Project Report: Application of Reinforcement Learning to Shift-Reduce UCCA Parser

Deep Reinforcement Learning(COSC-689), Spring 2019
Professor: Grace Hui Yang, Ph.D.

Yan Yang
Georgetown University
yy490@georgetown.edu

Tianyu Yang
Georgetown University
ty233@georgetown.edu

ABSTRACT

In this project, we offer a model-based reinforcement framework for shift-reduce parsing. An environment simulating the transition of states and correctness of action choices is modeled, and a parser agent is trained through policy gradient method and tested on the UCCA annotated data provided by CodaLab.

1 INTRODUCTION

Current semantic parsing schemes include UCCA[1], AMR[2], SDP[8], UD[7], etc. Both UCCA and AMR utilize the Directed Acyclic Graph (DAG) structure to represent semantic relations within sentences, or also beyond sentences. Producing semantic graphs is a challenging task due to its reentrancy and non-projectivity nature, and the currently focused task in semantic parsing, the Shift-Reduce parser, and algorithms to implement it, has been discussed widely in literature. In this work, we plan to apply Reinforcement Learning to Shift-Reduce parsing, training and testing a model on UCCA annotated data.

2 BACKGROUND

Reinforcement learning is widely used to train robot agents, such as AI game players[11], self-driving cars[9], etc. Mimicking human decisions as a robot is right the task a shift-reduce parser does, and compared to the prevalent sequence to sequence parsing, applying reinforcement learning is capable of making decisions at each transition step separately, which utilizes more fine-grained information.

Two works published on UCCA parsing [5] [6] presented TUPA, a DAG parser based on a BiLSTM-based classifier, both utilizing the shift-reduce transition set. Goodman and Vlachos[4] used imitation learning for shift-reduce task in AMR parsing. The reinforcement learning trials taken by Zhang & Chan[13] and Sallans & Hinton[10] both used the negative free energy as state-action function and applied SARSA to optimize it. The former considered only 4 action types, which is not suitable for the complex action set of UCCA, while the latter contains some settings which we could taken as future improvements to our own model. We established a policy gradient reinforcement learning framework for the task, performed some experiments, and will discuss these and other future improvements at the end of the paper.

3 DATA

The data comes from the competition SemEval 2019 Task 1: Cross-lingual Semantic Parsing with UCCA[1], on CodaLab. We experimented on the English Wiki data with UCCA annotation. Both training and development datasets contain pre-obtained linguistic features including lemma, part of speech, word shape, entity type, etc. Also, golden UCCA annotations are provided. Some Python scripts[2][3] from TUPA code repository shows that, as long as we have access to the UCCA-annotated .xml tree representation, we have access to its golden oracle sequence in the form of, for example, SHIFT, NODE-Terminal, REDUCE, SHIFT, NODE-C, REDUCE, SHIFT, RIGHT-EDGE-L, etc., each step being an option from the "transitions" column in Figure-2.

4 PRELIMINARY MODEL

With code available at https://github.com/DIMPLY/DRL_UCCA, we implemented three main building blocks for this Reinforcement Learning algorithm: 1. the "correctness" function training, including training data extraction and a neural network which is trained, tested, and stored; 2. a customized "environment" for openAI gym package[3], where a reward function based on the trained "correctness" function is plugged in; 3. a Policy Gradient algorithm for the reinforcement training.

Other parts, such as the State model and Action model which are also plugged in in the customized environment, and the reader of the raw annotation files, are adapted from the Tupa code repository[5] and UCCA[1] Python package.

The high level flow chart of this program is shown in Figure 1. We will further describe details of some of the building blocks, and these descriptions are followed by our trials and analyses to improve this preliminary model.

4.1 The Environment

An episodic[4] reinforcement learning environment requires minimally an initial state setter, a state-transition model, and a reward function.

4.1.1 Transition Model. The transition model is straightforward from Figure 2. And in practice, this is realized by directly calling the "transition" method of the "State" class reused from Tupa repository.

[1]https://competitions.codalab.org/competitions/19160
[2]https://github.com/danielhers/tupa/blob/master/tupa/oracle.py
[3]https://github.com/danielhers/tupa/blob/master/tests/test_oracle.py
[4]An episodic task is divided into trajectories each with an ending state, compared to continuous tasks which never end. For shift-reduce parsing, each episode parses one sentence and ends when the "FINISH" action is taken.

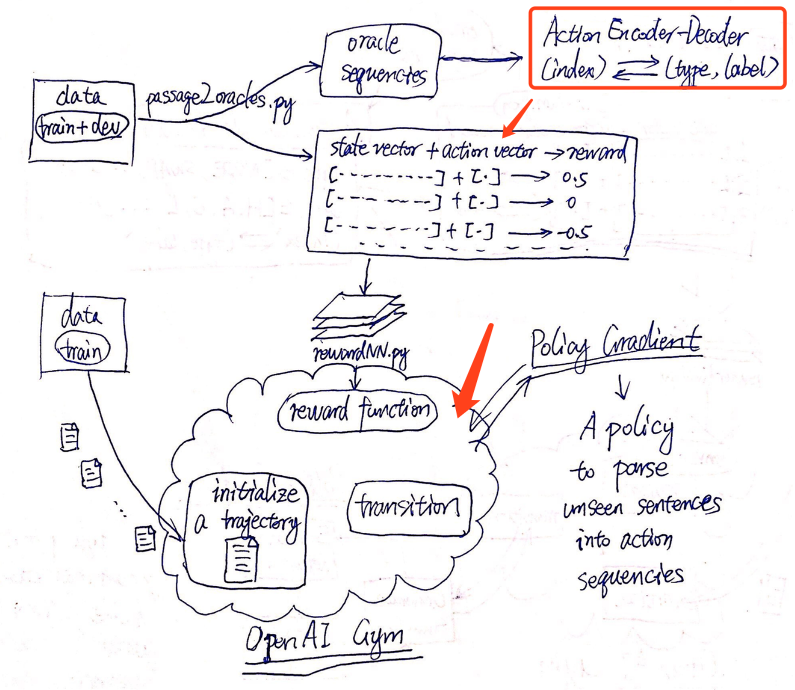
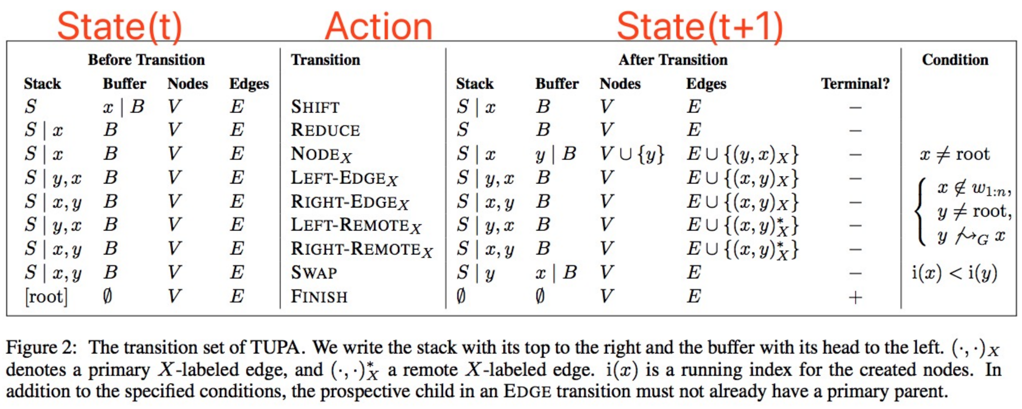


Figure 1: The Flow Chart of Method Used in This Paper.


State(t)
Action
State(t+1)

Before Transition				Transition	After Transition					Condition
Stack	Buffer	Nodes	Edges		Stack	Buffer	Nodes	Edges	Terminal?	
S	$x \mid B$	V	E	SHIFT	$S \mid x$	B	V	E	$-$	
$S \mid x$	B	V	E	REDUCE	S	B	V	E	$-$	
$S \mid x$	B	V	E	NODE$_X$	$S \mid x$	$y \mid B$	$V \cup \{y\}$	$E \cup \{(y,x)_X\}$	$-$	$x \neq$ root
$S \mid y, x$	B	V	E	LEFT-EDGE$_X$	$S \mid y, x$	B	V	$E \cup \{(x,y)_X\}$	$-$	$\left\{ \begin{array}{l} x \notin w_{1:n}, \\ y \neq \text{root}, \\ y \not\leadsto_G x \end{array} \right.$
$S \mid x, y$	B	V	E	RIGHT-EDGE$_X$	$S \mid x, y$	B	V	$E \cup \{(x,y)_X\}$	$-$	
$S \mid y, x$	B	V	E	LEFT-REMOTE$_X$	$S \mid y, x$	B	V	$E \cup \{(x,y)^*_X\}$	$-$	
$S \mid x, y$	B	V	E	RIGHT-REMOTE$_X$	$S \mid x, y$	B	V	$E \cup \{(x,y)^*_X\}$	$-$	
$S \mid x, y$	B	V	E	SWAP	$S \mid y$	$x \mid B$	V	E	$-$	$\mathrm{i}(x) < \mathrm{i}(y)$
[root]	$\emptyset$	V	E	FINISH	$\emptyset$	$\emptyset$	V	E	$+$	

Figure 2: The transition set of TUPA. We write the stack with its top to the right and the buffer with its head to the left. $(\cdot,\cdot)_X$ denotes a primary X-labeled edge, and $(\cdot,\cdot)^*_X$ a remote X-labeled edge. $\mathrm{i}(x)$ is a running index for the created nodes. In addition to the specified conditions, the prospective child in an EDGE transition must not already have a primary parent.

Figure 2: The transition model for Shift-Reduce parsing. [5]

The Tupa "State" class also offered validation to exclude invalid transition commands.

4.1.2 Reward Function. The reward function is built upon a mapping from the current state/action pair to the score we designed by judging the correctness of taking that action under that state. We call this score "correctness", and the reward function in our environment is a combined use of the correctness function and some hard-coded constraint checking.

By translating all golden sentence tree structures to golden oracle sequences, and recording the state/action pairs along the oracle sequence, we get a large number of training data. For a state along the trajectory, we each time pair it with one action, and enumerate all actions in the available set. When the action accords with the current golden oracle, we record the score as 0.5, if not, the score is -0.5. A half match situation is considered when the action type matches but the label type doesn't. The score at this situation is 0.

The intuition for this is that we balance the positive and negative rewards to avoid encouraging the agent to take longer episodes as when all rewards are positive, or shorter episodes as when all rewards are negative. The actual result and an analysis of this reward setting is discussed in Section 5.

Data collected in this way allows us to train a model of score prediction, which can be generalized to state/action pairs that are never seen in the training set. Considering that the state representations, consisting of words and their features, are in a very sparse vector space, and we will seldom see a state for a second time, this ability of generalization is important.

4.1.3 Episode Reset. The last requirement of a customized environment is the initial state setter. It's also adapted from Tupa by straightforwardly initializing a "tupa.State" class.

4.2 Vector Representations

Actions and states in shift-reduce parsing are not mathematical abstract. in Tupa, they are represented as class instances. We need vectors to represent them for our various models.

4.2.1 State Representation. Tupa's DenseFeatureExtractor provides a 35-dimensional representation of a "State" instance, but during data preprocessing, we found that 10 of the 35 dimensions are always 0. We directly use this remaining 25-dimensional information as the representation of a state, and as the Tupa paper says, it contains information about "words, POS tags, syntactic dependency labels, edge labels, punctuation, gap type and parser actions."[5]

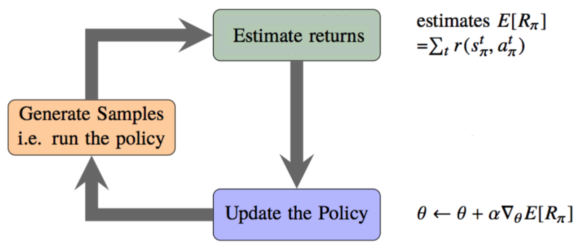


Figure 3: The anatomy of Reinforcement Learning using Policy Gradient.

In our work, we used this representation in both the correctness function trainer and the policy trainer. The necessity for a further investigation of this feature representation will be discussed in Section 6, together with results of the neural learners fed with it.

4.2.2 Action Representation. We went through all the golden oracles in training and development data sets using a script, and concluded that there are 4 kinds of actions without labels: 'SHIFT', 'REDUCE', 'SWAP', and 'FINISH', and 6 kinds of actions each used with one label chosen from a 14 label set. The 6 actions are 'IMPLICIT', 'NODE', 'RIGHT-EDGE', 'LEFT-EDGE', 'RIGHT-REMOTE', and 'LEFT-REMOTE', and the 14 labels are 'H', 'A', 'C', 'L', 'D', 'E', 'G', 'S', 'N', 'P', 'R', 'F', 'Terminal', and 'U'.

Enumerating the possibilities of action type/label pairs produces 4 kinds of non-labeled actions and 6×14 kinds of labeled actions in total, so we use each number in range 0–87 to represent one action. Converting it back to a type name and an optional label name only requires simple maths calculations.

4.3 Policy Gradient

The anatomy of policy gradient is shown in Figure 3 [5]. After defining the network and initialize its weights θ, we repeatedly go through three steps as illustrated in this figure. First, we run the current policy to generate a batch of trajectory samples. Each trajectory takes one training file as its input to decide the initial state. Then we calculate the actual return at each step and use it as the estimated average return (a Monte Carlo method). Last, we use the original state, the action to take, and the calculated return at each step of each trajectory to update the policy network parameters θ iteratively. This is approximately the algorithm in Figure 4, but differs in that we generate multiple trajectories in each iteration rather than a single one, and we finish calculating all returns in an iteration before continuing to update θ step by step.

5 ENVIRONMENT AND MODEL ADJUSTMENT

The initial round of testing results will be shown together with final results in Section 6. They suggested that improvements on several aspects are needed.

5.1 Correctness Score

5.1.1 In Preliminary Model. We considered different correctness score schemes for the correctness function training data. As discussed in Section 4.1.2, an all positive correctness score schema, such as giving 1.0 when action taken correctly and 0.0 when it goes wrong, will encourage the agent to produce trajectory lengths approaching infinity. And an all negative correctness score schema, such as giving -1.0 when the predicted action is wrong and 0.0 when it's right, will encourage the agent to take "Finish" action immediately when an episode starts.

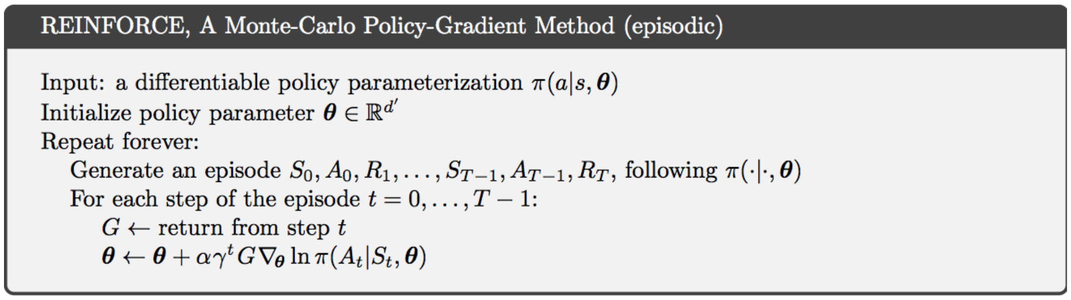
A neutral schema sounds reasonable, and in training practice, due to the limitation of disk space, we used only 100 files to extract correctness function training data. Approximately 2 million items were stored, each containing a state, an action, and their corresponding correctness value. The learning curve of the correctness function shown in Figure 5 looks good.

However, actually the resulted trajectory becomes either too short or too long. The golden oracle sequences usually have several hundred oracles, and my sampled trajectories contain relatively reasonable numbers of steps using a randomly initialized policy, such as a batch of 10, 132, 318, 38, 89, 141, ..., and 54. However, as the policy gradient method is trained on more trajectories, sometimes it becomes like, 3, 13650, 2, 1, 1, 56779, etc., sometimes it just produce all 1's.

It seems that the correctness score setting need to be further tuned, such as, presumably, using -0.3 for wrong actions and +0.7 for right ones, but we first increased the accuracy of training in the following ways.

5.1.2 Improvements. First, we used one-hot vector to better represent the action part of the input. This is reasonable because different

[5]Adapted from 2019 Spring Deep Reinforcement Learning course slides



REINFORCE, A Monte-Carlo Policy-Gradient Method (episodic)

```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$
Repeat forever:
    Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
    For each step of the episode $t = 0, \ldots, T-1$:
        $G \leftarrow$ return from step $t$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t G \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t, \boldsymbol{\theta})$
```

Figure 4: REINFORCE: the Policy Gradient Algorithm.[12]

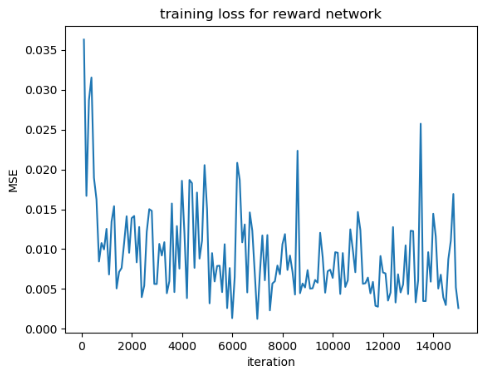


Figure 5: Training curve of correctness function, plotted every 1000 iterations, with each iteration taking a batch of 128 data points.

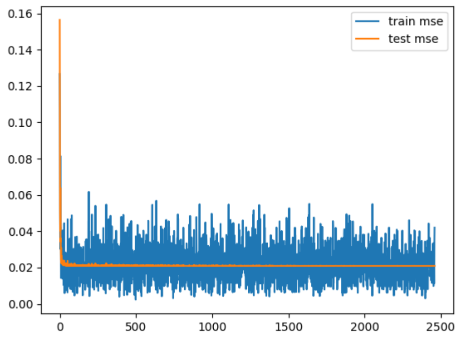


Figure 6: Training curve of correctness function, plotted every 1000 iterations, with each iteration taking a batch of 128 data points. Y axis is the square loss.

actions are not conceived in an ordered way like numbers from 0 to 87. If they are treated equally like in one-hot vectors, the result might be better.

Second, we increased the amount of files for training using a GPU server and used the development set separately from training set to watch for overfitting. The result as in Figure 6 shows no overfitting, but nor was improvements shown after an initial rapid loss decreasing.

Third, the original class is skewed, with around 95% incorrect instances, 3%–4% half-correct instances, and only 1% correct ones. Therefore, we down-sampled the incorrect action instances to have balanced classes. And the resulting learning curve as in Figure 7 shows that less data has introduced overfitting, which further requires to cut off training early.

Considering the overfit, we dropped the rest of the huge amount of data we have collected.

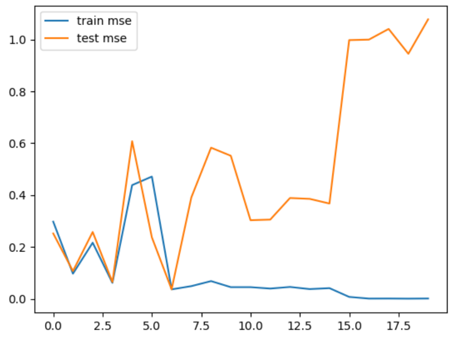


Figure 7: Training curve of correctness function with down-sampling, plotted every 1000 iterations, with each iteration taking a batch of 128 data points.(Y axis is the square loss)

5.2 Reward Function

5.2.1 In Preliminary Model. Note that the above discussion of correctness score settings are all about collecting correctness function training data. After the function is trained and stored, we don't need to decide the correctness score ourselves. It will be calculated from the trained correctness function, and can be any value between correct (like 0.5) and incorrect (like -0.5), and for calculating the reward, we catch any error produced by non-viable actions in Tupa's State class to give feedback on clearly invalid actions produced by the RL agent. Besides giving a manually set negative reward value (here we used -0.5 to be identical with all other "incorrects"), when invalid actions are detected, we can continue updating the policy network parameter θ here. This latter is to be done in the future.

However, with the better trained correctness function, and the reward being the correctness value when no hard-coded constraint is violated, and -0.5 when the selected action is rejected due to the constraints, the previous described agent behavior in Section 5.1.1 lasts.

5.2.2 Improvements. We considered that the single correctness value 0.5, which denotes the successful prediction of a single step, accounts for different proportions in sentence parsing of different lengths. To solve this, we simply divided all positive correctness values by the sentence length as the first improvement to our preliminary model. The non-positive correctness values are left as they are, because due to the propagation of errors to later steps, we consider an incorrect step in a longer sentence as influential as in shorter sentences, or if it's in earlier stage of the parsing, maybe even worse.

Our second improvement is based on the observation of program output, where a lot of episodes with length one appeared. We inferred that there is no restriction of taking "FINISH" action before the stack and the buffer are both empty, as apposed to the rules in Figure 2. We speculate that there are more.

Therefore, validation rules are inserted to modify the original Tupa code, so that the RL agent is encouraged to strictly obey all rules we have, or otherwise receives negative reward as punishment.

We continued to add more rules besides those in Figure 2 according to our observation of outputs and our understanding of the nature of the task, which will be presented next.

5.3 Constraints on Actions

The valid action set changes as the agent experiences different states. We think taking actions that are invalid under current state is an important reason of the length-one episodes and the never ending episodes mentioned above. After adding constraints according to Figure 2, we observed that there're more unreasonable results. Therefore, we handcrafted our own conditional constraints.

First, we observed that too many "IMPLICIT" actions are making nodes faster than reducing them, which made an episode never end. A rule was added only allowing one Implicit child for each substructure.

Second, we observed that multiple edges are created between exactly the same pair of nodes, which never happens in manual UCCA annotation. A rule was created to forbid this.

Third, leaf nodes that are raw text tokens from the sentence have special rules. They allow no remote edge and only one incoming edge. Their parent nodes allow only one child when it's not a punctuation, while parents of punctuation leafs allow no remote edge and only one incoming edge, either.

Adding each of the rules has improved the result prominently according to our test.

5.4 Policy Gradient

Besides adding constraints in the environment, we also improved the policy gradient model.

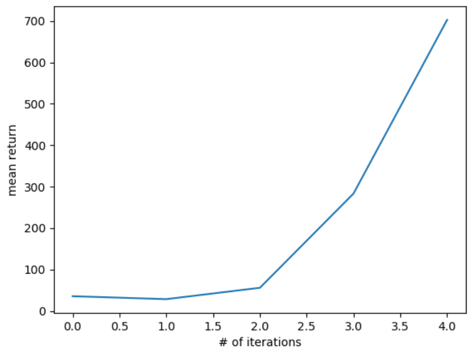


Figure 8: The average returns for 5 iterations (we produce a batch of 40 trajectories in each iteration). This is stopped earlier than the default 20 iterations because trajectories are driven toward either 1 or infinity, the latter of which never ends.

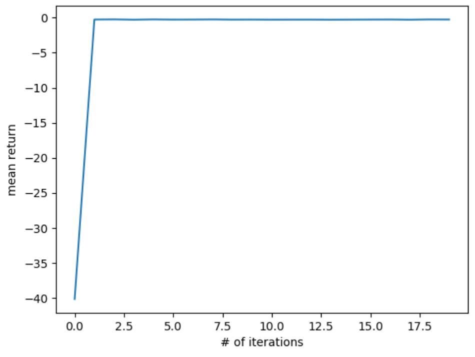


Figure 9: The average returns for 20 iterations (we produce a batch of 40 trajectories in each iteration).

5.4.1 In Preliminary Model. The first results of the policy learner are plotted in Figure 8 and Figure 9 as average returns per iteration. We used a batch of 40 trajectories for each iteration, and 20 iterations per training. After running the whole project for multiple times, we mostly got similar plots as in Figure 9, but luckily got some similar to Figure 8. Different computers running the identical code repository edition have produced different results.

While situations similar to Figure 9 usually end up with all episodes having only one step, an improvement to which has been presented in Section 5.1.1, situations similar to Figure 8 are problematical, too. With golden oracle sequences usually having several hundred steps, and each correct step getting 0.5 reward, the average return couldn't reach as high as around 800. It accords with that the actual lengths of episodes in this situation are driven toward either 1 or infinity, also mentioned in Section 5.1.1. The long episodes approaching infinity produced the large returns.

5.4.2 Weight Initializing. To better solve the problem of unending episodes, we used the same data set as in correctness function training, which was collected from the states along golden oracle sequences. We trained the neural network weights on this data set first, but only using the mapping from state vector to action one-hot, without considering the correctness score. Then we used the resulting weights as a better starting point in policy gradient.

The intuition was that since the golden actions picked on states along golden trajectories produce reasonable lengths, the weights trained on them are more close to producing reasonable lengths on unseen states, and are more suitable as starting weights in policy gradient since the latter requires running multiple trajectories before any weight update is run.

With the initializer training curve as in Figure 10, this effort combined with previous mentioned ones offered a much better functioning learning framework.

5.4.3 Parameter Tuning. During training, we decreased the learning rate from 0.1 to 0.001 to avoid "NaN" propagating throughout the weights. We decreased the batch size to 1 in the end, but during the tuning process, we also tried the varying batch size starting from 1 and increasing up to 40. We meant to shift weight updating in the beginning iterations to an earlier point, and after converging to reasonable episode lengths, recover the original batch size. But experiments proved that the constant batch size of 1 did the job.

Figure 11 shows the training curve after all the tuning on 60 iterations. The upper bound of the fluctuations show little improvement, but we get steadily improving lower bound of mean return.

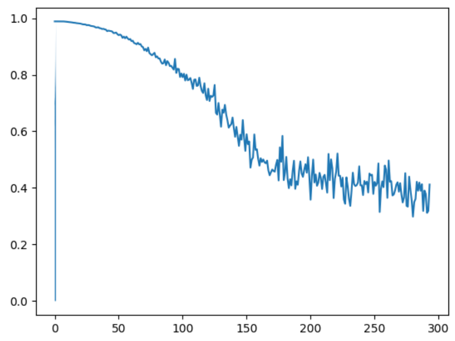


Figure 10: Training curve of policy weight initializer, plotted every 3200 iterations, with each iteration taking a batch of 128 data points. Y axis is the sum of squared loss averaged on a batch of data.

6 FUTURE WORK

6.1 State Representation

We didn't look into the representation of states, but used the one designed for TUPA directly. A better representation here is actually very important to further improvements. We might be able to learn from the usage of free energy in [13] and [10], which is a representation of state/action combination.

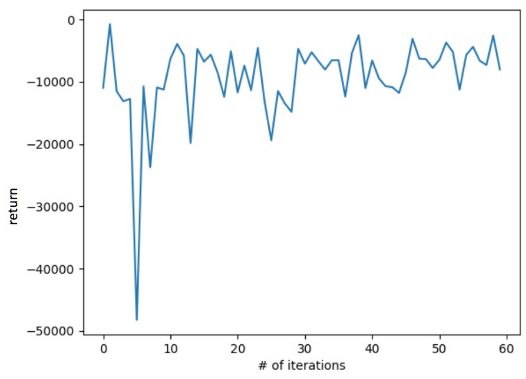


Figure 11: Performance of the policy network

6.2 Correctness and Reward

Although we applied various ways to improve correctness score schema and the reward function built upon it, when comparing the shape of the policy gradient training curve on the first 8 iterations to the lengths of first 8 files, we found that the episode returns are still prominently correlated with sentence lengths. This means better designs are needed for the correctness scores, the constraint punishments, and the combining rules of the previous two for the reward function.

Sallans and Hinton's work [10] described their way to model a reward function.

Also, by observing the test script output, we found structures of chain of nodes with single parent and single child, like 1.1 -N-> 84 -R-> 83 (remote) -L-> 82 -A-> 80 -C-> 79 -F-> 78 ...... -L-> 63 -G-> 61 -Terminal-> "National", which apparently contains redundant information for the mere purpose of representing sentence semantics. New action constraints can be designed for this.

6.3 Policy Gradient

We have tuned the parameters and gained some improvements in this part, especially reducing the learning rate to avoid 0 denominators in back propagation producing NaN. The remaining problems here are partly passed down from upstream blocks, so we need to improve the upstream performance before we can examine the extra information loss here. However, some general ways to improve a deep reinforcement learning method can be thought of.

For example, high-dimensional state space and the non-linearity of neural network usually cause the model to be highly unstable, so improvements such as using a value function as a baseline might be useful. We didn't implement it in this work, because a better observation of upstream performance improvement is needed and it's better to keep other parts stable.

Another future consideration is that, in contemporary DRL, reward function is learned via inverse RL[6]. From this perspective, we might be able to replace our current reward function learning method.

[6]Quote from Spring 2019 Deep Reinforcement Learning course slides.

7 CONCLUSIONS

We were able to connect a complete program flow according to the mere intuition of taking shift-reduce parser as “a robot making decisions step by step”. This proved the viability of applying deep reinforcement learning to shift-reduce parsing. Although the result is still not good, with unreasonably long episodes and all returns negative, we see great space of potential improvement. When the further improvements are done, we can decide its efficiency besides viability.

REFERENCES

[1] Omri Abend and Ari Rappoport. 2013. Universal Conceptual Cognitive Annotation (UCCA). In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 228–238. http://aclweb.org/anthology/P13-1023

[2] Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*. 178–186.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *CoRR* abs/1606.01540 (2016). arXiv:1606.01540 http://arxiv.org/abs/1606.01540

[4] James Goodman, Andreas Vlachos, and Jason Naradowsky. 2016. Noise reduction and targeted exploration in imitation learning for abstract meaning representation parsing. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Vol. 1. 1–11.

[5] Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2017. A Transition-Based Directed Acyclic Graph Parser for UCCA. *CoRR* abs/1704.00552 (2017). arXiv:1704.00552 http://arxiv.org/abs/1704.00552

[6] Daniel Hershcovich, Omri Abend, and Ari Rappoport. 2018. Multitask Parsing Across Semantic Representations. *CoRR* abs/1805.00287 (2018). arXiv:1805.00287 http://arxiv.org/abs/1805.00287

[7] Ryan McDonald, Joakim Nivre, Yvonne Quirmbach-Brundage, Yoav Goldberg, Dipanjan Das, Kuzman Ganchev, Keith Hall, Slav Petrov, Hao Zhang, Oscar Täckström, et al. 2013. Universal dependency annotation for multilingual parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, Vol. 2. 92–97.

[8] Stephan Oepen, Marco Kuhlmann, Yusuke Miyao, Daniel Zeman, Dan Flickinger, Jan Hajic, Angelina Ivanova, and Yi Zhang. 2014. Semeval 2014 task 8: Broad-coverage semantic dependency parsing. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)*. 63–72.

[9] Xinlei Pan, Yurong You, Ziyan Wang, and Cewu Lu. 2017. Virtual to real reinforcement learning for autonomous driving. *arXiv preprint arXiv:1704.03952* (2017).

[10] Brian Sallans and Geoffrey E Hinton. 2004. Reinforcement learning with factored states and actions. *Journal of Machine Learning Research* 5, Aug (2004), 1063–1088.

[11] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354.

[12] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (second ed.). The MIT Press. http://incompleteideas.net/book/the-book-2nd.html

[13] Lidan Zhang and Kwok Ping Chan. 2009. Dependency parsing with energy-based reinforcement learning. In *Proceedings of the 11th International Conference on Parsing Technologies*. Association for Computational Linguistics, 234–237.