SI 630: Homework 3 – Dependency Parsing

Verion 1.0

Due: Wednesday, March 31, 11:59pm

1 Introduction

Despite its seeming chaos, natural language has lots of structure. We've already seen some of this structure in part of speech tags and how the order of parts of speech are predictive of what kinds of words might come next (via their parts of speech). In Homework 3, you'll get a deeper view of this structure by implementing a **dependency parser**. We covered this topic in Week 10 of the course and it's covered extensively in Speech & Language Processing chapter 13, if you want to brush up .[1] Briefly, dependency parsing identifies the syntactic relationship between word pairs to create a *parse tree*, like the one seen in Figure 1.

In Homework 3, you'll implement the *shift-reduce* neural dependency parser of Chen and Manning [2014],[2] which was one of the first neural network-based parser and is quite famous. Thankfully, its neural network is also fairly straight-forward to implement. We've provided the parser's skeleton code in Python 3 that you can use to finish the implementation, with comments that outline the steps you'll need to finish. And, importantly, we've provided a *lot* of boilerplate code that handles loading in the training, evaluation, and test dataset, and converting that data into a representation suitable for the network. Your part essentially boils down to two steps: (1) fill in the implementation of the neural network and (2) fill in the main training loop that processes each batch of instances and does backprop. Thankfully, *unlike* in Homeworks 1 and 2, you'll be leveraging the miracles of modern deep learning libraries to accomplish both of these!

Homework 3 has the following learning goals:

[1] `https://web.stanford.edu/~jurafsky/slp3/13.pdf`

[2] `https://cs.stanford.edu/~danqi/papers/emnlp2014.pdf`

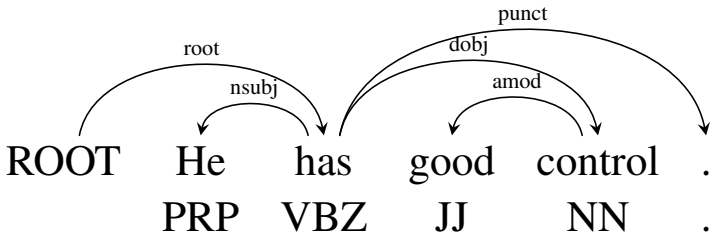

Figure 1: An example dependency parse from the Chen and Manning [2014] paper. Note that each word is connected to another word to symbolize its syntactic relationship. Your parser will determine these edges and their types!

1. Gain a working knowledge of the PyTorch library, including constructing a basic network, using layers, dropout, and loss functions.
2. Learn how to train a network with PyTorch
3. Learn how to use pre-trained embeddings in downstream applications
4. Learn about the effects of changing different network hyper parameters and designs
5. Gain a basic familiarity with dependency parsing and how a shift-reduce parser works.

You'll notice that most of the learning goals are based on deep learning topics, which is the primary focus of this homework. The skills you learn with this homework will hopefully help you with your projects and (ideally) with any real-world situation where you'd need to build a new network. However, you're welcome—encouraged, even!—to wade into the parsing setup and evaluation code to understand more of how this kind of model works.

In Homework 3, we've also included several *optional* tasks for those that feel ambitious. Please finish the regular homework first before even considering these tasks. There is no extra credit for completing any of these optional tasks, only glory and knowledge.

2 PyTorch

Homework 3 will use the PyTorch deep learning library. However, your actual implementation will use only a small part of the library's core functionality, which should be enough to get you building networks. Rather than try to explain all of PyTorch in a mere homework write-up, we'll refer you to the fantastic PyTorch community tutorials[3] for comprehensive coverage. Note that you *do not* need to read all these tutorials! We're only building a feed forward network here, so there's no need to read up on Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), or any variant thereof. Instead, as a point of departure into deep learning land, try walking through this tutorial on Logistic Regression[4] which is the PyTorch version of what you implemented in Homework 1. That tutorial will hopefully help you see how the things you had to implement in HW1 get greatly simplified when using these deep learning libraries (e.g., compare their stochastic gradient descent code with yours!).

The biggest conceptual change for using PyTorch with this homework will be using *batching*. We talked briefly about batching during the Logistic Regression lecture, where instead of using just a single data point to compute the gradient, you use several—or a *batch*. In practice, using a batch of instances greatly speeds up the convergence of the model. Further when using a GPU, often batching is significantly more computationally efficient because you'll be using more of the special matrix multiplication processor at once (GPUs are designed to do lots of multiplications in parallel, so batching helps "fill the capacity" of work that can be done in a single time step). In practice, we've already set up the code for you to be in batches so when you get an instance with k features, you're really getting a tensor[5] of size $b \times k$ where b is the batch size. Thanks to the magic

[3] `https://pytorch.org/tutorials/`

[4] `https://www.kaggle.com/negation/pytorch-logistic-regression-tutorial`

[5] Tensor is a fancier name for multi-dimensional data. A vector is a 1-dimensional tensor and a matrix is a 2-dimensional tensor. Most of the operations for deep learning libraries will talk about "tensors" so it's important to get used to this terminology.

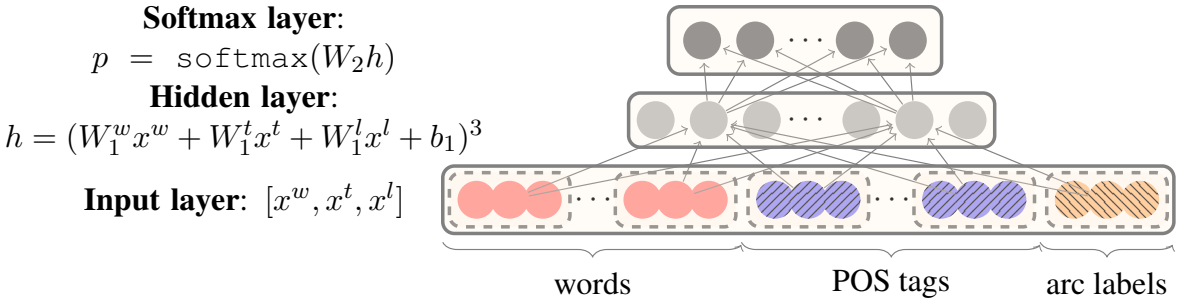

Figure 2: The network architecture for the Chen and Manning [2014] parser. Note that this is a feed-forward neural network, which is just one more layer than logistic regression!

of PyTorch, you can effectively treat this as a vector of size k and PyTorch will deal with the fact that it's "batched" for you;[6] i.e., you can write your neural network in a way that ignores batching and it will just happen naturally (and more efficiently).[7]

3 The Parser

The Chen and Manning [2014] parser is a feed-forward neural network that encodes lexical features from the context (i.e., words on the stack and words on the buffer) as well as their parts of speech and the current dependency arcs that have been produced. Figure 2 shows a diagram of the network. The input layer consists of three pieces, x^w, w^t, and x^l, which denote the embeddings for the words, POS tags, and dependency args. Each of these embeddings is actually *multiple* embeddings concatenated together; i.e., if we represent the two words on the top if the stack, each with 50-dimensional embeddings, then x^w has a total length of $50 \times 2 = 100$. Said another way, $x^w = [e^w_{w_1}; e^w_{w_2}, \ldots, e^w_{w_n}]$ where $e^w_{w_i}]$ denotes the embedding for w_i and ; is the concatenation operator. Each of the input types has separate weights W_1 for computing hidden layer raw output (before the activation function is applied), i.e., W_1^w, W_1^t, and W_1^l.

Normally, we've talked about activation functions like sigmoid or a Rectified Linear Unit (ReLU); however, in the Chen and Manning [2014] parser, we'll use a different activation function. Specifically, you'll implement a *cubic* activation function that cubes the raw output value of a neuron. As a result, the activation of the hidden layer in the original paper is computed as $h = (W_1^w x^w + W_1^t x^t + W_1^l x^l + b_1)^2$ where b_1 is the bias term.

Chen and Manning [2014] use separate weight matrices for words, POS tags, and dependencies to make use of several optimizations which are not described here. However, we can simplify this equation and the implementation by using one weight matrix W_1 that takes in the concatenated inputs to make the activation $h = (W_1[x^w; x^t; x^l; b_1])^3$. In your implementation, this means you can use a single layer to represent all three weight matrices (W_1^w, W_1^t, and W_1^l) which should simplify your bookkeeping.

[6] For a useful demo of how this process works, see `https://adventuresinmachinelearning.com/pytorch-tutorial-deep-learning/`

[7] In later assignments where we have sequences, we'll need to revise this statement to make things *even more* efficient!

Transition	Stack	Buffer	A
	[ROOT]	[He has good control .]	$\emptyset$
SHIFT	[ROOT He]	[has good control .]	
SHIFT	[ROOT He has]	[good control .]	
LEFT-ARC(nsubj)	[ROOT has]	[good control .]	$A\cup$ nsubj(has,He)
SHIFT	[ROOT has good]	[control .]	
SHIFT	[ROOT has good control]	[.]	
LEFT-ARC(amod)	[ROOT has control]	[.]	$A\cup$amod(control,good)
RIGHT-ARC(dobj)	[ROOT has]	[.]	$A\cup$ dobj(has,control)
...	...	...	...
RIGHT-ARC(root)	[ROOT]	[]	$A\cup$ root(ROOT,has)

Figure 3: An example snippet of the parsing stack and buffer from Chen and Manning [2014] using the sentence in Figure 1. This diagram is an example of how a transition-based dependency parser works. At each step, the parser decides whether to (1) *shift* a word from the buffer onto the stack or (2) *reduce* the size of the stack by forming an edge between the top two words on the stack (further deciding which direction the edge goes).

The final outputs are computed by multiplying the hidden layer activation by the second layer's weights and passing that through a softmax:[8] $p = \texttt{softmax}(W_2 h)$.

This might all seem like a lot of math and/or implementation, but PyTorch is going to take care of most of this for you!

4 Implementation Notes

The implementation is broken up into five key files, only two of which you need to deal with:

- `main.py` — This file drives the whole program and is what you'll run from the command line. It also contains the basic training loop, which you'll need to implement. You can run `python main.py -h` to see all the options
- `model.py` — This file specifies the neural network model used to do the parsing, which you'll implement.
- `feature_extraction.py` — This file contains all the gory details of reading the Penn Treebank data and turning it into training and test instances. This is where most of the parsing magic happens. You don't need to read any of this file, but if you're curious, please do!
- `test_functions.py` — This code does all the testing and, crucially, computes the **Unlabeled Attachment Score** (UAS) you'll use to evaluate your parser. You don't need to read this file either.
- `general_utils.py` — Random utility functions.

[8] Reminder: the softmax is the generalization of the sigmoid (σ) function for multi-class problems here.

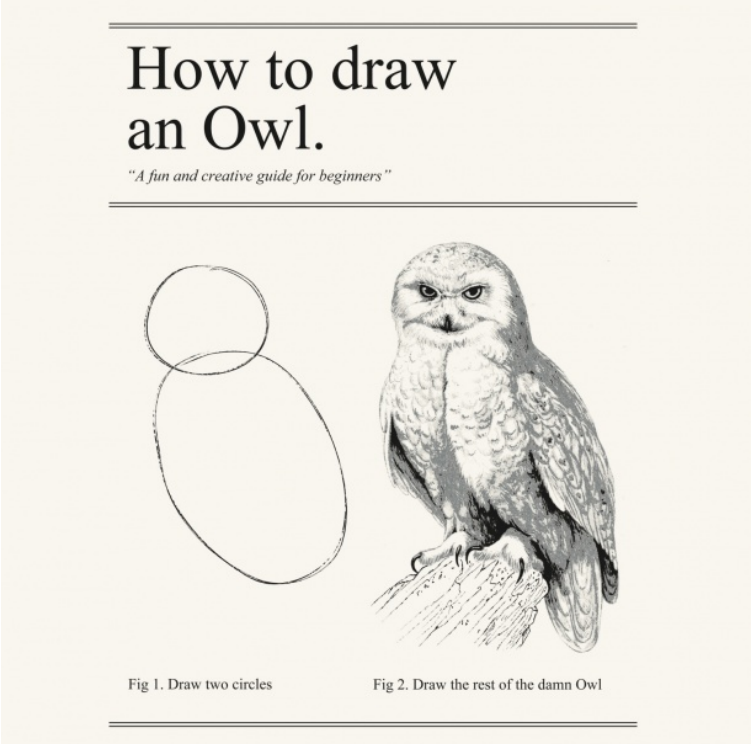

Figure 4: The analogy for how it *might* seem to implement this parser based on the instructions, but in reality, it's not too hard!

We've provided skeleton code with TODOs for where the main parts of what you need to do are sketched out. All of your required code should be written in `main.py` or `model.py`, though you are welcome to read or adjust the other file's code to help debug, be more verbose, or just poke around to see what it does.

5 Data

Data has already been provided for you in the `data/` directory in CoNLL format. You do not need to deal with the data itself, as the `feature_extraction.py` code can already read the data and generate training instances.

6 Task 1: Finish the implementation

In Task 1, you'll implement the feed-forward neural network in `main.py` based on the description in this write-up or the original paper. Second, you'll implement the core training loop which will

1. Loop through the dataset for the specified number of epochs
2. Sample a batch of instances
3. Produce predictions for each instance in the batch
4. Score those predictions using your loss function

5. Perform backpropagation and update the parameters.

Many of these tasks are straightforward with PyTorch and none of them should require complex operations. Having a good understanding of how to implement/train logistic regression in PyTorch will go a long way.

The `main.py` file works with command line flags to enable quick testing and training. To train your system, run `python main.py --train`. Note that this will break on the released code since the model is not implemented! However, once it's working, you should see an output that looks something like the following after one epoch:

```
Loading dataset for training
Loaded Train data
Loaded Dev data
Loaded Test data
Vocab Build Done!
embedding matrix Build Done
converting data into ids..
Done!
Loading embeddings
Creating new trainable embeddings
words:  39550

some hyperparameters
{'load_existing_vocab': True, 'word_vocab_size': 39550,
'pos_vocab_size': 48, 'dep_vocab_size': 42,
'word_features_types': 18, 'pos_features_types': 18,
'dep_features_types': 12, 'num_features_types': 48,
'num_classes': 3}

Epoch: 1 [0], loss: 99.790, acc: 0.309
Epoch: 1 [50], loss: 4.170, acc: 0.786
Epoch: 1 [100], loss: 2.682, acc: 0.795
Epoch: 1 [150], loss: 1.795, acc: 0.818
Epoch: 1 [200], loss: 1.320, acc: 0.840
Epoch: 1 [250], loss: 1.046, acc: 0.837
Epoch: 1 [300], loss: 0.841, acc: 0.843
Epoch: 1 [350], loss: 0.715, acc: 0.848
Epoch: 1 [400], loss: 0.583, acc: 0.854
Epoch: 1 [450], loss: 0.507, acc: 0.864
Epoch: 1 [500], loss: 0.495, acc: 0.863
Epoch: 1 [550], loss: 0.487, acc: 0.863
Epoch: 1 [600], loss: 0.423, acc: 0.869
Epoch: 1 [650], loss: 0.386, acc: 0.867
Epoch: 1 [700], loss: 0.338, acc: 0.867
Epoch: 1 [750], loss: 0.340, acc: 0.874
Epoch: 1 [800], loss: 0.349, acc: 0.868
Epoch: 1 [850], loss: 0.320, acc: 0.873
Epoch: 1 [900], loss: 0.322, acc: 0.879
End of epoch
Saving current state of model to saved_weights/parser-epoch-1.mdl
Evaluating on valudation data after epoch 1
Validation acc: 0.341
- validation UAS: 70.42
```

Here, the core training loop is printing out the accuracy at each step as well as the cross-entropy loss. At the end of the epoch, the core loop scores the model on the validation data and reports the UAS, which is the score we care about. Further, the core loop will save the model after each epoch in `saved_weights`.

7 Task 2: Score Your System

We want to build a good parser and to measure how good our parser is doing, we'll use UAS for this assignment, which corresponds to the percentage of words that have the correct head in the dependency arc. Note that this score isn't looking at the particular label (e.g., `nsubj`), just whether we've created the correct parsing structure. For lots more details on how to evaluate parsers, see Kübler et al. [2009] page 79.

For Task 2, you'll measure the performance of your system performance relative to the number of training epochs and evaluate on the final test data. This breaks down into the following problems to solve.

Problem 2.1. Train your system for *at least* 5 epochs, which should generate 5 saved models in `saved-weights`. For each of these saved models, compute the UAS score.[9] You'll make three plots: (1) the loss during training for each epoch, (2) the accuracy for each epoch during training, and (3) the UAS score for the test data for each epoch's model.[10] You can make a nice plot each run's performance using Seaborn: `http://seaborn.pydata.org/examples/wide_data_lineplot.html`

Problem 2.2. Write *at least* three sentences describing what you see in the graphs and when you would want to stop training.

8 Task 3: Try different network designs and hyperparameters

Why stop at 1 hidden layer?? And why not use a ReLU instead of Cubic for an activation function? In Task 3, you get to try out different network architectures. We suggest trying out some of the following and then repeating Task 2 to see how performance changes. For easier debugging and replicability, you should make a new class that is a copy of `ParserModel` (once it's working for Task 2) and make all your modifications to that class.

Some suggested modifications are:

1. Add 1 or more layers to the network.
2. Add normalization or regularization to the layers.
3. Change to a different activation function
4. Change the size (number of neurons) in layers

[9] The code makes it easy to do this where you can specify which saved model to use, e.g., `python main.py --test --load_model_file saved_weights/parser-epoch-2.mdl`

[10] For a fun exercise, try doing this process 4-5 times and see how much variance there is.

5. Change the embedding size

How high can you get the performance to go?

Problem 3.1. Train your system for *at least* 5 epochs and generate the same plots as in Problem 2.1 for this new model's performance but include both the old model and the new model's performances in each.

8.1 Task 4: What's the parser doing, anyway?

A big part of the assignment is learning about how to build neural networks that solve NLP problems. However, we care about more than just a single metric! In Task 4, you'll look at the actual shift-reduce parsing output to see how well your model is doing. We've already provided the functionality for you to input a sentence and have the model print out (1) the steps that the shift-reduce parser takes and (2) the resulting parse. This functionality is provided using the `--parse_sentence` argument that takes in a string.

```
python main.py --parse_sentence "I eat" --load_model_file \
  saved_weights/parser-epoch-5.mdl
[...model loading stuff...]
Done!
----
buffer: ['i', 'eat']
stack:  ['<root>']
action: shift
----
buffer: ['eat']
stack:  ['<root>', 'i']
action: shift
----
buffer: []
stack:  ['<root>', 'i', 'eat']
action: left arc, <d>:compound:prt
----
buffer: []
stack:  ['<root>', 'eat']
action: right arc, <d>:det:predet
<root>
  |
 eat
  |
  i
```

In Task 4, you'll take a look at these outputs and determine whether they were correct.

Problem 4.1. Using one of your trained models, report the shift-reduce output for the sentence "The big dog ate my homework" and the parse tree

Problem 4.2. More than likely, the model has made a mistake somewhere. For the output, report what was the correct operation to make at each time step: shift, left-arc, right-arc (you do not need to worry about the specific dependency arc labels for this homework).

9 Optional Task

Homework 3 has lots of potential for exploration if you find parsing interesting or want to try building models. Here, we've listed a few different *fully optional* tasks you could try to help provide guidance. These are only for glory and will not change your score. Please please please make sure you finish the homework before trying any of these.

Optional Task 1: Measure the Effect of Pre-Trained Embeddings

In Task 2, your model learned word embeddings from scratch. However, there's plenty of rare words in the dataset which may not have useful embeddings. Another idea is to pre-train word embeddings from a large corpus and then use those during training. This leverages the massive corpus to learn the meanings so that your model can effectively make use of the information—even for words that are rare in training. But which corpus should we use? In Optional Task 1, we've conveniently pre-trained 50-dimensional vectors for you from two sources: all of Wikipedia and 1B words from Twitter.

Specifically, for Optional Task 1, you will update your code to allow providing a file containing word embedding in word2vec's binary format and use those embeddings in training *instead* of pretraining. You shouldn't update these vectors like you would do if you were learning from scratch, so you'll need to turn off the gradient descent for them. Part of Optional Task 1 is thinking about *why* you shouldn't change these vectors.

Finally, once you have the vectors loaded, you'll measure the performance just like you did in Task 2. This breaks down to the following steps:

Problem 3.1. Write code that loads in word vectors in word2vec's binary format (see Homework 2's code which has something like this). You'll need to convert these vectors into PyTorch's `Embedding` object to use.

Problem 3.2. Prevent the pretrained embeddings from being updated during gradient descent.

Problem 3.3. Write a few sentences about *why* we would turn off training. Be sure to describe what effect allowing the weights to change might have on future performance?[11]

Problem 3.4. Repeat the 5 epochs training like you did in Task 2 using the Twitter and Wikipedia embeddings and plot the performance of each on the development data (feel free to include the learned-embedding performance in this plot too). Write a few sentences describing what you see and why you think the performance is the way it is. Are you surprised?

10 Submission

Please upload the following to Canvas **as separate files** by the deadline:

1. a PDF (preferred) or .docx with your responses and plots for the questions above
2. your code for the parser[12]

[11] If you're struggling to write this part, try allowing their values to get updated and compare the performance difference between the development and test data. Feel free to report the scores!

[12] No zip files

Code should be submitted for any `.py` file you modified. Please upload your code and response-document separately. We reserve the right to run any code you submit; **code that does not run or produces substantially different outputs will receive a zero**.

11 Academic Honesty

Unless otherwise specified in an assignment all submitted work must be your own, original work. Any excerpts, statements, or phrases from the work of others must be clearly identified as a quotation, and a proper citation provided. Any violation of the University's policies on Academic and Professional Integrity may result in serious penalties, which might range from failing an assignment, to failing a course, to being expelled from the program. Violations of academic and professional integrity will be reported to Student Affairs. Consequences impacting assignment or course grades are determined by the faculty instructor; additional sanctions may be imposed.

References

Danqi Chen and Christopher Manning. A fast and accurate dependency parser using neural networks. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 740–750, 2014.

Sandra Kübler, Ryan McDonald, and Joakim Nivre. Dependency parsing. *Synthesis Lectures on Human Language Technologies*, 1(1):1–127, 2009.