INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY

HYDERABAD

Introduction to NLP (CS7.401)

Assignment 2

Submitted by:

Bhanuj Gandhi

2022201068

March 7, 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

Steps followed to implement the Neural POS Tagger

1. Analysis of the dataset, carefully look at what all data is given and how it is formatted.
2. Created sequence of the dataset in order to feed the model

```
    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.
   (a) Embedding Layer
   (b) Bidirectional LSTM Layer
   (c) Linear Layer

   I have also used Dropout, as out corpus is very small and it is possible for model to *overfit* , thus dropping some of the weights and re-learn them in further iterations.

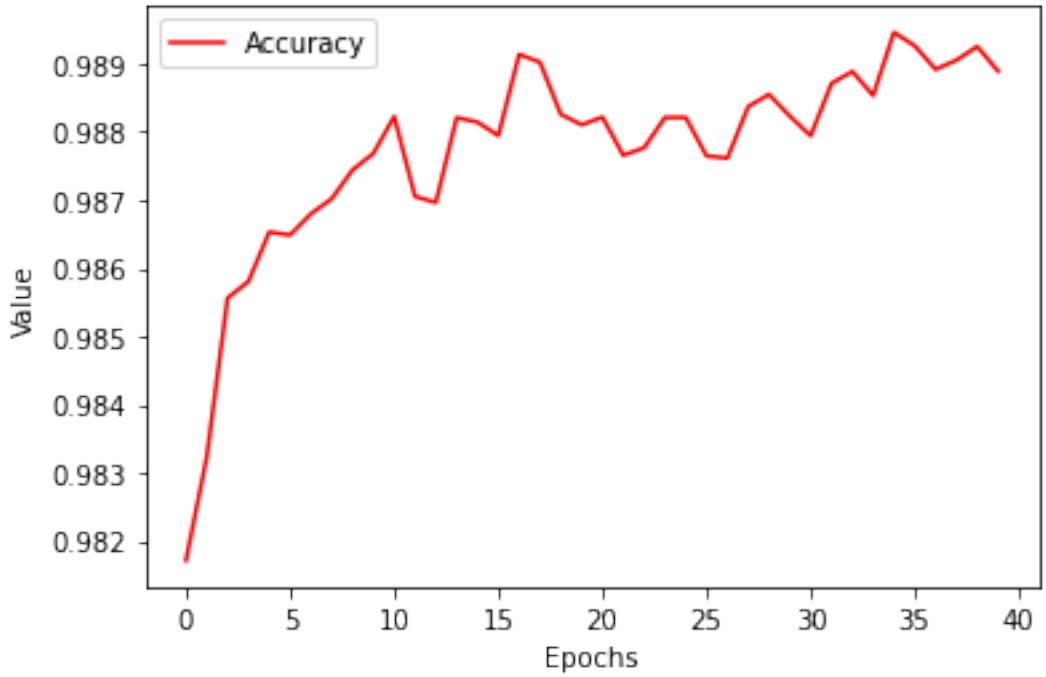
```
    class LSTMTagger(nn.Module):
        def __init__(
            self,
            word_embedding_dim,
            word_hidden_dim,
            vocab_size,
            tagset_size,
        ):
            super(LSTMTagger, self).__init__()
            self.word_hidden_dim = word_hidden_dim
            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)

```

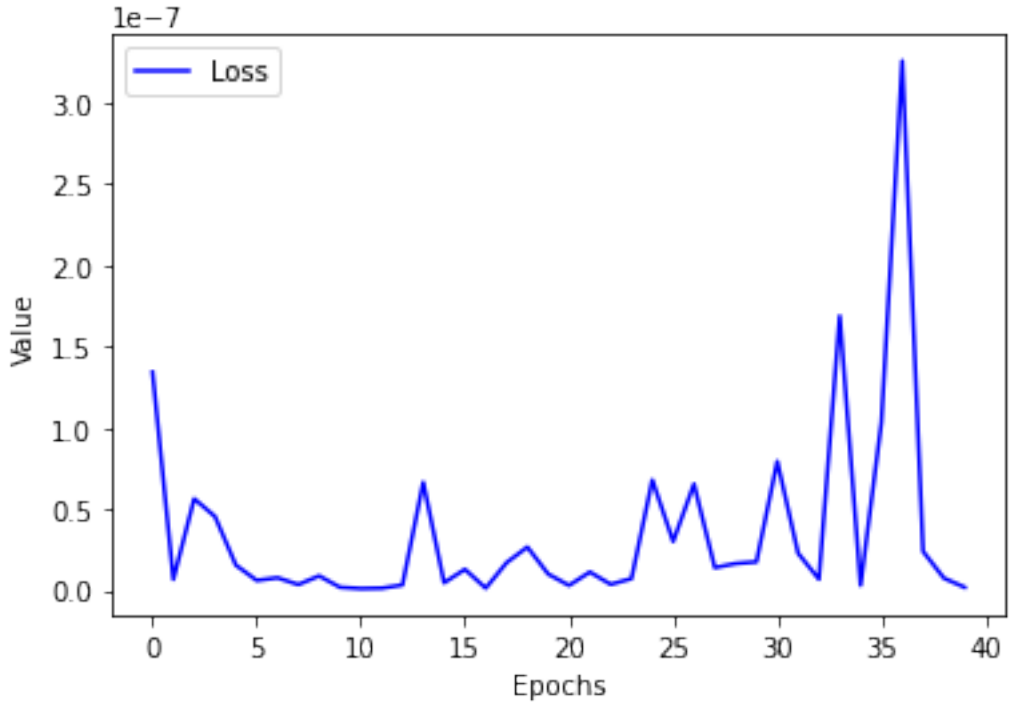
```
        def forward(self, sentence):
            embeds = self.dropout(self.word_embeddings(sentence))
            lstm_out, _ = self.lstm(embeds.view(len(sentence), 1, -1))
            tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
            tag_scores = F.log_softmax(tag_space,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


```

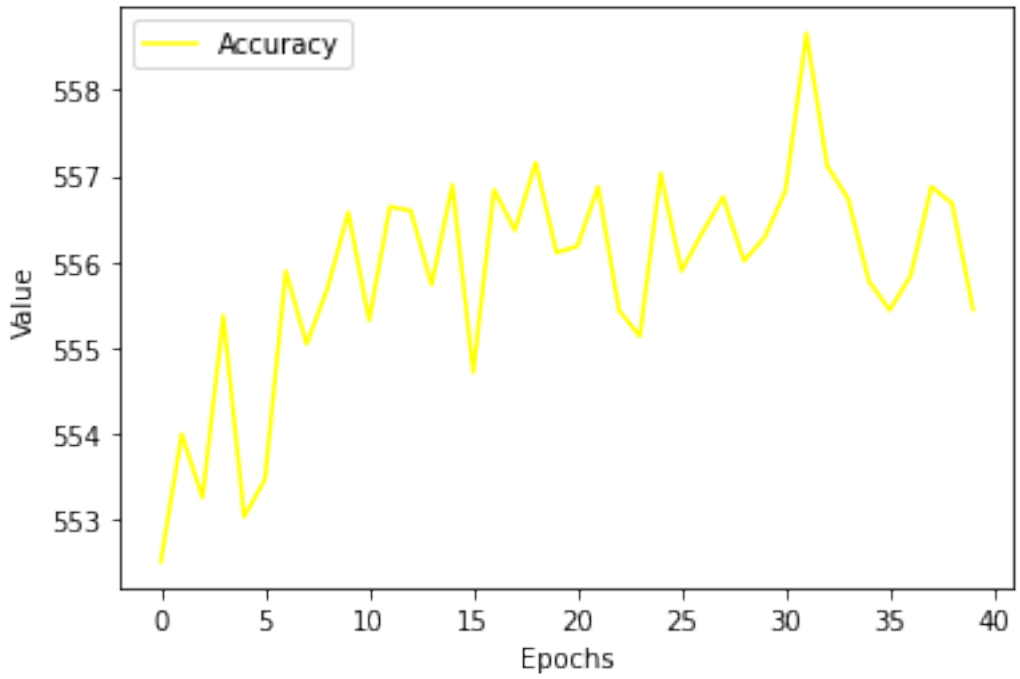
5. To calculate loss and to calculate the *Gradient Descent* in each iteration, I have used Negative Loss Likelihood function and to optimize the learning rate I have used Adam as optimizer.

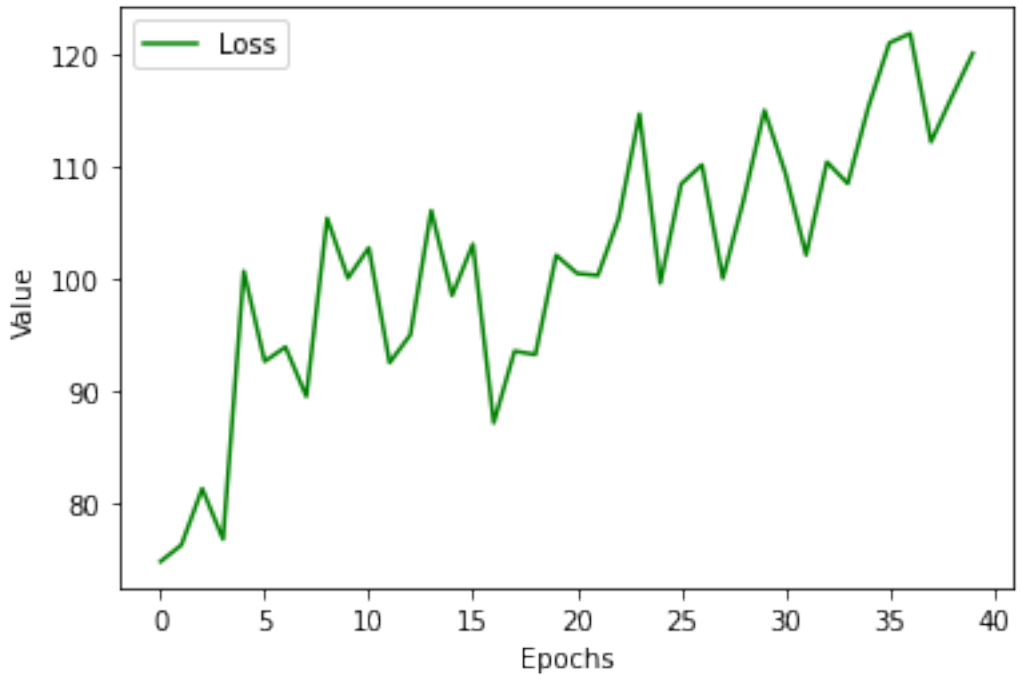
```
    # Define the loss function as the Negative Log Likelihood loss (NLLLoss
    )
    loss_function = nn.NLLLoss()

    # We will be using a simple SGD optimizer
    optimizer = optim.Adam(model.parameters(), lr=0.01)

```

3 Results


Accuracy
0.989
0.988
0.987
0.986
0.985
0.984
0.983
0.982
Value
0
5
10
15
20
25
30
35
40
Epochs


1e−7
Loss
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Value
0
5
10
15
20
25
30
35
40
Epochs


Accuracy
558
557
556
555
554
553
Value
0
5
10
15
20
25
30
35
40
Epochs


Loss
120
110
100
90
80
Value
0
5
10
15
20
25
30
35
40
Epochs