# Learning to Cut with Reinforcement Learning

Ishaan Preetam Chandratreya *Department of Computer Science*
*Columbia University*
New York, USA
ipc2107@columbia.edu

***Abstract*—Integer programming is a computationally difficult problem in which a objective function must be minimized inside a feasible region with the optimization variables required to take on integral values. Amidst various heuristics available for solving this problem, a popular class of algorithms are 'cutting-plane' methods, which iteratively restrict the feasible region using information from solving a sequence of linear programs instead. At each iteration, these algorithm pick an additional constraint which further restricts the feasible region. Such choices are Markovian given the current set of constraint and available options to restrict or "cut" the region, and hence lend themselves to formulation as a Markov Decision Process. In this project, I replicate results that build on this MDP assumption and formulate the choice of which region to cut as a reinforcement learning problem, combining the strength of SOTA linear program solvers and the approximation power of supervised learning. I explore some powerful RL techniques we have studied in class as an alternative to evolutionary strategy training, and report results on two policy gradient techniques. Noticing that calls to simplex solvers are the main time-sink in training, I further reflect on how we can use pre-training or alternate formulations of the problem to improve the sample efficiency of the learning task.**

***Index Terms*—Reinforcement learning, Integer Programming, Policy Gradient, Sequential Models, NP Completeness**

## I. Introduction and Background

A wide variety of optimization problems that are useful to humans can be formulated as linear programs, an optimization problem with linear objective functions and affine constraints. Indeed, such problems are relevant in the domains of networks and flows, game theory, robotics and so on. The standard form of the linear program can be expressed as follows, and any linear program can be modified so that it is represented in such form (with the variable in bold the vector of optimization parameters):

$$\begin{aligned} \textbf{max} \quad & c \cdot \mathbf{x} \\ \textbf{s.t.} \quad & A\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq 0 \end{aligned} \tag{1}$$

This problem can be solved with a worst case exponential-time algorithm, Simplex, of which highly emperically efficient implementations are available, as well as less used linear time algorithms, such as Interior Point Methods. Integer programming is the hard "cousin" of linear programming, where the optimization variable **x** must further be consisting only of integer entries. This makes the problem far harder, as the search space is highly narrowed (the integers form 0 percent the reals) and has been shown to be NP-complete by reduction from generalization of zero-one equations (ZOE). A trivial way to form an approximation of the solution is by removing the integral constraint, quickly solving the linear program in polynomial time and rounding the required output: however emperically this can lead to catastrophic results in higher dimensions and it is very easy to break any algorithm that relies solely on a particular schema of rounding.

To this end, *cutting plane methods* come up with a better strategy to use the simplicity of solving linear programs. Given an integer program, they obtain the corresponding linear program without the integral constraints and solve for it. However, instead of rounding the solution, they use it to find the optimization variables on which the solution to the linear program violates the integral constraint and crafts a series of candidate constraints that would prevent such a case from occuring, esentially candidate 'chops' that bring one closer to the solution. A cut is chosen from the set of candidate cuts and then combined with the former constraints to make a new, refined linear program which can be solved again and the process repeated until all variables are integral. Given that multiple optimization variables can violate the integral rule at any stage, the set of candidate cuts is usually large. The choice of cuts taken at any stage can be critical in how quickly the algorithm converges and hence choosing such cuts is an important problem.

Classic NP complete approximation approaches to improve on simple random choices could include *branch and bound* methods that backtrack upon making an arbitrary poor choice, or use a *heuristic* to make a locally optimal decision. In this project, we will instead

make use of the context available through the chosen constraints and the optimization problem itself in making this choice at every stage, formulating the iterative classification as a Markov Decision Process and presenting a reinforcement learning solution.

## II. Method Description

[1] sets up the crucial RL formulation for this task, with a state space $\mathcal{S}$, an action space $\mathcal{A}$, a reward function $\mathbb{R}$ and a transition model to move between states at different timeteps $s_t, s_{t+1} \in \mathcal{S}$ conditioned on a particular action $a_t \in \mathcal{A}$. In short, $\mathcal{S}$ is formed by the set of available constraints at any given stage for an LP in standard form, the set of possible candidate cuts, the optimal solution to the linear program corresponding to the integer program (current linear relaxation) and the object function coefficients which stay constant across timesteps. $\mathcal{A}$ is composed of the set of indexes for the available candidate cuts at each timestep, with an action choosing a particular index. The transition is using $a_t$, forming a new integer program with all constraints from $s_t$ and the additional chosen cut moved to the constraints set. This is then solved to create the next state $s_{t+1}$. Throughout my experiments, I have used the reward $c^T x_{LP} * (t+1) - c^T x_{LP} * (t)$ from [1] aka the instant decrease in solution from the cut: the larger this is the more constraining the new cut and hence more useful.This reward is always non negative as a new cut lessens the feasible region and allows access to fewer regions on the objective function. The goal is to learn a probabilistic policy $\pi : \mathcal{S} \mapsto \mathcal{P}(\mathcal{A})$. This description translates to the following general pseudocode of my algorithm:

- Initialize a **trainer** that handles all neural networks and deploys them for training or evaluation.
- For **num _iter** iterations:
  - For **num_episode** trajectories:
    - **collect _episode**
  - **combine _episodes**
  - **train_models**

In the following section I will detail the **collect _episode**, **combine _episodes** and **train_models** procedures for simplicity but the overall algorithm will follow the structure shown above. The same structure can be found in my **example.py** code submitted to courseworks, with models and further details found in **policy.py**.

## III. Experiments

All results reported share the following common tropes. The moving average is calculated over 100 episodes and not calculated for the first 20 iterations until atleast 100 episodes have been amassed. Unless otherwise noted, experiments are run with 5 trajectories unrolled using the current policy at each iteration: hence 20 iterations are required for the first moving average report and then a report is made over each of the next iterations. All experiments use a learning rate of $0.01$ for both the policy and where applicable for the value baseline. Unless otherwise noted, a discount factor $\gamma = 0.99$ is used in all settings. All experiments are run for a variable number of total iterations as indicated by the varying number of steps on the x axis of the graphs. Experiments are run on a single Tesla T4 GPU with CUDA enabled and trajectories are rolled out on policy using Python's concurrent futures library.

### A. Random runs

In each of the graph with more than two plots, **drawn-pond** run shows the random policy moving average result on hard config and **graceful-music** run shows the random policy moving average result on easy config. Random policy is simulated by drawing the action chosen from a uniform distribution (no prior) at each stage.

### B. Vanilla Policy Gradient

We know that we seek an optimal strategy for finding the best cuts. In class we studied the two basic class of algorithms that help us achieve this: q-learning based methods and policy gradient based methods. While both seek to estimate some kind of model for the environment, the focus on environment modelling is more on the Q/TD learning side of methods, where we must estimate a value for every state, action pair (using dynamic programming or function approximation) and then greedily choose a policy according to that. In policy gradient, we instead parametrize the policy and directly optimize it, making these methods useful in the continuous space. In the problem at hand, the space is indeed continuous: it is additionally not a constant dimension state/action space, and hence Q-learning based methods are likely a bad choice for this. While it is complicated to estimate the policy gradient because of joint modelling need for the stationary distribution of states and because of the need to select actions, we know from class that we can use the policy gradient theorem [2] to use this, and this is what i leverage here.

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi[Q^\pi(s,a)\nabla_\theta \log \pi_\theta(a|s)]$$

Consider the relevant procedure for this algorithm:
**collect _episode**

- Sample initial **A**, **b**, **cuts a**, **cuts b**, **c0** from allowed variables in given config.
- Pad **A**, **b**, **cuts a**, **cuts b** to the required max length (here set to 111) with $0x \leq 0$ trivial equations so that these can be later concatenated
- while not done or while max $<$ max length of episode:
  - Use current policy $\pi$ to get probability distribution over actions given state
  - Zero out all the $0x \leq 0$ probabilities and renormalize so that they are never chosen and used as false learning symbol
  - Sample an action from the resultant distribution. Step on the provided Gurobi environment, and observe the reward and the next state as a result

This process closely follows algorithm 1 in [1] with additional padding. Please refer to function **unroll episodes** in **policy.py** for details. Then we encounter the following **combine _episodes** strategy.

**combine _episodes**

- Stack all **A**, **b**, **cuts a**, **cuts b**, **c0** from an episode along one dimension
- Stack all **A**, **b**, **cuts a**, **cuts b**, **c0** from all episodes along another dimension

Note that it is our padding scheme that allows this to work. Finally, the train strategy is as follows:

**train**

- For given batch of **A**, **b**, **cuts a**, **cuts b**, concatenate **A**, **b** and **cuts a**, **cuts b**
- Run LSTM + dense policy on the concatenated values. Note that the LSTM aggregates over the dimension of number of variables (60 in our case). The dense model and LSTM both use num_episodes * max_episode _length as batch size.
- Perform the attention mechanism from [1]
- Using Monte Carlo estimates of rewards as $Qs$, and probabilities from the attention operation, evaluate and aggregate the policy gradient. Directly optimize this using backpropagation.

Please note the following hyperparams for the policy: the LSTM is a uni-directional LSTM with unit input size and a single layer with hidden size 61 (as per size of given problem A+b). We unroll the LSTM for sequeence length 60, as much as the number of variables in the optimization problem. The dense network is a 2 layer network that maps from the last hidden state of the LSTM with 2 layers of 64 dimensions each and tanh non linearities. Please see the **Policy** class in **policy.py** for further details.


Fig. 1. Vanilla Policy Gradient: Easy and Hard Moving Average

Please consider 1 for the result on this experiment. **astral-puddle** is on hard config, while **swift-sea** is on easy config. We observe that hard config indeed yields less reward than easy config, and that while hard config achieves a maximum of about 0.7 in moving average over 100 episodes, easy config achieves about 0.8.

### C. Using context embeddings

In this process, we retain all aspects from (A) except that we also include the optimization multipliers in the policy for context. I do this by using a separate single layer Dense layer from the number of variables (60) to the output size of the dense network from above (64). I then add the context from c0 to the encoded [A, b] and [cuts a, cuts b] before the attention pass.

Please look at 2, 3, 4, 5 for experimnts. The first plot shows a basic run with this method: **different-dawn** is hard config, **bright-microwave** is easy config. As you can see the performance is increasing even at the end of the run and the performance for hard and easy configs is closer on this one and both significantly outperform the random baseline. The second plot is easy config only, and does extensive hyperparam tuning for the results- include learning rates and gamma: this achieves a max moving average of about 0.9, which is the highest expected given our constraints. The third of these plots repeats the experiment on longer runs to ensure a plateau is found with **cerulean-eon** the hard config and **solar-forest** the easy config. The final plot is an evaluate only plot, with the moving average window lessened to increase noise of the plot on the fully trained best model. As you can see, this still outperforms the random config even with the additional variance and achieves above 1.2 reward performance on both easy (**sunny-violet**) and hard config (**scarlet-jazz**) in some cases.

### D. Actor Critic

This replaces the given **train** procedure with an actor critic method. In class, we learned that additionally

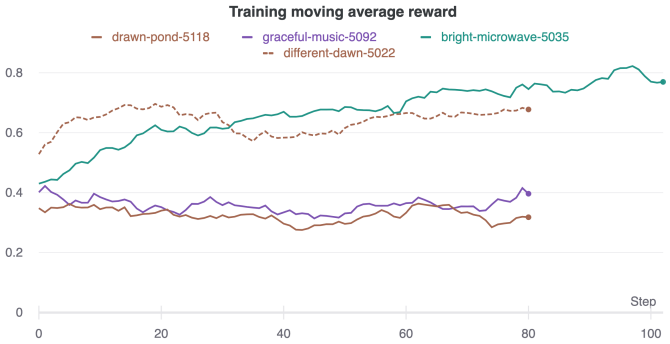


Fig. 2. Context embedding basic results: Easy and Hard Config Moving Average

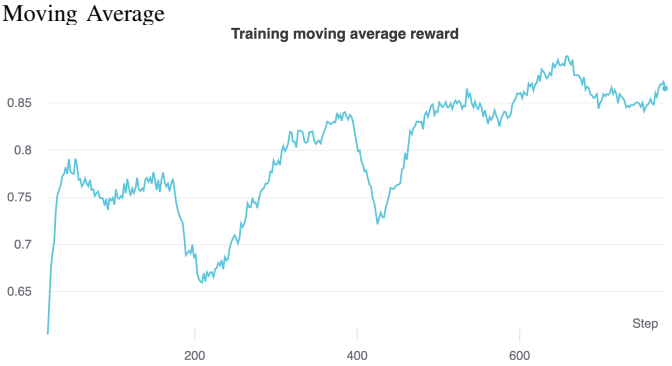


Fig. 3. Context embedding Tuned Hyperparams: Easy Config Moving Average

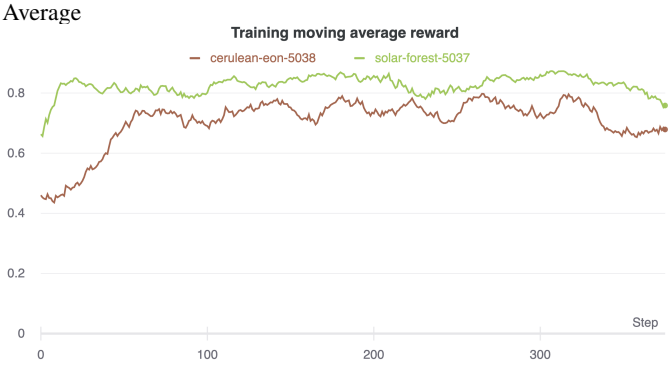


Fig. 4. Context embedding Long Run results: Easy and Hard Moving Average

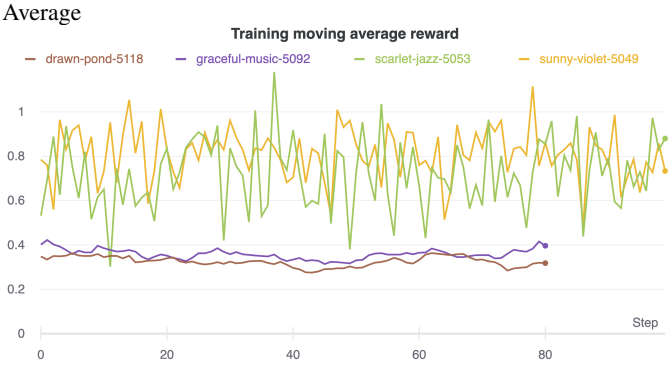


Fig. 5. Context embedding + vanilla policy gradient moving average reported with smaller window to compare noise against random baseline performance

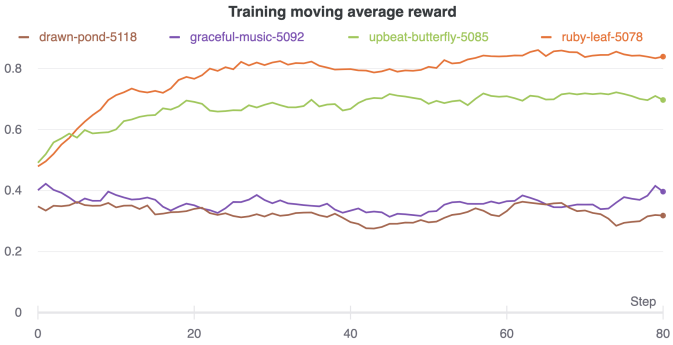


Fig. 6. Actor Critic: Easy and Hard Moving Average

learning the value function can help reduce the variance in gradients achieved during training via baselining. I use a value function which is very similar to the policy encoders (LSTM + dense) as we want a function which is still invariant to optimization variable order and builds context. Additionally, I also used a context embedding for the value function as in estimating the value of the state it will be important to consider relation of feasible region to the optimization function. The value function is trained using MSE loss, and is used to replace the monte carlo estimates with advantage estimates from the value function, closely following the pseudocode in lab4. Please see the class **Value** in **policy.py** for details.

6 shows the results on actor critic. While we still outperform baselines, I did not see an emperical advantage from using this method except that the reward curves seem a bit more smoothened (perhaps as a result of lower gradient variance). Here, easy config is **ruby-leaf** and hard config is **upbeat-butterfly**.

### *E. Smaller batching*

One of the redundancies in my code is my use of padding while collecting the trajectories. This complicates the min/max normalization on piazza (as 0 is found for minimum always-hence my initial implementation did not have scale invariance) and adds unnecessary latency to the code. Additionally, my code as it stands does not scale well to GPU training because I use a very large batch size in passing all trajectories from a single iteration (and all their timesteps) through the model at once- which is enabled through the padding. This can lead to fluctuating gradients in Adam optimizer and also CUDA OOM errors unless a very large GPU such as the T4 is used. To scale my results better and allow for the use of (min-max) normalization [This scales every **A**, **b**, **cuts a**, **cuts b**] using the min and max digit found in each], I implement a smaller batch based actor critic, which encodes every single episode independently. For

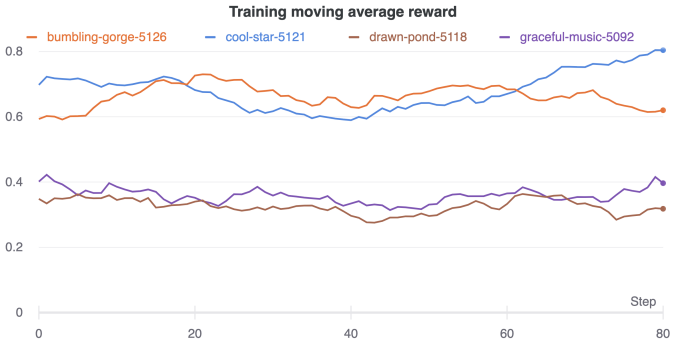


Fig. 7. Smaller batching with actor critic

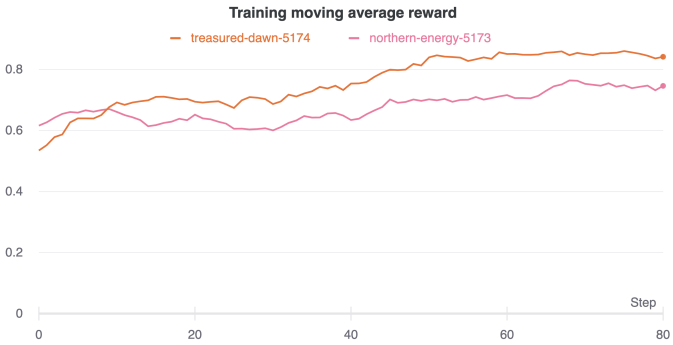


Fig. 8. Smaller batching with actor critic + normalize

this option, no padding is used and the variable size **A**, **b**, **cuts a**, **cuts b** are instead all added to a list and passed to the training procedure, where a for loop is used and the variable sized constraints are abstracted into the batch size. The code for this is tenuous but comparably fast, and can be found by following the **if smaller _batch** and **if not smaller _statements** in **policy.py** and **example.py**. Most parametrizations of policies stay the same with the exception being the forward pass in the value function accounting for the variable number of constraints.

7 shows the moving average for this experiment without normalization. The only noticeable difference is that the highest value is reached earlier (Recall that the first result shown is the moving average for the first 20 episodes and this has already reached a high number compared to other plots). **cool-star** is hard config and **bumbling-gorge** is easy config.

8 shows the moving average for the experiment with normalization. The results are similar, except that the moving averagee is still climbing at the time of termination quickly, and can be expected to grow. **treasured-dawn** is the easy config and **northern-energy** is the hard config. A high of 0.85 and 0.75 is achieved respectively over about 100 episodes.

**After this point, this is all additional information and further work/suggestions so the 4 page limit is hopefully met**

### *F. Exploring alternate sequential models*

In this approach, we have used LSTM [3] to encourage sequential coverage over the 60 different variables and the aggregation of information across them. In [1], the LSTM was more a means to aggregate information from and jointly learn over problems of different sizes but the task presented to us was more centered on similar problem sizes and so I still deployed LSTM with the hope that empirical risk minimization method would generalize to larger or smaller problems without access to the same in the training distribution (the model architecture still allows for a forward pass on larger instances). However, it would be better if such a sequential model could be used that allows for a more focused learning of instances of the same size. Another thing that was crucial in [1] was the focus on attention [4] to build certain invariance into the model. Attention means that the sequence in which the embeddings are accessed does not affect the choice of the action at that stage. But it is reasonable to hope that such a attention-scheme and associated invariance is built earlier on in the model. The need for such invariance is well explored in the natural language processing community, where it might be important that certain aspects of the sentence are attended to more, and context must be bidirectional. To this end, we can tap into the literature on the hugely successful transformers [5] models that have been used for various multimodal tasks from video classification to language entailment and have shown potential in working with raw data with spatial organization in addition to perceptual data, such as physical trajectories. Essentially, transformers can be interpreted as graph neural networks [6] and this interpretation allows for a pass on any kind of data, as long as the corresponding tokenization and attention mechnanism is modified akin to required ”message passing”. These would have the following utility:

1) Bidirectional context and multi-head attention: Using transformers would move the attention mechanism far ahead and allow different variables to attend over each other in different subspaces, perhaps leading to stronger geometric acquisition
2) More amenable to pretraining tasks (see G.): transformers are typically pretrained and then generalize well to downstream tasks: while we can do the same with LSTMs transformers are SOTA and proven in this space.
3) Explicitly train to build certain invariance: We would like our equations to be invariant to both

the order of the constraints as well as the variables inside of it. Certain variants of transformers (such as Set Transformers [7] are useful in such regard).

While I set up the configurations and code required to build a modified BERT variant [8] using the HuggingFace library [9] I had trouble running this on a single GPU even for a shallow config and my request for quota increase of Google Cloud was denied so I was not able to verify this promising direction. The configuration I used was a 2-layer, 2 -head BERT model which upon receiving a set of constraints padded them with [PAD] tokenized embeddings on one side and attached a [CLS] token on the other side. I was planning to train the model jointly with the rest of the RL scheme and connecting the output of the [CLS] token with the fully connected layers.

### G. Further ideas

An extension of this work, which is heavily based on policy gradient approaches, could be improved upon by drawing on literature from integer programming and NP completeness, as well as reinforcement learning. Namely, I was considering the following goals as the most pertinent next steps given more time.

1) Opportunities to build stronger invariances during learning to similar optimization problems using **contrastive learning** [10]
2) Opportunities to re-purpose failed states with a different role to improve the experience replay using **Hindsight Experience Replay** [11].

For 1) we notice that we have desired the likes of scale and permutation invariance throughout our training task. However, we would also like our method to be invariant to different formulations of the same integer program. For example, different formulations could include non standard forms, forms with slack variables, eliminating equalities and inequalities, strong surrogate relaxations etc, in addition to scale and permutation. One way of formulating this could be to add a reward penalty at each time step for violating such invariance. Aka, we could create "equivalent" integer programs at each stage of the markov decision process and add a penalty associated with the difference in the actions that they predict, forcing representations to align. Contrastive learning with positives and negatives is amenable to this kind of approach and has been used for RL tasks before [12].

For 2) we hope to use some sort of goal metric to add supervision and additional conditioning while learning the policy. At each stage of the current formulation we are solving some approximation of the integer program, so one idea is to use the scale of the approximation as the goal. Emperically, most of the trajectories found during training do not reach "done" stage and this method allows us to repurpose existing states by conditioning them as successful "done" stage for the approximation achieved at that stage. This is likely useful for other NP complete formulations that are written as IP as well.

## IV. Discussion and Conclusion

We can see that using policy gradient techniques, we can achieve a reasonable reward for the integer programming problem, and hence this is a promising direction to take and further improve on. From the graphs, we achieve roughly 30 percent of the best possible groundtruth results on the best runs, which is what is expected in this case. We notice that actor critic improves policy gradient by reducing variance, and that there are some benefits to using the optimization problem as a context embedding for the choice of cuts. We also notice that an alternate formulation of the problem without 0 padding allows us to use normalization and build scale invariance into the model as well as make the problem feasible on GPUs of lower memory, in addition to improving results marginally. We notice that given more time and resource, extensive hyperparameter tuning can significantly improve results (as with the context embedding results). Given more time and compute resource, also, I would love to explore the transformer and self-supervised directions outlined above.

## V. Acknowledgement

I thank Prof. Shipra Agrawal and TAs Yunhao and Abhi for the base code for this problem, the former TA's paper [1] and Prof's lectures for forming the basics of all algorithms. Thanks for a great semester!

## References

[1] Y. Tang, S. Agrawal, and Y. Faenza, "Reinforcement learning for integer programming: Learning to cut," *CoRR*, vol. abs/1906.04859, 2019. [Online]. Available: http://arxiv.org/abs/1906.04859

[2] L. Weng, "Policy gradient algorithms," *lilianweng.github.io/lil-log*, 2018. [Online]. Available: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

[3] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, p. 1735–1780, Nov. 1997. [Online]. Available: https://doi.org/10.1162/neco.1997.9.8.1735

[4] L. Weng, "Attention? attention!" *lilianweng.github.io/lil-log*, 2018. [Online]. Available: http://lilianweng.github.io/lil-log/2018/06/24/attention-attention.html

[5] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," 2017.

[6] C. Joshi, "Transformers are graph neural networks," *The Gradient*, 2020.


Fig. 9. Vanilla Policy Gradient: Easy and Hard Reward


Fig. 10. Basic Context Embedding: Reward


Fig. 11. Tuned Context Embedding: Reward

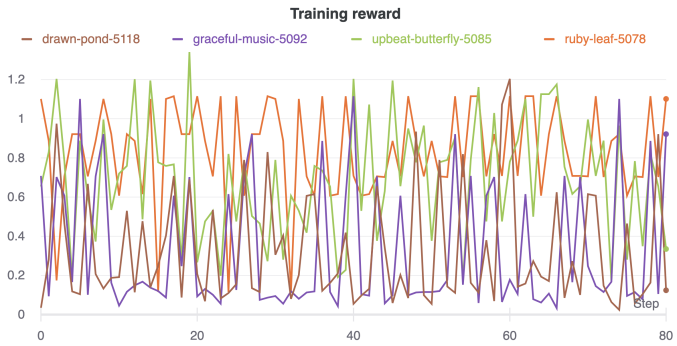


Fig. 12. Actor Critic: Reward

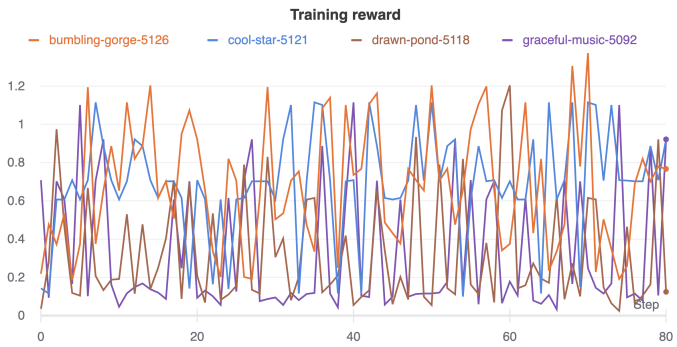


Fig. 13. Smaller batch method: Reward

[7] J. Lee, Y. Lee, J. Kim, A. R. Kosiorek, S. Choi, and Y. W. Teh, "Set transformer," *CoRR*, vol. abs/1810.00825, 2018. [Online]. Available: http://arxiv.org/abs/1810.00825

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019.

[9] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T. L. Scao, S. Gugger, M. Drame, Q. Lhoest, and A. M. Rush, "Huggingface's transformers: State-of-the-art natural language processing," 2020.

[10] A. van den Oord, Y. Li, and O. Vinyals, "Representation learning with contrastive predictive coding," *CoRR*, vol. abs/1807.03748, 2018. [Online]. Available: http://arxiv.org/abs/1807.03748

[11] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel, and W. Zaremba, "Hindsight experience replay," *CoRR*, vol. abs/1707.01495, 2017. [Online]. Available: http://arxiv.org/abs/1707.01495

[12] L. Weng, "Self-supervised representation learning," *lilianweng.github.io/lil-log*, 2019. [Online]. Available: https://lilianweng.github.io/lil-log/2019/11/10/self-supervised-learning.html

## VI. Appendix

Please find the corresponding absolute rewards achieved for the last episode in each iteration corresponding to the moving average results presented in the main paper.

### *A. Absolute rewards for all experiments*