Implementing Syntactic Parsing with Beam Search
Post project paper for the course TDDE09 at Linköping University

Fabian Bergström (fabbe820)

1 Description

This section describes the project.

1.1 Summary and objectives

The project involved enhancing a simple syntactic parser by implementing a technique known as beam search with error states, replacing the greedy search algorithm currently used in the system, called baseline. The baseline consists of a tagger-parser system which in this context has the objective to take a language treebank as input, tag each word with its corresponding word label (verb, objective for example) and parse the words with these labels, their context and a lot of training in mind to be able to predict the sequence of words in a sentence. Our results were then compared to the results obtained by (Vaswani and Sagae, 2016), which our error state implementation is based on. Error states were implemented in the training phase to help teach the parser to recognize incorrect states, hopefully leading to a higher accuracy when predicting sentences. The results however only showed an improved accuracy on the Swedish treebank, in contrast to the results in (Vaswani and Sagae, 2016). This led to experiments with beam widths, seeds and number of features, discussed further in 1.3.3.

1.2 Why the project was chosen

The project seemed interesting since the lab focusing on dependency parsing (2.1.2) was exciting and proved its strengths when it comes to sentence prediction. However it did not go into detail of how the results could be improved. This made the baseline project description interesting. When it came to deciding which proposed algorithm to use, beam search was chosen since it was recognized the most at first sight and also recommended during an in-class assignment. Therefore beam search seemed like the most appropriate. The addition of using error states during training also seemed like an interesting experiment since it mostly sounded great due to it generating examples with high loss in unwanted states, much like q-learning which many in the group had prior knowledge of, but it also sounded like it could lead to a lot of overfitting, causing the tagger-parsers ability to decrease.

1.3 Implementation and results

This section describes the implementation of beam search and error states, and shows its results in comparison to the baseline system.

1.3.1 Beam search

Beam search is an expansion of greedy-first search, where instead of just choosing the local high scoring branch at each level in a tree, the best X number of states are chosen at each tree level instead, where X is a specified beam size.

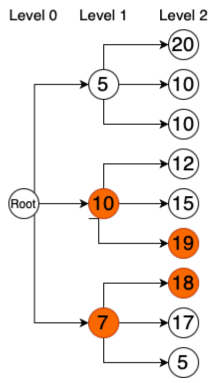

Figure 1: Beam search example with beam size 2

The Beam search implementation is based on the pseudo code given by (Zhang and Clark, 2008) where predicted next words given the current configuration and its scores are generated. The score used to evaluate each move at each configuration was the logarithm of the softmax of the list of scores. These moves, also called transitions between configurations includes shift, left-arc and right-arc, where shift moves the top word from the buffer to the top of the stack, left-arc adds a dependency from the topmost word on the stack to the second-topmost word, and removes the second-topmost word and right-arc adds a dependency from the second-topmost word on the stack to the topmost word, and removes the stack top word. The list of scores includes all possible moves at that state, leading to a list of normalized scores for each possible move, and then picking the best one.

1.3.2 Error states

The error state implementation meant that during training, when the ArcStandard algorithm generated training examples consisting of states and the best possible moves to do in those states according to the gold standard, examples of what to not do in those states were also generated . In more detail, an error move label was created and then connected to all non gold standard (non optimal) moves of that current state, leading to not only giving the parser directions of what to do in each state, but also giving examples of what to not do. The error state implementation was inspired by the description by (Vaswani and Sagae, 2016).

1.3.3 Results

Experiments on 5 different language-datasets with varying sizes, including Swedish, ancient Hebrew, Icelandic and 2 English were made and their Unlabelled Attachment Score (UAS) were measured. UAS is the percentage of accurate word dependency predictions. Beam search improved performance by up to 2 percent regardless of dataset size and language. The addition of error states however, only led to improvement when evaluating the Swedish treebank, where it yielded an improvement of 2 percent against the beam search version, showing a total of 4 percent gained against the baseline, shown in the figure below.

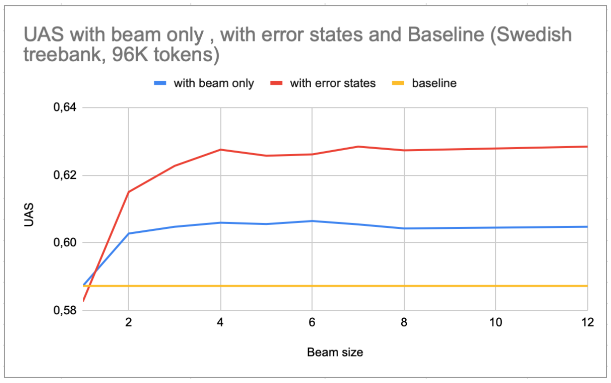

Figure 2: UAS performance on swedish treebank

In regards of all other languages and their datasets error states generally led to 2 percent loss in UAS, placing the performance at the same level as the baseline, which caused the group to look more into how these error states were weighted during training, how they might cause overfitting and how much of an impact randomness had. After finding an optimal beam size of 6 during testing, experiments included using different seeds to find an UAS variance, illustrated below. It showed a huge variance of up to 3 percent, even causing the error states addition to lose on the Swedish treebank with seed 25.

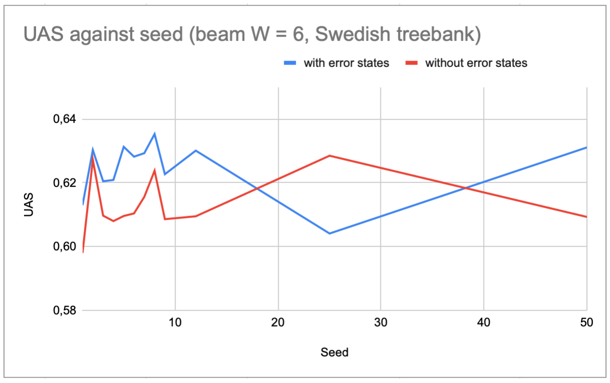

Figure 3: Seed variance on swedish treebank

Since there are up to 3 possible moves (shift, left arc, right arc) in each state, where only one is the optimal move (gold standard move), the possibility that generating error states for each incorrect move led to overfitting or teaching the parser what not to do more than what to do was brought up. This theory was tested by introducing different probabilities of creating an error state in each state for each move, showing an improvement overall, often beating the baseline and even beating the beam-search-only implementation for a specific value which it never did before when using 100 percent error states on all treebanks except the swedish one.

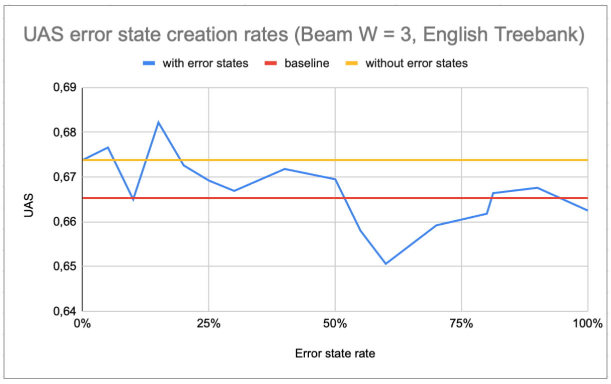

Figure 4: Error state creation rate experiment, tested on smaller english treebank

Lastly, experiments with the number of features used in each prediction were made, the number of features used in other experiments were 3. The number of features used is the number of words and tags the parser is looking at, or is dependent on when predicting the next word. The results showed a huge improvement in adding a few features and a bigger loss when using less, shown below.

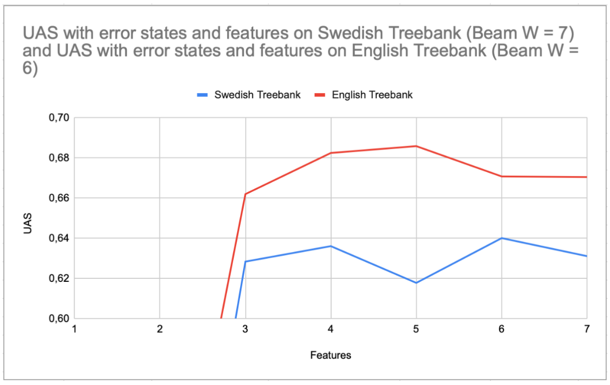

Figure 5: number of features experiment, tested on swedish and smaller english treebank

Overall, the introduction of error states in the system caused a loss of UAS, in contrast to the results obtained in (Vaswani and Sagae, 2016).

2 Examination

This section describes relevant course concepts and how they were used in the project.

2.1 Relevant course concepts

This section describes relevant course concepts in depth.

2.1.1 Sequence labeling

The baseline of the project is as said before a simple tagger-parser, to understand the tagger part, sequence labeling had to be a clear concept to everyone in the group. As described during the course, a sentence consists of a sequence of words, where each word in the sentence needs a part-of-speech tag.

A part-of-speech tag is for example a verb or an adjective, this tag is necessary since a sentence built by the same combination of words can have multiple different meanings depending on order and emphasis, making it very hard for a computer to predict the next word in a sentence. This tagging system makes it possible to showcase the relationship between words to make it possible to predict sentences with higher accuracy.

The baseline uses a fixed-window model to limit the decision of choosing a tag for a word to only take a number of previous and following tags inside the specified context window into account when making a prediction.

2.1.2 Transition based dependency parsing

The sequence labels described in the previous section creates dependencies between words in a sentence, where each sentence has a root word. Each word has its own position in the sentence, its head-word, meaning the word that it is dependent on, and the actual dependency relation, for example objective or adverb.

In the context of the project these sentences are represented by dependency trees built by the arc standard algorithm, and later on translated into a sequence of transitions or moves needed in each tree state to move to the best next state, this translation is created by an oracle algorithm. The arc standard algorithm uses a classifier that predicts the transitions needed of the parser to reach the next configuration and repeats the classifications until a full dependency tree is built. A configuration contains three parts, a stack, a buffer and partial dependency tree, this to represent the current state at each position in the dependency tree.

Understanding the different stack actions behind making moves or transitions were essential to understand the reasoning behind adding error states, how it affects training and how changing the number of features affect dependencies.

2.1.3 Parsing

To understand where and why to implement beam search knowledge of the parsing phase was essential. The objective of the parser is to take a dependency tree as input, parse its branches depending on each state's score, based on a scoring function, and return the highest scoring results which in this project's context is the highest scoring next words. The initial baseline parser uses a greedy-first algorithm which in many cases does not lead to the optimal solution since a local high scoring branch might lead to a much lower scoring branch in the end. The purpose of beam search is to explore more high scoring branches in each tree level leading to a generally better score.

2.1.4 Training

The implementation of error states meant that knowledge of the training process was necessary to understand where and why to implement them. Training loops and loss functions have been widely used in previous labs during the course, in the project cross entropy loss between a state configuration and its next move/transition was used to evaluate how good that move is to make at that state. This was also important to understand since experimenting with the number of features in each context window affects the cross entropy loss in this step since it gives more information about the configuration and its predicted best next transitions. Lastly, understanding the concept behind overfitting was also necessary to realize that introducing error states might cause the model to overweight error states and become less efficient at generalizing data not seen before, leading to experiments with different error state creation rates.

2.1.5 Enhanced knowledge

The project gave me a lot of knowledge not part of the course content, for example beam search, error states and how to further evaluate performance described both in the description and examination sections. During the course UAS, F1 score and other evaluation metrics were used, but learning about the effect of hyperparameters on the evaluation gave further knowledge. The project also enhanced the understanding of multiple concepts brought up during the course, the labs introduced the overall idea behind the tagger-parser structure, the project however was where the concepts had to be understood deeper to successfully implement the algorithms and evaluate performance, combining the lab and lecture contents fully.

3 Articulation of learning

This section describes what I learned during the project, how I learned it and why the learning matters.

3.1 What I learned

The project taught me that implementing specific algorithms may increase performance in some cases while decreasing it in others. I learned this through studying the implementation described in (Vaswani and Sagae, 2016), implementing it, and comparing the achieved results. I thought that the addition of error states would increase performance regardless of the used language treebank, especially since the overall methods used were both transition based dependency parsing. However, as the results showed, error states seemed to generate overfitting or over weighting error states in our model. I also learned how to better pinpoint potential model performance factors and construct experiments to analyze the effect of hyperparameters for example, leading to knowledge of the big effect randomness and overall language structures has on language based dependency parsing performance.

3.2 How I learned it

I learned this by exploring literature related to applying beam search on transition based dependency parsing, going in depth with lecture slides and testing. The beam search and error state implementations were inspired by the description in (Vaswani and Sagae, 2016) and pseudo code in (Zhang and Clark, 2008). Many conclusions were drawn from testing multiple language treebanks of varying sizes and structures, including several Germanic languages and Hebrew.

3.3 Why the learning matters

The learning matters since it enables me to more effectively find and understand relevant literature within the field of natural language processing, including implementing proposed algorithms and effectively evaluating their effects and performance. My findings contribute to research in the area by showing a possible downside of introducing error states without using a "error state creation rate", that beam search is a reliable algorithm for parsing and that there is a sweet spot when it comes to the number of features used during parsing, more is not always better.

References

Ashish Vaswani and Kenji Sagae. 2016. Efficient structured inference for transition-based parsing with neural networks and error states. *Transactions of the Association for Computational Linguistics*, 4:183–196.

Yue Zhang and Stephen Clark. 2008. A tale of two parsers: Investigating and combining graph-based and transition-based dependency parsing. In *Proceedings of the 2008 Conference on Empirical Methods in Natural Language Processing*, pages 562–571.