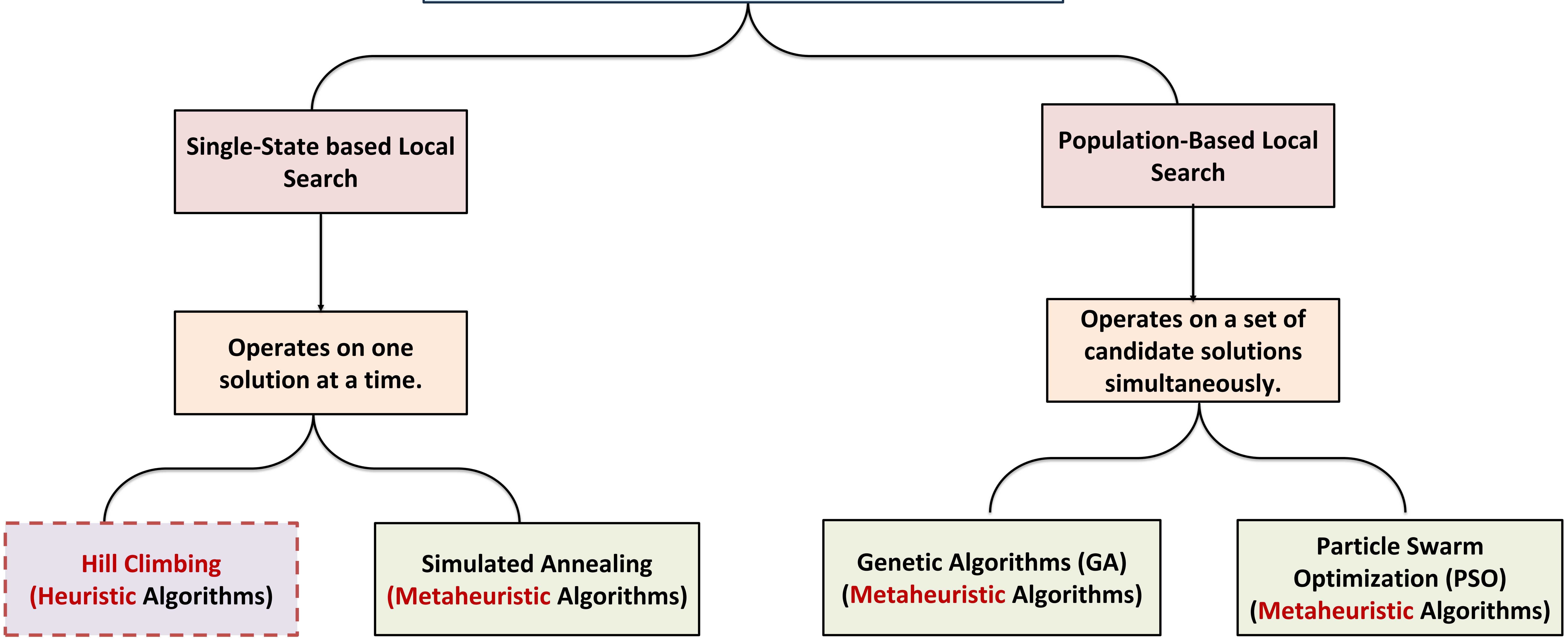
Nature Inspired Computation
DSAI 403

Assoc. Prof. **Mohamed Maher Ata**
Zewail city of science, technology, and innovation
momaher@zewailcity.edu.eg
Room: S028- Zone D

Types of Local Search Algorithms


Single-State based Local Search
Operates on one solution at a time.
Hill Climbing (Heuristic Algorithms)
Simulated Annealing (Metaheuristic Algorithms)
Population-Based Local Search
Operates on a set of candidate solutions simultaneously.
Genetic Algorithms (GA) (Metaheuristic Algorithms)
Particle Swarm Optimization (PSO) (Metaheuristic Algorithms)

Local Search & Hill Climbing

(Heuristic Algorithms)

1) Local Search as a Heuristic technique

1) Local Search as a Heuristic

Why Local Search is a heuristic:

- Works with one solution at a time
- Greedy: always moves to the **best neighbor**.
- Fast and simple, but **problem-specific**.
- Often gets stuck in local optima. This means that the algorithm may stop at a solution that looks best in its neighborhood, but is not the best overall solution.

Why Local Search is Not a Metaheuristic:

- No global strategy to explore the search space.
- No randomness/diversity as it depends heavily on the starting point.
- No escape mechanism from local optima.
- Metaheuristics = higher-level frameworks with exploration + exploitation balance

Local Search as a Heuristic

Neighborhood-based Local Search

1. Define a neighborhood (a set of nearby solutions).
2. Evaluate them all (or many of them).
3. Move to the best.
4. Good for small problems (where neighborhood size is manageable).

Perturbation-based Local Search

1. Instead of checking all nearby solutions, a perturbation rule just picks one random nearby solution to try.
2. Good for huge search spaces (hyperparameters, tuning CNNs).

Neighborhood-based Local Search

What is a Neighborhood?

- In optimization, a neighborhood of a solution s is the set of candidate solutions you can “reach” by applying a small change (move) to s.
- Type of neighborhood depends on the nature of variables (discrete vs continuous).

1. Discrete Neighborhoods

- Used when parameters are **categorical** or **integer-valued**.
- Examples: number of neurons, number of clusters, number of layers.
- **Move** = small integer change (e.g., ±1, ±5).

Example **(Neural Network hyperparameter — hidden neurons):**

- Current solution: $h = 32$neurons.
- Neighborhood:

$$N(h = 32) = \{24, 28, 36, 40\}$$

(if step size = 4).

- These represent **discrete jumps** in the architecture.

Advantages

- Easy to compute.
- Naturally fits problems with integer/categorical decisions.

Challenges

- May miss optimal values if step size too large.
- Slow search if step size too small.

Types of Neighborhoods

Why Neighborhood Matters

- Determines the search direction and step size.
- Affects whether the algorithm can escape local maxima or gets stuck.
- Balances between exploration (bigger neighborhoods) and efficiency (smaller neighborhoods).

Type of Neighborhoods:

1) Additive (Linear Step), change a variable by a fixed amount.

- **Example**: Dropout $d \in [0,1]$, move by ±0.05.

2) Multiplicative (Log-Scale Step), multiply/divide parameters by a factor.

- **Example**: Learning rate $\eta \in [10^{-5}, 10^{-1}]$, neighbors = $\eta/2$, $\eta \times 2$.

3) Combinatorial (Discrete Swaps/Edits), modify discrete elements .

Example:

Imagine we want to train a CNN on handwritten digit images (MNIST). We have 50 candidate handcrafted features (edges, textures, histogram bins, etc.), but we don't want to use all of them.

- Current Solution: Selected features = {f1, f2, f3, f7, f9, f12}
- Neighborhood Move (Combinatorial):
  - Swap: remove one feature and add another.
    - Example: replace f7 with f15 → {f1, f2, f3, f15, f9, f12}
  - Flip: toggle inclusion of a feature.
    - Example: add f20 → {f1, f2, f3, f7, f9, f12, f20}
  - Swap pair: exchange two features at once.
- Each modification = a neighbor solution.

4) Probabilistic Neighborhoods

- Instead of evaluating all possible neighbors, we randomly sample only a few candidates, which is especially useful in large neural networks where the search space is huge.
- **For example**, when tuning a CNN, if the current learning rate is 0.01 and dropout is 0.3, the possible neighbors could include many slight variations (e.g., η = 0.005, 0.02 and dropout = 0.25, 0.35). Instead of testing all of them, we randomly pick a small subset, such as (0.02, 0.35) and (0.005, 0.25), and evaluate only these. If one improves performance, we move there; otherwise, we continue sampling. This saves computation while still exploring the neighborhood effectively.

Finally:

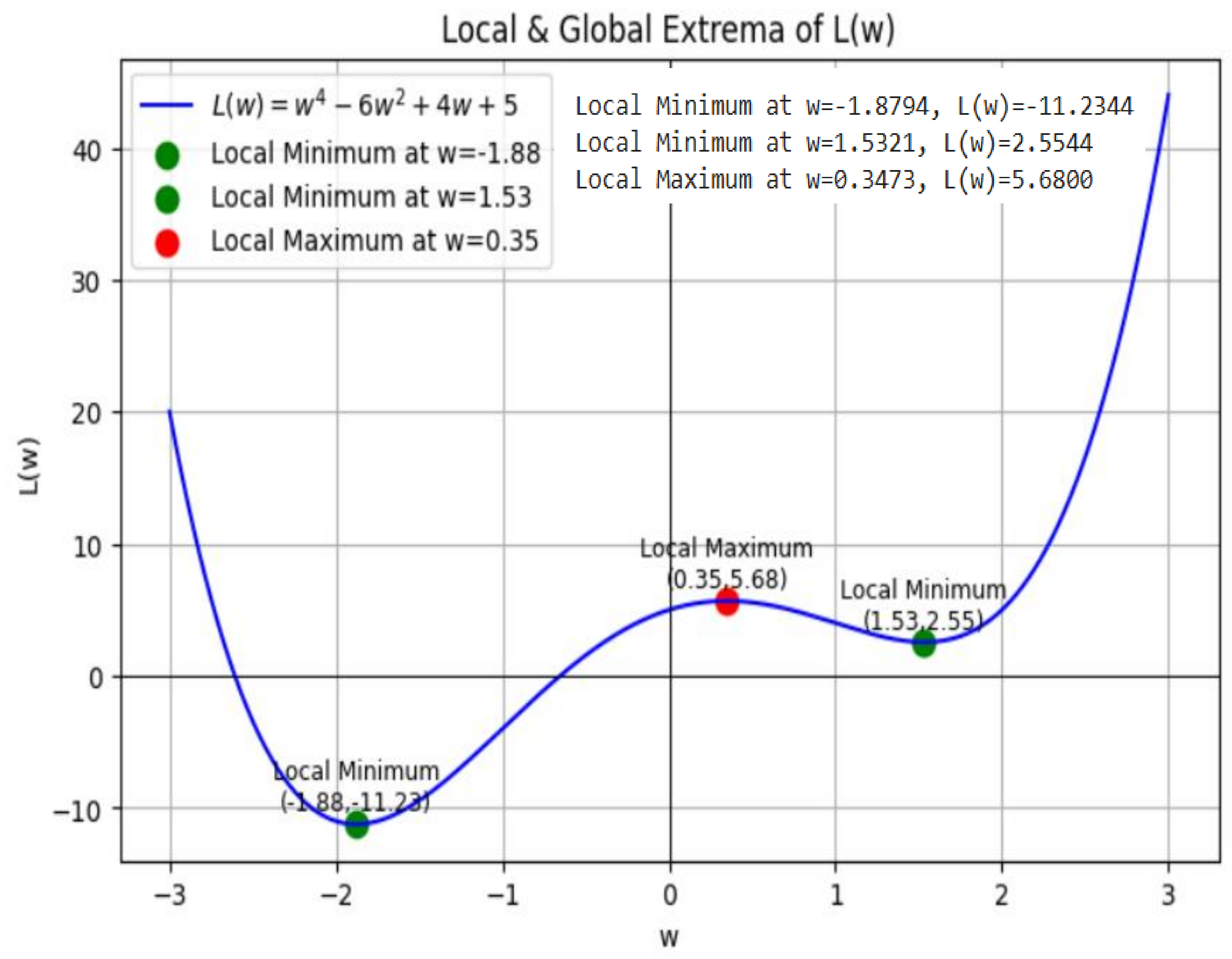
- **Small neighborhoods** = efficient but risk local maxima.
- **Large neighborhoods** = more global exploration but higher computational cost.
- Smartly designed neighborhoods improve both **solution quality** and **search efficiency**.

Example:

- Assume the loss function in simple Neural network model is: $L(w) = w^4 - 6w^2 + 4w + 5$
- This is a non convex function
- First Derivative: $\frac{dL}{dw} = 4w^3 - 12w + 4$
- Critical points: $4w^3 - 12w + 4 = 0$

  $w = -1.88, 0.35, 1.53$
- Second derivative: $L'' = 12w^2 - 12$

$w = -1.88$	$L'' > 0$ $(global\ minimum)$
$w = 0.35$	$L'' < 0$ $(local\ maximum)(hill)$
$w = 1.58$	$L'' > 0$ $(local\ minimum)$



- In mathematical practice, we compute critical points first.
- In AI/engineering practice, we don't know them,
  we just start somewhere and see where local search takes us.
- Assume mathematical practice; we will pick; $-2.5, 0, 2$ because:

  They are in different regions of the curve (left side, middle, right side).

  This shows how the same algorithm behaves differently depending on the start.

- Gradient descent update rule: $w_{t+1} = w_t - \eta \frac{dL}{dw}$ (assume η=0.1)
- Generate $neighborhood\ function\ N(w) = w_t + \Delta$ (**additive neighborhood**)

  Where: Δ is a small step around $w \rightarrow\ \Delta = -\eta \frac{dL}{dw}$
- **Important rule:** "Instead of checking all neighbors, we pick the direction of steepest descent (negative gradient)."
- **At $w = 0$**, possible neighbors: $-0.1, -0.2, +0.1, +0.2, \ldots..$
- If we check each, we'd calculate $L(-0.1), L(-0.2), L(+0.1), L(+0.2), \ldots..$however the derivative gives: $\frac{dL}{dw} = 4w^3 - 12w + 4$
- $\frac{dL}{dw} = 4\ (+ve)$ derivative which means that loss will increase if we go right (positive w). Accordingly go left (negative w). This is the **steepest descent direction**.

  Assume one iteration:

$$L(w) = w^4 - 6w^2 + 4w + 5$$

Start with: $w_0 = 0,\ then\ loss \quad L(0) = 5$

$$w_{t+1} = w_t - \eta \frac{dL}{dw} = 0 - 0.1 \times 4 = -0.4$$

Old loss $L(0) = 5$	New loss $L(-0.4)$ =2.4656	2.4656 < 5, so we accept -0.4 as the new one

At $w = -2.5$

$\mathrm{L}(-2.5) = -3.44$

$\frac{dL}{dw} = 4w^3 - 12w + 4 = -28.8$ *(-ve)* we should move right

$$w_{t+1} = w_t - \eta \frac{dL}{dw} = -2.5 - 0.1 \times -28.5 = 0.35$$

Old loss $L(-2.5) = -3.44$	New loss $L(0.35\ ) = 4.3$	$4.3 > -3.44$, Loss increased, not improved. We can solve this by choosing smaller η, let us say η=0.01

At $w = 2$

$\mathrm{L}(2) = 5$

$\frac{dL}{dw} = 4w^3 - 12w + 4 = 12$

$$w_{t+1} = w_t - \eta \frac{dL}{dw} = 2 - 0.1 \times 12 = 0.8$$

Old loss $L(2) = 5$	New loss $L(0.8\ ) = 4.8$	$4.8 < 5$, Loss decreased, we accept 0.8 as the new one

Summary: from the previous results we can conclude that:

Improvement depends on starting point	▪ At w=0, the search quickly found a lower loss (5→2.46565) ▪ At w=−2.5, the update made the loss worse (−3.44→4.3), meaning the step overshot. ▪ At w=2, the loss improved (5→4.85) but only slightly. ▪ This shows that local search is sensitive to the initial guess
Step size (η) matters a lot	▪ Large η (0.1) caused overshooting at w=−2.5. ▪ Reducing η to 0.01 prevents overshooting and allows smaller, safer steps. ▪ Choosing η is trial-and-error → if too big → instability; if too small → very slow progress
No guarantee of global optimum	▪ From some starting points (e.g., w=2), the algorithm may settle at a nearby local minimum instead of the global one. ▪ The method only "looks around locally" and does not explore the entire space
Heuristic behavior	▪ Local search doesn't solve the problem optimally in all cases, but it gives a "good enough" solution. ▪ It works like a greedy strategy: accept the move if it improves the loss

Pros of local search	**Cons of local search**
▪ Simple and easy to implement. ▪ Works well if the function is smooth and unimodal. ▪ Low memory requirement. ▪ Useful for large-scale problems where exact search is impossible	▪ Highly dependent on the starting point → different starts gave different outcomes. ▪ Can get stuck in local minima and miss the global minimum. ▪ Requires careful choice of step size η to avoid overshooting or slow progress. ▪ Purely greedy → no memory of past moves, cannot escape bad traps

Final pseudo code and mathematical representation of the Neighborhood-based Local Search algorithm

```
1:  x = GenerateInitialSolution()                    // Initialize
2:  **repeat**
3:      Neighborhood N = Generate Neighbors(x)        // Define candidate moves
4:      $x' =$ Find Improving Neighbor(N)             //Find a better solution in the neighborhood
5:      **if** $f(x') < f(x)$ **then**                //If an improving  neighbor exists ...
6:      $x = x'$        // ⋯ move to it(Descent)
7:      **else**
8:          **break**       // ⋯ else, terminate(LocalOptimum)
9:      **end if**
10: **until termination** condition met
```

$$x^{(t+1)} = \arg\min_{z \in N(x^{(t)})} f(z)$$

$$x^{(t+1)} = \begin{cases} x' \in N(x^{(t)}) & \text{if } \exists x' \in N(x^{(t)}) : f(x') < f(x^{(t)}) \\ x^{(t)} & \text{otherwise (terminate)} \end{cases}$$

Where:

- **Solution** (x): A candidate answer to the optimization problem.
- **Fitness Function** $(f(x))$: A function that measures the quality of a solution (lower is better for minimization).
- **Neighborhood** $(N(x))$: The set of all solutions that can be reached from x by a predefined "move" (e.g., swapping two elements, changing a variable slightly).
- **Local Optimum**: A solution x where no neighbor has a better fitness value, i.e.,$f(x') \geq f(x)\ for\ all\ x'\ in\ N(x)$.

b) Perturbation-based Local Search

Problem (goal)

Tune hyperparameters of a small CNN for image classification (**for example:** CIFAR-10).

Search space:

- $x1$= Learning rate, $[0.001, 0.1]$
- $x2$= Batch size, $[16, 128]$

Objective:

$$\max_{x \in X} f(x), \quad f(x) = \text{Validation Accuracy}(x_1, x_2)$$

Updated Rule (Adaptive Perturbation)

we try **local adaptive jumps**:

$$x_{\text{new}} = x_{\text{current}} + \underbrace{\beta \cdot \delta \cdot \text{sign}\left(f(x_{\text{current}}) - f\left(x_{\text{previous}}\right)\right)}_{\text{Step size}}$$

Where:

- β = step size scaling (small, e.g. 0.05).
- δ = random perturbation from $[-1, 1]$. In python (**delta = random.uniform(-1, 1)**)
- $sign(f(xcurr) - f(xprev))$ = adaptively moves in direction of improvement:

  i) If the new accuracy is better → sign = +1 → keep moving same way.

  ii) If worse → sign = −1 → flip the direction.

Step 1: Initialization:

Candidate	Learning rate (x_1)	Batch size (x_2)	Accuracy
V1	0.01	32	79%
V2	0.05	64	82% ← best
V3	0.08	100	76%

Step 2: Apply Adaptive Update

$$x_{\text{new}} = x_{\text{current}} + \beta \cdot \delta \cdot \text{sign}\left(f(x_{\text{current}}) - f\left(x_{\text{previous}}\right)\right)$$

For V1 compared to an initial:

$f(x_{\text{current}}) = 79\%$

$f\left(x_{\text{previous}}\right) = 82\%\ (assumption\ as\ there\ was\ no\ previous)$

$f(x_{\text{current}}) - f\left(x_{\text{previous}}\right) = -3$, sign = −1 (worse → flip).

a) Learning rate update

Assume: $\delta_{x1} = -0.6$

$Incement = \beta \cdot \delta_{x1} sign = 0.05 \times -0.6 \times -1 = +0.03$

$x_{\text{new}} = 0.01 + 0.03 = 0.04$

b) Batch size update

Assume: $\delta_{x2} = -0.5$

$Incement = \beta \cdot \delta_{x2} sign = 0.05 \times -0.5 \times -1 = +0.025$

$x_{\text{new}} = 32 + 0.025 = 32.025 \approx 32$

For V2 compared with V1:

$f(x_{\text{current}}) = 82\%$

$f\left(x_{\text{previous}}\right) = 79\%$

$f(x_{\text{current}}) - f\left(x_{\text{previous}}\right) = +3$, sign = +1 (improved → keep direction).

a) Learning rate update

Assume: $\delta_{x1} = 0.3$

$Incement = \beta \cdot \delta_{x1} sign = 0.05 \times 0.3 \times +1 = +0.015$

$x_{\text{new}} = 0.05 + 0.015 = 0.065$

b) Batch size update

Assume: $\delta_{x2} = 0.15$

$Incement = \beta \cdot \delta_{x2} sign = 0.05 \times 0.15 \times +1 = +0.0075$

$x_{\text{new}} = 64 + 0.0075 = 64.0075 \approx 64$

For V3 compared with V2:

$f(x_{\text{current}}) = 76\%$

$f\left(x_{\text{previous}}\right) = 82\%$

$f(x_{\text{current}}) - f\left(x_{\text{previous}}\right) = -6$, sign = -1 (worse → flip).

a) Learning rate update

Assume: $\delta_{x1} = 0.4$

$Incement = \beta \cdot \delta_{x1} sign = 0.05 \times 0.4 \times -1 = -0.02$

$x_{\text{new}} = 0.08 - 0.02 = 0.06$

b) Batch size update

Assume: $\delta_{x2} = 0.2$

$Incement = \beta \cdot \delta_{x2} sign = 0.05 \times 0.2 \times -1 = -0.01$

$x_{\text{new}} = 100 - 0.01 = 99.99 \approx 100$

Step 3: Re-evaluate

Candidate	Learning rate (x_1)	Batch size (x_2)	New Accuracy
V1	0.04	32	80%
V2	0.065	64	84%
V3	0.06	100	78%

Final recommendations:

Best solution after update:

- $x1$≈0.065
- $x2$≈64
- Accuracy ≈ 84%.

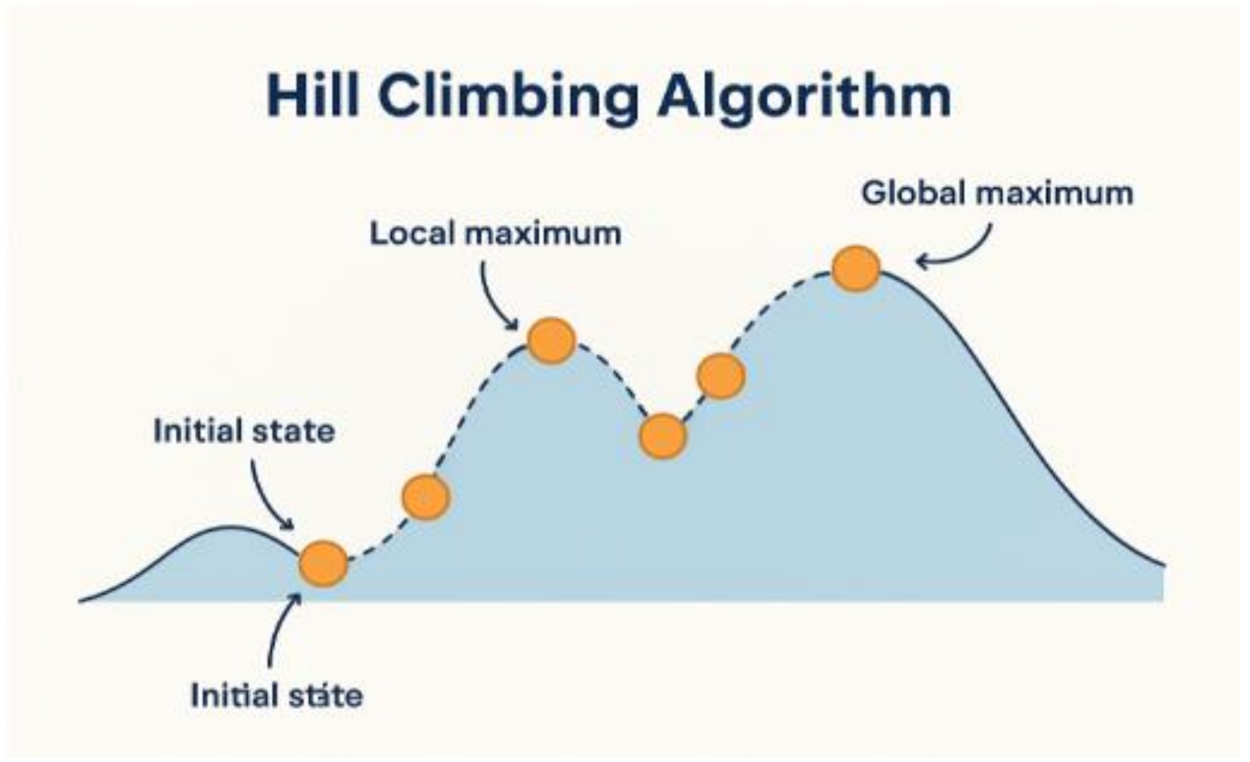
Here we use a momentum-flip rule:

- If accuracy improves → keep same direction.
- If accuracy worsens → flip direction.

Important question: Why do we need the sign? Isn't δ already random positive/negative?

- δ alone can move in both directions. But the **sign** is not about giving positive or negative direction randomly, it's about **remembering** whether the last move was good or bad.
- Without **sign** → you just step randomly forward/backward.
- With **sign** → you actually learn from the last step.

2) Hill climbing

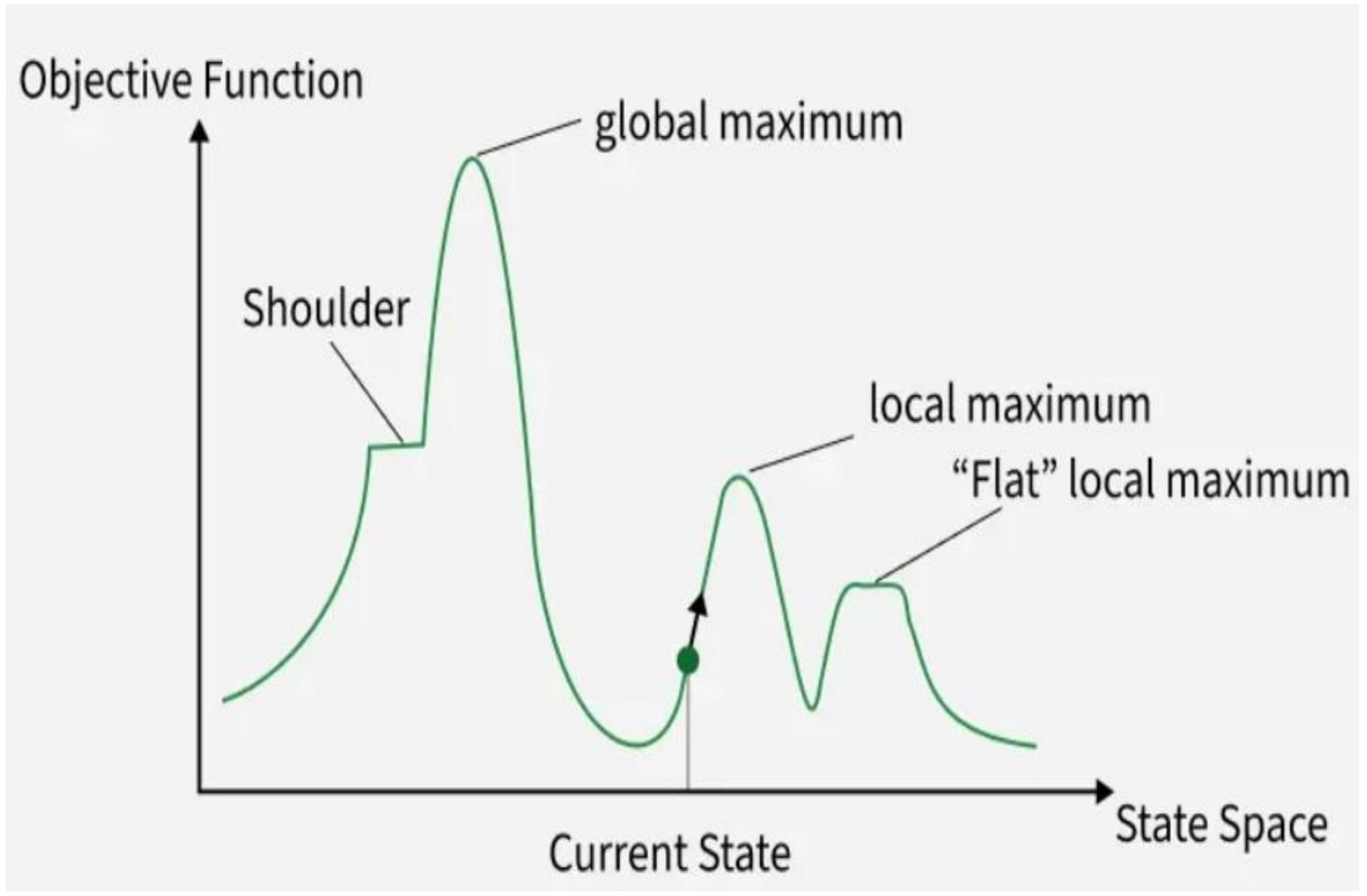


- A special case of local search and It's called hill climbing because the process resembles climbing a hill step by step until you reach the top. It stops when no better neighbor is found (local optimum).
- Classic hill climbing always moves toward higher values of the objective function. That's why it's naturally for maximization.
- Hill climbing **can be adapted** for minimization, but it's naturally a maximization algorithm
- If we need to use hill climbing for minimizing error (or loss):

  Invert the function: define a new function $f'(x) = -f(x)$ where $f(x)$ is the error. Then, maximizing $f'(x)$ using hill climbing is equivalent to minimizing the original error.
- Or, use a variant designed for minimization, like steepest descent (gradient descent) or other local search methods that move toward lower values.

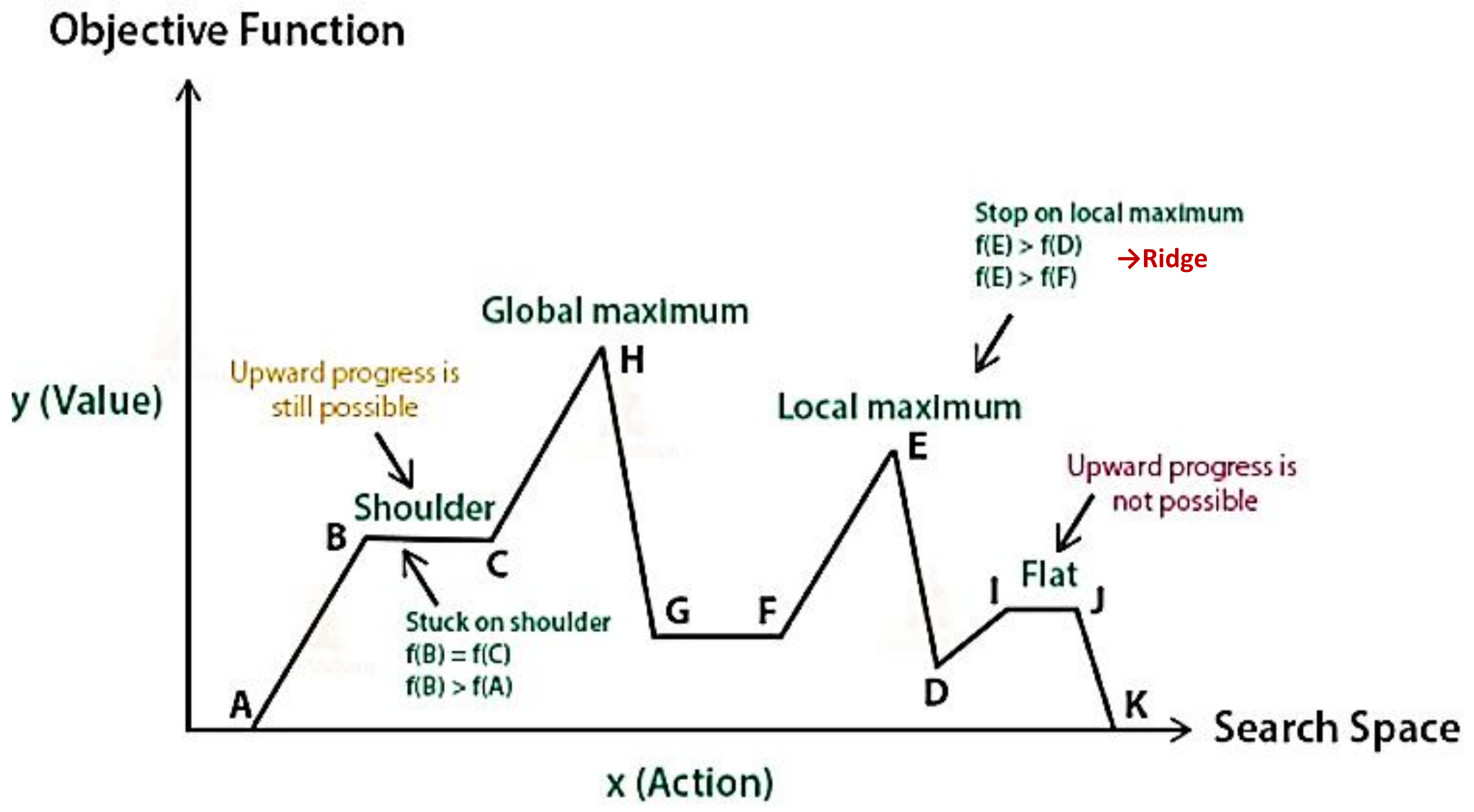
State-Space Diagram in Hill Climbing

- State-space diagram is a visual representation of all possible states the search algorithm can reach, plotted against the values of the objective function (the function we aim to maximize).
- The optimal solution in the state-space diagram is represented by the state where the **objective function** reaches its maximum value, also known as the **global maximum**.
- In the state-space diagram:
- **X-axis**: Represents the state space which includes all the possible states or configurations that the algorithm can reach.
- **Y-axis**: Represents the values of the objective function corresponding to each state.



Analogy of State-Space Diagram

1. **Local Maximum**: A local maximum is a state better than its neighbors but not the best overall. While its objective function value is higher than nearby states, a global maximum may still exist.
2. **Global Maximum**: The global maximum is the best state in the state-space diagram where the objective function achieves its highest value. This is the optimal solution the algorithm seeks.
3. **Plateau/Flat Local Maximum**: A plateau is a flat region where neighboring states have the same objective function value, making it difficult for the algorithm to decide on the best direction to move.
4. **Ridge**: A ridge is a higher region with a slope which can look like a peak. This may cause the algorithm to stop prematurely, missing better solutions nearby.
5. **Current State**: The current state refers to the algorithm's position in the state-space diagram during its search for the optimal solution.
6. **Shoulder**: A shoulder is a plateau with an uphill edge allowing the algorithm to move toward better solutions if it continues searching beyond the plateau.



Hill Climbing General Formulation

What is Hill Climbing?

- A local search **heuristic** optimization algorithm.
- Starts with an initial solution S_0 and repeatedly moves to a better solution in its neighborhood $N(s)$.
- Works best for problems where we don't know the global structure of the search space.

Core Idea

- Objective function (a "fitness" or "evaluation" function):

$$f : S \to R$$

  - f is the objective function (fitness function).
  - It assigns a numeric value to each state s in the search space S.

- Iterative update rule:

$$s_{t+1} = \arg \max_{s' \in N(s_t)} f\,(s')$$

▪ $N(s_t)$: This is the neighborhood function: it gives all possible solutions you can move to from the current state s_t

▪ $\arg \max_{s' \in N(s_t)} f\,(s')$: Among all neighbors s', find the one with the maximum value of $f(s')$

▪ s_{t+1}:This is your new state, the "best move" at time step t+1.

Example:

In deep learning, the validation accuracy (ValAcc) depends on hyperparameters in a nonlinear way.
Let hyperparameters be:
η= learning rate
d= dropout rate
Assume we have a **mock** validation accuracy function:

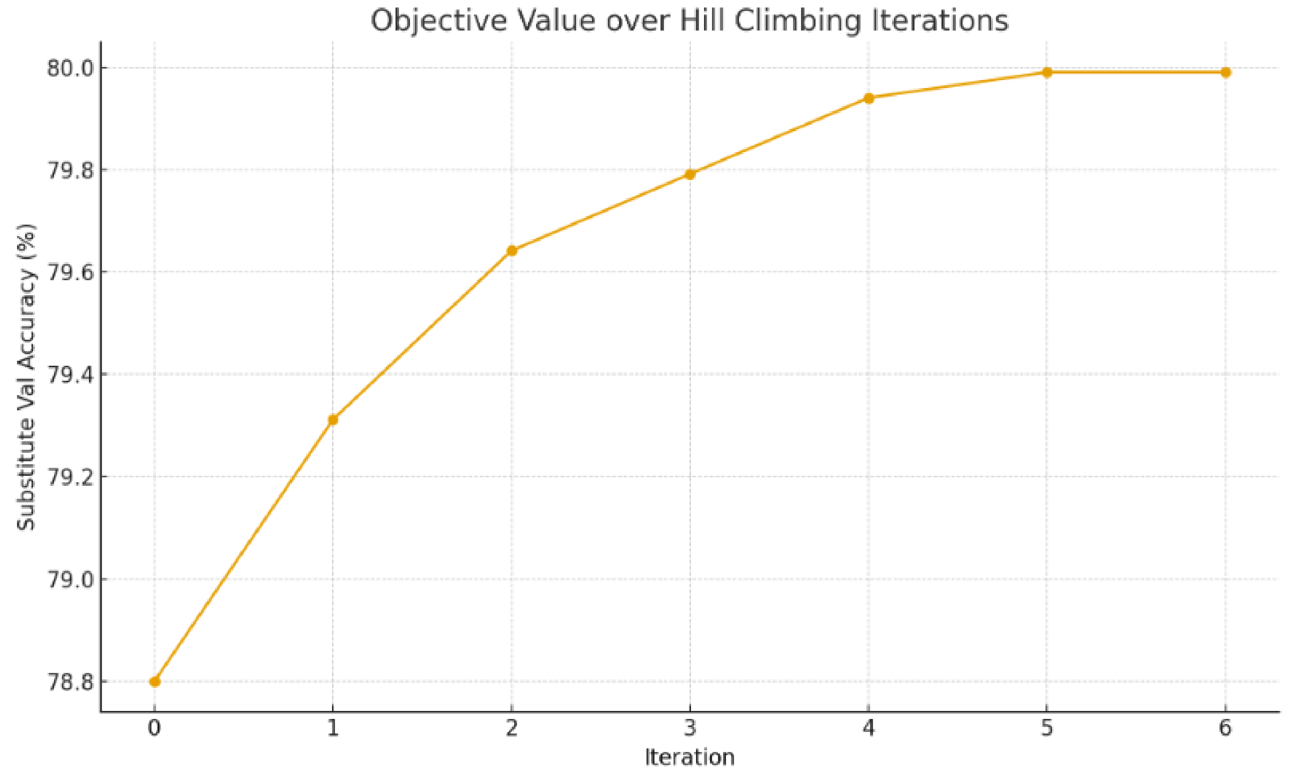
$$f(\eta, d) = 80 - (\log_{10}(\eta) + 2)^2 - 20(d - 0.2)^2$$

where:
The maximum validation is 80 and this will happens only at $\eta = 0.01 \ since \ (\log_{10(}0.01_{)} = -2)$, and dropout $d = 0.2$.
By Applying Hill Climbing
Initial State

$$s_0 = (\eta' = 0.001\ , d = 0.3)$$

Evaluate:

$$f(0.001, 0.3) = 80 - (-3 + 2)^2 - 20 \times (0.3 - 0.2)^2 = 80 - 1 - 0.2 = 78.8$$



Neighbors:

Neighborhood: change η by ×2 or ÷2, change d by ±0.05.

$f(0.0005, 0.3) = 80 - (-3.3 + 2)^2 - 20 \times (0.3 - 0.2)^2 \approx 77.3$

$f(0.002, 0.3) = 80 - (-2.7 + 2)^2 - 20 \times (0.3 - 0.2)^2 \approx 79.0$

$f(0.001, 0.25) = 80 - (-3 + 2)^2 - 20 \times (0.25 - 0.2)^2 \approx 79.95$

$f(0.001, 0.35) = 80 - (-3 + 2)^2 - 20 \times (0.35 - 0.2)^2 \approx 78.55$

Best = (0.001, 0.25) → 79.95

$$s_1 = (0.001, 0.25)$$

“The next iteration yielded no improvement, and the algorithm was consequently terminated.”

Note:

We change the neighborhood in that way because:

Learning rate (η) spans orders of magnitude, so ×2 or ÷2 is a natural log-scale step.

Dropout (d) is a probability between 0 and 1, so ±0.05 is a reasonable linear step that changes regularization without breaking training.

Stage 1:

```python
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.datasets import mnist
import numpy as np

# Load MNIST data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
x_train = x_train[..., np.newaxis]
x_test = x_test[..., np.newaxis]

# Build a simple CNN model
def build_model(eta, d):
    model = models.Sequential([
        layers.Conv2D(32, (3,3), activation='relu', input_shape=(28,28,1)),
        layers.MaxPooling2D((2,2)),
        layers.Dropout(d),
        layers.Flatten(),
        layers.Dense(64, activation='relu'),
        layers.Dropout(d),
        layers.Dense(10, activation='softmax')
    ])
    opt = optimizers.Adam(learning_rate=eta)
    model.compile(optimizer=opt, loss='sparse_categorical_crossentropy', metrics=['accuracy'])
    return model
```

Stage 2:

```python
# Evaluate model performance (validation accuracy)
def evaluate(eta, d, epochs=2):
    model = build_model(eta, d)
    history = model.fit(x_train, y_train, epochs=epochs, batch_size=128,
                        validation_split=0.2, verbose=0)
    val_acc = history.history['val_accuracy'][-1]
    return val_acc
```

Stage 3: The optimization part

```python
# Hill climbing algorithm
def hill_climb(initial_eta=0.001, initial_d=0.3, max_iters=5):
    eta, d = initial_eta, initial_d
    best_score = evaluate(eta, d)
    print(f"Initial: eta={eta}, d={d}, ValAcc={best_score:.4f}")
    for it in range(max_iters):
        # Define neighborhood
        neighbors = [
            (eta * 2, d), (eta / 2, d),
            (eta, d + 0.1), (eta, d - 0.1)
        ]
        # Keep valid neighbors only
        neighbors = [(e, dr) for e, dr in neighbors if e > 0 and 0 < dr < 1]
        # Evaluate neighbors
        scores = [(evaluate(e, dr), e, dr) for e, dr in neighbors]
        # Pick best neighbor
        best_neighbor = max(scores, key=lambda x: x[0], default=(best_score, eta, d))
        if best_neighbor[0] > best_score:  # improvement
            best_score, eta, d = best_neighbor
            print(f"Iter {it+1}: eta={eta}, d={d}, ValAcc={best_score:.4f}")
        else:
            print("No better neighbor found. Stopping.")
            break
    print(f"\nFinal solution: eta={eta}, d={d}, ValAcc={best_score:.4f}")
    return eta, d, best_score
# Run hill climbing
hill_climb()
```

Results:

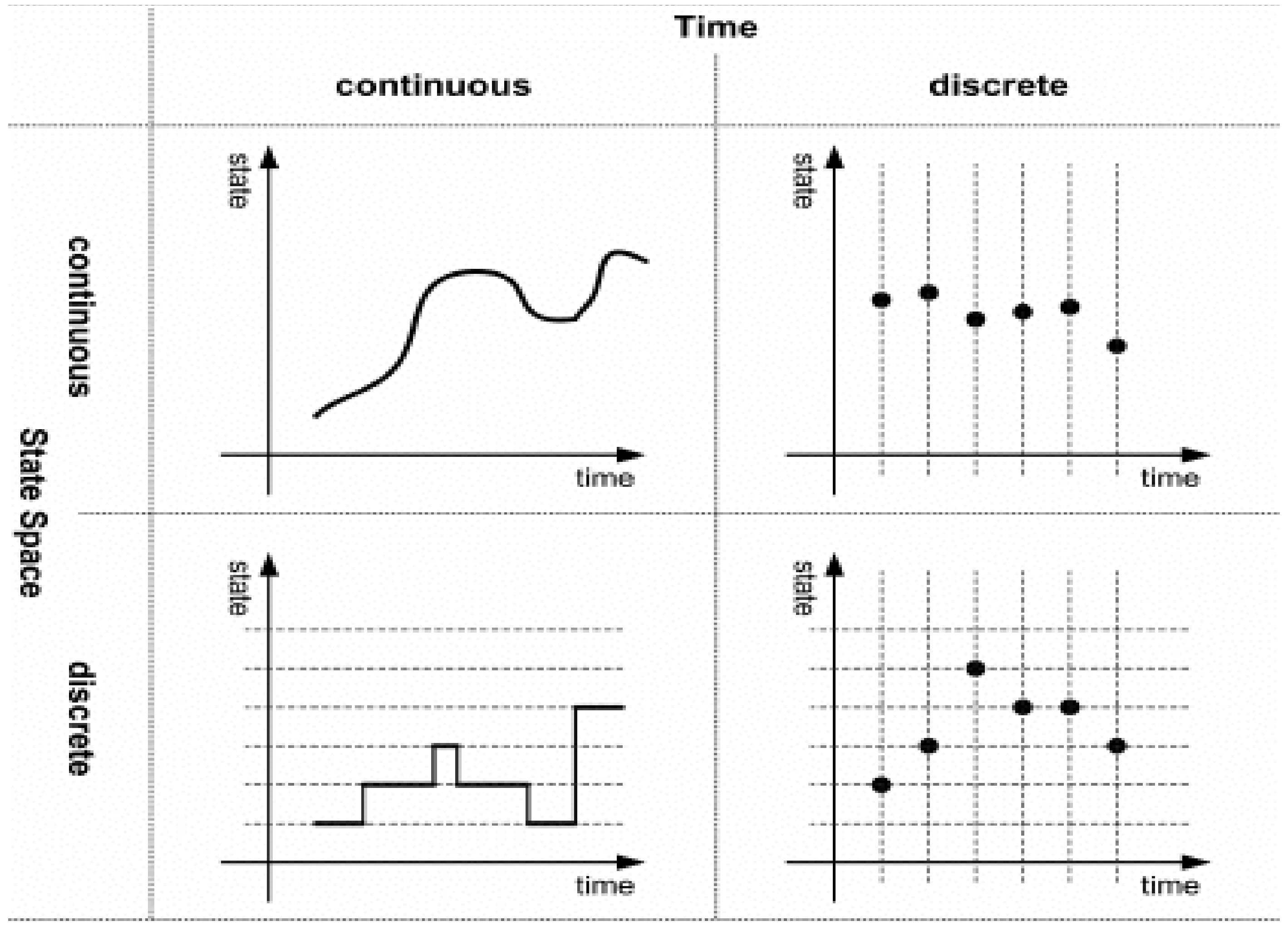
```
Initial: eta=0.001, d=0.3, ValAcc=0.9768
Iter 1: eta=0.002, d=0.3, ValAcc=0.9816
No better neighbor found. Stopping.

Final solution: eta=0.002, d=0.3, ValAcc=0.9816
(0.002, 0.3, 0.9815833568572998)
```

Neighborhood Structures: Continuous vs Discrete

The neighborhood structure determines:

- The efficiency of search
- The quality of local optima
- The computational cost of each iteration

Feature	Discrete	Continuous
Search space	Finite (combinatorial)	Infinite
Neighbor definition	Flip, swap, insert	Perturb, step
Visualization	Graph-based	Geometric/surface
Distance metric	Hamming / edit	Euclidean / norm-based



Why Hill Climbing Converges

Converge means that the algorithm stops changing significantly and settles down to a stable value.

1. Monotonic Improvement

- At each iteration, we choose a neighbor x_{t+1}such that: $f(x_{t+1}) \geq f(x_t)$
- This ensures the sequence of objective values is non-decreasing: $f(x_0) \leq f(x_1) \leq f(x_2) \leq \cdots$

2. Bounded Objective

- Most optimization problems have a bounded objective (e.g., accuracy ≤1, error ≥0). Validation accuracy can't exceed 100%, so improvements will slow down and eventually stop.
- By the **Monotone Convergence Theorem** states that:

A non-decreasing sequence that is bounded must converge: $\lim_{t\to\infty} f(x_t) = f^*$

3. Local Optimality

- Hill climbing stops when **no better neighbor exists**:

$$\forall x' \in N(x^*): f(x') \leq f(x^*)$$

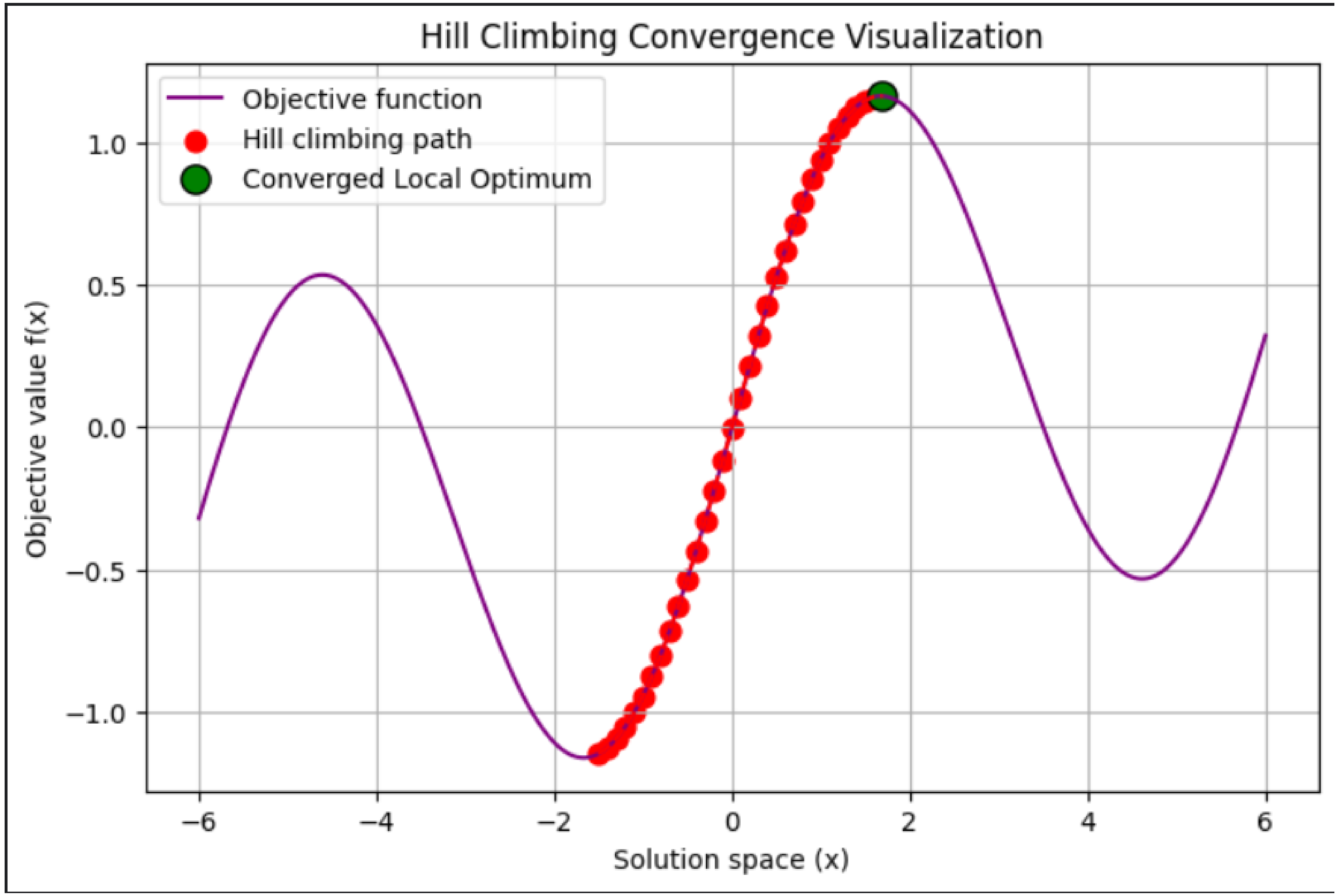
- This point x^* is a local optimum (may not be global).

4. Takeaways

- Hill climbing always converges to some solution.
- Guarantee: Local optimum, not global.
- Can be trapped → solution is to use **random restarts** or **metaheuristics** (like simulated annealing, genetic algorithms, particle swarm).

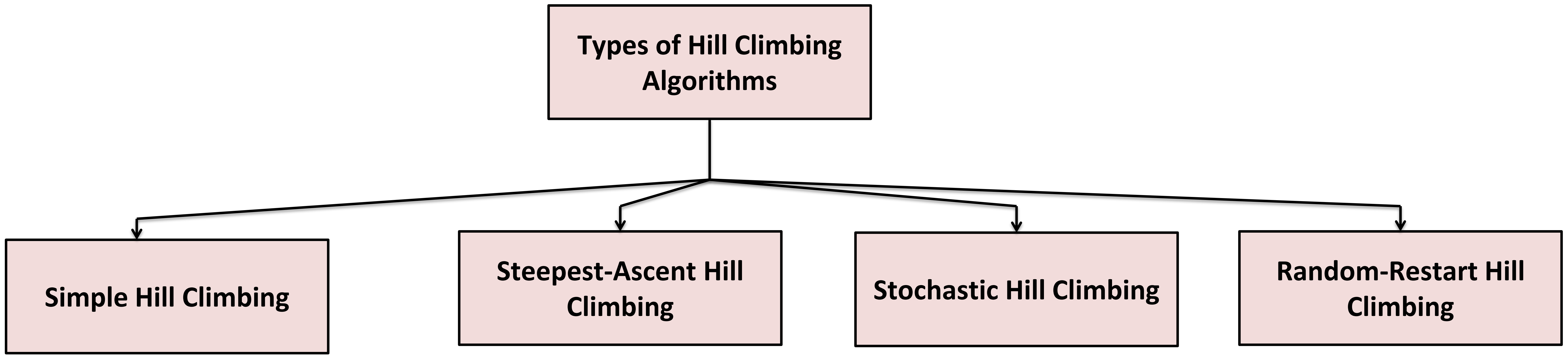
Why Hill Climbing Converges

Hill Climbing Convergence step by step on a simple 1D function (a parabola + sine bumps to create local optima).



Types of Hill Climbing Algorithms

Hill Climbing itself has **variants**, which improve performance depending on the problem:


Types of Hill Climbing Algorithms
Simple Hill Climbing
Steepest-Ascent Hill Climbing
Stochastic Hill Climbing
Random-Restart Hill Climbing

1) Simple Hill Climbing

- **Process:** scans neighbors in some (arbitrary) order and stops at the **first** improving neighbor.
- **Equation:** if first neighbor x' then $f(x') > f(x_t)$
- **Pros:** Fast, simple, low computation.
- **Cons:** May miss better neighbors → more prone to local maxima.

Example:

Increase learning rate if accuracy improves.

When to use:

- Quick tuning of a single hyperparameter (e.g., learning rate only).
- Situations where evaluations are cheap and you just need a “good enough” setting .
- In deep learning: Try increasing/decreasing learning rate step by step until validation accuracy stops improving.

2) Steepest-Ascent Hill Climbing

- **Process:** scans **all neighbors**, move to the one with the **largest improvement**.
- **Equation:** $x_{t+1} = \arg \max_{x' \in N(x_t)} f(x')$
- **Pros:** More systematic, better chance of escaping poor moves.
- **Cons:** Higher computation cost (must check all neighbors).

Example:

Test dropout ±0.05 and learning rate ×2/÷2, then pick the best.

When to use:

- Hyperparameter spaces are small enough that you can evaluate all neighbors .
- You need the best possible move each step.
- In deep learning: for tuning (η,d) = (learning rate, dropout), evaluate all 4 neighbors at each step and pick the one with the best validation accuracy.

3) Stochastic Hill Climbing

- **Process:** Choose a **random improving neighbor** instead of the best.
- **Equation (probabilistic):** $P(x_{t+1} = x') \propto f(x') \; for \; f(x') > f(x_t)$
- **Pros:** Introduces randomness, helps avoid local maxima.
- **Cons:** May accept smaller gains, slower convergence.

Example:

Randomly select between several better learning rates.

When to use:

- Training runs are expensive, so evaluating all neighbors is not feasible.
- You want some randomness to avoid always picking the same local maxima.
- **In deep learning,** randomly test one or two new dropout values from the neighborhood, accept any that improve validation accuracy.

4) Random-Restart Hill Climbing

- Process: Run Hill Climbing **multiple times** from different starting states.
- Equation: $x^* = \arg \max_{i=1,2,\dots.k} f\left(x^i_{final}\right)$
- Pros: Escapes local maxima, increases chance of global optimum.
- Cons: More computationally expensive.

Example:

Try learning rates starting from 0.001, 0.01, 0.1 separately, keep the best.

When to use:

- Hyperparameter space is rugged with many local maxima.
- You can afford several short runs instead of one long run.
- **In deep learning,** restart tuning from different initial learning rates (0.001, 0.01, 0.1) and keep the configuration that gives the highest validation accuracy.

Hill Climbing Variants: Greedy vs Steepest vs Stochastic

Variant 1	**Greedy (First Improvement)**	■ Scans neighbors in random or fixed order ■ Moves to the first neighbor that improves the cost ■ Fast but can miss better neighbors
Variant 2	**Steepest Ascent (Best Improvement)**	■ Evaluates all neighbors ■ Chooses the one with the best cost decrease ■ Finds deeper optima
Variant 3	**Stochastic Hill Climbing**	■ Picks a random neighbor ■ Accepts it only if it's better ■ Can escape shallow traps ■ May stagnate randomly

Failure Cases in Hill Climbing: Plateaus, Ridges & Local Minima

Failure Cases in Hill Climbing: Plateaus, Ridges & Local Minima

Failure Case	Cause	Effect on Search	Fix / Improvement
Plateaus	Flat region with equal neighbor values	Algorithm stops early (no direction to move)	Random restarts, add noise
Ridges	Gradient exists but only along diagonal directions	Slow zigzag progress	Use diagonal/variable neighborhood
Local Minima (false trap)	A local minimum is a point where all neighbors are worse, but it's not the global minimum.	Algorithm stuck in sub-optimal solution	Simulated annealing, tabu search

Example :

Task: classify digits (toy MNIST-like task).

- Model: small neural network (single hidden layer).
- Search space S: pairs $s = (h, \eta)$ where:

  h = number of hidden neurons (integer)

  η= learning rate (real)

Objective function: validation accuracy $f(h,\eta)$ (higher is better).

Neighborhood definition

From current s=(h,η), neighbors are :

$$N(s) = \{(h \pm 8, \eta), (h, \eta \pm 0.01)\}$$

$$with\ bounds\ 8 \leq h \leq 256\ , 0.0005 \leq \eta \leq 0.5.$$

Hill climbing type: Steepest-Ascent (choose the neighbor with highest validation accuracy).

Initialize:

- $s_0 = (h_0, \eta_0) = (32,\ 0.05)$
- Assume that : $f(32, 0.05) = 0.8700$ (87.00% validation accuracy)

Iteration 1:

- Evaluate neighbors of $(32, 0.05)$
- Assume that neighbors and their accuracies:

Candidate (h, η)	**$f(h, \eta)$**
(24, 0.05)	0.8500
(40, 0.05)	0.8900 ← best
(32, 0.04)	0.8800
(32, 0.06)	0.8600

- Best neighbor = (40, 0.05) with accuracy 0.8900.
- Move: $s1$=(40, 0.05) .

Iteration 2:

Neighbors of $(40, 0.05)$

Candidate (h, η)	**$f(h, \eta)$**
(32, 0.05)	0.8700
(48, 0.05)	0.9050 ← best
(40, 0.04)	0.9000
(40, 0.06)	0.8920

Iteration 3 :

Neighbors of $(48, 0.05)$

Candidate $(\mathbf{h}, \boldsymbol{\eta})$	$\mathbf{f}(\mathbf{h}, \boldsymbol{\eta})$
(40, 0.05)	0.8900
(56, 0.05)	0.9030
(48, 0.04)	0.9040
(48, 0.06)	0.9010

- The best neighbor accuracy = 0.9040 at (48,0.04) or 0.9030 at (56,0.05); none exceed current f(48,0.05)=0.9050.
- No improving neighbor found → terminate.

Final solution (local optimum):

$$(h^*, \eta^*) = (48, 0.05), \quad f(h^*, \eta^*) = 0.9050$$

- **Conclusion:** hill climbing improved from 0.8700 → 0.9050 (3.5% absolute gain in validation accuracy) but may still be a local optimum (not guaranteed global).