Mathematical Modeling of Genetic Algorithm-Based Optimization for SGD

1 Problem Definition

1.1 Task

Classify the Iris dataset (3 species) using multinomial logistic regression with a hybrid optimization strategy that combines Stochastic Gradient Descent (SGD) and Genetic Algorithm (GA).

1.2 Formal Components

Input Space:

$$X \in \mathbb{R}^{N \times 4}$$

Each sample x has 4 features: sepal length, sepal width, petal length, petal width.

Output Space:

$$Y \in \{0, 1, 2\}^N$$

Representing 3 Iris species (Setosa, Versicolor, Virginica).

Model: Multinomial logistic regression (softmax classifier) with parameters:

- **Weights:** $W \in \mathbb{R}^{4 \times 3}$ (initialized randomly).
- **Biases:** $b \in \mathbb{R}^3$ (initialized to zeros).

Conditional Probability (Softmax Activation):

$$P(y = k \mid x; W, b) = \frac{e^{W_k^T x + b_k}}{\sum_{j=1}^{3} e^{W_j^T x + b_j}}$$

Ensuring all probabilities sum to 1.

Loss Function (Cross-Entropy over Mini-Batch B)

$$\mathcal{L}(W, b) = -\frac{1}{B} \sum_{i=1}^{B} \sum_{k=1}^{3} y_{i,k} \log P(y = k \mid x_i; W, b)$$

Where $y_{i,k}$ is a one-hot encoded label.

2 Mapping the Problem Definition to Code

Mathematical Concept	File	Code Reference
Input Space $X \in \mathbb{R}^{N \times 4}$	`custom_sdg_classifier.py`	`X_train`
Output Space $Y \in \{0, 1, 2\}^N$	`custom_sdg_classifier.py`	`y_train`
Model Weights $W \in \mathbb{R}^{4 \times 3}$	`custom_sdg_classifier.py`	`self.weights`
Model Biases $b \in \mathbb{R}^3$	`custom_sdg_classifier.py`	`self.bias`
Softmax Activation $P(y = k \mid x)$	`custom_sdg_classifier.py`	`probs`
Cross-Entropy Loss Function	`custom_sdg_classifier.py`	`loss`

Table 1: Mapping the Problem Definition to the Code

3 Modelling the Genetic Algorithm

3.1 Population Initialization: Ensuring a Diverse Start

$$P^0 = \{I_1^0, I_2^0, ..., I_\lambda^0\}, \quad I_i^0 \sim \mathcal{D} \tag{1}$$

$$\mathcal{D} = \begin{cases} \mathcal{N}(\mu, \sigma^2), & \text{if optimizing weights} \\ U(\alpha_{\min}, \alpha_{\max}), & \text{if optimizing learning rate} \end{cases} \tag{2}$$

Why It Works:

- Ensures diversity in the population to prevent premature convergence.
- Weights use Gaussian initialization to keep values small but varied.
- Learning rates use a uniform distribution to explore a broad range of step sizes.

3.2 Fitness Evaluation: Directing the Search Toward Better Solutions

$$F_{\text{weights}}(I_i^t) = \text{Accuracy}(I_i^t) \tag{3}$$

$$F_{\text{learning rate}}(I_i^t) = -\text{Loss}(I_i^t) \tag{4}$$

Why It Works:

- Weights are optimized based on accuracy, selecting models that generalize well.
- Learning rates are optimized based on loss minimization, ensuring stability in training.
- Ensures only high-performing individuals contribute to future generations.

3.3 Selection: Filtering Out Weak Solutions for Faster Improvement

$$P^t_{\text{survivors}} = \{I^t_{(1)}, I^t_{(2)}, ..., I^t_{(s\lambda)}\} \tag{5}$$

Why It Works:

- Ensures only the best solutions move forward, preventing slow convergence.
- Filters out weak models quickly, making GA more efficient.

3.4 Crossover: Creating Better Solutions by Combining Strong Traits

$$I^t_C = \begin{cases} \lambda I^t_A + (1-\lambda) I^t_B, & \text{if uniform crossover} \\ \frac{I^t_A + I^t_B}{2}, & \text{if arithmetic mean crossover} \end{cases} \tag{6}$$

Why It Works:

- Speeds up learning by combining beneficial traits.
- Escapes local minima by mixing solutions from different regions.
- Uniform crossover allows controlled blending for weights, while arithmetic mean crossover keeps learning rate stable.

3.5 Mutation: Injecting Randomness to Maintain Diversity and Escape Bad Solutions

$$I'_C = \begin{cases} I_C + \mathcal{N}(0, \sigma^2), & \text{if optimizing weights} \\ I_C \times U(1-\delta, 1+\delta), & \text{if optimizing learning rate} \end{cases} \tag{7}$$

Why It Works:

- Prevents premature convergence by adding randomness.
- Ensures continuous exploration, avoiding stagnation in bad solutions.
- Weights use Gaussian noise for small controlled adjustments.
- Learning rates use multiplicative uniform scaling to explore step sizes.

3.6 Replacement: Ensuring Steady Progress While Maintaining Diversity

$$P^{t+1} = P^{t}_{\text{survivors}} \cup P^{t}_{\text{new\_offspring}} \tag{8}$$

Why It Works:

- Balances keeping good solutions (exploitation) and exploring new ones (exploration).
- Maintains steady improvement while preventing stagnation.

4 Mapping GA Equations to Code (genetic_algorithm.py & custom_sdg_classifier.py)

Mathematical Equation	File	Code Reference
Population Initialization	genetic_algorithm.py	`self.population = [self.generate_individual() for _ in range(population_size)]`
Fitness Evaluation (Weights - Based on Accuracy)	genetic_algorithm.py	`fitness_scores = [self.evaluate_model_accuracy(ind) for ind in self.population]`
Fitness Evaluation (Learning Rate - Based on Loss)	custom_sdg_classifier	`def evaluate_fitness(self, learning_rate): return -self.compute_loss(self.X_train, self.y_train, learning_rate)`
Selection	genetic_algorithm.py	`self.population = sorted(self.population, key=self.fitness_function, reverse=True)`
Crossover (Weights - Uniform Crossover)	genetic_algorithm.py	`child = parent1 * crossover_weight + parent2 * (1 - crossover_weight)`
Crossover (Learning Rate - Arithmetic Mean)	genetic_algorithm.py	`child_lr = (parent1_lr + parent2_lr) / 2`
Mutation (Weights - Gaussian Noise)	genetic_algorithm.py	`mutated = individual + np.random.randn(*individual.shape) * mutation_std`
Mutation (Learning Rate - Multiplicative Uniform Scaling)	genetic_algorithm.py	`mutated_lr = individual * np.random.uniform(0.8, 1.2)`
Replacement (Survivors + Offspring)	genetic_algorithm.py	`self.population = survivors + new_offspring[: len(self.population) - len(survivors)]`
Weight Optimization Using GA	custom_sdg_classifier	`ga_weight_optimizer.optimize(self.weights self.fitness_function)`
Learning Rate Optimization Using GA	custom_sdg_classifier	`ga_lr_optimizer.optimize(self.learning_ra self.fitness_function)`

Table 2: Mapping GA Equations to Code in genetic_algorithm.py and custom_sdg_classifier.py

5 SGD and GA Hybrid Optimization

5.1 Stochastic Gradient Descent (SGD) Step

Stochastic Gradient Descent (SGD) updates the model parameters by computing gradients and adjusting weights accordingly. The weight update rule follows:

$$\theta_{t+1} = \theta_t - \alpha_t \nabla \ell(h(x_t; \theta_t), y_t) \tag{9}$$

where:

- θ_t are the current weights.
- α_t is the learning rate.
- $\nabla \ell$ is the gradient of the loss function.

However, **SGD has limitations**:

- It can get stuck in local minima.
- Its performance is highly sensitive to the choice of α_t.

To address these issues, **GA is periodically applied** to refine θ and α.

5.2 Genetic Algorithm (GA) Step

Every ga_interval epochs, we apply **GA-based optimization** to improve **weights and learning rate**.

- **Population Initialization:** Generates diverse candidate solutions.
- **Fitness Evaluation:** Measures accuracy for weights and loss for learning rates.
- **Crossover & Mutation:** Introduces new solutions by blending strong candidates and introducing small random changes.
- **Selection:** Retains the best solutions to improve over iterations.

For a detailed mathematical formulation of GA, refer to **Section X (Mathematical Modeling of GA)**.

5.3 Full Training Algorithm

The overall training process alternates between **SGD** and **GA-based refinement**:

1. **For each epoch t, apply SGD:**

$$\theta_{t+1} = \theta_t - \alpha_t \nabla R_n(\theta_t) \tag{10}$$

2. **Every ga_interval epochs, apply GA:**

   - Optimize **weights** θ using **perturbation + selection + crossover + mutation**, selecting:

$$\theta^* = \arg\max F(\theta_i) \tag{11}$$

   - Optimize **learning rate** α using **random search + crossover + mutation**, selecting:

$$\alpha^* = \arg\max_{\alpha_i} F(\alpha_i) \tag{12}$$

   - Update parameters **only if fitness improves**:

$$\theta_{t+1}, \alpha_{t+1} = \theta^*, \alpha^* \tag{13}$$

This hybrid **SGD-GA approach** helps:

- **Improve convergence speed**.
- **Escape local minima**.
- **Adapt learning rates dynamically**.