IEOR 4575 Reinforcement Learning Project Report

Han Lin
hl3199@columbia.edu

1 INTRODUCTION

In this project, we aim at solving cutting planes problem in integer programming. Due to space limit, we decide to only present our important effort in the main body, and defer other materials (including pseudo-codes) to the appendix. The structure of our report is as follows:

In section 2, we first stated the attention and LSTM networks architectures we used.

In section 3, we present both **Actor-Critic** and **Evolution Strategy** approaches for training.

In section 4, we mention several useful tricks, including **reward shaping, exploration, etc.**

In section 5, we include our preliminary experimental results.

In section 6, We take a closer look at the data, and try to find what causes the bottle neck.

In section 7, we test the generalization abilities of our best models on random new instances.

2 ARCHITECTURES

In this section, we briefly explain the structure of our LSTM model, and attention networks.

2.1 Attention Network

Because the ordering of constraints does not reflect the geometry of the feasible set, we use attention network [4] for order-agnostic cut selection.

- Firstly, we following the paper to concatenate $[a_i, b_i], i \in [N_t]$ and $[e_j, d_j], i \in [I_t]$.
- Then for a given parametric function $F_\theta : \mathbb{R}^{n+1} \to \mathbb{R}^k$ for some k (we will specify this function later in the LSTM section), we could get embeddings $h_i = F_\theta([a_i, b_i]), i \in [N_t]$, and $g_j = F_\theta([e_j, d_j]), j \in [I_t]$.
- Next, we compute the score $S_j = \frac{1}{N_t} \sum_{i=1}^{N_t} g_j^\top h_i$, which uses dot product to represent similarity.
- Finally, we define probabilities $p_1, ..., p_{I_t}$ by a softmax function softmax($S_1, ..., S_{I_t}$). The resulting I_t-way categorical distribution is the distribution over actions given by policy $\pi_\theta(.|s_t)$ in the current state s_t. (we also tested the alternative way to define $\pi_\theta(.|s_t) = \arg\max_j S_j$ as our deterministic action. See Appendix Sec. 9.2).

Following [3], we define our attention embedding F_θ as a 2-layer NN with 64 units per layer and tanh activation as the default setting. Additional results with deeper and wider NNs is in Sec. 6 Fig. 4.

2.2 LSTM

In addition, we want our RL agent to be able to handle IP instances of different sizes, so we embed each constraint using a LSTM network with hidden state of size $n + 1$ for fixed $n = 10$. In particular, $\tilde{h}_i = LSTM_\theta([\tilde{a}_i, \tilde{b}_i])$ where $\tilde{h}_i \in \mathbb{R}^{n+1}$ is the last hidden state of the LSTM network, and we use it inplace of $[\tilde{a}_i, \tilde{b}_i]$ in the attention network. Same as the settings in paper, the hidden size of our LSTM network is 10. Additional results which double the hidden size is in Sec. 6 Fig. 4.

Code for our project is here:
`https://drive.google.com/drive/folders/1LtfljqGeaVpG_0ty6hXGaZ4gavRGN9Um?usp=sharing`

3 TRAINING

We tried two training methods in our report: Actor-Critic and Evolution Strategy.

3.1 Training Method 1: Actor-Critic

We first use Actor-Critic as our training method. The code is adapted from Lab 4. A key problem here is how to choose the **baseline**. The policy network in Lab 4 contains several linear layers with relu activation, while the baseline network is similar to the policy network, except that the output from the last layer is a scalar value. This is reasonable since we want our baseline to be only dependent on states but agnostic to actions. Similarly, we could define our baseline network with the same architecture as the policy network, except that it uses a scalar value **average score** as output:

$$\frac{1}{N_t}\sum_{j=1}^{N_t} S_j = \frac{1}{N_t^2}\sum_{j=1}^{N_t}\sum_{i=1}^{N_t} g_j^\top h_i \tag{1}$$

Another reasonable approach is to use the **moving average** of returns as baseline. Since we found the choice of baseline is not likely to be a bottleneck of the training performance, we did not test further for other benchmarks as baselines. Pseudo-code for Actor-Critic is shown in Appendix Algo. 2.

3.2 Training Method 2: Evolution Strategy

In addition to Actor-Critic methods, we also tested Evolution strategies (ES) [2] which is used in [3] to train RL agents. ES algorithm is usually used for blackbox optimization, where we used the queried function values to estimate the gradient. We approximate the gradient as $\hat{g}_\theta = \frac{1}{N}\sum_{i=1}^{N} J(\pi_{\theta_i'})\frac{\epsilon_i}{\sigma}$, where $\epsilon_i \sim \mathcal{N}(0, \mathbb{I})$ comes from multivariate gaussian distribution, $\theta_i^{'} = \theta + \sigma\epsilon_i$ is the perturbed θ with σ as some fixed constant, and we estimate $J(\pi_{\theta_i'})$ as $\sum_{t=0}^{T-1} r_t\gamma^t$. To facilitate computation, we use **Ray**[1] to create 8 workers (we trained our model on a 8-core cpu, one worker per core) to process trajectories concurrently. Pseudo-code for Evolution Strategy is shown in Appendix Algo. 3.

3.3 Training Settings

In this part, we briefly explains the settings we use for both training methods. Details about the tricks we used during training are deferred to the next section.

In easy and hard configurations, we train our model for 150 and 250 steps (iterations) respectively, with linear decay for exploration rate. Since there are 10 IP instances in the easy configuration, we call $env.reset()$ 10 times in each training step to ensure we could train each instance without randomness coming from the selection of IP instances. However, for the hard configuration, it takes too long to train all 100 instances in each training step, so we also call $env.reset()$ 10 times in each step (saying this, we still tested the Actor-Critic method that trained on all 100 instances in each step for curiosity of its performance. Details see Sec. 5 Fig. 1).

For Actor-Critic, we set learning rate as 0.001 for both policy and baseline networks. For Evolution Strategy, with benefits from parallelism, we train 8 workers concurrency with each worker call $env.reset()$ 10 times. This is equivalent to train 10 instances with 8 different permutations per step.

4 METHODS

Reward Shaping: This is one of the tricks we found most useful. The intuition for reward shaping is that the rollout reward coming from each IP instance is not directly comparable. As we could see from Table 1, the max gap of the 7^{th} instance is only 1.62, which is much less than the 4^{th} instance. This means that getting a rollout reward of 1.5 is more approaching optimal for the 7^{th} instance than the 4^{th} instance. Therefore, we reshape our reward by the remaining gaps to make the rewards in each period different instances directly comparable:

$$RS = \sum_{t=0}^{T-1} \frac{r_t}{\text{Remaining Gap}_t}\gamma^t \tag{2}$$

[1] `https://github.com/JorgeCeja/evolution-strategies`

where Remaining Gap$_t$ is defined as Max Gap - (rollout reward accumulated until period t).

Table 1: Max Gap for each environment index in the easy config

	idx0	idx1	idx2	idx3	idx4	idx5	idx6	idx7	idx8	idx9
Max Gap	2.89	3.10	2.71	3.61	3.71	2.70	3.12	1.62	3.11	2.92

Normalized Rewards: After collecting 10 rollouts in each training step, we also normalized our rewards. The idea is to encourage the RL agent to update towards the directions which achieve high rewards, and also try to stay away from the directions with low rewards.

$$\text{Normalized RS}_i = \frac{\text{RS}_i - \text{mean(RS)}}{\text{std(RS)}} \tag{3}$$

Standardized Constraints: Multiplying a constant number on all coefficients of a linear constraint does not change the constraint, and we want to be invariant in the scale of coefficients (A, b, E, d), so we standardize each of A, b, E, d across all m constraints given in an epoch. For example, with $A \in \mathbb{R}^{m,n}$, we could denote $\min$ and $\max$ as the minimum/maximum entry in matrix A, and we can standardize it as follows. Same applies to b, E and d.

$$\tilde{A} = \frac{A - \min(A)}{\max(A) - \min(A)} \tag{4}$$

Exploration Strategy: A key problem in this project is how to do exploration. We observe that we could achieve reward between 0.74-0.85 averaged over all 10 instances in the easy configuration. However, closing the gap further is very hard. The reason for this is that our current algorithm is quite "myopic" which chooses the cuts with high rewards in the first several steps. We do experiments with two exploration strategies: the first with **linear decay** in exploration probability, and the second strategy first do random exploration with probability 0.95 in the fist 50 iterations, and then we lower down the exploration probability **step-wise** until 0.1 as we train for more iterations (see Fig. 1). But we found such strategies still insufficient. There exist ES algorithms that encourages exploration [1], and it's a pity that we don't have enough time to try these methods before the deadline.

Discount Factor: We also tuned the discount parameter γ by setting it as either 0.95 or 1(no discount). But our results (see Appendix Sec. 9.2)show that the performance is not very sensitive to γ when it is ≤ 1. We further tried to use $\gamma > 1$, motivations for this idea is in Speculation 3 of Sec. 6.

5 PRELIMINARY RESULTS

In this section, we briefly present the results we trained with the methods mentioned in above sections. Further endeavors are in the next section.

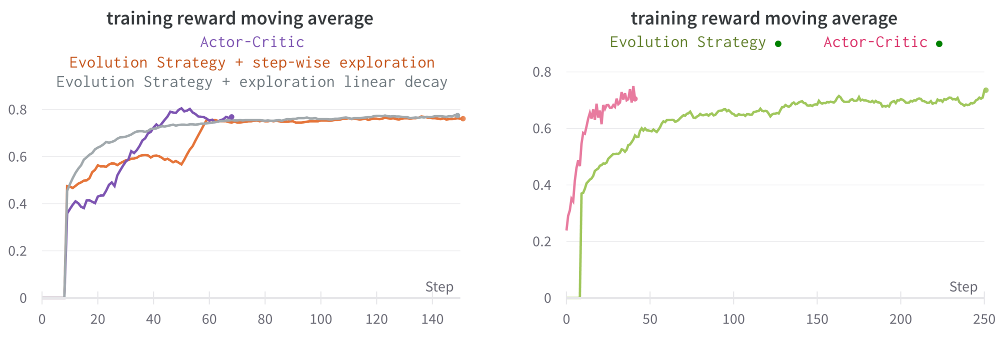


Figure 1: **Left:** Training reward MA for easy config. **Right:** Training reward MA for hard config.

The **Left** subplot in Fig. 1 shows our training results with Actor-Critic algorithm, as well as Evolution Strategy with different exploration methods mentioned above in the easy config. Actor-Critic converges faster than Evolution Strategy since we use lower exploration decay rate in ES algorithms. However, all of them finally converge to rewards around **0.78**, and there is no significant difference between different exploration strategies in the ES algorithm.

The **Right** subplot in Fig. 1 shows our training results with Actor-Critic[2] and Evolution Strategy in the hard config. They could finally converge to reward around **0.74**.

[2]we train Actor-Critic with all 100 instances without parallelism, so we could not train for very long steps.

6 BOTTLENECKS

From Fig. 1, we could observe that the training performance is not sensitive to both the training methods and the exploration methods. Therefore, it is worthwhile to take a closer look at the IP instances (we choose the easy configuration here for simplicity), and try to speculate the reasons that cause our reward not going higher.

Table 2 takes records of the rollout rewards we achieved for each IP instance in the easy configuration trained with ES algorithm. We did not use exploration during training. For all except for the 8^{th} instance, the rollout rewards finally stablize to the values we listed in the table. However, the 8^{th} instance has reward switches between 0.11 and 1.11, which causes the models we trained in Fig. 2 with moving average rewards oscillate between 0.74 and 0.85. In some rare cases, the 3^{rd} instance could achieve reward of 1.61, which causes the peaks with reward of 0.94 in the left subplot of Fig. 2.

Table 2: Rollout Rewards for each environment index when the average reward is 0.74-0.84

	idx0	**idx1**	**idx2**	**idx3**	**idx4**	**idx5**	**idx6**	**idx7**	**idx8**	**idx9**
Rollout Reward	0.88	1.10	0.71	0.61	0.71	0.70	1.12	0.62	$0.11 \sim 1.11$	0.92

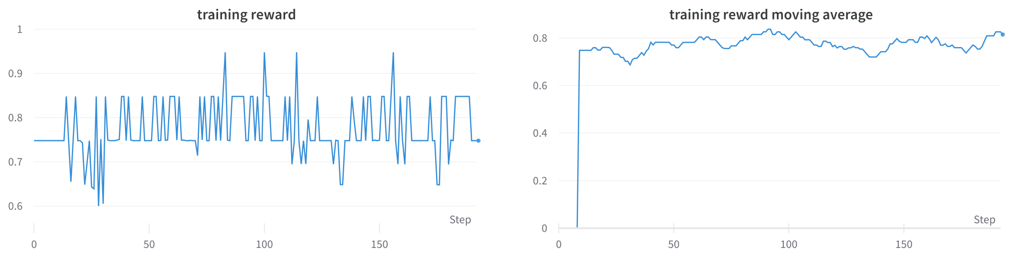


Figure 2: Results for easy configuration. **Left:** Training Reward. **Right:** Training Reward Moving Average.

Therefore, it is natural to ask why our model cannot keep the good directions with higher rewards.

Speculation 1: Bad reward normalization? Even though normalize rewards is common in practice, we guess it might have negative impact on minimizing the gap for some hard IP instances. For example, instance 8 in the easy config usually gives rollout reward of 0.11 (see Table 2), but sometimes a certain permutation in the ES algorithm could discover reward of 1.11. We definitely want to go in this good direction and update our model. However, if we normalized rewards, our model will only be updated towards this good direction if the average reward from all permutations (8 in our setting) of instance 8 is above the mean reward across all instances.

Therefore, after collecting 8 permutation results for an instance in the ES algorithm, we tried not to apply the normalization, instead only use the top 2 rewards to update the model, so that we are more likely to update our model towards these rare good directions. We call this variant as **Elite Rewards**. We add elite rewards filtering criteria on the trained checkpoint models from our last section. As we can see from Fig. 3, it is useful for the hard config, which boost the moving average training reward to **0.7786** at the highest after training for 365 steps, but not significant for the easy config.

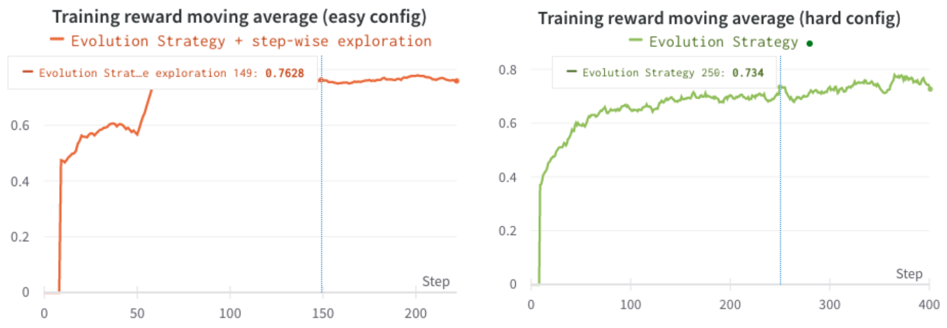


Figure 3: Training reward moving average before & after using Elite Reward. We add elite reward criteria after the models trained for 150 steps and 250 steps respectively for **Left:** easy config, and **Right:** hard config.

Speculation 2: Attention model not complicated enough? Our default model with ES algorithm replicates the architectures in the paper. A reasonable guess is that our model is not complicated enough to learn good reward directions. Therefore, we double the hidden layer size and the output size of the attention network and LSTM layers, and increase the number of linear embedding layers in attention module to three. We found that this stabilizes the training curve a little bit (see Fig. 4).

Speculation 3: Should we really follow the setting with discount factor γ no larger than 1? A discount factor $\gamma < 1$ tends to encourage myopic decisions to select high rewards in the early periods. We guess this might prevent us from closing the integer gap further. Therefore, we also tried a version with $\gamma = 1.1$, which encourages the RL agent to be not that desperate to choose high rewards in the early periods. Even though such strategy is not consistent with the paper, we found we could indeed have some gains by combining it with our Speculation 2 (see Fig. 4).

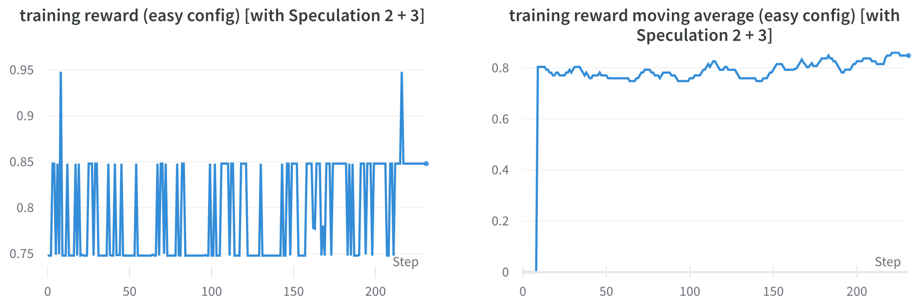


Figure 4: Training rewards and its moving average in the easy config. We use the more complex architectures in speculation 2, as well as setting the discount factor equal to 1.1 as mentioned in speculation 3.

From the left subplot in Fig. 4, we could see that as we train for more steps, we have larger change of achieving rollout reward of 1.11 for the 8^{th} instance. And the training reward moving average curve could finally stablize around **0.8479**! We rephrase here that since we use all 10 instances for training in each step, there's no randomness from the selection of instances (we think this is a more **fair** way to report our result). It is possible for us to achieve higher instant training reward moving average if we randomly select instances in each training step, even though still hard to stabilize at that level.

7 GENERALIZATION

Finally, we tested our models on randomly generated new instances (see Fig. 5). We use linear decay of exploration rate, the more complex architectures as stated in Speculation 2.

Training results on new instances: We trained new models with both $\gamma = 0.95$ and $\gamma = 1.1$. There's not much difference between these two settings, and the highest moving average we could achieve is **0.75** trained on 10 random instances, **0.70** on 50 instances, and **0.68** on 100 instances.

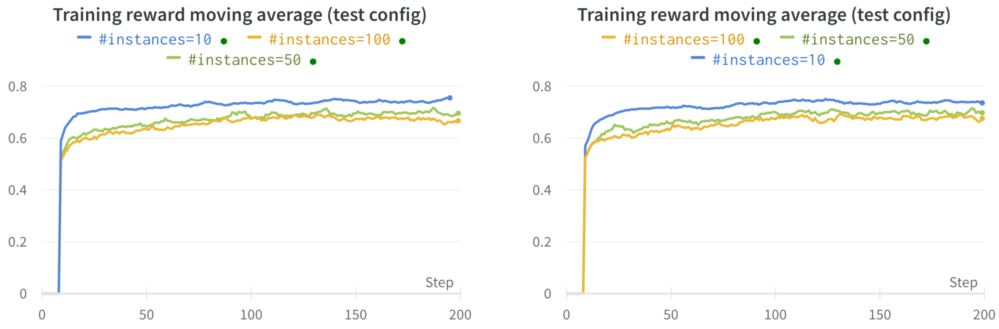


Figure 5: Training rewards moving average for newly generated random instances. We trained with 10, 50 and 100 instances (so that they represent easy, medium, and hard configurations). **Left:** $\gamma = 0.95$, **Right:** $\gamma = 1.1$.

Testing results on new instances: We also reuse the best models we trained on easy and hard configs before, and test their performance directly on new instances without training. Our best model trained on easy and hard configs achieve **0.74** and **0.80** respectively (which means that the randomly generated 100 instances are not that hard to solve compared with the 100 instances in the hard config). It is intuitive that the test score from the model trained on hard config is higher than the one from easy config, since it is trained on more instances and is more likely to generalize better.

Here's the end of the main body of our report. Additional results are in the appendix.

Thanks for your reading!

References

[1] E. Conti, V. Madhavan, F. P. Such, J. Lehman, K. O. Stanley, and J. Clune. Improving exploration in evolution strategies for deep reinforcement learning via a population of novelty-seeking agents. *CoRR*, abs/1712.06560, 2017.

[2] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning, 2017.

[3] Y. Tang, S. Agrawal, and Y. Faenza. Reinforcement learning for integer programming: Learning to cut. *CoRR*, abs/1906.04859, 2019.

[4] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

8 APPENDIX 1: PSEUDO-CODES

We follow the paper [3] and use the following algorithm to rollout our policies. The training algorithms with Actor-Critic and Evolution Strategy are also listed below.

Algorithm 1: Rollout of the Policy

```
Input: policy network parameter $\theta$, IP instance parameterized by $c, A, b$, number of iterations $T$.
Output: state $s_t$, rewards $r_t$, actions $a_t$, with $t = 1...T$
Initialize iteration counter $t = 0$. Initialize minimization LP with constraints $\mathcal{C}^{(0)} = \{Ax \leq b\}$
  and cost vector $c$. Solve to obtain $x^*_{\text{LP}}(0)$. Generate set of candidate cuts $\mathcal{D}^{(0)}$.
while $x^*_{\text{LP}}(t)$ not all integer-valued and $t \leq T$ do
    Constract state $s_t = \{\mathcal{C}^{(t)}, c, x^*_{\text{LP}}(t), \mathcal{D}^{(t)}\}$.
    Standardize matrix $A, E$ and vectors $b, d$ in $\mathcal{C}^{(t)}, \mathcal{D}^{(t)}$ by Eq. (4)
    Sample an action using the distribution over candidate cuts given by policy $\pi_\theta$, as
  $a_t \sim \pi_\theta(.|s_t)$. Here the action $a_t$ corresponds to a cut $\{e^\top x \leq d\} \in \mathcal{D}^{(t)}$.
    Append the cut to the constraint set, $\mathcal{C}^{(t+1)} = \mathcal{C}^{(t)} \cup \{e^\top x \leq d\}$. Solve for $x^*_{\text{LP}}(t+1)$,
  generate $\mathcal{D}^{(t+1)}$.
    Compute reshaped reward $r_t$ by Eq. (2), set $t \leftarrow t + 1$.
return state $s_t$, rewards $r_t$, actions $a_t$, with $t = 1...T$
```

Algorithm 2: Training with Actor-Critic

```
Input: IP instance parameterized by $c, A, b$, number of iterations $T$, learning rate $\alpha, \beta$, number
  of steps $S$, discount factor $\gamma$, number of IP instances $N_{IP}$ (10 for easy config, and 100 for hard)
Output: policy network parameters $\phi$, baseline network parameters $\theta$
for step = 1: $S$:
  for instance = 1: $N_{IP}$:
    Rollout of the Policy: run Algo. 1 above.
    Update value function (baseline network):
    Compute $\hat{V}(s_i) = \sum_{i=t}^{T} r_i \gamma^{i-t}$ as discounted reward over the rest of the path.
    Compute loss: $L = \frac{1}{T+1} \sum_{i=0}^{T} (V_\theta(s_i) - \hat{V}(s_i))^2$.
    Update baseline network by $\theta \leftarrow \theta - \beta \nabla_\theta L$
    Update policy using PG:
    Use monte carlo estimate action-value function $\hat{Q}(s_i, a_i)$ using discounted rewards similar
  as above.
    Use value function as a baseline to compute advantage $\hat{A}(s_i, a_i) = \hat{Q}(s_i, a_i) - V_\theta(s_i)$
    Compute surrogate loss $L = -\frac{1}{T+1} \sum_i \hat{A}(s_i, a_i) \log \pi(a_i|s_i)$
    Update policy network by $\phi \leftarrow \phi - \alpha \nabla_\phi L$
return policy network parameters $\phi$, baseline network parameters $\theta$
```

Algorithm 3: Training with Evolution Strategy

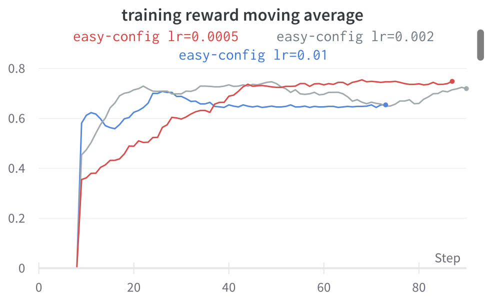
```
Input: IP instance parameterized by $c, A, b$, number of iterations $T$, learning rate $\alpha$, noise
  standard deviation $\sigma$, number of steps $S$, discount factor $\gamma$, number of IP instances $N_{IP}$ (10 for
  both easy and hard config)
Output: policy network parameters $\theta$
for s = 1: $S$:
  Sample permutations $\epsilon_1, \epsilon_2, ...\epsilon_k \sim \mathcal{N}(0, 1)$
  Set permuted policy network parameters $\theta'_1 = \theta + \sigma\epsilon_1, \theta'_2 = \theta + \sigma\epsilon_2, ..., \theta'_k = \theta + \sigma\epsilon_k$.
  for permutation $i = 1 : k$
    for instance j = 1: $N_{IP}$:
      Rollout of the Policy: run Algo. 1 with $\theta'_i$, and collect returns $r_{i,j}$.
    Run Eq. 3 if choose to normalize reward,
    Calculate $J(\pi_{\theta'_i}) = \frac{1}{N_{IP}} \sum_{j=0}^{N_{IP}} \sum_{t=0}^{T-1} r_{i,j,t} \gamma^t$
  Run Eq. () if choose elite rewards.
  Set $\theta_{s+1} \leftarrow \theta_s + \alpha \frac{1}{k\sigma} \sum_{i=1}^{k} J(\pi_{\theta'_i}) \epsilon_i$.
return policy network parameters $\theta$
```

9 APPENDIX 2: EMPIRICAL TESTS

In this section, we report several additional trails including changing the learning rate (see Appendix Sec. 9.1), using deterministic and probabilistic actions (see Appendix Sec. 9.2), etc. Most of them does not work well[3], so we do not include them in our main report due to space limit.

9.1 Learning Rate

In Fig. 6, we plotted the moving average training reward from Actor-Critic method with different learning rates. The left subplot is for easy config, and the right for hard. As we can see, a small learning rate will be more stable, despite longer training time to converge.



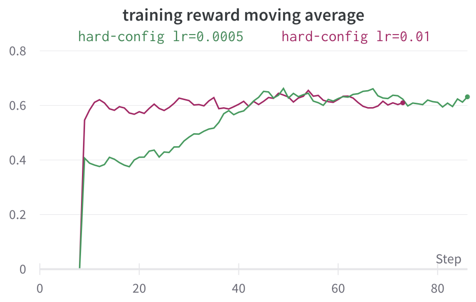


Figure 6: Results with different learning rates. **Left:** easy config. **Right:** hard config.

9.2 Discount factor γ and action selection strategies in easy config

In this section, we first test our Actor-Critic method with different discount factors with $\gamma = 1$ (no discount) and $\gamma = 0.95$ (see Fig. 7). We choose $\gamma = 0.95$ rather than some other values less than 1 because this scale could make the reward at period 50 with weight less than 10%, which we believe is small enough to encourage the RL agent to select cuts with high rewards in the early periods. As we could see from Fig. 7, we found that the choice of γ is not that crucial to our results.

Besides, we also shows the deterministic and probabilistic action selection methods in Fig. 7. Usually we do not want to use deterministic actions (argmax in the Fig.) since it might overfit and not generalize well. Luckily, we find that the probabilistic approach (select actions with probabilities as their softmax values) could converge to the same performance as the deterministic approach. Therefore, it is fine for us to proceed with probabilistic actions.

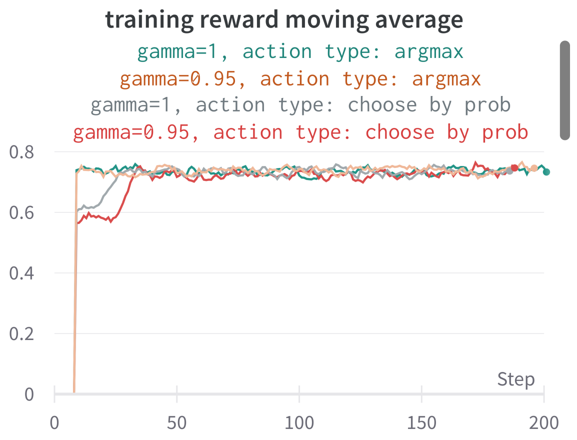


Figure 7: Results with different gamma values ($\gamma = 0.95$ and $\gamma = 1$) and deterministic & probabilistic action selection criteria.

[3]Didn't realize I made so many trails until seeing Fig. 10

We also tested different γ with Evolution Strategy training algorithm. The result is shown in Fig. 8. The conclusion is the same, training performance is not sensitive to γ when we set it not larger than 1.

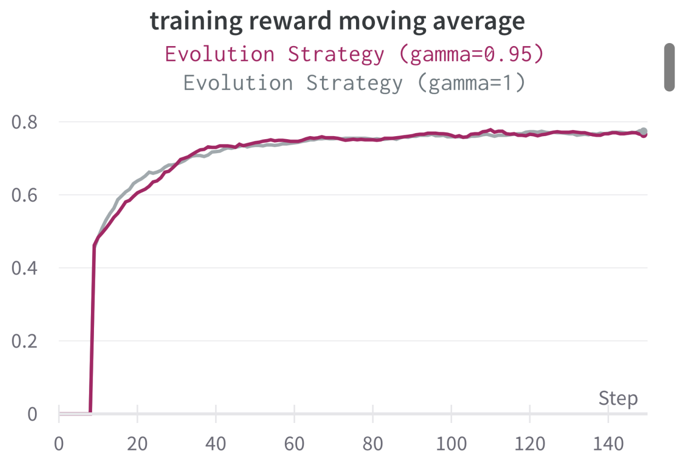


Figure 8: Results with different discount factor γ and deterministic/probabilistic actions.

9.3 Discount factor γ in hard config

In this section, we reported the effect of $\gamma > 1$ with ES algorithm in the hard configuration. As we could see from Fig. 9, the green curve with $\gamma = 1.1$ performs slightly better than the purple curve with $\gamma = 0.95$. After training for 250 steps, we saved the checkpoint of the model in the green curve, and applied elite reward selection, which boost the performance a little bit further.

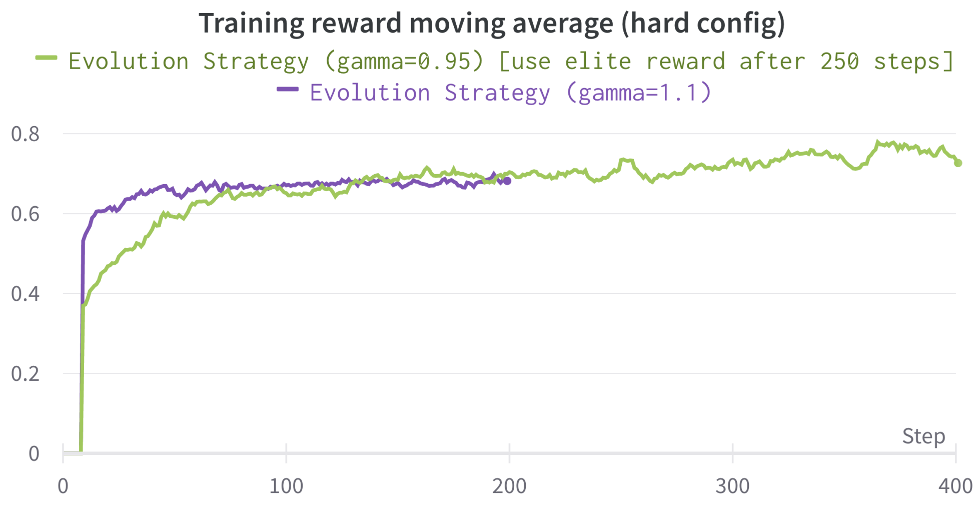


Figure 9: Evolution Strategy with $\gamma = 1.1$ and $\gamma = 0.95$ in the hard config.