Optimizing LLVM Pass List using Reinforcement Learning

Yilin Sun
Magdalene College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the Computer Science Tripos, Part III

University of Cambridge
Computer Laboratory
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: ys512@cam.ac.uk

May 27, 2022

Declaration

I Yilin Sun of Magdalene College, being a candidate for the Computer Science Tripos, Part III, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11939[1]

Signed: Yilin Sun

Date: 27/05/22

[1]measured using the in-built word counter in Overleaf

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Code optimization [2, 3] refers to a program transformation process that aims to produce high performance code that runs faster and consumes fewer resources. Conventional code optimization is commonly based on human heuristics [4, 5] which requires significant expert knowledge and has the potential risk of suboptimal performance. In this project, I will present a framework that carries out code optimization in a fully-automated manner, implemented using a Reinforcement Learning approach [6]. The framework enables compiler users to conduct code optimization with ease, even without the expertise to do so manually.

In this report, we will present a detailed implementation of our framework, from environment setup to agent implementation. We will then present a set of results by optimizing a stable benchmark **CoreMark** [7] with our framework. Finally, based on the results, we analyze the effectiveness of various methods employed and conclude with possible future extensions.

We will begin by demonstrating to the reader that code optimization is indeed an interesting problem with academic significance. We will introduce some challenges which motivate us to carry out the investigations.

1.1 Code Optimization: Background and Challenges

Optimizing code is never an easy task as extensive analysis and transformations are involved at each stage. However, the emergence of **LLVM** [8], a compilation framework focused on modularity and compatibility, offers new insights and solutions. In particular, LLVM provides an **intermediate representation** (LLVM IR), that can be optimized using **passes** in a iterative manner. These passes can either come from a stable collection that is predefined by the LLVM framework or could be hand-written by the user, allowing for great flexibility.

However, such flexibility usually comes with a cost in usability as a vast amount of time and effort is needed to understand the underlying working principles of individual passes in order to use them effectively. Moreover, the situation is only more complex as the target to be optimized is often in the form of a bitcode or assembly file, neither of which is easy to understand from a human perspective. This naturally leads to a need to develop a general-purpose and stable pass pipeline for developers. In the LLVM case, the optimization flag `-O3` would be the most direct solution. It invokes the compiler with a predefined series of passes which has been tested to be generally effective.

The question, therefore, is whether the pass list used by `-O3` is truly the best we can get. The answer is clearly not. A close inspection of the passes used by O3 reveals over 200 passes in total are used and many are performed multiple times. This shows an opportunity for simplification as some passes might be redundant and performing them multiple times could lead to diminishing marginal utility.

Additionally, passes provided by the LLVM framework are of a much greater variety than those used by `-O3`. Inside the O3 pass list, only 73 distinct passes are used while at least 200 passes are registered in LLVM version 14.0.0. As newer versions of LLVM are released, only an increasing number of passes will be supported. In addition, LLVM gives users the ability to define their own passes. How should a pass writer know when to run his pass given the existing optimization pipeline is already enormously complex?

Finally, organizing the passes is a non-trivial task as the order they are applied is crucial. Passes could interact by enabling or disabling opportunities for one another. This leaves us with a combinatorial problem with an astronomical search space. Even a simple greedy search algorithm takes days to run with a pass pool of size 200. This means we need tools with a rich understanding of the environment and intelligent decision-making capabilities.

1.2 Motivation

Such challenges motivate us to search for an effective solution through the lens of Reinforcement Learning (RL) [6]. RL is a learning paradigm that concerns with how agents perceive and interact with the environment and take actions to maximize the cumulative reward. As code optimization is ultimately interactions with code using passes to yield the greatest run time improvements, the RL paradigm fits seamlessly with our task.

In summary, through this project, we aim to address some of the following problems:

- Redundancies and rigidity in the default `-O3` optimization pipeline could lead to suboptimal performance when facing programs of different forms and scales.
- Large size of the pass pool and the ordering of pass application leads to an exploding search space infeasible for brute-force search algorithms.
- Poor interpretability of intermediate representation of the code makes heuristics-based approaches ineffective in general as an insufficient amount of information

could be extracted for a rich understanding of the environment.

- The `-O3` framework lacks customizability. Pass writers often lack the knowledge of new passes' performance when integrated with the existing framework.

It is then our goal to make a framework that:

- executes passes in a clean and efficient manner to attain high performance of code in terms of short run time and high throughput
- is efficient to train and use in terms of time and computation resources and requires little or no supervision during training
- understands intermediate representations well and makes decisions intelligently

We choose LLVM as the underlying compilation framework for our investigations due to its flexibility and popularity. However, in theory, any compiler that supports manual code optimization could fit into our framework with minimal modification. This suggests that the depth and impact of our project are much more profound.

1.3 Work Carried Out

With such motivations in mind, we have carried out the following work:

- Formalized code optimization tasks in the RL setting
- Implemented an environment with a top-down systematic approach, in compliance with the OpenAI gym [9] standard
- Implemented agents to explore and learn from the environment defined above, covering a range of model-free and model-based approaches
- Implemented a benchmarking pipeline to evaluate the effectiveness of individual agents

We hope that through this project, we could devise a useful tool to help users of the LLVM framework to optimize their programs effectively without complete knowledge of the working principles behind it. The framework would also benefit compiler developers when writing new passes and developing new pass pipelines. If successful, this would bridge a significant gap between developers and users of compilation frameworks.

1.4 Related Work

This work is inspired by the work of Szymon Makula [10], where the author explores the possibility of optimizing LLVM pass lists using **Structured Bayesian Optimisation (SBO)** [11]. In his work, Makula showed that Bayesian Optimization methods are generally ineffective due to the large search space caused by the permutation of the potentially infinite pass list. At the end of his work, he proposed that a Reinforcement Learning approach might be more suitable.

An important contribution of Makula regarding this project is that he has implemented a patched version of **Clang** [12] - a widely used frontend for LLVM. It overrides the original code optimization phase to allow for manual specifications of the passes to run. A member of Dr Yoneki's group transformed the patched compiler into an RL environment, allowing dynamic interactions. Despite being theoretically sound, the framework suffers from practical flaws which would be further explored in Chapter 3. Given these issues, we have completely rebuilt the environment in a much more systematic and modular approach.

Finally, the idea of optimizing compilers with RL is not completely unexplored. In fact, in his 2008 paper [13], Mammadli explored the possibility of optimizing LLVM pass lists using the Deep Q Network (DQN) [14]. However, there are areas that Mammadli did not explore, such as using different agents or different environmental setups. This gives the potential to further develop the topic from different perspectives.

Chapter 2

Background

In this chapter, we are going to prepare the reader with a basic understanding of the problem. We are going to focus on two aspects, one being the LLVM infrastructure and the other being Reinforcement Learning.

2.1 LLVM Tools and Functionalities

The LLVM Project [8] is a collection of modular and reusable compiler and toolchain technologies. It started as a research project at the University of Illinois. As of today, it has a large user community and has been widely used in academic research. In this section, we will introduce some of the tools we used in our environment setup and familiarize the reader with common terms and concepts in compiler design.

2.1.1 LLVM and Clang

The compiler is a specialized program that translates source code written in a high-level language to machine instructions. In general, a compiler has to go through the following stages to accomplish such transformations:

1. Lexing: Source code to token stream
2. Parsing: Token stream to Abstract Syntax Tree (AST)
3. Intermediate Code generation: AST to Intermediate Representation (IR)
4. Optimization: IR to IR, this could be iterative and carried out multiple times
5. Code generation: IR to target machine code

For larger-scale projects, there is also a need to go through the linking stage to combine emitted binary objects into a single binary executable.

LLVM on its own is not a compiler. In fact, it is a compiler backend since it only provides functionality for the last two stages of compilation: optimization and machine code generation. LLVM does come with its own intermediate language and an LLVM IR file can take two forms:

- .ll file: a human-readable form containing LLVM assembly in ASCII text
- .bc file: a bitcode form containing encoded bit streams

However, a backend alone is not sufficient for a complete compilation. Therefore, LLVM offers a frontend - Clang [12], which compiles C source code into LLVM IR. Moreover, Clang is able to orchestrate the frontend preprocessing (lexing and parsing) and the backend optimizations and code generation altogether, producing the binary directly from the source code. The user also has the freedom to use the tool to output the intermediate IR bitcode (.bc) or assembly language file (.ll). The bash code in Listing 2.1 offers some examples for a better understanding of different flags and options used by Clang.

Listing 2.1: Clang compilation with flags

```bash
clang -S prog.c
# output prog.s: assembler source, ASCII text

clang -c prog.c
# output prog.o: ELF 64-bit LSB relocatable, x86-64

clang -S -emit-llvm prog.c
# output prog.ll: LLVM assembly language

clang -c -emit-llvm prog.c
# output prog.bc: LLVM IR bitcode
```

2.1.2 LLVM Passes, -O3 and Opt

The power of LLVM lies in its modular approach to code optimization. These optimizations are termed as “passes” and are implemented in an object-oriented hierarchical approach. Most passes inherit from common parent `ModulePass` or `FunctionPass` classes. Beyond the standard passes that are packaged in the compiler, LLVM gives its user the ability to write their own passes. In general, passes can be broadly classified into two categories:

- Analysis Passes: Passes that analyse the program and injects information into the IR to assist later passes. These passes could add runtime overheads and brings little improvement in code performance. However, they are critical in making the transform passes work efficiently, thus allowing the program to perform significantly better in subsequent optimizations.
- Transform Passes: Passes that transform the program structure, making changes

to the code while preserving its integrity. They usually bring direct performance improvements to the code. However, running transform passes without analysis passes could lead to minimal performance benefits. In some cases, a transform pass could also lead to a deterioration in performance. However, these temporary losses can usually be offset by corresponding transform passes that are intended to be applied immediately afterwards for maximum benefit.

Some of the passes used in our experiments are shown below together with their functionalities. In particular, the passes "-instcount" and "-simplifycfg" [4] would be particularly useful in our experiments. "-instcount" is an analysis pass that returns the count of LLVM instructions by name, allowing us to have a simplistic overview of the overall program complexity and properties. On the other hand, "-simplifycfg" is a transform pass that simplifies the control flow graph. On its own, it brings very little improvement to the program's performance. However, its true power is revealed when combined with transform passes that improve code structure at the cost of extra redundancies. Appropriate uses of "-simplifycfg" usually bring drastic improvements to the code.

Analysis Passes	
-aa	Alias Analysis
-instcount	Count Instructions by Type
-domtree	Dominator Tree Construction
-loops	Natural Loop Information
Transform Passes	
-simplifycfg	Simplify Control Flow Graph
-instcombine	Combine redundant instructions
-loop-simplify	Canonicalize natural loops
-dce	Dead Code Elimination

Table 2.1: Example LLVM Passes

As mentioned in the Introduction, the list of existing inbuilt passes is relatively complex. This makes it difficult for most developers to manually optimize their code. Therefore, LLVM and Clang offer a set of inbuilt pass lists, expressed as optimization levels O0, O1, O2 and O3. As the number increases, more optimizations are executed and the optimization process typically takes longer. In particular, O0 performs no optimization while O3 optimizes aggressively, significantly shortening the execution time. For our experiments, these two optimization levels serve as important baselines for the evaluation of agent performances.

In our experiment, we need to run a customized list of passes. Fortunately, to make individual passes accessible, therefore allowing the user to run a customized pass pipeline, LLVM offers a tool named **opt**. By simply running:

```
opt -pass1 -pass2 -pass3 ... [input file] -o [output file]
```

We would be able to apply a customized pass list `[pass1, pass2, pass3, ...]` in sequence to the input bitcode file and write the resulting code to the output file. This mirrors how one would take actions sequentially in a Reinforcement Learning environment. In fact, the **opt** tool plays a pivotal role in the environment construction.

2.1.3 Link Time Optimization (LTO)

While opt tool itself is convenient and powerful, getting from C source code to LLVM IR is still a non-trivial step. We have demonstrated previously in Section 2.1.1 that Clang has the ability to emit LLVM IR from single C files. However, in reality, we usually need to deal with projects containing multiple C programs. How then can we run a customized pass sequence for a C project?

A viable way would be to compile every source program to a bitcode and run the same sequence of optimizations on every bitcode. In the end, we simply compile all bitcode files into binaries and link them to obtain the executable. However, such an approach is cumbersome and requires the repeated creation of optimization threads which could be costly, especially when the number of files is large. What if we can shift the linking stage before optimization? If so, we essentially would be able to link individual bitcode files to form a larger, shared bitcode file for the project. The optimizer and code generator will then only have to deal with a single bitcode file. It is clearly much cleaner and more efficient.

Again, with extensive research, we do find the answer - a feature called Link Time Optimization (LTO) [15] provided by Clang. As the name suggests, the LTO feature is designed primarily for intermodular optimization during the link stage. However, one of its abilities is to allow linking on LLVM IR, which is exactly the functionality we want.

To enable LTO, the user needs to install LLVM with the gold plugin and run the following command:

```
clang -flto -fuse-ld="gold -Wl,-plugin-opt=emit-llvm"
     [bitcode 1] [bitcode 2] ... -o [output bitcode]
```

In particular, the "-fuse-ld" flag indicates that LTO should use the specified linker, in this case, 'gold -Wl,-plugin-opt=emit-llvm'. The flag -plugin-opt=emit-llvm indicates that the linker will output IR bitcode files during linking. Such a powerful feature not only allows us to manually compile and link bitcode files but also to automatically generate them from cmake projects, by setting the corresponding cmake flags. In fact, we successfully transformed the cmake project lepton into a single bitcode for experimentation (Section 2.1.4).

Finally, it is almost trivial to compile the optimized IR bitcode to binary as .bc suffix will be automatically recognised by Clang. By running `clang [bitcode file] -o [binary file]`, we would be able to obtain the executable.

2.1.4 An Example: Lepton

To summarize the technical details and allow the reader to have a better understanding of how we construct our pass pipeline, we will give a brief example with cmake project lepton [16].

To obtain an unoptimized IR bitcode for the lepton project, we invoke cmake with the settings in Section 2.1.4:

```
CC=clang CXX=clang++ cmake
-DCMake_C_Flags=
    "-O0 -XClang -disable-O0-optnone -flto"
-DCMake_CXX_Flags=
    "-O0 -XClang -disable-O0-optnone -flto"
-DCMake_EXE_LINKER_Flags=
    "-flto -fuse-ld=ld -Wl,-plugin-opt=emit-llvm" ..
```

The need to use the option "`-XClang -disable-O0-optnone`" here is slightly surprising but important. In fact, it is only through many rounds of trial and error that have we discovered that the default option `-O0` in later releases of Clang marks most functions with "optnone" flags, which prevents them to be optimized by most subsequent passes.

The commands above will produce a file called 'lepton'. However, the underlying content of the file is not a binary ELF (Executable and Link Format), but rather IR bitcode. For clarity, we will rename it to 'lepton.bc'. We can run any optimization we want on lepton.bc, in an iterative manner.

```
opt -instcombine -simplifycfg lepton.bc -o lepton.bc
opt -aa -loops -loop-unroll -loop-simplify -simplifycfg lepton.bc
   -o lepton.bc
```

Finally, to measure the performance of lepton, we could compile it to an executable and run it with some test data. Notice that linking with certain standard libraries is still necessary but we have successfully shifted the linking stage within the project ahead.

```
clang lepton.bc -o lepton -lstdc++ -lpthread -lm
time ./lepton data/example.jpg
```

2.2 Reinforcement Learning

Reinforcement learning [6] concerns with how intelligent agents make decisions through interactions with the environment. Two key elements of Reinforcement Learning tasks are the environment and agent. What differentiates Reinforcement Learning from classical Machine Learning is that the exploration of the environment is often combinatorial in nature, allowing us to have only limited coverage of the environment's behaviours. However, the dynamic nature of the RL environment also offers the advantage that we could exploit our policies to collect meaningful data to train our agents efficiently. This makes RL a unique and interesting area for research.

2.2.1 Environment

The environment of a Reinforcement Learning task often takes the form of a Markov Decision Process (MDP) [17], which could be formulated as a 4-tuple (S, A, P_a, R_a) and two special states (s_0, s_T). The definitions of these variables are as follows:

- S is the set of states called *state space*
- A is the set of actions called *action space*
- $P_a(s, s') = \Pr(s_{t+1} = s'|s_T = s, a_t = a)$ is the probability taking action a in state s at time t will make the state transition to s' at $t + 1$. In settings where transitions are fully deterministic:
$$P_a(s, s') = \begin{cases} 1, & s' = f(s) \\ 0, & \text{otherwise} \end{cases}$$
- $R_a(s, s')$ is the immediate reward after transition from s to s' with action a.

In addition, we also have two special states s_0 and s_T which are referred to as the initial state and terminal state. We start the process from the initial state and end when we reach the terminal state. At the terminal state, no more transitions are allowed to be made.

In exact correspondence with the mathematical definition, another set of terminologies we use more often in implementations and documentation defines the following:

- **Observation Space**: Observations reveal to the agent what the current state of the environment is like. (S)
- **Action Space**: Actions describe the allowed interactions of the agent with the environment. (A)
- **Step**: Step is a deterministic transition: $(s, a) \to s'$, describing the effect of an action a in a given state s. (P_a)
- **Reward**: Reward is a function $r(s, s', a)$ describing the immediate gain by taking action a under state s and transitioning to s'. (R_a)
- **Reset**: Reset is a function: $() \to s$. It takes the environment back to the original state, allowing experiments to be restarted. (s_0)
- **Termination**: Termination is implicitly infused inside the step function as every call to the step function should return information about whether the experiment has reached the end. (s_T)

2.2.2 Agent

Given the environment, a policy π is defined as a function $A \times S \to [0,1]$, where:

$$\pi(a,s) = \Pr(a_t = a | s_t = s)$$

An agent then is an entity whose goal is to learn an optimal policy π which maximizes the expected cumulative reward defined as: $R = \sum_{t=0}^{\infty} \gamma^t r_t$. The term $\gamma \in [0,1)$ here is commonly referred to as the discount rate. It reflects how events in the distant future are weighed against those in the immediate future.

Over the years, a range of learning algorithms has been developed, leveraging techniques in deep learning and neural networks. We are intentionally brief about their characteristics as we would elaborate on their detailed implementations again in Chapter 4.

Value function Methods

The class of value function methods refers to agents that learns a approximated value function in either of the two forms:

$$V(s) = \max_{\pi} \mathbb{E}[R|s,\pi] \quad \text{or} \quad Q(s,a) = \max_{\pi} \mathbb{E}[R|s,a,\pi]$$

The functions V and Q represent the estimation of the maximum expected to return from the current state s (and taking action a). If these functions could be approximated, the agent would be able to act greedily by picking the action that yields the maximum V value or Q value. The DQN algorithm [14] falls into this category as it approximates the Q function defined above using deep neural networks.

Direct Policy Search

An alternative method is to search policies directly in the space of all possible policies. In particular, a family of methods leverage the technique of policy gradient in their optimization.

To allow the use of policy gradients, we first need to parameterise our policies by a set of parameters θ. The end goal of any policy optimization is to maximize the expected return defined by:

$$J(\theta) = \mathbb{E}\Big(\sum_{k=0}^{N} \gamma^k r_k\Big)$$

where r_k is the reward gained at step k by using the policy parameterised by θ.

If we can somehow differentiate the expected return function $J(\theta)$ against parameters θ (i.e. $\nabla_\theta J$), we would be able to optimize the policy. Just like most optimization problems, to achieve the optimal θ that gives the highest J, we could use the gradient ascent methods [18] leveraging the policy gradient $\nabla_\theta J$.

The difficulty then lies in finding the policy gradient $\nabla_\theta J$ as we do not have a clear, analytical relation between J and θ. The common approach is therefore to estimate the gradient from historical interactions. PPO algorithm [19] is a classic example of policy gradient method where it uses an "advantage" function to approximate the policy gradient.

Model-based Approaches

Model-based agents refer to agents that first learn a model of the underlying environment, including how it transitions and produces a reward. The underlying model could take various forms and the main complexity in the training actually lies in training the model to be as precise as possible.

With an accurate description of the environment, embedded in the model. Model-based agents could then be trained almost entirely in the "dream environment" described by the model. This introduces a great advantage in terms of sample efficiency and training time since interactions with the real environment could be costly. This is especially the case in system environments where drawing samples via direct interactions introduce severe overhead.

However, a common issue with model-based approaches also arises from the possible imperfections in the underlying model. This is because the model may not be able to capture all characteristics of the real environment. As a result, an agent trained in an imperfect "dream" environment could potentially exploit such imperfections which eventually lead to suboptimal performance in the real environment.

In our experiment, we choose to use the World Model proposed by Ha in his 2018 paper [1]. The world model consists of three key modules - Vision (V), Memory (M) and Controller (C) with interesting interactions to model the underlying dynamic transitions of the environment. Such a model provides great generalizability and proved to be effective in complex gaming environments like DOOM and Car Racing.

Chapter 3

Environment

In this chapter, we will introduce the existing infrastructure and some of its limitations in practice. Learning from its flaws, we will formalize a new approach using the Reinforcement Learning paradigm. We would illustrate how we successfully automated the entire compilation process in a much more systematic approach and produced consistent run scores for each executable.

3.1 Existing Work and Limitations

As mentioned in Section 1.4, our project's starting point is based on Makula's work in 2017 [10]. He has constructed a patched version of Clang with the pipeline illustrated in Figure 3.1. The patch overrides the compiler backend to first parse the user-defined pass file into a list of pass objects and then execute them in sequence.

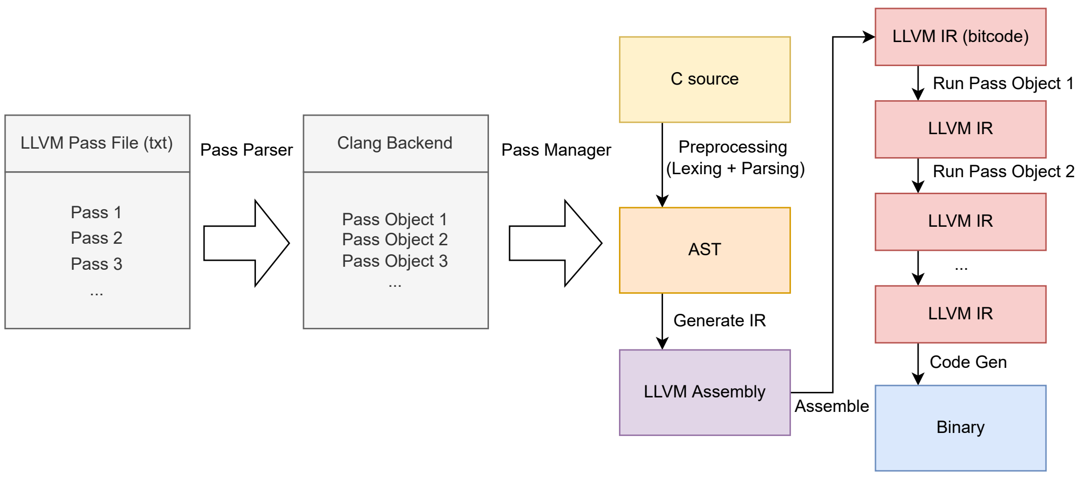


Figure 3.1: Illustration of Existing Patched Clang Pipeline

A member of Dr Yoneki's group subsequently transformed this pipeline into an RL environment as shown in Figure 3.2. To simulate a Markov Decision Process with state transitions, we iteratively add passes to the pass list and rebuild the program. To reset the environment to the starting state, we clear the pass file and rebuild the program without any optimization. While such an approach fits well with theory, it comes with serious limitations in practice. In particular, the environment suffers from numerous issues, including result reproducibility, maintenance difficulties and experiment scale.

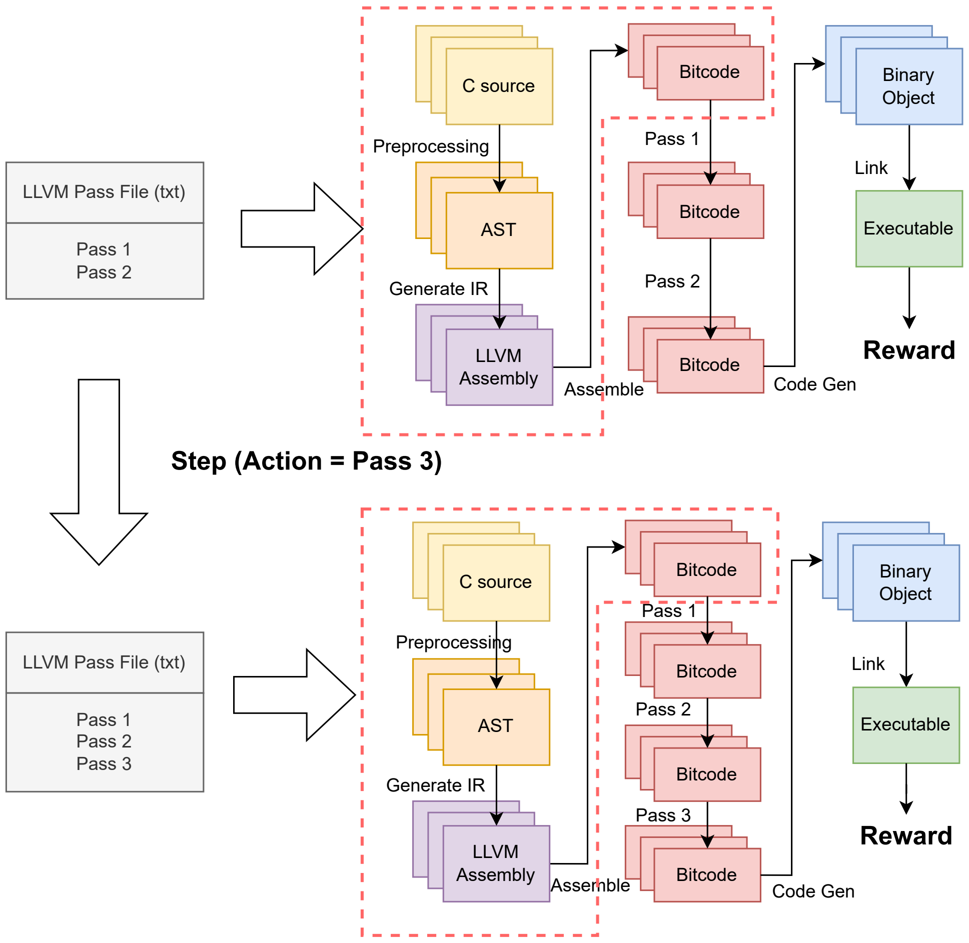


Figure 3.2: Illustration of Existing RL Framework

The first problem is the compatibility issue. The entire framework was implemented in 2017 which makes many of the tools involved outdated. In particular, the patch is targeted at LLVM 5.0.0 while the latest release was 14.0.0 (Mar 22). This makes the experiment hard to reproduce as time progresses since building legacy versions of LLVM requires compatible versions of C compilers and linkers. This also means that this pipeline has to be regularly maintained, especially with each new release of LLVM as the patch is hard-coded with specific line numbers. From a usability perspective, this pipeline does not allow registering of new passes as the pass parser is hand-written and new passes would not be recognized. The underlying cause for such problems lies in the fact that the pipeline does not leverage existing LLVM infrastructure and chooses to patch the code in a non-modular and hard-coded manner.

The second issue is related to efficiency. As introduced in Section 2.1.1, a compiler has to undergo different stages to transform source code to target machine code. However, as illustrated in Figure 3.2, every time a step is taken in the environment, a new pass is appended to the pass list. This implies that Clang has to be invoked to restart the entire compilation process. However, it is not difficult to see that the first few stages of the compilation: Lexing, Parsing and IR Generation (outlined in the red box) are independent of the passes to run. Such unnecessary repetitions introduce severe overhead when we interact with the environment frequently, especially given that at least tens of thousands of steps are needed in agent training.

3.2 Overview of the Improved Framework

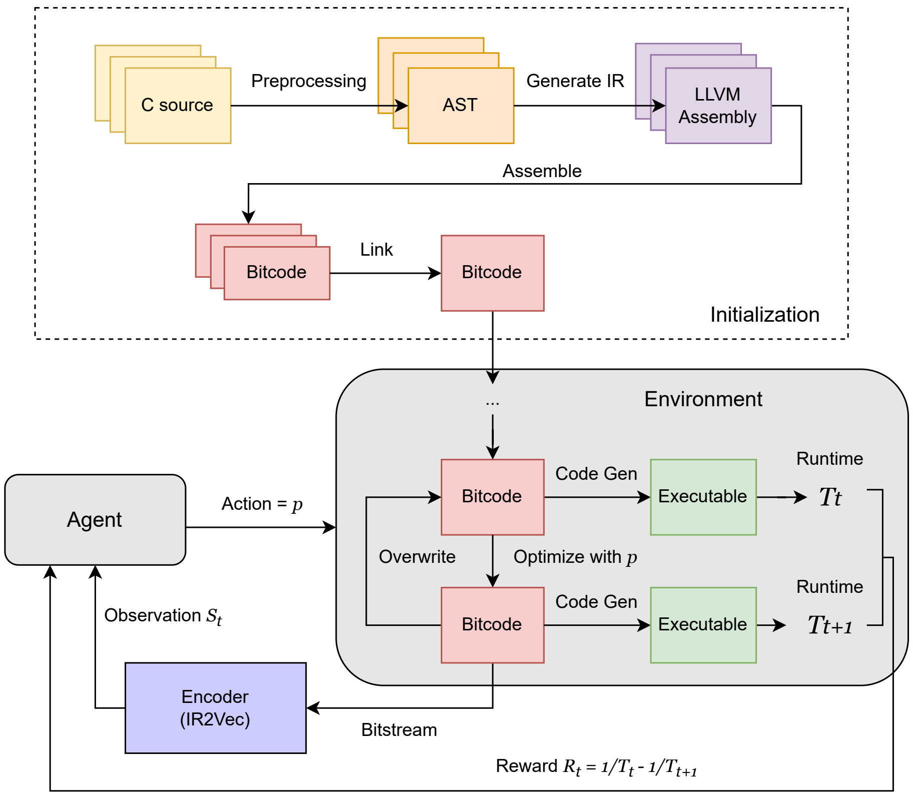


Figure 3.3: Illustration of Improved RL Framework

The limitations mentioned above suggest that we need a new framework focused on modularity and efficiency. Following the OpenAI gym standard [9] and using inbuilt LLVM tools, we have successfully implemented a new framework with a more systematic approach.

The pivotal part in the environment construction lies in the definition of two functions: `step` and `reset`. `step` encapsulates MDP state transitions or how execution of actions affects the underlying environment. In return, an agent should receive information such as the next observation, reward and termination. Interpreted in the code optimization setting, the step is simply running an additional pass on the code.

The reason why the old framework needs to recompile the program with each additional pass is because by default the intermediate representation of the code is not exposed to the user directly. Therefore, the key in making the `step` function efficient lies in the exposing and preservation of such intermediate states. As seen in Chapter 2, the solution is to invoke Clang with special flags and potentially use LTO to link multiple bitcode files altogether. At each step, we save the IR for the next iteration. This allows the framework to reuse the previous code states and avoid unnecessary repetitions of code preprocessing and IR generation.

The `reset` function will correspond to producing an IR file that is completely unoptimized. This could be achieved by calling Clang with the flag "-O0" and "-Xclang -disable-O0-optnone". If the project contains multiple source files, we could link them all together using the LTO feature.

Figure 3.3 provides a general overview of the new pipeline setup. The new pipeline design groups up all preprocessing steps so that only a single bitcode is being operated on throughout the experiment. To make the entire pipeline even more efficient, we could generate and save a copy of the clean, unoptimized bitcode during environment initialization, so that the preprocessing only needs to be run once globally. By doing so, reset is just a single copy operation while stepping is just a single optimization operation. This makes our new framework significantly faster to experiment with.

3.3 Formalization of Task

With a general idea of the environment setup, we will now formalize it using RL terminologies. As introduced in Section 2.2.1, classical Reinforcement Learning problems are defined by a tuple of functions and variables, namely: *observation*, *action*, *step*, *reset*, *reward* and *termination*. To formalize these notions in the LLVM settings, we need to first define mathematically some features of the code optimization system. In particular, we define the following variables and functions:

- $c_x \in C$: the source code for program x, in the space of all C source code C
- $b_x \in B$: LLVM IR bitcode for x, in the space of all LLVM IR bitcode B
- $e_x \in E$: executable for x, in the space of all binary executable E
- $P = \{p_1, p_2, ..., p_n\}$: list of possible passes we can run using opt
- $\mathsf{genIR}(c_x) = b_x$ $(c_x \in C, b_x \in B)$: compiling source code c_x to get bitcode b_X using LTO, without any optimization (O0)
- $\mathsf{comp}(b_x) = e_x$ $(b_x \in B, e_x \in E)$: compiling bitcode b_x to get executable e_x
- $\mathsf{runtime}(e_x)$ $(e_x \in E)$: measuring the run time of executable e_x
- $\mathsf{opt}(b_x, p) = b'_x$ $(b_x, b'_x \in B, p \in P)$: run pass p on bitcode b_x to obtain bitcode of the same program b'_x.

As such, we could define our RL task formally as follows:

1. **Observation**: The most ideal observation space would be the space of all possible IR bitcode B as they reflect all information of the program completely and faithfully at each stage of optimization. Following such definition, $b_x \in B$ would be a valid observation for program x at some stage.

2. **Action**: The action space would be the list of all valid passes P that could be successfully ran on any arbitrary bitcode in B.

3. **Step**: The step function corresponds perfectly with our opt operator defined earlier, in terms of both meaning and form. Thus taking a step with action p is the same as running $\mathsf{opt}(b_x, p)$.

4. **Reset**: Reset in the context of the LLVM Pass environment would mean to return the bitcode to the original, unoptimized state. As such, $\mathsf{reset}() = \mathsf{genIR}(c_x)$.

5. **Reward**: Reward function is based on run time. We will first compile the optimized bitcode to executable and then carry out measurements. We then take the difference between inverses of run time as our reward, i.e.

$$R(b_x, p) = \frac{1}{\mathsf{runtime}\Big(\mathsf{comp}\big(\mathsf{opt}(b_x, p)\big)\Big)} - \frac{1}{\mathsf{runtime}\Big(\mathsf{comp}\big(b_x\big)\Big)}$$

6. **Termination**: There is no terminal state in code optimization. This is because any intermediate representation is a valid state for future optimization. However, it is possible to add synthetic conditions for termination. For example, we could terminate all interactions after a fixed number of steps.

Although the environment is well-established theoretically, such definitions are difficult to implement in practice. Therefore, we will now discuss some practical challenges and methods to overcome them.

3.4 State Space and Code Encoding

In the theoretical model, we have defined states to be the IR bitcode and actions to be passes. However, bitcode is not only inconvenient to pass into the neural network during training, but also suffers from the issue that it has a variable length.

The usual technique of padding would not be possible here as programs could scale up to an arbitrary length. As such, some form of encoding has to be applied to ensure these bitcode programs could be transformed into a finite-dimensional vector. This simplifies the operations of the neural networks and allows us to inject prior knowledge of what is truly important to be observed by our agents.

One choice of encoding is to use the instruction count of the bitcode. This involves the use of the previously mentioned pass '-instcount'. By running:

Listing 3.1: Instruction Count

```
opt -enable-new-pm=0 -stats -instcount prog.bc
```

The opt tool will run through the entire bitcode and count occurrences of individual instructions and write them to standard output. By scanning the output with a regex filter, we would be able to obtain a dictionary mapping from the instruction name to the number of occurrences. We then select a reasonable set of instructions that reveal the most vital information about the program. Taking occurrence counts of these instructions would yield the program encoding.

In particular, across our experiments, we have chosen the following instructions for different purposes:

1. **Add**, **Mul**: Basic arithmetic operations
2. **Alloca**: Stack memory allocation (reveals information about variable assignment).
3. **Load**, **Store**: Main memory operations (crucial for memory optimizations)
4. **IComp**, **Br**: Comparisons and branching (reveal information about conditionals and loops, and thus the control flow)
5. **Call**: Function calls (important for function-related passes like inlining)

In total 8 types of instructions are being counted. This yields a 8-dimensional state space. However, although this method is simple to implement and interpret, we must recognise its inherent limitation as instruction count captures code structure very poorly. By using this encoding, we lose information on control flow, a defining characteristic of code quality.

Another choice of embedding is using a pre-trained model. In this case, we choose to experiment with the IR2vec encoding introduced by S. VenkataKeerthy [20] in 2019. It embeds an LLVM assembly file to a vector of length of 300. As this encoding was trained rigorously and primarily targets code optimization, we believe it could offer better insights into the code structure and hence be more likely to produce better results during training.

However, with code embedding, we clearly limit the information expressed by our states. Without sufficient information, the transitions $(s, a) \rightarrow s'$ might be difficult for our agents to learn from. To reveal more information for the purpose of effective training, we decide to append the action history, i.e. the passes used so far to the observations. This could potentially allow the agent to understand what passes are meant to be run in combination, thus making training slightly easier. In particular, adding the pass history to the state does not violate the Markovian property as given an action history in state s_t, the action history in state s_{t+1} by taking action a can be uniquely determined by appending a to s_t.

3.5 Action Space and Sanitization

The action space defined above seems relatively simple. However, there are hidden constraints we must recognise. In particular, we need to ensure any action a is a valid interaction with the environment at any state s. This implies that any pass p must be valid to run at any stage of the optimization. However, this is intrinsically difficult as not all passes are valid to be run on all programs. For example, passes like 'chr' (Control Height Reduction) and 'ir-similarity' will result in errors in the optimization phase while passes like 'memprof' (Memory Profiling) and 'asan-module' (Module Address Sanitizer) will run successfully but produce errors during binary generation.

To make the matter worse, certain optimization could take arbitrarily long to execute. For example passes like 'attributor' which orchestrates attribute deductions and 'dot-cfg' which constructs the control flow graph and writes it to .dot files, could be I/O intensive and takes a significant amount of time for large-scale projects.

To address these issues, we have two options:

1. Sanitize the pass list before conducting the experiment
2. Allow the running of all passes, but incur a penalty when passes fail

The advantage of the first method is that it shrinks the action space and reduces the risk of unexpected behaviour in the environment. Smaller action space usually means easier training as less exploration is needed. In addition, the stability of the environment implies less error handling and higher throughput since less time is wasted on trialling with invalid actions.

The advantage of the second approach is that it generalizes better as the agent will be eventually trained to mark these passes as 'invalid' or 'costly' and avoid running them unless necessary. This gives the costly passes a chance to be run. In addition, the environment should be prepared to handle the unexpected as manual sanitization could be error-prone and it is costly to test all combinations of passes for validity.

Although it seems that with sufficient training, the shortcomings of the second approach could be completely overcome, efficiency and training speed are still our main priority. As such, what we use is still primarily based on manual sanitation. We automated the sanitization process by running passes individually on a clean, unoptimized program. This is able to automatically eliminate most passes that are either invalid or costly.

In the actual implementation of the step function, we have prepared for the worst case by capturing the return code every time an optimization is run. We also set a time limit to avoid stalling of training from optimizations with high time costs. If a certain optimization fails or timeouts, we restore the code to the last clean state and incur a negative penalty to discourage rerunning of the pass. Note that the penalty should be carefully chosen since multiple experiments showed that a penalty that deviates too much from the average reward could lead to numerical instability and disrupt the entire experiment.

3.6 Reward

The reward function defined in the theoretical model uses the difference between the inverse of running time. The question is then why use the inverse and why use the difference? We will now explain the reasoning behind our choice.

3.6.1 Reward Function

The choice of our reward is not coincidental or trivial. The rationale behind lies in the fact that the run time does not reflect the code performance in a linear, uniform manner.

Assume we have a program that has a run time of 10 seconds. After some optimizations, its run time is reduced to 3 seconds. Considering the same absolute run time reduction of 1 second, clearly, the task is significantly more difficult for the optimized code as it has a much smaller room for improvement.

However, by taking the inverse of the run time, which could be commonly understood as the program's throughput, we find that the same absolute reduction in run time gets amplified accordingly. For the unoptimized code, the reward increased by only $1/9 - 1/10 = 1/90$ but for the optimized code, the increase is $1/2 - 1/3 = 1/6$, which is clearly accentuated. Although under both definitions, the reward should both be able to provide the correct signal for an agent, the choice of throughput as the reward would allow for faster convergence and better numerical stability.

However, why do we not use the throughput directly but take its difference between consecutive steps? This is because the final and ultimate goal of any RL agent is to maximize the cumulative reward from interaction. By choosing our reward to be:

$$R_t = 1/T_t - 1/T_{t-1}$$

the agent would be able to maximize the cumulative reward defined by:

$$\begin{aligned}\hat{R} &= \sum_{t=1}^{N} R_t = \sum_{t=1}^{N} (1/T_t - 1/T_{t-1}) \\ &= 1/T_1 - 1/T_0 + 1/T_2 - 1/T_1 + 1/T_3 - 1/T_2 + ... - T_N \\ &= 1/T_N - 1/T_0\end{aligned}$$

which precisely corresponds to the difference between the final throughput and the initial throughput!

An alternative choice of reward function is to use:

$$R_t = \log \frac{T_{t-1}}{T_t} \quad \text{or equivalently} \quad R_t = \log T_{t-1} - \log T_t$$

which leads to a cumulative reward of $\hat{R} = \log \frac{T_0}{T_N}$ as proposed in Mammadli's 2008 paper [13]. However, the intuition behind is very similar as a close inspection on the graph $y = 1/x$ and $y = -\log x$ reveals that they have similar shapes (Figure 3.4).

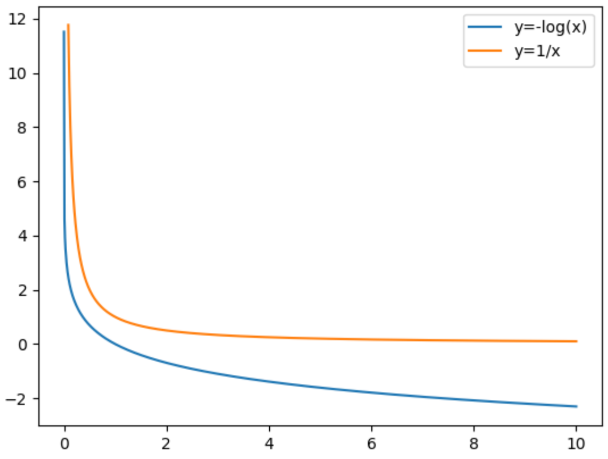


Figure 3.4: Graphs for $y = -\log(x)$ and $y = 1/x$

3.6.2 Choice of program and measurement metrics

Even with a well-designed reward function, another critical challenge to be addressed stems from reward measurement. Unlike in many other simulated environments (e.g. CartPole, Car Racing), where the reward is directly derived from mathematical calculations or simulations, the reward in the LLVM settings must be measured in real-time, which unavoidably suffers from noise and statistical variations. In addition, the run time depends heavily on the current system load and available computing resources (CPU, GPU, memory etc.) This imposes two critical challenges:

- How can we get a stable, consistent and statistically sound measurement of the reward? (i.e. low variance within the same system)
- How can we make the reward measured invariant or at least resistant to system load and other system-related parameters? (i.e. low variance across systems)

To address these problems, naturally, we would seek help from statistical tools. However, other than relying on statistics, an important but also easy-to-miss technique in improving the quality of the LLVM pass environment lies in program selection.

Our LLVM Pass environment is built around a chosen program. Therefore, its quality depends heavily on the underlying program we use. In our experiment, we deliberately choose to use programs that come from certified benchmarks, for the following reasons:

- These programs run consistently, giving run time that is more accurate than programs handwritten by average users.
- These programs output metrics that are better to use than raw run time measurements, which could be arbitrarily prolonged or shortened due to context switching. For example, in the *Himeno* benchmark [21], the metric Floating Point Operations Per Second (FLOPS) is used, giving a more direct measurement of performance.
- These programs have employed more accurate measurement methods, making the metrics obtained statistically more stable.

A list of programs successfully converted to RL environments in this project is shown below. Due to the fact that we frequently rely on metrics measured by the benchmarks instead of the raw run time, we will use the term **run score** throughout this report to refer to such metrics in general. For programs where only run time could be measured, we will use its inverse (i.e. throughput) as the run score.

Benchmark	Programs
PolyBench [22]	2mm, 3mm, adi, atax, bicg, cholesky, correlation, covariance, deriche, doitgen, durbin, fdtd-2d, gemm, gemver, gesummv, gramschmidt, head-3d, jacobi-1D, jacobi-2D, lu, ludcmp, mvt, nussinov, seidel, symm, syr2k, syrk, trisolv, trmm
Stanford [23]	Bublesort, FloatMM, IntMM, Oscar, Perm, Puzzle, Queens, Quicksort, RealMM, Towers, Treesort
CoreMark [7]	coremark
Dhrystone [24]	dhrystone
Himeno [21]	himenobmtxpa

Table 3.1: Benchmarks used as LLVM Environments

3.6.3 Error Reduction and Metric Normalization

Although these benchmarks are reasonably accurate by themselves, to ensure the stability of measurement, we execute the binary multiple times and take the average of run scores to reduce the standard deviation. In our experiments, we create 6 threads of execution simultaneously for each measurement of run score as 6 threads are the maximum the system can handle without competition of CPU resources. The execution is in parallel as sequential execution takes significantly longer and is highly inefficient for training. By repeated measurement of the run score, we are able to reduce the error from system noise.

To reduce the variance across systems, we choose to normalize the run score into a common scoring range. While the program may take longer or shorter to execute depending on the system specifications, the relationship between the run time and the system conditions is relatively predictable. In other words, if a program takes longer to execute in a given system, so are its optimized variants and other programs. Therefore by normalizing the run score against a stable baseline, we will be able to get similar scoring across different system environments.

An extra benefit of normalization is that different programs would also have similar scoring ranges. This allows for two very exciting possibilities mentioned in Chapter 6:

- Transfer the agent trained in one program to another.
- Train a single agent with multiple programs.

In our experiment, we introduced three metrics for normalization shown below:
(s_p - score for current program p, s'_p - normalized score for current program p,
s_{O0} - score for O0-optimized executable, s_{O3} - score for O3-optimized executable)

- O0 Normalization: $s'_p = s_p/s_{O0}$ (0 = maximally inefficient, 1 = O0)
- O3 Normalization: $s'_p = s_p/s_{O3}$ (0 = maximally inefficient, 1 = O3)
- O0-O3 difference Normalization: $s'_p = (s_p - s_{O0})/(s_p - s_{O3})$ (<0 = worse than O0, 0 = O0, 1 = O3, >1 = better than O3)

In most of our experiments, we choose to use the last metric, i.e. O0-O3 Normalization due to the fact that the value is usually contained in the range $[0, 1]$. The metric also gives a direct indication of performance against both baselines.

3.6.4 Reward Amplification and Denoising

Finally, we would introduce two transformations applied to the reward to assist training or stabilize measurements. These transformations act as utilities and are purely optional.

Amplification

The reward is simply a signal. It tells the agent how well it performs. However, the LLVM pass environment is not an easy environment to navigate through, especially when a large number of passes can lead to local minima that are hard to escape. Therefore, we could incentivise our agents by amplifying higher rewards while discounting lower rewards. This is achieved by exponentiation of the throughput terms in the reward function:

$$R_t = e^{1/T_t} - e^{1/T_{t-1}}$$

By applying the transform, we encourage the agent to reach a higher reward, even if this is at the cost that the intermediate steps yield poor results.

This fits very closely with our objective. In code optimization, intermediate bitcode files produced have absolutely no significance. In fact, we are only concerned about the final outcome. Hence, the agent should be prioritizing the high final return over poor intermediate rewards.

However, such amplification could also cause numerical instability as over-amplifying the reward can lead to large steps size when updating neural networks, resulting in oscillation without actual improvements.

Denoising

Finally, another reward transformation is denoising. We choose the simplest approach where reward within a certain range would be scaled down as follows:

Listing 3.2: Reward Denoising

```
def denoise(x, denoise_thres, strength=3):
    if abs(x) > denoise_thres:
        return x
    else:
        return (x / denoise_thres) ** strength * denoise_thres
```

The following graph (Figure 3.5) shows the effect of the denoising function with some choices of denoising threshold and strength. We see that the denoising strength controls how much the reward is tuned down while the threshold controls what we consider as random noise, or the range in which reward needs to be tuned.

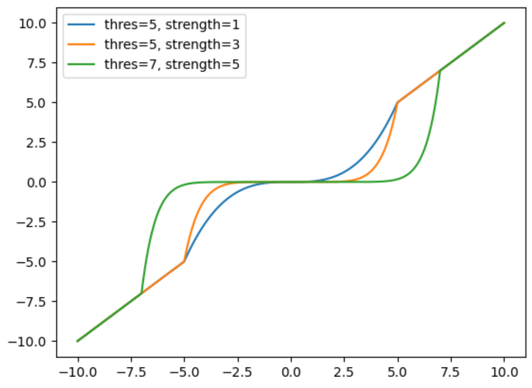


Figure 3.5: Denoising Function

By applying the denoising function to the reward, we avoid the issue of the agent being over-sensitive to the random noise present in the system since it only captures significant changes which are more meaningful.

Due to the difficulties in hyper-parameter tuning, the improvements brought by these two methods are limited. However, both are provided in the form of environment wrappers for maximal flexibility.

Chapter 4

Agent

We have implemented three agents covering two common model-free approaches and a slight variation of the model-based approach from *World Models* [1] by David Ha. In this chapter, we will go through some technical details on how we implemented these agents and the challenges we encountered.

4.1 Deep Q Network (DQN)

We started by implementing a simple DQN [14, 25, 26] agent due to the simplicity of its design. As mentioned in Section 2.2.2, DQN is a Value Function method. It learns a Q function which predicts the expected cumulative future reward and acts according to the action that maximizes the Q value.

The way to train a DQN could be expressed as Algorithm 1. However, in reality, we found that training in this manner could lead to the agent wasting many of the steps exploring deeply in the initial stages of the training. This is because our environment sets no explicit conditions to terminate. Therefore, we need to help our agent in deciding when to stop exploring and reset so that it would not waste time exploring states which are of little value.

The way we achieve this is to wrap the environment in a `TimeLimit` wrapper which limits the number of actions the agent is allowed to take before returning a synthetic `done=True` to reset the environment. Hence, we restrict the agent to explore shallowly at the start when its action is more random. As time progresses, we give the agent more freedom by allowing it to explore more deeply given that it has some basic understanding of the environment.

We have implemented our own DQN agent. The agent uses 2 hidden layers of size 64 each. Between each layer is a linear transformation followed by a rectifier. Such a simple model suffices to capture the main features of an optimal policy and adding additional layers to the neural network only complicates the Q function and leads to poorer outcomes.

Algorithm 1: DQN Agent

```
Initialize two networks, Q and Q';
Initialize empty replay buffer;
Reset environment and initialize oberservation;
for t=0,1,2,... do
    Calculate $\epsilon$ with linear decay;
    Pick random number $\theta \in [0, 1]$;
    if $\theta < \epsilon$ then
        Pick action randomly
    else
        Pick action = $\mathsf{argmax}_{a \in A}\ Q(S_t, a)$
    end
    /* Acting                                                    */
    Step in the environment with action and record current state $S_t$, next state $S_{t+1}$,
      reward $R_t$ and action $a_t$;
    Add $(S_t, S_{t+1}, R_t, a_t)$ to replay buffer;
    /* Training                                                  */
    Sample transitions from replay buffer;
    Calculate the loss by:
        $$\mathsf{Loss} = \Big(Q'(S_t, a_t) - R_t - \gamma * \max_{a_{t+1} \in A} Q(S_{t+1}, a_{t+1})\Big)^2$$
      Backpropagate loss and update Q with mini-batch SGD [18]
    /* Reset and update                                          */
    If done then reset environment;
    Every fixed number of iterations, assign $Q' = Q$;
end
```

4.2 Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [19] falls into the class of Direct Policy Search algorithms. It learns a distribution of actions based on individual states and acts by sampling the action directly from the distribution.

The way PPO is trained is via the so-called policy gradient. The term policy-gradient could be understood as to what extent an alternative policy is compared to the current policy. Just like in conventional gradient ascent methods where we wish to update the parameters iteratively in the direction that gives the maximal increase in output, i.e. the gradient, the situation is very similar here. In particular, we estimate the gradient using the expected advantage, defined as:

$$L(s, a, \theta_k, \theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s, a)$$

where the advantage $A^{\pi_{\theta_k}}(s, a)$ is known as the GAE (Generalized Advantage Estimator)

Algorithm 2: PPO Clip

```
Input  : Initial policy parameter $\theta_0$ (Actor)
         Initial value function parameter $\phi_0$ (Critic)
for $k$ = 0, 1, 2, ... do
    Collect trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
    Compute set of rewards-to-go $\hat{R}_t$.
    Compute advantage estimates, $\hat{A}_t$, using GAE based on the value function $V_{\theta_k}$.
    Update the policy (actor) by maximizing the PPO-Clip objective (with
      minibatch SGD [18]):
          $$\sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} L^{\mathsf{CLIP}}(s_t, a_t, \theta_k, \theta)$$
    Update the state value estimation (critic) by minimising the mse loss (with
      minibatch SGD):
          $$\sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t\right)^2$$
end
```

[27], which could be calculated as follows:

$$A_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+1} \quad \text{or} \quad A_t = \delta_t + \lambda\gamma A_{t+1} \text{ (recursive definition)}$$

Here γ and λ refer to the discount factor due to uncertainty in the future and the term δ_t is the TD advantage estimate for time step t. To find the TD advantage, we would use:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

This V function here is commonly referred to as the "critic" as it estimates the state value and hence gives an evaluation of the action taken by policy (known as the "actor"). With the advantages being estimated, we will then attempt to maximize the expected advantage L via back-propagation to update the policy (the "actor"). However, a common technique widely used in PPO training is clipping. This avoids the network being updated too aggressively. We do so by clipping our expected advantage to get:

$$L^{\mathsf{CLIP}}(s, a, \theta_k, \theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\theta_k}(s, a),\ \text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon\right) A^{\theta_k}(s, a)\right)$$

This essentially limits the ratio $\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}$ in the range $(-\infty, 1+\epsilon]$ when advantage is positive and $[1-\epsilon, \infty)$ when advantage is negative.

Meanwhile, the critic network also needs to be updated by back-propagating the mean square error between the measured reward-to-go and its predicted values. To update the actor and critic in one go, we use a compound loss consisting of both policy loss and critic loss as well as an entropy term to encourage exploration. In short, clipped PPO implemented by us could be summarized by the pseudocode in Algorithm 2 (modified from Stable Baselines 3 [28]).

4.3 World Model Agent

The term "model-based agent" as introduced in Chapter 2 refers to agents that learn a model of the environment and make decisions based on the learned model. The "model" learned by the agent should capture vital information about all the responses of the environment, namely how it transitions and how rewards are produced. Unlike how we have trained the model-free agents, where the agent learns the value function or the policy by directly interacting with the environment, the main complexity in the model-based approach lies primarily in learning the model through interactions. After the model is learned, the agent could learn to make decisions even without interactions with the environment at all. This is aptly termed "training inside a dream".

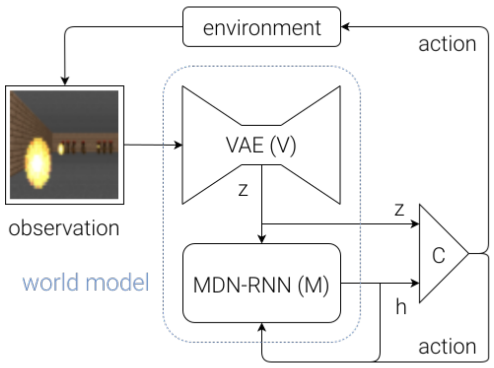


Figure 4.1: Illustration of the World Model (excerpted from [1])

In our implementation, we choose to reproduce and modify the world model agent proposed by David Ha [1]. The world model consists of three parts as shown in Figure 4.1:

- A visual module (V), used to encode the observation into fixed-length feature vectors in smaller dimensions, called *latent vectors*.
- A memory module (M), used to capture how the environment transitions from one state to another, along with rewards and termination conditions. It usually takes the form of an MDN-RNN, involving a *hidden state* being iteratively transformed.
- A controller (C), essentially an agent being trained in the "dream", controlling how decisions are made, based on *latent* and *hidden* vectors produced by V and M respectively.

The major difficulty in implementation lies in the M layer as it encapsulates the most critical information about the environment. In fact, we already have a V module in our toolbox - the 'ir2vec' encoding! Using ir2vec, we are able to encode our program into a feature vector of length 300, capturing most of the control flow information. This saves significant work in training a Variational Encoder (VAE) [29] as proposed in the original paper. Moreover, training a VAE to encode LLVM assembly code is extremely challenging, as neither do we have enough source code to cover a wide range of programs nor do we have the time and computation resources to train it.

As such, for the next sections, we will focus on how we implemented the M layer in the form of an MDN-RNN model and how the agent is trained.

4.3.1 MDN-RNN

An MDN-RNN, as the name suggests, is a combination of the Mixture Density Network (MDN) [30] and the Recurrent Neural Network (RNN) [31]. The RNN element allows capturing of the temporal dynamic behaviour of the system while the MDN produces observations, rewards and terminations in the form of distribution parameters. We will soon elaborate on what the distributions actually are and how the loss would be measured.

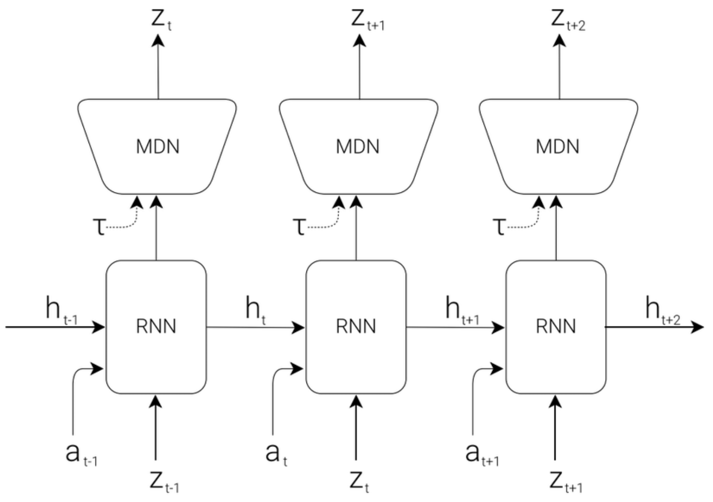


Figure 4.2: MDN-RNN

To start off, we implement the RNN by using a long short-term memory (LSTM) [32] network that takes the latent encoding z_t along with the action a_t as the input at each step and outputs a hidden vector h_t which is going to be reused at next round. Finally, the hidden vector h_t also serves as the input to the MDN.

In terms of its form, the MDN is similar to any other neural network. In our implementation, we just used a simple linear mapping. However, what makes MDN different is how its output is interpreted. Just as the name "mixture density" suggests, its output is parameters of a "mixture" of distributions which gives a probability "density" function of the variable of interest.

As proposed in the original paper, our MDN transforms the hidden vector output h_t of the RNN into parameters of a mixture of Gaussian distributions for latent prediction at the next timestep (z_{t+1}). For reward and termination, we extract directly from the output. However, with such a network, the problem is then how we could calculate its loss in terms of output values. This is because the ground-truths are fully deterministic measurements while the network output is parameters of distributions of these values.

One way is to derive a loss function, which when minimized, would maximize the log-likelihood of ground-truth observations. The loss calculation for reward and termination prediction is almost trivial, as they are extracted directly from network output. As such, we would use a Mean Square Error loss (MSE) and Binary Cross Entropy loss (BCE) respectively. The complexity lies in finding the loss from the latent prediction as a Gaussian Mixture Model (GMM) is used.

The GMM distribution can be interpreted as having a family of Gaussian distributions:

$$N(\mu_1, \sigma_1),\ N(\mu_2, \sigma_2), ...,\ N(\mu_n, \sigma_n)$$

along with a set of weights for each distribution:

$$\{\pi_1,\ \pi_2, ...,\ \pi_n\}, \text{ such that } \sum_{i=1}^{n} \pi_i = 1$$

The probability density function $f_{X \sim \mathsf{GMM}(\mu,\sigma,\pi)}(X = x)$ is then defined as:

$$f_{X \sim \mathsf{GMM}(\mu,\sigma,\pi)}(X = x) = \sum_{i=1}^{n} \Big(\pi_i * f_{X \sim N(\mu_i,\sigma_i)}(X = x)\Big)$$

This could be interpreted as: first random sampling which Gaussian Distribution X should be sampled from and computed the probability density that $X = x$ with the chosen Gaussian. To maximize the log-likelihood, we will define the loss as:

$$\mathsf{GMM}(\mu, \sigma, \pi, x) = -f_{X \sim \mathsf{GMM}(\mu,\sigma,\pi)}(X = x)$$

From this perspective, we see that the MDN should produce parameters as follows:

- $\mu = \{\mu_i\}, \sigma = \{\sigma_i\}$: means and standard deviations of the Gaussians in GMM
- $\pi = \{\pi_i\}$: weights of the Gaussians in GMM for latent
- r: direct prediction of reward
- t: logit prediction of termination

Assume we have the output as described above as well as ground-truth value of latent L, reward R, termination T collected from the environment, then we could compute the loss function computed as below:

$$\mathsf{Loss}(L, R, T, \mu, \sigma, \pi, r, t) = \mathsf{GMM}(\mu, \sigma, \pi, L)\ +\ \mathsf{MSE}(r, R) + \mathsf{BCE}(t, T)$$

where MSE and BCE are defined as:

$$\mathsf{MSE}(r, R) = (r - R)^2 \quad \text{and} \quad \mathsf{BCE}(t, T) = T * \log t + (1 - T) * \log(1 - t)$$

With the loss function clearly defined, we could use back-propagation with mini-batch SGD to update our MDN-RNN iteratively. This would be essentially how we train our Memory (M) layer in the model-based approach.

4.3.2 Training the Agent

With the Vision (V) and Memory (M) layers of our world model defined and implemented, we will have to first collect data to train the M layer and then train a Controller (C) inside the dream environment simulated by M.

To collect data for MDN-RNN training, we have to perform rollouts using the real environment. Using a random agent is clearly a viable choice since we are simply interested in how the environment transitions between states. However, we observe that a random agent usually gives very sub-optimal results compared to a trained agent. This means that a potential limitation with this data collection approach is that the rollouts collected are primarily located in the poor states while the better states are hardly explored. However, we want our model to be able to model all transitions well, regardless of what state it is in. In fact, we might even bias slightly more towards the good states as we want our agents to be constantly improving, meaning it would be spending more time exploring these better states.

As such, we employ a mixed data collection approach. We collect 70% of our data by first randomly exploring. We collect the reset by performing rollouts using our pretrained PPO agent. We used PPO agent trained with different hyper-parameters to cover a wider range of transitions. An additional benefit of this technique is that it couples closely with the training of the Controller (C), allowing us to test the choice of hyper-parameters more effectively by reducing the chance of transitioning to states out of our "comfort zone".

Finally, having dealt with the majority of technical difficulties, the only part left is to train the controller (C) in our dream environment. In our experiment, we train a controller using PPO. The difficult part is how we could leverage our MDN-RNN to simulate the dream environment. As mentioned before, we use the MDN-RNN to output parameters of distributions. Therefore, to generate the actual output, we would need to use random sampling. As a reward, there is no randomness in the output, allowing us to use it directly. For termination, we produce the variable by sampling from a logistic distribution. Finally, for latent, we assumed a Gaussian Mixture Model. To sample from the GMM, we first select the index of Gaussian to sample from (k) by sampling from a Categorical distribution parameterized by π. We then carry out the sampling of the latent vector from the Gaussian parameterized by (μ_k, σ_k).

Chapter 5

Evaluation

In this chapter, we will present experimental results collected by training various agents and evaluate their performances. We will first present our experimental setup, including the benchmarks used and the system environment. We will then look at scoring for some baseline agents as a foundation. Finally, we compare the performances of our trained agents against the baselines to evaluate their effectiveness.

5.1 Training and Evaluation Setup

We fix certain parameters in our environment so that all agents will be trained and evaluated in a levelled playground. In particular, we fix the number of actions carried out by the agent before termination to be 200. The number corresponds approximately to the total number of passes run in O3 optimizations. For the reward, we use difference between normalized run scores without scaling to train all agents. Finally, for model-free agents, namely DQN and PPO, we train for the same number of global steps (24,000) to ensure a stable performance and no advantage in terms of more training time.

For the model-based agent, we have collected around 200 complete rollouts and trained the MDN-RNN on the set. In theory, we would like to collect more data, but even 200 rollouts take close to 16 hours to be produced. Finally, We used a mini-batch size of 10 rollouts at each epoch and trained for 50 epochs.

In total, 51 rounds of training have been carried out. The average time for individual training takes around 8-10 hours for 24000 steps. The training and evaluation has been conducted on a laptop with the following specifications:

- CPU: 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
- RAM: 16.0 GB RAM
- GPU: NVIDIA GeForce RTX 3060

5.2 Benchmarks Used

In our training and evaluation, we have used CoreMark as our primary source of environment. However, we have also tested our agent on other programs, including the Himeno and Polybench benchmark, which gave similar results. We will give a brief description of the main benchmark used, i.e. CoreMark.

CoreMark [7] is a benchmark developed by EEMBC that measures the performance of microcontrollers and CPUs. Replacing the antiquated Dhrystone benchmark, Coremark contains implementations of the following algorithms: list processing (find and sort), matrix manipulation (common matrix operations), state machine (determine if an input stream contains valid numbers), and CRC (cyclic redundancy check).

To ensure compilers cannot pre-compute the results, every operation in the benchmark derives a value that is not available at compile time. Furthermore, the benchmark contains no library calls within its timed portion, allowing for consistent measurements.

5.3 Baselines

5.3.1 Baseline Agent Design

We have implemented three simple agents as our benchmark: a random agent, an agent that takes actions based on simple greedy heuristics and an agent that takes actions according to the O3 pass list.

The use of random agents as baselines for RL algorithms is pervasive. The idea behind is to sample randomly from the action space at each iteration. To allow for reproducibility of results, we seed the random number generator so that for a given seed, the action sequence produced by the agent is fully deterministic.

For the greedy "agent", it is not an RL agent in the strict sense as it makes use of the internal states of the environment other than the given functions “reset” and “step”. In particular, the environment has to support either duplication of states or step back of actions, in order to allow the trial of multiple actions in a given state. We will explain how a greedy agent functions with the pseudo-code in Algorithm 3.

To avoid the unnecessarily long exploration time and the unfair advantage introduced, we limit the number of actions trialled at each step to 30, out of our complete action space of size 200. This means the greedy agent still has a relatively significant chance to get a locally close-to-optimal action at each step.

Finally, O0 and O3 run times serve as important baselines. Since all run scores are normalized using O0-O3 difference, O0 will correspond to score 0 while O3 will correspond to 1. In addition, we have broken down the O3 pass list into individual steps so that it behaves like an agent. This allows for easy comparisons with trained agents.

Algorithm 3: Greedy Agent

```
Seed the random number generator;
Reset environment $E$ and set done = False;
while not done do
    Sample a fixed number of actions $A = a_1, a_2, ..., a_n$ from action space;
    for $a_i$ in $A$ do
        Try taking action $a_i$ in $E$ but do not step;
        This could be accomplished by either:
        1. Duplicate $E$'s state and step in the duplicated state, or
        2. Step with action $a_i$ and then step back;
        Record the reward $r_i$ associated with action $a_i$
    end
    Get the best action $a_{best}$ which gives the best reward $r_{best}$;
    Step with action $a_{best}$ and record the reward and done;
end
```

5.3.2 Baseline Performance

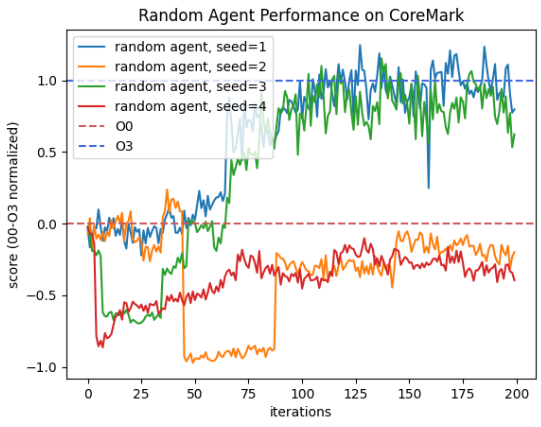
We have measured the performances of the baseline agents. Looking at the graphs (Figure 5.1), we are able to get some interesting insights.

Firstly, looking at the performances of the random agent (Figure 5.1a), we find that as expected, the agent has a large variation in terms of performance. For certain runs, it is able to get close to O3 performance, while for the others, the performance is significantly worse, reaching up to -0.5 in the end. This suggests that our environment is indeed very challenging as random exploration can lead to poor results.

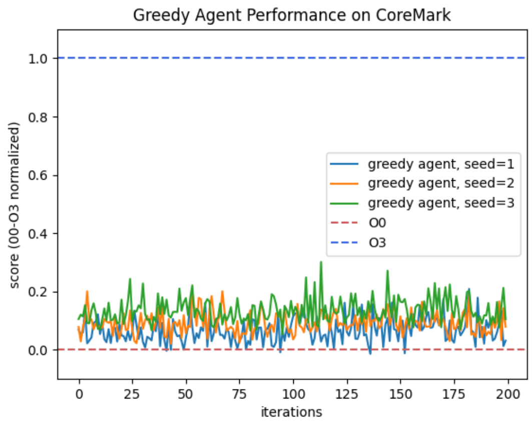
For the greedy agent (Figure 5.1b), it is mildly surprising that it performs suboptimally as no significant performance boost is observed. However, compared to the random agent, its performance is much more stable as it almost never leads to an intermediate state worse than the initial state. However, it is clear that the greedy agent is not able to extract the maximum potential from the code. To explain this, we will look at how O3 optimizations actually improve the code's performance.

Looking at the O3 agent (Figure 5.1c), the first thing observed is that the code performance is not always strictly increasing. In fact, at the start, the optimizations actually lead to a poorer state, followed by a series of significant improvements.

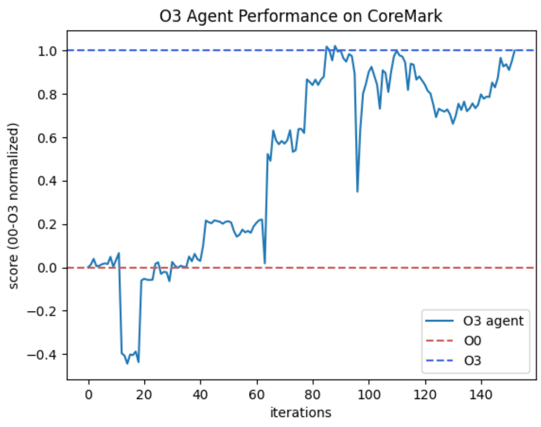
This phenomenon is common in code optimization. The reason behind is that certain optimizations do improve code quality, however, they do so by introducing significant redundancies and complexities into the code structure. Therefore, certain passes like '-simplifycfg' are necessary to be called regularly to maintain the code in a clean state. This means having a short-term loss can sometimes lead to a long-term gain in this context. This could be verified in Figure 5.1d, where all locations of '-simplifycfg' passes have been labelled. If we view the O3 optimization as phases separated by the '-simplifycfg's, we see that each optimization phase stably leads to performance improvements.



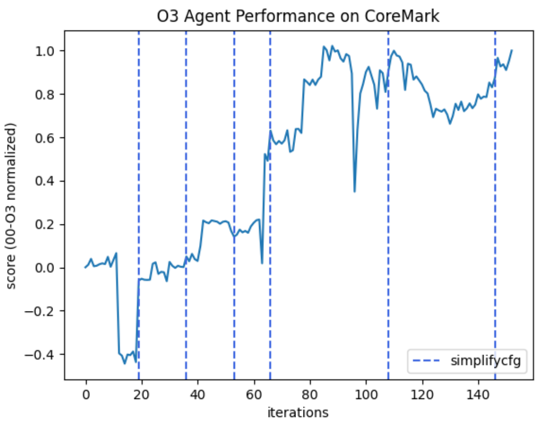
(a) Random Agent



(b) Greedy Agent



(c) O3 Agent



(d) O3 Agent ('simplifycfg' labelled)

Figure 5.1: Baseline Performance

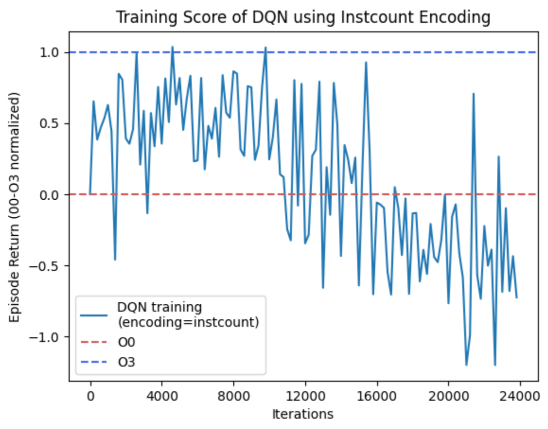
Such heuristics also explains why a greedy agent performs suboptimally. As demonstrated in fig. 5.1d, the major performance boosts are often preceded by a mild performance deterioration. Thus an intelligent agent should learn to sacrifice short-term rewards for long-term ones. However, by making our agent choose action greedily, we intentionally avoid these short-term sacrifices, making it never able to reach the optimal state.

5.4 Model-free Agents

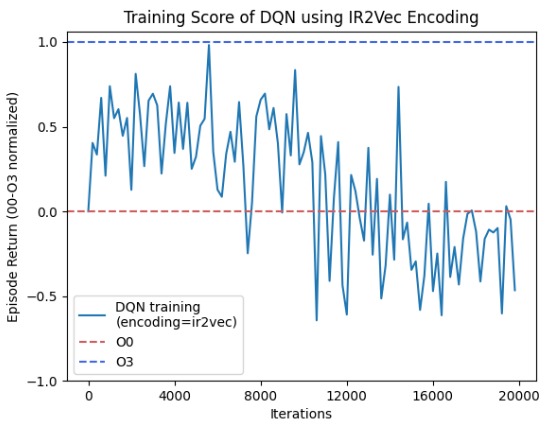
We have conducted numerous rounds of training for our DQN and PPO agents. In general, we find that the DQN agent is not able to learn well in our environment while a PPO agent is able to perform exceptionally well.

5.4.1 DQN

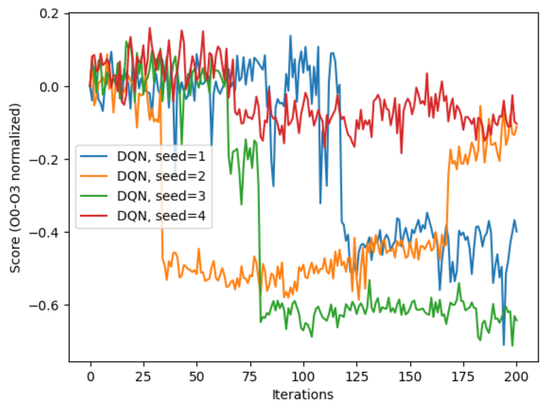
We have trained our DQN agents with over 20 sessions. However, its performance has not been up to expected. In particular, some of the training and evaluation statistics have been shown in Figure 5.2.



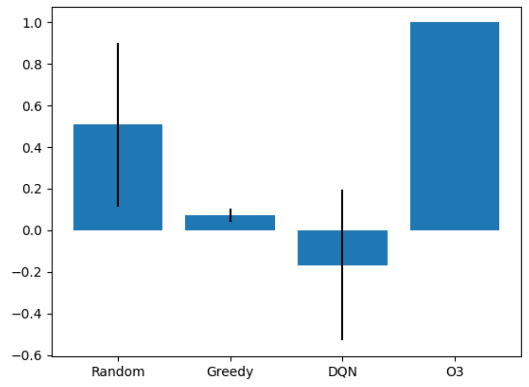
(a) DQN Agent Training with Instcount



(b) DQN Agent Training with IR2Vec



(c) DQN Agent Evaluation Score



(d) Comparison between DQN and Baselines

Figure 5.2: DQN Performance

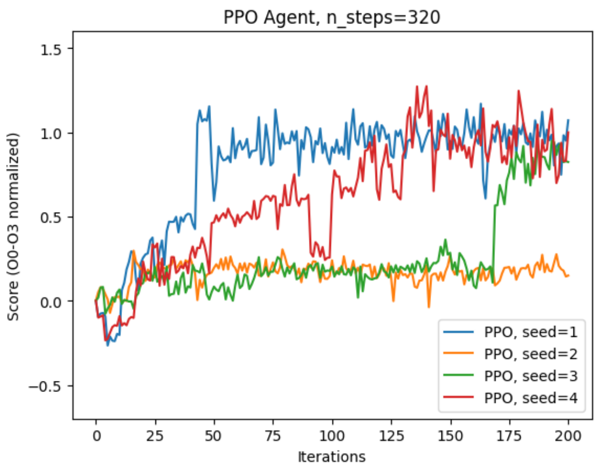
We attempt to use both types of code encodings: instruction count (Figure 5.2a) and ir2vec (Figure 5.2b). We have also attempted to shorten the update delay, increasing the learning rate or using exponential scaling to encourage risky behaviours. However, none of the above methods works effectively as there is no obvious sign of convergence near the end of each training session.

We suspect the reason behind this is due to the fact that the agent is under-trained since close inspections of the Q values of the agent reveal that they are very similar and rarely change based on the state. The Q value represents an expectation of the future return by the agent. Since the agent chooses its action greedily based on the Q value, it usually executes one action repeatedly for multiple iterations, which is extremely inefficient.

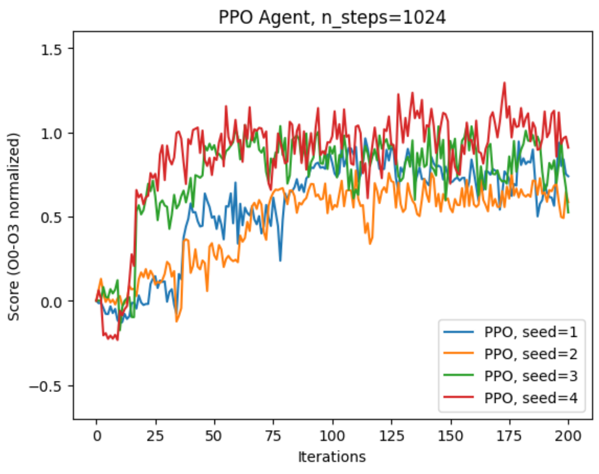
Such a phenomenon is often an indication that the training has been left in an intermediate state where Q values are just starting to converge and granularity is still being injected into the model. In fact, a brief overview of a related paper reveals that the author trained a DQN agent in a similar environment for over 800,000 global steps for it to converge. In contrast, with the existing computation resources, we can only afford to train the agent for around 24,000 steps, which would already take around 12 hours.

Running the under-trained agent for evaluation clearly yields very poor results (Figure 5.2c, Figure 5.2d), obtaining a average score of -0.16 and a standard deviation of 0.39. In fact, an undertrained DQN agent performs substantially worse than a random agent. We believe with the existing resources, it is infeasible to train a DQN agent effectively. To further analyze DQN's performance, we suggest running the experiment for a prolonged period of time and also with high-performance computing resources.

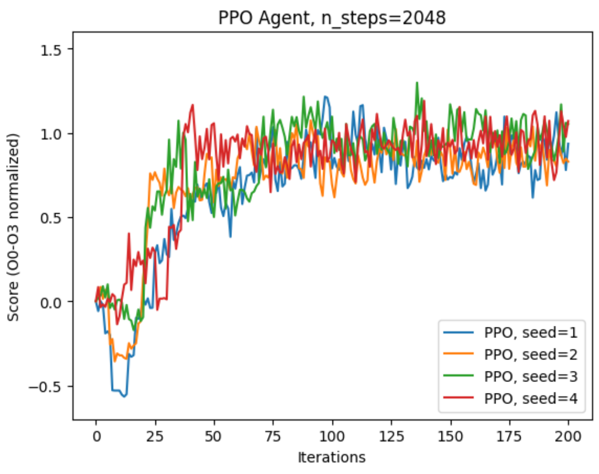
5.4.2 PPO



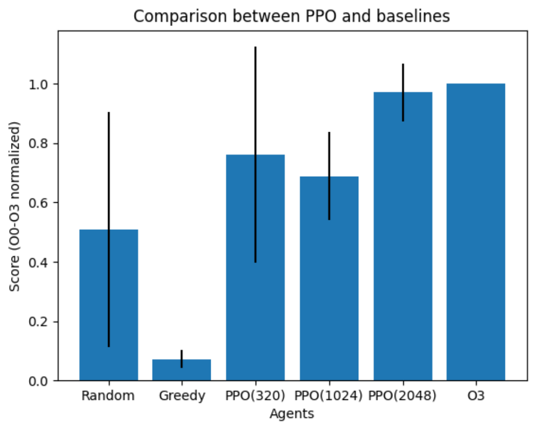
(a) PPO Agent (n_steps=320)



(b) PPO Agent (n_steps=1024)



(c) PPO Agent (n_steps=2048)



(d) Comparison between PPO and Baselines

Figure 5.3: PPO Performance

In contrast to the DQN agents, the PPO agents yield surprisingly great training outcomes. As the PPO agent is trained using the policy gradient update method, the update frequency is the main hyper-parameter to be tuned. We have trained three PPO agents, using different update delays (n_steps) of 320, 1024 and 2048.

Looking at Figure 5.3a, Figure 5.3b and Figure 5.3c, we can see that almost all PPO agents we trained showed promising results when evaluated. For PPO trained with an update delay of 320 (Figure 5.3a), the agent is able to reach a final performance very close to O3 when seeded with 1, 3, and 4. In particular, in the experiment with seed 1, the agent is even able to surpass O3 slightly by 7%, showing very promising potential. However, out of all PPO agents, the first one also performs unstably, yielding a score of 0.15 when seeded with 2.

For PPO agents trained with update delays of 1024 and 2048, the results are much more stable. In Figure 5.3c, we see that all evaluation runs are able to converge to close to O3 performance in 75 iterations. Two of the runs (seed=3 and 4) even yield a final performance 5% and 6% better than O3.

In general, we observe that with longer update delay and hence less frequent updates, the agent would enjoy greater stability in terms of the final run score. This trend is more clear as demonstrated in fig. 5.3d. As the update delay increases from 320 to 2048, we see that the mean score increased from 0.76 to 0.97 while the standard deviation decreased from 0.36 to 0.09. This provides some valuable insights for future training.

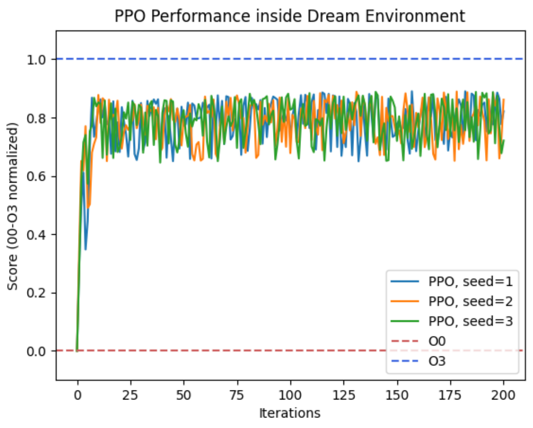
5.5 Model-based Agent

For the model-based agent, we have collected 200 rollouts with 160 from a random agent and 40 from the PPO agent. We use the rollouts to train the MDN-RNN with 50 epochs of training. We manually monitor the loss function at each epoch so that it does not suffer from underfitting or overfitting.

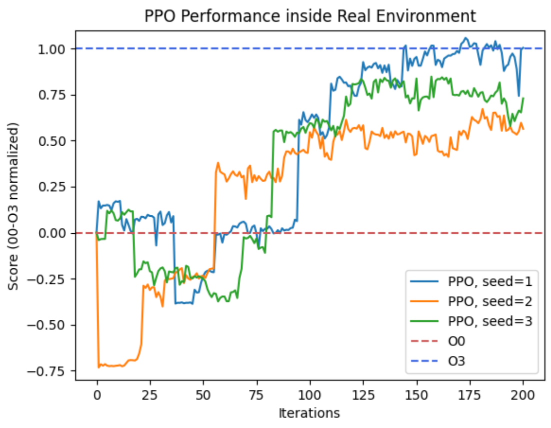
Since the "dream environment" using the trained MDN-RNN is extremely fast to interact with, we could train the PPO agent with a much larger number of global steps. We used 50,000 steps in total for training the PPO agent, with an update latency of 2048 steps.

Figure 5.4a and Figure 5.4b show the performance of our agent inside the dream environment and the real environment. As we can observe, the agent performs reasonably well in the dream environment, reaching a normalized score of over 0.8 in very early iterations and maintaining that advantage in later steps. In the real environment, although the agent takes longer to converge to the optimal score (0.73), it is able to stay in the state for an extended period of time, allowing it to outperform the random baseline consistently.

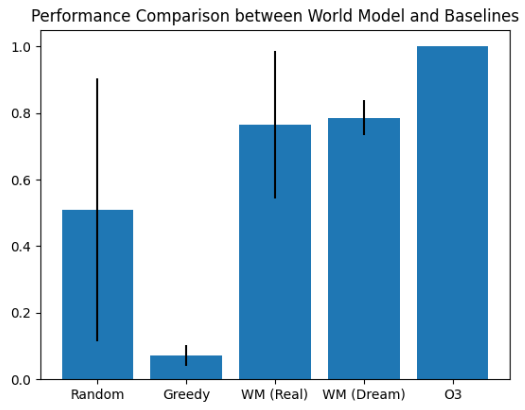
Due to the limited data we were able to collect, there are unavoidable gaps between the dream and the real environment. Therefore, although the PPO agent performs exceptionally stably in the dream environment, its performance is discounted in the real one. However, given that the performance is already much better and more stable than the baselines, we believe that the use of the World Model to address the problem of code optimizations has been a successful attempt.



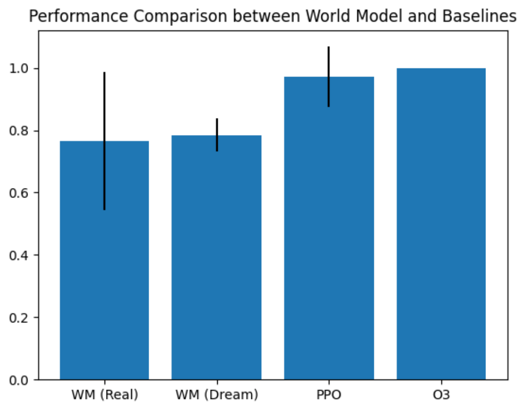
(a) World Model (Dream Environment)



(b) World Model (Real Environment)



(c) Comparison between WM and Baselines



(d) Comparison between WM and PPO

Figure 5.4: World Model Performance

5.6 Final Remarks

To summarize, the experiment results reveal both PPO and the World Model provide superior performance compared to the baselines, while the training of DQN has been unsuccessful. In particular, PPO offers better performance than the World Model as it offers faster convergence and smaller variance.

However, we believe that there is still much potential to be exploited from the DQN and World Model approach as the suboptimal performance of both could be mainly attributed to limited sample size. To further investigate their effectiveness, we propose using High-Performance Computing facilities to gather more samples. Another solution is to use programs of smaller size. However, an unavoidable issue with small programs is that their run time often suffers from more drastic fluctuations, increasing the difficulties of training.

Overall, we believe we have successfully produced a framework that runs optimizations in a fully automated fashion while guaranteeing close to optimal performance.

Chapter 6

Conclusion

In conclusion, this project has met all the success criteria mentioned in the beginning. In particular, we have carried out in-depth investigations on applying Reinforcement Learning to code optimization tasks. Throughout the process, we managed to convert the code optimization tools into a fully-automated, interactive RL framework.

Using the RL framework, we experimented with a selection of conventional and recent RL algorithms, covering a wide range of model-free and model-based approaches. By optimizing a stable benchmark program CoreMark, we are able to determine that the DQN agent was ineffective given the resource limitations while the PPO and World Model agent yield very promising results. We believe that Reinforcement Learning could indeed be an elegant and beautiful solution to the complex problem of code optimization.

An interesting area we wish to explore in the future is the generalizability of models. In particular, a generalizable RL agent should be able to transfer information gained from one environment to another. To accomplish such tasks, we have implemented an environment that supports learning with multiple programs. The environment either samples programs randomly at each reset or cycles through the choices in a round-robin style.

However, numerous challenges are associated with generalization. Firstly, it is still unclear whether training with multiple programs would need more time to converge as the complexity of the problem is further increased. Secondly, the selection of program training set is non-trivial as it should cover a broad range of complexities and structures. Even so, code could scale up to arbitrary complexity. How can we train the agent to not only be able to interpolate but also to extrapolate? Finally, in theory, model-based agents should be able to generalize better than model-free agents due to the richer understanding of the environment. However, whether that is indeed so is still open to question.

Overall, we believe there are still plenty of areas left to be explored. The challenge of a generalized RL framework would not only be interesting in the context to of code optimization but also may bring new insights to the RL research as a whole.

Bibliography

[1] David Ha and Jürgen Schmidhuber. World models. 2018.

[2] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, jan 1969.

[3] Timothy Jones. Optimizing compilers, 2021.

[4] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, jul 1987.

[5] Gary A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, page 194–206, New York, NY, USA, 1973. Association for Computing Machinery.

[6] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

[7] Markus Levy Shay Gal-On. Exploring coremark™ - a benchmark maximizing simplicity and efficacy.

[8] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Tech. Report UIUCDCS-R-2003-2380, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Sep 2003.

[9] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[10] Szymon Makula. Optimising clang compiler with auto-tuners leveraging structured bayesian optimisation. 2017.

[11] Valentin Dalibard, Michael Schaarschmidt, and Eiko Yoneki. Boat: Building auto-tuners with structured bayesian optimization. In *Proceedings of the 26th International Conference on World Wide Web*, pages 479–488, 2017.

[12] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, 2008.

[13] Rahim Mammadli, Ali Jannesari, and Felix Wolf. Static neural compiler optimization via deep reinforcement learning, 2020.

[14] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[15] Teresa Johnson, Mehdi Amini, and Xinliang David Li. Thinlto: scalable and incremental lto. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121. IEEE, 2017.

[16] Daniel Reiter Horn, Ken Elkabany, Chris Lesniewski-Laas, and Keith Winstein. The design, implementation, and deployment of a system to transparently compress hundreds of petabytes of image files for a file-storage service. 2017.

[17] Richard Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[18] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2017.

[19] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[20] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. Ir2vec: Llvm ir based scalable program embeddings. *ACM Trans. Archit. Code Optim.*, 17(4), dec 2020.

[21] Ryutaro Himeno. Himeno benchmark, 2010.

[22] Matthias Reisinger. Polybench/c, the polyhedral benchmark suite, 2015.

[23] LLVM. Llvm test suite.

[24] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[25] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. Dueling network architectures for deep reinforcement learning, 2015.

[26] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning, 2015.

[27] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2015.

[28] Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 2021.

[29] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2013.

[30] Christopher M Bishop. Mixture density networks. 1994.

[31] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[32] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.