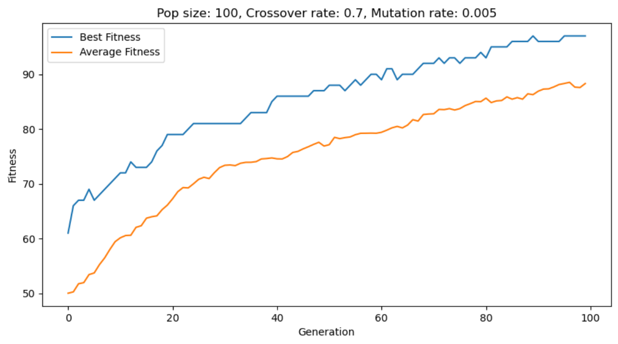

Figure 42: Best and average fitness per generation for specific algorithm parameters

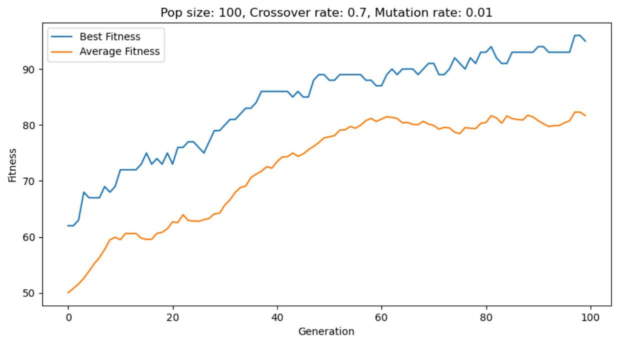

Figure 43: Best and average fitness per generation for specific algorithm parameters

results have minor changes. Also we can see that the lower the population size is the closer the best fitness and the average fitness are, which is a logical conclusion too, because the more chromosomes you have, your population has more and more diversity.

- **Crossover rate:** we can see that when the crossover rate is low, the results is based on the randomness of the mutation, so we use more exploration tactics than exploitation tactics. Furthermore, when crossover rate is 0 we solve the problem based on pure exploration, we don't use exploitation. When the crossover rate is high, we use more exploitation tactics than exploration tactics, this isn't always optimal because we need exploration too, that's why a crossover rate around 0.7 is the best case why get most of the times.

- **Mutation rate:** as described above the mutation rate is our main tactic for exploration, this means that when the mutation rate is high we explore main scenarios in few generations but due to this high mutation rate we might destroy some good chromosomes so we may reduce the best fitness, that's why the best fitness and average fitness is not that close in the plots. On the other hand when we use low mutation rate we get low exploration capability, so the explore phase may take more generation to find good chromosomes, but at the same time you lower the possibility to rune a good chromosome, also the best fitness and the average fitness is close so you have better chromosomes to crossover.