Project Report

Implementation of Neural Simulated Annealing

DSE/ECS 311: Applied Optimisation

Submitted by:
Ishanya (21329) ishanya21@iiserb.ac.in
HariKrishna (22236) peddinti22@iiserb.ac.in
Hiba KT (22146) hiba22@iiserb.ac.in
Astha (22063) astha22@iiserb.ac.in

Project Repository: https://github.com/ishanyaa/SAOptimisation

Indian Institute of Science Education and Research (IISER)
Bhopal

24th April 2025

Contents

Abstract

This report provides a preliminary overview of our optimization project, which aims to implement the Neural Simulated Annealing (Neural SA) technique as described in the paper "Neural Simulated Annealing" by Alvaro H.C. Correia, Daniel E. Worrall, and Roberto Bondesan. We outline the core concepts of the paper, our progress in implementing the algorithm and comparing performances of different optimizers for solving the Knapsack problem using Neural SA, and our future steps in evaluating its performance on combinatorial optimization problems.

Chapter 1

Introduction

Optimization plays a crucial role in solving complex combinatorial problems. Simulated Annealing (SA) is a well-established metaheuristic used to approximate global optima for such problems. The paper [1] introduces a novel approach called Neural Simulated Annealing (Neural SA), which leverages reinforcement learning to optimize the proposal distribution in SA, leading to faster convergence and improved solution quality.

1.1 Motivation

Optimization is crucial for solving real-world problems like route planning, resource allocation, and scheduling. Traditional Simulated Annealing (SA) is effective but relies on fixed, hand-tuned rules that can limit its performance. Neural Simulated Annealing (Neural SA) improves upon this by using a small neural network to learn and adapt its move proposals. This project is motivated by the need to: (1) enhance optimization efficiency; (2) reduce manual parameter tuning in SA; and (3) leverage machine learning for smarter, faster solutions in complex combinatorial problems.

1.2 Work done in the original paper:

The original work focuses on implementing Neural SA to improve the efficiency of combinatorial optimization tasks such as the Knapsack problem, Bin Packing problem, and the Traveling Salesperson problem (TSP). Neural

SA utilizes a policy network to learn an optimal proposal distribution, which enhances the search process in SA by balancing exploration and exploitation.

1.3 What is our project about?

Our project focuses on implementing Neural SA using Proximal Policy Optimization (PPO) and Evolution Strategies (ES). We have experimented with multiple optimizers: ADAM, ADAMW, AdaGrad, RMSProp, SGD with momentum (for both PPO and ES), and RAdam (for PPO). We have compared their performances based on the training loss curves, the time taken to train, and sampled cost per sampling rate K.

1.4 Some combinatorial problems Neural SA can optimize

- **Travelling Salesman Problem:** Given cities $i \in \{0, 1, \ldots, N-1\}$ with spatial coordinates $\mathbf{c}_i \in [0,1]^2$, we wish to find a linear ordering of the cities, a *tour*, denoted by the permutation vector $\mathbf{x} = (x_0, x_1, \ldots, x_{N-1})$ for $x_i \in \{0, 1, \ldots, N-1\}$ such that

$$\min \sum_{i=0}^{N-1} \|\mathbf{c}_{x_{i+1}} - \mathbf{c}_{x_i}\|^2$$

subject to:

$$x_i \neq x_j \quad \text{for all } i \neq j \quad \text{and} \quad x_i \in \{0, 1, \ldots, N-1\}$$

where we have defined $x_N = x_0$ for convenience of notation.

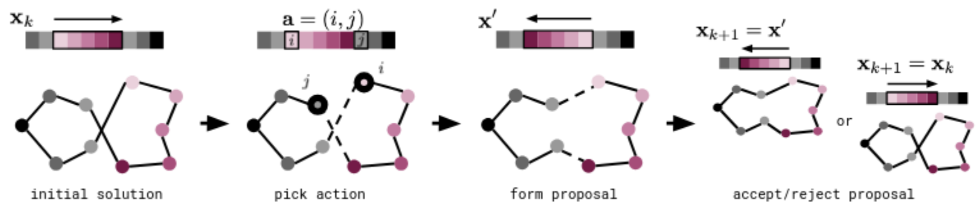

Figure 1.1: This figure illustrates how Neural SA works for the Traveling Salesperson Problem (TSP).(Image taken from [1])

- **Knapsack Problem:** Given a set of N items, each of different value $v_i > 0$ and weight $w_i > 0$, the goal is to find a subset that maximizes the sum of values while respecting a maximum total weight of W. This is the 0-1 Knapsack Problem, which is weakly NP-complete, has a search space of size 2^N, and its Integer Linear Programming formulation is:

  $$\min -\sum_{i=0}^{N-1} v_i x_i,$$

  subject to the constraint:

  $$\sum_{i=0}^{N-1} w_i x_i \leq W,$$

  where solutions are represented as a binary vector $\mathbf{x}$, with $x_i = 0$ for 'out of the bin' and $x_i = 1$ for 'in the bin'.

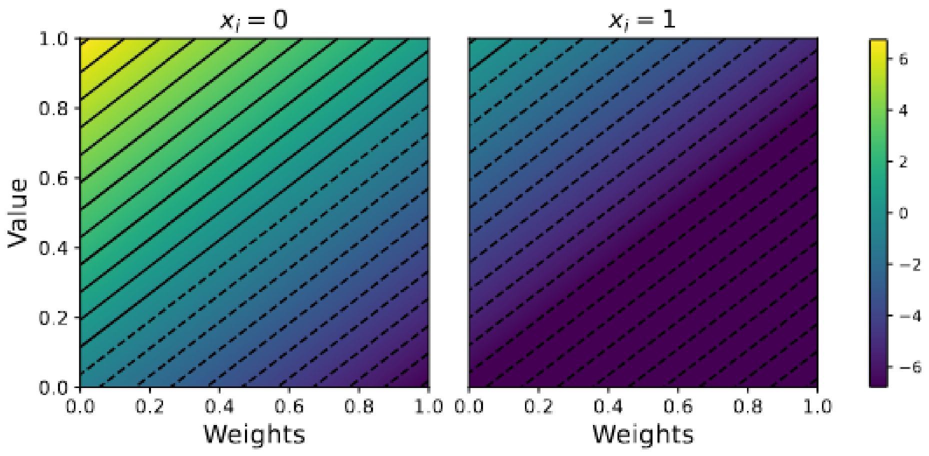

Figure 1.2: This figure visualizes how Neural SA selects items for the Knapsack Problem based on their value-to-weight ratio.(Image taken from [1])

- **Bin Packing Problem:** Given N items, pack *all* N items into the smallest possible number of bins, where each item $i \in \{1, \ldots, N\}$ has the weight w_i and we assume without loss of generality, N bins of equal capacity $W \geq \max_i(w_i)$. The objective is to:

$$\min \sum_{j=0}^{N-1} y_j$$

subject to the following constraints:

$$\text{Bin capacity constraint: } \sum_{i=0}^{N-1} w_i x_{ij} \leq W$$

$$\text{One bin per item: } \sum_{j=0}^{N-1} x_{ij} = 1$$

$$\text{Bin occupancy indicator: } y_j = \min\left(1, \sum_{i=0}^{N-1} x_{ij}\right)$$

where $x_{ij} \in \{0, 1\}$ denotes item i occupying bin j, and the constraints apply for all i and j.

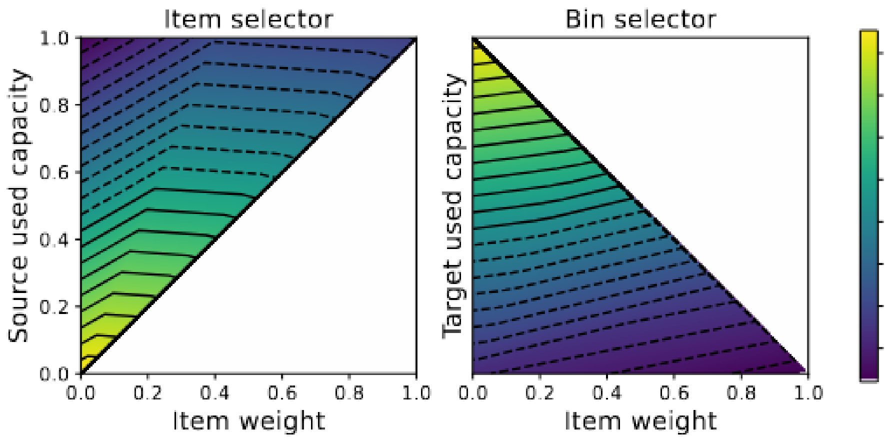

Figure 1.3: This figure explains how Neural SA selects items and bins for the Bin Packing Problem, showing the learned policy's behavior.(Image taken from [1])

Chapter 2

Implepentation and Methodology

2.1 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) is a policy gradient method used in reinforcement learning to optimize stochastic policies in a stable and efficient manner. It improves upon traditional policy gradient techniques by introducing a clipped surrogate objective function that limits the deviation of the updated policy from the old one, thus maintaining a trust region.

The clipped objective is defined as:

$$L^{\mathrm{CLIP}}(\theta) = E_t \left[\min \left(r_t(\theta)\hat{A}_t,\ \mathrm{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right],$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\mathrm{old}}}(a_t|s_t)}$ is the probability ratio between the new and old policies, $\hat{A}_t$ is the estimated advantage function, and ϵ is a hyperparameter that controls the amount of clipping.

This approach prevents excessively large policy updates and encourages reliable convergence, making PPO suitable for a wide range of control and optimization tasks.

Actor-Critic Architecture in Neural Simulated Annealing

Policy Network (Actor):

- In reinforcement learning, the policy is the agent's strategy for selecting actions based on the current state.

- The actor network in Neural Simulated Annealing (Neural SA) learns a *proposal distribution*, which maps the current solution (state) to a probability distribution over possible modifications (actions).
- For example, in the Traveling Salesman Problem (TSP), the actor observes the current tour and proposes edge swaps that are likely to improve the solution.

Value Network (Critic):

- The value function estimates the expected cumulative reward from a given state, essentially measuring the state's quality.
- The critic network approximates this value function and outputs a scalar value representing how good the current solution is.
- This serves as a *baseline* to guide the actor: if an action results in a higher value (better state), the actor is positively reinforced.

Together, the actor and critic work in tandem: the actor proposes new solutions, and the critic evaluates them, enabling the system to iteratively improve its search strategy within the SA framework.

2.2 Evolution Strategies Optimization

Evolutionary Strategies (ES) are a class of black-box optimization algorithms inspired by the principles of natural selection. These algorithms are particularly effective in discrete and combinatorial domains where gradient-based methods fail to perform well. ES leverage stochastic search techniques to explore the solution space and are applied successfully to a wide range of optimization problems, including the Knapsack problem.

Evolutionary Strategies operate by iteratively sampling candidate solutions from a stochastic distribution, evaluating their performance using a fitness function, and then updating the distribution based on performance. This iterative process aims to improve the quality of candidate solutions over time.

Key Components of Evolutionary Strategies

- **Search Distribution:** The distribution from which candidate solutions are sampled, often parameterized by a neural network that evolves through training.
- **Fitness Function:** A function that evaluates candidate solutions and guides the search process. It plays a critical role in the selection of better solutions.
- **Policy Update:** The process of adjusting the search distribution based on feedback from the evaluation of candidate solutions, typically performed using policy gradients or similar techniques.

2.2.1 Fitness Evaluation in Knapsack

The fitness function used in the ES optimization process for the Knapsack problem is designed to evaluate how well a candidate solution satisfies the problem constraints and maximizes the total value of selected items.

The fitness function is defined as:

```
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])
```

Explanation of the Fitness Function

- **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.

2.2.2 Behavior of the Fitness Function

- If the solution is feasible (i.e., the total weight w is less than or equal to the capacity), the fitness is the negative total value $-v$. This negative value is used because Evolutionary Strategies aim to minimize the cost function, and minimizing the negative value corresponds to maximizing the total value. - If the solution is infeasible (i.e., the total weight exceeds the capacity), the fitness function returns 0, representing a poor solution that does not meet the constraints.

Thus, the fitness function penalizes infeasible solutions and encourages the exploration of high-value feasible solutions.

2.2.3 Evolutionary Strategy Process for Knapsack

The process of solving the Knapsack problem using Evolutionary Strategies follows an iterative sequence of sampling, evaluation, selection, and update.

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.

2.3 Implementation

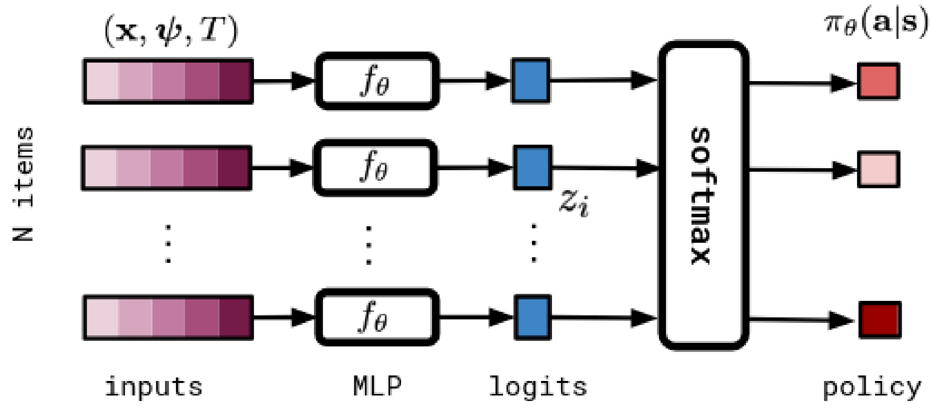
$(\mathbf{x}, \boldsymbol{\psi}, T)$
$\pi_\theta(\mathbf{a}|\mathbf{s})$
N items
f_θ
f_θ
f_θ
z_i
softmax
inputs
MLP
logits
policy

Figure 2.1: This figure shows the architecture of the lightweight policy network used in Neural SA, highlighting the embedding and decision-making process.(Image taken from [1])

2.3.1 Training Pipeline: `scripts/main.py`

The `scripts/main.py` file orchestrates the complete training pipeline for Neural Simulated Annealing (NSA). It utilizes the Hydra configuration system for experiment management, initializes models, configures problems, and launches training using either Evolution Strategies (ES) or Proximal Policy Optimization (PPO).

Configuration and Setup

- Hydra is used to manage experiment configurations, with settings defined in the `conf/config.yaml` file.
- The device is set to GPU or CPU depending on availability.
- Seeds are fixed across PyTorch, NumPy, and Python's random module for reproducibility.
- The temperature decay parameter α for Simulated Annealing (SA) is computed based on the initial and stopping temperatures and total outer steps.

Problem and Model Initialization

Depending on the specified problem in the config file, the script dynamically initializes the problem and its corresponding actor and critic networks:

- **Knapsack:** Uses `KnapsackActor` and `KnapsackCritic`.
- **BinPacking:** Uses `BinPackingActor` and `BinPackingCritic`.
- **TSP:** Uses `TSPActor` and `TSPCritic`.

The problem is seeded and initialized with a given number of instances and parameters.

Training Algorithm Selection

The training method is selected based on the configuration:

- **PPO:** Initializes Adam optimizers for actor and critic networks. Trains the policy using data collected by SA and updates it via policy gradients.
- **ES:** Initializes an `EvolutionStrategies` optimizer (SGD with momentum), and a learning rate scheduler with predefined milestones.

Training Loop

The training loop runs for a specified number of epochs:

1. Random problem instances are generated and transferred to the target device.
2. Initial solutions are sampled for SA.
3. The selected training algorithm (PPO or ES) is used to update the model:
    - `train_ppo()`: Collects transitions during SA and updates policy via PPO.
    - `train_es()`: Samples perturbations, evaluates losses using SA, and performs gradient-free updates.

4. After training, the model is re-evaluated using SA to obtain the final training loss.
5. The actor network is saved to disk under a folder named `models/`.

Function: `train_es`

This function performs one full ES update:

- Zeroes out previous updates.
- Generates antithetic perturbations and evaluates the policy via SA.
- Collects the objective (loss) values.
- Applies a single gradient-free update based on aggregated fitness scores.

Function: `train_ppo`

This function implements policy learning with PPO:

- SA is run with a replay buffer to collect transitions.
- The PPO algorithm optimizes the actor and critic based on these transitions.

Key Advantages

- **Modularity:** Supports multiple problems and training methods with minimal changes.
- **Reproducibility:** Ensures consistent results across runs through seed management.
- **Extensibility:** New problems and training strategies can be easily added.

This script serves as the central entry point to launch experiments, encapsulating the NSA pipeline from setup to model saving.

2.3.2 Implementation details of Evolution Strategies

In this section, we discuss the practical implementation of Evolutionary Strategies for the Knapsack problem within the Neural Simulated Annealing framework.

The primary objective is to optimize the selection of items in the Knapsack using ES, with the following key components:

Fitness Function

The fitness function is implemented in Python as shown below:

```
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])
```

This function computes the total value of selected items and penalizes infeasible solutions by returning a fitness of 0 if the total weight exceeds the knapsack's capacity. If the solution is feasible, the negative of the total value is returned to guide the optimization process.

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

- Trace decay: 0.9
- Epsilon clip (ϵ_clip): 0.25
- Discount factor (γ): 0.9
- 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

Simulated Annealing (SA) Parameters:

- Initial temperature ($init\_temp$): 1.0
- Stop temperature ($stop\_temp$): 0.1
- Outer steps: 100
- Inner steps: 1
- Alpha: 0.9772372209558107

Problem:

- Type: Knapsack
- Capacity: 12.5
- Device: CPU

2.3.3 Implementation details of Proximal Policy optimization

Problem Parameters:

- $n\_problems$: 256
- $problem\_dim$: 50
- $embed\_dim$: 16
- Problem: Knapsack

Training Parameters:

- Method: ES (Evolution Strategies)
- Reward: min_cost
- n_epochs: 200
- Learning rate (lr): 0.001
- Batch size: 500
- Trace decay: 0.9
- Epsilon clip (ϵ_clip): 0.25
- Discount factor (γ): 0.9
- Weight decay: 0.01
- Momentum: 0.9
- Standard deviation ($stddev$): 0.05
- Population size: 16
- Milestones: [0.9]
- PPO epochs: 10
- Optimizer: SGD (Stochastic Gradient Descent with momentum), Adam, AdamW, RM- SProp, Adagrad, RAdam

Simulated Annealing (SA) Parameters:

- Initial temperature ($init\_temp$): 1.0
- Stop temperature ($stop\_temp$): 0.1
- Outer steps: 100
- Inner steps: 1
- Alpha: 0.9772372209558108

Chapter 3

Results

3.1 Results for PPO

Optimizer	**Training Loss at Epochs = 200**	**Time Taken for 200 epochs**
Adam	-19.3397	08m 06s
AdamW	-19.5645	07m 53s
Adagrad	-16.9414	07m 40s
RMSprop	-19.5419	07m 47s
SGD	-19.5489	08m 11s
RAdam	-19.3009	08m 01s

Table 3.1: Training Loss and Time Taken for 200 Epochs with Different Optimizers in PPO

The table summarizes the performance of various optimizers when used with Evolution Strategies (ES) over 200 training epochs. The training loss values represent the objective being minimized (cost), where lower (more negative) values indicate better performance (higher reward)

Main observations:

- Adagrad performs worst: Its loss plateaus early; poor convergence and learning stagnation, likely due to aggressive learning rate decay
- SGD shows slightly more variance but ends up with very low loss.
- RMSprop and AdamW show smoother curves, suggesting consistent learning.

- Adam and Radam are also strong contenders with steady convergence

Here, ADAMW acheived the lowest loss value of -19.5645 with a decently good training time of 7 minutes and 53 seconds. AdaGrad takes the least training time, however it performs the worst among these optimizers.

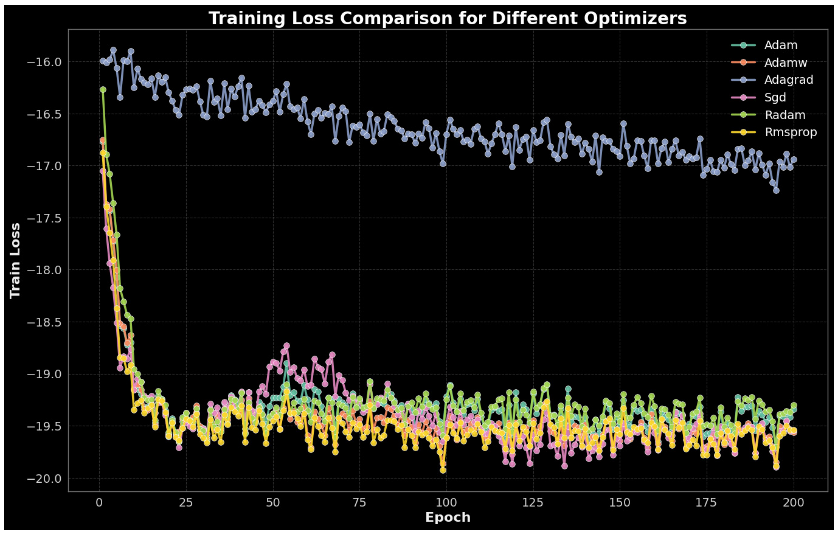

Figure 3.1: Comparision of different optimisers with PPO

3.2 Results for Evolution Strategies

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

Table 3.2: Training Loss and Time Taken for 200 Epochs with Different Optimizers for ES

The table summarizes the performance of various optimizers when used with Evolution Strategies (ES) over 200 training epochs. The training loss values represent the objective being minimized (cost), where lower (more negative) values indicate better performance (higher reward).

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.

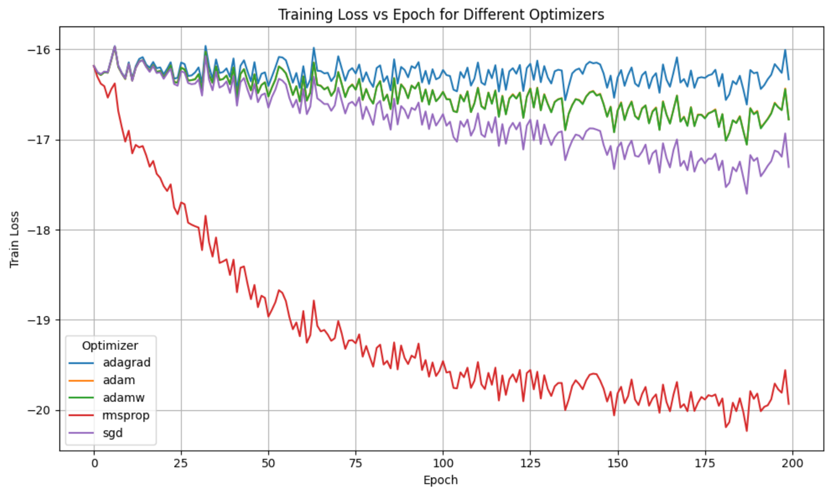

Figure 3.2: Comparision of different optimisers with ES

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

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

Discussions regarding the Evaluation results

The table presents performance logs from evaluating the model on multiple instances of the Knapsack problem under different configurations. Here, K represents the number of Simulated Annealing (SA) iterations. Given that the problem size is fixed at `problem_dim` $= 50$, the number of SA steps is computed as $K = m \times$ `problem_dim`, where $m \in \{1, 2, 5, 10\}$ is a predefined multiplier.

This setup ensures that the number of optimization steps (i.e., SA iterations) scales proportionally with the problem size, allowing the Neural SA algorithm a fair and consistent amount of search effort across different configurations.

Chapter 4

Conclusion and Future Works

This work presents a comparative study of various gradient-based optimizers applied within the Evolution Strategies (ES) framework for solving combinatorial optimization problems, specifically the Knapsack problem. The experiments were conducted under consistent conditions—fixed problem dimensionality, batch size, and training epochs—to evaluate optimizer impact on performance and efficiency.

The results indicate that among the tested optimizers (Adam, AdamW, Adagrad, RMSprop, and SGD), **RMSprop** achieved the best final training loss (`-19.9331`), outperforming others in terms of solution quality. However, it also required the longest training time (`17 minutes 25 seconds`). In contrast, **AdamW** demonstrated the fastest convergence (`16 minutes 21 seconds`) with competitive loss performance (`-16.7800`). These results highlight the trade-off between convergence quality and training time when selecting optimizers for ES-based training.

Inference:

- **RMSprop** is the most effective optimizer in terms of solution quality, especially suitable for scenarios where optimal performance is critical regardless of time cost.

- **AdamW** offers the best trade-off between convergence quality and computational efficiency.

- **Adagrad** underperforms across all metrics and is not recommended for ES-based training in discrete combinatorial settings.

- **SGD** provides decent convergence but may require further tuning to match adaptive optimizers.

In conclusion, Evolutionary Strategies, as implemented in the Neural Simulated Annealing framework, provide a powerful tool for solving combinatorial optimization problems like the Knapsack problem. By utilizing a fitness function that penalizes infeasible solutions and encourages high-value selections, ES iteratively improves solution quality without the need for gradient information. This approach is well-suited to discrete optimization problems and can be further extended to other combinatorial domains.

Future Works

Several avenues remain for further research:

- **Generalization to other problems:** Extend the evaluation to other combinatorial problems such as the Traveling Salesman Problem (TSP) and Capacitated Vehicle Routing Problem (CVRP) to validate optimizer robustness across tasks.

- **Hybrid optimization techniques:** Investigate combining ES with policy gradient methods or surrogate modeling to improve sample efficiency and convergence.

- **Meta-optimization:** Explore adaptive or learned optimizer configurations through meta-learning to automatically tune optimizer hyperparameters for different problem classes.

- **GPU acceleration:** Re-evaluate optimizer performance in a CUDA-enabled environment to determine improvements in training time and scalability.

- **Real-world applications:** Apply the proposed ES framework to real-world combinatorial tasks such as supply chain optimization, scheduling, network design, and portfolio selection where traditional methods are computationally expensive or suboptimal.

In summary, this study provides foundational insights into optimizer selection within ES and offers practical implications for the design of efficient neural combinatorial optimization systems.

4.1 Application of the combinatorial problems

- Travelling Salesman Problem is used in logistics and route planning like in optimizing delivery routes for courier deliveries.
- Knapsack Problem is used in portfolio management: selecting a subset of investments that maximize returns while staying within set budget limit or risk tolerance.
- Bin packing problem is used cloud storage systems,where the goal is to efficiently distribute files into a limited number of storage devices to minimize unused space, while staying within storage capacity limits.

Bibliography

[1] A. H. C. Correia, D. E. Worrall, and R. Bondesan.
"Neural Simulated Annealing."
arXiv preprint, 2022. Available at: https://arxiv.org/pdf/2203.02201.pdf

[2] E. Aarts and J. Korst.
Simulated Annealing and Boltzmann Machines.
Wiley, 1989. Available at: https://www.wiley.com/en-us/Simulated+Annealing+and+Boltzmann+Machines-p-9780471920862.

[3] D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook.
The Traveling Salesman Problem: A Computational Study.
Princeton University Press, 2007. Available at: https://press.princeton.edu/books/hardcover/9780691129938/the-traveling-salesman-problem.

[4] S. Martello and P. Toth.
Knapsack Problems: Algorithms and Computer Implementations.
John Wiley & Sons, Inc., 1990. Available at: https://www.wiley.com/en-us/Knapsack+Problems%3A+Algorithms+and+Computer+Implementations-p-9780471924202.