```python
##### DESCRIPTION #####
# Use to recall a previously trained model to keep generating more text if desired.

import string
import torch
import torch.nn as nn
import sys
import os

DATASET = 'Complete Sherlock Holmes'

##### HYPERPARAMETERS #####
# These must be the same as a previously trained model present in the files
CELL_TYPE = 'RNN'
OPTIM_TYPE = 'Adam'
HIDDEN_LAYERS = 1
HIDDEN_SIZE = 100

##### GENERATION PARAMETERS #####
# Alter the length of the new generated sequence and it's initial sequence
PREDICTION_LENGTH = 100
INITIAL_SEQUENCE = '\n'

print('Recalling model',CELL_TYPE,OPTIM_TYPE,HIDDEN_LAYERS,HIDDEN_SIZE)
print('Generating {} char, with init_seq: {}'.format(PREDICTION_LENGTH, INITIAL_SEQUENCE))

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
print('Device =', device)

all_chars = string.printable
n_chars = len(all_chars)

def seq_to_onehot(seq):
    tensor = torch.zeros(len(seq), 1, n_chars)
    for t, char in enumerate(seq):
        index = all_chars.index(char)
        tensor[t][0][index] = 1
    return tensor

class Net(nn.Module):
    def __init__(self):
        # Initialization.
        super(Net, self).__init__()
        self.input_size  = n_chars # Input size: Number of unique chars.
        self.hidden_size = HIDDEN_SIZE
        self.output_size = n_chars # Output size: Number of unique chars.

        # Ensures the size of the hidden layer stack does not exceed 3
        self.layers = HIDDEN_LAYERS
        if HIDDEN_LAYERS>3:
            self.layers=3

        # Create a rnn cell for the stack
        def create_cell(size_in, size_out):
            if CELL_TYPE=='LSTM':
                return nn.LSTMCell(size_in, size_out)
```

```python
            elif CELL_TYPE=='GRU':
                return nn.GRUCell(size_in, size_out)
            elif CELL_TYPE=='RELU': # Not used in testing report
                return nn.RNNCell(size_in, size_out, nonlinearity='relu')
            else:
                return nn.RNNCell(size_in, size_out)

        self.rnn = create_cell(self.input_size, self.hidden_size)

        if HIDDEN_LAYERS>=2:
            self.rnn2 = create_cell(self.hidden_size, self.hidden_size)
        if HIDDEN_LAYERS>=3:
            self.rnn3 = create_cell(self.hidden_size, self.hidden_size)

        self.linear = nn.Linear(self.hidden_size, self.output_size)

    def forward(self, input, hidden, cell, hidden2=False, cell2=False, hidden3=False, cell3=False):
        # Forward function.
        #   takes in the 'input' and 'hidden' tensors,
        #   can also take in 'cell state' tensor if cell type is 'LSTM',
        #   takes additional hidden and cell state tensors for each layer
        if CELL_TYPE=='LSTM':
            hidden, cell = self.rnn(input, (hidden,cell))
            if self.layers>=2:
                hidden2, cell2 = self.rnn2(hidden, (hidden2,cell2))
            if self.layers>=3:
                hidden3, cell3 = self.rnn3(hidden2, (hidden3,cell3))
        else:
            hidden = self.rnn(input, hidden)
            if self.layers>=2:
                hidden2 = self.rnn2(hidden, hidden2)
            if self.layers>=3:
                hidden3 = self.rnn3(hidden2, hidden3)

        # Linear transformation (fully connected layer) to the output
        if self.layers==3:
            output = self.linear(hidden3)
            return output, hidden, cell, hidden2, cell2, hidden3, cell3
        elif self.layers==2:
            output = self.linear(hidden2)
            return output, hidden, cell, hidden2, cell2
        else:
            output = self.linear(hidden)
            return output, hidden, cell
    def init_hidden(self):
        # Initial hidden state.
        return torch.zeros(1, self.hidden_size).to(device)
    def init_cell(self):
        # Initial cell state.
        return torch.zeros(1, self.hidden_size).to(device)

def eval_step(net, init_seq='\n', predicted_len=100):
    # Initialize the hidden state, input and the predicted sequence
    hidden        = net.init_hidden()
    cell          = net.init_cell()
    if HIDDEN_LAYERS >=2:
```

```
        hidden2 = net.init_hidden()
        cell2 = net.init_cell()
    if HIDDEN_LAYERS >=3:
        hidden3 = net.init_hidden()
        cell3 = net.init_cell()
    init_input    = seq_to_onehot(init_seq).to(device)
    predicted_seq = init_seq

    # Use initial string to "build up" hidden state.
    for t in range(len(init_seq) - 1):
        if HIDDEN_LAYERS==3:
            output, hidden, cell, hidden2, cell2, hidden3, cell3 = net(init_input[t], hidde
cell, hidden2, cell2, hidden3, cell3)
        elif HIDDEN_LAYERS==2:
            output, hidden, cell, hidden2, cell2 = net(init_input[t], hidden, cell, hidden2
cell2)
        else:
            output, hidden, cell = net(init_input[t], hidden, cell)
    # Set current input as the last character of the initial string.
    input = init_input[-1]

    # Predict more characters after the initial string.
    for t in range(predicted_len):
        # Get the current output and hidden state.
        if HIDDEN_LAYERS==3:
            output, hidden, cell, hidden2, cell2, hidden3, cell3 = net(input, hidden, cell,
hidden2, cell2, hidden3, cell3)
        elif HIDDEN_LAYERS==2:
            output, hidden, cell, hidden2, cell2 = net(input, hidden, cell, hidden2, cell2)
        else:
            output, hidden, cell = net(input, hidden, cell)

        # Sample from the output as a multinomial distribution.
        try:
            predicted_index = torch.multinomial(output.view(-1).exp(), 1)[0]
        except: # Added post to resolve errors with tensors containing 'inf'/'nan' values
            predicted_index = torch.multinomial(output.view(-1).exp().clamp(0.0,3.4e38), 1)
[0]
        # Add predicted character to the sequence and use it as next input.
        predicted_char  = all_chars[predicted_index]
        predicted_seq  += predicted_char
        # Use the predicted character to generate the input of next round.
        input = seq_to_onehot(predicted_char)[0].to(device)

    return predicted_seq

PATH = 'Results'
for folder in [DATASET, CELL_TYPE, OPTIM_TYPE, HIDDEN_LAYERS, HIDDEN_SIZE]:
    folder=str(folder)
    PATH+='/'+folder
PATH+='/model.pt'

net=Net()
net.to(device)

net.load_state_dict(torch.load(PATH))
net.eval()
```

```

print(eval_step(net, init_seq=INITIAL_SEQUENCE, predicted_len=PREDICTION_LENGTH))
```