End-to-End PPO-Based Reinforcement Learning Framework for Scalable 0/1 Knapsack Problem Solving

M.Sc. Data Science

Gang Lin

School of Computer Science

College of Engineering and Physical Sciences

University of Birmingham

2024-25

Abstract

The 0/1 Knapsack Problem (KP) is a canonical NP-complete combinatorial optimization challenge with wide-ranging applications. While traditional exact algorithms like Dynamic Programming are computationally infeasible for large-scale instances due to exponential time or memory complexity, recent deep learning approaches often struggle with generalization to problem sizes unseen during training. This paper addresses this scalability challenge by proposing a novel, end-to-end reinforcement learning (RL) framework designed for high-performance generalization.

We present a solution based on the Proximal Policy Optimization (PPO) algorithm, employing a sophisticated model architecture that features a shared Transformer encoder to capture the global, combinatorial relationships between items, an actor head with a Pointer Network-based decoder for item selection, and a separate Multi-Layer Perceptron (MLP) critic head for stable value estimation. This entire workflow is encapsulated in a reusable Python package, which handles data generation, preprocessing, model training, and comparative evaluation against various benchmarks.

Trained exclusively on small-scale KP instances with 5 to 50 items, our model demonstrates powerful generalization capabilities. When tested on large-scale instances of up to 200 items, the framework achieves a stable Mean Relative Error (MRE) below 30% against optimal solutions derived from the Gurobi solver. Experimental results confirm that our approach significantly outperforms baseline neural architectures, including pure MLPs and earlier Pointer Network models, in terms of solution quality and exhibits greater training stability than the REINFORCE algorithm. This work validates the effectiveness of a PPO-based framework for solving combinatorial optimization problems, presenting a robust and scalable method that bridges the gap between traditional solvers and modern neural networks.

Keywords: 0/1 Knapsack Problem, Reinforcement Learning, Proximal Policy Optimization (PPO), Generalization, Transformer, Pointer Network

Acknowledgements

Declarations

I certify that this project is my own work. Code development and debugging was assisted by Gemini 2.5 Pro combined with Qwen3-235B-A22B-2507. The report structure and initial drafts are mine; final editing was assisted by Gemini 2.5 Pro for clarity and consistency. All outputs have been reviewed and edited to accurately reflect my work.

Abbreviations

COP	Combinatorial Optimization Problem
CVRP	Capacitated Vehicle Routing Problem
DACT	Dual-Aspect Collaborative Transformer
DQN	Deep Q-Network
DP	Dynamic Programming
DRL	Deep Reinforcement Learning
FPTAS	Fully Polynomial-Time Approximation Scheme
GA	Genetic Algorithm
GAE	Generalized Advantage Estimation
GCN	Graph Convolutional Network
JSP	Job Shop Scheduling
KP	Knapsack Problem
LKH	Lin-Kernighan-Helsgaun
MDP	Markov Decision Process
ML	Machine Learning
MLP	Multi-Layer Perceptron
MRE	Mean Relative Error
NP	Non-deterministic Polynomial-time
PPO	Proximal Policy Optimization
Ptr-Net	Pointer Network
RL	Reinforcement Learning
RNN	Recurrent Neural Network
SA	Simulated Annealing
SL	Supervised Learning
TSP	Traveling Salesman Problem

Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

1.1 Research Overview

This dissertation documents the development of an end-to-end reinforcement learning (RL) framework for solving the 0/1 Knapsack Problem (KP) with a primary focus on scalability and generalization. The research journey commenced with an empirical validation of classical optimization algorithms. Initial experiments with methods such as Dynamic Programming and Branch and Bound confirmed their well-documented theoretical limitations; they proved to be computationally infeasible for problems with large capacity constraints or a high number of items, suffering from excessive memory usage and exponential runtime complexity respectively.

Following this, the investigation transitioned to machine learning paradigms. A preliminary exploration using a standard Multi-Layer Perceptron (MLP) architecture quickly revealed its inability to learn a generalizable solving policy, as evidenced by its poor performance on unseen problem sizes. To overcome the limitations of supervised learning, which relies on the availability of high-quality, optimal solution labels, this research pivoted towards reinforcement learning. The theoretical foundation for this shift is the inherent connection between dynamic programming and RL, as the DP recurrence for the Knapsack Problem can be viewed as a deterministic instance of the Bellman equation [6].

The initial RL approach was inspired by the seminal work of Bello et al. [1], which utilized a Pointer Network with the REINFORCE algorithm. However, our replication and experimentation with this method highlighted significant training instability and high variance, which are common challenges associated with vanilla policy gradient methods. To address these shortcomings, this work proposes a substantially evolved framework. The final architecture employs a shared Transformer encoder to effectively capture the combinatorial inter-dependencies of the items, a Pointer Network-based decoder as the actor, and an MLP critic head. The training is stabilized and made more efficient by leveraging the Proximal Policy Optimization (PPO) algorithm [5]. This novel framework successfully learns a scalable policy, achieving a solution accuracy of approximately 70% (equivalent to a Mean Relative Error below 30%) on large-scale generalization tests.

1.2 Motivation

The 0/1 Knapsack Problem (KP) is a canonical problem in the field of combinatorial optimization and computer science, formally classified as NP-complete [2]. Its importance is twofold. Theoretically, as many NP-complete problems are reducible to one another, advancements in solving the KP can provide insights applicable to a wide range of other computationally hard problems. Practically, the KP serves as a mathematical abstraction for numerous real-world decision-making scenarios, including portfolio optimization, resource allocation, and logistics planning.

Despite its significance, the development of scalable and generalizable solvers remains a major challenge. Traditional exact algorithms, while guaranteeing optimality, are not viable for large-scale instances that are common in industrial applications. This has motivated a growing interest in machine learning-based approaches [8]. However, a review of recent literature reveals a critical research gap: the majority of existing neural network-based solvers are designed and trained for specific, fixed problem sizes [1]. These models often fail to generalize their learned policies to instances with a different number of items than they were trained on, severely limiting their practical utility.

Furthermore, within the sub-field of deep reinforcement learning for combinatorial optimization, the fundamental 0/1 Knapsack Problem has received surprisingly little attention compared to other problems like the Traveling Salesperson Problem. The lack of a truly end-to-end framework that can be trained on smaller instances and effectively generalize to larger, unseen ones represents a significant and compelling opportunity for research. This dissertation is motivated by the goal of addressing this gap by developing a novel RL framework that learns a robust and scalable policy for the 0/1 Knapsack Problem.

1.3 Contributions

This dissertation presents a series of contributions to the field of applying deep reinforcement learning to combinatorial optimization problems. The primary contributions are summarized as follows:

- **A Novel, Generalizable RL Framework for the 0/1 Knapsack Problem:** We propose and validate an end-to-end deep reinforcement learning framework built upon the Proximal Policy Optimization (PPO) algorithm [5]. The architecture's core, a shared Transformer encoder [4] paired with a Pointer Network decoder [1] and an MLP critic, is specifically designed to learn the underlying structure of the KP, enabling it to scale effectively.
- **Empirical Demonstration of Scalable Generalization:** A key contribution is the rigorous empirical evidence that our model learns a genuine, scalable solving policy rather than merely memorizing solutions for fixed-size problems. Trained exclusively on small-scale instances (5 to 50 items), the model demonstrates strong generalization to large, unseen instances (up to 200 items), maintaining a stable and low Mean Relative Error below 30% when compared to the optimal solutions provided by the Gurobi solver. This capability marks a significant advancement over prior neural approaches that were limited to fixed problem dimensions.
- **A Comprehensive and Reproducible Software Platform:** The entire framework has been implemented as a robust and reusable Python package. By leveraging well-regarded libraries such as Gymnasium and Stable Baselines 3, this work provides a comprehensive platform for the entire research pipeline, including procedural data generation, model training, and the comparative evaluation of RL agents against both traditional algorithms and commercial solvers.

This platform serves as a valuable tool for ensuring reproducibility and fostering further research in the domain.

CHAPTER 2

Background

2.1 What is the 0/1 Knapsack Problem?

The Knapsack Problem (KP) is a quintessential problem in combinatorial optimization, serving as a fundamental model for resource allocation under constraints. It is formally classified as an NP-complete problem, meaning there is no known algorithm that can find the optimal solution in polynomial time for all instances. This work focuses on the most common and classic variant: the 0/1 Knapsack Problem.

2.1.1 Problem Definition

In the 0/1 Knapsack Problem, one is given a set of items, each with an associated weight and value, and a knapsack with a fixed capacity. The objective is to select a subset of these items to place into the knapsack such that the total value of the selected items is maximized, without exceeding the knapsack's weight capacity.

The "0/1" property is a crucial constraint: for each item, the decision is binary. The item can either be fully included in the knapsack (represented by 1) or completely excluded (represented by 0). It is not possible to include a fraction of an item.

2.1.2 Mathematical Formulation

The problem can be formulated formally as follows. Let there be n items. For each item $i \in \{1, 2, \dots, n\}$, let:

- $v_i > 0$ be its value,
- $w_i > 0$ be its weight.

Let W be the maximum capacity of the knapsack.

We define a binary decision variable x_i for each item i:

$$x_i = \begin{cases} 1 & \text{if item } i \text{ is selected,} \\ 0 & \text{if item } i \text{ is not selected.} \end{cases}$$

The goal is to choose the values for $x_1, x_2, \dots, x_n$ in order to solve the following integer linear programming problem:

$$\text{maximize} \quad Z = \sum_{i=1}^{n} v_i x_i \tag{2.1}$$

$$\text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq W \tag{2.2}$$

$$\text{with} \quad x_i \in \{0,1\} \quad \forall i \in \{1,2,\ldots,n\} \tag{2.3}$$

Here, Equation 2.1 represents the objective function, which is the total value of the items selected. Equation 2.2 is the capacity constraint, ensuring that the total weight of the selected items does not surpass the knapsack's limit. Finally, Equation 2.3 enforces the binary nature of the decision for each item.

2.2 What is Dynamic Programming?

Dynamic Programming (DP) is a classical method for solving complex optimization problems by breaking them down into simpler, overlapping subproblems. As an exact algorithm, DP guarantees finding the optimal solution. For the 0/1 Knapsack Problem, DP is not only a cornerstone of traditional computer science but also provides the theoretical bedrock for virtually all modern reinforcement learning approaches. The core principle connecting them is the Bellman equation, which is fundamental to both methodologies.

DP can be implemented in two primary ways:

- **Tabulation (Bottom-up):** This approach systematically solves every subproblem, starting from the smallest ones, and stores the results in a table (or a multi-dimensional array). It then builds upon these stored results to solve larger and more complex subproblems until the final solution is reached. This method is exhaustive as it computes the value for every possible state.
- **Memoization (Top-down):** This approach is a recursive method that solves the main problem by breaking it down. However, it stores the result of each computed subproblem in a lookup table. Before computing a subproblem, it first checks if the result is already stored. If it is, the stored result is reused; otherwise, the subproblem is computed and its result is stored for future use. This way, it only computes the states that are actually reached during the recursive calls.

2.2.1 The State-Transition Equation

For the 0/1 Knapsack Problem, the DP approach relies on defining a state and a value function. Let $V(i, w)$ be the maximum value that can be obtained using a subset of the first i items with a knapsack capacity of w. The state-transition equation, or DP recurrence, is defined as follows:

$$V(i,w) = \begin{cases} V(i-1,w) & \text{if } w_i > w \\ \max(V(i-1,w), v_i + V(i-1, w-w_i)) & \text{if } w_i \leq w \end{cases} \tag{2.4}$$

The first case corresponds to the decision of not including item i (as it exceeds the current capacity w), so the optimal value is the same as the one obtained with $i-1$ items. The second case represents the core decision: we take the maximum of either excluding item i (the value remains $V(i-1,w)$) or including it (the value is v_i plus the optimal value achievable with $i-1$ items and the remaining capacity $w - w_i$).

This recurrence is directly analogous to the Bellman equation, which is central to reinforcement learning:

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R_{t+1} + \gamma V^{\pi}(S_{t+1}) | S_t = s] \tag{2.5}$$

The principles are identical: the value of a state is determined by the immediate reward plus the value of the subsequent state(s). In the context of a deterministic, model-based problem like the Knapsack Problem, the DP recurrence (Equation 2.4) is an exact, specific instance of the Bellman equation (Equation 2.5). The key distinction is that the Bellman equation provides a more general framework that can be extended to model-free scenarios where the state-transition probabilities are unknown and must be learned.

2.2.2 Complexity and Limitations

Despite its optimality, the standard DP approach for the knapsack problem has significant drawbacks that limit its applicability to large-scale instances:

- **High Complexity:** Both the tabulation and memoization methods suffer from high computational complexity. The time and space complexity are both $O(nW)$, where n is the number of items and W is the knapsack capacity. This leads to slow execution times and the risk of memory explosion when W is exponentially large relative to the input size.
- **Pseudo-Polynomial Time:** The $O(nW)$ complexity is considered pseudo-polynomial because the runtime depends on the numeric value of the input W, not just the size of the input (i.e., the number of bits required to represent W). When W grows exponentially with respect to its bit-length (e.g., $W = 2^n$), the runtime becomes exponential in the input size—a hallmark of NP-hard problems.

2.2.3 Optimizations and Their Inadequacies

Several optimizations have been developed to mitigate these issues, but they do not fully resolve the scalability problem for large instances.

- **Space Optimization:** The space complexity of the tabulation method can be optimized from $O(nW)$ to $O(W)$ by noticing that the calculation for row i only depends on the results from row $i - 1$. This can be achieved by using a one-dimensional array. However, this optimization comes at the cost of losing the ability to reconstruct the optimal item set without additional bookkeeping, as intermediate results are overwritten. Furthermore, the time complexity remains $O(nW)$, which is still prohibitive.
- **Fully Polynomial-Time Approximation Scheme (FPTAS):** For the 0/1 Knapsack Problem, an FPTAS exists, which guarantees a solution with a value of at least $(1 - \epsilon)$ times the optimal value. It works not by normalizing capacity, but by scaling and rounding the item *values*. Given a desired approximation ratio $\epsilon > 0$, the algorithm proceeds by setting a scaling factor $K = (\epsilon v_{\max})/n$, where $v_{\max} = \max_i v_i$. Each item's value is then transformed to $v'_i = \lfloor v_i/K \rfloor$. A standard DP algorithm is then run on these scaled, integer values. The time complexity of this approach is $O(n^3/\epsilon)$, which is polynomial in both n and $1/\epsilon$. Despite this, for the large-scale problems with many items addressed in this research, the computational cost can still be unacceptable.

2.3 Basics of Reinforcement Learning

Given the inherent limitations of both exact algorithms and traditional heuristics for solving large-scale combinatorial optimization problems, this research turns to Reinforcement Learning (RL) as a promising alternative paradigm.

2.3.1 Why Reinforcement Learning?

The motivation for employing RL is best understood by comparing it to other dominant approaches for solving complex problems: supervised learning, commercial solvers, and metaheuristics.

- **In Comparison to Supervised Learning (SL):** The primary advantage of RL is its independence from large datasets of high-quality, labeled solutions. For a problem like the 0/1 Knapsack, generating such labels would require running a commercial exact solver (e.g., Gurobi). This presents a paradox: if a commercial solver can efficiently find the optimal solution to generate a label, the problem for that specific instance is already solved, rendering the training of an SL model for that purpose somewhat redundant. RL circumvents this by learning directly from interaction and a scalar reward signal. However, it is worth noting that SL can be synergistically combined with RL, for instance, by using imitation learning to pre-train an agent from expert (solver-generated) demonstrations to accelerate the learning process.
- **In Comparison to Commercial Solvers:** While solvers like Gurobi or CPLEX are powerful tools that guarantee optimality, their runtime often scales exponentially with problem size for NP-hard problems. For very large or complex instances, finding the optimal solution can become computationally intractable. Furthermore, they typically operate as deterministic "black boxes," offering less flexibility in stochastic or dynamic environments. In contrast, a trained RL agent learns a *policy*—a mapping from states to actions—that can generate high-quality solutions very quickly (often in constant or near-constant time) once trained, making it highly suitable for applications requiring rapid decision-making.
- **In Comparison to Metaheuristics:** Algorithms like Genetic Algorithms (GA) or Simulated Annealing (SA) provide another alternative. However, their performance is highly sensitive to a set of hyperparameters that require extensive, problem-specific tuning. Crucially, these parameters often need to be re-tuned when the scale or characteristics of the problem instances change. Moreover, they can be prone to premature convergence to local optima. An RL model, once successfully trained, encapsulates a generalized solving strategy. It can be invoked directly to solve any problem instance within the distribution it was trained on, up to a specified maximum scale, without the need for re-tuning for each new instance.

2.3.2 What is Reinforcement Learning?

Reinforcement Learning is a paradigm of machine learning where an *agent* learns to make a sequence of optimal decisions by interacting with an *environment*. The agent's goal is to learn a policy that maximizes a cumulative reward signal over time. This interaction is typically modeled as a Markov Decision Process (MDP).

The core of RL is the agent-environment interaction loop, as depicted in Figure 2.1. At each timestep t, the agent observes the environment's state S_t and, based on its policy, selects an action A_t. The environment transitions to a new state S_{t+1} and provides the agent with a scalar reward R_{t+1} as feedback on its action. This process repeats, allowing the agent to learn from the consequences of its actions.

A critical distinction from other machine learning paradigms is the nature of the training data. The agent does not learn from a static dataset of raw problem data (e.g., item weights and values). Instead, the learning process is driven by data generated dynamically through interaction. This data takes the form of trajectories or experiences, which are sequences of tuples:

$$(S_t, A_t, R_{t+1}, S_{t+1})$$

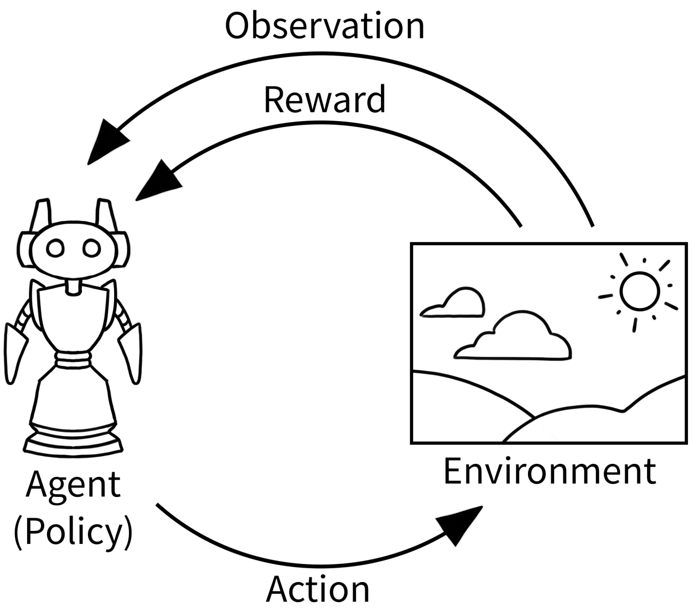

Figure 2.1: The standard agent-environment interaction loop in Reinforcement Learning. Source: Gymnasium Documentation.

By collecting these experiences, the agent refines its policy to favor actions that lead to higher long-term cumulative rewards. For the Knapsack Problem, this means learning a sequence of item selections that ultimately yields the highest total value without violating the capacity constraint.

2.3.3 From Bellman to PPO: An Evolutionary Trajectory

The transition from Dynamic Programming (DP) to modern reinforcement learning algorithms like Proximal Policy Optimization (PPO) is not arbitrary. It follows a coherent, problem-driven evolution rooted in the Bellman equation. This evolution can be understood as a sequence of adaptations, each designed to overcome a critical limitation of the previous approach to achieve the scalability, efficiency, and stability required for complex problems like the 0/1 Knapsack Problem.

The Foundational Bellman Equation

As established in Section 2.2, the DP recurrence is a specific instance of the Bellman principle. The general, element-wise Bellman equation for the state-value function $V^\pi(s)$ is:

$$V_\pi(s) = \sum_{a\in A} \pi(a|s) \left[\sum_{r\in R} p(r|s,a)r + \gamma \sum_{s'\in S} p(s'|s,a)V_\pi(s') \right] \tag{2.6}$$

Here, the terms are defined as:

- $\pi(a|s)$ is the **policy**, defining the probability of taking action a in state s.
- $p(s'|s,a)$ and $p(r|s,a)$ together form the **model** of the environment, specifying the probabilities of transitioning to state s' and receiving reward r, respectively.

- γ is the **discount factor** ($0 \leq \gamma \leq 1$), which balances immediate and future rewards. Although the Knapsack Problem is theoretically an undiscounted, finite-horizon problem, our framework adopts $\gamma = 0.99$. This choice aligns with standard practice in deep reinforcement learning, where a discount factor close to 1 is often used to stabilize training and reduce the variance of value estimates. The state transitions in the KP environment are also deterministic constants.

DP is a **model-based** approach because it requires full knowledge of the model (the transition and reward probabilities) to compute the value function. In contrast, most RL applications are **model-free**, learning a policy through trial-and-error without explicit knowledge of the environment's dynamics.

Step 1: Bellman to Value Function Approximation

Limitation of DP: A tabular representation of $V(s)$ is intractable for large-scale problems due to exponential memory and computational requirements.

Adaptation: Replace the exact, tabular value function with a *learnable function approximator* $V_\phi(s)$, typically a neural network with parameters ϕ. This allows generalization across states and enables application to high-dimensional state spaces.

$$V^\pi(s) \quad \longrightarrow \quad V_\phi(s) \approx V^\pi(s)$$

Step 2: Value Approximation to Policy Gradient

Limitation of Value-Based Methods: Methods like DQN, which learn a value function and derive a policy indirectly (e.g., $\pi(s) = \arg\max_a Q(s,a)$), are ill-suited for the Knapsack Problem. The action space (the set of available items) is large, discrete, and dynamically changes at each step. Iterating through all possible actions to find the 'argmax' is inefficient and does not scale. A more effective approach is to directly learn a stochastic policy that outputs a probability distribution over the available items.

Adaptation: Directly parameterize the policy as $\pi_\theta(a|s)$ and optimize its parameters θ to maximize the expected return. This transition involves several layers of approximation:

1. **Function Approximation:** The policy π is approximated by a neural network π_θ.
2. **Sampling Approximation:** The expectation $\mathbb{E}_{\tau \sim \pi_\theta}[\cdot]$ over all possible trajectories is approximated by averaging over a finite batch of trajectories sampled from the current policy.
3. **Policy Gradient Theorem:** This provides a theoretical tool to compute the gradient of the expected return, enabling optimization via gradient ascent.

The REINFORCE algorithm implements this via the policy gradient, using a Monte Carlo estimate of the expected return.

Step 3: Monte Carlo Estimation to Actor-Critic (TD)

Limitation of Monte Carlo Estimation: The REINFORCE algorithm uses the full cumulative return $G_t = \sum_{k=t}^{T} \gamma^{k-t} R_{k+1}$ to update the policy. This Monte Carlo estimator, while unbiased, suffers from high variance because the return for an action at timestep t is affected by all subsequent rewards and stochastic decisions. This results in noisy gradient estimates and unstable learning.

Adaptation: Replace the MC estimator with a lower-variance Temporal Difference (TD) estimator. TD learning updates an estimate based on an observed reward and another learned estimate (a process called bootstrapping). The TD error, $\delta_t = R_{t+1} + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$, provides a less noisy, step-by-step learning signal. In the Actor-Critic framework, this principle is used to compute the *advantage function*, $\hat{A}_t$, which measures how much better an action is than the average expectation from that state.

$$G_t \quad \longrightarrow \quad \hat{A}_t = R_{t+1} + \gamma V_\phi(s_{t+1}) - V_\phi(s_t)$$

Using $\hat{A}_t$ as the credit assignment signal in the policy gradient update reduces variance and enables more efficient and stable online learning.

Step 4: Actor-Critic to Proximal Policy Optimization (PPO)

Limitation of standard Actor-Critic: Unconstrained policy updates can still lead to catastrophic performance drops. A single bad mini-batch could result in an overly large gradient step, moving the policy to a poor region of the parameter space from which it cannot easily recover.

Adaptation: PPO [5] introduces a *clipped surrogate objective* that limits the magnitude of policy change at each update, effectively creating a trust region.

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

where the probability ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ measures the policy change. The clipping mechanism prevents $r_t(\theta)$ from deviating too far from 1, thereby penalizing large policy updates. This simple but effective stabilization allows PPO to achieve robust, high-performance learning, making it the algorithm of choice for this research.

Summary of the Evolutionary Path

The entire trajectory, driven by the need to address specific limitations at each stage, is summarized in Table 2.1.

Table 2.1: Evolutionary Path from Dynamic Programming to PPO

Stage	**Key Change from Previous Stage**
1. DP	(Baseline) Solves problems using a complete, exact state-value table.
2. Value-Based RL	Replaces the exact table with a neural network (*function approximator*) to handle large state spaces.
3. Policy Gradient	Directly parameterizes and optimizes the policy network, instead of learning values to infer a policy.
4. Actor-Critic / PPO	Uses a TD-based advantage estimate to replace high-variance Monte Carlo returns, and adds a clipping mechanism to stabilize training.

CHAPTER 3

Related Work

While traditional algorithms such as Dynamic Programming provide the theoretical foundation for solving Combinatorial Optimization Problems (COPs), contemporary research has predominantly shifted towards Machine Learning (ML) methodologies to overcome their scalability limitations. Within the last decade, and especially in the last few years, Reinforcement Learning (RL) has emerged as the dominant paradigm for training agents to solve these problems, owing to its ability to learn effective heuristics through direct interaction with a problem environment. Following the effective taxonomy proposed in the survey by Wang et al. [8], this chapter reviews the state-of-the-art by categorizing these DNN-based approaches into two main families: *constructive* and *improvement* methods.

3.1 Constructive Methods

Constructive methods build a solution from scratch, typically in a sequential, element-by-element manner. This paradigm is largely dominated by encoder-decoder architectures that learn to output a sequence or a set of items that form the final solution.

3.1.1 Vanilla Pointer Networks

The initial application of neural networks to constructive solutions was marked by the Pointer Network (Ptr-Net), first proposed by Vinyals et al. to be trained with Supervised Learning (SL) on solution labels from existing solvers [8]. However, the reliance on expensive-to-generate optimal labels was a significant bottleneck. A major breakthrough came from **Bello et al. (2017)** [1], who successfully trained a Ptr-Net using the REINFORCE algorithm with a critic baseline. This pivotal work demonstrated that an agent could learn to solve both the Traveling Salesman Problem (TSP) and the Knapsack Problem effectively without supervised data. Following this, Nazari et al. extended this RL-based Ptr-Net approach to handle problems with dynamic elements, such as the Capacitated Vehicle Routing Problem (CVRP).

3.1.2 Pointer Networks with Self-Attention

A crucial architectural evolution was the replacement of Recurrent Neural Network (RNN) based encoders with the Transformer architecture, which uses self-attention mechanisms to better capture the global context of the entire problem instance at once. Deudon et al. were among the first to apply this to COPs. A highly influential paper by **Kool et al. (2019)** [4] solidified this trend with their "Attention Model". They paired a Transformer encoder with a more efficient "rollout baseline" for the REINFORCE algorithm, achieving state-of-the-art results on TSP and CVRP and setting a new benchmark for subsequent research.

3.1.3 Hierarchical Pointer Networks

To tackle large-scale COPs or those with complex constraints, researchers have developed hierarchical frameworks. These methods typically decompose a large problem into smaller, manageable subproblems, which are then solved by a neural model. For instance, Ma et al. presented a Graph Pointer Network within a hierarchical RL architecture to address TSP with time windows. Similarly, other works like Pan et al. have successfully applied hierarchical models to solve large-scale TSP (LSTSP), demonstrating a promising direction for scaling neural COP solvers [8].

3.2 Improvement Methods

In contrast to constructive methods, improvement methods begin with a complete (often randomly or greedily generated) solution and iteratively refine it. This is typically achieved by learning a policy that guides a local search or other heuristic optimization process.

3.2.1 Local Search Based on Neural Networks

This class of methods trains a neural network to act as a sophisticated operator within a local search framework. The network learns to identify which parts of a current solution are most promising to modify. Chen and Tian developed the "NeuRewriter" framework, where a Ptr-Net trained with an Actor-Critic algorithm learns to select both a rewrite rule and a region of the solution to apply it to, showing strong results on Job Shop Scheduling (JSP) and CVRP. More recently, architectures have shifted towards Transformers. For example, Ma et al. proposed the Dual-Aspect Collaborative Transformer (DACT) and used **PPO** to train a policy that improves CVRP solutions iteratively [8].

3.2.2 Heuristics Assisted by Neural Networks

Another effective improvement strategy is to use a DNN to enhance a critical component of a powerful traditional heuristic. The Lin-Kernighan-Helsgaun (LKH) algorithm is one of the most effective traditional heuristics for the TSP. Recognizing this, Xin et al. proposed NeuroLKH, which uses a Graph Convolutional Network (GCN) to predict a promising set of candidate edges. This learned knowledge is then used to guide and prune the search space of the LKH algorithm, achieving remarkable performance by combining the strengths of both deep learning and classical heuristics [8].

3.3 The Rise of Proximal Policy Optimization

The training stability of early policy gradient methods like REINFORCE was a significant challenge, as they are known to suffer from high variance in gradient estimates, leading to inefficient learning. A landmark development in this area was the introduction of the Proximal Policy Optimization (PPO) algorithm by **Schulman et al. (2017)** [5]. PPO stabilizes the training process by optimizing a "clipped surrogate objective" function, which discourages the policy from changing too drastically in a single update. This provides the reliability of more complex trust-region methods but with a much simpler implementation, striking a favorable balance between sample complexity, simplicity, and performance. Due to its robustness, PPO has become a go-to algorithm in the RL community and has been successfully applied to COPs. For example, **Gholipour et al. (2023)** [3] leveraged a Transformer-PPO architecture to effectively solve the task offloading problem in edge computing, demonstrating the power of this algorithmic combination.

3.4 Positioning the Present Work

This dissertation situates itself within the modern, end-to-end constructive paradigm while leveraging state-of-the-art advancements in RL algorithms. The proposed framework is built upon the following foundations:

- It follows the **constructive, reinforcement learning** approach pioneered by Bello et al. [1], learning a policy from scratch without reliance on supervised labels.
- It adopts a **Transformer-based encoder**, in line with the current state-of-the-art for capturing complex, non-sequential relationships in combinatorial problems, following the success of models like that of Kool et al. [4].
- It employs the **Proximal Policy Optimization (PPO)** algorithm for its proven training stability and high performance, as established by Schulman et al. [5].

While many advanced DRL applications have focused on routing problems like TSP and CVRP, this work applies the powerful Transformer-PPO architecture specifically to the 0-1 Knapsack Problem with a strong emphasis on **generalization** from small to large-scale instances. This addresses a critical gap, as many prior works were confined to fixed problem sizes [1]. Recent related studies, such as that by Zhang et al. (2025) [9], have also tackled the KP with RL but have explored different avenues, such as value-based Dueling DQN combined with state representation normalization and Noisy layers for exploration. Our focus on a state-of-the-art policy gradient method contributes a valuable perspective to the ongoing effort to develop scalable and generalizable solvers for this fundamental combinatorial optimization problem.

CHAPTER 4

Methodology

This chapter details the methodology developed to solve the 0/1 Knapsack Problem (KP) using a scalable, end-to-end deep reinforcement learning framework. We begin by reformulating the combinatorial optimization problem as a sequential decision process suitable for RL. Subsequently, we describe the overall reinforcement learning framework, justify the selection of the Proximal Policy Optimization (PPO) algorithm, and present our novel neural network architecture. Finally, we elaborate on the critical mechanisms of reward design and action masking that ensure solution feasibility, and conclude with the specifics of our training strategy and implementation to ensure reproducibility.

4.1 Problem Formulation as a Sequential Decision Process

The first critical step in applying reinforcement learning is to reframe the KP from a static combinatorial optimization problem into a sequential decision-making task. To this end, we formally model the problem as a finite-horizon, discounted Markov Decision Process (MDP) with a discount factor $\gamma = 0.99$. An agent interacts with a KP environment over a sequence of discrete timesteps, selecting one item at a time to place into the knapsack. The components of the MDP are defined as follows:

- **State (s_t)**: The state at timestep t provides a complete snapshot of the current problem-solving status. It is a dictionary containing all necessary information for the agent to make a decision. The state space S is defined by the tuple $s_t = \{\mathbf{I}, C_t, \mathbf{m}_t\}$, where:
  - $\mathbf{I} \in \mathbb{R}^{N_{max} \times 2}$ is a tensor representing the features of all items (normalized weight and value), padded to a maximum length N_{max}.
  - $C_t \in \mathbb{R}^+$ is a scalar representing the remaining capacity of the knapsack at timestep t.
  - $\mathbf{m}_t \in \{0, 1\}^{N_{max}}$ is a binary mask vector indicating which actions (items) are currently valid.

- **Action (a_t)**: An action a_t corresponds to the selection of a single item i from the set of currently available items. This selection is determined by the policy network, which, in our case, is implemented as a Pointer Network that outputs a probability distribution over the items.
- **Action Space (A_t)**: The action space is dynamic and state-dependent. At any given state s_t, the set of valid actions consists of all items i that have not yet been selected and whose weight w_i does not exceed the remaining capacity C_t. This dynamic control over the action space is enforced through the action mask $\mathbf{m}_t$.
- **Reward (r_t)**: To guide the agent's learning process, we employ a dense reward scheme. When the agent selects a valid item i, it receives an immediate reward equal to that item's value, $r_t = v_i$. This provides a direct and immediate signal about the quality of each incremental decision.
- **Episode Termination**: An episode concludes when no more items can be placed into the knapsack, i.e., when the action mask $\mathbf{m}_t$ becomes a zero vector. This occurs either because all items have been considered or because the weight of every remaining item exceeds the current capacity.

By structuring the KP in this autoregressive manner, we transform it into a constrained combinatorial search process that is exceptionally well-suited for being solved by a neural agent that learns a sequential decoding policy.

4.2 Reinforcement Learning Framework

Our framework is built upon two widely adopted, high-quality open-source libraries: `Gymnasium` and `Stable-Baselines3`. `Gymnasium`, a fork of OpenAI's original Gym library, provides a standardized API for defining and interacting with reinforcement learning environments. By conforming to this API, our custom ‘KnapsackEnv‘ becomes compatible with a vast ecosystem of RL algorithms.

For the algorithm implementation, we use `Stable-Baselines3`, a library that provides robust, well-tested, and state-of-the-art implementations of deep RL algorithms. The choice of Proximal Policy Optimization (PPO) [5] from this library is deliberate. PPO is an on-policy, actor-critic algorithm renowned for its stability and sample efficiency. Its signature feature is a clipped surrogate objective function, which prevents excessively large policy updates and maintains a trust region around the current policy. This mechanism ensures more stable and reliable training convergence, a crucial factor when dealing with complex policy landscapes like those in combinatorial optimization. This stability was empirically validated in our preliminary experiments, where PPO consistently converged to high-quality policies.

4.3 Neural Architecture Design

The core of our proposed method is a novel neural architecture designed to effectively capture the complex, non-sequential relationships inherent in the Knapsack Problem. The model follows an encoder-decoder structure with a shared encoder and two distinct heads for the actor and the critic, as depicted in Figure 4.1.

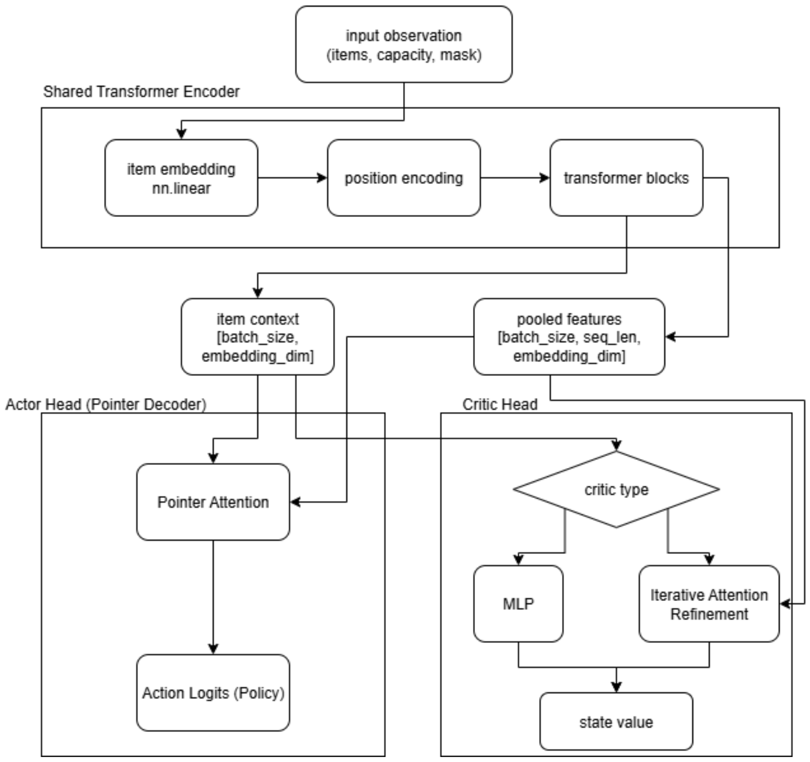
input observation (items, capacity, mask)
Shared Transformer Encoder
item embedding nn.linear
position encoding
transformer blocks
item context [batch_size, embedding_dim]
pooled features [batch_size, seq_len, embedding_dim]
Actor Head (Pointer Decoder)
Critic Head
Pointer Attention
Action Logits (Policy)
critic type
MLP
Iterative Attention Refinement
state value

Figure 4.1: The proposed neural architecture. An input observation is processed by a shared Transformer encoder to produce item-specific contexts and a global pooled feature vector. These are then consumed by a Pointer Network-based actor head and an MLP-based critic head. In our best-performing model, a simple MLP architecture was used for the critic.

4.3.1 Shared Transformer Encoder

The encoder's role is to process the raw problem instance and generate context-aware embeddings for each item. The input for each item i is its normalized weight w_i and value v_i. We employ a **Transformer encoder** rather than a sequential model like an RNN. Its multi-head self-attention mechanism allows the model to weigh the importance of all other items when encoding a specific item, thereby capturing the global combinatorial structure of the problem. This is critical for the KP, where the decision to include one item is heavily dependent on the properties of all other available items. The encoder outputs two key tensors: a set of rich, context-aware embeddings for each item, $\mathbf{H}_{context} \in \mathbb{R}^{B \times N_{max} \times D}$, and a global problem embedding, $\mathbf{h}_{pooled} \in \mathbb{R}^{B \times D}$, derived by mean-pooling the item embeddings.

4.3.2 Actor Head: Pointer Network-based Decoder

The actor head, which implements the policy $\pi_\theta(a|s)$, is a **Pointer Network** [7]. This architecture is perfectly suited for our selection task as it directly outputs a probability distribution over the input items, rather than predicting a class from a fixed-size vocabulary. This avoids the limitation of a standard MLP, which would struggle to handle variable numbers of items and lacks the mechanism to directly reference inputs. At the same time, it is more efficient than a full Transformer decoder, as it does not need to generate new embeddings, only to point to existing ones.

Our implementation, the ‘PointerDecoder‘, operates autoregressively. At each decoding step, it uses an attention mechanism to decide which item to select next. Its operation can be broken down as follows:

1. **Initial Query**: At the first step, the decoder's initial query vector is the global problem embedding $\mathbf{h}_{\text{pooled}}$ from the encoder.
2. **Glimpse Mechanism**: To refine this query, we employ a ”glimpse” mechanism, as implemented in `custom_policy.py`. The query is used to attend over all item embeddings in $\mathbf{H}_{\text{context}}$. A weighted sum of the context vectors, based on the attention scores, creates a new, more refined query vector. This process is repeated for a fixed number of iterations (`n_glimpses=2`), allowing the model to iteratively focus its attention.
3. **Additive Attention**: The attention scores are calculated using an additive attention mechanism. Both the item context vectors and the current query vector are projected into a common space. Their sum is passed through a tanh activation, and the final score is computed via a dot product with a learnable parameter vector $\mathbf{v}$. The logits u_i for item i are computed as:

$$u_i = \mathbf{v}^T \tanh(\mathbf{W}_c \mathbf{h}_i + \mathbf{W}_q \mathbf{q})$$

   where $\mathbf{h}_i$ is the context embedding for item i, $\mathbf{q}$ is the current query vector, and $\mathbf{W}_c, \mathbf{W}_q, \mathbf{v}$ are learnable parameters.
4. **Output**: After the final glimpse, the refined query is used one last time to compute the final attention scores (logits) over all items. These logits, after applying the action mask, are passed through a softmax function to produce the final probability distribution $\pi_\theta(a_t|s_t)$.
5. **Autoregressive Update**: For subsequent steps in the decoding sequence, the query is updated to be the embedding of the item that was just selected in the previous step, informing the next decision.

4.3.3 Critic Head: MLP for Value Estimation

Based on our most successful experimental results (‘Exp26‘), the critic head is a simple yet effective Multi-Layer Perceptron (MLP) that estimates the state-value function $V_\phi(s)$. This contrasts with more complex attention-based critics, which we found did not yield superior performance for this problem.

The critic's architecture is intentionally straightforward. Its sole input is the **global problem embedding**, $\mathbf{h}_{pooled}$, which is the mean-pooled output of the shared Transformer encoder. This vector serves as a condensed representation of the entire state. The MLP then processes this vector through several linear layers with ReLU activations, LayerNorm, and Dropout to produce a single scalar value. This scalar represents the critic's estimate of the expected discounted cumulative reward (the value) from the current state.

By sharing the powerful Transformer encoder, the critic benefits from the same rich feature representation as the actor. However, using a simple MLP head for the value function, which is independent of the more complex actor head, improves training stability and efficiency.

4.4 Training Techniques

To ensure the reproducibility and robustness of our results, we employed several key training techniques and followed a standardized procedure.

- **Input Pre-processing**: Before an instance is fed to the agent, a crucial pre-processing step is applied: the items are sorted in descending order based on their value-to-weight ratio. This heuristic provides a strong inductive bias, presenting the items to the model in an order that is generally correlated with good solutions, which can simplify the policy learning landscape.
- **Observation Normalization**: We utilize `Stable-Baselines3`'s 'VecNormalize' wrapper. This powerful tool maintains a running average of the mean and variance of observations and normalizes them on-the-fly. Normalizing the item features (weights, values) and the remaining capacity to have zero mean and unit variance is critical for the stability of neural network training, preventing issues caused by shifting input distributions or disparate feature scales.
- **Reward Scaling**: The 'VecNormalize' wrapper is also configured to normalize rewards. This stabilizes the training of the critic by ensuring the value function targets remain within a consistent range, which is particularly important in PPO.
- **Action Masking**: As detailed in Section 4.1, we implement a mandatory action mask at each step. This is a hard constraint embedded into the decoding process, which offloads the burden of learning feasibility from the agent and guarantees that every generated solution is valid. This is fundamental to the framework's success.
- **Hyperparameter Tuning**: The final model was trained for 2.5 million timesteps with a linearly decaying learning rate from 1×10^{-5} to 1×10^{-6}, a batch size of 64, and an entropy coefficient of 0.01 to encourage exploration. All experiments were conducted with fixed random seeds to ensure that results can be replicated.

4.5 Training Algorithm

The training process is orchestrated by the Proximal Policy Optimization (PPO-Clip) algorithm, enhanced with Generalized Advantage Estimation (GAE) for variance reduction. The complete procedure, as implemented through `Stable-Baselines3`, is detailed in Algorithm 1. This algorithm iteratively collects a batch of experience from parallel environments and then performs multiple optimization epochs on the collected data.

Algorithm 1: PPO with GAE for Knapsack Problem Training

```
   Phase 1: Initialization
1  Initialize actor network $\pi_\theta(a|s)$ and critic network $V_\phi(s)$ with shared
     encoder;
2  Initialize Adam optimizers for parameters $\theta$ and $\phi$;
3  Initialize $N$ parallel ‘KnapsackEnv‘ environments;
4  for each training iteration do
       #--- Phase 2: Rollout / Data Collection ---
5      Initialize an empty Rollout Buffer $\mathcal{D}$;
6      for $t = 1 \to n\_steps$ do
7          for each parallel environment $i = 1 \to N$ do
8              Observe state $s_t^{(i)}$;
9              Sample action $a_t^{(i)} \sim \pi_{\theta_{\text{old}}}(\cdot|s_t^{(i)})$;
10             Compute value $V_\phi(s_t^{(i)})$ and log-probability $\log \pi_{\theta_{\text{old}}}(a_t^{(i)}|s_t^{(i)})$;
11             Execute action $a_t^{(i)}$ to get next state $s_{t+1}^{(i)}$ and reward $r_t^{(i)}$;
12             Let $\tau_t^{(i)} = (s_t^{(i)}, \ldots, \log \pi_{\theta_{\text{old}}}(a_t^{(i)}|s_t^{(i)}))$;
13             Store transition $\tau_t^{(i)}$ in $\mathcal{D}$;

       #--- Phase 3: Advantage and Return Estimation ---
14     Compute GAE advantage estimates $\hat{A}_t$ for all transitions in $\mathcal{D}$;
15     Compute returns-to-go $R_t \leftarrow \hat{A}_t + V_\phi(s_t)$ for all transitions in $\mathcal{D}$;
       #--- Phase 4: Optimization ---
16     for $k = 1 \to n\_epochs$ do
17         for each mini-batch of $(s, a, R, \hat{A}, \log \pi_{old})$ from $\mathcal{D}$ do
18             Calculate ratio: $r(\theta) \leftarrow \exp(\log \pi_\theta(a|s) - \log \pi_{\text{old}})$;
19             $L^{\text{CLIP}}(\theta) \leftarrow -\mathbb{E}\left[\min(r(\theta)\hat{A}, \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A})\right]$;
20             $L^{\text{VF}}(\phi) \leftarrow \mathbb{E}\left[(V_\phi(s) - R)^2\right]$;
21             $S[\pi_\theta] \leftarrow \mathbb{E}\left[\text{Entropy}(\pi_\theta(s))\right]$;
22             $L(\theta, \phi) \leftarrow L^{\text{CLIP}}(\theta) + c_1 L^{\text{VF}}(\phi) - c_2 S[\pi_\theta]$;
23             Update parameters $\theta, \phi$ by descending the gradient
                 $\nabla_{\theta,\phi} L(\theta, \phi)$;
```

CHAPTER 5

Results and Evaluation

5.1 Experimental Setup

5.2 Results

5.3 Evaluation and Discussion

CHAPTER 6

Conclusion and Future Work

6.1 Conclusions

6.2 Future Work

Bibliography

[1] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural Combinatorial Optimization with Reinforcement Learning, Jan. 2017.

[2] Q. Cappart, D. Chételat, E. Khalil, A. Lodi, C. Morris, and P. Veličković. Combinatorial optimization and reasoning with graph neural networks, Sept. 2022.

[3] N. Gholipour, M. D. de Assuncao, P. Agarwal, j. gascon-samson, and R. Buyya. TPTO: A Transformer-PPO based Task Offloading Solution for Edge Computing Environments, Dec. 2023.

[4] W. Kool, H. van Hoof, and M. Welling. Attention, Learn to Solve Routing Problems!, Feb. 2019.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal Policy Optimization Algorithms, Aug. 2017.

[6] A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel. Value Iteration Networks, Mar. 2017.

[7] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer Networks, Jan. 2017.

[8] F. Wang, Q. He, and S. Li. Solving Combinatorial Optimization Problems with Deep Neural Network: A Survey. *Tsinghua Science and Technology*, 29(5):1266–1282, Oct. 2024.

[9] Z. Zhang, H. Yin, L. Zuo, and P. Lai. Reinforcement Learning for Solving the Knapsack Problem. *Computers, Materials & Continua*, 0(0):1–10, 2025.

APPENDIX A

Software Manual

This appendix provides a comprehensive guide to the software platform developed for this research. The platform is an integrated framework designed to handle the entire research pipeline, including procedural data generation, model training, and comparative evaluation of various solvers.

A.1 Code Repository

The source code is hosted on both GitHub for public access and the University of Birmingham's internal GitLab server for academic version control.

- **Public GitHub Repository**: `https://github.com/Gavin-spring/ann_kp.git`
- **Internal Academic Repository**: `https://git.cs.bham.ac.uk/projects-2024-25/gxl386.git`

A.2 Installation Guide

A.2.1 System Requirements

- **Operating System**: A Linux-based distribution is highly recommended. The framework leverages Triton to compile and optimize the neural network models, which currently has the most robust support on Linux.
- **Hardware**: An NVIDIA GPU is required for training and evaluation. The codebase is optimized for CUDA and supports mixed-precision training to accelerate performance.
- **NVIDIA CUDA**: CUDA version 12.1 or newer is required.

A.2.2 Dependencies

All required Python packages are listed in the `requirements.txt` file in the root of the repository. Key dependencies include PyTorch, Gymnasium, and Stable-Baselines3.

A.2.3 Installation Steps

A bash script, `setup_vlab.sh`, is provided to automate the entire setup process. It is the recommended method for creating a clean and correct environment. Executing this script will perform the following actions:

1. Download and install Miniconda, a minimal installer for the Conda package manager.
2. Create a new Conda virtual environment to isolate project dependencies.
3. Install all required packages listed in `requirements.txt` using 'pip'.
4. Install Rclone, a command-line tool for managing files on cloud storage, to facilitate data transfer.
5. Install ngrok, a reverse proxy tool, to enable remote monitoring of training progress (e.g., via TensorBoard).

A.3 File Structure

The repository is organized into a modular structure to separate concerns and improve maintainability.

`data/` Contains the datasets for training, validation, and testing. Each problem instance is stored in a '.csv' file.

`src/` Contains the primary source code for the project, including:

- The implementation of the custom Gymnasium environment (`knapsack_env.py`).
- The source code for all solvers, including classical algorithms and neural network architectures (`custom_policy.py`, etc.).

`scripts/` Contains the main entry-point scripts for executing key pipeline stages, such as data generation, model training, and evaluation.

`artifacts/` The default output directory for results generated by the `train_model.py` and `evaluate_solvers.py` scripts. This includes trained models, logs, and evaluation metrics for the MLP, Pointer Network (REINFORCE), and classical solvers.

`artifacts_sb3/` A dedicated output directory for all experiments related to the Stable-Baselines3 PPO implementation. Results from `train_sb3.py` and `evaluate_sb3.py` are stored here.

`docs/` Contains supplementary documentation, including this manual.

`config.yaml` A centralized configuration file for managing all hyperparameters, paths, and experimental settings.

`README.md` The main project README file with an overview and basic instructions.

A.4 Running the Code

The project is packaged using `setup.py`, which creates convenient command-line entry points for all major scripts. After setting up the environment, the following commands become available system-wide:

```
generate      # For generating new problem instances
preprocess    # For any data preprocessing steps
train         # For training the MLP and REINFORCE models
train-sb3     # For training the final PPO model
evaluate      # For evaluating all solvers in a unified test
evaluate-sb3  # For dedicated evaluation of a PPO model
```

A.4.1 Data Generation

To generate new datasets for the Knapsack Problem:

1. Modify the `data_gen` section of `config.yaml` to specify the desired characteristics of the problem instances (e.g., number of items, correlation type).
2. Ensure the output paths in the ‘paths‘ section of `config.yaml` are set correctly.
3. Run the command: `generate`

A.4.2 Model Training

The framework supports training three different neural architectures. All hyperparameters and settings are managed through the `config.yaml` file.

Training the Transformer-PPO Model (Primary Contribution)

This is the primary model presented in this dissertation.

1. Navigate to the `ml.rl.ppo` section in `config.yaml`.
2. Adjust the model architecture and training hyperparameters as required. This includes settings for the Transformer encoder (`embedding_dim`, `nhead`, `num_layers`), the PPO algorithm (`n_steps`, `gamma`, `clip_range`), and the learning rate schedule.
3. Verify that the `data_training` and `data_validation` paths are correct.
4. Execute the training script using the command: `train-sb3 --name <YourExperimentName>`

Training the MLP Model

1. Navigate to the `ml.dnn` section in `config.yaml`.
2. Configure the model and training hyperparameters, such as `total_epochs`, `batch_size`, and `learning_rate`.
3. Verify that the data paths are correct.
4. Execute the training script using the command: `train` (ensure `training_mode` is set to `dnn` in the config).

Training the Pointer Network (REINFORCE / Actor-Critic)

1. Navigate to the `ml.rl` section in `config.yaml`.
2. Select the desired training algorithm by setting `training_mode` to either `"reinforce"` (for EMA baseline) or `"actor_critic"` (for critic baseline).
3. Configure the hyperparameters under the corresponding section (`reinforce` or `actor_critic`).
4. Verify that the data paths are correct.
5. Execute the training script using the command: `train`

A.4.3 Model Evaluation

The framework provides two primary methods for evaluation.

Dedicated PPO Model Evaluation

To evaluate a trained PPO model against the Gurobi baseline on a test dataset:

1. Execute the command, providing the path to the directory where the trained model and its statistics are saved:
   ```
   evaluate-sb3 --run-dir /path/to/your/artifacts_sb3/training/ExperimentRun/
   ```
2. Results, including performance plots and a '.csv' file with detailed metrics, will be saved in a new timestamped directory within `artifacts_sb3/evaluation/`.

Unified Solver Evaluation

To compare multiple solvers (classical, MLP, Ptr-Net, PPO) simultaneously:

1. Open `src/utils/config_loader.py` and ensure the solvers you wish to test are active in the `ALGORITHM_REGISTRY`.
2. Execute the `evaluate` command, providing paths to the trained neural network models:
   ```
   evaluate --dnn-model-path <path> --rl-model-path <path> --ppo-run-dir
   <path>
   ```
3. Results are saved in the `artifacts/results` directory.