HW3: Dependency Parsing

Yanze Wang
Department of Computer Science
University of Southern California
Los Angeles, CA 90089 USA
yanzewan@usc.edu

Abstract

Dependency parsing is the task of extracting a dependency parse of a sentence that represents its grammatical structure and defines the relationships between words. In this paper, we build an arc-standard shift-reduce neural dependency parser based on the excellent application by Danqi Chen from 2014. We evaluate the performance of our dependency parser using unlabeled and labeled attachment scores on local Dev dataset and hidden test dataset. It performs well on both dataset with a fast sentence parse speed.

1 Introduction

With the explosion of world-wide language data, text analysis becomes a pervasive problem. Since human language is so rich and complex, It is almost impossible to accurately explain all possible expressions of the text we are searching for, without a more intelligent understanding of the structure and relationships involved in word combinations. Dependency parsing provides this information which can be used to find the relationships between words. An excellent application of neural networks to dependency parsing is the work of Danqi Chen in paper "A Fast and Accurate Dependency Parser using Neural Networks". In this paper, we build a neural dependency parser using arc-standard algorithm and neural network model in PyTorch. Machine learning based classification neural networks can be trained on converted data from dependency trees information and learn to predict the relationships between words from the extracted features to form parse trees. We evaluate the dependency parser using unlabeled attachment scores (UAS) and labeled attachment scores (LAS). More detailed discussion about the experiment and results is also included.

The paper is organized as follows: section 2 explains the format of datasets provided for training and developing, section 3 introduces parsing methods and detailed structure of arc-standard algorithm, section 4 introduces the structure of the neural network language model used for training and predicting, section 5 reports the experimental settings and the performances of our dependency parser on Dev and Test datasets, section 6 contains the discussion about some important aspects of the dependency parser and section 7 is the conclusion.

2 Datasets

The datasets we use to train and develop are in CoNLL format. The dataset has one word per line, with an empty space to separate two sentences. Each line has 10 tab-separete fields. Some important attributes are token id, word, POS tag, head of the word and the dependency label. The training data contains 39832 sentences and dev data contains 1700 sentences. These data should be converted into features of the parser configuration paired with parser decisions which are used for further training.

3 Transition-based Dependency Parsing

In this task, we develop the **arc-standard** algorithm which is the algorithm for transition-based dependency parsing. It can only predict projective dependency trees (Nivre, 2004). Algorithms for non-projective trees also exist(Nivre, 2009). Te parser starts in the initial configuration which is a empty dependency tree, it then calls a classifier, which predicts the transition that the parser should update to the next configuration. This process is repeated until the parser reaches a terminal configuration.

3.1 Configuration

A parser configuration c consists of three parts: a stack s which contains those words in the sentence that are currently being processed, a buffer b which contains the words that are not processed yet and a set of dependency arcs A. Initially, the buffer contains all words and the stack is empty. When the buffer is empty and the stack contains only the root node, the configuration is terminal.

3.2 Transition

The arc-standard system defines three types of transitions:

- SHIFT: removes the most front word from the buffer and pushes it to the top of the stack.
- LEFT-ARC-label: creates a dependency from the topmost word to the second-topmost word on the stack, and pops the second-topmost word from the stack.
- RIGHT-ARC-label: creates a dependency from the second-topmost word to the topmost word on the stack, and pops the topmost word from the stack

With these three transitions combined with labels, this becomes a classification problem with *2N+1* transition choices where *N* is the number of arc labels.

We need to convert from a dependency tree to the sequence of configuration parse steps which is used for further classification model training. Given a configuration and a correct dependency tree, we choose the correct operation based on the following principles:

- If stack[0] is the parent of stack[1], do LEFT-ARC-label and save as (left-arc, label of stack[1]) choice.
- If stack[1] is the parent of stack[0] and no dependents of stack[0] are still in the buffer, do RIGHT-ARC-label and save as (right-arc, label of stack[0]) choice.
- Otherwise, do SHIFT and save as (shift) choice

We always need to check whether the transition is valid. For SHIFT transition, it is valid if the buffer contains at least one word. For RIGHT-ARC and LEFT-ARC transitions, they are valid if the stack contains at least two words.

3.3 Feature Extraction

For each configuration, we extract these features to form three feature sets S^w, S^p, S^l for the classifier:

- S^w:the first three words on the stack and the buffer, the words of the first and second leftmost and rightmost children of the first two words on the stack, the words of the leftmost child of the leftmost child and rightmost child of rightmost child of the first two words on the stack. (18 features)
- S^p:the corresponding POS tags for S^w. (18 features)
- S^l:the corresponding arc labels for S^w except the first three words on the stack and the buffer. (12 features)

Instead of using pre-trained word embeddings, we implement the all embeddings from scratch. The algorithm extracts features from the current configuration and tries to predict next transition using the extracted features. The algorithm is greedy which means a bad early decision may lead to later possible problems.

4 Neural Network

Given the features extracted from generated configuration, we train a single feedforward model to predict transitions and arc labels. The whole feedforward model contains the following part: Given input layer $[x^w, x^p, x^l]$ from feature sets $[S^w, S^p, S^l]$, we can compute as:

$$h = W_1^w \cdot x^w + W_1^p \cdot x^p + W_1^l \cdot x^l + b_1 \quad (4.1)$$

$$H = Tanh(h) \quad (4.2)$$

$$z = W_2 \cdot H \quad (4.3)$$

$$o = Softmax(z) \quad (4.4)$$

where W_1 and W_2 are the weight matrix, B_1 is the bias vector. There are also two functions $Tanh$ and $Softmax$ which is defined as :

$$Tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \quad (4.5)$$

and

$$Softmax(z_i) = \frac{\exp z_i}{\sum_{j=1}^{n} \exp z_j} 1 \leq i \leq n \quad (4.6)$$

for any vector z of dimensionality n.

We then use **cross-entropy** loss with a l_2-regularization to compute the loss between the predicted output and the ground truth. An AdaGrad

Data	#Words	#POS	#Labels	#Sentences	Projective percent
Train	44389	45	39	39832	99.699
Dev	6840	44	39	1700	99.705

Table 1: Data Statistics.

optimizer (Duchi et al.,2011) is used to update weights and bias for optimization and a dropout with 0.5 rate is used for regularization and preventing the co-adaptation of neurons (Hinton et al., 2012). In practice, all these functions are implemented using PyTorch.

5 Evaluation

5.1 Experimental Set-up

As mentioned above, the training and developing data is given in the format of CoNLL. We firstly convert training and dev data into a (feature vector, label) format which is suitable for further training. Since Shift-reduce parser only admit projective trees, non-projective sentences are detected and removed from the training data and developing data. We then build the architecture of the parser and train the feed-forward model which is implemented in PyTorch. The trained model is then used to predict transitions and dependency labels which can be converted into dependency tree information in CoNLL format.

For evaluation and debugging, we use the provided Stanford parser jar to get UAS and LAS to score the output on dev and test datasets.

We execute the experiments on Apple M2 CPU. We tried to use the GPUs of Nvidia Tesla P100 GPU(16GB memory) on Google Cloud to execute the experiments. We managed to call the GPUs but the utilization was so low that it ran even slower than on CPU, We will test the reason of this phenomenon in further study.

Different values of embedding size (d), hidden layer size (h), number of epochs (e) are tested and reported combined with time consuming, unlabeled attachment scores (UAS) and labeled attachment scores (LAS).

5.2 Results

The following hyper-parameters are used in experiments: embedding size d = 10, hidden layer size h = 50, regularization parameter $\lambda = 10^{-8}$,initial learning rate of Adagrad $\alpha = 0.01$, number of training epoch e = 5.

Because of the training speed, we didn't use the embedding size and hidden layer size in Danqi's paper which takes about 3 hours to train one epoch on CPU to do the experiments. But we test the performance of different parameters to show a tendency. We add <null> and <unk> tokens to form embeddings to represent empty values and unknown words.

Table 1 gives statistics of the training and developing datasets. In particular, 120 sentences are non-projective in training dataset and 5 sentences are non-projective in developing dataset. Over 99 percent of the trees are projective in both datasets.

In Table 2, we reported unlabeled attachment scores (UAS) and labeled attachment scores (LAS) computed by same trained model on dev dataset locally and on test dataset on Vocareum. It shows similar scores on both datasets.

In Table 3, we reported UAS,LAS and parsing speed tested on Dev dataset using different hyper-parameters to examine the influence of embedding size, hidden layer size and number of training epoch. Base parameters are introduced above. We can find a overall positive tendency of embedding size, hidden layer size and number of training epochs influencing the LAS and UAS.

6 Discussion

Some of the discussion and analysis is covered in the previous section. I will discuss some more here.

For learning rates, we need to find a suitable rate which is Neither too high nor too low. A too high learning rate may make the learning process jump over the minimal value of the loss function and a too low learning rate may take too long to converge or ends into a undesirable local minimal value. We examined learning rate 0.01, 0.05 and 0.1 here, we find that the training loss becomes quite high when have a higher learning rate which shows that 0.01 is a suitable learning rate.

For overfitting, we use dropout (Hinton et al., 2012) in the model with a probability 0.5 which randomly zeroes some of the elements of the input tensor. I think this function reduces the probability of overfitting and thus I didn't discover a overfitting

dataset	UAS	LAS
Dev	0.784	0.755
Test	0.785	0.758

Table 2: Score comparation between local Dev dataset and Vocareum hidden Test dataset

parameters	UAS	LAS	Speed
$d = 20$	0.791	0.772	42
$d = 50$	0.786	0.758	28
$h = 100$	0.818	0.781	46
$h = 200$	0.823	0.788	39
$e = 2$	0.754	0.72	51
$e = 10$	0.792	0.757	51
Base	0.784	0.755	51

Table 3: LAS and UAS score on Dev dataset for different d, h, e. Speed is the number of sentences trained in one second.

on training data.

The embedding method of words, POS and labels is random, and the embedding for NULL and Unknown words are also random. I think some pre-trained embedding models or other initialization method may perform better in this task. Model training took me too long to do more experiments. Further studies can be done using GPUs and change some components to test the performance of dependency parser.

During feature extraction, I find that at the beginning part of whole walkthrough, the children of words are often ignored. This is not good for extraction since shift transition are always predicted before left and right transitions. Arc-eager algorithm may be a good choice to deal with this problem.

7 Conclusion

As discussed above, We need to find a trade-off between the training time consumption and the evaluation scores when choosing hyper-parameters. Overall, the experiments show that the dependency parser performs well on dev and test datasets.

References

Chris Manning Danqi Chen. A fast and accurate dependency parser using neural networks.

John Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(61):2121–2159.

Jacob Eisenstein. 2018. Natural language processing. *Jacob Eisenstein.*

Geoffrey E. Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2012. Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv*, abs/1207.0580.

Jonathan May. 2022. Dependency syntax.

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

Joakim Nivre. 2009. Non-projective dependency parsing in expected linear time. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, pages 351–359, Suntec, Singapore. Association for Computational Linguistics.