1 Particle Swarm Optimization Algorithm

1.1 Concept of Particle Swarm Optimization

Example: Swarm Intelligence Metaphor

Biological inspiration: Bird flocking or fish schooling behavior
Search space: n-dimensional optimization landscape
Particles: Individual solutions exploring the space
Swarm intelligence: Collective knowledge sharing
Simple example in 2D:

- **Objective**: Find minimum of $f(x, y) = x^2 + y^2$
- **Particles**: 10 points in 2D space
- **Positions**: $(x_1, y_1), (x_2, y_2), \ldots, (x_{10}, y_{10})$
- **Velocities**: Movement direction and speed for each particle
- **Global best**: Best position found by any particle

The swarm collectively explores the search space, balancing individual experience with group knowledge.

Particle Swarm Optimization (PSO) is a population-based stochastic optimization technique inspired by social behavior patterns of organisms like bird flocking or fish schooling.

1.2 Mathematical Foundation

1.2.1 PSO Core Equations

Example: Velocity and Position Update

For each particle i and dimension j:

$$\begin{aligned}
v_{ij}^{t+1} &= w \cdot v_{ij}^{t} + c_1 r_1 (pbest_{ij} - x_{ij}^{t}) + c_2 r_2 (gbest_j - x_{ij}^{t}) \\
x_{ij}^{t+1} &= x_{ij}^{t} + v_{ij}^{t+1}
\end{aligned}$$

Where:

- $w = 0.729$: inertia weight maintaining motion momentum
- $c_1 = 1.49445$: cognitive coefficient for personal best attraction
- $c_2 = 1.49445$: social coefficient for global best attraction
- $r_1, r_2 \sim U(0, 1)$: random exploration factors
- $pbest_i$: personal best position of particle i
- $gbest$: global best position of entire swarm

Physical interpretation:

- **Inertia**: $w \cdot v_{ij}^{t}$ - continues previous direction
- **Cognitive**: $c_1 r_1 (pbest_{ij} - x_{ij}^{t})$ - moves toward personal best
- **Social**: $c_2 r_2 (gbest_j - x_{ij}^{t})$ - moves toward global best

The PSO algorithm follows these fundamental update equations:

$$\vec{v}_i^{t+1} = w \cdot \vec{v}_i^{t} + c_1 r_1 (\vec{pbest}_i - \vec{x}_i^{t}) + c_2 r_2 (\vec{gbest} - \vec{x}_i^{t}) \quad (1)$$

$$\vec{x}_i^{t+1} = \vec{x}_i^{t} + \vec{v}_i^{t+1} \quad (2)$$

2 Bayesian Optimization Algorithm

2.1 Concept of Bayesian Optimization

Example: Bayesian Optimization Metaphor

Scenario: Optimizing a black-box function with expensive evaluations
Problem: Find $\max_{x\in[-5,5]} f(x)$ where f is costly to compute
Bayesian approach:

- Build probabilistic model (surrogate) of $f(x)$
- Use model to decide where to evaluate next
- Balance exploration (uncertain regions) and exploitation (promising regions)
- Converge to optimum with few function evaluations

Simple 1D example:

1. **Initial points**: $X = [-4, 0, 3]$, $y = [f(-4), f(0), f(3)]$
2. **GP model**: Predicts mean and uncertainty everywhere
3. **Acquisition**: Choose next point $x^* = 1.2$ (high potential)
4. **Update**: Add $(1.2, f(1.2))$ to observations, update model

Bayesian optimization treats the objective function as a random function and uses Bayesian inference to model it.

Bayesian Optimization (BO) is a powerful framework for global optimization of black-box functions that are expensive to evaluate, using probabilistic surrogate models and acquisition functions to guide the search.

2.2 Mathematical Foundation

2.2.1 Gaussian Process Framework

Example: Gaussian Process Intuition

Gaussian Process definition:

- A GP is a collection of random variables, any finite number of which have a joint Gaussian distribution
- Defined by mean function $m(\vec{x})$ and covariance function $k(\vec{x}, \vec{x}')$
- For BO: $f(\vec{x}) \sim \mathcal{GP}(m(\vec{x}), k(\vec{x}, \vec{x}'))$

Initial GP parameters:

$$\begin{aligned} \text{Mean :} &\quad m(\vec{x}) = 0 \quad \text{(self.gp\_mean)} \\ \text{Standard deviation :} &\quad \sigma = 1 \quad \text{(self.gp\_std)} \\ \text{Covariance :} &\quad k(\vec{x}, \vec{x}') = \sigma^2 \cdot \text{correlation}(\vec{x}, \vec{x}') \end{aligned}$$

Prior distribution:

- Before seeing data, believe $f(\vec{x}) \sim \mathcal{N}(0, 1)$
- This is an uninformative prior
- Will be updated as observations are collected

Visualization: Imagine a "cloud" of possible functions centered around zero with unit variance.

The Gaussian Process is defined as:

$$f(\vec{x}) \sim \mathcal{GP}(m(\vec{x}), k(\vec{x}, \vec{x}')) \tag{3}$$

with initial parameters:

$$m(\vec{x}) = 0, \quad k(\vec{x}, \vec{x}') = \sigma^2 \cdot \rho(\vec{x}, \vec{x}'), \quad \sigma = 1 \tag{4}$$

2.3 Class Architecture and Initialization

Example: BO State Initialization

For bounds $= [(-5,5),(-3,3)]$ (2-dimensional problem):

$$\begin{aligned} \text{bounds} &= [(-5,5),(-3,3)] \quad \text{(search space boundaries)} \\ \text{X\_observed} &= [] \quad \text{(empty list, no data yet)} \\ \text{y\_observed} &= [] \quad \text{(empty list, no evaluations yet)} \\ \text{gp\_mean} &= 0 \quad \text{(prior mean function)} \\ \text{gp\_std} &= 1 \quad \text{(prior standard deviation)} \end{aligned}$$

Initial state properties:

- No observations: $n = 0$ data points
- Prior belief: $f(\vec{x}) \sim \mathcal{N}(0,1)$ for all $\vec{x}$
- Search space: $\mathcal{X} = [-5,5] \times [-3,3]$
- Ready to collect first observation

Dimensional analysis:

$$\begin{aligned} \text{Dimensions}: &\quad D = \text{len(bounds)} = 2 \\ \text{Volume}: &\quad V = \prod_{j=1}^{D} (b_j - a_j) = 10 \times 6 = 60 \end{aligned}$$

2.3.1 Search Space Definition

Example: Bounds Specification and Interpretation

Bounds structure: List of tuples $(lower\_bound, upper\_bound)$
Common patterns:

- **1D optimization**: $[(-10, 10)]$
- **2D problem**: $[(-5, 5), (-3, 3)]$
- **High-dimensional**: $[(0, 1)] \times D$ (normalized space)
- **Mixed scales**: $[(0, 100), (-1, 1), (0.001, 0.1)]$

Normalization importance:

$$\begin{aligned} \text{Original bounds}: &\quad [(-10, 10), (-100, 100)] \\ \text{Problem}: &\quad \text{Second dimension dominates distance calculations} \\ \text{Solution}: &\quad \text{Normalize to } [(-1, 1), (-1, 1)] \text{ or use length scales} \end{aligned}$$

Feasible region:

$$\mathcal{X} = \{\vec{x} \in \mathbb{R}^D : a_j \leq x_j \leq b_j \text{ for } j = 1, \ldots, D\} \tag{5}$$

For our example: $\vec{x} \in [-5, 5] \times [-3, 3] \subset \mathbb{R}^2$

The search space is defined as:

$$\mathcal{X} = \{\vec{x} \in \mathbb{R}^D : a_j \leq x_j \leq b_j \text{ for } j = 1, \ldots, D\} \tag{6}$$

where $D = \text{len(bounds)}$.

Algorithm 1 Bayesian Optimizer Initialization

```
1: function BayesianOptimizer_Init(bounds)
2:     $self.bounds \leftarrow bounds$                    ▷ Search space boundaries
3:     $self.X\_observed \leftarrow [\,]$                 ▷ Observed input points
4:     $self.y\_observed \leftarrow [\,]$                 ▷ Observed function values
5:     $self.gp\_mean \leftarrow 0$                       ▷ Prior mean function
6:     $self.gp\_std \leftarrow 1$                        ▷ Prior standard deviation
7: end function
```

2.4 Algorithm Initialization

2.5 Data Collection Structure

Example: Observation Storage Format

After 3 evaluations:

$$X\_observed = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \end{bmatrix} = \begin{bmatrix} 2.1 & -1.3 \\ -3.8 & 2.7 \\ 4.2 & 0.5 \end{bmatrix}$$
$$y\_observed = \begin{bmatrix} f(\vec{x}_1) \\ f(\vec{x}_2) \\ f(\vec{x}_3) \end{bmatrix} = \begin{bmatrix} -3.2 \\ -8.1 \\ -5.7 \end{bmatrix}$$

Data matrix properties:

- $X \in \mathbb{R}^{n \times D}$: n observations, D dimensions
- $y \in \mathbb{R}^n$: n function evaluations
- Maintains evaluation history for model training
- Enables sequential decision making

Storage efficiency:

$$\begin{aligned} \text{Memory for } X: &\quad n \times D \text{ floats} \\ \text{Memory for } y: &\quad n \text{ floats} \\ \text{For } n = 100, D = 5: &\quad 500 + 100 = 600 \text{ floats} \approx 4.8\text{KB} \end{aligned}$$

Growth pattern: Linear in number of evaluations.

The observation data is structured as:

$$\mathbf{X} = \begin{bmatrix} \vec{x}_1 \\ \vec{x}_2 \\ \vdots \\ \vec{x}_n \end{bmatrix}, \quad \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{7}$$

2.6 Prior Distribution Specification

Example: Prior Belief Interpretation

Constant mean prior: $m(\vec{x}) = 0$

- Assumes function values centered around zero
- Reasonable when function is normalized
- Can be updated as data is observed

Unit variance prior: $\sigma = 1$

- Scale of function values assumed to be around 1
- Works well with normalized functions
- Provides reasonable uncertainty quantification initially

Bayesian interpretation:

$$\begin{aligned} \text{Prior}: &\quad p(f) = \mathcal{N}(f|0,1) \\ \text{Likelihood}: &\quad p(\text{data}|f) \quad \text{(from observations)} \\ \text{Posterior}: &\quad p(f|\text{data}) \propto p(\text{data}|f) \cdot p(f) \end{aligned}$$

Initial uncertainty:

- Before any data: high uncertainty everywhere
- Standard deviation = 1 across entire space
- As data accumulates: uncertainty reduces near observations

Alternative priors:

- **Empirical mean**: Estimate from initial samples
- **Linear trend**: $m(\vec{x}) = \vec{w}^T\vec{x} + b$
- **Problem-specific**: Domain knowledge incorporation

The prior distribution represents initial beliefs:

$$p(f(\vec{x})) = \mathcal{N}(f(\vec{x})|\mu_0, \sigma_0^2) \quad \text{with} \quad \mu_0 = 0, \sigma_0 = 1 \tag{8}$$

2.7 Implementation Considerations

2.7.1 Initialization Validation

Example: Input Validation and Edge Cases

Bounds validation:

- Check: $\forall j, \text{low}_j < \text{high}_j$
- Handle: Empty bounds list (theoretical edge case)
- Consider: Infinite bounds (not supported in practice)

Data structure choices:

- `X_observed`: List of lists or numpy array
- `y_observed`: List of floats
- Conversion to numpy arrays for efficient computation

Numerical stability:

- GP mean = 0 avoids numerical issues initially
- Unit variance provides well-scaled computations
- Empty observation lists handle gracefully

Extension points:

```
def __init__(self, bounds, initial_mean=0, initial_std=1):
    self.bounds = bounds
    self.X_observed = []
    self.y_observed = []
    self.gp_mean = initial_mean
    self.gp_std = initial_std
    # Additional: kernel parameters, acquisition function, etc.
```

2.7.2 Memory Management

Example: Storage Requirements Analysis

Memory growth analysis:

$$\begin{aligned}\text{Per observation}: &\quad D \text{ floats for } X + 1 \text{ float for } y\\ \text{Total for } n \text{ observations}: &\quad n \times (D+1) \text{ floats}\\ \text{For } D=5, n=100: &\quad 100 \times 6 = 600 \text{ floats} \approx 4.8\text{KB}\end{aligned}$$

Comparison with other optimizers:

- **PSO**: Stores $3ND + N$ floats (constant)
- **BO**: Grows linearly with evaluations
- **Advantage**: BO typically needs fewer evaluations
- **Trade-off**: Memory vs. function evaluations

Practical limits:

- GP training cost: $O(n^3)$ for matrix inversion
- Typical $n_{\max}$: 100-1000 observations
- Memory rarely the limiting factor

Storage optimization strategies:

- **Subsampling**: Keep only most informative points
- **Forgetting**: Remove old/uninformative observations
- **Sparse GP**: Use inducing points for large n

The memory requirements grow as:

$$M(n) = O(n \cdot D) \quad \text{for storing observations} \tag{9}$$

2.8 Bayesian Inference Framework

Example: Sequential Learning Process

Initial state (n=0):

$$\begin{aligned} \text{Prior}: &\quad f(\vec{x}) \sim \mathcal{N}(0,1) \quad \forall \vec{x} \in \mathcal{X} \\ \text{Uncertainty}: &\quad \text{High everywhere} \end{aligned}$$

After first observation (n=1):

Update GP to get posterior

$$\text{Uncertainty}: \quad \text{Low near } \vec{x}_1, \text{ high elsewhere}$$

After multiple observations:

Posterior reflects data pattern

$$\text{Uncertainty}: \quad \text{Low near data, high in unexplored regions}$$

Bayesian update mathematics:

$$\begin{aligned} \text{Prior}: &\quad p(f) \\ \text{Likelihood}: &\quad p(\mathbf{X}, \mathbf{y} | f) \\ \text{Posterior}: &\quad p(f | \mathbf{X}, \mathbf{y}) \propto p(\mathbf{X}, \mathbf{y} | f) \cdot p(f) \end{aligned}$$

For Gaussian processes:

- Prior: $f \sim \mathcal{GP}(0, k(\cdot, \cdot))$
- Posterior: $f | \mathbf{X}, \mathbf{y} \sim \mathcal{GP}(\mu_n(\cdot), k_n(\cdot, \cdot))$
- Update equations available in closed form

The Bayesian update process follows:

$$p(f|\mathcal{D}) \propto p(\mathcal{D}|f) \cdot p(f) \tag{10}$$

where $\mathcal{D} = \{(\vec{x}_i, y_i)\}_{i=1}^n$ is the observed data.

2.9 Theoretical Foundations

Example: Convergence Guarantees

No-regret property:

- Cumulative regret grows sublinearly
- Implies convergence to global optimum
- Formal guarantees for certain acquisition functions

Information-theoretic view:

$$\begin{aligned} \text{Goal}: &\quad \max_{\vec{x}} f(\vec{x}) \\ \text{Approach}: &\quad \max_{\vec{x}} \text{Acquisition}(\vec{x}) \\ \text{Where}: &\quad \text{Acquisition} = \text{Expected Improvement, UCB, etc.} \end{aligned}$$

Global convergence:

$$\lim_{n\to\infty} P\left(\max_{\vec{x}\in\mathcal{X}} f(\vec{x}) - f(\vec{x}_n^*) < \epsilon\right) = 1 \tag{11}$$

Rate of convergence:

- Depends on kernel smoothness and acquisition function
- Typically exponential for smooth functions
- Formal rates available for specific settings

Practical implications:

- BO provably finds global optimum with enough evaluations
- Much faster than grid search or random search
- Particularly efficient for low-dimensional expensive functions

The theoretical foundation ensures:

$$\lim_{n\to\infty} f(\vec{x}_n^*) = \max_{\vec{x}\in\mathcal{X}} f(\vec{x}) \quad \text{almost surely} \tag{12}$$

for proper acquisition functions and kernel choices.

2.10 Application Domains

Example: Practical BO Applications

Hyperparameter tuning:

- Neural network architecture search
- Learning rate, regularization parameters
- Feature engineering parameters

Experimental design:

- Materials science: alloy composition optimization
- Drug discovery: molecular parameter optimization
- Physics: experimental parameter tuning

Engineering optimization:

- Aerodynamic shape optimization
- Structural design parameters
- Control system tuning

Advantages over other methods:

- **Sample efficiency**: Fewer function evaluations
- **Uncertainty quantification**: Built-in error bars
- **Global optimization**: Finds global optima, not just local
- **Black-box handling**: No gradient information needed

Limitations:

- **Scalability**: Curse of dimensionality for $D > 20$
- **Computational cost**: GP training cost $O(n^3)$
- **Kernel sensitivity**: Performance depends on kernel choice

Bayesian Optimization excels in problems where:

$$\text{Cost}(f(\vec{x})) \gg \text{Cost}(\text{BO update}) \quad \text{and} \quad D \leq 20 \tag{13}$$

3 Acquisition Function Algorithm

3.1 Concept of Acquisition Functions

Example: Bayesian Decision Making

Scenario: We have observed points at $x = [-3, 0, 2]$ with values $y = [4, 1, 3]$
Question: Where should we evaluate next to maximize expected improvement?
Acquisition function role:

- Quantifies "promisingness" of candidate points
- Balances exploration (high uncertainty) vs exploitation (high predicted mean)
- Guides sequential decision making in Bayesian optimization

Simple example calculation:

$$\begin{aligned}
\text{Candidate } x &= 1.5 \\
\text{Distances}: &\quad d_1 = |1.5 - (-3)| = 4.5, \quad d_2 = 1.5, \quad d_3 = 0.5 \\
\text{Weights}: &\quad w_1 = e^{-4.5} \approx 0.011, \quad w_2 = e^{-1.5} \approx 0.223, \quad w_3 = e^{-0.5} \approx 0.607 \\
\text{Weighted mean}: &\quad \mu = \frac{0.011 \times 4 + 0.223 \times 1 + 0.607 \times 3}{0.011 + 0.223 + 0.607} \approx 2.42 \\
\text{Variance}: &\quad \sigma^2 = \frac{1}{0.841 + 1} \approx 0.543 \\
\text{Acquisition}: &\quad \alpha(1.5) = 2.42 + 2.0 \times \sqrt{0.543} \approx 3.90
\end{aligned}$$

The acquisition function combines local predictions with uncertainty estimates.

Acquisition functions are the decision-making engine of Bayesian optimization, quantifying how desirable it is to evaluate a candidate point based on the current surrogate model.

3.2 Mathematical Foundation

3.2.1 Upper Confidence Bound (UCB) Formulation

Example: UCB Acquisition Mathematics

Upper Confidence Bound formulation:

$$\alpha_{\text{UCB}}(\vec{x}) = \mu(\vec{x}) + \kappa \cdot \sigma(\vec{x}) \tag{14}$$

Components:

- $\mu(\vec{x})$: Predicted mean at point $\vec{x}$
- $\sigma(\vec{x})$: Predicted standard deviation at $\vec{x}$
- κ: Exploration weight (balancing parameter)

Interpretation:

- $\mu(\vec{x})$: Exploitation term (go where model predicts high values)
- $\kappa \cdot \sigma(\vec{x})$: Exploration term (go where model is uncertain)
- κ: Controls exploration-exploitation trade-off

Theoretical guarantee: For proper κ_t, UCB achieves sublinear regret.
Our implementation: Uses kernel-weighted local averaging instead of full GP.

The acquisition function follows the Upper Confidence Bound (UCB) principle:

$$\alpha(\vec{x}) = \mu(\vec{x}) + \kappa \cdot \sigma(\vec{x}) \tag{15}$$

where κ is the exploration weight.

3.3 Algorithm Step-by-Step

3.3.1 Distance-Based Weighting

Example: Local Model Construction

Input: $\vec{x} = [1.2, -0.5]$, observed points: $\vec{x}_1 = [0, 0], \vec{x}_2 = [2, 1], \vec{x}_3 = [-1, -1]$

Distance calculation:

$$d_1 = \|\vec{x} - \vec{x}_1\| = \sqrt{(1.2 - 0)^2 + (-0.5 - 0)^2} = \sqrt{1.44 + 0.25} = \sqrt{1.69} = 1.3$$
$$d_2 = \|\vec{x} - \vec{x}_2\| = \sqrt{(1.2 - 2)^2 + (-0.5 - 1)^2} = \sqrt{0.64 + 2.25} = \sqrt{2.89} = 1.7$$
$$d_3 = \|\vec{x} - \vec{x}_3\| = \sqrt{(1.2 - (-1))^2 + (-0.5 - (-1))^2} = \sqrt{4.84 + 0.25} = \sqrt{5.09} = 2.26$$

Weight calculation:

$$w_1 = e^{-d_1} = e^{-1.3} \approx 0.273$$
$$w_2 = e^{-d_2} = e^{-1.7} \approx 0.183$$
$$w_3 = e^{-d_3} = e^{-2.26} \approx 0.104$$

Properties:

- Weights decay exponentially with distance
- Nearby points have higher influence
- Acts as a soft nearest-neighbors approach

The distance weighting uses exponential kernels:

$$w_i = \exp(-d_i) = \exp(-\|\vec{x} - \vec{x}_i\|) \tag{16}$$

3.3.2 Local Mean and Variance Estimation

Example: Statistical Estimation Process

Given: Weights $w = [0.273, 0.183, 0.104]$, observations $y = [2.1, 3.4, 1.8]$

Weighted mean calculation:

$$\begin{aligned}
\text{Numerator} &= 0.273 \times 2.1 + 0.183 \times 3.4 + 0.104 \times 1.8 \\
&= 0.573 + 0.622 + 0.187 = 1.382 \\
\text{Denominator} &= 0.273 + 0.183 + 0.104 = 0.560 \\
\mu &= \frac{1.382}{0.560} \approx 2.468
\end{aligned}$$

Variance estimation:

$$\begin{aligned}
\sigma^2 &= \frac{1}{\sum w_i + 1} = \frac{1}{0.560 + 1} = \frac{1}{1.560} \approx 0.641 \\
\sigma &= \sqrt{0.641} \approx 0.801
\end{aligned}$$

Variance interpretation:

- More nearby points $\rightarrow$ lower variance (more certain)
- Fewer/distant points $\rightarrow$ higher variance (less certain)
- $+1$ in denominator prevents division by zero

Final acquisition value:

$$\alpha(\vec{x}) = \mu + \kappa \cdot \sigma = 2.468 + 2.0 \times 0.801 = 2.468 + 1.602 = 4.070$$

The local statistics are computed as:

$$\mu(\vec{x}) = \frac{\sum_{i=1}^{n} w_i y_i}{\sum_{i=1}^{n} w_i}, \quad \sigma^2(\vec{x}) = \frac{1}{\sum_{i=1}^{n} w_i + 1} \tag{17}$$

Algorithm 2 Acquisition Function Algorithm

```
function AcquisitionFunction(x, exploration_weight)
    if $self.X\_observed$ is empty then
        return $exploration\_weight$                    ▷ Pure exploration initially
    end if
    $distances \leftarrow [\ ]$
    for each $x\_obs$ in $self.X\_observed$ do
        $d \leftarrow \|x - x\_obs\|$                  ▷ Euclidean distance
        $distances.append(d)$
    end for
    $weights \leftarrow [\exp(-d)$ for $d$ in $distances]$
    $weight\_sum \leftarrow \sum(weights)$
    if $weight\_sum > 0$ then
        $mean \leftarrow \sum(weights[i] \times self.y\_observed[i])/weight\_sum$
        $variance \leftarrow 1.0/(weight\_sum + 1)$
    else
        $mean \leftarrow self.gp\_mean$                ▷ Fallback to prior
        $variance \leftarrow self.gp\_std^2$
    end if
    return $mean + exploration\_weight \times \sqrt{variance}$
end function
```

3.4 Algorithm Implementation

3.5 Exploration-Exploitation Trade-off

Example: Exploration Weight Sensitivity Analysis

Effect of exploration weight κ:

κ value	Exploration	Exploitation	Typical Use
0.1	Very Low	Very High	Final refinement
0.5	Low	High	Mature optimization
1.0	Balanced	Balanced	General purpose
2.0	High	Low	Early exploration
5.0	Very High	Very Low	Pure exploration

Scenario analysis:

- $\kappa = 0.1$: Focuses on best predicted regions, may miss global optima
- $\kappa = 2.0$: Balanced search, explores promising uncertain regions
- $\kappa = 5.0$: Extensive exploration, good for multi-modal functions

Adaptive strategies:

- Start with high κ, decrease over time
- Use problem-dependent tuning
- Employ bandit algorithms for κ selection

Theoretical optimality: $\kappa = \sqrt{2\log(t)}$ for UCB in bandits.

The exploration weight controls the trade-off:

$$\text{Exploration} \propto \kappa, \quad \text{Exploitation} \propto \frac{1}{\kappa} \tag{18}$$

3.6 Distance Metric Properties

Example: Euclidean Distance Analysis

Euclidean distance properties:

$$d(\vec{x}, \vec{x}') = \|\vec{x} - \vec{x}'\| = \sqrt{\sum_{j=1}^{D} (x_j - x_j')^2} \tag{19}$$

Scale sensitivity:

- Sensitive to different scales across dimensions
- Large-range dimensions dominate distance calculations
- Normalization often required for good performance

Alternative distance metrics:

- **Mahalanobis**: Accounts for correlation structure
- **Manhattan**: L_1 distance, robust to outliers
- **Cosine**: Angle-based similarity for high dimensions
- **Kernel-based**: Directly use GP covariance

Exponential kernel properties:

$$\begin{aligned} k(d) &= e^{-d} \\ k(0) &= 1 \quad \text{(maximum weight at zero distance)} \\ \lim_{d\to\infty} k(d) &= 0 \quad \text{(zero weight at infinite distance)} \\ \text{Length scale :} &\quad \text{Implicitly 1 in our implementation} \end{aligned}$$

Effective neighborhood radius: Points beyond distance 3 have weight < 0.05.

The distance metric follows:

$$d(\vec{x}, \vec{x}') = \|\vec{x} - \vec{x}'\|_2 = \sqrt{\sum_{j=1}^{D} (x_j - x'_j)^2} \tag{20}$$

3.7 Edge Cases and Robustness

Example: Boundary Condition Handling

No observations case:

- Returns pure exploration value: $\alpha(\vec{x}) = \kappa$
- Encourages initial exploration throughout space
- Prevents division by zero errors

Zero weight sum case:

- All points are extremely far away
- Use global prior: $\mu = \text{gp\_mean}$, $\sigma = \text{gp\_std}$
- Ensures reasonable behavior in unexplored regions

Numerical stability:

- Exponential avoids negative weights
- $+1$ in variance denominator prevents division by zero
- Euclidean distance always non-negative

Degenerate cases:

- Identical points: Weight $= 1$, normal calculation
- Very close points: High weights, low variance
- All points far away: Fallback to prior

Example: Completely unexplored region:

$$\begin{aligned}
\text{Weights} &\approx [0.001, 0.0005, 0.0002] \\
\text{Weight sum} &\approx 0.0017 \\
\mu &\approx \text{weighted average of distant points} \\
\sigma^2 &= 1/(0.0017 + 1) \approx 0.998 \\
\alpha(\vec{x}) &\approx \mu + \kappa \cdot 0.999 \approx \text{prior} + \kappa
\end{aligned}$$

The algorithm handles edge cases through:

$$\alpha(\vec{x}) = \begin{cases} \kappa & \text{if } n = 0 \\ \mu_{\text{local}} + \kappa \cdot \sigma_{\text{local}} & \text{if } \sum w_i > 0 \\ \mu_{\text{prior}} + \kappa \cdot \sigma_{\text{prior}} & \text{otherwise} \end{cases} \tag{21}$$

3.8 Computational Complexity

Example: Performance Analysis

Time complexity:

$$\begin{aligned}\text{Distance calculations}: &\quad O(n \cdot D)\\ \text{Weight calculations}: &\quad O(n)\\ \text{Mean calculation}: &\quad O(n)\\ \text{Total}: &\quad O(n \cdot D)\end{aligned}$$

Space complexity:

- Temporary arrays: $O(n)$ for distances and weights
- No persistent additional storage

For typical parameters:

- $n = 50$ observations, $D = 5$ dimensions
- Operations: $50 \times 5 = 250$ distance calculations
- Very efficient compared to full GP ($O(n^3)$)

Optimization considerations:

- Efficient for sequential decision making
- Suitable for real-time applications with moderate n
- Bottleneck: Distance calculations for large n

Comparison with full GP acquisition:

Method	**Time Complexity**	**Accuracy**
Our approach	$O(nD)$	Moderate
Full GP UCB	$O(n^3 + n^2 D)$	High
Sparse GP	$O(m^2 n + m^3)$	Good

The computational complexity is:

$$C(n, D) = O(n \cdot D) \tag{22}$$

3.9 Theoretical Properties

Example: Convergence Analysis

Local consistency:

- As $n \to \infty$ and points become dense, local mean converges to true function
- Variance estimate decreases with more nearby points
- Acquisition function becomes more accurate

Exploration guarantee:

$$\lim_{n\to\infty} \max_{\vec{x}\in\mathcal{X}} \alpha(\vec{x}) = \max_{\vec{x}\in\mathcal{X}} f(\vec{x}) \tag{23}$$

No-regret properties:

- Under certain conditions, achieves sublinear regret
- Formal guarantees for UCB-style acquisition functions
- Our approximation preserves key UCB properties

Smoothness properties:

- Acquisition function is continuous (exponential kernel)
- Differentiable almost everywhere
- Suitable for gradient-based optimization

Limitations of our approach:

- Not a proper Bayesian method (approximate)
- No correlation structure between points
- Simpler than full GP but less accurate

The theoretical properties include:

$$\forall \vec{x} \in \mathcal{X}, \quad \lim_{n\to\infty} \mu_{\text{local}}(\vec{x}) = f(\vec{x}), \quad \lim_{n\to\infty} \sigma_{\text{local}}(\vec{x}) = 0 \tag{24}$$

3.10 Practical Usage Guidelines

Example: Parameter Tuning and Application

Choosing exploration weight:

- **Default**: $\kappa = 2.0$ (balanced exploration-exploitation)
- **Exploration-heavy**: $\kappa = 3.0 - 5.0$ for multi-modal functions
- **Exploitation-heavy**: $\kappa = 0.5 - 1.0$ for unimodal functions
- **Adaptive**: Decrease κ over iterations

Normalization recommendations:

- Normalize search space to $[0, 1]^D$ or $[-1, 1]^D$
- Normalize function values to zero mean, unit variance
- Improves distance metric performance

Integration with optimization:

```
def suggest_next_point(self, exploration_weight=2.0):
    # Optimize acquisition function over search space
    best_x = None
    best_acq = -float('inf')

    for candidate in generate_candidates():
        acq_value = self.acquisition_function(
            candidate, exploration_weight)
        if acq_value > best_acq:
            best_acq = acq_value
            best_x = candidate

    return best_x
```

Monitoring acquisition landscape:

- Plot acquisition function to understand decision process
- Check if acquisition is too flat (need more exploration)
- Verify diversity of selected points

The acquisition function enables efficient global optimization through:

$$\vec{x}_{n+1} = \arg\max_{\vec{x}\in\mathcal{X}} \alpha(\vec{x}) \tag{25}$$

4 Next Point Suggestion Algorithm

4.1 Concept of Candidate Selection

Example: Sequential Decision Making in Bayesian Optimization

Scenario: We have observed 5 points and want to choose the 6th evaluation point
Current knowledge:

$$\begin{aligned} X_{\text{obs}} &= [[-3, 2], [0, -1], [2, 3], [-1, -2], [1, 0]] \\ y_{\text{obs}} &= [4.1, 2.3, 5.7, 1.8, 3.2] \end{aligned}$$

Process:

1. Generate 1000 random candidate points in $[-5, 5] \times [-3, 3]$
2. Evaluate acquisition function for each candidate
3. Select candidate with highest acquisition value
4. Example result: $\vec{x}^* = [3.8, -2.1]$ with $\alpha = 7.3$

Interpretation: The algorithm suggests evaluating at a point that balances high predicted value and high uncertainty based on current model.

The next point suggestion algorithm implements the core decision-making process in Bayesian optimization, selecting the most promising point to evaluate next based on the acquisition function.

4.2 Mathematical Foundation

4.2.1 Optimization Problem Formulation

Example: Acquisition Maximization Problem

Formal optimization problem:

$$\vec{x}_{n+1} = \arg\max_{\vec{x} \in \mathcal{X}} \alpha(\vec{x}) \tag{26}$$

Where:

- $\mathcal{X}$: Search space defined by bounds
- $\alpha(\vec{x})$: Acquisition function value at $\vec{x}$
- $\vec{x}_{n+1}$: Next point to evaluate

Properties of the problem:

- Non-convex and potentially multi-modal
- Expensive to evaluate (requires GP predictions)
- No analytical gradient available in general
- Requires global optimization approach

Random search approach:

$$\vec{x}_{n+1} \approx \arg\max_{\vec{x}^{(i)} \sim U(\mathcal{X}), i=1,\ldots,N} \alpha(\vec{x}^{(i)}) \tag{27}$$

Theoretical guarantee: As $N \to \infty$, random search finds global optimum of acquisition function.

The core optimization problem is:

$$\vec{x}^* = \arg\max_{\vec{x} \in \mathcal{X}} \alpha(\vec{x}) \tag{28}$$

where $\mathcal{X} = [a_1, b_1] \times [a_2, b_2] \times \cdots \times [a_D, b_D]$.

4.3 Algorithm Step-by-Step

4.3.1 Random Candidate Generation

Example: Uniform Sampling Process

For bounds $= [(-5,5),(-3,3)]$:
Candidate generation:

$$\begin{aligned} x_1 &\sim U(-5,5) \\ x_2 &\sim U(-3,3) \\ \vec{x} &= [x_1, x_2] \end{aligned}$$

Sampling properties:

- Each dimension sampled independently
- Uniform distribution ensures space coverage
- Expected number in subregion proportional to volume

Example candidates:

$$\begin{aligned} \vec{x}^{(1)} &= [2.3, -1.8] \\ \vec{x}^{(2)} &= [-4.1, 2.7] \\ \vec{x}^{(3)} &= [0.5, 0.2] \\ &\vdots \\ \vec{x}^{(1000)} &= [3.8, -2.1] \end{aligned}$$

Coverage analysis: With 1000 points in 2D space of area 60, expected nearest-neighbor distance ≈ 0.25.

Candidates are generated as:

$$x_j \sim U(a_j, b_j) \quad \text{for } j = 1, \ldots, D \tag{29}$$

4.3.2 Acquisition Evaluation and Selection

Example: Sequential Evaluation Process

Initial state: $best\_x = \text{None}$, $best\_acq = -\infty$
Iteration 1:

- Candidate: $\vec{x} = [2.3, -1.8]$
- Acquisition: $\alpha = 3.2$
- Update: $best\_x = [2.3, -1.8]$, $best\_acq = 3.2$

Iteration 2:

- Candidate: $\vec{x} = [-4.1, 2.7]$
- Acquisition: $\alpha = 2.8$
- No update $(2.8 < 3.2)$

Iteration 3:

- Candidate: $\vec{x} = [0.5, 0.2]$
- Acquisition: $\alpha = 1.5$
- No update

Iteration 500:

- Candidate: $\vec{x} = [3.8, -2.1]$
- Acquisition: $\alpha = 7.3$
- Update: $best\_x = [3.8, -2.1]$, $best\_acq = 7.3$

Final result: Return $[3.8, -2.1]$ as next evaluation point.

Algorithm 3 Next Point Suggestion Algorithm

```
function SuggestNext
    $best\_x \leftarrow$ None                                  ▷ Initialize best candidate
    $best\_acq \leftarrow -\infty$                             ▷ Initialize best acquisition value
    for $i \leftarrow 1$ to 1000 do                            ▷ Random search over candidates
        $x \leftarrow [\,]$                                    ▷ New candidate point
        for $(low, high)$ in $self.bounds$ do
            $x_j \leftarrow$ random.uniform$(low, high)$       ▷ Sample dimension
            $x.append(x_j)$
        end for
        $x \leftarrow$ numpy.array$(x)$                        ▷ Convert to array
        $acq \leftarrow self.acquisition\_function(x)$         ▷ Evaluate acquisition
        if $acq > best\_acq$ then                              ▷ Update if improvement
            $best\_acq \leftarrow acq$
            $best\_x \leftarrow x$
        end if
    end for
    return $best\_x$                                           ▷ Return best candidate found
end function
```

4.4 Random Search Properties

Example: Random Search Analysis

Probability of finding near-optimal solution:

$$P(\text{find } \epsilon\text{-optimal}) = 1 - \left(1 - \frac{V_\epsilon}{V_{\text{total}}}\right)^N$$

$$\begin{aligned} &\text{Where:} \\ V_\epsilon &= \text{Volume of } \epsilon\text{-optimal region} \\ V_{\text{total}} &= \text{Total search space volume} \\ N &= \text{Number of candidates (1000)} \end{aligned}$$

For different dimensions:

- **1D**: With 1000 points, high probability of finding near-optimum
- **2D**: Good coverage, reasonable probability
- **5D**: Sparse coverage, may miss optima
- **10D+:** Very sparse, likely to miss optima

Expected best acquisition value:

$$\mathbb{E}[\max_{i=1}^{N} \alpha(\vec{x}^{(i)})] \approx \max_{\vec{x}\in\mathcal{X}} \alpha(\vec{x}) - O\left(\frac{1}{N^{1/D}}\right) \tag{30}$$

Curse of dimensionality: Required N grows exponentially with D.

The random search performance follows:

$$\mathbb{E}[\alpha(\vec{x}^*_{\text{found}})] \geq \max_{\vec{x}\in\mathcal{X}} \alpha(\vec{x}) - C \cdot N^{-1/D} \tag{31}$$

4.5 Computational Complexity

Example: Performance Analysis

Time complexity:

$$\begin{aligned} \text{Candidate generation}: &\quad O(N \cdot D) \\ \text{Acquisition evaluations}: &\quad O(N \cdot n \cdot D) \\ \text{Total}: &\quad O(N \cdot n \cdot D) \end{aligned}$$

Where:

- $N = 1000$: Number of random candidates
- n: Number of observed points
- D: Problem dimension

For typical parameters:

- $N = 1000$, $n = 20$, $D = 2$
- Operations: $1000 \times 20 \times 2 = 40,000$ distance calculations
- Time: ≈ 0.1 seconds (reasonable for expensive functions)

Memory complexity:

- Temporary storage: $O(D)$ per candidate
- No persistent additional memory
- Very memory efficient

Bottleneck analysis:

- Acquisition function evaluation is the bottleneck
- Candidate generation is negligible
- Linear scaling with number of observations

The computational complexity is:

$$C(N, n, D) = O(N \cdot n \cdot D) \tag{32}$$

4.6 Parameter Selection: Number of Candidates

Example: Candidate Count Trade-offs

Fixed at 1000 candidates:
Advantages:

- Predictable computation time
- Consistent performance across runs
- Simple implementation
- Works well for low to moderate dimensions

Disadvantages:

- May be insufficient for high dimensions
- Could be excessive for very simple problems
- No adaptation to problem difficulty

Adaptive alternatives:

- **Dimension-based**: $N = 1000 \cdot D$
- **Convergence-based**: Stop when improvement $<$ threshold
- **Time-based**: Fixed time budget instead of candidate count

Empirical guidelines:

Dimension	Recommended N	Success Rate
1-2	500-1000	>95%
3-5	1000-5000	80-95%
6-10	5000-20000	50-80%
>10	>20000	<50%

Our choice: $N = 1000$ works well for typical Bayesian optimization problems (D $\leq$ 10).

The candidate count follows the rule:

$$N = 1000 \quad \text{(fixed for simplicity and predictability)} \tag{33}$$

4.7 Comparison with Alternative Methods

Example: Optimization Strategy Comparison

Random search (our approach):

- **Pros**: Simple, robust, parallelizable, no derivatives needed
- **Cons**: May miss narrow optima, inefficient for high dimensions
- **Use case**: Low to moderate dimensions, multi-modal acquisition

Gradient-based optimization:

- **Pros**: Fast convergence, efficient for high dimensions
- **Cons**: Requires gradients, may get stuck in local optima
- **Use case**: When acquisition function is smooth and differentiable

Multi-start local optimization:

- **Pros**: Combines global and local search
- **Cons**: More complex, requires good starting points
- **Use case**: When acquisition has multiple local optima

Evolutionary algorithms:

- **Pros**: Good global search, no gradient needed
- **Cons**: Many function evaluations, parameter tuning
- **Use case**: Complex multi-modal landscapes

Performance comparison:

Method	**Speed**	**Accuracy**	**Robustness**
Random Search	Medium	Medium	High
Gradient-based	Fast	High	Low
Multi-start	Slow	High	Medium
Evolutionary	Slow	Medium	High

4.8 Theoretical Convergence Properties

Example: Global Optimization Guarantees

Probability of success:

$$P(\text{success}) = 1 - \left(1 - \frac{V_{\text{optimal}}}{V_{\text{total}}}\right)^N$$

Where V_{optimal} = Volume of region where $\alpha(\vec{x}) \geq \alpha^* - \epsilon$

For acquisition functions:

- UCB-style: Optimal region typically has reasonable volume
- EI-style: May have very small optimal regions
- Our method: Works well for UCB, may struggle with peaked EI

Asymptotic behavior:

$$\lim_{N \to \infty} P\left(\alpha(\vec{x}^*_{\text{found}}) \geq \max_{\vec{x} \in \mathcal{X}} \alpha(\vec{x}) - \epsilon\right) = 1 \tag{34}$$

Rate of convergence:

$$\mathbb{E}[\alpha^* - \alpha(\vec{x}^*_{\text{found}})] = O\left(N^{-1/D}\right) \tag{35}$$

Practical implication: For $D \leq 5$, 1000 candidates usually sufficient.

The convergence guarantee is:

$$\lim_{N \to \infty} P\left(\|\vec{x}^*_{\text{found}} - \vec{x}^*_{\text{true}}\| < \delta\right) = 1 \quad \forall \delta > 0 \tag{36}$$

4.9 Implementation Considerations

Example: Practical Implementation Details

Initialization handling:

- When no observations: acquisition function returns constant
- All candidates equally good initially
- First suggestion is essentially random
- Reasonable behavior for cold start

Numerical stability:

- Initial $best\_acq = -\infty$ ensures first candidate always accepted
- Floating-point comparisons handle ties correctly
- Array conversion ensures consistent data types

Parallelization potential:

```
# Parallel version using multiprocessing
def suggest_next_parallel(self, n_workers=4):
    with multiprocessing.Pool(n_workers) as pool:
        candidates = [self._generate_candidate()
                      for _ in range(1000)]
        acq_values = pool.map(self.acquisition_function,
                              candidates)
        best_idx = np.argmax(acq_values)
        return candidates[best_idx]
```

Alternative sampling strategies:

- **Latin Hypercube**: Better space-filling properties
- **Sobol sequences**: Quasi-random for improved coverage
- **Adaptive sampling**: Focus on promising regions
- **Hybrid approaches**: Combine random with local search

4.10 Integration with Bayesian Optimization Loop

Example: Complete Optimization Workflow

Complete Bayesian optimization cycle:

1. **Initialize**: Create Bayesian optimizer with bounds
2. **Suggest**: Call `suggest_next()` to get candidate point
3. **Evaluate**: Compute expensive function at suggested point
4. **Update**: Add observation to GP model
5. **Repeat**: Until budget exhausted or convergence

Code example:

```
def optimize(self, objective_func, n_iterations=50):
    for i in range(n_iterations):
        # Suggest next point to evaluate
        x_next = self.suggest_next()

        # Evaluate expensive objective function
        y_next = objective_func(x_next)

        # Update model with new observation
        self.X_observed.append(x_next)
        self.y_observed.append(y_next)

        print(f"Iteration {i}: x={x_next}, y={y_next}")

    return self.get_best_observation()
```

Convergence monitoring:

- Track best objective value over iterations
- Monitor acquisition function values
- Check for stagnation in improvements

Stopping criteria:

- Maximum iterations reached
- No improvement for several iterations
- Acquisition values become very small
- Computational budget exhausted

The method integrates into the Bayesian optimization framework as:

$$\vec{x}_{n+1} = \text{SuggestNext}(), \quad y_{n+1} = f(\vec{x}_{n+1}), \quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(\vec{x}_{n+1}, y_{n+1})\} \tag{37}$$

4.11 Performance Optimization Strategies

Example: Efficiency Improvements

Candidate generation optimizations:

- **Vectorized generation**: Generate all candidates at once
- **Memory pre-allocation**: Pre-allocate candidate array
- **Batch acquisition**: Evaluate multiple candidates simultaneously

Acquisition function optimizations:

- **Caching**: Cache distances for repeated calculations
- **Approximations**: Use approximate nearest neighbors
- **Early stopping**: Stop if acquisition clearly suboptimal

Adaptive candidate count:

```
def suggest_next_adaptive(self):
    # Start with small N, increase if needed
    for N in [100, 500, 1000, 2000]:
        best_x = self._random_search(N)
        if self._is_confident(best_x):
            return best_x
    return best_x
```

Warm start strategies:

- Use previous best candidates as starting points
- Focus search around promising regions from past iterations
- Maintain diversity to avoid getting stuck

Hybrid approaches:

- Random search for global exploration
- Local optimization around best candidates for refinement
- Balance computation between global and local search

The optimization can be enhanced through:

$$N_{\text{effective}} = N_{\text{global}} + N_{\text{local}} \cdot K_{\text{refinement}} \tag{38}$$

5 Bayesian Optimizer Update Algorithm

5.1 Concept of Sequential Model Updates

Example: Incremental Learning Process

Scenario: We have 3 observations and receive a new data point
Current state:

$$\begin{aligned} X_{\text{obs}} &= [[-2,1],[0,-1],[3,2]] \\ y_{\text{obs}} &= [3.2, 1.8, 4.5] \\ \mu &= 3.17, \quad \sigma = 1.10 \end{aligned}$$

New observation: $\vec{x}_{\text{new}} = [1,0]$, $y_{\text{new}} = 2.8$
Update process:

1. Append new point: $X_{\text{obs}} \leftarrow [[-2,1],[0,-1],[3,2],[1,0]]$
2. Append new value: $y_{\text{obs}} \leftarrow [3.2, 1.8, 4.5, 2.8]$
3. Recompute mean: $\mu_{\text{new}} = \frac{3.2+1.8+4.5+2.8}{4} = 3.075$
4. Recompute std: $\sigma_{\text{new}} = \sqrt{\frac{(3.2-3.075)^2+\cdots+(2.8-3.075)^2}{3}} \approx 0.98$

Result: Model now reflects all 4 observations with updated statistics.

The update method incorporates new observations into the Bayesian optimization model, refining the Gaussian process prior parameters based on accumulated data.

5.2 Mathematical Foundation

5.2.1 Sequential Bayesian Learning

Example: Bayesian Update Mathematics

Bayesian learning framework:

$$\begin{aligned} \text{Prior} : &\quad p(\theta) \\ \text{Likelihood} : &\quad p(\mathcal{D}|\theta) \\ \text{Posterior} : &\quad p(\theta|\mathcal{D}) \propto p(\mathcal{D}|\theta) \cdot p(\theta) \end{aligned}$$

For Gaussian process:

- $\theta = (\mu, \sigma)$: GP hyperparameters
- $\mathcal{D} = \{(\vec{x}_i, y_i)\}_{i=1}^n$: Observed data
- Our approach: Uses empirical Bayes (point estimates)

Mean update formula:

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} y_i = \frac{(n-1)\mu_{n-1} + y_n}{n} \tag{39}$$

Standard deviation update:

$$\sigma_n = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \mu_n)^2} \quad \text{for } n \geq 2 \tag{40}$$

Sequential computation:

$$S_n = \sum_{i=1}^{n}(y_i - \mu_n)^2$$

$$\sigma_n = \sqrt{\frac{S_n}{n-1}}$$

The Bayesian update follows the empirical Bayes approach:

$$\mu_n = \mathbb{E}[y|\mathcal{D}_n], \quad \sigma_n = \sqrt{\mathrm{Var}[y|\mathcal{D}_n]} \tag{41}$$

5.3 Algorithm Step-by-Step

5.3.1 Data Incorporation

Example: Observation Storage Process

Input: $\vec{x}_{\text{new}} = [1.5, -0.8]$, $y_{\text{new}} = 2.3$
Before update:

$$\begin{aligned} X_{\text{observed}} &= [[0.0, 0.0], [2.0, 1.0]] \\ y_{\text{observed}} &= [1.8, 3.2] \\ \text{Storage size :} &\quad n = 2 \end{aligned}$$

Update step 1 - Append input:

$$\begin{aligned} X_{\text{observed}} &\leftarrow [[0.0, 0.0], [2.0, 1.0], [1.5, -0.8]] \\ \text{Storage :} &\quad \text{Now contains 3 points} \end{aligned}$$

Update step 2 - Append output:

$$\begin{aligned} y_{\text{observed}} &\leftarrow [1.8, 3.2, 2.3] \\ \text{Storage :} &\quad \text{Now contains 3 values} \end{aligned}$$

Data structure properties:

- Maintains complete evaluation history
- Enables model retraining if needed
- Supports analysis of optimization progress

5.3.2 Statistical Parameter Updates

Example: Incremental Statistics Calculation

Before update:

$$\begin{aligned} &n = 2 \\ &\mu = \frac{1.8 + 3.2}{2} = 2.5 \\ &\sigma = 1.0 \quad \text{(since } n \leq 2\text{)} \end{aligned}$$

After adding $y_3 = 2.3$:
Mean calculation:

$$\mu_{\text{new}} = \frac{1.8 + 3.2 + 2.3}{3} = \frac{7.3}{3} \approx 2.433$$

Standard deviation calculation:

$$\begin{aligned} \text{Variance} &= \frac{(1.8 - 2.433)^2 + (3.2 - 2.433)^2 + (2.3 - 2.433)^2}{2} \\ &= \frac{(-0.633)^2 + (0.767)^2 + (-0.133)^2}{2} \\ &= \frac{0.401 + 0.588 + 0.018}{2} = \frac{1.007}{2} \approx 0.504 \\ \sigma_{\text{new}} &= \sqrt{0.504} \approx 0.710 \end{aligned}$$

Model evolution: Statistics now reflect all 3 observations.

Algorithm 4 Bayesian Optimizer Update Algorithm

```
1: function UPDATE(x, y)
2:     $self.X\_observed.append(x)$                         ▷ Store input point
3:     $self.y\_observed.append(y)$                         ▷ Store function value
4:     if $self.y\_observed$ is not empty then
5:         $n \leftarrow$ len($self.y\_observed$)
6:         $self.gp\_mean \leftarrow$ mean($self.y\_observed$)          ▷ Update mean
7:         if $n > 1$ then
8:             $self.gp\_std \leftarrow$ std($self.y\_observed$)        ▷ Update std with Bessel's correction
9:         else
10:            $self.gp\_std \leftarrow 1.0$                   ▷ Default std for single point
11:        end if
12:    end if
13: end function
```

5.4 Statistical Properties

5.4.1 Mean Update Properties

Example: Mean Behavior Analysis

Sequential mean formula:

$$\mu_n = \frac{(n-1)\mu_{n-1} + y_n}{n} \tag{42}$$

Properties:

- **Unbiased estimator**: $\mathbb{E}[\mu_n] = \mathbb{E}[y]$
- **Consistent**: $\lim_{n\to\infty} \mu_n = \mathbb{E}[y]$ (almost surely)
- **Variance**: $\text{Var}(\mu_n) = \frac{\sigma^2}{n}$

Convergence rate:

$$|\mu_n - \mathbb{E}[y]| = O\left(\frac{1}{\sqrt{n}}\right) \tag{43}$$

Example progression:

n	y_n	μ_n	Error ($\lvert\mu_n - 3.0\rvert$)
1	2.5	2.500	0.500
2	3.2	2.850	0.150
3	3.1	2.933	0.067
4	2.9	2.925	0.075
5	3.0	2.940	0.060
10	3.2	2.980	0.020

Robustness: Mean is sensitive to outliers but converges to true mean.

The mean estimator satisfies:

$$\mu_n = \frac{1}{n}\sum_{i=1}^{n} y_i, \quad \mathbb{E}[\mu_n] = \mu, \quad \text{Var}(\mu_n) = \frac{\sigma^2}{n} \tag{44}$$

5.4.2 Standard Deviation Update Properties

Example: Variance Estimation Analysis

Sample standard deviation:

$$s_n = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \mu_n)^2} \tag{45}$$

Properties:

- **Unbiased**: $\mathbb{E}[s_n^2] = \sigma^2$ (but s_n is biased)
- **Consistent**: $\lim_{n\to\infty} s_n = \sigma$
- **Bessel's correction**: $\frac{1}{n-1}$ instead of $\frac{1}{n}$

Small sample behavior:

- $n = 1$: Standard deviation undefined, use default 1.0
- $n = 2$: First meaningful variance estimate
- $n \geq 3$: Reliable variance estimation

Example progression:

$$\begin{aligned} n = 1: &\quad s = 1.0 \quad \text{(default)} \\ n = 2: &\quad s = \sqrt{\frac{(y_1-\mu)^2 + (y_2-\mu)^2}{1}} \\ n = 3: &\quad s = \sqrt{\frac{(y_1-\mu)^2 + (y_2-\mu)^2 + (y_3-\mu)^2}{2}} \end{aligned}$$

Convergence: $s_n \to \sigma$ as $n \to \infty$.

The standard deviation estimator follows:

$$s_n = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(y_i - \mu_n)^2}, \quad \mathbb{E}[s_n^2] = \sigma^2 \tag{46}$$

5.5 Computational Complexity

Example: Update Cost Analysis

Time complexity per update:

$$\begin{aligned}
\text{Append operations}: &\quad O(1) \quad \text{(amortized)} \\
\text{Mean calculation}: &\quad O(n) \\
\text{Standard deviation}: &\quad O(n) \\
\text{Total}: &\quad O(n)
\end{aligned}$$

Space complexity:

$$\begin{aligned}
\text{Data storage}: &\quad O(n \cdot D) \quad \text{for } X\_observed \\
&\quad O(n) \quad \text{for } y\_observed \\
\text{Total}: &\quad O(n \cdot D)
\end{aligned}$$

Cumulative cost over N updates:

$$\begin{aligned}
\text{Total time}: &\quad \sum_{n=1}^{N} O(n) = O(N^2) \\
\text{For } N = 100: &\quad \approx 5000 \text{ operations} \\
\text{For } N = 1000: &\quad \approx 500,000 \text{ operations}
\end{aligned}$$

Optimization potential:

- Use incremental formulas to avoid recomputation
- Maintain running sums for mean and variance
- Consider periodic recomputation for large n

Practical considerations: For typical BO with $n \leq 100$, cost is negligible.

The computational complexity is:

$$C(n) = O(n) \quad \text{per update}, \quad C_{\text{total}}(N) = O(N^2) \tag{47}$$

5.6 Incremental Computation Formulas

Example: Efficient Sequential Updates

Incremental mean formula:

$$\mu_n = \mu_{n-1} + \frac{y_n - \mu_{n-1}}{n} \tag{48}$$

Incremental variance formula:

$$S_n = S_{n-1} + (y_n - \mu_{n-1})(y_n - \mu_n)$$
$$\sigma_n = \sqrt{\frac{S_n}{n-1}} \quad \text{for } n \geq 2$$

Implementation with incremental updates:

```
def update_incremental(self, x, y):
    self.X_observed.append(x)
    self.y_observed.append(y)

    n = len(self.y_observed)
    if n == 1:
        self.gp_mean = y
        self.M2 = 0  # Running sum of squares
        self.gp_std = 1.0
    else:
        # Incremental mean update
        delta = y - self.gp_mean
        self.gp_mean += delta / n

        # Incremental variance update
        delta2 = y - self.gp_mean
        self.M2 += delta * delta2

        # Update standard deviation
        if n > 1:
            self.gp_std = np.sqrt(self.M2 / (n - 1))
```

Benefits:

- Constant time per update: $O(1)$ instead of $O(n)$

- No need to recompute from all data
- Numerically stable

The incremental formulas are:

$$\mu_n = \mu_{n-1} + \frac{y_n - \mu_{n-1}}{n}, \quad S_n = S_{n-1} + (y_n - \mu_{n-1})(y_n - \mu_n) \tag{49}$$

5.7 Model Evolution and Convergence

Example: Learning Process Visualization

Initial state ($n = 0$):

- Prior: $\mu = 0$, $\sigma = 1$
- High uncertainty, no data

After first observation ($n = 1$):

- $\mu = y_1$, $\sigma = 1.0$ (default)
- Model centered around first observation
- Uncertainty remains high

After several observations ($n = 5$):

- $\mu \approx$ sample mean, $\sigma \approx$ sample std
- Model reflects observed data distribution
- Uncertainty decreasing

At convergence ($n = 50$):

- $\mu \approx \mathbb{E}[y]$, $\sigma \approx \sqrt{\text{Var}[y]}$
- Model accurately represents function statistics
- Low uncertainty estimates

Example trajectory:

n	μ	σ	Model State
0	0.000	1.000	Prior
1	2.500	1.000	First observation
5	2.940	0.850	Learning
10	3.020	0.720	Converging
20	2.990	0.690	Stable
50	3.005	0.701	Converged

The model evolution follows:

$$\lim_{n\to\infty} \mu_n = \mathbb{E}[y], \quad \lim_{n\to\infty} \sigma_n = \sqrt{\mathrm{Var}[y]} \tag{50}$$

5.8 Robustness and Edge Cases

Example: Boundary Condition Handling

Empty initial state:

- $n = 0$: No observations yet
- Update adds first data point
- Mean becomes y_1, std remains 1.0

Single observation ($n = 1$):

- Standard deviation undefined mathematically
- Use default value 1.0
- Prevents division by zero

Constant function:

- All $y_i = c$: $\mu = c$, $\sigma = 0$
- Acquisition function becomes purely exploitative
- Valid mathematical behavior

Outlier handling:

- Single outlier affects mean significantly
- Standard deviation increases with outliers
- Robust statistics could be used for noisy functions

Numerical stability:

- Avoid catastrophic cancellation in variance calculation
- Use numerically stable algorithms for large n
- Handle very small variances gracefully

Memory management:

- Data lists grow indefinitely
- Consider pruning for very long runs
- Alternative: Use fixed-size sliding window

The edge case handling includes:

$$\sigma_n = \begin{cases} 1.0 & \text{if } n \leq 1 \\ \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (y_i - \mu_n)^2} & \text{if } n \geq 2 \end{cases} \tag{51}$$

5.9 Integration with Bayesian Optimization

Example: Complete Optimization Loop Integration

Bayesian optimization workflow:

1. **Initialize**: Create optimizer with search space bounds
2. **Suggest**: Use acquisition function to pick next point
3. **Evaluate**: Compute expensive function at suggested point
4. **Update**: Call `update(x, y)` to incorporate new data
5. **Repeat**: Until convergence or budget exhausted

Code example:

```
def optimize(self, objective_func, n_iterations=50):
    for i in range(n_iterations):
        # Suggest next point using current model
        x_next = self.suggest_next()

        # Evaluate expensive black-box function
        y_next = objective_func(x_next)

        # Update model with new observation
        self.update(x_next, y_next)

        print(f"Iteration {i}: f({x_next}) = {y_next}")
        print(f"Current model: μ={self.gp_mean:.3f}, $"
              f"$\sigma$={self.gp_std:.3f}")

    return self.get_best_solution()
```

Model evolution during optimization:

- Early iterations: High uncertainty, exploratory behavior
- Middle iterations: Balancing exploration and exploitation
- Late iterations: Low uncertainty, refinement near optimum

Convergence detection:

- Monitor changes in model statistics
- Check for stabilization of best value
- Track acquisition function values

The update method completes the Bayesian optimization cycle:

$$\mathcal{D}_{n+1} = \mathcal{D}_n \cup \{(\vec{x}_{n+1}, y_{n+1})\}, \quad \theta_{n+1} = \text{Update}(\theta_n, \vec{x}_{n+1}, y_{n+1}) \tag{52}$$

5.10 Alternative Model Update Strategies

Example: Enhanced Update Methods

Full Gaussian process update:

- Update full covariance matrix, not just mean/std
- More accurate but computationally expensive
- Required for proper uncertainty quantification

Forgetting mechanisms:

```
def update_with_forgetting(self, x, y, forgetting_factor=0.95):
    # Add new observation
    self.X_observed.append(x)
    self.y_observed.append(y)

    # Apply forgetting to old observations
    if len(self.y_observed) > max_observations:
        self.X_observed.pop(0)
        self.y_observed.pop(0)

    # Update statistics
    self._update_statistics()
```

Robust statistics:

- Use median instead of mean for noisy functions
- Use MAD (median absolute deviation) instead of std
- More resistant to outliers

Adaptive prior updating:

- Start with uninformative prior
- Switch to empirical prior after sufficient data
- Balance prior knowledge and observed data

Batch updates:

- Process multiple observations simultaneously
- More efficient for parallel function evaluation
- Requires careful implementation

The update strategy can be enhanced through:

$$\theta_{n+1} = \alpha \cdot \theta_n + (1 - \alpha) \cdot \text{Update}(\theta_n, \mathcal{D}_{\text{new}}) \tag{53}$$

6 Reinforcement Learning Optimizer

6.1 Concept of Q-Learning for Optimization

Example: RL Optimization Metaphor

Scenario: Optimizing a black-box function using reinforcement learning
State representation: Discretized parameter space (e.g., temperature settings)
Actions: Parameter adjustments (e.g., increase/decrease temperature)
Rewards: Function improvement or quality metric
Simple example:

- **States**: 10 temperature levels from 0°C to 100°C
- **Actions**: Increase/Decrease temperature by 10°C
- **Reward**: Negative of objective function value
- **Goal**: Learn optimal temperature sequence

Q-learning process:

$$\begin{aligned} \text{State } s &: \text{Current parameter configuration} \\ \text{Action } a &: \text{Parameter adjustment} \\ \text{Reward } r &: \text{Quality of new configuration} \\ \text{Q-value } Q(s,a) &: \text{Expected cumulative reward} \end{aligned}$$

The optimizer treats parameter optimization as a sequential decision-making problem.

The Reinforcement Learning Optimizer implements Q-learning to solve optimization problems by framing them as Markov Decision Processes, where states represent parameter configurations and actions represent parameter adjustments.

6.2 Mathematical Foundation

6.2.1 Q-Learning Algorithm

Example: Q-Learning Mathematics

Q-learning update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{54}$$

Components:

- $\alpha = 0.1$: Learning rate (step size)
- $\gamma = 0.95$: Discount factor (future reward importance)
- r_{t+1}: Immediate reward after taking action
- $\max_a Q(s_{t+1}, a)$: Maximum future value estimate

Exploration vs Exploitation:

$$a_t = \begin{cases} \text{random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s_t, a) & \text{with probability } 1 - \epsilon \end{cases} \tag{55}$$

Bellman optimality equation:

$$Q^*(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right] \tag{56}$$

Convergence: Q-learning converges to optimal Q-values under certain conditions.

The Q-learning algorithm follows the update rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{57}$$

6.3 Class Architecture and Initialization

Example: RL Optimizer State Initialization

For `n_states=100, n_actions=50`:

$$\begin{aligned}
\text{n\_states} &= 100 \quad \text{(state space size)} \\
\text{n\_actions} &= 50 \quad \text{(action space size)} \\
\text{q\_table} &= \text{defaultdict}(\lambda : \text{np.zeros}(50)) \\
\text{experience\_replay} &= \text{deque}(\text{maxlen} = 1000) \\
\epsilon &= 0.1 \quad \text{(exploration rate)} \\
\alpha &= 0.1 \quad \text{(learning rate)} \\
\gamma &= 0.95 \quad \text{(discount factor)}
\end{aligned}$$

Initial Q-table:

- All Q-values initialized to 0
- Lazy initialization: States created on first access
- Sparse representation for large state spaces

Experience replay:

- Circular buffer of 1000 experiences
- Stores $(s, a, r, s', \text{done})$ tuples
- Enables sample reuse and correlation reduction

Parameter interpretation:

- Small ϵ: Mostly exploitation, little exploration
- Small α: Slow learning, stable updates
- High γ: Long-term planning, future rewards important

6.3.1 Q-Table Structure

Example: Q-Table Representation

Q-table as function: $Q : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$
Default dictionary behavior:

- Unseen states: Return zero vector of length `n_actions`
- Automatic initialization of new states
- Memory efficient for sparse state visits

Example Q-table entries:

$$\begin{aligned} Q[0] &= [0.0, 0.0, \ldots, 0.0] \quad \text{(50 zeros)} \\ Q[1] &= [0.0, 0.0, \ldots, 0.0] \\ Q[99] &= [0.0, 0.0, \ldots, 0.0] \end{aligned}$$

After learning:

$$\begin{aligned} Q[42] &= [1.2, 0.8, -0.3, \ldots, 2.1] \quad \text{(learned values)} \\ Q[15] &= [0.1, 3.2, 1.7, \ldots, 0.4] \end{aligned}$$

State-action value interpretation:

- $Q(s, a)$: Expected cumulative reward from taking action a in state s
- Positive: Good action, should be taken
- Negative: Bad action, should be avoided
- Zero: Unknown quality, needs exploration

The Q-table is represented as:

$$Q : \mathcal{S} \to \mathbb{R}^{|\mathcal{A}|}, \quad Q(s) = [Q(s, a_1), Q(s, a_2), \ldots, Q(s, a_{n\_actions})] \tag{58}$$

6.4 Algorithm Initialization

Algorithm 5 Reinforcement Learning Optimizer Initialization

```
1: function RL_Optimizer_Init(n_states, n_actions)
2:     $self.n\_states \leftarrow n\_states$                                   ▷ State space size
3:     $self.n\_actions \leftarrow n\_actions$                                 ▷ Action space size
4:     $self.q\_table \leftarrow$ defaultdict($\lambda$ : zeros($n\_actions$))   ▷ Q-value storage
5:     $self.experience\_replay \leftarrow$ deque(maxlen = 1000)             ▷ Experience buffer
6:     $self.epsilon \leftarrow 0.1$                                          ▷ Exploration rate
7:     $self.alpha \leftarrow 0.1$                                            ▷ Learning rate
8:     $self.gamma \leftarrow 0.95$                                           ▷ Discount factor
9: end function
```

6.5 Parameter Analysis and Selection

6.5.1 Exploration-Exploitation Trade-off (ϵ)

Example: Epsilon-Greedy Strategy Analysis

ϵ-greedy policy:

$$\pi(s) = \begin{cases} \text{uniform over } \mathcal{A} & \text{with probability } \epsilon \\ \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \end{cases} \tag{59}$$

Effect of ϵ values:

- $\epsilon = 0.0$: Pure exploitation (greedy policy)
- $\epsilon = 0.1$: Mostly exploitation, occasional exploration
- $\epsilon = 0.5$: Balanced exploration-exploitation
- $\epsilon = 1.0$: Pure exploration (random policy)

Typical schedules:

- Constant: Fixed $\epsilon = 0.1$ (our approach)
- Decaying: $\epsilon_t = \epsilon_0 \cdot e^{-kt}$
- Adaptive: Adjust based on learning progress

Theoretical considerations:

- Need sufficient exploration for convergence
- Too much exploration slows learning
- Optimal balance depends on problem

Our choice: $\epsilon = 0.1$ provides reasonable exploration while focusing on exploitation.

The exploration strategy follows:

$$\pi(s) = \begin{cases} \text{Random action} & \text{with probability } \epsilon \\ \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon \end{cases} \tag{60}$$

6.5.2 Learning Rate (α)

Example: Learning Rate Impact

Learning rate role:

$$\Delta Q = \alpha \cdot \text{Temporal Difference Error} \tag{61}$$

Effect of α values:

- $\alpha = 0.0$: No learning, Q-values never change
- $\alpha = 0.1$: Slow, stable learning (our choice)
- $\alpha = 0.5$: Moderate learning speed
- $\alpha = 1.0$: Complete overwrite, unstable

Convergence requirements:

- Must satisfy Robbins-Monro conditions: $\sum \alpha_t = \infty$, $\sum \alpha_t^2 < \infty$
- Constant α: Faster but may oscillate
- Decaying α: Slower but converges surely

Numerical stability:

- Small α prevents Q-value explosion
- Stable for stochastic environments
- Robust to reward scaling

Our choice: $\alpha = 0.1$ balances learning speed and stability.

The learning rate controls update magnitude:

$$|\Delta Q(s,a)| \propto \alpha, \quad \text{Stability} \propto \frac{1}{\alpha} \tag{62}$$

6.5.3 Discount Factor (γ)

Example: Discount Factor Interpretation

Discount factor meaning:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \tag{63}$$

Effect of γ values:

- $\gamma = 0.0$: Myopic, only care about immediate reward
- $\gamma = 0.95$: Farsighted, future rewards important (our choice)
- $\gamma = 0.99$: Very farsighted, long-term planning
- $\gamma = 1.0$: Infinite horizon, may not converge

Horizon analysis:

$$\text{Effective horizon} \approx \frac{1}{1-\gamma}$$
$$\gamma = 0.95 : \text{Horizon} \approx 20 \text{ steps}$$
$$\gamma = 0.99 : \text{Horizon} \approx 100 \text{ steps}$$

Optimization implications:

- High γ: Good for multi-step optimization strategies
- Low γ: Better for immediate reward maximization
- Our choice: $\gamma = 0.95$ suitable for medium-term planning

The discount factor determines planning horizon:

$$\text{Planning Horizon} \approx \frac{1}{1-\gamma}, \quad \text{Importance of step } k \propto \gamma^k \tag{64}$$

6.6 Experience Replay Mechanism

Example: Experience Replay Benefits

Experience tuple: $(s, a, r, s', \text{done})$
Replay buffer properties:

- Maximum size: 1000 experiences
- FIFO eviction when full
- Enables mini-batch learning
- Breaks temporal correlations

Usage pattern:

1. Collect experience $(s_t, a_t, r_{t+1}, s_{t+1}, \text{done})$
2. Store in replay buffer
3. Sample random mini-batch for learning
4. Update Q-values using sampled experiences

Benefits:

- **Sample efficiency**: Reuse experiences multiple times
- **Stability**: Reduce variance through averaging
- **Decorrelation**: Break sequential dependencies
- **Memory**: Learn from past experiences

Buffer size trade-off:

- Small buffer: Fast learning, may forget
- Large buffer: Stable, but slow to adapt
- Our choice: 1000 provides good balance

The experience replay stores:

$$\mathcal{D} = \{(s_i, a_i, r_i, s_i', \text{done}_i)\}_{i=1}^{1000} \tag{65}$$

6.7 State and Action Space Design

Example: Discrete Optimization Spaces

State representation strategies:

- **Discretization**: Continuous parameters $\rightarrow$ discrete bins
- **Binary encoding**: Bit representation of parameters
- **Feature-based**: Engineered state features
- **Raw parameters**: Direct parameter values

Action space design:

- **Parameter adjustments**: Δ for each parameter
- **Discrete steps**: Fixed increment/decrement
- **Relative changes**: Percentage adjustments
- **Absolute settings**: Direct parameter values

Example: 2D optimization:

$$\begin{aligned} \text{States} &: \quad 10 \times 10 \text{ grid } (n\_states = 100) \\ \text{Actions} &: \quad \{\text{up, down, left, right, stay}\}(n\_actions = 5) \\ \text{Reward} &: \quad -\text{objective function value} \end{aligned}$$

Dimensionality considerations:

- State space grows exponentially with dimensions
- Action space typically grows linearly
- Curse of dimensionality for high-dimensional problems

Our parameters: 100 states and 50 actions suitable for moderate complexity.

The state and action spaces follow:

$$|\mathcal{S}| = n\_states, \quad |\mathcal{A}| = n\_actions \tag{66}$$

6.8 Theoretical Convergence Properties

Example: Q-Learning Convergence Analysis

Convergence theorem: Q-learning converges to optimal Q^* with probability 1 if:

- All state-action pairs visited infinitely often
- Learning rates satisfy Robbins-Monro conditions
- The environment is a finite MDP

Robbins-Monro conditions:

$$\sum_{t=1}^{\infty} \alpha_t = \infty \quad \text{(infinite updates)}$$
$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty \quad \text{(finite variance)}$$

Our parameter compliance:

- Constant $\alpha = 0.1$: Violates Robbins-Monro ($\sum \alpha_t^2 = \infty$)
- Practical compromise: Works well in practice
- True convergence requires decaying learning rate

Convergence rate:

$$\|Q_t - Q^*\| \leq \gamma^t \|Q_0 - Q^*\| + O(\alpha) \tag{67}$$

Practical convergence:

- Typically requires thousands of episodes
- Depends on state space size and exploration
- Our setup: Suitable for moderate problems

The convergence guarantee requires:

$$\lim_{t\to\infty} Q_t(s,a) = Q^*(s,a) \quad \text{for all } s,a \tag{68}$$

6.9 Implementation Considerations

Example: Memory and Computation Analysis

Memory requirements:

$$\begin{aligned}\text{Q-table}: &\quad O(|\mathcal{S}| \cdot |\mathcal{A}|) \text{ floats}\\ \text{Experience replay}: &\quad O(1000 \cdot (2|\mathcal{S}| + 2)) \text{ floats}\\ \text{For our parameters}: &\quad 100 \times 50 + 1000 \times 202 \approx 207,000 \text{ floats}\\ &\quad \approx 1.66 \text{ MB (8-byte floats)}\end{aligned}$$

Time complexity:

$$\begin{aligned}\text{Q-update}: &\quad O(|\mathcal{A}|) \quad \text{per step}\\ \text{Action selection}: &\quad O(|\mathcal{A}|) \quad \text{per step}\\ \text{Experience storage}: &\quad O(1) \quad \text{amortized}\end{aligned}$$

Numerical considerations:

- Q-values initialized to 0 (optimistic initialization)
- Floating-point precision adequate for most applications
- Experience replay prevents catastrophic forgetting

Scalability:

- Works well for $|\mathcal{S}| \leq 10,000$, $|\mathcal{A}| \leq 1000$
- For larger spaces: Use function approximation
- Deep Q-networks for high-dimensional problems

Alternative representations:

```
# For continuous state spaces
class ContinuousRLOptimizer:
    def __init__(self, state_dim, n_actions):
        self.q_network = NeuralNetwork(state_dim, n_actions)
        self.target_network = NeuralNetwork(state_dim, n_actions)
        self.experience_replay = ReplayBuffer(10000)
```

The computational complexity is:

$$C_{\text{update}} = O(|\mathcal{A}|), \quad C_{\text{selection}} = O(|\mathcal{A}|) \tag{69}$$

6.10 Application to Optimization Problems

Example: Optimization as RL Problem

Mapping optimization to RL:

$$\begin{aligned} \text{State } s_t &: \text{Current parameter vector } \vec{x}_t \\ \text{Action } a_t &: \text{Parameter adjustment } \Delta\vec{x} \\ \text{Reward } r_t &: f(\vec{x}_t) - f(\vec{x}_{t-1}) \quad \text{(improvement)} \\ \text{Done} &: \text{Maximum iterations or convergence} \end{aligned}$$

State discretization example:

- Continuous parameter $x \in [0, 1]$
- Discretize into 100 states: $s = \lfloor 100 \cdot x \rfloor$
- Actions: $\{-0.1, -0.01, 0, +0.01, +0.1\}$
- Reward: Negative of function value

Advantages over gradient methods:

- No gradient information required
- Handles non-differentiable objectives
- Can escape local optima through exploration
- Learns optimization strategy

Limitations:

- Curse of dimensionality for high-dimensional problems
- Requires careful state-action design
- Slower convergence than specialized optimizers

Hybrid approaches:

- RL for high-level strategy
- Local search for fine-tuning
- Combine exploration with exploitation

The optimization problem is framed as:

$$\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid \pi\right], \quad r_t = \text{improvement in } f(\vec{x}_t) \tag{70}$$

6.11 Parameter Tuning Guidelines

Example: Hyperparameter Selection Strategies

Problem-dependent tuning:

Problem Type	ϵ	α	γ
Stochastic	0.2-0.3	0.05-0.1	0.9-0.95
Deterministic	0.05-0.1	0.1-0.2	0.95-0.99
Multi-modal	0.1-0.2	0.1	0.9
Simple	0.05	0.2	0.8

Adaptive strategies:

- **Decaying** ϵ: Start high, decrease over time
- **Adaptive** α: Larger for uncertain states
- **Curriculum learning**: Start simple, increase complexity

State-action space sizing:

- **Fine discretization**: Better approximation, slower learning
- **Coarse discretization**: Faster learning, may miss optima
- **Balanced approach**: Adjust based on problem complexity

Experience replay sizing:

- Small problems: 100-500 experiences
- Medium problems: 500-2000 experiences
- Large problems: 2000-10000 experiences
- Our choice: 1000 suitable for typical optimization

Validation approach:

- Cross-validation on similar problems
- Grid search over hyperparameters
- Performance monitoring during training

The parameter selection follows empirical guidelines:

$$\epsilon \in [0.05, 0.3], \quad \alpha \in [0.01, 0.3], \quad \gamma \in [0.8, 0.99] \tag{71}$$

7 State Hashing Algorithm

7.1 Concept of State Representation

Example: State Feature Hashing

Scenario: Representing a complex optimization state in discrete Q-table
State features: Parameter values, history, and context information
Challenge: Map high-dimensional or complex states to discrete state indices
Example state features:

```
state_features = {
    "temperature": 23.5,
    "pressure": 101.3,
    "iteration": 15,
    "previous_reward": -2.1,
    "convergence_rate": 0.05
}
```

Hashing process:

1. Serialize to JSON string: `{"temperature": 23.5, "pressure": 101.3, ...}`
2. Compute hash: `hash("...")` → large integer
3. Apply modulo: `hash % 100` → state index in [0, 99]

Result: Complex state mapped to discrete index for Q-table lookup.

The state hashing algorithm converts complex, potentially high-dimensional state representations into discrete indices suitable for Q-table lookup in reinforcement learning systems.

7.2 Mathematical Foundation

7.2.1 Hash Function Properties

Example: Hash Function Mathematics

Hash function definition: $h : \mathcal{X} \to \{0, 1, \ldots, m-1\}$
Desirable properties:

- **Deterministic**: Same input $\to$ same output
- **Uniform distribution**: Inputs spread evenly across outputs
- **Fast computation**: Efficient hashing for real-time use
- **Consistent**: Similar inputs may map to different outputs

Our composite hash function:

$$h_{\text{total}}(x) = (h_{\text{python}}(\text{JSON}(x))) \mod m$$
$$\text{where } m = n\_states$$

Probability analysis:

$$P(h(x) = i) \approx \frac{1}{m} \quad \text{(uniform distribution)}$$
$$\text{Collision probability} = 1 - \prod_{i=1}^{n-1} \left(1 - \frac{i}{m}\right)$$

For $m = 100$, $n = 50$ states: Collision probability ≈ 0.395

The hash function composition is:

$$h(\text{state}) = (\text{hash}(\text{JSON}(\text{state}))) \mod n\_states \tag{72}$$

7.3 Algorithm Step-by-Step

7.3.1 State Serialization

Example: JSON Serialization Process

Input state features:

```
{
    "param1": 0.75,
    "param2": -1.2,
    "step_count": 42,
    "performance": 0.88
}
```

Serialization with sorted keys:

```
json.dumps(state_features, sort_keys=True)
```

Resulting string:

```
'{"param1": 0.75, "param2": -1.2, "performance": 0.88, "step_count":
```

Key sorting importance:

- Ensures consistent ordering of dictionary items
- Same semantic state → same string representation
- Avoids hash inconsistencies due to key ordering

Supported data types:

- Numbers: integers, floats
- Strings: feature names, categorical values
- Booleans: flags, binary features
- Nested structures: within JSON compatibility

Limitations:

- Floating-point precision issues
- Non-serializable objects not supported
- Large states produce long strings

7.3.2 Hash Computation and Modulo Operation

Example: Hash and Modulo Calculation

String to hash conversion:

$$\begin{aligned}
\text{Input string}: &\quad s = \texttt{"}\{\ddot{\text{p}}\text{aram1}\ddot{:}\ 0.75, \ldots\}\texttt{"} \\
\text{Python hash}: &\quad h_{\text{raw}} = \text{hash}(s) \\
\text{Typical output}: &\quad h_{\text{raw}} = 1234567890123456789
\end{aligned}$$

Modulo operation:

$$\begin{aligned}
\text{State index} &= h_{\text{raw}} \mod n\_states \\
&= 1234567890123456789 \mod 100 \\
&= 89
\end{aligned}$$

Range guarantee:

$$\text{state\_index} \in [0, n\_states - 1] = [0, 99] \tag{73}$$

Collision example:

$$\begin{aligned}
\text{State A}: &\quad \{"x" : 1.0, "y" : 2.0\} \rightarrow \text{index } 42 \\
\text{State B}: &\quad \{"temp" : 300, "press" : 100\} \rightarrow \text{index } 42 \\
\text{Result}: &\quad \text{Both states share Q-table row}
\end{aligned}$$

Collision impact: States share learning, may interfere but also generalize.

Algorithm 6 State Hashing Algorithm

```
1: function GetStateHash(state_features)
2:     $state\_str \leftarrow json.dumps(state\_features, sort\_keys = True)$    ▷ Serialize to string
3:     $raw\_hash \leftarrow hash(state\_str)$    ▷ Compute integer hash
4:     $state\_index \leftarrow raw\_hash \mod self.n\_states$    ▷ Map to state space
5:     return $state\_index$
6: end function
```

7.4 Hash Function Properties Analysis

Example: Python Hash Function Characteristics

Python `hash()` function properties:

- **Deterministic**: Same string $\rightarrow$ same hash within process
- **Uniform**: Good distribution across integer space
- **Fast**: $O(n)$ time where n is string length
- **Platform-dependent**: May vary between Python versions

Hash value range:

$$\begin{aligned}\text{For 64-bit Python}: &\quad h \in [-2^{63}, 2^{63} - 1]\\ \text{For 32-bit Python}: &\quad h \in [-2^{31}, 2^{31} - 1]\end{aligned}$$

String hashing algorithm:

$$h(s) = \left(\sum_{i=0}^{n-1} c_i \cdot p^{n-1-i}\right) \mod 2^m \tag{74}$$

where c_i is character code, p is prime multiplier.
Modulo operation properties:

- **Surjective**: Covers entire range $[0, n\_states - 1]$
- **Uniform**: If hash is uniform, modulo preserves uniformity
- **Fast**: Single integer operation

Cryptographic vs non-cryptographic:

- Python `hash()` is non-cryptographic (fast)
- Not secure against collision attacks
- Suitable for internal state management

The hash function has expected properties:

$$\mathbb{E}[\text{collisions}] = \frac{n^2}{2m}, \quad \text{where } n = \text{unique states}, m = n\_states \tag{75}$$

7.5 State Space Coverage Analysis

Example: State Distribution Analysis

Theoretical state coverage:

$$\begin{aligned}
\text{Possible states}: &\quad \text{Infinite (continuous parameters)} \\
\text{Discrete states}: &\quad n\_states = 100 \\
\text{Expected collisions}: &\quad E[\text{collisions}] = \frac{k^2}{2n} \quad \text{for } k \text{ unique states}
\end{aligned}$$

For different state counts:

Unique States	Collision Probability	Expected Collisions
10	0.41	0.45
50	0.93	12.25
100	1.00	49.50
200	1.00	199.00

Load factor implications:

$$\begin{aligned}
\text{Load factor} = \frac{\text{unique states}}{n\_states} \\
\lambda = 0.5: &\text{ Good performance, few collisions} \\
\lambda = 1.0: &\text{ Moderate collisions, acceptable} \\
\lambda = 2.0: &\text{ Many collisions, potential interference}
\end{aligned}$$

Optimal usage: Keep $\lambda < 1.0$ for best performance.

The state space utilization follows:

$$\lambda = \frac{\#\text{ unique states}}{n\_states}, \quad P(\text{collision}) \approx 1 - e^{-\lambda} \tag{76}$$

7.6 Collision Handling and Implications

Example: Hash Collision Effects

Collision scenarios:

- **Benign collision**: Similar states share learning (generalization)
- **Harmful collision**: Different states interfere (confusion)
- **Neutral collision**: Unrelated states, minimal interaction

Learning interference:

$$Q(s_1, a) \leftarrow Q(s_1, a) + \alpha[r_1 + \gamma \max Q(s'_1, a') - Q(s_1, a)]$$
$$Q(s_2, a) \leftarrow Q(s_2, a) + \alpha[r_2 + \gamma \max Q(s'_2, a') - Q(s_2, a)]$$

If s_1 and s_2 collide, they share Q-values and updates.
Positive effects:

- **Generalization**: Similar states benefit from shared learning
- **Reduced memory**: Fewer distinct Q-table entries
- **Faster learning**: Knowledge transfer between states

Negative effects:

- **Interference**: Conflicting updates for different states
- **Suboptimal policies**: Compromised decision making
- **Oscillations**: Unstable learning behavior

Collision mitigation:

- Increase n_states for finer discrimination
- Use feature engineering to reduce state similarity
- Employ hashing tricks like Bloom filters

The collision impact depends on state similarity:

$$\text{Interference} \propto \frac{1}{\text{similarity}(s_1, s_2)} \cdot \text{collision frequency} \tag{77}$$

7.7 Alternative Hashing Strategies

Example: Comparison of Hashing Methods

JSON-based hashing (current approach):

- **Pros**: Handles complex nested structures, human-readable
- **Cons**: Overhead of serialization, floating-point issues

Feature vector hashing:

```
def hash_feature_vector(features):
    # Convert to tuple of values
    values = tuple(sorted(features.values()))
    return hash(values) % n_states
```

Custom hash functions:

```
def custom_state_hash(features):
    # Weight important features more
    hash_val = (hash(features['param1']) * 31 +
                hash(features['param2']) * 17 +
                hash(features['context']) * 7)
    return hash_val % n_states
```

Locality-sensitive hashing:

- Similar states → similar hashes (controlled collisions)
- Useful for generalization in RL
- More complex implementation

Performance comparison:

Method	**Speed**	**Collision Control**	**Flexibility**
JSON hashing	Medium	Low	High
Feature tuple	Fast	Medium	Medium
Custom weighted	Fast	High	Low
LSH	Slow	High	Medium

Alternative hashing approaches include:

$$h_{\text{custom}}(\vec{x}) = \left(\sum_{i=1}^{d} w_i \cdot h(x_i)\right) \mod m \tag{78}$$

7.8 Implementation Considerations

Example: Practical Implementation Details

Floating-point precision issues:

```
# Problem: Slightly different floats → different hashes
state1 = {"temp": 0.1 + 0.2}  # 0.30000000000000004
state2 = {"temp": 0.3}        # 0.3
hash1 != hash2  # Different hashes!
```

Solution: Normalize floating-point values:

```
def normalize_floats(obj):
    if isinstance(obj, float):
        return round(obj, 10)  # 10 decimal precision
    elif isinstance(obj, dict):
        return {k: normalize_floats(v) for k, v in obj.items()}
    elif isinstance(obj, list):
        return [normalize_floats(v) for v in obj]
    else:
        return obj
```

Memory and performance:

- JSON serialization: $O(n)$ time and space
- Hash computation: $O(n)$ time
- Suitable for states with ≤ 100 features
- Consider caching for frequently accessed states

Thread safety:

- `json.dumps()` is thread-safe
- `hash()` is thread-safe in CPython
- Method can be used in parallel environments

Error handling:

```
def get_state_hash(self, state_features):
    try:
        state_str = json.dumps(state_features, sort_keys=True)
        return hash(state_str) % self.n_states
    except (TypeError, ValueError) as e:
        # Fallback: use simple string representation
        state_str = str(sorted(state_features.items()))
        return hash(state_str) % self.n_states
```

7.9 Theoretical Bounds and Limitations

Example: Hash Function Limitations

Birthday paradox implications:

$$P(\text{no collision}) \approx \exp\left(-\frac{k(k-1)}{2m}\right) \tag{79}$$

For $m = 100$ **states**:

$$k = 12 : P(\text{collision}) \approx 0.5$$
$$k = 23 : P(\text{collision}) \approx 0.9$$
$$k = 38 : P(\text{collision}) \approx 0.99$$

Information loss:

- Continuous parameters $\rightarrow$ discrete bins
- Feature relationships may be lost
- State semantics not preserved in hash

Optimal state space sizing:

$$m_{\text{optimal}} = \alpha \cdot \sqrt{\text{expected unique states}} \tag{80}$$

where $\alpha = 2 - 5$ for good performance.
Alternative for large state spaces:

- Function approximation (neural networks)
- Tile coding for continuous spaces
- State aggregation techniques
- Deep Q-networks for high-dimensional states

Our parameter rationale: *n_states* = 100 suitable for moderate complexity problems.

The theoretical limitations include:

$$\text{Information loss} = 1 - \frac{\log_2(m)}{\log_2(\text{possible states})} \tag{81}$$

7.10 Application in Reinforcement Learning

Example: RL State Management

Integration with Q-learning:

```
class RLOptimizer:
    def get_action(self, state_features):
        state_idx = self.get_state_hash(state_features)

        # Epsilon-greedy action selection
        if random.random() < self.epsilon:
            return random.randint(0, self.n_actions - 1)
        else:
            return np.argmax(self.q_table[state_idx])

    def update_q_value(self, state_features, action, reward, next_st
        state_idx = self.get_state_hash(state_features)
        next_state_idx = self.get_state_hash(next_state_features)

        # Q-learning update
        td_error = (reward + self.gamma * np.max(self.q_table[next_s
                    - self.q_table[state_idx][action])
        self.q_table[state_idx][action] += self.alpha * td_error
```

State feature design guidelines:

- Include relevant optimization parameters
- Add context information (iteration, performance history)
- Normalize features to similar scales
- Avoid redundant or correlated features

Monitoring state usage:

```
def get_state_statistics(self):
    state_counts = Counter()
    for experience in self.experience_replay:
        state_idx = self.get_state_hash(experience['state'])
        state_counts[state_idx] += 1
```

```
    return {
        'unique_states': len(state_counts),
        'most_used_state': state_counts.most_common(1),
        'collision_rate': 1 - len(state_counts) / len(self.experienc
    }
```

Adaptive state space:

The hashing method enables:

$$\text{State management} = O(1) \text{ lookup}, \quad \text{Memory} = O(m \cdot |\mathcal{A}|) \tag{82}$$

where $m = n\_states$ is the state space size.

8 Epsilon-Greedy Action Selection Algorithm

8.1 Concept of Exploration vs Exploitation

Example: Exploration-Exploitation Trade-off

Scenario: RL agent in state 42 with learned Q-values
Q-values for state 42: $[1.2, 3.5, 0.8, -0.3, 2.1]$
Exploitation choice: Action 1 (index 1, value 3.5) - best known action
Exploration possibility: Any random action from 0 to 4
Decision process with $\epsilon = 0.1$:

- Generate random number: $r \sim U(0, 1)$
- If $r < 0.1$: Explore $\rightarrow$ random action (e.g., action 3)
- If $r \geq 0.1$: Exploit $\rightarrow$ action 1 (best Q-value)

Expected behavior:

- 90% of time: Choose best known action
- 10% of time: Try random action for discovery
- Balance between using knowledge and gaining new knowledge

The epsilon-greedy strategy maintains a balance between exploiting current knowledge and exploring new possibilities.

The epsilon-greedy action selection method implements the fundamental exploration-exploitation trade-off in reinforcement learning, ensuring the agent occasionally explores suboptimal actions to discover potentially better strategies.

8.2 Mathematical Foundation

8.2.1 Epsilon-Greedy Policy Definition

Example: Policy Mathematics

Epsilon-greedy policy definition:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = \arg\max_{a'} Q(s, a') \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{83}$$

Probability distribution:

$$P(\text{greedy action}) = 1 - \epsilon + \frac{\epsilon}{n\_actions}$$
$$P(\text{non-greedy action}) = \frac{\epsilon}{n\_actions}$$

For our parameters ($\epsilon = 0.1$, $n\_actions = 50$):

$$P(\text{best action}) = 0.9 + \frac{0.1}{50} = 0.902$$
$$P(\text{other actions}) = \frac{0.1}{50} = 0.002$$

Expected value:

$$\mathbb{E}[Q(s, a)] = (1 - \epsilon) \max_a Q(s, a) + \frac{\epsilon}{|\mathcal{A}|} \sum_a Q(s, a) \tag{84}$$

Optimality guarantee: With proper ϵ schedule, converges to optimal policy.

The epsilon-greedy policy is formally defined as:

$$\pi(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{if } a = a^* \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{85}$$

where $a^* = \arg\max_{a'} Q(s, a')$.

8.3 Algorithm Step-by-Step

8.3.1 Random Number Generation and Comparison

Example: Decision Process Execution

Input: state = 25, Q-table row: $[0.8, 2.3, -0.5, 1.7, 0.9]$

Step 1: Random number generation

$$\begin{aligned} r &\sim U(0,1) \\ \text{Example outcome} :& r = 0.07 \end{aligned}$$

Step 2: Epsilon comparison

$$\begin{aligned} r &= 0.07 \\ \epsilon &= 0.1 \\ \text{Condition} :& 0.07 < 0.1 \Rightarrow \text{True} \\ \text{Decision} :& \text{Explore (random action)} \end{aligned}$$

Step 3: Random action selection

$$\begin{aligned} \text{Random integer} :& \in [0,4] \\ \text{Example outcome} :& \text{action} = 2 \end{aligned}$$

Alternative scenario:

- If $r = 0.15$: $0.15 < 0.1 \rightarrow$ False $\rightarrow$ Exploit
- Greedy action: $\arg\max([0.8, 2.3, -0.5, 1.7, 0.9]) = 1$
- Return action 1

8.3.2 Greedy Action Selection

Example: Greedy Action Computation

Q-table state row: $Q[42] = [1.2, 3.5, 0.8, -0.3, 2.1]$
Argmax computation:

$$\max(Q[42]) = \max([1.2, 3.5, 0.8, -0.3, 2.1]) = 3.5$$
$$\arg\max(Q[42]) = \text{index of } 3.5 = 1$$

Tie-breaking behavior:

- NumPy `argmax` returns first maximum in case of ties
- $[2.0, 2.0, 1.0] \rightarrow$ returns index 0
- Deterministic behavior for same Q-values

Edge cases:

- All Q-values equal: Random selection among equals
- Negative Q-values: Still selects maximum (least bad)
- NaN values: Would cause errors (should be handled)

Efficiency: $O(n\_actions)$ time complexity, very fast for moderate action spaces.

Algorithm 7 Epsilon-Greedy Action Selection Algorithm

```
1: function SelectAction(state)
2:     $r \leftarrow$ random.random()          ▷ Generate uniform random number [0,1)
3:     if $r < self.epsilon$ then
4:         return random.randint($0, self.n\_actions - 1$)          ▷ Exploration
5:     else
6:         return np.argmax($self.q\_table[state]$)          ▷ Exploitation
7:     end if
8: end function
```

8.4 Probability Analysis

Example: Action Selection Probabilities

Probability distribution analysis:

Action Type	**Probability**	**Interpretation**
Greedy action	$1 - \epsilon + \frac{\epsilon}{n\_actions}$	Mostly exploitation
Other actions	$\frac{\epsilon}{n\_actions}$	Uniform exploration
Total exploration	ϵ	Random actions
Total exploitation	$1 - \epsilon$	Best actions

For our parameters ($\epsilon = 0.1$, $n\_actions = 50$):

$$P(\text{greedy}) = 0.9 + \frac{0.1}{50} = 0.902$$
$$P(\text{each non-greedy}) = \frac{0.1}{50} = 0.002$$
$$P(\text{any exploration}) = 0.1$$

Expected number of explorations:

$$\mathbb{E}[\text{explorations in } N \text{ steps}] = N \cdot \epsilon \tag{86}$$

For 1000 steps: Expected 100 exploratory actions.
Variance:

$$\begin{aligned} \text{Var}(\text{explorations}) &= N \cdot \epsilon \cdot (1 - \epsilon) \\ &= 1000 \cdot 0.1 \cdot 0.9 = 90 \\ \text{Std dev} &= \sqrt{90} \approx 9.5 \end{aligned}$$

The action selection probabilities are:

$$P(a|s) = \begin{cases} 1 - \epsilon + \frac{\epsilon}{|\mathcal{A}|} & \text{for } a = \arg\max Q(s, a) \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases} \tag{87}$$

8.5 Exploration-Exploitation Trade-off Analysis

Example: Epsilon Value Impact

Effect of different ϵ values:

ϵ	**Exploration**	**Exploitation**	**Use Case**
0.01	1%	99%	Final stages, refinement
0.1	10%	90%	Balanced learning (our choice)
0.3	30%	70%	Early exploration
0.5	50%	50%	Heavy exploration
0.9	90%	10%	Pure exploration phase

Learning progression with fixed $\epsilon = 0.1$:

- **Early learning**: Many exploratory actions discover good strategies
- **Mid learning**: Balance maintains option for improvement
- **Late learning**: Continued exploration finds refinements

Alternative: Decaying epsilon:

$$\epsilon_t = \epsilon_0 \cdot e^{-\lambda t} \tag{88}$$

Comparison:

- **Fixed** ϵ: Always maintains some exploration
- **Decaying** ϵ: Focuses on exploitation over time
- **Our choice**: Fixed for simplicity and continual adaptation

Theoretical optimality: With decaying ϵ, converges to optimal policy.

The exploration-exploitation balance follows:

$$\text{Exploration rate} = \epsilon, \quad \text{Exploitation rate} = 1 - \epsilon \tag{89}$$

8.6 Computational Complexity and Performance

Example: Runtime Analysis

Time complexity analysis:

$$\begin{aligned}\text{Random number generation}: &\quad O(1)\\ \text{Comparison}: &\quad O(1)\\ \text{Random action}: &\quad O(1)\\ \text{Argmax}: &\quad O(n\_actions)\\ \text{Total}: &\quad O(n\_actions)\end{aligned}$$

Space complexity: $O(1)$ additional space
For our parameters ($n\_actions = 50$):

- Operations: ≈ 50 comparisons for argmax
- Time: ≈ 0.1 microseconds (negligible)
- Suitable for real-time decision making

Bottleneck analysis:

- Argmax is the most expensive operation
- Random number generation is very fast
- Memory access to Q-table is efficient

Optimization considerations:

- For large action spaces: Use more efficient data structures
- Cache greedy actions for repeated states
- Vectorize for batch state processing

Comparison with alternative methods:

Method	**Time**	**Implementation**
Epsilon-greedy	$O(\|\mathcal{A}\|)$	Simple
Softmax	$O(\|\mathcal{A}\|)$	More complex
UCB	$O(\|\mathcal{A}\|)$	Requires counts

The computational complexity is:

$$C(n\_actions) = O(n\_actions) \tag{90}$$

8.7 Implementation Details and Edge Cases

Example: Robust Implementation Considerations

Random number generation:

```
random.random()  # Returns float in [0.0, 1.0)
# Examples: 0.0, 0.5, 0.999999...
# Never returns exactly 1.0
```

Boundary conditions:

- $\epsilon = 0.0$: Pure exploitation, always greedy
- $\epsilon = 1.0$: Pure exploration, always random
- $n\_actions = 1$: Always returns action 0
- Uninitialized Q-table: All zeros, random argmax

Tie-breaking in argmax:

```
Q = [1.0, 1.0, 0.5]  # Multiple maxima
np.argmax(Q)  # Returns 0 (first maximum)
```

Error handling:

```
def select_action(self, state: int) -> int:
    if state not in self.q_table:
        # Initialize state if not present
        self.q_table[state] = np.zeros(self.n_actions)

    if random.random() < self.epsilon:
        return random.randint(0, self.n_actions - 1)
    else:
        return np.argmax(self.q_table[state])
```

Thread safety:

- `random.random()` uses thread-local state
- Q-table access should be synchronized in parallel environments
- Method is generally thread-safe with proper synchronization

8.8 Alternative Action Selection Strategies

Example: Comparison with Other Methods

Softmax (Boltzmann exploration):

$$P(a|s) = \frac{e^{Q(s,a)/\tau}}{\sum_{a'} e^{Q(s,a')/\tau}} \tag{91}$$

- **Pros**: Smooth probability distribution
- **Cons**: Computationally expensive, temperature tuning

Upper Confidence Bound (UCB):

$$a_t = \arg\max_a \left[Q(s,a) + c\sqrt{\frac{\ln t}{N(s,a)}} \right] \tag{92}$$

- **Pros**: Theoretical optimality, adaptive exploration
- **Cons**: Requires action counts, more complex

Thompson Sampling:

- **Pros**: Bayesian optimal, natural uncertainty handling
- **Cons**: Requires posterior distributions, more complex

Our choice rationale:

- **Simplicity**: Easy to implement and understand
- **Effectiveness**: Works well in practice
- **Computational efficiency**: Fast execution
- **Tunability**: Single parameter (ϵ)

Hybrid approaches:

```
def select_action_adaptive(self, state):
    # Decaying epsilon
    current_epsilon = self.epsilon * (0.99 ** self.episode_count)
    if random.random() < current_epsilon:
        return random.randint(0, self.n_actions - 1)
    else:
        return np.argmax(self.q_table[state])
```

Alternative strategies include:

$$\pi_{\text{softmax}}(a|s) = \frac{\exp(Q(s,a)/\tau)}{\sum_{a'} \exp(Q(s,a')/\tau)}, \quad \pi_{\text{UCB}}(a|s) = Q(s,a) + c\sqrt{\frac{\ln t}{N(s,a)}} \tag{93}$$

8.9 Theoretical Convergence Properties

Example: Convergence Analysis

Greedy in the Limit with Infinite Exploration (GLIE):

- All state-action pairs visited infinitely often: $\lim_{t\to\infty} N_t(s,a) = \infty$
- Policy becomes greedy in the limit: $\lim_{t\to\infty} \pi_t(a|s) = \mathbf{1}_{a=\arg\max Q^*(s,a)}$

Our fixed ϵ policy:

- Does not satisfy GLIE (exploration doesn't decrease)
- Still converges to near-optimal in practice
- Maintains continual adaptation to environment changes

Convergence with decaying ϵ:

$$\epsilon_t = \frac{1}{t} \quad \text{satisfies GLIE conditions} \tag{94}$$

Q-learning convergence guarantee:

- With GLIE policy and Robbins-Monro learning rates
- Q-values converge to optimal Q^* with probability 1
- Our method: Practical compromise for real-world use

Regret analysis:

$$\text{Regret}(T) = \sum_{t=1}^{T} [V^*(s_t) - Q(s_t, a_t)] \tag{95}$$

Epsilon-greedy achieves $O(\log T)$ regret for multi-armed bandits.

The convergence properties include:

$$\lim_{t\to\infty} Q_t(s,a) = Q^*(s,a) \quad \text{with GLIE policy and proper learning rates} \tag{96}$$

8.10 Application in Optimization Context

Example: RL for Parameter Optimization

Optimization problem mapping:

$$\begin{aligned}
\text{State } s &: \text{Current parameter configuration} \\
\text{Action } a &: \text{Parameter adjustment} \\
\text{Reward } r &: \text{Objective function improvement} \\
\text{Q-value } Q(s,a) &: \text{Expected cumulative improvement}
\end{aligned}$$

Action selection in optimization:

```
def optimize_parameters(self, initial_params):
    current_params = initial_params
    for step in range(max_steps):
        # Convert params to state
        state_features = self.params_to_features(current_params)
        state_idx = self.get_state_hash(state_features)

        # Select parameter adjustment action
        action = self.select_action(state_idx)
        param_delta = self.actions[action]  # Lookup adjustment

        # Apply adjustment and evaluate
        new_params = current_params + param_delta
        reward = -objective_function(new_params)  # Negative for min

        # Update Q-values
        new_state_features = self.params_to_features(new_params)
        new_state_idx = self.get_state_hash(new_state_features)
        self.update_q_value(state_idx, action, reward, new_state_idx

        current_params = new_params
```

Exploration benefits in optimization:

- Discovers new regions of parameter space
- Escapes local optima through random jumps
- Maintains diversity in search strategy
- Adapts to changing objective landscapes

Parameter tuning for optimization:

- Higher ϵ for multi-modal functions
- Lower ϵ for unimodal functi

In optimization contexts, the method enables:

$$\text{Exploration} \rightarrow \text{Global search}, \quad \text{Exploitation} \rightarrow \text{Local refinement} \tag{97}$$

8.11 Performance Monitoring and Adaptation

Example: Monitoring Exploration Behavior

Tracking exploration statistics:

```
def get_exploration_stats(self):
    total_actions = self.step_count
    exploration_actions = self.exploration_count
    exploration_rate = exploration_actions / total_actions

    return {
        'total_actions': total_actions,
        'exploration_actions': exploration_actions,
        'exploration_rate': exploration_rate,
        'expected_exploration_rate': self.epsilon
    }
```

Adaptive epsilon adjustment:

```
def adaptive_epsilon_greedy(self, state):
    # Adjust epsilon based on learning progress
    if self.learning_stagnant():
        self.epsilon = min(0.3, self.epsilon * 1.1)  # More explorat
    else:
        self.epsilon = max(0.01, self.epsilon * 0.99)  # Less explor

    return self.select_action(state)
```

Quality of exploration monitoring:

- Track rewards from exploratory vs greedy actions
- Monitor discovery of new high-reward states
- Balance exploration cost vs potential benefit

Convergence detection:

- Stable Q-values across episodes
- Consistent policy behavior
- Diminishing returns from exploration

The exploration behavior can be monitored through:

$$\text{Actual exploration rate} = \frac{\#\text{ exploratory actions}}{\text{total actions}}, \quad \text{Target} = \epsilon \tag{98}$$

9 Q-Learning Update Algorithm

9.1 Concept of Temporal Difference Learning

Example: Q-Learning Update Process

Scenario: Agent transitions from state 25 to state 42 with reward
Current Q-values:

$$\begin{aligned} Q(25,3) &= 2.1 \quad \text{(current state-action pair)} \\ Q(42,:) &= [1.2, 3.5, 0.8, -0.3, 2.1] \quad \text{(next state values)} \end{aligned}$$

Update calculation:

$$\begin{aligned} \text{Target} &= \text{reward} + \gamma \cdot \max Q(42,:) \\ &= 1.5 + 0.95 \cdot 3.5 = 1.5 + 3.325 = 4.825 \\ \text{Temporal Difference} &= \text{Target} - Q(25,3) = 4.825 - 2.1 = 2.725 \\ \text{New Q-value} &= Q(25,3) + \alpha \cdot 2.725 = 2.1 + 0.1 \cdot 2.725 = 2.3725 \end{aligned}$$

Result: Q(25,3) updated from 2.1 to 2.3725 based on new experience.

The Q-learning update method implements the core temporal difference learning algorithm, adjusting action-value estimates based on immediate rewards and future value predictions.

9.2 Mathematical Foundation

9.2.1 Q-Learning Update Equation

Example: Q-Learning Mathematics

Standard Q-learning update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{99}$$

Components breakdown:

- $Q(s_t, a_t)$: Current Q-value estimate
- $\alpha = 0.1$: Learning rate (step size)
- r_{t+1}: Immediate reward received
- $\gamma = 0.95$: Discount factor
- $\max_a Q(s_{t+1}, a)$: Maximum future value estimate
- $r_{t+1} + \gamma \max_a Q(s_{t+1}, a)$: TD target
- $r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)$: TD error

Bellman optimality equation:

$$Q^*(s, a) = \mathbb{E} \left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a \right] \tag{100}$$

Q-learning as stochastic approximation:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha_t \left[r + \gamma \max_{a'} Q_t(s', a') - Q_t(s, a) \right] \tag{101}$$

The Q-learning update follows the temporal difference learning rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right] \tag{102}$$

9.3 Algorithm Step-by-Step

9.3.1 Current Q-Value Retrieval

Example: Q-Table Access

Input: state = 25, action = 3

Q-table access:

$$\text{self.q\_table}[25] = [0.8, 1.2, -0.5, 2.1, 0.3]$$
$$\text{current\_q} = \text{self.q\_table}[25][3] = 2.1$$

Default dictionary behavior:

- If state 25 not in Q-table: initialized to zeros
- Automatic state creation on first access
- Memory efficient for sparse state visits

Data structure:

```
q_table = {
    25: [0.8, 1.2, -0.5, 2.1, 0.3],
    42: [1.2, 3.5, 0.8, -0.3, 2.1],
    # ... other states
}
```

Efficiency: $O(1)$ average case access time.

9.3.2 Next State Value Computation

Example: Maximum Future Value Calculation

Input: next_state = 42
Q-values for next state:

$$Q(42,:) = [1.2, 3.5, 0.8, -0.3, 2.1]$$
$$\max Q(42,:) = \max([1.2, 3.5, 0.8, -0.3, 2.1]) = 3.5$$

Argmax behavior:

- Returns maximum value, not action index
- In case of ties: returns first maximum
- $[2.0, 2.0, 1.0] \rightarrow 2.0$ (first occurrence)

Bootstrapping interpretation:

- Uses current estimate of optimal future value
- "Learning a guess from a guess"
- Key to temporal difference methods

Off-policy nature: Uses $\max Q(s', a')$ regardless of actual policy used.

9.3.3 Temporal Difference Update

Example: Complete Update Calculation

Given parameters:

$$\begin{aligned}
\text{current\_q} &= 2.1 \\
\text{reward} &= 1.5 \\
\max\_next\_q &= 3.5 \\
\alpha &= 0.1 \\
\gamma &= 0.95
\end{aligned}$$

Step-by-step calculation:

$$\begin{aligned}
\text{TD target} &= \text{reward} + \gamma \cdot \max\_next\_q \\
&= 1.5 + 0.95 \cdot 3.5 = 1.5 + 3.325 = 4.825 \\
\text{TD error} &= \text{TD target} - \text{current\_q} = 4.825 - 2.1 = 2.725 \\
\text{Update} &= \alpha \cdot \text{TD error} = 0.1 \cdot 2.725 = 0.2725 \\
\text{new\_q} &= \text{current\_q} + \text{update} = 2.1 + 0.2725 = 2.3725
\end{aligned}$$

Interpretation:

- Positive TD error: Action was better than expected
- Negative TD error: Action was worse than expected
- Zero TD error: Perfect prediction

Algorithm 8 Q-Learning Update Algorithm

```
1: function UpdateQValue(state, action, reward, next_state)
2:     $current\_q \leftarrow self.q\_table[state][action]$    ▷ Current Q-value
3:     $max\_next\_q \leftarrow \max(self.q\_table[next\_state])$    ▷ Best next state value
4:     $td\_target \leftarrow reward + self.gamma \cdot max\_next\_q$    ▷ TD target
5:     $td\_error \leftarrow td\_target - current\_q$    ▷ Temporal difference error
6:     $new\_q \leftarrow current\_q + self.alpha \cdot td\_error$    ▷ Q-learning update
7:     $self.q\_table[state][action] \leftarrow new\_q$    ▷ Store updated value
8:     $self.experience\_replay.append((state, action, reward, next\_state))$    ▷ Store experience
9: end function
```

9.4 Experience Replay Storage

Example: Experience Buffer Management

Experience tuple structure: (s, a, r, s')
Storage in deque:

```
experience_replay = deque([
    (25, 3, 1.5, 42),
    (42, 1, -0.2, 15),
    (15, 4, 2.1, 33),
    # ... up to 1000 experiences
], maxlen=1000)
```

Buffer properties:

- Maximum capacity: 1000 experiences
- FIFO eviction when full
- Circular buffer implementation
- Efficient append and pop operations

Benefits of experience replay:

- **Sample efficiency**: Reuse experiences multiple times
- **Stability**: Break temporal correlations
- **Variance reduction**: Average over multiple experiences
- **Memory**: Learn from past interactions

Usage pattern:

1. Store each new experience
2. Sample random mini-batch for learning
3. Perform multiple Q-updates from single experience

9.5 Parameter Analysis

9.5.1 Learning Rate (α) Impact

Example: Learning Rate Effects

Role of learning rate:

$$\Delta Q = \alpha \cdot \text{TD error} \tag{103}$$

Effect of different α values:

- $\alpha = 0.0$: No learning, Q-values never change
- $\alpha = 0.1$: Slow, stable learning (our choice)
- $\alpha = 0.5$: Moderate learning speed
- $\alpha = 1.0$: Complete overwrite, potentially unstable

Convergence requirements:

$$\sum_{t=1}^{\infty} \alpha_t = \infty \quad \text{(infinite updates)}$$
$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty \quad \text{(finite variance)}$$

Our constant $\alpha = 0.1$:

- Violates Robbins-Monro ($\sum \alpha_t^2 = \infty$)
- Practical compromise for stability
- Works well in practice for many problems

Numerical stability: Small α prevents Q-value explosion.

The learning rate controls update magnitude:

$$|\Delta Q| \leq \alpha \cdot (|r|_{max} + \gamma \cdot |Q|_{max} + |Q|_{max}) = \alpha \cdot (R_{max} + (1+\gamma)Q_{max}) \tag{104}$$

9.5.2 Discount Factor (γ) Analysis

Example: Discount Factor Interpretation

Discount factor role:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots \tag{105}$$

Horizon analysis:

$$\begin{aligned}\text{Effective horizon} &\approx \frac{1}{1-\gamma} \\ \gamma = 0.95 &: \text{Horizon} \approx 20 \text{ steps} \\ \gamma = 0.99 &: \text{Horizon} \approx 100 \text{ steps}\end{aligned}$$

Value propagation:

- High γ: Long-term planning, slow value propagation
- Low γ: Short-term focus, fast value propagation
- Our choice $\gamma = 0.95$: Medium-term planning

In optimization context:

- High γ: Considers long-term consequences of actions
- Low γ: Focuses on immediate improvements
- Balance depends on problem structure

The discount factor determines value propagation:

$$\text{Importance of step } k = \gamma^k, \quad \text{Effective horizon} = \frac{1}{1-\gamma} \tag{106}$$

9.6 Convergence Properties

Example: Q-Learning Convergence Analysis

Theoretical convergence guarantee:

- Q-learning converges to Q^* with probability 1 if:
- All state-action pairs visited infinitely often
- Learning rates satisfy Robbins-Monro conditions
- The environment is a finite MDP

Robbins-Monro conditions:

$$\sum_{t=1}^{\infty} \alpha_t = \infty \quad \text{(infinite learning)}$$
$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty \quad \text{(finite variance)}$$

Our parameter compliance:

- Constant $\alpha = 0.1$: Violates $\sum \alpha_t^2 < \infty$
- Still converges in practice for many problems
- Alternative: Use decaying learning rate $\alpha_t = 1/t$

Convergence rate:

$$\|Q_t - Q^*\| \leq \gamma^t \|Q_0 - Q^*\| + O(\alpha) \tag{107}$$

Practical convergence:

- Typically requires thousands of updates
- Monitor Q-value changes for convergence detection
- Our setup suitable for moderate problems

The convergence guarantee is:

$$\lim_{t\to\infty} Q_t(s,a) = Q^*(s,a) \quad \text{for all } s, a \text{ with probability 1} \tag{108}$$

9.7 Computational Complexity

Example: Performance Analysis

Time complexity per update:

$$\begin{aligned}
\text{Q-table access} &: \quad O(1) \quad \text{(average case)} \\
\text{Max computation} &: \quad O(n\_actions) \\
\text{Arithmetic operations} &: \quad O(1) \\
\text{Experience storage} &: \quad O(1) \quad \text{(amortized)} \\
\text{Total} &: \quad O(n\_actions)
\end{aligned}$$

Space complexity:

- Q-table: $O(|\mathcal{S}| \cdot |\mathcal{A}|)$ floats
- Experience replay: $O(1000 \cdot 4)$ elements
- Temporary variables: $O(1)$

For our parameters:

- $n\_actions = 50$: ≈ 50 comparisons per update
- Time: ≈ 0.1 microseconds per update
- Memory: ≈ 200 KB for Q-table + 32 KB for replay

Bottleneck analysis:

- Max computation is most expensive operation
- Memory access patterns affect performance
- Suitable for real-time learning

Optimization opportunities:

- Cache max values for frequently visited states
- Use more efficient data structures
- Vectorize multiple updates

The computational complexity is:

$$C(n\_actions) = O(n\_actions) \quad \text{per update} \tag{109}$$

9.8 Implementation Considerations

Example: Practical Implementation Details

Initialization handling:

```
# Default dictionary automatically handles new states
self.q_table[new_state]  # Returns zeros if not present
```

Floating-point precision:

- Single-precision floats usually sufficient
- Avoid catastrophic cancellation in TD error
- Handle very large/small Q-values gracefully

Edge cases:

- Terminal states: $\max Q(s', a') = 0$ if s' is terminal
- Infinite/NaN rewards: Should be handled by environment
- Unvisited states: Automatically initialized to zeros

Error handling:

```
def update_q_value(self, state, action, reward, next_state):
    try:
        current_q = self.q_table[state][action]
        max_next_q = np.max(self.q_table[next_state])

        # Handle terminal states
        if self.is_terminal(next_state):
            max_next_q = 0.0

        td_target = reward + self.gamma * max_next_q
        new_q = current_q + self.alpha * (td_target - current_q)
        self.q_table[state][action] = new_q
        self.experience_replay.append((state, action, reward, next_s

    except Exception as e:
        print(f"Q-update error: {e}")
        # Fallback: initialize state if missing
        if state not in self.q_table:
            self.q_table[state] = np.zeros(self.n_actions)
```

Thread safety:

- Q-table updates should be synchronized in parallel environments

9.9 Alternative Update Strategies

Example: Comparison with Other Methods

Double Q-learning:

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha \left[r + \gamma Q_2(s', \arg\max_a Q_1(s',a)) - Q_1(s,a) \right] \tag{110}$$

- **Pros**: Reduces maximization bias
- **Cons**: More complex, two Q-functions

Expected SARSA:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \mathbb{E}[Q(s',a')] - Q(s,a) \right] \tag{111}$$

- **Pros**: Lower variance, on-policy
- **Cons**: Requires policy probabilities

Prioritized experience replay:

- **Pros**: Focus on important experiences
- **Cons**: More complex, requires priority maintenance

Our choice rationale:

- **Simplicity**: Easy to implement and understand
- **Effectiveness**: Proven performance
- **Efficiency**: Fast computation
- **Theoretical foundation**: Well-understood properties

Enhanced version with clipping:

```
def update_q_value_clipped(self, state, action, reward, next_state):
    current_q = self.q_table[state][action]
    max_next_q = np.max(self.q_table[next_state])

    td_target = reward + self.gamma * max_next_q
    td_error = td_target - current_q

    # Clip TD error for stability
    clipped_error = np.clip(td_error, -1.0, 1.0)

    new_q = current_q + self.alpha * clipped_error
    self.q_table[state][action] = new_q
```

Alternative update methods include:

$$\text{Double Q-learning: } Q_1(s,a) \leftarrow Q_1(s,a) + \alpha \left[r + \gamma Q_2(s', \arg\max_a Q_1(s',a)) - Q_1(s,a) \right] \tag{112}$$

9.10 Application in Optimization Context

Example: RL for Parameter Optimization

Optimization problem mapping:

$$\begin{aligned}\text{State } s &: \text{Current parameter configuration}\\ \text{Action } a &: \text{Parameter adjustment}\\ \text{Reward } r &: f(\vec{x}_{new}) - f(\vec{x}_{current}) \quad \text{(improvement)}\\ \text{Q-value } Q(s,a) &: \text{Expected cumulative improvement}\end{aligned}$$

Complete optimization loop:

```
def optimize(self, objective_func, initial_params, n_steps):
    current_params = initial_params
    for step in range(n_steps):
        # Convert to state representation
        state_features = self.param_to_features(current_params)
        state = self.get_state_hash(state_features)

        # Select parameter adjustment
        action = self.select_action(state)
        delta = self.action_to_delta(action)

        # Apply adjustment and evaluate
        new_params = current_params + delta
        reward = objective_func(new_params) - objective_func(current

        # Convert new params to state
        new_state_features = self.param_to_features(new_params)
        new_state = self.get_state_hash(new_state_features)

        # Update Q-values
        self.update_q_value(state, action, reward, new_state)

        current_params = new_params
```

Reward design considerations:

- **Relative improvement**: $f_{new} - f_{current}$
- **Normalized rewards**: Scale to reasonable range
- **Sparse rewards**: Only for significant improvements
- **Shaped rewards**: Include intermediate progress

Convergence monitoring:

In optimization contexts, Q-learning enables:

$$\text{Learning optimal parameter adjustment strategy} = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t (f(\vec{x}_{t+1}) - f(\vec{x}_t))\right] \tag{113}$$

9.11 Performance Monitoring and Debugging

Example: Learning Diagnostics

Tracking learning progress:

```
def get_learning_stats(self):
    td_errors = []
    q_changes = []

    for experience in self.experience_replay:
        state, action, reward, next_state = experience
        current_q = self.q_table[state][action]
        max_next_q = np.max(self.q_table[next_state])
        td_target = reward + self.gamma * max_next_q
        td_errors.append(td_target - current_q)

    return {
        'mean_td_error': np.mean(td_errors),
        'std_td_error': np.std(td_errors),
        'max_q_value': max(np.max(q_row) for q_row in self.q_table.v
        'min_q_value': min(np.min(q_row) for q_row in self.q_table.v
    }
```

Convergence detection:

- TD errors approaching zero
- Q-values stabilizing
- Policy becoming consistent
- Performance plateauing

Debugging common issues:

- **Q-values exploding**: Reduce learning rate
- **No learning**: Check reward signal, increase exploration
- **Oscillations**: Experience replay, target networks
- **Slow convergence**: Adjust learning parameters

Visualization:

- Plot Q-value distributions over time
- Track best action probabilities
- Monitor exploration vs exploitation balance

Learning progress can be monitored through:

$$\text{Learning quality} = \frac{1}{|\mathcal{S}||\mathcal{A}|}\sum_{s,a}|\text{TD error}(s,a)|, \quad \text{Convergence when } \to 0 \tag{114}$$

10 Experience Replay Algorithm

10.1 Concept of Batch Learning from Memory

Example: Experience Replay Process

Scenario: Agent has collected 500 experiences in replay buffer
Batch size: 32 experiences sampled randomly
Replay process:

1. Check buffer size: $500 \geq 32 \to$ proceed
2. Sample 32 random experiences without replacement
3. For each (s, a, r, s') in batch: perform Q-learning update
4. Total: 32 Q-value updates from diverse experiences

Example batch composition:

$$\begin{aligned}
\text{Experience 1}: &\quad (25, 3, 1.5, 42) \quad \text{(recent)}\\
\text{Experience 2}: &\quad (18, 1, -0.2, 25) \quad \text{(old)}\\
\text{Experience 3}: &\quad (42, 4, 2.1, 15) \quad \text{(medium)}\\
&\vdots\\
\text{Experience 32}: &\quad (33, 2, 0.8, 28)
\end{aligned}$$

Benefits: Breaks temporal correlations, reuses experiences, stabilizes learning.

The experience replay method enables batch learning from past experiences, breaking temporal correlations and improving sample efficiency by reusing historical data for multiple learning updates.

10.2 Mathematical Foundation

10.2.1 Stochastic Gradient Descent Perspective

Example: Batch Learning Mathematics

Objective function: Minimize expected TD error

$$J(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)^2 \right] \tag{115}$$

Stochastic gradient update:

$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta) \tag{116}$$

For tabular Q-learning:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right) \tag{117}$$

Batch update interpretation:

$$\mathbb{E}[\Delta Q] \approx \frac{1}{B} \sum_{i=1}^{B} \alpha \left(r_i + \gamma \max_{a'} Q(s'_i, a') - Q(s_i, a_i) \right) \tag{118}$$

Variance reduction:

$$\text{Var(single update)} = \sigma^2$$
$$\text{Var(batch mean)} = \frac{\sigma^2}{B}$$

For batch size 32: Variance reduced by factor of 32.

The experience replay minimizes the expected squared TD error:

$$J = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}} \left[\left(r + \gamma \max_{a'} Q(s', a') - Q(s, a) \right)^2 \right] \tag{119}$$

10.3 Algorithm Step-by-Step

10.3.1 Buffer Sufficiency Check

Example: Minimum Experience Requirement

Buffer size check:

$$\begin{aligned} \text{Current buffer size}: &\quad n = |\text{experience\_replay}| \\ \text{Required batch size}: &\quad B = 32 \\ \text{Condition}: &\quad n \geq B \end{aligned}$$

Example scenarios:

- $n = 25, B = 32$: $25 < 32 \rightarrow$ return early (no learning)
- $n = 32, B = 32$: $32 \geq 32 \rightarrow$ proceed with learning
- $n = 1000, B = 32$: $1000 \geq 32 \rightarrow$ proceed with learning

Rationale for minimum batch size:

- Statistical significance: Larger batches reduce variance
- Computational efficiency: Amortize overhead of sampling
- Learning stability: More representative gradient estimates

Early exit benefits:

- Prevents learning from too few examples
- Avoids biased updates from small samples
- Computational savings when insufficient data

10.3.2 Random Batch Sampling

Example: Uniform Random Sampling Process

Sampling without replacement:

$$\text{batch} \sim \text{Uniform}(\text{experience\_replay}, B) \tag{120}$$

Sampling properties:

- Each experience equally likely to be selected
- No duplicates in batch (without replacement)
- Preserves temporal independence
- Breaks sequential correlations

Example with buffer of 100 experiences:

$$P(\text{any experience selected}) = \frac{32}{100} = 0.32$$
$$\text{Expected unique updates}: \quad \text{32 distinct state-action pairs}$$

Implementation details:

```
random.sample(population, k)  # Returns k unique samples
# Time complexity: O(k) for deque
# Memory: Creates new list of size k
```

Alternative sampling strategies:

- **Sequential**: Maintains correlations (not used)
- **Prioritized**: Focus on important experiences
- **Stratified**: Ensure state coverage

10.3.3 Batch Processing Loop

Example: Parallel Experience Processing

Batch iteration:

$$\begin{aligned}\text{Batch size}: &\quad B = 32\\ \text{Iterations}: &\quad \text{32 Q-learning updates}\\ \text{Total TD errors}: &\quad \text{32 temporal difference calculations}\end{aligned}$$

Update independence:

- Each experience processed independently
- No ordering dependencies within batch
- Can be parallelized (though implemented sequentially)
- Q-table updates may have conflicts (handled naturally)

Example batch processing:

```
Batch: [(s1,a1,r1,s1'), (s2,a2,r2,s2'), ..., (s32,a32,r32,s32')]
For each (s,a,r,s') in batch:
    current_q = Q[s][a]
    max_next_q = max(Q[s'])
    td_target = r + gamma * max_next_q
    Q[s][a] += alpha * (td_target - current_q)
```

Computational pattern: 32 independent Q-learning updates.

Algorithm 9 Experience Replay Algorithm

```
1: function ReplayExperiences(batch_size)
2:     if len($self.experience\_replay$) < $batch\_size$ then
3:         return                ▷ Insufficient experiences for learning
4:     end if
5:     $batch$ ← random.sample($self.experience\_replay, batch\_size$)     ▷
   Sample random batch
6:     for ($state, action, reward, next\_state$) in $batch$ do
7:         $self.update\_q\_value(state, action, reward, next\_state)$     ▷
   Q-learning update
8:     end for
9: end function
```

10.4 Statistical Properties

Example: Batch Learning Statistics

Expected batch composition:

$$\mathbb{E}[\#\text{ experiences from time } t] = \frac{B}{N} \quad \text{for } t = 1, \ldots, N \tag{121}$$

For buffer size $N = 1000$, batch size $B = 32$:

$$\begin{aligned} \mathbb{E}[\text{experiences per batch}] &= 32 \\ \text{Std dev} &\approx \sqrt{32 \cdot \frac{1}{1000} \cdot \frac{999}{1000}} \approx 0.18 \\ \text{Distribution} :& \quad \text{Approximately Poisson} \end{aligned}$$

Temporal decorrelation:

- Sequential experiences: Highly correlated
- Random batch: Breaks temporal dependencies
- Improves learning stability
- Reduces variance in updates

Coverage properties:

$$\text{Expected unique states per batch} \approx B \cdot \left(1 - \left(1 - \frac{1}{|\mathcal{S}|}\right)^N\right)$$
$$\text{For } |\mathcal{S}| = 100, B = 32, N = 1000 : \approx 32 \cdot (1 - e^{-10}) \approx 31.9$$

Almost all states represented in large buffers.

The batch sampling provides:

$$\mathbb{E}[\text{TD error}] \approx \frac{1}{B} \sum_{i=1}^{B} \delta_i, \quad \text{Var} \approx \frac{\sigma_\delta^2}{B} \tag{122}$$

10.5 Buffer Management and Dynamics

Example: Replay Buffer Evolution

Buffer properties:

- Maximum capacity: 1000 experiences
- FIFO eviction policy
- Circular buffer implementation
- Constant memory footprint

Buffer growth phases:

1. **Initialization** (0-32 experiences): No replay possible
2. **Learning phase** (32-1000 experiences): Increasing diversity
3. **Steady state** (1000 experiences): Full coverage, old experiences forgotten

Experience age distribution:

$$\begin{aligned} \text{Newest experience}: &\quad \text{age} = 0 \\ \text{Oldest experience}: &\quad \text{age} = 999 \\ \text{Average age}: &\quad \approx 500 \text{ steps} \end{aligned}$$

Forgetting mechanism:

- FIFO eviction: oldest experiences discarded first
- Ensures buffer reflects recent environment dynamics
- Prevents memory from growing indefinitely
- Adapts to non-stationary environments

Buffer utilization metrics:

$$\text{Utilization} = \frac{\text{current size}}{1000}, \quad \text{Turnover rate} = \frac{\text{new experiences}}{\text{time}} \tag{123}$$

The buffer dynamics follow:

$$\text{Buffer state at time } t : \mathcal{D}_t = \{e_{t-999}, e_{t-998}, \ldots, e_t\} \tag{124}$$

10.6 Computational Complexity

Example: Performance Analysis

Time complexity:

$$\begin{aligned} \text{Size check}: &\quad O(1) \\ \text{Random sampling}: &\quad O(B) \\ \text{Batch processing}: &\quad O(B \cdot n\_actions) \\ \text{Total}: &\quad O(B \cdot n\_actions) \end{aligned}$$

Space complexity:

- Batch storage: $O(B)$ experiences
- Temporary variables: $O(1)$
- No persistent additional memory

For our parameters:

- $B = 32$, $n\_actions = 50$
- Operations: $32 \times 50 = 1600$ comparisons
- Time: ≈ 3.2 milliseconds per replay
- Suitable for intermittent batch learning

Comparison with online learning:

Method	**Time per step**	**Sample efficiency**
Online Q-learning	$O(\lvert\mathcal{A}\rvert)$	Low
Experience replay	$O(B\lvert\mathcal{A}\rvert)$ per replay	High

Amortized cost: With replay every K steps, average cost per step is $O\left(\frac{B|\mathcal{A}|}{K}\right)$.

The computational complexity is:

$$C(B, n\_actions) = O(B \cdot n\_actions) \tag{125}$$

10.7 Batch Size Selection Analysis

Example: Batch Size Trade-offs

Effect of different batch sizes:

Batch Size	Variance	Compute Time	Use Case
8	High	Fast	Early learning
32	Medium	Balanced	General purpose (our choice)
64	Low	Slow	Stable learning
128	Very low	Very slow	Final refinement
Online (1)	Highest	Fastest	Maximum data freshness

Variance reduction:

$$\text{Var(batch mean)} = \frac{\text{Var(single update)}}{B} \tag{126}$$

For $B = 32$: Variance reduced to $\frac{1}{32}$ of single update variance.
Learning stability:

- Small batches: High variance, unstable learning
- Large batches: Low variance, stable but slow
- Our choice $B = 32$: Good balance for most problems

Adaptive batch sizing:

```
def adaptive_batch_size(self):
    base_size = 32
    if self.learning_instable():  # High TD error variance
        return min(128, base_size * 2)  # Larger batch
    else:
        return base_size
```

Theoretical optimal batch size: Depends on problem complexity and noise level.

The batch size affects learning through:

$$\text{Stability} \propto \frac{1}{\sqrt{B}}, \quad \text{Freshness} \propto \frac{1}{B} \tag{127}$$

10.8 Implementation Considerations

Example: Practical Implementation Details

Random sampling efficiency:

```
# random.sample on deque: O(k) time complexity
batch = random.sample(self.experience_replay, batch_size)

# Alternative for very large buffers:
indices = np.random.choice(len(self.experience_replay),
                          batch_size, replace=False)
batch = [self.experience_replay[i] for i in indices]
```

Edge cases and error handling:

```
def replay_experiences(self, batch_size=32):
    if len(self.experience_replay) < batch_size:
        return  # Early exit

    try:
        batch = random.sample(self.experience_replay, batch_size)
        for experience in batch:
            state, action, reward, next_state = experience
            self.update_q_value(state, action, reward, next_state)
    except ValueError as e:
        print(f"Sampling error: {e}")
        # Fallback: use smaller batch
        actual_batch = min(batch_size, len(self.experience_replay))
        batch = random.sample(self.experience_replay, actual_batch)
        for experience in batch:
            state, action, reward, next_state = experience
            self.update_q_value(state, action, reward, next_state)
```

Thread safety considerations:

- `random.sample` is thread-safe
- Q-table updates may need synchronization
- Experience replay append/read should be synchronized
- Consider locking for concurrent access

Memory management:

- Batch is temporary and garbage collected
- No memory leaks from repeated replays
- Deque maintains constant memory footprint

10.9 Alternative Replay Strategies

Example: Advanced Replay Methods

Prioritized Experience Replay:

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \quad p_i = |\delta_i| + \epsilon \tag{128}$$

- **Pros**: Focuses on important experiences
- **Cons**: More complex, requires priority maintenance

Stratified sampling:

```
def stratified_sample(self, batch_size):
    # Group by state or reward
    positive_experiences = [e for e in self.experience_replay
                            if e[2] > 0]
    negative_experiences = [e for e in self.experience_replay
                            if e[2] <= 0]

    # Sample from each group
    n_pos = min(batch_size // 2, len(positive_experiences))
    n_neg = batch_size - n_pos

    batch = (random.sample(positive_experiences, n_pos) +
            random.sample(negative_experiences, n_neg))
    return batch
```

Recent-biased sampling:

- Higher probability for recent experiences
- Better for non-stationary environments
- Implement with weighted sampling

Our choice rationale:

- **Simplicity**: Easy to implement and understand
- **Effectiveness**: Works well in practice
- **Efficiency**: Fast sampling and processing
- **Theoretical basis**: Well-understood properties

Hybrid approach:

```
def hybrid_replay(self, batch_size=32):
    # Mix of uniform and recent experiences
```

Alternative replay strategies include:

$$\text{Prioritized: } P(i) \propto |\delta_i|^\alpha, \quad \text{Stratified: } P(i) \propto \text{group weight} \tag{129}$$

10.10 Integration with Learning Process

Example: Complete Learning Loop

Typical usage pattern:

```
def learn(self, environment, total_steps=10000):
    state = environment.reset()

    for step in range(total_steps):
        # Select and take action
        action = self.select_action(state)
        next_state, reward, done = environment.step(action)

        # Store experience
        self.experience_replay.append((state, action, reward, next_s

        # Learn from replay periodically
        if step % 4 == 0:  # Replay every 4 steps
            self.replay_experiences(batch_size=32)

        state = next_state
        if done:
            state = environment.reset()
```

Replay frequency strategies:

- **Every step**: Maximum learning, high computation
- **Every N steps**: Balanced approach (common)
- **When buffer full**: Efficient but delayed learning
- **Adaptive**: Based on learning progress

Our recommended pattern: Replay every 4-10 steps with batch size 32.
Convergence monitoring:

```
def learning_progress(self):
    td_errors = []
    for experience in random.sample(self.experience_replay,
                                     min(100, len(self.experience_repla
        state, action, reward, next_state = experience
        current_q = self.q_table[state][action]
        max_next_q = np.max(self.q_table[next_state])
        td_errors.append(abs(reward + self.gamma * max_next_q - curr

    return np.mean(td_errors)  # Lower means better convergence
```

The integration pattern is:

$$\text{Data collection} \rightarrow \text{Experience storage} \rightarrow \text{Periodic batch learning} \rightarrow \text{Policy improvement} \tag{130}$$

10.11 Theoretical Benefits and Limitations

Example: Experience Replay Analysis

Theoretical benefits:

- **Sample efficiency**: Reuse each experience multiple times
- **Reduced variance**: Averaging over batch reduces update noise
- **Decorrelated updates**: Breaks temporal dependencies
- **Stable learning**: Smoother convergence to optimal policy

Limitations and considerations:

- **Memory requirements**: Storage of past experiences
- **Computation overhead**: Additional sampling and processing
- **Non-stationarity**: Old experiences may become outdated
- **Delayed learning**: Experiences learned after some delay

Convergence guarantees:

- With experience replay, Q-learning still converges to Q^*
- Requires all state-action pairs visited infinitely often
- Batch updates provide better gradient estimates

For optimization problems:

- Particularly beneficial for expensive function evaluations
- Allows learning from limited data
- Stabilizes learning in noisy environments
- Enables transfer learning across similar problems

Empirical results: Typically 2-10x sample efficiency improvement over online learning.

The theoretical improvement is:

$$\text{Sample efficiency improvement} = O\left(\frac{\text{buffer size}}{\text{batch size}}\right) \tag{131}$$

10.12 Performance Optimization

Example: Efficiency Enhancements

Vectorized updates (for large batches):

```
def vectorized_replay(self, batch_size=32):
    if len(self.experience_replay) < batch_size:
        return

    batch = random.sample(self.experience_replay, batch_size)
    states, actions, rewards, next_states = zip(*batch)

    # Vectorized Q-value computation
    current_qs = [self.q_table[s][a] for s, a in zip(states, actions
    max_next_qs = [np.max(self.q_table[ns]) for ns in next_states]

    # Vectorized updates
    for i, (s, a, r, ns, cq, mnq) in enumerate(zip(
            states, actions, rewards, next_states, current_qs, max_n
        new_q = cq + self.alpha * (r + self.gamma * mnq - cq)
        self.q_table[s][a] = new_q
```

Parallel processing:

- Process batch experiences in parallel
- Use thread pool for independent Q-updates
- Requires careful synchronization of Q-table access

Caching strategies:

- Cache max Q-values for frequently visited next states
- Precompute TD targets for similar experiences
- Use memoization for repeated state-action pairs

Memory optimization:

- Use more compact experience representation
- Compress old experiences
- Implement circular buffer manually for control

Monitoring and adaptation:

```
def adaptive_replay(self):
    # Adjust batch size based on learning progress
```

Optimization strategies include:

$$\text{Speedup} = \frac{\text{sequential time}}{\text{parallel time}} \approx \frac{B \cdot t_{\text{update}}}{t_{\text{sync}} + \frac{B}{P} \cdot t_{\text{update}}} \tag{132}$$

11 Simulated Annealing Optimization Algorithm

11.1 Concept of Simulated Annealing

Example: Physical Annealing Metaphor

Physical analogy: Metal cooling and crystallization process
Optimization interpretation:

- **Temperature**: Controls acceptance of worse solutions
- **Cooling**: Gradual reduction in exploration
- **Annealing**: Progressive refinement toward optimum

Simple optimization example:

$$\begin{aligned} \text{Problem} &: \quad \min f(x) = x^2 \quad \text{for } x \in [-10, 10] \\ \text{Initial state} &: \quad x = 8.0, f(x) = 64.0 \\ \text{High temperature} &: \quad \text{Accept some worse moves to escape local minima} \\ \text{Low temperature} &: \quad \text{Mostly accept improving moves only} \end{aligned}$$

Key insight: By occasionally accepting worse solutions, the algorithm can escape local optima and find global optima.

Simulated Annealing is a probabilistic optimization technique inspired by the physical process of annealing in metallurgy, where a material is heated and slowly cooled to reduce defects and achieve a low-energy crystalline state.

11.2 Mathematical Foundation

11.2.1 Boltzmann Distribution and Acceptance Probability

Example: Statistical Mechanics Foundation

Boltzmann distribution: In thermal equilibrium, probability of state with energy E:

$$P(E) \propto \exp\left(-\frac{E}{k_B T}\right) \tag{133}$$

Optimization adaptation: For minimization problem, acceptance probability:

$$P_{\text{accept}} = \begin{cases} 1 & \text{if } \Delta f \leq 0 \\ \exp\left(-\frac{\Delta f}{T}\right) & \text{if } \Delta f > 0 \end{cases} \tag{134}$$

Where:

- $\Delta f = f_{\text{new}} - f_{\text{current}}$: Change in objective function
- T: Current temperature (control parameter)
- k_B: Boltzmann constant (absorbed into temperature scale)

For maximization (our case):

$$P_{\text{accept}} = \begin{cases} 1 & \text{if } \Delta f \geq 0 \\ \exp\left(\frac{\Delta f}{T}\right) & \text{if } \Delta f < 0 \end{cases} \tag{135}$$

Key property: As $T \to 0$, $P_{\text{accept}} \to 0$ for worse moves.

The acceptance probability follows the Metropolis criterion:

$$P(\text{accept}) = \min\left(1, \exp\left(\frac{-\Delta E}{T}\right)\right) \tag{136}$$

where ΔE is the energy change (objective function change).

11.3 Class Architecture and Initialization

Example: SA Optimizer State Initialization

For `initial_temp=100, cooling_rate=0.95`:

$$\begin{aligned}
\text{temperature} &= 100.0 \quad \text{(initial temperature)}\\
\text{cooling\_rate} &= 0.95 \quad \text{(geometric cooling)}\\
\text{current\_solution} &= \text{None} \quad \text{(awaiting initialization)}\\
\text{current\_score} &= -\infty \quad \text{(sentinel value)}\\
\text{best\_solution} &= \text{None} \quad \text{(best found so far)}\\
\text{best\_score} &= -\infty \quad \text{(best score so far)}
\end{aligned}$$

Initial state properties:

- No current solution: Must be initialized before optimization
- Infinite negative score: Indicates no evaluation yet
- Ready to start annealing process

Temperature interpretation:

- High temperature: High exploration, accept many worse moves
- Low temperature: High exploitation, mostly improving moves
- Cooling: Transition from exploration to exploitation

Memory usage: Stores only current and best solutions, very memory efficient.

11.3.1 Parameter Selection Rationale

Example: Temperature and Cooling Rate Analysis

Initial temperature selection:

- **Too high**: Excessive random exploration, slow convergence
- **Too low**: Insufficient exploration, stuck in local optima
- **Rule of thumb**: T_0 should allow $\approx 80\%$ acceptance of worse moves initially
- **Our choice**: $T_0 = 100$ works for many normalized problems

Cooling rate analysis:

$$T_{k+1} = \alpha \cdot T_k, \quad \alpha = 0.95 \tag{137}$$

Cooling schedule properties:

$$\begin{aligned}\text{After 100 iterations :} \quad & T_{100} = 100 \cdot 0.95^{100} \approx 0.59 \\ \text{After 200 iterations :} \quad & T_{200} = 100 \cdot 0.95^{200} \approx 0.0035\end{aligned}$$

Alternative cooling schedules:

- **Linear**: $T_k = T_0 - k \cdot \delta$
- **Logarithmic**: $T_k = \frac{T_0}{\ln(k+1)}$ (theoretical optimal)
- **Exponential**: $T_k = T_0 \cdot \alpha^k$ (our choice)

Our parameter rationale: $\alpha = 0.95$ provides smooth transition from exploration to exploitation.

The cooling schedule follows geometric progression:

$$T_k = T_0 \cdot \alpha^k, \quad \alpha \in (0, 1) \tag{138}$$

Algorithm 10 Simulated Annealing Optimizer Initialization

```
1: function SimulatedAnnealingOptimizer_Init(initial_temp, cooling_rate)
2:     $self.temperature \leftarrow initial\_temp$                ▷ Initial temperature
3:     $self.cooling\_rate \leftarrow cooling\_rate$              ▷ Geometric cooling factor
4:     $self.current\_solution \leftarrow$ None             ▷ Current candidate solution
5:     $self.current\_score \leftarrow -\infty$                 ▷ Current solution quality
6:     $self.best\_solution \leftarrow$ None                    ▷ Best solution found
7:     $self.best\_score \leftarrow -\infty$                     ▷ Best quality found
8: end function
```

11.4 Algorithm Initialization

11.5 State Management and Tracking

Example: Solution Tracking Strategy

Dual tracking system:

- **Current solution**: Currently exploring candidate
- **Best solution**: Historical best found
- **Separation**: Allows exploration while preserving best

Initialization requirements:

- Must call initialization method before optimization
- Sets initial current solution and evaluates it
- Updates both current and best solutions

Memory efficiency:

$$\begin{aligned} \text{Storage}: &\quad 2 \times \text{solution size} + 2 \times \text{float} \\ \text{For D-dimensional problem}: &\quad O(D) \text{ memory} \end{aligned}$$

Example state evolution:

$$\begin{aligned} \text{Iteration 0}: &\quad \text{current} = \vec{x}_0, \text{best} = \vec{x}_0 \\ \text{Iteration 1}: &\quad \text{current} = \vec{x}_1, \text{best} = \max(\vec{x}_0, \vec{x}_1) \\ \text{Iteration k}: &\quad \text{current} = \vec{x}_k, \text{best} = \max(\vec{x}_0, \dots, \vec{x}_k) \end{aligned}$$

Property: Best solution is monotonic improvement (for maximization).

The solution tracking maintains:

$$\text{best\_score}_k = \max_{i=0}^{k} f(\vec{x}_i), \quad \text{best\_solution}_k = \arg\max_{i=0}^{k} f(\vec{x}_i) \tag{139}$$

11.6 Temperature Dynamics Analysis

Example: Temperature Evolution

Geometric cooling progression:

$$T_k = T_0 \cdot \alpha^k \tag{140}$$

For our parameters ($T_0 = 100$, $\alpha = 0.95$):

Iteration	Temperature	Acceptance Behavior
0	100.00	High exploration (80% worse moves accepted)
10	59.87	Moderate exploration
50	7.69	Limited exploration
100	0.59	Mostly exploitation
150	0.05	Very limited exploration
200	0.0035	Essentially greedy search

Half-life analysis:

$$\begin{aligned} \text{Half-temperature iterations}: &\quad k_{1/2} = \frac{\ln(0.5)}{\ln(\alpha)} \\ \text{For } \alpha = 0.95: &\quad k_{1/2} \approx 13.5 \text{ iterations} \end{aligned}$$

Effective exploration period:

- Significant exploration: $T > 1.0$ (first ≈ 90 iterations)
- Transition phase: $1.0 > T > 0.1$ (iterations 90-150)
- Exploitation phase: $T < 0.1$ (after 150 iterations)

Total iterations for convergence: Typically 200-1000 depending on problem.

The temperature evolution follows:

$$T_k = T_0 \cdot e^{k \ln \alpha}, \quad \text{Half-life} = \frac{\ln 2}{|\ln \alpha|} \tag{141}$$

11.7 Acceptance Probability Analysis

Example: Worse Move Acceptance Behavior

Acceptance probability function:

$$P_{\text{accept}}(\Delta f, T) = \exp\left(\frac{\Delta f}{T}\right) \quad \text{for } \Delta f < 0 \tag{142}$$

For different temperature regimes:

Δf	$T = 100$	$T = 10$	$T = 1$
-0.1	0.999	0.990	0.905
-1.0	0.990	0.905	0.368
-5.0	0.951	0.607	0.007
-10.0	0.905	0.368	0.000045
-50.0	0.607	0.007	≈ 0

Interpretation:

- High T: Accept large deteriorations frequently
- Medium T: Accept small deteriorations, reject large ones
- Low T: Reject almost all deteriorations

Threshold analysis:

$$\begin{aligned} \text{Worse moves with } P_{\text{accept}} > 0.5 : &\quad \Delta f > -T \ln 2 \\ \text{For } T = 100 : &\quad \Delta f > -69.3 \\ \text{For } T = 10 : &\quad \Delta f > -6.93 \\ \text{For } T = 1 : &\quad \Delta f > -0.693 \end{aligned}$$

Practical implication: Temperature controls the "depth" of local minima that can be escaped.

The acceptance probability has the properties:

$$\lim_{T\to\infty} P_{\text{accept}} = 1, \quad \lim_{T\to 0} P_{\text{accept}} = 0, \quad \frac{\partial P}{\partial T} > 0 \tag{143}$$

11.8 Theoretical Convergence Properties

Example: Global Convergence Analysis

Homogeneous Markov chain model:

- Each temperature level: Finite Markov chain
- Stationary distribution: Boltzmann distribution
- Cooling schedule: Sequence of Markov chains

Boltzmann distribution at temperature T:

$$\pi_T(\vec{x}) = \frac{\exp(f(\vec{x})/T)}{\sum_{\vec{y}} \exp(f(\vec{y})/T)} \tag{144}$$

As $T \to 0$:

$$\lim_{T \to 0} \pi_T(\vec{x}) = \begin{cases} 1 & \text{if } \vec{x} = \arg\max f(\vec{x}) \\ 0 & \text{otherwise} \end{cases} \tag{145}$$

Convergence theorem: With logarithmic cooling schedule $T_k = \frac{c}{\ln(k+1)}$, simulated annealing converges to global optimum with probability 1.

Our geometric cooling:

- Does not satisfy theoretical convergence conditions
- Works well in practice for finite-time optimization
- More computationally efficient than logarithmic cooling

Practical convergence: Geometric cooling finds near-optimal solutions efficiently.

The theoretical convergence requires:

$$T_k \geq \frac{c}{\ln(k+1)}, \quad \lim_{k \to \infty} T_k = 0, \quad \sum_{k=1}^{\infty} \exp\left(-\frac{\Delta}{T_k}\right) = \infty \tag{146}$$

11.9 Implementation Considerations

Example: Practical Implementation Details

Initialization requirements:

```
# Must initialize before optimization
def initialize(self, initial_solution, initial_score):
    self.current_solution = initial_solution
    self.current_score = initial_score
    self.best_solution = initial_solution.copy()
    self.best_score = initial_score
```

Edge case handling:

- **Uninitialized state**: Check before optimization steps
- **Extreme scores**: Handle very large/small objective values
- **NaN/infinity**: Check for valid scores
- **Memory management**: Copy solutions to avoid reference issues

Numerical stability:

```
def safe_acceptance_probability(self, delta_score, temperature):
    if delta_score >= 0:
        return 1.0
    else:
        # Avoid underflow for very negative delta_score/temperature
        exponent = delta_score / temperature
        if exponent < -100:  # exp(-100) is essentially 0
            return 0.0
        else:
            return math.exp(exponent)
```

Alternative initialization strategies:

- **Random start**: Multiple random initial solutions
- **Heuristic start**: Good initial solution from domain knowledge
- **Grid start**: Systematic coverage of search space
- **Warm start**: Continue from previous optimization

11.10 Parameter Tuning Guidelines

Example: Adaptive Parameter Selection

Problem-dependent tuning:

Problem Type	Initial Temperature	Cooling Rate
Simple unimodal	10-50	0.85-0.90
Moderate multi-modal	50-100	0.90-0.95
Complex multi-modal	100-500	0.95-0.98
Very complex	500-1000	0.98-0.99

Initial temperature calibration:

```
def auto_calibrate_temperature(self, initial_solution, objective_fun
    scores = []
    current_score = objective_func(initial_solution)

    for _ in range(n_samples):
        neighbor = self.generate_neighbor(initial_solution)
        neighbor_score = objective_func(neighbor)
        scores.append(abs(neighbor_score - current_score))

    # Set temperature to achieve ~80% initial acceptance
    avg_delta = np.mean(scores)
    self.temperature = -avg_delta / math.log(0.8)
```

Adaptive cooling:

```
def adaptive_cooling(self, improvement_rate):
    base_rate = self.cooling_rate
    if improvement_rate > 0.1:  # Good progress
        return base_rate * 0.99  # Slow cooling
    else:  # Stagnating
        return base_rate * 1.01  # Faster cooling (with lower bound)
```

Stopping criteria:

- Temperature below threshold (e.g., $T < 10^{-6}$)
- Maximum iterations reached
- No improvement for many iterations
- Computational budget exhausted

Parameter selection follows empirical rules:

$$T_0 \propto \text{problem difficulty}, \quad \alpha \propto \frac{1}{\text{desired iterations}} \tag{147}$$

11.11 Comparison with Other Optimizers

Example: Optimization Method Comparison

Simulated Annealing advantages:

- **Global optimization**: Can escape local optima
- **Theoretical guarantees**: Converges to global optimum under conditions
- **Simplicity**: Easy to implement and understand
- **Flexibility**: Handles various problem types
- **Memory efficiency**: Minimal storage requirements

Limitations:

- **Parameter sensitivity**: Performance depends on cooling schedule
- **Slow convergence**: May require many iterations
- **No gradient use**: Doesn't exploit problem structure
- **Monte Carlo nature**: Stochastic, different runs may vary

Comparison with other methods:

Method	**Global**	**Speed**	**Memory**
Simulated Annealing	Yes	Medium	Low
Gradient Descent	No	Fast	Low
Particle Swarm	Yes	Medium	Medium
Genetic Algorithms	Yes	Slow	High
Bayesian Optimization	Yes	Slow	Medium

Best applications:

- Black-box optimization with expensive evaluations
- Multi-modal objective functions
- Problems with many local optima
- When global optimum is essential

The method selection criteria include:

$$\text{Choose SA when: } P_{\text{local optima}} \gg P_{\text{global optima}} \text{ and } C_{\text{evaluation}} \gg C_{\text{iteration}} \tag{148}$$

11.12 Application Domains

Example: Practical Application Scenarios

Combinatorial optimization:

- Traveling Salesman Problem
- Scheduling problems
- Graph partitioning
- Vehicle routing

Continuous optimization:

- Neural network training
- Protein folding
- Financial portfolio optimization
- Engineering design

Hyperparameter tuning:

- Machine learning model selection
- Feature selection
- Architecture search

Real-world success stories:

- **VLSI design**: Circuit layout optimization
- **Image processing**: Parameter tuning for filters
- **Bioinformatics**: Molecular structure prediction
- **Operations research**: Logistics and supply chain

Integration example:

```
def optimize_parameters(self, objective_func, param_bounds, max_iter
    # Initialize with random solution
    initial_solution = self.random_solution(param_bounds)
    initial_score = objective_func(initial_solution)
    self.initialize(initial_solution, initial_score)

    for iteration in range(max_iterations):
        # Generate neighbor solution
        neighbor = self.generate_neighbor(self.current_solution, par
```

Simulated Annealing excels in problems where:

$$f(\vec{x}) \text{ is expensive to evaluate}, \quad |\mathcal{X}| \text{ is large}, \quad \text{multiple local optima exist} \tag{149}$$

11.13 Performance Characteristics

Example: Computational Analysis

Time complexity per iteration:

$$\begin{aligned}
\text{Neighbor generation}: &\quad O(D) \quad \text{(problem dimension)} \\
\text{Objective evaluation}: &\quad O(C_f) \quad \text{(function cost)} \\
\text{Acceptance decision}: &\quad O(1) \\
\text{Cooling update}: &\quad O(1) \\
\text{Total per iteration}: &\quad O(D + C_f)
\end{aligned}$$

Space complexity:

$$\begin{aligned}
\text{Solution storage}: &\quad O(D) \\
\text{Parameters}: &\quad O(1) \\
\text{Total}: &\quad O(D)
\end{aligned}$$

For typical problems:

- $D = 10 - 1000$ dimensions
- C_f = milliseconds to hours (highly variable)
- Iterations = 1000-100000
- Memory usage = negligible for most problems

Convergence time:

$$T_{\text{convergence}} \propto \frac{1}{|\ln \alpha|} \cdot \frac{1}{\text{acceptance rate}} \tag{150}$$

Parallelization potential:

- Can evaluate multiple neighbors in parallel
- Main loop remains sequential due to acceptance decisions
- Suitable for expensive objective functions

Optimization tips:

- Use efficient neighbor generation
- Cache expensive computations
- Implement early rejection for clearly bad moves
- Use problem-specific knowledge when available

The computational characteristics are:

$$\text{Total time} = N_{\text{iterations}} \cdot (t_{\text{neighbor}} + t_{\text{evaluation}} + t_{\text{acceptance}}) \tag{151}$$