Neural Simulated Annealing (NSA)

Ishanya (21329)
Hari Krishna (22236)
Hiba (22146)
Astha (22063)

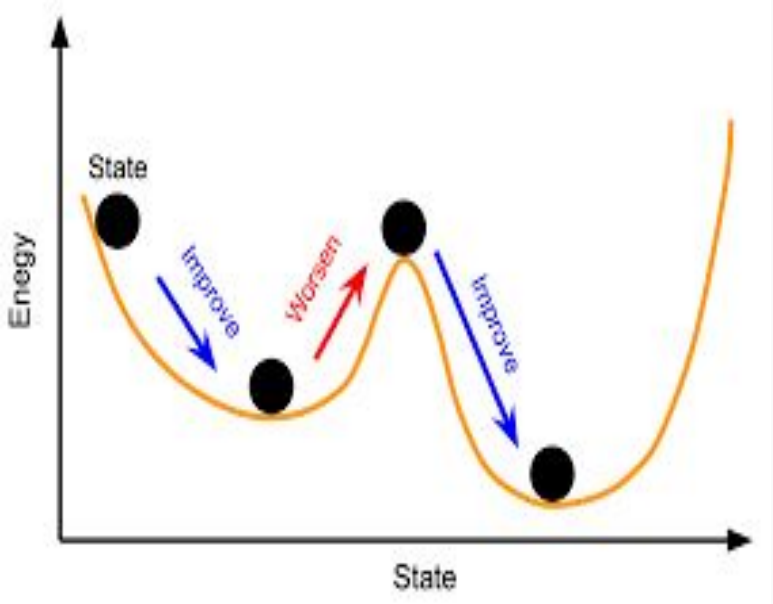
GitHub link: ishanyaa/SAOptimisation

DSE/ECS 311
Project Presentation

Indian Institute of Science Education and Research Bhopal (IISERB)

Simulated Annealing(SA)

- **Mechanism:**
  Based on how metals cool and crystallize—starts “hot” (exploring freely) and gradually “cools” (becomes more selective), occasionally accepting worse moves to escape local optima.
- **Key Components:**
  State: a candidate solution
  Neighbourhood: small random perturbations of the state
  Temperature Schedule: controls acceptance of worse moves, decreasing over time
- **Goal**: Find a (near-)optimal solution by balancing exploration (high temperature) and exploitation (low temperature).



Neural Simulated Annealing (Neural SA)

1. **Learned Proposals**: Instead of picking random changes, a neural network learns which changes are likely to work better and suggests those.

2. **Adaptive Strategy**: The network adapts its proposal distribution based on past search history, guiding the SA process more intelligently.

3. **Hybrid Optimization**: Retains SA's acceptance rule (temperature-based) but uses learned insights to explore high-value regions faster.

Why Use Neural SA?

1. **Better Proposals → Faster Convergence**: Learned moves focus on promising areas, reducing wasted random exploration.

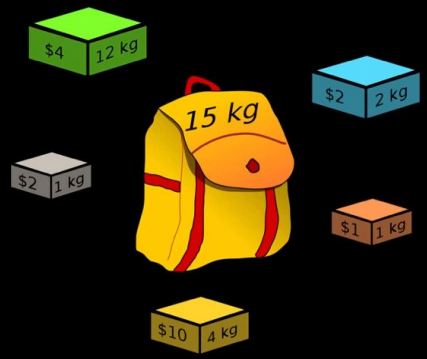
2. **Generalization Across Instances**: Once trained, the network can guide SA on similar problem instances without hand-crafting heuristics.

3. **Higher Solution Quality**: Combining neural guidance with SA's principled exploration often yields superior final solutions compared to vanilla SA alone.

Knapsack Problem

Problem statement

Given a set of objects which have both a value and a weight (v_i, w_i) what is the **maximum value** we can obtain by selecting a subset of these objects such that the sum of the weights does not exceed a certain capacity (i.e the knapsack capacity)?



Knapsack

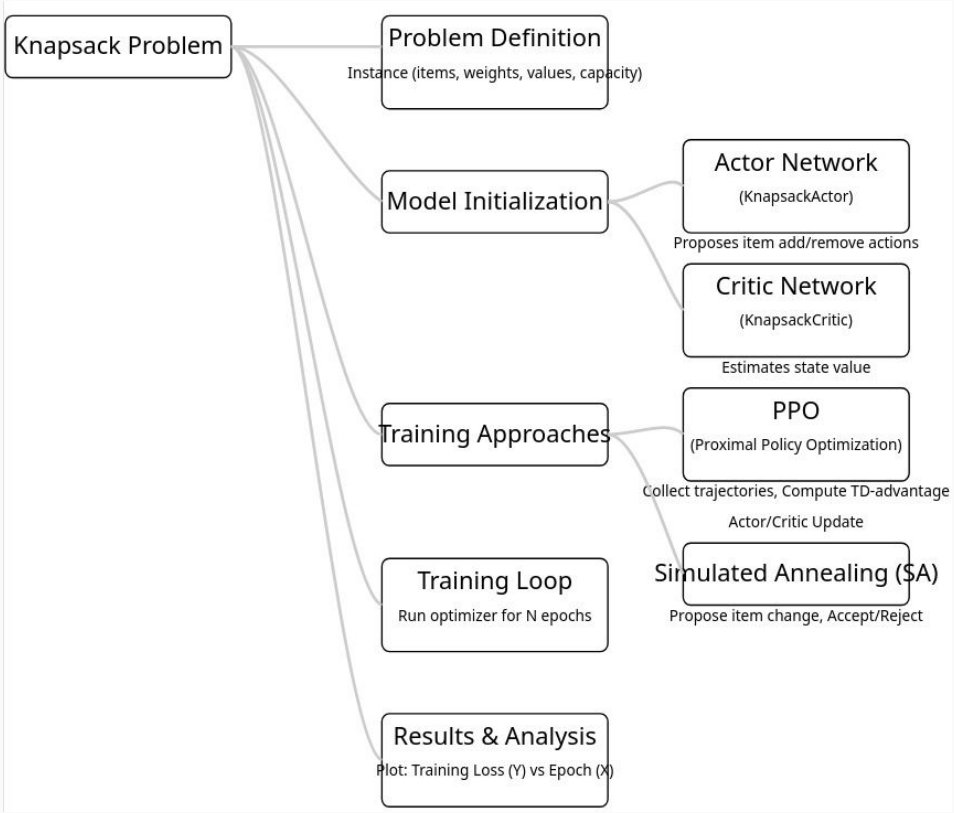
Problem: Select a subset of `N` items with values `v_i > 0` and weights `w_i > 0` to maximize total value without exceeding total weight `W`.

Formulation:

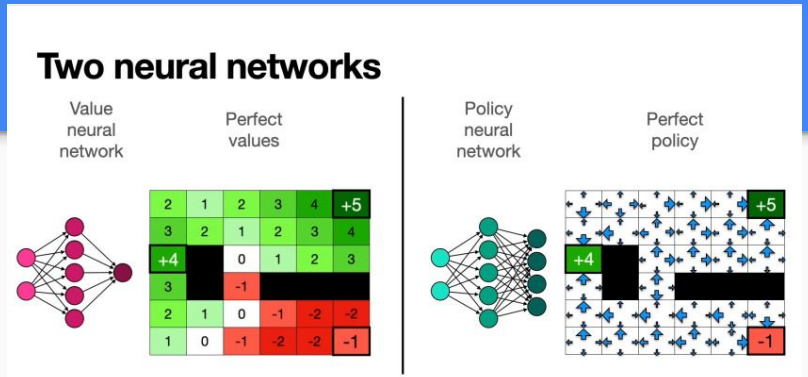
- Maximize: $\sum_{i=0}^{N-1} v_i x_i$
- Subject to: $\sum_{i=0}^{N-1} w_i x_i \leq W$, with $x_i \in \{0, 1\}$

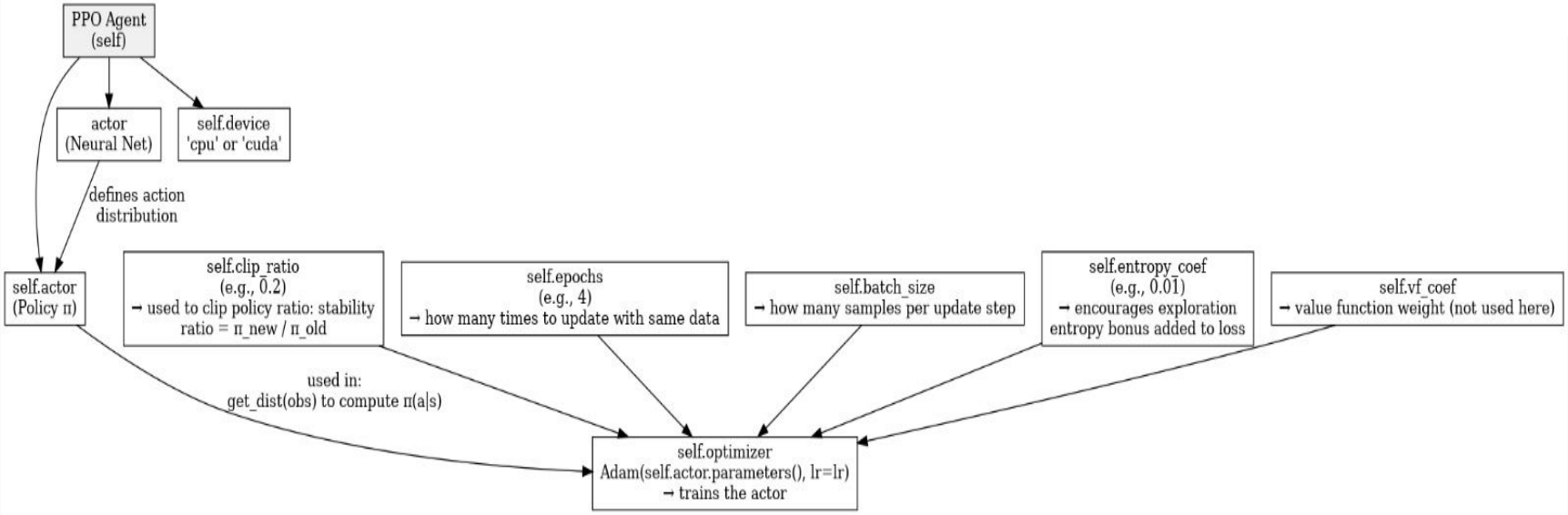
PPO-Based Neural Knapsack Solver

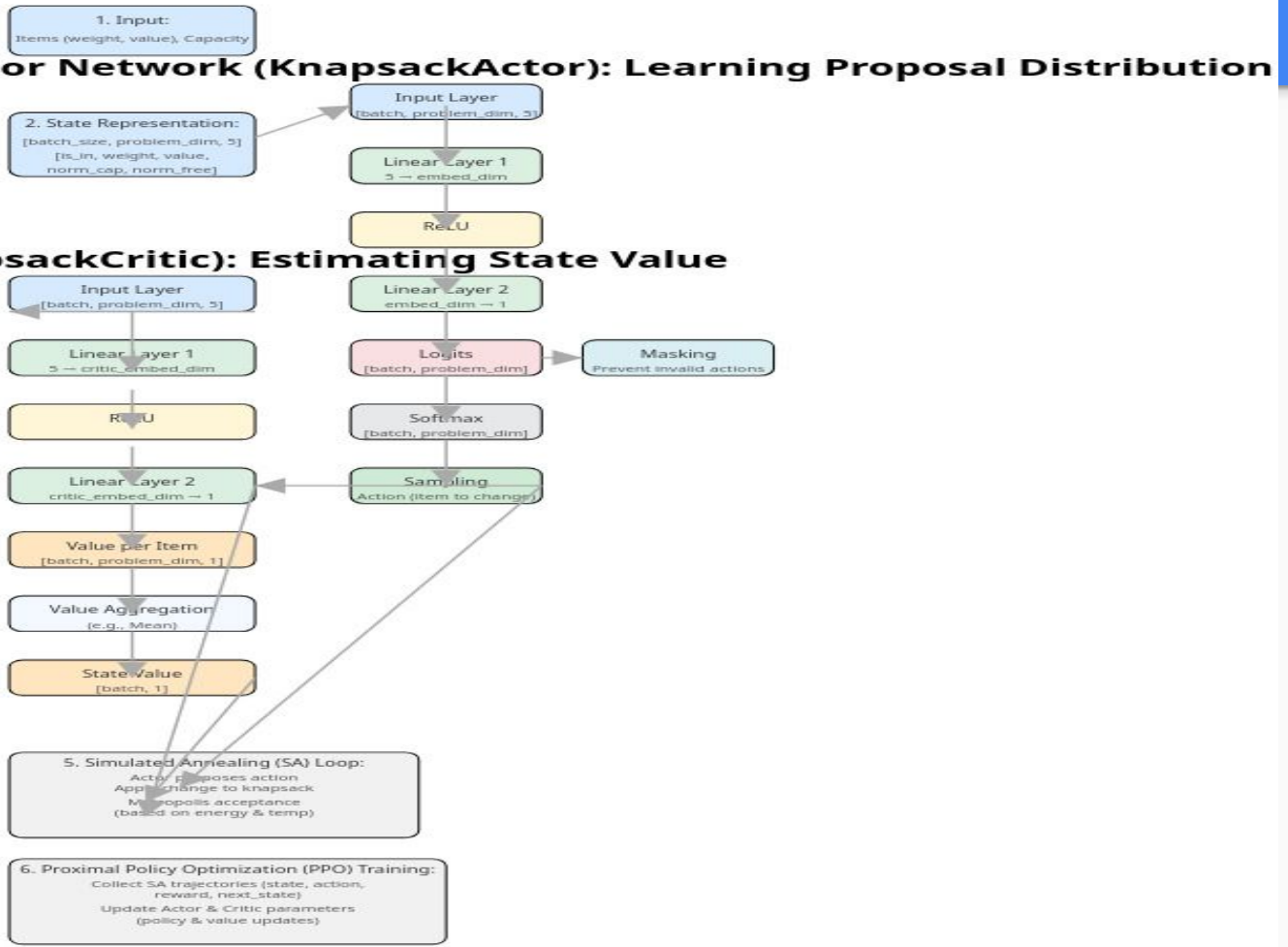
- **High-Level Summary:** We employ a neural network-based actor-critic architecture, trained with Proximal Policy Optimization (PPO), to intelligently select and evaluate item combinations for the knapsack problem.
- **Key Components:**
    - **Actor Network:** Learns a policy to propose which items to add or remove.
    - **Critic Network:** Estimates the value of the current knapsack state (items selected so far).
    - **PPO Training:** Optimizes the actor and critic to make better decisions over time.
- **Simulated Annealing Integration:** In the architecture diagram loop section this actor will propse and action and will be accepted and rejcted depend on state.



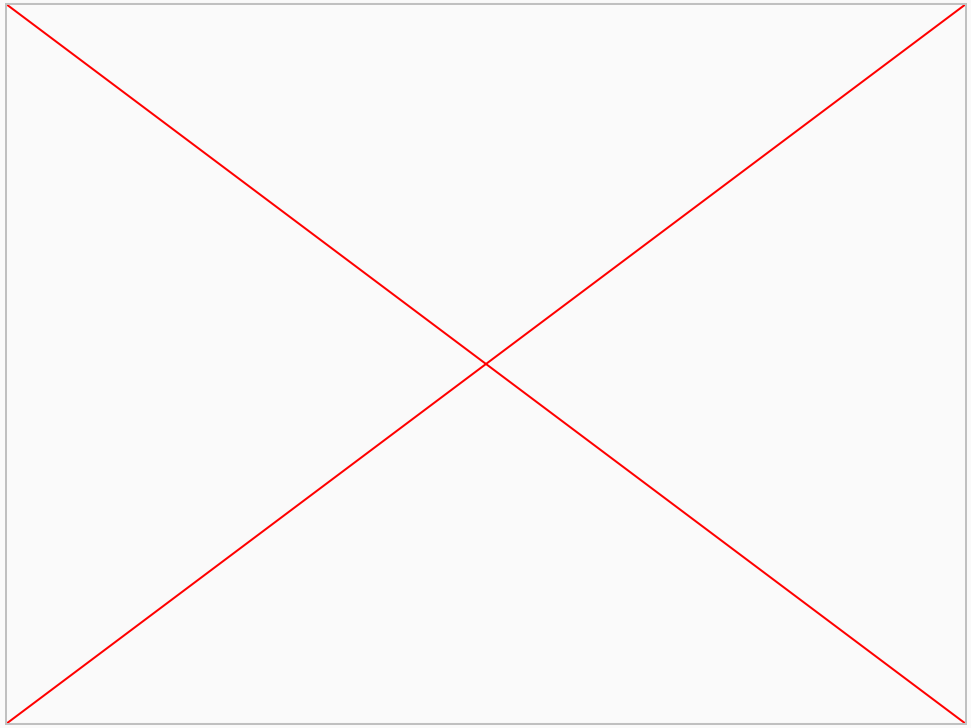
PPO Agent Architecture






1. Input:
Items (weight, value), Capacity
3. Actor Network (KnapsackActor): Learning Proposal Distribution
2. State Representation:
[batch_size, problem_dim, 5]
[is_in, weight, value,
norm_cap, norm_free]
Input Layer
[batch, problem_dim, 5]
Linear Layer 1
5 → embed_dim
ReLU
4. Critic Network (KnapsackCritic): Estimating State Value
Input Layer
[batch, problem_dim, 5]
Linear Layer 1
5 → critic_embed_dim
ReLU
Linear Layer 2
critic_embed_dim → 1
Value per Item
[batch, problem_dim, 1]
Value Aggregation
(e.g., Mean)
State Value
[batch, 1]
Linear Layer 2
embed_dim → 1
Logits
[batch, problem_dim]
Masking
Prevent invalid actions
Softmax
[batch, problem_dim]
Sampling
Action (item to change)
5. Simulated Annealing (SA) Loop:
Actor proposes action
Apply change to knapsack
Metropolis acceptance
(based on energy & temp)
6. Proximal Policy Optimization (PPO) Training:
Collect SA trajectories (state, action,
reward, next_state)
Update Actor & Critic parameters
(policy & value updates)

Stochastic Gradient Descent (and importance of momentum)



Vanilla Gradient Descent

$$W_{t+1} = W_t - \alpha \nabla W_t$$

Gradient Descent with Momentum

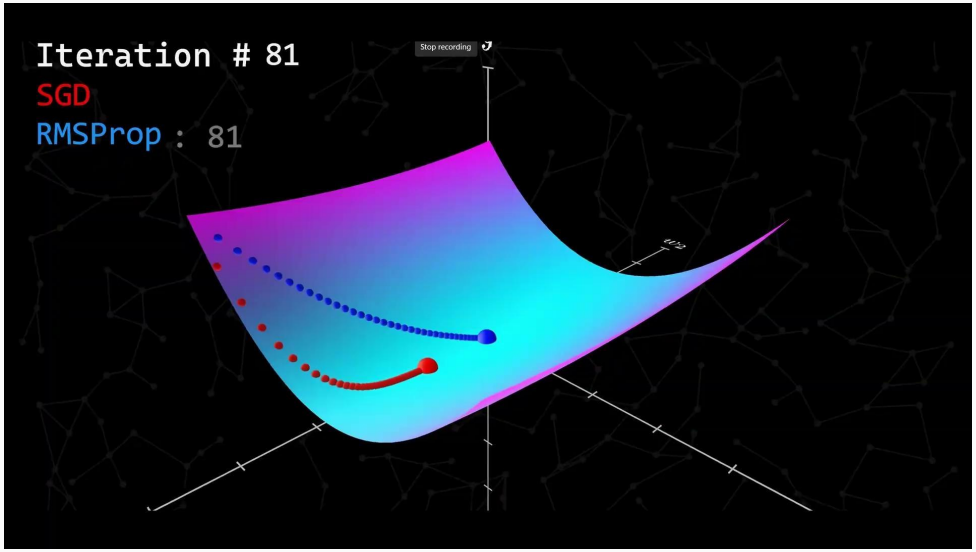
$$V_{t+1} = \beta V_t + (1 - \beta) \nabla W_t$$

$$W_{t+1} = W_t - \alpha V_{t+1}$$

$$\beta = 0.9$$

$$V_{t+1} = \beta^3 V_{t-2} + \beta^2 (1 - \beta) \nabla W_{t-2} + \beta (1 - \beta) \nabla W_{t-1} + (1 - \beta) \nabla W_t$$

RMS Prop



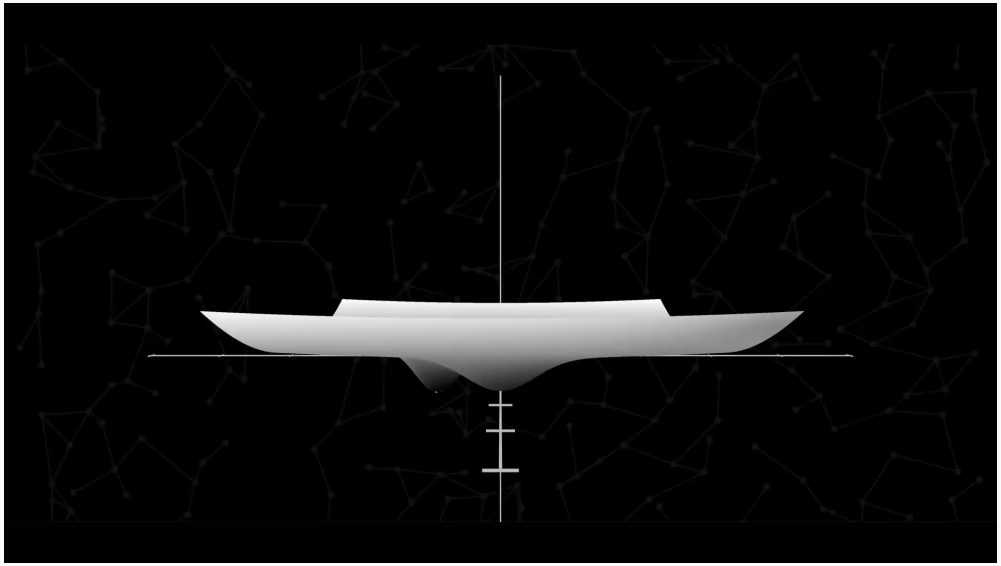
Root Mean Squared Propagation

$$V_{t+1} = \beta V_t + (1-\beta)\nabla W_t^2$$

$$W_{t+1} = W_t - \alpha \frac{\nabla W_t}{\sqrt{V_{t+1} + \epsilon}}$$

$$\beta = 0.9$$

ADAM



Adaptive Moment Estimation

$$M_t = \beta_1 M_{t-1} + (1-\beta_1)\nabla W_t$$

$$V_t = \beta_2 V_{t-1} + (1-\beta_2)\nabla W_t^2$$

$$\hat{M}_t = \frac{M_t}{1-\beta_1^t}$$

$$\hat{V}_t = \frac{V_t}{1-\beta_2^t}$$

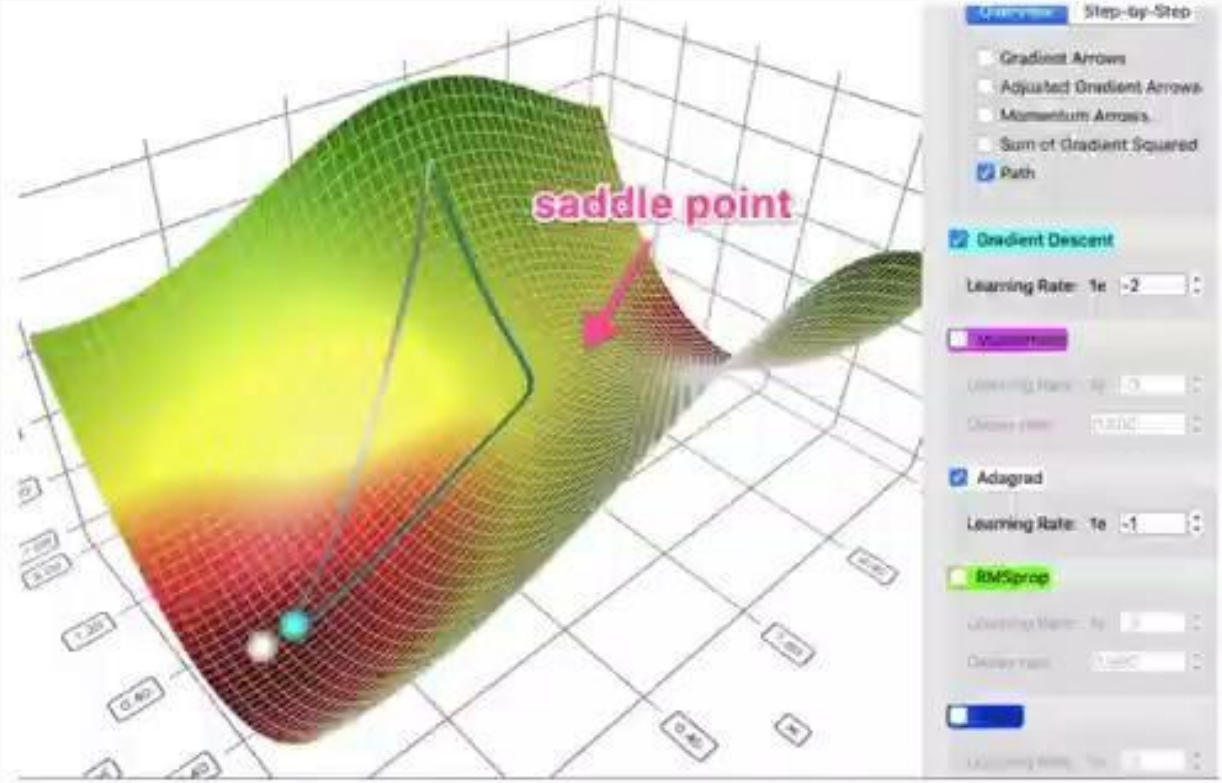
$$W_{t+1} = W_t - \alpha \frac{\hat{M}_t}{\sqrt{\hat{V}_t} + \epsilon}$$

$\beta_1 = 0.9$

$\beta_2 = 0.99$

$\epsilon = 10^{-8}$

Adagrad



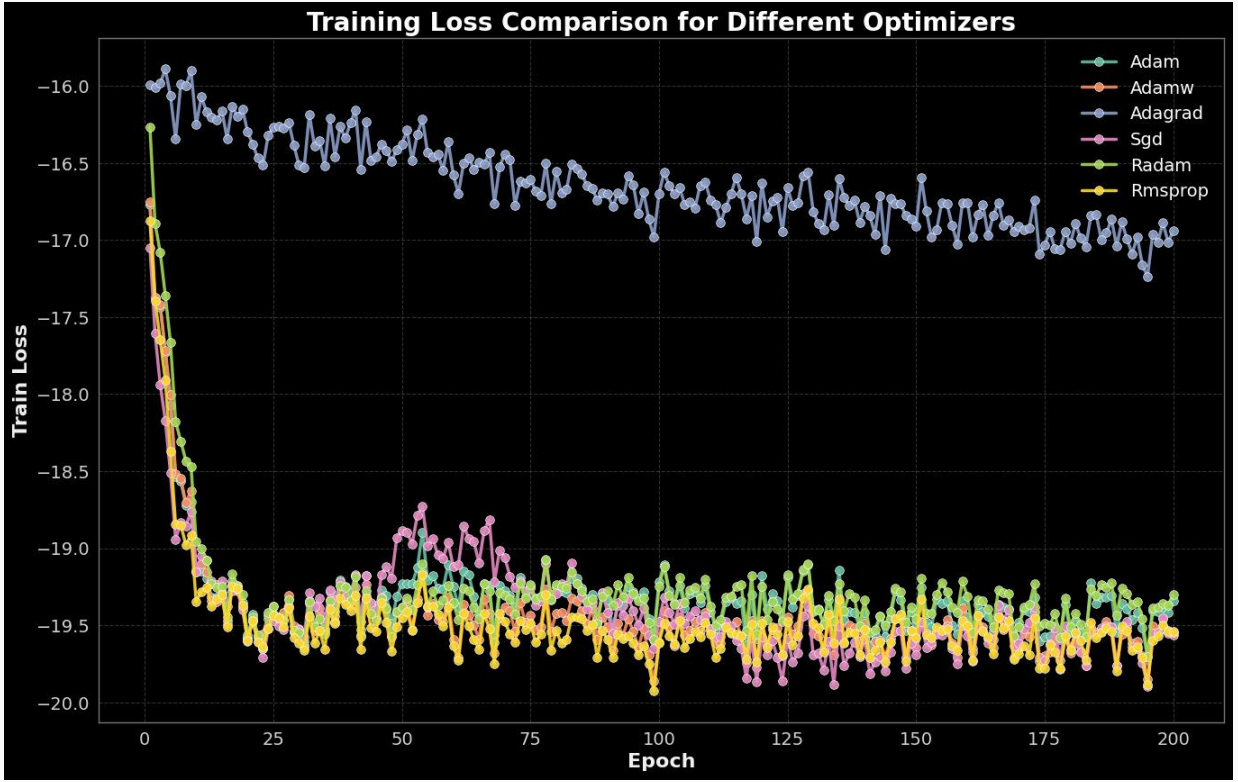
AdaGrad (white) vs. gradient descent (cyan) on a terrain with a saddle point. The learning rate of AdaGrad is set to be higher than that of gradient descent, but the point that AdaGrad's path is straighter stays largely true regardless of learning rate.

Adaptive Gradient

$$V_t = V_{t-1} + \nabla W_t^2$$

$$W_{t+1} = W_t - \alpha \frac{\nabla W_t}{\sqrt{V_t + \epsilon}}$$

Result graph-PPO



Our interpretation of the results:

Main observations:

1. Adagrad performs worst: Its loss plateaus early; poor convergence and learning stagnation, likely due to aggressive learning rate decay
2. SGD shows slightly more variance but ends up with very low loss.
3. RMSprop and AdamW show smoother curves, suggesting consistent learning.
4. Adam and Radam are also strong contenders with steady convergence

Optimizer	ADAM	ADAMW	AdaGrad	Radam	RMSProp	SGD
Final Loss	-19.3397	-19.5645	-16.9414	-19.301	-19.5419	-19.549
Min Loss	-19.7072	-19.863	-17.2371	-19.702	-19.9275	-19.895
Avg Loss	-19.3062	-19.4363	-16.639	-19.2773	-19.4839	-19.384
Std Dev	0.344827	0.37247	0.28294	0.410025	0.342591	0.360655

Optimizer	Training Loss at Epoch = 200	Time taken for 200 epochs
Adam	-19.33971786	8.06
AdamW	-19.56446648	7.53
Adagrad	-16.94142914	7.4
RMSprop	-19.54190063	7.47
SGD	-19.54896164	8.11
Radam	-19.30096054	8.01

Evolution strategies

Evolutionary Strategies (ES) are a class of optimization algorithms inspired by the principles of natural selection.

The below is the process flow for ES:

1. **Sampling:** The controller generates a batch of candidate solutions s by sampling from a search distribution.
2. **Evaluation:** Each solution s is evaluated using the fitness function, which returns a scalar value representing the quality of the solution.
3. **Selection:** Solutions with lower cost (i.e., higher value and feasible weight) are favored for selection. These solutions contribute to the formation of the next generation.
4. **Update:** The policy (controller) is updated using policy gradient methods to increase the probability of generating better solutions in subsequent iterations.

This process is repeated over multiple iterations, progressively improving the quality of solutions generated by the controller.

Problem Parameters:

- *n_problems*: 256
- *problem_dim*: 50
- *embed_dim*: 16
- Capacity: 12.5

Training Parameters:

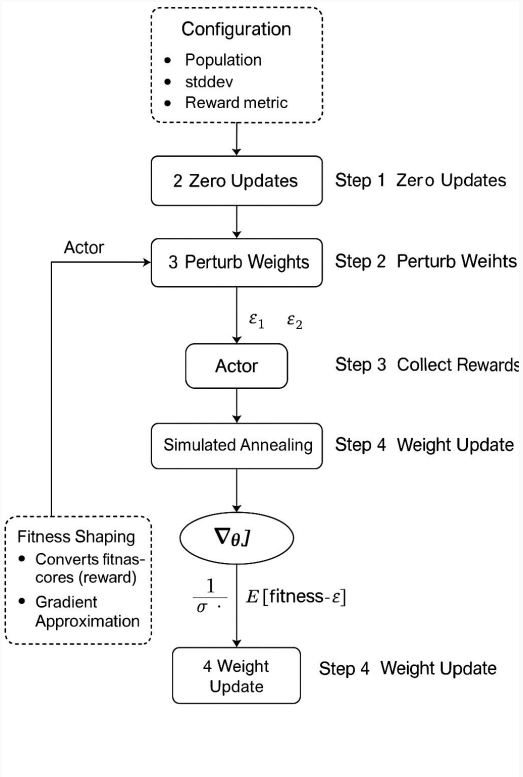
- Method: ES (Evolution Strategies)
- Reward: min_cost
- *n_epochs*: 200
- Learning rate (*lr*): 0.001
- Batch size: 500
- Weight decay: 0.01
- Momentum: 0.9
- Standard deviation (*stddev*): 0.05
- Population size: 16
- Milestones: [0.9]
- Optimizer: SGD (Stochastic Gradient Descent), Adam, AdamW, RMSProp, Adagrad

Evolution strategies

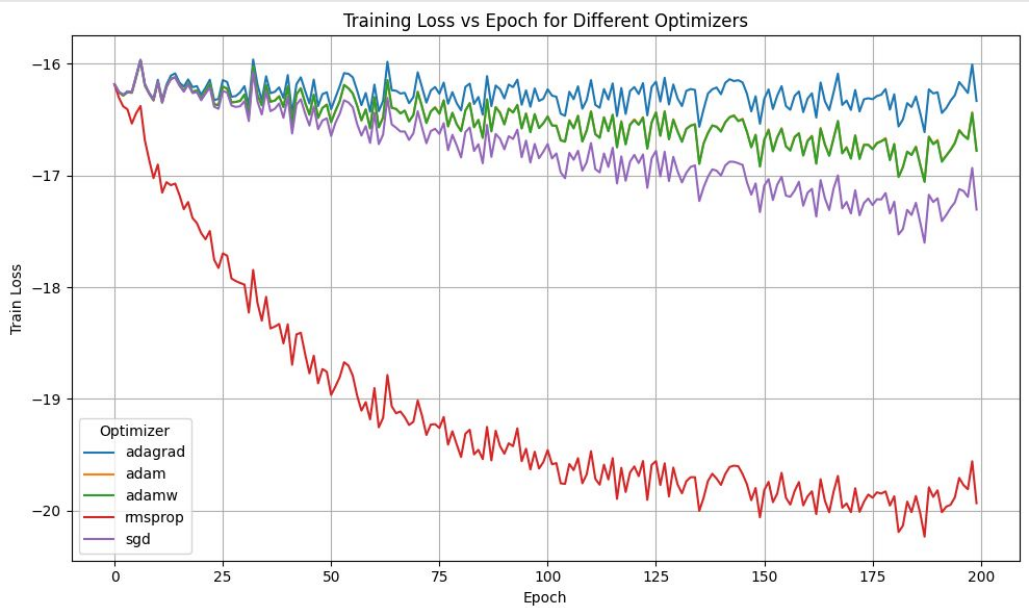
Evolutionary Strategies (ES) are a class of optimization algorithms inspired by the principles of natural selection

```
def cost(self, s: torch.Tensor) -> torch.Tensor:
    v = torch.sum(self.values * s[..., 0], -1)
    w = torch.sum(self.weights * s[..., 0], -1)
    return -v * (w < self.capacity[..., 0])
```

- **s**: A binary tensor representing the selection of items in the Knapsack. The tensor values are either 0 or 1, indicating whether an item is included in the solution.
- **v**: The total value of the selected items, computed as the sum of the values of all items where the corresponding selection variable is 1.
- **w**: The total weight of the selected items, computed similarly as the sum of the weights of selected items.
- **(w < capacity)**: A Boolean mask that ensures the total weight does not exceed the knapsack's capacity. If the total weight exceeds the capacity, the solution is considered infeasible.



Evolution strategies result and interpretation



Optimizer	Training Loss at Epochs = 200	Time Taken for 200 epochs
Adam	-16.7765	16m 53s
AdamW	-16.7800	16m 21s
Adagrad	-16.3340	16m 49s
RMSprop	-19.9331	17m 25s
SGD	-17.3060	17m 11s

Evolution strategies result and interpretation

Table 3.3: Evaluation Results using Adam Optimizer

Instances	Random Seed	Sampled Value
1x, K=50	1–5	-14.86, -14.79, -14.96, -14.86, -14.86
2x, K=100	1–5	-16.71, -16.77, -16.86, -16.75, -16.69
5x, K=250	1–5	-18.18, -18.21, -18.22, -18.16, -18.17
10x, K=500	1–5	-18.77, -18.76, -18.79, -18.76, -18.73

Table 3.4: Evaluation Results using AdamW Optimizer

Instances	Random Seed	Sampled Value
1x, K=50	1–5	-14.87, -14.78, -14.96, -14.84, -14.86
2x, K=100	1–5	-16.73, -16.75, -16.84, -16.75, -16.69
5x, K=250	1–5	-18.18, -18.22, -18.22, -18.16, -18.16
10x, K=500	1–5	-18.79, -18.76, -18.80, -18.76, -18.74

Table 3.5: Evaluation Results using Adagrad Optimizer

Instances	Random Seed	Sampled Value
1x, K=50	1–5	-14.30, -14.22, -14.49, -14.25, -14.36
2x, K=100	1–5	-16.34, -16.18, -16.54, -16.33, -16.23
5x, K=250	1–5	-17.97, -17.99, -18.01, -17.93, -17.95
10x, K=500	1–5	-18.61, -18.61, -18.61, -18.59, -18.61

Evolution strategies Results

Table 3.6: Evaluation Results using RMSprop Optimizer

Instances	Random Seed	Sampled Value
1x, K=50	1–5	-19.65, -19.63, -19.64, -19.64, -19.69
2x, K=100	1–5	-19.88, -19.88, -19.88, -19.89, -19.87
5x, K=250	1–5	-19.99, -19.99, -19.99, -20.00, -19.99
10x, K=500	1–5	-20.05, -20.04, -20.04, -20.03, -20.04

Table 3.7: Evaluation Results using SGD Optimizer

Instances	Random Seed	Sampled Value
1x, K=50	1–5	-15.63, -15.52, -15.61, -15.58, -15.58
2x, K=100	1–5	-17.31, -17.21, -17.33, -17.26, -17.30
5x, K=250	1–5	-18.48, -18.46, -18.52, -18.44, -18.51
10x, K=500	1–5	-18.92, -18.93, -19.01, -18.91, -18.99

Among all tested optimizers, RMSprop achieved the best training performance with a loss of -19.9331, albeit with the highest training time of 17 minutes 25 seconds. SGD and AdamW followed with losses of -17.3060 and -16.7800, respectively. While AdamW was the fastest to converge (16 minutes 21 seconds), its final loss was approximately 3.15 units higher than RMSprop, indicating a trade-off between training time and convergence quality.

Conclusion: RMSprop demonstrated the best overall optimizer performance for this ES-based training setup, offering the lowest training loss at the expense of slightly increased computational time.

Conclusions

Main Idea: Augment classic simulated annealing (SA) with learnable proposal distributions using neural networks.

- **Advantages**:
  - Faster convergence.
  - Adaptivity via side-information.
  - No need for ground-truth labels.
  - Lightweight architectures (CPU-friendly).
  - Scales to large problem sizes.
  - Generalises across problem sizes and rollout lengths.
- **Limitations**:
  - No built-in stopping criterion or solution quality certification.
  - Fixed temperature schedule; tuning not explored.

References and Future work

References:

- Qualcomm-AI-research/neural-simulated-annealing at 76056e019c7362917a07c0c28597533501789b2b
- Visually Explained: Newton's Method in Optimization

- **Future Directions**:
  - Extend to multi-trajectory methods (parallel tempering, genetic algorithms).
  - Exchange information across a population of solutions.

Thank you !