Blog of Qing

NLP-Dependency Parser

2019-06-12 | NLP

Dependency structure of sentences shows which words depend on (modify or are arguments of) which other words. These binary asymmetric relations between the words are called dependencies and are depicted as arrows going from the head (or governor, superior, regent) to the dependent (or modifier, inferior, subordinate).

What is Dependency Parsing

Dependency parsing is the task of analyzing the syntactic dependency structure of a given input sentence S. The output of a dependency parser is a dependency tree where the words of the input sentence are connected by typed dependency relations. Formally, the dependency parsing problem asks to create a mapping from the input sentence with words $S = w_0 w_1 \dots w_n$ (where w_0 is the ROOT) to its dependency tree graph G.

To be precise, there are two subproblems in dependency parsing:

1. Learning: Given a training set D of sentences annotated with dependency graphs, induce a parsing model M that can be used to
   parse new sentences.
2. Parsing: Given a parsing model M and a sentence S, derive the
   optimal dependency graph D for S according to M.

How to train a Dependency Parsing

Gready Deterministic Transition-Based Parsing

This transition system is a state machine, which consists of states and transitions between those states. The model induces a sequence of transitions from some initial state to one of several terminal states.

States:

For any sentence $S = w_0 w_1 \dots w_n$, a state can be described with a triple $c = (\alpha, \beta, A)$:

1. a stack α of words w_i from S
2. a buffer β of words w_i from S
3. a set of dependency arcs A of the form (w_i, r, w_j), where r describes a dependency relation

It follows that for any sentence $S = w_0 w_1 \dots w_n$,

1. an initial state c_0 is of the form $([w_0]_\alpha, [w_1, \dots, w_n]_{beta}, \empty)$ (only the ROOT is on the stack, all other words are in the buffer and no actions have been chosen yet)
2. a terminate state has the form $(\alpha, []_\beta, A)$

Transitions:

There are three types of transitions between states:

1. Shift: Remove the first word in the buffer and push it on top of the stack. (Pre-condition: buffer has to be non-empty.)

2. LEFT-ARC: Add a dependency arc (w_j, r, w_i) to the arc set A, where w_i is the word second to the top of the stack and w_j is the word at the top of the stack. Remove w_i from the stack.

   > $[w_0, w_1, \dots, w_i \leftarrow w_j]$

3. RIGHT-ARC: Add a dependency arc (w_i, r, w_j) to the arc set A, where w_i is the word second to the top of the stack and w_j is the word at the top of the stack. Remove w_j from the stack.

   > $[w_0, w_1, \dots, w_i \rightarrow w_j]$

Neural Dependency Parsing

The network is to predict the transitions between words, i.e., $Shift, Left-Arc, Right-Arc$.

Feature Selection:

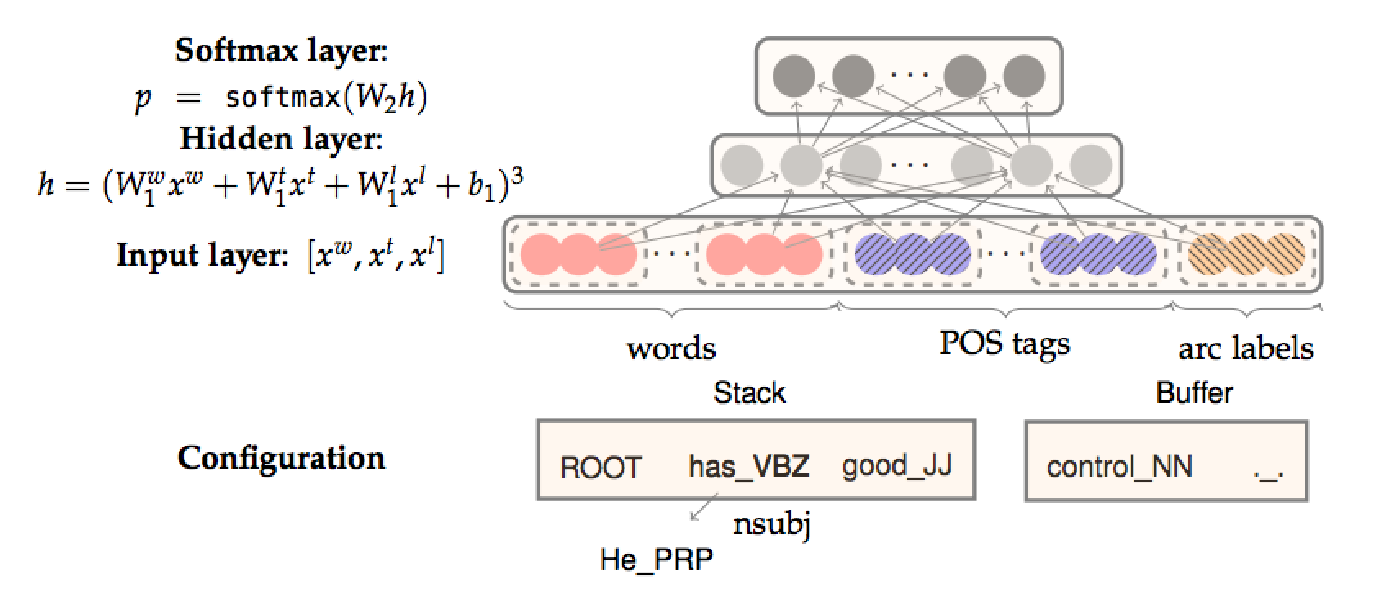
Depending on the desired complexity of the model, there is flexibility in defining the input to the neural network. The features for a given sentence S generally include some subset of:

1. S_{word}: Vector representations for some of the words in S (and their dependents) at the top of the stack σ and buffer β.
2. S_{tag}: Part-of-Speech (POS) tags for some of the words in S. POS tags comprise a small, discrete set: $\mathcal{P} = \{NN, NNP, NNS, DT, JJ, ...\}$
3. S_{label}: The arc-labels for some of the words in S. The arc-labels comprise a small, discrete set, describing the dependency relation: $\mathcal{L} = \{amod, tmod, nsubj, csubj, dobj, ...\}$

For each feature type, we will have a corresponding embedding matrix, mapping from the feature's one hot encoding, to a d-dimensional
dense vector representation.

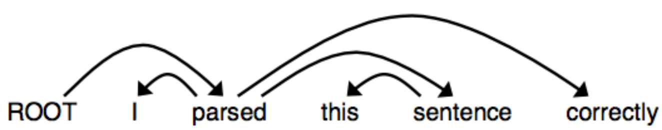
Feedforward Neural Network Model:

The network contains an input layer $[x^w, x^t, x^l]$, a hidden layer, and a final softmax layer with a cross-entropy loss function. We can either define a single weight matrix in the hidden layer, to operate on a concatenation of $[x^w, x^t, x^l]$, or we can use three weight matrices $[W_1^w, W_1^t, W_1^l]$, one for each input type, as shown in Figure 3. We then apply a non-linear function and use one more affine layer $[W_2]$ so that there are an equivalent number of softmax probabilities to the number of possible transitions (the output dimension).



Example

(a) (6 points) Go through the sequence of transitions needed for parsing the sentence *"I parsed this sentence correctly"*. The dependency tree for the sentence is shown below. At each step, give the configuration of the stack and buffer, as well as what transition was applied this step and what new dependency was added (if any). The first three steps are provided below as an example.



Stack	Buffer	New dependency	Transition
[ROOT]	[I, parsed, this, sentence, correctly]		Initial Configuration
[ROOT, I]	[parsed, this, sentence, correctly]		SHIFT
[ROOT, I, parsed]	[this, sentence, correctly]		SHIFT
[ROOT, parsed]	[this, sentence, correctly]	parsed→I	LEFT-ARC

Answer:

Stack	Buffer	New dependency	Transition
[ROOT]	[I, parsed, this, sentence, correctly]		Initial Configuration
[ROOT, I]	[parsed, this, sentence, correctly]		SHIFT
[ROOT,I, parsed]	[this , sentence ,correctly]		SHIFT
[ROOT, parsed]	[this, sentence, correctly]	parsed → I	LEFT-ARC
[ROOT, parsed, this]	[sentence, correctly]		SHIFT
[ROOT, parsed, this,sentence]	[correctly]		SHIFT
[ROOT, parsed, sentence]	[correctly]	sentence → this	LEFT-ARC
[ROOT, parsed]	[correctly]	parsed → sentence	RIGHT-ARC
[ROOT,parsed,correctly]	[]		SHIFT
[ROOT, parsed]	[]	parsed → correctly	RIGHT-ARC
[ROOT]	[]	ROOT → parsed	RIGHT-ARC

> A sentence containing n words will be parsed in $2n$ steps. Because for each word, it will be pushed from buffer to stack, which result in n steps. Then each word in the stack has to be assigned a transition, i.e., LEFT-ARC or RIGHT-ARC, which results in another n steps. Therefore, the total steps are $2n$.

Pytorch Implementation

```
class PartialParse(object):
    def __init__(self, sentence):
        """Initializes this partial parse.

        @param sentence (list of str): The sentence to be parsed as a list of words.
                                        Your code should not modify the sentence.
        """
```

```
        # The sentence being parsed is kept for bookkeeping purposes. Do not alter it in yo
        self.sentence = sentence

        ### YOUR CODE HERE (3 Lines)
        ### Your code should initialize the following fields:
        ###     self.stack: The current stack represented as a list with the top of the sta
        ###                 last element of the list.
        ###     self.buffer: The current buffer represented as a list with the first item o
        ###                  buffer as the first item of the list
        ###     self.dependencies: The list of dependencies produced so far. Represented as
        ###             tuples where each tuple is of the form (head, dependent).
        ###             Order for this list doesn't matter.
        ###
        ### Note: The root token should be represented with the string "ROOT"
        ###
        self.stack = ['ROOT']
        self.buffer = sentence.copy()
        self.dependencies = []
        ### END YOUR CODE


    def parse_step(self, transition):
        """Performs a single parse step by applying the given transition to this partial pa
```

```
        @param transition (str): A string that equals "S", "LA", or "RA" representing the s
                                left-arc, and right-arc transitions. You can assume the pro
                                transition is a legal transition.
        """
        ### YOUR CODE HERE (~7-10 Lines)
        ### TODO:
        ###     Implement a single parsing step, i.e. the logic for the following as
        ###     described in the pdf handout:
        ###         1. Shift
        ###         2. Left Arc
        ###         3. Right Arc
        if transition=='S':
            self.stack.append(self.buffer.pop(0))
        elif transition=='LA':
            stack_second = self.stack.pop(-2)
            self.dependencies.append((self.stack[-1],stack_second))
        else:
            stack_top = self.stack.pop()
            self.dependencies.append((self.stack[-1],stack_top))


        ### END YOUR CODE

    def parse(self, transitions):
```

```
        """Applies the provided transitions to this PartialParse

        @param transitions (list of str): The list of transitions in the order they should

        @return dsependencies (list of string tuples): The list of dependencies produced wh
                                                        parsing the sentence. Represented a
                                                        tuples where each tuple is of the f
        """
        for transition in transitions:
            self.parse_step(transition)
        return self.dependencies


def minibatch_parse(sentences, model, batch_size):
    """Parses a list of sentences in minibatches using a model.

    @param sentences (list of list of str): A list of sentences to be parsed
                                            (each sentence is a list of words and each word
    @param model (ParserModel): The model that makes parsing decisions. It is assumed to ha
                                model.predict(partial_parses) that takes in a list of Parti
                                returns a list of transitions predicted for each parse. Tha
                                    transitions = model.predict(partial_parses)
                                transitions[i] will be the next transition to apply to part
    @param batch_size (int): The number of PartialParses to include in each minibatch
```

```
    @return dependencies (list of dependency lists): A list where each element is the depen
                                                    list for a parsed sentence. Ordering sh
                                                    same as in sentences (i.e., dependencie
                                                    contain the parse for sentences[i]).
    """
    dependencies = []

    ### YOUR CODE HERE (~8-10 Lines)
    ### TODO:
    ###     Implement the minibatch parse algorithm as described in the pdf handout
    ###
    ###     Note: A shallow copy (as denoted in the PDF) can be made with the "=" sign in p
    ###                 unfinished_parses = partial_parses[:].
    ###             Here `unfinished_parses` is a shallow copy of `partial_parses`.
    ###             In Python, a shallow copied list like `unfinished_parses` does not cont
    ###             of the object stored in `partial_parses`. Rather both lists refer to th
    ###             In our case, `partial_parses` contains a list of partial parses. `unfin
    ###             contains references to the same objects. Thus, you should NOT use the `
    ###             to remove objects from the `unfinished_parses` list. This will free the
    ###             is being accessed by `partial_parses` and may cause your code to crash.

    #initialization
```

```
    partial_parses = []
    for sentence in sentences:
        partial_parses.append(PartialParse(sentence))
    unfinished_parses = partial_parses[:]

    while unfinished_parses:
        minibatch = unfinished_parses[:batch_size]
        transitions = model.predict(minibatch)
        for i,parses in enumerate(minibatch):
            parses.parse([transitions[i]])
            if len(parses.buffer)==0 and len(parses.stack)==1:
                dependencies.append(parses.dependencies)
                unfinished_parses.remove(parses)
    ### END YOUR CODE
    return dependencies
```

Train Parser Network

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class ParserModel(nn.Module):
    """ Feedforward neural network with an embedding layer and single hidden layer.
    The ParserModel will predict which transition should be applied to a
```

```
    given partial parse configuration.

    PyTorch Notes:
        - Note that "ParserModel" is a subclass of the "nn.Module" class. In PyTorch all ne
            are a subclass of this "nn.Module".
        - The "__init__" method is where you define all the layers and their respective par
            (embedding layers, linear layers, dropout layers, etc.).
        - "__init__" gets automatically called when you create a new instance of your class
            when you write "m = ParserModel()".
        - Other methods of ParserModel can access variables that have "self." prefix. Thus,
            you should add the "self." prefix layers, values, etc. that you want to utilize
            in other ParserModel methods.
        - For further documentation on "nn.Module" please see https://pytorch.org/docs/stab
    """
    def __init__(self, embeddings, n_features=36,
        hidden_size=200, n_classes=3, dropout_prob=0.5):
        """ Initialize the parser model.

        @param embeddings (Tensor): word embeddings (num_words, embedding_size)
        @param n_features (int): number of input features
        @param hidden_size (int): number of hidden units
        @param n_classes (int): number of output classes
        @param dropout_prob (float): dropout probability
```

```
        """
        super(ParserModel, self).__init__()
        self.n_features = n_features
        self.n_classes = n_classes
        self.dropout_prob = dropout_prob
        self.embed_size = embeddings.shape[1]
        self.hidden_size = hidden_size
        self.pretrained_embeddings = nn.Embedding(embeddings.shape[0], self.embed_size)
        self.pretrained_embeddings.weight = nn.Parameter(torch.tensor(embeddings))

        ### YOUR CODE HERE (~5 Lines)
        ### TODO:
        ###     1) Construct `self.embed_to_hidden` linear layer, initializing the weight m
        ###         with the `nn.init.xavier_uniform_` function with `gain = 1` (default)
        ###     2) Construct `self.dropout` layer.
        ###     3) Construct `self.hidden_to_logits` linear layer, initializing the weight
        ###         with the `nn.init.xavier_uniform_` function with `gain = 1` (default)
        ###
        ### Note: Here, we use Xavier Uniform Initialization for our Weight initialization.
        ###         It has been shown empirically, that this provides better initial weight
        ###         for training networks than random uniform initialization.
        ###         For more details checkout this great blogpost:
        ###             http://andyljones.tumblr.com/post/110998971763/an-explanation-of-xa
        ### Hints:
```

```
        ###     - After you create a linear layer you can access the weight
        ###       matrix via:
        ###         linear_layer.weight
        ###
        ### Please see the following docs for support:
        ###     Linear Layer: https://pytorch.org/docs/stable/nn.html#torch.nn.Linear
        ###     Xavier Init: https://pytorch.org/docs/stable/nn.html#torch.nn.init.xavier_u
        ###     Dropout: https://pytorch.org/docs/stable/nn.html#torch.nn.Dropout

        self.embed_to_hidden = nn.Linear(self.embed_size*self.n_features,self.hidden_size)
        nn.init.xavier_uniform_(self.embed_to_hidden.weight)
        self.dropout = nn.Dropout(self.dropout_prob)
        self.hidden_to_logits = nn.Linear(hidden_size,self.n_classes)
        nn.init.xavier_uniform_(self.hidden_to_logits.weight)

        ### END YOUR CODE

    def embedding_lookup(self, t):
        """ Utilize `self.pretrained_embeddings` to map input `t` from input tokens (intege
            to embedding vectors.

            PyTorch Notes:
                - `self.pretrained_embeddings` is a torch.nn.Embedding object that we defin
                - Here `t` is a tensor where each row represents a list of features. Each f
```

```
                - In PyTorch the Embedding object, e.g. `self.pretrained_embeddings`, allow
                    go from an index to embedding. Please see the documentation (https://py
                    to learn how to use `self.pretrained_embeddings` to extract the embeddi

            @param t (Tensor): input tensor of tokens (batch_size, n_features)

            @return x (Tensor): tensor of embeddings for words represented in t
                                (batch_size, n_features * embed_size)
        """
        ### YOUR CODE HERE (~1-3 Lines)
        ### TODO:
        ###     1) Use `self.pretrained_embeddings` to lookup the embeddings for the input
        ###     2) After you apply the embedding lookup, you will have a tensor shape (batc
        ###         Use the tensor `view` method to reshape the embeddings tensor to (batch
        ###
        ### Note: In order to get batch_size, you may need use the tensor .size() function:
        ###         https://pytorch.org/docs/stable/tensors.html#torch.Tensor.size
        ###
        ###  Please see the following docs for support:
        ###     Embedding Layer: https://pytorch.org/docs/stable/nn.html#torch.nn.Embedding
        ###     View: https://pytorch.org/docs/stable/tensors.html#torch.Tensor.view
        batch_size = t.shape[0]
        x = self.pretrained_embeddings(t).view(batch_size,-1)
        ### END YOUR CODE
```

```
        return x


    def forward(self, t):
        """ Run the model forward.

            Note that we will not apply the softmax function here because it is included in

            PyTorch Notes:
                - Every nn.Module object (PyTorch model) has a `forward` function.
                - When you apply your nn.Module to an input tensor `t` this function is app
                    For example, if you created an instance of your ParserModel and applied
                    the `forward` function would called on `t` and the result would be stor
                        model = ParserModel()
                        output = model(t) # this calls the forward function
                - For more details checkout: https://pytorch.org/docs/stable/nn.html#torch.

        @param t (Tensor): input tensor of tokens (batch_size, n_features)

        @return logits (Tensor): tensor of predictions (output after applying the layers of
                                 without applying softmax (batch_size, n_classes)
        """
        ###  YOUR CODE HERE (~3-5 lines)
        ### TODO:
```

```
        ###     1) Apply `self.embedding_lookup` to `t` to get the embeddings
        ###     2) Apply `embed_to_hidden` linear layer to the embeddings
        ###     3) Apply relu non-linearity to the output of step 2 to get the hidden units
        ###     4) Apply dropout layer to the output of step 3.
        ###     5) Apply `hidden_to_logits` layer to the output of step 4 to get the logits
        ###
        ### Note: We do not apply the softmax to the logits here, because
        ### the loss function (torch.nn.CrossEntropyLoss) applies it more efficiently.
        ###
        ### Please see the following docs for support:
        ###     ReLU: https://pytorch.org/docs/stable/nn.html?highlight=relu#torch.nn.funct
        features_embeddings = self.embedding_lookup(t)
        o = F.relu(self.embed_to_hidden(features_embeddings))
        o = self.dropout(o)
        logits = self.hidden_to_logits(o)
        ### END YOUR CODE
        return logits
```

```
def train(parser, train_data, dev_data, output_path, batch_size=1024, n_epochs=10, lr=0.0005
    """ Train the neural dependency parser.

    @param parser (Parser): Neural Dependency Parser
    @param train_data ():
    @param dev_data ():
```

```python
    @param output_path (str): Path to which model weights and results are written.
    @param batch_size (int): Number of examples in a single batch
    @param n_epochs (int): Number of training epochs
    @param lr (float): Learning rate
    """
    best_dev_UAS = 0


    ### YOUR CODE HERE (~2-7 lines)
    ### TODO:
    ###      1) Construct Adam Optimizer in variable `optimizer`
    ###      2) Construct the Cross Entropy Loss Function in variable `loss_func`
    ###
    ### Hint: Use `parser.model.parameters()` to pass optimizer
    ###       necessary parameters to tune.
    ### Please see the following docs for support:
    ###     Adam Optimizer: https://pytorch.org/docs/stable/optim.html
    ###     Cross Entropy Loss: https://pytorch.org/docs/stable/nn.html#crossentropyloss
    optimizer = optim.Adam(parser.model.parameters(),lr=lr)
    loss_func = nn.CrossEntropyLoss()

    ### END YOUR CODE

    for epoch in range(n_epochs):
```

```
    for epoch in range(n_epochs):
        print("Epoch {:} out of {:}".format(epoch + 1, n_epochs))
        dev_UAS = train_for_epoch(parser, train_data, dev_data, optimizer, loss_func, batch_
        if dev_UAS > best_dev_UAS:
            best_dev_UAS = dev_UAS
            print("New best dev UAS! Saving model.")
            torch.save(parser.model.state_dict(), output_path)
        print("")


def train_for_epoch(parser, train_data, dev_data, optimizer, loss_func, batch_size):
    """ Train the neural dependency parser for single epoch.

    Note: In PyTorch we can signify train versus test and automatically have
    the Dropout Layer applied and removed, accordingly, by specifying
    whether we are training, `model.train()`, or evaluating, `model.eval()`

    @param parser (Parser): Neural Dependency Parser
    @param train_data ():
    @param dev_data ():
    @param optimizer (nn.Optimizer): Adam Optimizer
    @param loss_func (nn.CrossEntropyLoss): Cross Entropy Loss Function
    @param batch_size (int): batch size
    @param lr (float): learning rate
```

```
    @return dev_UAS (float): Unlabeled Attachment Score (UAS) for dev data
    """
    parser.model.train() # Places model in "train" mode, i.e. apply dropout layer
    n_minibatches = math.ceil(len(train_data) / batch_size)
    loss_meter = AverageMeter()
    dev_UAS, _ = parser.parse(dev_data)

    with tqdm(total=(n_minibatches)) as prog:
        for i, (train_x, train_y) in enumerate(minibatches(train_data, batch_size)):
            optimizer.zero_grad()   # remove any baggage in the optimizer
            loss = 0. # store loss for this batch here
            train_x = torch.from_numpy(train_x).long()
            train_y = torch.from_numpy(train_y.nonzero()[1]).long()

            ### YOUR CODE HERE (~5-10 lines)
            ### TODO:
            ###      1) Run train_x forward through model to produce `logits`
            ###      2) Use the `loss_func` parameter to apply the PyTorch CrossEntropyLoss
            ###         This will take `logits` and `train_y` as inputs. It will output the
            ###         between softmax(`logits`) and `train_y`. Remember that softmax(`logi
            ###         are the predictions (y^ from the PDF).
            ###      3) Backprop losses
            ###      4) Take step with the optimizer
            ### Please see the following docs for support:
```

```
            ### Please see the following docs for support:
            ###     Optimizer Step: https://pytorch.org/docs/stable/optim.html#optimizer-ste
            logits = parser.model(train_x)
            loss = loss_func(logits,train_y)
            loss.backward()
            optimizer.step()
            ### END YOUR CODE
            prog.update(1)
            loss_meter.update(loss.item())

    print ("Average Train Loss: {}".format(loss_meter.avg))

    print("Evaluating on dev set",)
    parser.model.eval() # Places model in "eval" mode, i.e. don't apply dropout layer
    dev_UAS, _ = parser.parse(dev_data)
    print("- dev UAS: {:.2f}".format(dev_UAS * 100.0))
    return dev_UAS
```

Deep Learning # NLP

< DP-Neural Network NLP-Seq2Seq >