Usage of Recurrent Neural Networks in Transition-based Dependency Parsing

Daniel Ziegler and Jerry Wu

1 Abstract

Transition-based parsing has the advantage of being very fast because it makes local decisions instead of trying to find a global optimum. Danqi and Manning[1] created a fast and accurate transition-based parser using a neural network for the oracle, which decides which of the several possible actions take at each timestep. However, transition-based parsing, including this neural parser, tends to make myopic decisions precisely because it makes said decisions locally. In this paper, we attempt to address this using Gated Recurrent Units (GRUs) to give the oracle more global information for decision making.[2]

2 Introduction

A key task in natural language processing is to parse sentences into dependency trees, where each word has an "arc" that points to its "parent", the word which it modifies or is otherwise subordinate to, and which also contains a label describing the relationship between the words. Often, this is done with transition-based parsers, which are state machines that proceed through input sentences, holding a stack of partially processed words and a buffer of to-be-processed words, as well as remembering the arcs already drawn, greedily taking actions like advancing to the next word or connecting two words with an arc, based on the decisions made by an "oracle". Transition-based parsers are consequently significantly faster at parsing sentences than other parsers (such as search-based parsers which don't use a greedy optimization) while achieving comparable, if slightly lower, accuracy. However, they are limited because the oracle for choosing the appropriate transition is myopic, focusing only on nearby elements of the sentence and words directly related to them.

Chen and Manning created a transition-based dependency parser using a neural-based oracle[3], which offered noticeable performance improvements and was very fast, but still was very myopic. We set out to reproduce their paper, and hoped to then improve on it by using a recurrent neural network to enable it to take advantage of information that is farther away in the sentence.

3 Transition oracle: Neural network

The primary motivation uf using a neural-based oracle is that more conventional oracles, such as logistic classifiers or SVMs, rely on a carefully chosen but large and sparse set of features. Using a neural network oracle allows the usage of embeddings, and the neural network will further compute a dense feature representation (the hidden layer) from the embeddings of selected words. This requires significantly less human effort, resolves some issues with incompleteness of the selected features, and greatly speeds up parsing speed because with conventional oracles, most of the parsing time is actually spent computing the features.

The possible actions are shift (move head of buffer to top of processing stack), left-arc (draw an arc from the top of the stack (parent) to the second word on the stack (child), and remove the second word from the stack), and right-arc (the opposite). For each of the arcing actions, there are 40 possibilities– the number of arc labels in the system, though a few of them are dummy arc labels used to represent nonexistent arcs for feature input purposes. This gives a total of 81 possible actions.

[1] Danqi Chen and Christopher D Manning. 2014. A Fast and Accurate Dependency Parser using Neural Networks. Proceedings of EMNLP 2014.

[2] Project code: `https://github.mit.edu/werryju/6867-proj`

[3] `http://cs.stanford.edu/~danqi/papers/emnlp2014.pdf`

3.1 Embeddings

Based on a previous paper, Chen and Manning selected 18 words to pass to the oracle out of the top few words on the stack and the queue, and out of the descendants of the top few words on the stack. The neural network was passed each of these words, as well as their part-of-speech tags (as generated by the Stanford Tagger program) and the arc labels between these words. In all cases, 50-dimensional embeddings were passed to the oracle. We found this rather confusing, because there are actually fewer than 50 different POS tags and arc labels (and consequently the embeddings would *increase* the dimensionality over the one-hot vector representation), so we instead decided to use 10-dimensional embeddings for most of our testing.

3.2 Architecture

The neural network used had a hidden layer of 200 units followed by a softmax output layer. For the hidden layer, Chen and Manning used a cubic activation function (i.e. $z = a^3$) instead of the conventional hyperbolic tangent, finding this to be a significant improvement. The theory behind this is that the expression includes the product of up to 3 terms in the input, allowing the neuron output to depend on conjunctions of multiple features in the input.

3.3 Regularization

Chen and Manning additionally used Frobenius-norm regularization with a coefficient of 10^{-8}, and also used dropout with a dropout rate of 0.5. It wasn't entirely clear from the paper, but we think they only used dropout on the hidden layer units; dropout on the input units would not have interacted well with the cubic activation function, especially since it explicitly was designed to exploit cross-terms between different inputs.

4 RNN: Gated recurrent unit

To address the myopicity of the transition-based parser, we decided to augment the information passed to the oracle with information about the sentence as a whole. This information would be an encoding of the sentence using a recurrent neural network; in particular, we decided to use the Gated Recurrent Unit (GRU) architecture, a simpler version of the Long-Short-Term-Memory (LSTM) architecture, a modification of the usual RNN designed explicitly to allow the RNN to retain information in the hidden layer over extended periods of time.

4.1 GRU Architecture

A GRU is first initialized with a hidden state of all zeroes. The hidden state is concatenated with the input vector to provide an input for two gates: the update gate and the reset gate. In both cases, the concatenated input is multiplied by a weight matrix with a bias added, then fed through a sigmoid (logistic function). The incoming hidden state is then multiplied componentwise by the reset gate output, then concatenated with the input vector to form a combined input, which is multiplied by a weight vector with a bias added, then fed through a hyperbolic tangent activation function. This is then multiplied componentwise by the update gate and added to the componentwise product of the previous hidden state and 1 minus the update gate to form the new input. The output of the cell is simply the hidden state.

So, where $h^{(i)}$ denotes the ith hidden state, with $h^{(0)} = \vec{0}$, $x^{(i)}$ is the ith input, one-indexed, W denoting weight matrices (subscript indicates which one), b denoting biases, $\otimes$ denoting componentwise multiplication, $\ominus$ denoting broadcasted componentwise subtraction:

$$r^{(i)} = \sigma(W_{rh}h^{(i-1)} + W_{rx}x^{(i)} + b_r)$$

$$u^{(i)} = \sigma(W_{uh}h^{(i-1)} + W_{ux}x^{(i)} + b_u)$$
$$z^{(i)} = \tanh(W_{zh}(h^{(i-1)} \otimes r^{(i)}) + W_{zx}x^{(i)} + b_z)$$
$$h^{(i)} = u^{(i)} \otimes z^{(i)} + (1 \ominus u^{(i)}) \otimes h^{(i-1)}$$

4.2 Encoder-decoder Architecture

The architecture for the encoder is based on LSTM networks for neural machine translation.[4] The encoder is a GRU cell which inputs the sentence one word at a time, repeatedly updating its hidden state. The decoder is a separate GRU and is initialized with hidden state equal to the last hidden state of the encoder, and then is called repeatedly, outputting the POS tag of each word in the sentence as a one-hot vector, though each time the decoder outputs the POS tag of a word, it first inputs the outputted POS tag of the previous word, represented as a one-hot vector. (For the first word, it inputs a vector of all zeroes.)

Once the encoder-decoder is trained, the encoder can be used to encode a sentence simply by running it over the words in the sentence. This produces a vector at each word in the sentence (the hidden state at that word) which can be fed to the transition-based parser oracle.

We used GRUs of hidden state width 200 for this. We decided to run the encoder over the sentence *backwards*, instead of forwards. This is because the transition-based parser's features contain information about what has already been parsed, and as a consequence the oracle already has a significant amount of information about the beginning of the sentence but little about the end. By running the encoder over the sentence backwards, the generated encoding about each word will encode information about the sentence after that point. (An alternative would be to run two encoders over the sentence, one forwards and one backwards, but we elected not to in order to avoid increasing the size of the feature set too much.)

The encoder additionally made use of 50-dimensional embeddings of the input words, but these embeddings are independent of the embeddings used by the parser.

4.2.1 Usage

The encoder is used to augment the features of the transition-based parser by first preprocessing the sentence, generating an encoding at each word of the sentence; the encoding of the word at the top of the stack is passed to the oracle in addition to all the previous features. This increases the dimensionality of the input to the oracle by 200, from 1200 to 1400.

4.2.2 Training

The encoder-decoder is trained end-to-end. We first process an entire sentence, which produces a sequence of probabilities for POS tags. The decoder, instead of inputting its previous prediction, instead inputs the true previous word. The total cross-entropy loss is then computed on the entire sentence, and the weights are updated using backpropogation through time.

Once the encoder is trained, the parser is trained on top of that, holding the encoder fixed. Theoretically, we could simply initialize the encoder in this fashion but continue to tweak it via backpropogation while training the parser, but this is more complex and we didn't have time to do this. Instead, we simply added the appropriate features to the parser-state snapshots fed to the oracle for training; see below for more details on how the oracle was trained.

[4] `http://arxiv.org/pdf/1406.1078v3.pdf`

5 Implementation

5.1 TensorFlow

We created an implementation of Chen and Manning's paper from scratch in Python. Because it seemed like an interesting tool to try, we based the implementation on TensorFlow, Google's newly released machine learning framework. TensorFlow models computations as graphs of operations, and then decides how to schedule the computations, potentially distributing them across CPU cores or GPUs. Most of the time, using TensorFlow in Python feels fairly similar to using numpy: you can call various functions to perform mathematical operations, matrix manipulations, and so forth. Instead of actually executing the computations right away, however, TensorFlow simply constructs the graph representation behind the scenes and waits for you to run feed it input data to run on. This has some convenient advantages: For instance, TensorFlow can examine the graph and automatically differentiate it — it's no longer necessary to manually compute gradients for gradient descent! Also, it's very simple to save and restore (partial) results of computations; TensorFlow has infrastructure to do so nearly automatically. On the flip side, TensorFlow's lazy nature means it's more difficult to explore interactively. It's not possible to simply write a few operations and see e.g. what the resulting matrix dimensions are, as it would be in Python. Instead, much of the time, it's necessary to create new tensors for each value of interest, including for the shapes of other tensors, and then call an evaluation method to evaluate the whole graph up until the desired tensor.

5.2 Dataset: English Penn Treebank

Chen and Manning tested their parser on the English Penn Treebank (PTB) as well as on the Chinese Penn Treebank. For our purposes, we decided to focus only on the English Penn Treebank dataset. It took some back-and-forth until we were able to have full access to the data, but once we did, we could use the same dataset that Chen and Manning used as well. Among other things, the PTB contains 43,948 English sentences with corresponding constituency trees, which we converted to dependency trees using the old Stanford Dependencies format (since this was used by Chen and Manning's paper) using a tool available as part of the Stanford Parser. These are the dependency trees that we tried to predict. For scoring, we decided to use LAS primarily, though we also evaluated to judge the percentage of trees which were entirely correct, including labels.

The input trees were parsed *without* punctuation; in particular, we simply recreated the input sentences based on the dependency trees, which did not include punctuation. This is because the raw text files contained several errors, and matching up the sentences to the trees proved difficult, so after some initial efforts were made, we decided to simply reconstruct the sentences.

Similar to Chen and Manning, we used sets 2 through 21 for training and 22 as the development set, and 23 as the test set, although we did little model selection, instead mostly relying on the hyperparameters set by Chen and Manning.

5.3 POS tagger

One of the most important inputs to the neural network is the part-of-speech (POS) annotations on the words. The words in the PTB dataset come labeled with parts-of-speech already, but apparently, the labels are not as accurate as they could be. Thus, we followed Chen and Manning once again, making an additional pass over the input data using the state-of-the art Stanford POS Tagger[5]. We were able to access it from Python by installing Python's NLTK natural language processing package[6], which has a module for calling into a downloaded Stanford POS Tagger Java JAR.

[5] `http://nlp.stanford.edu/software/tagger.shtml`
[6] `http://www.nltk.org/`

The POS tagger was used for preprocessing as a part of building the datasets used for training and testing. The sentences passed to the POS tagger were in the format of lists of words, without punctuation.

Except insofar as we used TensorFlow's libraries/tutorials and the tools available from Stanford NLP (the converter and the tagger), all our work was from scratch in Python. (For example, we implemented our own GRU unit instead of using TensoFlow's; most notably, however, we did *not* implement our own versions of the optimization algorithms we used.)

5.4 Shortest stack oracle

Since the neural network predicts parsing actions, not dependency trees themselves, it also must be trained on parsing actions. To generate parsing actions from our data, we implemented what's called a "shortest stack oracle", which figures out the sequence of transitions to create a particular tree with minimal use of the stack. Then, each transition becomes a training point for the neural network: The input is (a subset of) the state of the parser, and the output is the transition that was taken. This setup means that overall prediction accuracy falls off drastically: Incorrectly predicting an action means that most of the following arcs will probably be incorrect. If any arcs for a sentence are incorrect, the dependency tree for that sentence will be incorrect.

6 Results

6.1 Experimentation

Initially, our performance was poor as far as we were able to train the feed-forward neural net oracle (without augmentation by GRU encoding). In particular, despite training for very long, we were only able to achieve a LAS of about 72%. We tried a number of variations to fix our lackluster performance, despite none of them being mentioned by the paper. We were ultimately able to improve significantly on our initial results, to the extent where we came close to reproducing the paper, but we include details on our other attempts because we feel they still contain useful information.

6.1.1 Varying regularization constant

When we initially implemented the feed-forward neural network, we did so without dropout, and thus, we increased the regularization constant λ from 10^{-8} to 10^{-4}. Later on, we suspected that this may have been too extreme and caused us to seriously underfit, so we reduced it again. However, this did not result in the drastic improvement we had hoped for, a couple percent at most.

6.1.2 Dropout or not

We eventually implemented dropout, which proved to be quite simple using TensorFlow; however, our performance actually decreased using dropout. At the very least, it did not perform as well after a fixed number of training examples; we did not have the computation time to determine whether it would converge to an ultimately better result.

6.1.3 50 dimensions, not 10

We were somewhat puzzled by Chen and Manning's choice to use 50-dimensional word vectors for the POS tags and arc labels – there aren't even 50 different types of either of them, so those vectors are even bigger than one-hot vectors would be. As they explain, it still makes sense to use word vectors in order to exploit semantic similarities between the tags and arc labels. However, we decided to reduce the number of

dimensions for these to 10. When we got bad results, we wondered whether this change might have caused problems, so we attempted a training run where we put all the dimensions back to 50. Unsurprisingly, it trained much slower and did not do significantly better.

6.1.4 Different activation functions

Though Chen and Manning had explicitly claimed the cube activation function was better than tanh for this problem, we decided to see for ourselves; indeed, the performance was significantly worse.

We then tried the composition of hyperbolic tangent and cube; this would allow us to keep the benefits of having cross-terms between the different inputs while also restricting the output to $[-1, 1]$. Unfortunately, this did at best comparably to hyperbolic tangent. It's possible this is because the composition of tanh and x^3 is flat both close to 0 and far from 0, making it easy to get stuck during training.

6.1.5 Manual investigation of output

We inspected the output of our predictor to determine why it was doing so badly. First, we checked accuracy rates for the *actions* on the training set and a test set; we were achieving in the vicinity of 90% accuracy on the actions for both sets, but since arcs are not independent, that meant we achieved a noticeably lower accuracy on the arcs themselves, and much lower accuracy (about 10%) on the entire trees. This seemed reasonable, though part of the reason we suspected underfitting and reduced the regularization from $1e-4$ to $1e-8$ was because 90% was still significantly lower than we expected on the training set, and the training and test sets had approximately the same accuracy.

We then printed out a sample of the predicted actions, the true actions, and the predicted probabiities for both. The main result we noticed was that the majority of incorrect actions were predicted as "shift" instead of one of the 80 total possible arc-drawing actions. Furthermore, the predicted probability of the shift actions were above 50%, so it wasn't simply a case of similar arc labels splitting the probability, but regardless the oracle was significantly overpredicting the shift action, almost certainly because shift is the most common action. Furthermore, by getting the class of action wrong, the oracle would severely hurt its chances to get arcs correct later on parsing the same sentence, because the produced result must always be a tree.

6.1.6 New loss function

To combat this, we decided to make the loss function encourage the oracle to focus on selecting the correct type of action first, then choosing the correct type of arc. Our new loss function is:

$$\lambda \text{cross\_entropy}(\hat{y}, y) + (1-\lambda)\text{cross\_entropy}(acts(\hat{y}), acts(y))$$

where λ is a scalar parameter, $\hat{y}$ is the prediction, y is the true value, and $acts(p)$ inputs an action probability vector (dimension 81) and returns the corresponding action class probability vector (dimension 3).

Unfortunately, this didn't work either.

6.1.7 Other Analysis

We also took one of our trained models and examined the various parameters. We found that the weights for words, POS tags, and labels were all about the same, but the embeddings for the words were about an order of magnitude smaller than the embeddings for POS tags and labels– so their importance probably corresponds to that as well. This seemed reasonable, however, because indeed the majority of the syntactical information on a word should be based on its part of speech, and we didn't know if this was actually usable information.

We also checked the quality of our POS embeddings by computing the pairwise dot product between different POS tags; nominally, we should have used cosine similarity (which would just be the normalized version) but the embeddings were approximately the same magnitude so this was unimportant. Indeed, similar POS tags had similar embeddings (for example, the different variants on noun), so that part of our code probably worked, and we did not find a way to improve our result through this investigation.

6.1.8 Adam optimizer

We initially used mini-batched Adagrad, as they noted in the paper; however, we decided to switch to the Adam optimizer since it was purported to be better. Our first attempt did not work, but subsequent attempts achieved useful results.

6.2 Testing the Encoding-Augmented Parser

We came close to reproducing the results in the paper, though we were not entirely successful. Our preliminary tests, where we only trained our parsers for 4 epochs (for a total of approximately 6.8 million training examples), achieved a LAS of approximately 0.84 on the dev set.

This, we decided, was sufficient to test against. Our final tests ran each parser for 10 epochs, for 17190 mini-batches of 1000 examples each; we didn't have time to instead run until some convergence condition or another. In all cases, we used a hidden layer with 200 nodes using the cubic activation function, a regularization constant of $1e-8$, 50-dimensional embeddings for words, and 10-dimensional embeddings for POS tags and arc labels. Optimization was performed using the Adam optimization algorithm with $\epsilon = 1e-6$ for numerical stability. The stepsize followed $(0.03 \cdot 200)/(200 + n^{0.61})$, where n is the number of epochs which have elapsed, as an integer. The loss function used was the cross-entropy loss function.

Based on our earlier experimentation training to 4 epochs, we originally decided not to use dropout for testing, but noticed that the loss function had significantly decreased over the course of training but the validation error had not, so we suspected overfitting and consequently additionally ran tests using dropout.

6.3 Data

Note that all data provided are a single sample; we did not have time to train for multiple iterations and take the average result.

Results from our initial experimention is given in Figures 1 and 2. These are trained only on sections 2 through 5 of the Treebank for faster runtime (due to memory limitations, we would not be able to perform as much precomputation on the entire training set), run for 10 epochs.

Next, the results from preliminary testing after we started using mini-batched Adam are in Figures 3 and 4. All training here and in the next section is performed on sections 2 through 21 of the English Penn Treebank, drawn in batches of 1000 without replacement.

The results from our final round of testing are in Figures 5 and 6. Despite the relatively high performance of tanh in the previous round of training, we elected to proceed with cubic activation functions simply because the original paper did as well.

7 Analysis

First, we would like to note that it's evident from the training error graphs that the networks have *not* completely converged, and this probably one reason our LAS scores did not match up to those given by Chen and Manning, but we did not have time to train the oracles to convergence. This is especially evident

Figure 1: Loss during training, which was conducted with mini-batched AdaGrad. The reported loss is on the batch being trained on, before the gradient step. `default` is our main implementation with 10-dimensional embeddings for POS tags and arc lables, with a cubic activation function, no dropout, and cross-entropy loss. `dropout` is with dropout on the hidden layer only; dropping out both layers is worse. `exper` uses the experimental activation function (composition of tanh and cube). `largedim` uses 50-dimensional embeddings everywhere. `encod` is with the third implementation of the encoder; the loss before about the 2400th iteration is very high due to bad training parameters and has been omitted for readability; in fact we're surprised it managed to recover. `adjloss` is with the modified loss function. Not shown is `hireg` with a large regularization constant, since that just has a very high loss due to said regularization constant.

name	train	test	arc	tree
default	0.808705536	0.801493526	0.532384188	0.014705882
dropout	0.670354143	0.668070011	0.27447015	0.001176471
exper	0.676487048	0.674778456	0.291727336	0
laredim	0.857649894	0.849073793	0.619721728	0.044705882
tanh	0.687494382	0.684316594	0.296796635	0
hireg	0.817964339	0.807332358	0.548401014	0.027647059
adjloss	0.822015247	0.813612898	0.572992504	0.024705882

Figure 2: Names correspond to those in Fig. 1. `train` is the accuracy on the actions in the training set; `dev` is the accuracy on the actions in the development set; `LAS` is the accuracy on the predicted arcs when used to parse the development set; `tree` is the accuracy on the whole tree when used to parse the development set.

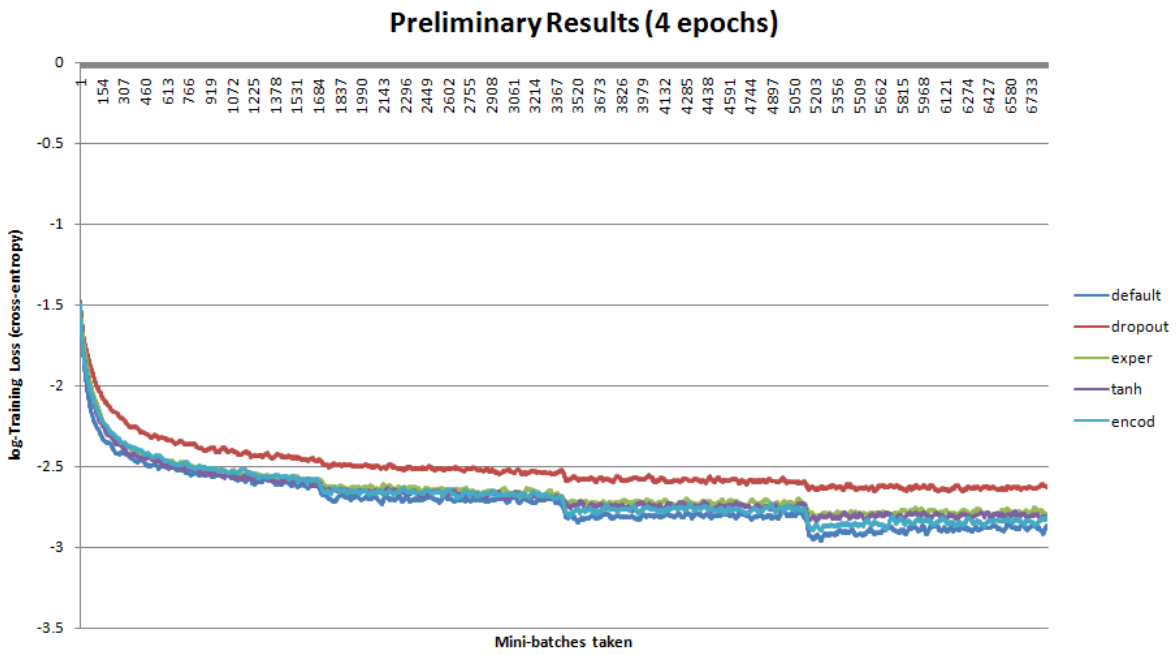

Figure 3: Each datapoint represents the log-loss during training, averaged over 30 iterations for readability. (Yes, the loss function is already the negative log-liklihood of the gold tags; we take the log here strictly for readability reasons.) We did not use `adjloss` or `largedim` in this phase.

name	default	dropout	exper	tanh	hireg	encod
LAS	0.84331014	0.83845656	0.841341746	0.847462654	0.744377932	0.843795502
tree	0.23705882	0.242352941	0.243529412	0.270588235	0.106470588	0.268823529

Figure 4: These are the results from evaluating on the dev set.

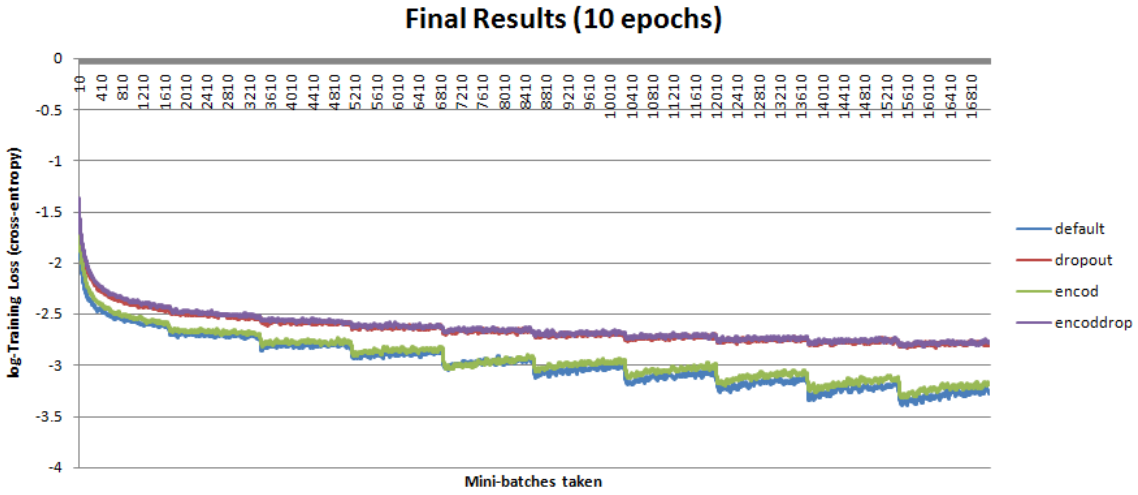

Figure 5: Here, we stopped using most of our more experimental parser setups. `encoddrop` is the parser with GRU encoding augmentation, but also with dropout.

name	LAS (dev)	tree(dev)	LAS (test)	tree (test)	wor	tag	arc	encod
default	0.8423394	0.2582352	0.848171	0.26365894	2.2	1.6	1	–
dropout	0.8503208	0.2670588	0.852024	0.273178808	2.3	1.8	1	–
encod	0.8428517	0.2511764	0.847941	0.26365894	2.6	1.9	1	2.2
encoddrop	0.8522083	0.2694117	0.852867	0.274834437	2.1	1.6	1	1

Figure 6: `wor`, `tag`, `arc`, and `encode` denote the *relative* importance of each section of the feature space. To compute these, we took a sample of 50 training points (snapshots of the parser state with the appropriate action), parsed each input into the 1200-length or 1400-length vector with embeddings of the input words, tags, etc., and multiplied each section by the appropriate section of the input weight matrix. This produces four vectors, which when added together produce the activations of the hidden layer. We then took the RMS values of the components.

from the fact that training error seems to drop significantly at the beginning of each epoch, which is perhaps indicative that the embeddings are still improving.

Likewise, the encoder was not optimized very highly both in terms of hyperparameters and in terms of training length; just enough to be able to decode with some accuracy (approximately 75%) the POS tags of the output words. I notice, incidentally, that the input to the encoder does *not* include the POS tags, since we wanted to include information about the words and thought that inputting the POS tags would just cause the word input to be disregarded, though in retrospect this may have caused the encoding to not have been quite as good as they could have been; we didn't have too much time to experiment with this.

Ultimately, the results are inconclusive. I note that indeed it seems that there is overfitting after training for about 10 epochs, so use of dropout is beneficial. In this case, the neural parser augmented by the encoding seems to do marginally better than the original neural parser, though it's possible this is simply due to luck, so from this it's unclear that the inclusion of the encodings is helpful.

On the other hand, when analyzing the importance of the various factors to the input of the hidden layer, it's clear that though the oracle prefers to make use of the word and POS embeddings, it in no way disregards the information given by the encoding, so it seems unlikely said encodings are entirely unhelpful; but since the inclusion of the encodings does serve to increase the dimensionality of the feature space, it's unclear that including this extra feature is more of a benefit than including, say, information about additional words in the context. It seems further exploration is needed, ideally with more computational power. For example, it's also unclear what the relevance performance would be in the limit where the oracles have converged.

We never tested the speed of the parser with GRU-encoder augmentation. Realistically, the expected usage is you first preprocess the entire sentence using the encoder, then feed appropriate values to the oracle, but we didn't have time to set up testing this way, especially because the person running the training and tests had limited RAM; this is also the main reason training took so long, because we did not have the memory to store precomputation for the entire corpus, and due to some inefficiencies we were recomputing the encoding for the entire sentence, once for each *snapshot* instead of once for each tree. Theoretically, though, expect the parser to be at most a couple times slower with GRU-encoder augmentation, since the runtime of the encoder for each input should be comparable to the runtime of the neural oracle for each action.

8 Division of Labor

As planned, since Jerry used this project for both 6.806 and 6.867, he took over more of the work than Daniel. He wrote the GRU, shortest-stack oracle, and an initial TensorFlow feed-forward NN, as well as some miscellaneous tasks. Daniel wrote the Stanford POS tagger integration and finished up the feed-forward NN, integrating the POS tagger and word vectors and building the actual training data using the shortest-stack oracle. During the optimization (or rather, salvage) process, both of us spent a lot of time thinking through the code and trying out various alternatives.

9 Works Cited

Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser using Neural Networks. In EMNLP 2014. http://cs.stanford.edu/ danqi/papers/emnlp2014.pdf

Cho et al. 2014. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In CoRR 2014. http://arxiv.org/pdf/1406.1078v3.pdf

Stanford Natural Language Processing Group. 2015. Stanford Log-linear Part-Of-Speech Taggers. http://nlp.stanford.edu/software/tagger.shtml

Joakim Nivre. 2003. An efficient algorithm for projective dependency parsing. In Proceedings of the Eighth International Workshop on Parsing Technologies (IWPT), pages 149160, Nancy, France.

NLTK Project. 2015. Natural Language Toolkit - NLTK 3.0 documentation. http://www.nltk.org/