Intoduction To NLP 2023 : Assignment 2

Course Name : Introduction to NLP
Course Code : CS7.401
Semester : Spring'23
Instructor Name : Prof. Manish Shrivastava

Sk. Abukhoyer
Roll : 2021201023
Mtech CSE

IIIT Hyderabad

March 9, 2023

Contents

1 Abstract

This report is about how assignment has been done and analysis of my result. 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. Tune for optimal hyperparameters (embedding size, hidden size, number of layers, learning rate, complexity of decoding network) and report accuracy, precision, recall and F1-score of your trained model. Analyse the results (both the scores as well as the optimal hyperparameters).

2 Dataset

Use the Universal Dependencies dataset, downloadable here. We recommend the files located at **ud-treebanks-v2.11/UD_English-Atis/en_atis-ud-train,dev,test.conllu**. Use the first, second and fourth columns only (word index, lowercase word, and POS tag). The UD dataset does not include punctuation. You may filter the input sentence to remove punctuation before tagging it. Note that many languages' data are downloadable from this resource. We expect a model trained on the English data at least, but you are free to train on other languages in addition.

3 Neural PosTag Model

The process of classifying words into their parts of speech and labeling them accordingly is known as part-of-speech tagging, or simply POS-tagging. The NLTK library has a number of corpora that contain words and their POS tag. I will be using the POS tagged corpora from **ud-treebanks-v2.11/UD_English-Atis**. Part-of-speech (POS) tagging is a fundamental task in natural language processing (NLP) that involves assigning each word in a text its corresponding part of speech tag, such as noun, verb, adjective, etc. In this task, we will design a POS tagging model using the UD_English-Atis dataset from the Universal Dependencies project.

3.1 Vanilla RNN

Next, let's build the RNN model. We're going to use word embeddings to represent the words. Now, while training the model, you can also train the word embeddings along with the network weights. These are often called the embedding weights. While training, the embedding weights will be treated as normal weights of the network which are updated in each iteration.

In the next few sections, we will try the following three RNN models:

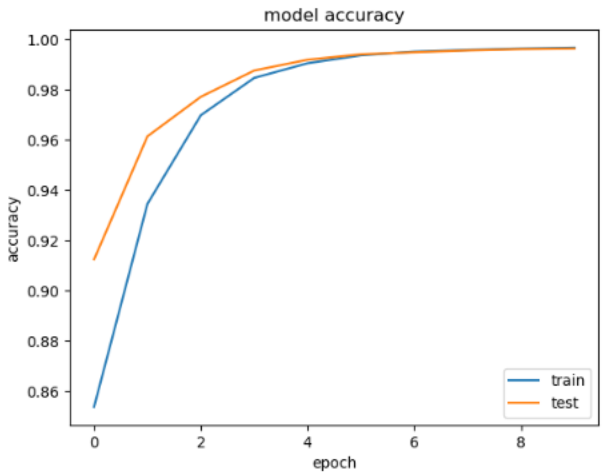
RNN with arbitrarily initialized, untrainable embeddings: In this model, we will initialize the embedding weights arbitrarily. Further, we'll freeze the embeddings, that is, we won't allow the network to train them. RNN with arbitrarily initialized, trainable embeddings: In this model, we'll allow the network to train the embeddings. RNN with trainable word2vec embeddings: In this experiment, we'll use word2vec word embeddings and also allow the network to train them further.

```
Model: "sequential_6"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_6 (Embedding)     (None, 100, 300)          280200

 simple_rnn_3 (SimpleRNN)    (None, 100, 64)           23360

 time_distributed_6 (TimeDis  (None, 100, 15)          975
 tributed)

=================================================================
Total params: 304,535
Trainable params: 304,535
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
34/34 [==============================] - 5s 105ms/step - loss: 0.9184 - acc: 0.8539 - val_loss: 0.3432 - val_acc: 0.9124
Epoch 2/10
34/34 [==============================] - 3s 91ms/step - loss: 0.2647 - acc: 0.9345 - val_loss: 0.2051 - val_acc: 0.9613
Epoch 3/10
34/34 [==============================] - 3s 91ms/step - loss: 0.1588 - acc: 0.9697 - val_loss: 0.1273 - val_acc: 0.9770
Epoch 4/10
34/34 [==============================] - 3s 94ms/step - loss: 0.0972 - acc: 0.9846 - val_loss: 0.0799 - val_acc: 0.9875
Epoch 5/10
34/34 [==============================] - 3s 93ms/step - loss: 0.0615 - acc: 0.9904 - val_loss: 0.0529 - val_acc: 0.9918
Epoch 6/10
34/34 [==============================] - 3s 89ms/step - loss: 0.0416 - acc: 0.9935 - val_loss: 0.0380 - val_acc: 0.9939
Epoch 7/10
34/34 [==============================] - 3s 88ms/step - loss: 0.0306 - acc: 0.9950 - val_loss: 0.0295 - val_acc: 0.9948
Epoch 8/10
34/34 [==============================] - 3s 93ms/step - loss: 0.0240 - acc: 0.9957 - val_loss: 0.0243 - val_acc: 0.9955
Epoch 9/10
34/34 [==============================] - 3s 92ms/step - loss: 0.0198 - acc: 0.9962 - val_loss: 0.0209 - val_acc: 0.9960
Epoch 10/10
34/34 [==============================] - 3s 98ms/step - loss: 0.0169 - acc: 0.9966 - val_loss: 0.0185 - val_acc: 0.9962
```




```
19/19 [==============================] - 0s 14ms/step - loss: 0.0175 - acc: 0.9962
Loss: 0.017508236691355705,
Accuracy: 0.9961603879928589
```

```
              precision    recall  f1-score   support

       PROPN       1.00      1.00      1.00     52020
         ADP       0.98      0.99      0.99      1567
        NOUN       0.98      0.99      0.99      1434
        VERB       0.99      0.98      0.99      1166
         DET       0.98      0.95      0.96       629
        PRON       0.97      0.87      0.92       512
         AUX       0.86      0.97      0.91       392
         ADJ       0.91      0.98      0.94       256
         NUM       0.91      0.97      0.94       220
       CCONJ       0.99      0.87      0.92       127
         ADV       0.99      0.96      0.98       109
        PART       1.00      0.54      0.70        76
        INTJ       0.96      0.80      0.87        56
         SYM       1.00      1.00      1.00        36

    accuracy                           1.00     58600
   macro avg       0.97      0.92      0.94     58600
weighted avg       1.00      1.00      1.00     58600
```

3.2 Bidirection Model

For example, when you want to assign a sentiment score to a piece of text (say a customer review), the network can see the entire review text before assigning them a score. On the other hand, in a task such as predicting the next word given previous few typed words, the network does not have access to the words in the future time steps while predicting the next word. These two types of tasks are called offline and online sequence processing respectively. Now, there is a neat trick you can use with offline tasks — since the network has access to the entire sequence before making predictions, why not use this task to make the network 'look at the future elements in the sequence' while training, hoping that this will make the network learn better?

This is the idea exploited by what is called bidirectional RNNs.

By using bidirectional RNNs, it is almost certain that you'll get better results. However, bidirectional RNNs take almost double the time to train since the number of parameters of the network increase. Therefore, you have a tradeoff between training time and performance. The decision to use a bidirectional RNN depends on the computing resources that you have and the performance you are aiming for.

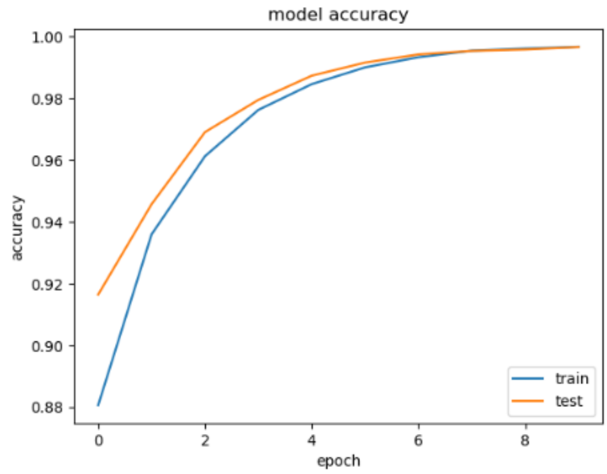
Finally, let's build one more model — a bidirectional LSTM and compare its performance in terms of accuracy and training time as compared to the previous models.

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 embedding_3 (Embedding)     (None, 100, 300)          280200

 bidirectional_1 (Bidirectio  (None, 100, 128)         186880
 nal)

 time_distributed_3 (TimeDis  (None, 100, 15)          1935
 tributed)

=================================================================
Total params: 469,015
Trainable params: 469,015
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
34/34 [==============================] - 15s 337ms/step - loss: 1.0392 - acc: 0.8806 - val_loss: 0.2709 - val_acc: 0.9164
Epoch 2/10
34/34 [==============================] - 11s 323ms/step - loss: 0.2149 - acc: 0.9359 - val_loss: 0.1878 - val_acc: 0.9457
Epoch 3/10
34/34 [==============================] - 11s 311ms/step - loss: 0.1545 - acc: 0.9612 - val_loss: 0.1374 - val_acc: 0.9690
Epoch 4/10
34/34 [==============================] - 10s 303ms/step - loss: 0.1089 - acc: 0.9762 - val_loss: 0.0948 - val_acc: 0.9795
Epoch 5/10
34/34 [==============================] - 11s 321ms/step - loss: 0.0728 - acc: 0.9846 - val_loss: 0.0634 - val_acc: 0.9873
Epoch 6/10
34/34 [==============================] - 10s 305ms/step - loss: 0.0487 - acc: 0.9900 - val_loss: 0.0438 - val_acc: 0.9916
Epoch 7/10
34/34 [==============================] - 10s 303ms/step - loss: 0.0342 - acc: 0.9933 - val_loss: 0.0323 - val_acc: 0.9942
Epoch 8/10
34/34 [==============================] - 10s 308ms/step - loss: 0.0254 - acc: 0.9954 - val_loss: 0.0250 - val_acc: 0.9953
Epoch 9/10
34/34 [==============================] - 10s 302ms/step - loss: 0.0196 - acc: 0.9961 - val_loss: 0.0200 - val_acc: 0.9958
Epoch 10/10
34/34 [==============================] - 10s 307ms/step - loss: 0.0157 - acc: 0.9966 - val_loss: 0.0166 - val_acc: 0.9966
```




```
19/19 [==============================] - 1s 39ms/step - loss: 0.0151 - acc: 0.9971
Loss: 0.015067975968122482,
Accuracy: 0.9970819354057312
```

	precision	recall	f1-score	support
PROPN	1.00	1.00	1.00	52020
ADP	0.99	0.99	0.99	1567
NOUN	0.96	1.00	0.98	1434
VERB	0.99	0.98	0.99	1166
DET	0.98	0.97	0.97	629
PRON	0.98	0.99	0.99	512
AUX	0.99	0.98	0.98	392
ADJ	0.96	1.00	0.98	256
NUM	0.90	0.98	0.94	220
CCONJ	0.97	0.90	0.93	127
ADV	1.00	0.95	0.98	109
PART	1.00	0.54	0.70	76
INTJ	1.00	0.23	0.38	56
SYM	1.00	1.00	1.00	36
accuracy			1.00	58600
macro avg	0.98	0.89	0.91	58600
weighted avg	1.00	1.00	1.00	58600

4 Result & Discussion

Let's now try the third experiment — RNN with trainable word2vec embeddings. Recall that we had loaded the word2vec embeddings in a matrix called embedding_weights.

Using word2vec embeddings is just as easy as including this matrix in the model architecture.

The network architecture is the same as above but instead of starting with an arbitrary embedding matrix, we'll use pre-trained embedding weights (weights = [embedding_weights]) coming from word2vec. The accuracy, in this case, has gone even further to approx 99.04

The bidirectional LSTM did increase the accuracy substantially (considering that the accuracy was already hitting the roof). This shows the power of bidirectional LSTMs. However, this increased accuracy comes at a cost. The time taken was almost double than of a normal LSTM network. The bidirectional LSTM did increase the accuracy substantially (considering that the accuracy was already hitting the roof). This shows the power of bidirectional LSTMs. However, this increased accuracy comes at a cost. The time taken was almost double than of a normal LSTM network.