Part 5

Name

Simon Raviv

Best Parameters

NER Tagging (with pretrained word embedding)

- Epochs: 15
- Hidden layer size: 64
- Batch size: 64
- Learning rate: 0.002
- Learning rate scheduling gamma: 0.5
- Learning rate scheduling step size: 2
- Dropout probability: 0.5
- Weight decay: 0.00004
- Char embedding dimension: 30
- CNN char number of filters: 5
- CNN char kernel size: 1
- CNN char stride: 1

POS Tagging (with pretrained word embedding)

- Epochs: 30
- Hidden layer size: 64
- Batch size: 128
- Learning rate: 0.0012
- Learning rate scheduling gamma: 0.5
- Learning rate scheduling step size: 10
- Dropout probability: 0.5
- Weight decay: 0.000005
- Char embedding dimension: 30
- CNN char number of filters: 30
- CNN char kernel size: 3
- CNN char stride: 1

Accuracy:

TODO: Update.

The final accuracy at the end didn't show too much of a difference. The difference was in the training process itself. See Results Analysis section for more information.

NER Tagging:

The dev accuracy (without considering common tag O) ~83.7 [%] with pretrained embedding.

POS Tagging:

The dev accuracy is ~95.4 [%] with pretrained embedding.

Results Analysis

Methodology

I have started with doing multiple mini experiments for NER and POS tasks. The goal of the mini experiments was to get the filling and intuition of how the different hyper parameters of the model impact the performance, in addition to trying to give reasoning for the different phenomena I observe.

Following the mini experiments, I have built an experiment plan for both tasks and followed them carefully. The first experiment was the baseline experiment where the hyper parameters from part 4 were chosen with some modifications e.g., without regularization techniques. And the new CNN related parameters were chosen to be the parameters from the article. This is done to get the baseline experiment and improve from there, based on the empirical results.

Then, I have added groups of experiments, where each group meant to test the impact of a specific hyper parameter under test while keeping all others constant (for the most of them). The order of the groups was chosen to get the correct path for the experiments as I see it. Following the results gained from each group, the next groups parameters were updated to reflect a better experiment for the next groups with the results of previous groups.

The parameters for the next groups were chosen not just by the final dev accuracy, but also following the behavior in the fitting process, e.g., how quick the model converges, the accuracy improvement rate, the loss improvement rate, etc.

The last group at each table consist of experiments (some of them, I have conducted many more) with parameters changed following the knowledge I gained during the process described above.

Results

Tables legend:

- Index 1 is the baseline experiment
- The groups separated by bold line borders
- The hyper parameters under test bolded and colored with orange
- The best (final) epoch result marked with light green background
- The experiment marked with green in the last group is the final and best experiment, the one with the chosen parameters.

NER Tagging

This is the summary of the results, see Detailed Results section for the full table. The most impacting parameters are number of filters, kernel size, regularization and learning rate with scheduling.

General

Overall, compared to part 4, there is slightly better final accuracy 82.3 [%] vs 83.7 [%]. In addition, the convergence process is faster using the CNN characters embeddings. See experiment 32.

Filters

Using 30 (as in the article) or more filters the results are worst. In NER tagging, it looks like the fewer the better, up to about 5 filters totally. See experiments 1-6.

Kernel Size

In the preliminary experiments, bigger kernel size then 3 (as in the article) showed better results. See experiments 10-13. But after optimization at the end, the kernel size with best results was 1. See experiments 27-32.

Char Embedding Dimension

Increasing or decreasing the dimension of the embedding vectors didn't improve the results. Decreasing the dimension did make the first few epochs to converge faster, but that make sense, since with lower dimension, there are fewer weights to learn. See experiments 14-15.

Regularization

I have used 2 regularization techniques, L2 weight decay and dropout. Both helped with generalization and achieving better accuracy on the dev. See experiments 7-9, 16-17, 24-32.

Batch Size

Changing the batch size from 64 didn't improve the accuracy. See experiments 20-21.

Hidden Layer Dimension

Changing the hidden layer size from 64 didn't improve the accuracy. See experiments 22-23.

Learning Rate

The learning rate and learning rate decay scheduling did good impact on the generalization and dev accuracy. See experiments 24-32.

Stride Size

Additional experiments (that are not shown here) didn't show advantage for changing stride size.

Detailed Results

Index	Parameters												NER Acc. [%]	
	Epochs	Hidden Dimension	Batch Size	LR	LR Sched Step	LR Sched Gamma	Dropout	Weight Decay	Char Embedding Dim	Filters Num	Kernel Size	Stride Size	Test	Dev
1	16	64	64	0.001	1	1	0	0	30	30	3	1	99.3567	66.2292
2	16	64	64	0.001	1	1	0	0	30	40	3	1	99.2951	65.3234
3	16	64	64	0.001	1	1	0	0	30	10	3	1	99.3146	69.1994
4	16	64	64	0.001	1	1	0	0	30	7	3	1	99.2403	65.5199
5	16	64	64	0.001	1	1	0	0	30	5	3	1	99.3034	66.5096
6	16	64	64	0.001	1	1	0	0	30	3	3	1	99.2413	67.5821
7	16	64	64	0.001	1	1	0.5	0.00001	30	10	3	1	99.1489	72.4066
8	16	64	64	0.001	1	1	0.5	0.00001	30	7	3	1	99.1181	72.7354
9	16	64	64	0.001	1	1	0.5	0.00001	30	5	3	1	99.2144	75.1129
10	16	64	64	0.001	1	1	0.5	0.00001	30	5	5	1	99.2003	78.7546
11	16	64	64	0.001	1	1	0.5	0.00001	30	5	2	1	99.1895	70.6018
12	16	64	64	0.001	1	1	0.5	0.00001	30	5	3	1	99.2193	71.1671
13	16	64	64	0.001	1	1	0.5	0.00001	30	5	1	1	99.1318	73.9372
14	16	64	64	0.001	1	1	0.5	0.00001	20	5	5	1	99.1577	73.5836
15	16	64	64	0.001	1	1	0.5	0.00001	40	5	5	1	99.1641	73.2652
16	16	64	64	0.001	1	1	0.5	0.00003	30	5	5	1	99.323	77.1195
17	16	64	64	0.001	1	1	0.5	0.00004	30	5	5	1	99.2883	72.7254
18	16	64	64	0.001	1	1	0.65	0.00001	30	5	5	1	99.1436	74.9376
19	16	64	64	0.001	1	1	0.75	0.00001	30	5	5	1	99.1988	74.8364
20	16	64	80	0.001	1	1	0.5	0.00001	30	5	5	1	99.2785	73.1959
21	16	64	32	0.001	1	1	0.5	0.00001	30	5	5	1	98.992	77.0327
22	16	32	32	0.001	1	1	0.5	0.00001	30	5	5	1	99.1035	72.6751
23	16	128	32	0.001	1	1	0.5	0.00001	30	5	5	1	98.9363	74.1564
24	16	64	64	0.004	2	0.6	0.5	0.00003	30	5	5	1	99.7052	79.965
25	16	64	64	0.003	3	0.5	0.5	0.000025	30	5	5	1	99.7131	75.8224
26	16	64	64	0.002	4	0.4	0.5	0.000035	30	5	5	1	99.6974	72.5624
27	16	64	64	0.002	2	0.5	0.5	0.00004	30	5	1	1	99.6265	83.3185
28	16	64	64	0.002	4	0.5	0.6	0.00004	30	5	3	1	99.6783	77.401
29	16	64	64	0.0025	3	0.5	0.7	0.00004	30	5	3	1	99.6852	77.91
30	16	64	90	0.002	2	0.5	0.5	0.00004	30	5	1	1	99.6251	81.2514
31	16	64	64	0.002	3	0.5	0.5	0.00004	30	5	1	1	99.6392	82.6665
32	**15**	**64**	**64**	**0.002**	**2**	**0.5**	**0.5**	**0.00004**	**30**	**5**	**1**	**1**	**99.6128**	**83.6961**

POS Tagging

This is the summary of the results, see Detailed Results section for the full table. The most impacting parameters are regularization and learning rate with scheduling.

General

Overall, compared to part 4, there is slightly better final accuracy 94.9 [%] vs 95.4 [%]. In addition, the convergence process is faster using the CNN characters embeddings. See experiment 28.

Filters

Using more or less than 30 filters (as in the article) didn't show too much of a difference. See experiments 1-7.

Kernel Size

In the preliminary experiments, bigger kernel size (compare to 3 in the article) of 5 or smaller kernel size of 2/1 showed better results. See experiments 5-10. But after optimization at the end, the kernels sizes with best results were 5/3. See experiments 24-32.

Char Embedding Dimension

Increasing or decreasing the dimension of the embedding vectors didn't improve the results. Decreasing the dimension did make the first few epochs to converge faster, but that make sense, since with lower dimension, there are fewer weights to learn. See experiments 11-12.

Regularization

I have used 2 regularization techniques, L2 weight decay and dropout. Both helped with generalization and achieving better accuracy on the dev. See experiments 5-7, 13-16, 24-32.

Batch Size

Changing the batch size from 64 didn't improve the accuracy. See experiments 17-18,21-23,26.

Hidden Layer Dimension

Changing the hidden layer size from 64 didn't improve the accuracy. See experiments 19-20.

Learning Rate

The learning rate and learning rate decay scheduling did good impact on the generalization and dev accuracy. See experiments 21-32.

Stride Size

Additional experiments (that are not shown here) didn't show advantage for changing stride size.

Detailed Results

Index	Parameters												POS Acc. [%]	
	Epochs	Hidden Dimension	Batch Size	LR	LR Sched Step	LR Sched Gamma	Dropout	Weight Decay	Char Embedding Dim	Filters Num	Kernel Size	Stride Size	Test	Dev
1	20	64	128	0.001	1	1	0	0	30	30	3	1	97.5204	94.4497
2	20	64	128	0.001	1	1	0	0	30	45	3	1	97.7591	94.9884
3	20	64	128	0.001	1	1	0	0	30	10	3	1	97.5825	94.4559
4	20	64	128	0.001	1	1	0	0	30	7	3	1	97.3397	94.1114
5	20	64	128	0.001	1	1	0.5	0.00001	30	30	3	1	96.8805	95.0114
6	20	64	128	0.001	1	1	0.5	0.00001	30	45	3	1	96.8643	94.9905
7	20	64	128	0.001	1	1	0.5	0.00001	30	20	3	1	96.8598	95.0866
8	20	64	128	0.001	1	1	0.5	0.00001	30	30	5	1	96.905	95.0949
9	20	64	128	0.001	1	1	0.5	0.00001	30	30	2	1	96.8162	95.1074
10	20	64	128	0.001	1	1	0.5	0.00001	30	30	1	1	96.7104	94.8652
11	20	64	128	0.001	1	1	0.5	0.00001	20	30	3	1	96.8411	95.0218
12	20	64	128	0.001	1	1	0.5	0.00001	40	30	3	1	96.7896	94.8944
13	20	64	128	0.001	1	1	0.5	0.00003	30	30	3	1	96.2854	94.5228
14	20	64	128	0.001	1	1	0.5	0.00004	30	30	3	1	94.7234	93.0026
15	20	64	128	0.001	1	1	0.65	0.00003	30	30	3	1	96.2908	94.552
16	20	64	128	0.001	1	1	0.75	0.00003	30	30	3	1	96.3031	94.5687
17	20	64	64	0.001	1	1	0.5	0.00003	30	30	3	1	95.8793	94.1991
18	20	64	32	0.001	1	1	0.5	0.00003	30	30	3	1	95.5492	94.0153
19	20	32	128	0.001	1	1	0.6	0.00004	30	30	3	1	94.7556	92.9984
20	20	128	128	0.001	1	1	0.6	0.00004	30	30	3	1	94.6518	92.8982
21	30	64	80	0.004	3	0.6	0.6	0.00005	30	30	3	1	95.051	93.1216
22	30	64	80	0.003	4	0.5	0.6	0.00005	30	30	3	1	95.0197	93.0694
23	30	64	80	0.002	5	0.4	0.6	0.00005	30	30	3	1	96.4102	94.4956
24	30	64	128	0.0015	8	0.5	0.5	0.000045	30	30	3	1	95.2108	93.2385
25	30	64	128	0.0015	8	0.5	0.5	0.000045	30	30	5	1	95.0259	93.1258
26	30	64	512	0.008	4	0.7	0.5	0.000045	30	30	3	1	96.6841	94.6543
27	30	64	128	0.0015	8	0.6	0.5	0.000015	30	30	3	1	97.5574	95.2244
28	**30**	**64**	**128**	**0.0012**	**10**	**0.5**	**0.5**	**0.000005**	**30**	**30**	**3**	**1**	**97.6641**	**95.3705**
29	45	64	128	0.008	10	0.8	0.5	0.00001	30	30	3	1	97.4477	95.0197
30	50	64	128	0.0012	10	0.5	0.5	0.000003	30	30	5	1	98.0537	95.3079
31	50	64	128	0.0019	8	0.8	0.5	0.000003	30	30	5	1	97.7989	95.1805
32	50	64	128	0.002	10	0.5	0.5	0.000003	30	30	5	1	98.1159	95.2181

Meaningful Filters Analysis

Algorithm

I have defined and implemented the following algorithm to investigate the filters meaning.

In short:

1. For every word in vocabulary, pass it through the convolution layer.
2. For every filter in the filters, do dimensionality reduction of the convolved words from the output channel dimension to 2 using TSNE.
3. For every filter, plot all words in 2 D.
4. The meaning of the filters can be analyzed by examining the generated cluster of Euclidean close words.

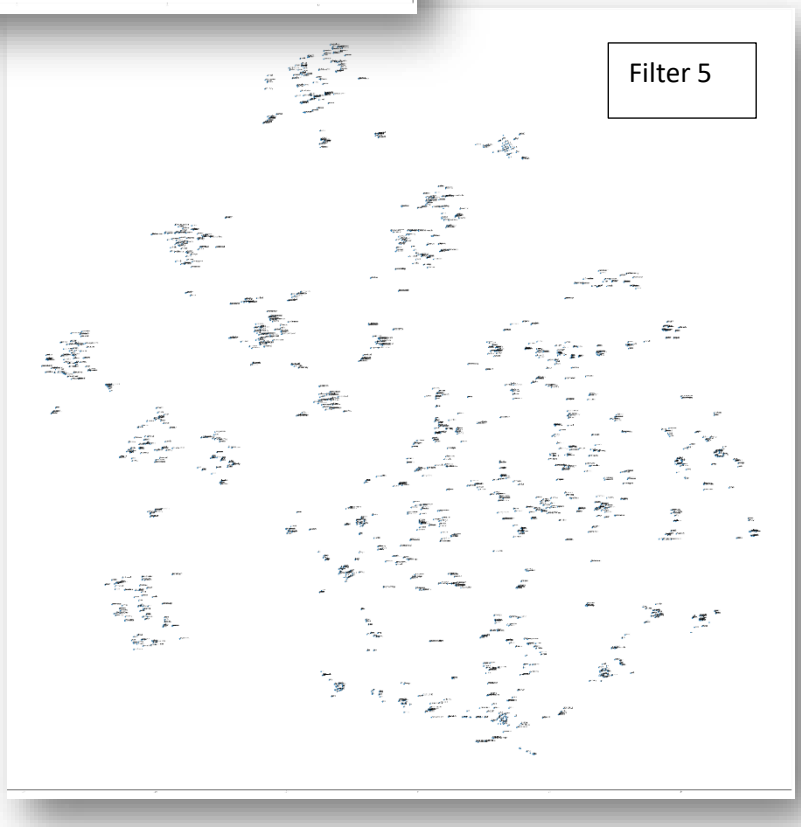
The algorithm shown below, it's part of the application, see the README file for usage instructions. Note, in case you run it, be aware it might take several minutes to compute.

```python
def visualize_cnn_filters(dataset: Dataset, model: torch.nn.Module,
                          word_frequency: int = 1, plot_path: str = "") -> None:
    """
    @brief: Shows the filters explanation.

    The method is supported when sklearn is installed.
    It will save the filters to plot files.

    @param dataset: The train dataset.
    @param model: The model to use.
    @param word_frequency: The number of words to show.
    @param plot_path: The path to the plot file.

    @return: None.
    """
    if TSNE is None:
        print("Please install sklearn to visualize the filters.")
        return None

    print("Visualizing filters, please wait, this might take a while...")

    # Get the filters
    convolved_filters = []
    vocabulary_data_loader = get_vocabulary_data_loader(
        dataset=dataset, batch_size=model.batch_size * 5, word_frequency=word_frequency)
    for batch in vocabulary_data_loader:
        convolved_batch_words = model.get_convolved_chars(batch)
        convolved_filters.append(convolved_batch_words)

    convolved_filters = torch.cat(convolved_filters, dim=0)
    convolved_filters = convolved_filters.permute(1, 0, 2).detach().cpu().numpy()

    # Reduce filters dimensionality:
    dim_2_filters = []
    for filter_idx in range(convolved_filters.shape[0]):
        dim_2_filter = TSNE(n_components=2, learning_rate='auto',
                            init='random').fit_transform(convolved_filters[filter_idx])
        dim_2_filters.append(dim_2_filter)

    # Save filters plots:
    plt.ioff()
    words = [word for word in vocabulary_data_loader.dataset.vocabulary]
    for filter_idx, _filter in enumerate(dim_2_filters):
        points_2d = [(word[0], word[1]) for word in _filter]
        fig, ax = plt.subplots()
        fig.set_size_inches(80, 80)
        ax.scatter(*zip(*points_2d))
        for index, word in enumerate(words):
            ax.annotate(word, points_2d[index])

        figure_name = f"{plot_path}/{model.tag_type}_filter_{filter_idx+1}"
        fig.savefig(f"{figure_name}.png")
        plt.close(fig)
```

Filter Analysis

Note that this is subset of the words, the words with frequency higher than 10. Its done due to the time takes to compute them all. If needed, I have the figures of all the filters, the below are just some samples.

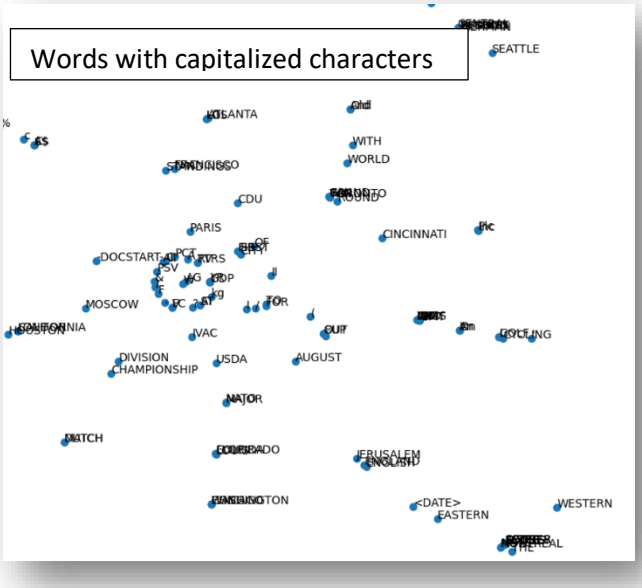
NER Filters

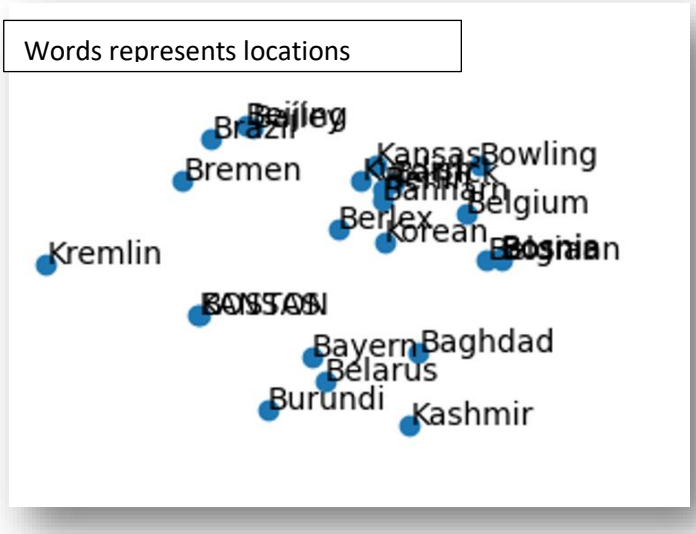


The filters above are the examples of 3 out of 5 filters learned in NER tagging. We can see the different clusters in the different filters.

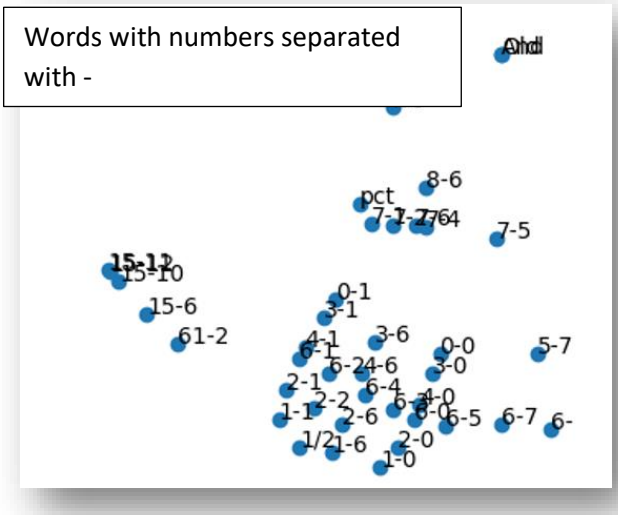
Let's examine some of the clusters to spot the why are the words close to each other.

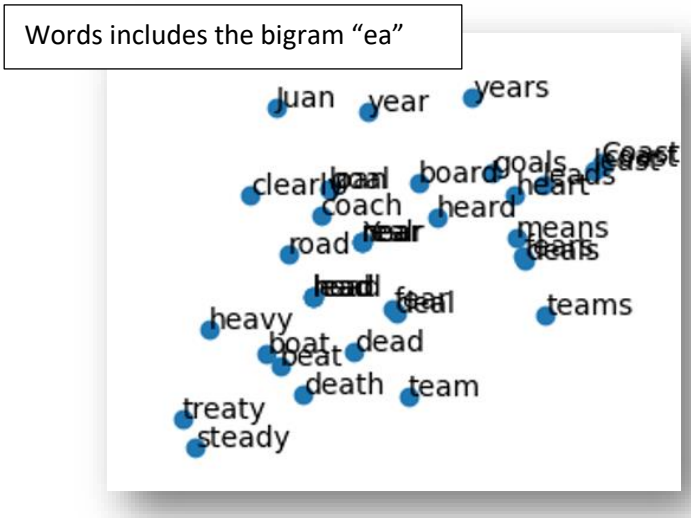
Filter 1



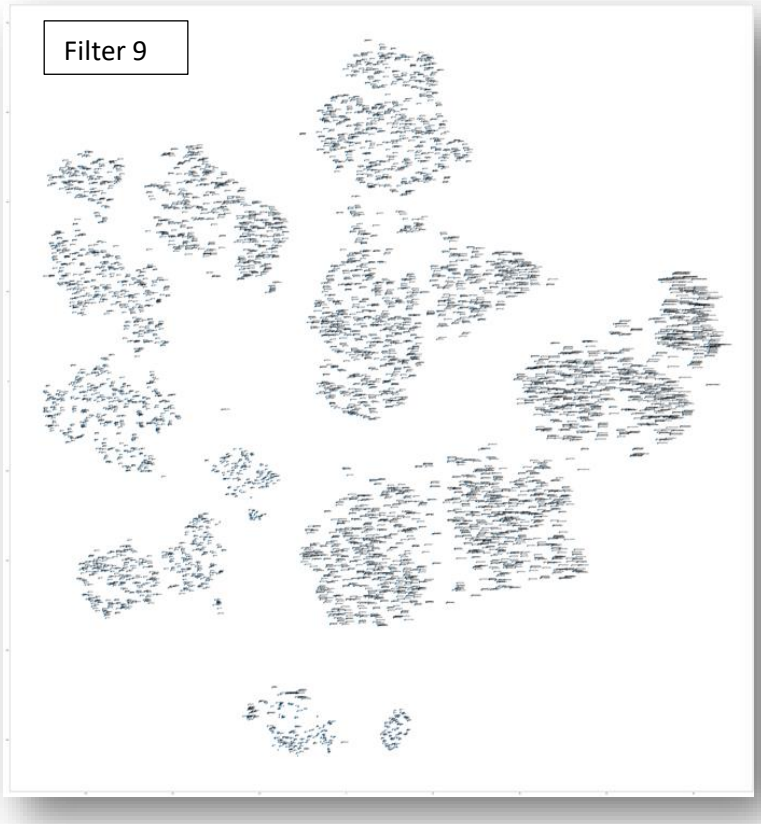


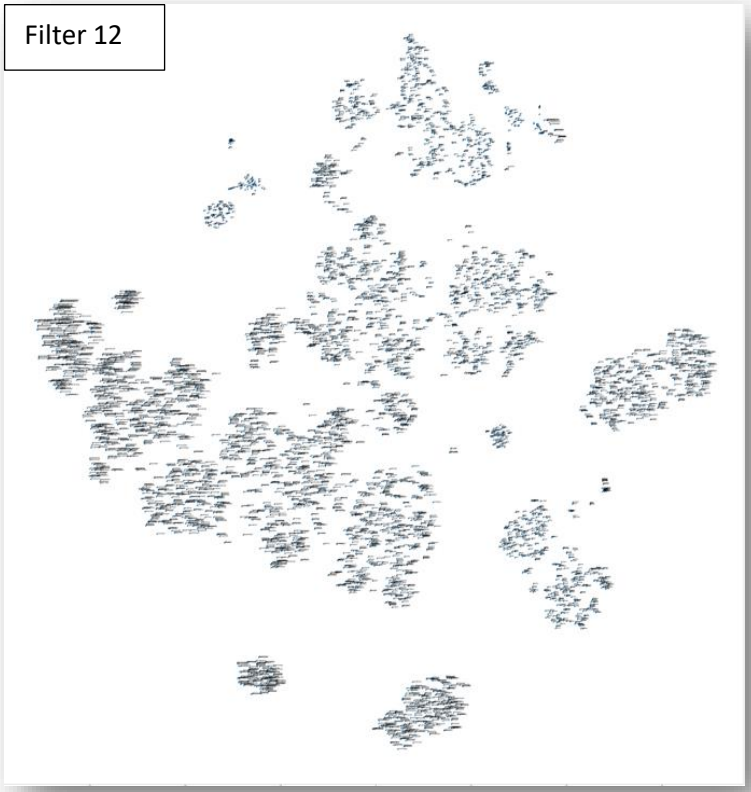
Filter 3





POS Filters

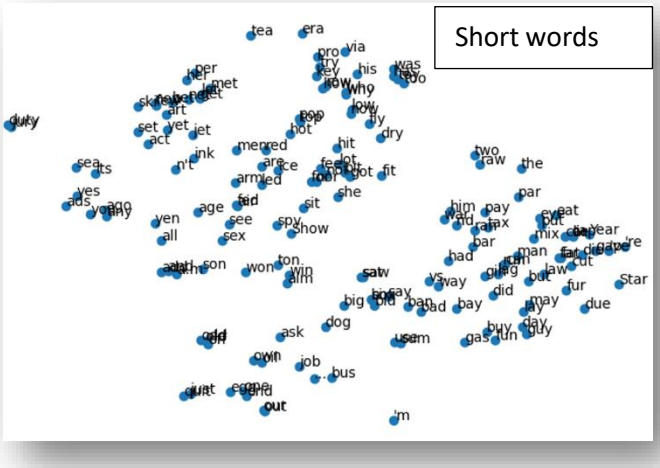


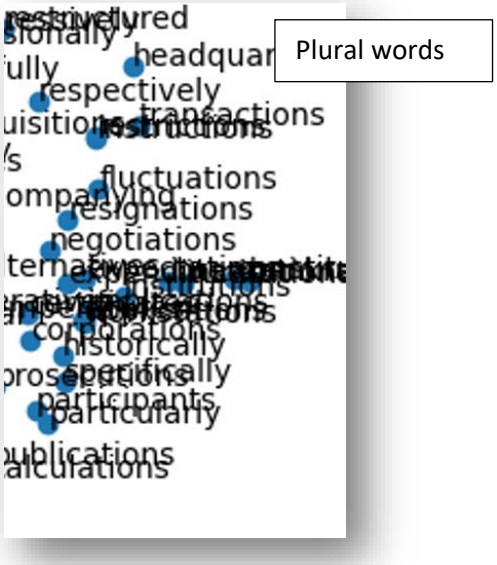


The filters above are the examples of 3 out of 30 filters learned in POS tagging. We can see the different clusters in the different filters.

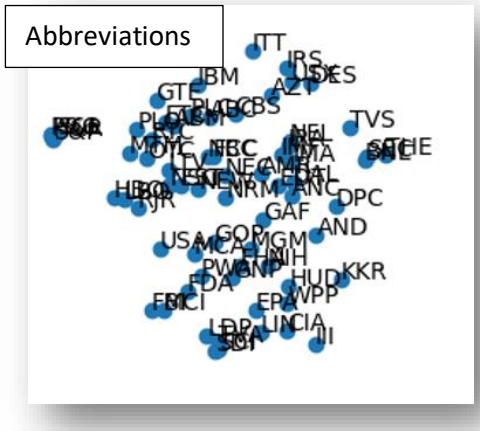
Let's examine some of the clusters to spot the why are the words close to each other.

Filter 9

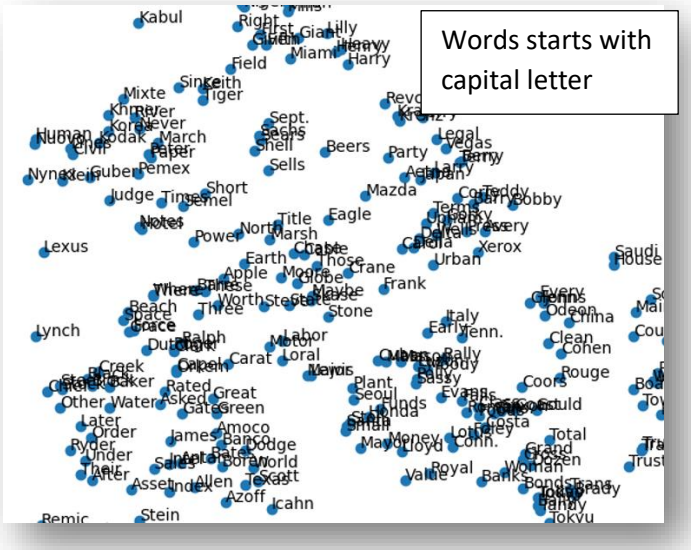




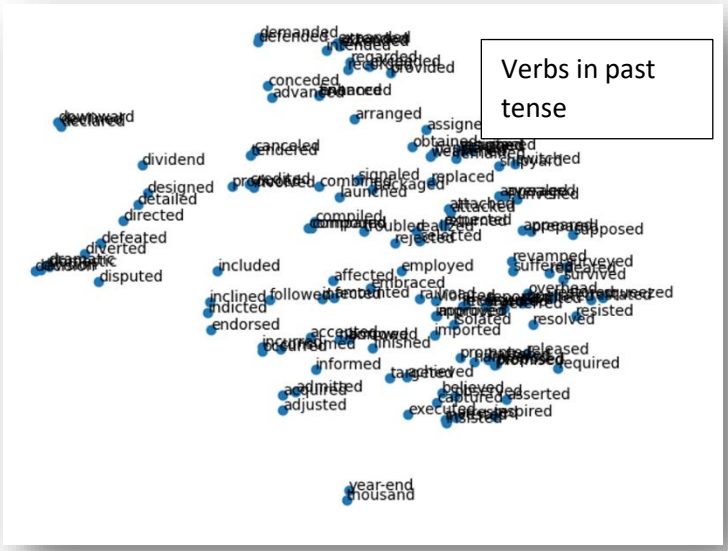
Filter 12



Filter 25



Filter 28 (Not presented above)



NER vs POS Filters

There were many filters, so I am summarizing the ones that I saw. There are some similarities and some differences between the learned filters.

NER filters learned more semantics on the of locations and names; capital worded letters; short words. Etc.

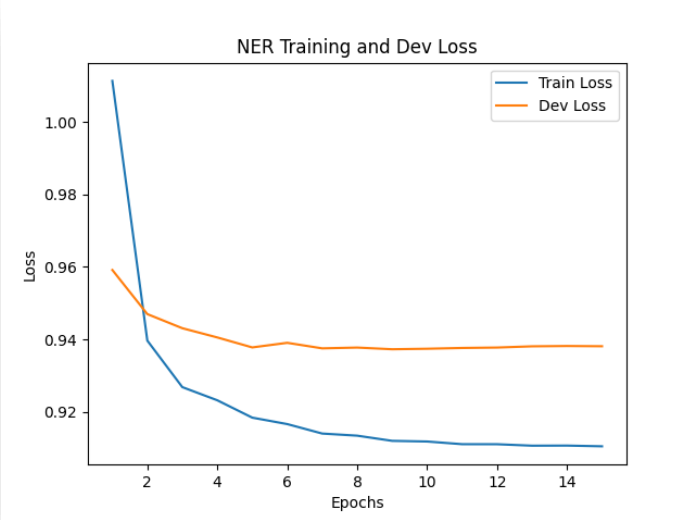
POS filters were able to learn some of more complex structures, such as verbs in past tense, words with plural tense, etc.

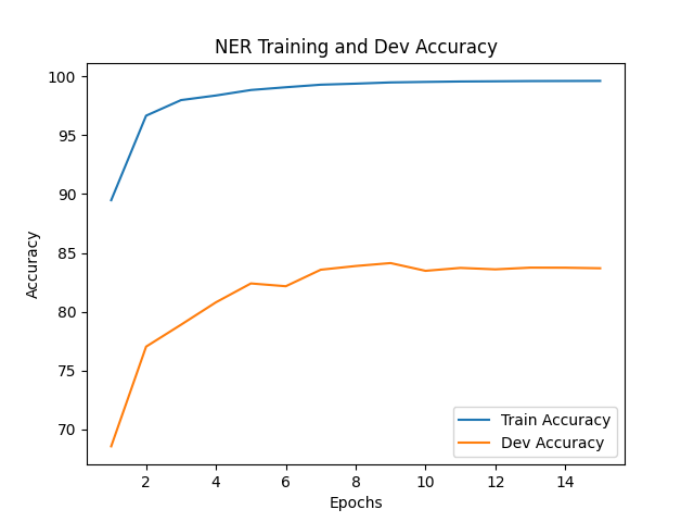
There were also similarities, both were able to learn all capital words, short words. Etc.

Training Graphs

NER Tagging

Note: The calculation for the dev done without considering the common label ‘O’ as requested. The calculation for the train done with it.





POS Tagging