Tang Poem Generation

Abstract
The midterm assignment requires the use of LSTM to generate acrostic poems(a type of poetry where the first, last or other letters in a line spell out a particular word or phrase) using a dataset of Tang poems. The goal is to train a model that can learn the patterns and structure of Tang poems and generate new poems that follow the same style and conventions. The project involves data preprocessing, model design and training, and evaluation of the generated poems. This experiment aims to develop students' understanding of RNNs and their applications in natural language processing, as well as their ability to design and implement machine learning models for creative tasks. **We will provide an update on this work in our final report, stay tuned.**

Keywords
Natural Language Processing — Language Modeling — Long Short-Term Memory

[1]*Department of Intelligent Systems Engineering, Sun Yat-Sen University, Shenzhen, China*

Contents

1. Experiment Overview

1.1 Introduction

Text generation is a crucial area of research in natural language processing, with the ultimate goal of enabling computers to produce high-quality natural language texts that are indistinguishable from those written by humans. This experiment focuses on acrostic poetry generation and uses LSTM recurrent neural networks to automatically generate poems.

1.2 Experiment Objective

The main content of this experiment is to implement text generation based on *Python 3.6* and *TensorFlow*. Through this experiment, students can learn about the use of AI frameworks for recurrent neural networks, text preprocessing operations, familiarize themselves with the structure of neural networks and the process of text generation, and master the training and prediction of network models, as well as relevant operations in TensorFlow. and table 1 is the checklist of this experiment. Note that we adopt **local computer** in this experiment because currently there's no corresponding version of Tensorflow on ModelArts.

Table 1. Experiment Checklist

Experiment	Acrostic Poetry Generation
Difficulty	Intermediate
Soft Env	*Python3.7.5, Tensorflow 1.13.1*
Dev Env	Modelarts(Local Computer)

2. Tang Poem Generation

2.1 Introduction

The objective of this experiment is to generate acrostic poems. The experiment is based on 34,646 Tang Dynasty poems and uses a 2-layer LSTM recurrent neural network (with 128 hidden nodes per layer) to train the model. After training, the model and network structure can be loaded to test and generate poems. By inputting the first character of each line of poetry, a poem can be generated with that character as the beginning. Fig 1 is an example.

输入： 今天是个好日子

输出：

今日共何初，不是无处回。
天径忘归容，芳桥宵半春。
是异独娆境，前廷今日年。
个尤英靡入，静穆小长擎。
好省同分发，不得苦情频。
日日桃青轻，楚江在云寺。
子德不横云，当朝改化生。

Figure 1. Example of Acrostic Poetry

2.2 Preliminaries

- Text Preprocessing
- Python
- TensorFlow
- LSTM

The term "long short-term memory" comes from the following intuition. Simple recurrent neural networks have long-term memory in the form of weights. The weights change slowly during training, encoding general knowledge about the data. They also have short-term memory in the form of ephemeral activations, which pass from each node to successive nodes. The LSTM model introduces an intermediate type of storage via the memory cell. A memory cell is a composite unit, built from simpler nodes in a specific connectivity pattern, with the novel inclusion of multiplicative nodes.

Each memory cell is equipped with an internal state and a number of multiplicative gates that determine whether (i) a given input should impact the internal state (the input gate), (ii) the internal state should be flushed to 0 (the forget gate), and (iii) the internal state of a given neuron should be allowed to impact the cell' s output (the output gate).

2.2.1 Gated Hidden State

The key distinction between vanilla RNNs and LSTMs is that the latter support gating of the hidden state. This means that we have dedicated mechanisms for when a hidden state should be updated and also when it should be reset. These mechanisms are learned and they address the concerns listed above. For instance, if the first token is of great importance we will learn not to update the hidden state after the first observation. Likewise, we will learn to skip irrelevant temporary observations. Last, we will learn to reset the latent state whenever needed. We discuss this in detail below.

2.2.2 Input, Forget, Output Gate

The data feeding into the LSTM gates are the input at the current time step and the hidden state of the previous time step, as illustrated in Fig 2 . Three fully connected layers with sigmoid activation functions compute the values of the input, forget, and output gates. As a result of the sigmoid activation, all values of the three gates are in the range of (0,1). Additionally, we require an input node, typically computed with a tanh activation function. Intuitively, the input gate determines how much of the input node' s value should be added to the current memory cell internal state. The forget gate determines whether to keep the current value of the memory or flush it. And the output gate determines whether the memory cell should influence the output at the current time step.

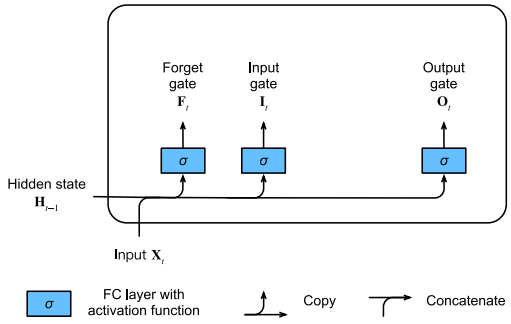


Figure 2. Computing the input gate, the forget gate, and the output gate in an LSTM model.

Mathematically, suppose that there are h hidden units, the batch size is n, and the number of inputs is d. Thus, the input is $\mathbf{X}_t \in \mathbb{R}^{n\times d}$ and the hidden state of the previous time step is $\mathbf{H}_{t-1} \in \mathbb{R}^{n\times h}$. Correspondingly, the gates at time step t are defined as follows: the input gate is $\mathbf{I}_t \in \mathbb{R}^{n\times h}$, the forget gate is $\mathbf{F}_t \in \mathbb{R}^{n\times h}$, and the output gate is $\mathbf{O}_t \in \mathbb{R}^{n\times h}$. They are calculated as follows:

$$\mathbf{I}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xi} + \mathbf{H}_{t-1}\mathbf{W}_{hi} + \mathbf{b}_i),$$
$$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f),$$
$$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o),$$

where $\mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{hi}, \mathbf{W}_{hf}, \mathbf{W}_{ho} \in \mathbb{R}^{h\times h}$ are weight parameters and $\mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1\times h}$ are bias parameters. Note that broadcasting is triggered during the summation. We use sigmoid functions to map the input values to the interval $(0,1)$.

2.2.3 Input Node

Next we design the memory cell. Since we have not specified the action of the various gates yet, we first introduce the *input node* $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n\times h}$. Its computation is similar to that of the three gates described above, but using a tanh function with a value range for $(-1,1)$ as the activation function. This leads to the following equation at time step t:

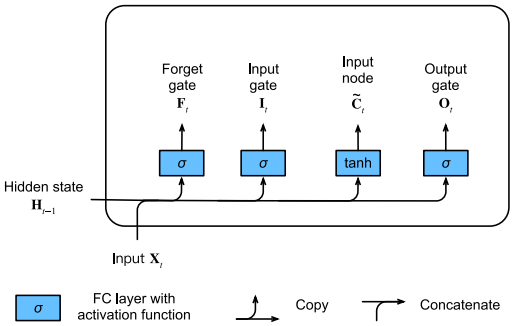


Figure 3. Computing the input node in an LSTM model.

$$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t \mathbf{W}_{xc} + \mathbf{H}_{t-1} \mathbf{W}_{hc} + \mathbf{b}_c),$$

where $\mathbf{W}_{xc} \in \mathbb{R}^{d \times h}$ and $\mathbf{W}_{hc} \in \mathbb{R}^{h \times h}$ are weight parameters and $\mathbf{b}_c \in \mathbb{R}^{1 \times h}$ is a bias parameter.

A quick illustration of the input node is shown in Fig 3.

2.2.4 Memory Cell Internal State

In LSTMs, the input gate $\mathbf{I}_t$ governs how much we take new data into account via $\tilde{\mathbf{C}}_t$ and the forget gate $\mathbf{F}_t$ addresses how much of the old cell internal state $\mathbf{C}_{t-1} \in \mathbb{R}^{n \times h}$ we retain. Using the Hadamard (elementwise) product operator $\odot$ we arrive at the following update equation:

$$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t.$$

If the forget gate is always 1 and the input gate is always 0, the memory cell internal state $\mathbf{C}_{t-1}$ will remain constant forever, passing unchanged to each subsequent time step. However, input gates and forget gates give the model the flexibility to learn when to keep this value unchanged and when to perturb it in response to subsequent inputs. In practice, this design alleviates the vanishing gradient problem, resulting in models that are much easier to train, especially when facing datasets with long sequence lengths.

We thus arrive at the flow diagram in Fig 4.

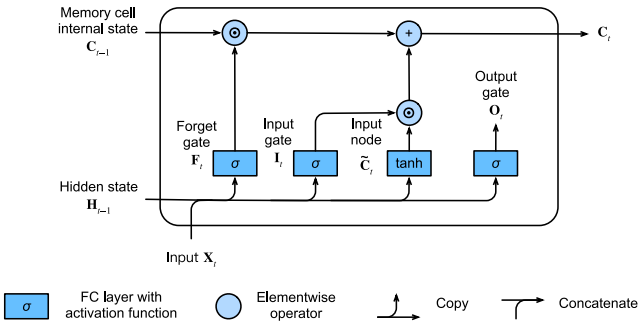


Figure 4. Computing the memory cell internal state in an LSTM model.

2.2.5 Hidden State

Last, we need to define how to compute the output of the memory cell, i.e., the hidden state $\mathbf{H}_t \in \mathbb{R}^{n \times h}$, as seen by other layers. This is where the output gate comes into play. In LSTMs, we first apply tanh to the memory cell internal state and then apply another point-wise multiplication, this time with the output gate. This ensures that the values of $\mathbf{H}_t$ are always in the interval $(-1, 1)$:

$$\mathbf{H}_t = \mathbf{O}_t \odot \tanh(\mathbf{C}_t).$$

Whenever the output gate is close to 1, we allow the memory cell internal state to impact the subsequent layers uninhibited, whereas for output gate values close to 0, we prevent the current memory from impacting other layers of the network at the current time step. Note that a memory cell can accrue information across many time steps without impacting the rest of the network (so long as the output gate takes values close to 0), and then suddenly impact the network at a subsequent time step as soon as the output gate flips from values close to 0 to values close to 1.

Fig 5 has a graphical illustration of the data flow.

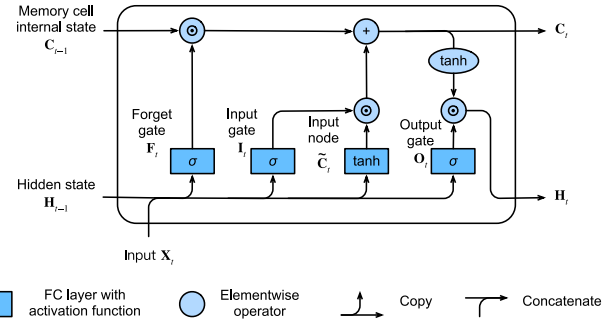


Figure 5. Computing the hidden state in an LSTM model.

3. Implementation

3.1 Create Environment

The modules needed for the experiment are as follows:

```
import os
import datetime
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.utils import shuffle
```

3.2 Data Preparation

The dataset for the experiment is **poetry.txt**, which contains **34646** complete poems. The data is stored as follows: each line is a poem, starting with the title of the poem, followed by the content after ":".

```
首春:
寒随穷律变，春逐鸟声开.初风飘带柳，晚雪间花梅。
碧林青旧竹，绿沼翠新苔。芝田初雁去，绮树巧莺来。
初晴落景:
晚霞聊自怡，初晴弥可喜。日晃百花色，风动千林翠。
池鱼跃不同，园鸟声还异。寄言博通者，知予物外志。
```

Import the poetry data, do preliminary processing, split the title, irrelevant items, etc., and sort by the number of characters in the poem.

```
# 数据集路径
poetry_file ='poetry.txt'
# 导入诗集
poetrys = []
with open(poetry_file, "r" ,
          encoding = 'UTF-8') as f:
    for line in f:
        try:
            #line = line.decode('UTF-8')
            line = line.strip(u'\n')
            title, content = line.strip(u' ').split
                (u':')
            content = content.replace(u' ',u'')
            if u'_' in content or \
               u'(' in content or \
               u'（' in content or \
               u'《' in content or \
               u'[' in content:
                continue
            if len(content) < 5 or \
               len(content) > 79:
                continue
            content = u'[' + content + u']'
            poetrys.append(content)
        except Exception as e:
            pass

# 按诗的字数排序
poetrys = sorted(poetrys,key=lambda line: len(
    line))
print('唐诗总数: ', len(poetrys))
print('\n'.join(poetrys[:10]))
```

The result is as follows:

```
唐诗总数:  34646
[长宜子孙。]
[李下无蹊。]
[罗钳吉网。]
[常杂鲍帖。]
[扬一益二。]
[枫落吴江冷。]
[人生分外愁。]
[木末上明星。]
[犬熟护邻房。]
[兔子上金床。]
```

Figure 6. Dataset list

In natural language processing tasks, natural language needs to be encoded into computable data. Create a vocabulary and obtain a unique ID for each character:

```
vocab = sorted(set([char for poe in poIDrys for
    char in poe]))
vocab.insert(0, "[PAD]")
print ('词汇表长度: {}'.format(len(vocab)))
char2int = {c:i for i, c in enumerate(vocab)}
int2char = np.array(vocab)
print('字符-id 映射: ')
for char,_ in zip(char2int, range(20)):
   print(' {:4s}: {:3d},'.format(repr(char),
       char2int[char]))
print('id-字符映射: ')
print('id 为 10 对应的字符为: {}'.format(int2char
    [10]))
```

The results are as follows:

```
词汇表长度: 6110
字符-id 映射:
 '[PAD]':   0,
 '2' :   1,
 'F' :   2,
 '[' :   3,
 ']' :   4,
 'p' :   5,
 'ē' :   6,
 'ń' :   7,
 '□' :   8,
 '、' :   9,
 '。' :  10,
 '】' :  11,
 '一' :  12,
 '丁' :  13,
 '七' :  14,
 '万' :  15,
 '丈' :  16,
 '三' :  17,
 '上' :  18,
 '下' :  19,
id-字符映射:
id 为 10 对应的字符为: 。
```

Figure 7. Result

So each character has a corresponding number to replace it, and can also be converted to each other. An example is as follows:

```
poetrys_as_int = [[char2int[char]
                   for char in poetry]
                   for poetry in poetrys]
print('{}\n mapped to integers:\n {}'
.format(repr(poetrys[0]), poetrys_as_int[0]))
```

```
'[长宜子孙。]'
 mapped to integers:
 [3, 5459, 1108, 1070, 1076, 10, 4]
```

Figure 8. Result

Take the first 28,000 poems in the dataset as the training set, the rest as the validation set, and build tf.data.Dataset, and split samples and targets:

```
poetrys_shuffle = shuffle(poetrys_as_int,
    random_state=0)
train_inputs = poetrys_shuffle[:28000]
valid_inputs = poetrys_shuffle[28000:]

m_tr = max([len(seq) for seq in train_inputs])
train_padded = tf.keras.preprocessing.sequence.
    pad_sequences(train_inputs, maxlen=m_tr,
    padding='post', value=char2int['[PAD]'])
valid_padded = tf.keras.preprocessing.sequence.
    pad_sequences(valid_inputs, maxlen=m_tr,
    padding='post', value=char2int['[PAD]'])

train_ds = tf.data.Dataset.from_tensor_slices(
    train_padded)
valid_ds = tf.data.Dataset.from_tensor_slices(
    valid_padded)

def split_input_target(seq):
    inputs = seq[:-1]
    targets = seq[1:]
    return inputs, targets

train_ds = train_ds.map(split_input_target).
    shuffle(buffer_size=10000).batch(batch_size
    =64, drop_remainder=True)
valid_ds = valid_ds.map(split_input_target).
    shuffle(buffer_size=10000).batch(batch_size
    =64, drop_remainder=True)
```

It is important to note that, due to the different lengths of poems, uniform length should be achieved before batch training by padding them to a fixed length. Here, padding is unified to the longest poem length: 81.

The splitting part actually removes the last and first characters of the original sample respectively. The purpose of doing this is to make the input character at the corresponding position in the target be the next input character, thus achieving the effect of ”prediction”.

3.3 Model Building and Loading

Define model parameters and create model:

```
embedding_dim = 256
rnn_units = 1024
batch_size = 64
vocab_size = len(vocab)

import tf.keras.layers as layers
```

```
def build_model(vocab_size, embedding_dim,
    rnn_units, batch_size):
    model = tf.keras.Sequential([
        layers.Embedding(vocab_size,
                         embedding_dim,
                         batch_input_shape=\
                         [batch_size, None]),
        layers.Dropout(0.2),
        layers.LSTM(rnn_units,
                    return_sequences=True,
                    stateful=True,
                    recurrent_initializer=\
                    'glorot_uniform'),
        layers.Dropout(0.2),
        layers.LSTM(rnn_units,
                    return_sequences=True,
                    stateful=True,
                    recurrent_initializer=\
                    'glorot_uniform'),
        layers.Dropout(0.2),
        layers.Dense(vocab_size)
    ])
    return model

model = build_model(
        vocab_size = vocab_size,
        embedding_dim=embedding_dim,
        rnn_units=rnn_units,
        batch_size=batch_size)
```

Using the *summary* function to display the model structure.Fig 9. The first layer is the embedding layer, which performs dropout after embedding to avoid overfitting; the next two lstm layers are the core part of the model, which are used to learn ”poetry creation”. Finally, the output is obtained through the fully connected layer.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (64, None, 256)           1564160
_________________________________________________________________
dropout (Dropout)            (64, None, 256)           0
_________________________________________________________________
lstm (LSTM)                  (64, None, 1024)          5246976
_________________________________________________________________
dropout_1 (Dropout)          (64, None, 1024)          0
_________________________________________________________________
lstm_1 (LSTM)                (64, None, 1024)          8392704
_________________________________________________________________
dropout_2 (Dropout)          (64, None, 1024)          0
_________________________________________________________________
dense (Dense)                (64, None, 6110)          6262750
=================================================================
Total params: 21,466,590
Trainable params: 21,466,590
Non-trainable params: 0
_________________________________________________________________
```

Figure 9. Model Architecture

After the model is built, we try to test it before training:

```
Input:
 '[秋至触物愁，况当离别筵。短歌销夜烛，繁绪遍高弦。桂水舟始泛，兰堂榻讵悬。一杯勾离阻，三载奉周旋。鸦噪更漏飒，露濡风景鲜。斯须不共此，且为更留连。][PAD][PAD][PAD][PAD][PAD][PAD]'

Predictions:
 '瓜宪悒鲍光雹酺鬏脑尤騑泳聪嚜志约鹑绞伏桔蛩混占咀兑卖丈薄谭脊墟鸡囚苊政吁裙枰三秀约箽緉柞啧鹀鞲稼鑫嶓虒抉扑屺淙酲掊查嫂迭绤押汪颀惚笙崾湮椽鄂跛头瞰滕漆郤插鼉縱偻'
```

Figure 10. Result

```
for input_example_batch, target_example_batch in
    train_ds.take(1):
    example_batch_predictions = model(
        input_example_batch)
    print(example_batch_predictions.shape, "
        respectively: batch_size, sequence_length
        , vocab_size")

sampled_indices = tf.random.categorical(
    example_batch_predictions[0], num_samples=1)
sampled_indices = tf.squeeze(sampled_indices,
    axis=-1).numpy()
print("Input: \n", repr("".join(int2char[
    input_example_batch[0]])))
print()
print("Predictions: \n", repr("".join(int2char[
    sampled_indices ])))
```

We can see what poetry an untrained model can generate in Fig.10.

The test loss and accuracy of the model can be computed as follow:

```
def loss(labels, logits):
    return tf.keras.losses.
        sparse_categorical_crossentropy(labels,
        logits, from_logits=True)
def accuracy(labels, logits):
    return tf.keras.metrics.
        sparse_categorical_accuracy(labels,
        logits)

example_batch_loss = loss(target_example_batch,
    example_batch_predictions)
example_batch_acc = accuracy(target_example_batch
    , example_batch_predictions)
print(" # (batch_size, sequence_length,
    vocab_size)")
print("Prediction shape: ",
    example_batch_predictions.shape)
print("Loss:", example_batch_loss.numpy().mean())
print("Acc:", example_batch_acc.numpy().mean())
```

```
 # (batch_size, sequence_length, vocab_size)
Prediction shape:  (64, 80, 6110)
Loss: 8.719297
Acc: 0.0
```

Figure 11. Pre-Test

As we can see, the model output is a bunch of gibberish, which means the model has no ability to generate poetry.

3.4 Model Training

Next, we define the optimizer, loss function, callback function and train:

```
# 编译优化器及损失函数
optimizer = tf.keras.optimizers.Adam()
model.compile(optimizer=optimizer, loss=loss)

# 定义 early stop 回调函数
patience = 6
early_stop = tf.keras.callbacks.EarlyStopping(
    monitor='val_loss', patience=patience)

# 定义 checkpoint 回调函数
checkpoint_prefix = './checkpoints/poetry_ckpt'
checkpoint_callback=tf.keras.callbacks.
    ModelCheckpoint(
   filepath=checkpoint_prefix,
   save_weights_only=True,
   save_best_only=True)

history = model.fit(train_ds, epochs=50,
    callbacks=[checkpoint_callback, early_stop] ,
     validation_data=valid_ds)
```

The training process is as follows:

```
Epoch 1/50
437/437 [==============================] - 38s 82ms/step - loss: 3.8401 - val_loss: 3.5046
Epoch 2/50
437/437 [==============================] - 36s 82ms/step - loss: 3.4104 - val_loss: 3.2624
Epoch 3/50
437/437 [==============================] - 36s 83ms/step - loss: 3.2034 - val_loss: 3.1327
Epoch 4/50
437/437 [==============================] - 36s 83ms/step - loss: 3.0880 - val_loss: 3.0452
Epoch 5/50
437/437 [==============================] - 37s 83ms/step - loss: 2.9933 - val_loss: 2.9711
Epoch 6/50
437/437 [==============================] - 37s 83ms/step - loss: 2.9104 - val_loss: 2.9127
Epoch 7/50
437/437 [==============================] - 37s 83ms/step - loss: 2.8357 - val_loss: 2.8598
Epoch 8/50
437/437 [==============================] - 37s 83ms/step - loss: 2.7666 - val_loss: 2.8247
Epoch 9/50
437/437 [==============================] - 37s 83ms/step - loss: 2.7041 - val_loss: 2.7905
Epoch 10/50
437/437 [==============================] - 37s 84ms/step - loss: 2.6447 - val_loss: 2.7594
Epoch 11/50
437/437 [==============================] - 37s 84ms/step - loss: 2.5863 - val_loss: 2.7389
Epoch 12/50
437/437 [==============================] - 37s 83ms/step - loss: 2.5311 - val_loss: 2.7231
Epoch 13/50
437/437 [==============================] - 37s 84ms/step - loss: 2.4806 - val_loss: 2.7119
Epoch 14/50
437/437 [==============================] - 37s 83ms/step - loss: 2.4333 - val_loss: 2.7034
Epoch 15/50
437/437 [==============================] - 37s 83ms/step - loss: 2.3875 - val_loss: 2.7002
Epoch 16/50
437/437 [==============================] - 37s 84ms/step - loss: 2.3457 - val_loss: 2.6954
Epoch 17/50
437/437 [==============================] - 37s 83ms/step - loss: 2.3053 - val_loss: 2.6952
Epoch 18/50
437/437 [==============================] - 36s 83ms/step - loss: 2.2666 - val_loss: 2.6970
Epoch 19/50
437/437 [==============================] - 37s 83ms/step - loss: 2.2302 - val_loss: 2.6994
Epoch 20/50
437/437 [==============================] - 37s 83ms/step - loss: 2.1959 - val_loss: 2.7051
Epoch 21/50
437/437 [==============================] - 37s 83ms/step - loss: 2.1636 - val_loss: 2.7093
Epoch 22/50
437/437 [==============================] - 37s 84ms/step - loss: 2.1329 - val_loss: 2.7136
Epoch 23/50
437/437 [==============================] - 37s 84ms/step - loss: 2.1025 - val_loss: 2.7216
```

Figure 12. Trainning process

To get a more intuitive feel for the training effect, we visualize the training loss and validation loss during the training process:

```
plt.figure(figsize=(12,9))
plt.plot(history.history['loss'], 'g')
plt.plot(history.history['val_loss'], 'rx')
plt.title('Model Loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train'], loc='upper right')
plt.legend(['Train', 'Validation'],
           loc='upper right')
plt.show()
```

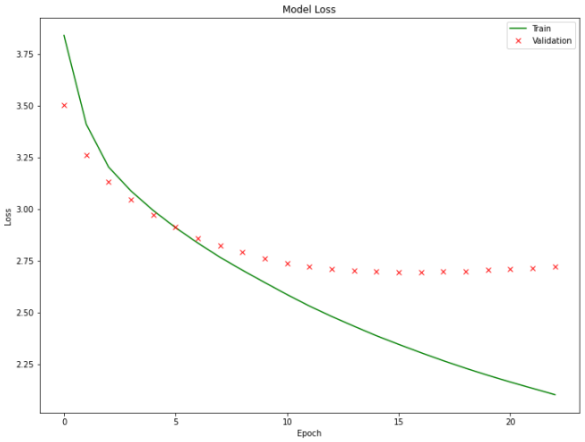


Figure 13. Training and Validation Loss

As the training progresses, both the training loss and validation loss gradually decrease. However, after reaching epoch 15, although the training loss continues to decrease, the validation loss has started to level off or even increase, indicating that the model has begun to overfit. Therefore, the model achieved the best performance for this training session at around epoch 15.

3.5 Model Evaluation

```
for input_example_batch, target_example_batch in
    train_ds.take(1):
   example_batch_predictions = model(
       input_example_batch)
sampled_indices = tf.random.categorical(
    example_batch_predictions[0], num_samples=1)
sampled_indices = tf.squeeze(sampled_indices, axis
    =-1).numpy()
print("Input: \n", repr("".join(int2char[
    input_example_batch[0]])))
print()
print("Predictions: \n", repr("".join(int2char[
    sampled_indices ])))
```

The results are as follows:

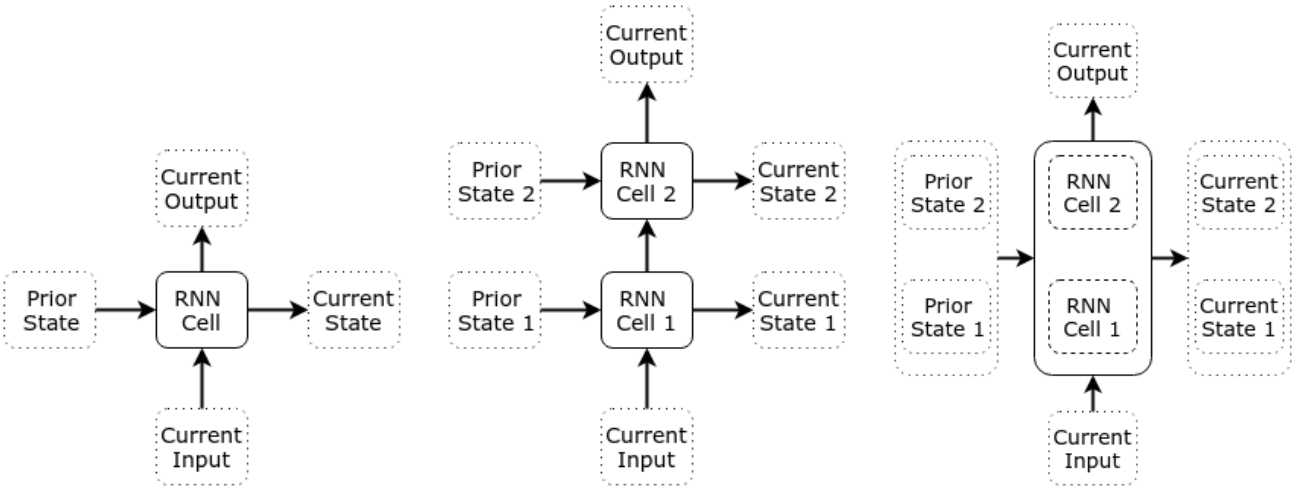
```
Input:
 '[仲宣楼上望重湖，君到潇湘得健无。病遇何人分药饵，
     诗逢谁子论功夫。杉萝寺里寻秋早，橘柚洲边度日
     晡。许送自身归华岳，待来朝暮拂瓶盂。][PAD][PAD
     ][PAD][PAD][PAD][PAD][PAD][PAD][PAD][PAD][
     PAD][PAD][PAD][PAD]'
Predictions:
 '三武仙下谢仙冈，日子扬湘信郡时。得起虽人无我饵，贫
     成求羡一吟文。常松乍带黄真色，杨浦洲阴背日晡。
     昨疾春师同礼岳，赤他闲市二群盂。][PAD][PAD][
     PAD][PAD][PAD][PAD][PAD][PAD][PAD][PAD][PAD
     ][PAD][PAD][PAD][PAD]'
```

Compared to the simple test, the model after training has clearly acquired the ability to generate poetry and can generate a poem according to the format. Although it can be found that the generated poetry has many parts that are not smooth and meaningful, which may be due to poor training results and the model being too simple.

3.6 Model Inference

We define infernce function as follow:

```
def generate_poetry(model, head_string):
    print("藏头诗生成中...., {}".format(
        head_string))
    poem = ""
    for head in head_string:
        if head not in char2int:
            print("抱歉，不能生成以{}开头的诗".
                format(head))
            return

        sentence = head
        max_sent_len = 20

        input_eval = [char2int[s] for s in '[' +
            head]
        input_eval = tf.expand_dims(input_eval, 0)

        model.reset_states()
        for i in range(max_sent_len):
            predictions = model(input_eval)
            predictions=tf.squeeze(predictions, 0)
            predicted_id = tf.random.categorical(
                predictions, num_samples=1)[-1,0].
                numpy()
            char_generated = int2char[predicted_id]
            if char_generated == '。':
                break
            input_eval = tf.expand_dims([
                predicted_id], 0)
            sentence += char_generated

        poem += '\n' + sentence

    return poem
```



(a) Single RNN Cell (b) Ungrouped RNN Cells (c) Grouped RNN Cells

Figure 14. Illustration for MultiRNNCell API

藏头诗生成中....，自然语言处理

自由
然涤青巾里，早留玄
语扫城门下玉步，不隔北流尘
言西
处为说上白毛年岁暮年年？，欲别心情展
理参频

Figure 15. Inference

It can be seen that although poems can be generated, these poems are not fluent, and they do not have any meaning. This reflects that our model still needs improvements:

4. Open-Ended Question

The official API for *tf.contrib.rnn.MultiRNNCell* can be found at MultiRNNCell API.

To see why this works, consider that while a single cell as in Fig. 14(a). Then Fig. 14(b) is two cells stacked on top of each other. Then we can use the *MultiRNNCell* to wrap the two cells into a single two-layer to make them look and behave as a single cell in Fig. 14(c).

This class is an RNN cell composed of multiple simple RNN cells in sequence. It inherits from the **RNNCell** class and implements the **state_size** and **output_size** attributes as well as the **zero_state** and **call** methods. This is a example of how to use it:

```
num_units = [128, 64]
cells = [BasicLSTMCell(num_units=n)
       for n in num_units]
stacked_rnn_cell = MultiRNNCell(cells)
```

The **___init___** method of this class takes a list of cells **cells** and a boolean value **state_is_tuple** indicating whether the state should be returned as a tuple. If **state_is_tuple** is **False**, all states are concatenated along the column axis. The method also checks whether the cell list is empty, whether it is a sequence, and whether there are cells that return a state tuple but **state_is_tuple** is **False**.

The **state_size** attribute returns the size of the state. If **state_is_tuple** is **True**, it returns a tuple containing the state sizes of each cell. Otherwise, it returns the sum of all cell state sizes.

The **output_size** attribute returns the output size of the last cell.

The **zero_state** method returns an initial state, which is calculated by the **zero_state** method for all cells. If **state_is_tuple** is **True**, it returns a tuple containing the initial state of each cell. Otherwise, it returns the concatenation of all initial states.

The **trainable_weights**, **non_trainable_weights** return lists of trainable and non-trainable weights, which are composed of their attributes of each cell.

The **call** method performs forward propagation of the RNN cell. This method takes input **inputs** and state **state** and returns output and new state. The method first passes the input to the first cell, then passes the output to the next cell until all cells are processed. In each cell, the current state is obtained from **state**, and the new state is added to the **new_states** list. Finally, the new state is returned as a tuple if **state_is_tuple** is **True**, or concatenates all new states and returns them.

5. Conclusion

In this experiment, we have learned how to use LSTM to generate acrostic poems. But this model is still far from perfect, we will dive deeper into this problem in our final report.