Lab 8 Neural Language Model

A language model predicts the next word in the sequence based on the specific words that have come before it in the sequence.

It is also possible to develop language models at the character level using neural networks. The benefit of character-based language models is their small vocabulary and flexibility in handling any words, punctuation, and other document structure. This comes at the cost of requiring larger models that are slower to train.

Nevertheless, in the field of neural language models, character-based models offer a lot of promise for a general, flexible and powerful approach to language modeling.

As a prerequisite for the lab, make sure to pip install:

- keras
- tensorflow
- h5py

Source Text Creation

To start out with, we'll be using a simple nursery rhyme. It's quite short so we can actually train something on your CPU and see relatively interesting results. Please copy and paste the following text in a text file and save it as "rhyme.txt". Place this in the same directory as this jupyter notebook:

```
!pip install tensorflow
!pip install keras
!pip install h5py
```

```
    Requirement already satisfied: tensorflow in /usr/local/lib/python3.7/dist-packa
    Requirement already satisfied: astunparse>=1.6.0 in /usr/local/lib/python3.7/dis
    Requirement already satisfied: opt-einsum>=2.3.2 in /usr/local/lib/python3.7/dis
    Requirement already satisfied: flatbuffers<3.0,>=1.12 in /usr/local/lib/python3.
    Requirement already satisfied: libclang>=9.0.1 in /usr/local/lib/python3.7/dist-
    Requirement already satisfied: wrapt>=1.11.0 in /usr/local/lib/python3.7/dist-pa
    Requirement already satisfied: tensorboard~=2.6 in /usr/local/lib/python3.7/dist
    Requirement already satisfied: absl-py>=0.4.0 in /usr/local/lib/python3.7/dist-p
    Requirement already satisfied: six>=1.12.0 in /usr/local/lib/python3.7/dist-pack
    Requirement already satisfied: h5py>=2.9.0 in /usr/local/lib/python3.7/dist-pack
    Requirement already satisfied: tensorflow-estimator<2.8,~=2.7.0rc0 in /usr/local
    Requirement already satisfied: tensorflow-io-gcs-filesystem>=0.21.0 in /usr/loca
    Requirement already satisfied: gast<0.5.0,>=0.2.1 in /usr/local/lib/python3.7/di
    Requirement already satisfied: keras-preprocessing>=1.1.1 in /usr/local/lib/pyth
```

```
Requirement already satisfied: google-pasta>=0.1.1 in /usr/local/lib/python3.7/d
Requirement already satisfied: keras<2.8,>=2.7.0rc0 in /usr/local/lib/python3.7/
Requirement already satisfied: grpcio<2.0,>=1.24.3 in /usr/local/lib/python3.7/d
Requirement already satisfied: typing-extensions>=3.6.6 in /usr/local/lib/python
Requirement already satisfied: protobuf>=3.9.2 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: wheel<1.0,>=0.32.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: termcolor>=1.1.0 in /usr/local/lib/python3.7/dist
Requirement already satisfied: numpy>=1.14.5 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-
Requirement already satisfied: google-auth-oauthlib<0.5,>=0.4.1 in /usr/local/li
Requirement already satisfied: tensorboard-data-server<0.7.0,>=0.6.0 in /usr/loc
Requirement already satisfied: markdown>=2.6.8 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: setuptools>=41.0.0 in /usr/local/lib/python3.7/di
Requirement already satisfied: tensorboard-plugin-wit>=1.6.0 in /usr/local/lib/p
Requirement already satisfied: werkzeug>=0.11.15 in /usr/local/lib/python3.7/dis
Requirement already satisfied: google-auth<3,>=1.6.3 in /usr/local/lib/python3.7
Requirement already satisfied: requests<3,>=2.21.0 in /usr/local/lib/python3.7/d
Requirement already satisfied: rsa<5,>=3.1.4 in /usr/local/lib/python3.7/dist-pa
Requirement already satisfied: cachetools<5.0,>=2.0.0 in /usr/local/lib/python3.
Requirement already satisfied: pyasn1-modules>=0.2.1 in /usr/local/lib/python3.7
Requirement already satisfied: requests-oauthlib>=0.7.0 in /usr/local/lib/python
Requirement already satisfied: importlib-metadata in /usr/local/lib/python3.7/di
Requirement already satisfied: pyasn1<0.5.0,>=0.4.6 in /usr/local/lib/python3.7/
Requirement already satisfied: chardet<4,>=3.0.2 in /usr/local/lib/python3.7/dis
Requirement already satisfied: idna<3,>=2.5 in /usr/local/lib/python3.7/dist-pac
Requirement already satisfied: urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1 in /usr/l
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.7/di
Requirement already satisfied: oauthlib>=3.0.0 in /usr/local/lib/python3.7/dist-
Requirement already satisfied: zipp>=0.5 in /usr/local/lib/python3.7/dist-packag
Requirement already satisfied: keras in /usr/local/lib/python3.7/dist-packages (
Requirement already satisfied: h5py in /usr/local/lib/python3.7/dist-packages (3
Requirement already satisfied: cached-property in /usr/local/lib/python3.7/dist-
Requirement already satisfied: numpy>=1.14.5 in /usr/local/lib/python3.7/dist-pa
```

```
s='Sing a song of sixpence,\
A pocket full of rye.\
Four and twenty blackbirds,\
Baked in a pie.\
When the pie was opened\
The birds began to sing;\
Wasn’t that a dainty dish,\
To set before the king.\
The king was in his counting house,\
Counting out his money;\
The queen was in the parlour,\
Eating bread and honey.\
The maid was in the garden,\
Hanging out the clothes,\
When down came a blackbird\
And pecked off her nose.'

with open('rhymes.txt','w') as f:
  f.write(s)
```

```
Sing a song of sixpence,
A pocket full of rye.
Four and twenty blackbirds,
Baked in a pie.

When the pie was opened
The birds began to sing;
Wasn’t that a dainty dish,
To set before the king.

The king was in his counting house,
Counting out his money;
The queen was in the parlour,
Eating bread and honey.

The maid was in the garden,
Hanging out the clothes,
When down came a blackbird
And pecked off her nose.
```

Sequence Generation

A language model must be trained on the text, and in the case of a character-based language model, the input and output sequences must be characters.

The number of characters used as input will also define the number of characters that will need to be provided to the model in order to elicit the first predicted character.

After the first character has been generated, it can be appended to the input sequence and used as input for the model to generate the next character.

Longer sequences offer more context for the model to learn what character to output next but take longer to train and impose more burden on seeding the model when generating text.

We will use an arbitrary length of 10 characters for this model.

There is not a lot of text, and 10 characters is a few words.

We can now transform the raw text into a form that our model can learn; specifically, input and output sequences of characters.

```
#load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text
# save tokens to file, one dialog per line
def save_doc(lines, filename):
    data = '\n'.join(lines)
    file = open(filename, 'w')
    file.write(data)
    file.close()


#load text
raw_text = load_doc('rhymes.txt')
print(raw_text)

# clean
tokens = raw_text.split()
raw_text = ' '.join(tokens)

# organize into sequences of characters
length = 10
sequences = list()
for i in range(length, len(raw_text)):
    # select sequence of tokens
    seq = raw_text[i-length:i+1]
    # store
    sequences.append(seq)
print('Total Sequences: %d' % len(sequences))
```

```
Sing a song of sixpence,A pocket full of rye.Four and twenty blackbirds,Baked in
Total Sequences: 384
```

```
# save sequences to file
out_filename = 'char_sequences.txt'
save_doc(sequences, out_filename)
```

Train a Model

In this section, we will develop a neural language model for the prepared sequence data.

The model will read encoded characters and predict the next character in the sequence. A Long Short-Term Memory recurrent neural network hidden layer will be used to learn the context from the input sequence in order to make the predictions.

```
from numpy import array
from pickle import dump
from tensorflow.keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

# load doc into memory
def load_doc(filename):
    # open the file as read only
    file = open(filename, 'r')
    # read all text
    text = file.read()
    # close the file
    file.close()
    return text


# load

in_filename = 'char_sequences.txt'
raw_text = load_doc(in_filename)
lines = raw_text.split('\n')
```

The sequences of characters must be encoded as integers.This means that each unique character will be assigned a specific integer value and each sequence of characters will be encoded as a sequence of integers. We can create the mapping given a sorted set of unique characters in the raw input data. The mapping is a dictionary of character values to integer values.

Next, we can process each sequence of characters one at a time and use the dictionary mapping to look up the integer value for each character. The result is a list of integer lists.

We need to know the size of the vocabulary later. We can retrieve this as the size of the dictionary mapping.

```
# integer encode sequences of characters
chars = sorted(list(set(raw_text)))
mapping = dict((c, i) for i, c in enumerate(chars))
sequences = list()
for line in lines:
    # integer encode line
    encoded_seq = [mapping[char] for char in line]
    # store
    sequences.append(encoded_seq)

# vocabulary size
vocab_size = len(mapping)
print('Vocabulary Size: %d' % vocab_size)
```

```
# separate into input and output
sequences = array(sequences)
X, y = sequences[:,:-1], sequences[:,-1]
sequences = [to_categorical(x, num_classes=vocab_size) for x in X]
X = array(sequences)
y = to_categorical(y, num_classes=vocab_size)
```

```
    Vocabulary Size: 38
```

The model is defined with an input layer that takes sequences that have 10 time steps and 38 features for the one hot encoded input sequences. Rather than specify these numbers, we use the second and third dimensions on the X input data. This is so that if we change the length of the sequences or size of the vocabulary, we do not need to change the model definition.

The model has a single LSTM hidden layer with 75 memory cells. The model has a fully connected output layer that outputs one vector with a probability distribution across all characters in the vocabulary. A softmax activation function is used on the output layer to ensure the output has the properties of a probability distribution.

The model is learning a multi-class classification problem, therefore we use the categorical log loss intended for this type of problem. The efficient Adam implementation of gradient descent is used to optimize the model and accuracy is reported at the end of each batch update. The model is fit for 50 training epochs.

To Do:

- Try different numbers of memory cells
- Try different types and amounts of recurrent and fully connected layers
- Try different lengths of training epochs
- Try different sequence lengths and pre-processing of data
- Try regularization techniques such as Dropout

```
# define model
model = Sequential()
model.add(LSTM(75, input_shape=(X.shape[1], X.shape[2])))
model.add(Dense(vocab_size, activation='softmax'))
print(model.summary())
# compile model
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy']
# fit model
history=model.fit(X, y, epochs=100)
```

```
    12/12 [==============================] - 0s 8ms/step - loss: 0.6730 - accuracy:
    Epoch 72/100
    12/12 [==============================] - 0s 8ms/step - loss: 0.6410 - accuracy:
    Epoch 73/100
    12/12 [==============================] - 0s 8ms/step - loss: 0.6170 - accuracy:
```

```
12/12 [==============================] - 0s 8ms/step - loss: 0.6170 - accuracy:
Epoch 74/100
12/12 [==============================] - 0s 8ms/step - loss: 0.5922 - accuracy:
Epoch 75/100
12/12 [==============================] - 0s 9ms/step - loss: 0.5723 - accuracy:
Epoch 76/100
12/12 [==============================] - 0s 8ms/step - loss: 0.5464 - accuracy:
Epoch 77/100
12/12 [==============================] - 0s 7ms/step - loss: 0.5290 - accuracy:
Epoch 78/100
12/12 [==============================] - 0s 7ms/step - loss: 0.5076 - accuracy:
Epoch 79/100
12/12 [==============================] - 0s 8ms/step - loss: 0.4947 - accuracy:
Epoch 80/100
12/12 [==============================] - 0s 8ms/step - loss: 0.4761 - accuracy:

Epoch 81/100
12/12 [==============================] - 0s 8ms/step - loss: 0.4560 - accuracy:
Epoch 82/100
12/12 [==============================] - 0s 8ms/step - loss: 0.4353 - accuracy:
Epoch 83/100
12/12 [==============================] - 0s 8ms/step - loss: 0.4182 - accuracy:
Epoch 84/100
12/12 [==============================] - 0s 8ms/step - loss: 0.3965 - accuracy:
Epoch 85/100
12/12 [==============================] - 0s 7ms/step - loss: 0.3831 - accuracy:
Epoch 86/100
12/12 [==============================] - 0s 9ms/step - loss: 0.3721 - accuracy:
Epoch 87/100
12/12 [==============================] - 0s 7ms/step - loss: 0.3556 - accuracy:
Epoch 88/100
12/12 [==============================] - 0s 8ms/step - loss: 0.3415 - accuracy:
Epoch 89/100
12/12 [==============================] - 0s 8ms/step - loss: 0.3307 - accuracy:
Epoch 90/100
12/12 [==============================] - 0s 8ms/step - loss: 0.3183 - accuracy:
Epoch 91/100
12/12 [==============================] - 0s 7ms/step - loss: 0.3083 - accuracy:
Epoch 92/100
12/12 [==============================] - 0s 9ms/step - loss: 0.2971 - accuracy:
Epoch 93/100
12/12 [==============================] - 0s 8ms/step - loss: 0.2852 - accuracy:
Epoch 94/100
12/12 [==============================] - 0s 7ms/step - loss: 0.2694 - accuracy:
Epoch 95/100
12/12 [==============================] - 0s 8ms/step - loss: 0.2594 - accuracy:
Epoch 96/100
12/12 [==============================] - 0s 8ms/step - loss: 0.2499 - accuracy:
Epoch 97/100
12/12 [==============================] - 0s 7ms/step - loss: 0.2393 - accuracy:
Epoch 98/100
12/12 [==============================] - 0s 8ms/step - loss: 0.2346 - accuracy:
Epoch 99/100
12/12 [==============================] - 0s 8ms/step - loss: 0.2275 - accuracy:
Epoch 100/100
12/12 [==============================] - 0s 7ms/step - loss: 0.2193 - accuracy:
```

```
# save the model to file
model.save('model.h5')
# save the mapping
dump(mapping, open('mapping.pkl', 'wb'))
```

Generating Text

We must provide sequences of 10 characters as input to the model in order to start the generation process. We will pick these manually. A given input sequence will need to be prepared in the same way as preparing the training data for the model.

```
from pickle import load
import numpy as np
from keras.models import load_model
from tensorflow.keras.utils import to_categorical
from keras.preprocessing.sequence import pad_sequences

# generate a sequence of characters with a language model
def generate_seq(model, mapping, seq_length, seed_text, n_chars):
    in_text = seed_text
    # generate a fixed number of characters
    for _ in range(n_chars):
        # encode the characters as integers
        encoded = [mapping[char] for char in in_text]
        # truncate sequences to a fixed length
        encoded = pad_sequences([encoded], maxlen=seq_length, truncating='pre')
        # one hot encode
        encoded = to_categorical(encoded, num_classes=len(mapping))
        # predict character
        yhat = np.argmax(model.predict(encoded), axis=-1)
        # reverse map integer to character
        out_char = ''
        for char, index in mapping.items():
            if index == yhat:
                out_char = char
                break
        # append to input
        in_text += char
    return in_text

# load the model
model = load_model('model.h5')
# load the mapping
mapping = load(open('mapping.pkl', 'rb'))
```

Running the example generates three sequences of text.

The first is a test to see how the model does at starting from the beginning of the rhyme. The second is a test to see how well it does at beginning in the middle of a line. The final example is a test to see how well it does with a sequence of characters never seen before.

```
# test start of rhyme
print(generate_seq(model, mapping, 10, 'Sing a son', 20))
# test mid-line
print(generate_seq(model, mapping, 10, 'king was i', 20))
# test not in original
print(generate_seq(model, mapping, 10, 'hello worl', 20))
```

```
Sing a song of sixpence,A pock
king was in his counting house
hello worlsWWae dththi bldrre
```

If the results aren't satisfactory, try out the suggestions above or these below:

- Padding. Update the example to provides sequences line by line only and use padding to fill out each sequence to the maximum line length.
- Sequence Length. Experiment with different sequence lengths and see how they impact the behavior of the model.
- Tune Model. Experiment with different model configurations, such as the number of memory cells and epochs, and try to develop a better model for fewer resources.

Deliverables to receive credit

1. (4 points) Optimize the cells above to tune the model so that it generates text that closely resembles the orginal line from the rhyme, or at least generates sensible words. It's okay if the third example using unseen text still looks somewhat strange though. Again, this is a toy problem, as language models require a lot of computation. This toy problem is great for rapid experimentation to explore different aspects of deep learning language models.
2. (3 points) Write a function to split the text corpus file into training and validation and pipe the validation data into the model.fit() function to be able to track validation error per epoch. Lookup Keras documentation to see how this is handled.
3. (3 points) Write a summary (methods and results) in the cells below of the different things you applied. You must include your intuitions behind what did work and what did not work well.
4. (Extra Credit 2.5 points) Do something even more interesting. Try a different source text. Train a word-level model. We'll leave it up to your creativity to explore and write a summary of your methods and results.

Question 1:

1. I increase the epoch sizes to 500. The accuracy improves to 0.9948 from 0.9896 and the loss decreases to 0.0092 from 0.2193.
2. Added a mask layer and set the mask_value to 1.0

-loss: 0.0049 - accuracy: 0.9974

```
# Using Dropout to do regularization
# import tensorflow as tf
# tf.random.set_seed(0)
# layer = tf.keras.layers.Dropout(.2, input_shape=(2,))
# outputs = layer(X, training=True)
# print(outputs)

#Using Dropout to do regularization to the dataset, then the accuracy improves from 0.
#I use dropout to do regularization decreases loss to 0.0049 and increases accuracy to


# define model
# from keras.layers import Masking

model1 = Sequential()
model1.add(Masking(mask_value=1.0))  #masking layer. loss: 0.0097 - accuracy: 0.9948.
model1.add(LSTM(75, input_shape=(X.shape[1], X.shape[2])))  #I changed the numbers of
model1.add(Dense(vocab_size, activation='softmax'))      #loss: 0.0090 - accuracy: 0.9

# print(model1.summary())
# compile model
model1.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy']
# fit model
history1=model1.fit(X, y, epochs=500)
```

```
    12/12 [==============================] - 0s 12ms/step - loss: 0.0099 - accuracy:
    Epoch 472/500
    12/12 [==============================] - 0s 12ms/step - loss: 0.0099 - accuracy:
    Epoch 473/500
    12/12 [==============================] - 0s 12ms/step - loss: 0.0102 - accuracy:
    Epoch 474/500
    12/12 [==============================] - 0s 12ms/step - loss: 0.0103 - accuracy:
    Epoch 475/500
    12/12 [==============================] - 0s 12ms/step - loss: 0.0104 - accuracy:
    Epoch 476/500
    12/12 [==============================] - 0s 12ms/step - loss: 0.0104 - accuracy:
    Epoch 477/500
    12/12 [==============================] - 0s 13ms/step - loss: 0.0103 - accuracy:
    Epoch 478/500
    12/12 [==============================] - 0s 12ms/step - loss: 0.0099 - accuracy:
    Epoch 479/500
    12/12 [==============================] - 0s 12ms/step - loss: 0.0097 - accuracy:
    Epoch 480/500
```

```
12/12 [==============================] - 0s 12ms/step - loss: 0.0100 - accuracy:
Epoch 481/500
12/12 [==============================] - 0s 13ms/step - loss: 0.0096 - accuracy:
Epoch 482/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0098 - accuracy:
Epoch 483/500
12/12 [==============================] - 0s 13ms/step - loss: 0.0099 - accuracy:
Epoch 484/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0103 - accuracy:
Epoch 485/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0098 - accuracy:
Epoch 486/500
12/12 [==============================] - 0s 13ms/step - loss: 0.0101 - accuracy:
Epoch 487/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0101 - accuracy:
Epoch 488/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0098 - accuracy:
Epoch 489/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0099 - accuracy:
Epoch 490/500
12/12 [==============================] - 0s 15ms/step - loss: 0.0098 - accuracy:
Epoch 491/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0106 - accuracy:
Epoch 492/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0098 - accuracy:
Epoch 493/500
12/12 [==============================] - 0s 13ms/step - loss: 0.0096 - accuracy:

Epoch 494/500
12/12 [==============================] - 0s 13ms/step - loss: 0.0098 - accuracy:
Epoch 495/500
12/12 [==============================] - 0s 13ms/step - loss: 0.0100 - accuracy:
Epoch 496/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0098 - accuracy:
Epoch 497/500
12/12 [==============================] - 0s 13ms/step - loss: 0.0099 - accuracy:
Epoch 498/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0098 - accuracy:
Epoch 499/500
12/12 [==============================] - 0s 12ms/step - loss: 0.0093 - accuracy:
Epoch 500/500
12/12 [==============================] - 0s 13ms/step - loss: 0.0097 - accuracy:
```

Question 2:

```
# shuffle the data and split the data into training set and validation set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=

# # # Using Dropout to do regularization
# tf.random.set_seed(0)
# layer = tf.keras.layers.Dropout(.2, input_shape=(2,))
# reg_X_train = layer(X_train, training=True)
# print(reg_X_train)

# tf.random.set_seed(0)
# layer = tf.keras.layers.Dropout(.2, input_shape=(2,))
# reg_X_test = layer(X_test, training=True)
# print(reg_X_test)



# define model
# model2 = Sequential()
# model2.add(LSTM(75, input_shape=(X.shape[1], X.shape[2])))
# model2.add(Dense(vocab_size, activation='softmax'))

# print(model2.summary())


# compile model
# model2.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy

# fit model
print("Fit model on training data")
history2=model1.fit(X_train, y_train, epochs=500, validation_data=(X_test, y_test))
```

```
    10/10 [==============================] - 0s 21ms/step - loss: 0.0052 - accuracy:
    Epoch 472/500

    10/10 [==============================] - 0s 20ms/step - loss: 0.0051 - accuracy:
    Epoch 473/500
    10/10 [==============================] - 0s 19ms/step - loss: 0.0052 - accuracy:
    Epoch 474/500
    10/10 [==============================] - 0s 21ms/step - loss: 0.0050 - accuracy:
    Epoch 475/500
    10/10 [==============================] - 0s 20ms/step - loss: 0.0054 - accuracy:
    Epoch 476/500
    10/10 [==============================] - 0s 20ms/step - loss: 0.0048 - accuracy:
    Epoch 477/500
    10/10 [==============================] - 0s 21ms/step - loss: 0.0054 - accuracy:
    Epoch 478/500
    10/10 [==============================] - 0s 19ms/step - loss: 0.0050 - accuracy:
    Epoch 479/500
    10/10 [==============================] - 0s 18ms/step - loss: 0.0053 - accuracy:
    Epoch 480/500
    10/10 [==============================] - 0s 20ms/step - loss: 0.0050 - accuracy:
    Epoch 481/500
    10/10 [==============================] - 0s 20ms/step - loss: 0.0052 - accuracy:
    Epoch 482/500
    10/10 [==============================] - 0s 20ms/step - loss: 0.0060 - accuracy:
    Epoch 483/500
    10/10 [==============================] - 0s 21ms/step - loss: 0.0049 - accuracy:
    Epoch 484/500
    10/10 [==============================] - 0s 19ms/step - loss: 0.0048 - accuracy:
    Epoch 485/500
    10/10 [==============================] - 0s 19ms/step - loss: 0.0051 - accuracy:
    Epoch 486/500
    10/10 [==============================] - 0s 18ms/step - loss: 0.0051 - accuracy:
    Epoch 487/500
    10/10 [==============================] - 0s 20ms/step - loss: 0.0049 - accuracy:
```

```
10/10 [==============================] - 0s 20ms/step - loss: 0.0049 - accuracy:
Epoch 488/500
10/10 [==============================] - 0s 18ms/step - loss: 0.0049 - accuracy:
Epoch 489/500
10/10 [==============================] - 0s 20ms/step - loss: 0.0051 - accuracy:
Epoch 490/500
10/10 [==============================] - 0s 22ms/step - loss: 0.0051 - accuracy:
Epoch 491/500
10/10 [==============================] - 0s 19ms/step - loss: 0.0051 - accuracy:
Epoch 492/500
10/10 [==============================] - 0s 19ms/step - loss: 0.0051 - accuracy:
Epoch 493/500
10/10 [==============================] - 0s 20ms/step - loss: 0.0049 - accuracy:
Epoch 494/500
10/10 [==============================] - 0s 20ms/step - loss: 0.0049 - accuracy:
Epoch 495/500
10/10 [==============================] - 0s 20ms/step - loss: 0.0049 - accuracy:
Epoch 496/500
10/10 [==============================] - 0s 19ms/step - loss: 0.0049 - accuracy:
Epoch 497/500
10/10 [==============================] - 0s 21ms/step - loss: 0.0048 - accuracy:
Epoch 498/500
10/10 [==============================] - 0s 21ms/step - loss: 0.0051 - accuracy:
Epoch 499/500
10/10 [==============================] - 0s 20ms/step - loss: 0.0049 - accuracy:
Epoch 500/500
10/10 [==============================] - 0s 19ms/step - loss: 0.0053 - accuracy:
```

Question 3:

Summary:

Methods & Result:

- I increase the epoch sizes to 500. The accuracy improves from 0.9896 to 0.9948 and the loss decreases from 0.2193 to 0.0092.
    - When I set the epoch sizes to 50, the training accuracy is only 0.6120, I think it is because it loses the generalization capacity of the neural network.
    - Then I set the epoch sizes to 200, the loss decreases but the training accuracy is slightly lower than the original one. Finally we set the epoch sizes to 500 which is an optimal number of epochs for this model.
    - The purpose of tuning epoch sizes is to mitigate overfitting and to increase the generalization capacity of the neural network, so we need to care about the validation loss and accuracy in q2. But the result is great, so we keep the epoch sizes as 500.
- (When I use this model to fit the q2 training set, the validation accuracy is very low, so I go back to q1 and add a mask layer) Then, I added a mask layer and set the mask_value to 1.0, the loss decreases to 0.0090 and the accuracy increases to 0.9948 in all training samples.

  - After we split the training and validation set in q2, this model's validation accuracy is 0.9870 which is also pretty good. (loss: 0.0053 - accuracy: 0.9935 - val_loss: 0.1168 - val_accuracy: 0.9870)
  - Why I want to add a mask layer? Because not all samples have a uniform length, the model must be informed that some part of the data is actually padding and should be ignored. So we apply the masking mechanism here.

Other trials but not working:

- Using Dropout to do regularization, I tried to regularized the training set then apply it to the model, the training model is good, but the validation accuracy is pretty poor. So we don't use it in our final model.
- I also changed the numbers of memory cells from 75 to 120, but the training and validation accuracy are not good as the previous model.

Question 4:

```
#Love Story Lyrics(Taylor's Version)
lyrics='We were both young when I first saw you,\
I close my eyes and the flashback starts,\
I am standin there,\
On a balcony in summer air,\
See the lights see the party the ball gowns,\
See you make your way through the crowd,\
And say Hello,\
Little did I know,\
That you were Romeo you were throwin pebbles,\
And my daddy said Stay away from Juliet,\
And I was cryin on the staircase,\
Beggin you Please dont go and I said,\
Romeo take me somewhere we can be alone,\
I will be waiting all there is left to do is run,\
You will be the prince and I will be the princess,\
It is a love story baby just say Yes,\
So I sneak out to the garden to see you,\
We keep quiet cause we are dead if they knew,\
So close your eyes,\
Escape this town for a little while oh oh,\
Cause you were Romeo I was a scarlet letter,\
And my daddy said Stay away from Juliet,\
But you were everything to me,\
I was beggin you Please dont go and I said,\
Romeo take me somewhere we can be alone,\
I will be waiting all there is left to do is run,\
You will be the prince and I will be the princess,\
It is a love story baby just say Yes,\
Romeo save me they are tryna tell me how to feel,\
This love is difficult but it is real,\
Dont be afraid we will make it out of this mess,\
It is a love story baby just say Yes,\
Oh oh,\
I got tired of waiting,\
Wonderin if you were ever comin around,\
My faith in you was fading,\
When I met you on the outskirts of town and I said,\
Romeo save me I have been feeling so alone,\
I keep waiting for you but you never come,\
Is this in my head? I dont know what to think,\
He knelt to the ground and pulled out a ring,\
And said Marry me Juliet,\
You will never have to be alone,\
I love you and that is all I really know,\
I talked to your dad go pick out a white dress,\
It is a love story baby just say Yes,\
Oh oh oh,\
Oh oh oh oh,\
Cause we were both young when I first saw you'

with open('lyrics.txt','w') as f:
  f.write(lyrics)



#load text
raw_lyrics = load_doc('lyrics.txt')
print(raw_lyrics)

# clean
tokens = raw_lyrics.split()
raw_lyrics = ' '.join(tokens)

# organize into sequences of characters
length = 10
lyrics_sequences = list()
for i in range(length, len(raw_lyrics)):
    # select sequence of tokens
    seq = raw_lyrics[i-length:i+1]
    # store
    lyrics_sequences.append(seq)
print('Total Sequences: %d' % len(lyrics_sequences))
```

```
We were both young when I first saw you,I close my eyes and the flashback starts
Total Sequences: 1785
```

```
# save sequences to file
out_filename2 = 'char_sequences_lyrics.txt'
save_doc(lyrics_sequences, out_filename2)
```

```
# load
in_filename2 = 'char_sequences.txt'
raw_lyrics = load_doc(in_filename2)
lines2 = raw_lyrics.split('\n')


# integer encode sequences of characters
chars = sorted(list(set(raw_lyrics)))
mapping = dict((c, i) for i, c in enumerate(chars))
sequences = list()
for line in lines2:
    # integer encode line
    encoded_seq = [mapping[char] for char in line]
    # store
    sequences.append(encoded_seq)

# vocabulary size
vocab_size = len(mapping)
print('Vocabulary Size: %d' % vocab_size)

# separate into input and output
sequences = array(sequences)
X2, y2 = sequences[:,:-1], sequences[:,-1]
sequences = [to_categorical(x, num_classes=vocab_size) for x in X2]
X2 = array(sequences)
y2 = to_categorical(y2, num_classes=vocab_size)
```

```
     Vocabulary Size: 38
```

```
# define model
model3 = Sequential()
model3.add(Masking(mask_value=1.0))
model3.add(LSTM(300, input_shape=(X.shape[1], X.shape[2])))
model3.add(Dense(vocab_size, activation='softmax'))
# print(model3.summary())
# compile model
model3.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy']
# fit model
history3=model3.fit(X2, y2, epochs=500)
```

```
     12/12 [==============================] - 1s 49ms/step - loss: 0.0084 - accuracy:
     Epoch 472/500
     12/12 [==============================] - 1s 51ms/step - loss: 0.0083 - accuracy:
     Epoch 473/500
     12/12 [==============================] - 1s 50ms/step - loss: 0.0091 - accuracy:
     Epoch 474/500
     12/12 [==============================] - 1s 50ms/step - loss: 0.0094 - accuracy:
     Epoch 475/500
     12/12 [==============================] - 1s 50ms/step - loss: 0.0092 - accuracy:
     Epoch 476/500
     12/12 [==============================] - 1s 53ms/step - loss: 0.0092 - accuracy:
     Epoch 477/500
```

```
12/12 [==============================] - 1s 54ms/step - loss: 0.0093 - accuracy:
Epoch 478/500
12/12 [==============================] - 1s 49ms/step - loss: 0.0086 - accuracy:
Epoch 479/500
12/12 [==============================] - 1s 50ms/step - loss: 0.0083 - accuracy:
Epoch 480/500
12/12 [==============================] - 1s 50ms/step - loss: 0.0086 - accuracy:
Epoch 481/500
12/12 [==============================] - 1s 53ms/step - loss: 0.0079 - accuracy:
Epoch 482/500
12/12 [==============================] - 1s 47ms/step - loss: 0.0083 - accuracy:
Epoch 483/500
12/12 [==============================] - 1s 48ms/step - loss: 0.0087 - accuracy:
Epoch 484/500
12/12 [==============================] - 1s 52ms/step - loss: 0.0095 - accuracy:
Epoch 485/500
12/12 [==============================] - 1s 51ms/step - loss: 0.0084 - accuracy:
Epoch 486/500
12/12 [==============================] - 1s 50ms/step - loss: 0.0089 - accuracy:
Epoch 487/500
12/12 [==============================] - 1s 51ms/step - loss: 0.0089 - accuracy:
Epoch 488/500
12/12 [==============================] - 1s 54ms/step - loss: 0.0085 - accuracy:
Epoch 489/500
12/12 [==============================] - 1s 52ms/step - loss: 0.0091 - accuracy:
Epoch 490/500
12/12 [==============================] - 1s 55ms/step - loss: 0.0085 - accuracy:

Epoch 491/500
12/12 [==============================] - 1s 53ms/step - loss: 0.0099 - accuracy:
Epoch 492/500
12/12 [==============================] - 1s 53ms/step - loss: 0.0087 - accuracy:
Epoch 493/500
12/12 [==============================] - 1s 55ms/step - loss: 0.0083 - accuracy:
Epoch 494/500
12/12 [==============================] - 1s 53ms/step - loss: 0.0086 - accuracy:
Epoch 495/500
12/12 [==============================] - 1s 51ms/step - loss: 0.0089 - accuracy:
Epoch 496/500
12/12 [==============================] - 1s 51ms/step - loss: 0.0088 - accuracy:
Epoch 497/500
12/12 [==============================] - 1s 52ms/step - loss: 0.0092 - accuracy:
Epoch 498/500
12/12 [==============================] - 1s 49ms/step - loss: 0.0086 - accuracy:
Epoch 499/500
12/12 [==============================] - 1s 49ms/step - loss: 0.0078 - accuracy:
Epoch 500/500
12/12 [==============================] - 1s 52ms/step - loss: 0.0086 - accuracy:
```

```
# define model
model4 = Sequential()
model4.add(Masking(mask_value=1.0))
model4.add(LSTM(300, input_shape=(X.shape[1], X.shape[2])))
model4.add(Dense(vocab_size, activation='softmax'))
# print(model3.summary())
# compile model
```

```
model4.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'
# fit model
history4=model4.fit(X2, y2, epochs=100)
```

```
Epoch 1/100
12/12 [==============================] - 5s 47ms/step - loss: 3.5436 - accuracy:
Epoch 2/100
12/12 [==============================] - 1s 48ms/step - loss: 3.1358 - accuracy:
Epoch 3/100
12/12 [==============================] - 1s 48ms/step - loss: 3.0888 - accuracy:
Epoch 4/100
12/12 [==============================] - 1s 47ms/step - loss: 3.0358 - accuracy:
Epoch 5/100
12/12 [==============================] - 1s 46ms/step - loss: 3.0083 - accuracy:
Epoch 6/100
12/12 [==============================] - 1s 47ms/step - loss: 2.9940 - accuracy:
Epoch 7/100
12/12 [==============================] - 1s 45ms/step - loss: 2.9506 - accuracy:
Epoch 8/100
12/12 [==============================] - 1s 46ms/step - loss: 2.8866 - accuracy:
Epoch 9/100
12/12 [==============================] - 1s 46ms/step - loss: 2.8528 - accuracy:
Epoch 10/100
12/12 [==============================] - 1s 45ms/step - loss: 2.7987 - accuracy:
Epoch 11/100
12/12 [==============================] - 1s 44ms/step - loss: 2.7092 - accuracy:
Epoch 12/100
12/12 [==============================] - 1s 46ms/step - loss: 2.6580 - accuracy:
Epoch 13/100
12/12 [==============================] - 1s 45ms/step - loss: 2.5729 - accuracy:
Epoch 14/100
12/12 [==============================] - 1s 47ms/step - loss: 2.4974 - accuracy:
Epoch 15/100
12/12 [==============================] - 1s 45ms/step - loss: 2.4549 - accuracy:
Epoch 16/100
12/12 [==============================] - 1s 49ms/step - loss: 2.3532 - accuracy:
Epoch 17/100
12/12 [==============================] - 1s 45ms/step - loss: 2.2581 - accuracy:
Epoch 18/100
12/12 [==============================] - 1s 48ms/step - loss: 2.1807 - accuracy:
Epoch 19/100
12/12 [==============================] - 1s 47ms/step - loss: 2.0423 - accuracy:
Epoch 20/100
12/12 [==============================] - 1s 45ms/step - loss: 1.9275 - accuracy:
Epoch 21/100
12/12 [==============================] - 1s 47ms/step - loss: 1.7541 - accuracy:
Epoch 22/100
12/12 [==============================] - 1s 46ms/step - loss: 1.6547 - accuracy:
Epoch 23/100
12/12 [==============================] - 1s 46ms/step - loss: 1.5312 - accuracy:
Epoch 24/100
12/12 [==============================] - 1s 46ms/step - loss: 1.4493 - accuracy:
Epoch 25/100
12/12 [==============================] - 1s 44ms/step - loss: 1.3201 - accuracy:
Epoch 26/100
12/12 [==============================] - 1s 45ms/step - loss: 1.1762 - accuracy:
Epoch 27/100
```

```
12/12 [==============================] - 1s 45ms/step - loss: 1.0817 - accuracy:
Epoch 28/100
12/12 [==============================] - 1s 46ms/step - loss: 0.9919 - accuracy:
Epoch 29/100
12/12 [==============================] - 1s 44ms/step - loss: 0.8576 - accuracy:
Epoch 30/100
```

```
from sklearn.model_selection import train_test_split
X_train2, X_test2, y_train2, y_test2 = train_test_split(X2, y2, test_size=0.2, random_
```

```
print("Fit model on training data")
history5=model4.fit(X_train2, y_train2, epochs=100, validation_data=(X_test2, y_test2)
```

```
Fit model on training data
Epoch 1/100
10/10 [==============================] - 7s 244ms/step - loss: 0.0832 - accuracy
Epoch 2/100
10/10 [==============================] - 1s 61ms/step - loss: 0.1383 - accuracy:
Epoch 3/100
10/10 [==============================] - 1s 61ms/step - loss: 0.2202 - accuracy:
Epoch 4/100
10/10 [==============================] - 1s 61ms/step - loss: 0.2702 - accuracy:
Epoch 5/100
10/10 [==============================] - 1s 56ms/step - loss: 0.2519 - accuracy:
Epoch 6/100
10/10 [==============================] - 1s 58ms/step - loss: 0.1566 - accuracy:
Epoch 7/100
10/10 [==============================] - 1s 61ms/step - loss: 0.1140 - accuracy:
Epoch 8/100
10/10 [==============================] - 1s 57ms/step - loss: 0.0590 - accuracy:
Epoch 9/100
10/10 [==============================] - 1s 56ms/step - loss: 0.0630 - accuracy:
Epoch 10/100
10/10 [==============================] - 1s 60ms/step - loss: 0.0452 - accuracy:
Epoch 11/100
10/10 [==============================] - 1s 57ms/step - loss: 0.0499 - accuracy:
Epoch 12/100
10/10 [==============================] - 1s 58ms/step - loss: 0.0496 - accuracy:
Epoch 13/100
10/10 [==============================] - 1s 63ms/step - loss: 0.0431 - accuracy:
Epoch 14/100
10/10 [==============================] - 1s 63ms/step - loss: 0.0269 - accuracy:
Epoch 15/100
10/10 [==============================] - 1s 62ms/step - loss: 0.0229 - accuracy:
Epoch 16/100
10/10 [==============================] - 1s 56ms/step - loss: 0.0165 - accuracy:
Epoch 17/100
10/10 [==============================] - 1s 56ms/step - loss: 0.0162 - accuracy:
Epoch 18/100
10/10 [==============================] - 1s 57ms/step - loss: 0.0146 - accuracy:
Epoch 19/100
10/10 [==============================] - 1s 60ms/step - loss: 0.0141 - accuracy:
Epoch 20/100
10/10 [==============================] - 1s 62ms/step - loss: 0.0124 - accuracy:
Epoch 21/100
```

```
10/10 [==============================] - 1s 61ms/step - loss: 0.0120 - accuracy:
Epoch 22/100
10/10 [==============================] - 1s 61ms/step - loss: 0.0117 - accuracy:
Epoch 23/100
10/10 [==============================] - 1s 62ms/step - loss: 0.0117 - accuracy:
Epoch 24/100
10/10 [==============================] - 1s 64ms/step - loss: 0.0117 - accuracy:
Epoch 25/100
10/10 [==============================] - 1s 59ms/step - loss: 0.0109 - accuracy:
Epoch 26/100
10/10 [==============================] - 1s 60ms/step - loss: 0.0113 - accuracy:
Epoch 27/100
10/10 [==============================] - 1s 59ms/step - loss: 0.0114 - accuracy:
Epoch 28/100
10/10 [==============================] - 1s 60ms/step - loss: 0.0111 - accuracy:
Epoch 29/100
10/10 [==============================] - 1s 62ms/step - loss: 0.0096 - accuracy:
```

- We take a Taylor Swift's song Love Story as out text data. The sequence size (1785) is much larger than the previous one(384). And using the same method as the last model, we get the result: loss -- 0.0056; training accuracy -- 0.9935; val_loss -- 0.4442; val_accuracy -- 0.8701.
- Then I started exploring what is the effect for different numbers of memory cells in an LSTM, I increased it from 75 to 300. Because my total sequences are around 1785 which are larger this time. It is quite interesting that the validation accuracy is 0.9870 in the first 100 epochs, then it decreases to 0.7403 at my last epoch. So I re-train my model with 100 epochs and get the result: loss: 0.0069 - accuracy: 0.9967 - val_loss: 0.3506 - val_accuracy: 0.9091. It has the higher validation accuracy. Why 500 epochs has worse validation accurancy? I think it is because we overfit the model so that the model is well-trained for our training data but can not have a good performance in our validation set.

✓ 1m 26s completed at 1:30 AM