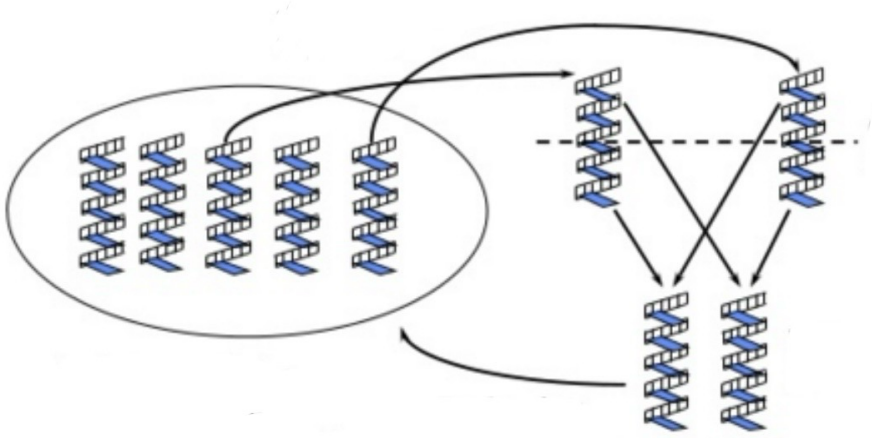
MATHEMATICS FOR MACHINE LEARNING

Genetic Algorithm



Krittaya Kruapat 麗心 411855017

Instructure: HUANG-WEN HUANG

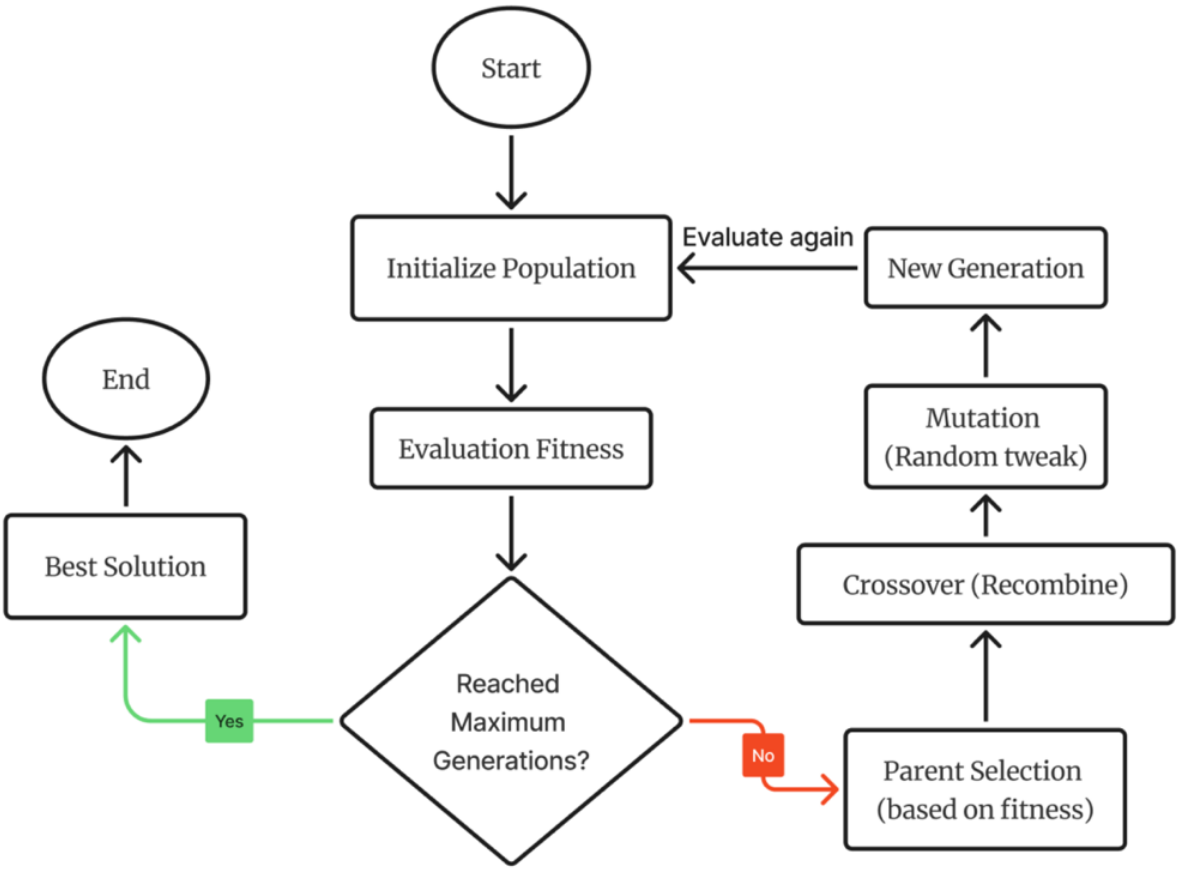
Tamkang University Academic Year 114-1

Genetic Algorithm (GA)

Overview

Genetic Algorithm(GAs) are a class of evolutionary-based optimizers that are motivated by natural selection and genetics. They are especially suitable for handling non-linear, high-dimensional or with-discontinuous objective functions. Genetic Algorithms do not rely on the gradient which are then applicable to non-differentiable and discontinuous problems, or also for the cases where mathematical optimization is infeasible. GAs use a multi-generation iterative evolution to converge to an optimal or near-optimal solution.

GAs are a category of Evolutionary Computation, such the problem solving process is viewed in evolutionary terms. The algorithm acts on a population of candidate solutions, called chromosomes (this kind of data is often a vector or binary code). Evolution narrows the search space by measuring how well each candidate performs on the optimization task at hand, a fitness function.

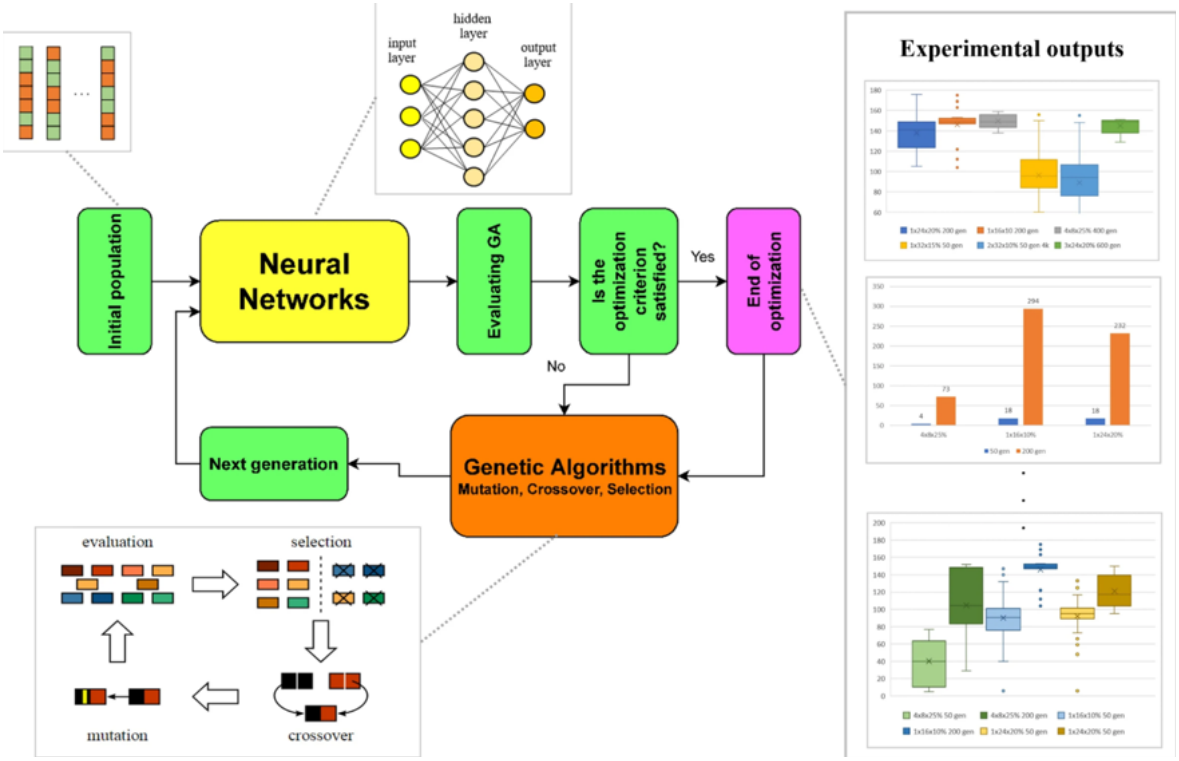


Applications

Genetic Algorithms are widely used in real-world optimization tasks, particularly when the search space is large, complex, or difficult to describe mathematically. Since GAs do not require gradient information and can evaluate many candidate solutions in parallel, they are well suited to problems where traditional optimization methods struggle or become inefficient.

1. Machine Learning Hyperparameter Tuning

Genetic Algorithms can automatically search for effective hyperparameter settings in machine learning models such as neural networks. Instead of manually testing combinations of learning rates, batch sizes, or layer configurations, each hyperparameter set is treated as a candidate solution and evaluated using validation accuracy as the fitness score. Through selection, crossover, and mutation over multiple generations, the GA gradually identifies configurations that improve model performance without exhaustive trial-and-error.



2. Scheduling and Timetable Optimization

Genetic Algorithms are widely used in scheduling problems such as university timetables, staff rosters, and resource allocation. Each possible schedule is encoded as a candidate solution, and its fitness reflects how well it avoids conflicts and satisfies constraints. By evolving solutions through selection, crossover, and mutation, the GA can efficiently produce high-quality schedules that are difficult to design manually.

Mathematics Involved

Key Concepts

1. *Optimization*

The core goal of a Genetic Algorithm is optimization, meaning the algorithm searches for solutions that maximize or minimize a fitness function. This function quantifies how "good" each candidate solution is. In machine learning, the fitness might instead be:

$$f(\theta) = validation\ accuracy\ of\ model\ M(\theta)$$

So optimization is tied to whatever equation defines "goodness."

2. *Probability*

Probability plays a central role in Genetic Algorithms. Selection, crossover, and mutation are all governed by probabilistic rules rather than deterministic ones.

2.1 *Fitness-Proportionate Selection*

A common selection strategy is roulette-wheel selection, where the probability of selecting an individual is proportional to its fitness:

$$P(i) = \frac{f(i)}{\sum_{j=1}^{N} f(j)}$$

Where: $P(i) =$ probability of selecting solution i
$f(i) =$ fitness of solution i
$N =$ population size

This approach favors stronger solutions while still allowing weaker ones to contribute, maintaining diversity.

2.2 *Population Mean Fitness*

The average fitness of the population is often used to monitor overall learning progress across generations:

$$\bar{f} = \frac{1}{N}\sum_{i=1}^{N} f(i)$$

2.3 *Fitness Variance*

Fitness variance measures how diverse the population is in terms of solution quality:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N} (f(i) - \bar{f})^2$$

High variance indicates exploration, while low variance suggests convergence.

2.3 *Mutation and Crossover Probabilities*

Mutation and crossover introduce controlled randomness:

$$P(mutation) = \mu, \;\; P(crossover) = c$$

If the chromosome length is L, the expected number of mutations per chromosome is:

$$E[\text{mutations}] = \mu \cdot L$$

3. *Representation and Search Space*

For a GA to operate, each solution must be encoded in a manipulable form such as a binary string or vector. This encoding is called the representation.

If using binary encoding of length L:

$$| \Omega | = 2^L$$

4. *Elitism*

Some GAs use elitism to guarantee that the best solution survives to the next generation:

$$x_{t+1}^* = \arg \max_{x \in G_t} f(x)$$

Core Equation

Genetic Algorithms rely on two key mathematical components: the fitness function, which evaluates how good each solution is, and the selection probability, which determines which solutions are more likely to reproduce.

Formula 1: Fitness Function

$$f(i) = fitness\ of\ individual\ i$$

This function assigns a numerical score to each candidate. Higher values mean better performance according to the problem's objective. For example, the function $f(x) = -x^2$ returns larger values when $x$is close to zero, guiding the GA toward that region.

Formula 2: Selection Probability Function

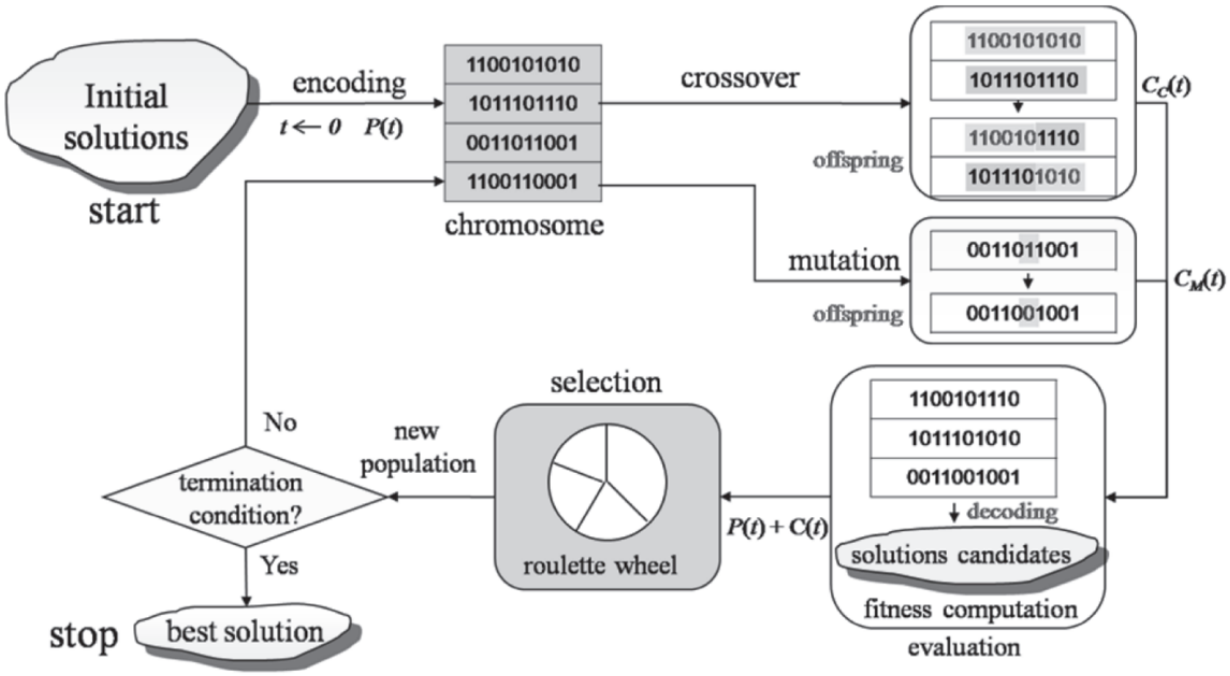
Once each solution has a fitness value, the algorithm decides which solutions get to reproduce. This decision is made using probability, not strict ranking. The most common approach is fitness-proportionate selection, often called *roulette-wheel selection*.

$$P(i) = \frac{f(i)}{\sum_{j=1}^{N} f(j)}$$

This formula gives higher-fitness solutions a greater chance of being chosen while still allowing weaker solutions to survive occasionally. That small amount of randomness helps keep the population diverse, which is important for avoiding premature convergence.

Algorithm Mechanics

How It Works



Flowchart illustrating the main stages of a Genetic Algorithm, including population initialization, fitness evaluation, selection, crossover, mutation, and termination.

A Genetic Algorithm operates through an iterative evolutionary cycle that gradually improves a population of candidate solutions. The process begins with an initial population of randomly generated individuals, each encoding a possible solution to the problem. Every individual is evaluated using a fitness function, which measures how well it performs the task. Based on these fitness values, the algorithm selects a subset of individuals to act as parents; fitter candidates are more likely to be chosen, although some randomness is preserved to maintain diversity within the search.

Once parents are selected, the algorithm applies crossover, exchanging segments of their encoded representations to create offspring that inherit traits from both contributors. Mutation then introduces small random changes to some offspring, helping the population explore new areas of the search space and preventing premature convergence. These offspring form the next generation, and the cycle of evaluation, selection, crossover, and mutation repeats until a stopping condition is met, such as reaching a maximum number of generations or achieving a satisfactory fitness level. Through this iterative process, the population evolves toward increasingly effective solutions.

Strengths and Limitations

Strengths

1. **Effective in complex search spaces**
   GAs perform well when the problem landscape is non-linear, discontinuous, noisy, or poorly understood, where traditional optimization methods fail.
2. **No gradient required**
   They do not depend on derivatives or smoothness, making them suitable for problems where gradients cannot be computed.
3. **Global exploration**
   Because GAs use a population of solutions, they search multiple regions of the solution space simultaneously and are less likely to get stuck in local minima.

Limitations

1. **Parameter sensitivity**
   Performance depends on choosing appropriate values for mutation rate, crossover rate, and population size; poor tuning may lead to slow progress.
2. **Potential for premature convergence**
   If diversity is lost too early, the algorithm may settle on suboptimal solutions and fail to explore better alternatives.
3. **No guarantee of finding the absolute optimum**
   GAs usually find good solutions rather than mathematically perfect ones, especially for very large search spaces.

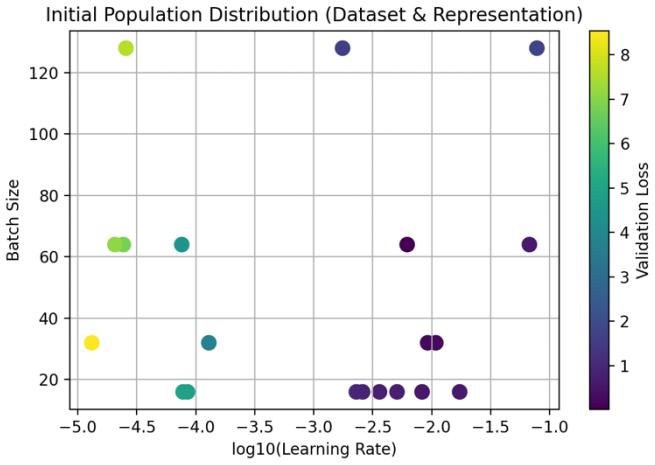
Example Implementation

Dataset and Representation Example:

```
random.seed(42)
np.random.seed(42)

# Learning rate bounds (scientific notation)
LR_MIN = 1e-5   # 0.00001
LR_MAX = 1e-1   # 0.1

BS_OPTIONS = [16, 32, 64, 128] # Batch size options
POP_SIZE = 20
GENS = 20
```

```
Initial Population (Sample):
   Individual  Learning Rate  Batch Size  Initial Loss
0           1       0.003612          16      0.782959
1           2       0.009251          32      0.244229
2           3       0.000078          16      5.034995
3           4       0.005091          16      0.724619
4           5       0.002301          16      0.957875
5           6       0.000013          32      8.537366
6           7       0.000085          16      4.923697
7           8       0.001758         128      1.608427
```

Initial Population Plot:



Code Snippet:

```
Tabnine | Edit | Test | Explain | Document
def random_individual():
    lr = 10 ** random.uniform(-5, -1)
    bs = random.choice(BS_OPTIONS)
    return (lr, bs)


Tabnine | Edit | Test | Explain | Document
def roulette_selection(pop, fits):
    min_fit = min(fits)
    weights = [f - min_fit + 1e-6 for f in fits]
    return random.choices(pop, weights=weights, k=1)[0]


Tabnine | Edit | Test | Explain | Document
def crossover(p1, p2, p=0.9):
    if random.random() < p:
        lr = (p1[0] + p2[0]) / 2
        bs = random.choice([p1[1], p2[1]])
        return (lr, bs), (lr, bs)
    return p1, p2


Tabnine | Edit | Test | Explain | Document
def mutate(ind, p=0.2):
    lr, bs = ind
    if random.random() < p:
        lr *= 10 ** np.random.normal(0, 0.2)
        lr = np.clip(lr, LR_MIN, LR_MAX)
    if random.random() < p:
        bs = random.choice(BS_OPTIONS)
    return (lr, bs)
```

```
Tabnine | Edit | Test | Explain | Document
def run_ga():
    pop = initial_population.copy()
    lr_div_history = []

    best_fit = -np.inf
    best_ind = None

    for g in range(GENS):
        fits = [fitness(ind) for ind in pop]

        gen_best_fit = max(fits)
        gen_best_ind = pop[np.argmax(fits)]

        if gen_best_fit > best_fit:
            best_fit = gen_best_fit
            best_ind = gen_best_ind

        lr_div_history.append(lr_diversity(pop))

        # elitism
        new_pop = [best_ind]

        while len(new_pop) < POP_SIZE:
            p1 = roulette_selection(pop, fits)
            p2 = roulette_selection(pop, fits)
            c1, c2 = crossover(p1, p2)
            new_pop.append(mutate(c1))
            if len(new_pop) < POP_SIZE:
                new_pop.append(mutate(c2))

        pop = new_pop

    return lr_div_history, best_ind, -best_fit
```

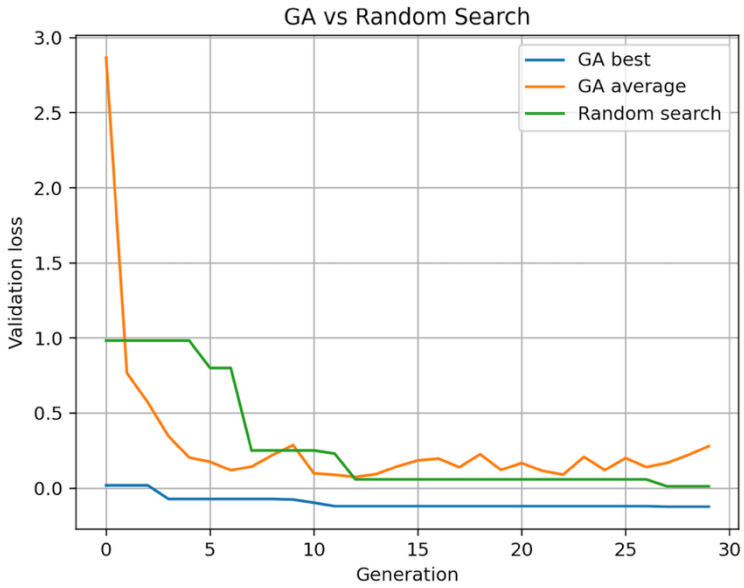
GA Result Generation Log:

```
Running Genetic Algorithm...

Gen 00 | Best fitness: -0.1356 | Best (lr, bs): (0.00620, 64)
Gen 01 | Best fitness: -0.0800 | Best (lr, bs): (0.00620, 64)
Gen 02 | Best fitness: 0.0168 | Best (lr, bs): (0.00620, 64)
Gen 03 | Best fitness: 0.0374 | Best (lr, bs): (0.01126, 64)
Gen 04 | Best fitness: 0.0425 | Best (lr, bs): (0.01212, 64)
Gen 05 | Best fitness: 0.0542 | Best (lr, bs): (0.01218, 64)
Gen 06 | Best fitness: 0.0445 | Best (lr, bs): (0.00997, 64)
Gen 07 | Best fitness: 0.0538 | Best (lr, bs): (0.01237, 64)
Gen 08 | Best fitness: 0.0658 | Best (lr, bs): (0.00755, 64)
Gen 09 | Best fitness: 0.0970 | Best (lr, bs): (0.01162, 64)
Gen 10 | Best fitness: 0.0666 | Best (lr, bs): (0.01175, 64)
Gen 11 | Best fitness: 0.0945 | Best (lr, bs): (0.01137, 64)
Gen 12 | Best fitness: 0.0402 | Best (lr, bs): (0.01065, 64)
Gen 13 | Best fitness: 0.0565 | Best (lr, bs): (0.01162, 64)
Gen 14 | Best fitness: 0.0485 | Best (lr, bs): (0.01098, 64)
Gen 15 | Best fitness: 0.1062 | Best (lr, bs): (0.00993, 64)
Gen 16 | Best fitness: 0.0766 | Best (lr, bs): (0.00977, 64)
Gen 17 | Best fitness: 0.1033 | Best (lr, bs): (0.01011, 64)
Gen 18 | Best fitness: 0.0593 | Best (lr, bs): (0.01055, 64)
Gen 19 | Best fitness: 0.1082 | Best (lr, bs): (0.00825, 64)

================ FINAL RESULT ================
Best learning rate found: 0.008254
Best batch size found:    64
Best validation loss:     -0.108155
```

The Genetic Algorithm successfully optimized the learning rate and batch size over 20 generations. The best configuration found was a learning rate of 0.00825 and a batch size of 64, achieving a minimum validation loss of −0.108.

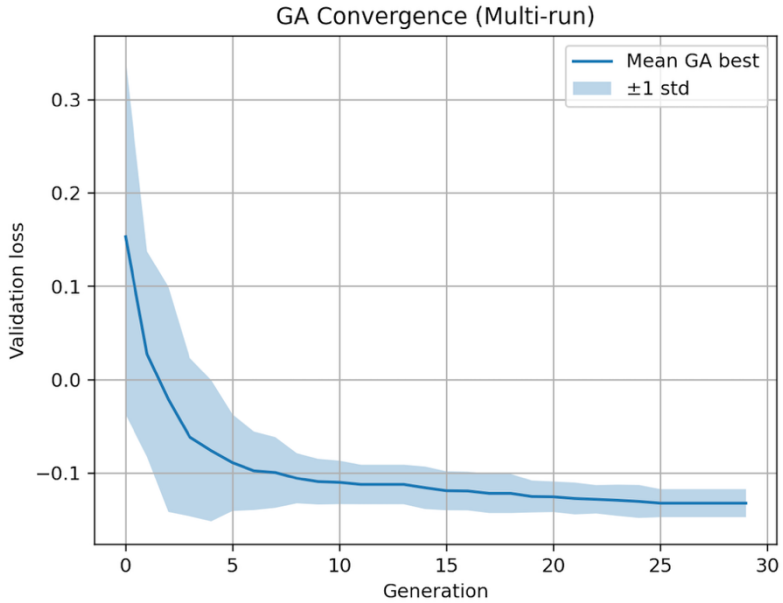
Result:

Figure 1. Comparison of Genetic Algorithm and Random Search



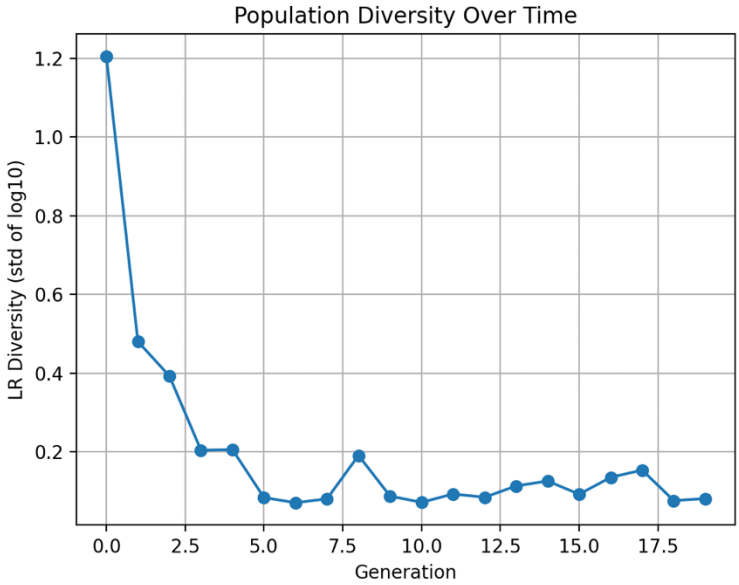
This figure compares Genetic Algorithm (GA) and random search performance using validation loss. GA converges to lower loss earlier and more consistently, demonstrating its effectiveness for black-box hyperparameter tuning.

Figure 2. GA Convergence Across Multiple Runs (Mean ± Standard Deviation)



This figure shows the mean best validation loss across multiple GA runs, with shaded regions indicating ±1 standard deviation. Despite stochastic fluctuations, the overall trend demonstrates stable convergence.

Figure 3. Population Diversity Over Generations



This figure illustrates population diversity over successive generations. Diversity decreases as the GA shifts from exploration to exploitation, indicating convergence toward promising hyperparameter regions.

The results demonstrate that the Genetic Algorithm effectively explored the hyperparameter search space and identified a high-quality solution. Although full convergence to the global optimum was not guaranteed, this behavior is expected given the small population size and limited number of generations used in this demonstration.

Conclusion

Reflection

This project helped me understand Genetic Algorithms as a practical approach to hyperparameter tuning rather than just an abstract optimization method. By tuning the learning rate and batch size, I was able to observe how a population of candidate solutions evolves across generations toward lower validation loss, even though performance does not improve smoothly at every step. I also learned that Genetic Algorithms rely mainly on simple arithmetic and probabilistic operations, and that their effectiveness comes from the evolutionary process that balances exploration and exploitation in a noisy, black-box search space.

Future Questions

- How much improvement would I see if I used adaptive mutation instead of a fixed mutation rate?
- Could the GA be improved by combining it with another optimization technique, such as simulated annealing or gradient descent?