NEURAL PROGRAM SYNTHESIS WITH SOFT ACTOR CRITIC

FINAL REPORT FOR CS395T

Lucas Kabela
lak2258
lucaskabela@utexas.edu
University of Texas at Austin
Department of Computer Science

ABSTRACT

In a programming by example (PBE) program synthesis framework, supervised learning has shown to be an effective approach to synthesis for sequential programs over string transformations as in Devlin et al. (2017). However, supervised learning only optimizes for exact program matches to a reference program, which may ignore programs which are consistent with the reference program on the examples provided. To combat the under specification of PBE, a more direct training objective with consistency in mind should be utilized. Thus, to directly optimize for program consistency, we apply reinforcement learning to the domain of sequential string transforming programs. We propose using Soft Actor Critic for this setting as it has low sample complexity, and it maximizes entropy, which encourages exploration and is thus able to return a larger space of candidate solutions. Using SAC to fine tune a pretrained policy network, we observe more beams with greater than 4 consistent programs. Furthermore, we note an improvement of consistent programs over REINFORCE by roughly 4.4%, and a 10.4% higher consistency than the supervised learning baseline.[1]

1 INTRODUCTION

In program synthesis, automated learning has recently rose to prominence, with the most popular of such techniques being neural program synthesis. In neural program synthesis, a neural network is trained on a handful of input/output pairs to predict a program. This task can be viewed as a form of supervised sequential generation, as in Devlin et al. (2017), which take the approach of modeling synthesis as a sequence to sequence generation. This task first encodes the i/o examples into a hidden vector using a recurrent neural network (RNN), which is then decoded by another RNN, which produces probabilities over the tokens in the domain specific language. Viewed in this manner, neural program synthesis can be quite successful - in Devlin et al. (2017), the accuracy of this neural approach with a beam decoder of size 1000 rivals that of the enumerative Flashfill by Gulwani (2011), and is even shown to be more robust to extreme noise.

In the PBE setting, however, this supervised learning objective often does not capture useful evaluation metrics. For instance, even in the simple string transformation DSL proposed in Gulwani (2011), there are many programs which can perform the same operation on a set of input strings, or are consistent with each other. In PBE, we are often given a small handful of examples which under-specify the task. As such, a training objective which maximizes the likelihood of generating programs which are consistent is preferred to one which maximizes exact matches. Such an objective will thus provide a larger space of potentially satisfying programs, with further input/output examples refining the program. One such technique which enables us to encourage consistency by our models is reinforcement learning (RL).

RL has seen a resurgence thanks to the revolutionary results of Deep Q Networks as in the cases of Mnih et al. (2013), which developed a remarkably successful agent for Atari games, and Silver et al. (2016), which produced AlphaGo. In these game applications, there are often large action spaces -

[1] Code available online at https://github.com/Lucaskabela/prog-synth-SAC

for instance in the game of Go, there are 250 possible moves per turn. The state spaces can often be just as large, as was the case in Mnih et al. (2013), where the state was screen resolution (160 x 192) times the number of colors available. Since these agents succeeded in remarkably large state and action spaces, with sparse rewards as in the case of AlphaGo, where rewards were 0 for losing a game and 1 for winning, it is conceivable using such techniques may yield promising results in synthesis.

Two such techniques of particular interest are Soft Actor Critic (SAC) and Hindsight Experience Replay (HER). SAC, by Haarnoja et al. (2018), is an off policy gradient technique over continuous action spaces, where an actor network and several critic networks, which estimate the goodness of states, explore the environment with the goal of maximizing entropy. This objective encourages randomness and thus exploration while being sample efficient due to its off policy approach. This work further incorporates dual soft q networks, which minimizes the variance of learning. Similarly, HER by Andrychowicz et al. (2017) seeks to maximize sample efficiency through storing experiences from episodes in a replay buffer, with a modified goal, which is the state the agent visited next in the episode. The agent then learns from this buffer, which is enabled through an off policy technique, effectively learning from mistakes.

In this work we attempt to use these deep learning techniques mentioned above to fine tune the performance of Neural Program Synthesis with the goal of returning beams with a larger number of consistent programs. In the following section, we survey the literature for relevant works. In sections 3 and 4, we provide an overview of the methods used such as the domain specific language (DSL), Markov decision process (MDP) and neural architectures. In sections 5 and 6 we present our results and offer discussion.

2 Relevant Works

Our paper's primary approach is a modified seq-2-seq model, introduced by Sutskever et al. (2014). This model has seen wide success in many natural language processing problems such as neural machine translation in Vaswani et al. (2018), and applications in neural program synthesis as well in Devlin et al. (2017).

Seq-2-seq models however are conventionally supervised models and trained through a technique such as teacher forcing, where the golden labels are fed into the decoder. Keneshloo et al. (2018) present a method for modifying the seq-2-seq model to be more compatible with deep reinforcement learning techniques, and our particular formulation follows most closely to Kandasamy et al. (2017), who modify an off policy gradient method for use with training chat bots.

To address the many issues with applying reinforcement learning in this setting, we turn to prior work such as Christodoulou (2019), which provides a formulation of soft actor critic for discrete action spaces, Andrychowicz et al. (2017), which serves as the source of our Hindsight experience replay, and Haarnoja et al. (2018), which provides the neural architecture for Soft Actor Critic.

Prior program synthesis approaches have operated on the same or very similar string transformation DSL's. Our approach builds directly on the work of Devlin et al. (2017), who use a seq-2-seq model with attention to synthesize transformations in a supervised setting. A similar approach is taken in Parisotto et al. (2016), which uses a R3NN to produce a neuro-symbolic approach to synthesis and is compared to Balog et al. (2016) which uses neural networks to predict properties of programs, augmenting enumerative and SMT approaches. Zohar & Wolf (2018) further builds on the algorithm of Deepcoder and Robustfill to synthesize longer programs and learn a garbage collector to do so. Finally, all of these works directly build on the enumerative synthesis presented in Gulwani (2011).

3 Problem Setting

We now define the problem setting, domain-specific language, and Markov decision process.

3.1 Domain Specific Language

The Domain Specific Language (DSL) used in this work is identical to that of Devlin et al. (2017). This language is used for string transformations and is based on operations performing string con-

Program P := `Concat(e1, e2, e3, ...)`
Expression e := `f | n | n1(n2) | n(f) | ConstStr(c)`
Substring f := `SubStr(k1, k2)| GetSpan(r1, i1, y1, r2, i2, y2)`
Nesting n := `GetToken(t, i) | ToCase(s) | Replace(` δ_1, δ_2`)`
`Trim() | GetUpto(r) | GetFrom(r) | GetFirst(t, i) | GetAll(t)`
Regex r := `t1 | · · · | tn |` δ_1 `| · · · |` δ_m
Type t := `Number | Word | AllCaps | PropCase | Lower | Digit | Char`
Case s := `Proper | AllCaps | Lower`
Position k := `-100, -99, ..., 1, 2, ..., 100`
Index i := `-5, -4, -3, -2, 1, 2, 3, 4, 5`
Character c := `A - Z, a - z, 0 - 9, !?, @`
Delimiter δ := `&, .?!@()[]\%{}/ :;#"`
Boundary y := `Start | End`

Figure 1: Syntax of the DSL

versions, substring extractions, constant strings, and nested expressions. The syntax is provided here in Figure 1. See Devlin et al. (2017) for the precise semantics.

A program, P, maps strings to strings, and is comprised of a concat operator used to concatenate substring expressions, nestings, or a constant string. Substrings are either defined by indices or the get span operator. Nesting expressions allow other forms of string modification such as casing, trimming, replacing, or getting up to or from occurrences of characters.

In total, we have 1,132 tokens in the DSL which are derived from the operators, types, cases, boundaries, integer positions, characters, and all enumerated arguments for Replace, GetUpto, GetFrom, GetFirst, GetAll, the end of sequence (EOS) token and the start of sequence (SOS) token. It is worth noting that we could have chosen to greatly reduce the number of tokens by not enumerating the arguments for the operators listed, but then the length of programs would vastly increase, which is a major issue for this simple seq-2-seq models. Thus, we chose to enumerate these tokens to limit the length of an interesting program.

3.2 Markov Decision Process

In our work, we use the formalism of the Markov Decision Process (MDP). In particular, we formulate our problem as a MDP defined by the tuple (S, A, r) where S is the infinitely large state space, A is the finite action space, and $r : S$ x $A \rightarrow \{0, 1\}$. States comprise of tuples, *(program sequence, examples)* where *program sequences* are a list of tokens which is unbounded in length, and *examples* are the PBE input/output examples. In this work, we used a fixed number of 4 examples as this number of i/o examples performed best in Devlin et al. (2017). An action is a integer, $a \in [0, 1132]$, where 0 is the end of sequence (EOS) token, 1132 is the start of sequence (SOS) token, and other integers map to the tokens described in the previous example, which are computed in a table. Finally, the reward function r is defined: $r(s_t, a_t)$ and is 0 for all $a_t \neq 0$, and if $a_t = 0, r = 1$ if and only if *program sequence* $\in s_t$ is consistent with *examples* $\in s_t$. We define consistency as, for an interpreter I, I(program sequence)$(i) = o$ for each i, o in *examples*.

3.3 Datasets and Formulation

To generate training data, and reference programs used by our environment for reinforcement learning, we synthesize data automatically via random sampling. Similar to Devlin et al. (2017) we sample programs up to a max length, then randomly generate input strings which the program executes successfully on. We then use the output from this execution and input as one of the four examples. Contrary to prior works, we did not have access to FlashFillTest, so we evaluated on randomly generated data as well.

While in theory, the random data will train the model to understand the semantics of the operators, it does not closely resemble real data. Thus, we believe, if work continues in this domain, it would be worthwhile for researchers to invest in creating a composite, real world dataset, as real use cases for

these string transformation programs are likely to have a much higher degree of exploitable structure than this randomly synthesized data.

Lastly, we note one of the attempts to make reinforcement learning tractable on this problem was to use curriculum learning. In particular, we began by restricting generation to 2 - 3 length programs which mostly comprised of constant string and simple nested expressions such as casing or trimming. Then, for every 1000 programs we get correct, we expand the maximum length of programs sampled by 1. In this way, we attempt to first have the models learn the individual operators, and then learn how they interact through composition and concatenation.

4 Model Architectures

We build on prior neural architectures used in this domain. We use a seq-to-seq model introduced by Sutskever et al. (2014) with the P network, which is used for generating the output program, attending to O, the encoder for output examples, which attends to I, the input encoder. We use an embedding layer to encode the initial characters as well as an embedding layer to encode the tokens from partially generated sequences. Since multiple examples are provided for each program, we use the same strategy of maxpooling among examples, and finally softmax to produce a probability distribution over tokens. (note, this is the Seq-2-seq Attention A architecture from Devlin et al. (2017))

4.1 Supervised Baseline

We used the model above with supervised learning techniques to establish a baseline performance. In particular, we use teacher learning, with a teacher force ratio parameter we decrease as we train longer. With teacher learning, we use the ground truth sequences as the input into the decoder instead of the argmax predicted. This is so that the model can observe the transitions which are in the actual programs, and not bias itself on the output produced. For hyperparameters, we used embeddings of size 128, recurrent and hidden sizes of 512, and the Adam optimizer with a learning rate of 5e-4 and gradient clipping of .25, in accordance with the best hyperparameters from Devlin et al. (2017). The baseline model was trained for 20,000 batches with batch sizes of 128, with 4 examples per element in the batch, for a total number of 2.56 million programs.

4.2 Reinforce with Baseline

Next, we took the model defined above as the policy network for the REINFORCE algorithm to establish a baseline performance for reinforcement learning. We further use a value network as a baseline to reduce variance. This value network is comprised of a RNN with 2 fully connected layers. Thus, we take the program sequence from the state, embed it, then compute the final hidden state with the RNN, which we then pass into the fc layers to produce a scalar value. This value serves to estimate the reward we get from the state.

REINFORCE is an on policy method, which means it can only learn from the episodes it sees. Thus, starting with a random policy network, it is almost impossible to learn anything meaningful. Based on random initialization, we have that the probability we correctly generate a size 2 sequence (the smallest generated by our sampler) is $\frac{1}{1,281,424}$ - in other words, we must train on over 1 million sequences before we are likely to see rewards. Therefore, we use the supervised network as a pre-trained policy, and attempt to improve performance from there. The policy network hyperparameters are the same as described above, and we have a hidden size of 512 for the value network. Note the token embedding is shared between the value and policy network, and our Adam optimize had an learning rate of 1e-6.

4.3 Soft Actor Critic

Finally, we implement a more sophisticated soft actor critic model. In the original version of the paper, SAC is only defined for continuous action spaces - however, using a simple trick with the GumbelSoftmax distribution, we get a differentiable distribution over discrete actions, and thus can operate on discrete action spaces.

We use two q networks, and two target q networks to minimize variants, with soft tau updates of 1e-3, as well as an automatically learned entropy term. We visualize this network (noting the target q networks are soft copies of the q networks) in Figure 2

However, once more we observe the action space is so large the chance of randomly reaching a reward is very small. Therefore, we turn to Hindsight Experience Replay (HER), a simple idea to handle this sparse reward space. The primary idea motivating HER is to turn trajectories which fail to acquire rewards into informative examples by changing the "goal" of the trajectory from the environment reward to the final state which was reached. In this way, the model can learn from failures by examining what state it ended up end, which "simulates" making the reward space dense. Once more our Adam optimizer had a learning rate of 1e-6

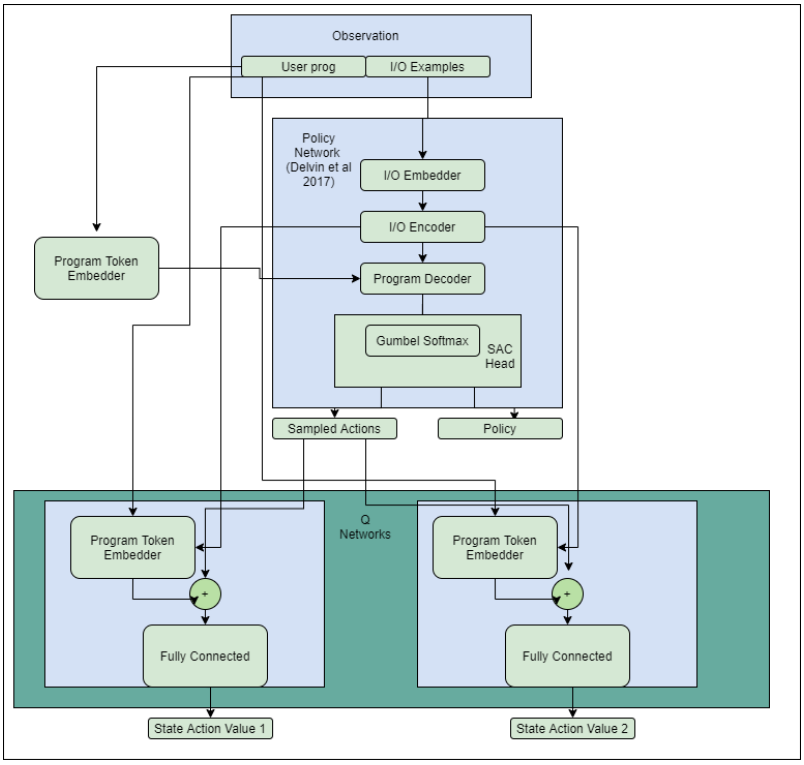

Figure 2: The SAC architecture for this project

5 RESULTS

We trained the baseline model, as described for 20,000 batches (2.56 million programs) on a P100 for 24 hours to provide a pretrained policy network. For evaluation purposes, we then further trained with supervised learning, REINFORCE, SAC, and SAC + HER for 10,000 batches (1.28 million programs), and evaluated the best models on exact matches and consistency. We evaluated two metrics, exact match (EM) and consistency on beam sizes of 1, 10, and 100. We report the percentage of beams which contained at least 1 exact match or 1 program consistent with the i/o examples in tables 1 and 2, averaged over 5 trails of 1000 reference programs. Finally, to test our hypothesis on the entropy of SAC and the space of candidate solutions, we run one more experiment, which evaluates how many programs in the beam were consistent with the i/o examples for a beam size of 10. These results are included in Figure 3.

6 DISCUSSION AND FUTURE WORKS

These results should be interpreted with some skepticism as we were not able to reproduce the accuracy of Devlin et al. (2017), so it is possible these results would not be as pronounced if the

Beam Size	1	10	100
Supervised	18.12	**26.94**	**34.30**
REINFORCE	18.94	25.82	32.20
SAC	**19.58**	26.02	31.60
SAC + HER	18.40	25.34	29.8

Table 1: Exact Match Accuracy vs Beam Size

Beam Size	1	10	100
Supervised	29.36	42.60	54.00
REINFORCE	33.32	51.12	63.80
SAC	37.68	**51.91**	**64.40**
SAC + HER	**37.76**	51.46	61.12

Table 2: Consistency vs Beam Size

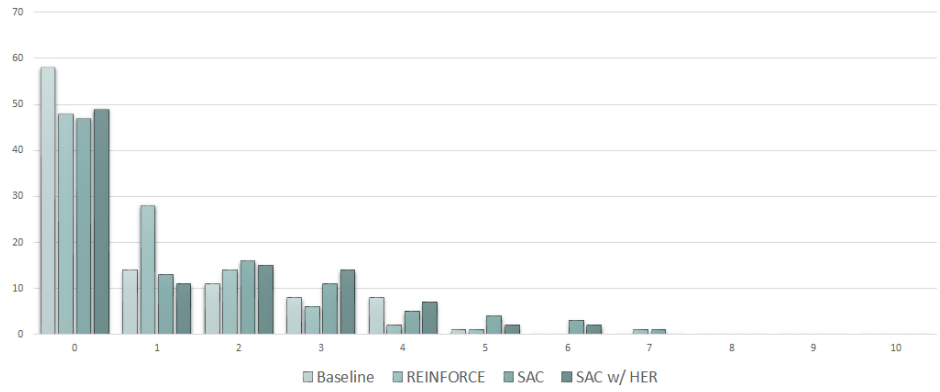

Figure 3: Number of consistent programs in a beam of size 10 over 100 samples

model was trained to its full capacity (we note we only trained on 1/10th of the programs as the original work due to inefficiency in our implementation).

We observe that the improvement in consistency of RL methods is inline with our initial hypothesis and indicates RL can be used to fine tune neural program synthesis, with up to 10.4% higher consistency at non statistically significant differences in EM. Furthermore, SAC and SAC + HER had significantly (p = .000106) higher consistency for beam size 1 than REINFORCE, which we believe to be due to the sample efficiency of off policy learning.

And, consistent with our hypothesis on entropy encouraging exploration, we observed that SAC and SAC + Her had a greater number of beams with greater than 4 consistent programs. However, we note that this could be a property of off policy learning in general and believe more work should be done with off policy deep learning algorithms to investigate the true effectiveness of entropy.

One disappointing note was we were not able to train an RL algorithm from scratch (hence why we used a pretrained policy network). We attribute this to the sparsity of the reward and high dimensionality of the action space. HER, which we expected to overcome the sparsity of the space was also ineffective, but we believe this issue to be caused by the static input/output examples. Modifications such as hallucinations by Sahni et al. (2019) may be able to overcome this limitation by "hallucinating" the correct i/o examples for trajectories. Another option for maximizing sample efficiency is Prioritized Experience Replay by Sahni et al. (2019), which may lead to better results for training from scratch by weighting experiences with non zero reward. For now our results are consistent with Zohar & Wolf (2018), who experimented with RL approaches in a similar domain and found such techniques intractable in learning from scratch. We do not believe the problem however to be unsolvable with reinforcement learning, but more sophisticated techniques for augmenting the reward space will be required.

7 Conclusion

In conclusion, we observe that reinforcement learning techniques can be used as effective tools for fine tuning the performance of neural program synthesis with regards to objectives such as consistency on input/output examples. We observe up to 10.4% more beams with at least one consistent program using Soft Actor Critic vs a supervised baseline with no significant drop in exact match accuracy. Furthermore, we observe that Soft Actor Critic contains significantly more beams with four or more consistent programs than supervised learning, which we attribute to the techniques maximization of entropy. This larger beam of viable candidate solutions can be used with further input/output examples to refine programs returned by our system. While we note our results are limited due to not reaching the upper bounds of accuracy with this technique and reliance on a pretrained policy network with supervised learning, we are hopeful that future works can explore techniques in reinforcement learning for making the reward space richer to train models from scratch in this domain.

Acknowledgments

We would like to acknowledge Dr. Swarat Chaudhuri for the suggestion of using HER to improve model performance and a great semester. Thank you!

References

Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017. URL http://arxiv.org/abs/1707.01495.

Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *CoRR*, abs/1611.01989, 2016. URL http://arxiv.org/abs/1611.01989.

Petros Christodoulou. Soft actor-critic for discrete action settings, 2019.

Jacob Devlin, Jonathan Uesato, Surya Bhupatiraju, Rishabh Singh, Abdel rahman Mohamed, and Pushmeet Kohli. Robustfill: Neural program learning under noisy i/o, 2017.

Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*, January 2011. URL https://www.microsoft.com/en-us/research/publication/automating-string-processing-spreadsheets-using-input-output-examples/.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor, 2018.

Kirthevasan Kandasamy, Yoram Bachrach, Ryota Tomioka, Daniel Tarlow, and David Carter. Batch policy gradient methods for improving neural conversation models, 2017.

Yaser Keneshloo, Tian Shi, Naren Ramakrishnan, and Chandan K. Reddy. Deep reinforcement learning for sequence to sequence models. *CoRR*, abs/1805.09461, 2018. URL http://arxiv.org/abs/1805.09461.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013.

Emilio Parisotto, Abdel rahman Mohamed, Rishabh Singh, Lihong Li, Dengyong Zhou, and Pushmeet Kohli. Neuro-symbolic program synthesis, 2016.

Himanshu Sahni, Toby Buckley, Pieter Abbeel, and Ilya Kuzovkin. Visual hindsight experience replay. *CoRR*, abs/1901.11529, 2019. URL http://arxiv.org/abs/1901.11529.

David Silver, Aja Huang, Christopher J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–503, 2016. URL http://www.nature.com/nature/journal/v529/n7587/full/nature16961.html.

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. *CoRR*, abs/1409.3215, 2014. URL http://arxiv.org/abs/1409.3215.

Ashish Vaswani, Samy Bengio, Eugene Brevdo, François Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Lukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, and Jakob Uszkoreit. Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416, 2018. URL http://arxiv.org/abs/1803.07416.

Amit Zohar and Lior Wolf. Automatic program synthesis of long programs with a learned garbage collector. *CoRR*, abs/1809.04682, 2018. URL http://arxiv.org/abs/1809.04682.