CS5740: Project X
`https://github.com/cornell-cs5740-20sp/final-ks20`

Kushal Singh
ks2377

20 April 2020

1 Introduction (6pt)

In this paper, we propose a model for name-entity recognition (NER) of Twitter tweets in order to identify sub-spans of words that represent named entities. The training and development data includes a token on every line, and is annotated with its respective label: B for beginning of named entity, I for continuation of a named entity, or O for a token that is not part of a named entity. Each named entity starts with B and all other tokens of the named entity, except the first one, are annotated with I. Sentences are separated by a new line.

For implementing aforementioned NER task on colloquial tweets, I began by perusing existing abstracts [1] in the field [2]. These papers demonstrated state-of-the-art performance in NER by virtue of bidirectional LSTM with a CRF model.

I used the training data to train the neural network model. Primary experiments included testing for optimal embedding dimensions, hidden layers, and number of training epochs. Cursory results of these experiments included an embedding dimension size of 100, number of hidden layers at 300, and 30 training epochs. The model was evaluated by testing development data prediction against the ground truth, with the F1 score generated from precision and recall. Final score achieved on leaderboard was **0.5321**.

2 Task (5pt)

As discussed above, we propose a model for name-entity recognition (NER) of Twitter tweets in order to identify sub-spans of words that represent named entities. To that extent, we define a Bi-LSTM CRF model, which takes as input a word-level and character-level embedding and yields as the output a matrix P of the emission probability of each word corresponding to each label, where $P_{a,b}$ is a projection probability of word w_a to tag_b. For CRF, we assume a transition matrix A of tag_a to tag_b with transition probability.

3 Data (4pt)

For data preprocessing, I generated sentences, labels (i.e. B, I, O), and tags (i.e. parts of speech, such as NN, VT) for the training, development, and test data. I also vectorized words and characters by creating a *word_to_id* dictionary that mapped words to integers and a *char_to_id* dictionary that mapped characters to integers, from the training set. Finally, I created a *label_to_id* dictionary that vectorized the tagged labels as follows: {START_TAG: 0, 'O': 1, 'B': 2, 'I': 3, STOP_TAG:4}.

Table 1: Data Statistics

Metric	Value
Word Vocab Size (Training)	9069
Character Vocab Size (Training)	67
Number of Sentences (Training)	2394
Word Vocab Size (Dev)	4572
Character Vocab Size (Dev)	261
Number of Sentences (Dev)	959
Word Vocab Size (Test)	9315
Character Vocab Size (Test)	378
Number of Sentences (Test)	2377
Total Parts of Speech	46

I created a list of dictionaries for each sentence. Each of these dictionaries contained the following keys: 'caps', 'chars', 'words', 'tags', 'str_words'. The 'caps' key mapped to a list of integers, where the ith value in the list indicated whether the ith word in the sentence was either: all lowercase (0), all uppercase (1), only first letter uppercase (2), only one letter except the first uppercase (3). The 'chars' key mapped to a list of lists, where the jth sublist contained a list of integers representing the vectorized characters of the jth word in a sentence based on the *char_to_id* dictionary discussed in the paragraph above. The 'words' key mapped to a list of integers, where the kth integer represented the vectorized version of the kth word in a sentence, based on the *word_to_id* dictionary discussed in the paragraph above. The 'tags' key mapped to a list of integers, where the lth integer represented the vectorized version of the tagged label for the lth word, as discussed in the *label_to_id* dictionary in the paragraph above. The 'str_words' key mapped to a list of strings, where the mth string represented the mth word in a sentence.

4 Model (25pt)

The LSTM model comprises of two components: word representation and bidirectional LSTM. The word embedding is represented as either raw text words or character-based encodings of the raw text words.

For generating the word-level embeddings from the training data, I built the *word_to_id* dictionary which mapped words to unique integers. This dictionary also contained a 'RARE' key to account for words that did not appear in the training set, but appeared in the dev set and/or test set (i.e. UNKNOWN words).

[1] https://arxiv.org/abs/1508.01991v1
[2] https://www.aclweb.org/anthology/Y18-1061.pdf

Each word is represented as a 100-dimensional vector which is subsequently trained in LSTM. Based on experimentation which will be discussed below, word embeddings in 100 dimensions yielded better results than word embeddings in 25, 50, 200, and 300 dimensions. To initialize the word embedding, we used the *glove.6B.100d.txt* pre-trained dataset [3].

For generating the character-level word embeddings from the training data, I built the *char_to_id* dictionary which mapped characters to unique integers. In the end, each word is represented as a list of z integers, where z is the length of the word. To create the input vector, we applied a small Bi-LSTM to handle the fact that words and, thus, the character-level embeddings have different lengths. Each character is represented as a 100-dimensional vector and is input into the Bi-LSTM to obtain the last step's output as input to the bidirectional LSTM.

The Bidirectional LSTM for Twitter NER is shown in Figure 1 below, and is motivated by the model presented in this abstract [4]. For each sentence, we generate word vector and character-based representations. Similar word representations are associated to the same words, are concatenated, and sequentially fed into bi-LSTM to learn word-based contexts within a sentence. At the output layer, I optimize the log-likelihood of labeling the whole sentence correctly by modeling interactions between two successive labels using Viterbi algorithm.

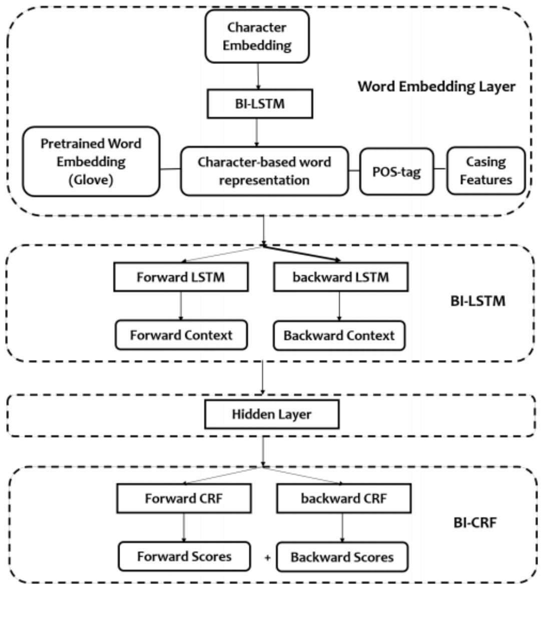


Figure 1: Architecture of the network

5 Learning (10pt)

In Bi-LSTM CRF, there are two kinds of potentials: emission and transition. The output of Bi-LSTM layer is a matrix P of the emission probability of each word corresponding to each label, where $P_{i,j}$ is a projection probability of word w_i to tag_j. For CRF, we assume a transition matrix A of tag_i to tag_j with transition probability. Let y be a tag sequence and X an input sequence of words.

$$S(X,y) = \sum_{i=0}^{n} A_{y_i,y_{i+1}} + \sum_{i=1}^{n} P_{i,y_i}$$

Given a word sequence X, the probability of tag sequence y can be written with Softmax:

$$\Pr(y|X) = \frac{e^{s(X,y)}}{\sum_{y' \in (Y_X)} e^{s(X,y)}}$$

The objective is to maximize $P(y|X)$, and the loss function is defined as $-log(P(y|X))$, with SGD as the optimizer.

The *word_to_id* dictionary discussed above contains a 'RARE' key to account for words that did not appear in the training set, but appeared in the dev set and/or test set (i.e. UNKNOWN words). For instance, there are a lot of emojis that appear in the test set and not the training set. These emojis were treated as unknowns and did not make a significant impact on the model, since they do not play a big role in determining named entities. The 'RARE' key is sufficient in this case—the need for chunking unknown words into different categories (e.g. EMOJI) is unnecessary because words also have their respective character-based embeddings.

6 Implementation Details (3pt)

The LSTM model uses stochastic gradient descent (SGD) as the optimization algorithm. It comprised of the following hyperparameters and their values:

```
1 word_embed_dim = 100
2 word_lstm_dim = 100
3 char_embed_dim = 100
4 char_lstm_dim = 100
5 hidden_dim = 300
6 pre_embed_file = 'glove.6B.100d.txt'
7 learning_rate = 0.05
8 momentum = 0.9
9 epochs = 30
```

During the results and experimentation phase, the following variables were fine-tuned to optimize performance: embedding size (for both character-level and word-level representations), hidden layer size, learning rate, number of epochs, and the GloVe pre-trained embeddings file.

7 Experimental Setup (4pt)

The variables fine-tuned in the paragraph above were evaluated against the F1 score. While fine-tuning a particular variable, all other variables were held constant to prevent result obfuscation.

The evaluation metric is the F1 score which is calculated against the precision and recall. Specifically, the precision is calculated as $\frac{TP}{TP \ + FP}$, where TP is True Positives and FP is False Positives. Recall is calculated as $\frac{TP}{TP \ + FN}$, where FN is False Negatives. The F1 score is calculated as $\frac{2 \times Precision \times Recall}{Precision \ + Recall}$.

[3]https://nlp.stanford.edu/projects/glove/
[4]https://www.aclweb.org/anthology/Y18-1061.pdf

8 Results

Test Results (3pt) We generated this test score using our procedure for handling unknown words described above. The hyper-paramater values were $d = 100$ for character and word embeddings, $h = 300$ for hidden layers, and $epochs = 30$.

Table 2: Leaderboard Performance

F1	Precision	Recall
0.5321	0.6557	0.44771

Development Results (7pt) The F1 score generated in Tables 3 and 4 were trained on the dev dataset for 5 epochs. Based on experimentation, the Wiki + Gigaword dataset appeared to outperform the Twitter dataset, regardless of embedding dimension.

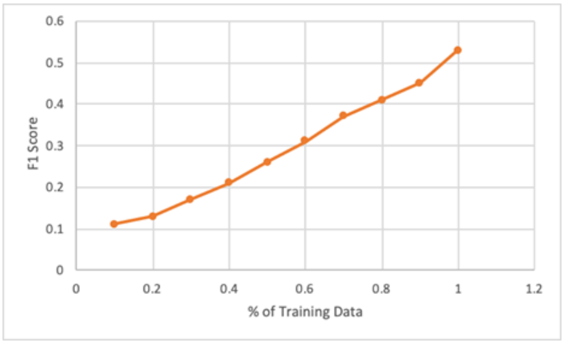
Table 3: Different Pre-trained Embeddings

GloVe File	**F1 Score**
Wikipedia 2014 + Gigaword 5 6B tokens, 400K vocab, 50 dimensions	0.2834
Wikipedia 2014 + Gigaword 5 6B tokens, 400K vocab, 100 dimensions	0.3658
Wikipedia 2014 + Gigaword 5 6B tokens, 400K vocab, 200 dimensions	0.2773
Twitter 2B tweets, 27B tokens, 25 dimensions	0.2329
Twitter 2B tweets, 27B tokens, 50 dimensions	0.2542
Twitter 2B tweets, 27B tokens, 100 dimensions	0.2901

Table 4: Hidden Layer Performance

Hidden Layers	**Embed Dims**	**F1 Score**
200	100	0.2951
300	100	0.3658
400	100	0.3458

Learning Curves (4pt) From the graph below, we can see that as the percentage of training data increased, the F1 score increased proportionally.



Speed Analysis (4pt) Table 5 below shows the total time the model takes to go through an entire epoch, the total time per sentence, and the total time per token. Based on these calculations, we can estimate the algorithm speed to be approximately 4 mins / epoch * 30 epochs = 120 mins = 2 hours. The model was run on Google Colab with 12.72 GB RAM.

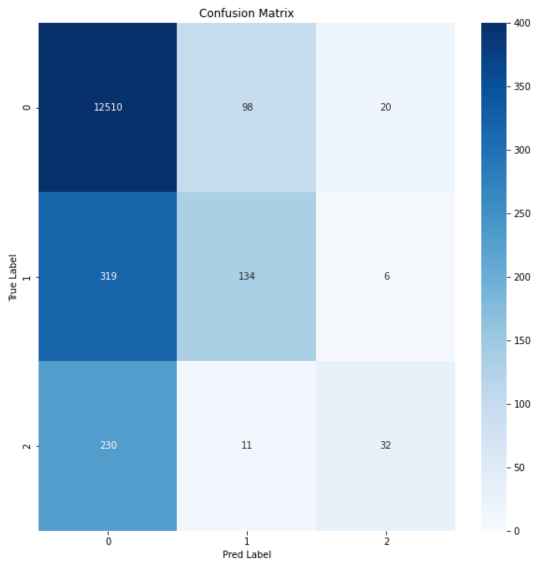
Table 5: Speed Analysis

Total Time	**Time / Sentence**	**Time / Token**
3:58	10.06 it/s	1.2575 tok/s
4:25	9.03 it/s	1.1288 tok/s
4:30	8.86 it/s	1.1075 tok/s

9 Analysis

Error Analysis (7pt) Looking at the first example in the dev set: ['rt', '@aleynastam', ':', 'i', "can't", 'wait', 'until', 'cedar', 'point', 'opens'], we can see that one category of mis-classifications occurs when named entities comprise words that also appear in regular expressions. For instance, the words 'cedar' and 'point' juxtaposed refer to the named entity of a theme park, but are more commonly seen as individual terms, which is what the network learns. Another category of mis-classifications stemmed from homonoyms, such as 'turkey' which may refer to the country, but which the network recognizes as food.

Confusion Matrix (5pt) Looking at the confusion matrix below, we can see that most tweets contain tokens that are not part of a named entity (indicated by the 12510 O tags).



10 Conclusion (3pt)

The largest improvement in NER task performance stemmed from CRF. Furthermore, character-level embeddings enriched contextual knowledge gleaned during neural network training phase, and also provided substantial improvements. Fine-tuning hyper-parameters (e.g. number of hidden layers, embedding dimensions, learning, rate, number of epochs) proved crucial in making incremental performance progress. For instance, fixing the embedding dimensions at 100 increased the F1 score by nearly 0.1, fixing the number of hidden layers at 300 increased the F1 score by nearly 0.15, and training for around 30 epochs seemed to be the sweet spot between underfitting and overfitting the model.

CS5740: Assignment 1 https://github.com/cornell-cs5740-20sp/assignment-1-fixed-kushandjay

Jack Rossi
jdr342

Kushal Singh
ks2377

1 Introduction (5pt)

This paper explores the text classification problem using Perceptron learning, Maximum Entropy Learning, and multi-layer Perceptron learning. The data comes from two corpora, news documents and proper name samples.

We implemented and tested the performance of several featurizations of the data. For news documents, we used bag of words, bigrams, and document length, and for proper names we used letter-grams. For news group classification, we found that the Perceptron with bag-of-words performed best. Similarly, the Perceptron was best able to classify proper names, this time using letter tri-grams.

2 Features (20pt)

Bag of words and n-grams were expected to be very sparse features, so we represented features for a given example as a dictionary, where entries represented non-zero features, and everything else was zero.

Word n-grams We featurized the newsgroup dataset using word n-grams with n values of 1 and 2. the word n-gram with n = 1 is the same as bag of words, but sometimes our 'words' were sets of special characters separated by spaces. A document and vectorization example from the misc.forsale group follows.

> "From: lorne@sun.com (Lorne R. Johnson - Sun IC Region SE) Subject: WARRIORS TICKETS FOR SALE Organization: Sun Microsystems, Inc. Lines: 22 Distribution: ca Reply-To: lorne@sun.com NNTP-Posting-Host: normajean.west.sun.com **************************** * WARRIORS TICKETS FOR SALE * **************************** I have 2 tickets that I can't use (Last pair this year). Section 109, Row P, Seats 8 9 DAY DATE OPPONENT TIME — —- ——– —- WED 4/21 Sacremento 7:30 Price: 45.00 = MY COST Call or email if you are interested in these tickets. Lorne Johnson lorne@sun.com (408) 562-6003 "

{'From:' : 1, 'lorne@sun.com' : 1, '(Lorne' : 1, ...}

Letter n-grams We featurized the proper name dataset using letter n-grams of size 1 through 3. These letter n-grams consisted of adjacent letters and/or special characters. An example from the proper names corpus and the 2-gram letter vectorization follow:

> Two Days, Nine Lives

{ 'tw' : 1, 'wo' : 1, 'o ' : 1, ' d' : 1, ...}

Document length One of the features we tried was the length of the proper name. However, this feature appeared to have detrimental results on our accuracy. One possible hypothesis is that a lot of proper names have the same word length, but different n-grams. Therefore, the likelihood that those words are similar increases, due to the match in the word length feature. We can corroborate this assumption by looking at the corpus of proper names, where the key is the length of the word, and the value is the number of proper names with word length equal to that key.

('Corpus: ', 1: 3, 2: 16, 3: 91, 4: 232, 5: 517, 6: 954, 7: 1345, 8: 1506, 9: 1490, 10: 1419, 11: 1389, 12: 1367, 13: 1356, 14: 1251, 15: 1154, 16: 1010, 17: 904, 18: 788, 19: 666, 20: 594, 21: 536, 22: 445, 23: 428, 24: 388, 25: 333, 26: 289, 27: 281, 28: 281, 29: 234, 30: 205, 31: 158, 32: 146, 33: 147, 34: 132, 35: 106, 36: 112, 37: 87, 38: 76, 39: 78, 40: 59, 41: 57, 42: 51, 43: 38, 44: 47, 45:

Dataset	Count
Newsgroups	
Train	9051
Test	2263
Propernames	
Train	23121
Test	2894

Table 1: Size of training and development sets for Newsgroups and Propernames.

37, 46: 28, 47: 29, 48: 29, 49: 30, 50: 37, 51: 24, 52: 19, 53: 11, 54: 11, 55: 14, 56: 10, 57: 7, 58: 14, 59: 5, 60: 8, 61: 8, 62: 2, 63: 4, 64: 1, 65: 4, 66: 3, 67: 3, 68: 2, 69: 4, 70: 3, 74: 1, 76: 2, 78: 1, 79: 1, 81: 1, 83: 1, 88: 1.

It is apparent that there are a lot of proper names with word length $\in [7, 16]$.

3 Experimental Setup

Data (5pt) Table 1 shows the high-level statistics for the datasets. The Proper Name and Newsgroup datasets are both split into three subsets: training, development and test. The training, dev, and test for Proper Names have 23121, 2893, 2862 samples, respectively. Each subset contains proper names belonging to one of five different categories; person, place, movie, drug, company. The training, dev, and test for Newsgroup have 9053, 2263, 7530 samples, respectively. Each subset contains articles belonging to one of six different themes: comp, rec, sci, politics, religion, misc. These can be further categorized into 20 categories.

Data Preprocessing (5pt) News data was tokenized and lower cased. Special characters were not removed, because we felt they would have different distributions between news groups. For example, strings of special characters might indicate page style that was characteristic of a particular news group. Tokenization split strings on spaces.

Proper name data was also lower cased but it was not tokenized. Special characters were again retained.

Perceptron Implementation Details (3pt) Our Perceptron model maintained a weight vector for each class in the text classification problem. The weight vectors were represented as dictionaries for consistency with the feature representation. Training of the model requires specifying the number of epochs, training data, and development data. After each epoch of training, the model calculated accuracy on both the training and development data. This information allowed us to retrain using the smallest number of epochs that provided high performance on the development set.

MaxEnt Implementation Details (3pt) For our MaxEnt model, we used SciPy's optimization function: *scipy.optimize.*$fmin\_l\_bfgs\_b$. We passed in $self.fn\_log\_likelihood$ as the loss function, which uses softmax to calculate the probabilities of a particular label, given the dot product between the query instance and the weights dictionary. We also passed in $self.fn\_log\_likelihood$ as the derivative with respect to the weights of the loss function. The final result was an estimation of the position of the optima. With regards to hyperparamters, we set the *maxiter* parameter in the optimization function above equal to 20, as that yielded the best results from experimentation. We also had to set the weights dictionary to be $= (4 * (len(corpus) + 1) * dimensions)$, since this was the minimum size that ensured no indices were out of bounds.

MLP Implementation Details (10pt) In terms of architecture for the computation graph, we used 2 total layers (250 dimensions for each), with tanh as the activation function in the first layer and softmax in the final output layer, ultimately choosing the label that yielded the highest probability from softmax. Through experimentation, we found that tanh for activation in the first layer yielded better results than sinh and cosh. The optimizer we used was the AdamTrainer, as this yielded better accuracy results compared to the SimpleSGDTrainer, MomentumSGDTrainer, and AdagradTrainer. Our learning rate was set to 0.001, which is the default rate for AdamTrainer. Our stopping criteria is determined by the number of epochs through the training data. Through experimentation, we concluded that the loss and accuracy appeared to stagnate after around 8-10 epochs. We also used a lookup variable that maps numbers to vectors, specifically for word embedding. We decided to have 7000 rows, each with 10 dimensions, as this ensured that we did not have any lookups out of bounds.

Model	Test Acc
Newsgroups	
Perceptron BoW	0.6013
MLP, BoW, Alphanum	0.4782
MaxEnt, BoW, Alphanum	0.0521
Propernames	
Perceptron Trigrams	0.7795
MaxEnt Bigrams	0.6359
MLP Bigrams	0.6512

Table 2: Test accuracy for final chosen models. Legend: BoW = Bag of Words; Alphanum = text is reduced to alphanumeric characters only.

Model	Dev Acc
Newsgroups	
Perceptron BoW	0.7101
Perceptron Bigrams	0.6893
Perceptron, BoW, Alphanum	0.7021
Perceptron, Bigrams, Alphanum	0.6999
Propernames	
Perceptron Unigrams	0.6013
Perceptron Bigrams	0.5499
Perceptron Trigrams	0.7912

Table 3: Accuracy of Perceptron ablations on both datasets.

4 Results and Analysis

4.1 Proper Name Classification (11pt)

Table 2 shows the test performance of the selected Propernames classifiers. The Perceptron performed best with accuracy of 0.7795, followed by MaxEnt with accuracy of 0.6359, and finally MLP with accuracy 0.6512.

In testing various alternatives, we found that trigrams worked best for the Perceptron, and then Bigrams worked best for both MaxEnt and MLP. And example of the ablation analysis is presented in Table 3. We also found that adding the length of the proper name as an additional feature did not improve the dev accuracy.

From our results, we noticed that the labels for people and movies were often classified incorrectly. Since we used n-Grams as our feature, this observation makes sense. From the training data, we see that a lot of movies have places in their titles: *Going to California, Adieu Babylone, California Girls, Loin de Syracuse, Hollywoodrymlingar.*

4.2 Newsgroup Classification (11pt)

Table 2 shows the test accuracy for the best fit newsgroups models. The perceptron model with bag of words featurization led in accuracy with a score of 0.6013. Next up was MLP, with an accuracy score of 0.4782. Last was MLP, with an accuracy score of just 0.0521. This last test score is hypothesized to result from a bug in the MaxEnt model when running newsgroup features. Table 3 shows the ablations we tested for newsgroup classification. We ended up using bag of words with all characters included because it provided the highest accuracy on the development set, a score of 0.601.

5 Conclusion (3pt)

Overall, word and character $n - grams$ seemed to be a very good baseline feature, regardless of the model or dataset. The Perceptron algorithm yielded the best results on the test set, compared to Max-Ent and MLP (Multi-layer Perceptron). However, MLP yielded better results on the training and dev set, compared to Perceptron and Max-Ent. For the future, identifying and experimenting with additional, nuanced features, such as ratio of vowels to consonants, ratio of consonants to vowels, number of syllables could result in better accuracy.

CS5740: Assignment 2 https://github.com/cornell-cs5740-20sp/assignment-2-nlpeace

Jack Rossi
jdr342

Kushal Singh
ks2377

1 Introduction (5pt)

Word embeddings are an approach to featurizing text that seeks to embed a sparse word space into a dense vector. In this paper we seek to derive these vectors for a selection of relevant words. We will use a neural network approximating the skip-gram model with negative sampling to derive these vectors, and evaluate our embeddings against human-derived similarity rankings.

We experimented with corpus sizes, optimization algorithms, objective functions and many other model hyperparameters. Ultimately, using batch sizes of 50,000 words, window-based contexts, the an extended corpus of 3 million sentences, and binary cross entropy loss performed best, providing development correlation of 0.48, and test correlation of 0.160.

2 Model (10pt)

Formally, this model uses embeddings to calculate the probability that a given word, context (w, c) pair is found in the training corpus. We used window-based contexts. Thus, for a sentence consisting of words $x_1, ..., x_n$, we defined for each x_i the window around it, $x_{i-d}, ..., x_i, ..., x_{i+d}$ where the window size is $2d + 1$. We seek to find the embeddings, θ, such that the following objective is maximized.

$$argmax_\theta \prod p(D = 1|w, c) \prod p(D = 0|w, c) \quad (1)$$

Where D is an indicator of whether or not the pair was found in the corpus. In practice, this model computes the dot product of embeddings of pairs, subsequently using the sigmoid function to scale the output to probabilities. The embeddings are then updated by backpropagating errors. We tried two loss functions, the first of which is Binary Cross Entropy, given in equation 2.

$$BCE = -(ylog(f(x)) + (1-y)log(1-f(x))) \quad (2)$$

Where x is a specific (w, c) pair, $f(x)$ is the network output for example x, and y is class identity of example x, $y \epsilon \{0, 1\}$. Another loss function we tried was Mean Squared Error, given in equation 3.

$$MSE = \frac{1}{n} sqrt(\sum_{i=1}^{n} (y_i - f(x_i))^2) \quad (3)$$

Where x, $f(x)$, and y are given by the previous equation 2, and n is the number of training examples.

3 Experiments (12pt)

In experiment 1 we explored basic choices in the structure of the neural network, including the dimensionality of the word embeddings, the loss function used, and the size of the data used for training. We tried two loss functions as described in the previous section, two corpus sizes (1000, 10,000), and three embedding dimensionalities (10, 50, 100). These options were permuted together to give twelve unique results. In experiment 2 we tested additional hyperparameters of the model and training process, including the optimization algorithm used, the learning rate, and the number of epochs for which the neural network was trained. We used two optimization algorithms, Stochastic Gradient Descent (SGD), and the Adam adaptive learning rate algorithm. We supplied these algorithms with two potential learning rates, 0.1 and 0.01, and we trained the models for either 2, 5, or 7 epochs. As before, these three factors were combined to produce twelve unique results.

4 Data

Data (5pt) Initially, we used sentences from the training-data.1m file provided by statmt.org to formulate our contexts, which we then passed into our neural network. To improve our performance on the test set, we decided to also use sentences from the 1 billion word language modeling benchmark, also provided by statmt.org. This gave us a lot more contexts to train our network on, which yielded better performance in the test set. We computed our total vocabulary as the sum of all the words that appeared more than once among the 1,779,691 sentences.

Table 1: Data statistics

Metric	Value
Vocab Size	115,338
Number of Sentences (1m file)	897,691
Number of Sentences (1b file)	882,000
Total Number of Sentences	1,779,691
Positive Contexts	90,375,336
Negative Contexts	42,318,878
Total Contexts	132,694,214
Number of Development Examples	353
Number of Test Examples	2034

Pre-processing and Handling Unknowns (10pt) We preprocessed the data by loading sentences into a dataframe and then processing each sentence in turn. We lowercased all words and removed words that were not strictly alphanumeric characters. We also removed stop words (via nltk.corpus stopwords API). We did not conduct lemmatization because we felt that various word forms should maintain distinct embeddings. We generated word, context pairs according to the definition in the Model section, and we generated negative samples by pairing target words with contexts randomly sampled from the corpus. For windows looking d units to the left and right of each target word, we generated 2d+1 negative samples for each positive sample. Initially we trained embeddings for all the words in the training set, and assigned to new words the average of all the other embeddings. After observing poor performance of this method, we instead implemented a procedure of trimming rare words and replacing them with a RARE catch-all. For each word viewed once or less in the training corpus, we replaced those tokens with a special RARE token. This RARE token had it's embedding learned as any other word. Then, when new words were observed in the development and test sets, they were assigned the embedding of this RARE token. We also used nltk's wordnet functionality to find synonyms and, consequently, tag parts of speech on these synonyms for words that did not appear in our training set. We then appended the different parts of speech ('N' for noun, 'A' for adjective, 'V' for verb, 'R' for adverb) to our word-to-index dictionary. So, all nouns pointed to a particular index, all adjectives pointed to another index, as such. This remarkably improved our performance because our contexts (which are underpinned by these indices) now contain information about parts of speech, which can scale a lot easier to rare and unseen words. Specifically, as opposed to placing all rare and unseen words into a catch-all RARE bucket, we can now extract more granular information about these words (parts of speech), which improves the performance of the neural network.

5 Implementation Details (3pt)

We implemented the model using PyTorch, and experimented with many hyperparameters during the development process. The batch size, learning rate, optimization algorithm, epochs, and loss functions were decided through experimentation.

6 Results and Analysis

Test Results (3pt) We evaluated three alternative models, whose resulting performance is summarized in Table 1. The first model used 5000 training sentences and 30-dimensional embeddings, but showed very low performance on the test set. As we experimented on the development data, we found that better handling of rare words, and larger training corpora led to better performance. Ultimately the model with 50-dimensional embeddings, enhanced handling of rare words, and 3 milllion training sentences produced our best results, test correlation of 0.16077.

Table 2: Correlation scores on the test data by the size of the dataset and the number of dimensions in the embeddings.

data	**dim**	**Corr**
5K	30	0.021
100K	100	0.118
3M	50	0.16077

Development Results (15pt) In our first experiment, we looked at the interrelationship between loss functions, dimensionality of the embeddings, and training data size and these choices' impact on development correlation. Table 1 summarizes our findings; one, BCE loss performed better than MSE loss. Another straightforward finding is that larger training corpora produced better results. The impact of embedding dimensionality was less straightforward. With 1000 training samples, higher-dimensional embeddings fared better. However, this trend was reverse with more data. As our final model used much more data than any of our experiments, we used an intermediate value for embedding dimensionality.

Table 3: Correlation performance from experiment 1

MSE Loss	**NumSentence**	
dimEmbeds	1000	10000
10	0.0003	0.121
50	0.025	0.102
100	0.044	0.096
BCE Loss	**NumSentence**	
dimEmbeds	1000	10000
10	0.035	0.138
50	0.053	0.091
100	0.068	0.047

Next, we looked at additional design decisions in the neural network modelling and training process: optimizer choice, learning rate, and number of epochs. We channeled the learnings from the previous experiment and used 50 dimensional embeddings, 10000 training sentences, and BCE Loss. Table 5 shows the results. In general, the Adam optimizer proved more effective than SGD, though the latter was suprisingly performant with a high learning rate and few epochs. Typically, more epochs provided better performance, but the Adam optimizer clearly worked better with a learning rate of 0.01.

Qualitative Analysis (10pt) To assess causes of error in our model, we rank-ordered the development examples by the difference in their predicted and actual similarity. We reviewed the top 50 examples, and assigned groups iteratively, combining and removing groups until we settled on the summary shown in Table 4. We hypothesize that our model can be improved by including more deliberate handling of synonymy, especially when one synonym is a rare word, as in "dollar" and "buck". We also think that better performance would results from handling proper nouns through our rare word approach. Some errors are more mysterious, and these examples happened to be the second most frequent in our selected set.

Table 4: Correlation performance from experiment 2

Adam	**Learn. Rate**	
Epochs	0.01	0.1
2	0.1217	0.0367
5	0.1006	0.0932
7	0.1664	0.0256
SGD	**Learn. Rate**	
Epochs	0.01	0.1
2	0.0713	0.1505
5	0.0299	0.0743
7	0.0756	0.0364

Table 5: Error analysis.

Group	Count	Example
Synonym	21	gem, jewel
Other	10	chord, smile
Science	9	planet, star
Proper Noun	7	Arafat, terror

7 Conclusion (3pt)

Our best results on the development set yielded a correlation coefficient = 0.487. Our bet results on the test set yielded a similarity coefficient = 0.168. For further experimentation, we would have definitely liked to use more sentences to generate our positive and negative contexts. Due to limited compute time and power, we were only able to generate contexts for approximately 1.8 million sentences. However, we noticed that this immediately yielded an improvement in performance on our development and test sets.

CS5740: Assignment 3
https://github.com/cornell-cs5740-20sp/assignment-3-asd

Chris Shei
cs856

Kushal Singh
ks2377

1 Introduction (5pt)

In this abstract, we implement and discuss several models for part-of-speech (POS) tagging. A POS tagger classifies each word in a sentence according to its lexical category. For example, consider the following sentence: "A shareholder filed suit". In this case, the correct POS tags would be: ("A", *DT*), ("shareholder", *NN*), ("filed", *VBD*), ("suit", *NN*), where *DT* is abbrev. for "Determiner", *NN* is abbrev. for "Noun", *VBD* is abbrev. for "Verb, past tense".

The data used in this assignment contains approx. 1 million words of text from the Wall Street Journal. The words and symbols that comprise each document are written in column format, where every document begins with the tag -DOCSTART-.

2 Data (5pt)

Table 1: Data Statistics

Metric	Value
Total Unprocessed Tokens (Training)	696475
Unique Vocab Size (Training)	37504
Number of Sentences (Training)	28837
Total Unprocessed Tokens (Dev)	243021
Unique Vocab Size (Dev)	20705
Number of Sentences (Dev)	10078
Total Unprocessed Tokens (Test)	236582
Number of Sentences (Test)	9818
Total Parts of Speech	46

From the training data statistics, we can observe that tags and words are fairly sparse. Excluding the START and STOP tags, there are 46 tags, with the most common tag being *NN* (accounting for 13.9% of all tags). The top 5 tags are *NN, IN, NNP, DT, JJ* and they comprise 49.0% of all tags. There are 37,504 unique words in the training data, of which 17,592 words (46.9%) only appear once.

For our preprocessing, we first loaded the data, and applied a mapper function to each word. This mapper function either: (1) returned the raw word, as is, if it appeared more than twice in the training set, OR (2) returned a token, based on its category, which is defined in the section below, OR (3) returned a suffix, if it was informative, which is also defined in the section below, OR (4) returned the RARE key to denote that it didn't meet any of the other criteria mentioned in options (1)-(3).

Then, we create unigrams, bigrams, trigrams, and quadgrams of words in sentences and tag them with how many times a sequence of words is followed by a particular tag. For instance, for unigrams, we count how many times an *NNP* is followed by another *NNP*, how many times a *DT* is followed by an *NN*, etc. Similarly, for bigrams, we count how many times an (*NNP*, NNP) sequence is followed by another *NNP*. The same logic holds for trigrams and quadgrams. We then store all of these dictionaries in a global dictionary ($transition - counts$) representing the total counts of transitioning from different n-gram sequences to a particular POS tag, where the keys 1, 2, 3, 4 correspond to the dictionaries for unigrams, bigrams, trigrams, and quadgrams, respectively.

We also create a dictionary mapping how many times a sequence of tags appears for each n-gram. For instance, in the unigram case, we store how many NNs, NNPs, NNSs, etc., appear. In the bigram case, we store how many (*RBS*, *VBN*), (*NNP*, *NNP*), etc., sequences actually appear. We then store all of these dictionaries in a global dictionary ($state - counts$), where the keys 1, 2, 3, 4 correspond to the dictionaries for unigrams, bigrams, trigrams, and quadgrams, respectively.

3 Handling Unknown Words (15pt)

To handle unknown and rare words, we leverage three different methods.

Categories: We categorize unknown words based on certain shared properties. Among other characteristics, these words are considered special, since they contain numbers, hyphens, and are case sensitive. The categories for these words include: Numbers (e.g. *25*), Words that only start with one capital letter (e.g. *Grace*), Words with all capital letters (e.g. *FERC*), Words with all capital letters that end in '*s*' (e.g. *UFOs*), Words that contain both letters and numbers (e.g. *pre-1917*), Numbers that end in '*s*' (e.g. *1950s*), Words that are connected to a lower-case word by a hyphen (e.g. *dollar-denominated*), Upper-case words that are separated by a hyphen (e.g. *Chinese-American*, *No-Smoking*). Words in these categories are generally tagged as *CD* (cardinal number), *NNP* (noun, proper singular), *NNP*, *NNS* (noun, plural), *JJ* (adjective), *CD* or *NNS*, *JJ*, *NNP*, respectively.

Suffixes: Suffixes ended up being a strong predictor for part-of-speech tags. If an unknown word does not fit into one of the granular categories defined above, then we consider its suffix of length 2. We create a key, value pair of (*suffix, POS tag*) and append that to our suffix dictionary. We only consider a suffix to be informative if its (*suffix, POS tag*) appears more than 7 times. We observed that the total number of suffixes in the training data prior to and subsequently after applying the 7-count filter was equal to 850 and 109, respectively. The (*suffix, POS tag*) tuples that appeared the most were (*VBG, "-ng"*) at 974 times, (*VBN, "-ed"*) at 709 times, (*NNS, "-es"*) at 609 times, (*RB, "-ly"*) at 444 times.

RARE Key: If a particular word does not fall into any of the aforementioned categories detailed above, or did not have an informative suffix associated with it, then that word is replaced with the RARE key. In total, there were 697 such words in the training set.

The table below describes our performance on handling unknown words for the development set.

Table 2: Unknown Word Performance

Group	Count	Percent
Numbers	1736	99.5%
Numbers + 's'	11	72.8%
Uppercase sep. by hyphen	118	73.8%
Word + Hyphen + Lowercase	1314	72.7%
First Letter Uppercase	4990	79.6%
All Letters Uppercase	775	70.2%
All Letters Uppercase + 's'	2	100%
Words w/ letters and numbers	156	82.1%
Total Number of Suffixes	6104	80.9%
Pure Unknowns	328	66.2 %

4 Smoothing (10pt)

Smoothing flattens a probability distributions of particular n-grams, such that **all** word sequences can occur with some probability. The motivation behind this is so that unseen words don't have emission probabilities equal to 0, if the training data is sparse. We can achieve this by redistributing weights from high probability regions to zero probability regions.

For our first smoothing technique, we used Add-k (aka Laplace) smoothing, which pretends that we saw each sequence k more times than we actually did. In the case of add-1 smoothing, for example, we pretend that we saw each word one more time than we actually did (so unseen words aren't completely whitened). This is a non-maximum likelihood estimator because we are changing the counts of what we observed in our training data to hope to generalize better in our development and test data. The general form of Add-k smoothing is:

$$P_{add-k} = \frac{c(x_{i-1}, x_i) + k}{c(x_{i-1}) + kV}$$

The second smoothing technique we used was backoff (aka Katz) smoothing. In our implementation, we computed the adjusted counts of different n-grams, and for n-grams with nonzero count r, we discounted according to a discount ratio d_r, which is detailed in the Lecure 7 slide. We used this smoothing technique because it tends to perform better on large training sets and on n-grams with large counts.

5 Implementation Details (5pt)

We implement and discuss three models for POS tagging: Greedy, Viterbi, and Beam Search.

The greedy search algorithm maximizes the likelihood between neighboring states by searching for the most likely transition between the current state and the subsequent state. Beam search is an extension of greedy search, which finds the top k possible transitions between the $i-1_{st}$ token and the i_{th} token. Viterbi uses dynamic programming to find the global optimal POS tags.

No optimizations were needed to improve runtime performance. The hyper-parameters used included: threshold for generating suffixes, n for optimal n-gram sequences, k for beam search, k in add-k smoothing.

6 Experiments and Results

Test Results (3pt) We generated this test score using our procedure for handling unknown words described above. The hyper-paramater values were $n = 3$ for trigram sequences, $k = 3$ for beam search, and $k = 0.2$ for add-k smoothing.

N-gram	Beam	Add-k	Test Accuracy
n = 3	k = 3	k = 0.2	0.94857

Smoothing (5pt)

	Add-K Smoothing	Unknown Acc
k=0.2	0.94857	0.8061
k=0.4	0.94848	0.7985
k=0.6	0.94829	0.7926
k=0.8	0.94807	0.7910
	Backoff	
	0.9402	0.7965
	No Smoothing	
	0.9438	0.7929

As shown in the table above, add-k smoothing with $k = 2$ had the best performance, but only marginally. Interestingly enough, no smoothing technique seems to perform better than backoff smoothing.

Bi-gram vs. Tri-gram (5pt)

	Accuracy	
Methods	Overall	Unknown
Bigram, k=3	0.94216	0.79128
Trigram, k=3	0.94857	0.80487
Bigram, Viterbi	0.94220	0.79265
Trigram, Viterbi	0.94848	0.80229

Based on the results from the table above, we can see that increasing gram size does not show remarkable improvement in accuracy for overall and unknown words. The bigram and trigram models performed fairly similar to each other. Despite the smaller context window size, the bigram results seem to be higher than expected, suggesting that bigrams have enough context to achieve good results on the training set. Increasing gram size also increases the search space sparsity and potential noise for transition and emission probabilities.

Greedy vs. Viterbi vs. Beam (10pt)

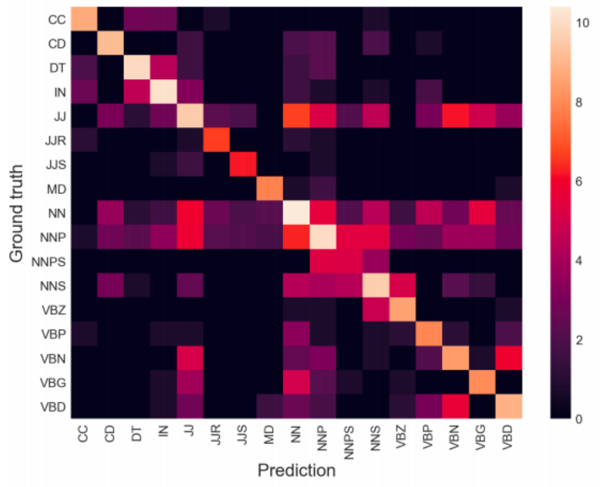
Method	Accuracy	Suboptimal	Unk Acc
Greedy	0.93965	0.785	0.7809
Beam, k=2	0.94109	0.702	0.7987
Beam, k=3	0.94857	0.701	0.8049
Beam, k=4	0.94762	0.705	0.8017
Beam, k=5	0.94513	0.709	0.7992
Viterbi	0.94798	0	0.8022

From the table, we can see that beam size had minimal impact on accuracy and solution optimality. Even though greedy rarely finds the optimal sequence, it still yielded a fairly high accuracy.

7 Analysis

Error Analysis (7pt) Based on the error analysis we conducted and the confusion matrix shown below, we observed that one of the most common misclassifications was when the tagger mapped *NN* and *NNP* to *JJ* (i.e. nouns to adjectives). There were also some *VBN* and *VBG* that were misclassified as *JJ* because a significant amount of *VBN* and *VBG* that end with *-ed* and *-ing* are actually adjectives. One such example was the word "retired", which was tagged as *VBD* when it was actually *JJ*. This type of error is hard to handle because it is an edge case in the English language, where a verb's past tense can act as as an adjective.

Confusion Matrix (5pt) With regards to the POS tags, there were two predominant clusters which align with normal English sentences: Nouns (e.g. *NN, NNP, NNPS, NNS*) and Verbs (e.g. *VBP, VBN, VBG, VBD*). Because the differences within each of these clusters are very nuanced, most misclassifications occurred within these clusters. For example, the word "Books" was classified as *NNS*, even though its actual POS tag is *NNPS*.



8 Conclusion (3pt)

We explored different search methods (greedy, beam, Viterbi), unknown words handling, and gram sizes. Handling unknown words proved crucial to predicting POS tags due to their direct impact on common words via transition emissions. To handle this, we grouped unknown words into one of several categories, used bigram suffixes where necessary, and added a RARE Key for words that didn't fit into either of the above handles. Gram size didn't play a huge factor in determining accuracy, possibly because context information was not as crucial as unknown word handling and smoothing. Greedy search showed the worst performance, while beam search had a very similar performance with Viterbi. Increasing width of beam search marginally improved accuracy. Next steps would be to leverage a better smoothing method to improve model performance.

CS5740: Assignment 4

`https://github.com/cornell-cs5740-20sp/assignment-4-nlprime`

Kushal Singh
ks2377

Willy Lin
wl677

8 May 2020

We are using `DyNet`.

1 Introduction (5pt)

The goal of this assignment is to implement a sequence to sequence model to map instructions to action in the Alchemy environment, which contains a certain number of beakers, colors, and beaker positions. The system can move colored liquid content from one beaker to another, mix content, or remove content from the set of beakers. The data provided contains:
1) a unique ID (e.g. train-A9164)
2) an initial environment configuration which represents the state of the initial Alchemy beaker setup, prior to executing the sequence of actions (e.g. 1 2 3 4p 5bb 6yyy 7rrrr represents the initial state of the seven beakers such that beakers 1 through 3 are empty, beaker 4 contains one part purple, beaker 5 contains two parts brown, beaker 6 contains three parts yellow, and beaker 7 contains four parts red
3) a list of utterances, where the key is the natural language instruction (e.g. "pour purple beaker into yellow one"), and the value is a set of actions corresponding to that instruction (e.g. "pop 3, push 5 p, stop"), as well as the final environment configuration which represents the state of the Alchemy beaker setup, once the action has been executed.

Primary experiments included testing for optimal DyNet builder and optimizer, making incremental additions to the model (e.g. adding state information, adding attention, adding interaction history) and fine-tuning hyper-parameters (e.g. number of hidden layers, number of training epochs). Cursory results of these experiments included an embedding dimension size of 50 for word-level embeddings and 20 for character-level embeddings, number of hidden layers at 300, and 30 training epochs. The model was evaluated by testing development data prediction against the ground truth, using two key metrics: single-instruction task completion and interaction task completion. Interaction-level accuracy tended to be lower than instruction-level accuracy, since the former requires correct execution of ALL the instructions in the sequence. Final instruction-level accuracy achieved on leaderboard was **0.48552**.

2 Model (35pt)

To achieve the objective detailed above, we used a sequence-to-sequence model that comprises of two neural networks: an encoder and a decoder. In tandem, the encoder and decoder help the model learn to perform a sequence of actions corresponding to language instructions and environment states. Specifically, the encoder encodes instructions into a context vector. The decoder, in turn, takes this context vector and based off the environment state predicts the actions to be performed.

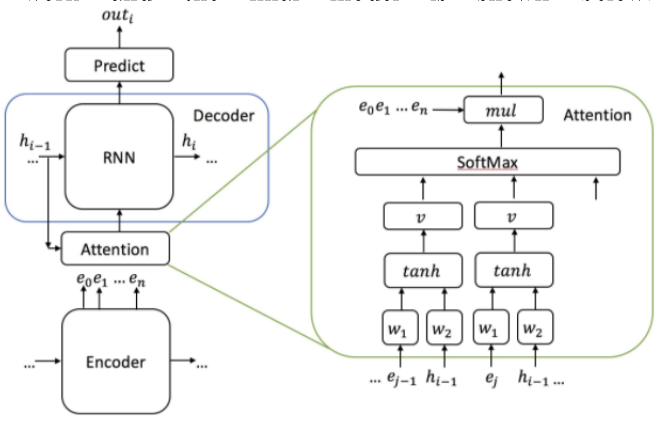
The first optimization we made to our model was to incorporate environment state information, so that when the model is provided with the natural language expression *throw out two units of brown one* it knows which beaker has the brown color and in which positions in order to predict the right action. In order to incorporate this behavior we created an embedding matrix for all 7 beakers, where each individual entry contained color by parts information for each beaker. Subsequently, we add attention, thereby allowing the decoder to pick the encoded inputs that are relevant to make the prediction at hand. Finally, the output is concatenated to each input of the decoder.

The second optimization we made to our model was adding an attention mechanism, which computes a weighted combination of multiple inputs, where the weights are determined by the current state of the decoder. The attention mechanism attends on both the output of the instruction encoder and the output of the environment encoder, since these were the two aspects that most directly determined an optimal decoded state.

The third optimization we made to our model was adding interaction history, so that when the model is presented with the series of instructions [*pour sixth beaker into last one*, *it turns brown*], it knows that the word "it" is referring to "last one" as its modifier. To achieve this, we concatenate the previous instruction to the current instruction, and pass this as input to the encoder, instead of passing each instruction individually. So, the instruction sequence above becomes *pour sixth beaker into last one <end> it turns brown*].

We used Finite State Automata (FSA), which was provided as part of the template code, to model the transitions provided by the decoder's output, given the initial environment state and final environment state.

A visualization of the encoder-decoder network and the final model is shown below.



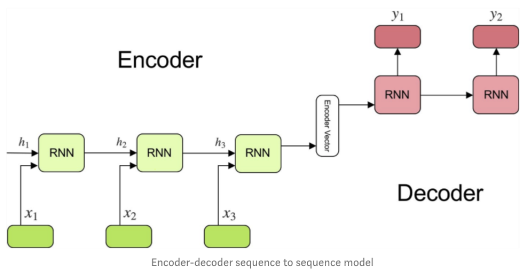
3 Learning (15pt)

To maintain consistency between training and testing data, we grouped series of five sentences into a single batch, and trained on this batch. By doing this, our model was robust enough to handle aforementioned discrepancy of training on single sentences versus testing on sequences, which could have lead to errors where antecedents were not correctly referred back to their modifiers.

To calculate instruction and interaction level losses, we made the MLE prediction, took the negative log, and then used the DyNet esum function to calculate loss expressions. During training, we grouped series of five instructions into one batch, took the average loss for each expression tied to an instruction, propagated this loss back to readjust network weights, and then reset batch loss.

The stopping criteria for making an interaction-level prediction was based on whether a prediction was even possible for a given environment configuration and, if so, making a sequence of action predictions until hitting the <end> tag. The stopping criteria for training the model was determined by the number of epochs. The training data was bundled into a list of instructions, actions, environment states, and IDs.

The graph below visualizes the encoder-decoder sequence discussed earlier. Within each of the encoder cells is a recurrent neural network (RNN), which we chose to represent as an LSTM, since instructions and actions can be determined by long term memories.



$$
\begin{aligned}
i_t &= \sigma(W_{ix}x_t + W_{ih}h_{t-1} + b_i) \\
f_t &= \sigma(W_{fx}x_t + W_{fh}h_{t-1} + b_f + 1) \\
o_t &= \sigma(W_{ox}x_t + W_{oh}h_{t-1} + b_o) \\
\tilde{c}_t &= \tanh(W_{cx}x_t + W_{ch}h_{t-1} + b_c) \\
c_t &= c_{t-1} \circ f_t + \tilde{c}_t \circ i_t \\
h_t &= \tanh(c_t) \circ o_t
\end{aligned}
$$

4 Data (5pt)

Shallow statistics of the data can be found in the table below. The filtered instructions vocab size represents the number of instruction words that appeared more than 7 times throughout the entire training set. Words that appeared less than 7 times were tagged as unknown via the '<UNK>' key. Further analysis revealed that several of these rare words were typos of relevant words. For instance, *thirds* was misspelled as *thrids*, while *beaker* was misspelled as *beeker* and *breaker*. Therefore, the total number of unknown words was calculated as the difference between the length of the corpus after the above filter was applied. In total, there were 610 - 245 = 365 unknown words in training set, 133 - 74 = 59 unknown words in dev set, and 252 - 121 = 131 unknown words in the test set.

Table 1: Data Statistics

Metric	Value
Instruction and Interaction Size (Training)	18285
Raw Instructions Vocab Size (Training)	148568
Raw Actions Vocab Size (Training)	77449
Unique Instructions Vocab Size (Training)	610
Unique Actions Vocab Size (Training, Dev)	51
Filtered Instructions Vocab Size (Training)	245
ID Size (Training)	3657
Instruction and Interaction Size (Dev)	1225
Filtered Instructions Vocab Size (Dev)	74
ID Size (Dev)	245
Instruction and Interaction Size (Test)	2245
Filtered Instructions Vocab Size (Test)	121
ID Size (Test)	449

For data preprocessing, I extracted sentence instructions (e.g. *throw out first beaker*), action sequences (i.e. *pop x*, *push y o*), environment stage configurations, and ID's (i.e. *train-A9164*) for the training, development, and test data. I also vectorized words by creating a *vocab_dict* dictionary that mapped all words from sentence-level instructions to integers and a *vocab_actions_dict* that mapped all words from action sequences to integers, from the training set. Finally, I created a *conv_char_to_int* dictionary that vectorized the tagged beaker colors as follows: {_: 0, 'b': 1, 'g': 2, 'o': 3, 'p':4, 'r':5, 'y':6}.

5 Implementation Details (2pt)

The final sequence-to-sequence model used simple stochastic gradient descent (SimpleSGD) as the optimization algorithm. It comprised of the following hyper-parameters and their values:

```
LSTM_NUM_OF_LAYERS = 1
EMBEDDINGS_SIZE_INSTRUCTION = 50
EMBEDDINGS_SIZE_ACTION = 50
EMBEDDINGS_SIZE_BEAKER_STATE = 50
HIDDEN_SIZE_ENCODER_1 = 100
HIDDEN_SIZE_ENCODER_2 = 75
HIDDEN_SIZE_DECODER = 100
ATTENTION_SIZE = 100
EPOCHS = 50
```

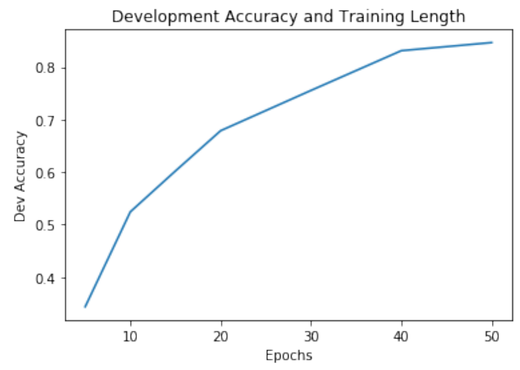
During the results and experimentation phase, the following variables were fine-tuned to optimize performance: embedding size (for instruction, action, beaker state representations), DyNet builder, model optimizer, hidden layer size, and number of epochs.

6 Experiments and Results

Test Results (3pt) We generated this test score using our procedure for handling unknown words described above. The hyper-paramater values were $d = 50$ for input embeddings and $h = 100$ for attention size.

Builder	Optimizer	Test Accuracy
LSTM	SimpleSGD	0.48552

Ablations (15pt) Number of epochs had a directly positive correlation with instruction-level accuracy on the development data, as seen from the plot below.



Method	Optimizer	Instruction Acc
VanillaLSTM	SimpleSGD	0.5492
CompactVanillaLSTM	SimpleSGD	0.4985
CoupledLSTM	SimpleSGD	0.4456

Progressive task completions played a crucial role in increasing instruction-level and interaction-level accuracy, with state information producing the highest jump in accuracy, as seen from the table below.

	Accuracy	
Task	Instruction	Interaction
Seq2Seq Model	0.1285	0.0143
W/ State Info	0.4458	0.2271
W/ Attention	0.4944	0.2619
W/ Interaction History	0.5492	0.2835

Different model optimizers yielded noticeable performances in instruction-level and interaction-level accuracy, with SimpleSGD netting the best overall accuracy.

	Accuracy	
Optimizer	Instruction	Interaction
Adam	0.5286	0.2628
SimpleSGD	0.5492	0.2835
CyclicalSGD	0.5161	0.2543
MomentumSGD	0.4677	0.2179
Adagrad	0.4827	0.2394

7 Error Analysis (7pt)

We identified several classes of error in interpretation. For instance, with transferance between containers, there exist inaccurate key phrases that the model fails to recognize such as commands that indicate multiple repeated actions such as "add remaining" or "empty into". However, the model seems to perform better on commands that specifically dictate the quantity of transferance, e.g. "pour two". This class of "vague" actions seem to trick the model whereas concise instructions with a discrete value are interpreted well. Similarly, actions that dictate relational modifiers perform worse than ones targeting a single beaker by color, e.g. "rightmost" vs "orange chemical" as there is an extra step of comparing the relation of the entire set which introduces complexity.

8 Conclusion (3pt)

The largest improvement in sequence-to-sequence model performance stemmed from adding state information (Task 2 in the spec report). Specifically, the instruction-level accuracy jumped from 12% to 44% after incorporating the initial state information. Furthermore, adding state information, attending on color states and instructions, and adding history interaction utterance, enriched contextual knowledge gleaned during neural network training phase, and also provided substantial improvements. Fine-tuning hyper-parameters (e.g. number of hidden layers, embedding dimensions, attention method, optimizer, number of epochs) proved crucial in making incremental performance progress. Next steps would be to increase the n-gram size for interaction history to consider multiple prior instructions (n>1) instead of only the previous instruction (n=1), thereby providing more contexts for pronouns and their antecedents, as well as exploring other attention mechanisms.