IEOR 4575 Final Project

Integer Programming with Proximal Policy Optimization and Random Network Distillation

Thomas Tran[*]

April 24, 2021

Abstract

Integer programming, an optimization problem used in a variety of applications, is a well known and actively researched NP-Hard problem. Although there exist no algorithms that can solve all integer programs in polynomial time, there are many methods to determine an efficient method to search. Cutting plane methods incrementally add constraints to reduce the feasible region of the optimal solution; however, choosing which constraints, or cuts, is a nontrivial problem. Tang, Agrawal, & Faenza (2020)[6] frame the process of choosing cuts from a reinforcement learning (RL) perspective.

We extend off of their work, proposing an actor-critic policy gradient method with Proximal Policy Optimization (PPO) and Random Network Distillation (RND) to encourage high sample efficiency and sufficient exploration throughout the instances. Furthermore, we examine the model's performance compared to other proposed policy gradient methods on small instances, and train on larger instances. The trained models are then evaluated on large unseen instances to analyze the generalizability of the models. We show that this policy gradient method is a promising approach to determine cuts with RL.

1 Introduction

The topic of linear programming (LP) discusses the mathematical procedure to choose a set of decision variables that maximize or minimize a linear objective function with respect to a set of linear constraints. The standard form of a linear function with n decision variables and k constraints is often expressed as $\arg\min_x c^T x$ such that $Ax \leq b, \quad x \geq 0$. where $c, x \in \mathbb{R}^n$, $A \in \mathbb{R}^{k \times n}$, $b \in \mathbb{R}^k$.

Often used in real-world optimization problems, linear programming is a computationally efficient method; any linear program can be solved in a polynomial amount of time. However, many applications require our decision variables to be integers, placing the additional constraint of $x \in \mathbb{Z}^n$. This seemingly simple constraint is deceptive: determining the optimal solution to all integer programming problems is placed in the set of NP-Hard problems, and can no longer be solved in polynomial time.

As a result, many heuristic approaches have been proposed to determine the optimal solution in a reasonable amount of time. Many commercial integer programming solvers use cutting plane methods – these methods relax the integer programming solution back into a linear program, and incrementally add linear constraints, or *cuts*, over T timesteps to the preexisting k constraints, while maintaining the optimal integer programming solution within the constraints. One approach to the cutting plane method is Gomory's algorithm [2], which can solve any integer program in a finite amount of time. However, given a set of proposal cuts at a timestep t, determining a "good" cut is difficult.

Consequently, Tang et. al. (2020)[6] propose a reinforcement learning (RL) approach, treating the Gomory algorithm as a Markov Decision Process (MDP) on the basis that an agent can effectively select cutting planes within a limited time T, and generalize well across instance sizes of varying n and k. We build upon this approach, empirically analyzing various policy gradient methods, and suggest an actor-critic approach with Proximal Policy Optimization [3] and Random Network Distillation [1] to train with high stability and sample efficiency.

[*]Columbia University (Email: tkt2120@columbia.edu)

2 Problem Formulation

This section is dedicated to describing cutting plane methods in a reinforcement learning framework, and the various policy gradient methods explored.

2.1 Cutting Plane Method as a Markov Decision Process

Formally, an MDP is defined as a set of states $\mathcal{S}$ with transition probabilities $\mathcal{P} : s_t, a_t \to p(s_{t+1})$, through which an agent makes actions $a_t \in \mathcal{A}$ in timestep t using a policy $\pi : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, receiving a reward for the transition $r_t \in \mathbb{R}$. Within a reinforcement learning framework, the objective is to determine a policy π^* that maximizes the discounted reward $\mathbb{E}_\pi[\sum_{t=1}^T \gamma^{t-1} r_t]$ over time horizon T and discount factor γ.

The approach to modifying a cutting plane method to a Markov Decision Process (MDP) is similar to that of Tang et. al. (2020)[6], where each state s_t is composed of the objective function and the set of constraints, and the actions a_t are the set of additional constraints (Gomory cuts) available at timestep t. More formally, $\mathcal{S} = \{s_t : Ax \le b\}$, $\mathcal{A} = \{a_t \in \mathcal{A}^{(t)} : Ex \le d\}$.

The state and action space at timestep t are linearly scaled as a type of observation normalization. The reward is formulated as the difference between the LP optimal value at timestep $t+1$ and t, or formally, $r_t = c^T x_{LP}^{t+1} - c^T x_{LP}^t$. This formulation presents various challenges not present in most RL problems: as cutting planes are chosen, the state space increases, and at each timestep, the actions available varies.

2.2 Network Architecture

This section describes the RL framework behind our work. A pseudocode outline of our approach is defined in Algorithm 1.

Algorithm 1: Cutting with PPO and RND

```
Input      : Environment Instances env
Output     : An objective value $f(p)$
Parameters: Iterations $N$, Number of
            Rollouts $K$, Epochs $E$, Model
            Parameters $(\pi_\theta, V^\pi, \mathcal{N}_T, \mathcal{N}_P)$

Function Parallel-Rollout($p$):
    for $t$ = 1, . . . , $T$ do
        $a_t \sim \pi_\theta(s_t)$
        $R_t^{(E)}, s_{t+1}$ = env.step($s_t$, $a_t$)
        $R_t^{(I)} = \|\mathcal{N}_P(s_{t+1}) - \mathcal{N}_T(s_{t+1})\|^2$
    Discount Rewards $R^{(E)}$, $R^{(I)}$

Function Train-PPO-RND($p$):
    for $i$ = 1, . . . , $N$ do
        Initialize RolloutMemory $\mathcal{M}$
        for $k$ = 1, . . . , $K$ do
            Add Parallel-Rollout($p$) to $\mathcal{M}$
        Update $\mathcal{N}_P$ with $\mathcal{M}$
        for $e$ = 1 . . . , $E$ do
            $\forall s_t, a_t \in \mathcal{M}$, Calculate $A^\pi(s_t, a_t)$
            Update $\pi_\theta$, $V^\pi$ with $L^{CLIP}$
```

2.2.1 Deep Policy Gradient

Within a finite horizon MDP, the objective of policy gradient methods is to determine a policy π_θ (parameterized by θ) that maximizes expected discounted reward $\rho(\theta) = \mathbb{E}_{\pi_\theta}[\sum_{t=1}^T \gamma^{t-1} r_t]$. To maximize this expression, the gradient of $\rho(\theta)$ is taken and gradient ascent is performed:

$$\nabla_\theta \rho(\theta) = \mathbb{E}_{\pi_\theta}[Q^{\pi_\theta}(s_t, a_t) \nabla_\theta \log \pi_\theta(a_t|s_t)]$$
$$\theta \leftarrow \theta + \alpha \nabla_\theta \rho(\theta)$$

where $Q^{\pi_\theta}(s, a)$ is some estimated value function: we use a discounted Monte Carlo estimate over many trajectories τ. This is popularly known as the REINFORCE algorithm [5].

To parameterize $\pi_\theta : \mathcal{S} \to \mathcal{P}(\mathcal{A})$, two possible actor network π_θ architectures were examined: a dense multilayer perceptron network (MLP) and an adapted attention network. The dense MLP $\mathcal{D}_\theta$ takes a state s_t (current constraints) and action $c_j \in \mathcal{A}^{(t)}$ (candidate cutting plane) pair vector and outputs a scalar score S_{s_t, c_j}; although the state size changes across time, the maximum state size can be determined by the time horizon T. Each state action pair is then padded with 0 to compensate for state sizes less than the maximum state size. This formulation is intuitive: a state action pair represents the new constraints if we were to add the cutting plane.

An alternative approach using attention networks to parameterize the policy is described in Tang et. al. (2020)[6]. This approach learns a parametric function $F_\theta : \mathbb{R}^{n+1} \to \mathbb{R}^k$, where n is the number of decision variables and k is a hyperparameter. The network computes $F_\theta(p_i)$ for all current constraints $q_i \in s_t$ and $F_\theta(c_j)$ for all candidate actions $c_j \in \mathcal{A}^{(t)}$. The score for each action is computed as $S_{s_t, c_j} = \frac{1}{N_t} \sum_{i=1}^{N_t} F_\theta(c_j)^T F_\theta(q_i)$.

Both methods output a score S for each state s_t

action c_j pair, and a softmax is applied on all state action pair scores to obtain a probability distribution over all actions, $\pi_\theta(\cdot|s_t)$.

2.2.2 Actor-Critic Network

Policy gradient methods have zero bias; however, the gradient estimates often have high variance due to the value function $Q^{\pi_\theta}(s,a)$. To reduce variance, a baseline b is often subtracted from Q^{π_θ}; however, determining a baseline is difficult. In our implementation, a critic network V^π is used to estimate the baseline at any state s_t. We define a new function $A^\pi(s_t,a_t) := Q^{\pi_\theta}(s_t,a_t) - V^\pi(s_t)$ and normalize the advantages using the Z-score. The critic network is parameterized using a dense MLP in a similar vein to that of the actor network. Our new gradient ascent method to update our actor network is then given as

$$\nabla_\theta \rho(\theta) = \mathbb{E}_{\pi_\theta}[A^{\pi_\theta}(s_t,a_t)\nabla_\theta \log \pi_\theta(a_t|s_t)]$$

2.2.3 Proximal Policy Optimization

Building off of the actor-critic policy gradient method, Proximal Policy Optimization (PPO)[3] attempts to optimize a policy within a “trust region” [4], taking a conservative approach to parameter search. As PPO is only used to update gradients, the actor and critic definitions are maintained.

Compared to vanilla actor-critic methods, PPO has been shown to be empirically insensitive to step size and has high sample efficiency, achieving state of the art performance on many games. As instance sizes and time limits grow larger, trajectory rollout takes much longer computationally, so sample efficiency is valued highly within this problem.

2.2.4 Random Network Distillation

As rewards in this environment are sparse and many cuts do not achieve good performance, we must ensure that we explore the state space prior to convergence; otherwise, it is likely that local optima is achieved. To encourage exploration, we employ Random Network Distillation (RND) presented by Burda et. al. (2018)[1]. Within our implementation of RND, a randomly initialized dense MLP target network $\mathcal{N}_T$ and predictor network $\mathcal{N}_P$ s.t. $s_t \to \mathbb{R}^k$ are built – the networks are parameterized in a similar light to the padded MLP actor network. The objective of the predictor network is to match the output of the target network, and the predictor network is trained by minimizing the mean squared error.

An intrinsic reward $R^{(I)}$ is defined for action a_t for exploring states unfamiliar to $\mathcal{N}_P$. Our implementation defines the intrinsic reward to be equal to the mean squared error between $\mathcal{N}_P$ and $\mathcal{N}_T$:

$$R^{(I)}(s_t,a_t) = \left\|\mathcal{N}_P(s_{t+1}) - \mathcal{N}_T(s_{t+1})\right\|^2$$

Intuitively, this error will be high for unfamiliar and untrained states, as $\mathcal{N}_P$ has not had the opportunity to learn from it. We incorporate a discounted, normalized intrinsic reward into the advantage function as $A^\pi(s_t,a_t) := Q^{\pi_\theta}(s_t,a_t) - V^\pi(s_t) + R^{(I)}_\gamma(s_{t+1})$.

3 Experiments

Our experiments consist of training on various configurations, each consisting of K instances with m initial constraints, n decision variables and some limited time T to make cuts. Our evaluation criterion is the sum of rewards over trajectories; in other words, we evaluate how effectively the optimality gap is closed.

3.1 Experiment 1: Simple Configuration

Due to time constraints and computational complexity, we first examine various architectures on smaller instances. We create a new environment consisting of $k = 5$ instances, with $m = 15$ constraints and $n = 15$ decision variables over $T = 20$ timesteps. As the easy configuration and hard configuration have more instances with larger decision variables and constraints, even with a parallel rollout, training every architecture on the evaluation instances is infeasible for this project scale.

The reward over time for each architecture is plotted in Figure 1; PPO methods greatly outperform the classical REINFORCE method. Furthermore, PPO with RND has poor performance within the earlier rollouts; however, it eventually exceeds all other architectures with more training. This is most likely due to the explorative nature of RND. Although not a comprehensive examination of all architectures, this experiment serves as a general guide to each model’s performance on larger instances – we proceed our further experiments only examining with an Actor-Critic PPO with RND approach.

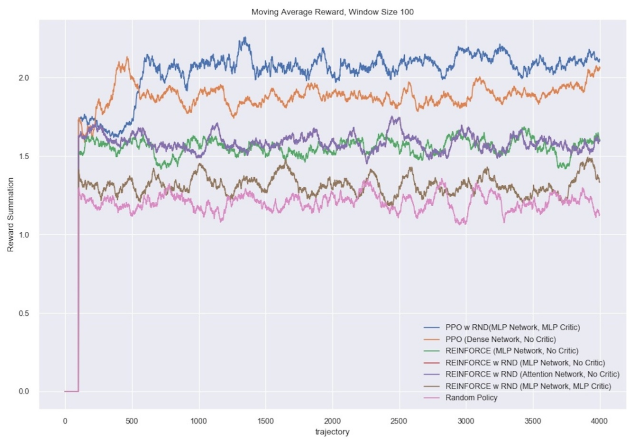

Figure 1: Model Training on a Simple Configuration: $k = 5$, $m = 15$, $n = 15$, $T = 20$.

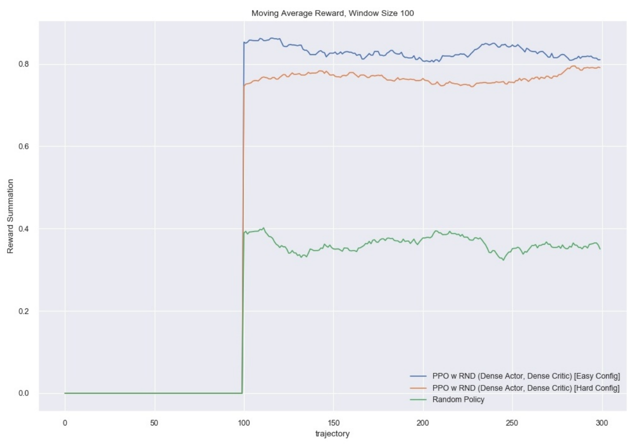

Figure 3: Trained Model Performance on a Test Configuration: $k = 10$, $m = 60$, $n = 60$, $T = 50$.

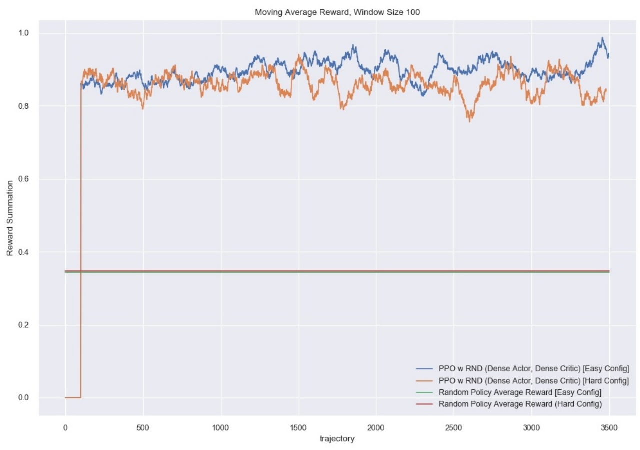

Figure 2: Model Training on a Easy and Hard Configurations: $k = 10$ and 100, $m = 60$, $n = 60$, $T = 50$.

3.2 Experiment 2: Easy and Hard Configuration Training

The easy and hard configurations both consist of $m = 60$ constraints and $n = 60$ variables with $T = 50$ timesteps, but contain $k = 10$ and 100 instances respectively. The reward for these configurations is shown in Figure 3. As a baseline reference, we plot the average reward of 300 trajectories of a uniform policy.

3.3 Experiment 3: Testing on Unseen Instances

Using the models trained from the easy and hard configurations, we examine each model's performance on a test configuration consisting of $k = 10$ unseen instances with the same parameters ($m = 60$, $n = 60$, $T = 60$). A uniform policy is also evaluated on the test instances for reference. Note that these models are not further trained on the test configuration; we fix the parameters to evaluate the generalizability of our models.

Although we expected the model trained on the hard configuration to generalize better than its easy configuration counterpart, the results prove otherwise. We attribute this to the hard configuration having too many instances to learn well in a sufficient amount of time.

4 Discussion

The results achieved on both the training and the test instances exceeded that of the random policy significantly. Although we were able to achieve positive results, there are many extensions of this project that should (*and hopefully will!*) be explored. First, there was unfortunately insufficient time and computational power to perform parameter tuning; most parameters were extracted from each model's respective paper. Second, training the models for higher iterations would have most likely improved the model's performance significantly. Although it appears as if it has locally converged, we hypothesize that with higher weighting on intrinsic reward $R^{(I)}$, we can achieve better convergence. Lastly, normalization methods and reward shaping must definitely be explored; each instance has different coefficients and constraints, and the reward may be small even if we achieve the optimal solution.

Acknowledgements

Thank you to the IEOR 4575 teaching staff for providing the environment, instruction, and guidance!

References

[1] Yuri Burda et al. *Exploration by Random Network Distillation.* 2018. arXiv: `1810.12894 [cs.LG]`.

[2] Ralph Edward Gomory. *An algorithm for the mixed integer problem.* Santa Monica, CA: RAND Corporation, 1960.

[3] John Schulman et al. *Proximal Policy Optimization Algorithms.* 2017. arXiv: `1707.06347 [cs.LG]`.

[4] John Schulman et al. *Trust Region Policy Optimization.* 2017. arXiv: `1502.05477 [cs.LG]`.

[5] Richard S Sutton et al. “Policy Gradient Methods for Reinforcement Learning with Function Approximation”. In: *Advances in Neural Information Processing Systems.* Ed. by S. Solla, T. Leen, and K. Müller. Vol. 12. MIT Press, 2000. URL: `https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf`.

[6] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. “Reinforcement Learning for Integer Programming: Learning to Cut”. In: *CoRR* abs/1906.04859 (2019). arXiv: `1906.04859`. URL: `http://arxiv.org/abs/1906.04859`.