CSC 461: Machine Learning

Deep Reinforcement Learning Applied to Compiler Optimization With LLVM Intermediate Representation

Team Members:

Raymond Turrisi	Mechanical Egr.ing & Computer Science
Alfred Timperley	Computer Science & Math
Brennan Richards	Data Science & Entrepreneurship

University of Rhode Island

Department of Computer Science and Statistics

Project Report

Fall 2021

Contents

1 Introduction

Compilers are powerful tools. They allow programmers to build software at a high level of abstraction by translating high-level semantics to low-level machine code. Not only do they serve as a translator, but they also mathematically evaluate how code can be optimized without changing the semantics of the program. Compilers such as Clang from the LLVM project apply optimizations from a discrete set of "flags" one after another in an attempt to optimize the program. Because one optimization flag is applied at a time, changing the code before the next optimization step can be applied, the NP-hard problem remains of picking a sequence of flags which maximizes the positive effects of the optimization pass on the final, compiled code ("phase ordering"). Common optimization goals include speeding up the compiled code's run-time, and/or reducing its size in memory.

Finding better approximate solutions for the phase ordering problem has important side effects in high performance and cloud computing centers, where even a small improvement in code size or run time translates to significant monetary savings. We have speculated that this is what has drawn Facebook/Meta to this problem, as we would expect a metaverse to require no shortage of computation.

In our project, we employ deep reinforcement learning (deep RL) methods in an attempt to find sequences of LLVM optimization flags that reduce the size of the compiled code – one of the simplest formulations of the phase ordering of problem.

2 Related Work

Compiler Optimization and specifically the phase ordering problem has been a large area of computer science research for decades. Before the deep learning revolution, the majority of solution involved expert tuned and developed heuristic algorithms [1]. However, now, most research is centered around using machine learning techniques for compiler optimization, such as Deep Reinforcement Learning.

2.1 AutoPhase: Applying Random Forest Towards the Phase Ordering

One of the most popular papers to come out of the machine learning approach to the phase ordering problem is AutoPhase [2]. In a paper published in 2020 they proposed a method using random forests to analyze the relationships between program features and optimization passes. They then used that method to reduce the search space and achieved 28% better performance than compiling with -O3. Another interesting challenge they identified in the phase ordering problem, is how in some steps of the ordering of optimizations, concrete sequences of optimizations may greatly improve performance when applied initially, however if reintroduced to the order later in the sequence after other optimizations have been applied, would be very harmful towards the objective [2]. This raises the question in our project, where current deep reinforcement learning methods predict the future, and ignore the past (in most cases). They estimate the value function for transitory states, and evaluate the next best action transitioning between states. When applying deep RL to this problem, if we do not consider previous states and conditions, actions with high-value rewards may be over emphasized and used at the wrong time in the sequence, ultimately being unstable in the application.

2.2 Compiler Gym

This project stemmed from recent research in the application of machine learning towards compiler optimizations. For our project, we used a platform called *Compiler Gym*, which wraps the LLVM compiler in a runtime compilation session in which individual optimizations can be applied, and then the current state of the code can be reduced to some abstract set of observations. The lead engineer on this project, Chris Cummins, has a long history at the intersection of machine learning and compiler research dating back to his studies at the University of Edinburgh. For the following discussion, we will discuss an overview of this intersection in research, the challenges Chris Cummins and his Principal Investigator were primarily interested in for their early research citing their related work that led towards Compiler Gym, and then discuss the scope of current research.

2.3 Overview

Machine learning has been used in compiler optimization problems since the mid-1990's to address two core problems: choosing the right optimizations to apply for a given program, and then the order to apply them [3]. Compiler research has been slower than many other fields in the computer science discipline since there is a steep learning curve before being able to make meaningful contributions towards highly complex problems which are highly critical to computer infrastructure. In addition to these two challenges within the field of compilers, at the interface with machine learning, there is the challenge for representing the source code features abstractly to be processed with modern machine learning techniques. Traditionally, when a new compiled programming language would be created, it would have its own compiler, meaning it would have its own intermediate representation and optimization functions. This evidently would slow compiler *research* and higher level research challenges if the aforementioned problems would vary significantly between popular languages.

These challenges shine light on the advantages of representing most high level code in a standard intermediate representation (IR). One project which has growing popularity and has been tremendously successful to standardize compilers while improving their performance, is the LLVM project and associated compiler tools. LLVM is an open-sourced project that is supported by a large community and many big tech companies, including Apple, for how impactful the project has been towards their work. For example, the LLVM group developed Clang, which is the standard compiler used by Apple for compiling all C and C++ binaries which they deploy to their devices [4]. As demonstrated in Fig. 1, the design of the compiler is to accept an intermediate representation, in which this converted code can leverage a standardized set of high quality and well tested optimizations. The ability to convert this intermediate representation to a variety of target architectures becomes independent and much less work for language developers, where they instead need to make their compiler convert their language to LLVM's IR which has all the tools modern languages need to support, and the rest is taken care of.

Currently, LLVM supports many languages and high level compilers: Ada, C, C++, D, Delphi, Fortran, Haskell, Julia, Objective-C, Rust, Swift, and more [Wiki]. Altogether, this addresses the challenge for being able to represent a large variety of code in a standard intermediate representation, where methods for representing this IR abstractly can be simplified and be more conducive to be used with modern machine learning.

This leads us to the next topic, which are methods in code characterization. An objective for computer architects and compiler designers is to understand characteristics of the programs, so behavior can be better projected onto targeted hardware. Current techniques for code char-

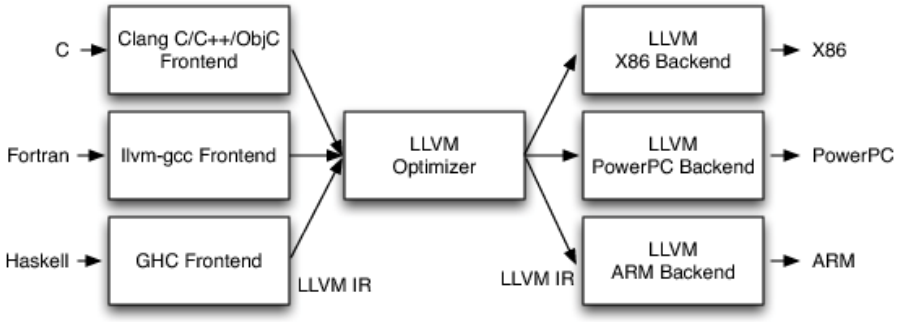


Figure 1: The LLVM compiler is meant to work with an intermediate representation, to work with a variety of languages and target architectures, while standardizing the optimization algorithms used in many different languages [src].

acterization revolve around producing tools which scan target code to extract different types of features in an abstract representation [3]. These tools are classified as static, dynamic, or hybrid classification methods. Dynamic and hybrid classification methods will not be discussed, since they were not used in our project.

Static classification methods include both source code features and graph-based features, and both are independent of run-time behavior. Static classification methods are rather broad, where these techniques cover linting tools, preliminary debugging tools such as those found in popular Integrated Development Environments (IDEs), and more. For their application in machine learning, these feature sets of source code analysis methods are mostly limited to counting specific types of instructions in the IR, or general observations. Graph-based features are much more flexible, and describe code behavior rather than just instruction counts. Graph kernels have become a popular field of research which allow complex relationships, sequences, and behavior in data to be described abstractly. Graph methods and graph kernels in compiler optimization problems allow code to be represented in a rich data structure which can be used to describe code similarity, control flow, and convey high level behavior to machine learning methods [3].

Another objective of this work is to reduce compiler development cost. As demonstrated in Fig. 2, there have been substantial investments in developing robust compilers for each of these languages. Much of these languages fulfill the same objectives, where LLVM is not only opening doors to novel research, but it is also has the ability to drastically reduce the cost and overhead for low level and complex features which are used in all compilers, so investments can instead be put towards novel objectives and generate more value. A great example of a language which has leveraged the power of LLVM, while introducing many novel features, is the Rust programming language. The Rust language is a system-level programming language while introducing many functional paradigms and zero cost abstractions. A core objective of Rust, is to address one of the largest problems that the most popular low level languages have faced in recent years, such as C and C++, which is the lack of memory safety [Wiki]. With the advancements of LLVM, Rust has been able to mature efficiently since its conception, using the underlying LLVM compiler infrastructure to produce code with comparable performance to C and C++, while introducing new features which guarantee memory safety, by introducing many new ideas which prevent run-time errors related to poor memory management, addressing problems which are often associated in billions of dollars in damages since conception. All of this, is also achieved without the use of a garbage collector, which introduces unpredictable

runtime behavior which is not appropriate for sophisticated control system implementations.

Project	Started	Developers	LOC	Estimated Cost
FreePascal	2005	54	3,845,685	$198,619,187
GCC 9.2.0	1988	617	5,591,759	$425,747,278
Glasgow Haskell Compiler 8.8.1	2001	1,119	761,097	$52,449,098
Intel Graphics Compiler 1.0.10	2018	149	684,688	$46,934,626
LLVM 8.0.1	2001	1,210	6,887,489	$529,895,190
OpenJDK 14+10	2007	883	7,955,827	$616,517,786
Roslyn .NET 16.2	2014	496	2,705,092	$198,619,187
Rust 1.37.0	2010	2,737	852,877	$59,109,425
Swift	2010	857	665,238	$45,535,689
v8 7.8.112	2008	736	3,048,793	$225,195,832

Figure 2: Estimated cost for developing several compilers [5].

Altogether, these techniques in analyzing code behavior based on an abstract intermediate representation, have setup new fields of research, including the application of machine learning towards the phase ordering problem. Ultimately, since compiler research is gravitating towards open-sourced tools which fulfill much of the lower level compiler behavior which is synonymous to all languages, higher level challenges can be pursued which will generalize to more programming languages and target architectures.

2.4 Machine Learning towards Compiler Research

Traditionally, discovering the best order to apply optimizations has been an arduous and labor intensive process. Compiler Engineers would need to explore the solution space, gather statistical evidence on changes in performance, and the tuning for the individual methods involved various heuristics which could have benefited from the application of machine learning [5]. It has been believed that the use of machine learning can be used to solve a variety of problems in compiler research. For example, machine learning can help digest a large parameter space when optimizations are dependent on the program being compiled and its intrinsic features, but also target architecture such as the number of registers and cache size. For this previous example, machine learning can be used to fine tune the underlying optimization algorithm, or to learn how to apply a specific optimization algorithm to a program to achieve the best results. For the latter, this would have to be determined at compile time on a case by case basis.

For many other applications of machine learning in compilers, there is another challenge in determining which type of model to apply. For most of the challenges, including the subject of phase ordering, there is no one-size-fits-all approach, since the input features can vary drastically across different types of programs for both source code analysis and graph kernel input spaces [5]. This suggests that models should be able to generalize well in order to learn the overall heuristics for these challenges, where some experts strongly believe that this means high-level compiler behavior is a well suited problem for deep reinforcement learning techniques [6]. Dr.

Hugh Leathers, Chris Cummin's advisor, argues that research should fit compiler optimization problems into a *Machine Learning Mold*, meaning that these problems need to be designed and resources should be allocated in order to create the space where machine learning paradigms can do well [6]. This argues identifying how problems should be be learned, and identifying the best techniques for representing a programs features. For example, reinforcement learning would be more well suited to learn the heuristics of optimization functions and their associated weights given a set of input conditions rather than learning *the best* set of optimizations that should be applied for any given program [6].

2.4.1 Compiler Gym

Thus far we have discussed challenges and solutions in compiler design; their development cost, scarcity of expertise, and then the unique challenges machine learning techniques face when tackling these problems. The culmination of Dr. Chris Cummins and Dr. Hugh Leather's research, has led towards the development of a new framework at Facebook Research, *Compiler Gym*, which seeks to broaden the participation of research of machine learning towards compiler optimization. This framework models OpenAI's popular *gym* environment interface which is well suited for reinforcement learning problems, while introducing a plethora of back end features which address most of the aforementioned challenges.

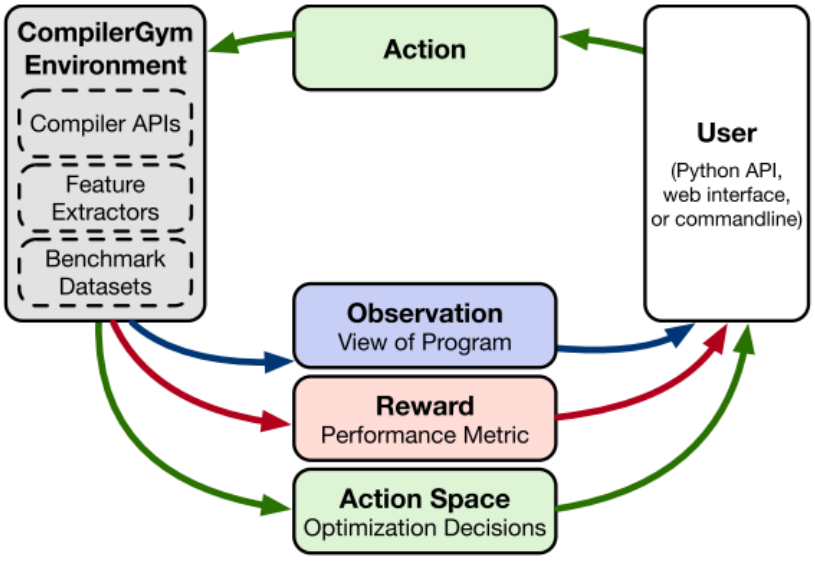


Figure 3: The CompilerGym interaction loop. A CompilerGym environment exposes an observation, reward, and action space. The user's goal is to select the action that will lead to the greatest cumulative reward. This may be through hand-crafted heuristics, search, supervised machine learning, or reinforcement learning [7].

This framework offers an end-to-end package with many features, which will be discussed more in-depth in Section 3. However, while this project is still in early development, it has already proved quite promising, even with the small number of participants. This project has already demonstrated impactful results towards understanding how reinforcement learning techniques can fit into compiler optimization problems. As demonstrated in Fig. 4, four popular deep reinforcement learning algorithms are compared with the standard `-Oz` LLVM optimization macro. A number greater than one is good, where it is expressed as the ratio of the applied

optimizations compared to the `-Oz` macro. Here, we can see that standard and non-specialized models when tested on a variety of datasets can outperform a method which was derived analytically. While it has not demonstrated total superiority, it shows that it can be done, and that there is opportunity for more research in developing novel machine learning models, use of existing models, and methods of abstracting intermediate representation.

	Geomean code size reduction			
Test Dataset	A2C [40]	APEX [41]	IMPALA [42]	PPO [43]
AnghaBench [16]	0.951×	0.659×	**0.958×**	0.776×
BLAS [17]	0.928×	**0.934×**	0.861×	0.906×
cBench [15]	0.804×	0.698×	0.814×	**0.964×**
CHStone [18]	0.823×	0.704×	0.707×	**1.014×**
CLgen [19]	**0.950×**	0.687×	0.916×	0.843×
Csmith [8]	1.023×	0.692×	1.144×	**1.245×**
GitHub [14]	0.975×	**0.987×**	0.976×	0.984×
Linux kernel	0.987×	**0.998×**	0.983×	0.995×
llvm-stress [11]	**0.838×**	0.493×	0.736×	0.097×
MiBench [20]	0.996×	0.996×	0.996×	**1.000×**
NPB [21]	**0.961×**	0.816×	0.958×	0.923×
OpenCV	0.976×	0.969×	**0.986×**	0.945×
POJ-104 [22]	0.778×	0.651×	**0.805×**	0.801×
TensorFlow [23]	0.976×	**0.976×**	0.966×	0.933×

Figure 4: Comparison of four reinforcement learning algorithms, comparing average code-size reduction with the standard **-Oz** compiler flag [7].

3 Methods

3.1 Tools

3.1.1 Stable-baselines3

Stable baselines is an open-source software package which features implementations of many of the state-of-the-art deep reinforcement learning algorithms. This library started as a fork of OpenAI's "baselines" library, which has had numerous complaints cited against it [8]. Working with the third iteration of this library, "stable-baselines3", we tested the *Deep Q-Network* (DQN), *Advantage Actor Critic* (A2C) and *Proximal Policy Optimization* (PPO) algorithms available in stable-baselines.

3.1.2 Optuna

Optuna is an "open source hyperparameter optimization framework to automate hyperparameter search" [9]. This framework provides a lightweight and flexible system for optimizing hyperparameters regardless of the library, and makes hyperparameter selection parallelizable. In the language of Optuna, a study is an encapsulation of some experiment, each with any number of trials, where hyperparameters are chosen and configured one time for each trial.

Optuna exposes a handful of helper methods which select parameters in specified ways such as selection based on a Gaussian distribution, categorical choice, choices between a range of values, and others. The package also intelligently "prunes" trials in which the model is performing poorly, so as to save time and computational expenditure. Optuna has a powerful hyperparam-

eter search tool, which statistically converges on hyperparameters which influence the result the greatest, and also descends on the optimal value for each of the hyperparameters in the search. The best method for using this platform, is to point the program to a remote database, in which it can use historic data between sessions, and also share data concurrently across different session instances (running programs). We leveraged this across at least four servers at any given time. For example, we would configure scripts in a git repository which was installed on these various servers, we would distribute the same software across the systems, and then have the programs running at five trials per session, six sessions on any given instance, across these four computers. In each of these six sessions, we would have the same script run continuously while we were investigating different techniques. The following figure is an illustration of how our tuning structure was implemented.

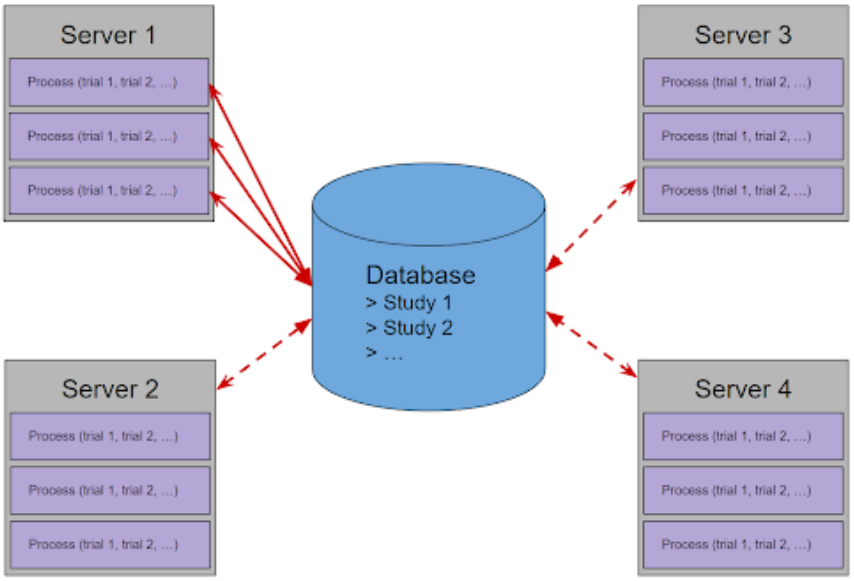


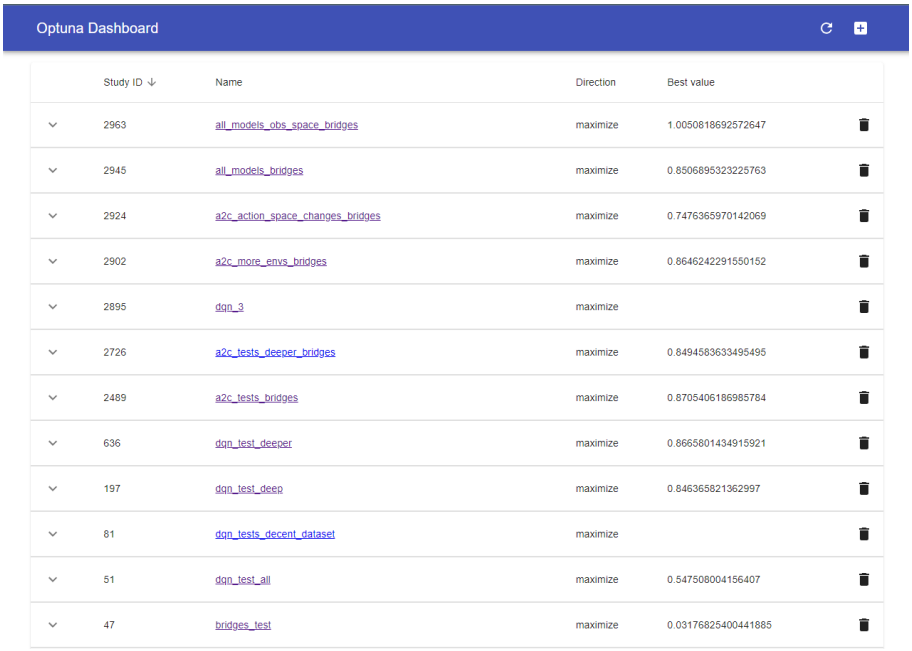
Figure 5: Optuna hyper-parameter search parallelization methods

Using the database also provides the benefit of the *Optuna dashboard* tool, which analyzes trial data to create a number of helpful plots, such as how correlated each hyperparameter is with the ultimate performance. For example, in Fig. 7, we demonstrate how Optuna analytically determines which hyperparameters greatest impact the result, where in the back-end, Optuna explores these spaces more granularly.

Below, we will walk through the user interface and explain how we used the tool more in-depth. When launching a dashboard session on localhost, we are presented with an overview of the studies which are pointed at this database. This is seen in Fig. 6, where we see the ID of the study (in the order in which they're started), our name for the study, our objective, and then the best value from the study.

When looking into the study, there is a lot of data available, however the most useful by far is the hyperparameter importance, in addition to the results of each of the trials. The hyperparameter importance is demonstrated in Fig. 7, and a sorted list of studies is presented in Fig. 8.

By automating the hyperparameter search process with a more robust method than conven-



Study ID	Name	Direction	Best value
2963	all_models_obs_space_bridges	maximize	1.0050818692572647
2945	all_models_bridges	maximize	0.8506895323225763
2924	a2c_action_space_changes_bridges	maximize	0.7476365970142069
2902	a2c_more_envs_bridges	maximize	0.8646242291550152
2895	dqn_3	maximize	
2726	a2c_tests_deeper_bridges	maximize	0.8494583633495495
2489	a2c_tests_bridges	maximize	0.8705406186985784
636	dqn_test_deeper	maximize	0.8665801434915921
197	dqn_test_deep	maximize	0.846365821362997
81	dqn_tests_decent_dataset	maximize	
51	dqn_test_all	maximize	0.547508004156407
47	bridges_test	maximize	0.03176825400441885

Figure 6: Optuna dashboard when linked to an external database.

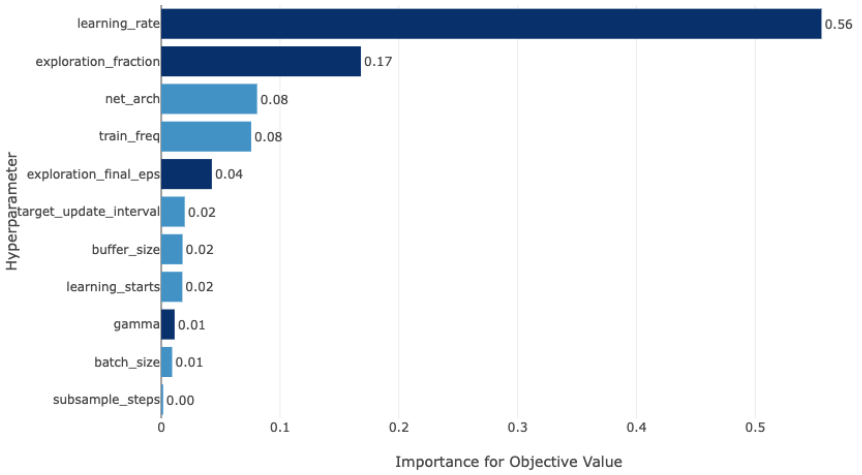


Figure 7: A chart of hyperparameter importance towards agent performance over more than 6,000 different hyperparameter configurations of a DQN algorithm.

tional grid search techniques, we were able to better explore the best hyper parameters for a given model, while we explored different methods for implementing our model, and using our policies.

	Number	State	Value ↓	Duration(ms)	Params
⌄	1413	Complete	1.0050818692572647	5135354	action_subset: leaderboard_actions, algo_type: DQN, dqn_batch_size: 256, dqn_buffer_size: 100000, dqn_exploration_final_eps: 0.1796559659051645, dqn_exploration_fraction: 0.48572600600279464, dqn_gamma: 0.995, dqn_learning_rate: 0.0006760923694541781, dqn_learning_starts: 0, dqn_net_arch: medium, dqn_subsample_steps: 2, dqn_target_update_interval: 15000, dqn_train_freq: 256, n_parallel_envs: 4, observation_space: InstCountNorm
⌄	1437	Complete	1.0003477152931737	9189700	action_subset: leaderboard_actions, algo_type: DQN, dqn_batch_size: 256, dqn_buffer_size: 100000, dqn_exploration_final_eps: 0.18872358265244693, dqn_exploration_fraction: 0.09013637856870513, dqn_gamma: 0.98, dqn_learning_rate: 0.0016478376683243263, dqn_learning_starts: 0, dqn_net_arch: medium, dqn_subsample_steps: 2, dqn_target_update_interval: 15000, dqn_train_freq: 16, n_parallel_envs: 2, observation_space: InstCountNorm
⌄	1199	Complete	0.9907483879040229	3683035	action_subset: cg_action_reference, algo_type: DQN, dqn_batch_size: 512, dqn_buffer_size: 50000, dqn_exploration_final_eps: 0.18604810363477464, dqn_exploration_fraction: 0.4398226450559795, dqn_gamma: 0.999, dqn_learning_rate: 0.00016019213292213234, dqn_learning_starts: 1000, dqn_net_arch: tiny, dqn_subsample_steps: 8, dqn_target_update_interval: 20000, dqn_train_freq: 1, n_parallel_envs: 4, observation_space: InstCountNorm
⌄	1429	Complete	0.9899110572179779	9250518	action_subset: leaderboard_actions, algo_type: DQN, dqn_batch_size: 256, dqn_buffer_size: 100000, dqn_exploration_final_eps: 0.04233307157202048, dqn_exploration_fraction: 0.4868227436152491, dqn_gamma: 0.98, dqn_learning_rate: 0.0007167262599084408, dqn_learning_starts: 0, dqn_net_arch: medium, dqn_subsample_steps: 2, dqn_target_update_interval: 15000, dqn_train_freq: 4, n_parallel_envs: 2, observation_space: InstCountNorm
⌄	1446	Complete	0.9892650857043919	8504428	action_subset: leaderboard_actions, algo_type: DQN, dqn_batch_size: 256, dqn_buffer_size: 100000, dqn_exploration_final_eps: 0.1894487797522927, dqn_exploration_fraction: 0.4859996553604364, dqn_gamma: 0.95, dqn_learning_rate: 0.001750017326453419, dqn_learning_starts: 0, dqn_net_arch: medium, dqn_subsample_steps: 2, dqn_target_update_interval: 15000, dqn_train_freq: 16, n_parallel_envs: 2, observation_space: InstCountNorm
⌄	1431	Complete	0.9892303151515079	9366201	action_subset: leaderboard_actions, algo_type: DQN, dqn_batch_size: 256, dqn_buffer_size: 100000, dqn_exploration_final_eps: 0.18963676683293404, dqn_exploration_fraction: 0.48898172907062354, dqn_gamma: 0.98, dqn_learning_rate: 0.000688491725354111, dqn_learning_starts: 0, dqn_net_arch: medium, dqn_subsample_steps: 2, dqn_target_update_interval: 15000, dqn_train_freq: 128, n_parallel_envs: 2, observation_space: InstCountNorm
⌄	1352	Complete	0.9885739001270849	5954913	action_subset: leaderboard_actions, algo_type: DQN, dqn_batch_size: 256, dqn_buffer_size: 100000, dqn_exploration_final_eps: 0.18971755470371343, dqn_exploration_fraction: 0.2448617979102589 2, dqn_gamma: 0.98, dqn_learning_rate: 0.0009052313753842884, dqn_learning_starts: 0, dqn_net_arch: medium, dqn_subsample_steps: 2, dqn_target_update_interval: 15000, dqn_train_freq: 256, n_parallel_envs: 4, observation_space: InstCountNorm
⌄	1414	Complete	0.9884509048424661	4984083	action_subset: leaderboard_actions, algo_type: DQN, dqn_batch_size: 256, dqn_buffer_size: 100000, dqn_exploration_final_eps: 0.18970175341034495, dqn_exploration_fraction: 0.4870059004564221, dqn_gamma: 0.995, dqn_learning_rate: 0.0023700232291501114, dqn_learning_starts: 0, dqn_net_arch: medium, dqn_subsample_steps: 2, dqn_target_update_interval: 15000, dqn_train_freq: 256, n_parallel_envs: 4, observation_space: InstCountNorm
⌄	1202	Complete	0.9836401638982352	4422349	action_subset: cg_action_reference, algo_type: DQN, dqn_batch_size: 512, dqn_buffer_size: 1000000, dqn_exploration_final_eps: 0.01055052381721959 5, dqn_exploration_fraction: 0.44535997349090795, dqn_gamma: 0.9999, dqn_learning_rate:

Figure 8: A list of example trial results from our best study.

3.1.3 Compiler Gym

Compiler Gym [7] is a compiler simulation tool being developed by Facebook/Meta's research team in order to make it easier to research compiler optimization via machine learning. As such, this tool that made our research into compiler optimization significantly less arduous. Some of the most important features of Compiler Gym include: datasets of programs separated categorically by the content of the code, various helper methods and wrappers for augmenting the environment to suit the wants and needs of the programmer, and compiler environments that can be used with any framework that supports OpenAI's gym software. Specifically, we used Compiler Gym's LLVM environment, which is based on the Clang compiler, with a goal of finding phase orderings (mentioned above) that reduce the final code size of a compiled program.

In Compiler Gym a optimization task is represented as a Markov Decision Processes (MDPs) and the developer interacts with the task as an environment using OpenAI's Gym interface [10]. Within these environments there are three different spaces (Action, Observation, and Reward Spaces) the developer can choose from to define the environment their model/agent operates in. One such space is the *Action Space* of the environment. The action space defines the possible

actions that can be taken from a given MDP state. Action spaces can either be discrete or continuous. In our project and in Compiler Gym in general the action space is a finite set of LLVM compiler optimizations.

Another concept used to define the environment in which our models operate is the *Observation Space*. The observation space defines how the environments report the current state to the model. Compiler Gym supports multiple observation spaces such as "numeric feature vectors generated by compiler analyses, control flow graphs, and strings of compiler IR" [7]. For our project we experimented on the different numeric feature vectors provided by compiler gym such as Autophase vectors [2], instruction count vectors, and normalized instruction count vectors.

The final space used to define the compiler gym environment is the *Reward Space*. The reward space defines the values used by the reward function to provide feedback to the model about the current chosen action. For our project we used the code size reduction reward space which is defined as:

$$R(s_t) = \frac{C(s_{t-1}) - C(s_t)}{C(s_{t=0}) - C(s_b)}$$

Where C(s) is a cost function that takes as input the current program state t. The subscript b is the baseline state according to the -03 baseline policy. This means the reward of the model is defined by the current code size reduction normalized by the code size reduction achieved from the -03 flag. A reward of 1 is defined as the success threshold of a model and would mean an equal code size reduction achieved by the current model and the -03 flag. Upon accumulating the reward at every state you would get the total reward of the model.

3.1.4 Compute Servers

To run our experiments at the scale we wanted using Optuna and stable-baselines3, we had to leverage a number of machines. We had access to the following servers:

1. Bridges-2: A Supercomputer at the Pittsburgh Supercomputing Center (PSC). Funded by a $10-million grant from the National Science Foundation.
2. Knuth: A research server at URI provided by Prof. Marco Alvarez.
3. Minksy: A research server at URI provided by Prof. Noah Daniels.
4. Homework Server: A server used by the computer science students at URI.

To make use of our time and resources most effectively we wrote scripts to have our models continually tuning on these servers.

4 Experiments

4.1 Techniques

4.1.1 Combining Optuna, Stable-baselines, and Compiler Gym

Our main approach to experimentation was to develop a helper function for sampling hyperparameters of each of the stable-baselines3 algorithms with the help of Optuna, and search a reasonably large hyperparameter space over many "trials". We tuned all of the models available to us by stable-baselines3 (DQN, PPO, A2N).

Separate of model hyperparameters, we were also able to formulate environment settings and the choice of algorithm as their own variables which were also selected by Optuna in our

trials. For environment settings, we tuned the observation spaces given by CompilerGym for this problem and the action space by providing different reductions in the set of actions that the agent can take. We also used an optuna helper function to select the algorithm that we were testing and sample hyperparameters accordingly.

The Compiler Gym repository is home to a leaderboard of research groups for the code size reduction problem. Each group is evaluated on a standardized test data set, and a corresponding utility method is provided for making evaluation easier. Unfortunately, this method proved very problematic for us, and instead of evaluating the performance of our models on the score returned by this method exactly, we needed to develop an approximate evaluation method that got as close to the leaderboard scoring method as reasonably possible. Therefore, our resulting scores are approximations of those on the leaderboard, and we would expect our scored to be slightly overestimated. Altogether, with these three tools, we ran 18 studies, and over 25,000 trials. In each of these trials, a model would be fully trained and evaluated autonomously, unless they were pruned for under performing early. To see which hyperparameters we tuned and exact methods please see the code on our public GitHub repository.

Author	Algorithm	Links	Date	Walltime (mean)	Codesize Reduction (geomean)
Facebook	Random search (t=10800)	write-up, results	2021-03	10,512.356s	**1.062×**
Facebook	Random search (t=3600)	write-up, results	2021-03	3,630.821s	1.061×
Facebook	Greedy search	write-up, results	2021-03	169.237s	1.055×
Facebook	Random search (t=60)	write-up, results	2021-03	91.215s	1.045×
Facebook	e-Greedy search (e=0.1)	write-up, results	2021-03	152.579s	1.041×
Jiadong Guo	Tabular Q (N=5000, H=10)	write-up, results	2021-04	2534.305	1.036×
Facebook	Random search (t=10)	write-up, results	2021-03	**42.939s**	1.031×
Patrick Hesse	DQN (N=4000, H=10)	write-up, results	2021-06	91.018s	1.029×
Jiadong Guo	Tabular Q (N=2000, H=5)	write-up, results	2021-04	694.105	0.988×

Figure 9: The Compiler Gym leaderboard from the README of the C.G. Github repository.

4.1.2 Assessing Code Similarity

As we've previously mentioned and referenced in Fig. 4, model performance varies on the different datasets available in compiler gym. This led us to questioning why this may be, and if these individual databases are similar. Therefore, we used t-stochastic neighbor embedding methods to assess whether or not samples from these datasets shared similarities and could be grouped. Specifically, if this is the case, we were optimistic towards producing an ensemble of methods, and train models to be more focused on specific types of programs. As seen in Fig. 10, there are clear clusters which these databases fall into. If there was more time for this project, we would have further investigated the development of an ensemble method which leverages more sophisticated techniques of applying learned policies.

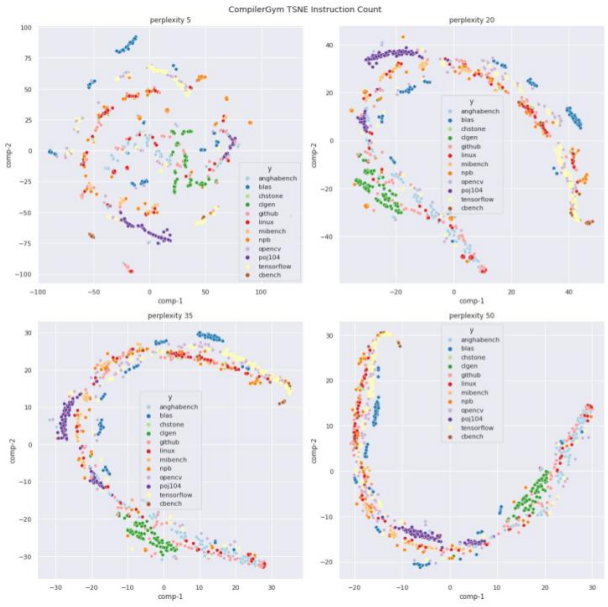


Figure 10: t-SNE plots, identifying code similarity across the different datasets supported by Compiler Gym. This figure presents randomly sampled benchmarks with an associated dataset, and is reduced to two dimensions to show clustering. This shows that the collection of benchmarks within a dataset share similar features.

4.1.3 Reducing the Action Space

Another strategy we did pursue, is reducing the action space. Previously stated in the report, LLVM and Compiler Gym supports 123 optimization flags, meaning that not only are there many actions that can be taken at every step, in a non-reducing fashion, the depth being potentially infinite means that it would be challenging to develop practical models which will generalize well under such conditions. While there were several steps towards confidently imposing this constraint, this also led towards better performing models. As seen in Fig. 11, Facebook Research has previously assessed the average cumulative reward for each optimization. By reducing the action space to the top 30-20 actions, we were able to produce a high number of models in the hyperparameter selection process which can close to performing as well as the standard -Oz macro.

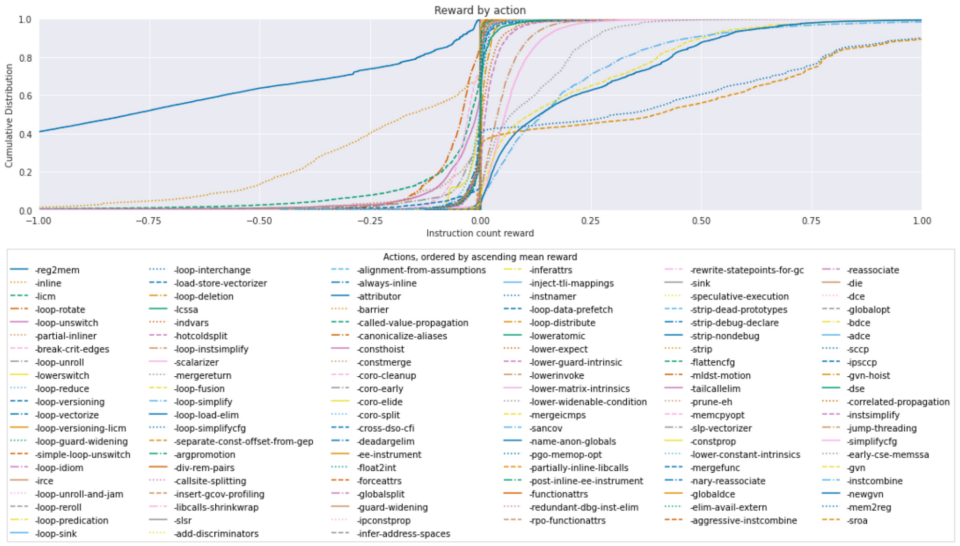


Figure 11: Cumulative reward distribution for each of the available actions within Compiler Gym. From Facebook Research [Source].

4.2 Optuna Study Results

Using the aforementioned methods, we made steady progress throughout the course of our work on this project as shown in the figure below, always evaluating our agents' performance on our approximate instantiation of the Compiler Gym evaluation environment.

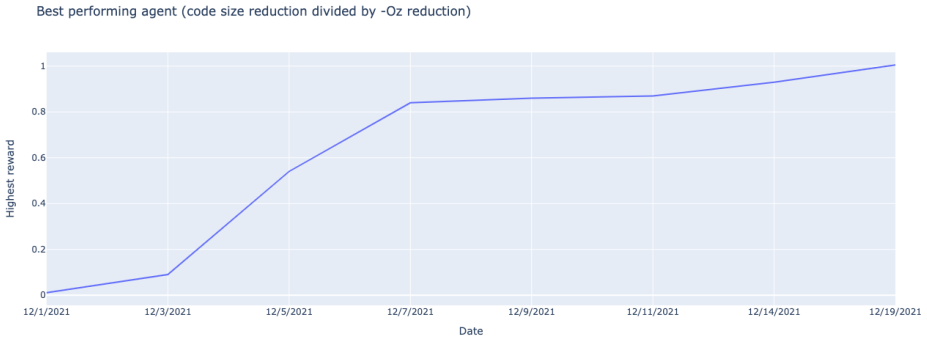


Figure 12: The results of our best performing trained agent on reducing code size relative to the code size reduction afforded by the builtin -Oz flag at different points over the course of our work on the project.

In the end, our best-performing trained agent scored 1.005x over the -Oz macro, narrowly outperforming the code size reduction afforded by conventional method. The following is a figure showing the hyper parameter correlation matrix of our most successful study. To achieve this we tightened the range of possible hyper-parameters learned from previous trials and changed to observation space to the normalized instruction counts. For the study that yielded these results, we ran 1600 trials, training roughly 1600 for those that weren't pruned, and the top 10 had near -Oz levels of performance, while the top 100 models all ranged between 0.85x and 1.005x. The results from this study are demonstrated in the plot below, Fig. 13, where we can see how Optuna began to prioritize DQN, ultimately converging to an optimal solution for the

allowed hyperparameter space imposed on the study. The cluster can be seen towards the end of the trials, converging in the top right.

Figure 13: Demonstrating how Optuna began to prioritize certain models over others while searching for the best hyperparameters of the individual models..

Lastly, another thing we learned from this particular study, is the correlation between the objective value, and the hyperparameters considered in this study (some of them). As seen below in the Pearson correlation matrix in Fig. 14, the observation space, the number of hidden layers, and the discount rate, were among some of the most impactful parameters which had the strongest correlation towards a successful model. This led us towards wanting to explore more models with different techniques for representing the observation space, if we had more time.

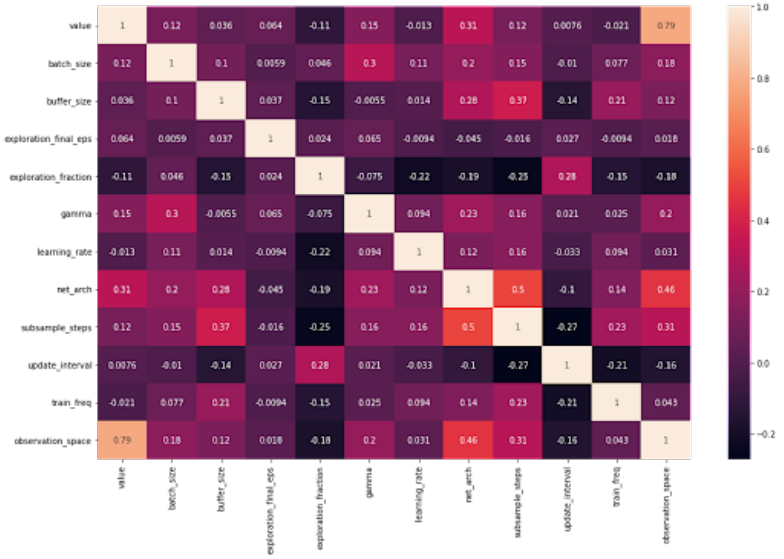


Figure 14: Pearson correlation matrix from our most successful study. Revealing that observation space has the greatest correlation with results.

5 Conclusions

In conclusion we are happy with what we learned through this project. This being our first exposure to Deep Reinforcement Learning we had to spend lots of time learning the theory and how to best approach our problem. Through this project our team got a deeper and practical understanding of Deep Reinforcement Learning, and how to approach and work with the models. The biggest insights we found was from the performance gaps in the hyper-parameter searching. Observation space and activation functions had a larger impact on our results than we had originally expected. This can be seen in the Experiments section above. We discovered and learned new tools such as Optuna which will assist us for all future machine learning work. We also gained lots of experience working with the libraries used in this project, and analyzing results from machine learning models.

Our results met our goal of making it onto the leader-board however we are not satisfied with the placing. After continued effort we believe we would continue to learn more about Deep Reinforcement Learning models and be able to improve upon our results. A important note is that our current top performing model is just from hyper-parameter tuning and not results from the Compiler Gym evaluation function. The weights of the models tuned from Optuna are not saved. We have trained models using our best hyper-parameters but were running into issues with using the Compiler Gym evaluation function on our code. This is something we want to amend.

For future work we have identified multiple areas of improvement. Correctly evaluating our models and continuing to tune and prune our models will hopefully lead to improvements. Improving the observation space for vector representation to something more descriptive of the code like graphs is also an area of improvement. To be able to implement this we anticipate needing to develop new ways to use Deep Reinforcement Learning algorithms so they can accept and evaluate graphical inputs. Towards the end of the project we also felt limited by the algorithms implemented in Stable-Baselines3 and seek making custom models and training methods. This would allow us to have a greater degree of flexibility in how we use the environments observation space. Another objective we would have liked to implement, is a more sophisticated use of the model once obtained. Since these models offer a best approximation for the value of an action, we would have liked to allow these models to branch and explore solution spaces before implementing concrete actions, using Compiler Gym's built-in forking tools. We see all of these tasks as achievable goals for future development.

Bibliography

[1] J. Davidson, G. S. Tyson, D. Whalley, and P. Kulkarni, "Evaluating heuristic optimization phase order search algorithms," in *International Symposium on Code Generation and Optimization (CGO'07)*, pp. 157–169, IEEE, 2007.

[2] Q. Huang, A. Haj-Ali, W. Moses, J. Xiang, I. Stoica, K. Asanovic, and J. Wawrzynek, "Autophase: Juggling hls phase orderings in random forests with deep reinforcement learning," *arXiv preprint arXiv:2003.00671*, 2020.

[3] A. H. Ashouri, W. Killian, J. Cavazos, G. Palermo, and C. Silvano, "A survey on compiler autotuning using machine learning," *ACM Computing Surveys*, vol. 51, no. 5, 2018.

[4] "Clang compiler user's manual." `https://opensource.apple.com/source/clang/clang-23/clang/tools/clang/docs/UsersManual.html`. (Accessed on 12/18/2021).

[5] C. Cummins, *Deep Learning for Compilers*. PhD dissertation, University of Edinburgh, 2020.

[6] H. Leather and C. Cummins, "Machine Learning in Compilers: Past, Present and Future," *Forum on Specification and Design Languages*, vol. 2020-Septe, 2020.

[7] C. Cummins, B. Wasti, J. Guo, B. Cui, J. Ansel, S. Gomez, S. Jain, J. Liu, O. Teytaud, B. Steiner, Y. Tian, and H. Leather, "CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research," pp. 1–12, 2021.

[8] Dataiku, "Dataiku: Choosing a deep reinforcement learning library," 2020.

[9] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.