11-411/11-611 Handout Template

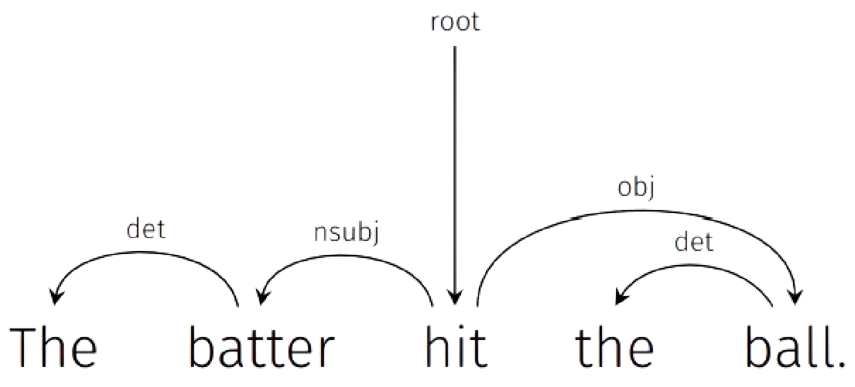
NLP Teaching Staff

Due: April 20th, 2023

1 Introduction

Dependency Grammar represents the relations between words

- Nouns can be subjects or objects of verbs
- Adjectives can be modifiers of nouns
- Adverbs can be modifiers of verbs, adjectives, and other adverbs



Dependency Trees are representations used for the syntactic analysis of sentences [2].To build a dependency tree, for each word we decide which other word it is a dependent of and what is the relationship they share.

A treebank is a corpus of sentences where each sentence has been parsed by humans or parsed with a parser and corrected by humans. For this homework we will be using a dependency treebank from Universal Dependency Treebanks.

In this homework, you are going to train a dependency parsing model based on the "arc-standard" system [3]. In the arc-standard system, each parse state is a configuration $C = (\sigma, \beta, \alpha)$ in which σ is a stack of processed words, β is an ordered list of unprocessed words and α is a set of already recovered dependency relations of the form $h \xrightarrow{l} d$ (d^{th} word is headed by h^{th} word with dependency label l).

Given this, the initial configuration can be represented as

$$C^0 = (\sigma = [root_0], \beta = [w_1, w_2, w_3 ... w_n], \alpha = [])$$

where $[root_0]$ is the dummy root node of the sentence and $[w_1, w_2, w_3 ... w_n]$ are the words in the sentence.

There are three main actions that change the state of a configuration in the algorithm: shift, left-arc and right-arc. For every specific dependency relation (e.g. subject or object), the left-arc and right-arc actions become fine-grained (e.g. RIGHT-ARC:nsubj, RIGHT-ARC:pobj, etc.).

Action	σ	β	$h \xrightarrow{l} d$
Shift	[root$_0$]	[I$_1$,live$_2$, in$_3$, New$_4$, York$_5$, city$_6$, .$_7$]	
Shift	[root$_0$, I$_1$]	[live$_2$, in$_3$, New$_4$, York$_5$, city$_6$, .$_7$]	
Left-Arc^{nsubj}	[root$_0$, I$_1$, live$_2$]	[in$_3$, New$_4$, York$_5$, city$_6$, .$_7$]	$2 \xrightarrow{nsubj} 1$
Shift	[root$_0$, live$_2$]	[in$_3$, New$_4$, York$_5$, city$_6$, .$_7$]	
Shift	[root$_0$, live$_2$, in$_3$]	[New$_4$, York$_5$, city$_6$, .$_7$]	
Shift	[root$_0$, live$_2$, in$_3$, New$_4$]	[York$_5$, city$_6$, .$_7$]	
Shift	[root$_0$, live$_2$, in$_3$, New$_4$, York$_5$]	[city$_6$, .$_7$]	
Left-Arc^{nn}	[root$_0$, live$_2$, in$_3$, New$_4$, York$_5$, city$_6$]	[.$_7$]	$6 \xrightarrow{nn} 5$
Left-Arc^{nn}	[root$_0$, live$_2$, in$_3$, New$_4$, city$_6$]	[.$_7$]	$6 \xrightarrow{nn} 4$
Right-Arc^{pobj}	[root$_0$, live$_2$, in$_3$, city$_6$]	[.$_7$]	$3 \xrightarrow{pobj} 6$
Right-Arc^{prep}	[root$_0$, live$_2$, in$_3$]	[.$_7$]	$2 \xrightarrow{prep} 3$
Shift	[root$_0$, live$_2$]	[.$_7$]	
Right-Arc^{punct}	[root$_0$, live$_2$, .$_7$]	[]	$2 \xrightarrow{punct} 7$
Right-Arc^{root}	[root$_0$, live$_2$]	[]	$0 \xrightarrow{root} 2$
Terminal	[root$_0$]	[]	

The above table shows a sample action sequence with the arc-standard actions.
You may learn more about how a dependency parser makes parsing decisions by referring to the lecture slides. For this homework, the parsing actions have already been provided to you and we will build a classifier to predict the next actions given the configurations.

2 Learning Objectives

- Build a Dependency Parse using Feed Forward Neural Networks
- Finalize and work with dependency tree representations

3 Task 1: Defining the Classifier

We have provided preprocessing code which converts the dependency trees from the GUM dataset into training data that consists of 3 kinds of features:

- Word features: 20 kinds of word features
- POS features: 20 kinds of POS features
- Dependency label features: 12 kinds of dependency features

This gives us a total of 52 input features. These features describe the configuration of the parser at a given timestep. If you are interested in what particular features are being used, you might find ”A Fast and Accurate Dependency Parser using Neural Networks” by Danqi Chen and Christopher D. Manning from Stanford interesting. Since we are trying to predict the next action, we have that be the output.

You will be defining a Classifier class that inherits from the nn.Module class from PyTorch. This is a template for the model you will be initializing and then training. This is done so that the hyperparameters may be defined with ease, making our code modular and robust.

3.1 Embedding Layer

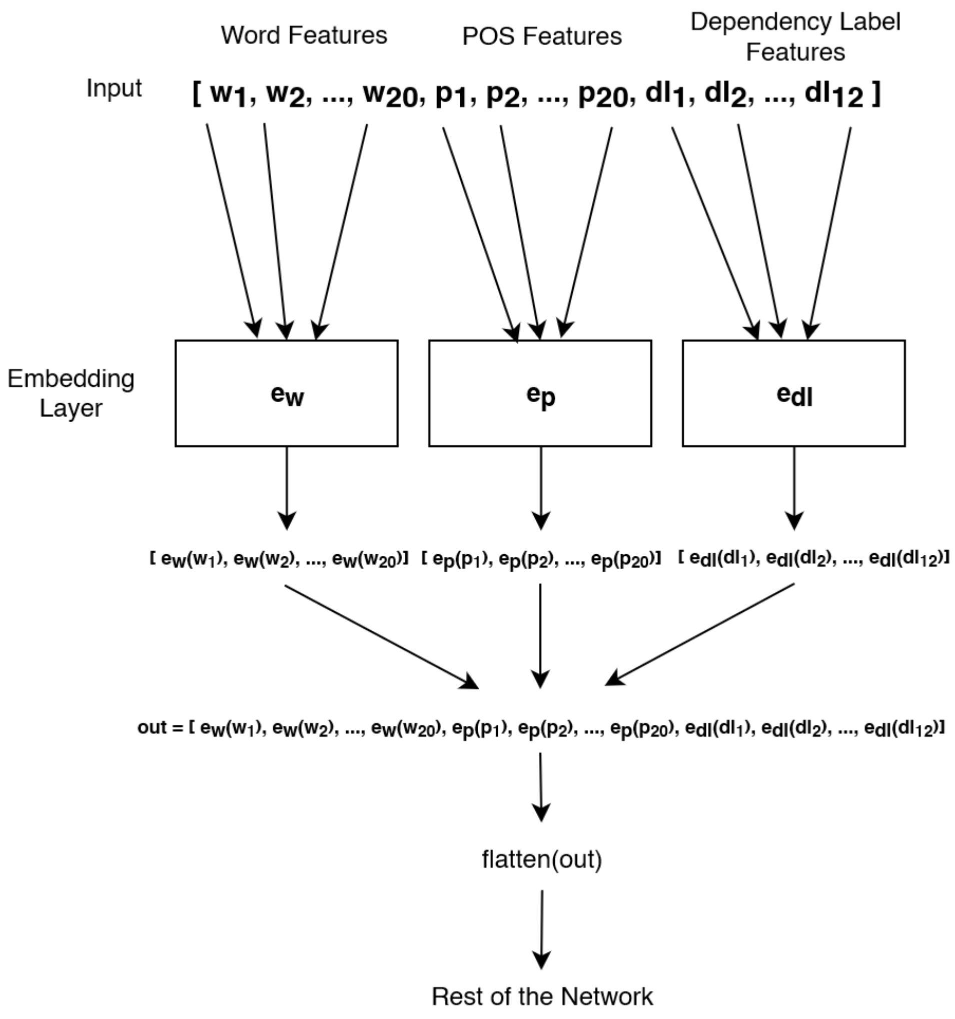
We will be passing the input data through an embedding layer consisting of the following embedding (lookup) parameters.

- Word embedding (E_w) with dimension (d_w). If we have (N_w) unique words, the size of the embedding dictionary will be $\mathbb{R}^{d_w \times N_w}$.
- POS embedding (E_t) with dimension (d_t). If we have (N_t) unique POS-tags, the size of the embedding dictionary will be $\mathbb{R}^{d_t \times N_t}$.
- Dependency labelling embedding (E_l) with dimension (d_l). If we have (N_l) unique dependency labels, the size of the embedding dictionary will be $\mathbb{R}^{d_l \times N_l}$.

You will be defining d_w, d_t, and d_l on your own, these are hyperparameters. However, N_w, N_t, and N_l will be extracted from variables obtained via preprocessing. Each feature gets mapped to an embedding of the corresponding dimension. The output of this layer is the concatenation of the flattened output of each embedding layer. Thus, we have that the output of this layer has dimension d_e where

$$d_e = 20(d_w + d_t) + 12d_l$$

The following diagram describes this system.


Word Features
POS Features
Dependency Label Features
Input
[w1, w2, ..., w20, p1, p2, ..., p20, dl1, dl2, ..., dl12]
Embedding Layer
ew
ep
edl
[ew(w1), ew(w2), ..., ew(w20)]
[ep(p1), ep(p2), ..., ep(p20)]
[edl(dl1), edl(dl2), ..., edl(dl12)]
out = [ew(w1), ew(w2), ..., ew(w20), ep(p1), ep(p2), ..., ep(p20), edl(dl1), edl(dl2), ..., edl(dl12)]
flatten(out)
Rest of the Network

3.2 Hidden Layers

The output of the embedding layer will be passed through our hidden layers. Given a list of the number of neurons for each layer, you will append them to the model. Each hidden layer has the same structure:

1. A Linear layer with the corresponding input and output sizes.
2. A LeakyReLu activation function.
3. A Dropout layer to combat overfitting with the dropout parameter.

The output of each hidden layer will be passed as input to the next.

3.3 Output Layer

The output of the final hidden layer will be passed through the output layer. Therefore, the input size would be determined by the output size of the final hidden layer. Since we are predicting the next action, we have N_a possible unique actions (this value will be extracted similarly to N_w, N_t, and N_l), and thus an output size of N_a.

Note that traditionally for classification problems we would apply 'Softmax' or 'Sigmoid' to the output layer. However, we won't be working with the probabilities and the raw, unnormalized values are enough to simply obtain the best action which is what we will be using to parse the sentence.

4 Task 2: Initializing the model

Having defined the Classifier class, we will be initializing the model using this. The model's parameters need to be defined now. The following are hyperparameters for which we have suggested a range of values in the notebook, where you may feel free to experiment:

- word_emb_size
- pos_emb_size
- depl_emb_size
- layer_sizes
- dropout

While the following are deterministic values depending on our dataset and need to be extracted from existing variables:

- word_vocab_size
- pos_vocab_size
- depl_vocab_size
- out_size

After initializing the model, you should define the optimizer, the loss function, and the number of epochs. This is standard practice for defining neural nets.

5 Task 3: Training the Model

We finally reach the training loop which you may have implemented before. You should perform the following steps, guidelines for which have been provided in the notebook:

1. Extract the input features and labels from the dataset.
2. Pass the inputs to the model to obtain the current output.
3. Calculate the loss comparing the output from the model to the reference label.

4. Reset the optimizer.
5. Backward propagate through the loss function.
6. Step through the optimizer.

6 Deliverables

We expect you to

- Copy your Classifier code from Task 1 into the 'classifier.py' file in the handout
- Download the dev.out file from the outputs folder

and submit both to Gradescope by April 20th, 2023 11:59 EST.

References

[1] Michael Collins (2019) `http://www.cs.columbia.edu/~mcollins/cs4705-spring2019/hw_nn/hw_nn_programming_spring2019.pdf`

[2] Daniel Jurfasky (2023) `https://web.stanford.edu/~jurafsky/slp3/14.pdf`

[3] Joakim Nivre (2004) `https://aclanthology.org/W04-0308.pdf`