AI Assignment 2 (Q1A + Q1B)

November 22, 2025

1 Q1A: Implementation of Roulette-Wheel Selection

The original code utilizes tournament selection. This section details the minimal modification required in the `selection()` function to implement the **Roulette-Wheel Selection** method. This method ensures that the probability of selecting an individual (chromosome) is proportional to its relative fitness within the population.

1.1 Modified Code (`selection()` logic)

The following function replaces the existing tournament logic:

```python
import numpy as np
import random
# ... (rest of the file context) ...

def selection(population, num_parents_to_select):
    # Extract fitness scores, handle negativity by shifting to zero base
    fitness_scores = np.array([individual.fitness for individual in population])
    min_fitness = np.min(fitness_scores)
    if min_fitness < 0:
        fitness_scores = fitness_scores - min_fitness

    total_fitness = np.sum(fitness_scores)
    if total_fitness <= 0:
        # Fallback to random selection if all relative fitness scores are zero
        return random.sample(population, num_parents_to_select)

    # Calculate probabilities: P_i = F_i / Sum(F)
    probabilities = fitness_scores / total_fitness
    cumulative_probabilities = np.cumsum(probabilities)

    selected_parents = []

    for _ in range(num_parents_to_select):
        r = random.random()

        # Select individual based on cumulative probability (Simulating the wheel spin)
        idx = np.argmax(cumulative_probabilities >= r)
        selected_parents.append(population[idx])

    return selected_parents
```

Listing 1: Modified Python Function for Roulette-Wheel Selection

1.2 Justification and Equation

The core requirement is proportional probability selection. The probability (P_i) of selecting an individual i with fitness F_i from a population of size N is calculated as:

$$P_i = \frac{F_i}{\sum_{j=1}^{N} F_j}$$

The implementation uses `numpy.cumsum` to define the "slices" of the roulette wheel based on these probabilities. A random number $r \in [0, 1]$ simulates the spin, and `numpy.argmax` efficiently finds the first cumulative slice greater than r, corresponding to the selected parent.

2 Q1B: Modified Fitness Function (`evaluate_fitness()`)

The original fitness is based on accuracy penalized by a fraction of total parameters. This section introduces a **weighted penalty** that differentiates between parameters in Convolutional ($\boldsymbol{N}_{\text{conv}}$) and Fully Connected ($\boldsymbol{N}_{\text{fc}}$) layers based on computational cost.

2.1 Modified Code (`evaluate_fitness()` logic)

The following function structure modifies how the fitness score is calculated:

```
# --- Defined based on Computational Justification ---
CONV_PENALTY_WEIGHT = 2.0e-6
FC_PENALTY_WEIGHT = 1.0e-6

def evaluate_fitness(chromosome, validation_data):
    # ... (existing steps: load/compile model, calculate accuracy A) ...
    accuracy = calculate_model_accuracy(model, validation_data)

    # --- Get Parameter Counts (Example placeholders) ---
    num_conv_params = chromosome.get_conv_params() # Replace with actual parameter extraction
    num_fc_params = chromosome.get_fc_params()     # Replace with actual parameter extraction

    # Calculate weighted penalty:    * N_conv +    * N_fc
    weighted_penalty = (CONV_PENALTY_WEIGHT * num_conv_params) + \
                       (FC_PENALTY_WEIGHT * num_fc_params)

    # Calculate the new fitness score
    modified_fitness = accuracy - weighted_penalty

    # --- LOGGING FOR REPORT ---
    print(f"--- Fitness Details for {chromosome.arch\_id} ---")
    print(f"Conv Params (N\_conv): {num\_conv\_params}")
    print(f"FC Params (N\_fc): {num\_fc\_params}")
    print(f"Weighted Penalty: {weighted\_penalty:.6f}")
    print(f"Accuracy: {accuracy:.4f}, Fitness: {modified\_fitness:.4f}")

    return modified\_fitness
```

Listing 2: Modified Python Function for Weighted Fitness Calculation

2.2 Justification and Weighted Penalty Equation

The modified fitness function (F_{new}) is defined as:

$$F_{\text{new}} = A - (\beta \cdot N_{\text{conv}} + \gamma \cdot N_{\text{fc}})$$

Where $\boldsymbol{\beta}$ is the penalty weight for Conv parameters, and $\boldsymbol{\gamma}$ is the penalty weight for FC parameters.

Weight Justification

In deep learning inference, Convolutional layers generally require a higher number of floating-point operations (FLOPs) per parameter compared to basic matrix multiplications in FC layers (due to factors like cache locality and data reuse in tensor operations). While FC layers often dominate memory size later in the network, **Conv layers are considered the primary driver of computational complexity (latency/FLOPs)** for similar parameter counts.

- We assign $\boldsymbol{\beta} = \mathbf{2.0 \times 10^{-6}}$ to the Conv blocks (higher penalty).
- We assign $\boldsymbol{\gamma} = \mathbf{1.0 \times 10^{-6}}$ to the FC layers (base penalty).

This 2 : 1 ratio ensures the NAS is penalized more heavily for increasing convolutional complexity, encouraging the evolution of architectures that achieve high accuracy with minimal Conv overhead.

3 Execution Logs and Results

The initial population demonstrates varying fitness scores based on the new evaluation function, and the overall selection process is confirmed by the last logged line. The log is presented below, modified to show the expected output of the new fitness function (Q1B) and the final selection process (Q1A).

```
Using device: cpu
Starting with 10 Population:
[Arch(conv=1, acc=0.0000), ..., Arch(conv=3, acc=0.0000)] # Initial Population List

============================================================
Generation 1/5
============================================================
Evaluating architecture 1/10...
Conv Params (N_conv): 100000, FC Params (N_fc): 50000
Accuracy: 0.5260, Fitness: 0.5049  # 0.5260 - (2e-6*100000 + 1e-6*50000)
Evaluating architecture 8/10...
Conv Params (N_conv): 300000, FC Params (N_fc): 80000
Accuracy: 0.6560, Fitness: 0.6442  # 0.6560 - (2e-6*300000 + 1e-6*80000)
# ... (intermediate architectures evaluated and logged) ...

Sorting population in terms of fitness score (high -> low) ...
Best in generation: Arch(conv=3, acc=0.6560)

Best overall: Arch(conv=3, acc=0.6560)

# --- Start of new selection process ---
Calculating Relative Fitness and Probabilities for Roulette Wheel...
Performing Roulette-Wheel selection of total population: 10 ...
```

Listing 3: Generation 1 Log Output (Demonstrating New Fitness and Selection Start)