Offline Imitation Learning from Multiple Baselines with Applications to Compiler Optimization

Teodor V. Marinov [1] Alekh Agarwal [1] Mircea Trofin [2]

Abstract

This work studies a Reinforcement Learning (RL) problem in which we are given a set of trajectories collected with K baseline policies. Each of these policies can be quite suboptimal in isolation, and have strong performance in complementary parts of the state space. The goal is to learn a policy which performs as well as the best combination of baselines on the entire state space. We propose a simple imitation learning based algorithm, show a sample complexity bound on its accuracy and prove that the the algorithm is minimax optimal by showing a matching lower bound. Further, we apply the algorithm in the setting of machine learning guided compiler optimization to learn policies for inlining programs with the objective of creating a small binary. We demonstrate that we can learn a policy that outperforms an initial policy learned via standard RL through a few iterations of our approach.

1. Introduction

When applying Reinforcement Learning (RL) to real-world applications, two key challenges often prove to be critical blockers to adoption. First is that the online interaction-then-update loop in conventional RL poses a significant engineering overhead in most large-scale systems, that are more naturally designed to take a static Machine Learning (ML) model as a dependency and update this model only periodically in an offline manner. Second is that RL algorithms typically begin tabula rasa, that is, they only leverage the information they glean about the task at hand through these online interactions. Typical scenarios, where the use of RL is often preceded by prior attempts using rule-based or supervised ML approaches, come with a treasure trove of valuable data about desirable and undesirable behaviors, ignoring which leads to undesirable sample complexity of learning from scratch for RL. More important, the previously tried decision making policies, even when individually suboptimal, provide a valuable source of insight into the plausibly good choices in many scenarios. In this work, we study the question of leveraging such prior policies and any data collected using them, without necessarily relying on online policy updates.

Given these shortcomings, offline RL (Ernst et al., 2005; Hester et al., 2018; Kumar et al., 2020; Cheng et al., 2022), where the agent just learns from a static dataset collected using some arbitrary policy, as well as hybrid protocols (Song et al., 2022; Haarnoja et al., 2018; Silver et al., 2014) interpolating the fully online and offline settings have been proposed in the literature to take advantage of existing datasets, as well as to ease the requirement of completely online policy updates. In a different thread of work, the substantial literature on imitation learning (Pomerleau, 1988; Ross et al., 2011; Abbeel & Ng, 2004; Ho & Ermon, 2016) aims to leverage any existing policies that we seek to improve upon, along with the data collected using them. While imitation learning is studied in both online and offline settings, the particular scenario of having access to multiple policies of variable qualities that is of interest here, is only previously studied in an online setting (Cheng et al., 2020; Barreto et al., 2020).

In this paper, we formalize this question of having access to K baseline policies $\pi_1, \dots, \pi_K$, each of which can be quite suboptimal in isolation, and which we hope are strong in complementary parts of the state space. We further restrict ourselves to only receiving static datasets D^i collected from each policy π_i, and seek to learn a policy which can combine the strengths of all the baseline policies. We are particularly interested in challenging settings where the underlying RL problem has a long horizon, and we only receive sparse trajectory-level feedback at the end of each trajectory. To motivate this setting, we consider a running example of optimizing the inlining decisions in a compiler. The horizon of the RL problem here corresponds to the number of callsites in the program or function being compiled., which can range from tens to tens of thousands. The reward of a trajectory is the size of the binary we obtain after compiling

[1]Google Research, USA [2]Google, USA. Correspondence to: Teodor V. Marinov <tvmarinov@google.com>, Alekh Agarwal <alekhagarwal@google.com>, Mircea Trofin <mtrofin@google.com>.

the entire function. The offline setting we study is extremely well motivated here, where each interaction with the RL environment involves the expensive operation of compiling and linking the program, and integrating this within the RL loop engenders a significant engineering overhead.. With this setup, our paper makes the following contributions:

- A natural behavior cloning algorithm, BC-MAX, that combines the multiple policies by executing each policy in every starting state in our dataset, and then imitating the trajectory of the policy with the highest reward in that state. We give an upper bound on the expected regret of the learned policy to the maximal reward obtained in each starting state by choosing the best baseline policy for that state.

- We complement our analysis with a lower bound showing that the result is unimprovable beyond polylogarithmic factors in our setting.

- We apply BC-MAX to two different real-world datasets for the task of optimizing compiler inlining for binary size, and show that we outperform strong baselines in both the cases. In both cases we start with a single baseline policy, which is a prior model trained using online RL, which already has a strong performance on this task. We demonstrate the versatility of BC-MAX by iteratively applying BC-MAX on the initial expert, along with all prior policies trained using previous BC-MAX iterations as the next set of baselines. We show that with a limited amount of interaction with the environment (to collect trajectories using each successive set of baselines), we obtain strong policies in a small number of iterations, creating a promising practical recipe for challenging real-world settings.

2. Setting and Related Work

We now define the problem setting formally, and then discuss some lines of prior work which are relevant to this setting.

2.1. Problem setting

Contextual MDP setting. We consider a contextual Markov Decision Process (MDP) with state space $\mathcal{S}$ and action space $\mathcal{A}$. We denote the initial state distribution D_1 and the sampled context (initial state) is $x \sim D$. Once the context is sampled, the transition kernel $\mathbb{P}_x$ is deterministic, that is $\mathbb{P}_x(\cdot|s,a)$ is a point-mass distribution. The reward function is $r_x(s,a)$ and we assume deterministic rewards. We note that both the transition kernel and reward kernel are context-dependent. We will omit the context subscript from our notation whenever it does not introduce ambiguity.

We work in the finite-horizon setting and denote the horizon as H. The value function of a policy π is

$$V_\pi(x) = \sum_{h=1}^{H} \mathbb{E}_{a_h \sim \pi(\cdot|s_h)}[r_x(s_h, a_h)],$$

where s_h is the the state at step h s.t. $\mathbb{P}_x(s_h|s_{h-1}, a_{h-1}) = 1$ and $s_1 \equiv x$. Importantly, the policy class is such that the action distribution at state s depends only on s and not on the context, that is $\pi(\cdot|s,x) = \pi(\cdot|s)$.

Goal. We assume that we are given a set of K baselines policies $\{\pi_i\}_{k\in[K]}$ together with n trajectories for each policy, which we denote by $\{\tau_{i,j}\}_{i\in[K],j\in[n]}$, where $x_j \sim D$ and a trajectory for policy π consists of $\{(s_h, a_h)\}_{h\in[H]}$, with $a_h \sim \pi(\cdot|s_h)$. For an arbitrary policy π and context x we use $\tau_\pi(x)$ to denote the trajectory generated by following π on context x. We also assume that we see the total reward for each trajectory and policy, that is for all $i \in [K], j \in [n]$ we only observe

$$r(\tau_{i,j}) = \sum_{(s_{j,h}, a_{j,h}) \in \tau_{i,j}} r(s_{j,h}, a_{j,h}),$$

instead of observing a dense reward across all the time steps of the trajectory. We assume that the rewards are bounded and non-negative, that is $r(\tau) \in [0, B]$ for all trajectories τ and some constant B.

We have access to a policy class Π and seek to find a find a policy in Π which ideally competes with each of the baselines, and is able to combine their strengths. Let $V_i(x^1) = V_{\pi_i}(x^1)$ denote the expected cumulative reward of baseline i, conditioned on the context x. Then we seek to minimize the regret:

$$\mathrm{Reg}(\pi) := \mathbb{E}_{x\sim D}\left[\max_{i=1}^{K} V_i(x) - V_\pi(x)\right]. \qquad (1)$$

That is, we seek to compete with the best of the baselines for each individual context.

Learning setup. We assume access to the policy class Π, but do not assume any other function approximators, such as for modeling value functions. This is partly due to the fact that the typical training of value functions using Bellman backups is not feasible in our sparse-reward setting. Furthermore, typical actor-critic techniques make strong completeness and realizability assumptions on the value function class, which are not realistic with a restricted notion of state which we encounter in our motivating problem of compiler optimization. This necessitates the development of purely policy-based methods.

2.2. Related work

Vanilla behavior cloning Behavior cloning (Widrow, 1964; Pomerleau, 1988) refers to the approach of learning a policy that matches the mapping from states to actions observed in the data. This is typically solved as a classification problem for deterministic policies, or by maximizing the log-likelihood of the chosen actions in the observed states for stochastic policies. It is unclear how to apply vanilla behavior cloning in the presence of multiple baselines. We will present a natural formulation to behavior clone the best baseline policy per context in the following section.

Value-based improvement upon multiple baselines (MAMBA) Cheng et al. (2020) show how to simultaneously improve upon multiple baseline policies to compete with the best policy *at each state* in the MDP, which is a significantly stronger notion that competing with the best baseline in each context only. However, this comes with two caveats. Their method requires value function estimation for the baselines and access to the MDP to execute trajectories under the learner's policy and/or baselines. Barreto et al. (2020) also study a problem which involves improving over multiple baseline policies, which they title General Policy Improvement. The policy improvement step again requires value function evaluation. We do not assume such access to additional function approximators or the MDP in this work.

Offline RL: Without access to the MDP, a natural approach is to consider offline reinforcement learning, with the data collection policy being a mixture of the baselines π_i, say chosen uniformly. Given the recent results on offline RL to compete with any policy that is covered by the data distribution (Kumar et al., 2020; Xie et al., 2021; Zhan et al., 2022), we can expect a favorable bound on the regret (1), since all the baselines have a good coverage under the uniform data collection policy. However, existing offline RL methods with theoretical guarantees are typically based on function approximation, relying on actor-critic or Q-learning style approaches and on strong credit assignment using per timestep rewards rather than the aggregated reward of a trajectory. Applying these techniques using policy-based function approximation alone and with aggregated reward feedback is not feasible as we argue through a simple lower bound example.

3. Algorithm and Regret Bound

We now describe our algorithm, BC-MAX, and give an upper bound on the regret it incurs to the best per-context baseline.

Algorithm. We describe BC-MAX in Algorithm 1. The basic idea of the algorithm is quite simple. For each context x_j in our dataset, we first choose the trajectory with the highest cumulative reward across all the baselines. Then we use a standard behavior cloning loss to mimic the choice of actions in this trajectory. For a context $x_j, j \in [n]$, we denote $i_j = \text{argmax}_{i\in[K]}\, r(\tau_{i,j})$, and BC-MAX tries to find a policy $\hat{\pi} \in \Pi$ that optimizes the following intuitive objective:

$$\hat{\pi} = \underset{\pi\in\Pi}{\text{argmin}} \sum_{j=1}^{n} \sum_{(s_{j,h},a_{j,h})\in\tau_{i_j,j}} \mathbf{1}(\pi(s_{j,h}) \neq a_{j,h}). \quad (2)$$

Algorithm 1 BC-MAX for cloning best per-context baseline.

```
Input: Base policies $\{\pi_i\}_{i\in[K]}$ and policy class $\Pi$.
Output: Policy $\hat{\pi} \in \Pi$.
  for $j \in [n]$ do
    Sample $x_j \sim D_1$, collect trajectories $\{\tau_{i,j}\}_{j\in[n]}$
    Compute highest reward policy $\pi_{i_j} = \text{argmin}_{i\in[K]} \sum_{(s_{j,h},a_{j,h})\in\tau_{i,j}} r(s_{j,h}, a_{j,h})$.
  end for
  $\hat{\pi} = \text{argmin}_{\pi\in\Pi} \sum_{j=1}^{n} \sum_{(s_{j,h},a_{j,h})\in\tau_{i_j,j}} \mathbf{1}(\pi(s_{j,h}) \neq a_{j,h})$.
```

One natural question at this point might be that if there are two trajectories with very similar high rewards in a context, can it help to leverage this information rather than only picking the one with the higher reward and cloning it. This is indeed a shortcoming of BC-MAX, and other behavior cloning style approaches. However, we note that we only have access to a trajectory-level reward, and hedging between two very different trajectories can create a very noisy learning setup for the algorithm. In situations where value-functions can be feasibly learned, such information is naturally modeled through the value function which assigns similar future rewards to similarly good actions, but we do not find a natural way for incorporating this information in our setup.

Performance guarantee for BC-MAX. We now give a bound on the suboptimality of the policy learned by BC-MAX, relative to the best per-context baseline, in terms of the rewards. The analysis mirrors the standard results for behavior cloning algorithms (Ross & Bagnell, 2010). We begin with a realizability assumption which governs how well the best per-context baseline can be approximated using the learner's policy class Π.

Assumption 3.1. Let $\tau^*(x) = \text{argmax}_{\tau_{\pi_i}(x)}\, r(\tau_{\pi_i}(x))$ be the trajectory with maximum return over all policies $\pi_i, i \in [K]$. There exists $\pi^* \in \Pi$ such that

$$\mathbb{P}_{x\sim D}(\tau_{\pi^*}(x) \neq \tau^*(x)) \leq \epsilon.$$

Here $\mathbb{P}_{x\sim D}(A)$ denotes the probability of an event A under the distribution D, which we recall is the distribution over the contexts x. The assumption is natural as BC-MAX cannot do a good job of approximating the best per-context baseline when no policy in the policy class has a small error in achieving this task. Note that the assumption does not take rewards into account as BC-MAX only matches the actions of $\tau^*(x)$, and does not reason about the reward sub-optimality of other actions, as is common in behavior cloning setups. Indeed this assumption is unavoidable in our problem setting as we illustrate in the next section.

Theorem 3.2. *Under Assumption 3.1, after collecting n trajectories from each of the K base policies, Algorithm 1 returns a policy $\hat{\pi}$ with regret at most*

$$\textit{Reg}(\hat{\pi}) \le O\left(\epsilon H + \frac{H^2 \log(H|\Pi|/\delta)}{n}\right),$$

with probability at least $1-\delta$.

Proof. Recall the definitions of $\tau^*(x)$ from Assumption 3.1, and let $\pi^* = \operatorname{argmin}_{\pi\in\Pi} \mathbb{E}_{x\sim D}(\sum_{h=1}^H \mathbf{1}(\pi(s(x)) \ne a(x)))$ where $(s_h(x), a_h(x))_{h=1}^H = \tau^*(x)$ form the best trajectory for x among the baseline policies. Under Assumption 3.1, we know that $\mathbb{E}_{x\sim D}(\sum_{h=1}^H \mathbf{1}(\pi(s(x)) \ne a(x))) \le \epsilon H$. Let us define

$$\hat{A}(\pi) = \sum_{j=1}^n \sum_{h=1}^H \mathbf{1}(\pi(s_{j,h}) \ne a_{j,h}),$$
$$A(\pi) = \mathbb{E}_x \left(\sum_{h=1}^H \mathbf{1}(\pi(s_h(x)) \ne a_h(x))\right).$$

Clearly we have that $\mathbb{E}[\hat{A}(\pi)] = A(\pi)$ for any fixed policy π, and $\hat{A}(\pi) = \sum_{j=1}^n Z_j(\pi)$ with $Z_j(\pi) = \sum_{h=1}^H \mathbf{1}(\pi(s_{j,h}) \ne a_{j,h}) \ge 0$. We note that the Z_j are i.i.d., with $\mathbb{E}[Z_j(\pi)] = A(\pi)$ and $\mathbb{E}[Z_j(\pi)^2] \le H\,\mathbb{E}[Z_j(\pi)] = H\,A(\pi)$. Then by Bernstein's inequality combined with a union bound over policies, we have with probability at least $1-\delta$, for all $\pi \in \Pi$:

$$|\hat{A}(\pi) - nA(\pi)| \le \sqrt{nH\,A(\pi)\log(2|\Pi|/\delta)} + H\log(2|\Pi|/\delta)$$
$$\le \frac{nA(\pi)}{2} + \frac{3}{2}H\log(2|\Pi|/\delta).$$

Applying the inequality with $\pi = \hat{\pi}$ and $\pi = \pi^*$, we obtain

$$A(\hat{\pi}) \le \frac{2}{n}\hat{A}(\hat{\pi}) + \frac{3H\log(2|\Pi|/\delta)}{n}$$
$$\frac{1}{n}\hat{A}(\pi^*) \le \frac{3}{2}A(\pi^*) + \frac{3}{2}\frac{H\log(2|\Pi|/\delta)}{n}.$$

Scaling the second inequality by 2 and adding them yields

$$A(\hat{\pi}) \le 3A(\pi^*) + \frac{6H\log(2|\Pi|/\delta)}{n} \le 3\epsilon + \frac{6H\log(2|\Pi|/\delta)}{n}, \tag{3}$$

where the second inequality follows by Assumption 3.1.

Now we note that for any policy π:

$$\mathrm{Reg}(\pi) = \mathbb{E}_x[\max_i V_i(x) - V_\pi(x)] = \mathbb{E}_x[r(\tau(x)) - r(\tau_\pi(x))]$$
$$\le \sum_{h=1}^H (H-h)\,\mathbb{E}_x[\mathbf{1}(\pi(s_h(x)) \ne a_h(x))|].$$

Plugging the bound from Equation 3 into the inequality above completes the proof. □

Implementation details In practice we can not directly compute $\hat{\pi}$ as defined in Equation 2. Instead we solve a proxy to the optimization problem by replacing the indicator function $\mathbf{1}(\pi(s_{j,h}) \ne a_{j,h})$ by the cross-entropy loss. Let $y_{j,h} \in \{0,1\}^A$ be the indicator with only entry equal to 1 the one which corresponds to the action $a_{j,h}$), and all other entries equal to 0. Further, we assume that all $\pi \in \Pi$ are such that $\pi(s) \in \Delta^{A-1}$, that is each $\pi(s)$ represents a distribution over the actions that policy π plays when in state s. We then use a first order method to minimize the loss

$$\min_{\pi\in\Pi} \sum_{j=1}^n w_j \sum_{h=1}^H \sum_a -y_{j,h}(a)\log(\pi(S_{j,h})),$$

where $w_j \in [0,1]$ are example weights which we find helpful in our practical implementation. We refer the reader to our experimental evaluation for mode details on how the weights are induced.

4. Lower bounds

In this section, we show a series of lower bounds which illustrate the necessity of various aspects of our guarantee in Theorem 3.2. We start with the necessity of Assumption 3.1

Necessity of Assumption 3.1 Let us consider a contextual multi-armed bandit problem, meaning that we fix $H = 1$. For any ϵ, we choose the context space $\mathcal{S} = [M]$ for $M = \lceil \frac{1}{\epsilon} \rceil$, and choose D to be the uniform distribution on $\mathcal{S}$. We fix $\mathcal{A} = \{a_1, a_2, a_3\}$, and $K = 1$ with the data collection policy π_1 choosing $a = a_1$ for each context $x \in \mathcal{S}$. We consider two possible environments, defined through rewards r_1, r_2. For a_1, we have $r_1(x, a_1) = r_2(x, a_2) = 1$. For the other two actions, we have $r_1(x, a_2) = 0$ and $r_1(x, a_3) = 1$, while the second environment has $r_2(x, a_2) = 1$ and $r_2(x, a_3) = 0$. We design a policy class π with two policies $\{\pi_1, \pi_2\}$ such that $\pi_1(x) = \pi_2(x) = a_1$ for all $x \ne 1$ and $\pi_1(1) = a_2$m $\pi_2(1) = a_3$. Clearly this policy class satisfies Assumption 3.1. But it also contains an optimal policy for both the rewards r_1, r_2 with a regret equal to 0. However, since the data contains no information about which one of r_1 or r_2 generated the data, the best we can do is to pick between

π_1 and π_2 uniformly at random, and incur a regret of at least 0.5ϵ. This argument shows that we cannot replace the $0-1$ loss for measuring the accuracy of a policy in Assupmtion 3.1, with a more reward-aware quantity. It is also evident from the example that we cannot avoid incurring a regret of $\Omega(\epsilon)$.

Necessity of the comparator choice As discussed earlier, we define regret relative to the best baseline per-context, but the broader literature on offline RL allows stronger benchmarks, such as the best policy covered by the data generating policies. To understand the difference between the two, we consider the case of $H=2$ and $K=2$, with $\mathcal{S}=\{x\}$ being a singleton. Suppose we have two actions $\mathcal{A}=\{a_1,a_2\}$ and the baselines choose the trajectories $\pi_1(x)=a_1$ and $\pi_2(x)=a_2$ at both $h=1,2$. There are two possible reward functions given by $r_1((a_1,a_1))=r_1((a_2,a_2))=0.5$, $r_1((a_1,a_2))=1$, $r_1((a_2,a_1))=0$ and $r_2(\tau)=1-r_1(\tau)$ for all trajectories τ. Now we observe that in the sense of the coverage studied in the offline RL literature, all four trajectories are covered by the dataset, since we get to observe both the actions at both the steps in the episode. However, since the data contains no useful information to distinguish between r_1 and r_2, no learning method can pick a covered policy which is better than our benchmark of best baseline per-context. This challenge arises in our scenario as we only observe the aggregate reward over a trajectory, which makes per-step credit assignment used in standard offline RL methods through the use of Bellman errors infeasible.

Tightness of horizon factor The tightness of the horizon factor follows from a simple reduction to Theorem 6.1 in Rajaraman et al. (2020), who show that in a finite horizon episodic MDP, there is no algorithm which only observes n optimal policy trajectories and returns a policy with regret better than $\Omega\left(\frac{SH^2}{n}\right)$. The MDP contructed by Rajaraman et al. (2020) for the lower bound has a non-deterministic transition kernel $\mathbb{P}$. Since we are assume that the transitions are deterministic, we instead use the randomness in sampling contexts, $x\sim D$ to simulate $\mathbb{P}$. Concretely, let $\xi_1,\ldots,\xi_H$ be the random bits sampled at the H steps in a fixed episode from the construction in Rajaraman et al. (2020). We define $x=(\xi_1,\ldots,\xi_H)$ and set $\mathbb{P}_x(s_{h+1}|s_h,a_h)=\mathbb{P}(s_{h+1}|s_h,a_h,\xi_h)$ to be the trasition taken for the realization of ξ_h. The size of the state space of contextual MDP with the above transition kernel can be set to $S=\Theta(\log_2(|\Pi|)$ and the action space to $\mathcal{A}=\{a_1,a_2\}$, so that each policy in Π is encoded by how it acts on the state space. This construction directly leads to the following lower bound.

Theorem 4.1. *For any number of samples n there exists a family of contextual MDPs with disribution over contexts given by D_1, such that the policy $\hat{\pi}=\mathbb{A}(\{\tau_{\pi^*}(x_i)\}_{i=1}^n)$ returned by any algorithm $\mathbb{A}$ satisfies*

$$\mathbb{E}_x[V_{\pi^*}(x)]-\mathbb{E}_x[V_{\hat{\pi}}(x)]\geq\min\left\{H,\frac{\log_2(|\Pi|)H^2}{n}\right\}.$$

5. Case study: Optimizing a compiler's inlining policy

5.1. The inlining for size problem

In short the inlining problem which we study in our experiments consists of deciding to inline or not to inline a callsite in a program with the goal of minimizing the size of the final program binary. We omit most compilation details and just give a brief overview which should be sufficient for understanding the problem from a RL perspective. In our specific scenario, compilation is split into a frontend (fe) and a backend (be). In our setup the frontend consists of translating the program into an Intermediate Representation (IR), doing some frontend optimizations, then a (thin) link step follows, which re-organizes functions in the various modules to improve inlining opportunities. For more details on the linking step see Johnson et al. (2017). The backend compilation follows the thin link step and is applied on the updated modules. It consists of further optimizations, including inlining decisions, final linking and lowering the IR to machine code, e.g., x86, ARM, etc. The IRs with which our RL algorithms work with are post frontend linking and pre backend optimization, that is we work only on backend optimizations. We note that a program is made up of multiple *modules*. In the fe, a module corresponds to a single C/C++ source file, after all the preprocessor directives have been applied. In the be, the module would consist of a mix of IRs from different fe modules. The inlining decisions will be taken at callsites in the IRs of each module, where the callee is also in the same module. We note that each module is ultimately compiled to a machine code-specific binary, with its own size, that will still need to be linked into the final executable. Hence we treat each module as a context x in our contextual MDP setting, and the value $V_i(x)$ of the baseline policy π_i is the size of the binary we get for module x, when we make inlining decisions for the module according to π_i.

$$\begin{aligned}&\texttt{Program}\rightarrow\texttt{IRs}\rightarrow\textbf{\texttt{fe optimizations}}\\&\rightarrow\texttt{ThinLinking}^{1}\xrightarrow{\texttt{Collecting IRs}}\textbf{\texttt{be optimization}}\\&\rightarrow\texttt{Final linking}\rightarrow\texttt{x86}\end{aligned}\tag{4}$$

In Equation 4 we outline the compilation process, together with the step at which we collect IRs from the respective modules to be used in our RL algorithms. It is important to note that the learned RL policy will make inlining decisions both at the fe optimization and be optimization parts in

[1]See Johnson et al. (2017)

Equation 4, however, the IRs for training are only collected after the fe optimization step.

The contextual MDP setting can now be tied together with the compilation process as follows. Each context is a module as mentioned above, with state-space defined by its IR. Each state corresponds to a callsite in the IR of the module x, and the action set is $\{\texttt{inline, don't inline}\}$ or $\{1, 0\}$ respectively. Each π_i is some base inlining policy and $V_i(x)$ is defined by the size of the compiled stand-alone module x. It is important to note that there is a mismatch between trying to maximize $\mathbb{E}_{x\sim D}[V_\pi(x)]$ and the overall goal of minimizing the binary size, as it is not necessarily true that the sum of module sizes equals the size of the binary. In fact, part of the post-inlining be and linker optimizations may introduce a significant distribution shift between the sum of module sizes and the size of the final binary. In our experiments, we try to minimize this distribution shift by turning off certain optimizations. For more details on the compilation pipeline we refer to Trofin et al. (2021).

We note that the entire process is fully deterministic, as we assumed in our theoretical setup, since the compiler is a deterministic program.

5.2. Dataset collection

We train and evaluate on two sets of binaries. In the first experiment we train on a proprietary search binary and evaluate the model on a different proprietary set of targets that are part of a cloud systems infrastructure. These targets need to be installed on a fixed size partition of each cloud machine and hence are size-constrained. In the second experiment we train and evaluate on the Chrome binary on Android. Training proceeds in two separate steps, which we repeat over several iterations. The two steps can be summarized as follows, first we collect a training dataset which consists of trajectories with smallest size over all base policies available at the current iteration. Next, we train a new base model using the objective defined in Equation 2. This conceptually applies Algorithm 1 repeatedly, where the set of baseline policies is updated at each iteration to include the new policy obtained from the previous iteration. We now describe each step carefully.

Training dataset collection. The dataset collection begins by creating a corpus of IRs of modules which make up the final binary. The corpus creation follows the work of Trofin et al. (2021); Grossman et al. (2023) and uses tools for extracting the corpus are available on GitHub[2]. The corpus is created at the beginning of training and remains the same throughout every iteration. Training begins under the assumption that there exists at least one base policy. In the first iteration a training dataset is collected from this initial base policy π_1, next, π_1 is behavior cloned by solving the optimization problem in Equation 2. We specify how the weights for the objective are computed in the following sections as they are different for the different targets. Let $\hat{\pi}_2$ denote the resulting policy after solving Equation 2. This policy is non-deterministic and so we construct the base policy π_2 by setting $\pi_2(s) = \text{argmax}_{a\in\{0,1\}} \hat{\pi}_2(s, a)$, that π_2 always plays the most likely action according to $\hat{\pi}_2$. This concludes the first iteration. More generally, if we have a larger initial set of baseline policies than just a singleton$\{\pi_1\}$, the iterations proceed similarly. However, instead of just using π_1, we use the full set $\{\pi_i\}_{i\in[K]}$ of baseline policies at every iteration at the first iteration.

Proceeding this way, at the t-th iteration the set of base policies is taken as a subset of $\{\pi_1, \ldots, \pi_{t-1}\}$ which always contains π_1 (or the larger set of all initial baselines). Then we again invoke BC-MAX with these baseline policies, and obtain a new randomized policy $\hat{\pi}_t$, and we refer to π_t as the corresponding deterministic greedy policy. When collecting a new training dataset we not only collect trajectories with the chosen subset of base policies but we also may force exploration by using $\hat{\pi}_{t-1}$ in the way discussed next.

Exploration in training dataset collection. For a fixed module x and a policy π, we choose a ceiling on the number of exploration steps as a hyper parameter, which is a fraction of the length of the trajectory $|\tau_{\pi_1}(x)|$. The call-sites at which exploration occurs are selected as follows. The first exploration call-site is selected as $\tilde{h} = \text{argmin}_h\{|\hat{\pi}(s_h)(0) - \hat{\pi}(s_h)(1)|\}_{S_h\in\tau_\pi(x)}$, as the call-site where the exploration policy $\hat{\pi}$ is the least confident about the action to choose. The exploration step is then played at $s_{\tilde{h}}$ by taking the action $1-\pi(s_{\tilde{h}})$ (recall that $\pi(s) \in \{0, 1\}$), and the remaining steps in the trajectory are completed by playing according to π. Let $\hat{\tau}$ denote the trajectory from the last round of exploration. In the following exploration round the exploration step is selected as the step h at which the gap, $|\hat{\pi}(s_h)(0) - \hat{\pi}(s_h)(0)|$, is smallest among all $h > \tilde{h}$, where $\tilde{h}$ is the exploration step in the previous round. Once the maximum number of exploration rounds is reached or the exploration step reaches the end of the trajectory, we return the trajectory which results in the smallest module size among all explored trajectories. Pseudo-code is presented in Algorithm 2. The exploration strategy is governed by $\hat{\pi}_{t-1}$, however, it can be updated by using the non-deterministic policy $\hat{\pi}$ which induces π. We leave such approaches as future work, as we have already observed significant benefit of using only $\hat{\pi}_{t-1}$ as the exploration policy.

Online versus offline learning. Our theoretical setup frames the problem in an offline learning scenario, yet Algo-

[2]A detailed example can be found at `https://github.com/google/ml-compiler-opt/blob/main/docs/inlining-demo/demo.md`

Algorithm 2 Explore module

```
Input: Base policy $\pi$, exploration policy $\hat{\pi}$, module $x$, maximum exploration steps $T$.
Output: Compilation trajectory $\tau_\pi(x)$ with reward $r_{\pi,x}$.
  Compute vanilla trajectory $\tau_\pi(x)$ by compiling with $\pi$ and receive reward $r^1_{\pi,x}$
  $t = 1$
  $\hat{\tau}_1 = \tau_\pi(x)$
  $\tilde{h}_1 = \text{argmin}_h\{|\hat{\pi}(S_h)(0) - \hat{\pi}(S_h)(1)|\}_{S_h \in \hat{\tau}_1}$
  while $t \leq T$ do
    Replay $\hat{\tau}_t$ until $\tilde{h}_t$
    Play $1 - \pi(S_{\tilde{h}_t})$ at $\tilde{h}_t$
    Complete trajectory $\hat{\tau}^{t+1}$ by playing $\pi$
    Receive reward $r^{t+1}_{\pi,x}$
    if $\tilde{h}_t < |\hat{\tau}_{t+1}|$ then
      $\tilde{h}_{t+1} = \text{argmin}_{h > \tilde{h}_t}\{|\hat{\pi}(S_h)(0) - \hat{\pi}(S_h)(1)|\}_{S_h \in \hat{\tau}^{t+1}}$
    else
      break
    end if
  end while
  $t^* = \text{argmax}_t\, r^t_{\pi,x}$, $r_{\pi,x} = r^{t^*}_{\pi,x}$, $\tau_\pi(x) = \hat{\tau}_{t^*}$
```

rithm 2 and the iterative procedure do rely on our ability to interact with the environment in an adaptive manner. Note, however, that the modality of interaction used in our approach is quite different, and significantly more practical than full-fledged online RL. Each round of policy learning, which happens using BC-MAX, is fully offline. This process, which involves a large number ($10^5 - 10^6$) stochastic gradient steps, happens without any interaction with the environment, and is where the bulk of the learning happens. Subsequently, we form a new data collection policy for the next iteration, and this policy is applied to collect one trajectory per module. The data collection process does not involve any policy updates, and hence is massively parallelizable with no interlocking bottlenecks with the learning process. In online RL, on the other hand, data collection and policy updating go hand-in-hand, which typically requires significantly more complex architecture (Mnih et al., 2016) to scale to domains where data collection is expensive. Our approach, on the other hand, simply requires interleaving standard supervised learning and batch data collection, which is quite desirable especially in the compiler application, where the ML training happens on GPUs, while the compilation happens on CPU machines.

5.3. Search application targets

Similarly to Trofin et al. (2021) we collect a corpus for training purposes from a search application binary with approximately 30000 modules. The initial base policy is an RL model trained using an Evolutionary Strategy (ES[3]) as in Trofin et al. (2021). After collecting a training dataset with the ES policy we noticed that the distribution of sizes of modules is fairly non-uniform, with few modules having very large sizes or very small sizes and majority of modules being somewhere in-between. Because we expect that the actions of the behavior cloning policy taken on larger size modules are more important for size saving we upweight the actions in such trajectories. The weights used in training are computed as follows. Let $\texttt{size}(x, \pi_1)$ denote the size of module x from the collected trajectory under policy π_1 (or in the case of multiple baseline policy under the best baseline policy). The modules are partitioned into buckets according to their sizes where the limits of the buckets are taken to be on exponentially scaling grid, that is the first bucket contains all modules with size $\texttt{size}(x, \pi_1) \in [0, 2^0)$, the second bucket all modules such that $\texttt{size}(x, \pi_1) \in [2^0, 2^1)$ etc., up to the final bucket with size $[2^{M-1}, 2^M)$. Let $b_m = \{x : \texttt{size}(x, \pi_1) \in [2^{m-1}, m)\}$ denote the m-th bucket and let $m(x)$ be the m for which $x \in b_{m(x)}$. The weight w_x for module x is computed as follows.

$$w_x = \frac{\max_m |b_m|}{|b_{m(x)}|}.$$

Algorithm 3 BC-MAX for cloning best per-context baseline with exploration

```
Input: Base policy $\pi_1$ and policy class $\Pi$. Max exploration steps $T$. Max number of iterations $N$.
Output: Policy $\hat{\pi} \in \Pi$.
  $l = 1$
  while $l \leq N$ do
    for $j \in [n]$ do
      Sample $x_j \sim D_1$
      for $i \in \{\pi_s\}_{s \leq l}$ do
        $r_{i,j}, \tau_{i,j} = \texttt{Algorithm 2}(\pi_i, \hat{\pi}_\ell, x_j, T)$
      end for
      Compute highest reward policy $\pi_{i_j} = \text{argmin}_{i \in [K]} \sum_{(s_{j,h}, a_{j,h}) \in \tau_{i,j}} r_{i,j}(s_{j,h}, a_{j,h})$.
    end for
    Compute module weights $\{w_j\}_{j \in [n]}$ (See Sect. 5.3,5.4).

    $$\hat{\pi}_l = \underset{\pi \in \Pi}{\text{argmin}} -w_j \sum_{j=1}^{n} \sum_{(s_{j,h}, a_{j,h}) \in \tau_{i_j,j}} a_{j,h} \log(\pi(a_{j,h}|s_{j,h}))$$

    $\pi_l(s) = \text{argmax}_a\, \hat{\pi}_l(a|s), \forall s \in S$
  end while
```

[3]The policy can be found here: https://github.com/google/ml-compiler-opt/releases/tag/inlining-Oz-v1.1

We train two sets of policies, one set is trained without exploration and is precisely in line with Algorithm 1. For full pseudo-code, which includes the exploration step, we refer the reader to Algorithm 3. The second set is trained with exploration as described in Section 5.2. In Figure 1 we

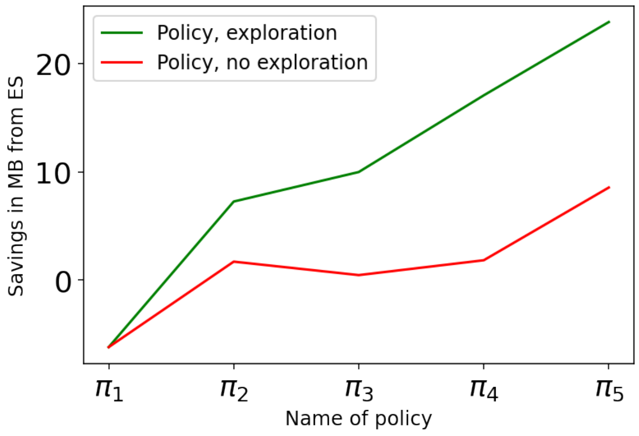

Figure 1. Savings in MB from ES on training binary

show savings of the trained policies to π_1, which is the ES policy, on the search binary from which the training dataset is collected. In Figure 2 we show the savings on a *different test binary*. On the x-axis of the figures we show the size savings of the policy π_i learned at each iteration i, with and without exploration respectively, where bc_0 is the behavior cloned policy from ES. Both figures demonstrate the success of our approach in improving significantly beyond the initial baseline, as well as the benefits from multiple iterations of the process. Furthermore, the gap between the lines with and without exploration highlights the benefits of the added exploration.

We note that the compilation for both the training and test binaries is carried out in the following way to minimize the distribution shift – the fe optimizations are carried out by ES, while the be optimizations are carried out by the trained policies. If we were to use the trained policies in both fe and be, this might lead to significant distribution shift, as Algorithm 1 works only on trajectories collected after the fe optimizations for which ES is always used. That is, if any of the trained policies, bc_i, act very differently on the fe, compared to ES, the resulting IRs before the be optimization might be completely different from the training set IRs, and hence the trained policy might take very sub-optimal actions.

5.4. Chrome on Android

In our second set of experiments we train an RL policy for Chrome on Android. The training and test binaries are the

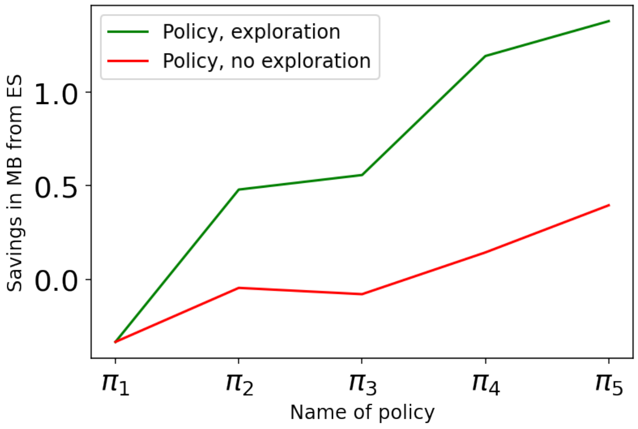

Figure 2. Savings in MB from ES on test binary

same in this case. The base policy with which we start is an RL policy trained using Proximal Policy Optimization (PPO[4]) (Schulman et al., 2017) as done in Trofin et al. (2021). There are two differences in training from Section 5.3. First, we focus only on the setting where we do exploration. The second difference in training is how the weights for the objective in Equation 2 are formed. The approach for computing the weights used here is inspired by the fact that we want to improve on PPO in each module and not just on the sum of module sizes. That is we want to maximize the size savings over the worst case module in our dataset. The following approach is natural when such max-min guarantees are desired.

Reusing notation from Section 5.3 we let

$$p_x^1 = \frac{|b_{m(x)}|}{\sum_m |b_m|}$$
$$w_x^1 = \frac{\max_m p_x^1}{p_{m(x)}^1},$$

be the weights in the first iteration of training. In following iterations the weights are set as $w_x^t = \frac{\max_m p_x^t}{p_{m(x)}^t}$, where p^t is update using the Hedge algorithm (Littlestone & Warmuth, 1994). The update uses the sum of sizes in each bucket as losses, normalized by the ℓ-infinity norm, that is

$$\tilde{L}_m^t = \sum_{x \in b_m} \texttt{size}(x, \pi_t)$$
$$L_m^t = \frac{\tilde{L}_m^t}{\|L^t\|_\infty},$$

[4]The policy can be found here: `https://commondatastorage.googleapis.com/chromium-browser-clang/tools/mlgo_model2.tgz`

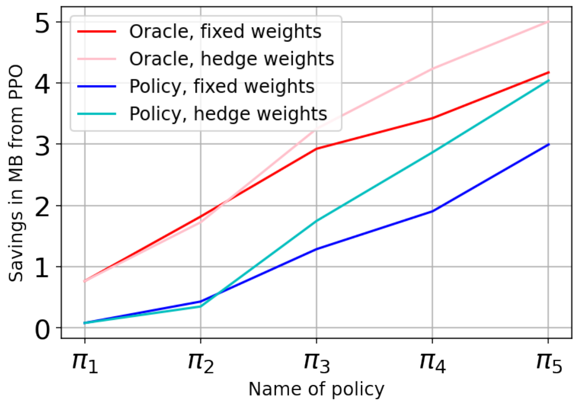

Figure 3. Savings in MBs from PPO on sum of module sizes

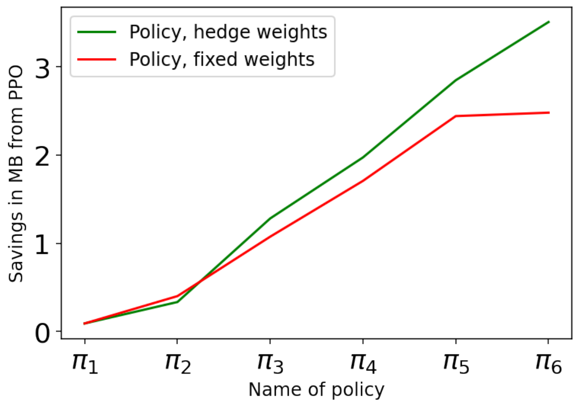

Figure 4. Savings in MBs from PPO on binary size

where L_m^t denotes the m-th coordinate of the loss vector L^t. The Hedge update is then

$$\tilde{p}_m^{t+1} = p_m^t \exp(-\eta L_m^t)$$
$$p_m^{t+1} = \frac{\tilde{p}_m^t}{\sum_m \tilde{p}_m^t}.$$

In Figures 3 and 4, we plot the savings of our learned policies across iterations, relative to the initial PPO policy, measured in two different ways. For Fig 3, we simply add up the sizes of the binaries produced by compiling each module in our training dataset. This is a clean metric, as the distribution shift between training and evaluation is small, and no artifacts from linker or post-inlining `be` optimizations are introduced in the evaluation. As we see, we improve rapidly beyond the PPO policy with the iterative applications of BC-MAX. Note that even the sum of module sizes suffers from the typical distribution shift between online and offline RL, since the data used from behavior cloning is collected using a different policy than the one we apply in evaluation. For the sum of module sizes metric, we can study the effect of this distribution shift rather carefully by also compiling with an *oracle* policy, which simply chooses the best baseline policy for each module, which is the target for training in BC-MAX. This oracle, shown in red in Figure 3 naturally provides larger gains relative to PPO than our learned policy as expected, but the gap reduces through the iterations of our process, indicating that the policies tend to stabilize through iterations, and the training data for later applications of BC-MAX is closer to on-policy data. We note that the oracle changes between different instantiations of our weights. This is because the i-th trained policy π_i depends on the choice of weights and so the oracle after the i-th iteration which chooses the best among $\{\pi_i\}_{\ell=1}^i$ also depends on the choice of weights. We also note that the gap between the learned and oracle policy's performance is smaller when we use the Hedge weights, and that the weighted version has a bigger gain relative to PPO, showing the efficacy of this approach.

Finally, in Figure 4 we present the savings in size of the Chrome on Android binary, which is the actual yardstick. Here we cannot easily evaluate the size of the oracle, so we only compare our policies to PPO, and again observe impressive gains, with the Hedge-weighted variant doing better. The binary size when compiled with the PPO policy is approximately 213.32 MB.

Acknowledgements

We would like to thank Ziteng Sun for great discussions on the lower bound.

References

Abbeel, P. and Ng, A. Y. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, pp. 1, 2004.

Barreto, A., Hou, S., Borsa, D., Silver, D., and Precup, D. Fast reinforcement learning with generalized policy updates. *Proceedings of the National Academy of Sciences*, 117(48):30079–30087, 2020.

Cheng, C.-A., Kolobov, A., and Agarwal, A. Policy improvement via imitation of multiple oracles. *Advances in Neural Information Processing Systems*, 33:5587–5598, 2020.

Cheng, C.-A., Xie, T., Jiang, N., and Agarwal, A. Adversarially trained actor critic for offline reinforcement learning. In *International Conference on Machine Learning*, pp. 3852–3878. PMLR, 2022.

Ernst, D., Geurts, P., and Wehenkel, L. Tree-based batch mode reinforcement learning. *Journal of Machine Learning Research*, 6, 2005.

Grossman, A., Paehler, L., Parasyris, K., Ben-Nun, T., Hegna, J., Moses, W., Diaz, J. M. M., Trofin, M., and Doerfert, J. Compile: A large ir dataset from production sources. *arXiv preprint arXiv:2309.15432*, 2023.

Haarnoja, T., Zhou, A., Hartikainen, K., Tucker, G., Ha, S., Tan, J., Kumar, V., Zhu, H., Gupta, A., Abbeel, P., et al. Soft actor-critic algorithms and applications. *arXiv preprint arXiv:1812.05905*, 2018.

Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., Horgan, D., Quan, J., Sendonaris, A., Osband, I., et al. Deep q-learning from demonstrations. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

Ho, J. and Ermon, S. Generative adversarial imitation learning. *Advances in neural information processing systems*, 29, 2016.

Johnson, T., Amini, M., and Li, X. D. Thinlto: scalable and incremental lto. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 111–121. IEEE, 2017.

Kumar, A., Zhou, A., Tucker, G., and Levine, S. Conservative q-learning for offline reinforcement learning. *Advances in Neural Information Processing Systems*, 33: 1179–1191, 2020.

Littlestone, N. and Warmuth, M. K. The weighted majority algorithm. *Information and computation*, 108(2):212–261, 1994.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pp. 1928–1937. PMLR, 2016.

Pomerleau, D. A. Alvinn: An autonomous land vehicle in a neural network. *Advances in neural information processing systems*, 1, 1988.

Rajaraman, N., Yang, L., Jiao, J., and Ramchandran, K. Toward the fundamental limits of imitation learning. *Advances in Neural Information Processing Systems*, 33: 2914–2924, 2020.

Ross, S. and Bagnell, D. Efficient reductions for imitation learning. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pp. 661–668. JMLR Workshop and Conference Proceedings, 2010.

Ross, S., Gordon, G., and Bagnell, D. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 627–635. JMLR Workshop and Conference Proceedings, 2011.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. Deterministic policy gradient algorithms. In *International conference on machine learning*, pp. 387–395. Pmlr, 2014.

Song, Y., Zhou, Y., Sekhari, A., Bagnell, J. A., Krishnamurthy, A., and Sun, W. Hybrid rl: Using both offline and online data can make rl efficient. *arXiv preprint arXiv:2210.06718*, 2022.

Trofin, M., Qian, Y., Brevdo, E., Lin, Z., Choromanski, K., and Li, D. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808*, 2021.

Widrow, B. Pattern recognition and adaptive control. *IEEE Transactions on Applications and Industry*, 83(74):269–277, 1964.

Xie, T., Cheng, C.-A., Jiang, N., Mineiro, P., and Agarwal, A. Bellman-consistent pessimism for offline reinforcement learning. *Advances in neural information processing systems*, 34:6683–6694, 2021.

Zhan, W., Huang, B., Huang, A., Jiang, N., and Lee, J. Offline reinforcement learning with realizability and single-policy concentrability. In *Conference on Learning Theory*, pp. 2730–2775. PMLR, 2022.