Compiler-RL: Reinforcement Learning Approaches to Phase Ordering

Tsz Hang Kiang
Georgia Institute of Technology
tkiang3@gatech.edu

Adithya Vasudev
Georgia Institute of Technology
avasudev8@gatech.edu

Rishabh Jain
Georgia Institute of Technology
rjain343@gatech.edu

Zheyi Zhang
Georgia Institute of Technology
zzhang3058@gatech.edu

Abstract

The order in which a compiler makes passes on code such as dead code elimination, register allocation, and code in-lining can have great impact on the quality of the resulting emitted assembly. However, finding the optimal order and number of times to perform these passes, given a specific goal such as binary size or execution time, is a hard problem known as phase ordering.

There is already research in applying RL techniques to phase ordering, targeting a wide variety of resulting benefits, such as smaller memory foot- print, lower instruction count (and consequently, smaller executable file size), as well as effective speed of executable. The problem can be formalized as learning a function that maps an Intermediate Representation (IR) of a program to a weight for each possible optimization pass and optimizing this function based on a reward. We wish to investigate further using Deep RL to improve upon existing results.

In this paper, we provide an overview of tackling a problem using Deep RL specific to the ProGraML [4] observation space, which represents code in graph form. To supplement this graphical approach to code representation, we further propose MultiProGraML, an augmentation of ProGraML that encodes multiplicity information to better encapsulate looping and repetition in code. Finally, we present two approaches to tackling this phase ordering problem involving reinforcement learning: graph-based Deep Q-Networks (DQN) and Proximal Policy Ordering [12]. We show that DQN models, selecting one pass at a time from a policy, do perform better than both the base LLVM compiler and randomly selecting passes, and that PPO models are excellent when it comes to selecting passes, performing significantly better than LLVM's preset order.

1. Introduction

In modern software development, compilers play an important role since they translate high-level programming languages into binary codes; however, correct translation is only one part of the responsibility of a compiler. The other part is to optimize the generated binary code while maintaining its correctness and "optimized" code has several benchmarks – speedup ratio, code size reduction, parallelization level, etc. There are many outstanding compilers like GCC and LLVM that can greatly optimize the generated binary code without inducing misinterpretations. In our work, we mainly focus on solving this problem using an RL method.

1.1. Problem Summary

The general purpose of this problem is to find the optimized binary machine code for given code in high-level programming language. For our project, we will focus on phase ordering for the LLVM compiler infrastructure, that is, given an input sequence of Intermediate Representation (IR) instructions, which is a form of source code that LLVM can optimize, we want to compute the series of compiler optimizations to perform in order to result in the smallest number of output IR instructions.

The problem can be formalized as learning a function that maps an immediate representation of a program (**input**) to a weight for each possible optimization pass and optimizing this function based on a reward (**output**). We wish to investigate further using Deep RL to improve upon existing results (**goal**).

This problem is clearly important, since compilers play a critical role in modern software development. Inefficient compilers can lead to poor utilization of compute resources, which in turn wastes energy when considered on a large scale, such as a large company's datacenters. In a standard compiler, the order of compiler phase passes is fixed, but with enough research, it may be possible to optimize for

specific programs and see economic and environmental improvements.

For a specified programming language, it is possible that a manual compiler can optimize it to a significant extent, especially if the language is easy enough. However, most modern programming languages are too complicated to be manually optimized, moreover, programming languages evolve in a fast speed by adding more and more new features, which induces extra difficulty to manually optimize a compiler. Therefore, Machine Learning based methods are important to this problem, and thus exploring Reinforcement Learning based approach has merits.

2. Related Work

2.1. CompilerGym

CompilerGym [6] is an open-source Python library developed by Facebook AI that formulates compiler optimization problems into more exploitable environments with friendly interfaces. Using this library, we can easily transform our optimization problem into an RL environment, and reproduce some experiments of RL for LLVM phase ordering as baseline methods. Our deep learning models were ran in the CompilerGym environment. Moreover, this platform is highly extensible, which is beneficial for future research.

2.2. Input Representation

A first approach at representing the input program is the IR (Intermediate Representation) described in [6]. However, this isn't necessarily the best representation if we want to try to optimize phase ordering. For example, in [11], the input program is further encoded as a Control Flow Graph, while in [2], the input program is encoded as a Contextual Flow Graph, then gets encoded as vectors by inst2vec in a 200-dimensional embedding space as the preprocessed input, similar to and inspired by word2vec. These methods provides us with important intuitions on the follow-up of our baseline method.

2.3. ProGraML

ProGraML [4] extracts a portable, language-agnostic graph representation of programs given IR. It resolves some potential problems of other code representation methods. For example, code2vec overvalues the importance of identifier names [1]; XFG does not capture operand order [2]; CDFG omits both type information and operand order. In contrast, ProGraML starts with constructing control flow graph, then add data flow to it, and finally add call flow. As a result, it produces a representation of IR as the union of a call graph, control-flow graph, and data-flow graph, as shown in Figure 1.

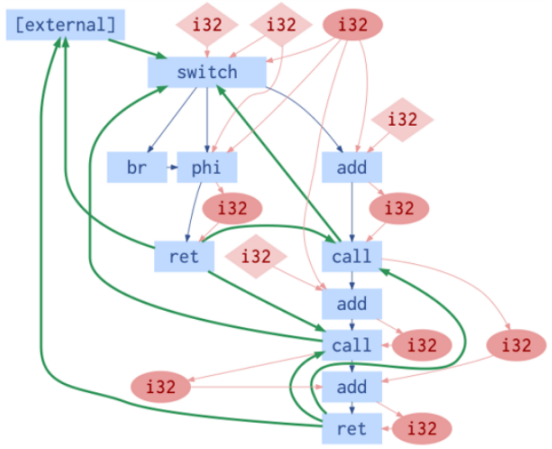

Figure 1. Sample ProGraML graph

2.4. CompilerGym Leaderboards

On the main page of the CompilerGym GitHub repository, there is a public leaderboard containing approaches and models contributed by other people. At our time of writing, types of approaches include random search, greedy search, graph attention, Tabular Q networks, and ϵ-Greedy search [6].

In our work for this paper, we were particularly inspired by the graph attention and DQN approaches and have attempted to build our own as well as improve on top of these existing implementations.

3. GraphConverter

3.1. Motivation

While ProGraML is a big step up from the Autophase representation of the state space, when it comes to actual training, we believe it has several drawbacks: (1) the observation space is very sparse, making it hard for a model to learn the representation, (2) semantically similar programs that vary in size may not result in very similar representations; this is an extension of the previous point about sparsity.

We will give an explicit example to demonstrate this issue. Imagine two programs, a program `A` consisting of a single for-loop, a second program `B` with the for-loop repeated 26 times. Pseudocode for the latter program is shown in Figure 2.

Each single loop can be represented in pseudo-IR as depicted in 3. Semantically, it is reasonable to consider `A` and `B` as similar. In the ProGraML representation of the IR in Figure 3, we would have a directed edge from `brneq` to `call`, representing the control flow of branching to the `call` instruction. In the ProGraML representation of program `B`, we would have 26 nodes of `brneq` and `call`, each

```
for i in range(100):
  a(i)

for i in range(100):
  b(i)

for i in range(100):
  c(i)
  .
  .
  .
for i in range(100):
  z(i)
```

Figure 2. Pseudocode of program `B`: for-loop repeated 26 times

```
loop_a:
    call a(i)
    add i i 1
    brneq i 99 loop_a
```

Figure 3. IR Representation of for-loop

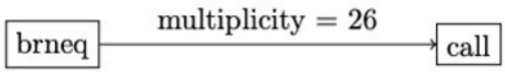

Figure 4. MultiProGraML Edge

with an edge that connects each pair. Ideally, we want to condense this into Figure 4, where a single edge and a multiplicity weight of 26 describes the 26 copies of for-loops in program `B`.

3.2. MultiProGraML

We now formally introduce a new graph representation which we will refer to as MultiProGramML and describe how we can calculate it. We calculate MultiProGraML from an input ProGraML graph by creating only one node per unique node type, and augmenting edges between MultiProGraML nodes with a multiplicity weight that represents the number of edges in the original ProGraML graph of the same edge type between the corresponding ProGraML nodes.

3.3. Calculating Embeddings

In order to embed edges of a MultiProGraML graph, we take the sum of all embeddings of all other properties besides the multiplicity first. Then, to take into account the multiplicity of the edge, we multiply the embedding sum by the softmax of the multiplicity with respect to the cumulative multiplicity of the graph.

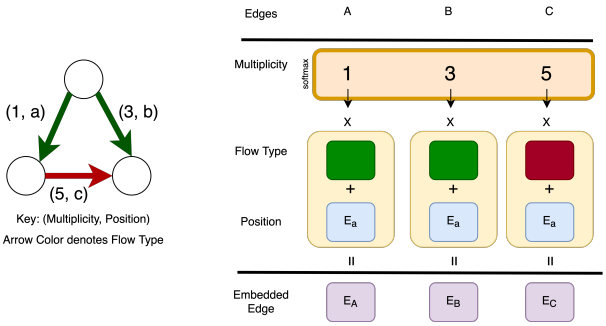

Figure 5. Calculating MultiProGraML Edge Embeddings

4. Problem Statement

4.1. Definitions

We first start with definitions. Let S be the set of all possible states, and A be the set of all possible actions. In the context of the rest of this paper, S will be the set of all possible ProGraML graphs, and A will be all the available compiler phase optimizations available in LLVM.

We define a cost function $C : S \to \mathbb{R}$ and a reward function $\mathcal{R} : S \times S \to \mathbb{R}$ as

$$C(s) = \text{Number of IR instructions in } s$$
$$\mathcal{R}(s_i, s_j) = C(s_i) - C(s_j).$$

That is, the reward function is defined as the difference in the number of IR instructions in the state observation between time $t-1$ and t. Thus, the reduction in IR instruction count in the transition from s_{t-1} to s_t can be denoted as $\mathcal{R}(s_{t-1}, s)$. Furthermore, we define a normalized reward function

$$\mathcal{R}_0(s_i) = \frac{R(s_0, s_i)}{C(s_0)},$$

where we calculate the reduction in IR size with respect to the initial code size. The normalized reward function $\mathcal{R}_0$ is important for providing a standardized score that can be compared for input IRs of varying size.

Thus, our goal is to find an optimal policy $\pi^*(s, a) = \mathbb{P}(a_t = a | s_t = s)$ such that for some preset discount factor $\gamma \in [0, 1)$, the value

$$Q^{\pi^*}(s_i, a_i) = \mathbb{E}\left[\sum_{k=1}^{\infty} \gamma^k \mathcal{R}(s_{i+1}, s_{i+k+1}) \middle| s_i, a_i, \pi^*\right]$$

is maximized.

An episode is defined as all the steps taken in the gym environment for a specific data point from the dataset. Termination happens when we either hit a set limit number of steps (256 in our case), or the model emits the `TERMINATION` action.

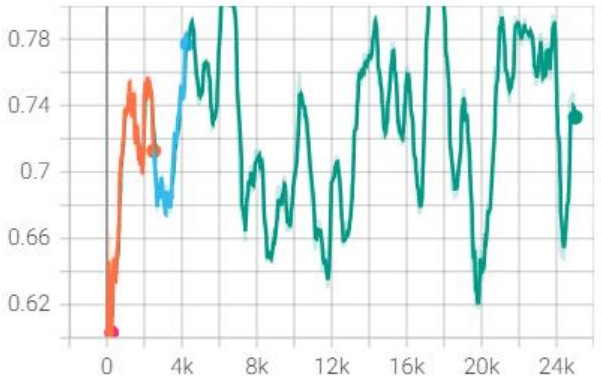

Figure 6. Baseline Method Episode Mean Reward

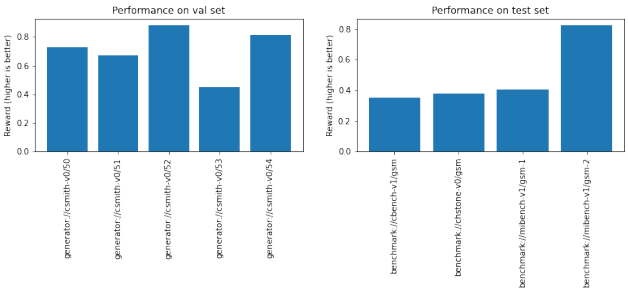

Figure 7. Baseline Method Test Result

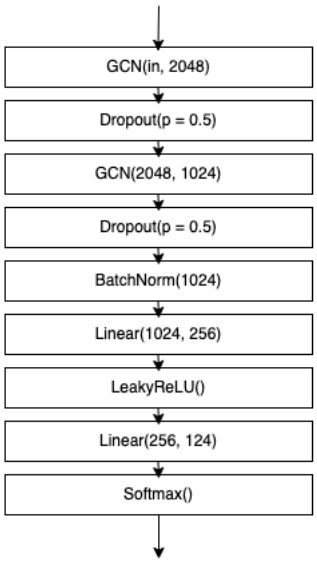

Figure 8. Deep Q-Learning Network Architecture

4.2. Datasets

CompilerGym provides many LLVM-IR benchmarks as datasets to use for reinforcement learning. These benchmarks are essentially the program that the environment is compiling and the model is trying to optimize. Specifically, we used `cbench-v1` [9] as the benchmark for the majority of our models, including DQN and PPO. `cbench-v1` includes a variety of different program categories, such as sorting, cybersecurity, and a lot more. It is a standard benchmark used in the compiler community and represents realistic and comprehensive coverage of workloads.

5. Methods

5.1. Baseline Method

Our baseline method is the *VII.G. Reinforcement Learning for LLVM Phase Ordering* part of [6]. Using CompilerGym, the optimization problem is formulated as an RL environment– every optimization step is mapped as an action, and the corresponding result in the specified metric is mapped to a reward; this RL environment is then solved using PPO [12].

As for the evaluation metric, we used Geomean code size reduction. The experiment is trained using CSmith dataset, tested by cBench, CHStone, MiBench dataset, to align with the last column of Table VI in [6]. Considering the current limit of computing resources, we pruned the number of training benchmarks to 50 and set the maximum training episode to 5000, which is 10k in [6].

As a result, 6 shows the episode mean reward, 7 shows the performance on validation set and testing set. As shown in the figure, the baseline result is desirable referring to the results in [6], especially for the smaller testing set MiBench/gsm-2, considering we used much less training episodes and a subset of the training benchmarks.

5.2. Deep Q-Learning

Our first reinforcement learning agent was a Deep Q-Learning model we trained over various benchmarks from the LLVM environment within [6].

The network itself is a Graph Convolutional Neural Network implemented using the PyTorch Geometric package [8]. The network architecture [8] consists of 5 major layers: 2 graph convolutional layers (2048 channels and 1024 channels respectively), with dropout after each convolution (p = 0.5), 2 linear layers (256 and 124 output features respectively) with a LeakyReLU activation function, and a batch normalization between the convolutional and linear layers. The Softmax function is applied to the values before they are output.

The agent was trained using the Adam optimizer (with a learning rate of 0.01) and the Huber Loss function (L1 loss between correct Q-values and approximated Q-values),

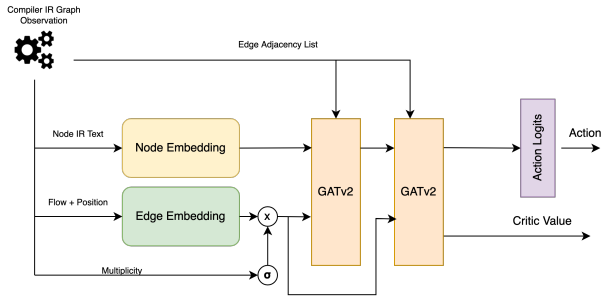

Figure 9. Graph Attention + PPO Architecture

which is determined through this equation: $L = Q(s,a) - (r + \gamma * max_a Q(s',a))$, with $\gamma = 0.999$ The agent was trained over 65 episodes, with a stopping length of 15 for each episode fixed during training (this prevents the issue of an episode stepping indefinitely).

Actions were picked through an epsilon greedy search, beginning with selection of a random action with probability p = 0.9, and decaying to a probability of p = 0.05 over the course of training.

5.3. Graph Attention + PPO

We have also trained an Actor-Critic model using a Graph Attention Network and Proximal Policy Optimization. A similar approach using graph attention and PPO exists. [10] Where we differ is the use of our preprocessing to convert a ProGraML observation into our MultiProGraML representation, as well as modified embedding logic.

The model [9] consists of embedding layers, stacked attention layers, followed by two separate linear outputs, one of dimension N representing the logits corresponding to the N available actions (Actor) and a scalar value (Critic). Specifically, the nodes are embedded by the text description (IR instruction) and edges are embedded by taking multiplying the softmax of the multiplicity w.r.t. the global total of multiplicity, by the sum of the embeddings of the flow and position values of the edge [7]. Positional encoding is used for the position value. As for the attention layers, 2 GATv2 [3] layers are consecutively connected.

The hyperparameters for the Graph Attention and embeddings are: Node Embedding Output Dimension (32), Edge Output Embedding Dimension (32), Output of first GATv2 layer, which is equal to input to second GATv2 layer (256).

The hyperparameters for PPO: Number of epochs per update (4), Actor Learning Rate (0.0001), Critic Learning Rate (0.0001), Discount Factor γ (0.9), PPO ϵ-clip (0.1).

Finally, the training procedure via PPO is as follows. Using the action logits, a corresponding discrete distribution $[1, N]$ is defined. Then, for K epochs, we sample an action from our distribution, which we then perform. Each new state and associated actual reward is stored into a replay buffer. After the K epochs, the model parameters are updated via PPO by analyzing the replay buffer and the value output from the model.

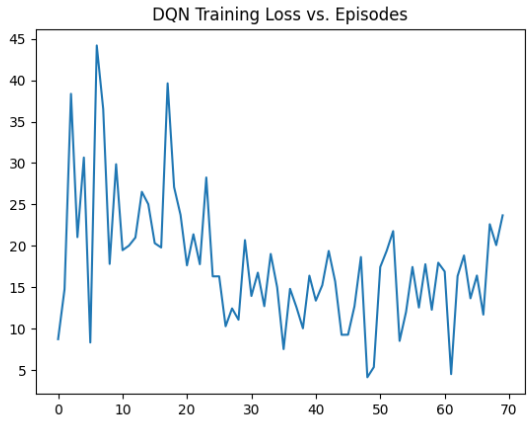

Figure 10. Deep Q-Learning Loss over 70 Episodes

6. Results

6.1. Deep Q-Learning

The agent's performance was evaluated on three metrics: loss, benchmarking reward (via CompilerGym) and game-playing reward. The loss per episode over 70 and 120 episodes can be seen. In Figure 10, we show a training graph for 70 episodes, and in Figure 11, we have a training graph for 120 episodes. In both graphs, it is clear to see that the model rapidly learns within the first 25 and then begins to capture some edge cases. At around episode 65, however, the model's loss actually begins to increase, which can be captured by an overshot of the local minimum or overfitting. We cross verified this when we trained the model on 120 episodes, where a steady increase in the loss begins at about episode 65 and continues until termination.

For benchmarking, we evaluated the model on CompilerGym's standard leaderboard benchmarks, which is the the "Synthetically generated OpenCL kernels" dataset, alternatively known as `cbench-v1` [5]. Over this dataset, the CompilerGym evaluator evaluated a geometric mean reward of 0.30, thereby showing that the model did show improvement over the native LLVM compiler.

Finally, we evaluated the performance of the Deep Q-Learning agent against the a baseline random agent, which picked compilation passes with complete randomness at each step. 150 unique programs were selected at random and both agents were passed the programs and allowed to run. The total reward over its execution was retrieved for each agent over each run and a winner was determined by

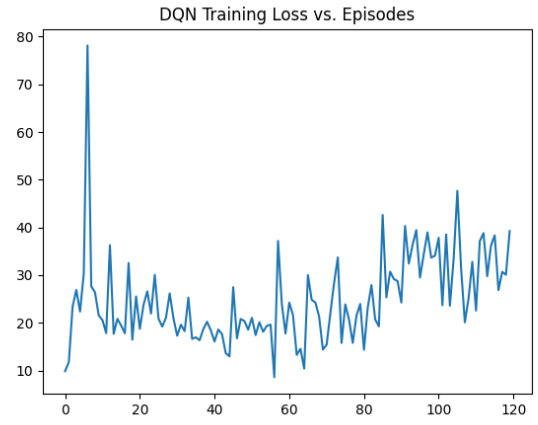

Figure 11. Deep Q-Learning Loss over 120 Episodes

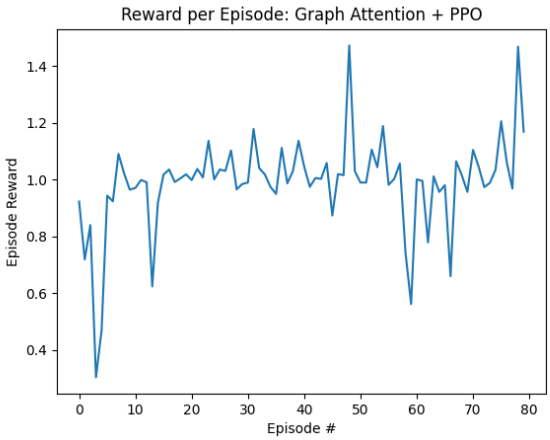

Figure 13. Reward per episode for Graph Attention/PPO model

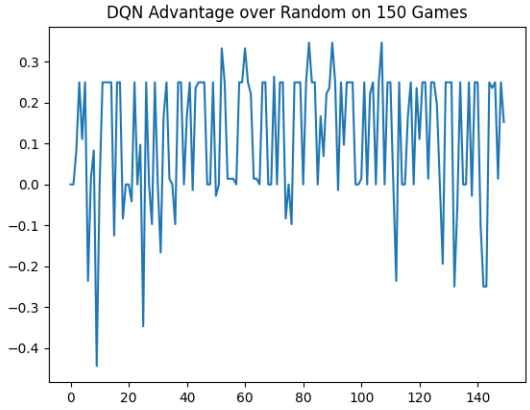

Figure 12. DQN Reward Advantage over Random Agent

the agent that had the higher overall reward.

In the 150 trials that both agents were presented, the Q-Learning agent won 133 of them (an 88.6% win rate). We believe that the 17 losses. were either due to both agents finding in-optimal passes (yielding a reward of 0) or the random agent getting rather lucky and picking a surprisingly optimal set of passes for the particular code. Nevertheless, in the large majority of scenarios, the Deep Q-Learning agent handily beat the random model. A subset of the performance metrics for the two agents can be found in Figure 12.

6.2. Graph Attention + PPO

After 80 episodes of training, when tested on all the IR corresponding to programs from the `cbench-v1` dataset, we were able to achieve a geometric mean normalized reward of $\sqrt[\mathcal{D}]{\prod_{i=1}^{\mathcal{D}} \mathcal{R}_0(s_{\text{end}})} = 0.9874$ where $\mathcal{D} = 230$ is the number of programs in the `cbench-v1` test dataset.

Figure 13 displays the reward $\mathcal{R}_0(s_{\text{end}})$ per episode. Note that each episode uses a randomly selected data point from the dataset, which can explain the major fluctuations in reward between episodes. Overall, there is a trend of the increasing reward as episode count increases.

6.3. Overall Discussion

The CompilerGym GitHub repository comes with a leaderboard. Unfortunately, we were unable to beat an existing Graph Attention + PPO-based implementation with our MultiProGraML state space augmentation. We believe that this was due to our budget and time constraints; we had to restrict our number of GATv2 layers to only two. With more layer, more time, and more powerful hardware, we expect to achieve even better results.

Furthermore, we could take advantage of distributed RL frameworks such as Ray's RLlib, which implement distributed PPO and other asynchronous RL policies such as A3C and DD-PPO. During training, we realized that a lot of time was spent inside `env.step()`, which is where the CompilerGym environment performs the compilation phase optimization selected by our model. This compilation workload is CPU-intensive and unparallelizable, meaning that in our training, only 1 of our CPU cores was being utilized fully. We can fully utilize our available CPU compute power by running multiple gym environments simultaneously and having Ray handle observation and update aggregation.

7. Class Visualization

In our experiments of class visualization, we are training a simple neural network to detect the most optimal greedy phase pass in order to minimize final instruction count.

Each data sample is of shape $(X, 200)$, representing an IR file with X instructions that were converted into X vectors embedded into a 200-dimensional space.

Prior to feeding into the neural network, we took a random sample of 20 IR instructions, with replacement, to account for potential input IR files that may have less than 20 instructions.

The overall network architecture is a classical fully connected neural network with five layers (input layer, three hidden layers, and output layer) and a LeakyReLU activation function. The size of each of the hidden layers were initially determined randomly, and then fine-tuned by experimentation, resulting in hidden layer sizes of 16384, 10000, and 2048 nodes respectively.

The network has a final output linear layer with an output of size N where N represents the number of available actions (potential compiler pass optimizations), which is then used to predict the optimal action.

The optimal action is computed each epoch ahead of time. Then, the correct action is used to compute the loss via soft-max cross-entropy.

We trained using 10 epochs, with the Adam optimizer.

In order to convert the set of input IR vectors that has been backpropagated to produce the the class visualization, for each of the input IR vectors v_i, we want to find the corresponding $v_i^\star \in V$, where V is the vocabulary of the IR embedding, such that $v_i(v_i^\star)^T$, the dot product is maximized. This is to maximize the cosine similarity.

Unfortunately, we didn't get to experimenting with the backpropagation on the inputs.

We felt using *inst2vec* to represent the input IR is better since compared to the other observation space alternatives such as instruction counts, and control-flow graphs, the vector embeddings are continuous.

8. Offline Learning

As part of the suite of experiments done by CompilerGym [6], one such experiment entailed the use of a state transition dataset. This database stores detailed information such as observations (various representations) of each state and the transitions (which state is reached given an input and a specific compiler action). As an investigation, we wanted to implement a transformer model [13] to try to predict the next state. More specifically, the goal (which can be formulated as a prediction problem), was to predict the next intermediate representation (IR) of a program being compiled given an input for current IR and a specific action from the suite of actions for the compiler to take. Doing this in a supervised manner can help improve reinforcement learning algorithms and was useful for our own curiosity.

We started by basing our implementation off of the example given by the authors, where they attempted to predict the instruction count of a program after two compiler passes based on its ProGraML [4] graph representation. Unfortunately, there was a critical flaw that prevented us from successfully implementing the transformer. The transition table in the database was completely empty, and we needed that information to move from state to state. For a future investigation, it should be possible to generate our own test cases, either via storing states and transitions while running our own algorithm in the Gym environment, or via hacking together states and the specific action sequences that led to their generation, which the database does contain some data.

9. Conclusion and Future Work

Understanding how phases interact with each other is a hard and open problem. When introducing new optimization phases, we may introduce performance regressions in the resulting executable that are hard to reason about. A potential application of our trained agents would be diagnosing these regressions using model explainability techniques such as class visualization: we would like to use our model to tell us what kinds of input IR are conducive to what kinds of optimizations.

Additionally, an optimal compilation order for one program may not be the best for another - that's what makes this a difficult problem. In fact, it's hard to know when we've reached the absolute optimum, such as the point where reducing instruction count any further changes the behavior of the program. In the future,

Overall though, we were successful in creating and training models that improved on CompilerGym's benchmarks.

10. Team Contributions

Team Member	**Key contributions**
Rishabh Jain	State transition database for supervised learning; Transformer model for next state prediction; LaTeX/formatting/graph generation
Tsz Hang Kiang	MultiProGraML Embedding, Graph Attention + PPO
Adithya Vasudev	Deep Q-Learning Development and Training, Random Agent Game Simulation
Zheyi Zhang	Testing and baseline implementation, class visualization

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code, 2018. 2

[2] Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. *CoRR*, abs/1806.07336, 2018. 2

[3] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks?, 2021. 5

[4] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. Programl: Graph-based deep learning for program optimization and analysis. arXiv, 2020. 1, 2, 7

[5] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. Synthesizing benchmarks for predictive modeling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 86–99, 2017. 5

[6] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. Compilergym: Robust, performant compiler optimization environments for ai research. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, page 92–105. IEEE Press, 2022. 2, 4, 7

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. 2018. 5

[8] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019. 4

[9] Grigori Fursin. Collective tuning initiative, 2014. 4

[10] Anthony W. Jung. Graph attention network with dd-ppo, 12 2022. 5

[11] Ricardo Nobre, Luiz G. A. Martins, and João M. P. Cardoso. A graph-based iterative compiler pass selection and phase ordering approach. In *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, LCTES 2016, page 21–30, New York, NY, USA, 2016. Association for Computing Machinery. 2

[12] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. 1, 4

[13] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2017. 7