Using RL for Non-Greedy Dependency Parsing

Stanford CS224N Custom Project

Sidhart Krishnan
Department of Computer Science
Stanford University
skrish24@stanford.edu

Arvind Saligrama
Department of Computer Science
Stanford University
arvind02@stanford.edu

Abstract

Neural networks were introduced for dependency parsing by Chen and Manning (2014) and through using neural networks, they were able to to make massive strides in transition-based dependency parsing. However, these transition-based dependency parsing mostly rely on greedy decoding at inference stage which means that if the model makes an error early on, then it parser will continue to diverge further and further away from the ground truth. Thus, we reconstruct a reinforcement learning algorithm to perform non-greedy decoding for transition-based parsers first shown by Shen et al [2016]. Our RL-based parser was tested on the English Penn Treebank (PTB) dataset and it achieved a 89.2% accuracy and improves by about 0.4 percentage points over a supervised dependency parser.

1 Key Information to include

- Mentor: Allan Zhou
- External Collaborators (if you have any): None
- Sharing project: No

2 Introduction

Today, one of the most important tasks in Natural Language Processing is definitely dependency parsing. It has been recently shown that dependency relationships can actually improve performance on a variety of NLP tasks. For example,in 2014, Levy and Goldberg showed that word-embeddings created with dependency-based contexts are less topical and exhibit more functional similarity than the original skip-gram modeling. Furthermore, in 2015, incorporating dependency knowledge in their model allowed Angeli et al. to beat the state of the art on the end-to-end TAC-KBP 2013 Slot Filling task. However, if a dependency parser is prone to making errors, incorporating dependency relationships can actually harm performance on the downstream task. So, it is essential to improve the accuracy of parsing methods.

In the literature, the problem of dependency parsing has been tackled using primarily two approaches. The first of these approaches treats dependency parsing as a structured prediction task and makes use of graph-based search algorithms. The second involves creating a shift-reduce parser that greedily produce dependency arcs for each word in order. The second of these methods has been used often due to its decoding efficiency. However, this method suffers in a way similar to many greedy approaches. If our model diverges from the ground-truth parse at any step, it will continue to diverge from there.

With the intuition that Dependency parsing seemed like a game in a sense, we decided to consider Reinforcement Learning approaches for parsing. In 2016, Shen et al. of Carnegie Mellon University improved on the performance of transition-based dependency parsing using Deep Reinforcement Learning. They built a reinforcement learning agent with transition-based shift-reduce parsing which learns non-greedy decoding by considering future rewards.

3 Related Work

A prominent strategy used to tackle dependency parsing has been trasition-based dependency parsing. This involves predicting a sequence of actions to parse a sentence. Nivre and Scholtz [2004] describe an arc-standard parsing system that maintains a stack and a buffer. The stack and buffer keep track of the progress so far, and the actions considered are shift, left-arc, and right-arc. Other transition systems build on this, such as the arc-eager transition system of Nivre [2004] which takes advantage of the incrementality in parsing. Then Nivre and Fernandez-Gonzalez [2014] enforced a tree-constraint to the arc-eager system without modifying the parser training procedure. This method consistently outperformed the standard heuristics.

Other works introduce Neural networks to the dependency parsing problem. These papers use neural networks to work with the arc-standard parsing system described by Nivre and Scholtz. Chen and Manning [2014] used neural networks to solve the problem of extensive manually designed features and the enormous computational complexity of feature extraction. They use neural networks to predict the arc labels for a given configuration. Dozat and Manning [2016] built on this by using a biaffine layer to compute attention over the combination of features. Yong et al. [2016] actually consider unsupervised dependency parsing. They learn a dependency grammar from only Part of Speech tags. The main supervised neural dependency work that we consider in this paper is the paper by Chen and Manning which utilized a neural network to predict actions for transition-based parsers. It performs reasonably well and uses greedy decoding, and we aim build on it via RL approaches.

Several authors have experimented with reinforcement learning on dependency parsing tasks. Zhang and Chan [2009] studied value-based learning approaches. To approximated their state-action function, they calculated the negative free energies for the Restricted Boltzmann Machine. Using their neural network as a policy network, Le and Fokkens [2017] use the approximate policy gradient to maximize the expected reward. Furthermore, Neu and Szepesvari[2009] used inverse reinforcement learning algorithms to tackle dependency parsing with context-free grammars.

The main paper which we build off of is Shen et al. [2016] which uses actor-critic based reinforcement learning to train an agent to parse the sentence. The authors claim that the series of actions necessary to parse each token of the sentence lends itself well towards being framed as an RL task. Moreover, the main benefit they aim to demonstrate is that an RL framework would focus on non-greedy decoding. They claim that the supervised models are trained to perform greedy decoding as they produce the dependency arcs for each word incrementally and are trained as such. Thus they claim that an RL-based parser would reduce error propogation as the parser aims to improve the overall parse instead of taking actions greedily. They use actor-critic based RL methods to create their parser and the parser demonstrates moderate improvement over supervised methods. Our goal is to both replicate the results of Shen et al, as the code for their RL-based dependency parser is not publically available. Additionally, we aim to to test other reward functions and examine other RL algorithms for the parser as these were areas highlighted as possible future work in Shen et al.

4 Approach

4.1 Transition-Based Dependency Parsing

The transition-based dependency parser works to create a dependency tree by predicting a sequence of transitions. Specifically, we are using the arc standard transition system. In this system the parsing configuration consistsn of a triple $\langle \Sigma, B, A \rangle$ where Σ represents a stack of the partially processing tokens in the sentence, B is a buffer of unprocessed tokens and A are the arcs in the current parsing tree. Additionally, there are three transition operators – SHIFT, LEFT-ARC and RIGHT-ARC – which represent the actions the parser can perform. The SHIFT action moves b_1 onto the stack Σ. The LEFT-ARC action adds an arc from the first element in the stack to the second element in the stack, while the RIGHT-ARC action adds an arc from the second element to the first. The initial configuration for a sentence s with tokens $w_0, w_1, \ldots, w_n$ is $\Sigma = [ROOT]$, $B = [w_0, w_1, \ldots, w_n]$ and $A = \emptyset$. This parser terminates when the buffer is empty, the stack contains one element and thus A contains the parse tree.

The baseline model that we compared our RL parser to was the supervised neural dependency parser from Chen and Manning. The parser uses a single linear layer to output action logits which it then uses to select a transition for the transition parser.

4.2 Reinforcement Learning

Since the method for transition-based dependency parsing requires an actor to decide a series of actions that ultimate leads to a final dependency tree, the problem of dependency parsing tends itself well for reinforcement learning. In order to use RL methods to approach a problem, one must first frame the problem as a Markov Decision Process (MDP). Formally, an MDP is a tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$ where where S is the set of all possible states, A contains all possible actions, $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ is the transition function, $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function and γ is a discount factor which is used to discount future rewards.

At each time step, t, given the current state is $s_t \in \mathcal{S}$, the parser takes an action $a_t \in \mathcal{A}$. This action should be given by a model and we can generalize the model as a policy $\pi : \S \to \mathcal{P}(\mathcal{A})$. Thus using the policy, we are able to generate actions either deterministically or sample them. From this, the expected accumulative reward R_t is defined by

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{1}$$

at the time step t. Thus we see that actions that come later in the sequence are weighted slightly lower. The agent's objective is to maximize the expected accumulative reward.

4.3 Parsing Environment

To formulate transition-based dependency parsing as an MDP environment, we define the set of states $\S$ to be the set of configurations $\langle \Sigma, B, A \rangle$. The set of actions $\mathcal{A}$ is $\{SHIFT, LEFT - ARC, RIGHT - ARC\}$. The transition function is given by performing the action to the current parser configuration to get a new configuration that is our new state. Finally, the reward function that we decided to use deviates from the reward function that is used in Shen et al. In that paper, they used the negative Hamming Loss between the golden parse and the current partial parse meaning that for a sentence of length T, the loss is

$$r_t = -\sum_{i=1}^{T} 1[y_i \neq \hat{y_i^{(t)}}] \tag{2}$$

where y_i is the true head of the i-th token and $\hat{y_i^{(t)}}$ is the predicted head according to the partial dependency tree at time step t. However, they noted that when considering the cumulative loss, tokens which the parser gets initially will be summed as positive rewards over all future configurations. They noted that this leads the parser to finish as soon as possible and thus the parser performs mainly shifts at the beginning to maximize the reward. Thus to eliminate this greedy strategy, we decided to use a relative reward function. We modeled our reward function after the Unlabeled Attachement Score (UAS) of a partial parse which is similar to the Hamming Loss in that it is

$$u(y, \hat{y}) = \frac{1}{T} \sum_{i=1}^{T} 1[y_i = \hat{y}_i] \tag{3}$$

where y is the golden parse while $\hat{y}$ is the predicted parse. Then for the transition at time step t as we are transitioning from s_t to s_{t+1} with partial parses $\hat{y}^{(t)}$ and $\hat{y}^{(t+1)}$ respectively, the value of our reward function is

$$r_t = u(y, \hat{y}^{(t+1)}) - u(y, \hat{y}^{(t)}) \tag{4}$$

Note that we define $u(y, \hat{y}^{(0)}) = 0$ where $\hat{y}^{(0)}$ is the starting tree. Thus we see that the reward is based on the improvement of the parse tree from our action. Then we see that for $\gamma = 1$, $R_t = u(y, \hat{y})$ where $\hat{y}$ is the final dependency tree. Thus since the agent aims to maximize the expected accumulative reward, then in our environment, the agent will aim to maximize the final UAS of the parse.

Then to implement the environment, we had to write a **STEP** method which takes a current state s_t and an action a_t and then outputs the tuple (s_{t+1}, r_t, f_t) where f_t is a flag indicating whether the episode is done or not – i.e. whether the parser has terminated. Additionally, we needed to implement a **RESET** method which resets the configuration of the parser back to its initial state with the current sentence.

4.4 Actor-Critic Methods

For the RL algorithms which we used to maximize the value of R_t, we used an actor-critic architecture. This architecture was introduced by Sutton et al. [2000] and uses two neural networks. The actor network is the approximated policy function with parameters θ and produces an action distribution given the current state. The critic network can be viewed as the approximated value function with parameters ω and aims the estimate the optimal accumulated reward that can be gained from that state.

The architecture we used for the actor network was similar to the neural architecture proposed by Chen and Manning. The architecture for the critic network is a densely connected neural network. The critic evaluates the quality of the current policy by adapting the value function estimate.

From here, we have two actor-critic algorithms. The first is the Advantage Actor-Critic algorithm (A2C) that provides a balance between bootstrapping using the value function and using the full reward –i.e. the quality of the final parse. The definition of the advantage value is

$$A_\omega(s_t, a_t) = r(s_t, a_t) + V_\omega(s_{t+1}) - V_\omega(s_t) \tag{5}$$

as it indicates how much better the agent's actions are than expected. Then the actor and critic network losses are defined as

$$L(\theta) = \frac{1}{T}\sum_{t=0}^{T-1} A_\omega(s_t, a_t) \log \pi_\theta(a_t \mid s_t) \text{ and } L(\omega) = \frac{1}{T}\sum_{t=0}^{T-1} A_\omega(s_t, a_t)^2 \tag{6}$$

The other actor-critic method we used is the Proximal Policy Optimization method shown in Schulman et al. [2017]. As this was an actor-critic method that was developed after the Shen et al. paper was released, we decided that testing this new method as the RL architecture could be useful. It also has an actor and critic network with the same architecture as the A2C method but uses different losses. The actor loss is calculated with a KL penalty to ensure that the policy does not change too quickly,

$$L(\theta) = r_t A_\omega(s_t, a_t) - \beta \text{KL}[\pi_{\theta_{\text{old}}}, \pi_\theta] \tag{7}$$

where β is an adaptive parameter. Then the value loss is calculated from the original mean-squared error loss used by the A2C method and an entropy bonus calculated based on the state. Both the A2C and PPO algorithms were implemented using stable-baselines3 which is an open source RL library. All hyperparameters of these models that are not specified in the experiment section are the set to the default values that stable-baselines3 uses. We mainly needed to write the training scripts and write the script for pretraining the policy network.

5 Experiments

5.1 Data

We conducted our experiments on the English Penn Treebank (PTB) dataset created by Marcus [1993] with standard splits which has the input as the sentence and the output as the golden dependency parse tree of the sentence. The training, development and testing set contains 39832, 1700 and 2416 sentences respectively. Now the sentences themselves were loaded in one at a time into the environment as the sentence that the agent is attempting to parse. Then after the agent is done the episode resets to a new sentence.

The main evaluation metric which we used for the parser was the Unlabeled Attachement Score (UAS) which measures the percentage of tokens with the correct head. We see that to calculate the UAS of a particular parse using our parser environment, we simply need to sum all of the rewards during the episode and the result is the final UAS of the parse.

5.2 Experimental Details

First for both the A2C and PPO models, the actor network was identical to that of Chen and Manning – it had a single hidden layer of size 200. The critic had 4 hidden layers with $128, 128, 64$ and 32 as the hiddne sizes with RELU activation functions between the layers. A last hidden layer is used to predict a scalar as the value for the current state. For both networks the learning rates were $1e-5$ and the discount factor γ was set to 0.99.

Model	UAS
Neural Dependency Parser	88.8825 ± 13.5610
Advantage Actor-Critic	31.1950 ± 10.2854
Advantage Actor-Critic *	$\mathbf{89.1770 \pm 13.5316}$
Proximal Policy Optimization *	89.1217 ± 13.3844

Table 1: Comparison of UAS values between the parsers on the test set. The * indicates that the model used supervised pretraining to initialize the policy network.

The optimizer we used was the Adam optimizer. For each sentence, we have 5 environments running simultaneously on that episode and we run updates using batches with 50 timestep rollouts for each environment according to the policy.

We also initialized the policy network for both A2C and PPO with the same parameters of the model trained from the supervised learning method as a pretrained actor. To do this, we were able to use the actor network to sample many different transitions from the environment and then use these transitions to train an initial A2C and PPO policy network. Additionally, we tried training the A2C model with no supervised pretraining to determine whether the RL model was able to determine an optimal strategy by itself. The A2C method – both with and without supervised pretraining – was trained for 5 million time steps while the PPO method was trained over 2 million time steps.

One other issue that we came across was what to do when the model predicts an illegal action. To eliminate that, we limited the set of actions when the policy produces the most likely actions. This prevents us from having to impose a significant negative reward for illegal actions and thus improves the training.

Lastly, during testing, we ensured that the policies were deterministic in that they choose the most likely action each time. During training, we want the policy to be stochastic as this allows for increased exploration of the environment. However, during testing we want the policy to be deterministic as by that point we are assuming that there exists one unique correct action at each step.

5.3 Results

The performance of both the A2C and the PPO parsers did improve upon the supervised parser from Chen and Manning when using supervised pretraining. The UAS results are shown in Table 1. We see that when using supervised pretraining, the A2C model improves upon the supervised parser by around 0.4 percentage points while the PPO model improves upon the supervised parser by around 0.3 percentage points. These results are at a similar level to the results shown in the Shen et al. paper as the improvement that their RL methods posts over the supervised method also improves by around 0.4 percentage points.

Next, we see that Figures 1 and 2 show the test accuracy of the model after some number of training steps for the A2C models and PPO model respectively. The test accuracy was evaluated every 1000 time steps. Again recall that the accuracy of the model on the set is the same as its average episode reward. We see that the A2C-based parser even without a supervised pretrained model performs significantly better than a policy which selects legal moves at random as that policy achieves an average UAS of 12.9021 ± 11.1732. Thus the A2C algorithm is able to learn a policy with no starting information that performs significantly better than random.

Overall, the performance of the models was around in line with the results we would expect from the results of Shen et al. and thus we have succeeded in replicating the paper. This indicates that the change in the reward function that we adopted did not worsen the training of the model although we were not able to see significant improvement over the results of Shen et al. by simply altering the reward function. Moreover, we were able to test the more recent PPO algorithm and we determined that it also performs slightly better than the supervised model and in line with the performance of the A2C-based parser.

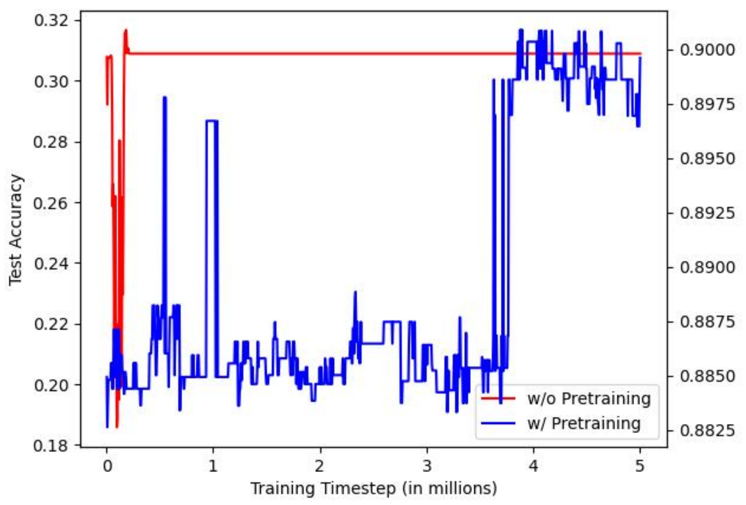

Figure 1: The accuracy on the test set during training for the A2C model w/ and w/o supervised pretraining. The scale on the left-hand side is for the model w/o supervised pretraining and the scale on the right-hand side is for the model w/ pretraining.

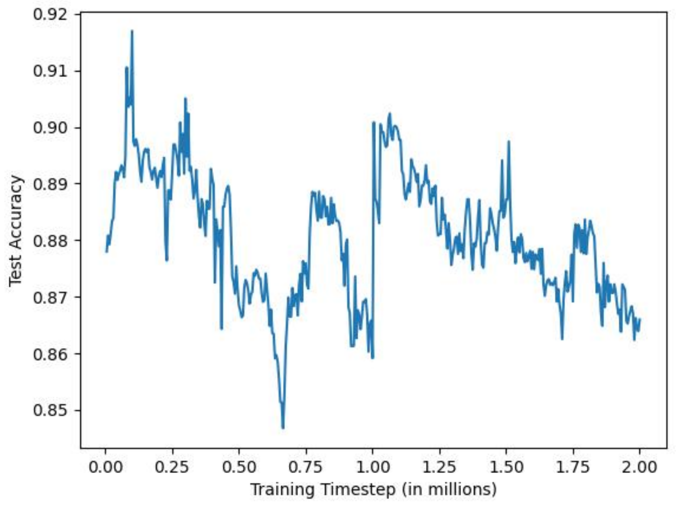

Figure 2: The accuracy on the test set during training for the PPO model

6 Analysis

While both the A2C and PPO algorithms perform slightly better than the supervised model on the test data, we can learn much about their characteristics by examining the loss curves. Figure 3 shows the loss curves for both the A2C models with and without pretraining. Note that the loss of the policy network fluctuates significantly more for the model without pretraining versus the model with pretraining. This makes sense intuitively as we would expect the model that already has a good policy to not have as high of a loss versus the model with a significantly worse initial policy. Moreover, we see that both policy losses converge to be within the interval $(-0.005, 0.005)$ after 3 million iteration indicating that at this point the model has settled on a policy and the magnitude of the updates decrease.

Now if we look at the value loss we see that for both models, the value loss trends downwards. However, note that the value loss is initially larger for the pretrained model versus the non-pretrained model. The reason this happens is because the pretrained model starts already trained on a particular policy and thus the accumulated rewards of the policy are quite high whereas the value network still has not been trained and is thus not predicting even relatively correct values for the states. Thus as the agent iterates through more episodes, the value loss declines as the model becomes more confident about the values of particular states that it has seen before. Especially considering that we run 5 environments in parallel, this means that for each sentence, it is repeated 5 times and assuming that the sequence of actions is roughly the same for each of the 5 iterations, then this means that the sequence of states that the value network trains on appear multiple times and thus the model learns those examples faster. Lastly, note that while the non-pretrained model arrives at a suboptimal policy, the value loss continues to decline as the probability of exploration success on this task is quite low. Thus once the agent finds a local optimum, it is likely to get stuck there.

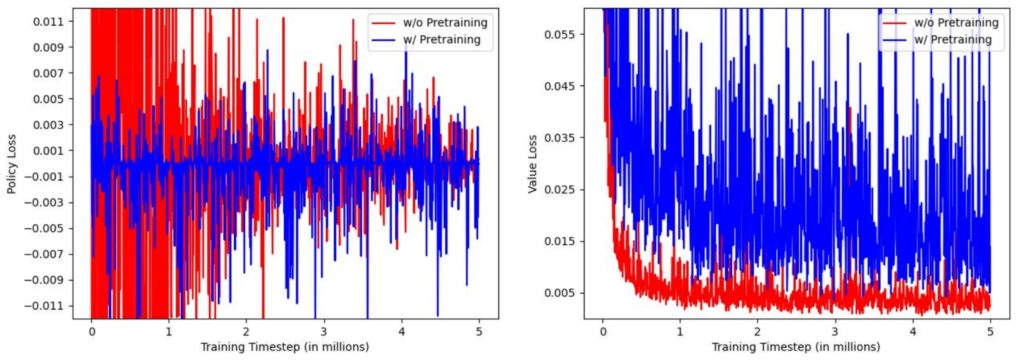

Figure 3: Losses for both the Policy and Value network of the A2C algorithm

We can examine a sample sentence under all three of the parsers – the neural dependency parser, the A2C parser and the PPO parser. For the sentence shown in Figure 4, we note that the A2C and PPO models achieve a UAS of 80 on sentence while the Neural Dependency Parser only achieves a UAS of 50. It is possible that the reason why this occured is due to the phenomenon that Shen et al. discussed. Namely, when the supervised model makes a mistake because it performs greedy decoding, the error propogates to the rest of the parse reducing the UAS further than the RL-based parsers as they use non-greedy decoding. Specifically, note that all three parsers made the same mistake of drawing an edge from 'point' to 'up' possibly because the training data has the phrase 'point up' one or multiple times. Moreover, note that this error is essentially just switching the direction of one of the arrows in the golden parse. Thus the RL-based parsers are able to recover and improve their UAS by drawing all the same arcs as in the golden parse and then attaching 'were' to 'point' instead of 'were' to 'up'. After the initial error, this is the optimal strategy to get a high UAS and we see that this is exactly the parse which the RL-based parsers produce. However, the supervised model performs greedy decoding and thus after it makes the error it is no longer in a domain which it is familiar with. Thus we see that the model makes numerous other errors ultimately culminating in labeling the wrong word as the root. Therefore, this example sentence highlights how the RL-based parser is able to recover from errors early in the parse while the supervised dependency parser struggles at the task exactly as Shen et al. predicted.

7 Conclusion

We were able to construct two RL-based dependency parsers that performed at least comparably if not better than a supervised Neural Dependency parser. Through this project, we were able to combine what we have learned about dependency parsing with some prior knowledge of reinforcement learning to create an RL-based dependency parser. We also showed that on many sentences, the supervised parser suffers from the error propogation issue whereas the RL models perform better and we highlighted the importance of initializing the parameters of the policy network as the parameters of supervised model as otherwise the state space is simply too large to explore. The primary limitations of our project is that we were unable to quantitatively test whether the supervised parser or the RL-based parser handles error propogation better. We were trying to find a way to systematically cause errors early in the parse that are still recoverable and then determining the final UAS which the RL-parser is able to achieve versus the supervised baseline. Future work testing the error propogation claims of Shen et al. specifically could be useful then.

Additionally, many of the possible extensions which Shen et al. discussed could be implemented on top of our framework. For example, using a custom model class, it is possible to replace the core policy network with a recurrent network that is able to take in all previous states of the environment and choose an action based off of that extra information. Lastly, while we believe that our reward function is a better candidate for the reward function over the negative Hamming loss, future work on reward shaping for the dependency parsing problem could yield interesting findings.

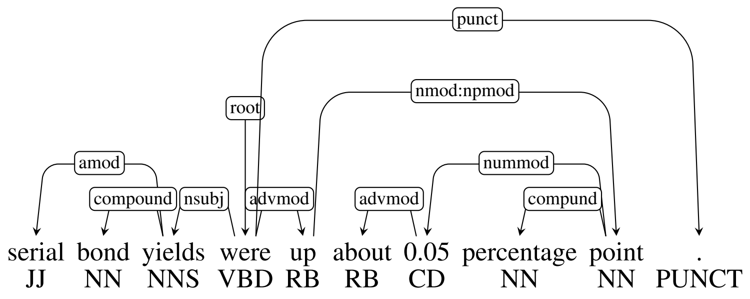
punct
nmod:npmod
root
amod
nummod
compound
nsubj
advmod
advmod
compund
serial bond yields were up about 0.05 percentage point .
JJ NN NNS VBD RB RB CD NN NN PUNCT

Figure 4: Gold parse of sentence

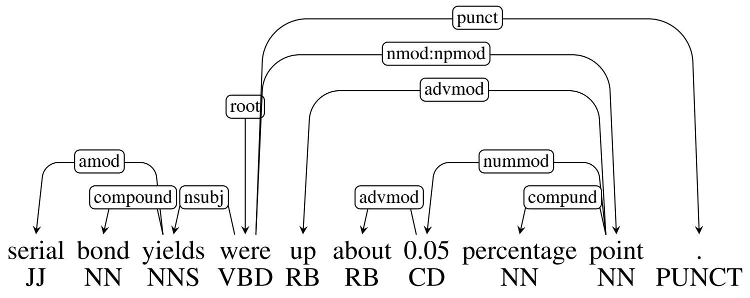
punct
nmod:npmod
advmod
root
amod
nummod
compound
nsubj
advmod
compund
serial bond yields were up about 0.05 percentage point .
JJ NN NNS VBD RB RB CD NN NN PUNCT

Figure 5: A2C and PPO parse of the sentence

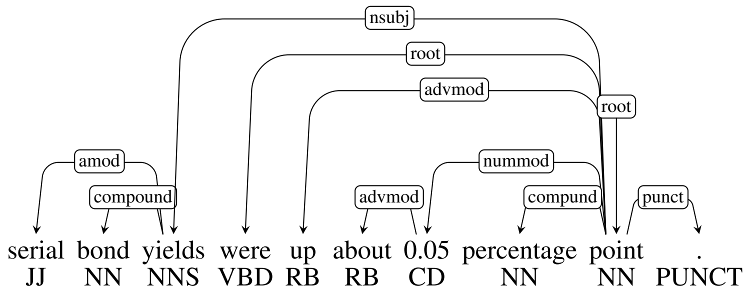
nsubj
root
advmod
root
amod
nummod
compound
advmod
compund
punct
serial bond yields were up about 0.05 percentage point .
JJ NN NNS VBD RB RB CD NN NN PUNCT

Figure 6: Neural Dependency Model parse of sentence

References

[1] Chen, Danqi and Manning, Christopher. A Fast and Accurate Dependency Parser using Neural Networks. 2014 Conference on Empirical Methods in Natural Language Processing, 29 Oct. 2014.

[2] G. Neu and C. Szepesvari. Training Parsers by Inverse Reinforcement Learning. *Machine Learning*, 77(2):303, Apr. 2009.

[3] Joakim Nivre. Incrementality in deterministic dependency parsing. *Proceedings of the Workshop on Incremental Parsing: Bringing Engineering and Cognition Together*, pages 50–57. Association for Computational Linguistics, 2004.

[4] Joakim Nivre and Daniel Fernández-González. Arc-eager parsing with the tree constraint. *Computational linguistics*, 40(2):259–267, 2014.

[5] Joakim Nivre and Mario Scholz. Deterministic Dependency Parsing of English Text . *In Proceedings of the 20th international conference on Computational Linguistics*, page 64. Association for Computational Linguistics, 2004.

[6] Jiang, Yong,Han, Wenjuan,Tu, Kewei. Unsupervised neural dependency parsing[C]:Association for Computational Linguistics (ACL),2016:763-771

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

[8] Marcus, Mitchell P, et al. Building a large annotated corpus of English: the penn treebank. Computational Linguistics, 01 Jun. 1993.

[9] Minh Le and Antske Fokkens. Tackling error propagation through reinforcement learning: A case of greedy dependency parsing. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 1, Long Papers*, volume 1, pages 677–687, 2017.

[10] Mnih, Volodymyr, et al. Playing Atari with Deep Reinforcement Learning DeepMind, 2013.

[11] Schulman, John, et al. Proximal Policy Optimization Algorithms OpenAI, 28 Aug. 2017.

[12] Shen, Ying, et al. Dependency Parsing With Deep Reinforcement Learning 29th Conference on Neural Information Processing Systems, 2016.

[13] Stable Baselines3 Documentation 2020.

[14] Sutton, Richard S, et al. Policy Gradient Methods for Reinforcement Learning with Function Approximation AT&T Labs, 2000.

[15] Timothy Dozat and Christopher D Manning. Deep biaffine attention for neural dependency parsing. *arXiv preprint arXiv:1611.01734*, 2016.