Computational Intelligence Course Log 2024

Week 1 (Sep 23-27)

- **Start Date:** September 23, 2024
- **End Date:** September 27, 2024
- **Lessons:**
  - 23/09: Intro! - Fine innate or empiric, Chomsky
  - 25/09: What-if scenarios, Artificial life
  - 26/09: Regular Lesson
- **Topics Covered:**
  - Course Introduction
  - Innate or Empiric Theories
  - Chomsky's Theories
  - What-if Scenarios
  - Artificial Life
- **Lab Work:**
  - Lab 0 Assignment

Week 2 (Sep 30-Oct 4)

- **Start Date:** September 30, 2024
- **End Date:** October 4, 2024
- **Lessons:**
  - 30/09: Regular Lesson
  - 02/10: Lab 1 Session
  - 03/10: Regular Lesson
- **Topics Covered:**
  - Introduction to Practical Work
  - Lab 1 Implementation
- **Lab Work:**
  - Lab 1 Assignment
  - Lab 0 Review

Week 3 (Oct 7-11)

- **Start Date:** October 7, 2024
- **End Date:** October 11, 2024
- **Lessons:**
  - 07/10: Regular Lesson
  - 10/10: Regular Lesson
- **Topics Covered:**

    - Regular Topics

Week 4 (Oct 14-18)

- **Start Date:** October 14, 2024
- **End Date:** October 18, 2024
- **Lessons:**
    - 14/10: Regular Lesson
    - 16/10: Regular Lesson
    - 17/10: Regular Lesson
    - Lab 1 review
- **Topics Covered:**
    - Regular Topics

Week 5 (Oct 21-25)

- **Start Date:** October 21, 2024
- **End Date:** October 25, 2024
- **Lessons:**
    - 21/10: Regular Lesson
    - 23/10: Regular Lesson
    - 24/10: Regular Lesson
- **Topics Covered:**
    - Regular Topics

Week 6 (Oct 28-Nov 1)

- **Start Date:** October 28, 2024
- **End Date:** November 1, 2024
- **Lessons:**
    - 28/10: Regular Lesson
    - 31/10: Regular Lesson
- **Topics Covered:**
    - Regular Topics
- **Lab's work**
    - Lab 2 Assignment

Week 7 (Nov 4-8)

- **Start Date:** November 4, 2024
- **End Date:** November 8, 2024
- **Lessons:**
    - 04/11: Regular Lesson
    - 06/11: Regular Lesson

    - 07/11: Regular Lesson
- **Topics Covered:**
    - Regular Topics

Week 8 (Nov 11-15)

- **Start Date:** November 11, 2024
- **End Date:** November 15, 2024
- **Lessons:**
    - 11/11: Regular Lesson
    - 14/11: Regular Lesson
- **Topics Covered:**
    - Regular Topics

Week 9 (Nov 18-22)

- **Start Date:** November 18, 2024
- **End Date:** November 22, 2024
- **Lessons:**
    - 18/11: Regular Lesson
    - 21/11: Regular Lesson
- **Topics Covered:**
    - Regular Topics
- **Lab's work**
    - Lab 3 Assignment

Week 10 (Nov 25-29)

- **Start Date:** November 25, 2024
- **End Date:** November 29, 2024
- **Lessons:**
    - 25/11: Regular Lesson
    - 27/11: Project Session
    - 28/11: Regular Lesson

Week 11 (Dec 2-6)

- **Start Date:** December 2, 2024
- **End Date:** December 6, 2024
- **Lessons:**
    - 02/12: Regular Lesson
- **Topics Covered:**
    - Regular Topics

- **Lab's work**
  - Lab 3 review
  - project_assignment

Lab_0_assignment

Lab_0_review

Lab_1_assignment

Algorithm Performance Analysis

Definitions

- **Data Set Size**: The number of elements in the universe.
- **Evaluation Count**: The number of iterations performed by the algorithm.
- **Cost**: The total cost of the selected sets.
- **Solution Length**: The number of sets in the final solution.
- **Number of Sets**: The number of available sets.

Results

Data Set Size	Evaluation Count	Cost	Solution Length	Number of Sets
100	10	276.3830031	9	30
1,000	100	5,868.58844	17	497
10,000	1,000	107,812.5503	16	4,997
100,000	10,000	1,518,872.434	60	49,997
100,000	10,000	1,734,564.598	32	89,997
100,000	10,000	1,774,409.13	21	49,997
100,000	10,000	1,734,564.598	32	89,997
100,000	10,000	1,774,409.13	21	49,997

Observations

1. **Cost and Data Set Size**:
   - The cost of the solution generally increases with the data set size, as expected.
2. **Solution Length and Data Set Size**:

- The solution length (number of sets in the solution) tends to increase with the data set size.

3. **Evaluation Count and Performance**:
    - The performance of the algorithm, in terms of cost and solution length, may improve with a higher evaluation count.
4. **Large Data Sets**:
    - For very large data sets (e.g., 100,000 elements), the algorithm's performance can vary depending on the specific structure of the data set. Computational expense increases significantly for these sizes.

Lab_1_review

Review 1

Feedback

1. **Strengths**:
    - Praised for its Object-Oriented Programming (OOP) approach.
2. **Areas for Improvement**:
    - **Limited Neighborhood Exploration**: The algorithm restricts neighborhood exploration to solutions obtained by flipping only one set, risking stagnation in local optima.
    - **Steepest Ascent Bias**: The tweaking function always picks the best solution, biasing the search toward local optima.
    - **Lack of Visualization**: The absence of plots makes it difficult to evaluate the algorithm's progress over time.
3. **Proposed Solutions**:
    - **Probabilistic Tabu**: Assign a decreasing probability of acceptance for revisits to tabu solutions instead of outright forbidding them.
    - **Graph-Based Neighborhood**: Implement recursive neighborhood exploration:
        - **Depth-Limited Search**: Restrict recursive exploration to a predefined depth to control computational cost.
        - **Bridge-Based Approach**: Use "bridges" between non-tabu regions for efficient extended neighborhood exploration.

Modifications Implemented

1. **Probabilistic Tabu**:
    - **Implementation**:
        - Replaced the binary tabu list with a dictionary (`tabu_list`) to track visit counts for solutions.

        - Adjusted the `is_tabu` function to calculate the acceptance probability as $\text{probability} = \frac{1}{\text{visits} + 1}$.
        - A solution is accepted only if a randomly generated number exceeds the calculated probability.
    - **Outcome**: The likelihood of revisiting a solution decreases with repeated visits, reducing the chance of stagnation.
2. **Steepest Ascent Bias**:
    - Introduced probabilistic neighbor selection:
        - With probability P_0, the best neighbor is chosen.
        - With probability $1 - P_0$, a random neighbor is selected.
    - **Benefit**: Introduced randomness into the search, helping to escape local optima.
3. **Visualization**:
    - Added plots to track:
        - **Cost**: Monitored to observe optimization progress.
        - **Solution Length**: Tracked to evaluate the number of sets in the solution over time.
    - **Impact**: Enhanced transparency and understanding of the algorithm's behavior and convergence.
4. **Further Considerations**:
    - The **Graph-Based Neighborhood** approach remains a promising avenue for future work but was not implemented due to its complexity.

Review 2

Feedback

1. **Strengths**:
    - The neighborhood generation and stopping criteria were commended.
2. **Suggestions**:
    - Explore **parallelization** to improve performance on large datasets.

Modifications Implemented

1. **Parallelization**:
    - **Status**: Considered but not implemented due to the significant code restructuring required.
    - **Future Plan**:
        - **Profiling**: Identify bottlenecks in the code to determine the best parts for parallelization (e.g., neighborhood generation, solution evaluation).
        - **Implementation Tools**: Explore Python libraries like `multiprocessing` or `concurrent.futures` for parallelization.

Summary of Key Improvements

1. Enhanced neighborhood exploration with **probabilistic tabu**.
2. Addressed steepest ascent bias with **probabilistic neighbor selection**.
3. Introduced **visualizations** for improved evaluation of algorithm behavior.
4. Identified **parallelization** as a potential improvement for future iterations.

Lab_2_assignment

Code Overview

1. Importing Libraries and Reading Data

- The script begins by importing essential libraries: `logging`, `itertools`, `pandas`, `numpy`, `geopy.distance`, `networkx`, and `icecream`.
- A CSV file (`cities/russia.csv`) is read into a pandas DataFrame named `CITIES`. This file contains city names, latitudes, and longitudes.

2. Distance Matrix Calculation

- A distance matrix (`DIST_MATRIX`) is constructed to store geodesic distances (in kilometers) between all pairs of cities.
- **Key Details:**
  - `geopy.distance.geodesic`: Calculates distances based on latitude and longitude.
  - `itertools.combinations`: Efficiently iterates through all unique city pairs.

3. TSP Cost Function

- The `tsp_cost` function computes the total distance of a given TSP tour (a list of city indices).
- **Assertions:**
  - The tour must start and end at the same city.
  - All cities must be visited exactly once.

4. Greedy Algorithm with 2-opt Optimization

a. Greedy Initialization

- Constructs an initial TSP tour:
  - Starts from an arbitrary city (city 0).
  - Iteratively selects the nearest unvisited city.
  - Closes the tour by returning to the starting city.
- **Logging:** Tracks each step of the greedy construction process.

b. 2-opt Optimization

- Implements the 2-opt local search algorithm:
  - Considers all non-adjacent edge pairs in the tour.
  - Checks if reversing the segment between edges reduces the tour length.
  - Performs swaps iteratively until no further improvements are found.
- **Logging:** Tracks improvements made by the 2-opt process.

5. Genetic Algorithm (GA)

The `GeneticTSP` class implements a genetic algorithm to solve the TSP.

a. Initialization

- Creates an initial population of tours:
  - One tour is generated using the greedy approach.
  - The rest are random permutations of cities.

b. Fitness Evaluation

- Uses `tsp_cost` to evaluate the fitness of each tour (lower distance is better).

c. Selection

- Ranks the population by fitness and selects the top 50% for reproduction.

d. Crossover

- Implements ordered crossover (OX) to generate offspring from parent tours.

e. Mutation

- Applies the 2-opt swap mutation with a probability defined by `mutation_rate`.

f. Evolution

- The `evolve` function runs the genetic algorithm:
  - Performs selection, crossover, and mutation for a specified number of generations.
  - Tracks the best tour found in each generation using `logging.info`.

Results and Analysis

Greedy + 2-opt

- **Initial Tour Length:** 42334.16 km (from greedy approach).
- **Optimized Tour Length:** 33602.75 km (after 2-opt).

Genetic Algorithm

- **Parameters:**
  - Generations: 20
  - Population Size: 100
  - Mutation Rate: 0.2
- **Results:**
  - Best Route Length: 1345.54 km (constant across generations).
  - Best Route: `[4, 3, 5, 6, 2, 0, 7, 1, 4]`.

Reviews and Observations

1. Greedy + 2-opt Effectiveness

- The combination significantly improves the initial solution, approaching an optimal route.

2. Mutation Dependency

- The reliance on 2-opt mutations may lead to local optima.
- Exploring other mutation operators (e.g., insertion, inversion) could enhance performance.

3. Crossover Comparison

- Comparing ordered crossover (OX) with methods like inver-over crossover might yield better results.

4. Generalizability

- Testing the algorithms on datasets for other countries could assess their adaptability to different TSP instances.
-

5. Genetic Algorithm Performance

- The GA converges quickly, indicating poor exploration of the search space:
  - **Potential Solutions:**
    - Increase population size and the number of generations.
    - Introduce more randomness in the initial population.
    - Experiment with different selection mechanisms (e.g., tournament selection).

Lab_2_review

The improvement of the first greedy approach (Greedy + 2 opt) is truly interesting and allows us to improve the solution considerably, bringing us closer to the best solution.

The mutation is cost dependent, it can lead us to a local optimum In the evolutionary algorithm using inver_over_crossover compared to Ordered_crossover can lead us to have slightly better results It would have been interesting to see the results for other countries as well

Lab_3_assignment

n^2-1 Puzzle Solver

This project implements an A* search algorithm to solve the **n^2-1 puzzle** (also known as the sliding puzzle). The puzzle is represented as a 2D numpy array, and the algorithm uses either the **Hamming distance** or **Manhattan distance** as the heuristic function.

Lab_3_review

Code Review Feedback

Strengths

Overall Observations

Your code demonstrates several strong points:

- **Well-structured:** The code is organized and clear, making it easy to follow.
- **Informative:** It includes meaningful components and avoids unnecessary clutter.

Highlights

1. **Algorithm Choice:**
    - Opting for the A* algorithm is a strong decision for this type of problem.
    - The heuristic function and cost metrics are clearly defined.
2. **Heuristic Exploration:**
    - Implementing both Manhattan and Hamming distance functions is a thoughtful approach to compare heuristics.
3. **Progress Feedback:**
    - The inclusion of progress updates every 5 seconds is excellent for tracking the algorithm's operation, especially given its potential time consumption.

Suggested Improvements

Key Areas to Address

1. **Path Length Initialization:**
    - Initializing `length_criteria` to `0` seems counterintuitive as it neglects the path cost. This could deviate from the original problem's intent. Consider initializing it with a small positive value to incorporate path length.
2. **Performance Metrics:**

    - Add information about the number of nodes/puzzles evaluated. This is critical for assessing the model's efficiency.
3. **Code Documentation:**
    - Enhance comments throughout the code for better clarity.
    - Print the initial state of the puzzle and intermediate states, in addition to the updates every 5 seconds.

Additional Suggestions

- **Readme Improvements:**
    - Clean up the README file for better readability and include reflections on the results.
- **Heuristic Refinement:**
    - Experiment with an additional heuristic. Though challenging, this could improve the algorithm's performance.

project_assignment

Instead of manually guessing the formula, we let a computer program discover it using an **evolutionary approach**, inspired by natural selection. In this process, the "fittest" individuals in a population survive, reproduce, and pass on their traits to the next generation.

The Algorithm:

Initialization

- Begin with a **population** of randomly generated mathematical expressions.
- Each expression acts as an individual, represented as a **tree structure**.
    - **Nodes**: Operators (e.g., +, -, *, sin, cos)
    - **Terminals**: Input variables or constants.

Evaluation

- Measure each expression's **fitness** by how well it predicts the output variable using the input data.
- Use **mean squared error (MSE)** as the fitness metric:
    - Lower MSE = Better fit.

Selection

- Choose the **fittest** expressions (lowest MSE) to serve as parents for the next generation.
- Use a **tournament method**:
    - Randomly select subsets of individuals and pick the best one from each subset.

Breeding

Create the next generation through:

1. **Crossover**:
    - Combine parts of two parent expressions to create an offspring.
    - Swap branches of expression trees.
2. **Mutation**:
    - Introduce random changes to an expression's genome:
        - Change an operator.
        - Replace a sub-expression.
        - Shuffle operands.

Iteration

- Repeat **Evaluation**, **Selection**, and **Breeding** for a fixed number of generations or until a satisfactory solution is found.

Methodology: Building and Evolving Expression Trees

Expression Trees

- Each formula is a tree:
    - **Internal nodes**: Operators (e.g., +, sin).
    - **Leaf nodes**: Variables (e.g., "x0", "x1") or constants.

Operators

- Define a set of operators, such as:
    - Arithmetic: +, -, *, /
    - Trigonometric: sin, cos
    - Other mathematical functions.

Constants

- Use a pool of random constant values.

Tree Generation

- Randomly generate trees up to a specified depth limit.

Mutation Operations

Introduce diversity using:

1. **Single Point Mutation**: Change one node in the tree.
2. **Sub-expression Mutation**: Replace part of the tree with a new sub-tree.
3. **Permutation Mutation**: Shuffle operands for an operator.
4. **Subtree Replacement**: Replace the entire expression with a subtree.
5. **Expansion**: Turn a terminal node into a complex sub-expression.
6. **Collapsing**: Replace a subtree with a single terminal node.

Crossover (Recombination)

- Swap subtrees between two parent trees to create a new offspring tree.

Experiments:

Parameters

- **DEPTH_LIMIT**: Maximum tree depth.
- **PROGENY_COUNT**: Number of offspring per generation.
- **COHORT_SIZE**: Population size.
- **EPOCH_COUNT**: Number of generations.
- **LITERAL_POOL_SIZE**: Number of random constants available.
- **LITERAL_SPAN**: Range of constant values.

Evolution Process

1. Initialize a population of random expressions.
2. Evaluate fitness using training data.
3. Select the best individuals as parents.
4. Create offspring through crossover and mutation.
5. Replace the old population with the new generation.

Evaluation

- Track **average training error** and **validation error** across generations.
- Use validation error to check for overfitting.
- Select the expression with the lowest training error as the best solution.
- Evaluate the best solution on the validation set for unbiased accuracy.

Results

This report analyzes the performance of a machine learning model on eight different regression problems. The evaluation metric is **Mean Squared Error (MSE)**, which measures the average squared difference between predicted and actual values. Each problem's results, analysis, and estimated time spent are detailed below.

Methodology

- **Training:** The model was trained for **200 epochs** on each problem.
- **Evaluation:** MSE was computed on a separate test set to assess the model's generalization ability.

Results Overview

Problem	Test MSE	Estimated Time (minutes)
1	0.000000000000000000000000000000000	75
2	29616986269792.97265625000000000000000000000000	92
3	18.380859195116443061124300584197 04	105
4	0.06846623301471516098715142106812	88
5	0.000000000000000005572810232617 32	65
6	0.000003943094975741775207525585 68	110
7	771.06499534112549554265569895505905	70
8	682150.56657352636102586984634399414062	98

Analysis

Problem 1

- **Function:** Complex non-linear equation with trigonometric and algebraic operations.
- **Test MSE:** `0.0`
- **Analysis:** A perfect fit due to the simplicity of the function. The model either captured the exact relationship or overfitted.
- **Estimated Time:** 75 minutes

Problem 2

- **Function:** Complex non-linear equation with trigonometric and algebraic operations.
- **Test MSE:** `29616986269792.972`
- **Analysis:** Extremely high MSE indicates a failure to learn the simple periodic relationship. Possible issues include undertraining or an unsuitable model.
- **Estimated Time:** 92 minutes

Problem 3

- **Function:** Arithmetic operations involving addition
- **Test MSE:** `18.38`
- **Analysis:** High MSE suggests the model failed to predict the constant value accurately, likely due to mismatched architecture or incorrect loss configuration.
- **Estimated Time:** 105 minutes

Problem 4

- **Function:** Complex non-linear equation with trigonometric and algebraic operations.
- **Test MSE:** `0.068`

- **Analysis:** Reasonable performance, but the model has room for improvement. Additional epochs may help.
- **Estimated Time:** 88 minutes

Problem 5

- **Function:** Nested trigonometric operations with custom functions.
- **Test MSE:** `0.00000000000000000557`
- **Analysis:** Excellent fit, suggesting the model either learned well or overfitted to the test data.
- **Estimated Time:** 65 minutes

Problem 6

- **Function:** Arithmetic operations involving addition, subtraction, multiplication, and division.
- **Test MSE:** `0.00000394`
- **Analysis:** Low MSE indicates good performance. The model likely found this function straightforward to learn.
- **Estimated Time:** 110 minutes

Problem 7

- **Function:** Arithmetic operations involving addition
- **Test MSE:** `771.065`
- **Analysis:** Poor performance for this linear function. Potential reasons include issues with the training process or dataset outliers.
- **Estimated Time:** 70 minutes

Problem 8

- **Function:** Repeated trigonometric operations with constants.
- **Test MSE:** `682150.566`
- **Analysis:** The extremely high MSE suggests the model struggled to capture the repetitive nature of the function or was severely undertrained.
- **Estimated Time:** 98 minutes

Conclusion

The model's performance varies significantly:

- **Strong Performance:** Problems 1, 5, and 6, demonstrating effective learning for certain types of functions.
- **Poor Performance:** Problems 2, 3, 7, and 8, revealing limitations in the model's architecture, training process, or ability to generalize.
