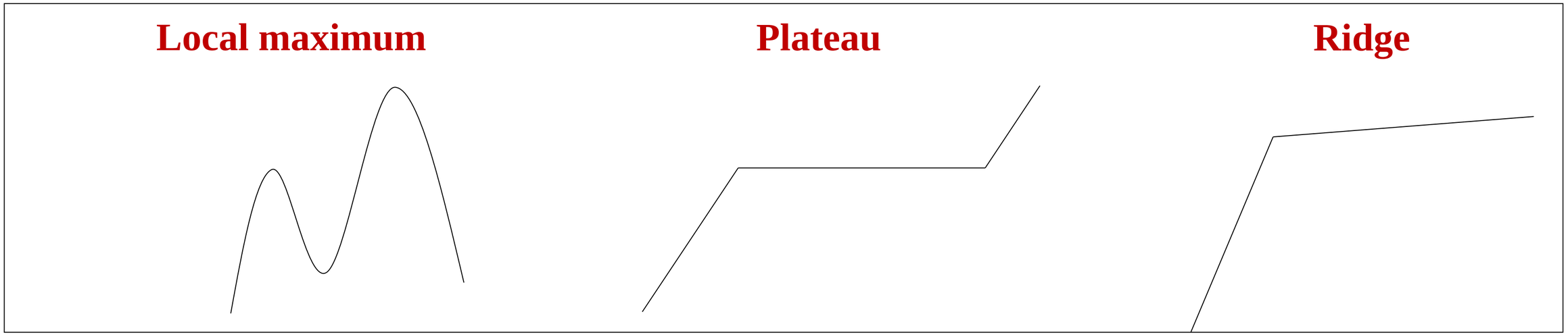
Nature Inspired Computation
DSAI 403

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

Hill-climbing search problems

- **Local maximum**: a peak that is lower than the highest peak, so a suboptimal solution is returned
- **Plateau**: the evaluation function is flat, resulting in a random walk
- **Ridges**: slopes very gently toward a peak, so the search may oscillate from side to side



Inspiration Families

Nature inspired

- These are inspired by biological, physical, or swarm intelligence
- **Examples:**

1. Simulated Annealing (thermal annealing process)
2. Gravitational Search Algorithm.
3. Particle Swarm Optimization (PSO)
4. Ant Colony Optimization (ACO)
5. Bee Colony Optimization
6. Genetic Algorithm (GA)
7. Differential Evolution (DE).

Human-Inspired

- These are inspired by human behavior, cognition, or learning
- **Examples:**

1. Tabu Search (memory-based decision making)
2. Scatter Search.

Hybrid

combinations of multiple inspirations

Nature inspired families of Metaheuristics

Physics/Chemistry-inspired

Swarm-inspired

Evolutionary-inspired

Simulated Annealing (SA)

It was motivated by the analogy between the physical annealing of metals and the process of searching for the optimal solution in a combinatorial optimization problem

One of the families of local search

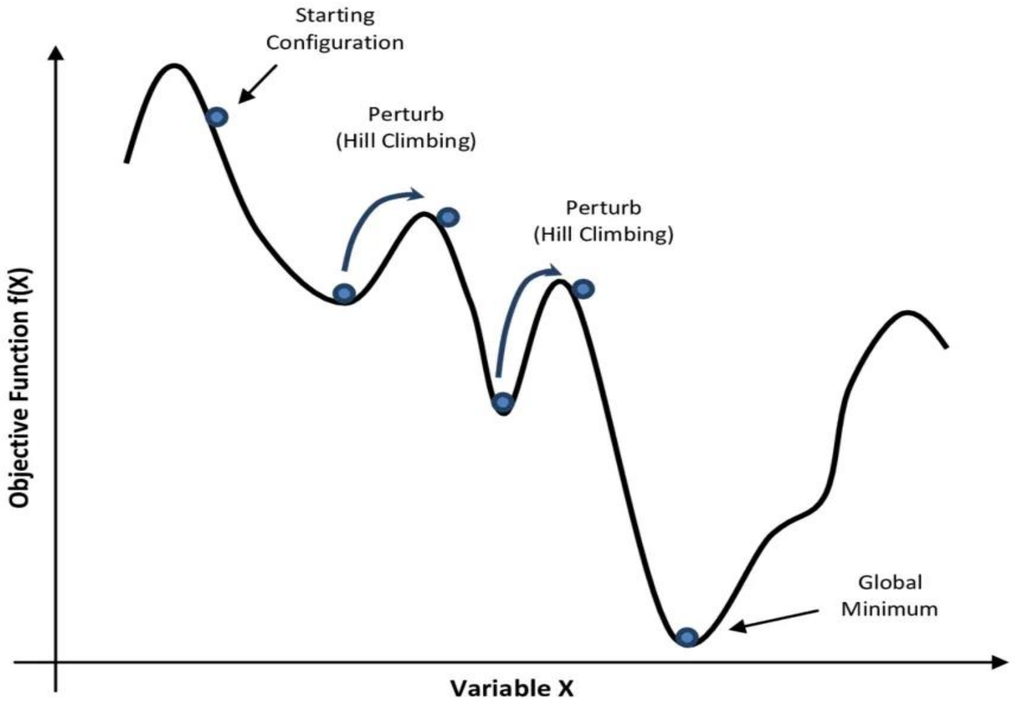
Simulated Annealing (SA)

(Metaheuristic Algorithm)

1) Local Search as a Metaheuristic

Why Hill Climbing / Local Search is Limited:

- Works with **one solution** at a time
- Works **only** with the current solution
- **Greedy**: always picks the best neighbor
- Fast and easy, but **problem-specific**
- Gets stuck in **local optima**, **plateaus**, **ridges**



Why We Need Something More

- Hill Climbing = no escape mechanism
- Needs global exploration strategy
- Needs controlled randomness to try worse solutions
- Real-world optimization is rugged → too many traps for greedy methods

Simulated Annealing

What is the meant by Annealing

- Annealing is a **physical** process used to **harden** **metals**, **glass**, etc.
- Starts at a high temperature and cools slowly.
- The temperature determines how much mobility the atoms have.
- How slowly you cool glass is critical.

Note that:

- When metal or glass is heated, its atoms move freely and randomly.
- If you **cool** it down **slowly**, the atoms have enough time to settle into a stable, low-energy arrangement, forming a strong, uniform structure.
- But if you **cool** it too **quickly**, the atoms get "frozen" in random positions, leading to a weaker or imperfect structure
- For cooling: $\boldsymbol{T_{new} = \alpha \times T_{old}}$, where α = cooling rate (a value between 0 and 1):

1. If you choose α close to **1** → slow cooling (temperature decreases slowly).
2. If you choose α smaller, e.g., 0.5 → fast cooling (temperature drops quickly).

Concept behind Simulated Annealing

- In Simulated Annealing, we model this process using optimization principles as follow:
- The “**temperature**” controls how much the algorithm explores random (even worse) solutions.
- At the start (high temperature), it explores widely, like atoms moving freely.
- As the “temperature” gradually decreases, the algorithm focuses on fine-tuning, like atoms slowly settling into their optimal structure.
- If “cooled” too fast, the algorithm might get stuck in a poor local minimum, just like glass that hardens too quickly.

Accordingly:

In simulated annealing:

- Start by exploring many possibilities.
- Allow some mistakes early.
- Gradually become more selective.
- Finish with the best possible answer

Simulated Annealing

Now think about solving a problem: we want to find the **best solution** (the strongest structure):

1. **Hill Climbing:**

- Acts like a person walking uphill in the dark, always stepping up, never down.
- If he reaches a small hill, he stops, thinking it's the top.
- But maybe there's a much higher mountain nearby however, he'll never find it.

2. **Simulated Annealing:** acts like a smarter explorer.

- At first, he sometimes goes **downhill on purpose** (tries worse solutions).
- Why? Because going down might help him later reach a **higher mountain**.
- As time goes on, he **reduces his risks** and focuses more carefully.
- In the end, he usually finds the **best overall peak**, not just a small hill.

Simulated Annealing

What is Simulated Annealing?

- A probabilistic search algorithm inspired by the annealing process in metallurgy.
- Tries to escape local minima by occasionally accepting worse solutions.
- Aims to find a near-global optimum instead of getting stuck.

How does the thermodynamic analogy map to the optimization algorithm?

- Physical System State → Current Candidate Solution: A possible solution to the problem you are trying to solve.
- Energy of the State → Value of the Objective Function: The "goodness" of the candidate solution (the algorithm tries to minimize or maximize this value).
- Temperature → Control Parameter (T): A numerical value that starts high and decreases over time, controlling the algorithm's behavior.
- Ground State → Global Optimum: The best possible solution to the problem.

Simulated Annealing (SA) is considered both a local search AND a metaheuristic.

Local Search

Local search starts from a solution and explores its neighborhood $N(x)$ to improve it.

Why SA fits:

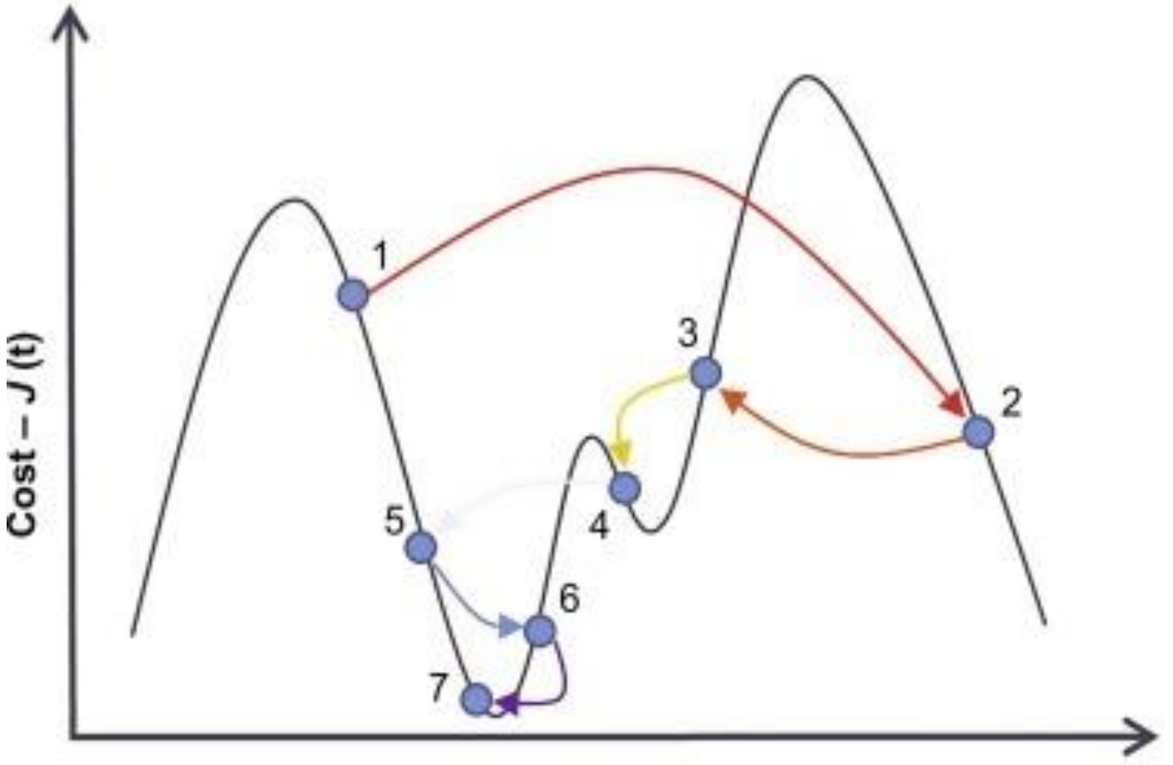
- SA starts from an initial solution.
- At each step, it selects a neighbor $x' \in N(x)$.
- It decides whether to move based on the objective $f(x)$and probability.

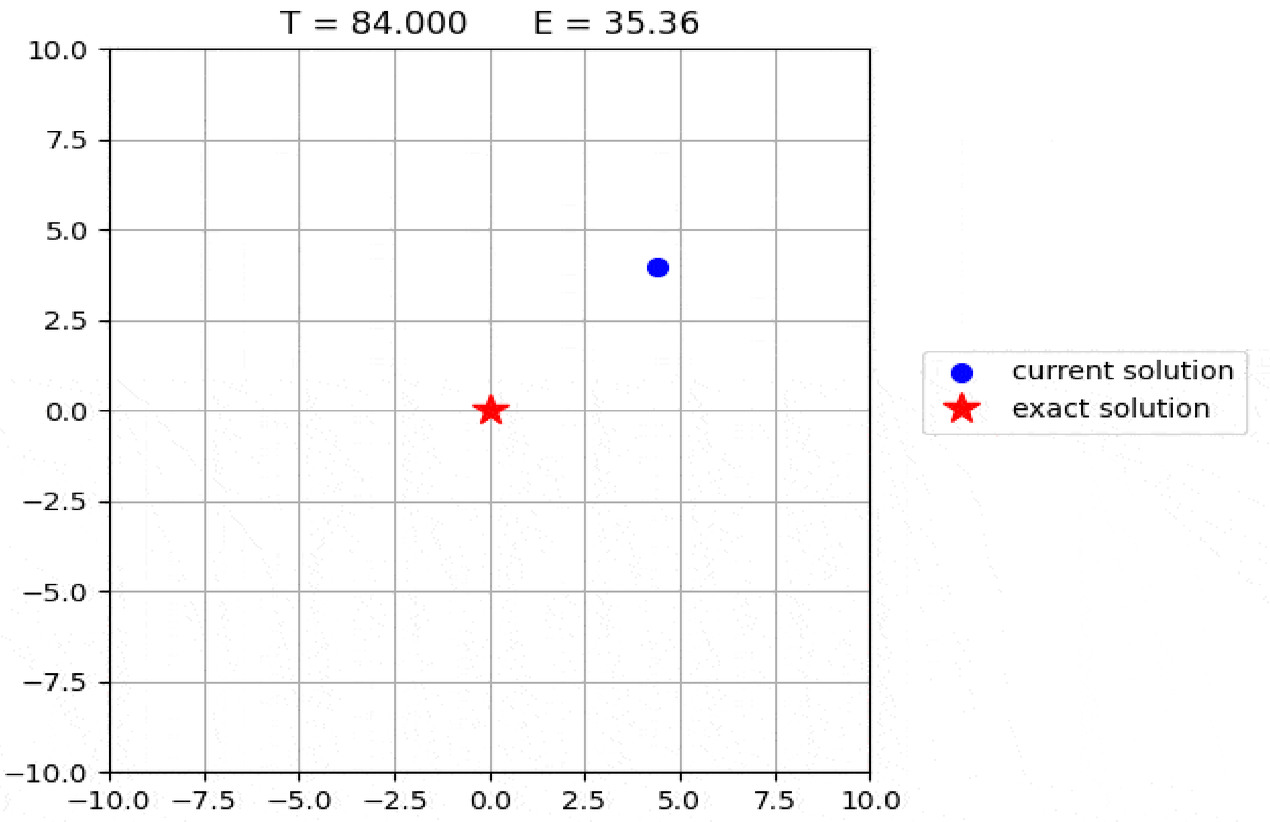
Metaheuristic

Metaheuristics are higher-level strategies that guide local search to avoid getting trapped in local optima and improve exploration.

Why SA fits:

- Unlike pure local search (hill climbing), SA allows probabilistic acceptance of worse solutions (via Boltzmann probability).
- The temperature schedule T is a control mechanism → starts exploratory (random moves) and gradually becomes greedy.
- This makes it general-purpose, flexible, and applicable to many optimization problems (neural nets, scheduling, combinatorial optimization, etc.)


Cost – J (t)
1
2
3
4
5
6
7


T = 84.000
E = 35.36
10.0
7.5
5.0
2.5
0.0
−2.5
−5.0
−7.5
−10.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
current solution
exact solution

Hyperparameter Tuning in a Neural Network

We want to tune a single Hyperparameter ; **the learning rate (η)** in order to maximize model accuracy (or equivalently minimize validation loss).
However, the relationship between learning rate and loss is highly non-linear and noisy, due to randomness in training and local minima.
We can model it with a mock **objective function:**

$$L(w) = sin(3w) + 0.5w^2 - cos(5w), w \in [-4, 4]$$

- Start: w_0=2.5
- Step size: δ=0.1
- Neighborhood: $N(w) = \{w - \delta, w + \delta\}$
- Rule: move to a neighbor only if its loss is strictly lower than current (greedy hill climbing).
- Stop when neither neighbor improves loss.

By Using Hill Climbing Technique
Compute each term of $L(w)$ (radians):
- sin(3w)=sin(3 x 2.5)≈0.1305
- 0.5 w^2=0.5×(2.5)^2=3.125
- cos(5w)=cos(5 x 2.5)≈0.9762

So $L(\mathrm{w}) = 0.1305 + 3.125 - 0.9762 = 2.2838$

For Left neighbor
W_L = W_0 - δ $= 2.5 - 0.1 = 2.4$
- sin(3w)=sin(3 x 2.4)≈0.1253
- 0.5 w^2=0.5×(2.4)^2=2.88
- cos(5w)=cos(5 x 2.4)≈0.9781

So $L(\mathrm{w}) = 0.1253 + 2.88 - 0.9781 = 2.0272$

For Right neighbor
W_L = W_0 + δ $= 2.5 + 0.1 = 2.6$
- sin(3w)=sin(3 x 2.6)≈0.1357
- 0.5 w^2=0.5×(2.6)^2=3.38
- cos(5w)=cos(5 x 2.6)≈0.9743

So $L(\mathrm{w}) = 0.1357 + 3.3 - 0.9743 = 2.4614$

Comparison & decision (first iteration):

Current: $L(2.5) = 2.2838$

Left neighbor: $L(2.4) = 2.0272$ → improvement of $\boldsymbol{\Delta}_{L\ left} = L(2.4) - L(2.5) \approx 2.0272 - 2.2838 = -0.2566$ (**Best**)

Right neighbor: $L(2.6) = 2.4614$ → change $\boldsymbol{\Delta}_{L\ right} \approx 2.4614 - 2.2838 = 0.1776$ (**worse**)

Rule: accept a neighbor only if it decreases L. The left neighbor decreases L, the right does not.

So we move to the left neighbor:

$$W_1 = 2.4\,, L(w_1) = 2.0272$$

Iter	w (current)	L(w) (value)
0	2.5	2.2838
1	2.4	2.0272
2	2.3	1.7852
…	…	…
12	1.3	-0.0805

Stopping condition reached after (for example) Iter 12: at w=1.3, neither neighbor 1.2 nor 1.4 yields a strictly lower loss, so the greedy hill-climber stops.

Final (**local**) solution found: $w^{\star}$=1.3, $L(w^{\star})\approx$–0.0805.

Short conclusion (why hill climbing got stuck):

- The hill-climbing run **improved monotonically** from $\boldsymbol{w = 2.5}$ to $\boldsymbol{w = 1.3}$, following immediate downhill neighbors.
- **It stops at $\boldsymbol{w = 1.3}$** because both neighbors are worse (no immediate downhill move). That is the definition of a **local minimum** for this neighborhood definition.
- A separate global scan over [-4 , 4] shows a **better (lower) loss near $\boldsymbol{w \approx -0.115}$ with $\boldsymbol{L \approx -1.1708}$.** Thus the hill-climber **did not** find the global minimum, it became trapped in a nearer local basin.
- classic hill-climbing con: **fast and greedy**, but **start-dependent** and **unable to cross temporary rises** (you would have to accept worse loss temporarily to reach the better basin).

Escape the Trap (Local Minimum Solution)
Using Simulated Annealing

For Example:

- We want to **minimize** $L(w)$.
- Problem: it has many local minima → hill climbing gets **stuck**.
- Solution: **Simulated Annealing** (SA) allows temporary uphill moves to **escape traps.**

Step 0: Initialization

Current solution: $w = 2.0$

Current loss:

$$L(w) = sin(3w) + 0.5w^2 - cos(5w), w \in [-4, 4]$$

$$L(w) = sin(3 \times 2\ ) + 0.5(2)^2 - cos(5 \times 2\ )\ =\ 1.119$$

So, Current = (2.0, Loss = 1.119).

Step 1: Generate a Neighbor

Let's perturb x randomly.

Suppose the neighbor is: $\boldsymbol{x'}$**=2.3**

Compute its loss:

$$L(w) = sin(3 \times 2.9) + 0.5(2.9)^2 - cos(5 \times 2.9)\ =\ 3.388$$

So, Neighbor = (2.9, Loss = 3.388).

Step 2: Compare Losses

Current Loss = 1.119

Neighbor Loss = 3.388

Since **3.388 > 1,** the new neighbor is **worse**.

- In **Hill Climbing**, we would reject this neighbor immediately.
- In **Simulated Annealing**, we give it a **chance to be accepted**.

Step 3: Compute the acceptance probability in SA (Metropolis criterion):

We compute acceptance probability according to **Boltzmann distribution** :

$$P = e^{(-\Delta L/\mathrm{T})}$$

where:

- $\Delta L = NeighborLoss - CurrentLoss = 3.388 - 1.119 = 2.269$
- Assume **Temperature** $T = 2.0$

$$P = e^{-2.269/2.0} = e^{-2.12} \approx 0.32$$

So, there's a **32% chance** we accept this worse neighbor.

Step 4: Random Decision

- Generate a random number $r \in [0, 1]$.
- Case A: If $r = 0.05 < 0.32$→ Accept the worse neighbor (we move to $x = 2.9$).
- Case B: If $r = 0.30 > 0.32$→ Reject, stay at $x = 2.0$.

In Summary

- Hill Climbing → rejects worse moves, gets stuck.
- Simulated Annealing → sometimes accepts worse moves → helps escape local minima.
- This "**controlled randomness**" is why SA is metaheuristic and more powerful.

Step 5: Cooling

- Now, we reduce the temperature slightly, for example:

$T_{new} = \alpha \times T_{old} = 0.9 \times 2 = 1.8,$ where α = cooling rate (a value between 0 and 1)

- T decreases, the algorithm becomes **less likely to accept worse moves**, focusing more on refinement.

Example:

It is required to optimize learning rate (LR) in order to minimize validation loss of a neural network:

$$E(LR) = (LR - 0.3)^2 + 0.01$$

Assume the following optimization parameters:

Initial learning rate	$LR = 0.6$
Initial temperature	$T_0 = 1$
Cooling factor	$\alpha = 0.5$
Minimum temperature	$T_{min} = 0.1$
Random values	$r_1 = 0.3, r_2 = 0.8, r_3 = 0.4$
Neighbor function	We generate neighbors by decreasing the learning rate by 0.1 twice, then increasing it by 0.1 once in order to explore both lower and higher regions step by step

Solution:

initialization	$E_0 = (0.6 - 0.3)^2 + 0.01 = \mathbf{0.1}$ And, $T_0 = 1$
Iteration: 1	$LR_{new} = 0.6 - 0.1 = 0.5$ $E_{new} = (0.5 - 0.3)^2 + 0.01 = \mathbf{0.05}$ $\Delta E = 0.05 - 0.1 = -0.05$ It is Better → accept directly $T_1 = \alpha \times T_0 = 0.5$
Iteration: 2	$LR_{new} = 0.5 - 0.1 = 0.4$ $E_{new} = (0.4 - 0.3)^2 + 0.01 = \mathbf{0.02}$ $\Delta E = 0.02 - 0.05 = -0.03$ It is Better → accept directly $T_2 = \alpha \times T_1 = 0.25$
Iteration: 3	$LR_{new} = 0.4 + 0.1 = 0.5$ $E_{new} = (0.5 - 0.3)^2 + 0.01 = \mathbf{0.05}$ $\Delta E = 0.05 - 0.02 = +0.03$ It is worse → apply Metropolis criterion and use the cooled temperature from previous step $P = e^{(-\Delta E/\mathrm{T})} = e^{-0.03/0.25} \approx 0.89$ Given that: $r_3 = 0.4$ Since, $r < P$, accept the worse move $T_3 = \alpha \times T_1 = 0.125$ (final cool reaches the minimum temperature, accordingly, we stop)
Final results	**Best LR found:** 0.4 **Minimum cost:** 0.02

Important note: If iteration 3 is **rejected**; that means the **worse solution is not accepted**. So the **current solution and cost remain the same** as from iteration 2.

```python
import numpy as np, random, math, tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.optimizers import Adam

# ---------- Step 0: Initialization ----------
(x_train, y_train), _ = mnist.load_data()
x_train = x_train[:3000] / 255.0; y_train = y_train[:3000]

def model_loss(lr):
    model = Sequential([
        Flatten(input_shape=(28,28)),
        Dense(64, activation='relu'),
        Dense(10, activation='softmax')
    ])
    model.compile(optimizer=Adam(learning_rate=lr),
                  loss='sparse_categorical_crossentropy')
    h = model.fit(x_train, y_train, epochs=1, batch_size=128,
                  validation_split=0.2, verbose=0)
    return h.history['val_loss'][-1]

T, alpha, min_T = 2.0, 0.85, 0.1
lr, loss = 0.001, model_loss(0.001)
print(f"Step 0: Start → lr={lr:.5f}, Loss={loss:.4f}")



# ---------- Step 1–6: Iterative Optimization ----------
best_lr, best_loss, step = lr, loss, 0
while T > min_T:
    step += 1
    print(f"\n--- Step {step} ---")
```

Step 1: Generate Neighbor	`new_lr = max(0.0001, min(lr + np.random.uniform(-0.0005, 0.0005), 0.01))`
Step 2: Compute Neighbor Loss	`new_loss = model_loss(new_lr)` `print(f"Neighbor → lr={new_lr:.5f}, Loss={new_loss:.4f}")`
Step 3: Compute ΔL	`dL = new_loss - loss` `print(f"ΔL={dL:+.4f}")`
Step 4: Acceptance Probability	`P = 1.0 if dL < 0 else math.exp(-dL / T)` `print(f"T={T:.3f}, P={P:.3f}")`

Step 5: Random Decision	`r = random.random()`
	`if r < P:`
	`  lr, loss = new_lr, new_loss`
	`  print(f"Accepted ✅ (r={r:.3f}<P)")`
	`else:`
	`  print(f"Rejected ❌ (r={r:.3f}≥P)")`
	
	`if loss < best_loss:`
	`  best_lr, best_loss = lr, loss`

Step 6: Cooling	`T *= alpha` `print(f"Cool → T={T:.3f}")`

Final stage: Print results	`print("\n====================")`
	`print(f"Best lr={best_lr:.5f}, Best Loss={best_loss:.4f}")`
	`print("====================")`

```
Step 0: Start → lr=0.00100, Loss=1.1387

--- Step 1 ---
Neighbor → lr=0.00144, Loss=0.9483
ΔL=-0.1904
T=2.000, P=1.000
Accepted ✅ (r=0.461<P)
Cool → T=1.700

--- Step 2 ---
Neighbor → lr=0.00168, Loss=0.7905
ΔL=-0.1578
T=1.700, P=1.000
Accepted ✅ (r=0.685<P)
Cool → T=1.445

--- Step 3 ---
Neighbor → lr=0.00167, Loss=0.8570
ΔL=+0.0665
T=1.445, P=0.955
Accepted ✅ (r=0.376<P)
Cool → T=1.228

--- Step 4 ---
Neighbor → lr=0.00140, Loss=0.9145
ΔL=+0.0574
T=1.228, P=0.954
Accepted ✅ (r=0.482<P)
Cool → T=1.044

--- Step 5 ---
Neighbor → lr=0.00116, Loss=1.1014
ΔL=+0.1870
T=1.044, P=0.836
Accepted ✅ (r=0.275<P)
Cool → T=0.887

--- Step 6 ---
Neighbor → lr=0.00071, Loss=1.4291
ΔL=+0.3276
T=0.887, P=0.691
Rejected ❌ (r=0.961≥P)
Cool → T=0.754

--- Step 7 ---
Neighbor → lr=0.00109, Loss=1.0744
ΔL=-0.0270
T=0.754, P=1.000
Accepted ✅ (r=0.155<P)
Cool → T=0.641

--- Step 8 ---
Neighbor → lr=0.00090, Loss=1.3035
ΔL=+0.2290
T=0.641, P=0.700
Rejected ❌ (r=0.805≥P)
Cool → T=0.545

--- Step 9 ---
Neighbor → lr=0.00129, Loss=1.0400
ΔL=-0.0345
T=0.545, P=1.000
Accepted ✅ (r=0.262<P)
Cool → T=0.463

--- Step 10 ---
Neighbor → lr=0.00116, Loss=1.0260
ΔL=-0.0140
T=0.463, P=1.000
Accepted ✅ (r=0.785<P)
Cool → T=0.394

--- Step 11 ---
Neighbor → lr=0.00084, Loss=1.2456
ΔL=+0.2197
T=0.394, P=0.572
Rejected ❌ (r=0.592≥P)
Cool → T=0.335

--- Step 12 ---
Neighbor → lr=0.00069, Loss=1.5566
ΔL=+0.5306
T=0.335, P=0.205
Rejected ❌ (r=0.234≥P)
Cool → T=0.284

--- Step 13 ---
Neighbor → lr=0.00073, Loss=1.5600
ΔL=+0.5341
T=0.284, P=0.153
Rejected ❌ (r=0.344≥P)
Cool → T=0.242

--- Step 14 ---
Neighbor → lr=0.00134, Loss=1.0117
ΔL=-0.0143
T=0.242, P=1.000
Accepted ✅ (r=0.563<P)
Cool → T=0.206

--- Step 15 ---
Neighbor → lr=0.00130, Loss=0.9844
ΔL=-0.0273
T=0.206, P=1.000
Accepted ✅ (r=0.828<P)
Cool → T=0.175

--- Step 16 ---
Neighbor → lr=0.00133, Loss=1.0032
ΔL=+0.0188
T=0.175, P=0.898
Accepted ✅ (r=0.562<P)
Cool → T=0.149

--- Step 17 ---
Neighbor → lr=0.00173, Loss=0.8209
ΔL=-0.1822
T=0.149, P=1.000
Accepted ✅ (r=0.927<P)
Cool → T=0.126

--- Step 18 ---
Neighbor → lr=0.00222, Loss=0.7279
ΔL=-0.0930
T=0.126, P=1.000
Accepted ✅ (r=0.932<P)
Cool → T=0.107

--- Step 19 ---
Neighbor → lr=0.00194, Loss=0.7358
ΔL=+0.0079
T=0.107, P=0.929
Accepted ✅ (r=0.465<P)
Cool → T=0.091
```

Final result

```
====================
Best lr=0.00222, Best Loss=0.7279
====================
```

Why we have about 19 iterations:

$$T_{K+1} = \alpha \times T_K$$

At start: $T_0 = 2$

1st iteration: $T_1 = \alpha \times T_0 = 2 \times \alpha$

2nd iteration: $T_2 = \alpha \times T_1 = 2 \times \alpha^2$

3rd iteration: $T_3 = \alpha \times T_2 = 2 \times \alpha^3$

After K iterations: $T_k = 2 \times \alpha^k = T_0 \times \alpha^k$

Given that:

$T_0 = 2$

$\alpha = 0.85$ (cooling rate)

$T_{min} = 0.1$

We will stop when: $T_K < T_{min}$

$$2 \times (0.85)^k < 0.1$$
$$(0.85)^k < 0.05$$
$$K > \frac{\ln(0.05)}{\ln(0.85)}$$
$$K > 18.4$$
$$\boldsymbol{K \approx 19}$$

why Simulated Annealing (SA) is better than Hill Climbing (HC).

Why Hill Climbing Gets Stuck

In Hill Climbing, the rule is simple:

- If the neighbor is better (lower loss) → move there.
- If the neighbor is worse (higher loss) → reject it.

This works fine until:

- You hit a local minimum (a "valley" where all nearby neighbors are worse).
- At that point, HC refuses to move because every neighbor looks worse.
- Result: stuck, even if a much better global minimum exists further away**.**

Example: Imagine trying to reach the lowest valley but you stop in a small hole because climbing out first increases your loss → HC can't escape.

What Simulated Annealing Does Differently

Simulated Annealing introduces controlled randomness:

- If neighbor is better → accept (same as HC).
- If neighbor is worse → maybe accept with probability:

$$P = e^{(-\Delta L/\mathrm{T})}$$

- ΔL = NeighborLoss−CurrentLoss
- T= "temperature" (starts high, then cools down).

Why This Helps

At high T (early in search):

- Even bad moves have a good chance to be accepted.
- Algorithm jumps around a lot, exploring widely.
- This lets it escape local minima.

At low T (later in search):

- Acceptance probability for worse moves becomes very small.
- Algorithm behaves more like Hill Climbing, refining the best area found.
- This ensures convergence near an optimum.

Simulated Annealing as a Metaheuristic

Exploration Phase (High Temperature)

1. Start with a random solution.
2. Accept both better and worse solutions (high probability of escape).
3. Ensures global exploration of the search space .
4. Helps avoid getting trapped in local minima early on.

Exploitation Phase (Low Temperature)

1. Temperature decreases gradually (cooling schedule).
2. Worse solutions are less likely to be accepted.
3. Focus shifts to refining and improving the solution.
4. Behaves like greedy local search at the end.

Simulated Annealing - Pseudocode

1. Initialize:

- Start with a random solution S
- Compute its cost E = cost(S)
- Set initial temperature T = T0

2. Repeat until T < Tmin:

a. Generate a new solution S_new by a small random change in S

b. Compute E_new = cost(S_new)

c. If (E_new < E):

Accept new solution → S = S_new, E = E_new

Else:

Compute P = exp(−(E_new − E) / T)

Generate random number r ∈ [0, 1]

If (r < P):

Accept S_new → S = S_new, E = E_new

Else:

Reject S_new

d. Decrease temperature: T = α × T (e.g., α = 0.85)

e. Record best solution so far

3. Return the best solution found

Failure Cases for Simulated Annealing

When Does Simulated Annealing Fail?

Premature Convergence (Stops Too Early)	▪ Cools too fast and stops exploring. ▪ Gets stuck in a local minimum (not the best answer). ▪ Like cooling metal too quickly, atoms cannot move to better position **How to solve it:** ▪ Adjust cooling rate, allow for more iterations
Bad Temperature Schedule	▪ Too fast cooling: algorithm freezes early. ▪ Too slow cooling: wastes time searching. ▪ Needs the right balance between exploration and speed **How to solve it:** ▪ Experiment with slower cooling rates or longer decay
Weak Neighbor Generation	▪ New solutions are too similar, causing poor exploration. ▪ The algorithm cannot escape bad areas. ▪ We need enough randomness to discover better solutions. **How to solve it:** ▪ Increase search diversity, use hybrid methods