ECS7001 - Neural Networks and NLP Lab 2: POS Tagging with RNNs

January 28

In this lab we will familiarize ourselves more with Keras by using it to create a deep learning model for part of speech tagging. Most of the labs in this module will be Python-based using Keras.

1. Build a Part-of-Speech Tagger with Keras

In this part we will demonstrate how can we use Keras to build a simple part-of-speech tagger.

1.1 Download and Explore the Data

For training and testing the part-of-speech (PoS) tagger, we will use a corpus from NLTK which is 10% of the penn treebank. First we need to download the data

```
>>> import nltk
>>> nltk.download('treebank')
>>> nltk.download('universal_tagset')
```

After that, you should have the corpus on your hard disk. We now can start to play around with the data. Let's first load the data:

```
>>> from nltk.corpus import treebank
>>> tagged_sentences = treebank.tagged_sents(tagset='universal')
```

We now have the sentences tagged with universal tagset. The universal POS tagset has 12 distinct tags:

- VERB - verbs (all tenses and modes)
- NOUN - nouns (common and proper)
- PRON - pronouns
- ADJ - adjectives
- ADV - adverbs
- ADP - adpositions (prepositions and postpositions)
- CONJ - conjunctions
- DET - determiners
- NUM - cardinal numbers

- PRT - particles or other function words
- X - other: foreign words, typos, abbreviations
- . - punctuation

The sentences are stored as lists of (token, PoS) tag pairs. We can take a look at the first sentence of the data set:

```
>>> print(tagged_sentences[0])
[(u'Pierre', u'NOUN'), (u'Vinken', u'NOUN'), (u',', u'.'), (u'61',
u'NUM'), (u'years', u'NOUN'), (u'old', u'ADJ'), (u',', u'.'), (u'will',
u'VERB'), (u'join', u'VERB'), (u'the', u'DET'), (u'board', u'NOUN'),
(u'as', u'ADP'), (u'a', u'DET'), (u'nonexecutive', u'ADJ'),
(u'director', u'NOUN'), (u'Nov.', u'NOUN'), (u'29', u'NUM'), (u'.',
u'.')]
```

1.2 Prepare the Data

Once we have the dataset we need to prepare the data for building our model. In order to assess the performance of our trained model we need to create a test set in which we don't touch during training. For our corpus we have in total 3914 sentences, your can check the number of sentences in the corpus by:

```
>>> print(len(tagged_sentences))
3914
```

For our task, we use the first 3000 sentences for training and the rest (914 sentences) for testing, we can create the train and test set by:

```
>>> train = tagged_sentences[:3000]
>>> test = tagged_sentences[3000:]
```

And then we need to do some preprocessing to get the data ready for training. In order to feed the data into the neural network, we need to map the actual tokens and tags into integers. First let's create a dictionary for tags:

```
>>> tagset = set([tag for sent in tagged_sentences for token,tag in sent])
>>> tag2ids = {tag:id for id,tag in enumerate(tagset)}
>>> print(tag2ids)
{u'ADV': 0, u'NOUN': 1, u'ADP': 2, u'PRT': 6, u'DET': 4, u'.': 5,
u'PRON': 3, u'NUM': 7, u'X': 8, u'CONJ': 9, u'ADJ': 10, u'VERB': 11}
```

As we can see each of the 12 tags in the universal tagset has been assigned an integer.

To create a dictionary for tokens, we want to replace the lower frequency tokens with an artificial token "<UNK>" (unknown words), as it is important that our model is able to handle tokens not presented in the training set. We replace tokens presented less or equal than 3 times in the training data with the token “<UNK>”. We also add another artificial token “<PAD>” for padding the data when needed. To count the frequency of the tokens we use collections.Counter() method, which output the number of times each word appeared in the training data.

```
>>> import collections
>>> word_counter = collections.Counter([token.lower() for sent in train for token,tag in sent])
```

The following code is version dependent. If you are using Python 2 you can use:

```
>>> print(word_counter.items()[:10])
[(u'equity-purchase', 1), (u'hallwood', 2), (u'foul', 2), (u'del.', 1),
(u'four', 16), (u'jihad', 1), (u'spiders', 1), (u'railing', 3),
(u'city-owned', 1), (u'centimeter', 1)]
```

To run in python 3 you need first to convert the set in a list:

```
>>>print(list(word_counter.items())[:10])
[('pierre', 1), ('vinken', 2), (',', 3779), ('61', 3), ('years', 102),
('old', 27), ('will', 204), ('join', 4), ('the', 3694), ('board', 53)]
```

The following instructions work in both Python 2 and Python 3:

```
>>> vocab = [k for k, v in word_counter.items() if v > 3]
>>> word2ids = {token:id+2 for id, token in enumerate(vocab)}
>>> word2ids['<UNK>'] = 0
>>> word2ids['<PAD>'] = 1
```

The following is also version-dependent as above. The following works with Python 2:

```
>>> print(word2ids.items()[:10])
[(u'limited', 1839), (u'four', 2), (u'asian', 1201), (u'controversial',
619), (u'poorly', 1222), (u'increase', 3), (u'violate', 1840),
(u'eligible', 4), (u'electricity', 5), (u'segments', 1203)]
```

Whereas the following works with Python 3:

```
print(list(word2ids.items())[:10])
[(',', 2), ('years', 3), ('old', 4), ('will', 5), ('join', 6), ('the', 7), ('board', 8), ('as', 9), ('a', 10),
('nonexecutive', 11)]
```

The second step is to convert the tokens and tags into the integers and create the inputs for our model. To do this, we introduce a helper method called preprocessing. The method takes a data set (either train or test) as an import and outputs tri-grams of tokens in the sentences and their corresponding PoS tags (gold labels). The tri-gram for our task is a list of three tokens (the token itself and the left and right neighbors in the sentences). If a token does not have a left/right neighbors (i.e. the first/last token of a sentence) the “<PAD>” will fill the place. In order to be used by keras, the outputs also need to be converted into numpy arrays.

```
>>> import numpy as np
```

The next bit of code is required to make the preprocessing function compatible with both Python 2 (in which xrange is defined) and Python 3 (in which it was renamed range)

```
try:
    # Python 2
    xrange
except NameError:
    # Python 3, xrange is now named range
    xrange = range
```

Here comes the definition of preprocessing:

```
>>> def preprocessing(data):
...   labels = [tag2ids[tag] for sent in data for token,tag in sent]
...   str_data = [['<PAD>' if i==0 else sent[i-1][0], sent[i][0], '<PAD>' if i==len(sent)-1 else
sent[i+1][0]] for sent in data for i in xrange(len(sent))]
...   out_data = [[word2ids[token] if token in word2ids else word2ids['<UNK>'] for token in
item] for item in str_data]
...   return np.asarray(out_data), np.asarray(labels)
```

And we can preprocess both train and test data set:

```
>>> train_data,train_labels = preprocessing(train)
>>> test_data, test_labels = preprocessing(test)
>>> print(train_data[:10])
[[   1    0    0]
 [   0    0 1255]
 [   0 1255    0]
 [1255    0 1139]
 [   0 1139  133]
 [1139  133 1255]
 [ 133 1255 2371]
 [1255 2371 1866]
 [2371 1866  733]
 [1866  733  736]]
```

```
>>> print(train_labels[:10])
[ 1  1  5  7  1 10  5 11 11  4]
```

As we can see, all the words in the sentences are converted into word-indices and also padded as a tri-gram, the labels are tag-indices for each corresponding word.

1.3 Build the Model

We are now ready to build and train our PoS tagger using keras, first let's load keras:

```
>>> from keras import models, layers, optimizers
```

To build the model, we need to decide how many layers to use,how many units per layer and how to combine them. Here we use the simple sequential model to append layers one after another. In total we use four layers:

- The first layer is an embedding layer which will convert the input word-indices into word embeddings by looking up the embedding vector for each word-index. The embedding vectors are randomly initialized and will be learned during the training.
- The second layer is a dropout layer which simply reset a curtain percentage of the values in the embedding vector to 0. Dropout is a useful way to prevent overfitting.
- The third layer is a simple RNN layer we use to encode the tri-grams, we also apply dropouts for the RNN layers as well.
- The last layer is densely connected with an output size of distinct tags. After the softmax activation function the output is a probabilistic distribution over the tags, i.e. how likely the current token belongs to individual tags. During the test, the tag that has the highest probability is assigned to the given token.

```
>>> vocab_size = len(word2ids.keys())
>>> tag_size = len(tag2ids.keys())
>>> model = keras.Sequential()
>>> model.add(layers.Embedding(vocab_size,100,input_shape=[3]))
>>> model.add(layers.Dropout(0.5))
>>> model.add(layers.SimpleRNN(200,dropout=0.2,recurrent_dropout=0.2))
>>> model.add(keras.layers.Dense(tag_size,activation='softmax'))
```

We can take a look of the model structure by:

```
>>> model.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_2 (Embedding)      (None, 3, 100)            243200
_________________________________________________________________
dropout_4 (Dropout)          (None, 3, 100)            0
_________________________________________________________________
simple_rnn_1 (SimpleRNN)     (None, 200)               60200
_________________________________________________________________
dense_4 (Dense)              (None, 12)                2412
=================================================================
Total params: 305,812
Trainable params: 305,812
Non-trainable params: 0
```

1.4 Train and Test the Model

A model needs a loss function and an optimizer for training. Since this is a multi-class classification problem and the model outputs a probabilistic distribution over classes, we will use the “sparse_categorical_crossentropy” loss function:

```
>>> model.compile(optimizer=optimizers.Adam(),
loss='sparse_categorical_crossentropy',metrics=['accuracy'])
```

We train the model for 40 epochs in mini-batches of 1000 samples (tri-grams). This is 40 iterations over 90% samples in the train data. Keras will create a validation set (10% of the train data) for us before starting the training. The validation set is used for evaluating the model during training and will not be used for training the model.

```
>>> history = model.fit(train_data, train_labels, epochs=40, batch_size=1000,
validation_split=0.1,verbose=2)
Train on 69759 samples, validate on 7752 samples
Epoch 1/40
 - 1s - loss: 1.4758 - acc: 0.5663 - val_loss: 0.7335 - val_acc: 0.7802
Epoch 2/40
 - 0s - loss: 0.5967 - acc: 0.8233 - val_loss: 0.4319 - val_acc: 0.8695
Epoch 3/40
 - 0s - loss: 0.4361 - acc: 0.8711 - val_loss: 0.3610 - val_acc: 0.8839
Epoch 4/40
 - 0s - loss: 0.3796 - acc: 0.8871 - val_loss: 0.3308 - val_acc: 0.8964
Epoch 5/40
 - 0s - loss: 0.3482 - acc: 0.8940 - val_loss: 0.3149 - val_acc: 0.8994

...

Epoch 40/40
 - 0s - loss: 0.2223 - acc: 0.9295 - val_loss: 0.2987 - val_acc: 0.9110
```

After training our model, we test it on the test set

```
>>> results = model.evaluate(test_data,test_labels)
>>> print(results)
23165/23165 [==============================] - 2s 80us/step
[0.3011983873537697, 0.9085257931993765]
```

As we can see our simple model achieved an accuracy just over 90%, which is not bad for a simple model like ours. The state-of-the-art PoS taggers usually have an accuracy over 97% for English.

1.5 Visualize the training[1]

model.fit() returns a History object that contains a dictionary with everything that happened during training:

```
>>> history_dict = history.history
>>> print(history_dict.keys())
['acc', 'loss', 'val_acc', 'val_loss']
```

There are four entries: one for each monitored metric during training and validation. We can use these to plot the training and validation loss for comparison, as well as the training and validation accuracy:

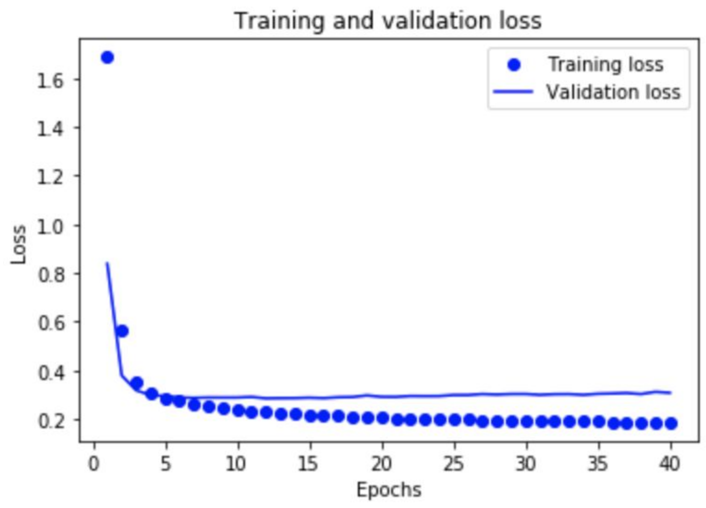
For Mac users we need to add the following two lines to avoid error messages:

```
>>> import matplotlib
>>> matplotlib.use('TkAgg')
```

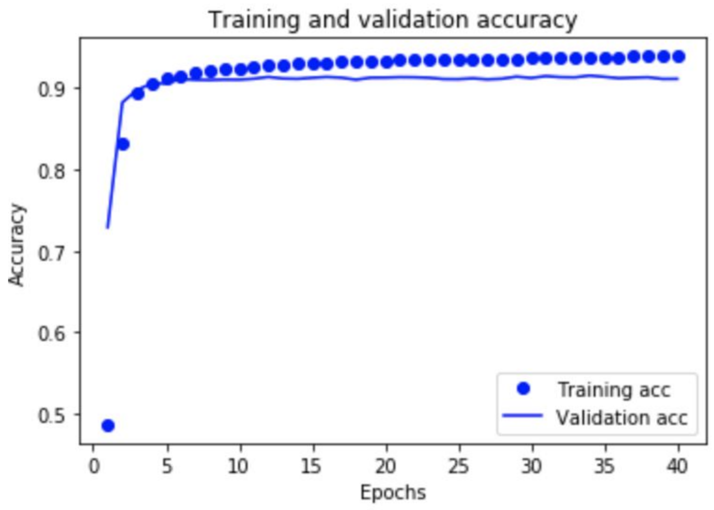
For all users:

```
>>> import matplotlib.pyplot as plt
>>> acc = history.history['acc']
>>> val_acc = history.history['val_acc']
>>> loss = history.history['loss']
>>> val_loss = history.history['val_loss']
>>> epochs = range(1, len(acc) + 1)
>>> plt.plot(epochs, loss, 'bo', label='Training loss') # "bo" is for "blue dot"
>>> plt.plot(epochs, val_loss, 'b', label='Validation loss') # b is for "solid blue line"
>>> plt.title('Training and validation loss')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Loss')
>>> plt.legend()
>>> plt.show()
```

[1] This section is taken from https://www.tensorflow.org/tutorials/keras/basic_text_classification



```
>>> plt.clf()   # clear figure
>>> acc_values = history_dict['acc']
>>> val_acc_values = history_dict['val_acc']
>>> plt.plot(epochs, acc, 'bo', label='Training acc')
>>> plt.plot(epochs, val_acc, 'b', label='Validation acc')
>>> plt.title('Training and validation accuracy')
>>> plt.xlabel('Epochs')
>>> plt.ylabel('Accuracy')
>>> plt.legend()
>>> plt.show()
```


Training and validation accuracy
Training acc
Validation acc
Accuracy
Epochs