ORCS E4529: Reinforcement Learning
Project Report

Ezequiel Piedras Romero [ep3039]
IEOR Department, School of Engineering, Columbia University

December 2022

1 Introduction

This report summarizes different alternatives that were explored for dealing with the problem presented in [Tang et al., 2020]. As it is mentioned in said paper, there is not an one-size-fits-all efficient algorithm for solving Integer Linear Programming problems (ILP). Hence, it is interesting from an Artificial Intelligence standpoint trying to propose a new approach. Specifically, we are trying to make an marginal improvement to the approach called "Learning to Cut" (LtC), which involves combining Gomory Cuts and Reinforcement Learning. Therefore, the goal of this project is to leverage on the theory we have learned during the semester to propose an improvement on "Learning to Cut".

The files for my project can be viewed at my Github repository (for the link refer to the Appendix 5.3). I made used of TensorFlow [Abadi et al., 2015] and Keras [Chollet et al., 2015] and their online documentations and examples, as well as the Lab4 skeleton provided in the course, for implementing my solution.

A list of people I talked about the project with is also available in the Appendix 5.4.

2 Proposal Description

This section describes the decisions I made during every stage of the project and my reasoning behind them.

2.1 Learning paradigm

This section relies heavily on what I learned from the Reinforcement Learning course and its lecture notes [Agrawal, 2022]. In class we mainly focus on two approaches for making an algorithm learn: Q-learning and policy gradient methods. The former may be seen as indirect approach to learning since what we are actually doing is estimating the reward of a state-action pair for every state and action using dynamic programming. The latter approach is more interesting, though. In policy gradient methods we are trying to learn a mapping from an state to an action (or probability distribution of actions) directly. This is more appealing because said function can be employed in different setting afterwards or may be analyzed more profoundly.

Following this reasoning, I chose to pursue a policy gradient approach for my proposal. Particularly, the actor of my problem is maximizing the expected (infinite) discounted reward of the cutting procedure. Mathematically, we are trying to maximize

$$\rho^{\pi}(s_1) = \lim_{T\to\infty} \mathbb{E}[\sum_{t=1}^{T} \gamma^{t-1} r_t | \pi, s_1]$$

and using its gradient to perform the ascend. From the Policy Gradient Theorem we know its Monte Carlo approximation (for the gradient) is the following if we choose to parameterize the policy in terms of θ:

$$\sum_{t=1}^{T} \gamma^{t-1} \hat{Q}_t \nabla_\theta \log(\pi_\theta(s_t, a_t))$$

Lastly, our goal is to make use of a neural network to model the policy function π. Moreover, θ will be the parameter vector of said neural network.

2.2 Algorithm description

To implement the policy gradient to the "Learning to Cut" problem I started by modifying the code I wrote for Lab 4. However, I made two changes to that algorithm. Firstly, I don't estimate a baseline model for the gradient update. Instead I follow the idea of Evolution Strategies (similar to [Tang et al., 2020]) to simplify the problem. Thus, I simply assume my Monte Carlo trajectories are a (noisy) unbiased estimate of the value of a certain state-action pair. Secondly, I modified the training function of the policy function to account for the varying size of the state vector. Hence, the training function is not longer vectorized and the loss is computed observation by observation.

Consequently, my algorithm is shown in Algorithm 1.

Algorithm 1 Policy gradient method

Require: E: number of iterations, R: number of rollouts, γ: discount factor of rewards, σ: inverse of std. dev. of Monte Carlo trajectories noise, α: learning rate of optimizer.

$\pi(\theta) \leftarrow$ Initialize NeuralNetwork(θ) with random weights
for $e = 1, 2, 3, ..., E$ **do**
  for $r = 1, 2, 3, ..., R$ **do**
    Collect rollouts using the current policy π_θ for $t = 1, ..., T$
    Compute Monte Carlo discounted reward estimates J_i
    Store the state vectors s_i, actions taken a_i, and reward estimates J_i
  end for
  Add Gaussian noise to every J_i such that $E_i = J_i + \frac{\epsilon}{\sigma}$ where $\epsilon \sim \mathcal{N}(0, 1)$
  Compute Function Loss as

$$L = -\frac{1}{R\,T} \sum_i E_i \log \pi(a_i|s_i)$$

  Update NeuralNetwork parameters as $\theta \leftarrow \theta - \alpha \nabla_\theta L$ using TensorFlow
end for

2.3 Policy function parametrization

So far we have established that the policy function is a mapping from the state to an action (or probability distribution of actions). In this part I will deepen into the mathematical modeling of said function.

Let us define the Markovian Decision Process (MDP) we are facing. At every moment t we are facing an ILP that is fully described by its set of constraints $[A_t\ b_t]$ and its set of possible cuts $[D_t\ e_t]$. Note $[\bullet]$ means concatenation. The way in which we solve this instance of the ILP does not depend on the past, and hence it is Markovian. Thus, these two sets define the state s_t of the problem. We define the action a_t as the choice of cut we make. However, in our case, the output of the policy function is the probability distribution over every action $a_{j,t}$ that we can take at every t (j being the index for the available cuts). Note that both the state size and the action size are variable.

To deal with the varying size of the input and output of the function, I adopted the same idea as in [Tang et al., 2020]. That means I perform an Attention operation to keep the output size consistent with the input size.

Let i be the index for the constraints in the ILP and j be the index for the available cuts. Then my policy function computes the $Score_j$ for every available cut and then normalizes them using a Softmax operation. The score is computed such that

$$Score_j = \frac{1}{N_t} \sum_{i=1}^{N_t} g_j^T h_i$$

Name	Input	Description	Num. of Trainable Parameters
Neural Network 1	Constraints	LSTM(8)->Dense(8,tanh)->Dense(8)	464
Neural Network 2	Available Cuts	LSTM(8)->Dense(8,tanh)->Dense(8)	464

Table 1: Details of Neural Network models.

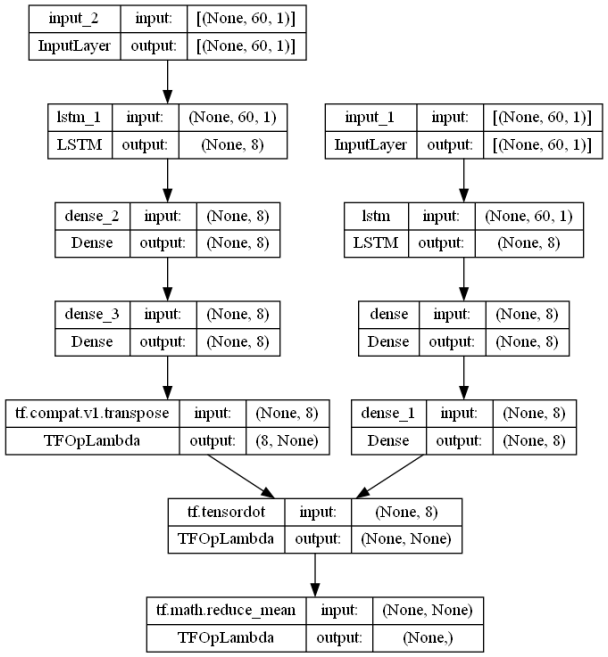

Figure 1: Computational graph representation for Policy Function.

where g_j and h_i are the output of two Neural Networks (one for the constraints and one for the cuts) that take the state vector as input. Details of the Neural Network models are summarized in Table 1. These two models are embedded in my policy function implementation in Python/TensorFlow. A computational graph representation for my policy function is also presented in Figure 1.

Note that using two different models is one of the modifications I propose to the procedure presented on [Tang et al., 2020]. Additionally, it is worth mentioning that my implementation of the policy function in TensorFlow performs a normalization to the input state vector. As mentioned above, the input state vector is made of $[A_t \ b_t]$ and $[D_t \ e_t]$ ($[\bullet]$ means concatenation). However, the vectors b_t and e_t can be discarded if we divide (row-wise) A_t by b_t and D_t by e_t to obtain "standarized" representations of the hyper-planes described by the rows of those matrices. Then, I normalize every column of the transformed A_t and D_t matrices. This way every number that enters my policy function's workflow, as described in Figure 1, is between 0 and 1.

2.4 Training

There are two setups for training: the "easy" setup and the "hard" setup. The former consists of 10 ILPs whereas the latter consists of 100 IPLs. Since I am using the policy gradient method, at every iteration I need to perform enough rollouts to obtain a representative performance of the policy. I chose 10 rollouts (i.e. $R = 10$) for the "easy" setup and 30 rollouts (i.e. $R = 30$) for the "hard" setup since there is trade-off between R and training time. Moreover, I chose a discount factor of 0.10 (i.e. $\gamma = 0.10$) to push the algorithm to maximize the immediate reward of cutting. This choice also reduces the variance of the Monte Carlo discount reward estimate, J_i, since estimated future rewards decay very quickly. Additionally, I chose $\sigma = 10$.

2.5 Exploration of hyper-parameters

Choosing the hyper-parameters for training the policy function model is perhaps the greatest challenge of this project. I chose the optimizer to be Adam because of its good reputation. After a few training attempts I set the learning rate to 0.01. Then I chose the number of iterations to be in range of 10 and 20, thus I could debug and discard bad ideas for the policy function model. The ideas that performed well were retested using more iterations (at least 50-100) to check for policy convergence.

Furthermore, I wrote a Python function that took advantage of the 'multiprocessing' library. Using this library I was able to run my training function simultaneously while varying the hyper-parameters. My function also output a plot with the average training rewards so I could summarize the results of my exploration. An example of said function is presented in Figure 4 in the Appendix. I mainly use this function to test several Neural Network architectures. By Neural Network architecture I mean the type and number of hidden layers, the number of neurons in every layer, and the activation functions.

2.6 Summary of bad ideas

Exploring and choosing the hyper-parameters for the policy function was a long iterative process. In this section I summarize some ideas that I discarded because they were not yielding an improvement relative to a uniform random policy function.

Some bad ideas:

a. Do not perform any normalization in the input state vector: The numbers in the state vectors of the training ILPs are large. If I did not normalize the output of the activation functions, they just converged to their asymptotic values.

b. Use a batch normalization layer just after the input layer: Normalization can be performed before the input layer without using the Normalization layer and hence without increasing the number of parameters.

c. The same model weights for both the constraints matrix and the cuts matrix: The policy did not seem to learn if I did that. Actually, the policy converged to a random uniform policy after a few iterations. I tested this using different setups.

3 Results

According to the (many) runs I made, a uninformative uniform random policy function yields around 0.35 of average training reward. Therefore, a higher average reward means an improvement over a uniform random policy.

My policy gradient method strategy yielded an improvement over the uninformative policy. Average training rewards for my policy function are shown in the following figures. Figure 2 shows the training results for the "Easy" setup, whereas Figure 3 shows the training results for the "Hard" setup. Moreover, the average test rewards on 10 random-generated ILPs suggest the model is able to generalize well on new ILPs. I submitted these same performance metrics to Weights & Biases as well. However, these metrics shown are only a fraction of the many runs I executed. Charts showing tens of runs I tried for the LSTM-Dense-Dense architecture are presented in Appendix 5.2.

4 Conclusion

The policy gradient approach works for the "Learning to Cut" problem. In fact, I was able to obtain similar performance in both setups. Further improvements and fine tuning to my Policy function would require more computational power and time to be achieved.

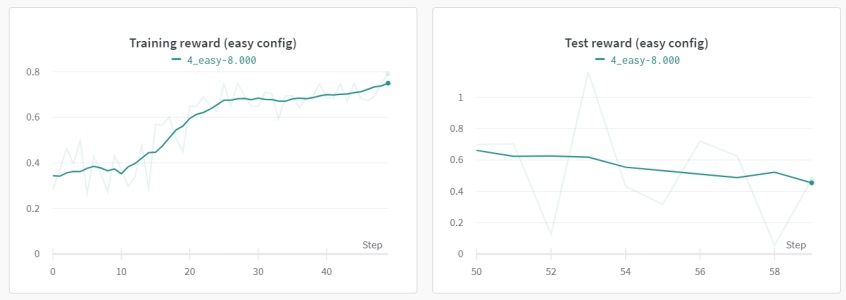

Figure 2: Running-average training rewards and average test rewards in the “Easy” setup.

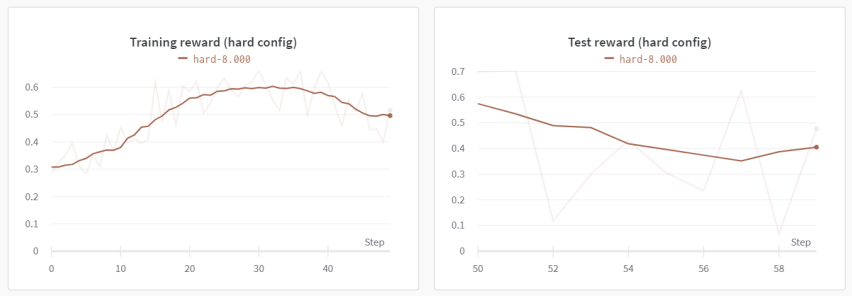

Figure 3: Running-average training rewards and average test rewards in the “Hard” setup.

5 Appendix

5.1 Code for hyper-parameter exploration using multi-processing.

```python
# Only to be executed by master process and not by slave processes
if __name__ == '__main__':

    start_time = time.time()

    pool = multiprocessing.Pool(processes=5)

    # List of hyperparameter values to try
    hyperparameter_list = [4,8,16,32,64]

    training_res = pool.map(run_training_loop, hyperparameter_list)

    training_results = pd.DataFrame(training_res).T
    training_results.columns = map(str,hyperparameter_list)

    print(training_results)

    training_results.plot(figsize=(10,5))
    plt.title("Different hidden sizes")
    plt.show()

    end_time = time.time()
    secs = end_time - start_time
    print("Finished multiprocessing in {secs:.0f} secs.".format(secs=secs))
```

Figure 4: Example code for hyper-parameter exploration using multi-processing.

5.2 Evidence of Policy function training attempts

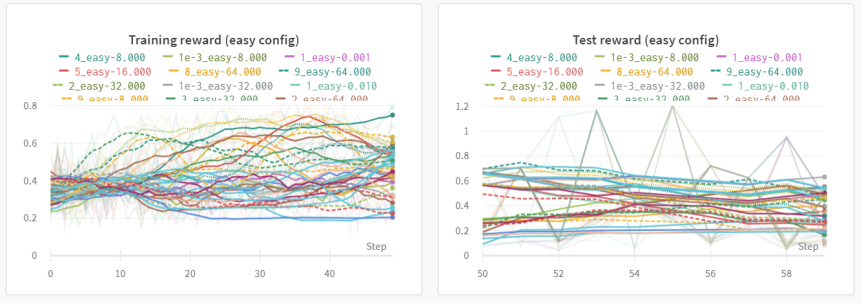

Figure 5: Tens of runs using "Easy" setup.

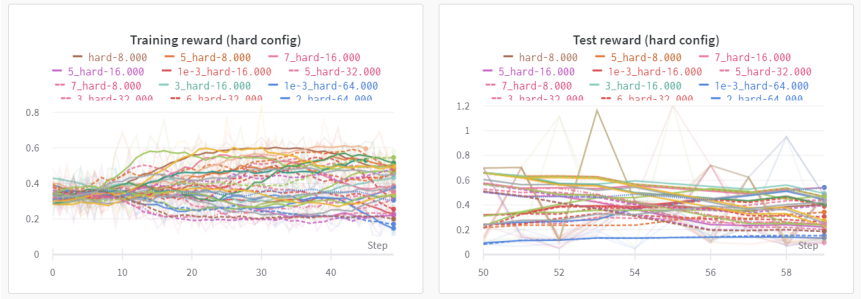

Figure 6: Tens of runs using "Hard" setup.

5.3 Link to Github repository

The files for my project can be viewed at the following Github repository: `https://github.com/chekecocol/LearningToCut`.

5.4 List of people I talked about the project with

a. Eduardo Yanquen

b. Xingyu Lan

c. Alexander Glass-Hardenberg

d. Stefano Benedetto

References

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Agrawal, 2022] Agrawal, S. (2022). Lecture notes in reinforcement learning.

[Chollet et al., 2015] Chollet, F. et al. (2015). Keras. `https://keras.io`.

[Tang et al., 2020] Tang, Y., Agrawal, S., and Faenza, Y. (2020). Reinforcement learning for integer programming: Learning to cut. In III, H. D. and Singh, A., editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 9367–9376. PMLR.