1 Preprocessing of Text

To encode each character into a one-hot vector as input of RNN

In [1]:
```
import os
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import warnings
warnings.filterwarnings("ignore")

#!pip install matplotlib
#!pip install scikit-learn

import tensorflow as tf
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker
import unicodedata
import re
import numpy as np
import os
import time
from pylab import *
from matplotlib.font_manager import FontProperties
import pandas as pd
```

In [9]:
```
tf.__version__
```

Out[9]: '2.5.0'

In [2]:
```python
import io
#"C:/Users/cluster/Desktop/lee/DL_HW3/shakespeare_train.txt"

data_URL = "C:/Users/stat_835/Desktop/DL/DL_HW3/shakespeare_train.txt"
with io.open( data_URL , 'r' , encoding="utf8" ) as f :
    text=f.read()
print ('Length of text: {} characters'.format(len(text)))

vocab = sorted(set(text)) #set=unique
print ('{} unique characters'.format(len(vocab)))

vocab_to_int={c : i for i , c in enumerate( vocab )}
int_to_vocab = dict(enumerate( vocab ) )
train_data=np.array([vocab_to_int[c] for c in text],dtype=np.int32)
```

```
Length of text: 4351312 characters
67 unique characters
```

In [3]:
```python
int_to_vocab
```

Out[3]:
```
{0: '\n',
 1: ' ',
 2: '!',
 3: '$',
 4: '&',
 5: "'",
 6: ',',
 7: '-',
 8: '.',
 9: '3',
 10: ':',
 11: ';',
 12: '?',
 13: 'A',
 14: 'B',
 15: 'C',
 16: 'D',
 17: 'E',
 18: 'F',
```

We find that the dataset has 67 unique characters.

In [4]:
```python
print ('{} characters mapped to int {}'.format(repr(text[:13]), [vocab_to_int[c] for c in text[:13]]))
```

```
'First Citizen' characters mapped to int [18, 49, 58, 59, 60, 1, 15, 49, 60, 49, 66, 45, 54]
```

In [5]:
```
train_data
```

Out[5]:
```
array([18, 49, 58, ..., 52,  2,  0])
```

In [6]:
```
data_URL = "C:/Users/stat_835/Desktop/DL/DL_HW3/shakespeare_valid.txt"
with io.open(data_URL, 'r', encoding ='utf8') as f:
    text2 = f.read()

valid_data = np.array([vocab_to_int[c] for c in text2], dtype = np.int32)
```

In [7]:
```
print ('{} characters mapped to int {}'.format(repr(text2[:5]), [vocab_to_int[c] for c in text2[:5]]))
```

```
'DUKE ' characters mapped to int [16, 33, 23, 17, 1]
```

In [8]:
```
valid_data
```

Out[8]:
```
array([16, 33, 23, ..., 45,  8,  0])
```

One hot encoding

In [10]:
```
train_one_hot = tf.one_hot(train_data, len(vocab))
valid_one_hot = tf.one_hot(valid_data, len(vocab))
```

In [11]:
```
train_one_hot.shape,valid_one_hot.shape
```

Out[11]:
```
(TensorShape([4351312, 67]), TensorShape([222025, 67]))
```

In [12]:
```
train_one_hot[0].numpy()
```

Out[12]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
      dtype=float32)
```

2 Recurrent Neural Network

In [13]:
```
idx2char = np.array(vocab)
idx2char
```

Out[13]:
```
array(['\n', ' ', '!', '$', '&', "'", ',', '-', '.', '3', ':', ';', '?',
       'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
       'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
       '[', ']', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k',
       'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
       'y', 'z'], dtype='<U1')
```

Create training examples and targets

In [14]:
```
# The maximum length sentence we want for a single input in characters
seq_length = 100

char_dataset = tf.data.Dataset.from_tensor_slices(train_one_hot)

for i in char_dataset.take(5):
    indices = tf.argmax(i, axis=-1).numpy()
    chars = idx2char[indices]
    print(chars)
```

```
F
i
r
s
t
```

In [16]:
```
sequences = char_dataset.batch(seq_length+1, drop_remainder=True) #101

for item in sequences.take(5):
    #print(idx2char[item.numpy()])
    indices = tf.argmax(item , axis=-1).numpy()
    chars = idx2char[indices]
    print(repr(''.join(chars)))
```

```
'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '
'are all resolved rather to die than to famish?\n\nAll:\nResolved. resolved.\n\nFirst Citizen:\nFirst, you k'
"now Caius Marcius is chief enemy to the people.\n\nAll:\nWe know't, we know't.\n\nFirst Citizen:\nLet us ki"
"ll him, and we'll have corn at our own price.\nIs't a verdict?\n\nAll:\nNo more talking on't; let it be d"
'one: away, away!\n\nSecond Citizen:\nOne word, good citizens.\n\nFirst Citizen:\nWe are accounted poor citi'
```

In [114]:
```
def split_input_target(chunk):
    input_text = chunk[:-1]
    target_text = chunk[1:]
    return input_text, target_text

#dataset contains pairs of input and target sequences
dataset = sequences.map(split_input_target)
dataset
```

Out[114]:
```
<MapDataset shapes: ((100, 67), (100, 67)), types: (tf.float32, tf.float32)>
```

In [23]:
```
for input_example, target_example in  dataset.take(1):
    indices = tf.argmax(input_example, axis=-1).numpy()
    indices2 = tf.argmax(target_example, axis=-1).numpy()
    chars = "".join(idx2char[indices])
    chars2 = "".join(idx2char[indices2])

    print ('Input data: ', repr(''.join(chars)))
    print ('Target data:', repr(''.join(chars2)))
```

```
Input data:  'First Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou'
Target data: 'irst Citizen:\nBefore we proceed any further, hear me speak.\n\nAll:\nSpeak, speak.\n\nFirst Citizen:\nYou '
```

In [24]:
```
for i, (input_idx, target_idx) in enumerate(zip(input_example[:5], target_example[:5])):
    print("Step {:4d}".format(i))
    indices = tf.argmax(input_idx, axis=-1).numpy()
    indices2 = tf.argmax(target_idx, axis=-1).numpy()
    chars = "".join(idx2char[indices])
    chars2 = "".join(idx2char[indices2])
    print("  input: {} ({:s})".format(indices, repr(chars)))
    print("  expected output: {} ({:s})".format(indices2, repr(chars2)))
```

```
Step    0
  input: 18 ('F')
  expected output: 49 ('i')
Step    1
  input: 49 ('i')
  expected output: 58 ('r')
Step    2
  input: 58 ('r')
  expected output: 59 ('s')
Step    3
  input: 59 ('s')
  expected output: 60 ('t')
Step    4
  input: 60 ('t')
  expected output: 1 (' ')
```

Create training batches

In [25]:
```
BATCH_SIZE = 64
examples_per_epoch=len(text)//(seq_length)
#BUFFER_SIZE = 10000

dataset_shuffle = dataset.shuffle(examples_per_epoch).batch(BATCH_SIZE, drop_remainder=True)
#drop_remainder=True 如果最後一個批次的數據樣本數不足一個完整的批次(小於batch size)，則將該批次丟棄。

dataset_shuffle #這表示每個批次的元素有兩個部分 模型處理每個序列的大小為 100，並且每個批次有 64 個序列。
```

Out[25]:
```
<BatchDataset shapes: ((64, 100, 67), (64, 100, 67)), types: (tf.float32, tf.float32)>
```

In [26]:
```
examples_per_epoch
```

Out[26]:
```
43513
```

Build The Model

In [27]:
```
def seq_len_split(seq_length, batch_size):
    char_dataset = tf.data.Dataset.from_tensor_slices(train_one_hot)
    char_dataset_valid = tf.data.Dataset.from_tensor_slices(valid_one_hot)

    sequences = tf.data.Dataset.batch(char_dataset, seq_length + 1, drop_remainder = True)
    sequences_valid = tf.data.Dataset.batch(char_dataset_valid, seq_length + 1, drop_remainder = True)

    dataset = sequences.map(split_input_target)
    dataset_valid = sequences_valid.map(split_input_target)

    dataset_shuffle = dataset.shuffle(examples_per_epoch)
    train_data1 = dataset_shuffle.batch(batch_size, drop_remainder = True)
    valid_data1 = dataset_valid.batch(batch_size, drop_remainder=True)

    return train_data1, valid_data1
```

In [28]:

```python
def model_rnn(rnn_unit, batch_size):

    model = tf.keras.models.Sequential()

    model.add(tf.keras.layers.SimpleRNN(
        input_dim=len(vocab),
        batch_size=batch_size,
        units=rnn_unit,
        return_sequences=True,
        stateful=True,
        recurrent_initializer='zeros'
    ))

    model.add(tf.keras.layers.Dense(len(vocab), activation='softmax'))

    return model
```

In [29]:

```python
def model_lstm(rnn_unit, batch_size):

    model = tf.keras.models.Sequential()

    model.add(tf.keras.layers.LSTM(
        input_dim=len(vocab),
        batch_size=batch_size,
        units=rnn_unit,
        return_sequences=True,
        stateful=True,
        recurrent_initializer='zeros'
    ))

    model.add(tf.keras.layers.Dense(len(vocab), activation='softmax'))

    return model
```

1. Construct a standard RNN

(1) network architecture

RNN standard Model (seq_length=100, rnn_unit=512)

In [69]:
```
train_data1, valid_data1= seq_len_split(seq_length=100, batch_size=64)
model_rnn_1 = model_rnn(rnn_unit=512, batch_size=64)
model_rnn_1.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_2 (SimpleRNN)     (64, None, 512)           296960
_________________________________________________________________
dense_2 (Dense)              (64, None, 67)            34371
=================================================================
Total params: 331,331
Trainable params: 331,331
Non-trainable params: 0
_________________________________________________________________
```

In [73]:
```
model_rnn_1.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

In [74]:
```
checkpoint_callback=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_rnn_1/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [75]:
```
model_rnn_1_history = model_rnn_1.fit(
    x = train_data1,
    validation_data = valid_data1,
    epochs = 60,
    callbacks=[checkpoint_callback]
)
```

```
Epoch 51/60
673/673 [==============================] - 79s 112ms/step - loss: 1.3514 - accuracy: 0.5880 - val_loss: 1.5253 - val_accuracy: 0.5498
Epoch 52/60
673/673 [==============================] - 84s 120ms/step - loss: 1.3506 - accuracy: 0.5881 - val_loss: 1.5228 - val_accuracy: 0.5503
Epoch 53/60
673/673 [==============================] - 84s 120ms/step - loss: 1.3498 - accuracy: 0.5882 - val_loss: 1.5260 - val_accuracy: 0.5515
Epoch 54/60
673/673 [==============================] - 84s 119ms/step - loss: 1.3510 - accuracy: 0.5880 - val_loss: 1.5269 - val_accuracy: 0.5487
Epoch 55/60
673/673 [==============================] - 80s 114ms/step - loss: 1.3487 - accuracy: 0.5886 - val_loss: 1.5258 - val_accuracy: 0.5496
Epoch 56/60
673/673 [==============================] - 87s 124ms/step - loss: 1.3487 - accuracy: 0.5885 - val_loss: 1.5231 - val_accuracy: 0.5501
Epoch 57/60
673/673 [==============================] - 82s 116ms/step - loss: 1.3468 - accuracy: 0.5890 - val_loss: 1.5200 - val_accuracy: 0.5500
Epoch 58/60
673/673 [==============================] - 94s 134ms/step - loss: 1.3456 - accuracy: 0.5894 - val_loss: 1.5236 - val_accuracy: 0.5506
Epoch 59/60
673/673 [==============================] - 92s 131ms/step - loss: 1.3463 - accuracy: 0.5891 - val_loss: 1.5256 - val_accuracy: 0.5484
Epoch 60/60
673/673 [==============================] - 93s 133ms/step - loss: 1.3450 - accuracy: 0.5896 - val_loss: 1.5224 - val_accuracy: 0.5522
```

In [76]:
```
model_rnn_1.save("model_rnn_1.h5")
```

In [78]:
```
history_rnn_1 = pd.DataFrame(model_rnn_1_history.history)
with open('history/history_rnn_1.json', 'w') as f:
    history_rnn_1.to_json(f)
```

(2) learning curve

We minimize the bits-per-character (BPC):

$$BPC = -\frac{1}{T}\sum_{t=1}^{T}\sum_{k=1}^{K} t_{t,k} log y_{t,k}(x_t, w)$$

where y denotes the output from RNN and t denotes the corresponding target value, and K is the length of the one-hot vector.

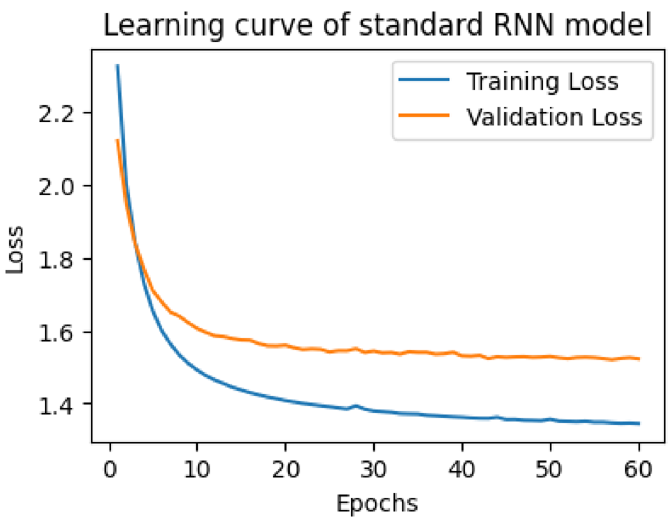
Because we use mini-batch of input data, consider the following objective function:

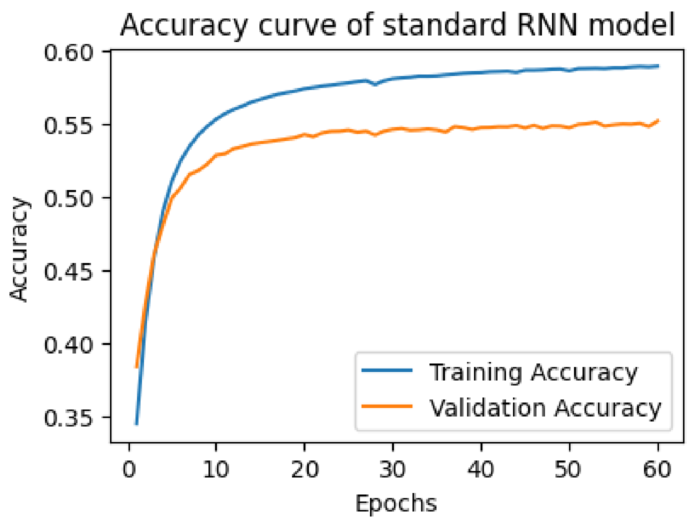
$$E(w) = -\frac{1}{NT}\sum^{N}\sum^{T}\sum^{K} t_{t,k} log y^{n}_{t,k}(x^{n}_{t}, w)$$

In [79]:
```
with open('./history/history_rnn_1.json', 'r') as f:
    history_rnn_1 = pd.read_json(f)
```

In [89]:

```
import matplotlib.pyplot as plt

epochs = range(1, 61)

plt.figure(figsize=(10, 3))
plt.subplots_adjust(wspace=0.3)

# 訓練損失
plt.subplot(1, 2, 1)
plt.plot(epochs,history_rnn_1['loss'], label='Training Loss')
plt.plot(epochs,history_rnn_1['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Learning curve of standard RNN model')
plt.legend()

# 訓練準確度
plt.subplot(1, 2, 2)
plt.plot(epochs,history_rnn_1['accuracy'], label='Training Accuracy')
plt.plot(epochs,history_rnn_1['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Accuracy curve of standard RNN model')
plt.legend()

plt.show()
```





In the left-hand plot, we can see that

- the training loss is lower than the validation loss.
- After epoch 30, two losses are tend to be stable.

In the right-hand plot, we can see that

- the training accuracy is higher than the validation accuracy.
- After epoch 30, the accuracy rates are tend to be stable.

(3) training error rate

$$training\ error\ rate = 1 - training\ accuracy\ rate$$

We take the result of the last epoch (epoch=60).

In [81]:
```
# 最後一個 epoch 的訓練training error rate
print(f"training error rate: {1-history_rnn_1['accuracy'].iloc[-1]}")
```

```
training error rate: 0.41039675470000003
```

(4) validation error rate

$$validation\ error\ rate = 1 - validation\ accuracy\ rate$$

In [82]:
```
# 最後一個 epoch 的validation error rate
print(f"validation error rate: {1-history_rnn_1['val_accuracy'].iloc[-1]}")
```

```
validation error rate: 0.4477849007
```

2. Choose 5 breakpoints during your training process to show how well your network learns through more epochs. Feed some part of your training text into RNN and show the text output.

We choose 5 breakpoints: epoch = $[1, 15, 30, 45, 60]$ during the training process, and show some part of output below.

In [161]:
```
dataset = sequences.map(split_input_target)
dataset            #(seq_length, vocab_size), (seq_length, vocab_size)
```

Out[161]:
```
<MapDataset shapes: ((100, 67), (100, 67)), types: (tf.float32, tf.float32)>
```

In [197]:
```
dataset_list = list(dataset.as_numpy_iterator())
len(dataset_list)
```

Out[197]:
```
43082
```

In [259]:
```
selected_batch=dataset_list[43081] #43081
len(selected_batch)
```

Out[259]:
```
2
```

In [331]:
```
selected_batch[0]
```

Out[331]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

In [260]:
```
selected_batch[0].shape #有100個字 67種字元
```

Out[260]:
```
(100, 67)
```

In [230]:
```
rnn_model_1_pred = model_rnn(512,1)
```

In [262]:

```
print("\n\n--------------------prediction---------------------")
def predict_print_output(model, ckpt_epochs):
    for epoch in ckpt_epochs:

        checkpoint_path = f'model_rnn_1/checkpoints/ckpt_{epoch}'

        # 載入模型權重
        model.load_weights(checkpoint_path)

        # 重置模型狀態
        model.reset_states()

        # 使用模型進行預測  rnn_model_1_pred(tf.expand_dims(selected_batch[0], 0)) #100*67
        predict = model(tf.expand_dims(selected_batch[0], 0)) #給他一個seq去預測 (100*67個機率)
        predict = predict.numpy()
        predict = predict.argmax(2)
        predict_result = predict.squeeze()

        print(f"\n\nOutput data (Epoch {epoch}): \n'", sep="", end="")
        for item in predict_result:
            print(int_to_vocab[item], sep="", end="")


predict_print_output(rnn_model_1_pred, ckpt_epochs=[1, 15, 30, 45, 60])

print("\n\n--------------------true---------------------")
print("Input data:")

for item in selected_batch[0]:
    idx=item.argmax()
    char=int_to_vocab[idx]
    print(char, sep="", end="")

print("\n\nTarget data:")

for item in selected_batch[1]:
    idx=item.argmax()
    char=int_to_vocab[idx]
    print(char, sep="", end="")
```

```
--------------------prediction--------------------


Output data (Epoch 1):
':u  Toetl he tr re te  tn tnl tnr th toue

hrs  toet  tnd thet er   tnd th tord r tnl
Ind tare tor

Output data (Epoch 15):
':uk Ahell be dv rnaned tn t l tna to soue

o nd toen  and phuepet   tnd th sordon hnl
Snd tore thc

Output data (Epoch 30):
':uk Yeall we ax rnaeed in t l tna fh tone

o nd teams and thitpet   and th bordon tnl
And tare thc

Output data (Epoch 45):
':uk Shall be an rnanen bn t l tnanto tome

o nd trams ond dhaepet   and th bondon tnl
Tnd tare thc

Output data (Epoch 60):
':uk Shall be tx rnanen bn t l tda to tome

o nd toams ond thuepets  tnd th tondon tsl
Tnd tare thc

--------------------true--------------------
Input data:
York
Shall be eternized in all age to come.
Sound drums and trumpets, and to London all:
And more su

Target data:
ork
Shall be eternized in all age to come.
Sound drums and trumpets, and to London all:
And more suc
```

- Epoch 1:

Output seems random and doesn't make much sense. The model is likely guessing.

- Epoch 15, 30

Some improvement, with English words appearing, for example, "be" "to" "and". Still not very meaningful.

- Epoch 45:

Output becomes more meaningful. "Shall" matches the target data.

- Epoch 60: More output makes sense, for example, "Shall" matches the target data. Overall, words are more similar to the target data.

In summary, as training progresses, the network is getting better at generating meaningful text through more epochs.

RNN Model 2 (seq_len=70, rnn_unit=512)

In [99]:

```
train_data2, valid_data2= seq_len_split(seq_length=70, batch_size=64)
model_rnn_2 = model_rnn(rnn_unit=256, batch_size=64)
model_rnn_2.summary()
model_rnn_2.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
simple_rnn_9 (SimpleRNN)    (64, None, 256)           82944

_________________________________________________________________
dense_9 (Dense)             (64, None, 67)            17219
=================================================================
Total params: 100,163
Trainable params: 100,163
Non-trainable params: 0

_________________________________________________________________
```

In [100]:

```
checkpoint_callback_rnn_2=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_rnn_2/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [101]:
```
model_rnn_2_history = model_rnn_2.fit(
    x = train_data2,
    validation_data = valid_data2,
    epochs = 60,
    callbacks=[checkpoint_callback_rnn_2]
)
```

```
957/957 [==============================] - 190s 194ms/step - loss: 1.4583 - accuracy: 0.5605 - val_loss: 1.5924 - val_accuracy: 0.5341
Epoch 52/60
957/957 [==============================] - 191s 195ms/step - loss: 1.4571 - accuracy: 0.5605 - val_loss: 1.5881 - val_accuracy: 0.5353
Epoch 53/60
957/957 [==============================] - 196s 199ms/step - loss: 1.4566 - accuracy: 0.5609 - val_loss: 1.5921 - val_accuracy: 0.5352
Epoch 54/60
957/957 [==============================] - 185s 188ms/step - loss: 1.4559 - accuracy: 0.5609 - val_loss: 1.5873 - val_accuracy: 0.5360
Epoch 55/60
957/957 [==============================] - 180s 183ms/step - loss: 1.4550 - accuracy: 0.5612 - val_loss: 1.5885 - val_accuracy: 0.5360
Epoch 56/60
957/957 [==============================] - 176s 180ms/step - loss: 1.4547 - accuracy: 0.5611 - val_loss: 1.5896 - val_accuracy: 0.5356

Epoch 57/60
957/957 [==============================] - 175s 179ms/step - loss: 1.4543 - accuracy: 0.5613 - val_loss: 1.5922 - val_accuracy: 0.5330
Epoch 58/60
957/957 [==============================] - 180s 183ms/step - loss: 1.4534 - accuracy: 0.5615 - val_loss: 1.5904 - val_accuracy: 0.5342
Epoch 59/60
957/957 [==============================] - 176s 178ms/step - loss: 1.4527 - accuracy: 0.5616 - val_loss: 1.5862 - val_accuracy: 0.5368
Epoch 60/60
957/957 [==============================] - 197s 194ms/step - loss: 1.4522 - accuracy: 0.5618 - val_loss: 1.5873 - val_accuracy: 0.5356
```

In [ ]:
```
model_rnn_2.save("model_rnn_2.h5")

history_rnn_2 = pd.DataFrame(model_rnn_2_history.history)
with open('./history/history_rnn2.json', 'w') as f:
    history_rnn_2.to_json(f)
```

RNN Model 3 (seq_len=30, rnn_unit=512)

In [ ]:
```
train_data3, valid_data3= seq_len_split(seq_length=30, batch_size=64)
model_rnn_3 = model_rnn(rnn_unit=512, batch_size=64)
model_rnn_3.summary()
model_rnn_3.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

In [25]:
```python
checkpoint_callback_rnn_3=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_rnn_3/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [105]:
```python
model_rnn_3_history = model_rnn_3.fit(
    x = train_data3,
    validation_data = valid_data3,
    epochs = 60,
    callbacks=[checkpoint_callback_rnn_3]
)
```

```
1
Epoch 55/60
2193/2193 [==============================] - 136s 57ms/step - loss: 1.4922 - accuracy: 0.5504 - val_loss: 1.6333 - val_accuracy: 0.520
5
Epoch 56/60
2193/2193 [==============================] - 138s 58ms/step - loss: 1.4903 - accuracy: 0.5510 - val_loss: 1.6366 - val_accuracy: 0.520
0
Epoch 57/60
2193/2193 [==============================] - 134s 56ms/step - loss: 1.4918 - accuracy: 0.5506 - val_loss: 1.6336 - val_accuracy: 0.522
9
Epoch 58/60
2193/2193 [==============================] - 132s 56ms/step - loss: 1.4910 - accuracy: 0.5508 - val_loss: 1.6334 - val_accuracy: 0.520
7
Epoch 59/60
2193/2193 [==============================] - 134s 57ms/step - loss: 1.4898 - accuracy: 0.5510 - val_loss: 1.6387 - val_accuracy: 0.520
6
Epoch 60/60
2193/2193 [==============================] - 134s 56ms/step - loss: 1.4939 - accuracy: 0.5502 - val_loss: 1.6398 - val_accuracy: 0.517
9
```

In [ ]:
```python
model_rnn_3.save("model_rnn_3.h5")

history_rnn_3 = pd.DataFrame(model_rnn_3_history.history)
with open('./history/history_rnn3.json', 'w') as f:
    history_rnn_3.to_json(f)
```

RNN Model 4 (seq_len=100, rnn_unit=1024)

In [30]:
```
train_data4, valid_data4= seq_len_split(seq_length=100, batch_size=64)
model_rnn_4 = model_rnn(rnn_unit=256, batch_size=64)
model_rnn_4.summary()
model_rnn_4.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
simple_rnn (SimpleRNN)       (64, None, 256)           82944     
_________________________________________________________________
dense (Dense)                (64, None, 67)            17219     
=================================================================
Total params: 100,163
Trainable params: 100,163
Non-trainable params: 0
_________________________________________________________________
```

In [31]:
```
checkpoint_callback_rnn_4=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_rnn_4/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [32]:
```
model_rnn_4_history = model_rnn_4.fit(
    x = train_data4,
    validation_data = valid_data4,
    epochs = 60,
    callbacks=[checkpoint_callback_rnn_4]
)
```

```
673/673 [==============================] - 48s 67ms/step - loss: 1.4410 - accuracy: 0.5655 - val_loss: 1.5734 - val_accuracy: 0.5413
Epoch 52/60
673/673 [==============================] - 49s 68ms/step - loss: 1.4393 - accuracy: 0.5660 - val_loss: 1.5730 - val_accuracy: 0.5395
Epoch 53/60
673/673 [==============================] - 49s 68ms/step - loss: 1.4390 - accuracy: 0.5659 - val_loss: 1.5691 - val_accuracy: 0.5410
Epoch 54/60
673/673 [==============================] - 49s 67ms/step - loss: 1.4384 - accuracy: 0.5660 - val_loss: 1.5721 - val_accuracy: 0.5392
Epoch 55/60
673/673 [==============================] - 49s 67ms/step - loss: 1.4370 - accuracy: 0.5664 - val_loss: 1.5691 - val_accuracy: 0.5396
Epoch 56/60
673/673 [==============================] - 48s 67ms/step - loss: 1.4364 - accuracy: 0.5667 - val_loss: 1.5671 - val_accuracy: 0.5421
Epoch 57/60

673/673 [==============================] - 49s 68ms/step - loss: 1.4357 - accuracy: 0.5667 - val_loss: 1.5683 - val_accuracy: 0.5403
Epoch 58/60
673/673 [==============================] - 48s 67ms/step - loss: 1.4352 - accuracy: 0.5669 - val_loss: 1.5649 - val_accuracy: 0.5416
Epoch 59/60
673/673 [==============================] - 49s 67ms/step - loss: 1.4343 - accuracy: 0.5672 - val_loss: 1.5684 - val_accuracy: 0.5403
Epoch 60/60
673/673 [==============================] - 50s 69ms/step - loss: 1.4335 - accuracy: 0.5674 - val_loss: 1.5710 - val_accuracy: 0.5400
```

In [35]:
```
model_rnn_4.save("model_rnn_4.h5")

history_rnn_4 = pd.DataFrame(model_rnn_4_history.history)
with open('./history/history_rnn4.json', 'w') as f:
    history_rnn_4.to_json(f)
```

RNN Model 5 (seq_len=100, rnn_unit=256)

In [36]:
```
train_data5, valid_data5= seq_len_split(seq_length=100, batch_size=64)
model_rnn_5 = model_rnn(rnn_unit=128, batch_size=64)
model_rnn_5.summary()
model_rnn_5.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
simple_rnn_1 (SimpleRNN)     (64, None, 128)           25088
_________________________________________________________________
dense_1 (Dense)              (64, None, 67)            8643
=================================================================
Total params: 33,731
Trainable params: 33,731
Non-trainable params: 0
_________________________________________________________________
```

In [37]:
```
checkpoint_callback_rnn_5=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_rnn_5/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [38]:
```
model_rnn_5_history = model_rnn_5.fit(
    x = train_data5,
    validation_data = valid_data5,
    epochs = 60,
    callbacks=[checkpoint_callback_rnn_5]
)
```

```
673/673 [==============================] - 47s 66ms/step - loss: 1.5749 - accuracy: 0.5317 - val_loss: 1.6892 - val_accuracy: 0.5106
Epoch 52/60
673/673 [==============================] - 48s 67ms/step - loss: 1.5733 - accuracy: 0.5322 - val_loss: 1.6928 - val_accuracy: 0.5104
Epoch 53/60
673/673 [==============================] - 48s 67ms/step - loss: 1.5728 - accuracy: 0.5323 - val_loss: 1.6925 - val_accuracy: 0.5097
Epoch 54/60
673/673 [==============================] - 49s 67ms/step - loss: 1.5717 - accuracy: 0.5326 - val_loss: 1.6914 - val_accuracy: 0.5111
Epoch 55/60
673/673 [==============================] - 54s 75ms/step - loss: 1.5709 - accuracy: 0.5327 - val_loss: 1.6910 - val_accuracy: 0.5108
Epoch 56/60
673/673 [==============================] - 52s 73ms/step - loss: 1.5704 - accuracy: 0.5329 - val_loss: 1.6884 - val_accuracy: 0.5102
Epoch 57/60

673/673 [==============================] - 49s 68ms/step - loss: 1.5695 - accuracy: 0.5331 - val_loss: 1.6895 - val_accuracy: 0.5124
Epoch 58/60
673/673 [==============================] - 52s 73ms/step - loss: 1.5682 - accuracy: 0.5334 - val_loss: 1.6872 - val_accuracy: 0.5109
Epoch 59/60
673/673 [==============================] - 51s 71ms/step - loss: 1.5677 - accuracy: 0.5334 - val_loss: 1.6943 - val_accuracy: 0.5093
Epoch 60/60
673/673 [==============================] - 49s 68ms/step - loss: 1.5674 - accuracy: 0.5337 - val_loss: 1.6873 - val_accuracy: 0.5125
```

In [39]:
```
model_rnn_5.save("model_rnn_5.h5")

history_rnn_5 = pd.DataFrame(model_rnn_5_history.history)
with open('./history/history_rnn5.json', 'w') as f:
    history_rnn_5.to_json(f)
```

3. Compare the results of choosing different size of hidden states and sequence length by plotting the training loss vs. different parameters.

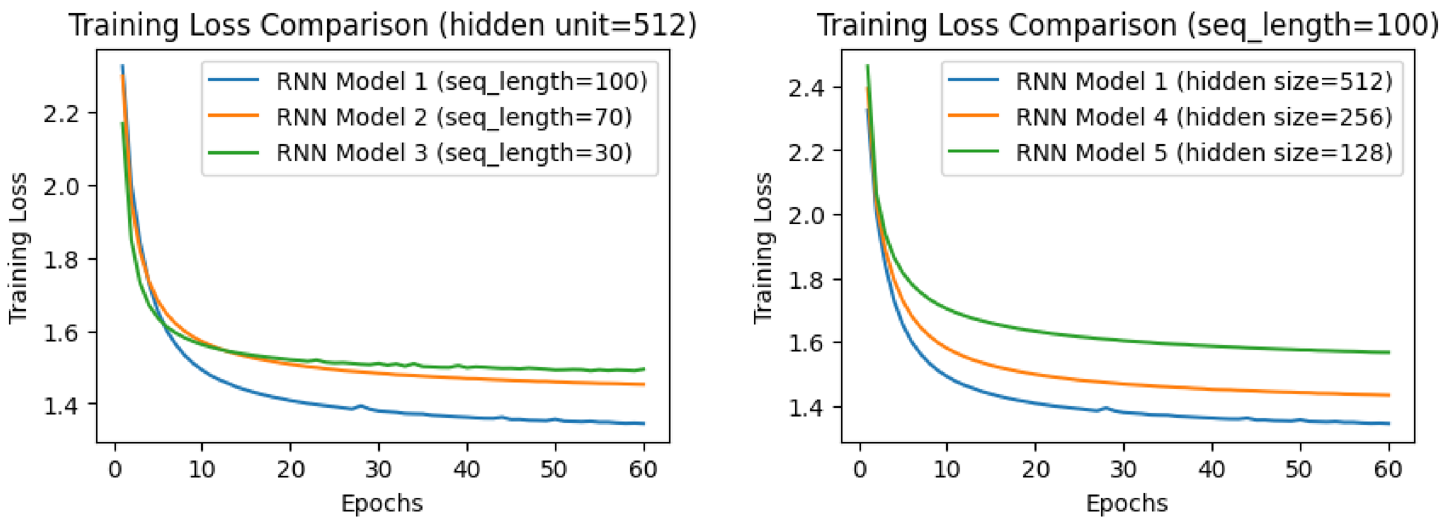
In [63]:

```
import json

with open(f'./history/history_rnn_1.json', 'r') as f:
    history_rnn_1 = pd.DataFrame(json.load(f))
with open(f'./history/history_rnn2.json', 'r') as f:
    history_rnn_2 = pd.DataFrame(json.load(f))
with open(f'./history/history_rnn3.json', 'r') as f:
    history_rnn_3 = pd.DataFrame(json.load(f))
with open(f'./history/history_lstm_5.json', 'r') as f:
    history_lstm_5 = pd.DataFrame(json.load(f))
```

In the left plot below, we compare the results of choosing different sequence length=(30,70,100) under fixed size of hidden states=512.

In the right plot below, we compare the results of choosing different size of hidden states=(128,256,512) under fixed sequence length=100.

In [91]:
```
epochs = range(1, 61)  # Assuming all models were trained for the same number of epochs

plt.figure(figsize=(10, 3))

plt.subplot(1, 2, 1)
plt.subplots_adjust(wspace=0.3)

plt.plot(epochs, history_rnn_1['loss'], label='RNN Model 1 (seq_length=100)')
plt.plot(epochs, history_rnn_2['loss'], label='RNN Model 2 (seq_length=70)')
plt.plot(epochs, history_rnn_3['loss'], label='RNN Model 3 (seq_length=30)')

plt.title('Training Loss Comparison (hidden unit=512)')
plt.xlabel('Epochs')
plt.ylabel('Training Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(epochs, history_rnn_1['loss'], label='RNN Model 1 (hidden size=512)')
plt.plot(epochs, history_rnn_4['loss'], label='RNN Model 4 (hidden size=256)')
plt.plot(epochs, history_rnn_5['loss'], label='RNN Model 5 (hidden size=128)')

plt.title('Training Loss Comparison (seq_length=100)')
plt.xlabel('Epochs')
plt.ylabel('Training Loss')
plt.legend()

plt.show()
```



According to the left plot, we can find that:

- When the sequence length is increased under a fixed hidden unit, the training loss tends to be lower.

According to the right plot, we can find that:

- When the size of hidden states is increased under a fixed seauence length, the training loss tends to be lower.

4. Construct another RNN with LSTM then redo 1. to 3. Also discuss the difference of the results between standard RNN and LSTM.

(1) network architecture

LSTM Model 1 (seq_len=100, rnn_unit=512)

In [40]:
```
train_data1, valid_data1= seq_len_split(seq_length=100, batch_size=64)
model_lstm_1 = model_lstm(rnn_unit=512, batch_size=64)
model_lstm_1.summary()
model_lstm_1.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

```
Model: "sequential_2"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm (LSTM)                  (64, None, 512)           1187840   
_________________________________________________________________
dense_2 (Dense)              (64, None, 67)            34371     
=================================================================
Total params: 1,222,211
Trainable params: 1,222,211
Non-trainable params: 0
_________________________________________________________________
```

In [41]:
```
checkpoint_callback_lstm_1=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_lstm_1/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [42]:
```
model_lstm_1_history = model_lstm_1.fit(
    x = train_data1,
    validation_data = valid_data1,
    epochs = 60,
    callbacks=[checkpoint_callback_lstm_1]
)
```

```
673/673 [==============================] - 42s 58ms/step - loss: 1.1268 - accuracy: 0.6531 - val_loss: 1.5257 - val_accuracy: 0.5597
Epoch 52/60
673/673 [==============================] - 43s 59ms/step - loss: 1.1248 - accuracy: 0.6538 - val_loss: 1.5346 - val_accuracy: 0.5586
Epoch 53/60
673/673 [==============================] - 43s 58ms/step - loss: 1.1233 - accuracy: 0.6541 - val_loss: 1.5307 - val_accuracy: 0.5607
Epoch 54/60
673/673 [==============================] - 43s 59ms/step - loss: 1.1226 - accuracy: 0.6544 - val_loss: 1.5375 - val_accuracy: 0.5588
Epoch 55/60
673/673 [==============================] - 43s 59ms/step - loss: 1.1202 - accuracy: 0.6555 - val_loss: 1.5354 - val_accuracy: 0.5588
Epoch 56/60
673/673 [==============================] - 42s 58ms/step - loss: 1.1191 - accuracy: 0.6556 - val_loss: 1.5376 - val_accuracy: 0.5592
Epoch 57/60

673/673 [==============================] - 43s 58ms/step - loss: 1.1181 - accuracy: 0.6560 - val_loss: 1.5453 - val_accuracy: 0.5588
Epoch 58/60
673/673 [==============================] - 43s 60ms/step - loss: 1.1175 - accuracy: 0.6562 - val_loss: 1.5404 - val_accuracy: 0.5587
Epoch 59/60
673/673 [==============================] - 42s 58ms/step - loss: 1.1152 - accuracy: 0.6567 - val_loss: 1.5420 - val_accuracy: 0.5584
Epoch 60/60
673/673 [==============================] - 42s 58ms/step - loss: 1.1137 - accuracy: 0.6574 - val_loss: 1.5437 - val_accuracy: 0.5583
```

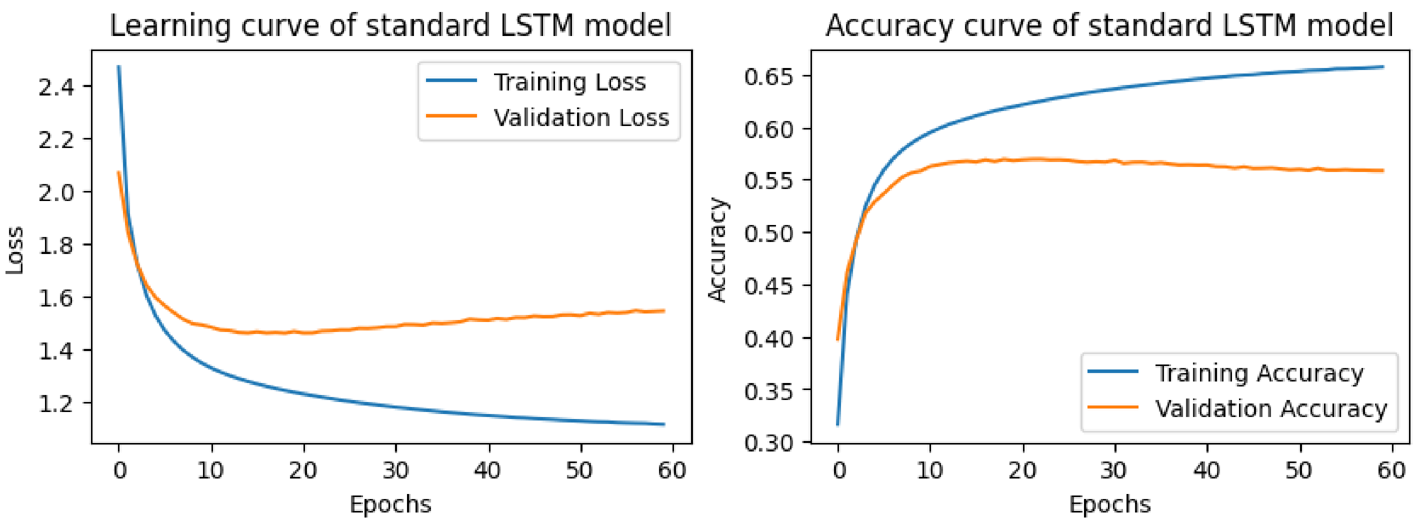
In [43]:
```
model_lstm_1.save("model_lstm_1.h5")

history_lstm_1 = pd.DataFrame(model_lstm_1_history.history)
with open('./history/history_lstm_1.json', 'w') as f:
    history_lstm_1.to_json(f)
```

(2) learning curve

In [77]:
```
with open('./history/history_lstm_1.json', 'r') as f:
    history_lstm_1 = pd.read_json(f)
```

In [79]:

```
import matplotlib.pyplot as plt

# 繪製訓練損失和準確度
plt.figure(figsize=(10, 3))

# 訓練損失
plt.subplot(1, 2, 1)
plt.plot(history_lstm_1['loss'], label='Training Loss')
plt.plot(history_lstm_1['val_loss'], label='Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss') ,
plt.title('Learning curve of standard LSTM model')

plt.legend()

# 訓練準確度
plt.subplot(1, 2, 2)
plt.plot(history_lstm_1['accuracy'], label='Training Accuracy')
plt.plot(history_lstm_1['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.title('Accuracy curve of standard LSTM model')
plt.legend()

plt.show()
```



(3) training error rate

$$training\ error\ rate = 1 - training\ accuracy\ rate$$

In [83]:
```
# 最後一個 epoch 的訓練training error rate
print(f"training error rate: {1-history_lstm_1['accuracy'].iloc[-1]}")
```

```
training error rate: 0.3426302671
```

The training error rate is lower than that of RNN Model 1.

(4) validation error rate

In [84]:
```
# 最後一個 epoch 的validation error rate
print(f"validation error rate: {1-history_lstm_1['val_accuracy'].iloc[-1]}")
```

```
validation error rate: 0.4416728019999999
```

Choose 5 breakpoints during your training process to show how well your network learns through more epochs. Feed some part of your training text into LSTM and show the text output.

We choose 5 breakpoints: epoch = $[1, 15, 30, 45, 60]$ during the training process, and show some part of output below.

In [596]:
```
dataset = sequences.map(split_input_target)
dataset            #(seq_length, vocab_size), (seq_length, vocab_size)
```

Out[596]:
```
<MapDataset shapes: ((100, 67), (100, 67)), types: (tf.float32, tf.float32)>
```

In [597]:
```
dataset_list = list(dataset.as_numpy_iterator())
len(dataset_list)
```

Out[597]:
```
43082
```

In [598]:
```
selected_batch=dataset_list[43081] #43081
len(selected_batch)
```

Out[598]:
```
2
```

In [599]:
```
selected_batch[0]
```

Out[599]:
```
array([[0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       ...,
       [0., 1., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.],
       [0., 0., 0., ..., 0., 0., 0.]], dtype=float32)
```

In [600]:
```
selected_batch[0].shape #有100個字 67種字元
```

Out[600]:
```
(100, 67)
```

In [601]:
```
lstm_model_1_pred = model_lstm(512,1)
```

In [602]:

```
print("\n\n-------------------prediction--------------------")
def predict_print_output(model, ckpt_epochs):
    for epoch in ckpt_epochs:

        checkpoint_path = f'model_lstm_1/checkpoints/ckpt_{epoch}'

        # 載入模型權重
        model.load_weights(checkpoint_path)

        # 重置模型狀態
        model.reset_states()

        # 使用模型進行預測  rnn_model_1_pred(tf.expand_dims(selected_batch[0], 0)) #100*67
        predict = model(tf.expand_dims(selected_batch[0], 0)) #給他一個seq去預測 (100*67個機率)
        predict = predict.numpy()
        predict = predict.argmax(2)
        predict_result = predict.squeeze()

        print(f"\n\nOutput data (Epoch {epoch}): \n'", sep="", end="")
        for item in predict_result:
            print(int_to_vocab[item], sep="", end="")


predict_print_output(lstm_model_1_pred, ckpt_epochs=[1, 15, 30, 45, 60])

print("\n\n-------------------true--------------------")
print("Input data:")

for item in selected_batch[0]:
    idx=item.argmax()
    char=int_to_vocab[idx]
    print(char, sep="", end="")

print("\n\nTarget data:")

for item in selected_batch[1]:
    idx=item.argmax()
    char=int_to_vocab[idx]
    print(char, sep="", end="")
```

```
--------------------prediction---------------------


Output data (Epoch 1):
':uu,MTivllwyatrhr ese  an tnl ane th tome

o ld aoese and theseer   and th tord u tnl
Ind ty e thr

Output data (Epoch 15):
':u  Thall se axernaned tn t l tna ao bome.

h nd wouns and shetpets  and th bondon wnl.
Tnd ware thc

Output data (Epoch 30):
':u  Toall be txernazed in t l oce ao tome.

o nd toums and mruepets, and th sondon wnl.
Tnd tare thc

Output data (Epoch 45):
':ut Thall be txernazed in t l tse oo bome

o nd drums and frumpets  and dh bondon wll.
Tnd tyre thc

Output data (Epoch 60):
':ut Thall be txernated in t l tce wo bome.

ofnd deums and trumpets  and th tondon ull.
Tnd tare thc

--------------------true---------------------
Input data:
York
Shall be eternized in all age to come.
Sound drums and trumpets, and to London all:
And more su

Target data:
ork
Shall be eternized in all age to come.
Sound drums and trumpets, and to London all:
And more suc
```

- Epoch 1:

Output seems random and doesn't make much sense. The model is likely guessing.

- Epoch 15, 30

Some improvement, with English words appearing, for example, "and" "be" "in". Still not very meaningful.

- Epoch 45:

Output becomes more meaningful. "drums" matches the target data.

- Epoch 60: Output becomes more meaningful. "trumpets" matches the target data. Overall, words are more similar to the target data.

In summary, as training progresses, the network is getting better at generating meaningful text through more epochs.

The results generated by LSTM in Epoch 60 are more similar to the target data than those of the RNN model.

LSTM Model 2 (seq_len=70, rnn_unit=512)

In [44]:

```
train_data2, valid_data2= seq_len_split(seq_length=70, batch_size=64)
model_lstm_2 = model_lstm(rnn_unit=512, batch_size=64)
model_lstm_2.summary()
model_lstm_2.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

```
Model: "sequential_3"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (64, None, 512)           1187840
_________________________________________________________________
dense_3 (Dense)              (64, None, 67)            34371
=================================================================
Total params: 1,222,211
Trainable params: 1,222,211
Non-trainable params: 0
_________________________________________________________________
```

In [45]:

```
checkpoint_callback_lstm_2=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_lstm_2/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [46]:

```
model_lstm_2_history = model_lstm_2.fit(
    x = train_data2,
    validation_data = valid_data2,
    epochs = 60,
    callbacks=[checkpoint_callback_lstm_2]
)
```

```
Epoch 1/60
957/957 [==============================] - 55s 53ms/step - loss: 2.3383 - accuracy: 0.3439 - val_loss: 1.9767 - val_accuracy: 0.4215
Epoch 2/60
957/957 [==============================] - 51s 50ms/step - loss: 1.8156 - accuracy: 0.4674 - val_loss: 1.7693 - val_accuracy: 0.4812
Epoch 3/60
957/957 [==============================] - 52s 52ms/step - loss: 1.6345 - accuracy: 0.5165 - val_loss: 1.6584 - val_accuracy: 0.5126
Epoch 4/60
957/957 [==============================] - 49s 49ms/step - loss: 1.5333 - accuracy: 0.5427 - val_loss: 1.5951 - val_accuracy: 0.5296
Epoch 5/60
957/957 [==============================] - 49s 49ms/step - loss: 1.4700 - accuracy: 0.5586 - val_loss: 1.5596 - val_accuracy: 0.5392
Epoch 6/60
957/957 [==============================] - 49s 49ms/step - loss: 1.4273 - accuracy: 0.5694 - val_loss: 1.5373 - val_accuracy: 0.5453
Epoch 7/60
957/957 [==============================] - 50s 49ms/step - loss: 1.3961 - accuracy: 0.5771 - val_loss: 1.5183 - val_accuracy: 0.5506
Epoch 8/60
957/957 [==============================] - 48s 48ms/step - loss: 1.3719 - accuracy: 0.5830 - val_loss: 1.5102 - val_accuracy: 0.5523
Epoch 9/60
957/957 [==============================] - 49s 49ms/step - loss: 1.3530 - accuracy: 0.5877 - val_loss: 1.5022 - val_accuracy: 0.5556
Epoch 10/60
```

In [47]:

```
model_lstm_2.save("model_lstm_2.h5")

history_lstm_2 = pd.DataFrame(model_lstm_2_history.history)
with open('./history/history_lstm_2.json', 'w') as f:
    history_lstm_2.to_json(f)
```

LSTM Model 3 (seq_len=30, rnn_unit=512)

In [48]:
```
train_data3, valid_data3= seq_len_split(seq_length=30, batch_size=64)
model_lstm_3 = model_lstm(rnn_unit=512, batch_size=64)
model_lstm_3.summary()
model_lstm_3.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

```
Model: "sequential_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_2 (LSTM)                (64, None, 512)           1187840

_________________________________________________________________
dense_4 (Dense)              (64, None, 67)            34371
=================================================================
Total params: 1,222,211
Trainable params: 1,222,211
Non-trainable params: 0
_________________________________________________________________
```

In [49]:
```
checkpoint_callback_lstm_3=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_lstm_3/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [50]:
```
model_lstm_3_history = model_lstm_3.fit(
    x = train_data3,
    validation_data = valid_data3,
    epochs = 60,
    callbacks=[checkpoint_callback_lstm_3]
)
```

```
Epoch 1/60
2193/2193 [==============================] - 37s 15ms/step - loss: 2.0915 - accuracy: 0.4009 - val_loss: 1.8217 - val_accuracy: 0.4659
Epoch 2/60
2193/2193 [==============================] - 35s 15ms/step - loss: 1.6671 - accuracy: 0.5073 - val_loss: 1.6948 - val_accuracy: 0.5000
Epoch 3/60
2193/2193 [==============================] - 35s 15ms/step - loss: 1.5601 - accuracy: 0.5344 - val_loss: 1.6398 - val_accuracy: 0.5136
Epoch 4/60
2193/2193 [==============================] - 35s 15ms/step - loss: 1.5094 - accuracy: 0.5471 - val_loss: 1.6167 - val_accuracy: 0.5224
Epoch 5/60
2193/2193 [==============================] - 35s 15ms/step - loss: 1.4793 - accuracy: 0.5546 - val_loss: 1.5958 - val_accuracy: 0.5294
Epoch 6/60
2193/2193 [==============================] - 36s 16ms/step - loss: 1.4579 - accuracy: 0.5599 - val_loss: 1.5909 - val_accuracy: 0.5309
Epoch 7/60
2193/2193 [==============================] - 36s 16ms/step - loss: 1.4424 - accuracy: 0.5638 - val_loss: 1.5828 - val_accuracy: 0.5330
Epoch 8/60
2193/2193 [==============================] - 36s 16ms/step - loss: 1.4294 - accuracy: 0.5669 - val_loss: 1.5808 - val_accuracy: 0.5341
Epoch 9/60
2193/2193 [==============================] - 35s 15ms/step - loss: 1.4194 - accuracy: 0.5699 - val_loss: 1.5744 - val_accuracy: 0.5358
Epoch 10/60
```

In [51]:
```
model_lstm_3.save("model_lstm_3.h5")

history_lstm_3 = pd.DataFrame(model_lstm_3_history.history)
with open('./history/history_lstm_3.json', 'w') as f:
    history_lstm_3.to_json(f)
```

LSTM Model 4 (seq_len=100, rnn_unit=256)

In [52]:
```
train_data4, valid_data4= seq_len_split(seq_length=100, batch_size=64)
model_lstm_4 = model_lstm(rnn_unit=256, batch_size=64)
model_lstm_4.summary()
model_lstm_4.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_3 (LSTM)                (64, None, 256)           331776
_________________________________________________________________
dense_5 (Dense)              (64, None, 67)            17219
=================================================================
Total params: 348,995
Trainable params: 348,995
Non-trainable params: 0
_________________________________________________________________
```

In [53]:
```
checkpoint_callback_lstm_4=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_lstm_4/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [54]:
```
model_lstm_4_history = model_lstm_4.fit(
    x = train_data4,
    validation_data = valid_data4,
    epochs = 60,
    callbacks=[checkpoint_callback_lstm_4]
)
```

```
673/673 [==============================] - 54s 71ms/step - loss: 1.3036 - accuracy: 0.6008 - val_loss: 1.4972 - val_accuracy: 0.5604
Epoch 52/60
673/673 [==============================] - 54s 71ms/step - loss: 1.3023 - accuracy: 0.6011 - val_loss: 1.5000 - val_accuracy: 0.5612
Epoch 53/60
673/673 [==============================] - 52s 68ms/step - loss: 1.3008 - accuracy: 0.6015 - val_loss: 1.4958 - val_accuracy: 0.5609
Epoch 54/60
673/673 [==============================] - 55s 72ms/step - loss: 1.2999 - accuracy: 0.6016 - val_loss: 1.4982 - val_accuracy: 0.5605
Epoch 55/60
673/673 [==============================] - 56s 75ms/step - loss: 1.2989 - accuracy: 0.6020 - val_loss: 1.4947 - val_accuracy: 0.5617
Epoch 56/60
673/673 [==============================] - 58s 76ms/step - loss: 1.2980 - accuracy: 0.6021 - val_loss: 1.4965 - val_accuracy: 0.5606
Epoch 57/60

673/673 [==============================] - 53s 71ms/step - loss: 1.2970 - accuracy: 0.6024 - val_loss: 1.4975 - val_accuracy: 0.5614
Epoch 58/60
673/673 [==============================] - 55s 72ms/step - loss: 1.2961 - accuracy: 0.6025 - val_loss: 1.4998 - val_accuracy: 0.5611
Epoch 59/60
673/673 [==============================] - 61s 81ms/step - loss: 1.2951 - accuracy: 0.6029 - val_loss: 1.4961 - val_accuracy: 0.5614
Epoch 60/60
673/673 [==============================] - 57s 76ms/step - loss: 1.2945 - accuracy: 0.6031 - val_loss: 1.4986 - val_accuracy: 0.5605
```

In [55]:
```
model_lstm_4.save("model_lstm_4.h5")

history_lstm_4 = pd.DataFrame(model_lstm_4_history.history)
with open('./history/history_lstm_4.json', 'w') as f:
    history_lstm_4.to_json(f)
```

LSTM Model 5 (seq_len=100, rnn_unit=128)

In [ ]:
```
train_data5, valid_data5= seq_len_split(seq_length=100, batch_size=64)
model_lstm_5 = model_lstm(rnn_unit=128, batch_size=64)
model_lstm_5.summary()
model_lstm_5.compile(optimizer='adam', metrics=['accuracy'], loss='categorical_crossentropy')
```

In [ ]:
```
checkpoint_callback_lstm_5=tf.keras.callbacks.ModelCheckpoint(
    filepath = os.path.join('model_lstm_5/checkpoints', 'ckpt_{epoch}'),
    save_weights_only=True
)
```

In [109]:
```
model_lstm_5_history = model_lstm_5.fit(
    x = train_data5,
    validation_data = valid_data5,
    epochs = 60,
    callbacks=[checkpoint_callback_lstm_5]
)
```

```
Epoch 51/60
673/673 [==============================] - 61s 86ms/step - loss: 1.4419 - accuracy: 0.5653 - val_loss: 1.5799 - val_accuracy: 0.5382
Epoch 52/60
673/673 [==============================] - 64s 90ms/step - loss: 1.4404 - accuracy: 0.5657 - val_loss: 1.5789 - val_accuracy: 0.5377
Epoch 53/60
673/673 [==============================] - 66s 94ms/step - loss: 1.4391 - accuracy: 0.5661 - val_loss: 1.5788 - val_accuracy: 0.5381
Epoch 54/60
673/673 [==============================] - 60s 85ms/step - loss: 1.4383 - accuracy: 0.5662 - val_loss: 1.5777 - val_accuracy: 0.5397
Epoch 55/60
673/673 [==============================] - 63s 89ms/step - loss: 1.4371 - accuracy: 0.5666 - val_loss: 1.5783 - val_accuracy: 0.5405
Epoch 56/60
673/673 [==============================] - 71s 100ms/step - loss: 1.4360 - accuracy: 0.5668 - val_loss: 1.5782 - val_accuracy: 0.5394
Epoch 57/60
673/673 [==============================] - 71s 101ms/step - loss: 1.4348 - accuracy: 0.5671 - val_loss: 1.5754 - val_accuracy: 0.5396
Epoch 58/60
673/673 [==============================] - 70s 99ms/step - loss: 1.4343 - accuracy: 0.5672 - val_loss: 1.5768 - val_accuracy: 0.5401
Epoch 59/60
673/673 [==============================] - 57s 81ms/step - loss: 1.4328 - accuracy: 0.5676 - val_loss: 1.5763 - val_accuracy: 0.5411
Epoch 60/60
673/673 [==============================] - 63s 89ms/step - loss: 1.4323 - accuracy: 0.5677 - val_loss: 1.5742 - val_accuracy: 0.5405
```

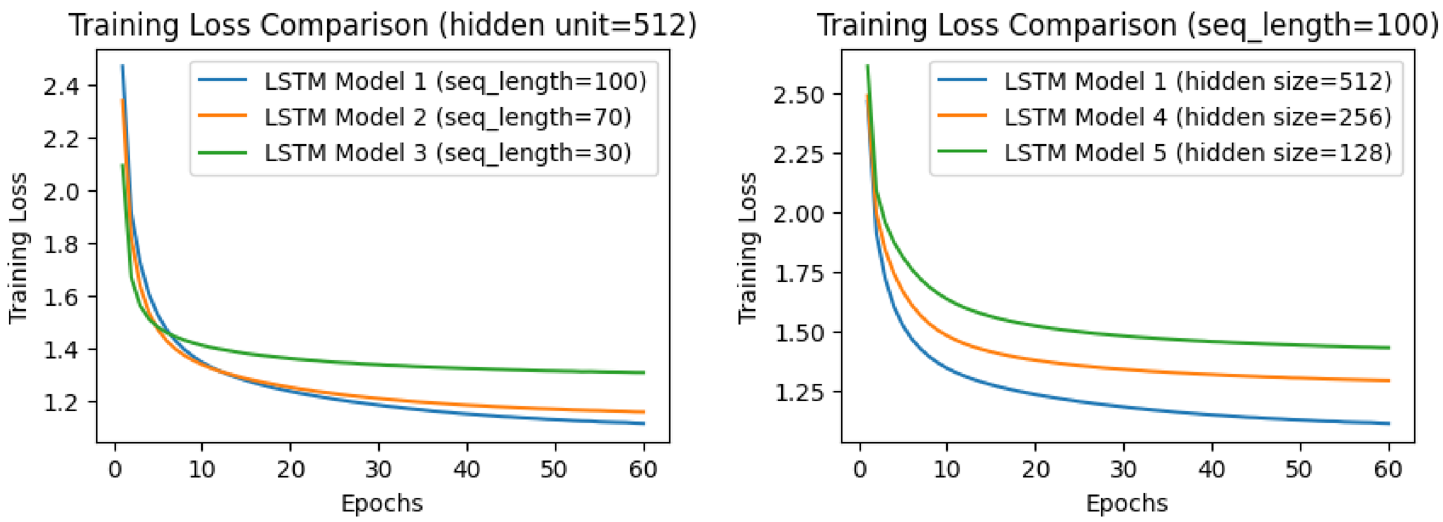
In [ ]:
```
model_lstm_5.save("model_lstm_5.h5")

history_lstm_5 = pd.DataFrame(model_lstm_5_history.history)
with open('./history/history_lstm_5.json', 'w') as f:
    history_lstm_5.to_json(f)
```

Compare the results of choosing different size of hidden states and sequence length by plotting the training loss vs. different parameters.

In the left plot below, we compare the results of choosing different sequence length=(30,70,100) under fixed size of hidden states=512.

In the right plot below, we compare the results of choosing different size of hidden states=(128,256,512) under fixed sequence length=100.

In [90]:
```
epochs = range(1, 61)  # Assuming all models were trained for the same number of epochs

plt.figure(figsize=(10, 3))

plt.subplot(1, 2, 1)
plt.subplots_adjust(wspace=0.3)

plt.plot(epochs, history_lstm_1['loss'], label='LSTM Model 1 (seq_length=100)')
plt.plot(epochs, history_lstm_2['loss'], label='LSTM Model 2 (seq_length=70)')
plt.plot(epochs, history_lstm_3['loss'], label='LSTM Model 3 (seq_length=30)')

plt.title('Training Loss Comparison (hidden unit=512)')
plt.xlabel('Epochs')
plt.ylabel('Training Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(epochs, history_lstm_1['loss'], label='LSTM Model 1 (hidden size=512)')
plt.plot(epochs, history_lstm_4['loss'], label='LSTM Model 4 (hidden size=256)')
plt.plot(epochs, history_lstm_5['loss'], label='LSTM Model 5 (hidden size=128)')

plt.title('Training Loss Comparison (seq_length=100)')
plt.xlabel('Epochs')
plt.ylabel('Training Loss')
plt.legend()

plt.show()
```



According to the left plot, we can find that:

- When the sequence length is increased under a fixed hidden unit, the training loss tends to be lower.

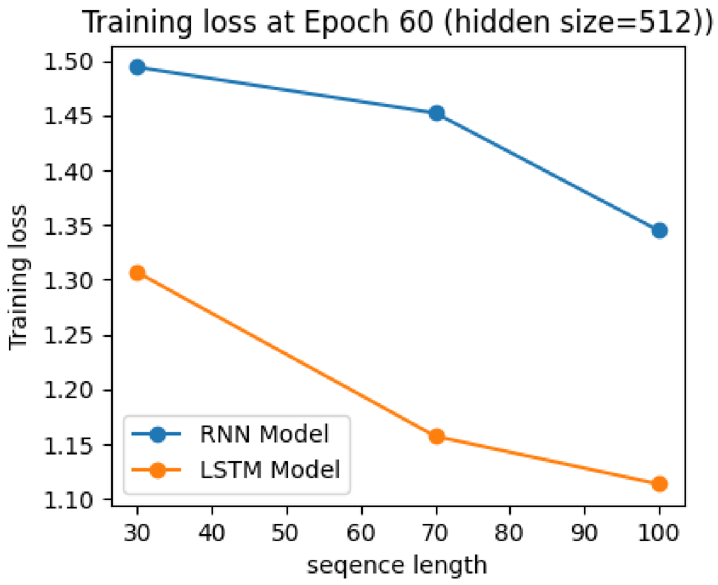
According to the right plot, we can find that:

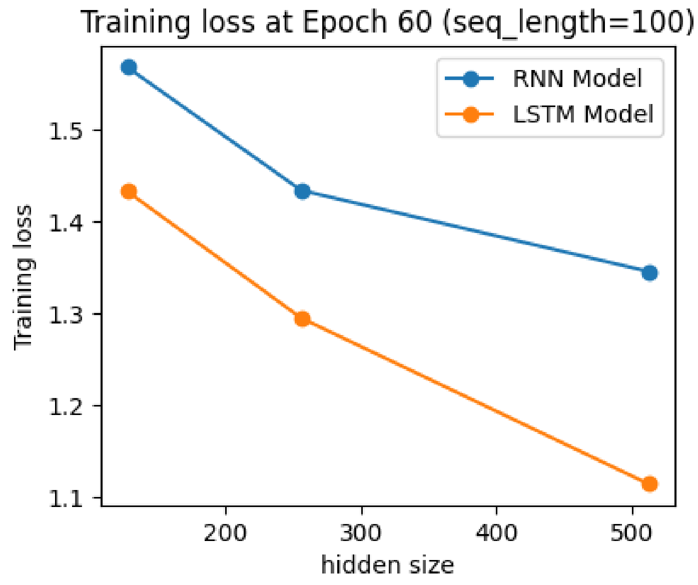
- When the size of hidden states is increased under a fixed seauence length, the training loss tends to be lower.

Discuss the difference of the results between standard RNN and LSTM.

In [111]:

```
plt.figure(figsize=(10, 3.5))
plt.subplots_adjust(wspace=0.3)

plt.subplot(1, 2, 1)
rnn_123_loss=[history_rnn_1['loss'].iloc[-1],history_rnn_2['loss'].iloc[-1],history_rnn_3['loss'].iloc[-1]]
lstm_123_loss=[history_lstm_1['loss'].iloc[-1],history_lstm_2['loss'].iloc[-1],history_lstm_3['loss'].iloc[-1]]

plt.plot([100,70,30],rnn_123_loss, marker='o', label='RNN Model')
plt.plot([100,70,30],lstm_123_loss, marker='o', label='LSTM Model')

plt.title('Training loss at Epoch 60 (hidden size=512))')
plt.xlabel('seqence length')
plt.ylabel('Training loss')
plt.legend()

plt.subplot(1, 2, 2)
rnn_145_loss=[history_rnn_1['loss'].iloc[-1],history_rnn_4['loss'].iloc[-1],history_rnn_5['loss'].iloc[-1]]
lstm_145_loss=[history_lstm_1['loss'].iloc[-1],history_lstm_4['loss'].iloc[-1],history_lstm_5['loss'].iloc[-1]]

plt.plot([512,256,128],rnn_145_loss, marker='o', label='RNN Model')
plt.plot([512,256,128],lstm_145_loss, marker='o', label='LSTM Model')

plt.title('Training loss at Epoch 60 (seq_length=100)')
plt.xlabel('hidden size')
plt.ylabel('Training loss')
plt.legend()

plt.show()
```





According to the left plot, we can find that:

- With a fixed number of hidden size, regardless of the sequence length=(30,70,100), the RNN's loss is consistently higher than that of the LSTM.

According to the right plot, we can find that:

- With a fixed sequence length, regardless of the hidden size=(128,256,512), the RNN's loss is consistently higher than that of the LSTM.

5. Use RNN or LSTM to generate some words by priming the model with a word related to your dataset. Priming the model means giving it some input text to create context and then take the output of the RNN. For example, use ”JULIET” as the prime text of Shakespeare dataset and run the model to generate 10 to 15 lines of output.

- Use LSTM model 1 (seq_length=100, rnn_unit=512) to generate some words.

In [592]:

```
def generate_text_with_primer(model, primer_text, num_char):
    # Convert the primer text to a sequence of indices
    primer_seq = [vocab_to_int[char] for char in primer_text]
    model.reset_states()

    generated_text = primer_text #'JULIET'

    for _ in range(num_char):
        # Use the model to predict the next character
        primer_seq_one_hot = tf.one_hot(primer_seq, len(vocab))
        predict = model(tf.expand_dims(primer_seq_one_hot, 0)).numpy().argmax(2)

        # Take the last predicted character
        predicted_char = int_to_vocab[predict[0, -1]]

        # Add the predicted character to the generated text
        generated_text += predicted_char

        # Update the primer sequence for the next iteration
        primer_seq = [vocab_to_int[predicted_char]]

    return generated_text
```

In [593]:

```
# Primer text
primer_text = "JULIET"

checkpoint_path = f'model_lstm_1/checkpoints/ckpt_40'
lstm_model_1_pred = model_lstm(512,1)
lstm_model_1_pred.load_weights(checkpoint_path)
```

Out[593]:

```
<tensorflow.python.training.tracking.util.CheckpointLoadStatus at 0x2409c833ec8>
```

In [594]:
```
output_prime=generate_text_with_primer(lstm_model_1_pred, primer_text, num_char=583)
print("\n\n--------------------LSTM Prediction with Primer---------------------")
print(output_prime, sep="", end="")
```

```
--------------------LSTM Prediction with Primer---------------------
JULIET:
What say you to the conscience of your children?

MENENIUS:
I will not see thee that the common sinews stays
May to the common place of the dead man.

DESDEMONA:
I do beseech you, sir, the lord protectors
Shall be the stroke of her that will not be any
thing to the commonwealth that would shall be so bold to the commonwealth.

DON PEDRO:
Why, then, I am sure they are not the most performance of the commonwealth,
and as the man is better than the most performance of a man of the
world but I will do with thee a word.

DON PEDRO:
And so will I.

FALSTAFF:
What says she to me?
```

- The model generates text in Shakespearean style, capturing different character voices and maintaining reasonable context.
- Despite some errors in individual words, the majority of the text is meaningful and makes sense.
- Compare to the predicted text generated by RNN Model 1 (in Part 2) , this LSTM Model has better performance.