INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Introduction to NLP (CS7.401)

Assignment 2

Submitted by:

Bhanuj Gandhi

2022201068

March 27, 2023

Contents

1 Neural POS Tagging

Q) Design, implement, and train a neural sequence model (RNN, LSTM, GRU, etc.) of your choice to (tokenize and) tag a given sentence with the correct part-of-speech tags. For example, given the input

```
Mary had a little lamb
```

your model should output

```
Mary NOUN
had VERB
a DET
little ADJ
lamb NOUN
```

Note that the part-of-speech tag is separated from each word by a tab \t character.

1.1 Introduction

Neural POS tagging is a method for automatically assigning parts-of-speech (POS) tags to words in a sentence using neural networks. POS tagging is an important task in natural language processing (NLP) because it helps in understanding the syntactic structure of sentences and is used in many downstream NLP tasks, such as text classification and named entity recognition.

1.2 Dataset Overview

The dataset used in this task is the *UD English-Atis* treebank, version 2.11, which includes data specific to the ATIS (Airline Travel Information System) domain. The dataset consists of the following files:

- `en_atis-ud-train.conllu`: This file contains the training set with annotated part-of-speech tags and syntactic dependency relations.
- `en_atis-ud-dev.conllu`: This file contains the development set, which is used for tuning the model hyperparameters and evaluating its performance during training.
- `en_atis-ud-test.conllu`: This file contains the test set, which is used to evaluate the final performance of the trained model.

The files are in the CoNLL-U format, which is a plain text format for representing syntactic dependency trees with annotations.

Each line in these files corresponds to a token in a sentence, and the fields in each line are separated by tabs. The first ten fields contain information about the token, including its index, word form, lemma, part-of-speech tag, and features. The eleventh field contains the index of the token's head in the sentence, and the twelfth field contains the syntactic dependency relation between the token and its head.

2 Methodology

2.1 Model Architecture

1. Embedding Layer
2. Bidirectional LSTM Layer
3. Linear Layer

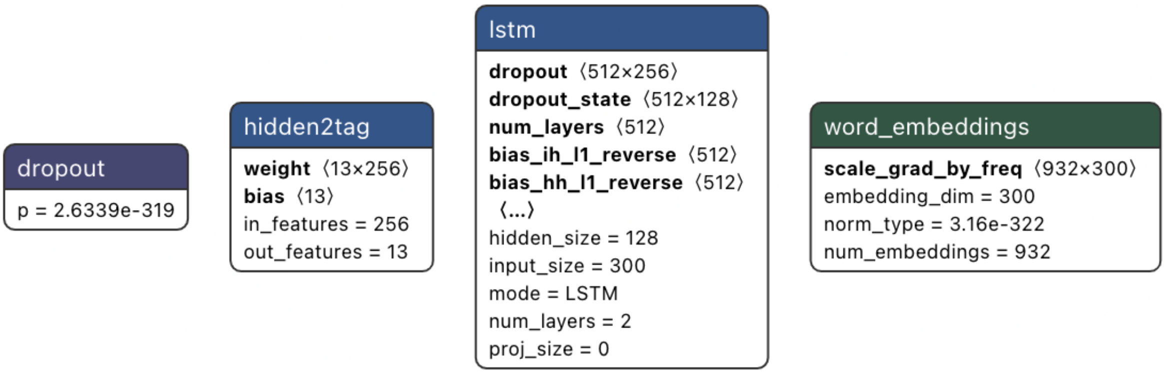


Figure 1: Model Architecture

- I have used Bidirectional LSTM in POS tagging of English language because in English language, the POS tag of a word can depend on the words that come before and after it. By using a bidirectional LSTM, we can take into account both the preceding and following words when predicting the POS tag for a particular word. This helps to capture the context of the sentence better and leads to more accurate POS tagging.
- I have incorporated Dropout in the model since the corpus being used is small, which increases the risk of overfitting. By dropping some of the weights and re-learning them in subsequent iterations, the model can prevent overfitting.

2.2 Implementational Steps

Steps followed to implement the Neural POS Tagger

1. Examine the dataset to understand the available data and its structure in detail.
2. Created sequence of the dataset in order to feed the model

```python
    def prepare_datasets(dataset):
        mod_data = []
        for idx in range(len(dataset)):
            tempword = []
            temptag = []
            for jdx in range(len(dataset[idx])):
                tempword.append(dataset[idx][jdx]["form"])
                temptag.append(dataset[idx][jdx]["upos"])

            mod_data.append([tempword, temptag])
        return mod_data

```

3. Create vocabulary for Train Dataset (both for word tokens as well as Tag Tokens), for this I have used `torchtext vocabulary builder`, which lets us created a dictionary which handles the unknowns.

```
word_vocab = torchtext.vocab.build_vocab_from_iterator(new_list)
word_vocab.insert_token('<unk>', 0)
word_vocab.set_default_index(word_vocab['<unk>'])

```

4. Create model, `pytorch` let's you create model using class inheritence. I have used 3 layers to define my model as defined in Model Architecture.

```
class LSTMTagger(nn.Module):
    def __init__(
        self,
        word_embedding_dim,
        word_hidden_dim,
        vocab_size,
        tagset_size,
    ):
        super(LSTMTagger, self).__init__()
        self.word_hidden_dim = word_hidden_dim
        self.word_embeddings = nn.Embedding(vocab_size, word_embedding_dim)
        self.lstm = nn.LSTM(word_embedding_dim, word_hidden_dim, num_layers = 1, bidirectional = True)

        self.hidden2tag = nn.Linear(word_hidden_dim*2, tagset_size)

        self.dropout = nn.Dropout(0.1)

    def forward(self, sentence):
        embeds = self.dropout(self.word_embeddings(sentence))
        lstm_out, _ = self.lstm(embeds.view(len(sentence), 1, -1))
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores


```

5. To optimise the model during training, I have used the *cross entropy loss function* and the *Adam optimiser* to update the model's parameters based on the gradients calculated during back-propagation. The *cross entropy* loss function is used because it is commonly used for multi-class classification tasks. The Adam optimiser is a popular optimisation algorithm that adapts the learning rate during training to improve convergence.

```
loss_function = nn.CrossEntropyLoss()
optimiser = optim.Adam(model.parameters(), lr=LEARNING_RATE)

```

6. I have experimented with different values of *hyper-parameters* and have decided on some based on my observations. I noticed that the model performed better with smaller values for the embedding size and hidden layer, which may be due to the small size of the corpus. The following are the finalised *hyper-parameters*

```
WORD_EMBEDDING_DIM = 64
WORD_HIDDEN_DIM = 64
EPOCHS = 50
BIDIRECTIONAL = True
DROPOUT = 0.5
LEARNING_RATE = 0.005

```

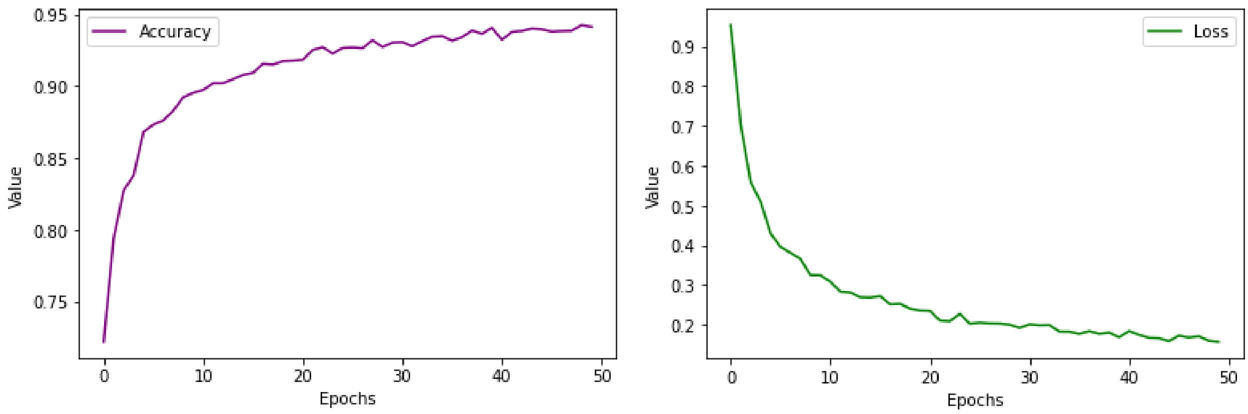
3 Results

3.1 Analysis

I have tried multiple approaches to choose different hyper-parameters and model architecture. A few of the possible analyses are displayed below.

3.1.1 Unidirectional LSTM

As previously mentioned, I tried using unidirectional LSTM, but as part-of-speech tags depend on both the previous and following words in English, we moved to bi-directional LSTM. When Uni-directional LSTM was applied, the model functioned as shown below



(a) Validation Set Accuracy

(b) Validation Set Loss

Figure 2: Validation Set Metrics on Uni-LSTM

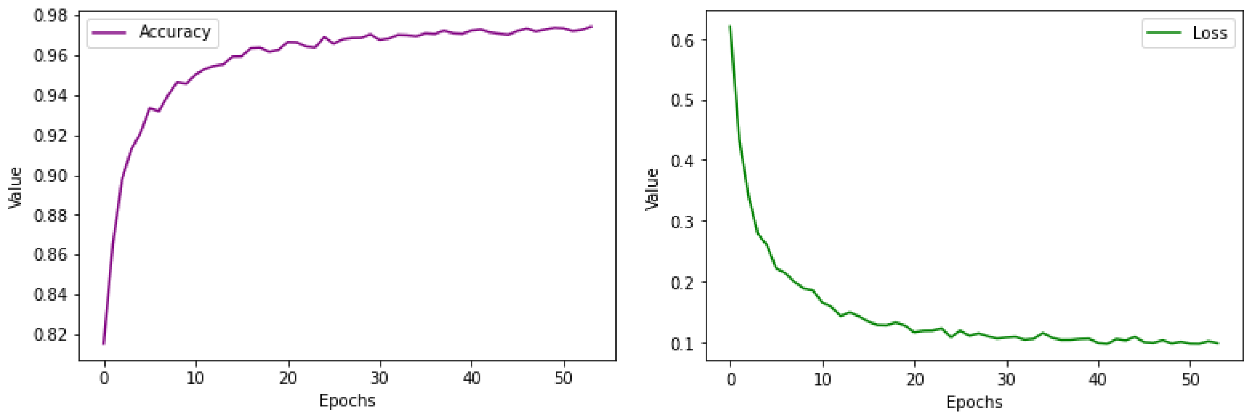
	precision	recall	f1-score	support
ADJ	0.92	0.85	0.89	220
ADP	0.94	0.99	0.97	1434
ADV	0.81	0.62	0.70	76
AUX	0.97	0.93	0.95	256
CCONJ	0.99	0.98	0.99	109
DET	0.84	0.97	0.90	512
INTJ	0.97	1.00	0.99	36
NOUN	0.98	0.97	0.98	1166
NUM	0.82	0.81	0.82	127
PART	0.96	0.95	0.95	56
PRON	0.97	0.77	0.86	392
PROPN	0.97	0.99	0.98	1567
VERB	0.93	0.87	0.90	629
accuracy			0.94	6580
macro avg	0.93	0.90	0.91	6580
weighted avg	0.95	0.94	0.94	6580

Figure 3: Classification Report for Uni-LSTM

3.1.2 Hyper-parameters Tuning

I adhered to the setting of the random search hyper-parameters method. For the Hidden Layer and Embedding Layer, I selected a set of values. I experimented with several different layers. These are some of the visual analyitcs

```
WORD_EMBEDDING_DIM = 32
WORD_HIDDEN_DIM = 32
EPOCHS = 50
BIDIRECTIONAL = True
DROPOUT = 0.5
LEARNING_RATE = 0.005
NUM_LAYERS = 2
```

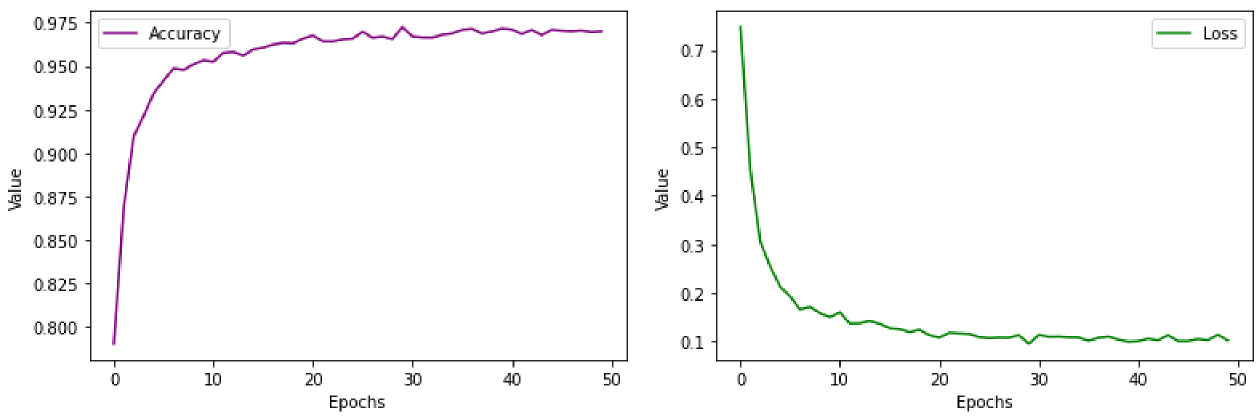


(a) Validation Set Accuracy

(b) Validation Set Loss

Figure 4: Validation Set Metrics on various parameters

```
WORD_EMBEDDING_DIM = 64
WORD_HIDDEN_DIM = 64
EPOCHS = 50
BIDIRECTIONAL = True
DROPOUT = 0.5
LEARNING_RATE = 0.005
NUM_LAYERS = 3
```

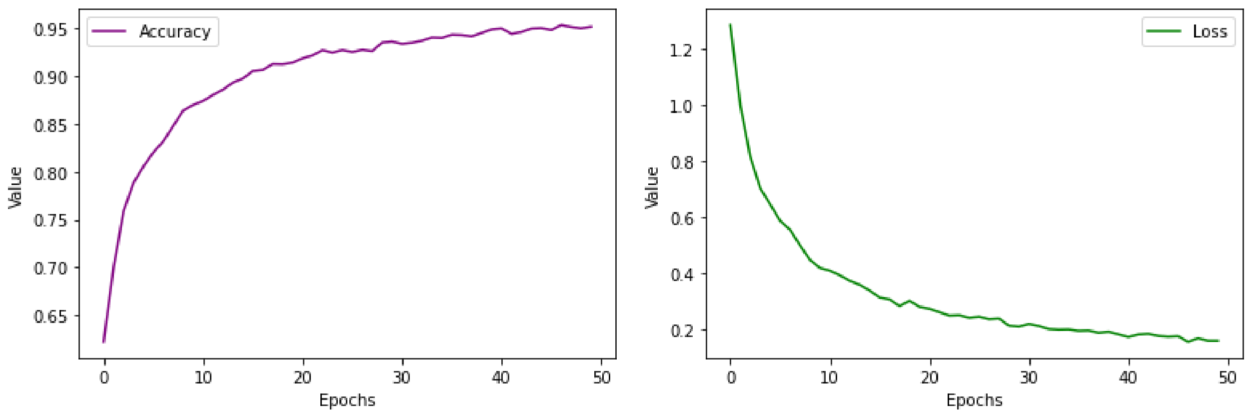


(a) Validation Set Accuracy

(b) Validation Set Loss

Figure 5: Validation Set Metrics on various parameters

```
    WORD_EMBEDDING_DIM = 16
    WORD_HIDDEN_DIM = 16
    EPOCHS = 50
    BIDIRECTIONAL = True
    DROPOUT = 0.65
    LEARNING_RATE = 0.005
    NUM_LAYERS = 2
```



(a) Validation Set Accuracy (b) Validation Set Loss

Figure 6: Validation Set Metrics on various parameters

3.2 Results

After following the approach discussed above, I was able to achieve **98% accuracy on the test dataset**.

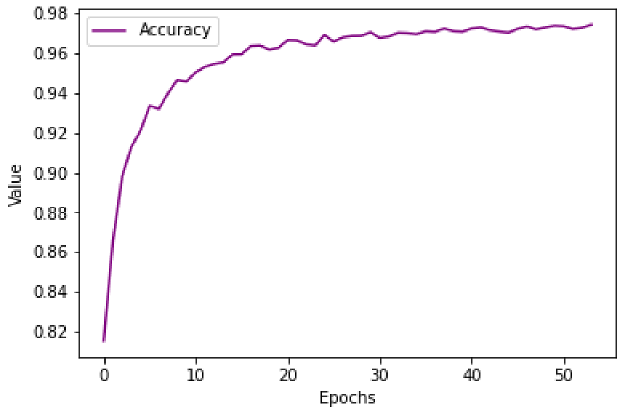
Below is the *classfication report* of the trained model

	precision	recall	f1-score	support
ADJ	0.98	0.90	0.94	1632
ADP	0.97	0.99	0.98	10791
ADV	0.88	0.89	0.88	431
AUX	0.98	0.98	0.98	1732
CCONJ	1.00	0.99	0.99	751
DET	0.96	0.99	0.98	3805
INTJ	0.94	0.99	0.96	319
NOUN	0.99	0.99	0.99	8621
NUM	0.99	0.98	0.99	933
PART	0.90	0.99	0.94	366
PRON	1.00	0.96	0.98	3022
PROPN	1.00	1.00	1.00	11657
VERB	0.99	0.93	0.96	4595
accuracy			0.98	48655
macro avg	0.97	0.97	0.97	48655
weighted avg	0.98	0.98	0.98	48655

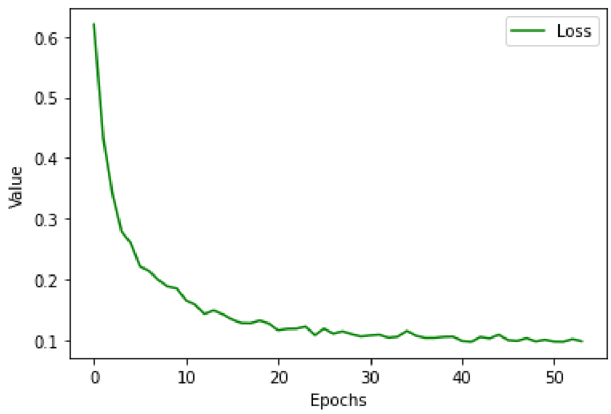
Figure 7: Classification Report

The figures above depict various metrics such as *precision, recall, and F1-score*, which provide a balance between precision and recall by computing their harmonic mean.

The below plots display the validation accuracy, as well as the loss during the training process. These plots provide an analysis of the model's performance with each epoch during training.



(a) Validation Set Accuracy



(b) Validation Set Loss

Figure 8: Validation Set Metrics

The results demonstrate that as the number of epochs increase, there is a reduction in overall loss and an increase in accuracy. This indicates that the model has been effectively trained.

3.3 Sample sentences test

Below are some sample sentences to check how our model is predicting.

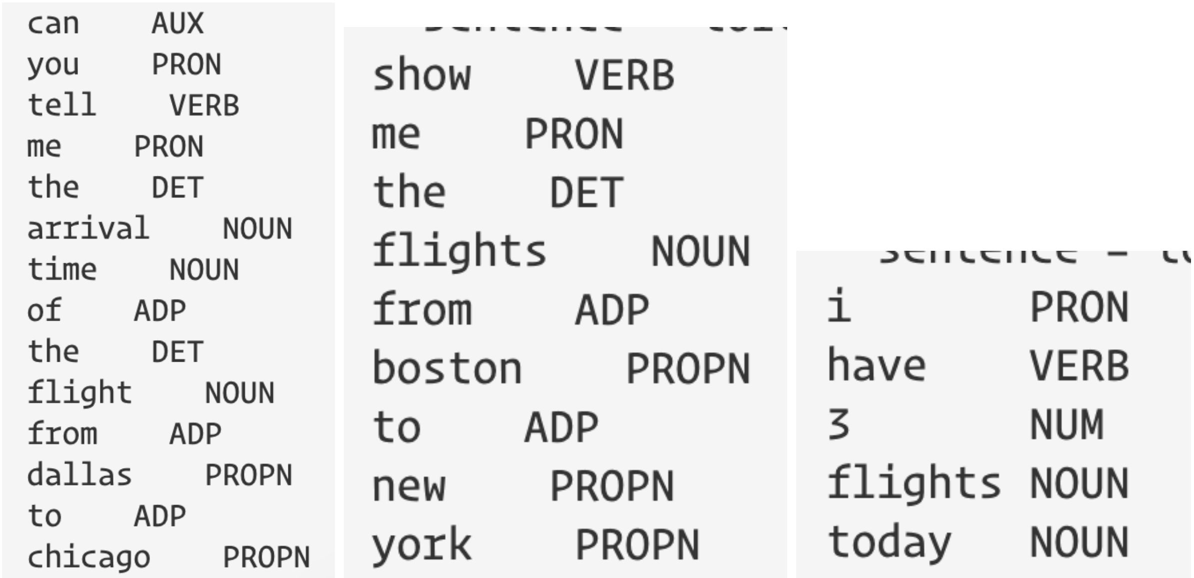


Figure 9: Sample Sentences