```python
##### DESCRIPTION #####
# Use to train a model on the specified dataset and hyperparameters.

import string
import random
import torch
import torch.nn as nn
import matplotlib.pyplot as plt
import numpy as np
import time
import sys
import os
from tqdm import tqdm

FILE_NAME = 'complete_sherlock_holmes.txt'
DATASET = 'Complete Sherlock Holmes'

##### HYPERPARAMETERS #####
CELL_TYPE = 'RNN' # DEFAULTS TO 'RNN'. Options: ['RNN', 'LSTM', 'GRU', 'RELU']
OPTIM_TYPE = 'Adam' # DEFAULTS TO 'Adam'. Options: ['Adam', 'ASGD', 'Adagrad','RMSprop']
HIDDEN_LAYERS = 1 # DEFAULT: 1
HIDDEN_SIZE = 100 # DEFAULT: 100

LEARNING_RATE = 0.005 # 0.005 for Adam, 0.05 for other optimizers
INPUT_SEQUENCE = 128 # DEFAULT: 128

TRAINING_ITERATIONS = 20000 # DEFAULT: 20000
INITIAL_SEQUENCE = '\n' # To use for eval

print('Running on',CELL_TYPE,OPTIM_TYPE,HIDDEN_LAYERS,HIDDEN_SIZE)

all_chars = string.printable
n_chars = len(all_chars)
file = open('./'+FILE_NAME).read()
file_len = len(file)

def get_random_seq():
    seq_len     = INPUT_SEQUENCE  # The length of an input sequence.
    start_index = random.randint(0, file_len - seq_len)
    end_index   = start_index + seq_len + 1
    return file[start_index:end_index]
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
def seq_to_index(seq):
    tensor = torch.zeros(len(seq), 1)
    for t, char in enumerate(seq):
        tensor[t] = all_chars.index(char)
    return tensor
def get_input_and_target():
    seq    = get_random_seq()
    input  = seq_to_onehot(seq[:-1]) # Input is represented in one-hot.
    target = seq_to_index(seq[1:]).long() # Target is represented in index.
```

```
    return input, target

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

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)
            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.fc = 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)

```

```python
        # Linear transformation (fully connected layer) to the output
        if self.layers==3:
            output = self.fc(hidden3)
            return output, hidden, cell, hidden2, cell2, hidden3, cell3
        elif self.layers==2:
            output = self.fc(hidden2)
            return output, hidden, cell, hidden2, cell2
        else:
            output = self.fc(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)

net = Net()
net.to(device)

# Training step function
def train_step(net, opt, input, target):
    seq_len = input.shape[0]
    hidden = net.init_hidden() # Initial hidden state
    cell = net.init_cell() # Initial cell state
    if HIDDEN_LAYERS >=2:
        hidden2 = net.init_hidden()
        cell2 = net.init_cell()
    if HIDDEN_LAYERS >=3:
        hidden3 = net.init_hidden()
        cell3 = net.init_cell()

    net.zero_grad()
    loss = 0 # Initial loss.

    for t in range(seq_len): # For each one in the input sequence
        if HIDDEN_LAYERS==3:
            output, hidden, cell, hidden2, cell2, hidden3, cell3 = net(input[t], hidden, cell, hidden2, cell2, hidden3, cell3)
        elif HIDDEN_LAYERS==2:
            output, hidden, cell, hidden2, cell2 = net(input[t], hidden, cell, hidden2, cell2)
        else:
            output, hidden, cell = net(input[t], hidden, cell)
        loss += loss_func(output, target[t])

    loss.backward() # Backward.
    opt.step() # Update the weights.

    return loss / seq_len

# Evaluation step function
def eval_step(net, init_seq=INITIAL_SEQUENCE, 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


##### MAIN ALGORITHM #####

iters = TRAINING_ITERATIONS # Number of training iterations.

# The loss variables.
all_losses = []
# Initialize the optimizer and the loss function.
if(OPTIM_TYPE=='ASGD'):
    opt=torch.optim.ASGD(net.parameters(), lr=LEARNING_RATE)
if(OPTIM_TYPE=='Adagrad'): # Not used in testing results
```

```python
    opt=torch.optim.Adagrad(net.parameters(), lr=LEARNING_RATE)
if(OPTIM_TYPE=='RMSprop'): # Not used in testing results
    opt=torch.optim.RMSprop(net.parameters(), lr=LEARNING_RATE)
else:
    opt = torch.optim.Adam(net.parameters(), lr=LEARNING_RATE)
loss_func = nn.CrossEntropyLoss()

# Training procedure.
start_time = time.time()
for i in tqdm(range(iters)):
    input, target = get_input_and_target() # Fetch input and target.
    input, target = input.to(device), target.to(device) # Move to GPU memory.
    loss = train_step(net, opt, input, target)   # Calculate the loss.
    all_losses.append(loss)

end_time = time.time()
total_time = end_time - start_time

# Calculates summary of losses
i_half = int(len(all_losses)*0.5)
i_quart = int(len(all_losses)*0.75)
loss_avg = np.sum(np.array(all_losses))/len(all_losses)
loss_avg_half = np.sum(np.array(all_losses[i_half:]))/len(all_losses[i_half:])
loss_avg_quart = np.sum(np.array(all_losses[i_quart:]))/len(all_losses[i_quart:])
loss_list=[elem.item() for elem in [loss_avg,loss_avg_half,loss_avg_quart]]
rolling_losses=[]
losses_copy = [i.item() for i in all_losses]
for i in range(len(losses_copy)):
    temp=losses_copy[np.max((i-100, 0)):i+1]
    rolling_losses.append(np.sum(temp)/len(temp))

plt.xlabel('iters')
plt.ylabel('loss')
plt.hlines(loss_list,0,len(losses_copy)-1,['red','orange','green'],'dashed')
plt.plot(rolling_losses)
plt.ylim(0,5)

print('Avg loss: {}'.format(loss_avg))
print('Avg loss last half: {}'.format(loss_avg_half))
print('Avg loss last quarter: {}'.format(loss_avg_quart))
print()
print('Training time: {} sec'.format(total_time))
print('{} min | {:.3f} hr'.format(total_time/60, total_time/3600))

# Creates a folder path to save training results to
PATH = 'Results'
for folder in [DATASET, CELL_TYPE, OPTIM_TYPE, HIDDEN_LAYERS, HIDDEN_SIZE]:
    folder=str(folder)
    if not os.path.isdir(PATH+'/'+folder):
        os.mkdir(PATH+'/'+folder)
    PATH+='/'+folder
PATH+='/'

print()
print('Results saved to: {}'.format(PATH))
model_path = PATH+'model.pt' # Saves model parameters

```

```python
torch.save(net.state_dict(), model_path)

# Sequence of all 20000 loss values
file = open(PATH+'all_losses.txt','w')
file.write(' '.join([str(elem) for elem in losses_copy]))
file.close()

# A 5000 char sample generated after training
file = open(PATH+'sample.txt','w')
file.write(eval_step(net, predicted_len=5000))
file.close()

# Information on training
file = open(PATH+'info.txt','w')
file.write('Iterations: {}\n\n'.format(TRAINING_ITERATIONS))
file.write('Dataset: {}\nInput Size: {}\nLearning Rate: {}\n\n'.format(DATASET, INPUT_SEQUENCE, LEARNING_RATE))
file.write('Cell Type: {}\nOptimizer: {}\nHidden Layers: {}\nHidden Size: {}\n\n'.format(CELL_TYPE, OPTIM_TYPE, HIDDEN_LAYERS, HIDDEN_SIZE))
```